HSR Hochschule für Technik Rapperswil

MRU: Sensors, Actors and Communication

Studienarbeit

Yolo auf Finger

im Studiengang Industrial Technologies

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

eingereicht am: 9. Februar 2018

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

Abstract

Aufgabenstellung

Das Ziel dieser Projektarbeit bestand darin, herauszufinden, ob Yolo geeignet wäre, die Fingerspitzen einer Hand in einem Bild zu klassifizieren und genau zu detektieren. Die Genauigkeit sollte bei maximal 0.1mm liegen. Yolo ist eine Möglichkeit, 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 um eine Echtzeitanwendung handeln soll.

Vorgehen

Mithilfe der Apparatur und Software von Tabea Méndez [2] wurden Daten generiert. Um diesen Aufwand klein zu halten, wurden nur Daten vom rechten Zeigefinger generiert. Gleichzeitig wurde in Tensorflow die Architektur von Yolo nachgebaut. Dies wäre nur begrenzt nötig gewesen, da fertige Architekturen in Keras oder Darknet online zur Verfügung stehen. 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 wurden 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 “echte“ Training auf den selber generierten Daten. Sobald dies alles aufgebaut und lauffähig war, wurde noch so viel wie möglich experimentiert und gleichzeitig letzte Fehler behoben. Es sollte herausgefunden werden, mit welchen Änderungen und Einstellungen das Lernresultat noch optimiert werden könnte.

Fazit

Das Pretraining und auch das Training hatten seine Tücken. Das originale Yolo-Netzwerk war extrem gross und brauchte entsprechend nahezu den ganzen RAM-Speicher einer GPU. Deswegen blieb nur noch begrenzt Platz für Daten übrig. Diese Probleme konnten einigermassen umgangen werden, hat-

ten 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 der 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 “gut“ aussehen. Dies ist ein einigermassen erstaunliches Resultat, wenn man bedenkt, dass zum Trainieren nur rund 18’000 Bilder verwendet wurden. Es ist anzunehmen, dass in naher Zukunft eine Genauigkeit von bis zu 1mm erreicht werden könnte. Dies sollte mit einer Verbesserung der Datengewinnung, entsprechend viel mehr Daten und mit einem verbesserten Konzept von Yolo möglich sein.

Inhaltsverzeichnis

1 Einleitung **6**
1.1 Ausgangslage 6
1.2 Ziel 6
1.3 Hauptquellen 6
1.4 Vorgehen 7
1.5 Aufbau 7

2 Daten-Pipeline **9**
2.1 Bilder aufnehmen 9
2.2 Fingerdetektion 10
2.3 CSV generieren 10
2.4 Python-Objekt generieren 11
2.4.1 Label-Tensor 12
2.4.2 Liste von Label-Tensoren 13
2.5 Daten in Neuronales Netzwerk einlesen 14

3 Architektur **16**
3.1 Auswahl der Architektur 16
3.2 Architekturaufbau 16
3.2.1 Hauptgraph 16
3.2.2 Minibatchnorm 19
3.2.3 Aktivierungsfunktion 20
3.2.4 Dropout 20
3.3 Fazit 21

4 Kostenfunktion **22**
4.1 Erster Wurf 22
4.2 Netzwerkoutput 22
4.3 Design Kostenfunktion 24
4.4 Fazit 26

5 Tests **28**
5.1 Erste Ansätze 28
5.2 Letztendlicher Test 29
5.3 Seed 30
5.4 Fazit 30

6 Resultate **31**
6.1 Testvoraussetzungen 31
6.2 Analyse 31
6.2.1 Distanz 31
6.2.2 Intersection Over Union IOU 35

7 Pretraining **38**
7.1 Daten 38
7.2 Architektur 38
7.3 Kostenfunktion und Optimierer 38
7.4 Tests & Resultate 40
7.5 Gewichte 41
7.6 Fazit 42

8 Fazit **43**
8.1 Gipshände 43
8.2 Yolo 43
8.3 Vorschläge für weitere Schritte 44

Literatur **45**

1 Einleitung

1.1 Ausgangslage

An der Hochschule für Technik Rapperswil wurde ein Flugsimulator gebaut, welcher mithilfe eines durch Motoren bewegbaren Sitzes und einer Virtual-Reality-Brille ein extrem echtes Fluggefühl vermittelt.

Um den Simulator möglichst echt und kommerziell nutzbar zu machen, muss die Person, welche die Virtual-Reality-Brille aufgesetzt hat die Knöpfe im Cockpit bedienen können. Dies geht selbstverständlich nur, wenn diese Person auch ihre eigenen Finger sieht. Momentan ist dies nicht der Fall. Und an diesem Punkt kommt "Yolo auf Finger" ins Spiel.

Das Ziel war es mithilfe von vier Kameras die Finger des Piloten in Echtzeit und im 3D-Raum zu tracken und anschliessend animiert in der Virtual-Reality-Brille wieder darzustellen.

Einige erste Schritte in diese Richtung wurden bereits mit der Arbeit "Fingerspitzen-Tracking im 3D-Raum" [2] erbracht. So wurde ein System entwickelt, welches mit vier Kameras Punkte von den 2D-Bilder in den 3D-Raum mappen kann. Ausserdem wurde im selben Rahmen ein Testaufbau gemacht, welcher es erlaubt mithilfe von UV Licht Label-Daten aufzunehmen.

1.2 Ziel

Es soll Folgendes herausgefunden werden. Sind Algorithmen, welche auf dem Konzept von Yolo [3] aufbauen geeignet um im 2D-Raum die Positon von Fingerspitzen auf 0.1mm genau zu finden?

1.3 Hauptquellen

Diese Studienarbeit wurde vor allem auf der Arbeit "You Only Look Once" [3] aufgebaut. Darin wird die Architektur eines Neuronalen Netzwerks beschrieben, welches mit nahezu unglaublicher Performance und einer akzeptablen Genauigkeit Gegenstände und Objekte in 2D-Bildern erkennt und deren Standort im Bild detektiert. Diese Arbeit ist zusammen mit Ihrer Nachfolgearbeit "Yolo9000" der aktuelle State of the Art was Klassifizierung kombiniert mit Detektion angeht.

Die Arbeit "Fingerspitzen-Tracking im 3D-Raum" von Tabea Méndez [2] war gerade wegen der gebauten Vorrichtung zur Datengeneration der Ausgangs-

punkt für diese Arbeit.

Teilweise als theoretische Grundlage diente das Buch Deeplearning [1], welches im Inhalt ebenfalls dem aktuellen State of the Art entspricht.

Die wahrscheinlich wertvollste Quelle dieser Arbeit waren die Diskussionen und Anregungen von und mit Prof. Dr. Guido Schuster, Tabea Méndez, Hannes Badertscher und Jonas Schmid [6]. An dieser Stelle ein herzliches Dankeschön an die betroffenen Personen, welche geduldig und kompetent für jegliche Fragen zur Verfügung standen.

1.4 Vorgehen

In einem ersten Schritt wurde mit Hilfe des Testaufbaus [2] genügend Daten aufgenommen, damit ein Neuronales Netzwerk darauf trainiert werden kann.

Als nächstes wurde Programmierung von Tensorflow in Python erarbeitet und gelernt.

Danach wurde eine erste, möglichst einfache Version von Yolo v1 [3] implementiert.

Sobald diese Version lauffähig war, wurde diese optimiert, sodass damit ein möglichst gutes Resultat erzeugt werden konnte.

Als letztes sollte das Netzwerk noch auf Yolo v2 [4] erweitert werden. Dazu kam es aber aus Gründen der Zeit und der Planung nicht mehr.

1.5 Aufbau

In den Kapiteln zwei bis fünf werden die einzelnen Punkte im Aufbau des Netzwerks vom Erzeugen der Daten bis zum Test der Resultate genau ausgeleuchtet. In diesen Kapiteln wird auch beschrieben, welche Hürden oder Schwierigkeiten sich zeigten und wie diese falls möglich überwunden wurden. Im Kapitel sechs, werden die erreichten Resultate dieser Arbeit beschrieben. In diesem Kapitel wird ersichtlich, wo diese Technologie aktuell steht, und was damit alles möglich ist.

Bevor im letzten Kapitel ein Fazit zur gesamten Arbeit gezogen wird, wird im Kapitel sieben genauer auf die Art und Weise eingegangen, wie das Netzwerk vor-trainiert (Pretraining) wurde. Dieser Punkt erhält ein eigenes Kapitel, weil diese Arbeit “parallel“ zum eigentlichen Training aufgebaut und ausgeführt wurde.

Das Ziel dieser Dokumentation ist es schnell einen Überblick über Gelern-

tes und die aufgetretenen Probleme erhalten. Dies soll künftigen Studenten/Assistenten helfen, welche sich mit dem Thema auseinandersetzen. So sollten allfällige Fehler nicht zweimal gemacht werden.

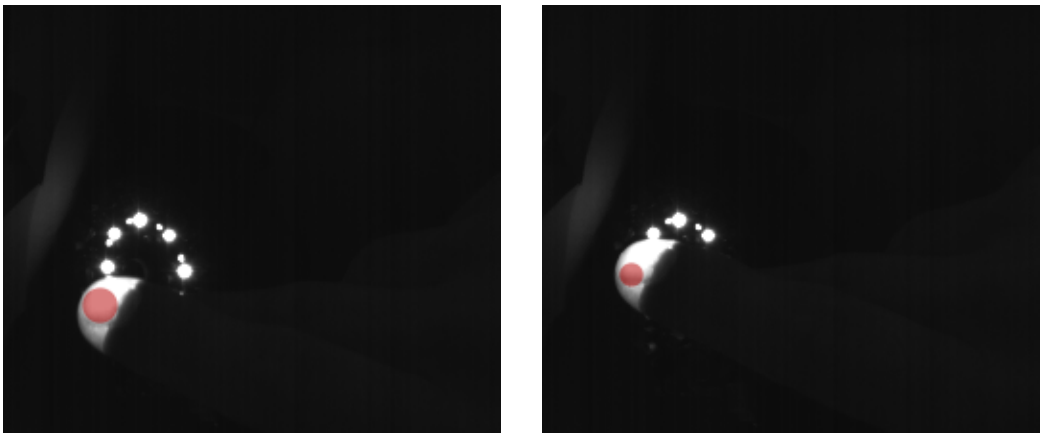

Abbildung 1: Resultate der Erosion

2 Daten-Pipeline

2.1 Bilder aufnehmen

Die Aufnahme der Bilder geschah unverändert mit der Apparatur und dem C++ Code von Tabea Méndez welche aus Ihrer Masterarbeit [2] entstanden. Das Ergebnis bestand jeweils aus acht Bildern von einer Situation. Eine Situation bestand aus vier Kameras. Dabei machte jede Kamera jeweils ein schwarzweiss-Bild mit UV-Beleuchtung und ein schwarzweiss-Bild mit normaler weisser Beleuchtung. Als die Daten gelablet wurden, wurden noch keine Zeitinformation verwendet. Das heisst die Finger wurden noch nicht von Bild zu Bild getrackt. So wurden helle Punkte im Hintergrund oft als Finger erkannt. Um dies zu umgehen wurde der Aufbau im Hintergrund mit schwarzem Papier abgedeckt.

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 zwei verschiedene Threads aufteilen würde. Dabei wäre ein Thread für das Aufnehmen der Daten und der andere für das Abspeichern zuständig. So könnten “zeitlich unbegrenzt“ Daten aufgenommen werden. Dies würde aber nur nötig, falls in Zukunft ein Roboter verwendet wird, um Daten aufzunehmen.

2.2 Fingerdetektion

Zur Fingerdetektion wurde der Matlab-Fingerdetektor aus der Masterarbeit von Tabea Méndez [2] verwendet. Um Zeit bei der Datenaufnahme einzusparen wurden Zeit- und Rauminformationen 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 weitgehend behoben werden, indem in den Matlab-Fingerdetektor noch einige Filter eingebaut wurden. Diese Filter hatten folgende Funktionen.

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

2. Aussortieren von erkannten Punkten, welche zu gross waren, als dass Sie eine Fingerspitze sein könnten. Dieser Punkt ist teilweise redundant mit dem ersten Punkt, da grosse Punkte im Hintergrund oft nur bei einer extrem grossen Helligkeit auftreten können.

3. Aussortieren von erkannten Punkten, welche zu klein waren, als dass Sie eine Fingerspitze sein konnten. Damit wurden die meisten LED-Punkte entfernt.

4. Von den übrigen Punkten wird dann nur noch der Grösste behalten. Diesem wurde 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. Es lag daran, dass die Detektionspunkte nicht immer genau in der Mitte des Fingers lagen. Weiter waren auch die Boundingboxen, welche sich aus dem Resultat der Erosion [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. Yolo 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 abspeicherte. In diesem

Skript wurden ausserdem die Daten in Test und in Trainingsdaten aufgeteilt und je in 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 Erosion
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 eine kleine Python-Klasse geschrieben. Die Daten mussten allerdings vor der 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 hiessen bei verschiedenen Kameras genau gleich. Mit der Vorverarbeitung wurden alle Bilder an einem gemeinsamen Ort gespeichert. Ausserdem erhielt jedes Bild einen neuen Namen / eine neue Nummerierung, wodurch es möglich wurde sie eindeutig 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.

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

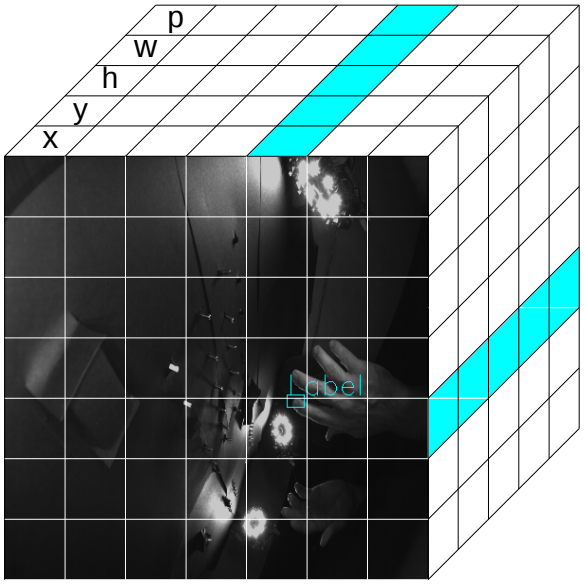

Abbildung 2: Label-Tensor

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, 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.

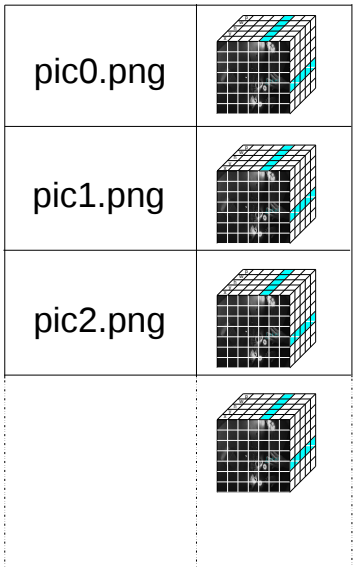

Abbildung 3: Liste von Label-Tensoren

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 Funktion übernimmt das organisieren des Minibatches. So kann die Grösse des Minibatches flexibel bestimmt werden.

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. der Kostenfunktion und 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.

Das Vorgehen (Kapitel 1.4) aber wurde im Nachhinein betrachtet falsch angeordnet. So ist das experimentieren/optimieren eines Neuronalen Netzwerks etwas extrem Zeitaufwendiges, das man nie wirklich abschliessen kann. So gesehen hätte das Vorgehen folgendermassen geplant werden müssen. Sobald eine lauffähige fehlerfreie Version von Yolo v1 erreicht wurde, hätte sofort mit dem Architekturaufbau von Yolo v2 begonnen werden müssen. Es wird davon ausgegangen, dass so noch bessere Resultate hätten erzielt werden können.

3.2 Architekturaufbau

3.2.1 Hauptgraph

Der grundlegende Aufbau der Architektur, wie sie letztendlich aussah kann man in der Tabelle 1 betrachten. Dies sah allerdings noch nicht immer so aus. Obwohl die Convolution-Filter schon immer in dieser Form angeordnet waren, sah der ursprüngliche Bildinput und entsprechend die Outputs der verschiedenen Layers mal anders aus. Nach ausführlicher Diskussion [6] 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 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 war dann auch der Output von Layer 1 nicht mehr 224x224x64

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

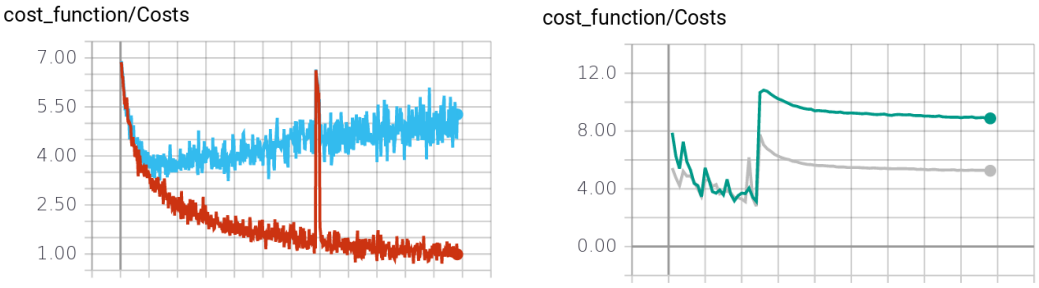

Abbildung 4: Effekte (Extremer Anstieg der Kosten innert einer Epoche) im Pretraining(links) und im Training(rechts). x-Achse=Zeit, y-Achse=Kosten. Weinrot=Pretraining-Trainingsdaten, Hellblau=Pretraining-Validierungsdaten, Grün=Training-Trainingsdaten, Grau=Training-Validierungsdaten

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 (Siehe Abbildung 4) 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 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 im Gegenzug 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 (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 wer-

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

den 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, Batchnorm, etc. 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.

3.2.2 Minibatchnorm

Zu Beginn des Trainings gab es immer wieder das Problem von vanishing Gradients (Die Gradienten wurden im Verlauf der Backpropagation immer kleiner, bis sie nur noch gleich Null waren.). Um diesem Problem entgegenzuwirken wurde die relativ junge Allzweckwaffe der Minibatch-Normalisierung angewandt. Dabei wird in jedem Convolutional Layer direkt nach dem Convolutional-Filter der Ausgang über den ganzen Minibatch normiert. Es wurde auch ausprobiert die Minibatchnorm nach der Relu einzusetzen, anstelle von davor, allerdings waren die Ergebnisse der Kostenfunktion bei der Anordnung in Abbildung 5 ganz leicht besser. Die Verbesserung war allerdings nur sehr marginal, weshalb sie auch gerade so gut zufällig sein konnte. Seit dem Einsetzen der Minibatchnorm war das Netzwerk hervorragend in der Lage die Gradienten per Backpropagation bis zum Eingang des Netzwerks zurück zu tragen.

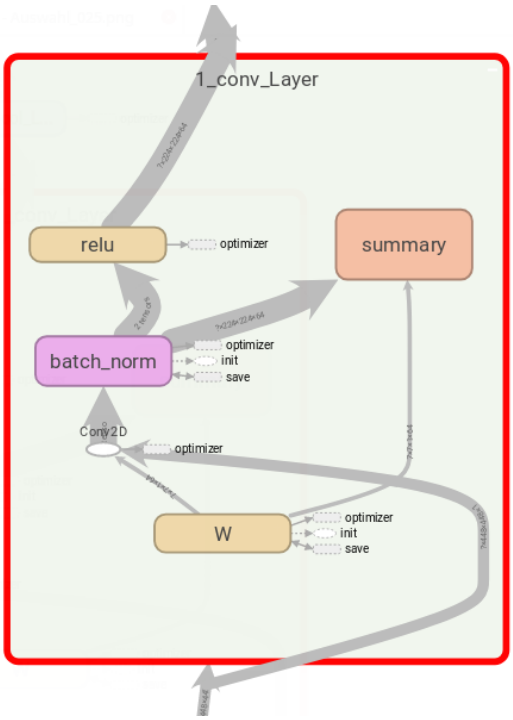

Abbildung 5: Aufbau eines Convolutional-Layers

3.2.3 Aktivierungsfunktion

Wie in Abbildung 5 ersichtlich wurde als Aktivierungsfunktion eine simple Relu (Rectified Linear Unit) verwendet. Dies obwohl in yolo v1 eine leaky Relu verwendet wurde. Der Grund für diesen Wandel war, dass seit dem verwenden der Minibatchnorm eine leaky Relu dieselbe Performance an den Tag legte wie eine normale Relu.

3.2.4 Dropout

Es wurde im Rahmen dieser Arbeit zur Optimierung viel mit Dropout Experimentiert. So wurde Dropout in allen, nur in den letzten, nur in den fully-connected Layer oder auch in gar keinem Layer getestet. Die besten Performance wurden erreicht, wenn entweder nur in den letzten paar Layern oder auch nur in den fully-connected Layern Dropout eingesetzt wurde. Wurde Dropout in allen Layern angewandt, war das Resultat massiv schlechter.

Wichtig zu wissen: Wenn man Dropout in Convolutional Layern anwendet, sollte darauf geachtet werden, dass man nicht einzelne Knoten von Dropout

“ausknipsen“ lässt, sondern gleich ganze Featuremaps. Der Grund dafür liegt darin, dass man mit der ursprünglichen Philosophie von Dropout eigentlich Gewichte zwischenzeitlich ausschalten möchte. Bei Fullyconnected Netzwerken ist dies problemlos möglich, indem man einfach Knoten zufällig auf Null setzt, womit automatisch die zugehörigen Gewichte auch ausgeschaltet wurden. In Convolutional Neural Networks hingegen werden die Gewichte nicht ausgeschaltet, indem man einen Knoten ausschaltet. Dies, weil die Gewichte über die Bilder oder Feature-Maps geschoben werden und nicht einzelnen Knoten zugeteilt sind.

3.3 Fazit

Was die Architektur angeht kann man aus dieser Arbeit die folgenden 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 einer bestimmten 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.

3. Es trat zu Beginn des Trainings öfters das Problem auf, dass das Programm abstürzte. Dabei war die Fehlermeldung jeweils, dass “NAN“ (Not a Number) oder “Infinity“ ins Tensorboard gespeichert werden sollte. Seit die Gradienten auf die Zahl 5/-5 begrenzt wurden, trat dieses Problem auch nie mehr auf. Daraus folgt: Gradient Clipping ist nur zu empfehlen, da es nicht schadet aber auf jeden Fall allfällige Probleme fernhalten 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 könnte. 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 später 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 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 Yolo v1 Kostenfunktion zu orientieren.

4.2 Netzwerkoutput

Im ersten Schritt wurde der Output des Netzwerks fast genau wie das Vorbild aus dem Yolo-Paper [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 6) 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 Output-Gitterzelle auch eine Vorhersage zur Confidence enthält.

Die Confidence war deshalb nicht explizit in den Labels enthalten, weil sie erst mit den Vorhersagen zu x, y, w und h sowie den entsprechenden Labels berechnet werden konnte. Die Label-Confidence ist eigentlich nichts ande-

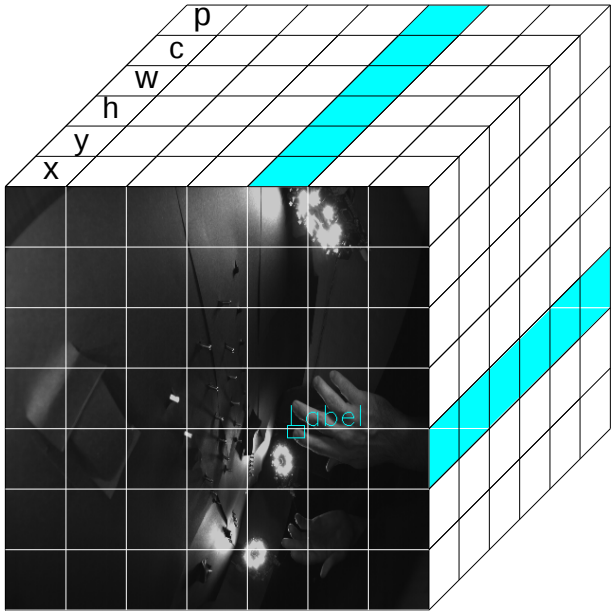

Abbildung 6: Output-Tensor

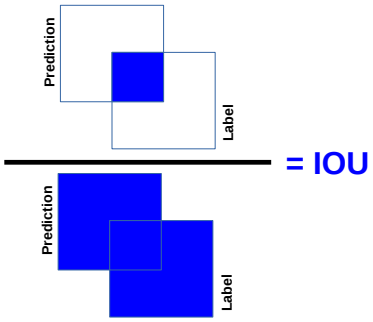

Abbildung 7: Berechnung der IOU

res als die 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 7 ersichtlich.

Es befinden sich allerdings in den meisten Gitterzellen keine rechten Zeigefingerspitzen. 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). 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 (also über mehrere Bounding-Boxen) weg. Zum anderen wurde nur eine Klasse(rechter Zeigefinger) gelabelt und vorherge-

[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 Outputs 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.

sagt. So fällt auf der Zeile 5 das Summenzeichen und deren entsprechende Iteration über alle Klassen weg.

Spannend war, dass es im aktuellen Fall für das Netzwerk nicht möglich war einen nach unten korrigierenden Einfluss auf die Variable $\hat{p}_i$ zu nehmen. So gingen die Vorhersagen für die $\hat{p}_i$'s je länger man lernte desto stärker gegen 1. Der Grund dafür lag darin, dass $\mathbb{1}_i^{obj}$ immer $= 0$ war, wenn **kein** rechter Zeigefingerspitz in dieser Gitterzelle enthalten ist. So wird das Netzwerk nie korrigiert, wenn es die Wahrscheinlichkeit höher einschätzt, als sie tatsächlich ist. Dies stellte allerdings kein Problem dar. Denn die letztendliche Identifikation des Fingers, welche mit dem resultierenden Wert aus $\hat{c}_i * \hat{p}_i$ ermittelt wurde, war wegen der Nähe von $\hat{p}_i$ zu 1 sozusagen gleichzusetzen mit $\hat{c}_i$.

Da auch mit mehreren Klassen $\hat{c}_i$ eine Dominante Rolle spielen würde, wenn es darum geht, ob ein Objekt erkannt wurde oder nicht, würde höchstwahrscheinlich die Performance nicht besser, wenn man Yolo mit mehr als einer Klasse trainieren würde.

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 mindestens zwei Boundingboxen pro Gitterzelle vorhergesagt würden. Dies aufgrund der Theorie [6], dass mit mehreren Boundingboxen verschiedene Features eines Bildes verwendet werden können, um einen Finger vorherzusagen.

3. Zum Vergleich: Im Paper Yolo v2 wurde anstelle von 7x7 Gitterzellen 13x13 Gitterzellen verwendet, was in einer höheren Präzision resultieren würde. Eine höhere Präzision wäre für das Endziel dieser Arbeit nur zu begrüssen. Wenn dies allerdings mit Yolo v1 umgesetzt würde, hätte man das Problem, dass wegen dem Grösseren Speicheraufwand, welcher für die zusätzlichen Elemente “verbraucht“ würde die Minibatchsize verkleinert werden müsste. So wird angenommen, dass mit einer kompletten Umstellung auf Yolo v2 ein besseres Resultat erzielt würde, als wenn man Yolo v1 einfach erweitern würde.

5 Tests

5.1 Erste Ansätze

Am Anfang ging man von der falschen Vorstellung aus, dass die Detektion der Finger über die Variable $\hat{p}_i$ im Output des Neuronalen Netzwerks (Tabelle 3) läuft. Aus diesem Grund wurden verschiedene Tests aufgebaut.

Ein Test ermittelte aufgrund von $\hat{p}_i$ und einem beliebigen Threshold einen Status für jede Gitterzelle. Diese Stati waren:

- "True-Positive"
- "True-Negative"
- "False-Positive"
- "False-Negative"

Wer das Kapitel 4.3 gelesen hat, kann jetzt schon schlussfolgern, dass dies keine brauchbaren Resultate liefern konnte. Denn wenn die Variable $\hat{p}_i$ mit andauerndem Lernen gegen 1 tendiert und nie nach unten korrigiert wird, übersteigt Sie somit irgendwann jeglichen Threshold und sagt in jeder Gitterzelle einen rechten Zeigefingerspitz voraus. Entsprechend ergaben die "True-Positives" und die "False-Positives" zusammen irgendwann = 1, während die "True-Negatives" und die "False-Negatives" zusammen = 0 ergaben. Somit war dieser Test gegenstandslos und man wusste, dass die Variable $\hat{p}_i$ keine Rolle spielen wird, solange nur Labels mit einem Finger verwendet werden und entsprechend nur eine Klasse existiert.

Ein weiterer Test hatte das Ziel, dass jeweils rund um das jeweilige Label ein Kreis mit einem bestimmten Radius gelegt wird. Die Vorhersagen wurden nun aufgeteilt in Vorhersagen, welche innerhalb des Kreises lagen, und Vorhersagen ausserhalb des Kreises. Die Frage war nur noch, wie bestimmt man, welche der 7x7 Boundingboxen als **die** Vorhersage verwendet wurde, welche mit dem Label verglichen werden konnte. Die Antwort ist dank dem Wissen über die Aufgabe, welche das Netz erfüllen muss relativ schnell beantwortet. Denn wir wissen, dass in der Aufgabe, welche gelöst werden soll immer nur eine rechte Zeigefingerspitze pro Bild vorhanden sein wird. So musste dafür kein Threshold bestimmt werden, sondern es wurde einfach diejenige Boundingbox gewählt, welche die grösste Confidence lieferte.

Mit diesem Ansatz hatte man nun ein Test, der tatsächlich etwas über das Resultat aussagte. So wurden verschieden grosse Kreise um die Labels gezogen um prozentuale Aussagen zu deren Genauigkeit zu kriegen. Allerdings wurde relativ bald klar, dass es keinen grossen Sinn machen würde für jede Genauigkeit einen Kreis zu ziehen und diese dann einzeln auszuwerten.

Ausserdem wurde von Guido Schuster [6] in einem Gespräch folgende Bemerkung gemacht: "Man sollte das Netzwerk darauf testen, worauf man es auch trainiert hatte."Dieser Bemerkung folgte schliesslich die Schlussfolgerung, dass mit den Tests auch die IOU der Predictions gegenüber den Labels genauer betrachtet werden sollte.

5.2 Letztendlicher Test

Aus den Erkenntnissen der ersten Ansätze konnte ermittelt werden, dass der Optimale Output aus den Tests ein Histogram, bzw. eine Wahrscheinlichkeitsdichteverteilung sein sollte. So konnte der Code relativ schnell so angepasst werden, dass bei einem Testlauf die Distanz von Label zu Prediction (L2-Norm) für jedes Testbild in ein Element eines Vektors gespeichert wurde. Aus diesem Vektor konnte dann ein schönes Histogramm erstellt werden, aus welchem mit einem Blick gelesen werden konnte, wie sich die Distanzen von Predictions zu den Labels über das gesamte Testset verhielten.

Nach Fertigstellung dieses Tests war es ein Leichtes dasselbe für die IOU anstelle der Distanz zu machen.

Die Wahl der besten Prediction wurde ebenfalls nochmals verbessert. Im Nachherein ist es ein wenig peinlich, dass in diesem Punkt so viel herumexperimentiert und ausprobiert wurde, da in der Gleichung 1 im Yolo-Paper [3] klar ersichtlich ist, dass für die Bestimmung der besten Prediction das Produkt aus $\hat{p}_i$ und $\hat{c}_i$ massgebend ist. Natürlich machte dies aber auch keinen Unterschied mehr, da $\hat{p}_i$ nahezu $= 1$ war, war die Prediction aus $\hat{p}_i * \hat{c}_i$ dieselbe wie wenn nur $\hat{c}_i$ verwendet wurde.

Somit war das Training des Netzwerks auf die Variable $\hat{p}_i$ sowie dessen Verwendung bisher nur irreführend und hatte keinen Nutzen. Für die Zukunft aber ist es wichtig, dass dieses Yolo auch mehrere Klassen vorhersagen kann. Dadurch macht es Sinn, diese Variable und die damit verbundenen Berechnungen in der Implementation zu belassen.

Die Wahrscheinlichkeitsdichtefunktion, welche schliesslich bei diesen Tests durch die L2-Distanz erzeugt wurde, hatte eine etwas spezielle Form (siehe Abbildung 12). Dies sorgte Anfangs für Verwirrung. Allerdings konnte

ein Gespräch mit Guido Schuster [6] schnell Klarheit bringen, da es sich “offensichtlich“ um eine Rayleigh-Verteilung handelte. Eine solche Verteilung entsteht, wenn zwei Gaussverteilte Variablen über die L2-Norm miteinander verbunden werden. Da genau dies in diesem Test geschieht, war somit dieser Punkt restlos geklärt.

5.3 Seed

Um während den vielen Tests endlich ganz genaue Vergleiche zu erhalten wurde im Laufe der Arbeit ein fixer Seed implementiert und an Tensorflow übergeben. Allerdings hatte dies zwei Tücken. Dies waren auch die Gründe, warum dieser fixe Seed wieder aufgehoben wurde.

Die erste Tücke war, dass trotz der Übergabe eines fixen Seeds an Tensorflow die Ergebnisse trotzdem nicht reproduzierbar waren. Offensichtlich hat es in Tensorflow noch weitere zufällige Werte, welche man auch mit einem festen Seed initialisieren müsste. Diese wurden allerdings nicht gefunden.

Die zweite Tücke war jedesmal, wenn man während dem Training zwischen dem Trainingsset und dem Validierungsset hin und her wechselte. So wurden die Bilder für das Training wieder in der genau gleichen Reihenfolge geladen wie in der letzten Epoche. Zuerst sah es so aus, als würde der Trainingsfehler einfach in ungeheurem Tempo gegen 0 gehen, während sich der Validierungsfehler relativ schnell von jeglich vernünftigem verabschiedete. Allerdings wurden einzig und allein die ersten paar Bilder auswendig gelernt.

Der Vorteil an der zweiten Tücke war, dass man aus einem Versehen heraus sogleich überprüft hatte, ob das Netzwerk in der Lage ist overzufitten. ==> Ja ist es!

5.4 Fazit

Bei einem Neuronalen Netzwerk sollte sich früher Gedanken gemacht werden, wie man das Resultat möglichst praxistauglich testen kann. Dies wurde in dieser Arbeit klar falsch gemacht. Man hatte eine funktionierende Kostenfunktion und wollte diese nach Möglichkeit verbessern. Allerdings ist das Ziel eines neuronalen Netzwerks nicht eine tiefe Kostenfunktion zu haben, sondern den Task wozu es verwendet wird möglichst gut zu erfüllen.

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 mit den Trainingsbildern 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 (siehe Kapitel 5.2). 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 der Fingerspitze 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 8 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

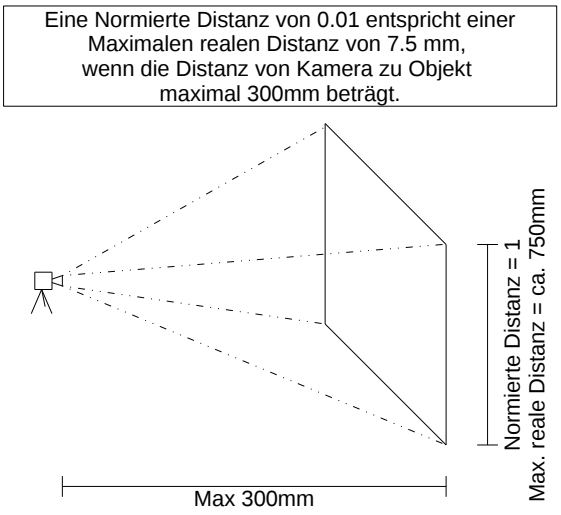

Abbildung 8: Bedeutung der normierten Distanzwerte in der realen Welt

genau der Breite eines menschlichen Fingers.

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 9 & 10 anzusehen, welche Bilder zeigen, die eine Distanz nahe dieses Thresholds aufweisen.

Um die Verteilung der Distanzen gut verstehen zu können, ist in Abbildung 11 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 “erfolgreich“ erkannt 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

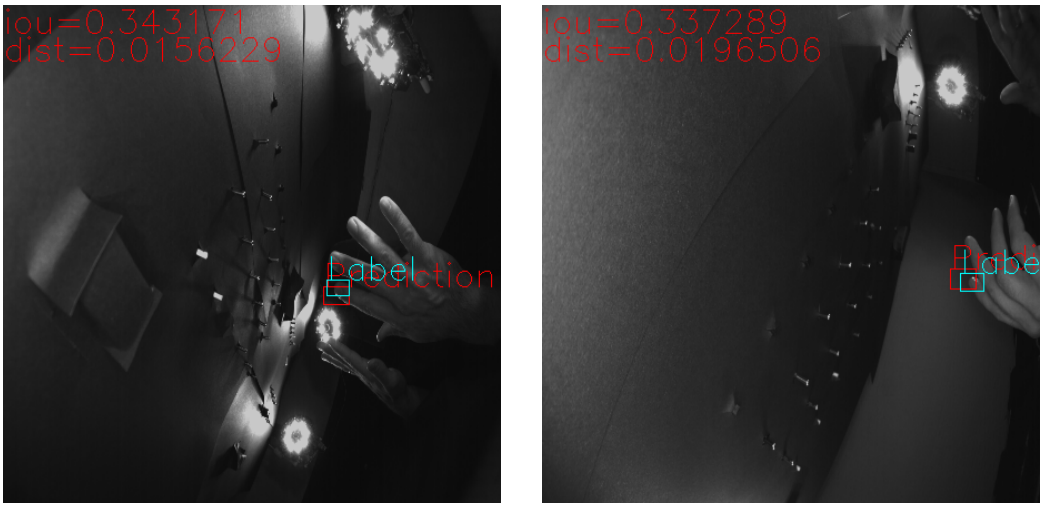

Abbildung 9: Prediction knapp besser als Distanz=0.02

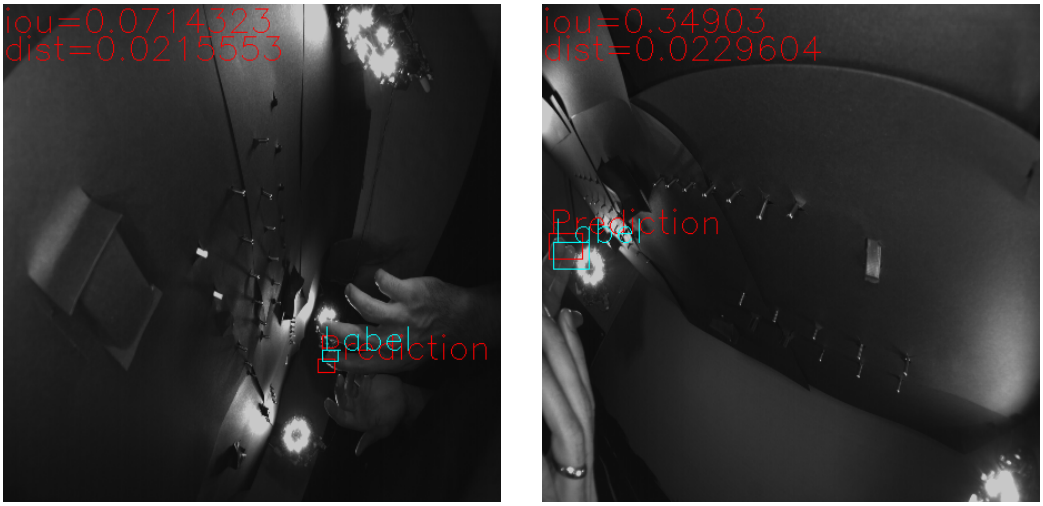

Abbildung 10: Prediction knapp schlechter als Distanz=0.02

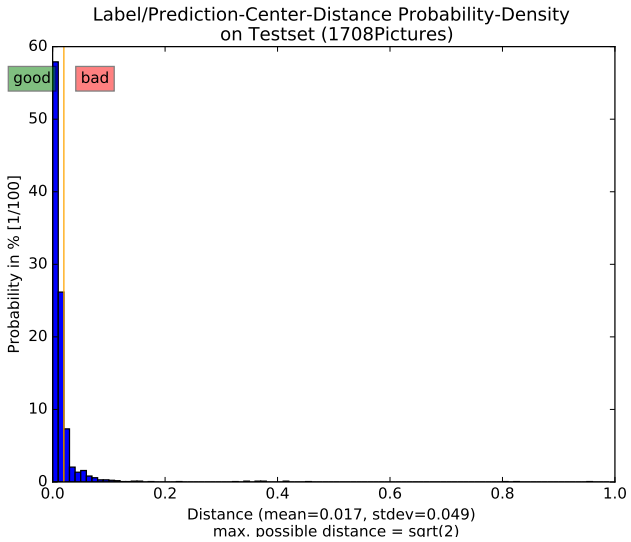

Abbildung 11: Komplette Wahrscheinlichkeits-Dichte-Funktion der Distanz (Grenze: Dist=0.02)

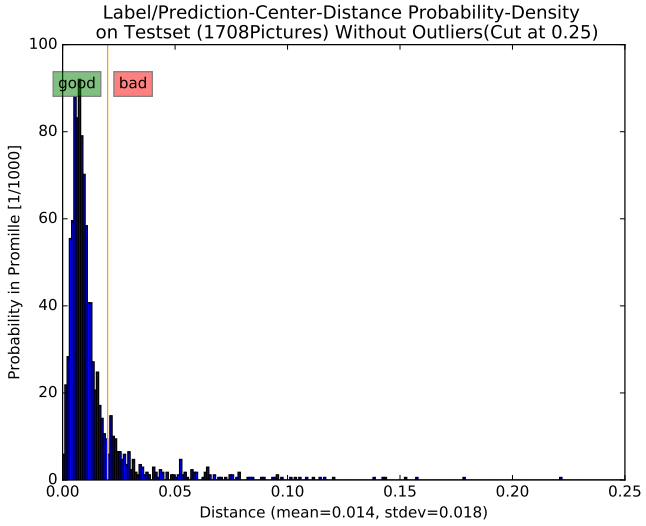

Abbildung 12: Wahrscheinlichkeits-Dichtefunktion der Distanz. Ausreisser nicht miteingerechnet (Grenze: Dist=0.02)

wurde, welcher tatsächlich ganz selten in den Labels als Finger markiert wurde.

Um die Statistik nicht von Ausreissern, welche aufgrund von falschen Labels entstanden sind verfälschen zu lassen, wurde wie in Abbildung 12 noch eine zweite Wahrscheinlichkeitsdichte-Funktion erstellt. Spannend: Der Mittelwert ist sofort um einen Drittel kleiner als zuvor.

Wie man ausserdem aus Kapitel 2.2 entnehmen kann, gab es schon in der Erzeugung der Labels eine gewisse Unschärfe. So hat auf Abbildung **??** der y-Wert im Vergleich vom einen Bild zum Anderen und relativ zum Finger eine Differenz von rund 10 Pixeln, was im normierten Mass einer Distanz von rund 0.02 entspricht. Klar, um dieses Problem aufzeigen zu können wurde ein Bild mit einem klaren Unterschied verwendet, was bedeutet dass die meisten Labels Fehler haben die kleiner als diese Grenze sind. Nichtsdestotrotz wird man wohl nie in der Lage sein viel bessere Resultate in der Genauigkeit einzufahren, wenn die Labels noch solche Abweichungen aufweisen.

Es kann sehr gut sein, dass in der Arbeit "Hand Pose Estimation" [5] bessere Labels erzeugt wurden. Denn in dieser Arbeit [5] wurden die Fingerspitzen im 3D-Raum bestimmt und anschliessend in den 2D-Raum zurückgemappt. Ausserdem wurde eine Fehlerrechnung gemacht, wie weit die 2D-Labels mit den zurückprojezierten Labels aus dem 3D-Raum übereinstimmen. Allerdings wurde in der erwähnten Arbeit [5] kein Vergleich gemacht wie dies in dieser Arbeit der Fall ist, weshalb keine abschliessende Aussage zu diesem Thema gemacht werden kann.

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 13 & 14 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

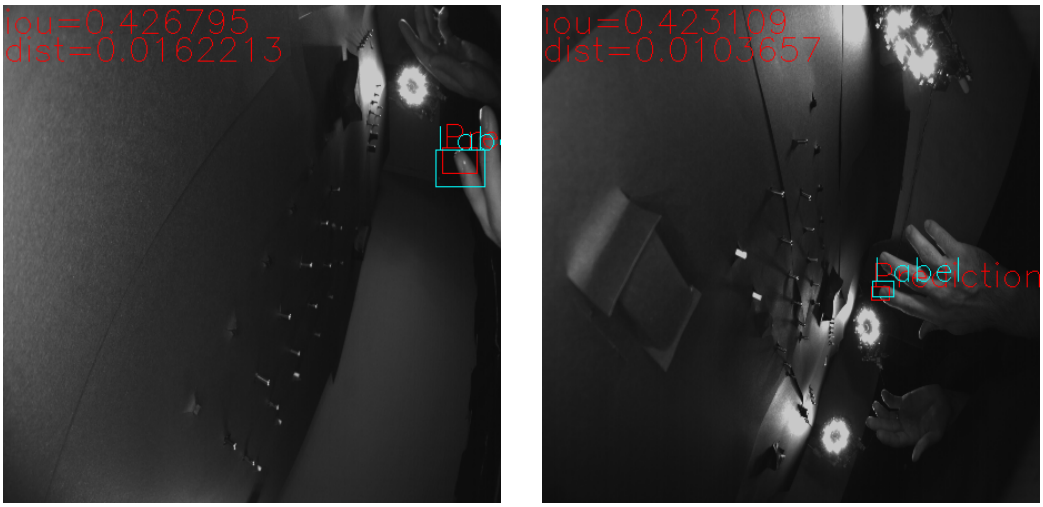

Abbildung 13: Prediction knapp besser als IOU=0.4

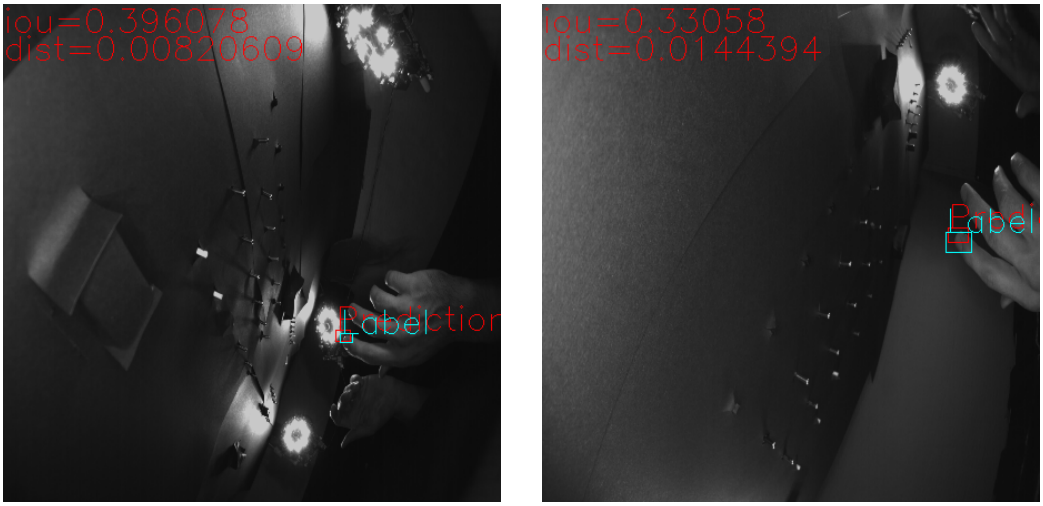

Abbildung 14: Prediction knapp schlechter als IOU=0.4

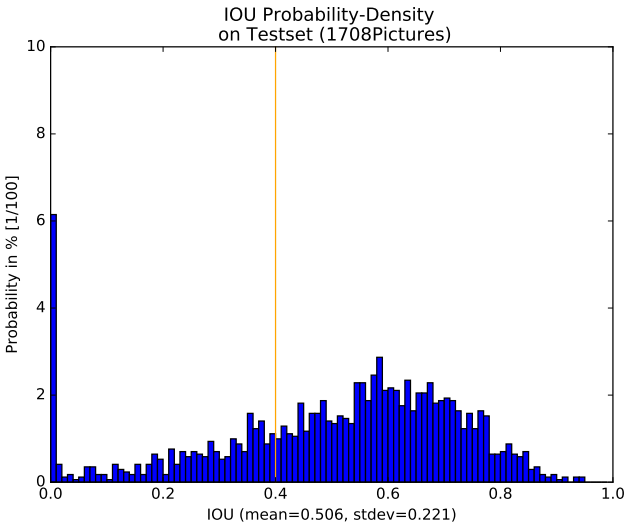

Abbildung 15: Wahrscheinlichkeits-Dichte-Funktion der IOU (Grenze: IOU=0.4)

in Abbildung 15 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.

Genau wie bei der Distanz gibt es auch bei der IOU, bzw. bei den Boundingboxen eine gewisse Unschärfe in den Labels (siehe Kapitel 2.2). So haben die beiden Bilder in Abbildung **??** wenn man Sie zueinander normiert gegenüber einander eine IOU von knapp 0.4. Wie bei den Labelfehlern in der Distanz dürften die Fehler auch bei den Boundingboxen klar kleiner als im hier berechneten Beispiel sein. Nichtsdestotrotz wird man wohl auch hier nie in der Lage sein viel bessere Resultate in der Genauigkeit einzufahren, wenn die Labels noch solche Abweichungen aufweisen.

Dass die Fehler in Distanz und IOU nahezu genau auf die gesetzten Grenzen zur Bewertung der Resultate fielen, war reiner Zufall. So sind die Grenzen gemacht worden, bevor IOU und Distanz der Labels dieser zwei Bilder (Abbildung 1) zueinander berechnet wurden.

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 (Tabelle 4) 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 originalen Yolo-Paper [3] ebenso gehandhabt und begründet wurde.

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

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)

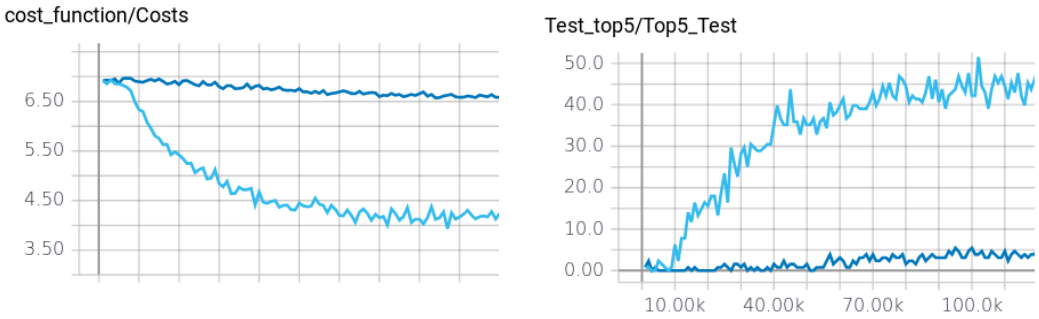

Abbildung 16: Kosten und Top5-Test von vergleichbaren Tasks im Pretraining über einen beschränkten Zeitraum. links: [x-Achse=Zeit, y-Achse=Kosten] rechts: [x-Achse=Zeit, y-Achse=Top-5 Treffer in %] SGD=dunkelblau. ADAM=hellblau.

auch einfach herausgestellt, dass die Performance von Adam gegenüber SGD massgeblich gesteigert hatte. (Abbildung 16)

Es gab allerdings auch eine Peinlichkeit im Zusammenhang mit dem Adam-optimizer. So wurde in dieser Arbeit viel mit verschiedenen Lernraten experimentiert, wie z.B. eine regelmässige Reduktion der Lernrate nach jeweils einer bestimmten Anzahl Epochen. Diese Experimente ergaben keinerlei verwertbare Resultate. Der Grund dafür war, dass man den Adam-Optimizer noch nicht richtig verstanden hatte. Denn ein zentraler Wert des Adam-Optimizers ist, dass dieser die Lernrate selber automatisch im Laufe des Trainings “anpasst“. Entsprechend ist eine manuelle Anpassung der Lernrate völlig Sinnlos.

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 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 17 erkennen kann, wurde nach 6 Tagen Lernzeit im Top5-Test im Schnitt eine Treffsicherheit von rund 80% erreicht. Da im originalen Yolo-Paper [3] eine Treffsicherheit von 88% erreicht wurde, war das Resultat dieser Arbeit um rund 8% schlechter.

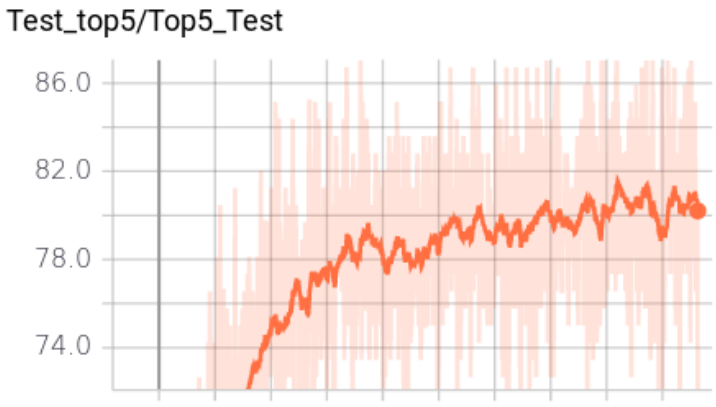

Abbildung 17: Test-Top5 nach 6 Tagen Lernzeit(x-Achse=Zeit, y-Achse=Top-5-Treffer in %)

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. Denn mit 80% Genauigkeit im Top5-Test war man genügend genau, um sich erst einmal auf das eigentliche Training fokussieren zu können.

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 Proble-

me gab vortrainierte Tensorflow-Gewichte direkt ins echte Training einzulesen, wenn man nicht genau dieselbe Architektur hatte wie im Pretraining, was offensichtlich der Fall war (Siehe Tabellen 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.

Obwohl der Aufbau einer Klassifizierungsarchitektur so einfach ist, darf das Debugging nicht unterschätzt werden. Viele Fehler sind (wenn überhaupt) erst nach mehrtägigem Training offen ersichtlich. Entsprechend musste auch beim Test von Korrekturen jedes Mal ein bis zwei Tage gewartet werden, bis man ein Feedback bekam. Neben Experimenten mit Lernraten, Dropout, etc., welche ebenfalls viel Zeit in Anspruch nahmen, bis ein Feedback verfügbar war, war dies der Hauptgrund für die zeitliche Verzögerungen in dieser Arbeit.

8 Fazit

8.1 Gipshände

Es wurde versucht eine Aussage machen zu können, ob Gipshände geeignet wären, um zukünftig Trainingsdaten zu generieren oder trainierte Netzwerke zu verifizieren. Leider war der Hintergrund, auf den Fotos mit den Gipshänden komplett anders als in den Trainingsdaten von Yolo (schwarz abgedeckt, siehe Kapitel 2.1). Entsprechend wurde die Gipshand kaum erkannt und viel mehr völlig willkürliche Predictions gemacht. Aus diesem Grund kann zu diesem Punkt an dieser Stelle leider keine Aussage gemacht werden.

8.2 Yolo

Die Genauigkeit von Yolo v1 auf Distanz und IOU ist aus menschlich subjektiver Sicht befriedigend. Leider aber ist die Genauigkeit noch weit vom gesteckten Ziel von 0.1 mm entfernt. Die härtesten Gründe dafür sind:

1. Die Bilder müssen für Ihre Verwendung in Yolo auf eine Grösse von 448x448 geschrumpft werden. Entsprechend ist das erreichen einer Genauigkeit unter 1mm in diesem Kontext nicht oder kaum vorstellbar.

2. Die Label-Daten hatten selber noch Fehler, die klar über 1mm lagen, entsprechend konnte es dem Netzwerk nicht möglich sein eine bessere Performance zu erlangen als dies das Trainingsset erlaubte.

3. Es wurde noch nicht alles aus dem Konzept Yolo herausgepresst, so stellt das Paper Yolo-v2 [4] einige Features vor, welche auch die Performance dieses Tasks verbessern könnten.

Um auf die eigentliche Frage zurück zu kommen. Ist Yolo, bzw. Deep-Learning im allgemeinen geeignet um die 2-D-Position von Fingerspitzen auf Bildern zu ermitteln? Lautet die Antwort höchstwahrscheinlich Ja. Ein Grund für diese Antwort ist, dass mit nur relativ wenigen Daten, welche sich auch als nicht allzu hochwertig herausgestellt haben mit dieser Methode trotzdem an der Grenze des möglichen gekratzt wurde. Wieviel mehr wäre da mit hochwertigeren Daten, mehr Daten und einem verbesserten Neuronalen Netzerk möglich...

8.3 Vorschläge für weitere Schritte

Um in erster Instanz das Ziel eines 2-D-Fingertrackers auf Basis von Deep-Learning zu erreichen werden folgende Punkte für die Zukunft vorgeschlagen:

1. Bessere Daten Sammeln.

   Es gibt die Faustregel, dass man dreimal Daten aufnimmt, bis man endlich die richtigen Daten hat, um sein Netzwerk optimal zu trainieren [6]. Entsprechend sollte dabei unbedingt darauf geachtet werden, dass man mit berücksichtigt, welche Handstellen (Finger, Knöchel, Gelenke, etc.) man überhaupt braucht, damit man die Daten nicht nochmals aufnehmen muss. Nach dieser Projektarbeit, welche Daten aus einer Hervorragenden Masterarbeit [2] verwenden durfte, ist klar, dass automatisch und maschinell erstellte Daten nicht zum optimalen Ziel führen können. Entsprechend sollte das Labeling von Fingern, etc. in Zukunft von Menschen gemacht werden.

2. Mehr Daten Sammeln.

   Wie Ian Goodfellow schon in seinem Buch Deeplearning [1] geschrieben hatte, wenn man alles ausprobiert hat und nichts mehr nützt, dann sammle mehr Daten. Entsprechend sollte bei der Generierung der Daten wie im Punkt 1 erwähnt darauf geachtet werden, dass die Daten so generiert werden, dass sie einfach per Computer vervielfältigt werden können.

3. Die besten Stücke aus Yolo v1 und Yolo v2 herauspicken und ein optimales Netzwerk für diesen Task erschaffen.

   Für diesen Punkt sollte allerdings noch die Arbeit von Jonas Schmid [5] berücksichtigt werden, weil dort höchstwahrscheinlich ebenfalls wichtige Erkenntnisse zu diesen Thema enthalten sind.

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] Jonas Schmid. Hand Pose Estimation: Hand and Fingertip Detection using Deep Convolutional Neural Networks. Master's thesis, HSR, Februar 2018.

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

Abbildungsverzeichnis

1 Resultate der Erosion 9

2 Label-Tensor 13

3 Liste von Label-Tensoren 14

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 18

5 Aufbau eines Convolutional-Layers 20

6 Output-Tensor 23

7 Berechnung der IOU 24

8 Bedeutung der normierten Distanzwerte in der realen Welt . . 32

9 Prediction knapp besser als Distanz=0.02 33

10 Prediction knapp schlechter als Distanz=0.02 33

11 Komplette Wahrscheinlichkeits-Dichte-Funktion der Distanz (Grenze: Dist=0.02) 34

12 Wahrscheinlichkeits-Dichtefunktion der Distanz. Ausreisser nicht miteingerechnet (Grenze: Dist=0.02) 34

13 Prediction knapp besser als IOU=0.4 36

14 Prediction knapp schlechter als IOU=0.4 36

15 Wahrscheinlichkeits-Dichte-Funktion der IOU (Grenze: IOU=0.4) 37

16 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. 40

17 Test-Top5 nach 6 Tagen Lernzeit(x-Achse=Zeit, y-Achse=Top-5-Treffer in %) 41