



# VERWENDEN VON GRAFIK-ENGINES ZUR SYSTEMATISCHEN AUSWERTUNG UND ZUM SZENENSPEZIFISCHEN TRAINING VON OBJEKTDETEKTOREN

bei der Bosch Sicherheitssysteme GmbH

Fakultät für Maschinenbau  
der Hochschule Stralsund

**Abschlussarbeit**  
zur Erlangung des akademischen Grades  
**Bachelor of Engineering**

vorgelegt von

**Arne Lück**

geboren am 24.08.1996 in Hildesheim

**Bearbeitungszeitraum:**

vom: 24. Mai 2019

bis: 02. August 2019

**Erstprüfer:** Prof. Dr.-Ing. J. Ladisch (Hochschule Stralsund)  
**Zweitprüfer:** Dr. rer. nat. F. Richter (Bosch Sicherheitssysteme GmbH)



# **Eidesstattliche Erklärung**

Ich versichere, die von mir vorgelegte Arbeit selbstständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel sind angegeben. Die Arbeit hat mit gleichem beziehungsweise in wesentlichen Teilen gleichem Inhalt noch keiner Prüfungsbehörde vorgelegen.

---

Ort und Datum

Unterschrift



# Danksagung

An dieser Stelle möchte ich mich sehr herzlich bei Herrn Dr. Florian Richter von *Bosch Sicherheitssysteme* für die großartige Unterstützung bei der Erstellung dieser Arbeit bedanken. Von seiner Expertise im Bereich des Machine Learning und seiner mitreißenden Begeisterung für das Thema habe ich sehr profitiert. Vielen Dank auch an Herrn Prof. Ladisch für die Betreuung dieser Arbeit von Seiten der Hochschule Stralsund und für seine Unterstützung während meines Studiums und bei der Praktikumssuche. Auch bei der restlichen Abteilung *BT-SC/ETP1.4* bedanke ich mich herzlich für ihre Unterstützung und meine herzliche Aufnahme in ihrer Runde. Bei Herrn Dr. Emil Schreiber möchte ich mich für die Vermittlung des Kontakts zu Herrn Dr. Richter und damit auch für die Möglichkeit, diese Arbeit zu schreiben, bedanken. Meinen Kommilitonen möchte ich für die gute Zusammenarbeit und gegenseitige Hilfe während des Studiums und speziell Heiko Hillenhagen für das Korrekturlesen dieser Arbeit danken. Zu guter Letzt möchte ich mich auch bei meinen Eltern für das Korrekturlesen dieser Arbeit sowie für ihre Unterstützung in meinem bisherigen Leben bedanken.



# Inhaltsverzeichnis

<b>Abbildungsverzeichnis</b>	<b>VIII</b>
<b>Abkürzungsverzeichnis</b>	<b>X</b>
<b>1 Einleitung</b>	<b>2</b>
1.1 Videoüberwachung heute . . . . .	2
1.2 Einsatz von Deep Learning in der Videoanalytik . . . . .	2
<b>2 Ziel der Bachelorarbeit</b>	<b>4</b>
2.1 Trainieren von Detektoren auf spezifische Anwendungsfälle . . . . .	5
2.2 Gezielte Suche nach Schwachstellen der Detektoren . . . . .	7
<b>3 Grundlagen</b>	<b>10</b>
3.1 Neuronale Netze . . . . .	10
3.1.1 Aufbau und Funktionsweise von Neuronalen Netzwerken . . .	10
3.1.2 Training von Neuronalen Netzen . . . . .	13
3.1.3 Beispiel: Erkennung handgeschriebener Ziffern . . . . .	14
3.2 Programmiersprachen . . . . .	15
3.2.1 Python . . . . .	15
3.2.2 C++ . . . . .	16
3.3 Unreal Engine 4 . . . . .	16
3.3.1 Aufbau von Personenmodellen in Game Engines . . . . .	17
3.3.2 UnrealCV . . . . .	18
3.3.3 Vergleich der <i>Unreal Engine 4</i> mit anderen Rendering Engines	19
<b>4 Datengenerierung</b>	<b>20</b>
4.1 Modellierung der Szene . . . . .	20
4.2 Automatisierung der Szene . . . . .	22
4.2.1 Spawner . . . . .	23
4.2.2 Navigation in der Szene . . . . .	23
4.3 Rendern der Bilder . . . . .	24
4.4 Keypoint Generierung . . . . .	25

4.5	Transformierung der Keypoint-Koordinaten aus dem <i>Welt</i> -Koordinatensystem ins Bild-Koordinatensystem . . . . .	28
<b>5</b>	<b>Training und Evaluierung von Objektdetektoren</b>	<b>32</b>
5.1	Training der Detektoren . . . . .	32
5.2	Evaluierung der Detektoren . . . . .	34
<b>6</b>	<b>Ergebnisse, Fazit und Ausblick</b>	<b>36</b>
6.1	Ergebnisse . . . . .	36
6.1.1	Training mit künstlichen Daten . . . . .	36
6.1.2	Evaluierung des Gesichtsdetektors in Abhängigkeit der Blick- richtung und der Beleuchtung . . . . .	42
6.2	Ausblick . . . . .	51
6.3	Fazit . . . . .	51
	<b>Literaturverzeichnis</b>	<b>XI</b>
	<b>Anhang</b>	<b>XIV</b>

# Abbildungsverzeichnis

2.1	Echtweltbild aus Trainingssequenz für 3D-Personendetektor . . . . .	6
2.2	Synthetisches Bild aus Trainingssequenz für 3D-Personendetektor . .	6
2.3	Übersicht über den Verlauf des Trainings mit künstlichen Daten . .	7
2.4	Übersicht über die gezielte Schwachstellensuche mit künstlichen Daten	8
3.1	Schematische Darstellung eines künstlichen Neurons . . . . .	10
3.2	Schematische Darstellung eines kleinen Neuronalen Netzwerks . . .	11
3.3	Graphische Darstellung der <i>Sigmoid Funktion</i> . . . . .	12
3.4	Graphische Darstellung der ReLU-Funktion . . . . .	13
3.5	Exemplarischer Auszug aus der <i>MNIST</i> Datenbank . . . . .	14
3.6	Schematische Darstellung eines Neuronalen Netzwerks . . . . .	15
3.7	Supermarktszene aus der <i>Unreal Engine 4</i> . . . . .	17
3.8	Skelett einer Person in der <i>Unreal Engine 4</i> . . . . .	18
4.1	Arbeitsablauf der Datengenerierung . . . . .	20
4.2	Vergleich <i>Birds-Eye-View</i> Ansicht und perspektivischer Ansicht . .	21
4.3	Vergleich: reale Szene mit digitalem Nachbau . . . . .	22
4.4	Grafische Programmierung des Spawners eines Characters . . . .	23
4.5	Navigationsbereiche einer Szene . . . . .	24
4.6	Beispielhaft vier Characters im Editor der <i>Unreal Engine 4</i> . . . .	27
4.7	Koordinatensystem der Kamera in dem Welt-Koordinatensystem der <i>Unreal Engine 4</i> . . . . .	29
4.8	Schematischer Aufbau einer Kamera zur Veranschaulichung des Blickfeldes . . . . .	30
4.9	Veranschaulichung der Abbildung von Objekten in einer Kamera . .	31
5.1	Bildpyramide mit vier Skalenstufen . . . . .	33
5.2	synthetisches Trainingsbild mit eingezeichneten Positiv-Patches . .	34
6.1	ROC-Kurve eines synthetisch trainierten Detektors mit verschiede- nen Test-Datenbanken . . . . .	37
6.2	Ausgewählte Bilder aus den drei Datenbanken . . . . .	38
6.3	ROC-Kurven zweier Kopf-Schulter-Detektoren (real und synthetisch trainiert) . . . . .	38

6.4	ROC-Kurven zweier Kopf-Schulter-Detektoren verschiedener Größen	40
6.5	Echtweltbild mit Auslösungen eines Personendetektors	41
6.6	ROC-Kurven zweier Kopf-Schulter-Detektoren (real und synthetisch trainiert)	42
6.7	Bilder von fünf Gesichtern	43
6.8	Zwei Bilder aus einer Datenbank an Bildern von künstlichen Gesichtern	44
6.9	Durchschnittliche winkelabhängige Leistungsfähigkeit eines Gesichts- detektors	45
6.10	Winkelabhängige Leistungsfähigkeit eines Gesichtsdetektors auf einzelnen Bildern	46
6.11	Fehlauslösung eines Gesichtsdetektors	47
6.12	Gesicht in fünf verschiedenen Lichtsituationen mit den Detektionen eines Gesichtsdetektors	48
6.13	Bilder eines künstlichen Gesichts, variable Beleuchtung von der Seite	49
6.14	Konfidenzen eines Gesichtsdetektors mit seitlicher Beleuchtung in verschiedenen Intensitäten	49
6.15	Bilder eines künstlichen Gesichts, variable Beleuchtung von oben	50
6.16	Konfidenzen eines Gesichtsdetektors mit Beleuchtung in verschiedenen Intensitäten von oben	50

# Abkürzungsverzeichnis

$\alpha$	Gierwinkel . . . . .	28
$\beta$	Nickwinkel . . . . .	28
<b>API</b>	application programming interface (engl. für Programmierschnittstelle)	
	15	
<b>b</b>	bias (engl. für Tendenz) . . . . .	11
<b>C</b>	Kostenfunktion . . . . .	12
$D_y$	Drehmatrix zur Drehung um die Y Achse . . . . .	28
$D_z$	Drehmatrix zur Drehung um die Z Achse . . . . .	28
<b>o</b>	Ausgabewert (engl. Output) . . . . .	10
$P_I$	Bildkoordinate des Objektes	
$P_K$	Position der Kamera . . . . .	28
$P_O$	Position des Objektes . . . . .	28
$P_t$	transformierte Position des Objektes . . . . .	28
$P_v$	verschobene Position des Objektes . . . . .	28
<b>ReLU</b>	rectified linear unit (engl. für gleichgerichtete Lineareinheit) . . . . .	12
<b>S</b>	Schwellenwert . . . . .	11
<b>vgl.</b>	vergleiche	
$w_j$	Gewicht . . . . .	10
$x_j$	Eingabewerte . . . . .	10



# **1 Einleitung**

In dieser Arbeit werden Objektdetektoren zur Auswertung von Videoüberwachungsdaten auf der Basis Neuronaler Netze erstellt.

## **1.1 Videoüberwachung heute**

Videoüberwachung ist ein Thema, das in den letzten Jahren an Bedeutung und Präsenz im öffentlichen Leben gewonnen hat. Sie ist in vielen Situationen wichtig, um Sicherheit und Ordnung zu gewährleisten. Trotzdem wird sie von vielen Menschen kritisch gesehen. In diesem Zusammenhang wird auch häufig der Einsatz von automatisierter Videoauswertung und künstlicher Intelligenz kritisiert. Diese Technologien können jedoch auch sehr sinnvoll genutzt werden, um zum Beispiel Sicherheitskräfte zu entlasten. Monotone und mental anstrengende Aufgaben, wie die Kontrolle einer größeren Anzahl an Überwachungskameras, können teilweise sehr gut automatisiert werden, sodass die Sicherheitskräfte automatisch informiert werden, wenn etwas detektiert wird, was das Eingreifen von Wachpersonal erfordert. Dies hat zur Folge, dass das Sicherheitspersonal nur noch auf vorgefilterte Ereignisse reagieren und sich nicht dauerhaft auf viele Videosignale gleichzeitig konzentrieren muss. Auf diese Weise kann die mentale Belastung für Sicherheitspersonal verringert werden und die Überwachung effizienter gestaltet werden. [1]

## **1.2 Einsatz von Deep Learning in der Videoanalytik**

Traditionelle Programmieransätze der Videoanalytik basieren auf vielen handgemachten Bedingungen, die zur Erkennung von Menschen und Objekten verwendet werden. Diese Methode ist jedoch sehr aufwendig, da man für jedes neue Objekt die Bedingungen neu definieren muss. Im Gegensatz hierzu kann man einem Deep Learning Algorithmus (vgl. Kapitel 3) Positiv- und Negativbeispiele, zum Beispiel in Form von Bildern, geben und der Algorithmus erstellt und optimiert selbst Filter, durch die er Objekte erkennen kann. Der Vorteil dabei ist, dass sich das Netzwerk selbst an die zu erkennenden Objekte anpasst, was die Erstellung verschiedener Detektoren erleichtert. So ist es möglich, schneller bessere Detektoren zu erstellen, als es händisch möglich wäre. Für Deep Learning Algorithmen werden jedoch viele

Daten benötigt. Dies ist in manchen Situationen schwer mit Daten aus der echten Welt zu realisieren. Aus diesem Grund soll in dieser Arbeit untersucht werden, wie gut es möglich ist, Trainingsdaten mit Hilfe der in Computerspiele-Engines, wie der *Unreal Engine 4* eingebauten Grafik-Engines, zu erstellen.

## 2 Ziel der Bachelorarbeit

Für das Training von Neuronalen Netzen (vgl. Kapitel 3.1.2) werden große Datenmengen benötigt. Im Fall der Videoanalyse bedeutet das, dass viele Bilder mit akkurate Metainformationen über deren Inhalt erforderlich sind, um einen guten Detektor zu trainieren. Die Qualität des Detektors hängt hierbei stark von der Menge an Bildern und der Qualität der Annotationen ab. Um Bilder aus der echten Welt zu nutzen, müssen diese mit Metadaten versehen werden. Hierzu zählen Informationen wie die Positionen von Objekten im Bild. Diese Annotationen werden meistens komplett manuell oder mit Hilfe von leistungsfähigen, schon existierenden Detektoren erstellt [2]. So erstellte Daten sind jedoch häufig teuer und teilweise fehlerhaft. Außerdem werden schärfere Datenschutzbestimmungen speziell in Europa zu einer immer größeren Hürde bei der Nutzung von Daten [2]. Aus diesen Gründen ist die künstliche Generierung von Trainings- und Evaluierungsdaten eine interessante Option. Zudem ist man in der Gestaltung der Szene erheblich uneingeschränkter als bei der Datengenerierung in der echten Welt, da Änderungen hieran wesentlich schneller und günstiger sind.

"Der Preis für Rechenleistung und Speicher ist in den letzten zehn Jahren dramatisch gesunken. Diese Abnahme läutete eine neue Ära der Computervision ein, eine Ära, die [...] riesige Mengen an annotierten Daten nutzt." [3] Auch die Qualität von künstlich erzeugten Welten wie etwa in Computerspielen wurde durch das Aufkommen von leistungsfähigeren Computern und Grafikkarten erheblich gesteigert. Aus diesen Welten können Bilder erstellt werden, die auch als Daten für die Arbeit in der Videoanalytik genutzt werden können. So ist es möglich, große Mengen an hochwertigen Daten zu erzeugen. Dass dies generell möglich ist, wurde schon in verschiedenen Projekten gezeigt [3, 4, 5]. So wurden an der *Autonomen Universität Barcelona* Detektoren zur Erkennung von Fußgängern mit künstlichen Daten trainiert und im Anschluss mit Daten aus der echten Welt getestet [6]. Auch die *Carnegie Mellon University* in Pittsburgh forscht gemeinsam mit der *Sony Corporation* an dem Training von Fußgängerdetektoren. In diesem Fall sollen diese an einzelne Szenen angepasst werden [7]. Selbst die Verwendung von Computerspielen wie *Grand Theft Auto V* zum Trainieren von selbstfahrenden Autos wurde schon getestet [8]. Die Arbeit mit künstlichen Daten ist seit einigen Jahren so erfolgreich und weitverbreitet, dass sogar kommerzielle Anbieter existieren,

die sich auf die Erstellung synthetischer Daten spezialisiert haben [9, 10]. Das externe Erstellen der Daten verringert jedoch die Flexibilität und Geschwindigkeit bei der Nutzung dieser und verursacht zusätzliche Kosten. Deshalb soll in diesem Projekt ein Arbeitsablauf erstellt werden, mit dem künstliche Daten zügig und einfach nach eigenen Vorstellungen erzeugt werden können. Die Arbeit mit diesen Daten soll im Anschluss in verschiedenen Anwendungsfällen getestet werden.

## 2.1 Trainieren von Detektoren auf spezifische Anwendungsfälle

Objektdetektoren können deutlich verbessert werden, wenn sie auch mit Bildern aus Szenen trainiert werden, in denen sie eingesetzt werden sollen. Der klassische Weg ist es, mit Hilfe von Kameraaufnahmen aus den Szenen den Detektor auf die gegebenen Kamerapositionen und Lichtverhältnisse zu trainieren. Es ist jedoch möglich, dass solches Videomaterial nicht zur Verfügung steht. Dafür kann es mehrere Gründe geben, einschließlich Persönlichkeits- und Urheberrechtsfragen gegenüber Dritten. Diese Probleme können mit künstlichen Daten sehr gut gelöst werden. Auch extreme Situationen können besser in das Training einbezogen werden. So sind beispielsweise Hintergründe und Lichtverhältnisse frei wählbar und auch die Anzahl und Dichte der Personen in der Szene lässt sich variabel gestalten. Auf diese Weise ist eine größere Flexibilität beim Trainieren der Detektoren gegeben. Ein Beispiel für eine Originalszene und das von mir nachgebaute Gegenstück dazu ist in den Abbildungen 2.1 und 2.2 zu sehen.

In dieser Arbeit wird das Trainieren von Detektoren mit Hilfe von künstlichen Daten am Beispiel eines Detektors zur Erkennung von Personen anhand ihrer Kopf-Schulter-Partien beschrieben. Eine schematische Darstellung dieses Prozesses ist in Abbildung 2.3 zu finden.

Der Prozess der Datengenerierung wird in Kapitel 4 näher beschrieben. Hierbei werden synthetische Bilder durch die *Unreal Engine 4* erstellt und parallel dazu die Positionen wesentlicher Körperteile der abgebildeten Personen in Textdokumenten gespeichert.



Abbildung 2.1: Bild aus Trainingssequenz für Objektdetektoren für den Einsatzzweck Flughafen. Diese Szene wurde von Mitarbeitern der Firma Bosch nachgestellt.



Abbildung 2.2: Künstlicher digitaler Nachbau einer realen Szene (vgl. Abbildung 2.1) als Grundlage zur Datengenerierung für das Training von Neuronalen Netzen

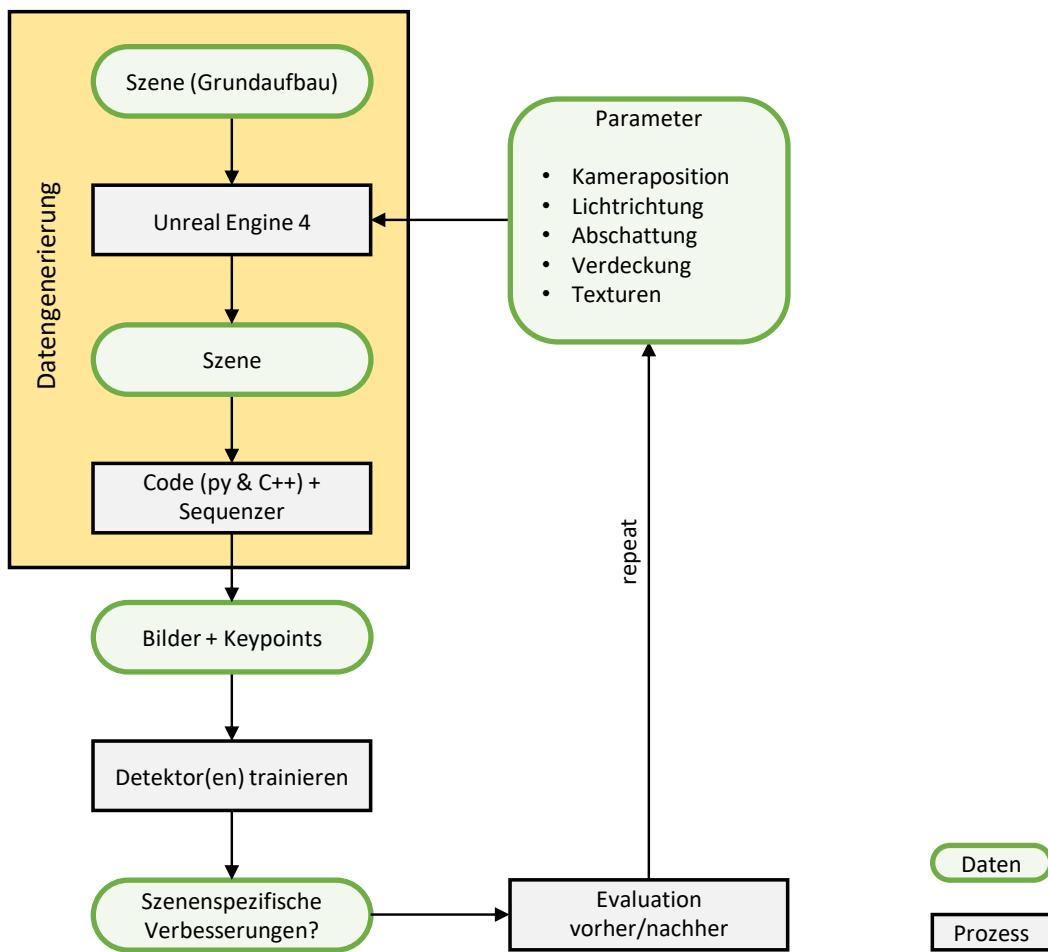


Abbildung 2.3: Übersicht über den Ablauf zum Trainieren eines Detektors. Enthalten hierin sind die Erstellung der Szene und das Extrahieren der Bilder sowie der Positionen von später gebrauchten Körperteilen. Im Anschluss wird der Detektor mit den zuvor erstellten Daten trainiert und bezüglich Verbesserungen evaluiert. Dieser Prozess kann je nach Bedarf wiederholt werden.

## 2.2 Gezielte Suche nach Schwachstellen der Detektoren

Ein zweiter Ansatz ist es, einen fertig trainierten Detektor mithilfe von künstlichen Daten zu testen. Dieses Verfahren hat den Vorteil, dass erkannte Schwachstellen systematisch untersucht werden können, da es möglich ist, Details in der Szene gezielt zu verändern und zu bestimmen, wie der Detektor darauf reagiert. Faktoren, wie die Intensität und Richtung der Beleuchtung und die Hintergründe, lassen sich einfacher und flexibler einstellen. Hierdurch ist es beispielsweise möglich, zu erfahren, bei welcher Beleuchtung die Detektoren gut arbeiten und welche Situationen

zu vermeiden sind. Zudem ist es möglich, die gefundenen Schwachstellen mithilfe der erstellten künstlichen Daten in das Training des Detektors einzubeziehen.

Ein großer Teil der Suche nach Schwachstellen lässt sich auch automatisiert durchführen. Ein Überblick über ein mögliches Vorgehen hierfür ist in Abbildung 2.4 zu sehen.

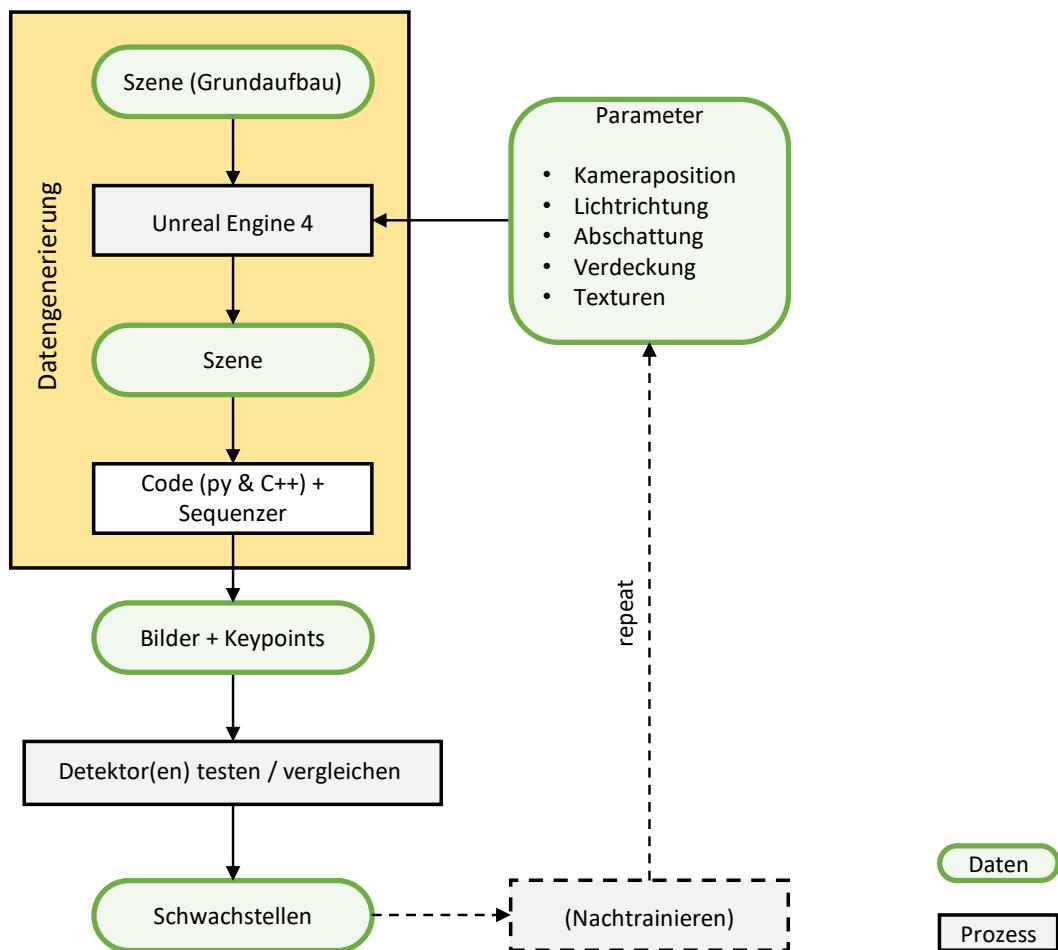


Abbildung 2.4: Übersicht über den Ablauf zum systematischen Evaluieren eines Detektors. Enthalten hierin sind die Erstellung einer Szene, das Extrahieren der Bilder sowie der Koordinaten der Positionen von später gebrauchten Körperteilen und das Evaluieren des Detektors mit den erstellten Daten. Dieser Vorgang wird wiederholt mit leicht abgewandelten Parametern durchgeführt.



# 3 Grundlagen

## 3.1 Neuronale Netze

"Neuronale Netze sind eines der schönsten Programmierparadigmen,  
die je erfunden wurden."

(Michael Nielsen [11])

*Dieses Kapitel ist in Anlehnung an das Buch 'Neural Networks and Deep Learning' von Michael Nielsen [11] geschrieben, welches eine gute verständliche und umfassende Einführung in das Themengebiet des maschinellen Lernens gibt.*

### 3.1.1 Aufbau und Funktionsweise von Neuronalen Netzwerken

Neuronale Netze sind in ihrem Grundkonzept von der Funktionsweise eines menschlichen Gehirns inspiriert. Wie der Name schon sagt, basieren sie auf einzelnen Neuronen, die auf Basis verschiedener Eingabewerte einen Ausgabewert ergeben (siehe Abbildung 3.1), ähnlich wie die Neuronen in einem menschlichen Gehirn. Die einfachste Art von Neuron ist das Perzeptron. [12].

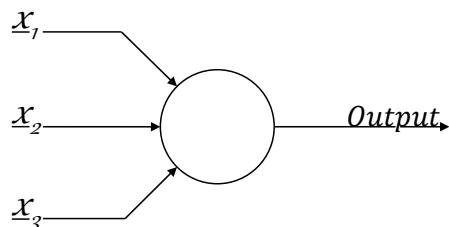


Abbildung 3.1: Darstellung eines Neurons mit drei Eingabewerten und einem Ausgabewert. Der Ausgabewert ergibt sich dabei aus den drei Eingabewerten.

Bei einem Perzeptron können sowohl die Eingabewerte ( $x_j$ ) =  $x_1, x_2, x_3$  als auch der Ausgabewert (engl. Output) (o) ausschließlich die Werte null und eins annehmen. Um den Ausgabewert zu bestimmen, werden die Eingabewerte jeweils mit einem Gewicht ( $w_j$ ) =  $w_1, w_2, w_3$  multipliziert und im Anschluss aufsummiert. Ist diese

Summe größer als ein vorgegebener Schwellenwert ( $S$ ), so wird der Ausgabewert zu eins.

$$o = \begin{cases} 0 & \text{wenn } \sum_j w_j x_j \leq S \\ 1 & \text{wenn } \sum_j w_j x_j > S \end{cases} \quad (3.1)$$

Um die Berechnung hierfür einfacher zu gestalten, wird der Schwellenwert durch einen bias (engl. für Tendenz) ( $b$ )

$$b = -S$$

ersetzt, so dass nun gilt:

$$o = \begin{cases} 0 & \text{wenn } \sum_j w_j x_j + b \leq 0 \\ 1 & \text{wenn } \sum_j w_j x_j + b > 0 \end{cases} \quad (3.2)$$

Um komplexere Entscheidungen treffen zu können, werden mehrere Neuronen hintereinander genutzt. Diese werden in Ebenen, sogenannten Layern, angeordnet, sodass die Neuronen in den hinteren Layern als Eingabewerte die Ausgabewerte der Neuronen in dem vorherigen Layer nutzen. So entsteht ein Netzwerk. Bildlich kann man sich dies vorstellen, wie in Abbildung 3.2 gezeigt. Sind alle Neuronen einer Ebene mit allen Neuronen der nächsten Ebene verbunden, spricht man von einem *fully-connected* (engl. für vollständig verbunden) Netzwerk (vgl. Abbildung 3.2). Die erste Ebene nennt man hierbei die Eingabeebene (engl.: Input Layer) und die letzte Ebene die Ausgabeebene (engl.: Output Layer). Die Ebenen dazwischen werden als verdeckte Ebenen (engl.: hidden Layer) bezeichnet, da diese Werte für den Nutzer des Netzes nicht direkt sichtbar sind und nur innerhalb des Netzes verwendet werden. Die Anzahl der Neuronen in einer Ebene sowie die Anzahl an verdeckten Ebenen sind vom Entwickler frei wählbar und Ergebnis der Entwicklungsarbeit an diesen Netzen.

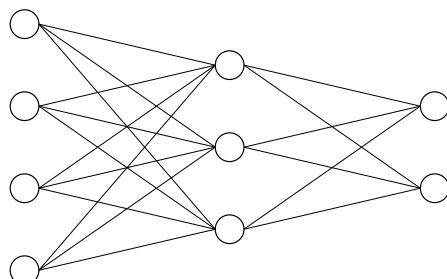


Abbildung 3.2: Schematische Darstellung eines kleinen vollständig verbundenen Neuronalen Netzwerks mit drei Ebenen. Die Kreise symbolisieren hierbei die einzelnen Neuronen mit den dazugehörigen biases, die Linien zwischen ihnen die Verbindungen mit den verschiedenen Gewichten.

Generell ist jedoch zu sagen, dass bei komplexeren Problemen größere Netze, also solche mit mehr Neuronen und Ebenen, bessere Ergebnisse erzielen. Die Größe der Eingabe- und Ausgabeebene sind an die gewünschte Art von Ein- und Ausgabe anzupassen. Soll beispielsweise ein Bild mit der Größe von  $20 \times 20$  Pixeln verarbeitet werden und als Ergebnis eine einfache Ja-Nein Entscheidung getroffen werden, so muss die Eingabeebene ( $20 \times 20 = 400$  Neuronen) umfassen, während für die Ausgabeebene ein einzelnes Neuron hingegen ausreichend ist. Mehr hierzu ist in dem Beispiel in Abschnitt 3.1.3 zu finden.

In den meisten Situationen wird durch einen kontinuierlichen Anstieg des Ausgabewertes in Abhängigkeit von kontinuierlich steigenden Eingabewerten das Training des Neuronalen Netzes vereinfacht. Daher nutzt man meist Neuronen mit einer stetigen Aktivierungsfunktion anstelle von einer Sprungantwort. Eine bislang weit verbreitete Funktion hierfür ist die *Sigmoid-Funktion* (vgl. Abbildung 3.3). Durch diese Funktion wird die Sprungantwort vermieden und durch eine kontinuierliche Kurve ersetzt, die ihren Eingabewert in den Bereich zwischen null und eins staucht.

$$\text{Sigmoid} : f(x) = \frac{1}{1 + e^{-x}} = \frac{e^x}{1 + e^x} = \frac{1}{2} \times \left(1 + \tanh \frac{x}{2}\right) \quad (3.3)$$

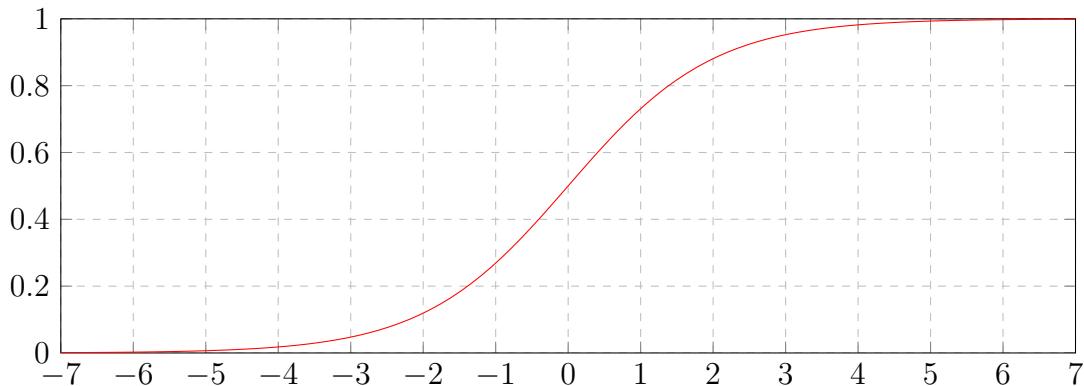


Abbildung 3.3: Graphische Darstellung der *Sigmoid Funktion*, einer speziell früher oft genutzten Aktivierungsfunktion für Neuronen. Sie staucht Zahlen in den Bereich zwischen null und eins (vgl. Gleichung 3.3)

Dies hat zur Folge, dass im Training (vgl. Abschnitt 3.1.2) die *Gewichte* und insbesondere die *biases* besser optimiert werden können, da sich diese Optimierungen an der Steigung der Kostenfunktion (C) (vgl. Gleichung 3.5) orientieren. Dieses Vorgehen verbessert den Trainingserfolg erheblich. Problematisch ist einzig, dass ein erhöhter Rechenaufwand erforderlich ist, um den Wert für eine Funktion mit einer Gleichung wie 3.3 zu berechnen. Aus diesem Grund wird heutzutage meistens keine *Sigmoid Funktionen* mehr genutzt. Stattdessen nutzt man sogenannte *rectified linear units* (engl. für gleichgerichtete Lineareinheiten) (ReLU's) (vgl. Gleichung

3.4, Abbildung 3.4), die einen extrem geringen Rechenaufwand erfordert.

$$\text{ReLU} : f(x) = \begin{cases} 0 & \text{wenn } x \leq 0 \\ x & \text{wenn } x > 0 \end{cases} \quad (3.4)$$

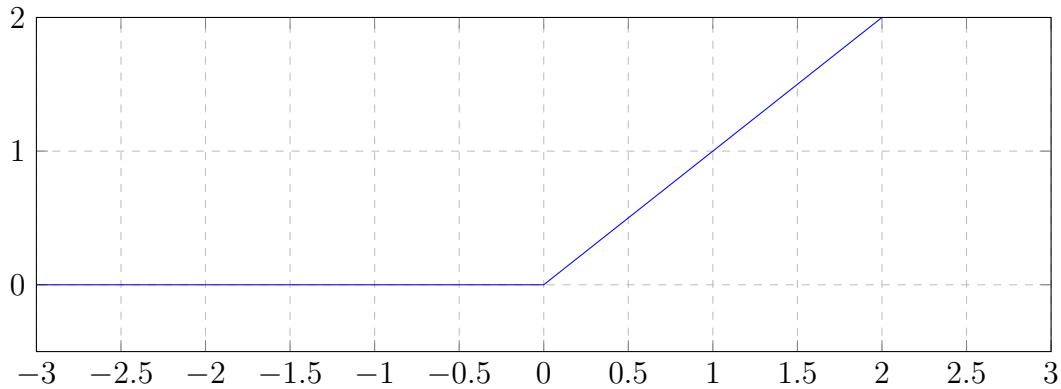


Abbildung 3.4: Graphische Darstellung der Aktivierungsfunktion der rectified linear unit (engl. für gleichgerichtete Lineareinheit) (vgl. Gleichung 3.4)

### 3.1.2 Training von Neuronalen Netzen

Um ein Neuronales Netz zu trainieren, benötigt man zuerst ein Maß für die aktuelle Leistung des Netzes. Mit der Leistung ist in diesem Fall gemeint, wie gut das Netz ein Eingangssignal korrekt klassifiziert. Hierfür wird im maschinellen Lernen eine Kostenfunktion genutzt:

$$C(w, b) = \frac{1}{2n} \sum_x \left| \overrightarrow{y(x)} - \overrightarrow{o(x)} \right|^2 \quad (3.5)$$

Hierbei sind  $x$  die Eingabewerte,  $o$  die von dem Neuronalen Netz aus den Eingabewerten abgeleiteten Ausgabewerte ( $o$  für Englisch *output*) und  $y(x)$  sind die theoretisch korrekten Ausgabewerte, die das Netz bestimmen soll, wenn es fertig gelernt hat. Bei den Ausgabewerten handelt es sich um Vektoren, deren Komponenten die Konfidenzen für verschiedene Ergebnisse widerspiegeln. Die Kostenfunktion ist somit ein Maß dafür, wie weit das tatsächliche Ergebnis einer Bestimmung durch ein Neuronales Netz von dem gewünschten Ergebnis entfernt ist. Um größere und damit schwerwiegender Abweichungen im Training höher zu gewichten, wird die Differenz zwischen  $y(x)$  und  $o(x)$  quadriert. Zu Beginn eines Trainings werden für alle Gewichte und biases zufällige Werte gewählt.

Während des Trainings werden die verschiedenen Parameter des Neuronalen Netzes durch ein stochastisches Gradientenverfahren [13, 14, 15] so angepasst, dass ein Minimum in der Kostenfunktion erreicht wird. Mit dieser Methode wird

ein lokales Minimum iterativ approximiert, indem die Werte in der Richtung des negativen Gradientenvektors bei abnehmender Schrittweite geändert werden. Diese Methode ermöglicht es aber nur ein lokales Minimum zu erreichen. Hierbei muss es sich nicht zwingend um das globale Minimum handeln. Es kann aus diesem Grund sinnvoll sein, einen Detektor mehrfach initial zu trainieren, da sich hierbei durch zufällig gewählte Startpunkte die Wahrscheinlichkeit erhöht, ein tieferes Minimum zu finden. Es ist wichtig, während des Trainings eine abnehmende Schrittweite zu verwenden. So kann einerseits ein dauerhaftes Überspringen des Minimums vermieden werden. Andererseits wird ein anfängliches Annähern an das Minimum mit hinreichend großen Schritten ermöglicht, um das Minimum zügig zu erreichen.

### 3.1.3 Beispiel: Erkennung handgeschriebener Ziffern

Ein klassisches Beispiel im Bereich des maschinellen Lernens ist die Erkennung von handschriftlich geschriebenen Ziffern. Hierfür existiert eine umfangreiche und mit guten Metadaten<sup>1</sup> versehene Datenbank Namens *MNIST* [16] (vgl. Abbildung 3.5)



Abbildung 3.5: Exemplarischer Auszug aus der *MNIST* Datenbank. Diese wird oft zum Trainieren von Netzen zur Erkennung von Ziffern genutzt. [11, 16]

Die Bilder in diesem Beispiel sind  $28 \times 28$  Pixel groß. Hieraus ergibt sich für die Eingangsebene eine Größe von  $(28 \times 28 =) 784$  Neuronen. Die Eingabewerte der Neuronen in der ersten Ebene entsprechen hierbei den Helligkeitswerten der entsprechenden Pixel. Die Ausgangsebene hat eine Größe von zehn Neuronen, da es - unter der Annahme, dass wir uns im Dezimalsystem befinden - zehn verschiedene Ziffern (0 bis 9) als Ergebnis gibt. In den Ausgabewerten spiegelt sich die Sicherheit (Konfidenz) wieder, mit der das Neuronale Netz 'denkt', dass es sich in dem Bild um die jeweilige Ziffer handelt. In diesem Beispiel wird eine versteckte Ebene mit 15 Neuronen genutzt (vgl. 3.6). Die Wahl von Größe und Anzahl der versteckten

<sup>1</sup>Bei den Metadaten handelt es sich in diesem Beispiel um die Werte der Zahlen, die handschriftlich dargestellt sind

Ebenen richtet sich nach der Komplexität des jeweiligen Problems und kann während der Entwicklung eines Detektors für dieses optimiert werden.

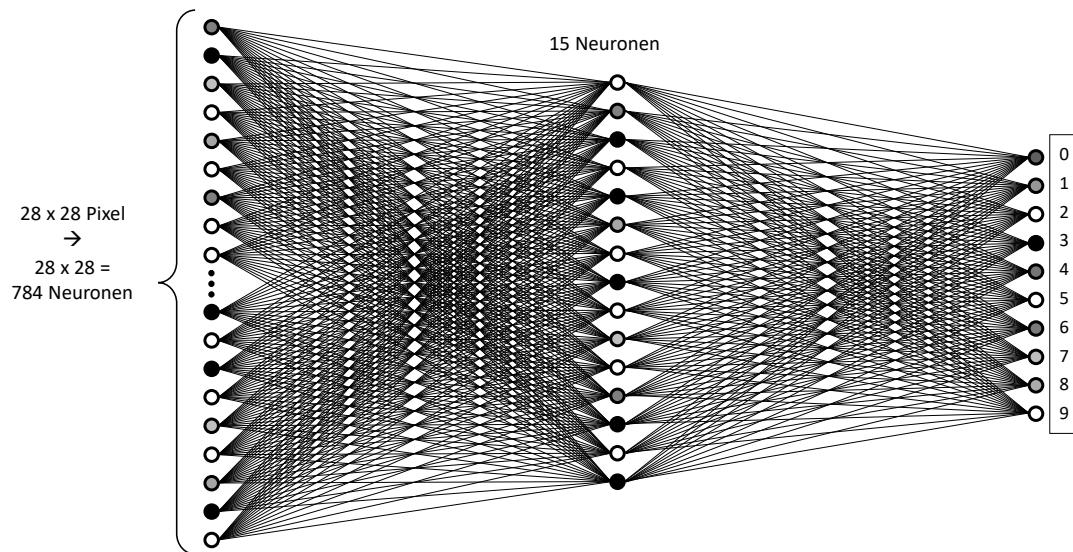


Abbildung 3.6: Schematische Darstellung eines Neuronalen Netzwerks zur automatischen Erkennung von Zahlen mit  $28 \times 28 = 784$  Eingangsneuronen, einer versteckten Ebene mit 15 Neuronen und 10 Ausgangsneuronen

Am Anfang des Trainings werden alle Gewichte und biases (engl. für Tendenzen) (wie in Abschnitt 3.1.2 beschrieben) zufällig gewählt. Zu diesem Zeitpunkt sind auch die Ausgabewerte komplett beliebig. Dies resultiert in einem hohen Wert für die Kostenfunktion. Dieser Wert wird im Laufe des Trainings minimiert und damit die Zuverlässigkeit des Detektors verbessert. Am Ende des Trainings ist der Detektor in der Lage die verschiedenen Ziffern korrekt zu erkennen. Ähnlich wie in diesem Beispiel verläuft jedes Training eines Objektdetektors.

## 3.2 Programmiersprachen

### 3.2.1 Python

Python ist eine objektorientierte Programmiersprache. Sie ist kostenfrei nutzbar und es gibt viele zusätzlich verfügbare application programming interfaces (engl. für Programmierschnittstellen) (APIs) wie beispielsweise *NumPy* (Bibliothek zum Verwalten von und Rechnen mit Vektoren und Matrizen) [17] und *TensorFlow* (Deep Learning Framework entwickelt vom Google Brain Team) [18]. Durch seine Syntax ist Python von Menschen gut lesbar und leicht verständlich. Diese Vorteile sorgen dafür, dass Python eine der am weitesten verbreiteten Programmiersprachen im Bereich des maschinellen Lernens ist. [2]

### 3.2.2 C++

C++ ist ebenfalls eine weit verbreitete Programmiersprache und genau wie Python objektorientiert. Sie ist eine Weiterentwicklung der Programmiersprache C und wie C sehr effizient und hardware-orientiert nutzbar. Aus diesem Grund wird sie für viele Computerspiele genutzt [19]. Auch die *Unreal Engine 4* (vgl. Abschnitt 3.3) ist in C++ programmiert. Aus diesem Grund habe ich neben Python in dieser Arbeit auch C++ benutzt. Um eine gute Leistungsfähigkeit bei C++ zu erreichen, müssen viele Einstellungen und Befehle expliziter und detaillierter programmiert werden. Es werden außerdem weniger Parameter von C++ selbstständig festgelegt. Variablentypen müssen beispielsweise vom Programmierer genau spezifiziert werden. Python legt diese selbstständig fest. Generell ist die Syntax von C++ komplexer. Aus diesem Grund gestaltet sich für mich das Programmieren in Python leichter als in C++. Deshalb nutze ich C++ nur an den Stellen, an denen ich an vorgefertigten C++ Scripten der *Unreal Engine 4* arbeite.

## 3.3 Unreal Engine 4

Die *Unreal Engine 4* ist eine Computerspiele-Engine von *EpicGames*. Computerspiele-Engines, auch Game-Engines genannt, sind ein Entwicklungsrahmen zum Designen, Zusammenstellen und Programmieren von Computerspielen. Teilweise werden diese von Entwicklern speziell für ein Spiel oder eine Spielereihe erstellt. Die Verwendung generischer Engines, wie der *Unreal Engine 4*, bietet jedoch den Vorteil, dass es sich dabei um vorgefertigte und bewährte Entwicklungsumgebungen handelt. Sie enthalten standardmäßig viele Funktionen, die in Spielen weit verbreitet sind und vereinfachen so die Entwicklung. Aus diesem Grund wird die *Unreal Engine 4* auch für kleinere Projekte häufig verwendet. Außerdem darf die *Unreal Engine 4* in vielen Bereichen kostenfrei genutzt werden und der gesamte Source-Code ist auf *GitHub*<sup>2</sup> verfügbar [20, 21]. Dies hat zur Folge, dass die Engine bei Bedarf einfach abgeändert werden kann.

Neben der Entwicklung von Spielen wird die *Unreal Engine 4* auch noch in vielen anderen Arbeitsfeldern wie zum Beispiel bei der Erstellung von Kinofilmen eingesetzt. Viele Unternehmen setzen sie auch in Forschungsprojekten wie in dieser Arbeit ein. Die Engine ist unter anderem in der Lage, selbst in Echtzeit hochqualitative Bilder zu erzeugen. Mit Hilfe von eigens dafür entwickelten Modulen in der Engine ist es außerdem möglich, mit mehr Zeitaufwand qualitativ noch höherwertige Bilder und Filme erzeugen, die teilweise nur noch sehr schwer von

---

<sup>2</sup><https://github.com/EpicGames/UnrealEngine>

der Realität zu unterscheiden sind. Beispielhaft für eine in der *Unreal Engine 4* gebaute Szene ist in Abbildung 3.7 die erste Szene zu sehen, die ich im Rahmen dieses Projektes erstellt habe.



Abbildung 3.7: Erste Szene, die ich mit der *Unreal Engine 4* für die Auswertung eines kamerabasierten Kopf-Schulter-Detektors auf Basis eines neuronalen Netzwerks erstellt habe.

### 3.3.1 Aufbau von Personenmodellen in Game Engines

In der *Unreal Engine 4* gibt es generell zwei Arten von Objekten. Diese beiden Objektarten unterscheiden sich darin, dass die eine Art sich in sich bewegen kann und die andere in sich statisch ist. Für die meisten Personenmodelle wird, wie auch in diesem Projekt, die bewegliche Art genutzt.

Bewegliche Modelle bestehen in der *Unreal Engine 4* aus mehreren Komponenten. Es gibt zum einen das *Skeletal Mesh*, ein Netz, das die Oberfläche des Objektes ergibt. Ein solches Netz gibt es in nicht beweglichen Objekten auch. Dort nennt es sich jedoch *Static Mesh*. Diese Netze können mit verschiedenen Materialien versehen werden, um ihre Oberfläche zu gestalten. Im Unterschied zu statischen Modellen verfügen bewegliche Modelle neben der Oberfläche auch noch über ein *Skelett*, wie es in Abbildung 3.8 zu sehen ist. Dieses besteht aus Knochen, die einzeln bewegt werden können. So ist es möglich, für Objekte wie beispielsweise Personen Animationen zu erstellen, die später im Spiel abgespielt werden können. Auf diese Weise können Personen zum Beispiel laufen [20]. Solche Animationen gibt es auch vorgefertigt im Internet zum Herunterladen auf Webseiten wie *Adobe Mixamo*[22].

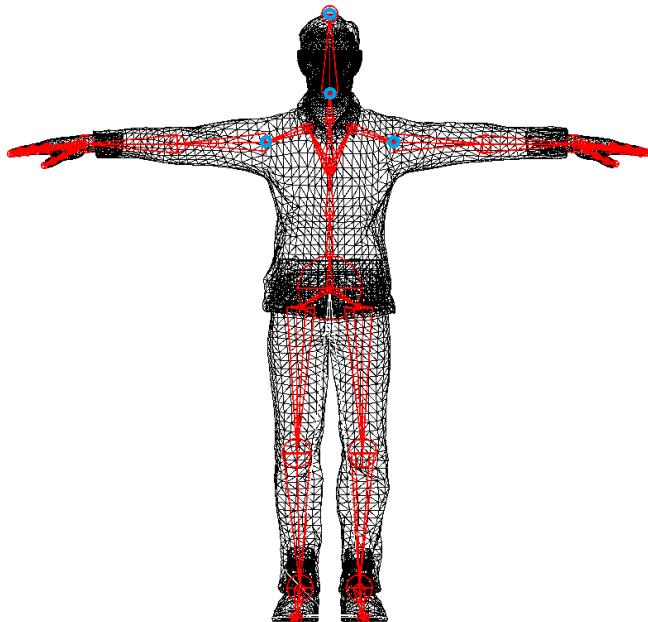


Abbildung 3.8: Skelett einer Person in der *Unreal Engine 4*. In schwarz ist das Netz zu sehen, das die Oberfläche der Person ergibt. In rot die Knochen, aus denen das Skelett besteht. Die blauen Kreise markierten die vier Punkte, deren Positionen in den Metadaten extrahiert werden.

### 3.3.2 UnrealCV

UnrealCV ist ein Plug-in für *Unreal Engine 4*, das von Weichao Qiu und Alan Yuille an der Johns Hopkins University Baltimore entwickelt wurde. Es ermöglicht die einfache Einbindung der *Unreal Engine 4* in Vorgänge im maschinellen Lernen. Zu den Funktionen gehören unter anderem die Positionierung einer Kamera im 3D-Raum sowie das Aufnehmen und Speichern von Bildern. Diese Funktionen werden über die ebenfalls zum Plug-in gehörende Python-API und einen lokalen Server gesteuert [23, 24, 25]. Die Nutzung des Plug-ins hat einige Vorteile, aber auch einige Nachteile gegenüber der in Kapitel 4.3 beschriebenen Methode auf Basis des *Sequenzers*. Für das Plug-in spricht, dass es komplett durch ein Python-Script nutzbar ist, sodass Kamerabewegungen, wie zum Beispiel Kreise um ein Objekt, einfach und exakt in diesem Script programmiert werden können. Darüber hinaus ist es möglich, die Rohdaten aus der Engine (vgl. Kapitel 4.4 und Kapitel 4.5) parallel zur Erstellung der Bilder im selben Skript umzurechnen. Dadurch ist es möglich, die Position der Kamera für die Umrechnung der Koordinaten (vgl. Kapitel 4.5) direkt zu übernehmen. Bei Verwendung des *Sequenzers* ist dies derzeit leider nicht möglich, sollte aber in Zukunft implementiert werden (vgl. Kapitel 6.2).

Für die Verwendung des *Sequenzers* (vgl. 4.3) und damit gegen das Plug-in spricht hingegen, dass es in dem Plug-in keine Einstellmöglichkeiten für die Kamera

wie die Bildrate, Brennweite, Blende und vieles mehr gibt. Außerdem ist UnrealCV langsamer als der *Sequenzer* und die Kamera wirft einen runden Schatten auf die *Welt*, der in den fertigen Bildern zu sehen ist (vgl. Abbildung 3.7 unten rechts).

Aufgrund der beschriebenen Vor- und Nachteile hängt es von der Art der zu erstellenden Sequenzen ab, welcher Weg zur Erstellung der Bilder gewählt werden sollte. Wegen der höheren Geschwindigkeit und der besseren Einstellmöglichkeiten sollte der *Sequenzer* genutzt werden, falls die Kamera sich in der Szene nicht bewegt. Für Szenen mit bewegter Kamera muss im Rahmen dieses Projektes das Plug-in genutzt werden, da nur damit die Kammerapositionen ausgelesen werden können.

In dieser Arbeit wird auf Grund der höheren Geschwindigkeit in den meisten Fällen mit dem *Sequenzer* gearbeitet. In Kapitel 6.1.2 wird wegen der benötigten Kamerabewegungen UnrealCV verwendet.

### 3.3.3 Vergleich der *Unreal Engine 4* mit anderen Rendering Engines

Neben der *Unreal Engine 4* gibt es noch andere Möglichkeiten, Bilder und Sequenzen zu erstellen und zu rendern. Bei Designern ist beispielsweise ein open-source Programm namens *Blender* relativ weit verbreitet. Im Gegensatz zu reinen Render-Engines hat die *Unreal Engine 4* einige Vor- und auch ein paar Nachteile, die ich am Beispiel von *Blender* erläutern möchte.

Programme wie *Blender* eignen sich bei komplexeren Geometrien erheblich besser zum Modellieren von Gegenständen und Charakteren als die *Unreal Engine 4*. Diese ermöglicht nur das Zusammensetzen von einfachen Geometrien oder importierten Objekten aus Programmen wie *Blender* in eine Szene, nicht jedoch das wirkliche Modellieren einzelner detaillierter Objekte. Außerdem verfügt *Blender* schon seit längerem über bessere Render-Verfahren wie zum Beispiel Ray-Tracing. Die *Unreal Engine 4* verfügt über diese Funktion erst seit sehr kurzer Zeit, sodass sie in dieser Arbeit noch nicht genutzt werden konnte. Dass diese Funktion erst jetzt in Computerspielen verwendet wird, liegt an einer anderen Eigenschaft der *Unreal Engine 4*. Sie ist so erstellt, dass es möglich ist, mit ihr in Echtzeit Szenen zu rendern. Dies ist in Computerspielen natürlich unumgänglich, da die Engine auf Eingaben des Spielers unmittelbar reagieren muss. Neben der Echtzeitfähigkeit bringt die *Unreal Engine 4* aber noch weitere Vorteile mit sich. So sind, wie bereits am Anfang des Kapitels erwähnt, einige Funktionen in der Engine vor-implementiert. Aufgrund der schnelleren Rendergeschwindigkeit und Faktoren wie der Möglichkeit zur Automatisierung von Personenmodellen (vgl. Kapitel 4.2.2) wird in dieser Arbeit mit der *Unreal Engine 4* gearbeitet.

# 4 Datengenerierung

Zum Generieren von künstlichen Daten sind mehrere Schritte erforderlich (vgl. Abbildung 4.1). Zuerst muss eine Szene erstellt werden (vgl. Abschnitt 4.1 und Abschnitt 4.2), von der im nächsten Schritt Bilder erzeugt werden (vgl. Abschnitt 4.3). Zeitgleich zu der Erzeugung der Bilder werden Metadaten zu den einzelnen Bildern gespeichert (vgl. Abschnitt 4.4). Bei diesen Metadaten handelt es sich primär um die Positionen von markanten Punkten des zu erkennenden Objektes, sogenannten Keypoints. Als letzter Schritt müssen die generierten Daten im einem gut nutzbaren Datenformat abgespeichert werden.

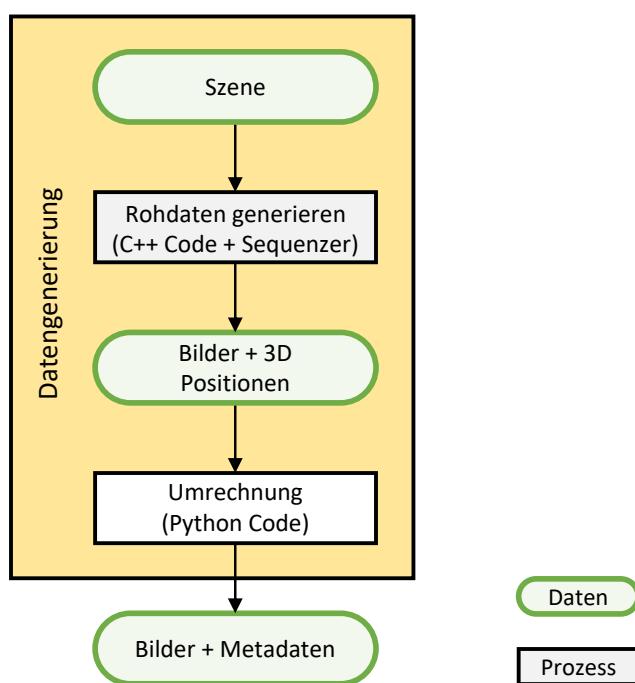


Abbildung 4.1: Arbeitsablauf zur Erstellung von Bildern und den entsprechenden Metadaten (Keypoints) mit der *Unreal Engine 4*

## 4.1 Modellierung der Szene

Die Szenen in dieser Arbeit wurden alle mit der *Unreal Engine 4* (vgl. Kapitel 3.3) erstellt.

Um Bilder und Keypoints zu generieren, werden drei grundlegende Dinge in einer

Szene benötigt. Zuerst braucht man einen geeigneten Hintergrund. Solche Hintergründe unterscheiden sich je nach Art des zu trainierenden Detektors stark im Aufbau und damit in der Komplexität. In Szenarien, wie sie etwa zum Training eines Birds Eye View (senkrecht von oben guckenden) Detektors verwendet werden ist ein planer Hintergrund (z.B. Boden) in der Regel ausreichend (vgl. Abbildung 4.2, links). Um einen Boden in der Szene zu erstellen, kann man zum Beispiel eine *Box* aus den vorgefertigten Geometrien in die Szene ziehen, diese auf die benötigte Größe skalieren und ein entsprechendes Bild als Textur wählen. Einige solche Texturen werden auch als '*Starter Content*' von Anfang an von der Engine zur Verfügung gestellt. Im Gegensatz dazu gibt es aber auch Situationen, wie in Abbildung 4.2 rechts zu sehen. Hierfür ist das Erstellen einer gesamten dreidimensionalen Szene erforderlich. Einfachere Geometrien können gut direkt in der *Unreal Engine 4* erstellt werden. Auch hier kann man sich der vorgefertigten Geometrien wie Quadern, Zylindern und Kugeln bedienen. Für diese Objekte kann man je nach Bedarf die Größe und die Textur ändern und sie in der Szene an der gewünschten Stelle platzieren. Auf diese Weise entstand beispielsweise der Großteil der in Abbildung 4.2 rechts zu sehenden Szene. Komplexere Objekte können entweder fertig aus dem Internet heruntergeladen werden, was auf Grund von Lizzenzen bei kommerziellen Anwendungen jedoch Probleme mit sich bringen kann, oder in anderen Grafikprogrammen wie *Blender*, *Autodesk Maya* oder ähnlichen erstellt werden und im Anschluss in die *Unreal Engine 4* importiert werden. Auf diesem Weg sind zum Beispiel die Computer, Stühle und Vorhänge in die oben genannte Szene gelangt.



Abbildung 4.2: Links: Bild aus einer künstlichen Trainingssequenz für Birds Eye View-Personen Detektoren. Rechts: digitaler Nachbau der realen Szene für das Training eines Personendetektors

Im Fall von komplexeren Szenen und speziell bei dem in Kapitel 2.1 beschriebenen Trainieren von Neuronalen Netzen, die an eine bestimmte Situation angepasst werden sollen, bietet es sich an, eine bestehende Szene digital nachzubauen. Ein Beispiel ist in Abbildung 4.3 zu sehen.

Zusätzlich zur Hintergrundszene benötigt man die zu erkennenden Objekte. In

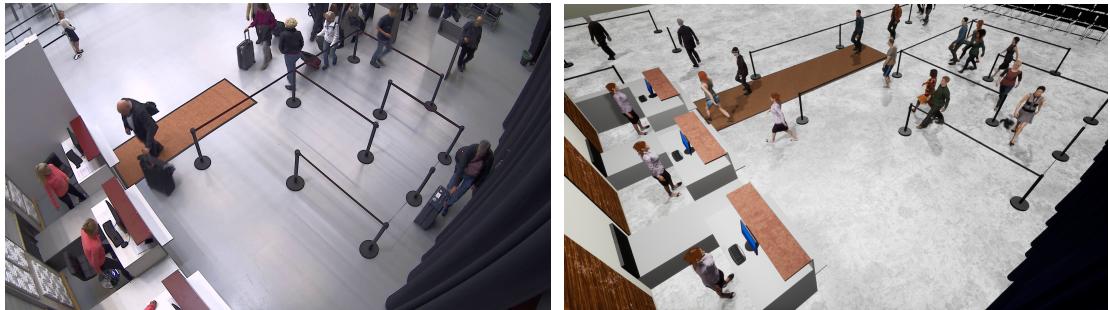


Abbildung 4.3: Links: Bild aus einer Trainingssequenz für den Einsatzort Flughafen. Diese Szene wurde von Mitarbeitern der Firma Bosch nachgestellt. Rechts: digitaler Nachbau der realen Szene als Grundlage zur künstlichen Datengenerierung

diesem Fall sind dies Personen. Falls erforderlich, können sie durch beliebige andere Objekte ersetzt werden, sofern genügend 3D-Modelle dieser Objekte vorhanden sind. Die Modelle müssen zwei Hauptanforderungen erfüllen. Zum einen müssen sie detailreich und realistisch genug sein, damit das neuronale Netzwerk nach dem Training reale Personen erkennen kann und nicht nur die künstlichen Modelle. Zum anderen müssen die Modelle mit einem Skelett versehen werden (vgl. Kapitel 3.3.1), da später aus diesem Skelett die Positionen der einzelnen Knochen ausgelesen werden.

## 4.2 Automatisierung der Szene

Die Personenmodelle werden anschließend in sogenannte Characters eingefügt. Characters sind in der Engine eine Art von Objekten, die in der *Welt* platziert oder automatisch generiert (gespawnt) werden können und von Spielern oder dem Spiel kontrolliert werden [25]. Sie haben außerdem die Fähigkeit, in der *Welt* herumzulaufen. In diesem Anwendungsfall sind sie so konfiguriert, dass sie in einem vorgegebenen an die Szene angepassten Bereich räumlich und zeitlich zufällig verteilt gespawnt werden und sich dann selbstständig zu einem anderen, vorher vorgegebenen Punkt bewegen. An diesem zweiten Punkt ist ein *Kill Volumen* platziert, welches dafür sorgt, dass die Characters wieder verschwinden (despawnen), sobald sie diesen Bereich in der *Welt* betreten. Dieses verhindert eine Überpopulation der Szene mit nicht mehr benötigten Characters. Auf dem Weg vom Spawnpunkt zum Despawnpunkt suchen sich die Characters automatisch den kürzesten Weg und umgehen selbstständig Hindernisse. Die Characters sind außerdem so modifiziert, dass sie, solange sie sich in der *Welt* befinden, dauerhaft ihre Position in eine Textdatei schreiben.

Weitere Informationen zur Programmierung dieser Characters, um die Keypoints

zu generieren, sind in Abschnitt 4.4 dieses Kapitels beschrieben.

### 4.2.1 Spawner

Abbildung 4.4 zeigt die grafische Programmierung (*Blueprints*) der hier verwendeten Spawner. In dem violetten Bereich befinden sich die Komponenten zur Steuerung einer anfänglichen Verzögerung. Auf diese Weise wird verhindert, dass zu Beginn des Spiels alle Characters gleichzeitig spawnnen. In rot sind die Bereiche zu sehen, die (hellrot) eine zufällige Position in einem bestimmten Bereich und (dunkelrot) eine zufällige Orientierung erzeugen, in der der Character spawnt. Die grün markierten Bausteine sind der eigentliche Spawner. Hier werden die Characters in der *Welt* platziert. Der blaue Bereich ist dafür verantwortlich, dass die Characters sich zu Ihrem Ziel in der Welt bewegen. Die gelb markierten Bausteine sorgen dafür, dass die Characters, nach einer festgelegten Zeit despawnen, auch wenn sie das Kill Volumen nicht erreichen sollten. Außerdem ist hier eine Schleife verbaut, die das Spawning eines Characters nach einer zufällig aus einem Bereich gewählten Zeit wiederholt.

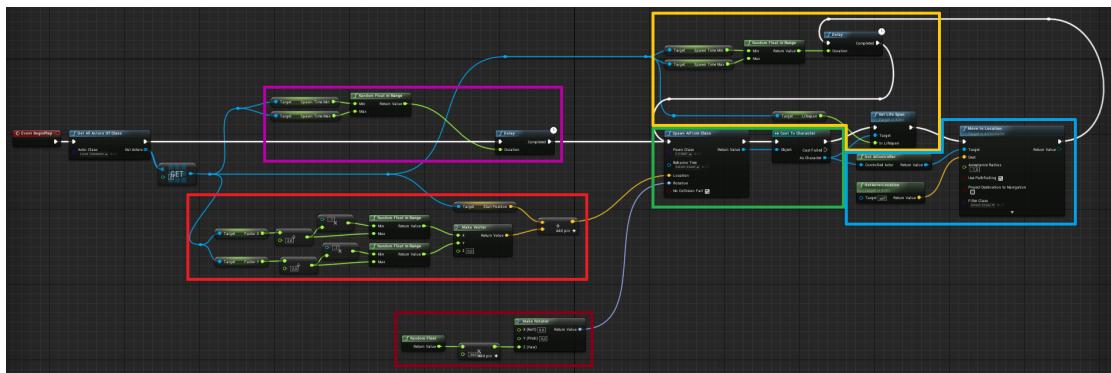


Abbildung 4.4: Graphische Programmierung des Spawners eines Characters. Eine genauere Beschreibung ist Abschnitt 4.2.1 zu entnehmen.

### 4.2.2 Navigation in der Szene

Die Komplexität der Navigation der Characters durch die *Welt* variiert je nach Szenentyp stark. Einerseits gibt es Szenen, die in der Modellierung simpler sind und in einer einfachen Draufsicht dargestellt werden. In diesem Fall werden die Characters auf einer Seite der *Welt* gespawnt und bewegen sich dann auf direktem Weg zu ihrem vorgegebenen Ziel auf der gegenüberliegenden Seite der *Welt*. Andererseits gibt es die komplexen dreidimensionalen Szenen, wie in Abbildung 4.3 dargestellt. Hier müssen sich die Characters auf den vorgesehenen Wegen durch die gebaute Szene bewegen. Dafür gibt es sogenannte *NavMesh's*. Dies sind unsichtbare

Quader, die in der *Welt* platziert werden können und innerhalb derer sich die Characters bewegen können. Die navigierbaren Bereiche werden von der *Unreal Engine 4* bei Bedarf grün dargestellt [25]. Dies ist für die gerade beschriebene Szene in Abbildung 4.5 zu sehen. In allen Szenen befinden sich sowohl der Spawn- als auch der Despawnpunkt außerhalb des Blickfelds der Kamera, da es sonst zu Problemen bei der Generierung der Keypoints kommt. Diese werden in Abschnitt 4.4 näher beschrieben.

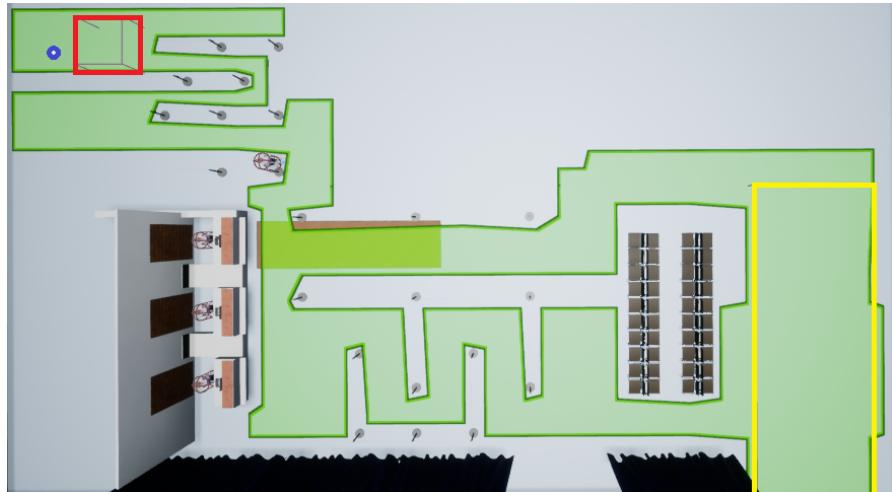


Abbildung 4.5: Draufsicht der während des Projektes nachgebauten Szene aus Abbildung 2.2, Abbildung 4.2 und Abbildung 4.3. In gelb zu sehen ist die Spawn Area, in rot das Kill Volumen und in blau das Ziel der Characters. In dem grünen Bereich können sich die Characters bewegen.

### 4.3 Rendern der Bilder

Zum Rendern der Bilder habe ich meistens den *Sequenzer* genutzt (vgl. Kapitel 3.3.2). Hierbei handelt es sich um einen Teil der *Unreal Engine 4*, der auf Basis der in der *Welt* platzierten Kameras Bilder und Videos generiert [20]. Er verfügt über einen großen Funktionsumfang. Unter anderem ist es möglich, viele Parameter wie Licht- und Kameraeinstellungen oder die Positionen von Gegenständen durch *Keyframes* zu verschiedenen Zeiten im späteren Video unterschiedlich einzustellen. Einige Einstellungen müssen verändert werden, um den *Sequenzer* in dieser Arbeit optimal zu nutzen. Funktionen wie Tiefen- und Bewegungsunschärfe sind zum Beispiel genauso unerwünscht wie eine automatische Belichtungskorrektur seitens der Engine und sollten aus diesem Grund abgestellt werden. Auch die Bildrate, Länge, der Speicherort und das Format der generierten Daten können umgestellt werden. Im diesem Projekt werden Serien von Bildern im *.png*-Format mit  $1920 \times 1080$  Pixeln

(*Full-HD*) erzeugt. Die Bildfrequenz wird auf ein Bild alle ein bis vier Sekunden gesetzt, damit sich die erzeugten Bilder nicht zu sehr gleichen und so mehr Variation in der Datenbank vorhanden ist. Benannt werden die Bilder mit 'export \_Frame', wobei *Frame* eine fortlaufender Bildnummer ist. Pro Szene werden zwischen 500 und 1000 Bilder generiert. Diese Anzahl ist frei wählbar. Die Zeit, die zum Erstellen der Bilder benötigt wird, hängt maßgeblich von dem Computer ab, auf dem die *Unreal Engine 4* läuft. Mit dem in dieser Arbeit verwendeten PC (Intel® Xeon® E3-1270, NVIDIA Quadro® P2000, 32 GB RAM, SATA SSD) dauert es etwa fünf Minuten, um 1000 Bilder zu generieren.

## 4.4 Keypoint Generierung

Zur weiteren Verarbeitung der Daten und für das Training der Neuronalen Netze (vgl. Kapitel 3.1.2) werden Metadaten benötigt. In diesem Fall handelt es sich um die Positionen der linken und rechten Schulter sowie dem oberen und unteren Ende des Kopfes der Personen in den Bildern (vgl. Abbildung 3.8). Um diese Daten zu erhalten, war die erste Idee, das in Kapitel 3.3.2 beschriebene Plug-in *UnrealCV* zu nutzen. Es stellte sich jedoch heraus, dass diese Funktion für das Plug-in zwar geplant aber zur Zeit noch nicht verfügbar ist.

Aus diesem Grund habe ich die Erstellung der Metadaten direkt in der *Unreal Engine 4* umgesetzt. Hierfür habe ich einen neuen *C++ Character* erstellt. Für jeden Character wird von der Engine automatisch ein Script (*C++ oder Blueprints*<sup>1</sup>) angelegt, in dem festgelegt werden kann, wie sich der Character verhält und welche Prozesse im Hintergrund durch diesen Character ausgelöst werden. In diesem Script gibt es zwei *C++*-Funktionen. Die Funktion *BeginPlay* wird einmal zu Beginn des Spieles ausgeführt, um festzulegen was passiert, wenn der Spieler das Spiel startet. Die Funktion *Tick* wird zu jedem gerenderten Bild des Spieles wieder neu ausgeführt. Hier können Prozesse umgesetzt werden, die während des gesamten Spiels wiederholt werden sollen wie etwa das Aktualisieren einer Lebensanzeige [20]. In diese zweite Funktion habe ich auch die Ausgabe der Metadaten implementiert.

Die Ausgabe der Metainformationen erfolgt mit der Funktion *Tick*:

```

1 void Alocation_output_character::Tick( float DeltaTime )
2 {
3     Super::Tick( DeltaTime );
4
5     FRenderCommandFence 1_fence;
6

```

<sup>1</sup> *Blueprints* ist die graphische Programmiersprache der *Unreal Engine 4*.

```

7     l_fence.BeginFence();
8
9     FString storage_location = "C:/Users/<user>/Documents/
10                  UE4_Data_Generation/Saved/VideoCaptures/" ;
11
12
13     filename = filename + FString::FromInt(GFrameNumber)+ ".txt" ;
14
15     FString Name = UObjectBaseUtility::GetName();
16
17     FVector l_bonePos = GetMesh()->GetBoneLocation("LeftArm",
18                                         EBoneSpaces::WorldSpace);
19     FVector r_bonePos = GetMesh()->GetBoneLocation("RightArm",
20                                         EBoneSpaces::WorldSpace);
21     FVector t_bonePos = GetMesh()->GetBoneLocation("HeadTop_End",
22                                         EBoneSpaces::WorldSpace);
23     FVector b_bonePos = GetMesh()->GetBoneLocation("Head",
24                                         EBoneSpaces::WorldSpace);
25
26     std::ofstream myfile;
27     myfile.open(TCHAR_TO_ANSI(*filename), std::ios_base::app);
28
29     myfile <<
30     "filename: " << TCHAR_TO_ANSI(*filename) <<
31     "\n charactername: " << TCHAR_TO_ANSI(*Name) <<
32     "\n Annotationtype: Keypoints" <<
33     "\n CoordinateSystem: WorldSpace" <<
34     "\n Coordinates: " <<
35     "\n LeftShoulder " << TCHAR_TO_ANSI(*l_bonePos.ToString()) <<
36     "\n RightShoulder" << TCHAR_TO_ANSI(*r_bonePos.ToString()) <<
37     "\n HeadTop " << TCHAR_TO_ANSI(*t_bonePos.ToString()) <<
38     "\n HeadBottom " << TCHAR_TO_ANSI(*b_bonePos.ToString()) <<
39     "\n" ;
40
41     l_fence.Wait();
42 }
```

In Zeile 5 und 7 wird ein *Render Fence* angefangen. Dieser bewirkt, dass das Script am Ende (Zeile 41) wartet, bis der *Sequenzer* ein Bild gerendert und gespeichert hat, bevor es wiederholt wird. So wird sichergestellt, dass das Spiel und der *Sequenzer* mit der gleichen Bildrate laufen und pro Bild nur eine Datei mit Annotationen erzeugt wird. In Zeile 9 wird festgelegt, wo die Annotationsdateien gespeichert werden. Sobald die *Unreal Engine 4* gestartet wird, fängt sie im Hintergrund an zu zählen, wie viele Bilder sie schon angezeigt hat. Dieser Zähler wird in Zeile 13

ausgelesen, um diese Zahl in Zeile 27 mit in den Namen der Textdatei zu schreiben. Alle Characters in der *Unreal Engine 4* haben einen Namen. Falls ein Character mehrfach in einer Welt gespawnt wird, so wird eine laufende Nummer an den Namen des Characters angehängt. Diese Namen werden in Zeile 15 ausgelesen. In Zeile 17 bis 24 werden die Positionen der Knochen 'LeftArm', 'RightArm', 'HeadTop\_End' und 'Head' (vgl. Abbildung 3.8) ausgelesen. Diese Positionen werden in dem globalen Koordinatensystem der *Unreal Engine 4* ausgegeben und müssen in einem späteren Schritt (vgl. Abschnitt 4.5) in Bildkoordinaten umgerechnet werden. In den Zeilen 26 und 27 wird ein Textdokument erstellt, in das in den Zeilen 29 bis 39 der Name des Characters, einige Informationen über die Szene sowie die Koordinaten der vier Keypoints geschrieben werden.

Von dem oben beschriebenen *C++ Character* werden in der Engine '*Blueprints*<sup>2</sup> *ChildCharacters*' (Englisch für Kinder-Charaktere) erstellt. Diese Kinder basieren auf dem gleichen Script wie der *C++ Character* und erben somit die Funktionen von ihm. Sie schreiben also ebenfalls ihre Position in das dementsprechende Textdokument. Die *ChildCharacters* werden dafür genutzt, die einzelnen Personen zu erstellen. Weitere Eigenschaften wie das Aussehen und die Animationen werden erst diesen *ChildCharacters* gegeben. Für jeden verschiedenen Menschen gibt es also jeweils einen Character mit dem dementsprechenden *Skeletal Mesh* und einer dazu passenden Animation. Diese Characters werden dann später in der *Welt* gespawnt. Das Arbeiten mit *ChildCharacters* hat den wesentlichen Vorteil, dass man nur ein weiteres Kind des bestehenden *C++ Characters* erstellen muss, wenn man eine neue Person einfügen möchte. Änderungen an dem Eltern Character werden für alle seine Kinder übernommen. So muss man nur den originalen *C++ Character* ändern, falls man etwas an der Programmierung aller *ChildCharacters* ändern möchte wie zum Beispiel den Namen oder Speicherort der Textdokumente.



Abbildung 4.6: Beispielhaft vier Characters im Editor der *Unreal Engine 4*

<sup>2</sup> *Blueprints* ist die graphische Programmiersprache der *Unreal Engine 4*.

Da das Script zum Speichern der Metadaten für einen *Character* immer dann ausgeführt wird, wenn der Character sich zu Beginn des Frames in der Welt befindet, wird die Position des Characters in dem Frame in dem der Character spawnt noch nicht erfasst. Der Character ist hier jedoch schon im Bild zu sehen. Andersherum wird in dem Frame, in dem der Character despawnt seine Position noch gespeichert, obwohl der Character nicht mehr sichtbar ist, da das entsprechende Script zu Beginn des Frames noch aktiv war. In Frames, in denen ein neuer Character spawnt oder ein existierender despawnt, ist hierdurch entweder ein Character zu sehen, für den noch keine Annotationen existieren, oder es existieren Keypoints, für die es keinen zugehörigen Character mehr im Bild gibt. Wie in Abschnitt 4.2.2 kurz angesprochen, müssen die Characters sich aus diesem Grund sowohl beim Spawning als auch beim Despawning außerhalb des Blickfeldes der Kamera befinden.

## 4.5 Transformierung der Keypoint-Koordinaten aus dem Welt-Koordinatensystem ins Bild-Koordinatensystem

Die in Abschnitt 4.4 erstellten Positionen sind in dem globalen dreidimensionalem Koordinatensystem der künstlich generierten *Welt* angegeben. Um diese Informationen zielführend zu nutzen, werden die globalen Koordinaten in zweidimensionale Positionen in dem Kamerabild umgewandelt.

Zuerst verschiebe ich das Koordinatensystem so, dass die Kamera im Ursprung liegt. Hierfür wird die Position der Kamera ( $P_K$ ) von der Position des Objektes ( $P_O$ ) abgezogen. Hieraus ergibt sich die verschobene Position des Objektes ( $P_v$ ).

$$\vec{P}_v = \vec{P}_O - \vec{P}_K \quad (4.1)$$

Nun drehe ich das Koordinatensystem so, dass seine X–Achse in die Blickrichtung der Kamera zeigt. Dafür wird das Koordinatensystem zuerst um den Gierwinkel ( $\alpha$ ) um die Z–Achse (vgl. Abbildung 4.7) gedreht. Im Anschluss wird es um den Nickwinkel ( $\beta$ ) um die Y–Achse gedreht (vgl. Abbildung 4.7). Um diese Drehungen zu vollziehen, werden die  $P_v$ s zuerst mit der Drehmatrix zur Drehung um die Z Achse ( $D_z$ ) und dann mit der Drehmatrix zur Drehung um die Y Achse ( $D_y$ ) multipliziert (vgl. Gleichungen 4.2, 4.3 und 4.4). So erhält man die transformierte Position des Objektes ( $P_t$ ) [26]. Die Verschiebung aus Gleichung 4.1 und 4.4 können in einer Matrix zusammengefasst werden. Sie werden an dieser Stelle jedoch aus Gründen der Übersichtlichkeit und besseren Nachvollziehbarkeit nacheinander

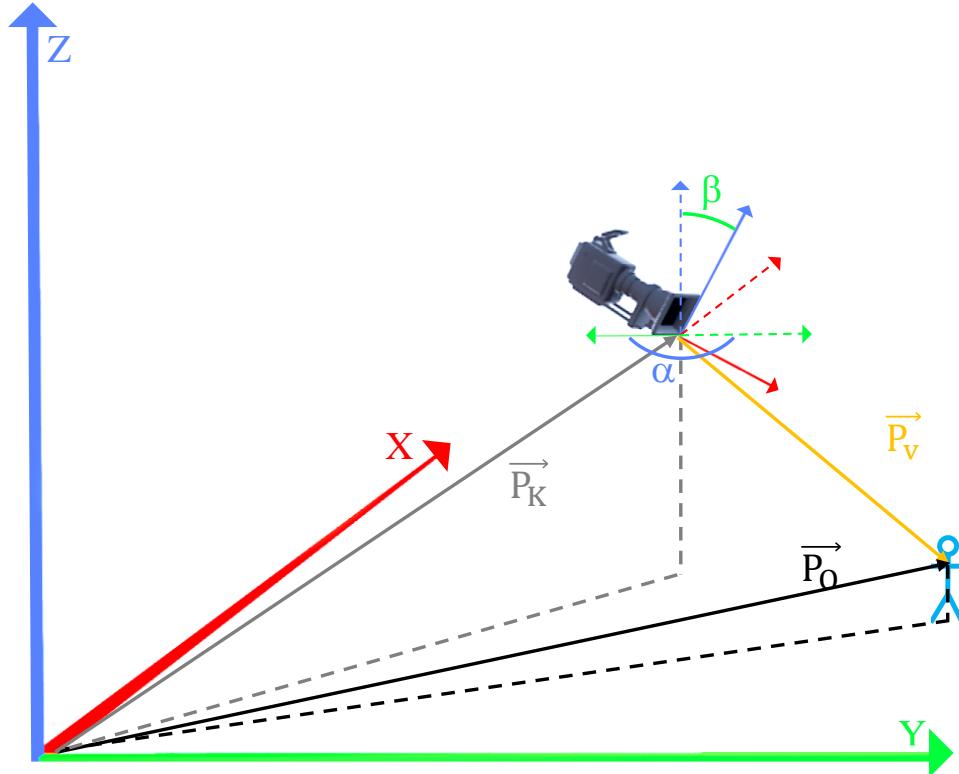


Abbildung 4.7: Koordinatensystem der Kamera in dem *Welt*-Koordinatensystem der *Unreal Engine 4*. Zu sehen ist das *Welt*-Koordinatensystem einmal in groß und einmal in klein und gestrichelt an der Kamera, sowie das Koordinatensystem der Kamera mit durchgezogenen Linien an der Kamera.

ausgeführt.

$$D_z = \begin{pmatrix} \cos \alpha & \sin \alpha & 0 \\ -\sin \alpha & \cos \alpha & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad (4.2)$$

$$D_y = \begin{pmatrix} \cos \beta & 0 & \sin \beta \\ 0 & 1 & 0 \\ -\sin \beta & 0 & \cos \beta \end{pmatrix} \quad (4.3)$$

$$\vec{P}_t = D_y \cdot D_z \cdot \vec{P}_v \quad (4.4)$$

Nun müssen die Koordinaten noch perspektivisch korrigiert werden. Das bedeutet, dass Objekte, die weiter von der Kamera entfernt sind, kleiner dargestellt werden und dadurch näher an der Bildmitte zu sehen sind (vgl. Abbildung 4.9). Um dies zu erreichen, werden die  $Y$  – und die  $Z$  – Komponenten der transformierten Positionen der Objekte durch die  $X$  – Komponente geteilt.

Die virtuelle Kamera der *Unreal Engine 4* simuliert einen Vollformatsensor mit einer Abmessung von  $36 \times 24 \text{ mm}$ . Bei der verwendeten Brennweite von  $18 \text{ mm}$  ergibt sich hieraus ein horizontaler Öffnungswinkel von  $90^\circ$  (vgl. Abbildung 4.8).

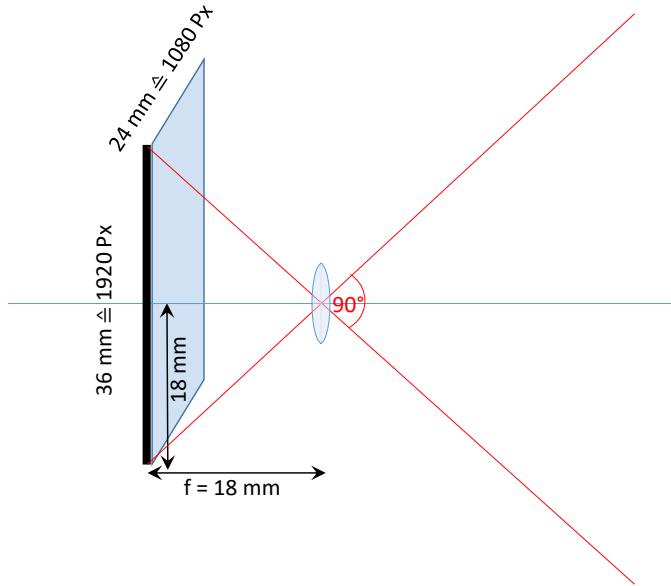


Abbildung 4.8: Schematischer Aufbau einer Kamera mit Linse und Sensor zur Veranschaulichung des Blickfeldes. Die Kamera wird von oben gezeigt. Man schaut somit auf die lange Seite des Sensors. Durch die Brennweite von  $18 \text{ mm}$  und die Sensorbreite von  $36 \text{ mm} = 2 \times 18 \text{ mm}$  ergibt sich der horizontale Öffnungswinkel von  $90^\circ$ .

Da die Bilder in *FULL-HD* Auflösung ( $1920 \times 1080$  Pixel) erstellt werden (vgl. Abschnitt 4.3), müssen die Bildkoordinaten der Objekte noch mit einem Faktor ( $1920/2 =$ ) 960 multipliziert werden. Alle Objekte, die unter demselben Winkel zu sehen sind, werden auf dieselbe Position auf dem Sensor abgebildet. Aus dem Strahlensatz ergibt sich dann, dass die Kamerakoordinaten ermittelt werden können, indem die Objektkoordinaten mit der Entfernung, also der X-Koordinate der Objekte skaliert werden (vgl. Abbildung 4.9). Da die Pixel quadratisch sind, ist der benötigte Skalierungsfaktor in X- und Y-Richtung gleich.

Der Koordinatenursprung eines Bildes in Python liegt meistens in der oberen, linken Ecke des Bildes. Aus diesem Grund müssen die Koordinaten der Annotationen nun noch um  $(1080/2 = ) 540$  Pixel nach oben und um  $(1920/2 = ) 960$  Pixel nach links verschoben werden. Diese Schritte werden in Gleichung 4.5 zusammengefasst.

$$\vec{P}_I = \frac{\vec{P}_t}{\vec{P}_{tx}} \cdot 1920/2 + \begin{pmatrix} 0 \\ 1920/2 \\ 1080/2 \end{pmatrix} \quad (4.5)$$

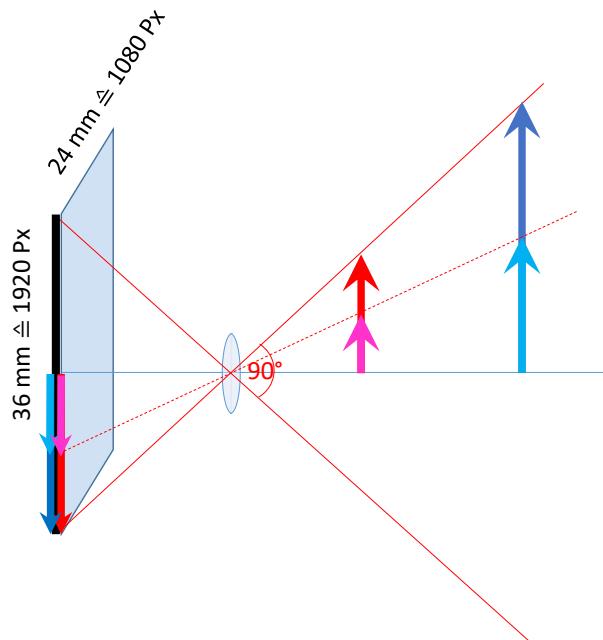


Abbildung 4.9: Veranschaulichung der Projektion von Objekten in verschiedenen Distanzen auf den Sensor einer Kamera. Zu erkennen ist, dass weiter entfernte Objekte kleiner abgebildet werden als Objekte, die sich näher an der Kamera befinden.

Die Y–Komponente des Vektors  $\vec{P}_I$  entspricht der X–Komponente der Koordinaten im Bild und die Z–Komponente gibt die Y–Bildkoordinate wieder.

Um die Weiterverarbeitung der Daten zu vereinfachen, werden die Daten in *XML*-Dateien gespeichert. XML-Dateien haben den Vorteil, dass sie in einer Baumstruktur aufgebaut sind und es dadurch einfacher ist, automatisiert Informationen daraus zu extrahieren. Diese sind wie folgt aufgebaut:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<root>
  <id name="F_P0005_C_7">
    <Keypoints>
      <KeyPoint Y="154" X="861" Name="LeftShoulder"/>
      <KeyPoint Y="152" X="822" Name="RightShoulder"/>
      <KeyPoint Y="115" X="838" Name="HeadTop"/>
      <KeyPoint Y="138" X="841" Name="HeadBottom"/>
    </Keypoints>
    <LeavingImageBoundaries LeavingImageBoundaries="False"/>
    <AnnotationType AnnotationType="groundtruth"/>
  </id>
</root>
```

Die in diesem Abschnitt beschriebene Umrechnung habe ich in einem Python-Script umgesetzt. Dies ist im Anhang dieser Arbeit zu finden.

# 5 Training und Evaluierung von Objektdetektoren

Um zu überprüfen, ob es möglich ist, mit künstlichen Daten einen Detektor zu trainieren, ist der einfachste Weg, einen Detektor zu trainieren und mit anderen Detektoren zu vergleichen. Die Detektoren, die für diese Vergleiche genutzt werden, müssen unter den gleichen Bedingungen trainiert und ausgewertet werden wie der zu testende Detektor. Zum Training werden jedoch Echtweltdaten genutzt. Ein Detektor, der mit künstlichen Daten trainiert ist, sollte in der Lage sein, in signifikantem Maß Objekte zu erkennen. Besonders positiv wäre es, wenn er dies besser kann als die Vergleichsdetektoren.

## 5.1 Training der Detektoren

Für das Training habe ich zuerst zwischen 4.000 und 10.000 Bilder von einer Szene generiert. Um das Training effektiver zu gestalten und den Detektor allgemeiner nutzbar zu machen, beinhaltet die so erstellte Datenbank je nach Art des Detektors verschiedene Blickwinkel oder Hintergründe.

Zum Trainieren der Detektoren wurde eine firmeninterne Toolbox genutzt. Sie basiert auf dem, von *Google Brain* entwickelten Framework *TensorFlow*, welches das eigentliche Training übernimmt [18]. Außerdem beinhaltet die Toolbox Funktionen zur Verarbeitung, Aufbereitung und Anzeige der Trainingsdaten und zur Evaluierung der fertigen Detektoren. Durch die Verwendung von Methoden wie *False Prediction Mining*<sup>1</sup>, die ebenfalls in der Toolbox vorhanden sind, ist es möglich, die Detektoren weiter zu verbessern.

Die Detektoren, die in dieser Arbeit erstellt werden, werden patchbasiert und auf mehreren Skalen trainiert. Das bedeutet, dass das Neuronale Netz nicht ein ganzes Bild auf einmal analysiert und bestimmt, wo sich Objekte befinden, sondern schrittweise kleine Teile des Bildes (Patches) untersucht und bestimmt, ob in dem

---

<sup>1</sup>*False Prediction Mining* ist ein Verfahren, bei dem nach jeder Trainingsiteration eine von der Trainingsdatenbank unabhängige Testdatenbank von dem Detektor analysiert wird und alle Bilder mit Fehldetections (Fehlauslösungen/Nichtauslösungen) in die Trainingsdatenbank eingefügt werden. So wird die Datenbank mit Bildern angereichert, die der Detektor noch nicht zuverlässig klassifizieren kann.

betrachteten Bereich ein gesuchtes Objekt vorhanden ist. Mit diesem Verfahren wird das gesamte Bild schrittweise abgerastert und auf Objekte untersucht. Dieser Prozess wird in einer Bildpyramide (vgl. Abbildung 5.1) mit verschiedenen großen Bildausschnitten wiederholt, um Objekte in verschiedenen Größen finden zu können [27].

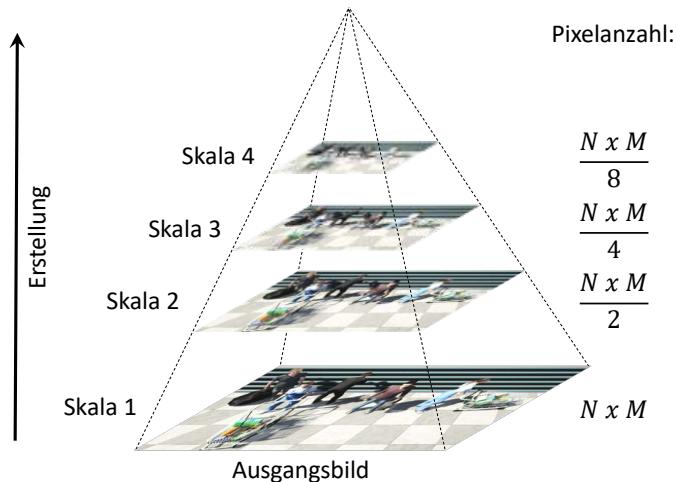


Abbildung 5.1: Bildpyramide mit vier Skalenstufen. Die Bilder in der Pyramide zeigen alle den gleichen Bildausschnitt jedoch mit verschiedenen Auflösungen. Dies hat zur Folge, dass der Detektor, der eine feste Pixelanzahl als Eingangsgrößen hat, einen größeren Ausschnitt des Bildes sieht, je geringer die Auflösung des Bildes ist.

Für das Training muss auch der Groundtruth (die Metadaten zu dem jeweiligen Bild) in Patches unterteilt werden, damit für jedes erstellte Bildpatch festgelegt ist, ob sich in dem Patch ein Objekt befindet oder nicht. Diese Daten soll das Neuronale Netz im Training lernen.

In Abbildung 5.2 sind diese positiven Patches mit blauen Boxen markiert.

In dieser Abteilung werden Detektoren mit besonders wenigen Koeffizienten und damit besonders kleine Neuronale Netze entwickelt. Dies liegt daran, dass die Detektoren als *Embedded Detectors* (engl. für eingebettete Detektoren) entwickelt sind und auf Chips eingesetzt werden, die nur über eine sehr begrenzte Rechenleistung verfügen und direkt in Kameras verbaut werden. Durch diesen Umstand sind die Detektoren, die mit diesen Netzen realisiert werden können, nicht so leistungsstark wie Detektoren, die auf wesentlich größeren Netzen basieren. Auf die Ergebnisse dieser Arbeit sollte dieser Umstand jedoch keine wesentlichen Auswirkungen haben, da hier nur gleichgroße Netze miteinander verglichen werden außer in Abbildung 6.4, wo die Auswirkung verschiedener Netzgrößen untersucht werden.

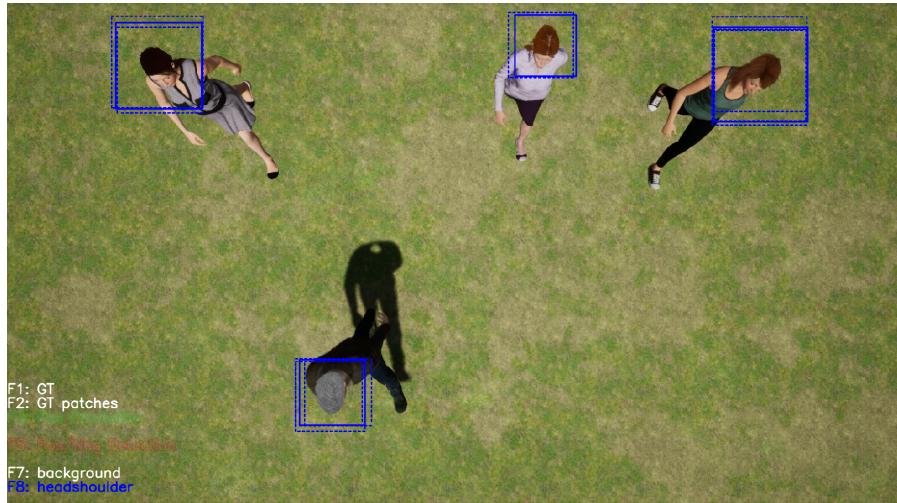


Abbildung 5.2: synthetisches Trainingsbild für einen Kopf-Schulter-Detektor mit Rahmen um die positiven Patches.

## 5.2 Evaluierung der Detektoren

Um eine aussagekräftige Evaluierung durchführen zu können, müssen die Detektionen zuerst mit einem *Postprocessing* (engl. für Nachbehandlung) bearbeitet werden. Hierbei werden die Detektionen von den verschiedenen Skalen zu einer Detektion kombiniert. Im Anschluss werden die Detektionen mit dem *Groundtruth*, also den Annotationsdaten, die im Voraus erstellt wurden und die sicher korrekt sind, verglichen und es wird überprüft, wie viele Objekte richtig erkannt wurden (*True Positives*), wie viele Objekte nicht erkannt wurden (*False Negatives*) und wie oft Auslösungen an Stellen vorkamen, an denen kein zu detektierendes Objekt war (*False Positives*). Diese Werte werden anschließend als *ROC-Kurve* (kurz für Receiver-Operating Characteristic [28]) aufgetragen. Dafür wird die *True Positive Rate* (erkannte Objekte im Verhältnis zu vorhandenen Objekten) über der Anzahl an *False Positives* pro Bild, für eine möglichst große Anzahl an Bildern gemittelt, in ein Diagramm eingezeichnet. Da die Detektoren keine klaren Ergebnisse liefern, sondern Wahrscheinlichkeiten, mit denen sich in einem Patch ein Objekt befindet, muss ein Schwellenwert (*Threshold*) festgelegt werden, ab dem etwas als Detektion gewertet wird. Um eine *ROC-Kurve* zu erzeugen, werden *True Positive Rate* und die *False Positives* pro Bild für einen gesamten, vom Training unabhängigen Datensatz ermittelt und mit dem Schwellenwerten als Kurvenparameter aufgetragen. Da die Anzahl an *True Positives* in einem Bild von der Anzahl an Objekten in dem jeweiligen Bild abhängt, kann dieser Wert nicht absolut betrachtet werden. Stattdessen wird er als Rate angegeben. Die Anzahl an negativen Patches ist im Verhältnis zu der Anzahl an positiven Patches so groß, dass die Anzahl an Objekten auf sie nur einen sehr geringen Einfluss hat. Aus diesem Grund kann sie bei gleichbleibender

Bildgröße als konstant angenommen werden. Die Anzahl an *False Positives* kann aus diesem Grund absolut betrachtet werden und muss nicht mit der Anzahl der negativen Patches ins Verhältnis gesetzt werden. Im Allgemeinen gilt: ein Detektor ist umso besser, je höher die *True Positive Rate* bei gleicher Anzahl an *False Positives* pro Bild ist, also je höher die *ROC-Kurve*. Die so entstehenden Kurven sind in Kapitel 6.1 zu sehen. Sie werden genutzt, um verschiedene Detektoren aussagekräftig miteinander vergleichen zu können.

# 6 Ergebnisse, Fazit und Ausblick

## 6.1 Ergebnisse

Wie in Kapitel 2 beschrieben, sollen die künstlichen Daten zum Training wie auch zur Evaluierung von Objektdetektoren eingesetzt werden. Das Trainieren und Testen der Detektoren wurde in Zusammenarbeit mit Kollegen in meiner Abteilung durchgeführt.

### 6.1.1 Training mit künstlichen Daten

Zuerst wurden Detektoren für zwei Kopf-Schulter-Detektoren ausschließlich mit künstlichen Daten trainiert. Mit diesen Detektoren wurden anschließend auf verschiedenen Datenbanken ROC-Kurven erstellt, um zu überprüfen, ob es generell möglich ist, mit künstlichen Daten funktionierende Detektoren zu erstellen.

#### ***Birds-Eye-View* Personendetektor**

Bei dem ersten Detektor handelt es sich um einen *Birds-Eye-View* Detektor. Das heißt, dass er die Anwesenheit von Personen erkennen soll, wenn die Kamera sich direkt darüber befindet. Für solche Blickrichtungen sind im Internet nur wenige Trainingsdaten vorhanden. Der hier genutzte Detektor wurde auf etwa 7000 synthetischen Bildern trainiert. Ausgewertet wurde der Detektor auf Basis von drei Datenbanken. Eine davon (*synthetic*) beinhaltete genau wie die Trainingsdaten künstlichen Daten und umfasste 1901 Bilder (vgl. Abbildung 6.2, links). Diese ist der Trainingsmenge zwar ähnlich, aber es handelt sich trotzdem um Bilder, die unabhängig von der Trainingsmenge erzeugt wurden. Das Ergebnis hierfür ist in Abbildung 6.1 in grün eingezzeichnet. Die zweite Datenbank (*real-training*) beinhaltet 519 Bilder aus der echten Welt (vgl. Abbildung 6.2, Mitte). Es handelt sich um dieselbe Datenbank, mit der auch der Echtwelt-Detektor in Abbildung 6.3 trainiert wurde. Bei der dritten Datenbank (*real-test*) handelt es sich um 10.000 Echtweltbilder (vgl. Abbildung 6.2, rechts). Diese Bilder dienten bei der Erstellung der künstlichen Daten als grober Anhaltspunkt für Parameter wie das Blickfeld der Kamera. Die ROC-Kurve für diese Datenbank ist in Abbildung 6.1 in rot eingezzeichnet.

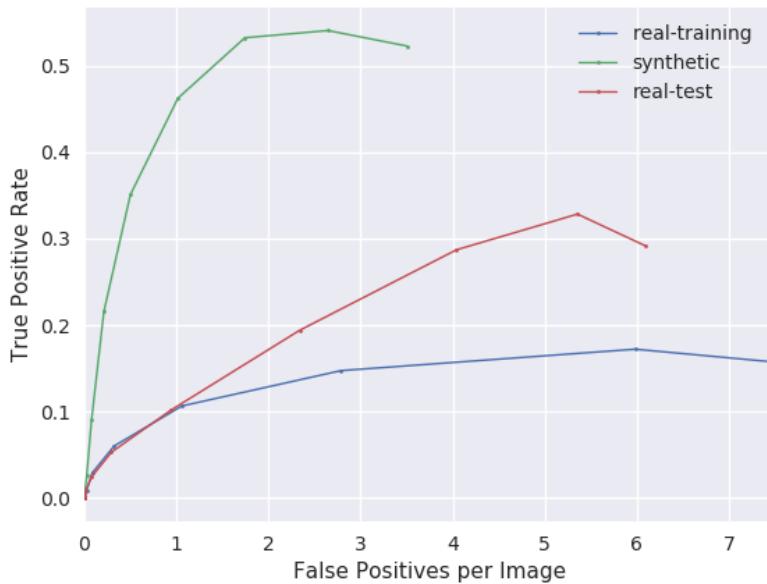


Abbildung 6.1: ROC-Kurve (vgl. Kapitel 5.2) eines Kopf-Schulter-Detektors, der ausschließlich auf synthetischen Daten trainiert ist. Die blaue Kurve wurde auf Basis von 519 Echtwelt-Bildern erstellt. Die rote Kurve wurde mit 10.000 Echtwelt-Bildern erstellt. Die grüne Kurve wurde mit 1901 synthetischen Bildern erstellt, die der Trainingsdatenbank zwar ähneln, selbst jedoch kein Teil dieser sind.

In Abbildung 6.1 ist zu sehen, dass der Detektor auf künstlichen Daten erheblich besser funktioniert, als auf Echtweltdaten. Auf künstlichen Daten erkennt der Detektor über 50 % der Objekte (True Positive Rate) bei durchschnittlich zwei Fehldetections pro Bild. Auf den Echtweltdaten liegt die Detektionsrate bei gleicher Anzahl an Fehldetections zwischen 10 % und 20 %. Auch diese True Positive Rate liegt noch wesentlich über einem zufälligen Ergebnis. Es ist also festzustellen, dass ein nur mit künstlichen Daten trainierter Detektor zwar schlechter funktioniert, als ein Detektor, der mit Echtweltdaten trainiert wurde. Falls jedoch keine oder nur sehr wenige Daten zum Trainieren vorhanden sind, ist es möglich, mit synthetischen Daten ebenfalls einen funktionsfähigen Detektor zu trainieren.

Weitere Verbesserungen sind gegebenenfalls möglich, wenn die Gestaltung der Szene und der Personen der realen Szene noch ähnlicher ist. Dies legt schon der Fakt nahe, dass die ROC-Kurve (rot) der Datenbank, die vom Aufbau der Bilder den synthetischen näher sind, wesentlich besser ist, als die der anderen Echtweltdatenbank (blaue Kurve). Dadurch ist zu sehen, wie sehr die Leistungsfähigkeit von der exakten Beschaffenheit der Daten abhängig ist.

Neben dem rein synthetisch trainierten Detektor wird auch einen Detektor mit Echtweltdaten trainiert. Bei der Trainingsdatenbank handelt es sich um dieselbe,



Abbildung 6.2: Ausgewählte Bilder aus den drei Datenbanken aus Abbildung 6.1. Links ist ein Bild aus der künstlichen Datenbank zu sehen (Abbildung 6.1, grüne ROC-Kurve). Bei dem Bild in der Mitte handelt es sich um ein Bild aus der Realdatenbank, die zu der blauen ROC-Kurve aus Abbildung 6.1 gehört. Rechts ist ein Bild aus der anderen Realdatenbank zu sehen (Abbildung 6.1, rote ROC-Kurve).

die zum Testen des künstlich trainierten Detektors in Abbildung 6.1 verwendet wurde (vgl. Abbildung 6.2, Mitte). In Abbildung 6.3 ist die Leistungsfähigkeit der beiden Detektoren im Vergleich zu sehen. Die orange Kurve ist hierbei das Ergebnis des synthetischen Detektors und die blaue Kurve ist die des Detektors, der mit Echtweltbildern trainiert wurde. Ausgewertet wurden beide Detektoren auf derselben Datenbank mit 10.000 Echtweltbildern (vgl. Abbildung 6.2, rechts).

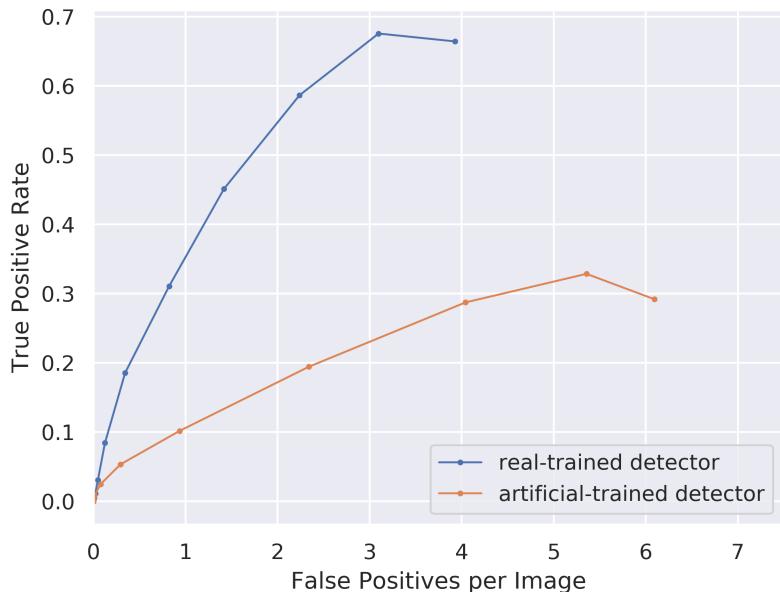


Abbildung 6.3: ROC-Kurven (vgl. Kapitel 5.2) zweier Kopf-Schulter-Detektoren. Die rote Kurve zeigt die Ergebnisse eines mit künstlichen Daten trainierten Detektors, während die blaue Kurve die Leistungsfähigkeit eines Detektors zeigt, der mit Echtweltdaten trainiert ist. Beide Kurven wurden mit derselben unabhängigen Datenbank mit 10.000 Bildern erstellt.

Hier ist zu sehen, dass der Detektor mit den realen Bildern etwa die dreifache

Anzahl an Objekten korrekt erkennt wie der synthetisch trainierte Detektor. Auch hier ist jedoch sichtbar, dass der synthetisch trainierte Detektor ebenfalls einige Objekte erkennt. Sehr wahrscheinlich ist außerdem, dass die Leistungsfähigkeit des Detektors auf Echtwelt-Basis nachlässt, wenn dieser nur mit wenigen Bildern trainiert werden würde. Falls es also Probleme mit der Beschaffung von Echtweltdaten gibt, wie zum Beispiel durch Persönlichkeits- und Urheberrechte, ist es gut möglich, dass sich dieses Defizit durch künstliche Daten kompensieren lässt. Dies sollte in Zukunft untersucht werden (vgl. Abschnitt 6.2).

Alle bis zu diesem Punkt getesteten Detektoren basieren auf Neuronalen Netzen der gleichen Größe. Diese Netze verfügen alle über 38.322 Koeffizienten (biases (engl. für Tendenzen) und Gewichte, vgl. Kapitel 3.1.1). Um zu überprüfen, ob mit einem größeren Netz eine Steigerung der Leistungsfähigkeit möglich ist, wird neben dem existierenden Detektor ein zusätzlicher Detektor mit 151.394 Koeffizienten trainiert und diesen mit dem kleineren Detektor verglichen. Die resultierenden ROC-Kurven sind in Abbildung 6.4 dargestellt. Hier ist zu erkennen, dass die ROC-Kurven sehr nah aneinander liegen. Der größere Detektor (grüne Kurve) funktioniert zwar ein wenig besser (die ROC-Kurve liegt etwas höher) als der kleinere Detektor (blaue Kurve), aber der Unterschied ist so gering, dass er nicht wesentlich ins Gewicht fällt. Ein Detektor benötigt außerdem mehr Rechenleistung je größer das Neuronale Netz ist. Aus diesen beiden Gründen lässt sich zusammenfassend sagen, dass eine Vergrößerung des Neuronalen Netzes in diesem Fall nicht sinnvoll ist. Dies deckt sich mit früheren Versuchen in der Abteilung, die mit anderen Detektoren durchgeführt wurden.

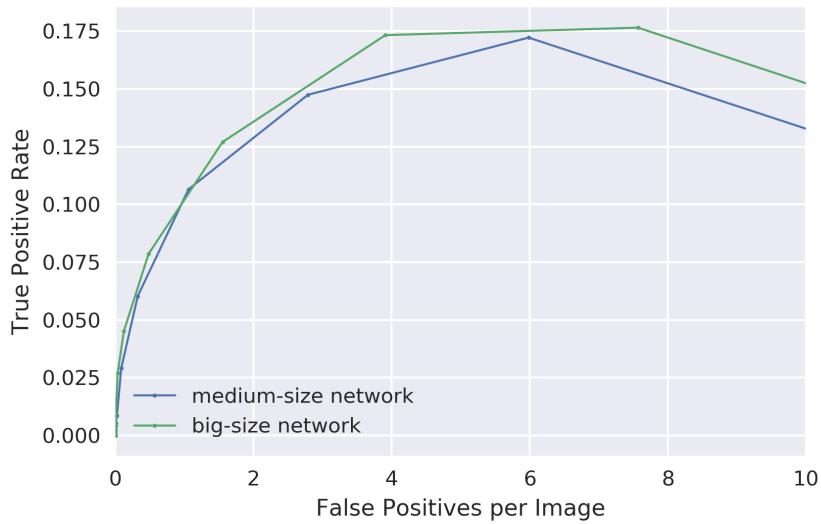


Abbildung 6.4: ROC-Kurven (vgl. Kapitel 5.2) zweier Kopf-Schulter-Detektoren.

Die blaue Kurve zeigt die Ergebnisse eines Detektors, der mit einem Neuronalen Netz mit 38.322 Koeffizienten arbeitet. Die grüne Kurve zeigt die Ergebnisse eines Detektors, der mit einem wesentlich größeren Neuronalen Netz mit 151.394 Koeffizienten arbeitet. Alle anderen Detektoren in dieser Arbeit funktionieren auf Basis von Netzen mit 38.322 Koeffizienten. Beide Kurven wurden mit derselben unabhängigen Datenbank mit 519 Bildern erstellt.

### 3D Personendetektor

Neben den in Abschnitt 6.1.1 beschriebenen Detektoren zur Erkennung von Personen direkt von oben wurde auch ein Detektor für seitliche Blickrichtungen (vgl. Abbildung 6.5) mit künstlichen Daten trainiert.

In Abbildung 6.6 sind die ROC-Kurven von zwei Detektoren zu sehen. Die blaue Kurve ist die ROC-Kurve des Detektors, der mit synthetischen Daten trainiert wurde. Die grüne Kurve gehört zu einem Personendetektor, welcher mit Echtweltdaten trainiert wurde. Dieser basiert jedoch auf einem kleineren Neuronalen Netz und ist mit einem etwas aufwendigeren Verfahren trainiert. Aus diesem Grund sind die beiden Detektoren nicht direkt vergleichbar. Zu sehen ist aber, dass auch der besser trainierte Echtwelldetektor noch recht viele False Positives (Fehlauslösungen) bei einer recht geringen True Positive Rate hat. Dies ist wahrscheinlich durch die größere Menge an Personen in den Bildern zu erklären, durch die die Wahrscheinlichkeit steigt, dass der Detektor eine Person übersieht. Außerdem stehen die Personen teilweise erheblich näher zusammen als etwa in den Bildern in Abschnitt 6.1.1 und verdecken sich zum Teil sogar. Hierdurch wird es zum einen schwieriger für den Detektor, Personen zu trennen, und zum anderen steigt durch die vollere



Abbildung 6.5: Echtweltbild mit Auslösungen eines Personendetektors, der auf künstlichen Daten trainiert wurde. In den Boxen steht die Konfidenz, mit der die Personen erkannt wurden.

Szene die Komplexität der Texturen und die Menge an Details und damit auch die Wahrscheinlichkeit für Fehlauslösungen. Auch in diesem Beispiel ist aber trotz der vermeintlich schlechten Leistungsfähigkeit, die die ROC-Kurve nahelegt, zu sehen, dass es möglich ist, mit künstlichen Daten Personendetektoren zu trainieren, die in der Lage sind, sinnvolle Ergebnisse zu erzielen.

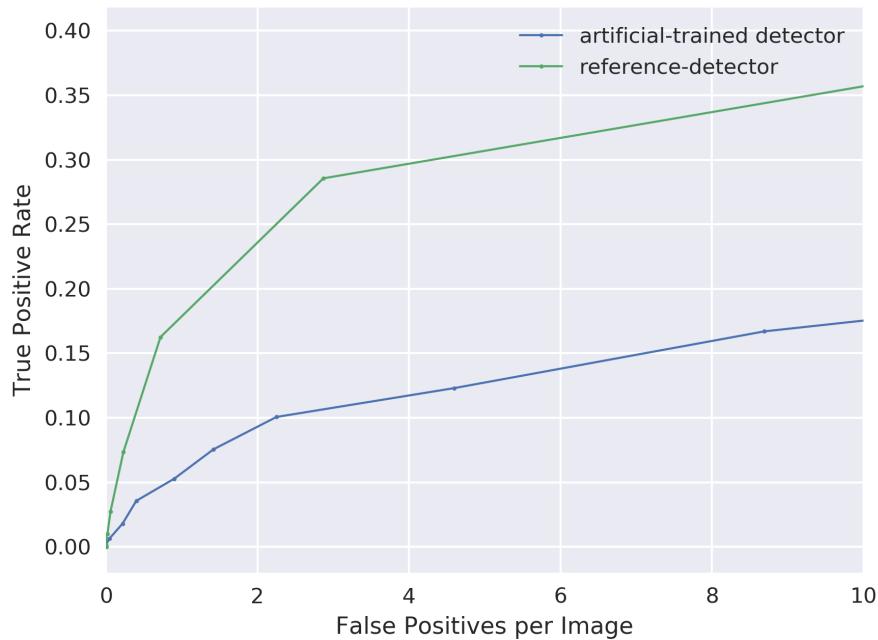


Abbildung 6.6: ROC-Kurven zweier Kopf-Schulter Detektoren zum seitlichen Erkennen von Personen. Beide ROC-Kurven wurden mit 200 Echtweltbildern erstellt. Bei der blauen Kurve handelt es sich um Daten von einem Detektor, der mit künstlichen Daten trainiert wurde. Als Referenz ist ein weiterer Detektor eingezeichnet. Dieser wurde auf Echtweltdaten trainiert. Der Referenz-Detektor arbeitet mit einer kleineren Architektur, ist dafür aber durch weitere Trainingsschritte noch verbessert worden. Dieser Detektor wurde verwendet, da zum Zeitpunkt der Erstellung kein anderer zur Verfügung stand.

### 6.1.2 Evaluierung des Gesichtsdetektors in Abhängigkeit der Blickrichtung und der Beleuchtung

Neben Detektoren zur Erkennung von Personen anhand von ihren Kopf-Schulter-Partien gibt es auch Detektoren, die Gesichter von Menschen erkennen können. Die Detektoren können in diesem Fall aber nur erkennen, ob sich in einem Bild ein Gesicht befindet und nicht zwischen verschiedenen Gesichtern unterscheiden. Solche einfachen Gesichtsdetektoren sind auch häufig in Fotokameras und Smartphones eingebaut.

#### Evaluierung des Gesichtsdetektors in Abhängigkeit der Blickrichtung

Ein Kriterium bei Gesichtsdetektoren ist die Leistungsfähigkeit des Detektors in Abhängigkeit vom Blickwinkel auf das Gesicht. So kann ein Detektor beispielsweise verhältnismäßig gut Gesichter erkennen, wenn die Personen direkt zur Kamera blicken. Sieht derselbe Detektor jedoch ein Gesicht von der Seite, so kann es passieren, dass er dieses Gesicht gar nicht erkennt. Um das Verhalten eines Gesichtsdetektors

in diesem Zusammenhang zu testen, werden Bilder benötigt, in denen ein Gesicht in einem vorgegebenen Winkel zu sehen ist. Solch einen Datensatz mit echten Menschen zu erstellen, ist zum einen ungenau und zum anderen sehr zeitaufwendig, wenn die Schrittweite der Winkel klein genug sein soll, um hochauflösende Ergebnisse zu erhalten. Aus diesen Gründen bietet es sich an, Auswertungen dieser Art mit künstlichen Daten durchzuführen.

Hierzu habe ich von fünf Gesichtern Bilder erzeugt (vgl. Abbildung 6.7).



Abbildung 6.7: Diese fünf Gesichter wurden zu Erstellung von Abbildung 6.10 und Abbildung 6.9 genutzt.

Die Blickrichtung lag hierbei horizontal zwischen  $-90^\circ$  und  $90^\circ$ , was jeweils einem Blick direkt von der Seite auf das Ohr der Person entspricht. Vertikal lag der Winkel zwischen  $-60^\circ$ , was einer Froschperspektive entspricht, und  $60^\circ$ , was einem Blick schräg von oben entspricht. Dieser Bereich wurde sowohl horizontal als auch vertikal in  $2^\circ$  Schritten abgegangen. So entstanden pro Gesicht  $\left(\left(\frac{180^\circ}{2^\circ} + 1\right) \times \left(\frac{120^\circ}{2^\circ} + 1\right)\right) = 5551$  Bilder, wie sie beispielhaft in Abbildung 6.8 zu sehen sind.

Diese Bilder wurden daraufhin von einem Gesichtsdetektor, der mit Echtweltda-



Abbildung 6.8: Zwei Bilder aus einer Datenbank an Bildern von künstlichen Gesichtern zur Bestimmung der Leistungsfähigkeit eines Gesichtsdetektors in Abhängigkeit vom Blickwinkel auf das Gesicht. Das linke Bild zeigt ein Gesicht aus einem Blickwinkel von  $-30^\circ$  in vertikaler Richtung und  $-50^\circ$  in horizontaler Richtung, während das rechte Bild unter einem vertikalen Winkel von  $+30^\circ$  und einem horizontalen Winkel von  $+50^\circ$  betrachtet wird.

ten trainiert worden war, analysiert. Hierbei wurde für jedes Bild die Konfidenz bestimmt, mit der der Detektor in dem Bild ein Gesicht erkannte. Um Fehlauslösungen gering zu halten, wurden alle Konfidenzwerte unter 0,5 ignoriert. Diese Werte wurden für alle Blickwinkel einzeln über die fünf Gesichter gemittelt.

In Abbildung 6.9 sind die gemittelten Konfidenzen dargestellt. Rote Bereiche entsprechen hierbei Winkeln, in denen der Detektor keine Gesichter erkannt hat, in schwarzen Bereichen hat der Detektor alle Gesichter mit einer Konfidenz nahe eins erkannt. Die weißen Bereiche liegen in der Mitte. Hier hat der Detektor die Gesichter nur mit recht geringer Sicherheit erkannt.

In Abbildung 6.10 ist zu sehen, dass auch an den Rändern der Abbildungen häufiger Auslösungen mit geringen Konfidenzen vorliegen. Hierbei handelt es sich sehr wahrscheinlich um Fehlauslösungen, wie zum Beispiel in Abbildung 6.11 zu sehen. Durch die Bildung des Durchschnitts in Abbildung 6.9 verringert sich dieser Effekt jedoch merklich, da es unwahrscheinlich ist, dass Fehlauslösungen in allen Bildern unter denselben Winkeln auftreten.

Abbildung 6.9 zeigt außerdem, dass der Gesichtsdetektor bis zu einem Winkel von etwa  $40^\circ$  in alle Richtungen im Durchschnitt gut funktioniert. In Abbildung 6.10 ist auf der anderen Seite ersichtlich, dass die Leistungsfähigkeit des Detektors auch von dem einzelnen Gesicht abhängt, was betrachtet wird. So wird das Gesicht, das in Abbildung 6.7 und Abbildung 6.10 oben rechts abgebildet ist, schon bei geringeren Winkeln nur noch schlecht erkannt, wohingegen Gesichter, wie das in den beiden zuvor genannten Abbildungen ganz unten gezeigte, auch unter größeren Winkeln noch gut erkannt werden.

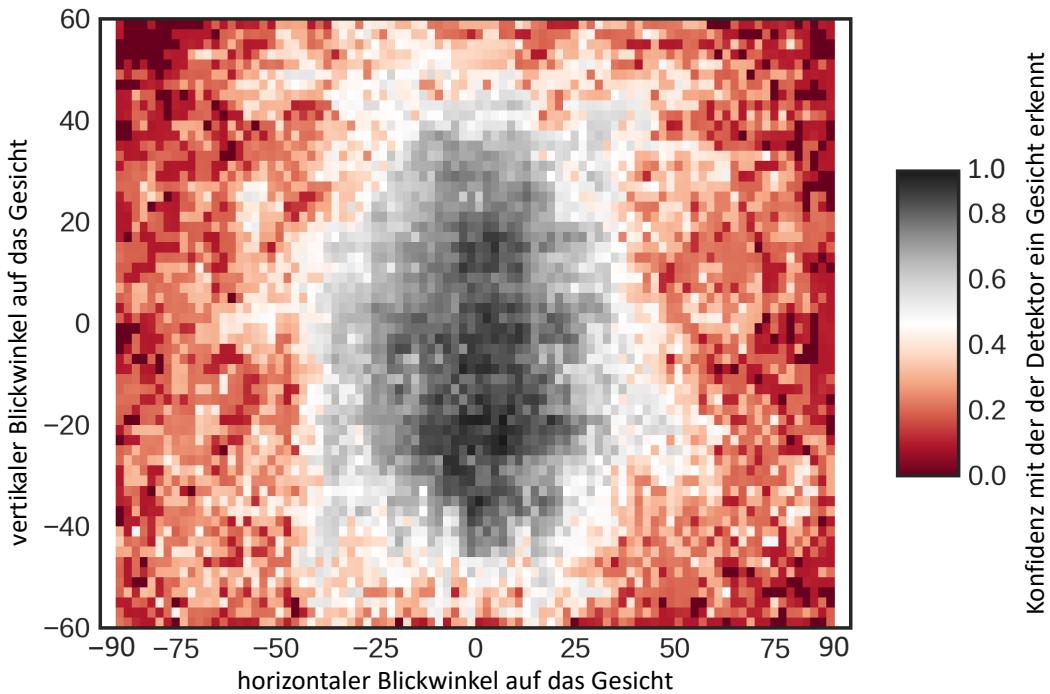


Abbildung 6.9: Durchschnitt der Detektionssicherheiten eines auf Echtweltdaten trainierten Gesichtsdetektors in Abhängigkeit von der Blickrichtung, aus der auf das Gesicht geschaut wird. Die Farbe stellt die Sicherheit dar, mit der der Detektor Gesichter erkennt. Die Werte auf der Hochachse geben den vertikalen Winkel an, aus dem auf das Gesicht geschaut wird.  $-60^\circ$  bedeutet hierbei, dass aus einer Froschperspektive geschaut wird, wohingegen  $+90^\circ$  einer Blickrichtung direkt von oben entsprechen würde. Die Werte an der Längsachse geben die Winkel in der Horizontalen an.  $-90^\circ$  entspricht hier einem Blick auf das rechte Ohr,  $90^\circ$  einem Blick auf das linke. Zur Erstellung der Grafik wurden jeweils 5551 Bilder von 5 Gesichtern gemittelt.

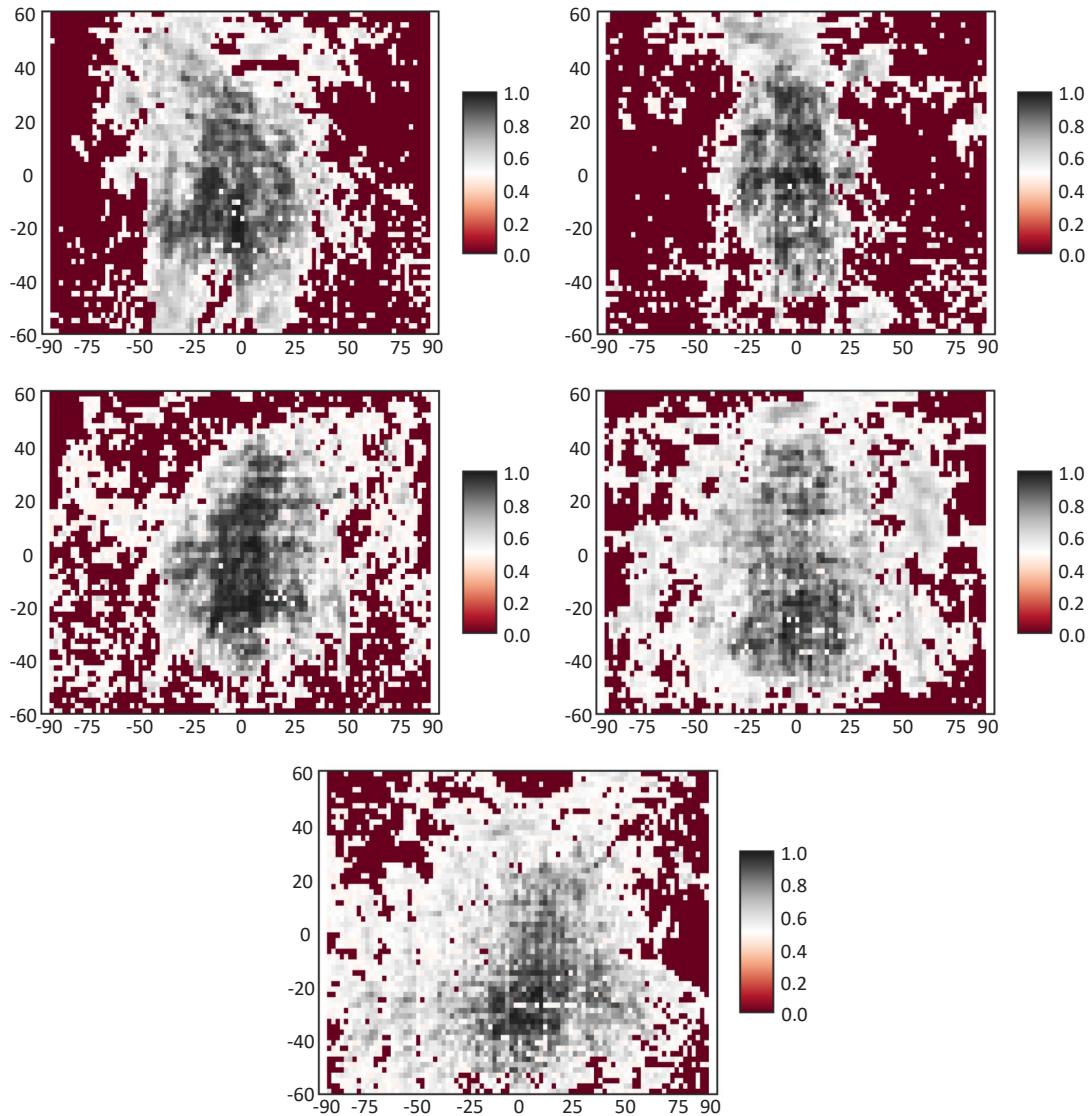


Abbildung 6.10: Detektionssicherheit eines auf Echtweltdaten trainierten Gesichtsdetektors in Abhängigkeit von der Blickrichtung, aus der auf das Gesicht geschaut wird. Die Farbe stellt die Sicherheit dar, mit der der Detektor Gesichter erkennt. Die Werte auf der Hochachse geben den vertikalen Winkel an, aus dem auf das Gesicht geschaut wird.  $-60^\circ$  bedeutet hierbei, dass aus einer Froschperspektive geschaut wird, wohingegen  $+90^\circ$  einer Blickrichtung direkt von oben entsprechen würde. Die Werte an der Längsachse geben die Winkel in der Horizontalen an.  $-90^\circ$  entspricht hier einem Blick auf das rechte Ohr,  $90^\circ$  einem Blick auf das linke. Zur Erstellung der Grafik wurden 5551 Bilder pro Gesicht genutzt. Die Reihenfolge der Gesichter entspricht der aus Abbildung 6.7.

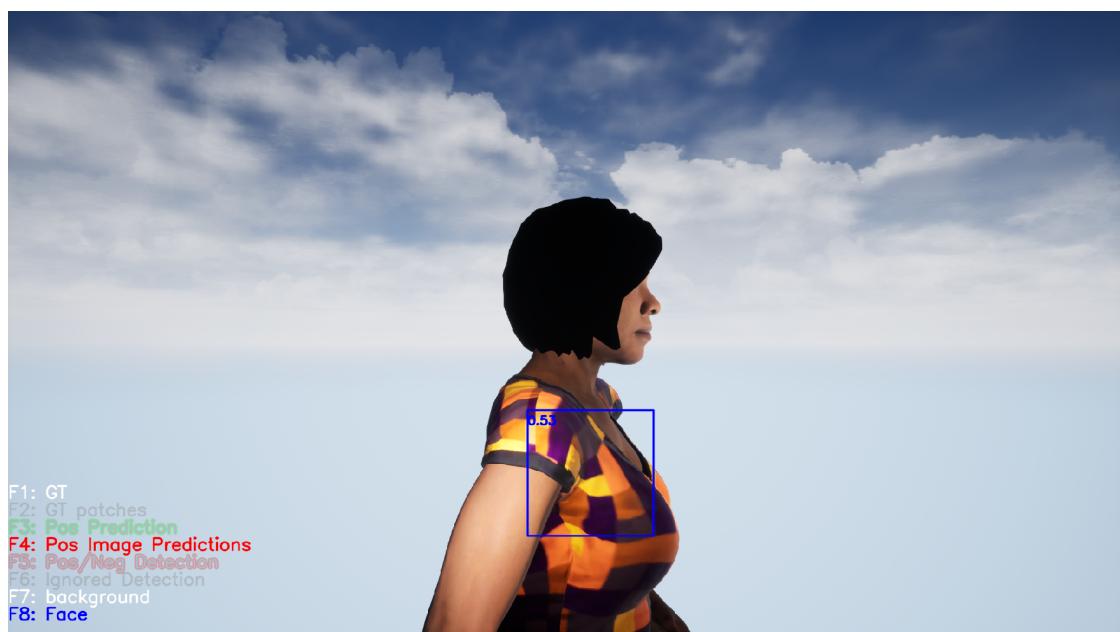


Abbildung 6.11: Fehlauslösung des verwendeten Gesichtsdetektors am Randbereich der Diagramme in Abbildung 6.9 und Abbildung 6.10.

## Evaluierung des Gesichtsdetektors in Abhängigkeit der Beleuchtung

Zum Testen des Gesichtsdetektors bei verschiedenen Beleuchtungssituationen habe ich außerdem ein Gesicht in fünf verschiedenen Beleuchtungssituationen aufgenommen und von dem Gesichtsdetektor untersuchen lassen. Sowohl die Gesichter als auch die Detektionen sind in Abbildung 6.12 zu sehen.

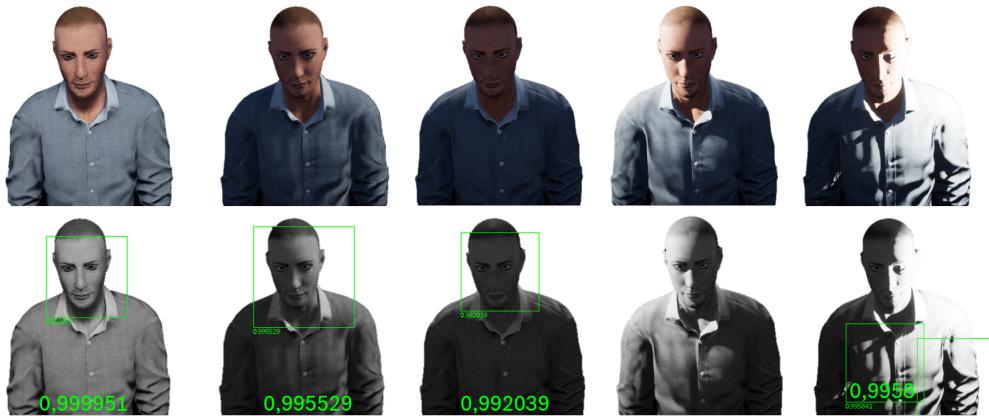


Abbildung 6.12: Ein Gesicht in fünf verschiedenen Lichtsituationen mit den Detektionen eines Gesichtsdetektors. Die grünen Zahlen geben die Konfidenzen wieder, mit denen der Detektor ein Gesicht erkannt hat.

Die Abbildung 6.12 zeigt gut, wie sehr die Leistungsfähigkeit eines Detektors von äußeren Bedingungen abhängt. Während der Detektor zum Beispiel das Gesicht im ganz linken Bild, welches recht gleichmäßig beleuchtet ist, gut und sicher erkennt, werden die beiden Gesichter ganz rechts nicht mehr ausreichend erkannt. Die beiden rechten Gesichter sind sehr ungleichmäßig von einer Seite beleuchtet. Dies sorgt dafür, dass die Form des Gesichts aufgebrochen wird. Unter anderem mit Situationen wie diesen hat der Detektor Probleme. Die Beleuchtung im ganz rechten Bild führt zudem zu Fehlauslösungen auf dem Oberkörper der Person.

Mithilfe von künstlichen Daten lassen sich solche Schwachstellen gut eingrenzen und systematisch analysieren. Im Folgenden habe ich hierfür ein Gesicht in verschiedenen Lichtsituationen erstellt.

In Abbildung 6.14 ist die Konfidenz eines Gesichtsdetektors für ein Gesicht zu sehen, welches von der Seite beleuchtet wird. Die Lichtquelle auf der linken Seite des Gesichts ist auf eine konstante Intensität eingestellt. Die Helligkeit des Hintergrunds wird ebenfalls konstant gehalten. Die Beleuchtungsstärke der Lichtquelle auf der rechten Seite ist variabel. Hierfür kann in der *Unreal Engine 4* ein Wert in der Einheit Candela eingestellt werden, welcher von der Engine simuliert wird. Die Stärke der Beleuchtung ist auf der X-Achse aufgetragen und die entsprechende

Konfidenz auf der Y-Achse. In Abbildung 6.13 sind exemplarisch drei der 50 Bilder zu sehen, die zur Erstellung von Abbildung 6.14 genutzt wurden. Das Gesicht links ist bei 0 Candela, das Gesicht in der Mitte bei 3 Candela und das Bild ganz rechts bei 50 Candela erstellt.



Abbildung 6.13: Bilder eines künstlichen Gesichts mit seitlicher Beleuchtung in verschiedenen Intensitäten.

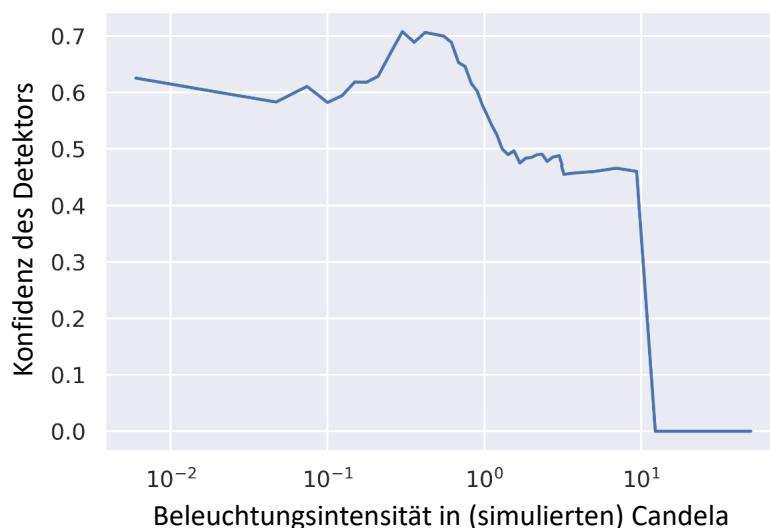


Abbildung 6.14: Konfidenzen eines Gesichtsdetektors mit seitlicher Beleuchtung in verschiedenen Intensitäten. Die Stärke der Beleuchtung ist auf der X-Achse aufgetragen und die entsprechende Konfidenz auf der Y-Achse.

In Abbildung 6.14 ist zu erkennen, dass der Gesichtsdetektor bei mittlerer und gleichmäßiger Beleuchtung am besten funktioniert. Bei etwa 0,5 Candela ist die Leistungsfähigkeit des Detektors maximal und ab etwa 10 Candela ist der Detektor nicht mehr in der Lage, das Gesicht zu erkennen.

Ähnlich wie in Abbildung 6.14 ist in Abbildung 6.16 die Konfidenz eines Gesichtsdetektors für ein Gesicht zu sehen, welches in diesem Fall von oben beleuchtet wird. In Abbildung 6.15 sind exemplarisch drei der 50 Bilder zu sehen, die zur Erstellung von Abbildung 6.16 genutzt wurden. Das Gesicht links ist bei 0 Candela, das Gesicht in der Mitte bei 1,3 Candela und das Bild ganz rechts bei 50 Candela erstellt.



Abbildung 6.15: Bilder eines künstlichen Gesichts mit Beleuchtung von oben in verschiedenen Intensitäten

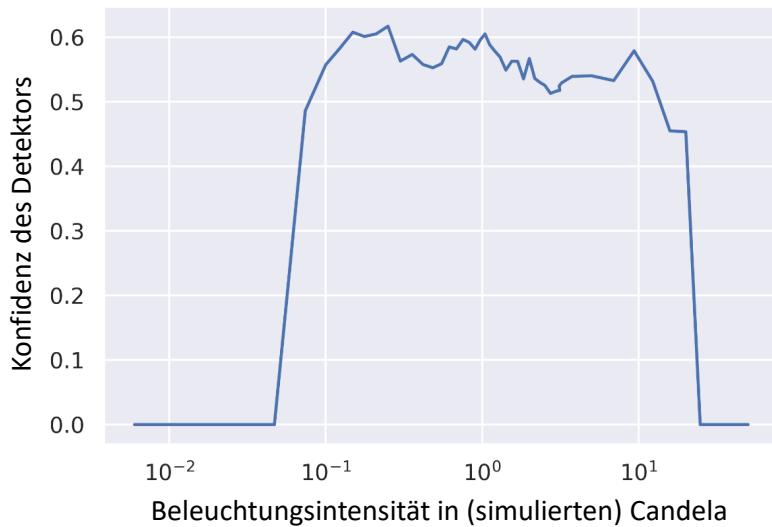


Abbildung 6.16: Konfidenzen eines Gesichtsdetektors mit Beleuchtung in verschiedenen Intensitäten von oben. Die Stärke der Beleuchtung ist auf der X-Achse aufgetragen und die entsprechende Konfidenz auf der Y-Achse.

In Abbildung 6.16 ist zu sehen, dass das Gesicht nur in einem Bereich zwischen etwa  $5 \times 10^{-2}$  Candela und 20 Candela erkannt wird. Dass das Gesicht nicht wie in Abbildung 6.14 ab 0 Candela erkannt wird, liegt daran, dass es in diesem Fall keine zweite Lichtquelle gibt und das Gesicht somit bei 0 Candela gänzlich dunkel ist. Auch in dem Bereich, in dem das Gesicht erkannt wird, sind die Konfidenzen des Detektors geringer als in Abbildung 6.14. Dies liegt sehr wahrscheinlich an den stärkeren Schatten, die durch die Beleuchtung von oben im Gesicht entstehen.

Auswertungen wie diese werden durch künstliche Datengenerierung erheblich vereinfacht. So war es mir in diesem Fall möglich, die Richtung und Intensität des Lichts, das auf die Person fällt, beliebig zu verändern, ohne dass sich andere Parameter wie der Gesichtsausdruck oder die Position der Person ebenfalls verändern. Diese Ergebnisse mit echten Menschen in der realen Welt zu erzeugen wäre, wenn überhaupt, nur mit hohem Aufwand möglich.

## 6.2 Ausblick

Einige Szenarien konnten aufgrund der zeitlichen Beschränkung nicht mehr getestet werden. So sollte in Zukunft zum Beispiel noch untersucht werden, ab welcher Anzahl an verfügbaren Echtweltbildern der Einsatz von künstlichen Daten im Training von Objektdetektoren das Ergebnis verbessert und wie sich dies in Abhängigkeit von der Qualität der künstlichen Daten ändert. Auch das Trainieren von existierenden Detektoren für spezielle Einsatzzwecke mit synthetischen Daten sollte noch weiter untersucht werden. In dieser Arbeit wurden ausschließlich Detektoren trainiert, die Menschen erkennen sollen. Es sollte weiterhin überprüft werden, ob sich künstliche Daten anders auf ein Training auswirken, wenn Detektoren für andere Objekte, wie beispielsweise Autos, erstellt werden.

Auch die Einbindung weiterer Datenquellen, wie zum Beispiel dreidimensionale CAD-Architekturmodelle von Gebäuden zur Erstellung von Szenen beim szenenspezifischen Trainieren von Detektoren, kann interessante Möglichkeiten für zukünftige Projekte bieten.

Das Verfahren zur Erstellung künstlicher Daten kann ebenfalls noch weiter verbessert werden. So sollte bei der Nutzung des Sequenzers auch die Position und Orientierung der Kamera mit an das Script zur Umrechnung der Koordinaten übergeben werden. Wenn diese Übergabe implementiert ist, sollte versucht werden, auch Kamerabewegungen im Sequenzer durch ein Script zu steuern. So wäre es möglich, die Nutzung von UnrealCV zu umgehen und so die Rendergeschwindigkeit auch bei Szenen mit Kamerabewegungen zu erhöhen. Dadurch wird auch der Schattenwurf der Kamera, der bei der Nutzung des UnrealCV Plug-in auftritt, vermieden.

## 6.3 Fazit

Künstliche Datengenerierung ist ein schneller und relativ einfacher Weg, um fertig annotierte Daten zu erhalten. Die wesentliche Arbeit fließt hierbei meistens in die Modellierung der Objekte, die detektiert werden sollen, sowie der umgebenden Welt. Sind diese Komponenten fertig, lassen sich in verhältnismäßig kurzer Zeit viele Trainings- und Testdaten erstellen.

Das Training eines Neuronalen Netzes mit synthetischen Daten ist möglich. Bei einer Qualität der künstlichen Daten, wie sie in diesem Projekt erzielt wurde, sind Echtweltdaten für das Training zu bevorzugen, da die resultierenden Detektoren leistungsfähiger sind. Es ist jedoch davon auszugehen, dass die Leistungsfähigkeit von künstlich trainierten Detektoren mit einer steigenden Qualität der Daten und

einem höheren Grad an Realismus zunimmt. Wenn also realistischere 3D-Modelle und Szenen zur Verfügung stehen oder nur eine geringe Menge an Echtweltdaten verfügbar ist, so ist das Training mit künstlichen Daten eine gute Alternative.

Zur Auswertung bestehender Detektoren eignen sich künstliche Daten aus meiner Sicht sehr gut. Besonders positiv hervorzuheben ist hierbei die Reproduzierbarkeit. Mit künstlichen Daten ist es einfach möglich, eine Szene mehrfach mit einigen veränderten Einstellungen (zum Beispiel Lichtverhältnisse oder Blickrichtungen) unter ansonsten aber exakt gleichbleibenden Bedingungen zu rendern. Dies ermöglicht die gezielte Analyse des Einflusses einzelner Parameter auf den Detektor unabhängig von weiteren Faktoren. Auch die Präzision, mit der Auswertungen, wie in Abschnitt 6.1.2 beschrieben, erstellt werden können, ist mit künstlich generierten Daten erheblich besser zu erreichen als mit Echtweltdaten. So ist es im Computer beispielsweise einfacher möglich, eine Person in exakten zwei-Grad-Schritten zu drehen, als in der Realität. Es muss bei Auswertungen mit künstlich generierten Daten jedoch bedacht werden, dass ihre Ergebnisse von denen, die man mit Echtweltdaten erreichen würde, leicht abweichen können. Sie bieten im Moment vielmehr einen Anhaltspunkt für den Einfluss verschiedener Parameter auf die Leistungsfähigkeit des Detektors. Es ist zu erwarten, dass die Verhältnisse der Messwerte, die mit synthetischen Daten bei varierten Testparametern ermittelt wurden, korrekt sind, während die Absolutwerte von denen, die mit Echtweltdaten bestimmt werden, abweichen können.

Für Einschätzungen über die Leistungsfähigkeit eines Detektors unter verschiedenen Bedingungen und für Vergleiche verschiedener Detektoren sind künstliche Daten meiner Meinung nach gut geeignet.

Das Projekt demonstriert erfolgreich die Erstellung und Verwendung von synthetischen Daten zum Training und zur systematischen Auswertung von Objektdetectoren. Die Trainingserfolge können in Zukunft durch Änderungen von Details (vgl. Abschnitt 6.2) noch weiter verbessert werden. Die hier gezeigten Ergebnisse zeigen bereits sehr nützliche und interessante Anwendungsgebiete auf.

# Literaturverzeichnis

- [1] Bosch Security and Safety Systems Germany. Intelligentere Videosysteme bedeuten mehr als nur Sicherheit. <https://www.boschsecurity.com/de/de/produkte/videosysteme/video-analytics/>, Mai 2019.
- [2] André Wiedemann. Intelligent Analysis and Creation of Training Data for simple Object Detectors based on Convolutional Neural Networks. Master's thesis, Institut für Photogrammetrie, Universität Stuttgart, 2018.
- [3] Yair Movshovitz-Attias, Takeo Kanade, and Yaser Sheikh. How useful is photo-realistic rendering for visual learning? *arXiv:1603.08152 [cs.CV]*, 03 2016.
- [4] Jonathan Tremblay, Aayush Prakash, David Acuna, Mark Brophy, Varun Jampani, Cem Anil, Thang To, Eric Cameracci, Shaad Boochoon, and Stan Birchfield. Training deep networks with synthetic data: Bridging the reality gap by domain randomization. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 969–977, 2018.
- [5] Nikolaus Mayer, Eddy Ilg, Philipp Fischer, Caner Hazırbaş, Daniel Cremers, Alexey Dosovitskiy, and Thomas Brox. What makes good synthetic training data for learning disparity and optical flow estimation? *International Journal of Computer Vision*, 01 2018.
- [6] David Vázquez, Antonio López, Javier Marín, Daniel Ponsa, and David Geronimo. Virtual and real world adaptation for pedestrian detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, page 13, 08 2014.
- [7] Hironori Hattori, Vishnu Naresh Boddeti, Kris Kitani, and Takeo Kanade. Learning scene-specific pedestrian detectors without real data. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 3819–3827, 06 2015.
- [8] Will Knight. Self-Driving Cars Can Learn a Lot by Playing Grand Theft Auto. <https://www.technologyreview.com/s/602317/self-driving-cars-can-learn-a-lot-by-playing-grand-theft-auto/>, Juli 2019.
- [9] Deep Vision Data. <https://synthetictrainingdata.com/>, Juli 2019.

- [10] AI.Reverie. <https://aireverie.com/>, Juli 2019.
- [11] Michael Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015.
- [12] Francois Chollet. *Deep Learning mit Python und Keras: Das Praxis-Handbuch vom Entwickler der Keras-Bibliothek*. MITP-Verlags GmbH & Co. KG, 2018.
- [13] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In Yves Lechevallier and Gilbert Saporta, editors, *Proceedings of COMPSTAT'2010*, pages 177–186, Heidelberg, 2010. Physica-Verlag HD.
- [14] Jorge Nocedal and Stephen J. Wright. *Numerical optimization*. Springer series in operations research and financial engineering. Springer, New York, NY, 2. ed. edition, 2006.
- [15] Saskia Hein. Stochastisches Lernen. Master's thesis, Westfälische Wilhelms-Universität Münster, Fachbereich Mathematik und Informatik, 2018.
- [16] Yann LeCun, Corinna Cortes, and Christopher J.C. Burges. THE MNIST DATABASE of handwritten digits. <http://yann.lecun.com/exdb/mnist/>, June 2019.
- [17] NumPy developers. NumPy. <https://www.numpy.org/>, Juni 2019.
- [18] Google Brain. TensorFlow. <https://www.tensorflow.org/overview>, Juni 2019.
- [19] Thomas Theis. C++ (Programmiersprache). <https://www.it-treff.de/it-lexikon/cpp-programmiersprache>, Juni 2019.
- [20] Unreal Engine 4 Documentation. <https://docs.unrealengine.com/en-us/>, Mai 2019.
- [21] Unreal Engine 4 Website. <https://www.unrealengine.com/en-us/>, June 2019.
- [22] Adobe Mixamo. <https://www.mixamo.com/>, Mai 2019.
- [23] Weichao Qiu and Alan Loddon Yuille. UnrealCV: Connecting Computer Vision to Unreal Engine. In *ECCV Workshops*, 2016.
- [24] Weichao Qiu Et al. UnrealCV: Virtual Worlds for Computer Vision. *ACM Multimedia Open Source Software Competition*, 2017.

- [25] UnrealCV 0.3.10 documentation. <http://docs.unrealcv.org/en/stable/>, Mai 2019.
- [26] Dieter Lasser. *Grundlagen der geometrischen Datenverarbeitung*. Springer-Verlag, 2013.
- [27] Bildpyramide. [www.spektrum.de/lexikon/kartographie-geomatik/bildpyramide/546](http://www.spektrum.de/lexikon/kartographie-geomatik/bildpyramide/546), Juli 2019.
- [28] Mark H Zweig and Gregory Campbell. Receiver-operating characteristic (ROC) plots: a fundamental evaluation tool in clinical medicine. *Clinical chemistry*, 39(4):561–577, 1993.

# Anhang

## Python Script zur Umrechnung der Koordinaten (Kapitel 4.5)

```
# -*- coding: utf-8 -*-
"""
Created on Tue Apr 30 15:49:01 2019

@author: lua4hi

Script to read .txt Annotation files from the Unreal Engine and
transform Coordinates from World- to Image-coordinate-system.

Parameters:
- .txt and .xml file path (line 26 & 29)
- camera position (line 34ff)
- camera orientation (line 36ff)

Importante Note:
you need to update the camera parameters before executing this script
"""

import xml.etree.cElementTree as ET
import numpy as np
import os
from tqdm import tqdm

storage_location_txt='C:/ Users/<user>/Documents/UE4_Data_Generation/
Saved/VideoCaptures/'

storage_location_xml = storage_location_txt
"""

3D camera           [    X,    Y,    Z]    ''
position_camera =np.array([-2000,-1000,540])
'',                   [X, Y, Z]    ''
orientation_camera = np.array([0,-40,90])
'',                   [X, Y, Z]    ''
BEV camera          [X,    Y,    Z]    ''
#position_camera = np.array([0, -700, 460])
'',                   [X, Y, Z]    ''
#orientation_camera = np.array([0, -90, 90])

print('ncamera position: ', position_camera)
print('camera orientation: ', orientation_camera)
```

```

camera_updated=input('Have you checked the camera parameters?(y/n):')

if camera_updated == 'n' or camera_updated == 'N':
    exit()

"""

correction Factor from lag between
Animation and Rendering in Unreal Engine
"""

filename_number = -1

folder = [filename for filename in
          os.listdir(storage_location_txt) if
          filename.endswith('.txt')]

if len(folder) == 0:
    print('Folder is empty! \nCheck folder path.\n',
          storage_location_txt)
    temp = input('press any key to exit')
    exit()
with tqdm(total=len(folder)) as pbar:
    for filename in folder:
        """
        reading txt file
        """
        file = open(storage_location_txt + filename, 'r')
        lines = file.readlines()
        file.close()
        filename_text = filename[: filename.find('_') + 1]

        if filename.find('.') != -1:
            filename = filename[: filename.find('.')]

        """
        Umrechnung World -> Bildkoordinaten
        """

# Rotation Matrix for rotating the coordinate system
# Matrix zur Drehung um Z-Achse
drehung_z = np.array(
    [[np.cos(-np.radians(orientation_camera[2])), -
      1 * np.sin(-np.radians(orientation_camera[2])),

```

```

        0] , [np.sin(-np.radians(orientation_camera[2])) ,
       np.cos(-np.radians(orientation_camera[2])) ,
       0] , [0, 0, 1]])

# Matrix zur Drehung um Y-Achse
drehung_y = np.array(
    [[np.cos(np.radians(orientation_camera[1])), 0,
      np.sin(np.radians(orientation_camera[1]))],
     [0, 1, 0],
     [-1 * np.sin(np.radians(orientation_camera[1])), 0,
      np.cos(np.radians(orientation_camera[1]))]])

# Matrix zur Drehung um Y-Achse und Z-Achse kombiniert
drehung = np.dot(drehung_y, drehung_z)

'''Erstellen der XML-Dateien'''

# Create xml Tree

root = ET.Element("root")

bildkoordinate_perspektivisch_left = np.array([0, 0, 0])
bildkoordinate_perspektivisch_right = np.array([0, 0, 0])
bildkoordinate_perspektivisch_head_top = np.array([0, 0, 0])
bildkoordinate_perspektivisch_head_bottom = np.array([0, 0, 0])

for line in lines:
    '''Read Character Name'''
    if line.find('charactername:') != -1:
        charactername = line[line.find(':') + 1:]

# Read Left Shoulder Position

    if line.find('LeftShoulder') != -1:
        x_pose = line.find('X=')
        y_pose = line.find('Y=')
        z_pose = line.find('Z=')
        x_world = float(line[x_pose + 2: y_pose - 1])
        y_world = float(line[y_pose + 2: z_pose - 1])
        z_world = float(line[z_pose + 2:])

# Calculate Bildkoordinaten
keypoint = np.array([x_world, y_world, z_world])

# rotating the coordinate system
bildkoordinate = np.dot(drehung,
                       (keypoint - position_camera))

```

```

# Perspectiv correction
bildkoordinate_perspektivisch_left = bildkoordinate /
(bildkoordinate[0])*960

# Read Right Shoulder Position

if line.find('RightShoulder') != -1:
    x_pose = line.find('X=')
    y_pose = line.find('Y=')
    z_pose = line.find('Z=')
    x_world = float(line[x_pose + 2: y_pose - 1])
    y_world = float(line[y_pose + 2: z_pose - 1])
    z_world = float(line[z_pose + 2:])
# Calculate Bildkoordinaten
keypoint = np.array([x_world, y_world, z_world])

# rotating the coordinate system
bildkoordinate = np.dot(drehung,
                        (keypoint - position_camera))

# Perspectiv correction
bildkoordinate_perspektivisch_left = bildkoordinate /
(bildkoordinate[0])*960

# Read Head Top Position

if line.find('HeadTop') != -1:
    x_pose = line.find('X=')
    y_pose = line.find('Y=')
    z_pose = line.find('Z=')
    x_world = float(line[x_pose + 2: y_pose - 1])
    y_world = float(line[y_pose + 2: z_pose - 1])
    z_world = float(line[z_pose + 2:])
# Calculate Bildkoordinaten
keypoint = np.array([x_world, y_world, z_world])

# rotating the coordinate system
bildkoordinate = np.dot(drehung,
                        (keypoint - position_camera))

# Perspectiv correction
bildkoordinate_perspektivisch_left = bildkoordinate /
(bildkoordinate[0])*960

# Read Head Bottom Position

if line.find('HeadBottom') != -1:
    x_pose = line.find('X=')
    y_pose = line.find('Y=')

```

```

z_pose = line.find('Z=')
x_world = float(line[x_pose + 2: y_pose - 1])
y_world = float(line[y_pose + 2: z_pose - 1])
z_world = float(line[z_pose + 2:])
# Calculate Bildkoordinaten
keypoint = np.array([x_world, y_world, z_world])
# rotating the coordinate system
bildkoordinate = np.dot(drehung,
                        (keypoint - position_camera))
# Perspektiv correction
bildkoordinate_perspektivisch_left = bildkoordinate /
(bildkoordinate[0])*960

if bildkoordinate_perspektivisch_left[2] != 0
and bildkoordinate_perspektivisch_right[2] != 0
and bildkoordinate_perspektivisch_head_top[2] != 0
and bildkoordinate_perspektivisch_head_bottom[2] != 0:
# TOP COORINATE
top = min(1080/2 - bildkoordinate_perspektivisch_left[2],
          1080/2 - bildkoordinate_perspektivisch_right[2],
          1080/2 - bildkoordinate_perspektivisch_head_top[2],
          1080/2
          - bildkoordinate_perspektivisch_head_bottom[2])
          - clearance_top

# BOTTOM COORINATE
bottom=max(1080/2 - bildkoordinate_perspektivisch_left[2],
           1080/2 - bildkoordinate_perspektivisch_right[2],
           1080/2
           - bildkoordinate_perspektivisch_head_top[2],
           1080/2
           - bildkoordinate_perspektivisch_head_bottom[2])
           + clearance_bottom

# LEFT COORINATE
left =min( bildkoordinate_perspektivisch_left[1] + 1920/2,
           bildkoordinate_perspektivisch_right[1]+ 1920/2,
           bildkoordinate_perspektivisch_head_top[1]
           + 1920 / 2,
           bildkoordinate_perspektivisch_head_bottom[1]
           + 1920 / 2) - clearance_left

# RIGHT COORINATE
right =max( bildkoordinate_perspektivisch_left[1]+1920 /2,
            
```

```

bildkoordinate_perspektivisch_right[1]+1920 /2,
bildkoordinate_perspektivisch_head_top[1]
+ 1920 / 2,
bildkoordinate_perspektivisch_head_bottom[1]
+ 1920 / 2) + clearance_right
"""

WRITE To XML
"""

if bottom >0 and top <1080 and left <1920 and right >0:
    name = ET.SubElement(root, 'id', name=charactername)
    keypoints = ET.SubElement(name, 'Keypoints')
    ET.SubElement(keypoints, 'KeyPoint', Name="LeftShoulder",
                  X=str(int(round(bildkoordinate_perspektivisch
                                   _left[1] + 1920 / 2))), 
                  Y=str(int(round(1080/2 -
                                   bildkoordinate_perspektivisch_left[2]))))

    ET.SubElement(keypoints, 'KeyPoint', Name="RightShoulder",
                  X=str(int(round(bildkoordinate_perspektivisch
                                   _right[1] + 1920 / 2))), 
                  Y=str(int(round(1080/2 -
                                   bildkoordinate_perspektivisch_right[2]))))

    ET.SubElement(keypoints, 'KeyPoint', Name="HeadTop",
                  X=str(int(round(bildkoordinate_perspektivisch
                                   _head_top[1] + 1920 / 2))), 
                  Y=str(round(int(1080/2 -
                                   bildkoordinate_perspektivisch_head_top[2]))))

    ET.SubElement(keypoints, 'KeyPoint', Name="HeadBottom",
                  X=str(int(round(bildkoordinate_perspektivisch
                                   _head_bottom[1] + 1920 / 2))), 
                  Y=str(int(round(1080/2 -
                                   bildkoordinate_perspektivisch_head_bottom[2]))))

if bottom > 1080 or top < 0 or left < 0 or right >1920:
    ET.SubElement(name, 'LeavingImageBoundaries',
                  LeavingImageBoundaries='True')
else:
    ET.SubElement(name, 'LeavingImageBoundaries',
                  LeavingImageBoundaries='False')
    ET.SubElement(name, 'AnnotationType',
                  AnnotationType='groundtruth')

tree = ET.ElementTree(root)
tree.write(storage_location_xml + filename_text +

```

```
    str(filename_number).zfill(4) + '.xml')
filename_number = filename_number + 1
pbar.update(1)
os.remove(storage_location_xml + "export_-001.xml")
```

---