Objekterkennung mit Hintergrundsubtraktion

Bachelorarbeit

an der Fakultät für Informatik und Mathematik
der Universität Passau

Prüfer:

Prof. Dr. Tomas Sauer

Felix Adler
Wintersemester 2019/20

Zusammenfassung

Die Objekterkennung zu präzisieren ist ein bekanntes Problem und eine wichtige Aufgabe für die Sicherheit im Bereich des autonomen Fahrens. Nicht relevante Objekte aus einem Bild zu entfernen kann dabei helfen, Fehler zu minimieren.
Der theoretische Teil dieser Arbeit beschreibt die Hintergrundsubtraktion durch iterative Bestimmung einer Singulärwertzerlegung. Anschließend wird ein Algorithmus vorgestellt, mit welchem diese realisiert wurde. Dadurch ist es möglich, den Hintergrund und Vordergrund für eine Folge von Bildern zu trennen. Nachfolgend wird auf Neuronale Netze sowie Objekterkennung durch Convolutional Neural Networks eingegangen. Im Zuge dessen werden der Aufbau der einfachsten Netze, bis hin zu den obig genannten Netzen, dargestellt, sowie Lernverfahren erläutert.
Im praktischen Abschnitt werden Verfahren implementiert, die es ermöglichen, die Vordergrundbilder, welche aus der Hintergrundsubtraktion gewonnen werden, zu nutzen. Diese Methoden werden dann evaluiert und aufgeschlüsselt. Zudem werden sowohl positive als auch negative Folgen aufgezeigt.
Abschließend wird das Problem des Stehenbleibens von Objekten bei der Hintergrundsubtraktion näher untersucht und die Anwendung des Trackings als Lösung vorgestellt.

Inhaltsverzeichnis

1 Einleitung **1**

2 Hintergrundsubtraktion mittels iterativer Singulärwertzerlegung **3**

2.1 Hintergrundbestimmung mittels Singulärwertzerlegung 3
2.2 Iterative Singulärwertzerlegung 4
2.3 Algorithmus zur Berechnung 4

3 Objekterkennung **7**

3.1 Neuronale Netze 7
3.1.1 Aufbau 7
3.1.2 Lernen im Neuronalen Netz 8
3.1.3 Mehrschichtige Netze 9
3.1.4 Lernen im mehrschichtigen Netz 10
3.2 Convolutional Neural Network 14
3.2.1 Aufbau 14
3.2.2 Detektionsteil 14
3.2.3 Identifikationsteil 16
3.2.4 Lernen im CNN 17
3.3 Objekterkennung auf Beispielszenen 18
3.3.1 Tensorflow 18
3.3.2 Objekterkennung auf Verkehrssituationen der Aschaffenburger Kreuzung 19

4 Anwendung der Hintergrundsubtraktion **21**

4.1 Verfahren 21
4.2 Probleme bei der Hintergrundsubtraktion 25

5 Vergleich **27**

5.1 Unterschiede der Vorgehensweisen 27
5.2 Evaluierung der Vorgehensweisen 27
5.2.1 Labeling 27
5.2.2 Metriken 29
5.2.3 Ergebnisse und deren Aufschlüsselung 31
5.3 Laufzeitverhalten 37

6 Tracking **39**

6.1 Modellierung und Implementierung 39
6.2 Evaluierung 43

7 Fazit **44**

Literaturverzeichnis **45**

Abbildungsverzeichnis **47**

1 Einleitung

Der Traum vom autonomen Fahren beschäftigt die Menschheit schon seit langem. Man lehnt sich als Fahrer zurück und gelangt von alleine an das gewünschte Ziel. Mittlerweile gibt es viele Aufgaben, welche selbstständig durch das Fahrzeug übernommen werden. Dazu zählen Adaptive Cruise Control, Lane Departure Warning, Lane Keeping Assistent und weitere Fahrerassistenzsysteme, welche schon bereits teilweise bei Neuzulassungen verpflichtend sind, weshalb Verkehrsmittel mit vielen Kameras und Sensoren ausgestattet sind.
Das autonome Fahren erfreut sich bereits seit mehreren Jahren großer Aufmerksamkeit in der Forschung und Entwicklung. Läuft alles nach Plan, sollte es in 10 bis 20 Jahren erste vollautomatisierte Fahrzeuge geben, denen es möglich ist, eigenständig in der Stadt zu fahren. Eine große Aufgabe besteht darin, dieses Fahren sicherer zu gestalten. Dabei gilt es nicht nur das Lenken und Bremsen zu optimieren, sondern herauszufinden, wo sich Objekte befinden und welchen Einfluss diese auf die Fahrt nehmen können.
Um Objekte wie Fahrradfahrer, Fußgänger, Tiere aber auch andere Fortbewegungsmittel lokalisieren zu können, wird eine genaue Erkennung dieser benötigt. Es existieren viele Ansätze zur Objekterkennung durch das Auswerten von zum Beispiel akustischen oder optischen Signalen. Damit die Anzahl falsch erkannter Objekte reduziert wird, muss mittels weiterer Verfahren die Erkennung präzisiert werden. Dies ist ein hochaktuelles Thema im Bereich der Informatik und insbesondere bei der Bilderkennung, also was auf dem Bild zu sehen ist, bzw. der Bildverarbeitung. Dabei verwendet eine Objekterkennungssoftware entweder modellbasierte Verfahren oder neuronale Netze.
Ziel dieser Arbeit ist es, darüber Aufschluss zu geben, inwiefern eine Hintergrundsubtraktion helfen kann, eine genauere Objekterkennung zu erreichen. Objekte in einem Bild, welche sich nicht bewegen, und somit meist nicht relevant für den Straßenverkehr sind, sollen dem Hintergrund zugeteilt werden und sich bewegende Objekte dem Vordergrund. Dies wird im Folgenden anhand einer iterativen Singulärwertzerlegung realisiert. Um die Objekterkennung zu unterstützen, werden verschiedene Ansätze getestet. Hier erzielte eine Vordergrundüberprüfung, welche die Existenz eines Objektes nachweist, die besten Ergebnisse in der Evaluierung.
Die Arbeit gliedert sich in fünf Abschnitte. Das erste Kapitel befasst sich mit der Hintergrundsubtraktion mittels einer iterativen Singulärwertzerlegung. Der darauffolgende Teil geht auf Objekterkennung durch neuronale Netze ein. Neuronale Netze werden allgemein vorgestellt sowie auch der spezielle Netztyp, welcher für Objekterkennung genutzt wird. Der dritte Abschnitt erläutert, wie die Hintergrundsubtraktion verwendet wurde. Bei diesen Vorgehensweisen haben sich Vorteile aber auch Nachteile herausgestellt. Im anschließenden vierten Teil werden die drei Verfahren, Objekterkennung auf den Vordergrund, Vordergrund als Maske auf Originalbildern und Vordergrundüberprüfung als Nachweis der Existenz von Objekten ausgewertet und anschließend wird das erfolgreichste näher ausgeführt.
Aus der Evaluierung geht hervor, dass die Hintergrundsubtraktion durchaus helfen kann, die Objekterkennung zu präziseren.

Das abschließende Kapitel der Arbeit handelt von der Lösung des Hauptproblems, der Hintergrundsubtraktion. Hierbei geht es vor allem um das Verschwinden von Objekten, nachdem diese die Bewegung für einen längeren Zeitraum eingestellt haben. Als Schlüssellösung hat sich die Betrachtung eines Trackings herausgestellt.

2 Hintergrundsubtraktion mittels iterativer Singulärwertzerlegung

Hintergrundsubtraktion bezeichnet die Zerlegung eines Bildes in sich bewegende und gleich bleibende Objekte basierend auf vorhergehenden Bildern. Dadurch teilt sich das Bild in zwei Bereiche auf, nämlich den Hintergrund und den Vordergrund. Das Ziel der Hintergrundsubtraktion ist, die Anzahl an „false positive“ Erkennungen (erkannte Objekte, welche falsch klassifiziert werden) zu vermindern, indem irrelevante Teile des Bildes nicht beachtet werden und somit dem Hintergrund zugeteilt werden. Bei einer Verkehrsszene können dies beispielsweise parkende Autos sein. Es soll dabei die Modellierung des Hintergrundes ohne bewegte Objekte erreicht werden.
Ein sehr erfolgreicher Ansatz ist die Anwendung von Convolutional Neuronal Networks (CNN). Der Nachteil dieser Methode ist, dass ein Erfolg sehr von den Daten, die zum Trainieren des neuronalen Netzes verwendet werden, abhängt. Diese müssen sowohl alle möglichen Fälle abdecken als auch sehr umfangreich und ausgewogen sein. Hierfür müssen Daten aufbereitet werden, zum Beispiel durch das Labeln der Bilddaten, indem der Hintergrund annotiert werden muss. Dafür muss für jedes Bild angegeben werden, was dem Hintergrund oder dem Vordergrund zuzuordnen ist. Dies geschieht meist manuell. Im Folgenden wird die Hintergrundsubtraktion mittels der Singulärwertzerlegung beschrieben, bei welcher keine Trainingsdaten oder Ähnliches benötigt werden.

2.1 Hintergrundbestimmung mittels Singulärwertzerlegung

Man betrachte eine Matrix A, welche eine Sequenz aus Bildern darstellt. Dabei ist $A \in \mathbb{R}^{d \times n}$, eine Spalte von A stellt ein Bild mit d Pixel dar und somit ergeben sich n Bilder. Weiter existiert eine Singulärwertzerlegung $A = U\Sigma V$. Für eine Singulärwertzerlegung gilt, dass U eine unitäre und V eine adjungierte unitäre Matrix ist. Die Singulärwerte von A sind die positiven Diagonaleinträge von Σ. Diese sind Invarianten von A. Dabei lässt sich A in den Hintergrund L und den Vordergrund S aufteilen. Der Hintergrund ist dabei definiert als der gleichbleibende Teil über mehrere Bilder. Alle anderen Objekte sind dem Vordergrund zuzuteilen.
Das Optimierungsproblem $\text{rank}(L) + \delta \|S\|_0$ lässt sich nicht einfach lösen, darum betrachtet man die ersten l Spalten von U, also die ersten l Singulärvektoren. Diese gehören dabei zu den l größten Singulärwerten. Außerdem spannt $U_l \coloneqq U_{:,1:l} \coloneqq [u_1, ..., u_l]$ einen Untervektorraum auf. Der Hintergrund eines Bildes J lässt sich dann durch die orthogonale Projektion

$$L = U_l(U_l^T J)$$

auf U_l bestimmen. Der Vordergrund ist folglich die Differenz von Bild und Hintergrund, also

$$S = J - U_l(U_l^T J) = (I - U_l U_l^T)J.$$

2.2 Iterative Singulärwertzerlegung

Ziel der iterativen Bestimmung ist es, die Singulärwertzerlegung rekursiv zu approximieren. Dazu wird für eine gegebene Singulärwertzerlegung $A_k = U_k \Sigma_k V_k^T$ einer bereits bestimmten Matrix $A_k \in \mathbb{R}^{d \times n_k}$ mit $n_k \ll d$ die Matrix A_{k+1} und eine Singulärwertzerlegung $A_{k+1} = U_{k+1}\Sigma_{k+1}V_{k+1}^T$ davon angegeben. Dabei gilt $A_{k+1} = [A_k, B_k]$ mit einer Matrix $B_k \in \mathbb{R}^{d \times m_k}$, $m_k := n_{k+1} - n_k$. Der Updateschritt wird wie folgt vollzogen:

$$\begin{aligned} A_{k+1} &= U_{k+1}\Sigma_{k+1}V_{k+1}^T \text{ mit} \\ U_{k+1} &= U_k Q \begin{bmatrix} \tilde{U} & 0 \\ 0 & I \end{bmatrix} \text{ sowie} \\ V_{k+1} &= \begin{bmatrix} V_k & 0 \\ 0 & I \end{bmatrix} (P\prime_k P_k)^T \begin{bmatrix} \tilde{V} & 0 \\ 0 & I \end{bmatrix} \end{aligned}$$

Dabei erhält man Q aus einer QR-Zerlegung, wobei Q eine orthogonale Matrix und R eine obere Dreiecksmatrix ist. Außerdem ergeben sich $\tilde{U}$ sowie $\tilde{V}$ aus einer Singulärwertzerlegung einer $(r_k + m_k) \times (r_k + m_k)$ Matrix, mit $r_k := rang(A)$. Mit P_k und $P\prime_k$ werden Permutationsmatrizen bezeichnet. In dem nachfolgenden Algorithmus wird die Zerlegung mittels Householdertransformationen bestimmt, die eine stabile Berechnung zulässt.

Adaptive SVD

Um relevante Informationen zu ermitteln, wird ein Threshold Level τ für das Verkleinern der Singulärwertzerlegung festgesetzt. Basierend auf diesem Threshold werden dann die interessanten Informationen der neuen Bildsequenz bestimmt. Die adaptive Singulärwertzerlegung hat den Vorteil, dass die Matrix V im Ausdruck $A = U\Sigma V$ nicht mehr benötigt wird. Dadurch entfällt Rechenleistung und Speicher, der nötig ist, die Matrix zu berechnen und weiter nutzen zu können. Damit man die Singulärwertzerlegung für A_{k+1} bestimmen kann, wird die Matrix $Z := U_k^T B_k$, sowie eine QR-Zerlegung mit Spaltenpivotisierung von $Z_{r_k+1:d,:} = QRP$ betrachtet. Dabei enthält die Matrix R Informationen der neuen Bildsequenz, welche nicht in U_k sind und kann auf τ zusammengeschrumpft werden.

2.3 Algorithmus zur Berechnung

Der Algorithmus zur Berechnung der Singulärwertzerlegung besteht aus zwei Teilen. Der erste Teil, genannt *SVD Comp*, ist für die Initialisierung der iterativen Bestimmung notwendig. Gegeben ist hierfür eine Matrix $A \in \mathbb{R}^{d \times n}$ und eine Spaltennummer l. Berechnet wird dann die beste Rang-l Approximation $A = U\Sigma V^T$, wobei beim Thresholding die Matrix R auf genau diese l Spalten reduziert wird. Dabei wird U_0 (Matrix U der ersten Singulärwertzerlegung) nicht explizit gespeichert, sondern nur die Matrizen $\tilde{U}_0 \in \mathbb{R}^{l \times l}$ und H_o (Matrix H der ersten Singulärwertzerlegung), welche sich aus den Householdervektoren h_j mit $j = 1, ..., l$ zusammensetzt. Die Matrix U_0 lässt sich dann durch

$$U_0 = \tilde{U_0} \prod_{j=1}^{l} (I - h_j h_j^T)$$

bestimmen.
Ein weiterer Bestandteil des Algorithmus ist die Prozedur *SVD Append*. Dabei wird A jetzt nur noch durch $\tilde{U_k}$, Σ_k und H_k beschrieben. Nun wird immer vor der neuen Bestimmung der Singulärwertzerlegung für jede Spalte, also für jedes neu angefügte Bild, mittels eines Thresholds geprüft, ob diese auch relevante Informationen enthält. Die Wahl dieses Thresholds τ ist somit auch ein wichtiger Faktor in der Performance. Ist dieser sehr groß gewählt, ist der Algorithmus zwar schneller, jedoch nimmt er dann auch nur sehr grobe Änderungen wahr. Um das optimale τ zu bestimmen, berechnet man den minimalen Unterschied der Singulärwerte mittels $\hat{i} := \min\{i : \sigma_i - \sigma_{i+1} < \tau^*\}$ und setzt dann $\tau = \sigma_{\hat{i}}$. Hier wird τ^* frei gewählt und bezeichnet den kleinsten Abstand zwischen zwei Singulärwerten, denn die Singulärwerte sind absteigend geordnet. Eine graphische Darstellung der Größe der Singulärwerte und ein Beispiel, wie τ^* gewählt werden kann, ist in [11, Kapitel 5.1] einzusehen.
Mit wachsenden Singulärwerten erhöht sich die Wichtigkeit der Singulärvektoren in U. Durch die zunehmende Größe der Matrizen durch den Appendschritt steigt auch der Speicherbedarf, welcher nötig ist, um diese zu speichern und zu berechnen. Dadurch ist eine Reinitialisierung notwendig. Es werden zwei relevante Ansätze erläutert. Im ersten Ansatz werden Blöcke bis zu einer gegebenen Anzahl μ betrachtet. Ist diese erreicht wird eine *SVD Comp* auf den Hintergrundbildern der Blöcke, welche gespeichert werden, ausgeführt. Der zweite Ansatz beinhaltet ein Thresholding. Dabei ist die Anzahl an Singulärwerten begrenzt, indem die Matrizen U_{k+1} und Σ_{k+1} gekürzt werden. Dadurch existiert eine obere Grenze an Spalten in U_k, weshalb keine Reinitialisierung notwendig ist. Egal welcher Ansatz verfolgt wird, beide haben zur Folge, dass die Größe der Singulärwerte steigt, und somit die Informationen, welche von nachfolgenden Frames beachtet werden, deutlich fehlerhafter werden und nur sehr starke Änderungen beachtet werden, da

$$\sum_{i=1}^{n_k+1} \sigma^2_{n_{k+1},i} \approx \|U_k\Sigma_k\|^2_F + \|B_k\|^2_F$$

gilt.
Um diesen Nachteil zu beheben, ist eine obere Grenze der Singulärwerte notwendig. Diese erhält man, indem man einen *forgetting Faktor* ϕ einführt. Nun wird ein Update auf $[\phi U_k\Sigma_k, B_k]$ angewendet. Dadurch werden aktuelle Frames wichtiger. Es ist außerdem anzumerken, dass der *forgetting Faktor* keinen Einfluss auf die Ordnung der Singulärwerte hat.
Abbildung 1 zeigt die Implementierung des Algorithmus. Als Eingabe erhält das Programm eine Sequenz von Bildern und eine Matrix A, welche die Initialisierungsbilder darstellt. Durch die oben beschriebene *SVD Comp*-Funktion wird die Singulärwertzerlegung für die Matrix A bestimmt. Anschließend werden alle Bilder durchlaufen und der Vordergrund sowie Hintergrund berechnet, indem iterativ jedes Bild gelesen wird. Weiter wird der Hintergrund anhand einer Projektion auf das aktuelle Modell erzeugt. Daraufhin lässt sich der Vordergrund durch Anwendung des Hintergrunds als Maske auf das Bild bestimmen. Danach wird das Bild zu einem Block von Eingabebildern hinzugefügt und nach Überschreitung der Blockgröße ein Update, welches durch die *SVD Append*-

Funktion umgesetzt ist, aufgerufen. Falls die Menge an Singulärvektoren einen Threshold erreicht, wird eine Reinitialisierung angestoßen. [11]

```
Data: Images of a static camera and a matrix A of
      initialization images.
Result: Background and foreground images for every
        input image.
1  U, Σ, î ← SVDComp(A, ℓ, τ*);
2  while there are input images do
3  |   B ← read, vectorize, and normalize the current
   |    image;
4  |   // project B onto the current background model;
5  |   J ← U_{:,1:î}(U^T_{:,1:î} B);
6  |   // subtract the background from B and use this as
   |    mask on the input image;
7  |   F ← B · (|B − J| > θ);
8  |   // build a block of input images;
9  |   M ← [M, B];
10 |   // append a block of images;
11 |   if M.cols == β then
12 |   |   U, Σ, î ← SVDAppend(U, Σ, M, τ*);
13 |   |   M ← [ ];
14 |   end
15 |   // re-initialization if maximum size is exceeded;
16 |   if U.cols > n* then
17 |   |   U, Σ ← SVDComp(UΣ, ℓ);
18 |   end
19 end
```

Abb. 1: *Iterativer Algorithmus zur Bestimmung der Singulärwertzerlegung*

Der Algorithmus wurde in C++ implementiert, wobei die Lineare Algebra-Bibliothek Armadillo verwendet wurde. Diese bietet eine gute Balance zwischen Performance und Nutzung von Berechnungen mit Matrizen und Vektoren und ähnelt bei der Syntax und Funktionalität Matlab. Der Vorteil einer C++ Implementierung gegenüber einer Matlab Implementierung ist dabei vor allem in der Performance zu finden, da hierbei auch eine parallele Version angefertigt wurde, welche dabei hilft, eine noch bessere Performance zu erhalten. [1]

3 Objekterkennung

Dieses Kapitel widmet sich der Objekterkennung mittels neuronaler Netze. Anfangs wird auf neuronale Netze, die Technik sowie die Theorie des maschinellen Lernens eingegangen. Darauffolgend wird die Objekterkennung in einer Beispielszene angewandt.

3.1 Neuronale Netze

Im Folgenden wird der Aufbau von neuronalen Netzen von der einfachsten Form eines künstlichen neuronalen Netz bis hin zu den großen Convolutional Neural Networks erläutert. Außerdem wird ein einfaches und ein komplexeres Lernverfahren dargestellt.

3.1.1 Aufbau

Oftmals wird bei Problemstellungen in der Wissenschaft an Lösungen geforscht, die in der Natur sehr gut funktionieren und seit Millionen von Jahren optimiert sind. Dies ist auch bei dem Aufbau eines künstlichen neuronalen Netzes der Fall. Hier orientiert man sich am menschlichen Gehirn, den Neuronen und Synapsen, welche in kürzester Zeit Informationen aufnehmen und verarbeiten können. Das menschliche Gehirn besteht aus Neuronen, in welchen mehrere Dendriten über eine Verbindung (Axon) zusammentreffen, wie Abbildung 2 zeigt. Ein Dendrit bzw. Axon wird durch eine Synapse am Endknöpfchen mit einem anderen Neuron verbunden, welche das eintreffende chemische Signal in einen elektrischen Impuls umwandelt.

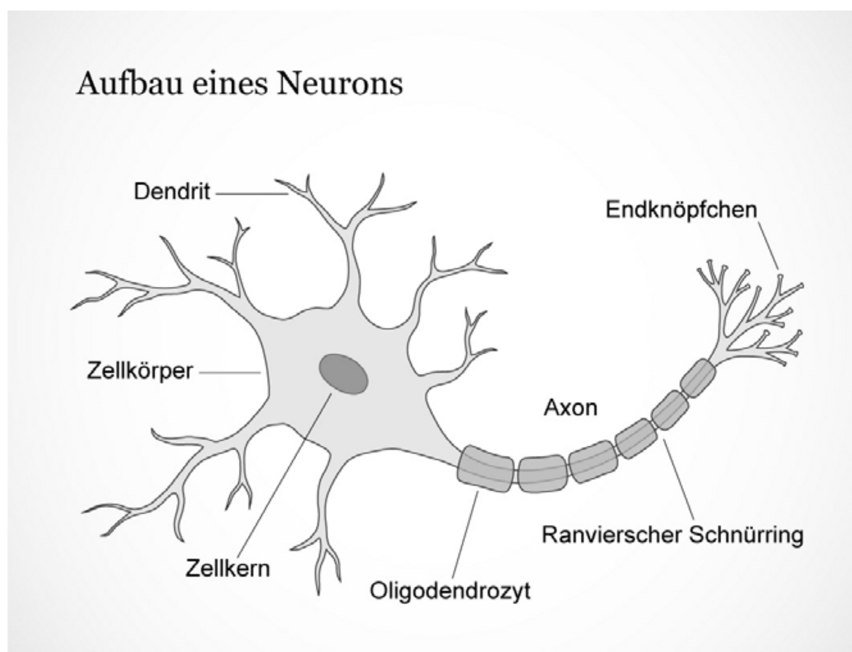

Abb. 2: *Menschliche Nervenzelle*

Ob ein Impuls durch ein Axon an ein anderes Neuron weitergeleitet wird, hängt von einem Schwellwert ab. Ist dieser zu niedrig, wird kein Impuls weitergeleitet [9]. Es wurde versucht, diese Funktionsweise künstlich nachzubauen. Das einfachste Künstliche Neuronale Netz (KNN) ist das Perceptron. Ein Perceptron besteht aus einem Input Vektor $x = \left(x_1, .., x_\mathrm{n}\right)$ sowie einem Gewichtsvektor $w = \left(w_1, ..., w_\mathrm{n}\right)$. Aus diesen Werten wird die gewichtete Summe

$$y = \sum_{i=1}^{n} x_\mathrm{i} \cdot w_\mathrm{i} = x^\mathrm{T} \cdot w$$

berechnet. Ist diese nun kleiner als ein Schwellwert s ist der Output null, anderenfalls eins. Diese Berechnung lässt sich durch die Stufenfunktion

$$output(x) = \begin{cases} 0, & \text{falls } y \leq s \\ 1, & \text{falls } y > s \end{cases}$$

beschreiben. Diese Stufenfunktion simuliert das zuvor genannte „Alles oder nichts“-Gesetz des menschlichen Gehirns.

Um den Threshold nicht mehr als Schwellwert in der Funktion zu haben, sondern den Output immer mit null vergleichen zu können, wurde der sogenannte Bias eines Neuron eingeführt. Dieser wird beschrieben durch $b = -thresh$. Der Bias ist sozusagen der Indikator, wie leicht ein Neuron „feuert“ („Alles oder nichts“-Gesetz). Bei einem hohen Bias gibt das Neuron eher 1 aus. Um nun diesen Bias auch lernbar für das Netz zu machen, wird der Bias an erster Stelle des Gewichtsvektors ergänzt. Folglich ist es jedoch nötig, den Input mit $x_0 = 1$ zu erweitern. Die neuen Vektoren für das Netz sind dann $x = \left(1, x_1, ..., x_n\right)$ sowie $w = \left(b, w_1, ..., w_n\right)$. [12, Kapitel 3] [8, Kapitel 1]

3.1.2 Lernen im Neuronalen Netz

Nun hat man bei einem Netz für eine gewisse Eingabe immer den Inputvektor gegeben. Um den gewünschten Output zu erreichen, benötigt man nun einen Gewichtsvektor, welcher für eine gegebene Problemstellung, wie zum Beispiel eine Klassifizierung, perfekt geeignet ist. Dafür wird ein Vektor oft zufällig initialisiert. Es existieren jedoch auch Strategien zum Initialisieren, wodurch das Lernen des Netzes erleichtert wird. Das Lernen wird nun durch Anpassen des Vektors erreicht. Der Output $\hat{y}$ wird mit dem gewünschten Output y verglichen. Daraufhin wird der Gewichtsvektor angepasst. Durch Wiederholung dieser Prozedur wird $\tilde{y}$ immer besser y approximieren. Es gilt also

$${w_\mathrm{i}}^\mathrm{neu} = {w_\mathrm{i}}^\mathrm{alt} + \Delta w_\mathrm{i} \text{ wobei } \Delta w_\mathrm{i} = (y_\mathrm{i} - \hat{y}_\mathrm{i}) \cdot x_\mathrm{i}. \text{ [12, Kapitel 4]}$$

3.1.3 Mehrschichtige Netze

Durch ein einzelnes Perceptron können einfache logische Funktionen, wie z.B. AND, gelernt werden. Das Perceptron lernt eine Trenngerade, die Abbildung 3 zeigt, um die Punkte A, B, D von C zu trennen. Die verschiedenen Punkte stellen hierbei die Lösungen für das bestimmte Problem dar, wobei Rot gefärbte Punkte Output eins und schwarz gefärbte Output null symbolisieren.
Eine Schwierigkeit des Lernens von Trenngeraden fällt jedoch bei der XOR Funktion auf. Weil das Perceptron nur jeweils eine Trenngerade lernen kann, führt dieses Problem zu mehrschichtigen Netzen, da dies nicht mehr mittels eines Perceptrons und einer Geraden zu lösen ist, wie Abbildung 4 zeigt. Eine Lösung für das Problem ist durch die logische Funktion $(\neg x_1 \wedge x_2) \vee (x_1 \wedge \neg x_2)$ beschrieben.

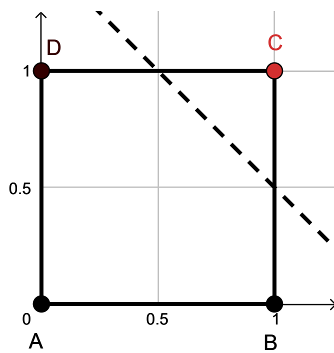

Abb. 3: *Gelernte Trenngerade beim AND Problem*

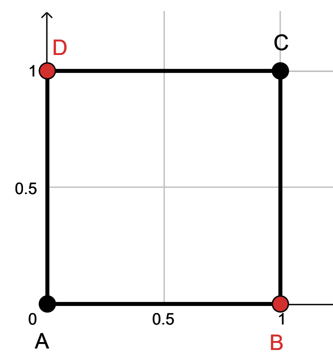

Abb. 4: *Das XOR Problem*

Dies wird nun erreicht, indem mehrere Perceptren, welche die einzelnen logischen Funktionen berechnen, hintereinander geschaltet werden und so zwei Trenngeraden lernen (siehe Abbildung 6). Perceptron 1 und Perceptron 2 lernen nun beide jeweils eine dieser Trenngeraden. Tabelle 1 zeigt die Inputs und gewünschten Outputs der beiden Perceptren. Abbildung 5 zeigt diese nochmal graphisch. Betrachtet man nun beide Outputs y_1 und y_2, so lässt sich erkennen, dass es nun nur noch drei Kombinationen gibt, also Punkte, die getrennt werden müssen. Dies zeigt Abbildung 7. Der Input (der Output der beiden ersten Perceptren) und der gewünschte Output ist in Tabelle 2 dargestellt. Zusammenfassend zeigt Tabelle 3, dass das konstruierte Netz das XOR Problem löst. Dies war ein entscheidender Schritt in der Entwicklung von Neuronalen Netzen, da nun komplexere logische Problemfälle gelöst werden konnten. Dies alles führte zu den mehrschichtigen Netzen. [12, Kapitel 5]

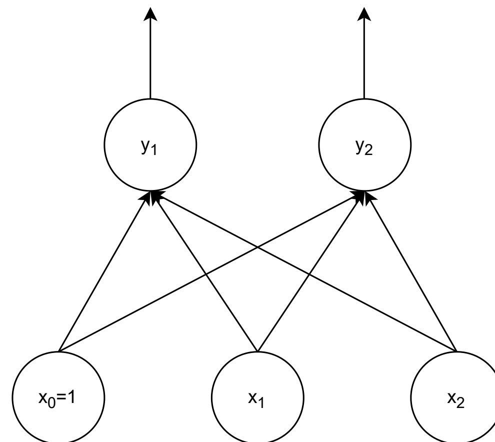

Abb. 5: *Beide Perceptren für das XOR Problem*

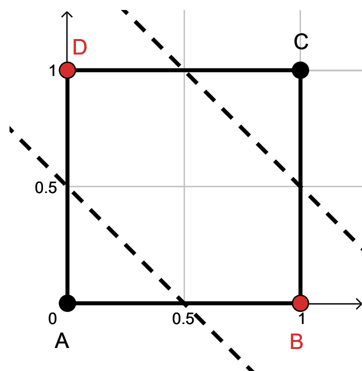

Abb. 6: *Zwei Trenngeraden*

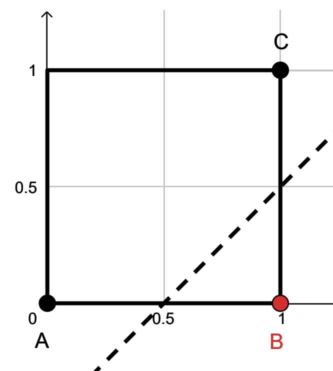

Abb. 7: *Trenngerade von Perceptron 3*

x_1	x_2	y_1	y_2
0	0	0	0
1	0	1	0
0	1	1	0
1	1	1	1

Tab. 1: *Input und Output der beiden Perceptren*

y_1	y_2	y_3
0	0	0
1	0	1
1	0	1
1	1	0

Tab. 2: *Kombination der beiden Outputs und benötigtes Ergebnis*

x_1	x_2	y_1	y_2	y_3
0	0	0	0	0
1	0	1	0	1
0	1	1	0	1
1	1	1	1	0

Tab. 3: *Zusammenfassung XOR*

3.1.4 Lernen im mehrschichtigen Netz

Das Lernen im mehrschichtigen Netz ist komplexer als das des Perceptrons, da mehr Gewichte existieren, welche angepasst werden müssen. Der Fehler des Outputs ist ein Maß dafür, wie die Gewichte verändert werden müssen. Das Ziel des Lernens ist, den Fehler gegen null konvergieren zu lassen. Dafür sind sehr viele Übungsdaten notwendig, welche auch dementsprechend aufbereitet sein müssen. Doch ein sehr großer Nachteil ei-

nes Perceptrons ist der harte Output null oder eins. Besser ist eine leichte Veränderung zwischen null und eins, da das Lernen erschwert wird, wenn der Output bei einem gewissen Schwellwert auf den gegenteiligen Output schaltet. Dies erfüllt das Sigmoid-Neuron. Dadurch wird ein weicher Übergang geschaffen und so vermieden, dass eine kleine Anpassung der Gewichte eine sehr große Auswirkung auf die Ausgabe hat, wodurch zwar dann oft eine korrekte Ausgabe für eine bestimmte Eingabe geschaffen würde, jedoch für eine andere Eingabe die Gewichte wiederum falsch sein können. Durch das Anpassen der Neuronen kann man sich nun an optimale Gewichte annähern. Dadurch haben dann kleine Änderungen in den Gewichten auch kleine Änderungen am Output zur Folge. Das Sigmoid-Neuron benutzt die Sigmoid-Funktion (siehe Abbildung 8)

$$o(z) = \frac{1}{1 + e^{-z}}$$

als Aktivierungsfunktion. Somit lässt sich der Neuronoutput für einen Eingabevektor $x = (x_1, ..., x_n)$ und einem Gewichtsvektor $w = (w_1, ..., w_n)$ folgendermaßen berechnen:

$$y = \frac{1}{1 + exp(-\sum_{i=1}^{n} x_i \cdot w_i)}$$

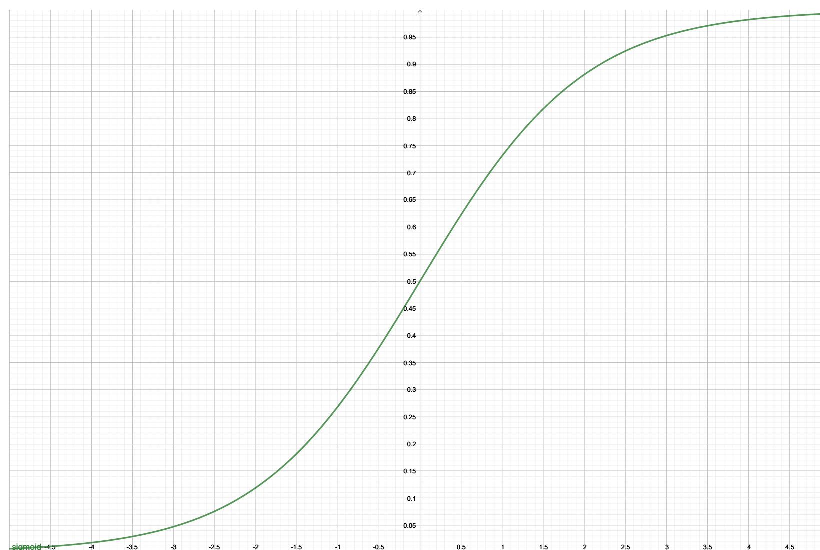

Abb. 8: *Sigmoid Funktion*

Die Ausgabe eines Netzes von Simgoid-Neuronen ist ein Wert zwischen null und eins. Wenn das Netz nun zum Beispiel die Zahl neun erkennen soll, lässt sich die Ausgabe für ein Sigmoid-Neuron als Wahrscheinlichkeit interpretieren und falls diese größer als 0.5 ist, ist die Eingabe als neun zu werten.
Damit man ein mehrschichtiges Netz trainieren kann, stellt sich die Herausforderung, alle Gewichtsvektoren anzupassen. Um den Fehler zu messen, der durch jedes Neuron bei einer falschen Ausgabe beigetragen wird, wurde der Backpropagation-Algorithmus entwickelt. [12, Kapitel 5]

Backpropagation-Algorithmus

Der Backpropagation-Algorithmus ist die bekannteste und auch die gängigste Methode, um ein Netz zu trainieren. Das Netz passt nach jedem Input die Gewichte an. Der Algorithmus berechnet den Fehler anhand einer Fehlerfunktion bzw. Kostenfunktion für jedes Neuron in Abhängigkeit seines Inputs und der Gewichte. Außerdem beachtet der Algorithmus wie viel ein Neuron zu einem Fehler beiträgt, also wie hoch die Gewichte des Neurons sind. Wie im Folgenden das Netz aussieht, beschriftet ist und worauf sich die folgenden Vektoren und Matrizen beziehen, zeigt Abbildung 9, welche ein Beispiel für ein Teilnetz darstellt.

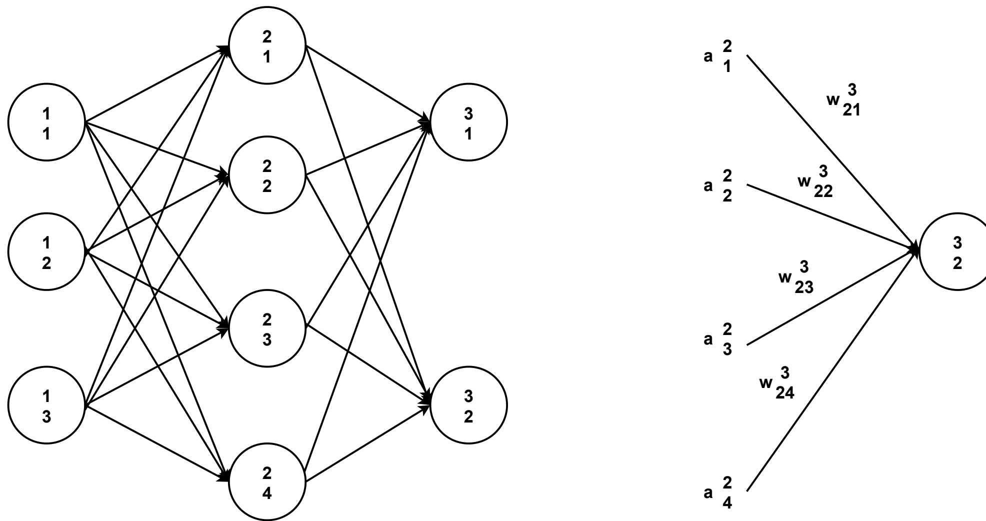

Abb. 9: *Teilnetz mit Beschriftungen*

Der Output für ein Neuron j in der Schicht l lässt sich wie folgt darstellen:

$$a_j^l = \sigma(\sum_k w_{jk}^l \cdot a_k^{l-1})$$

Das Neuron l,j hat als Inputvektor $a^{l-1} = \left(1, a_1^{l-1}, ..., a_k^{l-1}\right)$ und als Gewichtsvektor $w_j^l = \left(b_j^l, w_{j1}^l, ..., w_{jk}^l\right)$, wobei k die Anzahl an Neuronen im Layer $l-1$ und σ die Aktivierungsfunktion des Neurons ist. Der Outputvektor für eine Schicht l ist somit $a^l = \sigma(w^l a^{l-1})$, außerdem wird $z^l = w^l \cdot a^{l-1}$ als gewichteter Input für Layer l definiert, wobei sich dieser aus den gewichteten Inputs

$$z_j^l = \sum_k w_{jk}^l \cdot a_k^{l-1}$$

für jedes Neuron j zusammensetzt.
Als Kostenfunktion wird nun die quadratische Kostenfunktion

$$C = \frac{1}{2n} \sum_x ||y(x) - a^L(x)||^2$$

verwendet, wobei $y(x)$ der gewünschte Output für einen Input x, a^L der Vektor von aktivierten Outputs und n die Anzahl an Trainingsdaten ist. Außerdem lässt sich die Durchschnittskostenfunktion C als $C = \frac{1}{n}\sum_x C_x$ und die Kostenfunktion C_x für einen speziellen Trainingsinput x als

$$C_x = \frac{1}{2}||y - a^L||^2$$

darstellen. Im Folgenden wird die Funktion nur auf einen einzelnen Trainingsinput x bezogen und der Index weggelassen.
Weiter sind die Kosten für ein Netz durch

$$C(a^L) = \frac{1}{2}||x - a^L||^2 = \frac{1}{2}\sum_j (y_j - a_j^L)^2$$

gegeben. Der Fehler eines Neurons l,j ist definiert durch δ_j^l. Dabei gilt $\delta_j^l = \partial C / \partial z_j^l$. Außerdem beschreibt δ^l den Errorvektor des Layer l. Daraus folgt, dass der Fehler für das letzte Layer L durch $\delta_j^L = \partial C / \partial a_j^l \sigma'(z_j^L)$ gegeben ist.
Hierbei gilt $\partial C / \partial a_j^l = (a_j^L - y_i)$ und somit folgt

$$\delta^L = \nabla_a C \circ \sigma'(z^L) = (a^L - y) \circ \sigma'(z^L).$$

Dabei stellt $\partial C / \partial a_j^l$ dar, wie schnell sich die Kostenfunktion in Abhängigkeit des j-ten Outputs im Layer L ändert und $\sigma'(z^L)$ der Änderungsrate der Aktivierungsfunktion für die gewichtete Eingabe z_j^L entspricht.
Der Fehler der vorher liegenden Schichten wird nun in Abhängigkeit zur zuvor errechneten Fehlerrate mittels

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \circ \sigma'(z^l)$$

bestimmt.
Die Änderungsrate, bezogen auf jedes Gewicht im Netz, lautet dann $\partial C / \partial w_{jk}^l = (a_k^{l-1} \delta_j^l)$.
Der Algorithmus sieht nun folgende Schritte vor:

1. Input x. Berrechne a^1 für die Eingabe Schicht.
2. Feedforward:
   Für jedes weitere Layer $l = 2, ..., L$ berechne $z^l = w^l \cdot a^l - 1$, mit $a^l = \sigma(z^l)$.
3. Berechnung des Outputfehlers $\delta^L = \nabla_a C \circ \sigma'(z^L)$.
4. Berechnung des Backpropagationfehlers durch Zurückrechnen des Fehlers für jedes Layer $l = L - 1, ..., 2$ mittels:
   $\delta^l = ((w^{l+1})^T \delta^{l+1)} \circ \sigma'(z^l)$
5. Output des Algorithmus: Gradient der Kostenfunktion $\partial C / \partial w_{jk}^l = (a_k^{l-1} \delta_j^l)$.

Das Anpassen der Gewichte in jedem Layer folgt dann nach folgender Regel:

$$w^l \rightarrow w^l - y \cdot \delta^l \cdot (a^{l-1})^T.$$

[12, Kapitel 6] [8, Kapitel 2] [5, Kapitel 6]

3.2 Convolutional Neural Network

Convolutional Neural Networks (CNN) sind spezielle mehrschichtige Netze, welche für maschinelles Lernen bei Bild- und Audiodateien entwickelt und eingesetzt werden. Für kleine Bilder wäre es auch möglich, normale mehrschichtige Netze zu verwenden. Der große Inputvektor stellt hierbei aber ein Problem dar, denn je größer das Bild ist, desto größer ist die Anzahl an Neuronen, welche benötigt werden. Das zweidimensionale Bild wird als eindimensionaler Vektor dargestellt, wodurch jedoch Informationen über benachbarte Pixel verloren gehen. Bei der Entwicklung des CNNs orientiert man sich am menschlichen optischen Apparat, dem Auge. Dabei wurde der visuelle Kortex, welcher ein Teil im Gehirn ist, als Neuronen dargestellt. Die Neuronen sollen, wie im Gehirn, nur auf bestimmte Teilbereiche in einem Bild reagieren.

3.2.1 Aufbau

Das Convolutional Neuronal Network setzt sich aus zwei verschiedenen Schichten, die alle einen unterschiedlichen Aufgabenbereich haben, zusammen. Der erste Teil ist der sogenannte Detektionsteil, welcher sich aus dem Convolutional Layer, notwendig zur Faltung und Aktivierung, und dem Pooling Layer zusammensetzt. Durch diese verschiedenen Schichten ist eine Extraktion komplexer Eigenschaften des Bildes möglich. Der letzte Teil besteht aus einem Identifikationsteil, der oft durch ein herkömmliches mehrschichtiges Netz realisiert ist. Dieser klassifiziert den Output des Detektionsteils.

3.2.2 Detektionsteil

Der Detektionsteil, der oft mehrmals hintereinander geschaltet ist, bildet den ersten Teil eines CNNs. Durch diesen können einfache Strukturen wie Linien, Punkte oder Kanten im Bild erkannt werden. Im darauffolgenden Detektionsteil werden daraus dann komplexere Verbindungen, die wiederum durch die oben genannten Strukturen zusammengesetzt sind. Dieser Vorgang wiederholt sich dann, wodurch noch komplexere Strukturen beschrieben werden, solange bis ein Detektionsvektor übrig bleibt, welcher das Bild in seinen vollen Strukturen beschreibt.

Convolutional Layer

Die erste Convolutional-Schicht enthält eine Menge von Neuronen, wobei jedes Neuron auf einen kleinen Teil des Bildes reagiert. Dies ist meist ein Bildausschnitt von 5x5 oder 3x3 Pixel. Diese Faltungsmatrix wird über das gesamte Bild bewegt, wobei jede Faltungsmatrix auf unterschiedliche Muster reagieren. Die Convolutional Schicht besteht wiederum aus mehreren Ebenen, welche auf diese Muster reagieren und auch Feature Maps genannt werden. Somit wird jede Umgebung eines Pixel untersucht. Abbildung 10 zeigt die Abfolge von Convolutional-Schicht und dem Pooling.

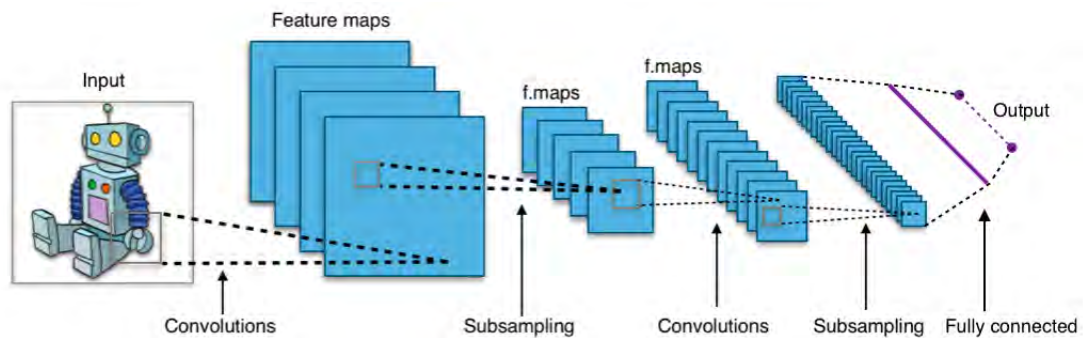

Abb. 10: *Aufbau eines CNN*

Als Aktivierungsfunktion der Neuronen im CNN wird meist die ReLU Funktion

$$fakt(c) = f_{ReLU}(c) = max(0, c)c$$

verwendet. Diese zeigt Abbildung 11.

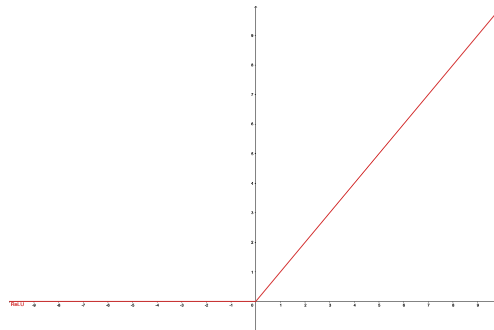

Abb. 11: *ReLU Funktion*

Pooling

Der Pooling Layer hat in erster Linie die Aufgabe, die Dimension des Bildes in x- und in y- Richtung zu minimieren. Es existieren verschiedene Verfahren wie dies geschehen kann. Beim Max Pooling wird in einem Filterbereich von 2x2 bzw. 3x3 jeweils der Wert vom maximalen Pixelwert übernommen. Beim Average Pooling wird der Durchschnittswert aller Pixel im Filter berechnet. Ein Beispiel des Max Pooling zeigt Abbildung 12.

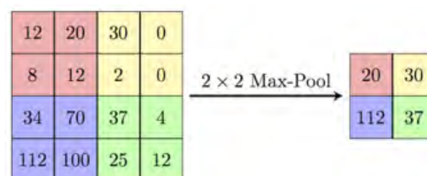

Abb. 12: *Max Pooling Beispiel mit 2x2 Filter*

Das Pooling hat den Vorteil, dass so der Rechenaufwand in darauffolgenden Schichten reduziert wird. Ein Problem, welches immer in der Bildverarbeitung präsent ist und so auch beim Pooling, ist die Behandlung von Randpixel. Dies wird mit dem Ausfüllen von fehlenden Pixel umgangen, was man als Padding bezeichnet. Ob sich die Filter beim Pooling überlappen, ist durch die Schrittlänge gegeben. Im Convolutional Layer wird meist mit Überlappung gearbeitet. Beim Padding gibt es ebenfalls unterschiedliche Verfahren, nämlich das Constant Padding (was auch als Zero Padding bezeichnet wird), das Symmetric Padding und das Reflect Padding. Alle drei Verfahren, die in der Bibliothek Tensorflow zur Verfügung stehen, sind in Abbildung 13 dargestellt.

Abb. 13: *Padding-Verfahren*

3.2.3 Identifikationsteil

Der Identifikationsteil ist in der Regel ein vollständiges Neuronales Netz, um die Klassen im Bild zu identifizieren. Der Output des Pooling Layer muss zu einem Vektor umgeformt werden, da das Netz als Input einen Vektor erwartet. Der Output der vorletzten Schicht ist ein Wahrscheinlichkeitsvektor. Dieser hat die Dimension der zu erkennenden Klassen. Pro Klasse wird dann eine Wahrscheinlichkeit angegeben. Der sogenannte Logits Layer gibt dann als Ausgabe die wahrscheinlichste Klasse, zu der das Objekt gehört, aus. Hierfür wird meist die Softmaxfunktion

$$f_{softmax}(c_j) = \frac{e^{c_j}}{\sum_{i=1}^{k} e^{c_i}}$$

zur Aktivierung verwendet.

3.2.4 Lernen im CNN

Durch die Größe eines CNN besteht das Lernen aus mehreren Herausforderungen. Allgemein wird für das Lernen der Backpropagation-Algorithmus verwendet. Durch die Größe entsteht das Problem der Fehleranpassung, da oftmals entweder extrem starke bzw. extrem kleine Fehlergradienten berechnet werden. Dies hat zur Folge, dass eine besser geeignete Aktivierungsfunktion gefunden werden muss, wodurch man von der Sigmoid zur oben genannten ReLU Funktion wechselt. Dadurch wurde dieses Problem so weit wie möglich verhindert.
Eine weitere Herausforderung ist das Overfitting. Dies tritt auf, wenn das Netz zum Beispiel zu lange auf einem Trainingsdatensatz trainiert wird, ein Datensatz nur einen Sonderfall oder nicht genug ausbalanciert ist, wodurch das Netz zwar optimal auf diesem Datensatz funktioniert, aber bei anderen Datensätzen wieder höhere Fehlerraten auftreten. Durch Early Stopping kann jedoch dagegen vorgegangen werden. Man stoppt also den Trainingsvorgang bereits bevor man eine optimal kleine Fehlerrate für den Trainingsdatensatz erreicht hat. [12, Kapitel 7] [8, Kapitel 6] [5, Kapitel 9]

3.3 Objekterkennung auf Beispielszenen

Im Folgenden wird der Objekterkenner, der mittels eines vortrainierten Netzes umgesetzt wurde, knapp beschrieben.

3.3.1 Tensorflow

Um Objekte erkennen zu können, wird ein künstliches Netz, ein CNN, das vorher bereits beschrieben wurde, benötigt. Ein solches Netz lässt sich mit Tensorflow realisieren. Tensorflow ist eine Open Source Software Bibliothek, welche von Google entwickelt wurde und der Nachfolger des ersten Tools „DistBelief“ ist. Aufgrund der Freischaltung der Bibliothek für alle Nutzer wird die Forschung und Weiterentwicklung von Tensorflow beschleunigt, wodurch sich die Technologie viel schneller entwickelt und verbessert. Ein weiterer Vorteil ist die Lauffähigkeit auf vielen Endgeräten wie Smartphones oder Tablets, wodurch Tensorflow in vielen Bereichen seinen Einsatz findet. Dies geschieht einerseits in der Medizin oder auch im Finanzsektor, aber andererseits vor allem auch in Google internen Produkten wie Google Bilder oder Google Maps zur Klassifizierung von Bildern und Ähnlichem. [4]
Mittels Tensorflow können eigene Modelle für Neuronale Netze entwickelt werden. Diese müssen jedoch eigens trainiert werden, wofür ein riesiger Datensatz vorhanden und auch gelabelt sein muss. Es existieren aber auch vortrainierte Modelle, welche auf unterschiedlichsten Datensätzen trainiert sind. Datensätze, auf welchen trainierte Modelle zur Verfügung gestellt werden, sind zum Beispiel der Coco Datensatz,[1] der Kittie Datensatz [2] oder der INaturlist Species Detection [3].
Für das nachfolgende Experiment wurde ein Netz verwendet, das auf dem Coco Datensatz vortrainiert wurde. Dies wurde aufgrund der unabhängigen Ergebnisse gemacht, welche bei einem eigenständig trainierten Netz auf Daten der Beispielkreuzung nicht mehr gegeben gewesen wäre. Beim Training auf die Straßenkreuzung, auf welcher das Experiment durchgeführt wurde, wären die Ergebnisse nicht mehr aussagekräftig für zufällige Kreuzungen gewesen. Man hat sich für ein Netz, das auf dem Coco Datensatz trainiert wurde, entschieden, da Coco eine der größten gelabelten Bild- und Videodatenbanken mit einer Vielzahl an Informationen ist.
Diese Netze sind sehr aufwendig trainiert. Außerdem umfasst die Datenbank mehr als 200.000 gelabelte Bilder mit über 1.5 Millionen Objekten. Das Netz *faster_rcnn_resnet_101_coco*[4] wird verwendet, da ein Netz benötigt wurde, welches nicht zu langsam ist, da sonst die Performance für die Erkennung zu schlecht ist, um im aktiven Straßenverkehr verwendet zu werden. Andererseits darf das Netz aber auch nicht allzu unpräzise sein (bezogen auf die mean Average Precision (mAP)). Außerdem wurde für die Implementierung und Auswertung eine Erkennung mit Objekten umgeben von Boxen bevorzugt, um

[1] http://cocodataset.org/#home
[2] http://www.cvlibs.net/datasets/kitti/
[3] https://github.com/visipedia/inat_comp/blob/master/2017/README.md
[4] http://download.tensorflow.org/models/object_detection/faster_rcnn_resnet101_coco_2018_01_28.tar.gz

die Evaluierung nicht zu aufwändig zu gestalten. Um all dies zu vereinen, wurde eben genau dieses Netz ausgewählt. Das Netz bietet mit 106 ms (bezogen auf Erkennung mit GPU Nvidea GeForce GTX Titan X) und 32 mAP gute Mittelwerte und ist nicht zu ungenau, aber auch nicht zu langsam.

3.3.2 Objekterkennung auf Verkehrssituationen der Aschaffenburger Kreuzung

Der Objekterkenner wurde in Python implementiert. Dieser wird iterativ auf Bilder, welche von einer Kamera, die an einer Kreuzung in Aschaffenburg aufgrund des DeCoInt2 Projektes in Zusammenarbeit mit FORWISS aufgenommen wurden, angewandt. Das Detecting Intentions of Vulnerable Road Users Based on Collective Intelligence wurde von der Deutschen Forschungsgemeinschaft ins Leben gerufen, um den Straßenverkehr für VRUs sicherer zu gestalten und legt auch eine Grundlage für das autonome Fahren. Durch Kollektive Intelligenz soll vorhergesagt werden, wie sich ein VRU fortbewegen wird. [10]
Es existiert ein großer Datensatz von Aufzeichnungen an verschiedenen Tagen, an denen der Verkehr auf der Kreuzung gefilmt wurde. Es wurden im Zuge der Arbeit vier verschiedene Sequenzen ausgewählt, mit welcher der Objekterkenner und die Hintergrundsubtraktion evaluiert werden. Durch das Python-Programm werden Boxen, die ein Objekt beschreiben, das durch den Objekterkenner gefunden wurde, erfasst und sowohl deren Position als auch die Klasse des erkannten Objekts gespeichert. Anschließend werden die gefundenen Objekte mit den gelabelten Objekten verglichen. Das Labeln der Objekte wurde mittels des Tools CVAT vorgenommen. Näheres zur Auswertung und zum Labeling wird in Kapitel 5 beschrieben.

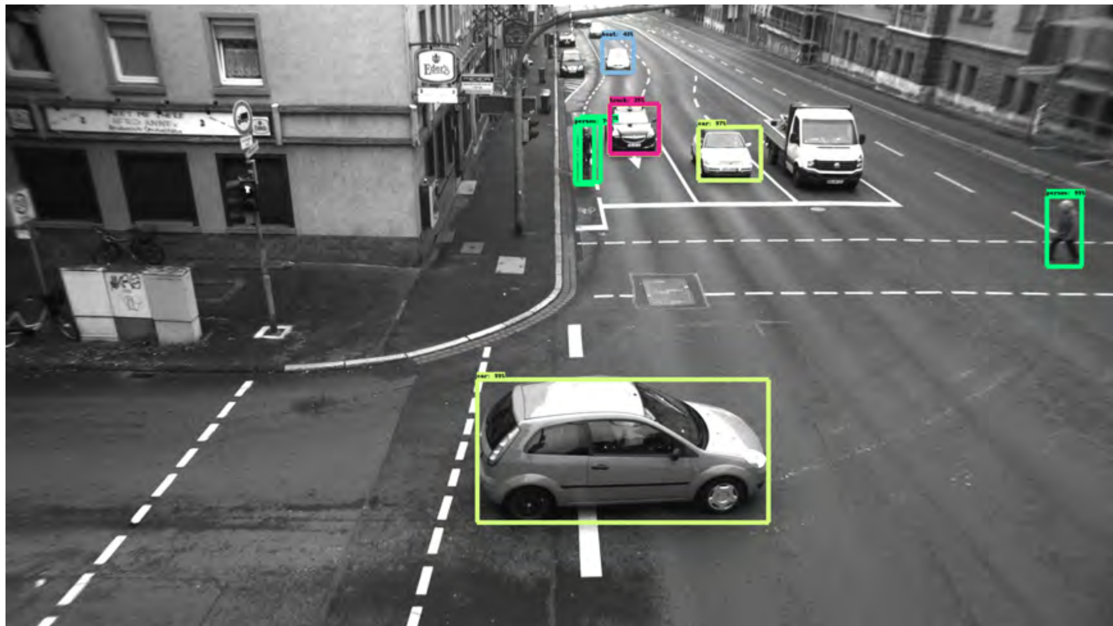

Abb. 14: *Ausgabebild des Objekterkenners*

Die Ausgabe des Objekterkenners wurde einerseits angepasst, sodass die erkannten Objekte in einem Format, welches dann zur Evaluierung geeignet ist, ausgegeben werden. Andererseits können die Bilder mit Boxen der erkannten Objekte gespeichert werden. Ein Beispiel zeigt Abbildung 14, in der die erkannten Objekte sichtbar sind. Zu jedem Objekt ist nicht nur die Klasse angegeben, sondern auch mit welcher Wahrscheinlichkeit das erkannte Objekt exakt dieser Klasse angehört.

4 Anwendung der Hintergrundsubtraktion

Um die Objekterkennung zu präzisieren und die Anzahl an Fehler zu minimieren, wird nun die Hintergrundsubtraktion mittels einer Singulärwertzerlegung (SVD) (siehe Kapitel 2) als Vorverarbeitungsschritt bzw. Nachbearbeitungsschritt genutzt. Die Idee dieser Hintergrundsubtraktion ist es, unbewegte Objekte zu ignorieren und diese aus dem Bild herauszufiltern. Dadurch existieren weniger Objekte, die falsch erkannt werden können. Im Laufe dieses Kapitels wird beschrieben, wie die Hintergrundsubtraktion genutzt wurde und welche Probleme sich dadurch ergeben haben.

4.1 Verfahren

Es werden im Folgenden drei Ansätze zur Präzisierung der Objekterkennung durch Hintergrundsubtraktion erläutert.

Vordergrundbilder als Input für den Objekterkenner

Im ersten Schritt wurde versucht, den Output der Hintergrundsubtraktion, also den Vordergrund, durch das Neuronale Netz zu überprüfen und Objekte zu erkennen. Der direkte Output der Hintergrundsubtraktion, also die schwarz-weißen Bilder, werden dem Objekterkenner übergeben. Am Ende der Hintergrundsubtraktion folgt ein Postprocessing, bei welchem der Hintergrund vom Bild entfernt wird. In diesem Postprocessing wird ein Thresholding durchgeführt nachdem der Hintergrund vom Gesamtbild abgezogen worden ist. Dadurch ergibt sich als Ausgabe ein schwarz-weißes Bild.
Der Nachteil dieser Methode ist gerade das schwarz-weiße Bild und das Thresholding, da so wichtige Informationen wie Kanten und Abstufungen der Farbwerte für den Objekterkenner verloren gehen (siehe Abbildung 15). Um dem ein wenig entgegenzuwirken, ist das Postprocessing geändert worden. Wie Codeausschnitt 1 zeigt, wird zuerst der Vordergrund bestimmt und dann nochmal mit dem Graustufenbild multipliziert. Da diese Matrizen für die Farbe Weiß den Wert 1 tragen, wird für das Ausgabebild noch mit 255 multipliziert. Ein Beispielbild zeigt Abbildung 16.

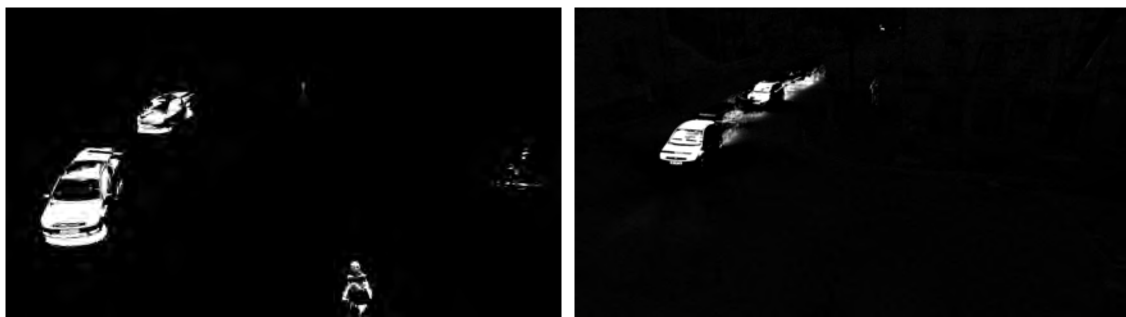

Abb. 15: *Ausgabe der Hintergrundsubtraktion*

Abb. 16: *Ausgabe Hintergrundsubtraktion mit Graustufenwerten*

Codeausschnitt 1: *Postprocessing mit Graustufenwerte*

```
arma::mat bg = sing_vecs * (arma::trans(sing_vecs) * currentimage);

// thresholding
arma::mat fg = arma::conv_to<arma::mat>::from(arma::abs(currentimage - bg
    ) > output_thres);

// grey
arma::mat mask = currentimage % fg;
fg *= 255;
```

Diese Methode stellt sich jedoch immer noch als suboptimal heraus, da der Objekterkenner weiterhin mit verlorenen Kanten und Farbwerten zurecht kommen muss. Ein Beispielbild zeigt Abbildung 17, bei dem ein Artefakt als Objekt erkannt wurde.

Abb. 17: *Artefakt als Objekt erkannt*

Vordergrundbilder als Maske für Originalbilder

Ein weiterer Ansatz ist die Anwendung der Vordergrundbilder als Maske auf die Originalbilder, denn so gehen Farbwerte und auch Konturen der Objekte nicht verloren. Codeausschnitt 2 zeigt den Pythoncode, um mit einer Maske aus schwarz-weißen Werten ein Bild mit bewegten Objekten zu generieren, das dann dem Objekterkenner übergeben wird. Mittels der „bitwise and"-Funktion werden nur die Pixel des Originalbildes übernommen, bei der im Vordergrund ein weißes Pixel, also der Wert 255, vorhanden ist. Ein Beispielbild zeigt Abbildung 18.

Codeausschnitt 2: *Vordergrund als Maske*

```
1 def applieMask(image, mask, imageName):
2     img = cv2.imread(image)
3     mask = cv2.imread(mask)
4 
5     res = cv2.bitwise_and(src1=img, src2=mask)
6     cv2.imwrite(os.path.join(appliedMaskOutput, imageName), res)
```

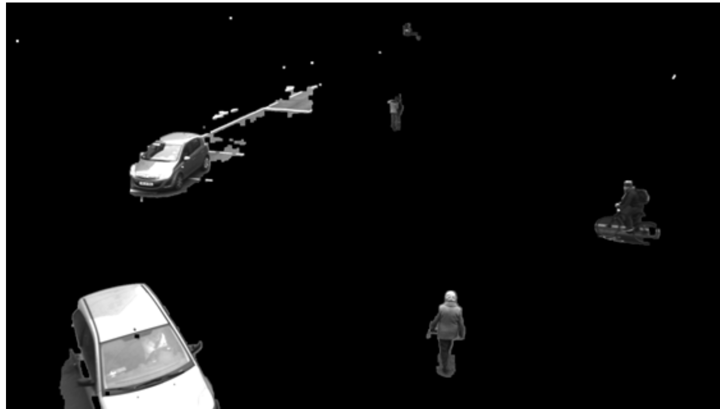

Abb. 18: *Bild nach Anwendung der Maske*

Für dieses Verfahren wurde das Postprocessing ebenfalls noch einmal angepasst. Wie Codeausschnitt 3 zeigt, werden morphologische Funktionen angewandt, um bessere Flächen für die Masken zu schaffen.

Codeausschnitt 3: *Postprocessing mit morphologischen Funktionen*

```
int morph_size = 5;
cv::Mat kernel_rect = cv::getStructuringElement(cv::MORPH_RECT,
        cv::Size( 2*morph_size + 1, 2*morph_size+1 ), cv::Point(
            morph_size, morph_size) );
cv::Mat kernel_rect2 = cv::getStructuringElement(cv::MORPH_RECT,
        cv::Size( 3*morph_size + 1, 3*morph_size+1 ), cv::Point(
            morph_size, morph_size) );
cv::Mat morph;

cv::dilate(im_out, morph, kernel_rect);
im_out = morph;
cv::morphologyEx(im_out, morph, cv::MORPH_CLOSE, kernel_rect2);
im_out = morph;
```

Anhand von Dilation und Closing werden größere zusammenhängende Flächen erzeugt. Dilation beschreibt eine morphologische Basisoperation in der digitalen Bildverarbeitung. Diese wird mittels einem strukturierenden Element angewandt. Das Element ist eine Matrix, in der das Element mittels Einsen und Nullen gebildet wird. Dieses kann rechteckig, aber auch kreisförmig oder ähnliches sein. Das Pixel, im Originalbild betrachtet, befindet sich dann im Zentrum des Elements. Das Hauptziel der Dilation ist, das Objekt (weiß) zu vergrößern. Hierfür werden Pixel an den Grenzen des Objektes hinzugefügt. Dadurch erhöht sich die Anzahl an weißen Pixel und es verringert sich die Anzahl an schwarzen Pixel. Ein Beispielbild, wie Dilation funktioniert, zeigt Abbildung 19. Ein strukturierendes Element kann zum Beispiel wie folgt aussehen:

$$B = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

Die Dilation ist mathematisch definiert durch $A \oplus B = \{z|(\hat{B})_z \cap A \neq \emptyset\}$. Das Gegenteil der Dilation ist die Erosion. Diese ist durch $A \ominus B = \{z|(B)_z \subseteq A\}$ definiert.

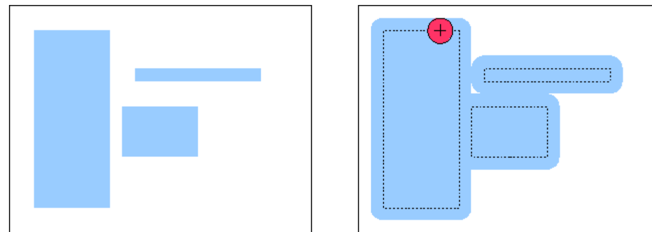

Abb. 19: *Anwendung von Dilation mit einem runden strukturierten Element*

Das morphologische Closing

$$A \cdot B = (A \oplus B) \ominus B$$

ist der Verbund von Dilation und Erosion.
Also wird bei Closing zuerst eine Dilation und dann eine Erosion vorgenommen. Das Hauptziel von Closing ist das Eliminieren kleiner Löcher und damit das Füllen von Lücken in den Objektgrenzen. Es werden dabei keine Bildpunkte gelöscht, sondern nur hinzugefügt. Die Funktionsweise von Closing ist in Abbildung 20 dargestellt. [14]

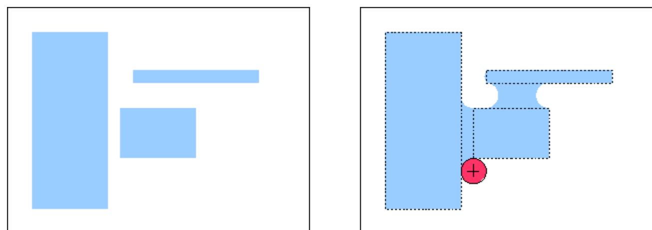

Abb. 20: *Anwendung von Closing mit einem runden strukturierten Element*

Ein Problem, das morphologische Operationen mit sich bringen, ist, dass Artefakte noch größer werden und damit weiterhin ein Problem für den Objekterkenner bleiben. Dadurch werden wieder durch übrig gebliebene Kanten Objekte erkannt, obwohl sich hier im Originalbild keine befinden.

Vordergrundbilder als Check für die Existenz von Objekten

Als dritte Überlegung ist das Verfahren wie folgt verändert worden: Das Outputbild der Kamera wurde einerseits dem Objekterkenner übergeben und andererseits der Hintergrundsubtraktion. Die Objekte des Objekterkenners werden dann überprüft, indem jede erkannte Box kontrolliert wird. Die Idee ist, dass sich bewegende Objekte im Vordergrundbild enthalten sind und so Hintergrundobjekte herausgefiltert werden.
Hat der Objekterkenner nun zum Beispiel eine Ampel als Mensch erkannt, ist die Ampel, welche Hintergrund ist, nicht im Vordergrund zu sehen. Nun wird die Box auf das Vordergrundbild projiziert. Weiter werden nun alle weißen Pixel gezählt, welche sich innerhalb

der Box befinden. Ist nun der Anteil an weißen Pixel im Verhältnis zu allen Pixel größer als ein Threshold, ist das Objekt dort vorhanden, anderenfalls ist das Objekt falsch erkannt und wird nicht beachtet (siehe Codeausschnitt 4).

Codeausschnitt 4: *checkBox Methode um eine Bounding Box zu überprüfen*

```
def checkBox(bbox, backgroundSubtractedImage, thresh):
    image = cv2.imread(backgroundSubtractedImage)
    x1 = round(bbox[1] * 1920)
    x2 = round(bbox[3] * 1920)
    y1 = round(bbox[0] * 1080)
    y2 = round(bbox[2] * 1080)
    imCrop = image[int(y1) : int(y2), int(x1) : int(x2)]
    n_white_Pixels = np.sum(imCrop == 255) / 3
    n_allPixels = imCrop.shape[0] * imCrop.shape[1]

    if((n_white_Pixels / n_allPixels) >= thresh):
        return True;

    return False
```

Dies stellt sich als der beste Ansatz heraus, da der Objekterkenner auf Originalbildern arbeiten kann. Das Modell ist auf genau solche Alltagsbilder trainiert. In diesem Ansatz dienen die Vordergrundbilder der Überprüfung von Objekten.

4.2 Probleme bei der Hintergrundsubtraktion

Der Nachteil bei einer Hintergrundsubtraktion ist der Verlust von Informationen. Mögliche Fehlerquellen werden zwar eingeschränkt, jedoch nimmt man auch Objekte von Interesse heraus. Dies geschieht zum Beispiel beim Stehenbleiben eines Objektes. Durch langes Aufhalten an einer Position wird dieses Objekt nach und nach zum Hintergrund und wird somit auch nicht mehr erkannt. Ein Beispiel zeigt Abbildung 22, bei der der Radfahrer bereits nicht mehr allzu gut zu erkennen ist. Ein Vergleich dazu ist Abbildung 21. Hier ist der Radfahrer 300 Bilder (6s bei einer Kamera mit 50 Hz) vorher noch deutlich besser erkennbar.

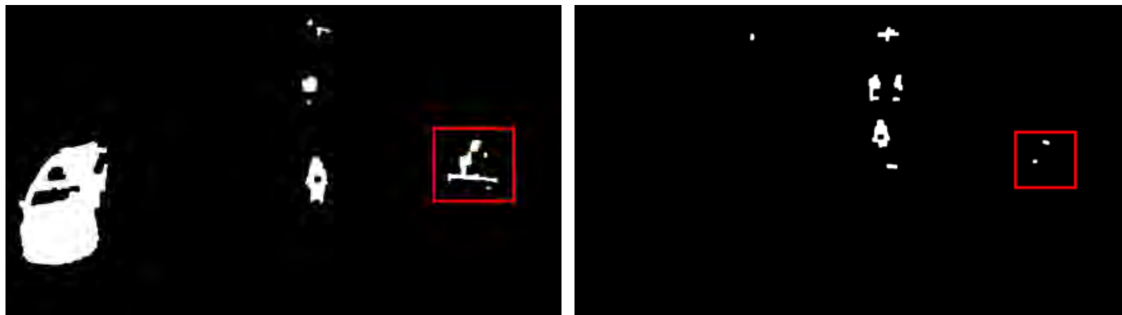

Abb. 21: *Vordergrundbild mit sich bewegendem Fahrradfahrer*

Abb. 22: *Verschwinden des Fahrradfahrer nach Stehenbleiben*

Weiter stellt die Wahl der Parameter für die Objekterkennung eine Herausforderung dar. Dies ist vor allem wichtig, um langes Vorhandensein von Artefakten zu verhindern. Dabei

ist die Wahl zweier Parameter entscheidend, die sich wiederum auch auf die Performance des Algorithmus auswirken: Zum einen, welches Bild an einem Block für den Appendschritt hinzugefügt wird, zum anderen, wie groß ein solcher Block sein muss, bevor dieser Schritt ausgeführt wird. Je höher die Framerate der Bilder ist und somit auch je kleiner die Unterschiede in den nachfolgenden Bildern sind, desto mehr kann der Abstand der Bilder für einen Block erhöht werden. Je größer ein Block ist, desto länger bleiben Artefakte im Vordergrund, da so mehr Zeit vergeht, bis ein Update ausgeführt wird.
Ein wichtiger Schritt ist auch die Initialisierung. Befinden sich in diesen Bildern viele stehende Objekte, werden diese zum Hintergrund und somit Artefakte, welche vorerst so im Hintergrund bestehen bleiben. Am besten geeignet ist eine Bildfolge, in der sich Objekte hauptsächlich bewegen.

5 Vergleich

Im folgenden Kapitel werden alle Verfahren verglichen und evaluiert. Dabei wird auf die Unterschiede der Verfahren eingegangen und die Evaluierung aufgeschlüsselt. Im Zuge dessen werden die benutzte Metrik sowie die Zusammensetzung der Messungen erläutert. Am Ende wird auf die Performance bei Objekterkennung mit Hintergrundsubtraktion im Vergleich zur reinen Objekterkennung eingegangen.

5.1 Unterschiede der Vorgehensweisen

Der wesentliche Unterschied der Vorgehen mit Hintergrundsubtraktion ist, welche Art von Bildern der Objekterkenner als Eingabe erhält. Bei reiner Objekterkennung kann der Objekterkenner auf Bildern arbeiten, die er gewohnt ist, also Originalbilder. Er kennt die Objekte mit ihren Kanten und kann so relativ genau arbeiten. Dieser Vorteil wird nun genutzt und die Hintergrundsubtraktion als Nachverarbeitungsschritt herangezogen. Da bei reinen Vordergrundbildern die Kanten verschwinden und es so für den Objekterkenner nicht mehr ausreichend zu erkennen ist, um welches Objekt es sich handelt, ist Ansatz 3, also die Vordergrundbilder zur Überprüfung der Objekte zu verwenden, die beste Lösung.

5.2 Evaluierung der Vorgehensweisen

Nachfolgend wird das Labeling erläutert und wozu es notwendig ist. Außerdem werden die Metriken genauer erklärt. Abschließend werden die Ergebnisse vorgestellt und aufgeschlüsselt, sowie die Messung der Performance dargestellt.

5.2.1 Labeling

Für die Auswertung ist ein Ground Truth Datensatz notwendig. Dieser Datensatz enthält Angaben zu allen gelabelten Objekten, also allen Objekten, die erkannt werden sollen. Diese setzen sich aus der Information zur Klasse des Objektes sowie der Position, welche durch die Koordinaten einer Box angegeben ist, zusammen. Ein Beispielobjekt im Coco Json Format, welches zur Evaluierung genutzt wird, ist im Codeausschnitt 5 zu sehen. Die Box ist mittels der links oberen x- und y-Koordinate sowie Breite und Höhe angegeben.

Codeausschnitt 5: *Gelabeltes Objekt im Coco Json Format*

```
1  {
2          "category_id": 2,
3      "id": 1,
4      "image_id": 0,
5      "iscrowd": 0,
6      "area": 38804.0,
7      "bbox": [296.0, 384.0, 218.0, 178.0]
8  }
```

Diese annotierten Objekte bilden die Grundlage der Evaluierung. Außerdem sind solche Datensätze ebenfalls notwendig, damit Netze trainiert werden können. Diese enthalten Informationen, was vom Modell erkannt werden soll und wie die Objekte klassifiziert werden sollen. Um einen solchen Ground Truth Datensatz zu erzeugen, wurde das schon erwähnte Computer Vision Annotation Tool (CVAT) verwendet.
CVAT wurde von Intel als Open Source Software entwickelt zu dem Zweck, Benutzern ein praktisches intuitiv bedienbares Programm zum Annotieren von Bildern und Videos anzubieten. Ein Vorteil des Tools ist seine leichte Installation in einer Docker Umgebung, wodurch das Tool mittels eines Chrome Browsers im lokalen Netz nutzbar ist. Dies ist wiederum der Nachteil des Tools, da es aktuell nur mit dem Chrome Browser bedient werden kann. Außerdem können öffentliche Tasks erstellt werden, wodurch eine leichte Zusammenarbeit innerhalb eines Teams möglich ist. Eine sehr große Hilfe für das Annotieren von großen Datensätzen ist die automatische Annotierfunktion, bei der das Annotieren von einzelnen Schlüsselframes (zum Beispiel jedes zehnte Bild) ausreicht und die Bilder dazwischen interpoliert werden. Das funktioniert fast immer sehr präzise. Nur für spezielle Situationen, wie zum Beispiel dem Verschwinden oder Verdecken von Objekten, muss eventuell per Hand nochmal nachgebessert werden. Das Tool ist wie folgt aufgebaut (siehe Abbildung 23).

Abb. 23: *Computer Vision Annotation Tool*

Im Mittelpunkt befindet sich das aktuelle Bild, welches annotiert wird. Rechts sind alle gelabelten Objekte mit Klassifizierung angegeben. Darunter ist ein Control Panel platziert mit der Möglichkeit, weitere Objekte hinzuzufügen und die Interpolation zu nutzen.

Dies bietet sich vor allem für Datensätze an, bei welchen die nachfolgenden Bilder sehr kleinschrittig fortlaufend sind. [13] Informationen zur Installation und Nutzung sind dem CVAT Github Repository zu entnehmen [5].

5.2.2 Metriken

Um die Qualität des Objekterkenners zu beurteilen, ist eine Auswertung der erkannten Objekte notwendig. Dabei werden Detektionen in verschiedene Klassen aufgeteilt. True positive (TP) sind alle Objekte, welche richtig erkannt wurden. Dies ist der Fall, wenn die „Intersection over Union“ der Boxen, oder auch der Jaccard-Koeffizient genannt, mindestens genauso groß wie ein bestimmter Prozentsatz (meist 50 %) ist. Die Zusammensetzung und Berechnung ist in Abbildung 24 und 25 dargestellt.

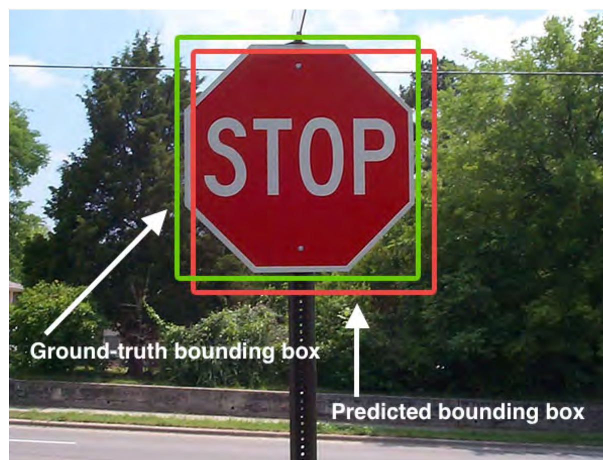

Abb. 24: *Erkannte Box und gelabelte Box*

$$\text{IoU} = \frac{\text{Area of Overlap}}{\text{Area of Union}}$$

Abb. 25: *Berechnung der Intersection over Union*

[5] https://github.com/opencv/cvat

False negative (FN) sind alle Objekte, welche nicht erkannt, aber gelabelt wurden, also Objekte, die eigentlich erkannt hätten werden sollen. Objekte, die erkannt, aber falsch klassifiziert sind oder erkannte Objekte, die sich dort nicht befinden, werden als false positive (FP) bezeichnet. True negative (TN) sind Objekte, welche korrekterweise nicht erkannt wurden. Die vierte Klassifizierung sind true positive (TP) Objekte. Diese sind alle richtig erkannten Objekte. Für eine Kreuzung, bei der nur lebendige Objekte von Relevanz sind, sind zum Beispiel eine erkannte Person als true positive, eine Person, welche als Auto erkannt wurde, als false positive, eine Ampel, die nicht erkannt wurde, als true negative und eine Person, die nicht erkannt wurde, als false negative zu werten. Abbildung 26 zeigt graphisch die Einteilung von erkannten Objekten.
Weiter ist anzumerken, dass sich alle erkannten Objekte aus $FP + TP$ zusammensetzen. Alle relevanten Objekte ergeben sich aus $FN + TP$, was wiederum allen gelabelten Objekten entspricht. Für die Bewertung aller Ansätze wird Recall und Precision herangezogen. Precision wird mittels $\frac{TP}{TP+FP}$ berechnet und stellt die Genauigkeit des Verfahrens dar, also wie viele Objekte richtig erkannt worden sind im Verhältnis zu allen erkannten Objekten. Der Recall, berechnet durch $\frac{TP}{FN+TP}$, gibt an, wie viele Objekte richtig erkannt wurden im Bezug zu allen zu erkennenden Objekten. Diese Metriken wurden ausgewählt, weil dadurch sehr gut die Genauigkeit, aber auch die Anzahl an erkannten Objekten, betrachtet werden kann.

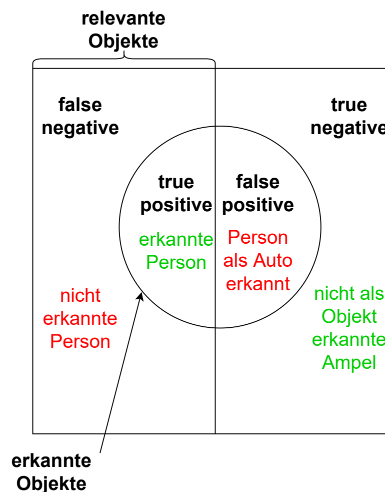

Abb. 26: *Darstellung der Einteilung von Objekten*

5.2.3 Ergebnisse und deren Aufschlüsselung

In der folgenden Auswertung werden vor allem Werte von Recall und Precision sowie die Anzahl der true positive bzw. false negative Objekte aufgeschlüsselt. Es werden vier Szenen betrachtet. Diese sind bezeichnet als *Versuch_110* HK1 bzw. HK2 und *Versuch_108* HK1 bzw. HK2. Die beiden ersten Szenen beinhalten jeweils 1.915 Bilder und die beiden letzteren jeweils 500 Bilder. Es wurde eine reine Objekterkennung auf allen vier Szenen und eine Objekterkennung mit Vordergrundbildern als Überprüfung durchgeführt. Die beiden anderen Ansätze wurden jeweils nur mit der ersten Szene getestet, um zu überprüfen, welche der Ansätze die besten Ergebnisse in Bezug auf Recall und Precision bilden. Die Vordergrundüberprüfung wurde mit einem $IoU = 0.5$ durchgeführt. Beim Versuch_110 HK1 gibt es insgesamt 39.392 Objekte zu erkennen. Mit reiner Objekterkennung ergibt sich eine Anzahl von 21.000 Objekten, welche korrekt (TP) und 10.000 welche falsch (FP) erkannt wurden. Damit liegt die Precision bei 67 %. Die FP-Fälle sind vor allem Klassenfehler und Doppelerkennungen, welche in der Evaluierung als FP gewertet werden. Außerdem werden auch teilweise Objekte dort erkannt, wo sich keine befinden.
Abbildung 27 zeigt zwei Boxen, welche Personen identifiziert haben. Diese sind jedoch viel zu groß und auch eine Doppelerkennung. Wie man in Abbildung 28 sehen kann, sind diese Boxen nach Überprüfung des Vordergrunds herausgefiltert.

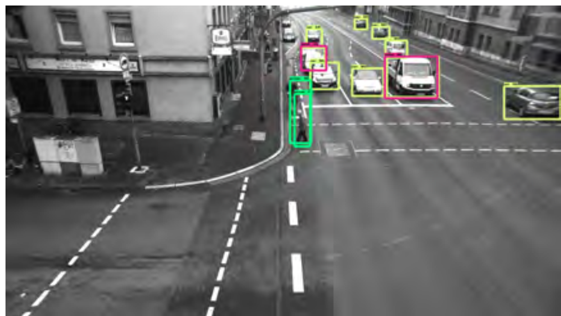

Abb. 27: *Zu groß erkannte Person*

Abb. 28: *Herausfiltern der zu groß erkannten Person*

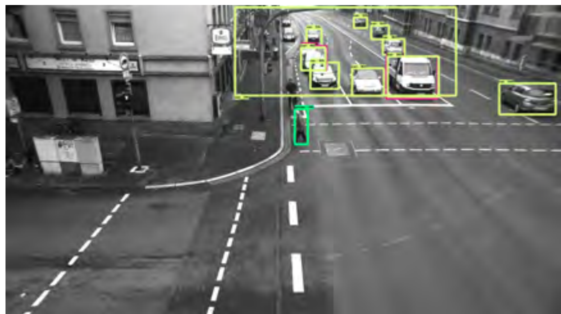

Abb. 29: *Falsch erkanntes Auto*

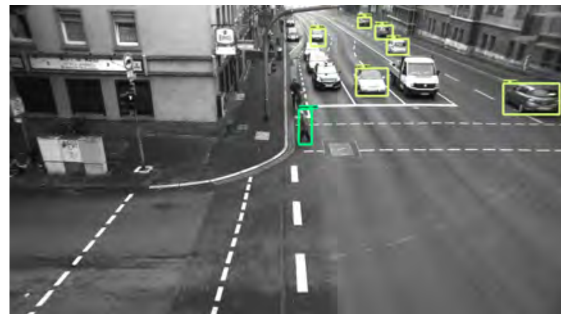

Abb. 30: *Herausfiltern des falsch erkannten Autos*

Ähnliches zeigt Abbildung 29, denn hier ist im oberen linken Eck ein Auto erkannt worden, das durch den Vordergrundcheck gelöscht worden ist (Abbildung 30).
Der Recall ist mit 53 % relativ gut, somit werden rund die Hälfte der zu erkennenden Objekte auch korrekt erkannt. Mit Hintergrundsubtraktion lässt sich die Precision auf 93 % steigern. Jedoch hat man nur noch einen Recall von 11 %. Hier ist anzumerken, dass parkende Autos natürlich nie erkannt werden. Diese machen im gelabelten Datensatz insgesamt 26.000 Objekte aus. Rechnet man diese heraus, ergibt sich für diesen Versuch ein Recall von 32 %. Die restlichen verlorenen Objekte lassen sich mit sehr langsamen Fußgängern bzw. Objekten, welche beispielsweise an einer Ampel stehen bleiben, erklären. Ebenfalls muss hinzugefügt werden, dass die Parameterwahl für die Mindestanzahl an Einsen, die in einer Box vorhanden sein müssen, um als bewegendes Objekt erkannt zu werden, entscheidend ist und mit 50 % vermutlich noch nicht optimal gewählt ist.
Wie in Abbildung 31 zu erkennen ist, steigt die Precision vor allem bei den Autos stark an, bei Rädern hingegen sinkt sie. Dies ist mit einer Falsch-Erkennung zu erklären, bei der der Objekterkenner eine Person als Fahrrad identifiziert. Da sich diese Person bewegt, wird dies nicht mit dem Vordergrundcheck herausgefiltert. Der Radfahrer, der an der Kreuzung lange stehen bleibt, wird bei reiner Objekterkennung erkannt, jedoch durch die Überprüfung mit dem Vordergrund gelöscht. Dies zeigt auch der Recall beim Rad im Vergleich zum Recall beim Fahrrad ohne Überprüfung. Der schwache Recallwert bei den Personen ist ebenfalls mit sehr langsamen und stehenbleibenden Objekten zu begründen.

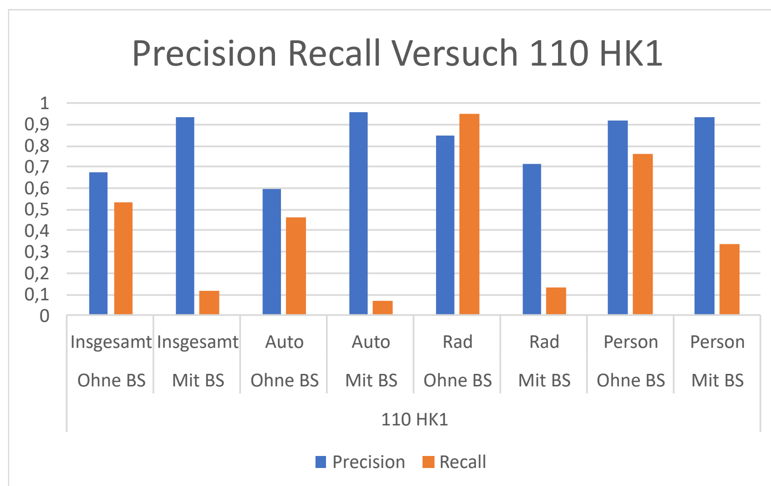

Abb. 31: *Precision Recall Versuch 110 HK1 nach Objektklasse*

Betrachtet man HK2 (siehe Abbildung 32), sieht man insgesamt eine ähnliche Precision. In der Summe ist außerdem wieder ein Verlust des Recalls zu erkennen. Das ist wiederum mit dem Stehenbleiben von Objekten zu erklären, da dieser Versuch einen anderen Kamerawinkel verwendet, bei der die andere Seite der Kreuzung gefilmt wird. Dieser Teil der Kreuzung hat dabei eine rote Ampel, wodurch viele Erkennungen durch den Vordergrundcheck herausfallen. Das Rad, das sich in dieser Szene bei der Kreuzung befindet, wird dabei nicht wie in HK1 von der Seite gefilmt, sondern von der Frontansicht. Da-

bei hat der Objekterkenner, wie man sieht, auch ohne Vordergrundüberprüfung bereits Schwierigkeiten, was der niedrige Recall bei Rad ohne Hintergrundsubtraktion zeigt. Insgesamt arbeiten aber beide Methoden bei diesem Versuch sehr genau mit einem Wert von ca. 90 % Precision.

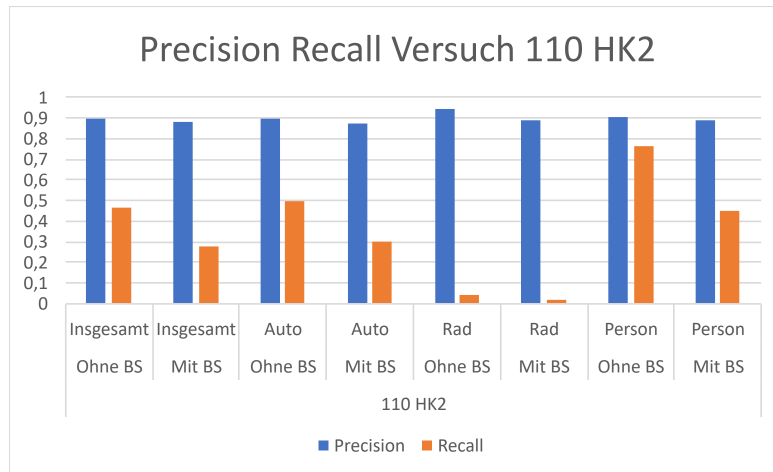

Abb. 32: *Precision Recall Versuch 110 HK2 nach Objektklasse*

Betrachtet man Versuch 108, ist wieder eine deutliche Steigerung der Precision zu sehen (vgl. Abbildung 33), denn sie verbessert sich von 64 % auf 82 %. Außerdem ist der Verlust von Recall hier ebenfalls zu sehen. Rechnet man hier wiederum die parkenden Autos weg, erreicht die Methode mit Hintergrundcheck statt 11 % immerhin 31 %. Die Precision ist auf viele Klassenfehler in diesem Versuch zurückzuführen, vor allem bei Autos.

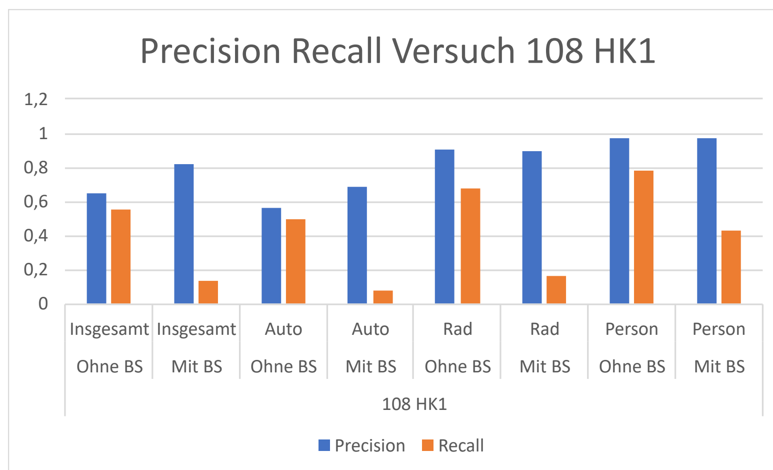

Abb. 33: *Precision Recall Versuch 108 HK1 nach Objektklasse*

Beim anderen Kamerawinkel HK2 sieht man sehr genau den Trade off bei Hintergrundsubtraktion (vgl. Abbildung 34). Es zeigt sich eine Steigerung der Precision, jedoch gleichzeitig ein Verlust beim Recall durch an der roten Ampel stehende Autos, beziehungsweise Radfahrer. Zusammenfassend stellt sich hier sogar eine Steigerung auf 92 % Precision heraus.

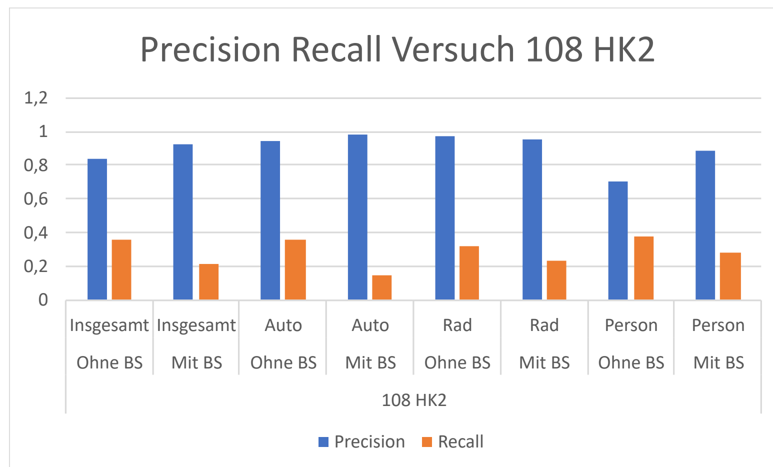

Abb. 34: *Precision Recall Versuch 108 HK2 nach Objektklasse*

Betrachtet man die reine Anzahl an Objekten, erkennt man, dass viele Autos unbewegt stehen (in Abbildung 35 der graue Balken im Gegensatz zu den anderen beiden). Dies fällt auch an der TP-Erkennung im Vergleich zur Auswertung mit Vordergrundcheck auf. Bei den stehenden Autos hat es der Objekterkenner leicht, relativ viele TP zu generieren. Die Anzahl an FP-Erkennungen sinkt vergleichsweise jedoch auch deutlich. Dies zeigt auch Recall und Precision, welche in Abbildung 36 zusammengefasst sind. Hier ist eine Steigerung der Precision von 14 % auf 88 % zu verzeichnen. Der Recall ohne parkende Autos fällt dagegen von 50 % auf 30 %.

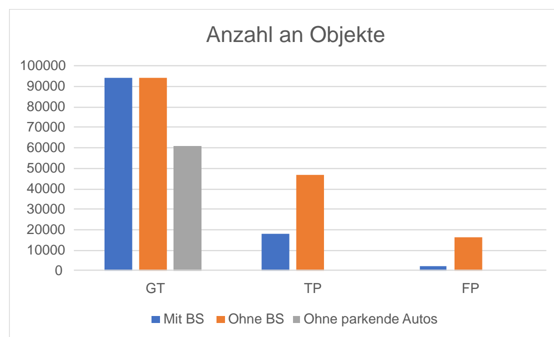

Abb. 35: *Anzahl an TP, GT und FP Erkennung in allen Versuchen*

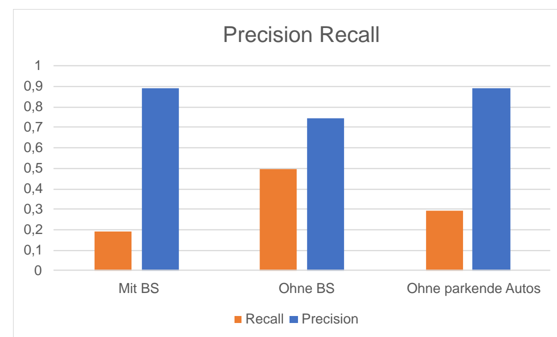

Abb. 36: *Precision Recall zusammengefasst*

Abbildung 37 zeigt nochmal eine Aufteilung der Anzahl in die verschiedenen Objektklassen, Abbildung 38 einen Vergleich der Precision- und Recall-Werte ebenfalls aufgeteilt nach Objektklassen in allen Versuchen. Hier zeigt sich, dass der Objekterkenner am besten mit Personen zurecht kommt und nur bedingt gut mit Autos, was jedoch auf viele Doppelerkennungen zurückzuführen ist, da die FP-Rate relativ hoch ist. Fahrräder stellen für den Objekterkenner noch die größte Herausforderung dar, wie Precision und Recall deutlich zeigen.

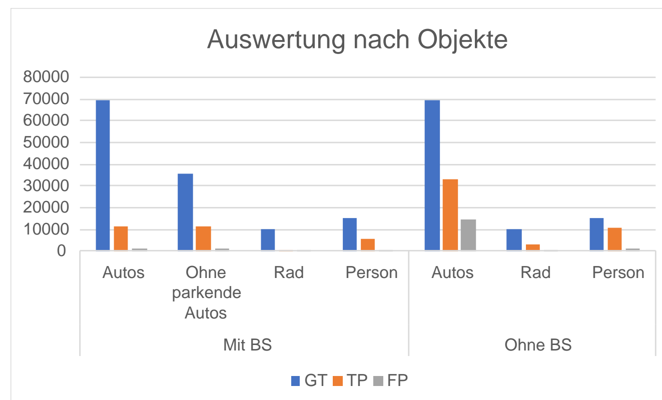

Abb. 37: *Anzahl Objekte nach Klassen in allen Versuchen*

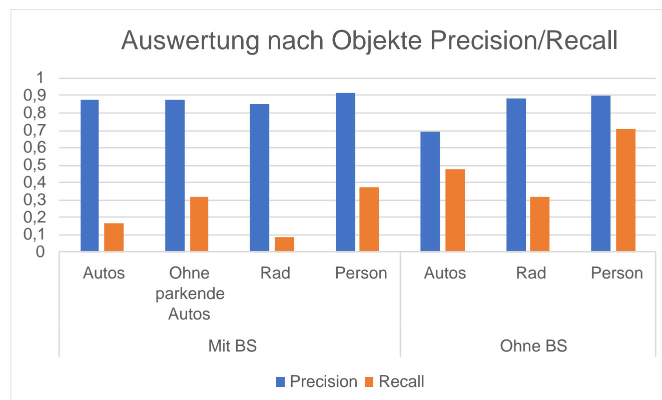

Abb. 38: *Precision Recall nach Klassen*

Allgemein zeigt das Gesamtbild jedoch den Trend, den alle Versuche haben. Eine Gesamtauswertung der Zahlen ist in Tabelle 4 aufgeführt. Im Vergleich dazu sind bei den anderen Ansätzen, bei welchen nur der erste Versuch ausgewertet wurde, schlechtere Ergebnisse erzielt worden. Bei den Graustufenbildern ergab sich als Precision 67 %, was die Precision der reinen Objekterkennung widerspiegelt. Jedoch ergab sich ein Recall von nur 6 %. Bei den Maskenbildern ergab sich hingegen eine Precision von 82 % gegenüber des Recalls mit 10 %, was immer noch schlechter ist, als die Überprüfung mit den Vordergrundbildern.

Versuch	Hintergrund-subtraktion	Objekte	Precision	Recall	Ground Truth (GT)	TP	FP
110 HK1	ohne	Insgesamt	0,67228931	0,53422015	39392	21044	10258
		Auto	0,59961094	0,45958123	30852	14179	9468
		Rad	0,85014138	0,94698163	1905	1804	318
		Person	0,91469366	0,76277317	6635	5061	472
	mit	Insgesamt	0,93125	0,11725731	39392	4619	341
		Auto	0,95965935	0,06939583	30852	2141	90
		Rad	0,71629213	0,13385827	1905	255	101
		Person	0,93678887	0,33504145	6635	2223	150
110 HK2	ohne	Insgesamt	0,89805299	0,46386747	37487	17389	1974
		Auto	0,89641093	0,50162146	27136	13612	1573
		Rad	0,94208494	0,04270214	5714	244	15
		Person	0,90150549	0,76191503	4637	3533	386
	mit	Insgesamt	0,87872376	0,27697602	37487	10383	1433
		Auto	0,87487979	0,30173939	27136	8188	1171
		Rad	0,88888889	0,01820091	5714	104	13
		Person	0,89358974	0,45093811	4637	2091	249
108 HK1	ohne	Insgesamt	0,64340682	0,55449005	11158	6187	3429
		Auto	0,55941645	0,49658583	8494	4218	3322
		Rad	0,90431267	0,67234469	998	671	71
		Person	0,97301349	0,77911164	1666	1298	36
	mit	Insgesamt	0,82059621	0,1356874	11158	1514	331
		Auto	0,68506494	0,07452319	8494	633	291
		Rad	0,89444444	0,16132265	998	161	19
		Person	0,97165992	0,43217287	1666	720	21
108 HK2	ohne	Insgesamt	0,84086664	0,35471163	6346	2251	426
		Auto	0,94046418	0,35464231	2628	932	59
		Rad	0,96780684	0,32002661	1503	481	16
		Person	0,70479394	0,37832957	2215	838	351
	mit	Insgesamt	0,92803802	0,21541128	6346	1367	106
		Auto	0,98496241	0,14954338	2628	393	6
		Rad	0,95543175	0,22821025	1503	343	16
		Person	0,88251748	0,28487585	2215	631	84

Tab. 4: *Ergebnisse der Objekterkennung*
(Die farbig hinterlegten Felder beziehen sich auf Nennungen im Text)

5.3 Laufzeitverhalten

Ein wichtiger Bestandteil der Auswertung ist auch die Bewertung der zeitlichen Performance. Da diese Objekterkennung vor allem in Live Systemen in Fahrzeugen vorgesehen ist, ist es entscheidend, wie stark eine Hintergrundsubtraktion die Laufzeit pro Bild verlängert. Dabei wurde im Folgenden jeweils die Objekterkennung sowie die Überprüfung des Vordergrundes betrachtet und die Laufzeit gemessen. Ebenfalls wurde dies bei der Hintergrundsubtraktion durchgeführt. Die Objekterkennung wurde auf einem Testrechner, der mit einer der besten GPUs zum jetzigen Zeitpunkt (Nvidia Titan V) ausgestattet war, ausgeführt. Im Vergleich dazu lief die Hintergrundsubtraktion auf einer Intel-CPU (i7-3820), die zum heutigen Zeitpunkt nicht die bestmögliche Performance liefert. Deshalb ist anzunehmen, dass die Hintergrundsubtraktion noch schneller ausgeführt werden kann als sich in diesem Laufzeittest gezeigt hat.
Ziel ist es, während der Objekterkennung bereits eine Hintergrundsubtraktion einzuleiten, um nach der Erkennung sofort mit dem Check fortzusetzen. Im Optimalfall geschieht dies, ohne dass auf die Hintergrundsubtraktion gewartet werden muss. Betrachtet man die Ergebnisse der Hintergrundsubtraktion des C++ Programms, welche in Tabelle 5 dargestellt sind, erkennt man, dass bei der parallelen Version ohne das Herausschreiben der Bilder ein Durchschnitt von 132 s erzielt wird. Man kann annehmen, dass das Herausschreiben der Bilder umgangen werden kann, indem man einen Wrapper entwirft, der das C++ Programm ausführt und die Ausgabe, die das Vordergrundbild sein würde, entgegennimmt. Da das Programm mit 1915 Bildern ausgeführt wurde, ergibt sich pro Bild ein Schnitt von ca. 0,069 s. Bei der Objekterkennung mit Vordergrundcheck beansprucht die Erkennung 160 s (vgl. Tabelle 6), was eine Zeit von 0,084 s pro Bild bedeutet. Deswegen ist zu erwarten, dass die Hintergrundsubtraktion das Bild vor Beendigung der Objekterkennung bereitstellen kann, wodurch kein Performance-Overhead durch die Hintergrundsubtraktion entstehen sollte.

	mit Herausschreiben der Bilder	ohne Herausschreiben der Bilder
parallel	133 s	132 s
nicht parallel	144 s	141 s

Tab. 5: *Performance Hintergrundsubtraktion als Durchschnitt von 10 Durchläufen*

	Erkennung insgesamt	Überprüfen der Objekte	Objekt Erkennung
mit Hintergrund-subtraktion	200 s	29 s	161 s
ohne Hintergrund-subtraktion	171 s	0 s	160 s

Tab. 6: *Performance Objekterkennung als Durchschnitt von 10 Durchläufen*

Würde man für die Hintergrundsubtraktion eine noch bessere CPU verwenden, ist nochmal mit einer Steigerung der Performance im Bezug auf die Hintergrundsubtraktion zu rechnen. Insgesamt ergibt sich bei Objekterkennung mit Vordergrundcheck eine Zeit von ca. 0,1 s pro Bild, also im Vergleich zu 0,085 s ein leichter Performanceverlust von 0,015 s pro Bild. Derzeit werden üblicherweise Kameras mit 33 Hz genutzt, die also 30 Bilder pro Sekunde liefern. Daraus kann man folgern, dass zu einer Liveanwendung noch der Faktor drei fehlt, was jedoch schon ein gutes Fundament bildet.

6 Tracking

Um das Problem des Verlustes eines Objektes durch Stehenbleiben zu verhindern, wird versucht, das Objekt mittels Tracking weiterzuverfolgen. Als Verfahren wird das Kalman-Filter verwendet. Das Kalman-Filter, entwickelt von R. E. Kalman, ist eine Standard Methode des Trackings. Diese fand einen ersten populären Einsatz in den 1960er Jahren in der Navigation des Apollo-Projektes, welche die Steuerung zum Mond unterstützen sollte. [6] Im Folgenden wird kurz auf die Implementierung des Kalman-Filters eingegangen und anschließend evaluiert, inwiefern sich die Anwendung auf die Objekterkennung im Vergleich zur Anwendung ohne Tracking auswirkt.

6.1 Modellierung und Implementierung

Betrachten wir zunächst die Modellierung und Implementierung des Kalman-Filters und des Trackings.

Modellierung

Der Kalman-Filter basiert auf eine Zustandsmodellierung, bei der ein dynamisches Übertragungssystem beschrieben wird. Im Wesentlichen besteht die Abfolge des Programms aus einer Vorhersage und einem Updateschritt, welcher sich durchgehend wiederholt. Um den Kalman-Filter zu implementieren sind einige Vektoren und Matrizen notwendig. Der Zustandsvektor $x = \begin{pmatrix} x & y & w & h & vx & vy & vw & vh \end{pmatrix}$ beschreibt die Box eines erkannten Objektes und die Geschwindigkeiten, welche die Änderung jedes Parameters darstellen. Weiter stellt die Übergangsmatrix A (8x8-Matrix)

$$A = \begin{pmatrix} 1 & 0 & 0 & 0 & \Delta t & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & \Delta t & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & \Delta t & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & \Delta t \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

den Unterschied im Zeitstempel des aktuellen und vorangegangenen Bilds dar. Die Größe Δt stellt den Zeitunterschied, meistens in Sekunden, dar. Bei einer 50 Hz Kamera entspricht also $\Delta t = 0.02$.
Weiter gibt die Rauschmatrix Q für den Prozess das Vertrauen zum System an. Es wird angenommen, dass das Rauschen in x, y und Höhe und Weite unabhängig voneinander ist. Die Matrix stellt die Kovarianz des Prozessrauschens dar. Es existieren unterschiedliche Ansätze zur Wahl dieser Matrix. Die Messmatrix

$$H = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

stellt die Messwerte dar.
Wie auch eine Rauschmatrix für den Prozess existiert, gibt es genauso eine Rauschmatrix R für die Messung. Hier wird wieder angenommen, dass das Rauschen in jede Richtung unabhängig voneinander ist. Die Rauschmatrix

$$R = \begin{pmatrix} \sigma_x^2 & 0 & 0 & 0 \\ 0 & \sigma_y^2 & 0 & 0 \\ 0 & 0 & \sigma_h^2 & 0 \\ 0 & 0 & 0 & \sigma_w^2 \end{pmatrix}$$

stellt das Vertrauen in die Messung beziehungsweise in dieser Arbeit das Vertrauen in die erkannte Box des Objekterkenners dar. Dabei wird als Varianz $\sigma^2 = 5$ verwendet. Der Messfehler ist also für jede Größe $5m^2$. Da, wie schon erwähnt, die Messungen unabhängig sind, sind alle anderen Werte außer der Diagonale 0.
Die Kovarianzmatrix P gibt das Vertrauen bei der Initialisierung zu den Koordinaten, also der Position an. In dieser Modellierung wird bei der Initialisierung

$$x = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

gesetzt. Da dies nur eine Zufallsinitialisierung ist, enthält P am Anfang sehr große Werte. Dies hat zur Folge, dass, sobald eine Messung gemacht wird, der aktuelle Zustand eher in Richtung der Messung als in die des aktuellen Zustandes geht. Die Matrix P ist in der folgenden Implementierung angegeben als:

$$P = \begin{pmatrix} 1000 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1000 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1000 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1000 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1000 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1000 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1000 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1000 \end{pmatrix}$$

Wäre zum Beispiel bei der Initialisierung die korrekte Position eines Objektes bekannt, so wäre P die Nullmatrix $P = 0_{8,8}$. [7] [3]

Implementierung des Kalman-Filters und Tracking

Beim Kalman-Filter wird die Vorhersage abwechselnd mit einer Korrektur angewandt. Also wird zuerst zu einem Zustand eine Vorhersage bezüglich des nächsten Zustandes getätigt. Anschließend wird der nächste Zustand mit einer Messung des neuen Zustands korrigiert. Codeausschnitt 6 zeigt den Prediction Schritt, Codeausschnitt 7 den Korrekturschritt.

Codeausschnitt 6: *Prediciton des Kalman-Filters*

```
1 def kalman_predict(self):
2         self.x_ = self.A @ self.x
3         self.P_ = self.A @ self.P @ self.A.transpose() + self.Q
```

Codeausschnitt 7: *Updateschritt des Kalman-Filters*

```
1  def kalman_update(self, y):
2          v = y - self.H @ self.x_
3          S = self.H @ self.P_ @ np.transpose(self.H) + self.R
4          k = self.P_ @ np.transpose(self.H) @ np.linalg.inv(S)
5
6                  self.x = self.x_ + k @ v
7          self.P = self.P_ - k @ S @ np.transpose(k)
```

Bei der Prediction wird die aktuelle Position, addiert mit der Geschwindigkeit und multipliziert mit dem Zeitunterschied, bestimmt. Außerdem wird durch $P_$ die neue Kovarianzmatrix berechnet, indem zusätzlich die Messunsicherheit hinzugerechnet wird.
Der Updateschritt ist der aufwändigere Teil des Kalman-Filters. Zuerst wird der Unterschied zwischen Prediction und Messung y berechnet. Weiter wird dann in den folgenden beiden Schritten bestimmt, ob den Messwerten oder der Vorhersage mehr vertraut werden soll. Anschließend wird der aktuelle Zustand neu berechnet. Zum Schluss wird noch eine aktuelle Kovarianzmatrix bestimmt. [7] [3]
Die Idee des Trackings besteht darin, die Initialisierung nur mit gesehenen Objekten zuzulassen. Also konkret, wenn ein Objekt mit dem Vordergrund überprüft wurde, ist es sehr sicher, dass sich das Objekt dort auch wirklich befindet und wird deshalb dem Tracker hinzugefügt. Für den Updateschritt sollen dann alle Messungen zur Verfügung stehen, auch die, die die Hintergrundsubtraktion herausfiltert. Das ist notwendig, da die Messungen den Tracks zugeordnet werden und somit nur Messungen, die schon von der Hintergrundsubtraktion überprüft wurden, genutzt werden. Dadurch umgeht man das Problem, dass Erkennungen nach Stehenbleiben verloren gehen. Abbildung 39 zeigt den Ablauf des Codes anschaulich und ist in Codeausschnitt 8 gekürzt dargestellt.

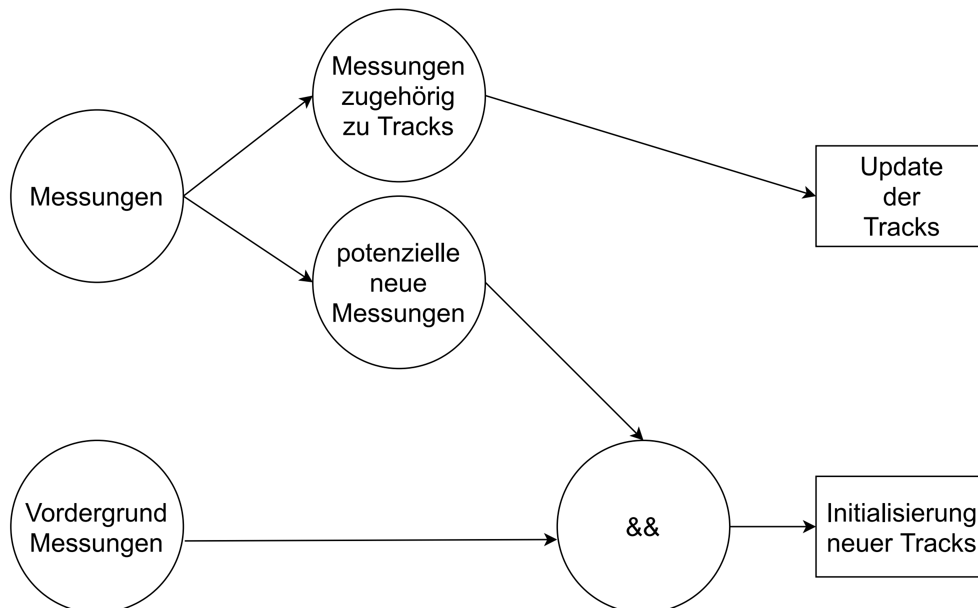

Abb. 39: *Abfolge des Tracking Algorithmus*

Alle Messungen, sowie die Messungen, welche eine Überprüfung bestanden haben, werden dem Tracker übergeben. Dieser führt anschließend eine Prediction durch. Nach dieser Vorhersage werden die neuen Messungen den vorhanden Tracks zugeordnet, um potentielle neue Objekte zu erkennen und den alten die nachfolgenden Messungen zuzuordnen. Dies wird hier mittels einer Kostenmatrix umgesetzt, welche die Intersection over Union für jede Messung bezüglich jeden Tracks speichert. Diese wird anschließend ausgewertet und ordnet die Messungen den Tracks zu. Jede Messung, welche keinen zugehörigen Track besitzt ist, eine potentielle neue Messung. Ein Track und eine Messung gehören zusammen, falls die Intersection over Union größer als ein Treshhold ist ($iOu > thresh$). Daraufhin werden die Vorhersagen mit den zugeordneten Messungen neu berechnet. Weiter werden nun alle potentiell neuen Tracks erzeugt und zwar nur dann, wenn diese auch von der Hintergrundsubtraktion erkannt wurden. Im Zuge dessen werden Tracks, die zu lange nicht mehr eine zugehörige Messung hatten, gelöscht.

Codeausschnitt 8: *Tracking von Boxen*

```
1  def detection_to_track_assignment(self, box_list, class_list, bs_boxes):
2          self._predict_new_locations_of_tracks()
3
4          for i in range(0, n_tracks):
5              for j in range(0, n_detections):
6                  cost_matrix[i][j] = 1-self.iouBBox(self.head_tracks[i].centroid,
7                                                     centroids_list[j])
8
9          track_assignments, det_assignments = self._cost_to_assertion(cost_matrix,
10                 1-self.DET_TO_TRACK_THRES)
11
12         # update new measurement to existing tracks
13     for i in range(0, len(track_assignments)):
14         if track_assignments[i] >= 0:
15             self.head_tracks[i].add_measurement(box_list[track_assignments[i]],
16                                     class_list[track_assignments[i]])
17
18         # add potential new tracks
19     for i in range(0, len(det_assignments)):
20         if det_assignments[i] == -1 and self.is_in_BS(bs_boxes, box_list[i]):
21             unassigned_measurements.append(box_list[i])
22             unassigned_classes.append(class_list[i])
23
24         # update, delete and create tracks
25     self._update()
26     self._delete_lost_tracks()
27     self._create_new_tracks(unassigned_measurements, unassigned_classes)
```

6.2 Evaluierung

Das Ziel des Trackings ist die Erhöhung von Detections, vor allem im Falle des Stehenbleibens eines Objektes, also der Erhöhung des Recalls. Die Ergebnisse des Trackings werden im Folgenden mit den Ergebnissen aus der Objekterkennung mit Überprüfung mittels der Hintergrundsubtraktion sowie reiner Objekterkennung verglichen. Es werden außerdem nur Fahrrad-Objekte betrachtet, da diese in dem verwendeten Versuch genau das Problem aufzeigen. Tabelle 7 stellt die Ergebnisse der Objekterkennung dar. Es zeigt sich, dass sich insgesamt noch mehr TP-Erkennungen ergeben, was mit dem Tracking zu begründen ist. Das hat zur Folge, dass sich hier eine weitere Steigerung bei der Precision ergibt. Es wurden somit 95 % aller zu erkennenden Räder auch wirklich erkannt. Der Rückgang der Precision und der damit verbundenen Erhöhung der FP-Erkennung ist mit einem Klassifizierungsfehler zu begründen. Eine Person wurde als Rad erkannt und somit als Rad weiter verfolgt, obwohl diese nachfolgend nicht mehr als Rad erkannt wurde. Insgesamt zeigt aber der Verbund von Nutzung der Objekte aus der reinen Objekterkennung und Initialisierung der Objekte, erkannt durch die Hintergrundsubtraktion, eine weitere Verbesserung der Objekterkennung.

	reine Objekterkennung	Ohne Tracking	Tracking
TP	1804	485	1817
FP	318	311	538
GT	1905	1905	1905
Recall	94 %	25 %	95 %
Precision	85 %	60 %	77 %

Tab. 7: *Ergebnisse des Trackings*

7 Fazit

Ziel der Arbeit war es, das Problem einer unzureichend präzisen Objekterkennung zu lösen. Dabei wurde versucht, den Vordergrund, der durch eine Hintergrundsubtraktion berechnet wird, zu nutzen. Im Laufe der Arbeit konnte auf die Bestimmung einer Singulärwertzerlegung durch einen iterativen Algorithmus eingegangen werden. Hierfür wurden die zwei nötigen Funktionen kurz erläutert und anschließend der ganze Ablauf dargestellt. Die C++ Version des Programms stellt dabei eine wichtige Rolle dar, weil dadurch eine Performance erreicht werden kann, die kurz davor wäre, in der Praxis einsetzbar zu sein. Weiter folgte die Erklärung der allgemeinen Funktionsweise eines neuronalen Netzes und die Notwendigkeit von mehrschichtigen Netzen für komplexere Aufgaben, veranschaulicht durch das XOR Problem. Für Objekterkennung sind Convolutional Neural Networks die „State of the Art"-Netze, die in der Praxis Anwendung finden.
Anschließend folgte eine Erklärung des Backpropagation-Algorithmus, wodurch, ausgehend vom Fehler am Ausgabeneuron, die Anpassung der Gewichte in jeder Schicht zurückgerechnet werden können. Daraufhin sind die Möglichkeiten, die eine Präzisierung der Objekterkennung durch die Hintergrundsubtraktion ermöglichen, klar aufgezeigt worden. Alle drei Überlegungen wurden beschrieben und mit Beispielbildern sowie Codeausschnitten veranschaulicht. Nach Durchführung einer Evaluation stellte sich die dritte Methode, in der ein Vordergrundcheck verwendet wird, um die Existenz eines Objektes zu zeigen, als beste heraus.
Im Zuge der Auswertung wurden die Ergebnisse bis hin zu den Klassen, welche erkannt werden sollten, aufgeschlüsselt. Dies geschah durch die Erzeugung eines Evaluierungsdatensatzes mittels Labeln von vier Szenen einer Kreuzung mit insgesamt 4830 Bildern. Es folgte die Reduzierung des Problems des Verschwindens von Objekten mithilfe von Tracking. Die Implementierung des Trackings unter Verwendung eines Kalman-Filters lässt eine Weiterverfolgung der Boxen der Objekterkennung zu.
Insgesamt lässt sich zusammenfassen, dass die Hintergrundsubtraktion eine gute Möglichkeit bietet, die Objekterkennung zu präzisieren, da der Vordergrund optimale Informationen über die Existenz von Objekten bietet. Vorausblickend würde sich dieser noch zusätzlich nutzen lassen, um Objekte anzuzeigen, welche noch gar nicht vom Objekterkenner gesehen werden. Diese sind z.B. noch zu klein, werden aber von der Hintergrundsubtraktion durch die Bewegung dieser Objekte bereits ermittelt.
Allgemein bietet der Anwendungsbereich der Objekterkennung noch einige Möglichkeiten, in der sich schon jetzt viele Innovationsmöglichkeiten finden lassen. Vor allem der Schutz von verletzlichen Verkehrsteilnehmern, wie Radfahrern und Fußgängern, ist ein Hauptfeld im Bereich des autonomen Fahrens und stellt die Forschung in Zukunft noch vor große Herausforderungen.
Doch wie sagte bereits Goethe:

„So eine Arbeit wird eigentlich nie fertig,
man muss sie für fertig erklären,
wenn man nach Zeit und Umständen
das Möglichste getan hat." [2]

Literatur

[1] Armadillo c++ library for linear algebra & scientific computing. http://arma.sourceforge.net. Aufgerufen: 31.03.2020.

[2] Zitat zum Thema: Kunst, Künstler. https://www.aphorismen.de/zitat/24370. Aufgerufen: 13.04.2020.

[3] Jeremy Cohen. Computer Vision for tracking. https://towardsdatascience.com/computer-vision-for-tracking-8220759eee85. Aufgerufen: 21.01.2020.

[4] Otto Geißler. So funktioniert Google TensorFlow. https://www.bigdata-insider.de/so-funktioniert-google-tensorflow-a-669777. Aufgerufen: 20.01.2020.

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

[6] Mohinder Grewal and Angus Andrews. Applications of kalman filtering in aerospace 1960 to the present [historical perspectives]. *Control Systems, IEEE*, 30:69 – 78, 07 2010.

[7] Roger R Labbe Jr. *Kalman and Bayesian Filters in Python*. August 29, 2018. https://github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python, Kapitel 8.

[8] Michael Nielsen. Neuronal Networks and Deep Learning. http://neuralnetworksanddeeplearning.com/about.html. Kapitel 1,2 Aufgerufen: 04.11.2019.

[9] Dominic Prinz. Alles oder nichts Gesetz. https://flexikon.doccheck.com/de/Alles-oder-Nichts-Gesetz. Aufgerufen: 05.11.2019.

[10] G. Reitberger, M. Bieshaar, S. Zernetsch, K. Doll, B. Sick, and E. Fuchs. Cooperative tracking of cyclists based on smart devices and infrastructure. In *2018 21st International Conference on Intelligent Transportation Systems (ITSC)*, pages 436–443, 2018.

[11] Günther Reitberger and Tomas Sauer. Background Subtraction using Adaptive Singular Value Decomposition. *CoRR*, abs/1906.12064, 2019.

[12] Joachim Steinwendner Roland Schwaiger. *Neuronale Netze programmieren mit Python*. Rheinwerk Computing, Rheinwerkallee 4, 53227 Bonn, 2019/01.

[13] Boris Sekachev, M. Nikita, and Andrey Z. Computer Vision Annotation Tool: A Universal Approach to Data Annotation. https://software.intel.com/en-us/articles/computer-vision-annotation-tool-a-universal-approach-to-data-annotation. Aufgerufen: 18.02.2020.

[14] Ravi Srisha and Am Khan. Morphological operations for image processing : Understanding and its applications, 12 2013.

Abbildungsverzeichnis

1 Iterativer Algorithmus zur Bestimmung der Singulärwertzerlegung. Entnommen aus [11] . . . 6
2 Nervenzelle. Quelle: https://www.dasgehirn.info/grundlagen/kommunikation-der-zellen/bild-aufbau-eines-neurons, Aufgerufen: 14.11.2019 . . . 7
3 Gelernte Trenngerade beim AND Problem . . . 9
4 Das XOR Problem . . . 9
5 Beide Perceptren für das XOR Problem . . . 10
6 Beide Trenngeraden . . . 10
7 Trenngerade von Perceptron 3 . . . 10
8 Simgoid . . . 11
9 Teilnetz mit Beschriftungen . . . 12
10 Aufbau eines CNN, Quelle: https://jaai.de/convolutional-neural-networks-cnn-aufbau-funktion-und-anwendungsgebiete-1691/, Aufgerufen am 19.03.2020 . . . 15
11 ReLU Funktion . . . 15
12 Max Pooling Beispiel mit 2x2 Filter, Quelle: https://computersciencewiki.org/index.php/Max-pooling_/_Pooling, Aufgerufen am 19.03.2020 . . . 15
13 Padding-Verfahren . . . 16
14 Ausgabebild des Objekterkenners . . . 20
15 Ausgabe der Hintergrundsubtraktion . . . 21
16 Ausgabe Hintergrundsubtraktion mit Graustufenwerten . . . 21
17 Artefakt als Objekt erkannt . . . 22
18 Bild nach Anwendung der Maske . . . 23
19 Anwendung von Dilation, Quelle: https://upload.wikimedia.org/wikipedia/commons/d/d6/MorphologicalDilation.png. Aufgerufen am 02.03.2020 . 24
20 Anwendung von Closing, Quelle: https://upload.wikimedia.org/wikipedia/commons/a/a2/MorphologicalClosing.png. Aufgerufen am 02.03.2020 . . 24
21 Vordergrundbild mit sich bewegendem Fahrradfahrer . . . 25
22 Verschwinden des Fahrradfahrer nach Stehenbleiben . . . 25
23 Computer Vision Annotation Tool . . . 28
24 Erkannte Box und gelabelte Box, Quelle: https://www.pyimagesearch.com/wp-content/uploads/2016/09/iou_stop_sign.jpg, Aufgerufen: 18.02.2020 29
25 Berechnung der Intersection over Union, Quelle: https://www.pyimagesearch.com/wp-content/uploads/2016/09/iou_equation.png, Aufgerufen: 18.02.2020 . . . 29
26 Darstellung der Einteilung von Objekten . . . 30
27 Zu groß erkannte Person . . . 31
28 Herausfiltern der zu groß erkannten Person . . . 31
29 Falsch erkanntes Auto . . . 31
30 Herausfiltern des falsch erkannten Autos . . . 31
31 Precision Recall Versuch 110 HK1 nach Objektklasse . . . 32

32 Precision Recall Versuch 110 HK2 nach Objektklasse 33
33 Precision Recall Versuch 108 HK1 nach Objektklasse 33
34 Precision Recall Versuch 108 HK2 nach Objektklasse 34
35 Anzahl an TP, GT und FP Erkennung in allen Versuchen 34
36 Precision Recall zusammengefasst 34
37 Anzahl Objekte nach Klassen in allen Versuchen 35
38 Precision Recall nach Klassen 35
39 Abfolge des Tracking Algorithmus 41

Eigenständigkeitserklärung

Hiermit bestätige ich, Felix Adler, dass ich die vorliegende Arbeit selbstständig und ohne unzulässige Hilfe verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie die wörtlich und sinngemäß übernommenen Passagen aus anderen Werken kenntlich gemacht habe.

Die Arbeit ist weder von mir noch von einer anderen Person an der Universität Passau oder an einer anderen Hochschule zur Erlangung eines akademischen Grades bereits eingereicht worden.

Neuburg am Inn, den 15. April 2020

____

Felix Adler