

Bachelorarbeit

Adem Dagdeviren

Semantische Bildsegmentierung mit neuronalen Netzen für
einen autonomen Roboter

Adem Dagdeviren

Semantische Bildsegmentierung mit neuronalen Netzen für einen autonomen Roboter

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang *Bachelor of Science Technische Informatik*
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Andreas Meisel
Zweitgutachter: Prof. Dr. Tim Tiedemann

Eingereicht am:

Adem Dagdeviren

Thema der Arbeit

Semantische Bildsegmentierung mit neuronalen Netzen für einen autonomen Roboter

Stichworte

Semantische Segmentierung, Bildverarbeitung, Neuronale Netze, NVIDIA Jetson Nano

Kurzzusammenfassung

In dieser Arbeit wird ein neuronales Netz zur semantischen Segmentierung untersucht. Das neuronale Netz soll auf einem kompakten Einplatinencomputer von NVIDIA in Echtzeit betrieben werden. Das Ziel ist es, eine bereits vorhandene Erkennungsrate durch die Anwendung verschiedener Techniken zu steigern. Dabei hat das Vortrainieren neuronaler Netze den größten Effekt erzielt. Zudem konnte durch das Modifizieren einzelner Bestandteile des Netzes ein geringfügiger Zuwachs der Erkennungsrate erreicht werden.

Adem Dagdeviren

Title of Thesis

Semantic image-segmentation with neural networks for autonomous robots

Keywords

semantic segmentation, image processing, neural networks, NVIDIA Jetson Nano

Abstract

In this thesis, a neural network for semantic segmentation will be investigated. The neural network is to be operated in real-time on a compact single board computer from NVIDIA. The aim is to increase an existing recognition rate by using various techniques. The pre-training of neural networks has achieved the greatest effect. In addition, a slight increase in the recognition rate could be achieved by modifying individual components of the network.

Inhaltsverzeichnis

Abbildungsverzeichnis	vi
Tabellenverzeichnis	viii
Auflistungsverzeichnis	ix
1 Einleitung	1
1.1 Motivation	1
1.2 Problemstellung und Ziele	1
2 Grundlagen und verwendete Techniken	3
2.1 Transfer-Lernen	3
2.1.1 Vortrainierte Netze	3
2.1.2 Feinabstimmung	4
2.2 Das Problem von tiefen neuronalen Netzen	4
2.3 ResNet	5
2.3.1 Architektur des ResNet	7
2.4 Semantische Segmentierung	10
2.5 Metriken	11
2.5.1 Pixelweise Genauigkeit	11
2.5.2 Intersection-Over-Union	13
2.6 ONNX - Open Neural Network Exchange	14
3 Design	15
3.1 Hardware	15
3.1.1 NVIDIA Jetson Nano	15
3.1.2 Webcam	16
3.1.3 Trainingsrechner für das neuronale Netz	16
3.2 Software	17
3.2.1 Trainingsskript	17

3.2.2	Cityscapes-Datensatz	17
4	Implementierung und Auswertung	19
4.1	Einrichten des Nvidia Jetson Nano	19
4.2	Datenvorverarbeitung	20
4.2.1	Herunterladen der Trainingsdatensätze	20
4.2.2	Ordnerstruktur der Trainingsbilder	20
4.2.3	Klassenreduktion	21
4.3	Trainieren der neuronalen Netze	22
4.3.1	ResNet-18 als Ausgangspunkt	22
4.3.1.1	Auswertung	25
4.3.2	ResNet-18: Unterschiedliche Batchsize	25
4.3.2.1	Auswertung	26
4.3.3	ResNet-18: Auswahl des modifizierten fc	27
4.3.3.1	Auswertung	29
4.3.4	ResNet-18: Modifiziertes fc + Feinabstimmung	29
4.3.4.1	Auswertung	29
4.3.5	ResNet-18: Vortrainiert + Modifiziertes fc + Feinabstimmung	30
4.3.5.1	Auswertung	31
4.4	Neuronales Netz sichern und übertragen	32
4.5	Das neuronale Netz in Betrieb	32
4.5.1	Semantische Segmentierung einiger Bilder	33
4.5.2	FPS Aufzeichnung	35
5	Fazit	37
	Literaturverzeichnis	39
	Veröffentlichungen	39
	Bücher	39
	Online	39
	Selbstständigkeitserklärung	43

Abbildungsverzeichnis

2.1	Trainings- und Testfehler bei 20- und 56-Layer [12]	5
2.2	Eine „Skip-connection“ aus dem ResNet [12]	6
2.3	Training mit ImageNet-Datensatz. Vergleich zwischen ResNet-18 und -34 sowie die herkömmliche Variante(plain-18 und plain-34). Dünne Linien: Trainingsfehler. Dicke Linien: Validierungsfehler [12]	7
2.4	Beispiel Netzwerk Architekturen. Links: VGG-19-Modell [29] (19,6 Mrd. FLOPs) als Referenz. Mitte: flaches Netzwerk mit 34 Layern (3,6 Mrd. FLOPs). Rechts: Residual-Netzwerk mit 34 Layern (3,6 Mrd. FLOPs). Die gepunkteten Linien erhöhen die Dimension. [12]	9
2.5	Kreuzung mit Passanten [3]	10
2.6	Semantische- (links) und Instanzsegmentierung (rechts) [27]	11
2.7	Satellitenaufnahme (links) und die dazugehörige Segmentierungsmaske (rechts) [28]	12
2.8	Vorhersage für die Segmentierung der Schiffe im Hafen (komplett schwarzes Bild). [32]	12
2.9	Beispielbild [3] in dem die Straße und der Himmel ein Großteil des Bildes ausmachen.	13
2.10	Beispiel einer Stopschilderkennung. Die Vorhersage ist im roten und der Ground-truth im grünen Rahmen dargestellt. [23]	13
2.11	Berechnung der IoU, indem der Schnitt beider Mengen durch die Vereinigung geteilt wird. [23]	14
3.1	Beispielbilder aus dem Cityscapes-Datensatz. [3]	18
4.1	ResNet-18 Version:Original, Vergleich der Metriken pixelweise Genauigkeit(84.7%) und mean-IoU(42.2%)	24
4.2	Auswirkung unterschiedlicher batchsize auf die mean-IoU	26
4.3	Modifiziertes fc mit unterschiedlichem Dropout	28
4.4	Modifiziertes fc mit Feinabstimmung, Vorher-nachher-Vergleich	30

4.5	Die ResNet-18-Netze mit ihren jeweiligen Ausgangs- und Endstadien . . .	31
4.6	Semantische Segmentierung auf dem Cityscapes-Datensatz mit den modifizierten Netzen. Parameter: <code>-filter-mode=point</code>	33
4.7	Auffällige Bilder bei der semantischen Segmentierung an einem Video mit den modifizierten Netzen. Parameter: <code>-filter-mode=linear</code>	35
4.8	Auffällige Bilder bei der semantischen Segmentierung an einem Video mit den modifizierten Netzen. Parameter: <code>-filter-mode=linear</code>	35
4.9	FPS Aufzeichnung der Netze im Einsatz auf dem Jetson Nano	36

Tabellenverzeichnis

2.1	Architekturen für das ResNet-Modell. Die eckigen Klammern weisen auf eine Layer-Gruppe hin. Dimensionsreduktion werden bei conv3_1, conv4_1, und conv5_1 eingesetzt. [12]	8
3.1	Cityscapes-Datensatz Klassendefinition [3]	18
4.1	Neuzuordnung der Cityscapes-Klassen. Aus 30 Klassen werden 21.	21
4.2	Erkennungsrate des ResNet-18 Modells mit den Metriken pixelweise Genauigkeit und mean-IoU	25
4.3	Maximale mIoU der unterschiedlichen Batchsize	25
4.4	Maximale mIoU der fc-Modifikation mit verschiedenen Dropout-Versionen	28
4.5	Maximale mIoU der Feinabstimmung vor- und nachher	29
4.6	Maximale mIoU der ResNet-18-Modelle mit und ohne Vortraining und die Auswirkung der Modifikation mit Feinabstimmung.	30
4.7	Farbkodierung für die Ausgabe der semantischen Segmentierung	33

Auflistungsverzeichnis

4.1	Befehle zur Installation des Projektes	20
4.2	Parameter für die Strukturierung der Trainingsdaten	20
4.3	Parameter für die Klassenreduktion der Trainingsdaten	21
4.4	Parameter zur Konfiguration des Trainings	22
4.5	Standardparameter zur Konfigurierung des Trainings	23
4.6	ResNet-18 Zusammenfassung (Version: Original)	23
4.7	ResNet-18 fc (Version: 0-Dropout)	27
4.8	ResNet-18 fc (Version: 2-Dropout)	27
4.9	ResNet-18 fc (Version: 4-Dropout)	28
4.10	Parameter für die Segmentantation eines Videos	32
4.11	Parameter für die Segmentantation über eine Live-Webcam	32
4.12	Parameter für die Segmentantation über einen RTSP-Stream	33

1 Einleitung

Das folgende Kapitel beschreibt die Motivation dieser Arbeit. Es beschreibt die Problemstellung und die Ziele die zu erreichen sind.

1.1 Motivation

Die Grundidee für diese Arbeit entstand mit der Überlegung einen Blindenführhund durch einen Roboter zu ersetzen. Für die Auswahl eines Blindenführhundes gibt es besondere Kriterien auf die geachtet werden muss, dazu zählen unter anderem Nervenstärke, Ängstlichkeit, Aggressionsverhalten und Jagdmotivation der Vierbeiner. Die Ausbildung jedes einzelnen Blindenführhundes dauert zwischen 6 bis 9 Monate[4]. Wäre es nicht möglich diesen Prozess mit dem momentanen Stand der Technik zu ersetzen? Folgendes Modell wurde entworfen:

Der Blindenführhund wird ersetzt durch einen kompakten Roboter, wie zum Beispiel einen Loomo Roboter von Segway[19]. Eine Kamera am Roboter dient dazu die Umgebung aufzuzeichnen. Das Kamerabild wird durch ein neuronales Netz berechnet und in unterschiedliche Bildsegmente aufgeteilt. Die Rechenleistung wird durch einen Einplatinencomputer von NVIDIA, dem Jetson Nano[20], zur Verfügung gestellt.

1.2 Problemstellung und Ziele

Damit der Roboter als Blindenführhund agieren kann muss es bestimmte Gefahren und Hindernisse erkennen können. Dabei liegt die Priorität beispielweise auf Personen, Autos und die Unterscheidung zwischen Gehweg und Fahrbahn. Damit diese Gefahren und Hindernisse rechtzeitig erkannt werden, muss das Kamerabild in Echtzeit verarbeitet werden.

Dies erfordert eine niedrige Reaktionszeit und gleichzeitig eine hohe Rechenleistung. Um mobil und kompakt agieren zu können wird auf ein vollwertigen Grafikprozessor verzichtet. Der Jetson Nano von Nvidia ermöglicht den Kompromiss zwischen reduzierter Rechenleistung und kompakter Bauform.

Um diese Ziele zu erreichen ist die grundlegende Funktionsweise von neuronalen Netzen vorauszusetzen. Zusätzlich verwendete Techniken und die dazu basierenden Grundlagen sind im nächsten Kapitel 2 Grundlagen und verwendete Techniken erläutert. Die Erhöhung der Erkennungsrate eingehend mit der Optimierung neuronaler Netze ist der hauptsächliche Bestandteil dieser Arbeit. Das Trainieren neuronaler Netze und die dafür notwendige Vorbereitung, der zu trainierenden Datensätze, sowie die Inbetriebnahme der Bildsegmentierung ist im Kapitel 4 Implementierung und Auswertung Schritt für Schritt erklärt. Als Programmiersprache wird Python für das Trainieren der neuronalen Netze verwendet.

2 Grundlagen und verwendete Techniken

2.1 Transfer-Lernen

Das Transfer-Lernen ist eine Lernmethode im Bereich des maschinellen Lernens. Es dient dazu das erlangte Wissen und die Ergebnisse einer Aufgabe auf eine neue Aufgabe zu transferieren. Neuronale Netze die bereits für ähnliche Aufgaben vortrainiert wurden dienen als Ausgangspunkte für das Training neuer Netze, denn nicht immer stehen für jede Aufgabe ausreichend Daten zur Verfügung. „Für künstliche Intelligenz ist Transfer-Lernen ein wichtiger Punkt, da Lernaufgaben wesentlich schneller abgeschlossen werden können und aufeinander aufbauen.“[34]

2.1.1 Vortrainierte Netze

Ein bekannter und hoch effizienter Ansatz, um neuronale Netze mit kleinen Datensätzen zu trainieren, ist die Verwendung von vortrainierten Netzen. Vortrainierte Netze wurden, wie man es aus dem Namen entnehmen kann, bereits mit einem groß angelegten Bildklassifizierungsdatensatz trainiert und abgespeichert. Dieses vortrainierte Netz kann somit allerlei Objekte aus dem Alltag erkennen und klassifizieren, sofern der verwendete Datensatz groß und allgemein genug ist. Diese Basis ist für neue Bildklassifizierungsaufgaben sehr von Vorteil. Zum Beispiel wird ein Netz mit dem ImageNet-Datensatz [14] trainiert welches hauptsächlich Tiere und Alltagsobjekte beinhaltet. Im Anschluss wird das selbe Netz umfunktioniert um verschiedene Möbel von einander zu unterscheiden. Die Übertragbarkeit der bereits gelernten Klassen zwischen verschiedenen Problemen gibt den entscheidenden Vorteil gegenüber vieler älterer Trainingsansätze. Gerade wenn nur sehr kleinen Datensätzen zur Verfügung stehen sind vortrainierte Netze sehr effektiv [2, S. 143].

2.1.2 Feinabstimmung

Eine weit verbreitete Technik die man auf vortrainierte Netze anwendet ist die Feinabstimmung. Beim Feinabstimmen ist es das Ziel, ein vortrainiertes Netz mit einem neuen Datensatz und einem neuen Fully-connected-classifier(kurz fc) zu trainieren. Es wird Feinabstimmung genannt weil es feine Anpassungen in den bereits vortrainierten Layer des Ausgangsnetzes erlaubt um diese relevanter für das neue Problem beziehungsweise für die neuen Datensätze zu machen [2, S. 152]. Dabei werden alle Layer des Netzes bis auf die letzten und das des fc eingefroren und somit für Veränderungen während des Trainieren blockiert.

Für die Feinabstimmung eines Netzes müssen fünf Schritte beachtet werden. [2, S. 154]

1. Einem vortrainierten Netz werden weitere Layer hinzugefügt oder der fc modifiziert.
2. Das vortrainierte Ausgangsnetz wird eingefroren und somit für Veränderungen während des Trainings blockiert.
3. Ausschließlich der Teil der neu hinzugefügt beziehungsweise modifiziert wurde wird trainiert.
4. Einige Layer aus dem Ausgangsnetz werden wieder für das Trainieren freigegeben.
5. Das Netz wird erneut trainiert.

2.2 Das Problem von tiefen neuronalen Netzen

Einem neuronalen Netz werden dann mehr Layer hinzugefügt, wenn man die Genauigkeit und Leistung erhöhen will. Der Gedanke dabei ist, dass bei steigender Anzahl der Layer auch die erlernte Informationsdichte eines Netzes steigt. Jedoch zeigen Versuche aus [12], wie in Abbildung 2.1 zu erkennen ist, dass die Fehlerrate bei Trainings- und Testdaten steigt. Deutlich zu sehen ist, dass das Netz mit 56-layer zu jedem Zeitpunkt schlechter abschneidet als das Netz mit 20-layer.

Doch eine Steigerung der Anzahl an Layern verursacht das Problem des verschwindenden Gradienten. In der Regel werden neuronale Netze mit Backpropagation trainiert.

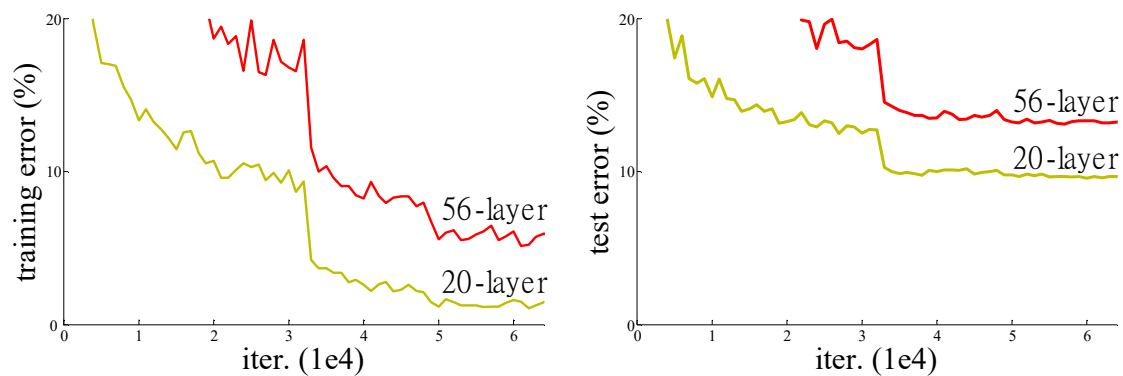


Abbildung 2.1: Trainings- und Testfehler bei 20- und 56-Layer [12]

Die Kettenregel der Differentialgleichung kommt zum Einsatz, um den Gradienten des nächsten Layers abzuleiten, dabei werden Gradienten aus früheren Layern multipliziert. Dieser Prozess findet wiederholt statt und führt dazu dass der mittlere Gradient in Richtung Eingangslayer einen immer kleineren Wert annimmt. Eine Steigerung der Anzahl der Layer erhöht auch die Anzahl der multiplizierten Gradienten und verstärkt somit dieses Problem zusätzlich. [33] Aufgrund des verschwindenden Gradienten kann das neuronale Netz keinen nennenswerten Fortschritt mehr nachweisen. In [12] wurde auch mit der Erhöhung der Trainingsiterationen experimentiert, doch dies ergab ebenfalls keine Verbesserung der Fehlerraten.

2.3 ResNet

Das neuronale Netz ResNet (Residual-Network) wurde 2015 von Kaiming He, Xiangyu Zhang, Shaoqing Ren und Jian Sun in ihrer Veröffentlichung „Deep Residual Learning for Image Recognition“ [12] vorgestellt. Das ResNet-Modell war ein großer Erfolg, wie zahlreiche Spitzenplätze bei renommierten Wettbewerben zeigen.

- #1 Platz im ILSVRC 2015 Klassifikationsherausforderung mit einer top-5-error Rate von 3,57%
- #1 Platz im ILSVRC und COCO 2015 Herausforderung in ImageNet-detection, ImageNet-localization, Coco-detection und Coco-segmentation

- Ersetzen der VGG-16 Layers in Faster-R-CNN durch das ResNet-101. Eine Verbesserung von 28% wurde beobachtet.
- Effizientes trainieren von Netzen mit einer Anzahl von 100 bis zu 1000 Layern.

ResNet-Modelle bedienen sich an neuen Methoden um, das Problem von tiefen neuronalen Netzen zu mindern. In einem ResNet-Modell werden mehrere Residual-Blocks eingesetzt [13], Abbildung 2.2. Die Eigenschaft eines Residual-Blocks ist das Überspringen von Layern auch genannt „Skip-connection.“ Skip-connections müssen die Dimension der Eingabe auf die Dimension der Ausgabe entsprechend anpassen. Eine Möglichkeit ist das Zero-padding, welches die Dimension mit weiteren Nullen erhöht. Diese Methode produziert keine weiteren Parameter für das Netz. Eine weitere Möglichkeit ist das Einsetzen von einem zusätzlichen 1x1 Layer, dies resultiert in weiteren Parametern je Anpassung der Dimension.

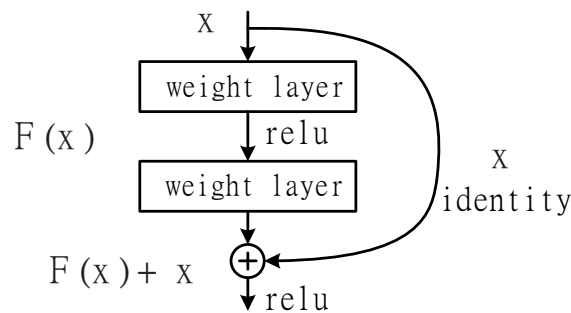


Abbildung 2.2: Eine „Skip-connection“ aus dem ResNet [12]

Mit den Skip-connections kann der Informationsfluss bis in die tiefen Layer eines Netzes fließen und erlauben somit tiefe neuronale Netze mit besseren Erkennungsraten zu erreichen [26]. Zudem wenden die Skip-connections die Identitätsfunktion an. (Abbildung 2.2 $F(x) + x$)

Nehmen wir zum Beispiel ein flaches und ein tiefes Netz, welches die Eingabe x zur Ausgabe y mit der Funktion $H(x)$ abbilden soll. Das Ziel ist es, dass das tiefere Netz nicht schlechter abschneidet als das flache Netz. Eine Möglichkeit ist es wenn die tieferen Layer im tiefen Netz die Identitätsfunktion erlernen und somit zu keiner Verminderung der Leistung, trotz zusätzlicher Layer, führen [12]. Herkömmliche Netze berechnen $H(x) = f(x)$ mit dem Ziel $f(x) = x$ um die Identitätsfunktion zu erreichen. Netze mit Residual

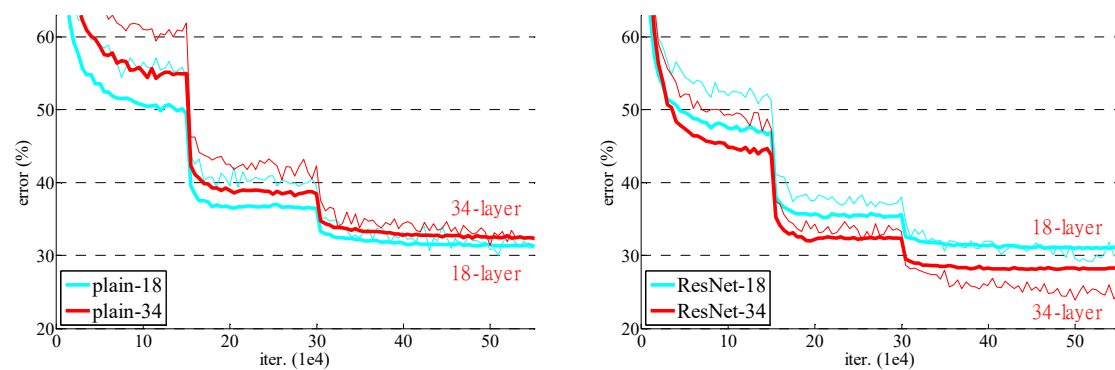


Abbildung 2.3: Training mit ImageNet-Datensatz. Vergleich zwischen ResNet-18 und -34 sowie die herkömmliche Variante (plain-18 und plain-34). Dünne Linien: Trainingsfehler. Dicke Linien: Validierungsfehler [12]

Blocks lernen die Identitätsfunktionen mit folgender Funktion: $H(x) = f(x) + x$. Sobald man im Optimalfall $f(x) = 0$ erreicht, ist die Eingabe gleich der Ausgabe $y = H(x) = x$. Mit diesem Konzept war es möglich tiefere Netze mit besseren Erkennungsraten zu erreichen. Abbildung 2.3 zeigt eindeutig den großen Mehrwert von einem ResNet-34 im Vergleich zu einem herkömmlichen Netz mit 34 Layern ohne Skip-connections (plain-34). Bei dem Vergleich der Netze mit 18 Layern (plain-18 und ResNet-18) erkennt man bei der Fehlerrate keinen Unterschied, aber das ResNet-18 konvergiert schneller als das plain-18.

2.3.1 Architektur des ResNet

In der Abbildung 2.4 sind die unterschiedlichen Architekturen von drei Modellen visualisiert. Das VGG-19 [29] dient bei der Veröffentlichung [12] als Referenz für die Entwicklung des ResNet-Modells. Das ResNet-Modell (Abb. 2.4 rechts) weist zu Beginn ein Convolutional-Layer und einen Pooling-Schritt auf, gefolgt von vier Layer-Gruppen. Jedes dieser vier Gruppen folgt dem gleichen Muster. Sie reichen die Eingabe nach jedem zweiten Layer über die Skip-connection weiter und behalten innerhalb ihrer Layer-Gruppen die Höhen- und Breitendimension konstant. Die gepunkteten Linien, zu Beginn einer neuen Layer-Gruppe, weisen auf eine Veränderung der Eingabedimension hin. Bei der Veränderung handelt es sich um eine Dimensionsreduktion die zwischen den Layer-Gruppen stattfindet. Erreicht wird dieses durch die Erhöhung des Stride-Werts von 1 auf 2. Während sich die Feature-map-Dimension halbiert, verdoppelt sich die Anzahl der Filter um die Zeitkomplexität pro Layer zu bewahren [12]. Dieser Vorgang wiederholt

2 Grundlagen und verwendete Techniken

sich abhängig von der Auswahl der Tiefe des ResNet-Modells. Die Tabelle 2.1 zeigt eine Zusammenfassung der internen Struktur der jeweiligen ResNet-Modelle (18-, 34-, 50-, 101-, 152-Layer).

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3 \times 3, 64 \\ 3 \times 3, 64 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 64 \\ 3 \times 3, 64 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$
conv3_x	28×28	$\begin{bmatrix} 3 \times 3, 128 \\ 3 \times 3, 128 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 128 \\ 3 \times 3, 128 \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 8$
conv4_x	14×14	$\begin{bmatrix} 3 \times 3, 256 \\ 3 \times 3, 256 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 256 \\ 3 \times 3, 256 \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 23$	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 36$
conv5_x	7×7	$\begin{bmatrix} 3 \times 3, 512 \\ 3 \times 3, 512 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 512 \\ 3 \times 3, 512 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Tabelle 2.1: Architekturen für das ResNet-Modell. Die eckigen Klammern weisen auf eine Layer-Gruppe hin. Dimensionsreduktion werden bei conv3_1, conv4_1, und conv5_1 eingesetzt. [12]

2.4 Semantische Segmentierung



(a) Ausgangsbild

(b) Semantische Segmentierung (Ground-truth)

Abbildung 2.5: Kreuzung mit Passanten [3]

Klassifikation eines Bildes, Objekterkennung im Bild und Segmentierung eines Bildes sind einige von vielen Aufgaben, die mit Hilfe von maschinellem Lernen, in der Bildverarbeitung angewendet werden.

Bei der Klassifikation eines Bildes ist man lediglich daran interessiert was auf dem Bild erkannt wurde. Die Objekterkennung in einem Bild geht ein Schritt weiter indem es zusätzlich die Objekte lokalisiert, einzeln einrahmt und somit deren Position wiedergibt. Bei der Segmentierung eines Bildes wird die exakte Grenze eines Objektes gesucht, welches eine deutlich höhere Präzision verlangt.

Im Grunde ist die Segmentierung eines Bildes auch eine Klassifikation eines Bildes. Der große Unterschied liegt darin, dass bei der Klassifikation ein Ergebnis pro Bild erwartet wird und bei der Segmentierung für jedes einzelne Pixel im Bild eine eigene Klassifikation durchgeführt wird. Die Segmentierung eines Bildes teilt sich in zwei Kategorien auf, die semantische Segmentierung und die Instanzsegmentierung. Die semantische Segmentierung gruppiert mehrere Objekte gleichen Typens zusammen. Wie auf der Abbildung 2.5b zu beobachten ist wird mehreren Passanten auf dem Gehweg gemeinsam ein Typ und somit eine Farbe zugeordnet. Die Instanzsegmentierung hingegen ordnet jedem erkannten Objekt eine eigenen Typ zu. Wie in der Abbildung 2.6 zu sehen wird jeder einzelnen Person ein eigener Typ zugeordnet.

Die pixelgenaue Klassifikation und die Gruppierung von gleichen Typen mit der semantischen Segmentierung ist ideal für das Vorhaben in dieser Arbeit, denn es ist nicht von Relevanz ob zum Beispiel drei Passanten erkannt werden sondern viel wichtiger ist wo im Bild sich diese Passanten befinden um ihnen ausweichen zu können.

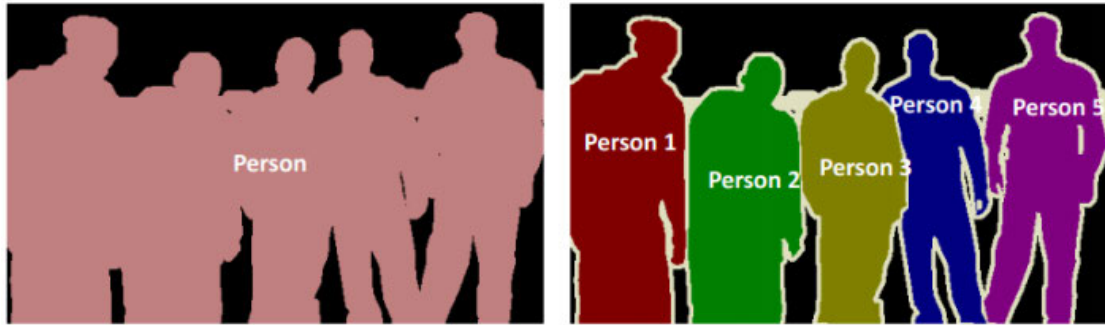


Abbildung 2.6: Semantische- (links) und Instanzsegmentierung (rechts) [27]

2.5 Metriken

Während neuronale Netze wiederholt trainiert werden, wird versucht das Netz mit der besten Leistung zu finden. Um diese sogenannte Leistung bewerten zu können sind Metriken vonnöten. Mit Hilfe von Metriken werden neuronale Netze bewertet und untereinander vergleichbar gemacht. Im folgenden sind die hier verwendeten Metriken vorgestellt.

2.5.1 Pixelweise Genauigkeit

Die pixelweise Genauigkeit ist im Grunde die intuitivste Metrik. Hierbei wird jeder einzelne Pixel auf die richtige Klassifikation überprüft. Diese Art der Messung kann prozentual hohe Genauigkeiten erreichen und trotzdem zu sehr schlechten Ergebnissen, bei der Klassifikation von Bildsegmenten, führen.

Man nehme an die Abbildung 2.7 rechts ist das Ground-truth-Bild der Abbildung 2.7 links. Ziel ist es die Schiffe im Hafen zu segmentieren. Wenn am Ende die Abbildung 2.8 als Ergebnis aus dem neuronalen Netz geliefert wird hat man eine pixelweise Genauigkeit von 95%. Obwohl die Schiffe überhaupt nicht erkannt wurden liegt die Fehlerrate nur bei 5%, denn der Anteil der Schiffe auf der Abbildung 2.7 links beträgt genau diese 5%. Der

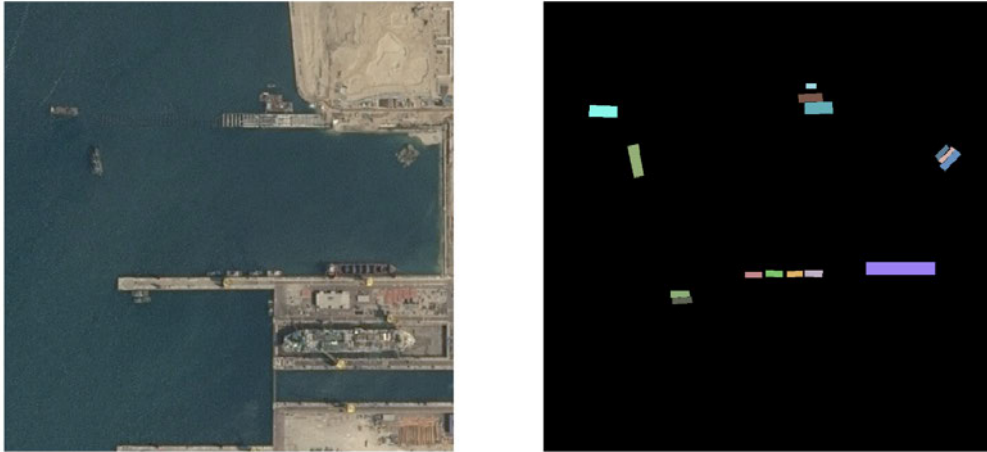


Abbildung 2.7: Satellitenaufnahme (links) und die dazugehörige Segmentierungsmaske (rechts) [28]

Rest des Bildes indem sich kein Schiff befindet wurde richtig als Hintergrund klassifiziert, welches die Erkennungsrate hoch drückt. Grund hierfür ist das Ungleichgewicht zwischen den Klassen, welches zu einer nicht geeigneten Metrik führt. [32]

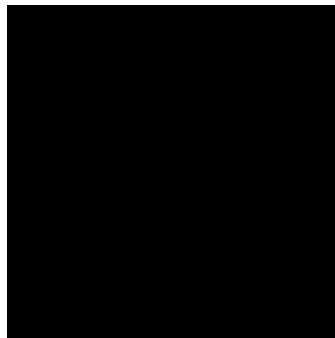


Abbildung 2.8: Vorhersage für die Segmentierung der Schiffe im Hafen (komplett schwarzes Bild). [32]

Die Trainingsbilder des Cityscapes-Datensatzes besitzen ebenfalls Bereiche im Bild die einen Großteil des Bildes ausmachen, wie zum Beispiel der Himmel oder die Fahrbahn, wie in Abbildung 2.9 zu sehen ist.

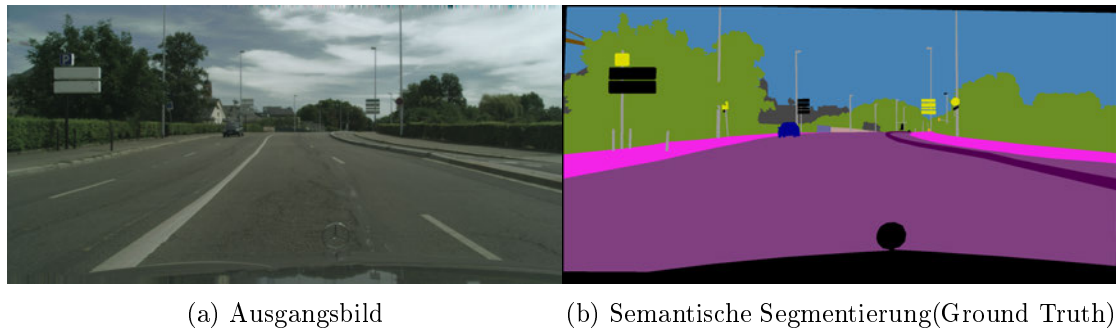


Abbildung 2.9: Beispielbild [3] in dem die Straße und der Himmel ein Großteil des Bildes ausmachen.

2.5.2 Intersection-Over-Union

Die Intersection-over-Union (kurz: IoU), auch bekannt als Jaccard Index, ist eine weitere Metrik die in der semantischen Segmentierung angewendet wird. Hierbei wird nicht jeder Pixel im einzelnen bewertet, sondern auf die Überschneidung zwischen der Vorhersage und dem Ground-truth geachtet.

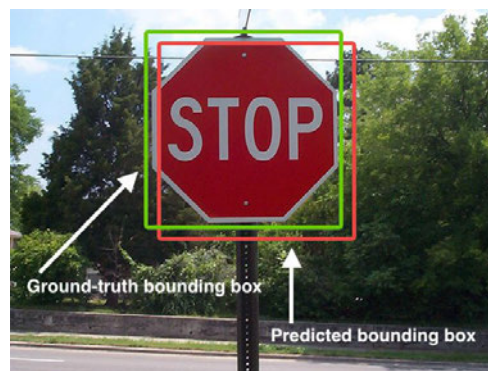


Abbildung 2.10: Beispiel einer Stoppschilderkennung. Die Vorhersage ist im roten und der Ground-truth im grünen Rahmen dargestellt. [23]

Auf der Beispiel Abbildung 2.10 ist zu erkennen wo die Vorhersage das Stoppschild erkannt hat und wo es eigentlich sein sollte. Anhand der Schnittmenge und der Vereinigungsmenge kann man mit Hilfe der IoU (Abbildung 2.11) die Treffgenauigkeit zwischen 0 und 1 bewerten. 0 bedeutet keine Übereinstimmung und 1 eine volle Übereinstimmung. Bei mehr als zwei Klassen die erkannt werden sollen wird der durchschnittliche IoU berechnet, indem der IoU jeder Klasse durch die Anzahl der Klassen geteilt wird. Dies wird mIoU genannt welches für mean-Intersection-over-Union steht.

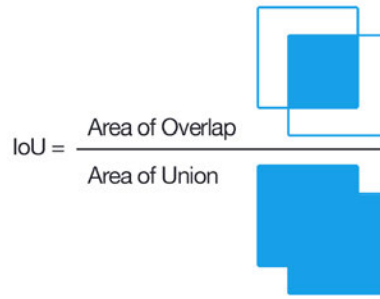


Abbildung 2.11: Berechnung der IoU, indem der Schnitt beider Mengen durch die Vereinigung geteilt wird. [23]

Vergleicht man das Beispielszenario mit den Schiffen aus dem Kapitel 2.5.1 erreicht man bei der mIoU einen Wert von 47,5%. Um die Rechnung zu vereinfachen wird angenommen, dass die Abbildung 2.8 aus 100 Pixel besteht. Demnach hat die Klasse Schiff eine Schnittmenge von 0 Pixel und eine Vereinigungsmenge von 5 Pixel. Die Klasse Hintergrund hat eine Schnittmenge von 95 Pixel und eine Vereinigungsmenge von 100 Pixel.

$$\text{Schiff-IoU} = \frac{\text{Schnittmenge}}{\text{Vereinigungsmenge}} = \frac{0p}{(5p + 0p) - 0p} = 0\% \quad (2.1)$$

$$\text{Hintergrund-IoU} = \frac{\text{Schnittmenge}}{\text{Vereinigungsmenge}} = \frac{95p}{(95p + 100p) - 95p} = 95\% \quad (2.2)$$

$$\text{mean-IoU} = \frac{\text{Schiff-IoU} + \text{Hintergrund-IoU}}{2} = \frac{0\% + 95\%}{2} = 47,5\% \quad (2.3)$$

2.6 ONNX - Open Neural Network Exchange

„Die Abkürzung ONNX steht für Open-Neural-Network-Exchange. Es handelt sich um ein offenes Format zur Repräsentation von Deep-Learning-Modellen in verschiedenen Frameworks. ONNX ist aus einem gemeinschaftlichen Projekt von Facebook und Microsoft entstanden, das mittlerweile von vielen weiteren Unternehmen wie IBM, Intel, AMD, Huawei, ARM, Qualcomm, NVIDIA und anderen unterstützt wird. Mit dem ONNX-Format ist es möglich, die Modelle ohne größeren Anpassungsaufwand zwischen verschiedenen Machine-Learning-Frameworks wie Caffe2, MXNet, Microsoft-Cognitive-Toolkit oder PyTorch zu verschieben.“[31]

3 Design

Im folgenden Kapitel wird die Systemarchitektur und dessen Komponenten vorgestellt.

3.1 Hardware

Die Architektur des Systems besteht aus Hardware- und Softwarekomponenten. Im folgenden Abschnitt werden zunächst die einzelnen Hardwarekomponenten bekannt gemacht.

3.1.1 NVIDIA Jetson Nano

„Jeder NVIDIA Jetson ist ein komplettes System-on-Module (SOM) inklusive CPU, GPU, Speicher, Energiemanagement, Hochgeschwindigkeitsschnittstellen und mehr. Jetson-Module sind in verschiedenen Kombinationen von Leistung, Energieeffizienz und Formfaktor erhältlich, sodass sie von Kunden aller Branchen verwendet werden können. Jetson-Ökosystempartner bieten Software, Hardware-Design-Services und kompatible Standardprodukte von Trägerplatinen bis hin zu vollständigen Systemen, sodass Sie Ihre Geräte mit eingebetteter KI und Edge-Geräte schneller auf den Markt bringen können.“[6]

Die günstigste Variante, mit einer UVP von 99 US-Dollar [5], der Jetson-Systeme ist der Nano. Es besitzt einen Quad-Core ARM-Cortex-A57 Prozessor mit 64-bit und einen Maxwell Grafikprozessor mit 128 Kernen. Beim Arbeitsspeicher wird auf ein LP-DDR4X Speicher mit 4GB gesetzt. Das X in LP-DDR4X weist auf einen Speicher hin dessen Spannung heruntergetaktet wurde. Somit wird bis zu 20 Prozent Energie für das Gesamtsystem gespart [30]. Folglich wird eine Rechenleistung von 472 GFLOPS erreicht und das bei einer Leistungsaufnahme von maximal 10W [17].

Die 128 Grafikprozessoren dienen dazu die Bilder in Echtzeit zu verarbeiten. Den Input für die Segmentation der Bilder erhält man über die USB 3.0 Anschlüsse, den MIPI-CSI-2 Anschluss oder über das Netzwerk gestreamt mit Hilfe des Gigabit-Ethernet-Anschlusses. Alternativ kann auch ein bereits aufgenommenes Videomaterial für Testzwecke verwendet werden.

3.1.2 Webcam

Die Logitech-Webcam C615 [18] kommt zum Einsatz um die Kamera eines Roboters zu simulieren. Die Webcam bietet eine maximale Auflösung von 1080p und eine Bildwiederholrate von bis zu 30 Bildern pro Sekunde an. Per USB-Anschluss kann die Webcam direkt an den Jetson Nano angeschlossen werden. Alternativ ist auch der Betrieb an einem externen Rechner möglich. Dabei wird das Bild über ein Netzwerk an den Jetson Nano gesendet, welches eine höhere Latenz zur Folge hat.

3.1.3 Trainingsrechner für das neuronale Netz

Das neuronale Netz wird auf einer Nvidia GTX 1660Ti Grafikkarte [11] berechnet. Diese Grafikkarte besitzt 1536 CUDA(Compute-Unified-Device-Architecture) Kerne und einen VRAM von 6GB. Da es sich hierbei um eine Mittelklasse-Grafikkarte aus meinem eigenen privaten Rechner handelt müssen im Trainingsskript einige Parameter angepasst werden. Dazu gehört die Reduzierung der Eingangsauflösung des neuronalen Netzes und die Begrenzung der Batchsize auf ein Maximum von 10. Grund hierfür ist die Limitierung des VRAMs. Im Vergleich dazu hat eine zu diesen Zeitpunkt (Q4/21) aktuellste Workstation Grafikkarte von Nvidia die RTX A6000 [21] 10752 CUDA-Recheneinheiten und bis zu 48GB VRAM. Desweiteren besitzt der Trainingsrechner eine Ryzen 2600 [24] CPU mit 6x 3,4 GHz und 16GB RAM.

Der Einsatz eines Jupyter-Notebooks wird hauptsächlich für Trainingssessions mit über 30 Epochen angewendet. Der Jupyter-Notebook hat Zugriff auf eine Workstation Grafikkarte die sich im Rechenzentrum der HAW befindet. Dabei handelt es sich um die NVIDIA V100s mit 5120 CUDA-Kernen und 32GB VRAM [22].

3.2 Software

Im folgendem Abschnitt wird die Funktionalität der einzelnen Softwarekomponenten erläutert und die Gründe für deren Auswahl genannt.

3.2.1 Trainingsskript

Das Trainigsskript welches als Grundlage für das Trainieren der Netze dient, stammt aus dem von den Nvidia-Entwicklern freigegebenem Github-Repository [25]. Ein Git-Fork [10] des Skriptes erweitert die Funktion um benutzerdefinierte Trainingsdatensätze zu verwenden. Zudem wurden folgende weitere Modifikationen vorgenommen. Die Möglichkeit Abschnitte der Netze für das Trainieren einzufrieren und die Ersetzung des Klassifikators am Ende eines Netzes. [9]

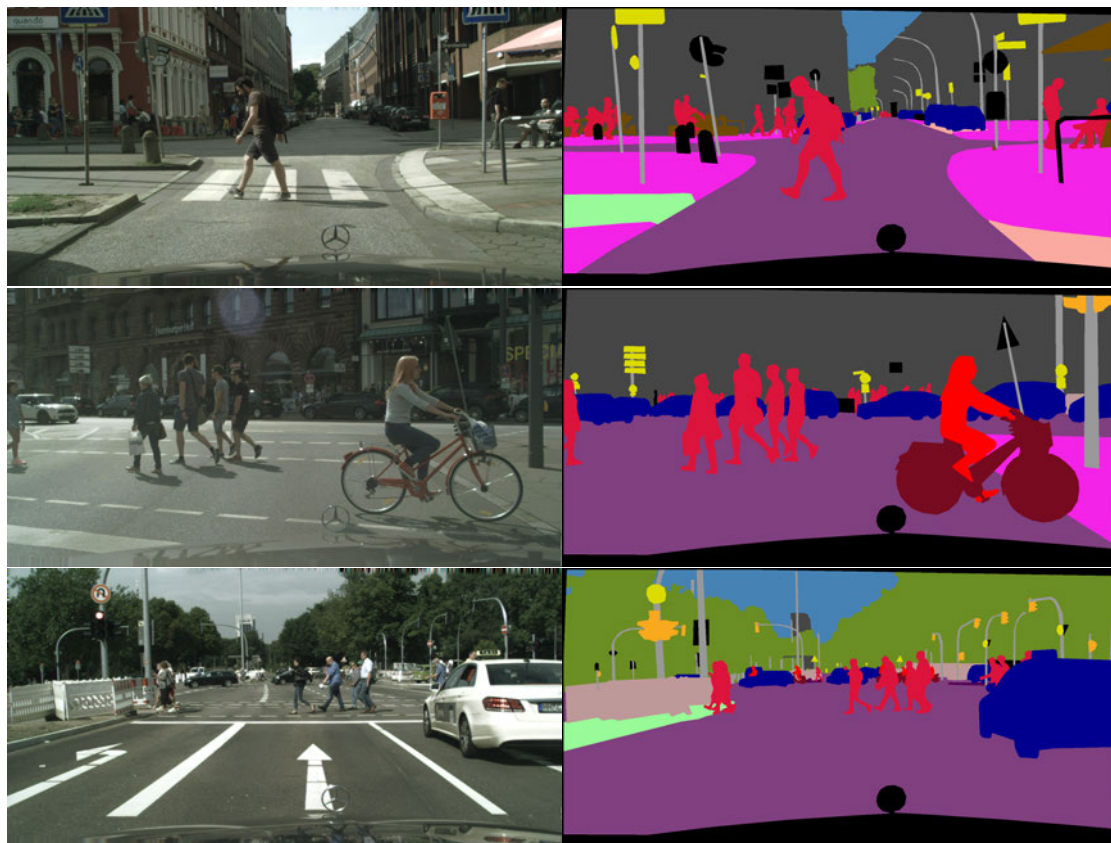
3.2.2 Cityscapes-Datensatz

Der Cityscapes-Datensatz bietet eine Vielzahl von Bild- und Videomaterial aus dem deutschen Straßenverkehr. Der Datensatz besteht aus insgesamt 50 deutschen Großstädten (unter anderem aus Hamburg, Berlin, München, Köln und Frankfurt am Main). Bis zu 5000 Bilder sind pixelweise semantisch einem von 30 Typen zugeordnet. Diese 30 Typen unterscheiden sich in Gruppen und untergeordneten Klassen. Gruppiert wird in Fläche, Mensch, Fahrzeug, Konstruktion, Objekt, Natur und Himmel. Menschen werden in Person oder Fahrer unterteilt. Fahrzeuge in PKW, LKW, Bus, Schienenverkehr, Motorrad, Fahrrad, Wohnwagen und Anhänger. Zudem variiert auch die Jahreszeit der aufgenommenen Szenarien [3]. Beispielbilder aus dem Datensatz mit den dazu gehörigen semantischen Segmentierungen(Ground-truth) sind in Abbildung 3.1 dargestellt.

Ein neuronales Netz kann nur das wiedererkennen was es bereits gesehen beziehungsweise gelernt hat, welches für die Wahl lokaler und realitätsnaher Datensätze aus dem deutschen öffentlichen Verkehrs spricht.

Gruppe	Klasse
Fläche	Straße, Gehweg, Parkplatz, Schienen
Mensch	Person, Fahrer
Fahrzeug	PKW, LKW, Bus, Schienenfahrzeug, Motorrad, Fahrrad, Wohnwagen, Anhänger
Konstruktion	Gebäude, Wand, Zaun, Leitplanke, Brücke, Tunnel
Objekt	Stange, Stangen(mehrere beieinander), Verkehrszeichen, Verkehrsampel
Natur	Vegetation, Gelände
Himmel	Himmel
Nichts	Boden, dynamische Obj.(z.b. Tiere), statische Obj.(z.B. Aufnahmefahrzeug)

Tabelle 3.1: Cityscapes-Datensatz Klassendefinition [3]



(a) Ausgangsbild

(b) Semantische Segmentierung(Ground-truth)

Abbildung 3.1: Beispielbilder aus dem Cityscapes-Datensatz. [3]

4 Implementierung und Auswertung

Im folgenden Kapitel werden die einzelnen Besonderheiten, Schritte und Vorbereitung für die Implementierung dieser Arbeit beschrieben. Unter anderem das Trainieren der neuronalen Netze, die Inbetriebnahme des Nvidia Jetson Nano, die visuelle Ausgabe der neuronalen Netze im Betrieb und mit welcher Metrik die Netze verglichen werden.

4.1 Einrichten des Nvidia Jetson Nano

Für die Inbetriebnahme des Nvidia Jetson Nano gibt es eine ausführliche Anleitung von Nvidia [7], dennoch gibt es einige Besonderheiten auf die geachtet werden müssen. Bei der Auswahl der microSD-Karte ist eine Mindestspeicherkapazität von 32GB und der UHS-I-Standard empfohlen, hier ist ein 64GB UHS-I-Speicherkarte im Einsatz. Für die Stromversorgung gibt es zwei Möglichkeiten. Erstens der Micro-USB-Eingang oder zweitens mit dem Hohlsteckereingang. Für eine stabile Stromversorgung wird der Hohlsteckereingang empfohlen. Beim Betrieb über ein MicroUSB-Netzteil sind Abstürze des Systems möglich. Ein Gleichstromnetzteil mit einer Versorgung von 5 Volt und 5 Ampere sorgt für eine ausreichende Stromversorgung im Betrieb.

Nachdem die Speicherkarte mit der Jetson-Nano-Developer-Kit-SD-Card-Image [16] beschrieben und ins Jetson Nano eingesteckt wurde, kann der erste Start des Jetson Nano vorgenommen werden. Vorher sollte ein Monitor sowie Eingabegeräte angesteckt werden. Nach der automatischen Installation wird das Github-Repository [8] benötigt. Eine detaillierte Anleitung ist im Repository aufgelistet [1].

Hier eine Kurzfassung der benötigten Terminal Befehle:

```
sudo apt-get update
sudo apt-get install git cmake libpython3-dev python3-numpy
git clone --recursive https://github.com/dusty-nv/jetson-inference
cd jetson-inference
mkdir build
cd build
cmake ../
make -j $(nproc)
sudo make install
sudo ldconfig
```

Auflistung 4.1: Befehle zur Installation des Projektes

4.2 Datenvorverarbeitung

4.2.1 Herunterladen der Trainingsdatensätze

Für das Herunterladen der Datensätze aus der Cityscapes-Webseite [3] ist eine vorherige Registrierung notwendig. Nach erfolgreicher Registrierung werden die Dateien *gtFine_trainvaltest.zip* und *leftImg8bit_trainvaltest.zip* benötigt. Im Anschluss entpackt man die ZIP-Dateien mit einem Packprogramm wie WinRAR [35]. Im entpacktem Zustand erhält man zwei Ordner namens *gtFine* und *leftImg8bit*. Der *leftImg8bit*-Ordner beinhaltet die Kameraaufnahmen und im *gtFine*-Ordner findet man die Ground-truth-Daten. In unserem Fall werden die Ground-truth-Daten mit der Endung *gtFine_labelIds.png* benötigt, da diese für jeden Pixel im Bild die richtige Klassen-ID vorgibt.

4.2.2 Ordnerstruktur der Trainingsbilder

Bevor man mit dem Training der Netze startet, müssen die Trainingsdaten in die richtige Ordnerstruktur transferiert werden. Hierfür kommt das *split_custom.py* [9] Skript zum Einsatz. Mit der Angabe der Verzeichnispfade der Ground-truth-Daten und der Kameraaufnahmen werden die Daten in Trainings- und Validierungsdaten unterteilt. 80% der Daten dienen als Trainingsdaten und 20% für die Validierung. Diese prozentuale Aufteilung kann bei Bedarf im Skript vorher angepasst werden.

```
python split_custom.py
--masks="Verzeichnis/der/Ground-Truth-Daten"
--images="Verzeichnis/der/Kameraaufnahmen"
```

```
|| --output="Verzeichnis/der/neuen/Ordnerstruktur"
```

Auflistung 4.2: Parameter für die Strukturierung der Trainingsdaten

4.2.3 Klassenreduktion

Aus dem Cityscapes-Datensatz werden insgesamt 3475 Bilder aus dem deutschen Straßenverkehr verwendet. Unterteilt sind diese Bilder in 2780 Trainingsbilder(80%) und 695 Testbilder(20%). Wie bereits im Kapitel 3.2.2 erwähnt klassifiziert der Cityscapes-Datensatz 30 unterschiedliche Klassen. Einige der Klassen werden zusammengefasst um die Klassenanzahl von 30 auf 21 herabzusetzen. Beispielweise werden die Klassen Person und Fahrer zusammengefasst. LKW, Wohnwagen und Anhänger gehören einer Klasse an, sowie Brücken und Tunnel. Grund hierfür ist das Ausgangsnetz der von den Nvidia-Entwicklern zum Einsatz kommt. Da Sie mehr als einen Datensatz (CityscapesDeepScene, Multi-Human, Pascal VOC, SUN RGB-D) analysiert haben war eine Reduzierung und somit eine vereinheitlichung der Ausgabeklassen notwendig. Die Klassenreduktion wird mit dem *dataset/cityscapes_remap.py* [9] Skript durchgeführt.

```
|| python datasets/cityscapes_remap.py "Verzeichnis/der/Ground-Truth-Daten"
```

Auflistung 4.3: Parameter für die Klassenreduktion der Trainingsdaten

kombinierte Klassen	Ursprungsklassen
LKW	LKW, Wohnwagen, Anhänger
Person	Person, Fahrer
Verkehrszeichen	Verkehrszeichen, Schilder, Stangen
Zaun	Zaun, Leitplanke
Straße	Straße, Parkplatz, Schienen, Brücke, Tunnel
Boden	Boden, dynamische Obj.(z.b. Tiere), statische Obj.(z.b. Berge)
Nichts	nicht klassifiziert, Verzerrung, Außerhalb des Bildes
nicht kombiniert	Fahrrad, Motorrad, Schienenfahrzeug, Bus, PKW, Himmel, Gelände, Vegetation, Verkehrsschild, Ampel, Wand, Gebäude, Gehweg, Aufnahmefahrzeug

Tabelle 4.1: Neuuzuordnung der Cityscapes-Klassen. Aus 30 Klassen werden 21.

4.3 Trainieren der neuronalen Netze

Das neuronale Netz wird nicht auf dem Jetson Nano trainiert, sondern auf einem leistungsstärkeren Rechner wie im Kapitel 3.1.3 bereits erwähnt. Das neuronale Netz ResNet-18 wird mit dem vorverarbeiteten Cityscapes-Datensatz trainiert. In den folgenden Unterkapiteln sind die einzelnen Modifikationen an dem ResNet-18 aufgeführt. Dabei handelt es sich nicht nur um zusätzliche Layer sondern auch ob die einzelnen Layer für das Lernen freigegeben sind oder nicht. Ein Überblick über die Modellstruktur ist mit Hilfe des Python *print*-Befehls erstellt. Für die Konfiguration des Trainingskriptes werden folgende Startparameter im *train.py* [9] Skript verwendet.

```
python train.py
<Datensatzpfad>
—dataset <Datensatztyp>
—width <Breite der Datensatzbilder>
—height <Höhe der Datensatzbilder>
—classes <Anzahl der Klassen>
—model-dir <Ausgabepfad des neuronalen Netzes>
—workers <Anzahl der Prozesse für das Laden der Datensätze>
—batch-size <Batchsize>
—epochs <Epochen>
—resume <Pfad des Ausgangsnetzes>
—lock <Einfrieren von Layern ,
    0: giebt alle Layer frei ,
    1: friert das Basisnetz ein ,
    2: (für Feinabstimmung) friert Basisnetz außer der letzten Layer-Gruppe ein>
—clear-classifier <Klassifikatorbereinigung ,
    0: behält die Klassifikatorgewichte ,
    1: entfernt die Klassifikatorgewichte>
```

Auflistung 4.4: Parameter zur Konfiguration des Trainings

4.3.1 ResNet-18 als Ausgangspunkt

Die Auflösung der Trainingsbilder, aus dem Cityscapes-Datensatz, werden vor dem Ein-speisen ins Netz von 2048 x 1024 auf 1024 x 512 Pixel herabgesetzt, um eine Entlastung des verfügbaren Arbeitsspeichers zu erreichen. Dies wird mit den Width- und Height-Parametern realisiert. Zudem wird der Workers-Parameter auf 1 gesetzt. Für die Konfiguration des ResNet-18 als Ausgangspunkt werden die vorgegebenen Konfigurationswerte verwendet. Diese lauten wie folgt:

```
python train.py
D:/Cityscapes_train/split/
--dataset custom
--width 1024
--height 512
--classes 21
--model-dir modelCityscapes
--workers 1
--batch-size
--epochs 30
--resume
--lock 0
--clear-classifier 0
```

Auflistung 4.5: Standardparameter zur Konfigurierung des Trainings

Das unveränderte ResNet-18 dient als Referenz und Ausgangspunkt für die modifizierten Netze. Hauptsächlich wird bei den modifizierten Netzen das Ende des Netzes bearbeitet, daher wird der unveränderte Teil in den weiteren Zusammenfassungen nicht wiederholt dargestellt.

```
FCN(
    (backbone): IntermediateLayerGetter(
      (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
      (layer1): Sequential(
        (0): BasicBlock(
          (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
          (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          (relu): ReLU(inplace=True)
          (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
          (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        )
        (1): BasicBlock(
          (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
          (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          (relu): ReLU(inplace=True)
          (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
          (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        )
      )
      (layer2): Sequential(
        (0): BasicBlock(
          (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
          (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          (relu): ReLU(inplace=True)
          (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
          (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          (downsample): Sequential(
            (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=False)
            (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          )
        )
        (1): BasicBlock(
          (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
          (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          (relu): ReLU(inplace=True)
          (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
          (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        )
      )
      (layer3): Sequential(
        (0): BasicBlock(
          (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
          (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          (relu): ReLU(inplace=True)
          (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
          (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          (downsample): Sequential(
            (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2), bias=False)
```

```

        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (1): BasicBlock(
      (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  (layer4): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (downsample): Sequential(
      (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
      (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (1): BasicBlock(
    (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  )
)
(classifier): FCNHead(
  (0): Conv2d(512, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (2): ReLU(inplace=True)
  (3): Dropout(p=0.1, inplace=False)
  (4): Conv2d(128, 21, kernel_size=(1, 1), stride=(1, 1))
)
)

```

Auflistung 4.6: ResNet-18 Zusammenfassung (Version: Original)

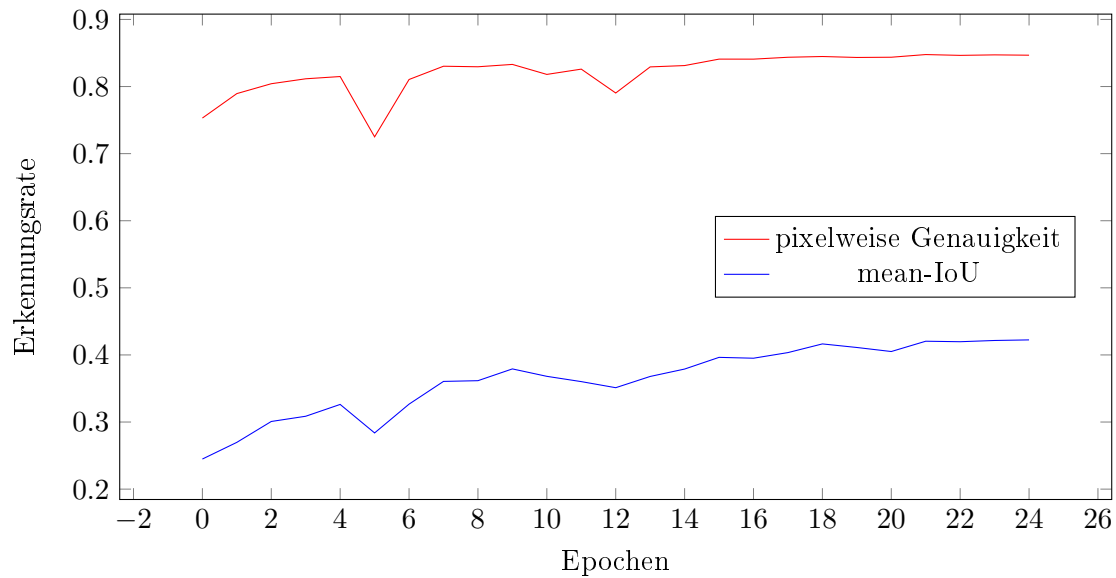


Abbildung 4.1: ResNet-18 Version:Original, Vergleich der Metriken pixelweise Genauigkeit(84.7%) und mean-IoU(42.2%)

Versionen	Erkennungsrate
pixelweise Genauigkeit	0,84765
mean-IoU	0,42241

Tabelle 4.2: Erkennungsrate des ResNet-18 Modells mit den Metriken pixelweise Genauigkeit und mean-IoU

4.3.1.1 Auswertung

Das unveränderte ResNet-18 als Ausgangsnetz erreicht bei der pixelweise Genauigkeit eine Erkennungsrate von 84.7% und bei der mean-IoU-Metrik eine Erkennungsrate von 42.2%. Die pixelweise Genauigkeit erreicht eine höhere Prozentzahl als die mean-IoU-Metrik, dennoch wird bei den nächsten Szenarien nur noch auf die mean-IoU-Metrik geachtet. Grund hierfür ist die Problematik die entsteht, dass trotz einer hohen Prozentzahl das Ergebniss nicht aussagekräftig ist, wie in dem Beispiel mit den Schiffen im Kapitel 2.5 Metriken. Dieses Ereigniss trifft auch hier in der Abbildung 4.1 auf.

4.3.2 ResNet-18: Unterschiedliche Batchsize

Das ResNet-18 als Ausgangsnetz wurde mit einer Batchsize von 4 trainiert. Dieser ist der Standardwert, sofern kein Parameter zur Auswahl der Batchsize ausgewählt wird. Ziel ist es die Auswirkungen auf die mean-IoU-Metrik zu beobachten bei einer Auswahl unterschiedlicher Batchsize. Folgende Batchsize werden analysiert: 1, 2, 4, 6, 8 und 10. Zu beachten ist dass die x-Achse nicht die Epochen darstellt, sondern die Anzahl der Iterationen indem die Gewichte der Layer aktualisiert werden. Wenn man dies mit Epochen gleichstellt muss man die Epoche 10 bei einer Batchsize von 10 mit der Epoche 1 bei einer batchsize von 1 vergleichen, denn erst dann haben beide Netze die gleiche Anzahl an Iterationen erhalten.

batchsize	1	2	4	6	8	10
max mIoU	0,41124	0,43621	0,43712	0,42269	0,42385	0,42166

Tabelle 4.3: Maximale mIoU der unterschiedlichen Batchsize

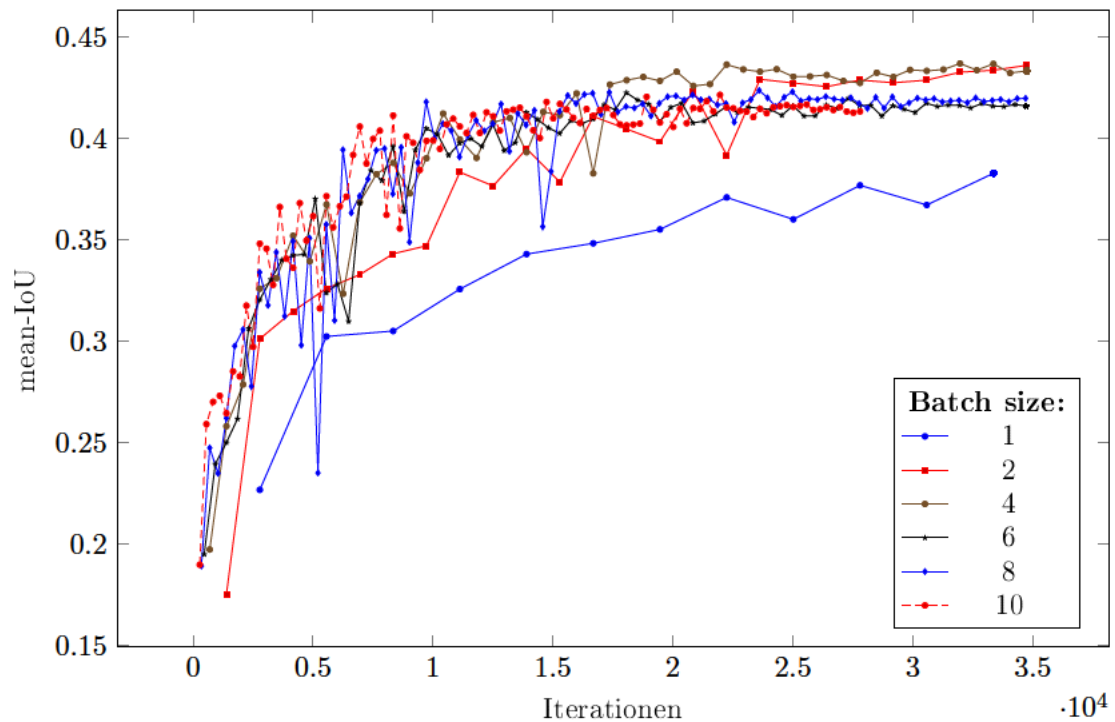


Abbildung 4.2: Auswirkung unterschiedlicher batchsize auf die mean-IoU

4.3.2.1 Auswertung

In der Abbildung 4.2 und in Tabelle 4.3 ist zu erkennen, dass das Netz mit einer Batchsize von 4 den höchsten mIoU erreicht hat. Bei der Batchsize von 4 handelt es sich um den Standardwert der bereits von den Nvidia-Entwicklern voreingestellt ist. Somit hat eine Änderung der Batchsize im Bereich zwischen 1 bis 10 keinen positiven Einfluss auf die Erkennungsrate des ResNet-18 mit der mean-IoU-Metrik.

Das Netz mit der Batchsize 4 hat seinen höchsten mIoU nach 31970 Iterationen erreicht, welches 46 Epochen entspricht. Trainiert wurde es für 50 Epochen, bei 695 Iterationen pro Epoche entspricht es insgesamt 34750 Iterationen. Für die weiteren Szenarien wurde die Anzahl der Epochen auf 50, pro Training, beschränkt um Zeit und Ressourcen zu sparen.

4.3.3 ResNet-18: Auswahl des modifizierten fc

In diesem Szenario werden die fc-Layer (Fully-connected-classifier) des ResNet-18 entfernt und durch ein neue Layer-Gruppe ausgetauscht. Als Basis dient das bereits im vorherigen Szenario trainierte ResNet-18 als Ausgangspunkt. Es wird nur das neu hinzugefügte fc trainiert, das restliche Netz wird eingefroren damit es für Veränderungen blockiert ist. Somit kann ein eindeutiger Vergleich zwischen den unterschiedlichen fc ermittelt werden.

Bei der Beobachtung des ursprünglichen fc ist aufgefallen, dass eine schnelle Herabstufung der Kanäle stattfindet. Das Convolutional-Layer (0) im FCNHead, in der Auflistung 4.6 ResNet-18 Zusammenfassung (Version: Original), viertelt einmalig die Anzahl der Kanäle von 512 auf 128 und das Convolutional-Layer (4) sechstelt ungefähr diese Anzahl von 128 auf 21. Infolgedessen entstand die Idee diese Herabstufung langsamer und gleichmäßiger zu verteilen. Die bisherigen beiden Convolutional-Layer wurden mit drei weiteren Layern erweitert, wobei jeder dieser Layer die Anzahl der Kanäle nur noch halbiert. Zudem wurde die unterschiedliche Verteilung der Dropout-Layer in den neu hinzugefügten Layern analysiert. In der ersten Version befinden sich keine Dropout-Layer (Auflistung 4.7), in der zweiten Version sind nach den ersten beiden Convolutional-Layern Dropout-Layer zu finden (Auflistung 4.8) und die dritte Version beinhaltet nach jedem Convolutional-Layer ein Dropout-Layer (Auflistung 4.9).

```
(classifier): FCNHead(
  (0): Conv2d(512, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (2): ReLU()
  (3): Conv2d(256, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (4): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (5): ReLU()
  (6): Conv2d(128, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (7): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (8): ReLU()
  (9): Conv2d(64, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (10): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (11): ReLU()
  (12): Conv2d(32, 21, kernel_size=(1, 1), stride=(1, 1))
)
```

Auflistung 4.7: ResNet-18 fc (Version: 0-Dropout)

```
(classifier): FCNHead(
  (0): Conv2d(512, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (2): ReLU()
  (3): Dropout(p=0.1, inplace=False)
  (4): Conv2d(256, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (5): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (6): ReLU()
  (7): Dropout(p=0.1, inplace=False)
  (8): Conv2d(128, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (9): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (10): ReLU()
  (11): Conv2d(64, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (12): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (13): ReLU()
  (14): Conv2d(32, 21, kernel_size=(1, 1), stride=(1, 1))
)
```

Auflistung 4.8: ResNet-18 fc (Version: 2-Dropout)

```
( classifier ): FCNHead(
  (0): Conv2d(512, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (2): ReLU()
  (3): Dropout(p=0.1, inplace=False)
  (4): Conv2d(256, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (5): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (6): ReLU()
  (7): Dropout(p=0.1, inplace=False)
  (8): Conv2d(128, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (9): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (10): ReLU()
  (11): Dropout(p=0.1, inplace=False)
  (12): Conv2d(64, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (13): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (14): ReLU()
  (15): Dropout(p=0.1, inplace=False)
  (16): Conv2d(32, 21, kernel_size=(1, 1), stride=(1, 1))
)
```

Auflistung 4.9: ResNet-18 fc (Version: 4-Dropout)

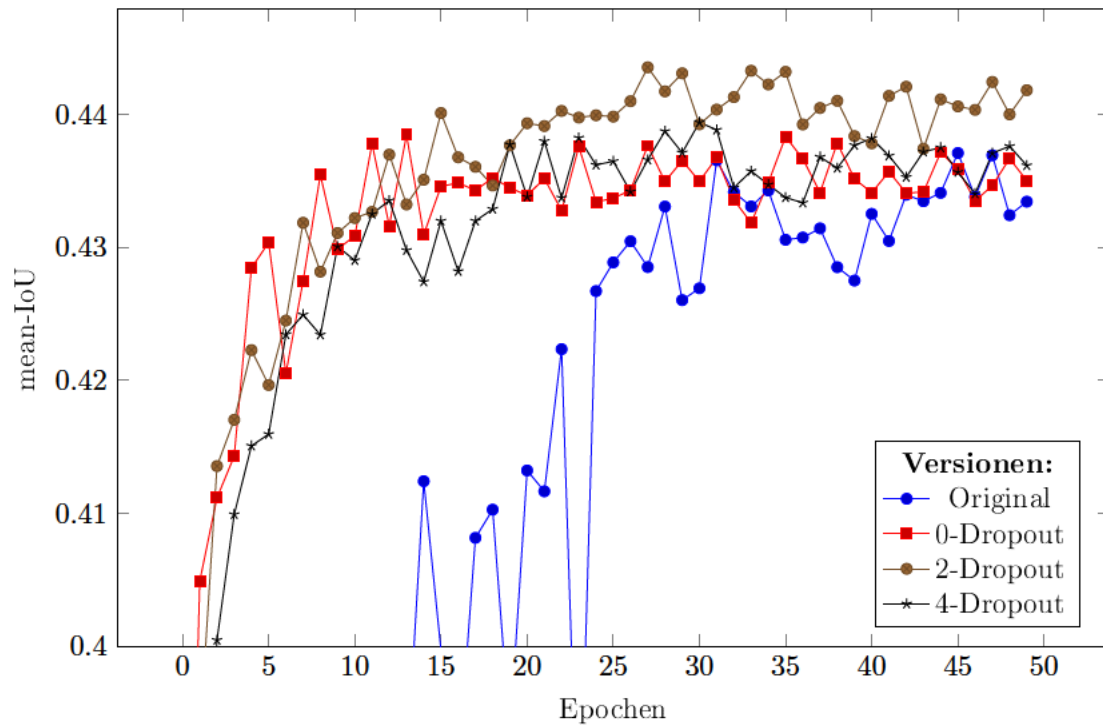


Abbildung 4.3: Modifiziertes fc mit unterschiedlichem Dropout

Versionen	ohne mod.	mod.+ 0-Dropout	mod.+ 2-Dropout	mod.+ 4-Dropout
max mIoU	0,43712	0,43848	0,44357	0,43943

Tabelle 4.4: Maximale mIoU der fc-Modifikation mit verschiedenen Dropout-Versionen

4.3.3.1 Auswertung

Unter den drei modifizierten fc-Versionen hat die Version mit 2-Dropout (Auflistung 4.8) mit Abstand das höchste mIoU von 44,36% erreicht und wird daher bei den nächsten Szenarien eingesetzt. Ab etwa Epoche 20 hat es mindestens die gleiche oder eine höhere mIoU, als die Versionen mit 0- und 4-Dropout, erreicht. Ein weiterer Möglichkeit der in [36] untersucht wurde ist das Einfügen von Dropouts innerhalb der Skip-connections, welches weitere Verbesserungen in der Testfehlerrate verspricht.

4.3.4 ResNet-18: Modifiziertes fc + Feinabstimmung

Mit der Auswahl des modifizierten fc aus dem vorherigen Kapitel wird die Feinabstimmung durchgeführt. Drei der fünf Schritte für die Feinabstimmung, die im Kapitel 2.1.2 aufgelistet wurden, sind bereits abgeschlossen. Im ersten Schritt wurde der fc modifiziert, im zweiten Schritt wurde das Basisnetz eingefroren und im dritten Schritt ein weiterer Trainingsdurchlauf gestartet. Dies erfolgte im Unterkapitel 4.3.3. Für die letzten zwei Schritte werden einige weitere Layer wieder für das Training freigegeben und das Netz ein weiteres mal trainiert. Dabei handelt es sich um die Layer-Gruppe (layer4) (Auflistung 4.6) die sich vor dem modifizierten fc befindet.

Feinabstimmung	vorher	nachher
max mIoU	0,44357	0,4465

Tabelle 4.5: Maximale mIoU der Feinabstimmung vor- und nachher

4.3.4.1 Auswertung

Im Vorher-nachher-Vergleich, in Abbildung 4.4, erreicht das ResNet-18 mit freigeschalteter Layer-Gruppe layer(4) (Auflistung 4.6) zu Beginn eine höhere mIoU als es vorher der Fall war. Dies geschieht auf Grund der Schwankungen innerhalb der ersten 20 Epochen. Nach dem abklingen der Schwankungen pendelt sich die Erkennungsrate wieder zurück auf den Ausgangswert von etwa 44,35%. Daraus kann man schließen, dass die Feinabstimmung keinen Mehrwert erbracht hat.

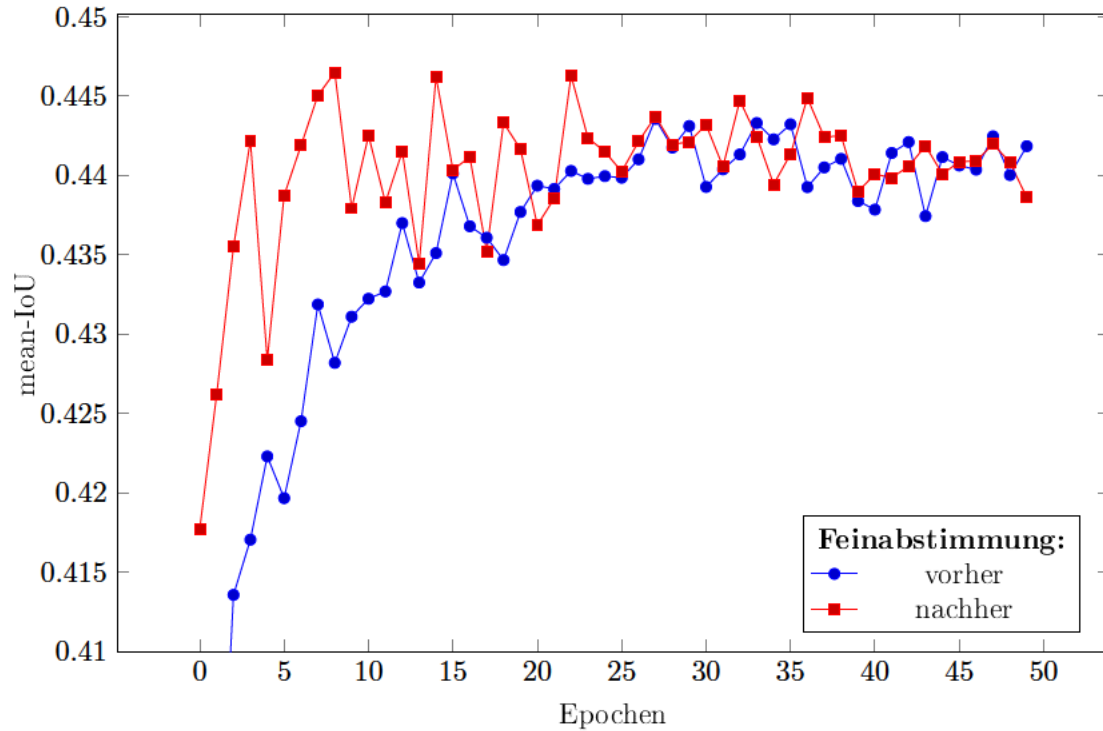


Abbildung 4.4: Modifiziertes fc mit Feinabstimmung, Vorher-nachher-Vergleich

4.3.5 ResNet-18: Vortrainiert + Modifiziertes fc + Feinabstimmung

In diesem Szenario wird der vorherige Versuch aus 4.3.4 wiederholt. Mit der wesentlichen Änderung, dass ein vortrainiertes Netz als Ausgangsnetz verwendet wird. Dabei handelt es sich um ein ResNet-18-Netz welches mit dem ImageNet-Datensatz [14] vortrainiert wurde. Im weiteren wird das vortrainierte ResNet-18 mit vResNet-18 abgekürzt. In Abbildung 4.5 sind die Endergebnisse mit ihren jeweiligen Ausgangszuständen für einen abschließenden Vergleich dargestellt.

Versionen	max mIoU
ResNet-18	0,43712
ResNet-18 + mod. fc + feinab.	0,4465
vResNet-18	0,54169
vResNet-18 + mod. fc + feinab.	0,55261

Tabelle 4.6: Maximale mIoU der ResNet-18-Modelle mit und ohne Vortraining und die Auswirkung der Modifikation mit Feinabstimmung.

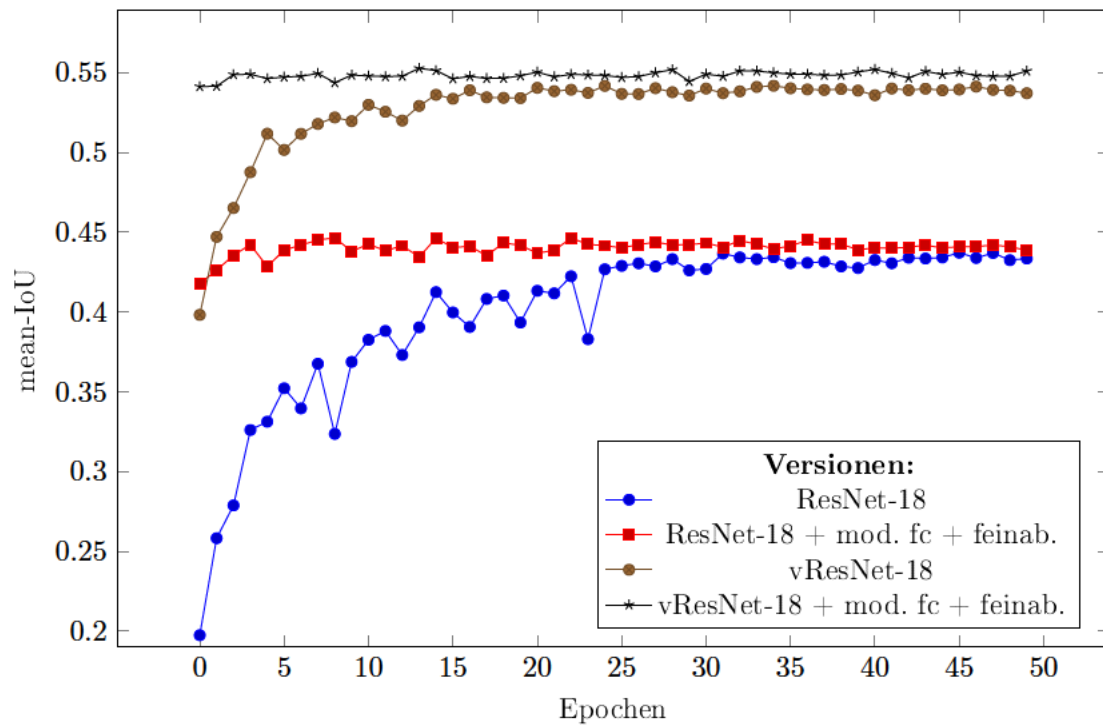


Abbildung 4.5: Die ResNet-18-Netze mit ihren jeweiligen Ausgangs- und Endstadien

4.3.5.1 Auswertung

Das vortrainierte Netz vResNet-18 hat eine mIoU von 54,2% erreicht. Nach den Modifikationen des fc und der Feinabstimmung konnte die mIoU auf 55,3% gesteigert werden. Vergleicht man das vResNet-18 mit dem ResNet-18 ist eine maximale Erhöhung der mIoU von 43,7% auf 54,2% zu registrieren. Zu beachten ist dass in beiden Netzen, ob vortrainiert oder nicht, die Experimente aus dieser Arbeit eine Verbesserung um etwa 1% erzielten. Das Anwenden eines vortrainierten Netzes alleine erzielte eine Verbesserung von 10,5%.

4.4 Neuronales Netz sichern und übertragen

Nachdem alle Epochen eines neuronalen Netzes durchgelaufen sind wird das Netz mit der besten mIoU-Metrik unter dem Namen *model_best.pth* im Ausgabepfad gespeichert. Dieses muss im nächsten Schritt in das ONNX-Format [31] umgewandelt werden. Hierzu dient das *onnx_export.py* Skript [9]. Mit der Angabe des Pfades als Parameter indem sich die *model_best.pth* befindet wird diese in das ONNX-Format umgewandelt. Mit dem ONNX-Format wird nicht nur die Dateigröße komprimiert sondern auch für den Betrieb, auf dem Jetson Nano, optimiert und vorbereitet.

4.5 Das neuronale Netz in Betrieb

Um das neuronale Netz auf dem Jetson Nano in Betrieb zu nehmen wird das vorher trainierte Netz auf den Speicher des Jetson Nano übertragen. Im Anschluss wird das *segnet.py* [8] Skript mit den folgenden Parametern verwendet:

```
python segnet.py
--model=Resnet18-custom-pretrained/fcn_resnet18.onnx
--input_blob=input_0
--output_blob=output_0
--labels=Resnet18-custom-pretrained/classes.txt
--colors=Resnet18-custom-pretrained/colors.txt
--filter-mode=linear
    input_video.mp4
    output_video.mp4
--input-codec=h264
```

Auflistung 4.10: Parameter für die Segmentation eines Videos

```
python segnet.py
--model=Resnet18-custom-pretrained/fcn_resnet18.onnx
--input_blob=input_0
--output_blob=output_0
--labels=Resnet18-custom-pretrained/classes.txt
--colors=Resnet18-custom-pretrained/colors.txt
/dev/video0
```

Auflistung 4.11: Parameter für die Segmentation über eine Live-Webcam

Statt */dev/video0* welches für eine am Jetson Nano angeschlossene Webcam spricht kann auch ein RTSP-Stream (Real-Time-Streaming-Protocol) mit den folgenden Parameter verwendet werden.

```
python segnet.py
--model=Resnet18-custom-pretrained/fcn_resnet18.onnx
--input_blob=input_0
--output_blob=output_0
--labels=Resnet18-custom-pretrained/classes.txt
--colors=Resnet18-custom-pretrained/colors.txt
rtsp://<remote-ip>:<port>
```

Auflistung 4.12: Parameter für die Segmentation über einen RTSP-Stream

4.5.1 Semantische Segmentierung einiger Bilder

Mit dem Ergebnis aus Kapitel 4.3 werden einige Beispielbilder aus dem Cityscapes-Datensatz ausgesucht und durch beide Netze geschickt.

Aufnahmefahrzeug	Boden	Straße	Gehweg	Person
Wand	Zaun	Stange	Verkehrsampel	Gebäude
Vegetation	Gelände	Himmel	Verkehrszeichen	Motorrad
LKW	Bus	Zug	Fahrrad	PKW

Tabelle 4.7: Farbkodierung für die Ausgabe der semantischen Segmentierung

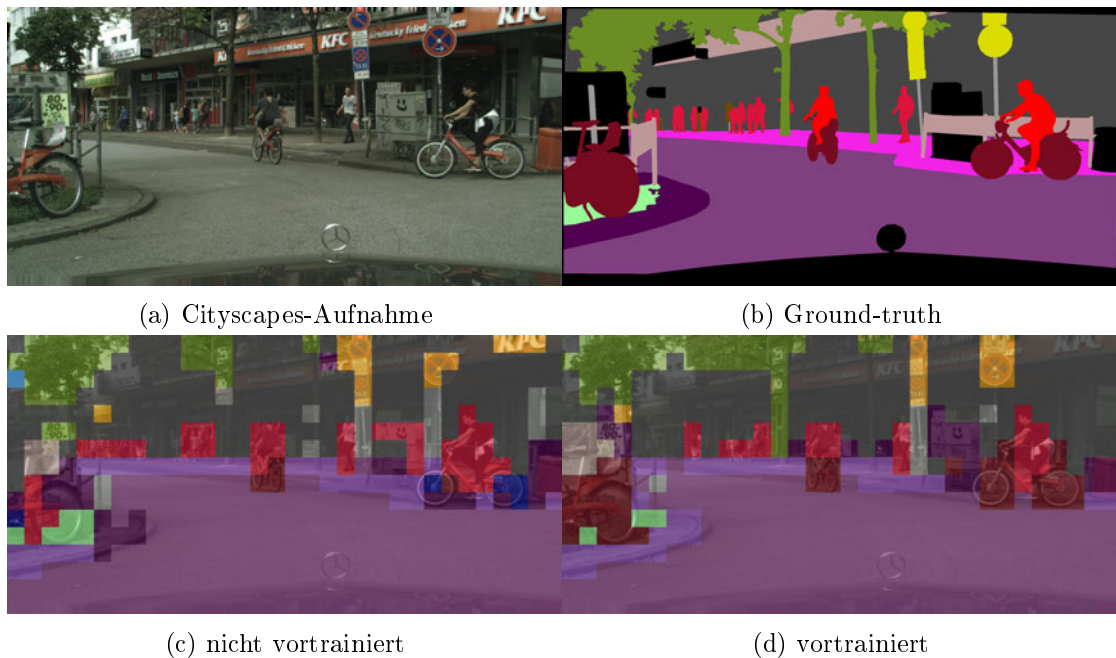


Abbildung 4.6: Semantische Segmentierung auf dem Cityscapes-Datensatz mit den modifizierten Netzen. Parameter: `-filter-mode=point`

Obwohl das vortrainierte Netz eine 10% höhere Erkennungsrate als das nicht vortrainierte Netz aufweist kann man bei der optischen Beobachtung der Ergebnisse nur wenige Unterschiede erkennen. In Abbildung 4.6 (c) wurde das Fahrrad auf der rechten Hälfte des Bildes irrtümlich teilweise als ein PKW segmentiert, welches bei Abbildung 4.6 (d) vollständig als Fahrrad erkannt wurde.

Da es sich hierbei um einzelne Momentaufnahmen handelt und in der Praxis hauptsächlich Videoaufnahmen in Echtzeit verarbeitet werden sollen, wurde zusätzlich ein Videoausschnitt vom Jungfernstieg in Hamburg ausgewählt. Die unterschiedlichen Erkennungsraten sind bei der Beobachtung des Videos, aus subjektiver Sicht, deutlich erkennbarer. In den Abbildungen 4.7 und 4.8 sind in den nicht vortrainierten Bildern, Bereiche im Bild in rot(Passanten) markiert obwohl sich dort keine Passanten befinden. Zudem wurden Schaufenster, Fahrräder und Gebäude fälschlicherweise als Passanten erkannt. Auffällig ist auch, dass in Abbildung 4.8 (a) der Gehweg als Fahrbahn klassifiziert wurde und in Abbildung 4.8 (b) dieser als Boden erkannt wurde. Dies ist zurückzuführen auf die Tatsache dass, das nicht vortrainierte Netz nur die Datensätze aus dem Cityscapes-Datensatz kennt. Die Cityscapes-Datensätze enthalten grundsätzlich Aufnahmen indem sich die Fahrbahn im unteren Bereich der Bilder befindet und die Gehwege auf der linken oder rechten Seite des Bildes. Die vortrainierten Bilder in den Abbildungen 4.7 und 4.8 haben die Passanten und Fahrzeuge relativ gut zuordnen können. Dies spricht für den Einsatz von vortrainierten Netzen, da diese Objekte aus dem Alltag wie Passanten, Fahrräder, Fahrzeuge und Gebäude bereits relativ gut Erkennen und Segmentieren können.

Die berechnete Segmentierungsmaske hat eine deutlich geringere Auflösung als das Eingangsbild. Die Auflösung wird um das 32-Fache reduziert um die Berechnung zu beschleunigen und Ausgaben in Echtzeit zu erhalten [15]. Die blockartige Segmentation in Abbildung 4.6 ist auf Grund des Parameters *filter – mode = point* entstanden. Mit diesem Parameter werden die segmentierten Felder mit eindeutigen Grenzen visualisiert. Setzt man den Parameter auf *filter – mode = linear*, wie in Abbildung 4.7 und 4.8, so wird eine bilineare Filterung angewendet um einen weicheren Farbverlauf zwischen den segmentierten Klassen zu erzeugen. Bei der Segmentation einzelner Bilder ist der Parameter auf *point* zu empfehlen, um einzelne Klassen besser zu indentifizieren. Bei Videoaufnahmen dagegenen ist das Erkennen von Bewegungen subjektiv angenehmer zu beobachten, wenn der Parameter auf *linear* gesetzt ist.



Abbildung 4.7: Auffällige Bilder bei der semantischen Segmentierung an einem Video mit den modifizierten Netzen. Parameter: `-filter-mode=linear`



Abbildung 4.8: Auffällige Bilder bei der semantischen Segmentierung an einem Video mit den modifizierten Netzen. Parameter: `-filter-mode=linear`

4.5.2 FPS Aufzeichnung

Mit der Aufzeichnung der FPS (Frame-Per-Second) des Jetson Nano, während es ein Video segmentiert, können Rückschlüsse auf die Belastung der Hardware gezogen werden. Wie in Abbildung 4.9 zu beobachten ist hat das ursprüngliche ResNet18 eine etwas höhere FPS als das vResNet18 (+mod. fc + feinab.). Auch wenn dieser Unterschied sehr gering ist, ist dieser dennoch konstant und eindeutig nachweisbar. Grund hierfür ist zu vermuten, dass das Hinzufügen weiterer Layer im modifiziertem Netz negative Auswirkungen auf die FPS des Jetson Nano hat.

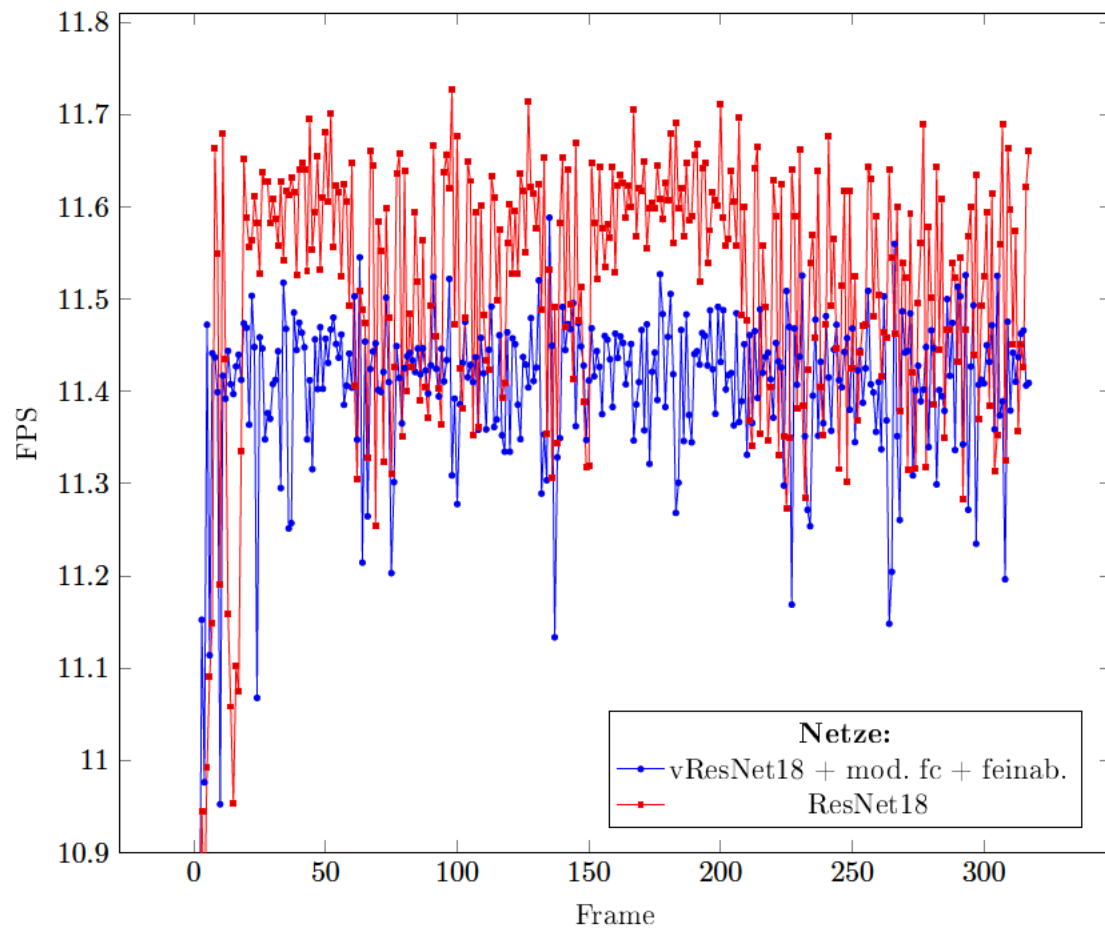


Abbildung 4.9: FPS Aufzeichnung der Netze im Einsatz auf dem Jetson Nano

5 Fazit

Das Ziel dieser Arbeit bestand darin, eine künstliche Intelligenz in Form eines neuronalen Netzwerks zu untersuchen und auf dem NVIDIA Jetson Nano zu betreiben, damit ein autonomer Roboter die Umwelt klassifizieren kann und somit eine Basis für die Gefahrenerkennung eines potenziellen Roboter-Blindenführhundes bietet. Bilder und Videos aus dem deutschen Straßenverkehr dienten dabei als Trainings- und Testdatensätze. Dafür war der Cityscapes-Datensatz eine relativ gute Wahl, wobei jede Cityscapes-Aufnahme von einem Fahrzeug auf der Fahrbahn aufgenommen wurde. Dies hat den Nachteil, dass dort, wo eigentlich sich die Fahrbahn immer befindet, nicht mehr die Fahrbahn zu finden ist, sondern der Gehweg.

Eine hohe Erkennungsrate war die ausschlaggebende Metrik, um die unterschiedlichen Variationen und Modifikationen der neuronalen Netze miteinander zu vergleichen. Die von den NVIDIA-Entwicklern erreichte Erkennungsrate von 87,3% [8] bei der pixelweisen Genauigkeit, mit einer Auflösung von 1024 x 512 Pixel, welches eine mIoU von etwa 53,5% entspricht, war das zu erreichende oder zu übertreffende Ziel. Verschiedene Kombinationen und Variationen wurden beim Trainieren der neuronalen Netze angewendet. Die Erkennungsraten jeder Epoche des Trainings wurde erfasst und aufgezeichnet, um im Anschluss die Netze untereinander zu vergleichen. Das Trainieren des ResNet-18 mit dem Cityscapes-Datensatz erreichte eine mIoU von 43,7%, mit Hilfe von Modifikationen am Fully-connected-classifier und der Feinabstimmungsmethode konnte ein Zuwachs von etwa 1% erzielt werden. Erst der Einsatz eines vortrainierten ResNet-18 mit dem ImageNet-Datensatz steigerte die mIoU auf 54,1%. Die Modifikation und Feinabstimmung erzielten im Anschluss erneut nur einen Zuwachs von etwa 1%. Am Ende war eine maximale mIoU von 55,3% erreicht, welches den angestrebten mIoU-Wert um 1,8% übertrifft.

Während des Betriebs des Jetson Nano konnte eine geringfügige Verschlechterung der FPS, zwischen dem unmodifiziertem und dem modifiziertem Netz, aufgezeichnet werden. Dies ist zurückzuführen auf das Hinzufügen zusätzlicher Layer im modifiziertem Netz. Auch der längere Betrieb über mehreren Minuten hinweg führte sporadisch zu Abstürzen. Zugleich war ein erhöhtes Aufheizen des passiven Kühlkörpers des Jetson Nano zu vermerken. Daher wurden während des Betriebs Testvideos mit einer Länge von einer Minute verwendet. Für zukünftige Versuche ist der Einsatz eines aktiven Lüfters zu empfehlen.

Literaturverzeichnis

Veröffentlichungen

- [12] Kaiming He u. a. “Deep Residual Learning for Image Recognition”. In: (Dez. 2015). arXiv: [1512.03385 \[cs.CV\]](#).
- [13] Kaiming He u. a. “Identity Mappings in Deep Residual Networks”. In: (März 2016). arXiv: [1603.05027 \[cs.CV\]](#).
- [26] Anish Shah u. a. “Deep Residual Networks with Exponential Linear Unit”. In: (Apr. 2016). DOI: [10.1145/2983402.2983406](#). arXiv: [1604.04112 \[cs.CV\]](#).
- [29] Karen Simonyan und Andrew Zisserman. “Very Deep Convolutional Networks for Large-Scale Image Recognition”. In: (Sep. 2014). arXiv: [1409.1556 \[cs.CV\]](#).
- [36] Sergey Zagoruyko und Nikos Komodakis. “Wide Residual Networks”. In: (Mai 2016). arXiv: [1605.07146 \[cs.CV\]](#).

Bücher

- [2] François Chollet. *Deep Learning with Python*. Manning, Nov. 2017. ISBN: 9781617294433.

Online

- [1] *Building the Project from Source*. URL: <https://github.com/dusty-nv/jetson-inference/blob/master/docs/building-repo-2.md> (besucht am 02.08.2021).
- [3] *Dataset Overview – Cityscapes Dataset*. URL: <https://www.cityscapes-dataset.com/dataset-overview/> (besucht am 02.08.2021).
- [4] *Der Blindenführhund*. URL: <https://www.easy-dogs.net/der-blindenfuehrhund/> (besucht am 03.01.2022).

- [5] *Developer Kit: Nvidia Jetson Nano ist deutlich kleiner und günstiger - Computer-Base*. URL: <https://www.computerbase.de/2019-03/nvidia-jetson-nano-developer-kit/> (besucht am 02.08.2021).
- [6] *Entwicklerkits und -module für eingebettete Systeme | NVIDIA Jetson*. URL: <https://www.nvidia.com/de-de/autonomous-machines/embedded-systems/> (besucht am 29.07.2021).
- [7] *Getting Started With Jetson Nano Developer Kit | NVIDIA Developer*. URL: <https://developer.nvidia.com/embedded/learn/get-started-jetson-nano-devkit> (besucht am 02.08.2021).
- [8] *GitHub: Jetson Inference*. URL: <https://github.com/dusty-nv/jetson-inference> (besucht am 02.08.2021).
- [9] *GitHub: pytorch-segmentation | adem404*. URL: <https://github.com/adem404/pytorch-segmentation> (besucht am 02.03.2022).
- [10] *GitHub: Training of semantic segmentation networks with PyTorch*. URL: <https://github.com/Onixaz/pytorch-segmentation> (besucht am 02.08.2021).
- [11] *Grafikkarte der GeForce GTX 16-Serie | NVIDIA*. URL: <https://www.nvidia.com/de-de/geforce/graphics-cards/gtx-1660-ti/> (besucht am 02.08.2021).
- [14] *ImageNet Dataset*. URL: <https://www.image-net.org/> (besucht am 03.01.2022).
- [15] *Issue 704 dusty-nv jetson-inference GitHub*. URL: <https://github.com/dusty-nv/jetson-inference/issues/704#issuecomment-806952597> (besucht am 17.02.2022).
- [16] *Jetson Nano Developer Kit SD Card Image*. URL: <https://developer.nvidia.com/jetson-nano-sd-card-image> (besucht am 02.08.2021).
- [17] *Jetson Nano: Künstliche Intelligenz für die Massen*. URL: <https://www.elektronikpraxis.vogel.de/jetson-nano-kuenstliche-intelligenz-fuer-die-massen-a-812943/> (besucht am 02.08.2021).
- [18] *Logitech C615 Full HD Webcam*. URL: <https://www.logitech.com/en-us/eol/c615-webcam.960-000733.html> (besucht am 02.08.2021).
- [19] *Loomo Personal Robot | Segway*. URL: <https://www.segway.com/loomo/> (besucht am 29.07.2021).

- [20] *NVIDIA Jetson Nano Developer Kit | NVIDIA Developer*. URL: <https://developer.nvidia.com/embedded/jetson-nano-developer-kit> (besucht am 29.07.2021).
- [21] *NVIDIA RTX A6000 datasheet*. URL: [https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/quadro-product-literature/proviz-print-nvidia-rtx-a6000-datasheet-us-nvidia-1454980-r9-web\(1\).pdf](https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/quadro-product-literature/proviz-print-nvidia-rtx-a6000-datasheet-us-nvidia-1454980-r9-web(1).pdf) (besucht am 02.08.2021).
- [22] *NVIDIA V100 TENSOR CORE GPU Specifications*. URL: <https://images.nvidia.com/content/technologies/volta/pdf/volta-v100-datasheet-update-us-1165301-r5.pdf> (besucht am 04.01.2022).
- [23] Adrian Rosebrock. *Intersection over Union (IoU) for object detection*. Nov. 2016. URL: <https://www.pyimagesearch.com/2016/11/07/intersection-over-union-iou-for-object-detection/> (besucht am 03.01.2022).
- [24] *Ryzen™ 5 2600 Desktop Prozessor der 2. Generation für Gamer | AMD*. URL: <https://www.amd.com/de/products/cpu/amd-ryzen-5-2600> (besucht am 02.08.2021).
- [25] *Semantic Segmentation with SegNet*. URL: <https://github.com/dusty-nv/jetson-inference/blob/master/docs/segnet-console-2.md> (besucht am 02.08.2021).
- [27] Pulkit Sharma. *Computer Vision Tutorial: A Step-by-Step Introduction to Image Segmentation Techniques (Part 1)*. Apr. 2019. URL: <https://www.analyticsvidhya.com/blog/2019/04/introduction-image-segmentation-techniques-python/> (besucht am 29.07.2021).
- [28] Vlad Shmyhlo. *Image of Ships in a Port*. URL: https://miro.medium.com/max/1400/1*DOTZV1XnEglbSZ0Yk4t1Vg.png (besucht am 03.01.2022).
- [30] *SK Hynix Announces 8 GB LPDDR4X-4266 DRAM Packages*. URL: <https://www.anandtech.com/show/11021/sk-hynix-announces-8-gb-lpddr4x4266-dram-packages> (besucht am 02.08.2021).
- [31] Nico Litzel Stefan Luber. *Was ist ONNX (Open Neural Network Exchange)?* Aug. 2019. URL: <https://www.bigdata-insider.de/was-ist-onnx-open-neural-network-exchange-a-851510/> (besucht am 03.01.2022).
- [32] Ekin Tiu. *Metrics to Evaluate your Semantic Segmentation Model*. URL: <https://towardsdatascience.com/metrics-to-evaluate-your-semantic-segmentation-model-6bcb99639aa2> (besucht am 03.01.2022).

- [33] *Transfer Learning mit ResNet*. URL: <https://www.statworx.com/de/blog/car-model-classification-i-transfer-learning-with-resnet/> (besucht am 03.01.2022).
- [34] *Was ist Transfer Learning?* URL: <https://www.industry-of-things.de/iot-basics-was-ist-transfer-learning-a-884674/> (besucht am 03.01.2022).
- [35] *WinRAR*. URL: <https://winrar.de/> (besucht am 04.01.2022).

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Gemäß der Allgemeinen Prüfungs- und Studienordnung ist zusammen mit der Abschlussarbeit eine schriftliche Erklärung abzugeben, in der der Studierende bestätigt, dass die Abschlussarbeit „— bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit [(§ 18 Abs. 1 APSO-TI-BM bzw. § 21 Abs. 1 APSO-INGI)] — ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt wurden. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich zu machen.“

Quelle: § 16 Abs. 5 APSO-TI-BM bzw. § 15 Abs. 6 APSO-INGI

Erklärung zur selbstständigen Bearbeitung der Arbeit

Hiermit versichere ich,

Name: _____

Vorname: _____

dass ich die vorliegende Bachelorarbeit – bzw. bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit – mit dem Thema:

Semantische Bildsegmentierung mit neuronalen Netzen für einen autonomen Roboter

ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort Datum 