

# **Erkennung und Auslesen von Nummernschildern aus Bilddaten mithilfe von Deep Learning und Optical Character Recognition**

Christian Peters

9. März 2021

Veranstaltung: Fallstudien II  
Dozent: Prof. Dr. Markus Pauly  
Gruppe: Anne-Sophie Bollmann, Susanne Klöcker,  
Pia von Kolken, Christian Peters

# Inhaltsverzeichnis

<b>1 Einleitung</b>	<b>1</b>
<b>2 Datenbeschreibung</b>	<b>1</b>
<b>3 Grundlagen neuronaler Netze</b>	<b>2</b>
3.1 Aufbau . . . . .	3
3.2 Schichten . . . . .	4
3.3 Aktivierungsfunktionen . . . . .	5
3.4 Training . . . . .	5
3.5 Convolutional Neural Networks . . . . .	8
<b>4 Pipeline</b>	<b>10</b>
4.1 Bildsegmentierung . . . . .	10
4.2 Texterkennung . . . . .	14
<b>5 Ergebnisse</b>	<b>16</b>
5.1 Ideen zur Verbesserung der Texterkennung . . . . .	17
<b>6 Zusammenfassung</b>	<b>20</b>
<b>Literatur</b>	<b>21</b>
<b>Abbildungsverzeichnis</b>	<b>23</b>

# 1 Einleitung

Die automatisierte Erkennung von Nummernschildern aus Bilddaten ist ein wichtiger Bestandteil vieler moderner Verkehrssysteme und kommt beispielsweise in Parkhäusern, an Mautstellen oder bei der Identifikation gestohlener Fahrzeuge zum Einsatz [15].

Das Ziel dieses Projektes ist es, einen Prototypen für ein solches Erkennungssystem zu entwickeln, welcher in der Lage sein soll, erfolgreich Nummernschilder aus Bilddaten erkennen und auszulesen zu können.

Zu diesem Zweck wird eine zweistufige Vorhersagepipeline konstruiert, die ausgehend von einer Bilddatei im ersten Schritt den Bildausschnitt bestimmt, der das Nummernschild enthält, und im zweiten Schritt anhand des zuvor ermittelten Ausschnitts die Zeichen des Nummernschildes ausliest.

Bei der Bestimmung des relevanten Bildausschnittes kommen sogenannte Convolutional Neural Networks zum Einsatz, eine spezielle Art von neuronalen Netzen, deren Grundlagen in Abschnitt 3 beschrieben werden. Zum Auslesen der Zeichen der Nummernschilder werden die Programmbibliotheken OpenCV [2] und Tesseract [16] verwendet.

Ein Überblick über das vorliegende Datenmaterial, welches bei der Erstellung der Pipeline und insbesondere zum Training der neuronalen Netze verwendet wurde, wird in Abschnitt 2 gegeben. Der gesamte Aufbau der Pipeline wird in Abschnitt 4 erläutert, im Anschluss werden die Ergebnisse in Abschnitt 5 beschrieben und die Stärken sowie die Schwächen des entwickelten Prototypen diskutiert.

Der gesamte Quellcode dieses Projekts ist Open Source und kann unter [https://github.com/cxan96/license\\_plate\\_detection](https://github.com/cxan96/license_plate_detection) abgerufen werden.

## 2 Datenbeschreibung

Der vorliegende Datensatz besteht aus insgesamt 949 Bildern von Autos mit Nummernschildern.<sup>1</sup> Die Bilder wurden in den Regionen Brasilien, Europa, Rumänien und USA aufgenommen.

Zu jedem dieser Bilder liegen sowohl Informationen zur Position des Nummernschildes innerhalb des Bildes, als auch zum Inhalt des Nummernschildes vor. Die Position des Nummernschildes ist dabei durch die Angabe von Pixel Koordinaten  $x_{\min}$ ,  $x_{\max}$ ,  $y_{\min}$ ,  $y_{\max}$  relativ zur linken oberen Ecke des Bildes eindeutig spezifiziert. In Abbildung 1 sind drei Beispielbilder aus dem Datensatz dargestellt. Die Koordinaten der Nummernschilder wurden anhand der roten Rechtecke eingezeichnet.

Zur weiteren Verarbeitung der Bilder werden diese von nun an als Matrizen aufgefasst, deren Einträge genau den Pixeln im Bild zuzuordnen sind. Da es sich hier um Farbaufnahmen handelt, benötigt man für jedes Bild sogar drei Matrizen, eine für jeden der drei Farbkanäle Rot, Grün und Blau (RGB). Beispielsweise kann ein Bild mit einer Auflösung von 100x100 Pixeln also als Element des  $\mathbb{R}^{100 \times 100 \times 3}$  beschrieben werden.

---

<sup>1</sup>Die Originaldaten sind unter [https://github.com/phibuc/Lab\\_FS\\_Data](https://github.com/phibuc/Lab_FS_Data), sowie unter <https://github.com/RobertLucian/license-plate-dataset> einsehbar.



Abbildung 1: Drei Beispiele aus dem vorliegenden Datensatz. Die Nummernschilder sind anhand ihrer Koordinaten rot umrandet.

Solche „dreidimensionalen Matrizen“ werden in der Fachsprache auch als **Tensoren** [5] bezeichnet.

### 3 Grundlagen neuronaler Netze

Neuronale Netze sind eine Modellklasse, welche zur Lösung des bereits ausführlich in [8] beschriebenen Problems des statistischen Lernens eingesetzt werden können. In dieser Problemsituation wird angenommen, dass sich der Zusammenhang zwischen beobachtbaren Prädiktorvariablen  $X_1, \dots, X_p$ , welche sich durch einen Vektor  $X = (X_1, \dots, X_p)$  zusammenfassen lassen, und einer Zielvariable  $Y$  durch eine Funktion  $f^*$  mit  $Y = f^*(X) + \epsilon$  modellieren lässt. Hierbei kann  $\epsilon$  als eine zufällige Störgröße angesehen werden, die hier im weiteren Verlauf aber keine wichtige Rolle spielt. Das Ziel von neuronalen Netzen ist es, die unbekannte Funktion  $f^*$  zu approximieren.

Damit  $f^*$  approximiert werden kann, ist es notwendig, sowohl  $X$  als auch  $Y$  zu Beobachten. Ist dies geschehen, so liegen die Beobachtungen in Form einer Menge von **Trainingsdaten**  $S$  vor, die sich wie folgt formalisieren lässt:

$$S = \{(x_1, y_1), \dots, (x_n, y_n)\}, \quad x_i \in \mathbb{R}^p, \quad y_i \in \mathbb{R}^d, \quad i = 1, \dots, n \quad (1)$$

Die  $x_i$  entsprechen hierbei den Beobachtungen des Zufallsvektors  $X$  und die  $y_i$  sind die Beobachtungen bezüglich  $Y$ . Dem aufmerksamen Leser wird aufgefallen sein, dass die  $y_i$  (und damit auch  $Y$ ) als Elemente des  $\mathbb{R}^d$ , also als  $d$ -dimensionale Vektoren modelliert wurden, was eine Abweichung zum Modell aus [8] darstellt. Dies hängt mit der vorliegenden Datensituation dieses Projektes zusammen.

Es kann an dieser Stelle nämlich schon vorweggenommen werden, dass innerhalb der  $y_i$  später die Positionen der Nummernschilder kodiert werden, die durch das neuronale Netz vorhergesagt werden sollen. Details zur Kodierung der  $y_i$  sollen aber an dieser Stelle bewusst noch nicht vertieft werden, da zunächst die wichtigsten Grundlagen neuronaler Netze entwickelt werden müssen.

Die folgenden Erklärungen zum Aufbau und zum Training neuronaler Netze stützen sich wesentlich auf [5] und sind hier auf das grundlegendste reduziert worden. Obwohl es der Name *neuronale* Netze suggeriert, wird auch hier genau wie in [5] ebenfalls auf jegliche biologische Motivation verzichtet, damit nicht der falsche Eindruck entstehen

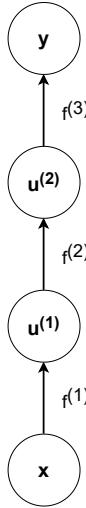


Abbildung 2: Verkettung dreier Funktionen dargestellt als gerichteter azyklischer Graph. Die Knoten  $u^{(1)}$  und  $u^{(2)}$  stellen die Zwischenergebnisse der Funktionen  $f^{(1)}$  und  $f^{(2)}$  dar, der Knoten  $y$  repräsentiert die Ausgabe des durch diesen Graphen gegebenen neuronalen Netzes.

kann, dass es sich bei neuronalen Netzen um Modelle von echten biologischen Gehirnen handelt. Das Ziel von neuronalen Netzen ist es viel eher, unbekannte Funktionen anhand von Trainingsdaten zu approximieren, um anschließend Vorhersagen auf neuen und ungesiehenen Daten anzustellen.

### 3.1 Aufbau

Wie oben in Abschnitt 3 schon angedeutet, definiert ein neuronales Netz also eine Abbildung  $f$ , welche den Zusammenhang zwischen einer Eingabe  $x \in \mathbb{R}^p$  und einer Ausgabe  $y \in \mathbb{R}^d$  approximieren soll. Eine Hauptcharakteristik von neuronalen Netzen ist es, dass die Funktion  $f$  durch die Verkettung weiterer Funktionen gebildet wird. Ist ein neuronales Netz beispielsweise durch  $f(x) = f^{(3)}(f^{(2)}(f^{(1)}(x)))$  gegeben, so setzt es sich aus der Verkettung der einzelnen Funktionen  $f^{(1)}$ ,  $f^{(2)}$  und  $f^{(3)}$  zusammen. Wie genau diese Funktionen aussehen können, soll an dieser Stelle bewusst erst einmal offen bleiben. Eine wichtige Voraussetzung an diese Funktionen ist allerdings die Differenzierbarkeit, welche später für das Training eines neuronalen Netzes eine wichtige Rolle spielt. Ansonsten ist es aber schon ausreichend, sich neuronale Netze als Verkettungen (nahezu) beliebiger differenzierbarer Funktionen vorzustellen.

Solche Ketten von Funktionen können gut durch gerichtete azyklische Graphen beschrieben werden. Hierbei wird jedes Zwischenergebnis durch einen Knoten repräsentiert, jede Kante zwischen zwei Knoten beschreibt die Operation, die von einem Ergebnis zum nächsten geführt hat. Der Beispielgraph zu  $f(x) = f^{(3)}(f^{(2)}(f^{(1)}(x)))$  ist in Abbildung 2 zu sehen.

Anhand dieses Beispielgraphen fällt auf, dass alle Pfeile in die gleiche Richtung hin zur Ausgabe zeigen. Da eine Eingabe  $x$  also bildlich gesprochen immer in eine Richtung *vorwärts* durch den Graphen fließt, spricht man hier auch von einem neuronalen

**feed-forward** Netz. Da diese Art von neuronalen Netzen in der Praxis am häufigsten vorkommt und auch in diesem Projekt verwendet wurde, beschränken sich alle folgenden Erklärungen auf feed-forward Netze.

Im Kontext von neuronalen Netzen wird jede der Funktionen  $f^{(1)}$ ,  $f^{(2)}$  und  $f^{(3)}$  auch als eine **Schicht** des Netzes bezeichnet. Da  $f^{(1)}$  und  $f^{(2)}$  im Inneren des Netzwerks liegen, bezeichnet man diese Schichten auch als **versteckte Schichten**. Die letzte Schicht eines neuronalen Netzes hingegen, also in diesem Fall die Funktion  $f^{(3)}$ , wird als die **Ausgabeschicht** des Netzwerks bezeichnet. Der Wert dieser Schicht ist gleichzeitig auch der Ausgabewert des gesamten Netzes für eine Eingabe  $x$ . Die Gesamtanzahl der Schichten eines neuronalen Netzes wird auch als die **Tiefe** des Netzes bezeichnet. Wenn in der Praxis also von **Deep Learning** gesprochen wird, geht es lediglich um tiefe neuronale Netze mit vielen Schichten, also vielen verketteten Funktionen. Wie genau die Schichten eines neuronalen Netzwerkes nun aussehen können, soll im nächsten Abschnitt etwas detaillierter beschrieben werden.

## 3.2 Schichten

Eine Schicht eines neuronalen Netzes ist eine Funktion  $f$ , die eine Eingabe  $x \in \mathbb{R}^n$  auf eine Ausgabe  $f(x) \in \mathbb{R}^m$  abbildet. Häufig hat  $f$  dabei die folgende Form:

$$f(x) = g(Wx + b) \quad (2)$$

$W \in \mathbb{R}^{m \times n}$  ist hierbei eine sogenannte Gewichtsmatrix, die die Eingabe  $x$  der Schicht linear transformiert. Ist  $W$  dicht besetzt, so spricht man auch von einer dichten (eng. **dense**) Schicht.  $b \in \mathbb{R}^m$  wird auch Bias genannt und wird auf das Ergebnis der Multiplikation von  $W$  mit  $x$  addiert. Die Funktion  $g : \mathbb{R}^m \rightarrow \mathbb{R}^m$  heißt Aktivierungsfunktion.

Oftmals ist es so, dass Aktivierungsfunktionen elementweise auf das Resultat von  $Wx + b$  angewendet werden. Hierzu kann man sich eine Funktion  $\Phi : \mathbb{R} \rightarrow \mathbb{R}$  definieren und dann  $g_i(u) = \Phi(u_i)$ ,  $i = 1, \dots, m$  setzen. Hierbei beschreibt  $g_i$  das  $i$ -te Element der vektorwertigen Funktion  $g$  und  $u_i$  das  $i$ -te Element des Eingabevektors  $u$  der Aktivierungsfunktion, also in diesem Fall  $u = Wx + b$ .

Für das später in Abschnitt 3.4 beschriebene Training neuronaler Netze ist es wichtig, dass  $f$  eine differenzierbare Funktion ist. Damit dies gegeben ist, muss in diesem Fall also auch die Aktivierungsfunktion  $g$ , beziehungsweise die elementweise angewendete Funktion  $\Phi$  differenzierbar sein.

Die Parameter  $W$  und  $b$  einer Schicht können während der Trainingsphase des neuronalen Netzwerks optimiert und an die Trainingsdaten angepasst werden. Die Aktivierungsfunktion einer Schicht hingegen ist ein sogenannter **Hyperparameter** des Netzwerks. Dies bedeutet, dass dieser Parameter vor dem Training vom Anwender spezifiziert werden muss. Der Parameter  $m$ , also die Ausgabedimension der Schicht, ist ebenfalls ein Hyperparameter.

### 3.3 Aktivierungsfunktionen

Im Folgenden werden ausschließlich elementweise Aktivierungsfunktionen  $\Phi : \mathbb{R} \rightarrow \mathbb{R}$  betrachtet, da diese die größte praktische Relevanz haben. Zwei sehr häufig eingesetzte Aktivierungsfunktionen, die auch in diesem Projekt verwendet wurden, sind die **Rectified Linear Unit (ReLU)** Funktion und die **Sigmoid** Funktion. Beide werden im Folgenden kurz beschrieben.

**ReLU** Die ReLU Funktion ist nach [4] quasi eine Standardempfehlung für die Aktivierungsfunktion in den versteckten Schichten tiefer neuronaler Netze. Sie ist durch folgenden Ausdruck gegeben:

$$\Phi_{\text{ReLU}}(x) = \max\{0, x\}$$

Wird diese Funktion elementweise auf einen Vektor angewendet, so werden alle negativen Elemente auf Null gesetzt. Die restlichen Elemente bleiben unberührt. Dieses Vorgehen hat in der Praxis viele Vorteile [4], unter anderem erreicht man durch das Setzen negativer Eingaben auf 0, dass Zwischenergebnisse im Netzwerk dünn besetzt sind, was das Training der neuronalen Netze erleichtern kann.

Es fällt sofort auf, dass die ReLU Funktion im Punkt  $x = 0$  nicht differenzierbar ist, was ja eigentlich für das Training neuronaler Netze unerwünscht ist. Dies stellt dank des Konzeptes von Subgradienten [14] (was hier bewusst nicht weiter vertieft werden soll) aber in der Praxis kein Problem dar, da man den Wert der Ableitung der ReLU Funktion an der Stelle  $x = 0$  in praktischen Implementierungen auf einen konstanten Wert zwischen 0 und 1 setzen kann.

**Sigmoid** Die Sigmoid Funktion ist durch folgenden Ausdruck gegeben:

$$\Phi_{\text{Sigmoid}}(x) = \frac{1}{1 + \exp(-x)}$$

Da der Wertebereich der Sigmoid Funktion auf das Intervall  $(0, 1)$  beschränkt ist, kommt diese Funktion häufig in der Ausgabeschicht von neuronalen Netzen zum Einsatz. Beispielsweise bietet es sich im Szenario einer binären Klassifikation, also  $y \in \{0, 1\}$ , häufig an, die Sigmoidfunktion in der Ausgabeschicht zu verwenden. Ähnlich zur logistischen Regression [8] wird dann ein Schwellenwert definiert, beispielsweise 0,5, anhand dessen die Ausgabe des Netzes dann auf 1 oder 0 gesetzt wird, je nachdem ob der Schwellenwert überschritten wurde.

### 3.4 Training

Möchte man ein neuronales Netz zur Approximation einer unbekannten Funktion  $f^*$  einsetzen, so muss man sich zunächst für eine grundlegende Architektur des Netzes entscheiden. Dies bedeutet, dass man festlegen muss, welche Art von Schichten wie miteinander

kombiniert werden sollen. Je nach Anwendungsfall können hier unterschiedliche Architekturen sinnvoll sein und es müssen oft mehrere Varianten getestet und gegeneinander abgewogen werden.

Gehen wir aber nun einmal davon aus, dass wir uns für eine konkrete Architektur entschieden haben (die tatsächliche Architektur, die in diesem Projekt zum Einsatz gekommen ist, wird später beschrieben). Die Frage ist nun, wie die freien Parameter, also die Gewichtsmatrizen  $W$ , sowie die Bias Vektoren  $b$ , in jeder Schicht angepasst werden können, damit das neuronale Netz die Funktion  $f^*$  möglichst gut approximiert.

Zur Beantwortung dieser Frage ist es hilfreich, sich in Erinnerung zu rufen, dass ein neuronales Netz lediglich durch eine Funktion  $f(x)$  beschrieben werden kann. Sämtliche Parameter dieser Funktion (also die Gewichtsmatrizen und die Bias-Vektoren) lassen sich o.B.d.A. zu einem einzigen Parametervektor  $\theta$  zusammenfassen, welcher die Funktion  $f$  parametrisiert (wir sprechen deshalb im Folgenden von  $f_\theta$ ).

Der mittlere quadratische Approximationsfehler unseres neuronalen Netzes  $f_\theta$  für einen Parametervektor  $\theta$  auf den bereits in Gleichung 1 beschriebenen Trainingsdaten  $S$  lässt sich dann wie folgt berechnen:

$$L(\theta) = \frac{1}{n} \sum_{i=1}^n \|y_i - f_\theta(x_i)\|_2^2 \quad (3)$$

Das Training eines neuronalen Netzes  $f_\theta$  besteht nun also darin, einen Parametervektor  $\theta$  zu finden, für den die sogenannte Verlustfunktion  $L(\theta)$  minimiert wird. Es liegt also ein nicht-lineares Optimierungsproblem einer geschlossenen und differenzierbaren Funktion  $L(\theta)$  vor.

Anstelle des mittleren quadratischen Fehlers können in der Praxis je nach Szenario auch andere Verlustfunktionen eingesetzt werden. Liegt beispielsweise ein binäres Klassifikationsproblem vor, so wird häufig in Kombination mit einer Sigmoid-Ausgabeschicht die logistische Verlustfunktion [5] verwendet:

$$L_{\text{logistisch}}(\theta) = -\frac{1}{n} \sum_{i=1}^n y_i \cdot \log(f_\theta(x_i)) + (1 - y_i) \cdot \log(1 - f_\theta(x_i)) \quad (4)$$

Auch hier muss ein passender Parametervektor  $\theta$  gefunden werden, welcher die Verlustfunktion minimiert.

Zur Lösung eines solchen Problems gibt es viele verschiedene Verfahren. In der Praxis kommen meist Varianten des stochastischen Gradientenabstiegs zum Einsatz. Dieses allgemeine Optimierungsverfahren ist in [14] im Bezug auf allgemeine Machine Learning Probleme ausführlich beschrieben und analysiert worden und soll hier nur einmal bezüglich des Trainings neuronaler Netze kurz umrissen werden.

**Stochastischer Gradientenabstieg** Stochastischer Gradientenabstieg ist eine Variante des allgemeinen Gradientenabstiegs, der nun zuerst beschrieben wird.

Möchte man die Funktion  $L(\theta)$  durch Gradientenabstieg minimieren, so muss man zunächst einen Startwert  $\theta^{(0)}$  auswählen, an welchem die Optimierung beginnen soll.

Hierbei gibt es viele mögliche Vorgehensweisen. Im Kontext tiefer neuronaler Netze hat sich beispielsweise die Glorot Initialisierung bewährt, welche erstmals in [3] vorgestellt wurde und auf eine zufällige, aber möglichst gleichmäßige Initialisierung der Parameter abzielt.

Hat man einen Startwert  $\theta^{(0)}$  gewählt, so wird dieser nun durch Gradientenabstieg schrittweise verbessert. Dies geschieht dadurch, dass sukzessive Schritte in die Richtung des negativen Gradienten von  $L(\theta)$  durchgeführt werden. Im Optimierungsschritt  $t$  wird der aktuelle Parametervektor  $\theta^{(t)}$  also wie folgt aktualisiert:

$$\theta^{(t+1)} = \theta^{(t)} - \eta \nabla L(\theta^{(t)}) \quad (5)$$

Hierbei gibt  $\eta$  eine Schrittweite an, auch **Lernrate** genannt, die zuvor vom Anwender spezifiziert werden muss.  $\nabla L(\theta^{(t)})$  ist der Gradient der Verlustfunktion  $L$  bezüglich der trainierbaren Parameter  $\theta$ , ausgewertet an der Stelle  $\theta^{(t)}$ .

Die Idee bei dieser Vorgehensweise ist, dass der negative Gradient immer in die Richtung des steilsten Abstiegs einer Funktion zeigt. Man erhofft sich durch diese lokale Minimierung der Funktion, dass man schlussendlich in einem globalen Minimum auskommt. Dies ist aber nur dann mathematisch garantiert, wenn die Funktion  $L(\theta)$  konvex ist (in diesem Fall gibt es nur ein einziges globales Minimum). Da es sich hier allerdings um ein hochgradig nicht-lineares Optimierungsproblem handelt, ist die Konvexität von  $L$  für tiefe neuronale Netze in den meisten Fällen nicht gegeben. Man kann also bestenfalls auf die Konvergenz des Verfahrens in ein lokales Minimum hoffen. Laut [6] scheint es aber in der Praxis Anhaltspunkte dafür zu geben, dass lokale Minima, anders als man denken könnte, oft kein großes Problem beim Training darstellen.

Der Unterschied zwischen stochastischem Gradientenabstieg und allgemeinem Gradientenabstieg ist, dass beim stochastischen Gradientenabstieg der Gradient der Verlustfunktion nur teilweise berechnet wird. Dies bedeutet, dass man immer nur einen Teil der Trainingsbeispiele für einen Gradientenschritt verwendet und nicht den ganzen Datensatz. Beim klassischen stochastischen Gradientenabstieg wird für jedes Update lediglich ein einziges Trainingsbeispiel verwendet. Es ist aber auch möglich, sogenannte **Batches** von Trainingsbeispielen für einen Gradientenschritt zu verwenden. Die Größe dieser Batches lässt sich dann passend zum vorliegenden Problem variieren.

**Berechnung der Gradienten** Eine weitere offene Frage ist, wie der Gradient von  $L$  berechnet werden kann. Hierbei hilft es, sich erneut in Erinnerung zu rufen, dass es sich bei  $L$  und auch bei  $f_\theta$  um geschlossene Funktionen handelt, deren Gradienten explizit angegeben werden können. Dennoch bleibt die Frage, wie dies algorithmisch effizient umgesetzt werden kann.

In der Praxis kommt hier meist der sogenannte Backpropagation-Algorithmus [13] zum Einsatz. Eine detaillierte Beschreibung dieses Algorithmus würde an dieser Stelle zu weit führen, die Grundidee jedoch lässt sich in wenigen Sätzen beschreiben.

Wie bereits in Abschnitt 3.1 und insbesondere anhand von Abbildung 2 gezeigt, lassen sich neuronale Netze als Verkettungen von Funktionen beschreiben, die durch azyklische Graphen dargestellt werden können. Die Verlustfunktion, deren Gradient berechnet werden soll, lässt sich ebenfalls durch einen solchen Graphen beschreiben. Der

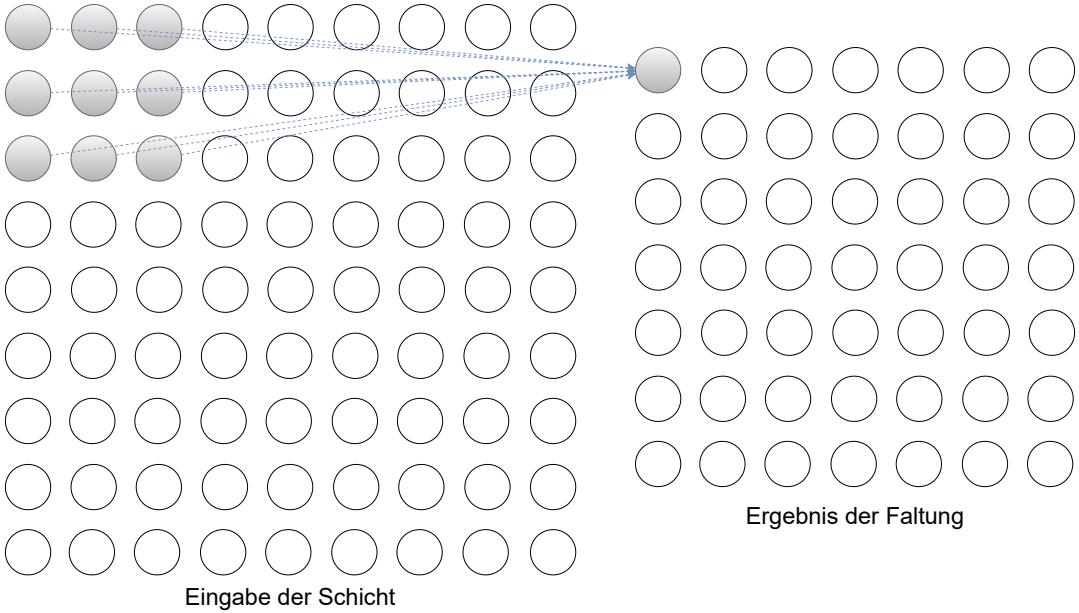


Abbildung 3: Schematische Darstellung einer Faltungsschicht

Backpropagation-Algorithmus ist ein allgemeines Verfahren, welches den Gradienten von Funktionen, die durch azyklische Graphen beschrieben werden können, durch Anwendung der **Kettenregel** berechnen kann.

Die Details werden hier bewusst ausgespart, es soll allerdings betont werden, dass es sich beim Backpropagation-Algorithmus lediglich um eine effiziente Berechnungsvorschrift der Kettenregel handelt. Bekannte Programmiersysteme wie Tensorflow [1] implementieren dieses Verfahren durch effiziente Ausnutzung vorhandener Hardware (insbesondere GPUs) und erfreuen sich daher großer Beliebtheit.

### 3.5 Convolutional Neural Networks

Convolutional Neural Networks (im folgenden **CNN**) sind eine spezielle Art von neuronalen feed-forward Netzen, die sich besonders für den Einsatz auf Bilddaten eignen [5]. Der Name röhrt daher, dass CNNs in mindestens einer ihrer Schichten eine sogenannte Faltungsoperation (eng. Convolution) verwenden.

Eine detaillierte und formale Beschreibung der Faltungsoperation würde hier an dieser Stelle zu weit führen. Daher wird im folgenden die Funktionsweise einer sogenannten **Faltungsschicht** eines CNNs lediglich kurz intuitiv umrissen. Für mehr Details wird auf Fachliteratur wie [5] verwiesen.

**Faltungsschichten** Eine schematische Darstellung einer Faltungsschicht ist in Abbildung 3 gegeben. Der erste Unterschied zur bereits in 3.2 beschriebenen dichten Schicht ist, dass die Eingabe eine zweidimensionale Struktur aufweist, wie es beispielsweise bei (schwarz-weißen) Bilddaten der Fall ist. Zur formalen Beschreibung solcher Daten können dann anstelle von Vektoren Matrizen zum Einsatz kommen. In Abbildung 3 würde

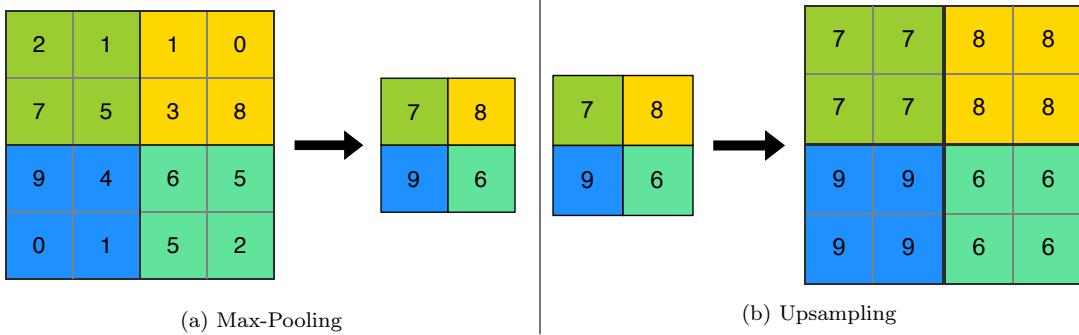


Abbildung 4: Die Pooling Schicht (links) reduziert die Dimension ihrer Eingabe, die Upsampling Schicht (rechts) bewirkt das Gegenteil.

dann jeder Punkt im linken Gitter einem Eintrag aus der Eingabematrix der Schicht entsprechen.

In der Praxis wird dieses Konzept häufig auf mehr als zwei Dimensionen verallgemeinert. Dies ist insbesondere dann nötig, wenn man es beispielsweise mit Farbbildern zu tun hat, die für jeden Pixel drei Farbwerte speichern. In diesem Fall spricht man dann nicht mehr von Matrizen, sondern von **Tensoren**.

Die Faltungsoperation auf einer solchen Eingabe kann man sich nun so vorstellen, dass eine Art Fenster (auch **Kernel** genannt) Schritt für Schritt über die Eingabe fährt und dabei eine gewichtete Summe berechnet. In Abbildung 3 hat dieses Fenster die Größe 3x3, die gewichtete Summe wird durch die blauen Pfeile symbolisiert. Analog zur dichten Schicht wird zudem auf jedes Ergebnis der gewichteten Summe ein Bias aufaddiert. Im Anschluss wird auch bei Faltungsschichten auf die Ausgabe eine (meist elementweise) Aktivierungsfunktion angewendet.

Die Gewichte des Kernels zusammen mit dem Bias und der Aktivierungsfunktion sind die trainierbaren Parameter einer Faltungsschicht. Die Kernel-Größe, sowie die Schrittwerte, mit welcher der Kernel über die Eingabe fährt, sind Hyperparameter, die vom Anwender vor dem Training spezifiziert werden müssen. Es ist darüber hinaus bei Faltungsschichten auch üblich, gleich mehrere Kernel gleichzeitig einzusetzen. Auf diese Weise erhält man dann auch mehrere Ausgabematrizen, die dann übereinander geschichtet werden können (man erhält also einen **Ausgabetensor**). Die Anzahl der Kernel einer Faltungsschicht ist ebenfalls ein Hyperparameter.

**Pooling und Upsampling Schichten** Zwei weitere Arten von Schichten, die oft in CNNs zum Einsatz kommen und auch in diesem Projekt angewendet wurden, sind Pooling und Upsampling Schichten. Beide sind in Abbildung 4 schematisch dargestellt.

Die Aufgabe von Pooling Schichten innerhalb eines CNN ist es, die Dimensionalität ihrer Eingabe zu verringern. Pooling Schichten verfügen über keine trainierbaren Parameter. Ähnlich zu Faltungsschichten basieren auch Pooling Schichten auf einem Kernel, der über die Eingabe fährt. Der Unterschied ist aber, dass der Kernel keine gewichtete Summe berechnet, sondern lediglich das Maximum zurückgibt.<sup>2</sup> Außerdem ist es im

---

<sup>2</sup>In diesem Fall spricht man auch von Max-Pooling.

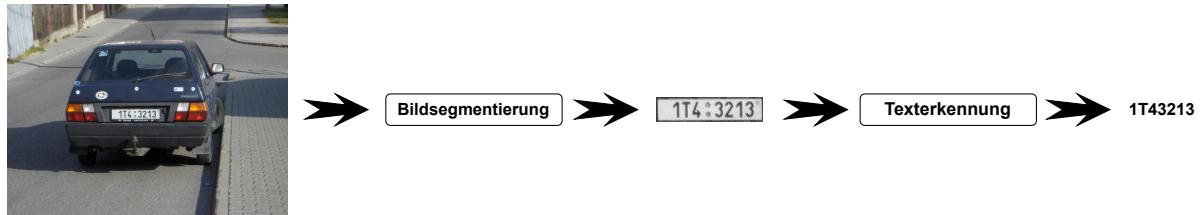


Abbildung 5: Schematische Darstellung der zweistufigen Vorhersagepipeline. Zunächst wird das Nummernschild mithilfe von Bildsegmentierung extrahiert. Anschließend werden die Zeichen durch Texterkennung ausgelesen.

Falle von Pooling üblich, dass der Kernel größere Schritte zurücklegt. Im Beispiel von Abbildung 4 ist die Kernel-Größe  $2 \times 2$  und die Schrittweite 2. So wird die Eingabe von  $4 \times 4$  auf  $2 \times 2$  reduziert, die Dimensionalität wird also halbiert.

Upsampling Schichten sind in gewisser Weise die Gegenspieler von Pooling Schichten, da sie die Dimensionalität ihrer Eingabe künstlich vergrößern, indem sie jeweils Zeilen und Spalten vervielfachen (in Abbildung 4 werden sowohl Zeilen als auch Spalten verdoppelt). Auch Upsampling Schichten haben keine trainierbaren Parameter.

## 4 Pipeline

Eine schematische Übersicht der in diesem Projekt eingesetzten zweistufigen Vorhersagepipeline ist in Abbildung 5 dargestellt.

Die Pipeline ist so aufgebaut, dass auf eine Bildeingabe zunächst eine Bildsegmentierung angewendet wird, um den Bereich des Bildes, welcher das Nummernschild enthält, herauszutrennen. Für die Bildsegmentierung wurde ein CNN trainiert. Die Einzelheiten hierzu werden in Abschnitt 4.1 näher beschrieben.

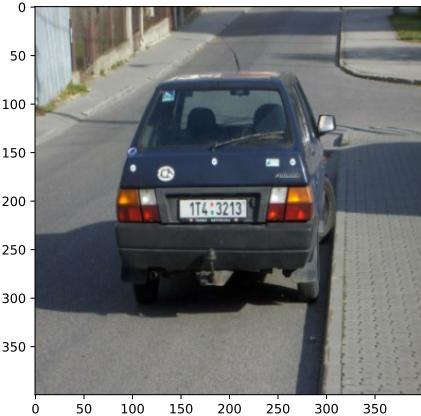
Nach der Bildsegmentierung wird auf den herausgetrennten Bereich eine Texterkennung angewendet, um die Zeichenfolge des Nummernschildes zu extrahieren. Dieses Verfahren wird in Abschnitt 4.2 beschrieben.

### 4.1 Bildsegmentierung

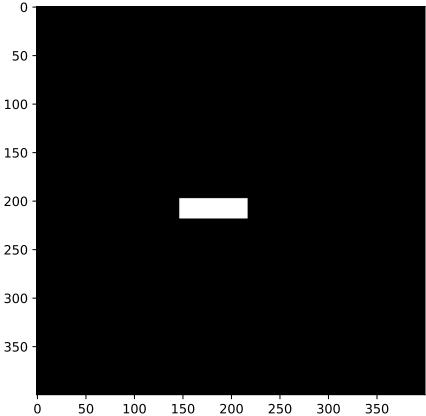
Das Ziel einer Bildsegmentierung ist es, die relevanten Bereiche eines Bildes von den restlichen zu trennen. In dieser Situation ist es also gefordert, den Bereich des Bildes, welcher das Nummernschild enthält, vom Rest des Bildes zu trennen.

Zu diesem Zweck kommt in diesem Projekt eine binäre Klassifikation zum Einsatz: Für jeden Pixel des Eingabebildes wird klassifiziert, ob er Teil des Nummernschildes ist, oder nicht. Das Ziel ist es also, für ein Eingabebild eine binäre Maske vorherzusagen, welche für jeden Pixel eine 1 enthält, falls er Teil des Nummernschildes ist und anderenfalls eine 0. Eine visuelle Darstellung dieses Konzeptes ist in Abbildung 6 gegeben.

Als Klassifikationsverfahren kommen zu diesem Zweck CNNs zum Einsatz, die bereits in Abschnitt 3.5 beschrieben wurden. Auf die genaue Architektur und das Training des Netzes soll nun genauer eingegangen werden. Der Ansatz, CNNs zur Bildsegmentierung



(a) Eingabebild



(b) Binäre Maske des Nummernschildes.

Abbildung 6: Die binäre Maske (rechts) ordnet jedem Pixel der Eingabe (links) einen Wert 1 (hier weiß dargestellt) zu, falls sich dieser im Nummernschild befindet. Andernfalls wird ein Wert von 0 (schwarz) zugeordnet.

zu verwenden, geht auf [11] zurück und wurde hier für die vorliegende Problemstellung adaptiert.

#### 4.1.1 Netzarchitektur

Wie schon in Abschnitt 3 erläutert, ordnen neuronale Netze einer Eingabe fester Größe eine zugehörige Ausgabe fester Größe zu. In diesem Fall soll die Ausgabe des Netzes zu einem Eingabebild genau die binäre Maske sein, welche das Nummernschild repräsentiert. Da die Bilder im Datensatz allerdings verschiedene Auflösungen und damit auch verschiedene Größen aufweisen, neuronale Netze aber auf eine feste einheitliche Größe angewiesen sind, müssen die Bilder vor der Eingabe in das neuronale Netz zunächst auf eine einheitliche Größe gebracht werden. In diesem Projekt wird zu diesem Zweck jedes Bild vor der Eingabe in das Netz auf eine Größe von 400x400 skaliert.

Die komplette Netzarchitektur, die in diesem Projekt zum Einsatz kam, ist in Abbildung 7 schematisch dargestellt. Genaue Details zu jeder Schicht, sowie die Anzahl der trainierbaren Parameter, können Tabelle 1 entnommen werden.

Bei der groben Struktur der Netzarchitektur wurde sich hierbei an [11] orientiert. Beispielsweise ist es zur Reduktion der trainierbaren Parameter sinnvoll, die Dimension der Faltungsschichten zuerst schrittweise durch Pooling-Schichten zu verkleinern und erst am Ende wieder durch Upsampling-Schichten zu vergrößern. Wie man in Abbildung 7 sehen kann, verlaufen die Ausgabedimensionen der Schichten also frei gesprochen von „breit und flach“ zu „schmal und tief“ im Inneren des Netzes und dann wieder zu „breit und flach“ hin zur Ausgabe der Vorhersage für die binäre Maske.

Während sich diese allgemeine Struktur zwar generell zur Bildsegmentierung etabliert hat, so wurde diese spezielle Architektur jedoch erst nach vielen Testläufen und Abwägungen gefunden. Des Weiteren scheint die Gesamtanzahl der trainierbaren Parameter von 191.297 zwar auf den ersten Blick sehr groß zu sein, allerdings darf man nicht ver-

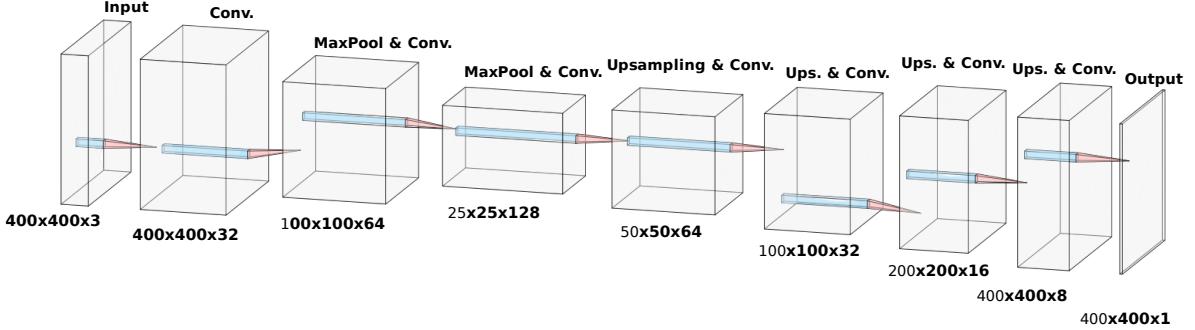


Abbildung 7: Schematische Darstellung der gewählten Netzarchitektur zur Bildsegmentierung. Die Eingabe des Netzes ist ein Tensor der Größe  $400 \times 400 \times 3$ , der das Eingabebild repräsentiert. Die Ausgabe ist eine Vorhersage der binären Maske und hat daher die Größe  $400 \times 400 \times 1$ , die Kernel der Faltungsschichten (Conv.) sind in Blau eingezzeichnet.

gessen, dass ein einziges Bild schon aus  $400 \cdot 400 \cdot 3 = 480.000$  Farbwerten besteht. Die Anzahl der Parameter dieser Architektur ist also wesentlich kleiner, als die Dimension der Modelleingabe. Auch wenn die Netzarchitektur also als recht komplex anmutet, so ist sie was die Zahl der freien Parameter betrifft, sogar weniger komplex als ein lineares Modell.

Abschließend sei noch ein kleines, aber spannendes Detail zur Ausgabeschicht erwähnt. Die Ausgabeschicht ist eine Faltungsschicht mit einem Kernel der Größe  $1 \times 1$ , welche die Sigmoidfunktion zur Bestimmung der Netzausgabe verwendet. Sie enthält also 9 trainierbare Parameter (8 gehen auf den Kernel zurück und 1 Parameter ist der Bias). Sieht man genauer hin, so entspricht dies genau dem Modell der logistischen Regression, welches als Features genau die Ausgabe der vorherigen Faltungsschicht verwendet. Im Grunde ist diese Netzarchitektur also eine logistische Regression, die gleichzeitig mit einer eingebauten Feature-Extraktion trainiert wird.

#### 4.1.2 Training

Die vorgestellte Netzarchitektur wurde mithilfe der Programmiersbibliothek Tensorflow [1] implementiert, das Training wurde auf einer GPU von Google Colab<sup>3</sup> in der Cloud durchgeführt. Als Optimierungsverfahren kam ADAM [9], eine Variante des stochastischen Gradientenabstiegs zum Einsatz. Die Initialisierung der Parameter wurde mit der schon in Abschnitt 3.4 kurz beschriebenen Glorot-Initialisierung durchgeführt. Als Verlustfunktion wurde hier analog zu [11] aufgrund des Szenarios der binären Klassifikation für jeden Pixel ebenfalls die logistische Verlustfunktion eingesetzt.

Vor dem Training wurde der Datensatz aufgrund der vergleichsweise geringen Anzahl an Trainingsbildern durch sogenanntes Data Augmentation vergrößert. Hierbei wurden verschiedene Eigenschaften der Trainingsbilder zufällig verändert:

- Die Bilder wurden horizontal gespiegelt

<sup>3</sup><https://colab.research.google.com/>

Tabelle 1: Die eingesetzte Netzarchitektur. Insgesamt enthält sie 191.297 trainierbare Parameter.

Schicht	Ausgabegröße	Anzahl Parameter
Eingabe	400x400x3	0
Faltungsschicht, 32 3x3 Kernel, ReLU	400x400x32	896
Max-Pooling, 4x4 Kernel	100x100x32	0
Faltungsschicht, 64 3x3 Kernel, ReLU	100x100x64	18496
Max-Pooling, 4x4 Kernel	25x25x64	0
Faltungsschicht, 128 3x3 Kernel, ReLU	25x25x128	73856
Upsampling	50x50x128	0
Faltungsschicht, 64 3x3 Kernel, ReLU	50x50x64	73792
Upsampling	100x100x64	0
Faltungsschicht, 32 3x3 Kernel, ReLU	100x100x32	18464
Upsampling	200x200x32	0
Faltungsschicht, 16 3x3 Kernel, ReLU	200x200x16	4624
Upsampling	400x400x16	0
Faltungsschicht, 8 3x3 Kernel, ReLU	400x400x8	1160
Faltungsschicht, 1 1x1 Kernel, Sigmoid	400x400x1	9

- Aus jedem Bild wurden zufällige Ausschnitte herausgetrennt (random cropping)
- Der Kontrast der Bilder wurde zufällig verändert
- Die Helligkeit der Bilder wurde zufällig verändert

Durch das wiederholte Anwenden dieser Methoden konnte der ursprüngliche Trainingsdatensatz von 949 Bildern auf 22776 Bilder vergrößert werden.

Als Validierungs-Datensatz, der während des Trainings beobachtet wurde, wurden schon vor der Data-Augmentation 20 Bilder per Hand ausgewählt, die möglichst viele der verschiedenen Szenarien abdecken sollten. Auf diese Weise war es auch möglich, verschiedene Architekturen anhand ihrer Ergebnisse auf diesen handselektierten Bildern miteinander zu vergleichen.

Der Wert der Verlustfunktion während der ersten 15 Epochen des Trainings ist in Abbildung 8 zu sehen. Eine Epoche bedeutet, dass jedes Bild im Datensatz einmal für einen Gradientenschritt verwendet wurde. Das Training wurde dann solange durchgeführt, bis sich der Verlust auf den 20 Validierungsbildern nicht mehr verbessert hat. Man bezeichnet diese Technik auch als **early stopping** [5]. Das Modell mit dem niedrigsten Validierungs-Verlust wurde dann für den weiteren Einsatz in der Pipeline ausgewählt.

#### 4.1.3 Verarbeitung der Modellvorhersagen

Um das Modell in der Pipeline einsetzen zu können, muss aus den Modellvorhersagen noch der Bereich extrahiert werden, welcher das Nummernschild enthält. Die hierzu eingesetzte Vorgehensweise ist in Abbildung 9 anhand eines Beispielbildes dargestellt.

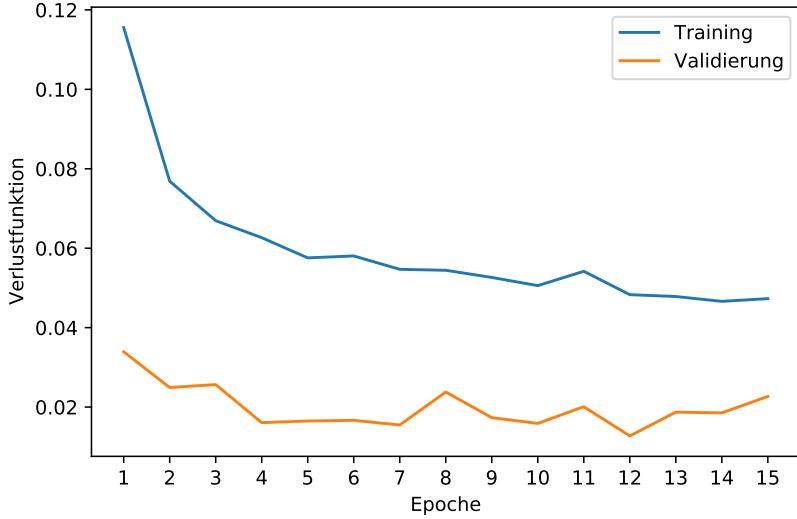


Abbildung 8: Wert der logistischen Verlustfunktion für Trainings- und Validierungsdatensatz innerhalb der ersten 15 Epochen.

Der erste Schritt besteht darin, die Vorhersagen des Modells in eine binäre Maske umzuwandeln, welche nur die Werte 1 und 0 enthält (1 für Pixel im Nummernschild, 0 für den Rest). Da die Ausgabeschicht des eingesetzten Modells aufgrund der Sigmoid-Funktion Werte im Intervall (0, 1) ausgibt, wird zu diesem Zweck (analog zur logistischen Regression) ein Schwellenwertverfahren eingesetzt. Hat ein Pixel in der Modellvorhersage einen Wert größer als 0,5, so wird dieser in der binären Maske auf 1 gesetzt, alle anderen Ausgaben werden auf 0 gesetzt.

Anschließend wird anhand der binären Maske ein rechteckiger Ausschnitt bestimmt, in welchem das Nummernschild vermutet wird. Dies geschieht so, dass zunächst für jeden zusammenhängenden Bereich aus Einen ein parallel zu den Koordinatenachsen verlaufendes umschließendes Rechteck bestimmt wird. Das flächenmäßig Größte dieser Rechtecke wird dann in der Pipeline zur Extraktion des Nummernschildes verwendet.

Man beachte, dass hier implizit angenommen wird, dass sich Nummernschilder stets durch ein parallel zur x- und y-Achse verlaufendes Rechteck beschreiben lassen. Mögliche Probleme mit diesem Ansatz, wie beispielsweise rotierte Nummernschilder, liegen auf der Hand und werden später in Abschnitt 5 diskutiert. Dennoch ist ein Vorteil dieses Ansatzes, dass er sehr leicht umzusetzen ist und trotzdem, wie wir später in Abschnitt 5 sehen werden, zu guten Ergebnissen führen kann.

## 4.2 Texterkennung

Das Auslesen der Zeichen des Nummernschildes aus dem zuvor mithilfe des CNN extrahierten Ausschnitt wurde mithilfe der Optical Character Recognition (im Folgenden OCR) Software Tesseract [16] realisiert.

Damit Tesseract optimal arbeiten kann, werden zunächst einige Vorverarbeitungs-

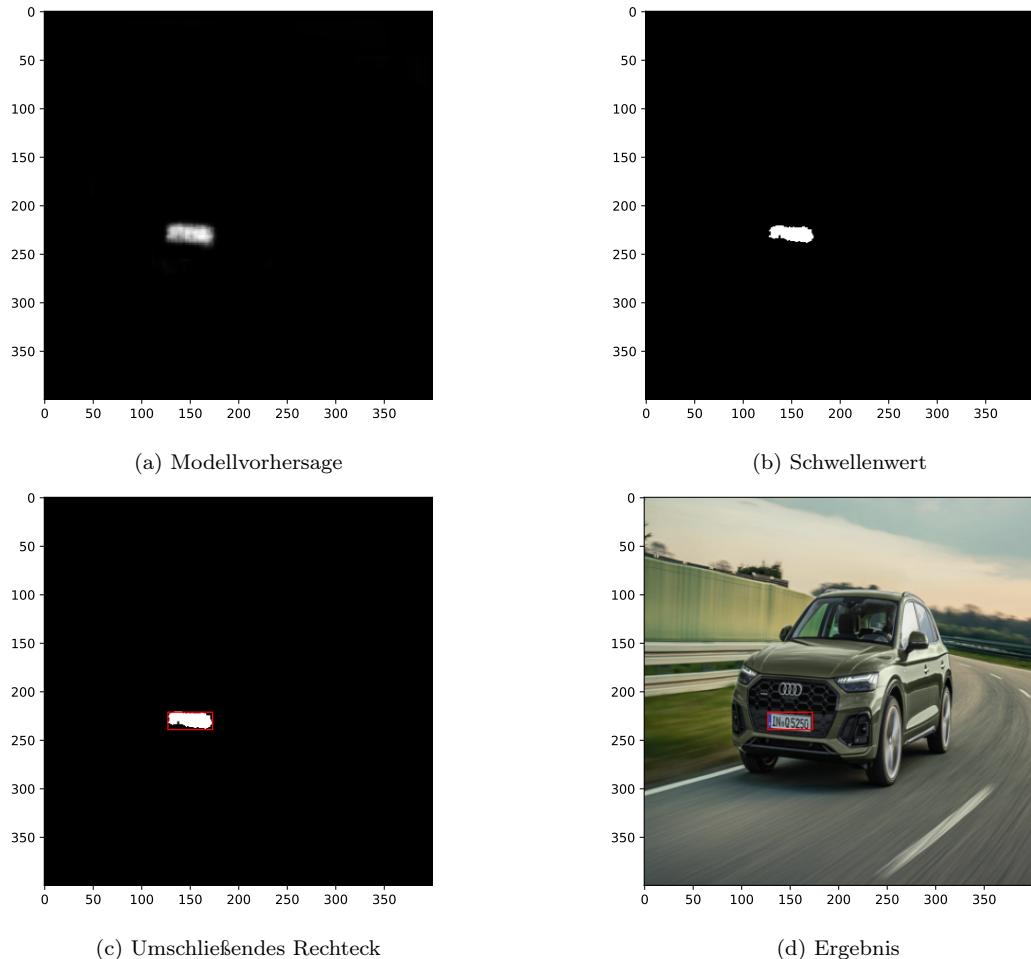


Abbildung 9: Verarbeitung der Modellvorhersage zur Extraktion des Nummernschildes. Das hier gezeigte Bild wurde weder zum Training noch zur Validierung während des Trainings verwendet.

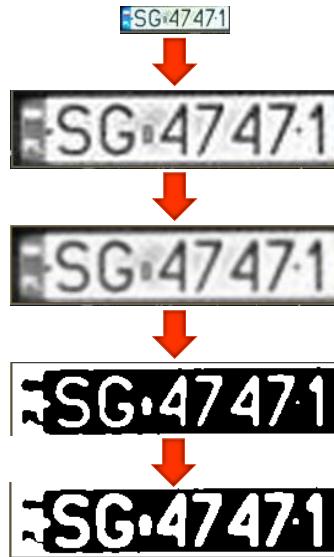


Abbildung 10: Die Vorverarbeitungsschritte eines Nummernschildes von oben nach unten aufgeführt:  
 (1) Vergrößerung und Umwandlung in Graustufen, (2) Bildglättung, (3) Schwellenwert und Invertierung, (4) Dilatation

schritte durchgeführt, die in Abbildung 10 dargestellt sind. Hierbei wurde das Bild zunächst in Graustufen umgewandelt und mit einem Weichzeichner geglättet, um kleinere Unregelmäßigkeiten und Bildrauschen zu entfernen. Anschließend wurde auch hier ein Schwellenwertverfahren eingesetzt, um die Zeichen klarer vom Hintergrund abzugrenzen. Zur Bestimmung des Schwellenwertes kam Otsu's Methode [12] zum Einsatz, die extra für eine möglichst klare Trennung von Vorder- und Hintergrund entwickelt wurde. Im nächsten Schritt wurde das Bild dann invertiert (Einsen wurden auf Nullen gesetzt und umgekehrt), um dann die weißen Bereiche, welche auch die zu erkennenden Zeichen enthalten, zu erweitern (man spricht auch von Dilatation). Sämtliche dieser Schritte wurden mithilfe der Computervision-Programmbibliothek OpenCV [2] durchgeführt.

Für den Einsatz von Tesseract müssen im nächsten Schritt noch die einzelnen Zeichen ausgeschnitten werden. Dies wurde mithilfe der automatisierten Konturenfindung von OpenCV umgesetzt. Die darauf folgenden Schritte der Pipeline sind in Abbildung 11 aufgeführt. Die ausgeschnittenen Zeichen werden invertiert und anschließend durch Tesseract erkannt.

## 5 Ergebnisse

Als Basis für eine Ergebnisdiskussion sind in Abbildung 12 die vorhergesagten Nummernschilder für 10 Testbilder zu sehen, die alle weder beim Training noch bei der Validierung zum early stopping verwendet wurden.

Was den ersten Teil der Pipeline angeht, also die Bildsegmentierung mithilfe eines CNN, so lässt sich erkennen, dass das CNN für jedes der Bilder den relevanten Bereich,

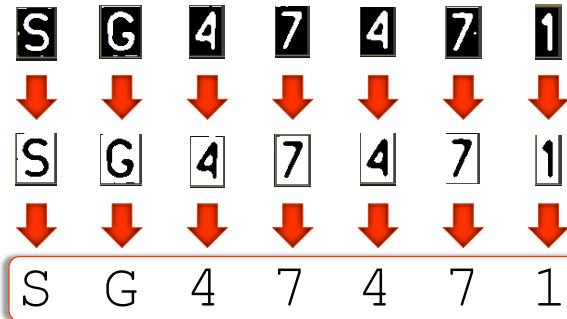


Abbildung 11: Die ausgeschnittenen Zeichen werden wieder invertiert und dann durch Tesseract erkannt.

der das Nummernschild enthält, korrekt vorhersagen konnte. Selbst die sehr abgedunkelten Lichtverhältnisse in Bild (i) konnten die Vorhersage nicht negativ beeinträchtigen. Dies kann als Anzeichen gedeutet werden, dass es sich hier um eine sehr vielversprechende Technologie handeln könnte, was das Auffinden der Nummernschilder angeht. Die Vorhersagen können also als proof-of-concept für den ersten Teil der Pipeline dienen.

Es muss allerdings beachtet werden, dass sich alle Nummernschilder in Abbildung 12 hinreichend gut durch achsenparallele Rechtecke modellieren lassen, was ja wie in 4.1 beschrieben, quasi eine Grundannahme für die Berechnung der umschließenden Rechtecke war. Für die Funktionsweise der Bildsegmentierung auf Daten, für die dies nicht zutrifft, kann also anhand der hier vorgenommenen Betrachtung keine Aussage getroffen werden.

Was den zweiten Teil der Pipeline, also die Texterkennung angeht, so fallen leider ein paar kleinere Mängel ins Auge. Lediglich bei Bild (a) wurde das Nummernschild korrekt ausgelesen. Ansonsten sind die Vorhersagen von einigen kleinen Schwachstellen geprägt. Ein Fehler, der häufig vorkommt, ist das scheinbar willkürliche Verdoppeln gelesener Zeichen, wie es beispielsweise in Bild (h) und auch in Bild (j) zu sehen ist. Darüber hinaus passiert es wie in Bild (d) und (g) häufig, dass einige Zeichen garnicht erst erkannt werden. In Bild (f) wurde sogar, obwohl der Bereich des Nummernschildes zuvor richtig ausgelesen wurde, kein einziges Zeichen erkannt.

Es lässt sich insgesamt also erkennen, dass die Texterkennung der aktuelle Problem-Punkt in der Pipeline ist. Im Folgenden sind daher einige Ideen aufgeführt, wie dieser Teil der Pipeline in Zukunft noch verbessert werden kann.

## 5.1 Ideen zur Verbesserung der Texterkennung

**Framework zur iterativen Verbesserung** Bevor konkrete Verbesserungsansätze eingebbracht werden können, bedarf es zunächst eines Frameworks, also einer allgemeinen Vorgehensweise, wie zielgerichtet auf eine Verbesserung der Pipeline hingearbeitet werden kann. Ein möglicher Ansatz ist es, sich zuerst auf eine Metrik festzulegen, die die Qualität der Pipeline beschreibt und dann anhand dieser Metrik mögliche Änderungen und Verbesserungen zu beurteilen. Neuerungen würden nur dann akzeptiert werden, wenn sie auch zu einer Verbesserung der zu Anfang festgelegten Metrik führen.



Abbildung 12: Modellvorhersagen für 10 ungewohnte Testbilder. Die vorhergesagten Nummernschilder sind durch rote Rechtecke markiert.

Eine mögliche Metrik, die in diesem Kontext sinnvoll erscheint, ist die Levenshtein-Distanz, die erstmals in [10] eingeführt wurde. Die Levenshtein-Distanz gibt an, wieviele Einfüge-, Lösch- und Ersetz-Operationen notwendig sind, um eine Zeichenkette in eine andere zu überführen. In der hier vorliegenden Situation könnte man also die Levenshtein-Distanzen zwischen den Modellvorhersagen und den tatsächlichen Nummernschildern messen und so eine Aussage über die Qualität der Pipeline erhalten.

Zur Verbesserung der Texterkennung könnte man nun also einen Teil der Daten beiseite legen, beispielsweise die Bilder in Abbildung 12, und sich als Ziel setzen, die mittlere Levenshtein-Distanz der Pipeline auf diesen Bildern zu minimieren. Hierbei könnte man dann iterativ vorgehen: Hat man eine Idee zur Verbesserung der Pipeline entwickelt, so wird diese anhand der mittleren Levenshtein-Distanz auf den Testdaten evaluiert. Nur dann, wenn die mittlere Levenshtein-Distanz durch das Einbringen dieser Idee verkleinert werden konnte, wird die Idee in die Pipeline aufgenommen.

Kurz angemerkt sei an dieser Stelle noch, dass man sich bei dieser Vorgehensweise möglicherweise anfällig für Overfitting machen kann, da man seine Entscheidungen maßgeblich von einigen zuvor selektierten Bildern abhängig macht, auf denen die mittlere Levenshtein-Distanz berechnet wird. Dies könnte aber dadurch kompensiert werden, dass man eine sehr vielseitige Bandbreite von Bildern zu dieser Validierung heranzieht, die möglichst viele der denkbaren Szenarien abdecken.

**Einfache Testfälle als „Sanity Checks“** Anhand der Vorhersagen auf den Beispielbildern fällt auf, dass es möglicherweise noch unentdeckte Fehler bei der Implementierung der Texterkennung geben könnte. Insbesondere das häufige doppelte Erkennen von Zeichen einerseits, sowie das häufige Auslassen von Zeichen andererseits, deuten darauf hin. Um in Zukunft mögliche „Bugs“ ausschließen zu können, ist es sinnvoll, ein paar einfache Tests einzuführen, die die Pipeline erfüllen muss.

Beispielsweise wäre denkbar, die Texterkennung mit einzelnen Zeichen zu testen, um sicherzustellen, dass diese korrekt erkannt werden. Darauf aufbauend könnte man auch ein paar sehr klar erkennbare Nummernschilder als Testfälle einbringen, die auf jeden Fall von der Pipeline korrekt erkannt werden müssen. Möchte man dann eine Verbesserung in die Pipeline einbringen, so muss diese Verbesserung zunächst alle Testfälle korrekt erfüllen. Dieses Konzept bezeichnet man in der Software-Entwicklung auch als Regressions-Tests [7].

**Wechsel der Technologie** Ein weiterer Punkt, der nicht vergessen werden darf, ist die Wahl der Technologie für die Texterkennung. Es wäre beispielsweise denkbar, anstelle der Kombination von Vorverarbeitungsschritten mit OpenCV und der anschließenden Erkennung durch Tesseract, ebenfalls CNNs zur Erkennung der Zeichen einzusetzen. Dieses Vorgehen kam beispielsweise in [15] zum Einsatz und hat zu sehr guten Ergebnissen geführt.

Dennoch ist anhand der Vorhersagen auf den Beispielbildern wohl davon auszugehen, dass das volle Potenzial des bisherigen Ansatzes noch nicht vollends ausgeschöpft werden konnte. Ein kompletter Technologiewechsel, und damit eine komplettene Neuim-

plementierung der Texterkennung, sollte daher eher als eine Art letzter Ausweg gesehen werden.

## 6 Zusammenfassung

In diesem Projekt wurde gezeigt, wie das automatisierte Auslesen von Nummernschildern aus Bilddaten mittels einer zweistufigen Vorhersagepipeline umgesetzt werden kann.

In der ersten Stufe kam ein CNN zur Extraktion des Nummernschildes durch Bildsegmentierung zum Einsatz. Dieses neuronale Netz wurde so trainiert, dass es für jedes Eingabebild eine binäre Maske vorhersagen soll, in welcher sich das Nummernschild befindet. Insgesamt zeigte der Einsatz des CNN zur Bildsegmentierung vielversprechende Resultate: Auf einem Testdatensatz von 10 Bildern, die weder zum Training noch zum early stopping verwendet wurden, konnte jedes der Nummernschilder trotz teils schlechter Lichtverhältnisse korrekt vom CNN erkannt werden.

Anhand der Beispieldaten konnten allerdings keine Aussagen bezüglich des Verhaltens auf rotierten Nummernschildern gemacht werden. Zu dieser Situation lassen sich aber zwei Überlegungen anstellen: Zum einen wird durch Betrachtung der in 2 beschriebenen Trainingsdaten klar, dass die Kameras, welche ein Nummernschild aufzeichnen, sehr häufig so positioniert sind, dass das Nummernschild achsenparallel zum Bildausschnitt verläuft. Die Rotationen, die in der Praxis zu beobachten sind, scheinen also wenn überhaupt nur recht klein auszufallen. Man könnte also einerseits argumentieren, dass rotierte Nummernschilder in der Praxis nicht häufig auftreten und dass kleinere Rotationen immer noch hinreichend durch ein achsenparalleles Rechteck erfasst werden können. Andererseits ist es so, dass sich der vorliegende Ansatz der Berechnung eines umschließenden Rechtecks auch auf rotierte Rechtecke verallgemeinern lässt. Am CNN müsste dazu nichts geändert werden, lediglich das Ausschneiden der Nummernschilder aus der vorhergesagten binären Maske müsste angepasst werden. Hierin bestünde eine mögliche zukünftige Verbesserung der ersten Stufe der Pipeline.

Die zweite Stufe der Pipeline, also die Zeichenerkennung aus einem extrahierten Nummernschild, wurde mithilfe der OCR-Software Tesseract unter Einsatz verschiedener Vorverarbeitungsschritte durch OpenCV realisiert. Die Ergebnisse dieser Zeichenerkennung waren jedoch nicht zufriedenstellend und müssen noch verbessert werden. In Abschnitt 5.1 wurden hierzu drei mögliche Ansätze diskutiert. Als letzter Ausweg wurde ein Wechsel der Technologie hin zu CNNs zur Texterkennung in Betracht gezogen, wie sie auch in [15] zum Einsatz kam. Es sollte aber zunächst sichergestellt werden, dass die teils noch recht fehlerbehafteten Vorhersagen der Texterkennung nicht auf eventuelle Implementierungsfehler zurückzuführen sind, da sonst möglicherweise noch nicht das volle Potential der Tesseract Software ausgenutzt wird.

Abschließend lässt sich jedoch feststellen, dass zumindest der erste Teil der Pipeline erfolgreich umgesetzt wurde, was ein weiterer Beleg für die Dominanz von CNNs im Bereich der Bildsegmentierung ist. Die so umgesetzte Lokalisierung der Nummernschilder kann nun also als solide Basis für alle weiteren Verbesserungen der Texterkennung dienen.

## Literatur

- [1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from <http://www.tensorflow.org>.
- [2] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.
- [3] X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256. JMLR Workshop and Conference Proceedings, 2010.
- [4] X. Glorot, A. Bordes, and Y. Bengio. Deep sparse rectifier neural networks. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, volume 15 of *Proceedings of Machine Learning Research*, pages 315–323. JMLR Workshop and Conference Proceedings, 2011.
- [5] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. <http://www.deeplearningbook.org>.
- [6] I. Goodfellow, O. Vinyals, and A. Saxe. Qualitatively characterizing neural network optimization problems. In *International Conference on Learning Representations*, 2015.
- [7] B. Homès. *Fundamentals of Software Testing*. John Wiley & Sons, Ltd, 2012.
- [8] G. James, D. Witten, T. Hastie, and R. Tibshirani. *An Introduction to Statistical Learning*. Springer-Verlag New York, 2013.
- [9] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. In Y. Bengio and Y. LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.
- [10] V. Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions, and Reversals. *Soviet Physics-Doklady*, 10(8):707–710, 1966.
- [11] J. Long, E. Shelhamer, and T. Darrell. Fully convolutional networks for semantic segmentation. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 3431–3440, 2015.

- [12] N. Otsu. A threshold selection method from gray-level histograms. *IEEE Transactions on Systems, Man, and Cybernetics*, 9:62–66, 1979.
- [13] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986.
- [14] S. Shalev-Shwartz and S. Ben-David. *Understanding Machine Learning - From Theory to Algorithms*. Cambridge University Press, 2014.
- [15] S. M. Silva and C. R. Jung. License plate detection and recognition in unconstrained scenarios. In *2018 European Conference on Computer Vision (ECCV)*, pages 580–596, Sep 2018.
- [16] R. Smith. An overview of the tesseract ocr engine. *Ninth International Conference on Document Analysis and Recognition (ICDAR 2007)*, 2:629–633, 2007.

## **Abbildungsverzeichnis**

1	Beispielbilder . . . . .	2
2	Verkettung als Graph . . . . .	3
3	Schematische Darstellung einer Faltungsschicht . . . . .	8
4	Max-Pooling und Upsampling . . . . .	9
5	Pipeline . . . . .	10
6	Binäre Maske . . . . .	11
7	Netzarchitektur . . . . .	12
8	Trainings- und Validierungsverlust . . . . .	14
9	Verarbeitung der Modellvorhersage . . . . .	15
10	Vorverarbeitung eines Nummernschildes . . . . .	16
11	Vorverarbeitung der einzelnen Zeichen . . . . .	17
12	Modellvorhersagen . . . . .	18