

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

Christian Peters

5. März 2021

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

Inhaltsverzeichnis

1	Einleitung	1
2	Datenbeschreibung	1
3	Grundlagen	1
3.1	Neuronale Netze	1
3.2	OpenCV	7
3.3	Tesseract	7
4	Pipeline	7
5	Ergebnisse	7
6	Zusammenfassung	7
	Literatur	8

1 Einleitung

Die automatisierte Erkennung von Nummernschildern aus Bilddaten ist ein wichtiger Bestandteil vieler moderner Verkehrssysteme und kommt beispielsweise in Parkhäusern, Mautstellen oder bei der Identifikation gestohlener Fahrzeuge zum Einsatz [12].

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 die Position des Nummernschildes bestimmt und im zweiten Schritt anhand der zuvor ermittelten Position die Zeichen des Nummernschildes ausliest.

Bei der Positionsbestimmung des Nummernschildes kommen sogenannte Convolutional Neural Networks zum Einsatz, eine spezielle Art von Neuronalen Netzen, deren Grundlagen in Abschnitt 3 beschrieben werden. Zum Auslesen der Zeichen werden die Programmbibliotheken OpenCV [2] und Tesseract [13] verwendet.

Ein Überblick über das vorliegende Datenmaterial, welches bei der Erstellung der Pipeline 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 abgerufen werden.

2 Datenbeschreibung

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

Zu jedem Bild liegen außerdem Informationen zur Position des Nummernschildes innerhalb des Bildes anhand von Pixel Koordinaten vor. Die Position des Nummernschildes ist dabei durch die 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 sind anhand der roten Rechtecke eingezeichnet.

3 Grundlagen

3.1 Neuronale Netze

Neuronale Netze sind eine Modellklasse, welche zur Lösung des bereits ausführlich in [8] beschriebenen Problems des statistischen Lernens eingesetzt werden können. In dieser

¹Die Originaldaten sind unter https://github.com/phibuc/Lab_FS_Data, sowie unter <https://github.com/RobertLucian/license-plate-dataset> einsehbar.

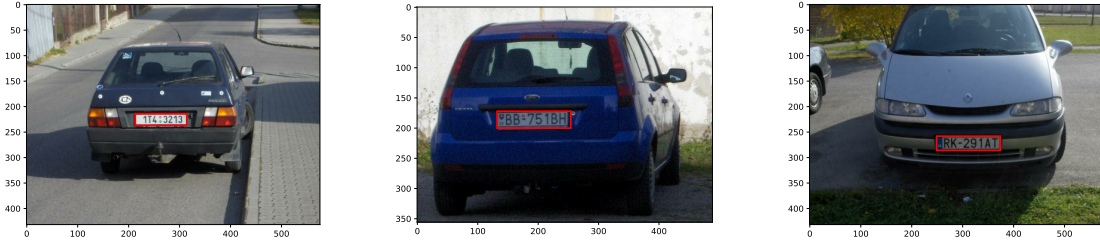


Abbildung 1: Drei Beispiele aus dem vorliegenden Datensatz. Die Nummernschilder sind anhand ihrer Koordinaten rot umrandet.

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 ϵ 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 auffallen 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 es sich bei den x_i hier um Bilddaten handelt, und zwar um Bilder von Autos mit Nummernschildern. Innerhalb der y_i sollen dann später Informationen zur Position der Nummernschilder kodiert werden, die durch das neuronale Netz vorhergesagt werden sollen. Details zur Kodierung der y_i und auch der x_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 [6] und sind hier auf das grundlegendste reduziert worden. Obwohl es der Name *neuronale* Netze suggeriert, wird auch hier genau wie in [6] ebenfalls auf jegliche biologische Motivation verzichtet, damit nicht der falsche Eindruck entstehen 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 ungesehenen Daten anzustellen.

3.1.1 Aufbau

Wie oben in 3.1 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 ersteinmal 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.

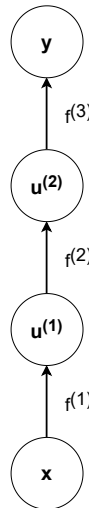


Abbildung 2: Verkettung dreier Funktionen dargestellt als gerichteter azyklischer Graph. Die Knoten $u^{(1)}$ und $u^{(2)}$ stellen die Zwischenergebnisse der Funktionen $f^{(1)}$ und $f^{(2)}$ dar, der Knoten y repräsentiert die Ausgabe des Netzes.

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. Wie genau die Schichten eines neuronalen Netzwerkes nun aussehen können, soll im nächsten Abschnitt etwas detaillierter beschrieben werden.

3.1.2 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 3.1.4 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 Φ 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.

3.1.3 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 [5] 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 [5], 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 [11] (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. Das weitere Vorgehen ist dann sehr ähnlich zur logistischen Regression [8].

3.1.4 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 $f(x)$ von einer Eingabe 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 θ zusammenfassen, welcher die Funktion f parametrisiert (wir sprechen deshalb im Folgenden von f_θ).

Der mittlere quadratische Approximationsfehler unseres neuronalen Netzes f_θ für einen Parametervektor θ 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_θ besteht nun also darin, einen Parametervektor θ 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.

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 [11] im Bezug auf allgemeine Machine Learning Probleme ausführlich beschrieben und analysiert worden und soll hier nur einmal bezüglich der vorliegenden Problemstellung kurz umrissen werden.

Stochastischer Gradientenabstieg Möchte man die Funktion $L(\theta)$ durch stochastischen 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 [4] 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 stochastischen 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:

$$\theta^{(t+1)} = \theta^{(t)} - \eta \nabla L(\theta^{(t)}) \quad (4)$$

Hierbei gibt η eine Schrittweite an, die zuvor vom Anwender spezifiziert werden muss.

Die Idee dabei ist, dass der negative Gradient immer in die Richtung des steilsten Abstieges 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 [7] 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.

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_θ 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 [10] 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 3.1.1 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 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 Programmbibliotheken wie Tensorflow [1] implementieren dieses Verfahren durch effiziente Ausnutzung vorhandener Hardware (insbesondere GPUs) und erfreuen sich daher großer Beliebtheit.

3.1.5 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 [6].

3.2 OpenCV

3.3 Tesseract

4 Pipeline

5 Ergebnisse

6 Zusammenfassung

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] F. Chollet. *Deep Learning with Python*. Manning Publications Co., 2017.
- [4] 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.
- [5] 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.
- [6] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. <http://www.deeplearningbook.org>.
- [7] I. Goodfellow, O. Vinyals, and A. Saxe. Qualitatively characterizing neural network optimization problems. In *International Conference on Learning Representations*, 2015.
- [8] G. James, D. Witten, T. Hastie, and R. Tibshirani. *An Introduction to Statistical Learning*. Springer-Verlag New York, 2013.
- [9] M. A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015. <http://neuralnetworksanddeeplearning.com/>.
- [10] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986.
- [11] S. Shalev-Shwartz and S. Ben-David. *Understanding Machine Learning - From Theory to Algorithms*. Cambridge University Press, 2014.
- [12] 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.

- [13] R. Smith. An overview of the tesseract ocr engine. *Ninth International Conference on Document Analysis and Recognition (ICDAR 2007)*, 2:629–633, 2007.