Hochschule Rhein-Waal

Fakultät Kommunikation und Umwelt

Prof. Dr. Frank Zimmer (Hochschule Rhein-Waal)

Dipl.-Ing. (FH) Stefan Peuser (Kommunales Rechenzentrum Niederrhein)

Künstliche Intelligenz in einer Geodateninfrastruktur

Digitalisierung durch eine Tabellenerkennung in bestehenden (Geo-)Datensätzen für eine DIN konforme Speicherung in einer offenen urbanen Datenplattform mit Hilfe von künstlichen neuronalen Netzen im Rechenzentrumsbetrieb

Bachelorarbeit

Vorgelegt von: Lutz Gooren

Hochschule Rhein-Waal

Fakultät Kommunikation und Umwelt

Prof. Dr. Frank Zimmer (Hochschule Rhein-Waal)

Dipl.-Ing. (FH) Stefan Peuser (Kommunales Rechenzentrum Niederrhein)

Künstliche Intelligenz in einer Geodateninfrastruktur

Digitalisierung durch eine Tabellenerkennung in bestehenden (Geo-)Datensätzen für eine DIN konforme Speicherung in einer offenen urbanen Datenplattform mit Hilfe von künstlichen neuronalen Netzen im Rechenzentrumsbetrieb

Bachelorarbeit im Studiengang
E-Government
zur Erlangung des akademischen Grades
Bachelor of Science

Vorgelegt von: Lutz Gooren
Matrikelnummer: 26868

Fälligkeitsdatum: 13.02.2023

Abstract

Künstliche Intelligenz in Form von neuronalen Netzen wird in der öffentlichen Verwaltung bislang wenig eingesetzt. Im Zuge der Digitalisierung im Bereich der öffentlichen Verwaltung müssen bestehende Datensätze, welche nicht- oder schwer maschinenlesbar sind, über offene Schnittstellen und Datenstandards bereitgestellt werden. Schätzungsweise 80 % der erfassten Daten sind in der öffentlichen Verwaltung nicht- oder schwer maschinenlesbar. In dieser Arbeit soll geprüft werden, ob ein künstliches neuronales Netz, welches auf der ImageNet Datenbank basiert und durch weitere Beispiele trainiert worden ist, bestehende (nicht-maschinenlesbare) Datensätze in Tabellenform aus dem Bereich der Geoinformatik erkennen und in einer passenden Datei mit offenem Datenstandard (teil-)automatisiert speichern kann, um diese für den Einsatz in einer offenen urbanen Datenplattform vorzubereiten. Zudem werden Optimierungsvorschläge für eine Verbesserung des neuronalen Netzes genannt und umgesetzt. Im Laufe der Arbeit ergibt sich, dass das trainierte Modell für den praktischen Einsatz nicht ausreichend genau funktioniert, aber eine gute Basis für weitere Optimierungen liefert, etwa durch eine Vergrößerung des Trainingsdatensatzes, welche die Anforderung einer Steigerung der Rechenleistung voraussetzt.

Schlüsselwörter:

Geodateninfrastruktur, Digitalisierung, Künstliche Intelligenz, Computer Vision, Tabellenerkennung, Datenplattformen

Inhaltsverzeichnis

1. Einleitung ... 1
1.1 Motivation ... 1
1.2 Problemstellung ... 1
1.3 Methodik ... 1
1.4 Struktur der Arbeit ... 2
2. Grundlagen ... 2
2.1 Aktueller Stand von Datenplattformen ... 2
2.2 Geodateninfrastrukturen ... 3
2.3 DIN-Norm 91357– Referenzarchitektur Urbane Datenplattform ... 3
2.4 Künstliche neuronale Netze ... 4
2.4.1 Einsatzgebiete ... 4
2.4.2 Modelle ... 4
2.4.3 Bedeutung von Trainingsdaten ... 5
2.4.4 Zuverlässigkeit ... 5
2.5 Verwendete Softwarewerkzeuge und Bibliotheken ... 6
2.5.1 Python ... 6
2.5.2 PyCharm ... 6
2.5.3 Allgemeine Python Bibliotheken ... 6
2.5.4 TensorFlow ... 6
2.5.5 Keras ... 8
2.5.6 cuDNN ... 8
2.5.7 Pytesseract ... 8
2.5.8 NumPy ... 8
2.5.9 Hasty.ai ... 9
3. Analyse des Problems ... 10
3.1 Auswahl eines Modelltyps für das neuronale Netz ... 10
3.2 Vorbereiten geeigneter Datensätze für das neuronale Netz ... 12
3.3 Manuelles digitalisieren der Daten ... 14

4. Ergebnisse und Diskussion 15
4.1 Aufbau des neuronalen Netzes 15
4.2 Ergebnisse der trainierten Modelle 17
4.3 Anpassungen für die Erzielung besserer Ergebnisse 19
4.3.1 Training des Modells in einem größeren Umfang 20
4.3.2 Anpassung der Parameter des Lernprozesses oder des Modells 20
4.3.3 Weitere Anpassungen 20
4.4 Ergebnisse des Modells mit angepassten Parametern 21
4.5 Anpassen des Netzes für Tabellen und Spaltenmaske 22
4.6 Ergebnisse des angepassten Tabellen-/Spalten-Modells 24
4.7 Auslesen der Informationen mit Pytesseract 26
4.8 Reevaluierung der Trainingsdaten 28
5. Fazit und Ausblick 29
5.1 Erkennung von Zusammenhängen in einer UDP 29
5.2 Auswertung von Punktwolken mit Hilfe von neuronalen Netzen 30
Literatur 31
Anhang 33
Anlage 1 - pdfToJPG.py: 33
Anlage 2 - preProcess.py: 33
Anlage 3 - trainModel_head_front_field.py: 38
Anlage 4 - loadAndPredict_head_front_field.py: 44
Anlage 5 - trainModel_table_col.py: 46
Anlage 6 - loadAndPredict_table_col.py: 52

Abkürzungsverzeichnis

CNN	Convolutional Neural Network
CPU	Central Processing Unit (Hauptprozessor)
CSV	Comma-seperated values
DIN	Deutsches Institut für Normung
GDI	Geodateninfrastruktur
GPU	Graphical Processing Unit (Grafikkarte)
IoT	Internet of Things
JSON	JavaScript Object Notation
KI (AI)	Künstliche Intelligenz (Artificial Intelligence)
OCR	Optical character recognition
TPU	Tensor Processing Unit
UDP	(Offene) urbane Datenplattform
VRAM	Video Random Access Memory
XML	Extensible markup language
WCS	Web Coverage Service
WFS	Web Feature Service
WMS	Web Map Service

Abbildungsverzeichnis

Abbildung 1: Schematischer Aufbau von Keras in Nutzung mit TensorFlow, Quelle: Chollet 8
Abbildung 2: Label für Tabelle und Spalte werden in hasty.ai erstellt, Quelle: eigene Abbildung 9
Abbildung 3: Samples von Bildern. Unterteilt in Farbkanäle, Quelle: Chollet 10
Abbildung 4: Funktionsweise eines Kernels in einem CNN, Quelle: Frochte 11
Abbildung 5: Input (links) und generierte Maske für Spalten aus Koordinaten der Label (rechts), Quelle: eigene Abbildung 14
Abbildung 6: Label für Kopfzeile, Vorspalte und Felder werden in hasty.ai erstellt, Quelle: eigene Abbildung 16
Abbildung 7: Aufbau des TableNet unter Verwendung der VGG19 Architektur, Quelle: basierend auf: Paliwal, D, et al 17
Abbildung 8: Input, Kopfzeilenmaske, Vorspaltenmaske, Feldmaske und zusammengesetzte Maske als Alpha-Kanal auf Input, Quelle: eigene Abbildung 19
Abbildung 9: Input, Dilation und Erosion in cv2, Quelle: OpenCV.org 21
Abbildung 10: y-Achse: Loss (Verlust), x-Achse: Epoch (Durchlauf), Verlauf gilt für Validierung, Quelle: eigene Abbildung 22
Abbildung 11: Input, Kopfzeilenmaske, Vorspaltenmaske, Feldmaske und zusammengesetzte Maske als Alpha-Kanal auf Input, Quelle: eigene Abbildung 24
Abbildung 12: y-Achse: Loss (Verlust), x-Achse: Epoch (Durchlauf), Verlauf gilt für Validierung, Quelle: eigene Abbildung 25
Abbildung 13: Bildinput bei 200DPI (links), Ausgabe von Pytesseract auf der Konsole (rechts), umgebrochen auf zwei Spalten, Quelle: eigene Abbildung 26
Abbildung 14: Ausgabe von Pytesseract auf der Konsole bei 600DPI Eingangsauflösung, umgebrochen auf drei Spalten, Quelle: eigene Abbildung 27
Abbildung 15: Ausgabe Pytesseract mit Modus: psm 6 bei 600DPI Eingangsauflösung, umgebrochen auf zwei Spalten, Quelle: eigene Abbildung 28

Tabellenverzeichnis

Tabelle 1: Leistungsmerkmale und Preis aktueller High End CPU/GPU, basierend auf: Raschka, Mirjalili , Leistungsdaten von: techpowerup.com ... 7
Tabelle 2: Verwendete Architekturen, Eingangsauflösungen der Bilder sowie die Tiefe der Netze, Quelle: eigene Tabelle ... 15
Tabelle 3: Netzarchitekturen, Epochs und Verlustmetrik des Validierungsdatendatzes (Kopfzeile, Vorspalte, Felder), aufsteigend sortiert nach Verlust, Quelle: eigene Abbildung ... 18
Tabelle 4: Parametereinstellungen für VGG19 Netz (Kopfzeile / Vorspalte / Felder) und minimaler Verlust (Validierung), Quelle: eigene Tabelle ... 22
Tabelle 5: Netzarchitekturen, Epochs und Verlustmetrik des Validierungsdatensatzes (Tabelle, Spalte), aufsteigend sortiert nach Verlust, Quelle: eigene Tabelle ... 23
Tabelle 6: Parametereinstellungen für VGG19 Netz (Tabelle/Spalte) und minimaler Verlust (Validierung), Quelle: eigene Tabelle ... 25

1. Einleitung

Die Arbeit behandelt den Einsatz künstlicher Intelligenz (KI) im Rechenzentrumsbetrieb in einer Geodateninfrastruktur (GDI), um bereits vorhandene, nicht-digitale Daten in Tabellenform aus verschiedenen Fachbereichen für die DIN-Norm 91357 anzupassen.

1.1 Motivation

Ungefähr 80 % der Daten, welche die öffentliche Verwaltung besitzt, sind nicht oder nur schwer maschinenlesbar (z.B. auf Papier oder nur als PDF-Datei gespeichert) [1, S. 25]. Bestehende Datensätze müssen zunächst gesichtet und manuell digitalisiert werden, was einem hohen Aufwand entspricht. In der heutigen Zeit können künstliche neuronale Netze eine Vielzahl von Aufgaben unterstützen oder sogar übernehmen. Als Beispiel können hier die Sprachassistenten Google Home oder Alexa gesehen werden [2, S. 32]. Daten der öffentlichen Verwaltung umfassen unter anderem in Tabellen gespeicherte Messwerte, welche zunächst digitalisiert und in einem geeigneten Datenstandard gespeichert werden müssen, bevor diese auf einer Datenplattform abgerufen werden können. Hier kann ein künstliches neuronales Netz zum Einsatz kommen, welches bestehende Datensätze in Form von Bildern auf Tabellen untersucht, und diese mit Hilfe von einer Text- bzw. Zeichenerkennung als Zeichenkette ausgibt.

1.2 Problemstellung

Das Problem sind die großen Mengen an nicht-digitalen Daten der öffentlichen Verwaltung, welche nicht vorbereitet sind, um in eine Datenplattform (und später in einer urbanen Datenplattform, siehe Kapitel 2.3) integriert zu werden. Diese Daten, welche z.B. in Tabellenform in einer PDF-Datei vorliegen können, müssen demnach mit hohem manuellem Aufwand in ein offenes Datenformat (siehe Kapitel 2.3) übertragen werden. Um diesen Prozess zu beschleunigen, können künstliche neuronale Netze trainiert werden, um den Arbeitsablauf zu automatisieren oder zu unterstützen.

1.3 Methodik

Die Arbeit soll prüfen, ob ein neuronales Netz Merkmale von Tabellen erlernen kann, um eine Maske zu berechnen, welche irrelevante Informationen ausblendet, damit eine Zeichenerkennung die Daten auslesen kann. Der Hauptteil dieser Arbeit befasst sich mit dem Erstellen und Trainieren eines künstlichen neuronalen Netzes mit unterschiedlichen Netzarchitekturen, welches genauer in den Kapiteln 3.1 und 4 erklärt wird. Im Vorfeld werden einige Daten gesichtet und auf Eignung für das Training eines künstlichen neuronalen Netzes geprüft. Der Prozess des Labelns (Kennzeichnung) der Trainingsdaten wird zudem auch anhand von einigen Beispielen gezeigt, um die Vorbereitung der Daten genauer verstehen zu können. Um zu untersuchen, ob eine Veränderung der Output-Layer eine Verbesserung bringt, wird in Kapitel 4.5 ein zweites Modell mit den gleichen

Netzarchitekturen wie beim ersten Modell trainiert, jedoch mit dem Unterschied, dass das erste Modell die Ergebnisse aus drei generierten Masken (Kopfzeile / Vorspalte / Felder) zusammensetzt, wohingegen das zweite Modell Masken für die Tabelle als solche und die einzelnen Spalten erstellt und diese übereinanderlegt. Bei beiden Modellen werden die Parameter des Lernprozesses angepasst, um die Ergebnisse auf eine Verbesserung der Erkennungsgenauigkeit der Tabellen zu überprüfen.

1.4 Struktur der Arbeit

Zu Beginn werden kurz einige Grundlagen aus dem Bereich der Datenplattformen (insbesondere Geodateninfrastrukturen) und die DIN-Norm 91357 für eine offene urbane Datenplattform erklärt. Außerdem wird auch die grundlegende Funktion eines neuronalen Netzes (Schwerpunkt liegt auf neuronalen Netzen, welche für den Anwendungsfall im Bereich Computer Vision geeignet sind) und die Bedeutung von Trainingsdaten und die Zuverlässigkeit der Netze erläutert. Um die verwendeten Bibliotheken und Softwarewerkzeuge besser verstehen zu können, werden diese kurz beschrieben. Für das Erstellen des neuronalen Netzes ist die Problemstellung ein wichtiger Bestandteil, um ein geeignetes Netz auswählen zu können, was in Kapitel 3.1 beschrieben wird. In Kapitel 4 werden die Ergebnisse des praktischen Teils präsentiert und einige Verbesserungsvorschläge anhand der Auswertungen genannt und implementiert. Im letzten Kapitel der Arbeit wird eine Zusammenfassung der Ergebnisse geliefert, sowie ein Ausblick auf die zukünftige Verwendung von künstlichen neuronalen Netzen in der öffentlichen Verwaltung gegeben.

2. Grundlagen

Um den Vorgang, den diese Arbeit behandelt, besser verstehen zu können, wird zunächst ein allgemeiner Überblick über den derzeitigen Stand von Datenplattformen gegeben, die Bestandteile einer Geodateninfrastruktur, sowie die DIN-Norm 91357 erklärt. Außerdem werden die genutzten Werkzeuge kurz erläutert und die zugrundeliegende Architektur des neuronalen Netzes beschrieben.

2.1 Aktueller Stand von Datenplattformen

Datenplattformen sind meist nur für die eigenen, dezentrale Bereiche (z.B. Geoportale mit Karten- und Gebäudeinformationen oder Messdaten von Luftqualitätsmessstationen) eingerichtet (sogenannte Datensilos). Oft fehlt, ohne einen persönlichen Austausch der Mitarbeitenden, sogar die Kenntnis über die Existenz von Daten in anderen Abteilungen [1, S. 25]. Eine übergreifende Integration zwischen verschiedenen Fachbereichen ist dabei noch nicht möglich. Selbst wenn die Daten miteinander verbunden sind, wird aus den meisten verbundenen Datensätzen aufgrund mangelnder Erfahrung in der öffentlichen Verwaltung kein Vorteil gewonnen, da die Verknüpfung nicht über den eigentlichen

Anwendungsfall hinaus geht [1, S. 26]. Dies bedeutet, dass es nicht nur eine einzelne Anlaufstelle für Informationen gibt. Somit müssen aus verschiedenen Quellen Informationen gewonnen werden, um einen Überblick über verschiedene Daten zu erhalten. Außerdem erschweren bzw. verhindern Datensilos den Zugriff auf vorhandene Daten über die Grenzen eines Fachbereiches hinweg. Dies ist selbst bei Vorreitern der digitalen Stadt in Deutschland, wie etwa Hamburg, der Fall [1, S. 44].

2.2 Geodateninfrastrukturen

Geodateninfrastrukturen (GDI) dienen der Realisierung verschiedener Anwendungsfälle in Bezug mit Geodaten. Diese Infrastruktur besteht aus den folgenden Komponenten, welche unter anderem auf internationalen Normen und Standards beruhen [basierend auf: 3]:

- **Geodaten:** Sind raumbezogene Daten wie z.B. Landschaften (Topografie), Bebauungspläne oder Flurstücke
- **Geodienste:** Stellen einen Zugriff auf Geodaten zur Verfügung, wie zum Beispiel einzelne Kartenausschnitte. Diese Dienste gehören unter anderem zur Kategorie *Web Coverage Service (WCS), Web Coverage Processing Service (WCPS), Web Feature Service (WFS), Web Map Service (WMS), Table Joining Service, Catalogue Service for the web (CSW), Web Processing Service (WPS), […]* [4, S. 23-28].
- **Metadaten:** Sind Informationen, die bestehende Daten, Dienste o.Ä. beschreiben. Zum Beispiel: Ansprechpartner eines Datensatzes oder Dienstes.
- **Netzwerke:** Infrastruktur der Server usw., welches die GDI zur Verfügung stellt
- **Normen und Standards:** Werden benötigt, um einen Austausch zwischen Anwendungen zu vereinfachen, was durch einen standardisierten Zugriff und Schnittstellen (wie WMS / WFS) erreicht wird.

2.3 DIN-Norm 91357– Referenzarchitektur Urbane Datenplattform

Die DIN-Norm 91357 beschreibt eine Referenzarchitektur für sogenannte (offene) urbane Datenplattformen (UDP). Eine UDP soll so konzipiert sein, dass in der Praxis deutschlandweit (urbane) Daten (z.B. Smart-City, IoT, Kontext Städtedaten) auf einer zentralen Datenplattform abgerufen werden können. Die Daten sind bereits auf verschiedenen Plattformen vorhanden und müssen durch eine Architektur wie die UDP verbunden werden [5, S. 6]. Wichtig ist, dass die UDP nur die Verknüpfung selbst vornimmt. Die eigentlichen Daten, bleiben auf den ursprünglich verwendeten Plattformen und werden über Schnittstellen zur Verfügung gestellt [1, S. 45-46]. Unter solche Daten fallen zum Beispiel Messdaten bezüglich der Luftqualität in Städten oder die Auslastung der Parkplätze in einer Einkaufsstraße. Ein möglicher Anwendungsfall wäre der Vergleich

der Luftqualität in verschiedenen Städten und Kommunen oder innerhalb eines Kreises. Außerdem könnte ein Austausch von Verkehrsdaten stattfinden, um den Verkehr städteübergreifend regulieren zu können. Dies setzt eine einheitliche Speicherung voraus, wobei die DIN-Norm 91357 vorgibt, dass die Daten in einem offenen Datenstandard gespeichert und über standardisierte Schnittstellen und Ressourcen zur Verfügung gestellt werden müssen [1, S. 46-47]. Durch die Anforderungen eines offenen Datenstandards wären unter anderem folgende Lösungen nutzbar:

- Comma-seperated values (CSV)
- JavaScript Object Notation (JSON)
- Extensible Markup Language (XML)

2.4 Künstliche neuronale Netze

Künstliche neuronale Netze gewinnen immer mehr an Bedeutung und sind mittlerweile ein wichtiger Bestandteil des alltäglichen Lebens. Der Begriff „Künstliche Intelligenz“ wird dabei oft synonym für künstliche neuronale Netze verwendet. Diese künstlichen Netze sind dem Aufbau und der Funktionalität des menschlichen Gehirns nachempfunden. Es existieren unterschiedliche Arten von künstlichen neuronalen Netzen, welche für unterschiedliche Problemstellungen geeignet sind. Künstliche Intelligenz ist dabei zusammengefasst: „der Versuch, normalerweise von Menschen erledigte geistige Aufgaben automatisiert zu lösen“ [2, S. 22].

2.4.1 Einsatzgebiete

Künstliche Intelligenz findet sich mittlerweile in einigen Anwendungsbereichen wieder. Der wohl bekannteste Einsatz dürfte dabei im Bereich der intelligenten Sprachassistenzsysteme liegen, wie zum Beispiel Amazon Alexa, Google Assistant oder Samsung Bixby. Weitere Einsatzgebiete umfassen zum Beispiel [basierend auf: 2, S. 32]:

- Klassifizierung von Bildern (oder auch von Gesichtern z.B. bei Fahndungen)
- Verbesserungen an Übersetzungen von Fremdsprachen (z.B. DeepL)
- Autonomes Fahren
- Verbesserungen von Suchanfragen im Internet

2.4.2 Modelle

Bei künstlichen neuronalen Netzen muss generell zwischen verschiedenen Modellen unterschieden werden. Jedes Modell ist dabei für verschiedene Anwendungsgebiete geeignet.

- **Feed Forward Neural Network:** Diese Netze zeichnen sich dadurch aus, dass die Informationen nur in einer Richtung durch das neuronale Netz durchgereicht

werden. Ein Convolutional Neural Network (CNN), wie es im Laufe dieser Arbeit trainiert und benutzt werden soll, ist dabei ein Feed Forward Neural Network.

- **Recurrent Neural Network:** Diese Netze haben im Gegensatz zum Feed Forward-Netz die Möglichkeit, Informationen in vorherigen Schichten „zurückzuschicken". Rekurrente Netze sind dabei immer von Bedeutung, wenn der Zustand einer Informationseingabe wichtig ist. Beispielhaft wäre hier das Lesen eines Satzes: Während des Lesens wird jedes Wort einzeln verarbeitet, jedoch bleiben die zuvor gelesenen Wörter im Gedächtnis. Die vorherigen Wörter sind die Zustände, welche ein rekurrentes Netz erfassen und speichern kann [2, S. 252].

2.4.3 Bedeutung von Trainingsdaten

Die Trainingsdaten sind ein wichtiger Bestandteil des maschinellen Lernens. Ein neuronales Netz kann immer nur auf Basis der Trainingsdaten agieren. Wenn die Datensätze im Training also eher ungeeignet sind, wird das Netzwerk für die vorgesehene Aufgabe vermutlich schlechter abschneiden als gewünscht. Ein Beispiel ist die Erkennung von Kleidungsstücken durch ein CNN. Die Trainingsdaten in Form von Bildern der Kleidungsstücke sind jedoch von schlechter Qualität. Dies kann dazu führen, dass das Netz keine richtigen Strukturen und Unterschiede zwischen den verschiedenen Bildern feststellen kann und somit Kleidungsstücke mit einer niedrigeren Konfidenz (Wert, der anzeigt, wie „sicher" sich das Netz ist, die Kategorie richtig einsortiert zu haben [6, S. 218]) in eine Kategorie (z.B. Schuhe werden als Hut klassifiziert) einsortiert. Außerdem sollten die Daten eine gewisse Varianz abdecken. Wenn ein CNN zum Beispiel Tiere klassifizieren soll, ist es von Vorteil, wenn verschieden aussehende Tiere der gleichen Gruppierung in das neuronale Netz gegeben werden. Eine schlechte Vorgehensweise ist zum Beispiel, einem neuronalen Netz für die Kategorie „Hund" ausschließlich Labradore mit einem dunklen Fell zu zeigen. Hier sind andere Hunderassen wie Shelties, Border Collies sowie verschiedene Fellfarben für eine Verbesserung des Netzes unerlässlich.

2.4.4 Zuverlässigkeit

Die Zuverlässigkeit von künstlichen neuronalen Netzen ist, wie oben bereits erwähnt, abhängig von dem Umfang und der Qualität der Trainingsdaten. Ein Netz, welches mit einem zu kleinen Umfang an Trainingsdaten gefüttert wurde, schneidet unzuverlässig ab. Das Gleiche gilt, wenn sich die Trainingsdaten sehr ähnlich sind und das Netzwerk nicht zwischen Kategorien unterscheiden kann. Auch wenn künstliche Intelligenz im Bereich des autonomen Fahrens (z.B. beim Tesla Autopilot) zum Einsatz kommt, muss darauf hingewiesen werden, dass die Zuverlässigkeit nicht hoch genug ist, sodass der Fahrer seine Aufmerksamkeit von der Straße nehmen könnte. Im Fall der Tabellenerkennung sollten die Ergebnisse im Anschluss manuell geprüft oder auch ausgebessert

werden, falls das neuronale Netz nicht dazu in der Lage ist, die Tabellen mit einer ausreichend hohen Zuverlässigkeit zu erkennen.

Die Zuverlässigkeit eines neuronalen Netzes kann während des Trainings- und Validierungsprozesses mit Hilfe des Verlustes oder auch mit der Genauigkeitsmetrik ausgelesen und beurteilt werden.

2.5 Verwendete Softwarewerkzeuge und Bibliotheken

Für die Vorbereitung der Trainingsdaten und das Entwerfen des neuronalen Netzes werden verschiedene Softwarewerkzeuge und Programmbibliotheken verwendet, welche in den folgenden Kapiteln kurz erläutert werden sollen.

2.5.1 Python

Im Bereich Data Science ist Python eine der beliebtesten Programmiersprachen [7, S. 22]. Für die Programmierung wird Python ausgewählt, da der Autor mit der Programmiersprache vertraut ist und Python durch eine Vielzahl von Programmbibliotheken erweitert werden kann. Zum Einsatz kommt Python in der Version v.3.10.

2.5.2 PyCharm

Als integrierte Entwicklungsumgebung wird PyCharm genutzt, da bestehende Bibliotheken sehr leicht über einen eigenen Reiter gesucht und installiert werden können.

2.5.3 Allgemeine Python Bibliotheken

Das Projekt integriert einige „Standardbibliotheken“, welche eine allgemeine Funktionalität aufweisen, welche keine genauere Erklärung bedürfen, weshalb diese nur mit einer kurzen Beschreibung aufgelistet werden:

- **re**: Support für reguläre Ausdrücke (z.B. genutzt beim Pattern-Matching)
- **os:** Support für die Interaktion mit dem Betriebssystem (z.B. Dateien anlegen)
- **pdf2image:** Umwandlung von PDF in einzelne Bilder
- **json:** Support für das Lesen/Schreiben von JSON
- **PIL:** Verarbeitung von Bilddateien in Python
- **pandas:** Datenanalyse und Manipulation, verwendet für Dataframes
- **pyplot:** Zeichnen von Vorhersagen des Netzes
- **cv2 (OpenCV):** Erstellung der Masken mit logischen Verknüpfungen

2.5.4 TensorFlow

TensorFlow (eingesetzte Version v.2.10.0): ist ein Framework für mathematische Funktionen im Bereich des Machine Learnings. Keras, welches nachfolgend vorgestellt wird, nutzt TensorFlow als Back-End, um Berechnungen durchzuführen [6, S. 159]. TensorFlow unterstützt die Berechnung durch einen oder mehrere Grafikprozessoren

(Graphical Processing Unit – GPU), welche den Lernprozess deutlich beschleunigen, da Grafikkarten gut darin sind, Aufgaben parallelisiert abzuarbeiten [7, S. 452].

Merkmale von TensorFlow umfassen [8]:

- Effiziente Berechnung von low-level Tensoren Operationen auf central processing unit (CPU), GPU oder tensor processing unit (TPU)
- Berechnung des Gradienten beliebiger differenzierbarer Ausdrücke
- Skalierung von Berechnungen auf viele Geräte, z. B. Cluster mit Hunderten von GPUs
- Export von Programmen ("Graphs") zu externen Laufzeiten wie Servern, Browsern, mobilen und eingebetteten Geräten

Um einen Vergleich der theoretischen Rechenleistung aufzustellen, wird in der folgenden Tabelle jeweils eine CPU und eine GPU für Endkonsumenten aus dem High End Bereich (Stand Dezember 2022) miteinander verglichen:

Merkmal	CPU: AMD Ryzen Threadripper 3990X	GPU: NVIDIA GeForce RTX 4090
Taktfrequenz	2,9GHz - 4,3GHz (Boost)	2,2GHz – 2,5GHz (Boost)
Gleitkommaoperationen/s (FP32)	13,2096 (TFLOPS)	82,58 (TFLOPS)
Preis (Launch – MSRP $)	3990 $	1599 $

Tabelle 1: Leistungsmerkmale und Preis aktueller High End CPU/GPU, basierend auf: Raschka, Mirjalili [1], Leistungsdaten von: techpowerup.com[2]

Aus der Tabelle lässt sich entnehmen, dass die Grafikkarte (RTX 4090) eine um Faktor 6,25 höhere theoretische Rechenleistung im Bereich FP32 (Floating-Point 32-Bit, „einfache Genauigkeit“) vorweisen kann. Dabei kostet die Karte selbst nur 40 % des Preises des Threadripper Prozessors.

Im praktischen Teil der Arbeit werden die neuronalen Netze auf einem System mit folgenden Spezifikationen trainiert:

- Prozessor: AMD Ryzen 7 5800X3D
- Grafikkarte: NVIDIA GeForce RTX 3080, 10 GB GDDR6X VRAM (29,77 TFLOPS FP32)
- Arbeitsspeicher: 2x16GB 3600MHz CL16 dual Rank

[1] [7, S. 452].
[2] [9, 10].

2.5.5 Keras

Keras ist, wie im vorangehenden Kapitel bereits erwähnt, eine Schnittstelle, welche sich über TensorFlow legt und dieses als Back-End nutzt. Die in dieser Arbeit eingesetzte Version ist: v.2.10.0.

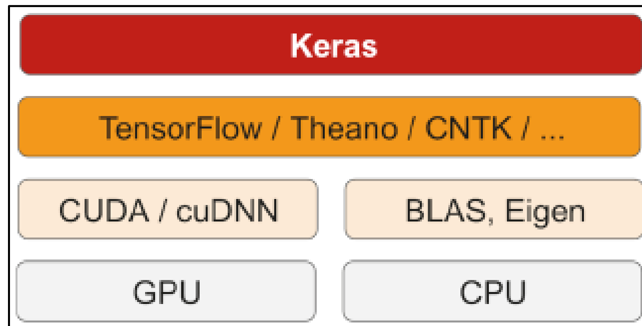


Abbildung 1: Schematischer Aufbau von Keras in Nutzung mit TensorFlow, Quelle: Chollet[3]

Abbildung 1 zeigt den schematischen Bezug von Hardware und Software. Keras legt sich als High-Level-API über TensorFlow, welches als Back-End dient. Darunter wird entweder cuDNN (Bibliothek, um NVIDIAs CUDA Kerne für die Beschleunigung verwenden zu können) genutzt, um die Grafikkarte anzusprechen oder BLAS (Basic Linear Algebra Subprograms) / Eigen, um die Berechnungen auf der CPU auszuführen.

2.5.6 cuDNN

cuDNN (Cuda Deep Neural Network Library) ist eine Bibliothek von NVIDIA, welche es ermöglicht, Berechnungen eines neuronalen Netzes durch die GPU zu beschleunigen. Unter anderem unterstützte Deep Learning Frameworks sind MATLAB, Keras, PyTorch und TensorFlow [11, S. 1]. Die verwendete Version von cuDNN ist v.8.3 und CUDA v.11.7.

2.5.7 Pytesseract

Die Bibliothek Pytesseract (Python-tesseract) ist ein Wrapper für die Tesseract Engine, welche eine Zeichenerkennung (optical character recognition, bzw.: OCR) zur Verfügung stellt. Diese kann unter anderem dazu genutzt werden, um handschriftliche Notizen zu digitalisieren. Tesseract selbst ist ursprünglich eine Entwicklung von HP [12, S. 1]. Für die Bibliothek sind mehrere Sprachen der Erkennung möglich [13]. Eingesetzte Version: v.0.3.10.

2.5.8 NumPy

NumPy ist eine vielfach eingesetzte Bibliothek, welche z.B. Arrays und eine umfassende Funktionalität im Bereich der linearen Algebra zur Verfügung stellt. Diese Funktionen sind in NumPy in der Programmiersprache C geschrieben, was in einer effizienten und

---

[3] [2, S. 91].

schnellen Ausführung des Codes resultiert, da anders als bei Python, keine Umwandlung durch einen Interpreter notwendig ist [7, S. 42]. Das schnelle Abarbeiten von in Arrays gespeicherten Informationen, ist im Rahmen dieser Arbeit, wie später in Kapitel 3.1 beschrieben, unerlässlich. Eingesetzte Version: v.1.23.3.

2.5.9 Hasty.ai

Hasty.ai ist eine Onlineplattform für das Labeln von Datensätzen. Ein Label ist dabei im einfachsten Fall eine Box, welche eine Information in einem Bild umfasst (zum Beispiel eine Tabelle oder Spalte). Der Vorteil gegenüber lokal installierten Programmen wie LabelImg oder LabelMe liegt auf der Hand: Datensätze können von einem PC in einen Workspace auf Hasty.ai hochgeladen und auf einem anderen Rechner abgerufen werden. Dadurch können auch ganze Teams an diesen Datensätzen arbeiten, da die Label in Echtzeit mit dem Workspace und somit auch bei den anderen Teammitgliedern aktualisiert werden. Anschließend können die Label-Informationen in weitverbreiteten Datenformaten exportiert werden, wie z.B. Pascal VOC oder COCO. Für die Arbeit wurde das COCO-Label / Datenformat gewählt.

Abbildung 2: Label für Tabelle und Spalte werden in hasty.ai erstellt, Quelle: eigene Abbildung

In Abbildung 2 ist beispielhaft eine Tabelle zu sehen, welche bereits entsprechende Label für „Tabelle“ (grün) und „Spalte“ (braun) erhalten hat, welche auch mit der zugewiesenen Farbe auf der rechten Seite eingesehen werden können. Die restlichen Label sind in diesem Fall zwar vergeben, aber ausgeblendet (rechts im Auswahlmenü rot

hinterlegt). Rechts lassen sich auch die zu zeichnenden Label auswählen, sowie bestehende Label unsichtbar schalten. Hasty.ai bietet außerdem eine KI-gestützte Label-Funktion an, welche die Merkmale der einzelnen Label lernen kann, um so den Prozess der Vorbereitung des Datensatzes zu beschleunigen. Beim Bearbeiten der Tabellen aus den verschiedenen Datensätzen ist jedoch aufgefallen, dass das KI unterstützte Labeln in diesem Fall unzuverlässig bzw. ungenau ist, da wiederholt falsche Bereiche als Tabelle / Spalte usw. erkannt worden sind.

3. Analyse des Problems

Mit der Digitalisierung der öffentlichen Verwaltung steht in Deutschland ein großer und wichtiger Schritt an, jedoch stehen hier vorerst die aktuellen und zukünftigen Daten im Mittelpunkt (neue Daten werden direkt digital erhoben und können zur Verfügung gestellt werden). Dennoch sind bereits bestehende, nicht-digitale Datensätze ein wichtiger Bestandteil, welche digitalisiert werden müssen. Ein manuelles Digitalisieren nimmt viel Zeit in Anspruch und kann eventuell durch ein künstliches neuronales Netz (teil-)automatisiert werden.

3.1 Auswahl eines Modelltyps für das neuronale Netz

Wie in Kapitel 1.0 bereits erwähnt, soll das Netzwerk dazu in der Lage sein, aus einem Dokument (in Bildform) Tabellen zu erkennen und für diese eine Maske zu generieren, womit Informationen wie Grafiken oder Fließtexte abgedeckt werden können, damit eine Texterkennung nur die relevanten Daten aus einer Tabelle auswertet. Das passende Netzwerk für diese Aufgabe ist das Convolutional Neural Network Modell, welches im Bereich Computer Vision zum Einsatz kommt.

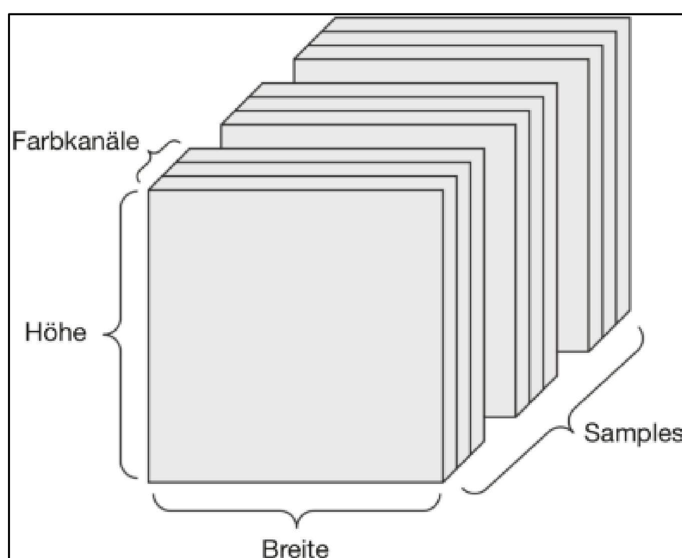


Abbildung 3: Samples von Bildern. Unterteilt in Farbkanäle, Quelle: Chollet[4]

[4] [2, S. 61].

Wie in Abbildung 3 zu sehen, wird ein bestehendes Bild in ein dreidimensionales Array abgelegt. Die Dimensionen setzen sich dabei aus der Größe des Bildes in Pixeln, sowie die Anzahl der Farbkanäle zusammen. Ein Bild mit der Größe 1920x1080 in RGB würde als Array beispielsweise die Form (1920, 1080, 3) annehmen, da das Bild auf die drei Farbkanäle Rot, Grün und Blau aufgeteilt und in eigene Arrays abgelegt wird. Das CNN arbeitet die Bilder in sogenannten Samples ab, welche eine Teilmenge des Datensatzes darstellen können [2, S. 61]. Ein Datensatz mit 100 Farbbildern bei einer Auflösung von 1920x1080 würde dementsprechend die Form (100, 1920, 1080, 3) annehmen. Das Spezielle an einem CNN ist der Schritt der Convolution (zu Deutsch: Faltung). Das Ziel eines solchen Faltungs-Layers in Bezug auf Computer Vision ist es, Eigenschaften aus einer bestimmten Anzahl von Pixeln auf einem einzelnen Pixel abzubilden. Dabei wird angenommen, dass nahe beieinanderliegende Pixel im Gegensatz zu weit entfernten, füreinander von hoher Bedeutung sind. Der Vorteil dabei ist, dass sich dadurch die Anzahl der Gewichte im Netz selbst deutlich reduzieren lassen [6, S. 68, 7, S. 551]. Gewichte sind vereinfacht gesagt die Faktoren, welche eine vorhandene Eingabe parametrisieren [6, S. 20]. Wenn vier Neuronen auf ein einzelnes Neuron abgebildet werden, entscheiden die Gewichte, zu welchem Anteil das jeweilige Eingangsneuron einen Einfluss auf das Ausgangsneuron hat.

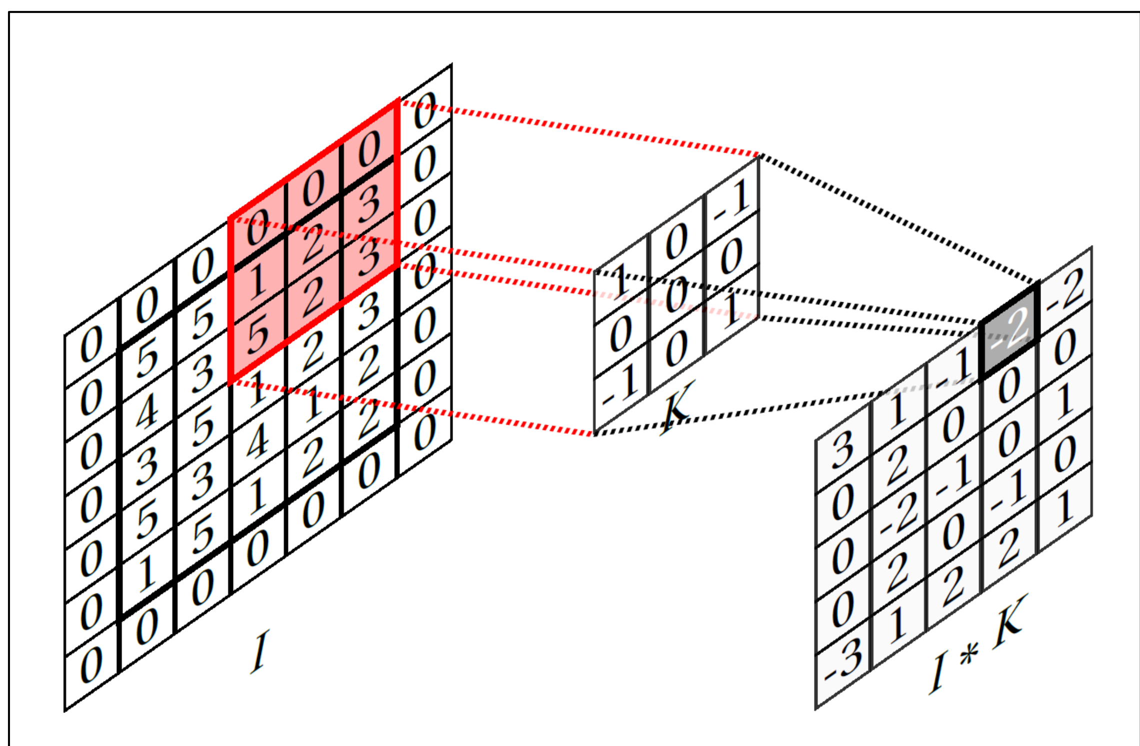


Abbildung 4: Funktionsweise eines Kernels in einem CNN, Quelle: Frochte[5]

[5] [14, S. 366].

Abbildung 4 zeigt die Arbeitsweise eines 3x3 Kernels auf einer 5x5 Eingabe, welche durch Nullen (Zero-Padding) auf 7x7 vergrößert worden ist. Ohne Zero-Padding würde der Kernel die Ausgabe verkleinern, da sich die Mitte des Kernels im ersten Durchlauf bereits in der zweiten Zeile und Spalte der Eingabe befinden würde. Die Ausgabe ergibt sich hier durch:

$$Output = Input\ (I) * Kernel\ (K)$$

Der Kernel bewegt sich über die Pixel des Bildes hinweg und multipliziert die Werte der einzelnen Pixel mit den hinterlegten Werten und addiert diese, um eine Reduktion auf einen Pixel zu erzielen. In Abbildung 4 wäre die vollständige Rechnung für den angegebenen Ausschnitt also:

$$0 * 1 + 0 * 0 + 0 * (-1) + 1 * 0 + 2 * 0 + 3 * 0 + 5 * (-1) + 2 * 0 + 3 * 1 = -2$$

Außerdem können in CNNs sogenannte Pooling-Layer eingesetzt werden, welche dazu dienen, die Größe von Merkmalen weiter zu reduzieren, um eine effizientere Berechnung zu ermöglichen. Der Einsatz dieser Layer ist aber nicht zwingend erforderlich für ein CNN [7, S. 562], da die Funktionsweise der oben beschriebenen Faltungsschicht in der Theorie erlaubt, eine Reduzierung der Auflösung innerhalb des Netzes vorzunehmen. In der Praxis ist es aber zu empfehlen, dass die Faltungsschichten eine mit Hilfe von Padding (siehe Abbildung 4) vergrößerte Eingabe erhalten, um die Eingangsauflösung der Schicht beizubehalten. Anschließend kann die Auflösung mit Hilfe der Pooling-Layer reduziert werden [7, S. 556]. Als Beispiel für ein neuronales Netz mit Pooling-Layern können die VGG-Netze gesehen werden. Aus diesen Pooling-Layern werden speziell bei den VGG-Netzen die Masken für die Tabellenerkennung berechnet, wie später in Kapitel 4.1 noch erläutert wird.

3.2 Vorbereiten geeigneter Datensätze für das neuronale Netz

Für das Training des neuronalen Netzes werden drei Datensätze verwendet, welche durch OpenData-Plattformen der Länder abgerufen werden können:

- Bodenarten der Böden Deutschlands (2006-2007, 43 Seiten), Link: *https://www.bgr.bund.de/DE/Themen/Boden/Produkte/Schriften/Downloads/Bodenarten_Bericht.pdf*, zuletzt abgerufen am 23.01.2023
- Hintergrundwerte stofflich gering beeinflusster Böden Schleswig-Holsteins (Dezember 2011, 85 Seiten), Link: *https://umweltportal.schleswig-holstein.de/trefferanzeige?docuuid=0cce542d-6828-41c8-8a89-1ec621440436*, zuletzt abgerufen am 23.01.2023

- Luftverunreinigungen in Berlin (Januar-Dezember 2021, 129 Seiten), Link: *https://www.berlin.de/sen/uvk/umwelt/luft/luftqualitaet/luftdaten-archiv/*, zuletzt abgerufen am 23.01.2023

Wichtig ist zu erwähnen, dass nur die Dokumentenseiten mit Tabellen für das Training genutzt werden. Dies resultiert in einem insgesamt kleineren Trainingsdatensatz und dadurch in einer kürzeren Berechnungszeit. Außerdem bringt das Hinzufügen von negativen Beispielen für die Ergebnisse des neuronalen Netzes keinen Mehrwert, da das neuronale Netz aus den Pixeln, welche nicht durch ein Label gekennzeichnet sind, genügend „negativen" Input bekommt [15, S. 3]. Am Beispiel der oben genannten Daten reduziert sich die Anzahl der Seiten (Bilder) durch diese Voraussetzung von 257 auf 128.

Der Quellcode ist verfügbar unter: *https://github.com/lutzgooren/tableDetection*

Die Daten werden in PDF-Form im Ordner „PDF_In" im Projektorder abgelegt. Anschließend wird das Skript ***pdfToJPG.py*** (Anlage 1) ausgeführt, wodurch die einzelnen Seiten im „PDF_Out" als .jpg-Datei abgelegt werden.

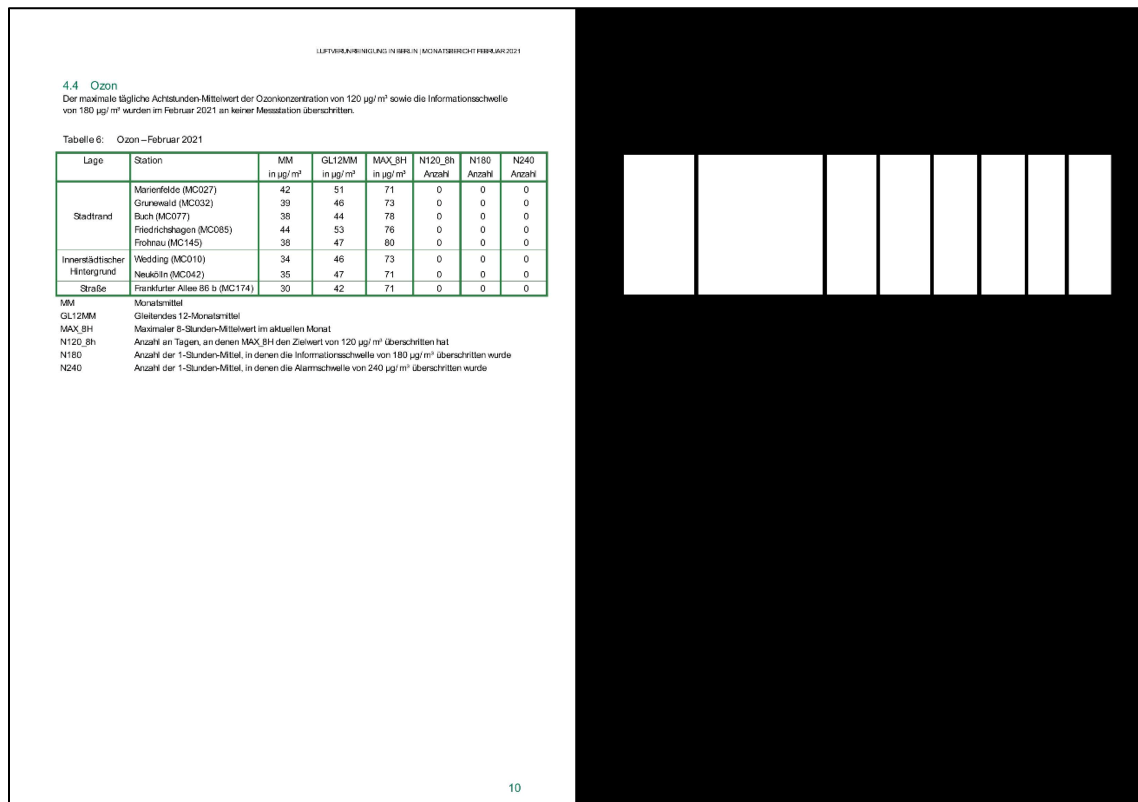
In Hasty werden diese Daten eingespielt und mit Labeln versehen. Für die Arbeit wurden für folgende Merkmale Label angelegt:

- Überschrift
- Tabelle
- Tabellenkopf
- Vorspalte
- Felder
- Spalte

Die Label für die Tabellenüberschriften wurden im Verlauf der Arbeit aufgrund einer schlechten Aufwand- und Nutzenbilanz nicht verwendet. Die ursprüngliche Idee, die Tabellen mit der Überschrift als Dateinamen zu speichern, wird deshalb verworfen.

Für die Vorbereitung des Datensatzes wird ein Python-Skript ***preProcess.py*** (Anlage 2) geschrieben, welches die Koordinaten der Label-Umrandung ausliest, um daraus eine neue .jpg-Datei zu erstellen, in welcher die Label als Maske fungieren. Die Masken werden nach Kategorie generiert und in eigenen Ordnern gespeichert. Die Tabellenmasken selbst liegen dabei z.B. im Arbeitsverzeichnis „./stuff/masked/table" und die Spaltenmasken in „./stuff/masked/columns". Der Dateipfad des Bildes wird inklusive der Dateipfade der verschiedenen Masken in eine .csv-Datei geschrieben. Diese Datei kann später von

TensorFlow gelesen und in einen entsprechenden Dataframe (2D Tabellendaten) importiert werden, um damit ein neuronales Netz zu trainieren.



LUFTVERUNREINIGUNG IN BERLIN | MONATSBERICHT FEBRUAR 2021

4.4 Ozon

Der maximale tägliche Achtstunden-Mittelwert der Ozonkonzentration von 120 µg/m³ sowie die Informationsschwelle von 180 µg/m³ wurden im Februar 2021 an keiner Messstation überschritten.

Tabelle 6: Ozon – Februar 2021

Lage	Station	MM in µg/m³	GL12MM in µg/m³	MAX_8H in µg/m³	N120_8h Anzahl	N180 Anzahl	N240 Anzahl
Stadtrand	Marienfelde (MC027)	42	51	71	0	0	0
	Grunewald (MC032)	39	46	73	0	0	0
	Buch (MC077)	38	44	78	0	0	0
	Friedrichshagen (MC085)	44	53	76	0	0	0
	Frohnau (MC145)	38	47	80	0	0	0
Innerstädtischer Hintergrund	Wedding (MC010)	34	46	73	0	0	0
	Neukölln (MC042)	35	47	71	0	0	0
Straße	Frankfurter Allee 86 b (MC174)	30	42	71	0	0	0

MM Monatsmittel
GL12MM Gleitendes 12-Monatsmittel
MAX_8H Maximaler 8-Stunden-Mittelwert im aktuellen Monat
N120_8h Anzahl an Tagen, an denen MAX_8H den Zielwert von 120 µg/m³ überschritten hat
N180 Anzahl der 1-Stunden-Mittel, in denen die Informationsschwelle von 180 µg/m³ überschritten wurde
N240 Anzahl der 1-Stunden-Mittel, in denen die Alarmschwelle von 240 µg/m³ überschritten wurde

10

Abbildung 5: Input (links) und generierte Maske für Spalten aus Koordinaten der Label (rechts), Quelle: eigene Abbildung

Abbildung 5 zeigt eine Gegenüberstellung eines Bildes aus dem Datensatz und die entsprechende, durch das Skript aus den Koordinaten generierte Maske für die einzelnen Spalten der Tabelle. Andere Masken werden entsprechend den unterschiedlichen Labeln generiert und in einem eigenen Ordner abgelegt.

3.3 Manuelles digitalisieren der Daten

In der öffentlichen Verwaltung, als auch in Unternehmen der freien Marktwirtschaft, existieren Datensätze, welche nicht digitalisiert sind. Als Beispiel führen viele Unternehmen ein Unternehmensarchiv, in welchem Dokumentationen für alte Produkte (Anleitungen, Reparaturanweisungen, technische Spezifikationen etc.) teilweise noch in Papierform aufgehoben werden. Diese Daten zu digitalisieren wäre durch das einmalige Einscannen gelöst, wenn nicht noch zusätzliche Anforderungen an die Form des Datensatzes gestellt werden, wie es bei der in Kapitel 2.3 erwähnten DIN-Norm der Fall ist.

Wie in Kapitel 1.1 beschrieben, sind über 80 % der Daten in der öffentlichen Verwaltung nicht digital verfügbar. Darunter befinden sich unter Umständen wichtige Datensätze, welche nur mit einem erheblichen manuellen Aufwand digitalisiert und bereitgestellt

werden können. Im Falle der Tabellen für die Luftqualität würde ein manuelles Digitalisieren in eine .csv-Datei lange brauchen.

4. Ergebnisse und Diskussion

Das folgende Kapitel dient zur Bewertung der trainierten Netze. Zunächst wird aus einer größeren Auswahl von Netzen das am besten abschneidende Netz ausgewählt und die Parameter dieses Netzes verändert, um eine Verbesserung der Ergebnisse zu erzielen.

4.1 Aufbau des neuronalen Netzes

Das zu trainierende neuronale Netz wird auf der Basis verschiedener, bereits bestehender Netzarchitekturen aufgebaut. Die folgende Liste umfasst die verwendeten Netzarchitekturen, die gewählte Eingangsauflösung der Bilder sowie die Tiefe der Netze, welche die Anzahl der Schichten darstellt („Die Anzahl der zu einem Datenmodell beitragenden Layer“ [2, S. 27]):

Name Architektur	Eingangsauflösung	Tiefe
VGG16	1024x1024	16
VGG19	1024x1024	19
ResNet50	768x768	107
Xception	768x768	81
NASNetLarge	512x512	533
EfficientNetB5	512x512	312
DenseNet201	512x512	402

Tabelle 2: Verwendete Architekturen, Eingangsauflösungen der Bilder sowie die Tiefe der Netze, Quelle: eigene Tabelle

Wie in Kapitel 2.5.4 bereits erwähnt, wurde das Training mit Hilfe einer RTX 3080 in der 10 GB Variante beschleunigt. Die unterschiedlichen Auflösungen der Netze kommen dadurch zustande, dass die Netzarchitekturen unterschiedlich komplex sind und dementsprechend mehr Speicher auf der GPU benötigen. Für die Berechnung wurde versucht, den Speicher der GPU voll auszureizen, um eine möglichst hohe Eingangsauflösung zu erzielen.

Zunächst wurden die Datensätze so mit Labeln versehen, dass die Tabelle in einzelne Bestandteile zerlegt wurde. Demzufolge gab es ein Label für die Kopfzeile, Vorspalte und die Felder. Das Ziel war am Ende alle drei Masken für eine Eingabe zu kombinieren, um so eine Maske für die komplette Tabelle zu erhalten.

Abbildung 6: Label für Kopfzeile, Vorspalte und Felder werden in hasty.ai erstellt, Quelle: eigene Abbildung

Abbildung 6 zeigt die Label, welche die Tabelle in Einzelteile zerlegen (Lila: Tabellenkopf, Gelb: Vorspalte, Orange: Felder). Wie in Kapitel 3.2 in Abbildung 5 zu sehen, werden entsprechend auch für die Label aus Abbildung 6 Masken durch das preProcess-Skript generiert. Die kombinierte Maske wird am Ende durch eine logische ODER-Verknüpfung zusammengesetzt.

Um später untersuchen zu können, ob eine andere Vorgehensweise bessere Ergebnisse erzielen würde, wurden auch Label für die Tabelle und die einzelnen Spalten (siehe Abbildung 2) angelegt, um diese anschließend mit einer logischen UND-Verknüpfung zur gewünschten Maske zusammenzusetzen.

Alle Netzarchitekturen wurden mit vorher trainierten Gewichten des ImageNet erstellt. Diese Gewichte können mit dem Argument weights=‘imagenet‘ beim Erstellen des Netzwerkes gewählt werden. ImageNet ist eine verfügbare Datenbank von Bildern, mit deren Hilfe die verschiedenen Netzarchitekturen trainiert werden – auch, um die Genauigkeit der einzelnen Netze zu bestimmen (Benchmark). Die Datenbank umfasst 1000 Objektklassen, 1.281.167 Trainingsbilder, 50.000 Validierungsbilder sowie 100.000 Testbilder [16].

Für den Aufbau des neuronalen Netzes wird in dieser Arbeit auf TableNet, einer bereits entworfenen Netzarchitektur aufgebaut, um Vorhersagen für die einzelnen Masken treffen zu können.

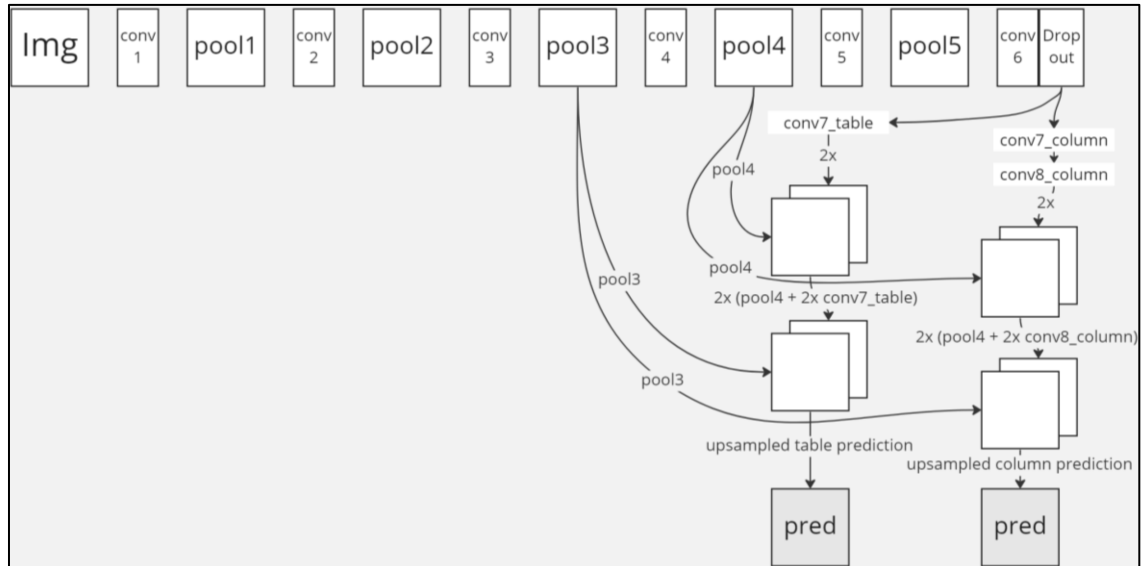


Abbildung 7: Aufbau des TableNet unter Verwendung der VGG19 Architektur, Quelle: basierend auf: Paliwal, D, et al[6]

Abbildung 7 zeigt den Aufbau des neuronalen Netzes (TableNet) mit der VGG19 Architektur. Das Netz besitzt jeweils eine Ausgabe für die Tabellen- sowie die Spaltenmaske, welche vor der Ausgabe auf die Eingangsauflösung hochskaliert wird. Die Vorhersagen der Masken werden aus verschiedenen Schichten des VGG19 Netzwerkes zusammengesetzt. In diesem Fall werden die Schichten *pool3, pool4 und pool5* genutzt. Für die anderen bereits in Tabelle 2 erwähnten Netzarchitekturen werden die entsprechend passenden Schichten für die Vorhersage der Masken gewählt. Die Erkennung der verschiedenen Merkmale einer Tabelle können als alleinstehende Probleme betrachtet werden, welche durch die dedizierten Ausgänge des Netzes unabhängig voneinander gelöst werden können [17, S. 2]. Für das erste Modell wird die TableNet Architektur um einen dritten Output erweitert, sodass das Netz eine Maske für die Kopfzeile, Vorspalte sowie Felder ausgeben kann, welche anschließend durch eine logische ODER-Verknüpfung die komplette Maske der Tabelle bilden. Dies geschieht mit dem Skript: ***trainModel_head_front_field.py*** (Anlage 3) bzw. ***trainModel_table_col.py*** (Anlage 4).

4.2 Ergebnisse der trainierten Modelle

Wie in Tabelle 2 beschrieben, wurden zunächst sieben verschiedene Netze mit den Daten bei höchstmöglicher Eingangsauflösung trainiert. Anhand einer Verlustmetrik während der Validierungsphase, lässt sich mit Hilfe von TensorBoard die Performance der verschiedenen Netze einschätzen. Außerdem ist auch eine visuelle Prüfung der

[6] [basierend auf: 17, S. 3].

generierten Masken des neuronalen Netzes hilfreich. Die Modelle wurden alle in 200 Epochs traniert (eine Epoch entspricht einem ganzen Durchlauf des kompletten Trainingsdatensatzes) und so konfiguriert, dass bei jeder Verbesserung der Verlustmetrik des Validierungsdatensatz ein entsprechender Checkpoint auf die Festplatte geschrieben wurde. Die Verlustmetrik gibt an, wie weit die berechnete Ausgabe vom zu erwartenden Wert abweicht [2, S. 30]. Bei den trainierten Modellen wäre dies z.B. die Abweichung zwischen der berechneten Maske für eine Tabelle und der zuvor mit Labeln gekennzeichnete Maske des Trainings- bzw. Validierungsdatensatzes. Ein Checkpoint sichert das komplette Netz inklusive der angepassten (trainierten) Gewichte. Diese können später für eine Vorhersage oder für ein weiteres Training abgerufen werden.

Anhand folgender Tabelle lassen sich die minimalen Werte und die dazugehörige Epoch, in welcher die Verlustmetrik minimal gewesen ist, der verschiedenen Netzarchitekturen in Bezug auf den Validierungsdatensatz ablesen:

Name der Architektur	Epoch	Minimaler Verlust (val_loss)
VGG19	177/200	0,0219
VGG16	38/200	0,0336
ResNet50	127/200	0,0382
Xception	74/200	0,0438
DenseNet201	137/200	0,0719
EfficientNetB5	91/200	0,0720
NASNetLarge	141/200	0,1234

Tabelle 3: Netzarchitekturen, Epochs und Verlustmetrik des Validierungsdatendatzes (Kopfzeile, Vorspalte, Felder), aufsteigend sortiert nach Verlust, Quelle: eigene Abbildung

Die Tabelle 3 zeigt recht eindeutig, dass die beiden VGG-Netze am besten abschneiden. Auf Platz 3 landet ResNet50 mit einem Abstand von 0,0046 zu VGG16. Die Ergebnisse spiegeln dabei ungefähr die Eingangsauflösung der Netze wider, da VGG16/19 die höchste Eingangsauflösung besitzen (siehe Tabelle 2 oben).

Um den Berechnungsaufwand für die kommenden Anpassungen kleinzuhalten, werden die Modelle fortan ausschließlich mit der VGG19 Architektur trainiert.

Die Vorhersagen können mit dem Skript ***loadAndPredict_head_front_field.py*** (Anlage 5) bzw. ***loadAndPredict_table_col.py*** (Anlage 6) berechnet werden. Die einzelnen Masken eines Testbildes sieht bei Nutzung von VGG19 folgendermaßen aus:

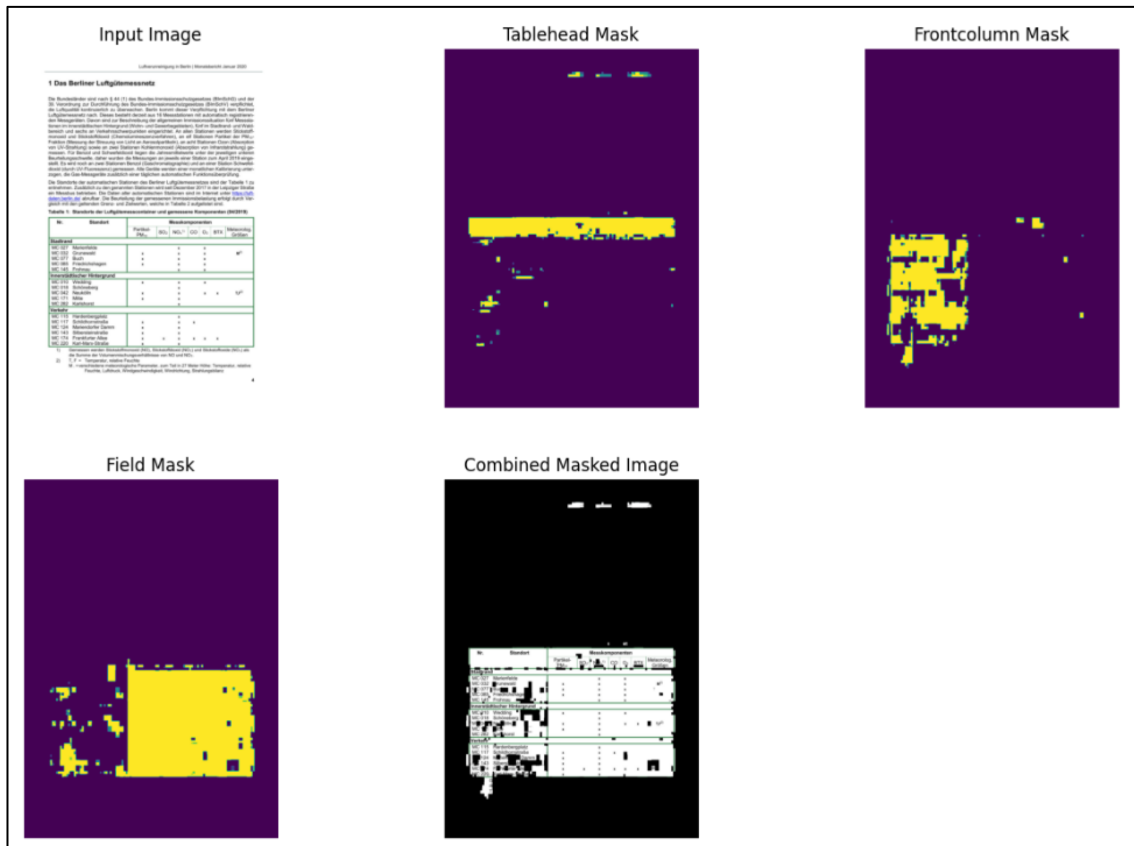


Abbildung 8: Input, Kopfzeilenmaske, Vorspaltenmaske, Feldmaske und zusammengesetzte Maske als Alpha-Kanal auf Input, Quelle: eigene Abbildung

Das Netzwerk generiert die Masken, welche zusammengesetzt als eine Art Alpha-Kanal auf dem originalen Input angewandt werden, um Informationen auszublenden. Aus Abbildung 8 lässt sich entnehmen, dass das Netzwerk bis auf ein paar Ausreißer oben rechts und unten links alle irrelevanten Informationen ausblendet. Innerhalb der Tabelle entstehen jedoch Löcher, welche auch Text betreffen. Dies führt dazu, dass Pytesseract die Wörter nicht richtig erkennen kann.

4.3 Anpassungen für die Erzielung besserer Ergebnisse

Um die Ergebnisse des neuronalen Netzes zu verbessern, werden Vorschläge genannt und einige in Python umgesetzt. Im Folgenden werden beschrieben:

- Vergrößerung des Datensatzes
- Anpassung der Lernparameter
- Weitere Anpassung wie z.B. Post-Processing (Nachbearbeitung)

Anschließend soll ein Vergleich zwischen den Modellen mit geänderten Parametern durchgeführt werden.

4.3.1 Training des Modells in einem größeren Umfang

Eine Möglichkeit für die Verbesserung der Erkennung ist eine Vergrößerung des Trainingsdatensatzes. Hier sollte auch darauf geachtet werden, dass sich die Tabellen nicht zu ähnlich sind, um dem Netzwerk einen großen Umfang von verschieden formatierten Tabellen zu bieten. Eine wesentliche Vergrößerung des Datensatzes ist im Umfang dieser Arbeit nicht möglich, da die Berechnungszeiten für das VGG19 Netzwerk mit einer Eingangsauflösung von 1024x1024 bereits bei über einer Minute pro Epoch liegen und das Labeln der Daten entsprechend Zeit kostet.

4.3.2 Anpassung der Parameter des Lernprozesses oder des Modells

Eine einfache Möglichkeit ist die Anpassung verschiedener Parameter des Modells, um im Anschluss die Ergebnisse zu vergleichen. Die beiden Parameter *learning-rate* und *Epsilon* des Optimizers Adam, lassen einige Anpassungen des Trainingsvorgangs zu. Die *learning-rate* hat direkte Auswirkungen darauf, wie schnell die Verlustfunktion konvergiert und *Epsilon* beeinflusst, wie „zufällig“ (numerische Stabilität) die Änderungen pro Schritt sind [18, S. 2, 19]. Außerdem könnten die Dropout-Layer angepasst werden, welche eine zufällige Anzahl an Bits im Eingang des Netzes während der Trainingsphase auf null setzen. Dies wäre hilfreich, wenn das Netz in das sogenannte Overfitting läuft. Overfitting meint dabei, dass das Netz zu lange mit den Trainingsdaten verbracht hat und Informationen extrahiert, welche nicht der eigentlichen Aufgabe entsprechen [7, S. 568, 14, S. 85-86]. Beispiel: Ein CNN soll Kleidungsstücke in Kategorien einsortieren. Im Trainingsdatensatz sind die Bilder für Sneaker verrauscht (z.B. jpeg-Kompressionsartefakte). Anstatt das Aussehen von Sneakern zu erlernen, fängt das Netz irgendwann an, Bilder mit Artefakten unabhängig vom Inhalt als Sneaker zu erkennen. Eine weitere Änderung, welche im Lernprozess vorgenommen werden kann, um die Ergebnisse zu verbessern, ist die Erhöhung der Eingangsauflösung [20, S. 456]. Zudem werden im derzeit trainierten Modell, für eine Reduzierung der Pixel, rechteckige Bilder auf ein quadratisches Format gestaucht und die Masken nach der Vorhersage auf Originalgröße gestreckt. Wie bereits erwähnt, ist eine Erhöhung der Eingangsauflösung im Verlauf der Arbeit durch die Limitierung des VRAM jedoch nicht möglich. Wichtig ist, dass eine Auflösung gewählt werden muss, die eine Balance zwischen Schnelligkeit und Genauigkeit findet. Zu große Eingangsauflösungen reduzieren die mögliche Batch-Größe und zu kleine Auflösungen verhindern ein Erkennen von Merkmalen [21, S. 1] (Batch-Größe: Anzahl an Bildern, welche die Grafikkarte zeitgleich durch das neuronale Netz schleusen kann).

4.3.3 Weitere Anpassungen

Neben den Anpassungen von Parametern wie *learning-rate* oder *Epsilon,* können auch Anpassungen an den generierten Masken vorgenommen werden (post-process). Durch

ein Vergrößern (Dilation) oder ein Verkleinern (Erosion), mit Hilfe von cv2, können die Masken angepasst werden. Dies ist z.B. hilfreich, wenn die Masken Löcher vorweisen, welche Pytesseract davon abhalten, den Text korrekt zu erkennen. Hier kann eine Vergrößerung helfen, diese Lücken der Maske zu schließen. Dadurch steigt jedoch das Risiko, dass die Masken über die Tabellen hinaus vergrößert werden und so ungewünschte Informationen, welche nicht zu den eigentlichen Tabellen (wie z.B. Fließtext) gehören, durch Pytesseract gelesen und ausgegeben werden.

Abbildung 9: Input, Dilation und Erosion in cv2, Quelle: OpenCV.org[7]

Abbildung 9 zeigt die verschiedenen Operationen durch cv2. Auf der linken Seite ist das originale Bild zu sehen. In der Mitte werden die weißen Pixel mit *Dilate* nach außen hin vergrößert. Im rechten Bild werden die weißen Pixel mit *Erode* nach innen hin geschrumpft.

Eine weitere Überlegung wäre, die Bilder des Trainingsdatensatzes vor dem Training in Graustufen bzw. in Bilder mit hohem Kontrast umzuwandeln (Schwarz / Weiß). Dies kann entweder dazu führen, dass die Farbe einer Tabelle nicht mehr entscheidend ist, um korrekt erkannt zu werden (Beispiel: Tabellen aus Berlin, siehe Abb. 5, haben einen grünen Rand). Auf der anderen Seite kann eine Reduzierung auf Schwarz / Weiß jedoch auch dazu führen, dass zu wenig Informationen verfügbar sind, um eine zuverlässige Erkennung der Tabellen zu erreichen.

4.4 Ergebnisse des Modells mit angepassten Parametern

Der Trainingsvorgang wurde mit folgenden Parametern durchgeführt (Farbkodierung für Grafik in Klammern):

- Learning-Rate: 1x10e-3, Epsilon: 1x10e-8 (Magenta), Standard / keine Änderung
- Learning-Rate: 1x10e-5, Epsilon: 1x10e-8 (Blau)
- Learning-Rate: 1x10e-5, Epsilon: 1x10e-4 (Grün)
- Learning-Rate: 1x10e-6, Epsilon: 1x10e-4 (Rot)

[7] [22].

Aus der folgenden Grafik lassen sich die Ergebnisse der Trainingsdurchläufe für den Verlust des Validierungsdatensatzes ablesen:

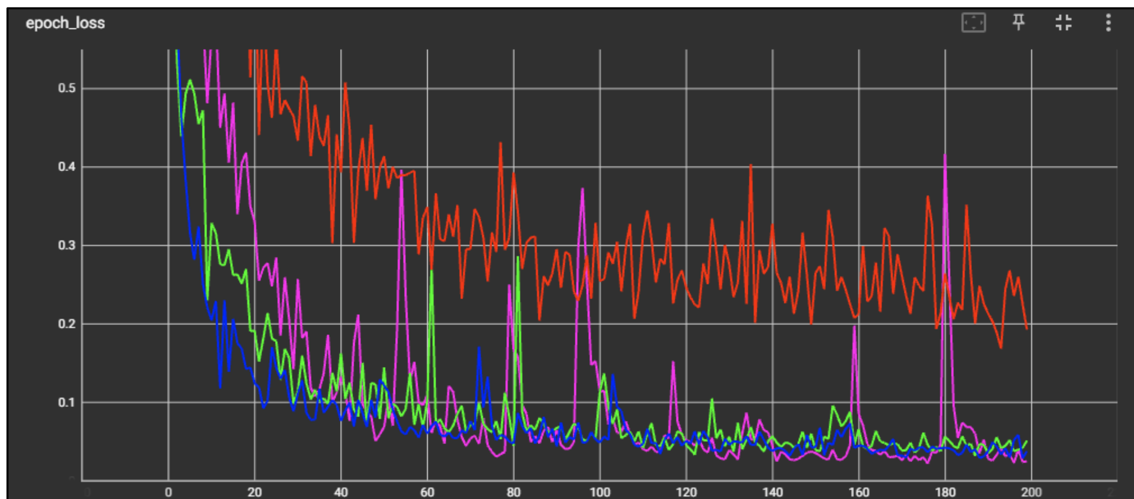


Abbildung 10: y-Achse: Loss (Verlust), x-Achse: Epoch (Durchlauf), Verlauf gilt für Validierung, Quelle: eigene Abbildung

Die langsamste Learning-Rate (rot) und ein kleiner Wert für Epsilon schneidet innerhalb der 200 Epochs des Trainingsvorgangs am schlechtesten ab. Die anderen Einstellungen der beiden Parameter verhalten sich alle ähnlich. Der ursprüngliche Durchlauf (Magenta) ist im Vergleich zu den anderen Durchläufen etwas unruhig, welches sich durch die Spitzen im Verlauf bemerkbar macht, da die höhere Learning-Rate stärkere Änderungen der Verlustmetrik zulässt.

Parametereinstellung (Learning-Rate / Epsilon)	Minimaler Verlust (Validierung)
1x10e-3 / 1x10e-8	0,0219
1x10e-5 / 1x10e-8	0,0286
1x10e-5 / 1x10e-4	0,0316
1x10-e6 / 1x10-e4	0,1684

Tabelle 4: Parametereinstellungen für VGG19 Netz (Kopfzeile / Vorspalte / Felder) und minimaler Verlust (Validierung), Quelle: eigene Tabelle

Aus Tabelle 4 lässt sich entnehmen, dass der Standarddurchlauf (Magenta) den kleinsten Verlust vorweist. Um weitere Verbesserungsoptionen zu untersuchen, wird der Aufbau des Netzes, wie in Kapitel 4.1 erwähnt, auf ein Tabellen- und Spalten-Modell abgeändert.

4.5 Anpassen des Netzes für Tabellen und Spaltenmaske

Wie in Abbildung 2 zu sehen, wurden bereits Label für die Spalten und Tabellen in Hasty.ai erstellt, um ein neuronales Netz zu trainieren, welches eine Maske für die Tabelle und für die einzelnen Spalten ausgibt (letzteres siehe Abbildung 5). Diese beiden

Masken sollen im Anschluss im Gegensatz zum ersten Modell miteinander UND-verknüpft werden. Dadurch entsteht eine kombinierte Maske, welche nur Pixel anzeigt, die durch das neuronale Netz als Tabelle und Spalte erkannt worden sind. Dies soll verhindern, dass das Netz falsche Informationen durch eine ODER-Verknüpfung in die zusammengesetzte Maske überträgt.

Erklärung: Beim ODER-verknüpften Modell werden drei Masken, jeweils für die Kopfzeile, die Vorspalte und die Felder vorhergesagt. Angenommen jede Maske enthält einige Pixel, welche fälschlicherweise als Kopfzeile etc. erkannt worden sind. Eine logische ODER-Verknüpfung würde möglicherweise dann die Anzahl der falsch erkannten Pixel erhöhen. Bei einer logischen UND-Verknüpfung reduziert sich das Risiko, falsche Pixel zu erkennen, da diese Pixel in beiden Masken fälschlicherweise erkannt werden müssen, um die UND-Verknüpfung zu „überleben".

Auch für dieses Netz wurden alle Architekturen getestet, um weitere Anpassungen am besten abschneidenden Netz vorzunehmen. Die Ergebnisse lassen sich aus der folgenden Tabelle entnehmen:

Name der Architektur	Epoch	Minimaler Verlust (val_loss)
VGG19	134/200	0,0322
VGG16	199/200	0,0392
ResNet50	190/200	0,0407
Xception	124/200	0,0583
DenseNet201	133/200	0,0784
EfficientNetB5	110/200	0,0820
NASNetLarge	179/200	0,1577

Tabelle 5: Netzarchitekturen, Epochs und Verlustmetrik des Validierungsdatensatzes (Tabelle, Spalte), aufsteigend sortiert nach Verlust, Quelle: eigene Tabelle

Aus Tabelle 5 lässt sich ablesen, dass auch bei dem Modell mit UND-verknüpfter Maske das VGG19 Netz am besten abschneidet, gefolgt von VGG16 und ResNet50, wobei letzteres mit einer niedrigeren Auflösung trainiert worden ist.

Für weitere Anpassungen wurde wie beim vorherigen Modell das VGG19-Netz ausgewählt, um den Berechnungsaufwand zu reduzieren.

Die Vorhersagen des VGG19-Netzes sehen für das gleiche Bild wie aus Abbildung 8 folgendermaßen aus:

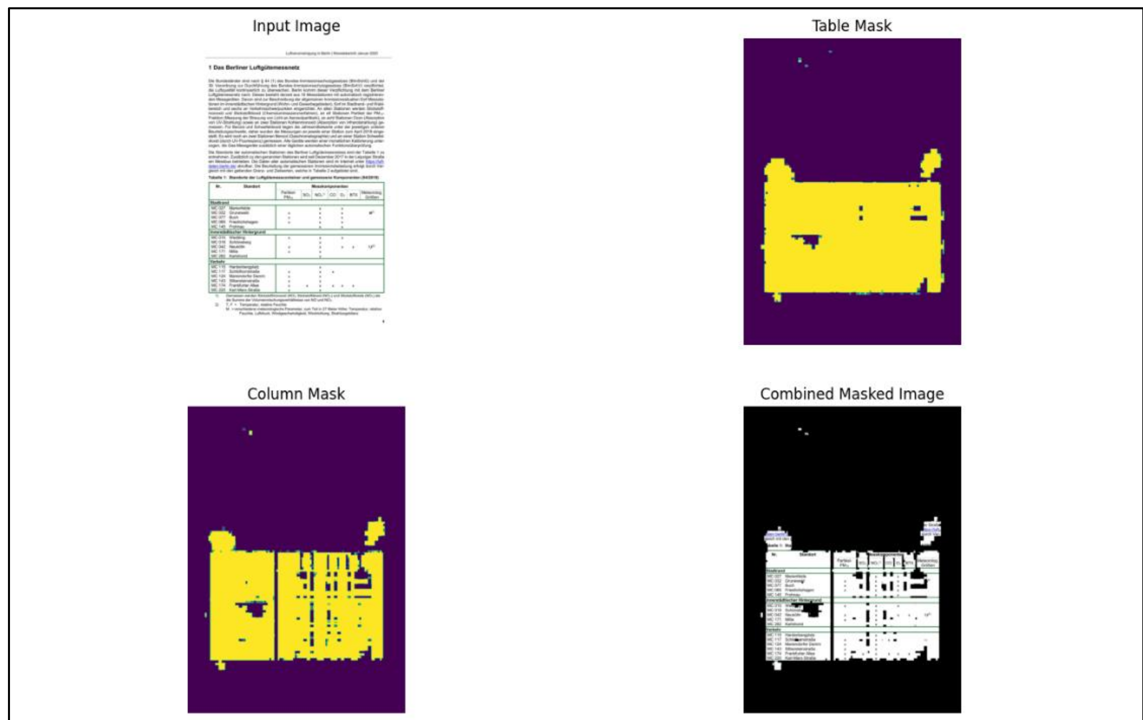


Abbildung 11: Input, Kopfzeilenmaske, Vorspaltenmaske, Feldmaske und zusammengesetzte Maske als Alpha-Kanal auf Input, Quelle: eigene Abbildung

Aus Abbildung 11 lässt sich entnehmen, dass die Spaltenerkennung nur dann wirklich ausreichend gut funktioniert, wenn die Spalten selbst durch einen Strich getrennt sind. Im Vergleich zu Abbildung 8 ist die Vorspalte besser erkannt worden. Dafür erkennt das neue Modell den Fließtext über der Tabelle teilweise ebenfalls als Tabelle bzw. Spalte.

4.6 Ergebnisse des angepassten Tabellen-/Spalten-Modells

Wie bereits für das vorherige Modell in Kapitel 4.4 gezeigt, werden auch für das neue Modell einige Anpassungen der Lernparameter vorgenommen.

Die Farbkodierung ist dabei folgendermaßen:

- Learning-Rate: 1x10e-3, Epsilon: 1x10e-8 (Magenta), Standard / keine Änderung
- Learning-Rate: 1x10e-5, Epsilon: 1x10e-8 (Blau)
- Learning-Rate: 1x10e-5, Epsilon: 1x10e-4 (Grün)
- Learning-Rate: 1x10e-6, Epsilon: 1x10e-4 (Rot)

Der Graph zeigt wieder den Verlauf des Verlustes während der Validierung im Trainingsvorgang:

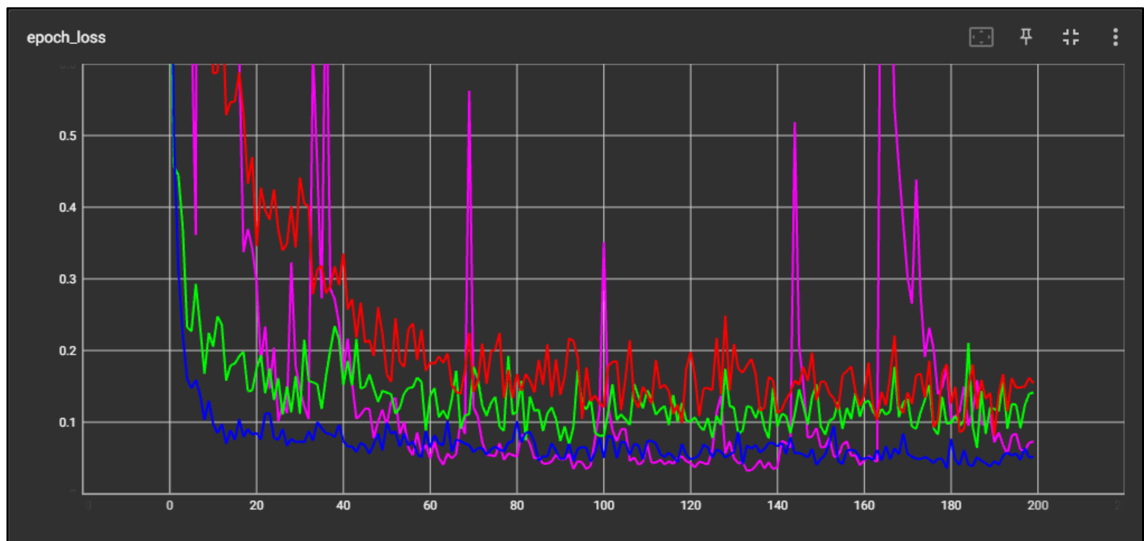


Abbildung 12: y-Achse: Loss (Verlust), x-Achse: Epoch (Durchlauf), Verlauf gilt für Validierung, Quelle: eigene Abbildung

Die Graphen verlaufen ähnlich zu den Gegenstücken aus Abbildung 10: Die ursprüngliche Einstellung (Magenta) verläuft im Vergleich zu den anderen (langsameren) learning-rate Einstellungen unruhig. Die langsamste learning-rate (Rot) schneidet über maximal 200 Epochs am schlechtesten ab. Der blaue Graph weist von allen den ruhigsten Verlauf auf. Dieser könnte bei einer höheren Anzahl Epochs einen kleineren Verlust als Magenta erreichen, da dieser aufgrund der gewählten Größe der learning-rate und von Epsilon feinere Abstimmungen der Gewichte vornehmen kann.

Um die minimalen Verluste der einzelnen Parametereinstellungen zu vergleichen, wird folgende Tabelle aufgestellt:

Parametereinstellung (Learning-Rate / Epsilon)	Minimaler Verlust (Validierung)
1x10e-3 / 1x10e-8	0,0322
1x10e-5 / 1x10e-8	0,0357
1x10e-5 / 1x10e-4	0,0642
1x10-e6 / 1x10-e4	0,0815

Tabelle 6: Parametereinstellungen für VGG19 Netz (Tabelle/Spalte) und minimaler Verlust (Validierung), Quelle: eigene Tabelle

Aus Tabelle 6 lässt sich entnehmen, dass die ursprüngliche Einstellung den minimalen Verlust während der Validierung vorweisen kann. Auch hier kann eine niedrigere learning-rate genutzt werden, um kleinere Anpassungen der Gewichte zuzulassen, um eventuell bessere Ergebnisse zu erzielen.

4.7 Auslesen der Informationen mit Pytesseract

Für das Auslesen der Informationen aus den Tabellen soll Pytesseract mit Hilfe der Masken ausschließlich Text aus den Tabellen erkennen und ausgeben. Es wird das gleiche Bild wie aus Abbildung 8 und 11 wiederverwendet.

Pytesseract gibt den ausgelesenen Text in der Konsole aus:

1 Das Berliner Luftgütemessnetz

Die Bundesländer sind nach § 44 (1) des Bundes-Immissionsschutzgesetzes (BImSchG) und der 39. Verordnung zur Durchführung des Bundes-Immissionsschutzgesetzes (BImSchV) verpflichtet, die Luftqualität kontinuierlich zu überwachen. Berlin kommt dieser Verpflichtung mit dem Berliner Luftgütemessnetz nach. Dieses besteht derzeit aus 16 Messstationen mit automatisch registrierenden Messgeräten. Davon sind zur Beschreibung der allgemeinen Immissionssituation fünf Messstationen im innerstädtischen Hintergrund (Wohn- und Gewerbegebieten), fünf im Stadtrand- und Waldbereich und sechs an Verkehrsschwerpunkten eingerichtet. An allen Stationen werden Stickstoffmonoxid und Stickstoffdioxid (Chemolumineszenzverfahren), an elf Stationen Partikel der PM_{10}-Fraktion (Messung der Streuung von Licht an Aerosolpartikeln), an acht Stationen Ozon (Absorption von UV-Strahlung) sowie an zwei Stationen Kohlenmonoxid (Absorption von Infrarotstrahlung) gemessen. Für Benzol und Schwefeldioxid liegen die Jahresmittelwerte unter der jeweiligen unteren Beurteilungsschwelle, daher wurden die Messungen an jeweils einer Station zum April 2019 eingestellt. Es wird noch an zwei Stationen Benzol (Gaschromatographie) und an einer Station Schwefeldioxid (durch UV-Fluoreszenz) gemessen. Alle Geräte werden einer monatlichen Kalibrierung unterzogen, die Gas-Messgeräte zusätzlich einer täglichen automatischen Funktionsüberprüfung.

Die Standorte der automatischen Stationen des Berliner Luftgütemessnetzes sind der Tabelle 1 zu entnehmen. Zusätzlich zu den genannten Stationen wird seit Dezember 2017 in der Leipziger Straße ein Messbus betrieben. Die Daten aller automatischen Stationen sind im Internet unter https://luftdaten.berlin.de/ abrufbar. Die Beurteilung der gemessenen Immissionsbelastung erfolgt durch Vergleich mit den geltenden Grenz- und Zielwerten, welche in Tabelle 2 aufgelistet sind.

Tabelle 1: Standorte der Luftgütemesscontainer und gemessene Komponenten (04/2019)

Nr.	Standort	Messkomponenten						
		Partikel-PM_{10}	SO_2	NO_x[1]	CO	O_3	BTX	Meteorolog. Größen
Stadtrand								
MC 027	Marienfelde			x		x		
MC 032	Grunewald	x		x		x		M[2]
MC 077	Buch	x		x		x		
MC 085	Friedrichshagen	x		x		x		
MC 145	Frohnau			x		x		
Innerstädtischer Hintergrund								
MC 010	Wedding	x		x		x		
MC 018	Schöneberg			x				
MC 042	Neukölln	x		x		x	x	T,F[2]
MC 171	Mitte	x		x				
MC 282	Karlshorst			x				
Verkehr								
MC 115	Hardenbergplatz			x				
MC 117	Schildhornstraße	x		x	x			
MC 124	Mariendorfer Damm	x		x				
MC 143	Silbersteinstraße	x		x				
MC 174	Frankfurter Allee	x	x	x	x	x	x	
MC 220	Karl-Marx-Straße	x		x				

1) Gemessen werden Stickstoffmonoxid (NO), Stickstoffdioxid (NO_2) und Stickstoffoxide (NO_x) als die Summe der Volumenmischungsverhältnisse von NO und NO_2.

2) T, F = Temperatur, relative Feuchte
M . = verschiedene meteorologische Parameter, zum Teil in 27 Meter Höhe: Temperatur, relative Feuchte, Luftdruck, Windgeschwindigkeit, Windrichtung, Strahlungsbilanz

```
Baten.berlin.d

Standort

Partikel
PMso

MC 027 Marienfelde

MC 032 Grunewald

MC 077 Buch

MC 085 Friedrichshagen
MC 145 Frohnau
```

```
U
Schönebg
Neukölln
Mitte
Karlshorst

Hardenbergplatz
Schildiaernstraße
Mariendorfer Damm
Silbersteinstraße
Frankfurter Allee
Karl-Marx-Straße
```

Abbildung 13: Bildinput bei 200DPI (links), Ausgabe von Pytesseract auf der Konsole (rechts), umgebrochen auf zwei Spalten, Quelle: eigene Abbildung

Hinweis Abbildung 13: Die Konsolenausgabe ist eigentlich eine Spalte breit, wurde jedoch zwecks besserer Lesbarkeit zwischen „MC 145 Frohnau“ und „U“ geschnitten und in zwei Spalten dargestellt.

Wie in Abbildung 13 zu erkennen, gibt Pytesseract lediglich einen Bruchteil der Tabelle aus. Außerdem werden einige Buchstaben/Zahlen nicht korrekt erkannt. Partikel PM_{10} wird dabei zu PMso, „Stadtrand“, „Innerstädtischer Hintergrund“ sowie „Verkehr“ und „MC 010 Wedding“ werden überhaupt nicht erkannt. „Schöneberg“ wird fehlerhaft ausgelesen.

Die Dokumentation von Tesseract empfiehlt, Bilder mit mindestens 300DPI zu verwenden [23]. Alle Seiten der PDF-Dokumente wurden mit Hilfe von pdf2image bei einer Auflösung von ungefähr 200DPI auf DIN A4 Größe umgewandelt. Dies entspricht einer Auflösung von 2339x1654. Das gleiche Bild aus Abbildung 13 wird erneut mit ca. 600DPI eingelesen (entspricht Auflösung von 7017x4961). Da das Bild nur eine höhere Auflösung hat, wird nur die Ausgabe auf der Konsole gezeigt:

```
MC 027
MC 032
MC 077
MC 085
MC 145

Standort

Marienfelde
Grunewald
Buch
Friedrichshagen
Frohnau

Wedding
Schöneberg
Neukölln
Mitte
Karlshorst

Hardenbergplatz
Schilekaonastraße
Mariendorfer Damm
Silbersteinstraße
Frankfurter Allee
Karl-Marx-Straße

passe

NNesskompongnter

aa ss
"= bobo-Jofkr[e
EEE EEE
```

Abbildung 14: Ausgabe von Pytesseract auf der Konsole bei 600DPI Eingangsauflösung, umgebrochen auf drei Spalten, Quelle: eigene Abbildung

Wie Abbildung 14 entnommen werden kann, verbessert sich die Erkennungsrate nur minimal. Interessanterweise wurden die Messtationsnummern und der Name des Ortes nun getrennt voneinander erkannt. „Schöneberg“ wird z.B. nun korrekt auf der Konsole ausgegeben.

Um die Ausgabe zu optimieren, bietet Tesseract insgesamt 14 verschiedene Betriebsmodi (page-segmentation-modes / psm) für die Zeichenerkennung an [23]. Auf das gleiche Bild (mit 600DPI) werden alle Betriebsmodi hintereinander angewandt, um den besten Betriebsmodus zu finden. Das beste Ergebnis liefert dabei der Modus psm 6, welcher beschrieben wird mit: „Assume a single uniform block of text“ [23].

Das Ergebnis dieses Modus sieht folgendermaßen aus:

```
|
ein Mies Eumss
Haten.berlin.c Bj,
gleich mit den <
abelle 1: Sta a
f} Mresskomponentee
Partikel- Meteorolog.
a [eberfofokrferr
MC 027 Marienfelde x x
MC 032 Grunewald x m x u‘ Mm“)
MC 077 Buch E BE BE i m
MC 085 Friedrichshagen xy m x Mi Mi x
MC 145 Frohnau u x x
Innerstädtischer Hintergrund Ü 0000
MC 010 Wedding u “ x x x
MC 018 Schöneberg Kon n
WE MC 042 Neukölln == x | x x x T,P)
MC 171 Mitte go m j x . n
MC 282 Karlshorst x u
Verkehr 000000000
MC 115 Hardenbergplatz or
MC 117 Schildaonmastraße x r x Ex WM u
WW MC 124 Mariendorfer Damm x x
MC 143 Silbersteinstraße x x
‘MC 174 Frankfurter Allee x" x a x
MC 220 Karl-Marx-Straße x x |
_ nessengve Ks
```

Abbildung 15: Ausgabe Pytesseract mit Modus: psm 6 bei 600DPI Eingangsauflösung, umgebrochen auf zwei Spalten, Quelle: eigene Abbildung

Abbildung 15 zeigt die Ausgabe im Modus psm 6, welcher auch viele „X“ aus der Tabelle erkennt und alles, neben der Messstationsnummer und dem Ort, in eine eigene Zeile schreibt. Auch bei diesem Modus gibt es jedoch deutliche Fehler beim Auslesen des Textes, wie zum Beispiel „f} Mresskomponentee“, welche eine weitere Verarbeitung durch einen Algorithmus zur Speicherung in eine .csv-Datei verhindert. In der Theorie wäre es bei einer korrekten Erkennung möglich, den Text direkt in eine entsprechende Datei zu schreiben und die Leerzeichen durch ein Semikolon oder ein Komma zu ersetzen, auch wenn dies z.B. bei „Frankfurter Allee“ zu einem falsch gesetzten Semikolon führen würde. Für diese Fälle wäre eine spezielle Anpassung des Algorithmus (oder des neuronalen Netzes) nötig.

4.8 Reevaluierung der Trainingsdaten

Der derzeit genutzte Datensatz kommt, wie in Kapitel 3.3 erwähnt, aus drei verschiedenen Quellen. Pro Quelle sehen die Tabellen ähnlich aus, was dazu führen kann, dass das neuronale Netz zu wenig Varianz im Input bekommt. Außerdem sind einige Tabellen des Datensatzes komplex. Hier wird mit vereinigten Spalten gearbeitet, welche wiederum in der nächsten Zeile durch kleinere Spalten unterteilt werden (z.B. für Messwert Min / Avg / Max). Die Trainingsdaten waren für den Fall der Erkennung der Tabellen nicht ausreichend, um anschließend mit Hilfe der Masken den Text der Tabellen auslesen zu können. Mit etwas mehr als 100 Beispielen für Tabellen, welche teilweise komplex ausfallen, bietet der Trainingsdatensatz nicht genügend Merkmale für die Extraktion durch das Netz, um zuverlässig eingesetzt werden zu können. Eine Vergrößerung des Datensatzes und eine Steigerung der Rechenleistung wäre für eine weitere Untersuchung notwendig.

5. Fazit und Ausblick

Um die Ergebnisse der Arbeit besser einschätzen zu können, wird im folgenden Kapitel ein kurzes Fazit und ein Ausblick auf die möglichen Einsätze von künstlicher Intelligenz in der öffentlichen Verwaltung der Zukunft gegeben.

Die Arbeit sollte prüfen, ob ein neuronales Netz dazu in der Lage ist, die visuellen Merkmale und Eigenschaften von Tabellen zu erlernen, um Masken zu generieren, welche irrelevante Informationen eines Dokumentes ausblenden, um mit Hilfe einer Texterkennung die Tabellendaten auszulesen und anschließend abzuspeichern. Zunächst wurden die Grundlagen von Datenplattformen und künstlichen neuronalen Netzen erklärt und anhand der Problemstellung ein Convolutional Neural Network für Computer Vision ausgewählt. Anschließend wurden zwei Modelle mit unterschiedlichen Netzarchitekturen und Lernparametern trainiert, auf Funktionalität geprüft und die Ergebnisse miteinander verglichen, um eine Aussage darüber zu treffen, welches der beiden Modelle (mit welcher Architektur und welchen Parameterwerten) besser für eine Tabellenerkennung geeignet ist. Das zweite Modell (Tabelle / Spalte) schnitt dabei im direkten Vergleich zum ersten Modell (Kopfzeile / Vorspalte / Felder) besser ab. In der Auswertung der Ergebnisse zeigte sich, dass es einem künstlichen neuronalen Netz grundlegend möglich ist, gewisse Merkmale von Tabellen zu erkennen (z.B. Kopfzeile / Vorspalte / Felder / Spalten), wobei die Ergebnisse beider Modelle im jetzigen Zustand keine Hilfe im praktischen Einsatz sind. Obwohl beide Modelle nicht dazu geeignet sind, nicht-maschinenlesbare Datensätze vollständig (oder teilweise) automatisiert zu digitalisieren, bieten die Netze (besonders das zweite Netz) eine Grundlage für weitere Anpassungen und einen Ausgangspunkt, um weitere Daten beim Training zu berücksichtigen. Eine Speicherung in einem offenen Datenstandard (wie CSV) ist dabei nicht erfolgt, da die ungenauen Ergebnisse der Texterkennung dies nicht zuließen.

Für den Einsatz dieses neuronalen Netzes stellt sich für die öffentliche Verwaltung die Frage, ob sich der Aufwand auch aus Kostengründen lohnt, da die Ressourcen zur Berechnung von größeren Datensätzen zur Verfügung stehen müssen (Rechenleistung, Personal für Training der KI usw.). Je nach Umfang der Daten muss abgewägt werden, ob ein manuelles Digitalisieren der Daten nicht effizienter wäre, da nach Abschließen der Digitalisierung in der Regel keine neuen, nicht-digitale Daten erhoben werden, wodurch der Einsatz dieses neuronalen Netzes überflüssig wird.

5.1 Erkennung von Zusammenhängen in einer UDP

Da in einer UDP verschiedene Fachdaten aus unterschiedlichen Bereichen zusammenlaufen, bietet sich eine Analyse der Daten durch eine KI an. Wie in Kapitel 2.1 erwähnt, fehlt es der öffentlichen Verwaltung an Erfahrung, um einen Nutzen aus der Verbindung

verschiedener Datensätze zu ziehen. Mit Hilfe einer KI kann ein Teil dieser Arbeit abgenommen werden, indem ein neuronales Netz (womöglich für den Menschen versteckte) Beziehungen zwischen Datensätzen erkennen und diese miteinander verknüpfen kann (Clustering) [24, S. 18].

5.2 Auswertung von Punktwolken mit Hilfe von neuronalen Netzen

Punktwolken sind eine Ansammlung von Punkten in einem dreidimensionalen Raum, welche mit Hilfe von LiDAR (Licht-Radar) Sensoren oder RBG-A Kameras erzeugt werden können, welche die Möglichkeit bieten, eine detailgetreue Abbildung der Realität zu erstellen [24, S. 141].

Künstliche neuronale Netze können Anwendung in der Analyse bzw. Kategorisierung von Punktwolken finden. Hierzu muss das entsprechende Modell eine Klassifizierung einzelner Punkte in einer Punktwolke vornehmen (zum Beispiel Einteilung in Straßen oder Gebäude), um ein automatisiertes Zuordnen von Kategorien zu ermöglichen. Dies kann unter anderem dazu genutzt werden, Straßenbaustellen zu überwachen [24, S. 143].

Literatur

[1] K. Schlüter *et al.* “Die Stadt der Zukunft mit Daten gestalten: Souveräne Städte – nachhaltige Investitionen in Dateninfrastrukturen.” https://www.staedtetag.de/files/dst/docs/Publikationen/Weitere-Publikationen/2021/stadt-der-Zukunft-mit-daten-gestalten-studie-2021.pdf (Zugriff am: 2. Feb. 2023).

[2] F. Chollet, *Deep Learning mit Python und Keras,* 1. Aufl. mitp-Verlag.

[3] GDI-NI. “Was ist Geodateninfrastruktur (GDI) ?” https://www.geodaten.niedersachsen.de/gdini/was_ist_geodateninfrastruktur_gdi/was-ist-geodateninfrastruktur-gdi--25394.html (Zugriff am: 2. Feb. 2023).

[4] K. Buehler und G. Percivall. “OGC Reference Model.” http://www.opengis.net/doc/orm/2.1 (Zugriff am: 2. Feb. 2023).

[5] *Referenzarchitekturmodell Offene Urbane Plattform (OUP); Text Englisch: Reference Architecture Model Open Urban Platform (OUP); Text in English*, 91357, DIN Deutsches Institut für Normung e. V., Berlin, Dez. 2017. [Online]. Verfügbar unter: https://www.beuth.de/de/technische-regel/din-spec-91357/281077528

[6] N. Ketkar, *Deep Learning with Python: A Hands-on Introduction*. Berkeley, CA: Apress (2017); Imprint: Apress, 2017.

[7] S. Raschka, V. Mirjalili und K. Lorenzen, *Machine learning mit Python und Keras, TensorFlow 2 und Scikit-learn: das umfassende Praxis-Handbuch für Data Science, Deep Learning und Predictive Analytics,* 3. Aufl. Frechen: mitp, 2021. [Online]. Verfügbar unter: https://content-select.com/de/portal/media/view/5f3148f0-fd00-481c-99d3-6bb9b0dd2d03?

[8] F. Chollet und Andere. “About Keras.” https://keras.io/about/ (Zugriff am: 2. Feb. 2023).

[9] Techpowerup. “AMD Ryzen Threadripper 3990X.” https://www.techpowerup.com/cpu-specs/ryzen-threadripper-3990x.c2271 (Zugriff am: 2. Feb. 2023).

[10] Techpowerup. “NVIDIA GeForce RTX 4090.” https://www.techpowerup.com/gpu-specs/geforce-rtx-4090.c3889 (Zugriff am: 23. Jan. 2023).

[11] NVIDIA Corporation & affiliates. “NVIDIA cuDNN: Developer Guide | NVIDIA Docs.” https://docs.nvidia.com/deeplearning/cudnn/pdf/cuDNN-Developer-Guide.pdf (Zugriff am: 2. Feb. 2023).

[12] R. Smith. “An Overview of the Tesseract OCR Engine.” https://github.com/tesseract-ocr/docs/blob/main/tesseracticdar2007.pdf (Zugriff am: 2. Feb. 2023).

[13] S. Hoffstaetter. “pytesseract 0.3.10 - Documentation.” https://pypi.org/project/pytesseract/ (Zugriff am: 2. Feb. 2023).

[14] J. Frochte, *Maschinelles Lernen: Grundlagen und Algorithmen in Python,* 3. Aufl. München: Hanser Verlag, 2020.

[15] T.-Y. Lin, P. Goyal, R. B. Girshick, K. He und P. Dollár, "Focal Loss for Dense Object Detection," *CoRR*, Jg. abs/1708.02002, 2017. doi: 10.48550/ARXIV.1708.02002. [Online]. Verfügbar unter: https://arxiv.org/abs/1708.02002

[16] L. Fei-Fei, J. Deng, O. Russakovsky, A. Berg und K. Li. "About ImageNet." https://www.image-net.org/about.php (Zugriff am: 2. Feb. 2023).

[17] S. Paliwal, V. D, R. Rahul, M. Sharma und L. Vig, "TableNet: Deep Learning model for end-to-end Table detection and Tabular data extraction from Scanned Document Images," *CoRR*, Jg. abs/2001.01469, 2020. doi: 10.48550/ARXIV.2001.01469. [Online]. Verfügbar unter: https://arxiv.org/abs/2001.01469

[18] D. P. Kingma und J. Ba. "Adam: A Method for Stochastic Optimization." https://arxiv.org/abs/1412.6980 (Zugriff am: 2. Feb. 2023).

[19] F. Chollet und Andere. "tf.keras.optimizers.Adam." https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/Adam (Zugriff am: 2. Feb. 2023).

[20] S. Kannojia und G. Jaiswal, "Effects of Varying Resolution on Performance of CNN based Image Classification An Experimental Study," *International Journal of Computer Sciences and Engineering*, Jg. 6, S. 451–456, 2018, doi: 10.26438/ijcse/v6i9.451456.

[21] C. F. Sabottke und B. M. Spieler, "The Effect of Image Resolution on Deep Learning in Radiography," *Radiology: Artificial Intelligence*, Jg. 2, Nr. 1, e190015, 2020. doi: 10.1148/ryai.2019190015. [Online]. Verfügbar unter: https://pubs.rsna.org/doi/full/10.1148/ryai.2019190015

[22] OpenCV.org. "Eroding and Dilating." https://docs.opencv.org/3.4/db/df6/tutorial_erosion_dilatation.html (Zugriff am: 2. Feb. 2023).

[23] R. Smith und Andere. "Improving the quality of the output." https://tesseract-ocr.github.io/tessdoc/ImproveQuality.html (Zugriff am: 2. Feb. 2023).

[24] W. Grunau, *Künstliche Intelligenz in Geodäsie und Geoinformatik: Potenziale und Best-Practice-Beispiele* (VDV-Schriftenreihe). Berlin: VDE Verlag, 2022. [Online]. Verfügbar unter: http://www.content-select.com/index.php?id=bib_view&ean=9783879077182; https://content-select.com/portal/media/view/62189548-9c84-4f87-9ea8-46a6b0dd2d03; https://content-select.com/portal/media/cover_image/62189548-9c84-4f87-9ea8-46a6b0dd2d03/500

Anhang

Anlage 1 - pdfToJPG.py:

```
from pdf2image import convert_from_path
import os

# skript wandelt pdf-dateien im ordner "PDF_IN" zu jpg-dateien um und
legt diese in "PDF_OUT" ab
# standardauflösung DPI~200

input_dir = "PDF_In"
output_dir = "PDF_Out"

files = os.listdir(input_dir)

for file in files:
    pdf = convert_from_path(os.path.join(input_dir, file))
    count = 0
    for page in pdf:
        path = os.path.join(output_dir, file)
        page.save(path + "-" + str(count) + ".jpg", "JPEG")
        count += 1
```

Anlage 2 - preProcess.py:

```
import json
import os
from PIL import Image, ImageDraw
import pandas as pd
import numpy as np

"""
struktur des ordners:
root
|_____data
        |_____*.jpg
|_____labels.json
"""

# setze pfade
img_dir = os.path.join("stuff", "data")
annot_file = os.path.join("stuff", "labels.json")


def extract_data():
    """
    lade daten und gib annotations, images und filtered annotations
als listen zurück
    id der kategorien gehören zu folgenden labeln:
    1 - ueberschrift
    2 - tabelle
    3 - kopfzeile
    4 - vorspalte
    5 - felder
    6 - spalte
    """

    # öffne annotation datei, lade daten in ein json objekt und
schließe anschließend den filestream
    f = open(annot_file)
    json_data = json.load(f)
    f.close()
```

```
    # hole image und annotation informationen aus dem json objekt und
speichere diese in eigenen listen
    j_annot = json_data["annotations"]
    j_imgs = json_data["images"]

    # erstelle eigene liste für jedes label
    hl_list = []
    t_list = []
    th_list = []
    fc_list = []
    f_list = []
    c_list = []

    # füge annotation objekt zur passenden liste hinzu
    for annot_object in j_annot:
        match annot_object["category_id"]:
            case 1:
                hl_list.append(annot_object)
            case 2:
                t_list.append(annot_object)
            case 3:
                th_list.append(annot_object)
            case 4:
                fc_list.append(annot_object)
            case 5:
                f_list.append(annot_object)
            case 6:
                c_list.append(annot_object)
    return j_imgs, j_imgs, hl_list, t_list, th_list, fc_list, f_list,
c_list


def retrieve_image_path(id, imgs):
    """
    funktion um einen dateinamen mit hilfe einer id aus einer liste zu
finden
    wird benötigt, damit das neuronale netz die später generierten
masken zu den dateinamen zuordnen kann
    """
    for image in imgs:
        if image["id"] == id:
            return image["file_name"]
    return -1


# hole alle benötigten informationen mit der extract_data() funktion
aus der annotation datei (.json)
json_annot, \
    json_imgs, \
    headline_list, \
    table_list, \
    tablehead_list, \
    frontcolumn_list, \
    field_list, \
    column_list = extract_data()


def get_bboxes(headlines, tables, tableheads, frontcolumns, fields,
columns, verbose=False):
    """
    hole bounding boxes aus den annotations der verschiedenen klassen
```

```
    bounding boxes werden als 2 koordinaten gespeichert ([x, y], [x,
y]) zwischen denen ein rechteck gespannt wird
    die rechtecke repräsentieren den umfang, in welchem eine gewisse
information zu sehen ist (z.B. tabelle/spalte)
    die funktion speichert die koordinaten in einem bounding box dic-
tionary mit folgender struktur:

    b_dict{}
    |_____[filename]
                |_____headlines
                        |_____list([[bounding_box_coordinates],[...]])
                |_____tables
                        |_____list([[bounding_box_coordinates],[...]])
                |_____tableheads
                        |_____list([[bounding_box_coordinates],[...]])
                |_____frontcolumns
                        |_____list([[bounding_box_coordinates],[...]])
                |_____fields
                        |_____list([[bounding_box_coordinates],[...]])
                |_____columns
                        |_____list([[bounding_box_coordinates],[...]])
    |_____[filename]
    [...]
    """
    b_dict = {}
    for u in headlines:
        fname = retrieve_image_path(u["image_id"], json_imgs)
        if fname in b_dict:
            if verbose:
                print("already exists")
        else:
            b_dict[fname] = dict()
        if "headlines" in b_dict[fname]:
            if verbose:
                print("already exists")
        else:
            b_dict[fname]["headlines"] = list()
        b_dict[fname]["headlines"].append(u["bbox"])

    for v in tables:
        fname = retrieve_image_path(v["image_id"], json_imgs)
        if fname in b_dict:
            if verbose:
                print("already exists")
        else:
            b_dict[fname] = dict()
        if "tables" in b_dict[fname]:
            if verbose:
                print("already exists")
        else:
            b_dict[fname]["tables"] = list()
        b_dict[fname]["tables"].append(v["bbox"])

    for w in tableheads:
        fname = retrieve_image_path(w["image_id"], json_imgs)
        if fname in b_dict:
            if verbose:
                print("already exists")
        else:
            b_dict[fname] = dict()
        if "tableheads" in b_dict[fname]:
            if verbose:
                print("already exists")
```

```
        else:
            b_dict[fname]["tableheads"] = list()
        b_dict[fname]["tableheads"].append(w["bbox"])

    for x in frontcolumns:
        fname = retrieve_image_path(x["image_id"], json_imgs)
        if fname in b_dict:
            if verbose:
                print("already exists")
        else:
            b_dict[fname] = dict()
        if "frontcolumns" in b_dict[fname]:
            if verbose:
                print("already exists")
        else:
            b_dict[fname]["frontcolumns"] = list()
        b_dict[fname]["frontcolumns"].append(x["bbox"])

    for y in fields:
        fname = retrieve_image_path(y["image_id"], json_imgs)
        if fname in b_dict:
            if verbose:
                print("already exists")
        else:
            b_dict[fname] = dict()
        if "fields" in b_dict[fname]:
            if verbose:
                print("already exists")
        else:
            b_dict[fname]["fields"] = list()
        b_dict[fname]["fields"].append(y["bbox"])

    for z in columns:
        fname = retrieve_image_path(z["image_id"], json_imgs)
        if fname in b_dict:
            if verbose:
                print("already exists")
        else:
            b_dict[fname] = dict()
        if "columns" in b_dict[fname]:
            if verbose:
                print("already exists")
        else:
            b_dict[fname]["columns"] = list()
        b_dict[fname]["columns"].append(z["bbox"])
    return b_dict


# speichere bounding boxes in bbox dictionary
bbox_dict = get_bboxes(headline_list, table_list, tablehead_list,
frontcolumn_list, field_list, column_list)


def save_masks(fname, list_bbox, dim, type):
    """
    erstellt masken aus bounding box liste und speichert diese in ei-
ner .jpg datei ab
    masken werden auf typbasis erstellt - masken für die spalten wer-
den z.B. in masked/columns gespeichert usw.
    """
    mask_dir = os.path.join("stuff", "masked", type)
    if not os.path.exists(mask_dir):
        os.makedirs(mask_dir)
```

```
    image = Image.new("RGB", dim)
    mask = ImageDraw.Draw(image)
    for each_list in list_bbox:
        [x, y, width, height] = each_list
        mask.rectangle([x, y, x + width, y + height], fill=255)
    mask_fname = os.path.join(mask_dir, fname)
    image = np.array(image)
    image = Image.fromarray(image[:, :, 0])
    image.save(mask_fname)
    return mask_fname


def create_masks_df():
    """
    nutze alles bisher gesammelte, um einen dataframe zu erstellen,
sodass alle informationen an einem ort sind
    spalten beinhalten den pfad des bildes und alle zugehörigen pfade
der masken
    """
    img_df = pd.DataFrame(
        columns=["image_path", "headlinemask_path", "tablemask_path",
"tableheadmask_path",
                 "frontcolumnmask_path", "fieldmask_path", "column-
mask_path"])
    csv_dir = os.path.join("stuff", "csv")
    if not os.path.exists(csv_dir):
        os.mkdir(csv_dir)
    for file_name, bboxes_dict in bbox_dict.items():
        img_path = os.path.join(img_dir, file_name)
        dim = Image.open(img_path).size
        for type, bboxes, in bboxes_dict.items():
            match type:
                case "headlines":
                    headlinemask_path = save_masks(file_name, bboxes,
dim, type)
                case "tables":
                    tablemask_path = save_masks(file_name, bboxes,
dim, type)
                case "tableheads":
                    tableheadmask_path = save_masks(file_name, bboxes,
dim, type)
                case "frontcolumns":
                    frontcolumnmask_path = save_masks(file_name, bbo-
xes, dim, type)
                case "fields":
                    fieldmask_path = save_masks(file_name, bboxes,
dim, type)
                case "columns":
                    columnmask_path = save_masks(file_name, bboxes,
dim, type)
        # füge alle gesammelten informationen aus dem derzeitigen
schleifendurchlauf am ende des dataframes ein
        img_df.loc[len(img_df.index)] = [img_path, headlinemask_path,
tablemask_path, tableheadmask_path,
                                          frontcolumnmask_path, field-
mask_path, columnmask_path]
    # speichere den dataframe in einer .csv datei für die spätere nut-
zung mit dem neuronalen netz
    csv_fname = os.path.join(csv_dir, "data.csv")
    img_df.to_csv(csv_fname, index=False)
    return img_df
```

```
"""
create_masks_df() muss nur einmal ausgeführt werden, um die masken und
die .csv für das netz zu erstellen
erneut ausführen, falls neue trainingsdaten hinzugefügt werden sollen
muss zu keiner variable zugewiesen werden - die funktion würde auch
ohne ein return funktionieren
nur für den fall, dass ich mich dazu entscheide die variable für de-
bugging zwecke zu untersuchen oder falls ich die
methode aus einem anderen python script aufrufen möchte
"""
img_df = create_masks_df()
print(img_df)
```

Anlage 3 - trainModel_head_front_field.py:

```
import tensorflow as tf
import os
import pandas as pd
import matplotlib.pyplot as plt
from keras.layers import Layer, Conv2D, UpSampling2D, Concatenate,
Conv2DTranspose, Dropout
from keras.applications import VGG16, VGG19, ResNet50, DenseNet201,
EfficientNetB5, Xception, NASNetLarge
from keras import Model
from keras.callbacks import TensorBoard, ModelCheckpoint
from keras.utils.vis_utils import plot_model

# setze alle notwendigen verzeichnisse
train_dir = os.path.join("stuff")
test_dataset_dir = "D:/datasets/test_data"
checkpoint_path = "D:\keras_models\head_front_field\learning_1e-03_ep-
silon_1e-08_dropout_0.8\VGG19_1024/cp_{epoch:04d}_{val_loss:.4f}.ckpt"
checkpoint_dir = os.path.dirname(checkpoint_path)
log_dir = "D:\keras_models\head_front_field\learning_1e-03_epsilon_1e-
08_dropout_0.8\VGG16_1024_logs"


def main():
    # erstelle dataframe aus zuvor ersteller .csv datei
    train_df = pd.read_csv(os.path.join(train_dir, "csv", "data.csv"))

    # setzt das backend zurück - nur zur sicherheit
    tf.keras.backend.clear_session()
    # from_tensor_slices erstellt jeweils ein element pro reihe des
datensatzes
    dataset = tf.data.Dataset.from_tensor_slices((train_df['image_pa-
th'].values,
                                                  train_df["tablehead-
mask_path"].values,
                                                  train_df["frontco-
lumnmask_path"].values,
                                                  train_df["field-
mask_path"].values))

    # erstellt train, validation und test datensatz
    dataset_size = len(dataset)
    train_size = int(0.8 * dataset_size)
    val_size = int(0.1 * dataset_size)
    test_size = int(0.1 * dataset_size)

    dataset = dataset.shuffle(buffer_size=42)
    train_dataset = dataset.take(train_size)
    test_dataset = dataset.skip(train_size)
```

```
    val_dataset = test_dataset.skip(val_size)
    test_dataset = test_dataset.take(test_size)

    train_dataset = train_dataset.map(map_function).batch(1)
    val_dataset = val_dataset.map(map_function).batch(1)
    test_dataset = test_dataset.map(map_function).batch(1)

    # test datensatz wird abgespeichert
    if not os.path.exists(test_dataset_dir):
        os.makedirs(test_dataset_dir)
    test_dataset.save(test_dataset_dir)

    model = build_model()
    model.summary()

    # gibt ein "flow-chart" des zuvor erstellten models aus
    plot_model(model, to_file="model_plot.png", show_shapes=True,
show_layer_names=True)

    # erstelle metriken für den verlust
    losses = {
        "tablehead": tf.keras.losses.SparseCategoricalCrossen-
tropy(from_logits=True),
        "frontcolumn": tf.keras.losses.SparseCategoricalCrossen-
tropy(from_logits=True),
        "fields": tf.keras.losses.SparseCategoricalCrossen-
tropy(from_logits=True)
    }

    # möglichkeit, um die gewichtung der verluste anzupassen - stan-
dard 1.0 für alle
    loss_weights = {
        "tablehead": 1.0,
        "frontcolumn": 1.0,
        "fields": 1.0
    }

    # einstellen des adam algorithmus für das neuronale netz
    # https://www.tensorflow.org/versions/r1.15/api_docs/py-
thon/tf/train/AdamOptimizer
    # https://www.tensorflow.org/api_docs/python/tf/keras/optimi-
zers/Adam
    optim = tf.keras.optimizers.Adam(learning_rate=1e-03, epsilon=1e-
08)

    # erstelle score für die genauigkeit (sc_acc)
    sc_acc = tf.keras.metrics.SparseCategoricalAccuracy(name="sc_ac-
curacy")

    # kompiliert das model mit den zuvor eingestellten parametern
    model.compile(optimizer=optim,
                  loss=losses,
                  metrics=[sc_acc],
                  loss_weights=loss_weights)

    # zeige vorhersagen, bevor das netz trainiert worden ist
    # ref: https://www.tensorflow.org/tutorials/images/segmentation
    """
    for image, msks in val_dataset.take(1):
        tablehead_mask, frontcolumn_mask, fields_mask = msks['tab-
lehead'], msks['frontcolumn'], msks['fields']
        pred_tabh_mask, pred_fc_mask, pred_fields_mask = model.pre-
dict(image)
```

```
        show_imgs([image[0], create_masks(pred_tabh_mask), cre-
ate_masks(pred_fc_mask), create_masks(pred_fields_mask)])
    """

    # erstelle model callbacks für checkpoints und tensorboard für
spätere analyse der logs
    if not os.path.exists(checkpoint_dir):
        os.makedirs(checkpoint_dir)
    if not os.path.exists(log_dir):
        os.makedirs(log_dir)

    # checkpoints werden bei verbesserter validation loss gespeichert
    cp_callback = ModelCheckpoint(checkpoint_path, verbose=1,
save_best_only=True, monitor='val_loss',
                                  save_weights_only=True)
    # tensorboard callback für des spätere auswerten der logs
    tb_callback = TensorBoard(log_dir)
    callbacks = [cp_callback, tb_callback]

    model.fit(
        train_dataset, epochs=200,
        batch_size=len(train_dataset) // 2,
        steps_per_epoch=len(train_dataset) // 1,
        validation_data=val_dataset,
        validation_steps=len(val_dataset) // 1,
        callbacks=callbacks
    )


# funktion um eine liste von bilder anzeigen zu lassen
def show_imgs(image_list):
    plt.figure(figsize=(30, 30))
    names = ["Input Image", "Tablehead Mask", "Frontcolumn Mask",
"Fieldmask Path"]
    for i in range(len(image_list)):
        plt.subplot(1, len(image_list), i + 1)  # +1, da pyplot nur
indizes größer 0 unterstützt
        plt.title(names[i])
        plt.imshow(tf.keras.utils.array_to_img(image_list[i]))
        plt.axis('off')
    plt.show()


# vorbereitung der trainingsdaten durch map_function()
def map_function(img, th_mask, fc_mask, f_mask):
    """
    nimmt jpg dateien aus dem dataset entgegen und dekodiert diese in
ein für tensorflow brauchbares format
    das bild selbst wird in RGB dekodiert - die masken werden mit nur
einem channel dekodiert
    alle bilder werden auf eine zuvor festgelegte größe (dim) skaliert
und die farben werden von 0 bis 1 normalisiert
    """
    dim = (1024, 1024)
    decoded = tf.io.decode_jpeg(tf.io.read_file(img), channels=3)  #
image in RGB
    img = tf.cast(decoded, tf.float32)
    img = tf.image.resize(img, dim)
    img = tf.cast(img, tf.float32) / 255

    th_msk_decoded = tf.io.decode_jpeg(tf.io.read_file(th_mask), chan-
nels=1)
    th_msk = tf.cast(th_msk_decoded, tf.float32)
```

```
    th_msk = tf.image.resize(th_msk, dim)
    th_msk = tf.cast(th_msk, tf.float32) / 255

    fc_msk_decoded = tf.io.decode_jpeg(tf.io.read_file(fc_mask), chan-
nels=1)
    fc_msk = tf.cast(fc_msk_decoded, tf.float32)
    fc_msk = tf.image.resize(fc_msk, dim)
    fc_msk = tf.cast(fc_msk, tf.float32) / 255

    f_msk_decoded = tf.io.decode_jpeg(tf.io.read_file(f_mask), chan-
nels=1)
    f_msk = tf.cast(f_msk_decoded, tf.float32)
    f_msk = tf.image.resize(f_msk, dim)
    f_msk = tf.cast(f_msk, tf.float32) / 255

    mask_dict = {
        'tablehead': th_msk,
        'frontcolumn': fc_msk,
        'fields': f_msk
    }

    return img, mask_dict


class TableheadConvLayer(Layer):
    """
    ref: TableNet: Deep Learning model for end-to-end
    Table detection and Tabular data extraction from
    Scanned Document Images
    Shubham Paliwal, Vishwanath D, Rohit Rahul, Monika Sharma, Love-
kesh Vig
    TCS Research, New Delhi
    {shubham.p3, vishwanath.d2, monika.sharma1, rohit.rahul, love-
kesh.vig}@tcs.com
    Creating new custom layer classes for neural network
    ref: https://faroit.com/keras-docs/2.0.1/layers/writing-your-own-
keras-layers/#writing-your-own-keras-layers
    """
    def __int__(self, output_dim, **kwargs):
        self.output_dim = output_dim
        super(TableheadConvLayer, self).__int__(**kwargs)

    def build(self, input_shape):
        self.conv7 = Conv2D(512, 1, activation='relu', name='conv7tab-
lehead')
        self.upsample_conv7 = UpSampling2D(2)
        self.concat_p4 = Concatenate()
        self.upsample_p4 = UpSampling2D(2)

        self.concat_p3 = Concatenate()
        self.upsample_p3 = UpSampling2D(2)

        self.upsample_p3_2 = UpSampling2D(2)
        self.convtranspose = Conv2DTranspose(3, 3, strides=2, pad-
ding='same')
        super(TableheadConvLayer, self).build(input_shape)

    def call(self, x):
        x, y, z = x
        x = self.conv7(x)
        x = self.upsample_conv7(x)
        x = self.concat_p4([x, z])
        x = self.upsample_p4(x)
```

```
        x = self.concat_p3([x, y])
        x = self.upsample_p3(x)
        x = self.upsample_p3_2(x)
        x = self.convtranspose(x)

        return x


class FrontcolumnConvLayer(Layer):

    def __int__(self, output_dim, **kwargs):
        self.output_dim = output_dim
        super(FrontcolumnConvLayer, self).__int__(**kwargs)

    def build(self, input_shape):
        self.conv7 = Conv2D(512, 1, activation='relu',
name='conv7frontcolumn')
        self.upsample_conv7 = UpSampling2D(2)
        self.concat_p4 = Concatenate()
        self.upsample_p4 = UpSampling2D(2)

        self.concat_p3 = Concatenate()
        self.upsample_p3 = UpSampling2D(2)

        self.upsample_p3_2 = UpSampling2D(2)
        self.convtranspose = Conv2DTranspose(3, 3, strides=2, pad-
ding='same')
        super(FrontcolumnConvLayer, self).build(input_shape)

    def call(self, x):
        x, y, z = x
        x = self.conv7(x)
        x = self.upsample_conv7(x)
        x = self.concat_p4([x, z])
        x = self.upsample_p4(x)
        x = self.concat_p3([x, y])
        x = self.upsample_p3(x)
        x = self.upsample_p3_2(x)
        x = self.convtranspose(x)

        return x


class FieldsConvLayer(Layer):

    def __int__(self, output_dim, **kwargs):
        self.output_dim = output_dim
        super(FieldsConvLayer, self).__int__(**kwargs)

    def build(self, input_shape):
        self.conv7 = Conv2D(512, 1, activation='relu',
name='conv7fields')
        self.upsample_conv7 = UpSampling2D(2)
        self.concat_p4 = Concatenate()
        self.upsample_p4 = UpSampling2D(2)

        self.concat_p3 = Concatenate()
        self.upsample_p3 = UpSampling2D(2)

        self.upsample_p3_2 = UpSampling2D(2)
        self.convtranspose = Conv2DTranspose(3, 3, strides=2, pad-
ding='same')
        super(FieldsConvLayer, self).build(input_shape)
```

```
    def call(self, x):
        x, y, z = x
        x = self.conv7(x)
        x = self.upsample_conv7(x)
        x = self.concat_p4([x, z])
        x = self.upsample_p4(x)
        x = self.concat_p3([x, y])
        x = self.upsample_p3(x)
        x = self.upsample_p3_2(x)
        x = self.convtranspose(x)

        return x


def build_model():
    """
    erstellt das model mit hilfe eines bereits existierenden netzes
und dem anschließenden anhängen der zuvor
    erstellten output layer
    input shape kann je nach verwendetem netz angepasst werden, um in
vorhandenen speicher zu passen
    """
    tf.keras.backend.clear_session()
    input_shape = (1024, 1024, 3)

    base = VGG19(input_shape=input_shape, include_top=False,
weights='imagenet')

    # VGG 16/19
    end_layers_list = ['block3_pool', 'block4_pool', 'block5_pool']

    # ResNet50
    # end_layers_list = ['conv3_block4_3_conv', 'conv4_block6_3_conv',
'conv5_block3_3_conv']

    # DenseNet201
    # end_layers_list = ['pool3_pool', 'pool4_pool', 'relu']

    # Xception
    # end_layers_list = ['block10_sepconv1', 'block12_sepconv1',
'block14_sepconv1']

    # NASNetLarge
    # end_layers_list = ['activation_238', 'activation_250', 'activa-
tion_259']

    # EfficientNetB5
    # end_layers_list = ['block6a_expand_activation', 'block7a_activa-
tion', 'top_activation']

    end_layers = [base.get_layer(i).output for i in end_layers_list]
    x = Conv2D(512, (1, 1), activation='relu')(end_layers[-1])  # last
element of end_layers
    x = Dropout(0.8)(x)
    x = Conv2D(512, (1, 1), activation='relu')(x)
    x = Dropout(0.8)(x)

    # VGG16/19
    y = end_layers[0]
    z = end_layers[1]

    # EfficientNetB5
```

```
    # y = UpSampling2D(2)(end_layers[0])
    # y = Conv2D(1024, 1)(y)
    # z = UpSampling2D(2)(end_layers[1])
    # z = Conv2D(512, 1)(z)

    # Xception
    # y = UpSampling2D(2)(end_layers[0])
    # y = Conv2D(256, 1)(y)
    # z = end_layers[1]
    # z = Conv2D(512, 1)(z)

    # NASNetLarge
    # y = UpSampling2D(4)(end_layers[0])
    # y = Conv2D(1024, 1)(y)
    # z = UpSampling2D(2)(end_layers[1])
    # z = Conv2D(512, 1)(z)

    # DenseNet201
    # y = UpSampling2D(2)(end_layers[0])
    # z = UpSampling2D(2)(end_layers[1])

    # ResNet50
    # y = end_layers[0]
    # y = Conv2D(1024, 1)(y)
    # z = end_layers[1]
    # z = Conv2D(512, 1)(z)

    tablehead_branch = TableheadConvLayer(name='tablehead')([x, y, z])
    frontcolumn_branch = FrontcolumnConvLayer(name='frontcolumn')([x,
y, z])
    fields_branch = FieldsConvLayer(name='fields')([x, y, z])

    mdl = Model(inputs=base.input, outputs=[tablehead_branch, frontco-
lumn_branch, fields_branch],
                name='ModelVGG19')

    return mdl


# funktion, welches die vorhersagen des netzes von shape (1, x, y) auf
(x, y) umrechnet
def create_masks(msk):
    tf_msk = tf.argmax(msk, axis=3)
    tf_msk = tf_msk[..., tf.newaxis]
    return tf_msk[0]


# starten des skripts
if __name__ == "__main__":
    main()
```

Anlage 4 - loadAndPredict_head_front_field.py:

```
import pytesseract
import tensorflow as tf
from trainModel_head_front_field import build_model, checkpoint_dir,
test_dataset_dir
import os
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
import cv2
```

```
def main():
    model = build_model()
    # test_dataset = tf.data.Dataset.load(test_dataset_dir)
    # lade trainiertes model
    latest_cp = tf.train.latest_checkpoint(checkpoint_dir)
    # expect_partial() unterdrückt warnungen, dass nicht alle gespei-
cherten variablen der checkpoints genutzt werden
    # dies hat den hintergrund, dass bei model.predict() die variablen
des optimizers für evtl. weitere Datensätze
    # nicht mehr benötigt werden
    model.load_weights(latest_cp).expect_partial()
    images = os.listdir(os.path.join("Berlin2020"))
    r = np.random.randint(len(images))
    img_fname = os.path.join("Berlin2020", "januar20-3.jpg")
    print(img_fname)
    masked_image = create_masked_image(img_fname, model)
    get_table(masked_image)


# ref: https://www.tensorflow.org/tutorials/images/segmentation
def create_masks(msk):
    tf_msk = tf.argmax(msk, axis=3)
    tf_msk = tf_msk[..., tf.newaxis]
    return tf_msk[0]


# ref: https://stackoverflow.com/questions/10469235/opencv-apply-mask-
to-a-color-image
def create_masked_image(image_fname, model):
    dim = (1024, 1024)
    orig_dim = Image.open(image_fname).size
    decoded = tf.io.decode_jpeg(tf.io.read_file(image_fname), chan-
nels=3)
    # Netzwerk erwartet batches, also wird durch expand_dims() eine
batch mit einem einzigen bild übergeben
    decoded = tf.expand_dims(decoded, axis=0)
    img = tf.cast(decoded, tf.float32)
    img = tf.image.resize(img, dim)
    img = tf.cast(img, tf.float32) / 255
    th_mask, fc_mask, f_mask = model.predict(img)
    th_mask, fc_mask, f_mask = create_masks(th_mask), cre-
ate_masks(fc_mask), create_masks(f_mask)

    # konvertiere vorhersagen zu bildern und skaliere
    th_mask_img = tf.keras.utils.array_to_img(th_mask).resize(o-
rig_dim)
    fc_mask_img = tf.keras.utils.array_to_img(fc_mask).resize(o-
rig_dim)
    f_mask_img = tf.keras.utils.array_to_img(f_mask).resize(orig_dim)

    th_mask_arr = np.array(th_mask_img)
    fc_mask_arr = np.array(fc_mask_img)
    f_mask_arr = np.array(f_mask_img)
    orig_image_array = np.array(Image.open(image_fname))

    combined_mask = th_mask_arr | fc_mask_arr | f_mask_arr
    masked_img = cv2.bitwise_and(orig_image_array, orig_image_array,
mask=combined_mask)

    image_list = [orig_image_array, th_mask_img, fc_mask_img,
f_mask_img, masked_img]
    plt.figure(figsize=(15, 10))
```

```
    names = ["Input Image", "Tablehead Mask", "Frontcolumn Mask",
"Field Mask", "Combined Masked Image"]
    for i in range(len(image_list)):
        plt.subplot(2, 3, i + 1)  # +1, da pyplot nur indizes größer 0
unterstützt
        plt.title(names[i])
        plt.imshow(image_list[i])
        plt.axis('off')
    plt.show()

    return masked_img


def get_table(masked_img):
    tessdata_dir_config = '--tessdata-dir "C:\\Program Files\\Tes-
seract-OCR\\tessdata"'
    text = pytesseract.image_to_string(masked_img, lang="deu", con-
fig=tessdata_dir_config)
    text = text
    print(text)


if __name__ == "__main__":
    main()
```

Anlage 5 - trainModel_table_col.py:

```
import tensorflow as tf
import os
import pandas as pd
import matplotlib.pyplot as plt
from keras.layers import Layer, Conv2D, UpSampling2D, Concatenate,
Conv2DTranspose, Dropout
from keras.applications import VGG16, VGG19, ResNet50, DenseNet201,
EfficientNetB5, Xception, NASNetLarge
from keras import Model
from keras.callbacks import TensorBoard, ModelCheckpoint
from keras.utils.vis_utils import plot_model

# setze verzeichnis der trainingsdaten und erstelle dataframe aus zu-
vor ersteller .csv datei
train_dir = os.path.join("stuff")
test_dataset_dir = "D:/datasets/test_data"
checkpoint_path = "D:/keras_models/table_col/learning_1e-03_epsi-
lon_1e-08_dropout_0.8/VGG19_1024/cp_{epoch:04d}_{val_loss:.4f}.ckpt"
checkpoint_dir = os.path.dirname(checkpoint_path)
log_dir = "D:/keras_models/table_col/learning_1e-03_epsilon_1e-
08_dropout_0.8/VGG19_1024_logs"


def main():
    # erstelle dataframe aus zuvor ersteller .csv datei
    train_df = pd.read_csv(os.path.join(train_dir, "csv", "data.csv"))

    # setzt das backend zurück - nur zur sicherheit
    tf.keras.backend.clear_session()
    # from_tensor_slices erstellt jeweils ein element pro reihe des
datensatzes
    dataset = tf.data.Dataset.from_tensor_slices((train_df['image_pa-
th'].values,
                                                  train_df["table-
mask_path"],
```

```
                                                    train_df["column-
mask_path"]))

    # erstellt train, validation und test datensatz
    dataset_size = len(dataset)
    train_size = int(0.8 * dataset_size)
    val_size = int(0.1 * dataset_size)
    test_size = int(0.1 * dataset_size)

    dataset = dataset.shuffle(buffer_size=42)
    train_dataset = dataset.take(train_size)
    test_dataset = dataset.skip(train_size)
    val_dataset = test_dataset.skip(test_size)
    test_dataset = test_dataset.take(test_size)

    train_dataset = train_dataset.map(map_function).batch(1)
    val_dataset = val_dataset.map(map_function).batch(1)
    test_dataset = test_dataset.map(map_function).batch(1)

    if not os.path.exists(test_dataset_dir):
        os.makedirs(test_dataset_dir)
    test_dataset.save(test_dataset_dir)

    model = build_model()
    model.summary()

    # gibt ein "flow-chart" des zuvor erstellten models aus
    plot_model(model, to_file="model_plot.png", show_shapes=True,
show_layer_names=True)

    # erstelle metriken für den verlust
    losses = {
        "table": tf.keras.losses.SparseCategoricalCrossen-
tropy(from_logits=True),
        "columns": tf.keras.losses.SparseCategoricalCrossen-
tropy(from_logits=True)
    }

    # möglichkeit, um die gewichtung der verluste anzupassen - stan-
dard 1.0 für alle
    lossWeights = {
        "table": 1.0,
        "columns": 1.0
    }

    # einstellen des adam algorithmus für das neuronale netz
    optim = tf.keras.optimizers.Adam(learning_rate=1e-03, epsilon=1e-
08)

    # erstelle score für die genauigkeit (sc_acc)
    sc_acc = tf.keras.metrics.SparseCategoricalAccuracy(name="sc_ac-
curacy")

    # kompiliert das model mit den zuvor eingestellten parametern
    model.compile(optimizer=optim,
                  loss=losses,
                  metrics=[sc_acc],
                  loss_weights=lossWeights)

    # zeige vorhersagen, bevor das netz trainiert worden ist
    # ref: https://www.tensorflow.org/tutorials/images/segmentation
    """
    for image, msks in val_dataset.take(1):
```

```
        table_mask, column_mask = msks['table'], msks['columns']
        pred_tab_mask, pred_c_mask = model.predict(image)
        show_imgs([image[0], create_masks(pred_tab_mask), cre-
ate_masks(pred_c_mask)])
    """

    # erstelle model callbacks für checkpoints und tensorboard für
spätere analyse der logs
    if not os.path.exists(checkpoint_dir):
        os.makedirs(checkpoint_dir)
    if not os.path.exists(log_dir):
        os.makedirs(log_dir)

    # checkpoints werden bei verbesserter validation loss gespeichert
    cp_callback = ModelCheckpoint(checkpoint_path, verbose=1,
save_best_only=True, monitor='val_loss',
                                  save_weights_only=True)
    # tensorboard callback für des spätere auswerten der logs
    tb_callback = TensorBoard(log_dir)
    callbacks = [cp_callback, tb_callback]

    model.fit(
        train_dataset, epochs=200,
        batch_size=len(train_dataset) // 2,
        steps_per_epoch=len(train_dataset) // 1,
        validation_data=val_dataset,
        validation_steps=len(val_dataset) // 1,
        callbacks=callbacks
    )


# funktion um eine liste von bilder anzeigen zu lassen
def show_imgs(image_list):
    plt.figure(figsize=(15, 10))
    names = ["Input Image", "Table Mask", "Column Mask"]
    for i in range(len(image_list)):
        plt.subplot(1, len(image_list), i + 1)  # +1, da pyplot nur
indizes größer 0 unterstützt
        plt.title(names[i])
        plt.imshow(tf.keras.utils.array_to_img(image_list[i]))
        plt.axis('off')
    plt.show()


# vorbereitung der trainingsdaten durch map_function()
def map_function(img, t_mask, col_mask):
    """
    nimmt jpg dateien aus dem dataset entgegen und dekodiert diese in
ein für tensorflow brauchbares format
    das bild selbst wird in RGB dekodiert - die masken werden mit nur
einem channel dekodiert
    alle bilder werden auf eine zuvor festgelegte größe (dim) skaliert
und die farben werden von 0 bis 1 normalisiert
    """
    dim = (1024, 1024)
    decoded = tf.io.decode_jpeg(tf.io.read_file(img), channels=3)  #
image in RGB
    img = tf.cast(decoded, tf.float32)
    img = tf.image.resize(img, dim)
    img = tf.cast(img, tf.float32) / 255

    t_msk_decoded = tf.io.decode_jpeg(tf.io.read_file(t_mask), chan-
nels=1)
```

```
    t_msk = tf.cast(t_msk_decoded, tf.float32)
    t_msk = tf.image.resize(t_msk, dim)
    t_msk = tf.cast(t_msk, tf.float32) / 255

    col_msk_decoded = tf.io.decode_jpeg(tf.io.read_file(col_mask),
channels=1)
    c_msk = tf.cast(col_msk_decoded, tf.float32)
    c_msk = tf.image.resize(c_msk, dim)
    c_msk = tf.cast(c_msk, tf.float32) / 255

    mask_dict = {
        'table': t_msk,
        'columns': c_msk
    }

    return img, mask_dict


class TableConvLayer(Layer):
    """
    ref: TableNet: Deep Learning model for end-to-end
    Table detection and Tabular data extraction from
    Scanned Document Images
    Shubham Paliwal, Vishwanath D, Rohit Rahul, Monika Sharma, Love-
kesh Vig
    TCS Research, New Delhi
    {shubham.p3, vishwanath.d2, monika.sharma1, rohit.rahul, love-
kesh.vig}@tcs.com
    Creating new custom layer classes for neural network
    ref: https://faroit.com/keras-docs/2.0.1/layers/writing-your-own-
keras-layers/#writing-your-own-keras-layers
    """
    def __int__(self, output_dim, **kwargs):
        self.output_dim = output_dim
        super(TableConvLayer, self).__int__(**kwargs)

    def build(self, input_shape):
        self.conv7 = Conv2D(512, 1, activation='relu', name='conv7tab-
le')
        self.upsample_conv7 = UpSampling2D(2)
        self.concat_p4 = Concatenate()
        self.upsample_p4 = UpSampling2D(2)

        self.concat_p3 = Concatenate()
        self.upsample_p3 = UpSampling2D(2)

        self.upsample_p3_2 = UpSampling2D(2)
        self.convtranspose = Conv2DTranspose(3, 3, strides=2, pad-
ding='same')
        super(TableConvLayer, self).build(input_shape)

    def call(self, x):
        x, y, z = x
        x = self.conv7(x)
        x = self.upsample_conv7(x)
        x = self.concat_p4([x, z])
        x = self.upsample_p4(x)
        x = self.concat_p3([x, y])
        x = self.upsample_p3(x)
        x = self.upsample_p3_2(x)
        x = self.convtranspose(x)

        return x
```

```
class ColumnConvLayer(Layer):

    def __int__(self, output_dim, **kwargs):
        self.output_dim = output_dim
        super(ColumnConvLayer, self).__int__(**kwargs)

    def build(self, input_shape):
        self.conv7 = Conv2D(512, 1, activation='relu', name='conv7co-
lumns')
        self.upsample_conv7 = UpSampling2D(2)
        self.concat_p4 = Concatenate()
        self.upsample_p4 = UpSampling2D(2)

        self.concat_p3 = Concatenate()
        self.upsample_p3 = UpSampling2D(2)

        self.upsample_p3_2 = UpSampling2D(2)
        self.convtranspose = Conv2DTranspose(3, 3, strides=2, pad-
ding='same')
        super(ColumnConvLayer, self).build(input_shape)

    def call(self, x):
        x, y, z = x
        x = self.conv7(x)
        x = self.upsample_conv7(x)
        x = self.concat_p4([x, z])
        x = self.upsample_p4(x)
        x = self.concat_p3([x, y])
        x = self.upsample_p3(x)
        x = self.upsample_p3_2(x)
        x = self.convtranspose(x)

        return x


def build_model():
    """
    erstellt das model mit hilfe eines bereits existierenden netzes
und dem anschließenden anhängen der zuvor
    erstellten output layer
    input shape kann je nach verwendetem netz angepasst werden, um in
vorhandenen speicher zu passen
    """
    tf.keras.backend.clear_session()
    input_shape = (1024, 1024, 3)

    base = VGG19(input_shape=input_shape, include_top=False,
weights='imagenet')

    # VGG 16/19
    end_layers_list = ['block3_pool', 'block4_pool', 'block5_pool']

    # ResNet50
    # end_layers_list = ['conv3_block4_3_conv', 'conv4_block6_3_conv',
'conv5_block3_3_conv']

    # DenseNet201
    # end_layers_list = ['pool3_pool', 'pool4_pool', 'relu']

    # Xception
```

```
    # end_layers_list = ['block10_sepconv1', 'block12_sepconv1', 
'block14_sepconv1']

    # NASNetLarge
    # end_layers_list = ['activation_238', 'activation_250', 'activa-
tion_259']

    # EfficientNetB5
    # end_layers_list = ['block6a_expand_activation', 'block7a_activa-
tion', 'top_activation']

    end_layers = [base.get_layer(i).output for i in end_layers_list]
    x = Conv2D(512, (1, 1), activation='relu')(end_layers[-1])  # last 
element of end_layers
    x = Dropout(0.8)(x)
    x = Conv2D(512, (1, 1), activation='relu')(x)
    x = Dropout(0.8)(x)

    # VGG16/19
    y = end_layers[0]
    z = end_layers[1]

    # EfficientNetB5
    # y = UpSampling2D(2)(end_layers[0])
    # y = Conv2D(1024, 1)(y)
    # z = UpSampling2D(2)(end_layers[1])
    # z = Conv2D(512, 1)(z)

    # Xception
    # y = UpSampling2D(2)(end_layers[0])
    # y = Conv2D(256, 1)(y)
    # z = end_layers[1]
    # z = Conv2D(512, 1)(z)

    # NASNetLarge
    # y = UpSampling2D(4)(end_layers[0])
    # y = Conv2D(1024, 1)(y)
    # z = UpSampling2D(2)(end_layers[1])
    # z = Conv2D(512, 1)(z)

    # DenseNet201
    # y = UpSampling2D(2)(end_layers[0])
    # z = UpSampling2D(2)(end_layers[1])

    # ResNet50
    # y = Conv2D(1024, 1)(end_layers[0])
    # z = Conv2D(512, 1)(end_layers[1])

    table_branch = TableConvLayer(name='table')([x, y, z])
    column_branch = ColumnConvLayer(name='columns')([x, y, z])

    mdl = Model(inputs=base.input, outputs=[table_branch, co-
lumn_branch],
                name='ModelVGG19')
    return mdl


# funktion, welches die vorhersagen des netzes von shape (1, x, y) auf 
(x, y) umrechnet
def create_masks(msk):
    tf_msk = tf.argmax(msk, axis=3)
    tf_msk = tf_msk[..., tf.newaxis]
    return tf_msk[0]
```

```
# starten des skripts
if __name__ == "__main__":
    main()
```

Anlage 6 - loadAndPredict_table_col.py:

```
import pytesseract
import tensorflow as tf
from trainModel_table_col import build_model, checkpoint_dir, test_da-
taset_dir
import os
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
import cv2


def main():
    model = build_model()
    # test_dataset = tf.data.Dataset.load(test_dataset_dir)
    # loading trained model
    latest_cp = tf.train.latest_checkpoint(checkpoint_dir)
    # expect_partial() unterdrückt warnungen, dass nicht alle gespei-
cherten variablen der checkpoints genutzt werden
    # dies hat den hintergrund, dass bei model.predict() die variablen
des optimizers für evtl. weitere Datensätze
    # nicht mehr benötigt werden
    model.load_weights(latest_cp).expect_partial()
    images = os.listdir(os.path.join("PDF_Out"))
    r = np.random.randint(len(images))
    img_fname = os.path.join("PDF_Out", "januar20.jpg")
    print(img_fname)
    masked_image = create_masked_image(img_fname, model)
    masked_image = cv2.cvtColor(masked_image, cv2.COLOR_BGR2GRAY)
    #plt.imshow(masked_image, cmap="gray")
    #plt.show()
    get_table(masked_image)


# ref: https://www.tensorflow.org/tutorials/images/segmentation
def create_masks(msk):
    tf_msk = tf.argmax(msk, axis=3)
    tf_msk = tf_msk[..., tf.newaxis]
    return tf_msk[0]


# ref: https://stackoverflow.com/questions/10469235/opencv-apply-mask-
to-a-color-image
def create_masked_image(image_fname, model):
    dim = (1024, 1024)
    orig_dim = Image.open(image_fname).size
    decoded = tf.io.decode_jpeg(tf.io.read_file(image_fname), chan-
nels=3)
    decoded = tf.expand_dims(decoded, axis=0)  # Network expects bat-
ches, so we're giving it a batch with a single image
    img = tf.cast(decoded, tf.float32)
    img = tf.image.resize(img, dim)
    img = tf.cast(img, tf.float32) / 255
    t_mask, c_mask = model.predict(img)
    t_mask, c_mask = create_masks(t_mask), create_masks(c_mask)
```

```
    t_mask_img = tf.keras.utils.array_to_img(t_mask).resize(orig_dim)
    c_mask_img = tf.keras.utils.array_to_img(c_mask).resize(orig_dim)

    t_mask_arr = np.array(t_mask_img)
    c_mask_arr = np.array(c_mask_img)
    orig_image_array = np.array(Image.open(image_fname))

    # logic AND to combine masks
    combined_mask = t_mask_arr & c_mask_arr
    masked_img = cv2.bitwise_and(orig_image_array, orig_image_array,
mask=combined_mask)

    image_list = [orig_image_array, t_mask_img, c_mask_img, mas-
ked_img]
    plt.figure(figsize=(15, 10))
    names = ["Input Image", "Table Mask", "Column Mask", "Combined
Masked Image"]
    for i in range(len(image_list)):
        plt.subplot(2, 2, i + 1)  # Adding 1, because pyplot only ac-
cepts indices greater than 0
        plt.title(names[i])
        plt.imshow(image_list[i])
        plt.axis('off')
    plt.show()

    return masked_img


def get_table(masked_img):
    tessdata_dir_config = '--tessdata-dir "C:\\Program Files\\Tes-
seract-OCR\\tessdata" --psm 13'
    text = pytesseract.image_to_string(masked_img, lang="deu", con-
fig=tessdata_dir_config)
    text = text
    print(text)


if __name__ == "__main__":
    main()
```

Selbstständigkeitserklärung

Hiermit erkläre ich, Lutz Gooren, dass ich die hier vorliegende Arbeit selbstständig und ohne unerlaubte Hilfsmittel angefertigt habe. Informationen, die anderen Werken oder Quellen dem Wortlaut oder dem Sinn nach entnommen sind, habe ich kenntlich gemacht und mit exakter Quellenangabe versehen. Sätze oder Satzteile, die wörtlich übernommen wurden, wurden als Zitate gekennzeichnet. Die hier vorliegende Arbeit wurde noch an keiner anderen Stelle zur Prüfung vorgelegt und weder ganz noch in Auszügen veröffentlicht. Bis zur Veröffentlichung der Ergebnisse durch den Prüfungsausschuss werde ich eine Kopie dieser Studienarbeit aufbewahren und wenn nötig zugänglich machen.

Geldern, 07.02.2023

Lutz Gooren