REPORT

im Modul Deep Learning

des Studienganges Elektrotechnik
in der Studienrichtung Fahrzeugelektronik
mit Schwerpunkt Embedded IT

an der Dualen Hochschule Baden-Württemberg Ravensburg
- Campus Friedrichshafen

von

Hanna Ludes

13. Juni 2022

Bearbeitungszeitraum	9 Wochen
Matrikelnummer	1292658
Kurs, Dozent	TFE19-2, Mark Schutera
Studiengangsleitung	Prof. Dr.-Ing. Thomas Kibler

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Der Begriff *Deep Learning* beschreibt eine spezielle Methode zur Informationsverarbeitung und bezeichnet ein Teilgebiet des *Machine Learnings* mit Fokus auf Analyse von großen Datenmengen unter Einsatz von künstlichen neuronalen Netzen.

Künstliche neuronale Netze werden schon seit den frühen 1940er Jahren entwickelt und heutzutage meist dazu eingesetzt, um große Datensätze zu analysieren und auf diese Weise Bilder zu erkennen, Texte zu verstehen oder Entscheidungen treffen zu können. Die Funktionsweise eines künstlichen neuronalen Netzes ist weitgehend vom biologischen Neuronennetz inspiriert und arbeitet somit ähnlich wie das menschliche Gehirn. Nur können so deutlich größere Datenberge viel schneller untersucht werden als es Menschen jemals möglich wäre. Die betreffenden Daten werden demnach zuerst extrahiert und anschließend analysiert, um schließlich zu einem Fazit zu gelangen oder eine Prognose zu stellen. Zwar schaffen es *Machine Learning*-Algorithmen inzwischen nahezu unlösbare Probleme zu lösen, doch sind sie aufgrund ihrer komplexen Architektur und großen Anzahl von Modell-Parametern extrem zeit- und rechenintensiv, weshalb das Themenfeld gerade infolge stark wachsender Rechenkraft durch Einsatz von Grafikkarten zunehmend an Bedeutung gewinnt.

So versucht man inzwischen *Deep Learning* überall dort einzusetzen, wo große Datenmengen nach Mustern und Trends untersucht werden sollen, wie beispielsweise im Rahmen von Gesichts-, Objekt- oder Spracherkennung. Auch in Zusammenhang mit der Entwicklung des autonomen Fahrens gibt es viele Bestrebungen, sich *Deep Learning* zu Nutze zu machen und auf diese Weise das Bewusstsein für drohende Gefahren zu schärfen. Während herkömmliche Lösungen zur Verhaltensvorhersage nur für einfache Fahrszenarien mit kurzen Zeithorizonten geeignet sind, ist man mithilfe neuronaler Netze in der Lage, auf Grundlage vergangener Beobachtungen und aktueller Umgebungsparameter das zukünftige Verhalten eines Fahrzeugs oder in der Nähe befindlicher Verkehrsteilnehmer vorauszusagen. Jedoch ist der Einsatz von KI mit Blick auf erforderliche Kontrollprozesse für autonomes Fahren und sicherheitskritische Anwendungen noch nicht ausreichend untersucht, weshalb derzeit v.a. die Validierung der Datenmodelle eine Herausforderung darstellt.

1.2 Zielsetzung

Diese Arbeit dient als ergänzende Dokumentation zum eingereichten Programmentwurf im Rahmen der Untersuchung von Sicherheitskonzepten für *Deep Learning* Anwendungen an der DHBW Ravensburg und gliedert sich in insgesamt drei Kapitel. Während im vorangegangenen Abschnitt zunächst auf die Motivation und Zielsetzung des Programmentwurfs im Hinblick auf die Validierung eines *Deep Learning* Modells eingegangen worden ist, soll in den darauffolgenden Kapiteln die zugrundeliegende Problematik einer derartigen Validierung erläutert sowie die Erarbeitung entsprechender Lösungsansätze dargelegt werden.

Wie bereits angedeutet wird *Deep Learning* in der Praxis oftmals zur Erkennung und Klassifizierung von Bildern verwendet. Ausgehend eines großen Volumens an Trainingsdaten soll die Maschine lernen, Muster im Datensatz zu erkennen, um dann bspw. auf Basis im Fahrzeug generierter Sensordaten eine hinreichend gute Aussage treffen und Verkehrszeichen unterscheiden zu können. Ungeachtet der Qualität bzw. dem Trainingsstand des Datenmodells und der tatsächlich im Fahrzeug eingesetzten Sensorik, ist die erreichte Vorhersage-Güte jedoch erheblich von der ausführlichen und sorgfältigen Beurteilung der erzielten Ergebnisse abhängig. Ziel ist es daher in erster Linie eine verbesserte Validierung anhand von Weiterentwicklung der implementierten Softwarefunktionen zu erreichen. Vor diesem Hintergrund sollen im Folgenden zwei Erweiterungen des zur Verfügung gestellten *Deep Learning*-Projektes vorgestellt werden, um auf diese Weise effektiv zur Optimierung der *validation_pipeline* beitragen zu können. Dazu wird zunächst eine Funktionserweiterung zur Bestimmung der Modellgenauigkeit je Klasse entwickelt, die in das vorhandene Skript zur Validierung des Datenmodells eingebunden werden kann. Darüber hinaus soll eine zweite Funktion zur automatisierten Suche und Speicherung von passenden Bilddateien im Internet realisiert werden. Diese können dann wiederum als zusätzliche Datengrundlage zur Validierung genutzt werden, sodass hierbei nicht immer dieselben Datensätze zum Einsatz kommen, sondern kontinuierlich wechselnde Bilder verwendet werden.

2 Problemstellung

Im Grunde stellt das Training von künstlichen neuronalen Netzen, d.h. das Schätzen der im Modell enthaltenen Parameter, ein großes, mehrdimensionales und nichtlineares Optimierungsproblem dar. Hauptschwierigkeit bei der Lösung derartiger Probleme ist es, sicherzustellen, dass es sich bei einem ermittelten Optimum wirklich um das globale und nicht lediglich eines von mehreren lokalen Optima handelt. Häufig kann eine globale Lösung nur durch zeitaufwändige Näherung in Form von vielfacher Wiederholung der Optimierung mit immer neuen Startwerten erreicht werden. Des Weiteren kann es im Laufe des Trainings zu einer potenziellen Überanpassung der gesammelten Daten kommen. Denn künstliche neuronale Netze können je nach Netzspezifikation dazu neigen, die Trainingsdaten infolge einer Übergeneralisierung (*overfitting*) einfach "auswendig zu lernen". In diesem Fall ist das Netz dann nicht mehr in der Lage, auf neue Daten zu verallgemeinern, was es ebenfalls zu vermeiden und in der Validierung zu überprüfen gilt. Auch hat die Art und Weise wie die Daten dem Netz präsentiert werden, großen Einfluss darauf, ob das Problem überhaupt von einem Netz gelernt werden kann und in welcher Geschwindigkeit dies erfolgt. Die Datenverarbeitung ist also umso erfolgreicher, je präziser ein Problem allein durch geeignete Vorverarbeitung an das Netz übergeben wird (vgl. [Wik22]). Aus diesem Grund wurde zu Beginn der Gruppenarbeit zunächst die Ordnerstruktur des Trainingsdatensatzes angepasst. Aufgefallen ist die Thematik bei Verwendung von *Batch_2*, als der eigentliche Ordner Nr.12 als Klasse Nr.4 angelegt worden ist. Denn die Ordner eines *SafetyBatches* mit Nummerierung der Form 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,... werden beim Training des Netzes (anders als intuitiv angenommen) von der Maschine stattdessen in der Reihenfolge 0, 1, 10, 11, 12,... durchlaufen. Abhilfe konnte durch neue Nummerierung der Trainingsdaten-Ordner nach in Abbildung 1 dargestelltem Schema geschaffen werden, wobei das Training des Modells danach erneut von Grund auf wiederholt.

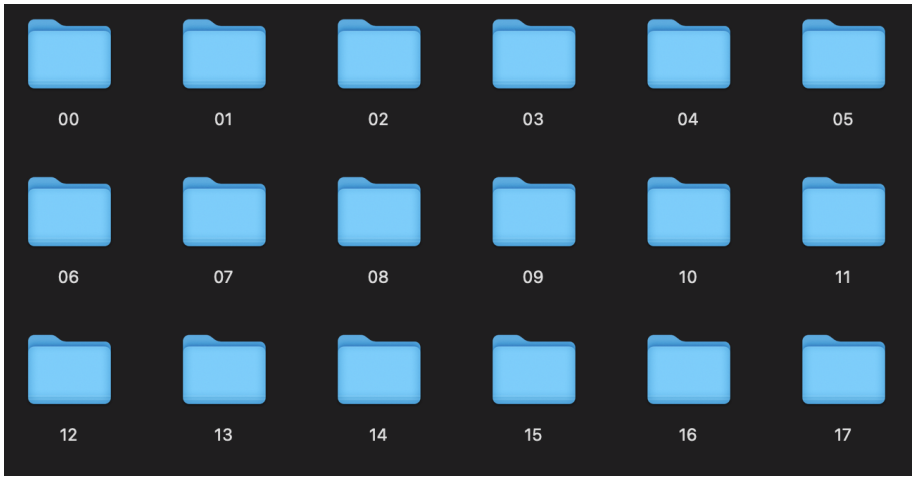

Abbildung 1: *Überarbeitete Ordnerstruktur für Trainingsdatensatz mit angepasster Nummerierung*

Zusätzlich zur Überarbeitung der Ordnerstruktur des Trainingsdatensatzes ist darüber hinaus noch eine weitere Vorbereitungsmaßnahme in der Gruppe getroffen worden: Da bei Ausgabe der *Ground truth* im Zuge der Validierung ebenfalls eine automatische Nummerierung nach der Ordnerzahl der *Batch*-Dateien stattfindet, müssen die darin enthaltenen Bilder im Vorfeld jeweils manuell dem passenden Ordnern zugewiesen werden. Um dies zu vermeiden, soll die *Ground truth*-Angabe nicht länger automatisch analog der Ordnernummerierung erfolgen, sondern stattdessen besser der jeweilige Ordnernamen aus der verwendeten *Batch*-Datei genutzt werden. Auf diese Weise können die Ergebnisse einfacher und schneller den Trainingsdaten zugeordnet werden, wodurch eine verbesserte Nachvollziehbarkeit bei der Beurteilung der Validierungsgüte möglich ist. Das bedeutet, die *Ground truth* erhält zukünftig die gleiche Nummerierung wie das eingesetzte *Batch* oder wird im Idealfall bei Weiterentwicklung dieses Ansatzes sogar mit der korrekten Bezeichnung (*string*) desjenigen Ordners versehen, in dem das jeweilige Verkehrsschild im Trainingsdatenverzeichnis abgelegt ist. Darüber hinaus ist es sehr schwierig, das Verhalten des Systems bei einer unendlichen Anzahl von möglichen, aber unbekannten Situationen vorherzusagen, weshalb die Verwendung unterschiedlicher und stetig wechselnder Datensätze in der Validierung ebenfalls von essentieller Bedeutung ist. Dazu muss eine Vielzahl und Breite Palette an Trainings- und Validierungsdaten gesammelt oder manuell erzeugt werden, was sich ebenfalls als mühsam und zeitaufwendig erweist. Hier gilt es zu verhindern, dass das Netz bestimmte Eigenschaften der Muster erlernt, die bei Betrachtung des Trainingsdatensatzes zwar mit dem Ergebnis in irgendeiner Weise korrelierend sind, aber in anderen Situationen nicht zur Entscheidung herangezogen werden können oder sollten. Weißt bspw. die Helligkeit der zum Training herangezogenen Bilder ein gewisses Muster auf, dann “achtet” das Netz u.U. nicht mehr auf die gewünschten Eigenschaften, sondern klassifiziert die Daten nur noch aufgrund seiner Erkenntnisse bzgl. Helligkeit.

Die zentrale Aufgabe besteht im Falle dieser Ausarbeitung also nicht aus dem Entwurf oder Training des künstlichen neuronalen Netzes, sondern liegt in der Optimierung der *validation_pipeline*, die im Zuge dieses Programmentwurfs durch eigene Ideen erweitert werden soll. So war es zu Beginn der Vorlesung z.B. mithilfe dieser Pipeline noch nicht möglich, die Ergebnisse der Validierung adäquat zu visualisieren oder automatisierte Tests durchzuführen. Im Rahmen dieser Arbeit soll die *validation_pipeline* nun so verbessert oder ergänzt werden, dass eine fundierte Aussage bezüglich der Modellgenauigkeit und Güte der Validierung gegeben werden kann sowie eine automatisierte Beschaffung von zusätzlichen Bildern möglich ist. Dazu soll einerseits die Genauigkeit des Modells nicht länger nur insgesamt für das jeweilige *Batch* sondern ebenfalls spezifisch pro Klasse berechnet und andererseits ein sog. *crawler*-Algorithmus implementiert werden.

3 Implementierung

Um die nachfolgenden Programme und Funktionserweiterungen nutzen zu können, sollten neben Python 3.9 ebenso alle in der Datei *requirements.txt* aufgeführten Package-Installationen und Importe vorgenommen werden. Es wird darauf hingewiesen, dass die Installation von *protobuf<=3.20.1* dabei nicht zwingend erforderlich sein muss, sondern in erster Linie vorsorglich aufgrund jüngster Probleme in Zusammenhang mit *Tensorflow* inkludiert worden ist (siehe [RTD22]).

3.1 Class Accuracy

Das Python Skript *class_accuracy.py* definiert eine Funktion, die verwendet werden kann, um die individuelle Genauigkeit jeder Klasse innerhalb einer *Batch*-Datei zu ermitteln. Der Funktionsaufruf erfolgt innerhalb der *validation_pipeline*, wodurch ein modularer Aufbau gegeben ist. Zur Ausführung der Funktion ist es also ausreichend, nach entsprechender Ergänzung den herkömmlichen Befehl zum Start der *validation_pipeline* wie folgt aufzurufen:

```
python ./validation_pipeline.py
```

Abbildung 2: *Kommandozeilen-Befehl zum Aufruf der validation_pipeline*

An dieser Stelle sei besonders erwähnt, dass die Dateien *accuracy_log.py* und *signnames.txt* Teil des Programmentwurfs von Sarah Theuerkauf sind und ausdrücklich **nicht** zum Umfang der hier beschriebenen Software-Implementierung dazugehören. Die beiden Dateien werden nur inkludiert, da zur Bestimmung der Klassengenauigkeit zunächst Sarahs *accuracy_logger* importiert wird, um im weiteren Verlauf von der dort definierten *Mapping*-Funktion Gebrauch zu machen. Dies ist erforderlich, um Zugriff auf die konkrete Anzahl der Klassen sowie die Liste der prognostizierten und tatsächlichen Verkehrszeichen zu erhalten.

Im Anschluss daran wird ein zweiter, interner Zähler erzeugt (*internal_cnt*), der mit jeder korrekten Vorhersage inkrementiert wird und daher maximal der Gesamtanzahl der im *Batch* vorhandenen Ordner bzw. Klassen entspricht. Diese Zählvariable repräsentiert demnach die Anzahl der richtigen Prädikationen pro Klasse und soll später jeweils zur Berechnung der individuellen Klassengenauigkeit genutzt werden. Daraufhin werden mehrere, teilweise ineinander verschachtelte *for*-Schleifen mit jeweils eigener Laufvariable benötigt, mit deren Hilfe die *Ground truth* für die Klassenanzahl im *Batch* durchlaufen wird, um dabei die Validierungsergebnisse an jeder Stelle zu vergleichen und um auf diese Weise die Anzahl der richtigen Vorhersagen für jede Klasse zu ermitteln. Wäh-

rend die Laufvariable n dabei die Nummerierung der verwendeten Klassennamen (*test labels*) darstellt und somit die jeweilige Nummer des gerade verglichenen Bildes angibt, repräsentiert die Laufvariable p jeweils die zugehörige bzw. erwartete Klassennummer für dieses Verkehrszeichen innerhalb der Gesamtanzahl der Klassen in der betreffenden *Batch*-Datei. Der interne Zähler wird beim Schleifendurchlauf immer dann inkrementiert, wenn die Vorhersage korrekt ist und gleichzeitig ebenso mit der erwarteten Klassennummer übereinstimmt. Schließlich werden zum Schluss mithilfe einer dritten *for*-Schleife nochmal alle Klassen des *SafetyBatches* durchlaufen, um ihre individuelle Genauigkeit anhand der Division der korrekt getätigten Vorhersagen durch die Gesamtzahl der Klassen zu berechnen und für den Nutzer in der Konsole auszugeben. Die Laufvariable c beschreibt hier die jeweilige Nummer der durchlaufenen Klasse im *Batch*.

3.2 Image Crawler

Um das *Machine Learning* Modell nach umfassendem Training angemessen zu validieren, sollten zur Validierung nicht genau dieselben Bilder verwendet werden, die zuvor bereits im Training zum Einsatz kamen. Um zu verhindern, dass der Benutzer viel Zeit und Mühe mit der manuellen Beschaffung einer großen Menge von Bildern verschwendet, sollte die Suche und Speicherung einer Vielzahl von Bildern aus dem Internet vollständig automatisiert werden. Das Skript *carwler.py* ist dazu in der Lage, indem es Bilder anhand von vordefinierten Stichworten automatisch aus dem Internet herunterlädt und im gewünschten Verzeichnis auf dem PC ablegt. Zu diesem Zweck wird das Paket *icrawler* verwendet, das Methoden für mehrere gängige Suchmaschinen bereitstellt. Die Suchbegriffe können bei Ausführung des Skripts entweder individuell vom Benutzer mitgegeben oder aus einer Textdatei gelesen werden. Zudem kann der Benutzer optional zwischen *Bing* und *Google* als genutzte Suchmaschine wählen und zusätzlich (sofern gewünscht) alle heruntergeladenen Bilder automatisch auf ein gewünschtes Format verkleinern. Hierzu wird das Paket *cv2* verwendet, das die Verkleinerung der Bilddateien ermöglicht. Mit diesem Paket könnte darüber hinaus ebenso leicht und schnell eine Umwandelung in Graustufe erreicht werden (siehe [ste21]).

Das Skript nimmt dabei bis zu vier Parameter entgegen, die bei Aufruf durch die Kommandozeile übergeben werden können. Zum Starten des *crawler*-Algorithmus wird der folgende Befehl ausgeführt:

```
python ./crawler.py -n [max. number of images] -r [no, yes] -e [bing, google] -k [optional: keyword]
```

Abbildung 3: *Kommandozeilen-Befehl zum Aufruf des crawler-Algorithmus*

Das Argument -n legt die maximale Anzahl zu beschaffender Bildern fest und muss dem Programm zwingend mitgegeben werden; ohne diese Angabe ist ein Starten nicht möglich. Die Obergrenze der erfassten Bilder beträgt dabei theoretisch 1000 Dateien, wobei aufgrund von bekannten Kommunikationsthemen zwischen der *icrawler*-Erweiterung und den verschiedenen Suchmaschinen effektiv meist nur etwa 700 Bilder abgespeichert werden. Die beiden Argumente -e und -r sind prinzipiell optional zu übergeben, da sie über eine default-Definition verfügen. Während mithilfe von -e als Suchmaschine zwischen *Bing* und *Google* ausgewählt werden kann, wird durch Eingabe von -r über die mögliche Größenanpassung der entschieden. Im Standardfall wird *Bing* zur Suche verwendet und keine Verkleinerung der Bilddatei vorgenommen. Das Argument -k ist grundsätzlich ebenso optional, muss jedoch zwangsläufig zur Ausführung des Skriptes mitgegeben werden, sofern keine Textdatei *keywords.txt* mit entsprechenden Suchbegriffen hinterlegt ist. Denn zur erleichterten Benutzung und noch schnelleren Bild-Beschaffung besteht die Möglichkeit, eine größere Menge von Suchbegriffe bereits im Vorfeld in einer externen Textdatei festzulegen, auf welche dann durch zeilenweises Auslesen vom Programm zugegriffen werden kann (siehe [lin20]). Dazu muss zu Beginn zunächst überprüft werden, ob eine solche Datei existiert. Ist die Datei nicht existent oder aus irgendeinem Grund nicht lesbar, wird ein *FileNotFoundError* geworfen. Falls die Datei existiert, darin allerdings keine Suchwörter definiert sind, wird der Nutzer zur Festlegung von Suchbegriffen aufgefordert (siehe [Sta21]). Im Normalfall werden alle dort angegebenen *keywords* bei Ausführung des Skriptes automatisch eingelesen und jeweils zur Suche verwendet. Findet trotz Vorhandensein der passenden Datei eine Eingabe von *keywords* über die Konsole statt, werden diese immer über den Inhalt der Textdatei priorisiert. Zur Eingabe mehrere *keywords* sind diese durch Komma zu trennen. Nach korrektem Funktionsaufruf in der Konsole wird daraufhin für jedes hinterlegte oder manuell mitgegebene *keyword* vom Programm ein *Crawl* durchgeführt und mithilfe der gewünschten Suchmaschine nach den entsprechenden Bildern gesucht. Diese werden anschließend vom *icrawler*-Package auf Basis bestimmter Parameter ausgewählt und erfasst. Neben Gegenprüfung der erlaubten Dateitypen *.jpg* oder *.png* wird im Zuge der hier verwendeten Implementierung ebenfalls nach Größe der Bilddatei “large” und deren Lizenz “commercial, modify” gefiltert. Nach erfolgreicher Suche werden die ausgewählten Bilddateien daraufhin automatisch heruntergeladen und abgespeichert. Zur Speicherung der Bilder wird außerdem überprüft, ob der angegebene Speicherpfad bereits existent ist oder erst noch erzeugt werden muss. Dabei handelt es sich wieder um einen separaten Ordner im gleichen Verzeichnis (*icrawled*), der ggf. automatisch neu erstellt wird. Innerhalb von *icrawled* finden sich weiterer Unterordner, die ihrerseits wiederum eigens je *keyword* angelegt und benannt werden, um darin die zugehörigen Bilder speichern zu können (siehe [Dha22]).

Literaturverzeichnis

[Dha22] Rajendra Dharmkar. *How can I create a directory if it does not exist using Python?* 25.05.2022. URL: %5Curl%7Bhttps://www.tutorialspoint.com/How-can-I-create-a-directory-if-it-does-not-exist-using-Python%7D.

[lin20] linuxtut.com. *Made icrawler easier to use for machine learning data collection*. 2020. URL: %5Curl%7Bhttps://linuxtut.com/en/64deae619f62ddb61558/%7D.

[RTD22] RTD. *Jupyter notebook in vscode with virtual environment fails to import tensorflow*. 2022. URL: %5Curl%7Bhttps://stackoverflow.com/questions/72411825/jupyter-notebook-in-vscode-with-virtual-environment-fails-to-import-tensorflow%7D.

[Sta21] Stack Vidhya. *How To Check If File Exists In Python?* 2021. URL: %5Curl%7Bhttps://www.stackvidhya.com/check-if-file-exists-in-python/%7D.

[ste21] steviesblog.de. *Image Crawler Python*. 2021. URL: %5Curl%7Bhttps://steviesblog.de/blog/2021/02/15/image-crawling-python/%7D.

[Wik22] Wikipedia, Hrsg. *Künstliches neuronales Netz*. 2022. DOI: \url{Page}. URL: %5Curl%7Bhttps://de.wikipedia.org/w/index.php?title=K%C3%BCnstliches_neuronales_Netz&oldid=222521209%7D.