Hochschule Pforzheim
Fakultät für Technik

Masterstudiengang Embedded Systems

Ausarbeitung

Künstliche Intelligenz

Prüfer Prof. Dr. rer. pol. Raphael Volz
Autor Anna-Lena Marx
Matrikelnummer 317593

Abgabedatum 5. August 2018

Inhaltsverzeichnis

1 Einleitung

Künstliche Intelligenz und maschinelles Lernen liegen in der Informatik momentan sehr im Trend, wie ein Blick auf den *Gartner Hype Cycle* vom Juli 2017 1 [5] zeigt. Neben der Anwendung von künstlicher Intelligenz z.B. in virtuellen Assistenten oder autonomer Fahrzeuge, ist auch das Gebiet des maschinellen Lernens allgemein, aber auch im Speziellen mit Deep Learning und Deep Reinforcement Learning vertreten. Zwar befinden sich beide genannten laut der Grafik mit Stand 2017 noch in einer relativ frühen Phase, aber es ist zu erwarten, dass beide Technologien die Industrie in absehbarer Zeit in der Breite erreichen. Laut der Abbildung dürfte Deep Learning erst in den nächsten Jahren tatsächlich produktiv in der Industrie zu finden sein, aber das Interesse dieser scheint groß. Die Idee hinter Deep Learning, beziehungsweise künstlichen neuronalen Netzen, existiert in der Informatik schon seit den 1940er Jahren [3] doch erst mit der Rechenleistung heutiger Computer kann sich diese auch außerhalb der Forschung durchsetzen. Damit wächst allerdings auch der Bedarf der Industrie an Experten schneller an der Lehrplan der Hochschulen daran angepasst werden kann und auch in anderen Fachgebieten wird immer mehr ein gewisses Grundwissen erwartet um eine reibungslose Zusammenarbeit zwischen den Disziplinen zu erreichen.

Mit der Lehrveranstaltung *Künstliche Intelligenz* der Hochschule Pforzheim wurde eine solche Grundlage geschaffen. Die vorliegende Arbeit dokumentiert die exemplarische Anwendung der erlernten theoretischen Grundlagen im Bereich des Deep Learnings. Im folgenden soll mit Hilfe des Frameworks *Keras* ein neuronales Netz erstellt und anhand des belgischen Verkehrszeichen Datensatzes trainiert und evaluiert werden.

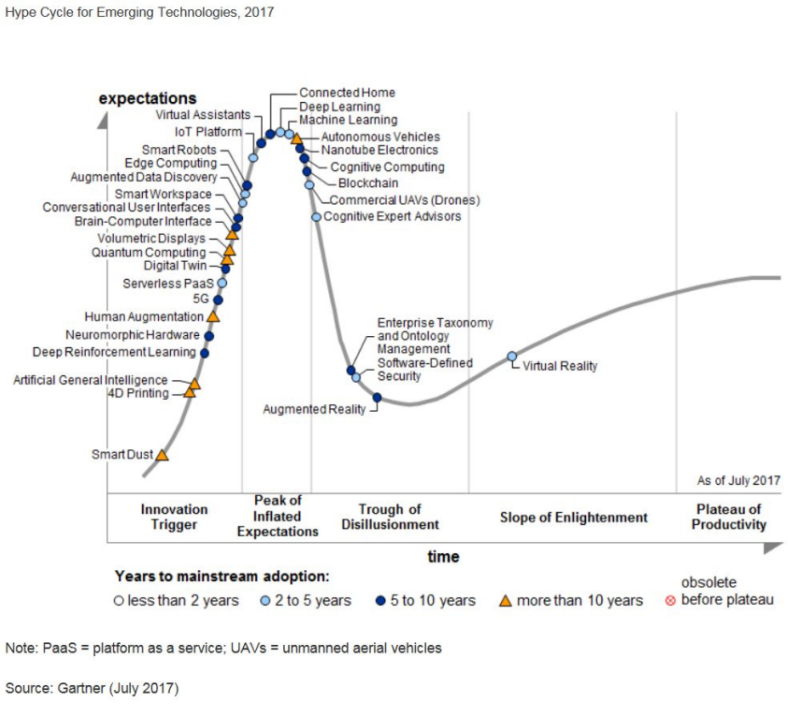


Abbildung 1: Gartner Hype Cycle 2017 [5]

1.1 Quellcodes

Alle Dateien und Quellcodes, die notwendig sind, um diese Arbeit nachzuvollziehen sind in dem GitHub Repository `https://github.com/Allegra42/ki-keras-belgian-traffic-sign` abgelegt. Die benötigten Abhängigkeiten können z.B. über `virtualenv` und den Befehl `pip install -r requirements.txt` installiert werden. Das Training der Modelle kann über den Befehl `python3 Keras.py` gestartet werden. Dabei wird angenommen, dass die Trainings- und Testdaten bereits heruntergeladen und unter dem Pfad `/data/` entpackt sind. Ist dies nicht der Fall kann im Quellcode die Zeile 222, `download_data()` in der Methode `setup_data()`, einkommentiert werden. Um ein Modell auf beliebige, entsprechend der Bildausschnitte der Trainingsbilder zugeschnittene, Bilder anzuwenden, dient der Befehl `python3 ClassifyImages.py -m <Pfad/zum/Modell> -i <Pfad/zum/Bild>`. Alle Anwendungen, die im Rahmen dieser Arbeit erstellt wurden, sind nur auf Linux entwickelt und getestet worden. Bei einem Systemwechsel müssen die Pfadangaben in den Quellcodes und den Aufrufen der Anwendungen entsprechend angepasst werden.

2 Maschinelles Lernen von belgischen Verkehrszeichen

Diese Arbeit basiert auf dem belgischen Verkehrszeichen Datensatz (Belgian Traffic Sign Dataset) [6], der zum Training und zur Validierung des Modells genutzt werden soll. Der Datensatz für Training und Testing ist frei unter `https://btsd.ethz.ch/shareddata/BelgiumTSC/` verfügbar.

Der Datensatz beeinhaltet Verkehrszeichen aus 62 Kategorien. Es handelt sich dabei also um eine Klassifikationsaufgabe, bei der jeweils ein einzelnes Bild eines Verkehrszeichen richtig zugeordnet werden soll. Im Rahmen dieser Hausarbeit sollen mit Hilfe des Frameworks *Keras* mehrere Modelle von neuronalen Netzen entwickelt, mit dem Trainingsdatensatz trainiert und abschließend anhand ihrer Leistung aber auch der Architektur verglichen und bewertet werden. Abschließend soll die Leistungsfähigkeit der Modelle durch Bilder, die nicht dem Trainings- oder Testdatensatz angehören überprüft werden.

3 Keras

Keras wird als High-Level API für neuronale Netze beschrieben, die eine sehr einfache und schnelle Entwicklung dieser zum Ziel hat [2]. Das Framework selbst ist in Python geschrieben und nutzt TensorFlow [1] (Google), CNTK [2] (Microsoft) oder Theano [3] (MILA, eingestellt) als Backend-Bibliothek für die eigentlichen Berechnungen. Keras selbst ist eine eigenständige Bibliothek und wird als solche entwickelt, gehört aber seit dem Release von TensorFlow 1.4 zur TensorFlow Core API. Wie auch TensorFlow ist Keras Open-Source und auf GitHub einsehbar. Innerhalb dieser Arbeit wird Keras in der Version 2.2.2 mit TensorFlow 1.9.0 als Backend auf einer Linux-Plattform genutzt.

Die Wahl von Keras als Framework ist durch die Klarheit und Verständlichkeit der API begründet. Dennoch steht mit TensorFlow eine bekannte, gut getestete und sehr mächtige Implementierung für maschinelles Lernen zur Verfügung. Für die Entwicklung und das Training wurde eine Intel(R) Core(TM) i7-7820HQ CPU mit 8 virtualisierten Kernen und 16 GB RAM verwendet. Auf eine für den Befehlssatz der CPU oder eine GPU optimierte Implementierung von TensorFlow wurde dabei verzichtet.

4 Der Datensatz

Der belgische Verkehrszeichen Datensatz unterscheidet zwischen 62 verschiedenen, aber teils sehr ähnlichen Kategorien. In den Abbildungen 2 und 3 ist dies anhand von Originalbildern aus den Trainingsdaten dargestellt. Während Bild (a) aus der Abbildung

[1] `https://www.tensorflow.org/`
[2] `https://github.com/Microsoft/cntk`
[3] `https://github.com/Theano/Theano`

2 zwar Ähnlichkeiten mit den anderen beiden aufweist, ist es durch die Form klar abzugrenzen. Zwischen den Bildern (b) und (c) ist dies deutlich schwieriger. Bei Bild (b) handelt es sich um ein Gefahrenzeichen, während (c) ein rein informativen Charakter hat [1]. Der Unterschied in Farbe und Form ist, gerade bei der Nutzung von Bildern in Graustufen nicht sehr groß. In der Anwendung einer künstlichen Intelligenz im Straßenverkehr ist die korrekte Unterscheidung zwischen Information und Gefahrenzeichen durchaus relevant. Noch deutlicher wird die Schwierigkeit in der Klassifikation in den unter Abbildung 3 dargestellten Originalbildern. Es handelt sich tatsächlich um das selbe Zeichen, dass durch seine Ausrichtung eine gewisse Bedeutung erlangt. Der Datensatz unterscheidet zwischen *mandatory direction (ahead)* (a) und *mandatory direction (others)* (b und c). Damit ist die Klassifikation in diesem Fall von der Ausrichtung der Eingangsbilder abhängig. Schon bei einer leichten Verzerrung ist es denkbar, dass ein Modell zur Klassifikation die falsche Klasse auswählt. Dies wäre in der realen Welt aber auch bei einem menschlichen Fahrer leicht möglich, zeigt aber gut eine Schwierigkeit die aus dem Datensatz selbst resultiert.

(a) (b) (c)

Abbildung 2: (a) Klasse 0 *Uneven road*, (b) Klasse 1 *speed bump* , (c) Klasse 59 *speed bump*

(a) (b) (c)

Abbildung 3: (a) Klasse 34 *mandatory direction (ahead)*, (b) Klasse 35 *mandatory direction (others)* , (c) Klasse 35 *mandatory direction (others)*

Eine andere solche Schwierigkeit ist die Größe und Aufteilung des Datensatzes. Auf der schon genannten Website kann jeweils ein Trainings- und ein Testdatensatz heruntergeladen werden. Ein klar abgegrenzter Datensatz zur Validierung ist nicht verfügbar.

Durch die Größe von Trainings- und Testdatensatz mit 4570 bzw. 2520 Bildern aufgeteilt sehr gering ist, kann nicht ohne weiteres ein Validierungsdatensatz abgetrennt werden. Auch die Aufteilung der Bilder im größeren Trainingsdatensatz erschwert dies. Abbildung 4 stellt die Problematik dabei gut dar. Während für 16 Klassen mindestens 100 Beispielbilder, anhand derer das neuronale Netz trainiert werden kann, vorhanden sind, müssen für 5 Klassen unter 10 Beispielbilder genügen. Machine Learning Verfahren leben von einer möglichst großen Datenmenge für das Training, es ist daher anzunehmen, dass für die 5 Klassen mit weniger als 10 Beispielbilder schlechtere Vorhersage-Ergebnisse als für die 16 Klassen mit mehr als 100 Beispielbildern erzielt werden. Zudem ist es auch hierdurch äußerst schwierig und wenig zielführend ein weiteren, distinkten Datensatz zur Verifikation abzugrenzen.

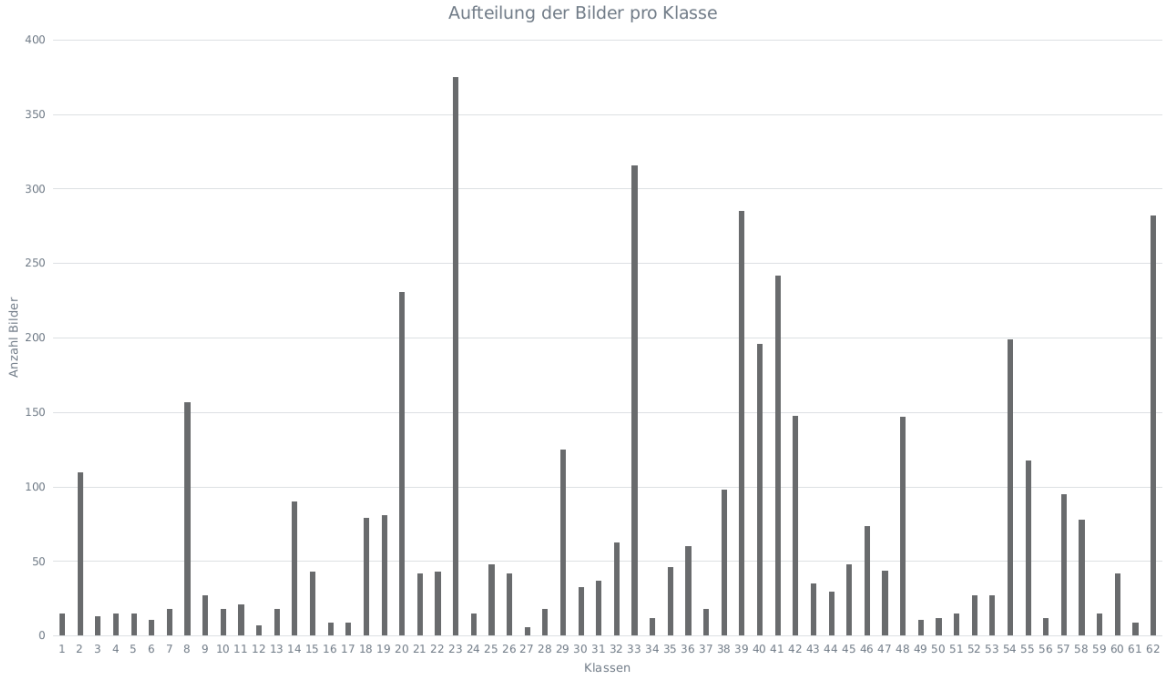


Abbildung 4: Verteilung der Bilder pro Klasse für den Trainingsdatensatz

In den Abbildungen 2 und 3 werden bereits Beispielbilder aus dem Trainingsdatensatz dargestellt. Die Bilder liegen als Farbbilder (RGB) im PPM (Portable Pixmap) Format, aber in unterschiedlichen Bildgrößen vor. Die Bilder sind für Trainings- und Testdatensatz jeweils in Unterordnern, welche die Klassen repräsentieren, angeordnet. Zusätzlich befindet sich in jedem Unterordner eine CSV-Datei, die zusätzliche Informationen, wie die Breite, die Höhe, die Koordinaten der Region of Interest (ROI) und die Klassennummer, zu jedem Bildnamen enthält. Die Informationen aus dieser Datei wurden in dieser Arbeit nicht verwendet.

4.1 Probleme im Datensatz

Im Datensatz *Training* sind in der Kategorie 6 drei Bilder fälschlicherweise eingeordnet, die eigentlich der Klasse 5 angehören. Natürlich hat dies Auswirkungen auf das Training, da das Modell somit nicht mit korrekten Informationen trainiert werden kann. Daher werden die Bilder in die richtige Kategorie verschoben. In der oben gezeigten Grafik 4 ist dies nicht berücksichtigt.

Der Testdatensatz enthält nicht für jede Kategorie Bildbeispiele. Das selbst hat keine weiteren Auswirkungen, sollte aber in der Auswertung der Ergebnisse berücksichtigt werden.

5 Laden des Datensatzes

Bevor die Datensätze für Training und Validierung zur weiteren Verwendung eingelesen werden können, müssen diese erst auf lokal gespeichert werden. Da dies in Python sehr einfach erledigt werden kann, soll an dieser Stelle nicht weiter darauf eingegangen werden. Im Quellcode `Keras.py` sind die entsprechenden Funktionen zum Herunterladen und entpacken der Datensätze in den Zeilen 23 bis 53 definiert.

Ein Blick auf die Referenzimplementierung zu dieser Hausarbeit von Professor Volz zeigt den Aufwand, der in TensorFlow notwendig ist, um die Bilddaten zu laden und für das Verwendung in einem neuronalen Netz vorzubereiten. Keras kann diesen Prozess deutlich vereinfachen. Neben gewöhnlichem Laden der Daten bietet Keras mit der Klasse `ImageDataGenerator` und deren Methode `flow_from_directory` eine sehr einfache aber unheimlich mächtige Möglichkeit Bilddaten direkt aus einer Verzeichnisstruktur wie für den hier verwendeten Datensatz vorhanden auszulesen und für die weitere Verwendung vorzubereiten. Dies umfasst nicht nur gewöhnliches Preprocessing der Bilddaten, wie die Änderung und Vereinheitlichung der Bildgröße oder Ändern des Farbmodus, auch eine Echtzeit-Erweiterung des Datensatzes durch z.B. Verzerrung oder Rotation ist möglich. An dieser Stelle könnte auch eine Aufspaltung der Trainingsdaten in Trainings- und Testdatensätze erfolgen, aber aus den bereits im Abschnitt Der Datensatz genannten Gründen ist dies hier nicht sinnvoll.

Die im folgenden Listing 1 abgebildete Methode stammt aus dem Quellcode `Keras.py` und zeigt, wie Trainings- und Validierungsdatensatz jeweils für das Training der neuronalen Netze geladen werden. Ab Zeile 5 wird ein Objekt der Klasse `ImageDataGenerator` für die Generierung der Trainingsdaten angelegt. Bei diesen soll eine Erweiterung (Augmentation) der teils wenigen vorhandenen Bilder stattfinden, um eine breitere Datenbasis für das Training zu erreichen. Der erste Parameter, `rescale` gibt einen Skalierungsfaktor an, hat aber mit der Erweiterung der Datenbasis nichts zu tun. Diese wird über die folgenden Parameter `shear_range`, `zoom_range` und `horizontal_flip` gesteuert.

Für die Validierungsdaten wird auf eine Erweiterung der Daten verzichtet, um die Resultate der Modelle vergleichbar mit anderen Ansätzen zu halten (Zeile 13).

Das Laden der Bilddaten aus den Verzeichnissen erfolgt dann äquivalent für beide Datensätze über die Methode `flow_from_directory`, die auf dem jeweiligen `ImageDataGenerator` Objekt aufgerufen wird (ab Zeile 15 bzw. 22). Für beide wird zuerst der Pfad, von dem die Bilddateien geladen werden sollen angegeben, z.B. `/data/Training`. Mit dem nächsten Parameter, `target_size` wird die Zielgröße der Bilder für, in diesem Fall, die Eingabe in ein neuronales Netz definiert. In dem beschriebenen Projekt wird eine Zielgröße von 32x32 Pixeln verwendet. Die `batch_size` gibt an wie viele Bilddateien in einem Schritt verarbeitet werden sollen. Weiter könnte man über den Parameter `colormode` die Bilddaten zu Grauwertbildern konvertieren. Allerdings wurde die Entscheidung getroffen in diese Arbeit Farbbildern zu nutzen.

```
1         def setup_data():
2         #download_data()
3
4         train_datagen = ImageDataGenerator(
5         rescale=1. / 255,
6         shear_range=0.3,
7         zoom_range=0.3,
8         horizontal_flip=True)
9
10        valid_datagen = ImageDataGenerator(rescale=1. / 255)
11
12        train_generator = train_datagen.flow_from_directory(
13        train_data_dir,
14        target_size=(img_width, img_height),
15        batch_size=batch_size,
16        #colormode='grayscale',
17        class_mode='categorical')
18
19        validation_generator = valid_datagen.flow_from_directory(
20        validation_data_dir,
21        target_size=(img_width, img_height),
22        batch_size=batch_size,
23        #color_mode='grayscale',
24        class_mode='categorical')
25
26        return train_generator, validation_generator
```

Listing 1: Laden der Trainings- und Validierungsdatensätze

Der letzte Parameter ist besonders wichtig, mit `class_mode` wird angegeben, von welchem Typ das Label sein soll. Die korrekte Zuordnung dieses Modus zur Aufgabenstellung ist notwendig, um sinnvolle Ergebnisse aus der Arbeit mit einem Modell zu

erhalten. Da es sich bei dem belgischen Verkehrszeichen Datensatz um ein Klassifizierungsproblem mit mehr als 2 Klassen handelt, sollte *categorical* als Modus gewählt werden.

Bei den Rückgabewerten (Zeile 29) handelt es sich um `DirectoryIterator` Objekte, die Tupel aus jeweils einem Batch aus Bilddateien und den zugehörigen Labeln enthalten. Diese können ohne weitere Transformationen für Training und Validierung der Modelle genutzt werden.

Details zu den hier verwendeten Keras Methoden können in der API unter `https://keras.io/preprocessing/image/` und `https://keras.io/preprocessing/image/#imagedatagenerator-methods` nachgelesen werden.

6 Entwurf der Modellarchitekturen

Für diese Arbeit wurden drei Modellarchitekturen, die auf einander aufbauen, entworfen. Um eine Orientierung für die Leistungsfähigkeit solcher Netze zu erhalten und die eigenen Modelle dagegen zu vergleichen wurde zusätzlich eine öffentlich verfügbare Implementierung von den bekannten neuronalen Netz *LeNet* zu Keras 2 portiert und an die Eingabedaten angepasst.

Bei allen vier Modellarchitekturen handelt es sich um sogenannte *Convolutional Neural Networks* (CNN). Dieser Architekturtyp eignet sich besonders für die Klassifikation von Bildern, kann aber auch für andere Eingabedaten, wie z.B. Audio-Daten, verwendet werden. Eine Convolution ist zu deutsch eine Faltung, also eine mathematische Operation, die ebenso in der Bildverarbeitung eingesetzt wird um beispielsweise Kanten im Bild zu hervorzuheben und zu erkennen. In den CNNs wird diese Technik häufig zusammen mit *Pooling* genutzt, um so genannte *Features*, also interessante Kanten, Formen oder dominierende Farben in Ausschnitten des Eingangsbildes zu finden. Über *Fully Connected Layer*, also Schichten bei denen jeder Knoten einer Schicht mit jedem der folgenden Verknüpft ist, wird aus den Features, die durch Convolution und Pooling gewonnen wurden, entschieden wie das Eingabebild kategorisiert werden soll [8]. Eine ausführlichere Beschreibung zu CNNs aber auch vielen anderen Modellarchitekturen ist unter `http://www.asimovinstitute.org/neural-network-zoo/` zu finden. Dort ist ebenfalls das Paper [4] von Yann LeCun et al. verlinkt, auf dem die Modell-Implementierung *LeNet*, die zum Vergleich mit den eigenen Modellen verwendet werden soll, beruht.

6.1 Layer

Bevor die Implementierung der Netze im Detail betrachtet wird, sollen zuerst die einzelnen Layer, die in den hier erstellten Modellen Verwendung finden, erläutert werden. Die Layer sind hierbei etwa in der Reihenfolge, in der sie in den Modellen genutzt werden, aufgeführt.

- **Conv2D**
  Der *Conv2D* Layer realisiert die schon beschriebene Faltung im zweidimensionalen Raum.

- **MaxPooling2D**
  Mit einem *MaxPooling2D* Layer soll die Größe der zu verarbeitenden Bilddaten und damit des Rechenaufwands reduziert werden, ohne relevante Informationen (Features) zu verlieren. Ein Bild ist eine Matrix aus Farb- bzw. Helligkeitsinformationen, z.B. mit der Größe 4x4. Durch *MaxPooling* wird diese Matrix in z.B. vier Teilbereiche aufgeteilt und aus jedem nur der höchste Wert in eine 2x2 Matrix übernommen. Werden wie in dieser Arbeit RGB-Bilddaten verwendet, arbeitet das Netz für jeden Farbkanal auf einer eigenen Matrix.

- **Activation**
  Bei *Activation* handelt es sich nicht um eine Schicht in dem Sinne. Hiermit wird eine Aktivierungsfunktion zu dem Output des vorhergehenden Layers hinzugefügt. Dies kann zwar auch über den Parameter `activation` der anderen Layer selbst geschehen, für eine bessere Lesbarkeit wurde allerdings entschieden die Aktivierung für diese Arbeit über Activation-Layer zu realisieren. Verfügbare Aktivierungsfunktionen sind beispielsweise:

  - softmax - Softmax Aktivierungsfunktion
  - elu - Exponentiell lineare Einheit
  - softplus - Softplus Aktivierungsfunktion
  - softsign - Softsign Aktivierungsfunktion
  - relu - Rectified Linear Unit oder die
  - sigmoid - Sigmoid Aktivierungsfunktion

- **Dropout**
  Bei einem *Dropout* Layer handelt es sich um eine Methode, um die Gefahr von *Overfitting* (Überanpassung) eines Modells an den Trainingsdatensatz zu verhindern. Dropout inaktiviert dabei eine bestimmte Quote von Neuronen in einem Layer, sodass diese nicht für die Berechnung verwendet werden.

- **Flatten**
  Der *Flattening* Layer ist einer der wichtigsten Bestandteile eines CNNs. Im ersten Teil eines solchen werden in der Regel zweidimensionale Bilder verarbeitet. Zur Klassifikation ist es notwendig die aus Convolution und Pooling hervorgehenden Ergebnisse in einen eindimensionalen kontinuierlichen Vektor zu transferieren. Dies geschieht durch diesen Layer.

- **Dense**
  Bei *Dense* handelt es sich um einen *fully connected layer*, bei dem alle Knoten (Neuronen) des Layers mit denen des folgenden verbunden sind.

6.2 Aufbau eines Modells

Als Grundlage für jedes hier verwendete Modell dient die `Sequential` model API von Keras. Ein `Sequential` Modell kann durch `model = Sequential()` erzeugt werden. Die einzelnen Layer können durch die Methode `model.add(<Layer>)` hinzugefügt werden. Bei einem sequenziellen Modell werden die einzelnen Layer linear durchlaufen. Dabei haben *Convolutional Neural Networks*, wie sie hier zum Einsatz kommen sollen, in der Regel folgenden Grobaufbau:

1. Convolution Layer
2. Pooling Layer
3. Flattening Layer
4. Fully Connected Layer

Oft werden dabei Bündel aus Convolution und Pooling gebildet, die mehrfach wiederholt werden. Anschließend wird üblicherweise ein Flattening Layer verwendet, um einen eindimensionalen Ergebnisvektor zu erzeugen, bevor mehrere Fully Connected Layer folgen. Bei dem Ausgangslayer handelt es sich in der Regel um einen Fully Connected Layer, bei dem die Anzahl der Neuronen der Anzahl der Klassen entspricht.

Bevor ein Modell trainiert werden kann, ist es notwendig es zu *compilieren*. Dies geschieht durch die Methode `model.compile()`, die drei Argumente entgegen nimmt:

- den Optimierer (optimizer), Keras bietet verschiedene Optimierungsfunktionen zur Verwendung an. Es handelt sich dabei um mathematische Funktionen zur Fehlerminimierung.
- die Verlustfunktion/Zielfunktion (loss function), beschreibt das Ziel für welches ein neuronales Netz optimiert werden soll. Nicht jede Funktion eignet sich für jede Aufgabe. Für eine Klassifizierungsaufgabe mit mehr als zwei Klassen wird `categorical_crossentropy` sehr häufig eingesetzt.
- eine Liste von Metriken (metrics), die während Training und Testing ausgewertet werden sollen. Üblicherweise wird hier `metrics=['accuracy']` verwendet.

Weiterführende Informationen über die eingesetzten Methoden können der Keras API [2] entnommen werden.

6.3 Die Modelle

Für diese Arbeit wurden drei Modelle als *Convolutional Neural Network* entworfen. Modell zwei und drei wurden jeweils ausgehend von dem vorherigen weiter entwickelt. Bei dem ersten Modell handelt es sich um eine möglichst einfache Basisimplementierung.

Um einen Vergleich zu den selbst entwickelten Modellen zu erhalten, wurde zusätzlich eine Implementierung von *LeNet* [7] nach Keras 2 portiert und an die Aufgabenstellung angepasst. Da die einzelnen Layer schon beschrieben wurden, soll im Folgenden nur deren Zusammenstellung als Architektur für die einzelnen Modelle betrachtet werden.

6.3.1 CNN1

Bei *CNN1* handelt es sich um eine sehr einfache, naive Implementierung eines Convolutional Neural Networks. Mit nur einem Convolution Layer, einem Flattening Layer und zwei Fully Connected Layern, wovon einer die Ausgabeschicht darstellt, ist das Netz sogar noch einfacher als der in Abschnitt 6.2 vorgestellte Grobaufbau. Auf einen Pooling Layer wurde komplett verzichtet. Die Abbildung 5 stellt ebenfalls die Architektur des Modells dar. Unter *Output Shape* wird dabei das Format des Eingangsdatums nach der jeweiligen Schicht aufgeführt, während unter *Param* die Anzahl der Parameter der Schicht abgebildet sind, die während des Trainings optimiert werden.

```
1  def cnn_model_1():
2    model = Sequential()
3    model.add(Conv2D(2, kernel_size=(5, 5),
4              strides=(2, 2),
5              padding='same',
6              input_shape=input_shape))
7    model.add(Activation('relu'))
8
9    model.add(Flatten())
10   model.add(Dense(200))
11   model.add(Activation('relu'))
12   model.add(Dense(num_classes))
13   model.add(Activation('softmax'))
14
15   model.compile(loss='categorical_crossentropy',
16                 optimizer='adam',
17                 metrics=['accuracy'])
18
19 return model
```

Listing 2: Implementierung CNN1

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 16, 16, 2)         152
_________________________________________________________________
activation_1 (Activation)    (None, 16, 16, 2)         0
_________________________________________________________________
flatten_1 (Flatten)          (None, 512)               0
_________________________________________________________________
dense_1 (Dense)              (None, 200)               102600
_________________________________________________________________
activation_2 (Activation)    (None, 200)               0
_________________________________________________________________
dense_2 (Dense)              (None, 62)                12462
_________________________________________________________________
activation_3 (Activation)    (None, 62)                0
=================================================================
Total params: 115,214
Trainable params: 115,214
Non-trainable params: 0
```

Abbildung 5: Zusammenfassung der Architektur für das Modell CNN1

6.3.2 CNN2

Das zweite CNN ist schon deutlich komplexer, als der erste Versuch. Das Eingabebild durchläuft zweimal eine Sequenz aus zweimaliger Faltung, MaxPooling und einem relativ hohen Dropout, bei dem ein Viertel der Neuronen des Pooling Layers ausgeschaltet werden. Beide Sequenzen unterscheiden sich nur durch den Parameter `filters`, der einmal auf 32 und einmal auf 64 gesetzt wurde (Listing 3, Zeile 3/6 und 11/13). Als Aktivierungsfunktion wurde in beide Fällen `relu` verwendet (Zeile 5, 7, 12, 14). Die Architektur des Netzes wird wieder durch Flattening und zwei Fully Connected Layer abgeschlossen. Im Unterschied zum vorherigen Modell enthält der der erste Fully Connected Layer deutlich mehr Neuronen und es gibt zwischen diesem und der Ausgabeschicht einen Dropout von 50%. Wie auch zuvor wurde für den ersten Fully Connected Layer `relu` als Aktivierungsfunktion verwendet, während in der Ausgabeschicht `softmax` genutzt wird (Zeile 18 - 23). Wieder im Kontrast zur ersten Implementierung wird in dieser `rmsprop` als Optimizer genutzt (Zeile 34). Abbildung 6 stellt die Architektur des Netzes noch einmal dar.

```
def cnn_model_2():
  model = Sequential()
  model.add(Conv2D(32, kernel_size=(3, 3), padding='same',
            input_shape=input_shape))
  model.add(Activation('relu'))
  model.add(Conv2D(32, kernel_size=(3, 3)))
  model.add(Activation('relu'))
  model.add(MaxPooling2D(pool_size=(2, 2)))
  model.add(Dropout(0.25))

  model.add(Conv2D(64, kernel_size=(3, 3), padding='same'))
  model.add(Activation('relu'))
  model.add(Conv2D(64, kernel_size=(3, 3)))
  model.add(Activation('relu'))
  model.add(MaxPooling2D(pool_size=(2, 2)))
  model.add(Dropout(0.25))

  model.add(Flatten())
  model.add(Dense(512))
  model.add(Activation('relu'))
  model.add(Dropout(0.5))
  model.add(Dense(num_classes))
  model.add(Activation('softmax'))

  opt = keras.optimizers.rmsprop(lr=0.0001, decay=1e-6)
  model.compile(loss='categorical_crossentropy',
                optimizer=opt,
                metrics=['accuracy'])

  return model
```

Listing 3: Implementierung CNN2

Layer (type)	Output Shape	Param #
conv2d_2 (Conv2D)	(None, 32, 32, 32)	896
activation_4 (Activation)	(None, 32, 32, 32)	0
conv2d_3 (Conv2D)	(None, 30, 30, 32)	9248
activation_5 (Activation)	(None, 30, 30, 32)	0
max_pooling2d_1 (MaxPooling2	(None, 15, 15, 32)	0
dropout_1 (Dropout)	(None, 15, 15, 32)	0
conv2d_4 (Conv2D)	(None, 15, 15, 64)	18496
activation_6 (Activation)	(None, 15, 15, 64)	0
conv2d_5 (Conv2D)	(None, 13, 13, 64)	36928
activation_7 (Activation)	(None, 13, 13, 64)	0
max_pooling2d_2 (MaxPooling2	(None, 6, 6, 64)	0
dropout_2 (Dropout)	(None, 6, 6, 64)	0
flatten_2 (Flatten)	(None, 2304)	0
dense_3 (Dense)	(None, 512)	1180160
activation_8 (Activation)	(None, 512)	0
dropout_3 (Dropout)	(None, 512)	0
dense_4 (Dense)	(None, 62)	31806
activation_9 (Activation)	(None, 62)	0

Total params: 1,277,534
Trainable params: 1,277,534
Non-trainable params: 0

Abbildung 6: Zusammenfassung der Architektur für das Modell CNN2

6.3.3 CNN3

Die dritte Implementierung unterscheidet sich nur durch die Anzahl der Layer und Änderungen weniger Parameter von der vorherigen Implementierung. Eine weitere Sequenz von zwei Convolution, einem MaxPooling und einem Dropout Layer wurde neben einer weiteren Fully Connected Schicht eingefügt (siehe Listing 4 und Abbildung 7). Die Idee hinter diesem Modell war es vor allem zu untersuchen, wie sich die Qualität des Netzes durch weitere Layer und Änderungen der Parameter verhält. Leider ist es schwierig dies im Rahmen einer schriftlichen Ausarbeitung zu dokumentieren. Auch die Git-History enthält nicht alle Versuchsschritte, gibt aber einen kleinen Einblick in diese.

```
def cnn_model_3():
  model = Sequential()
  model.add(Conv2D(16, kernel_size=(3, 3), padding='same',
            input_shape=input_shape))
  model.add(Activation('relu'))
  model.add(Conv2D(16, kernel_size=(3, 3)))
  model.add(Activation('relu'))
  model.add(MaxPooling2D(pool_size=(2, 2)))
  model.add(Dropout(0.125))

  model.add(Conv2D(32, kernel_size=(3, 3), padding='same'))
  model.add(Activation('relu'))
  model.add(Conv2D(32, kernel_size=(3, 3)))
  model.add(Activation('relu'))
  model.add(MaxPooling2D(pool_size=(2, 2)))
  model.add(Dropout(0.125))

  model.add(Conv2D(64, kernel_size=(3, 3), padding='same'))
  model.add(Activation('relu'))
  model.add(Conv2D(64, kernel_size=(3, 3)))
  model.add(Activation('relu'))
  model.add(MaxPooling2D(pool_size=(2, 2)))
  model.add(Dropout(0.125))

  model.add(Flatten())
  model.add(Dense(1024))
  model.add(Activation('relu'))
  model.add(Dense(512))
  model.add(Activation('relu'))
  model.add(Dropout(0.4))
  model.add(Dense(num_classes))
  model.add(Activation('softmax'))

  opt = keras.optimizers.rmsprop(lr=0.0001, decay=1e-6)
  model.compile(loss='categorical_crossentropy',
                optimizer=opt,
                metrics=['accuracy'])

  return model
```

Listing 4: Implementierung CNN3

Layer (type)	Output Shape	Param #
conv2d_6 (Conv2D)	(None, 32, 32, 16)	448
activation_10 (Activation)	(None, 32, 32, 16)	0
conv2d_7 (Conv2D)	(None, 30, 30, 16)	2320
activation_11 (Activation)	(None, 30, 30, 16)	0
max_pooling2d_3 (MaxPooling2	(None, 15, 15, 16)	0
dropout_4 (Dropout)	(None, 15, 15, 16)	0
conv2d_8 (Conv2D)	(None, 15, 15, 32)	4640
activation_12 (Activation)	(None, 15, 15, 32)	0
conv2d_9 (Conv2D)	(None, 13, 13, 32)	9248
activation_13 (Activation)	(None, 13, 13, 32)	0
max_pooling2d_4 (MaxPooling2	(None, 6, 6, 32)	0
dropout_5 (Dropout)	(None, 6, 6, 32)	0
conv2d_10 (Conv2D)	(None, 6, 6, 64)	18496
activation_14 (Activation)	(None, 6, 6, 64)	0
conv2d_11 (Conv2D)	(None, 4, 4, 64)	36928
activation_15 (Activation)	(None, 4, 4, 64)	0
max_pooling2d_5 (MaxPooling2	(None, 2, 2, 64)	0
dropout_6 (Dropout)	(None, 2, 2, 64)	0
flatten_3 (Flatten)	(None, 256)	0
dense_5 (Dense)	(None, 1024)	263168
activation_16 (Activation)	(None, 1024)	0
dense_6 (Dense)	(None, 512)	524800
activation_17 (Activation)	(None, 512)	0
dropout_7 (Dropout)	(None, 512)	0
dense_7 (Dense)	(None, 62)	31806
activation_18 (Activation)	(None, 62)	0

Total params: 891,854
Trainable params: 891,854
Non-trainable params: 0

Abbildung 7: Zusammenfassung der Architektur für das Modell CNN3

6.3.4 LeNet

Um eine Referenzwert für die Güte der Modelle abseits der in TensorFlow vorgegebenen Basisimplementierung zu erhalten wurde als viertes Modell eine Implementierung von *LeNet*, genauer *LeNet-5* auf der Website `https://www.kaggle.com/tupini07/predicting-mnist-labels-with-a-cnn-in-keras/notebook` [7] gefunden. Diese Architektur wurde unter anderem ausgewählt, weil Yann LeCun als Begründer von Convolutional Neural Networks gilt, aber auch weil das Problem, für das sie in der Referenzimplementierung eingesetzt wurde, ähnlich zur Aufgabenstellung dieser Arbeit ist. Da das Modell dort allerdings für Keras 1 entwickelt wurde musste zusätzlich zu

der Anpassung an den hier verwendeten Datensatz eine Portierung zu Keras 2 stattfinden.

Neben der Anpassung an die Keras 2 API wurden im Vergleich zum Orginalcode aus [7] folgende Punkte angepasst:

- Der `input_shape` wurde von (28, 28, 1) (28 x 28 Pixel, 1 Farbkanal (Grayscale)) zu (32, 32, 3) (3 Farbkanäle (RGB)) geändert.
- Der letzte Fully Connected Layer, die Ausgangsschicht, wurde auf die Anzahl der unterschiedlichen Verkehrszeichen (Klassen) gesetzt.

Die letzte Anpassung ist zwingend notwendig, um das Netz für die Klassifikation der belgischen Verkehrszeichen zu adaptieren. Die Formatierung der Eingangsdaten wurde schon beim Laden der Daten anders festgelegt und sollte, um vergleichbare Resultate zwischen den einzelnen Implementierungen zu erreichen, konstant gehalten werden.

Auch *LeNet* ähnelt den vorhergegangenen Architekturen bis auf die Wahl der Parameter und der Anzahl der Layer. Details können Listing 5 und Abbildung 8 entnommen werden.

```
def cnn_model_lenet():
  model = Sequential()

  model.add(Conv2D(6, kernel_size=(5, 5), input_shape=input_shape,
            use_bias=True,padding="same"))
  model.add(Activation('relu'))
  model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2), padding="same"))
  model.add(Dropout(rate=0.12))

  model.add(Conv2D(16, kernel_size=(5, 5), use_bias=True, padding="same"))
  model.add(Activation('relu'))
  model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2), padding="same"))

  model.add(Conv2D(35, kernel_size=(5, 5), use_bias=True, padding="same"))
  model.add(Activation('relu'))
  model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2), padding="same"))

  model.add(Flatten())
  model.add(Dense(120, use_bias=True))
  model.add(Activation('relu'))
  model.add(Dropout(rate=0.5))
  model.add(Dense(84, use_bias=True))
  model.add(Activation('relu'))
  model.add(Dense(num_classes, use_bias=True))
  model.add(Activation('softmax'))

  model.compile(optimizer=Adam(lr=0.001),
                loss="categorical_crossentropy",
                metrics=['accuracy'])

  return model
```

Listing 5: Implementierung LeNet

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_12 (Conv2D)           (None, 32, 32, 6)         456
_________________________________________________________________
activation_19 (Activation)   (None, 32, 32, 6)         0
_________________________________________________________________
max_pooling2d_6 (MaxPooling2 (None, 16, 16, 6)         0
_________________________________________________________________
dropout_8 (Dropout)          (None, 16, 16, 6)         0
_________________________________________________________________
conv2d_13 (Conv2D)           (None, 16, 16, 16)        2416
_________________________________________________________________
activation_20 (Activation)   (None, 16, 16, 16)        0
_________________________________________________________________
max_pooling2d_7 (MaxPooling2 (None, 8, 8, 16)          0
_________________________________________________________________
conv2d_14 (Conv2D)           (None, 8, 8, 35)          14035
_________________________________________________________________
activation_21 (Activation)   (None, 8, 8, 35)          0
_________________________________________________________________
max_pooling2d_8 (MaxPooling2 (None, 4, 4, 35)          0
_________________________________________________________________
flatten_4 (Flatten)          (None, 560)               0
_________________________________________________________________
dense_8 (Dense)              (None, 120)               67320
_________________________________________________________________
activation_22 (Activation)   (None, 120)               0
_________________________________________________________________
dropout_9 (Dropout)          (None, 120)               0
_________________________________________________________________
dense_9 (Dense)              (None, 84)                10164
_________________________________________________________________
activation_23 (Activation)   (None, 84)                0
_________________________________________________________________
dense_10 (Dense)             (None, 62)                5270
_________________________________________________________________
activation_24 (Activation)   (None, 62)                0
=================================================================
Total params: 99,661
Trainable params: 99,661
Non-trainable params: 0
```

Abbildung 8: Zusammenfassung der Architektur für das Modell LeNet

7 Training der Modelle

Dass für das Einlesen der Bilddaten der `ImageDataGenerator` verwendet wurde, macht sich auch beim Training der Modelle bezahlt. Auf den Modellen kann die Methode `fit_generator` (Listing 6, Zeile 5) aufgerufen werden, die ein Batch-basiertes Training auf Basis der `DirectoryIterator` Objekte anstößt, die von der Methode `setup_data` aus Listing 1 zurückgegeben wurden. Der Generator selbst läuft parallel zum Training, um die Erweiterung des Datensatzes in Echtzeit zu realisieren. Der `fit_generator` nimmt dabei zum einen den `DirectoryIterator` der Trainingsbilder (`_train_generator`) als Argument, aber auch den der Validierierungsdaten (`_val_generator`). Dies bedeutet aber nicht, dass das Modell mit den Validierungsdaten trainiert wird, sondern, dass zum Abschluss jeder Epoche, also jedes Trainingsschritts, das bisherige Modell einmal an den Validierungsdaten getestet wird. Dabei handelt es sich um einen optionalen Schritt, der keine Auswirkung auf das Modell selbst hat. Bei den Argumenten `steps_per_epoch` und `validation_steps` wurde der Wert entsprechend der Empfehlung der Keras API auf die Anzahl der Bilddateien des jeweiligen Datensatzes geteilt

durch die Größe eines Batches gesetzt. Der Wert für die Anzahl der Epochen wird beim Aufruf des Trainings übergeben und hier willkürlich auf 50 gesetzt (siehe Listing 7).

```
1  def train_test_evaluate(_model, _train_generator, _val_generator,
2                                  _nb_train_samples, _nb_val_samples,
3                                  _batch_size, _epochs, _model_name):
4
5    _hist = _model.fit_generator(_train_generator,
6                                 steps_per_epoch=_nb_train_samples // _batch_size,
7                                 epochs=_epochs,
8                                 validation_data=_val_generator,
9                                 validation_steps=_nb_val_samples // _batch_size)
10
11   _score = _model.evaluate_generator(_val_generator, _nb_val_samples)
12   print("Score for model: Test loss: ", _score[0])
13   print("Score for model: Test accuracy: ", _score[1])
14
15   _model.summary()
16   plot_loss_accuracy(_hist, _epochs, _model_name)
17   _model.save("./models/" + _model_name + ".h5")
18
19   return _model, _hist, _score
```

Listing 6: Training und Auswertung der Modelle

Die Methode `fit_generator` liefert als Rückgabewert ein `History` Objekt, dass die Metriken des Trainings und der Validierung für jede Epoche enthält. Dies wird genutzt, um die Resultate für jedes Modell grafisch darzustellen 6, Zeile 16. Die Implementierung der Funktion kann im beiliegenden Quellcode nachvollzogen werden.

Analog zur Trainingsmethode `fit_generator` wird zur Validierung des Modells die Methode `evaluate_generator` verwendet. Diese evaluiert das fertig trainierte Modell wie es schon als Teil des Trainings für jede Epoche geschehen ist. Zurückgegeben wird ein Score bestehend aus *loss* und *accuracy* des Modells.

Abschließend werden weitere Informationen zum Modell geloggt. Mit der Methode `summary` aus Listing 6, Zeile 15 wird eine Überblick über die Architektur des Modells und die einzelnen Layer sowie deren Parameter ausgegeben. Wie schon erwähnt werden die Trainings- und Validierungsergebnisse grafisch aufbereitet und wie auch das Modell selbst gespeichert (Zeile 17). Ein trainiertes Modell kann einfach gespeichert werden, sodass es ohne erneutes Training z.B. in einer Anwendung eingesetzt werden kann.

```
if __name__ == '__main__':
  nb_train_samples = 4575
  nb_validation_samples = 2520
  epochs = 50
  batch_size = 16

  train_gen, val_gen = setup_data()

  model_cnn_1 = cnn_model_1()
  m1_model, m1_hist, m1_score = train_test_evaluate(model_cnn_1, train_gen,
                                                    val_gen, nb_train_samples,
                                                    nb_validation_samples,
                                                    batch_size, epochs, "CNN1")
```

Listing 7: Starten des Trainings exemplarisch für das erste Modell

Das Listing 7 zeigt einen Ausschnitt aus der Main-Funktion des Programms, mit dem die Daten vorbereitet und das Training, exemplarisch für das erste Modell, gestartet wird. Die genutzten Parameter `epochs` und `batch_size` sind dabei willkürlich, aber in sinnvollen Grenzen, gewählt.

8 Auswertung der Trainingsergebnisse

Ein Blick auf die Ergebnisse aus Training und Validierung in Tabelle 1 zeigt eine Schwierigkeit im maschinellen Lernen: Die Ergebnisse liegen teils sehr dicht beisammen. In den ausgeführten Trainingsläufen konnte auch kein klar favorisiertes Modell ausgemacht werden. Meist hatte zwar das Modell *CNN2* um wenige Prozent das beste Ergebnis, aber auch *CNN3* führte in wenigen Tests mit ein bis zwei Prozent Vorsprung. *LeNet* war in fast allen Durchläufen schlechter als *CNN2* und *CNN3*, im Testing jedoch besser als *CNN1*, während dieses im Training klar besser war.

Netz	Orginal-Datensatz				Korrigierter Datensatz			
	Training		Testing		Training		Testing	
	Accuracy	Loss	Accuracy	Loss	Accuracy	Loss	Accuracy	Loss
CNN1	0.966	0.111	0.913	0.421	0.968	0.105	0.903	0.456
CNN2	0.935	0.230	0.965	0.137	0.927	0.244	0.957	0.167
CNN3	0.918	0.283	0.948	0.214	0.925	0.244	0.930	0.251
LeNet	0.899	0.300	0.943	0.224	0.903	0.293	0.927	0.252

Tabelle 1: Übersicht der Trainings- und Testergebnisse

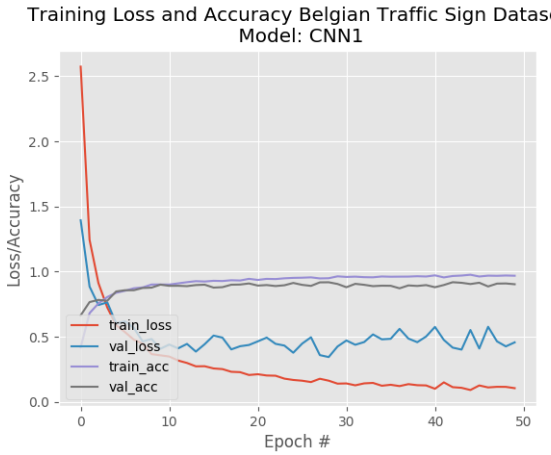
Bevor die Ergebnisse weiter diskutiert werden, soll noch einmal kurz darauf eingegangen werden, welche Werte in der Tabelle 1 und 9 abgebildet werden. Während für das

Testing durch die Methode `evaluate_generator` ein klarer Wert für Accuracy und Loss ausgegeben wird, gibt es im Training für jedes Sample einen neuen Wert, der durch ein History-Objekt ausgelesen werden kann (vgl. Listing 6, Zeile 11 und 5 aus Abschnitt 7). Damit gibt es allerdings auch keinen eindeutigen Wert für die Qualität des Modells im Training. Zwar könnte z.B. die Methode des Testings nochmals mit den Trainingsdaten angewandt werden, aber da das Modell diese Daten bereits kennt entspricht das Ergebnis nicht notwendigerweise der Qualität des Netzes. Daher ist in der Tabelle 1 für Trainings-Accuracy und -Loss jeweils der Wert aus der letzten Iteration der letzten Epoche des Trainings aufgeführt. Diese Werte sollen nur dem Vergleich mit denen aus der Validierung dienen. Alle Werte können in den beiliegenden Trainingslogs nachvollzogen werden. Auch die grafische Darstellung in Abbildung 9 soll dabei helfen die Qualität der Modelle zu beurteilen.

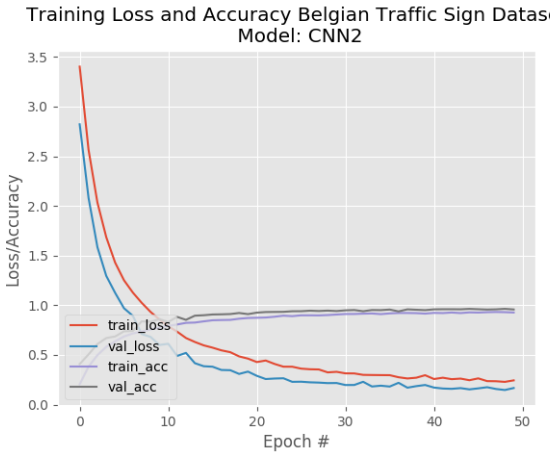
Wie schon der erste Blick auf die Resultate des Trainings gezeigt hat, sind die Ergebnisse schwierig zu interpretieren. Alle Netze liefern zufriedenstellende Resultate, die dennoch deutlich anders ausfallen, als erwartet wurde. Eine der Annahmen war, dass die Accuracy im Training besser als im Testing sein würde. Dies ist nur einmal mit *CNN1* und dem originalen, nicht korrigierten Datensatz der Fall. Eine mögliche Erklärung könnte darauf basieren, dass im Testing-Datensatz nicht für alle Klassen Bilder vorhanden sind. Werden Bildklassen, die für das Netz im Training schwierig einzuordnen waren, in der Validierung des Netzes nicht abgefragt, erscheint es valide, dass die Accuracy des Modells steigt. Dies führt direkt zur nächsten Annahme. Es wurde angenommen, dass durch die Korrektur der falsch eingeordneten Trainingsbilder die Trainings- und Testergebnisse verbessert würden. Die Veränderung in den Trainingsergebnissen ist allerdings minimal und im Rahmen der Schwankungen, die auch zwischen mehreren Trainingsläufen auf einem gleichbleibenden Datensatz beobachtet wurden. Dagegen haben sich die Ergebnisse für das *Testing* mit dem korrigierten Datensatz minimal, aber über den üblichen Schwankungen, verschlechtert. Als ein weiteres Resultat aus den Ergebnissen lässt sich anführen, dass ein komplexeres Modell mit mehr Schichten nicht zwingend eine bessere Leistung erbringt, als ein einfaches. Auch ein bekanntes Netz muss nicht zwingend die besten Ergebnisse liefern.

In einem Blick auf die Grafiken aus Abbildung 9 zeigen deutlichere Unterschiede zwischen den Netzen als die bloßen Zahlen. Gerade ein Blick auf den Verlauf des Trainings- und Validierungs-Losses ist interessant und korreliert mit der Güte der Netze. Die Kurve von *CNN1* weist die wenigsten Schwankungen auf, auch die Accuracy ist am Höchsten.

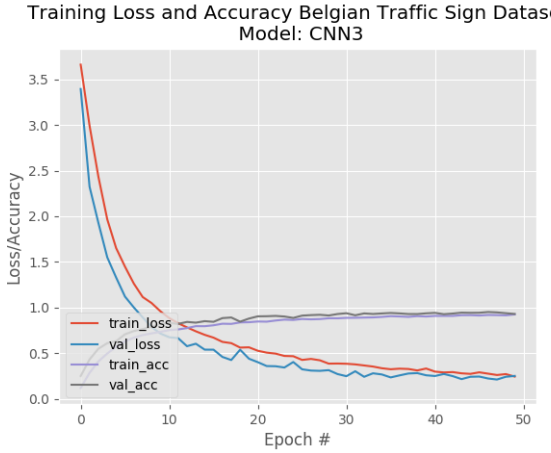
Wie sich die Netze für vollständig unbekannte Bilder unterscheiden, soll im nächsten Abschnitt untersucht werden.



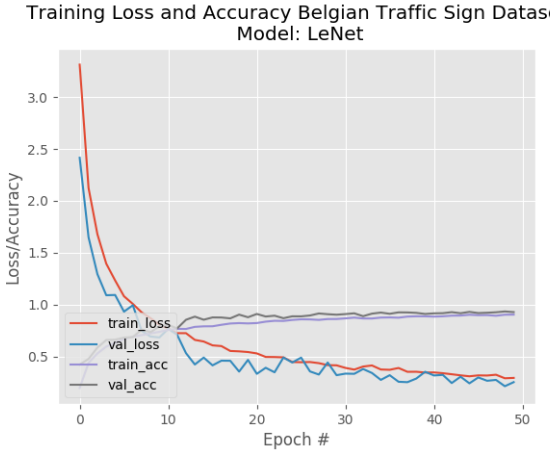
(a) CNN1



(b) CNN2



(c) CNN3



(d) LeNet

Abbildung 9: Grafische Auswertung der Ergebnisse aus dem Training

9 Vorhersagen für unbekannte Bilder

Um die Modelle für unbekannte Bilder nach dem Training zu nutzen, wurde eine eigenständige Anwendung in der Datei `ClassifyImages.py` erstellt. Ein Ausschnitt ist in Listing 8 dargestellt. Über die Kommandozeilen-Argumente `--model` und `--image` kann jeweils der Pfad zu einem bereits trainierten Modell im Format `.h5` und der zu dem zu klassifizierenden Bild angegeben werden. Bild und Modell werden geladen (8, Zeile 3 und 4) und eine Vorhersage für diese erstellt (Zeile 6). Neben der Ausgabe auf der Konsole wird das Eingangsbild, mit den Ergebnissen der Vorhersage angereichert, in eine Datei gespeichert.

Für den erneuten Test sollen alle vier Modelle im Bezug auf sechs Bilder von Verkehrszeichen, die nicht dem Trainings- oder Testdatensatz angehören getestet werden. Abbildung 10 zeigt diese inklusive derer Einordnung in die Klassen des Datensatzes.

```
if __name__ == '__main__':
  args = parse_args()
  image, org = load_image(args)
  model = load_model(args["model"])

  pred = model.predict_classes(image)

  print("Class: " + str(pred[0]))
  print("Class label: " + str(labels[pred[0]]))
```

Listing 8: Nutzung der erstellten Modelle für unbekannte Bilder

(a) Klasse 2 (b) Klasse 27 (c) Klasse 28

(d) Klasse 30 (e) Klasse 41 (f) Klasse 56

Abbildung 10: Zusätzliche Bilder

Die Bilder liegen in verschiedenen Größen und Auflösungen vor. Für den Test mussten die Originalbilder zugeschnitten werden. Da die Trainingsbilder fast keine Umgebung zu den Verkehrszeichen zeigen, sind die Ergebnisse alle Modelle nur für ähnliche Ausschnitte zuverlässig. Die in Abbildung 10 zeigt die Bilder schon entsprechend präpariert. In der folgenden Tabelle 2 sind die Vorhersagen der Modelle für diese Bilder abgedruckt. Nur bei Modell *CNN2* und *CNN3* kam es jeweils zu einem Fehler. Modell *CNN3* hat das Bild 10b der Klasse 27 fälschlicherweise der Klasse 24 zugeordnet. Beide Klassen sind allerdings auch sehr ähnlich, wie Abbildung 11 zeigt. Modell *CNN2* hat dagegen das Bild 10c der Klasse 28 der Klasse 32 zugeordnet. Dies kann durch die Verschmutzung im weisen Bereich des Testbildes erklärt wieder. Ein Vergleich zwischen den Klassen zeigt Abbildung 12.

Bild	CNN1	CNN2	CNN3	LeNet
Klasse 2 (slippery road)	2	2	2	2
Klasse 27 (maximum height allowed)	27	27	24	27
Klasse 28 (no traffic allowed in both directions)	28	32	28	28
Klasse 30 (no right turn)	30	30	30	30
Klasse 41 (no parking)	41	41	41	41
Klasse 56 (pedestrian crosswalk)	56	56	56	56

Tabelle 2: Ergebnisse für zusätzliche Bilder

(a) Klasse 27 (b) Klasse 24

Abbildung 11: Vergleich der Klassen 27 und 24

(a) Klasse 28 (b) Klasse 32

Abbildung 12: Vergleich der Klassen 28 und 32

Für diesen Test soll die Accuracy für die einzelnen Modelle nicht berechnet werden, da der Stichprobenumfang viel zur gering ist, um eine sinnvolle Aussage zu treffen. Dennoch zeigt dieser wie die trainierten Netze in einer Anwendung eingesetzt werden können. Interessant ist besonders, dass die Fehler der Modelle durchaus nachvollziehbar für einen menschlichen Betrachter sind.

10 Resümee

Durch die praktische Arbeit mit neuronalen Netzen wurden im Rahmen dieser Arbeit einige neue, unerwartete Erkenntnisse gewonnen, die durch eine rein theoretische Ausarbeitung oder eine Klausur schwer vermittelbar gewesen wären. Dabei sind vor allem folgende Punkte aufgefallen:

- Auch die Label eines bekannten, oft verwendeten Datensatzes sind nicht immer korrekt.
- Die Anzahl der Layer korreliert nicht mit der Güte eines neuronalen Netzes.
- Die Güte eines Netzes kann in verschiedenen Trainingsläufen merklich schwanken.
- Die Trainings-Accuracy kann niedriger als die Test-/Validierungs-Accuracy sein.
- Die Bildausschnitte, mit denen trainiert wird, bestimmen die Ausschnitte auf denen ein Modell sinnvoll angewandt werden kann.

Wie auch aus der Arbeit herauszulesen ist, fiel auch die Bewertung der Modelle schwer, da sie in den Werten sehr nah aneinander liegen. Dass ausgerechnet die beiden Modelle mit der höchsten Accuracy in der Validierung im Test mit externen Bildausschnitten einen Fehler machten, erschwert dies zusätzlich. Es bleibt die Frage, ob die Ergebnisse durch einen größeren Datensatz für Training, Testing und Validierung klarer ausgefallen wären.

A Anhang

Literatur

[1] *Belgian Road Signs Definitions*. URL: https://www.shape2day.com/arriving--leaving-shape/vehicles/driving-authorization.

[2] *Keras.io*. URL: https://keras.io.

[3] David Kriesel. *Ein kleiner Überblick über Neuronale Netze*. 2007. URL: http://www.dkriesel.com.

[4] Y. LeCun u. a. "Gradient-Based Learning Applied to Document Recognition". In: *Proceedings of the IEEE* 86.11 (Nov. 1998), S. 2278–2324. URL: http://yann.lecun.com/exdb/publis/pdf/lecun-98.pdf.

[5] Christiane Pütter. *Gartner nennt 3 Megatrends der Zukunft*. CIO, März 2018. URL: https://www.cio.de/a/gartner-nennt-3-megatrends-der-zukunft,3561336.

[6] Radu Timofte, Karel Zimmermann und Luc Van Gool. "Multi-view Traffic Sign Detection, Recognition, and 3D Localisation". In: *Machine Vision and Applications* (2011). DOI: 10.1007/s00138-011-0391-3.

[7] Andrea Tupini. *Predicting MNIST labels with a CNN in Keras*. URL: https://www.kaggle.com/tupini07/predicting-mnist-labels-with-a-cnn-in-keras/notebook.

[8] Fjodor van Veen. *The Neural Network Zoo*. The Asimov Institute, Sep. 2016. URL: http://www.asimovinstitute.org/neural-network-zoo/.

List of Listings

Abbildungsverzeichnis

Tabellenverzeichnis