

# Verkehrsschilder erkennen

Road Sign Observation (RoadSignObs)

Jonas Heinke

01/23/19

## Überblick

Ziel dieses Programmsystems ist das Erkennen und das Verarbeiten von relevanten Verkehrsschildern im Straßenverkehr. Mehrere Programmmodule werden dazu bereitgestellt. Das wichtigste Modul ist das Vorhersagemodul Prediction.py gemeinsam mit dem Klassenmodul Cnn.py. Deep Learning Modelle mit TensorFlow und Keras werden in diesem Zusammenhang verwendet. Nach dem Training können einzelne Verkehrsschilder, die völlig unabhängig von den Trainingsdaten sind, mit einer relativ hohen Wahrscheinlichkeit erkannt werden. Eine Erweiterung ist die Verarbeitung von Straßenszenarien, um Verkehrsschilder zu detektieren. Die Szenarien können über Bildmaterial oder direkt von einer Kamera (WebCam) bereitgestellt werden. Diese können ebenfalls dem Vorhersagemodell übergeben werden. Die Kommunikation zwischen den Programmmodulen erfolgt per ROS (Robot Operating System). Es kann auch computerübergreifend erfolgen. Tests dazu verliefen erfolgreich.

Das Projekt ist als Testumgebung zu betrachten, um zum Beispiel verschiedene Vorhersagemodelle oder Umgebungsszenarien zu testen. Die Trainingsdaten und Testdaten entstammen der Internetseite <http://benchmark.ini.rub.de/?section=gtsdb&subsection=dataset> mit dem Titel "The German Traffic Sign Detection Benchmark".



## Inhalt

<b>Überblick</b>	<b>1</b>
<b>1. Projektstruktur und Programmmodule</b>	<b>2</b>
<b>2. Modul “Prediction.py”</b>	<b>3</b>
<b>3. Modul “Cnn.py”</b>	<b>3</b>
<b>4. Modelldefinition im Modul “Cnn.py”</b>	<b>5</b>
<b>5. Modul “SubscribCarCrt.py”</b>	<b>7</b>
<b>6. Modul “PublishPic.py”</b>	<b>7</b>
<b>7. Modul “PublishCam.py” und Klassenmodul “Detector.py”</b>	<b>8</b>
<b>8. Test und Testergebnisse</b>	<b>9</b>
8.1 Training	9
8.2 Einzelne Verkehrszeichen erkennen	9
8.3 Verkehrszeichen in einer Straßenszene detektieren und erkennen	13
<b>9. Zusammenfassung, Schlussfolgerungen</b>	<b>15</b>
<b>Quellen</b>	<b>16</b>

## 1. Projektstruktur und Programmmodule

Die folgende Abbildung 1 zeigt die komplette Projektstruktur. Die Darstellung wurde aus der mit „roslaunch rqt\_graph rqt\_graph“ generierten Graphik abgeleitet und ergänzt. Die Programmmodule bzw. Nodes und Topics sind wie aus ROS bekannt dargestellt. Wichtige Basismethoden wurden in separaten Modulen, in Klassen ausgelagert. Diese Module sind blau umrandet. Die Bilddaten befinden sich in der Regel in separaten Unterverzeichnissen. Diese sind blau gestrichelt umrandet. Das gleiche gilt auch für CSV-Dateien. Die CSV-Dateien enthalten Zusatzinformationen über die Bilddateien und dienen zur Steuerung des Ladens der Bilddateien.

Die Programmmodule wurden in zwei unterschiedlichen Packages untergebracht. Auch das ist aus der Abbildung ersichtlich. Dieses Konzept erlaubt es, relativ einfach die Packages auf zwei unterschiedlichen Computern unterzubringen, die miteinander kommunizieren.

Der Schwerpunkt dieser Entwicklung liegt im Modul Prediction mit dem zugehörigen Modul „Cnn.py“ und deren Klasse „Gtsrb“. Das dort verwendete Verfahren ist auch unter Deep Learning bekannt. Dieser Teil soll im folgenden Abschnitt näher beschrieben werden.

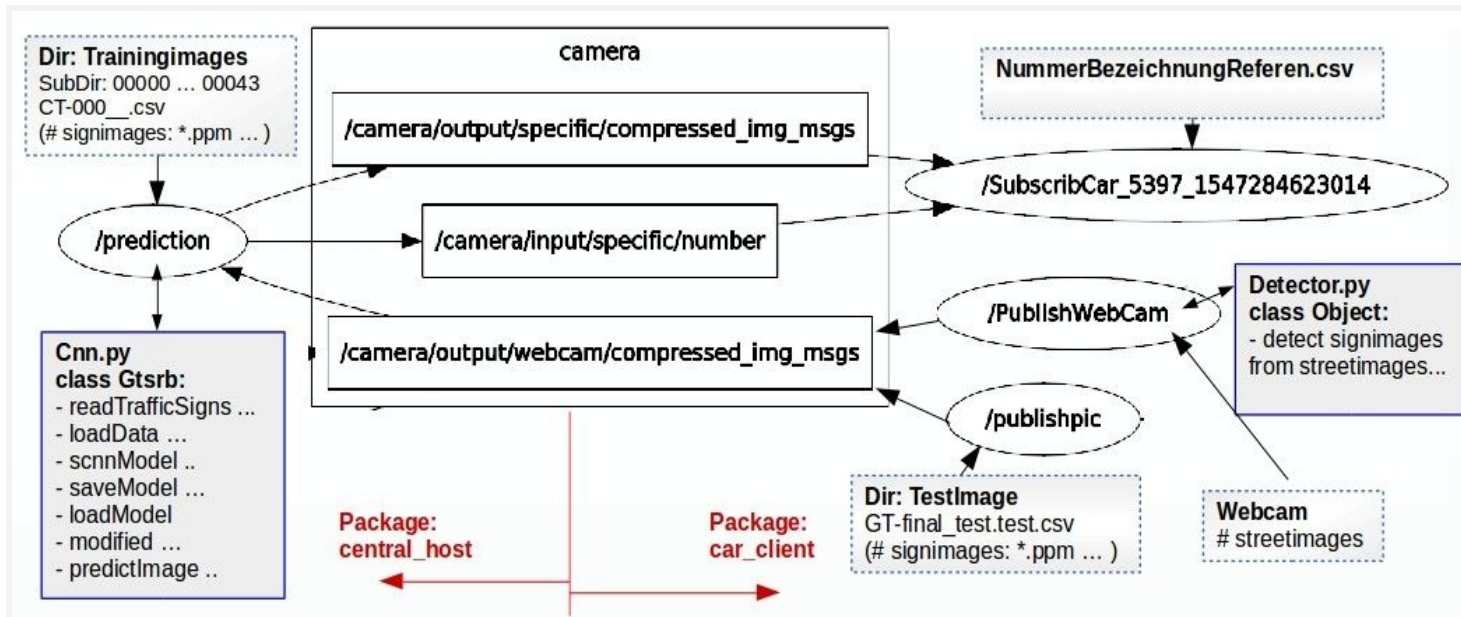


Abbildung 1: Projektstruktur

## 2. Modul “Prediction.py”

Das Modul Prediction und die gleichnamige, interne Klasse übernehmen im Wesentlichen die Kommunikation zu den anderen Modulen. Es besitzt einen Subscriber zum Empfang der Bilddaten und einen Publisher zum Senden der vorhergesagten Klassennummer einschließlich einer Aussage zur Erkennungsqualität. Zusätzlich wird das empfangene Bild mit Hilfe eines zweiten Publishers an einen Topic weitergereicht. Eine visuelle Beurteilung der tatsächlichen Erkennungsrate wird dadurch möglich. Der Subscriber korrespondiert mit der callbackCam-Methode. Von dort aus wird auch das Modul „Cnn“ mit der Erkennungsmethode „cnn.prediction(nplImage)“ gerufen. Rückgabewert ist dann das zugehörige und hoffentlich richtig erkannte Label bzw. die Klassennummer. Instanziiert wird die Klasse „cnn=Cnn.Gtsrb()“ einmalig global.

Eine zusätzliche Methode „saveBild“ ist für die Funktionsweise bedeutungslos. Sie diente lediglich zum Speichern eines Kontrollbildes zu Testzwecken.

## 3. Modul “Cnn.py”

Das Modul Cnn.py ist das wichtigste Modul in diesem Projekt. Hier wurden die Deep Learning Strategien implementiert. Es ist erweiterungsfähig und modifizierbar, derart dass unterschiedliche Modelle implementiert und getestet werden können.

Das Modul „Cnn.py“ besitzt eine Klasse „Gtsrb“ mit mehreren Methoden und einem Konstruktor. Der Konstruktor ist wesentlich. In ihm wird überprüft, ob ein Modelltraining bereits stattgefunden hat. Mit „tray“ wird versucht ein trainiertes Modell zu laden

„self.loadModel()“. Ist keines vorhanden, so wird ein Trainingslauf gestartet. Je nach Datenmenge und je nach eingestellter Durchlaufanzahl „epochs“ kann das mehr als eine Stunde in Anspruch nehmen. Die Methode „self.modified()“ übernimmt diese Aufgabe. Die Methode „modified()“ lädt die Trainings- und Testdaten per Methode „self.loadData()“. Die Methode „self.loadData()“ kennt den Speicherpfad der zur Verfügung stehenden Datenmenge und die Anzahl der Klassen. In dieser Methode wird auch eine Vorverarbeitung der Daten vorgenommen. Da das Datenmaterial sehr unterschiedlich bereitgestellt werden kann, ist diese Vorverarbeitung erforderlich und jeweils anzupassen. Der eigentliche Ladevorgang der Bilddaten erfolgt allerdings in der Methode „self.readTrafficSigns(rootpath=\"./TrainingImages\", subDirNo=num\_classes)“. Die Bilder der einzelnen Klassen sind in Unterverzeichnissen gespeichert. In jedem Unterverzeichnis befindet sich eine csv-Datei, die über Namen, Größe und Anzahl der Bilddaten der jeweiligen Klasse Auskunft gibt. Diese muss zuerst gelesen werden, um im Anschluss die Bilder laden zu können. Da die Daten als Trainings- und als Testdaten zur Verfügung stehen müssen, sind diese für die Weiterverarbeitung mit der Anweisung „X\_train, X\_test, y\_train, y\_test= train\_test\_split (nplImages, labels, test\_size=0.1, random\_state=33)“ zu splitten (aufzuteilen). Der gesamte Ladevorgang gestaltete sich dadurch recht aufwendig. Zurück zur Methode „modify()“. Der gesamte Trainingsvorgang lässt sich in Phasen einteilen. Diese Phasen sind im Quelltext auch kommentiert.

### **Phase 1:** Laden der Trainings- und der Testdaten

Diese Phase wurde bereits erläutert

### **Phase 2:** Definition des Models

Die Modelldefinition beinhaltet den kreativen Teil des Trainingsprozesses. Aus diesem Grunde wird im folgenden Abschnitt näheres dazu an einem Beispiel „scnnModel(num\_classes)“ erläutert. Grundsätzlich können unterschiedlich Modelle in dieser Klasse implementiert und deren Effizienz bewertet werden.

### **Phase 3:** Kompilieren

Das Kompilieren ist ein Übersetzungslauf.

### **Phase 4:** Fit Model

Abarbeitung, Ausführen des Trainings.

### **Phase 5:** Evaluate

Diese Phase liefert Ergebnisse zur Erkennungs- und Fehlerrate.

### **Phase 6:** Speichern des Modells

Das Speichern des Modells erfolgt mit der Klassenmethode „self.saveModel(fileName=\"cnnGtsrbModel\"). Das erlernte Wissen über die Bilder und deren Klassifizierung wird in zwei Dateien „\*.json“ und „\*.h5“ gesichert.

In Folge werden bei der Instanzierung der Klasse diese Trainings-Modelldateien geladen, um eine Vorhersage bezüglich der Verkehrsschilder zu ermöglichen. Das Laden geschieht mit der Klassenmethode „self.loadModel(self, fileName="cnnGtsrbModel")“:

Zur Vorhersage beinhaltet das Modul Cnn die Methode „predictImage(self, input\_data)“. Dort wird das Vorhersagemodell gerufen „prediction = self.model.predict(input\_data)“ und in Folge wird das Label herangezogen mit der höchsten Trefferwahrscheinlichkeit. In Ergänzung werden die zugehörigen Wahrscheinlichkeitswerte auch ausgewertet.

## 4. Modelldefinition im Modul “Cnn.py”

Die Modelldefinition hat beispielsweise folgendes Aussehen:

```
def scnnModel(self, num_classes):
    self.model.add(Conv2D(32, (3, 3), input_shape=(3, img_rows, img_cols), activation='relu',
                                                                    kernel_constraint=maxnorm(3)))

    self.model.add(Dropout(0.2))
    self.model.add(Conv2D(32, (3, 3), activation='relu', padding='same', kernel_constraint=maxnorm(3)))
    self.model.add(MaxPooling2D(pool_size=(2, 2)))
    self.model.add(Flatten())
    self.model.add(Dense(batch_size, activation='relu', kernel_constraint=maxnorm(3)))
    self.model.add(Dropout(0.5))
    self.model.add(Dense(num_classes, activation='softmax'))
    return self.model
```

Diese Lösung aus (2) von Jeson Brownlee wurde zu einer Klassenmethode umgeschrieben und einige Parameter wurden modifiziert. Ebenenweise werden die Bilddaten verarbeitet. Einige Erklärungen folgen:

**Ebene: Conv2D(filters(32), kernel\_size(3, 3), input\_shape=(3, img\_rows, img\_cols), activation='relu', kernel\_constraint=maxnorm(max\_value=3))**

- 2D-Faltungsschicht (<https://keras.io/layers/convolutional/>)

Nicht alle möglichen Parameter wurden gesetzt. In diesem Fall gilt:

filters(32) - Anzahl der Ausgabefilter für die Faltung

kernel\_size(3,3): Größe des Faltungskerns

input\_shape=(img\_rows, img\_cols, 3) – Eingangsbildgröße: Höhe, Breite und 3 Farben

activation='relu': gleichgerichtet linear (siehe <https://keras.io/activations/>)

kernel\_constraint=maxnorm(3): Faltungsfunktion (siehe <https://keras.io/constraints/>)

**Ebene: Dropout(0.2)**

- 20% der Eingangsdaten werden ignoriert (jedes 5 Bild)

**Eben: Convolution2D(32, (3, 3), activation='relu', padding='same', kernel\_constraint=maxnorm(3))**

- 2D-Faltung erneut angewendet

padding='same': gleiche Länge für Ein- und Ausgabe

**Ebene: MaxPooling2D(pool\_size=(2, 2))**

- Fenster von jeweils 2x2 Pixel werden zu einem Pixel zusammengefasst. Der größte Wert des Fensters wird übernommen. Dient der Datenreduktion.

**Ebene: Flatten()**

- Flacht die Eingabe ab, Dimensionsreduktion (<https://keras.io/layers/core/#dense>)

**Ebenen: Dense(units=batch\_size, activation='relu', kernel\_constraint=maxnorm(3))**

- Implementiert eine Aktivierungsfunktion:

„output = activation(dot(input, kernel) + bias)“ (<https://keras.io/layers/core/>),  
bias etwa 0.1 zur Vermeidung toter Neuronen

units=batch\_size (hier units=256): Ausgabekanäle

activation='relu': Aktivierungsfunktion

kernel\_constraint=maxnorm(3): Funktion zur Beschränkung der Gewichtung

**Ebene: Dropout(0.5)**

- nur jeder zweite Wert wird übernommen

**Ebene: Dense(units=num\_classes, activation='softmax')**

- Aktivierungsfunktion

units= classes: entspricht der Anzahl der Klassen, hier 44

activation='softmax': Klassennummern mit maximalen Wahrscheinlichkeiten werden übernommen

In der Klasse „Gtsrb“ des Cnn-Moduls befindet sich ein zweites, etwas umfangreicheres Modell „lcnModel“. Grundsätzlich kommen keine anderen Anweisungen, wie bereits besprochen, zur Anwendung. Allerdings wurden wesentlich mehr Ebenen implementiert. Diese Klassenmethode folgt:

```
def lcnModel(self, num_classes):
    self.model.add(Conv2D(32, (3, 3), input_shape=(img_rows, img_cols, 3),
                                                                activation= 'relu' , padding= 'same' ))
    self.model.add(Dropout(0.2))
    self.model.add(Conv2D(32, (3, 3), activation= 'relu' , padding= 'same' ))
    self.model.add(MaxPooling2D(pool_size=(2, 2), data_format='channels_last'))
    self.model.add(Conv2D(64, (3, 3), activation= 'relu' , padding= 'same' ))
    self.model.add(Dropout(0.2))
    self.model.add(Conv2D(64, (3, 3), activation= 'relu' , padding= 'same' ))
    self.model.add(MaxPooling2D(pool_size=(2, 2)))
    self.model.add(Conv2D(128, (3, 3), activation= 'relu' , padding= 'same' ))
    self.model.add(Dropout(0.2))
    self.model.add(Conv2D(128, (3, 3), activation= 'relu' , padding= 'same' ))
```

```

self.model.add(MaxPooling2D(pool_size=(2, 2), padding='same'))
self.model.add(Flatten())
self.model.add(Dropout(0.2))
self.model.add(Dense(1024, activation= 'relu' , kernel_constraint=maxnorm(3)))
self.model.add(Dropout(0.2))
self.model.add(Dense(512, activation= 'relu' , kernel_constraint=maxnorm(3)))
self.model.add(Dropout(0.2))
self.model.add(Dense(num_classes, activation= 'softmax' ))
return self.model

```

## 5. Modul “SubscribCarCrt.py”

Dieses Modul empfängt die gesichteten Verkehrszeichen und deren Bedeutung in Form einer Label-Nummer (Klassennummer). Da die Label-Nummer nicht sehr anschaulich ist, wird in einer Datei “nummerBezeichnungReferenz.csv” nach der Bedeutung gesucht und auf dem Display mit ausgegeben.

Grundsätzlich können diese Informationen zum teilweise autonomen Betrieb eines Fahrzeuges genutzt werden. Wird ein Stop-Schild erkannt, so würde das Fahrzeug in jedem Fall anhalten. Geschwindigkeitsüberschreitungen, auf Grund eines erkannten Verkehrsschildes, könnten eine akustische Warnung aktivieren.

Neben dem Label bzw. der Labelnummer enthält der Empfangsstring auch eine Information zur Zuverlässigkeit der Vorhersage. Diese stammt aus dem Modul “Prediction”.

Der Aufbau des Programms ist einfach gehalten. Es besitzt zwei Subscriber zum Empfang der Label-Nummer und der Verkehrszeichenabbildung. Die Methode “readReferenz(self, roadSignNumber)” sucht in der cnf-Datei nach einem zugehörigen Text. Neben der textuellen Ausgabe ist in Erweiterung eine akustische Ausgabe denkbar.

## 6. Modul “PublishPic.py”

Das Modul “PublishPic.py” wählt zufällig Verkehrszeichenbilder aus dem Verzeichnis “TestImages” aus und sendet diese an das Modul Prediction.py, natürlich über einen Topic. Die Bilddateien dieses Verzeichnisses “TestImages” sind völlig unabhängig von den Trainingsdaten.

Wichtigste Methoden dieses Moduls sind “readSigns”, “getRandomImages” und “publishPicture”.

## 7. Modul “PublishCam.py” und Klassenmodul “Detector.py”

Ziel dieser beiden Module ist es komplette Straßenszenen zu analysieren. Das heißt, das in einem Gesamtbild einzelne geometrische Objekte (Schilder) erkannt, isoliert und an das Modul Prediction geschickt werden sollen. Das Modul “PublishCam.py” übernimmt die Kommunikation und das Klassenmodul “Detector.py” detektiert einzelne Verkehrsschilder innerhalb einer Straßenszene. Die eigentliche Bewertung des erkannten Objektes übernimmt dann das Modul “Prediction.py”.

Mehrere Lösungsansätze wurden in diesem Zusammenhang versucht und in dem Modul “Detector.py” zusammengestellt. Eine Grundidee entstammt der Internetseite: <https://rdmilligan.wordpress.com/2015/03/01/road-sign-detection-using-opencv-orb/>.

Informationen zur eingesetzten Technologie sind auf der benannten Internetseite nachlesbar.

Eine weitere Variante arbeitet mit Farbfiltren. Diese Lösung ist vergleichsweise einfach und liefert akzeptable Ergebnisse. Analysiert wird nach roten, gelben und blauen Objekten im Gesamtbild. Hier die aktuellen Filtereinstellungen:

```
# Verbote (rot), Hauptstr. (gelb), Gebote (blau)
upLim = [[50, 50, 255], [50, 255, 255], [ 255, 50, 50]]
lowLim= [[ 0, 0, 60], [ 0, 80, 80], [ 70, 0, 0]]
```

Rot sind die Verbotsschilder, Blau die Gebotsschilder und Gelb wird für das Hauptstraßenschild benötigt. Das Gesamtbild wird für diese drei Farbeinstellungen mehrfach durchlaufen, bis kein weiteres Objekt einer Farbklasse erkannt wird.

Python unterstützt mit der Bibliothek “cv2” das Suchen von Farbobjekten. Der entscheidende Befehl lautet:

```
mask = cv2.inRange(<Straßenbild>, <untere Farbgrenzen>, <obere Farbgrenzen>)
```

Der komplette Quelltext ist in der Klassenmethode “detectContur(self,image, frameObjImage,filledObjImage)” zu finden.

Es werden allerdings auch farbige Nicht-Straßenschilder detektiert, zum Beispiel ein rotes Auto. Ein Ansatz beruht darauf, dass das CNN-Model diese als Nicht-Straßenschilder erkennt und aussondert.

Zur generellen Schildererkennung bedarf es eines erheblichen Entwicklungsaufwandes. Zum Beispiel haben Größenverhältnisse einen entscheidenden Einfluss auf den Erkennungserfolg und natürlich auch die Lichtverhältnisse.

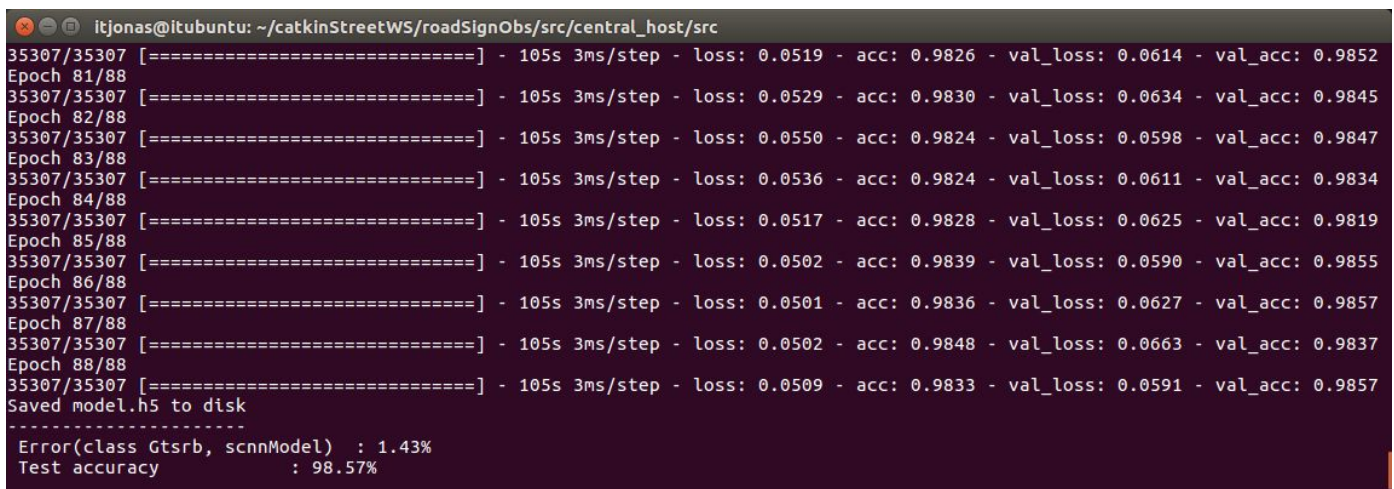


Das Modul "PublishCam.py" kann sowohl Bilder einer WebCam, aber auch vorhandene Bilddateien bereitstellen. Darüber entscheidet der Parameter `USE_WEBCAM = False/True`. Wird keine WEBCAM genutzt, so werden zufällige Straßenbilder dem Verzeichnis `"./objDetect/street/*.jpg"` entnommen. Zur Steuerung dient auch hier eine csv-Datei (`roadPictures.csv`).

## 8. Test und Testergebnisse

### 8.1 Training

Das Modul `Prediction.py` mit seinem Klassenmodul `Cnn.py` sucht nach dem Start bereits trainierte Modelldaten in den Dateien `cnnGtsrbModel.json` und `cnnGtsrbModel.h5`. Sind diese nicht vorhanden, so wird ein Trainingslauf durchgeführt. Bei einem Trainingslauf über 88 Epochen mit dem Modell "lcnnModel" wurde eine Testgenauigkeit von über 98 % erzielt. Siehe Abbildung:



```

Itjonas@itubuntu: ~/catkinStreetWS/roadSignObs/src/central_host/src
35307/35307 [=====] - 105s 3ms/step - loss: 0.0519 - acc: 0.9826 - val_loss: 0.0614 - val_acc: 0.9852
Epoch 81/88
35307/35307 [=====] - 105s 3ms/step - loss: 0.0529 - acc: 0.9830 - val_loss: 0.0634 - val_acc: 0.9845
Epoch 82/88
35307/35307 [=====] - 105s 3ms/step - loss: 0.0550 - acc: 0.9824 - val_loss: 0.0598 - val_acc: 0.9847
Epoch 83/88
35307/35307 [=====] - 105s 3ms/step - loss: 0.0536 - acc: 0.9824 - val_loss: 0.0611 - val_acc: 0.9834
Epoch 84/88
35307/35307 [=====] - 105s 3ms/step - loss: 0.0517 - acc: 0.9828 - val_loss: 0.0625 - val_acc: 0.9819
Epoch 85/88
35307/35307 [=====] - 105s 3ms/step - loss: 0.0502 - acc: 0.9839 - val_loss: 0.0590 - val_acc: 0.9855
Epoch 86/88
35307/35307 [=====] - 105s 3ms/step - loss: 0.0501 - acc: 0.9836 - val_loss: 0.0627 - val_acc: 0.9857
Epoch 87/88
35307/35307 [=====] - 105s 3ms/step - loss: 0.0502 - acc: 0.9848 - val_loss: 0.0663 - val_acc: 0.9837
Epoch 88/88
35307/35307 [=====] - 105s 3ms/step - loss: 0.0509 - acc: 0.9833 - val_loss: 0.0591 - val_acc: 0.9857
Saved model.h5 to disk
-----
Error(class Gtsrb, scnnModel) : 1.43%
Test accuracy                : 98.57%
```

Abbildung : Trainingsergebnis mit 88 Epochen

Ab der 20. Epoche erreichte die Testgenauigkeit bereits etwa 95%. Danach erfolgte folglich nur noch eine geringe Verbesserung.

Grundsätzlich ist nach einer Ergänzung der Trainingsdaten oder nach einer Änderung der Modellparameter (Bildgröße, Modellversion, ...) ein erneutes Training durchzuführen. Folglich sind in diesem Fall die Modelldateien zu entfernen.

### 8.2 Einzelne Verkehrszeichen erkennen

Gestartet werden die Programme vorzugsweise in der folgenden Reihenfolge:

`./roadSignObs/src/central_host/src/Prediction.py`

```
./roadSignObs/src/car_client/src/SubscribCarCrt.py
./roadSignObs/src/car_client/src/PublishPic.py
```

Die folgende Abbildung zeigt die zugehörige Kommunikationsstruktur.

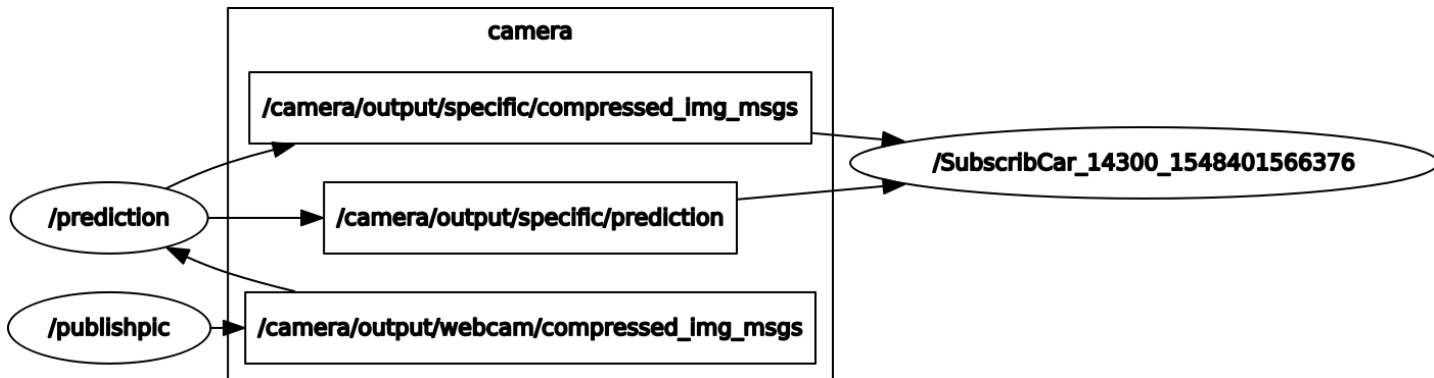


Abbildung : Graph mit Nodes und Topics

Grundsätzlich können die Programmierer mit `roslaunch <package> <programm.py>` oder direkt aus dem jeweiligen Verzeichnis mit `python <programm.py>` gestartet werden.

Es ist grundsätzlich zu beachten, dass der Zugriff auf die Bildverzeichnisse (TrainingsImages und TestImages) direkt aus dem Startverzeichnis gewährleistet wird. Aktuell sind diese ein Unterverzeichnis der jeweiligen Programmverzeichnisse.

Es wird davon ausgegangen, dass "roscore" bereits aktiv ist und dass ein Modelltraining bereits stattgefunden hat. Die Trainingsmodell-Dateien befinden sich ebenfalls im Startverzeichnis.

Nach dem Start von `Prediction.py` meldet sich das Programm mit Testinformationen und schließlich mit der Ausgabe:

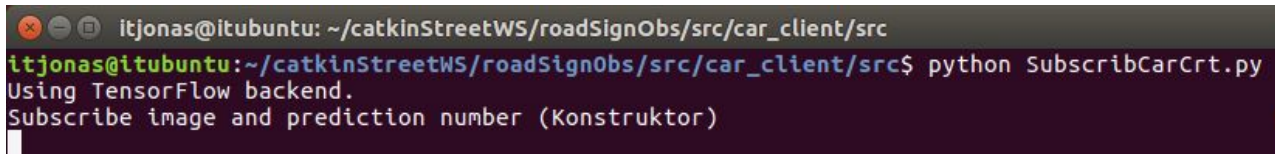
```

--- print in predictionTestImage() ---
index of the picture: 6
prediction label      : 1
real label           : 1
Bereit zum Empfang der Bilder zur Prediction ...

```

Abbildung : `Prediction.py` aktiv

Das Programm `SubscribCarCrt.py` meldet sich, solange keine Nachrichten empfangen werden, ebenfalls nur mit einer kurzen Information:



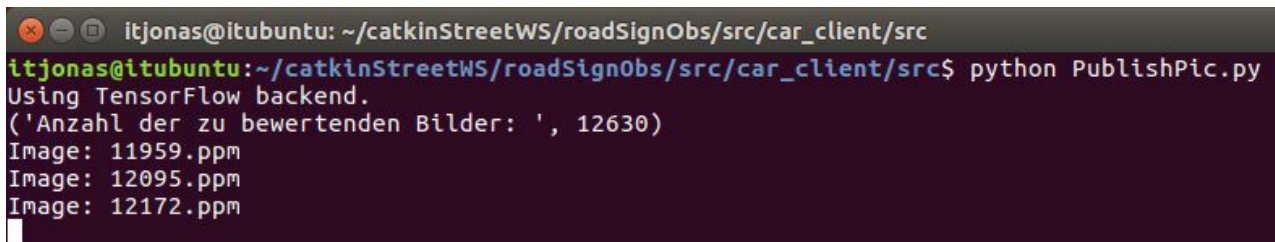
```

itjonas@itubuntu: ~/catkinStreetWS/roadSignObs/src/car_client/src
itjonas@itubuntu:~/catkinStreetWS/roadSignObs/src/car_client/src$ python SubscribCarCrt.py
Using TensorFlow backend.
Subscribe image and prediction number (Konstruktor)

```

Abbildung : SubscribCarCrt aktiv

Leben kommt erst in das System mit dem Start von "PublishPic.py". Dieses Programm wählt zufällig Bilder mit Verkehrszeichen aus dem Verzeichnis TestImages aus und sendet diese zum Node Prediction.py



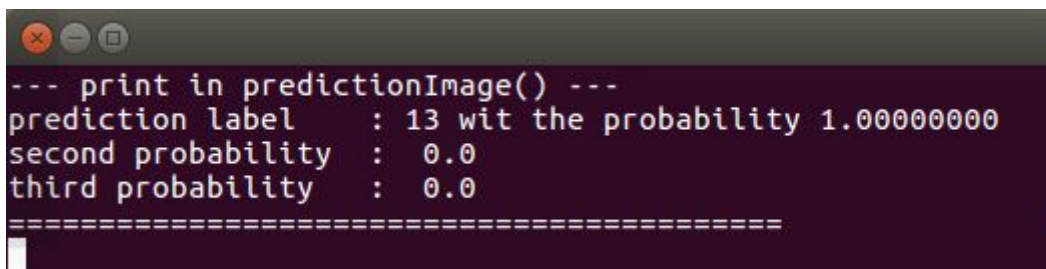
```

itjonas@itubuntu: ~/catkinStreetWS/roadSignObs/src/car_client/src
itjonas@itubuntu:~/catkinStreetWS/roadSignObs/src/car_client/src$ python PublishPic.py
Using TensorFlow backend.
('Anzahl der zu bewertenden Bilder: ', 12630)
Image: 11959.ppm
Image: 12095.ppm
Image: 12172.ppm

```

Abbildung : PublishPic.py sendet Bilder mit Verkehrszeichen

In "Prediction.py" wird die Label-Nummer (prediction-label) erkannt. Zusätzlich werden Wahrscheinlichkeitswerte (probability) für diese Vorhersage ausgegeben. Für das Vorfahrtsschild beträgt der Wahrscheinlichkeitswert 1. Die beiden folgenden, nächstgelegenen Wahrscheinlichkeitswerte sind in diesem Beispiel 0.



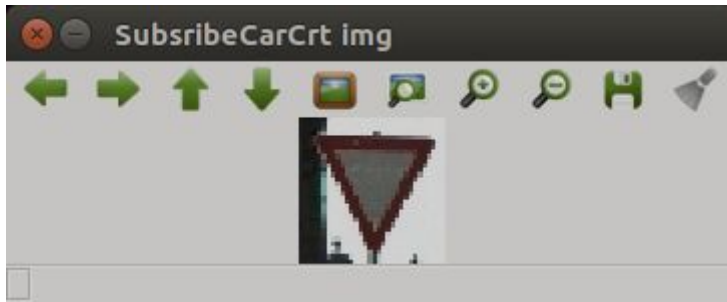
```

--- print in predictionImage() ---
prediction label      : 13 wit the probability 1.00000000
second probability    : 0.0
third probability     : 0.0
=====

```

Abbildung : Prediction.py klassifiziert Verkehrszeichen (Klassennummer 13)

Das Bild mit der Label-Nummer und deren Bewertung werden zum Programm SubscribCarCrt.py weitergeleitet. In dem Programm wird zur Labelnummer zusätzlich ein aussagefähiger Text generiert und das Bild des Verkehrszeichens ausgegeben.

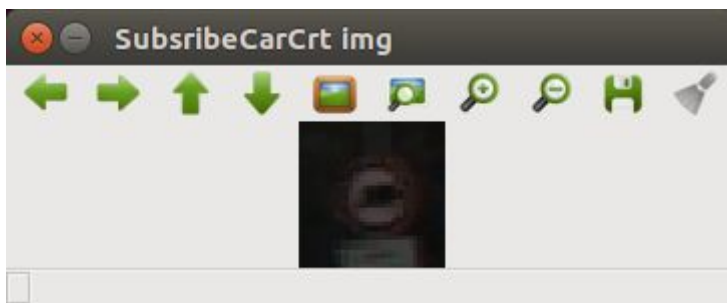


```
itjonas@itubuntu: ~/catkinStreetWS/roadSignObs/src/car_client/src
Label of the predicted road sign: 31 : Achtung: Wildwechsel
Label of the predicted road sign: 2 : Geschwindigkeitsbegrenzung 50 km/h
Label of the predicted road sign: 14 : Stop: Halten und Vorfahrt beachten
Label of the predicted road sign: 13 : Vorfahrt beachten
```

Abbildung : SubscribCarCrt.py zeigt das aktuelle Verkehrszeichen und gibt Bezeichnung aus

Grundsätzlich werden deutlich dargestellte Verkehrszeichen sehr gut erkannt. Für schlecht beleuchtete Verkehrszeichen und Verkehrszeichen aus einer ungünstigen Perspektive ist die Erkennung nicht immer eindeutig.

Hier ein Beispiel:



```
itjonas@itubuntu: ~/catkinStreetWS/roadSignObs/src/car_client/src
Label of the predicted road sign: 8 : Geschwindigkeitsbegrenzung 120 km/h
Label of the predicted road sign: 38 : rechts halten
Label of the predicted road sign: 28 : (SEHR UNSICHER) Achtung: Kinder
Label of the predicted road sign: 16 : (UNSICHER) Einfahrt für LKWs verboten
```

Abbildung : Verkehrszeichen wurde vom Modul Prediction.py nicht eindeutig erkannt

In diesem Fall wurde das Verkehrszeichen noch richtig erkannt, aber der zugehörige Wahrscheinlichkeitswert liegt unter 1 (0.99), so dass ein Kommentar (UNSICHER) als Zusatztext vermerkt wird.

### 8.3 Verkehrszeichen in einer Straßenszene detektieren und erkennen

Gestartet werden die Programme vorzugsweise in der folgenden Reihenfolge:

```
./roadSignObs/src/central_host/src/Prediction.py
./roadSignObs/src/car_client/src/SubscribCarCrt.py
./roadSignObs/src/car_client/src/PublishCam.py
```

Anstelle des Programms PublishPic.py liefert jetzt das Programm PublishCam.py die Bilder der Verkehrszeichen, die mittels Filter aus einer Straßenszene detektiert werden.

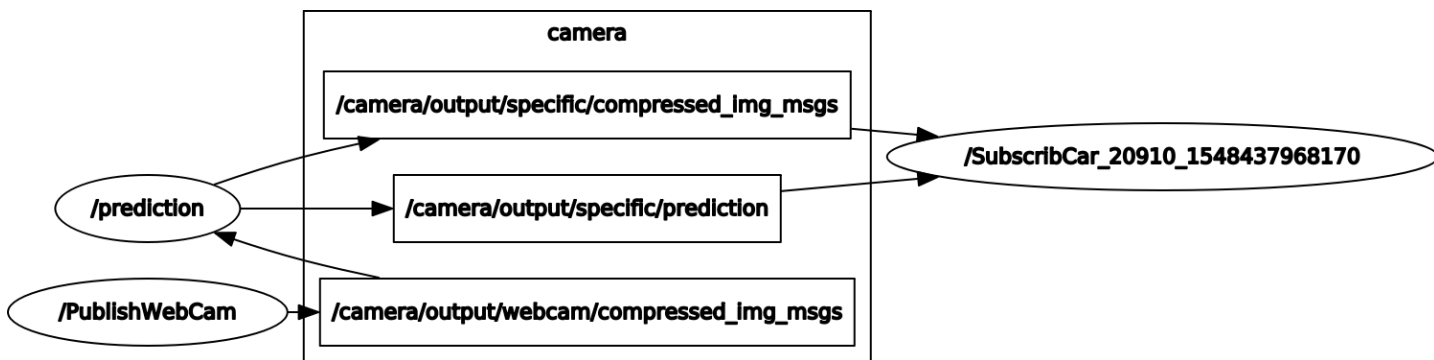


Abbildung : Graph mit Nodes und Topics

Die folgende Abbildung zeigt beispielhaft eine Tiefgaragenausfahrt mit den detektierten Verkehrsschildern.



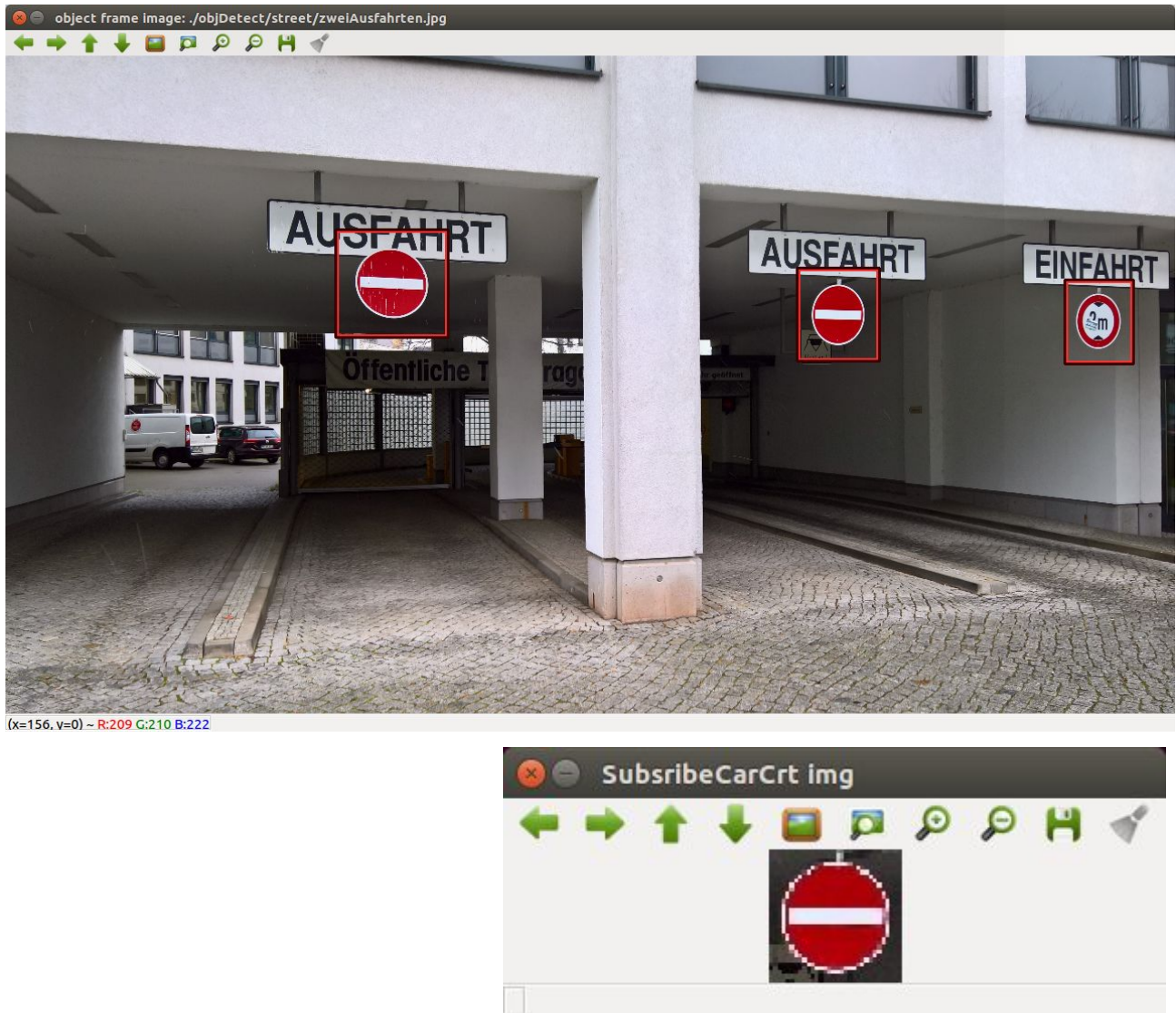


Abbildung : Straßenszene und detektiertes Verkehrszeichen

Aus dem Verzeichnis “./car\_client/src/objDetect/street” werden zufällige Straßenszenen ausgewählt und analysiert. Erkannte Verkehrszeichen werden zur Anschauung im Szenenbild umrandet. Diese Bildausschnitte werden zur Erkennung zum Modul “Prediction.py” übergeben. Dort erfolgt, wie bereits besprochen, die eigentliche Arbeit. Das Vorhersageergebnis wird im Anschluss an das Modul SubscribCarCrt.py gesandt.

Anstelle der Szenenbilder kann eine WebCam direkt Bilder liefern. Dazu ist im Modul “PublishCam.py” der Parameter “USE\_WEBCAM=True” entsprechend zu setzen. Die folgende Abbildung soll diese Funktionsweise demonstrieren.



Abbildung : WebCam zur Bilderfassung

## 9. Zusammenfassung, Schlussfolgerungen

- Die Klassifizierung von Verkehrszeichen mit Hilfe eines "Convolutional Neuronalen Netzes" (CNN) ist realisierbar. Die Erkennungsrate liegt bei über 90% (95%).
- Es ist notwendig, komplette Straßenszenen zu analysieren und die darin enthaltenen Verkehrsschilder zu detektieren. Mehrere Verfahren zur Lösung dieser Aufgabe sind denkbar. Praktikabel erscheint die Möglichkeit über Farbfiler eine Vorauswahl möglicher Objekte zu treffen.
- In Frage kommende Objekte sind von dem neuronalen Netz zu klassifizieren. Es müssen aber auch Nicht-Verkehrsschilder als solche erkannt werden. Dazu eignen sich die Wahrscheinlichkeitswerte der Klassenzugehörigkeit der zu analysierenden Objekte. Ist die Zuordnung eines Objektes zu einer Klasse nicht eindeutig, so könnte es auch kein Verkehrszeichen sein. Fehleinschätzungen sind möglich.
- Zusätzliche Klassen für Nicht-Verkehrszeichen können die Erkennungsrate verbessern. Typische Nicht-Verkehrszeichen sind zum Beispiel sonstige Schilder, Werbung, Einkaufstaschen, Fahrzeuge.
- Die Bildqualität (Auflösung und Belichtung) der Straßenszenen hat einen direkten Einfluss auf die Erkennungsrate. Eine Skalierung der Bilder auf ein einheitliches Format aber auch eine Kontrastanpassung der Bilder können hilfreich sein.

## Quellen

(1) ROS Tutorial (last edited 2018-08-18 07:30:27)

<http://wiki.ros.org/ROS/Tutorials>

(2) Jason Brownlee “Deep Learning With Python”

<https://machinelearningmastery.com/deep-learning-with-python/>

(3) The German Traffic Sign Recognition Benchmark

<http://benchmark.ini.rub.de/>

(4) “Save and Load Your Keras Deep Learning Models”

<https://machinelearningmastery.com/save-load-keras-deep-learning-models/>

(5) Thomas Theis “Einstieg in Python”

Galileo Computing, Bonn 2011

ISBN 978-3-8362-1738-5