KLASYFIKACJA ZNAKÓW DROGOWYCH Z WYKORZYSTANIEM KONWOLUCYJNYCH SIECI NEURONOWYCH

Jakub Tomaszewski

Spis treści

1. Opis i cel projektu

Założeniem projektu było stworzenie modelu klasyfikacyjnego, którego zadaniem było rozpoznawanie znaków drogowych. Klasyfikacji, jest typowym problemem rozwiązywanym przez algorytmy sztucznej inteligencji i polega na przypisywaniu określonym próbkom(w tym przypadku jest to zdjęcie znaku) konkretnych klas.

2. Dataset – zbiór danych

Wykorzystany zbiór danych pochodzi z benchmark.ini.rub.de i podzielony jest na trzy sekcje:

- Zbiór treningowy – służący do trenowania modelu (34799 próbek)
- Zbiór walidacyjny – służący do testowania modelu podczas trenowania (4410 próbek)
- Zbiór testowy – służący do testowania modelu po zakończeniu trenowania (12630 próbek)

Zbiór treningowy zawiera największą liczbę próbek, gdyż to na nim został wyuczony model. Każdy obraz ma format 32x32x3 gdzie pierwsze dwie wartości oznaczają szerokość oraz wysokość, a trzecia kanały kolorów tzw. color channels (RGB). Zbiór danych zawiera zdjęcia dla 43 różnych klas znaków.

3. Wybór języka

Python jest obecnie najpopularniejszym oraz stale rozwijanym językiem programowania. Istnieją setki modułów rozszerzających jego funkcjonalność i ułatwiających implementację zaawansowanych projektów. Powyższe powody zdecydowały o wykorzystaniu tego języka w wersji python=3.6.9

4. Wykorzystane moduły

- glob2==0.7 – wyszukiwanie plików w folderze
- pandas==1.0.5 – import datasetów
- numpy==1.18.5 – manipulacja danymi
- matplotlib==3.2.2 – wizualizacja danych
- openpyxl==2.5.9 – wizualizacja, data augmentation
- scikit-image==0.16.2 – data augmentation
- tensorflow==2.2.0 – tworzenie modelu klasyfikacyjnego
- Keras==2.3.1 – tworzenie modelu klasyfikacyjnego

5. Ogólny plan i opis rozwiązania

a. Zebranie danych

Początkowym i jednocześnie najważniejszym krokiem było zebranie danych oraz zaimportowanie ich do workspace'a (w tym przypadku Colab Notebook)

W celu załadowania danych utworzone zostały trzy funkcje:

```
def get_pickle_files_names(path=''):
  """Returns all pickle file names from a directory
  Parameters
  ----------
  path -- path to a directory
    default = ''

  Returns
  -------
  List of all files ending with '.p' from a specific directory
  """

  return glob.glob(path+'*.p')
```

```
def load_pickle_file(path=None):
  """Loads a pickle file

  Parameters
  ----------
  path -- path to a file
    default = None

  Returns
  -------
  Content of a pickle file
  """

  return pd.read_pickle(path)
```

```
def load_csv_file(path=None):
  """Loads a csv file

  Parameters
  ----------
  path -- path to a file
    default = None

  Returns
  -------
  Content of a csv file
  """

  return pd.read_csv(path)
```

Następnie dane załadowano do worskspac'e przy użyciu funkcji `load_picke_file()`

```
X_train_data, y_train_data = train_data['features'], train_data['labels']
X_test_data, y_test_data = test_data['features'], test_data['labels']
X_val_data, y_val_data = validation_data['features'], validation_data['labels']
```

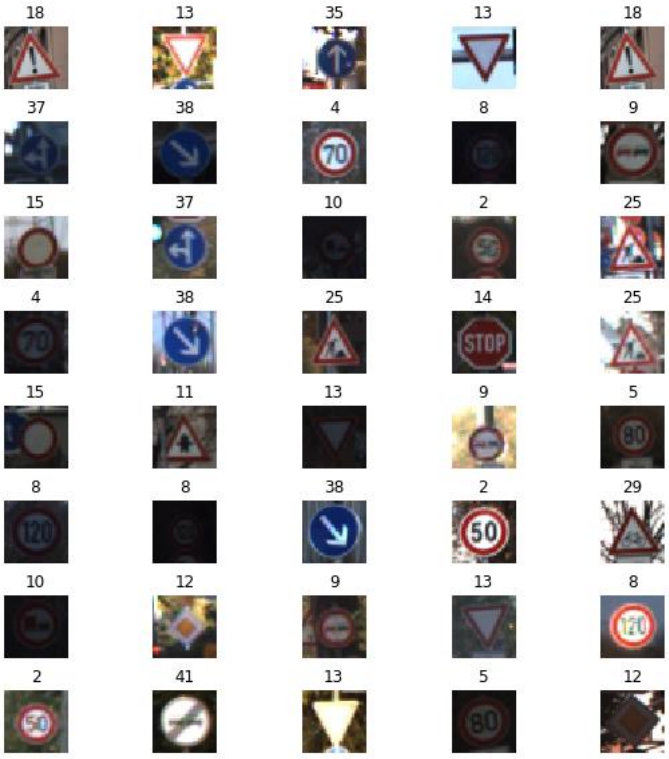
b. Wizualizacja oraz badanie zbioru danych

Zbiór zawiera 34799 przykładów trenujących, 12630 testujących oraz 4410 walidacyjnych. Każdy przykład jest obrazem o wymiarach 32x32x3

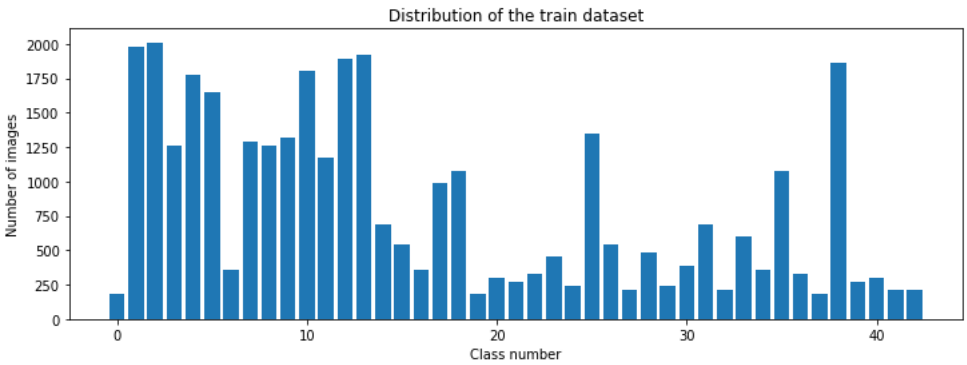
```
print(X_train_data.shape, y_train_data.shape)
print(X_test_data.shape, y_test_data.shape)
print(X_val_data.shape, y_val_data.shape)
```

```
(34799, 32, 32, 3) (34799,)
(12630, 32, 32, 3) (12630,)
(4410, 32, 32, 3) (4410,)
```

Dostępne są przykłady treningowe dla 43 różnych klas tzn. dla 43 różnych znaków.



Ilość przykładów trenujących dla każdej z klas jest różna co uwzględniono przy dalszej analizie.



c. Utworzenie nowych danych (Data augmentation)

W celu utworzenia większej liczby próbek i tym samym zwiększenia uniwersalności naszego modelu (tzw. data augmentation) wykorzystano dwie techniki:

- Manual data augmentation

  Pierwszym sposobem było wykorzystanie funkcji dostępnych w module OpenCV do wytworzenia nowych próbek.

```
def data_augment(image):
  """Applies rotation, warp and biteral filtering to an image

  Parameters
  ----------
  image -- numpy array representing an image

  Returns
  -------
  Image after all transformations
  """

  rows= image.shape[0]
  cols = image.shape[1]

  # Rotating
  M_rot = cv2.getRotationMatrix2D((cols/2,rows/2),10,1)

  # Translating
  M_trans = np.float32([[1,0,3],[0,1,6]])

  img = cv2.warpAffine(image,M_rot,(cols,rows))
  img = cv2.warpAffine(img,M_trans,(cols,rows))

  # Bilateral filtering
  img = cv2.bilateralFilter(img, 9, 75, 75)

  return img
```

- ImageDataGenerator

  Drugim sposobem na wygenerowanie większej ilości próbek było wykorzystanie klasy **ImageDataGenerator** z modułu Keras

  Początkowo zainicjowano klasę i ustalono parametry transformujące

```
data_geterator = ImageDataGenerator(
                                    width_shift_range=0.1,
                                    height_shift_range=0.1,
                                    zoom_range=0.2,
                                    shear_range=0.1,
                                    rotation_range=10
                                    )
```

Następnie przekazano dane treningowe do generatora oraz wytrenowano model, który podczas trenowania tworzył nowe próbki

```
data_geterator.fit(X_train)
```

```
history = model.fit_generator(
                              data_geterator.flow(X_train, y_train, batch_size=300),
                              steps_per_epoch=200,
                              epochs=10,
                              validation_data=(X_val, y_val),
                              shuffle=1,
                              verbose=1
                              )
```

d. Obrabianie danych (Data preprocessing)

W celu dokonania preprocessingu danych utworzone zostały 4 funkcje:

1. `convert_to_grayscale()` – konwertuje obraz w skalę szarości (pozostaje pojedynczy color channel). Po wywołaniu tej funkcji, obraz ma kształt 32x32x1

```
def convert_to_grayscale(image):
  """Converts an image to grayscale

  Parameters
  ----------
  image -- a numpy array represeting an image wth 3 color channels

  Returns
  -------
  image in grayscale without a color channel;
  """

  return cv2.cvtColor(image.astype('uint8') * 255, cv2.COLOR_BGR2GRAY)
```

2. `histogram_equalization()` – zwiększa globalny kontrast(nasycenie) obrazu

```
def histogram_equalization(image):
  """
  Parameters
  ----------

  image -- a numpy array representing an image in grayscale (without a color channel)

  Returns
  -------

  image of equalized grayscale
  """

  return cv2.equalizeHist(image)
```

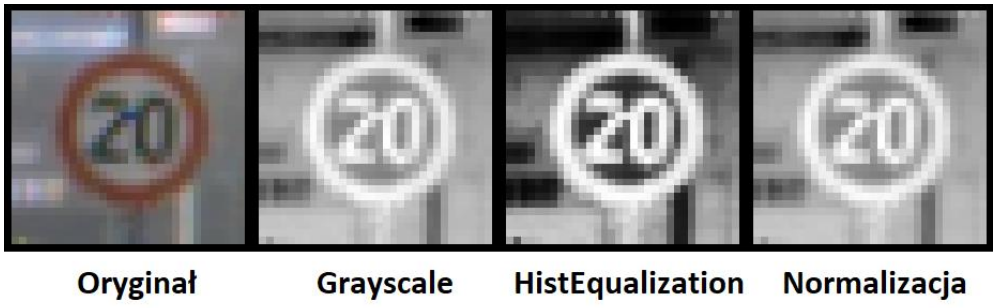
3. `normalize_image()` – Normalizuje obraz w taki sposób, aby wartości poszczególnych pikseli znajdowały się w przedziale od 0 do 1. Wpływa to korzystnie na działanie klasyfikatora.

```
def normalize_image(image):
  """Normalizes an image by dividing it by 255

  Parameters
  ----------
  image -- a numpy array representing an image

  Returns
  -------
  Normalized image
  """
  return image / 255
```

Tzw. Pipeline czyli funkcja łącząca wszystkie poprzednie funckje w jedną całość tak, aby zbiór danych mógł być w łatwy sposób obrobiony.

```
def preprocess_image(image):
  """Applies grayscale transformation, equalization and normalization to an image

  Parameters
  ----------

  image -- a numpy array representing an image

  Returns
  -------
  preprocessed image

  """
  gray = convert_to_grayscale(image)
  equalized = histogram_equalization(gray)
  normalized = normalize_image(equalized)

  return normalized
```

Przykład wykorzystania poszczególnych funkcji



6. Keras

Keras, to jedno z wielu ogólnodostępnych API biblioteki Tensorflow. Umożliwia ono w prosty oraz przejrzysty sposób tworzenie modeli uczenia maszynowego.

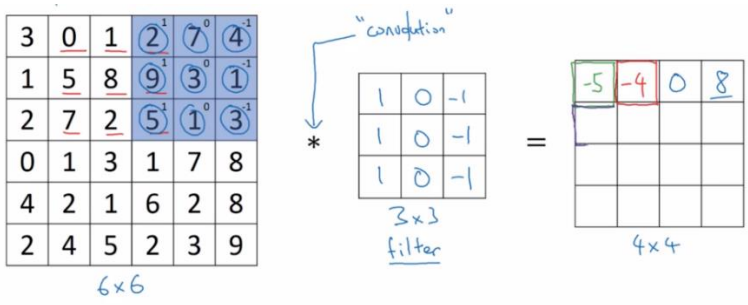
Jak pisze sam producent:

„Keras is an API designed for human beings, not machines. Keras follows best practices for reducing cognitive load: it offers consistent & simple APIs, it minimizes the number of user actions required for common use cases, and it provides clear & actionable error messages. It also has extensive documentation and developer guides."

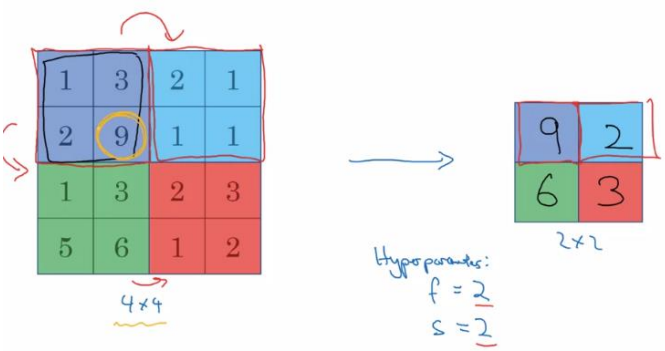
7. Wykorzystana architektura sieci i opis poszczególnych elementów

Wykorzystana architektura sieci znajduje się poniżej. Znajdują się w niej takie layery jak:

- Convolutional layer – Przy użyciu wielu filtrów stara się wydobyć specyficzne cechy danego obrazu. Każdy piksel z danej części obrazu zostaje pomnożony przez odpowiadającą mu liczbę, następnie te pomnożone wartości zostają zsumowane i wpisane jako nowy piksel do nowej macierzy reprezentującej obraz

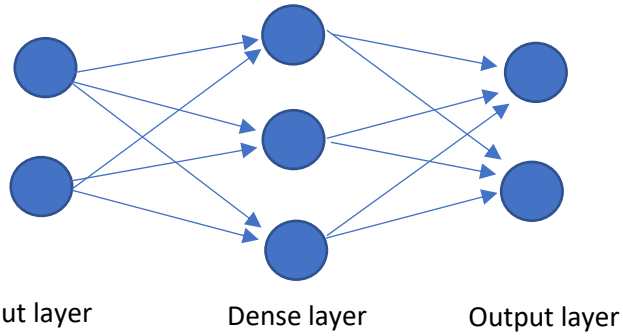


- Max pooling layer – zmniejsza wielkość obrazu przy użyciu filtra, który po nałożeniu na daną część obrazu wydobywa piksel o największej wartości.



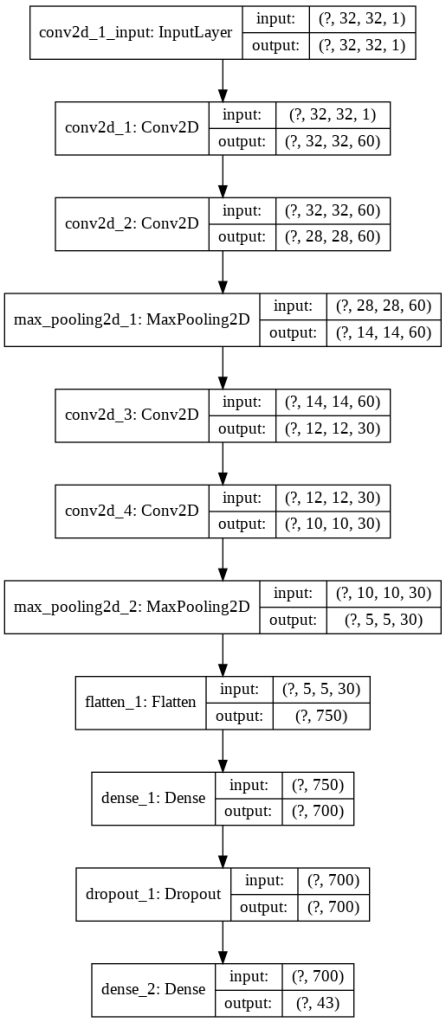
- Flattening layer – zmienia obraz w wektor kolumnowy (o jednej kolumnie) – reshape(-1, 1)

- Dense layer – to layer, w którym każdy node jest połączony z każdym node'm w następnym layerze



- Dropout layer – zapobiega overfittingu wyrzucając w każdej iteracji losową część próbek

Architektura sieci:

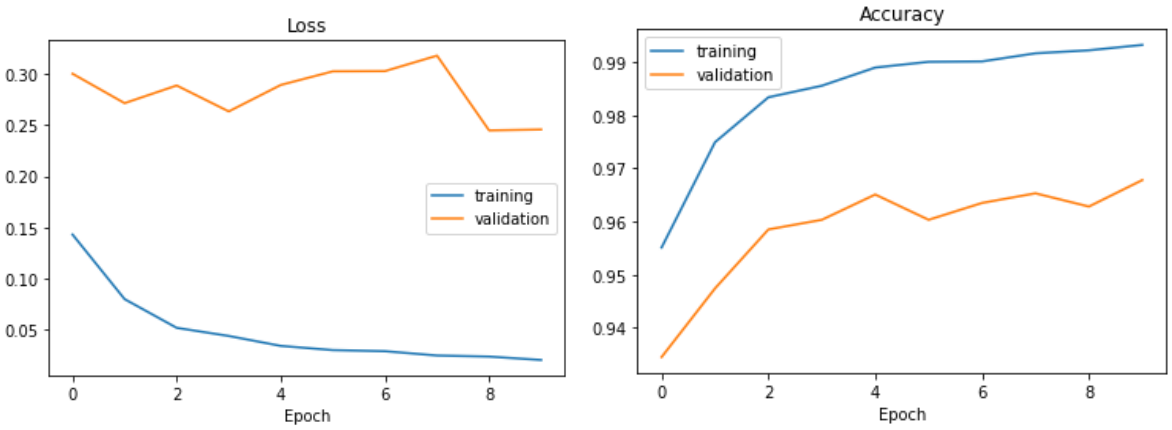


8. Wyniki

a. Manual data augmentation

Pomimo wielokrotnego trenowanie modelu na przygotowanym dodatkowo datasecie, cały czas zmagał się on ze zjawiskiem overfittingu (przetrenowania). Overfitting, to zbytnie dopasowanie się modelu do danych trenujących w taki sposób, że nie jest on w stanie poprawnie sklasyfikować nowych danych.

Rezultaty trenowania, walidacji oraz testowania dostępne są poniżej.



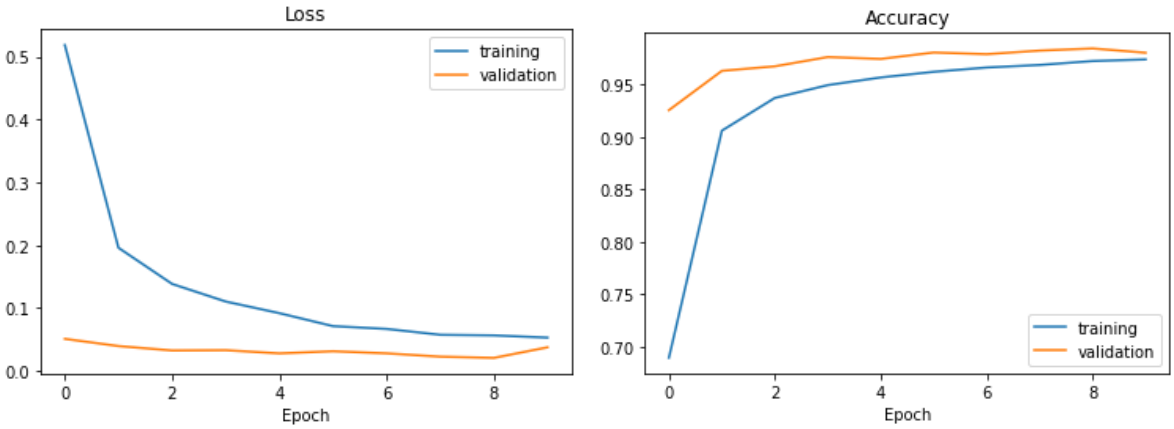

```
loss: 0.0206 - accuracy: 0.9933 - val_loss: 0.2456 - val_accuracy: 0.9678
```

```
Test score: 0.28678739776557216
Test accuracy: 0.9457640647888184
```

Powyższe wykresy przedstawiają przetrenowanie użytego modelu. Świadczy o tym znacznie wyższa dokładność modelu dla danych trenujących, niż dla danych walidacyjnych jak i testowych. Powodów powstania zjawiska overfittingu może być wiele. Najczęstszym jednak jest niewłaściwe dostosowanie hyperparametrów np. liczby iteracji. W tym przypadku pomimo wielokrotnej zmiany hyperparametrów nie udało się uzyskać zadowalających wyników. Problem tkwił w algorytmie przygotowującym dodatkowe dane, dlatego też zdecydowano użyć się klasy ImageDataGenerator dostępnej w module Keras.

b. ImageDataGenerator

Wykorzystanie **ImageDataGenerator** z modułu Keras pozwoliło na otrzymanie dużo bardziej obiecujących wyników, które widoczne są poniżej.




```
loss: 0.0531 - accuracy: 0.9839 - val_loss: 0.0377 - val_accuracy: 0.9882
```

```
Test loss: 0.1354836595316578
Test accuracy: 0.9695962071418762
```

Pomimo tego, że dokładność na zbiorze treningowym zmalała, znacząco wzrosła dokładność walidacyjna jak i testowa. W tym przypadku nie występuje zjawisko overfittingu i nasz model prawidłowo klasyfikuje większość wcześniej nie widzianych obrazów. Tworzenie próbek podczas samego trenowania okazało się w tym przypadku dużo bardziej wydajne.

Model, który uzyskał wyniki został zapisany w formie pliku o formacie **.p** z wykorzystaniem modułu **pickle**.

9. Klasyfikacja w czasie rzeczywistym

W celu weryfikacji działania modelu na danych rzeczywistych, utworzony został moduł `sign_detection.py .` Jego działanie prezentują zdjęcia poniżej.

10. Bibliografia

- *„Hands-On Machine Learning with Scikit-Learn and TensorFlow" Aurélion Géron*
- *„Deep Learning Specialization - Coursera" Andew Ng*
- *„Machine Learning Scientist Career Track" DataCamp*
- *„The Complete Self-Driving Car Course - Applied Deep Learning" Udemy*
- https://towardsdatascience.com/recognizing-traffic-signs-with-over-98-accuracy-using-deep-learning-86737aedc2ab -
- https://www.coursera.org/projects/traffic-sign-classification-deep-learning