AP_L16_20200323_1700_G02

March 31, 2020

1 Sprawozdanie wykonali:

1. Artur Połeć
2. Maciej Stroiński
3. Wojciech Pełka

2 AP/LAB16

2.1 Wprowadzenie

Tematem tego ćwiczenia jest wytrenowanie klasyfikatora znaków drogowych w oparciu o konwolucyjną sieć neuronową.

Jako zbiór danych wykorzystamy zbiór powstały na potrzeby German Traffic Sign Recognition Benchmark.

Przed przystąpieniem do laboratorium należy zapoznać się biblioteką PyTorch. Polecam do tego serię tutoriali dostępną na stronie projektu, a szczególnie ten tutorial oraz ten. Wymaganą wiedzą jest wiedza z wykładu odnośnie systemów wizyjnych i sztucznej inteligencji, a także podstawowa wiedza z zakresu analizy matematycznej, algebry liniowej, metod numerycznych czy programowania w języku Python.

Laboratorium ma formę wirtualnego notebooka, który można dowolnie edytować i przez to eksperymentować we własnym zakresie. W odpowiednich miejscach należy wykonać dodatkowe polecnia, które wraz z wykonanym kodem stanowić będą sprawozdanie. Sprawozdanie można wygenerować bezpośrednio z poziomu tego notebooka, wybierając z opcji *File -> Export Notebook As... -> Export Notebook to PDF*. Może być konieczna instalacja dodatkowych narzędzi w celu wygenerowania pliku PDF. Gdyby były z tym problemy, można zawsze wygenerować plik w formacie HTML, a następnie "wydrukować" go do PDF-a. Krótki tutorial jak korzystać z tego narzędzia można znaleźć np. tutaj.

W celu zaliczenia laboratorium należy przesłać wyeksportowany notebook z uzupełnionymi polami i odpowiedziami na platformę Moodle. Plik powinien być w formacie PDF i posiadać nazwę w formacie *AP_L16___OSOBA_A___OSOBA_B.pdf*, gdzie* OSOBA_A* i *OSOBA_B* oznacza imię i nazwisko odpowiednio pierwszej i drugiej osoby w zespole w formacie *IMIE_NAZWISKO*. **Plik PDF powinien zawierać rezultaty uruchomienia wszystkich niezbędnych do wykonania ćwiczenia poleceń.**

Notebook można uruchomić zarówno lokalnie, instalując paczkę `jupyterlab`, a następnie wołając polecenie `jupyterlab`, jak i korzystając z platformy Google Colab. Google Colab umożliwia wgranie dowolnego notebooka i darmowe uruchomienie go na wydajnych maszynach, również z dostępem do GPU, a nawet TPU. Mogą jednak wystąpić pewne ograniczenia w dostępie do akceleratorów w zależności od obciążenia serwerów.

2.2 Zbiór danych

Zbiór GTSRB składa się z ponad 50,000 obrazów przyporządkowanych do ponad 40 klas. Pobrać go można z tej strony. Potrzebne nam będą pliki: * GTSRB-Training_fixed.zip - zawiera zbiór treningowy * GTSRB_Final_Test_Images.zip - zawiera zbiór testowy * GTSRB_Final_Test_GT.zip - zawiera etykiety dla obrazów ze zbioru testowego

Poniżej znajduje się kod, który automatyzuje proces pobierania oraz przygotowywania zbioru danych. Jeśli jednak z jakichś powodów nie zadziałał, należy ręcznie pobrać w/w pliki, a następnie utworzyć z nich następującą strukturę: * ./GTSRB * ./GTSRB/train * ./GRSRB/train/00000 * ./GRSRB/train/00000/00000_00000.ppm * ./GTSRB/test * ./GRSRB/test/00000 * ./GRSRB/test/00000/00243.ppm.ppm

`train` oraz `test` oznaczają odpowiednio typ zbioru danych, `00000-00042` jest identyfikatorem klasy, a ostatni człon to nazwa pliku. Zbiór treningowy jest już dostępny w takiej formie po rozpakowaniu archiwum. W przypadku zbioru testowego, należy samemu utworzyć taką strukturę wykorzystując informację o klasie, do której przynależy dany obrazek, zawartej w pliku CSV dostępnym po rozpakowaniu trzeciego archiwum.

2.3 Konfiguracja środowiska

Ten notebook wymaga do prawidłowego działania kilka dodatkowych modułów. Poniższe polecania powinny je pobrać i zainstalować.

[18]:
```
%pip install numpy pandas wget matplotlib
```

```
Requirement already satisfied: numpy in
/home/smaket/anaconda3/lib/python3.7/site-packages (1.17.2)
Requirement already satisfied: pandas in
/home/smaket/anaconda3/lib/python3.7/site-packages (0.25.1)
Requirement already satisfied: wget in
/home/smaket/anaconda3/lib/python3.7/site-packages (3.2)
Requirement already satisfied: matplotlib in
/home/smaket/anaconda3/lib/python3.7/site-packages (3.1.1)
Requirement already satisfied: pytz>=2017.2 in
/home/smaket/anaconda3/lib/python3.7/site-packages (from pandas) (2019.3)
Requirement already satisfied: python-dateutil>=2.6.1 in
/home/smaket/anaconda3/lib/python3.7/site-packages (from pandas) (2.8.0)
Requirement already satisfied: cycler>=0.10 in
/home/smaket/anaconda3/lib/python3.7/site-packages (from matplotlib) (0.10.0)
Requirement already satisfied: kiwisolver>=1.0.1 in
/home/smaket/anaconda3/lib/python3.7/site-packages (from matplotlib) (1.1.0)
```

```
Requirement already satisfied: pyparsing!=2.0.4,!=2.1.2,!=2.1.6,>=2.0.1 in
/home/smaket/anaconda3/lib/python3.7/site-packages (from matplotlib) (2.4.2)
Requirement already satisfied: six>=1.5 in
/home/smaket/anaconda3/lib/python3.7/site-packages (from python-
dateutil>=2.6.1->pandas) (1.12.0)
Requirement already satisfied: setuptools in
/home/smaket/anaconda3/lib/python3.7/site-packages (from
kiwisolver>=1.0.1->matplotlib) (41.4.0)
Note: you may need to restart the kernel to use updated packages.
```

Aby zainstalować biblioteki `torch` oraz `torchvision` odkomentuj linijkę, która odpowiada Twojemu środowiskowu. Jeśli korzystasz z Google Colab `torch` i `torchvision` powinny być już zainstalowane.

[19]:
```
# Linux, CPU-only
#%pip install torch==1.4.0+cpu torchvision==0.5.0+cpu -f https://download.
 ↪pytorch.org/whl/torch_stable.html

# Linux, CUDA 9.2
#%pip install torch==1.4.0+cu92 torchvision==0.5.0+cu92 -f https://download.
 ↪pytorch.org/whl/torch_stable.html

# Linux, CUDA 10.1
#%pip install torch torchvision

# Windows, CPU-only
#%pip install torch==1.4.0+cpu torchvision==0.5.0+cpu -f https://download.
 ↪pytorch.org/whl/torch_stable.html

# Windows, CUDA 9.2
#%pip install torch==1.4.0+cu92 torchvision==0.5.0+cu92 -f https://download.
 ↪pytorch.org/whl/torch_stable.html

# Windows, CUDA 10.1
#%pip install torch===1.4.0 torchvision===0.5.0 -f https://download.pytorch.org/
 ↪whl/torch_stable.html
```

Pracując z algorytmami uczenia maszynowego często korzystamy z generatorów liczb pseudolosowych. Aby zapewnić powtarzalność i reprodukowalność naszych eksperymentów możemy ręcznie zdefiniować tzw. ziarno losowości. Dzięki temu liczby wygenerowane za pomocą takich generatorów wciąż będą *losowe*, ale za każdym razem zostaną wygenerowane dokładnie te same liczby.

Poniższe polecenia ustawiają ziarno losowości trzech generatorów liczb losowych: `random` z biblioteki standardowej Pythona, `np.random` z biblioteki NumPy oraz `torch` z biblioteki PyTorch.

[20]:
```
import random
import numpy as np
import torch
```

```
SEED = 1337
random.seed(SEED)
np.random.seed(SEED)
torch.manual_seed(SEED)
print(f'Using seed: {SEED}')
```

```
Using seed: 1337
```

Jeśli masz dostęp do karty graficznej NVIDIA ze wsparciem dla technologii CUDA możesz wykorzystać jej potencjał i znacząco przyspieszyć obliczenia. Wymaga to zainstalowanego toolkitu CUDA oraz PyTorch-a w odpowiedniej wersji.

[21]:
```
DEVICE = 'cuda' if torch.cuda.is_available() else 'cpu'
print(f'Using device: {DEVICE}')
```

```
Using device: cpu
```

2.4 Pobranie i przygotowanie zbioru danych

Jeśli zbiór danych został już pobrany i umieszczony w odpowiednich katalogach, proszę przejść do następnego punktu.

[22]:
```
import os
import pathlib
import shutil
import ssl
import zipfile

import pandas as pd
import wget

ssl._create_default_https_context = ssl._create_unverified_context

# Training dataset
print('Downloading training dataset...')
wget.download('https://sid.erda.dk/public/archives/
 ↪daaeac0d7ce1152aea9b61d9f1e19370/GTSRB-Training_fixed.zip')
with zipfile.ZipFile('GTSRB-Training_fixed.zip', 'r') as zf:
    zf.extractall('./')
shutil.move('./GTSRB/Training', './GTSRB/train')

# Test dataset
print('Downloading test dataset...')
wget.download('https://sid.erda.dk/public/archives/
 ↪daaeac0d7ce1152aea9b61d9f1e19370/GTSRB_Final_Test_Images.zip')
with zipfile.ZipFile('GTSRB_Final_Test_Images.zip', 'r') as zf:
```

```
    zf.extractall('./')
shutil.move('./GTSRB/Final_Test/Images', './GTSRB/test_raw')

# Test dataset - labels
print('Downloading test dataset labels...')
wget.download('https://sid.erda.dk/public/archives/
 ↪daaeac0d7ce1152aea9b61d9f1e19370/GTSRB_Final_Test_GT.zip')
with zipfile.ZipFile('GTSRB_Final_Test_GT.zip', 'r') as zf:
    zf.extractall('./')
shutil.move('./GT-final_test.csv', './GTSRB/test_labels.csv')

# Transform test dataset directories structure
print('Transforming test dataset into torchvision dataset structure...')
labels = pd.read_csv('./GTSRB/test_labels.csv', sep=';')

raw_dir_path = pathlib.Path(f'./GTSRB/test_raw')
for _, label in labels.iterrows():
    src_im_path = raw_dir_path / label['Filename']
    class_dir_path = pathlib.Path(f'./GTSRB/test/{label["ClassId"]:05d}')
    class_dir_path.mkdir(parents=True, exist_ok=True)
    dst_im_path = class_dir_path / label['Filename']
    shutil.move(src_im_path, dst_im_path)

# Remove unneccessary files
print('Removing files...')
os.remove('./GTSRB-Training_fixed.zip')
os.remove('./GTSRB_Final_Test_Images.zip')
os.remove('./GTSRB_Final_Test_GT.zip')
os.remove('./GTSRB/Readme-Images-Final-test.txt')
os.remove('./GTSRB/test_labels.csv')
shutil.rmtree('./GTSRB/Final_Test')
shutil.rmtree('./GTSRB/test_raw')
```

```
Downloading training dataset…
Downloading test dataset…
Downloading test dataset labels…
Transforming test dataset into torchvision dataset structure…
Removing files…
```

2.5 Wizualizacja zbioru treningowego

W tej części wczytamy zbiór danych z dysku i zwizualizujemy jego elementy.

[23]:
```
import random
import torch
import torchvision
```

[24]:
```
from torchvision.datasets import ImageFolder

pure_dataset = ImageFolder(r'./GTSRB/train')
print(f'Dataset contains {len(pure_dataset)} examples')
```

```
Dataset contains 26640 examples
```

[25]:
```
from enum import IntEnum

class TrafficSign(IntEnum):
    S20 = 0
    S30 = 1
    S50 = 2
    S60 = 3
    S70 = 4
    S80 = 5
    EOS80 = 6
    S100 = 7
    S120 = 8
    NOC = 9
    NOT = 10
    INTERSECTION = 11
    PRIORITY = 12
    YIELD = 13
    STOP = 14
    NO_TRAFFIC = 15
    NO_TRUCKS = 16
    NO_ENTRY = 17
    HAZARD = 18
    CURVE_LEFT = 19
    CURVE_RIGHT = 20
    CURVES = 21
    POT_HOLES = 22
    SLIPPY = 23
    NARROWING = 24
    CONSTRUCTION_ZONE = 25
    TRAFFIC_LIGHTS = 26
    PEDESTRIANS = 27
    CHILDREN = 28
    BIKES = 29
    SNOW = 30
    ANIMALS = 31
    EOS = 32
    TURN_RIGHT = 33
    TURN_LEFT = 34
    FORWARD = 35
    FORWARD_OR_TURN_RIGHT = 36
```

```
    FORWARD_OR_TURN_LEFT = 37
    DRIVE_ON_RIGHT = 38
    DRIVE_ON_LEFT = 39
    ROUNDABOUT = 40
    EONOC = 41
    EONOT = 42
```

Poniższy blok kodu służy do wybrania losowego obrazka ze zbioru i wyświetleniu go.

[26]:
```
# Press CTRL+ENTER multiple times to see some random images
im, target = random.choice(pure_dataset)
label = TrafficSign(target).name
print(f'Label: {label}')
im
```

```
Label: CONSTRUCTION_ZONE
```

[26]:

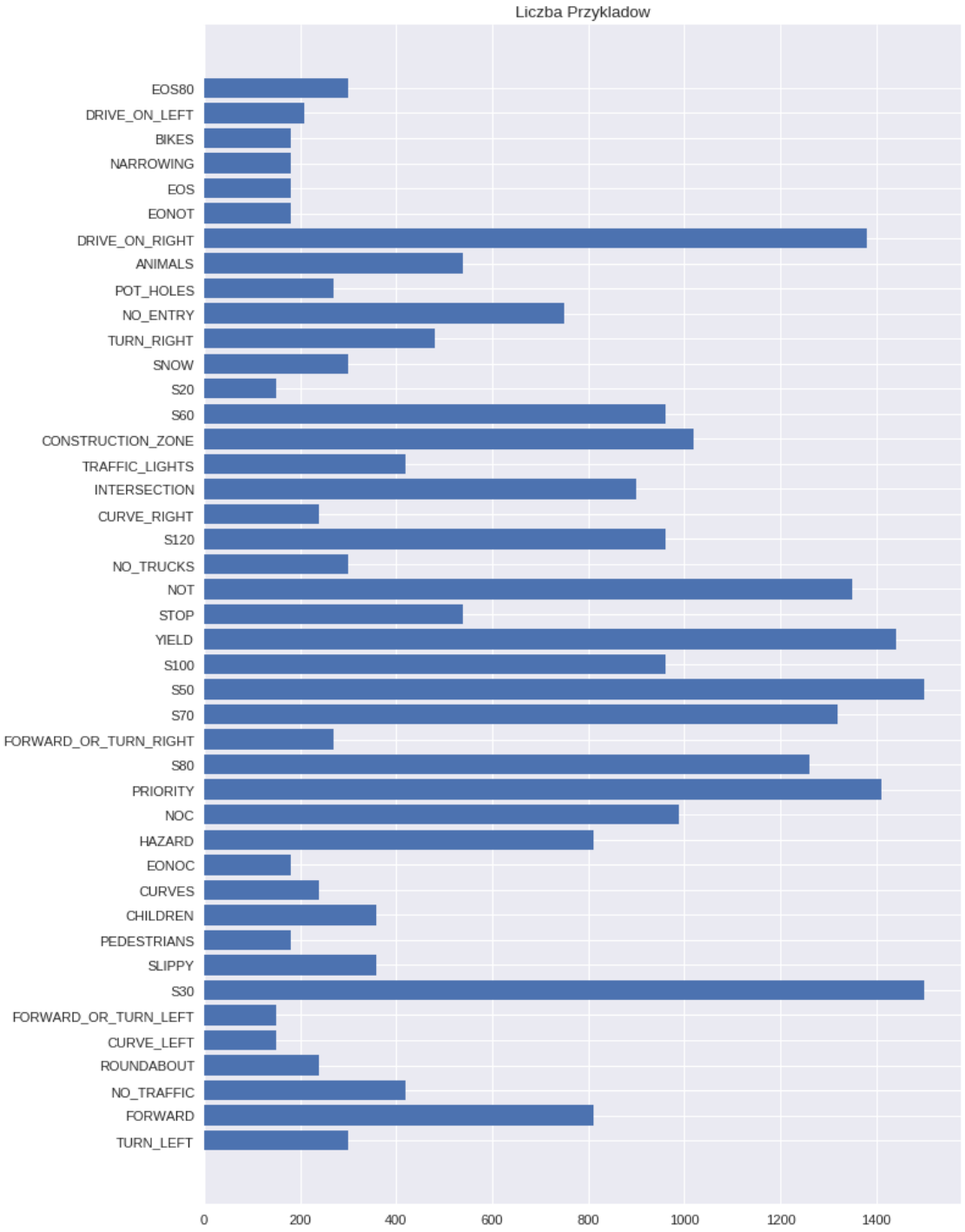
Jak łatwo zauważyć, obrazy różnią się od siebie nie tylko przedstawianym znakiem, ale również oświetleniem, ostrością czy wielkością.

Sprawozdanie (1) Policz liczebność każdej z klas i narysuj histogram przedstawiający tę statystykę.

Przydatne linki: * https://docs.scipy.org/doc/numpy/reference/generated/numpy.histogram.html

[27]:
```
# PUT CODE HERE
%matplotlib inline
import matplotlib.pyplot as plt
plt.style.use('seaborn')
def df_hist(dataset):
    Targets=[]
    for im,target in dataset:
        label = TrafficSign(target).name
        Targets.append(label)
    df=pd.DataFrame(columns=list(set(Targets)))
    dict_targets={key:[] for key in list(set(Targets))}

    for im,target in dataset:
        dict_targets[TrafficSign(target).name].append(target)

    for key,value in dict_targets.items():
        dict_targets[f"{key}"]=len(value)
```

```
    df=pd.DataFrame(dict_targets,index=[0],columns=dict_targets.keys())
    df=df.melt(value_vars=list(df.
 ↪columns),var_name='Target',value_name='Examples')
    return df

df=df_hist(pure_dataset)
### Liczba Przukladow

fig = plt.figure(num=None, figsize=(10, 16), dpi=80, facecolor='w',␣
 ↪edgecolor='k');
ax= plt.barh(y=df['Target'],width=df['Examples'],height=0.8);
plt.title("Liczba Przykladow");
```



2.6 Przygotowanie danych do treningu

Jak zauważyliśmy wyżej, obrazki różnią się od siebie chociażby rozmiarem. Nasza sieć wymaga jednak, aby obrazki były kwadratem o rozmiarze

NUM_CHANNELS x IMAGE_WIDTH x IMAGE_HEIGHT. W tym celu skorzystamy z modułu torchvision.transforms i stworzymy kompozycję transformacji, która każdy *surowy* obrazek w naszym zbiorze danych przekształci do wymaganego rozmiaru, a następnie stworzy z niego torch.Tensor.

[28]:
```
IMAGE_WIDTH, IMAGE_HEIGHT = 32, 32

resize_then_tensorify = torchvision.transforms.Compose([
    torchvision.transforms.Resize((IMAGE_WIDTH, IMAGE_HEIGHT)),
    torchvision.transforms.ToTensor()
])
dataset = torchvision.datasets.ImageFolder(r'./GTSRB/train',␣
 ↪transform=resize_then_tensorify)
```

Poniższy kod jest bardzo podobny do tego wykorzystanego do wizualizacji datasetu. Tym razem operuje on jednak na zbiorze obiektów torch.Tensor. W ten sposób można zobaczyć, jak wygląda tensor wejściowy do naszej sieci - w postaci numerycznej.

[29]:
```
im_t, target = random.choice(dataset)
label = TrafficSign(target).name
print(f'Label: {label}')
im_t, im_t.shape
```

```
Label: HAZARD
```

[29]:
```
(tensor([[[0.1294, 0.1137, 0.2000,  …, 0.1020, 0.1922, 0.1843],
          [0.1725, 0.1529, 0.2118,  …, 0.1490, 0.2510, 0.1765],
          [0.1922, 0.1804, 0.1686,  …, 0.1373, 0.2000, 0.1922],
          …,
          [0.0431, 0.0392, 0.0431,  …, 0.2000, 0.1922, 0.1882],
          [0.0431, 0.0431, 0.0471,  …, 0.1294, 0.1255, 0.1216],
          [0.0627, 0.0549, 0.0549,  …, 0.0549, 0.0588, 0.0549]],

         [[0.1569, 0.1216, 0.1686,  …, 0.0824, 0.1608, 0.1765],
          [0.1882, 0.1569, 0.1961,  …, 0.0902, 0.2039, 0.1922],
          [0.1922, 0.1882, 0.1686,  …, 0.1098, 0.1804, 0.1961],
          …,
          [0.0431, 0.0431, 0.0431,  …, 0.2235, 0.2118, 0.2039],
          [0.0471, 0.0431, 0.0471,  …, 0.1412, 0.1373, 0.1294],
          [0.0667, 0.0588, 0.0627,  …, 0.0510, 0.0549, 0.0549]],

         [[0.2000, 0.1490, 0.1843,  …, 0.0980, 0.1961, 0.1961],
          [0.2510, 0.2078, 0.2000,  …, 0.0902, 0.2314, 0.2118],
          [0.2275, 0.2314, 0.1843,  …, 0.1137, 0.1961, 0.2157],
          …,
          [0.0588, 0.0588, 0.0588,  …, 0.2471, 0.2353, 0.2353],
          [0.0667, 0.0627, 0.0627,  …, 0.1569, 0.1490, 0.1529],
          [0.0941, 0.0824, 0.0824,  …, 0.0667, 0.0706, 0.0706]]]),
```

```
 torch.Size([3, 32, 32]))
```

2.7 Wydzielenie zbioru walidacyjnego

Ucząc nasz klasyfikator chcielibyśmy na bieżąco wiedzieć, jak dobrze radzi sobie nie tylko z obrazkami, które znajdują się w jego zbiorze treningowym, ale także jak rozpoznaje obrazy, których nigdy wcześniej nie widział. W tym celu ze zbioru treningowego wydzielamy podzbiór, który nie wykorzystamy do uczenia, a jedynie do walidacji.

[30]:
```
from math import ceil, floor
from torch.utils.data import random_split

num_samples = len(dataset)
ratio = 0.8
num_train_samples = floor(ratio * num_samples)
num_val_samples = ceil((1.0 - ratio) * num_samples)
train_dataset, val_dataset = random_split(dataset, [num_train_samples,␣
 ↪num_val_samples])
print(f'Training dataset size: {len(train_dataset)}, validation dataset size:␣
 ↪{len(val_dataset)}')
```

```
Training dataset size: 21312, validation dataset size: 5328
```

2.8 Architektura sieci

Architektura naszej sieci składa się z: * warstwy wejściowej (conv1) - konwolucyjnej, ilość filtrów = 16, rozmiar filtru = 3x3 * warstwy ukrytej (conv2) - konwolucyjnej, ilość filtrów = 32, rozmiar filtru = 3x3 * warstwy ukrytej (fc1) - w pełni połączonej, która jako wejście przyjmuje wektor reprezentujący spłaszczone wyjście z poprzedniej warstwy, ilość neuronów = 512 * warstwy ukrytej (fc2) - w pełni połączonej, ilość neuronów = 128 * warstwy wyjściowej (fc3) - w pełni połączonej, ilość neuronów = ilość klas = 43

W celu redukcji wymiarowości, a co za tym idzie zmniejszenia wymagań dotyczących mocy obliczeniowej i pamięci, po każdej warstwie konwolucyjnej zastosowano *max pooling* o oknie 2x2.

Funkcja aktywacji jest wszędzie taka sama - ReLU.

Przydatne linki: * https://pytorch.org/docs/stable/nn.html * https://pytorch.org/docs/stable/nn.html#module * https://pytorch.org/docs/stable/nn.html#conv2d * https://pytorch.org/docs/stable/nn.html#linear

[33]:
```
import torch.nn as nn
import torch.nn.functional as F
from torch.autograd import Variable
```

```
class TrafficSignClassifierNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 16, 3)
        self.conv2 = nn.Conv2d(16, 32, 3)

        # TODO Switch comment on these two lines this after completing the task␣
↪below
        self.fc1 = nn.Linear(6 * 6 * 32, 512)
        #self.fc1 = nn.Linear(self.num_final_conv_features(IMAGE_WIDTH,␣
↪IMAGE_HEIGHT), 512)

        self.fc2 = nn.Linear(512, 128)
        self.fc3 = nn.Linear(128, 43)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        x = F.max_pool2d(F.relu(self.conv2(x)), (2, 2))
        x = x.view(x.size(0), -1)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

    def num_final_conv_features(self, width: int, height: int) -> int:
        #random saples of the given size
        inputs = torch.rand(*(1,3,width,height))
        # tensor after first convergence
        print(inputs.shape)
        out1 = self.conv1(inputs)
        print('tensor after first convergence:')
        print(out1.shape)

        out2=self.conv2(out1)
        print('tensor after second convergence:')
        print(out2.shape)
        total_nr_features=(3*2*3*2)*32
        return total_nr_features
```

Sprawozdanie (2) Zaimplementuj funkcję TrafficSignClassifierNet::num_final_conv_features(width height), która przyjmując szerokość i wysokość obrazka na wejściu, i znając konfigurację warstw konwolucyjnych w sieci (self.conv1, self.conv2), zwraca ilość feature-ów na wyjściu ostatniej warstwy konwolucyjnej (conv2) (po spłaszczeniu tego wyjścia do wektora). Zaprezentuj obliczenia.

Przydatne linki: * https://pytorch.org/docs/stable/nn.html#conv2d

[34]:
```
tc=TrafficSignClassifierNet()
print("Całkowita liczba feature'ów:",tc.num_final_conv_features(IMAGE_WIDTH,␣
 ↪IMAGE_HEIGHT))
```

```
torch.Size([1, 3, 32, 32])
tensor after first convergence:
torch.Size([1, 16, 30, 30])
tensor after second convergence:
torch.Size([1, 32, 28, 28])
Całkowita liczba feature'ów: 1152
```

2.9 Trening

Mając już zdefiniowaną architekturę sieci oraz przygotowany zbiór treningowy możemy przystąpić do uczenia sieci. W pierwszej kolejności tworzymy jej instancję.

[35]:
```
net = TrafficSignClassifierNet()
net
```

[35]:
```
TrafficSignClassifierNet(
  (conv1): Conv2d(3, 16, kernel_size=(3, 3), stride=(1, 1))
  (conv2): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1))
  (fc1): Linear(in_features=1152, out_features=512, bias=True)
  (fc2): Linear(in_features=512, out_features=128, bias=True)
  (fc3): Linear(in_features=128, out_features=43, bias=True)
)
```

Następnie definiujemy funkcję kosztu, którą wykorzystamy w procesie uczenia. Użyjemy do tego celu klasy CrossEntropyLoss, która implementuje entropię krzyżową.

Przydatne linki: * https://pytorch.org/docs/stable/nn.html#crossentropyloss * https://ml-cheatsheet.readthedocs.io/en/latest/loss_functions.html#cross-entropy

[36]:
```
criterion = nn.CrossEntropyLoss()
```

Kolejnym krokiem jest definicja algorytmu optymalizującego naszą sieć. W tym przypadku wykorzystamy algorytm Adam.

Przydatne linki: * https://arxiv.org/abs/1412.6980 * https://pytorch.org/docs/stable/optim.htm

[37]:
```
import torch.optim as optim

net_optim = optim.Adam(net.parameters())
```

Ostatnim elementem do zdefiniowania jest pomocnicza funkcja do generowania *minibatch*-y. Za pomocą stałej BATCH_SIZE można modyfikować wielkość *minibatch*-a.

[38]:
```
from torch.utils.data import BatchSampler, DataLoader, Dataset, RandomSampler

def make_data(dataset: Dataset, batch_size: int) -> DataLoader:
    sampler = BatchSampler(RandomSampler(dataset), batch_size=batch_size,␣
 ↪drop_last=False)
    data_loader = DataLoader(dataset, batch_sampler=sampler)
    return data_loader

BATCH_SIZE = 256
train_data = make_data(train_dataset, BATCH_SIZE)
val_data = make_data(val_dataset, BATCH_SIZE)
```

Przygotujemy również funkcję evaluate(net, data), która policzy metrykę *accuracy* sieci net na zbiorze danych data. Metryka ta mówi nam o tym, jaki stosunek wszystkich klasyfikacji stanowią klasyfikacje poprawne. Istnieje szereg różnych innych metryk pozwalających na szerszą ewaluację klasyfikatorów. Pojawią się one w następnym laboratorium. Tutaj skupimy się jedynie na tej metryce.

[39]:
```
def evaluate(net: nn.Module, data: DataLoader):
    targets = []
    predictions = []
    for input_b, target_b in data:
        out = net(input_b)
        targets.append(target_b)
        predictions.append(out.argmax(dim=1))
    targets_b = torch.cat(tuple(targets), dim=0)
    predicted_b = torch.cat(tuple(predictions), dim=0)
    cmp = torch.eq(targets_b, predicted_b).to(torch.float32)
    return cmp.mean()
```

Mając tak zdefiniowaną funkcję możemy policzyć *accuracy* na zbiorze treningowym i walidacyjnym dla klasyfikatora, który nie został jeszcze wytrenowany. Losowo zainicjalizowane wagi sieci neuronowej raczej nie pozwolą na jakąkolwiek sensowną klasyfikację, stąd wartość metryki powinien być bliski zeru.

[40]:
```
train_acc, val_acc = evaluate(net, train_data), evaluate(net, val_data)
print(f'Accuracy on training set: {train_acc:.2f}')
print(f'Accuracy on validation set: {val_acc:.2f}')
```

```
Accuracy on training set: 0.02
Accuracy on validation set: 0.02
```

Mając zdefiniowane wszystkie niezbędne moduły przystępujemy do trenowania naszej sieci. Proces uczenia składa się z NUM_EPOCHS epok. W każdej epoce iterujemy po całym zbiorze treningowym, a dokładnie *minibatch*-ach pokrywających ten zbiór.

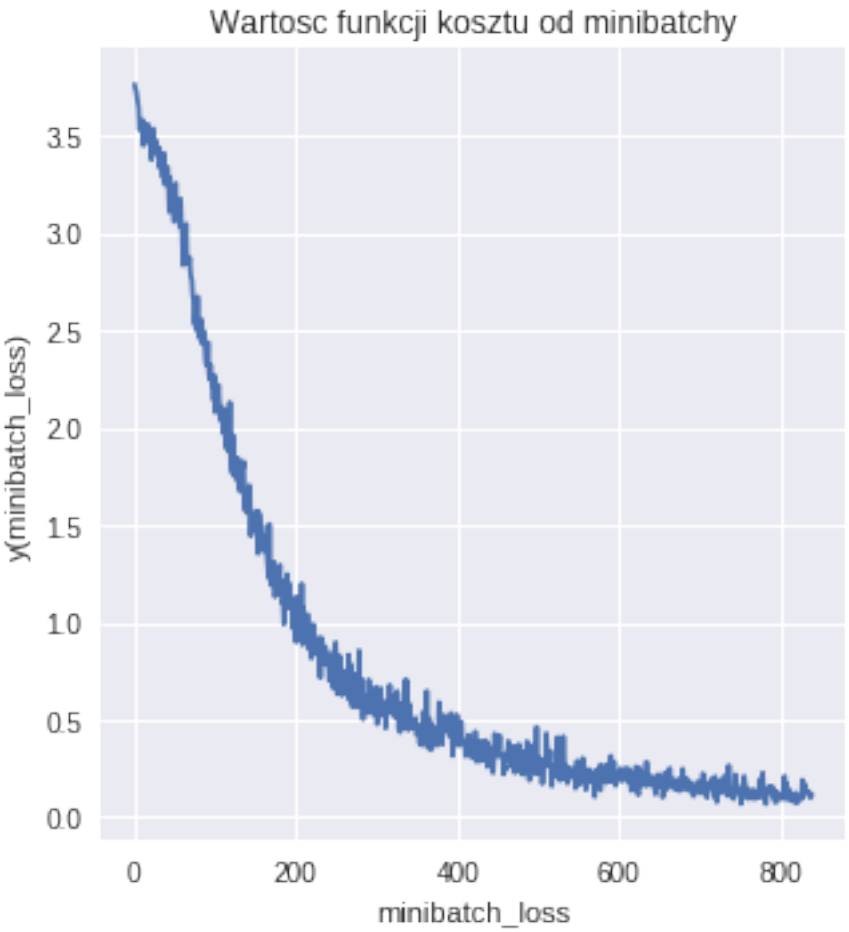
[41]:
```
NUM_EPOCHS = 10
y1=[]
```

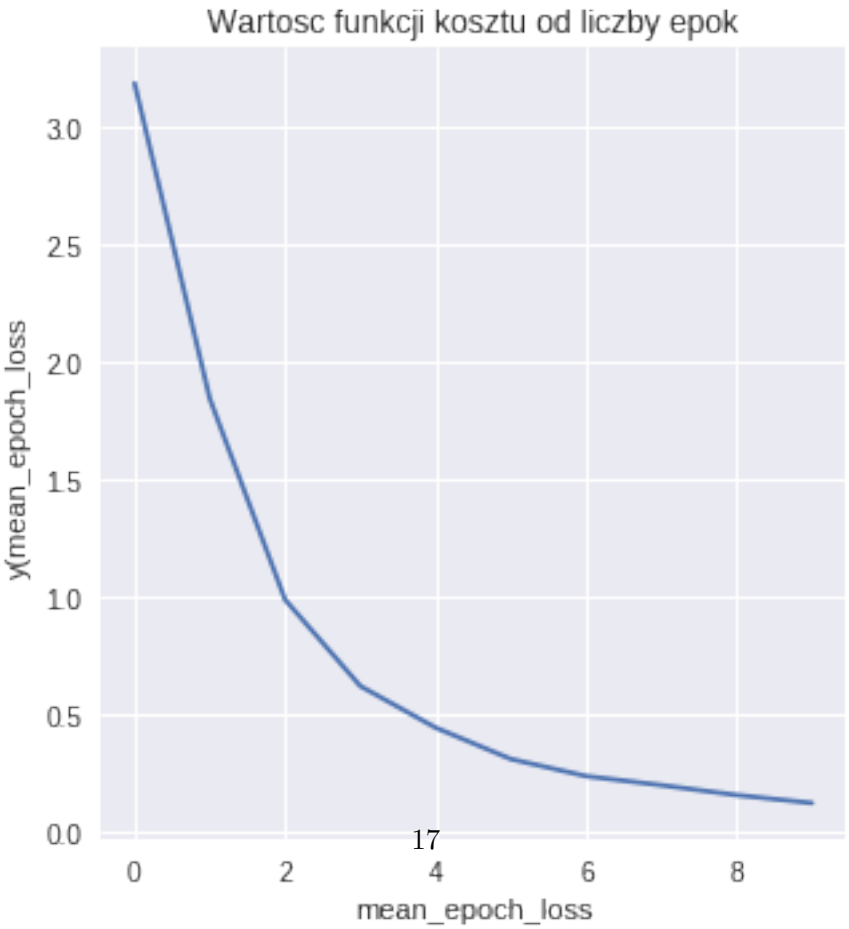
```
y2=[]
Accuracy=pd.DataFrame(columns=["Train","Validation"])
for epoch in range(NUM_EPOCHS):
    # Optimization
    losses = []
    for input_b, target_b in train_data:
        out = net(input_b)  # Pass minibatch of images through the network
        minibatch_loss = criterion(out, target_b)  # Calculate loss value
        net_optim.zero_grad()  # Zero gradients buffers in network parameters
        minibatch_loss.backward()  # Run backward pass
        net_optim.step()  # Apply weights updates
        losses.append(float(minibatch_loss))
        y1.append(float(minibatch_loss))
    mean_epoch_loss = float(torch.tensor(losses).mean())
    y2.append(mean_epoch_loss)
    print(f'Epoch: {epoch} | Mean epoch loss: {mean_epoch_loss:.4f}')

    # Accuracy
    train_acc, val_acc = evaluate(net, train_data), evaluate(net, val_data)
    Accuracy.loc[epoch]=[train_acc,val_acc]
    print(f'Epoch: {epoch} | Training set accuracy: {train_acc:.2f} |␣
 ↪Validation set accuracy: {val_acc:.2f}')
```

```
Epoch: 0 | Mean epoch loss: 3.1867
Epoch: 0 | Training set accuracy: 0.28 | Validation set accuracy: 0.28
Epoch: 1 | Mean epoch loss: 1.8424
Epoch: 1 | Training set accuracy: 0.60 | Validation set accuracy: 0.60
Epoch: 2 | Mean epoch loss: 0.9864
Epoch: 2 | Training set accuracy: 0.78 | Validation set accuracy: 0.78
Epoch: 3 | Mean epoch loss: 0.6181
Epoch: 3 | Training set accuracy: 0.86 | Validation set accuracy: 0.86
Epoch: 4 | Mean epoch loss: 0.4409
Epoch: 4 | Training set accuracy: 0.90 | Validation set accuracy: 0.89
Epoch: 5 | Mean epoch loss: 0.3072
Epoch: 5 | Training set accuracy: 0.93 | Validation set accuracy: 0.92
Epoch: 6 | Mean epoch loss: 0.2343
Epoch: 6 | Training set accuracy: 0.94 | Validation set accuracy: 0.93
Epoch: 7 | Mean epoch loss: 0.1957
Epoch: 7 | Training set accuracy: 0.96 | Validation set accuracy: 0.95
Epoch: 8 | Mean epoch loss: 0.1536
Epoch: 8 | Training set accuracy: 0.97 | Validation set accuracy: 0.96
Epoch: 9 | Mean epoch loss: 0.1199
Epoch: 9 | Training set accuracy: 0.98 | Validation set accuracy: 0.97
```

Sprawozdanie (3) Sporządź wykres przebiegu wartości funkcji kosztu od liczby przeprocesowanych minibatch-y (minibatch_loss) oraz epok (mean_epoch_loss) - dwa wykresy. Porównaj je ze sobą i napisz skąd wynika taka różnica. Przyjmij liczbę epok nie mniejszą niż 10.

[42]:
```
# PUT CODE HERE
fig,(ax1,ax2) =plt.subplots(2,1,figsize=(5,12))
ax1.plot(np.array(y1));
ax1.set(title="Wartosc funkcji kosztu od minibatchy",
       ylabel="y(minibatch_loss)",
       xlabel="minibatch_loss");
ax2.plot(np.array(y2));
ax2.set(title="Wartosc funkcji kosztu od liczby␣
 ↪epok",ylabel="y(mean_epoch_loss",xlabel="mean_epoch_loss");
```


Wartosc funkcji kosztu od minibatchy
y(minibatch_loss)
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0
200
400
600
800
minibatch_loss


Wartosc funkcji kosztu od liczby epok
y(mean_epoch_loss
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0
2
4
6
8
mean_epoch_loss

Sprawozdanie (4) Sporządź wykres przebiegu wartości metryki precyzji na zbiorze treningowym oraz walidacyjnym od liczby epok. Przyjmij liczbę epok nie mniejszą niż 10.

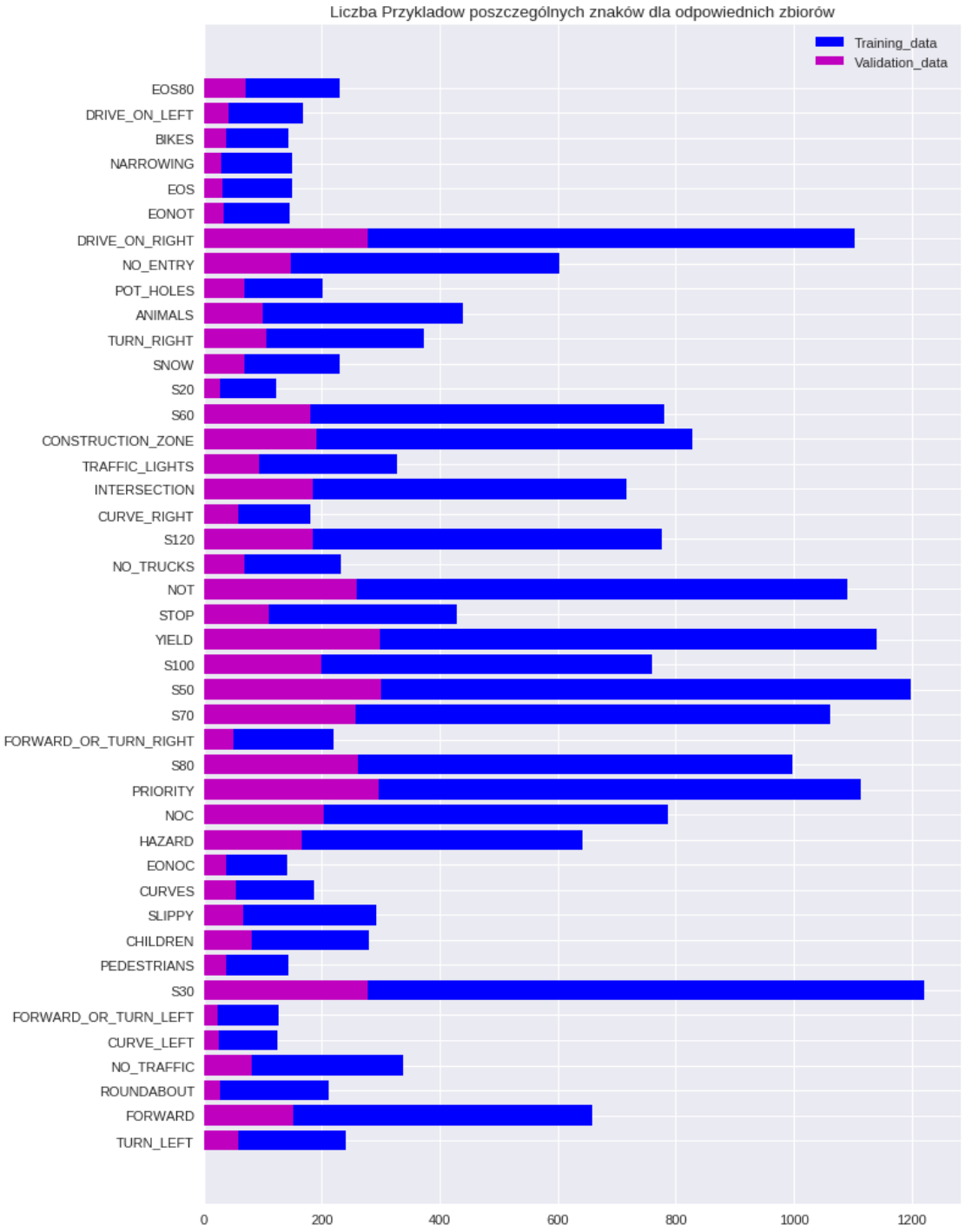
[43]:
```
# PUT CODE HERE
plt.plot(np.array(Accuracy["Train"]));
plt.plot(np.array(Accuracy["Validation"]),color='r');
plt.title("Dokladnosc w liczbie epok");
plt.legend(["Zbior treningowy","Zbior walidacyjny"]);
```

Sprawozdanie (5) Obserwując wykresy z poprzedniego zadania (4) czy zauważasz jakąś różnicę między *accuracy* na zbiorze treningowym, a walidacyjnym? Jak myślisz, dlaczego tak jest? Pomocne może okazać się przeglądnięcie katalogu ze zbiorem treningowym, z którego wydzieliliśmy zbiór walidacyjny, a także wyświetlenie pojedynczych obrazków ze zbioru treningowego i walidacyjnego.

Zarówno dokładność na zbiorze treningowym, jak i walidacyjnym rośnie wraz z liczbą wywołań. możemy wnioskować, że model jest zaprojektowany dobrze. Zbiór walidacyjny jest 4 razy mniejszy od zbioru teningowego, więc dokładność musi być mniejsza. Analogicznie zawyżona może być dokładność zbioru treningowego.

2.9.1 Sprawdzamy rozkład przykładów dla poszczególnych zbiorów

[44]:
```
train_df=df_hist(train_dataset)
val_df=df_hist(val_dataset)

fig = plt.figure(num=None, figsize=(10,16), dpi=80, facecolor='w',␣
 ↪edgecolor='k');
plt.barh(y=train_df['Target'],width=train_df['Examples'],height=0.8,color='b');
plt.barh(y=val_df['Target'],width=val_df['Examples'],height=0.8,color='m');
plt.legend(["Training_data","Validation_data"]);
plt.title("Liczba Przykladow poszczególnych znaków dla odpowiednich zbiorów");
```



Sprawozdanie (6) Na bazie poczynionych w poprzednim zadaniu obserwacji wysuń wnioski, czy taki podział na zbiór treningowy i walidacyjny jest poprawny? Jeśli nie, to w jaki sposób można byłoby go poprawić?

Obserwując podział na wykresie możemy zauważyć, że przykłady sa podzielone

proporcjonalnie pomiędzy odpowiednie zbiory. Co zdaje się dawać dobre efekty, o czym świadczy dokładność modelu dla obu zbiorów.

Warto zauważyć, że dla obu zbiorów dokładność jest bardzo dobra. Jeżeli dokładności dla zbioru walidacyjnego i treningowego nie odbiegają od siebie znacznie, i nie ma między nimi dużych wahań w czasie wywołań, to dobry prognostyk. Oznacza to, że model który wytrenujemy nie jest przesadzony, nie uwypukla niepożądanych cech obiektu przy klasyfikacji.

Jeżeli dokładność dla zbioru walidacyjnego byłaby niska, przy dobrych wynikach przykładów treningowych, to możnaby mieć watpliwości co do zachowania przez model ogólności i podejrzewać zbyt duży wpływ niektórych cech przykładów na ogólną klasyfikację.

2.10 Ewaluacja wyników

Ostatnim krokiem tego laboratorium, a zaraz krokiem rozpoczynającym następne laboratorium, jest ewaluacja wytrenowanego klasyfikatora na zbiorze testowym. Wykorzystamy do tego zdefiniowaną wcześniej funkcję evaluate(net, data).

[45]:
```
def make_random_iterator_on_data(dataset: Dataset) -> DataLoader:
    sampler = RandomSampler(dataset)
    data_loader = DataLoader(dataset, sampler=sampler)
    return data_loader
```

[46]:
```
test_dataset = torchvision.datasets.ImageFolder(r'./GTSRB/test',␣
 ↪transform=resize_then_tensorify)
test_data = make_data(test_dataset, BATCH_SIZE)
```

[47]:
```
test_acc = evaluate(net, test_data)
print(f'Accuracy on test set: {test_acc:.2f}')
```

```
Accuracy on test set: 0.83
```

Zwizualizujmy też sobie działanie tak wytrenowanego klasyfikatora.

[48]:
```
tensor_to_pil = torchvision.transforms.ToPILImage()
test_data_random_instance_iterator = make_random_iterator_on_data(test_dataset)

def visualize_target_prediction(net: nn.Module, data: DataLoader):
    im, target = next(iter(data))
    out = net(im).argmax(dim=1)
    return tensor_to_pil(im[0]), TrafficSign(int(target[0])).name,␣
 ↪TrafficSign(int(out[0])).name
```

[49]:
```
im, target, prediction = visualize_target_prediction(net,␣
 ↪test_data_random_instance_iterator)
print(f'Target: {target} | Prediction: {prediction}')
```

```
im
```

```
Target: S80 | Prediction: S80
```

[49]:

Jak widać, udało nam się wytrenować klasyfikator znaków drogowych z całkiem zadowalającą dokładnością na zbiorze testowym. Na następnym laboratorium przyjrzymy się sposobom na szerszą ewaluację takiego klasyfikatora oraz spróbujemy uczynić go jeszcze lepszym.