lab_4

January 14, 2023

1 Konwolucyjne sieci neuronowe

Dziś spróbujemy stworzyć i wytrenować prostą sieć konwolucyjną do rozpoznawania, co znajduje się na obrazie. Następnie omówimy kwestię identyfikowania obiektów na obrazie, oraz porozmawiamy o wykorzystaniu gotowej już sieci.

1.1 Problem klasyfikacji obrazów

Jak się za to zabrać? Naiwnym podejściem byłaby próba ręcznej specyfikacji pewnych cech (niemowlęta mają duże głowy, szczoteczki są długie, etc.). Szybko jednak stwierdziliśmy, że nawet dla niewielkiego zbioru kategorii jest to tytaniczna praca bez gwarancji sukcesu. Co więcej, istnieje wiele czynników zniekształcających zawartość naszych zdjęć. Obiekty mogą być przedstawiane z różnych ujęć, w różnych warunkach oświetleniowych, w różnej skali, częściowo niewidoczne, ukryte w tle…

Wszystkie wymienione problemy są skutkiem istnienia semantycznej przepaści między tym, jak reprezentowane są nasze dane wejściowe (tablica liczb), a tym, czego w nich szukamy, czyli kategorii i cech: zwierząt, nosów, głów, itp. Zamiast więc próbować samodzielnie napisać funkcję $f(x)$, spróbujemy skorzystać z dobrodziejstw uczenia maszynowego, aby automatycznie skonstruować reprezentację wejścia właściwą dla postawionego sobie zadania (a przynajmniej lepszą od pierwotnej). I tu z pomocą przychodzą nam konwolucyjne sieci neuronowe. Do tego trzeba zrozumieć, czym jest konwolucja (inaczej: splot), a do tego najlepiej nadają się ilustracje, jak to działa.

1.1.1 Konwolucja

Konwolucja (splot) to działanie określone dla dwóch funkcji, dające w wyniku inną, która może być postrzegana jako zmodyfikowana wersja oryginalnych funkcji.

Z naszego punktu widzenia polega to na tym, że mnożymy odpowiadające sobie elementy z dwóch macierzy: obrazu, oraz mniejszej, nazywanej filtrem (lub kernelem). Następnie sumujemy wynik i zapisujemy do macierzy wynikowej na odpowiedniej pozycji. Proces powtarza się aż do momentu przeskanowania całego obrazu. Taki filtr wykrywa, czy coś do niego pasuje w danym miejscu, i z tego wynika zdolność semantycznej generalizacji sieci - uczymy się cech, a wykrywamy je potem w dowolnym miejscu. Przydatne pojęcia

1.1.2 Stride

Krok algorytmu, albo przesunięcie.

1.1.3 Padding

Dopełnienie krawędzi obrazu. więcej

1.1.4 Pooling

Ma 2 warianty: max oraz avg. Pozwala on usunąć zbędne dane, np. jeżeli filtr wykrywa linie, to istnieje spora szansa, że linie te ciągną się przez sąsiednie piksele, więc nie ma powodu powielać tej informacji. Dzięki temu mamy pewną ilość inwariancji i jesteśmy odporni na niewielkie wahania pikseli, a skupiamy się na "bigger picture".

1.1.5 Sposoby redukcji przeuczenia

- warstwa dropout
- regularyzacja wag
- metoda wczesnego stopu (early stopping)
- batch normalization
- lub... więcej danych

1.1.6 Budowa sieci CNN do klasyfikacji obrazów

Sieć konwolucyjna składa się zawsze najpierw, zgodnie z nazwą, z części konwolucyjnej, której zadaniem jest wyodrębnienie przydatnych cech z obrazu za pomocą filtrów, warstw poolingowych etc.

W celu klasyfikacji obrazu musimy później użyć sieci MLP. Jako że wejściem do sieci MLP jest zawsze wektor, to musimy obraz przetworzony przez filtry konwolucyjne sprowadzić do takiego wektora, tzw. **embedding**, czyli reprezentacji obrazu jako punktu w pewnej ciągłej przestrzeni. Służą do tego warstwa spłaszczająca (flatten layer), zmieniająca macierze wielkowymiarowe na wektor, np $10 \times 10 \times 3$ na 300×1.

Część konwolucyjna nazywa się często **backbone**, a część MLP do klasyfikacji **head**. Głowa ma zwykle 1-2 warstwy w pełni połączone, z aktywacją softmax w ostatniej warstwie. Czasem jest nawet po prostu pojedynczą warstwą z softmaxem, bo w dużych sieciach konwolucyjnych ekstrakcja cech jest tak dobra, że taka prosta konstrukcja wystacza do klasyfikacji embeddingu.

[ ]:
```
import torch
import torchvision
import torchvision.transforms as transforms
```

[ ]:
```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print(device)
```

```
cuda:0
```

[ ]:
```
transform = transforms.Compose([transforms.ToTensor()])

batch_size = 32

trainset = torchvision.datasets.FashionMNIST(
    root="./data", train=True, download=True, transform=transform
)
trainloader = torch.utils.data.DataLoader(
    trainset, batch_size=batch_size, shuffle=True
)

testset = torchvision.datasets.FashionMNIST(
    root="./data", train=False, download=True, transform=transform
)
testloader = torch.utils.data.DataLoader(
    testset, batch_size=batch_size, shuffle=True
)

classes = (
    "top",
    "Trouser",
    "Pullover",
    "Dress",
    "Coat",
    "Sandal",
    "Shirt",
    "Sneaker",
    "Bag",
    "Ankle boot",
)
```

Zobaczmy, co jest w naszym zbiorze danych. Poniżej kawałek kodu, który wyświetli nam kilka przykładowych obrazków.

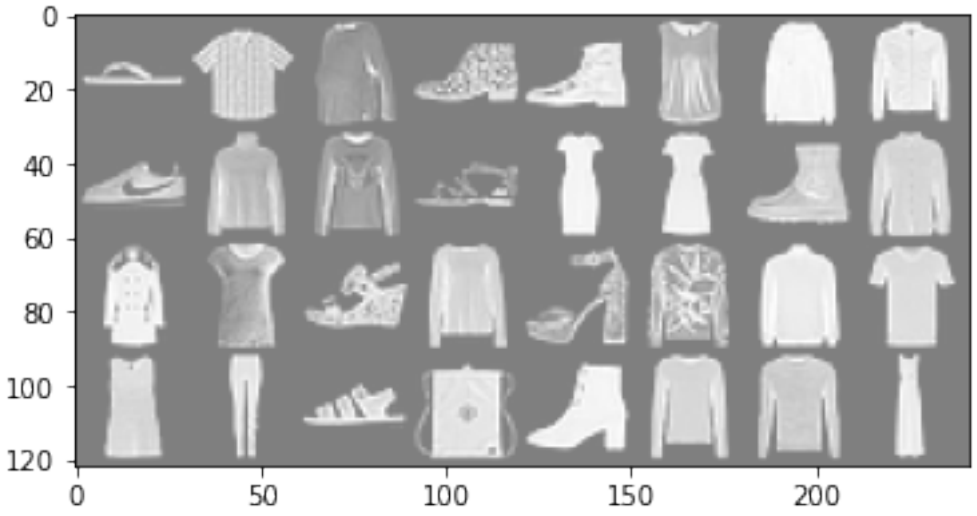
[ ]:
```
import matplotlib.pyplot as plt
import numpy as np


def imshow(img):
    img = img / 2 + 0.5
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.show()
```

```
dataiter = iter(trainloader)
images, labels = next(dataiter)

imshow(torchvision.utils.make_grid(images))

print(' '.join(f'{classes[labels[j]]:5s}' for j in range(batch_size)))
```




```
Sandal Shirt Pullover Ankle boot Ankle boot Shirt Pullover Coat  Sneaker
Pullover Pullover Sandal Dress Dress Ankle boot Shirt Coat  top   Sandal Shirt
Sandal Pullover Coat  top   top   Trouser Sandal Bag   Ankle boot Pullover
Pullover Dress
```

1.2 LeNet

LeNet to bardzo znany, klasyczny model sieci konwolucyjnej.

Warstwy: - obraz - konwolucja, kernel 5×5, bez paddingu, 6 kanałów (feature maps) - pooling, kernel 2×2, stride 2 - konwolucja, kernel 5×5, bez paddingu, 16 kanałów (feature maps) - pooling, kernel 2×2, stride 2 - warstwa w pełni połączona, 120 neuronów na wyjściu - warstwa w pełni połączona, 84 neurony na wyjściu - warstwa w pełni połączona, na wyjściu tyle neuronów, ile jest klas

1.2.1 Zadanie 1 (2 punkty)

Zaimplementuj wyżej opisaną sieć, używając biblioteki PyTorch. Wprowadzimy sobie jednak pewne modyfikacje, żeby było ciekawiej: - w pierwszej warstwie konwolucyjnej użyj 20 kanałów (feature maps) - w drugiej warstwie konwolucyjnej użyj 50 kanałów (feature maps) - w pierwszej warstwie gęstej użyj 300 neuronów - w drugiej warstwie gęstej użyj 100 neuronów

Przydatne elementy z pakietu `torch.nn`: * `Conv2d()` * `AvgPool2d()` * `Linear()`

Z pakietu `torch.nn.functional`: * `relu()`

[ ]:
```
import torch.nn as nn

class LeNet(nn.Module):
    def __init__(self, data_size, classes_count):
        super().__init__()
        self.backbone = nn.Sequential(
            nn.Conv2d(1, 20, kernel_size=5),
            nn.AvgPool2d(kernel_size=2, stride=2),
            nn.Conv2d(20, 50, kernel_size=5),
            nn.AvgPool2d(kernel_size=2, stride=2)
        )
        self.head = nn.Sequential(
            nn.Linear(data_size, 300),
            nn.ReLU(),
            nn.Linear(300, 100),
            nn.ReLU(),
            nn.Linear(100, classes_count)
        )

    def forward(self, x):
        return self.head(torch.flatten(self.backbone(x), 1))
```

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

def train_model_on_device(device):
  net = LeNet(800, len(classes)).to(device)

  criterion = nn.CrossEntropyLoss()
  optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)

  net.train()
  net.to(device)

  start_time = time.time()

  for epoch in range(5):

      running_loss = 0.0
      for i, data in enumerate(trainloader, 0):
          inputs, labels = data
          inputs = inputs.to(device)
          labels = labels.to(device)

          outputs = net(inputs)
```

```
            loss = criterion(outputs, labels)

            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            running_loss += loss.item()
            if i % 200 == 199:
                print(f"[{epoch + 1}, {i + 1:5d}] loss: {running_loss / 200:.3f}")
                running_loss = 0.0

    end_time = time.time()
    print(f"Training finished in {end_time - start_time:.2f} s on {device}")
    return net
```

1.2.2 Zadanie 2 (1 punkt)

Uzupełnij pętlę uczącą sieć na podstawie jej predykcji. Oblicz (wykonaj krok do przodu) funkcję straty, a następnie przeprowadź propagację wsteczną i wykonaj krok optymalizatora, porównaj czas uczenia na cpu i gpu.

[ ]:
```
net_cpu = train_model_on_device("cpu")
```

```
[1,   200] loss: 2.274
[1,   400] loss: 2.024
[1,   600] loss: 1.163
[1,   800] loss: 0.951
[1,  1000] loss: 0.842
[1,  1200] loss: 0.780
[1,  1400] loss: 0.790
[1,  1600] loss: 0.747
[1,  1800] loss: 0.746
[2,   200] loss: 0.691
[2,   400] loss: 0.707
[2,   600] loss: 0.691
[2,   800] loss: 0.666
[2,  1000] loss: 0.648
[2,  1200] loss: 0.628
[2,  1400] loss: 0.618
[2,  1600] loss: 0.611
[2,  1800] loss: 0.591
[3,   200] loss: 0.578
[3,   400] loss: 0.589
[3,   600] loss: 0.562
[3,   800] loss: 0.571
[3,  1000] loss: 0.564
[3,  1200] loss: 0.559
[3,  1400] loss: 0.540
```

```
[3,  1600] loss: 0.526
[3,  1800] loss: 0.526
[4,   200] loss: 0.497
[4,   400] loss: 0.537
[4,   600] loss: 0.514
[4,   800] loss: 0.499
[4,  1000] loss: 0.506
[4,  1200] loss: 0.507
[4,  1400] loss: 0.493
[4,  1600] loss: 0.477
[4,  1800] loss: 0.476
[5,   200] loss: 0.479
[5,   400] loss: 0.462
[5,   600] loss: 0.450
[5,   800] loss: 0.482
[5,  1000] loss: 0.473
[5,  1200] loss: 0.465
[5,  1400] loss: 0.457
[5,  1600] loss: 0.456
[5,  1800] loss: 0.445
Training finished in 210.34 s on cpu
```

[ ]:
```
net_cuda = train_model_on_device("cuda:0")
```

```
[1,   200] loss: 2.285
[1,   400] loss: 2.155
[1,   600] loss: 1.328
[1,   800] loss: 0.977
[1,  1000] loss: 0.867
[1,  1200] loss: 0.832
[1,  1400] loss: 0.782
[1,  1600] loss: 0.744
[1,  1800] loss: 0.728
[2,   200] loss: 0.713
[2,   400] loss: 0.683
[2,   600] loss: 0.663
[2,   800] loss: 0.659
[2,  1000] loss: 0.642
[2,  1200] loss: 0.629
[2,  1400] loss: 0.643
[2,  1600] loss: 0.606
[2,  1800] loss: 0.591
[3,   200] loss: 0.567
[3,   400] loss: 0.570
[3,   600] loss: 0.555
[3,   800] loss: 0.581
[3,  1000] loss: 0.548
[3,  1200] loss: 0.557
```

```
[3,  1400] loss: 0.551
[3,  1600] loss: 0.526
[3,  1800] loss: 0.524
[4,   200] loss: 0.518
[4,   400] loss: 0.497
[4,   600] loss: 0.518
[4,   800] loss: 0.487
[4,  1000] loss: 0.499
[4,  1200] loss: 0.478
[4,  1400] loss: 0.497
[4,  1600] loss: 0.494
[4,  1800] loss: 0.488
[5,   200] loss: 0.478
[5,   400] loss: 0.472
[5,   600] loss: 0.472
[5,   800] loss: 0.459
[5,  1000] loss: 0.475
[5,  1200] loss: 0.467
[5,  1400] loss: 0.456
[5,  1600] loss: 0.468
[5,  1800] loss: 0.451
Training finished in 46.77 s on cuda:0
```

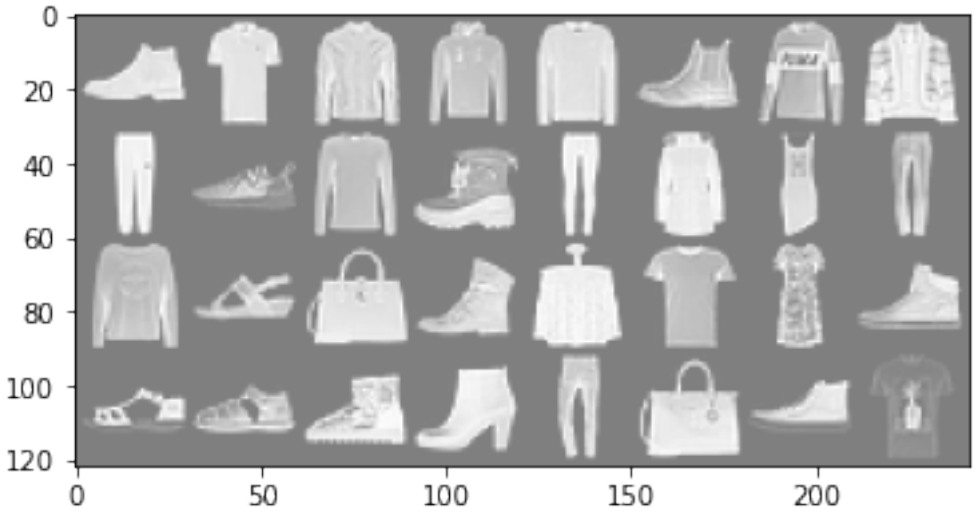
[ ]:
```
net = net_cuda
```

Skomentuj wyniki:

- Sieć neuronowa tranuje się prawie 5-krotnie szybciej z wykorzystaniem GPU niż z użyciem CPU.

[ ]:
```
dataiter = iter(testloader)
images, labels = next(dataiter)

# print images
imshow(torchvision.utils.make_grid(images))
print("GroundTruth: ", " ".join(f"{classes[labels[j]]:5s}" for j in␣
 ↪range(batch_size)))
```




```
GroundTruth:  Ankle boot top   Coat  Pullover Pullover Ankle boot Pullover Coat
Trouser Sneaker Pullover Ankle boot Trouser Coat  Dress Trouser Pullover Sandal
Bag   Ankle boot Shirt top   Dress Ankle boot Sandal Sandal Ankle boot Ankle
boot Trouser Bag   Sneaker top
```

[ ]:
```
outputs = net(images.to(device))
```

[ ]:
```
_, predicted = torch.max(outputs, 1)

print("Predicted: ", " ".join(f"{classes[predicted[j]]:5s}" for j in␣
 ↪range(batch_size)))
```

```
Predicted:  Ankle boot top   Pullover Coat  Pullover Ankle boot Shirt Coat
Trouser Sneaker Pullover Sandal Trouser Coat  Dress Trouser Pullover Sandal Bag
Ankle boot Coat  top   Dress Ankle boot Sandal Sandal Ankle boot Ankle boot
Trouser Bag   Sneaker top
```

[ ]:
```
correct = 0
total = 0
net.eval()
with torch.no_grad():
    for data in testloader:
        images, labels = data
        outputs = net(images.to(device))
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels.to(device)).sum().item()

print(f"Accuracy of the network on the 10000 test images: {100 * correct //␣
 ↪total} %")
```

```
Accuracy of the network on the 10000 test images: 82 %
```

1.2.3 Zadanie 3 (1 punkt)

Oblicz dokładność działania sieci (accuracy) dla każdej klasy z osobna.

[ ]:
```
correct_pred = {classname: 0 for classname in classes}
total_pred = {classname: 0 for classname in classes}
net.eval()

with torch.no_grad():
    for data in testloader:
        images, labels = data
        images = images.to(device)
        labels = labels.to(device)
        outputs = net(images)
        _, predictions = torch.max(outputs, 1)

        for idx, classname in enumerate(classes):
          mask = labels == idx
          class_pred = labels[mask] == predictions[mask]
          correct_pred[classname] += class_pred.sum()
          total_pred[classname] += class_pred.size(0)

for classname, correct_count in correct_pred.items():
    accuracy = 100 * float(correct_count) / total_pred[classname]
    print(f"Accuracy for class: {classname:5s} is {accuracy:.1f} %")
```

```
Accuracy for class: top   is 86.4 %
Accuracy for class: Trouser is 93.9 %
Accuracy for class: Pullover is 73.9 %
Accuracy for class: Dress is 85.8 %
Accuracy for class: Coat  is 74.7 %
Accuracy for class: Sandal is 93.6 %
Accuracy for class: Shirt is 37.3 %
Accuracy for class: Sneaker is 89.9 %
Accuracy for class: Bag   is 95.2 %
Accuracy for class: Ankle boot is 94.5 %
```

Skomentuj wyniki: - Klasy, które dotyczą ubrań, znacznie się od siebie różniących, są rozpoznawane z wysoką skutecznością, - Klasy ubrań, które są do siebie podobne (np. pullover, shirt oraz coat), rozpoznawane są ze znacznie mniejszą skutecznością niż pozostałe klasy ubrań.

1.3 Detekcja obiektów

Jest to problem odmienny od klasyfikacji obrazów, choć w praktyce ściśle z nim powiązany - modele do detekcji obiektów przeważnie do pewnego momentu wyglądają tak samo, jak modele klasyfikacji. Jednak pod koniec sieć jest dzielona na 2 wyjścia: jedno to standardowa klasyfikacja, a drugie to regresor określający pozycję obiektu na obrazie, tzw. bounding box. Najpopularniejszymi

przykładami takich sieci są YOLO i Mask R-CNN. Zbiór danych też jest odpowiednio przygotowany do tego zadania i oprócz właściwych zdjęć zawiera również maskę, gdzie tło i każdy istotny obiekt jest zaznaczony innym kolorem.

[ ]:
```
from torchvision.models import detection
import numpy as np
import cv2
from PIL import Image, ImageDraw
import urllib
```

Funkcja pozwalająca wczytać obraz z sieci:

[ ]:
```
def url_to_image(url):
    resp = urllib.request.urlopen(url)
    image = np.asarray(bytearray(resp.read()), dtype="uint8")
    image = cv2.imdecode(image, cv2.IMREAD_COLOR)
    return image
```

[ ]:
```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

classes = [
    "__background__",
    "person",
    "bicycle",
    "car",
    "motorcycle",
    "airplane",
    "bus",
    "train",
    "truck",
    "boat",
    "traffic light",
    "fire hydrant",
    "street sign",
    "stop sign",
    "parking meter",
    "bench",
    "bird",
    "cat",
    "dog",
    "horse",
    "sheep",
    "cow",
    "elephant",
    "bear",
    "zebra",
    "giraffe",
    "hat",
```

```
    "backpack",
    "umbrella",
    "handbag",
    "tie",
    "shoe",
    "eye glasses",
    "suitcase",
    "frisbee",
    "skis",
    "snowboard",
    "sports ball",
    "kite",
    "baseball bat",
    "baseball glove",
    "skateboard",
    "surfboard",
    "tennis racket",
    "bottle",
    "plate",
    "wine glass",
    "cup",
    "fork",
    "knife",
    "spoon",
    "bowl",
    "banana",
    "apple",
    "sandwich",
    "orange",
    "broccoli",
    "carrot",
    "hot dog",
    "pizza",
    "donut",
    "cake",
    "chair",
    "couch",
    "potted plant",
    "bed",
    "mirror",
    "dining table",
    "window",
    "desk",
    "toilet",
    "door",
    "tv",
    "laptop",
```

```
    "mouse",
    "remote",
    "keyboard",
    "cell phone",
    "microwave",
    "oven",
    "toaster",
    "sink",
    "refrigerator",
    "blender",
    "book",
    "clock",
    "vase",
    "scissors",
    "teddy bear",
    "hair drier",
    "toothbrush",
]

colors = np.random.randint(0, 256, size=(len(classes), 3))
```

[ ]:
```
models = {
    "frcnn-resnet": detection.fasterrcnn_resnet50_fpn,
    "frcnn-mobilenet": detection.fasterrcnn_mobilenet_v3_large_320_fpn,
    "retinanet": detection.retinanet_resnet50_fpn,
}
# load the model and set it to evaluation mode
model = models["frcnn-resnet"](
    weights=detection.FasterRCNN_ResNet50_FPN_Weights.DEFAULT,
    weights_backbone=torchvision.models.ResNet50_Weights.DEFAULT,
    progress=True,
    num_classes=len(classes)
).to(device)
model.eval()
```

[ ]:
```
FasterRCNN(
  (transform): GeneralizedRCNNTransform(
      Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
      Resize(min_size=(800,), max_size=1333, mode='bilinear')
  )
  (backbone): BackboneWithFPN(
    (body): IntermediateLayerGetter(
      (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3),
bias=False)
      (bn1): FrozenBatchNorm2d(64, eps=0.0)
      (relu): ReLU(inplace=True)
      (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1,
```

```
ceil_mode=False)
      (layer1): Sequential(
        (0): Bottleneck(
          (conv1): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (bn1): FrozenBatchNorm2d(64, eps=0.0)
          (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
          (bn2): FrozenBatchNorm2d(64, eps=0.0)
          (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1),
bias=False)
          (bn3): FrozenBatchNorm2d(256, eps=0.0)
          (relu): ReLU(inplace=True)
          (downsample): Sequential(
            (0): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
            (1): FrozenBatchNorm2d(256, eps=0.0)
          )
        )
        (1): Bottleneck(
          (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1),
bias=False)
          (bn1): FrozenBatchNorm2d(64, eps=0.0)
          (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
          (bn2): FrozenBatchNorm2d(64, eps=0.0)
          (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1),
bias=False)
          (bn3): FrozenBatchNorm2d(256, eps=0.0)
          (relu): ReLU(inplace=True)
        )
        (2): Bottleneck(
          (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1),
bias=False)
          (bn1): FrozenBatchNorm2d(64, eps=0.0)
          (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
          (bn2): FrozenBatchNorm2d(64, eps=0.0)
          (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1),
bias=False)
          (bn3): FrozenBatchNorm2d(256, eps=0.0)
          (relu): ReLU(inplace=True)
        )
      )
      (layer2): Sequential(
        (0): Bottleneck(
          (conv1): Conv2d(256, 128, kernel_size=(1, 1), stride=(1, 1),
bias=False)
          (bn1): FrozenBatchNorm2d(128, eps=0.0)
```

```
          (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1), bias=False)
          (bn2): FrozenBatchNorm2d(128, eps=0.0)
          (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1),
bias=False)
          (bn3): FrozenBatchNorm2d(512, eps=0.0)
          (relu): ReLU(inplace=True)
          (downsample): Sequential(
            (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
            (1): FrozenBatchNorm2d(512, eps=0.0)
          )
        )
        (1): Bottleneck(
          (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1),
bias=False)
          (bn1): FrozenBatchNorm2d(128, eps=0.0)
          (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
          (bn2): FrozenBatchNorm2d(128, eps=0.0)
          (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1),
bias=False)
          (bn3): FrozenBatchNorm2d(512, eps=0.0)
          (relu): ReLU(inplace=True)
        )
        (2): Bottleneck(
          (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1),
bias=False)
          (bn1): FrozenBatchNorm2d(128, eps=0.0)
          (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
          (bn2): FrozenBatchNorm2d(128, eps=0.0)
          (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1),
bias=False)
          (bn3): FrozenBatchNorm2d(512, eps=0.0)
          (relu): ReLU(inplace=True)
        )
        (3): Bottleneck(
          (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1),
bias=False)
          (bn1): FrozenBatchNorm2d(128, eps=0.0)
          (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
          (bn2): FrozenBatchNorm2d(128, eps=0.0)
          (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1),
bias=False)
          (bn3): FrozenBatchNorm2d(512, eps=0.0)
          (relu): ReLU(inplace=True)
```

```
        )
      )
      (layer3): Sequential(
        (0): Bottleneck(
          (conv1): Conv2d(512, 256, kernel_size=(1, 1), stride=(1, 1),
bias=False)
          (bn1): FrozenBatchNorm2d(256, eps=0.0)
          (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1), bias=False)
          (bn2): FrozenBatchNorm2d(256, eps=0.0)
          (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1),
bias=False)
          (bn3): FrozenBatchNorm2d(1024, eps=0.0)
          (relu): ReLU(inplace=True)
          (downsample): Sequential(
            (0): Conv2d(512, 1024, kernel_size=(1, 1), stride=(2, 2),
bias=False)
            (1): FrozenBatchNorm2d(1024, eps=0.0)
          )
        )
        (1): Bottleneck(
          (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1),
bias=False)
          (bn1): FrozenBatchNorm2d(256, eps=0.0)
          (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
          (bn2): FrozenBatchNorm2d(256, eps=0.0)
          (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1),
bias=False)
          (bn3): FrozenBatchNorm2d(1024, eps=0.0)
          (relu): ReLU(inplace=True)
        )
        (2): Bottleneck(
          (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1),
bias=False)
          (bn1): FrozenBatchNorm2d(256, eps=0.0)
          (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
          (bn2): FrozenBatchNorm2d(256, eps=0.0)
          (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1),
bias=False)
          (bn3): FrozenBatchNorm2d(1024, eps=0.0)
          (relu): ReLU(inplace=True)
        )
        (3): Bottleneck(
          (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1),
bias=False)
```

```
          (bn1): FrozenBatchNorm2d(256, eps=0.0)
          (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
          (bn2): FrozenBatchNorm2d(256, eps=0.0)
          (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1),
bias=False)
          (bn3): FrozenBatchNorm2d(1024, eps=0.0)
          (relu): ReLU(inplace=True)
        )
        (4): Bottleneck(
          (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1),
bias=False)
          (bn1): FrozenBatchNorm2d(256, eps=0.0)
          (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
          (bn2): FrozenBatchNorm2d(256, eps=0.0)
          (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1),
bias=False)
          (bn3): FrozenBatchNorm2d(1024, eps=0.0)
          (relu): ReLU(inplace=True)
        )
        (5): Bottleneck(
          (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1),
bias=False)
          (bn1): FrozenBatchNorm2d(256, eps=0.0)
          (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
          (bn2): FrozenBatchNorm2d(256, eps=0.0)
          (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1),
bias=False)
          (bn3): FrozenBatchNorm2d(1024, eps=0.0)
          (relu): ReLU(inplace=True)
        )
      )
      (layer4): Sequential(
        (0): Bottleneck(
          (conv1): Conv2d(1024, 512, kernel_size=(1, 1), stride=(1, 1),
bias=False)
          (bn1): FrozenBatchNorm2d(512, eps=0.0)
          (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1), bias=False)
          (bn2): FrozenBatchNorm2d(512, eps=0.0)
          (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1),
bias=False)
          (bn3): FrozenBatchNorm2d(2048, eps=0.0)
          (relu): ReLU(inplace=True)
          (downsample): Sequential(
```

```
            (0): Conv2d(1024, 2048, kernel_size=(1, 1), stride=(2, 2),
bias=False)
            (1): FrozenBatchNorm2d(2048, eps=0.0)
          )
        )
        (1): Bottleneck(
          (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1),
bias=False)
          (bn1): FrozenBatchNorm2d(512, eps=0.0)
          (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
          (bn2): FrozenBatchNorm2d(512, eps=0.0)
          (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1),
bias=False)
          (bn3): FrozenBatchNorm2d(2048, eps=0.0)
          (relu): ReLU(inplace=True)
        )
        (2): Bottleneck(
          (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1),
bias=False)
          (bn1): FrozenBatchNorm2d(512, eps=0.0)
          (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
          (bn2): FrozenBatchNorm2d(512, eps=0.0)
          (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1),
bias=False)
          (bn3): FrozenBatchNorm2d(2048, eps=0.0)
          (relu): ReLU(inplace=True)
        )
      )
    )
    (fpn): FeaturePyramidNetwork(
      (inner_blocks): ModuleList(
        (0): Conv2dNormActivation(
          (0): Conv2d(256, 256, kernel_size=(1, 1), stride=(1, 1))
        )
        (1): Conv2dNormActivation(
          (0): Conv2d(512, 256, kernel_size=(1, 1), stride=(1, 1))
        )
        (2): Conv2dNormActivation(
          (0): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1))
        )
        (3): Conv2dNormActivation(
          (0): Conv2d(2048, 256, kernel_size=(1, 1), stride=(1, 1))
        )
      )
      (layer_blocks): ModuleList(
```

```
        (0): Conv2dNormActivation(
          (0): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
        )
        (1): Conv2dNormActivation(
          (0): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
        )
        (2): Conv2dNormActivation(
          (0): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
        )
        (3): Conv2dNormActivation(
          (0): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
        )
      )
      (extra_blocks): LastLevelMaxPool()
    )
  )
  (rpn): RegionProposalNetwork(
    (anchor_generator): AnchorGenerator()
    (head): RPNHead(
      (conv): Sequential(
        (0): Conv2dNormActivation(
          (0): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
          (1): ReLU(inplace=True)
        )
      )
      (cls_logits): Conv2d(256, 3, kernel_size=(1, 1), stride=(1, 1))
      (bbox_pred): Conv2d(256, 12, kernel_size=(1, 1), stride=(1, 1))
    )
  )
  (roi_heads): RoIHeads(
    (box_roi_pool): MultiScaleRoIAlign(featmap_names=['0', '1', '2', '3'],
output_size=(7, 7), sampling_ratio=2)
    (box_head): TwoMLPHead(
      (fc6): Linear(in_features=12544, out_features=1024, bias=True)
      (fc7): Linear(in_features=1024, out_features=1024, bias=True)
    )
    (box_predictor): FastRCNNPredictor(
      (cls_score): Linear(in_features=1024, out_features=91, bias=True)
      (bbox_pred): Linear(in_features=1024, out_features=364, bias=True)
    )
  )
)
```

IPython, z którego korzystamy w Jupyter Notebooku, ma wbudowaną funkcję `display()` do wyświetlania obrazów.

[ ]:
```
!wget https://upload.wikimedia.org/wikipedia/commons/thumb/7/7a/
 ↪Toothbrush_x3_20050716_001.jpg/1280px-Toothbrush_x3_20050716_001.jpg␣
 ↪--output-document toothbrushes.jpg
```

```
--2023-01-14 12:14:00--  https://upload.wikimedia.org/wikipedia/commons/thumb/7/
7a/Toothbrush_x3_20050716_001.jpg/1280px-Toothbrush_x3_20050716_001.jpg
Resolving upload.wikimedia.org (upload.wikimedia.org)… 208.80.154.240,
2620:0:860:ed1a::2:b
Connecting to upload.wikimedia.org (upload.wikimedia.org)|208.80.154.240|:443…
connected.
HTTP request sent, awaiting response… 200 OK
Length: 244963 (239K) [image/jpeg]
Saving to: ‘toothbrushes.jpg’

toothbrushes.jpg    100%[===================>] 239.22K  --.-KB/s    in 0.1s

2023-01-14 12:14:00 (2.24 MB/s) - ‘toothbrushes.jpg’ saved [244963/244963]
```

[ ]:
```
image = Image.open("toothbrushes.jpg")

# make sure we have 3-channel RGB, e.g. without transparency
image = image.convert("RGB")

display(image)
```

PyTorch wymaga obrazów w kształcie [channels, height, width] (C, H, W) oraz z wartościami pikseli między 0 a 1. Pillow wczytuje obrazy z kanałami (H, W, C) oraz z wartościami pikseli między 0 a 255. Przed wykorzystaniem sieci neuronowej trzeba zatem: - zamienić obraz na tensor - zmienić kolejność kanałów - podzielić wartości pikseli przez 255

[ ]:
```
image_tensor = torch.from_numpy(np.array(image))
image_tensor = image_tensor.permute(2, 0, 1)
image_tensor_int = image_tensor  # useful for displaying, dtype = uint8
image_tensor = image_tensor / 255
image_tensor.shape, image_tensor.dtype
```

[ ]:
```
(torch.Size([3, 960, 1280]), torch.float32)
```

1.3.1 Zadanie 4 (1 punkt)

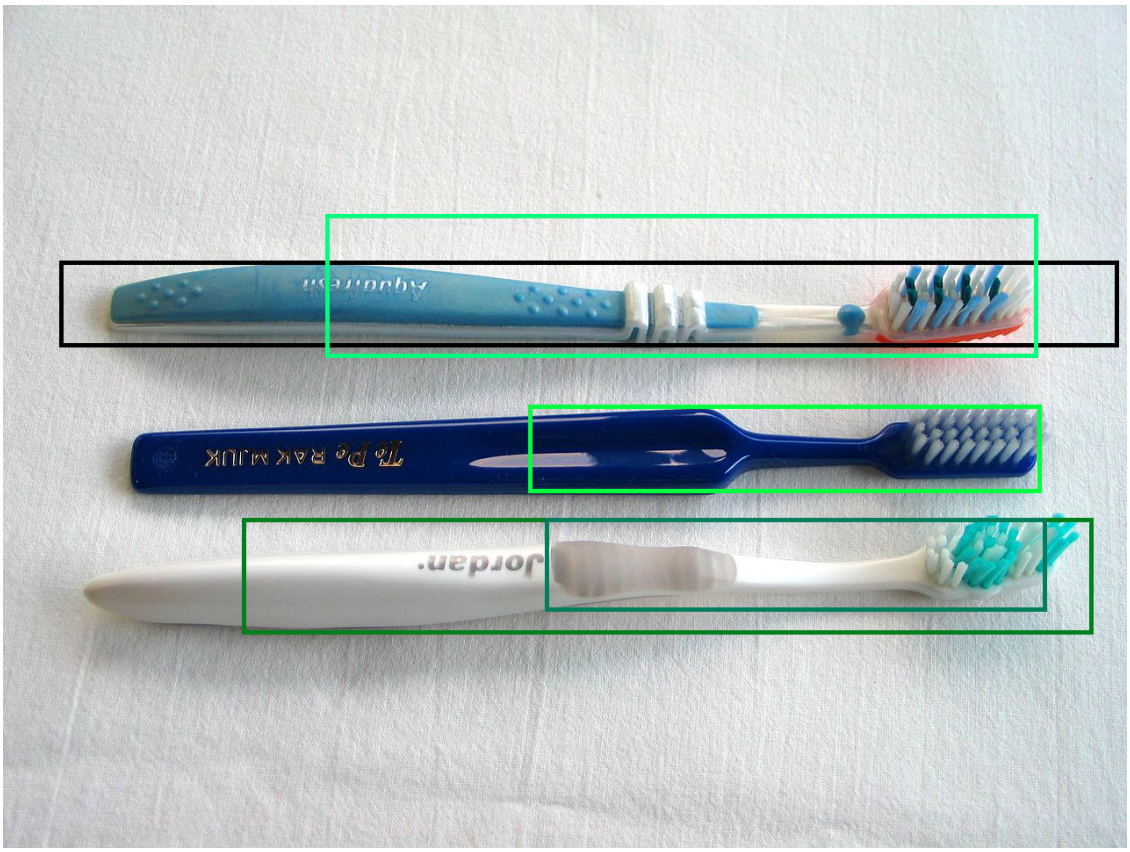
Użyj modelu do wykrycia obiektów na obrazie. Następnie wybierz tylko te bounding boxy, dla których mamy wynik powyżej 50%. Narysuj te bounding boxy, ich prawdopodobieństwa (w procentach) oraz nazwy klas.

Następnie wykorzystaj wyniki do zaznaczenia bounding box'a dla każdego wykrytego obiektu na obrazie oraz podpisz wykrytą klasę wraz z prawdopodobieństwem. Możesz tutaj użyć: - OpenCV - PyTorch - Torchvision

```
[ ]: from torchvision.utils import draw_bounding_boxes
     from torchvision.transforms import ToPILImage

     with torch.no_grad():
         predictions = model([image_tensor.to(device)])
         boxes = predictions[0]['boxes']
         scores = predictions[0]['scores']
         selected_boxes = boxes[scores > 0.5]

         image_with_boxes = draw_bounding_boxes(image_tensor_int, selected_boxes,␣
      ↪width=5)
         image = ToPILImage()(image_with_boxes)
         display(image)
```



1.4 Fine-tuning i pretrening

Jest to jedna z opcji transfer learningu. Mamy w nim już wytrenowaną sieć na dużym zbiorze danych (pretrening) i chcemy, żeby sieć poradziła sobie z nową klasą obiektów (klasyfikacja), albo lepiej radziła sobie z wybranymi obiektami, które już zna (fine-tuning). Możemy usunąć ostatnią warstwę sieci i na jej miejsce wstawić nową, identyczną, jednak z losowo zainicjalizowanymi wagami, a następnie dotrenować sieć na naszym nowym, bardziej specyficznym zbiorze danych. Przykład-

owo, jako bazę weźmiemy model wytrenowany na zbiorze ImageNet i będziemy chcieli użyć go do rozpoznawania nowych, nieznanych mu klas, np. ras psów.

Dla przykładu wykorzystamy zbiór danych z hotdogami. Będziemy chcieli stwierdzić, czy na obrazku jest hotdog, czy nie. Jako sieci użyjemy modelu ResNet-18, pretrenowanej na zbiorze ImageNet.

[ ]:
```
!wget http://d2l-data.s3-accelerate.amazonaws.com/hotdog.zip
```

```
--2023-01-14 12:14:01--  http://d2l-data.s3-accelerate.amazonaws.com/hotdog.zip
Resolving d2l-data.s3-accelerate.amazonaws.com
(d2l-data.s3-accelerate.amazonaws.com)… 13.226.15.4
Connecting to d2l-data.s3-accelerate.amazonaws.com
(d2l-data.s3-accelerate.amazonaws.com)|13.226.15.4|:80… connected.
HTTP request sent, awaiting response… 200 OK
Length: 261292301 (249M) [application/zip]
Saving to: ‘hotdog.zip.1’

hotdog.zip.1        100%[===================>] 249.19M  51.7MB/s    in 4.7s

2023-01-14 12:14:07 (52.8 MB/s) - ‘hotdog.zip.1’ saved [261292301/261292301]
```

[ ]:
```
!unzip -n hotdog.zip
```

```
Archive:  hotdog.zip
```

[ ]:
```
import os
import torch
import torch.nn as nn
import torchvision
```

Kiedy korzystamy z sieci pretrenowanej na zbiorze ImageNet, zgodnie z dokumentacją trzeba dokonać standaryzacji naszych obrazów, odejmując średnią i dzieląc przez odchylenie standardowe każdego kanału ze zbioru ImageNet.

```
All pre-trained models expect input images normalized in the same way, i.e. mini-batches of 3-c
expected to be at least 224. The images have to be loaded in to a range of [0, 1] and then norm
0.224, 0.225]. You can use the following transform to normalize:

normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                 std=[0.229, 0.224, 0.225])
```

[ ]:
```
torch.manual_seed(17)

normalize = transforms.Normalize(
    mean=[0.485, 0.456, 0.406],
    std=[0.229, 0.224, 0.225]
)
```

```
train_augs = torchvision.transforms.Compose(
    [
        torchvision.transforms.RandomResizedCrop(224),
        torchvision.transforms.RandomHorizontalFlip(),
        torchvision.transforms.ToTensor(),
        normalize,
    ]
)

test_augs = torchvision.transforms.Compose(
    [
        torchvision.transforms.Resize(256),
        torchvision.transforms.CenterCrop(224),
        torchvision.transforms.ToTensor(),
        normalize,
    ]
)
```

[ ]:
```
pretrained_net = torchvision.models.resnet18(weights=torchvision.models.
↪ResNet18_Weights.IMAGENET1K_V1)
```

[ ]:
```
pretrained_net.fc
```

[ ]:
```
Linear(in_features=512, out_features=1000, bias=True)
```

1.4.1 Zadanie 5 (1 punkt)

Dodaj warstwę liniową do naszej fine-fune'owanej sieci oraz zainicjuj ją losowymi wartościami.

[ ]:
```
finetuned_net = pretrained_net
finetuned_net.fc = nn.Linear(512, 2)
```

[ ]:
```
import time
import copy


def train_model(
    model, dataloaders, criterion, optimizer, num_epochs=25
):
    since = time.time()

    val_acc_history = []

    best_model_wts = copy.deepcopy(model.state_dict())
    best_acc = 0.0

    for epoch in range(1, num_epochs + 1):
        print("Epoch {}/{}".format(epoch, num_epochs))
```

```
        print("-" * 10)

        # Each epoch has a training and validation phase
        for phase in ["train", "val"]:
            if phase == "train":
                model.train()  # Set model to training mode
            else:
                model.eval()  # Set model to evaluate mode

            running_loss = 0.0
            running_corrects = 0

            # Iterate over data.
            for inputs, labels in dataloaders[phase]:
                inputs = inputs.to(device)
                labels = labels.to(device)

                # zero the parameter gradients
                optimizer.zero_grad()

                # forward
                # track history if only in train
                with torch.set_grad_enabled(phase == "train"):
                    # Get model outputs and calculate loss

                    outputs = model(inputs)
                    loss = criterion(outputs, labels)
                    _, preds = torch.max(outputs, 1)

                    # backward + optimize only if in training phase
                    if phase == "train":
                        loss.backward()
                        optimizer.step()

                # statistics
                running_loss += loss.item() * inputs.size(0)
                running_corrects += torch.sum(preds == labels.data)

            epoch_loss = running_loss / len(dataloaders[phase].dataset)
            epoch_acc = running_corrects.double() / len(dataloaders[phase].
↪dataset)

            print("{} Loss: {:.4f} Acc: {:.4f}".format(phase, epoch_loss,␣
↪epoch_acc))

            # deep copy the model
            if phase == "val" and epoch_acc > best_acc:
```

```
                best_acc = epoch_acc
                best_model_wts = copy.deepcopy(model.state_dict())
            if phase == "val":
                val_acc_history.append(epoch_acc)

        print()

    time_elapsed = time.time() - since
    print(
        "Training complete in {:.0f}m {:.0f}s".format(
            time_elapsed // 60, time_elapsed % 60
        )
    )
    print("Best val Acc: {:4f}".format(best_acc))

    # load best model weights
    model.load_state_dict(best_model_wts)
    return model, val_acc_history
```

[ ]:
```
data_dir = "hotdog"
batch_size = 32

model_ft = finetuned_net.to(device)
train_iter = torch.utils.data.DataLoader(
    torchvision.datasets.ImageFolder(
        os.path.join(data_dir, "train"), transform=train_augs
    ),
    batch_size=batch_size,
    shuffle=True,
)
test_iter = torch.utils.data.DataLoader(
    torchvision.datasets.ImageFolder(
        os.path.join(data_dir, "test"), transform=test_augs
    ),
    shuffle=True,
    batch_size=batch_size,
)
loss = nn.CrossEntropyLoss(reduction="none")
```

1.4.2 Zadanie 6 (1 punkt)

Zmodyfikuj tak parametry sieci, aby learning rate dla ostatniej warstwy był 10 razy wyższy niż dla pozostałych.

Trzeba odpowiednio podać pierwszy parametr `torch.optim.SGD` tak, aby zawierał parametry normalne, oraz te z `lr * 10`. Parametry warstw niższych to takie, które mają nazwę inną niż `fc.weight` albo `fc.bias` - może się przydać metoda sieci `named_parameters()`.

[ ]:
```python
def train_fine_tuning(net, learning_rate, num_epochs=15):
    begin_layers_params = []
    last_layer_params = []

    for name, param in net.named_parameters():
        if name in ('fc.weight', 'fc.bias'):
            last_layer_params.append(param)
        else:
            begin_layers_params.append(param)

    trainer = torch.optim.SGD([
        { 'params': begin_layers_params, 'lr': learning_rate },
        { 'params': last_layer_params, 'lr': learning_rate * 10 }
    ], weight_decay=0.001)

    dataloaders_dict = { 'train': train_iter, 'val': test_iter }
    criterion = nn.CrossEntropyLoss()
    model_ft, hist = train_model(
        net, dataloaders_dict, criterion, trainer, num_epochs=num_epochs
    )
    return model_ft, hist
```

[ ]:
```python
model_ft, hist = train_fine_tuning(model_ft, learning_rate=5e-5)
```

```
Epoch 1/15
----------
train Loss: 0.7730 Acc: 0.4610
val Loss: 0.6882 Acc: 0.5813

Epoch 2/15
----------
train Loss: 0.6543 Acc: 0.6030
val Loss: 0.5876 Acc: 0.7125

Epoch 3/15
----------
train Loss: 0.5714 Acc: 0.7210
val Loss: 0.5157 Acc: 0.7775

Epoch 4/15
----------
train Loss: 0.5165 Acc: 0.7695
val Loss: 0.4683 Acc: 0.8100

Epoch 5/15
----------
train Loss: 0.4761 Acc: 0.8080
val Loss: 0.4245 Acc: 0.8563
```

```
Epoch 6/15
----------
train Loss: 0.4484 Acc: 0.8225
val Loss: 0.3918 Acc: 0.8725

Epoch 7/15
----------
train Loss: 0.4125 Acc: 0.8465
val Loss: 0.3710 Acc: 0.8788

Epoch 8/15
----------
train Loss: 0.3912 Acc: 0.8615
val Loss: 0.3499 Acc: 0.8900

Epoch 9/15
----------
train Loss: 0.3837 Acc: 0.8570
val Loss: 0.3313 Acc: 0.8938

Epoch 10/15
----------
train Loss: 0.3690 Acc: 0.8625
val Loss: 0.3203 Acc: 0.9000

Epoch 11/15
----------
train Loss: 0.3625 Acc: 0.8570
val Loss: 0.3081 Acc: 0.8950

Epoch 12/15
----------
train Loss: 0.3539 Acc: 0.8765
val Loss: 0.3008 Acc: 0.9062

Epoch 13/15
----------
train Loss: 0.3347 Acc: 0.8805
val Loss: 0.2911 Acc: 0.9062

Epoch 14/15
----------
train Loss: 0.3310 Acc: 0.8840
val Loss: 0.2841 Acc: 0.9075

Epoch 15/15
----------
```

```
train Loss: 0.3198 Acc: 0.8790
val Loss: 0.2778 Acc: 0.9100

Training complete in 4m 39s
Best val Acc: 0.910000
```

skomentuj wyniki:

- Po 15 epokach otrzymujemy dosyć wysokie accuracy, a więc trening daje dobre rezultaty.

Przy wyświetlaniu predykcji sieci musimy wykonać operacje odwrotne niż te, które wykonaliśmy, przygotowując obrazy do treningu: - zamienić kolejność kanałów z (C, H, W) na (H, W, C) - zamienić obraz z tensora na tablicę Numpy'a - odwrócić normalizację (mnożymy przez odchylenie standardowe, dodajemy średnią) i upewnić się, że nie wychodzimy poza zakres [0, 1] (wystarczy proste przycięcie wartości)

[ ]:
```python
def imshow(img, title=None):
    img = img.permute(1, 2, 0).numpy()
    means = np.array([0.485, 0.456, 0.406])
    stds = np.array([0.229, 0.224, 0.225])
    img = stds * img + means
    img = np.clip(img, 0, 1)

    plt.imshow(img)
    if title is not None:
        plt.title(title)

    plt.pause(0.001)
```

[ ]:
```python
import matplotlib.pyplot as plt
plt.ion()


def visualize_model(model, num_images=6):
    class_names = ["hotdog", "other"]
    model.eval()
    images_so_far = 0
    fig = plt.figure()
    with torch.no_grad():
        for i, (inputs, labels) in enumerate(test_iter):
            inputs = inputs.to(device)
            labels = labels.to(device)

            outputs = model(inputs)
            _, preds = torch.max(outputs, 1)

            for j in range(inputs.size()[0]):
                images_so_far += 1
                ax = plt.subplot(num_images // 2, 2, images_so_far)
```

```
                ax.axis('off')
                ax.set_title(f'predicted: {class_names[preds[j]]}')

                imshow(inputs.data[j].cpu())

                if images_so_far == num_images:
                    return
```

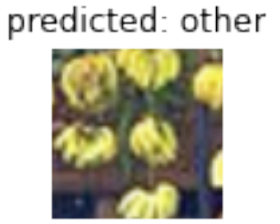
[ ]:
```
visualize_model(model_ft)
```

predicted: hotdog

predicted: other

predicted: hotdog

predicted: hotdog



1.5 Rozpoznawanie, kto jest na zdjęciu

[ ]:
```
!pip install facenet-pytorch
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-
wheels/public/simple/
Requirement already satisfied: facenet-pytorch in /usr/local/lib/python3.8/dist-
packages (2.5.2)
Requirement already satisfied: numpy in /usr/local/lib/python3.8/dist-packages
(from facenet-pytorch) (1.21.6)
Requirement already satisfied: pillow in /usr/local/lib/python3.8/dist-packages
(from facenet-pytorch) (7.1.2)
Requirement already satisfied: requests in /usr/local/lib/python3.8/dist-
packages (from facenet-pytorch) (2.25.1)
Requirement already satisfied: torchvision in /usr/local/lib/python3.8/dist-
packages (from facenet-pytorch) (0.14.0+cu116)
Requirement already satisfied: certifi>=2017.4.17 in
/usr/local/lib/python3.8/dist-packages (from requests->facenet-pytorch)
(2022.12.7)
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.8/dist-
packages (from requests->facenet-pytorch) (2.10)
Requirement already satisfied: chardet<5,>=3.0.2 in
/usr/local/lib/python3.8/dist-packages (from requests->facenet-pytorch) (4.0.0)
Requirement already satisfied: urllib3<1.27,>=1.21.1 in
/usr/local/lib/python3.8/dist-packages (from requests->facenet-pytorch) (1.24.3)
```

```
Requirement already satisfied: typing-extensions in
/usr/local/lib/python3.8/dist-packages (from torchvision->facenet-pytorch)
(4.4.0)
Requirement already satisfied: torch==1.13.0 in /usr/local/lib/python3.8/dist-
packages (from torchvision->facenet-pytorch) (1.13.0+cu116)
```

[ ]:
```
import urllib
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from PIL import Image
from facenet_pytorch import InceptionResnetV1, MTCNN
from torchvision.transforms.functional import to_pil_image
```

[ ]:
```
!wget https://raw.githubusercontent.com/timesler/facenet-pytorch/feature/
 ↪add_vggface2_labels/data/labels-vggface2.csv --output-document␣
 ↪vggface_labels.csv
```

```
--2023-01-14 12:18:50--  https://raw.githubusercontent.com/timesler/facenet-
pytorch/feature/add_vggface2_labels/data/labels-vggface2.csv
Resolving raw.githubusercontent.com (raw.githubusercontent.com)…
185.199.108.133, 185.199.109.133, 185.199.110.133, …
Connecting to raw.githubusercontent.com
(raw.githubusercontent.com)|185.199.108.133|:443… connected.
HTTP request sent, awaiting response… 200 OK
Length: 139040 (136K) [text/plain]
Saving to: ‘vggface_labels.csv’

vggface_labels.csv  100%[===================>] 135.78K  --.-KB/s    in 0.02s

2023-01-14 12:18:50 (5.88 MB/s) - ‘vggface_labels.csv’ saved [139040/139040]
```

1.6 Jak to działa?

Zanim w ogóle pomyślimy o rozpoznawaniu twarzy, najpierw musimy zorientować się, czy w ogóle twarz jest na obrazie, i jeśli tak, to ją wyodrębnić, bo reszta obrazu nas nie interesuje. Ten proces przeważnie znajduje wszystkie twarze na obrazie, nie tylko jedną. Istnieją dwie podstawowe metody znajdowania twarzy na obrazie: - kaskady Haara - klasyczne podejście, szybkie, w miarę skuteczne, nieodporne na dużę ruchy głowy, okulary itp. (są różne kaskady dla takich przypadków) - sieci neuronowe, np. MTCNN - bardziej współczesne podejście, są one wolniejsze (a nawet wolne), więc wymagają więcej mocy obliczeniowej, ale za to są skuteczniejsze i bardziej odporne na wszelakie zakłócenia.

Implementacja MTCNN z biblioteki `facenet-pytorch` ma 2 możliwości użycia: 1. Przez metodę `.detect()`, np. `mtcnn.detect(img)`. W ten sposób dostajemy koordynaty (bounding box) twarzy na oryginalnym obrazie. Ma to zastosowanie w samym wykrywaniu twarzy, lub kiedy chcemy później zastosować własne implementacje sieci do rozpoznawania twarzy. 2. Przez metodę `__call__()`, np. `mtcnn(img)`. W ten sposób dostajemy wyciętą i znormalizowaną twarz, go-

tową do późniejszego użycia w rozpoznawaniu twarzy za pomocą tej biblioteki. Normalizacja to przeskalowanie twarzy tak, żeby miała równe wymiary, oraz standaryzacja kolorów.

Zobaczmy teraz, jak działa pierwsza opcja.

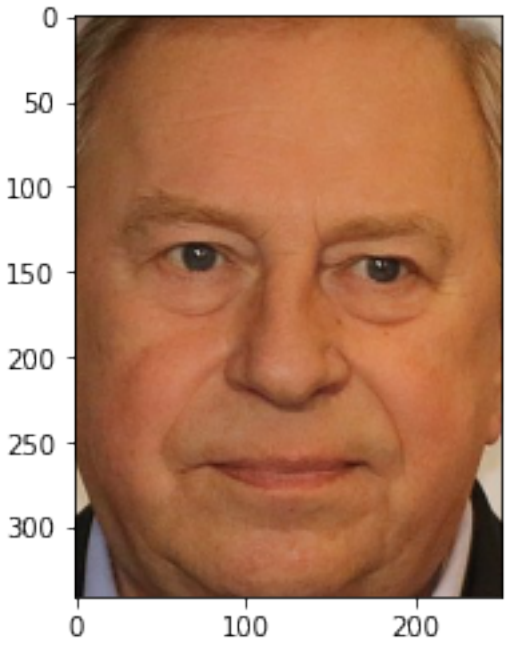
[ ]:
```
jerzy_stuhr_image_path = "https://upload.wikimedia.org/wikipedia/commons/thumb/
↪9/91/Jerzy_Stuhr_27_stycznia_2018.jpg/1024px-Jerzy_Stuhr_27_stycznia_2018.
↪jpg"

img = Image.open(urllib.request.urlopen(jerzy_stuhr_image_path)).convert("RGB")

detector = MTCNN(device=device)

boxes, probabilities = detector.detect(img)
box = boxes[0]
face = img.crop(box)

plt.imshow(face)
plt.show()
```



1.6.1 Zadanie 7 (1 punkt)

Uzupełnij kod funkcji `extract_face_haar()` tak, by działała tak samo, jak `extract_face_mtcnn()`, ale z wykorzystaniem kaskady Haara z OpenCV. Ma zwracać tensor PyTorcha, abyśmy mogli dalej wykorzystać go do identyfikacji z użyciem sieci neuronowej: - typu `float32` (OpenCV używa `uint8`) - z zakresem wartości `[0, 1]` (OpenCV używa `[0, 255]`)

- z kolejnością kanałów `(C, H, W)` (OpenCV, jak Numpy, używa `(H, W, C)`) - na urządzeniu `device`

Mogą się przydać: - `cvtColor()` - `CascadeClassifier()` - `detectMultiScale()`

W funkcji `extract_face_mtcnn` mamy dodatkowy argument `post_process`. Gdy ma on wartość `False`, to detektor twarzy nie normalizuje obrazu (jedynie zmienia rozmiar na kwadrat) i będzie się wyświetlał "normalnie". Kiedy natomiast chcemy użyć wyjścia z detektora twarzy jako wejścia do drugiej sieci, do identyfikacji osób, to trzeba użyć `post_process=True`, aby użyć odpowiednich transformacji. Typowo daje to lepsze wyniki.

[ ]:
```
!wget https://raw.githubusercontent.com/opencv/opencv/master/data/haarcascades/
↪haarcascade_frontalface_default.xml
```

```
--2023-01-14 12:18:51--  https://raw.githubusercontent.com/opencv/opencv/master/
data/haarcascades/haarcascade_frontalface_default.xml
Resolving raw.githubusercontent.com (raw.githubusercontent.com)…
185.199.108.133, 185.199.111.133, 185.199.110.133, …
Connecting to raw.githubusercontent.com
(raw.githubusercontent.com)|185.199.108.133|:443… connected.
HTTP request sent, awaiting response… 200 OK
Length: 930127 (908K) [text/plain]
Saving to: 'haarcascade_frontalface_default.xml.1'

haarcascade_frontal 100%[===================>] 908.33K  --.-KB/s    in 0.05s

2023-01-14 12:18:51 (18.6 MB/s) - 'haarcascade_frontalface_default.xml.1' saved
[930127/930127]
```

[ ]:
```
def extract_face_mtcnn(image_path, output_face_img_size=160,␣
↪post_process=False):
    # load image from URL
    img = Image.open(urllib.request.urlopen(image_path)).convert("RGB")

    # create the detector
    detector = MTCNN(
        output_face_img_size,
        device=device,
        post_process=post_process
    )
    # print (type(torch.Tensor(img).to(device)))

    # get face from image
    face = detector(img)

    # apply very basic normalization manually when postprocessing is
    # not used - change values from default [0, 255] to [0, 1]
    if not post_process:
```

```
        face /= 255

    return face
```

[ ]:
```
face_mtcnn = extract_face_mtcnn(jerzy_stuhr_image_path)

print(face_mtcnn.min(), face_mtcnn.max(), face_mtcnn.dtype)

# plot the extracted face
to_pil_image(face_mtcnn)
```

```
tensor(0.0118) tensor(0.8706) torch.float32
```

[ ]:

[ ]:
```
import cv2


def extract_face_haar(filename, required_size=160):
    image = np.array(Image.open(urllib.request.urlopen(filename)))

    # create the detector
    detector = cv2.CascadeClassifier()
    detector.load("./haarcascade_frontalface_default.xml")

    # get face from image
    gray = cv2.equalizeHist(cv2.cvtColor(image, cv2.COLOR_BGR2GRAY))
    predictions = detector.detectMultiScale(gray)
    # get only the first face as we need to predict only one face
    x, y, w, h = predictions[0]
    face = image[y:y+h, x:x+w, :]

    # resize pixels to the model size
    image = Image.fromarray(face)
    image = image.resize((required_size, required_size))
```

```
    face_array = np.asarray(image)
    # change channel order from (H, W, C) to (C, H, W)
    face = np.moveaxis(face_array, 2, 0)

    # Numpy array -> PyTorch tensor on appropriate device
    face = torch.Tensor(face).to(device)

    # convert value range
    face /= 255

    return face
```

[ ]:
```
face_haar = extract_face_haar(jerzy_stuhr_image_path)

# plot the extracted face
to_pil_image(face_haar)
```

[ ]:

Skomentuj wyniki uzyskane przez powyższe metody:

- W przypadku obu metod twarz została poprawnie rozpoznana. Druga metoda (Haara) pozwala na dokładniejsze wykadrowanie twarzy bez rozciągania obrazka.

Skoro udało nam się już znaleźć twarz na obrazku, to spróbujmy rozpoznać, kto się tam znajduje. Znów mamy kilka możliwości, jak to zrobić: - przygotować sami duży zbiór i nauczyć własną sieć neuronową - zdecydowanie za dużo pracy jak na nasze możliwości na zajęciach - wykorzystać gotowy dataset - też zbyt czasochłonne, bo uczenie sieci jest bardziej czasochłonne niż jej używanie - moglibyśmy w ogóle zrezygnować z sieci neuronowej - ale stracilibyśmy na jakości (dokładności) naszego rozwiązania - wykorzystać przygotowaną już wcześniej sieć

Użyjemy sieci InceptionResnetV1, wytrenowanej na zbiorze twarzy VGGFace2. Jeśli chcesz dowiedzieć się więcej o tej architekturze, to tutaj znajdziesz jej opis. Przykład użycia pochodzi z oficjalnego tutoriala.

[ ]:
```
identifier = InceptionResnetV1(pretrained="vggface2", classify=True,␣
↪device=device).eval()

labels = pd.read_csv("vggface_labels.csv", encoding="UTF-8").values.tolist()
```

[ ]:
```
from torch.nn.functional import softmax



def get_face_extractor(method, post_process):
    if method == "mtcnn":
        return lambda x: extract_face_mtcnn(x, post_process=post_process)
    elif method == "haar":
        return extract_face_haar
    else:
        raise ValueError(
            f"Method should be either 'mtcnn' or 'haar', got: '{method}'"
        )


def get_top_5_faces(image_path, method="mtcnn", post_process=False):
    detector = get_face_extractor(method, post_process)

    with torch.no_grad():
        # extract face
        face = detector(image_path)
        # VGGFace2 classification logits; have to provide batch (4D tensor),
        # so we add "fake" dimension via .unsqueeze()
        logits = identifier(face.to(device).unsqueeze(0)).cpu()

    # we know there is only 1 face, so we can select first (and only) element
    logits = logits[0]

    # get probabilities
    probas = softmax(logits, dim=0)

    # get top 5 predictions
    top_probas, top_indices = torch.topk(probas, k=5)

    top_probas = top_probas.tolist()
    top_indices = top_indices.tolist()
    top_labels = [labels[idx][0] for idx in top_indices]

    for label, proba in zip(top_labels, top_probas):
        label = label.replace("_", " ")
        print(f"{label:<20} {100 * proba:.2f}%")
```

[ ]:
```
get_top_5_faces(jerzy_stuhr_image_path, method="haar")
print()
get_top_5_faces(jerzy_stuhr_image_path, method="mtcnn", post_process=False)
print()
get_top_5_faces(jerzy_stuhr_image_path, method="mtcnn", post_process=True)
```

```
Jerzy Stuhr          79.74%
Felipe Gonzales      2.21%
Jean Ziegler         0.34%
JiÅ Ã LÃ¡bus        0.30%
Marty Markowitz      0.29%

Jerzy Stuhr          88.59%
Felipe Gonzales      1.18%
Eckart Witzigmann    0.22%
Ilkka Kanerva        0.15%
Marty Markowitz      0.14%

Jerzy Stuhr          83.96%
Felipe Gonzales      0.98%
Eckart Witzigmann    0.27%
Michael Bloomberg    0.22%
Hans Vestberg        0.18%
```

No fajne, ale co, jeśli chcielibyśmy rozpoznać kogoś, kto nie był częścią pierwotnego zbioru danych? Czy musimy trenować sieć całkowicie od nowa? Oczywiście nie - byłoby to bardzo nieefektywne, bo sieć ta była trenowana na ponad milionie obrazów 8631 ludzi, więc aby dodać jednego, bezsensem byłoby powtarzać cały proces od zera.

Więc jak to zrobić? Można znowu na różne sposoby. Przykładowo, nie ma potrzeby trenować całej sieci, a wystarczy wytrenować ostatnią warstwę, która odpowiada za rozpoznawanie danej osoby. Poprzednie w zasadzie tylko enkodują daną osobę w postaci wektora liczb, tak, aby wektory dla tej samej osoby były blisko sobie, a dla różnych daleko (w sensie pewnej metryki), wieć dla nieznanego obrazu nowej osoby też to zadziała. Musimy jedynie stwierdzić, że to ta sama osoba.

[ ]:
```
from torchvision.transforms.functional import to_tensor

def get_embeddings(image_paths, method="mtcnn", post_process=False):
    detector = get_face_extractor(method, post_process)
    embedder = InceptionResnetV1(pretrained="vggface2", classify=False,␣
 ↪device=device).eval()

    embeddings = []

    with torch.no_grad():
        for image_path in image_paths:
            face = detector(image_path)
            embedding = embedder(face.to(device).unsqueeze(0)).cpu()
```

```
            embedding = embedding.flatten()
            embeddings.append(embedding)

    return embeddings
```

1.6.2 Zadanie 8 (1 punkt)

Oblicz dystans euklidesowy i cosinusowy między embeddingami. Następnie przetestuj poszczególne metody przygotowanym kodem.

[ ]:
```
import torch.nn.functional as F

def is_match(known_embedding, candidate_embedding, thresh=0.4, euc_thresh=1.2):
    cosine_score = 1 - F.cosine_similarity(known_embedding,␣
 ↪candidate_embedding, dim=0)
    euclidean_score = (candidate_embedding - known_embedding).pow(2).sum().
 ↪sqrt()

    if cosine_score <= thresh:
        print(">face is a Match - cosine (%.3f <= %.3f)" % (cosine_score,␣
 ↪thresh))
        print(">face is a Match - euclidean (%.3f <= %.3f)" % (euclidean_score,␣
 ↪euc_thresh))
    else:
        print(">face is NOT a Match (%.3f > %.3f)" % (cosine_score, thresh))
        print(">face is NOT a Match (%.3f > %.3f)" % (euclidean_score,␣
 ↪euc_thresh))
```

[ ]:
```
def test_person_identification(method, post_process=False):
    # test: Jerzy Stuhr vs new face, Maciej Stuhr
    filenames = [
        "https://upload.wikimedia.org/wikipedia/commons/thumb/9/91/
 ↪Jerzy_Stuhr_27_stycznia_2018.jpg/1024px-Jerzy_Stuhr_27_stycznia_2018.jpg",

        "https://s3.viva.pl/newsy/jerzy-stuhr-276699-GALLERY_600.jpg",
        "https://ocdn.eu/images/pulscms/MjM7MDA_/
 ↪0d5516ac7244156c40d1366ee008d7e5.jpeg",

        "https://upload.wikimedia.org/wikipedia/commons/thumb/f/f0/
 ↪2016_Woodstock_328_Maciej_Stuhr.jpg/800px-2016_Woodstock_328_Maciej_Stuhr.
 ↪jpg",
        "https://secretum.pl/media/k2/items/cache/
 ↪7f2cd38b7681e6e2ef83b5a7a5385264_XL.jpg?t=20141001_064823",
    ]

    embeddings = get_embeddings(filenames, method, post_process)
```

```
    print("Positive Tests")
    is_match(embeddings[0], embeddings[1])
    is_match(embeddings[0], embeddings[2])
    print()

    print("Negative Tests")
    is_match(embeddings[0], embeddings[3])
    is_match(embeddings[0], embeddings[4])
    print()
```

[ ]:
```
test_person_identification(method="haar")
test_person_identification(method="mtcnn", post_process=False)
test_person_identification(method="mtcnn", post_process=True)
```

```
Positive Tests
>face is a Match - cosine (0.314 <= 0.400)
>face is a Match - euclidean (0.792 <= 1.200)
>face is a Match - cosine (0.380 <= 0.400)
>face is a Match - euclidean (0.872 <= 1.200)

Negative Tests
>face is NOT a Match (0.826 > 0.400)
>face is NOT a Match (1.285 > 1.200)
>face is NOT a Match (1.152 > 0.400)
>face is NOT a Match (1.518 > 1.200)

Positive Tests
>face is a Match - cosine (0.313 <= 0.400)
>face is a Match - euclidean (0.791 <= 1.200)
>face is a Match - cosine (0.234 <= 0.400)
>face is a Match - euclidean (0.683 <= 1.200)

Negative Tests
>face is NOT a Match (0.772 > 0.400)
>face is NOT a Match (1.242 > 1.200)
>face is NOT a Match (1.038 > 0.400)
>face is NOT a Match (1.441 > 1.200)

Positive Tests
>face is a Match - cosine (0.281 <= 0.400)
>face is a Match - euclidean (0.750 <= 1.200)
>face is a Match - cosine (0.278 <= 0.400)
>face is a Match - euclidean (0.746 <= 1.200)

Negative Tests
>face is NOT a Match (0.759 > 0.400)
>face is NOT a Match (1.232 > 1.200)
>face is NOT a Match (1.007 > 0.400)
```

```
>face is NOT a Match (1.419 > 1.200)
```

Skomentuj wyniki:

- Otrzymaliśmy poprawne wyniki testów

1.7 Pytania kontrolne (1 punkt)

1. Jakiego algorytmu użyłbyś do wykrywania obiektów w czasie rzeczywistym?

- YOLO (You Only Look Once) - ten algorytm pozwala na szybkie przetwarzanie obrazków z wysoką dokładnością predykcji. Wykorzystuje jedną konwulcyjną sieć neuronową (CCN) do przetwarzania całej ilustracji, dzięki czmeu dobrze nadaje się do precykcji w czasie rzeczywistym,
- SSD (Single Shot MultiBox Detector) - inny algorytm, wykorzystujący również pojedynczą konwulcyjną sięć neuronową (CCN). Jego szybkość i precyzja jest zbliżona do algorytmu YOLO.

2. Jaki krok (stride) jest najpowszechniej stosowany i dlaczego?

Najczęściej stosowanym krokiem w konwulcyjnych sieciach neuronowych (CCNs) jest stride o wartości 1. Oznacza to, że macierz filtrów przesuwa się z dokładnością jednego piksela w czasie.

Istnieje kilka powodów, dla których stosuje się taki krok: - stride o wartości 1 zachowuje rozdzielczość wejściowego obrazu, dzięki czemu zachowane zostają wszystkie detale, - krok 1 zapewnia, że filtry pokrywają cały obraz wejściowy, co zapobiega utracie informacji, - pozwala na bardziej efektywne wykorzystanie filtrów, ponieważ pokrywają one cały obrazek

3. Czy sieci konwolucyjne nadają się do analizy sygnału audio i dlaczego?

Sieci konwulcyjne nie nadają się do analizy sygnału audio, ponieważ: - są przystosowane do przetwarzania danych, które można reprezentować przy pomocy macierzy, a więc lepiej nadają się do analizy pojedynczych obrazków lub wideo (wideo składa się z wielu klatek, będących pojedynczymi obrazkami). Obrazki da się łatwo przekształcić na macierz pikseli, podczas gdy reprezentacja dźwięku w ten sposób jest dużo trudniejsza, - sygnały dźwiękowe są jednowymiarowe, w przeciwieństwie do obrazków, które mają 2 wymiary, - dźwięk może mieć różną długość (nagranie dźwiękowe), podczas gdy obrazek zwykle ma ustalone wymiary.

1.8 Zadanie dla chętnych

W zadaniach dotyczących klasyfikacji obrazu wykorzystywaliśmy prosty zbiór danych i sieć LeNet. Teraz zamień zbiór danych na bardziej skomplikowany, np. ten lub ten (lub inny o podobnym poziomie trudności) i zamiast prostej sieci LeNet użyj bardziej złożonej, np. AlexNet, ResNet, MobileNetV2.