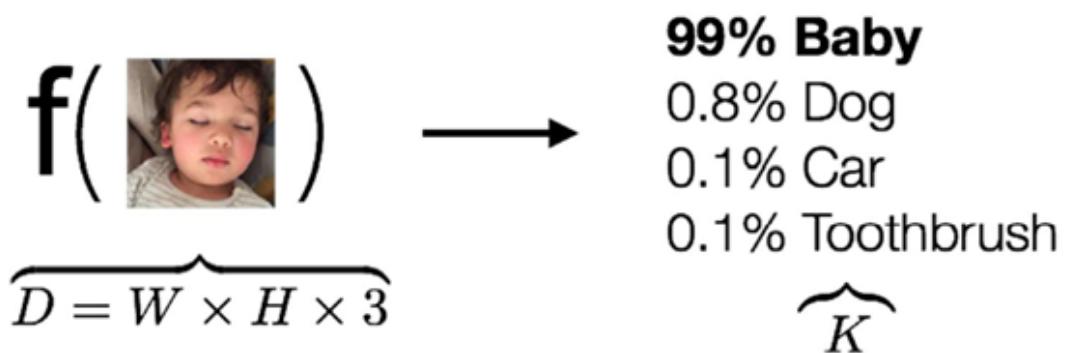


Konwolucyjne sieci neuronowe

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.

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 dobrze znanych algorytmów 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.

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 kerolem). Następnie sumujemy wynik i zapisujemy do macierzy wynikowej na odpowiedniej

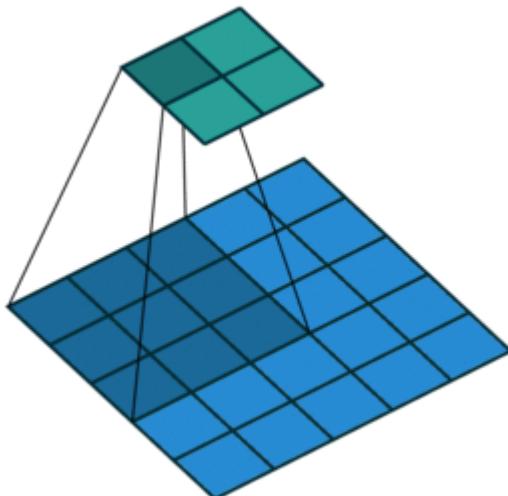
pozycji. Proces powtarza się aż do momentu przeskanowania całego obrazu. Taki filtr wykrywa, czy coś do niego pasuje w danym miejscu, i z tego wynika zdolność semantycznej generalizacji sieci – uczymy się cech, a wykrywamy je potem w dowolnym miejscu. [Przydatne pojęcia](#)

3 ₀	3 ₁	2 ₂	1	0
0 ₂	0 ₂	1 ₀	3	1
3 ₀	1 ₁	2 ₂	2	3
2	0	0	2	2
2	0	0	0	1

12.0	12.0	17.0
10.0	17.0	19.0
9.0	6.0	14.0

Stride

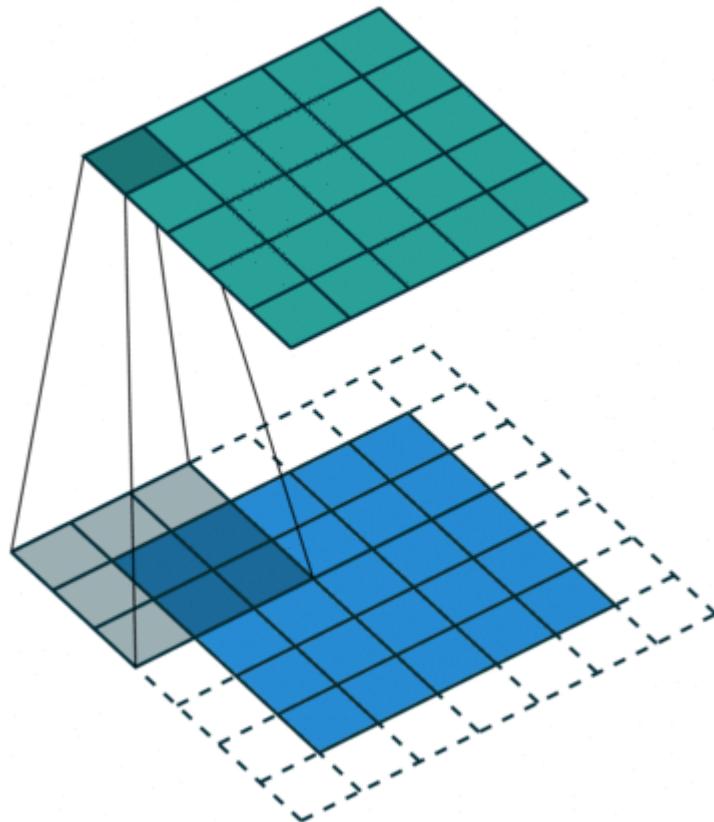
Stride to inaczej *krok algorytmu*, albo *przesunięcie*. Określa co ile komórek macierzy następuje aplikacja operatora konwolucji. Jeśli stride wynosi 1 a operator konwolucji ma rozmiar np. 3 x 3, to każdy piksel (z wyjątkiem skrajnych narożnych pikseli) będzie uczestniczył w wielu operacjach konwolucji. Jeśli natomiast krok wyniosłby 3, to każdy piksel uczestniczyłby tylko jednokrotnie w tych operacjach. Należy pamiętać, że krok stosujemy zarówno w poziomie, jak i pionie. Najczęściej w obu kierunkach wykorzystuje się ten sam krok.



Padding

Padding to inaczej *wypełnienie* krawędzi obrazu. Określa, w jaki sposób będą traktowane skrajne piksele. Jeśli padding wynosi 0, to skrajne piksele będą uczestniczyły w operacjach konwolucji rzadziej, niż pozostałe piksele (oczywiście jest to również uzależnione od wartości kroku). Aby zniwelować ten efekt, możemy dodać wypełnienie wokół całego obrazu. Te dodatkowe piksele mogą być zerami, albo mogą być również jakimiś uśrednionymi wartościami pikseli sąsiednich. Wypełnienie zerami oznacza de facto obramowanie całego obrazu czarną ramką.

[Więcej na temat wypełnienia.](#)



Pooling

Pooling jest procesem wykorzystywanym do redukcji rozmiaru obrazu. Występują 2 warianty: *max-pooling* oraz *avg-pooling*. 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 wprowadzamy pewną inwariancję w wagach sieci i jesteśmy odporni na niewielkie wahania lokalizacji informacji, a skupiamy się na "większym obrazie".

0	50	0	29
0	80	31	2
33	90	0	75
0	9	0	95

80	?
?	?

Sposoby redukcji przeuczenia

- warstwa dropout - wyłączanie losowych neuronów w trakcie uczenia,
- regularyzacja wag - ograniczenie sumy wartości wag,
- metoda wcześniego stopu (early stopping) - zatrzymanie uczenia, jeśli proces uczenia nie poprawia wyników,
- normalizacja paczki (batch normalization) - centrowanie i skalowanie wartości wektorów *w obrębie batcha danych*,
- rozszerzanie danych (data augmentation) - generowanie lekko zaburzonych danych, na podstawie danych treningowych,
- lub... więcej danych.

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.

Warstwa konwolucyjna sieci neuronowej składa się z wielu filtrów konwolucyjnych działających równolegle (tj. wykrywających różne cechy). Wagi kerneli, początkowo zainicjalizowane losowo, są dostrajane w procesie uczenia. Wynik działania poszczególnych filtrów jest przepuszczany przez funkcję nieliniową. Mamy tu do czynienia z sytuacją analogiczną jak w MLP: najpierw wykonujemy przekształcenie liniowe, a potem stosujemy funkcję aktywacji. Funkcji aktywacji nie stosuje się jednak po warstwach poolingowych, są to stałe operacje nie podlegające uczeniu.

W celu klasyfikacji obrazu musimy później użyć sieci MLP. Jako że wejściem do sieci MLP jest zawsze wektor, a wyjściem warstwy konwolucyjnej obraz. Musimy zatem obraz przetworzony przez filtry konwolucyjne sprowadzić do formy wektora, tzw. **embedding-u / osadzenia**, czyli reprezentacji obrazu jako punktu w pewnej ciągłej przestrzeni. Służy do tego warstwa spłaszczająca (flatten layer), rozwijająca 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**. Główka 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 wystarcza do klasyfikacji embeddingu.

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

Wybierzmy rodzaj akceleracji. Współczesne wersje PyTorch wspierają akcelerację nie tylko na kartach Nvidii i AMD, ale również na procesorach Apple z serii M. Obsługa AMD jest realizowana identycznie jak CUDA natomiast MPS (Apple) ma nieco inne API do sprawdzania dostępności i wybierania urządzenia. Zapisujemy wybrane urządzenie do zmiennej `device`, dzięki czemu w dalszych częściach kodu już nie będziemy musieli o tym myśleć.

```
In [ ]: if torch.cuda.is_available():
    device = torch.device("cuda:0")
elif torch.backends.mps.is_available():
    device = torch.device("mps")

print(device)
```

mps

W pakiecie torchvision mamy funkcje automatycznie pobierające niektóre najbardziej popularne zbiory danych z obrazami.

W tym ćwiczeniu wykorzystamy zbiór FashionMNIST, który zawiera małe (28x28) zdjęcia ubrań w skali szarości. Zbiór ten został stworzony przez Zalando i jest "modowym" odpowiednikiem "cyfrowego" MNIST-a, jest z nim kompatybilny pod względem rozmiarów i charakterystyki danych, ale jest od MNIST-a trudniejszy w klasyfikacji.

Do funkcji ładujących zbiory danych możemy przekazać przekształcenie, które powinno zostać na nim wykonane. Przekształcenia można łączyć przy użyciu `transforms.Compose`. W tym przypadku przekonwertujemy dane z domyślnej reprezentacji PIL.Image na torch-owe tensory.

Pobrany dataset przekazujemy pod kontrolę DataLoader-a, który zajmuje się podawaniem danych w batch-ach podczas treningu.

```
In [ ]: 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",
)
print(type(testset[0][0]))

```

<class 'torch.Tensor'>

Zobaczmy, co jest w naszym zbiorze danych. Poniżej kawałek kodu, który wyświetli nam kilka przykładowych obrazków. Wartości pikseli są znormalizowane do przedziału [0,1].

```

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

def imshow(img):
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.axis("off")
    plt.show()

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

def grid_show(images, nrow=8):
    imshow(torchvision.utils.make_grid(images, nrow=nrow))

def print_grid(labels, nrow=8):
    rows = [labels[n : n + nrow] for n in range(0, len(labels), nrow)]
    for r in rows:
        print(" ".join(f"{classes[c]:10s}" for c in r))

grid_show(images)
print_grid(labels)

```



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

LeNet

LeNet to bardzo znany, klasyczny model sieci konwolucyjnej.

Warstwy:

- obraz
- konwolucja, kernel 5×5 , bez paddingu, 6 kanałów (feature maps)
- average pooling, kernel 2×2 , stride 2
- konwolucja, kernel 5×5 , bez paddingu, 16 kanałów (feature maps)
- average 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

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()`

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

class LeNet(nn.Module):
    def __init__(self):
        super().__init__()

        self.cnn = nn.Sequential(
            nn.Conv2d(1, 20, kernel_size=5, padding=2),
            # (28 + 2 * 2 - (5 - 1), 28 + 2 * 2 - (5 - 1), 20)
            # (28, 28, 20)
            nn.ReLU(),
            nn.AvgPool2d(kernel_size=2, stride=2),
            # (28 / 2, 28 / 2, 20)
            nn.Conv2d(20, 50, kernel_size=5),
            # (10, 10, 50)
            nn.ReLU(),
            nn.AvgPool2d(kernel_size=2, stride=2),
            # (5, 5, 50)
            nn.Flatten(),
            nn.Linear(50 * 5**2, 300),
            nn.ReLU(),
            nn.Linear(300, 100),
            nn.ReLU(),
            nn.Linear(100, len(classes)),
        )

    def forward(self, x):
        return self.cnn.forward(x)
```

Do treningu użyjemy stochastycznego spadku po gradiencie (SGD), a jako funkcję straty Categorical Cross Entropy. W PyTorch-u funkcja ta operuje na indeksach klas (int), a nie na wektorach typu one-hot (jak w Tensorflow).

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

net = LeNet().to(device)

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

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.

```
In [ ]: net.train()
```

```
for epoch in range(5):
    for images, labels in trainloader:
        images, labels = images.to(device), labels.to(device)

        images_pred = net(images)

        loss = criterion(images_pred, labels)
        loss.backward()

        optimizer.step()
        optimizer.zero_grad()
```

Zobaczmy na kilku przykładach jak działa wytrenowana sieć.

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

grid_show(images)
print("Ground Truth")
print_grid(labels)

outputs = net(images.to(device))
_, predicted = torch.max(outputs, 1)

print()
print("Predicted")
print_grid(predicted)
```



Ground Truth						
Pullover	Shirt	Trouser	Sneaker	Ankle boot	Shirt	Coat
Sandal						
top	Coat	Bag	Pullover	top	Dress	Coat
Bag						
Pullover	Sneaker	Coat	Ankle boot	Dress	Pullover	Trouser
Ankle boot						
Pullover	Bag	Pullover	Sandal	Sandal	Ankle boot	Sneaker
Dress						
Predicted						
Pullover	Shirt	Trouser	Ankle boot	Ankle boot	Shirt	Coat
Sandal						
top	Coat	Bag	Coat	Dress	Dress	Coat
Bag						
Coat	Sneaker	Coat	Ankle boot	Dress	Shirt	Trouser
Ankle boot						
Shirt	Bag	Pullover	Sandal	Sandal	Ankle boot	Sneaker
Dress						

Obliczmy dokładności (accuracy) dla zbioru danych.

```
In [ ]: 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}%")
```

Accuracy of the network on the 10000 test images: 81 %

Komentarz

Dokładność przetrenowanego modelu jest bardzo przyzwoita, osiągneliśmy wynik na poziomie 85%. Widać, że najprawdopodobniej model nie radzi sobie dobrze z górną częścią odzieży: koszulkami, koszulkami bez ręków, swetrami i bluzkami.

Znając ogólny wynik klasyfikacji dla zbioru przeanalizujmy dokładniej, z którymi klasami jest największy problem.

Zadanie 3 (1 punkt)

Oblicz dokładność działania sieci (accuracy) dla każdej klasy z osobna. Podczas oceniania skuteczności modelu nie potrzebujemy, aby gradienty się liczyły. Możemy zatem zatrzymać obliczenia w bloku `with torch.no_grad()`:

```
In [ ]: correct = [0 for _ in classes]
total = [0 for _ in classes]

net.eval()
```

```

with torch.no_grad():
    for data in testloader:
        images, labels = data
        outputs = net(images.to(device))
        _, predicted = torch.max(outputs.data, 1)
        for i, label in enumerate(labels):
            total[label] += 1
            if predicted[i] == label:
                correct[label] += 1

class_name_max_len = max(map(lambda c: len(c), classes))
for i, cls in enumerate(classes):
    print(f"Accuracy of the network for class \'{cls}\':"
          f"{' ' * (class_name_max_len - len(cls) + 1)}" +
          f"\n{100 * correct[i] // total[i]}%")

```

```

Accuracy of the network for class "top":      79 %
Accuracy of the network for class "Trouser":   88 %
Accuracy of the network for class "Pullover":  61 %
Accuracy of the network for class "Dress":     80 %
Accuracy of the network for class "Coat":       71 %
Accuracy of the network for class "Sandal":    96 %
Accuracy of the network for class "Shirt":      63 %
Accuracy of the network for class "Sneaker":   80 %
Accuracy of the network for class "Bag":        96 %
Accuracy of the network for class "Ankle boot": 95 %

```

Komentarz

Powyższe spostrzeżenie się sprawdziło - model nie radzi sobie z górną częścią odzieży: koszulkami i swetrami, natomiast jednak jest dokładny przy klasyfikacji koszulek bez rękawów.

Detekcja obiektów

Problem detekcji polega na nie tylko sklasyfikowaniu obiektów na obrazie, ale również wyznaczeniu jego dokładnego położenia w postaci bounding-box-u. Choć jest to problem odmienny od klasyfikacji obrazów, to 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 listę bounding-box-ów i ich etykiety.

Zobaczmy jak działa detekcja na przykładzie już wytrenowanej sieci neuronowej. Autorzy skutecznych sieci często udostępniają ich wagę online, dzięki czemu jeżeli mamy doczynienia z analogicznym problemem jak ten, do którego dana sieć była przygotowana możemy z niej skorzystać "prosto z pudełka".

PyTorch pozwala nam na pobranie wytrenowanych wag dla kilku najpopularniejszych modeli. Sprawdzimy jak z tego skorzystać.

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

Poniżej znajduje się funkcja pozwalająca wczytać obraz z sieci. Przyda się do testowania działania sieci.

```
In [ ]: 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
```

Model, którym się zajmiemy to Faster R-CNN, który był trenowany na zbiorze COCO. Poniżej znajduje się lista klas (etykiet) dla tego zbioru danych.

```
In [ ]: 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))

```

Inizjalizacja modelu ResNet50-FPN wytrenowanymi wagami. Inicjalizujemy zarówno sieć backbone jak i RCNN.

```
In [ ]: model = detection.fasterrcnn_resnet50_fpn(  
    weights=detection.FasterRCNN_ResNet50_FPN_Weights.DEFAULT,  
    weights_backbone=torchvision.models.ResNet50_Weights.DEFAULT,  
    progress=True,  
    num_classes=len(classes)  
) .to(device)  
model.eval()
```

```
Out[ ]: 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), bia  
                        s=False)  
                    (bn1): FrozenBatchNorm2d(64, eps=0.0)  
                    (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), pad  
                        ding=(1, 1), bias=False)  
                    (bn2): FrozenBatchNorm2d(64, eps=0.0)  
                    (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bi  
                        as=False)  
                    (bn3): FrozenBatchNorm2d(256, eps=0.0)  
                    (relu): ReLU(inplace=True)  
                    (downsample): Sequential(  
                        (0): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bia  
                            s=False)  
                        (1): FrozenBatchNorm2d(256, eps=0.0)  
                    )  
                )  
                (1): Bottleneck(  
                    (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bi  
                        as=False)  
                    (bn1): FrozenBatchNorm2d(64, eps=0.0)  
                    (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), pad  
                        ding=(1, 1), bias=False)  
                    (bn2): FrozenBatchNorm2d(64, eps=0.0)  
                    (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bi  
                        as=False)  
                    (bn3): FrozenBatchNorm2d(256, eps=0.0)  
                    (relu): ReLU(inplace=True)  
                )  
                (2): Bottleneck(  
                    (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bi  
                        as=False)  
                    (bn1): FrozenBatchNorm2d(64, eps=0.0)  
                    (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), pad  
                        ding=(1, 1), bias=False)  
                    (bn2): FrozenBatchNorm2d(64, eps=0.0)  
                    (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bi  
                        as=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), b  
                        ias=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)
    )
)
```

```
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)
(downsampling): 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(
    (0-3): 4 x 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.

Do pobierania obrazów możemy się posłużyć wget-em.

```
In [ ]: # Pobieranie obrazka z sieci  
!wget https://upload.wikimedia.org/wikipedia/commons/thumb/7/7a/Toothbrus
```

```
--2023-12-04 23:19:56-- 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)... 185.15.59.240
Connecting to upload.wikimedia.org (upload.wikimedia.org)|185.15.59.240|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 338300 (330K) [image/jpeg]
Saving to: 'toothbrushes.jpg'
```

```
toothbrushes.jpg      100%[=====] 330,37K --.-KB/s    in 0,1s
```

```
2023-12-04 23:19:56 (2,62 MB/s) - 'toothbrushes.jpg' saved [338300/338300]
```

```
In [ ]: # Wyświetlanie obrazka
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

```
In [ ]: 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
```

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

Zadanie 4 (1 punkt)

Użyj modelu do wykrycia obiektów na obrazie. Następnie wybierz tylko te bounding boxy, dla których mamy wynik powyżej 50%. Wypisz 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

```
In [ ]: predicted_results = model([image_tensor.to(device)])[0]
predicted_results
```

```
Out[ ]: {'boxes': tensor([[ 271.9108,  585.5224, 1245.6252,  715.7845],
   [ 63.2615, 293.8127, 1204.4897, 395.0189],
   [ 602.6001, 456.1138, 1180.5505, 556.2330],
   [ 616.1780, 586.3022, 1191.9189, 689.5118],
   [ 166.2044, 459.9799, 1070.0201, 558.4254],
   [ 400.7661, 454.2596, 1198.2412, 557.9141],
   [ 155.4331, 462.4459, 933.0983, 561.9203],
   [ 170.1734, 459.9462, 1235.2539, 564.3173],
   [ 87.8447, 524.7531, 1232.4314, 731.7074],
   [ 96.9702, 509.5639, 1261.6742, 737.4170],
   [ 450.5747, 319.4155, 1035.3710, 388.5237],
   [ 141.2410, 460.8451, 1223.2361, 625.1956],
   [ 126.4219, 290.4581, 815.7283, 386.9609],
   [ 134.9710, 238.6473, 1192.7944, 451.7205],
   [ 59.2126, 592.4033, 1197.9298, 721.7292],
   [ 376.4023, 452.5932, 1205.3715, 562.6577],
   [ 160.3387, 459.0188, 1233.7699, 564.2238],
   [ 131.1700, 286.9063, 819.0837, 389.6934],
   [ 658.1013, 293.5099, 1152.4861, 395.8749],
   [ 117.5067, 231.7050, 1195.4241, 465.6245],
   [ 0.0000, 264.7669, 1280.0000, 939.8741],
   [ 131.6637, 287.6145, 813.5483, 386.2434],
   [ 111.2158, 362.9158, 934.2634, 700.9791],
   [ 72.6270, 589.2075, 1280.0000, 716.0181],
   [ 609.0575, 457.9680, 1182.7782, 553.0043],
   [ 547.4128, 588.2885, 1225.9056, 712.7369],
   [ 32.9076, 611.7830, 1032.1364, 723.1057]], device='mps:0',
   grad_fn=<StackBackward0>),
  'labels': tensor([90, 90, 50, 90, 49, 90, 84, 48, 87, 90, 90, 90, 39, 9
  0, 49, 87, 50, 84,
   90, 87, 67, 34, 90, 50, 48, 48, 90], device='mps:0'),
  'scores': tensor([0.9416, 0.8874, 0.6939, 0.6494, 0.3499, 0.3205, 0.226
  4, 0.1836, 0.1778,
   0.1697, 0.1678, 0.1678, 0.1580, 0.1510, 0.1367, 0.1363, 0.0977,
  0.0965,
   0.0923, 0.0881, 0.0854, 0.0819, 0.0704, 0.0703, 0.0698, 0.0663,
  0.0622], device='mps:0', grad_fn=<IndexBackward0>)}
```

```
In [ ]: above_acc_mask = predicted_results["scores"] > 0.5
predicted_results_filtered = {
    "boxes": predicted_results["boxes"] [above_acc_mask],
    "labels": predicted_results["labels"] [above_acc_mask],
    "scores": predicted_results["scores"] [above_acc_mask]
}
predicted_results_filtered

predicted_results_filtered_zipped = list(zip(
    predicted_results_filtered["boxes"],
    predicted_results_filtered["labels"],
    predicted_results_filtered["scores"]
))
for box, label, score in predicted_results_filtered_zipped:
    print(
        f"Class:\n" +
        f"\t{classes[label]}\n" +
        f"Probability:\n" +
        f"\t{score:.1%}\n" +
        f"Bounding box:\n" +
        f"\tUpper left ({box[0]}, {box[1]}),\n" +
```

```
        f"\tBottom right ({box[2]}, {box[3]})\n"
    )

from torchvision.utils import draw_bounding_boxes
from torchvision.transforms.v2.functional import to_pil_image
image_tensor_with_bboxes = to_pil_image(draw_bounding_boxes(
    image_tensor_int,
    boxes=predicted_results_filtered["boxes"],
    labels=list(map(
        lambda predicted: f"{classes[predicted[1]}}, {predicted[2]:.1%}",
        predicted_results_filtered_zipped
    )),
    width=3
))
display(image_tensor_with_bboxes)
```

Class:
 toothbrush
Probabilty:
 94.2%
Bounding box:
 Upper left (271.91082763671875, 585.5223999023438),
 Bottom right (1245.625244140625, 715.7844848632812)

Class:
 toothbrush
Probabilty:
 88.7%
Bounding box:
 Upper left (63.2614631652832, 293.81268310546875),
 Bottom right (1204.48974609375, 395.01885986328125)

Class:
 spoon
Probabilty:
 69.4%
Bounding box:
 Upper left (602.60009765625, 456.1138000488281),
 Bottom right (1180.550537109375, 556.2329711914062)

Class:
 toothbrush
Probabilty:
 64.9%
Bounding box:
 Upper left (616.177978515625, 586.3021850585938),
 Bottom right (1191.9189453125, 689.5118408203125)



Fine-tuning i pretrening

Trenowanie głębokich sieci neuronowych do przetwarzania obrazów jest zadaniem wymagającym bardzo dużych zbiorów danych i zasobów obliczeniowych. Często jednak, nie musimy trenować takich sieci od nowa, możemy wykorzystać wytrenowane modele i jedynie dostosowywać je do naszych problemów. Działanie takie nazywa się transfer learning-iem.

Przykładowo: mamy 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 obiektyami, które już zna (fine-tuning). Możemy usunąć ostatnią warstwę sieci i na jej miejsce wstawić nową, identyczną, jednak z losowo zainicjalizowanymi wagami, a następnie dotrenaować sieć na naszym nowym, bardziej specyficzny zbiorze danych. Przykładowo, jako bazę weźmiemy model wytrenowany na zbiorze ImageNet i będziemy chcieli użyć go do rozpoznawania nowych, nieznanych mu klas, np. ras psów.

Dla przećwiczenia takiego schematu działania 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.

```
In [ ]: # Download the hotdog dataset
!wget http://d2l-data.s3-accelerate.amazonaws.com/hotdog.zip
!unzip -n hotdog.zip
```

```
--2023-12-04 23:21:50-- http://d2l-data.s3-accelerate.amazonaws.com/hotdog.zip
Resolving d2l-data.s3-accelerate.amazonaws.com (d2l-data.s3-accelerate.amazonaws.com)... 18.244.103.202
Connecting to d2l-data.s3-accelerate.amazonaws.com (d2l-data.s3-accelerate.amazonaws.com)|18.244.103.202|: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  11,1MB/s   in 23s

2023-12-04 23:22:13 (10,8 MB/s) - 'hotdog.zip.1' saved [261292301/261292301]
```

Archive: hotdog.zip

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-channel RGB images of shape $(3 \times H \times W)$, where H and W are expected to be at least 224. The images have to be loaded in to a range of $[0, 1]$ and then normalized using mean = $[0.485, 0.456, 0.406]$ and std = $[0.229, 0.224, 0.225]$. 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])
```

```
In [ ]: 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,
    ]
)
```

```
    ]  
)
```

```
In [ ]: pretrained_net = torchvision.models.resnet18(weights=torchvision.models.R
```

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

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

Zadanie 5 (1 punkt)

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

```
In [ ]: from copy import deepcopy  
  
finetuned_net = deepcopy(pretrained_net)  
  
linear_layer = nn.Linear(512, 512)  
finetuned_net.fc = linear_layer  
  
# Weights are randomized from Kaiming's uniform distribution.
```

```
In [ ]: 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.float() / len(dataloaders[phase])

        print("{} Loss: {:.4f} Acc: {:.4f}".format(phase, epoch_loss,

        # 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
    
```

```

In [ ]: import os

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")

```

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()`.

```
In [ ]: def train_fine_tuning(net, learning_rate, num_epochs=15):

    last_parameters_names = ["fc.weight", "fc.bias"]
    trainer = torch.optim.SGD([
        {
            "params": map(
                lambda np: np[1],
                filter(
                    lambda np: np[0] not in last_parameters_names,
                    net.named_parameters()
                )
            )
        },
        {
            "params": map(
                lambda np: np[1],
                filter(
                    lambda np: np[0] in last_parameters_names,
                    net.named_parameters()
                )
            ),
            "lr": 10 * learning_rate
        }
    ], lr=learning_rate)

    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
```

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

Epoch 1/15

train Loss: 3.4466 Acc: 0.3930
val Loss: 1.4423 Acc: 0.5950

Epoch 2/15

train Loss: 1.1344 Acc: 0.6035
val Loss: 0.8211 Acc: 0.6913

Epoch 3/15

train Loss: 0.7836 Acc: 0.6860
val Loss: 0.6455 Acc: 0.7713

Epoch 4/15

train Loss: 0.6479 Acc: 0.7775
val Loss: 0.5528 Acc: 0.8163

Epoch 5/15

train Loss: 0.5629 Acc: 0.8055
val Loss: 0.4862 Acc: 0.8537

Epoch 6/15

train Loss: 0.5066 Acc: 0.8260
val Loss: 0.4405 Acc: 0.8675

Epoch 7/15

train Loss: 0.4741 Acc: 0.8380
val Loss: 0.4086 Acc: 0.8813

Epoch 8/15

train Loss: 0.4409 Acc: 0.8545
val Loss: 0.3941 Acc: 0.8875

Epoch 9/15

train Loss: 0.4305 Acc: 0.8500
val Loss: 0.3665 Acc: 0.8913

Epoch 10/15

train Loss: 0.4087 Acc: 0.8615
val Loss: 0.3422 Acc: 0.9025

Epoch 11/15

train Loss: 0.3877 Acc: 0.8660
val Loss: 0.3370 Acc: 0.9062

Epoch 12/15

train Loss: 0.3616 Acc: 0.8835
val Loss: 0.3215 Acc: 0.9100

```
Epoch 13/15
-----
train Loss: 0.3685 Acc: 0.8690
val Loss: 0.3014 Acc: 0.9112
```

```
Epoch 14/15
-----
train Loss: 0.3494 Acc: 0.8850
val Loss: 0.2985 Acc: 0.9125
```

```
Epoch 15/15
-----
train Loss: 0.3417 Acc: 0.8780
val Loss: 0.2950 Acc: 0.9112
```

Training complete in 5m 21s
Best val Acc: 0.912500

Komentarz

Otrzymaliśmy wynik na poziomie 89.8%, który jest bardzo dobry. Dzięki zastosowaniu transfer learning mogliśmy szybciej wytrenować nasz model, ponieważ skorzystaliśmy już z gotowego modelu, który uczył się na olbrzymiej bazie danych z wieloma różnymi klasami.

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)

```
In [ ]: 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)
```

```
In [ ]: 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 _, (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]]}', true:
                        imshow(inputs.data[j].cpu()))

        if images_so_far == num_images:
            return
```

In []: `visualize_model(model_ft)`

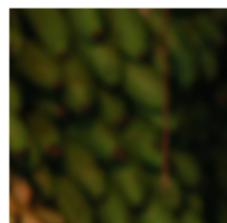
predicted: other, true: other



predicted: other, true: other



predicted: other, true: other



predicted: hotdog, true: hotdog



`predicted: other, true: other`



`predicted: hotdog, true: other`



Zadanie dla chętnych (3 punkty)

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.

In []: