**Splotowe sieci neuronowe** – CNN – do przetwarzania obrazów

**Torchvision.datasets.XXX** – do pobierania gotowych zbiorów danych, podajemy parametry jak :

**Train (bool)** – czy zbiór treningowy, jak false to testowy

**Download (bool)** – pobrać dane jeżeli nie ma, czy wczytać z lokalnego pliku

**Transform** – przekazujemy obiekt zawierający informacje jak wstępnie przetworzyć dane

**Torchvision.transforms.Compose** – tworzymy obiekt, podając listę transformacji jakie ma on wykonywać na danych, np. ToTensor, Normalize(parametry normalizacji, średnia i odchylenie standardowe)

**nn.Conv2d(in\_channels=3, out\_channels=16, kernel\_size=3, padding=1)** – tworzenie warstwy splotowej

**.unsqueeze(0)** – dodaje dodatkowy pierwszy wymiar aby dane były w poprawnym wymiarze kiedy przekazujemy tylko jedna obserwację

1. with torch.no\_grad():

2. conv.weight.fill\_(1 / 9)

3. conv.bias.zero\_()

4. lub

5. conv.weight[:] = torch.tensor([[-1.0, 0.0, 1.0],

6. [-1.0, 0.0, 1.0],

7. [-1.0, 0.0, 1.0]])

8.

**W ten sposób można ręcznie ustawić wagi filtra**

**img.permute(1, 2, 0))** – zmienia kolejność wymiarów

Wywołanie img.permute(1, 2, 0) zmienia kolejność wymiarów tensora z (C, H, W) na (H, W, C).

 C to liczba kanałów (np. 3 dla obrazów RGB),

 H to wysokość obrazu,

 W to szerokość obrazu.

To samo ale dla numpy zrobi **np.transpose(npimg, (1, 2, 0))**

**plt.imshow(output[0, 0].detach(), cmap='gray')** – pozwala wyświetlić jak wygląda obraz po przejściu przez wskazany filtr(warstwę)

**nn.MaxPool2d(kernel\_size=2, stride=2)** – warstwa MaxPool zmniejsza rozmiar obrazka, zachowując tylko najistotniejszy piksel z siatki 2 na 2

Definicja sieci neuronowej jako klasy

1. class Net(nn.Module):

2. def \_\_init\_\_(self):

3. super().\_\_init\_\_()

4. self.conv1 = nn.Conv2d(in\_channels=3, out\_channels=16, kernel\_size=3, padding=1)

5. self.act1 = nn.Tanh()

6. self.pool1 = nn.MaxPool2d(kernel\_size=2, stride=2)

7. self.conv2 = nn.Conv2d(in\_channels=16, out\_channels=8, kernel\_size=3, padding=1)

8. self.act2 = nn.Tanh()

9. self.pool2 = nn.MaxPool2d(kernel\_size=2, stride=2)

10. self.fc1 = nn.Linear(8 \* 8 \* 8, 32) – to mnożenie wynika z tego jakie mamy wyjście na ostatniej warstwie splotowej

11. self.act3 = nn.Tanh()

12. self.fc2 = nn.Linear(32, 2)

13. self.soft = nn.Softmax(dim=1)

14.

15. def forward(self, x):

16. out = self.conv1(x)

17. out = self.act1(out)

18. out = self.pool1(out)

19. out = self.conv2(out)

20. out = self.act2(out)

21. out = self.pool2(out)

22. out = out.view(-1, 8 \* 8 \* 8)

23. out = self.fc1(out)

24. out = self.act3(out)

25. out = self.fc2(out)

26. out = self.soft(out)

27. return out

28.

Definicja z wykorzystaniem **nn.functional,** aby nie definiować funkcji aktywacji itp. Jako struktury sieci

1. class Net(nn.Module):

2. def \_\_init\_\_(self):

3. super().\_\_init\_\_()

4. self.conv1 = nn.Conv2d(in\_channels=3, out\_channels=16, kernel\_size=3, padding=1)

5. self.conv2 = nn.Conv2d(in\_channels=16, out\_channels=8, kernel\_size=3, padding=1)

6. self.fc1 = nn.Linear(8 \* 8 \* 8, 32)

7. self.fc2 = nn.Linear(32, 2)

8. def forward(self, x):

9. out = F.tanh(self.conv1(x))

10. out = F.max\_pool2d(out, kernel\_size=2, stride=2)

11. out = F.tanh(self.conv2(out))

12. out = F.max\_pool2d(out, kernel\_size=2, stride=2)

13. out = out.view(-1, 8 \* 8 \* 8)

14. out = F.tanh(self.fc1(out))

15. out = F.softmax(self.fc2(out), dim=1)

16. return out

17.

Możliwość zapisywania i odczytywania wag modelu (same wagi, bez struktury)

1. torch.save(model.state\_dict(), "data/birds\_vs\_airplanes.pt")

2. loaded\_model = Net()

3. loaded\_model.load\_state\_dict(torch.load("data/birds\_vs\_airplanes.pt"))

4.

Sprawdzenie czy urządzenie z cuda jest dostępne, aby przenieść i trenować model na karcie graficznej

1. device = (torch.device('cuda') if torch.cuda.is\_available() else torch.device('cpu'))

2.

W takiej sytuacji należy pomiatać o stosowaniu **.to** dla modelu i danych, aby znajdowały się na tym samym urządzeniu:

1. def training\_loop(n\_epochs, optimizer, model, loss\_fn, train\_loader):

2. model.train()

3. for epoch in range(1, n\_epochs + 1):

4. loss\_train = 0.0

5. for imgs, labels in train\_loader:

6. imgs = imgs.to(device)

7. labels = labels.to(device)

8.

9. optimizer.zero\_grad()

10. outputs = model(imgs)

11. loss = loss\_fn(outputs, labels.long())

12. loss.backward()

13. optimizer.step()

14. loss\_train += loss.item()

15. if epoch == 1 or epoch % 10 == 0:

16. print(f"Epoch {epoch}, Training loss {loss\_train / len(train\_loader)}")

17. model\_gpu = Net().to(device)

18. loss\_fn = nn.CrossEntropyLoss()

19. optimizer = optim.SGD(model\_gpu.parameters(), lr=0.01)

20. train\_loader = DataLoader(cifar2, batch\_size=64, shuffle=True)

21.

22. training\_loop(n\_epochs=100, optimizer=optimizer, model=model\_gpu, loss\_fn=loss\_fn, train\_loader=train\_loader)

23.

**Rozbudowa modelu:** dla sieci splotowych model możemy rozbudowywać na szerokość, zwiększając liczbę filtrów (neuronów) w danej warstwie, albo na głębokość dodając kolejną warstwę

**Regularyzacja:**

**L2** – polega na dodaniu kary dla modelu za zbyt duże wagi

zaimplementowana ręcznie:

1. l2\_lambda = 0.001

2. for param in model.parameters():

3. loss += l2\_lambda \* torch.norm(param)\*\*2

4.

Automatycznie w optimizer, podajemy parametr **weight\_decey**:

1. optimizer = torch.optim.SGD(model.parameters(), lr=1e-2, weight\_decay=0.001)

2.

**Dropout** - polega na losowym odrzuceniu części wag przy przejściu do kolejnej warstwy, implementuje się to jako rodzaj warstwy z argumentem jaką część wag mamy odrzucić:

1. class NetDropout(nn.Module):

2. def \_\_init\_\_(self, n\_chans1=32):

3. super().\_\_init\_\_()

4. self.n\_chans = n\_chans1

5. self.conv1 = nn.Conv2d(3, n\_chans1, kernel\_size=3, padding=1)

6. self.conv2 = nn.Conv2d(n\_chans1, n\_chans1, kernel\_size=3, padding=1)

7. self.fc1 = nn.Linear(8 \* 8 \* n\_chans1, 32)

8. self.fc2 = nn.Linear(32, 2)

9. self.dropout = nn.Dropout2d(p=0.5)

10.

11. def forward(self, x):

12. out = nn.functional.max\_pool2d(torch.relu(self.conv1(x)), 2)

13. out = self.dropout(out)

14. out = nn.functional.max\_pool2d(torch.relu(self.conv2(out)), 2)

15. out = self.dropout(out)

16. out = out.view(-1, 8 \* 8 \* self.n\_chans)

17. out = torch.relu(self.fc1(out))

18. out = self.fc2(out)

19. return out

20.

**Batch** **normalization** – polega na znormalizowaniu danych przy przejściu miedzy warstwami:

1. class NetBatchNorm(nn.Module):

2. def \_\_init\_\_(self, n\_chans1=32):

3. super().\_\_init\_\_()

4. self.n\_chans = n\_chans1

5. self.conv1 = nn.Conv2d(3, n\_chans1, kernel\_size=3, padding=1)

6. self.bn1 = nn.BatchNorm2d(n\_chans1)

7. self.conv2 = nn.Conv2d(n\_chans1, n\_chans1, kernel\_size=3, padding=1)

8. self.bn2 = nn.BatchNorm2d(n\_chans1)

9. self.fc1 = nn.Linear(8 \* 8 \* n\_chans1, 32)

10. self.fc2 = nn.Linear(32, 2)

11.

12. def forward(self, x):

13. out = nn.functional.max\_pool2d(torch.relu(self.bn1(self.conv1(x))), 2)

14. out = nn.functional.max\_pool2d(torch.relu(self.bn2(self.conv2(out))), 2)

15. out = out.view(-1, 8 \* 8 \* self.n\_chans)

16. out = torch.relu(self.fc1(out))

17. out = self.fc2(out)

18. return out

19.

**Transfer** **learning** – czyli przenoszenie wstępnie wytrenowanego modelu splotowego, zdolnego wykrywać ogólne elementy szczególne w obrazie jak krawędzie i bardzie złożone struktury, faza nazywane „**pre-treningiem**”, a następnie wykorzystanie już wytrenowanych warstw spol9towych do innego problemu, gdzie uczenie następuje w warstwach gęstych na końcu sieci „**fine-tuning**”

**dir(models)** – lista dostępnych modeli w trochvision

**resnet18\_model** = models.resnet18(pretrained=True) – pobranie modelu, pretrained, czy model ma być wytrenowany, czy z wagami losowymi

**ImageFolder** – ładowanie danych z folderów, gdzie folder oznacza klasę dla jego zawartości:

1. train\_dataset = datasets.ImageFolder(DATA\_PATH / 'train')

2. val\_dataset = datasets.ImageFolder(DATA\_PATH / 'val')

3.

Domyślnie wczytuje obrazy jako PIL.Image, dlatego zmieniamy na tensory:

1. data\_transforms = transforms.Compose([

2. transforms.ToTensor()

3. ])

4.

5. train\_dataset = datasets.ImageFolder(DATA\_PATH / 'train', transform=data\_transforms)

6. val\_dataset = datasets.ImageFolder(DATA\_PATH / 'val', transform=data\_transforms)

7.

Dla treningu warto zastosować augmentację obrazków, poprzez zastosowanie losowych modyfikacji, pozostawiając jednak obraz możliwy do właściwej interpretacji (nie może być zmodyfikowany tak bardzo że nie wiadomo czym on był):

1. train\_transforms = transforms.Compose([

2. transforms.RandomResizedCrop(IMAGENET\_IMG\_SIZE),

3. transforms.RandomHorizontalFlip(),

4. transforms.CenterCrop(IMAGENET\_IMG\_SIZE),

5. transforms.ToTensor(),

6. transforms.Normalize(mean=IMAGENET\_MEANS, std=IMAGENET\_STD)

7. ])

8.

Podmiana tylko ostatnich warstw gęstych w pre-trenowanym modelu splotowym:

1. model = models.resnet18(pretrained=True)

2. num\_classes = 2

3. num\_ftrs = model.fc.in\_features

4. model.fc = nn.Linear(num\_ftrs, num\_classes)

5. model = model.to(device)

6.

W takim przypadku podczas treningu nadal dostosowywane są również warstwy splotowe, co wydłuża czas uczenia, a nie koniecznie przynosi pozytywny rezultat, dlatego możemy zamrozić wagi dla warstw splotowych:

1. model\_frozen = models.resnet18(pretrained=True)

2.

3. for param in model\_frozen.parameters():

4. param.requires\_grad = False

5.

6. num\_classes = 2

7. num\_ftrs = model\_frozen.fc.in\_features

8. model\_frozen.fc = nn.Linear(num\_ftrs, num\_classes)

9.

10. model\_frozen = model\_frozen.to(device)

11.

**Wizualizacja obrazka:**

1. def matplotlib\_imshow(img, one\_channel=False):

2. if one\_channel:

3. img = img.mean(dim=0)

4. img = img / 2 + 0.5 # unnormalize

5. npimg = img.numpy()

6. if one\_channel:

7. plt.imshow(npimg, cmap="Greys")

8. else:

9. plt.imshow(np.transpose(npimg, (1, 2, 0)))

10.

**Tensorboard**:

służy do logowania procesu uczenia modelu, poprzez tworzenie plików (domyślnie w folderze runs) zgodnie ze wskazaną funkcją, które później mogą zostać odczytane w przeglądarce za pomocą serwera tensorboard

Uruchomienie serwera: **tensorboard --logdir=ŚCIEŻKA\_DO\_KATALOGU**

Domyślnie port 6060

Należy w kodzie utworzyć obiekt **SummaryWriter**() i wywołać na nim interesująca nas funkcję podając dane do zapisania w logach np.:

1. writer = SummaryWriter()

2.

3. images, labels = next(iter(train\_loader))

4. img\_grid = torchvision.utils.make\_grid(images)

5.

6. writer.add\_image('images', img\_grid)

7. writer.close()

8.

1. images, labels = next(iter(train\_loader))

2. writer.add\_graph(model, images)

3.

1. writer.add\_figure('predictions vs. actuals',

2. plot\_classes\_preds(model, inputs, labels),

3. global\_step=i+1)

4.

Skomplikowany przykład dla krzywej precision-recall

1. class\_probs = []

2. class\_label = []

3. with torch.no\_grad():

4. for data in val\_loader:

5. images, labels = data

6. output = model(images)

7. class\_probs\_batch = [F.softmax(el, dim=0) for el in output]

8.

9. class\_probs.append(class\_probs\_batch)

10. class\_label.append(labels)

11.

12. test\_probs = torch.cat([torch.stack(batch) for batch in class\_probs])

13. test\_label = torch.cat(class\_label)

14.

15. # helper function

16. def add\_pr\_curve\_tensorboard(class\_index, test\_probs, test\_label, global\_step=0):

17. '''

18. Takes in a "class\_index" from 0 to 9 and plots the corresponding

19. precision-recall curve

20. '''

21. tensorboard\_truth = test\_label == class\_index

22. tensorboard\_probs = test\_probs[:, class\_index]

23.

24. writer = SummaryWriter()

25. writer.add\_pr\_curve(classes[class\_index],

26. tensorboard\_truth,

27. tensorboard\_probs,

28. global\_step=global\_step)

29. writer.close()

30.

31. # plot all the pr curves

32. for i in range(len(classes)):

33. add\_pr\_curve\_tensorboard(i, test\_probs, test\_label)

34.