Convolutional networks: from one filter to ResNets, detection and transfer learning

Convolution by hand and in PyTorch, output sizes, pooling, a CNN against a plain network, a residual block, depthwise separable convolutions, augmentation with torchvision.transforms.v2, IoU and non-max suppression, transfer learning with torchvision's pre-trained weights, triplet loss and the style (Gram) matrix.

Open in Google Colab · part of the inPace course library. Slides © DeepLearning.AI (CC BY-SA 2.0); this notebook: inPace (CC BY-SA 4.0).

Run it top to bottom (Runtime → Run all in Colab). Each part says what to look for.

```
# Works in Google Colab as it is (PyTorch is already installed there).
# On your own computer:  pip install torch torchvision scikit-learn scipy
matplotlib pandas
# One framework per notebook: PyTorch and TensorFlow can clash when loaded into
the same Python process.
import numpy as np, matplotlib.pyplot as plt, torch
from torch import nn
torch.manual_seed(0); np.random.seed(0)
device = "cuda" if torch.cuda.is_available() else "mps" if
torch.backends.mps.is_available() else "cpu"
print("PyTorch", torch.__version__, "| device:", device)
```

```
PyTorch 2.15.0+cu132 | device: cpu
```

1. Convolution by hand, then with PyTorch

A vertical-edge filter on a 6×6 image (the slide's example). PyTorch's `conv2d` computes the same thing (deep learning's 'convolution' is cross-correlation: no flipping).

```
img = np.array([[10, 10, 10, 0, 0, 0]] * 6, dtype=np.float32)
k = np.array([[1, 0, -1]] * 3, dtype=np.float32)
by_hand = np.array([[(img[i:i+3, j:j+3] * k).sum() for j in range(4)] for i in
range(4)])
torch_out = nn.functional.conv2d(torch.tensor(img)[None, None], torch.tensor(k)
[None, None])[0, 0].numpy()
print(by_hand); print("same as PyTorch:", np.allclose(by_hand, torch_out))
```

```
[[ 0. 30. 30.  0.]
 [ 0. 30. 30.  0.]
 [ 0. 30. 30.  0.]
 [ 0. 30. 30.  0.]]
same as PyTorch: True
```

2. Output size: ⌊(n + 2p − f) / s⌋ + 1

```
def out_size(n, f, p=0, s=1): return (n + 2 * p - f) // s + 1
x = torch.zeros(1, 3, 32, 32)
for f, p, s in ((3, 0, 1), (3, 1, 1), (5, 2, 1), (3, 1, 2), (7, 3, 2)):
```

```
    y = nn.Conv2d(3, 8, f, stride=s, padding=p)(x)
    print(f"f={f} p={p} s={s}: formula {out_size(32, f, p, s)}, PyTorch
{y.shape[-1]}")
print("max pool 2×2, stride 2:", nn.MaxPool2d(2)(x).shape)
```

```
f=3 p=0 s=1: formula 30, PyTorch 30
f=3 p=1 s=1: formula 32, PyTorch 32
f=5 p=2 s=1: formula 32, PyTorch 32
f=3 p=1 s=2: formula 16, PyTorch 16
f=7 p=3 s=2: formula 16, PyTorch 16
max pool 2×2, stride 2: torch.Size([1, 3, 16, 16])
```

3. A CNN against a plain network

Digits as 1×8×8 images. Compare accuracy and **number of parameters**.

```
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
d = load_digits()
Xa, Xb, ya, yb = train_test_split(d.images / 16.0, d.target, test_size=0.3,
random_state=0)
Xa, Xb = torch.tensor(Xa, dtype=torch.float32)[:, None], torch.tensor(Xb,
dtype=torch.float32)[:, None]
ya, yb = torch.tensor(ya), torch.tensor(yb)
def fit(m, epochs=60, X=Xa, y=ya):
    opt = torch.optim.AdamW(m.parameters(), lr=2e-3)
    dl = torch.utils.data.DataLoader(torch.utils.data.TensorDataset(X, y),
batch_size=64, shuffle=True)
    for _ in range(epochs):
        m.train()
        for xb, yb_ in dl: opt.zero_grad(); nn.functional.cross_entropy(m(xb),
yb_).backward(); opt.step()
    m.eval()
    with torch.no_grad(): return (m(Xb).argmax(1) == yb).float().mean().item()
count = lambda m: sum(p.numel() for p in m.parameters())
torch.manual_seed(0)
plain = nn.Sequential(nn.Flatten(), nn.Linear(64, 128), nn.ReLU(), nn.Linear(128,
10))
cnn = nn.Sequential(nn.Conv2d(1, 16, 3, padding=1), nn.ReLU(), nn.MaxPool2d(2),
# 16×4×4
                    nn.Conv2d(16, 32, 3, padding=1), nn.ReLU(), nn.MaxPool2d(2),
# 32×2×2
                    nn.Flatten(), nn.Linear(128, 10))
print(f"plain network: {count(plain):,} parameters, test acc {fit(plain):.3f}")
print(f"CNN:           {count(cnn):,} parameters, test acc {fit(cnn):.3f}")
```

```
plain network: 9,610 parameters, test acc 0.976
CNN:           6,090 parameters, test acc 0.978
```

5. A residual block

`a[l+2] = g(z[l+2] + a[l])` : the shortcut lets the block learn 'do nothing' easily, so very deep networks still train.

```
class ResidualBlock(nn.Module):
    def __init__(self, c):
        super().__init__()
```

```
        self.body = nn.Sequential(nn.Conv2d(c, c, 3, padding=1, bias=False),
nn.BatchNorm2d(c), nn.ReLU(),
                                  nn.Conv2d(c, c, 3, padding=1, bias=False),
nn.BatchNorm2d(c))
    def forward(self, x):
        return torch.relu(self.body(x) + x)          # the skip connection
x = torch.randn(2, 16, 8, 8)
print("shape kept:", ResidualBlock(16)(x).shape)
```

```
shape kept: torch.Size([2, 16, 8, 8])
```

6. Depthwise separable convolution (MobileNet)

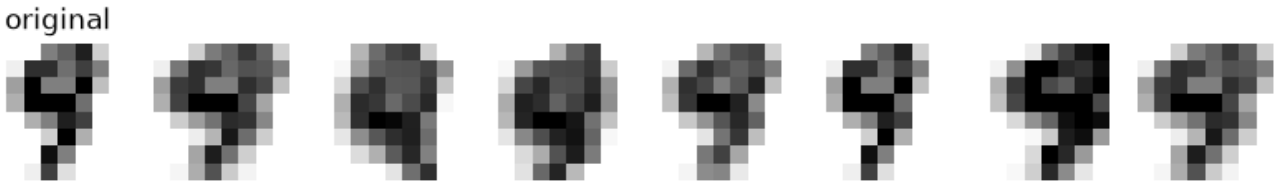
The same input and output shapes, far fewer weights and multiplications.

```
normal = nn.Conv2d(64, 128, 3, padding=1)
separable = nn.Sequential(nn.Conv2d(64, 64, 3, padding=1, groups=64),   #
depthwise: one filter per channel
                          nn.Conv2d(64, 128, 1))                        #
pointwise: 1×1 mixes channels
x = torch.randn(1, 64, 32, 32)
print(normal(x).shape, separable(x).shape)
print(f"weights: normal {count(normal):,} vs separable {count(separable):,}
({count(normal) / count(separable):.1f}× fewer)")
```

```
torch.Size([1, 128, 32, 32]) torch.Size([1, 128, 32, 32])
weights: normal 73,856 vs separable 8,960 (8.2× fewer)
```

7. Data augmentation with torchvision.transforms.v2

```
import torchvision
from torchvision.transforms import v2
aug = v2.Compose([v2.RandomResizedCrop(8, scale=(0.7, 1.0), antialias=True),
v2.RandomRotation(12),
                  v2.ColorJitter(brightness=0.3)])
fig, ax = plt.subplots(1, 8, figsize=(10, 1.5))
for i, a in enumerate(ax):
    a.imshow((Xa[0] if i == 0 else aug(Xa[0]))[0], cmap="gray_r"); a.axis("off");
a.set_title("original" if i == 0 else "")
plt.show()
print("torchvision", torchvision.__version__)
```




```
torchvision 0.29.2+cu132
```

8. IoU and non-max suppression

Our own versions, checked against `torchvision.ops`.

```
from torchvision.ops import box_iou, nms
def iou(a, b):
```

```
    x1, y1 = max(a[0], b[0]), max(a[1], b[1]); x2, y2 = min(a[2], b[2]), min(a[3],
b[3])
    inter = max(0, x2 - x1) * max(0, y2 - y1)
    return inter / ((a[2] - a[0]) * (a[3] - a[1]) + (b[2] - b[0]) * (b[3] - b[1])
- inter)
def my_nms(boxes, scores, thr=0.5):
    order, keep = list(np.argsort(-scores)), []
    while order:
        i = order.pop(0); keep.append(int(i))
        order = [j for j in order if iou(boxes[i], boxes[j]) <= thr]
    return keep
boxes = np.array([[30, 30, 130, 120], [40, 38, 140, 128], [35, 25, 125, 125],
[200, 60, 290, 160], [210, 70, 300, 165]], float)
scores = np.array([0.9, 0.75, 0.6, 0.85, 0.7])
print("IoU:", round(iou(boxes[0], boxes[1]), 3), "torchvision:",
round(box_iou(torch.tensor(boxes[:1]), torch.tensor(boxes[1:2])).item(), 3))
print("NMS keeps:", my_nms(boxes, scores), "torchvision:",
nms(torch.tensor(boxes), torch.tensor(scores), 0.5).tolist())
```

```
IoU: 0.695 torchvision: 0.695
NMS keeps: [0, 3] torchvision: [0, 3]
```

9. Transfer learning with pre-trained weights

The modern torchvision API: `weights=...DEFAULT` and the matching `transforms()`. Freeze the backbone, replace the last layer. (Downloads about 45 MB, so it needs internet: it runs in Colab.)

```
from torchvision.models import resnet18, ResNet18_Weights
try:
    weights = ResNet18_Weights.DEFAULT
    net = resnet18(weights=weights)
    for p in net.parameters(): p.requires_grad = False
    net.fc = nn.Linear(net.fc.in_features, 3)            # your 3 classes; only
this layer trains
    print("trainable parameters:", sum(p.numel() for p in net.parameters() if
p.requires_grad))
    print("preprocessing:", weights.transforms())
except Exception as e:
    print("Needs internet to download the weights (run this in Colab):",
type(e).__name__)
```

```
Downloading: "https://download.pytorch.org/models/resnet18-f37072fd.pth" to /root/.
cache/torch/hub/checkpoints/resnet18-f37072fd.pth
Needs internet to download the weights (run this in Colab): URLError
```

10. Triplet loss and the style matrix

```
def triplet_loss(a, p, n, margin=0.2):
    return torch.relu((a - p).pow(2).sum(1) - (a - n).pow(2).sum(1) +
margin).mean()
a = nn.functional.normalize(torch.randn(4, 128), dim=1)          # embeddings on
the unit sphere, like FaceNet
pos = nn.functional.normalize(a + 0.3 * torch.randn_like(a), dim=1)  # same
person: close
neg = nn.functional.normalize(torch.randn(4, 128), dim=1)            # someone
else: far
```

```
hard = nn.functional.normalize(a + 0.35 * torch.randn_like(a), dim=1)  # a "hard"
negative: almost as close
print("easy negatives:", round(triplet_loss(a, pos, neg).item(), 3), "| hard
negatives:", round(triplet_loss(a, pos, hard).item(), 3))
print("PyTorch's built-in (plain distances):",
round(nn.TripletMarginLoss(margin=0.2)(a, pos, hard).item(), 3))
def gram(features):                    # features: (channels, height, width)
    f = features.flatten(1)
    return f @ f.T / f.shape[1]          # which channels fire together = the
"style"
print("style matrix shape for 64 channels:", gram(torch.randn(64, 14, 14)).shape)
```

```
easy negatives: 0.0 | hard negatives: 0.188
PyTorch's built-in (plain distances): 0.179
style matrix shape for 64 channels: torch.Size([64, 64])
```

Try this

1. Remove the pooling layers in section 3 (and fix the `Linear` size). More parameters, better accuracy?
2. Train the CNN on 20% of the training data, with and without the augmentation from section 7.
3. In section 8, change the NMS threshold to 0.3 and 0.7. Which boxes survive?
4. In Colab, fine-tune section 9's ResNet on a small image folder of your own (`torchvision.datasets.ImageFolder`).