Schedule

Block 3 (1:30 - 3:00 pm)

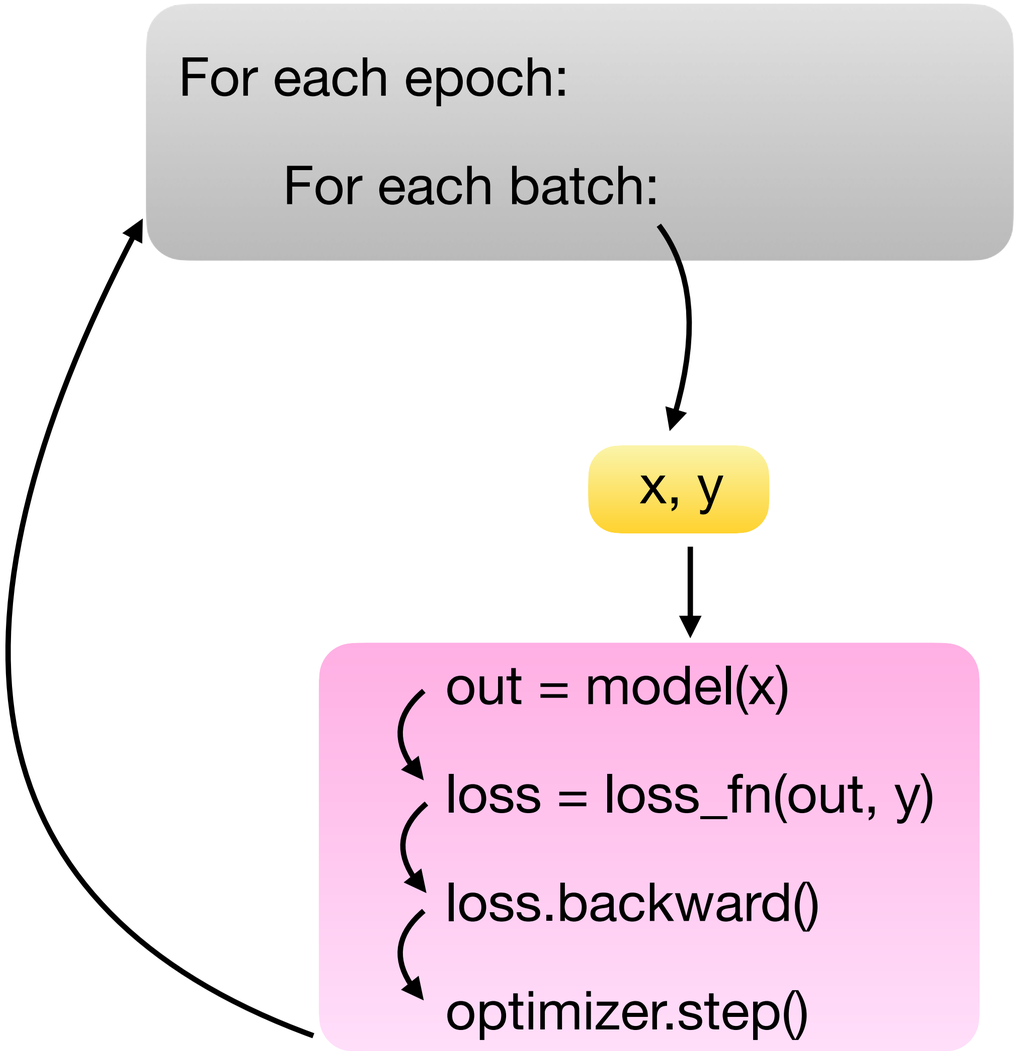
(5) Introduction to Deep Learning

(6) Understanding PyTorch

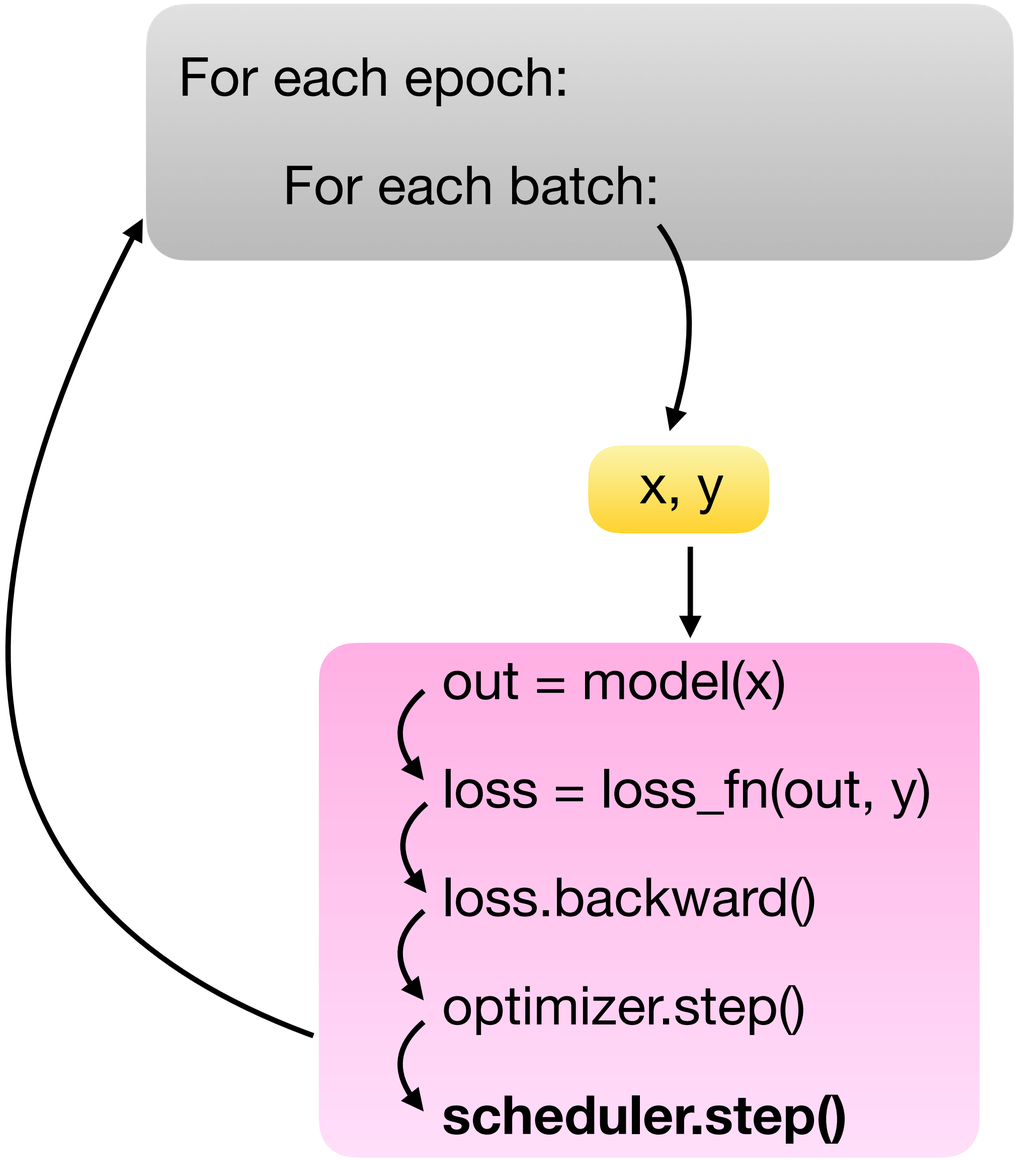
(7) Training Deep Neural Networks

30 min break (3:00 - 3:30 pm)

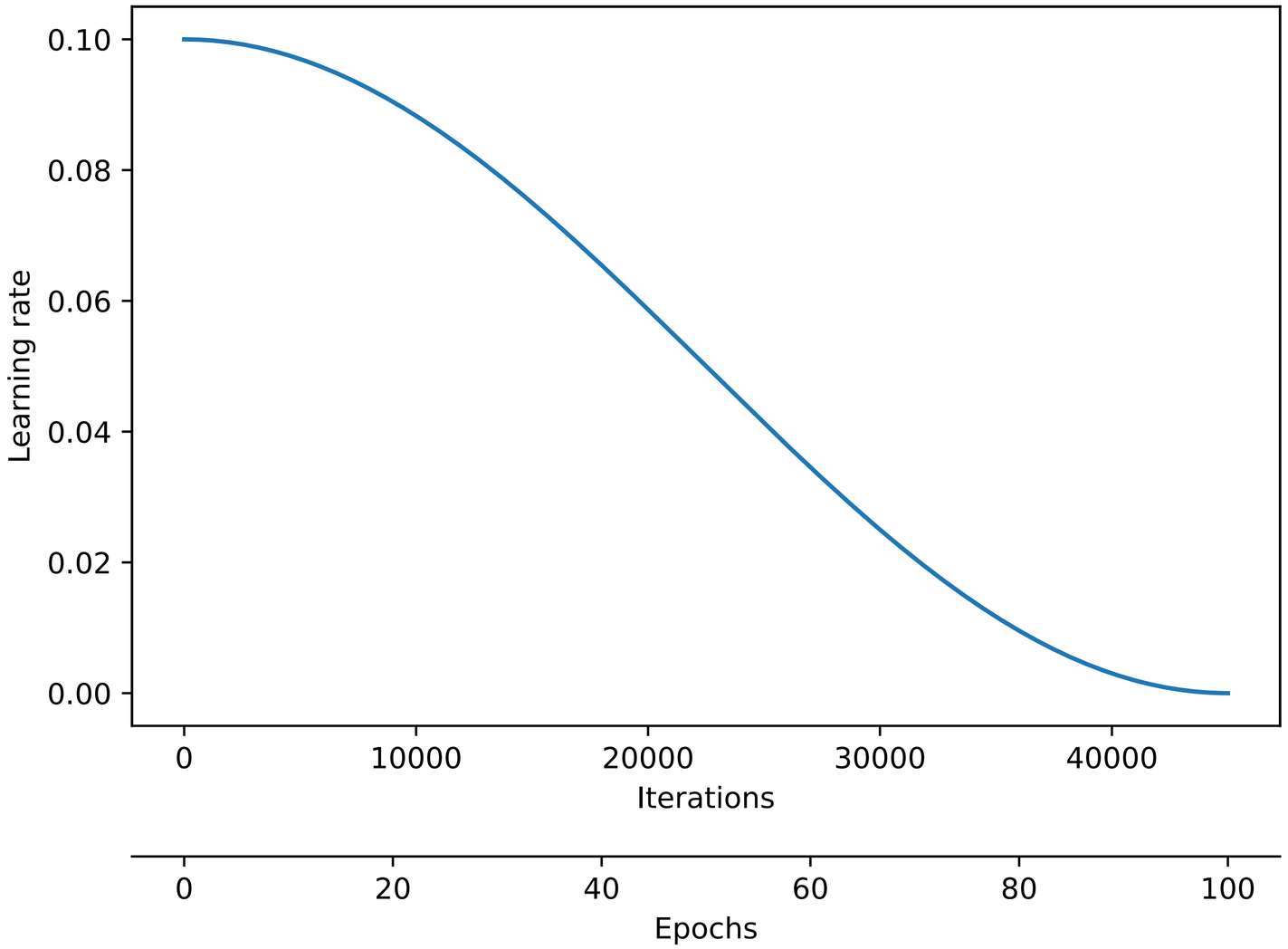
Recap


For each epoch:
For each batch:
x, y
out = model(x)
loss = loss_fn(out, y)
loss.backward()
optimizer.step()

Adding a scheduler



```
optimizer = torch.optim.SGD(model.parameters(), lr=0.1)


num_steps = num_epochs * len(train_dataloader())
scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=num_steps)

    for batch_idx, (features, targets) in enumerate(train_loader):
        model.train()

        features = features.to(device)
        targets = targets.to(device)

        ### FORWARD AND BACK PROP
        logits = model(features)
        loss = F.cross_entropy(logits, targets)

        optimizer.zero_grad()
        loss.backward()

        ### UPDATE MODEL PARAMETERS
        optimizer.step()
        scheduler.step()
```


0.10
0.08
0.06
0.04
0.02
0.00
Learning rate
0
10000
20000
30000
40000
Iterations
0
20
40
60
80
100
Epochs

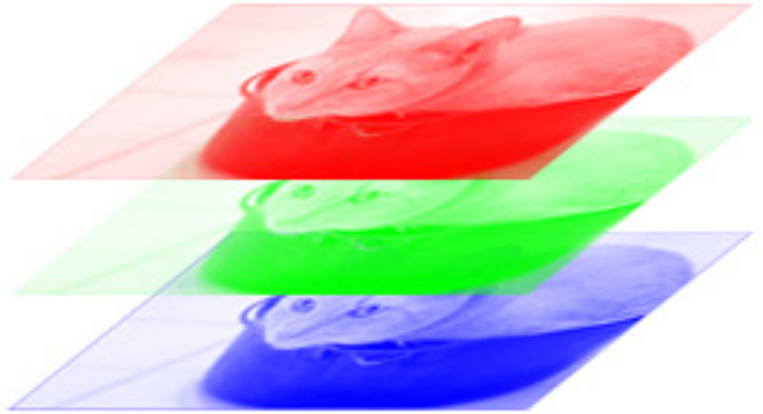
From tabular data to image data

RGB Image

R

G

B

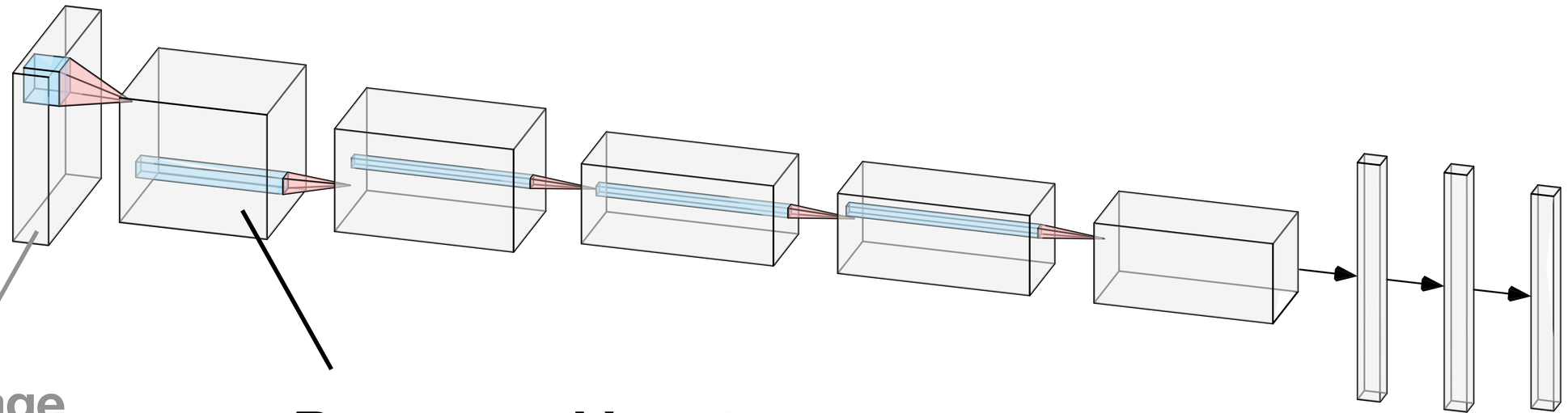


=

Image Source: https://code.tutsplus.com/tutorials/create-a-retro-crt-distortion-effect-using-rgb-shifting--active-3359

Color image as a stack of matrices

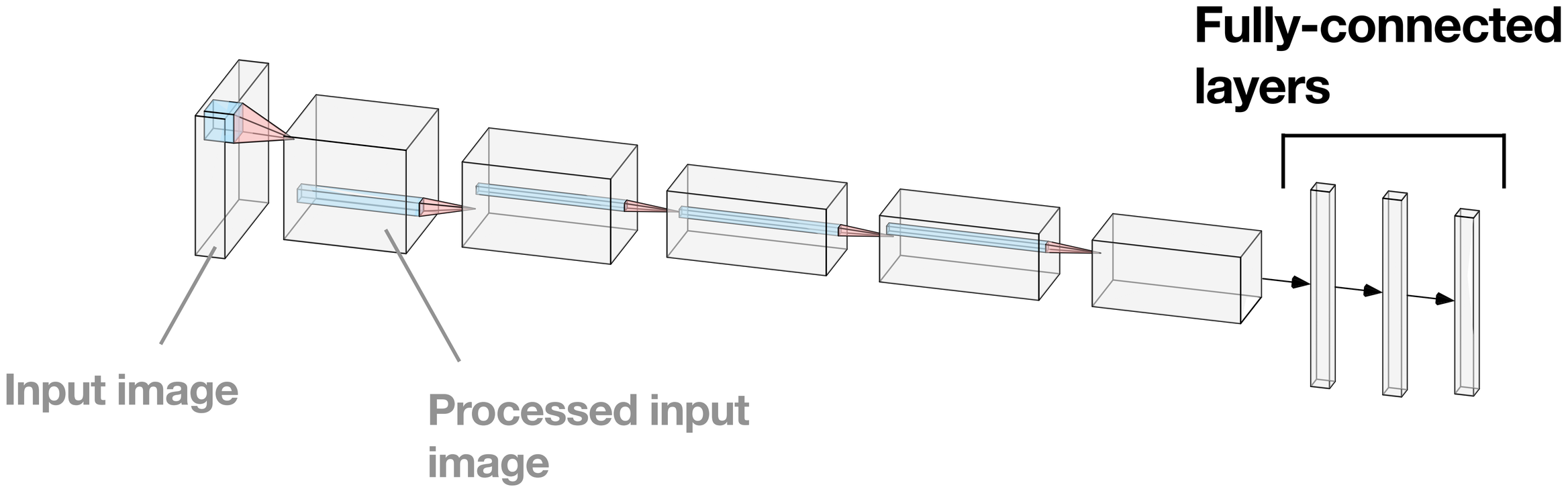
A typical convolutional neural network architecture



Input image

Processed input image

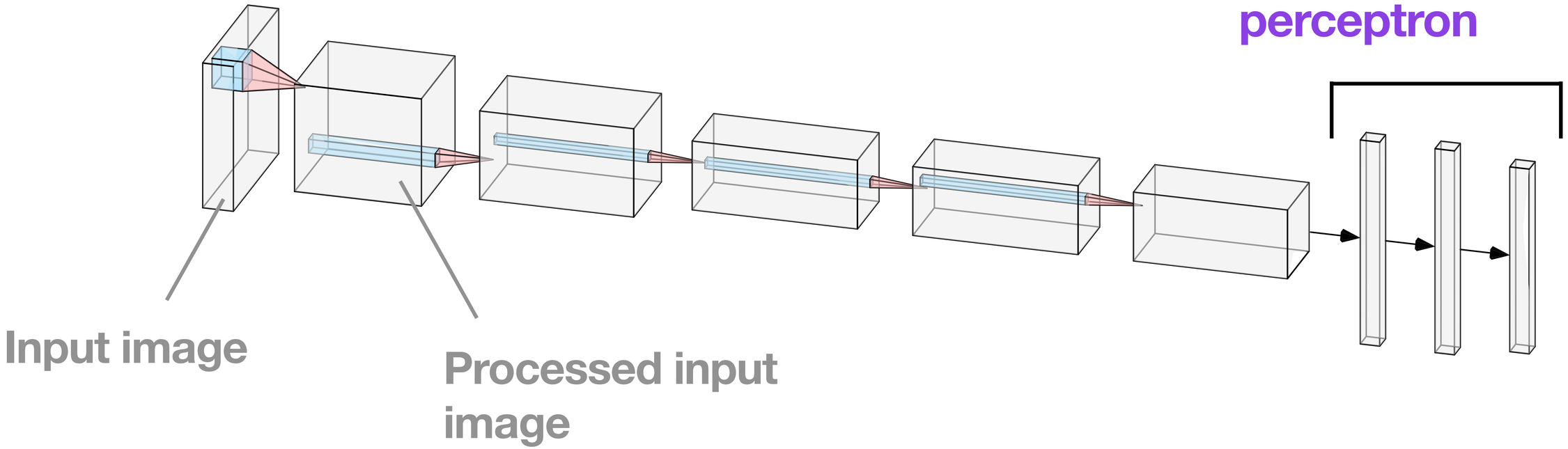
A typical convolutional neural network architecture



A typical convolutional neural network architecture

Fully-connected layers

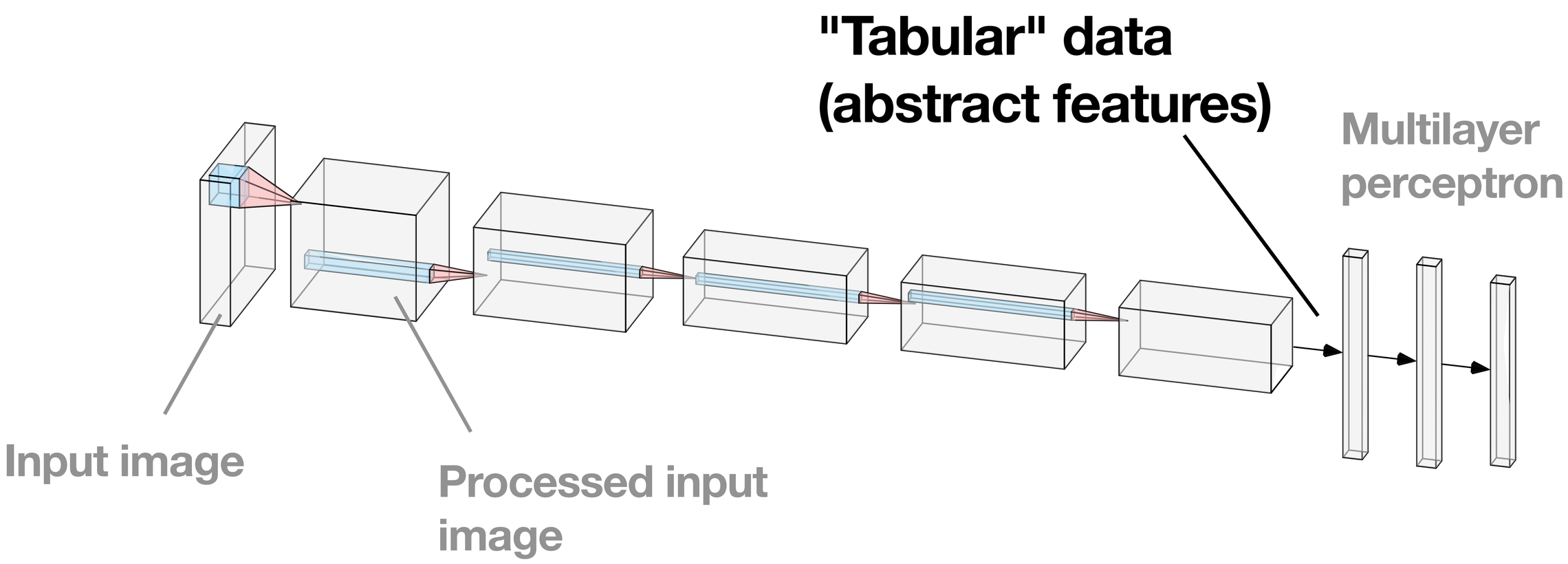
→ **Multilayer perceptron**



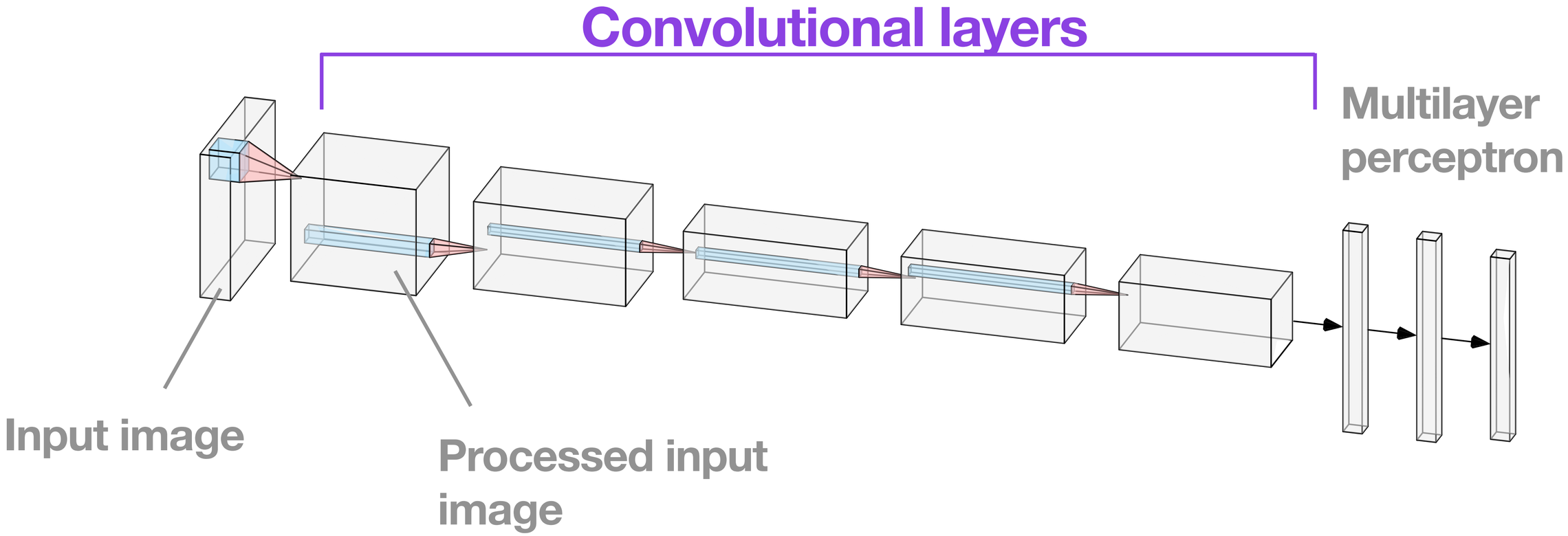
Input image

Processed input image

A typical convolutional neural network architecture

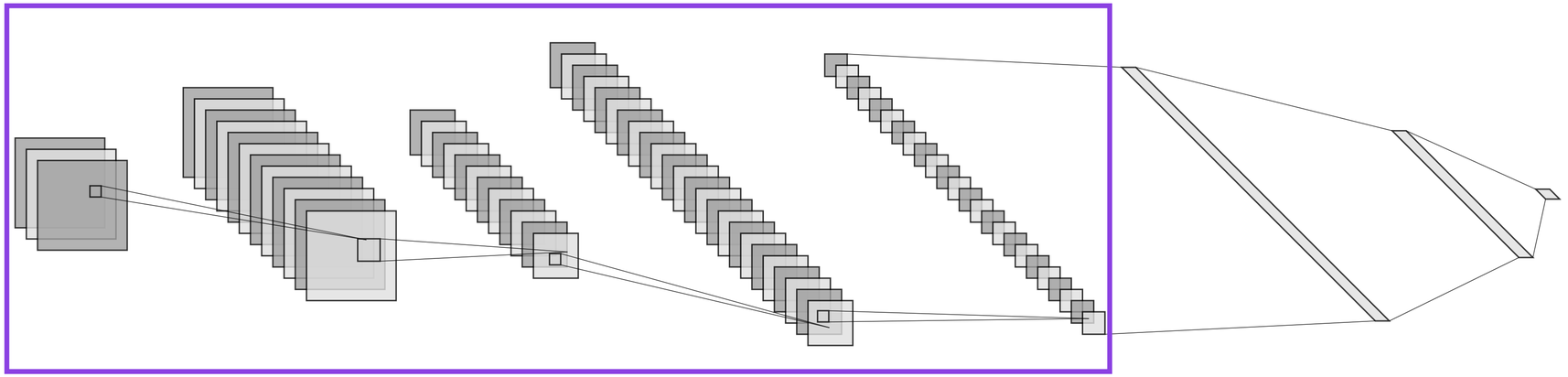


A typical convolutional neural network architecture



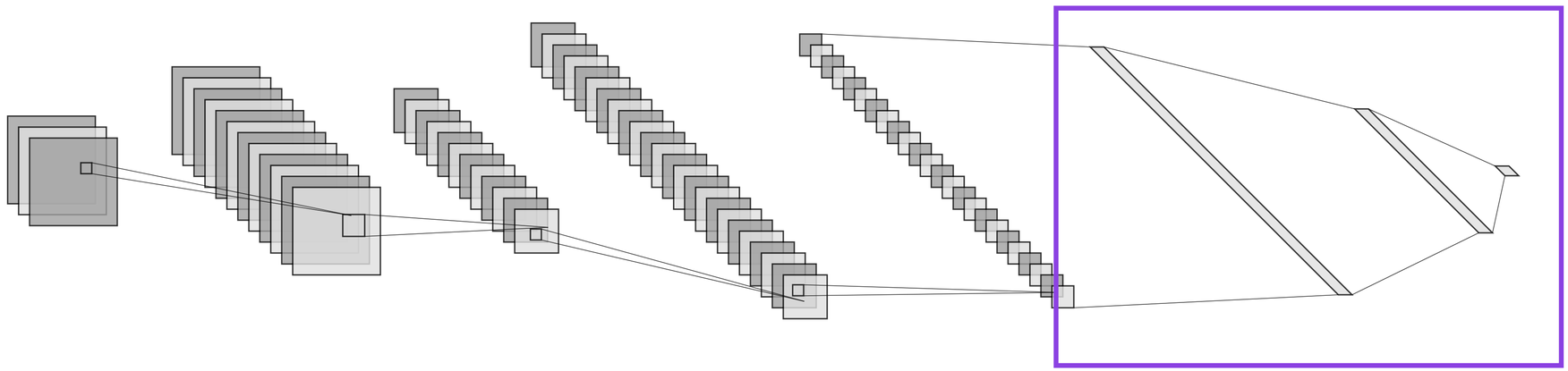
Convolutional part

```python
import torch.nn as nn


class PyTorchCNN(nn.Module):
    def __init__(self, num_classes):
        super().__init__()

        self.conv_layers = torch.nn.Sequential(
            nn.Conv2d(...),
            nn.MaxPool2d(...),
            nn.Conv2d(...),
            nn.MaxPool2d(...),
        )

        self.fc_layers = nn.Sequential(
            nn.Linear(24 * 16 * 16, 256),
            nn.ReLU(),
            nn.Linear(256, 128),
            nn.ReLU(),
            nn.Linear(128, num_classes),
        )

    def forward(self, x):
        features = self.conv_layers(x)
        features = torch.flatten(features, start_dim=1)
        logits = self.fc_layers(features)
        return logits
```



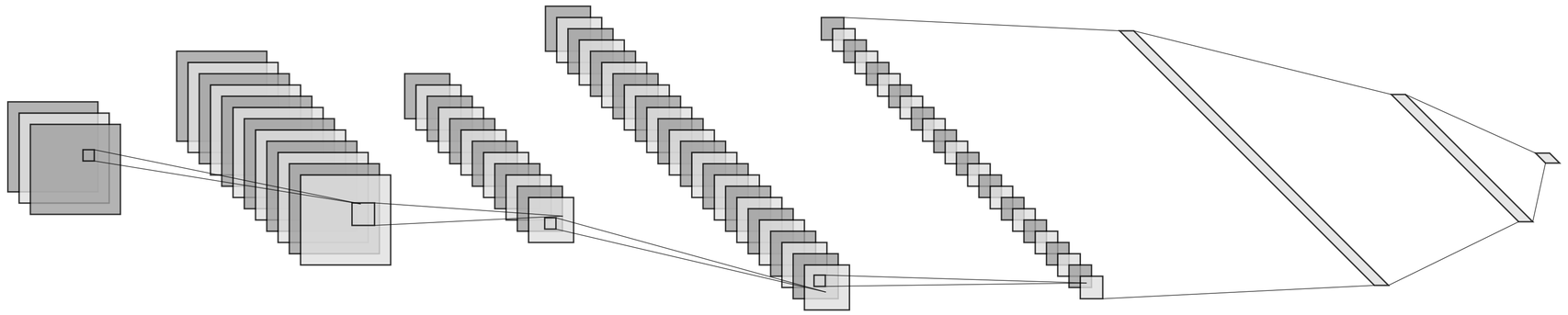
```python
import torch.nn as nn


class PyTorchCNN(nn.Module):
    def __init__(self, num_classes):
        super().__init__()

        self.conv_layers = torch.nn.Sequential(
            nn.Conv2d(...),
            nn.MaxPool2d(...),
            nn.Conv2d(...),
            nn.MaxPool2d(...),
        )

        self.fc_layers = nn.Sequential(
            nn.Linear(24 * 16 * 16, 256),
            nn.ReLU(),
            nn.Linear(256, 128),
            nn.ReLU(),
            nn.Linear(128, num_classes),
        )

    def forward(self, x):
        features = self.conv_layers(x)
        features = torch.flatten(features, start_dim=1)
        logits = self.fc_layers(features)
        return logits
```

Classifier part



```python
import torch.nn as nn


class PyTorchCNN(nn.Module):
    def __init__(self, num_classes):
        super().__init__()

        self.conv_layers = torch.nn.Sequential(
            nn.Conv2d(...),
            nn.MaxPool2d(...),
            nn.Conv2d(...),
            nn.MaxPool2d(...),
        )

        self.fc_layers = nn.Sequential(
            nn.Linear(24 * 16 * 16, 256),
            nn.ReLU(),
            nn.Linear(256, 128),
            nn.ReLU(),
            nn.Linear(128, num_classes),
        )

    def forward(self, x):
        features = self.conv_layers(x)
        features = torch.flatten(features, start_dim=1)
        logits = self.fc_layers(features)
        return logits
```

Connected via the forward method



Single-GPU Training

(More on multi-GPU training later)

Loading data onto a **single GPU** is easy!

```
[1]: import torch

     print(torch.cuda.is_available())
```

```
[1]: True
```

```
[2]: my_tensor = torch.tensor([1., 2., 3.])
     my_tensor
```

```
[2]: tensor([1., 2., 3.])
```

```
[3]: my_tensor = my_tensor.to('cuda')
     my_tensor
```

```
[3]: tensor([1., 2., 3.], device='cuda:0')
```

```
[4]: my_tensor = my_tensor.to('cpu')
     my_tensor
```

```
[4]: tensor([1., 2., 3.])
```

Only minor code modification required

```python
import torch.nn.functional as F

torch.manual_seed(1)

model = PyTorchMLP(num_features=784, num_classes=10)
model.to("cuda")

optimizer = torch.optim.SGD(model.parameters(), lr=0.05)

num_epochs = 10

for epoch in range(num_epochs):

    model = model.train()
    for batch_idx, (features, labels) in enumerate(train_loader):

        features = features.to("cuda")
        labels = labels.to("cuda")

        logits = model(features)

        loss = F.cross_entropy(logits, labels)

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

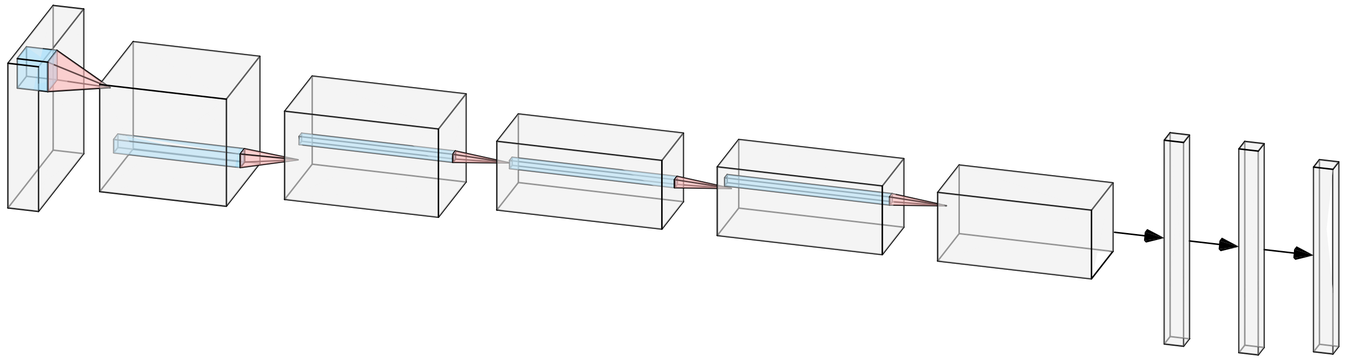
Multi-GPU is a different beast
(more on that later)

Transfer learning variants

Transfer learning

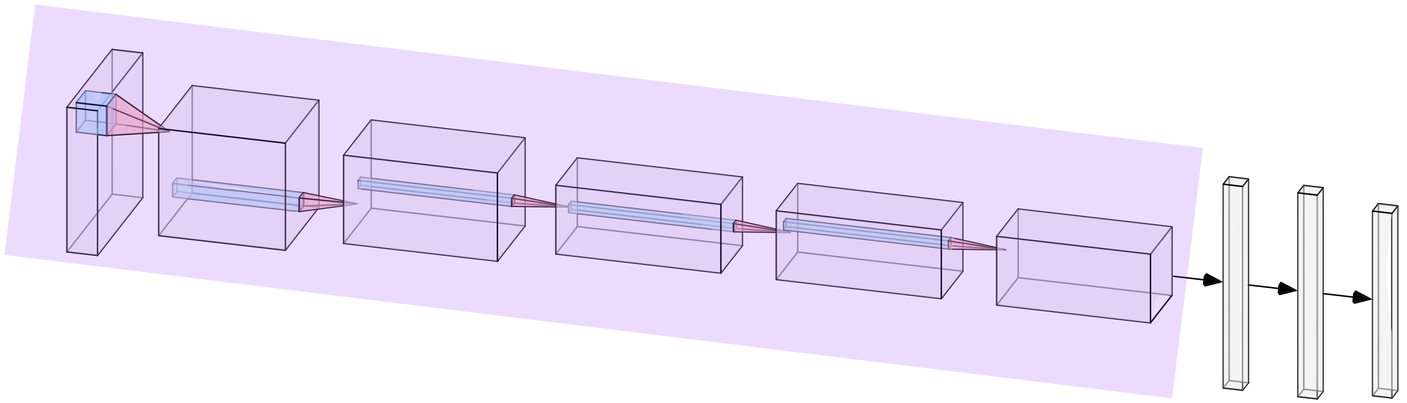
❶ Fine-tune last layer(s)

Step 1: train whole model on large dataset

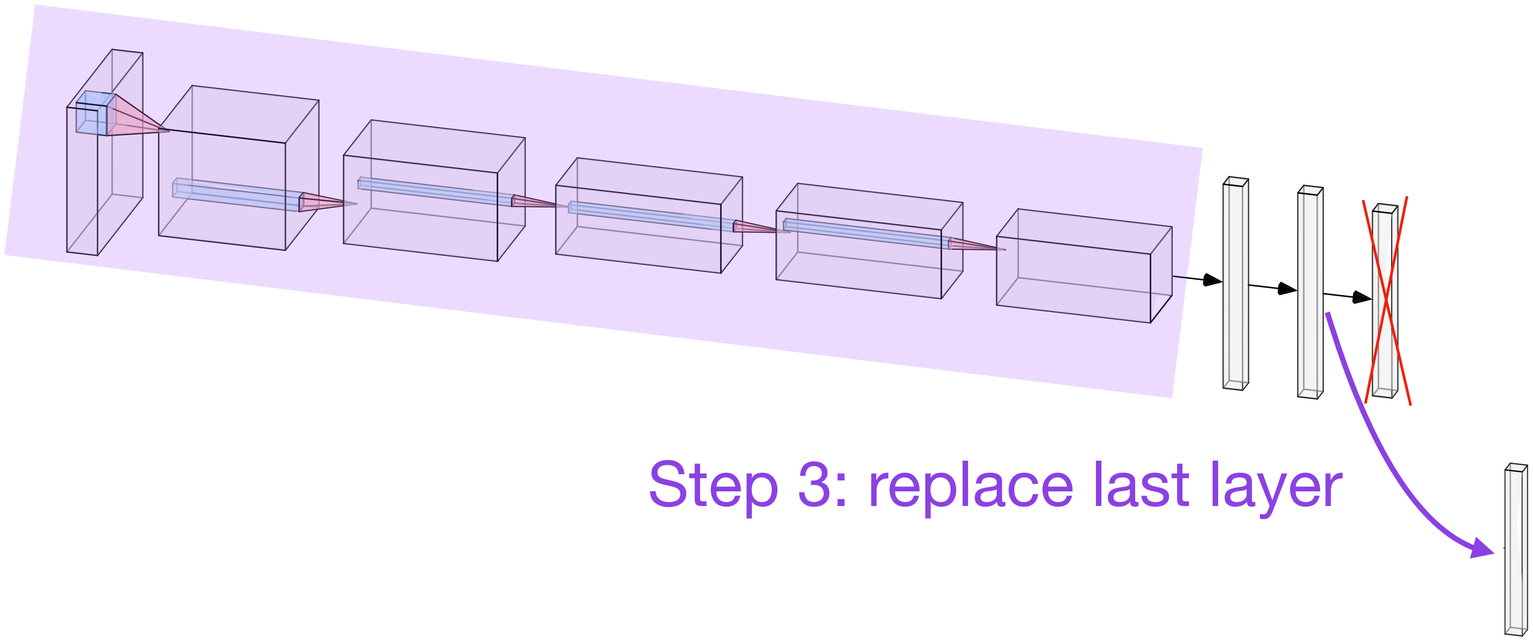


❶ Fine-tune last layer(s)

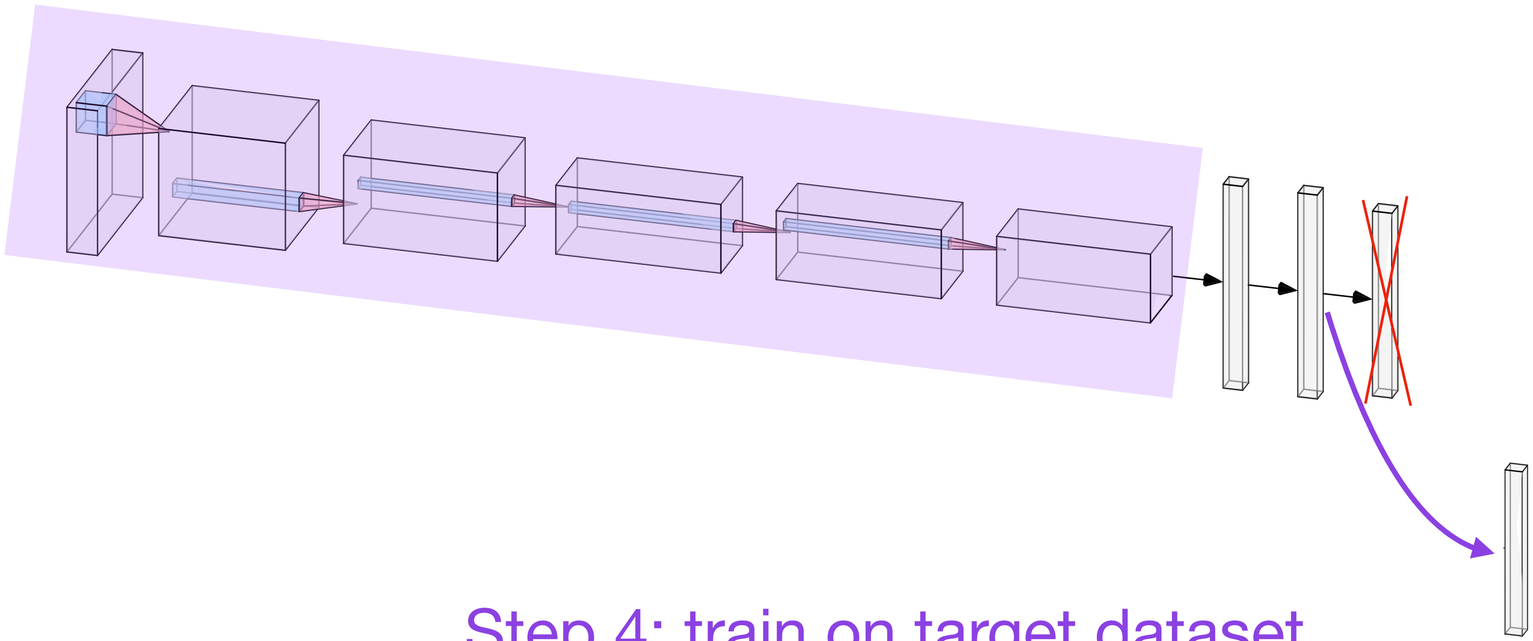
Step 2: freeze weights



❶ Fine-tune last layer(s)



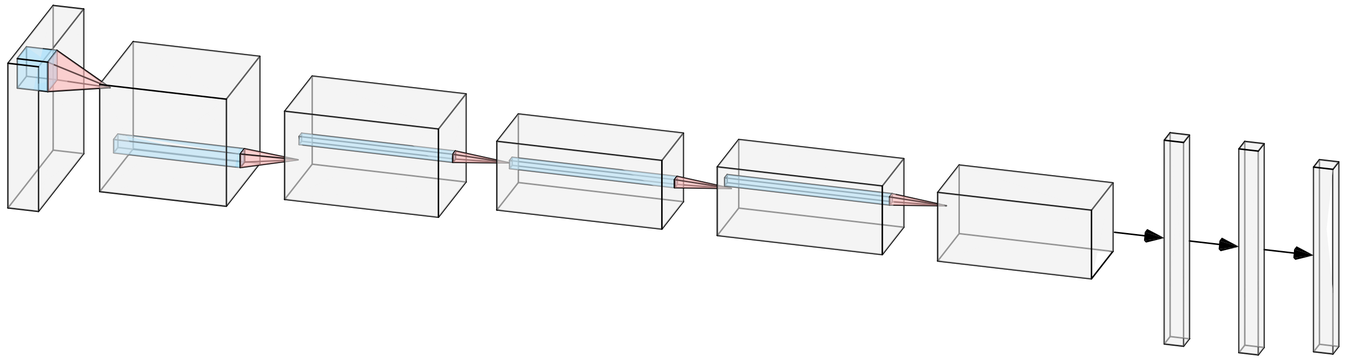
❶ Fine-tune last layer(s)



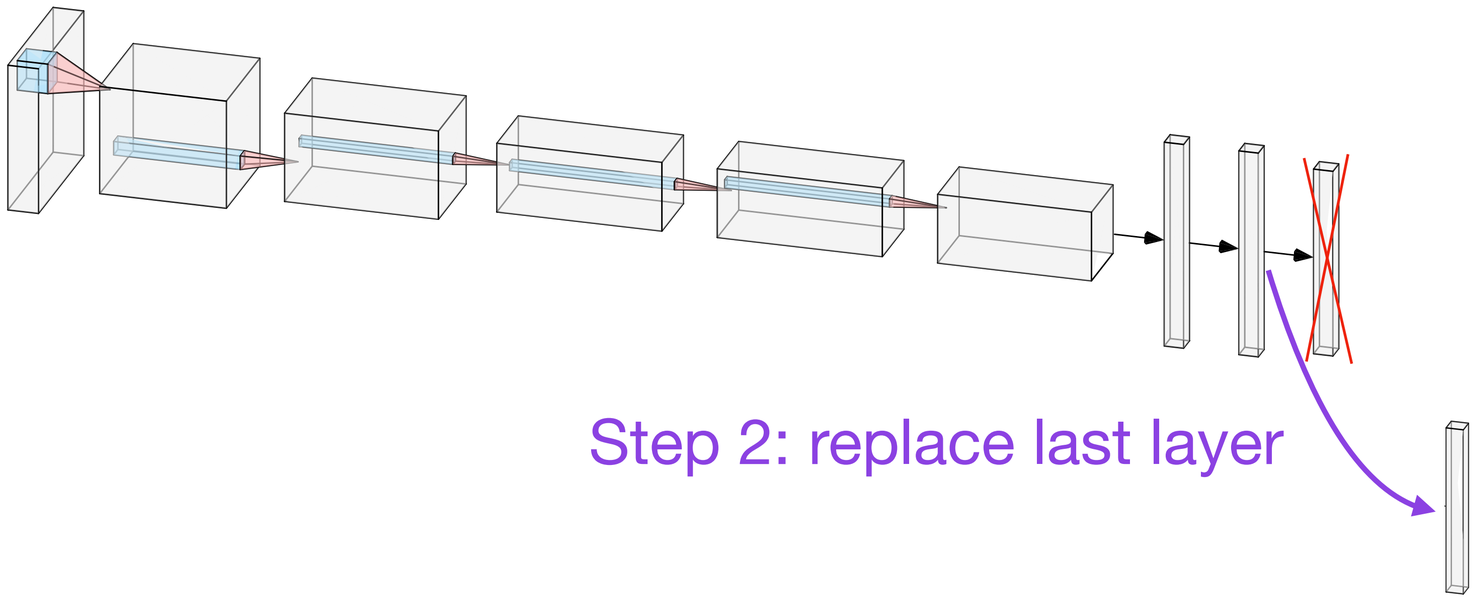
Step 4: train on target dataset,
but only update the last output layers

❷ Fine-tune the whole model

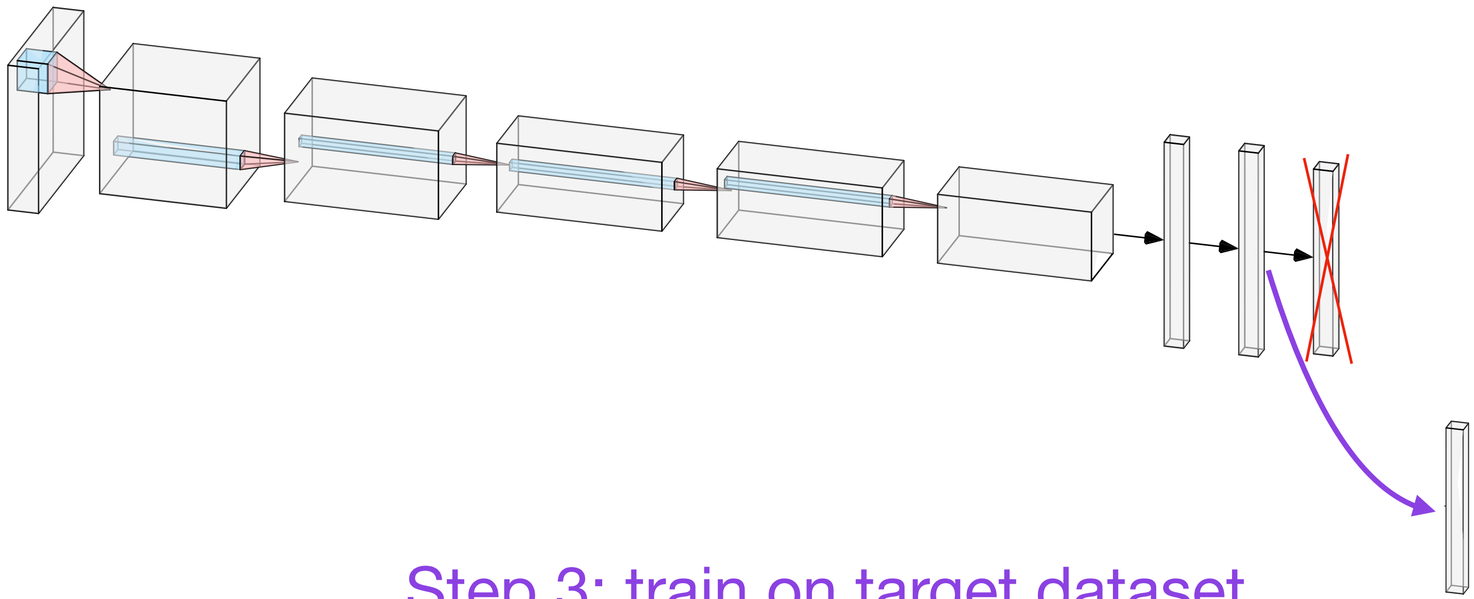
Step 1: train whole model on large dataset



❷ Fine-tune the whole model



❷ Fine-tune the whole model



Step 3: train on target dataset, update all layers

Exercise: Finetune a CNN from Torchvision

Replace CNN with a model from https://pytorch.org/vision/stable/models.html

Hint: Replace output layer with linear layer to specify 10 output classes

```
model = mobilenet_v3_large()

print(model)
```

```
MobileNetV3(
  (features): Sequential(
    (0): Conv2dNormActivation(
      (0): Conv2d(3, 16, kernel_size=(3, 3), stride=(2
      (1): BatchNorm2d(16, eps=0.001, momentum=0.01, a
      (2): Hardswish()
    )
    (1): InvertedResidual(
      (block): Sequential(
        (0): Conv2dNormActivation(
          (0): Conv2d(16, 16, kernel_size=(3, 3), stri
          (1): BatchNorm2d(16, eps=0.001, momentum=0.0
```