



Modern Deep Learning with PyTorch

07-10, 13:30–17:30 (America/Chicago), Classroom 106

3. Training Deep Neural Networks (2:30 - 3:00 pm)

Schedule

1. Introduction to Deep Learning (1:30 - 2:00 pm)
2. Understanding the PyTorch API (2:00 - 2:30 pm)
3. Training Deep Neural Networks (2:30 - 3:00 pm)

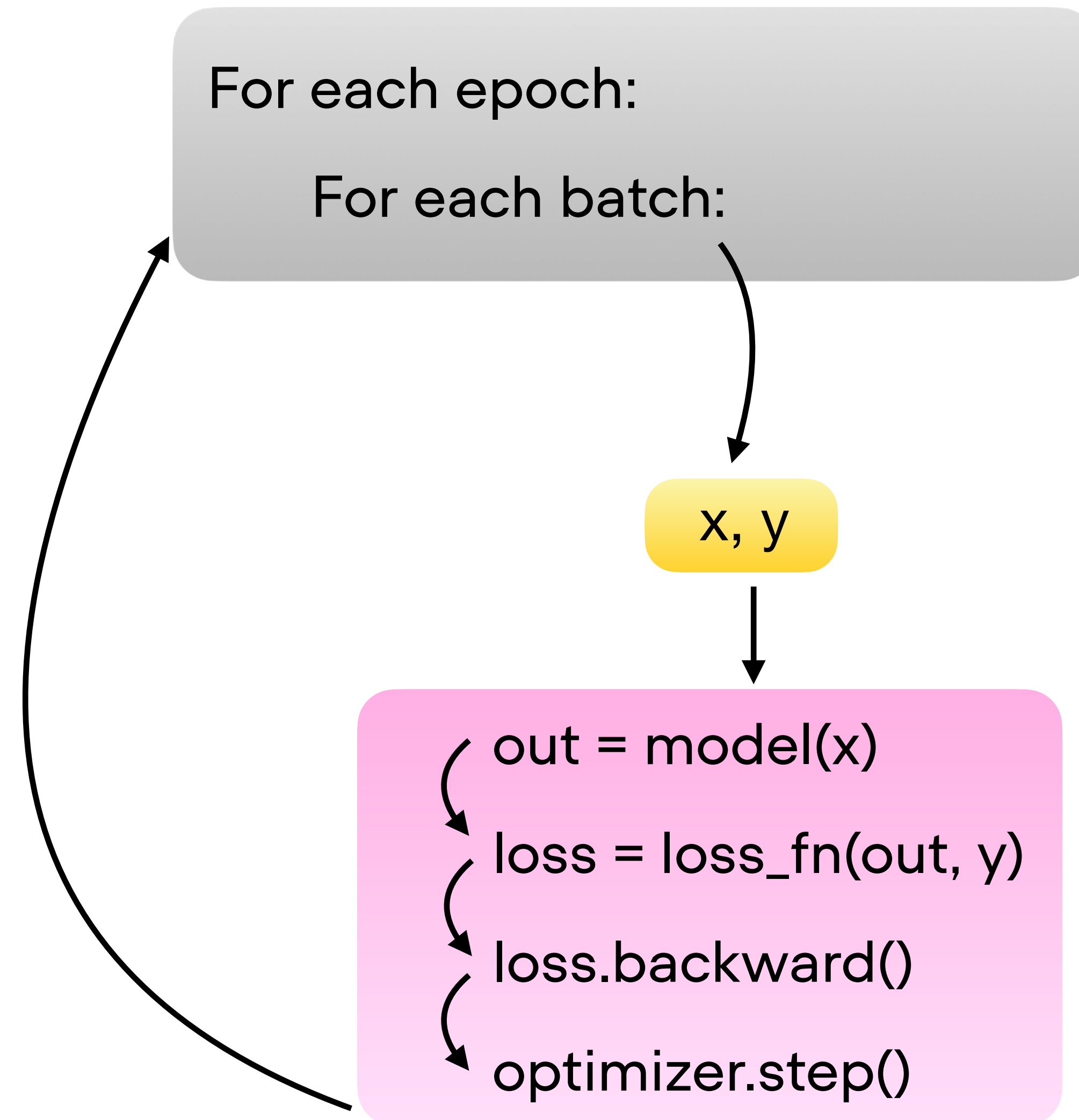
10 Min Break

4. Accelerating PyTorch Model Training (3:10 - 3:45 pm)
5. Organizing PyTorch Code (3:45 - 4:15 pm)
6. More Tips and Techniques (4:15 - 4:45 pm)

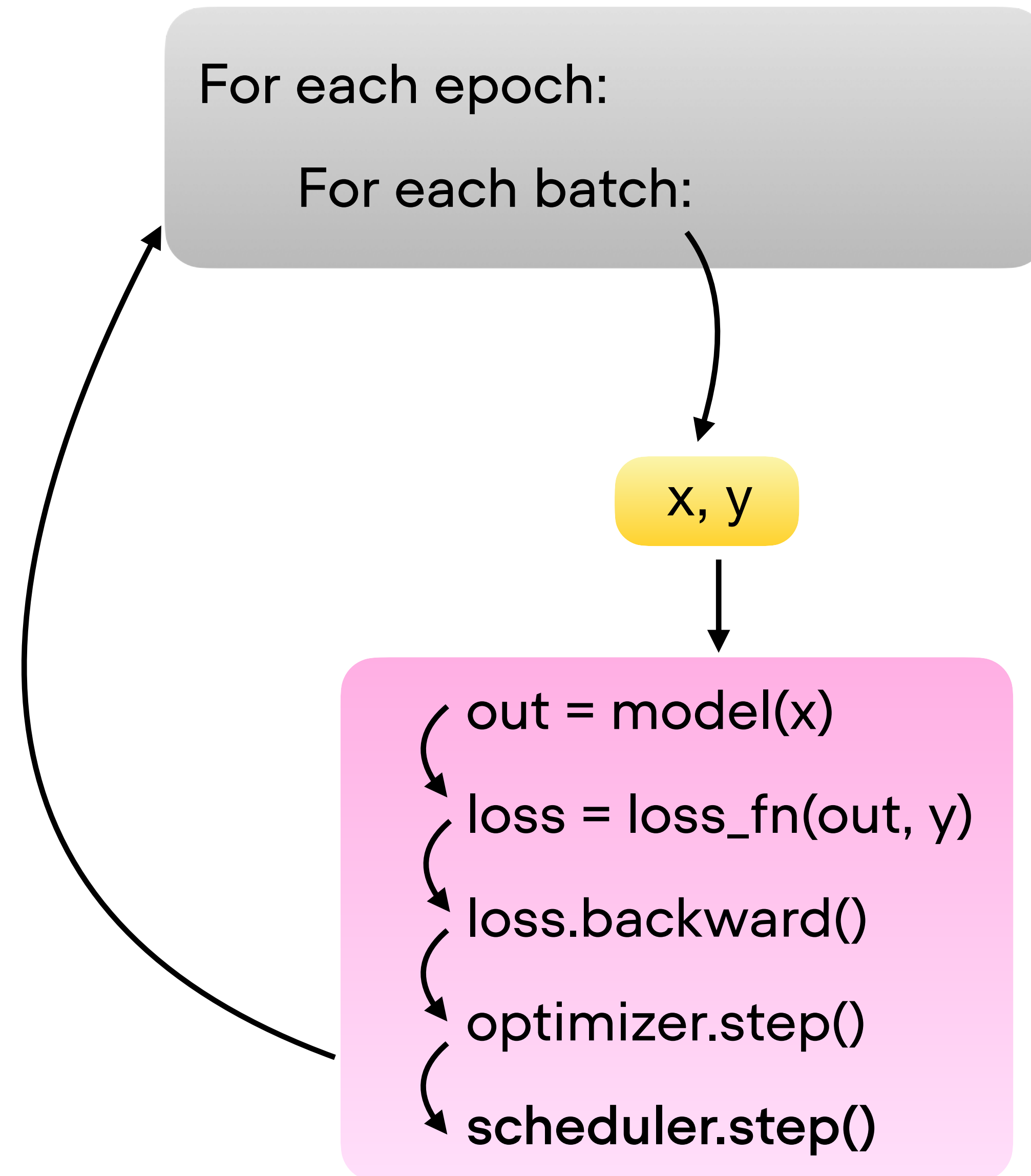
10 Min Break

7. Finetuning LLMs (4:55 - 5:25 pm)
8. Wrap Up & Questions (5:25 - 5:30 pm)

Recap



Adding a scheduler



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

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

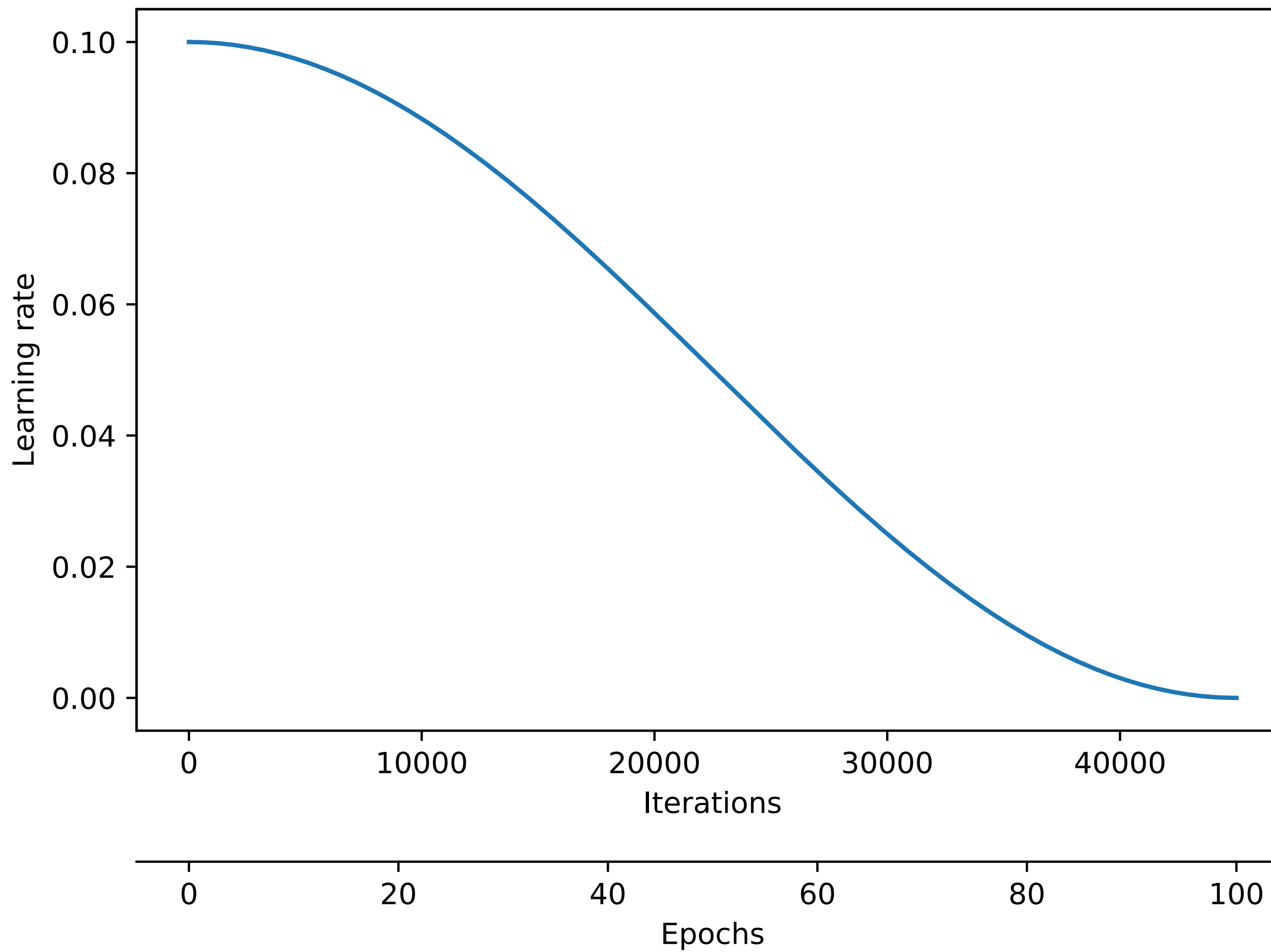
for batch_idx, (features, targets) in enumerate(train_loader):
    model.train()

    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()
```



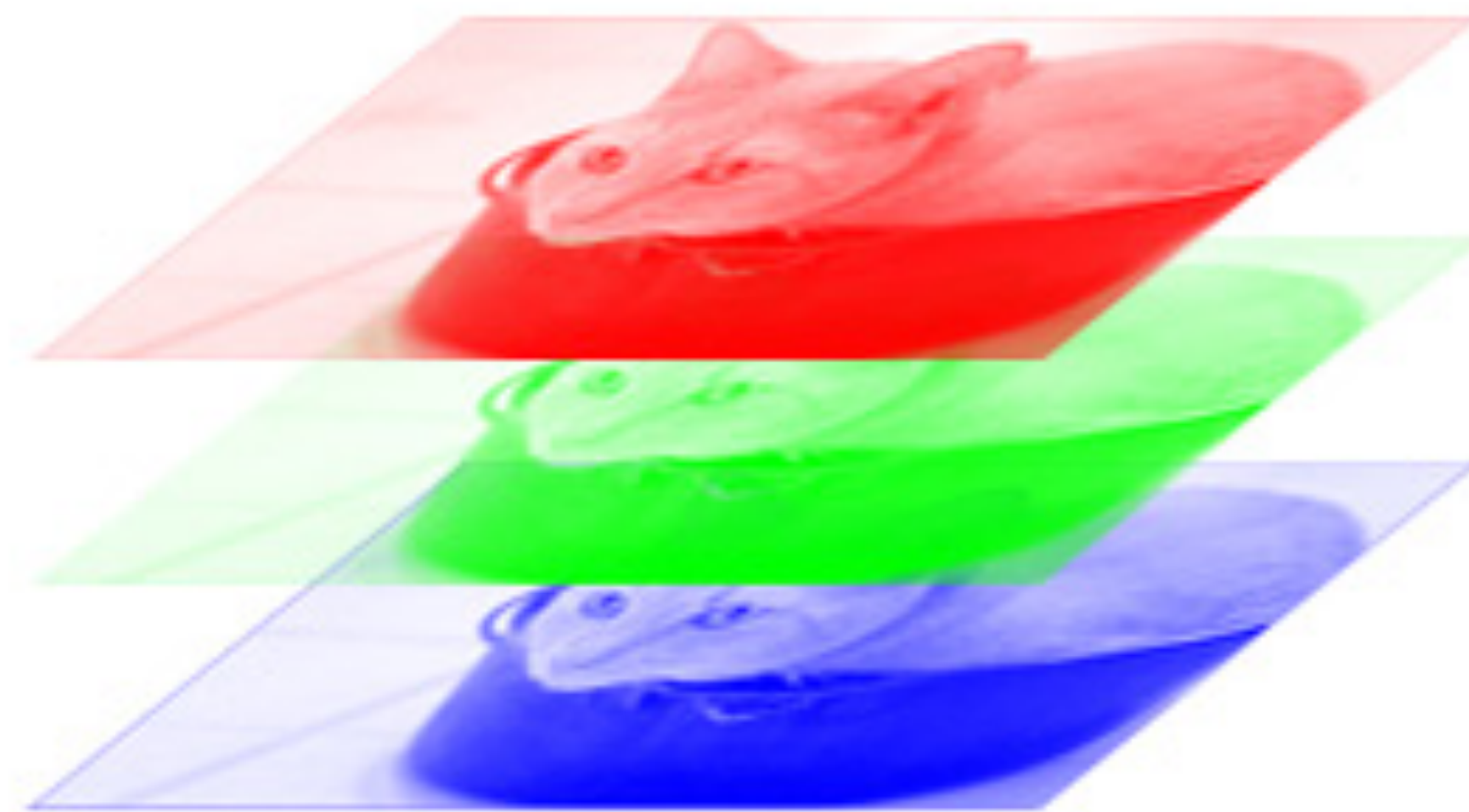
From tabular data to image data

RGB Image

R

G

B



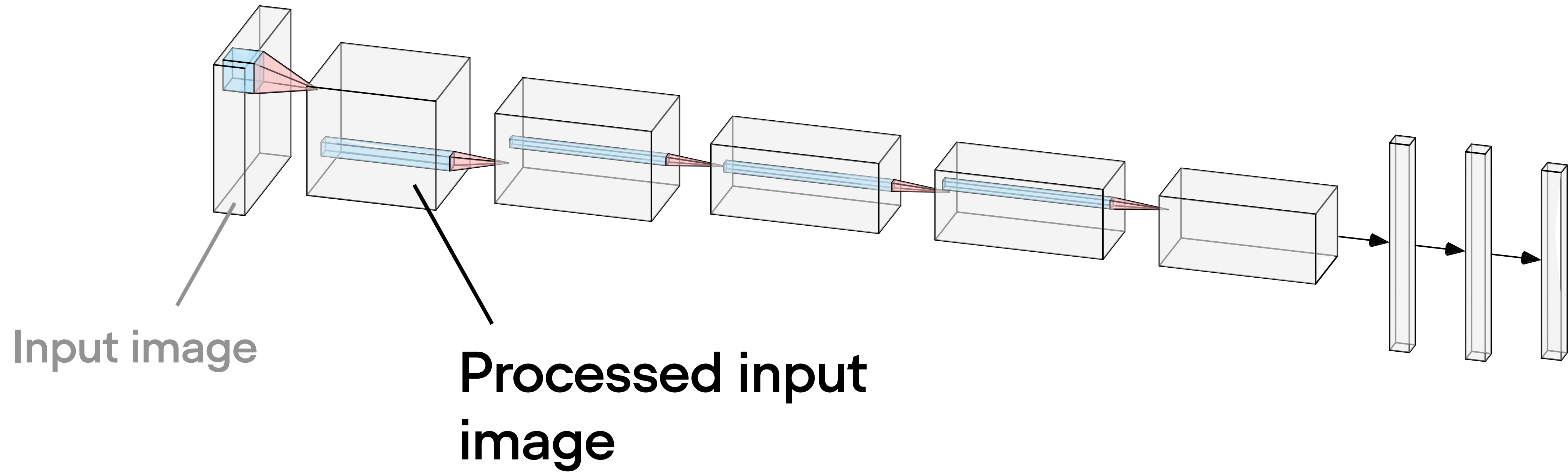
=



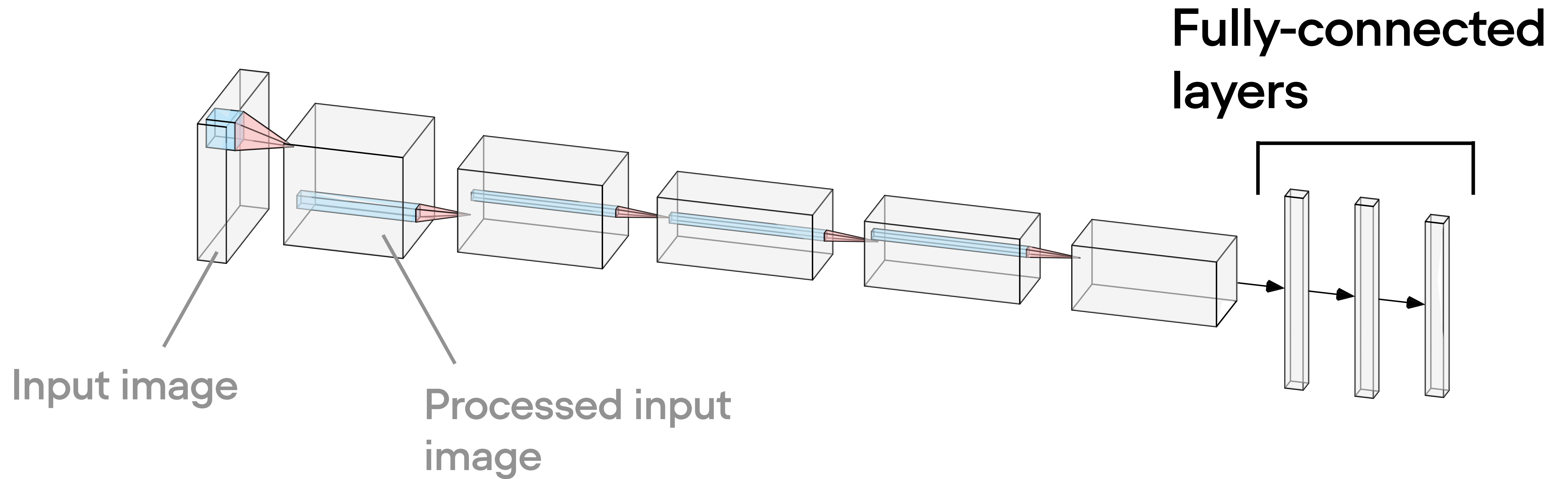
Image Source: <https://code.tutsplus.com/tutorials/create-a-retro-crt-distortion-effect-using-rgb-shifting--active-3359>

Color image as a stack of matrices

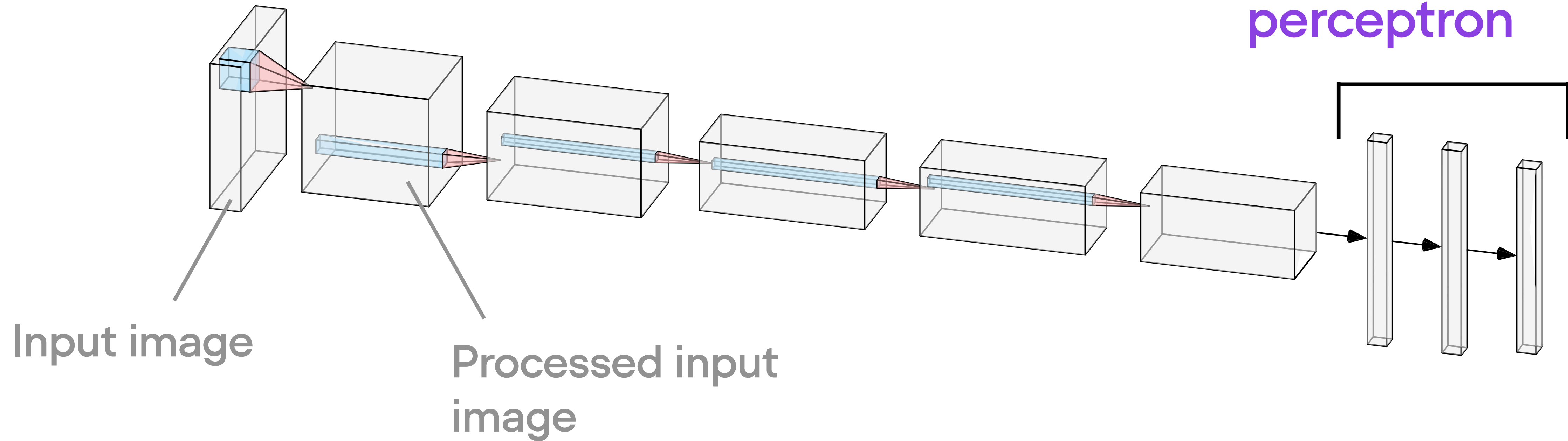
A typical convolutional neural network architecture



A typical convolutional neural network architecture



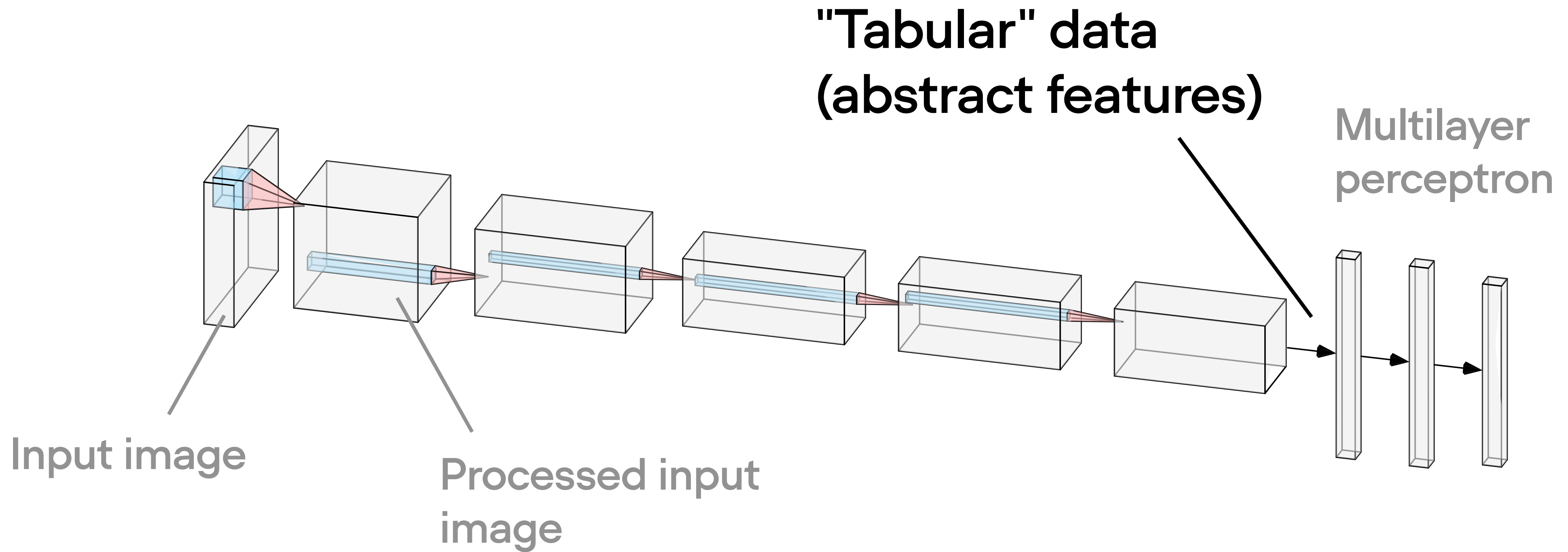
A typical convolutional neural network architecture



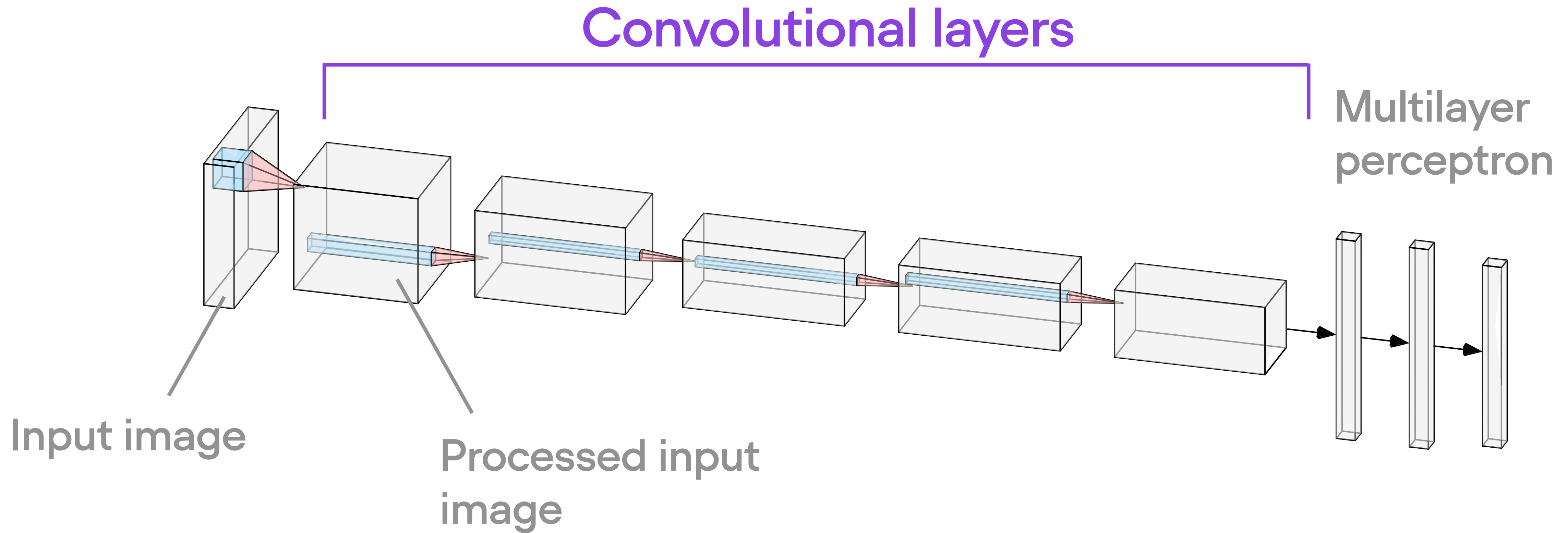
Fully-connected layers

→ Multilayer perceptron

A typical convolutional neural network architecture



A typical convolutional neural network architecture



Convolutional part

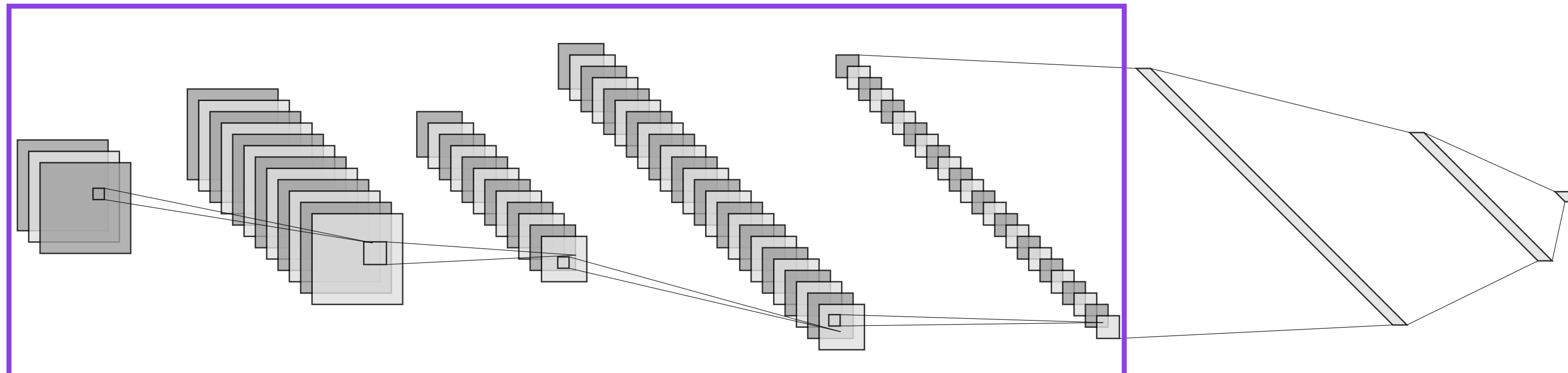
```
import torch.nn as nn

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
```



```

import torch.nn as nn

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

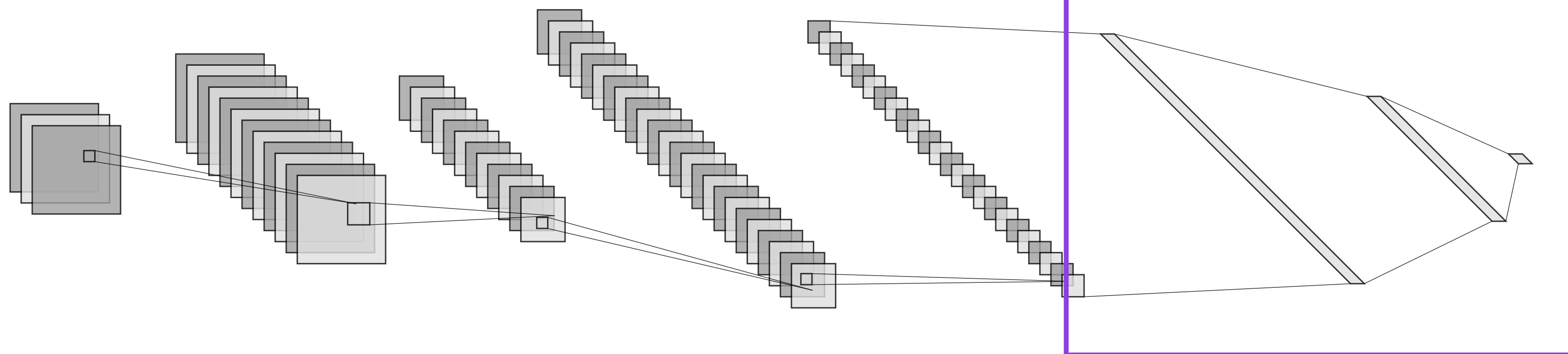
        self.conv_layers = torch.nn.Sequential(
            nn.Conv2d(...),
            nn.MaxPool2d(...),
            nn.Conv2d(...),
            nn.MaxPool2d(...),
        )

        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



```

import torch.nn as nn

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

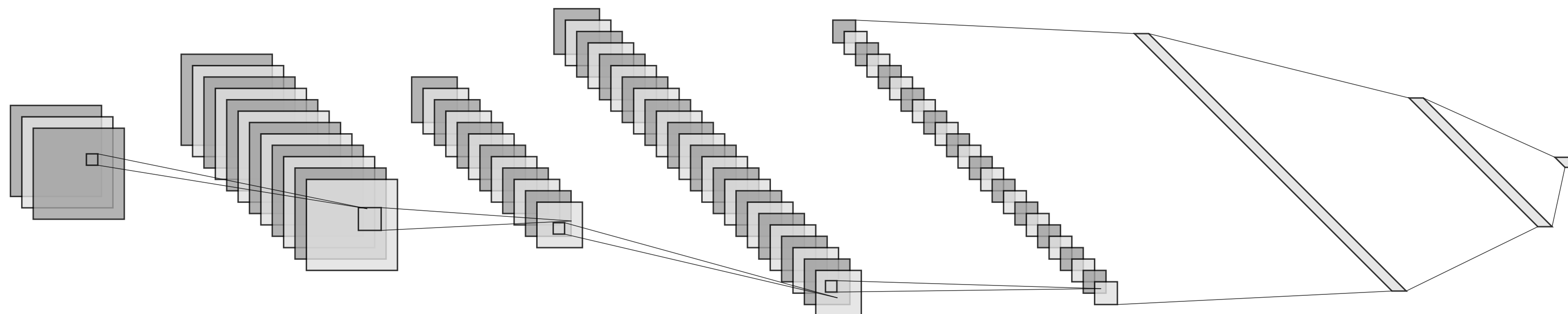
        self.conv_layers = torch.nn.Sequential(
            nn.Conv2d(...),
            nn.MaxPool2d(...),
            nn.Conv2d(...),
            nn.MaxPool2d(...),
        )

        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

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

```
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()
```

Transfer learning variants

Transfer learning

Step 1: train on ...

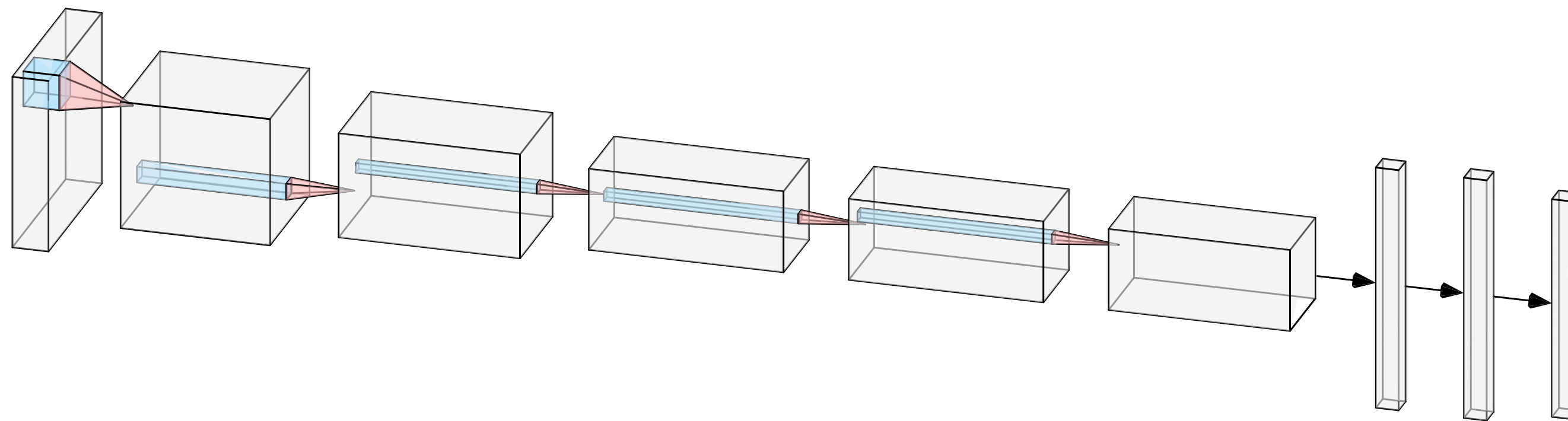
Large general dataset

Step 2: fine-tune on ...

Smaller target dataset

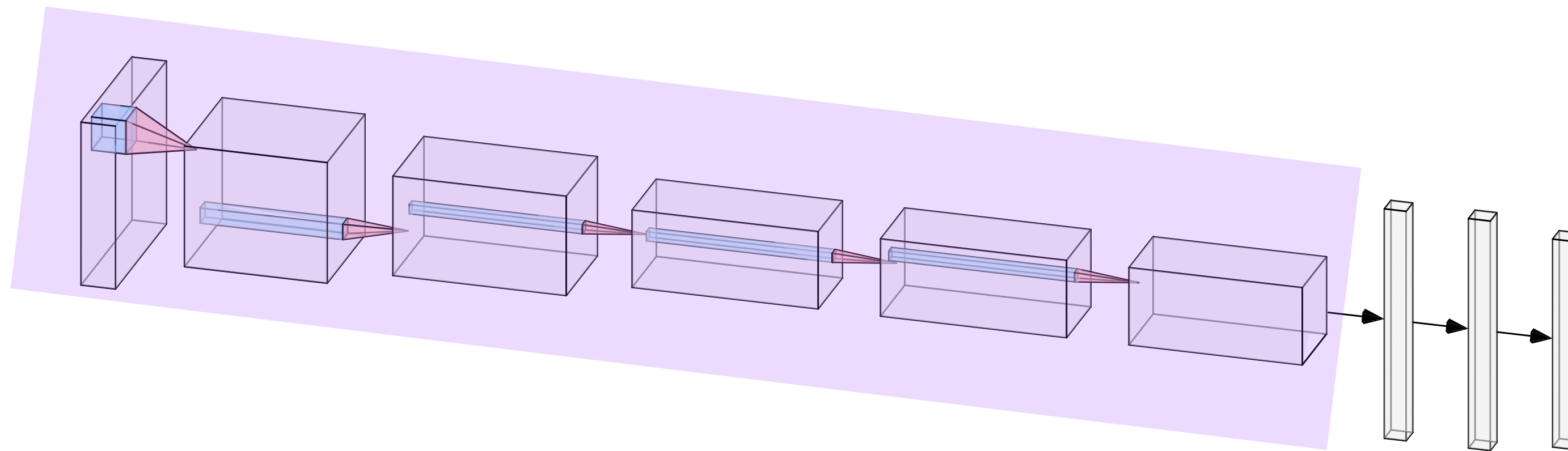
1 Fine-tune last layer(s)

Step 1: train whole model on large dataset

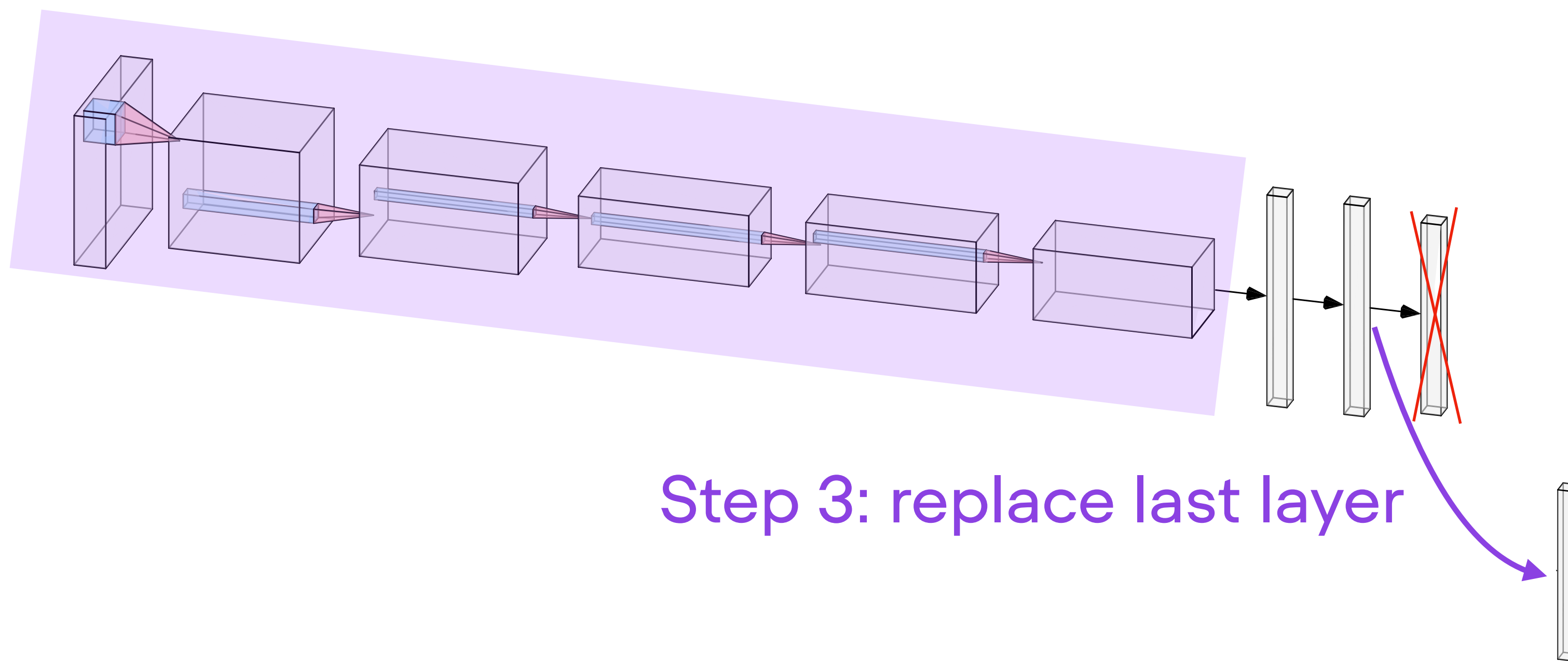


1 Fine-tune last layer(s)

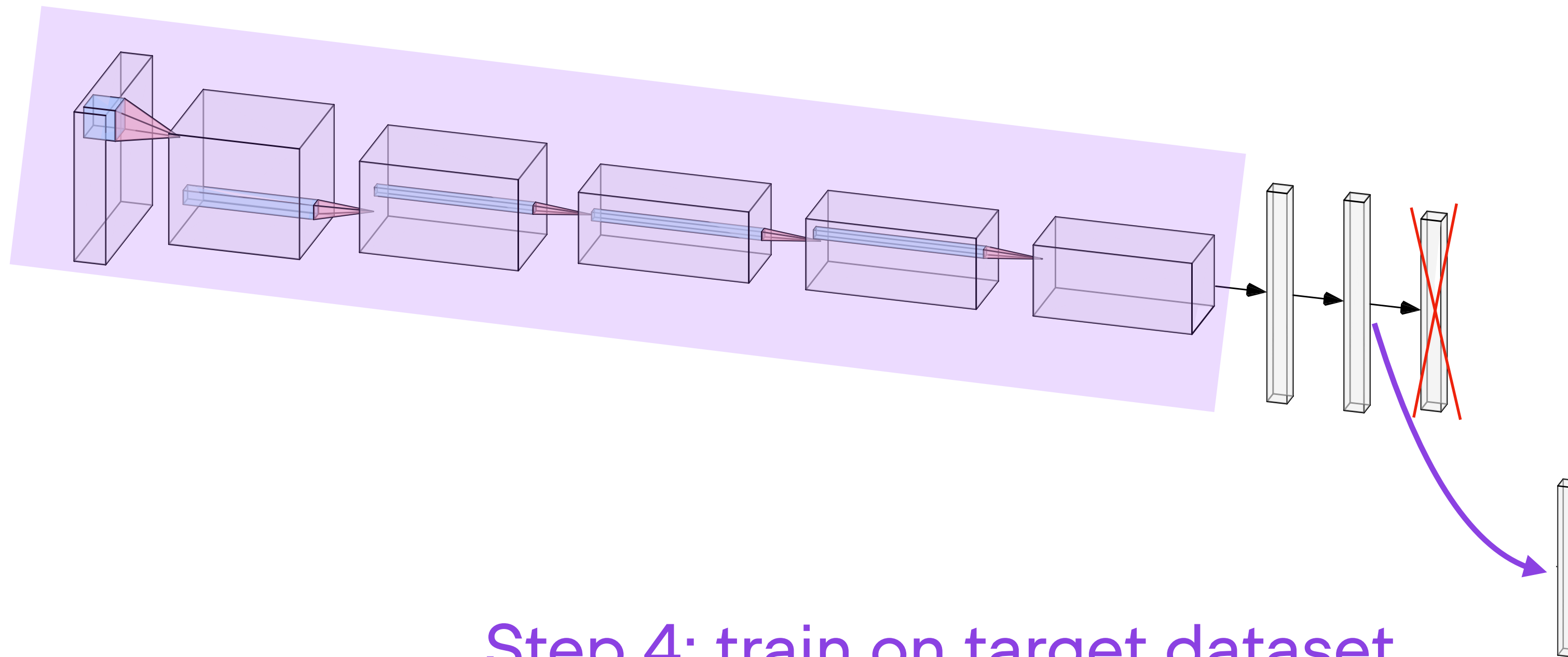
Step 2: freeze weights



1 Fine-tune last layer(s)



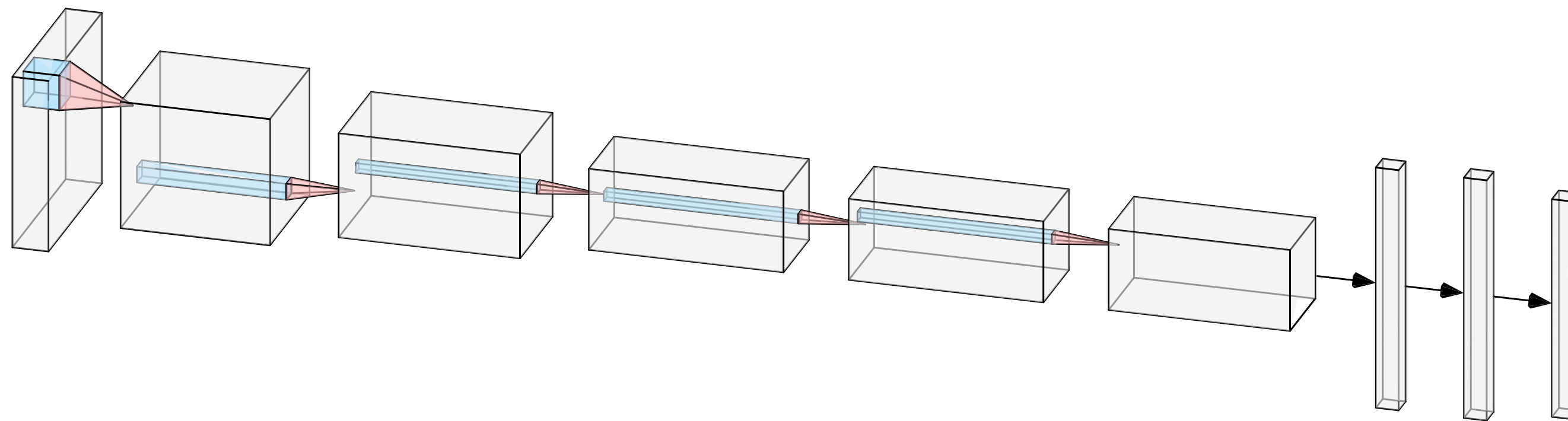
1 Fine-tune last layer(s)



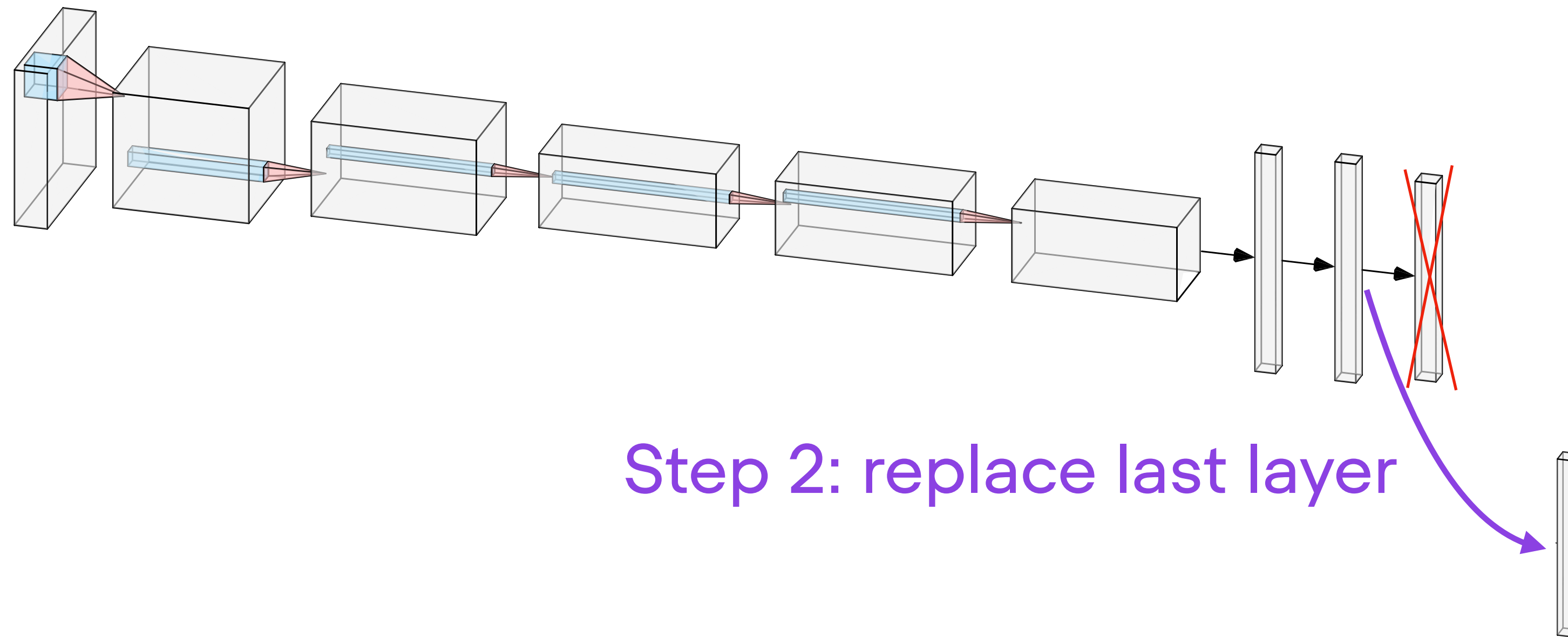
Step 4: train on target dataset,
but only update the last output layers

2 Fine-tune the whole model

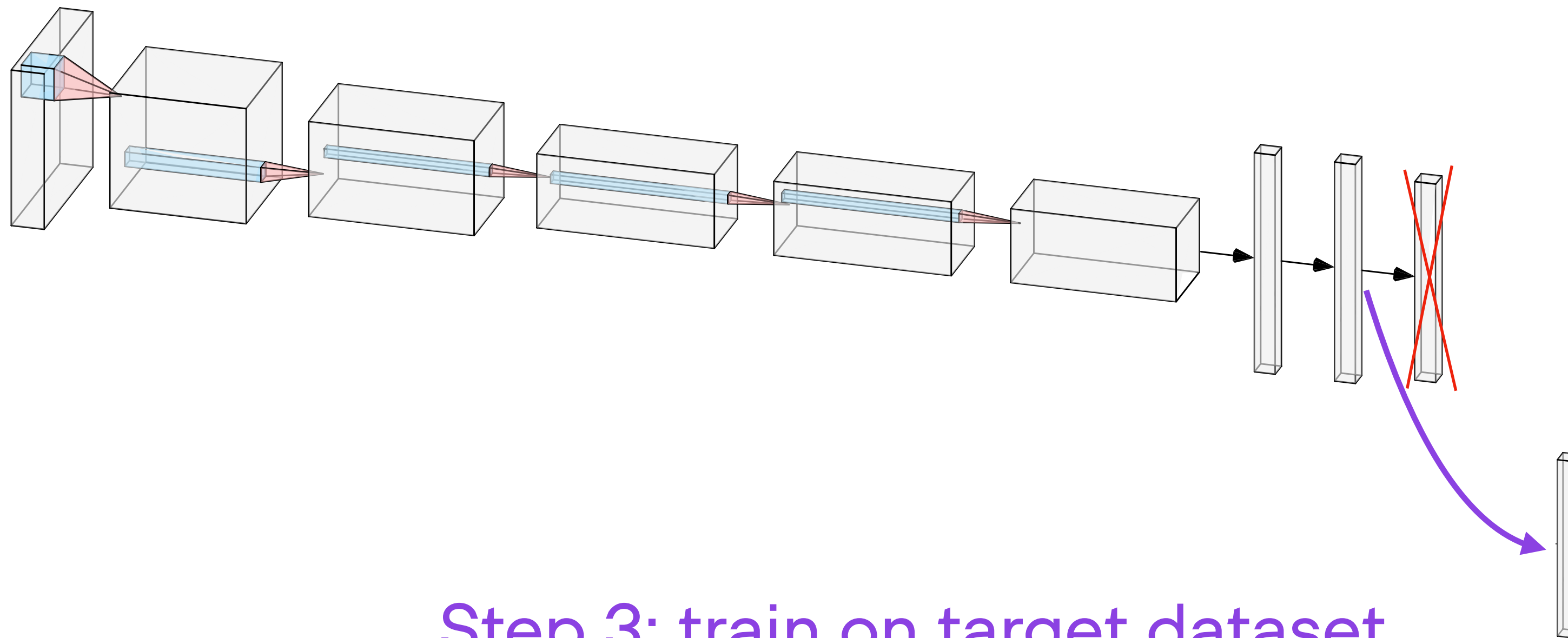
Step 1: train whole model on large dataset



2 Fine-tune the whole model



2 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, 2), bias=False)
      (1): BatchNorm2d(16, eps=0.001, momentum=0.01, affine=True, track_running_stats=True)
      (2): Hardswish()
    )
    (1): InvertedResidual(
      (block): Sequential(
        (0): Conv2dNormActivation(
          (0): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), bias=False)
          (1): BatchNorm2d(16, eps=0.001, momentum=0.01, affine=True, track_running_stats=True)
          (2): Hardswish()
        )
        (1): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), bias=False)
        (2): BatchNorm2d(16, eps=0.001, momentum=0.01, affine=True, track_running_stats=True)
        (3): Hardswish()
      )
    )
  )
)
```