In [1]:

```
import copy
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision
import torchvision.transforms as transforms
```

PyTorch

In this notebook you will gain some hands-on experience with PyTorch (https://pytorch.org/), one of the major frameworks for deep learning. To install PyTorch. follow the official installation instructions (https://pytorch.org/get-started/locally/). Make sure that you select the correct OS & select the version with CUDA if your computer supports it. If you do not have an Nvidia GPU, you can install the CPU version by setting `CUDA` to `None`. However, in this case we recommend using Google Colab (https://colab.research.google.com/). Make sure that you enable GPU acceleration in `Runtime > Change runtime type`.

You will start by re-implementing some common features of deep neural networks (dropout and batch normalization) and then implement a very popular modern architecture for image classification (ResNet) and improve its training loop.

1. Dropout

Dropout is a form of regularization for neural networks. It works by randomly setting activations (values) to 0, each one with equal probability `p`. The values are then scaled by a factor $\frac{1}{1-p}$ to conserve their mean.

Dropout effectively trains a pseudo-ensemble of models with stochastic gradient descent. During evaluation we want to use the full ensemble and therefore have to turn off dropout. Use `self.training` to check if the model is in training or evaluation mode.

Do not use any dropout implementation from PyTorch for this!

In [2]:

```
x=torch.rand(4,2,10)
x.var(dim=0,keepdims=True).shape
```

Out[2]:

```
torch.Size([1, 2, 10])
```

In [3]:

```
class Dropout(nn.Module):
    """
    Dropout, as discussed in the lecture and described here:
    https://pytorch.org/docs/stable/nn.html#torch.nn.Dropout

    Args:
        p: float, dropout probability
    """
    def __init__(self, p):
        super().__init__()
        self.p = p

    def forward(self, input):
        """
        The module's forward pass.
        This has to be implemented for every PyTorch module.
        PyTorch then automatically generates the backward pass
        by dynamically generating the computational graph during
        execution.

        Args:
            input: PyTorch tensor, arbitrary shape

        Returns:
            PyTorch tensor, same shape as input
        """

        tmp=torch.tensor((torch.rand(*input.shape)>self.p),dtype=torch.int)

        return tmp*input*1/(1-self.p)
```

In [4]:

```
# Test dropout
test = torch.rand(10_000)
dropout = Dropout(0.2)
test_dropped = dropout(test)

# These assertions can in principle fail due to bad luck, but
# if implemented correctly they should almost always succeed.
assert np.isclose(test_dropped.mean().item(), test.mean().item(), atol=1e-2)
assert np.isclose((test_dropped > 0).float().mean().item(), 0.8, atol=1e-2)
```

```
c:\users\kashif ai\i2dl_exercises\exercise_01\.myenv\lib\site-packages\ipykernel_launcher.py:28: Use
rWarning: To copy construct from a tensor, it is recommended to use sourceTensor.clone().detach() or
sourceTensor.clone().detach().requires_grad_(True), rather than torch.tensor(sourceTensor).
```

2. Batch normalization

Batch normalization is a trick use to smoothen the loss landscape and improve training. It is defined as the function

$$y = \frac{x - \mu_x}{\sigma_x + \epsilon} \cdot \gamma + \beta$$

, where γ and β and learnable parameters and ϵ is a some small number to avoid dividing by zero. The Statistics μ_x and σ_x are taken separately for each feature. In a CNN this means averaging over the batch and all pixels.

Do not use any batch normalization implementation from PyTorch for this!

In [5]:

```
class BatchNorm(nn.Module):
    """
    Batch normalization, as discussed in the lecture and similar to
    https://pytorch.org/docs/stable/nn.html#torch.nn.BatchNorm1d

    Only uses batch statistics (no running mean for evaluation).
    Batch statistics are calculated for a single dimension.
    Gamma is initialized as 1, beta as 0.

    Args:
        num_features: Number of features to calculate batch statistics for.
    """
    def __init__(self, num_features):
        super().__init__()


        self.gamma = nn.Parameter(torch.ones(num_features))
        self.beta = nn.Parameter(torch.zeros(num_features))

    def forward(self, input):
        """
        Batch normalization over the dimension C of (N, C, L).

        Args:
            input: PyTorch tensor, shape [N, C, L]

        Return:
            PyTorch tensor, same shape as input
        """
        N,C,L=input.shape
        eps = 1e-5
        input=input.permute(1,0,2).reshape(C,-1)
        mean=input.mean(dim=-1,keepdims=True)
        std=input.std(dim=-1,keepdims=True)
        out=(input-mean)/(std+eps)
        out=self.gamma.unsqueeze(dim=-1)*out+self.beta.unsqueeze(dim=-1)
        #print(self.gamma.unsqueeze(dim=0).unsqueeze(dim=-1).shape)
        return out.reshape(C,N,L).permute(1,0,2)
```

In [6]:

```
# Tests the batch normalization implementation
torch.random.manual_seed(42)
test = torch.randn(8, 2, 4)

b1 = BatchNorm(2)
test_b1 = b1(test)

b2 = nn.BatchNorm1d(2, affine=False, track_running_stats=False)
test_b2 = b2(test)

assert torch.allclose(test_b1, test_b2, rtol=0.02)
```

3. ResNet

ResNet is the models that first introduced residual connections (a form of skip connections). It is a rather simple, but successful and very popular architecture. In this part of the exercise we will re-implement it step by step.

Note that there is also an [improved version of ResNet (https://arxiv.org/abs/1603.05027)](https://arxiv.org/abs/1603.05027) with optimized residual blocks. Here we will implement the [original version (https://arxiv.org/abs/1512.03385)](https://arxiv.org/abs/1512.03385) for CIFAR-10. Your dropout and batchnorm implementations won't help you here. Just use PyTorch's own layers.

This is just a convenience function to make e.g. `nn.Sequential` more flexible. It is e.g. useful in combination with `x.squeeze()` .

In [7]:

```
class Lambda(nn.Module):
    def __init__(self, func):
        super().__init__()
        self.func = func

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

We begin by implementing the residual blocks. The block is illustrated by this sketch:

Note that we use 'SAME' padding, no bias, and batch normalization after each convolution. You do not need `nn.Sequential` here. The skip connection is already implemented as `self.skip` . It can handle different strides and increases in the number of channels.

In [8]:

```
class ResidualBlock(nn.Module):
    """
    The residual block used by ResNet.

    Args:
        in_channels: The number of channels (feature maps) of the incoming embedding
        out_channels: The number of channels after the first convolution
        stride: Stride size of the first convolution, used for downsampling
    """

    def __init__(self, in_channels, out_channels, stride=1):
        super().__init__()
        if stride > 1 or in_channels != out_channels:
            # Add strides in the skip connection and zeros for the new channels.
            self.skip = Lambda(lambda x: F.pad(x[:, :, ::stride, ::stride],
                                               (0, 0, 0, 0, 0, out_channels - in_channels),
                                               mode="constant", value=0))
        else:
            self.skip = nn.Sequential()

        self.conv1=nn.Conv2d(in_channels,out_channels,3,padding=1,stride=stride,bias=False)
        self.relu=nn.ReLU()
        self.bn1=nn.BatchNorm2d(out_channels)
        self.conv2=nn.Conv2d(out_channels,out_channels,3,padding=1,stride=1,bias=False)

        self.bn2=nn.BatchNorm2d(out_channels)

    def forward(self, input):
        x=self.conv1(input)
        x=self.bn1(x)
        x=self.relu(x)
        x=self.conv2(x)
        x=self.bn2(x)
        x=self.relu(self.skip(input)+x)
        return x
```

Next we implement a stack of residual blocks for convenience. The first layer in the block is the one changing the number of channels and downsampling. You can use `nn.ModuleList` to use a list of child modules.

In [9]:

```
class ResidualStack(nn.Module):
    """
    A stack of residual blocks.

    Args:
        in_channels: The number of channels (feature maps) of the incoming embedding
        out_channels: The number of channels after the first layer
        stride: Stride size of the first layer, used for downsampling
        num_blocks: Number of residual blocks
    """

    def __init__(self, in_channels, out_channels, stride, num_blocks):
        super().__init__()

        self.modules1=nn.ModuleList([ResidualBlock(in_channels,out_channels,stride=stride)])
        self.module_list=self.modules1.extend([ResidualBlock(out_channels,out_channels,stride=1) for i in range(n
um_blocks-1)])


    def forward(self, input):
        x=input
        for layer in self.module_list:
            x=layer(x)
        return x
```

Now we are finally ready to implement the full model! To do this, use the `nn.Sequential` API and carefully read the following paragraph from the paper (Fig. 3 is not important):

Note that a convolution layer is always convolution + batch norm + activation (ReLU), that each ResidualBlock contains 2 layers, and that you might have to `squeeze` the embedding before the dense (fully-connected) layer.

In [10]:

```
n = 5
num_classes = 10
layers=[nn.Conv2d(3,16,3,1,1,bias=False),nn.BatchNorm2d(16),nn.ReLU()]
in_channel=16
for i in [16,32,64]:
    if i==16:
        layers.append(ResidualStack(in_channel,i,1,n))
    else:
        layers.append(ResidualStack(in_channel,i,2,n))
    in_channel=i
layers=layers+[nn.AdaptiveAvgPool2d(1),nn.Flatten(1),nn.Linear(64,num_classes),nn.Softmax(dim=-1)]

resnet=nn.Sequential(*layers)
# TODO: Implement ResNet via nn.Sequential
```

Next we need to initialize the weights of our model.

In [11]:

```
def initialize_weight(module):
    if isinstance(module, (nn.Linear, nn.Conv2d)):
        nn.init.kaiming_normal_(module.weight, nonlinearity='relu')
    elif isinstance(module, nn.BatchNorm2d):
        nn.init.constant_(module.weight, 1)
        nn.init.constant_(module.bias, 0)

resnet.apply(initialize_weight);
```

4. Training

So now we have a shiny new model, but that doesn't really help when we can't train it. So that's what we do next.

First we need to load the data. Note that we split the official training data into train and validation sets, because you must not look at the test set until you are completely done developing your model and report the final results. Some people don't do this properly, but you should not copy other people's bad habits.

In [12]:

```
class CIFAR10Subset(torchvision.datasets.CIFAR10):
    """
    Get a subset of the CIFAR10 dataset, according to the passed indices.
    """
    def __init__(self, *args, idx=None, **kwargs):
        super().__init__(*args, **kwargs)

        if idx is None:
            return

        self.data = self.data[idx]
        targets_np = np.array(self.targets)
        self.targets = targets_np[idx].tolist()
```

We next define transformations that change the images into PyTorch tensors, standardize the values according to the precomputed mean and standard deviation, and provide data augmentation for the training set.

In [13]:

```
normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                 std=[0.229, 0.224, 0.225])
transform_train = transforms.Compose([
    transforms.RandomHorizontalFlip(),
    transforms.RandomCrop(32, 4),
    transforms.ToTensor(),
    normalize,
])
transform_eval = transforms.Compose([
    transforms.ToTensor(),
    normalize
])
```

In [14]:

```
ntrain = 45_000
train_set = CIFAR10Subset(root='./data', train=True, idx=range(ntrain),
                          download=True, transform=transform_train)
val_set = CIFAR10Subset(root='./data', train=True, idx=range(ntrain, 50_000),
                        download=True, transform=transform_eval)
test_set = torchvision.datasets.CIFAR10(root='./data', train=False,
                                        download=True, transform=transform_eval)
```

```
Files already downloaded and verified
Files already downloaded and verified
Files already downloaded and verified
```

In [15]:

```
dataloaders = {}
dataloaders['train'] = torch.utils.data.DataLoader(train_set, batch_size=128,
                                                   shuffle=True, num_workers=0,
                                                   pin_memory=True)
dataloaders['val'] = torch.utils.data.DataLoader(val_set, batch_size=128,
                                                 shuffle=False, num_workers=0,
                                                 pin_memory=True)
dataloaders['test'] = torch.utils.data.DataLoader(test_set, batch_size=128,
                                                  shuffle=False, num_workers=0,
                                                  pin_memory=True)
```

Next we push the model to our GPU (if there is one).

Next we define a helper method that does one epoch of training or evaluation. We have only defined training here, so you need to implement the necessary changes for evaluation!

In [17]:

```
device = torch.device('cuda') if torch.cuda.is_available() else torch.device('cpu')

resnet.to(device);
print(resnet)
```

```
Sequential(
  (0): Conv2d(3, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (2): ReLU()
  (3): ResidualStack(
    (modules1): ModuleList(
      (0): ResidualBlock(
        (skip): Sequential()
        (conv1): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (relu): ReLU()
        (bn1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv2): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
      (1): ResidualBlock(
        (skip): Sequential()
        (conv1): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (relu): ReLU()
        (bn1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv2): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
      (2): ResidualBlock(
        (skip): Sequential()
        (conv1): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (relu): ReLU()
        (bn1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv2): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
      (3): ResidualBlock(
        (skip): Sequential()
        (conv1): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (relu): ReLU()
        (bn1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv2): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
      (4): ResidualBlock(
        (skip): Sequential()
        (conv1): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (relu): ReLU()
```

```
      (bn1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (module_list): ModuleList(
    (0): ResidualBlock(
      (skip): Sequential()
      (conv1): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (relu): ReLU()
      (bn1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (1): ResidualBlock(
      (skip): Sequential()
      (conv1): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (relu): ReLU()
      (bn1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (2): ResidualBlock(
      (skip): Sequential()
      (conv1): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (relu): ReLU()
      (bn1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (3): ResidualBlock(
      (skip): Sequential()
      (conv1): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (relu): ReLU()
      (bn1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (4): ResidualBlock(
      (skip): Sequential()
      (conv1): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (relu): ReLU()
      (bn1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
)
(4): ResidualStack(
  (modules1): ModuleList(
    (0): ResidualBlock(
      (skip): Lambda()
      (conv1): Conv2d(16, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (relu): ReLU()
      (bn1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (1): ResidualBlock(
      (skip): Sequential()
      (conv1): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (relu): ReLU()
      (bn1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (2): ResidualBlock(
      (skip): Sequential()
      (conv1): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (relu): ReLU()
      (bn1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (3): ResidualBlock(
      (skip): Sequential()
      (conv1): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (relu): ReLU()
      (bn1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (4): ResidualBlock(
```

```
      (skip): Sequential()
      (conv1): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (relu): ReLU()
      (bn1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (module_list): ModuleList(
    (0): ResidualBlock(
      (skip): Lambda()
      (conv1): Conv2d(16, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (relu): ReLU()
      (bn1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (1): ResidualBlock(
      (skip): Sequential()
      (conv1): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (relu): ReLU()
      (bn1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (2): ResidualBlock(
      (skip): Sequential()
      (conv1): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (relu): ReLU()
      (bn1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (3): ResidualBlock(
      (skip): Sequential()
      (conv1): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (relu): ReLU()
      (bn1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (4): ResidualBlock(
      (skip): Sequential()
      (conv1): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (relu): ReLU()
      (bn1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
)
(5): ResidualStack(
  (modules1): ModuleList(
    (0): ResidualBlock(
      (skip): Lambda()
      (conv1): Conv2d(32, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (relu): ReLU()
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (1): ResidualBlock(
      (skip): Sequential()
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (relu): ReLU()
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (2): ResidualBlock(
      (skip): Sequential()
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (relu): ReLU()
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (3): ResidualBlock(
      (skip): Sequential()
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (relu): ReLU()
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
```

```
        (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
      (4): ResidualBlock(
        (skip): Sequential()
        (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (relu): ReLU()
        (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (module_list): ModuleList(
      (0): ResidualBlock(
        (skip): Lambda()
        (conv1): Conv2d(32, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
        (relu): ReLU()
        (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
      (1): ResidualBlock(
        (skip): Sequential()
        (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (relu): ReLU()
        (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
      (2): ResidualBlock(
        (skip): Sequential()
        (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (relu): ReLU()
        (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
      (3): ResidualBlock(
        (skip): Sequential()
        (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (relu): ReLU()
        (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
      (4): ResidualBlock(
        (skip): Sequential()
        (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (relu): ReLU()
        (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
  )
  (6): AdaptiveAvgPool2d(output_size=1)
  (7): Flatten(start_dim=1, end_dim=-1)
  (8): Linear(in_features=64, out_features=10, bias=True)
  (9): Softmax(dim=-1)
)
```

In [19]:

```python
def run_epoch(model, optimizer, dataloader, train):
    """
    Run one epoch of training or evaluation.

    Args:
        model: The model used for prediction
        optimizer: Optimization algorithm for the model
        dataloader: Dataloader providing the data to run our model on
        train: Whether this epoch is used for training or evaluation

    Returns:
        Loss and accuracy in this epoch.
    """
    # TODO: Change the necessary parts to work correctly during evaluation (train=False)

    device = next(model.parameters()).device

    # Set model to training mode (for e.g. batch normalization, dropout)


    epoch_loss = 0.0
    epoch_acc = 0.0

    # Iterate over data
    if train:
        model.train()
        for xb, yb in dataloader:
            xb, yb = xb.to(device), yb.to(device)

            # zero the parameter gradients
            optimizer.zero_grad()

            # forward
            with torch.set_grad_enabled(True):
                pred = model(xb)
                loss = F.cross_entropy(pred, yb)
                top1 = torch.argmax(pred, dim=1)
                ncorrect = torch.sum(top1 == yb)

                loss.backward()
                optimizer.step()

            # statistics
            epoch_loss += loss.item()
            epoch_acc += ncorrect.item()
    else:
        model.eval()
        for xb, yb in dataloader:
            xb, yb = xb.to(device), yb.to(device)

            # zero the parameter gradients


            # forward
            with torch.no_grad():
                pred = model(xb)
                loss = F.cross_entropy(pred, yb)
                top1 = torch.argmax(pred, dim=1)
                ncorrect = torch.sum(top1 == yb)



            # statistics
            epoch_loss += loss.item()
            epoch_acc += ncorrect.item()
    epoch_loss /= len(dataloader.dataset)
    epoch_acc /= len(dataloader.dataset)
    return epoch_loss, epoch_acc
```

Next we implement a method for fitting (training) our model. For many models early stopping can save a lot of training time. Your task is to add early stopping to the loop (based on validation accuracy). Early stopping usually means exiting the training loop if the validation accuracy hasn't improved for `patience` number of steps. Don't forget to save the best model parameters according to validation accuracy. You will need `copy.deepcopy` and the `state_dict` for this.

In [20]:

```
def fit(model, optimizer, lr_scheduler, dataloaders, max_epochs, patience):
    """
    Fit the given model on the dataset.

    Args:
        model: The model used for prediction
        optimizer: Optimization algorithm for the model
        lr_scheduler: Learning rate scheduler that improves training
                      in late epochs with learning rate decay
        dataloaders: Dataloaders for training and validation
        max_epochs: Maximum number of epochs for training
        patience: Number of epochs to wait with early stopping the
                  training if validation loss has decreased

    Returns:
        Loss and accuracy in this epoch.
    """

    best_acc = 0
    curr_patience = 0

    for epoch in range(max_epochs):
        train_loss, train_acc = run_epoch(model, optimizer, dataloaders['train'], train=True)
        lr_scheduler.step()
        print(f"Epoch {epoch + 1: >3}/{max_epochs}, train loss: {train_loss:.2e}, accuracy: {train_acc * 100:.2f}
%")

        val_loss, val_acc = run_epoch(model, None, dataloaders['val'], train=False)
        print(f"Epoch {epoch + 1: >3}/{max_epochs}, val loss: {val_loss:.2e}, accuracy: {val_acc * 100:.2f}%")

        # TODO: Add early stopping and save the best weights (in best_model_weights)
        if val_acc>best_acc:
            best_acc=val_acc
            best_model_weights=copy.deepcopy(model.state_dict())
            curr_patience=0
        else:
            curr_patience=curr_patience+1
        if curr_patience==patience:
            break

    model.load_state_dict(best_model_weights)
```

In most cases you should just use the Adam optimizer for training, because it works well out of the box. However, a well-tuned SGD (with momentum) will in most cases outperform Adam. And since the original paper gives us a well-tuned SGD we will just use that.

In [21]:

```
optimizer = torch.optim.SGD(resnet.parameters(), lr=0.1, momentum=0.9, weight_decay=1e-4)
lr_scheduler = torch.optim.lr_scheduler.MultiStepLR(optimizer, milestones=[100, 150], gamma=0.1)

# Fit model
fit(resnet, optimizer, lr_scheduler, dataloaders, max_epochs=200, patience=50)
```

```
Epoch   1/200, train loss: 1.73e-02, accuracy: 23.90%
Epoch   1/200, val loss: 1.77e-02, accuracy: 24.40%
Epoch   2/200, train loss: 1.70e-02, accuracy: 28.42%
Epoch   2/200, val loss: 1.74e-02, accuracy: 27.82%
Epoch   3/200, train loss: 1.67e-02, accuracy: 31.39%
Epoch   3/200, val loss: 1.68e-02, accuracy: 35.62%
Epoch   4/200, train loss: 1.65e-02, accuracy: 34.62%
Epoch   4/200, val loss: 1.73e-02, accuracy: 28.70%
Epoch   5/200, train loss: 1.64e-02, accuracy: 36.18%
Epoch   5/200, val loss: 1.64e-02, accuracy: 40.32%
Epoch   6/200, train loss: 1.58e-02, accuracy: 44.15%
Epoch   6/200, val loss: 1.61e-02, accuracy: 44.84%
Epoch   7/200, train loss: 1.56e-02, accuracy: 46.74%
Epoch   7/200, val loss: 1.60e-02, accuracy: 45.38%
Epoch   8/200, train loss: 1.51e-02, accuracy: 52.48%
Epoch   8/200, val loss: 1.61e-02, accuracy: 44.00%
Epoch   9/200, train loss: 1.49e-02, accuracy: 54.80%
Epoch   9/200, val loss: 1.56e-02, accuracy: 51.40%
Epoch  10/200, train loss: 1.48e-02, accuracy: 56.42%
Epoch  10/200, val loss: 1.52e-02, accuracy: 55.92%
Epoch  11/200, train loss: 1.47e-02, accuracy: 57.75%
Epoch  11/200, val loss: 1.51e-02, accuracy: 56.56%
Epoch  12/200, train loss: 1.46e-02, accuracy: 58.99%
Epoch  12/200, val loss: 1.52e-02, accuracy: 55.10%
Epoch  13/200, train loss: 1.46e-02, accuracy: 59.90%
Epoch  13/200, val loss: 1.50e-02, accuracy: 58.26%
Epoch  14/200, train loss: 1.45e-02, accuracy: 60.78%
```

```
Epoch  14/200, val loss: 1.48e-02, accuracy: 60.54%
Epoch  15/200, train loss: 1.44e-02, accuracy: 61.74%
Epoch  15/200, val loss: 1.51e-02, accuracy: 57.88%
Epoch  16/200, train loss: 1.42e-02, accuracy: 64.69%
Epoch  16/200, val loss: 1.53e-02, accuracy: 54.62%
Epoch  17/200, train loss: 1.41e-02, accuracy: 65.67%
Epoch  17/200, val loss: 1.47e-02, accuracy: 62.40%
Epoch  18/200, train loss: 1.36e-02, accuracy: 72.02%
Epoch  18/200, val loss: 1.43e-02, accuracy: 68.14%
Epoch  19/200, train loss: 1.35e-02, accuracy: 73.74%
Epoch  19/200, val loss: 1.38e-02, accuracy: 73.68%
Epoch  20/200, train loss: 1.34e-02, accuracy: 74.60%
Epoch  20/200, val loss: 1.40e-02, accuracy: 70.78%
Epoch  21/200, train loss: 1.33e-02, accuracy: 75.76%
Epoch  21/200, val loss: 1.38e-02, accuracy: 73.18%
Epoch  22/200, train loss: 1.33e-02, accuracy: 76.70%
Epoch  22/200, val loss: 1.41e-02, accuracy: 69.92%
Epoch  23/200, train loss: 1.32e-02, accuracy: 76.86%
Epoch  23/200, val loss: 1.35e-02, accuracy: 77.56%
Epoch  24/200, train loss: 1.32e-02, accuracy: 77.28%
Epoch  24/200, val loss: 1.37e-02, accuracy: 75.48%
Epoch  25/200, train loss: 1.32e-02, accuracy: 77.67%
Epoch  25/200, val loss: 1.36e-02, accuracy: 76.00%
Epoch  26/200, train loss: 1.31e-02, accuracy: 78.27%
Epoch  26/200, val loss: 1.34e-02, accuracy: 77.96%
Epoch  27/200, train loss: 1.31e-02, accuracy: 78.85%
Epoch  27/200, val loss: 1.35e-02, accuracy: 78.44%
Epoch  28/200, train loss: 1.31e-02, accuracy: 79.12%
Epoch  28/200, val loss: 1.35e-02, accuracy: 77.92%
Epoch  29/200, train loss: 1.30e-02, accuracy: 79.56%
Epoch  29/200, val loss: 1.33e-02, accuracy: 80.14%
Epoch  30/200, train loss: 1.30e-02, accuracy: 80.03%
Epoch  30/200, val loss: 1.39e-02, accuracy: 73.00%
Epoch  31/200, train loss: 1.30e-02, accuracy: 80.08%
Epoch  31/200, val loss: 1.34e-02, accuracy: 78.88%
Epoch  32/200, train loss: 1.30e-02, accuracy: 80.16%
Epoch  32/200, val loss: 1.35e-02, accuracy: 77.98%
Epoch  33/200, train loss: 1.30e-02, accuracy: 80.63%
Epoch  33/200, val loss: 1.33e-02, accuracy: 79.86%
Epoch  34/200, train loss: 1.29e-02, accuracy: 80.94%
Epoch  34/200, val loss: 1.34e-02, accuracy: 79.14%
Epoch  35/200, train loss: 1.29e-02, accuracy: 80.99%
Epoch  35/200, val loss: 1.32e-02, accuracy: 81.40%
Epoch  36/200, train loss: 1.29e-02, accuracy: 81.18%
Epoch  36/200, val loss: 1.36e-02, accuracy: 76.38%
Epoch  37/200, train loss: 1.29e-02, accuracy: 81.72%
Epoch  37/200, val loss: 1.32e-02, accuracy: 81.08%
Epoch  38/200, train loss: 1.29e-02, accuracy: 81.62%
Epoch  38/200, val loss: 1.34e-02, accuracy: 79.46%
Epoch  39/200, train loss: 1.29e-02, accuracy: 81.69%
Epoch  39/200, val loss: 1.42e-02, accuracy: 69.26%
Epoch  40/200, train loss: 1.28e-02, accuracy: 82.06%
Epoch  40/200, val loss: 1.33e-02, accuracy: 80.34%
Epoch  41/200, train loss: 1.28e-02, accuracy: 82.16%
Epoch  41/200, val loss: 1.34e-02, accuracy: 78.56%
Epoch  42/200, train loss: 1.28e-02, accuracy: 82.23%
Epoch  42/200, val loss: 1.34e-02, accuracy: 78.90%
Epoch  43/200, train loss: 1.28e-02, accuracy: 82.51%
Epoch  43/200, val loss: 1.34e-02, accuracy: 79.02%
Epoch  44/200, train loss: 1.28e-02, accuracy: 82.77%
Epoch  44/200, val loss: 1.36e-02, accuracy: 76.74%
Epoch  45/200, train loss: 1.28e-02, accuracy: 82.97%
Epoch  45/200, val loss: 1.33e-02, accuracy: 79.72%
Epoch  46/200, train loss: 1.28e-02, accuracy: 82.88%
Epoch  46/200, val loss: 1.36e-02, accuracy: 76.64%
Epoch  47/200, train loss: 1.27e-02, accuracy: 83.31%
Epoch  47/200, val loss: 1.32e-02, accuracy: 81.88%
Epoch  48/200, train loss: 1.27e-02, accuracy: 83.33%
Epoch  48/200, val loss: 1.33e-02, accuracy: 81.20%
Epoch  49/200, train loss: 1.28e-02, accuracy: 82.82%
Epoch  49/200, val loss: 1.34e-02, accuracy: 79.36%
Epoch  50/200, train loss: 1.27e-02, accuracy: 83.60%
Epoch  50/200, val loss: 1.35e-02, accuracy: 78.74%
Epoch  51/200, train loss: 1.28e-02, accuracy: 83.24%
Epoch  51/200, val loss: 1.36e-02, accuracy: 76.06%
Epoch  52/200, train loss: 1.27e-02, accuracy: 83.53%
Epoch  52/200, val loss: 1.33e-02, accuracy: 80.72%
Epoch  53/200, train loss: 1.27e-02, accuracy: 83.76%
Epoch  53/200, val loss: 1.32e-02, accuracy: 81.30%
Epoch  54/200, train loss: 1.27e-02, accuracy: 83.63%
Epoch  54/200, val loss: 1.34e-02, accuracy: 78.90%
Epoch  55/200, train loss: 1.27e-02, accuracy: 84.07%
Epoch  55/200, val loss: 1.32e-02, accuracy: 81.90%
```

```
Epoch  56/200, train loss: 1.27e-02, accuracy: 83.75%
Epoch  56/200, val loss: 1.34e-02, accuracy: 78.34%
Epoch  57/200, train loss: 1.27e-02, accuracy: 84.35%
Epoch  57/200, val loss: 1.31e-02, accuracy: 82.62%
Epoch  58/200, train loss: 1.27e-02, accuracy: 84.39%
Epoch  58/200, val loss: 1.32e-02, accuracy: 81.76%
Epoch  59/200, train loss: 1.27e-02, accuracy: 84.10%
Epoch  59/200, val loss: 1.34e-02, accuracy: 79.54%
Epoch  60/200, train loss: 1.27e-02, accuracy: 84.27%
Epoch  60/200, val loss: 1.31e-02, accuracy: 83.14%
Epoch  61/200, train loss: 1.27e-02, accuracy: 84.02%
Epoch  61/200, val loss: 1.31e-02, accuracy: 82.68%
Epoch  62/200, train loss: 1.26e-02, accuracy: 84.50%
Epoch  62/200, val loss: 1.35e-02, accuracy: 78.06%
Epoch  63/200, train loss: 1.26e-02, accuracy: 84.83%
Epoch  63/200, val loss: 1.34e-02, accuracy: 79.50%
Epoch  64/200, train loss: 1.26e-02, accuracy: 84.60%
Epoch  64/200, val loss: 1.35e-02, accuracy: 77.22%
Epoch  65/200, train loss: 1.26e-02, accuracy: 84.64%
Epoch  65/200, val loss: 1.32e-02, accuracy: 81.72%
Epoch  66/200, train loss: 1.26e-02, accuracy: 84.59%
Epoch  66/200, val loss: 1.33e-02, accuracy: 80.48%
Epoch  67/200, train loss: 1.27e-02, accuracy: 84.49%
Epoch  67/200, val loss: 1.31e-02, accuracy: 82.74%
Epoch  68/200, train loss: 1.27e-02, accuracy: 84.46%
Epoch  68/200, val loss: 1.32e-02, accuracy: 80.54%
Epoch  69/200, train loss: 1.26e-02, accuracy: 84.93%
Epoch  69/200, val loss: 1.31e-02, accuracy: 82.54%
Epoch  70/200, train loss: 1.26e-02, accuracy: 84.86%
Epoch  70/200, val loss: 1.36e-02, accuracy: 75.92%
Epoch  71/200, train loss: 1.26e-02, accuracy: 84.67%
Epoch  71/200, val loss: 1.31e-02, accuracy: 83.04%
Epoch  72/200, train loss: 1.26e-02, accuracy: 85.27%
Epoch  72/200, val loss: 1.32e-02, accuracy: 81.70%
Epoch  73/200, train loss: 1.26e-02, accuracy: 85.10%
Epoch  73/200, val loss: 1.33e-02, accuracy: 80.74%
Epoch  74/200, train loss: 1.26e-02, accuracy: 85.14%
Epoch  74/200, val loss: 1.32e-02, accuracy: 80.96%
Epoch  75/200, train loss: 1.26e-02, accuracy: 85.36%
Epoch  75/200, val loss: 1.32e-02, accuracy: 82.22%
Epoch  76/200, train loss: 1.26e-02, accuracy: 85.50%
Epoch  76/200, val loss: 1.32e-02, accuracy: 81.30%
Epoch  77/200, train loss: 1.26e-02, accuracy: 85.35%
Epoch  77/200, val loss: 1.32e-02, accuracy: 81.94%
Epoch  78/200, train loss: 1.26e-02, accuracy: 85.49%
Epoch  78/200, val loss: 1.32e-02, accuracy: 81.26%
Epoch  79/200, train loss: 1.26e-02, accuracy: 85.36%
Epoch  79/200, val loss: 1.32e-02, accuracy: 81.74%
Epoch  80/200, train loss: 1.26e-02, accuracy: 85.68%
Epoch  80/200, val loss: 1.30e-02, accuracy: 84.22%
Epoch  81/200, train loss: 1.26e-02, accuracy: 85.70%
Epoch  81/200, val loss: 1.30e-02, accuracy: 83.42%
Epoch  82/200, train loss: 1.26e-02, accuracy: 85.75%
Epoch  82/200, val loss: 1.34e-02, accuracy: 78.72%
Epoch  83/200, train loss: 1.26e-02, accuracy: 85.36%
Epoch  83/200, val loss: 1.32e-02, accuracy: 81.20%
Epoch  84/200, train loss: 1.26e-02, accuracy: 85.82%
Epoch  84/200, val loss: 1.33e-02, accuracy: 79.86%
Epoch  85/200, train loss: 1.26e-02, accuracy: 85.66%
Epoch  85/200, val loss: 1.30e-02, accuracy: 83.60%
Epoch  86/200, train loss: 1.26e-02, accuracy: 85.71%
Epoch  86/200, val loss: 1.30e-02, accuracy: 83.72%
Epoch  87/200, train loss: 1.25e-02, accuracy: 85.79%
Epoch  87/200, val loss: 1.31e-02, accuracy: 82.74%
Epoch  88/200, train loss: 1.25e-02, accuracy: 86.05%
Epoch  88/200, val loss: 1.35e-02, accuracy: 77.60%
Epoch  89/200, train loss: 1.25e-02, accuracy: 85.87%
Epoch  89/200, val loss: 1.32e-02, accuracy: 81.98%
Epoch  90/200, train loss: 1.25e-02, accuracy: 85.90%
Epoch  90/200, val loss: 1.30e-02, accuracy: 83.68%
Epoch  91/200, train loss: 1.25e-02, accuracy: 86.47%
Epoch  91/200, val loss: 1.34e-02, accuracy: 79.48%
Epoch  92/200, train loss: 1.26e-02, accuracy: 85.74%
Epoch  92/200, val loss: 1.32e-02, accuracy: 81.06%
Epoch  93/200, train loss: 1.25e-02, accuracy: 86.05%
Epoch  93/200, val loss: 1.31e-02, accuracy: 82.70%
Epoch  94/200, train loss: 1.25e-02, accuracy: 86.01%
Epoch  94/200, val loss: 1.34e-02, accuracy: 78.36%
Epoch  95/200, train loss: 1.25e-02, accuracy: 86.16%
Epoch  95/200, val loss: 1.31e-02, accuracy: 83.08%
Epoch  96/200, train loss: 1.25e-02, accuracy: 85.98%
Epoch  96/200, val loss: 1.32e-02, accuracy: 81.24%
Epoch  97/200, train loss: 1.25e-02, accuracy: 86.47%
```

```
Epoch  97/200, val loss: 1.31e-02, accuracy: 82.78%
Epoch  98/200, train loss: 1.25e-02, accuracy: 86.47%
Epoch  98/200, val loss: 1.29e-02, accuracy: 85.34%
Epoch  99/200, train loss: 1.25e-02, accuracy: 86.83%
Epoch  99/200, val loss: 1.33e-02, accuracy: 80.76%
Epoch 100/200, train loss: 1.25e-02, accuracy: 86.17%
Epoch 100/200, val loss: 1.30e-02, accuracy: 83.84%
Epoch 101/200, train loss: 1.23e-02, accuracy: 89.63%
Epoch 101/200, val loss: 1.26e-02, accuracy: 89.02%
Epoch 102/200, train loss: 1.22e-02, accuracy: 90.91%
Epoch 102/200, val loss: 1.26e-02, accuracy: 88.80%
Epoch 103/200, train loss: 1.21e-02, accuracy: 91.63%
Epoch 103/200, val loss: 1.26e-02, accuracy: 89.32%
Epoch 104/200, train loss: 1.21e-02, accuracy: 91.96%
Epoch 104/200, val loss: 1.26e-02, accuracy: 89.34%
Epoch 105/200, train loss: 1.21e-02, accuracy: 92.21%
Epoch 105/200, val loss: 1.26e-02, accuracy: 89.90%
Epoch 106/200, train loss: 1.20e-02, accuracy: 92.59%
Epoch 106/200, val loss: 1.25e-02, accuracy: 90.04%
Epoch 107/200, train loss: 1.20e-02, accuracy: 92.44%
Epoch 107/200, val loss: 1.26e-02, accuracy: 89.72%
Epoch 108/200, train loss: 1.20e-02, accuracy: 92.77%
Epoch 108/200, val loss: 1.25e-02, accuracy: 89.92%
Epoch 109/200, train loss: 1.20e-02, accuracy: 92.74%
Epoch 109/200, val loss: 1.25e-02, accuracy: 90.26%
Epoch 110/200, train loss: 1.20e-02, accuracy: 92.98%
Epoch 110/200, val loss: 1.25e-02, accuracy: 90.04%
Epoch 111/200, train loss: 1.20e-02, accuracy: 93.05%
Epoch 111/200, val loss: 1.25e-02, accuracy: 90.04%
Epoch 112/200, train loss: 1.20e-02, accuracy: 93.24%
Epoch 112/200, val loss: 1.25e-02, accuracy: 89.98%
Epoch 113/200, train loss: 1.20e-02, accuracy: 93.36%
Epoch 113/200, val loss: 1.25e-02, accuracy: 90.24%
Epoch 114/200, train loss: 1.20e-02, accuracy: 93.38%
Epoch 114/200, val loss: 1.25e-02, accuracy: 90.02%
Epoch 115/200, train loss: 1.20e-02, accuracy: 93.39%
Epoch 115/200, val loss: 1.25e-02, accuracy: 90.16%
Epoch 116/200, train loss: 1.20e-02, accuracy: 93.50%
Epoch 116/200, val loss: 1.25e-02, accuracy: 90.26%
Epoch 117/200, train loss: 1.19e-02, accuracy: 93.74%
Epoch 117/200, val loss: 1.25e-02, accuracy: 90.54%
Epoch 118/200, train loss: 1.19e-02, accuracy: 93.73%
Epoch 118/200, val loss: 1.25e-02, accuracy: 90.12%
Epoch 119/200, train loss: 1.19e-02, accuracy: 93.69%
Epoch 119/200, val loss: 1.25e-02, accuracy: 90.26%
Epoch 120/200, train loss: 1.19e-02, accuracy: 93.95%
Epoch 120/200, val loss: 1.25e-02, accuracy: 90.34%
Epoch 121/200, train loss: 1.19e-02, accuracy: 93.96%
Epoch 121/200, val loss: 1.25e-02, accuracy: 90.16%
Epoch 122/200, train loss: 1.19e-02, accuracy: 94.14%
Epoch 122/200, val loss: 1.25e-02, accuracy: 90.36%
Epoch 123/200, train loss: 1.19e-02, accuracy: 94.27%
Epoch 123/200, val loss: 1.25e-02, accuracy: 90.60%
Epoch 124/200, train loss: 1.19e-02, accuracy: 94.25%
Epoch 124/200, val loss: 1.25e-02, accuracy: 90.24%
Epoch 125/200, train loss: 1.19e-02, accuracy: 94.45%
Epoch 125/200, val loss: 1.25e-02, accuracy: 90.60%
Epoch 126/200, train loss: 1.19e-02, accuracy: 94.42%
Epoch 126/200, val loss: 1.25e-02, accuracy: 90.30%
Epoch 127/200, train loss: 1.19e-02, accuracy: 94.30%
Epoch 127/200, val loss: 1.25e-02, accuracy: 90.44%
Epoch 128/200, train loss: 1.19e-02, accuracy: 94.50%
Epoch 128/200, val loss: 1.25e-02, accuracy: 90.34%
Epoch 129/200, train loss: 1.19e-02, accuracy: 94.63%
Epoch 129/200, val loss: 1.25e-02, accuracy: 90.30%
Epoch 130/200, train loss: 1.19e-02, accuracy: 94.55%
Epoch 130/200, val loss: 1.25e-02, accuracy: 90.66%
Epoch 131/200, train loss: 1.19e-02, accuracy: 94.81%
Epoch 131/200, val loss: 1.25e-02, accuracy: 90.08%
Epoch 132/200, train loss: 1.19e-02, accuracy: 94.88%
Epoch 132/200, val loss: 1.25e-02, accuracy: 90.42%
Epoch 133/200, train loss: 1.19e-02, accuracy: 94.78%
Epoch 133/200, val loss: 1.25e-02, accuracy: 90.54%
Epoch 134/200, train loss: 1.18e-02, accuracy: 94.90%
Epoch 134/200, val loss: 1.25e-02, accuracy: 90.74%
Epoch 135/200, train loss: 1.18e-02, accuracy: 94.98%
Epoch 135/200, val loss: 1.25e-02, accuracy: 90.54%
Epoch 136/200, train loss: 1.19e-02, accuracy: 94.79%
Epoch 136/200, val loss: 1.25e-02, accuracy: 90.62%
Epoch 137/200, train loss: 1.18e-02, accuracy: 94.92%
Epoch 137/200, val loss: 1.25e-02, accuracy: 90.78%
Epoch 138/200, train loss: 1.18e-02, accuracy: 95.00%
Epoch 138/200, val loss: 1.25e-02, accuracy: 90.68%
```

```
Epoch 139/200, train loss: 1.18e-02, accuracy: 95.15%
Epoch 139/200, val loss: 1.25e-02, accuracy: 90.32%
Epoch 140/200, train loss: 1.18e-02, accuracy: 95.14%
Epoch 140/200, val loss: 1.25e-02, accuracy: 90.68%
Epoch 141/200, train loss: 1.18e-02, accuracy: 95.20%
Epoch 141/200, val loss: 1.25e-02, accuracy: 90.82%
Epoch 142/200, train loss: 1.18e-02, accuracy: 95.16%
Epoch 142/200, val loss: 1.25e-02, accuracy: 90.44%
Epoch 143/200, train loss: 1.18e-02, accuracy: 95.13%
Epoch 143/200, val loss: 1.24e-02, accuracy: 91.12%
Epoch 144/200, train loss: 1.18e-02, accuracy: 95.34%
Epoch 144/200, val loss: 1.25e-02, accuracy: 90.78%
Epoch 145/200, train loss: 1.18e-02, accuracy: 95.34%
Epoch 145/200, val loss: 1.25e-02, accuracy: 90.70%
Epoch 146/200, train loss: 1.18e-02, accuracy: 95.37%
Epoch 146/200, val loss: 1.25e-02, accuracy: 90.58%
Epoch 147/200, train loss: 1.18e-02, accuracy: 95.34%
Epoch 147/200, val loss: 1.25e-02, accuracy: 90.36%
Epoch 148/200, train loss: 1.18e-02, accuracy: 95.44%
Epoch 148/200, val loss: 1.25e-02, accuracy: 90.56%
Epoch 149/200, train loss: 1.18e-02, accuracy: 95.58%
Epoch 149/200, val loss: 1.25e-02, accuracy: 91.10%
Epoch 150/200, train loss: 1.18e-02, accuracy: 95.54%
Epoch 150/200, val loss: 1.25e-02, accuracy: 90.72%
Epoch 151/200, train loss: 1.18e-02, accuracy: 95.88%
Epoch 151/200, val loss: 1.24e-02, accuracy: 91.32%
Epoch 152/200, train loss: 1.18e-02, accuracy: 95.93%
Epoch 152/200, val loss: 1.24e-02, accuracy: 91.50%
Epoch 153/200, train loss: 1.18e-02, accuracy: 96.12%
Epoch 153/200, val loss: 1.24e-02, accuracy: 91.38%
Epoch 154/200, train loss: 1.17e-02, accuracy: 96.20%
Epoch 154/200, val loss: 1.24e-02, accuracy: 91.44%
Epoch 155/200, train loss: 1.17e-02, accuracy: 96.22%
Epoch 155/200, val loss: 1.24e-02, accuracy: 91.42%
Epoch 156/200, train loss: 1.17e-02, accuracy: 96.26%
Epoch 156/200, val loss: 1.24e-02, accuracy: 91.36%
Epoch 157/200, train loss: 1.17e-02, accuracy: 96.29%
Epoch 157/200, val loss: 1.24e-02, accuracy: 91.42%
Epoch 158/200, train loss: 1.17e-02, accuracy: 96.34%
Epoch 158/200, val loss: 1.24e-02, accuracy: 91.42%
Epoch 159/200, train loss: 1.17e-02, accuracy: 96.28%
Epoch 159/200, val loss: 1.25e-02, accuracy: 91.20%
Epoch 160/200, train loss: 1.17e-02, accuracy: 96.38%
Epoch 160/200, val loss: 1.24e-02, accuracy: 91.38%
Epoch 161/200, train loss: 1.17e-02, accuracy: 96.38%
Epoch 161/200, val loss: 1.24e-02, accuracy: 91.36%
Epoch 162/200, train loss: 1.17e-02, accuracy: 96.34%
Epoch 162/200, val loss: 1.24e-02, accuracy: 91.22%
Epoch 163/200, train loss: 1.17e-02, accuracy: 96.44%
Epoch 163/200, val loss: 1.24e-02, accuracy: 91.22%
Epoch 164/200, train loss: 1.17e-02, accuracy: 96.43%
Epoch 164/200, val loss: 1.24e-02, accuracy: 91.24%
Epoch 165/200, train loss: 1.17e-02, accuracy: 96.54%
Epoch 165/200, val loss: 1.24e-02, accuracy: 91.50%
Epoch 166/200, train loss: 1.17e-02, accuracy: 96.48%
Epoch 166/200, val loss: 1.24e-02, accuracy: 91.30%
Epoch 167/200, train loss: 1.17e-02, accuracy: 96.35%
Epoch 167/200, val loss: 1.24e-02, accuracy: 91.30%
Epoch 168/200, train loss: 1.17e-02, accuracy: 96.54%
Epoch 168/200, val loss: 1.24e-02, accuracy: 91.22%
Epoch 169/200, train loss: 1.17e-02, accuracy: 96.50%
Epoch 169/200, val loss: 1.24e-02, accuracy: 91.44%
Epoch 170/200, train loss: 1.17e-02, accuracy: 96.52%
Epoch 170/200, val loss: 1.24e-02, accuracy: 91.30%
Epoch 171/200, train loss: 1.17e-02, accuracy: 96.44%
Epoch 171/200, val loss: 1.24e-02, accuracy: 91.46%
Epoch 172/200, train loss: 1.17e-02, accuracy: 96.60%
Epoch 172/200, val loss: 1.24e-02, accuracy: 91.28%
Epoch 173/200, train loss: 1.17e-02, accuracy: 96.47%
Epoch 173/200, val loss: 1.24e-02, accuracy: 91.28%
Epoch 174/200, train loss: 1.17e-02, accuracy: 96.50%
Epoch 174/200, val loss: 1.24e-02, accuracy: 91.28%
Epoch 175/200, train loss: 1.17e-02, accuracy: 96.54%
Epoch 175/200, val loss: 1.24e-02, accuracy: 91.28%
Epoch 176/200, train loss: 1.17e-02, accuracy: 96.57%
Epoch 176/200, val loss: 1.24e-02, accuracy: 91.54%
Epoch 177/200, train loss: 1.17e-02, accuracy: 96.64%
Epoch 177/200, val loss: 1.24e-02, accuracy: 91.30%
Epoch 178/200, train loss: 1.17e-02, accuracy: 96.50%
Epoch 178/200, val loss: 1.24e-02, accuracy: 91.28%
Epoch 179/200, train loss: 1.17e-02, accuracy: 96.59%
Epoch 179/200, val loss: 1.24e-02, accuracy: 91.12%
Epoch 180/200, train loss: 1.17e-02, accuracy: 96.55%
```

```
Epoch 180/200, val loss: 1.24e-02, accuracy: 91.22%
Epoch 181/200, train loss: 1.17e-02, accuracy: 96.60%
Epoch 181/200, val loss: 1.24e-02, accuracy: 91.10%
Epoch 182/200, train loss: 1.17e-02, accuracy: 96.60%
Epoch 182/200, val loss: 1.24e-02, accuracy: 91.24%
Epoch 183/200, train loss: 1.17e-02, accuracy: 96.59%
Epoch 183/200, val loss: 1.24e-02, accuracy: 91.30%
Epoch 184/200, train loss: 1.17e-02, accuracy: 96.61%
Epoch 184/200, val loss: 1.24e-02, accuracy: 91.30%
Epoch 185/200, train loss: 1.17e-02, accuracy: 96.63%
Epoch 185/200, val loss: 1.24e-02, accuracy: 91.24%
Epoch 186/200, train loss: 1.17e-02, accuracy: 96.78%
Epoch 186/200, val loss: 1.24e-02, accuracy: 91.28%
Epoch 187/200, train loss: 1.17e-02, accuracy: 96.63%
Epoch 187/200, val loss: 1.24e-02, accuracy: 91.32%
Epoch 188/200, train loss: 1.17e-02, accuracy: 96.62%
Epoch 188/200, val loss: 1.24e-02, accuracy: 91.30%
Epoch 189/200, train loss: 1.17e-02, accuracy: 96.70%
Epoch 189/200, val loss: 1.24e-02, accuracy: 91.40%
Epoch 190/200, train loss: 1.17e-02, accuracy: 96.80%
Epoch 190/200, val loss: 1.24e-02, accuracy: 91.46%
Epoch 191/200, train loss: 1.17e-02, accuracy: 96.68%
Epoch 191/200, val loss: 1.24e-02, accuracy: 91.44%
Epoch 192/200, train loss: 1.17e-02, accuracy: 96.70%
Epoch 192/200, val loss: 1.24e-02, accuracy: 91.52%
Epoch 193/200, train loss: 1.17e-02, accuracy: 96.72%
Epoch 193/200, val loss: 1.24e-02, accuracy: 91.56%
Epoch 194/200, train loss: 1.17e-02, accuracy: 96.67%
Epoch 194/200, val loss: 1.24e-02, accuracy: 91.54%
Epoch 195/200, train loss: 1.17e-02, accuracy: 96.66%
Epoch 195/200, val loss: 1.24e-02, accuracy: 91.18%
Epoch 196/200, train loss: 1.17e-02, accuracy: 96.84%
Epoch 196/200, val loss: 1.24e-02, accuracy: 91.40%
Epoch 197/200, train loss: 1.17e-02, accuracy: 96.69%
Epoch 197/200, val loss: 1.24e-02, accuracy: 91.42%
Epoch 198/200, train loss: 1.17e-02, accuracy: 96.82%
Epoch 198/200, val loss: 1.24e-02, accuracy: 91.72%
Epoch 199/200, train loss: 1.17e-02, accuracy: 96.75%
Epoch 199/200, val loss: 1.24e-02, accuracy: 91.66%
Epoch 200/200, train loss: 1.17e-02, accuracy: 96.63%
Epoch 200/200, val loss: 1.24e-02, accuracy: 91.64%
```

Once the model is trained we run it on the test set to obtain our final accuracy. Note that we can only look at the test set once, everything else would lead to overfitting. So you *must* ignore the test set while developing your model!

In [22]:

```
test_loss, test_acc = run_epoch(resnet, None, dataloaders['test'], train=False)
print(f"Test loss: {test_loss:.1e}, accuracy: {test_acc * 100:.2f}%")
```

```
Test loss: 1.2e-02, accuracy: 90.95%
```

That's almost what was reported in the paper (92.49%) and we didn't even train on the full training set.

Optional task: Squeeze out all the juice!

Can you do even better? Have a look at [A Recipe for Training Neural Networks (https://karpathy.github.io/2019/04/25/recipe/)](https://karpathy.github.io/2019/04/25/recipe/) and some state-of-the-art architectures such as [EfficientNet architecture (https://ai.googleblog.com/2019/05/efficientnet-improving-accuracy-and.html)](https://ai.googleblog.com/2019/05/efficientnet-improving-accuracy-and.html). Play around with the possibilities PyTorch offers you and see how close you can get to the [state of the art on CIFAR-10 (https://paperswithcode.com/sota/image-classification-on-cifar-10)](https://paperswithcode.com/sota/image-classification-on-cifar-10).