Exercise

10

TUM Department of Informatics

Supervised by Prof. Dr. Stephan Günnemann

Informatics 3 - Professorship of Data Mining and Analytics

Submitted by Marcel Bruckner (03674122)

Julian Hohenadel (03673879)

Kevin Bein (03707775)

Submission date Munich, January 2, 2020

Deep Learning II

Problem 1:

a) Since all weights are positive, $\max(0, {W_j^*}^T x_i) = {W_j^*}^T x_i$ and the following holds:

$$
\begin{aligned}
\mathcal{L}_{NN}(W_{NN}^*) &= \frac{1}{2}\sum_{i=1}^{N}(-y_i)^2 \\
&= \frac{1}{2}\sum_{i=1}^{N}({W_{L+1}^*}^T({W_L^*}^T \max(0, \dots (0, {W_2^*}^T \max(0, {W_1^*}^T x_i)))) - y_i)^2 \\
&= \frac{1}{2}\sum_{i=1}^{N}({W_{L+1}^*}^T({W_L^*}^T (\dots ({W_2^*}^T ({W_1^*}^T x_i)))) - y_i)^2 \\
&= \frac{1}{2}\sum_{i=1}^{N}({W_{L+1}^*}^T \cdot \ldots \cdot {W_1^*}^T x_i - y_i)^2 \\
&= \frac{1}{2}\sum_{i=1}^{N}({W_{NN}^*}^T x_i - y_i)^2
\end{aligned}
$$

Since the solutions W_{NN}^* and w_{LS}^* are a global optimum, we can set $W_{NN}^* = w_{LS}^*$ and we get $\mathcal{L}_{NN}(W_{NN}^*) = \mathcal{L}_{LS}(w_{LS}^*)$.

b) Since simple linear regression is a special case of a Feed-Forward Neural Network, the Loss is the same. This does not work the other way around because NNs can learn more complex functions. w_{LS}^* being non-negative does not imply anything about the optimal weights of the network W_{NN}^*.Therefore we can conclude that $\mathcal{L}_{NN}(W_{NN}^*) \leq \mathcal{L}_{LS}(w_{LS}^*)$.

Problem 2:

exercise_10_notebook_90_36

January 5, 2020

[0]:
```python
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
```

[2]:
```
!nvidia-smi
```

```
Fri Jan  3 19:33:22 2020
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 440.44       Driver Version: 418.67       CUDA Version: 10.1     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Tesla P100-PCIE...  Off  | 00000000:00:04.0 Off |                    0 |
| N/A   35C    P0    28W / 250W |      0MiB / 16280MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

1 PyTorch

In this notebook you will gain some hands-on experience with PyTorch, one of the major frameworks for deep learning. To install PyTorch run `conda install pytorch torchvision cudatoolkit=10.1 -c pytorch`, with cudatoolkit set to whichever CUDA version you have installed. You can check this by running `nvcc --version`. If you do not have an Nvidia GPU you can run `conda install pytorch torchvision cpuonly -c pytorch` instead. However, in this case we recommend using Google Colab.

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.

2 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.

[0]:
```
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
        """
        if self.training:
            mask = np.random.random(input.shape)
            return torch.from_numpy(np.where(mask <= self.p, 0, (1 / (1 - self.
↪p))))

        # TODO: Set values randomly to 0.
```

[4]:
```python
# Test dropout
test = torch.ones(10_000)
dropout = Dropout(0.5)
test_dropped = dropout(test)

print(test_dropped.sum().item())
print((test_dropped > 0).sum().item())

print(np.isclose(test_dropped.sum().item(), 10_000, atol=400))
print(np.isclose((test_dropped > 0).sum().item(), 5_000, atol=200))

# These assertions can in principle fail due to bad luck, but
# if implemented correctly they should almost always succeed.
assert np.isclose(test_dropped.sum().item(), 10_000, atol=400)
assert np.isclose((test_dropped > 0).sum().item(), 5_000, atol=200)
```

```
10104.0
5052
True
True
```

3 2. Batch normalization

Batch normalization is a trick used 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.

[0]:
```python
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__()

        # TODO: Initialize the required parameters
```

```
        self.gamma = nn.Parameter(torch.ones(num_features)).unsqueeze(0).
↪unsqueeze(-1)
        #self.gamma = self.gamma.
        self.beta = nn.Parameter(torch.zeros(num_features)).unsqueeze(0).
↪unsqueeze(-1)
        #self.beta = self.beta.

    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
        """
        eps = 1e-5

        mean = input.mean(dim=[0, 2], keepdim=True)
        input_mean_norm = input - mean
        var = torch.sqrt(input.var(dim=[0, 2],  keepdim=True))

        return (input_mean_norm / (var + eps)) * self.gamma + self.beta

        # TODO: Implement the required transformation
```

[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)

print(test_b1)
print("-----")
print(test_b2)

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

```
tensor([[[ 1.6380,  1.2470,  0.7253, -1.9484],
         [ 0.6971, -1.2133, -0.0234, -1.5829]],
```

```
        [[-0.7447,  1.3905, -0.4249, -1.3242],
         [-0.7073, -0.5391, -0.7482,  0.7810]],

        [[ 1.3849, -0.2178, -0.5182,  0.3152],
         [-0.7375,  1.0964,  0.8193,  1.6979]],

        [[ 1.0618,  1.0772,  0.4671,  1.1113],
         [-0.2117,  0.0613, -0.2317,  0.8782]],

        [[-1.3073, -0.8507, -0.2745,  1.4516],
         [ 0.3380, -0.4044,  0.3249, -0.7540]],

        [[-1.4611,  0.8097, -0.8583, -0.6105],
         [-1.2529,  2.1395, -1.2134, -0.4677]],

        [[-0.8886, -0.6611, -0.0064,  0.3918],
         [-0.4678,  1.2093, -0.7933, -0.7154]],

        [[-1.3238, -0.0438, -0.1323,  0.5251],
         [-0.0781,  1.8616, -1.1634,  1.4012]]], grad_fn=<AddBackward0>)
-----
tensor([[[ 1.6642,  1.2669,  0.7369, -1.9796],
         [ 0.7082, -1.2327, -0.0238, -1.6083]],

        [[-0.7567,  1.4128, -0.4317, -1.3454],
         [-0.7186, -0.5477, -0.7602,  0.7935]],

        [[ 1.4070, -0.2212, -0.5265,  0.3202],
         [-0.7493,  1.1140,  0.8324,  1.7251]],

        [[ 1.0788,  1.0945,  0.4746,  1.1291],
         [-0.2151,  0.0622, -0.2354,  0.8923]],

        [[-1.3282, -0.8643, -0.2789,  1.4748],
         [ 0.3434, -0.4109,  0.3301, -0.7660]],

        [[-1.4845,  0.8227, -0.8720, -0.6202],
         [-1.2729,  2.1737, -1.2328, -0.4752]],

        [[-0.9028, -0.6717, -0.0065,  0.3981],
         [-0.4753,  1.2287, -0.8060, -0.7269]],

        [[-1.3450, -0.0445, -0.1344,  0.5335],
         [-0.0794,  1.8914, -1.1820,  1.4237]]])
True
```

4 3. ResNet

ResNet is the model 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 with optimized residual blocks. Here we will implement the original version for CIFAR-10.

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

[0]:
```python
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:

attachment:residual_connection.png

Note that we use 'SAME' padding, no bias, and batch normalization after each convolution.

[0]:
```python
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()

        # TODO: Initialize the required layers
```

```
        #self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3,␣
↪padding="SAME", bias=False, stride=stride)
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3,␣
↪padding=1, bias=False, stride=stride, groups=1)
        self.bn1 = nn.BatchNorm2d(out_channels) #BatchNorm(out_channels)
        #self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3,␣
↪padding="SAME", bias=False, stride=stride)
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3,␣
↪padding=1, bias=False, stride=1, groups=1)
        self.bn2 = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU()

    def forward(self, input):
        # TODO: Execute the required layers and functions
        #print("input.shape:", input.shape)
        identity = self.skip(input)
        #print("identity.shape:", identity.shape)

        out = self.conv1(input)
        out = self.bn1(out)
        out = self.relu(out)
        #print("out1.shape:", out.shape)

        out = self.conv2(out)
        out = self.bn2(out)

        #print("out2.shape:", out.shape)
        out += identity
        out = self.relu(out)

        #print("out.shape:", out.shape)
        #print("")
        return out
```

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.

[0]:
```
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__()

        # TODO: Initialize the required layers (blocks)
        self.layers = nn.ModuleList([ResidualBlock(in_channels, out_channels,␣
↪stride)])
        for i in range(1, num_blocks):
          self.layers.append(ResidualBlock(out_channels, out_channels, 1))

    def forward(self, input):
        # TODO: Execute the layers (blocks)
        #print("Execute " + str(len(self.layers)) + " ResidualBlock layers")
        for layer in self.layers:
          input = layer(input)
        return input
```

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):

attachment:resnet_cifar10_description.png

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.

[0]:
```
n = 5
num_classes = 10

class Squeeze(torch.nn.Module):
    def forward(self, input):
      out = input.squeeze()
      return out

# TODO: Implement ResNet via nn.Sequential
resnet = nn.Sequential(
            ResidualStack(3, 16, 1, 1),
            ResidualStack(16, 16, 1, 2 * n),
            ResidualStack(16, 32, 2, 2 * n),
            ResidualStack(32, 64, 2, 2 * n),
            nn.AdaptiveAvgPool2d(1),
            #nn.AdaptiveAvgPool2d(64),
            Squeeze(),
            nn.Linear(64, num_classes)
          )
```

Next we need to initialize the weights of our model.

[0]:
```python
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);
```

5 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.

[0]:
```python
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.

[0]:
```python
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
])
```

[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)
```

```
0it [00:00, ?it/s]

Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to
./data/cifar-10-python.tar.gz

170500096it [00:06, 27285630.82it/s]

Extracting ./data/cifar-10-python.tar.gz to ./data
Files already downloaded and verified
Files already downloaded and verified
```

[0]:
```
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).

[16]:
```
device = torch.device('cuda') if torch.cuda.is_available() else torch.
 ↪device('cpu')
print(torch.cuda.is_available())
resnet.to(device);
```

```
True
```

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!

[0]:
```
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)
    model.train()

    epoch_loss = 0.0
    epoch_acc = 0.0

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

        # zero the parameter gradients
        if train != False:
            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)

            if train != False:
                loss.backward()
                optimizer.step()

        # 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)! And don't forget to save the best model parameters according to validation accuracy. You will need `copy.deepcopy` and the `state_dict` for this.

[0]:
```python
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 = 0.0
    curr_patience = 0
    best_model_weights = None
    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)
        curr_patience = 1 if val_acc < best_acc else curr_patience + 1

        updated_model_weights = False
        if best_acc < val_acc:
            best_model_weights = copy.deepcopy(model.state_dict())
            updated_model_weights = True
        best_acc = max(val_acc, best_acc)
```

```
        print(f"Updates: curr_patience={curr_patience}, best_acc={best_acc},␣
↪updated={updated_model_weights}")

        if curr_patience > patience or epoch + 1 >= max_epochs:
            print(f"Stop early: curr_patience > patience or epoch + 1 >=␣
↪max_epochs")
            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.

[19]:
```
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: 1.80e-02, accuracy: 18.42%
Epoch   1/200, val loss: 1.49e-02, accuracy: 29.16%
Updates: curr_patience=1, best_acc=0.2916, updated=True
Epoch   2/200, train loss: 1.33e-02, accuracy: 36.39%
Epoch   2/200, val loss: 1.22e-02, accuracy: 43.74%
Updates: curr_patience=2, best_acc=0.4374, updated=True
Epoch   3/200, train loss: 1.10e-02, accuracy: 48.72%
Epoch   3/200, val loss: 1.02e-02, accuracy: 56.32%
Updates: curr_patience=3, best_acc=0.5632, updated=True
Epoch   4/200, train loss: 9.33e-03, accuracy: 57.26%
Epoch   4/200, val loss: 8.92e-03, accuracy: 61.78%
Updates: curr_patience=4, best_acc=0.6178, updated=True
Epoch   5/200, train loss: 8.20e-03, accuracy: 62.74%
Epoch   5/200, val loss: 8.05e-03, accuracy: 66.16%
Updates: curr_patience=5, best_acc=0.6616, updated=True
Epoch   6/200, train loss: 7.22e-03, accuracy: 67.73%
Epoch   6/200, val loss: 7.06e-03, accuracy: 70.32%
Updates: curr_patience=6, best_acc=0.7032, updated=True
Epoch   7/200, train loss: 6.42e-03, accuracy: 71.36%
Epoch   7/200, val loss: 6.53e-03, accuracy: 73.50%
Updates: curr_patience=7, best_acc=0.735, updated=True
Epoch   8/200, train loss: 5.82e-03, accuracy: 74.03%
Epoch   8/200, val loss: 6.09e-03, accuracy: 74.92%
Updates: curr_patience=8, best_acc=0.7492, updated=True
Epoch   9/200, train loss: 5.37e-03, accuracy: 76.19%
```

```
Epoch   9/200, val loss: 5.90e-03, accuracy: 76.60%
Updates: curr_patience=9, best_acc=0.766, updated=True
Epoch  10/200, train loss: 4.98e-03, accuracy: 77.90%
Epoch  10/200, val loss: 5.14e-03, accuracy: 78.54%
Updates: curr_patience=10, best_acc=0.7854, updated=True
Epoch  11/200, train loss: 4.67e-03, accuracy: 79.30%
Epoch  11/200, val loss: 4.94e-03, accuracy: 79.10%
Updates: curr_patience=11, best_acc=0.791, updated=True
Epoch  12/200, train loss: 4.41e-03, accuracy: 80.37%
Epoch  12/200, val loss: 5.12e-03, accuracy: 79.14%
Updates: curr_patience=12, best_acc=0.7914, updated=True
Epoch  13/200, train loss: 4.18e-03, accuracy: 81.44%
Epoch  13/200, val loss: 4.56e-03, accuracy: 80.24%
Updates: curr_patience=13, best_acc=0.8024, updated=True
Epoch  14/200, train loss: 3.97e-03, accuracy: 82.44%
Epoch  14/200, val loss: 4.46e-03, accuracy: 81.68%
Updates: curr_patience=14, best_acc=0.8168, updated=True
Epoch  15/200, train loss: 3.85e-03, accuracy: 82.88%
Epoch  15/200, val loss: 4.50e-03, accuracy: 81.58%
Updates: curr_patience=1, best_acc=0.8168, updated=False
Epoch  16/200, train loss: 3.64e-03, accuracy: 83.89%
Epoch  16/200, val loss: 4.29e-03, accuracy: 82.32%
Updates: curr_patience=2, best_acc=0.8232, updated=True
Epoch  17/200, train loss: 3.46e-03, accuracy: 84.62%
Epoch  17/200, val loss: 4.17e-03, accuracy: 83.06%
Updates: curr_patience=3, best_acc=0.8306, updated=True
Epoch  18/200, train loss: 3.39e-03, accuracy: 85.13%
Epoch  18/200, val loss: 4.26e-03, accuracy: 82.46%
Updates: curr_patience=1, best_acc=0.8306, updated=False
Epoch  19/200, train loss: 3.26e-03, accuracy: 85.57%
Epoch  19/200, val loss: 4.17e-03, accuracy: 83.46%
Updates: curr_patience=2, best_acc=0.8346, updated=True
Epoch  20/200, train loss: 3.14e-03, accuracy: 86.09%
Epoch  20/200, val loss: 4.19e-03, accuracy: 83.56%
Updates: curr_patience=3, best_acc=0.8356, updated=True
Epoch  21/200, train loss: 3.08e-03, accuracy: 86.30%
Epoch  21/200, val loss: 4.05e-03, accuracy: 83.22%
Updates: curr_patience=1, best_acc=0.8356, updated=False
Epoch  22/200, train loss: 2.98e-03, accuracy: 86.71%
Epoch  22/200, val loss: 3.97e-03, accuracy: 83.44%
Updates: curr_patience=1, best_acc=0.8356, updated=False
Epoch  23/200, train loss: 2.91e-03, accuracy: 87.10%
Epoch  23/200, val loss: 4.06e-03, accuracy: 83.42%
Updates: curr_patience=1, best_acc=0.8356, updated=False
Epoch  24/200, train loss: 2.83e-03, accuracy: 87.44%
Epoch  24/200, val loss: 3.85e-03, accuracy: 83.90%
Updates: curr_patience=2, best_acc=0.839, updated=True
Epoch  25/200, train loss: 2.79e-03, accuracy: 87.55%
```

```
Epoch  25/200, val loss: 3.73e-03, accuracy: 84.46%
Updates: curr_patience=3, best_acc=0.8446, updated=True
Epoch  26/200, train loss: 2.76e-03, accuracy: 87.70%
Epoch  26/200, val loss: 3.97e-03, accuracy: 83.82%
Updates: curr_patience=1, best_acc=0.8446, updated=False
Epoch  27/200, train loss: 2.66e-03, accuracy: 88.26%
Epoch  27/200, val loss: 3.75e-03, accuracy: 84.78%
Updates: curr_patience=2, best_acc=0.8478, updated=True
Epoch  28/200, train loss: 2.59e-03, accuracy: 88.55%
Epoch  28/200, val loss: 3.61e-03, accuracy: 84.90%
Updates: curr_patience=3, best_acc=0.849, updated=True
Epoch  29/200, train loss: 2.52e-03, accuracy: 88.80%
Epoch  29/200, val loss: 3.62e-03, accuracy: 85.06%
Updates: curr_patience=4, best_acc=0.8506, updated=True
Epoch  30/200, train loss: 2.50e-03, accuracy: 88.82%
Epoch  30/200, val loss: 3.58e-03, accuracy: 85.20%
Updates: curr_patience=5, best_acc=0.852, updated=True
Epoch  31/200, train loss: 2.47e-03, accuracy: 89.05%
Epoch  31/200, val loss: 3.59e-03, accuracy: 84.94%
Updates: curr_patience=1, best_acc=0.852, updated=False
Epoch  32/200, train loss: 2.41e-03, accuracy: 89.30%
Epoch  32/200, val loss: 3.64e-03, accuracy: 85.38%
Updates: curr_patience=2, best_acc=0.8538, updated=True
Epoch  33/200, train loss: 2.35e-03, accuracy: 89.53%
Epoch  33/200, val loss: 3.55e-03, accuracy: 86.08%
Updates: curr_patience=3, best_acc=0.8608, updated=True
Epoch  34/200, train loss: 2.34e-03, accuracy: 89.71%
Epoch  34/200, val loss: 3.59e-03, accuracy: 85.30%
Updates: curr_patience=1, best_acc=0.8608, updated=False
Epoch  35/200, train loss: 2.31e-03, accuracy: 89.72%
Epoch  35/200, val loss: 3.56e-03, accuracy: 85.54%
Updates: curr_patience=1, best_acc=0.8608, updated=False
Epoch  36/200, train loss: 2.25e-03, accuracy: 89.81%
Epoch  36/200, val loss: 3.66e-03, accuracy: 85.32%
Updates: curr_patience=1, best_acc=0.8608, updated=False
Epoch  37/200, train loss: 2.22e-03, accuracy: 89.99%
Epoch  37/200, val loss: 3.77e-03, accuracy: 84.86%
Updates: curr_patience=1, best_acc=0.8608, updated=False
Epoch  38/200, train loss: 2.26e-03, accuracy: 89.89%
Epoch  38/200, val loss: 3.47e-03, accuracy: 85.88%
Updates: curr_patience=1, best_acc=0.8608, updated=False
Epoch  39/200, train loss: 2.13e-03, accuracy: 90.43%
Epoch  39/200, val loss: 3.66e-03, accuracy: 85.62%
Updates: curr_patience=1, best_acc=0.8608, updated=False
Epoch  40/200, train loss: 2.13e-03, accuracy: 90.50%
Epoch  40/200, val loss: 3.66e-03, accuracy: 85.08%
Updates: curr_patience=1, best_acc=0.8608, updated=False
Epoch  41/200, train loss: 2.10e-03, accuracy: 90.51%
```

```
Epoch  41/200, val loss: 3.49e-03, accuracy: 85.82%
Updates: curr_patience=1, best_acc=0.8608, updated=False
Epoch  42/200, train loss: 2.07e-03, accuracy: 90.70%
Epoch  42/200, val loss: 3.41e-03, accuracy: 85.86%
Updates: curr_patience=1, best_acc=0.8608, updated=False
Epoch  43/200, train loss: 2.04e-03, accuracy: 91.04%
Epoch  43/200, val loss: 3.64e-03, accuracy: 85.92%
Updates: curr_patience=1, best_acc=0.8608, updated=False
Epoch  44/200, train loss: 2.02e-03, accuracy: 90.98%
Epoch  44/200, val loss: 3.52e-03, accuracy: 86.28%
Updates: curr_patience=2, best_acc=0.8628, updated=True
Epoch  45/200, train loss: 1.97e-03, accuracy: 91.14%
Epoch  45/200, val loss: 3.73e-03, accuracy: 85.34%
Updates: curr_patience=1, best_acc=0.8628, updated=False
Epoch  46/200, train loss: 1.95e-03, accuracy: 91.30%
Epoch  46/200, val loss: 3.82e-03, accuracy: 85.20%
Updates: curr_patience=1, best_acc=0.8628, updated=False
Epoch  47/200, train loss: 1.98e-03, accuracy: 91.17%
Epoch  47/200, val loss: 3.38e-03, accuracy: 86.98%
Updates: curr_patience=2, best_acc=0.8698, updated=True
Epoch  48/200, train loss: 1.93e-03, accuracy: 91.34%
Epoch  48/200, val loss: 3.77e-03, accuracy: 85.46%
Updates: curr_patience=1, best_acc=0.8698, updated=False
Epoch  49/200, train loss: 1.91e-03, accuracy: 91.52%
Epoch  49/200, val loss: 3.58e-03, accuracy: 85.54%
Updates: curr_patience=1, best_acc=0.8698, updated=False
Epoch  50/200, train loss: 1.87e-03, accuracy: 91.49%
Epoch  50/200, val loss: 3.35e-03, accuracy: 86.56%
Updates: curr_patience=1, best_acc=0.8698, updated=False
Epoch  51/200, train loss: 1.86e-03, accuracy: 91.70%
Epoch  51/200, val loss: 3.23e-03, accuracy: 87.06%
Updates: curr_patience=2, best_acc=0.8706, updated=True
Epoch  52/200, train loss: 1.89e-03, accuracy: 91.59%
Epoch  52/200, val loss: 3.36e-03, accuracy: 87.02%
Updates: curr_patience=1, best_acc=0.8706, updated=False
Epoch  53/200, train loss: 1.82e-03, accuracy: 91.88%
Epoch  53/200, val loss: 3.52e-03, accuracy: 86.30%
Updates: curr_patience=1, best_acc=0.8706, updated=False
Epoch  54/200, train loss: 1.81e-03, accuracy: 92.00%
Epoch  54/200, val loss: 3.50e-03, accuracy: 86.86%
Updates: curr_patience=1, best_acc=0.8706, updated=False
Epoch  55/200, train loss: 1.81e-03, accuracy: 91.93%
Epoch  55/200, val loss: 3.45e-03, accuracy: 86.64%
Updates: curr_patience=1, best_acc=0.8706, updated=False
Epoch  56/200, train loss: 1.80e-03, accuracy: 91.83%
Epoch  56/200, val loss: 3.57e-03, accuracy: 86.52%
Updates: curr_patience=1, best_acc=0.8706, updated=False
Epoch  57/200, train loss: 1.72e-03, accuracy: 92.24%
```

```
Epoch  57/200, val loss: 3.13e-03, accuracy: 87.72%
Updates: curr_patience=2, best_acc=0.8772, updated=True
Epoch  58/200, train loss: 1.74e-03, accuracy: 92.21%
Epoch  58/200, val loss: 3.41e-03, accuracy: 86.68%
Updates: curr_patience=1, best_acc=0.8772, updated=False
Epoch  59/200, train loss: 1.74e-03, accuracy: 92.22%
Epoch  59/200, val loss: 3.19e-03, accuracy: 87.32%
Updates: curr_patience=1, best_acc=0.8772, updated=False
Epoch  60/200, train loss: 1.69e-03, accuracy: 92.51%
Epoch  60/200, val loss: 3.34e-03, accuracy: 86.86%
Updates: curr_patience=1, best_acc=0.8772, updated=False
Epoch  61/200, train loss: 1.71e-03, accuracy: 92.39%
Epoch  61/200, val loss: 3.46e-03, accuracy: 87.16%
Updates: curr_patience=1, best_acc=0.8772, updated=False
Epoch  62/200, train loss: 1.71e-03, accuracy: 92.21%
Epoch  62/200, val loss: 3.41e-03, accuracy: 86.38%
Updates: curr_patience=1, best_acc=0.8772, updated=False
Epoch  63/200, train loss: 1.71e-03, accuracy: 92.17%
Epoch  63/200, val loss: 3.24e-03, accuracy: 86.92%
Updates: curr_patience=1, best_acc=0.8772, updated=False
Epoch  64/200, train loss: 1.67e-03, accuracy: 92.52%
Epoch  64/200, val loss: 3.53e-03, accuracy: 86.58%
Updates: curr_patience=1, best_acc=0.8772, updated=False
Epoch  65/200, train loss: 1.65e-03, accuracy: 92.71%
Epoch  65/200, val loss: 3.40e-03, accuracy: 87.32%
Updates: curr_patience=1, best_acc=0.8772, updated=False
Epoch  66/200, train loss: 1.67e-03, accuracy: 92.53%
Epoch  66/200, val loss: 3.47e-03, accuracy: 87.02%
Updates: curr_patience=1, best_acc=0.8772, updated=False
Epoch  67/200, train loss: 1.59e-03, accuracy: 92.79%
Epoch  67/200, val loss: 3.63e-03, accuracy: 86.06%
Updates: curr_patience=1, best_acc=0.8772, updated=False
Epoch  68/200, train loss: 1.69e-03, accuracy: 92.32%
Epoch  68/200, val loss: 3.37e-03, accuracy: 87.30%
Updates: curr_patience=1, best_acc=0.8772, updated=False
Epoch  69/200, train loss: 1.58e-03, accuracy: 92.79%
Epoch  69/200, val loss: 3.41e-03, accuracy: 86.68%
Updates: curr_patience=1, best_acc=0.8772, updated=False
Epoch  70/200, train loss: 1.61e-03, accuracy: 92.82%
Epoch  70/200, val loss: 3.25e-03, accuracy: 87.28%
Updates: curr_patience=1, best_acc=0.8772, updated=False
Epoch  71/200, train loss: 1.57e-03, accuracy: 92.92%
Epoch  71/200, val loss: 3.47e-03, accuracy: 86.84%
Updates: curr_patience=1, best_acc=0.8772, updated=False
Epoch  72/200, train loss: 1.61e-03, accuracy: 92.76%
Epoch  72/200, val loss: 3.41e-03, accuracy: 86.86%
Updates: curr_patience=1, best_acc=0.8772, updated=False
Epoch  73/200, train loss: 1.57e-03, accuracy: 92.85%
```

```
Epoch  73/200, val loss: 3.42e-03, accuracy: 87.22%
Updates: curr_patience=1, best_acc=0.8772, updated=False
Epoch  74/200, train loss: 1.56e-03, accuracy: 92.99%
Epoch  74/200, val loss: 3.34e-03, accuracy: 87.14%
Updates: curr_patience=1, best_acc=0.8772, updated=False
Epoch  75/200, train loss: 1.52e-03, accuracy: 93.29%
Epoch  75/200, val loss: 3.32e-03, accuracy: 87.04%
Updates: curr_patience=1, best_acc=0.8772, updated=False
Epoch  76/200, train loss: 1.48e-03, accuracy: 93.42%
Epoch  76/200, val loss: 3.25e-03, accuracy: 88.00%
Updates: curr_patience=2, best_acc=0.88, updated=True
Epoch  77/200, train loss: 1.56e-03, accuracy: 93.10%
Epoch  77/200, val loss: 3.36e-03, accuracy: 86.92%
Updates: curr_patience=1, best_acc=0.88, updated=False
Epoch  78/200, train loss: 1.52e-03, accuracy: 93.17%
Epoch  78/200, val loss: 3.47e-03, accuracy: 86.96%
Updates: curr_patience=1, best_acc=0.88, updated=False
Epoch  79/200, train loss: 1.51e-03, accuracy: 93.28%
Epoch  79/200, val loss: 3.39e-03, accuracy: 86.76%
Updates: curr_patience=1, best_acc=0.88, updated=False
Epoch  80/200, train loss: 1.51e-03, accuracy: 93.23%
Epoch  80/200, val loss: 2.96e-03, accuracy: 88.20%
Updates: curr_patience=2, best_acc=0.882, updated=True
Epoch  81/200, train loss: 1.51e-03, accuracy: 93.16%
Epoch  81/200, val loss: 3.22e-03, accuracy: 87.18%
Updates: curr_patience=1, best_acc=0.882, updated=False
Epoch  82/200, train loss: 1.47e-03, accuracy: 93.48%
Epoch  82/200, val loss: 3.16e-03, accuracy: 88.14%
Updates: curr_patience=1, best_acc=0.882, updated=False
Epoch  83/200, train loss: 1.45e-03, accuracy: 93.40%
Epoch  83/200, val loss: 3.43e-03, accuracy: 86.44%
Updates: curr_patience=1, best_acc=0.882, updated=False
Epoch  84/200, train loss: 1.51e-03, accuracy: 93.24%
Epoch  84/200, val loss: 3.21e-03, accuracy: 87.50%
Updates: curr_patience=1, best_acc=0.882, updated=False
Epoch  85/200, train loss: 1.47e-03, accuracy: 93.30%
Epoch  85/200, val loss: 3.37e-03, accuracy: 87.58%
Updates: curr_patience=1, best_acc=0.882, updated=False
Epoch  86/200, train loss: 1.50e-03, accuracy: 93.12%
Epoch  86/200, val loss: 3.38e-03, accuracy: 87.70%
Updates: curr_patience=1, best_acc=0.882, updated=False
Epoch  87/200, train loss: 1.49e-03, accuracy: 93.28%
Epoch  87/200, val loss: 3.39e-03, accuracy: 87.30%
Updates: curr_patience=1, best_acc=0.882, updated=False
Epoch  88/200, train loss: 1.43e-03, accuracy: 93.54%
Epoch  88/200, val loss: 3.27e-03, accuracy: 87.44%
Updates: curr_patience=1, best_acc=0.882, updated=False
Epoch  89/200, train loss: 1.50e-03, accuracy: 93.20%
```

```
Epoch  89/200, val loss: 3.42e-03, accuracy: 87.02%
Updates: curr_patience=1, best_acc=0.882, updated=False
Epoch  90/200, train loss: 1.48e-03, accuracy: 93.43%
Epoch  90/200, val loss: 3.18e-03, accuracy: 88.12%
Updates: curr_patience=1, best_acc=0.882, updated=False
Epoch  91/200, train loss: 1.47e-03, accuracy: 93.41%
Epoch  91/200, val loss: 3.07e-03, accuracy: 88.04%
Updates: curr_patience=1, best_acc=0.882, updated=False
Epoch  92/200, train loss: 1.43e-03, accuracy: 93.60%
Epoch  92/200, val loss: 3.27e-03, accuracy: 87.80%
Updates: curr_patience=1, best_acc=0.882, updated=False
Epoch  93/200, train loss: 1.45e-03, accuracy: 93.43%
Epoch  93/200, val loss: 3.30e-03, accuracy: 88.28%
Updates: curr_patience=2, best_acc=0.8828, updated=True
Epoch  94/200, train loss: 1.43e-03, accuracy: 93.68%
Epoch  94/200, val loss: 3.32e-03, accuracy: 87.36%
Updates: curr_patience=1, best_acc=0.8828, updated=False
Epoch  95/200, train loss: 1.42e-03, accuracy: 93.63%
Epoch  95/200, val loss: 3.28e-03, accuracy: 88.18%
Updates: curr_patience=1, best_acc=0.8828, updated=False
Epoch  96/200, train loss: 1.43e-03, accuracy: 93.50%
Epoch  96/200, val loss: 3.59e-03, accuracy: 87.06%
Updates: curr_patience=1, best_acc=0.8828, updated=False
Epoch  97/200, train loss: 1.43e-03, accuracy: 93.69%
Epoch  97/200, val loss: 3.36e-03, accuracy: 87.60%
Updates: curr_patience=1, best_acc=0.8828, updated=False
Epoch  98/200, train loss: 1.45e-03, accuracy: 93.45%
Epoch  98/200, val loss: 3.55e-03, accuracy: 86.86%
Updates: curr_patience=1, best_acc=0.8828, updated=False
Epoch  99/200, train loss: 1.39e-03, accuracy: 93.53%
Epoch  99/200, val loss: 3.01e-03, accuracy: 88.44%
Updates: curr_patience=2, best_acc=0.8844, updated=True
Epoch 100/200, train loss: 1.38e-03, accuracy: 93.80%
Epoch 100/200, val loss: 3.61e-03, accuracy: 86.70%
Updates: curr_patience=1, best_acc=0.8844, updated=False
Epoch 101/200, train loss: 7.20e-04, accuracy: 96.86%
Epoch 101/200, val loss: 2.71e-03, accuracy: 89.98%
Updates: curr_patience=2, best_acc=0.8998, updated=True
Epoch 102/200, train loss: 4.79e-04, accuracy: 98.01%
Epoch 102/200, val loss: 2.67e-03, accuracy: 90.60%
Updates: curr_patience=3, best_acc=0.906, updated=True
Epoch 103/200, train loss: 4.08e-04, accuracy: 98.33%
Epoch 103/200, val loss: 2.69e-03, accuracy: 90.68%
Updates: curr_patience=4, best_acc=0.9068, updated=True
Epoch 104/200, train loss: 3.61e-04, accuracy: 98.54%
Epoch 104/200, val loss: 2.77e-03, accuracy: 90.70%
Updates: curr_patience=5, best_acc=0.907, updated=True
Epoch 105/200, train loss: 3.31e-04, accuracy: 98.62%
```

```
Epoch 105/200, val loss: 2.77e-03, accuracy: 90.70%
Updates: curr_patience=6, best_acc=0.907, updated=False
Epoch 106/200, train loss: 2.93e-04, accuracy: 98.82%
Epoch 106/200, val loss: 2.89e-03, accuracy: 90.58%
Updates: curr_patience=1, best_acc=0.907, updated=False
Epoch 107/200, train loss: 2.73e-04, accuracy: 98.89%
Epoch 107/200, val loss: 2.88e-03, accuracy: 90.98%
Updates: curr_patience=2, best_acc=0.9098, updated=True
Epoch 108/200, train loss: 2.46e-04, accuracy: 98.98%
Epoch 108/200, val loss: 2.92e-03, accuracy: 90.92%
Updates: curr_patience=1, best_acc=0.9098, updated=False
Epoch 109/200, train loss: 2.32e-04, accuracy: 99.04%
Epoch 109/200, val loss: 2.95e-03, accuracy: 90.86%
Updates: curr_patience=1, best_acc=0.9098, updated=False
Epoch 110/200, train loss: 1.99e-04, accuracy: 99.18%
Epoch 110/200, val loss: 3.06e-03, accuracy: 90.78%
Updates: curr_patience=1, best_acc=0.9098, updated=False
Epoch 111/200, train loss: 1.95e-04, accuracy: 99.27%
Epoch 111/200, val loss: 3.08e-03, accuracy: 90.64%
Updates: curr_patience=1, best_acc=0.9098, updated=False
Epoch 112/200, train loss: 1.89e-04, accuracy: 99.27%
Epoch 112/200, val loss: 3.04e-03, accuracy: 91.00%
Updates: curr_patience=2, best_acc=0.91, updated=True
Epoch 113/200, train loss: 1.70e-04, accuracy: 99.38%
Epoch 113/200, val loss: 3.11e-03, accuracy: 90.62%
Updates: curr_patience=1, best_acc=0.91, updated=False
Epoch 114/200, train loss: 1.74e-04, accuracy: 99.31%
Epoch 114/200, val loss: 3.11e-03, accuracy: 90.70%
Updates: curr_patience=1, best_acc=0.91, updated=False
Epoch 115/200, train loss: 1.69e-04, accuracy: 99.33%
Epoch 115/200, val loss: 3.19e-03, accuracy: 90.94%
Updates: curr_patience=1, best_acc=0.91, updated=False
Epoch 116/200, train loss: 1.52e-04, accuracy: 99.42%
Epoch 116/200, val loss: 3.29e-03, accuracy: 90.48%
Updates: curr_patience=1, best_acc=0.91, updated=False
Epoch 117/200, train loss: 1.43e-04, accuracy: 99.44%
Epoch 117/200, val loss: 3.37e-03, accuracy: 90.54%
Updates: curr_patience=1, best_acc=0.91, updated=False
Epoch 118/200, train loss: 1.37e-04, accuracy: 99.48%
Epoch 118/200, val loss: 3.31e-03, accuracy: 90.36%
Updates: curr_patience=1, best_acc=0.91, updated=False
Epoch 119/200, train loss: 1.31e-04, accuracy: 99.48%
Epoch 119/200, val loss: 3.29e-03, accuracy: 90.68%
Updates: curr_patience=1, best_acc=0.91, updated=False
Epoch 120/200, train loss: 1.27e-04, accuracy: 99.50%
Epoch 120/200, val loss: 3.30e-03, accuracy: 90.62%
Updates: curr_patience=1, best_acc=0.91, updated=False
Epoch 121/200, train loss: 1.31e-04, accuracy: 99.47%
```

```
Epoch 121/200, val loss: 3.48e-03, accuracy: 90.50%
Updates: curr_patience=1, best_acc=0.91, updated=False
Epoch 122/200, train loss: 1.12e-04, accuracy: 99.60%
Epoch 122/200, val loss: 3.40e-03, accuracy: 90.68%
Updates: curr_patience=1, best_acc=0.91, updated=False
Epoch 123/200, train loss: 1.15e-04, accuracy: 99.52%
Epoch 123/200, val loss: 3.55e-03, accuracy: 90.34%
Updates: curr_patience=1, best_acc=0.91, updated=False
Epoch 124/200, train loss: 1.09e-04, accuracy: 99.59%
Epoch 124/200, val loss: 3.42e-03, accuracy: 90.62%
Updates: curr_patience=1, best_acc=0.91, updated=False
Epoch 125/200, train loss: 1.06e-04, accuracy: 99.59%
Epoch 125/200, val loss: 3.45e-03, accuracy: 90.58%
Updates: curr_patience=1, best_acc=0.91, updated=False
Epoch 126/200, train loss: 9.50e-05, accuracy: 99.66%
Epoch 126/200, val loss: 3.41e-03, accuracy: 90.88%
Updates: curr_patience=1, best_acc=0.91, updated=False
Epoch 127/200, train loss: 1.05e-04, accuracy: 99.64%
Epoch 127/200, val loss: 3.53e-03, accuracy: 90.42%
Updates: curr_patience=1, best_acc=0.91, updated=False
Epoch 128/200, train loss: 9.98e-05, accuracy: 99.62%
Epoch 128/200, val loss: 3.42e-03, accuracy: 90.54%
Updates: curr_patience=1, best_acc=0.91, updated=False
Epoch 129/200, train loss: 1.00e-04, accuracy: 99.58%
Epoch 129/200, val loss: 3.55e-03, accuracy: 90.66%
Updates: curr_patience=1, best_acc=0.91, updated=False
Epoch 130/200, train loss: 8.87e-05, accuracy: 99.68%
Epoch 130/200, val loss: 3.49e-03, accuracy: 90.94%
Updates: curr_patience=1, best_acc=0.91, updated=False
Epoch 131/200, train loss: 8.15e-05, accuracy: 99.73%
Epoch 131/200, val loss: 3.52e-03, accuracy: 90.76%
Updates: curr_patience=1, best_acc=0.91, updated=False
Epoch 132/200, train loss: 8.20e-05, accuracy: 99.70%
Epoch 132/200, val loss: 3.52e-03, accuracy: 90.74%
Updates: curr_patience=1, best_acc=0.91, updated=False
Epoch 133/200, train loss: 7.63e-05, accuracy: 99.74%
Epoch 133/200, val loss: 3.59e-03, accuracy: 90.70%
Updates: curr_patience=1, best_acc=0.91, updated=False
Epoch 134/200, train loss: 8.69e-05, accuracy: 99.69%
Epoch 134/200, val loss: 3.54e-03, accuracy: 90.84%
Updates: curr_patience=1, best_acc=0.91, updated=False
Epoch 135/200, train loss: 7.38e-05, accuracy: 99.76%
Epoch 135/200, val loss: 3.54e-03, accuracy: 90.74%
Updates: curr_patience=1, best_acc=0.91, updated=False
Epoch 136/200, train loss: 7.20e-05, accuracy: 99.72%
Epoch 136/200, val loss: 3.64e-03, accuracy: 90.50%
Updates: curr_patience=1, best_acc=0.91, updated=False
Epoch 137/200, train loss: 7.26e-05, accuracy: 99.76%
```

```
Epoch 137/200, val loss: 3.79e-03, accuracy: 90.32%
Updates: curr_patience=1, best_acc=0.91, updated=False
Epoch 138/200, train loss: 7.19e-05, accuracy: 99.76%
Epoch 138/200, val loss: 3.56e-03, accuracy: 90.62%
Updates: curr_patience=1, best_acc=0.91, updated=False
Epoch 139/200, train loss: 7.23e-05, accuracy: 99.74%
Epoch 139/200, val loss: 3.60e-03, accuracy: 90.44%
Updates: curr_patience=1, best_acc=0.91, updated=False
Epoch 140/200, train loss: 7.39e-05, accuracy: 99.74%
Epoch 140/200, val loss: 3.68e-03, accuracy: 90.82%
Updates: curr_patience=1, best_acc=0.91, updated=False
Epoch 141/200, train loss: 7.02e-05, accuracy: 99.77%
Epoch 141/200, val loss: 3.74e-03, accuracy: 91.10%
Updates: curr_patience=2, best_acc=0.911, updated=True
Epoch 142/200, train loss: 6.90e-05, accuracy: 99.75%
Epoch 142/200, val loss: 3.73e-03, accuracy: 90.80%
Updates: curr_patience=1, best_acc=0.911, updated=False
Epoch 143/200, train loss: 6.73e-05, accuracy: 99.77%
Epoch 143/200, val loss: 3.74e-03, accuracy: 90.86%
Updates: curr_patience=1, best_acc=0.911, updated=False
Epoch 144/200, train loss: 6.83e-05, accuracy: 99.78%
Epoch 144/200, val loss: 3.70e-03, accuracy: 90.84%
Updates: curr_patience=1, best_acc=0.911, updated=False
Epoch 145/200, train loss: 6.36e-05, accuracy: 99.78%
Epoch 145/200, val loss: 3.64e-03, accuracy: 91.04%
Updates: curr_patience=1, best_acc=0.911, updated=False
Epoch 146/200, train loss: 6.35e-05, accuracy: 99.78%
Epoch 146/200, val loss: 3.82e-03, accuracy: 90.70%
Updates: curr_patience=1, best_acc=0.911, updated=False
Epoch 147/200, train loss: 6.96e-05, accuracy: 99.74%
Epoch 147/200, val loss: 3.80e-03, accuracy: 90.50%
Updates: curr_patience=1, best_acc=0.911, updated=False
Epoch 148/200, train loss: 6.83e-05, accuracy: 99.74%
Epoch 148/200, val loss: 3.73e-03, accuracy: 90.80%
Updates: curr_patience=1, best_acc=0.911, updated=False
Epoch 149/200, train loss: 5.86e-05, accuracy: 99.80%
Epoch 149/200, val loss: 3.81e-03, accuracy: 90.68%
Updates: curr_patience=1, best_acc=0.911, updated=False
Epoch 150/200, train loss: 5.65e-05, accuracy: 99.79%
Epoch 150/200, val loss: 3.82e-03, accuracy: 91.18%
Updates: curr_patience=2, best_acc=0.9118, updated=True
Epoch 151/200, train loss: 5.20e-05, accuracy: 99.80%
Epoch 151/200, val loss: 3.79e-03, accuracy: 91.14%
Updates: curr_patience=1, best_acc=0.9118, updated=False
Epoch 152/200, train loss: 4.47e-05, accuracy: 99.86%
Epoch 152/200, val loss: 3.79e-03, accuracy: 91.16%
Updates: curr_patience=1, best_acc=0.9118, updated=False
Epoch 153/200, train loss: 4.52e-05, accuracy: 99.86%
```

```
Epoch 153/200, val loss: 3.78e-03, accuracy: 91.16%
Updates: curr_patience=1, best_acc=0.9118, updated=False
Epoch 154/200, train loss: 4.77e-05, accuracy: 99.84%
Epoch 154/200, val loss: 3.80e-03, accuracy: 90.94%
Updates: curr_patience=1, best_acc=0.9118, updated=False
Epoch 155/200, train loss: 4.24e-05, accuracy: 99.87%
Epoch 155/200, val loss: 3.79e-03, accuracy: 90.98%
Updates: curr_patience=1, best_acc=0.9118, updated=False
Epoch 156/200, train loss: 4.05e-05, accuracy: 99.87%
Epoch 156/200, val loss: 3.79e-03, accuracy: 90.94%
Updates: curr_patience=1, best_acc=0.9118, updated=False
Epoch 157/200, train loss: 3.85e-05, accuracy: 99.91%
Epoch 157/200, val loss: 3.79e-03, accuracy: 90.92%
Updates: curr_patience=1, best_acc=0.9118, updated=False
Epoch 158/200, train loss: 3.74e-05, accuracy: 99.88%
Epoch 158/200, val loss: 3.78e-03, accuracy: 90.88%
Updates: curr_patience=1, best_acc=0.9118, updated=False
Epoch 159/200, train loss: 3.89e-05, accuracy: 99.89%
Epoch 159/200, val loss: 3.78e-03, accuracy: 90.92%
Updates: curr_patience=1, best_acc=0.9118, updated=False
Epoch 160/200, train loss: 3.73e-05, accuracy: 99.91%
Epoch 160/200, val loss: 3.79e-03, accuracy: 90.94%
Updates: curr_patience=1, best_acc=0.9118, updated=False
Epoch 161/200, train loss: 3.73e-05, accuracy: 99.90%
Epoch 161/200, val loss: 3.81e-03, accuracy: 90.86%
Updates: curr_patience=1, best_acc=0.9118, updated=False
Epoch 162/200, train loss: 3.79e-05, accuracy: 99.88%
Epoch 162/200, val loss: 3.79e-03, accuracy: 90.92%
Updates: curr_patience=1, best_acc=0.9118, updated=False
Epoch 163/200, train loss: 3.51e-05, accuracy: 99.92%
Epoch 163/200, val loss: 3.79e-03, accuracy: 90.94%
Updates: curr_patience=1, best_acc=0.9118, updated=False
Epoch 164/200, train loss: 3.81e-05, accuracy: 99.90%
Epoch 164/200, val loss: 3.78e-03, accuracy: 91.00%
Updates: curr_patience=1, best_acc=0.9118, updated=False
Epoch 165/200, train loss: 3.74e-05, accuracy: 99.89%
Epoch 165/200, val loss: 3.77e-03, accuracy: 91.10%
Updates: curr_patience=1, best_acc=0.9118, updated=False
Epoch 166/200, train loss: 3.34e-05, accuracy: 99.90%
Epoch 166/200, val loss: 3.78e-03, accuracy: 91.04%
Updates: curr_patience=1, best_acc=0.9118, updated=False
Epoch 167/200, train loss: 3.06e-05, accuracy: 99.93%
Epoch 167/200, val loss: 3.80e-03, accuracy: 91.10%
Updates: curr_patience=1, best_acc=0.9118, updated=False
Epoch 168/200, train loss: 3.33e-05, accuracy: 99.91%
Epoch 168/200, val loss: 3.80e-03, accuracy: 91.04%
Updates: curr_patience=1, best_acc=0.9118, updated=False
Epoch 169/200, train loss: 3.57e-05, accuracy: 99.89%
```

```
Epoch 169/200, val loss: 3.78e-03, accuracy: 91.14%
Updates: curr_patience=1, best_acc=0.9118, updated=False
Epoch 170/200, train loss: 3.24e-05, accuracy: 99.91%
Epoch 170/200, val loss: 3.80e-03, accuracy: 91.12%
Updates: curr_patience=1, best_acc=0.9118, updated=False
Epoch 171/200, train loss: 3.44e-05, accuracy: 99.91%
Epoch 171/200, val loss: 3.80e-03, accuracy: 91.02%
Updates: curr_patience=1, best_acc=0.9118, updated=False
Epoch 172/200, train loss: 3.54e-05, accuracy: 99.90%
Epoch 172/200, val loss: 3.80e-03, accuracy: 91.08%
Updates: curr_patience=1, best_acc=0.9118, updated=False
Epoch 173/200, train loss: 3.16e-05, accuracy: 99.92%
Epoch 173/200, val loss: 3.81e-03, accuracy: 91.00%
Updates: curr_patience=1, best_acc=0.9118, updated=False
Epoch 174/200, train loss: 3.56e-05, accuracy: 99.90%
Epoch 174/200, val loss: 3.80e-03, accuracy: 90.88%
Updates: curr_patience=1, best_acc=0.9118, updated=False
Epoch 175/200, train loss: 3.16e-05, accuracy: 99.92%
Epoch 175/200, val loss: 3.81e-03, accuracy: 90.88%
Updates: curr_patience=1, best_acc=0.9118, updated=False
Epoch 176/200, train loss: 3.28e-05, accuracy: 99.90%
Epoch 176/200, val loss: 3.80e-03, accuracy: 90.82%
Updates: curr_patience=1, best_acc=0.9118, updated=False
Epoch 177/200, train loss: 3.20e-05, accuracy: 99.91%
Epoch 177/200, val loss: 3.81e-03, accuracy: 90.88%
Updates: curr_patience=1, best_acc=0.9118, updated=False
Epoch 178/200, train loss: 3.41e-05, accuracy: 99.89%
Epoch 178/200, val loss: 3.81e-03, accuracy: 90.96%
Updates: curr_patience=1, best_acc=0.9118, updated=False
Epoch 179/200, train loss: 3.32e-05, accuracy: 99.90%
Epoch 179/200, val loss: 3.81e-03, accuracy: 90.94%
Updates: curr_patience=1, best_acc=0.9118, updated=False
Epoch 180/200, train loss: 3.01e-05, accuracy: 99.93%
Epoch 180/200, val loss: 3.81e-03, accuracy: 90.94%
Updates: curr_patience=1, best_acc=0.9118, updated=False
Epoch 181/200, train loss: 3.18e-05, accuracy: 99.92%
Epoch 181/200, val loss: 3.82e-03, accuracy: 90.78%
Updates: curr_patience=1, best_acc=0.9118, updated=False
Epoch 182/200, train loss: 2.84e-05, accuracy: 99.93%
Epoch 182/200, val loss: 3.82e-03, accuracy: 90.80%
Updates: curr_patience=1, best_acc=0.9118, updated=False
Epoch 183/200, train loss: 3.19e-05, accuracy: 99.92%
Epoch 183/200, val loss: 3.81e-03, accuracy: 90.82%
Updates: curr_patience=1, best_acc=0.9118, updated=False
Epoch 184/200, train loss: 3.21e-05, accuracy: 99.92%
Epoch 184/200, val loss: 3.81e-03, accuracy: 90.82%
Updates: curr_patience=1, best_acc=0.9118, updated=False
Epoch 185/200, train loss: 3.09e-05, accuracy: 99.90%
```

```
Epoch 185/200, val loss: 3.82e-03, accuracy: 90.82%
Updates: curr_patience=1, best_acc=0.9118, updated=False
Epoch 186/200, train loss: 3.11e-05, accuracy: 99.92%
Epoch 186/200, val loss: 3.82e-03, accuracy: 90.84%
Updates: curr_patience=1, best_acc=0.9118, updated=False
Epoch 187/200, train loss: 3.06e-05, accuracy: 99.92%
Epoch 187/200, val loss: 3.82e-03, accuracy: 90.80%
Updates: curr_patience=1, best_acc=0.9118, updated=False
Epoch 188/200, train loss: 2.91e-05, accuracy: 99.94%
Epoch 188/200, val loss: 3.82e-03, accuracy: 90.86%
Updates: curr_patience=1, best_acc=0.9118, updated=False
Epoch 189/200, train loss: 3.08e-05, accuracy: 99.94%
Epoch 189/200, val loss: 3.83e-03, accuracy: 90.94%
Updates: curr_patience=1, best_acc=0.9118, updated=False
Epoch 190/200, train loss: 3.01e-05, accuracy: 99.91%
Epoch 190/200, val loss: 3.81e-03, accuracy: 90.80%
Updates: curr_patience=1, best_acc=0.9118, updated=False
Epoch 191/200, train loss: 2.96e-05, accuracy: 99.93%
Epoch 191/200, val loss: 3.81e-03, accuracy: 90.94%
Updates: curr_patience=1, best_acc=0.9118, updated=False
Epoch 192/200, train loss: 3.31e-05, accuracy: 99.91%
Epoch 192/200, val loss: 3.82e-03, accuracy: 90.86%
Updates: curr_patience=1, best_acc=0.9118, updated=False
Epoch 193/200, train loss: 2.78e-05, accuracy: 99.93%
Epoch 193/200, val loss: 3.82e-03, accuracy: 90.82%
Updates: curr_patience=1, best_acc=0.9118, updated=False
Epoch 194/200, train loss: 2.98e-05, accuracy: 99.92%
Epoch 194/200, val loss: 3.83e-03, accuracy: 90.72%
Updates: curr_patience=1, best_acc=0.9118, updated=False
Epoch 195/200, train loss: 2.58e-05, accuracy: 99.94%
Epoch 195/200, val loss: 3.83e-03, accuracy: 90.76%
Updates: curr_patience=1, best_acc=0.9118, updated=False
Epoch 196/200, train loss: 2.89e-05, accuracy: 99.93%
Epoch 196/200, val loss: 3.82e-03, accuracy: 90.84%
Updates: curr_patience=1, best_acc=0.9118, updated=False
Epoch 197/200, train loss: 2.61e-05, accuracy: 99.94%
Epoch 197/200, val loss: 3.83e-03, accuracy: 90.80%
Updates: curr_patience=1, best_acc=0.9118, updated=False
Epoch 198/200, train loss: 2.73e-05, accuracy: 99.94%
Epoch 198/200, val loss: 3.84e-03, accuracy: 90.80%
Updates: curr_patience=1, best_acc=0.9118, updated=False
Epoch 199/200, train loss: 2.86e-05, accuracy: 99.93%
Epoch 199/200, val loss: 3.83e-03, accuracy: 90.82%
Updates: curr_patience=1, best_acc=0.9118, updated=False
Epoch 200/200, train loss: 3.48e-05, accuracy: 99.90%
Epoch 200/200, val loss: 3.82e-03, accuracy: 90.88%
Updates: curr_patience=1, best_acc=0.9118, updated=False
Stop early: curr_patience > patience or epoch + 1 >= max_epochs
```

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!

[20]:
```
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: 3.6e-03, accuracy: 90.36%
```

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

6 Optional task: Squeeze out all the juice!

Can you do even better? Have a look at A Recipe for Training Neural Networks and at the EfficientNet architecture we discussed in the lecture. Play around with the possibilities PyTorch offers you and see how close you can get to the state of the art on CIFAR-10.

Hint: You can use Google Colab to access some free GPUs for your experiments.

Appendix

We confirm that the submitted solution is original work and was written by us without further assistance. Appropriate credit has been given where reference has been made to the work of others.

Munich, January 2, 2020, Signature Marcel Bruckner (03674122)

Munich, January 2, 2020, Signature Julian Hohenadel (03673879)

Munich, January 2, 2020, Signature Kevin Bein (03707775)