3_resnet

April 1, 2025

Deadline: January 22, 2025 (Wednesday) 23:00

1 Exercise 3. Convolutional networks. ResNet.

In the third part you need to train a convolutional neural network with a ResNet architecture (He et al, 2016).

[23]:
```
skip_training = True  # Set this flag to True before validation and submission
```

[2]:
```
# During evaluation, this cell sets skip_training to True
# skip_training = True

import tools, warnings
warnings.showwarning = tools.customwarn
```

[3]:
```
import os
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

import torch
import torchvision
import torchvision.transforms as transforms

import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

import tools
import tests
```

[4]:
```
# When running on your own computer, you can specify the data directory by:
# data_dir = tools.select_data_dir('/your/local/data/directory')
data_dir = tools.select_data_dir()
```

```
The data directory is /coursedata
```

[5]:
```
# Select the device for training (use GPU if you have one)
#device = torch.device('cuda:0')
device = torch.device('cpu')
```

[6]:
```
if skip_training:
    # The models are always evaluated on CPU
    device = torch.device("cpu")
```

1.1 FashionMNIST dataset

Let us use the FashionMNIST dataset. It consists of 60,000 training images of 10 classes: 'T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat', 'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot'.

[7]:
```
transform = transforms.Compose([
    transforms.ToTensor(),  # Transform to tensor
    transforms.Normalize((0.5,), (0.5,))  # Scale images to [-1, 1]
])

trainset = torchvision.datasets.FashionMNIST(root=data_dir, train=True,␣
 ↪download=True, transform=transform)
testset = torchvision.datasets.FashionMNIST(root=data_dir, train=False,␣
 ↪download=True, transform=transform)

classes = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat', 'Sandal',
           'Shirt', 'Sneaker', 'Bag', 'Ankle boot']

trainloader = torch.utils.data.DataLoader(trainset, batch_size=32, shuffle=True)
testloader = torch.utils.data.DataLoader(testset, batch_size=5, shuffle=False)
```

1.2 ResNet

We create a network with an architecure inspired by ResNet.

1.2.1 ResNet block

Our ResNet consists of blocks with two convolutional layers and a skip connection.

In the most general case, our implementation should have:

- Two convolutional layers with:
  - 3x3 kernel
  - no bias terms
  - padding with one pixel on both sides
  - 2d batch normalization after each convolutional layer.
- **The first convolutional layer also (optionally) has:**
  - different number of input channels and output channels
  - change of the resolution with stride.
- The skip connection:
  - simply copies the input if the resolution and the number of channels do not change.

  - if either the resolution or the number of channels change, the skip connection should have one convolutional layer with:
    * 1x1 convolution **without bias**
    * change of the resolution with stride (optional)
    * different number of input channels and output channels (optional)
  - if either the resolution or the number of channels change, the 1x1 convolutional layer is followed by 2d batch normalization.
- The ReLU nonlinearity is applied after the first convolutional layer and at the end of the block.

Note: Batch normalization is expected to be right after a convolutional layer.

The implementation should also handle specific cases such as:

Left: The number of channels and the resolution do not change. There are no computations in the skip connection.

Middle: The number of channels changes, the resolution does not change.

Right: The number of channels does not change, the resolution changes.

Your task is to implement this block. You should use the implementations of layers in `nn.Conv2d`, `nn.BatchNorm2d` as the tests rely on those implementations.

[8]:
```
class Block(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1):
        """
        Args:
          in_channels (int):  Number of input channels.
          out_channels (int): Number of output channels.
          stride (int):       Controls the stride.
        """
        super(Block, self).__init__()
        # YOUR CODE HERE
        # First convolutional layer
        self.conv1 = nn.Conv2d(
            in_channels=in_channels,
            out_channels=out_channels,
            kernel_size=3,
            stride=stride,
            padding=1,  # 1 pixel padding on both sides
            bias=False
        )
        self.bn1 = nn.BatchNorm2d(out_channels)

        # Second convolutional layer
        self.conv2 = nn.Conv2d(
            in_channels=out_channels,
            out_channels=out_channels,
            kernel_size=3,
```

```
            stride=1,  # Stride is always 1 for the second conv layer
            padding=1,
            bias=False
        )
        self.bn2 = nn.BatchNorm2d(out_channels)

        # Skip connection
        self.skip = None  # Default: identity connection
        if stride != 1 or in_channels != out_channels:
            # If the number of channels or resolution changes
            self.skip = nn.Sequential(
                nn.Conv2d(
                    in_channels=in_channels,
                    out_channels=out_channels,
                    kernel_size=1,  # 1x1 convolution
                    stride=stride,
                    bias=False
                ),
                nn.BatchNorm2d(out_channels)
            )

        # ReLU activation
        self.relu = nn.ReLU()

    def forward(self, x):
        # YOUR CODE HERE
        # Main path
        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)

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

        # Skip connection
        if self.skip is not None:
            skip_out = self.skip(x)
        else:
            skip_out = x  # Identity connection

        # Add skip connection and apply ReLU
        out += skip_out
        out = self.relu(out)

        return out
```

[9]:
```
def test_Block_shapes():

    # The number of channels and resolution do not change
    batch_size = 20
    x = torch.zeros(batch_size, 16, 28, 28)
    block = Block(in_channels=16, out_channels=16)
    y = block(x)
    assert y.shape == torch.Size([batch_size, 16, 28, 28]), "Bad shape of y: y.
 ↪shape={}".format(y.shape)

    # Increase the number of channels
    block = Block(in_channels=16, out_channels=32)
    y = block(x)
    assert y.shape == torch.Size([batch_size, 32, 28, 28]), "Bad shape of y: y.
 ↪shape={}".format(y.shape)

    # Decrease the resolution
    block = Block(in_channels=16, out_channels=16, stride=2)
    y = block(x)
    assert y.shape == torch.Size([batch_size, 16, 14, 14]), "Bad shape of y: y.
 ↪shape={}".format(y.shape)

    # Increase the number of channels and decrease the resolution
    block = Block(in_channels=16, out_channels=32, stride=2)
    y = block(x)
    assert y.shape == torch.Size([batch_size, 32, 14, 14]), "Bad shape of y: y.
 ↪shape={}".format(y.shape)

    print('Success')

test_Block_shapes()
```

```
Success
```

[10]:
```
tests.test_Block(Block)
tests.test_Block_relu(Block)
tests.test_Block_batch_norm(Block)
```

```
Success
Success
Success
```

1.2.2 Group of blocks

ResNet consists of several groups of blocks. The first block in a group may change the number of channels (often multiples the number by 2) and subsample (using strides).

[11]:
```
# We implement a group of blocks in this cell
class GroupOfBlocks(nn.Module):
    def __init__(self, in_channels, out_channels, n_blocks, stride=1):
        super(GroupOfBlocks, self).__init__()

        first_block = Block(in_channels, out_channels, stride)
        other_blocks = [Block(out_channels, out_channels) for _ in range(1,␣
 ↪n_blocks)]
        self.group = nn.Sequential(first_block, *other_blocks)

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

[12]:
```
# Let's print a block
group = GroupOfBlocks(in_channels=10, out_channels=20, n_blocks=3)
print(group)
```

```
GroupOfBlocks(
  (group): Sequential(
    (0): Block(
      (conv1): Conv2d(10, 20, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn1): BatchNorm2d(20, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv2): Conv2d(20, 20, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn2): BatchNorm2d(20, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (skip): Sequential(
        (0): Conv2d(10, 20, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(20, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
      (relu): ReLU()
    )
    (1): Block(
      (conv1): Conv2d(20, 20, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn1): BatchNorm2d(20, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv2): Conv2d(20, 20, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn2): BatchNorm2d(20, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU()
    )
    (2): Block(
```

```
      (conv1): Conv2d(20, 20, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn1): BatchNorm2d(20, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv2): Conv2d(20, 20, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn2): BatchNorm2d(20, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU()
    )
  )
)
```

1.2.3 ResNet

Next we implement a ResNet with the following architecture. It contains three groups of blocks, each group having two basic blocks.

The cell below contains the implementation of our ResNet.

[13]:
```
class ResNet(nn.Module):
    def __init__(self, n_blocks, n_channels=64, num_classes=10):
        """
        Args:
          n_blocks (list):   A list with three elements which contains the␣
↪number of blocks in
                             each of the three groups of blocks in ResNet.
                             For instance, n_blocks = [2, 4, 6] means that the␣
↪first group has two blocks,
                             the second group has four blocks and the third one␣
↪has six blocks.
          n_channels (int):  Number of channels in the first group of blocks.
          num_classes (int): Number of classes.
        """
        assert len(n_blocks) == 3, "The number of groups should be three."
        super(ResNet, self).__init__()
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=n_channels,␣
↪kernel_size=5, stride=1, padding=2, bias=False)
        self.bn1 = nn.BatchNorm2d(n_channels)
        self.relu = nn.ReLU(inplace=True)
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)

        self.group1 = GroupOfBlocks(n_channels, n_channels, n_blocks[0])
        self.group2 = GroupOfBlocks(n_channels, 2*n_channels, n_blocks[1],␣
↪stride=2)
        self.group3 = GroupOfBlocks(2*n_channels, 4*n_channels, n_blocks[2],␣
↪stride=2)
```

```
        self.avgpool = nn.AvgPool2d(kernel_size=4, stride=1)
        self.fc = nn.Linear(4*n_channels, num_classes)

        # Initialize weights
        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                n = m.kernel_size[0] * m.kernel_size[1] * m.out_channels
                m.weight.data.normal_(0, np.sqrt(2. / n))
            elif isinstance(m, nn.BatchNorm2d):
                m.weight.data.fill_(1)
                m.bias.data.zero_()

    def forward(self, x, verbose=False):
        """
        Args:
          x of shape (batch_size, 1, 28, 28): Input images.
          verbose: True if you want to print the shapes of the intermediate␣
↪variables.

        Returns:
          y of shape (batch_size, 10): Outputs of the network.
        """
        if verbose: print(x.shape)
        x = self.conv1(x)
        if verbose: print('conv1:  ', x.shape)
        x = self.bn1(x)
        if verbose: print('bn1:    ', x.shape)
        x = self.relu(x)
        if verbose: print('relu:   ', x.shape)
        x = self.maxpool(x)
        if verbose: print('maxpool:', x.shape)

        x = self.group1(x)
        if verbose: print('group1: ', x.shape)
        x = self.group2(x)
        if verbose: print('group2: ', x.shape)
        x = self.group3(x)
        if verbose: print('group3: ', x.shape)

        x = self.avgpool(x)
        if verbose: print('avgpool:', x.shape)

        x = x.view(-1, self.fc.in_features)
        if verbose: print('x.view: ', x.shape)
        x = self.fc(x)
        if verbose: print('out:    ', x.shape)
```

```
        return x
```

[14]:
```
def test_ResNet_shapes():
    # Create a network with 2 block in each of the three groups
    n_blocks = [2, 2, 2]  # number of blocks in the three groups
    net = ResNet(n_blocks, n_channels=10)
    net.to(device)

    # Feed a batch of images from the training data to test the network
    with torch.no_grad():
        images, labels = next(iter(trainloader))
        images = images.to(device)
        print('Shape of the input tensor:', images.shape)

        y = net.forward(images, verbose=True)
        print(y.shape)
        assert y.shape == torch.Size([trainloader.batch_size, 10]), "Bad shape␣
 ↪of y: y.shape={}".format(y.shape)

    print('Success')

test_ResNet_shapes()
```

```
Shape of the input tensor: torch.Size([32, 1, 28, 28])
torch.Size([32, 1, 28, 28])
conv1:   torch.Size([32, 10, 28, 28])
bn1:     torch.Size([32, 10, 28, 28])
relu:    torch.Size([32, 10, 28, 28])
maxpool: torch.Size([32, 10, 14, 14])
group1:  torch.Size([32, 10, 14, 14])
group2:  torch.Size([32, 20, 7, 7])
group3:  torch.Size([32, 40, 4, 4])
avgpool: torch.Size([32, 40, 1, 1])
x.view:  torch.Size([32, 40])
out:     torch.Size([32, 10])
torch.Size([32, 10])
Success
```

2 Train the network

[15]:
```
# This function computes the accuracy on the test dataset
def compute_accuracy(net, testloader):
    net.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for images, labels in testloader:
```

```
            images, labels = images.to(device), labels.to(device)
            outputs = net(images)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
    return correct / total
```

2.0.1 Training loop

In the cell below, implement the training loop. The recommended hyperparameters: * Adam optimizer with learning rate 0.01. * Cross-entropy loss. Note that we did not use softmax nonlinearity in the final layer of our network. Therefore, we need to use a loss function with log_softmax implemented, such as nn.CrossEntropyLoss. * Number of epochs: 10

We recommend you to use function `compute_accuracy()` defined above to track the accaracy during training. The test accuracy should be above 0.9.

Note: function `compute_accuracy()` sets the network into the evaluation mode which changes the way the batch statistics are computed in batch normalization. You need to set the network into the training mode (by calling `net.train()`) when you want to perform training.

[16]:
```
# Create the network
n_blocks = [2, 2, 2]  # number of blocks in the three groups
net = ResNet(n_blocks, n_channels=16)
net.to(device)
```

[16]:
```
ResNet(
  (conv1): Conv2d(1, 16, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2),
bias=False)
  (bn1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1,
ceil_mode=False)
  (group1): GroupOfBlocks(
    (group): Sequential(
      (0): Block(
        (conv1): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
        (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)
        (relu): ReLU()
      )
```

```
      (1): Block(
        (conv1): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
        (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)
        (relu): ReLU()
      )
    )
  )
  (group2): GroupOfBlocks(
    (group): Sequential(
      (0): Block(
        (conv1): Conv2d(16, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1,
1), bias=False)
        (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)
        (skip): Sequential(
          (0): Conv2d(16, 32, kernel_size=(1, 1), stride=(2, 2), bias=False)
          (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        )
        (relu): ReLU()
      )
      (1): Block(
        (conv1): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
        (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)
        (relu): ReLU()
      )
    )
  )
  (group3): GroupOfBlocks(
    (group): Sequential(
      (0): Block(
```

```
        (conv1): Conv2d(32, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1,
1), bias=False)
        (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)
        (skip): Sequential(
          (0): Conv2d(32, 64, kernel_size=(1, 1), stride=(2, 2), bias=False)
          (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        )
        (relu): ReLU()
      )
      (1): Block(
        (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
        (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)
        (relu): ReLU()
      )
    )
  )
  (avgpool): AvgPool2d(kernel_size=4, stride=1, padding=0)
  (fc): Linear(in_features=64, out_features=10, bias=True)
)
```

[18]:
```python
if not skip_training:
    # YOUR CODE HERE
    # Hyperparameters
    learning_rate = 0.01
    num_epochs = 10

    # Loss function and optimizer
    criterion = nn.CrossEntropyLoss()  # Cross-entropy loss for classification
    optimizer = optim.Adam(net.parameters(), lr=learning_rate)  # Adam optimizer

    # Training loop
    for epoch in range(num_epochs):
        net.train()  # Set the network to training mode
        running_loss = 0.0
```

```
        for inputs, labels in trainloader:
            # Move data to the correct device
            inputs, labels = inputs.to(device), labels.to(device)

            # Zero the parameter gradients
            optimizer.zero_grad()

            # Forward pass
            outputs = net(inputs)

            # Compute the loss
            loss = criterion(outputs, labels)

            # Backward pass
            loss.backward()

            # Update the parameters
            optimizer.step()

            # Accumulate the loss
            running_loss += loss.item()

        # Compute accuracy on the test set
        accuracy = compute_accuracy(net, testloader)

        # Print epoch statistics
        print(f"Epoch {epoch + 1}/{num_epochs}, Loss: {running_loss /␣
 ↪len(trainloader):.4f}, Accuracy: {accuracy:.4f}")
```

```
Epoch 1/10, Loss: 0.1357, Accuracy: 0.9174
Epoch 2/10, Loss: 0.1225, Accuracy: 0.9174
Epoch 3/10, Loss: 0.1155, Accuracy: 0.9197
Epoch 4/10, Loss: 0.1024, Accuracy: 0.9156
Epoch 5/10, Loss: 0.0982, Accuracy: 0.9193
Epoch 6/10, Loss: 0.0850, Accuracy: 0.9173
Epoch 7/10, Loss: 0.0814, Accuracy: 0.9176
Epoch 8/10, Loss: 0.0744, Accuracy: 0.9149
Epoch 9/10, Loss: 0.0691, Accuracy: 0.9162
Epoch 10/10, Loss: 0.0666, Accuracy: 0.9179
```

[19]:
```
# Save the model to disk (the pth-files will be submitted automatically␣
 ↪together with your notebook)
# Set confirm=False if you do not want to be asked for confirmation before␣
 ↪saving.
if not skip_training:
    tools.save_model(net, '3_resnet.pth', confirm=True)
```

```
Do you want to save the model (type yes to confirm)?  yes
Model saved to 3_resnet.pth.
```

[20]:
```python
if skip_training:
    net = ResNet(n_blocks, n_channels=16)
    tools.load_model(net, '3_resnet.pth', device)
```

[21]:
```python
# Compute the accuracy on the test set
accuracy = compute_accuracy(net, testloader)
print('Accuracy of the network on the test images: %.3f' % accuracy)
n_blocks = sum(type(m) == Block for _, m in net.named_modules())
assert n_blocks == 6, f"Wrong number ({n_blocks}) of blocks used in the network.
 ↪"

assert accuracy > 0.9, "Poor accuracy ({:.3f})".format(accuracy)
print('Success')
```

```
Accuracy of the network on the test images: 0.918
Success
```