2_vgg

April 1, 2025

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

1 Exercise 2. Convolutional networks. VGG-style network.

In the second part you need to train a convolutional neural network with an architecture inspired by a VGG-network (Simonyan & Zisserman, 2015).

[19]:
```
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)
```

2 VGG-style network

Let us now define a convolution neural network with an architecture inspired by the VGG-net.

The architecture: - A block of three convolutional layers with: - 3x3 kernel - 20 output channels - one pixel zero-pading on both sides - 2d batch normalization after each convolutional layer - ReLU nonlinearity after each 2d batch normalization layer - Max pooling layer with 2x2 kernel and stride 2. - A block of three convolutional layers with: - 3x3 kernel - 40 output channels - one pixel zero-pading on both sides - 2d batch normalization after each convolutional layer - ReLU nonlinearity after each 2d batch normalization layer - Max pooling layer with 2x2 kernel and stride 2. - One convolutional layer with: - 3x3 kernel - 60 output channels - *no padding* - 2d batch normalization after the convolutional layer - ReLU nonlinearity after the 2d batch normalization layer - One convolutional layer with: - 1x1 kernel - 40 output channels - *no padding* - 2d batch normalization after the convolutional layer - ReLU nonlinearity after the 2d batch normalization layer - One convolutional layer with: - 1x1 kernel - 20 output channels - *no padding* - 2d batch normalization after the convolutional layer - ReLU nonlinearity after the 2d batch normalization layer - Global average pooling (compute the average value of each channel across all the input locations): - 5x5 kernel (the input of the layer should be 5x5) - A fully-connected layer with 10 outputs (no nonlinearity)

Notes: * Batch normalization is expected to be right after a convolutional layer, before nonlinearity. * We recommend that you check the number of modules with trainable parameters in your network.

[9]:
```
class VGGNet(nn.Module):
    def __init__(self):
        super(VGGNet, self).__init__()
        # YOUR CODE HERE
        # First block: 3 Conv layers with 20 channels, 3x3 kernels, 1 pixel␣
↪padding
        self.block1 = nn.Sequential(
            nn.Conv2d(1, 20, kernel_size=3, padding=1),
            nn.BatchNorm2d(20),
            nn.ReLU(),
            nn.Conv2d(20, 20, kernel_size=3, padding=1),
            nn.BatchNorm2d(20),
            nn.ReLU(),
            nn.Conv2d(20, 20, kernel_size=3, padding=1),
            nn.BatchNorm2d(20),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2)  # Max pooling
        )

        # Second block: 3 Conv layers with 40 channels, 3x3 kernels, 1 pixel␣
↪padding
        self.block2 = nn.Sequential(
            nn.Conv2d(20, 40, kernel_size=3, padding=1),
            nn.BatchNorm2d(40),
            nn.ReLU(),
            nn.Conv2d(40, 40, kernel_size=3, padding=1),
            nn.BatchNorm2d(40),
            nn.ReLU(),
            nn.Conv2d(40, 40, kernel_size=3, padding=1),
            nn.BatchNorm2d(40),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2)  # Max pooling
        )

        # Third block: 1 Conv layer with 60 channels, 3x3 kernel, no padding
        self.conv3 = nn.Sequential(
            nn.Conv2d(40, 60, kernel_size=3, padding=0),
            nn.BatchNorm2d(60),
            nn.ReLU()
        )

        # Fourth block: 1 Conv layer with 40 channels, 1x1 kernel, no padding
```

```python
        self.conv4 = nn.Sequential(
            nn.Conv2d(60, 40, kernel_size=1, padding=0),
            nn.BatchNorm2d(40),
            nn.ReLU()
        )

        # Fifth block: 1 Conv layer with 20 channels, 1x1 kernel, no padding
        self.conv5 = nn.Sequential(
            nn.Conv2d(40, 20, kernel_size=1, padding=0),
            nn.BatchNorm2d(20),
            nn.ReLU()
        )

        # Global Average Pooling and Fully Connected Layer
        self.global_avg_pool = nn.AdaptiveAvgPool2d((1, 1))  # Output size is␣
↪(1, 1)
        self.fc = nn.Linear(20, 10)  # Fully connected layer for 10 outputs

    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.
        """
        # YOUR CODE HERE
        if verbose: print("Input:", x.shape)
        x = self.block1(x)
        if verbose: print("After Block 1:", x.shape)
        x = self.block2(x)
        if verbose: print("After Block 2:", x.shape)
        x = self.conv3(x)
        if verbose: print("After Conv3:", x.shape)
        x = self.conv4(x)
        if verbose: print("After Conv4:", x.shape)
        x = self.conv5(x)
        if verbose: print("After Conv5:", x.shape)

        x = self.global_avg_pool(x)  # Global average pooling
        if verbose: print("After Global Average Pooling:", x.shape)

        x = torch.flatten(x, 1)  # Flatten to (batch_size, 20)
        if verbose: print("After Flatten:", x.shape)
```

```
        x = self.fc(x)  # Fully connected layer
        if verbose: print("After Fully Connected:", x.shape)

        return x
```

[10]:
```
def test_VGGNet_shapes():
    net = VGGNet()
    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(images, verbose=True)
        assert y.shape == torch.Size([trainloader.batch_size, 10]), f"Bad y.
 ↪shape: {y.shape}"

    print('Success')

test_VGGNet_shapes()
```

```
Shape of the input tensor: torch.Size([32, 1, 28, 28])
Input: torch.Size([32, 1, 28, 28])
After Block 1: torch.Size([32, 20, 14, 14])
After Block 2: torch.Size([32, 40, 7, 7])
After Conv3: torch.Size([32, 60, 5, 5])
After Conv4: torch.Size([32, 40, 5, 5])
After Conv5: torch.Size([32, 20, 5, 5])
After Global Average Pooling: torch.Size([32, 20, 1, 1])
After Flatten: torch.Size([32, 20])
After Fully Connected: torch.Size([32, 10])
Success
```

[11]:
```
# Check the number of layers
def test_vgg_layers():
    net = VGGNet()

    # get gradients for parameters in forward path
    net.zero_grad()
    x = torch.randn(1, 1, 28, 28)
    outputs = net(x)
    outputs[0,0].backward()

    n_conv_layers = sum(1 for module in net.modules()
```

```
                        if isinstance(module, nn.Conv2d) and next(module.
↪parameters()).grad is not None)
    assert n_conv_layers == 9, f"Wrong number of convolutional layers␣
↪({n_conv_layers})"

    n_bn_layers = sum(1 for module in net.modules()
                      if isinstance(module, nn.BatchNorm2d) and next(module.
↪parameters()).grad is not None)
    assert n_bn_layers == 9, f"Wrong number of batch norm layers␣
↪({n_bn_layers})"

    n_linear_layers = sum(1 for module in net.modules()
                          if isinstance(module, nn.Linear) and next(module.
↪parameters()).grad is not None)
    assert n_linear_layers == 1, f"Wrong number of linear layers␣
↪({n_linear_layers})"

    print('Success')

def test_vgg_net():
    net = VGGNet()

    # get gradients for parameters in forward path
    net.zero_grad()
    x = torch.randn(1, 1, 28, 28)
    outputs = net(x)
    outputs[0,0].backward()

    parameter_shapes = sorted(tuple(p.shape) for p in net.parameters() if p.
↪grad is not None)
    print(parameter_shapes)
    expected = [
        (10,), (10, 20), (20,), (20,), (20,), (20,), (20,), (20,), (20,),␣
↪(20,), (20,),
        (20,), (20,), (20,), (20, 1, 3, 3), (20, 20, 3, 3), (20, 20, 3, 3),␣
↪(20, 40, 1, 1),
        (40,), (40,), (40,), (40,), (40,), (40,), (40,), (40,), (40,), (40,),␣
↪(40,), (40,),
        (40, 20, 3, 3), (40, 40, 3, 3), (40, 40, 3, 3), (40, 60, 1, 1), (60,),␣
↪(60,), (60,),
        (60, 40, 3, 3)]
    assert parameter_shapes == expected, "Wrong number of training parameters."

    print('Success')

test_vgg_layers()
```

```
test_vgg_net()
```

```
Success
[(10,), (10, 20), (20,), (20,), (20,), (20,), (20,), (20,), (20,), (20,), (20,),
(20,), (20,), (20,), (20, 1, 3, 3), (20, 20, 3, 3), (20, 20, 3, 3), (20, 40, 1,
1), (40,), (40,), (40,), (40,), (40,), (40,), (40,), (40,), (40,), (40,), (40,),
(40,), (40, 20, 3, 3), (40, 40, 3, 3), (40, 40, 3, 3), (40, 60, 1, 1), (60,),
(60,), (60,), (60, 40, 3, 3)]
Success
```

3 Train the network

[12]:
```
# 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
```

3.0.1 Training loop

Your task is to 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 0.89.

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.

[13]:
```
net = VGGNet()
```

[15]:
```
# Implement the training loop in this cell
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

    # Move the network to the correct device
    net.to(device)

    # 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}")

else:
    print("Training skipped.")
```

```
Epoch 1/10, Loss: 0.2566, Accuracy: 0.9021
```

```
Epoch 2/10, Loss: 0.2349, Accuracy: 0.9039
Epoch 3/10, Loss: 0.2153, Accuracy: 0.9067
Epoch 4/10, Loss: 0.2033, Accuracy: 0.9127
Epoch 5/10, Loss: 0.1907, Accuracy: 0.9129
Epoch 6/10, Loss: 0.1816, Accuracy: 0.9215
Epoch 7/10, Loss: 0.1720, Accuracy: 0.9215
Epoch 8/10, Loss: 0.1641, Accuracy: 0.9201
Epoch 9/10, Loss: 0.1558, Accuracy: 0.9283
Epoch 10/10, Loss: 0.1458, Accuracy: 0.9198
```

[16]:
```python
# 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, '2_vgg_net.pth', confirm=True)
```

```
Do you want to save the model (type yes to confirm)?  yes
```

```
Model saved to 2_vgg_net.pth.
```

[17]:
```python
if skip_training:
    net = VGGNet()
    tools.load_model(net, '2_vgg_net.pth', device)
```

[18]:
```python
# Compute the accuracy on the test set
accuracy = compute_accuracy(net, testloader)
print(f'Accuracy of the VGG net on the test images: {accuracy: .3f}')
assert accuracy > 0.89, 'Poor accuracy'
print('Success')
```

```
Accuracy of the VGG net on the test images:  0.920
Success
```