# CNN

October 3, 2024

## 1 st124092

## 2 Kaung Htet Cho

## 3 Multi-Class Classification with Pytorch

Today, we will

- learn how to build and train a model using Pytorch
- learn about MNIST dataset
- experiment with hyper-parameters tuning

Model development `Life-cycle`: 1. Prepare the `data` 2. Define the `model architecture` **3. Train the model** 4. Evaluate the model 5. Deploy the model

** : what we are going to focus today !

[1]:
```
# For our puffer surver we need to browse via a proxy!!
import os
# Set HTTP and HTTPS proxy
os.environ['http_proxy'] = 'http://192.41.170.23:3128'
os.environ['https_proxy'] = 'http://192.41.170.23:3128'
```

[2]:
```
# !pip3 install tensorboard
# from torch.utils.tensorboard import SummaryWriter
# %load_ext tensorboard
```

[3]:
```
import numpy as np
import torch
import matplotlib.pyplot as plt
```

[4]:
```
import torchvision
from torchvision import datasets, transforms, models
from torch.utils.data import DataLoader, Subset
```

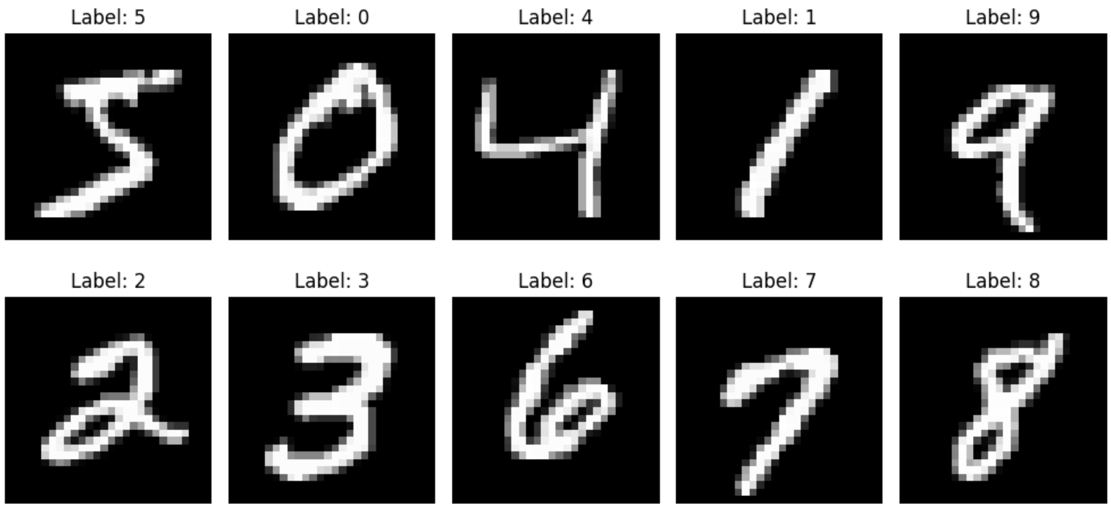
### 3.1 Lets download our data

[5]:
```
# load the training data
transform = torchvision.transforms.Compose([
                                            torchvision.transforms.ToTensor(),
                                            torchvision.transforms.
 ↪Normalize(mean=0.1307, std=0.3081)
                                            ])

train_ds = torchvision.datasets.MNIST(root='.', train=True, download=True,␣
 ↪transform=transform)
test_ds = torchvision.datasets.MNIST(root='.', train=False, download=True,␣
 ↪transform=transform)
```

[6]:
```
print(len(train_ds))
print(len(test_ds))
```

```
60000
10000
```

[7]:
```
# Create a dictionary to store one image per label
images_per_label = {}

# Loop through the dataset to find one image per label
for img, label in train_ds:
    if label not in images_per_label:
        images_per_label[label] = img
    if len(images_per_label) == 10:  # Break the loop once we have all labels
        break

# Plot the images, one per label
fig, axes = plt.subplots(2, 5, figsize=(10, 5))

for i, (label, img) in enumerate(images_per_label.items()):
    ax = axes[i // 5, i % 5]
    ax.imshow(img.squeeze(), cmap='gray')
    ax.set_title(f'Label: {label}')
    ax.axis('off')

plt.tight_layout()
plt.show()
```




[8]:
```python
# Hyperparameters
lr = 0.01
batch_size = 64
num_epoch = 10
classes = ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
```

[9]:
```python
train_ds = list(train_ds)[:10000]
train_loader = torch.utils.data.DataLoader(train_ds,
                                           batch_size=batch_size,
                                           shuffle=True,
                                           num_workers=2)


test_loader = torch.utils.data.DataLoader(test_ds,
                                          batch_size=batch_size,
                                          shuffle=False,
                                          num_workers=2)
```

[10]:
```python
train_ds[0][0].shape
```

[10]: torch.Size([1, 28, 28])

# 4 1. Define the model

# 5 Convolutional Neural Network (CNN)

[11]:
```python
import torch.nn as nn
import torch.nn.functional as F
```

[12]:
```
device = 'cuda:0' if torch.cuda.is_available() else 'cpu'
print(device)
```

```
cuda:0
```

**The model architecture that we are going to build**

Input => conv1 => maxpooling => FC => output

[13]:
```
class MyCNN(nn.Module):            # define own MyCNN which inherits nn.Module␣
 ↪as a base class
  def __init__(self):
    super(MyCNN, self).__init__()
    self.conv1 = nn.Conv2d(1, 16, kernel_size=3) # conv2d(in_channel,␣
 ↪out_channel, kernel_size, stride=1, padding=0, bias=True)
    # (28+2*0-3)/1 + 1 = 26
    self.maxpool = nn.MaxPool2d(2)
    # (26/2)
    self.fc1 = nn.Linear(13*13*16, 10) # Flattened output from convolution␣
 ↪followed by pooling layer

  def forward(self, x):
    x = F.relu(self.conv1(x))
    x = self.maxpool(x)
    x = torch.flatten(x,1) # feature maps flattened to 1D tensor output. second␣
 ↪dimension (1) refers to flattening across the features, leaving the batch␣
 ↪dimension intact.
    x = self.fc1(x)        # produces an output of size 10 (one score for each␣
 ↪class)
    return x, F.log_softmax(x, dim=1) # raw output from x (logits), log of␣
 ↪softmax of x which normalizes the logits into prob.
```

[14]:
```
cnn_model = MyCNN()
cnn_model = cnn_model.to(device)
optimizer = torch.optim.SGD(cnn_model.parameters(), lr=lr)
loss_fn = nn.CrossEntropyLoss()
```

[15]:
```
print(cnn_model)
```

```
MyCNN(
  (conv1): Conv2d(1, 16, kernel_size=(3, 3), stride=(1, 1))
  (maxpool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
  (fc1): Linear(in_features=2704, out_features=10, bias=True)
)
```

[16]:

```
train_losses = []
train_accuracies = []
test_losses = []
test_accuracies = []

def train():
    cnn_model.train()                                          # sets model to␣
 ↪training mode (dropout and batchnorm behaves in this mode)
    train_corr, train_total, train_running_loss = 0, 0, 0  # counters for␣
 ↪tracting trainning accuracy, total examples, running loss

    for step, (data, y) in enumerate(train_loader):        # loops over batch␣
 ↪of data in train_loader
        data, y = data.to(device), y.to(device)
        optimizer.zero_grad()                                  # resets gradients␣
 ↪to prevent accumulation
        _, logits = cnn_model(data)                            # gets the logits
        loss = loss_fn(logits, y)                              # calculates loss␣
 ↪comparing with true label
        loss.backward()                                        # back propagation␣
 ↪is performed to compute gradients
        optimizer.step()                                       # optimizer␣
 ↪updates model params

        y_pred = torch.argmax(logits, 1)                       # selects the␣
 ↪predicted class (the index with the highest value)
        train_corr += torch.sum(torch.eq(y_pred, y).float()).item()      #␣
 ↪counts correct predictions
        train_total += len(data)                               # tracks total no.␣
 ↪of samples
        train_running_loss += loss.item()                      # accumulates loss

    # Calculate average loss and accuracy for this epoch
    epoch_loss = train_running_loss / len(train_loader)
    epoch_accuracy = train_corr / train_total

    # Append to lists for plotting
    train_losses.append(epoch_loss)
    train_accuracies.append(epoch_accuracy)

    print(f'Epoch [{epoch+1}] Train Loss: {epoch_loss:.4f}, Accuracy:␣
 ↪{epoch_accuracy:.4f}')

##########################################################################################

def test():
```

```
    cnn_model.eval()                                        # sets model to␣
↪evalutaion mode
    test_corr, test_total, test_running_loss = 0, 0, 0
    with torch.no_grad():
        for step, (data, y) in enumerate(test_loader):
            data, y = data.to(device), y.to(device)
            _, logits = cnn_model(data)
            loss = loss_fn(logits, y)
            y_pred = torch.argmax(logits, 1)
            test_corr += torch.sum(torch.eq(y_pred, y).float()).item()
            test_total += len(data)
            test_running_loss += loss.item()
    # Calculate average loss and accuracy for this epoch
    epoch_loss = test_running_loss / len(test_loader)
    epoch_accuracy = test_corr / test_total

    # Append to lists for plotting
    test_losses.append(epoch_loss)
    test_accuracies.append(epoch_accuracy)

    print(f'Epoch [{epoch+1}] Test Loss: {epoch_loss:.4f}, Accuracy:␣
↪{epoch_accuracy:.4f}')
```

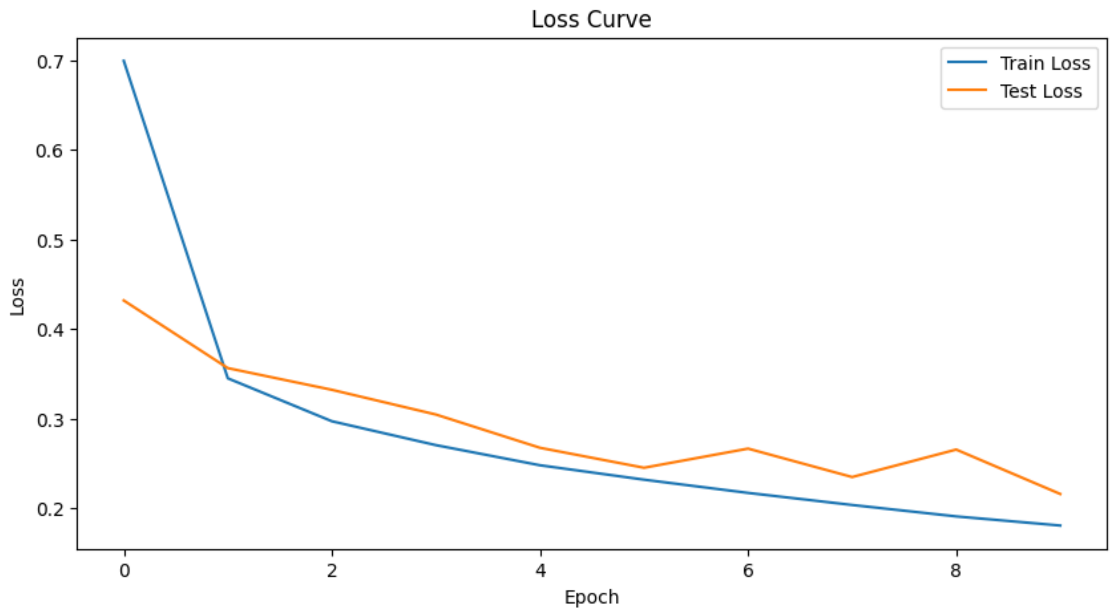
[17]:
```
for epoch in range(num_epoch):
  print(f"--------- Train EPOCH {epoch} -------------")
  train()
  print(f"--------- Test EPOCH {epoch} -------------")
  test()
```

```
--------- Train EPOCH 0 -------------
Epoch [1] Train Loss: 0.6997, Accuracy: 0.8257
--------- Test EPOCH 0 -------------
Epoch [1] Test Loss: 0.4318, Accuracy: 0.8756
--------- Train EPOCH 1 -------------
Epoch [2] Train Loss: 0.3448, Accuracy: 0.9017
--------- Test EPOCH 1 -------------
Epoch [2] Test Loss: 0.3562, Accuracy: 0.8941
--------- Train EPOCH 2 -------------
Epoch [3] Train Loss: 0.2968, Accuracy: 0.9146
--------- Test EPOCH 2 -------------
Epoch [3] Test Loss: 0.3320, Accuracy: 0.9017
--------- Train EPOCH 3 -------------
Epoch [4] Train Loss: 0.2701, Accuracy: 0.9239
--------- Test EPOCH 3 -------------
Epoch [4] Test Loss: 0.3043, Accuracy: 0.9107
--------- Train EPOCH 4 -------------
Epoch [5] Train Loss: 0.2476, Accuracy: 0.9314
--------- Test EPOCH 4 -------------
```

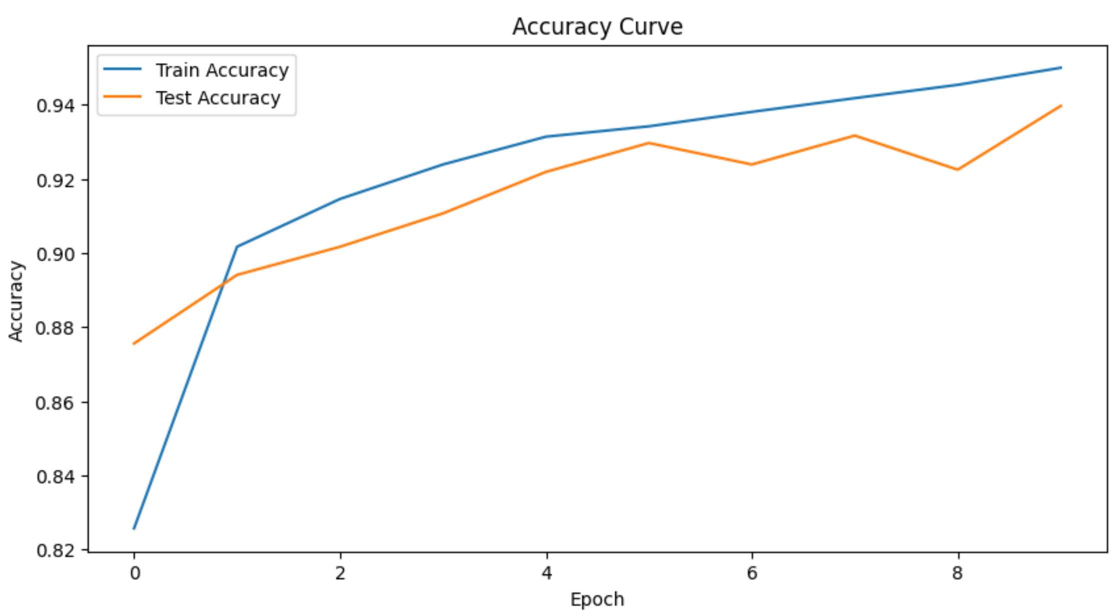
```
Epoch [5] Test Loss: 0.2672, Accuracy: 0.9219
--------- Train EPOCH 5 -------------
Epoch [6] Train Loss: 0.2315, Accuracy: 0.9342
--------- Test EPOCH 5 -------------
Epoch [6] Test Loss: 0.2448, Accuracy: 0.9297
--------- Train EPOCH 6 -------------
Epoch [7] Train Loss: 0.2167, Accuracy: 0.9381
--------- Test EPOCH 6 -------------
Epoch [7] Test Loss: 0.2662, Accuracy: 0.9239
--------- Train EPOCH 7 -------------
Epoch [8] Train Loss: 0.2032, Accuracy: 0.9418
--------- Test EPOCH 7 -------------
Epoch [8] Test Loss: 0.2345, Accuracy: 0.9317
--------- Train EPOCH 8 -------------
Epoch [9] Train Loss: 0.1905, Accuracy: 0.9454
--------- Test EPOCH 8 -------------
Epoch [9] Test Loss: 0.2652, Accuracy: 0.9225
--------- Train EPOCH 9 -------------
Epoch [10] Train Loss: 0.1803, Accuracy: 0.9500
--------- Test EPOCH 9 -------------
Epoch [10] Test Loss: 0.2155, Accuracy: 0.9397
```

[18]:

```
# Plot the training and test loss
plt.figure(figsize=(10, 5))
plt.plot(train_losses, label='Train Loss')
plt.plot(test_losses, label='Test Loss')
plt.title('Loss Curve')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.show()
```

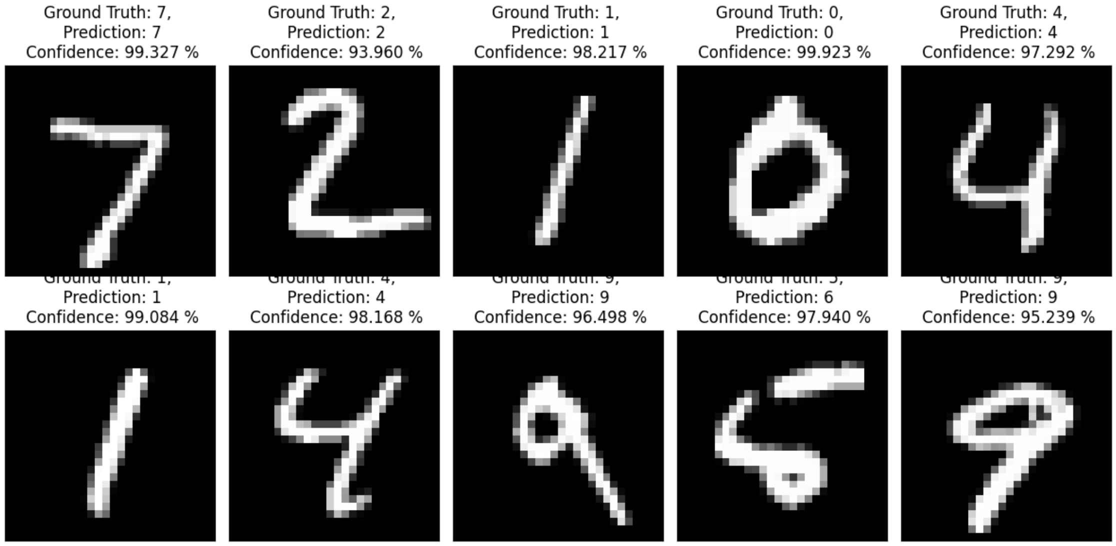



[19]:
```
# Plot the training and test accuracy
plt.figure(figsize=(10, 5))
plt.plot(train_accuracies, label='Train Accuracy')
plt.plot(test_accuracies, label='Test Accuracy')
plt.title('Accuracy Curve')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```



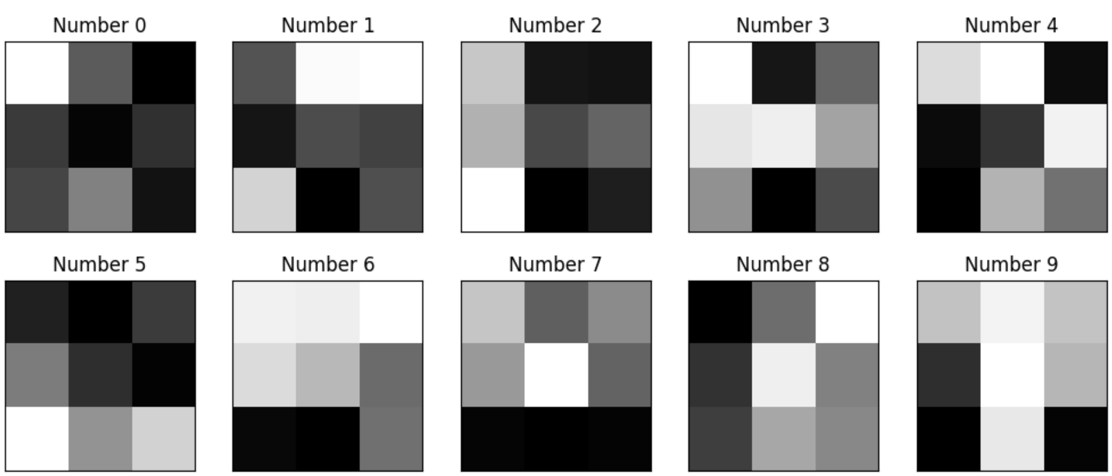
### 5.0.1 Plot some prediction

[20]:
```
cnn_model.eval()
data, y = next(iter(test_loader))

# 1. push the data to the selected device
data, y = data.to(device), y.to(device)

# 2. feed the data into the model and the model makes predictions
_, logits = cnn_model(data)    # raw prediction before applying softmax ;␣
↪unnormalised scores for each class

# 3. get the class with highest prob.
y_pred = torch.argmax(logits, 1) # finds the index of the class with the␣
↪highest value (i.e., the predicted class) along dimension 1,

get_prob = torch.nn.Softmax(dim=1) # converts logits into probabilities that␣
↪sums to 1
prob = get_prob(logits)            # prob is a tensor where each row␣
↪corresponds to a sample, and each column contains the probability of that␣
↪sample belonging to a particular class.

# Plot
fig = plt.figure(figsize=(12,6))
for i in range(10):
    plt.subplot(2, 5, i+1)
    plt.imshow(data[i].cpu().detach().numpy().reshape((28,28)), cmap='gray')

    # detach(): Detaches the tensor from the computation graph, so no gradients␣
↪are tracked.
    # cpu(): Moves the tensor back to the CPU (important if you're using a GPU).
    # numpy(): Converts the tensor to a NumPy array.

    plt.title(f"Ground Truth: {y[i].cpu().detach().numpy()}, \n  Prediction:␣
↪{y_pred[i].cpu().detach().numpy()} \n Confidence: {prob[i][y_pred[i]] * 100:.
↪3F} %")
    plt.xticks([])
    plt.yticks([])
plt.tight_layout() # Adjusts the subplot parameters to make sure that subplots␣
↪fit into the figure area nicely, avoiding overlaps.
plt.show()
```



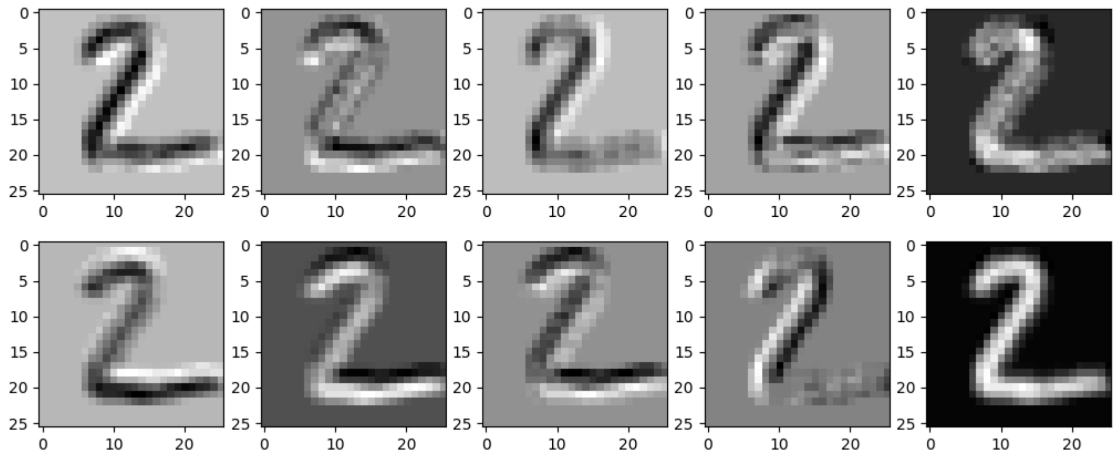

### 5.0.2 Visualize filter weights

[21]:
```
fig = plt.figure(figsize=[5*2.5, 2*2.5])
for i in range(10):                                    # loops through first 10 filters, Each filter corresponds to a learned weight matrix, which is applied to the input image.
  ax = fig.add_subplot(2, 5, i+1)
  # access the ith weight, reshapes to 3*3 matrix
  ws = cnn_model.conv1.weight[i].reshape([3, 3]).cpu().detach().numpy() # Change here if your filter size is changed
  ax.imshow(ws, cmap='gray')
  plt.title(f"Number {i}")
  plt.xticks([])
  plt.yticks([])
```



### 5.0.3 Visualize feature map

[22]:
```
# Visualize feature maps
activation = {}          # initializes empty dictionary to store the feature maps

# returns a hook function,
def get_activation(name):
    # hook will capture the layer's output (output) and store it in the␣
↪activation dictionary with the specified name.
    def hook(model, input, output):
        activation[name] = output.detach()
    return hook

cnn_model.conv1.register_forward_hook(get_activation('conv1')) # This line␣
↪registers the hook on the first convolutional layer (conv1).
data, _ = test_ds[1]                                            # retrieves a␣
↪single test sample from the dataset test_ds (data,label)
data.unsqueeze_(0)     # Since this is a single image, unsqueeze_(0) changes␣
↪its shape from [1, 28, 28] to [1, 1, 28, 28], where 1 is the batch size
output = cnn_model(data.to(device))  # output is not required for this case␣
↪since we stored activations

fm_cov1 = activation['conv1'].squeeze().cpu().detach().numpy()  # .squeeze():␣
↪Removes the extra batch dimension added earlier, so the feature map has the␣
↪shape [out_channels, height, width].
fig = plt.figure(figsize=[5*2.5, 2*2.5])
for i in range(10):
  ax = fig.add_subplot(2, 5, i+1)
  ax.imshow(fm_cov1[i], cmap='gray')

# The feature maps are the result of applying the learned filters to the input␣
↪image, so they represent specific patterns or structures detected by the␣
↪filters.
```



## 5.1 Lets now try using RESNET18 model

- We can load a pretrained model on Imagenet dataset or train from scratch. For now we are using a pretrained model.

### 5.1.1 Keypoints:

1. Customizing the Final Layer: Since ResNet-18's final fully connected (fc) layer is designed for ImageNet (1000 classes), we modify it to suit our dataset by setting resnet18.fc = nn.Linear(resnet18.fc.in_features, num_classes).
2. Transformations: The input image size for ResNet-18 is 224x224, so we resize the CIFAR-10 images (originally 32x32) using transforms.Resize(224).
3. Training and Testing: The model is trained using train_model and evaluated using test_model.

[23]:
```
# Load ResNet-18 pre-trained model
from torchvision.models import ResNet18_Weights
resnet18 = models.resnet18(weights=ResNet18_Weights.IMAGENET1K_V1) # or use .
 ↪DEFAULT for the latest weights

# This code will raise the deprecation warning:
# model = models.resnet18(pretrained=True)
```

[24]:
```
# Modify the final layer to match the number of classes (for example, CIFAR-10␣
 ↪has 10 classes)
num_classes = 10
resnet18.fc = nn.Linear(resnet18.fc.in_features, num_classes)
```

[25]:
```
# Transfer the model to the GPU if available
resnet18 = resnet18.to(device)
```

```
# Define transforms (resize to 224x224 since ResNet expects that input size)
transform = transforms.Compose([
    transforms.Resize(224),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])
```

[26]:
```
# Download CIFAR-10 dataset (or use your own dataset)
train_dataset = datasets.CIFAR10(root='./data', train=True,␣
 ↪transform=transform, download=True)
test_dataset = datasets.CIFAR10(root='./data', train=False,␣
 ↪transform=transform, download=True)



# Subsample the training and test datasets

# Function to subsample CIFAR-10 dataset
def subsample_dataset(dataset, sample_size=1000):
    indices = np.random.choice(len(dataset), sample_size, replace=False)
    subset = Subset(dataset, indices)
    return subset

sample_size = 1000
train_subset = subsample_dataset(train_dataset, sample_size=sample_size)
test_subset = subsample_dataset(train_dataset, sample_size=int(sample_size * 0.
 ↪4))

# Load the data
train_loader = torch.utils.data.DataLoader(dataset=train_subset, batch_size=64,␣
 ↪shuffle=True)
test_loader = torch.utils.data.DataLoader(dataset=test_subset, batch_size=64,␣
 ↪shuffle=False)
```

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

[27]:
```
print(resnet18)
```

```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3),
bias=False)
  (bn1): BatchNorm2d(64, 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)
  (layer1): Sequential(
```

```
    (0): BasicBlock(
      (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)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
    (1): BasicBlock(
      (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)
      (relu): ReLU(inplace=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)
    )
  )
  (layer2): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1,
1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
```

```
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (layer3): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1,
1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (layer4): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1,
1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
```

```
        )
      )
      (1): BasicBlock(
        (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
        (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
        (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Linear(in_features=512, out_features=10, bias=True)
)
```

[28]:

```
# Define loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(resnet18.parameters(), lr=0.001)

# Training function
def train_model(model, train_loader, criterion, optimizer, num_epochs=5):
    model.train()
    for epoch in range(num_epochs):
        running_loss = 0.0
        for images, labels in train_loader:
            images, labels = images.to(device), labels.to(device)

            # Forward pass
            outputs = model(images)
            loss = criterion(outputs, labels)

            # Backward pass and optimization
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            running_loss += loss.item()

        print(f"Epoch [{epoch+1}/{num_epochs}], Loss: {running_loss/
↪len(train_loader):.4f}")
```

[29]:

```
def test_model(model, test_loader):
    model.eval()
    correct = 0
```

```
    total = 0
    with torch.no_grad():
        for images, labels in test_loader:
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    print(f'Accuracy of the model on the test images: {100 * correct / total:.
↪2f}%')
```

[30]:
```
# Training the model
train_model(resnet18, train_loader, criterion, optimizer, num_epochs=5)
```

```
Epoch [1/5], Loss: 1.2798
Epoch [2/5], Loss: 0.5322
Epoch [3/5], Loss: 0.2812
Epoch [4/5], Loss: 0.1641
Epoch [5/5], Loss: 0.0950
```

[31]:
```
# Testing the trained model
test_model(resnet18, test_loader)
```

```
Accuracy of the model on the test images: 73.50%
```

### 5.2 Fine-tuning vs. Feature Extraction

- **Fine-tuning**: During fine-tuning, you update the weights of **all layers** in the network during training. This is typically done when you want to adapt a pre-trained model to a new task. By default, when calling `optimizer.step()` on all parameters, the weights of all layers are updated.
- **Feature Extraction**: In feature extraction, you freeze the weights of the pre-trained layers and only train the final layer (or a few newly added layers). This allows the model to use the learned features from the pre-trained network while adjusting the output to the new task.

To freeze the layers in PyTorch, you can set the `requires_grad` attribute of the parameters to `False` like this:

"'python for param in resnet18.parameters(): param.requires_grad = False

## 6 Take Home Exercise

## 7 Assignment: Custom CNN on MNIST Dataset

### 7.1 Objective

In this assignment, you will implement and train a custom Convolutional Neural Network (CNN) to classify handwritten digits from the MNIST dataset.

---

## 7.2 Task Breakdown

### 7.2.1 Part 1: Data Preparation

1. **Load and Visualize Data**:

    - Load the MNIST dataset using PyTorch's `torchvision.datasets` and visualize some sample images along with their labels.
    - **Deliverable**: Submit a grid of at least 25 sample images and their respective labels from the MNIST dataset.

### 7.2.2 Part 2: Custom CNN Implementation

1. **Define Your CNN Model**:

    - Implement a custom CNN class using PyTorch's `torch.nn.Module`. The network should consist of:
        - 2 convolutional layers
        - 2 max-pooling layers
        - At least 1 fully connected layer
        - Use ReLU activations and appropriate dropout layers.

    **Hints**:

    - First conv layer: Input channels = (grayscale image), Output channels = 16, Kernel size = 3x3.
    - Second conv layer: Output channels = 32, Kernel size = 3x3.
    - Max-pooling layers with 2x2 window.

    **Deliverable**: Submit your `MyCNN` class code.

### 7.2.3 Part 3: Model Training and Evaluation

1. **Training the Model**:

    - Train the CNN on the MNIST training set.
    - Use Cross-Entropy Loss and an optimizer (e.g., SGD or Adam).
    - Plot the training and validation loss curves over 10-20 epochs.

    **Deliverable**: Submit code for training and a plot showing the loss curves.

2. **Accuracy Evaluation**:

    - Evaluate the model on the MNIST test set and report the accuracy.

    **Deliverable**: Submit the final accuracy on the test set.

### 7.2.4 Part 4: Visualization

1. **Feature Map Visualization**:
    - Register a forward hook on the second convolutional layer to capture the feature maps.

- Plot the feature maps for a few random test images (at least 3 different digits).

**Deliverable**: Submit code and images showing feature maps from the second convolutional layer.

---

### 7.3 Submission Requirements

- Submit a Jupyter Notebook with the following sections:
  1. Data Loading and Visualization
  2. CNN Model Implementation
  3. Training and Loss Curves
  4. Final Test Accuracy
  5. Feature Map Visualizations

### 7.4 Evaluation Criteria

- Correctness and clarity of the CNN implementation.
- Proper visualizations of data, loss and accuracy curves, and feature maps.

#### 7.4.1 Part 1: Data Preparation

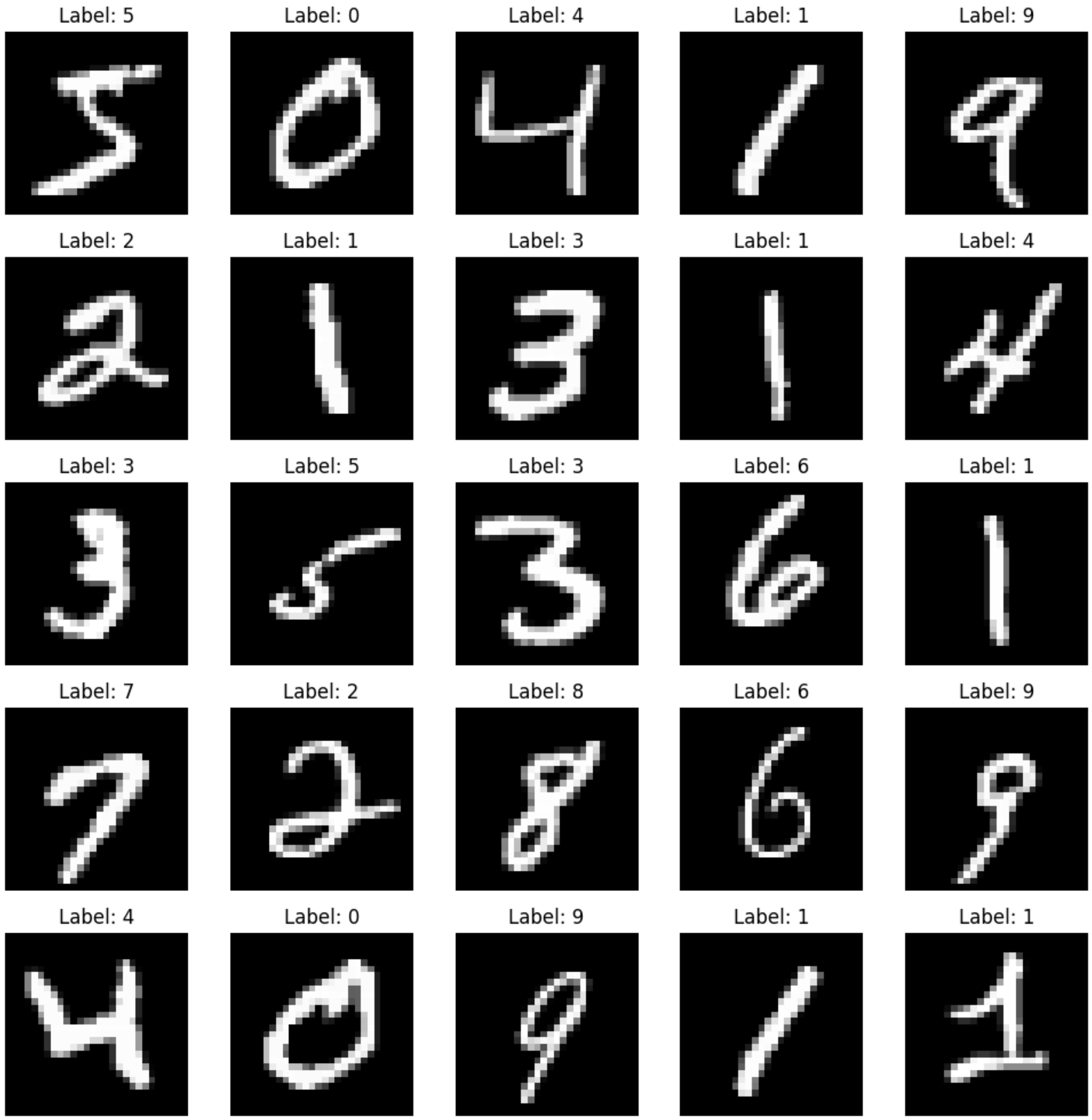
1. **Load and Visualize Data**:
   - Load the MNIST dataset using PyTorch's `torchvision.datasets` and visualize some sample images along with their labels.
   - **Deliverable**: Submit a grid of at least 25 sample images and their respective labels from the MNIST dataset.

[32]:
```
import numpy as np
import torch
import matplotlib.pyplot as plt
```

[33]:
```
import torchvision
from torchvision import datasets, transforms, models
from torch.utils.data import DataLoader, Subset

# load the training data
transform = torchvision.transforms.Compose([
                                            torchvision.transforms.ToTensor(),
                                            torchvision.transforms.
 ↪Normalize(mean=0.1307, std=0.3081)
                                            ])

train_ds = torchvision.datasets.MNIST(root='.', train=True, download=True,␣
 ↪transform=transform)
test_ds = torchvision.datasets.MNIST(root='.', train=False, download=True,␣
 ↪transform=transform)
```

[63]:
```python
# Create a list to store 25 random images and their labels
random_images = []
random_labels = []

# Loop through the dataset to find 25 random images
for img, label in train_ds:
    random_images.append(img)
    random_labels.append(label)
    if len(random_images) == 25:  # Break the loop once we have 25 images
        break

# Plot the 25 images
fig, axes = plt.subplots(5, 5, figsize=(10, 10))
print(f'type_random_images: {type(random_images[0])}')
print(f'shape_random_images: {random_images[0].shape}')

print(f'shape_random_images_squeeze: {random_images[0].squeeze().shape}')


for i in range(25):
    ax = axes[i // 5, i % 5]
    ax.imshow(random_images[i].squeeze(), cmap='gray')
    ax.set_title(f'Label: {random_labels[i]}')
    ax.axis('off')

plt.tight_layout()
plt.show()
```

```
type_random_images: <class 'torch.Tensor'>
shape_random_images: torch.Size([1, 28, 28])
shape_random_images_squeeze: torch.Size([28, 28])
```




### 7.4.2 Part 2: Custom CNN Implementation

1. **Define Your CNN Model**:

   - Implement a custom CNN class using PyTorch's `torch.nn.Module`. The network should consist of:
     - 2 convolutional layers
     - 2 max-pooling layers
     - At least 1 fully connected layer
     - Use ReLU activations and appropriate dropout layers.

   **Hints**:

- First conv layer: Input channels = (grayscale image), Output channels = 16, Kernel size = 3x3.
- Second conv layer: Output channels = 32, Kernel size = 3x3.
- Max-pooling layers with 2x2 window.

**Deliverable**: Submit your `MyCNN` class code.

[35]:
```python
import torch
import torch.nn as nn
import torch.nn.functional as F
```

[36]:
```python
device = 'cuda' if torch.cuda.is_available() else 'cpu'
device
```

[36]: 'cuda'

[37]:
```python
class MyCNN(nn.Module):
    def __init__(self):
        super(MyCNN,self).__init__()
        # First Convolution Block
        self.conv1 = nn.Conv2d(1, 16, kernel_size=3)  # 28x28x1 -> 26x26x16
        self.dropout1 = nn.Dropout2d(0.25)

        # Second Convolution Block
        self.conv2 = nn.Conv2d(16, 32, kernel_size=3)  # 13x13x16 -> 11x11x32
        self.dropout2 = nn.Dropout2d(0.25)

        # Fully Connected Layers
        self.fc1 = nn.Linear(5*5*32, 128)  # Calculated from final conv output
        self.dropout3 = nn.Dropout(0.5)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        # First Convolution Block
        x = self.conv1(x)          # Convolution
        x = F.relu(x)              # ReLU activation
        x = F.max_pool2d(x, 2)     # Max pooling
        x = self.dropout1(x)       # Dropout

        # Second Convolution Block
        x = self.conv2(x)          # Convolution
        x = F.relu(x)              # ReLU activation
        x = F.max_pool2d(x, 2)     # Max pooling
        x = self.dropout2(x)       # Dropout

        # Flatten
        x = torch.flatten(x, 1)    # Flatten all dimensions except batch
```

```
        # Fully Connected Layers
        x = self.fc1(x)             # First FC layer
        x = F.relu(x)               # ReLU activation
        x = self.dropout3(x)        # Dropout
        x = self.fc2(x)             # Output layer

        return x, F.log_softmax(x, dim=1)
```

### 7.4.3 Part 3: Model Training and Evaluation

1. **Training the Model**:
    - Train the CNN on the MNIST training set.
    - Use Cross-Entropy Loss and an optimizer (e.g., SGD or Adam).
    - Plot the training and validation loss curves over 10-20 epochs.

   **Deliverable**: Submit code for training and a plot showing the loss curves.

2. **Accuracy Evaluation**:
    - Evaluate the model on the MNIST test set and report the accuracy.

   **Deliverable**: Submit the final accuracy on the test set.

[38]:
```
cnn_model = MyCNN()
cnn_model = cnn_model.to(device)

optimizer = torch.optim.SGD(cnn_model.parameters(), lr=lr)
loss_fn = nn.CrossEntropyLoss()
```

[39]:
```
cnn_model
```

[39]:
```
MyCNN(
  (conv1): Conv2d(1, 16, kernel_size=(3, 3), stride=(1, 1))
  (dropout1): Dropout2d(p=0.25, inplace=False)
  (conv2): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1))
  (dropout2): Dropout2d(p=0.25, inplace=False)
  (fc1): Linear(in_features=800, out_features=128, bias=True)
  (dropout3): Dropout(p=0.5, inplace=False)
  (fc2): Linear(in_features=128, out_features=10, bias=True)
)
```

[40]:
```
# Hyperparameters
lr = 0.01
batch_size = 64
num_epoch = 20
classes = ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
```

[41]:
```
train_losses = []
train_accuracies = []
```

```
test_losses = []
test_accuracies = []

def train():
    cnn_model.train()                                        # sets model to␣
↪training mode (dropout and batchnorm behaves in this mode)
    train_corr, train_total, train_running_loss = 0, 0, 0  # counters for␣
↪tracting trainning accuracy, total examples, running loss

    for step, (data, y) in enumerate(train_loader):        # loops over batch␣
↪of data in train_loader
        data, y = data.to(device), y.to(device)
        optimizer.zero_grad()                               # resets gradients␣
↪to prevent accumulation
        _, logits = cnn_model(data)                         # gets the logits
        loss = loss_fn(logits, y)                           # calculates loss␣
↪comparing with true label
        loss.backward()                                     # back propagation␣
↪is performed to compute gradients
        optimizer.step()                                    # optimizer␣
↪updates model params

        y_pred = torch.argmax(logits, 1)                    # selects the␣
↪predicted class (the index with the highest value)
        train_corr += torch.sum(torch.eq(y_pred, y).float()).item()      #␣
↪counts correct predictions
        train_total += len(data)                            # tracks total no.␣
↪of samples
        train_running_loss += loss.item()                   # accumulates loss

    # Calculate average loss and accuracy for this epoch
    epoch_loss = train_running_loss / len(train_loader)
    epoch_accuracy = train_corr / train_total

    # Append to lists for plotting
    train_losses.append(epoch_loss)
    train_accuracies.append(epoch_accuracy)

    print(f'Epoch [{epoch+1}] Train Loss: {epoch_loss:.4f}, Accuracy:␣
↪{epoch_accuracy:.4f}')

###########################################################################################
```

[54]:
```
def test(cnn_model, test_loader):
    cnn_model.eval()
```

```
    test_corr, test_total = 0, 0

    with torch.no_grad():  # Disable gradient calculation
        for data, y in test_loader:
            data, y = data.to(device), y.to(device)

            # cnn_model returns a tuple, and logits is the second element
            output = cnn_model(data)
            if isinstance(output, tuple):
                logits = output[1]  # Adjust this index if necessary
            else:
                logits = output  # If it directly returns logits

            y_pred = torch.argmax(logits, 1)
            test_corr += (y_pred == y).sum().item()
            test_total += y.size(0)

    accuracy = test_corr / test_total

    print(f'Accuracy of the model on the test set: {100 * accuracy:.2f}%')
```

[43]:
```
train_ds = list(train_ds)
train_loader = torch.utils.data.DataLoader(train_ds,
                                           batch_size=batch_size,
                                           shuffle=True,
                                           num_workers=2)

test_loader = torch.utils.data.DataLoader(test_ds,
                                          batch_size=batch_size,
                                          shuffle=False,
                                          num_workers=2)
```

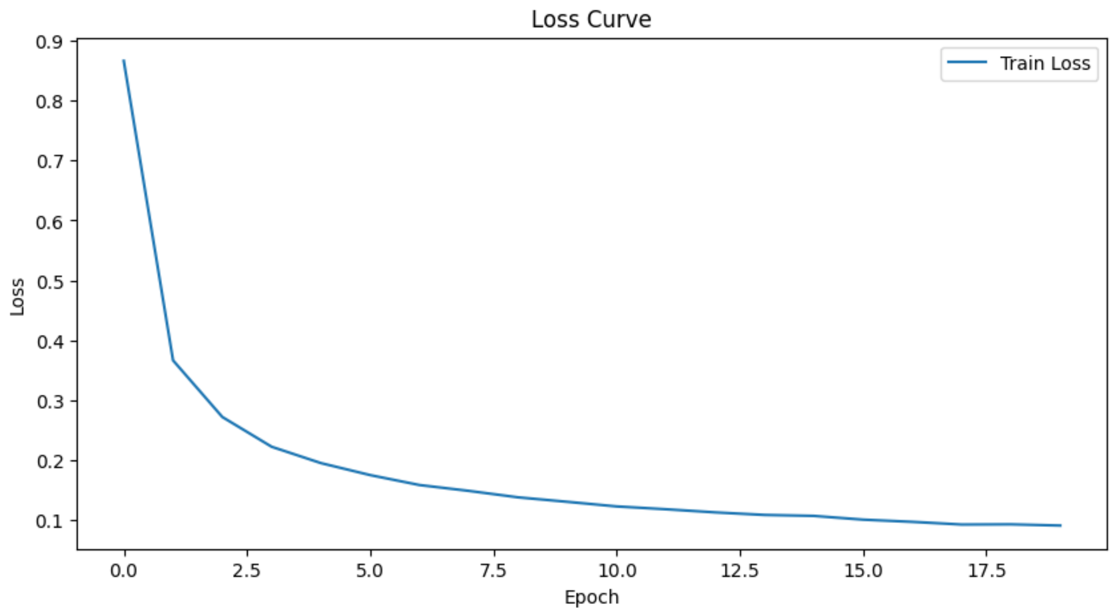
[44]:
```
for epoch in range(20):
  print(f"--------- Train EPOCH {epoch} -------------")
  train()
```

```
--------- Train EPOCH 0 -------------
Epoch [1] Train Loss: 0.8664, Accuracy: 0.7271
--------- Train EPOCH 1 -------------
Epoch [2] Train Loss: 0.3667, Accuracy: 0.8899
--------- Train EPOCH 2 -------------
Epoch [3] Train Loss: 0.2722, Accuracy: 0.9207
--------- Train EPOCH 3 -------------
Epoch [4] Train Loss: 0.2226, Accuracy: 0.9341
--------- Train EPOCH 4 -------------
Epoch [5] Train Loss: 0.1954, Accuracy: 0.9423
--------- Train EPOCH 5 -------------
Epoch [6] Train Loss: 0.1753, Accuracy: 0.9483
```

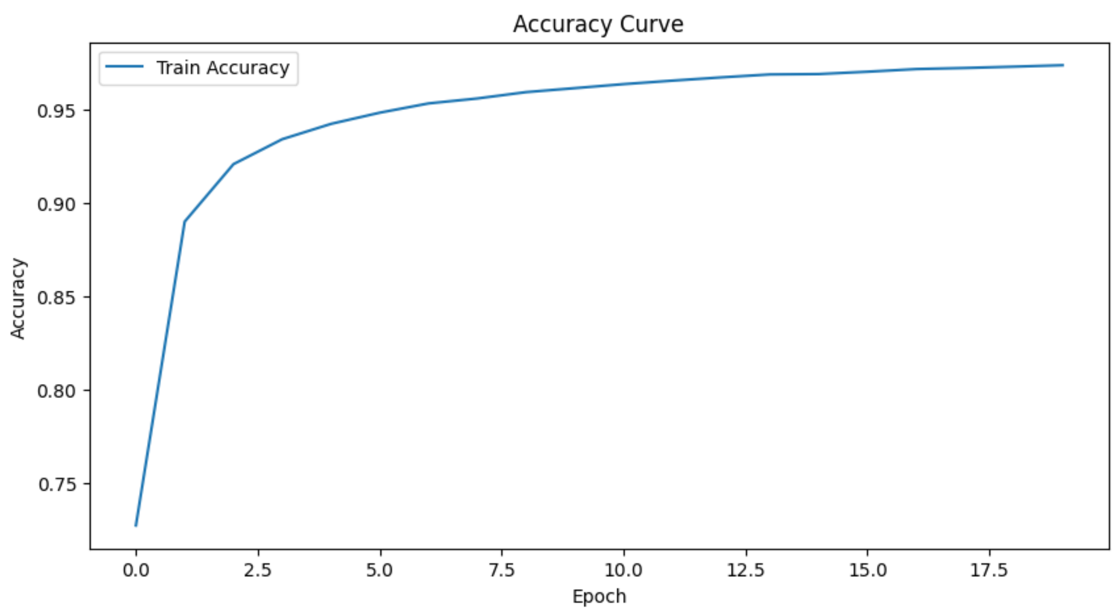
```
--------- Train EPOCH 6 -------------
Epoch [7] Train Loss: 0.1588, Accuracy: 0.9533
--------- Train EPOCH 7 -------------
Epoch [8] Train Loss: 0.1490, Accuracy: 0.9559
--------- Train EPOCH 8 -------------
Epoch [9] Train Loss: 0.1382, Accuracy: 0.9593
--------- Train EPOCH 9 -------------
Epoch [10] Train Loss: 0.1308, Accuracy: 0.9615
--------- Train EPOCH 10 -------------
Epoch [11] Train Loss: 0.1231, Accuracy: 0.9636
--------- Train EPOCH 11 -------------
Epoch [12] Train Loss: 0.1184, Accuracy: 0.9655
--------- Train EPOCH 12 -------------
Epoch [13] Train Loss: 0.1131, Accuracy: 0.9672
--------- Train EPOCH 13 -------------
Epoch [14] Train Loss: 0.1089, Accuracy: 0.9688
--------- Train EPOCH 14 -------------
Epoch [15] Train Loss: 0.1073, Accuracy: 0.9690
--------- Train EPOCH 15 -------------
Epoch [16] Train Loss: 0.1010, Accuracy: 0.9702
--------- Train EPOCH 16 -------------
Epoch [17] Train Loss: 0.0973, Accuracy: 0.9717
--------- Train EPOCH 17 -------------
Epoch [18] Train Loss: 0.0929, Accuracy: 0.9722
--------- Train EPOCH 18 -------------
Epoch [19] Train Loss: 0.0932, Accuracy: 0.9730
--------- Train EPOCH 19 -------------
Epoch [20] Train Loss: 0.0912, Accuracy: 0.9737
```

[45]:

```python
# Plot the training and test loss
plt.figure(figsize=(10, 5))
plt.plot(train_losses, label='Train Loss')
plt.title('Loss Curve')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.show()
```




[46]:
```
# Plot the training and test accuracy
plt.figure(figsize=(10, 5))
plt.plot(train_accuracies, label='Train Accuracy')
plt.title('Accuracy Curve')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```



[56]:
```
test(cnn_model,test_loader)
```

```
Accuracy of the model on the test set: 98.82%
```

### 7.4.4 Part 4: Visualization

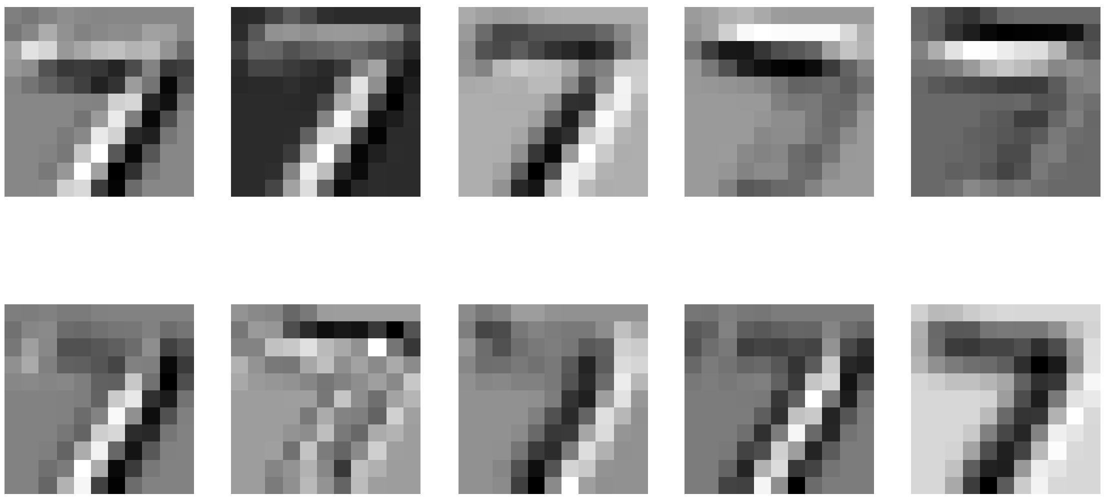
1. **Feature Map Visualization**:
   - Register a forward hook on the second convolutional layer to capture the feature maps.
   - Plot the feature maps for a few random test images (at least 3 different digits).

   **Deliverable**: Submit code and images showing feature maps from the second convolutional layer.

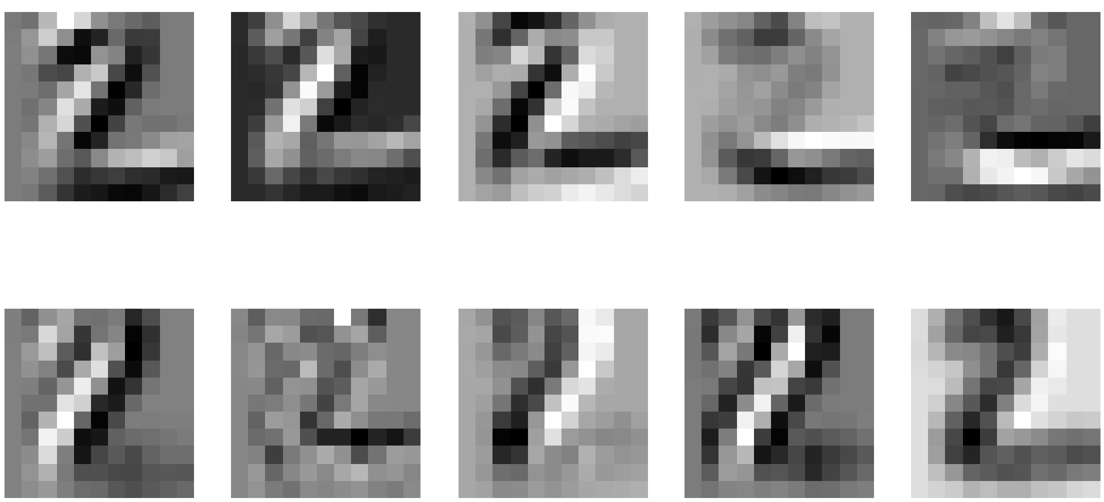
[62]:
```
import torch
import matplotlib.pyplot as plt

activation = {}

def get_activation(name):
    def hook(model, input, output):
        activation[name] = output.detach()
    return hook

cnn_model.conv2.register_forward_hook(get_activation('conv2'))

num_images = 3

for idx in range(num_images):
    data, label = test_ds[idx]   # Get a random test sample
    data.unsqueeze_(0)           # Add batch dimension, so shape becomes [1, 1,␣
 ↪28, 28]

    # Forward pass through the network
    output = cnn_model(data.to(device))

    # Get the feature map from conv2 layer
    fm_conv2 = activation['conv2'].squeeze().cpu().detach().numpy()  # Shape:␣
 ↪[num_channels, height, width]

    fig = plt.figure(figsize=(10, 5))

    # Plot the first 10 feature maps
    for i in range(10):
        ax = fig.add_subplot(2, 5, i+1)
        ax.imshow(fm_conv2[i], cmap='gray')
        ax.axis('off')
```

```
    plt.suptitle(f"Feature maps from conv2 for test image {idx} (Label: {label})")
    plt.show()
```

Feature maps from conv2 for test image 0 (Label: 7)



Feature maps from conv2 for test image 1 (Label: 2)



Feature maps from conv2 for test image 2 (Label: 1)