limhpone / computervision-final-prep

Code Issues Pull requests Actions Projects Wiki Security In

computervision-final-prep / lab / Lab 07 (CNN)-20251128 / lab7_CNN.ipynb

limhpone Lab 7 40d7fbe · 2 hours ago

994 lines (994 loc) · 309 KB

Preview Code Blame Raw

CNN Classification

In this lab, we will

- learn how to build and train a CNN 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

In [1]:

```
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import DataLoader, Subset
import torchvision
from torchvision import datasets, transforms, models
import matplotlib.pyplot as plt
import time
```

Lets download MNIST data

In [2]:

```
# Load the training data
transform = torchvision.transforms.Compose([
                                            torchvision.transforms.ToTensor(),
                                            torchvision.transforms.Normalize(m
                                            ])

train_ds = torchvision.datasets.MNIST(root='./MNIST', train=True, download=T
test_ds = torchvision.datasets.MNIST(root='./MNIST', train=False, download=T
```

In [3]:

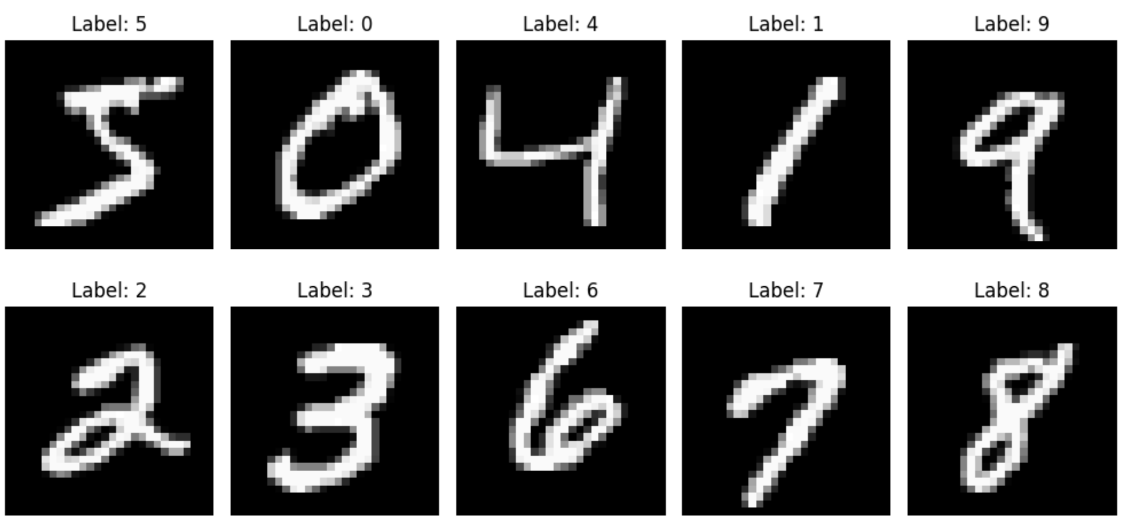
```
print(len(train_ds))
print(len(test_ds))
```

```
60000
10000
```

In [4]:

```
# Create a dictionary to store one image per label
images_per_label = {}
```

```python
# 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 label
        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()
```



Initialize HyperParams

In [5]:

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

In [6]:

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

```
cuda:0
```

In [7]:

```python
# keep original before truncating
full_train = list(train_ds)

train_ds  = full_train[:10000]
valid_ds  = full_train[10000:12000]
```

```
valid_ds   = full_train[10000:12000]
test_ds    = list(test_ds)[:2000]

train_loader = torch.utils.data.DataLoader(train_ds,
                                           batch_size=batch_size,
                                           shuffle=True,
                                           num_workers=2)

valid_loader = torch.utils.data.DataLoader(valid_ds,
                                           batch_size=batch_size,
                                           shuffle=False,
                                           num_workers=2)

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

In [8]:

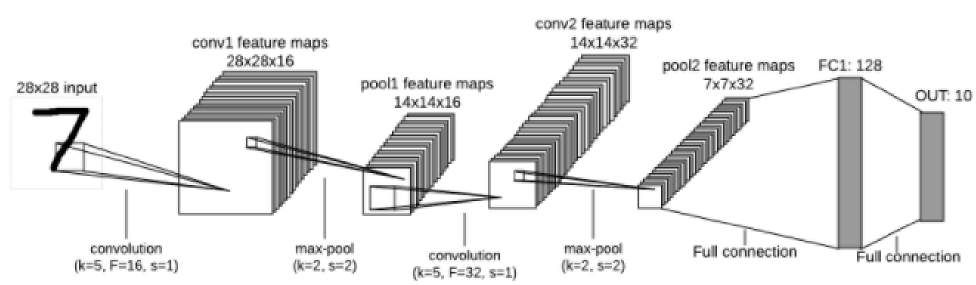
```
train_ds[0][0].shape
```

Out[8]:

```
torch.Size([1, 28, 28])
```

Define the model

- Convolutional Neural Network (CNN)



The model architecture that we are going to build

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

In [9]:

```
class MyCNN(nn.Module):                                  # define own MyCNN which
  def __init__(self):
    super(MyCNN, self).__init__()
    self.conv1 = nn.Conv2d(1, 16, kernel_size=3)         # conv2d(in_channel, out
    # (28-3+2*0)/1 + 1 = 26                              # output size = (W-K+2P)
    self.maxpool = nn.MaxPool2d(2)
    # (26/2)
    self.fc1 = nn.Linear(13*13*16, 10)                   # Flattened output from

  def forward(self, x):
    x = F.relu(self.conv1(x))
    x = self.maxpool(x)
    x = torch.flatten(x,1)                               # feature maps flattened
    x = self.fc1(x)                                      # produces an output of
```

```
        return x, F.log_softmax(x, dim=1)          # raw output from x (log
```

In [10]:

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

In [11]:

```
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_mo
de=False)
  (fc1): Linear(in_features=2704, out_features=10, bias=True)
)
```

In [12]:

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

def train(train_loader=train_loader):
    cnn_model.train()                                     # sets model to t
    train_corr, train_total, train_running_loss = 0, 0, 0  # counters for tr

    for step, (data, y) in enumerate(train_loader):       # Loops over batc
        data, y = data.to(device), y.to(device)
        optimizer.zero_grad()                               # resets gradie
        _, logits = cnn_model(data)                         # gets the logi
        loss = loss_fn(logits, y)                           # calculates lo
        loss.backward()                                     # back propagat
        optimizer.step()                                    # optimizer upd

        y_pred = torch.argmax(logits, 1)                    # selects the p
        train_corr += torch.sum(torch.eq(y_pred, y).float()).item()      # c
        train_total += len(data)                            # tracks total
        train_running_loss += loss.item()                   # accumulates l

    # 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

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

def test(test_loader=test_loader):
    cnn_model.eval()                                        # sets model to
    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}] Valid/Test Loss: {epoch_loss:.4f}, Accuracy: {
```

In [13]:

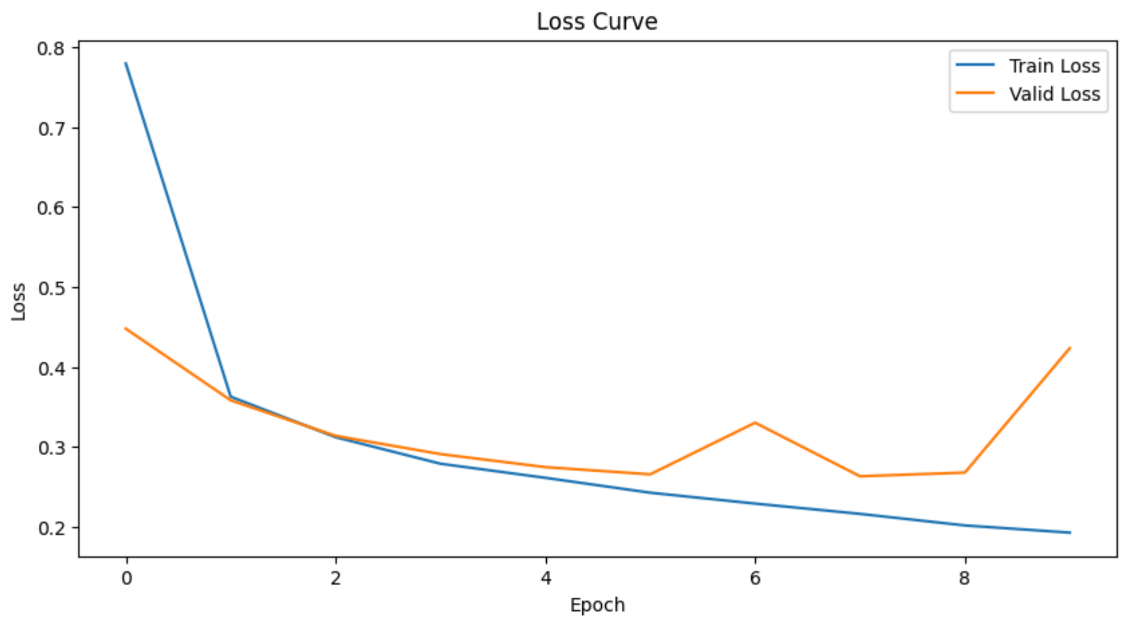
```
for epoch in range(num_epoch):
  print(f"--------- Train EPOCH {epoch} -------------")
  train(train_loader)
  print(f"--------- Valid EPOCH {epoch} -------------")
  test(valid_loader)
```

```
--------- Train EPOCH 0 -------------
Epoch [1] Train Loss: 0.7796, Accuracy: 0.8075
--------- Valid EPOCH 0 -------------
Epoch [1] Valid/Test Loss: 0.4480, Accuracy: 0.8700
--------- Train EPOCH 1 -------------
Epoch [2] Train Loss: 0.3630, Accuracy: 0.8994
--------- Valid EPOCH 1 -------------
Epoch [2] Valid/Test Loss: 0.3585, Accuracy: 0.8885
--------- Train EPOCH 2 -------------
Epoch [3] Train Loss: 0.3124, Accuracy: 0.9113
--------- Valid EPOCH 2 -------------
Epoch [3] Valid/Test Loss: 0.3140, Accuracy: 0.9090
--------- Train EPOCH 3 -------------
Epoch [4] Train Loss: 0.2792, Accuracy: 0.9198
--------- Valid EPOCH 3 -------------
Epoch [4] Valid/Test Loss: 0.2912, Accuracy: 0.9140
--------- Train EPOCH 4 -------------
Epoch [5] Train Loss: 0.2615, Accuracy: 0.9257
--------- Valid EPOCH 4 -------------
Epoch [5] Valid/Test Loss: 0.2749, Accuracy: 0.9170
--------- Train EPOCH 5 -------------
Epoch [6] Train Loss: 0.2429, Accuracy: 0.9307
--------- Valid EPOCH 5 -------------
Epoch [6] Valid/Test Loss: 0.2659, Accuracy: 0.9225
--------- Train EPOCH 6 -------------
Epoch [7] Train Loss: 0.2293, Accuracy: 0.9338
--------- Valid EPOCH 6 -------------
Epoch [7] Valid/Test Loss: 0.3305, Accuracy: 0.8985
--------- Train EPOCH 7 -------------
Epoch [8] Train Loss: 0.2165, Accuracy: 0.9371
--------- Valid EPOCH 7 -------------
Epoch [8] Valid/Test Loss: 0.2635, Accuracy: 0.9185
--------- Train EPOCH 8 -------------
Epoch [9] Train Loss: 0.2020, Accuracy: 0.9431
```

```
--------- Valid EPOCH 8 -------------
Epoch [9] Valid/Test Loss: 0.2681, Accuracy: 0.9215
--------- Train EPOCH 9 -------------
Epoch [10] Train Loss: 0.1930, Accuracy: 0.9456
--------- Valid EPOCH 9 -------------
Epoch [10] Valid/Test Loss: 0.4236, Accuracy: 0.8660
```

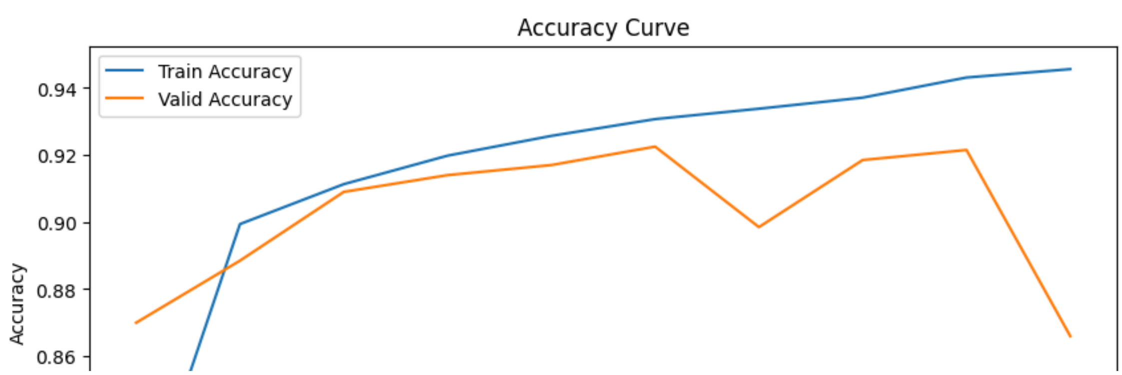
In [14]:

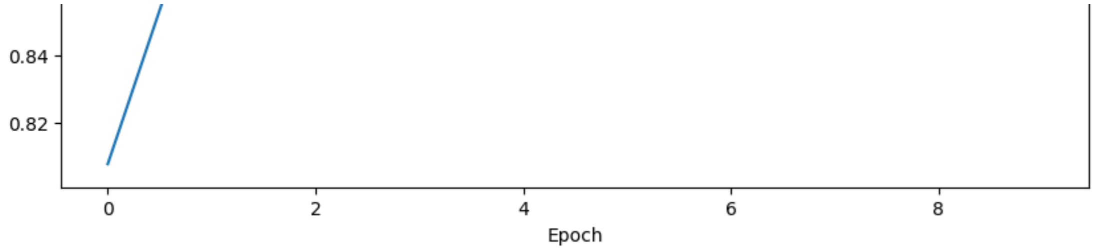
```python
# Plot the training and test loss
plt.figure(figsize=(10, 5))
plt.plot(train_losses, label='Train Loss')
plt.plot(test_losses, label='Valid Loss')
plt.title('Loss Curve')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.show()
```



In [15]:

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

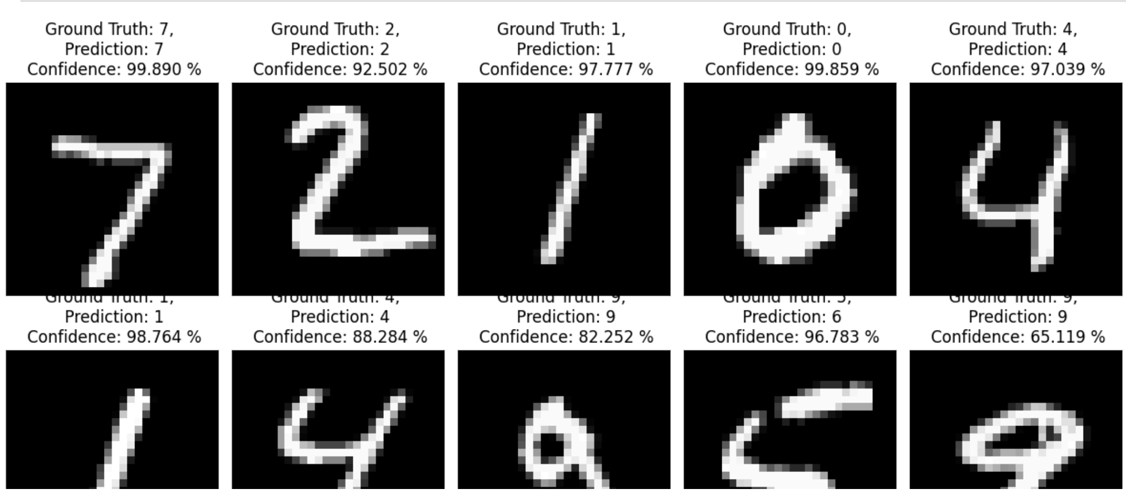




Plot some prediction

In [16]:

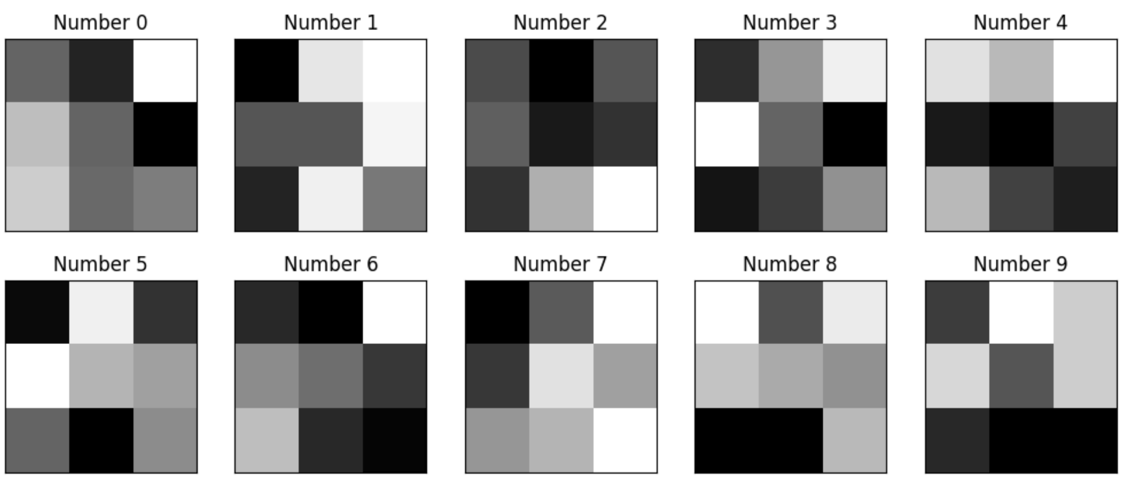
```
cnn_model.eval()
data, y = next(iter(test_loader))

# 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 ; unn

# 3. get the class with highest prob.
y_pred = torch.argmax(logits, 1) # finds the index of the class with the hig

get_prob = torch.nn.Softmax(dim=1) # converts logits into probabilities that
prob = get_prob(logits)            # prob is a tensor where each row corresp

# 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 gradie
    # cpu(): Moves the tensor back to the CPU (important if you're using a G
    # numpy(): Converts the tensor to a NumPy array.

    plt.title(f"Ground Truth: {y[i].cpu().detach().numpy()}, \n  Prediction:
    plt.xticks([])
    plt.yticks([])
plt.tight_layout() # Adjusts the subplot parameters to make sure that subplo
plt.show()
```



Visualize filter weights

In [17]:

```
fig = plt.figure(figsize=[5*2.5, 2*2.5])
for i in range(10):                              # Loops through first 10 filter
  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() # Ch
  ax.imshow(ws, cmap='gray')
  plt.title(f"Number {i}")
  plt.xticks([])
  plt.yticks([])
```

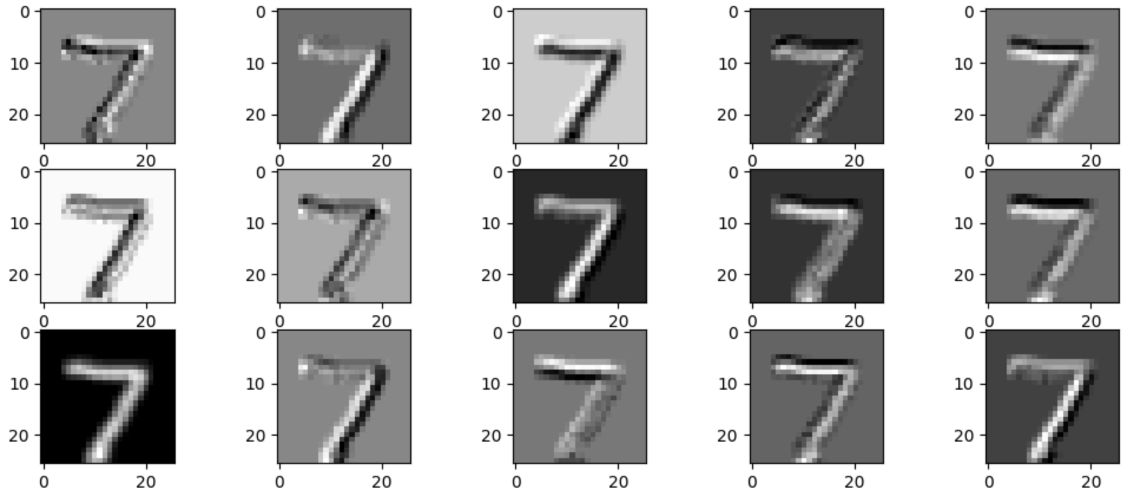


Visualize feature map

In [18]:

```
# Visualize feature maps
activation = {}          # initializes empty dictionary to store the feature

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

cnn_model.conv1.register_forward_hook(get_activation('conv1')) # This line r
data, _ = test_ds[0]                                            # retrieves a
data.unsqueeze_(0)      # Since this is a single image, unsqueeze_(0) changes
output = cnn_model(data.to(device))  # output is not required for this case

fm_cov1 = activation['conv1'].squeeze().cpu().detach().numpy()  # .squeeze()
fig = plt.figure(figsize=[5*2.5, 2*2.5])
for i in range(15):
  ax = fig.add_subplot(3, 5, i+1)
  ax.imshow(fm_cov1[i], cmap='gray')
```

```
# The feature maps are the result of applying the learned filters to the inp
```



Take HOME RESNET18 model

- Load a pretrained model on Imagenet dataset.

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.

In [19]:

```
# Load ResNet-18 pre-trained model
from torchvision.models import ResNet18_Weights
resnet18 = models.resnet18(weights=ResNet18_Weights.IMAGENET1K_V1) # or use
```

```
Downloading: "https://download.pytorch.org/models/resnet18-f37072fd.pth" to /h
ome/jupyter-dsai-st123439/.cache/torch/hub/checkpoints/resnet18-f37072fd.pth
100%|██████████| 44.7M/44.7M [00:00<00:00, 89.7MB/s]
```

In [20]:

```
# Modify the final layer to match the number of classes (for example, CIFAR-
num_classes = 10
resnet18.fc = nn.Linear(resnet18.fc.in_features, num_classes)
```

In [21]:

```
# 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, 0.22
])
```

In [22]:

```
# Download CIFAR-10 dataset (or use your own dataset)
train_dataset = datasets.CIFAR10(root='./data', train=True, transform=transf
test_dataset = datasets.CIFAR10(root='./data', train=False, transform=transf


# 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 *

# Load the data
train_loader = torch.utils.data.DataLoader(dataset=train_subset, batch_size=
test_loader = torch.utils.data.DataLoader(dataset=test_subset, batch_size=64
```

In [23]:

```
print(resnet18)
```

```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bi
as=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_s
tats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mo
de=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_runni
ng_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_runni
ng_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_runni
ng_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_runni
ng_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_runn
ing_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_runn
ing_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_runn
ing_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_runn
ing_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_runn
ing_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_runn
ing_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_runn
ing_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_runn
ing_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_runn
ing_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_runn
ing_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_runn
ing_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_runn
ing_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_runn
ing_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_runn
ing_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_runn
ing_stats=True)
    )
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Linear(in_features=512, out_features=10, bias=True)
)
```

In [24]:

```python
# 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(trai
```

In [25]:

```
# Re-write train_model to implement Validation Loop and finally use test_mod

# Your code here
```

In [26]:

```
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
```

In [27]:

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

```
Epoch [1/5], Loss: 1.3716
Epoch [2/5], Loss: 0.5920
Epoch [3/5], Loss: 0.2437
Epoch [4/5], Loss: 0.1424
Epoch [5/5], Loss: 0.1066
```

In [28]:

```
# Testing the trained model
test_model(resnet18, test_loader)
```

```
Accuracy of the model on the test images: 61.50%
```

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.