<u>Submission instructions</u>

Submission in pairs unless otherwise authorized

- This notebook contains all the questions. You should follow the instructions below.
- Solutions for both theoretical and practical parts should be written in this notebook

Moodle submission

You should submit three files:

- IPYNB notebook:
  - All the wet and dry parts, including code, graphs, discussion, etc.
- PDF file:
  - Export the notebook to PDF. Make sure that all the cells are visible.
- Pickle files:
  - As requested in Q2.a and Q3.a
- PY file:
  - As requested in Q3.a

All files should be in the following format: "HW1_ID1_ID2.file"
Good Luck!

Question 1

I. Softmax Derivative (10pt)

Derive the gradients of the softmax function and demonstrate how the expression can be reformulated solely by using the softmax function, i.e., in some expression where only $softmax(x)$, but not x, is present). Recall that the softmax function is defined as follows:

$$softmax(x)_i = \frac{e^{x_i}}{\sum_{j=1}^{N} e^{x_j}}$$

I. Softmax Derivative - Answer:

I)

$$\frac{\partial softmax(x_i)}{\partial x_k} = \frac{\partial}{\partial x_k}\left(\frac{e^{x_i}}{\sum_{j=1}^{N} e^{x_j}}\right)$$

Case $i = k$:

Applying the quotient rule:

$$\frac{\partial softmax(x_i)}{\partial x_k} = \frac{e^{x_i} \cdot \frac{\partial}{\partial x_k}\left(\sum_{j=1}^{N} e^{x_j}\right) - \left(\frac{\partial e^{x_i}}{\partial x_k}\right) \cdot \sum_{j=1}^{N} e^{x_j}}{\left(\sum_{j=1}^{N} e^{x_j}\right)^2} = \frac{e^{x_i} \cdot \sum_{j=1}^{N} e^{x_j} - e^{x_i} \cdot e^{x_k}}{\left(\sum_{j=1}^{N} e^{x_j}\right)^2}$$

$$= softmax(x_i) \cdot (1 - softmax(x_i))$$

Case $i \neq k$:

Note that $\frac{\partial e^{x_i}}{\partial x_k} = 0$, hence:

$$\frac{\partial softmax(x_i)}{\partial x_k} = -\frac{e^{x_i} e^{x_k}}{\left(\sum_{j=1}^{N} e^{x_j}\right)^2} = -softmax(x_i) \cdot softmax(x_k)$$

II. Cross-Entropy Gradient (10pt)

Derive the gradient of cross-entropy loss with regard to the inputs of a softmax function. i.e., find the gradients with respect to the softmax input vector θ, when the prediction is denoted by $\hat{y} = softmax(\theta)$. Remember the cross entropy function is:

$$CE(y, \hat{y}) = -\sum_i y_i log(\hat{y}_i)$$

where y is the one-hot label vector, and $\hat{y}$ is the predicted probability vector for all classes.

II. Cross-Entropy Gradient - Answer

II)

$$\frac{\partial CE(y,\hat{y})}{\partial \theta_k} = \frac{\partial CE(y,\hat{y})}{\partial \hat{y}} \cdot \frac{\partial softmax(\theta_i)}{\partial \theta_k}$$

Note that $\hat{y} = softmax(\theta_i)$

Case $i = k$:

$$\frac{\partial CE(y,\hat{y})}{\partial \theta_k} = \frac{\partial CE(y,\hat{y})}{\partial \hat{y}} \cdot softmax(\theta_i) \cdot (1 - softmax(\theta_i))$$

Case $i \neq k$:

$$\frac{\partial CE(y,\hat{y})}{\partial \theta_k} = \frac{\partial CE(y,\hat{y})}{\partial \hat{y}} \cdot -softmax(\theta_i) \cdot softmax(\theta_k)$$

Question 2

I. Derivative Of Activation Functions (10pt)

The following cell contains an implementation of some activation functions. Implement the corresponding derivatives.

In [84]:

```
import torch

def sigmoid(x):
    return 1 / (1 + torch.exp(-x))

def tanh(x):
    return torch.div(torch.exp(x) - torch.exp(-x), torch.exp(x) +
torch.exp(-x))



def softmax(x):
    exp_x = torch.exp(x.T - torch.max(x, dim=-1).values).T  #
Subtracting max(x) for numerical stability
    return exp_x / exp_x.sum(dim=-1, keepdim=True)
```

In [86]:

```
def d_sigmoid(x):
    """
    Derivative of the sigmoid function.
```

```
    sigmoid'(x) = sigmoid(x) * (1 - sigmoid(x))
    """
    sig = sigmoid(x)
    return sig * (1 - sig)


def d_tanh(x):
    """
    Derivative of the tanh function.
    tanh'(x) = 1 - tanh^2(x)
    """
    tanh_x = tanh(x)
    return 1 - tanh_x ** 2

def d_softmax(x):
    """
    Derivative of the softmax function.
    This is the Jacobian matrix for softmax.
    """
    s = softmax(x)
    # Construct the diagonal matrix for the softmax
    s_diag = torch.diag_embed(s)
    # Outer product for the Jacobian
    s_outer = torch.matmul(s.unsqueeze(-1), s.unsqueeze(-2))
    return s_diag - s_outer
```

II. Train a Fully Connected network on MNIST (30pt)

In the following exercise, you will create a classifier for the MNIST dataset. You should write your own training and evaluation code and meet the following constraints:

- You are only allowed to use torch tensor manipulations.
- You are NOT allowed to use:
  - Auto-differentiation - backward()
  - Built-in loss functions
  - Built-in activations
  - Built-in optimization
  - Built-in layers (torch.nn)
       </h4>

a) The required classifier class is defined.

- You should implement the forward and backward passes of the model.
- Train the model and plot the model's accuracy and loss (both on train and test sets) as a function of the epochs.
- You should save the model's weights and biases. Change the student_ids to yours.

In this section, you **must** use the "set_seed" function with the given seed and **sigmoid** as an activation function.

In [88]:

```
import torch
import torchvision
from torch.utils.data import DataLoader

import os
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()

# Constants
SEED = 42
EPOCHS = 16
BATCH_SIZE = 32
NUM_OF_CLASSES = 10

# Setting seed
def set_seed(seed):
    torch.manual_seed(seed)
    torch.cuda.manual_seed(seed)
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False
    os.environ["PYTHONHASHSEED"] = str(seed)


# Transformation for the data
transform = torchvision.transforms.Compose(
    [torchvision.transforms.ToTensor(),
     torch.flatten])


# Cross-Entropy loss implementation
```

```
def one_hot(y, num_of_classes=10):
    hot = torch.zeros((y.size()[0], num_of_classes))
    hot[torch.arange(y.size()[0]), y] = 1
    return hot

def cross_entropy(y, y_hat):
    return -torch.sum(one_hot(y) * torch.log(y_hat)) / y.size()[0]
```

In [100…

```
# Create dataloaders
train_dataset = torchvision.datasets.MNIST(root='./data', train=True,
                                            download=True,
transform=transform)
train_dataloader = torch.utils.data.DataLoader(train_dataset,
batch_size=BATCH_SIZE)


test_dataset = torchvision.datasets.MNIST(root='./data', train=False,
                                           download=True,
transform=transform)
test_dataloader = torch.utils.data.DataLoader(test_dataset,
batch_size=BATCH_SIZE,)
```

In [102…

```
class FullyConnectedNetwork:
    def __init__(self, input_size, output_size, hidden_size1,
activiation_func, lr=0.01):
        # parameters
        self.input_size = input_size
        self.output_size = output_size
        self.hidden_size1 = hidden_size1

        # activation function
        self.activation_func = activiation_func

        # weights
        self.W1 = torch.randn(self.input_size, self.hidden_size1)
        self.b1 = torch.zeros(self.hidden_size1)

        self.W2 = torch.randn(self.hidden_size1, self.output_size)
        self.b2 = torch.zeros(self.output_size)

        self.lr = lr
```

```
        # For storing intermediate values and gradients
        self.z1 = None
        self.a1 = None
        self.dL_dW1 = None
        self.dL_db1 = None
        self.dL_dW2 = None
        self.dL_db2 = None

    def forward(self, x):
        # First layer (hidden layer)
        self.z1 = torch.matmul(x, self.W1) + self.b1
        self.a1 = self.activation_func(self.z1)  # Apply the activation
function and store a1

        # Second layer (output layer)
        z2 = torch.matmul(self.a1, self.W2) + self.b2
        y_hat = sigmoid(z2)  # Apply sigmoid for output
        return y_hat

    def backward(self, x, y, y_hat):
        # Gradient for the output layer (backpropagation)
        loss_grad = y_hat - one_hot(y)  # Assuming softmax + cross-
entropy
        self.dL_dW2 = torch.matmul(self.a1.T, loss_grad)
        self.dL_db2 = loss_grad.sum(dim=0)

        # Gradient for the hidden layer (backpropagation)
        d_a1 = torch.matmul(loss_grad, self.W2.T)  # Backprop to hidden
layer
        d_z1 = d_a1 * self.activation_func(self.z1) * (1 -
self.activation_func(self.z1))  # Derivative of sigmoid

        # Gradient for the weights and biases of the first layer
        self.dL_dW1 = torch.matmul(x.T, d_z1)
        self.dL_db1 = d_z1.sum(dim=0)
```

In [128…

```
students_ids = "321817405_208912675"
model345 = FullyConnectedNetwork(784, 10,128, torch.sigmoid)
##torch.save({"W1": model.W1, "W2": model.W2, "b1": model.b1, "b2":
model.b2}, f"HW1_Q2_{students_ids}.pkl")
```

b) Train the model with various learning rates (at least 3).

- Plot the model's accuracy and loss (both on train and test sets) as a function of the epochs.
- Discuss the differences in training with different learning rates. Support your answer with plots.

In [126…

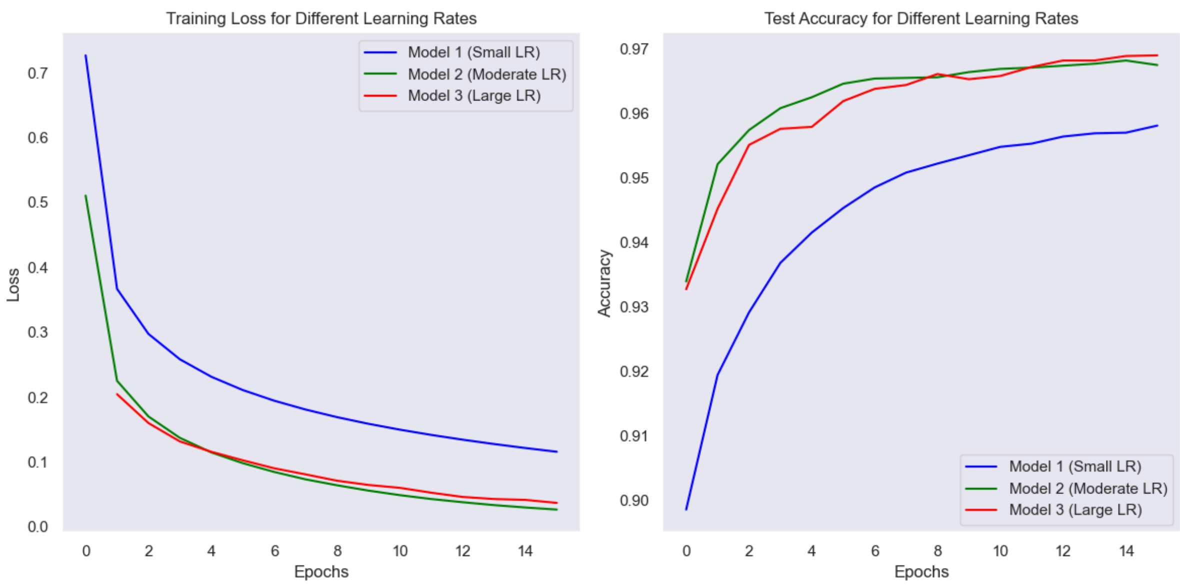
```
set_seed(SEED)
model1 = FullyConnectedNetwork(784, 10, 128, sigmoid, lr=0.01)
model2 = FullyConnectedNetwork(784, 10, 128, sigmoid, lr=0.05)
model3 = FullyConnectedNetwork(784, 10, 128, sigmoid, lr=0.1)



# Update parameters function (outside the class)
def update_parameters(model):
    model.W1 -= model.lr * model.dL_dW1
    model.b1 -= model.lr * model.dL_db1
    model.W2 -= model.lr * model.dL_dW2
    model.b2 -= model.lr * model.dL_db2

def evaluate(model, dataloader):
    correct = 0
    total = 0
    loss = 0
    with torch.no_grad():
        for data in dataloader:
            inputs, labels = data
            outputs = model.forward(inputs)
            _, predicted = torch.max(outputs, 1)
            loss += cross_entropy(labels, outputs)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
    return correct / total, loss / total

# Training function (outside the class)
def train(model, train_dataloader, test_dataloader):
    # Initialize lists to store training losses and test accuracies
    train_losses = []
    test_accuracies = []
```

```
    for epoch in range(EPOCHS):
        running_loss = 0.0
        for i, data in enumerate(train_dataloader, 0):
            inputs, labels = data
            y_hat = model.forward(inputs)  # Forward pass
            model.backward(inputs, labels, y_hat)  # Backward pass (using stored a1 and z1)
            update_parameters(model)  # Make sure this function is defined elsewhere

            loss = cross_entropy(labels, y_hat)
            running_loss += loss

        accuracy, _ = evaluate(model, test_dataloader)
        train_losses.append(running_loss / len(train_dataloader))
        test_accuracies.append(accuracy)

    # Return the lists containing the metrics for plotting
    return train_losses, test_accuracies
```

In [9]:

```
# Assume train() is called for each model and returns losses and accuracies
train_losses1, test_accuracies1 = train(model1, train_dataloader, test_dataloader)
train_losses2, test_accuracies2 = train(model2, train_dataloader, test_dataloader)
train_losses3, test_accuracies3 = train(model3, train_dataloader, test_dataloader)

# Plot accuracy and loss for all models
plt.figure(figsize=(12, 6))

# Plot training losses
plt.subplot(1, 2, 1)
plt.plot(train_losses1, label='Model 1 (lr = 0.01)', color='blue')
plt.plot(train_losses2, label='Model 2 (lr = 0.05)', color='green')
plt.plot(train_losses3, label='Model 3 (lr = 0.1)', color='red')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Training Loss')
plt.legend()
```

```
plt.grid()

# Plot test accuracies
plt.subplot(1, 2, 2)
plt.plot(test_accuracies1, label='Model 1 (lr = 0.01)', color='blue')
plt.plot(test_accuracies2, label='Model 2 (lr = 0.05)', color='green')
plt.plot(test_accuracies3, label='Model 3 (lr = 0.1)', color='red')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.title('Test Accuracy')
plt.legend()
plt.grid()

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



Small learning rates (blue) lead to slow convergence and lower final accuracy due to minimal weight updates. Moderate learning rates (green) strike the best balance, achieving rapid loss reduction and the highest accuracy with stability. Large learning rates (red) converge quickly but may cause fluctuations and risk instability if increased further. Overall, the moderate learning rate provides the best performance and generalization.

Question 3

I. Implement and Train a CNN (30pt)

As you might know, there are many dogs on campus. Sometimes, understanding the emotions of a dog can be challenging, and people might mistakenly try to pet it when it is sad or angry. As a data scientist, you have

been asked to assist Technion's students. Your task is to create a "dog emotion classifier.
Your code should meet the following constraints:

- Your classifier must be CNN based
- You are not allowed to use any pre-trained model

To satisfy your boss, your model must achieve at least 70% accuracy on the test set. Your boss also emphasized that the model will be deployed on smartphones, so it should have a small number of parameters. 25% of your grade for this task will be based on the number of parameters your model uses — fewer parameters will yield a higher grade.

Stages

1. Perform a short EDA (Exploratory Data Analysis).
2. Train the model and plot its accuracy and loss (for both the training and validation sets) as a function of the epochs.
3. Report the test set accuracy.
4. Discuss the progress you made and describe your final model.

   Your data is in `hw1_data/dog_emotion`.
   You can define a custom dataset (as in tutorial 3) or use `torchvision.datasets.ImageFolder`.

   ## Submission

   In addition to the code in the notebook, you should submit:

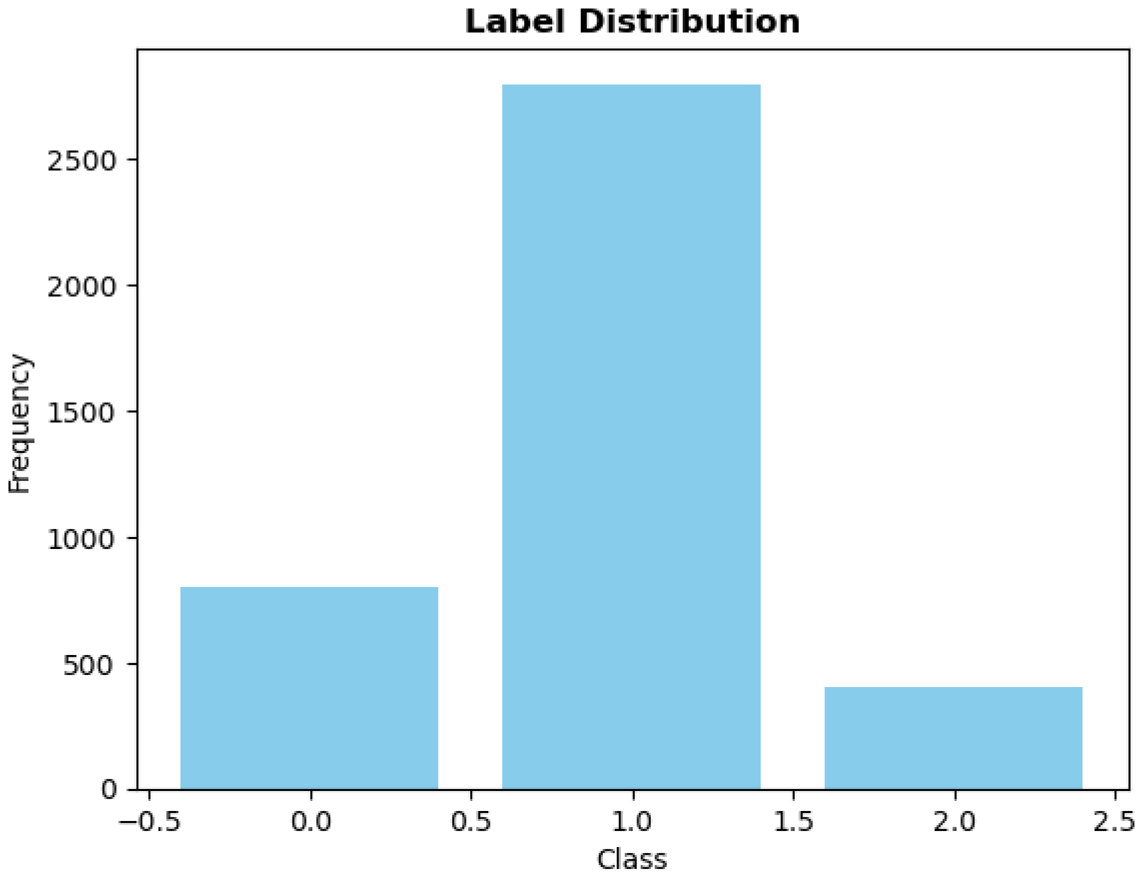
   - a `.py` file containing your model class.
   - a `.pkl` file containing the weight of your model

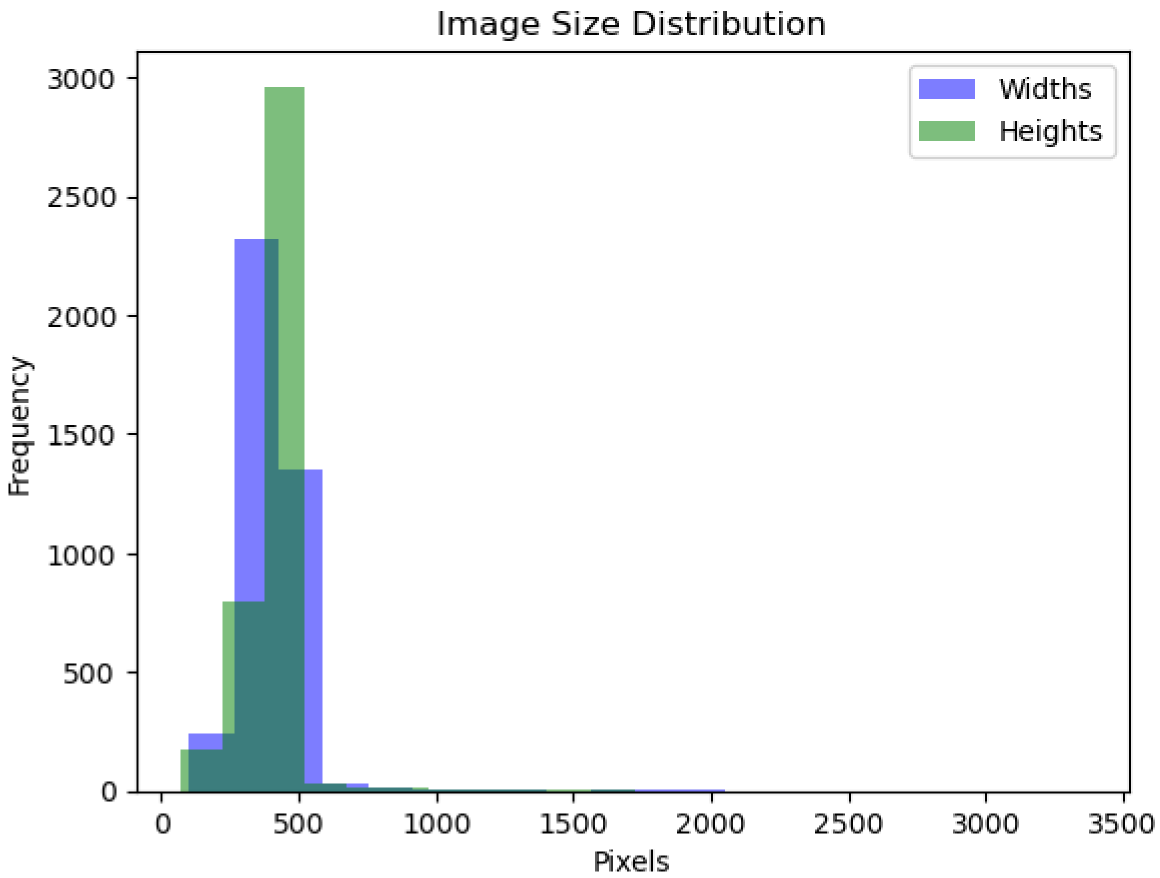
1. EDA:

In [47]:
```
import numpy as np
import matplotlib.pyplot as plt
from torchvision import datasets

# EDA Function
def perform_eda(dataset, train_loader):
```

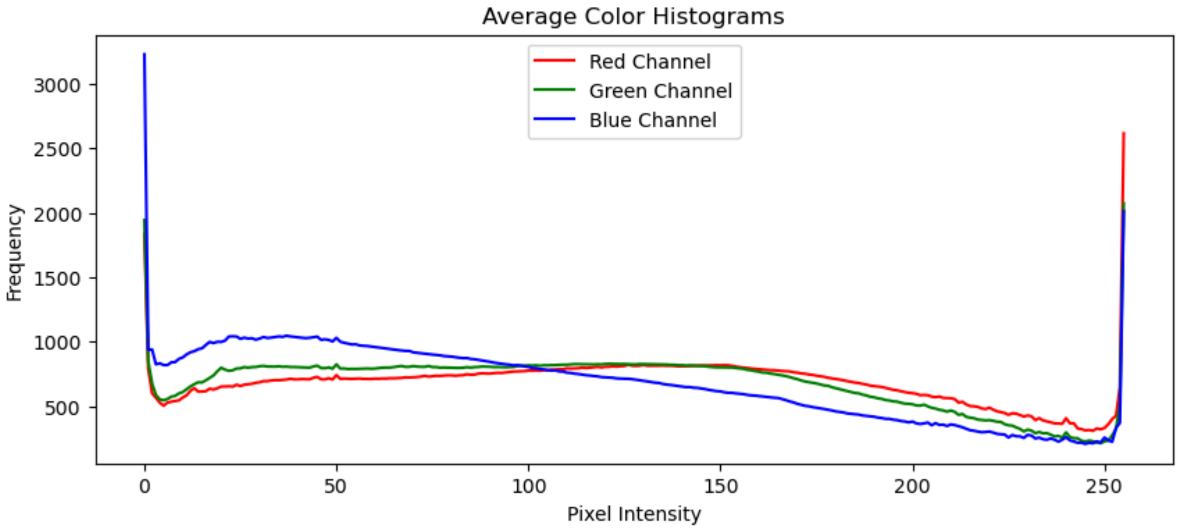
```
    # 1. Label Distribution
    def analyze_label_distribution(dataset):
        labels = [label for _, label in dataset.samples]
        class_counts = np.bincount(labels)
        plt.bar(range(len(class_counts)), class_counts, color='skyblue')
        plt.title("Label Distribution", fontweight="bold")
        plt.xlabel("Class")
        plt.ylabel("Frequency")
        plt.show()

    # 2. Image Size Analysis
    def analyze_image_sizes(dataset):
        sizes = [img.size for img, _ in dataset]
        widths, heights = zip(*sizes)
        plt.hist(widths, bins=20, alpha=0.5, label='Widths',
color='blue')
        plt.hist(heights, bins=20, alpha=0.5, label='Heights',
color='green')
        plt.legend()
        plt.title("Image Size Distribution")
        plt.xlabel("Pixels")
        plt.ylabel("Frequency")
        plt.show()
        print(f"Average Image Size: ({np.mean(widths):.2f},
{np.mean(heights):.2f})")

    # 3. Color Histograms
    def analyze_color_histograms(dataset):
        total_pixels = np.zeros((3, 256))  # For storing sums of pixel
intensities for each channel
        total_intensity = np.zeros(3)  # For storing total intensity per
channel
        num_images = 0

        for img, _ in dataset:
            img_array = np.array(img)

            if img_array.ndim == 3:  # Ensure image has color channels
                num_images += 1
                for channel in range(3):
                    hist, _ = np.histogram(img_array[:, :,
```

```
channel].ravel(), bins=256, range=(0, 256))
                    total_pixels[channel] += hist
                    total_intensity[channel] += img_array[:, :,
channel].sum()

        # Compute average histogram
        avg_pixels = total_pixels / num_images

        # Compute average color intensity
        avg_intensity = total_intensity / (num_images *
img_array.shape[0] * img_array.shape[1])

        # Plot averaged histograms
        plt.figure(figsize=(10, 4))
        for channel, color in enumerate(['red', 'green', 'blue']):
            plt.plot(avg_pixels[channel], color=color, label=f"
{color.capitalize()} Channel")
        plt.title("Average Color Histograms")
        plt.xlabel("Pixel Intensity")
        plt.ylabel("Frequency")
        plt.legend()
        plt.show()

        # Print average color intensity
        for channel, color in enumerate(['Red', 'Green', 'Blue']):
            print(f"Average {color} Intensity:
{avg_intensity[channel]:.4f}")

    # Execute EDA steps
    analyze_label_distribution(dataset)
    analyze_image_sizes(dataset)
    analyze_color_histograms(dataset)

# Integrating EDA
train_loader, val_loader, test_loader = load_data(batch_size)
num_classes = 4
perform_eda(datasets.ImageFolder("hw1_data/Dog_Emotion"), train_loader)
```






```
Average Image Size: (417.82, 388.91)
```




```
Average Red Intensity: 143.2194
Average Green Intensity: 133.6950
Average Blue Intensity: 117.0914
```

2. Model Training:

In [34]:

```
import torch
import torch.nn as nn
import torchvision.datasets as datasets
import torchvision.transforms as transforms
from sklearn.metrics import accuracy_score
import matplotlib.pyplot as plt

# Function to load the data and apply necessary transformations
def load_data(batch_size):
    transform_train = transforms.Compose([
        transforms.Resize((128, 128)),
        transforms.RandomHorizontalFlip(p=0.5),
        transforms.RandomRotation(15),
        transforms.ColorJitter(brightness=0.2, contrast=0.2,
saturation=0.2, hue=0.1),
        transforms.GaussianBlur(kernel_size=3, sigma=(0.1, 2.0)),
        transforms.ToTensor(),
        transforms.Normalize(mean=(0.5, 0.5, 0.5), std=(0.5, 0.5, 0.5))
    ])

    transform_test = transforms.Compose([
        transforms.Resize((128, 128)),
        transforms.ToTensor(),
        transforms.Normalize(mean=(0.5, 0.5, 0.5), std=(0.5, 0.5, 0.5))
    ])
```

```
    train_dataset =
datasets.ImageFolder(root='hw1_data/dog_emotion/train',
transform=transform_train)
    val_dataset = datasets.ImageFolder(root='hw1_data/dog_emotion/val',
transform=transform_test)
    test_dataset =
datasets.ImageFolder(root='hw1_data/dog_emotion/test',
transform=transform_test)

    train_loader = torch.utils.data.DataLoader(train_dataset,
batch_size=batch_size, shuffle=True)
    val_loader = torch.utils.data.DataLoader(val_dataset,
batch_size=batch_size, shuffle=False)
    test_loader = torch.utils.data.DataLoader(test_dataset,
batch_size=batch_size, shuffle=False)

    return train_loader, val_loader, test_loader

# CNN Model
class DogEmotionCNN(nn.Module):
    def __init__(self):
        super(DogEmotionCNN, self).__init__()
        self.conv1 = nn.Sequential(
            nn.Conv2d(3, 16, kernel_size=1, padding=1),
            nn.BatchNorm2d(16),
            nn.ReLU(),
            nn.MaxPool2d(2)
        )
        self.conv2 = nn.Sequential(
            nn.Conv2d(16, 36, kernel_size=2, padding=1),
            nn.BatchNorm2d(36),
            nn.ReLU(),
        )
        self.conv3 = nn.Sequential(
            nn.Conv2d(36, 72, kernel_size=6, padding=3),
            nn.BatchNorm2d(72),
            nn.ReLU(),
        )
        self.conv4 = nn.Sequential(
            nn.Conv2d(72, 36, kernel_size=6, padding=3),
            nn.BatchNorm2d(36),
```

```
            nn.ReLU(),
        )
        self.conv5 = nn.Sequential(
            nn.Conv2d(36, 72, kernel_size=6, padding=3),
            nn.BatchNorm2d(72),
            nn.ReLU(),
        )
        self.conv6 = nn.Sequential(
            nn.Conv2d(72, 36, kernel_size=6, padding=3),
            nn.BatchNorm2d(36),
            nn.ReLU(),
            nn.MaxPool2d(2)
        )
        self.conv7 = nn.Sequential(
            nn.Conv2d(36, 72, kernel_size=6, padding=3),
            nn.BatchNorm2d(72),
            nn.ReLU(),
            nn.MaxPool2d(2)
        )
        self.conv8 = nn.Sequential(
            nn.Conv2d(72, 36, kernel_size=6, padding=3),
            nn.BatchNorm2d(36),
            nn.ReLU(),
            nn.MaxPool2d(2)
        )

        dummy_input = torch.zeros(1, 3, 128, 128)
        dummy_output = self.conv8(self.conv7(self.conv6(self.conv5(

self.conv4(self.conv3(self.conv2(self.conv1(dummy_input))))))))
        flattened_size = dummy_output.view(1, -1).size(1)

        self.fc1 = nn.Linear(flattened_size, 128)
        self.fc2 = nn.Linear(128, 4)
        self.dropout = nn.Dropout(p=0.5)
        self.logsoftmax = nn.LogSoftmax(dim=1)

    def forward(self, x):
        out = self.conv1(x)
        out = self.conv2(out)
        out = self.conv3(out)
        out = self.conv4(out)
```

```
        out = self.conv5(out)
        out = self.conv6(out)
        out = self.conv7(out)
        out = self.conv8(out)
        out = out.view(out.size(0), -1)
        out = self.dropout(out)
        out = self.fc1(out)
        out = self.fc2(out)
        return self.logsoftmax(out)

# Training Function
def train_model_with_scheduler(model, data_loaders, optimizer,
scheduler, num_epochs, criterion):
    errors = {'train': [], 'val': []}
    losses = {'train': [], 'val': []}

    for epoch in range(num_epochs):

        for phase in ['train', 'val']:
            if phase == 'train':
                model.train()
            else:
                model.eval()

            labels_list, preds_list = [], []
            running_loss = 0.0
            counter_batches = 0

            for data_images, data_labels in data_loaders[phase]:
                counter_batches += 1
                images, labels = data_images, data_labels

                if phase == 'train':
                    optimizer.zero_grad()
                    outputs = model(images)
                    loss = criterion(outputs, labels)
                    loss.backward()
                    optimizer.step()
                else:
                    with torch.no_grad():
                        outputs = model(images)
                        loss = criterion(outputs, labels)
```

```python
                running_loss += loss.item()
                _, predicted = torch.max(outputs, 1)
                labels_list += labels.tolist()
                preds_list += predicted.tolist()

            epoch_acc = accuracy_score(labels_list, preds_list)
            errors[phase].append(1 - epoch_acc)
            losses[phase].append(running_loss / counter_batches)

        scheduler.step()

    return losses, errors

# Plot Function
def plot_metrics(losses, errors, num_epochs):
    epochs = range(1, num_epochs + 1)

    plt.figure(figsize=(12, 5))
    plt.subplot(1, 2, 1)
    plt.plot(epochs, losses['train'], label='Train Loss')
    plt.plot(epochs, losses['val'], label='Validation Loss')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.title('Training and Validation Loss')
    plt.legend()

    plt.subplot(1, 2, 2)
    train_accuracy = [1 - e for e in errors['train']]
    val_accuracy = [1 - e for e in errors['val']]
    plt.plot(epochs, train_accuracy, label='Train Accuracy')
    plt.plot(epochs, val_accuracy, label='Validation Accuracy')
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy')
    plt.title('Training and Validation Accuracy')
    plt.legend()

    plt.tight_layout()
    plt.show()

# Main Execution
if __name__ == "__main__":
```

```
    # Set parameters
    batch_size = 128
    num_epochs = 150
    learning_rate = 0.001

    # Load data
    train_loader, val_loader, test_loader =
load_data(batch_size=batch_size)

    # Initialize model
    model = DogEmotionCNN()

    # Define optimizer, scheduler, and loss
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate,
weight_decay=1e-4)
    scheduler =
torch.optim.lr_scheduler.CosineAnnealingWarmRestarts(optimizer, T_0=10,
T_mult=2)
    criterion = nn.CrossEntropyLoss()

    # Train the model
    losses, errors = train_model_with_scheduler(model, {'train':
train_loader, 'val': val_loader}, optimizer, scheduler, num_epochs,
criterion)

    # Plot metrics
    plot_metrics(losses, errors, num_epochs)
```

In [37]:

```
# Function to evaluate the model on the test set
def evaluate_model(model, test_loader, criterion):
    model.eval()  # Set the model to evaluation mode
```

```
    labels_list, preds_list = [], []
    running_loss = 0.0
    counter_batches = 0

    with torch.no_grad():  # Disable gradient computation
        for data_images, data_labels in test_loader:
            counter_batches += 1
            images, labels = data_images, data_labels

            outputs = model(images)
            loss = criterion(outputs, labels)
            running_loss += loss.item()

            _, predicted = torch.max(outputs, 1)
            labels_list += labels.tolist()
            preds_list += predicted.tolist()

    # Calculate accuracy
    test_acc = accuracy_score(labels_list, preds_list)
    test_loss = running_loss / counter_batches

    print(f'Test Loss: {test_loss:.4f}')
    print(f'Test Accuracy: {test_acc:.4f}')

    return test_loss, test_acc

# Add this to your code and then call the function
evaluate_model(model, test_loader, criterion)
```

```
Test Loss: 0.9301
Test Accuracy: 0.6963
```

Out[37]:

```
(0.9301320825304303, 0.69625)
```

In [ ]:

```
students_ids = "321817405_208912675"
#model = DogEmotionCNN()
torch.save(model.state_dict(), f'HW1_Q3_{students_ids}_state_dict.pkl')
```

4. Discussion

The training loss steadily decreases, showing effective learning, while the validation loss generally declines with some fluctuations, stabilizing around epoch 60. Training accuracy consistently improves, and validation accuracy follows a similar trend, though slightly lower and more variable. Both metrics plateau toward the end, with minimal overfitting evident as the validation performance closely matches the training performance. This indicates good generalization of the model.

II. Analyzing a Pre-trained CNN (Filters) (10pt)

In this part, you are going to analyze a (large) pre-trained model. Pre-trained models are quite popular these days, as big companies can train really large models on large datasets (something that personal users can't do as they lack the sufficient hardware). These pre-trained models can be used to fine-tune on other/small datasets or used as components in other tasks (like using a pre-trained classifier for object detection).

All pre-trained models expect input images normalized in the same way, i.e. mini-batches of 3-channel RGB images of shape (3 x H x W), where H and W are expected to be at least 224. The images have to be loaded in to a range of [0, 1] and then normalized using mean = [0.485, 0.456, 0.406] and std = [0.229, 0.224, 0.225].

You can use the following transform to normalize:

`normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])` Read more here

1. Load a pre-trained VGG16 with PyTorch using torchvision.models.vgg16(pretrained=True, progress=True, **kwargs) (read more here). Don't forget to use the model in evaluation mode (`model.eval()`).

2. Load the images in the `hw1_data/birds` folder and display them.

3. Pre-process the images to fit VGG16's architecture. What steps did you take?

4. Feed the images (forward pass) to the model. What are the outputs?

5. Choose an image of a dog in the `hw1_data/dogs` folder, display it and feed it to network. What are the outputs?

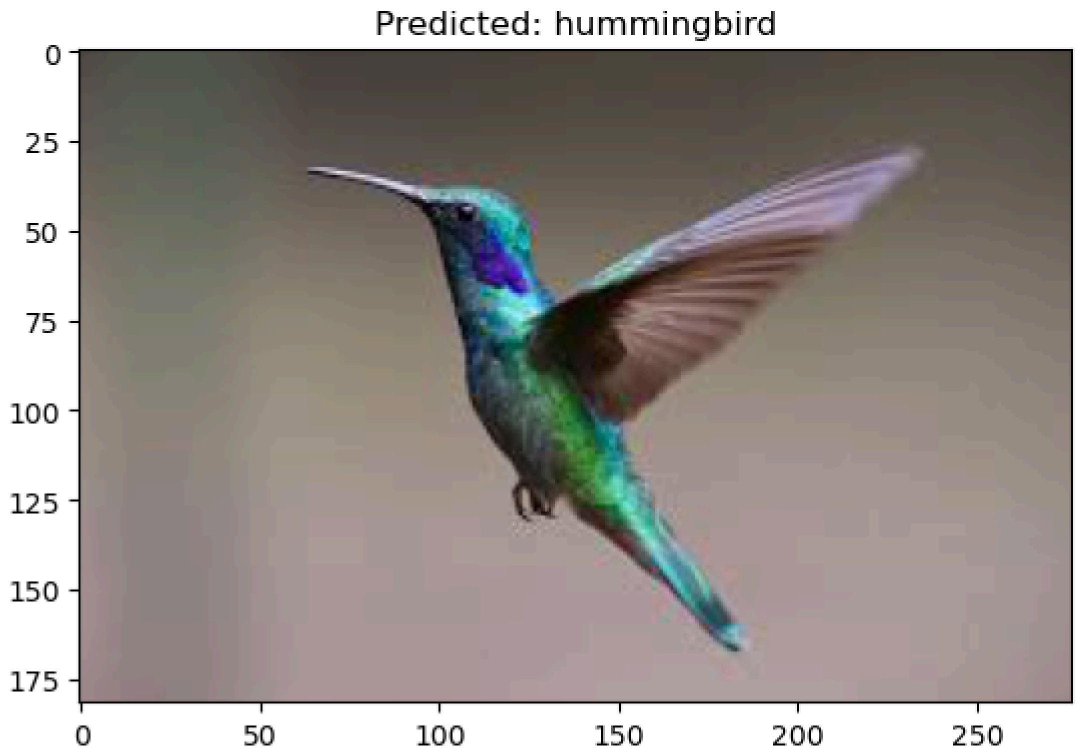
6. For the first 3 filters in the first layer of VGG16, plot their response (their output) for the image from section 5. Explain what do you see.

In [151…

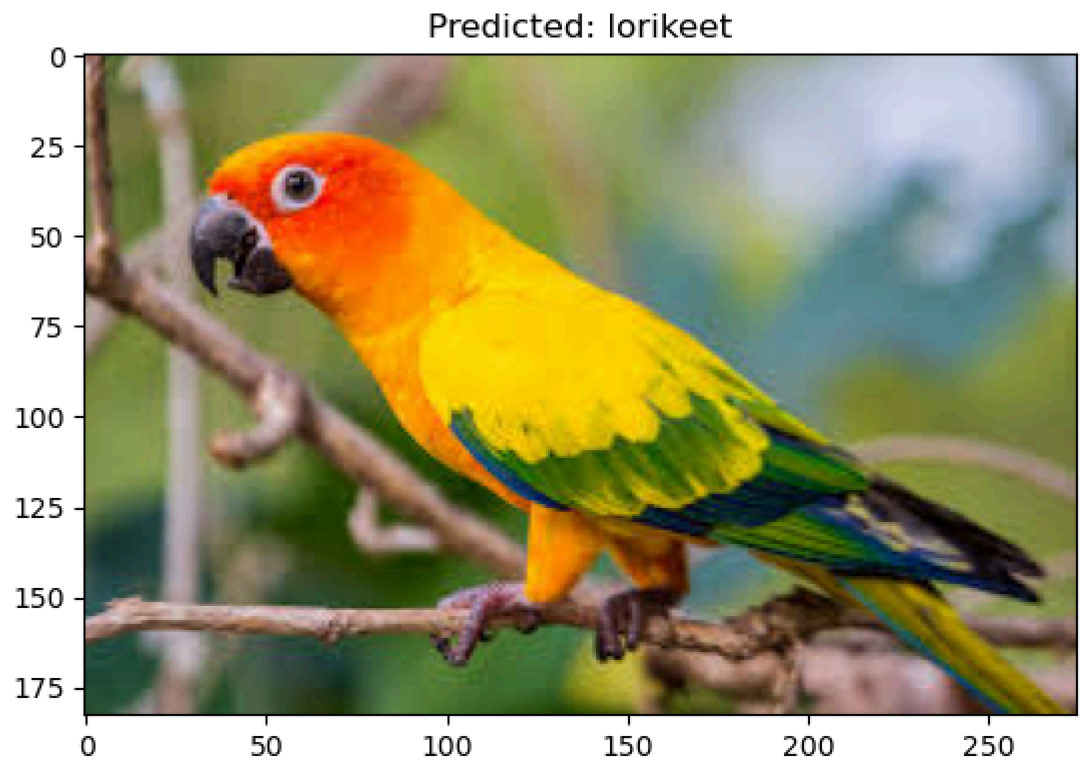
```
import os
import torch
from torchvision import models, transforms
from PIL import Image
import matplotlib.pyplot as plt
import json

# Step 1: Load the pre-trained VGG16 model
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")  # Check if CUDA is available
```

```python
vgg16 = models.vgg16(pretrained=True).to(device)  # Move the model to the appropriate device
vgg16.eval()  # Set the model to evaluation mode

# Step 2: Load ImageNet class labels from the json file
LABELS_PATH = "hw1_data/imagenet_class_index.json"

with open(LABELS_PATH, 'r') as f:
    class_idx = json.load(f)

# Map class indices to their labels
class_names = {int(key): value[1] for key, value in class_idx.items()}

# Step 3: Load and display the images
images_folder = "hw1_data/birds"  # Assuming images are in the "data" folder
image_names = os.listdir(images_folder)

# Step 4: Pre-process the images
preprocess = transforms.Compose([
    transforms.Resize((224, 224)),  # Resize to 224x224
    transforms.ToTensor(),  # Convert to Tensor
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])  # Normalize
])

# Step 5: Process and predict each image
with torch.no_grad():
    for img_name in image_names:
        img_path = os.path.join(images_folder, img_name)
        img = Image.open(img_path).convert("RGB")

        # Preprocess image
        img_tensor = preprocess(img).unsqueeze(0).to(device)  # Add batch dimension and move to device

        # Perform forward pass
        output = vgg16(img_tensor)

        # Get the index of the class with the highest score
        _, predicted_class = torch.max(output, 1)
```

```
        # Get the class label using the class index
        predicted_class_label = class_names[predicted_class.item()]

        # Display the image and predicted class
        plt.imshow(img)
        plt.show()

        # Print the predicted class label for each image
        print(f"Predicted label for {img_name}:
{predicted_class_label}")
```




```
Predicted label for bird_0.jpg: hummingbird
```




```
Predicted label for bird_1.jpg: lorikeet
```

3. Steps that were taken:

1. Resize the images to 224x224 (expected input size for VGG16).
2. Normalize the images using the given mean and standard deviation.
3. Convert the images to tensors.

5:

In [165…

```python
import os
import torch
from torchvision import models, transforms
from PIL import Image
import matplotlib.pyplot as plt
import json

# Step 1: Load the pre-trained VGG16 model
vgg16 = models.vgg16(pretrained=True).to(device)  # Move the model to the appropriate device
vgg16.eval()  # Set the model to evaluation mode

# Step 2: Load ImageNet class labels from the json file
LABELS_PATH = "hw1_data/imagenet_class_index.json"

with open(LABELS_PATH, 'r') as f:
    class_idx = json.load(f)
```

```python
# Map class indices to their labels
class_names = {int(key): value[1] for key, value in class_idx.items()}

# Step 3: Load and display the images
images_folder = "hw1_data/dogs"  # Assuming images are in the "data" folder
image_names = os.listdir(images_folder)

# Step 4: Pre-process the images
preprocess = transforms.Compose([
    transforms.Resize((224, 224)),  # Resize to 224x224
    transforms.ToTensor(),  # Convert to Tensor
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])  # Normalize
])

# Step 5: Process and predict the first image
with torch.no_grad():
    img_name = image_names[0]  # Only process the first image
    img_path = os.path.join(images_folder, img_name)
    img = Image.open(img_path).convert("RGB")

    # Preprocess image
    img_tensor = preprocess(img).unsqueeze(0).to(device)  # Add batch dimension and move to device

    # Perform forward pass
    output = vgg16(img_tensor)

    # Get the index of the class with the highest score
    _, predicted_class = torch.max(output, 1)

    # Get the class label using the class index
    predicted_class_label = class_names[predicted_class.item()]

    # Display the image and predicted class
    plt.imshow(img)
    plt.show()
```

```
    # Print the predicted class label for the first image
    print(f"Predicted label for {img_name}: {predicted_class_label}")
```


```
Predicted label for dog_9.jpg: Blenheim_spaniel
```

6:

In [169…

```
import os
import torch
from torchvision import models, transforms
from PIL import Image
import matplotlib.pyplot as plt
import json

# Step 1: Load the pre-trained VGG16 model
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")  # Check if CUDA is available
vgg16 = models.vgg16(pretrained=True).to(device)  # Move the model to the appropriate device
vgg16.eval()  # Set the model to evaluation mode

# Step 2: Load ImageNet class labels from the json file
LABELS_PATH = "hw1_data/imagenet_class_index.json"

with open(LABELS_PATH, 'r') as f:
    class_idx = json.load(f)

# Map class indices to their labels
class_names = {int(key): value[1] for key, value in class_idx.items()}
```

```
# Step 3: Load the image
images_folder = "hw1_data/dogs"  # Assuming images are in the "data" folder
image_names = os.listdir(images_folder)

# Step 4: Pre-process the image
preprocess = transforms.Compose([
    transforms.Resize((224, 224)),  # Resize to 224x224
    transforms.ToTensor(),  # Convert to Tensor
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])  # Normalize
])

# Step 5: Process the first image
with torch.no_grad():
    img_name = image_names[0]  # Only process the first image
    img_path = os.path.join(images_folder, img_name)
    img = Image.open(img_path).convert("RGB")

    # Preprocess image
    img_tensor = preprocess(img).unsqueeze(0).to(device)  # Add batch dimension and move to device

    # Step 6: Hook to capture the output of the first convolutional layer
    def hook_fn(module, input, output):
        return output

    # Register hook to the first convolutional layer (Conv2d in the first layer of VGG16)
    first_conv_layer = vgg16.features[0]  # First conv layer in VGG16
    hook = first_conv_layer.register_forward_hook(hook_fn)

    # Perform forward pass to get feature maps
    feature_maps = vgg16.features[:4](img_tensor)  # Only get the output from the first conv layer

    # Remove the hook after use
    hook.remove()

    # Step 7: Plot the responses of the first three filters
```

```
    # Extract the first 3 feature maps
    for i in range(3):
        feature_map = feature_maps[0, i].cpu().numpy()  # Get the ith
feature map (channel)

        plt.subplot(1, 3, i+1)
        plt.imshow(feature_map, cmap='gray')
        plt.title(f"Filter {i+1}")
        plt.axis('off')

    plt.show()

    # Print the predicted class label for the first image
    output = vgg16(img_tensor)
    _, predicted_class = torch.max(output, 1)
    predicted_class_label = class_names[predicted_class.item()]
```