isharadilshanra / Deeplearning

Type / to search

Code Issues Pull requests Actions Projects Wiki Security Insights Settings

Deeplearning / CNN / MA4144InClass4(CNN).ipynb

isharadilshanra Custom defined CNN 9343f1e · 1 minute ago History

Convolutional Neural Networks

Inclass Project 4 - MA4144

This project contains 5 tasks/questions to be completed, some require written answers. Open a markdown cell below the respective question that require written answers and provide (type) your answers. Questions that required written answers are given in blue fonts. Almost all written questions are open ended, they do not have a correct or wrong answer. You are free to give your opinions, but please provide related answers within the context.

After finishing project run the entire notebook once and **save the notebook as a pdf** (File menu -> Save and Export Notebook As -> PDF). You are **required to upload both the PDF and the ipynb file on moodle**.

Outline of the project

The aim of the project is to practically learn and implement about CNN. This project will have two main sections.

Section 1: Build a convolutional layer and pooling layer from scratch. Then test them on a sample image.

Section 2: Use the Keras library to implement a CNN to classify images on the CIFAR10 dataset.

Use the below cell to use any include any imports

In [17]:
```
import numpy as np
import matplotlib.pyplot as plt
import random
from keras.preprocessing.image import load_img
from numpy.lib.stride_tricks import sliding_window_view
from keras.preprocessing.image import load_img, img_to_array
import keras
from scipy import signal
```

Section 1: Convolution and Pooling

Q1 In the following cell, implement a method called create_padding. The method will take in input_image ($n \times m$) and will return a zero-padded image called output_image of dimension $(n+2d) \times (m+2d)$ where d is the padding thickness on either side.

In [4]:
```
def create_padding(input_image, d):

    output_image = np.pad(input_image, pad_width=((d, d), (d, d)), mode='constant', constant_values=0)

    return output_image
```

Q2 In the following cell, implement a method called convolution. The method will take in input_image ($n \times m$), kernel ($k \times k$) and will return output_image of dimension $(n-k+1) \times (m-k+1)$. The output_image is the result of the convolution between input_image and kernel. You may assume that the stride is 1.

In [13]:
```
def convolution(input_image, kernel):

    # Flip kernel for convolution (remove flip if cross-correlation is intended)
    kernel = np.flipud(np.fliplr(kernel))

    # Extract sliding windows
    windowed = sliding_window_view(input_image, kernel.shape)

    # Perform element-wise multiplication and sum across the last two axes
    output_image = np.einsum('ijkl,kl->ij', windowed, kernel)

    return output_image
```

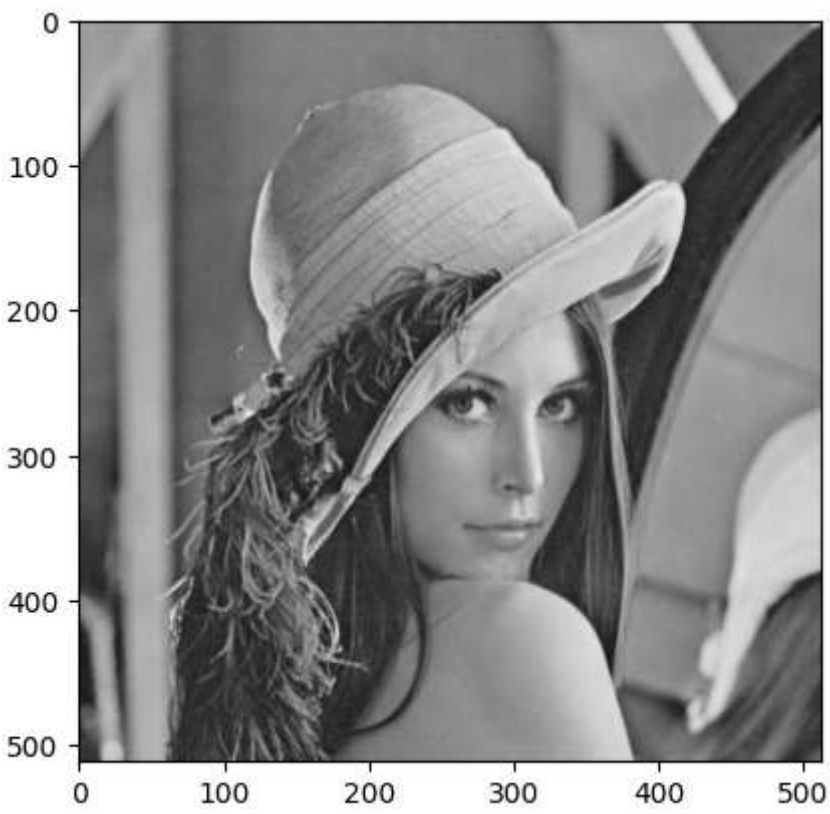
Q3 In the following cell, implement a method called pooling. The method will take in input_image ($n \times m$), p the pooling dimension, pooling_type (either max_pooling or avg_pooling) and will return output_image of dimension $(n-p+1) \times (m-p+1)$. The output_image is the result of performing pooling on input_image by a window of dimension $p \times p$. You may assume that the stride is 1.

In [14]:
```
def pooling(input_image, p, pooling_type = "max_pooling"):

    # Extract sliding windows of shape (p, p)
    windows = sliding_window_view(input_image, (p, p))  # shape: (n-p+1, m-p+1, p, p)

    if pooling_type == "max_pooling":
        output_image = np.max(windows, axis=(2, 3))
```

```
    elif pooling_type == "avg_pooling":
        output_image = np.mean(windows, axis=(2, 3))

    else:
        print("Error: Invalid pooling type")
        return

    return(output_image)
```

The 'lena' image is widely used for image processing experiments and has been a benchmark image until recently. We will use a 512×512 grayscale lena sample to test our convolution and pooling implementations.

In [19]:
```
lena = load_img('lena.gif')
plt.imshow(lena)
plt.show()

lena = img_to_array(lena)  # shape will be (H, W, 1)
lena = lena[:, :, 0]
```



Q4 In the following perform convolution on lena. Make sure you use padding appropriately to maintain the image size after convolution. However, pooling should be done on an unpadded image and image size may not be preseved after pooling. Use the following kernels to perform convolution and pooling separately.

1. $\begin{bmatrix} +1 & 0 & -1 \\ +1 & 0 & -1 \\ +1 & 0 & -1 \end{bmatrix}$
2. $\begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ +1 & +1 & +1 \end{bmatrix}$
3. $\begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix}$
4. $\begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$
5. Any other kernel that you may find interesting.

The following outputs are expected:

1. lena kernel 1 convolution and then max pooled, set your output to the variable lena_kernel_1_maxpool.
2. lena kernel 2 convolution and then average pooled, set your output to the variable lena_kernel_2_avgpool.
3. lena kernel 3 convolution and then max pooled, set your output to the variable lena_kernel_3_maxpool.
4. lena kernel 4 convolution and then average pooled, set your output to the variable lena_kernel_4_avgpool.

In [25]:
```
kernel_1 = np.array([[+1, 0, -1],
                     [+1, 0, -1],
                     [+1, 0, -1]])

kernel_2 = np.array([[-1, -1, -1],
                     [ 0,  0,  0],
                     [+1, +1, +1]])
```

```
kernel_3 = np.array([[-1, 0, +1],
                     [-2, 0, +2],
                     [-1, 0, +1]])

kernel_4 = np.array([[+1, +2, +1],
                     [ 0,  0,  0],
                     [-1, -2, -1]])

kernel_5 = np.array([[ 0,  1,  0],
                     [ 1, -4,  1],
                     [ 0,  1,  0]])
d = 1
p = 2  # Pooling window size

# Convolution and pooling steps
conv_1 = convolution(create_padding(lena, d), kernel_1)
lena_kernel_1_maxpool = pooling(conv_1, p, "max_pooling")

conv_2 = convolution(create_padding(lena, d), kernel_2)
lena_kernel_2_avgpool = pooling(conv_2, p, "avg_pooling")

conv_3 = convolution(create_padding(lena, d), kernel_3)
lena_kernel_3_maxpool = pooling(conv_3, p, "max_pooling")

conv_4 = convolution(create_padding(lena, d), kernel_4)
lena_kernel_4_avgpool = pooling(conv_4, p, "avg_pooling")

conv_5 = convolution(create_padding(lena, d), kernel_5)
lena_kernel_5_avgpool = pooling(conv_5, p, "avg_pooling")
```

Explain what each of the above kernels (including your choice) will do to the image.

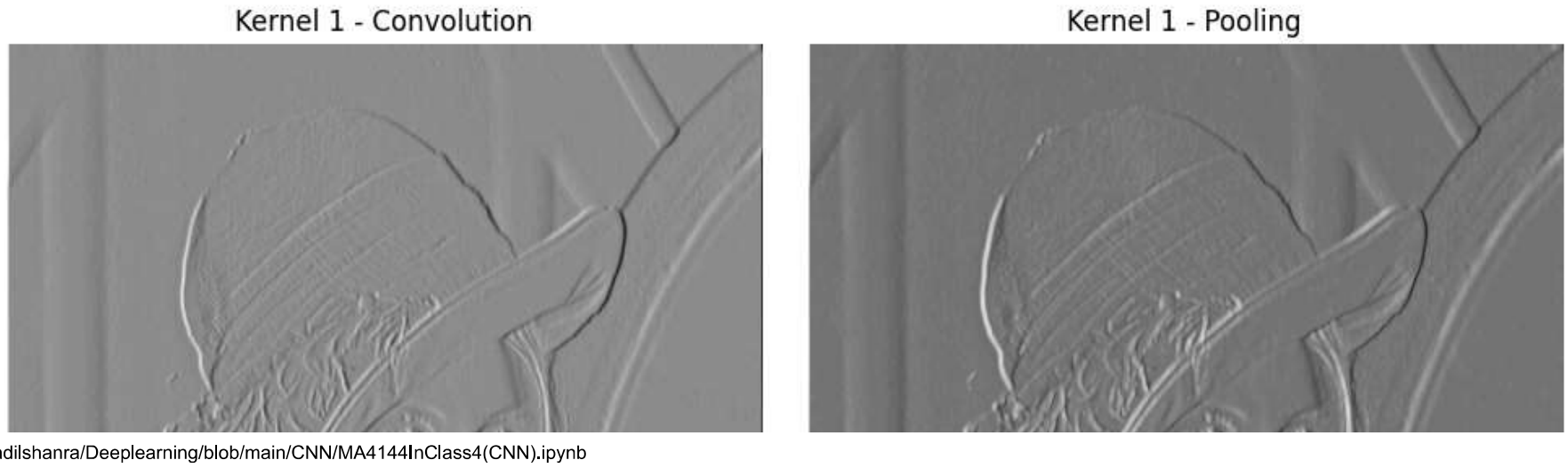
In [26]:

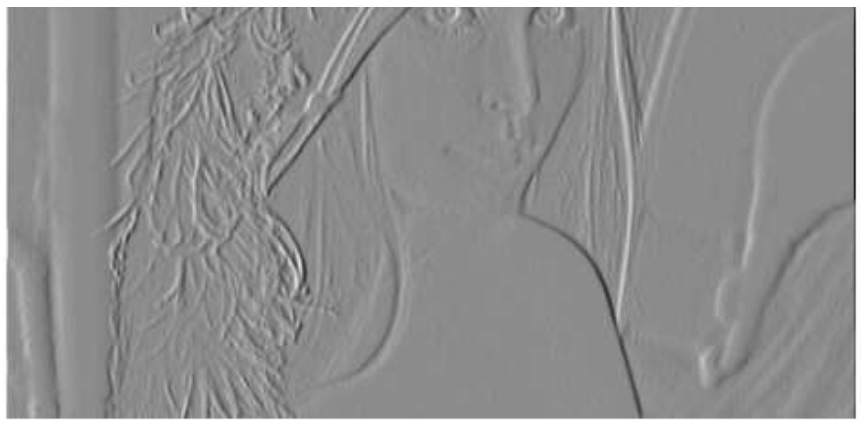
```
Ans_Kernel_1 = "Kernel 1 detects vertical edges by highlighting intensity changes in the horizontal direction. It emphasi

Ans_Kernel_2 = "Kernel 2 detects horizontal edges by responding to vertical intensity changes. It brings out horizontal f

Ans_Kernel_3 = "Kernel 3 is a Sobel filter in the X direction, emphasizing vertical edges with weighted gradients. It pro

Ans_Kernel_4 = "Kernel 4 is a Sobel filter in the Y direction, enhancing horizontal edges with greater sensitivity. It hi

Ans_Kernel_5 = "Kernel 5 is a Laplacian filter that detects edges in all directions by measuring second-order intensity c
```

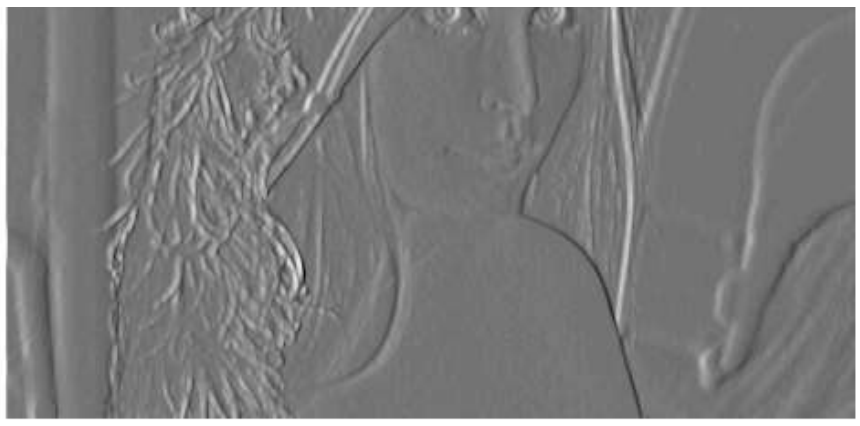
Show the resulting image after convolution and pooling separately on two subplots (of the same plot) for each kernel. There should be 5 plots with two sub plots in each.

In [28]:

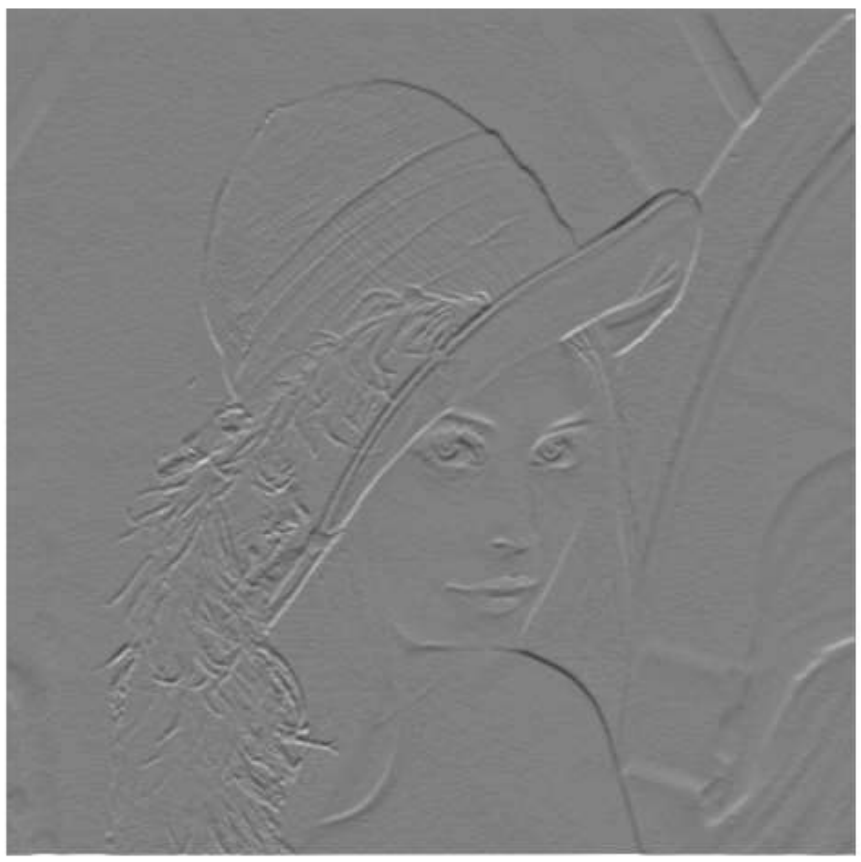
```
def plot_conv_and_pool(conv_img, pool_img, title, cmap='gray'):
    fig, ax = plt.subplots(1, 2, figsize=(10, 5))
    ax[0].imshow(conv_img, cmap=cmap)
    ax[0].set_title(f'{title} - Convolution')
    ax[0].axis('off')

    ax[1].imshow(pool_img, cmap=cmap)
    ax[1].set_title(f'{title} - Pooling')
    ax[1].axis('off')

    plt.tight_layout()
    plt.show()

# Plot for Kernel 1
plot_conv_and_pool(conv_1, lena_kernel_1_maxpool, "Kernel 1")

# Plot for Kernel 2
plot_conv_and_pool(conv_2, lena_kernel_2_avgpool, "Kernel 2")

# Plot for Kernel 3
plot_conv_and_pool(conv_3, lena_kernel_3_maxpool, "Kernel 3")

# Plot for Kernel 4
plot_conv_and_pool(conv_4, lena_kernel_4_avgpool, "Kernel 4")

# Plot for Kernel 5 (laplacian)
plot_conv_and_pool(conv_5, lena_kernel_5_avgpool, "Kernel 5")
```



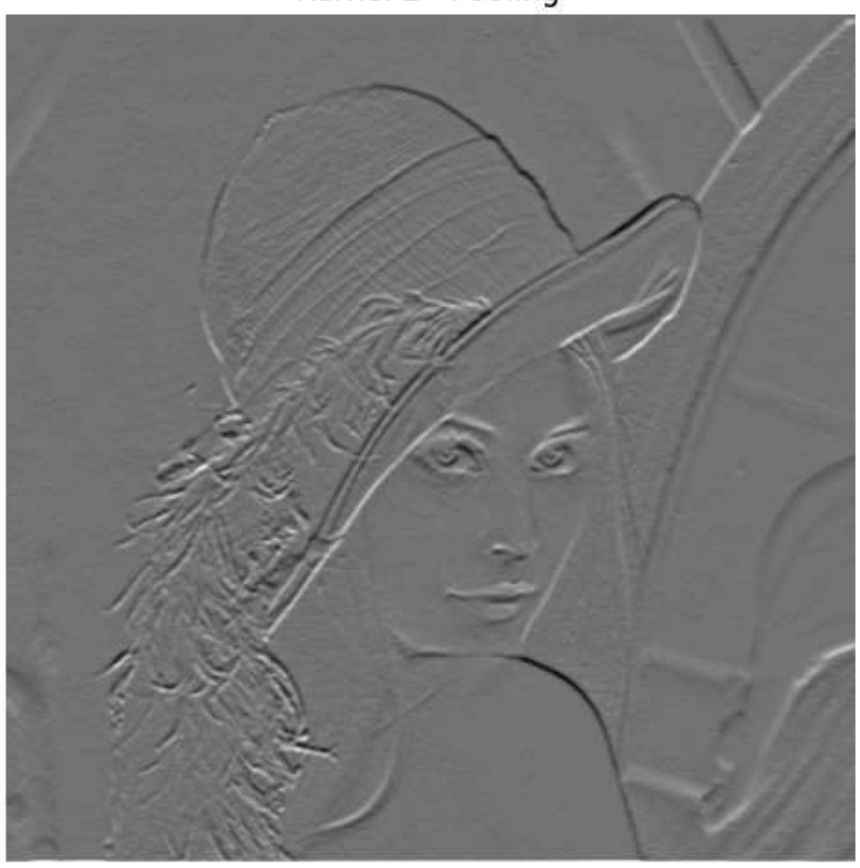




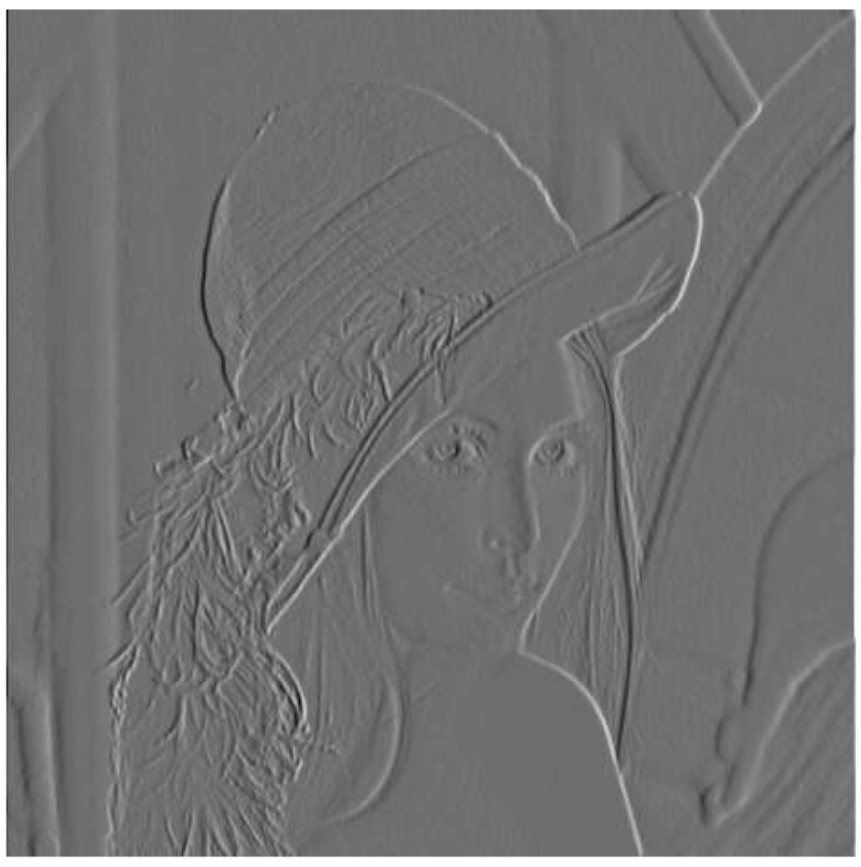
Kernel 2 - Convolution



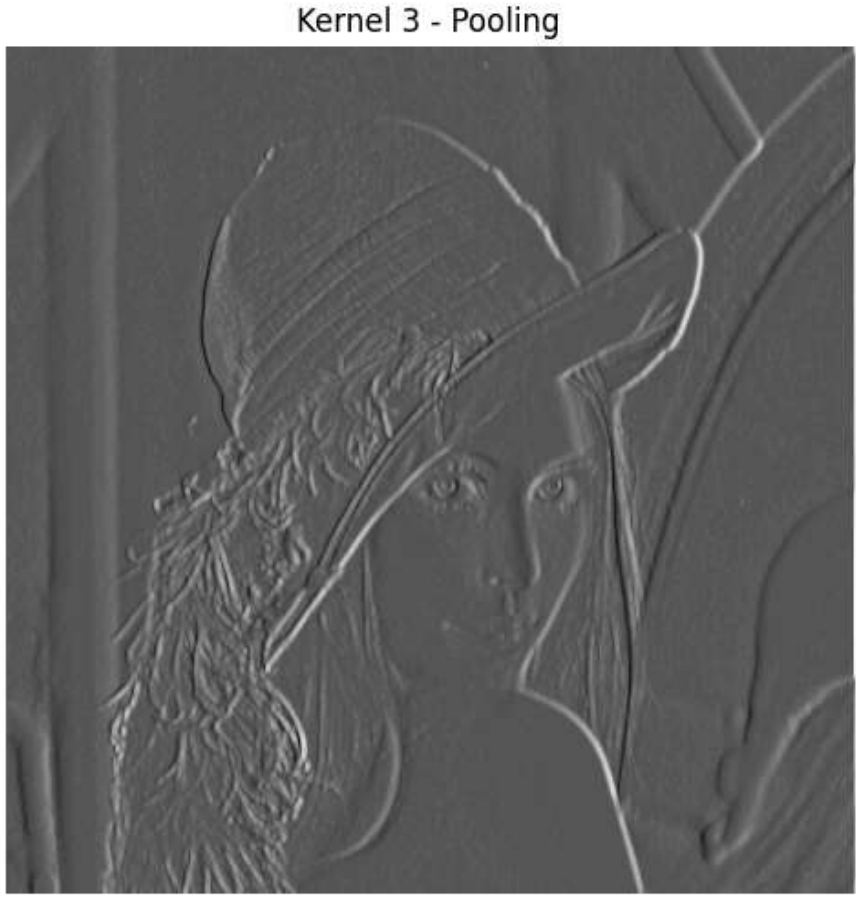
Kernel 2 - Pooling



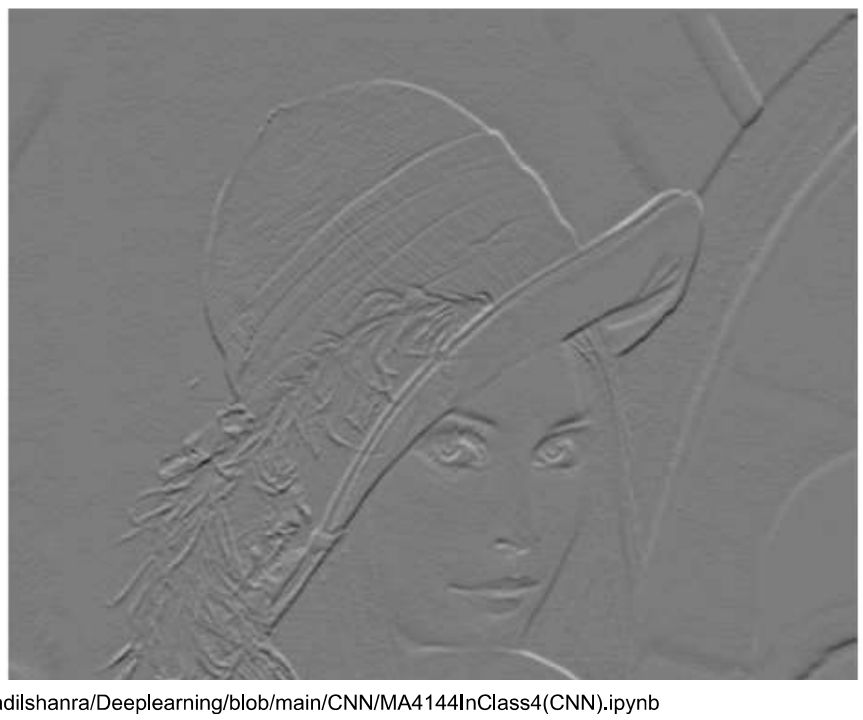
Kernel 3 - Convolution



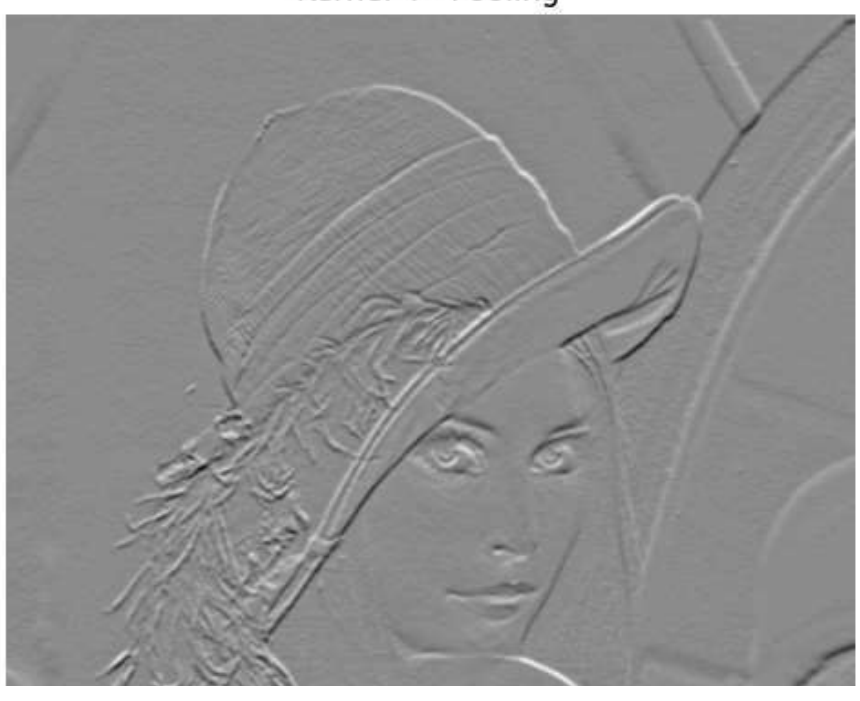
Kernel 3 - Pooling

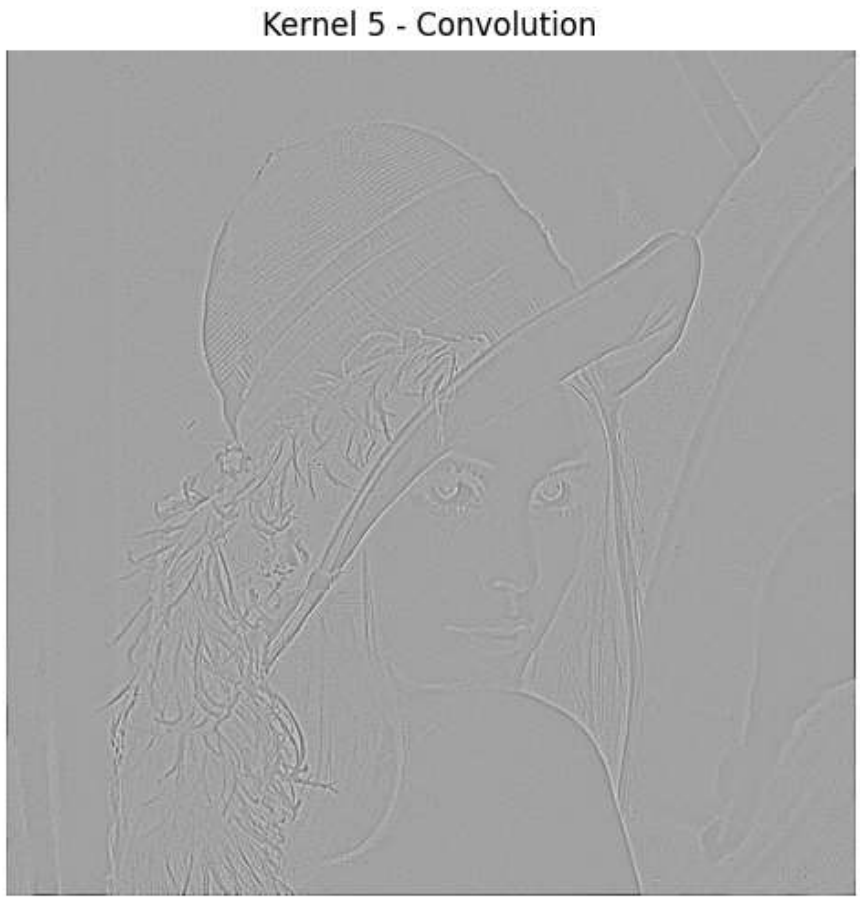


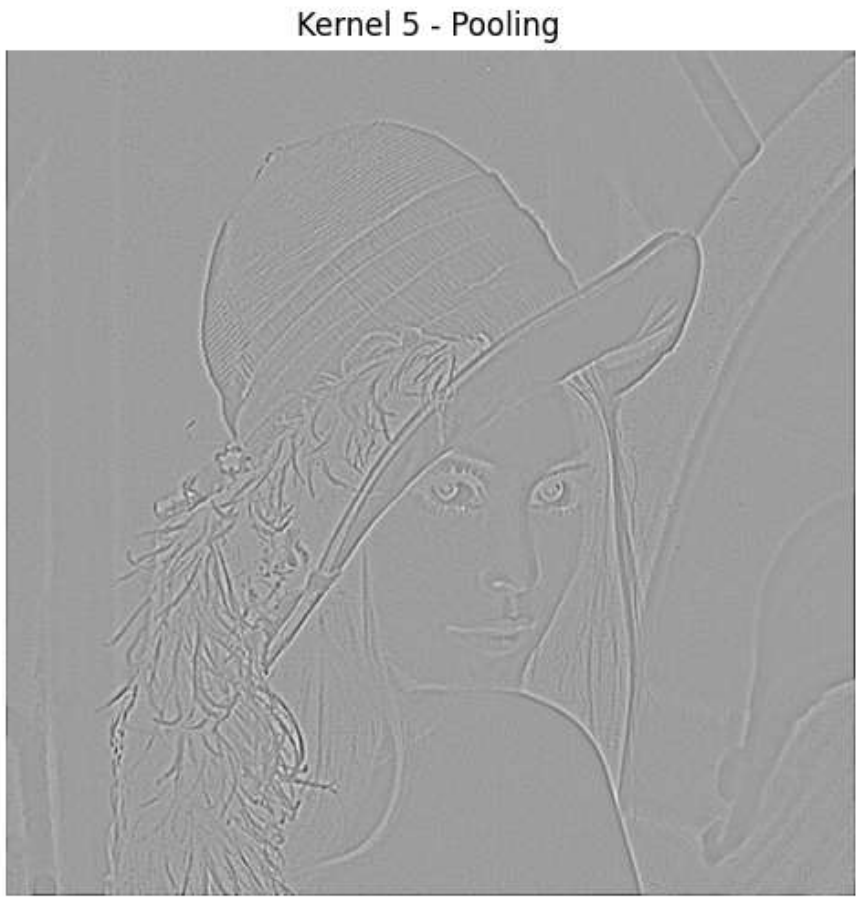
Kernel 4 - Convolution



Kernel 4 - Pooling







Section 2: Using Keras to implement CNN for image classification

This section, unlike the previous projects you are granted full liberty to **build the structure of your model appropriately using keras**. I have provided only the code to download the cifar10 dataset. (Note that cifar10 contains rgb images with 3 channels unlike the grayscake image lena we used earlier.)

Final expected outcome: For the best CNN model architecture and parameters you find, **your model is required to be able to provide accurate predictions**. The **accuracy rate will determine your score**. For this **implement a predict function** - instructions given below.

Hint: To improve your model you may use the following techniques.

1. 5-fold cross validation accuracy.
2. Testing accuracy.
3. Confusion matrix of the result.
4. Precision recall for each class.

Test on different hyperparameters and network architectures and select decide the best performer based on the cross-validation accuracy.

In [29]:
```
(x_train, y_train), (x_test, y_test) = keras.datasets.cifar10.load_data()
```

```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
170498071/170498071 [==============================] - 732s 4us/step
```

In [30]:
```
import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers, models
from sklearn.model_selection import KFold
from sklearn.metrics import classification_report, confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns

# Load CIFAR-10 dataset
(x_train, y_train), (x_test, y_test) = keras.datasets.cifar10.load_data()
y_train = y_train.flatten()
y_test = y_test.flatten()

# Normalize the image data
x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0

# Define class names
class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer',
               'dog', 'frog', 'horse', 'ship', 'truck']

# Define CNN architecture function
def create_model():
    model = models.Sequential()
    model.add(layers.Conv2D(32, (3,3), activation='relu', input_shape=(32,32,3)))
    model.add(layers.Conv2D(32, (3,3), activation='relu'))
    model.add(layers.MaxPooling2D((2,2)))
```

```
    model.add(layers.Dropout(0.25))

    model.add(layers.Conv2D(64, (3,3), activation='relu'))
    model.add(layers.Conv2D(64, (3,3), activation='relu'))
    model.add(layers.MaxPooling2D((2,2)))
    model.add(layers.Dropout(0.25))

    model.add(layers.Flatten())
    model.add(layers.Dense(512, activation='relu'))
    model.add(layers.Dropout(0.5))
    model.add(layers.Dense(10, activation='softmax'))

    model.compile(optimizer='adam',
                  loss='sparse_categorical_crossentropy',
                  metrics=['accuracy'])
    return model

# 5-fold Cross Validation
kf = KFold(n_splits=5, shuffle=True, random_state=42)
cv_scores = []

for fold, (train_idx, val_idx) in enumerate(kf.split(x_train)):
    print(f"\n--- Fold {fold + 1} ---")
    model = create_model()
    model.fit(x_train[train_idx], y_train[train_idx],
              epochs=10, batch_size=64,
              validation_data=(x_train[val_idx], y_train[val_idx]), verbose=2)
    val_loss, val_acc = model.evaluate(x_train[val_idx], y_train[val_idx], verbose=0)
    cv_scores.append(val_acc)

print(f"\nAverage Cross-Validation Accuracy: {np.mean(cv_scores):.4f}")

# Train final model on full training data
final_model = create_model()
final_model.fit(x_train, y_train, epochs=20, batch_size=64, verbose=2,
                validation_split=0.1)

# Evaluate on test set
test_loss, test_acc = final_model.evaluate(x_test, y_test, verbose=0)
print(f"\nTest Accuracy: {test_acc:.4f}")

# Confusion matrix and classification report
y_pred = final_model.predict(x_test)
y_pred_classes = np.argmax(y_pred, axis=1)

print("\nClassification Report:")
print(classification_report(y_test, y_pred_classes, target_names=class_names))

cm = confusion_matrix(y_test, y_pred_classes)
plt.figure(figsize=(10, 8))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=class_names, yticklabels=class_names)
plt.title("Confusion Matrix")
plt.ylabel("Actual")
plt.xlabel("Predicted")
plt.show()
```

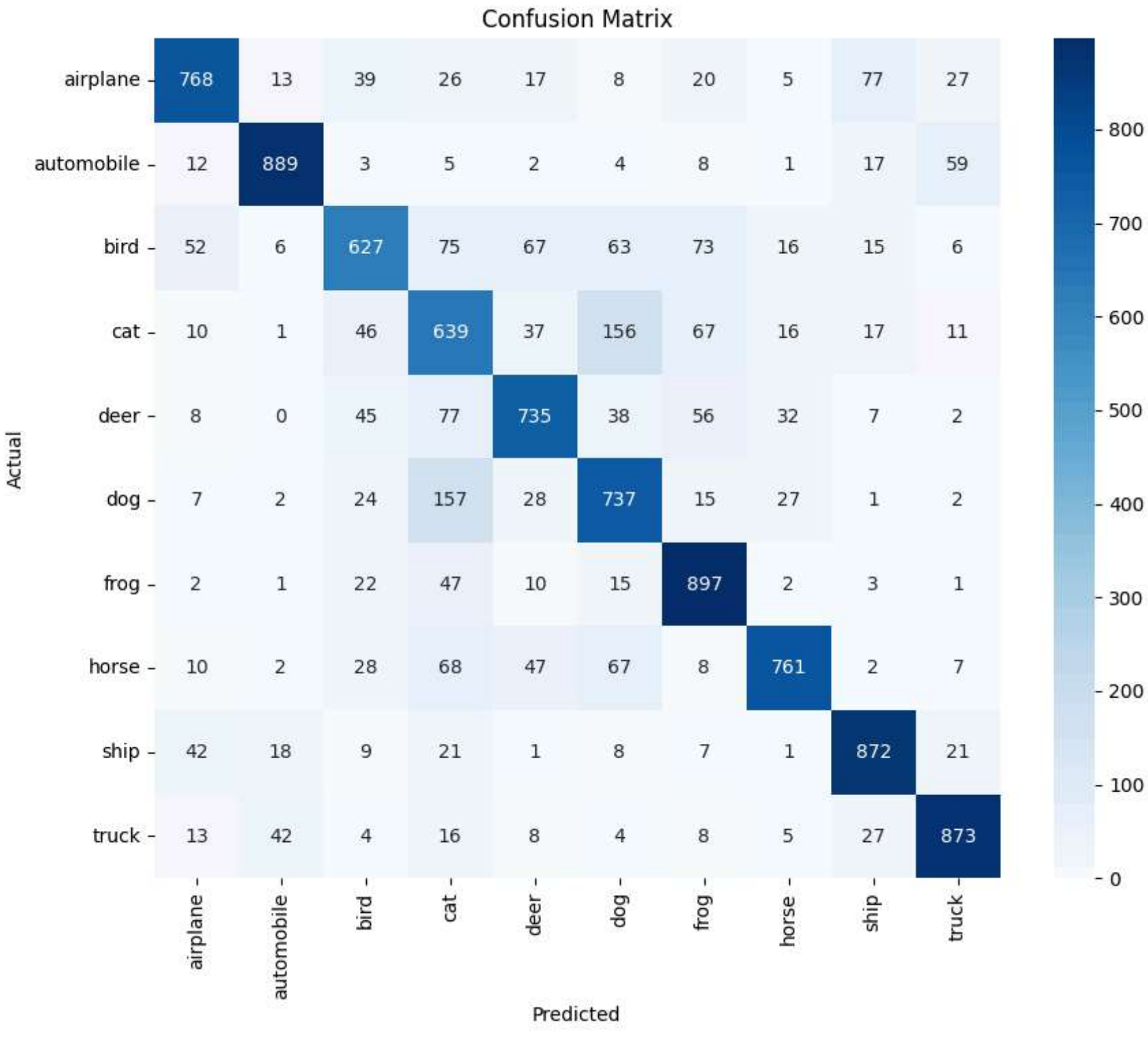
```
--- Fold 1 ---
Epoch 1/10
625/625 - 96s - loss: 1.6418 - accuracy: 0.3981 - val_loss: 1.3692 - val_accuracy: 0.5102 - 96s/epoch - 154ms/step
Epoch 2/10
625/625 - 86s - loss: 1.2513 - accuracy: 0.5533 - val_loss: 1.0842 - val_accuracy: 0.6107 - 86s/epoch - 137ms/step
Epoch 3/10
625/625 - 81s - loss: 1.0838 - accuracy: 0.6161 - val_loss: 0.9169 - val_accuracy: 0.6758 - 81s/epoch - 129ms/step
Epoch 4/10
625/625 - 80s - loss: 0.9744 - accuracy: 0.6571 - val_loss: 0.8601 - val_accuracy: 0.7004 - 80s/epoch - 128ms/step
Epoch 5/10
625/625 - 82s - loss: 0.9000 - accuracy: 0.6841 - val_loss: 0.8060 - val_accuracy: 0.7188 - 82s/epoch - 131ms/step
Epoch 6/10
625/625 - 80s - loss: 0.8354 - accuracy: 0.7076 - val_loss: 0.8186 - val_accuracy: 0.7129 - 80s/epoch - 127ms/step
Epoch 7/10
625/625 - 80s - loss: 0.7856 - accuracy: 0.7239 - val_loss: 0.7674 - val_accuracy: 0.7348 - 80s/epoch - 128ms/step
Epoch 8/10
625/625 - 80s - loss: 0.7467 - accuracy: 0.7361 - val_loss: 0.7202 - val_accuracy: 0.7531 - 80s/epoch - 128ms/step
Epoch 9/10
625/625 - 82s - loss: 0.7099 - accuracy: 0.7552 - val_loss: 0.7460 - val_accuracy: 0.7451 - 82s/epoch - 131ms/step
Epoch 10/10
625/625 - 80s - loss: 0.6820 - accuracy: 0.7613 - val_loss: 0.7027 - val_accuracy: 0.7592 - 80s/epoch - 128ms/step

--- Fold 2 ---
Epoch 1/10
625/625 - 94s - loss: 1.6510 - accuracy: 0.3942 - val_loss: 1.2935 - val_accuracy: 0.5325 - 94s/epoch - 151ms/step
Epoch 2/10
625/625 - 80s - loss: 1.2441 - accuracy: 0.5556 - val_loss: 1.0868 - val_accuracy: 0.6149 - 80s/epoch - 129ms/step
Epoch 3/10
625/625 - 78s - loss: 1.0813 - accuracy: 0.6184 - val_loss: 1.0024 - val_accuracy: 0.6447 - 78s/epoch - 125ms/step
Epoch 4/10
625/625 - 78s - loss: 0.9821 - accuracy: 0.6548 - val_loss: 0.9602 - val_accuracy: 0.6619 - 78s/epoch - 125ms/step
Epoch 5/10
625/625 - 78s - loss: 0.9086 - accuracy: 0.6798 - val_loss: 0.8844 - val_accuracy: 0.6899 - 78s/epoch - 125ms/step
Epoch 6/10
625/625 - 80s - loss: 0.8476 - accuracy: 0.7021 - val_loss: 0.8295 - val_accuracy: 0.7098 - 80s/epoch - 128ms/step
Epoch 7/10
625/625 - 78s - loss: 0.8043 - accuracy: 0.7188 - val_loss: 0.7812 - val_accuracy: 0.7254 - 78s/epoch - 124ms/step
```

```
Epoch 8/10
625/625 - 78s - loss: 0.7625 - accuracy: 0.7345 - val_loss: 0.7742 - val_accuracy: 0.7311 - 78s/epoch - 125ms/step
Epoch 9/10
625/625 - 79s - loss: 0.7314 - accuracy: 0.7420 - val_loss: 0.7847 - val_accuracy: 0.7279 - 79s/epoch - 127ms/step
Epoch 10/10
625/625 - 79s - loss: 0.6924 - accuracy: 0.7576 - val_loss: 0.7183 - val_accuracy: 0.7479 - 79s/epoch - 126ms/step

--- Fold 3 ---
Epoch 1/10
625/625 - 79s - loss: 1.6414 - accuracy: 0.3961 - val_loss: 1.2579 - val_accuracy: 0.5554 - 79s/epoch - 127ms/step
Epoch 2/10
625/625 - 77s - loss: 1.2268 - accuracy: 0.5604 - val_loss: 1.0539 - val_accuracy: 0.6295 - 77s/epoch - 124ms/step
Epoch 3/10
625/625 - 77s - loss: 1.0584 - accuracy: 0.6243 - val_loss: 0.9210 - val_accuracy: 0.6770 - 77s/epoch - 124ms/step
Epoch 4/10
625/625 - 78s - loss: 0.9587 - accuracy: 0.6615 - val_loss: 0.8961 - val_accuracy: 0.6866 - 78s/epoch - 125ms/step
Epoch 5/10
625/625 - 77s - loss: 0.8789 - accuracy: 0.6905 - val_loss: 0.8166 - val_accuracy: 0.7190 - 77s/epoch - 123ms/step
Epoch 6/10
625/625 - 77s - loss: 0.8240 - accuracy: 0.7110 - val_loss: 0.7763 - val_accuracy: 0.7308 - 77s/epoch - 123ms/step
Epoch 7/10
625/625 - 77s - loss: 0.7747 - accuracy: 0.7259 - val_loss: 0.7915 - val_accuracy: 0.7287 - 77s/epoch - 123ms/step
Epoch 8/10
```



```
--- Fold 4 ---
Epoch 1/10
625/625 - 81s - loss: 1.6676 - accuracy: 0.3823 - val_loss: 1.3017 - val_accuracy: 0.5307 - 81s/epoch - 130ms/step
Epoch 2/10
625/625 - 77s - loss: 1.2846 - accuracy: 0.5390 - val_loss: 1.1119 - val_accuracy: 0.6116 - 77s/epoch - 124ms/step
Epoch 3/10
625/625 - 77s - loss: 1.1392 - accuracy: 0.5928 - val_loss: 1.0021 - val_accuracy: 0.6427 - 77s/epoch - 123ms/step
Epoch 4/10
625/625 - 77s - loss: 1.0330 - accuracy: 0.6317 - val_loss: 0.8971 - val_accuracy: 0.6834 - 77s/epoch - 123ms/step
Epoch 5/10
625/625 - 77s - loss: 0.9695 - accuracy: 0.6581 - val_loss: 0.8930 - val_accuracy: 0.6800 - 77s/epoch - 124ms/step
Epoch 6/10
625/625 - 77s - loss: 0.9086 - accuracy: 0.6786 - val_loss: 0.8119 - val_accuracy: 0.7159 - 77s/epoch - 124ms/step
Epoch 7/10
625/625 - 77s - loss: 0.8520 - accuracy: 0.6976 - val_loss: 0.7941 - val_accuracy: 0.7188 - 77s/epoch - 123ms/step
Epoch 8/10
625/625 - 77s - loss: 0.8044 - accuracy: 0.7171 - val_loss: 0.7652 - val_accuracy: 0.7313 - 77s/epoch - 123ms/step
Epoch 9/10
625/625 - 79s - loss: 0.7660 - accuracy: 0.7293 - val_loss: 0.7519 - val_accuracy: 0.7342 - 79s/epoch - 126ms/step
Epoch 10/10
625/625 - 78s - loss: 0.7368 - accuracy: 0.7403 - val_loss: 0.7358 - val_accuracy: 0.7394 - 78s/epoch - 125ms/step

--- Fold 5 ---
Epoch 1/10
625/625 - 80s - loss: 1.6311 - accuracy: 0.3986 - val_loss: 1.2716 - val_accuracy: 0.5348 - 80s/epoch - 129ms/step
Epoch 2/10
625/625 - 78s - loss: 1.2332 - accuracy: 0.5581 - val_loss: 1.0474 - val_accuracy: 0.6273 - 78s/epoch - 125ms/step
Epoch 3/10
625/625 - 79s - loss: 1.0739 - accuracy: 0.6160 - val_loss: 0.9635 - val_accuracy: 0.6605 - 79s/epoch - 126ms/step
Epoch 4/10
625/625 - 86s - loss: 0.9670 - accuracy: 0.6587 - val_loss: 0.8999 - val_accuracy: 0.6839 - 86s/epoch - 138ms/step
Epoch 5/10
625/625 - 82s - loss: 0.8971 - accuracy: 0.6812 - val_loss: 0.8621 - val_accuracy: 0.6983 - 82s/epoch - 131ms/step
Epoch 6/10
625/625 - 78s - loss: 0.8366 - accuracy: 0.7027 - val_loss: 0.7699 - val_accuracy: 0.7297 - 78s/epoch - 125ms/step
Epoch 7/10
625/625 - 78s - loss: 0.7877 - accuracy: 0.7225 - val_loss: 0.7895 - val_accuracy: 0.7222 - 78s/epoch - 125ms/step
Epoch 8/10
625/625 - 83s - loss: 0.7543 - accuracy: 0.7342 - val_loss: 0.7475 - val_accuracy: 0.7358 - 83s/epoch - 132ms/step
Epoch 9/10
625/625 - 89s - loss: 0.7145 - accuracy: 0.7484 - val_loss: 0.7234 - val_accuracy: 0.7475 - 89s/epoch - 142ms/step
Epoch 10/10
625/625 - 82s - loss: 0.6791 - accuracy: 0.7612 - val_loss: 0.7053 - val_accuracy: 0.7545 - 82s/epoch - 132ms/step

Average Cross-Validation Accuracy: 0.7508
Epoch 1/20
704/704 - 91s - loss: 1.6128 - accuracy: 0.4053 - val_loss: 1.5120 - val_accuracy: 0.4738 - 91s/epoch - 129ms/step
Epoch 2/20
704/704 - 90s - loss: 1.2065 - accuracy: 0.5702 - val_loss: 1.0528 - val_accuracy: 0.6336 - 90s/epoch - 129ms/step
Epoch 3/20
704/704 - 87s - loss: 1.0394 - accuracy: 0.6295 - val_loss: 0.9055 - val_accuracy: 0.6856 - 87s/epoch - 124ms/step
Epoch 4/20
704/704 - 87s - loss: 0.9323 - accuracy: 0.6697 - val_loss: 0.7842 - val_accuracy: 0.7254 - 87s/epoch - 123ms/step
Epoch 5/20
704/704 - 87s - loss: 0.8590 - accuracy: 0.6985 - val_loss: 0.7684 - val_accuracy: 0.7224 - 87s/epoch - 123ms/step
Epoch 6/20
704/704 - 86s - loss: 0.8039 - accuracy: 0.7174 - val_loss: 0.7320 - val_accuracy: 0.7518 - 86s/epoch - 123ms/step
Epoch 7/20
704/704 - 86s - loss: 0.7580 - accuracy: 0.7318 - val_loss: 0.6996 - val_accuracy: 0.7548 - 86s/epoch - 123ms/step
Epoch 8/20
704/704 - 88s - loss: 0.7139 - accuracy: 0.7467 - val_loss: 0.6675 - val_accuracy: 0.7680 - 88s/epoch - 125ms/step
Epoch 9/20
704/704 - 87s - loss: 0.6881 - accuracy: 0.7570 - val_loss: 0.6571 - val_accuracy: 0.7732 - 87s/epoch - 123ms/step
Epoch 10/20
```

```
704/704 - 87s - loss: 0.6580 - accuracy: 0.7672 - val_loss: 0.6550 - val_accuracy: 0.7730 - 87s/epoch - 123ms/step
Epoch 11/20
704/704 - 86s - loss: 0.6316 - accuracy: 0.7765 - val_loss: 0.6279 - val_accuracy: 0.7816 - 86s/epoch - 122ms/step
Epoch 12/20
704/704 - 87s - loss: 0.6115 - accuracy: 0.7859 - val_loss: 0.6407 - val_accuracy: 0.7794 - 87s/epoch - 124ms/step
Epoch 13/20
704/704 - 86s - loss: 0.5967 - accuracy: 0.7898 - val_loss: 0.6448 - val_accuracy: 0.7778 - 86s/epoch - 123ms/step
Epoch 14/20
704/704 - 87s - loss: 0.5702 - accuracy: 0.7987 - val_loss: 0.6352 - val_accuracy: 0.7782 - 87s/epoch - 123ms/step
Epoch 15/20
704/704 - 86s - loss: 0.5542 - accuracy: 0.8032 - val_loss: 0.6287 - val_accuracy: 0.7920 - 86s/epoch - 122ms/step
Epoch 16/20
704/704 - 88s - loss: 0.5431 - accuracy: 0.8090 - val_loss: 0.6120 - val_accuracy: 0.7920 - 88s/epoch - 125ms/step
Epoch 17/20
704/704 - 88s - loss: 0.5263 - accuracy: 0.8140 - val_loss: 0.6476 - val_accuracy: 0.7802 - 88s/epoch - 125ms/step
Epoch 18/20
704/704 - 88s - loss: 0.5179 - accuracy: 0.8147 - val_loss: 0.6192 - val_accuracy: 0.7942 - 88s/epoch - 125ms/step
Epoch 19/20
704/704 - 87s - loss: 0.5048 - accuracy: 0.8204 - val_loss: 0.6337 - val_accuracy: 0.7862 - 87s/epoch - 124ms/step
Epoch 20/20
704/704 - 88s - loss: 0.4964 - accuracy: 0.8237 - val_loss: 0.6452 - val_accuracy: 0.7862 - 88s/epoch - 124ms/step

Test Accuracy: 0.7798
313/313 [==============================] - 6s 12ms/step

Classification Report:
              precision    recall  f1-score   support

    airplane       0.83      0.77      0.80      1000
  automobile       0.91      0.89      0.90      1000
        bird       0.74      0.63      0.68      1000
         cat       0.56      0.64      0.60      1000
        deer       0.77      0.73      0.75      1000
         dog       0.67      0.74      0.70      1000
        frog       0.77      0.90      0.83      1000
       horse       0.88      0.76      0.82      1000
        ship       0.84      0.87      0.86      1000
       truck       0.87      0.87      0.87      1000

    accuracy                           0.78     10000
   macro avg       0.78      0.78      0.78     10000
weighted avg       0.78      0.78      0.78     10000
```



In [33]:
```
import torch
```

```
import numpy as np
from tensorflow.keras.datasets import cifar10

# Load CIFAR-10 data
(_, _), (x_test, _) = cifar10.load_data()

# Pick 64 test images
x_test_sample = x_test[:64]  # Shape: [64, 32, 32, 3]

# Normalize and convert to PyTorch format: [64, 3, 32, 32]
x_test_sample = x_test_sample.astype('float32') / 255.0
x_test_sample = np.transpose(x_test_sample, (0, 3, 1, 2))  # [64, 3, 32, 32]
torch_test_batch = torch.tensor(x_test_sample)
```

In [34]:

```
#In this cell define the function predict, which will predict the class of test_images using your best CNN-model.
#The output should be a python list of strings of classes (airplane, automobile, bird, cat, deer, dog, frog, horse, ship,
#Your work will be mainly evaluated based on this.
#The grader will test it against input images and your score will be determined based on the number of correct prediction
#Expect the input to be a torch tensor of shape [64, 3, 32, 32]

import numpy as np
import torch

class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer',
               'dog', 'frog', 'horse', 'ship', 'truck']

def predict(test_images):

    # Convert from PyTorch tensor to NumPy array
    if isinstance(test_images, torch.Tensor):
        test_images = test_images.detach().cpu().numpy()

    # Transpose from [N, 3, 32, 32] to [N, 32, 32, 3] for Keras model
    test_images = np.transpose(test_images, (0, 2, 3, 1))

    # Normalize pixel values to [0, 1]
    test_images = test_images.astype('float32') / 255.0

    # Predict using the trained Keras model (assumed to be named `final_model`)
    preds = final_model.predict(test_images, verbose=0)

    # Get class indices and convert to class names
    class_indices = np.argmax(preds, axis=1)
    predictions = [class_names[i] for i in class_indices]

    return predictions

predicted_classes = predict(torch_test_batch)
print(predicted_classes)  # → ['cat', 'dog', 'frog', ..., 'airplane']
```

```
['deer', 'deer', 'deer', 'deer', 'deer', 'deer', 'deer', 'deer', 'deer', 'deer', 'deer', 'deer', 'deer', 'deer', 'deer', 'deer', 'd
eer', 'deer', 'deer', 'deer', 'deer', 'deer', 'deer', 'deer', 'deer', 'deer', 'deer', 'deer', 'deer', 'deer', 'deer', 'deer', 'dee
r', 'deer', 'deer', 'deer', 'deer', 'deer', 'deer', 'deer', 'deer', 'deer', 'deer', 'deer', 'deer', 'deer', 'deer', 'deer', 'deer',
'deer', 'deer', 'deer', 'deer', 'deer', 'deer', 'deer', 'deer', 'deer', 'deer', 'deer', 'deer', 'deer', 'deer', 'deer', 'deer', 'de
er', 'deer', 'deer']
```