Deep Learning Assignment : Image Classification Using CNNs

Name = Manoj Kumar

Batch = 1st September Batch

Course = Data Science Placement Guarantee Course

Email = manojkumarrajput@gmail.com

Objective

This assignment aims to help you understand the basics of Convolutional Neural Networks

(CNNs), their implementation, and evaluation in image classification tasks

Instructions

Complete the tasks below. Each task specifies the marks assigned. Submit your code,

outputs, and a brief explanation for each step. Use the CIFAR-10 dataset for this

assignment.

Importing the required packages

In [1]:
```
import tensorflow as tf
from tensorflow.keras import layers, models
import matplotlib.pyplot as plt
import numpy as np
from sklearn.metrics import classification_report, confusion_matrix
import seaborn as sns
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.regularizers import l2
from sklearn.model_selection import train_test_split
from tensorflow.keras.callbacks import EarlyStopping
```

Loading the dataset

In [3]:
```
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
```

Task 1: Data Exploration and Preparation

Verifying the shapes

In [7]:
```
print("Training data shape:", x_train.shape)
print("Test data shape:", x_test.shape)
```

```
Training data shape: (50000, 32, 32, 3)
Test data shape: (10000, 32, 32, 3)
```

Count of unique labels.

In [10]:
```
print("Unique labels:", np.unique(y_train))
```

```
Unique labels: [0 1 2 3 4 5 6 7 8 9]
```

Displaying 5 sample images with labels

In [13]:
```
def show_images(x_train, y_train, x_test, y_test):
    class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer',
                   'dog', 'frog', 'horse', 'ship', 'truck']

    plt.figure(figsize=(10, 5))
    for i in range(5):
        plt.subplot(1, 5, i+1)
        plt.imshow(x_train[i])
        plt.title(class_names[y_train[i][0]])
        plt.axis('off')

    plt.show()

    return x_train, y_train, x_test, y_test
```

displaying sample images using "show_image" function which i created above from the training and testing data to

help visualize the dataset.

In [16]:
```
x_train, y_train, x_test, y_test = show_images(x_train, y_train, x_test, y_test)
```

Printing the count of unique labels

In [19]:
```
unique_labels, counts = np.unique(y_train, return_counts=True)
print("Unique Labels:", unique_labels)
print("Label Distribution:", counts)
```

```
Unique Labels: [0 1 2 3 4 5 6 7 8 9]
Label Distribution: [5000 5000 5000 5000 5000 5000 5000 5000 5000 5000]
```

Preprocess Data

1. Normalizing pixel values to [0,1] range

Here i am making a function and this function prepares the data before training. It first scales the pixel values between 0 and 1

to help the model learn better.Then, it splits 5000 images from the training set for validation, so we can check the model's progress.

The rest is used for training, while the test set stays the same for final evaluation. This process helps improve accuracy and

performance of the model.

In [24]:
```
def preprocess_data(x_train, y_train, x_test, y_test):
    x_train, x_test = x_train / 255.0, x_test / 255.0
    x_val, y_val = x_train[:5000], y_train[:5000]
    x_train, y_train = x_train[5000:], y_train[5000:]

    return x_train, y_train, x_val, y_val, x_test, y_test
```

Converting data to float and dividing by 255 to ensures all values are between 0 and 1, making it easier for the model to process.

In [27]:
```
def normalize_data(x):
    if x.max() <= 1.0:
        print("Data is already normalized! Skipping normalization.")
        return x
    x = x.astype('float32') / 255.0
    return x
```

In [29]:
```
x_train = normalize_data(x_train)
x_test = normalize_data(x_test)
```

Spliting training data into 80% training and 20% validation

In [32]:
```
def split_train_validation(x_train, y_train, test_size=0.2, random_state=42):
    x_train_new, x_val, y_train_new, y_val = train_test_split(x_train, y_train, tes
    return x_train_new, y_train_new, x_val, y_val
```

In [34]:
```
x_train, y_train, x_val, y_val = split_train_validation(x_train, y_train)
```

In [36]:
```
x_train.min()
```

Out[36]: 0.0

In [38]:
```
x_train.max()
```

Out[38]: 1.0

In [40]:
```
print("Training Set Shape:", x_train.shape, y_train.shape)
print("Validation Set Shape:", x_val.shape, y_val.shape)
```

```
Training Set Shape: (40000, 32, 32, 3) (40000, 1)
Validation Set Shape: (10000, 32, 32, 3) (10000, 1)
```

Task 2: Build and Train a CNN Model

In [43]:
```
def build_model():
    model = models.Sequential([
        layers.Conv2D(64, (3,3), activation='relu', padding='same', input_shape=(32
                      kernel_regularizer=l2(0.0005)),
        layers.BatchNormalization(),
        layers.MaxPooling2D((2,2)),
        layers.Dropout(0.3),

        layers.Conv2D(128, (3,3), activation='relu',padding='same' , kernel_regular
        layers.BatchNormalization(),
        layers.MaxPooling2D((2,2)),
        layers.Dropout(0.3),

        layers.Conv2D(256, (3,3), activation='relu', padding='same', kernel_regular
        layers.BatchNormalization(),
        layers.MaxPooling2D((2,2)),
        layers.Dropout(0.4),

        layers.Flatten(),
        layers.Dense(512, activation='relu', kernel_regularizer=l2(0.0005)),
        layers.Dropout(0.4),
        layers.Dense(10, activation='softmax')])
    model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics
    return model
```

In [45]:
```
model = build_model()
model.summary()
```

```
C:\Users\msgme\anaconda3\Lib\site-packages\keras\src\layers\convolutional\base_conv.
py:107: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. W
hen using Sequential models, prefer using an `Input(shape)` object as the first laye
r in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

Model: "sequential"

Layer (type)	Output Shape	
conv2d (Conv2D)	(None, 32, 32, 64)	
batch_normalization (BatchNormalization)	(None, 32, 32, 64)	
max_pooling2d (MaxPooling2D)	(None, 16, 16, 64)	
dropout (Dropout)	(None, 16, 16, 64)	
conv2d_1 (Conv2D)	(None, 16, 16, 128)	
batch_normalization_1 (BatchNormalization)	(None, 16, 16, 128)	
max_pooling2d_1 (MaxPooling2D)	(None, 8, 8, 128)	
dropout_1 (Dropout)	(None, 8, 8, 128)	
conv2d_2 (Conv2D)	(None, 8, 8, 256)	
batch_normalization_2 (BatchNormalization)	(None, 8, 8, 256)	
max_pooling2d_2 (MaxPooling2D)	(None, 4, 4, 256)	
dropout_2 (Dropout)	(None, 4, 4, 256)	
flatten (Flatten)	(None, 4096)	
dense (Dense)	(None, 512)	
dropout_3 (Dropout)	(None, 512)	
dense_1 (Dense)	(None, 10)	

Total params: 2,475,402 (9.44 MB)

Trainable params: 2,474,506 (9.44 MB)

Non-trainable params: 896 (3.50 KB)

CNN Model Summary Explanation:

This model is built to classify images from the CIFAR-10 dataset. It has three layers that detect patterns in images,

like edges and shapes. Batch Normalization helps the model learn faster, and MaxPooling reduces image size to focus on

important details. Dropout layers prevent overfitting, so the model doesn't just memorize but learns properly. The final

part of the model flattens the data and passes it through a dense layer before making a prediction. Overall, it has about

2.4 million trainable settings, making it strong enough to recognize different objects

Training the Model

Ploting training & validation loss and accuracy

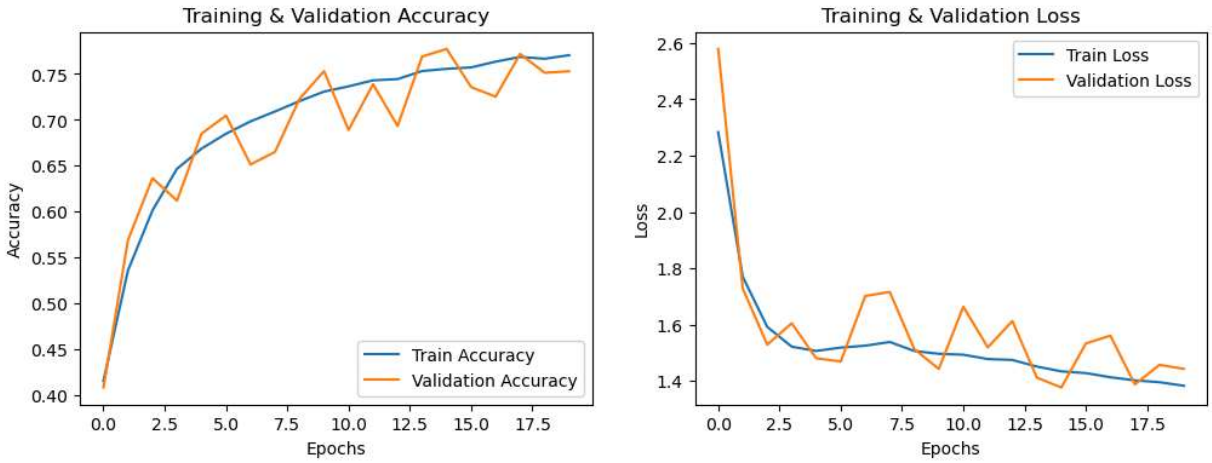
In [49]:
```
early_stopping = EarlyStopping(monitor='val_loss', patience=5, restore_best_weights
```

In [51]:
```
def train_model(model, x_train, y_train, x_val, y_val):
    history = model.fit(x_train, y_train, epochs=20, validation_data=(x_val, y_val)

    plt.figure(figsize=(12, 4))

    plt.subplot(1, 2, 1)
    plt.plot(history.history['accuracy'], label='Train Accuracy')
    plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy')
    plt.legend()
    plt.title('Training & Validation Accuracy')

    plt.subplot(1, 2, 2)
    plt.plot(history.history['loss'], label='Train Loss')
    plt.plot(history.history['val_loss'], label='Validation Loss')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.legend()
    plt.title('Training & Validation Loss')

    plt.show()
```

In [53]:
```
train_model(model, x_train, y_train, x_val, y_val)
```

```
Epoch 1/20
625/625 ━━━━━━━━━━━━━━━━━━━━ 165s 248ms/step - accuracy: 0.3439 - loss: 2.8669 - val
_accuracy: 0.4084 - val_loss: 2.5792
Epoch 2/20
625/625 ━━━━━━━━━━━━━━━━━━━━ 155s 247ms/step - accuracy: 0.5193 - loss: 1.8250 - val
_accuracy: 0.5690 - val_loss: 1.7282
Epoch 3/20
625/625 ━━━━━━━━━━━━━━━━━━━━ 154s 247ms/step - accuracy: 0.5946 - loss: 1.6031 - val
_accuracy: 0.6365 - val_loss: 1.5286
Epoch 4/20
625/625 ━━━━━━━━━━━━━━━━━━━━ 154s 246ms/step - accuracy: 0.6443 - loss: 1.5116 - val
_accuracy: 0.6120 - val_loss: 1.6044
Epoch 5/20
625/625 ━━━━━━━━━━━━━━━━━━━━ 154s 246ms/step - accuracy: 0.6668 - loss: 1.4796 - val
_accuracy: 0.6849 - val_loss: 1.4801
Epoch 6/20
625/625 ━━━━━━━━━━━━━━━━━━━━ 154s 246ms/step - accuracy: 0.6830 - loss: 1.5075 - val
_accuracy: 0.7047 - val_loss: 1.4688
Epoch 7/20
625/625 ━━━━━━━━━━━━━━━━━━━━ 151s 241ms/step - accuracy: 0.7012 - loss: 1.5071 - val
_accuracy: 0.6514 - val_loss: 1.7014
Epoch 8/20
625/625 ━━━━━━━━━━━━━━━━━━━━ 124s 198ms/step - accuracy: 0.7143 - loss: 1.5174 - val
_accuracy: 0.6651 - val_loss: 1.7158
Epoch 9/20
625/625 ━━━━━━━━━━━━━━━━━━━━ 128s 204ms/step - accuracy: 0.7284 - loss: 1.4817 - val
_accuracy: 0.7229 - val_loss: 1.5128
Epoch 10/20
625/625 ━━━━━━━━━━━━━━━━━━━━ 129s 206ms/step - accuracy: 0.7370 - loss: 1.4715 - val
_accuracy: 0.7531 - val_loss: 1.4423
Epoch 11/20
625/625 ━━━━━━━━━━━━━━━━━━━━ 130s 209ms/step - accuracy: 0.7398 - loss: 1.4779 - val
_accuracy: 0.6890 - val_loss: 1.6633
Epoch 12/20
625/625 ━━━━━━━━━━━━━━━━━━━━ 130s 208ms/step - accuracy: 0.7498 - loss: 1.4550 - val
_accuracy: 0.7389 - val_loss: 1.5187
Epoch 13/20
625/625 ━━━━━━━━━━━━━━━━━━━━ 134s 214ms/step - accuracy: 0.7569 - loss: 1.4426 - val
_accuracy: 0.6933 - val_loss: 1.6126
Epoch 14/20
625/625 ━━━━━━━━━━━━━━━━━━━━ 132s 211ms/step - accuracy: 0.7607 - loss: 1.4308 - val
_accuracy: 0.7689 - val_loss: 1.4117
Epoch 15/20
625/625 ━━━━━━━━━━━━━━━━━━━━ 132s 212ms/step - accuracy: 0.7601 - loss: 1.4184 - val
_accuracy: 0.7773 - val_loss: 1.3760
Epoch 16/20
625/625 ━━━━━━━━━━━━━━━━━━━━ 133s 212ms/step - accuracy: 0.7628 - loss: 1.4043 - val
_accuracy: 0.7357 - val_loss: 1.5329
Epoch 17/20
625/625 ━━━━━━━━━━━━━━━━━━━━ 132s 211ms/step - accuracy: 0.7727 - loss: 1.3868 - val
_accuracy: 0.7253 - val_loss: 1.5605
Epoch 18/20
625/625 ━━━━━━━━━━━━━━━━━━━━ 132s 212ms/step - accuracy: 0.7754 - loss: 1.3752 - val
_accuracy: 0.7718 - val_loss: 1.3873
Epoch 19/20
625/625 ━━━━━━━━━━━━━━━━━━━━ 133s 213ms/step - accuracy: 0.7738 - loss: 1.3743 - val
```

```
_accuracy: 0.7513 - val_loss: 1.4568
Epoch 20/20
625/625 ━━━━━━━━━━━━━━━━━━━━ 133s 212ms/step - accuracy: 0.7752 - loss: 1.3686 - val
_accuracy: 0.7529 - val_loss: 1.4425
```



The model started with low accuracy (34.39%) but kept improving, reaching 77.52% on training data and 75.29% on validation data.

The loss went down, meaning the model learned well, but the ups and downs in validation accuracy show some overfitting.

To make it better, I have tried techniques like dropout, adjusting the learning rate, or adding regularization.

Creating a Function to Detect Overfitting by Comparing Training and Validation Accuracy

Creating a **function** below which checks if the model is overfitting by comparing how well it performs on training vs.

validation data.

It makes predictions on both training and validation sets.

Converts the predictions into actual class labels.

Compares predicted labels with real labels to find accuracy.

Prints both accuracies to see how well the model is learning.

If training accuracy is much higher than validation accuracy, the model might be overfitting, it means memorizing training

data instead of learning patterns.

In [56]:
```
def check_overfitting(model, x_train, y_train, x_val, y_val):
    train_acc = np.mean(np.argmax(model.predict(x_train), axis=1) == y_train.flatte
    val_acc = np.mean(np.argmax(model.predict(x_val), axis=1) == y_val.flatten())
```

```
    print(f"Train Accuracy: {train_acc:.4f}")
    print(f"Validation Accuracy: {val_acc:.4f}")

    return train_acc, val_acc
```

Adding **Comment** briefly on overfitting or underfitting based on the plots

In [59]:
```
def analyze_performance(train_acc, val_acc):
    if train_acc > val_acc + 0.1:
        print(" Possible Overfitting: Training accuracy is much higher than validat
    elif val_acc > train_acc:
        print(" Possible Underfitting: Validation accuracy is higher than training
    else:
        print(" Model is generalizing well without significant overfitting or under
```

In [61]:
```
train_acc, val_acc = check_overfitting(model, x_train, y_train, x_val, y_val)
analyze_performance(train_acc, val_acc)
```

```
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 24s 19ms/step
313/313 ━━━━━━━━━━━━━━━━━━━━ 7s 22ms/step
Train Accuracy: 0.8409
Validation Accuracy: 0.7773
 Model is generalizing well without significant overfitting or underfitting.
```

Task 3: Evaluate the Model

In [64]:
```
def evaluate_model(model, x_test, y_test):
    test_loss, test_acc = model.evaluate(x_test, y_test)
    print("Test Accuracy:", test_acc)
```

Generating Confusion Matrix and Classification Report

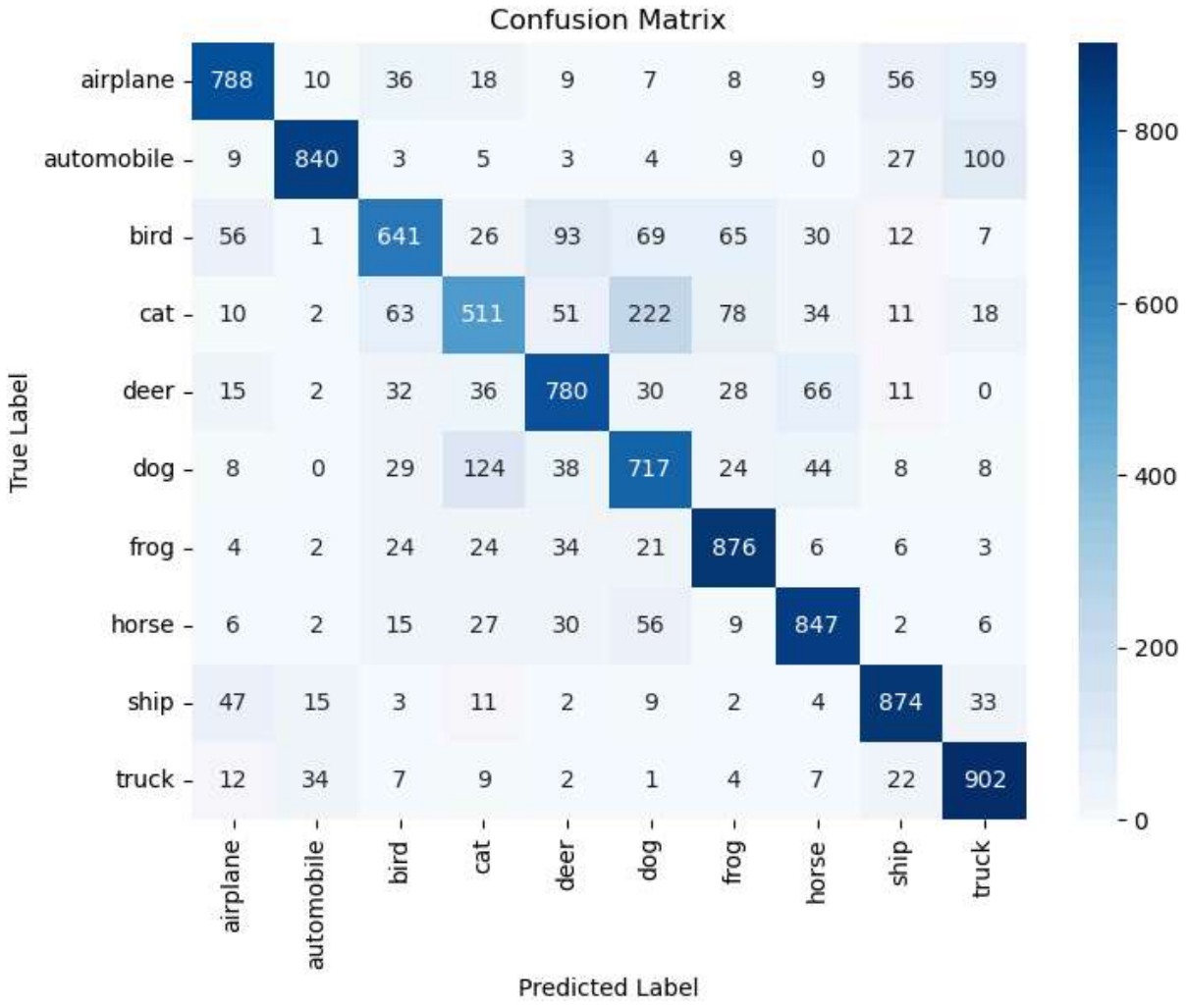
In [67]:
```
y_pred = model.predict(x_test)
y_pred_classes = np.argmax(y_pred, axis=1)
y_test_classes = y_test.flatten()
```

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 6s 18ms/step
```

In [69]:
```
cm = confusion_matrix(y_test_classes, y_pred_classes)
```

In [71]:
```
class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer',
               'dog', 'frog', 'horse', 'ship', 'truck']
```

In [73]:
```
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
            xticklabels=class_names, yticklabels=class_names)
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.title('Confusion Matrix')
plt.show()
```



In [75]:
```
print("Classification Report:\n", classification_report(y_test_classes, y_pred_clas
```

```
Classification Report:
              precision    recall  f1-score   support

    airplane       0.83      0.79      0.81      1000
  automobile       0.93      0.84      0.88      1000
        bird       0.75      0.64      0.69      1000
         cat       0.65      0.51      0.57      1000
        deer       0.75      0.78      0.76      1000
         dog       0.63      0.72      0.67      1000
        frog       0.79      0.88      0.83      1000
       horse       0.81      0.85      0.83      1000
        ship       0.85      0.87      0.86      1000
       truck       0.79      0.90      0.84      1000

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

Insights from Classification Report:

Accuracy: 78%

Examples of Correctly and Incorrectly Classified Images

Precision & Recall: The model struggles with Cats and Birds, as their F1-scores are low (~0.57-0.69), meaning they are often

misclassified.

Strong Performance: The model correctly identifies Ships, Automobiles, and Horses with high precision and recall (~0.85+),

showing confidence in these categories.

Dogs & Deer: The model shows imbalanced recall and precision, meaning it sometimes misidentifies them or confuses them with

similar classes.

Overall: The model performs fairly well but struggles with certain categories. Some tuning or better data augmentation might

help improve its performance.

In [79]:
```
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.utils import to_categorical
```

Here i am loading the CIFAR-10 dataset, preprocessing it, training the model, and then evaluating its

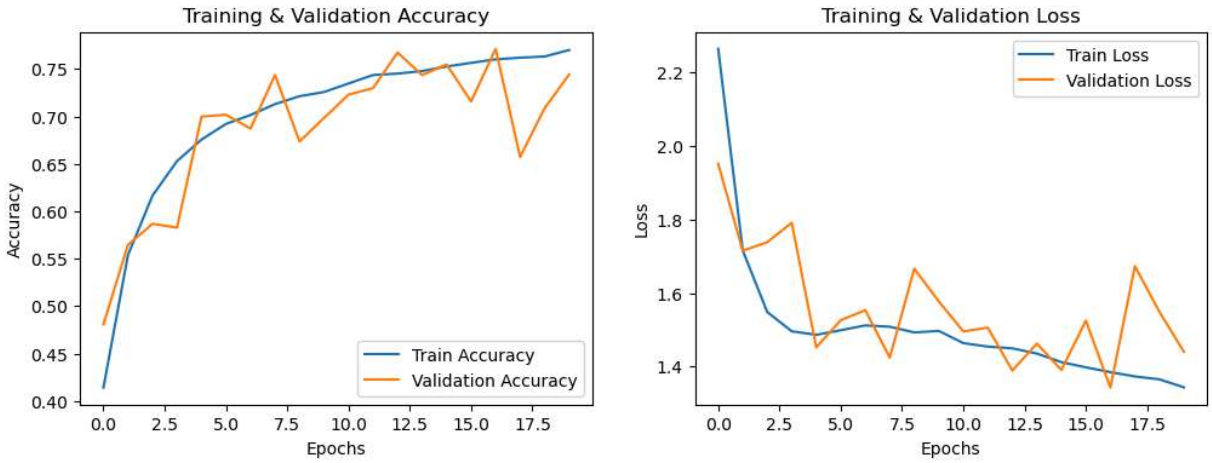
performance on test data.

In [81]:
```
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
x_train, y_train, x_val, y_val, x_test, y_test = preprocess_data(x_train, y_train,
model = build_model()
history = train_model(model, x_train, y_train, x_val, y_val)
evaluate_model(model, x_test, y_test)
```

```
C:\Users\msgme\anaconda3\Lib\site-packages\keras\src\layers\convolutional\base_conv.
py:107: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. W
hen using Sequential models, prefer using an `Input(shape)` object as the first laye
r in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

```
Epoch 1/20
704/704 ━━━━━━━━━━━━━━━━━━━━ 139s 190ms/step - accuracy: 0.3384 - loss: 2.8308 - val
_accuracy: 0.4808 - val_loss: 1.9515
Epoch 2/20
704/704 ━━━━━━━━━━━━━━━━━━━━ 136s 193ms/step - accuracy: 0.5389 - loss: 1.7730 - val
_accuracy: 0.5646 - val_loss: 1.7150
Epoch 3/20
704/704 ━━━━━━━━━━━━━━━━━━━━ 138s 196ms/step - accuracy: 0.6063 - loss: 1.5621 - val
_accuracy: 0.5868 - val_loss: 1.7382
Epoch 4/20
704/704 ━━━━━━━━━━━━━━━━━━━━ 142s 201ms/step - accuracy: 0.6493 - loss: 1.4935 - val
_accuracy: 0.5828 - val_loss: 1.7911
Epoch 5/20
704/704 ━━━━━━━━━━━━━━━━━━━━ 140s 199ms/step - accuracy: 0.6809 - loss: 1.4592 - val
_accuracy: 0.6998 - val_loss: 1.4525
Epoch 6/20
704/704 ━━━━━━━━━━━━━━━━━━━━ 142s 201ms/step - accuracy: 0.6930 - loss: 1.4859 - val
_accuracy: 0.7018 - val_loss: 1.5261
Epoch 7/20
704/704 ━━━━━━━━━━━━━━━━━━━━ 142s 202ms/step - accuracy: 0.7047 - loss: 1.4889 - val
_accuracy: 0.6872 - val_loss: 1.5535
Epoch 8/20
704/704 ━━━━━━━━━━━━━━━━━━━━ 141s 200ms/step - accuracy: 0.7174 - loss: 1.4857 - val
_accuracy: 0.7436 - val_loss: 1.4245
Epoch 9/20
704/704 ━━━━━━━━━━━━━━━━━━━━ 149s 212ms/step - accuracy: 0.7280 - loss: 1.4703 - val
_accuracy: 0.6736 - val_loss: 1.6661
Epoch 10/20
704/704 ━━━━━━━━━━━━━━━━━━━━ 169s 240ms/step - accuracy: 0.7306 - loss: 1.4713 - val
_accuracy: 0.6984 - val_loss: 1.5773
Epoch 11/20
704/704 ━━━━━━━━━━━━━━━━━━━━ 169s 239ms/step - accuracy: 0.7419 - loss: 1.4423 - val
_accuracy: 0.7230 - val_loss: 1.4958
Epoch 12/20
704/704 ━━━━━━━━━━━━━━━━━━━━ 168s 239ms/step - accuracy: 0.7438 - loss: 1.4500 - val
_accuracy: 0.7298 - val_loss: 1.5060
Epoch 13/20
704/704 ━━━━━━━━━━━━━━━━━━━━ 169s 240ms/step - accuracy: 0.7525 - loss: 1.4232 - val
_accuracy: 0.7674 - val_loss: 1.3891
Epoch 14/20
704/704 ━━━━━━━━━━━━━━━━━━━━ 168s 239ms/step - accuracy: 0.7543 - loss: 1.4192 - val
_accuracy: 0.7436 - val_loss: 1.4628
Epoch 15/20
704/704 ━━━━━━━━━━━━━━━━━━━━ 169s 240ms/step - accuracy: 0.7603 - loss: 1.3947 - val
_accuracy: 0.7548 - val_loss: 1.3909
Epoch 16/20
704/704 ━━━━━━━━━━━━━━━━━━━━ 170s 241ms/step - accuracy: 0.7605 - loss: 1.3786 - val
_accuracy: 0.7158 - val_loss: 1.5251
Epoch 17/20
704/704 ━━━━━━━━━━━━━━━━━━━━ 171s 243ms/step - accuracy: 0.7643 - loss: 1.3714 - val
_accuracy: 0.7712 - val_loss: 1.3430
Epoch 18/20
704/704 ━━━━━━━━━━━━━━━━━━━━ 172s 244ms/step - accuracy: 0.7683 - loss: 1.3534 - val
_accuracy: 0.6572 - val_loss: 1.6730
Epoch 19/20
704/704 ━━━━━━━━━━━━━━━━━━━━ 171s 242ms/step - accuracy: 0.7674 - loss: 1.3517 - val
```

```
_accuracy: 0.7088 - val_loss: 1.5491
Epoch 20/20
704/704 ━━━━━━━━━━━━━━━━━━━━ 169s 241ms/step - accuracy: 0.7745 - loss: 1.3260 - val
_accuracy: 0.7442 - val_loss: 1.4402
```




```
313/313 ━━━━━━━━━━━━━━━━━━━━ 10s 32ms/step - accuracy: 0.7715 - loss: 1.3624
Test Accuracy: 0.7670000195503235
```

Model learned well and got 77.45% accuracy on training data, 74.42% on validation data, and 76.7% on test data.

This means it's doing a good job but slightly overfits. The training loss kept decreasing, but the validation loss

went up and down, showing the model struggled a bit with new data. To improve, The model can improve by using try

Dropout, L2 regularization, or data augmentation to make it more stable. Also, Improving settings like the learning

rate or optimizer might boost accuracy!

Task 4: Experimentation with Model Improvements.

Comparing Different Optimizers for Model Performance

In [86]:
```
def experiment_with_optimizers():
    optimizers_list = ['adam', 'sgd', 'rmsprop']
    results={}
    for opt in optimizers_list:
        print(f"\nTraining with {opt} optimizer:\n")
        model = build_model()
        model.compile(optimizer=opt, loss='sparse_categorical_crossentropy', metric
        history = train_model(model, x_train, y_train, x_val, y_val)
        test_loss, test_acc = model.evaluate(x_test, y_test)
        results[opt]=test_acc
    print("\nPerformance Comparison:")
    for opt, acc in results.items():
        print(f"{opt}: Test Accuracy = {acc:.4f}")
    return results
```

Creating a function to tests three different optimizers (Adam, SGD, and RMSprop) to see which one gives the best accuracy.

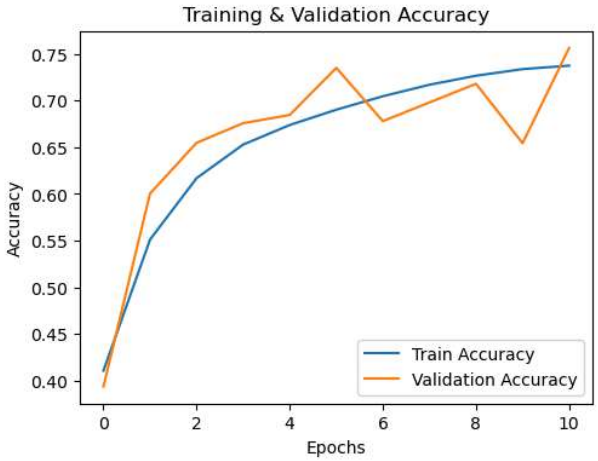
It will train a model with each optimizer, evaluate its performance, and compare the test accuracy of all three. It will

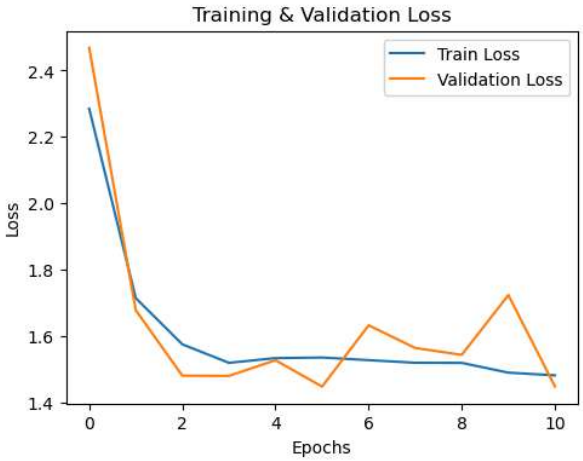
help in choosing the best optimizer for better model performance.

Comparing Optimizers Adam vs. SGD vs. RMSprop to Find the Best One!

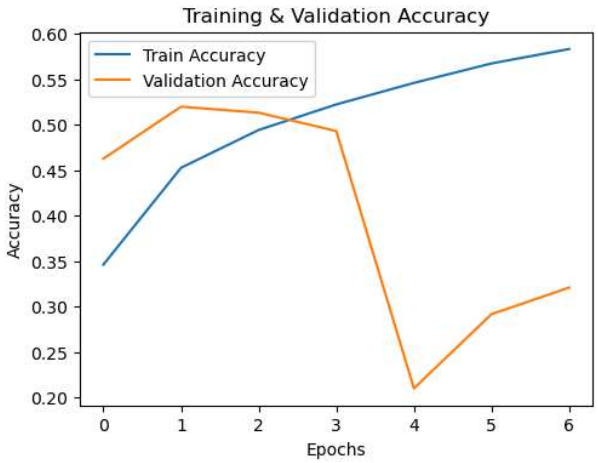
In [91]:
```
results = experiment_with_optimizers()
```

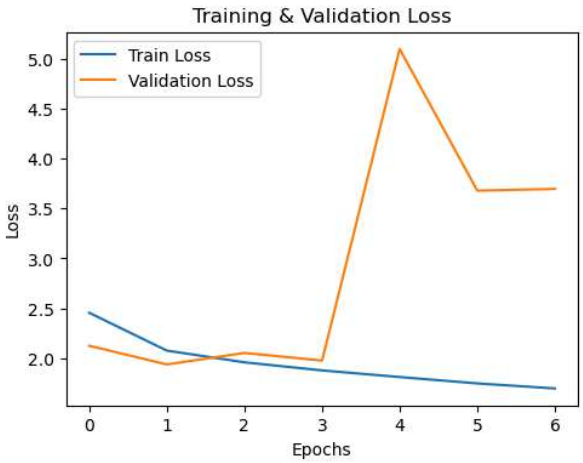
```
Training with adam optimizer:

Epoch 1/20
704/704 ━━━━━━━━━━━━━━━━━━━━ 183s 244ms/step - accuracy: 0.3334 - loss: 2.9400 - val
_accuracy: 0.3938 - val_loss: 2.4682
Epoch 2/20
704/704 ━━━━━━━━━━━━━━━━━━━━ 171s 243ms/step - accuracy: 0.5285 - loss: 1.7781 - val
_accuracy: 0.6004 - val_loss: 1.6773
Epoch 3/20
704/704 ━━━━━━━━━━━━━━━━━━━━ 172s 244ms/step - accuracy: 0.6087 - loss: 1.5871 - val
_accuracy: 0.6548 - val_loss: 1.4800
Epoch 4/20
704/704 ━━━━━━━━━━━━━━━━━━━━ 172s 244ms/step - accuracy: 0.6507 - loss: 1.5186 - val
_accuracy: 0.6758 - val_loss: 1.4793
Epoch 5/20
704/704 ━━━━━━━━━━━━━━━━━━━━ 171s 243ms/step - accuracy: 0.6721 - loss: 1.5132 - val
_accuracy: 0.6846 - val_loss: 1.5266
Epoch 6/20
704/704 ━━━━━━━━━━━━━━━━━━━━ 171s 242ms/step - accuracy: 0.6945 - loss: 1.5114 - val
_accuracy: 0.7350 - val_loss: 1.4470
Epoch 7/20
704/704 ━━━━━━━━━━━━━━━━━━━━ 170s 242ms/step - accuracy: 0.7101 - loss: 1.4950 - val
_accuracy: 0.6780 - val_loss: 1.6318
Epoch 8/20
704/704 ━━━━━━━━━━━━━━━━━━━━ 171s 242ms/step - accuracy: 0.7206 - loss: 1.5034 - val
_accuracy: 0.6980 - val_loss: 1.5634
Epoch 9/20
704/704 ━━━━━━━━━━━━━━━━━━━━ 171s 243ms/step - accuracy: 0.7271 - loss: 1.5082 - val
_accuracy: 0.7180 - val_loss: 1.5428
Epoch 10/20
704/704 ━━━━━━━━━━━━━━━━━━━━ 171s 243ms/step - accuracy: 0.7377 - loss: 1.4793 - val
_accuracy: 0.6544 - val_loss: 1.7231
Epoch 11/20
704/704 ━━━━━━━━━━━━━━━━━━━━ 171s 243ms/step - accuracy: 0.7433 - loss: 1.4600 - val
_accuracy: 0.7564 - val_loss: 1.4474
```



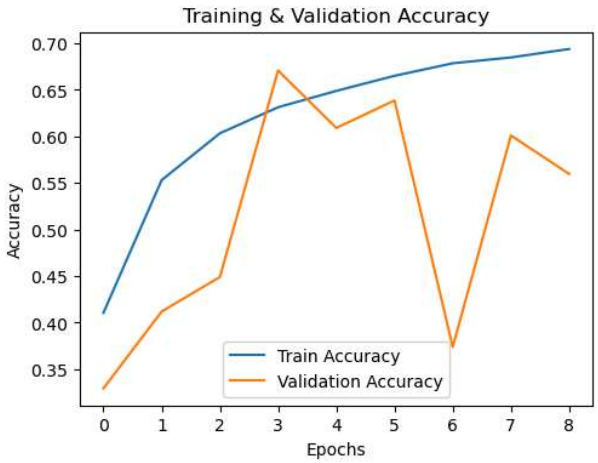


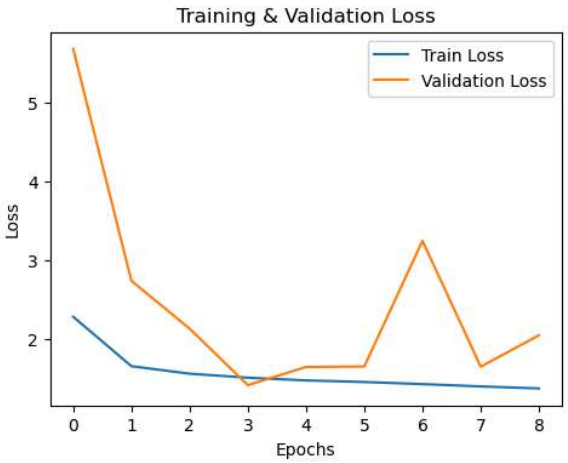

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 10s 31ms/step - accuracy: 0.7284 - loss: 1.4646

Training with sgd optimizer:

Epoch 1/20
704/704 ━━━━━━━━━━━━━━━━━━━━ 157s 218ms/step - accuracy: 0.2879 - loss: 2.8484 - val
_accuracy: 0.4628 - val_loss: 2.1239
Epoch 2/20
704/704 ━━━━━━━━━━━━━━━━━━━━ 126s 178ms/step - accuracy: 0.4413 - loss: 2.1069 - val
_accuracy: 0.5200 - val_loss: 1.9368
Epoch 3/20
704/704 ━━━━━━━━━━━━━━━━━━━━ 128s 182ms/step - accuracy: 0.4894 - loss: 1.9705 - val
_accuracy: 0.5134 - val_loss: 2.0517
Epoch 4/20
704/704 ━━━━━━━━━━━━━━━━━━━━ 128s 182ms/step - accuracy: 0.5190 - loss: 1.8908 - val
_accuracy: 0.4932 - val_loss: 1.9756
Epoch 5/20
704/704 ━━━━━━━━━━━━━━━━━━━━ 129s 184ms/step - accuracy: 0.5399 - loss: 1.8217 - val
_accuracy: 0.2098 - val_loss: 5.1013
Epoch 6/20
704/704 ━━━━━━━━━━━━━━━━━━━━ 128s 182ms/step - accuracy: 0.5646 - loss: 1.7521 - val
_accuracy: 0.2916 - val_loss: 3.6802
Epoch 7/20
704/704 ━━━━━━━━━━━━━━━━━━━━ 127s 181ms/step - accuracy: 0.5800 - loss: 1.7066 - val
_accuracy: 0.3208 - val_loss: 3.6975
```





```
313/313 ━━━━━━━━━━━━━━━━━━━━ 7s 22ms/step - accuracy: 0.5118 - loss: 1.9487

Training with rmsprop optimizer:

Epoch 1/20
704/704 ━━━━━━━━━━━━━━━━━━━━ 137s 189ms/step - accuracy: 0.3354 - loss: 3.2163 - val
_accuracy: 0.3294 - val_loss: 5.6820
Epoch 2/20
704/704 ━━━━━━━━━━━━━━━━━━━━ 136s 193ms/step - accuracy: 0.5366 - loss: 1.6880 - val
_accuracy: 0.4120 - val_loss: 2.7398
Epoch 3/20
704/704 ━━━━━━━━━━━━━━━━━━━━ 136s 194ms/step - accuracy: 0.5965 - loss: 1.5672 - val
_accuracy: 0.4490 - val_loss: 2.1329
Epoch 4/20
704/704 ━━━━━━━━━━━━━━━━━━━━ 137s 194ms/step - accuracy: 0.6286 - loss: 1.5126 - val
_accuracy: 0.6708 - val_loss: 1.4158
Epoch 5/20
704/704 ━━━━━━━━━━━━━━━━━━━━ 138s 197ms/step - accuracy: 0.6448 - loss: 1.4682 - val
_accuracy: 0.6090 - val_loss: 1.6472
Epoch 6/20
704/704 ━━━━━━━━━━━━━━━━━━━━ 139s 197ms/step - accuracy: 0.6654 - loss: 1.4482 - val
_accuracy: 0.6386 - val_loss: 1.6529
Epoch 7/20
704/704 ━━━━━━━━━━━━━━━━━━━━ 138s 196ms/step - accuracy: 0.6806 - loss: 1.4246 - val
_accuracy: 0.3740 - val_loss: 3.2477
Epoch 8/20
704/704 ━━━━━━━━━━━━━━━━━━━━ 141s 200ms/step - accuracy: 0.6874 - loss: 1.3863 - val
_accuracy: 0.6010 - val_loss: 1.6501
Epoch 9/20
704/704 ━━━━━━━━━━━━━━━━━━━━ 139s 197ms/step - accuracy: 0.6920 - loss: 1.3703 - val
_accuracy: 0.5596 - val_loss: 2.0491
```






```
313/313 ━━━━━━━━━━━━━━━━━━━━ 8s 25ms/step - accuracy: 0.6700 - loss: 1.4248

Performance Comparison:
adam: Test Accuracy = 0.7271
sgd: Test Accuracy = 0.5046
rmsprop: Test Accuracy = 0.6678
```

Displaying Test Accuracy for Each Optimizer!

In [99]:
```
for opt, acc in results.items():
    print(f"{opt}: Test Accuracy = {acc:.4f}")
```

```
adam: Test Accuracy = 0.7271
sgd: Test Accuracy = 0.5046
rmsprop: Test Accuracy = 0.6678
```

I tested three optimizers—Adam, SGD, and RMSprop—to find the best one. **Adam performed** the best with **72.71%** accuracy,

while **RMSprop** with **66.78%**, and **SGD** struggled at **50.46%**. This means Adam helped the model learn better, but

SGD didn't perform well. **The result indicates that choosing the right optimizer is important for getting good accuracy!**

In [ ]: