Image Classification with CIFAR-10 Dataset and Scikit-Learn

In this notebook, we will explore the basics of image classification using the CIFAR-10 dataset and build a simple image classifier using Scikit-Learn. We will perform the following steps:

1. Load and preprocess the CIFAR-10 dataset.
2. Extract features from the images.
3. Train a machine learning model on the extracted features.
4. Evaluate the model's performance.

Step 1: Installing and Importing Libraries

Installing Libraries

Before we start, we need to ensure that we have all the necessary libraries installed. We can use `!pip install` to install any missing libraries directly from the Jupyter Notebook. This command is useful for installing Python packages from the Python Package Index (PyPI).

Why Install Libraries?

Libraries provide pre-written code that we can use to perform various tasks without having to write everything from scratch. Installing libraries ensures we have access to the necessary functions and tools needed for our project.

Here are the libraries we need for this notebook:

- **numpy:** For numerical operations.
- **matplotlib:** For plotting and visualizing images.
- **tensorflow:** For loading the CIFAR-10 dataset.
- **scikit-learn (sklearn):** For machine learning models and evaluation metrics.

Installing Libraries Individually vs. All at Once

You can install each library separately by writing a `!pip install` command for each library. This looks like:

```
!pip install numpy
!pip install matplotlib
!pip install tensorflow
!pip install scikit-learn


Alternatively, you can install all the libraries together in a single !pip install command by separating the library n

!pip install numpy matplotlib tensorflow scikit-learn
```

Differences and Considerations

- **Convenience**: Installing all libraries at once is more convenient and requires fewer lines of code. It can save time when writing and running the notebook.
- **Execution Time**: Running a single !pip install command can be faster than running multiple commands, as it reduces the overhead of initiating separate installation processes for each library.
- **Dependency Management**: Installing libraries together can help Pip resolve dependencies more efficiently, potentially avoiding conflicts that might arise when installing libraries separately.
- **Debugging**: Installing libraries individually can make it easier to identify which specific library caused an issue if an installation error occurs. However, this is generally only a concern if you encounter frequent installation problems.

For this notebook, we will install all necessary libraries together for convenience.

```
!pip install numpy matplotlib tensorflow scikit-learn
```

```
Requirement already satisfied: numpy in /usr/local/lib/python3.12/dist-packages (2.0.2)
Requirement already satisfied: matplotlib in /usr/local/lib/python3.12/dist-packages (3.10.0)
Requirement already satisfied: tensorflow in /usr/local/lib/python3.12/dist-packages (2.19.0)
Requirement already satisfied: scikit-learn in /usr/local/lib/python3.12/dist-packages (1.6.1)
Requirement already satisfied: contourpy>=1.0.1 in /usr/local/lib/python3.12/dist-packages (from matplotlib) (1.3.3)
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.12/dist-packages (from matplotlib) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in /usr/local/lib/python3.12/dist-packages (from matplotlib) (4.59.2)
Requirement already satisfied: kiwisolver>=1.3.1 in /usr/local/lib/python3.12/dist-packages (from matplotlib) (1.4.9)
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.12/dist-packages (from matplotlib) (25.0)
Requirement already satisfied: pillow>=8 in /usr/local/lib/python3.12/dist-packages (from matplotlib) (11.3.0)
Requirement already satisfied: pyparsing>=2.3.1 in /usr/local/lib/python3.12/dist-packages (from matplotlib) (3.2.3)
Requirement already satisfied: python-dateutil>=2.7 in /usr/local/lib/python3.12/dist-packages (from matplotlib) (2.9.0.post0)
Requirement already satisfied: absl-py>=1.0.0 in /usr/local/lib/python3.12/dist-packages (from tensorflow) (1.4.0)
Requirement already satisfied: astunparse>=1.6.0 in /usr/local/lib/python3.12/dist-packages (from tensorflow) (1.6.3)
```

```
Requirement already satisfied: flatbuffers>=24.3.25 in /usr/local/lib/python3.12/dist-packages (from tensorflow) (25.2.10)
Requirement already satisfied: gast!=0.5.0,!=0.5.1,!=0.5.2,>=0.2.1 in /usr/local/lib/python3.12/dist-packages (from tensorflow) (0.6
Requirement already satisfied: google-pasta>=0.1.1 in /usr/local/lib/python3.12/dist-packages (from tensorflow) (0.2.0)
Requirement already satisfied: libclang>=13.0.0 in /usr/local/lib/python3.12/dist-packages (from tensorflow) (18.1.1)
Requirement already satisfied: opt-einsum>=2.3.2 in /usr/local/lib/python3.12/dist-packages (from tensorflow) (3.4.0)
Requirement already satisfied: protobuf!=4.21.0,!=4.21.1,!=4.21.2,!=4.21.3,!=4.21.4,!=4.21.5,<6.0.0dev,>=3.20.3 in /usr/local/lib/py
Requirement already satisfied: requests<3,>=2.21.0 in /usr/local/lib/python3.12/dist-packages (from tensorflow) (2.32.4)
Requirement already satisfied: setuptools in /usr/local/lib/python3.12/dist-packages (from tensorflow) (75.2.0)
Requirement already satisfied: six>=1.12.0 in /usr/local/lib/python3.12/dist-packages (from tensorflow) (1.17.0)
Requirement already satisfied: termcolor>=1.1.0 in /usr/local/lib/python3.12/dist-packages (from tensorflow) (3.1.0)
Requirement already satisfied: typing-extensions>=3.6.6 in /usr/local/lib/python3.12/dist-packages (from tensorflow) (4.15.0)
Requirement already satisfied: wrapt>=1.11.0 in /usr/local/lib/python3.12/dist-packages (from tensorflow) (1.17.3)
Requirement already satisfied: grpcio<2.0,>=1.24.3 in /usr/local/lib/python3.12/dist-packages (from tensorflow) (1.74.0)
Requirement already satisfied: tensorboard~=2.19.0 in /usr/local/lib/python3.12/dist-packages (from tensorflow) (2.19.0)
Requirement already satisfied: keras>=3.5.0 in /usr/local/lib/python3.12/dist-packages (from tensorflow) (3.10.0)
Requirement already satisfied: h5py>=3.11.0 in /usr/local/lib/python3.12/dist-packages (from tensorflow) (3.14.0)
Requirement already satisfied: ml-dtypes<1.0.0,>=0.5.1 in /usr/local/lib/python3.12/dist-packages (from tensorflow) (0.5.3)
Requirement already satisfied: scipy>=1.6.0 in /usr/local/lib/python3.12/dist-packages (from scikit-learn) (1.16.1)
Requirement already satisfied: joblib>=1.2.0 in /usr/local/lib/python3.12/dist-packages (from scikit-learn) (1.5.2)
Requirement already satisfied: threadpoolctl>=3.1.0 in /usr/local/lib/python3.12/dist-packages (from scikit-learn) (3.6.0)
Requirement already satisfied: wheel<1.0,>=0.23.0 in /usr/local/lib/python3.12/dist-packages (from astunparse>=1.6.0->tensorflow) (0
Requirement already satisfied: rich in /usr/local/lib/python3.12/dist-packages (from keras>=3.5.0->tensorflow) (13.9.4)
Requirement already satisfied: namex in /usr/local/lib/python3.12/dist-packages (from keras>=3.5.0->tensorflow) (0.1.0)
Requirement already satisfied: optree in /usr/local/lib/python3.12/dist-packages (from keras>=3.5.0->tensorflow) (0.17.0)
Requirement already satisfied: charset_normalizer<4,>=2 in /usr/local/lib/python3.12/dist-packages (from requests<3,>=2.21.0->tensor
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.12/dist-packages (from requests<3,>=2.21.0->tensorflow) (3.10)
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.12/dist-packages (from requests<3,>=2.21.0->tensorflow)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.12/dist-packages (from requests<3,>=2.21.0->tensorflow)
Requirement already satisfied: markdown>=2.6.8 in /usr/local/lib/python3.12/dist-packages (from tensorboard~=2.19.0->tensorflow) (3.
Requirement already satisfied: tensorboard-data-server<0.8.0,>=0.7.0 in /usr/local/lib/python3.12/dist-packages (from tensorboard~=2
Requirement already satisfied: werkzeug>=1.0.1 in /usr/local/lib/python3.12/dist-packages (from tensorboard~=2.19.0->tensorflow) (3.
Requirement already satisfied: MarkupSafe>=2.1.1 in /usr/local/lib/python3.12/dist-packages (from werkzeug>=1.0.1->tensorboard~=2.19
Requirement already satisfied: markdown-it-py>=2.2.0 in /usr/local/lib/python3.12/dist-packages (from rich->keras>=3.5.0->tensorflow
Requirement already satisfied: pygments<3.0.0,>=2.13.0 in /usr/local/lib/python3.12/dist-packages (from rich->keras>=3.5.0->tensorfl
Requirement already satisfied: mdurl~=0.1 in /usr/local/lib/python3.12/dist-packages (from markdown-it-py>=2.2.0->rich->keras>=3.5.0
```

Importing Libraries

After installing the libraries, we need to import them into our notebook. Importing libraries allows us to use their functionalities in our code. Each library is imported using an alias (short name) to make the code cleaner and more readable.

Let's import the required libraries:

```
# Importing necessary libraries
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.datasets import cifar10
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.metrics import classification_report, accuracy_score
```

Step 2: Loading and Preprocessing the CIFAR-10 Dataset

The CIFAR-10 dataset is readily available in the `keras` library. We will load the dataset and preprocess it by converting the images to grayscale and flattening them.

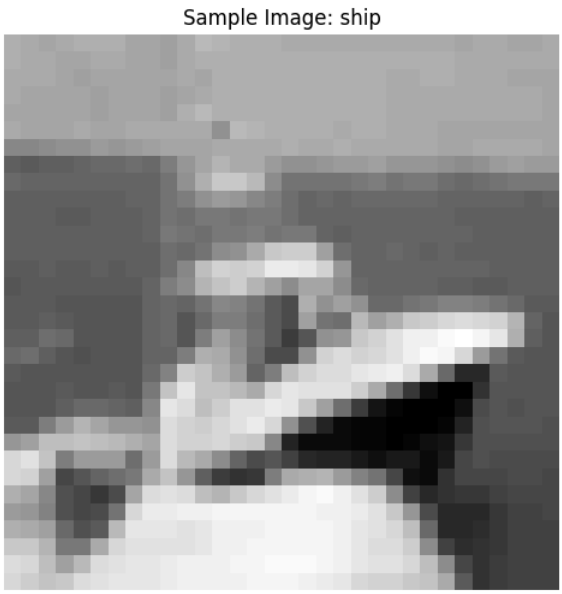
```
# Load CIFAR-10 dataset
(X_train, y_train), (X_test, y_test) = cifar10.load_data()

# To minimize computational demands lets work with three classes of your choice

# CIFAR-10 classes
class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer',
               'dog', 'frog', 'horse', 'ship', 'truck']


# Choose a subset of classes
chosen_classes = ['cat', 'dog', 'ship']
class_indices = [class_names.index(cls) for cls in chosen_classes]
```

```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
170498071/170498071 ━━━━━━━━━━━━━━━━━━━━ 3s 0us/step
```

```
# Filter data for the chosen classes
mask_train = np.isin(y_train, class_indices)
mask_test = np.isin(y_test, class_indices)
X_train_subset = X_train[mask_train.flatten()]
y_train_subset = y_train[mask_train]
X_test_subset = X_test[mask_test.flatten()]
y_test_subset = y_test[mask_test]
```

```
# Convert images to grayscale
X_train_gray = np.dot(X_train_subset[...,:3], [0.2989, 0.5870, 0.1140])
X_test_gray = np.dot(X_test_subset[...,:3], [0.2989, 0.5870, 0.1140])

# Normalize the images
X_train_normalized = X_train_gray / 255.0
X_test_normalized = X_test_gray / 255.0

# Flatten the images
X_train_flat = X_train_normalized.reshape(X_train_normalized.shape[0], -1)
X_test_flat = X_test_normalized.reshape(X_test_normalized.shape[0], -1)
```

```
# Display a sample image
plt.figure(figsize=(6, 6))
plt.imshow(X_train_gray[0], cmap='gray')
plt.title(f'Sample Image: {chosen_classes[np.where(class_indices == y_train_subset[0])[0][0]]}')
plt.axis('off')
plt.show()

print("Training set size:", X_train_flat.shape)
print("Testing set size:", X_test_flat.shape)
```




```
Training set size: (15000, 1024)
Testing set size: (3000, 1024)
```

Step 3: Training a Machine Learning Model

What is SVM (Support Vector Machine)?

Support Vector Machine (SVM) is a supervised machine learning algorithm that can be used for both classification and regression tasks. However, it is mostly used for classification problems. The objective of the SVM algorithm is to find a hyperplane in an N-dimensional space (N is the number of features) that distinctly classifies the data points.

Key Concepts:

- **Hyperplane:** A decision boundary that separates different classes in the feature space. In 2D, it's a line; in 3D, it's a plane.
- **Support Vectors:** Data points that are closest to the hyperplane and influence its position and orientation. These points help in maximizing the margin of the classifier.
- **Margin:** The distance between the hyperplane and the closest data points from either class. SVM aims to maximize this margin.

Why Use SVM?

- **Effective in high-dimensional spaces:** SVM is very effective when the number of features is large.
- **Memory efficient:** It uses a subset of training points (support vectors) in the decision function, making it memory efficient.
- **Versatile:** Different kernel functions can be specified for the decision function. Common kernels include linear, polynomial, and radial basis function (RBF).

What does `SVC(kernel='linear')` mean?

`SVC` stands for Support Vector Classification, which is a class in the Scikit-Learn library used to implement the SVM algorithm for classification tasks. The `kernel` parameter in the `SVC` class specifies the type of hyperplane used to separate the data.

Kernel Types:

- **Linear Kernel:** The data is linearly separable (i.e., a straight line or hyperplane can separate the data). This is the simplest kernel.
  - When we use `SVC(kernel='linear')`, it means we are using a linear kernel for our SVM. This kernel is appropriate when the data can be separated by a straight line (or hyperplane in higher dimensions).
- **Polynomial Kernel:** The data is not linearly separable, but a polynomial function of the input features can separate the data.
- **Radial Basis Function (RBF) Kernel:** The data is not linearly separable, but mapping the data into a higher-dimensional space using a Gaussian (RBF) function can separate the data.

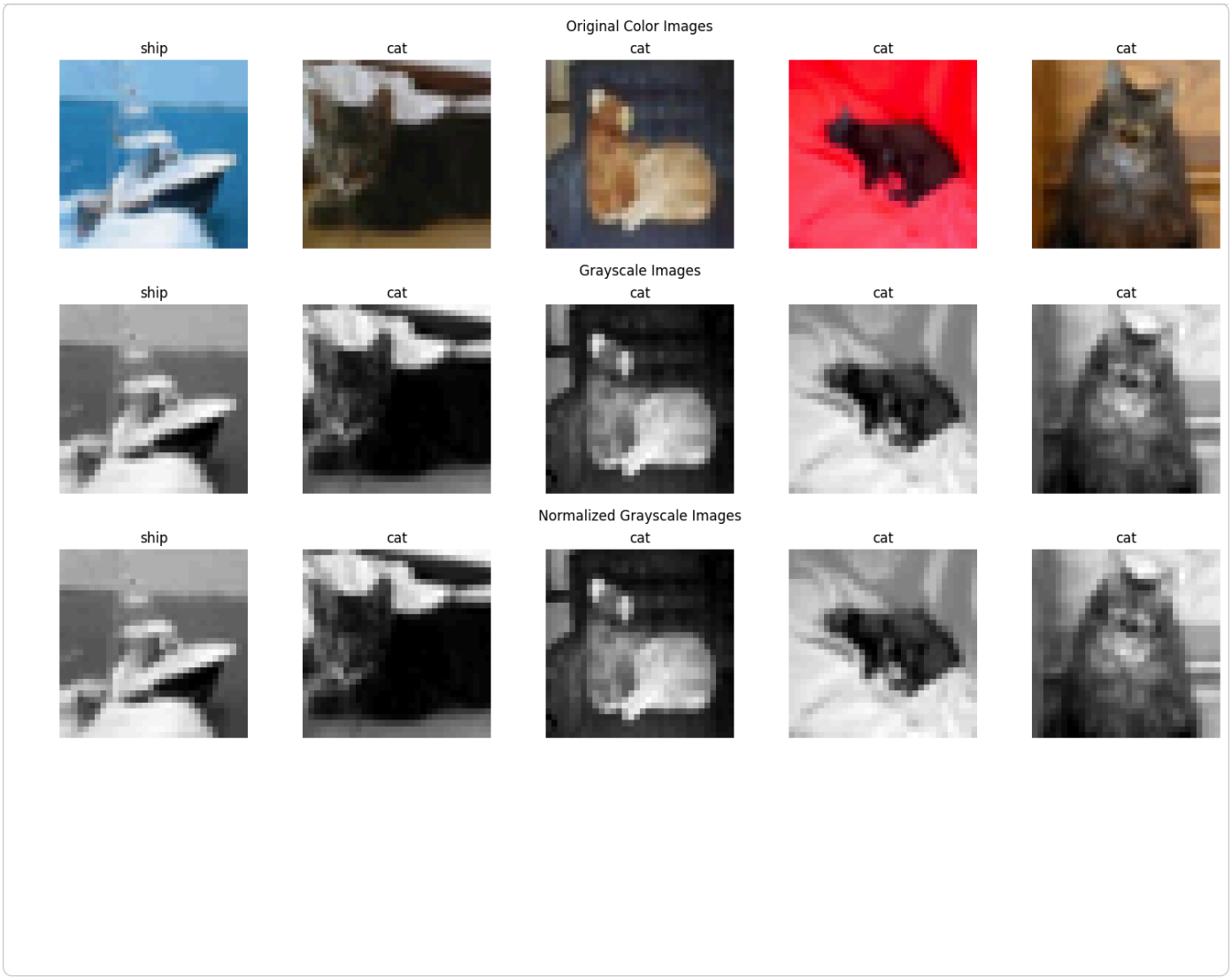
Training the SVM Model

We will use a Support Vector Machine (SVM) classifier from Scikit-Learn to train our model on the extracted features.

```
# Train an SVM classifier
model = SVC(kernel='linear')
model.fit(X_train_flat, y_train_subset.ravel())

# Predict on the test set
y_pred = model.predict(X_test_flat)

# Evaluate the model
print("Accuracy:", accuracy_score(y_test_subset, y_pred))
print("Classification Report:\n", classification_report(y_test_subset, y_pred, target_names=chosen_classes))
```

```
Accuracy: 0.547
Classification Report:
              precision    recall  f1-score   support

         cat       0.48      0.48      0.48      1000
         dog       0.49      0.48      0.49      1000
        ship       0.66      0.68      0.67      1000

    accuracy                           0.55      3000
   macro avg       0.54      0.55      0.55      3000
weighted avg       0.54      0.55      0.55      3000
```

```
# Lets see some images of the  datasset in the different stages
# Function to display images
def display_images(images, titles, main_title, cmap=None):
    fig, axes = plt.subplots(1, 5, figsize=(15, 3))
    fig.suptitle(main_title)
    for i, ax in enumerate(axes):
        if cmap:
            ax.imshow(images[i], cmap=cmap)
        else:
            ax.imshow(images[i])
        ax.set_title(titles[i])
        ax.axis('off')
    plt.tight_layout()
    plt.show()

# Display original color images
display_images(X_train_subset[:5],
               [chosen_classes[np.where(class_indices == y)[0][0]] for y in y_train_subset[:5]],
               'Original Color Images')

# Display grayscale images
display_images(X_train_gray[:5],
               [chosen_classes[np.where(class_indices == y)[0][0]] for y in y_train_subset[:5]],
               'Grayscale Images', cmap='gray')

# Display normalized images
display_images(X_train_normalized[:5],
               [chosen_classes[np.where(class_indices == y)[0][0]] for y in y_train_subset[:5]],
               'Normalized Grayscale Images', cmap='gray')
```



Step 4: Conclusion

In this notebook, we:

1. Loaded and preprocessed the CIFAR-10 dataset.
2. Converted the images to grayscale and flattened them to use as features.
3. Trained an SVM classifier on the extracted features.
4. Evaluated the model's performance.

Summary of SVM

Support Vector Machines (SVM) are a powerful tool for classification tasks. They work by finding the optimal hyperplane that maximizes the margin between different classes. The key points include:

- **Hyperplane:** The decision boundary.
- **Support Vectors:** Critical data points that define the hyperplane.
- **Margin:** The gap between the hyperplane and the nearest data points from any class.

SVMs are effective in high-dimensional spaces and are versatile due to the use of different kernel functions. However, they can be computationally intensive for large datasets and less effective for overlapping classes.

This exercise provided a basic introduction to image classification using classical machine learning techniques. Next few modules will explore advanced applications, using deep learning models like Convolutional Neural Networks (CNNs) using libraries such as TensorFlow or PyTorch.

Reflective Journal: L03 B Image Classification with CIFAR-10 Dataset and Scikit-Learn

This journal reflects on the steps taken and outcomes seen during the image classification notebook using the CIFAR-10 dataset and Scikit-Learn.

Step 1: Installing and Importing Libraries

Setting up the environment was the first step I took. This involved running the cell to install necessary Python libraries like `numpy`, `matplotlib`, `tensorflow`, and `scikit-learn`. The `!pip install` command was used right in the notebook. I understand that installing libraries is key because they provide ready-made code I can use, saving a lot of time and effort. The notebook provided an option to install all libraries together in one command (`!pip install numpy matplotlib tensorflow scikit-learn`), which I executed. This seemed more convenient and could help with managing dependencies compared to installing each one separately. After installation, I ran the cell to import the libraries into the notebook using standard import statements, giving them short names like `np` for numpy and `plt` for matplotlib. I see how this makes the code cleaner and easier to read. When I ran the installation cell, the output showed that the libraries were already in the environment. This saved time and confirmed I was ready to go.

Step 2: Loading and Preprocessing the CIFAR-10 Dataset

Next, I ran the cells to get the image data ready for the machine learning model. The CIFAR-10 dataset, a common benchmark for image classification, was loaded directly using the `tensorflow.keras.datasets` module. I observed that this loaded the data and split it into training and testing sets. The original dataset has 10 classes, but the notebook was set up to pick a smaller subset of three classes ('cat', 'dog', 'ship') to keep things manageable for this first try. The code found the index for each chosen class and then filtered the training and testing data to include only images from these classes. The images, originally in color (RGB), were converted to grayscale. I saw that this was done by applying a weighted sum to the color channels which the markdown explained is a standard way to convert to grayscale that considers how humans see color intensity. Then the grayscale images were normalized. I understand this means scaling the pixel values from the original 0-255 range to be between 0 and 1. Normalization appears important because it helps many machine learning algorithms work better by making sure all features are on a similar scale. Finally, I ran the cell that flattened the 2D image data (height by width) into a single 1D array for each image. I was shocked that this flattening is needed because the chosen traditional machine learning model SVM expects a 1D list of features as input. I also ran the cell that displayed a sample grayscale image to see how the conversion and normalization looked.

Running the dataset loading cell showed the download progress confirming it was retrieved successfully. The sample image display clearly showed a grayscale 'ship', visually confirming the grayscale conversion worked as expected. The printed output confirmed the size of the flattened training set (15,000 samples each with 1024 features) and testing set (3,000 samples each with 1024 features). This tells me I have 15,000 training images and 3,000 testing images each represented by a vector of 1024 pixels.

Step 3: Training a Machine Learning Model

With the data ready and flattened, I proceeded to train a machine learning model for classification by running the relevant cells. The notebook used a Support Vector Machine (SVM) classifier from Scikit-Learn specifically using the `SVC` class. For this first attempt a linear kernel (`SVC(kernel='linear')`) was selected. The markdown explained what SVM is including hyperplanes support vectors and margins and why a linear kernel is suitable for data that can be separated by a straight line (or hyperplane). I ran the cell to train the model using the `fit()` method on the flattened training data and their labels. I noted that the `.ravel()` part made sure the labels were in the right 1D format for the `fit()` method. After training I ran the cell that used the `predict()` method to make predictions on the testing data. I then ran the cell that checked how well the model did using `accuracy_score` and `classification_report` from `sklearn.metrics`. The accuracy score gave me the overall percentage of correct predictions. The classification report gave me more detailed metrics like precision recall and f1-score for each class. I also ran the cell that included code to show sample images at different stages (original color grayscale normalized grayscale) using a helper function. This helped me visually compare the data transformations.

The output from running the evaluation cells showed an accuracy score of about 0.55 on the test set. This means the model correctly classified around 55% of the test images. The classification report gave me more detail. Precision recall and f1-score for 'cat' and 'dog' were around 0.48-0.49 while 'ship' was higher at 0.66-0.68. This suggests the model was better at classifying ships than cats and dogs. The overall macro and weighted averages reflected this performance too. The displayed images visually confirmed the preprocessing steps showing the change from color to grayscale and then to normalized grayscale. The notebook displayed 5 original color images, 5 grayscale images, and 5 normalized grayscale images. Looking at the original color images, I could see the vibrant colors of the 'ship' and the distinct fur patterns of the 'cats'. The grayscale images showed the loss of color information but retained the structural details. The normalized grayscale images looked similar to the grayscale ones but with pixel values scaled between 0 and 1.

Step 4: Conclusion

This final section summarized everything and gave a quick overview of SVM. The markdown highlighted the main steps: loading and preprocessing the data (grayscale conversion and flattening) training the SVM classifier and evaluating its performance. The SVM summary repeated the core ideas: hyperplanes support vectors and margins explaining how SVM finds the best dividing line. It also covered the benefits of SVM like working well in high-dimensional spaces and being memory efficient because it uses support vectors. I also noted considerations like SVM being computationally intensive for large datasets and not as effective with classes that overlap a lot. The conclusion also pointed to future steps mentioning deep learning models like CNNs using libraries like TensorFlow or PyTorch as ways to potentially get better results on complex image data.

Potential Reflective Questions (What if):

1. **What if I needed specific versions of the toolkits?** If I needed exact versions I'd have to tell the installation command which version to get like `!pip install numpy==1.20.0`. This helps make sure the code works the same way each time but it can cause issues if different libraries need conflicting versions of other libraries they rely on. It's like needing a specific size screwdriver

for one job and a different size for another and you only have one set. To fix this people often use special setups that keep different projects and their toolkits separate like using virtual environments.

2. **What if I used color pictures instead of black and white?** Using color pictures means more information for the computer (three color channels: red green and blue instead of just one for grayscale). This extra information can be helpful for recognizing things especially if color is an important clue (like a red car). But it also means more data to process which takes more computer power and time. How I get the pictures ready really matters for how well the model works. Good preparation can help the model find the important patterns.
3. **What if I used a different type of dividing line for the SVM?** I used a simple straight line (linear kernel). But SVM can also use curved or more complex lines using different kernels like RBF (Radial Basis Function) or polynomial. These complex lines can be better for data that isn't easily separated by a straight line which is often the case with pictures. They work by essentially moving the data into a higher-dimensional space where a straight line can separate it. When picking a kernel I'd think about whether the data looks like it can be separated simply or if it needs a more complex approach how much computation it will take (non-linear kernels are usually more complex) and the number of features. Trying different options and using techniques like cross-validation is usually the best way to find the right one.
4. **Why wasn't the simple SVM model super accurate and why are newer methods better for pictures?** The accuracy of around 55% suggests that my chosen classes ('cat' 'dog' 'ship') aren't perfectly separable with a straight line using the flattened grayscale features. A simple linear classifier struggles to pick up on the complex non-linear patterns and layered features in images. Deep learning methods especially Convolutional Neural Networks (CNNs) are better for this because they can automatically learn layered features directly from the raw image data through special layers called convolution and pooling layers. This helps them learn more abstract and consistent representations of the images leading to much higher accuracy on complex image classification tasks like CIFAR-10.