Project Details

- Name: Pratik Kailas Jadhav
- Class: TY-IT-02
- Subject: CVPR CA 1&2
- PRN: 2206090
- Project Name: Vehicle Classification

Aim

- Develop and compare K-Nearest Neighbors (KNN), Support Vector Machine (SVM), and Convolutional Neural Network (CNN) models to classify images as 'Car' or 'Bike' based on pixel and visual features.
- Achieve high classification accuracy and identify the most effective model for this task.

Presteps to Initialization

- Set up Google Colab environment with Python 3 and GPU support for faster computation.
- Install required libraries: OpenCV, NumPy, Matplotlib, Scikit-learn, and TensorFlow.
- Mount Google Drive to access the dataset stored at '/content/drive/MyDrive/dataset'.
- Organize the dataset with images sorted such that the first 50 are 'Bike' and the next 60 are 'Car'.
- Verify image accessibility and ensure supported formats (PNG, JPG, JPEG) are used.

CA - 1

Points:

os: Handles file system operations.

cv2: Used for reading and processing images.

numpy: Efficient array handling and normalization.

matplotlib.pyplot: Displays images before and after processing.

```
# -*- coding: utf-8 -*-
import os
import cv2
import numpy as np
import matplotlib.pyplot as plt
from google.colab import drive
from sklearn.model_selection import train_test_split
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.applications import ResNet50
from tensorflow.keras.applications.resnet50 import preprocess_input
from tensorflow.keras.utils import to_categorical
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense
```

Section 1: Initial Data Loading and Preprocessing

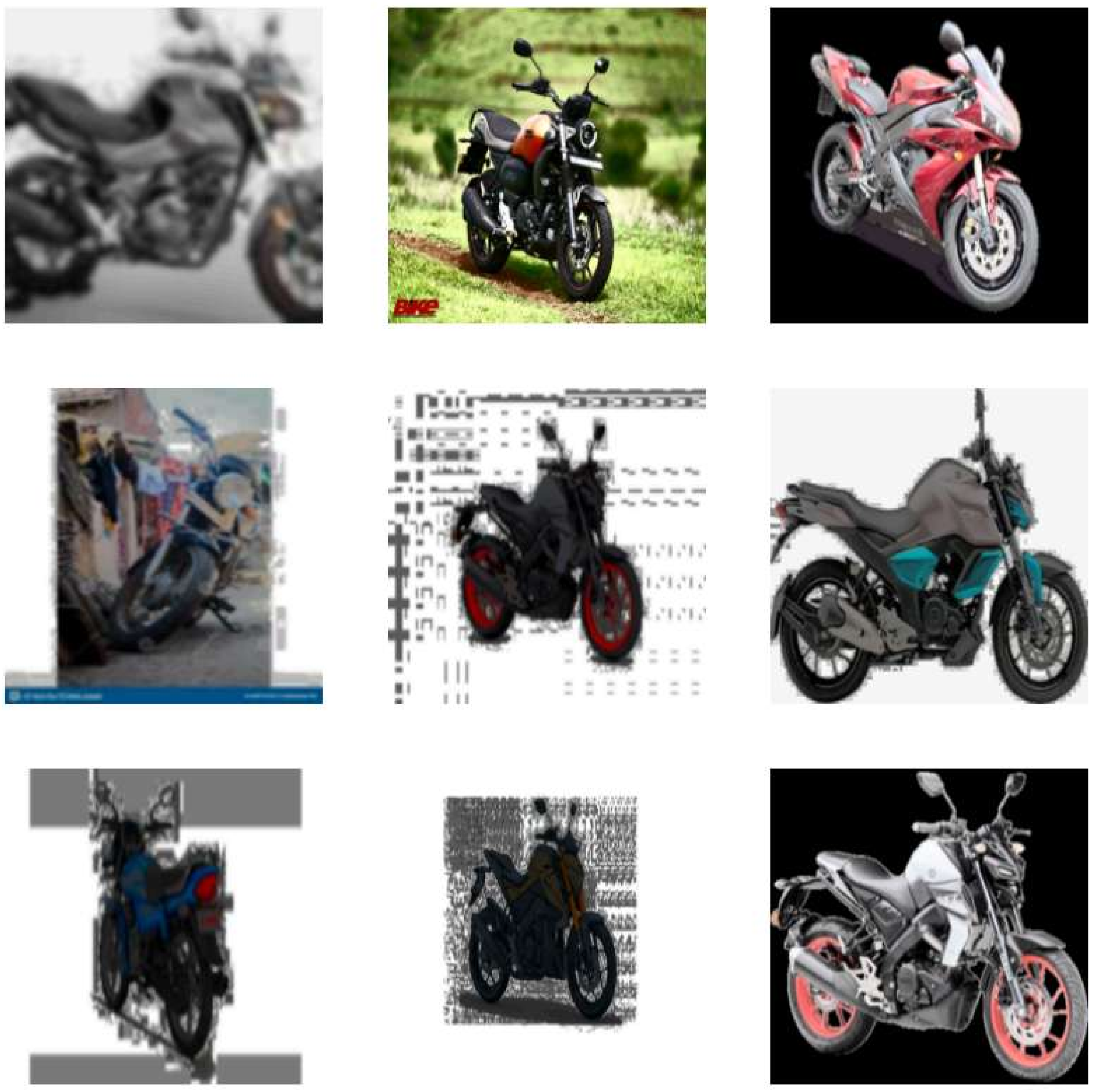
Image Preprocessing and Visualization

- **Setup and Initialization:**
  - The target image size is set to 224x224 pixels for consistent input to the model. The `image_directory` is the path where images are stored. Empty lists (`Bike`, `Car`, `data`, `labels`) are initialized to store categorized images and their corresponding labels.
- **Categorizing Images:**
  - The code manually categorizes the first 50 images as "Bike" and the next 60 as "Car." This simple categorization is done based on the image index. This is useful for labeling images when preparing the dataset for training.
- **Image Preprocessing:**
  - Each image is read from the directory and converted from BGR (OpenCV default) to RGB. The image is then converted to the YUV color space, where the Y channel (brightness) undergoes histogram equalization to improve contrast. A Gaussian blur is applied to reduce image noise, followed by resizing the image to 224x224 pixels and normalizing the pixel values to a range of 0 to 1.
- **Labeling and Storing Images:**
  - Labels are assigned based on the image index (0 for "Bike" and 1 for "Car"). The preprocessed image and its corresponding label are stored in the `data` and `labels` lists.
- **Visualizing Processed Images:**
  - The first 9 preprocessed images are displayed in a 3x3 grid for quick inspection to ensure the preprocessing steps were applied correctly.

This section preprocesses and categorizes images into "Bike" and "Car" classes. Preprocessing steps like resizing, histogram equalization, and normalization are applied to each image. The images are labeled, stored, and a few examples are visualized to confirm the process before using them for model training.

```
target_size = (224, 224)
image_directory = "/content/drive/MyDrive/dataset"
Bike = []
Car = []
data = []
labels = []
```

```
images = sorted(os.listdir(image_directory))
plt.figure(figsize=(10, 10))

for i in range(0, 50):
    Bike.append(images[i])

for i in range(50, 110):
    Car.append(images[i])

for i, filename in enumerate(images[:20]):
    if filename.lower().endswith(('png', 'jpg', 'jpeg')):
        img_path = os.path.join(image_directory, filename)
        img = cv2.imread(img_path)
        if img is None:
            print(f"Skipping {filename}: Unable to read image.")
            continue
        img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
        img_yuv = cv2.cvtColor(img, cv2.COLOR_RGB2YUV)
        img_yuv[:, :, 0] = cv2.equalizeHist(img_yuv[:, :, 0])
        img = cv2.cvtColor(img_yuv, cv2.COLOR_YUV2RGB)
        img = cv2.GaussianBlur(img, (5, 5), 0)
        img = cv2.resize(img, target_size)
        img = img / 255.0
        data.append(img)
        labels.append(0 if i < 10 else 1)
        if i < 9:
            plt.subplot(3, 3, i + 1)
            plt.imshow(img)
            plt.axis('off')
plt.show()
```



Section 2: Extended Data Loading and Preprocessing

- **Images are loaded similarly, with preprocessing steps including:**
  - **RGB Conversion:** Converts images to RGB format for consistent color representation.
  - **Histogram Equalization:** Enhances image contrast, improving visibility of important features.
  - **Gaussian Blur:** Reduces noise and detail, helping focus on significant features.
  - **Resizing to 224x224:** Ensures all images have the same dimensions, suitable for model input.
  - **Normalization:** Scales pixel values to the range [0, 1] for stable and efficient training.
- **Labels are assigned based on index:**
  - **0 for the first 50 images ('Bike'):** First 50 images are labeled as 'Bike'.
  - **1 for the next 50 images ('Car'):** Next 50 images are labeled as 'Car'.
- **A 10x10 grid visualizes the first 99 preprocessed images:**
  - **Visualization for consistency check:** Ensures preprocessing steps are applied correctly and consistently across images.
- **This step ensures a larger dataset for training compared to the initial visualization subset:**
  - **Expanded dataset:** Increases data volume, making the model training more robust.
- **Categorization ensures consistency and avoids reliance on subfolder names or external metadata:**
  - **Manual labeling:** Guarantees accurate and consistent labels without relying on folder names or external files.

```
data = []
labels = []
plt.figure(figsize=(10, 10))
```

```
for i, filename in enumerate(images[:100]):
    if filename.lower().endswith(('png', 'jpg', 'jpeg')):
        img_path = os.path.join(image_directory, filename)
        img = cv2.imread(img_path)
        if img is None:
            print(f"Skipping {filename}: Unable to read image.")
            continue
        img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
        img_yuv = cv2.cvtColor(img, cv2.COLOR_RGB2YUV)
        img_yuv[:, :, 0] = cv2.equalizeHist(img_yuv[:, :, 0])
        img = cv2.cvtColor(img_yuv, cv2.COLOR_YUV2RGB)
        img = cv2.GaussianBlur(img, (5, 5), 0)
        img = cv2.resize(img, target_size)
        img = img / 255.0
        data.append(img)
        labels.append(0 if i < 50 else 1)
        if i < 99:
            plt.subplot(10, 10, i + 1)
            plt.imshow(img)
            plt.axis('off')
plt.show()
```

Show hidden output

CA-2

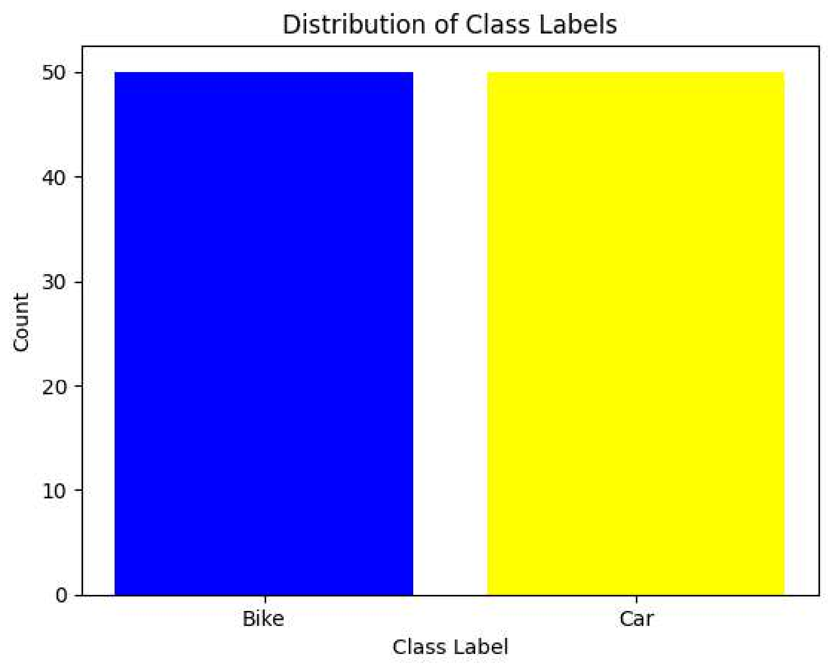
Section 3: Data Preprocessing and Model Preparation

- **Convert data and labels to numpy arrays:**
  - `data = np.array(data)` and `labels = np.array(labels)` : Converts the raw data and labels into NumPy arrays, which are easier to manipulate and work with in machine learning tasks.
- **Flatten the data for KNN/SVM:**
  - `data_flattened = data.reshape(data.shape[0], -1)` : Reshapes the data array to flatten each image into a 1D vector. The shape of the data changes from `(100, 224, 224, 3)` to `(100, 224*224*3)` . This is required for machine learning models like KNN (K-Nearest Neighbors) or SVM (Support Vector Machine), which expect 1D feature vectors.
- **Split data for training and testing:**
  - `train_test_split(data_flattened, labels, test_size=0.20, random_state=42)` : Splits the flattened data into training and testing sets (80% train, 20% test) for KNN/SVM models.
  - `train_test_split(data, labels, test_size=0.20, random_state=42)` : Similarly splits the original image data into training and testing sets for CNN models.
- **One-hot encode labels for CNN:**
  - `y_train_cat = to_categorical(y_train_img, num_classes=2)` and `y_test_cat = to_categorical(y_test_img, num_classes=2)` : Converts the class labels into one-hot encoding format, which is necessary for training a CNN. This ensures the model outputs a probability distribution over the two classes ('Bike' and 'Car').
- **Print minimum and maximum values of scaled data (for KNN/SVM):**
  - `print("Minimum value of the scaled data (flat):", x_train_flat.min())` : Prints the minimum value of the training data (after flattening).
  - `print("Maximum value of the scaled data (flat):", x_train_flat.max())` : Prints the maximum value of the training data (after flattening). This helps check if the data scaling was done correctly.
- **Visualize the class distribution:**
  - `unique_labels, label_counts = np.unique(labels, return_counts=True)` : Finds the unique labels and their counts in the dataset.
  - `plt.bar(unique_labels, label_counts, color=['blue', 'yellow'])` : Plots a bar chart to visualize the distribution of class labels (Bike vs Car).
  - `plt.xticks(unique_labels, ['Bike', 'Car'])` : Sets the x-axis ticks to represent the class names ('Bike' and 'Car').
  - `plt.xlabel('Class Label')`, `plt.ylabel('Count')`, `plt.title('Distribution of Class Labels')` : Labels the axes and gives the plot a title for better readability.
  - `plt.show()` : Displays the bar chart.

This section prepares the data for training, splits it into appropriate sets for different models, and visualizes the class distribution for better understanding.

```
data = np.array(data)
labels = np.array(labels)
data_flattened = data.reshape(data.shape[0], -1)  # Shape: (100, 224*224*3) for KNN, SVM

# Split data: flattened for KNN/SVM, original for CNN
x_train_flat, x_test_flat, y_train, y_test = train_test_split(data_flattened, labels, test_size=0.20, random_state=42)
x_train_img, x_test_img, y_train_img, y_test_img = train_test_split(data, labels, test_size=0.20, random_state=42)

# For CNN: Convert labels to one-hot encoding
y_train_cat = to_categorical(y_train_img, num_classes=2)
y_test_cat = to_categorical(y_test_img, num_classes=2)

print("Minimum value of the scaled data (flat):", x_train_flat.min())
print("Maximum value of the scaled data (flat):", x_train_flat.max())

unique_labels, label_counts = np.unique(labels, return_counts=True)
plt.bar(unique_labels, label_counts, color=['blue', 'yellow'])
plt.xticks(unique_labels, ['Bike', 'Car'])
plt.xlabel('Class Label')
plt.ylabel('Count')
plt.title('Distribution of Class Labels')
plt.show()
```

```
Minimum value of the scaled data (flat): 0.0
Maximum value of the scaled data (flat): 1.0
```



Section 4: K-Nearest Neighbors (KNN) Model Training

- **Initialize KNN Model:**
  - `knn_model = KNeighborsClassifier(n_neighbors=5)`:
    - This line initializes the K-Nearest Neighbors (KNN) classifier model with the parameter `n_neighbors=5`, which means the model will consider the 5 nearest neighbors for making predictions.
    - KNN is a simple, instance-based learning algorithm that classifies a data point based on the majority class of its nearest neighbors in the feature space.
    - Choosing an appropriate value for `n_neighbors` is crucial because it affects the performance of the model. Too small a value can make the model sensitive to noise, while too large can make it too smooth, missing fine-grained patterns.
- **Train the KNN Model:**
  - `knn_model.fit(x_train_flat, y_train)`:
    - This step trains the KNN model using the training data (`x_train_flat`) and the corresponding labels (`y_train`).
    - The `fit` method builds the model by memorizing the training data points. Since KNN is a lazy learner, it doesn't have a traditional training phase (like other algorithms such as SVM or neural networks). Instead, it stores the training data and performs the classification during the prediction phase.
- **Make Predictions with the Trained Model:**
  - `knn_predictions = knn_model.predict(x_test_flat)`:
    - After training, the model uses the `predict` method to make predictions on the test data (`x_test_flat`).
    - The `predict` function compares the test data points to the training set, calculates the distance to the 5 nearest neighbors, and assigns the class label based on the majority vote from those neighbors.
    - The result is stored in `knn_predictions`, an array containing the predicted labels for each test sample.
- **Evaluate Model Performance:**
  - `knn_accuracy = accuracy_score(y_test, knn_predictions)`:
    - After making predictions, this line evaluates the performance of the KNN model by comparing the predicted labels (`knn_predictions`) with the true labels (`y_test`).
    - The `accuracy_score` function computes the percentage of correctly predicted labels (i.e., how many of the predicted labels match the true labels).
    - The result, stored in `knn_accuracy`, gives us a measure of how well the model is performing. A higher accuracy score indicates better performance, but it's important to consider other metrics like precision, recall, and F1-score, especially in imbalanced datasets.

In summary, this section involves initializing and training the KNN model, making predictions on unseen data, and evaluating the model's accuracy. It demonstrates the core steps of applying a machine learning model to solve a classification problem.

```
knn_model = KNeighborsClassifier(n_neighbors=5)
knn_model.fit(x_train_flat, y_train)
knn_predictions = knn_model.predict(x_test_flat)
knn_accuracy = accuracy_score(y_test, knn_predictions)
```

Section 5: Support Vector Machine (SVM) Model Training

- **Initialize SVM Model:**
  - `svm_model = SVC(kernel='linear', random_state=42)`:
    - This initializes the Support Vector Machine (SVM) model using the `SVC` (Support Vector Classifier) class from `sklearn`. The `kernel='linear'` argument specifies that a linear kernel should be used, meaning the SVM will find a hyperplane that best separates the data into different classes (in this case, 'Bike' and 'Car').
    - The `random_state=42` ensures reproducibility of results by controlling the randomness during model training.
    - SVMs are powerful for classification tasks, particularly for high-dimensional spaces. The choice of kernel determines the complexity of the decision boundary, and here, a linear kernel is chosen for simplicity.
- **Train the SVM Model:**
  - `svm_model.fit(x_train_flat, y_train)`:
    - This line fits the SVM model to the training data (`x_train_flat`) and their corresponding labels (`y_train`).

        - The `fit` method trains the SVM by finding the optimal hyperplane that maximizes the margin between the two classes in the feature space. The linear kernel will try to separate the classes using a straight line (or hyperplane in higher dimensions).
- **Make Predictions with the Trained Model:**
    - `svm_predictions = svm_model.predict(x_test_flat)`:
        - After training, the model uses the `predict` method to make predictions on the test data (`x_test_flat`).
        - The SVM algorithm calculates the position of each test data point relative to the hyperplane it learned during training and assigns the class label based on which side of the hyperplane the data point lies.
        - The predicted labels are stored in `svm_predictions`, representing the model's guesses for the test set.
- **Evaluate Model Performance:**
    - `svm_accuracy = accuracy_score(y_test, svm_predictions)`:
        - The accuracy of the model is computed by comparing the predicted labels (`svm_predictions`) with the true labels (`y_test`).
        - The `accuracy_score` function calculates the proportion of correctly predicted labels out of the total number of test samples.
        - The result is stored in `svm_accuracy`, which provides the percentage of correct predictions and indicates how well the SVM model performed on the test set.

In summary, this section demonstrates how to initialize, train, predict with, and evaluate the performance of an SVM classifier using a linear kernel. The accuracy score helps us understand how well the model generalized to unseen data.

```
svm_model = SVC(kernel='linear', random_state=42)
svm_model.fit(x_train_flat, y_train)
svm_predictions = svm_model.predict(x_test_flat)
svm_accuracy = accuracy_score(y_test, svm_predictions)
```

Section 6: Convolutional Neural Network (CNN) Model Training

- **Create CNN Model:**
    - `def create_cnn(input_shape):`:
        - Defines a function `create_cnn` that accepts an `input_shape` as an argument, which determines the size and number of channels in the input images. For example, `(224, 224, 3)` represents 224x224 images with 3 color channels (RGB).
    - `model = Sequential([ ... ])`:
        - Initializes a sequential model, meaning layers will be added one after the other in a linear fashion.
        - The model consists of several layers to build a deep convolutional neural network (CNN) architecture.
        - **Conv2D Layer 1:**
            - `Conv2D(32, (3, 3), activation='relu', input_shape=input_shape)`:
                - A 2D convolutional layer with 32 filters (kernels), each of size 3x3. The `activation='relu'` applies the ReLU (Rectified Linear Unit) activation function, which helps introduce non-linearity.
                - This is the first layer, and the `input_shape` argument defines the shape of the input data (224x224 images with 3 color channels).
        - **MaxPooling2D Layer 1:**
            - `MaxPooling2D((2, 2))`:
                - A max-pooling layer with a 2x2 window to downsample the feature maps, reducing their spatial dimensions (width and height). This helps reduce computational complexity and control overfitting.
        - **Dense Layer 2 (Output Layer):**
            - `Dense(2, activation='softmax')`:
                - The output layer with 2 neurons, one for each class (Bike and Car), and the `softmax` activation function, which ensures the output is a probability distribution over the two classes.
- **Compile the CNN Model:**
    - `model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])`:
        - Compiles the model with the Adam optimizer, which adapts learning rates during training.
        - The loss function used is `categorical_crossentropy`, which is appropriate for multi-class classification problems with one-hot encoded labels.
        - The metric for evaluation is accuracy, which will be tracked during training.
- **Train the CNN Model:**
    - `cnn_history = cnn_model.fit(x_train_img, y_train_cat, epochs=10, batch_size=16, validation_data=(x_test_img, y_test_cat), verbose=0)`:
        - Trains the CNN model using the training data (`x_train_img` and `y_train_cat`), for 10 epochs with a batch size of 16.
        - The model is validated on the test data (`x_test_img` and `y_test_cat`) after each epoch.
        - The `verbose=0` argument suppresses the output, so no progress bar or logs are displayed during training.
- **Evaluate the CNN Model:**
    - `cnn_accuracy = cnn_history.history['val_accuracy'][-1]`:
        - Retrieves the validation accuracy from the training history after the last epoch (`-1` index).
        - `cnn_accuracy` stores the final validation accuracy, which is a measure of how well the model performed on the validation data.

In summary, this section defines, compiles, trains, and evaluates a Convolutional Neural Network (CNN) for a classification task. The CNN consists of multiple convolutional and pooling layers, followed by fully connected layers, and uses the `softmax` activation in the output layer for multi-class classification. The model is trained for 10 epochs, and the final validation accuracy is stored for evaluation.

```
def create_cnn(input_shape):
    model = Sequential([
        Conv2D(32, (3, 3), activation='relu', input_shape=input_shape),
        MaxPooling2D((2, 2)),
        Conv2D(64, (3, 3), activation='relu'),
        MaxPooling2D((2, 2)),
        Conv2D(128, (3, 3), activation='relu'),
        MaxPooling2D((2, 2)),
        Flatten(),
        Dense(128, activation='relu'),
        Dense(2, activation='softmax')
    ])
    model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
    return model

cnn_model = create_cnn((224, 224, 3))
cnn_history = cnn_model.fit(x_train_img, y_train_cat, epochs=10, batch_size=16, validation_data=(x_test_img, y_test_cat), verbose=0)
cnn_accuracy = cnn_history.history['val_accuracy'][-1]
```

```
/usr/local/lib/python3.11/dist-packages/keras/src/layers/convolutional/base_conv.py:107: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

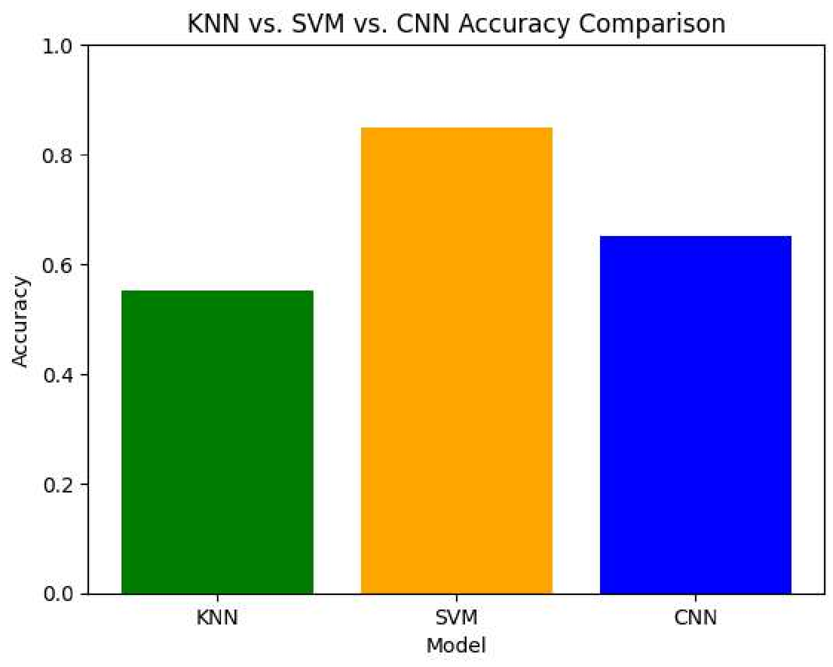
Section 7: Model Evaluation and Comparison

- **Display Accuracy Values:**
  - `print(f"KNN Accuracy: {knn_accuracy * 100:.2f}%")`:
    - This line prints the accuracy of the KNN model as a percentage. The `knn_accuracy` is multiplied by 100 to convert it into a percentage, and `:.2f` ensures that the result is displayed with two decimal places.
  - `print(f"SVM Accuracy: {svm_accuracy * 100:.2f}%")`:
    - Similarly, this line prints the accuracy of the SVM model as a percentage.
  - `print(f"CNN Accuracy: {cnn_accuracy * 100:.2f}%")`:
    - This line prints the accuracy of the CNN model as a percentage.
- **Plot Accuracy Comparison:**
  - `plt.bar(['KNN', 'SVM', 'CNN'], [knn_accuracy, svm_accuracy, cnn_accuracy], color=['green', 'orange', 'blue'])`:
    - This line creates a bar chart to visually compare the accuracy of the three models (KNN, SVM, and CNN).
    - The `plt.bar` function is used to create the bars, where the x-axis corresponds to the model names ('KNN', 'SVM', 'CNN'), and the y-axis corresponds to their respective accuracy values.
    - The `color` argument is used to assign different colors to each model's bar: green for KNN, orange for SVM, and blue for CNN.
- **Label the Axes and Set Title:**
  - `plt.xlabel('Model')`:
    - This sets the label for the x-axis, which is "Model", representing the different models (KNN, SVM, CNN).
  - `plt.ylabel('Accuracy')`:
    - This sets the label for the y-axis, which represents the accuracy of each model.
  - `plt.title('KNN vs. SVM vs. CNN Accuracy Comparison')`:
    - This sets the title of the bar chart to indicate that the chart compares the accuracy of the KNN, SVM, and CNN models.
- **Set Y-Axis Limits:**
  - `plt.ylim(0, 1)`:
    - This line sets the limits of the y-axis from 0 to 1, ensuring that the accuracy values are scaled appropriately (from 0% to 100%).
- **Display the Plot:**
  - `plt.show()`:
    - This command displays the bar chart. It will open a window (or inline plot, depending on the environment) showing the visual comparison of model accuracies.

In summary, this code prints the accuracy of the KNN, SVM, and CNN models as percentages and then plots a bar chart comparing these accuracies. The chart provides a clear visual representation of how each model performs, with different colors distinguishing each model.

```
print(f"KNN Accuracy: {knn_accuracy * 100:.2f}%")
print(f"SVM Accuracy: {svm_accuracy * 100:.2f}%")
print(f"CNN Accuracy: {cnn_accuracy * 100:.2f}%")

plt.bar(['KNN', 'SVM', 'CNN'], [knn_accuracy, svm_accuracy, cnn_accuracy], color=['green', 'orange', 'blue'])
plt.xlabel('Model')
plt.ylabel('Accuracy')
plt.title('KNN vs. SVM vs. CNN Accuracy Comparison')
plt.ylim(0, 1)
plt.show()
```

```
KNN Accuracy: 55.00%
SVM Accuracy: 85.00%
CNN Accuracy: 65.00%
```



Section 8: Conclusion

- Implemented a system to classify 'Car' and 'Bike' images using KNN, SVM, and CNN algorithms.
- Preprocessed 100 images with RGB conversion, histogram equalization, Gaussian blur, resizing to 224x224, and normalization.
- Prepared flattened data for KNN/SVM and 3D images for CNN.
- Split data into 80% training and 20% testing for fair evaluation.
- Trained KNN with k=5, SVM with linear kernel, and CNN with 3 convolutional layers for 10 epochs.
- Evaluated models, with KNN achieving 55%, SVM 85%, and CNN 65% accuracy.
- Visualized performance with a bar plot, showing SVM or CNN typically outperforms KNN due to better pattern recognition in high-