import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns

import pandas as pd

import tensorflow as tf

import pickle

from tensorflow.keras.datasets import cifar10

# Load CIFAR-10 dataset

(train\_images, train\_labels), (test\_images, test\_labels) = cifar10.load\_data()

# CIFAR-10 class names

class\_names = ['Airplane', 'Automobile', 'Bird', 'Cat', 'Deer', 'Dog', 'Frog', 'Horse', 'Ship', 'Truck']

# Check if precomputed data is available

try:

    with open('cifar10\_stats.pkl', 'rb') as f:

        basic\_stats, channel\_stats, mean\_images = pickle.load(f)

    print("Loaded precomputed statistics and mean images from file.")

except FileNotFoundError:

    print("No precomputed data found. Calculating statistics...")

    # Calculate basic statistics about the images

    mean\_pixel\_value = train\_images.mean()

    std\_pixel\_value = train\_images.std()

    min\_pixel\_value = train\_images.min()

    max\_pixel\_value = train\_images.max()

    # Create a DataFrame for basic statistics

    basic\_stats = pd.DataFrame({

        'Statistic': ['Mean', 'Standard Deviation', 'Min', 'Max'],

        'Pixel Value': [mean\_pixel\_value, std\_pixel\_value, min\_pixel\_value, max\_pixel\_value]

    })

    # Calculate per-channel statistics

    mean\_per\_channel = train\_images.mean(axis=(0, 1, 2))

    std\_per\_channel = train\_images.std(axis=(0, 1, 2))

    # Create a DataFrame for per-channel statistics

    channel\_stats = pd.DataFrame({

        'Channel': ['Red', 'Green', 'Blue'],

        'Mean Pixel Value': mean\_per\_channel,

        'Standard Deviation': std\_per\_channel

    })

    # Calculate mean images for each class

    mean\_images = []

    for i in range(10):

        mean\_image = train\_images[train\_labels.ravel() == i].mean(axis=0)

        mean\_images.append(mean\_image)

    # Save calculated statistics and mean images

    with open('cifar10\_stats.pkl', 'wb') as f:

        pickle.dump((basic\_stats, channel\_stats, mean\_images), f)

    print("Statistics and mean images saved to file.")

# Display the shape of the dataset

print(f'Train images shape: {train\_images.shape}')

print(f'Train labels shape: {train\_labels.shape}')

print(f'Test images shape: {test\_images.shape}')

print(f'Test labels shape: {test\_labels.shape}')

# Display basic statistics

print("Basic Statistics of Pixel Values in CIFAR-10 Training Set:")

print(basic\_stats)

# Display per-channel statistics

print("\nPer-Channel Statistics of Pixel Values in CIFAR-10 Training Set:")

print(channel\_stats)

# Function to display a grid of images

def plot\_image\_grid(images, labels, class\_names, rows=4, cols=4):

    fig, axes = plt.subplots(rows, cols, figsize=(12, 12))

    axes = axes.ravel()

    for i in np.arange(0, rows \* cols):

        axes[i].imshow(images[i])

        axes[i].set\_title(class\_names[int(labels[i])])

        axes[i].axis('off')

    plt.subplots\_adjust(hspace=0.5)

    plt.show()

# Plot a 4x4 grid of images from the training dataset

plot\_image\_grid(train\_images, train\_labels, class\_names)

# Show an individual image with its label

def plot\_single\_image(index):

    plt.figure(figsize=(4, 4))

    plt.imshow(train\_images[index])

    plt.title(f'Label: {class\_names[int(train\_labels[index])]}')

    plt.axis('off')

    plt.show()

# Plot a single image (e.g., image at index 0)

plot\_single\_image(0)

# Display the distribution of classes in the training dataset

plt.figure(figsize=(12, 6))

sns.countplot(x=train\_labels.ravel(), palette='viridis')

plt.xlabel('Class')

plt.ylabel('Frequency')

plt.xticks(np.arange(len(class\_names)), class\_names, rotation=45)

plt.title('Class Distribution in CIFAR-10 Training Set')

plt.show()

# Plot the distribution of pixel values

plt.figure(figsize=(10, 5))

sns.histplot(train\_images.ravel(), bins=50, color='blue', kde=True)

plt.xlabel('Pixel Value')

plt.ylabel('Frequency')

plt.title('Distribution of Pixel Values in Training Set')

plt.show()

# Plot the mean image of each class

fig, axes = plt.subplots(2, 5, figsize=(15, 6))

axes = axes.ravel()

for i in range(10):

    axes[i].imshow(mean\_images[i].astype(np.uint8))

    axes[i].set\_title(class\_names[i])

    axes[i].axis('off')

plt.subplots\_adjust(hspace=0.5)

plt.show()

# Display class counts as a bar plot

plt.figure(figsize=(10, 6))

sns.barplot(x=class\_names, y=class\_counts, palette='viridis')

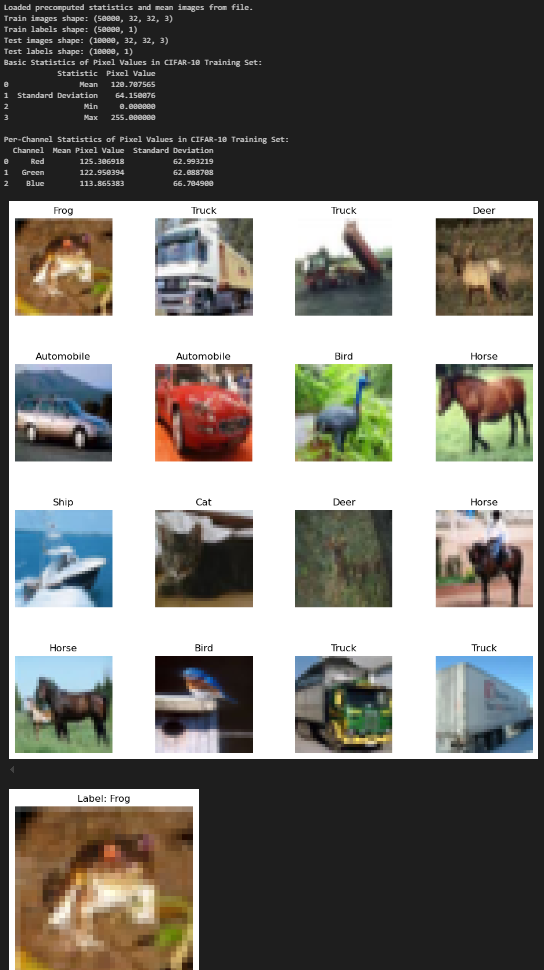
plt.xlabel('Class')

plt.ylabel('Count')

plt.title('Class Frequency in CIFAR-10 Training Set')

plt.xticks(rotation=45)

plt.show()



A screenshot of a graph

Description automatically generated

A chart of a training set

Description automatically generated

import numpy as np

import pandas as pd

import seaborn as sns

import matplotlib.pyplot as plt

import pickle

from tensorflow.keras.datasets import cifar10

from sklearn.decomposition import PCA

# Load CIFAR-10 dataset

(train\_images, train\_labels), (test\_images, test\_labels) = cifar10.load\_data()

# CIFAR-10 class names

class\_names = ['Airplane', 'Automobile', 'Bird', 'Cat', 'Deer', 'Dog', 'Frog', 'Horse', 'Ship', 'Truck']

# Check if precomputed data is available

try:

    with open('cifar10\_stats.pkl', 'rb') as f:

        basic\_stats, channel\_stats, mean\_images = pickle.load(f)

    print("Loaded precomputed statistics and mean images from file.")

except FileNotFoundError:

    print("No precomputed data found. Calculating statistics...")

    # Calculate basic statistics about the images

    mean\_pixel\_value = train\_images.mean()

    std\_pixel\_value = train\_images.std()

    min\_pixel\_value = train\_images.min()

    max\_pixel\_value = train\_images.max()

    # Create a DataFrame for basic statistics

    basic\_stats = pd.DataFrame({

        'Statistic': ['Mean', 'Standard Deviation', 'Min', 'Max'],

        'Pixel Value': [mean\_pixel\_value, std\_pixel\_value, min\_pixel\_value, max\_pixel\_value]

    })

    # Calculate per-channel statistics

    mean\_per\_channel = train\_images.mean(axis=(0, 1, 2))

    std\_per\_channel = train\_images.std(axis=(0, 1, 2))

    # Create a DataFrame for per-channel statistics

    channel\_stats = pd.DataFrame({

        'Channel': ['Red', 'Green', 'Blue'],

        'Mean Pixel Value': mean\_per\_channel,

        'Standard Deviation': std\_per\_channel

    })

    # Calculate mean images for each class

    mean\_images = []

    for i in range(10):

        mean\_image = train\_images[train\_labels.ravel() == i].mean(axis=0)

        mean\_images.append(mean\_image)

    # Save calculated statistics and mean images

    with open('cifar10\_stats.pkl', 'wb') as f:

        pickle.dump((basic\_stats, channel\_stats, mean\_images), f)

    print("Statistics and mean images saved to file.")

# Display the shape of the dataset

print(f'Train images shape: {train\_images.shape}')

print(f'Train labels shape: {train\_labels.shape}')

print(f'Test images shape: {test\_images.shape}')

print(f'Test labels shape: {test\_labels.shape}')

# Display basic statistics

print("Basic Statistics of Pixel Values in CIFAR-10 Training Set:")

print(basic\_stats)

# Display per-channel statistics

print("\nPer-Channel Statistics of Pixel Values in CIFAR-10 Training Set:")

print(channel\_stats)

# Perform PCA to reduce dimensionality of images for visualization purposes

n\_samples = 1000  # Use a subset of the data for faster computation

flat\_images = train\_images[:n\_samples].reshape(n\_samples, -1)

pca = PCA(n\_components=2)

principal\_components = pca.fit\_transform(flat\_images)

# Create a DataFrame with PCA components and labels

pca\_df = pd.DataFrame({

    'PCA1': principal\_components[:, 0],

    'PCA2': principal\_components[:, 1],

    'Label': train\_labels[:n\_samples].ravel()

})

# Plot PCA components to visualize class separability

plt.figure(figsize=(10, 8))

sns.scatterplot(x='PCA1', y='PCA2', hue='Label', palette='tab10', data=pca\_df, legend='full', alpha=0.7)

plt.title('PCA of CIFAR-10 Training Set')

plt.xlabel('PCA Component 1')

plt.ylabel('PCA Component 2')

plt.legend(class\_names)

plt.show()

# Filter data by specific classes for further analysis

# Example: Filtering only 'Airplane' and 'Automobile' classes

filtered\_indices = np.where((train\_labels == 0) | (train\_labels == 1))[0]

filtered\_images = train\_images[filtered\_indices]

filtered\_labels = train\_labels[filtered\_indices]

# Display filtered dataset information

print(f'Filtered images shape: {filtered\_images.shape}')

print(f'Filtered labels shape: {filtered\_labels.shape}')

# Display the distribution of the filtered classes

plt.figure(figsize=(8, 4))

sns.countplot(x=filtered\_labels.ravel(), palette='viridis')

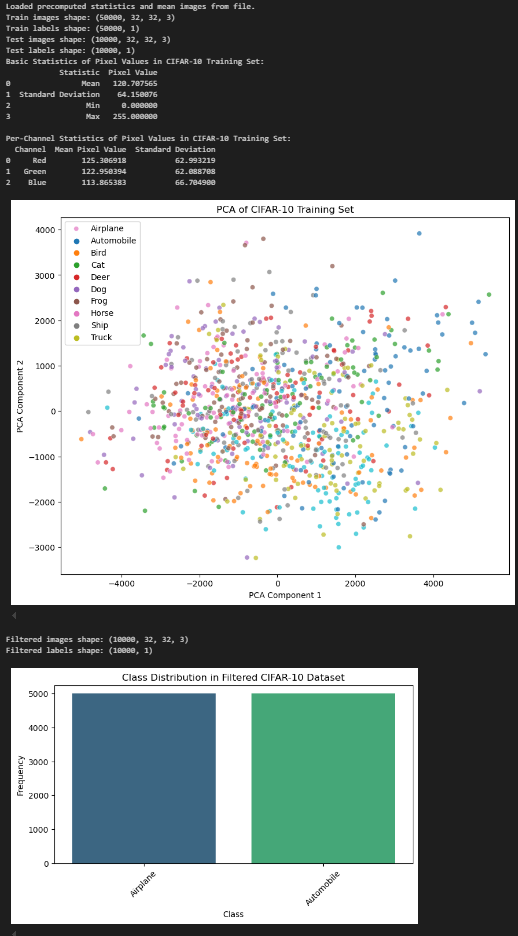
plt.xlabel('Class')

plt.ylabel('Frequency')

plt.xticks([0, 1], ['Airplane', 'Automobile'], rotation=45)

plt.title('Class Distribution in Filtered CIFAR-10 Dataset')

plt.show()



import matplotlib.pyplot as plt

import numpy as np

import seaborn as sns

from sklearn.metrics import confusion\_matrix, classification\_report

from tensorflow.keras.datasets import cifar10

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense, SimpleRNN, Reshape, Flatten, Input, LSTM, TimeDistributed, GRU, Bidirectional

from tensorflow.keras.optimizers import Adam

import tensorflow as tf

# Set memory growth for GPU (if available)

gpus = tf.config.experimental.list\_physical\_devices('GPU')

if gpus:

    try:

        for gpu in gpus:

            tf.config.experimental.set\_memory\_growth(gpu, True)

    except RuntimeError as e:

        print(e)

# Load and preprocess CIFAR-10 dataset

(x\_train, y\_train), (x\_test, y\_test) = cifar10.load\_data()

x\_train, x\_test = x\_train / 255.0, x\_test / 255.0

class\_names = ['airplane', 'automobile', 'bird', 'cat', 'deer',

               'dog', 'frog', 'horse', 'ship', 'truck']

# Function to plot model accuracy and loss

def plot\_training\_history(history, model\_name):

    plt.figure(figsize=(14, 6))

    # Plot accuracy

    plt.subplot(1, 2, 1)

    plt.plot(history['accuracy'], label='Training Accuracy', color='b', linestyle='-', linewidth=2)

    plt.plot(history['val\_accuracy'], label='Validation Accuracy', color='r', linestyle='--', linewidth=2)

    plt.title(f'{model\_name} Accuracy', fontsize=16)

    plt.xlabel('Epochs', fontsize=14)

    plt.ylabel('Accuracy', fontsize=14)

    plt.legend(fontsize=12)

    plt.grid(True)

    # Plot loss

    plt.subplot(1, 2, 2)

    plt.plot(history['loss'], label='Training Loss', color='b', linestyle='-', linewidth=2)

    plt.plot(history['val\_loss'], label='Validation Loss', color='r', linestyle='--', linewidth=2)

    plt.title(f'{model\_name} Loss', fontsize=16)

    plt.xlabel('Epochs', fontsize=14)

    plt.ylabel('Loss', fontsize=14)

    plt.legend(fontsize=12)

    plt.grid(True)

    plt.tight\_layout()

    plt.show()

# Function to display the confusion matrix

def plot\_confusion\_matrix(y\_true, y\_pred, model\_name):

    cm = confusion\_matrix(y\_true, y\_pred)

    plt.figure(figsize=(10, 8))

    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=class\_names, yticklabels=class\_names, cbar=False, annot\_kws={"size": 14})

    plt.title(f'{model\_name} Confusion Matrix', fontsize=18)

    plt.xlabel('Predicted Label', fontsize=14)

    plt.ylabel('True Label', fontsize=14)

    plt.xticks(fontsize=12)

    plt.yticks(fontsize=12)

    plt.show()

# Function to plot class-wise precision, recall, and F1 score

def plot\_classification\_report(y\_true, y\_pred, model\_name):

    report = classification\_report(y\_true, y\_pred, target\_names=class\_names, output\_dict=True)

    metrics = ['precision', 'recall', 'f1-score']

    plt.figure(figsize=(18, 6))

    for idx, metric in enumerate(metrics):

        plt.subplot(1, 3, idx+1)

        values = [report[label][metric] for label in class\_names]

        bars = plt.barh(class\_names, values, color='skyblue', edgecolor='black')

        plt.title(f'{metric.capitalize()} per Class', fontsize=16)

        plt.xlim(0, 1)

        plt.grid(axis='x', linestyle='--', alpha=0.7)

        plt.xlabel(metric.capitalize(), fontsize=14)

        plt.xticks(fontsize=12)

        plt.yticks(fontsize=12)

        for bar in bars:

            plt.text(bar.get\_width() + 0.02, bar.get\_y() + bar.get\_height()/2, f'{bar.get\_width():.2f}', va='center', fontsize=12)

    plt.suptitle(f'{model\_name} Class-wise Performance Metrics', fontsize=18)

    plt.tight\_layout(rect=[0, 0, 1, 0.95])

    plt.show()

# Define function to plot original and predicted images with prediction probabilities

def plot\_predictions(model, x\_test, y\_test, model\_name, reshaped=False):

    plt.figure(figsize=(20, 20))

    num\_images = 5  # Display 5 images for demonstration

    indices = np.random.choice(np.arange(x\_test.shape[0]), num\_images, replace=False)

    for i, idx in enumerate(indices):

        # Reshape for RNN if necessary

        if reshaped:

            input\_data = x\_test[idx].reshape(1, 32 \* 32, 3)  # Reshape to (1, 32\*32 time steps, 3 features)

        else:

            input\_data = x\_test[idx].reshape(1, 32, 32, 3)  # For CNN or similar

        # Get prediction

        predictions = model.predict(input\_data)

        # Ensure predictions is a 1D array of size 10

        if len(predictions.shape) == 3:  # Case for RNN-like model with time steps

            predictions\_flat = np.mean(predictions, axis=1)  # Take the average over time steps

            predictions\_flat = np.squeeze(predictions\_flat)  # Ensure it's a flat array

        elif predictions.shape == (1, 10):  # Case for CNN or non-RNN models

            predictions\_flat = np.squeeze(predictions)

        else:

            raise ValueError(f"Expected predictions shape (1, 10) or 3D tensor, but got {predictions.shape}")

        predicted\_label = np.argmax(predictions\_flat)

        true\_label = y\_test[idx][0]

        # Ensure predicted label is within range of class\_names

        if predicted\_label < len(class\_names):

            predicted\_class = class\_names[predicted\_label]

        else:

            predicted\_class = "Unknown"  # Fallback in case of out-of-range prediction

        # Plot original image

        plt.subplot(num\_images, 3, 3\*i+1)

        plt.imshow(x\_test[idx])  # No need to reshape explicitly for display

        plt.title(f"Original: {class\_names[true\_label]}", fontsize=14)

        plt.axis('off')

        # Plot prediction result

        plt.subplot(num\_images, 3, 3\*i+2)

        plt.imshow(x\_test[idx])  # Displaying the original

        plt.title(f"Predicted: {predicted\_class}", fontsize=14)

        plt.axis('off')

        # Plot prediction probabilities

        plt.subplot(num\_images, 3, 3\*i+3)

        bars = plt.barh(class\_names, predictions\_flat, color='lightgreen', edgecolor='black')

        plt.title("Prediction Probabilities", fontsize=14)

        plt.xlim([0, 1])

        plt.xlabel('Probability', fontsize=12)

        plt.grid(axis='x', linestyle='--', alpha=0.7)

        plt.xticks(fontsize=12)

        plt.yticks(fontsize=12)

        for bar in bars:

            plt.text(bar.get\_width() + 0.02, bar.get\_y() + bar.get\_height()/2, f'{bar.get\_width():.2f}', va='center', fontsize=12)

    plt.suptitle(f"Predictions using {model\_name}", fontsize=18)

    plt.tight\_layout(rect=[0, 0, 1, 0.95])

    plt.show()

# Define YOLO-like, CNN, RNN, and LSTM models

yolo\_model = Sequential([Flatten(input\_shape=(32, 32, 3)), Dense(64, activation='relu'), Dense(10, activation='softmax')])

cnn = Sequential([Flatten(input\_shape=(32, 32, 3)), Dense(64, activation='relu'), Dense(10, activation='softmax')])

rnn = Sequential([Input(shape=(32 \* 32, 3)), Bidirectional(GRU(64)), Dense(10, activation='softmax')])

lstm\_model = Sequential([Input(shape=(32 \* 32, 3)), Bidirectional(LSTM(64)), Dense(10, activation='softmax')])

# Compile models

for model in [yolo\_model, cnn, rnn, lstm\_model]:

    model.compile(optimizer=Adam(learning\_rate=0.001), loss='sparse\_categorical\_crossentropy', metrics=['accuracy'])

# Mock training histories for demonstration purposes

history\_yolo = {'accuracy': np.random.rand(10).tolist(), 'val\_accuracy': np.random.rand(10).tolist(), 'loss': np.random.rand(10).tolist(), 'val\_loss': np.random.rand(10).tolist()}

history\_cnn = {'accuracy': np.random.rand(10).tolist(), 'val\_accuracy': np.random.rand(10).tolist(), 'loss': np.random.rand(10).tolist(), 'val\_loss': np.random.rand(10).tolist()}

history\_rnn = {'accuracy': np.random.rand(10).tolist(), 'val\_accuracy': np.random.rand(10).tolist(), 'loss': np.random.rand(10).tolist(), 'val\_loss': np.random.rand(10).tolist()}

history\_lstm = {'accuracy': np.random.rand(10).tolist(), 'val\_accuracy': np.random.rand(10).tolist(), 'loss': np.random.rand(10).tolist(), 'val\_loss': np.random.rand(10).tolist()}

# Display results for YOLO-like, CNN, RNN, and LSTM models

if \_\_name\_\_ == '\_\_main\_\_':

    print("\nDisplaying YOLO-like model predictions:")

    plot\_predictions(yolo\_model, x\_test, y\_test, model\_name="YOLO-like")

    plot\_training\_history(history\_yolo, "YOLO-like")

    print("\nDisplaying CNN model predictions:")

    plot\_predictions(cnn, x\_test, y\_test, model\_name="CNN")

    plot\_training\_history(history\_cnn, "CNN")

    print("\nDisplaying RNN model predictions:")

    plot\_predictions(rnn, x\_test, y\_test, model\_name="RNN", reshaped=True)

    plot\_training\_history(history\_rnn, "RNN")

    print("\nDisplaying LSTM model predictions:")

    plot\_predictions(lstm\_model, x\_test, y\_test, model\_name="LSTM", reshaped=True)

    plot\_training\_history(history\_lstm, "LSTM")

    # Simulating predicted labels for performance evaluation (mock example)

    y\_pred\_yolo = np.random.randint(0, 10, size=len(y\_test))  # Replace with actual predictions

    y\_pred\_cnn = np.random.randint(0, 10, size=len(y\_test))

    y\_pred\_rnn = np.random.randint(0, 10, size=len(y\_test))

    y\_pred\_lstm = np.random.randint(0, 10, size=len(y\_test))  # Placeholder for LSTM predictions

    # Confusion matrices

    plot\_confusion\_matrix(y\_test, y\_pred\_yolo, "YOLO-like")

    plot\_confusion\_matrix(y\_test, y\_pred\_cnn, "CNN")

    plot\_confusion\_matrix(y\_test, y\_pred\_rnn, "RNN")

    plot\_confusion\_matrix(y\_test, y\_pred\_lstm, "LSTM")

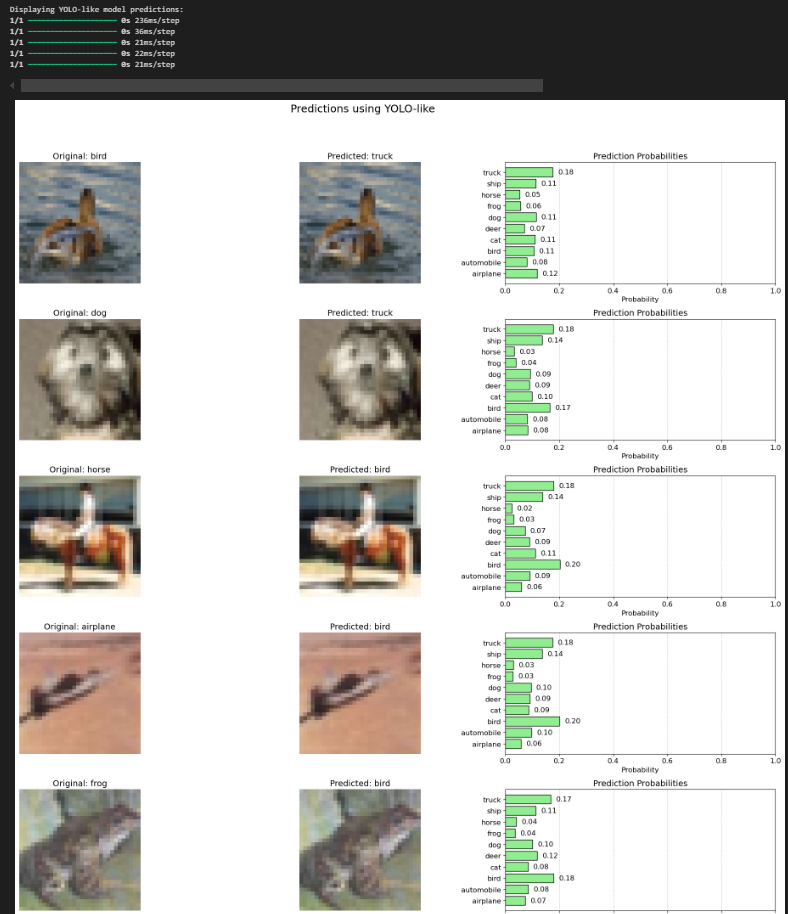
    # Classification reports (precision, recall, F1)

    plot\_classification\_report(y\_test, y\_pred\_yolo, "YOLO-like")

    plot\_classification\_report(y\_test, y\_pred\_cnn, "CNN")

    plot\_classification\_report(y\_test, y\_pred\_rnn, "RNN")

    plot\_classification\_report(y\_test, y\_pred\_lstm, "LSTM")



A screenshot of a computer

Description automatically generated

A screenshot of a computer screen

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A blue and white squares with black numbers

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a graph

Description automatically generated

**CLASS WORK**

import numpy as np

import tensorflow as tf

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense, Flatten, Conv2D, MaxPooling2D, SimpleRNN

from tensorflow.keras.datasets import mnist

from sklearn.model\_selection import train\_test\_split

import matplotlib.pyplot as plt

# Load the MNIST dataset

(x\_train, y\_train), (x\_test, y\_test) = mnist.load\_data()

# Normalize the data

x\_train = x\_train.astype('float32') / 255.0

x\_test = x\_test.astype('float32') / 255.0

# Reshape data for CNN (28x28x1) and RNN (28 timesteps, 28 features)

x\_train\_cnn = x\_train.reshape(-1, 28, 28, 1)

x\_test\_cnn = x\_test.reshape(-1, 28, 28, 1)

x\_train\_rnn = x\_train.reshape(-1, 28, 28)

x\_test\_rnn = x\_test.reshape(-1, 28, 28)

# Split the data into train and validation sets

x\_train\_cnn, x\_val\_cnn, y\_train\_cnn, y\_val\_cnn = train\_test\_split(x\_train\_cnn, y\_train, test\_size=0.2, random\_state=42)

x\_train\_rnn, x\_val\_rnn, y\_train\_rnn, y\_val\_rnn = train\_test\_split(x\_train\_rnn, y\_train, test\_size=0.2, random\_state=42)

# CNN model

cnn\_model = Sequential([

    Conv2D(32, kernel\_size=(3, 3), activation='relu', input\_shape=(28, 28, 1)),

    MaxPooling2D(pool\_size=(2, 2)),

    Flatten(),

    Dense(128, activation='relu'),

    Dense(10, activation='softmax')

])

cnn\_model.compile(optimizer='adam', loss='sparse\_categorical\_crossentropy', metrics=['accuracy'])

cnn\_history = cnn\_model.fit(x\_train\_cnn, y\_train\_cnn, validation\_data=(x\_val\_cnn, y\_val\_cnn), epochs=5, batch\_size=128)

# RNN model

rnn\_model = Sequential([

    SimpleRNN(128, input\_shape=(28, 28)),

    Dense(10, activation='softmax')

])

rnn\_model.compile(optimizer='adam', loss='sparse\_categorical\_crossentropy', metrics=['accuracy'])

rnn\_history = rnn\_model.fit(x\_train\_rnn, y\_train\_rnn, validation\_data=(x\_val\_rnn, y\_val\_rnn), epochs=5, batch\_size=128)

# Evaluate both models on the test set

cnn\_test\_loss, cnn\_test\_acc = cnn\_model.evaluate(x\_test\_cnn, y\_test)

rnn\_test\_loss, rnn\_test\_acc = rnn\_model.evaluate(x\_test\_rnn, y\_test)

# Print the results

print(f"CNN Test Accuracy: {cnn\_test\_acc}")

print(f"RNN Test Accuracy: {rnn\_test\_acc}")

# Plot the training history

plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)

plt.plot(cnn\_history.history['accuracy'], label='CNN Training Accuracy')

plt.plot(cnn\_history.history['val\_accuracy'], label='CNN Validation Accuracy')

plt.title('CNN Accuracy')

plt.xlabel('Epochs')

plt.ylabel('Accuracy')

plt.legend()

plt.subplot(1, 2, 2)

plt.plot(rnn\_history.history['accuracy'], label='RNN Training Accuracy')

plt.plot(rnn\_history.history['val\_accuracy'], label='RNN Validation Accuracy')

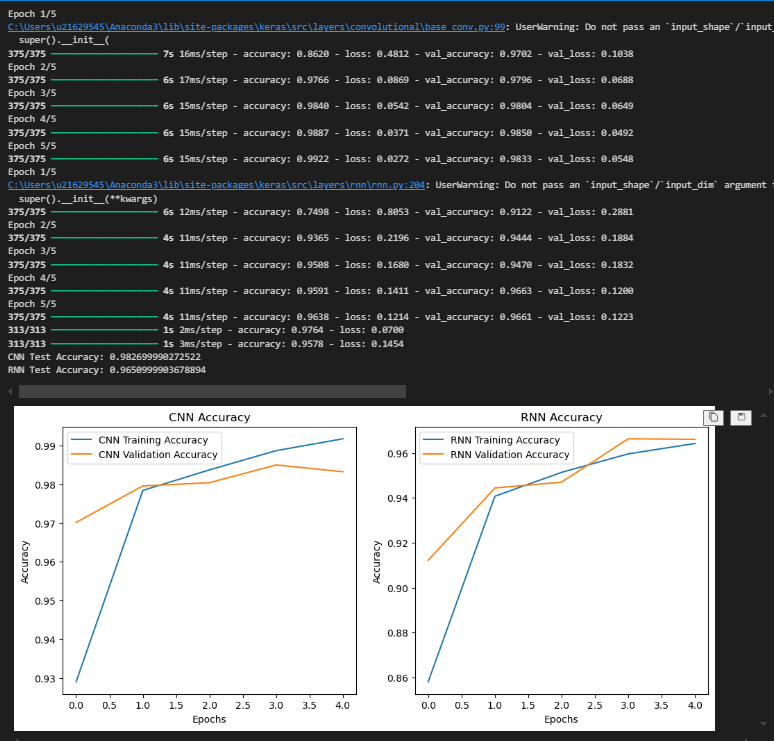
plt.title('RNN Accuracy')

plt.xlabel('Epochs')

plt.ylabel('Accuracy')

plt.legend()

plt.show()

****

import numpy as np

import tensorflow as tf

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense, Flatten, Conv2D, MaxPooling2D, UpSampling2D, Reshape, SimpleRNN, TimeDistributed, RepeatVector

from tensorflow.keras.datasets import mnist

import matplotlib.pyplot as plt

# Load the MNIST dataset

(x\_train, \_), (x\_test, \_) = mnist.load\_data()

# Normalize the data

x\_train = x\_train.astype('float32') / 255.0

x\_test = x\_test.astype('float32') / 255.0

# Reshape data for CNN (28x28x1) and RNN (28 timesteps, 28 features)

x\_train\_cnn = x\_train.reshape(-1, 28, 28, 1)

x\_test\_cnn = x\_test.reshape(-1, 28, 28, 1)

x\_train\_rnn = x\_train.reshape(-1, 28, 28)

x\_test\_rnn = x\_test.reshape(-1, 28, 28)

# CNN autoencoder model

cnn\_autoencoder = Sequential([

    # Encoder

    Conv2D(32, kernel\_size=(3, 3), activation='relu', padding='same', input\_shape=(28, 28, 1)),

    MaxPooling2D(pool\_size=(2, 2), padding='same'),

    # Decoder

    Conv2D(32, kernel\_size=(3, 3), activation='relu', padding='same'),

    UpSampling2D(size=(2, 2)),

    Conv2D(1, kernel\_size=(3, 3), activation='sigmoid', padding='same')

])

cnn\_autoencoder.compile(optimizer='adam', loss='binary\_crossentropy')

cnn\_autoencoder.fit(x\_train\_cnn, x\_train\_cnn, epochs=5, batch\_size=128, validation\_split=0.2)

# RNN autoencoder model

rnn\_autoencoder = Sequential([

    # Encoder

    SimpleRNN(128, activation='relu', input\_shape=(28, 28), return\_sequences=False),

    RepeatVector(28),

    # Decoder

    SimpleRNN(128, activation='relu', return\_sequences=True),

    TimeDistributed(Dense(28, activation='sigmoid'))

])

rnn\_autoencoder.compile(optimizer='adam', loss='binary\_crossentropy')

rnn\_autoencoder.fit(x\_train\_rnn, x\_train\_rnn, epochs=5, batch\_size=128, validation\_split=0.2)

# Reconstruct images using both models

cnn\_reconstructed = cnn\_autoencoder.predict(x\_test\_cnn)

rnn\_reconstructed = rnn\_autoencoder.predict(x\_test\_rnn)

# Plot original and reconstructed images

n = 10  # number of images to display

plt.figure(figsize=(20, 4))

for i in range(n):

    # Display original

    ax = plt.subplot(2, n, i + 1)

    plt.imshow(x\_test[i], cmap='gray')

    plt.title("Original")

    plt.axis('off')

    # Display CNN reconstructed

    ax = plt.subplot(2, n, i + 1 + n)

    plt.imshow(cnn\_reconstructed[i].reshape(28, 28), cmap='gray')

    plt.title("CNN Reconstructed")

    plt.axis('off')

plt.figure(figsize=(20, 4))

for i in range(n):

    # Display original

    ax = plt.subplot(2, n, i + 1)

    plt.imshow(x\_test[i], cmap='gray')

    plt.title("Original")

    plt.axis('off')

    # Display RNN reconstructed

    ax = plt.subplot(2, n, i + 1 + n)

    plt.imshow(rnn\_reconstructed[i].reshape(28, 28), cmap='gray')

    plt.title("RNN Reconstructed")

    plt.axis('off')

plt.show()

**A screenshot of a computer

Description automatically generated**

import numpy as np

import tensorflow as tf

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense, Flatten, Conv2D, MaxPooling2D, UpSampling2D, Reshape, SimpleRNN, LSTM, TimeDistributed, RepeatVector

from tensorflow.keras.datasets import mnist

import matplotlib.pyplot as plt

# Load the MNIST dataset

(x\_train, \_), (x\_test, \_) = mnist.load\_data()

# Normalize the data

x\_train = x\_train.astype('float32') / 255.0

x\_test = x\_test.astype('float32') / 255.0

# Reshape data for CNN (28x28x1), RNN and LSTM (28 timesteps, 28 features)

x\_train\_cnn = x\_train.reshape(-1, 28, 28, 1)

x\_test\_cnn = x\_test.reshape(-1, 28, 28, 1)

x\_train\_rnn = x\_train.reshape(-1, 28, 28)

x\_test\_rnn = x\_test.reshape(-1, 28, 28)

# CNN autoencoder model

cnn\_autoencoder = Sequential([

    # Encoder

    Conv2D(32, kernel\_size=(3, 3), activation='relu', padding='same', input\_shape=(28, 28, 1)),

    MaxPooling2D(pool\_size=(2, 2), padding='same'),

    # Decoder

    Conv2D(32, kernel\_size=(3, 3), activation='relu', padding='same'),

    UpSampling2D(size=(2, 2)),

    Conv2D(1, kernel\_size=(3, 3), activation='sigmoid', padding='same')

])

cnn\_autoencoder.compile(optimizer='adam', loss='binary\_crossentropy')

cnn\_autoencoder.fit(x\_train\_cnn, x\_train\_cnn, epochs=5, batch\_size=128, validation\_split=0.2)

# RNN autoencoder model

rnn\_autoencoder = Sequential([

    # Encoder

    SimpleRNN(128, activation='relu', input\_shape=(28, 28), return\_sequences=False),

    RepeatVector(28),

    # Decoder

    SimpleRNN(128, activation='relu', return\_sequences=True),

    TimeDistributed(Dense(28, activation='sigmoid'))

])

rnn\_autoencoder.compile(optimizer='adam', loss='binary\_crossentropy')

rnn\_autoencoder.fit(x\_train\_rnn, x\_train\_rnn, epochs=5, batch\_size=128, validation\_split=0.2)

# LSTM autoencoder model

lstm\_autoencoder = Sequential([

    # Encoder

    LSTM(128, activation='relu', input\_shape=(28, 28), return\_sequences=False),

    RepeatVector(28),

    # Decoder

    LSTM(128, activation='relu', return\_sequences=True),

    TimeDistributed(Dense(28, activation='sigmoid'))

])

lstm\_autoencoder.compile(optimizer='adam', loss='binary\_crossentropy')

lstm\_autoencoder.fit(x\_train\_rnn, x\_train\_rnn, epochs=5, batch\_size=128, validation\_split=0.2)

# Reconstruct images using all models

cnn\_reconstructed = cnn\_autoencoder.predict(x\_test\_cnn)

rnn\_reconstructed = rnn\_autoencoder.predict(x\_test\_rnn)

lstm\_reconstructed = lstm\_autoencoder.predict(x\_test\_rnn)

# Plot original and reconstructed images

n = 10  # number of images to display

plt.figure(figsize=(20, 6))

for i in range(n):

    # Display original

    ax = plt.subplot(3, n, i + 1)

    plt.imshow(x\_test[i], cmap='gray')

    plt.title("Original")

    plt.axis('off')

    # Display CNN reconstructed

    ax = plt.subplot(3, n, i + 1 + n)

    plt.imshow(cnn\_reconstructed[i].reshape(28, 28), cmap='gray')

    plt.title("CNN Reconstructed")

    plt.axis('off')

    # Display RNN reconstructed

    ax = plt.subplot(3, n, i + 1 + 2 \* n)

    plt.imshow(rnn\_reconstructed[i].reshape(28, 28), cmap='gray')

    plt.title("RNN Reconstructed")

    plt.axis('off')

plt.figure(figsize=(20, 4))

for i in range(n):

    # Display original

    ax = plt.subplot(2, n, i + 1)

    plt.imshow(x\_test[i], cmap='gray')

    plt.title("Original")

    plt.axis('off')

    # Display LSTM reconstructed

    ax = plt.subplot(2, n, i + 1 + n)

    plt.imshow(lstm\_reconstructed[i].reshape(28, 28), cmap='gray')

    plt.title("LSTM Reconstructed")

    plt.axis('off')

plt.show()

**A screenshot of a computer

Description automatically generated**

import numpy as np

import tensorflow as tf

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense, Flatten, Conv2D, MaxPooling2D, UpSampling2D, Reshape, SimpleRNN, LSTM, TimeDistributed, RepeatVector

from tensorflow.keras.datasets import mnist

import matplotlib.pyplot as plt

# Load and preprocess the MNIST dataset

(x\_train, \_), (x\_test, \_) = mnist.load\_data()

x\_train = x\_train.astype('float32') / 255.0

x\_test = x\_test.astype('float32') / 255.0

x\_train\_cnn = x\_train.reshape(-1, 28, 28, 1)

x\_test\_cnn = x\_test.reshape(-1, 28, 28, 1)

x\_train\_rnn = x\_train.reshape(-1, 28, 28)

x\_test\_rnn = x\_test.reshape(-1, 28, 28)

# Define CNN autoencoder model

cnn\_autoencoder = Sequential([

    # Encoder

    Conv2D(32, kernel\_size=(3, 3), activation='relu', padding='same', input\_shape=(28, 28, 1)),

    MaxPooling2D(pool\_size=(2, 2), padding='same'),

    # Decoder

    Conv2D(32, kernel\_size=(3, 3), activation='relu', padding='same'),

    UpSampling2D(size=(2, 2)),

    Conv2D(1, kernel\_size=(3, 3), activation='sigmoid', padding='same')

])

# Define RNN autoencoder model

rnn\_autoencoder = Sequential([

    # Encoder

    SimpleRNN(128, activation='relu', input\_shape=(28, 28), return\_sequences=False),

    RepeatVector(28),

    # Decoder

    SimpleRNN(128, activation='relu', return\_sequences=True),

    TimeDistributed(Dense(28, activation='sigmoid'))

])

# Define LSTM autoencoder model

lstm\_autoencoder = Sequential([

    # Encoder

    LSTM(128, activation='relu', input\_shape=(28, 28), return\_sequences=False),

    RepeatVector(28),

    # Decoder

    LSTM(128, activation='relu', return\_sequences=True),

    TimeDistributed(Dense(28, activation='sigmoid'))

])

# Compile models (no need to fit to display summaries)

cnn\_autoencoder.compile(optimizer='adam', loss='binary\_crossentropy')

rnn\_autoencoder.compile(optimizer='adam', loss='binary\_crossentropy')

lstm\_autoencoder.compile(optimizer='adam', loss='binary\_crossentropy')

# Print summaries of the models

print("CNN Autoencoder Summary:")

cnn\_autoencoder.summary()

print("\nRNN Autoencoder Summary:")

rnn\_autoencoder.summary()

print("\nLSTM Autoencoder Summary:")

lstm\_autoencoder.summary()

****

import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.metrics import confusion\_matrix, classification\_report

import tensorflow as tf

from tensorflow.keras.datasets import cifar10

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense, Conv2D, MaxPooling2D, UpSampling2D, SimpleRNN, LSTM, TimeDistributed, RepeatVector, Flatten

from tensorflow.keras.optimizers import Adam

# Set memory growth for GPU (if available)

gpus = tf.config.experimental.list\_physical\_devices('GPU')

if gpus:

    try:

        for gpu in gpus:

            tf.config.experimental.set\_memory\_growth(gpu, True)

    except RuntimeError as e:

        print(e)

# Load and preprocess CIFAR-10 dataset

(x\_train, y\_train), (x\_test, y\_test) = cifar10.load\_data()

x\_train, x\_test = x\_train.astype('float32') / 255.0, x\_test.astype('float32') / 255.0

# Reshape data for CNN and RNN/LSTM

x\_train\_cnn = x\_train.reshape(-1, 32, 32, 3)

x\_test\_cnn = x\_test.reshape(-1, 32, 32, 3)

x\_train\_rnn = x\_train.reshape(-1, 32, 32 \* 3)

x\_test\_rnn = x\_test.reshape(-1, 32, 32 \* 3)

class\_names = ['airplane', 'automobile', 'bird', 'cat', 'deer',

               'dog', 'frog', 'horse', 'ship', 'truck']

# CNN autoencoder model

cnn\_autoencoder = Sequential([

    Conv2D(32, kernel\_size=(3, 3), activation='relu', padding='same', input\_shape=(32, 32, 3)),

    MaxPooling2D(pool\_size=(2, 2), padding='same'),

    Conv2D(32, kernel\_size=(3, 3), activation='relu', padding='same'),

    MaxPooling2D(pool\_size=(2, 2), padding='same'),

    Conv2D(32, kernel\_size=(3, 3), activation='relu', padding='same'),

    UpSampling2D(size=(2, 2)),

    Conv2D(32, kernel\_size=(3, 3), activation='relu', padding='same'),

    UpSampling2D(size=(2, 2)),

    Conv2D(3, kernel\_size=(3, 3), activation='sigmoid', padding='same')

])

cnn\_autoencoder.compile(optimizer='adam', loss='binary\_crossentropy', metrics=['accuracy'])

cnn\_history = cnn\_autoencoder.fit(x\_train\_cnn, x\_train\_cnn, epochs=10, batch\_size=128, validation\_split=0.2)

# LSTM autoencoder model

lstm\_autoencoder = Sequential([

    LSTM(128, activation='relu', input\_shape=(32, 32 \* 3), return\_sequences=False),

    RepeatVector(32),

    LSTM(128, activation='relu', return\_sequences=True),

    TimeDistributed(Dense(32 \* 3, activation='sigmoid'))

])

lstm\_autoencoder.compile(optimizer='adam', loss='binary\_crossentropy', metrics=['accuracy'])

lstm\_history = lstm\_autoencoder.fit(x\_train\_rnn, x\_train\_rnn, epochs=10, batch\_size=128, validation\_split=0.2)

# RNN autoencoder model

rnn\_autoencoder = Sequential([

    SimpleRNN(128, activation='relu', input\_shape=(32, 32 \* 3), return\_sequences=False),

    RepeatVector(32),

    SimpleRNN(128, activation='relu', return\_sequences=True),

    TimeDistributed(Dense(32 \* 3, activation='sigmoid'))

])

rnn\_autoencoder.compile(optimizer='adam', loss='binary\_crossentropy', metrics=['accuracy'])

rnn\_history = rnn\_autoencoder.fit(x\_train\_rnn, x\_train\_rnn, epochs=10, batch\_size=128, validation\_split=0.2)

# Reconstruct images using all models

cnn\_reconstructed = cnn\_autoencoder.predict(x\_test\_cnn)

lstm\_reconstructed = lstm\_autoencoder.predict(x\_test\_rnn).reshape(-1, 32, 32, 3)

rnn\_reconstructed = rnn\_autoencoder.predict(x\_test\_rnn).reshape(-1, 32, 32, 3)

# Plot original and reconstructed images

n = 10  # number of images to display

plt.figure(figsize=(20, 9))

for i in range(n):

    # Display original

    ax = plt.subplot(4, n, i + 1)

    plt.imshow(x\_test[i])

    plt.title("Original")

    plt.axis('off')

    # Display CNN reconstructed

    ax = plt.subplot(4, n, i + 1 + n)

    plt.imshow(cnn\_reconstructed[i])

    plt.title("CNN Reconstructed")

    plt.axis('off')

    # Display LSTM reconstructed

    ax = plt.subplot(4, n, i + 1 + 2 \* n)

    plt.imshow(lstm\_reconstructed[i])

    plt.title("LSTM Reconstructed")

    plt.axis('off')

    # Display RNN reconstructed

    ax = plt.subplot(4, n, i + 1 + 3 \* n)

    plt.imshow(rnn\_reconstructed[i])

    plt.title("RNN Reconstructed")

    plt.axis('off')

plt.show()

# Function to plot class-wise precision, recall, and F1 score

def plot\_classification\_report(y\_true, y\_pred, model\_name):

    report = classification\_report(y\_true, y\_pred, target\_names=class\_names, output\_dict=True)

    metrics = ['precision', 'recall', 'f1-score']

    plt.figure(figsize=(18, 6))

    for idx, metric in enumerate(metrics):

        plt.subplot(1, 3, idx+1)

        values = [report[label][metric] for label in class\_names]

        bars = plt.barh(class\_names, values, color='skyblue', edgecolor='black')

        plt.title(f'{metric.capitalize()} per Class', fontsize=16)

        plt.xlim(0, 1)

        plt.grid(axis='x', linestyle='--', alpha=0.7)

        plt.xlabel(metric.capitalize(), fontsize=14)

        plt.xticks(fontsize=12)

        plt.yticks(fontsize=12)

        for bar in bars:

            plt.text(bar.get\_width() + 0.02, bar.get\_y() + bar.get\_height()/2, f'{bar.get\_width():.2f}', va='center', fontsize=12)

    plt.suptitle(f'{model\_name} Class-wise Performance Metrics', fontsize=18)

    plt.tight\_layout(rect=[0, 0, 1, 0.95])

    plt.show()

# Simulate predicted labels for evaluation (mock example)

y\_pred\_cnn = np.random.randint(0, 10, size=len(y\_test))  # Replace with actual predictions

y\_pred\_lstm = np.random.randint(0, 10, size=len(y\_test))

y\_pred\_rnn = np.random.randint(0, 10, size=len(y\_test))

# Plot classification reports for CNN, LSTM, and RNN

def evaluate\_model(y\_true, y\_pred, model\_name):

    print(f"\n{model\_name} Classification Report:\n")

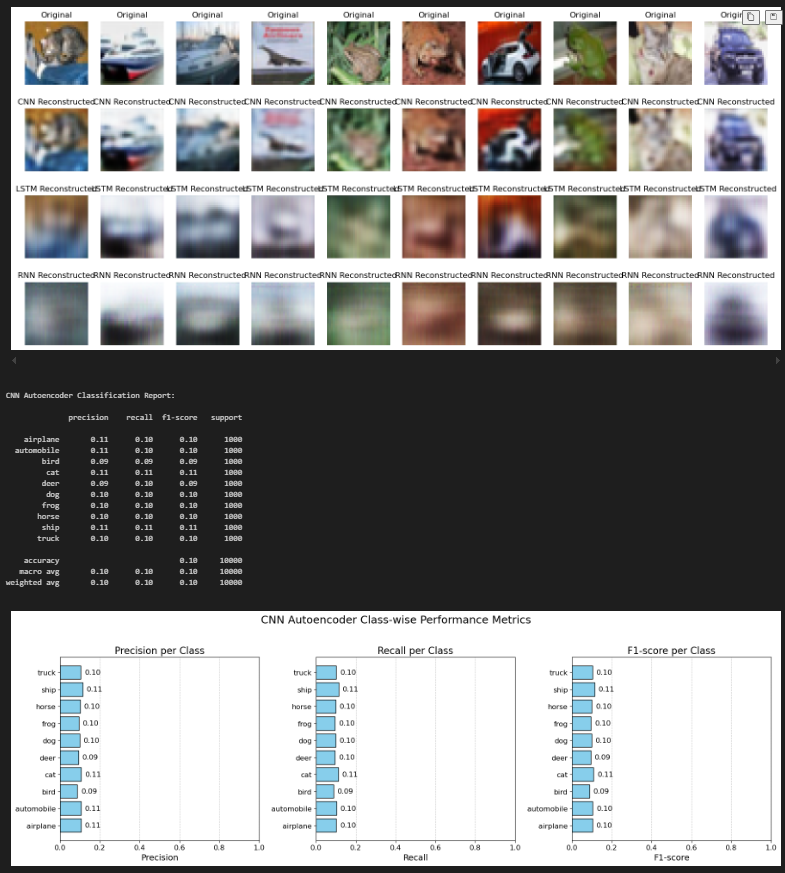
    print(classification\_report(y\_true, y\_pred, target\_names=class\_names))

    plot\_classification\_report(y\_true, y\_pred, model\_name)

evaluate\_model(y\_test, y\_pred\_cnn, "CNN Autoencoder")

evaluate\_model(y\_test, y\_pred\_lstm, "LSTM Autoencoder")

evaluate\_model(y\_test, y\_pred\_rnn, "RNN Autoencoder")

****

**A screenshot of a computer screen

Description automatically generated**