Importing modules

```
!pip install ucimlrepo numpy pandas matplotlib scikit-learn hmmlearn
seaborn
```

```
Collecting ucimlrepo
  Downloading ucimlrepo-0.0.7-py3-none-any.whl.metadata (5.5 kB)
Requirement already satisfied: numpy in
/usr/local/lib/python3.12/dist-packages (2.0.2)
Requirement already satisfied: pandas in
/usr/local/lib/python3.12/dist-packages (2.2.2)
Requirement already satisfied: matplotlib in
/usr/local/lib/python3.12/dist-packages (3.10.0)
Requirement already satisfied: scikit-learn in
/usr/local/lib/python3.12/dist-packages (1.6.1)
Collecting hmmlearn
  Downloading hmmlearn-0.3.3-cp312-cp312-
manylinux_2_17_x86_64.manylinux2014_x86_64.whl.metadata (3.0 kB)
Requirement already satisfied: seaborn in
/usr/local/lib/python3.12/dist-packages (0.13.2)
Requirement already satisfied: certifi>=2020.12.5 in
/usr/local/lib/python3.12/dist-packages (from ucimlrepo) (2025.8.3)
Requirement already satisfied: python-dateutil>=2.8.2 in
/usr/local/lib/python3.12/dist-packages (from pandas) (2.9.0.post0)
Requirement already satisfied: pytz>=2020.1 in
/usr/local/lib/python3.12/dist-packages (from pandas) (2025.2)
Requirement already satisfied: tzdata>=2022.7 in
/usr/local/lib/python3.12/dist-packages (from pandas) (2025.2)
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) (4.60.1)
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) (3.2.5)
Requirement already satisfied: scipy>=1.6.0 in
/usr/local/lib/python3.12/dist-packages (from scikit-learn) (1.16.2)
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: six>=1.5 in
/usr/local/lib/python3.12/dist-packages (from python-dateutil>=2.8.2-
>pandas) (1.17.0)
Downloading ucimlrepo-0.0.7-py3-none-any.whl (8.0 kB)
Downloading hmmlearn-0.3.3-cp312-cp312-
manylinux_2_17_x86_64.manylinux2014_x86_64.whl (165 kB)
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 166.0/166.0 kB 8.0 MB/s eta
0:00:00
lrepo, hmmlearn
Successfully installed hmmlearn-0.3.3 ucimlrepo-0.0.7
```

Defining Classifiers

Graph Plotting

```
# Function to plot confusion matrix
def plot_confusion_matrix(cm, classifier_name):
    plt.figure(figsize=(8, 6))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
    plt.title(f'{classifier_name} Confusion Matrix')
    plt.xlabel('Predicted')
    plt.ylabel('True')
    plt.tight_layout()
    plt.show()


# Function to plot ROC Curve and AUC
def plot_roc_curve(fpr, tpr, auc_value, classifier_name):
    plt.figure(figsize=(8, 6))
    plt.plot(fpr, tpr, color='blue', label=f'ROC curve (AUC =
{auc_value:.2f})')
    plt.plot([0, 1], [0, 1], color='gray', linestyle='--')
    plt.title(f"ROC Curve for {classifier_name}")
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.legend(loc="lower right")
    plt.tight_layout()
    plt.show()

# Function to plot accuracy bar graph
def plot_accuracy_bar_graph(accuracies, test_sizes, classifier_name):
    plt.figure(figsize=(8, 6))
    plt.bar([f'{int(100*(1-size))}:{int(100*size)}' for size in
test_sizes], accuracies, align='center')
    plt.title(f'Accuracy for {classifier_name} Across Different Test
Sizes')
    plt.xlabel('Train-Test Split')
```

```
    plt.ylabel('Accuracy')
    plt.ylim(0, 1.0)
    plt.tight_layout()
    plt.show()
```

Gaussian HMM

```
from hmmlearn.hmm import GaussianHMM
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import label_binarize, LabelEncoder
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np
from sklearn.metrics import accuracy_score, confusion_matrix,
classification_report, roc_curve, roc_auc_score

# Function for Gaussian HMM
def train_gaussian_hmm(X, Y, test_sizes=[0.2, 0.3, 0.4, 0.5]):
    """
    Train and evaluate Gaussian HMM for multiple train-test splits.
    Only plots results for the best test size (highest accuracy).
    """
    best_accuracy = 0
    best_test_size = None
    best_cm = None
    best_class_report = None
    best_fpr = None
    best_tpr = None
    best_auc = None

    # Label Encoding
    le = LabelEncoder()
    Y_encoded = le.fit_transform(Y)  # Convert string labels to
numeric labels

    # Store accuracies for plotting
    accuracies = []

    for test_size in test_sizes:
        print(f"Running Gaussian HMM with test size {test_size}...")

        # Split the data with a fixed random state for reproducibility
        X_train, X_test, Y_train, Y_test = train_test_split(X,
Y_encoded, test_size=test_size, random_state=42)

        # Train Gaussian HMM with fixed random_state
        model = GaussianHMM(n_components=2, covariance_type="full",
n_iter=1000, random_state=42)
        model.fit(X_train)
        Y_pred = model.predict(X_test)
```

```
        # Calculate accuracy and confusion matrix
        accuracy = accuracy_score(Y_test, Y_pred)
        cm = confusion_matrix(Y_test, Y_pred)
        class_report = classification_report(Y_test, Y_pred)

        # Store the accuracy for comparison
        accuracies.append(accuracy)

        # Track the best performing test size
        if accuracy > best_accuracy:
            best_accuracy = accuracy
            best_test_size = test_size
            best_cm = cm
            best_class_report = class_report

            # --- ROC Curve and AUC Calculation ---
            Y_pred_proba = model.predict_proba(X_test)
            class_labels = np.unique(Y_encoded)

            Y_test_bin = label_binarize(Y_test, classes=class_labels)
            fpr, tpr, _ = roc_curve(Y_test_bin[:, 0], Y_pred_proba[:,
0])
            auc_value = roc_auc_score(Y_test_bin[:, 0],
Y_pred_proba[:, 0])

            best_fpr = fpr
            best_tpr = tpr
            best_auc = auc_value

        print(f"Accuracy: {accuracy}")
        print(f"Classification Report:\n{class_report}")

print("-----------------------------------------------------------------------------------------------")

    # Plotting the results for the best test size
    print(f"Best Test Size: {best_test_size} with Accuracy:
{best_accuracy}")

    # Plot confusion matrix
    print("plotting heatmap.........")
    plot_confusion_matrix(best_cm, "Gaussian HMM")

print("------------------------------------------------------------------------------------------------------------------")

    # Plot ROC Curve and AUC
    print("plotting ROC-AUC curve.........")
    plot_roc_curve(best_fpr, best_tpr, best_auc, "Gaussian HMM")
```

```
print("-----------------------------------------------------------------------------------------------------------------")

    # Accuracy bar graph
    print("plotting bargraph.........")
    plot_accuracy_bar_graph(accuracies, test_sizes, "Gaussian HMM")

print("-----------------------------------------------------------------------------------------------------------------")
```

Multinomial HMM

```
from hmmlearn.hmm import MultinomialHMM
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import label_binarize, LabelEncoder
from sklearn.preprocessing import KBinsDiscretizer
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np
from sklearn.metrics import accuracy_score, confusion_matrix,
classification_report, roc_curve, roc_auc_score

# Function to discretize data
def discretize_data(X, bins=5):
    """Discretizes continuous features into bins."""
    # Ensure that the discretizer returns integer values.
    discretizer = KBinsDiscretizer(n_bins=bins, encode='ordinal',
strategy='uniform')
    X_discretized = discretizer.fit_transform(X)

    # Check if the result is indeed integers
    X_discretized = np.floor(X_discretized).astype(int)  # Floor to
ensure integer type

    # Ensure no negative values exist
    if np.any(X_discretized < 0):
        print("Error: Discretized data contains negative values.")
        return None

    return X_discretized

# Function for Multinomial HMM with discretized data
def train_multinomial_hmm(X, Y, test_sizes=[0.2, 0.3, 0.4, 0.5]):
    """
    Train and evaluate Multinomial HMM for multiple train-test splits.
    Only plots results for the best test size (highest accuracy).
    """
    best_accuracy = 0
    best_test_size = None
```

```
    best_cm = None
    best_class_report = None
    best_fpr = None
    best_tpr = None
    best_auc = None

    # Label Encoding
    le = LabelEncoder()
    Y_encoded = le.fit_transform(Y)  # Convert string labels to
numeric labels

    # Discretize the data
    X_discretized = discretize_data(X)

    if X_discretized is None:
        print("Error: Failed to discretize the data. Exiting.")
        return

    # Store accuracies for plotting
    accuracies = []

    for test_size in test_sizes:
        print(f"Running Multinomial HMM with test size
{test_size}...")

        # Split the data
        X_train, X_test, Y_train, Y_test =
train_test_split(X_discretized, Y_encoded, test_size=test_size,
random_state=89)

        # Train Multinomial HMM
        # The number of components should be equal to the number of
unique classes
        model = MultinomialHMM(n_components=len(np.unique(Y_encoded)),
n_iter=1000, random_state=89)
        model.fit(X_train)
        Y_pred = model.predict(X_test)

        # Calculate accuracy and confusion matrix
        accuracy = accuracy_score(Y_test, Y_pred)
        cm = confusion_matrix(Y_test, Y_pred)

        # Decode predicted labels back to original strings for
classification report
        Y_pred_decoded = le.inverse_transform(Y_pred)
        Y_test_decoded = le.inverse_transform(Y_test)
        class_report = classification_report(Y_test_decoded,
Y_pred_decoded)

        # Store the accuracy for comparison
```

```
        accuracies.append(accuracy)

        # Track the best performing test size
        if accuracy > best_accuracy:
            best_accuracy = accuracy
            best_test_size = test_size
            best_cm = cm
            best_class_report = class_report

            # --- ROC Curve and AUC Calculation ---

            Y_pred_proba = model.predict_proba(X_test)
            class_labels = np.unique(Y_encoded)

            Y_test_bin = label_binarize(Y_test, classes=class_labels)
            fpr, tpr, _ = roc_curve(Y_test_bin[:, 0], Y_pred_proba[:,
0])
            auc_value = roc_auc_score(Y_test_bin[:, 0],
Y_pred_proba[:, 0])

            best_fpr = fpr
            best_tpr = tpr
            best_auc = auc_value

        print(f"Accuracy: {accuracy}")
        print(f"Classification Report:\n{class_report}")

print("-----------------------------------------------------------------------------------------------")
    # Plotting the results for the best test size
    print(f"Best Test Size: {best_test_size} with Accuracy:
{best_accuracy}")

    # Plot confusion matrix
    print("plotting heatmap.........")
    plot_confusion_matrix(best_cm, "Multinomial HMM")

print("----------------------------------------------------------------------------------------------------------------")

    # Plot ROC Curve and AUC (only if AUC was calculated)
    if best_auc is not None:
        print("plotting ROC-AUC curve.........")
        plot_roc_curve(best_fpr, best_tpr, best_auc, "Multinomial
HMM")

print("----------------------------------------------------------------------------------------------------------------")
    else:
        print("ROC Curve and AUC not plotted: Only one class present
```

```
in the best test split.")

    # Accuracy bar graph
    print("plotting bargraph.........")
    plot_accuracy_bar_graph(accuracies, test_sizes, "Multinomial HMM")

print("-----------------------------------------------------------------
------------------------------------------------------")
```

Data Fetching

1. **Ionosphere dataset**

```
from ucimlrepo import fetch_ucirepo

# fetch dataset
ionosphere = fetch_ucirepo(id=52)

import pandas as pd
import numpy as np

# Convert features and targets into DataFrames
ionosphere_features = pd.DataFrame(ionosphere.data.features,
columns=ionosphere.data.feature_names)
ionosphere_targets = pd.DataFrame(ionosphere.data.targets,
columns=ionosphere.data.target_names)

# Optionally, combine features and target into a single DataFrame
ionosphere_dataset = pd.concat([ionosphere_features,
ionosphere_targets], axis=1)
ionosphere_dataset.head()
```


{"type":"dataframe","variable_name":"ionosphere_dataset"}


```
ionosphere_dataset.isnull().sum()
```

Attribute1	0
Attribute2	0
Attribute3	0
Attribute4	0
Attribute5	0
Attribute6	0
Attribute7	0
Attribute8	0
Attribute9	0
Attribute10	0
Attribute11	0

```
Attribute12    0
Attribute13    0
Attribute14    0
Attribute15    0
Attribute16    0
Attribute17    0
Attribute18    0
Attribute19    0
Attribute20    0
Attribute21    0
Attribute22    0
Attribute23    0
Attribute24    0
Attribute25    0
Attribute26    0
Attribute27    0
Attribute28    0
Attribute29    0
Attribute30    0
Attribute31    0
Attribute32    0
Attribute33    0
Attribute34    0
Class          0
dtype: int64
```

Data Preprocessing

```
from sklearn.preprocessing import StandardScaler

# Step 1: Initialize the scaler
scaler = StandardScaler()
# Step 2: Fit the scaler on the features and transform
scaled_features = scaler.fit_transform(ionosphere_features)
# Step 3: Convert scaled features back to DataFrame (to preserve
column names)
ionosphere_features_scaled = pd.DataFrame(scaled_features,
columns=ionosphere_features.columns)
```

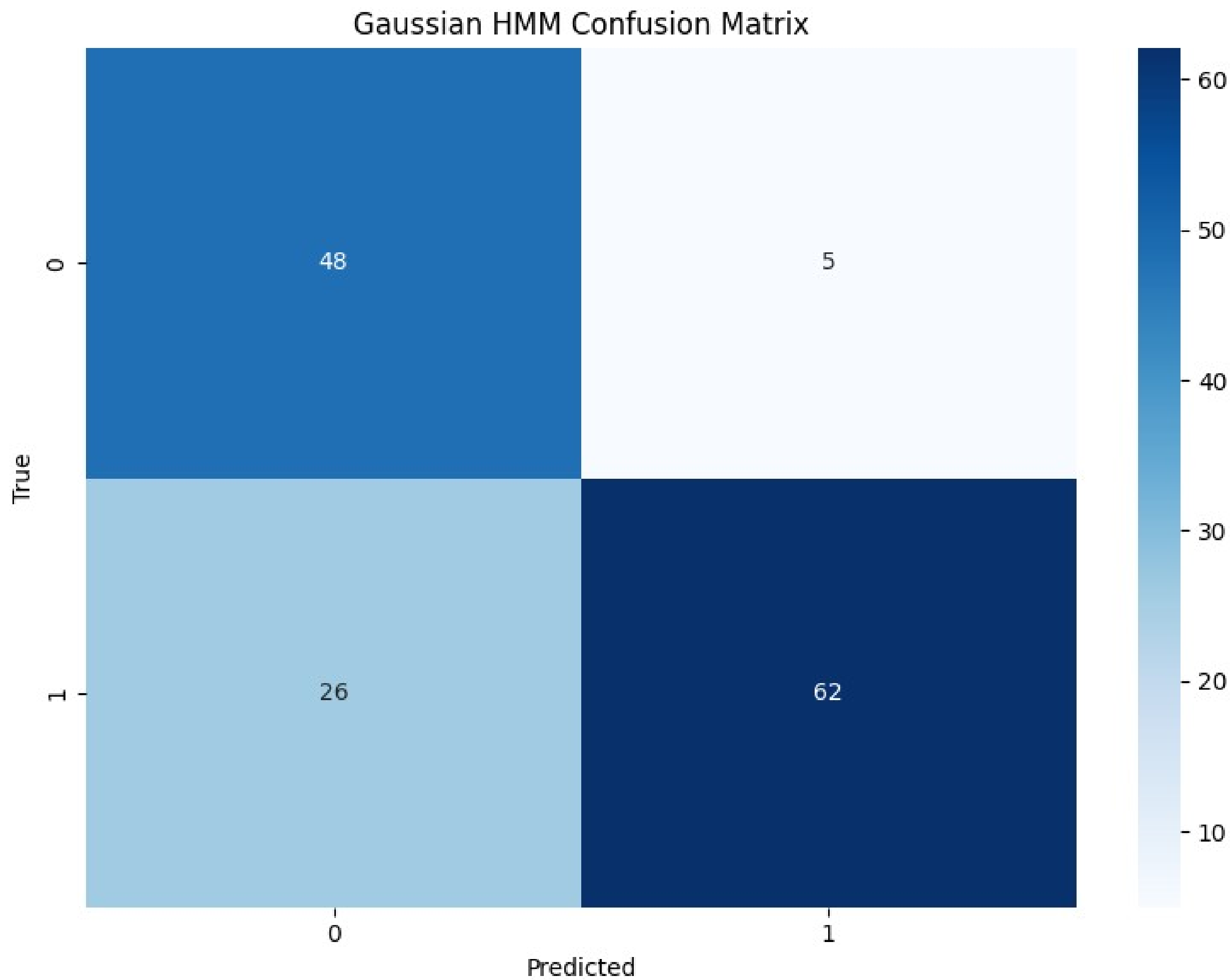
Gaussian HMM Classifier

```
train_gaussian_hmm(ionosphere_features_scaled,
ionosphere_targets['Class'], test_sizes=[0.2, 0.3, 0.4, 0.5])
```

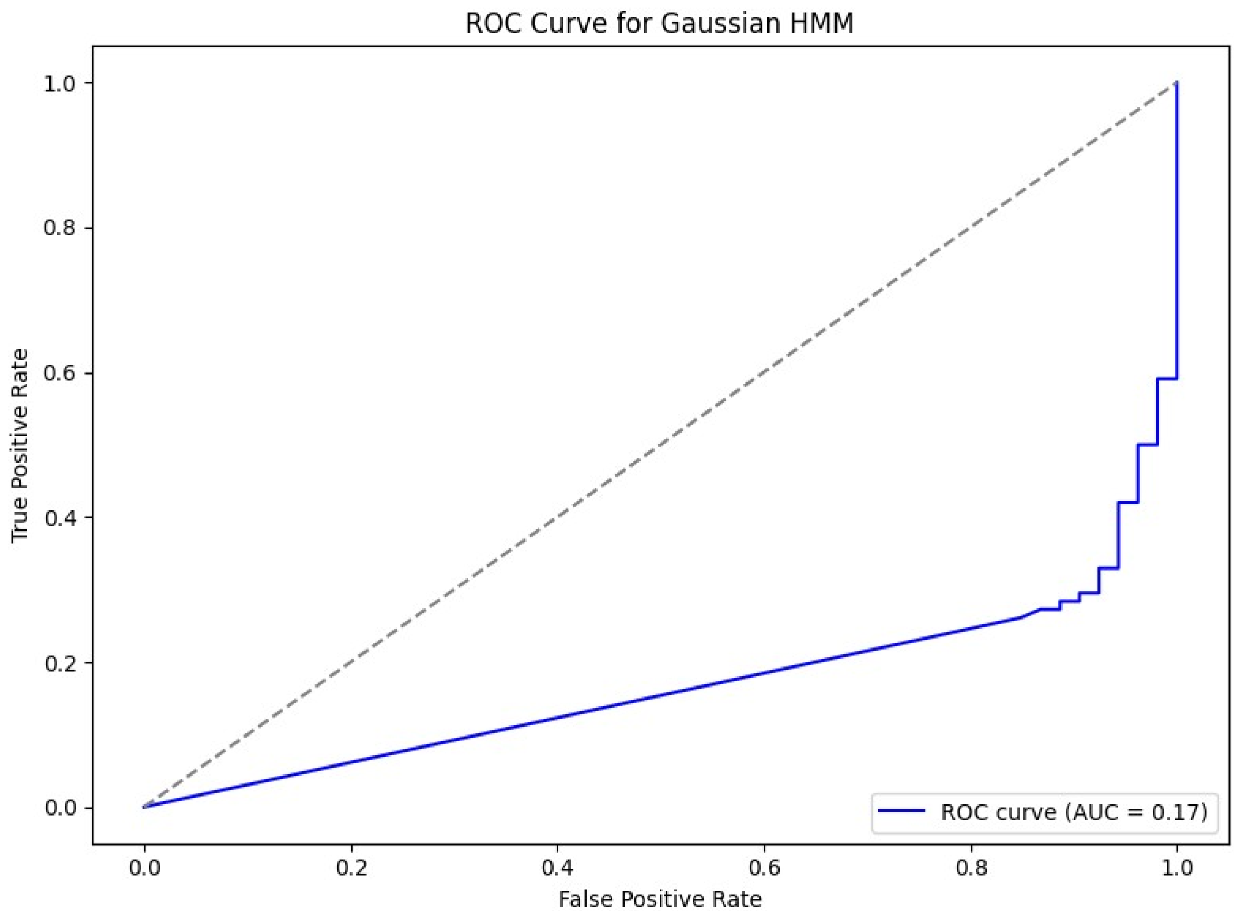
```
Running Gaussian HMM with test size 0.2...
Accuracy: 0.7605633802816901
Classification Report:
              precision    recall  f1-score   support

           0       0.65      0.86      0.74        28
```

```
           1       0.88      0.70      0.78        43

    accuracy                           0.76        71
   macro avg       0.77      0.78      0.76        71
weighted avg       0.79      0.76      0.76        71

----------------------------------------------------------------------
------------------------
Running Gaussian HMM with test size 0.3...
Accuracy: 0.1509433962264151
Classification Report:
              precision    recall  f1-score   support

           0       0.19      0.38      0.25        39
           1       0.04      0.01      0.02        67

    accuracy                           0.15       106
   macro avg       0.11      0.20      0.14       106
weighted avg       0.09      0.15      0.11       106

----------------------------------------------------------------------
------------------------
Running Gaussian HMM with test size 0.4...
Accuracy: 0.7801418439716312
Classification Report:
              precision    recall  f1-score   support

           0       0.65      0.91      0.76        53
           1       0.93      0.70      0.80        88

    accuracy                           0.78       141
   macro avg       0.79      0.81      0.78       141
weighted avg       0.82      0.78      0.78       141

----------------------------------------------------------------------
------------------------
Running Gaussian HMM with test size 0.5...
Accuracy: 0.6931818181818182
Classification Report:
              precision    recall  f1-score   support

           0       0.55      0.92      0.69        65
           1       0.93      0.56      0.70       111

    accuracy                           0.69       176
   macro avg       0.74      0.74      0.69       176
weighted avg       0.79      0.69      0.69       176

----------------------------------------------------------------------
------------------------
```

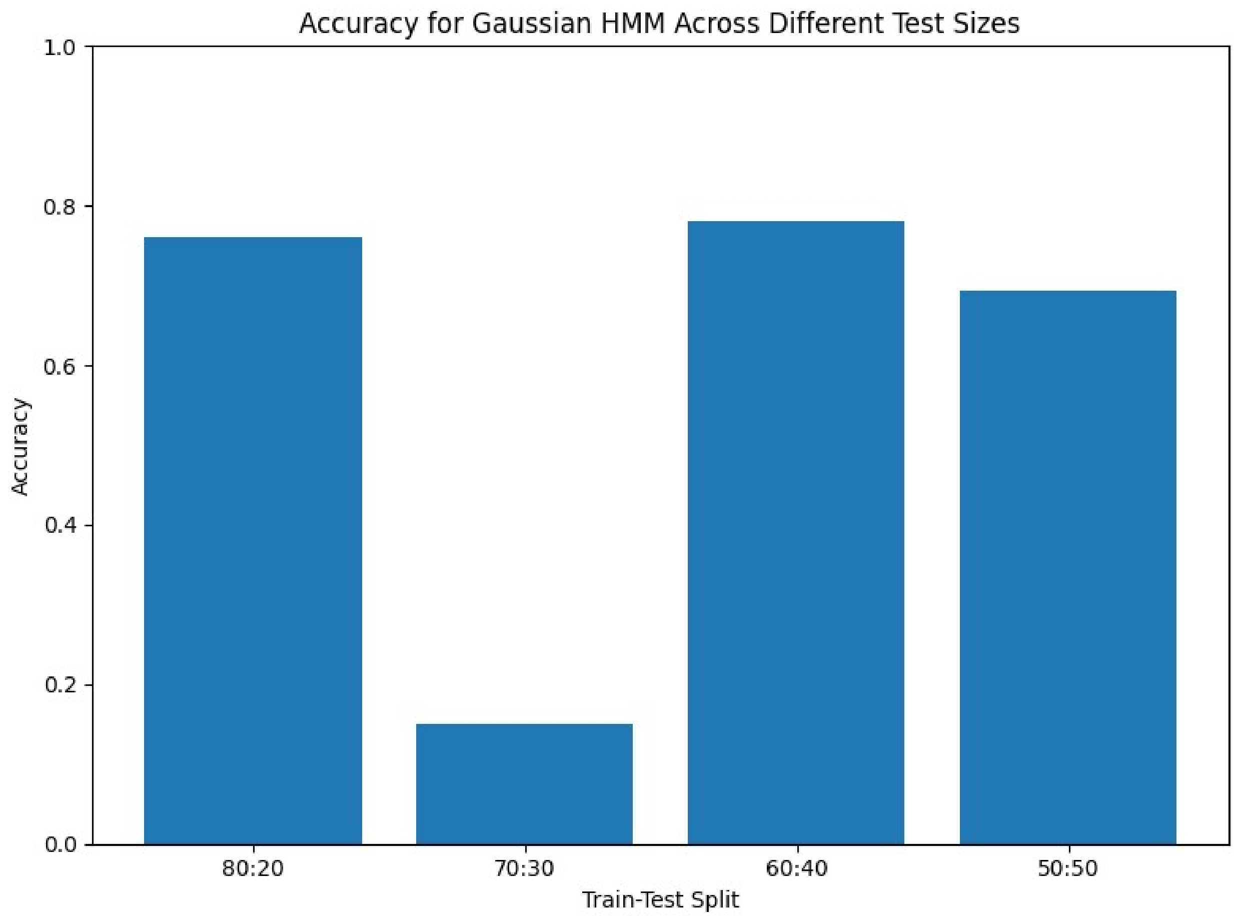
```
Best Test Size: 0.4 with Accuracy: 0.7801418439716312
plotting heatmap.........
```




```
----------------------------------------------------------------------
-----------------------------------------------
plotting ROC-AUC curve.........
```




```
------------------------------------------------------------------
--------------------------------------------
plotting bargraph.........
```




```
----------------------------------------------------------------------
----------------------------------------------
```

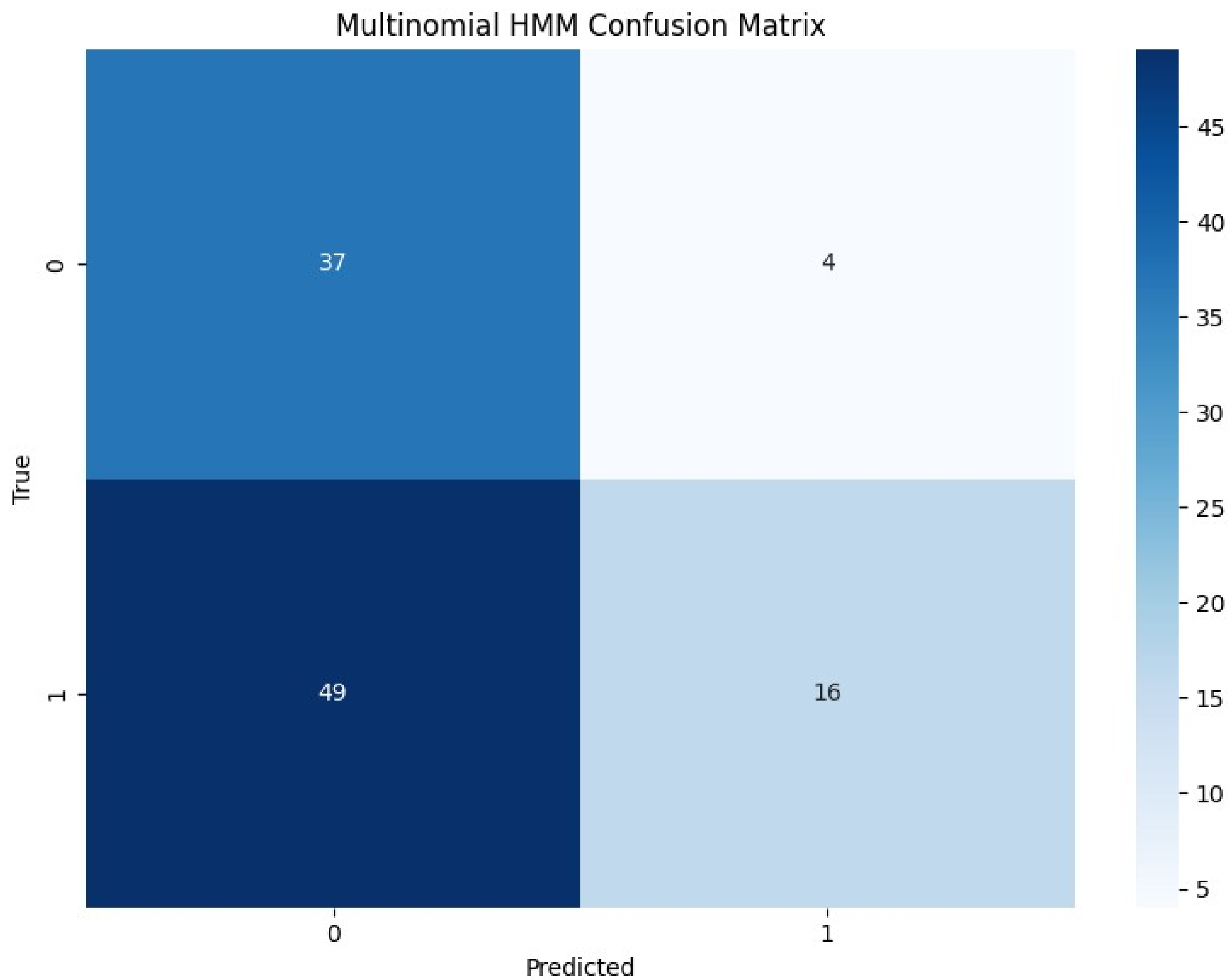
Multionomial HMM Classifier

```
train_multinomial_hmm(ionosphere_features_scaled,
ionosphere_targets['Class'], test_sizes=[0.2, 0.3, 0.4, 0.5])
```

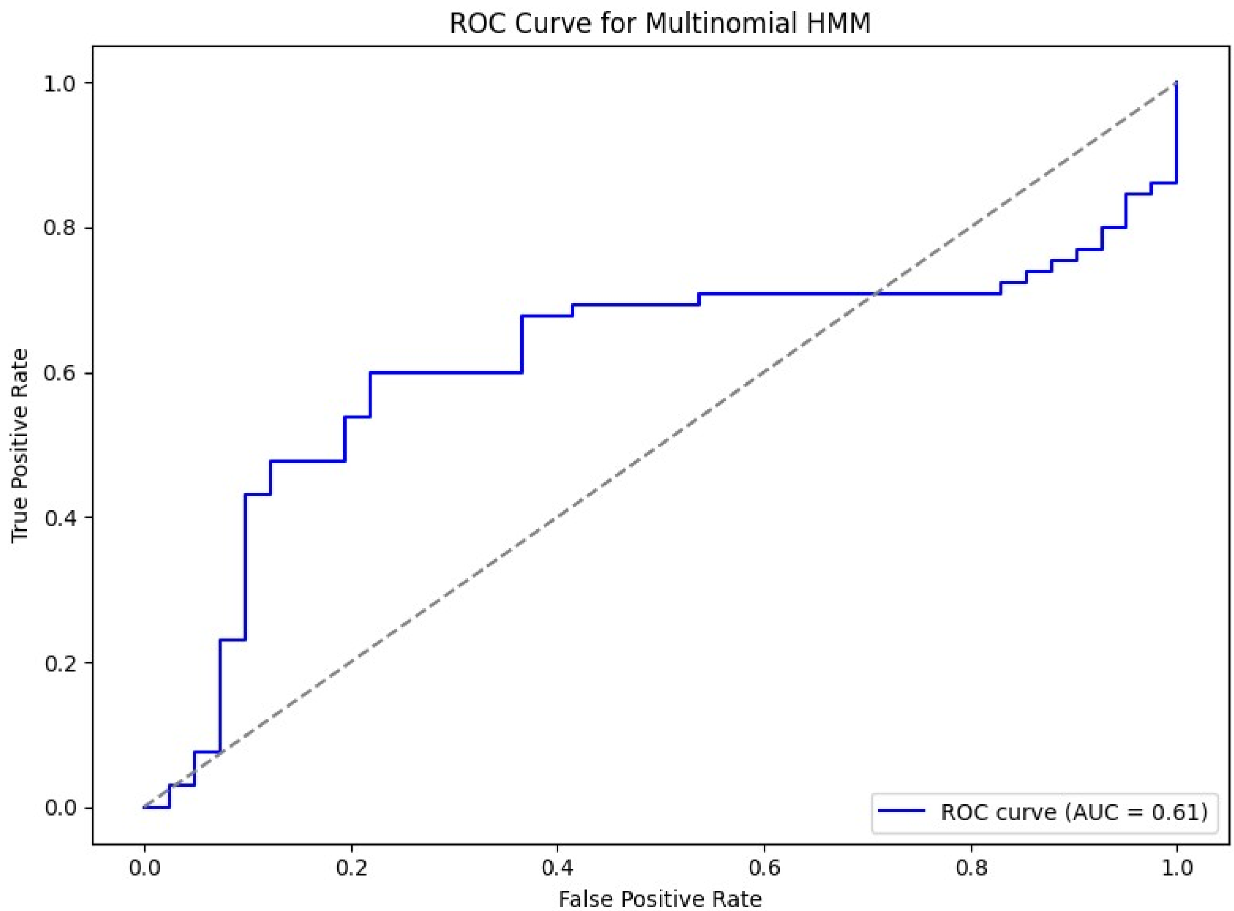
```
/usr/local/lib/python3.12/dist-packages/sklearn/preprocessing/
_discretization.py:262: UserWarning: Feature 1 is constant and will be
replaced with 0.
  warnings.warn(
WARNING:hmmlearn.hmm:MultinomialHMM has undergone major changes. The
previous version was implementing a CategoricalHMM (a special case of
MultinomialHMM). This new implementation follows the standard
definition for a Multinomial distribution (e.g. as in
https://en.wikipedia.org/wiki/Multinomial_distribution). See these
issues for details:
https://github.com/hmmlearn/hmmlearn/issues/335
https://github.com/hmmlearn/hmmlearn/issues/340
WARNING:hmmlearn.hmm:MultinomialHMM has undergone major changes. The
previous version was implementing a CategoricalHMM (a special case of
```

```
MultinomialHMM). This new implementation follows the standard
definition for a Multinomial distribution (e.g. as in
https://en.wikipedia.org/wiki/Multinomial_distribution). See these
issues for details:
https://github.com/hmmlearn/hmmlearn/issues/335
https://github.com/hmmlearn/hmmlearn/issues/340
WARNING:hmmlearn.hmm:MultinomialHMM has undergone major changes. The
previous version was implementing a CategoricalHMM (a special case of
MultinomialHMM). This new implementation follows the standard
definition for a Multinomial distribution (e.g. as in
https://en.wikipedia.org/wiki/Multinomial_distribution). See these
issues for details:
https://github.com/hmmlearn/hmmlearn/issues/335
https://github.com/hmmlearn/hmmlearn/issues/340
WARNING:hmmlearn.hmm:MultinomialHMM has undergone major changes. The
previous version was implementing a CategoricalHMM (a special case of
MultinomialHMM). This new implementation follows the standard
definition for a Multinomial distribution (e.g. as in
https://en.wikipedia.org/wiki/Multinomial_distribution). See these
issues for details:
https://github.com/hmmlearn/hmmlearn/issues/335
https://github.com/hmmlearn/hmmlearn/issues/340

Running Multinomial HMM with test size 0.2...
Accuracy: 0.4647887323943662
Classification Report:
              precision    recall  f1-score   support

           b       0.42      0.89      0.57        28
           g       0.73      0.19      0.30        43

    accuracy                           0.46        71
   macro avg       0.57      0.54      0.43        71
weighted avg       0.60      0.46      0.40        71

----------------------------------------------------------------------
------------------------
Running Multinomial HMM with test size 0.3...
Accuracy: 0.5
Classification Report:
              precision    recall  f1-score   support

           b       0.43      0.90      0.58        41
           g       0.80      0.25      0.38        65

    accuracy                           0.50       106
   macro avg       0.62      0.57      0.48       106
weighted avg       0.66      0.50      0.46       106

----------------------------------------------------------------------
```

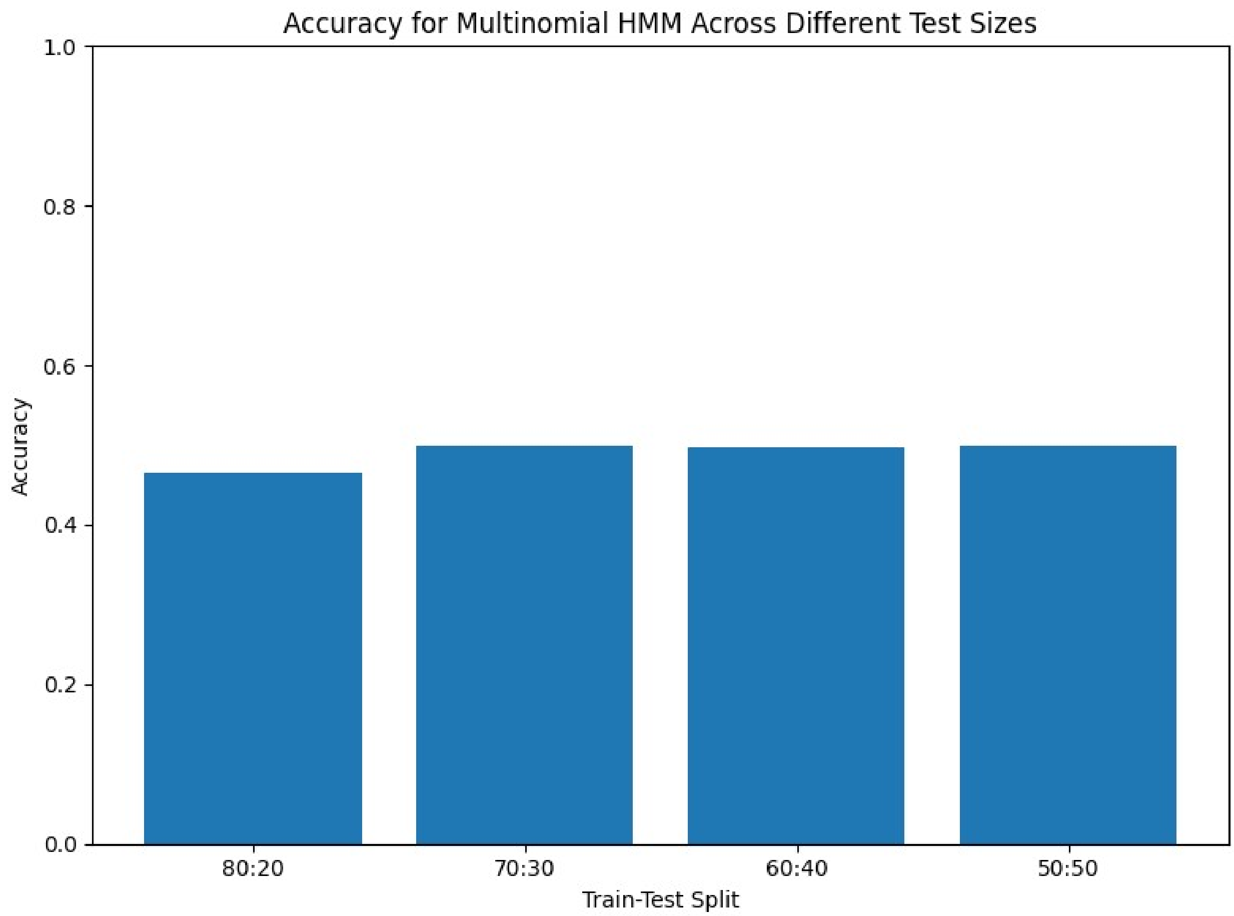
```
------------------------
Running Multinomial HMM with test size 0.4...
Accuracy: 0.49645390070921985
Classification Report:
              precision    recall  f1-score   support

           b       0.43      0.93      0.59        55
           g       0.83      0.22      0.35        86

    accuracy                           0.50       141
   macro avg       0.63      0.57      0.47       141
weighted avg       0.67      0.50      0.44       141

----------------------------------------------------------------------
------------------------
Running Multinomial HMM with test size 0.5...
Accuracy: 0.5
Classification Report:
              precision    recall  f1-score   support

           b       0.44      0.93      0.60        70
           g       0.82      0.22      0.34       106

    accuracy                           0.50       176
   macro avg       0.63      0.57      0.47       176
weighted avg       0.67      0.50      0.44       176

----------------------------------------------------------------------
------------------------
Best Test Size: 0.3 with Accuracy: 0.5
plotting heatmap.........
```




```
-------------------------------------------------------------------------------------------------------------------
plotting ROC-AUC curve.........
```




```
-----------------------------------------------------------------------------------------------------------
plotting bargraph.........
```




```
----------------------------------------------------------------------
-----------------------------------------------
```

2. **Wisconsin Breast Cancer Dataset:**

```
from ucimlrepo import fetch_ucirepo

# fetch dataset
breast_cancer_wisconsin_diagnostic = fetch_ucirepo(id=17)

import pandas as pd
import numpy as np

# Convert features and targets into DataFrames
cancer_features =
pd.DataFrame(breast_cancer_wisconsin_diagnostic.data.features,
columns=breast_cancer_wisconsin_diagnostic.data.feature_names)
cancer_targets =
pd.DataFrame(breast_cancer_wisconsin_diagnostic.data.targets,
columns=breast_cancer_wisconsin_diagnostic.data.target_names)
```

```
# Optionally, combine features and target into a single DataFrame
cancer_dataset = pd.concat([cancer_features, cancer_targets], axis=1)
cancer_dataset.head()
```

{"type":"dataframe","variable_name":"cancer_dataset"}

```
cancer_dataset.isnull().sum()
```

```
radius1               0
texture1              0
perimeter1            0
area1                 0
smoothness1           0
compactness1          0
concavity1            0
concave_points1       0
symmetry1             0
fractal_dimension1    0
radius2               0
texture2              0
perimeter2            0
area2                 0
smoothness2           0
compactness2          0
concavity2            0
concave_points2       0
symmetry2             0
fractal_dimension2    0
radius3               0
texture3              0
perimeter3            0
area3                 0
smoothness3           0
compactness3          0
concavity3            0
concave_points3       0
symmetry3             0
fractal_dimension3    0
Diagnosis             0
dtype: int64
```

Data Preprocessing

```
from sklearn.preprocessing import StandardScaler

# Step 1: Initialize the scaler
scaler = StandardScaler()
# Step 2: Fit the scaler on the features and transform
scaled_features = scaler.fit_transform(cancer_features)
# Step 3: Convert scaled features back to DataFrame (to preserve
```

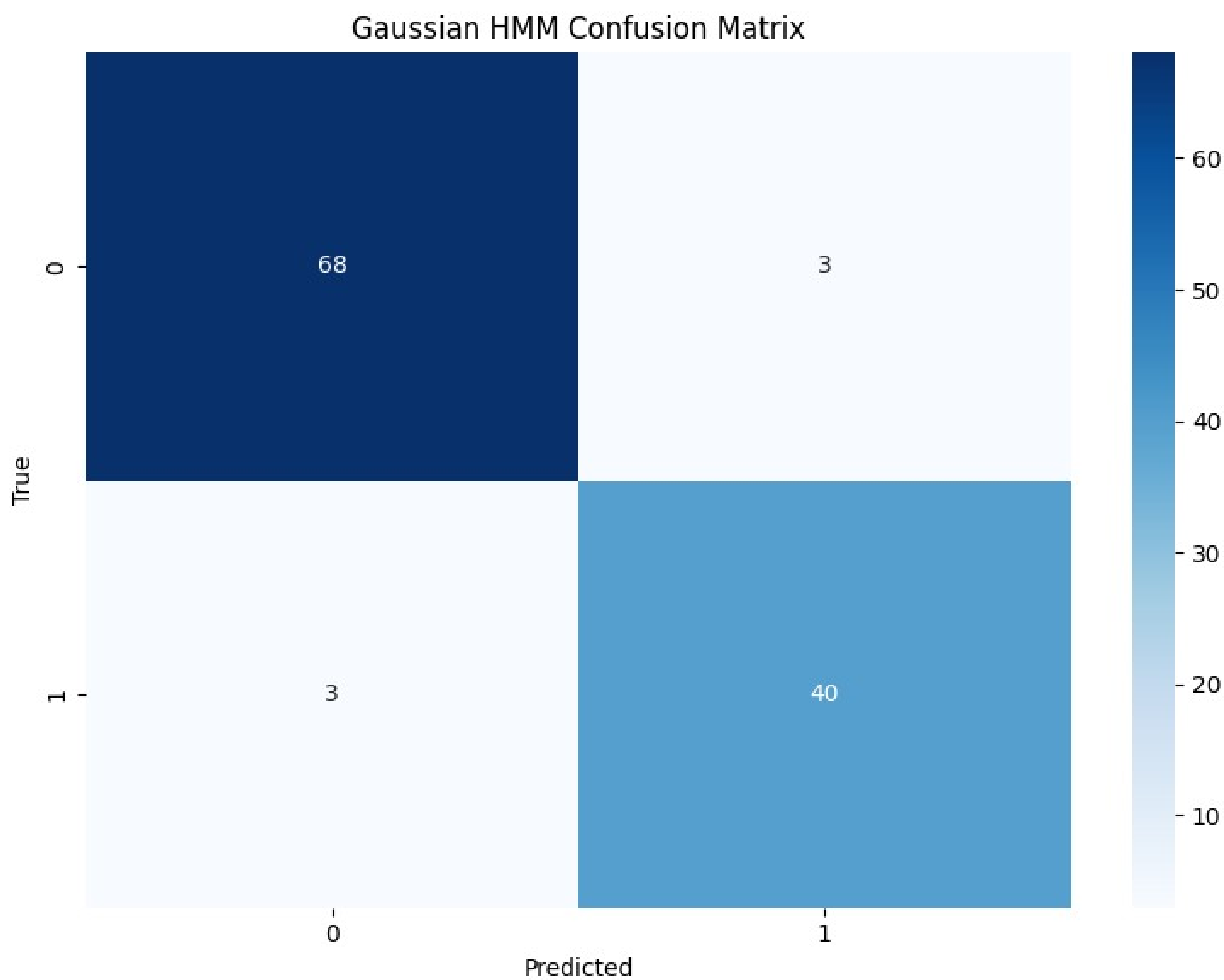
```
column names)
cancer_features_scaled = pd.DataFrame(scaled_features,
columns=cancer_features.columns)
```

Gaussian HMM Classifier

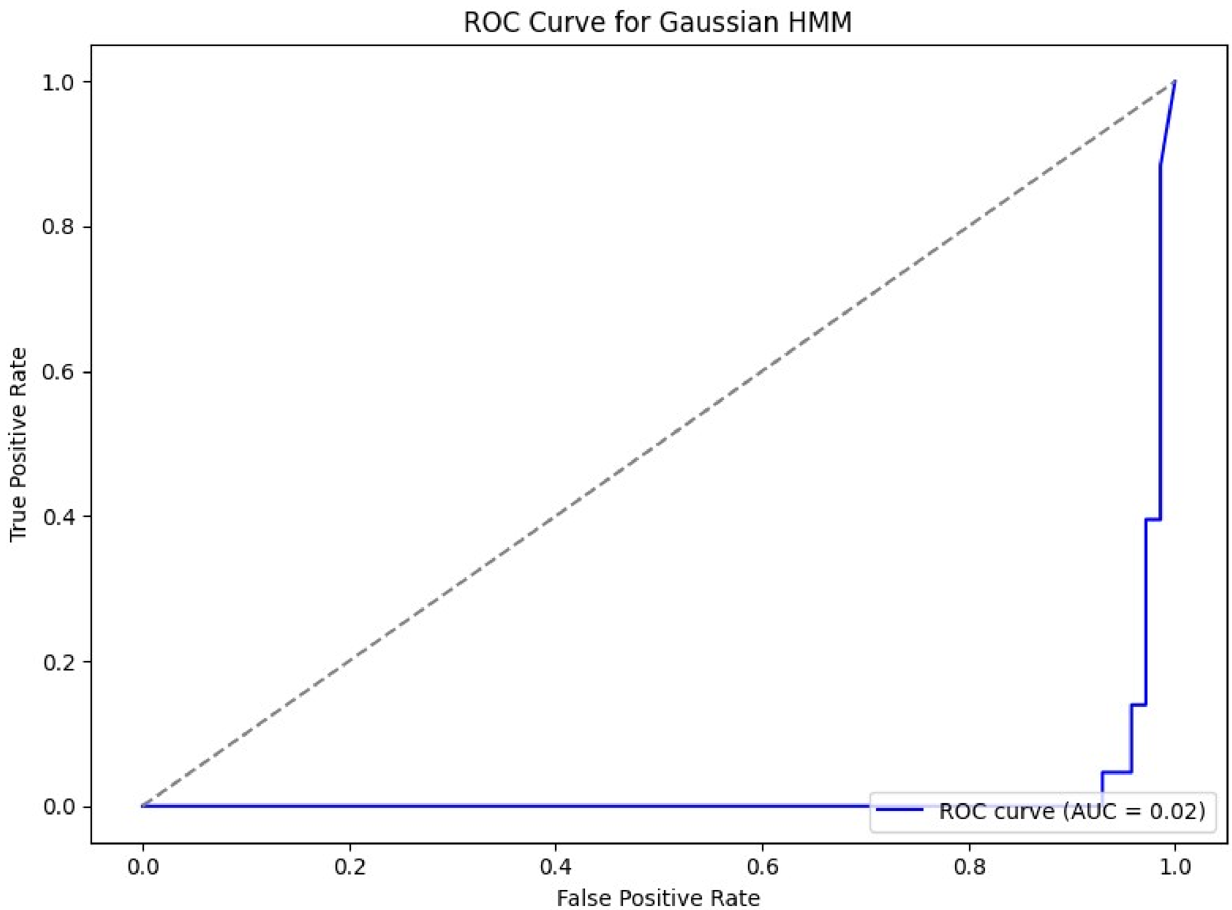
```
train_gaussian_hmm(cancer_features_scaled,
cancer_targets['Diagnosis'], test_sizes=[0.2, 0.3, 0.4, 0.5])
```

```
Running Gaussian HMM with test size 0.2...
Accuracy: 0.9473684210526315
Classification Report:
              precision    recall  f1-score   support

           0       0.96      0.96      0.96        71
           1       0.93      0.93      0.93        43

    accuracy                           0.95       114
   macro avg       0.94      0.94      0.94       114
weighted avg       0.95      0.95      0.95       114

----------------------------------------------------------------------
------------------------
Running Gaussian HMM with test size 0.3...
Accuracy: 0.10526315789473684
Classification Report:
              precision    recall  f1-score   support

           0       0.19      0.13      0.15       108
           1       0.04      0.06      0.05        63

    accuracy                           0.11       171
   macro avg       0.12      0.10      0.10       171
weighted avg       0.14      0.11      0.12       171

----------------------------------------------------------------------
------------------------
Running Gaussian HMM with test size 0.4...
Accuracy: 0.9473684210526315
Classification Report:
              precision    recall  f1-score   support

           0       0.96      0.96      0.96       148
           1       0.93      0.93      0.93        80

    accuracy                           0.95       228
   macro avg       0.94      0.94      0.94       228
weighted avg       0.95      0.95      0.95       228

----------------------------------------------------------------------
```

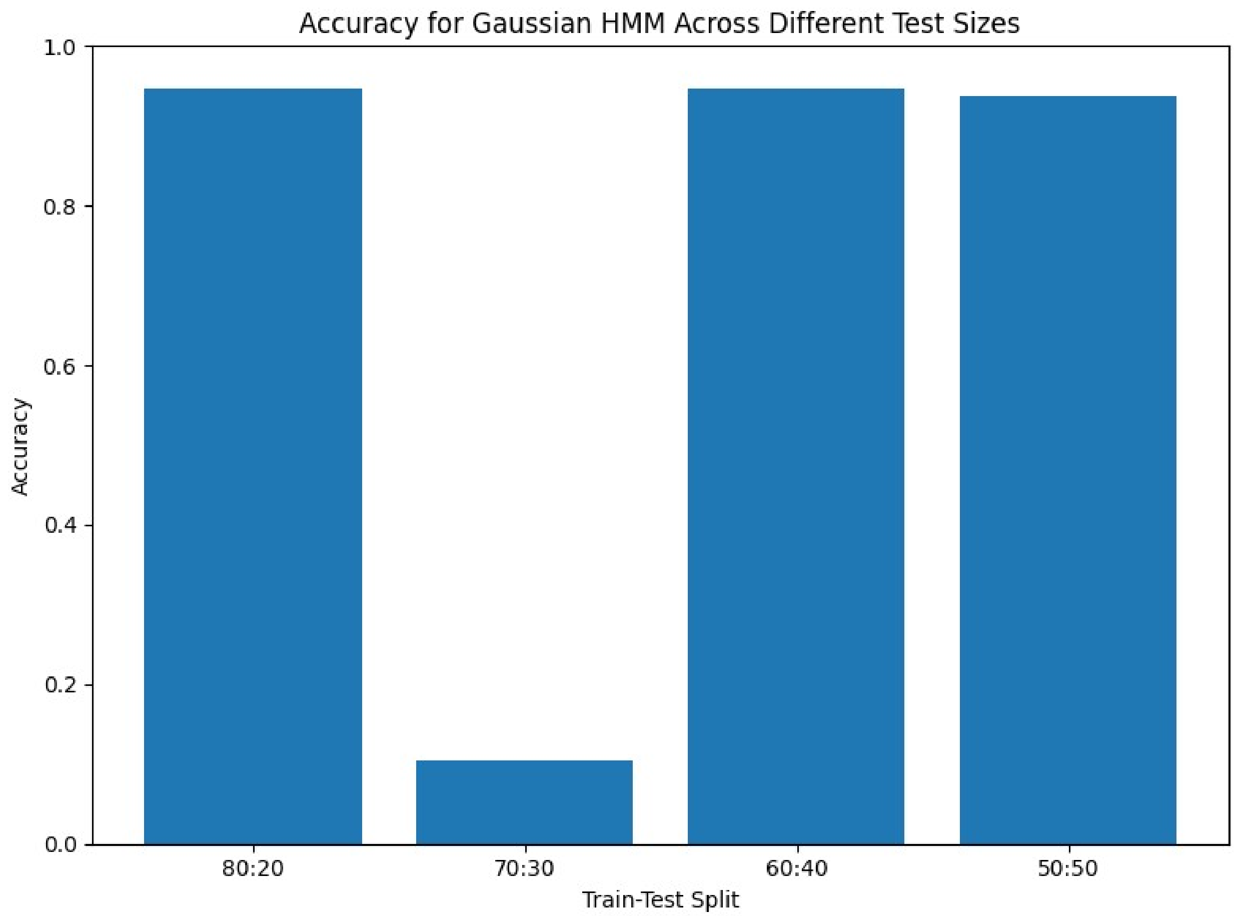
```
------------------------
Running Gaussian HMM with test size 0.5...
Accuracy: 0.9368421052631579
Classification Report:
              precision    recall  f1-score   support

           0       0.92      0.99      0.95       187
           1       0.98      0.84      0.90        98

    accuracy                           0.94       285
   macro avg       0.95      0.91      0.93       285
weighted avg       0.94      0.94      0.94       285

----------------------------------------------------------------------
------------------------
Best Test Size: 0.2 with Accuracy: 0.9473684210526315
plotting heatmap.........
```



```
----------------------------------------------------------------------
-----------------------------------------------
plotting ROC-AUC curve.........
```




```
----------------------------------------------------------------------
-----------------------------------------------
plotting bargraph.........
```




```
----------------------------------------------------------------------
----------------------------------------------
```

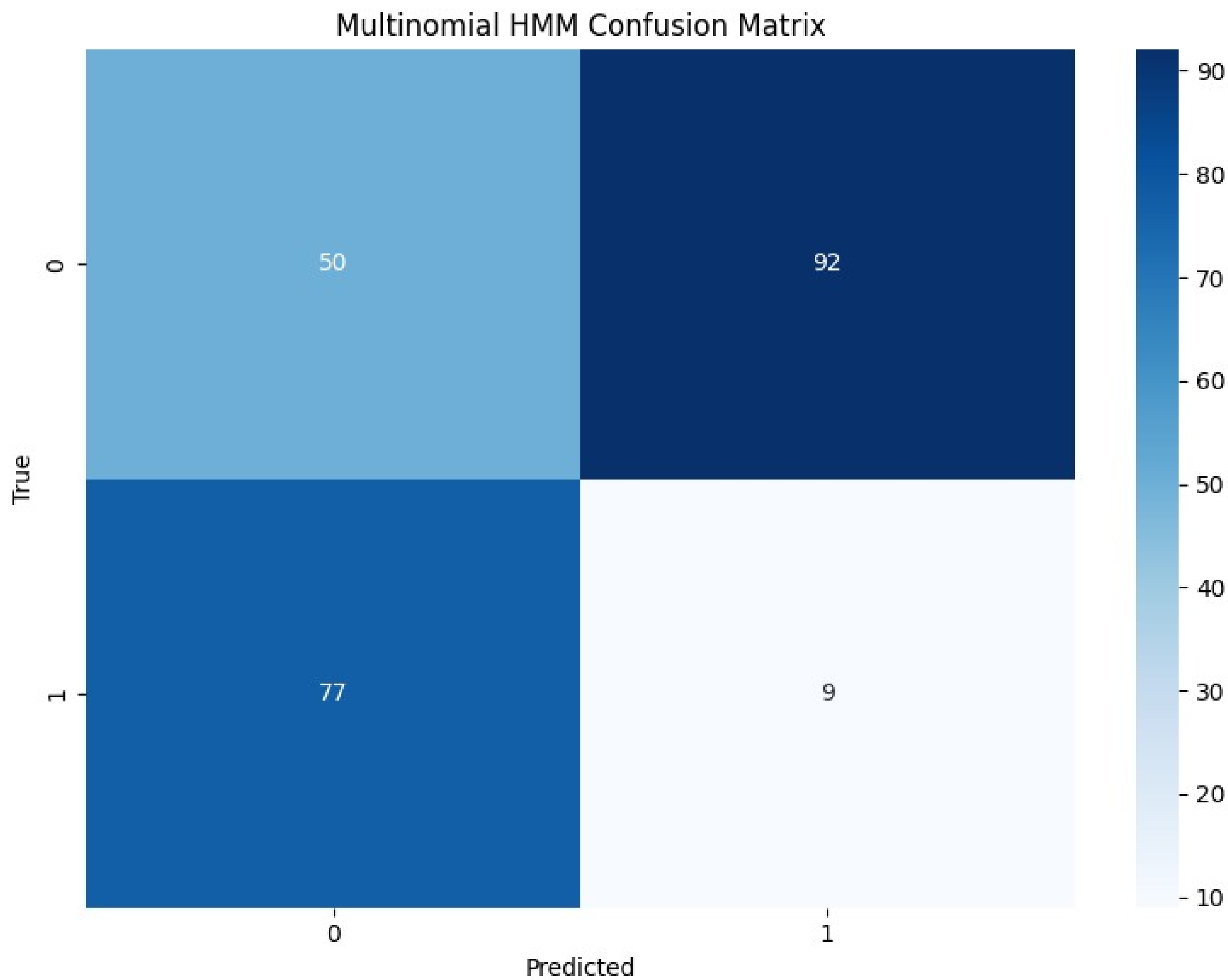
Multinomial HMM Classifier

```
train_multinomial_hmm(cancer_features_scaled,
cancer_targets['Diagnosis'], test_sizes=[0.2, 0.3, 0.4, 0.5])
```

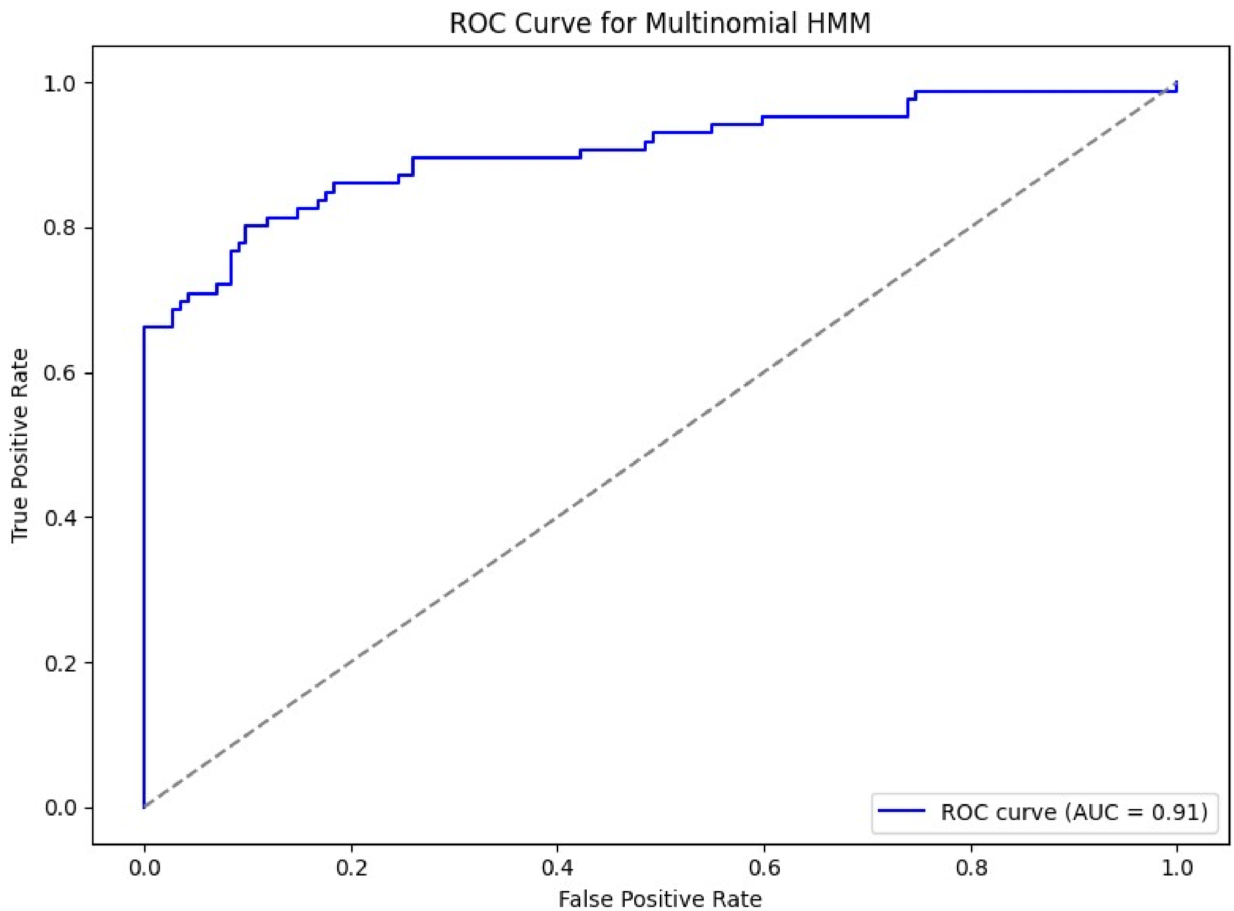
```
WARNING:hmmlearn.hmm:MultinomialHMM has undergone major changes. The
previous version was implementing a CategoricalHMM (a special case of
MultinomialHMM). This new implementation follows the standard
definition for a Multinomial distribution (e.g. as in
https://en.wikipedia.org/wiki/Multinomial_distribution). See these
issues for details:
https://github.com/hmmlearn/hmmlearn/issues/335
https://github.com/hmmlearn/hmmlearn/issues/340
WARNING:hmmlearn.hmm:MultinomialHMM has undergone major changes. The
previous version was implementing a CategoricalHMM (a special case of
MultinomialHMM). This new implementation follows the standard
definition for a Multinomial distribution (e.g. as in
https://en.wikipedia.org/wiki/Multinomial_distribution). See these
issues for details:
```

```
https://github.com/hmmlearn/hmmlearn/issues/335
https://github.com/hmmlearn/hmmlearn/issues/340
WARNING:hmmlearn.hmm:MultinomialHMM has undergone major changes. The
previous version was implementing a CategoricalHMM (a special case of
MultinomialHMM). This new implementation follows the standard
definition for a Multinomial distribution (e.g. as in
https://en.wikipedia.org/wiki/Multinomial_distribution). See these
issues for details:
https://github.com/hmmlearn/hmmlearn/issues/335
https://github.com/hmmlearn/hmmlearn/issues/340

Running Multinomial HMM with test size 0.2...
Accuracy: 0.17543859649122806
Classification Report:
              precision    recall  f1-score   support

           B       0.27      0.24      0.25        67
           M       0.07      0.09      0.08        47

    accuracy                           0.18       114
   macro avg       0.17      0.16      0.17       114
weighted avg       0.19      0.18      0.18       114

----------------------------------------------------------------------
------------------------
Running Multinomial HMM with test size 0.3...
Accuracy: 0.21052631578947367
Classification Report:
              precision    recall  f1-score   support

           B       0.33      0.25      0.29       107
           M       0.10      0.14      0.12        64

    accuracy                           0.21       171
   macro avg       0.22      0.20      0.20       171
weighted avg       0.24      0.21      0.22       171

----------------------------------------------------------------------
------------------------
Running Multinomial HMM with test size 0.4...

WARNING:hmmlearn.hmm:MultinomialHMM has undergone major changes. The
previous version was implementing a CategoricalHMM (a special case of
MultinomialHMM). This new implementation follows the standard
definition for a Multinomial distribution (e.g. as in
https://en.wikipedia.org/wiki/Multinomial_distribution). See these
issues for details:
https://github.com/hmmlearn/hmmlearn/issues/335
https://github.com/hmmlearn/hmmlearn/issues/340
```

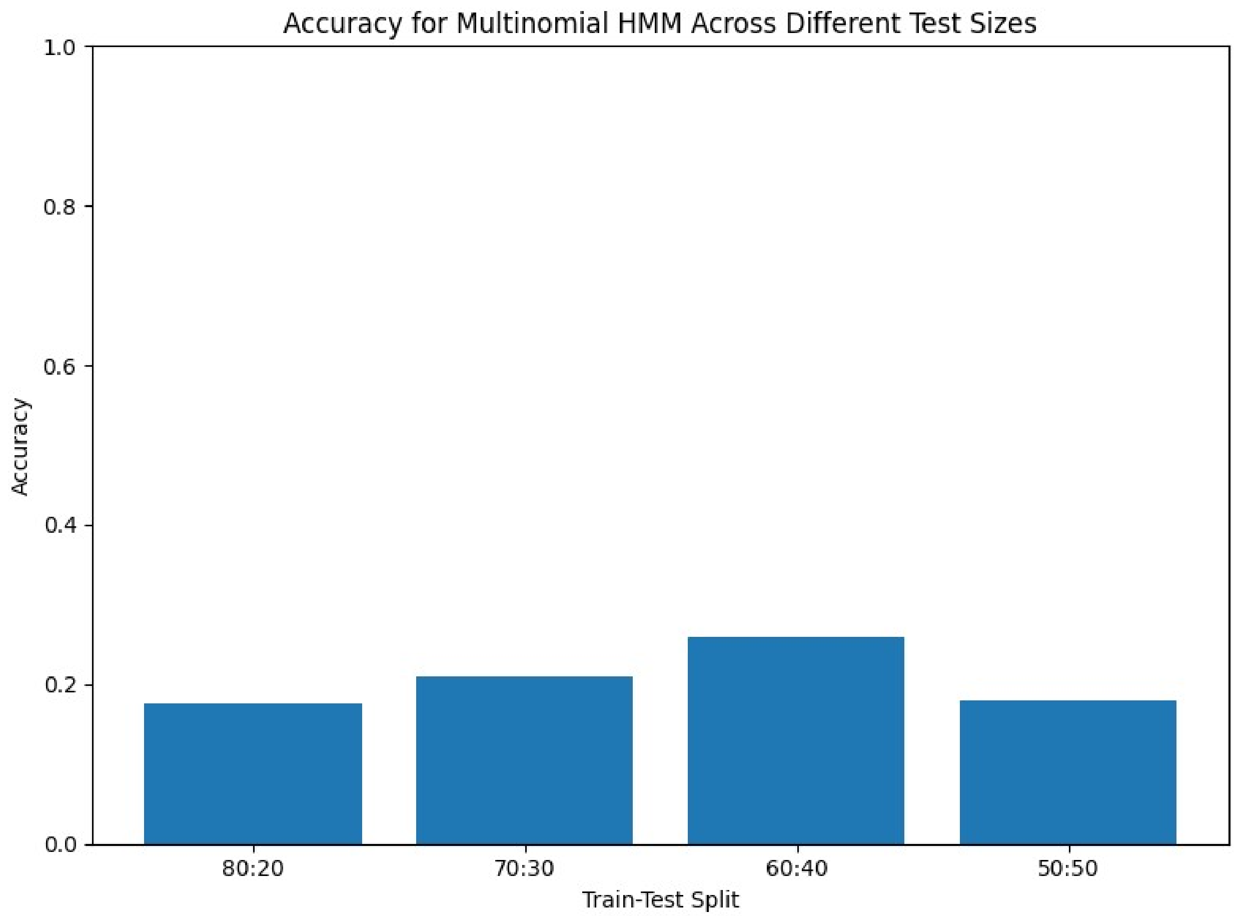
```
Accuracy: 0.25877192982456143
Classification Report:
              precision    recall  f1-score   support

           B       0.39      0.35      0.37       142
           M       0.09      0.10      0.10        86

    accuracy                           0.26       228
   macro avg       0.24      0.23      0.23       228
weighted avg       0.28      0.26      0.27       228

----------------------------------------------------------------------------------------------------
Running Multinomial HMM with test size 0.5...
Accuracy: 0.17894736842105263
Classification Report:
              precision    recall  f1-score   support

           B       0.27      0.21      0.24       173
           M       0.10      0.13      0.11       112

    accuracy                           0.18       285
   macro avg       0.18      0.17      0.17       285
weighted avg       0.20      0.18      0.19       285

----------------------------------------------------------------------------------------------------
Best Test Size: 0.4 with Accuracy: 0.25877192982456143
plotting heatmap.........
```




```
---------------------------------------------------------------------
-----------------------------------------------
plotting ROC-AUC curve.........
```




```
-------------------------------------------------------------------------------------------------------------------
plotting bargraph.........
```




```
--------------------------------------------------------------------------------------------------------------------------------
```

CNN

CIFAR-10

```
import tensorflow as tf
from tensorflow import keras
from keras import datasets

(train_images, train_labels), (test_images, test_labels) = 
datasets.cifar10.load_data()

#normalization
train_images, test_images = train_images / 255.0, test_images / 255.0

print("CIFAR-10 dataset loaded successfully.")
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}")
```

```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-
python.tar.gz
170498071/170498071 ━━━━━━━━━━━━━━━━━━━━ 14s 0us/step
CIFAR-10 dataset loaded successfully.
Train images shape: (50000, 32, 32, 3)
Train labels shape: (50000, 1)
Test images shape: (10000, 32, 32, 3)
Test labels shape: (10000, 1)
```

CNN model

```
# Import necessary libraries from TensorFlow/Keras
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten,
Dense, Dropout, BatchNormalization
from tensorflow.keras.optimizers import Adam

# Initialize the Sequential model
model = Sequential()

# First Convolutional Block
# Use padding='same' to maintain the spatial dimensions
model.add(Conv2D(32, kernel_size=(3, 3), activation='relu',
padding='same', input_shape=(32, 32, 3)))
model.add(BatchNormalization())
model.add(Conv2D(32, kernel_size=(3, 3), activation='relu',
padding='same'))
model.add(MaxPooling2D(pool_size=(2, 2))) # Changed pool_size to 2x2
model.add(Dropout(0.25))

# Second Convolutional Block
model.add(Conv2D(64, kernel_size=(3, 3), activation='relu',
padding='same'))
model.add(BatchNormalization())
model.add(Conv2D(64, kernel_size=(3, 3), activation='relu',
padding='same'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

# Third Convolutional Block
model.add(Conv2D(128, kernel_size=(3, 3), activation='relu',
padding='same'))
model.add(BatchNormalization())
model.add(Conv2D(128, kernel_size=(3, 3), activation='relu',
padding='same'))
```

```python
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

# Flatten Layer
model.add(Flatten())

# Fully Connected Layers
model.add(Dense(256, activation='relu'))
model.add(Dropout(0.25))

# Output Layer: 10 neurons with softmax activation
model.add(Dense(10, activation='softmax'))

# Compile the Model
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

# Print model summary
model.summary()
```

```
Model: "sequential_1"

┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━
━━━━━┓
┃ Layer (type)                      ┃ Output Shape           ┃
Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━
━━━━━┩
│ conv2d_3 (Conv2D)                 │ (None, 32, 32, 32)     │
896 │
├───────────────────────────────────┼────────────────────────┼──────────
─────┤
│ batch_normalization               │ (None, 32, 32, 32)     │
128 │
│ (BatchNormalization)              │                        │
│
├───────────────────────────────────┼────────────────────────┼──────────
─────┤
│ conv2d_4 (Conv2D)                 │ (None, 32, 32, 32)     │
9,248 │
├───────────────────────────────────┼────────────────────────┼──────────
─────┤
│ max_pooling2d_2 (MaxPooling2D)    │ (None, 16, 16, 32)     │
0 │
├───────────────────────────────────┼────────────────────────┼──────────
─────┤
│ dropout_1 (Dropout)               │ (None, 16, 16, 32)     │
0 │
├───────────────────────────────────┼────────────────────────┼──────────
```

```
───────┤
│ conv2d_5 (Conv2D)                 │ (None, 16, 16, 64)        │
18,496 │
├───────────────────────────────────┼───────────────────────────┼──────────
───────┤
│ batch_normalization_1             │ (None, 16, 16, 64)        │
256 │
│ (BatchNormalization)              │                           │
│
├───────────────────────────────────┼───────────────────────────┼──────────
───────┤
│ conv2d_6 (Conv2D)                 │ (None, 16, 16, 64)        │
36,928 │
├───────────────────────────────────┼───────────────────────────┼──────────
───────┤
│ max_pooling2d_3 (MaxPooling2D)    │ (None, 8, 8, 64)          │
0 │
├───────────────────────────────────┼───────────────────────────┼──────────
───────┤
│ dropout_2 (Dropout)               │ (None, 8, 8, 64)          │
0 │
├───────────────────────────────────┼───────────────────────────┼──────────
───────┤
│ conv2d_7 (Conv2D)                 │ (None, 8, 8, 128)         │
73,856 │
├───────────────────────────────────┼───────────────────────────┼──────────
───────┤
│ batch_normalization_2             │ (None, 8, 8, 128)         │
512 │
│ (BatchNormalization)              │                           │
│
├───────────────────────────────────┼───────────────────────────┼──────────
───────┤
│ conv2d_8 (Conv2D)                 │ (None, 8, 8, 128)         │
147,584 │
├───────────────────────────────────┼───────────────────────────┼──────────
───────┤
│ max_pooling2d_4 (MaxPooling2D)    │ (None, 4, 4, 128)         │
0 │
├───────────────────────────────────┼───────────────────────────┼──────────
───────┤
│ dropout_3 (Dropout)               │ (None, 4, 4, 128)         │
0 │
├───────────────────────────────────┼───────────────────────────┼──────────
───────┤
│ flatten_1 (Flatten)               │ (None, 2048)              │
0 │
├───────────────────────────────────┼───────────────────────────┼──────────
───────┤
```

```
│ dense_2 (Dense)                      │ (None, 256)               │
524,544 │
├──────────────────────────────────────┼───────────────────────────┼──────────
──────┤
│ dropout_4 (Dropout)                  │ (None, 256)               │
0 │
├──────────────────────────────────────┼───────────────────────────┼──────────
──────┤
│ dense_3 (Dense)                      │ (None, 10)                │
2,570 │
└──────────────────────────────────────┴───────────────────────────┴──────────
──────┘

 Total params: 815,018 (3.11 MB)

 Trainable params: 814,570 (3.11 MB)

 Non-trainable params: 448 (1.75 KB)
```

model compilation

```
# 1. Train the model
# Use the consistent variable names: x_train, y_train, x_test, y_test
history = model.fit(train_images, train_labels, epochs=30,
validation_data=(test_images, test_labels))

# 2. Evaluate the model
# Use the same consistent variable names for the evaluation step
test_loss, test_acc = model.evaluate(test_images, test_labels,
verbose=2)

print(f"\nTest Accuracy: {test_acc*100:.2f}%")
print(f"Test Loss: {test_loss:.4f}")

Epoch 1/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 25s 10ms/step - accuracy: 0.3762 -
loss: 1.7432 - val_accuracy: 0.5938 - val_loss: 1.1209
Epoch 2/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 9s 6ms/step - accuracy: 0.5940 - loss:
1.1271 - val_accuracy: 0.6912 - val_loss: 0.8887
Epoch 3/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 10s 6ms/step - accuracy: 0.6794 - loss:
0.9247 - val_accuracy: 0.7266 - val_loss: 0.8025
Epoch 4/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 11s 6ms/step - accuracy: 0.7189 - loss:
0.8196 - val_accuracy: 0.6203 - val_loss: 1.2245
Epoch 5/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 10s 6ms/step - accuracy: 0.7436 - loss:
0.7480 - val_accuracy: 0.7465 - val_loss: 0.7400
Epoch 6/30
```

```
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 9s 6ms/step - accuracy: 0.7681 - loss:
0.6774 - val_accuracy: 0.7527 - val_loss: 0.7318
Epoch 7/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 10s 6ms/step - accuracy: 0.7860 - loss:
0.6241 - val_accuracy: 0.7867 - val_loss: 0.6555
Epoch 8/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 9s 6ms/step - accuracy: 0.7994 - loss:
0.5884 - val_accuracy: 0.7997 - val_loss: 0.6269
Epoch 9/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 11s 6ms/step - accuracy: 0.8089 - loss:
0.5507 - val_accuracy: 0.7727 - val_loss: 0.6954
Epoch 10/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 10s 6ms/step - accuracy: 0.8202 - loss:
0.5175 - val_accuracy: 0.8064 - val_loss: 0.5987
Epoch 11/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 9s 6ms/step - accuracy: 0.8265 - loss:
0.5026 - val_accuracy: 0.8118 - val_loss: 0.5902
Epoch 12/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 10s 6ms/step - accuracy: 0.8347 - loss:
0.4845 - val_accuracy: 0.8028 - val_loss: 0.6015
Epoch 13/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 10s 6ms/step - accuracy: 0.8453 - loss:
0.4471 - val_accuracy: 0.8008 - val_loss: 0.6354
Epoch 14/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 9s 6ms/step - accuracy: 0.8504 - loss:
0.4317 - val_accuracy: 0.8062 - val_loss: 0.6259
Epoch 15/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 10s 6ms/step - accuracy: 0.8567 - loss:
0.4195 - val_accuracy: 0.8194 - val_loss: 0.5836
Epoch 16/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 9s 6ms/step - accuracy: 0.8552 - loss:
0.4110 - val_accuracy: 0.8303 - val_loss: 0.5565
Epoch 17/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 10s 6ms/step - accuracy: 0.8670 - loss:
0.3871 - val_accuracy: 0.8182 - val_loss: 0.5885
Epoch 18/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 9s 6ms/step - accuracy: 0.8703 - loss:
0.3724 - val_accuracy: 0.8334 - val_loss: 0.5491
Epoch 19/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 9s 6ms/step - accuracy: 0.8750 - loss:
0.3595 - val_accuracy: 0.8281 - val_loss: 0.5469
Epoch 20/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 10s 6ms/step - accuracy: 0.8811 - loss:
0.3440 - val_accuracy: 0.8244 - val_loss: 0.5802
Epoch 21/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 10s 6ms/step - accuracy: 0.8819 - loss:
0.3353 - val_accuracy: 0.8300 - val_loss: 0.5862
Epoch 22/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 10s 6ms/step - accuracy: 0.8860 - loss:
```

```
0.3334 - val_accuracy: 0.8163 - val_loss: 0.6240
Epoch 23/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 9s 6ms/step - accuracy: 0.8891 - loss:
0.3176 - val_accuracy: 0.8271 - val_loss: 0.5915
Epoch 24/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 9s 6ms/step - accuracy: 0.8876 - loss:
0.3156 - val_accuracy: 0.8336 - val_loss: 0.5605
Epoch 25/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 11s 6ms/step - accuracy: 0.8975 - loss:
0.2943 - val_accuracy: 0.8324 - val_loss: 0.5528
Epoch 26/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 10s 6ms/step - accuracy: 0.8981 - loss:
0.2953 - val_accuracy: 0.8377 - val_loss: 0.5311
Epoch 27/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 10s 6ms/step - accuracy: 0.9011 - loss:
0.2853 - val_accuracy: 0.8384 - val_loss: 0.5772
Epoch 28/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 10s 6ms/step - accuracy: 0.9034 - loss:
0.2755 - val_accuracy: 0.8381 - val_loss: 0.5472
Epoch 29/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 10s 6ms/step - accuracy: 0.9062 - loss:
0.2685 - val_accuracy: 0.8414 - val_loss: 0.5574
Epoch 30/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 10s 6ms/step - accuracy: 0.9081 - loss:
0.2625 - val_accuracy: 0.8455 - val_loss: 0.5617
313/313 - 1s - 3ms/step - accuracy: 0.8455 - loss: 0.5617

Test Accuracy: 84.55%
Test Loss: 0.5617
```

MNIST

```python
import tensorflow as tf
from tensorflow import keras
from keras import datasets

(mnist_train_images, mnist_train_labels), (mnist_test_images,
mnist_test_labels) = datasets.mnist.load_data()

#normalization
mnist_train_images = mnist_train_images.reshape(-1, 28, 28,
1).astype('float32') / 255.0
mnist_test_images = mnist_test_images.reshape(-1, 28, 28,
1).astype('float32') / 255.0

print("MNIST dataset loaded successfully.")
print(f"Train images shape: {mnist_train_images.shape}")
print(f"Train labels shape: {mnist_train_labels.shape}")
```

```
print(f"Test images shape: {mnist_test_images.shape}")
print(f"Test labels shape: {mnist_test_labels.shape}")
```

```
CIFAR-10 dataset loaded successfully.
Train images shape: (60000, 28, 28, 1)
Train labels shape: (60000,)
Test images shape: (10000, 28, 28, 1)
Test labels shape: (10000,)
```

```
# Import necessary libraries from TensorFlow/Keras
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten,
Dense, Dropout, BatchNormalization
from tensorflow.keras.optimizers import Adam

# Initialize the Sequential model
mnist_model = Sequential()

# First Convolutional Block
# Use padding='same' to maintain the spatial dimensions
mnist_model.add(Conv2D(32, kernel_size=(3, 3), activation='relu',
padding='same', input_shape=(28, 28, 1)))
mnist_model.add(BatchNormalization())
mnist_model.add(Conv2D(32, kernel_size=(3, 3), activation='relu',
padding='same'))
mnist_model.add(MaxPooling2D(pool_size=(2, 2))) # Changed pool_size to
2x2
mnist_model.add(Dropout(0.25))

# Second Convolutional Block
mnist_model.add(Conv2D(64, kernel_size=(3, 3), activation='relu',
padding='same'))
mnist_model.add(BatchNormalization())
mnist_model.add(Conv2D(64, kernel_size=(3, 3), activation='relu',
padding='same'))
mnist_model.add(MaxPooling2D(pool_size=(2, 2)))
mnist_model.add(Dropout(0.25))

# Third Convolutional Block
mnist_model.add(Conv2D(128, kernel_size=(3, 3), activation='relu',
padding='same'))
mnist_model.add(BatchNormalization())
mnist_model.add(Conv2D(128, kernel_size=(3, 3), activation='relu',
padding='same'))
mnist_model.add(MaxPooling2D(pool_size=(2, 2)))
mnist_model.add(Dropout(0.25))

# Flatten Layer
mnist_model.add(Flatten())
```

```
# Fully Connected Layers
mnist_model.add(Dense(256, activation='relu'))
mnist_model.add(Dropout(0.25))

# Output Layer: 10 neurons with softmax activation
mnist_model.add(Dense(10, activation='softmax'))

# Compile the Model
mnist_model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

# Print model summary
mnist_model.summary()
```

```
/usr/local/lib/python3.12/dist-packages/keras/src/layers/
convolutional/base_conv.py:113: UserWarning: Do not pass an
`input_shape`/`input_dim` argument to a layer. When using Sequential
models, prefer using an `Input(shape)` object as the first layer in
the model instead.
  super().__init__(activity_regularizer=activity_regularizer,
**kwargs)
```

```
Model: "sequential_2"
```

```
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━
━━━━━┓
┃ Layer (type)                     ┃ Output Shape            ┃
Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━
━━━━━┩
│ conv2d_9 (Conv2D)                │ (None, 28, 28, 32)      │
320 │
├──────────────────────────────────┼─────────────────────────┼──────────
─────┤
│ batch_normalization_3            │ (None, 28, 28, 32)      │
128 │
│ (BatchNormalization)             │                         │
│
├──────────────────────────────────┼─────────────────────────┼──────────
─────┤
│ conv2d_10 (Conv2D)               │ (None, 28, 28, 32)      │
9,248 │
├──────────────────────────────────┼─────────────────────────┼──────────
─────┤
│ max_pooling2d_5 (MaxPooling2D)   │ (None, 14, 14, 32)      │
0 │
├──────────────────────────────────┼─────────────────────────┼──────────
─────┤
```

```
│ dropout_5 (Dropout)               │ (None, 14, 14, 32)        │
0 │
├───────────────────────────────────┼───────────────────────────┼────────
───────┤
│ conv2d_11 (Conv2D)                │ (None, 14, 14, 64)        │
18,496 │
├───────────────────────────────────┼───────────────────────────┼────────
───────┤
│ batch_normalization_4             │ (None, 14, 14, 64)        │
256 │
│ (BatchNormalization)              │                           │
│
├───────────────────────────────────┼───────────────────────────┼────────
───────┤
│ conv2d_12 (Conv2D)                │ (None, 14, 14, 64)        │
36,928 │
├───────────────────────────────────┼───────────────────────────┼────────
───────┤
│ max_pooling2d_6 (MaxPooling2D)    │ (None, 7, 7, 64)          │
0 │
├───────────────────────────────────┼───────────────────────────┼────────
───────┤
│ dropout_6 (Dropout)               │ (None, 7, 7, 64)          │
0 │
├───────────────────────────────────┼───────────────────────────┼────────
───────┤
│ conv2d_13 (Conv2D)                │ (None, 7, 7, 128)         │
73,856 │
├───────────────────────────────────┼───────────────────────────┼────────
───────┤
│ batch_normalization_5             │ (None, 7, 7, 128)         │
512 │
│ (BatchNormalization)              │                           │
│
├───────────────────────────────────┼───────────────────────────┼────────
───────┤
│ conv2d_14 (Conv2D)                │ (None, 7, 7, 128)         │
147,584 │
├───────────────────────────────────┼───────────────────────────┼────────
───────┤
│ max_pooling2d_7 (MaxPooling2D)    │ (None, 3, 3, 128)         │
0 │
├───────────────────────────────────┼───────────────────────────┼────────
───────┤
│ dropout_7 (Dropout)               │ (None, 3, 3, 128)         │
0 │
├───────────────────────────────────┼───────────────────────────┼────────
───────┤
│ flatten_2 (Flatten)               │ (None, 1152)              │
```

```
0 │
├──────────────────────────────────┼─────────────────────────┼──────────
──────┤
│ dense_4 (Dense)                  │ (None, 256)             │
295,168 │
├──────────────────────────────────┼─────────────────────────┼──────────
──────┤
│ dropout_8 (Dropout)              │ (None, 256)             │
0 │
├──────────────────────────────────┼─────────────────────────┼──────────
──────┤
│ dense_5 (Dense)                  │ (None, 10)              │
2,570 │
└──────────────────────────────────┴─────────────────────────┴──────────
──────┘

 Total params: 585,066 (2.23 MB)

 Trainable params: 584,618 (2.23 MB)

 Non-trainable params: 448 (1.75 KB)

# Train
mnist_history = mnist_model.fit(
    mnist_train_images, mnist_train_labels,
    epochs=15,
    batch_size=64,
    validation_data=(mnist_test_images, mnist_test_labels)
)

# Evaluate
test_loss, test_acc = mnist_model.evaluate(mnist_test_images,
mnist_test_labels, verbose=2)
print(f"\nTest Accuracy: {test_acc*100:.2f}%")

Epoch 1/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 20s 13ms/step - accuracy: 0.9961 - loss:
0.0144 - val_accuracy: 0.9955 - val_loss: 0.0253
Epoch 2/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 7s 7ms/step - accuracy: 0.9976 - loss:
0.0084 - val_accuracy: 0.9952 - val_loss: 0.0260
Epoch 3/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 7s 7ms/step - accuracy: 0.9973 - loss:
0.0096 - val_accuracy: 0.9961 - val_loss: 0.0232
Epoch 4/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 7s 7ms/step - accuracy: 0.9971 - loss:
0.0101 - val_accuracy: 0.9946 - val_loss: 0.0241
Epoch 5/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 7s 7ms/step - accuracy: 0.9972 - loss:
0.0087 - val_accuracy: 0.9947 - val_loss: 0.0243
```

```
Epoch 6/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 7s 7ms/step - accuracy: 0.9971 - loss:
0.0090 - val_accuracy: 0.9958 - val_loss: 0.0294
Epoch 7/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 7s 7ms/step - accuracy: 0.9971 - loss:
0.0109 - val_accuracy: 0.9955 - val_loss: 0.0244
Epoch 8/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 7s 7ms/step - accuracy: 0.9975 - loss:
0.0089 - val_accuracy: 0.9953 - val_loss: 0.0218
Epoch 9/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 7s 7ms/step - accuracy: 0.9976 - loss:
0.0086 - val_accuracy: 0.9944 - val_loss: 0.0267
Epoch 10/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 10s 7ms/step - accuracy: 0.9979 - loss:
0.0076 - val_accuracy: 0.9950 - val_loss: 0.0286
Epoch 11/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 7s 7ms/step - accuracy: 0.9976 - loss:
0.0095 - val_accuracy: 0.9946 - val_loss: 0.0285
Epoch 12/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 7s 7ms/step - accuracy: 0.9974 - loss:
0.0093 - val_accuracy: 0.9946 - val_loss: 0.0332
Epoch 13/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 10s 7ms/step - accuracy: 0.9967 - loss:
0.0249 - val_accuracy: 0.9944 - val_loss: 0.0312
Epoch 14/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 7s 7ms/step - accuracy: 0.9982 - loss:
0.0066 - val_accuracy: 0.9945 - val_loss: 0.0318
Epoch 15/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 7s 7ms/step - accuracy: 0.9985 - loss:
0.0053 - val_accuracy: 0.9950 - val_loss: 0.0348
313/313 - 1s - 2ms/step - accuracy: 0.9950 - loss: 0.0348

Test Accuracy: 99.50%
```

VGG-16

CIFAR-10

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from keras.applications.vgg16 import VGG16
from keras.layers import Dense, GlobalAveragePooling2D,
BatchNormalization, Dropout
import numpy as np
from keras.models import Model
from keras.optimizers import Adam

# Use a data augmentation generator to prevent overfitting
datagen = ImageDataGenerator(
```

```
    rotation_range=15,
    width_shift_range=0.1,
    height_shift_range=0.1,
    horizontal_flip=True,
)
datagen.fit(train_images)

# 2. Build the Model from scratch
# Load the VGG16 base with random weights and without the top layers
base_model = VGG16(weights=None, include_top=False,
input_shape=(32,32,3))

# Add a new classification head on top of the VGG16 base
x = base_model.output
x = GlobalAveragePooling2D()(x) # Use GlobalAveragePooling for
simplicity
x = Dense(512, activation='relu')(x)
x = BatchNormalization()(x)
x = Dropout(0.5)(x)
x = Dense(128, activation='relu')(x)
x = BatchNormalization()(x)
x = Dropout(0.5)(x)
predictions = Dense(10, activation='softmax')(x)

model = Model(inputs=base_model.input, outputs=predictions)

# 3. Compile the Model
# Use a suitable learning rate for training from scratch
model.compile(optimizer=Adam(learning_rate=0.001),
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

model.summary()

# 4. Train the Model in a single step
print("\nTraining the model from scratch...")
vgg_history = model.fit(datagen.flow(train_images, train_labels,
batch_size=64),
                    epochs=30,
                    validation_data=(test_images, test_labels))

# 5. Evaluate the Final Model
test_loss, test_acc = model.evaluate(test_images, test_labels,
verbose=2)
print(f"\nFinal Test Accuracy: {test_acc*100:.2f}%")
```

```
Model: "functional_175"

┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━
━━━━━┓
```

```
┃ Layer (type)                      ┃ Output Shape            ┃
Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━
━━━━━┩
│ input_layer_24 (InputLayer)       │ (None, 32, 32, 3)       │
0 │
├───────────────────────────────────┼─────────────────────────┼──────────
─────┤
│ block1_conv1 (Conv2D)             │ (None, 32, 32, 64)      │
1,792 │
├───────────────────────────────────┼─────────────────────────┼──────────
─────┤
│ block1_conv2 (Conv2D)             │ (None, 32, 32, 64)      │
36,928 │
├───────────────────────────────────┼─────────────────────────┼──────────
─────┤
│ block1_pool (MaxPooling2D)        │ (None, 16, 16, 64)      │
0 │
├───────────────────────────────────┼─────────────────────────┼──────────
─────┤
│ block2_conv1 (Conv2D)             │ (None, 16, 16, 128)     │
73,856 │
├───────────────────────────────────┼─────────────────────────┼──────────
─────┤
│ block2_conv2 (Conv2D)             │ (None, 16, 16, 128)     │
147,584 │
├───────────────────────────────────┼─────────────────────────┼──────────
─────┤
│ block2_pool (MaxPooling2D)        │ (None, 8, 8, 128)       │
0 │
├───────────────────────────────────┼─────────────────────────┼──────────
─────┤
│ block3_conv1 (Conv2D)             │ (None, 8, 8, 256)       │
295,168 │
├───────────────────────────────────┼─────────────────────────┼──────────
─────┤
│ block3_conv2 (Conv2D)             │ (None, 8, 8, 256)       │
590,080 │
├───────────────────────────────────┼─────────────────────────┼──────────
─────┤
│ block3_conv3 (Conv2D)             │ (None, 8, 8, 256)       │
590,080 │
├───────────────────────────────────┼─────────────────────────┼──────────
─────┤
│ block3_pool (MaxPooling2D)        │ (None, 4, 4, 256)       │
0 │
├───────────────────────────────────┼─────────────────────────┼──────────
─────┤
│ block4_conv1 (Conv2D)             │ (None, 4, 4, 512)       │
```

```
1,180,160 │
├─────────────────────────────────┼─────────────────────────┼─────────
───────┤
│ block4_conv2 (Conv2D)           │ (None, 4, 4, 512)       │
2,359,808 │
├─────────────────────────────────┼─────────────────────────┼─────────
───────┤
│ block4_conv3 (Conv2D)           │ (None, 4, 4, 512)       │
2,359,808 │
├─────────────────────────────────┼─────────────────────────┼─────────
───────┤
│ block4_pool (MaxPooling2D)      │ (None, 2, 2, 512)       │
0 │
├─────────────────────────────────┼─────────────────────────┼─────────
───────┤
│ block5_conv1 (Conv2D)           │ (None, 2, 2, 512)       │
2,359,808 │
├─────────────────────────────────┼─────────────────────────┼─────────
───────┤
│ block5_conv2 (Conv2D)           │ (None, 2, 2, 512)       │
2,359,808 │
├─────────────────────────────────┼─────────────────────────┼─────────
───────┤
│ block5_conv3 (Conv2D)           │ (None, 2, 2, 512)       │
2,359,808 │
├─────────────────────────────────┼─────────────────────────┼─────────
───────┤
│ block5_pool (MaxPooling2D)      │ (None, 1, 1, 512)       │
0 │
├─────────────────────────────────┼─────────────────────────┼─────────
───────┤
│ global_average_pooling2d_10     │ (None, 512)             │
0 │
│ (GlobalAveragePooling2D)        │                         │
│
├─────────────────────────────────┼─────────────────────────┼─────────
───────┤
│ dense_50 (Dense)                │ (None, 512)             │
262,656 │
├─────────────────────────────────┼─────────────────────────┼─────────
───────┤
│ batch_normalization_14          │ (None, 512)             │
2,048 │
│ (BatchNormalization)            │                         │
│
├─────────────────────────────────┼─────────────────────────┼─────────
───────┤
│ dropout_32 (Dropout)            │ (None, 512)             │
0 │
```

```
├─────────────────────────────────┼─────────────────────────┼──────────
──────┤
│ dense_51 (Dense)                │ (None, 128)             │
65,664 │
├─────────────────────────────────┼─────────────────────────┼──────────
──────┤
│ batch_normalization_15          │ (None, 128)             │
512 │
│ (BatchNormalization)            │                         │
│
├─────────────────────────────────┼─────────────────────────┼──────────
──────┤
│ dropout_33 (Dropout)            │ (None, 128)             │
0 │
├─────────────────────────────────┼─────────────────────────┼──────────
──────┤
│ dense_52 (Dense)                │ (None, 10)              │
1,290 │
└─────────────────────────────────┴─────────────────────────┴──────────
──────┘

 Total params: 15,046,858 (57.40 MB)

 Trainable params: 15,045,578 (57.39 MB)

 Non-trainable params: 1,280 (5.00 KB)


Training the model from scratch...
Epoch 1/30

/usr/local/lib/python3.12/dist-packages/keras/src/trainers/
data_adapters/py_dataset_adapter.py:121: UserWarning: Your `PyDataset`
class should call `super().__init__(**kwargs)` in its constructor.
`**kwargs` can include `workers`, `use_multiprocessing`,
`max_queue_size`. Do not pass these arguments to `fit()`, as they will
be ignored.
  self._warn_if_super_not_called()

782/782 ━━━━━━━━━━━━━━━━━━━━ 72s 74ms/step - accuracy: 0.1532 - loss:
2.5668 - val_accuracy: 0.1362 - val_loss: 2.7775
Epoch 2/30
782/782 ━━━━━━━━━━━━━━━━━━━━ 50s 63ms/step - accuracy: 0.2041 - loss:
1.9419 - val_accuracy: 0.2094 - val_loss: 2.2372
Epoch 3/30
782/782 ━━━━━━━━━━━━━━━━━━━━ 50s 64ms/step - accuracy: 0.2950 - loss:
1.7770 - val_accuracy: 0.1298 - val_loss: 4.7294
Epoch 4/30
782/782 ━━━━━━━━━━━━━━━━━━━━ 50s 63ms/step - accuracy: 0.3714 - loss:
1.5750 - val_accuracy: 0.3644 - val_loss: 1.7125
Epoch 5/30
```

```
782/782 ━━━━━━━━━━━━━━━━━━━━ 50s 64ms/step - accuracy: 0.4242 - loss: 1.4592 - val_accuracy: 0.4662 - val_loss: 1.3612
Epoch 6/30
782/782 ━━━━━━━━━━━━━━━━━━━━ 50s 64ms/step - accuracy: 0.4728 - loss: 1.3770 - val_accuracy: 0.4666 - val_loss: 1.5189
Epoch 7/30
782/782 ━━━━━━━━━━━━━━━━━━━━ 50s 63ms/step - accuracy: 0.5035 - loss: 1.3041 - val_accuracy: 0.4007 - val_loss: 1.6510
Epoch 8/30
782/782 ━━━━━━━━━━━━━━━━━━━━ 50s 63ms/step - accuracy: 0.5334 - loss: 1.2472 - val_accuracy: 0.5571 - val_loss: 1.1954
Epoch 9/30
782/782 ━━━━━━━━━━━━━━━━━━━━ 49s 63ms/step - accuracy: 0.5586 - loss: 1.1876 - val_accuracy: 0.5517 - val_loss: 1.4366
Epoch 10/30
782/782 ━━━━━━━━━━━━━━━━━━━━ 49s 63ms/step - accuracy: 0.5789 - loss: 1.1461 - val_accuracy: 0.4974 - val_loss: 1.6371
Epoch 11/30
782/782 ━━━━━━━━━━━━━━━━━━━━ 50s 64ms/step - accuracy: 0.6021 - loss: 1.0935 - val_accuracy: 0.6306 - val_loss: 1.0340
Epoch 12/30
782/782 ━━━━━━━━━━━━━━━━━━━━ 50s 64ms/step - accuracy: 0.6123 - loss: 1.0690 - val_accuracy: 0.6393 - val_loss: 1.0406
Epoch 13/30
782/782 ━━━━━━━━━━━━━━━━━━━━ 49s 63ms/step - accuracy: 0.6320 - loss: 1.0252 - val_accuracy: 0.6545 - val_loss: 713.3715
Epoch 14/30
782/782 ━━━━━━━━━━━━━━━━━━━━ 49s 63ms/step - accuracy: 0.6626 - loss: 0.9521 - val_accuracy: 0.6980 - val_loss: 0.8770
Epoch 15/30
782/782 ━━━━━━━━━━━━━━━━━━━━ 50s 63ms/step - accuracy: 0.6799 - loss: 0.8904 - val_accuracy: 0.7101 - val_loss: 0.8354
Epoch 16/30
782/782 ━━━━━━━━━━━━━━━━━━━━ 50s 64ms/step - accuracy: 0.6996 - loss: 0.8590 - val_accuracy: 0.7004 - val_loss: 63.4949
Epoch 17/30
782/782 ━━━━━━━━━━━━━━━━━━━━ 49s 63ms/step - accuracy: 0.7202 - loss: 0.8168 - val_accuracy: 0.7228 - val_loss: 0.8135
Epoch 18/30
782/782 ━━━━━━━━━━━━━━━━━━━━ 50s 64ms/step - accuracy: 0.7381 - loss: 0.7808 - val_accuracy: 0.7464 - val_loss: 0.7694
Epoch 19/30
782/782 ━━━━━━━━━━━━━━━━━━━━ 50s 63ms/step - accuracy: 0.7445 - loss: 0.7569 - val_accuracy: 0.6701 - val_loss: 504.3535
Epoch 20/30
782/782 ━━━━━━━━━━━━━━━━━━━━ 49s 63ms/step - accuracy: 0.7625 - loss: 0.7270 - val_accuracy: 0.7309 - val_loss: 0.8723
Epoch 21/30
782/782 ━━━━━━━━━━━━━━━━━━━━ 49s 63ms/step - accuracy: 0.7697 - loss:
```

```
0.6967 - val_accuracy: 0.7700 - val_loss: 0.7232
Epoch 22/30
782/782 ━━━━━━━━━━━━━━━━━━━━ 49s 63ms/step - accuracy: 0.7768 - loss:
0.6870 - val_accuracy: 0.7516 - val_loss: 0.8347
Epoch 23/30
782/782 ━━━━━━━━━━━━━━━━━━━━ 50s 64ms/step - accuracy: 0.7882 - loss:
0.6605 - val_accuracy: 0.7854 - val_loss: 0.6630
Epoch 24/30
782/782 ━━━━━━━━━━━━━━━━━━━━ 50s 64ms/step - accuracy: 0.7978 - loss:
0.6296 - val_accuracy: 0.7511 - val_loss: 0.8311
Epoch 25/30
782/782 ━━━━━━━━━━━━━━━━━━━━ 50s 64ms/step - accuracy: 0.8026 - loss:
0.6052 - val_accuracy: 0.7798 - val_loss: 28436.8672
Epoch 26/30
782/782 ━━━━━━━━━━━━━━━━━━━━ 50s 63ms/step - accuracy: 0.8099 - loss:
0.5842 - val_accuracy: 0.7713 - val_loss: 0.7134
Epoch 27/30
782/782 ━━━━━━━━━━━━━━━━━━━━ 50s 63ms/step - accuracy: 0.8149 - loss:
0.5694 - val_accuracy: 0.8017 - val_loss: 0.6376
Epoch 28/30
782/782 ━━━━━━━━━━━━━━━━━━━━ 82s 63ms/step - accuracy: 0.8212 - loss:
0.5480 - val_accuracy: 0.7936 - val_loss: 576.2103
Epoch 29/30
782/782 ━━━━━━━━━━━━━━━━━━━━ 50s 64ms/step - accuracy: 0.8295 - loss:
0.5307 - val_accuracy: 0.8303 - val_loss: 0.5371
Epoch 30/30
782/782 ━━━━━━━━━━━━━━━━━━━━ 50s 64ms/step - accuracy: 0.8342 - loss:
0.5206 - val_accuracy: 0.8226 - val_loss: 0.5700
313/313 - 2s - 7ms/step - accuracy: 0.8226 - loss: 0.5700

Final Test Accuracy: 82.26%
```

MNIST

```
# 1 Load MNIST dataset
(mnist_train_images, mnist_train_labels), (mnist_test_images,
mnist_test_labels) = datasets.mnist.load_data()

# 2 Preprocess: Resize to (32,32) and convert to 3 channels
mnist_train_images = np.stack((mnist_train_images,)*3, axis=-1)  #
make 3 channels
mnist_test_images = np.stack((mnist_test_images,)*3, axis=-1)

mnist_train_images = tf.image.resize(mnist_train_images, (32,
32)).numpy()
mnist_test_images = tf.image.resize(mnist_test_images, (32,
32)).numpy()

# Normalize pixel values
mnist_train_images = mnist_train_images.astype('float32') / 255.0
```

```
mnist_test_images = mnist_test_images.astype('float32') / 255.0

print("MNIST dataset prepared for VGG16:")
print(f"Train images shape: {mnist_train_images.shape}")
print(f"Test images shape: {mnist_test_images.shape}")
```

```
MNIST dataset prepared for VGG16:
Train images shape: (60000, 32, 32, 3)
Test images shape: (10000, 32, 32, 3)
```

```
import tensorflow as tf
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.applications.vgg16 import VGG16
from tensorflow.keras.layers import Dense, GlobalAveragePooling2D,
BatchNormalization, Dropout
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam
from tensorflow.keras import datasets
import numpy as np


#  Data Augmentation
datagen = ImageDataGenerator(
    rotation_range=15,
    width_shift_range=0.1,
    height_shift_range=0.1,
    horizontal_flip=False,
)
datagen.fit(mnist_train_images)

#  Build VGG16 base model
base_model = VGG16(weights=None, include_top=False, input_shape=(32,
32, 3))

#  Add classification layers
x = base_model.output
x = GlobalAveragePooling2D()(x)
x = Dense(512, activation='relu')(x)
x = BatchNormalization()(x)
x = Dropout(0.5)(x)
x = Dense(128, activation='relu')(x)
x = BatchNormalization()(x)
x = Dropout(0.5)(x)
predictions = Dense(10, activation='softmax')(x)

model = Model(inputs=base_model.input, outputs=predictions)

# 6 Compile
model.compile(
    optimizer=Adam(learning_rate=0.001),
```

```
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)

model.summary()
```

```
Model: "functional_48"

┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━
━━━━━┓
┃ Layer (type)                     ┃ Output Shape           ┃
Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━
━━━━━┩
│ input_layer_4 (InputLayer)       │ (None, 32, 32, 3)      │
0 │
├──────────────────────────────────┼────────────────────────┼──────────
─────┤
│ block1_conv1 (Conv2D)            │ (None, 32, 32, 64)     │
1,792 │
├──────────────────────────────────┼────────────────────────┼──────────
─────┤
│ block1_conv2 (Conv2D)            │ (None, 32, 32, 64)     │
36,928 │
├──────────────────────────────────┼────────────────────────┼──────────
─────┤
│ block1_pool (MaxPooling2D)       │ (None, 16, 16, 64)     │
0 │
├──────────────────────────────────┼────────────────────────┼──────────
─────┤
│ block2_conv1 (Conv2D)            │ (None, 16, 16, 128)    │
73,856 │
├──────────────────────────────────┼────────────────────────┼──────────
─────┤
│ block2_conv2 (Conv2D)            │ (None, 16, 16, 128)    │
147,584 │
├──────────────────────────────────┼────────────────────────┼──────────
─────┤
│ block2_pool (MaxPooling2D)       │ (None, 8, 8, 128)      │
0 │
├──────────────────────────────────┼────────────────────────┼──────────
─────┤
│ block3_conv1 (Conv2D)            │ (None, 8, 8, 256)      │
295,168 │
├──────────────────────────────────┼────────────────────────┼──────────
─────┤
│ block3_conv2 (Conv2D)            │ (None, 8, 8, 256)      │
590,080 │
```

```
├──────────────────────────────────┼─────────────────────────┼──────────
───────┤
│ block3_conv3 (Conv2D)            │ (None, 8, 8, 256)       │
590,080 │
├──────────────────────────────────┼─────────────────────────┼──────────
───────┤
│ block3_pool (MaxPooling2D)       │ (None, 4, 4, 256)       │
0 │
├──────────────────────────────────┼─────────────────────────┼──────────
───────┤
│ block4_conv1 (Conv2D)            │ (None, 4, 4, 512)       │
1,180,160 │
├──────────────────────────────────┼─────────────────────────┼──────────
───────┤
│ block4_conv2 (Conv2D)            │ (None, 4, 4, 512)       │
2,359,808 │
├──────────────────────────────────┼─────────────────────────┼──────────
───────┤
│ block4_conv3 (Conv2D)            │ (None, 4, 4, 512)       │
2,359,808 │
├──────────────────────────────────┼─────────────────────────┼──────────
───────┤
│ block4_pool (MaxPooling2D)       │ (None, 2, 2, 512)       │
0 │
├──────────────────────────────────┼─────────────────────────┼──────────
───────┤
│ block5_conv1 (Conv2D)            │ (None, 2, 2, 512)       │
2,359,808 │
├──────────────────────────────────┼─────────────────────────┼──────────
───────┤
│ block5_conv2 (Conv2D)            │ (None, 2, 2, 512)       │
2,359,808 │
├──────────────────────────────────┼─────────────────────────┼──────────
───────┤
│ block5_conv3 (Conv2D)            │ (None, 2, 2, 512)       │
2,359,808 │
├──────────────────────────────────┼─────────────────────────┼──────────
───────┤
│ block5_pool (MaxPooling2D)       │ (None, 1, 1, 512)       │
0 │
├──────────────────────────────────┼─────────────────────────┼──────────
───────┤
│ global_average_pooling2d_1       │ (None, 512)             │
0 │
│ (GlobalAveragePooling2D)         │                         │
│
├──────────────────────────────────┼─────────────────────────┼──────────
───────┤
│ dense_9 (Dense)                  │ (None, 512)             │
```

```
262,656 │
├──────────────────────────────────┼─────────────────────────┼──────────
───────┤
│ batch_normalization_8            │ (None, 512)             │
2,048 │
│ (BatchNormalization)             │                         │
│
├──────────────────────────────────┼─────────────────────────┼──────────
───────┤
│ dropout_11 (Dropout)             │ (None, 512)             │
0 │
├──────────────────────────────────┼─────────────────────────┼──────────
───────┤
│ dense_10 (Dense)                 │ (None, 128)             │
65,664 │
├──────────────────────────────────┼─────────────────────────┼──────────
───────┤
│ batch_normalization_9            │ (None, 128)             │
512 │
│ (BatchNormalization)             │                         │
│
├──────────────────────────────────┼─────────────────────────┼──────────
───────┤
│ dropout_12 (Dropout)             │ (None, 128)             │
0 │
├──────────────────────────────────┼─────────────────────────┼──────────
───────┤
│ dense_11 (Dense)                 │ (None, 10)              │
1,290 │
└──────────────────────────────────┴─────────────────────────┴──────────
───────┘

 Total params: 15,046,858 (57.40 MB)

 Trainable params: 15,045,578 (57.39 MB)

 Non-trainable params: 1,280 (5.00 KB)

#  Train
print("\nTraining the VGG16 model on MNIST...")
history = model.fit(
    datagen.flow(mnist_train_images, mnist_train_labels,
batch_size=64),
    epochs=15,
    validation_data=(mnist_test_images, mnist_test_labels)
)

#  Evaluate
test_loss, test_acc = model.evaluate(mnist_test_images,
```

```
mnist_test_labels, verbose=2)
print(f"\nFinal Test Accuracy: {test_acc*100:.2f}%")
```

```
Training the VGG16 model on MNIST...
Epoch 1/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 81s 76ms/step - accuracy: 0.4713 - loss:
1.5785 - val_accuracy: 0.9183 - val_loss: 0.2644
Epoch 2/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 62s 66ms/step - accuracy: 0.9360 - loss:
0.2396 - val_accuracy: 0.9726 - val_loss: 0.0955
Epoch 3/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 61s 65ms/step - accuracy: 0.9686 - loss:
0.1213 - val_accuracy: 0.8617 - val_loss: 1.7803
Epoch 4/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 59s 63ms/step - accuracy: 0.9773 - loss:
0.0938 - val_accuracy: 0.9793 - val_loss: 0.0751
Epoch 5/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 59s 63ms/step - accuracy: 0.9823 - loss:
0.0745 - val_accuracy: 0.9893 - val_loss: 0.0389
Epoch 6/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 59s 63ms/step - accuracy: 0.9847 - loss:
0.0610 - val_accuracy: 0.9921 - val_loss: 0.0333
Epoch 7/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 60s 64ms/step - accuracy: 0.9865 - loss:
0.0572 - val_accuracy: 0.9729 - val_loss: 0.1247
Epoch 8/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 59s 63ms/step - accuracy: 0.9877 - loss:
0.0514 - val_accuracy: 0.9906 - val_loss: 0.4313
Epoch 9/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 59s 63ms/step - accuracy: 0.9878 - loss:
0.0514 - val_accuracy: 0.9887 - val_loss: 1677.7125
Epoch 10/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 82s 64ms/step - accuracy: 0.9894 - loss:
0.0448 - val_accuracy: 0.9928 - val_loss: 0.0290
Epoch 11/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 59s 63ms/step - accuracy: 0.9911 - loss:
0.0357 - val_accuracy: 0.9920 - val_loss: 0.0307
Epoch 12/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 59s 63ms/step - accuracy: 0.9919 - loss:
0.0353 - val_accuracy: 0.9921 - val_loss: 0.0293
Epoch 13/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 59s 63ms/step - accuracy: 0.9916 - loss:
0.0351 - val_accuracy: 0.9937 - val_loss: 3.2172
Epoch 14/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 59s 63ms/step - accuracy: 0.9924 - loss:
0.0323 - val_accuracy: 0.9944 - val_loss: 0.0218
Epoch 15/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 59s 63ms/step - accuracy: 0.9933 - loss:
0.0299 - val_accuracy: 0.9900 - val_loss: 0.0443
```

```
313/313 - 2s - 7ms/step - accuracy: 0.9900 - loss: 0.0443

Final Test Accuracy: 99.00%
```

RNN

CIFAR-10

```
import tensorflow as tf
from tensorflow import keras
from keras import datasets
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.layers import Input, LSTM, Dense, Dropout,
BatchNormalization
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam


# 2. Data Augmentation
datagen = ImageDataGenerator(
    rotation_range=15,
    width_shift_range=0.1,
    height_shift_range=0.1,
    horizontal_flip=True
)
datagen.fit(train_images)

# 3. Reshape for RNN input (32 timesteps, 96 features)
train_images_rnn = train_images.reshape(-1, 32, 96)
test_images_rnn = test_images.reshape(-1, 32, 96)

# Custom data generator for RNN
def rnn_data_generator(generator):
    while True:
        x_batch, y_batch = next(generator)
        x_batch = x_batch.reshape(-1, 32, 96)
        yield x_batch, y_batch

train_gen = rnn_data_generator(datagen.flow(train_images,
train_labels, batch_size=64))

# 4. Build the RNN model (Functional API)
inputs = Input(shape=(32, 96))

x = LSTM(256, return_sequences=True)(inputs)
x = BatchNormalization()(x)
x = Dropout(0.3)(x)
```

```
x = LSTM(128)(x)
x = BatchNormalization()(x)
x = Dropout(0.3)(x)

x = Dense(128, activation='relu')(x)
x = BatchNormalization()(x)
x = Dropout(0.5)(x)

outputs = Dense(10, activation='softmax')(x)

model = Model(inputs=inputs, outputs=outputs,
name="RNN_CIFAR10_Model")

# 5. Compile the model
model.compile(
    optimizer=Adam(learning_rate=0.001),
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)

model.summary()

# 6. Train the model
print("\nTraining the RNN model from scratch...")
rnn_history = model.fit(
    train_gen,
    steps_per_epoch=len(train_images) // 64,
    epochs=30,
    validation_data=(test_images_rnn, test_labels)
)

# 7. Evaluate the model
test_loss, test_acc = model.evaluate(test_images_rnn, test_labels,
verbose=2)
print(f"\nFinal Test Accuracy: {test_acc * 100:.2f}%")
```

```
Model: "RNN_CIFAR10_Model"

┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━
━━━━━┓
┃ Layer (type)                     ┃ Output Shape           ┃
Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━
━━━━━┩
│ input_layer (InputLayer)         │ (None, 32, 96)         │
0 │
├──────────────────────────────────┼────────────────────────┼──────────
─────┤
│ lstm (LSTM)                      │ (None, 32, 256)        │
361,472 │
```

```
├───────────────────────────────────┼──────────────────────────┼────────
───────┤
│ batch_normalization               │ (None, 32, 256)          │
1,024 │
│ (BatchNormalization)              │                          │
│
├───────────────────────────────────┼──────────────────────────┼────────
───────┤
│ dropout (Dropout)                 │ (None, 32, 256)          │
0 │
├───────────────────────────────────┼──────────────────────────┼────────
───────┤
│ lstm_1 (LSTM)                     │ (None, 128)              │
197,120 │
├───────────────────────────────────┼──────────────────────────┼────────
───────┤
│ batch_normalization_1             │ (None, 128)              │
512 │
│ (BatchNormalization)              │                          │
│
├───────────────────────────────────┼──────────────────────────┼────────
───────┤
│ dropout_1 (Dropout)               │ (None, 128)              │
0 │
├───────────────────────────────────┼──────────────────────────┼────────
───────┤
│ dense (Dense)                     │ (None, 128)              │
16,512 │
├───────────────────────────────────┼──────────────────────────┼────────
───────┤
│ batch_normalization_2             │ (None, 128)              │
512 │
│ (BatchNormalization)              │                          │
│
├───────────────────────────────────┼──────────────────────────┼────────
───────┤
│ dropout_2 (Dropout)               │ (None, 128)              │
0 │
├───────────────────────────────────┼──────────────────────────┼────────
───────┤
│ dense_1 (Dense)                   │ (None, 10)               │
1,290 │
└───────────────────────────────────┴──────────────────────────┴────────
───────┘

 Total params: 578,442 (2.21 MB)

 Trainable params: 577,418 (2.20 MB)

 Non-trainable params: 1,024 (4.00 KB)
```

```
Training the RNN model from scratch...
Epoch 1/30
781/781 ━━━━━━━━━━━━━━━━━━━━ 42s 46ms/step - accuracy: 0.2091 - loss: 
2.5342 - val_accuracy: 0.3180 - val_loss: 1.8348
Epoch 2/30
781/781 ━━━━━━━━━━━━━━━━━━━━ 35s 44ms/step - accuracy: 0.3337 - loss: 
1.8289 - val_accuracy: 0.3917 - val_loss: 1.6542
Epoch 3/30
781/781 ━━━━━━━━━━━━━━━━━━━━ 35s 45ms/step - accuracy: 0.3844 - loss: 
1.6832 - val_accuracy: 0.3724 - val_loss: 1.7804
Epoch 4/30
781/781 ━━━━━━━━━━━━━━━━━━━━ 36s 46ms/step - accuracy: 0.4205 - loss: 
1.5976 - val_accuracy: 0.4066 - val_loss: 1.5832
Epoch 5/30
781/781 ━━━━━━━━━━━━━━━━━━━━ 34s 44ms/step - accuracy: 0.4491 - loss: 
1.5251 - val_accuracy: 0.4953 - val_loss: 1.4071
Epoch 6/30
781/781 ━━━━━━━━━━━━━━━━━━━━ 35s 45ms/step - accuracy: 0.4758 - loss: 
1.4633 - val_accuracy: 0.5072 - val_loss: 1.3508
Epoch 7/30
781/781 ━━━━━━━━━━━━━━━━━━━━ 34s 44ms/step - accuracy: 0.4960 - loss: 
1.4182 - val_accuracy: 0.4725 - val_loss: 1.4310
Epoch 8/30
781/781 ━━━━━━━━━━━━━━━━━━━━ 34s 44ms/step - accuracy: 0.5133 - loss: 
1.3632 - val_accuracy: 0.5449 - val_loss: 1.2670
Epoch 9/30
781/781 ━━━━━━━━━━━━━━━━━━━━ 36s 46ms/step - accuracy: 0.5231 - loss: 
1.3376 - val_accuracy: 0.5644 - val_loss: 1.2120
Epoch 10/30
781/781 ━━━━━━━━━━━━━━━━━━━━ 33s 42ms/step - accuracy: 0.5304 - loss: 
1.3199 - val_accuracy: 0.5491 - val_loss: 1.2625
Epoch 11/30
781/781 ━━━━━━━━━━━━━━━━━━━━ 35s 44ms/step - accuracy: 0.5458 - loss: 
1.2839 - val_accuracy: 0.5594 - val_loss: 1.2151
Epoch 12/30
781/781 ━━━━━━━━━━━━━━━━━━━━ 36s 46ms/step - accuracy: 0.5562 - loss: 
1.2539 - val_accuracy: 0.5859 - val_loss: 1.1478
Epoch 13/30
781/781 ━━━━━━━━━━━━━━━━━━━━ 36s 46ms/step - accuracy: 0.5673 - loss: 
1.2314 - val_accuracy: 0.5503 - val_loss: 1.2672
Epoch 14/30
781/781 ━━━━━━━━━━━━━━━━━━━━ 35s 45ms/step - accuracy: 0.5700 - loss: 
1.2161 - val_accuracy: 0.6170 - val_loss: 1.0745
Epoch 15/30
781/781 ━━━━━━━━━━━━━━━━━━━━ 35s 44ms/step - accuracy: 0.5797 - loss: 
1.1939 - val_accuracy: 0.6154 - val_loss: 1.0713
Epoch 16/30
781/781 ━━━━━━━━━━━━━━━━━━━━ 34s 43ms/step - accuracy: 0.5907 - loss: 
1.1715 - val_accuracy: 0.6157 - val_loss: 1.0917
```

```
Epoch 17/30
781/781 ━━━━━━━━━━━━━━━━━━━━ 35s 45ms/step - accuracy: 0.5903 - loss:
1.1648 - val_accuracy: 0.5923 - val_loss: 1.1362
Epoch 18/30
781/781 ━━━━━━━━━━━━━━━━━━━━ 35s 45ms/step - accuracy: 0.5980 - loss:
1.1450 - val_accuracy: 0.6103 - val_loss: 1.0930
Epoch 19/30
781/781 ━━━━━━━━━━━━━━━━━━━━ 34s 44ms/step - accuracy: 0.6043 - loss:
1.1306 - val_accuracy: 0.6148 - val_loss: 1.0817
Epoch 20/30
781/781 ━━━━━━━━━━━━━━━━━━━━ 35s 45ms/step - accuracy: 0.6129 - loss:
1.1116 - val_accuracy: 0.6328 - val_loss: 1.0319
Epoch 21/30
781/781 ━━━━━━━━━━━━━━━━━━━━ 34s 44ms/step - accuracy: 0.6190 - loss:
1.0982 - val_accuracy: 0.6323 - val_loss: 1.0395
Epoch 22/30
781/781 ━━━━━━━━━━━━━━━━━━━━ 34s 43ms/step - accuracy: 0.6217 - loss:
1.0875 - val_accuracy: 0.6448 - val_loss: 1.0000
Epoch 23/30
781/781 ━━━━━━━━━━━━━━━━━━━━ 34s 44ms/step - accuracy: 0.6273 - loss:
1.0744 - val_accuracy: 0.6352 - val_loss: 1.0221
Epoch 24/30
781/781 ━━━━━━━━━━━━━━━━━━━━ 35s 45ms/step - accuracy: 0.6305 - loss:
1.0625 - val_accuracy: 0.6297 - val_loss: 1.0564
Epoch 25/30
781/781 ━━━━━━━━━━━━━━━━━━━━ 34s 44ms/step - accuracy: 0.6341 - loss:
1.0548 - val_accuracy: 0.6417 - val_loss: 1.0134
Epoch 26/30
781/781 ━━━━━━━━━━━━━━━━━━━━ 33s 43ms/step - accuracy: 0.6398 - loss:
1.0399 - val_accuracy: 0.6565 - val_loss: 0.9814
Epoch 27/30
781/781 ━━━━━━━━━━━━━━━━━━━━ 34s 44ms/step - accuracy: 0.6472 - loss:
1.0269 - val_accuracy: 0.6449 - val_loss: 1.0058
Epoch 28/30
781/781 ━━━━━━━━━━━━━━━━━━━━ 34s 44ms/step - accuracy: 0.6497 - loss:
1.0103 - val_accuracy: 0.6461 - val_loss: 1.0115
Epoch 29/30
781/781 ━━━━━━━━━━━━━━━━━━━━ 36s 47ms/step - accuracy: 0.6542 - loss:
1.0088 - val_accuracy: 0.6710 - val_loss: 0.9522
Epoch 30/30
781/781 ━━━━━━━━━━━━━━━━━━━━ 34s 44ms/step - accuracy: 0.6543 - loss:
1.0000 - val_accuracy: 0.6708 - val_loss: 0.9501
313/313 - 1s - 4ms/step - accuracy: 0.6708 - loss: 0.9501

Final Test Accuracy: 67.08%
```

MNIST

```
import tensorflow as tf
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.layers import Input, LSTM, Dense,
BatchNormalization, Dropout
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam
from tensorflow.keras import datasets
import numpy as np

# 1 Load and normalize MNIST data
(mnist_train_images, mnist_train_labels), (mnist_test_images,
mnist_test_labels) = datasets.mnist.load_data()

# MNIST images are grayscale (28x28) → expand to (28, 28, 1)
mnist_train_images = np.expand_dims(mnist_train_images, -1)
mnist_test_images = np.expand_dims(mnist_test_images, -1)

# Normalize to [0,1]
mnist_train_images, mnist_test_images = mnist_train_images / 255.0,
mnist_test_images / 255.0

print("MNIST dataset loaded successfully.")
print(f"Train images shape: {mnist_train_images.shape}")
print(f"Train labels shape: {mnist_train_labels.shape}")
print(f"Test images shape: {mnist_test_images.shape}")
print(f"Test labels shape: {mnist_test_labels.shape}")

# 2 Data Augmentation
datagen = ImageDataGenerator(
    rotation_range=10,
    width_shift_range=0.1,
    height_shift_range=0.1,
)
datagen.fit(mnist_train_images)

# 3 Reshape MNIST for RNN input
# Treat each row (28 pixels) as a time step, and 28 features per step
train_images_rnn = mnist_train_images.reshape(-1, 28, 28)
test_images_rnn = mnist_test_images.reshape(-1, 28, 28)

# Custom generator to reshape augmented images for RNN
def rnn_data_generator(generator):
    while True:
        x_batch, y_batch = next(generator)
        x_batch = x_batch.reshape(-1, 28, 28)
        yield x_batch, y_batch

train_gen = rnn_data_generator(datagen.flow(mnist_train_images,
mnist_train_labels, batch_size=64))
```

```python
# 4 Build the RNN model using Functional API
inputs = Input(shape=(28, 28))

x = LSTM(128, return_sequences=True)(inputs)
x = BatchNormalization()(x)
x = Dropout(0.3)(x)

x = LSTM(64)(x)
x = BatchNormalization()(x)
x = Dropout(0.3)(x)

x = Dense(128, activation='relu')(x)
x = BatchNormalization()(x)
x = Dropout(0.5)(x)

predictions = Dense(10, activation='softmax')(x)

model = Model(inputs=inputs, outputs=predictions,
name="RNN_MNIST_Model")

# 5 Compile
model.compile(
    optimizer=Adam(learning_rate=0.001),
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)

model.summary()

# 6 Train
print("\nTraining RNN model on MNIST dataset...")
history = model.fit(
    train_gen,
    steps_per_epoch=len(mnist_train_images) // 64,
    epochs=20,
    validation_data=(test_images_rnn, mnist_test_labels)
)

# 7 Evaluate
test_loss, test_acc = model.evaluate(test_images_rnn,
mnist_test_labels, verbose=2)
print(f"\nFinal Test Accuracy: {test_acc * 100:.2f}%")
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-
keras-datasets/mnist.npz
11490434/11490434 ━━━━━━━━━━━━━━━━━━━━ 2s 0us/step
MNIST dataset loaded successfully.
Train images shape: (60000, 28, 28, 1)
Train labels shape: (60000,)
```

```
Test images shape: (10000, 28, 28, 1)
Test labels shape: (10000,)

Model: "RNN_MNIST_Model"

┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━
━━━━━┓
┃ Layer (type)                    ┃ Output Shape           ┃
Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━
━━━━━┩
│ input_layer_1 (InputLayer)      │ (None, 28, 28)         │
0 │
├─────────────────────────────────┼────────────────────────┼──────────
─────┤
│ lstm_2 (LSTM)                   │ (None, 28, 128)        │
80,384 │
├─────────────────────────────────┼────────────────────────┼──────────
─────┤
│ batch_normalization_3           │ (None, 28, 128)        │
512 │
│ (BatchNormalization)            │                        │
│
├─────────────────────────────────┼────────────────────────┼──────────
─────┤
│ dropout_3 (Dropout)             │ (None, 28, 128)        │
0 │
├─────────────────────────────────┼────────────────────────┼──────────
─────┤
│ lstm_3 (LSTM)                   │ (None, 64)             │
49,408 │
├─────────────────────────────────┼────────────────────────┼──────────
─────┤
│ batch_normalization_4           │ (None, 64)             │
256 │
│ (BatchNormalization)            │                        │
│
├─────────────────────────────────┼────────────────────────┼──────────
─────┤
│ dropout_4 (Dropout)             │ (None, 64)             │
0 │
├─────────────────────────────────┼────────────────────────┼──────────
─────┤
│ dense_2 (Dense)                 │ (None, 128)            │
8,320 │
├─────────────────────────────────┼────────────────────────┼──────────
─────┤
│ batch_normalization_5           │ (None, 128)            │
512 │
│ (BatchNormalization)            │                        │
```

```
│
├──────────────────────────────────┼─────────────────────────┼───────
───────┤
│ dropout_5 (Dropout)              │ (None, 128)             │
0 │
├──────────────────────────────────┼─────────────────────────┼───────
───────┤
│ dense_3 (Dense)                  │ (None, 10)              │
1,290 │
└──────────────────────────────────┴─────────────────────────┴───────
───────┘

 Total params: 140,682 (549.54 KB)

 Trainable params: 140,042 (547.04 KB)

 Non-trainable params: 640 (2.50 KB)


Training RNN model on MNIST dataset...
Epoch 1/20
937/937 ━━━━━━━━━━━━━━━━━━━━ 28s 27ms/step - accuracy: 0.5374 - loss:
1.4747 - val_accuracy: 0.8795 - val_loss: 0.3830
Epoch 2/20
937/937 ━━━━━━━━━━━━━━━━━━━━ 25s 27ms/step - accuracy: 0.9216 - loss:
0.2766 - val_accuracy: 0.9718 - val_loss: 0.0971
Epoch 3/20
937/937 ━━━━━━━━━━━━━━━━━━━━ 25s 27ms/step - accuracy: 0.9494 - loss:
0.1799 - val_accuracy: 0.9792 - val_loss: 0.0759
Epoch 4/20
937/937 ━━━━━━━━━━━━━━━━━━━━ 26s 27ms/step - accuracy: 0.9628 - loss:
0.1350 - val_accuracy: 0.9839 - val_loss: 0.0515
Epoch 5/20
937/937 ━━━━━━━━━━━━━━━━━━━━ 26s 28ms/step - accuracy: 0.9686 - loss:
0.1146 - val_accuracy: 0.9827 - val_loss: 0.0648
Epoch 6/20
937/937 ━━━━━━━━━━━━━━━━━━━━ 26s 28ms/step - accuracy: 0.9747 - loss:
0.0942 - val_accuracy: 0.9858 - val_loss: 0.0477
Epoch 7/20
937/937 ━━━━━━━━━━━━━━━━━━━━ 26s 27ms/step - accuracy: 0.9742 - loss:
0.0937 - val_accuracy: 0.9903 - val_loss: 0.0363
Epoch 8/20
937/937 ━━━━━━━━━━━━━━━━━━━━ 25s 27ms/step - accuracy: 0.9790 - loss:
0.0793 - val_accuracy: 0.9891 - val_loss: 0.0368
Epoch 9/20
937/937 ━━━━━━━━━━━━━━━━━━━━ 26s 28ms/step - accuracy: 0.9784 - loss:
0.0809 - val_accuracy: 0.9862 - val_loss: 0.0483
Epoch 10/20
937/937 ━━━━━━━━━━━━━━━━━━━━ 27s 28ms/step - accuracy: 0.9810 - loss:
0.0726 - val_accuracy: 0.9872 - val_loss: 0.0442
```

```
Epoch 11/20
937/937 ━━━━━━━━━━━━━━━━━━━━ 26s 27ms/step - accuracy: 0.9821 - loss:
0.0682 - val_accuracy: 0.9836 - val_loss: 0.0659
Epoch 12/20
937/937 ━━━━━━━━━━━━━━━━━━━━ 25s 27ms/step - accuracy: 0.9839 - loss:
0.0610 - val_accuracy: 0.9876 - val_loss: 0.0489
Epoch 13/20
937/937 ━━━━━━━━━━━━━━━━━━━━ 25s 27ms/step - accuracy: 0.9838 - loss:
0.0622 - val_accuracy: 0.9911 - val_loss: 0.0316
Epoch 14/20
937/937 ━━━━━━━━━━━━━━━━━━━━ 26s 27ms/step - accuracy: 0.9868 - loss:
0.0523 - val_accuracy: 0.9902 - val_loss: 0.0357
Epoch 15/20
937/937 ━━━━━━━━━━━━━━━━━━━━ 25s 27ms/step - accuracy: 0.9853 - loss:
0.0533 - val_accuracy: 0.9932 - val_loss: 0.0232
Epoch 16/20
937/937 ━━━━━━━━━━━━━━━━━━━━ 26s 27ms/step - accuracy: 0.9852 - loss:
0.0561 - val_accuracy: 0.9924 - val_loss: 0.0256
Epoch 17/20
937/937 ━━━━━━━━━━━━━━━━━━━━ 26s 28ms/step - accuracy: 0.9863 - loss:
0.0505 - val_accuracy: 0.9920 - val_loss: 0.0298
Epoch 18/20
937/937 ━━━━━━━━━━━━━━━━━━━━ 26s 28ms/step - accuracy: 0.9881 - loss:
0.0471 - val_accuracy: 0.9899 - val_loss: 0.0374
Epoch 19/20
937/937 ━━━━━━━━━━━━━━━━━━━━ 26s 28ms/step - accuracy: 0.9878 - loss:
0.0467 - val_accuracy: 0.9907 - val_loss: 0.0358
Epoch 20/20
937/937 ━━━━━━━━━━━━━━━━━━━━ 26s 27ms/step - accuracy: 0.9888 - loss:
0.0431 - val_accuracy: 0.9909 - val_loss: 0.0342
313/313 - 1s - 4ms/step - accuracy: 0.9909 - loss: 0.0342

Final Test Accuracy: 99.09%
```

AlexNet

CIFAR-10

```
import tensorflow as tf
from tensorflow import keras
from keras import datasets, layers, models, optimizers
import numpy as np

# -------------------- Load and preprocess CIFAR-10
--------------------
(train_images, train_labels), (test_images, test_labels) =
datasets.cifar10.load_data()
```

```
train_images, test_images = train_images / 255.0, test_images / 255.0

print("CIFAR-10 dataset loaded successfully.")
print(f"Train images shape: {train_images.shape}")
print(f"Test images shape: {test_images.shape}")

# -------------------- Define AlexNet for CIFAR-10
--------------------
def build_alexnet(input_shape=(32, 32, 3), num_classes=10):
    model = models.Sequential([
        layers.Conv2D(64, (3, 3), activation='relu',
input_shape=input_shape, padding='same'),
        layers.BatchNormalization(),
        layers.MaxPooling2D((2, 2)),

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

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

        layers.Flatten(),
        layers.Dense(512, activation='relu'),
        layers.Dropout(0.5),
        layers.Dense(256, activation='relu'),
        layers.Dropout(0.5),
        layers.Dense(num_classes, activation='softmax')
    ])
    return model

model = build_alexnet()
model.summary()

# -------------------- Compile --------------------
model.compile(optimizer=optimizers.Adam(learning_rate=1e-3),
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

# -------------------- Callbacks --------------------
callbacks = [
    keras.callbacks.ReduceLROnPlateau(monitor='val_loss', factor=0.5,
patience=3, verbose=1),
    keras.callbacks.EarlyStopping(monitor='val_loss', patience=7,
restore_best_weights=True, verbose=1)
]

# -------------------- Train --------------------
history = model.fit(train_images, train_labels,
```

```
                    epochs=30,
                    batch_size=64,
                    validation_data=(test_images, test_labels),
                    callbacks=callbacks)

# -------------------- Evaluate --------------------
test_loss, test_acc = model.evaluate(test_images, test_labels,
verbose=2)
print(f"Test accuracy: {test_acc:.4f}")

# -------------------- Save --------------------
model.save('alexnet_cifar10.h5')
print("Model saved as alexnet_cifar10.h5")
```

```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-
python.tar.gz
170498071/170498071 ━━━━━━━━━━━━━━━━━━━━ 12s 0us/step
CIFAR-10 dataset loaded successfully.
Train images shape: (50000, 32, 32, 3)
Test images shape: (10000, 32, 32, 3)
```

```
/usr/local/lib/python3.12/dist-packages/keras/src/layers/
convolutional/base_conv.py:113: UserWarning: Do not pass an
`input_shape`/`input_dim` argument to a layer. When using Sequential
models, prefer using an `Input(shape)` object as the first layer in
the model instead.
  super().__init__(activity_regularizer=activity_regularizer,
**kwargs)
```

```
Model: "sequential"

┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━
━━━━━┓
┃ Layer (type)                    ┃ Output Shape           ┃
Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━
━━━━━┩
│ conv2d (Conv2D)                 │ (None, 32, 32, 64)     │
1,792 │
├─────────────────────────────────┼────────────────────────┼──────────
─────┤
│ batch_normalization             │ (None, 32, 32, 64)     │
256 │
│ (BatchNormalization)            │                        │
│
├─────────────────────────────────┼────────────────────────┼──────────
─────┤
│ max_pooling2d (MaxPooling2D)    │ (None, 16, 16, 64)     │
0 │
├─────────────────────────────────┼────────────────────────┼──────────
```

```
───────┤
│ conv2d_1 (Conv2D)                │ (None, 16, 16, 128)     │
73,856 │
├──────────────────────────────────┼─────────────────────────┼────────
───────┤
│ batch_normalization_1            │ (None, 16, 16, 128)     │
512 │
│ (BatchNormalization)             │                         │
│
├──────────────────────────────────┼─────────────────────────┼────────
───────┤
│ max_pooling2d_1 (MaxPooling2D)   │ (None, 8, 8, 128)       │
0 │
├──────────────────────────────────┼─────────────────────────┼────────
───────┤
│ conv2d_2 (Conv2D)                │ (None, 8, 8, 256)       │
295,168 │
├──────────────────────────────────┼─────────────────────────┼────────
───────┤
│ conv2d_3 (Conv2D)                │ (None, 8, 8, 256)       │
590,080 │
├──────────────────────────────────┼─────────────────────────┼────────
───────┤
│ max_pooling2d_2 (MaxPooling2D)   │ (None, 4, 4, 256)       │
0 │
├──────────────────────────────────┼─────────────────────────┼────────
───────┤
│ flatten (Flatten)                │ (None, 4096)            │
0 │
├──────────────────────────────────┼─────────────────────────┼────────
───────┤
│ dense (Dense)                    │ (None, 512)             │
2,097,664 │
├──────────────────────────────────┼─────────────────────────┼────────
───────┤
│ dropout (Dropout)                │ (None, 512)             │
0 │
├──────────────────────────────────┼─────────────────────────┼────────
───────┤
│ dense_1 (Dense)                  │ (None, 256)             │
131,328 │
├──────────────────────────────────┼─────────────────────────┼────────
───────┤
│ dropout_1 (Dropout)              │ (None, 256)             │
0 │
├──────────────────────────────────┼─────────────────────────┼────────
───────┤
│ dense_2 (Dense)                  │ (None, 10)              │
2,570 │
```

```
└──────────────────────────────────┴─────────────────────────┴──────────
─────┘

 Total params: 3,193,226 (12.18 MB)

 Trainable params: 3,192,842 (12.18 MB)

 Non-trainable params: 384 (1.50 KB)

Epoch 1/30
782/782 ━━━━━━━━━━━━━━━━━━━━ 25s 20ms/step - accuracy: 0.3494 - loss:
1.8008 - val_accuracy: 0.4600 - val_loss: 1.4928 - learning_rate:
0.0010
Epoch 2/30
782/782 ━━━━━━━━━━━━━━━━━━━━ 9s 11ms/step - accuracy: 0.6044 - loss:
1.1357 - val_accuracy: 0.5929 - val_loss: 1.1774 - learning_rate:
0.0010
Epoch 3/30
782/782 ━━━━━━━━━━━━━━━━━━━━ 9s 11ms/step - accuracy: 0.6819 - loss:
0.9192 - val_accuracy: 0.6518 - val_loss: 1.0075 - learning_rate:
0.0010
Epoch 4/30
782/782 ━━━━━━━━━━━━━━━━━━━━ 9s 11ms/step - accuracy: 0.7315 - loss:
0.7924 - val_accuracy: 0.6450 - val_loss: 1.0349 - learning_rate:
0.0010
Epoch 5/30
782/782 ━━━━━━━━━━━━━━━━━━━━ 9s 11ms/step - accuracy: 0.7634 - loss:
0.6959 - val_accuracy: 0.7285 - val_loss: 0.7965 - learning_rate:
0.0010
Epoch 6/30
782/782 ━━━━━━━━━━━━━━━━━━━━ 11s 14ms/step - accuracy: 0.7925 - loss:
0.6165 - val_accuracy: 0.7348 - val_loss: 0.7737 - learning_rate:
0.0010
Epoch 7/30
782/782 ━━━━━━━━━━━━━━━━━━━━ 9s 11ms/step - accuracy: 0.8130 - loss:
0.5524 - val_accuracy: 0.7418 - val_loss: 0.7834 - learning_rate:
0.0010
Epoch 8/30
782/782 ━━━━━━━━━━━━━━━━━━━━ 9s 11ms/step - accuracy: 0.8353 - loss:
0.4909 - val_accuracy: 0.7059 - val_loss: 0.8991 - learning_rate:
0.0010
Epoch 9/30
778/782 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - accuracy: 0.8503 - loss:
0.4459
Epoch 9: ReduceLROnPlateau reducing learning rate to
0.0005000000237487257.
782/782 ━━━━━━━━━━━━━━━━━━━━ 9s 12ms/step - accuracy: 0.8502 - loss:
0.4460 - val_accuracy: 0.7364 - val_loss: 0.8328 - learning_rate:
0.0010
Epoch 10/30
```

```
782/782 ━━━━━━━━━━━━━━━━━━━━ 9s 11ms/step - accuracy: 0.8962 - loss:
0.3093 - val_accuracy: 0.7797 - val_loss: 0.7793 - learning_rate:
5.0000e-04
Epoch 11/30
782/782 ━━━━━━━━━━━━━━━━━━━━ 9s 11ms/step - accuracy: 0.9318 - loss:
0.2015 - val_accuracy: 0.7876 - val_loss: 0.8387 - learning_rate:
5.0000e-04
Epoch 12/30
777/782 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - accuracy: 0.9432 - loss:
0.1635
Epoch 12: ReduceLROnPlateau reducing learning rate to
0.0002500000118743628.
782/782 ━━━━━━━━━━━━━━━━━━━━ 9s 11ms/step - accuracy: 0.9431 - loss:
0.1636 - val_accuracy: 0.7737 - val_loss: 0.9555 - learning_rate:
5.0000e-04
Epoch 13/30
782/782 ━━━━━━━━━━━━━━━━━━━━ 9s 11ms/step - accuracy: 0.9621 - loss:
0.1109 - val_accuracy: 0.7904 - val_loss: 0.9733 - learning_rate:
2.5000e-04
Epoch 13: early stopping
Restoring model weights from the end of the best epoch: 6.
313/313 - 2s - 6ms/step - accuracy: 0.7348 - loss: 0.7737

WARNING:absl:You are saving your model as an HDF5 file via
`model.save()` or `keras.saving.save_model(model)`. This file format
is considered legacy. We recommend using instead the native Keras
format, e.g. `model.save('my_model.keras')` or
`keras.saving.save_model(model, 'my_model.keras')`.

Test accuracy: 0.7348
Model saved as alexnet_cifar10.h5
```

MNIST

```python
import numpy as np
import tensorflow as tf
from tensorflow.keras import datasets, layers, models, optimizers,
callbacks

# -------------------- Load & preprocess MNIST --------------------
(train_images, train_labels), (test_images, test_labels) =
datasets.mnist.load_data()

# MNIST is (N,28,28) grayscale; expand channel dim to (28,28,1)
train_images = np.expand_dims(train_images, -1).astype('float32') /
255.0
test_images = np.expand_dims(test_images, -1).astype('float32') /
255.0

print("MNIST dataset loaded successfully.")
```

```python
print(f"Train images shape: {train_images.shape}")
print(f"Test images shape: {test_images.shape}")

# -------------------- AlexNet-style model for MNIST
--------------------
def build_alexnet_mnist(input_shape=(28,28,1), num_classes=10):
    model = models.Sequential([
        layers.Conv2D(32, (3,3), activation='relu',
input_shape=input_shape, padding='same'),
        layers.BatchNormalization(),
        layers.MaxPooling2D((2,2)),

        layers.Conv2D(64, (3,3), activation='relu', padding='same'),
        layers.BatchNormalization(),
        layers.MaxPooling2D((2,2)),

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

        layers.Flatten(),
        layers.Dense(256, activation='relu'),
        layers.Dropout(0.5),
        layers.Dense(128, activation='relu'),
        layers.Dropout(0.5),
        layers.Dense(num_classes, activation='softmax')
    ])
    return model

model = build_alexnet_mnist()
model.summary()

# -------------------- Compile --------------------
model.compile(optimizer=optimizers.Adam(learning_rate=1e-3),
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

# -------------------- Callbacks --------------------
cb = [
    callbacks.ReduceLROnPlateau(monitor='val_loss', factor=0.5,
patience=3, verbose=1),
    callbacks.EarlyStopping(monitor='val_loss', patience=7,
restore_best_weights=True, verbose=1)
]

# -------------------- Train --------------------
history = model.fit(train_images, train_labels,
                    epochs=25,
                    batch_size=128,
                    validation_data=(test_images, test_labels),
```

```
                    callbacks=cb)

# -------------------- Evaluate & Save --------------------
test_loss, test_acc = model.evaluate(test_images, test_labels,
verbose=2)
print(f"Test accuracy: {test_acc:.4f}")

model.save('alexnet_mnist.h5')
print("Model saved as alexnet_mnist.h5")
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-
keras-datasets/mnist.npz
11490434/11490434 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
MNIST dataset loaded successfully.
Train images shape: (60000, 28, 28, 1)
Test images shape: (10000, 28, 28, 1)

/usr/local/lib/python3.12/dist-packages/keras/src/layers/
convolutional/base_conv.py:113: UserWarning: Do not pass an
`input_shape`/`input_dim` argument to a layer. When using Sequential
models, prefer using an `Input(shape)` object as the first layer in
the model instead.
  super().__init__(activity_regularizer=activity_regularizer,
**kwargs)

Model: "sequential"

┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━
━━━━━┓
┃ Layer (type)                     ┃ Output Shape           ┃
Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━
━━━━━┩
│ conv2d (Conv2D)                  │ (None, 28, 28, 32)     │
320 │
├──────────────────────────────────┼────────────────────────┼───────
─────┤
│ batch_normalization              │ (None, 28, 28, 32)     │
128 │
│ (BatchNormalization)             │                        │
│
├──────────────────────────────────┼────────────────────────┼───────
─────┤
│ max_pooling2d (MaxPooling2D)     │ (None, 14, 14, 32)     │
0 │
├──────────────────────────────────┼────────────────────────┼───────
─────┤
│ conv2d_1 (Conv2D)                │ (None, 14, 14, 64)     │
18,496 │
├──────────────────────────────────┼────────────────────────┼───────
```

```
────────┤
│ batch_normalization_1           │ (None, 14, 14, 64)      │
256 │
│ (BatchNormalization)            │                         │
│
├─────────────────────────────────┼─────────────────────────┼──────────
────────┤
│ max_pooling2d_1 (MaxPooling2D)  │ (None, 7, 7, 64)        │
0 │
├─────────────────────────────────┼─────────────────────────┼──────────
────────┤
│ conv2d_2 (Conv2D)               │ (None, 7, 7, 128)       │
73,856 │
├─────────────────────────────────┼─────────────────────────┼──────────
────────┤
│ conv2d_3 (Conv2D)               │ (None, 7, 7, 128)       │
147,584 │
├─────────────────────────────────┼─────────────────────────┼──────────
────────┤
│ max_pooling2d_2 (MaxPooling2D)  │ (None, 3, 3, 128)       │
0 │
├─────────────────────────────────┼─────────────────────────┼──────────
────────┤
│ flatten (Flatten)               │ (None, 1152)            │
0 │
├─────────────────────────────────┼─────────────────────────┼──────────
────────┤
│ dense (Dense)                   │ (None, 256)             │
295,168 │
├─────────────────────────────────┼─────────────────────────┼──────────
────────┤
│ dropout (Dropout)               │ (None, 256)             │
0 │
├─────────────────────────────────┼─────────────────────────┼──────────
────────┤
│ dense_1 (Dense)                 │ (None, 128)             │
32,896 │
├─────────────────────────────────┼─────────────────────────┼──────────
────────┤
│ dropout_1 (Dropout)             │ (None, 128)             │
0 │
├─────────────────────────────────┼─────────────────────────┼──────────
────────┤
│ dense_2 (Dense)                 │ (None, 10)              │
1,290 │
└─────────────────────────────────┴─────────────────────────┴──────────
────────┘

 Total params: 569,994 (2.17 MB)
```

```
 Trainable params: 569,802 (2.17 MB)

 Non-trainable params: 192 (768.00 B)

Epoch 1/25
469/469 ━━━━━━━━━━━━━━━━━━━━ 18s 19ms/step - accuracy: 0.7635 - loss:
0.7319 - val_accuracy: 0.9705 - val_loss: 0.0966 - learning_rate:
0.0010
Epoch 2/25
469/469 ━━━━━━━━━━━━━━━━━━━━ 4s 8ms/step - accuracy: 0.9796 - loss:
0.0769 - val_accuracy: 0.9857 - val_loss: 0.0536 - learning_rate:
0.0010
Epoch 3/25
469/469 ━━━━━━━━━━━━━━━━━━━━ 4s 8ms/step - accuracy: 0.9852 - loss:
0.0569 - val_accuracy: 0.9872 - val_loss: 0.0498 - learning_rate:
0.0010
Epoch 4/25
469/469 ━━━━━━━━━━━━━━━━━━━━ 4s 8ms/step - accuracy: 0.9883 - loss:
0.0442 - val_accuracy: 0.9905 - val_loss: 0.0426 - learning_rate:
0.0010
Epoch 5/25
469/469 ━━━━━━━━━━━━━━━━━━━━ 5s 8ms/step - accuracy: 0.9909 - loss:
0.0335 - val_accuracy: 0.9849 - val_loss: 0.0512 - learning_rate:
0.0010
Epoch 6/25
469/469 ━━━━━━━━━━━━━━━━━━━━ 4s 8ms/step - accuracy: 0.9916 - loss:
0.0306 - val_accuracy: 0.9926 - val_loss: 0.0298 - learning_rate:
0.0010
Epoch 7/25
469/469 ━━━━━━━━━━━━━━━━━━━━ 4s 8ms/step - accuracy: 0.9933 - loss:
0.0266 - val_accuracy: 0.9899 - val_loss: 0.0404 - learning_rate:
0.0010
Epoch 8/25
469/469 ━━━━━━━━━━━━━━━━━━━━ 5s 10ms/step - accuracy: 0.9936 - loss:
0.0227 - val_accuracy: 0.9923 - val_loss: 0.0302 - learning_rate:
0.0010
Epoch 9/25
465/469 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9944 - loss:
0.0201
Epoch 9: ReduceLROnPlateau reducing learning rate to
0.0005000000237487257.
469/469 ━━━━━━━━━━━━━━━━━━━━ 4s 8ms/step - accuracy: 0.9944 - loss:
0.0202 - val_accuracy: 0.9917 - val_loss: 0.0323 - learning_rate:
0.0010
Epoch 10/25
469/469 ━━━━━━━━━━━━━━━━━━━━ 4s 8ms/step - accuracy: 0.9954 - loss:
0.0162 - val_accuracy: 0.9941 - val_loss: 0.0258 - learning_rate:
5.0000e-04
Epoch 11/25
469/469 ━━━━━━━━━━━━━━━━━━━━ 4s 8ms/step - accuracy: 0.9986 - loss:
```

```
0.0049 - val_accuracy: 0.9947 - val_loss: 0.0271 - learning_rate:
5.0000e-04
Epoch 12/25
469/469 ━━━━━━━━━━━━━━━━━━━━ 5s 8ms/step - accuracy: 0.9979 - loss:
0.0071 - val_accuracy: 0.9953 - val_loss: 0.0265 - learning_rate:
5.0000e-04
Epoch 13/25
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9992 - loss:
0.0030
Epoch 13: ReduceLROnPlateau reducing learning rate to
0.0002500000118743628.
469/469 ━━━━━━━━━━━━━━━━━━━━ 4s 8ms/step - accuracy: 0.9992 - loss:
0.0030 - val_accuracy: 0.9925 - val_loss: 0.0433 - learning_rate:
5.0000e-04
Epoch 14/25
469/469 ━━━━━━━━━━━━━━━━━━━━ 4s 8ms/step - accuracy: 0.9988 - loss:
0.0046 - val_accuracy: 0.9952 - val_loss: 0.0280 - learning_rate:
2.5000e-04
Epoch 15/25
469/469 ━━━━━━━━━━━━━━━━━━━━ 4s 8ms/step - accuracy: 0.9995 - loss:
0.0022 - val_accuracy: 0.9947 - val_loss: 0.0340 - learning_rate:
2.5000e-04
Epoch 16/25
466/469 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9995 - loss:
0.0017
Epoch 16: ReduceLROnPlateau reducing learning rate to
0.0001250000059371814.
469/469 ━━━━━━━━━━━━━━━━━━━━ 4s 8ms/step - accuracy: 0.9995 - loss:
0.0017 - val_accuracy: 0.9951 - val_loss: 0.0357 - learning_rate:
2.5000e-04
Epoch 17/25
469/469 ━━━━━━━━━━━━━━━━━━━━ 4s 9ms/step - accuracy: 0.9999 - loss:
6.2019e-04 - val_accuracy: 0.9949 - val_loss: 0.0365 - learning_rate:
1.2500e-04
Epoch 17: early stopping
Restoring model weights from the end of the best epoch: 10.
313/313 - 2s - 7ms/step - accuracy: 0.9941 - loss: 0.0258

WARNING:absl:You are saving your model as an HDF5 file via
`model.save()` or `keras.saving.save_model(model)`. This file format
is considered legacy. We recommend using instead the native Keras
format, e.g. `model.save('my_model.keras')` or
`keras.saving.save_model(model, 'my_model.keras')`.

Test accuracy: 0.9941
Model saved as alexnet_mnist.h5
```

GoogleNet

CIFAR-10

```
import numpy as np
import tensorflow as tf
from tensorflow.keras import datasets, layers, models, optimizers,
callbacks

# -------------------- Load & preprocess CIFAR-10 --------------------
(train_images, train_labels), (test_images, test_labels) =
datasets.cifar10.load_data()
train_images, test_images = train_images.astype('float32') / 255.0,
test_images.astype('float32') / 255.0

print("CIFAR-10 dataset loaded successfully.")
print(f"Train images shape: {train_images.shape}")
print(f"Test images shape: {test_images.shape}")

# -------------------- Inception / GoogLeNet-style helper
--------------------
def inception_module(x, f1, f3r, f3, f5r, f5, fp):
    b1 = layers.Conv2D(f1, (1,1), padding='same', activation='relu')
(x)

    b3 = layers.Conv2D(f3r, (1,1), padding='same', activation='relu')
(x)
    b3 = layers.Conv2D(f3, (3,3), padding='same', activation='relu')
(b3)

    b5 = layers.Conv2D(f5r, (1,1), padding='same', activation='relu')
(x)
    b5 = layers.Conv2D(f5, (3,3), padding='same', activation='relu')
(b5)
    b5 = layers.Conv2D(f5, (3,3), padding='same', activation='relu')
(b5)

    bp = layers.MaxPooling2D((3,3), strides=1, padding='same')(x)
    bp = layers.Conv2D(fp, (1,1), padding='same', activation='relu')
(bp)

    return layers.Concatenate(axis=-1)([b1, b3, b5, bp])

# -------------------- Build compact GoogLeNet for CIFAR
--------------------
def build_googlenet_cifar(input_shape=(32,32,3), num_classes=10):
    inputs = layers.Input(shape=input_shape)

    x = layers.Conv2D(64, (3,3), padding='same', activation='relu')
(inputs)
```

```
    x = layers.BatchNormalization()(x)
    x = layers.MaxPooling2D((2,2))(x)

    x = inception_module(x, 32, 32, 64, 16, 16, 16)
    x = inception_module(x, 64, 48, 96, 16, 32, 32)
    x = layers.MaxPooling2D((2,2))(x)

    x = inception_module(x, 96, 64, 128, 16, 32, 32)
    x = layers.GlobalAveragePooling2D()(x)
    x = layers.Dropout(0.4)(x)
    outputs = layers.Dense(num_classes, activation='softmax')(x)

    model = models.Model(inputs, outputs, name='googlenet_cifar')
    return model

model = build_googlenet_cifar()
model.summary()

# -------------------- Compile --------------------
model.compile(optimizer=optimizers.Adam(learning_rate=1e-3),
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

# -------------------- Callbacks --------------------
cb = [
    callbacks.ReduceLROnPlateau(monitor='val_loss', factor=0.5,
patience=3, verbose=1),
    callbacks.EarlyStopping(monitor='val_loss', patience=7,
restore_best_weights=True, verbose=1)
]

# -------------------- Train --------------------
history = model.fit(train_images, train_labels,
                    epochs=40,
                    batch_size=64,
                    validation_data=(test_images, test_labels),
                    callbacks=cb)

# -------------------- Evaluate & Save --------------------
test_loss, test_acc = model.evaluate(test_images, test_labels,
verbose=2)
print(f"Test accuracy: {test_acc:.4f}")

model.save('googlenet_cifar10.h5')
print("Model saved as googlenet_cifar10.h5")
```

MNIST

```
import numpy as np
import tensorflow as tf
```

```
from tensorflow.keras import datasets, layers, models, optimizers,
callbacks

# -------------------- Load & preprocess MNIST --------------------
(train_images, train_labels), (test_images, test_labels) =
datasets.mnist.load_data()

# Expand to (N,28,28,1) and normalize
train_images = np.expand_dims(train_images, -1).astype('float32') /
255.0
test_images = np.expand_dims(test_images, -1).astype('float32') /
255.0

print("MNIST dataset loaded successfully.")
print(f"Train images shape: {train_images.shape}")
print(f"Test images shape: {test_images.shape}")

# -------------------- Inception helper (works with 1-channel too)
--------------------
def inception_module(x, f1, f3r, f3, f5r, f5, fp):
    b1 = layers.Conv2D(f1, (1,1), padding='same', activation='relu')
(x)

    b3 = layers.Conv2D(f3r, (1,1), padding='same', activation='relu')
(x)
    b3 = layers.Conv2D(f3, (3,3), padding='same', activation='relu')
(b3)

    b5 = layers.Conv2D(f5r, (1,1), padding='same', activation='relu')
(x)
    b5 = layers.Conv2D(f5, (3,3), padding='same', activation='relu')
(b5)
    b5 = layers.Conv2D(f5, (3,3), padding='same', activation='relu')
(b5)

    bp = layers.MaxPooling2D((3,3), strides=1, padding='same')(x)
    bp = layers.Conv2D(fp, (1,1), padding='same', activation='relu')
(bp)

    return layers.Concatenate(axis=-1)([b1, b3, b5, bp])

# -------------------- Build compact GoogLeNet for MNIST
--------------------
def build_googlenet_mnist(input_shape=(28,28,1), num_classes=10):
    inputs = layers.Input(shape=input_shape)

    x = layers.Conv2D(32, (3,3), padding='same', activation='relu')
(inputs)
    x = layers.BatchNormalization()(x)
    x = layers.MaxPooling2D((2,2))(x)
```

```
    x = inception_module(x, 16, 16, 32, 8, 8, 8)
    x = inception_module(x, 32, 24, 48, 8, 16, 16)
    x = layers.MaxPooling2D((2,2))(x)

    x = inception_module(x, 48, 32, 64, 8, 16, 16)
    x = layers.GlobalAveragePooling2D()(x)
    x = layers.Dropout(0.4)(x)
    outputs = layers.Dense(num_classes, activation='softmax')(x)

    model = models.Model(inputs, outputs, name='googlenet_mnist')
    return model

model = build_googlenet_mnist()
model.summary()

# -------------------- Compile --------------------
model.compile(optimizer=optimizers.Adam(learning_rate=1e-3),
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

# -------------------- Callbacks --------------------
cb = [
    callbacks.ReduceLROnPlateau(monitor='val_loss', factor=0.5,
patience=3, verbose=1),
    callbacks.EarlyStopping(monitor='val_loss', patience=7,
restore_best_weights=True, verbose=1)
]

# -------------------- Train --------------------
history = model.fit(train_images, train_labels,
                    epochs=30,
                    batch_size=128,
                    validation_data=(test_images, test_labels),
                    callbacks=cb)

# -------------------- Evaluate & Save --------------------
test_loss, test_acc = model.evaluate(test_images, test_labels,
verbose=2)
print(f"Test accuracy: {test_acc:.4f}")

model.save('googlenet_mnist.h5')
print("Model saved as googlenet_mnist.h5")
```