MACHINE LEARNING LABORATORY

ASSIGNMENT 3
Name: Atmik Goswami
Roll No: 002211001104
Section: A2

Github: github.com/atmikgoswami/ML-Lab

Question 1

DATASETS:

1. **Ionosphere Dataset**
   - Features: 34
   - Classes: good(g) and bad(b)
   - Total Samples: 351

2. **Wisconsin Breast Cancer Dataset**
   - Features: 30
   - Classes: 0 or 1
   - Total Samples: 569

Implement Hidden Markov Model (HMM)

Code:

```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, confusion_matrix,
accuracy_score, precision_score, recall_score, f1_score, roc_curve, auc
from sklearn.preprocessing import StandardScaler, MinMaxScaler
from sklearn.model_selection import GridSearchCV, learning_curve
from sklearn.preprocessing import KBinsDiscretizer, LabelEncoder
from hmmlearn import hmm
from tqdm import tqdm
from IPython.display import display

from ucimlrepo import fetch_ucirepo
```

```
ionosphere = fetch_ucirepo(id=52)
X_continuous = ionosphere.data.features
y = ionosphere.data.targets

X.head()

n_bins = 4
discretizer = KBinsDiscretizer(n_bins=n_bins, encode='ordinal',
strategy='quantile', subsample=None)
X_discrete = discretizer.fit_transform(X_continuous).astype(int)

n_features = X_discrete.shape[1]
powers = np.array([n_bins**i for i in range(n_features)])
X_univariate = (X_discrete * powers).sum(axis=1)

label_encoder = LabelEncoder()
X_encoded = label_encoder.fit_transform(X_univariate)

X_final = X_encoded.reshape(-1, 1)
X_train, X_test, y_train, y_test = train_test_split(
    X_final, y, test_size=0.20, random_state=42, stratify=y
)

classifier = hmm.MultinomialHMM(n_components=2, n_iter=1000,
random_state=42)
classifier.fit(X_train)

# 8. Predict and Evaluate
y_pred = classifier.predict(X_test)
y_test_enc = y_test['Class'].map({'g': 0, 'b': 1})

print("Confusion Matrix (MultinomialHMM with Combined & Encoded
Features):")
print(confusion_matrix(y_test_enc, y_pred))

print("\nPerformance Evaluation:")
print(classification_report(y_test_enc, y_pred))

classifier = hmm.GaussianHMM(n_components=2, covariance_type="full",
algorithm="viterbi" n_iter=1000, random_state=22)

classifier.fit(X_train)

y_pred = classifier.predict(X_test)

y_test_enc = y_test['Class'].map({'g': 0, 'b': 1})

print("Confusion Matrix:")
print(confusion_matrix(y_test_enc, y_pred))
```

```
print("----------------------------------------------------")
print("Performance Evaluation:")
print(classification_report(y_test_enc, y_pred))

splits = [0.5, 0.4, 0.3, 0.2]
results = []

for test_size in splits:
    print(f"\n=== Train-Test Split:
{int((1-test_size)*100)}:{int(test_size*100)} ===")

    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=test_size, random_state=42, stratify=y
    )

    sc = StandardScaler()
    X_train = sc.fit_transform(X_train)
    X_test = sc.transform(X_test)

    # Train
    classifier = hmm.GaussianHMM(n_components=2, random_state=22)

    param_grid = {
        'covariance_type': ['spherical', 'diag', 'full', 'tied'],
        'algorithm': ['viterbi', 'map'],
        'n_iter': [500, 1000, 5000]
    }

    grid = GridSearchCV(classifier, param_grid, cv=5)
    grid.fit(X_train)

    best_model = grid.best_estimator_
    print("Best Parameters:", grid.best_params_)

    y_pred = best_model.predict(X_test)
    y_proba = best_model.predict_proba(X_test)

    y_test_enc = y_test['Class'].map({'g': 0, 'b': 1})

    # Metrics
    acc = accuracy_score(y_test_enc, y_pred)
    prec = precision_score(y_test_enc, y_pred, average="weighted")
    rec = recall_score(y_test_enc, y_pred, average="weighted")
    f1 = f1_score(y_test_enc, y_pred, average="weighted")
    results.append([test_size, acc, prec, rec, f1])

    print(classification_report(y_test_enc, y_pred))

    # Confusion Matrix Heatmap
    plt.figure(figsize=(6,5))
```

```
    sns.heatmap(confusion_matrix(y_test_enc, y_pred), annot=True,
fmt="d", cmap="Blues")
    plt.title(f"Confusion Matrix
({int((1-test_size)*100)}:{int(test_size*100)})")
    plt.xlabel("Predicted"); plt.ylabel("Actual")
    plt.show()

    # # Learning Curve
    # train_sizes, train_scores, test_scores = learning_curve(
    #     classifier, X_train, y_train, cv=5, n_jobs=-1,
    #     train_sizes=np.linspace(0.1, 1.0, 10)
    # )
    # plt.figure()
    # plt.plot(train_sizes, np.mean(train_scores, axis=1), label="Train
Score")
    # plt.plot(train_sizes, np.mean(test_scores, axis=1),
label="Cross-val Score")
    # plt.title(f"Learning Curve
({int((1-test_size)*100)}:{int(test_size*100)})")
    # plt.xlabel("Training examples"); plt.ylabel("Accuracy")
    # plt.legend(); plt.show()

    # ROC Curve
    fpr, tpr, roc_auc = {}, {}, {}
    for i, cls in [(0,0), (1,1)]:
        fpr[i], tpr[i], _ = roc_curve(y_test_enc == cls, y_proba[:, i])
        roc_auc[i] = auc(fpr[i], tpr[i])

    plt.figure()
    for i, cls in [(0,0), (1,1)]:
        plt.plot(fpr[i], tpr[i], label=f"Class {cls}
(AUC={roc_auc[i]:.2f})")
    plt.plot([0,1],[0,1],"k--")
    plt.title(f"ROC Curve
({int((1-test_size)*100)}:{int(test_size*100)})")
    plt.xlabel("False Positive Rate"); plt.ylabel("True Positive Rate")
    plt.legend(); plt.show()

results_df = pd.DataFrame(results, columns=["Test Size", "Accuracy",
"Precision", "Recall", "F1"])
display(results_df)

from sklearn.datasets import load_breast_cancer

data = load_breast_cancer()

X = data.data
y = data.target

X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.20)
```

```
msc = MinMaxScaler()
X_train = msc.fit_transform(X_train)
X_test = msc.transform(X_test)

classifier = hmm.GaussianHMM(n_components=2, covariance_type="full",
algorithm="viterbi", n_iter=1000, random_state=22)

classifier.fit(X_train)

y_pred = classifier.predict(X_test)

print("Confusion Matrix:")
print(confusion_matrix(y_test, y_pred))

print("--------------------------------------------------")
print("Performance Evaluation:")
print(classification_report(y_test, y_pred))

splits = [0.5, 0.4, 0.3, 0.2]
results = []

for test_size in splits:
    print(f"\n=== Train-Test Split:
{int((1-test_size)*100)}:{int(test_size*100)} ===")

    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=test_size, random_state=42, stratify=y
    )

    sc = StandardScaler()
    X_train = sc.fit_transform(X_train)
    X_test = sc.transform(X_test)

    # Train
    classifier = hmm.GaussianHMM(n_components=2, random_state=22)

    param_grid = {
        'covariance_type': ['spherical', 'diag', 'full', 'tied'],
        'algorithm': ['viterbi', 'map'],
        'n_iter': [500, 1000, 5000]
    }

    grid = GridSearchCV(classifier, param_grid, cv=5)
    grid.fit(X_train)

    best_model = grid.best_estimator_
    print("Best Parameters:", grid.best_params_)

    y_pred = best_model.predict(X_test)
    y_proba = best_model.predict_proba(X_test)
```

```
    # Metrics
    acc = accuracy_score(y_test, y_pred)
    prec = precision_score(y_test, y_pred, average="weighted")
    rec = recall_score(y_test, y_pred, average="weighted")
    f1 = f1_score(y_test, y_pred, average="weighted")
    results.append([test_size, acc, prec, rec, f1])

    print(classification_report(y_test, y_pred))

    # Confusion Matrix Heatmap
    plt.figure(figsize=(6,5))
    sns.heatmap(confusion_matrix(y_test, y_pred), annot=True, fmt="d",
cmap="Blues")
    plt.title(f"Confusion Matrix
({int((1-test_size)*100)}:{int(test_size*100)})")
    plt.xlabel("Predicted"); plt.ylabel("Actual")
    plt.show()

    # # Learning Curve
    # train_sizes, train_scores, test_scores = learning_curve(
    #     classifier, X_train, y_train, cv=5, n_jobs=-1,
    #     train_sizes=np.linspace(0.1, 1.0, 10)
    # )
    # plt.figure()
    # plt.plot(train_sizes, np.mean(train_scores, axis=1), label="Train
Score")
    # plt.plot(train_sizes, np.mean(test_scores, axis=1),
label="Cross-val Score")
    # plt.title(f"Learning Curve
({int((1-test_size)*100)}:{int(test_size*100)})")
    # plt.xlabel("Training examples"); plt.ylabel("Accuracy")
    # plt.legend(); plt.show()

    # ROC Curve
    fpr, tpr, roc_auc = {}, {}, {}
    for i, cls in [(0,0), (1,1)]:
        fpr[i], tpr[i], _ = roc_curve(y_test == cls, y_proba[:, i])
        roc_auc[i] = auc(fpr[i], tpr[i])

    plt.figure()
    for i, cls in [(0,0), (1,1)]:
        plt.plot(fpr[i], tpr[i], label=f"Class {cls}
(AUC={roc_auc[i]:.2f})")
    plt.plot([0,1],[0,1],"k--")
    plt.title(f"ROC Curve
({int((1-test_size)*100)}:{int(test_size*100)})")
    plt.xlabel("False Positive Rate"); plt.ylabel("True Positive Rate")
    plt.legend(); plt.show()

results_df = pd.DataFrame(results, columns=["Test Size", "Accuracy",
"Precision", "Recall", "F1"])
```

```
display(results_df)

X_mean = np.mean(X, axis=0)
X_mean

X1 = X > X_mean
X2 = X1.astype(int)

X_train, X_test, y_train, y_test = train_test_split(X2, y,
test_size=0.20)

classifier = hmm.MultinomialHMM(n_components=2, algorithm="viterbi",
n_iter=1000, random_state=22)

classifier.fit(X_train)

y_pred = classifier.predict(X_test)

print("Confusion Matrix:")
print(confusion_matrix(y_test, y_pred))

print("--------------------------------------------------")
print("Performance Evaluation:")
print(classification_report(y_test, y_pred))
```

Results and Discussion:

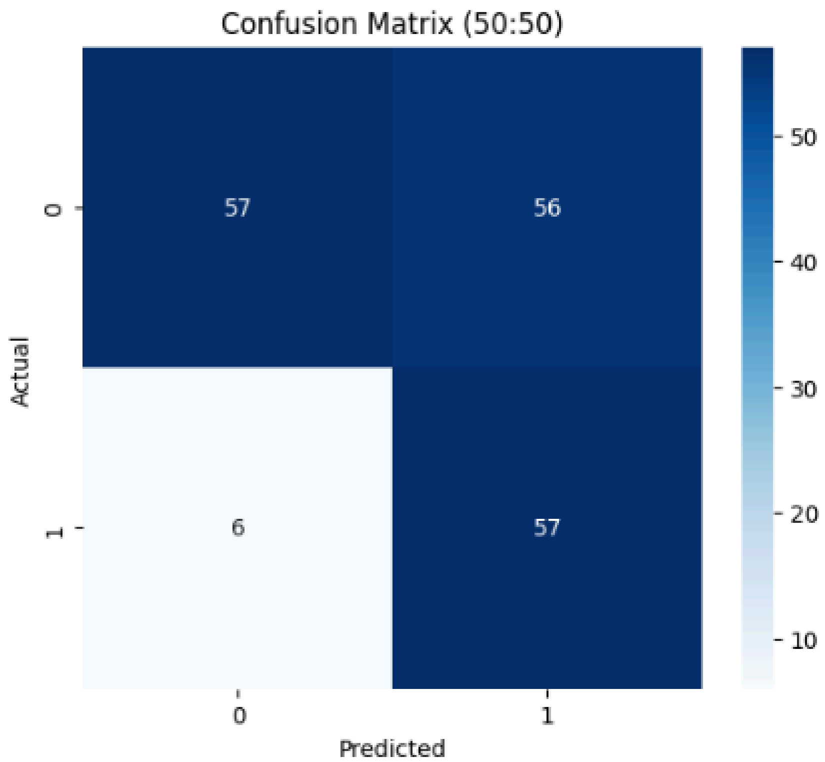
IONOSPHERE DATASET

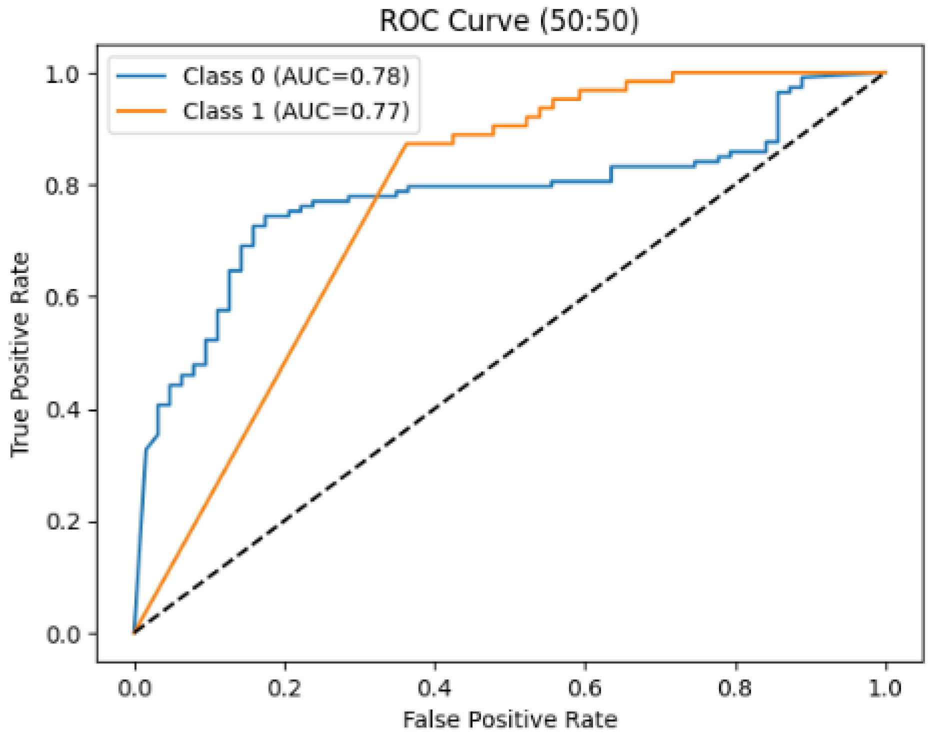
Multinomial HMM:

```
Confusion Matrix (MultinomialHMM with Combined & Encoded Features):
[[45  1]
 [25  0]]

Performance Evaluation:
              precision    recall  f1-score   support

           0       0.64      0.98      0.78        46
           1       0.00      0.00      0.00        25

    accuracy                           0.63        71
   macro avg       0.32      0.49      0.39        71
weighted avg       0.42      0.63      0.50        71
```

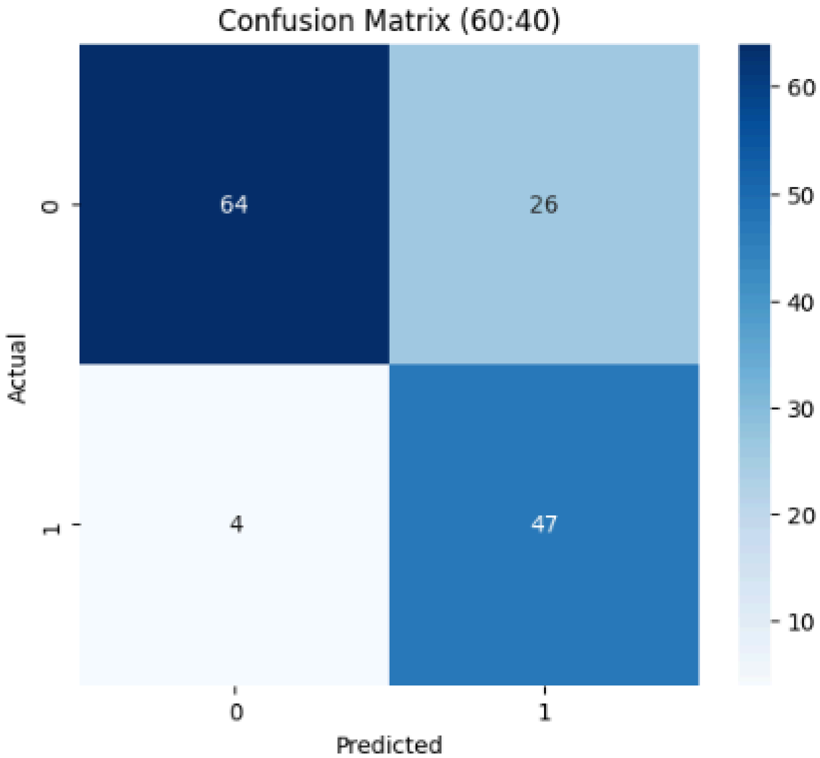
Gaussian HMM:

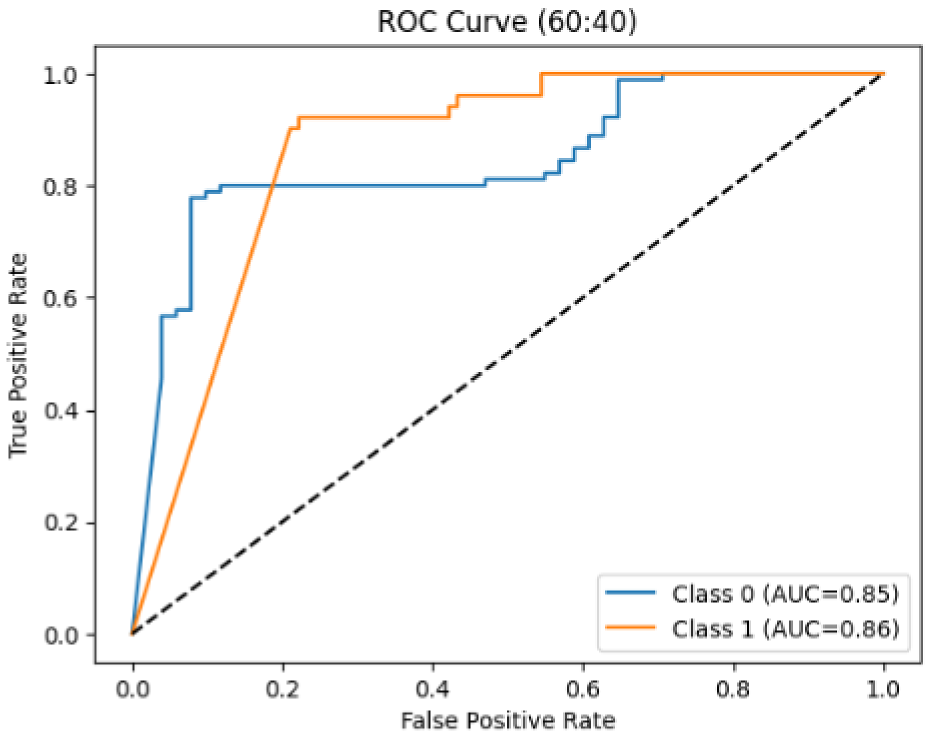
```
=== Train-Test Split: 50:50 ===
Best Parameters: {'algorithm': 'viterbi', 'covariance_type': 'diag', 'n_iter': 500}
              precision    recall  f1-score   support

           0       0.90      0.50      0.65       113
           1       0.50      0.90      0.65        63

    accuracy                           0.65       176
   macro avg       0.70      0.70      0.65       176
weighted avg       0.76      0.65      0.65       176
```



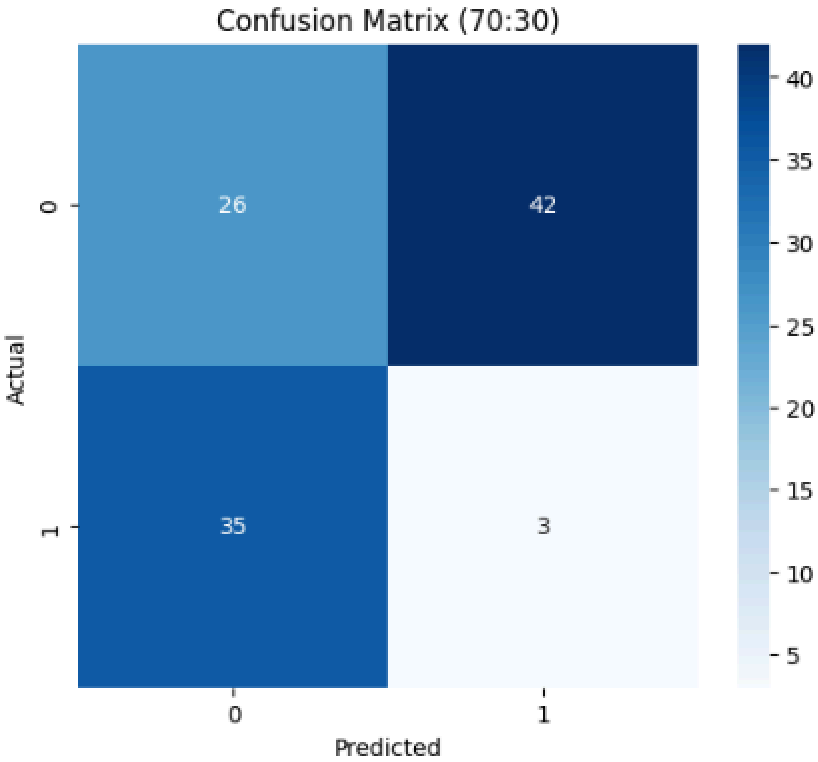

ROC Curve (50:50)
Class 0 (AUC=0.78)
Class 1 (AUC=0.77)
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0

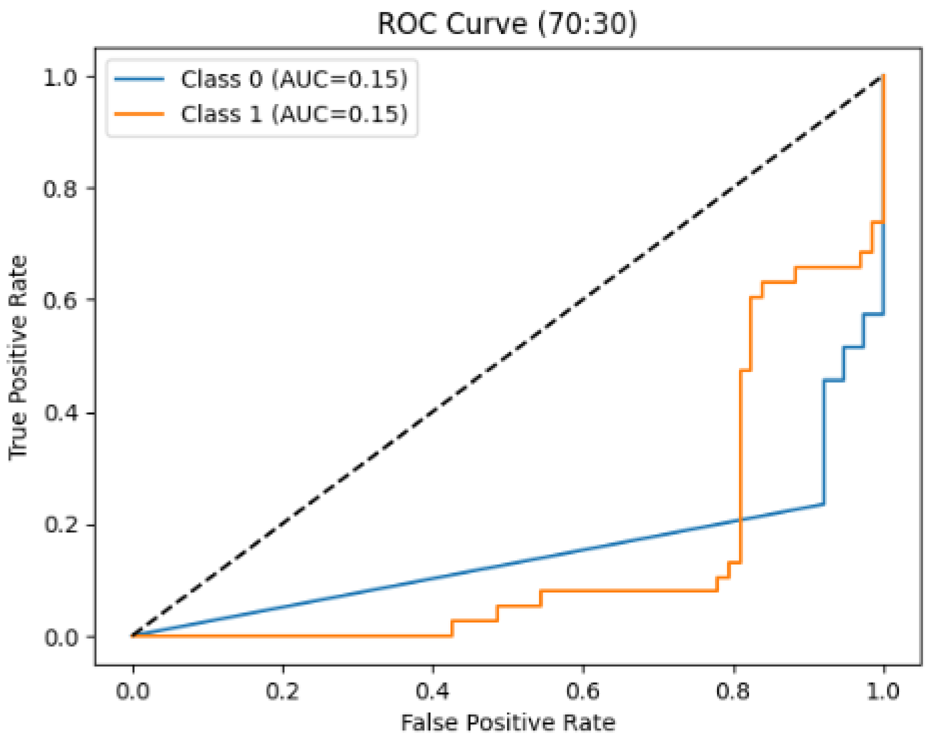
```
=== Train-Test Split: 60:40 ===
Best Parameters: {'algorithm': 'viterbi', 'covariance_type': 'diag', 'n_iter': 500}
              precision    recall  f1-score   support

           0       0.94      0.71      0.81        90
           1       0.64      0.92      0.76        51

    accuracy                           0.79       141
   macro avg       0.79      0.82      0.78       141
weighted avg       0.83      0.79      0.79       141
```



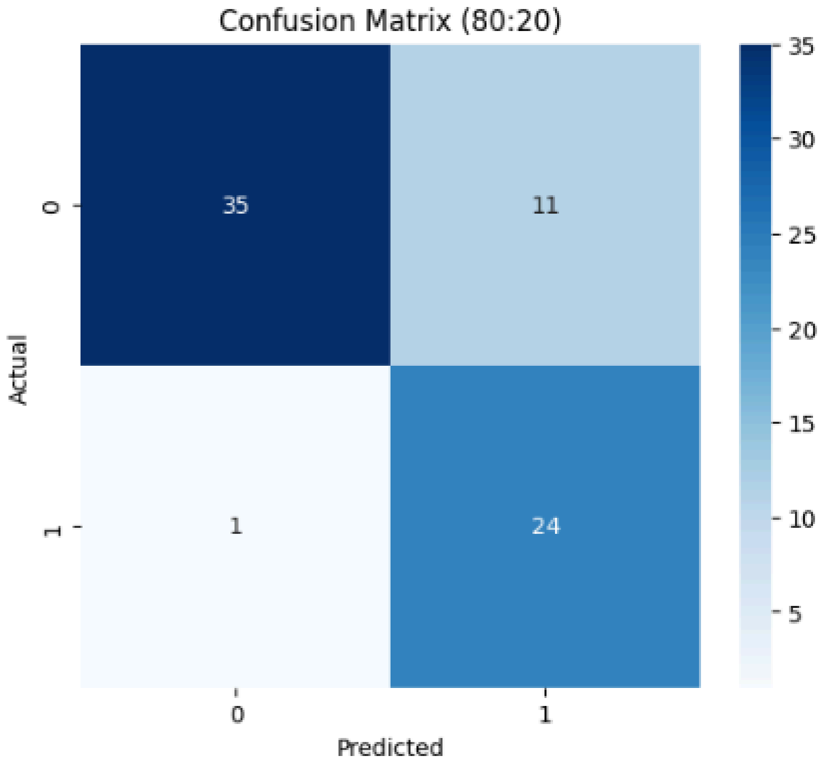

ROC Curve (60:40)
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
Class 0 (AUC=0.85)
Class 1 (AUC=0.86)

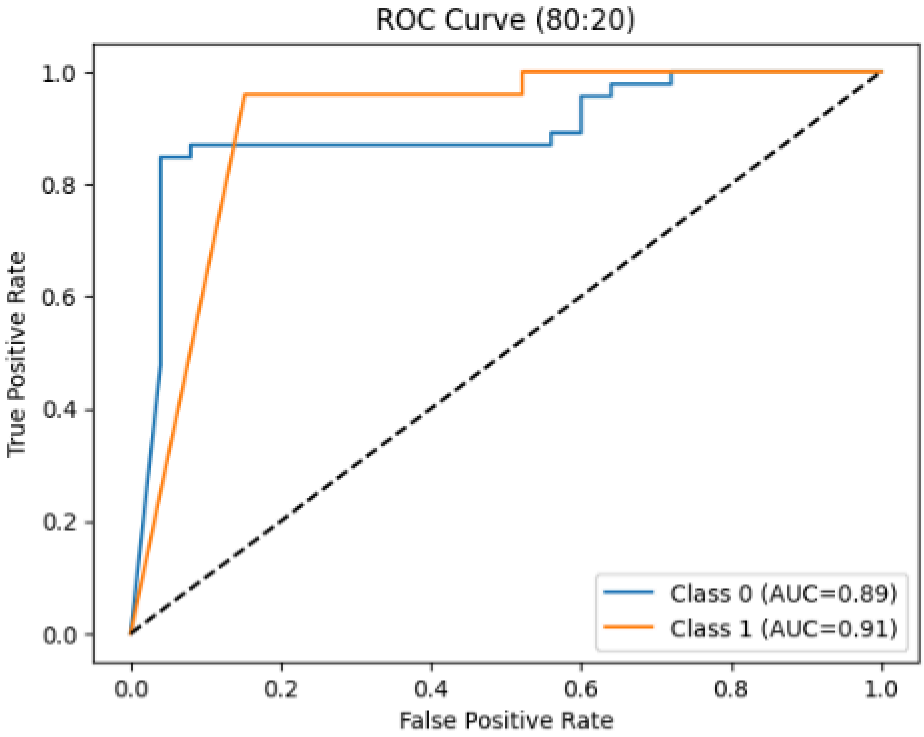
```
=== Train-Test Split: 70:30 ===
Best Parameters: {'algorithm': 'viterbi', 'covariance_type': 'diag', 'n_iter': 500}
              precision    recall  f1-score   support

           0       0.43      0.38      0.40        68
           1       0.07      0.08      0.07        38

    accuracy                           0.27       106
   macro avg       0.25      0.23      0.24       106
weighted avg       0.30      0.27      0.28       106
```




ROC Curve (70:30)
Class 0 (AUC=0.15)
Class 1 (AUC=0.15)
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0

```
=== Train-Test Split: 80:20 ===
Best Parameters: {'algorithm': 'viterbi', 'covariance_type': 'diag', 'n_iter': 500}
              precision    recall  f1-score   support

           0       0.97      0.76      0.85        46
           1       0.69      0.96      0.80        25

    accuracy                           0.83        71
   macro avg       0.83      0.86      0.83        71
weighted avg       0.87      0.83      0.83        71
```





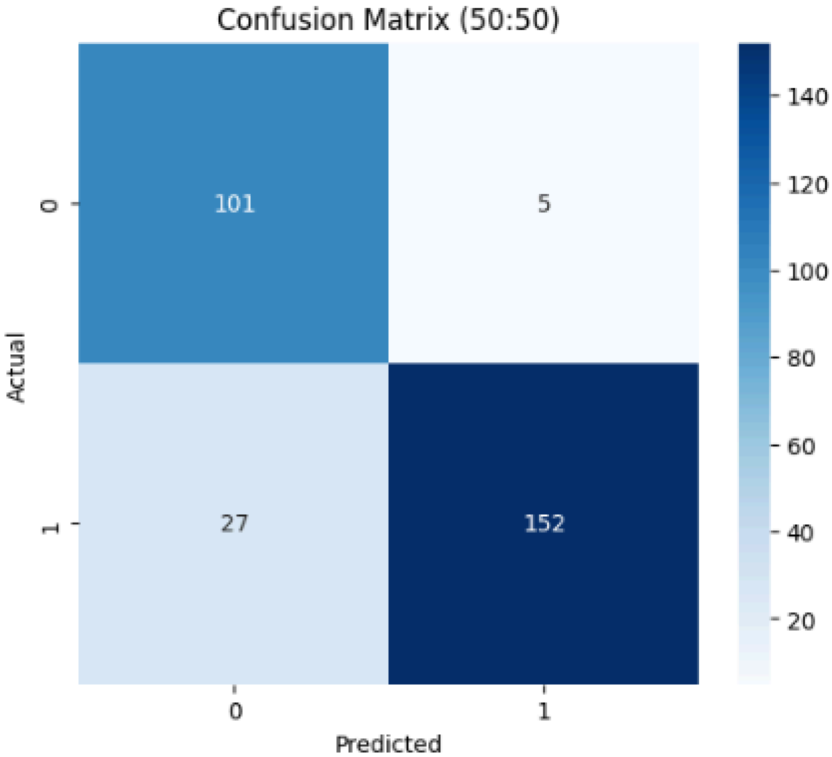
	Test Size	Accuracy	Precision	Recall	F1
0	0.5	0.647727	0.761459	0.647727	0.647727
1	0.4	0.787234	0.833628	0.787234	0.791296
2	0.3	0.273585	0.297330	0.273585	0.284508
3	0.2	0.830986	0.871339	0.830986	0.834765

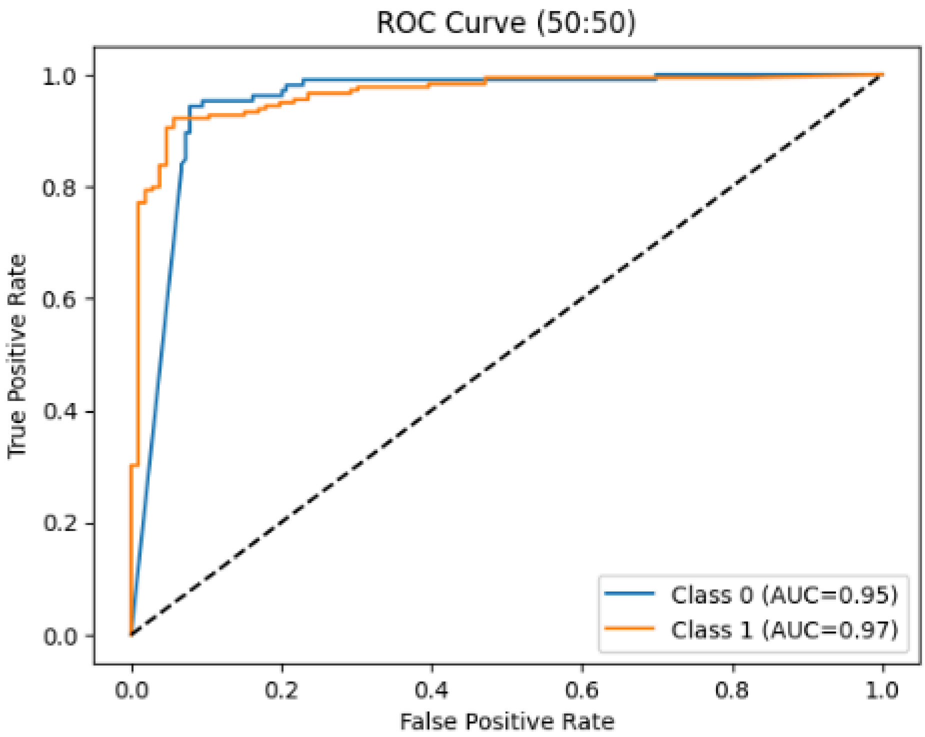
WINCONSIN BREAST CANCER DATASET

Gaussian HMM:

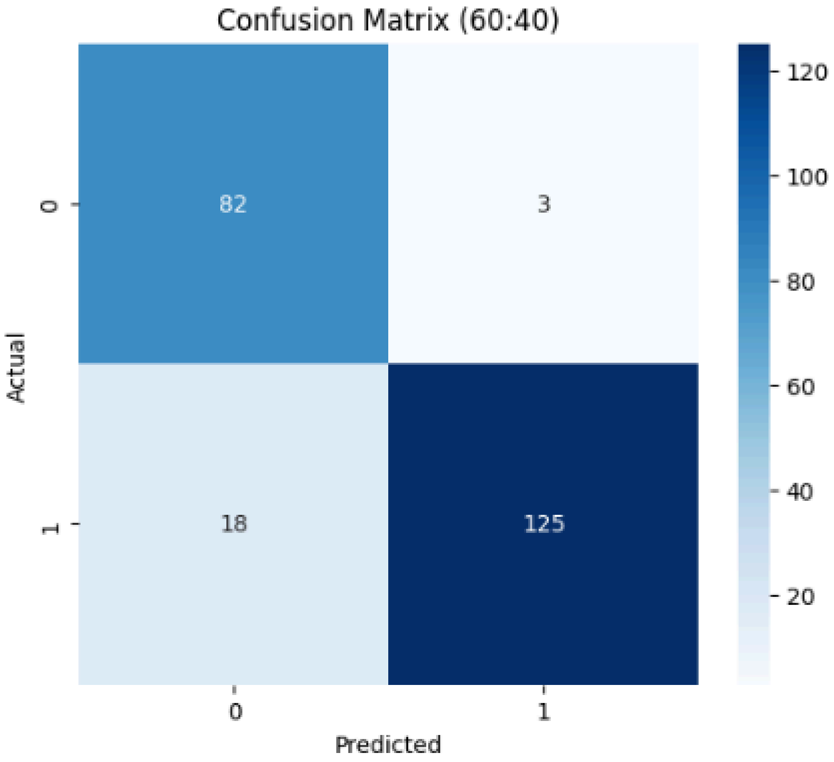
```
Confusion Matrix:
[[38  1]
 [ 4 71]]
--------------------------------------------------
Performance Evaluation:
              precision    recall  f1-score   support

           0       0.90      0.97      0.94        39
           1       0.99      0.95      0.97        75

    accuracy                           0.96       114
   macro avg       0.95      0.96      0.95       114
weighted avg       0.96      0.96      0.96       114
```

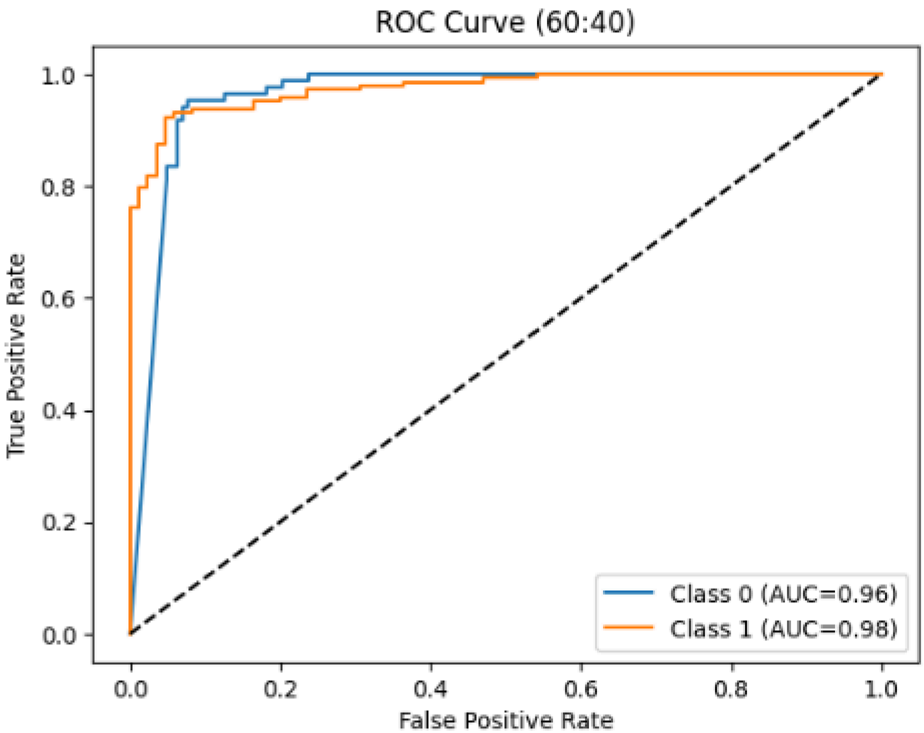
```
=== Train-Test Split: 50:50 ===
Best Parameters: {'algorithm': 'viterbi', 'covariance_type': 'full', 'n_iter': 500}
              precision    recall  f1-score   support

           0       0.79      0.95      0.86       106
           1       0.97      0.85      0.90       179

    accuracy                           0.89       285
   macro avg       0.88      0.90      0.88       285
weighted avg       0.90      0.89      0.89       285
```



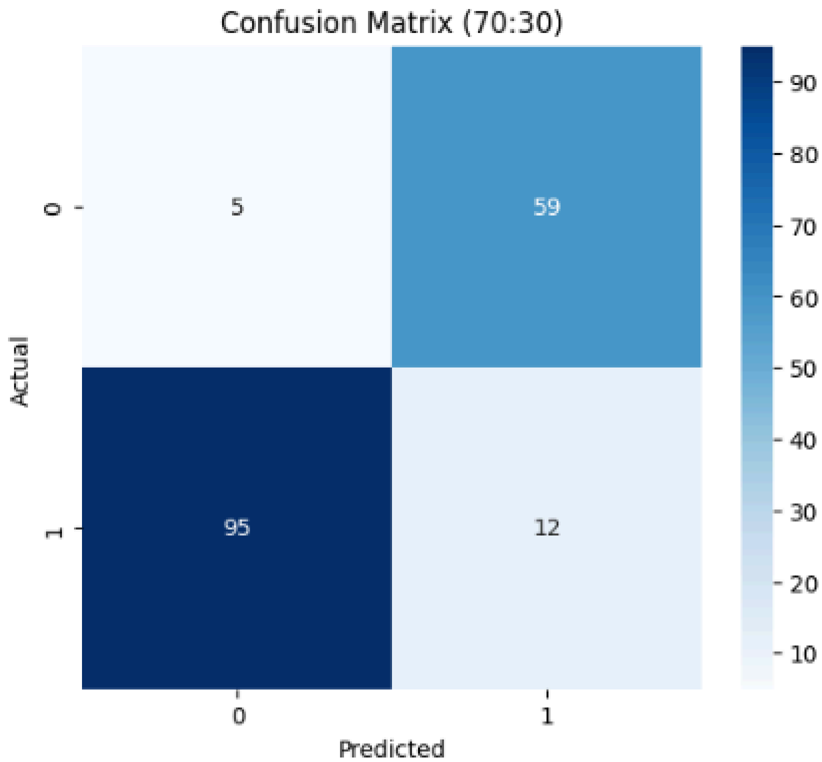


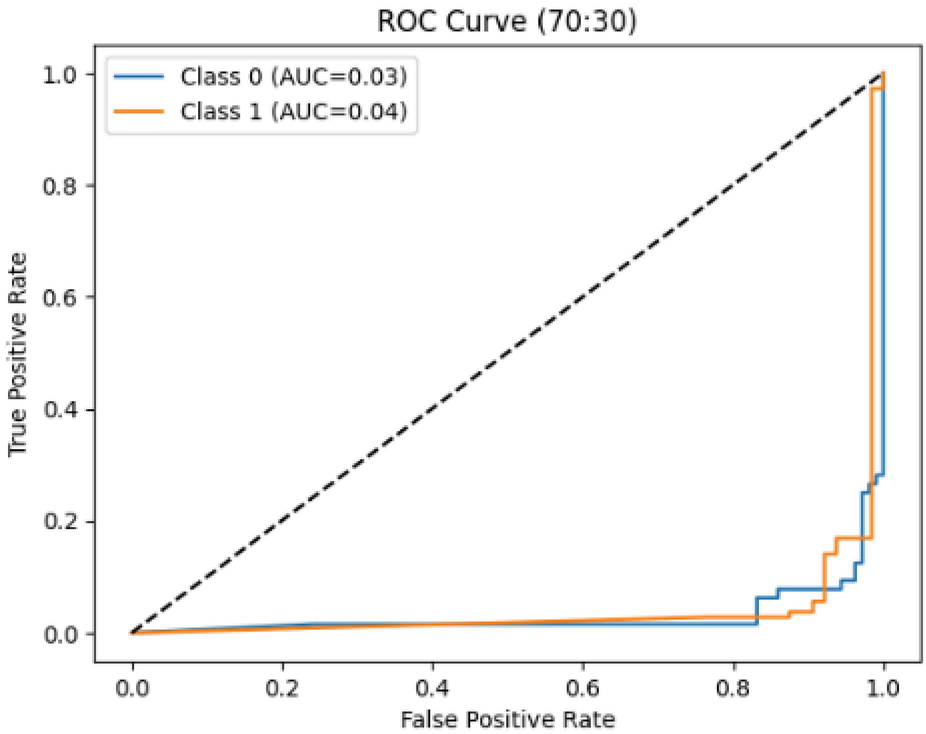

```
=== Train-Test Split: 60:40 ===
Best Parameters: {'algorithm': 'viterbi', 'covariance_type': 'full', 'n_iter': 500}
              precision    recall  f1-score   support

           0       0.82      0.96      0.89        85
           1       0.98      0.87      0.92       143

    accuracy                           0.91       228
   macro avg       0.90      0.92      0.90       228
weighted avg       0.92      0.91      0.91       228
```


Confusion Matrix (60:40)
82
3
18
125
Actual
Predicted
0
1
120
100
80
60
40
20

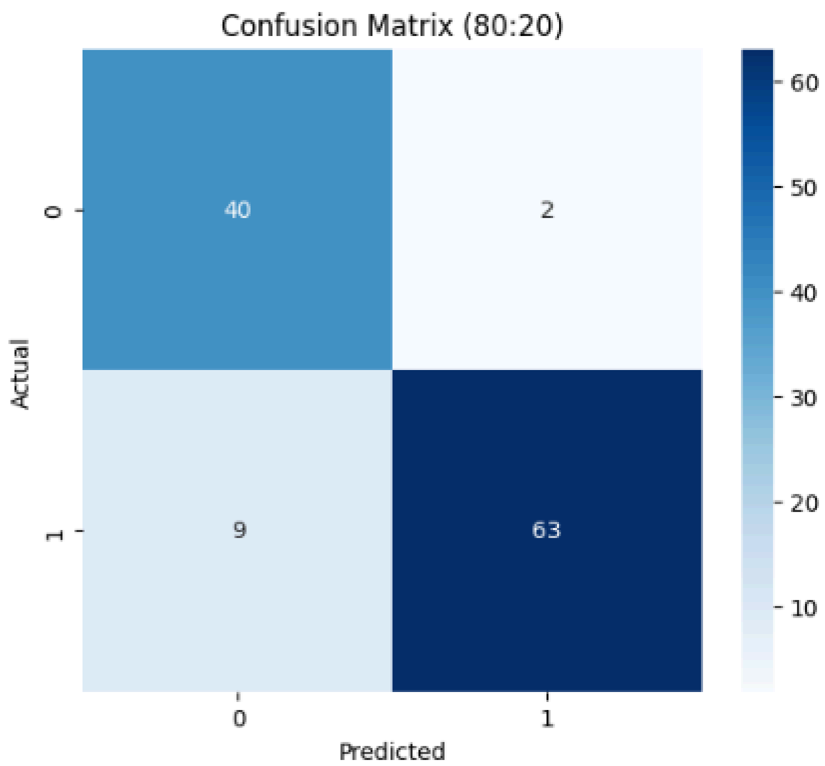

ROC Curve (60:40)
True Positive Rate
False Positive Rate
Class 0 (AUC=0.96)
Class 1 (AUC=0.98)
0.0
0.2
0.4
0.6
0.8
1.0

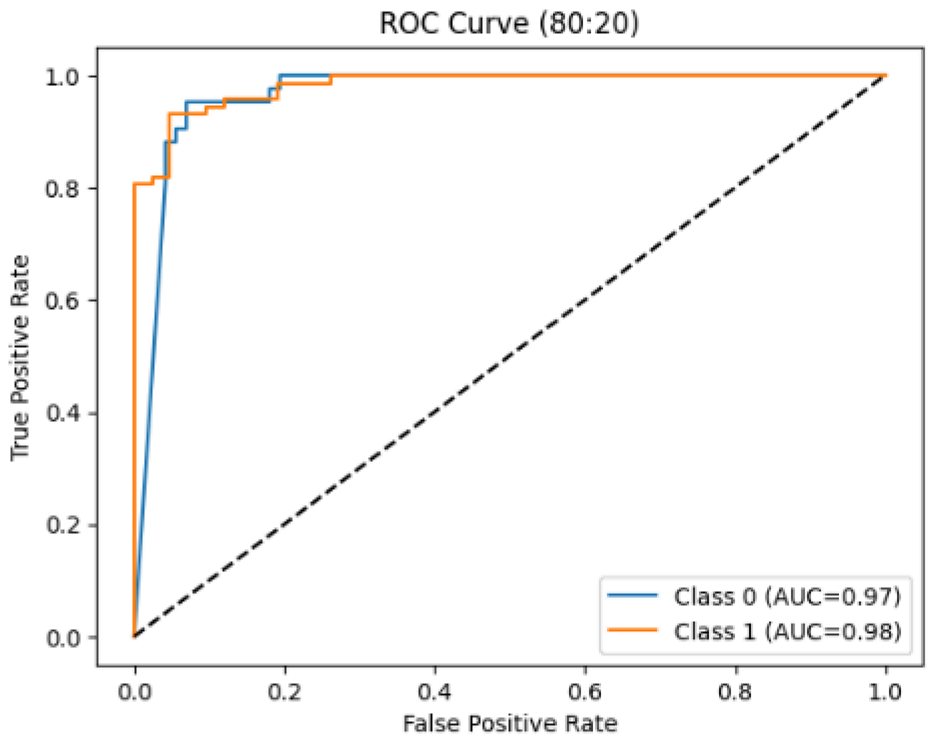
```
=== Train-Test Split: 70:30 ===
Best Parameters: {'algorithm': 'viterbi', 'covariance_type': 'full', 'n_iter': 500}
              precision    recall  f1-score   support

           0       0.05      0.08      0.06        64
           1       0.17      0.11      0.13       107

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






```
=== Train-Test Split: 80:20 ===
Best Parameters: {'algorithm': 'viterbi', 'covariance_type': 'full', 'n_iter': 500}
              precision    recall  f1-score   support

           0       0.82      0.95      0.88        42
           1       0.97      0.88      0.92        72

    accuracy                           0.90       114
   macro avg       0.89      0.91      0.90       114
weighted avg       0.91      0.90      0.90       114
```





	Test Size	Accuracy	Precision	Recall	F1
0	0.5	0.887719	0.901544	0.887719	0.889322
1	0.4	0.907895	0.918195	0.907895	0.909080
2	0.3	0.099415	0.124471	0.099415	0.107190
3	0.2	0.903509	0.912898	0.903509	0.904755

Multinomial HMM:

```
Confusion Matrix:
[[46  0]
 [11 57]]
--------------------------------------------------
Performance Evaluation:
              precision    recall  f1-score   support

           0       0.81      1.00      0.89        46
           1       1.00      0.84      0.91        68

    accuracy                           0.90       114
   macro avg       0.90      0.92      0.90       114
weighted avg       0.92      0.90      0.90       114
```

Discussion:

Ionosphere Dataset

The performance of Hidden Markov Models (HMMs) on the Ionosphere dataset varied significantly between the Multinomial and Gaussian approaches.

- Multinomial HMM: This model performed very poorly, achieving an accuracy of only 63%. The model completely failed to identify the minority class (class '1'), with its precision, recall, and F1-score all being 0.00. This indicates that the method of discretizing the continuous features using KBinsDiscretizer and then combining them was ineffective for this dataset, leading to a model that could not distinguish between the two classes .
- Gaussian HMM: This model was far more effective. The performance was evaluated across different train-test splits, with hyperparameter tuning conducted via GridSearchCV for each.
    - The 80:20 split yielded the best results, achieving an accuracy of approximately 83.1% and an F1-score of 83.5%. The ROC curves for this split showed strong performance with AUC values of 0.89 and 0.91 for the two classes.
    - Interestingly, performance was not consistently correlated with the amount of training data. The 70:30 split resulted in a catastrophic failure, with accuracy dropping to just 27.4% and an AUC of 0.15, which is worse than random guessing. This suggests that this particular stratified split may have resulted in a validation set that was not representative, causing the model to learn incorrect patterns.
    - The optimal hyperparameters were consistently found to be algorithm: 'viterbi' and covariance_type: 'diag' across the different splits.

Wisconsin Breast Cancer Dataset

The HMM models performed notably better on the Wisconsin Breast Cancer dataset, suggesting its features may have more discernible sequential patterns.

- Gaussian HMM: This model demonstrated strong performance.
    - Across various train-test splits, the model achieved high accuracy, peaking at 90.8% with a 60:40 split. The 80:20 and 50:50 splits also performed well, with accuracies of 90.4% and 88.8% respectively.

    - Similar to the Ionosphere results, the 70:30 split was a significant anomaly, with the model's accuracy plummeting to a mere 9.9%. This reinforces the idea that the 70:30 data partition was problematic for the model's learning process.
    - For this dataset, the best hyperparameter for covariance_type was consistently 'full', unlike the 'diag' for the Ionosphere dataset, highlighting how optimal model configurations can be data-dependent.
- Multinomial HMM: After binarizing the dataset by comparing features to their mean, the Multinomial HMM performed very well, achieving an accuracy of 90%. Unlike its application on the Ionosphere data, this model successfully identified both classes with high precision and recall, demonstrating that its effectiveness is highly contingent on the preprocessing and nature of the input data.

In summary, the Gaussian HMM proved to be a more robust model for these continuous datasets, although its performance was highly sensitive to the specific train-test data split, as seen in the recurring failure at the 70:30 ratio.

Question 2 & 3

DATASETS:

1. **CIFAR-10 Dataset**
    - Classes: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck
    - Total Samples: 60000 32x32 colour images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images.

2. **MNIST Dataset**
    - Classes: 10 classes, representing the digits from 0 to 9
    - Total Samples: Training set of 60,000 examples, and a test set of 10,000 examples

Convolutional Neural Network (CNN) for classification

Code:

```
import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers, models
from sklearn.metrics import confusion_matrix, classification_report,
roc_curve, auc
from sklearn.preprocessing import label_binarize
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
import seaborn as sns
from itertools import cycle
import pandas as pd
```

```
(x_train_mnist, y_train_mnist), (x_test_mnist, y_test_mnist) =
keras.datasets.mnist.load_data()
x_train_mnist = x_train_mnist.reshape(-1, 28, 28, 1).astype('float32')
/ 255.0
x_test_mnist = x_test_mnist.reshape(-1, 28, 28, 1).astype('float32') /
255.0

(x_train_cifar, y_train_cifar), (x_test_cifar, y_test_cifar) =
keras.datasets.cifar10.load_data()
x_train_cifar = x_train_cifar.astype('float32') / 255.0
x_test_cifar = x_test_cifar.astype('float32') / 255.0
y_train_cifar = y_train_cifar.flatten()
y_test_cifar = y_test_cifar.flatten()

def create_custom_cnn_mnist():
    model = models.Sequential([
        layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28,
28, 1)),
        layers.MaxPooling2D((2, 2)),
        layers.Conv2D(64, (3, 3), activation='relu'),
        layers.MaxPooling2D((2, 2)),
        layers.Conv2D(64, (3, 3), activation='relu'),
        layers.Flatten(),
        layers.Dense(64, activation='relu'),
        layers.Dropout(0.5),
        layers.Dense(10, activation='softmax')
    ])
    return model

def create_custom_cnn_cifar():
    model = models.Sequential([
        layers.Conv2D(32, (3, 3), activation='relu', padding='same',
input_shape=(32, 32, 3)),
        layers.BatchNormalization(),
        layers.Conv2D(32, (3, 3), activation='relu', padding='same'),
        layers.BatchNormalization(),
        layers.MaxPooling2D((2, 2)),
        layers.Dropout(0.2),

        layers.Conv2D(64, (3, 3), activation='relu', padding='same'),
        layers.BatchNormalization(),
        layers.Conv2D(64, (3, 3), activation='relu', padding='same'),
        layers.BatchNormalization(),
        layers.MaxPooling2D((2, 2)),
        layers.Dropout(0.3),

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

```
        layers.MaxPooling2D((2, 2)),
        layers.Dropout(0.4),

        layers.Flatten(),
        layers.Dense(128, activation='relu'),
        layers.BatchNormalization(),
        layers.Dropout(0.5),
        layers.Dense(10, activation='softmax')
    ])
    return model

split_ratios = [0.6, 0.7, 0.8, 0.9]
results_mnist = []
results_cifar = []
histories_mnist = {}
histories_cifar = {}
models_mnist = {}
models_cifar = {}

print("="*80)
print("TRAINING CUSTOM CNN WITH DIFFERENT TRAIN-TEST SPLITS")
print("="*80)

for split in split_ratios:
    print(f"\n{'='*80}")
    print(f"Training with {int(split*100)}% train -
{int((1-split)*100)}% test split")
    print(f"{'='*80}")

    train_size = int(len(x_train_mnist) * split)
    x_tr_mnist = x_train_mnist[:train_size]
    y_tr_mnist = y_train_mnist[:train_size]
    x_val_mnist = x_train_mnist[train_size:]
    y_val_mnist = y_train_mnist[train_size:]

    y_tr_mnist_cat = keras.utils.to_categorical(y_tr_mnist, 10)
    y_val_mnist_cat = keras.utils.to_categorical(y_val_mnist, 10)
    y_test_mnist_cat = keras.utils.to_categorical(y_test_mnist, 10)

    print(f"\nMNIST - Training samples: {len(x_tr_mnist)}, Validation
samples: {len(x_val_mnist)}")
    model_mnist = create_custom_cnn_mnist()
    model_mnist.compile(optimizer='adam',
loss='categorical_crossentropy', metrics=['accuracy'])
    history_mnist = model_mnist.fit(x_tr_mnist, y_tr_mnist_cat,
epochs=15, batch_size=128,
                                    validation_data=(x_val_mnist,
y_val_mnist_cat), verbose=0)
    test_loss, test_acc = model_mnist.evaluate(x_test_mnist,
y_test_mnist_cat, verbose=0)
```

```
    results_mnist.append({
        'split': f"{int(split*100)}-{int((1-split)*100)}",
        'train_acc': history_mnist.history['accuracy'][-1],
        'val_acc': history_mnist.history['val_accuracy'][-1],
        'test_acc': test_acc
    })
    histories_mnist[split] = history_mnist
    models_mnist[split] = model_mnist
    print(f"MNIST Test Accuracy: {test_acc:.4f}")

    train_size = int(len(x_train_cifar) * split)
    x_tr_cifar = x_train_cifar[:train_size]
    y_tr_cifar = y_train_cifar[:train_size]
    x_val_cifar = x_train_cifar[train_size:]
    y_val_cifar = y_train_cifar[train_size:]

    y_tr_cifar_cat = keras.utils.to_categorical(y_tr_cifar, 10)
    y_val_cifar_cat = keras.utils.to_categorical(y_val_cifar, 10)
    y_test_cifar_cat = keras.utils.to_categorical(y_test_cifar, 10)

    print(f"CIFAR-10 - Training samples: {len(x_tr_cifar)}, Validation
samples: {len(x_val_cifar)}")
    model_cifar = create_custom_cnn_cifar()
    model_cifar.compile(optimizer='adam',
loss='categorical_crossentropy', metrics=['accuracy'])
    history_cifar = model_cifar.fit(x_tr_cifar, y_tr_cifar_cat,
epochs=50, batch_size=128,
                                    validation_data=(x_val_cifar,
y_val_cifar_cat), verbose=0)
    test_loss, test_acc = model_cifar.evaluate(x_test_cifar,
y_test_cifar_cat, verbose=0)

    results_cifar.append({
        'split': f"{int(split*100)}-{int((1-split)*100)}",
        'train_acc': history_cifar.history['accuracy'][-1],
        'val_acc': history_cifar.history['val_accuracy'][-1],
        'test_acc': test_acc
    })
    histories_cifar[split] = history_cifar
    models_cifar[split] = model_cifar
    print(f"CIFAR-10 Test Accuracy: {test_acc:.4f}")

df_mnist = pd.DataFrame(results_mnist)
df_cifar = pd.DataFrame(results_cifar)

print("\n" + "="*80)
print("MNIST RESULTS - DIFFERENT TRAIN-TEST SPLITS")
print("="*80)
print(df_mnist.to_string(index=False))

print("\n" + "="*80)
```

```
print("CIFAR-10 RESULTS - DIFFERENT TRAIN-TEST SPLITS")
print("="*80)
print(df_cifar.to_string(index=False))

fig, axes = plt.subplots(2, 2, figsize=(16, 12))

splits_labels = [f"{int(s*100)}-{int((1-s)*100)}" for s in
split_ratios]
mnist_train_accs = [r['train_acc'] for r in results_mnist]
mnist_val_accs = [r['val_acc'] for r in results_mnist]
mnist_test_accs = [r['test_acc'] for r in results_mnist]

x_pos = np.arange(len(splits_labels))
width = 0.25

axes[0, 0].bar(x_pos - width, mnist_train_accs, width, label='Train',
alpha=0.8)
axes[0, 0].bar(x_pos, mnist_val_accs, width, label='Validation',
alpha=0.8)
axes[0, 0].bar(x_pos + width, mnist_test_accs, width, label='Test',
alpha=0.8)
axes[0, 0].set_xlabel('Train-Test Split')
axes[0, 0].set_ylabel('Accuracy')
axes[0, 0].set_title('MNIST - Accuracy vs Train-Test Split')
axes[0, 0].set_xticks(x_pos)
axes[0, 0].set_xticklabels(splits_labels)
axes[0, 0].legend()
axes[0, 0].grid(True, alpha=0.3)

cifar_train_accs = [r['train_acc'] for r in results_cifar]
cifar_val_accs = [r['val_acc'] for r in results_cifar]
cifar_test_accs = [r['test_acc'] for r in results_cifar]

axes[0, 1].bar(x_pos - width, cifar_train_accs, width, label='Train',
alpha=0.8)
axes[0, 1].bar(x_pos, cifar_val_accs, width, label='Validation',
alpha=0.8)
axes[0, 1].bar(x_pos + width, cifar_test_accs, width, label='Test',
alpha=0.8)
axes[0, 1].set_xlabel('Train-Test Split')
axes[0, 1].set_ylabel('Accuracy')
axes[0, 1].set_title('CIFAR-10 - Accuracy vs Train-Test Split')
axes[0, 1].set_xticks(x_pos)
axes[0, 1].set_xticklabels(splits_labels)
axes[0, 1].legend()
axes[0, 1].grid(True, alpha=0.3)

axes[1, 0].plot(splits_labels, mnist_test_accs, marker='o',
linewidth=2, markersize=8)
axes[1, 0].set_xlabel('Train-Test Split')
axes[1, 0].set_ylabel('Test Accuracy')
```

```
axes[1, 0].set_title('MNIST - Test Accuracy Trend')
axes[1, 0].grid(True, alpha=0.3)
axes[1, 0].set_ylim([min(mnist_test_accs)-0.01,
max(mnist_test_accs)+0.01])

axes[1, 1].plot(splits_labels, cifar_test_accs, marker='o',
linewidth=2, markersize=8, color='orange')
axes[1, 1].set_xlabel('Train-Test Split')
axes[1, 1].set_ylabel('Test Accuracy')
axes[1, 1].set_title('CIFAR-10 - Test Accuracy Trend')
axes[1, 1].grid(True, alpha=0.3)
axes[1, 1].set_ylim([min(cifar_test_accs)-0.02,
max(cifar_test_accs)+0.02])

plt.tight_layout()
plt.savefig('cnn_split_comparison.png', dpi=300, bbox_inches='tight')
plt.show()

best_split_mnist = split_ratios[np.argmax([r['test_acc'] for r in
results_mnist])]
best_split_cifar = split_ratios[np.argmax([r['test_acc'] for r in
results_cifar])]

print(f"\nBest split for MNIST:
{int(best_split_mnist*100)}-{int((1-best_split_mnist)*100)}")
print(f"Best split for CIFAR-10:
{int(best_split_cifar*100)}-{int((1-best_split_cifar)*100)}")

best_model_mnist = models_mnist[best_split_mnist]
best_model_cifar = models_cifar[best_split_cifar]
best_history_mnist = histories_mnist[best_split_mnist]
best_history_cifar = histories_cifar[best_split_cifar]

y_test_mnist_cat = keras.utils.to_categorical(y_test_mnist, 10)
y_test_cifar_cat = keras.utils.to_categorical(y_test_cifar, 10)

y_pred_mnist = best_model_mnist.predict(x_test_mnist, verbose=0)
y_pred_mnist_classes = np.argmax(y_pred_mnist, axis=1)

y_pred_cifar = best_model_cifar.predict(x_test_cifar, verbose=0)
y_pred_cifar_classes = np.argmax(y_pred_cifar, axis=1)

cm_mnist = confusion_matrix(y_test_mnist, y_pred_mnist_classes)
cm_cifar = confusion_matrix(y_test_cifar, y_pred_cifar_classes)

fig, axes = plt.subplots(1, 2, figsize=(18, 7))
sns.heatmap(cm_mnist, annot=True, fmt='d', cmap='Blues', ax=axes[0],
cbar_kws={'label': 'Count'})
axes[0].set_title(f'Custom CNN - MNIST Confusion Matrix\n(Best Split:
{int(best_split_mnist*100)}-{int((1-best_split_mnist)*100)})',
fontsize=14, fontweight='bold')
```

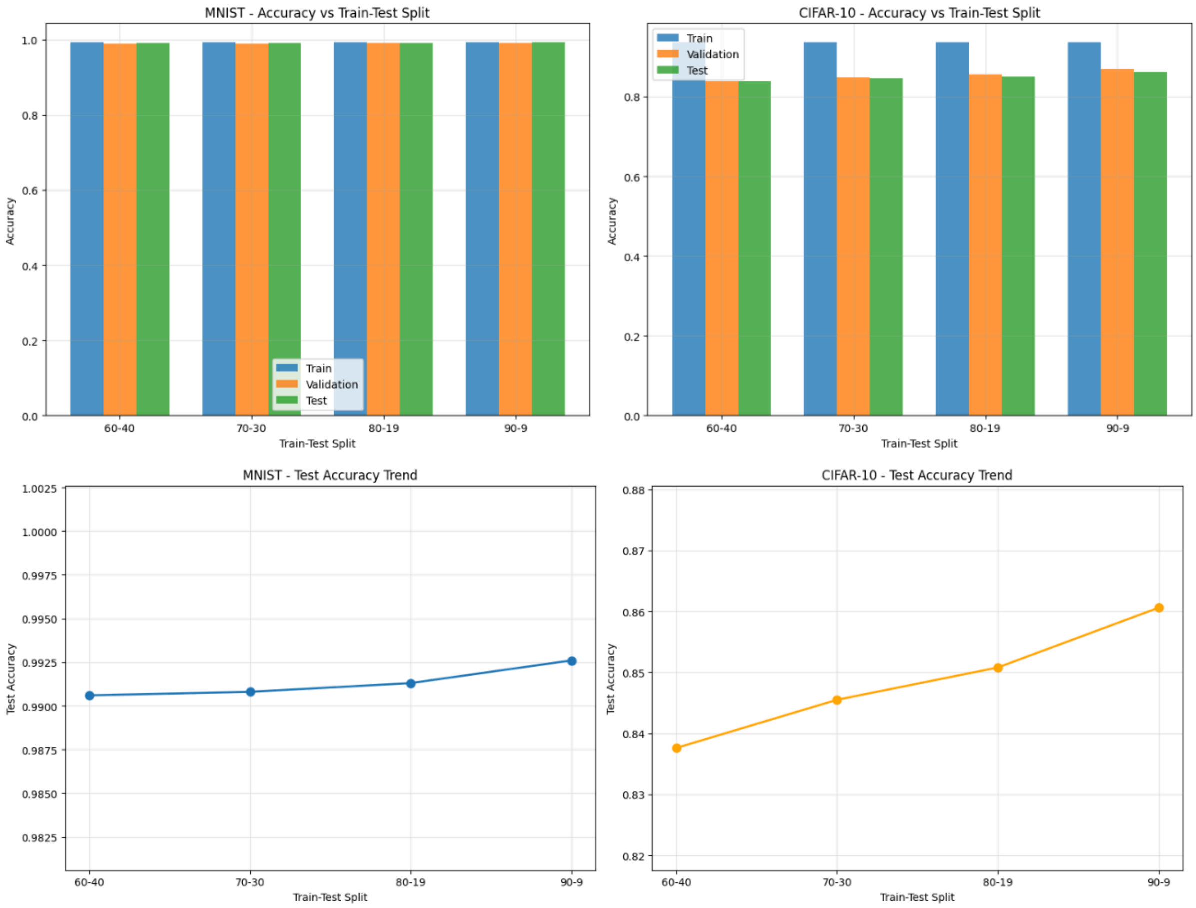
```
axes[0].set_ylabel('True Label', fontsize=12)
axes[0].set_xlabel('Predicted Label', fontsize=12)

sns.heatmap(cm_cifar, annot=True, fmt='d', cmap='Greens', ax=axes[1],
cbar_kws={'label': 'Count'})
axes[1].set_title(f'Custom CNN - CIFAR-10 Confusion Matrix\n(Best
Split: {int(best_split_cifar*100)}-{int((1-best_split_cifar)*100)})',
fontsize=14, fontweight='bold')
axes[1].set_ylabel('True Label', fontsize=12)
axes[1].set_xlabel('Predicted Label', fontsize=12)

plt.tight_layout()
plt.savefig('cnn_confusion_matrices_best.png', dpi=300,
bbox_inches='tight')
plt.show()

fig, axes = plt.subplots(2, 2, figsize=(16, 12))

axes[0, 0].plot(best_history_mnist.history['accuracy'], label='Train',
linewidth=2)
axes[0, 0].plot(best_history_mnist.history['val_accuracy'],
label='Validation', linewidth=2)
axes[0, 0].set_title(f'Custom CNN - MNIST Accuracy\n(Split:
{int(best_split_mnist*100)}-{int((1-best_split_mnist)*100)})',
fontsize=12, fontweight='bold')
axes[0, 0].set_xlabel('Epoch', fontsize=11)
axes[0, 0].set_ylabel('Accuracy', fontsize=11)
axes[0, 0].legend(fontsize=10)
axes[0, 0].grid(True, alpha=0.3)

axes[0, 1].plot(best_history_mnist.history['loss'], label='Train',
linewidth=2)
axes[0, 1].plot(best_history_mnist.history['val_loss'],
label='Validation', linewidth=2)
axes[0, 1].set_title(f'Custom CNN - MNIST Loss\n(Split:
{int(best_split_mnist*100)}-{int((1-best_split_mnist)*100)})',
fontsize=12, fontweight='bold')
axes[0, 1].set_xlabel('Epoch', fontsize=11)
axes[0, 1].set_ylabel('Loss', fontsize=11)
axes[0, 1].legend(fontsize=10)
axes[0, 1].grid(True, alpha=0.3)

axes[1, 0].plot(best_history_cifar.history['accuracy'], label='Train',
linewidth=2)
axes[1, 0].plot(best_history_cifar.history['val_accuracy'],
label='Validation', linewidth=2)
axes[1, 0].set_title(f'Custom CNN - CIFAR-10 Accuracy\n(Split:
{int(best_split_cifar*100)}-{int((1-best_split_cifar)*100)})',
fontsize=12, fontweight='bold')
axes[1, 0].set_xlabel('Epoch', fontsize=11)
axes[1, 0].set_ylabel('Accuracy', fontsize=11)
```

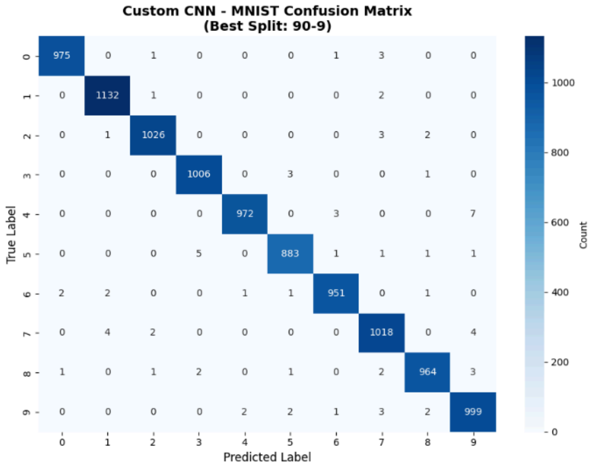
```
axes[1, 0].legend(fontsize=10)
axes[1, 0].grid(True, alpha=0.3)

axes[1, 1].plot(best_history_cifar.history['loss'], label='Train',
linewidth=2)
axes[1, 1].plot(best_history_cifar.history['val_loss'],
label='Validation', linewidth=2)
axes[1, 1].set_title(f'Custom CNN - CIFAR-10 Loss\n(Split:
{int(best_split_cifar*100)}-{int((1-best_split_cifar)*100)})',
fontsize=12, fontweight='bold')
axes[1, 1].set_xlabel('Epoch', fontsize=11)
axes[1, 1].set_ylabel('Loss', fontsize=11)
axes[1, 1].legend(fontsize=10)
axes[1, 1].grid(True, alpha=0.3)

plt.tight_layout()
plt.savefig('cnn_training_loss_curves_best.png', dpi=300,
bbox_inches='tight')
plt.show()

y_test_mnist_bin = label_binarize(y_test_mnist, classes=range(10))
y_test_cifar_bin = label_binarize(y_test_cifar, classes=range(10))

fpr_mnist = {}
tpr_mnist = {}
roc_auc_mnist = {}
for i in range(10):
    fpr_mnist[i], tpr_mnist[i], _ = roc_curve(y_test_mnist_bin[:, i],
y_pred_mnist[:, i])
    roc_auc_mnist[i] = auc(fpr_mnist[i], tpr_mnist[i])

fpr_cifar = {}
tpr_cifar = {}
roc_auc_cifar = {}
for i in range(10):
    fpr_cifar[i], tpr_cifar[i], _ = roc_curve(y_test_cifar_bin[:, i],
y_pred_cifar[:, i])
    roc_auc_cifar[i] = auc(fpr_cifar[i], tpr_cifar[i])

fig, axes = plt.subplots(1, 2, figsize=(18, 7))

colors = cycle(['blue', 'red', 'green', 'orange', 'purple', 'brown',
'pink', 'gray', 'olive', 'cyan'])
for i, color in zip(range(10), colors):
    axes[0].plot(fpr_mnist[i], tpr_mnist[i], color=color, lw=2.5,
                 label=f'Class {i} (AUC={roc_auc_mnist[i]:.3f})')
axes[0].plot([0, 1], [0, 1], 'k--', lw=2, label='Random Classifier')
axes[0].set_xlim([0.0, 1.0])
axes[0].set_ylim([0.0, 1.05])
axes[0].set_xlabel('False Positive Rate', fontsize=12)
axes[0].set_ylabel('True Positive Rate', fontsize=12)
```

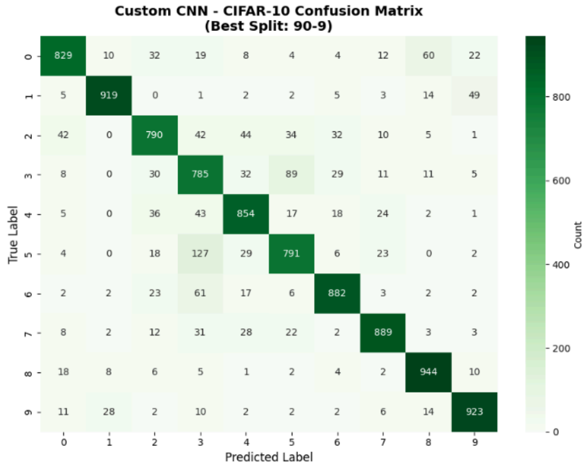
```
axes[0].set_title(f'Custom CNN - MNIST ROC Curves\n(Split:
{int(best_split_mnist*100)}-{int((1-best_split_mnist)*100)}, Avg
AUC={np.mean(list(roc_auc_mnist.values())):.3f})',
                  fontsize=12, fontweight='bold')
axes[0].legend(loc="lower right", fontsize=9)
axes[0].grid(True, alpha=0.3)

colors = cycle(['blue', 'red', 'green', 'orange', 'purple', 'brown',
'pink', 'gray', 'olive', 'cyan'])
for i, color in zip(range(10), colors):
    axes[1].plot(fpr_cifar[i], tpr_cifar[i], color=color, lw=2.5,
                 label=f'Class {i} (AUC={roc_auc_cifar[i]:.3f})')
axes[1].plot([0, 1], [0, 1], 'k--', lw=2, label='Random Classifier')
axes[1].set_xlim([0.0, 1.0])
axes[1].set_ylim([0.0, 1.05])
axes[1].set_xlabel('False Positive Rate', fontsize=12)
axes[1].set_ylabel('True Positive Rate', fontsize=12)
axes[1].set_title(f'Custom CNN - CIFAR-10 ROC Curves\n(Split:
{int(best_split_cifar*100)}-{int((1-best_split_cifar)*100)}, Avg
AUC={np.mean(list(roc_auc_cifar.values())):.3f})',
                  fontsize=12, fontweight='bold')
axes[1].legend(loc="lower right", fontsize=9)
axes[1].grid(True, alpha=0.3)

plt.tight_layout()
plt.savefig('cnn_roc_auc_curves_best.png', dpi=300,
bbox_inches='tight')
plt.show()

print("\n" + "="*80)
print("CUSTOM CNN - BEST CASE RESULTS SUMMARY")
print("="*80)
print(f"\nMNIST (Best Split:
{int(best_split_mnist*100)}-{int((1-best_split_mnist)*100)})")
print(f"  Test Accuracy:
{results_mnist[split_ratios.index(best_split_mnist)]['test_acc']:.4f}")
print(f"  Average AUC: {np.mean(list(roc_auc_mnist.values())):.4f}")
print(f"\nCIFAR-10 (Best Split:
{int(best_split_cifar*100)}-{int((1-best_split_cifar)*100)})")
print(f"  Test Accuracy:
{results_cifar[split_ratios.index(best_split_cifar)]['test_acc']:.4f}")
print(f"  Average AUC: {np.mean(list(roc_auc_cifar.values())):.4f}")
print("="*80)
```

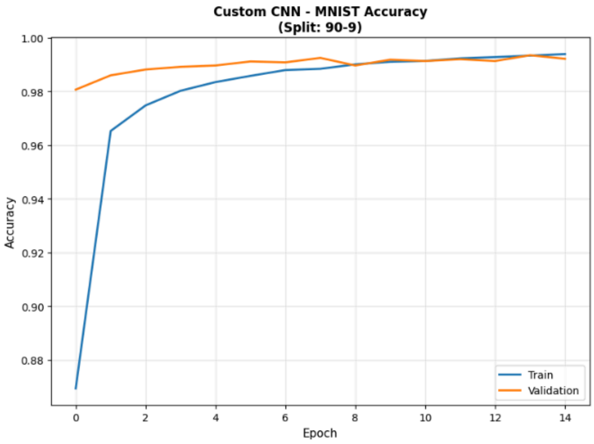
Results for CNN

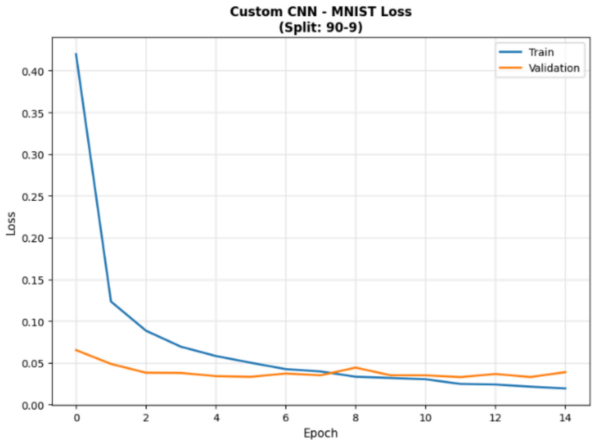
```
================================================================================
MNIST RESULTS - DIFFERENT TRAIN-TEST SPLITS
================================================================================
split  train_acc  val_acc  test_acc
60-40   0.992556 0.988833    0.9906
70-30   0.993095 0.989556    0.9908
80-19   0.993479 0.991917    0.9913
 90-9   0.993926 0.992167    0.9926

================================================================================
CIFAR-10 RESULTS - DIFFERENT TRAIN-TEST SPLITS
================================================================================
split  train_acc  val_acc  test_acc
60-40     0.9366  0.83875    0.8376
70-30     0.9358  0.84760    0.8455
80-19     0.9370  0.85600    0.8508
 90-9     0.9370  0.86940    0.8606
```

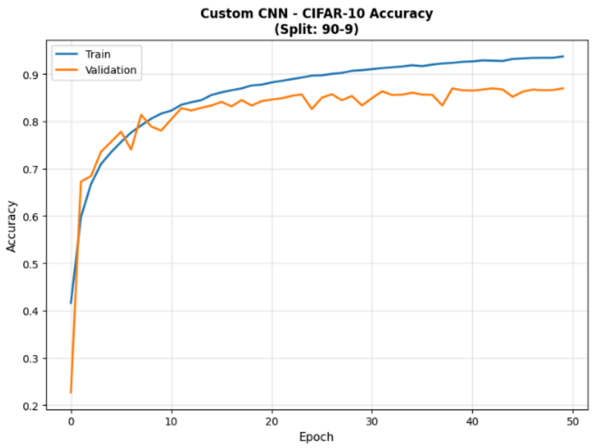


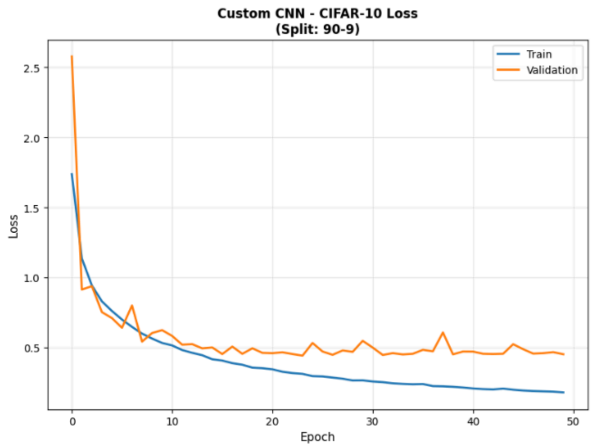

Custom CNN - MNIST Confusion Matrix
(Best Split: 90-9)
True Label
Predicted Label
Count

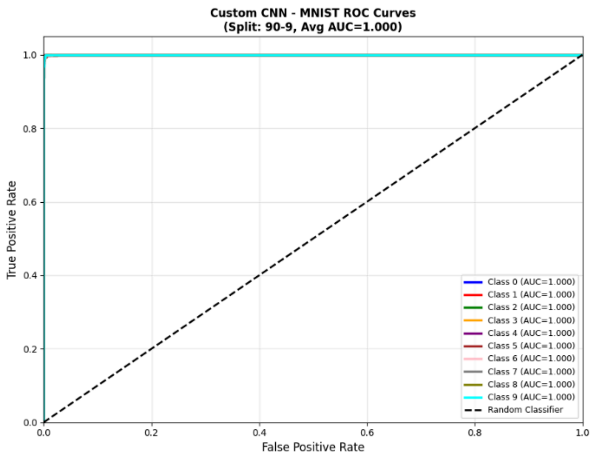

Custom CNN - CIFAR-10 Confusion Matrix
(Best Split: 90-9)
True Label
Predicted Label
Count

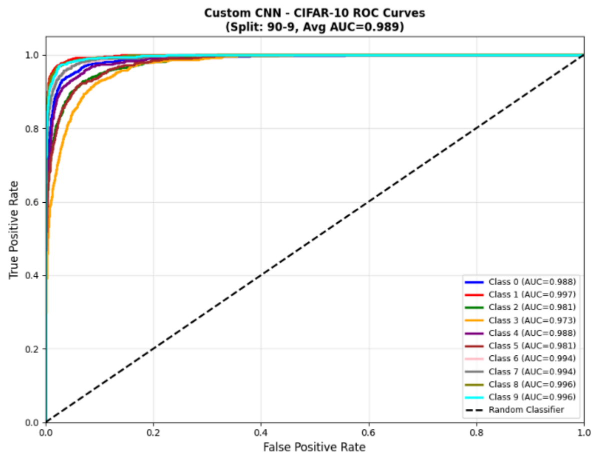

Custom CNN - MNIST Accuracy
(Split: 90-9)
Accuracy
Epoch
Train
Validation


Custom CNN - MNIST Loss
(Split: 90-9)
Loss
Epoch
Train
Validation


Custom CNN - CIFAR-10 Accuracy
(Split: 90-9)
Accuracy
Epoch
Train
Validation


Custom CNN - CIFAR-10 Loss
(Split: 90-9)
Loss
Epoch
Train
Validation






```
================================================================================
CUSTOM CNN - BEST CASE RESULTS SUMMARY
================================================================================

MNIST (Best Split: 90-9)
  Test Accuracy: 0.9926
  Average AUC: 0.9999

CIFAR-10 (Best Split: 90-9)
  Test Accuracy: 0.8606
  Average AUC: 0.9887
================================================================================
```

VGG16

Code:

```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
from torchvision import models
from torch.utils.data import DataLoader, Subset

import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import confusion_matrix, roc_curve, auc
from sklearn.preprocessing import label_binarize
from itertools import cycle
import time
import copy

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")

def get_datasets():
```

```
    # Resize
    transform_std = transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229,
0.224, 0.225]),
    ])

    transform_mnist = transforms.Compose([
        transforms.Resize(224),
        transforms.Grayscale(num_output_channels=3), # Convert to 3
channels
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5,
0.5]),
    ])

    cifar10_train = torchvision.datasets.CIFAR10(root='./data',
train=True, download=True, transform=transform_std)
    cifar10_test = torchvision.datasets.CIFAR10(root='./data',
train=False, download=True, transform=transform_std)

    mnist_train = torchvision.datasets.MNIST(root='./data', train=True,
download=True, transform=transform_mnist)
    mnist_test = torchvision.datasets.MNIST(root='./data', train=False,
download=True, transform=transform_mnist)

    datasets = {
        'CIFAR-10': {'train': cifar10_train, 'test': cifar10_test,
'classes': cifar10_train.classes},
        'MNIST': {'train': mnist_train, 'test': mnist_test, 'classes':
mnist_train.classes}
    }

    return datasets

def get_model(num_classes):

    model = models.vgg16(pretrained=True)

    for param in model.parameters():
        param.requires_grad = False

    model.classifier[0] = nn.Linear(512 * 7 * 7, 4096)
    model.classifier[3] = nn.Linear(4096, 1024)
    model.classifier[6] = nn.Linear(1024, num_classes)

    return model.to(device)

def plot_curves(history, dataset_name, model_name):
```

```
    """Plots training & validation accuracy and loss curves."""
    plt.figure(figsize=(12, 5))

    # Plot Accuracy
    plt.subplot(1, 2, 1)
    plt.plot(history['train_acc'], label='Train Accuracy')
    plt.plot(history['val_acc'], label='Validation Accuracy')
    plt.title(f'Accuracy Curves: {model_name} on {dataset_name}')
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy')
    plt.legend()

    # Plot Loss
    plt.subplot(1, 2, 2)
    plt.plot(history['train_loss'], label='Train Loss')
    plt.plot(history['val_loss'], label='Validation Loss')
    plt.title(f'Loss Curves: {model_name} on {dataset_name}')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.legend()

    plt.tight_layout()
    plt.show()

def plot_confusion_matrix(y_true, y_pred, classes, dataset_name,
model_name):
    """Generates and plots a confusion matrix heatmap."""
    cm = confusion_matrix(y_true, y_pred)
    plt.figure(figsize=(10, 8))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
xticklabels=classes, yticklabels=classes)
    plt.xlabel('Predicted Label')
    plt.ylabel('True Label')
    plt.title(f'Confusion Matrix: {model_name} on {dataset_name}')
    plt.show()

def plot_roc_auc(y_true, y_score, n_classes, classes, dataset_name,
model_name):
    """Plots ROC curves and calculates AUC for each class."""
    # Binarize the output
    y_true_bin = label_binarize(y_true, classes=list(range(n_classes)))

    # Compute ROC curve and ROC area for each class
    fpr = dict()
    tpr = dict()
    roc_auc = dict()
    for i in range(n_classes):
        fpr[i], tpr[i], _ = roc_curve(y_true_bin[:, i], y_score[:, i])
        roc_auc[i] = auc(fpr[i], tpr[i])

    # Plot all ROC curves
```

```
    plt.figure(figsize=(10, 8))
    colors = cycle(['aqua', 'darkorange', 'cornflowerblue', 'green',
'red', 'purple', 'brown', 'pink', 'gray', 'olive'])
    for i, color in zip(range(n_classes), colors):
        plt.plot(fpr[i], tpr[i], color=color, lw=2,
                 label=f'ROC curve of class {classes[i]} (area =
{roc_auc[i]:0.2f})')

    plt.plot([0, 1], [0, 1], 'k--', lw=2)
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title(f'Receiver Operating Characteristic (ROC): {model_name}
on {dataset_name}')
    plt.legend(loc="lower right")
    plt.show()

def train_and_evaluate(model, train_loader, val_loader, criterion,
optimizer, num_epochs=5):

    history = {'train_loss': [], 'train_acc': [], 'val_loss': [],
'val_acc': []}
    best_model_wts = copy.deepcopy(model.state_dict())
    best_acc = 0.0

    for epoch in range(num_epochs):
        print(f'Epoch {epoch+1}/{num_epochs}')
        print('-' * 10)

        for phase in ['train', 'val']:
            if phase == 'train':
                model.train()
                dataloader = train_loader
            else:
                model.eval()
                dataloader = val_loader

            running_loss = 0.0
            running_corrects = 0

            for inputs, labels in dataloader:
                inputs, labels = inputs.to(device), labels.to(device)

                optimizer.zero_grad()

                with torch.set_grad_enabled(phase == 'train'):
                    outputs = model(inputs)
                    _, preds = torch.max(outputs, 1)
                    loss = criterion(outputs, labels)
```

```
                    if phase == 'train':
                        loss.backward()
                        optimizer.step()

                running_loss += loss.item() * inputs.size(0)
                running_corrects += torch.sum(preds == labels.data)

            epoch_loss = running_loss / len(dataloader.dataset)
            epoch_acc = running_corrects.double() /
len(dataloader.dataset)

            print(f'{phase} Loss: {epoch_loss:.4f} Acc:
{epoch_acc:.4f}')

            if phase == 'train':
                history['train_loss'].append(epoch_loss)
                history['train_acc'].append(epoch_acc.item())
            else:
                history['val_loss'].append(epoch_loss)
                history['val_acc'].append(epoch_acc.item())

            if phase == 'val' and epoch_acc > best_acc:
                best_acc = epoch_acc
                best_model_wts = copy.deepcopy(model.state_dict())

    print(f'Best val Acc: {best_acc:4f}')
    model.load_state_dict(best_model_wts)
    return model, history

def get_predictions(model, dataloader):
    model.eval()
    all_preds = torch.tensor([]).to(device)
    all_labels = torch.tensor([]).to(device)
    all_scores = torch.tensor([]).to(device)

    with torch.no_grad():
        for inputs, labels in dataloader:
            inputs, labels = inputs.to(device), labels.to(device)
            outputs = model(inputs)
            scores = torch.softmax(outputs, dim=1)
            _, preds = torch.max(outputs, 1)

            all_preds = torch.cat((all_preds, preds), dim=0)
            all_labels = torch.cat((all_labels, labels), dim=0)
            all_scores = torch.cat((all_scores, scores), dim=0)

    return all_labels.cpu().numpy(), all_preds.cpu().numpy(),
all_scores.cpu().numpy()

def main():
    datasets = get_datasets()
```

```
    model_names = ['VGG16']
    split_sizes = [0.6, 0.7, 0.8]
    num_epochs = 10

    for dataset_name, dataset_info in datasets.items():
        print(f"\n{'='*20} DATASET: {dataset_name} {'='*20}")
        train_dataset = dataset_info['train']
        test_dataset = dataset_info['test']
        classes = dataset_info['classes']
        num_classes = len(classes)

        test_loader = DataLoader(test_dataset, batch_size=32,
shuffle=False)

        for model_name in model_names:
            print(f"\n{'--'*10} MODEL: {model_name} {'--'*10}")

            overall_best_accuracy = 0.0
            best_case_results = {}
            best_split_size = 0

            for split in split_sizes:
                print(f"\n--- Training with {int(split*100)}% of data
---")

                num_train = len(train_dataset)
                indices = list(range(num_train))
                np.random.shuffle(indices)
                split_idx = int(np.floor(split * num_train))
                train_idx = indices[:split_idx]
                train_subset = Subset(train_dataset, train_idx)

                num_subset_train = len(train_subset)
                val_split = 0.2
                val_idx_end = int(np.floor(val_split *
num_subset_train))
                val_indices = list(range(num_subset_train))
                np.random.shuffle(val_indices)

                final_train_indices = val_indices[val_idx_end:]
                val_indices = val_indices[:val_idx_end]

                final_train_subset = Subset(train_subset,
final_train_indices)
                val_subset = Subset(train_subset, val_indices)

                train_loader = DataLoader(final_train_subset,
batch_size=32, shuffle=True)
                val_loader = DataLoader(val_subset, batch_size=32,
shuffle=False)
```

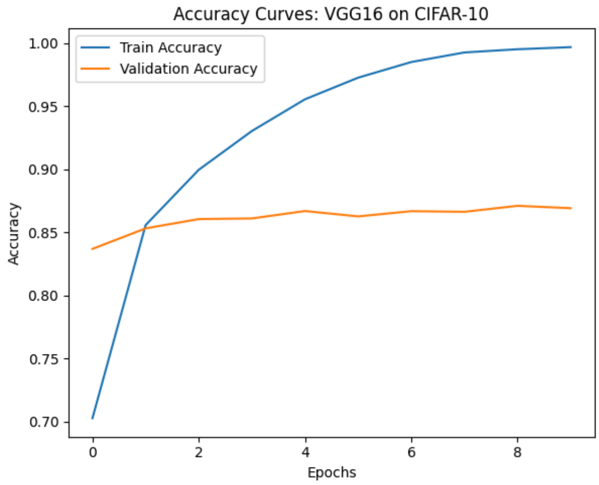
```
                # Train
                model = get_model(num_classes)
                criterion = nn.CrossEntropyLoss()
                params_to_update = [param for param in
model.parameters() if param.requires_grad]
                optimizer = optim.SGD(params_to_update, lr=0.001,
momentum=0.9)

                trained_model, history = train_and_evaluate(model,
train_loader, val_loader, criterion, optimizer, num_epochs)

                # Evaluate
                y_true, y_pred, _ = get_predictions(trained_model,
test_loader)
                test_accuracy = np.mean(y_true == y_pred)
                print(f"Final Test Accuracy for {int(split*100)}%
split: {test_accuracy:.4f}")

                if test_accuracy > overall_best_accuracy:
                    print(f"*** New best model found with accuracy:
{test_accuracy:.4f} ***")
                    overall_best_accuracy = test_accuracy
                    best_split_size = split
                    best_case_results['model'] = trained_model
                    best_case_results['history'] = history

            if 'model' in best_case_results:
                print(f"\n--- Generating plots for best case (from
{int(best_split_size*100)}% split with {overall_best_accuracy:.4f}
accuracy) ---")
                best_model = best_case_results['model']
                best_history = best_case_results['history']

                plot_curves(best_history, dataset_name, model_name)

                y_true, y_pred, y_score = get_predictions(best_model,
test_loader)
                plot_confusion_matrix(y_true, y_pred, classes,
dataset_name, model_name)
                plot_roc_auc(y_true, y_score, num_classes, classes,
dataset_name, model_name)

if __name__ == '__main__':
    main()
```

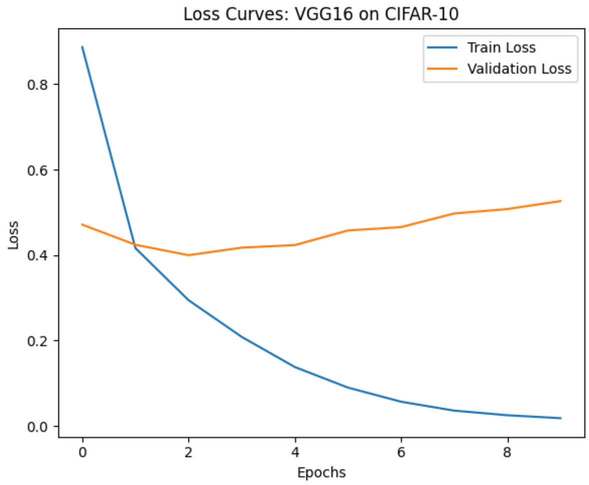
Results for VGG16:

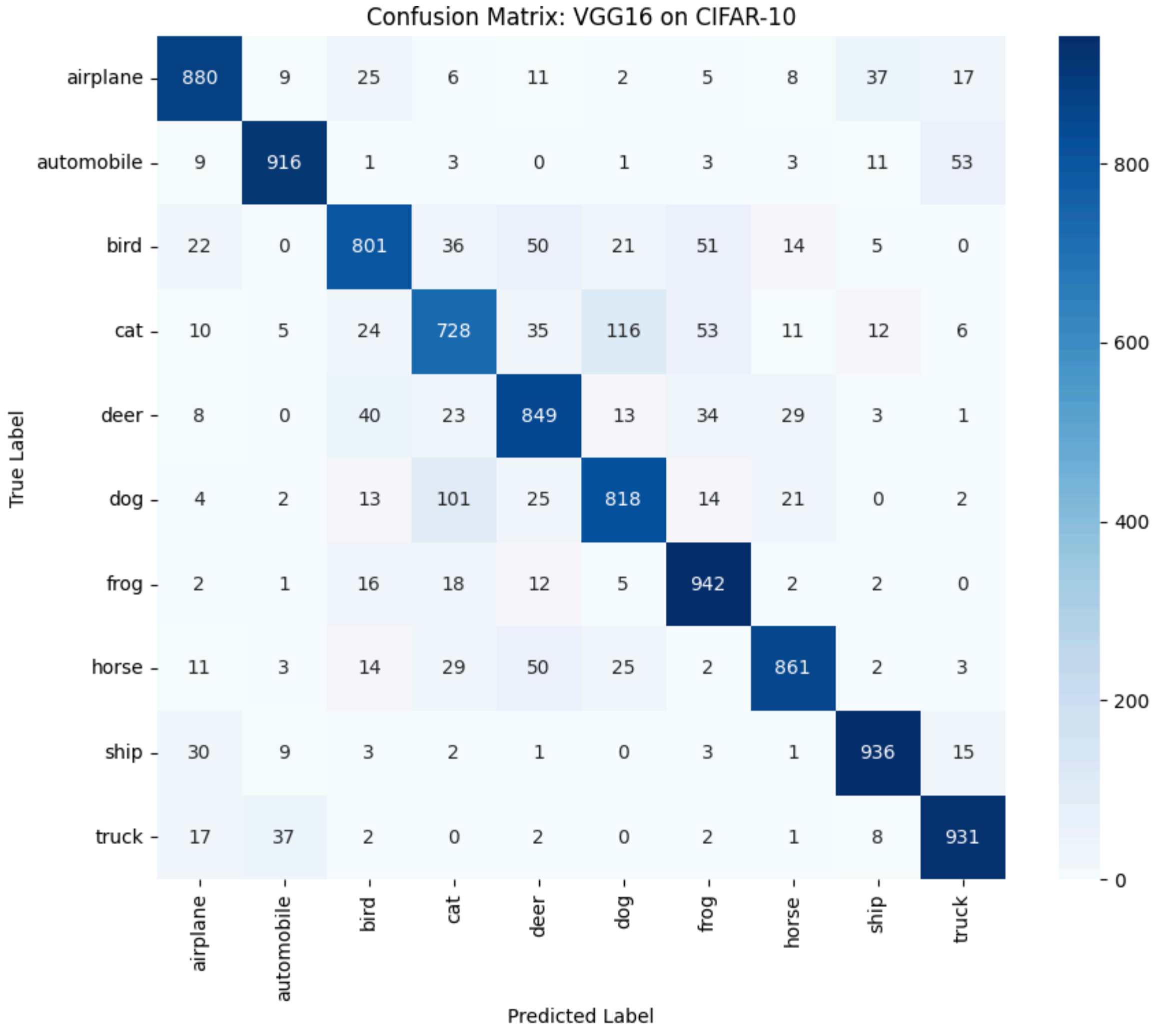
CIFAR-10 Dataset

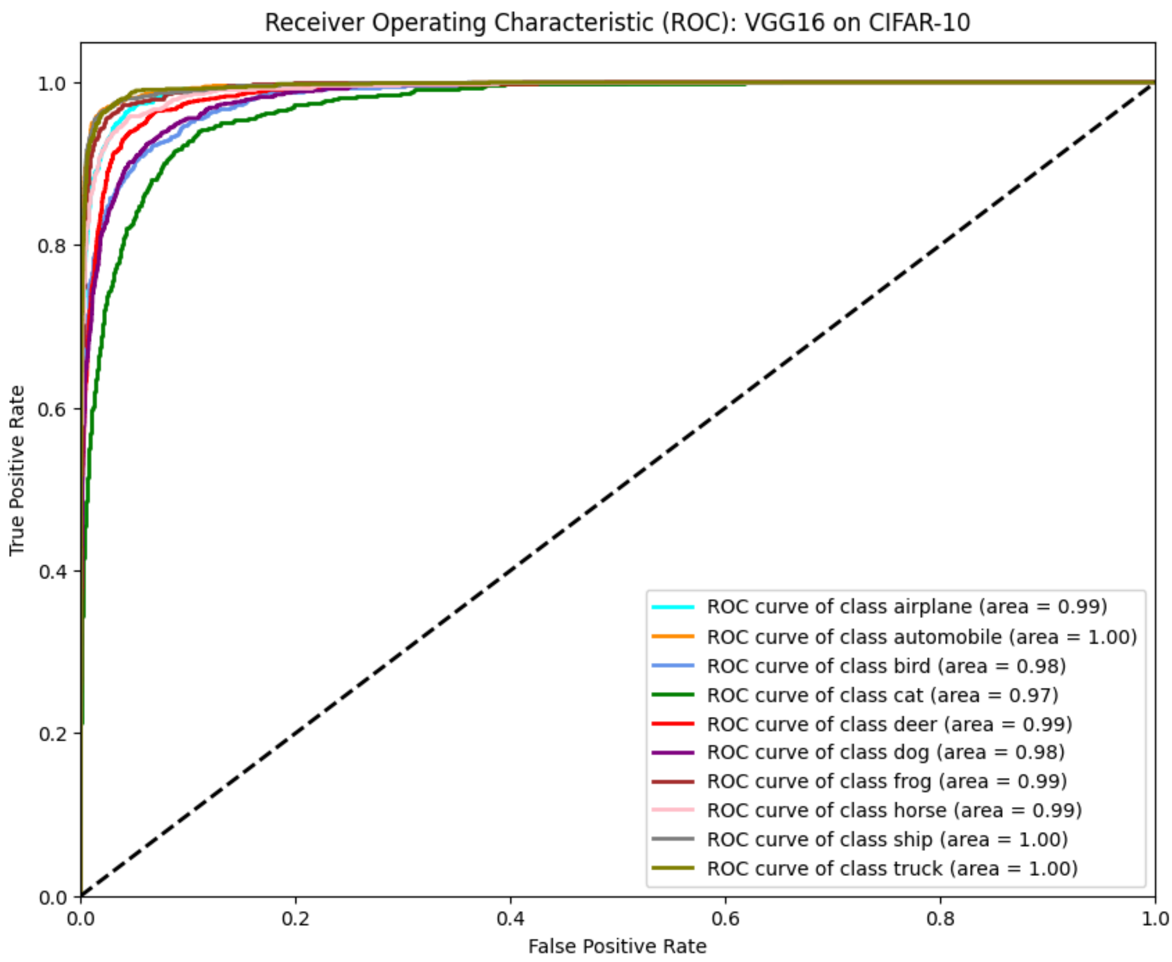
Split Size (Train-Test)	Accuracy
60%-40%	0.8376
70%-30%	0.8455
80%-20%	0.8508
90%-10%	0.8606

Plots for best split model:





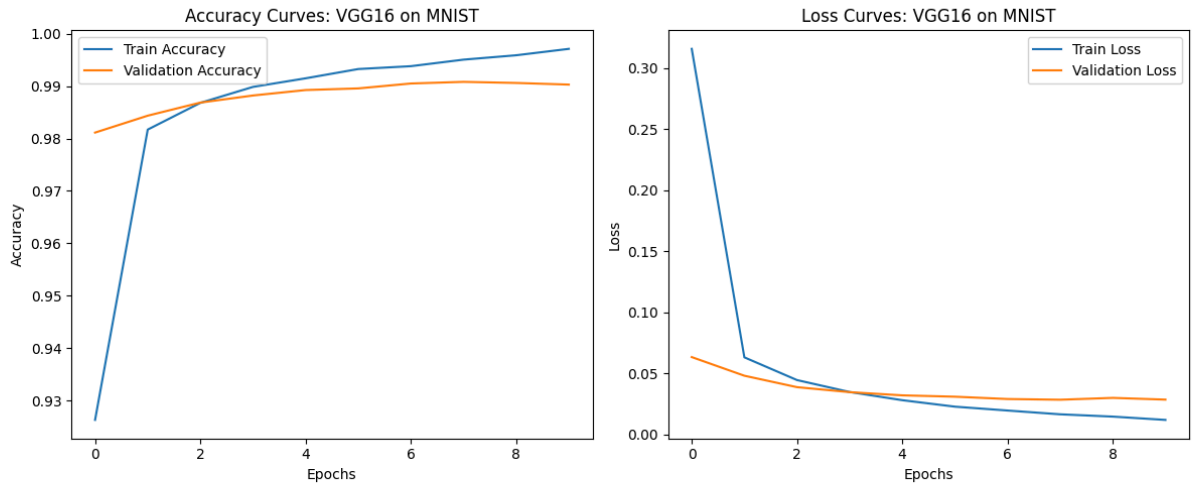

Confusion Matrix: VGG16 on CIFAR-10
True Label
Predicted Label
airplane
automobile
bird
cat
deer
dog
frog
horse
ship
truck
880 9 25 6 11 2 5 8 37 17
9 916 1 3 0 1 3 3 11 53
22 0 801 36 50 21 51 14 5 0
10 5 24 728 35 116 53 11 12 6
8 0 40 23 849 13 34 29 3 1
4 2 13 101 25 818 14 21 0 2
2 1 16 18 12 5 942 2 2 0
11 3 14 29 50 25 2 861 2 3
30 9 3 2 1 0 3 1 936 15
17 37 2 0 2 0 2 1 8 931
800
600
400
200
0

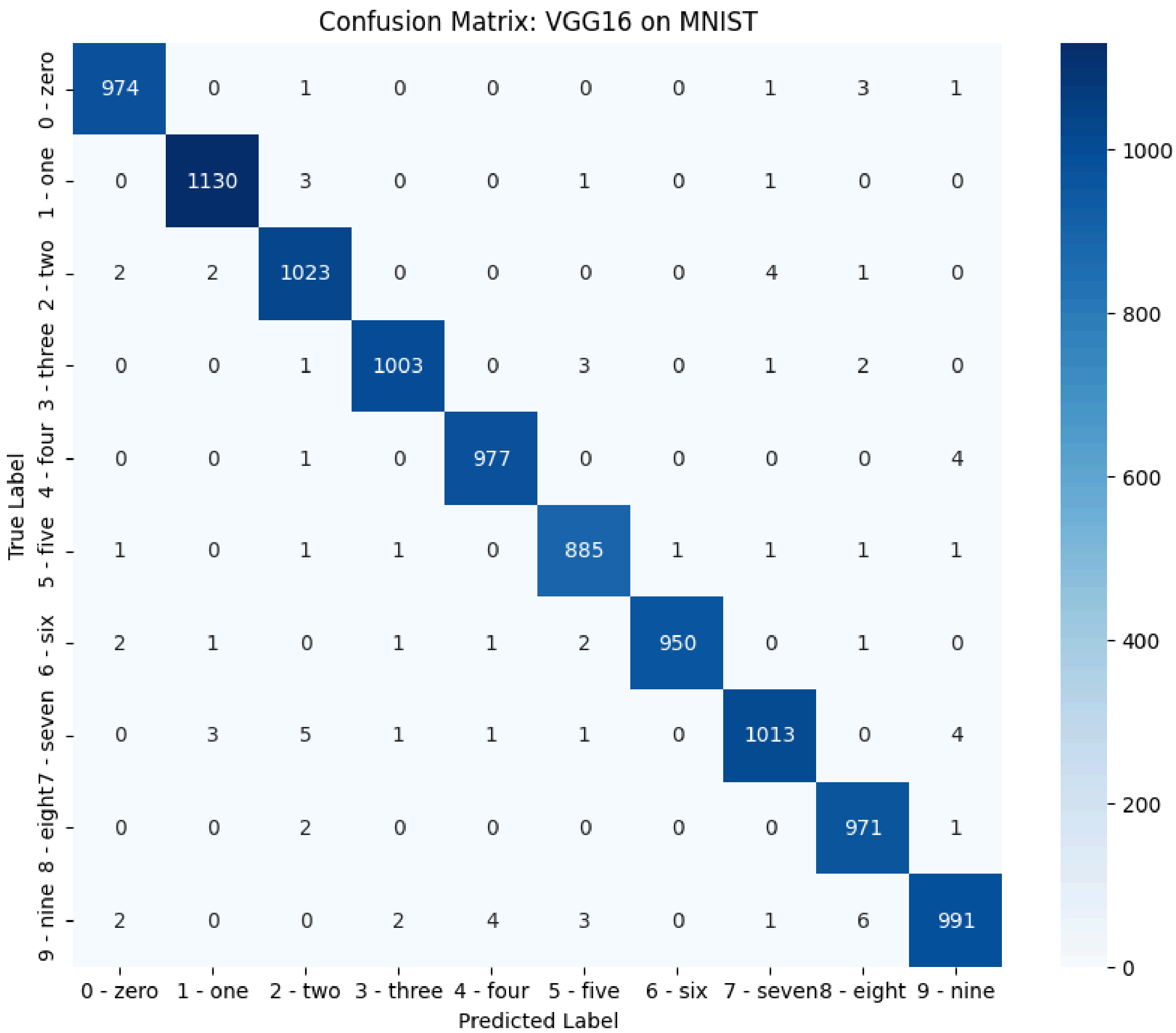


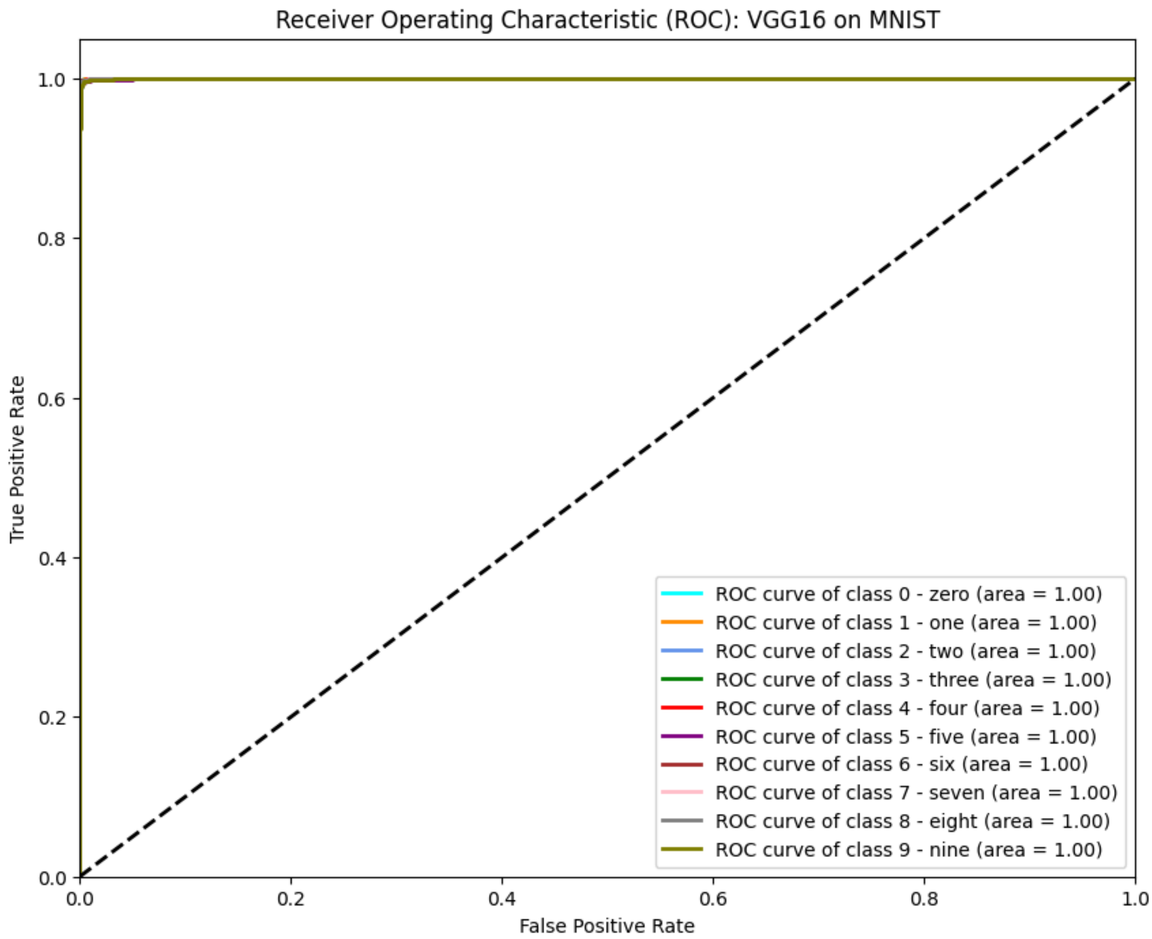
MNIST Dataset

Split Size (Train-Test)	Accuracy
60%-40%	0.9906
70%-30%	0.9908
80%-20%	0.9913
90%-10%	0.9926

Plots for best split model:


Accuracy Curves: VGG16 on MNIST
Train Accuracy
Validation Accuracy
Accuracy
Epochs
Loss Curves: VGG16 on MNIST
Train Loss
Validation Loss
Loss
Epochs


Confusion Matrix: VGG16 on MNIST
True Label
Predicted Label
0 - zero
1 - one
2 - two
3 - three
4 - four
5 - five
6 - six
7 - seven
8 - eight
9 - nine



Recurrent Neural Networks

Code:

```
import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers, models
from sklearn.metrics import confusion_matrix, classification_report,
roc_curve, auc
from sklearn.preprocessing import label_binarize
import matplotlib.pyplot as plt
import seaborn as sns
from itertools import cycle
import pandas as pd

# MNIST Dataset
(x_train_mnist, y_train_mnist), (x_test_mnist, y_test_mnist) =
keras.datasets.mnist.load_data()
x_train_mnist = x_train_mnist.astype('float32') / 255.0
x_test_mnist = x_test_mnist.astype('float32') / 255.0

# CIFAR-10 Dataset
```

```
(x_train_cifar, y_train_cifar), (x_test_cifar, y_test_cifar) =
keras.datasets.cifar10.load_data()
x_train_cifar = x_train_cifar.astype('float32') / 255.0
x_test_cifar = x_test_cifar.astype('float32') / 255.0
y_train_cifar = y_train_cifar.flatten()
y_test_cifar = y_test_cifar.flatten()

def create_rnn_model_mnist():
    model = models.Sequential([
        layers.LSTM(128, input_shape=(28, 28)),
        layers.Dropout(0.3),
        layers.Dense(64, activation='relu'),
        layers.Dropout(0.5),
        layers.Dense(10, activation='softmax')
    ])
    return model

def create_rnn_model_cifar():
    model = models.Sequential([
        layers.LSTM(128, return_sequences=True, input_shape=(32, 96)),
        layers.LSTM(128),
        layers.Dropout(0.4),
        layers.Dense(128, activation='relu'),
        layers.BatchNormalization(),
        layers.Dropout(0.5),
        layers.Dense(10, activation='softmax')
    ])
    return model

split_ratios = [0.6, 0.7, 0.8]
results_mnist = []
results_cifar = []
histories_mnist = {}
histories_cifar = {}
models_mnist = {}
models_cifar = {}

print("="*80)
print("TRAINING RNN MODELS WITH DIFFERENT TRAIN-TEST SPLITS")
print("="*80)

for split in split_ratios:
    print(f"\n{'='*80}")
    print(f"Training with {int(split*100)}% train -
{int((1-split)*100)}% validation split")
    print(f"{'='*80}")

    # MNIST RNN Training
    train_size_mnist = int(len(x_train_mnist) * split)
    x_tr_mnist, y_tr_mnist = x_train_mnist[:train_size_mnist],
y_train_mnist[:train_size_mnist]
```

```
    x_val_mnist, y_val_mnist = x_train_mnist[train_size_mnist:],
y_train_mnist[train_size_mnist:]

    y_tr_mnist_cat = keras.utils.to_categorical(y_tr_mnist, 10)
    y_val_mnist_cat = keras.utils.to_categorical(y_val_mnist, 10)
    y_test_mnist_cat = keras.utils.to_categorical(y_test_mnist, 10)

    print(f"\nMNIST - Training samples: {len(x_tr_mnist)}, Validation
samples: {len(x_val_mnist)}")
    model_mnist = create_rnn_model_mnist()
    model_mnist.compile(optimizer='adam',
loss='categorical_crossentropy', metrics=['accuracy'])
    history_mnist = model_mnist.fit(x_tr_mnist, y_tr_mnist_cat,
epochs=15, batch_size=128,
                                    validation_data=(x_val_mnist,
y_val_mnist_cat), verbose=0)
    test_loss_mnist, test_acc_mnist =
model_mnist.evaluate(x_test_mnist, y_test_mnist_cat, verbose=0)

    results_mnist.append({
        'split': f"{int(split*100)}-{int((1-split)*100)}",
        'train_acc': history_mnist.history['accuracy'][-1],
        'val_acc': history_mnist.history['val_accuracy'][-1],
        'test_acc': test_acc_mnist
    })
    histories_mnist[split] = history_mnist
    models_mnist[split] = model_mnist
    print(f"MNIST Test Accuracy: {test_acc_mnist:.4f}")

    # CIFAR RNN Training
    # Reshape
    x_train_cifar_rnn = x_train_cifar.reshape(-1, 32, 32 * 3)
    x_test_cifar_rnn = x_test_cifar.reshape(-1, 32, 32 * 3)

    train_size_cifar = int(len(x_train_cifar_rnn) * split)
    x_tr_cifar, y_tr_cifar = x_train_cifar_rnn[:train_size_cifar],
y_train_cifar[:train_size_cifar]
    x_val_cifar, y_val_cifar = x_train_cifar_rnn[train_size_cifar:],
y_train_cifar[train_size_cifar:]

    y_tr_cifar_cat = keras.utils.to_categorical(y_tr_cifar, 10)
    y_val_cifar_cat = keras.utils.to_categorical(y_val_cifar, 10)
    y_test_cifar_cat = keras.utils.to_categorical(y_test_cifar, 10)

    print(f"\nCIFAR-10 - Training samples: {len(x_tr_cifar)},
Validation samples: {len(x_val_cifar)}")
    model_cifar = create_rnn_model_cifar()
    model_cifar.compile(optimizer='adam',
loss='categorical_crossentropy', metrics=['accuracy'])
    history_cifar = model_cifar.fit(x_tr_cifar, y_tr_cifar_cat,
epochs=50, batch_size=128,
```

```
                                    validation_data=(x_val_cifar,
y_val_cifar_cat), verbose=0)
    test_loss_cifar, test_acc_cifar =
model_cifar.evaluate(x_test_cifar_rnn, y_test_cifar_cat, verbose=0)

    results_cifar.append({
        'split': f"{int(split*100)}-{int((1-split)*100)}",
        'train_acc': history_cifar.history['accuracy'][-1],
        'val_acc': history_cifar.history['val_accuracy'][-1],
        'test_acc': test_acc_cifar
    })
    histories_cifar[split] = history_cifar
    models_cifar[split] = model_cifar
    print(f"CIFAR-10 Test Accuracy: {test_acc_cifar:.4f}")

df_mnist = pd.DataFrame(results_mnist)
df_cifar = pd.DataFrame(results_cifar)

print("\n" + "="*80)
print("RNN MNIST RESULTS - DIFFERENT TRAIN-TEST SPLITS")
print("="*80)
print(df_mnist.to_string(index=False))

print("\n" + "="*80)
print("RNN CIFAR-10 RESULTS - DIFFERENT TRAIN-TEST SPLITS")
print("="*80)
print(df_cifar.to_string(index=False))

# Accuracy Comparison Bar Charts
fig, axes = plt.subplots(1, 2, figsize=(16, 6))
splits_labels = [f"{int(s*100)}-{int((1-s)*100)}" for s in
split_ratios]
mnist_test_accs = [r['test_acc'] for r in results_mnist]
cifar_test_accs = [r['test_acc'] for r in results_cifar]
x_pos = np.arange(len(splits_labels))

axes[0].bar(x_pos, mnist_test_accs, width=0.5, alpha=0.8)
axes[0].set_xlabel('Train-Validation Split')
axes[0].set_ylabel('Test Accuracy')
axes[0].set_title('RNN MNIST - Test Accuracy vs. Split')
axes[0].set_xticks(x_pos)
axes[0].set_xticklabels(splits_labels)
axes[0].grid(True, alpha=0.3)
axes[0].set_ylim([min(mnist_test_accs)-0.01, 1.0])

axes[1].bar(x_pos, cifar_test_accs, width=0.5, alpha=0.8,
color='orange')
axes[1].set_xlabel('Train-Validation Split')
axes[1].set_ylabel('Test Accuracy')
axes[1].set_title('RNN CIFAR-10 - Test Accuracy vs. Split')
axes[1].set_xticks(x_pos)
```

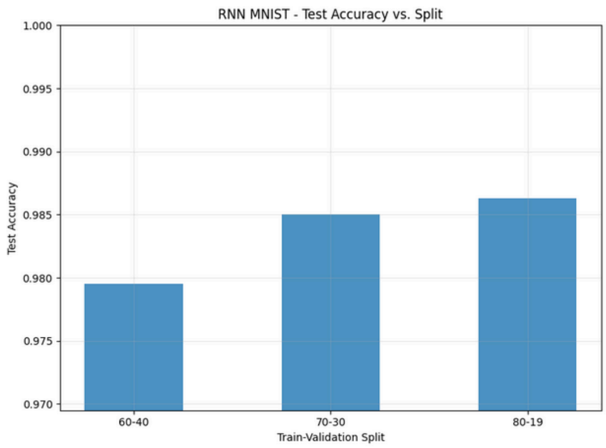
```
axes[1].set_xticklabels(splits_labels)
axes[1].grid(True, alpha=0.3)
axes[1].set_ylim([min(cifar_test_accs)-0.02,
max(cifar_test_accs)+0.02])

plt.tight_layout()
plt.show()

# Find and visualize the best models
best_split_mnist = split_ratios[np.argmax([r['test_acc'] for r in
results_mnist])]
best_split_cifar = split_ratios[np.argmax([r['test_acc'] for r in
results_cifar])]

best_model_mnist = models_mnist[best_split_mnist]
best_model_cifar = models_cifar[best_split_cifar]
best_history_mnist = histories_mnist[best_split_mnist]
best_history_cifar = histories_cifar[best_split_cifar]

# Confusion Matrices for Best Models
y_pred_mnist = best_model_mnist.predict(x_test_mnist.reshape(-1, 28,
28), verbose=0)
y_pred_mnist_classes = np.argmax(y_pred_mnist, axis=1)

y_pred_cifar = best_model_cifar.predict(x_test_cifar.reshape(-1, 32,
96), verbose=0)
y_pred_cifar_classes = np.argmax(y_pred_cifar, axis=1)

cm_mnist = confusion_matrix(y_test_mnist, y_pred_mnist_classes)
cm_cifar = confusion_matrix(y_test_cifar, y_pred_cifar_classes)

fig, axes = plt.subplots(1, 2, figsize=(18, 7))
sns.heatmap(cm_mnist, annot=True, fmt='d', cmap='Blues', ax=axes[0])
axes[0].set_title(f'RNN - MNIST Confusion Matrix (Best Split)')
axes[0].set_ylabel('True Label')
axes[0].set_xlabel('Predicted Label')

sns.heatmap(cm_cifar, annot=True, fmt='d', cmap='Greens', ax=axes[1])
axes[1].set_title(f'RNN - CIFAR-10 Confusion Matrix (Best Split)')
axes[1].set_ylabel('True Label')
axes[1].set_xlabel('Predicted Label')

plt.tight_layout()
plt.show()

# Training & Loss Curves for Best Models
fig, axes = plt.subplots(2, 2, figsize=(16, 12))
axes[0, 0].plot(best_history_mnist.history['accuracy'], label='Train')
axes[0, 0].plot(best_history_mnist.history['val_accuracy'],
label='Validation')
axes[0, 0].set_title(f'RNN - MNIST Accuracy (Best Split)')
```

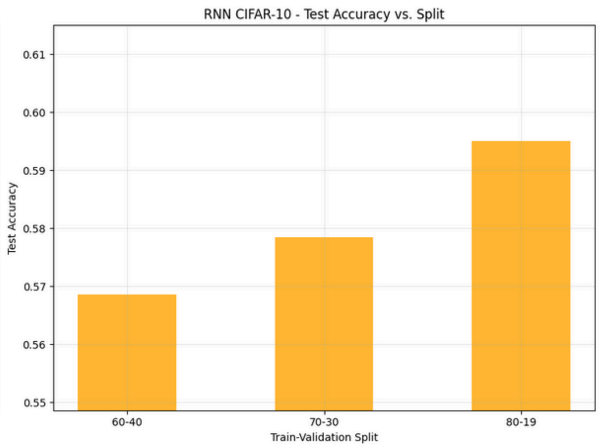
```
axes[0, 0].legend()
axes[0, 0].grid(True, alpha=0.3)
axes[0, 1].plot(best_history_mnist.history['loss'], label='Train')
axes[0, 1].plot(best_history_mnist.history['val_loss'],
label='Validation')
axes[0, 1].set_title(f'RNN - MNIST Loss (Best Split)')
axes[0, 1].legend()
axes[0, 1].grid(True, alpha=0.3)
axes[1, 0].plot(best_history_cifar.history['accuracy'], label='Train')
axes[1, 0].plot(best_history_cifar.history['val_accuracy'],
label='Validation')
axes[1, 0].set_title(f'RNN - CIFAR-10 Accuracy (Best Split)')
axes[1, 0].legend()
axes[1, 0].grid(True, alpha=0.3)
axes[1, 1].plot(best_history_cifar.history['loss'], label='Train')
axes[1, 1].plot(best_history_cifar.history['val_loss'],
label='Validation')
axes[1, 1].set_title(f'RNN - CIFAR-10 Loss (Best Split)')
axes[1, 1].legend()
axes[1, 1].grid(True, alpha=0.3)
plt.tight_layout()
plt.show()

# ROC/AUC Curves for Best Models
y_test_mnist_bin = label_binarize(y_test_mnist, classes=range(10))
y_test_cifar_bin = label_binarize(y_test_cifar, classes=range(10))
n_classes = 10
fpr_mnist, tpr_mnist, roc_auc_mnist = {}, {}, {}
fpr_cifar, tpr_cifar, roc_auc_cifar = {}, {}, {}

for i in range(n_classes):
    fpr_mnist[i], tpr_mnist[i], _ = roc_curve(y_test_mnist_bin[:, i],
y_pred_mnist[:, i])
    roc_auc_mnist[i] = auc(fpr_mnist[i], tpr_mnist[i])
    fpr_cifar[i], tpr_cifar[i], _ = roc_curve(y_test_cifar_bin[:, i],
y_pred_cifar[:, i])
    roc_auc_cifar[i] = auc(fpr_cifar[i], tpr_cifar[i])

fig, axes = plt.subplots(1, 2, figsize=(18, 7))
colors = cycle(['blue', 'red', 'green', 'orange', 'purple', 'brown',
'pink', 'gray', 'olive', 'cyan'])
for i, color in zip(range(n_classes), colors):
    axes[0].plot(fpr_mnist[i], tpr_mnist[i], color=color, lw=2,
label=f'Class {i} (AUC={roc_auc_mnist[i]:.3f})')
axes[0].plot([0, 1], [0, 1], 'k--', lw=2)
axes[0].set_title(f'RNN - MNIST ROC Curves (Best Split)')
axes[0].legend(loc="lower right")
axes[0].grid(True, alpha=0.3)

colors = cycle(['blue', 'red', 'green', 'orange', 'purple', 'brown',
'pink', 'gray', 'olive', 'cyan'])
```

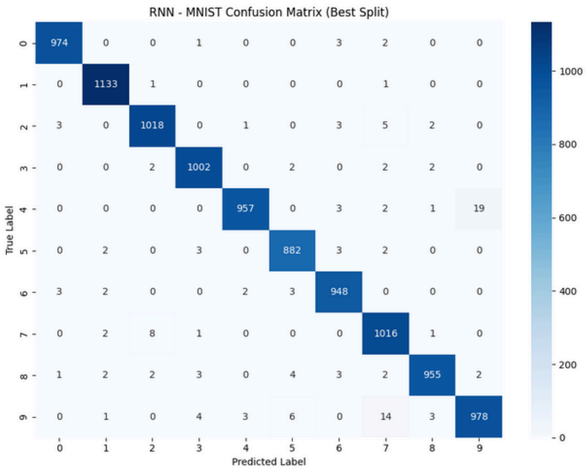
```
for i, color in zip(range(n_classes), colors):
    axes[1].plot(fpr_cifar[i], tpr_cifar[i], color=color, lw=2,
label=f'Class {i} (AUC={roc_auc_cifar[i]:.3f})')
axes[1].plot([0, 1], [0, 1], 'k--', lw=2)
axes[1].set_title(f'RNN - CIFAR-10 ROC Curves (Best Split)')
axes[1].legend(loc="lower right")
axes[1].grid(True, alpha=0.3)

plt.tight_layout()
plt.show()
```

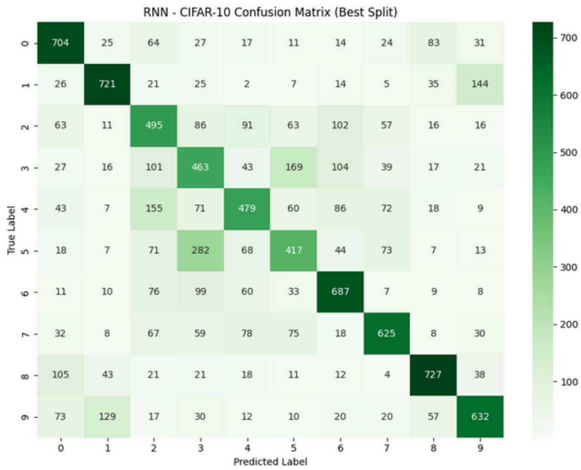
Results for RNN:

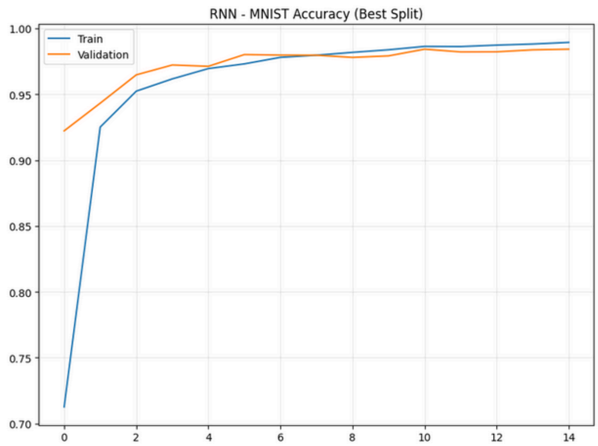
```
============================================================
RNN MNIST RESULTS - DIFFERENT TRAIN-TEST SPLITS
============================================================
split  train_acc  val_acc  test_acc
60-40   0.987778 0.979083   0.9795
70-30   0.988500 0.983444   0.9850
80-19   0.989396 0.984250   0.9863

============================================================
RNN CIFAR-10 RESULTS - DIFFERENT TRAIN-TEST SPLITS
============================================================
split  train_acc  val_acc  test_acc
60-40   0.858900  0.56085   0.5686
70-30   0.871629  0.58920   0.5784
80-19   0.872725  0.60380   0.5950
```

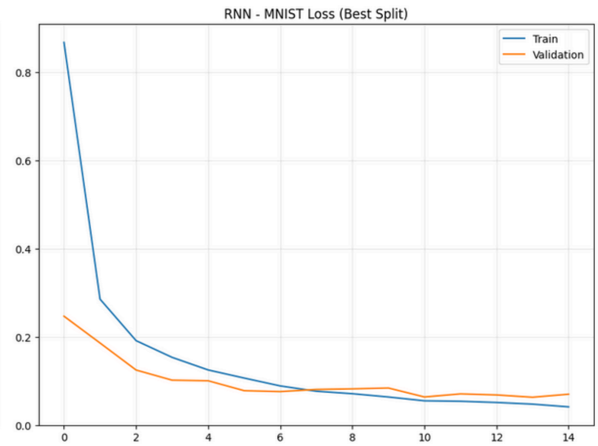


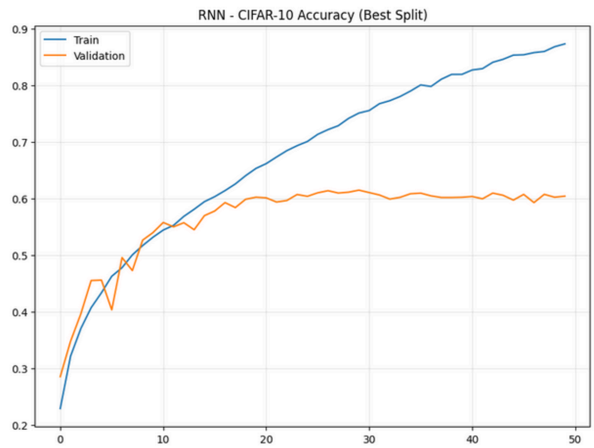


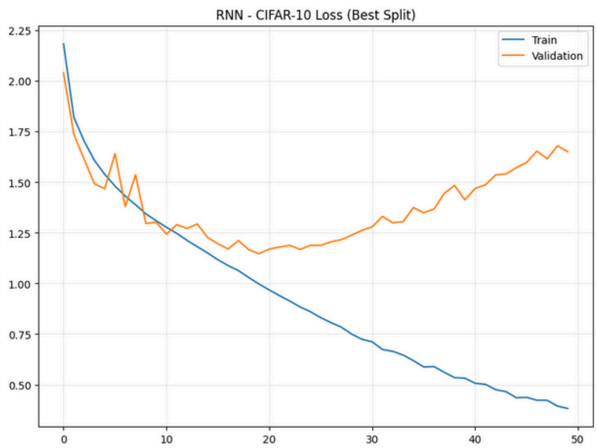

RNN - MNIST Confusion Matrix (Best Split)
True Label
Predicted Label

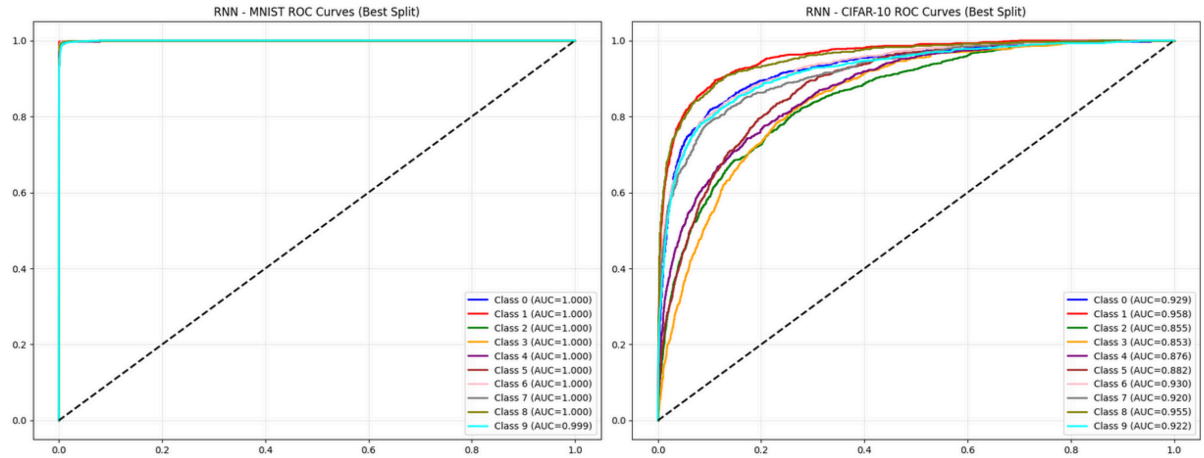

RNN - CIFAR-10 Confusion Matrix (Best Split)
True Label
Predicted Label


RNN - MNIST Accuracy (Best Split)
Train
Validation


RNN - MNIST Loss (Best Split)
Train
Validation


RNN - CIFAR-10 Accuracy (Best Split)
Train
Validation


RNN - CIFAR-10 Loss (Best Split)
Train
Validation



AlexNet

Code:

```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
from torchvision import models
from torch.utils.data import DataLoader, Subset

import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import confusion_matrix, roc_curve, auc
from sklearn.preprocessing import label_binarize
from itertools import cycle
import time
import copy

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")

def get_datasets():
    transform_std = transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229,
0.224, 0.225]),
    ])

    transform_mnist = transforms.Compose([
        transforms.Resize(224),
        transforms.Grayscale(num_output_channels=3),
```

```
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5,
0.5]),
    ])

    cifar10_train = torchvision.datasets.CIFAR10(root='./data',
train=True, download=True, transform=transform_std)
    cifar10_test = torchvision.datasets.CIFAR10(root='./data',
train=False, download=True, transform=transform_std)

    mnist_train = torchvision.datasets.MNIST(root='./data', train=True,
download=True, transform=transform_mnist)
    mnist_test = torchvision.datasets.MNIST(root='./data', train=False,
download=True, transform=transform_mnist)

    datasets = {
        'CIFAR-10': {'train': cifar10_train, 'test': cifar10_test,
'classes': cifar10_train.classes},
        'MNIST': {'train': mnist_train, 'test': mnist_test, 'classes':
[str(i) for i in range(10)]}
    }

    return datasets

def get_model(num_classes):
    model = models.alexnet(weights=models.AlexNet_Weights.DEFAULT)

    for param in model.parameters():
        param.requires_grad = False

    model.classifier[1] = nn.Linear(9216, 4096)
    model.classifier[4] = nn.Linear(4096, 1024)
    model.classifier[6] = nn.Linear(1024, num_classes)

    return model.to(device)

def plot_curves(history, dataset_name, model_name):
    plt.figure(figsize=(12, 5))

    plt.subplot(1, 2, 1)
    plt.plot(history['train_acc'], label='Train Accuracy')
    plt.plot(history['val_acc'], label='Validation Accuracy')
    plt.title(f'Accuracy Curves: {model_name} on {dataset_name}')
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy')
    plt.legend()

    plt.subplot(1, 2, 2)
    plt.plot(history['train_loss'], label='Train Loss')
    plt.plot(history['val_loss'], label='Validation Loss')
    plt.title(f'Loss Curves: {model_name} on {dataset_name}')
```

```
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.legend()

    plt.tight_layout()
    plt.show()

def plot_confusion_matrix(y_true, y_pred, classes, dataset_name,
model_name):
    cm = confusion_matrix(y_true, y_pred)
    plt.figure(figsize=(10, 8))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
xticklabels=classes, yticklabels=classes)
    plt.xlabel('Predicted Label')
    plt.ylabel('True Label')
    plt.title(f'Confusion Matrix: {model_name} on {dataset_name}')
    plt.show()

def plot_roc_auc(y_true, y_score, n_classes, classes, dataset_name,
model_name):
    y_true_bin = label_binarize(y_true, classes=list(range(n_classes)))

    fpr = dict()
    tpr = dict()
    roc_auc = dict()
    for i in range(n_classes):
        fpr[i], tpr[i], _ = roc_curve(y_true_bin[:, i], y_score[:, i])
        roc_auc[i] = auc(fpr[i], tpr[i])

    plt.figure(figsize=(10, 8))
    colors = cycle(['aqua', 'darkorange', 'cornflowerblue', 'green',
'red', 'purple', 'brown', 'pink', 'gray', 'olive'])
    for i, color in zip(range(n_classes), colors):
        plt.plot(fpr[i], tpr[i], color=color, lw=2,
                 label=f'ROC curve of class {classes[i]} (area =
{roc_auc[i]:0.2f})')

    plt.plot([0, 1], [0, 1], 'k--', lw=2)
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title(f'Receiver Operating Characteristic (ROC): {model_name}
on {dataset_name}')
    plt.legend(loc="lower right")
    plt.show()

def train_and_evaluate(model, train_loader, val_loader, criterion,
optimizer, num_epochs=5):
    history = {'train_loss': [], 'train_acc': [], 'val_loss': [],
'val_acc': []}
```

```
    best_model_wts = copy.deepcopy(model.state_dict())
    best_acc = 0.0

    for epoch in range(num_epochs):
        print(f'Epoch {epoch+1}/{num_epochs}')
        print('-' * 10)

        for phase in ['train', 'val']:
            if phase == 'train':
                model.train()
                dataloader = train_loader
            else:
                model.eval()
                dataloader = val_loader

            running_loss = 0.0
            running_corrects = 0

            for inputs, labels in dataloader:
                inputs, labels = inputs.to(device), labels.to(device)

                optimizer.zero_grad()

                with torch.set_grad_enabled(phase == 'train'):
                    outputs = model(inputs)
                    _, preds = torch.max(outputs, 1)
                    loss = criterion(outputs, labels)

                    if phase == 'train':
                        loss.backward()
                        optimizer.step()

                running_loss += loss.item() * inputs.size(0)
                running_corrects += torch.sum(preds == labels.data)

            epoch_loss = running_loss / len(dataloader.dataset)
            epoch_acc = running_corrects.double() /
len(dataloader.dataset)

            print(f'{phase} Loss: {epoch_loss:.4f} Acc:
{epoch_acc:.4f}')

            if phase == 'train':
                history['train_loss'].append(epoch_loss)
                history['train_acc'].append(epoch_acc.item())
            else:
                history['val_loss'].append(epoch_loss)
                history['val_acc'].append(epoch_acc.item())

            if phase == 'val' and epoch_acc > best_acc:
                best_acc = epoch_acc
```

```
                best_model_wts = copy.deepcopy(model.state_dict())

    print(f'Best val Acc: {best_acc:4f}')

    model.load_state_dict(best_model_wts)
    return model, history

def get_predictions(model, dataloader):
    model.eval()
    all_preds = torch.tensor([]).to(device)
    all_labels = torch.tensor([]).to(device)
    all_scores = torch.tensor([]).to(device)

    with torch.no_grad():
        for inputs, labels in dataloader:
            inputs, labels = inputs.to(device), labels.to(device)
            outputs = model(inputs)
            scores = torch.softmax(outputs, dim=1)
            _, preds = torch.max(outputs, 1)

            all_preds = torch.cat((all_preds, preds), dim=0)
            all_labels = torch.cat((all_labels, labels), dim=0)
            all_scores = torch.cat((all_scores, scores), dim=0)

    return all_labels.cpu().numpy(), all_preds.cpu().numpy(),
all_scores.cpu().numpy()

def main():
    datasets = get_datasets()
    model_names = ['AlexNet']
    split_sizes = [0.6, 0.7, 0.8]
    num_epochs = 10

    for dataset_name, dataset_info in datasets.items():
        print(f"\n{'='*20} DATASET: {dataset_name} {'='*20}")
        train_dataset = dataset_info['train']
        test_dataset = dataset_info['test']
        classes = dataset_info['classes']
        num_classes = len(classes)

        test_loader = DataLoader(test_dataset, batch_size=32,
shuffle=False)

        for model_name in model_names:
            print(f"\n{'--'*10} MODEL: {model_name} {'--'*10}")

            overall_best_accuracy = 0.0
            best_case_results = {}
            best_split_size = 0

            for split in split_sizes:
```

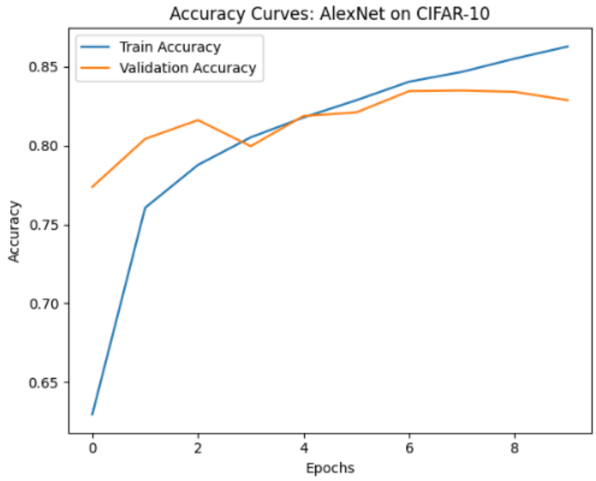
```
                print(f"\n--- Training with {int(split*100)}% of data
---")

                num_train = len(train_dataset)
                indices = list(range(num_train))
                np.random.shuffle(indices)
                split_idx = int(np.floor(split * num_train))
                train_idx = indices[:split_idx]
                train_subset = Subset(train_dataset, train_idx)

                num_subset_train = len(train_subset)
                val_split = 0.2
                val_idx_end = int(np.floor(val_split *
num_subset_train))
                val_indices_shuffled = list(range(num_subset_train))
                np.random.shuffle(val_indices_shuffled)

                final_train_indices_subset =
val_indices_shuffled[val_idx_end:]
                val_indices_subset = val_indices_shuffled[:val_idx_end]

                final_train_subset = Subset(train_subset,
final_train_indices_subset)
                val_subset = Subset(train_subset, val_indices_subset)

                train_loader = DataLoader(final_train_subset,
batch_size=32, shuffle=True)
                val_loader = DataLoader(val_subset, batch_size=32,
shuffle=False)

                model = get_model(num_classes)
                criterion = nn.CrossEntropyLoss()
                params_to_update = [param for param in
model.parameters() if param.requires_grad]
                optimizer = optim.SGD(params_to_update, lr=0.001,
momentum=0.9)

                trained_model, history = train_and_evaluate(model,
train_loader, val_loader, criterion, optimizer, num_epochs)

                y_true, y_pred, _ = get_predictions(trained_model,
test_loader)
                test_accuracy = np.mean(y_true == y_pred)
                print(f"Final Test Accuracy for {int(split*100)}%
split: {test_accuracy:.4f}")

                if test_accuracy > overall_best_accuracy:
                    print(f"*** New best model found with accuracy:
{test_accuracy:.4f} ***")
                    overall_best_accuracy = test_accuracy
                    best_split_size = split
```

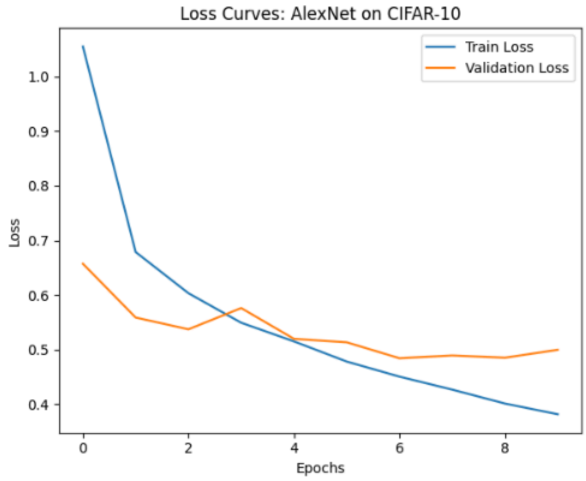
```
                    best_case_results['model'] = trained_model
                    best_case_results['history'] = history

            if 'model' in best_case_results:
                print(f"\n--- Generating plots for best case (from
{int(best_split_size*100)}% split with {overall_best_accuracy:.4f}
accuracy) ---")
                best_model = best_case_results['model']
                best_history = best_case_results['history']

                plot_curves(best_history, dataset_name, model_name)

                y_true, y_pred, y_score = get_predictions(best_model,
test_loader)

                plot_confusion_matrix(y_true, y_pred, classes,
dataset_name, model_name)

                plot_roc_auc(y_true, y_score, num_classes, classes,
dataset_name, model_name)

if __name__ == '__main__':
    main()
```

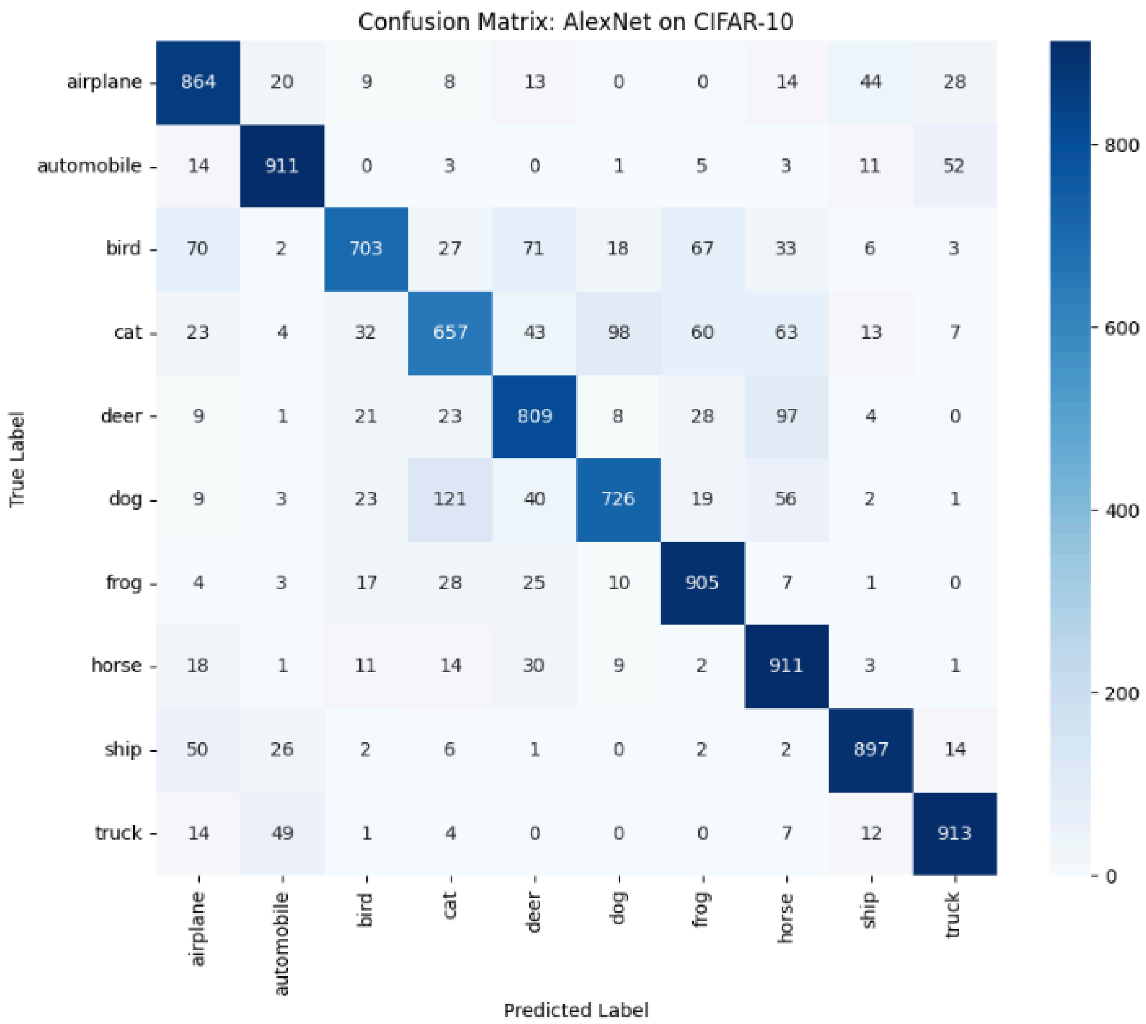
Results for AlexNet:

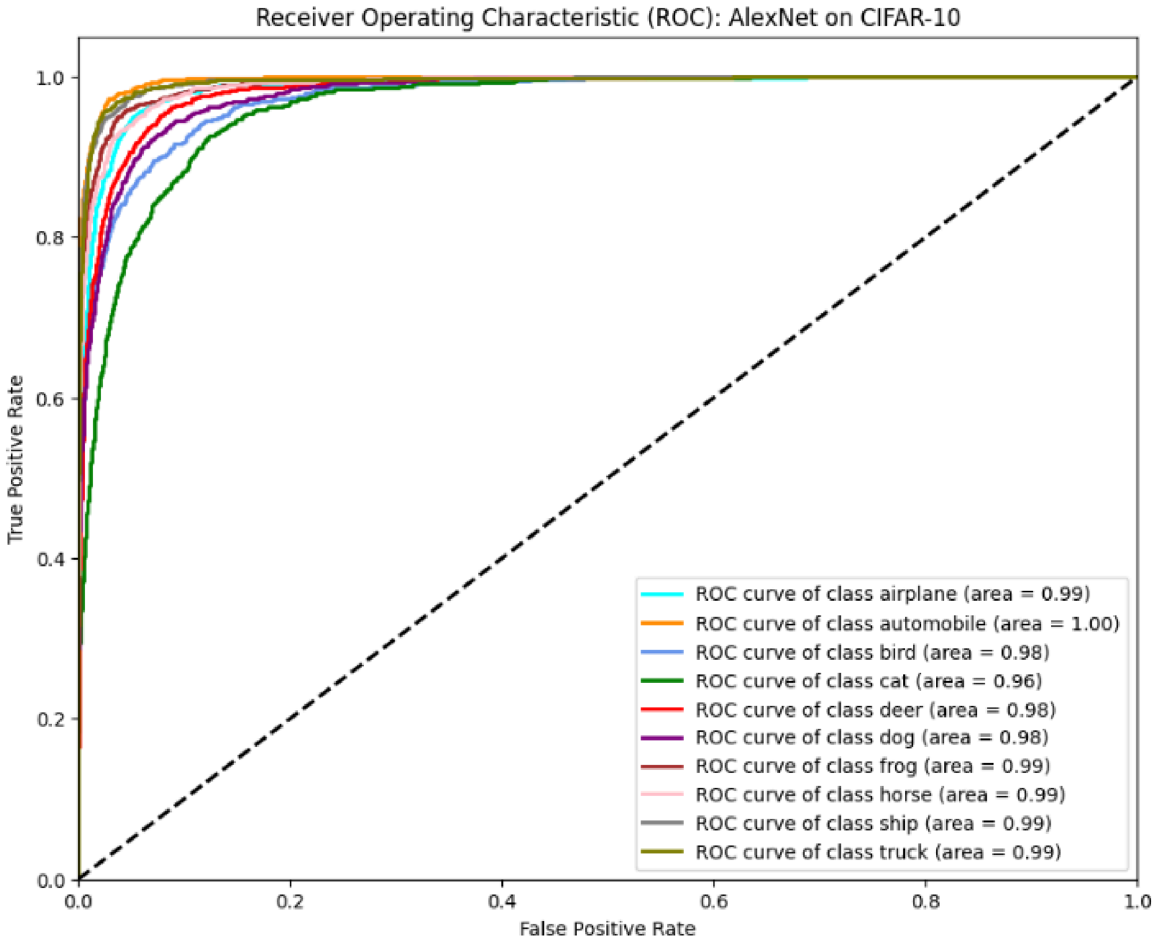
CIFAR10:

Split Size	Final Test Accuracy
60%	0.8216
70%	0.8290
80%	**0.8296**



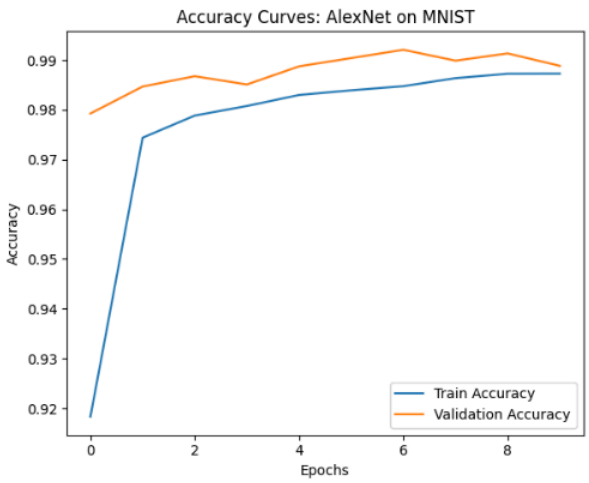


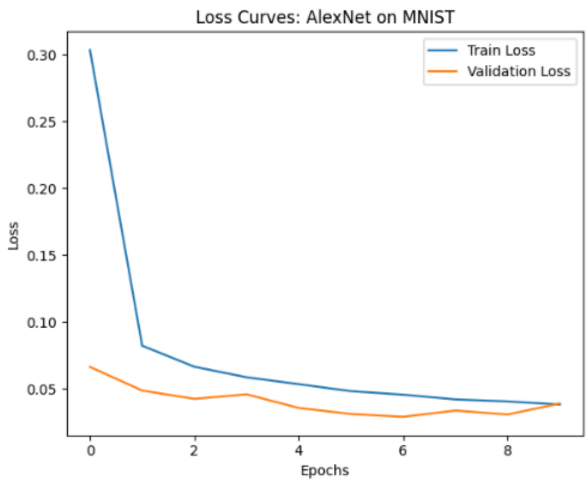

Confusion Matrix: AlexNet on CIFAR-10
True Label
Predicted Label
airplane 864 20 9 8 13 0 0 14 44 28
automobile 14 911 0 3 0 1 5 3 11 52
bird 70 2 703 27 71 18 67 33 6 3
cat 23 4 32 657 43 98 60 63 13 7
deer 9 1 21 23 809 8 28 97 4 0
dog 9 3 23 121 40 726 19 56 2 1
frog 4 3 17 28 25 10 905 7 1 0
horse 18 1 11 14 30 9 2 911 3 1
ship 50 26 2 6 1 0 2 2 897 14
truck 14 49 1 4 0 0 0 7 12 913
airplane
automobile
bird
cat
deer
dog
frog
horse
ship
truck
800
600
400
200
0

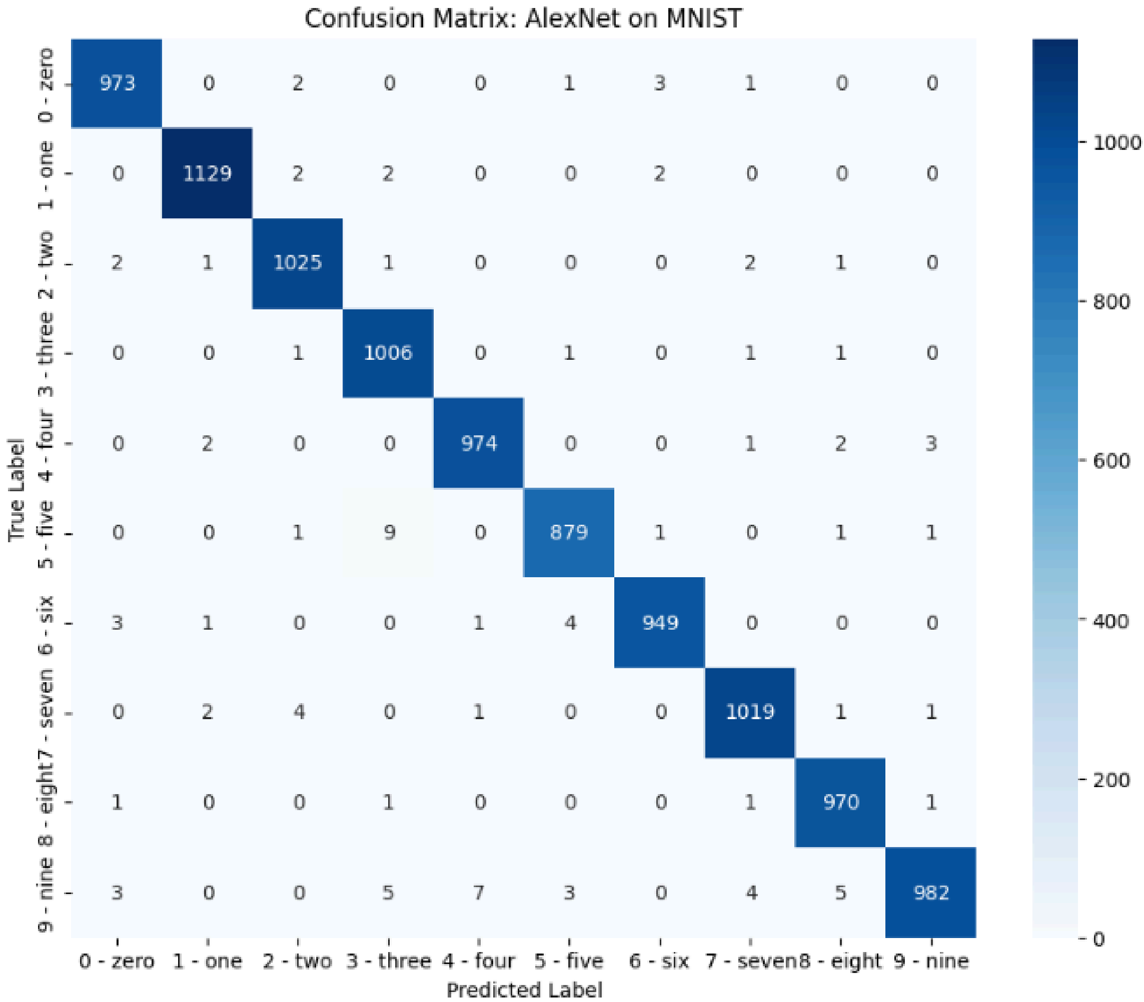


MNIST:

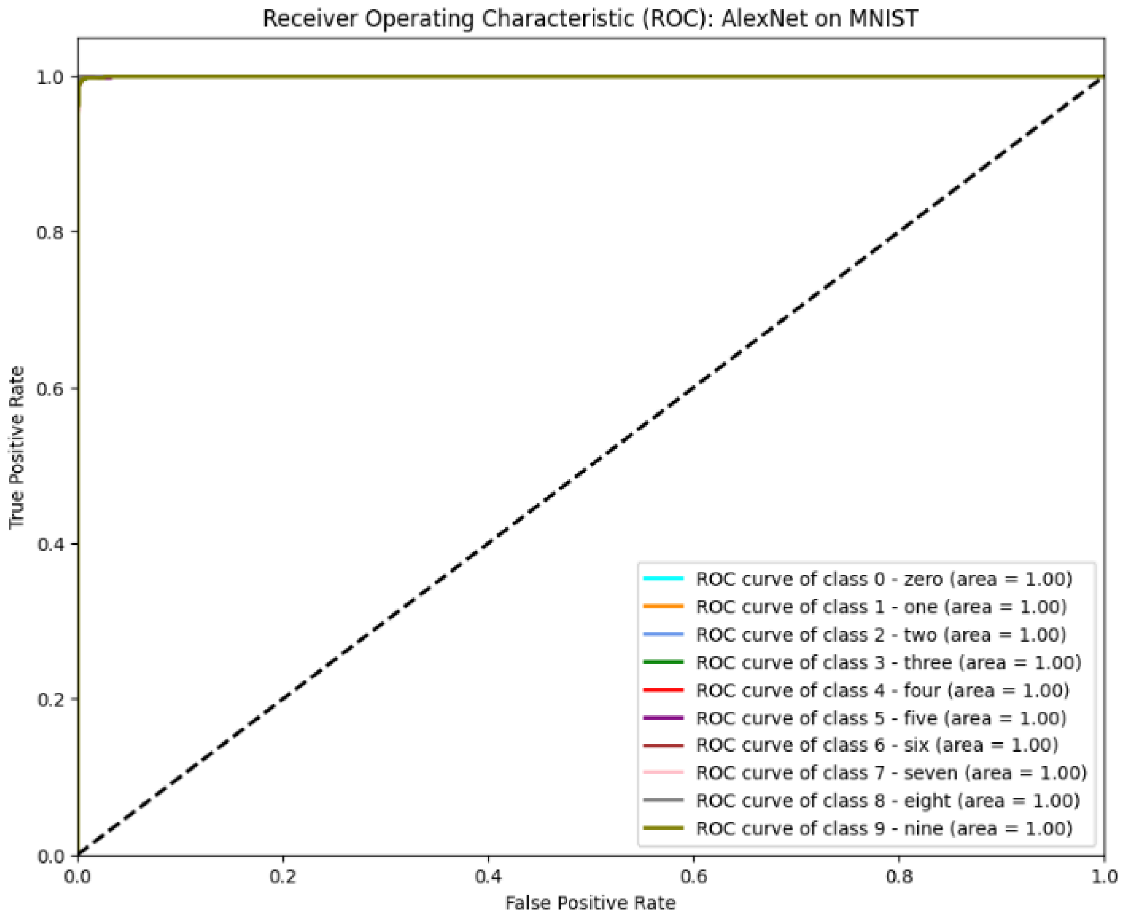
Split Size	Final Test Accuracy
60%	0.9895
70%	0.9896
80%	**0.9906**







Confusion Matrix: AlexNet on MNIST
True Label
Predicted Label
0 - zero
1 - one
2 - two
3 - three
4 - four
5 - five
6 - six
7 - seven
8 - eight
9 - nine
973 0 2 0 0 1 3 1 0 0
0 1129 2 2 0 0 2 0 0 0
2 1 1025 1 0 0 0 2 1 0
0 0 1 1006 0 1 0 1 1 0
0 2 0 0 974 0 0 1 2 3
0 0 1 9 0 879 1 0 1 1
3 1 0 0 1 4 949 0 0 0
0 2 4 0 1 0 0 1019 1 1
1 0 0 1 0 0 0 1 970 1
3 0 0 5 7 3 0 4 5 982
1000
800
600
400
200
0



GoogleNet

Code:

```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
from torchvision import models
from torch.utils.data import DataLoader, Subset
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import confusion_matrix, roc_curve, auc
from sklearn.preprocessing import label_binarize
from itertools import cycle
import time
import copy

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")

def get_datasets():
```

```
    transform_std = transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229,
0.224, 0.225]),
    ])

    transform_mnist = transforms.Compose([
        transforms.Resize(224),
        transforms.Grayscale(num_output_channels=3),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5,
0.5]),
    ])

    cifar10_train = torchvision.datasets.CIFAR10(root='./data',
train=True, download=True, transform=transform_std)
    cifar10_test = torchvision.datasets.CIFAR10(root='./data',
train=False, download=True, transform=transform_std)

    mnist_train = torchvision.datasets.MNIST(root='./data', train=True,
download=True, transform=transform_mnist)
    mnist_test = torchvision.datasets.MNIST(root='./data', train=False,
download=True, transform=transform_mnist)

    datasets = {
        'CIFAR-10': {'train': cifar10_train, 'test': cifar10_test,
'classes': cifar10_train.classes},
        'MNIST': {'train': mnist_train, 'test': mnist_test, 'classes':
mnist_train.classes}
    }

    return datasets

def get_model(num_classes=2, learning_rate=0.001):
    model = torchvision.models.googlenet(pretrained=True)
    model.fc = nn.Sequential(
        nn.Linear(in_features=1024, out_features=512),
        nn.ReLU(),
        nn.Linear(in_features=512, out_features=128),
        nn.ReLU(),
        nn.Linear(in_features=128, out_features=32),
        nn.ReLU(),
        nn.Linear(in_features=32, out_features=num_classes, bias=True)
    )
    model.to(device)
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=learning_rate)
    return model, criterion, optimizer
```

```
def plot_curves(history, dataset_name, model_name):
    plt.figure(figsize=(12, 5))

    plt.subplot(1, 2, 1)
    plt.plot(history['train_acc'], label='Train Accuracy')
    plt.plot(history['val_acc'], label='Validation Accuracy')
    plt.title(f'Accuracy Curves: {model_name} on {dataset_name}')
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy')
    plt.legend()

    plt.subplot(1, 2, 2)
    plt.plot(history['train_loss'], label='Train Loss')
    plt.plot(history['val_loss'], label='Validation Loss')
    plt.title(f'Loss Curves: {model_name} on {dataset_name}')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.legend()

    plt.tight_layout()
    plt.show()

def plot_confusion_matrix(y_true, y_pred, classes, dataset_name,
model_name):
    cm = confusion_matrix(y_true, y_pred)
    plt.figure(figsize=(10, 8))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
xticklabels=classes, yticklabels=classes)
    plt.xlabel('Predicted Label')
    plt.ylabel('True Label')
    plt.title(f'Confusion Matrix: {model_name} on {dataset_name}')
    plt.show()

def plot_roc_auc(y_true, y_score, n_classes, classes, dataset_name,
model_name):
    y_true_bin = label_binarize(y_true, classes=list(range(n_classes)))

    fpr = dict()
    tpr = dict()
    roc_auc = dict()
    for i in range(n_classes):
        fpr[i], tpr[i], _ = roc_curve(y_true_bin[:, i], y_score[:, i])
        roc_auc[i] = auc(fpr[i], tpr[i])

    plt.figure(figsize=(10, 8))
    colors = cycle(['aqua', 'darkorange', 'cornflowerblue', 'green',
'red', 'purple', 'brown', 'pink', 'gray', 'olive'])
    for i, color in zip(range(n_classes), colors):
        plt.plot(fpr[i], tpr[i], color=color, lw=2,
                 label=f'ROC curve of class {classes[i]} (area =
{roc_auc[i]:0.2f})')
```

```
    plt.plot([0, 1], [0, 1], 'k--', lw=2)
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title(f'Receiver Operating Characteristic (ROC): {model_name}
on {dataset_name}')
    plt.legend(loc="lower right")
    plt.show()

def train_and_evaluate(model, train_loader, val_loader, criterion,
optimizer, num_epochs=5):
    history = {'train_loss': [], 'train_acc': [], 'val_loss': [],
'val_acc': []}
    best_model_wts = copy.deepcopy(model.state_dict())
    best_acc = 0.0

    for epoch in range(num_epochs):
        print(f'Epoch {epoch+1}/{num_epochs}')
        print('-' * 10)

        for phase in ['train', 'val']:
            if phase == 'train':
                model.train()
                dataloader = train_loader
            else:
                model.eval()
                dataloader = val_loader

            running_loss = 0.0
            running_corrects = 0

            for inputs, labels in dataloader:
                inputs, labels = inputs.to(device), labels.to(device)
                optimizer.zero_grad()

                with torch.set_grad_enabled(phase == 'train'):
                    outputs = model(inputs)
                    loss = criterion(outputs, labels)
                    _, preds = torch.max(outputs, 1)

                    if phase == 'train':
                        loss.backward()
                        optimizer.step()

                running_loss += loss.item() * inputs.size(0)
                running_corrects += torch.sum(preds == labels.data)

            epoch_loss = running_loss / len(dataloader.dataset)
```

```
            epoch_acc = running_corrects.double() /
len(dataloader.dataset)

            print(f'{phase} Loss: {epoch_loss:.4f} Acc:
{epoch_acc:.4f}')

            if phase == 'train':
                history['train_loss'].append(epoch_loss)
                history['train_acc'].append(epoch_acc.item())
            else:
                history['val_loss'].append(epoch_loss)
                history['val_acc'].append(epoch_acc.item())

            if phase == 'val' and epoch_acc > best_acc:
                best_acc = epoch_acc
                best_model_wts = copy.deepcopy(model.state_dict())

    print(f'Best val Acc: {best_acc:.4f}')
    model.load_state_dict(best_model_wts)
    return model, history

def get_predictions(model, dataloader):
    model.eval()
    all_preds = torch.tensor([]).to(device)
    all_labels = torch.tensor([]).to(device)
    all_scores = torch.tensor([]).to(device)

    with torch.no_grad():
        for inputs, labels in dataloader:
            inputs, labels = inputs.to(device), labels.to(device)
            outputs = model(inputs)
            scores = torch.softmax(outputs, dim=1)
            _, preds = torch.max(outputs, 1)

            all_preds = torch.cat((all_preds, preds), dim=0)
            all_labels = torch.cat((all_labels, labels), dim=0)
            all_scores = torch.cat((all_scores, scores), dim=0)

    return all_labels.cpu().numpy(), all_preds.cpu().numpy(),
all_scores.cpu().numpy()

def main():
    datasets = get_datasets()
    model_names = ['GoogleNet']
    split_sizes = [0.6, 0.7, 0.8]
    num_epochs = 10

    for dataset_name, dataset_info in datasets.items():
        print(f"\n{'='*20} DATASET: {dataset_name} {'='*20}")
        train_dataset = dataset_info['train']
        test_dataset = dataset_info['test']
```

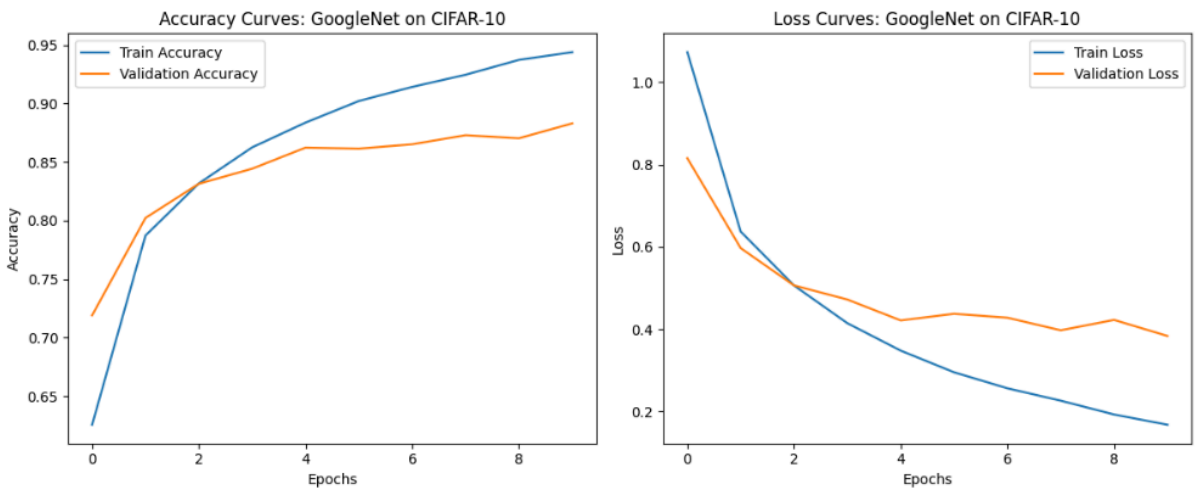
```
        classes = dataset_info['classes']
        num_classes = len(classes)

        test_loader = DataLoader(test_dataset, batch_size=32,
shuffle=False)

        for model_name in model_names:
            print(f"\n{'--'*10} MODEL: {model_name} {'--'*10}")

            overall_best_accuracy = 0.0
            best_case_results = {}
            best_split_size = 0

            for split in split_sizes:
                print(f"\n--- Training with {int(split*100)}% of data
---")

                num_train = len(train_dataset)
                indices = list(range(num_train))
                np.random.shuffle(indices)
                split_idx = int(np.floor(split * num_train))
                train_idx = indices[:split_idx]
                train_subset = Subset(train_dataset, train_idx)

                num_subset_train = len(train_subset)
                val_split = 0.2
                val_idx_end = int(np.floor(val_split *
num_subset_train))
                val_indices = list(range(num_subset_train))
                np.random.shuffle(val_indices)

                final_train_indices = val_indices[val_idx_end:]
                val_indices = val_indices[:val_idx_end]

                final_train_subset = Subset(train_subset,
final_train_indices)
                val_subset = Subset(train_subset, val_indices)

                train_loader = DataLoader(final_train_subset,
batch_size=32, shuffle=True)
                val_loader = DataLoader(val_subset, batch_size=32,
shuffle=False)

                model, criterion, optimizer = get_model(num_classes)

                trained_model, history = train_and_evaluate(model,
train_loader, val_loader, criterion, optimizer, num_epochs)

                y_true, y_pred, _ = get_predictions(trained_model,
test_loader)
                test_accuracy = np.mean(y_true == y_pred)
```

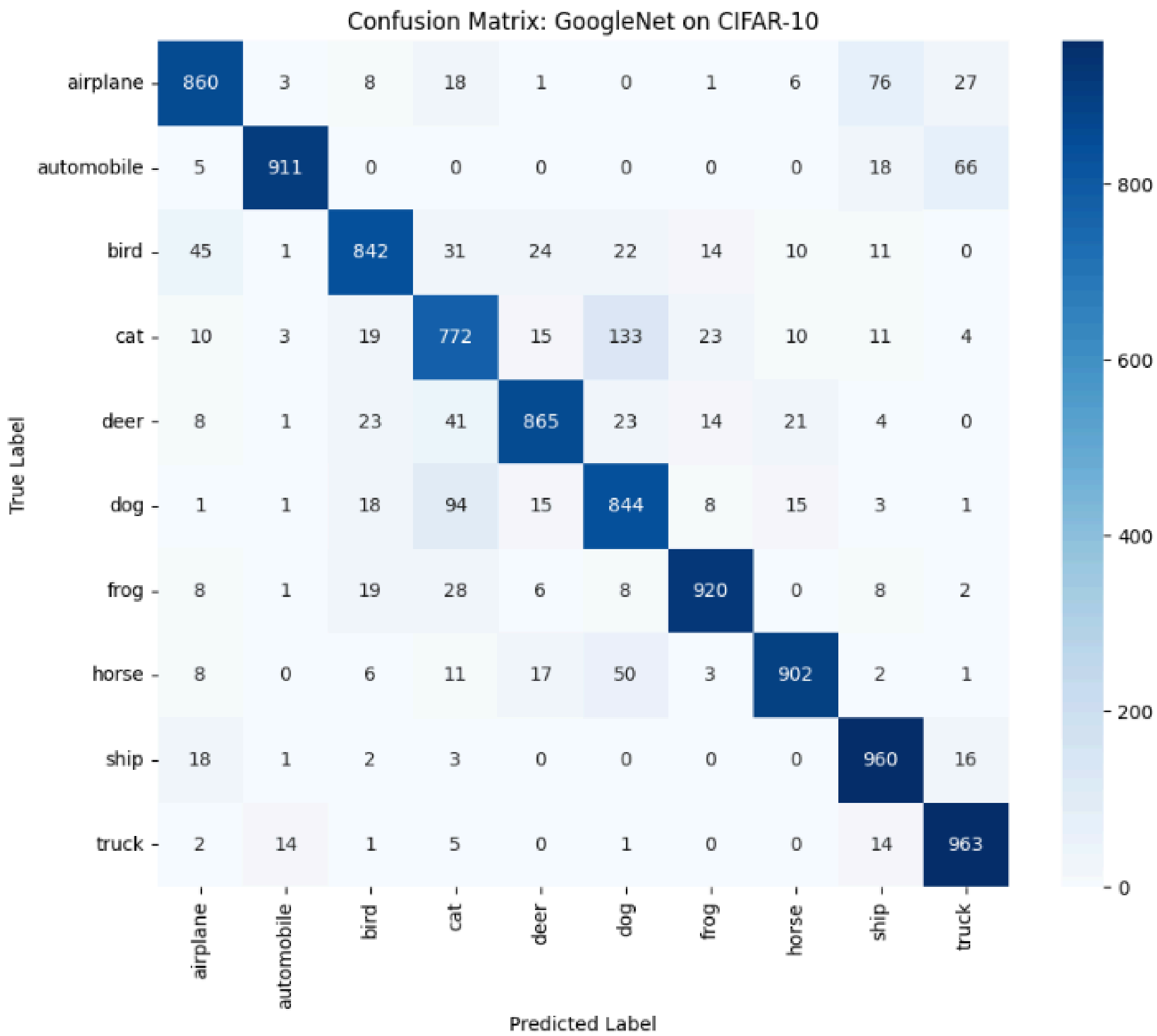
```
                print(f"Final Test Accuracy for {int(split*100)}%
split: {test_accuracy:.4f}")

                if test_accuracy > overall_best_accuracy:
                    print(f"*** New best model found with accuracy:
{test_accuracy:.4f} ***")
                    overall_best_accuracy = test_accuracy
                    best_split_size = split
                    best_case_results['model'] = trained_model
                    best_case_results['history'] = history

            if 'model' in best_case_results:
                print(f"\n--- Generating plots for best case (from
{int(best_split_size*100)}% split with {overall_best_accuracy:.4f}
accuracy) ---")
                best_model = best_case_results['model']
                best_history = best_case_results['history']

                plot_curves(best_history, dataset_name, model_name)

                y_true, y_pred, y_score = get_predictions(best_model,
test_loader)
                plot_confusion_matrix(y_true, y_pred, classes,
dataset_name, model_name)
                plot_roc_auc(y_true, y_score, num_classes, classes,
dataset_name, model_name)

if __name__ == '__main__':
    main()
```

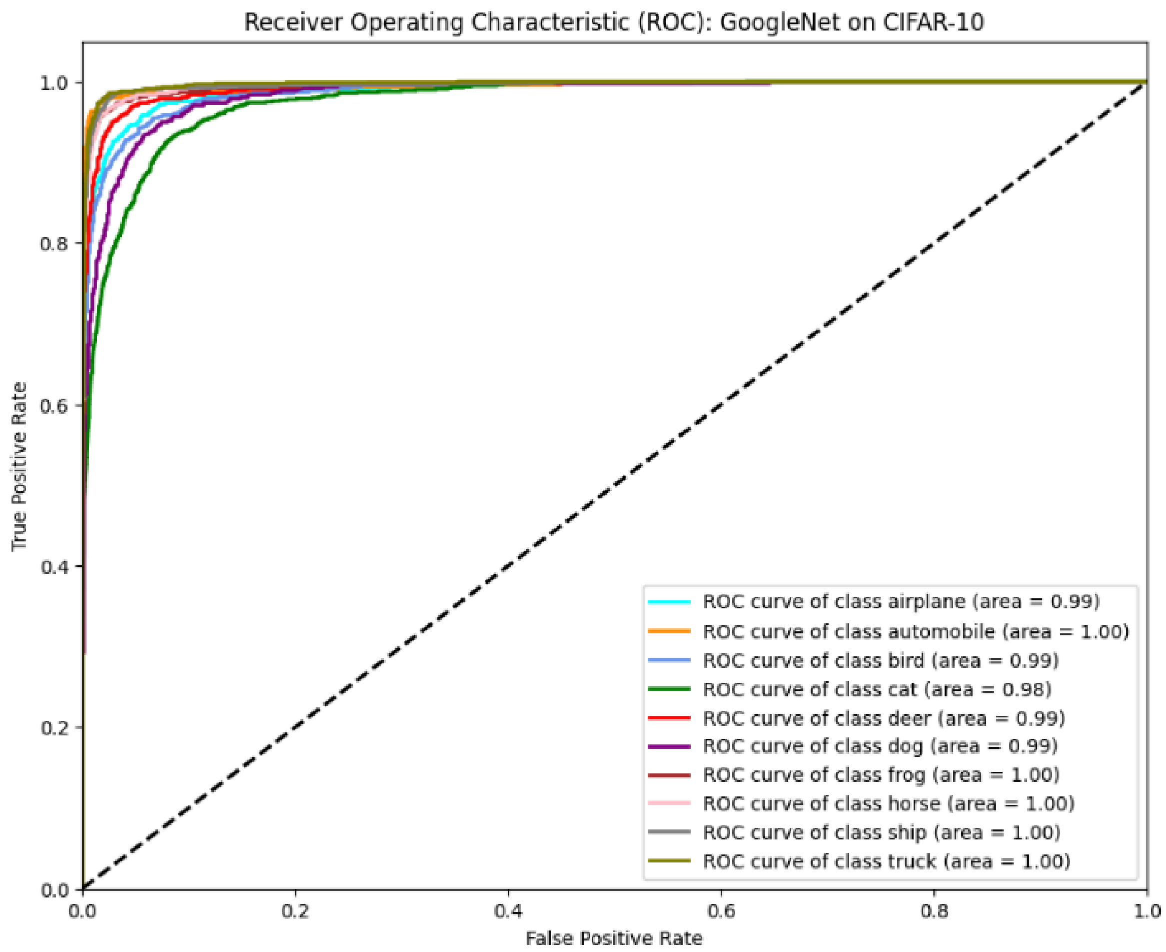
Results for GoogleNet:

CIFAR-10

Split Size	Test Accuracy
60%	0.8638
70%	0.8744
80%	0.8839

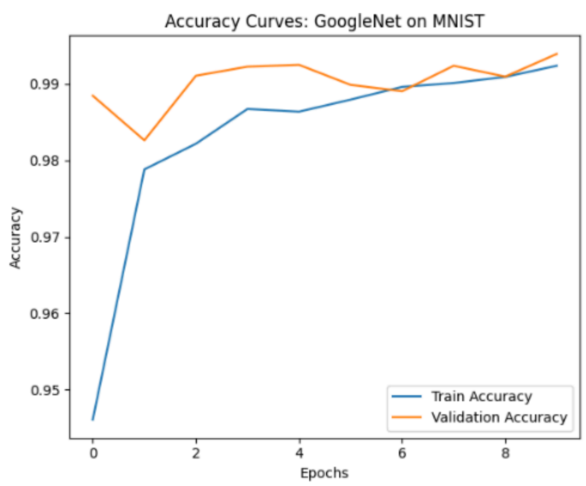

Accuracy Curves: GoogleNet on CIFAR-10
Train Accuracy
Validation Accuracy
Accuracy
Epochs
0.95
0.90
0.85
0.80
0.75
0.70
0.65
0 2 4 6 8
Loss Curves: GoogleNet on CIFAR-10
Train Loss
Validation Loss
Loss
Epochs
1.0
0.8
0.6
0.4
0.2
0 2 4 6 8

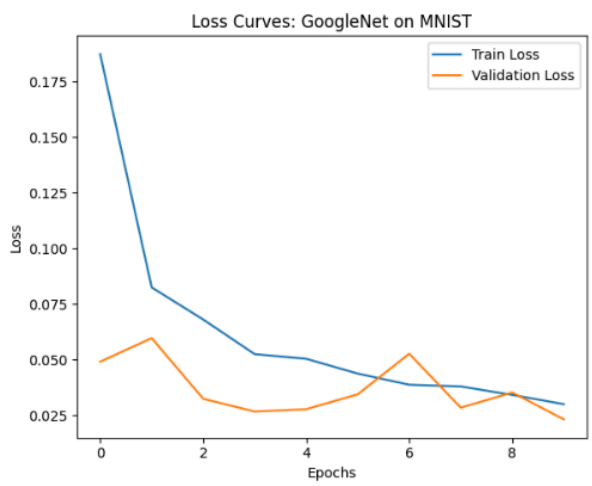

Confusion Matrix: GoogleNet on CIFAR-10
True Label
Predicted Label
airplane 860 3 8 18 1 0 1 6 76 27
automobile 5 911 0 0 0 0 0 0 18 66
bird 45 1 842 31 24 22 14 10 11 0
cat 10 3 19 772 15 133 23 10 11 4
deer 8 1 23 41 865 23 14 21 4 0
dog 1 1 18 94 15 844 8 15 3 1
frog 8 1 19 28 6 8 920 0 8 2
horse 8 0 6 11 17 50 3 902 2 1
ship 18 1 2 3 0 0 0 0 960 16
truck 2 14 1 5 0 1 0 0 14 963
airplane automobile bird cat deer dog frog horse ship truck
800
600
400
200
0

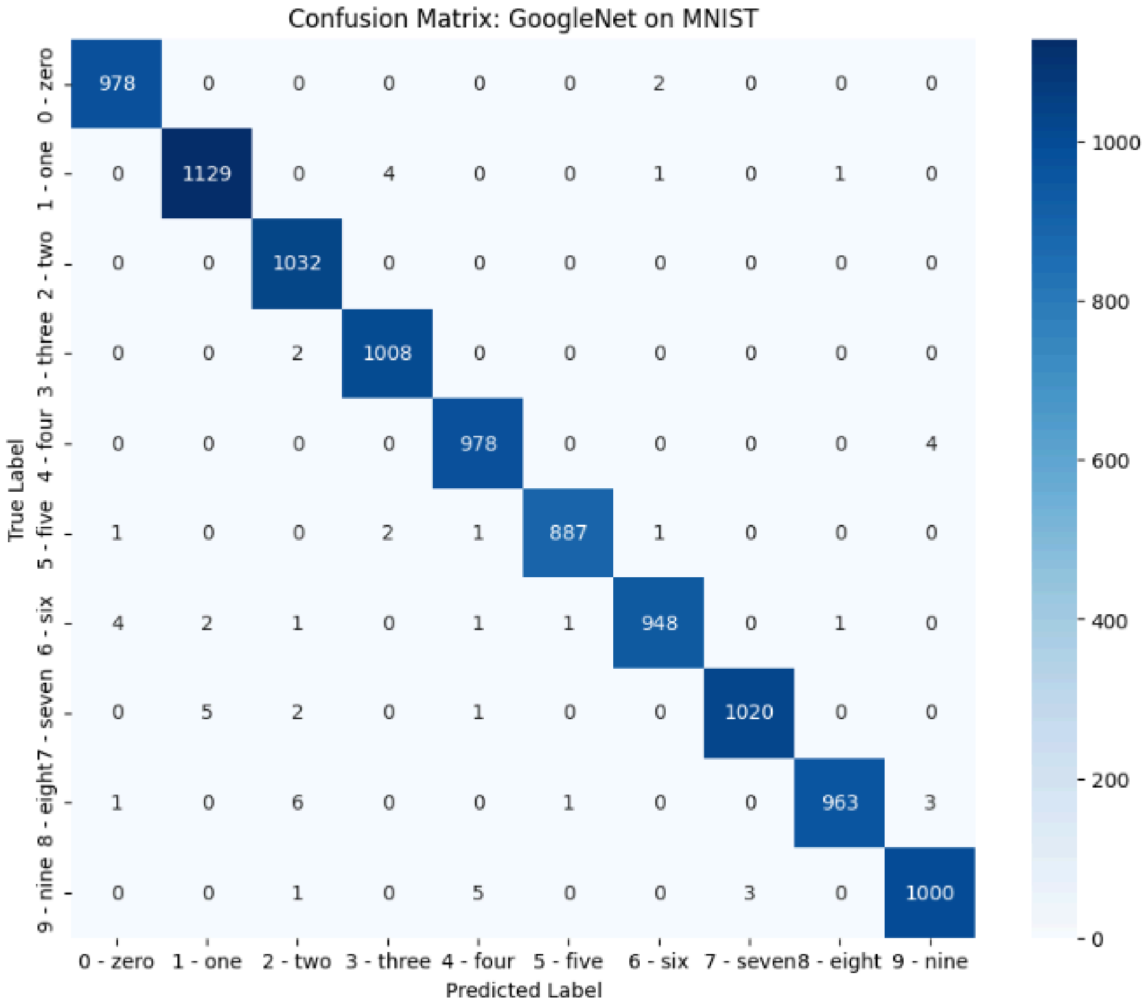


MNIST

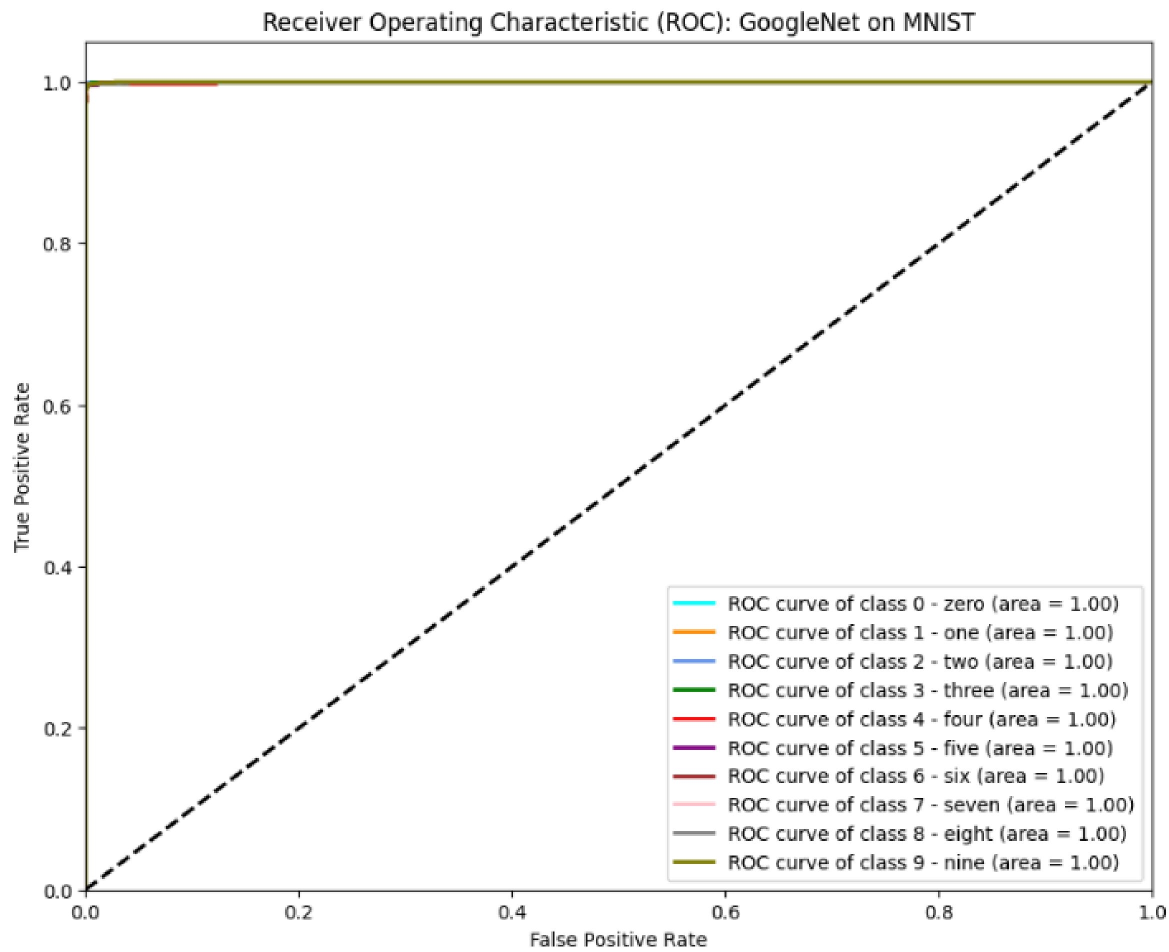
Split Size	Test Accuracy
60%	0.9912
70%	0.9943
80%	0.9929







Confusion Matrix: GoogleNet on MNIST
True Label
Predicted Label
0 - zero
1 - one
2 - two
3 - three
4 - four
5 - five
6 - six
7 - seven
8 - eight
9 - nine
978 0 0 0 0 0 2 0 0 0
0 1129 0 4 0 0 1 0 1 0
0 0 1032 0 0 0 0 0 0 0
0 0 2 1008 0 0 0 0 0 0
0 0 0 0 978 0 0 0 0 4
1 0 0 2 1 887 1 0 0 0
4 2 1 0 1 1 948 0 1 0
0 5 2 0 1 0 0 1020 0 0
1 0 6 0 0 1 0 0 963 3
0 0 1 0 5 0 0 3 0 1000
1000
800
600
400
200
0



Discussion and Comparison:

MNIST: On the relatively simple MNIST dataset of handwritten digits, all tested models performed exceptionally well.

- CNN-based models (Custom CNN, VGG16, AlexNet, GoogleNet) all achieved near-perfect test accuracies, exceeding 99%. GoogleNet was the top performer with 99.43% accuracy.
- The RNN model also achieved a very high accuracy of 98.63%, though it was slightly edged out by the CNNs. For this task, the dataset's simplicity meant that even an architecture not specialized for images could succeed.

CIFAR-10: The more complex, real-world images of the CIFAR-10 dataset highlighted the significant differences in architectural suitability.

- GoogleNet was the clear winner, achieving the highest test accuracy of 88.39%, demonstrating the power of its advanced Inception architecture.
- The custom CNN and VGG16 performed very well and were tied, both reaching a peak accuracy of 86.06%. AlexNet followed with a solid 82.96%.
- The RNN failed significantly on this task, with a top accuracy of only 59.50%. This poor performance is because processing an image as a sequence of rows loses the critical 2D spatial information that CNNs are designed to capture.

Conclusion and Key Takeaways

The results lead to several key conclusions:

1. Architecture is crucial: CNNs are fundamentally better suited for image classification than RNNs. The performance gap on CIFAR-10 makes this indisputable.
2. Advanced CNNs Perform Better: The performance ranking on CIFAR-10 (GoogleNet > VGG16 > AlexNet) reflects the evolution of CNN design, with more modern architectures achieving better results.
3. More Data Boosts Performance: For the challenging CIFAR-10 dataset, all models showed a clear improvement in accuracy as the size of the training set was increased.

In summary, the experiment confirms that while many architectures can solve simple problems, complex tasks like real-world image classification require specialized models like CNNs, with modern architectures and larger datasets yielding the best results.