```
import numpy as np
import matplotlib.pyplot as plt
import cv2
from tensorflow.keras.datasets import mnist
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report,
roc_curve, auc


(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train = x_train.astype("float32") / 255.0
x_test = x_test.astype("float32") / 255.0
x_train = x_train[..., np.newaxis]
x_test = x_test[..., np.newaxis]
y_train_cat = tf.keras.utils.to_categorical(y_train, 10)
y_test_cat = tf.keras.utils.to_categorical(y_test, 10)


x_train_new, x_val, y_train_new, y_val = train_test_split(x_train, y_train_cat,
test_size=0.1, random_state=42)


train_datagen = ImageDataGenerator(
        rotation_range=10,
        zoom_range=0.1,
        width_shift_range=0.1,
        height_shift_range=0.1
)


def build_cnn():
        model = tf.keras.Sequential([
        tf.keras.layers.Input(shape=(28, 28, 1)),
        tf.keras.layers.Conv2D(32, 3, activation='relu'),
        tf.keras.layers.MaxPooling2D(),
        tf.keras.layers.Conv2D(64, 3, activation='relu'),
        tf.keras.layers.MaxPooling2D(),
```

```
        tf.keras.layers.Flatten(),
        tf.keras.layers.Dense(128, activation='relu'),
        tf.keras.layers.Dropout(0.4),
        tf.keras.layers.Dense(10, activation='softmax')
        ])
        model.compile(optimizer='adam', loss='categorical_crossentropy',
metrics=['accuracy'])
        return model


def build_rnn():
        model = tf.keras.Sequential([
        tf.keras.layers.Input(shape=(28, 28)),
        tf.keras.layers.LSTM(128, return_sequences=True),
        tf.keras.layers.LSTM(64),
        tf.keras.layers.Dense(128, activation='relu'),
        tf.keras.layers.Dropout(0.4),
        tf.keras.layers.Dense(10, activation='softmax')
        ])
        model.compile(optimizer='adam', loss='categorical_crossentropy',
metrics=['accuracy'])
        return model


cnn_model1 = build_cnn()
cnn_model2 = build_cnn()
rnn_model = build_rnn()


print("Training CNN Model 1...")
cnn_model1.fit(train_datagen.flow(x_train_new, y_train_new, batch_size=128),
        epochs=5, validation_data=(x_val, y_val), verbose=2)


print("Training CNN Model 2...")
cnn_model2.fit(train_datagen.flow(x_train_new, y_train_new, batch_size=128, seed=42),
        epochs=5, validation_data=(x_val, y_val), verbose=2)
```

```
print("Training RNN Model...")
rnn_model.fit(x_train_new.squeeze(), y_train_new, batch_size=128,
        epochs=5, validation_data=(x_val.squeeze(), y_val), verbose=2)


def tta_predict(model, x_test, num_augments=5):
        preds = []
        aug_gen = ImageDataGenerator(
        rotation_range=10,
        zoom_range=0.1,
        width_shift_range=0.1,
        height_shift_range=0.1
        )
        for _ in range(num_augments):
        augmented = aug_gen.flow(x_test, batch_size=128, shuffle=False)
        preds.append(model.predict(augmented, verbose=0))
        return np.mean(preds, axis=0)


print("Generating predictions with test-time augmentation...")
preds_cnn1 = tta_predict(cnn_model1, x_test)
preds_cnn2 = tta_predict(cnn_model2, x_test)


preds_rnn = rnn_model.predict(x_test.squeeze(), verbose=0)


final_preds = (preds_cnn1 + preds_cnn2 + preds_rnn) / 3
final_labels = np.argmax(final_preds, axis=1)


final_accuracy = accuracy_score(y_test, final_labels)
print(f"\n Final CNN + RNN Ensemble Test Accuracy: {final_accuracy * 100:.2f}%")


conf_matrix = confusion_matrix(y_test, final_labels)
print("\n Confusion Matrix:")
print(conf_matrix)
```

```
print("\n Classification Report:")
print(classification_report(y_test, final_labels))


fpr = {}
tpr = {}
roc_auc = {}


for i in range(10):
        fpr[i], tpr[i], _ = roc_curve(y_test_cat[:, i], final_preds[:, i])
        roc_auc[i] = auc(fpr[i], tpr[i])


plt.figure(figsize=(10, 8))
for i in range(10):
        plt.plot(fpr[i], tpr[i], label=f"Digit {i} (AUC = {roc_auc[i]:.2f})")


plt.plot([0, 1], [0, 1], "k--")
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("Multi-class ROC Curve (CNN + RNN Ensemble)")
plt.legend()
plt.savefig('roc_curve.png')
plt.show()


def improved_image_preprocessing(image_path):
        try:
        img_original = cv2.imread(image_path)
        if img_original is None:
        raise FileNotFoundError(f"Could not read image at {image_path}")
        except Exception as e:
        print(f"Error loading image: {e}")
        return None, None, None
```

```
        if len(img_original.shape) == 3:
        img_gray = cv2.cvtColor(img_original, cv2.COLOR_BGR2GRAY)
        else:
        img_gray = img_original.copy()


        img_original_display = img_gray.copy()


        img_resized = cv2.resize(img_gray, (28, 28))


        mean_val = np.mean(img_resized)
        if mean_val > 127:
        img_inverted = 255 - img_resized
        else:
        img_inverted = img_resized


        img_normalized = img_inverted.astype("float32") / 255.0


        img_for_thresh = (img_normalized * 255).astype(np.uint8)
        img_thresh = cv2.adaptiveThreshold(
        img_for_thresh, 255, cv2.ADAPTIVE_THRESH_GAUSSIAN_C,
cv2.THRESH_BINARY, 11, 2)


        img_binary = img_thresh.astype("float32") / 255.0


        kernel = np.ones((2, 2), np.uint8)
        img_binary_uint8 = (img_binary * 255).astype(np.uint8)
        img_cleaned = cv2.morphologyEx(img_binary_uint8, cv2.MORPH_OPEN,
kernel)
        img_cleaned = cv2.morphologyEx(img_cleaned, cv2.MORPH_CLOSE, kernel)
```

```
img_cleaned = img_cleaned.astype("float32") / 255.0


img_cnn = np.expand_dims(img_cleaned, axis=[0, -1])
img_rnn = np.expand_dims(img_cleaned, axis=0)


plt.figure(figsize=(15, 10))


plt.subplot(231)
plt.imshow(img_original_display, cmap='gray')
plt.title('Original Image')
plt.axis('off')


plt.subplot(232)
plt.imshow(img_resized, cmap='gray')
plt.title('Resized (28x28)')
plt.axis('off')


plt.subplot(233)
plt.imshow(img_inverted, cmap='gray')
plt.title('Inverted (if needed)')
plt.axis('off')


plt.subplot(234)
plt.imshow(img_normalized, cmap='gray')
plt.title('Normalized')
plt.axis('off')


plt.subplot(235)
plt.imshow(img_binary, cmap='gray')
plt.title('Binary Threshold')
plt.axis('off')
```

```
    plt.subplot(236)
    plt.imshow(img_cleaned, cmap='gray')
    plt.title('Cleaned (Final)')
    plt.axis('off')


    plt.tight_layout()
    plt.savefig('preprocessing_steps.png')
    plt.show()


    _, img_standard = cv2.threshold(img_normalized, 0.3, 1,
cv2.THRESH_BINARY)
    img_cnn_standard = np.expand_dims(img_standard, axis=[0, -1])
    img_rnn_standard = np.expand_dims(img_standard, axis=0)


    preprocessing_methods = {
    'standard': (img_cnn_standard, img_rnn_standard, img_standard),
    'advanced': (img_cnn, img_rnn, img_cleaned)
    }


    return preprocessing_methods, img_original_display


def predict_with_visualization(image_path):
    preprocessing_methods, original_img =
improved_image_preprocessing(image_path)


    if preprocessing_methods is None:
    print(f"Failed to process image at {image_path}")
    return None
```

```
    results = {}

    for method_name, (img_cnn, img_rnn, img_display) in
preprocessing_methods.items():
    pred_cnn1 = cnn_model1.predict(img_cnn, verbose=0)
    pred_cnn2 = cnn_model2.predict(img_cnn, verbose=0)
    pred_rnn = rnn_model.predict(img_rnn, verbose=0)

    cnn1_digit = np.argmax(pred_cnn1[0])
    cnn2_digit = np.argmax(pred_cnn2[0])
    rnn_digit = np.argmax(pred_rnn[0])

    final_pred = (pred_cnn1 + pred_cnn2 + pred_rnn) / 3
    digit_class = np.argmax(final_pred[0])

    results[method_name] = {
    'image': img_display,
    'pred_cnn1': cnn1_digit,
    'pred_cnn2': cnn2_digit,
    'pred_rnn': rnn_digit,
    'pred_ensemble': digit_class,
    'confidence': final_pred[0][digit_class],
    'all_probs': final_pred[0]
    }

    plt.figure(figsize=(16, 12))

    plt.subplot(3, 3, 1)
    plt.imshow(original_img, cmap='gray')
    plt.title('Original Image')
    plt.axis('off')
```

```
for i, (method_name, result) in enumerate(results.items()):
plt.subplot(3, 3, i+2)
plt.imshow(result['image'], cmap='gray')
plt.title(f'{method_name.capitalize()} Preprocessing')
plt.axis('off')


plt.subplot(3, 3, i+4)
probs = result['all_probs']
bars = plt.bar(range(10), probs)
plt.xticks(range(10))
plt.title(f'{method_name.capitalize()} Probabilities')
plt.xlabel('Digit')
plt.ylabel('Probability')


bars[result['pred_ensemble']].set_color('red')


plt.subplot(3, 3, i+6)
pred_text = (
f"CNN1: {result['pred_cnn1']}\n"
f"CNN2: {result['pred_cnn2']}\n"
f"RNN: {result['pred_rnn']}\n"
f"Ensemble: {result['pred_ensemble']} (Conf: {result['confidence']:.2f})"
)
plt.text(0.1, 0.5, pred_text, fontsize=12)
plt.title(f'{method_name.capitalize()} Predictions')
plt.axis('off')

plt.tight_layout()
plt.savefig('prediction_results.png')
plt.show()
print("\n" + "="*50)
print(" PREDICTION RESULTS")
print("="*50)
for method_name, result in results.items():
print(f"\n {method_name.upper()} PREPROCESSING:")
```

```
        print(f"  CNN Model 1 predicts: {result['pred_cnn1']}")
        print(f"  CNN Model 2 predicts: {result['pred_cnn2']}")
        print(f"  RNN Model predicts: {result['pred_rnn']}")
        print(f"  → Ensemble predicts: {result['pred_ensemble']} with
{result['confidence']*100:.2f}% confidence")

        print("\n" + "="*50)
        print(" RECOMMENDATION:")
        standard_conf = results['standard']['confidence']
        advanced_conf = results['advanced']['confidence']

        if standard_conf > advanced_conf:
        best_method = 'standard'
        best_confidence = standard_conf
        else:
        best_method = 'advanced'
        best_confidence = advanced_conf

        best_prediction = results[best_method]['pred_ensemble']

        print(f"The most likely digit is {best_prediction} (using {best_method}
preprocessing)")
        print(f"Confidence: {best_confidence*100:.2f}%")
        print("="*50)

        return results
def tta_predict_custom(image_path, num_augments=10):
        preprocessing_methods, _ = improved_image_preprocessing(image_path)

        if preprocessing_methods is None:
        return None

        results = {}

        for method_name, (img_cnn, img_rnn, _) in preprocessing_methods.items():
        # Create augmentation generator
        aug_gen = ImageDataGenerator(
```

```
rotation_range=15,
zoom_range=0.2,
width_shift_range=0.2,
height_shift_range=0.2,
shear_range=0.2,
fill_mode='nearest'
)
cnn1_preds = []
cnn2_preds = []

it = aug_gen.flow(img_cnn, batch_size=1)

for i in range(num_augments):
batch = next(it)
cnn1_preds.append(cnn_model1.predict(batch, verbose=0))
cnn2_preds.append(cnn_model2.predict(batch, verbose=0))

avg_cnn1 = np.mean(cnn1_preds, axis=0)
avg_cnn2 = np.mean(cnn2_preds, axis=0)



rnn_pred = rnn_model.predict(img_rnn, verbose=0)
final_pred = (avg_cnn1 + avg_cnn2 + rnn_pred) / 3
digit_class = np.argmax(final_pred[0])

results[method_name] = {
'pred_ensemble': digit_class,
'confidence': final_pred[0][digit_class],
'all_probs': final_pred[0]
}
print("\n" + "="*50)
print(" TEST-TIME AUGMENTATION RESULTS")
print("="*50)

for method_name, result in results.items():
print(f"\n{method_name.upper()} PREPROCESSING + TTA:")
```

```
        print(f"  → Ensemble predicts: {result['pred_ensemble']} with
{result['confidence']*100:.2f}% confidence")
        top_indices = np.argsort(result['all_probs'])[-3:][::-1]
        print("  Top 3 candidates:")
        for i, idx in enumerate(top_indices):
        print(f"    {i+1}. Digit {idx}: {result['all_probs'][idx]*100:.2f}%")

        print("="*50)

        return results
def show_mnist_references():
        plt.figure(figsize=(12, 5))
        plt.suptitle("MNIST Reference Examples", fontsize=14)

        for i in range(10):
        # Find examples of digit i
        indices = np.where(y_train == i)[0]
        for j in range(min(5, len(indices))):
        plt.subplot(5, 10, j*10 + i + 1)
        plt.imshow(x_train[indices[j]].squeeze(), cmap='gray')
        plt.axis('off')

        if j == 0:
                plt.title(f"Digit {i}")

        plt.tight_layout()
        plt.subplots_adjust(top=0.9)
        plt.savefig('mnist_references.png')
        plt.show()
def predict_custom_image(image_path):
        print(f"\n Processing image: {image_path}")
        show_mnist_references()
        regular_results = predict_with_visualization(image_path)
        tta_results = tta_predict_custom(image_path)
        if regular_results and tta_results:
        standard_regular = regular_results['standard']['confidence']
        standard_tta = tta_results['standard']['confidence']
```

```
advanced_regular = regular_results['advanced']['confidence']
advanced_tta = tta_results['advanced']['confidence']
confidences = {
'standard_regular': standard_regular,
'standard_tta': standard_tta,
'advanced_regular': advanced_regular,
'advanced_tta': advanced_tta
}

best_method = max(confidences, key=confidences.get)

if 'standard' in best_method:
preproc = 'standard'
else:
preproc = 'advanced'

if 'tta' in best_method:
pred_method = 'tta'
prediction = tta_results[preproc]['pred_ensemble']
confidence = tta_results[preproc]['confidence']
else:
pred_method = 'regular'
prediction = regular_results[preproc]['pred_ensemble']
confidence = regular_results[preproc]['confidence']

print("\n" + "="*50)
print(" FINAL VERDICT")
print("="*50)
print(f"Best method: {preproc} preprocessing with {pred_method}")
print(f"Final prediction: {prediction}")
print(f"Confidence: {confidence*100:.2f}%")
print("="*50)

return prediction, confidence
else:
print("Failed to process image")
return None, None
```

```
if __name__ == "__main__":
        # Replace with your image path
        image_path = "/content/1.png"

        prediction, confidence = predict_custom_image(image_path)

        if prediction is not None:
        print(f"\n The image at {image_path} is most likely a {prediction} with
{confidence*100:.2f}% confidence")
```