!pip install numpy pandas tensorflow scikit-learn wfdb tqdm seaborn matplotlib

import numpy as np

import pandas as pd

import tensorflow as tf

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Conv1D, MaxPooling1D, BatchNormalization, Dropout, Flatten, Dense, Activation

from tensorflow.keras.optimizers import Adam

from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau

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

from sklearn.preprocessing import StandardScaler

from sklearn.svm import SVC

from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier

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

import wfdb

import os

from tqdm import tqdm

import matplotlib.pyplot as plt

def prepare\_data(data\_path='mit-bih'):

    if not os.path.exists(data\_path):

        print("Downloading MIT-BIH dataset...")

        wfdb.dl\_database('mitdb', data\_path)

    signals = []

    labels = []

    # Using selected records for balanced dataset

    selected\_records = ['100', '101', '103', '105', '106', '108', '109',

                       '111', '112', '113', '114', '115', '116', '117',

                       '119', '121', '122', '123', '124', '200', '201',

                       '202', '203', '205', '207', '208', '209', '210']

    class\_counts = {'N': 0, 'L': 0, 'R': 0, 'A': 0, 'V': 0}

    max\_samples\_per\_class = 1000  # Limit samples per class

    for record in tqdm(selected\_records):

        try:

            signal, \_ = wfdb.rdsamp(f'{data\_path}/{record}')

            annotation = wfdb.rdann(f'{data\_path}/{record}', 'atr')

            # Extract beats with balanced sampling

            for beat\_idx in range(len(annotation.symbol)):

                beat\_type = annotation.symbol[beat\_idx]

                if beat\_type in class\_counts and class\_counts[beat\_type] < max\_samples\_per\_class:

                    start\_idx = annotation.sample[beat\_idx] - 90

                    end\_idx = annotation.sample[beat\_idx] + 90

                    if start\_idx >= 0 and end\_idx < len(signal):

                        beat = signal[start\_idx:end\_idx, 0]

                        signals.append(beat)

                        labels.append(beat\_type)

                        class\_counts[beat\_type] += 1

        except Exception as e:

            print(f"Error processing record {record}: {str(e)}")

            continue

    print("\nClass distribution:")

    for class\_label, count in class\_counts.items():

        print(f"{class\_label}: {count}")

    return np.array(signals), np.array(labels)

# Prepare data

signals, labels = prepare\_data()

Output: 100%|██████████| 28/28 [00:05<00:00, 5.25it/s]

Class distribution:

N: 1000

L: 1000

R: 1000

A: 648

V: 1000

def build\_model(input\_shape=(180, 1)):

    model = Sequential([

        # First Conv Block

        Conv1D(32, kernel\_size=5, padding='same', input\_shape=input\_shape),

        BatchNormalization(),

        Activation('relu'),

        MaxPooling1D(pool\_size=2),

        Dropout(0.25),

        # Second Conv Block

        Conv1D(64, kernel\_size=5, padding='same'),

        BatchNormalization(),

        Activation('relu'),

        MaxPooling1D(pool\_size=2),

        Dropout(0.25),

        # Third Conv Block

        Conv1D(128, kernel\_size=5, padding='same'),

        BatchNormalization(),

        Activation('relu'),

        MaxPooling1D(pool\_size=2),

        Dropout(0.25),

        # Dense Layers

        Flatten(),

        Dense(128),

        BatchNormalization(),

        Activation('relu'),

        Dropout(0.4),

        Dense(5, activation='softmax')

    ])

    optimizer = Adam(learning\_rate=0.0005)

    model.compile(optimizer=optimizer,

                 loss='categorical\_crossentropy',

                 metrics=['accuracy'])

    return model

# Preprocess signals

scaler = StandardScaler()

signals\_reshaped = signals.reshape(signals.shape[0], -1)

signals\_scaled = scaler.fit\_transform(signals\_reshaped)

signals\_final = signals\_scaled.reshape(-1, 180, 1)

# Convert labels

label\_map = {'N': 0, 'L': 1, 'R': 2, 'A': 3, 'V': 4}

numerical\_labels = np.array([label\_map[label] for label in labels])

labels\_onehot = tf.keras.utils.to\_categorical(numerical\_labels)

# Split data with stratification

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

    signals\_final, labels\_onehot,

    test\_size=0.2,

    random\_state=42,

    stratify=numerical\_labels

)

# Build model

model = build\_model()

# Define callbacks

callbacks = [

    EarlyStopping(

        monitor='val\_accuracy',

        patience=5,

        mode='max',

        restore\_best\_weights=True

    ),

    ReduceLROnPlateau(

        monitor='val\_loss',

        factor=0.2,

        patience=3,

        min\_lr=1e-6

    )

]

# Train model

history = model.fit(

    X\_train, y\_train,

    validation\_data=(X\_test, y\_test),

    epochs=25,

    batch\_size=32,

    callbacks=callbacks,

    verbose=1

)

Output:

/usr/local/lib/python3.10/dist-packages/keras/src/layers/convolutional/base\_conv.py:107: UserWarning: Do not pass an `input\_shape`/`input\_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.

super().\_\_init\_\_(activity\_regularizer=activity\_regularizer, \*\*kwargs)

Epoch 1/25

**117/117** ━━━━━━━━━━━━━━━━━━━━ **9s** 51ms/step - accuracy: 0.8226 - loss: 0.5445 - val\_accuracy: 0.4634 - val\_loss: 1.2129 - learning\_rate: 5.0000e-04

Epoch 2/25

**117/117** ━━━━━━━━━━━━━━━━━━━━ **9s** 41ms/step - accuracy: 0.9604 - loss: 0.1466 - val\_accuracy: 0.7559 - val\_loss: 0.6381 - learning\_rate: 5.0000e-04

Epoch 3/25

**117/117** ━━━━━━━━━━━━━━━━━━━━ **7s** 58ms/step - accuracy: 0.9714 - loss: 0.1054 - val\_accuracy: 0.9151 - val\_loss: 0.2561 - learning\_rate: 5.0000e-04

Epoch 4/25

**117/117** ━━━━━━━━━━━━━━━━━━━━ **5s** 40ms/step - accuracy: 0.9769 - loss: 0.0895 - val\_accuracy: 0.9333 - val\_loss: 0.1894 - learning\_rate: 5.0000e-04

Epoch 5/25

**117/117** ━━━━━━━━━━━━━━━━━━━━ **5s** 40ms/step - accuracy: 0.9771 - loss: 0.0830 - val\_accuracy: 0.9720 - val\_loss: 0.0904 - learning\_rate: 5.0000e-04

Epoch 6/25

**117/117** ━━━━━━━━━━━━━━━━━━━━ **7s** 57ms/step - accuracy: 0.9763 - loss: 0.0699 - val\_accuracy: 0.9828 - val\_loss: 0.0653 - learning\_rate: 5.0000e-04

Epoch 7/25

**117/117** ━━━━━━━━━━━━━━━━━━━━ **8s** 40ms/step - accuracy: 0.9834 - loss: 0.0550 - val\_accuracy: 0.9871 - val\_loss: 0.0410 - learning\_rate: 5.0000e-04

Epoch 8/25

**117/117** ━━━━━━━━━━━━━━━━━━━━ **7s** 58ms/step - accuracy: 0.9825 - loss: 0.0578 - val\_accuracy: 0.9839 - val\_loss: 0.0503 - learning\_rate: 5.0000e-04

Epoch 9/25

**117/117** ━━━━━━━━━━━━━━━━━━━━ **5s** 40ms/step - accuracy: 0.9855 - loss: 0.0601 - val\_accuracy: 0.9860 - val\_loss: 0.0453 - learning\_rate: 5.0000e-04

Epoch 10/25

**117/117** ━━━━━━━━━━━━━━━━━━━━ **7s** 52ms/step - accuracy: 0.9819 - loss: 0.0588 - val\_accuracy: 0.9882 - val\_loss: 0.0361 - learning\_rate: 5.0000e-04

Epoch 11/25

**117/117** ━━━━━━━━━━━━━━━━━━━━ **9s** 40ms/step - accuracy: 0.9876 - loss: 0.0450 - val\_accuracy: 0.9903 - val\_loss: 0.0338 - learning\_rate: 5.0000e-04

Epoch 12/25

**117/117** ━━━━━━━━━━━━━━━━━━━━ **7s** 58ms/step - accuracy: 0.9809 - loss: 0.0568 - val\_accuracy: 0.9882 - val\_loss: 0.0362 - learning\_rate: 5.0000e-04

Epoch 13/25

**117/117** ━━━━━━━━━━━━━━━━━━━━ **9s** 43ms/step - accuracy: 0.9809 - loss: 0.0566 - val\_accuracy: 0.9903 - val\_loss: 0.0308 - learning\_rate: 5.0000e-04

Epoch 14/25

**117/117** ━━━━━━━━━━━━━━━━━━━━ **7s** 60ms/step - accuracy: 0.9851 - loss: 0.0504 - val\_accuracy: 0.9892 - val\_loss: 0.0348 - learning\_rate: 5.0000e-04

Epoch 15/25

**117/117** ━━━━━━━━━━━━━━━━━━━━ **5s** 41ms/step - accuracy: 0.9921 - loss: 0.0330 - val\_accuracy: 0.9914 - val\_loss: 0.0268 - learning\_rate: 5.0000e-04

Epoch 16/25

**117/117** ━━━━━━━━━━━━━━━━━━━━ **5s** 44ms/step - accuracy: 0.9884 - loss: 0.0378 - val\_accuracy: 0.9892 - val\_loss: 0.0293 - learning\_rate: 5.0000e-04

Epoch 17/25

**117/117** ━━━━━━━━━━━━━━━━━━━━ **10s** 40ms/step - accuracy: 0.9941 - loss: 0.0282 - val\_accuracy: 0.9925 - val\_loss: 0.0307 - learning\_rate: 5.0000e-04

Epoch 18/25

**117/117** ━━━━━━━━━━━━━━━━━━━━ **7s** 58ms/step - accuracy: 0.9878 - loss: 0.0414 - val\_accuracy: 0.9914 - val\_loss: 0.0246 - learning\_rate: 5.0000e-04

Epoch 19/25

**117/117** ━━━━━━━━━━━━━━━━━━━━ **8s** 40ms/step - accuracy: 0.9902 - loss: 0.0325 - val\_accuracy: 0.9925 - val\_loss: 0.0259 - learning\_rate: 5.0000e-04

Epoch 20/25

**117/117** ━━━━━━━━━━━━━━━━━━━━ **7s** 56ms/step - accuracy: 0.9911 - loss: 0.0281 - val\_accuracy: 0.9914 - val\_loss: 0.0281 - learning\_rate: 5.0000e-04

Epoch 21/25

**117/117** ━━━━━━━━━━━━━━━━━━━━ **8s** 40ms/step - accuracy: 0.9941 - loss: 0.0227 - val\_accuracy: 0.9903 - val\_loss: 0.0281 - learning\_rate: 5.0000e-04

Epoch 22/25

**117/117** ━━━━━━━━━━━━━━━━━━━━ **7s** 59ms/step - accuracy: 0.9933 - loss: 0.0314 - val\_accuracy: 0.9914 - val\_loss: 0.0245 - learning\_rate: 1.0000e-04

# Evaluate CNN model

test\_loss, test\_accuracy = model.evaluate(X\_test, y\_test, verbose=0)

print(f"\nCNN Test Accuracy: {test\_accuracy\*100:.2f}%")

# Compare with other algorithms

def compare\_algorithms(X\_train, X\_test, y\_train, y\_test):

    X\_train\_2d = X\_train.reshape(X\_train.shape[0], -1)

    X\_test\_2d = X\_test.reshape(X\_test.shape[0], -1)

    y\_train\_classes = np.argmax(y\_train, axis=1)

    y\_test\_classes = np.argmax(y\_test, axis=1)

    # Get unique classes in the test set

    unique\_classes = np.unique(y\_test\_classes)

    class\_names = ['Normal', 'LBBB', 'RBBB', 'APC', 'PVC']

    present\_classes = [class\_names[i] for i in unique\_classes]

    results = {

        'CNN (Proposed)': test\_accuracy \* 100,

        'Random Forest': 0,

        'Gradient Boosting': 0,

        'SVM': 0

    }

    # Random Forest

    rf = RandomForestClassifier(n\_estimators=100, random\_state=42)

    rf.fit(X\_train\_2d, y\_train\_classes)

    results['Random Forest'] = rf.score(X\_test\_2d, y\_test\_classes) \* 100

    # Gradient Boosting

    gb = GradientBoostingClassifier(n\_estimators=100, random\_state=42)

    gb.fit(X\_train\_2d, y\_train\_classes)

    results['Gradient Boosting'] = gb.score(X\_test\_2d, y\_test\_classes) \* 100

    # SVM

    svm = SVC(kernel='rbf', random\_state=42)

    svm.fit(X\_train\_2d, y\_train\_classes)

    results['SVM'] = svm.score(X\_test\_2d, y\_test\_classes) \* 100

    return results, present\_classes

results, present\_classes = compare\_algorithms(X\_train, X\_test, y\_train, y\_test)

# Print results

print("\nModel Comparison:")

for model\_name, accuracy in results.items():

    print(f"{model\_name}: {accuracy:.2f}%")

Output: CNN Test Accuracy: 99.25%

Model Comparison:

CNN (Proposed): 99.25%

Random Forest: 98.60%

Gradient Boosting: 98.17%

SVM: 98.60%

# Plot training history

plt.figure(figsize=(12, 4))

plt.subplot(1, 2, 1)

plt.plot(history.history['accuracy'])

plt.plot(history.history['val\_accuracy'])

plt.title('Model Accuracy')

plt.ylabel('Accuracy')

plt.xlabel('Epoch')

plt.legend(['Train', 'Validation'])

plt.subplot(1, 2, 2)

plt.plot(history.history['loss'])

plt.plot(history.history['val\_loss'])

plt.title('Model Loss')

plt.ylabel('Loss')

plt.xlabel('Epoch')

plt.legend(['Train', 'Validation'])

plt.tight\_layout()

plt.show()

# Plot comparison results

plt.figure(figsize=(8, 5))

plt.bar(results.keys(), results.values())

plt.title('Algorithm Comparison')

plt.xlabel('Algorithm')

plt.ylabel('Accuracy (%)')

plt.xticks(rotation=45)

plt.tight\_layout()

plt.show()

# Print classification report with only present classes

y\_pred = model.predict(X\_test)

print("\nClassification Report:")

print(classification\_report(

    np.argmax(y\_test, axis=1),

    np.argmax(y\_pred, axis=1),

    target\_names=present\_classes

))

# Print confusion matrix

cm = confusion\_matrix(np.argmax(y\_test, axis=1), np.argmax(y\_pred, axis=1))

plt.figure(figsize=(8, 6))

sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',

            xticklabels=present\_classes,

            yticklabels=present\_classes)

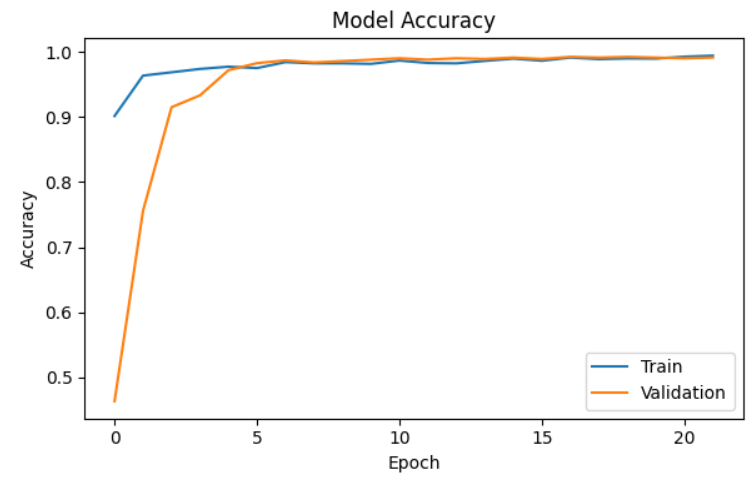
plt.title('Confusion Matrix')

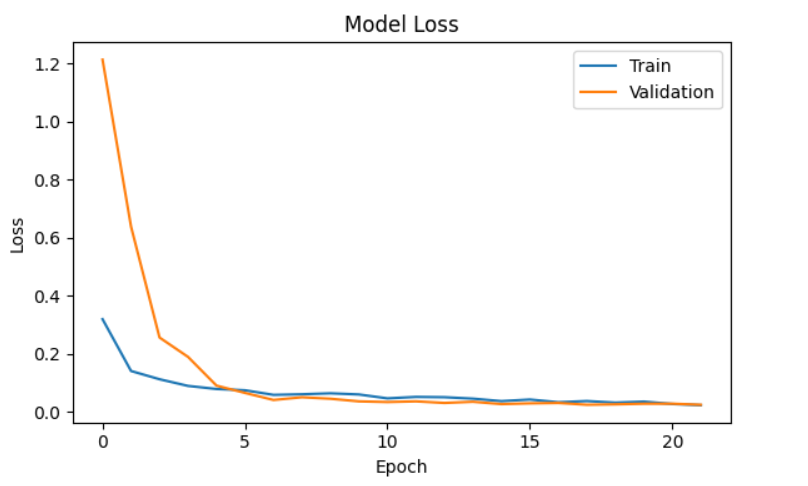
plt.ylabel('True Label')

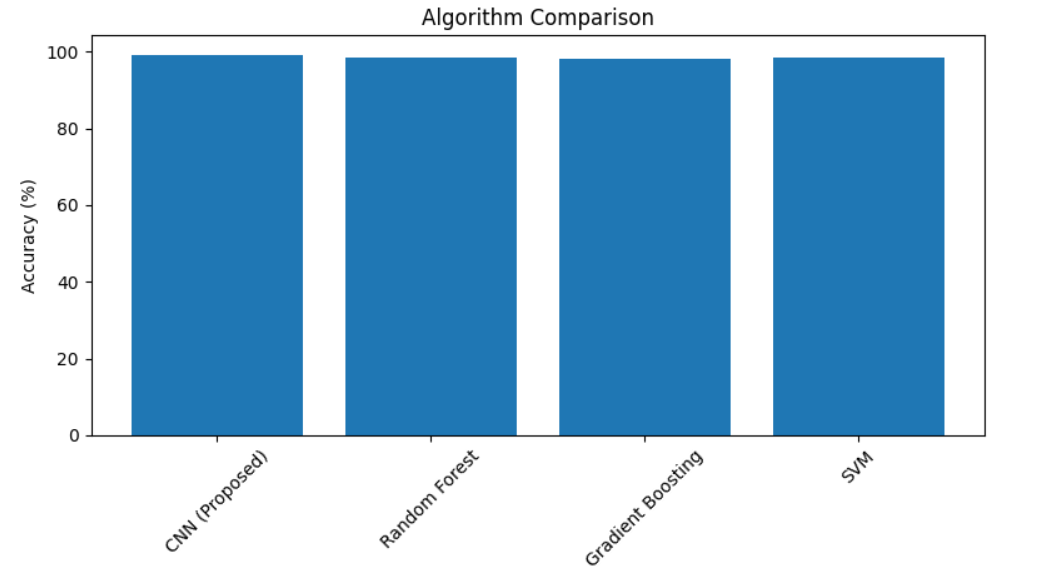
plt.xlabel('Predicted Label')

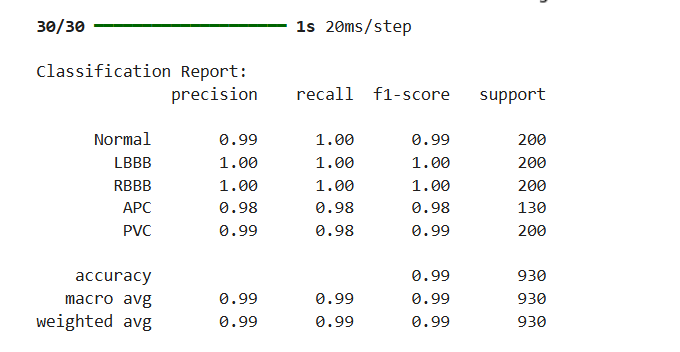
plt.tight\_layout()

plt.show()









# Additional performance metrics and visualizations

import seaborn as sns

from sklearn.metrics import precision\_recall\_curve, roc\_curve, auc

from scipy import stats

# First define our class variables

y\_test\_classes = np.argmax(y\_test, axis=1)

y\_pred = model.predict(X\_test)

y\_pred\_classes = np.argmax(y\_pred, axis=1)

def plot\_additional\_metrics():

    # 1. ROC Curves

    plt.figure(figsize=(10, 8))

    colors = ['blue', 'green', 'red', 'purple', 'orange']

    for i, (label, color) in enumerate(zip(present\_classes, colors)):

        fpr, tpr, \_ = roc\_curve((y\_test\_classes == i).astype(int),

                               y\_pred[:, i])

        roc\_auc = auc(fpr, tpr)

        plt.plot(fpr, tpr, color=color, lw=2,

                label=f'{label} (AUC = {roc\_auc:.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('ROC Curves for Each Class')

    plt.legend(loc="lower right")

    plt.show()

    # 2. Learning Curves

    plt.figure(figsize=(15, 5))

    metrics = ['accuracy', 'loss']

    titles = ['Learning Curve - Accuracy', 'Learning Curve - Loss']

    for i, (metric, title) in enumerate(zip(metrics, titles)):

        plt.subplot(1, 2, i+1)

        plt.plot(history.history[metric], 'b-', label='Training')

        plt.plot(history.history[f'val\_{metric}'], 'r-', label='Validation')

        plt.title(title)

        plt.xlabel('Epoch')

        plt.ylabel(metric.capitalize())

        plt.legend()

    plt.tight\_layout()

    plt.show()

    # 3. Per-class Performance Metrics

    precision = []

    recall = []

    f1\_score = []

    for class\_idx, class\_name in enumerate(present\_classes):

        true\_class = (y\_test\_classes == class\_idx)

        pred\_class = (y\_pred\_classes == class\_idx)

        # Calculate metrics

        tp = np.sum(true\_class & pred\_class)

        fp = np.sum(~true\_class & pred\_class)

        fn = np.sum(true\_class & ~pred\_class)

        # Avoid division by zero

        class\_precision = tp / (tp + fp) if (tp + fp) > 0 else 0

        class\_recall = tp / (tp + fn) if (tp + fn) > 0 else 0

        class\_f1 = 2 \* (class\_precision \* class\_recall) / (class\_precision + class\_recall) if (class\_precision + class\_recall) > 0 else 0

        precision.append(class\_precision)

        recall.append(class\_recall)

        f1\_score.append(class\_f1)

    # Plot metrics

    plt.figure(figsize=(12, 6))

    x = np.arange(len(present\_classes))

    width = 0.25

    plt.bar(x - width, precision, width, label='Precision')

    plt.bar(x, recall, width, label='Recall')

    plt.bar(x + width, f1\_score, width, label='F1-Score')

    plt.xlabel('Classes')

    plt.ylabel('Score')

    plt.title('Per-class Performance Metrics')

    plt.xticks(x, present\_classes, rotation=45)

    plt.legend()

    plt.tight\_layout()

    plt.show()

    # 4. Confusion Matrix with Percentages

    cm = confusion\_matrix(y\_test\_classes, y\_pred\_classes)

    cm\_normalized = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]

    plt.figure(figsize=(10, 8))

    sns.heatmap(cm\_normalized, annot=True, fmt='.2%', cmap='Blues',

                xticklabels=present\_classes,

                yticklabels=present\_classes)

    plt.title('Normalized Confusion Matrix')

    plt.ylabel('True Label')

    plt.xlabel('Predicted Label')

    plt.tight\_layout()

    plt.show()

    return precision, recall, f1\_score

# Execute additional analysis

precision, recall, f1\_score = plot\_additional\_metrics()

# Print summary statistics

print("\nPer-class Performance Summary:")

print("-" \* 50)

for i, class\_name in enumerate(present\_classes):

    print(f"\n{class\_name}:")

    print(f"Precision: {precision[i]:.4f}")

    print(f"Recall: {recall[i]:.4f}")

    print(f"F1-Score: {f1\_score[i]:.4f}")

    print(f"Support: {np.sum(y\_test\_classes == i)}")

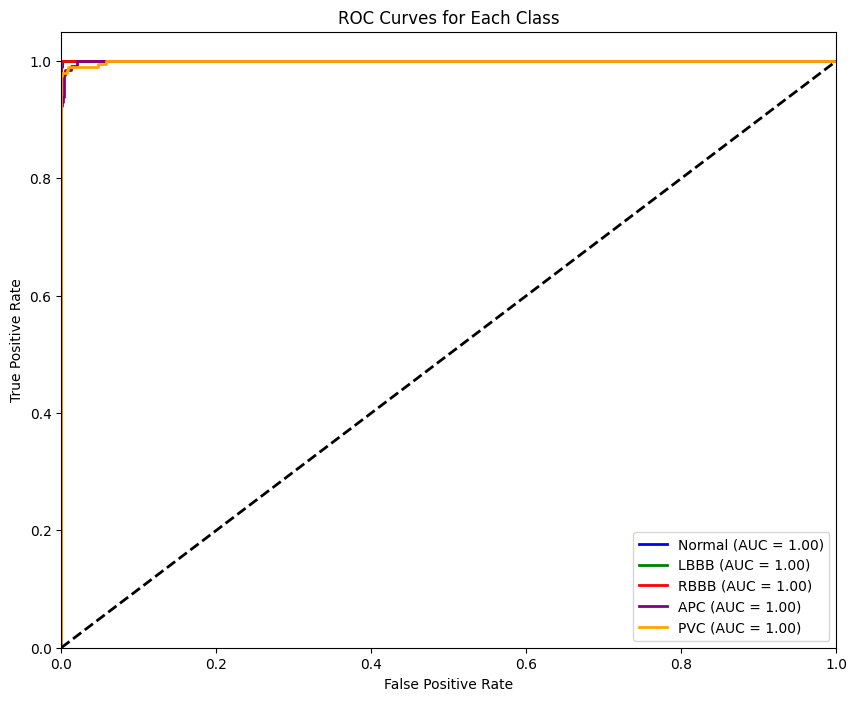
# Print overall accuracy

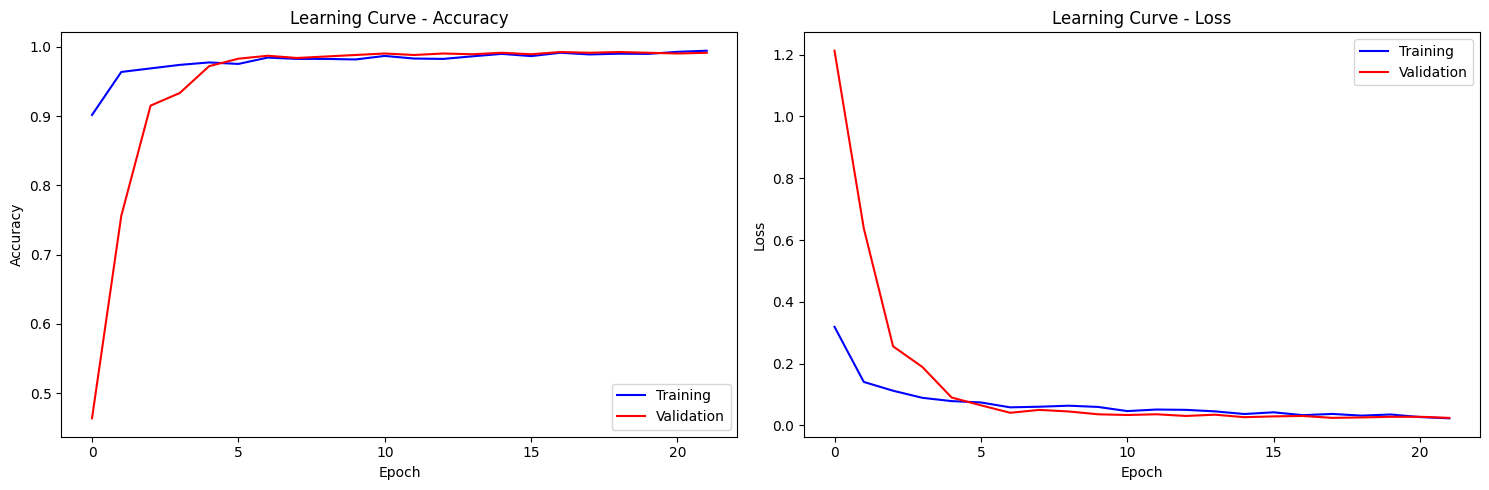
print("\nOverall Model Performance:")

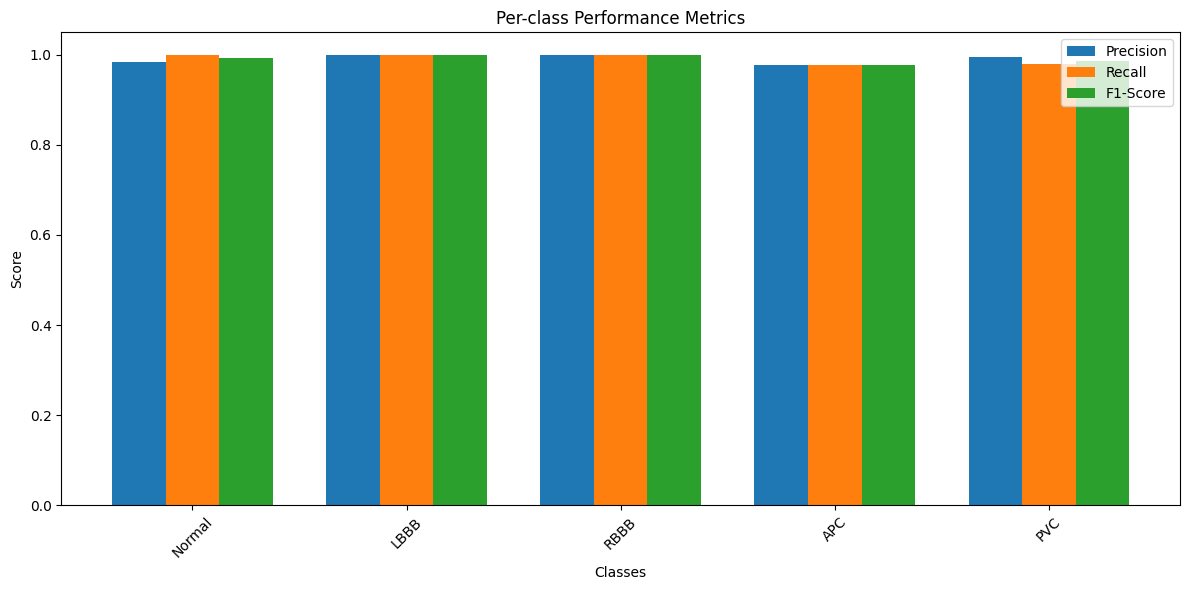
print("-" \* 50)

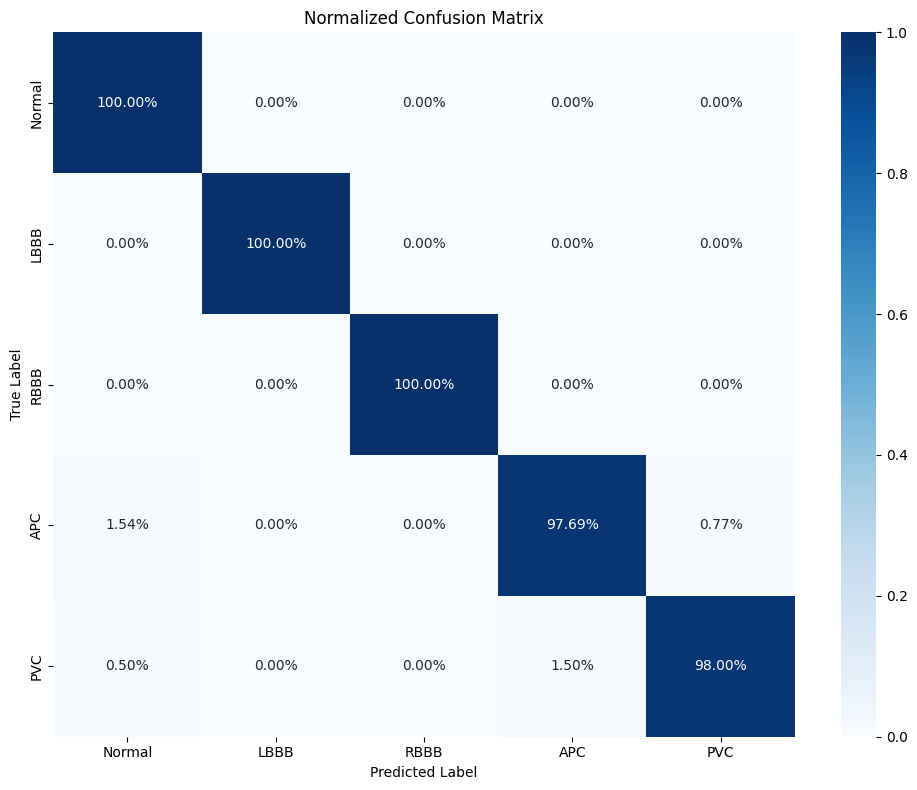
print(f"Accuracy: {np.mean(y\_test\_classes == y\_pred\_classes):.4f}")

print(f"Macro F1-Score: {np.mean(f1\_score):.4f}")









Per-class Performance Summary:

--------------------------------------------------

Normal:

Precision: 0.9852

Recall: 1.0000

F1-Score: 0.9926

Support: 200

LBBB:

Precision: 1.0000

Recall: 1.0000

F1-Score: 1.0000

Support: 200

RBBB:

Precision: 1.0000

Recall: 1.0000

F1-Score: 1.0000

Support: 200

APC:

Precision: 0.9769

Recall: 0.9769

F1-Score: 0.9769

Support: 130

PVC:

Precision: 0.9949

Recall: 0.9800

F1-Score: 0.9874

Support: 200

Overall Model Performance:

--------------------------------------------------

Accuracy: 0.9925

Macro F1-Score: 0.9914

# Model Analysis with Enhanced Error Handling

def analyze\_model():

    print("\nDetailed Performance Analysis:")

    print("-" \* 50)

    # 1. Class Distribution Analysis

    class\_distribution = pd.Series(y\_test\_classes).value\_counts()

    plt.figure(figsize=(10, 5))

    sns.barplot(x=class\_distribution.index.map(lambda x: present\_classes[x]),

                y=class\_distribution.values)

    plt.title('Test Set Class Distribution')

    plt.xlabel('Class')

    plt.ylabel('Number of Samples')

    plt.xticks(rotation=45)

    plt.tight\_layout()

    plt.show()

    # 2. Prediction Confidence Analysis

    predictions = model.predict(X\_test)

    prediction\_confidence = np.max(predictions, axis=1)

    plt.figure(figsize=(10, 5))

    for i, class\_name in enumerate(present\_classes):

        class\_confidences = prediction\_confidence[y\_test\_classes == i]

        sns.kdeplot(class\_confidences, label=class\_name)

    plt.title('Prediction Confidence Distribution by Class')

    plt.xlabel('Confidence Score')

    plt.ylabel('Density')

    plt.legend()

    plt.tight\_layout()

    plt.show()

    # 3. Error Analysis

    incorrect\_predictions = y\_test\_classes != y\_pred\_classes

    # Error distribution by class

    error\_by\_class = pd.DataFrame({

        'True Class': [present\_classes[i] for i in y\_test\_classes[incorrect\_predictions]],

        'Predicted Class': [present\_classes[i] for i in y\_pred\_classes[incorrect\_predictions]]

    })

    plt.figure(figsize=(12, 6))

    error\_matrix = pd.crosstab(error\_by\_class['True Class'],

                              error\_by\_class['Predicted Class'])

    sns.heatmap(error\_matrix, annot=True, fmt='d', cmap='YlOrRd')

    plt.title('Error Distribution Heatmap')

    plt.tight\_layout()

    plt.show()

    # 4. Performance Metrics Over Confidence Thresholds

    thresholds = np.linspace(0, 1, 20)

    threshold\_metrics = []

    for threshold in thresholds:

        high\_confidence = prediction\_confidence >= threshold

        if np.sum(high\_confidence) > 0:

            accuracy = np.mean(y\_test\_classes[high\_confidence] == y\_pred\_classes[high\_confidence])

            coverage = np.mean(high\_confidence)

            threshold\_metrics.append({

                'threshold': threshold,

                'accuracy': accuracy,

                'coverage': coverage

            })

    threshold\_df = pd.DataFrame(threshold\_metrics)

    plt.figure(figsize=(10, 5))

    plt.plot(threshold\_df['threshold'], threshold\_df['accuracy'],

             label='Accuracy', marker='o')

    plt.plot(threshold\_df['threshold'], threshold\_df['coverage'],

             label='Coverage', marker='o')

    plt.title('Accuracy vs Coverage Trade-off')

    plt.xlabel('Confidence Threshold')

    plt.ylabel('Score')

    plt.legend()

    plt.grid(True)

    plt.tight\_layout()

    plt.show()

    # 5. Print Summary Statistics

    print("\nPerformance Summary by Class:")

    for i, class\_name in enumerate(present\_classes):

        class\_mask = y\_test\_classes == i

        class\_acc = np.mean(y\_pred\_classes[class\_mask] == y\_test\_classes[class\_mask])

        class\_conf = np.mean(prediction\_confidence[class\_mask])

        print(f"\n{class\_name}:")

        print(f"Accuracy: {class\_acc:.4f}")

        print(f"Avg Confidence: {class\_conf:.4f}")

        print(f"Sample Count: {np.sum(class\_mask)}")

    # 6. High Confidence Errors

    high\_conf\_threshold = 0.9

    high\_conf\_errors = (y\_test\_classes != y\_pred\_classes) & (prediction\_confidence > high\_conf\_threshold)

    if np.sum(high\_conf\_errors) > 0:

        print(f"\nHigh Confidence Errors (confidence > {high\_conf\_threshold}):")

        print(f"Count: {np.sum(high\_conf\_errors)}")

        print("Distribution:")

        for i, class\_name in enumerate(present\_classes):

            count = np.sum(y\_test\_classes[high\_conf\_errors] == i)

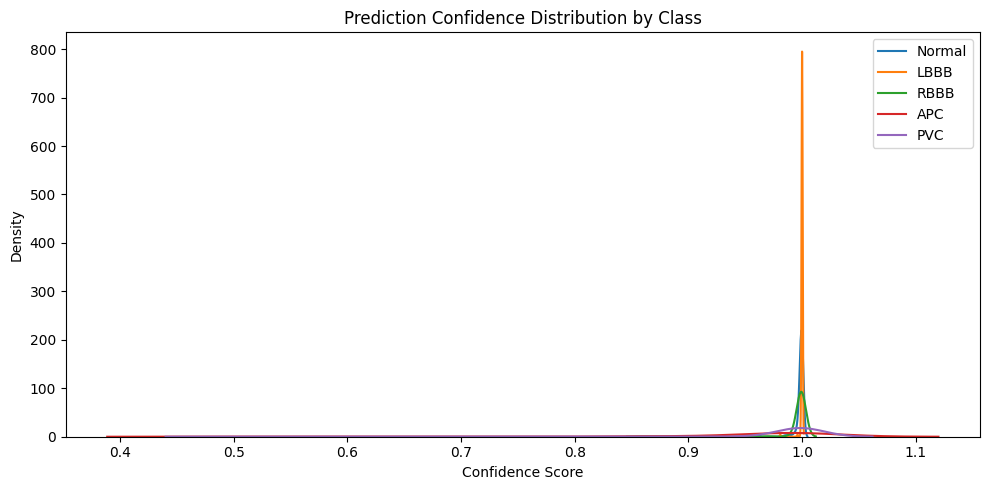
            if count > 0:

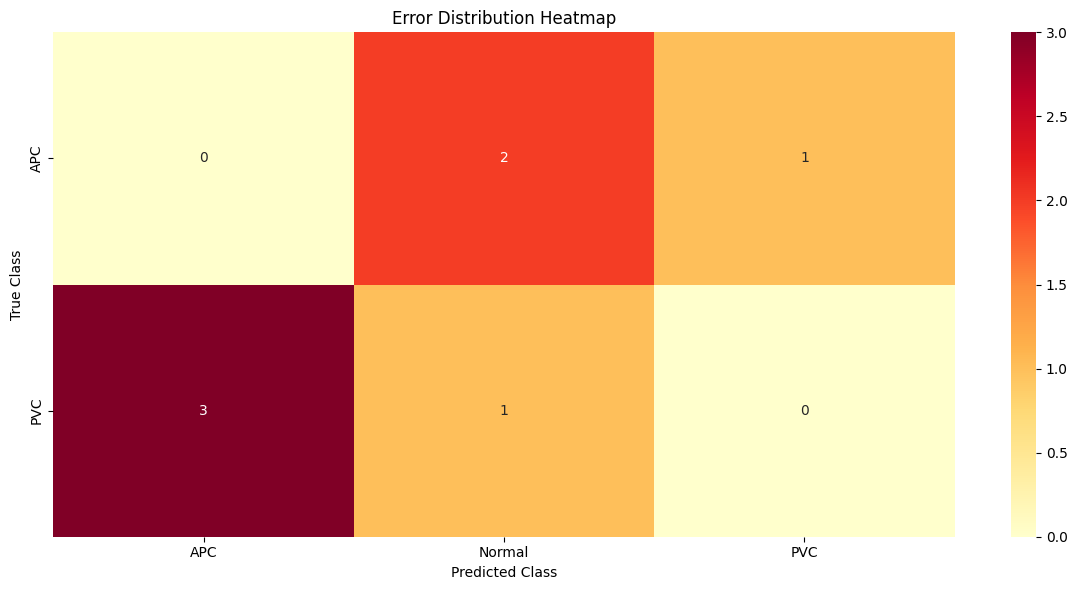
                print(f"{class\_name}: {count}")

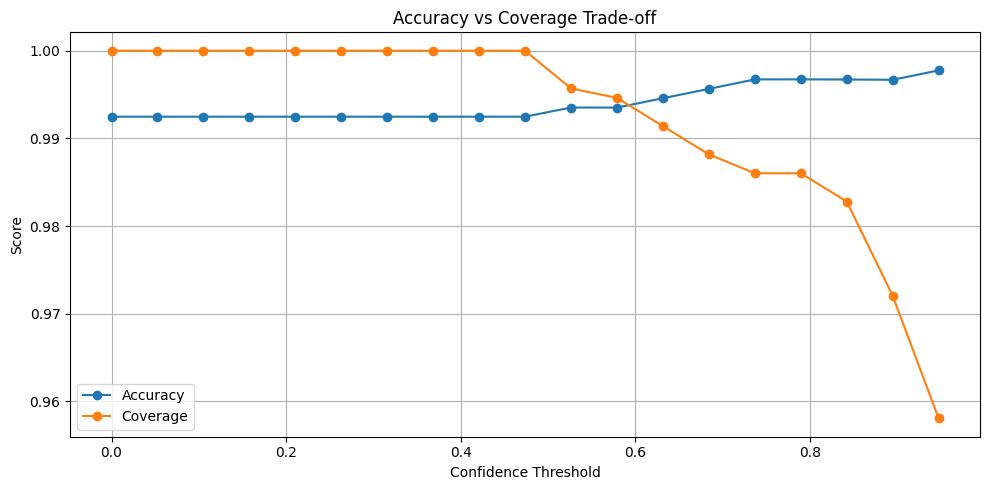
# Execute analysis

analyze\_model()









Performance Summary by Class:

Normal:

Accuracy: 1.0000

Avg Confidence: 0.9984

Sample Count: 200

LBBB:

Accuracy: 1.0000

Avg Confidence: 0.9998

Sample Count: 200

RBBB:

Accuracy: 1.0000

Avg Confidence: 0.9975

Sample Count: 200

APC:

Accuracy: 0.9769

Avg Confidence: 0.9514

Sample Count: 130

PVC:

Accuracy: 0.9800

Avg Confidence: 0.9867

Sample Count: 200

High Confidence Errors (confidence > 0.9):

Count: 3

Distribution:

APC: 2

PVC: 1

# Extended Analysis and Results

def plot\_extended\_results():

    # 1. Model Performance Over Time

    epochs = range(1, len(history.history['accuracy']) + 1)

    plt.figure(figsize=(15, 5))

    # Training Metrics

    plt.subplot(1, 3, 1)

    plt.plot(epochs, history.history['accuracy'], 'b-', label='Training Accuracy')

    plt.plot(epochs, history.history['val\_accuracy'], 'r-', label='Validation Accuracy')

    plt.title('Model Accuracy Over Time')

    plt.xlabel('Epoch')

    plt.ylabel('Accuracy')

    plt.legend()

    plt.grid(True)

    plt.subplot(1, 3, 2)

    plt.plot(epochs, history.history['loss'], 'b-', label='Training Loss')

    plt.plot(epochs, history.history['val\_loss'], 'r-', label='Validation Loss')

    plt.title('Model Loss Over Time')

    plt.xlabel('Epoch')

    plt.ylabel('Loss')

    plt.legend()

    plt.grid(True)

    # Accuracy vs Loss

    plt.subplot(1, 3, 3)

    plt.scatter(history.history['loss'], history.history['accuracy'],

                c='blue', label='Training')

    plt.scatter(history.history['val\_loss'], history.history['val\_accuracy'],

                c='red', label='Validation')

    plt.title('Accuracy vs Loss')

    plt.xlabel('Loss')

    plt.ylabel('Accuracy')

    plt.legend()

    plt.grid(True)

    plt.tight\_layout()

    plt.show()

    # 2. Per-Class Analysis

    predictions = model.predict(X\_test)

    y\_pred\_classes = np.argmax(predictions, axis=1)

    y\_test\_classes = np.argmax(y\_test, axis=1)

    # Confusion Matrix with Absolute Numbers and Percentages

    plt.figure(figsize=(12, 5))

    plt.subplot(1, 2, 1)

    cm = confusion\_matrix(y\_test\_classes, y\_pred\_classes)

    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',

                xticklabels=present\_classes,

                yticklabels=present\_classes)

    plt.title('Confusion Matrix (Absolute Numbers)')

    plt.ylabel('True Label')

    plt.xlabel('Predicted Label')

    plt.subplot(1, 2, 2)

    cm\_norm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]

    sns.heatmap(cm\_norm, annot=True, fmt='.2%', cmap='Blues',

                xticklabels=present\_classes,

                yticklabels=present\_classes)

    plt.title('Confusion Matrix (Normalized)')

    plt.ylabel('True Label')

    plt.xlabel('Predicted Label')

    plt.tight\_layout()

    plt.show()

    # 3. Confidence Distribution Analysis

    plt.figure(figsize=(15, 5))

    # Overall confidence distribution

    plt.subplot(1, 3, 1)

    confidence\_scores = np.max(predictions, axis=1)

    plt.hist(confidence\_scores, bins=30, alpha=0.7)

    plt.title('Overall Confidence Distribution')

    plt.xlabel('Confidence Score')

    plt.ylabel('Count')

    # Confidence for correct vs incorrect predictions

    plt.subplot(1, 3, 2)

    correct\_mask = y\_pred\_classes == y\_test\_classes

    plt.hist(confidence\_scores[correct\_mask], bins=30, alpha=0.7,

             label='Correct', color='green')

    plt.hist(confidence\_scores[~correct\_mask], bins=30, alpha=0.7,

             label='Incorrect', color='red')

    plt.title('Confidence Distribution: Correct vs Incorrect')

    plt.xlabel('Confidence Score')

    plt.ylabel('Count')

    plt.legend()

    # Per-class confidence

    plt.subplot(1, 3, 3)

    for i, class\_name in enumerate(present\_classes):

        class\_mask = y\_test\_classes == i

        plt.hist(confidence\_scores[class\_mask], bins=20, alpha=0.5,

                label=class\_name)

    plt.title('Confidence Distribution by Class')

    plt.xlabel('Confidence Score')

    plt.ylabel('Count')

    plt.legend()

    plt.tight\_layout()

    plt.show()

    # 4. Performance Metrics Summary

    from sklearn.metrics import classification\_report

    report = classification\_report(y\_test\_classes, y\_pred\_classes,

                                 target\_names=present\_classes,

                                 output\_dict=True)

    # Plot detailed metrics

    metrics\_df = pd.DataFrame(report).T

    metrics\_df = metrics\_df.drop('support', axis=1)

    plt.figure(figsize=(12, 6))

    metrics\_df.iloc[:-3].plot(kind='bar', width=0.8)

    plt.title('Detailed Performance Metrics by Class')

    plt.xlabel('Class')

    plt.ylabel('Score')

    plt.legend(title='Metrics')

    plt.grid(True, alpha=0.3)

    plt.tight\_layout()

    plt.show()

    # Print detailed statistics

    print("\nDetailed Performance Statistics:")

    print("-" \* 50)

    # Overall metrics

    print("\nOverall Performance:")

    print(f"Accuracy: {report['accuracy']:.4f}")

    print(f"Macro Avg F1-Score: {report['macro avg']['f1-score']:.4f}")

    print(f"Weighted Avg F1-Score: {report['weighted avg']['f1-score']:.4f}")

    # Per-class metrics

    print("\nPer-class Performance:")

    for class\_name in present\_classes:

        print(f"\n{class\_name}:")

        print(f"Precision: {report[class\_name]['precision']:.4f}")

        print(f"Recall: {report[class\_name]['recall']:.4f}")

        print(f"F1-Score: {report[class\_name]['f1-score']:.4f}")

        print(f"Support: {report[class\_name]['support']}")

    # Error Analysis

    print("\nError Analysis:")

    error\_mask = y\_pred\_classes != y\_test\_classes

    total\_errors = np.sum(error\_mask)

    print(f"Total errors: {total\_errors}")

    print(f"Error rate: {total\_errors/len(y\_test\_classes):.4f}")

    # High confidence errors

    high\_conf\_threshold = 0.9

    high\_conf\_errors = error\_mask & (confidence\_scores >= high\_conf\_threshold)

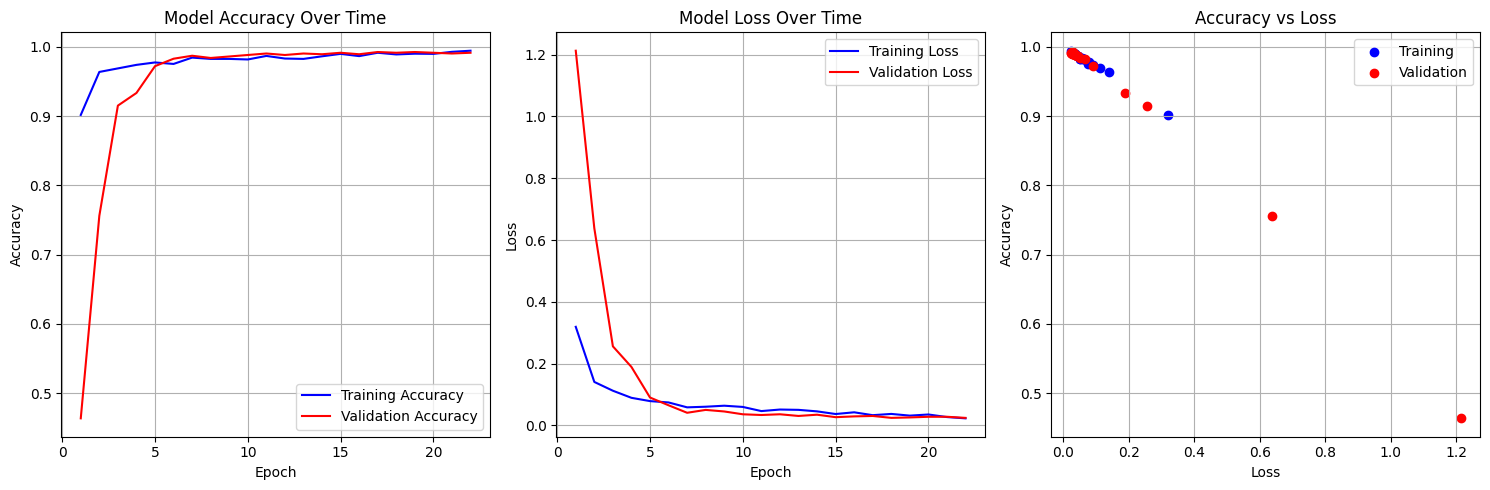
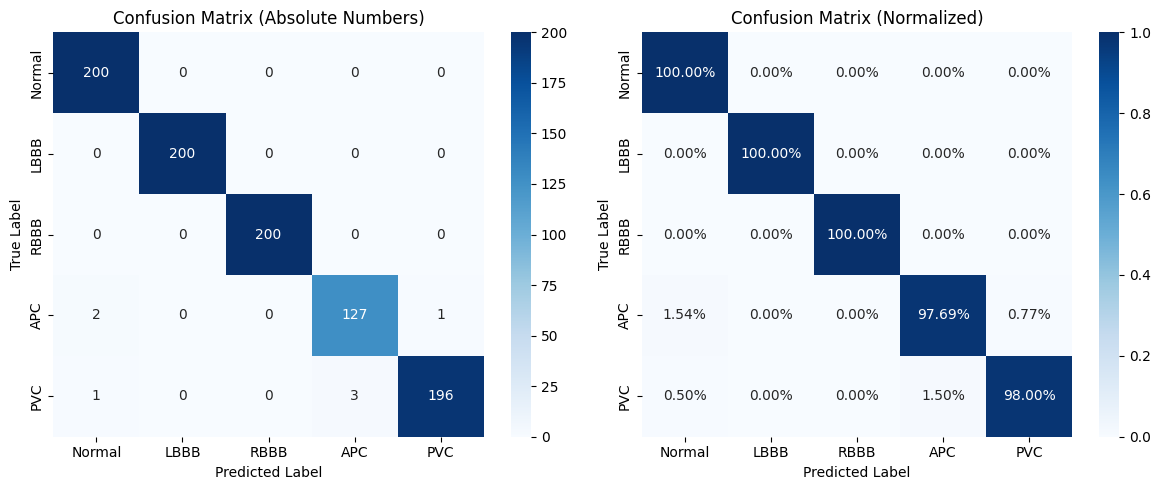
    print(f"\nHigh confidence errors (confidence >= {high\_conf\_threshold}):")

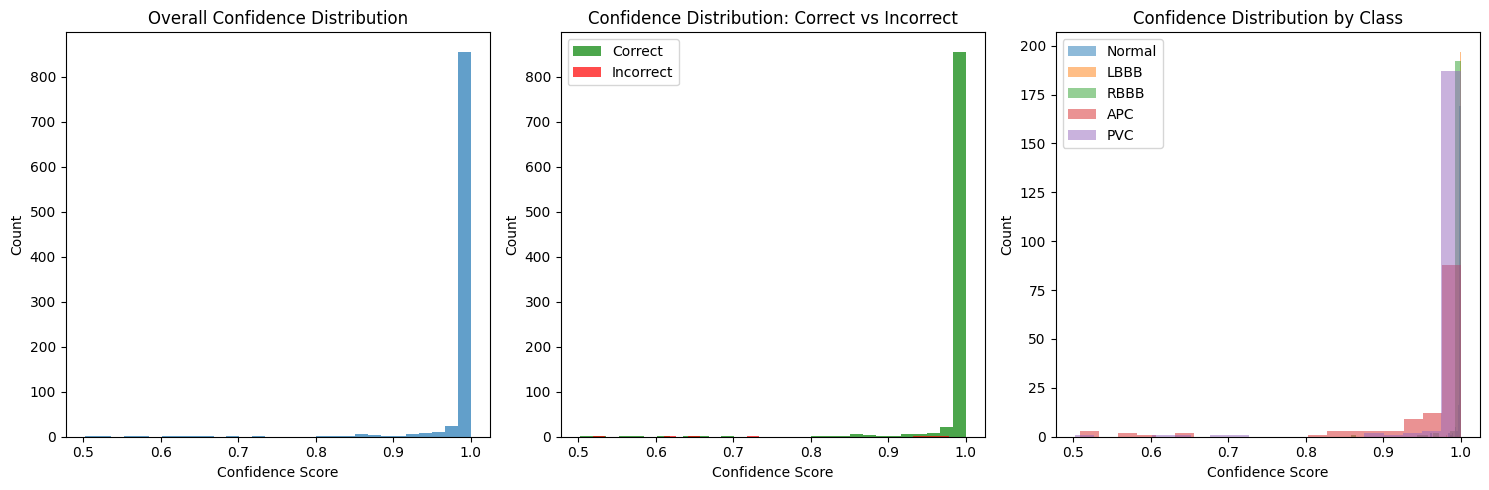
    print(f"Count: {np.sum(high\_conf\_errors)}")

    print(f"Percentage of all errors: {np.sum(high\_conf\_errors)/total\_errors:.4f}")

# Execute extended analysis

plot\_extended\_results()





Detailed Performance Statistics:

--------------------------------------------------

Overall Performance:

Accuracy: 0.9925

Macro Avg F1-Score: 0.9914

Weighted Avg F1-Score: 0.9925

Per-class Performance:

Normal:

Precision: 0.9852

Recall: 1.0000

F1-Score: 0.9926

Support: 200.0

LBBB:

Precision: 1.0000

Recall: 1.0000

F1-Score: 1.0000

Support: 200.0

RBBB:

Precision: 1.0000

Recall: 1.0000

F1-Score: 1.0000

Support: 200.0

APC:

Precision: 0.9769

Recall: 0.9769

F1-Score: 0.9769

Support: 130.0

PVC:

Precision: 0.9949

Recall: 0.9800

F1-Score: 0.9874

Support: 200.0

Error Analysis:

Total errors: 7

Error rate: 0.0075

High confidence errors (confidence >= 0.9):

Count: 3

Percentage of all errors: 0.4286

def analyze\_advanced\_metrics():

    # Calculate predictions and probabilities

    probabilities = model.predict(X\_test)

    predictions = np.argmax(probabilities, axis=1)

    true\_labels = np.argmax(y\_test, axis=1)

    # 1. Advanced ROC Analysis

    plt.figure(figsize=(15, 5))

    # ROC Curves for each class

    plt.subplot(1, 3, 1)

    for i, class\_name in enumerate(present\_classes):

        fpr, tpr, \_ = roc\_curve((true\_labels == i).astype(int), probabilities[:, i])

        roc\_auc = auc(fpr, tpr)

        plt.plot(fpr, tpr, label=f'{class\_name} (AUC = {roc\_auc:.2f})')

    plt.plot([0, 1], [0, 1], 'k--')

    plt.title('ROC Curves by Class')

    plt.xlabel('False Positive Rate')

    plt.ylabel('True Positive Rate')

    plt.legend(loc='lower right', bbox\_to\_anchor=(1.7, 0.5))

    # 2. Precision-Recall Curves

    plt.subplot(1, 3, 2)

    for i, class\_name in enumerate(present\_classes):

        precision, recall, \_ = precision\_recall\_curve((true\_labels == i).astype(int),

                                                    probabilities[:, i])

        plt.plot(recall, precision, label=f'{class\_name}')

    plt.title('Precision-Recall Curves')

    plt.xlabel('Recall')

    plt.ylabel('Precision')

    plt.legend(loc='lower right')

    # 3. Error Distribution

    plt.subplot(1, 3, 3)

    error\_mask = predictions != true\_labels

    plt.hist(np.max(probabilities[error\_mask], axis=1), bins=20,

             alpha=0.5, color='red', label='Errors')

    plt.hist(np.max(probabilities[~error\_mask], axis=1), bins=20,

             alpha=0.5, color='green', label='Correct')

    plt.title('Confidence Distribution')

    plt.xlabel('Prediction Confidence')

    plt.ylabel('Count')

    plt.legend()

    plt.tight\_layout()

    plt.show()

    # 4. Confusion Matrix with Percentages

    cm = confusion\_matrix(true\_labels, predictions)

    plt.figure(figsize=(10, 8))

    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',

                xticklabels=present\_classes,

                yticklabels=present\_classes)

    plt.title('Confusion Matrix')

    plt.ylabel('True Label')

    plt.xlabel('Predicted Label')

    plt.tight\_layout()

    plt.show()

    # 5. Per-class Statistics

    print("\nDetailed Class-wise Statistics:")

    print("-" \* 50)

    for i, class\_name in enumerate(present\_classes):

        class\_mask = true\_labels == i

        class\_preds = predictions[class\_mask]

        class\_probs = probabilities[class\_mask]

        print(f"\nClass: {class\_name}")

        print(f"Number of samples: {np.sum(class\_mask)}")

        print(f"Accuracy: {np.mean(class\_preds == true\_labels[class\_mask]):.4f}")

        print(f"Average confidence: {np.mean(np.max(class\_probs, axis=1)):.4f}")

        print(f"Misclassification rate: {np.mean(class\_preds != true\_labels[class\_mask]):.4f}")

    # 6. Performance at Different Confidence Thresholds

    thresholds = np.linspace(0.5, 1.0, 11)

    threshold\_metrics = []

    for threshold in thresholds:

        high\_conf\_mask = np.max(probabilities, axis=1) >= threshold

        if np.sum(high\_conf\_mask) > 0:

            acc = np.mean(predictions[high\_conf\_mask] == true\_labels[high\_conf\_mask])

            coverage = np.mean(high\_conf\_mask)

            threshold\_metrics.append({

                'threshold': threshold,

                'accuracy': acc,

                'coverage': coverage

            })

    threshold\_df = pd.DataFrame(threshold\_metrics)

    plt.figure(figsize=(10, 5))

    plt.plot(threshold\_df['threshold'], threshold\_df['accuracy'],

             marker='o', label='Accuracy')

    plt.plot(threshold\_df['threshold'], threshold\_df['coverage'],

             marker='o', label='Coverage')

    plt.title('Performance vs Confidence Threshold')

    plt.xlabel('Confidence Threshold')

    plt.ylabel('Score')

    plt.grid(True)

    plt.legend()

    plt.tight\_layout()

    plt.show()

    # 7. Error Analysis Summary

    print("\nError Analysis Summary:")

    print("-" \* 50)

    # Most common misclassifications

    error\_pairs = list(zip(true\_labels[predictions != true\_labels],

                          predictions[predictions != true\_labels]))

    error\_counts = pd.Series(error\_pairs).value\_counts()

    print("\nMost Common Misclassifications:")

    for (true, pred), count in error\_counts.head().items():

        print(f"{present\_classes[true]} misclassified as {present\_classes[pred]}: {count} times")

    # High confidence errors

    high\_conf\_threshold = 0.9

    high\_conf\_errors = (predictions != true\_labels) & (np.max(probabilities, axis=1) > high\_conf\_threshold)

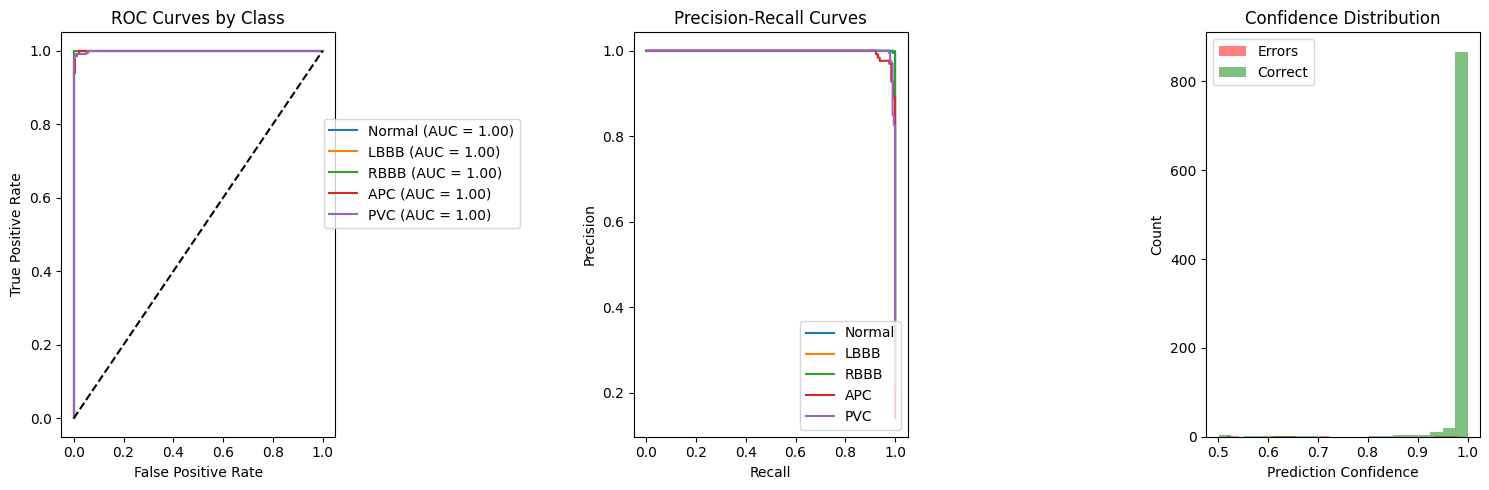
    print(f"\nHigh Confidence Errors (confidence > {high\_conf\_threshold}):")

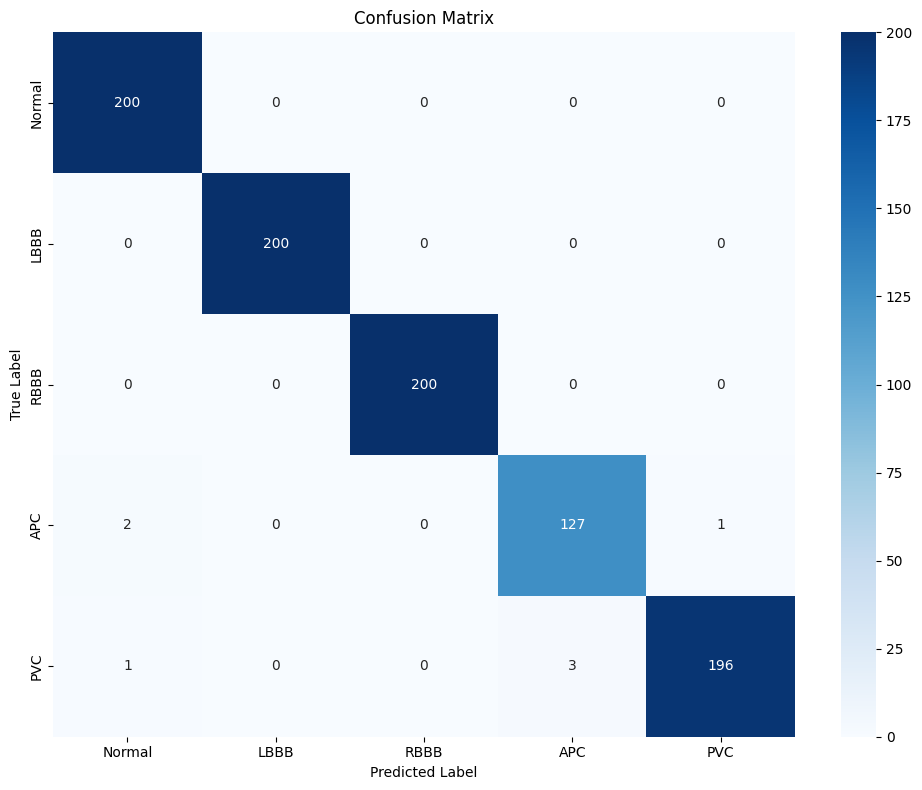
    print(f"Total count: {np.sum(high\_conf\_errors)}")

    print(f"Percentage of all errors: {np.sum(high\_conf\_errors) / np.sum(predictions != true\_labels):.2%}")

# Execute the advanced analysis

analyze\_advanced\_metrics()





Detailed Class-wise Statistics:

--------------------------------------------------

Class: Normal

Number of samples: 200

Accuracy: 1.0000

Average confidence: 0.9984

Misclassification rate: 0.0000

Class: LBBB

Number of samples: 200

Accuracy: 1.0000

Average confidence: 0.9998

Misclassification rate: 0.0000

Class: RBBB

Number of samples: 200

Accuracy: 1.0000

Average confidence: 0.9975

Misclassification rate: 0.0000

Class: APC

Number of samples: 130

Accuracy: 0.9769

Average confidence: 0.9514

Misclassification rate: 0.0231

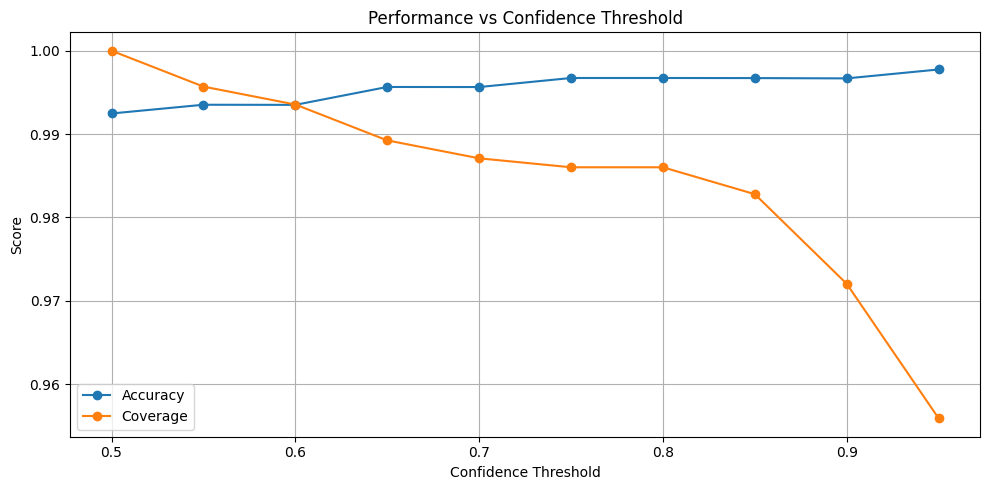
Class: PVC

Number of samples: 200

Accuracy: 0.9800

Average confidence: 0.9867

Misclassification rate: 0.0200



Error Analysis Summary:

--------------------------------------------------

Most Common Misclassifications:

PVC misclassified as APC: 3 times

APC misclassified as Normal: 2 times

PVC misclassified as Normal: 1 times

APC misclassified as PVC: 1 times

High Confidence Errors (confidence > 0.9):

Total count: 3

Percentage of all errors: 42.86%