```python
import os
import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
from tensorflow.keras.utils import to_categorical, plot_model
from tensorflow.keras.callbacks import EarlyStopping,
ReduceLROnPlateau, ModelCheckpoint
from sklearn.metrics import (
    classification_report,
    confusion_matrix,
    roc_curve,
    auc,
    f1_score,
    accuracy_score,
    precision_score,
    recall_score
)
from sklearn.preprocessing import label_binarize
import matplotlib.pyplot as plt
import seaborn as sns

# ======================================================================
# Helper Functions for Visualization
# ======================================================================

def save_classification_report_as_image(report_text,
filename="classification_report.png"):
    """Saves the classification report text as a PNG image."""
    plt.figure(figsize=(12, 12))
    plt.text(0.01, 1.0, report_text, {'fontsize': 12},
fontproperties='monospace')
    plt.axis('off')
    plt.savefig(filename, bbox_inches='tight', dpi=300)
    plt.close()

def plot_training_history(history, filename="training_history.png"):
    """Plots and saves the model's accuracy and loss history."""
    acc = history.history['accuracy']
    val_acc = history.history['val_accuracy']
    loss = history.history['loss']
    val_loss = history.history['val_loss']
    epochs = range(1, len(acc) + 1)

    plt.figure(figsize=(14, 6))

    # Accuracy plot
    plt.subplot(1, 2, 1)
```

```
    plt.plot(epochs, acc, 'b', label='Training Accuracy')
    plt.plot(epochs, val_acc, 'orange', label='Validation Accuracy')
    plt.title('Model Accuracy')
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy')
    plt.legend()

    # Loss plot
    plt.subplot(1, 2, 2)
    plt.plot(epochs, loss, 'b', label='Training Loss')
    plt.plot(epochs, val_loss, 'orange', label='Validation Loss')
    plt.title('Model Loss')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.legend()

    plt.tight_layout()
    plt.savefig(filename)
    plt.close()

def plot_roc_curves(y_true, y_probs, classes,
filename="roc_curve.png"):
    """Plots and saves the ROC curves for multi-class
classification."""
    y_true_bin = label_binarize(y_true,
classes=np.arange(len(classes)))
    n_classes = y_true_bin.shape[1]

    plt.figure(figsize=(12, 10))

    # Compute ROC curve and AUC for each class
    for i in range(n_classes):
        fpr, tpr, _ = roc_curve(y_true_bin[:, i], y_probs[:, i])
        roc_auc = auc(fpr, tpr)
        plt.plot(fpr, tpr, lw=2, label=f"{classes[i]} (AUC =
{roc_auc:.3f})")

    # Micro-average ROC
    fpr_micro, tpr_micro, _ = roc_curve(y_true_bin.ravel(),
y_probs.ravel())
    roc_auc_micro = auc(fpr_micro, tpr_micro)
    plt.plot(fpr_micro, tpr_micro, linestyle="--", color="black",
             label=f"Micro-average (AUC = {roc_auc_micro:.3f})", lw=3)

    plt.plot([0, 1], [0, 1], "k--", lw=2) # Diagonal line
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel("False Positive Rate (FPR)")
    plt.ylabel("True Positive Rate (TPR)")
    plt.title("ROC Curve for Multi-Class Classification")
```

```
    plt.legend(loc="lower right", fontsize=8)
    plt.tight_layout()
    plt.savefig(filename, dpi=300)
    plt.show()
    plt.close()

# =====================================================================
# Constants
# =====================================================================
DATA_DIR = r"C:\Users\SUBARNA MONDAL\Desktop\kaggle-asl\
processed_data"
gestures = list("ABCDEFGHIJKLMNOPQRSTUVWXYZ")
num_classes = len(gestures)
NUM_FRAMES = 50   # updated to match the new model
NUM_FEATURES = 63

# =====================================================================
# Load Dataset
# =====================================================================
def load_and_analyze_original_frames():
    """Loads data, prints a structure report, and analyzes original
frame counts."""
    X, Y = [], []

    # List to store the original frame count of each file
    original_frame_counts = []

    # Initialize counters for our report
    stats = {
        'total_files': 0,
        'processed': 0,
        'single_frame': 0,
        'multiframe_padded': 0,
        'multiframe_trimmed': 0,
        'skipped_invalid': 0
    }

    for gesture in gestures:
        gesture_path = os.path.join(DATA_DIR, gesture)
        if not os.path.exists(gesture_path):
            continue

        files = os.listdir(gesture_path)
        stats['total_files'] += len(files)

        for file in files:
            file_path = os.path.join(gesture_path, file)
            try:
                data = np.load(file_path)
```

```
                # --- NEW: Record original frame count BEFORE
processing ---
                if data.ndim == 1:
                    original_frame_counts.append(1) # A single frame
                elif data.ndim == 2:
                    original_frame_counts.append(data.shape[0]) # Get
frame count from shape

                # Case 1: Single frame (63,)
                if data.ndim == 1 and data.shape[0] == NUM_FEATURES:
                    stats['single_frame'] += 1
                    data = np.tile(data, (NUM_FRAMES, 1))

                # Case 2: Sequence of frames (N, 63)
                elif data.ndim == 2 and data.shape[1] == NUM_FEATURES:
                    if data.shape[0] >= NUM_FRAMES:
                        stats['multiframe_trimmed'] += 1
                        data = data[:NUM_FRAMES]
                    else:
                        stats['multiframe_padded'] += 1
                        pad_len = NUM_FRAMES - data.shape[0]
                        padding = np.repeat(data[-1:], pad_len,
axis=0)
                        data = np.concatenate([data, padding], axis=0)
                else:
                    stats['skipped_invalid'] += 1
                    continue

                X.append(data)
                Y.append(gesture)
                stats['processed'] += 1

            except Exception as e:
                print(f"Skipping {file_path} due to error: {e}")
                stats['skipped_invalid'] += 1
                continue

    # --- Print the original report ---
    print("\n" + "="*50)
    print("      DATASET STRUCTURE REPORT")
    print("="*50)
    # ... (rest of the report printing is the same) ...
    print(f"Total files found: {stats['total_files']}")
    print(f"Successfully processed files: {stats['processed']}")
    print("-" * 25)
    print("Processed File Types:")
    print(f"  - Single-frame (repeated): {stats['single_frame']}")
    print(f"  - Multi-frame (padded):
{stats['multiframe_padded']}")
    print(f"  - Multi-frame (trimmed):
```

```
{stats['multiframe_trimmed']}")
    print("-" * 25)
    print(f"Skipped invalid files: {stats['skipped_invalid']}")
    print("="*50 + "\n")

    # --- NEW: Print the Original Frame Count Analysis ---
    if original_frame_counts:
        print("\n" + "="*50)
        print("      ORIGINAL FRAME COUNT ANALYSIS")
        print("="*50)
        print(f"Minimum original frames in a file:
{np.min(original_frame_counts)}")
        print(f"Maximum original frames in a file:
{np.max(original_frame_counts)}")
        print(f"Average original frames per file:
{np.mean(original_frame_counts):.2f}")
        print("="*50 + "\n")


    X, Y = np.array(X), np.array(Y)
    # Shuffle dataset
    indices = np.arange(len(X))
    np.random.shuffle(indices)
    return X[indices], Y[indices]

print("Loading dataset...")
X, Y = load_and_analyze_original_frames()
print(f"Dataset loaded. X shape: {X.shape}, Y shape: {Y.shape}")

# ===================================================================
# Preprocess Data
# ===================================================================
# 1. Encode Labels
label_encoder = LabelEncoder()
Y_encoded = label_encoder.fit_transform(Y)
Y_categorical = to_categorical(Y_encoded, num_classes=num_classes)

# 2. Split Data into Training and Validation Sets
print("Splitting data into training and validation sets...")
X_train, X_val, Y_train, Y_val = train_test_split(
    X, Y_categorical,
    test_size=0.20,
    random_state=42,
    stratify=Y_encoded  # stratify expects 1D labels
)
print(f"Training data: {X_train.shape}, {Y_train.shape}")
print(f"Validation data: {X_val.shape}, {Y_val.shape}")

# ===================================================================
# New Model Architecture (as requested)
```

```python
# ======================================================================

from tensorflow import keras
from tensorflow.keras import layers

# ==========================
# Learnable Positional Encoding
# ==========================
class LearnablePositionalEncoding(layers.Layer):
    """Trainable positional encoding for transformer input."""
    def __init__(self, d_model, **kwargs):
        super().__init__(**kwargs)
        self.d_model = d_model

    def build(self, input_shape):
        seq_len = input_shape[1]
        self.pos_encoding = self.add_weight(
            name="pos_encoding", shape=(1, seq_len, self.d_model),
            initializer="random_normal", trainable=True
        )

    def call(self, x):
        return x + self.pos_encoding

# ==========================
# Transformer Encoder Block
# ==========================
def transformer_encoder_block(inputs, d_model, num_heads, ff_dim,
dropout_rate=0.2):
    attn = layers.MultiHeadAttention(
        num_heads=num_heads, key_dim=d_model // num_heads,
dropout=dropout_rate
    )(inputs, inputs)

    # Residual + Norm
    x = layers.Add()([inputs, attn])
    x = layers.LayerNormalization(epsilon=1e-6)(x)

    # Feed Forward
    ffn = keras.Sequential([
        layers.Dense(ff_dim, activation="relu"),
        layers.Dropout(dropout_rate),
        layers.Dense(d_model)
    ])
    y = ffn(x)

    # Residual + Norm
    x = layers.Add()([x, y])
    x = layers.LayerNormalization(epsilon=1e-6)(x)
    return x
```

```python
# ==========================
# Build Hybrid Model
# ==========================
def build_model(num_classes, num_frames=NUM_FRAMES,
num_features=NUM_FEATURES,
                num_transformer_blocks=3, d_model=128, num_heads=8,
ff_dim=256):
    """
    CNN + BiLSTM + Transformer for sign language recognition.
    """
    inputs = keras.Input(shape=(num_frames, num_features))

    # --- CNN feature extractor ---
    x = layers.Conv1D(128, 3, padding="same")(inputs)
    x = layers.BatchNormalization()(x); x = layers.ReLU()(x)
    x = layers.MaxPooling1D(2)(x); x = layers.Dropout(0.2)(x)

    x = layers.Conv1D(256, 3, padding="same")(x)
    x = layers.BatchNormalization()(x); x = layers.ReLU()(x)
    x = layers.Dropout(0.2)(x)

    # Project to transformer dimension
    x = layers.Dense(d_model)(x)

    # --- BiLSTM stack ---
    x = layers.Bidirectional(layers.LSTM(128, return_sequences=True,
dropout=0.25))(x)
    x = layers.Bidirectional(layers.LSTM(64,  return_sequences=True,
dropout=0.25))(x)

    # --- Positional Encoding + Transformer ---
    x = LearnablePositionalEncoding(d_model)(x)
    for _ in range(num_transformer_blocks):
        x = transformer_encoder_block(x, d_model, num_heads, ff_dim)

    # --- Classification head ---
    x = layers.GlobalAveragePooling1D()(x)
    x = layers.Dense(256, activation="relu")(x); x =
layers.Dropout(0.4)(x)
    x = layers.Dense(128, activation="relu")(x); x =
layers.Dropout(0.3)(x)
    outputs = layers.Dense(num_classes, activation="softmax")(x)

    return keras.Model(inputs, outputs)

# ===================================================================
# Compile and Train Model
# ===================================================================
print("Building and compiling model...")
```

```
model = build_model(num_classes=num_classes)
model.compile(
    optimizer=keras.optimizers.Adam(learning_rate=0.0005),
    loss="categorical_crossentropy",
    metrics=["accuracy"]
)
model.summary()
plot_model(model, to_file="model_architecture.png", show_shapes=True,
dpi=120)

callbacks = [
    EarlyStopping(monitor="val_loss", patience=10,
restore_best_weights=True, verbose=1),
    ReduceLROnPlateau(monitor="val_loss", factor=0.5, patience=5,
min_lr=1e-5, verbose=1),
    ModelCheckpoint("best_model.keras", save_best_only=True,
monitor="val_loss")
]

print("\nStarting model training...")
history = model.fit(
    X_train, Y_train,
    epochs=40,
    batch_size=128,
    validation_data=(X_val, Y_val),
    callbacks=callbacks
)
print("Training finished.")

# ======================================================================
# Evaluate Model
# ======================================================================
print("\nEvaluating model performance...")

# Load the best weights saved by ModelCheckpoint
model.load_weights("best_model.keras")

# Make predictions on the validation set
Y_pred_probs = model.predict(X_val)
Y_pred_classes = np.argmax(Y_pred_probs, axis=1)
Y_true_classes = np.argmax(Y_val, axis=1) # Convert one-hot encoded
validation labels back

# 1. Classification Report
report = classification_report(
    Y_true_classes,
    Y_pred_classes,
    target_names=label_encoder.classes_,
    digits=4
)
```

```python
print("\nClassification Report:\n", report)
save_classification_report_as_image(report)

# 2. Key Metrics
f1 = f1_score(Y_true_classes, Y_pred_classes, average='weighted')
acc = accuracy_score(Y_true_classes, Y_pred_classes)
prec = precision_score(Y_true_classes, Y_pred_classes,
average='weighted')
rec = recall_score(Y_true_classes, Y_pred_classes, average='weighted')
print(f"Overall Metrics: F1 Score: {f1:.4f} | Accuracy: {acc:.4f} |
Precision: {prec:.4f} | Recall: {rec:.4f}\n")

# 3. Save model and history plot
model.save("sign_language_model_final.h5")
plot_training_history(history)

# 4. Confusion Matrix
cm = confusion_matrix(Y_true_classes, Y_pred_classes)
plt.figure(figsize=(14, 12))
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues",
            xticklabels=label_encoder.classes_,
            yticklabels=label_encoder.classes_)
plt.title("Confusion Matrix")
plt.xlabel("Predicted Label")
plt.ylabel("True Label")
plt.tight_layout()
plt.savefig("confusion_matrix.png")
plt.show()

# 5. ROC Curve
plot_roc_curves(Y_true_classes, Y_pred_probs, label_encoder.classes_)

print("\nAll evaluation artifacts (plots, reports, model) have been
saved.")
```