```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten,
Dense, Dropout
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from sklearn.model_selection import train_test_split
import numpy as np
import os
import cv2

# --- 1. Configuration and Hyperparameters ---
IMAGE_SIZE = (128, 128) # Resize images to this dimension
BATCH_SIZE = 32
EPOCHS = 20
DATA_DIR = 'path/to/your/fabric_dataset' # IMPORTANT: Change this to
your dataset directory

# --- 2. Create the File Dictionary (Filepath -> Label String) ---

def create_file_label_dictionary(data_dir):
    """
    Scans the data_dir and its subdirectories to create a dictionary
    mapping image file paths to their string labels.
    Assumes directory structure:
    data_dir/
        class1/
            img1.jpg
            img2.jpg
        class2/
            img3.jpg
            ...
    Returns:
        dict: A dictionary where keys are absolute image file paths
and values are string labels.
        list: A sorted list of unique class names (labels).
    """
    file_label_dict = {}
    class_names = []

    if not os.path.exists(data_dir):
        print(f"Error: Data directory not found at '{data_dir}'.
Please update DATA_DIR.")
        return {}, []

    # Get unique class names (subdirectories)
    subdirectories = [d for d in os.listdir(data_dir) if
os.path.isdir(os.path.join(data_dir, d))]
    class_names = sorted(subdirectories)
```

```
    if not class_names:
        print(f"Warning: No class subdirectories found in
'{data_dir}'.")
        print("Please ensure your dataset is organized with
subdirectories for each class (e.g., 'fabric_dataset/stripes',
'fabric_dataset/floral').")
        return {}, []

    print(f"Detected classes: {class_names}")

    for class_name in class_names:
        class_path = os.path.join(data_dir, class_name)
        for img_name in os.listdir(class_path):
            img_path = os.path.join(class_path, img_name)
            if os.path.isfile(img_path) and
img_name.lower().endswith(('.png', '.jpg', '.jpeg', '.gif', '.bmp')):
                file_label_dict[img_path] = class_name
            else:
                # Optionally, print warnings for non-image files if
needed
                # print(f"Skipping non-image file: {img_path}")
                pass

    return file_label_dict, class_names

# Create the file dictionary
print("Creating file dictionary...")
files_dictionary, class_names = create_file_label_dictionary(DATA_DIR)

if not files_dictionary:
    print("No files found or an error occurred. Cannot proceed without
data.")
    # Provide placeholder data for demonstration if actual data is not
available
    NUM_CLASSES_PLACEHOLDER = 5
    class_names = [f'class_{i}' for i in
range(NUM_CLASSES_PLACEHOLDER)]
    # Create some dummy file paths and labels
    dummy_filepaths = [f"dummy/path/img_{i}.jpg" for i in range(100)]
    dummy_labels = [class_names[np.random.randint(0,
NUM_CLASSES_PLACEHOLDER)] for _ in range(100)]
    files_dictionary = dict(zip(dummy_filepaths, dummy_labels))
    print("Using placeholder data for demonstration.")


# --- 3. Load Images and Prepare Labels from the File Dictionary ---
```

```
image_filepaths = list(files_dictionary.keys())
string_labels = [files_dictionary[fp] for fp in image_filepaths]

# Map string labels to integer indices
class_to_idx = {name: i for i, name in enumerate(class_names)}
numerical_labels = np.array([class_to_idx[label] for label in
string_labels])

# Load images
X = []
y_loaded = [] # Store labels corresponding to successfully loaded
images
print(f"Loading {len(image_filepaths)} images...")
for i, img_path in enumerate(image_filepaths):
    try:
        img = cv2.imread(img_path)
        if img is not None:
            img = cv2.resize(img, IMAGE_SIZE)
            X.append(img)
            y_loaded.append(numerical_labels[i]) # Use
numerical_labels[i] for the corresponding image
        else:
            print(f"Warning: Could not read image {img_path}")
    except Exception as e:
        print(f"Error loading image {img_path}: {e}")

X = np.array(X)
y = np.array(y_loaded) # Update y to only include labels for
successfully loaded images

if len(X) == 0:
    print("No images were successfully loaded. Exiting.")
    exit() # Exit if no images are loaded

print(f"Successfully loaded {len(X)} images.")

# Normalize pixel values to [0, 1]
X = X.astype('float32') / 255.0

# Convert labels to one-hot encoding
y = tf.keras.utils.to_categorical(y, num_classes=len(class_names))

# --- 4. Split Data into Training, Validation, and Test Sets ---
# First, split into training + validation and test sets
X_train_val, X_test, y_train_val, y_test = train_test_split(
    X, y, test_size=0.15, random_state=42, stratify=y
)
```

```
# Then, split the training + validation set into training and
validation sets
X_train, X_val, y_train, y_val = train_test_split(
    X_train_val, y_train_val, test_size=0.176, random_state=42,
stratify=y_train_val
) # 0.176 of 0.85 is roughly 0.15 of total data (0.85 * 0.176 =
0.1496)

print(f"Training set shape: {X_train.shape}, {y_train.shape}")
print(f"Validation set shape: {X_val.shape}, {y_val.shape}")
print(f"Test set shape: {X_test.shape}, {y_test.shape}")

# --- 5. Data Augmentation (for training data) ---
train_datagen = ImageDataGenerator(
    rotation_range=20,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
    fill_mode='nearest'
)

# --- 6. Define the CNN Model Architecture ---
model = Sequential([
    # Convolutional Block 1
    Conv2D(32, (3, 3), activation='relu', input_shape=(IMAGE_SIZE[0],
IMAGE_SIZE[1], 3)),
    MaxPooling2D((2, 2)),
    Dropout(0.25), # Dropout for regularization

    # Convolutional Block 2
    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    Dropout(0.25),

    # Convolutional Block 3
    Conv2D(128, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    Dropout(0.25),

    # Flatten the output for the Dense layers
    Flatten(),

    # Fully Connected (Dense) Layers
    Dense(256, activation='relu'),
    Dropout(0.5), # More dropout before the final classification layer
    Dense(len(class_names), activation='softmax') # Output layer with
```

```
softmax for multi-class classification
])

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

model.summary()

# --- 8. Train the Model ---
print("Training the model...")
history = model.fit(
    train_datagen.flow(X_train, y_train, batch_size=BATCH_SIZE),
    epochs=EPOCHS,
    validation_data=(X_val, y_val),
    verbose=1
)

# --- 9. Evaluate the Model on the Test Set ---
print("\nEvaluating the model on the test set...")
loss, accuracy = model.evaluate(X_test, y_test, verbose=0)
print(f"Test Loss: {loss:.4f}")
print(f"Test Accuracy: {accuracy:.4f}")

# Optional: Make predictions on the test set
# predictions = model.predict(X_test)
# predicted_classes = np.argmax(predictions, axis=1)
# true_classes = np.argmax(y_test, axis=1)

# print("\nSome sample predictions vs true labels:")
# for i in range(10): # Print for first 10 test samples
#     print(f"Sample {i+1}: True: {class_names[true_classes[i]]},
Predicted: {class_names[predicted_classes[i]]}")

# --- 10. Save the Model (Optional) ---
# model.save('fabric_pattern_classifier.h5')
# print("\nModel saved as 'fabric_pattern_classifier.h5'")
```