Solar Panel Classifier - Code Explanation

This document explains the Python code for building an image classification model using EfficientNetB0 for detecting issues in solar panels such as dust, bird droppings, snow, and damage.

---

1. Imports

```
import os
import cv2
import numpy as np
from PIL import Image
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.applications import EfficientNetB0
from tensorflow.keras.applications.efficientnet import preprocess_input
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Dense, GlobalAveragePooling2D, Dropout
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import ModelCheckpoint, EarlyStopping
from tensorflow.keras.utils import to_categorical
```

Purpose: These libraries support image processing, model training, evaluation, and visualization.

---

2. Class Definition: SolarPanelDataProcessor

```
class SolarPanelDataProcessor:
```

Encapsulates methods for loading data, building a model, and training it.

---

3. Initialization

```
def __init__(self, data_dir, img_size=(224, 224)):
```

- Initializes the class with data directory and image size.
- Sets class names and valid image extensions.
- Prepares a dictionary to track class-wise image distribution.

```
self.data_dir = data_dir                    # Root directory containing image folders per class
    self.img_size = img_size                # Resize all images to this size
    self.classes = ['Clean', 'Dusty', 'Bird-Drop', 'Electrical-Damage', 'Physical-Damage',
'Snow-Covered']
    self.valid_exts = ('.jpg', '.jpeg', '.png', '.bmp', '.tif', '.tiff')
    self.class_distribution = {}            # Dictionary to hold class-wise image counts
```

---

4. Loading Data

def load_data(self):

- Iterates over each class folder, loads valid images, preprocesses them, and appends to lists.

```
images, labels = [], []
    for class_idx, class_name in enumerate(self.classes):
        class_dir = os.path.join(self.data_dir, class_name)          # Path to each class folder
        self.class_distribution[class_name] = 0                     # Initialize count
        for img_name in os.listdir(class_dir):
            if not img_name.lower().endswith(self.valid_exts):       # Skip unsupported files
                continue
            img_path = os.path.join(class_dir, img_name)             # Full image path
            img = cv2.imread(img_path)                               # Read image using OpenCV
            if img is None:
                continue                                             # Skip unreadable images
            img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)               # Convert from BGR to RGB
            img = cv2.resize(img, self.img_size)                     # Resize to standard size
            img = preprocess_input(img)                              # EfficientNet-specific preprocessing
            images.append(img)                                       # Add image to list
            labels.append(class_idx)                                 # Add corresponding label index
            self.class_distribution[class_name] += 1                 # Update count
    return np.array(images), np.array(labels)                       # Return as NumPy arrays
```

---

5. Build Model with EfficientNetB0

def build_model(self, num_classes):

- Builds a CNN model using EfficientNetB0 as the base.

```
  base_model = EfficientNetB0(include_top=False, weights='imagenet', input_shape=(224, 224,
3))
   x = base_model.output
   x = GlobalAveragePooling2D()(x)          # Reduces feature maps to vector by averaging
   x = Dropout(0.5)(x)                      # Prevents overfitting
   predictions = Dense(num_classes, activation='softmax')(x)   # Final classification layer
   model = Model(inputs=base_model.input, outputs=predictions)
   model.compile(optimizer=Adam(1e-4), loss='categorical_crossentropy', metrics=['accuracy'])
# Compile model
   return model
```

---

6. Training and Saving Model

def train_and_save_model(self, save_path='solar_panel_classifier.h5'):

- Loads and splits the dataset, applies data augmentation, trains the model, and saves it.

```
  X, y = self.load_data()                                   # Load and preprocess images
   y_cat = to_categorical(y, num_classes=len(self.classes))          # One-hot encode labels

   # Split into train/test sets
   X_train, X_test, y_train, y_test = train_test_split(X, y_cat, test_size=0.2, stratify=y,
random_state=42)

   # Further split training into train/validation
   X_train, X_val, y_train, y_val = train_test_split(
      X_train, y_train, test_size=0.2, stratify=np.argmax(y_train, axis=1), random_state=42)

   # Create data generators
   train_gen = ImageDataGenerator(rotation_range=20, width_shift_range=0.1,
                              height_shift_range=0.1, horizontal_flip=True)
   val_gen = ImageDataGenerator()
   test_gen = ImageDataGenerator()
```

```
    model = self.build_model(num_classes=len(self.classes))          # Build the model

    # Define callbacks
    callbacks = [
        EarlyStopping(monitor='val_loss', patience=5, restore_best_weights=True),
        ModelCheckpoint(save_path, monitor='val_accuracy', save_best_only=True)
    ]

    # Train the model
    model.fit(train_gen.flow(X_train, y_train, batch_size=32),
              validation_data=val_gen.flow(X_val, y_val),
              epochs=20, callbacks=callbacks)

    print(f"✅ Model saved to: {save_path}")
```

---

7. Main Script Execution

```
if __name__ == "__main__":
    processor = SolarPanelDataProcessor(data_dir='Solar_Panel_Dataset')
    processor.train_and_save_model()
```

Creates an instance of the processor and starts the training pipeline.

---

8. Streamlit App - Solar Panel Classifier UI

Imports and Configuration

```
import streamlit as st
import numpy as np
import cv2
import os
from PIL import Image
from tensorflow.keras.models import load_model
from tensorflow.keras.applications.efficientnet import preprocess_input

MODEL_PATH = 'solar_panel_classifier.h5'
CLASS_NAMES = ['Clean', 'Dusty', 'Bird-Drop', 'Electrical-Damage', 'Physical-Damage',
'Snow-Covered']
IMG_SIZE = (224, 224)
```

Purpose: Import libraries and define constants for loading the model and preprocessing images.

---

9. Function: load_classification_model

```
def load_classification_model():


Loads the trained model from disk.

  try:
      model = load_model(MODEL_PATH)
      return model
   except Exception as e:
      st.error(f"Error loading model: {str(e)}")
      return None
```

---

10. Function: preprocess_image

```
def preprocess_image(image):


Prepares the uploaded image for prediction.

  try:
      img_array = np.array(image)                    # Convert PIL image to NumPy array
      img_array = cv2.resize(img_array, IMG_SIZE)      # Resize to model input size

      if img_array.shape[-1] == 1:                     # Grayscale to RGB
         img_array = cv2.cvtColor(img_array, cv2.COLOR_GRAY2RGB)
      elif img_array.shape[-1] == 4:                   # RGBA to RGB
         img_array = cv2.cvtColor(img_array, cv2.COLOR_RGBA2RGB)

      img_array = preprocess_input(img_array)          # EfficientNet preprocessing
      img_array = np.expand_dims(img_array, axis=0)    # Add batch dimension

      return img_array
   except Exception as e:
      st.error(f"Error preprocessing image: {str(e)}")
      return None
```

---

11. Function: display_prediction

def display_prediction(image, model):

Displays the uploaded image and prediction result.

```
  col1, col2 = st.columns(2)

  with col1:
     st.image(image, caption='Uploaded Solar Panel', use_column_width=True)

  with col2:
     processed_img = preprocess_image(image)                  # Preprocess image
     if processed_img is not None and model is not None:
        prediction = model.predict(processed_img)             # Make prediction
        pred_class = CLASS_NAMES[np.argmax(prediction)]       # Get predicted class name
        confidence = np.max(prediction) * 100                 # Confidence percentage

        st.subheader("Prediction Results")
        st.write(f"*Condition:* {pred_class}")
        st.write(f"*Confidence:* {confidence:.2f}%")

        st.subheader("Probability Distribution")
        prob_data = {
           'Condition': CLASS_NAMES,
           'Probability': prediction[0]
        }
        st.bar_chart(prob_data, x='Condition', y='Probability')
```

---

12. Main Function: Streamlit App

def main():

Sets up and runs the Streamlit app interface.

```
  st.set_page_config(page_title="SolarGuard", page_icon="☀", layout="wide")

  @st.cache_resource
  def load_model():
     return load_classification_model()

  model = load_model()
```

```
    st.title("☀ SolarGuard: Solar Panel Defect Detection")
    st.markdown("""
    Upload an image of a solar panel to detect defects and classify its condition.
    """)

    uploaded_file = st.file_uploader(
        "Choose a solar panel image...",
        type=["jpg", "jpeg", "png"],
        accept_multiple_files=False
    )

    if uploaded_file is not None:
        try:
            image = Image.open(uploaded_file)
            display_prediction(image, model)
        except Exception as e:
            st.error(f"Error processing image: {str(e)}")

    st.markdown("---")
    st.subheader("About SolarGuard")
    st.write("""
    SolarGuard is an AI-powered system that automatically detects defects and classifies
    the condition of solar panels. It helps in:
    - Automated inspection of solar farms
    - Optimizing maintenance schedules
    - Improving energy efficiency
    - Reducing operational costs
    """)

    st.subheader("Defect Classes")
    cols = st.columns(3)
    for i, class_name in enumerate(CLASS_NAMES):
        with cols[i % 3]:
            st.markdown(f"{class_name}")
            st.write("Clean panels" if class_name == "Clean" else f"Panels with
{class_name.lower()}")
```

---

13. Script Execution Entry Point

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

Executes the Streamlit app when run directly.

This section of code defines a complete user interface for uploading and classifying solar panel images in real-time using a trained deep learning model.