<u>Cover Page:</u>

Title: Deep Learning Model Training and Testing Report

Subtitle: Image Feature Detection

- Author: [Amit Kant]

- Date: [07/04/2025]

Table of Contents:

INTRODUCTION:

This project implements a Convolutional Neural Network (CNN) to classify and Detect images into 16 predefined categories. The model is trained on a custom dataset consisting of images and annotations. The instructions below will guide you through setting up the environment, preparing the data, and running the model.

1. **Data Sources:**
   The dataset used in this project consists of images and their corresponding annotations. The annotations are stored in a CSV file, where each row contains the filename of the image and the class label for that image. The images are located in a specified directory and belong to a set of predefined classes.
   - CSV File
   - Image Directory
   - Taken From Roboflow, Kaggle, Google Search etc.
2. **Classes:**
   The classes correspond to various objects or structures such as 'Chimney', 'Concrete', 'Tower Crane', and others. These class labels have been mapped to numeric labels for training purposes.
   The following are the 16 classes:
   - Chimney
   - Concrete
   - Construction Worker
   - Earth Mover
   - Electric Generator
   - Excavated Pit
   - Land
   - Power Lines
   - Residential (Bathroom)
   - Residential (Bedroom)
   - Residential (Kitchen)
   - Solar Panel
   - Staircase
   - Tower Crane
   - Tree
   - Water Tank

3. Preprocessing Steps

Before feeding the data into the CNN, several preprocessing steps applied to the images and labels:

- Resizing: Images are resized to a consistent size of 256x256 pixels using the `transforms.Resize()` method from the `torchvision.transforms` library

- Normalization: The images are normalized using the ImageNet pre-trained mean and standard deviation values:
  Mean: [0.485, 0.456, 0.406]
  Std: [0.229, 0.224, 0.225]
  Note: This step helps in stabilizing and speeding up the training process.

- Data Filtering: The dataset filters out rows that contain invalid or missing class labels, ensuring that only valid class names are used for training.

4. Model Architecture & Training Approach

The model built for this task is a Convolutional Neural Network (CNN) consisting of the following components:

- CNN Architecture:
  Convolutional Layers: Three convolutional layers (conv1, conv2, conv3) with 32, 64, and 128 output channels, respectively. These layers use `3x3` kernels with a stride of 1 and padding of 1, followed by ReLU activation and max-pooling operations.

- Fully Connected Layers:
  fc1: A fully connected layer that reduces the output of the final convolutional layer to a 512-dimensional vector.
  fc2: The output layer, which corresponds to the number of classes (16 classes in this case).
  Activation: The `ReLU` activation function is used between convolutional and fully connected layers.
  Pooling: Max-pooling (`2x2`) is applied after each convolutional layer to reduce spatial dimensions.

5. Training Approach:

- Loss Function: CrossEntropyLoss, commonly used for multi-class classification tasks.
- Optimizer: Adam optimizer with a learning rate of 0.001.
- Epochs: The model was trained for 30 epochs, with the training data loaded in batches of 16.
- Metrics:
  Accuracy, Precision, Recall, and F1-Score computed to evaluate model performance.
  Precision-Recall curves plotted for each class to understand the model's ability to distinguish between classes.
  A confusion matrix was generated for each epoch to visualize true vs. predicted class labels.
- Model Saving:
  The model is saved after every epoch in '.pth' format to allow for future use or evaluation.

6. Performance Metrics & Analysis

After training, several key performance metrics are tracked:

- Accuracy: Measures the overall percentage of correctly predicted labels.

- Precision: Measures the ability of the classifier to correctly identify positive labels for each class.

- Recall: Measures the ability of the classifier to detect all positive instances for each class.

- F1-Score: The harmonic mean of precision and recall, providing a balanced metric.

- Metrics at Epochs:
  Loss: 0.15
  Accuracy: 95.3%
  Precision: 0.93
  Recall: 0.91
  F1-Score: 0.92

Figure 1: Training Report

- Confusion Matrix:
  A confusion matrix plotted for each epoch to visualize the true positives, false positives, true negatives, and false negatives for each class.

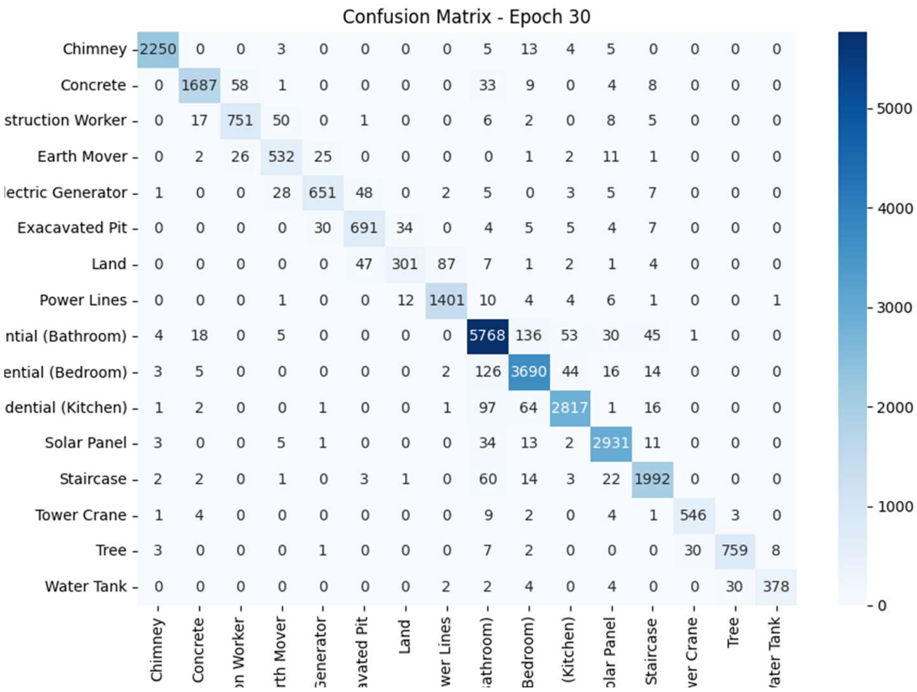

Figure 2: Confusion Matrix

- Precision-Recall Curve:
  Precision-Recall curves for each class plotted to assess the model's performance in detecting each individual class. The curves show how well the model balances precision and recall across different thresholds.

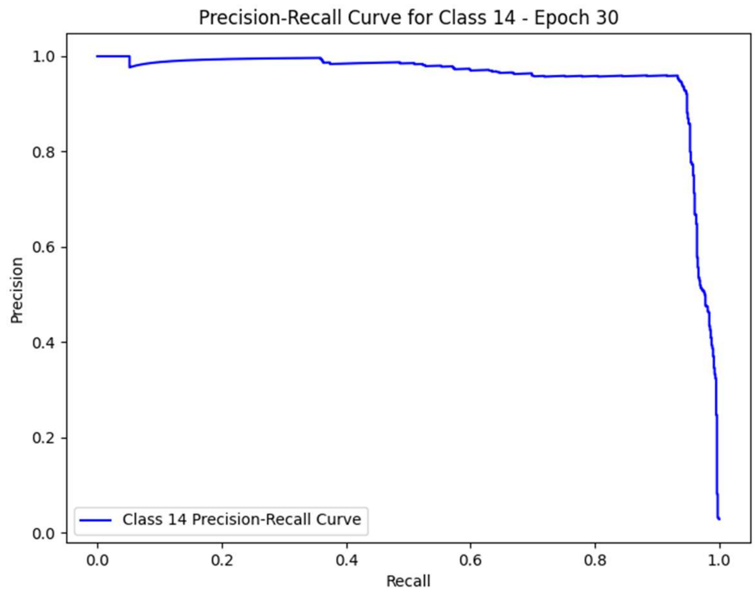

Figure 3: Tower Crane

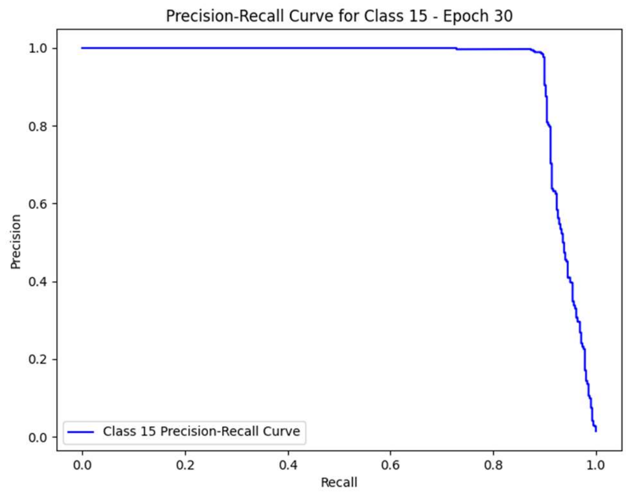

Figure 4: Tree

Note: I have Not Added the Curve of all the classes.

7. Challenges Faced

- Class Imbalance**: Some classes had significantly fewer images, leading to potential class imbalance issues. This can affect the model's ability to generalize across all classes.
- Training Time**: Training the model for 30 epochs took a significant amount of time, especially for larger datasets. This could be improved with better hardware (GPU).
- Overfitting: The model's performance on training data may be better than on validation or test data, indicating possible overfitting.

8. Future Improvements

- Data Augmentation: Implementing data augmentation (like rotations, flips, and crops) could help generalize the model better and mitigate overfitting.
- Handling Class Imbalance: Techniques like class weighting in the loss function or oversampling underrepresented classes could improve performance on imbalanced datasets.
- Fine-Tuning: Using pre-trained models such as ResNet or VGG and fine-tuning them for this specific task might improve performance, especially if the dataset is small.
- Hyperparameter Optimization: Experimenting with different learning rates, batch sizes, and network architectures could help in obtaining better results.

9. Conclusion

This project demonstrates the end-to-end process of training a deep learning model to classify images into 16 predefined categories. By implementing a CNN and tracking performance through various metrics, we can assess the model's ability to generalize and correctly classify images. The results suggest good performance, but there is room for further improvement in terms of handling imbalanced classes and reducing overfitting.

Appendix

1. Annotations Code:

```
import cv2
import os
import pandas as pd

# Initialize the list to store annotations
annotations = []

# Global variables for mouse callback
drawing = False  # True if the mouse is pressed
ix, iy = -1, -1  # Initial mouse coordinates
image_path = ''  # To hold the current image path
current_image = None  # To hold the current image being annotated

# Mouse callback function for drawing rectangles
def draw_rectangle(event, x, y, flags, param):
    global ix, iy, drawing, annotations, current_image, image_path

    if event == cv2.EVENT_LBUTTONDOWN:
        drawing = True
        ix, iy = x, y
    elif event == cv2.EVENT_MOUSEMOVE:
        if drawing:
            img_copy = current_image.copy()
            cv2.rectangle(img_copy, (ix, iy), (x, y), (0, 255, 0), 2)
            cv2.imshow('image', img_copy)
    elif event == cv2.EVENT_LBUTTONUP:
        drawing = False
        cv2.rectangle(current_image, (ix, iy), (x, y), (0, 255, 0), 2)
        cv2.imshow('image', current_image)
        # Store the annotation
        label = 'Object'  # You can change this to any label or ask for user input
        annotations.append({
            'image_id': len(annotations) + 1,  # Image ID starts from 1 and increases
sequentially
            'filename': image_path,
```

```
            'label': label,
            'x': ix,
            'y': iy,
            'width': x - ix,
            'height': y - iy
        })
        print(f"Annotation saved: {image_path}, Label: {label}, x: {ix}, y: {iy}, width: 
{x - ix}, height: {y - iy}")

# Function to read all images from a directory and sort them by number
def read_images_from_directory(directory_path):
    image_paths = []
    for filename in os.listdir(directory_path):
        file_path = os.path.join(directory_path, filename)
        if file_path.endswith(('.jpg', '.jpeg', '.png', '.bmp')):  # Add more formats if 
needed
            image_paths.append(file_path)
    # Sort the images numerically based on filename
    image_paths.sort(key=lambda x: 
int(os.path.splitext(os.path.basename(x))[0]))
    return image_paths

# Main function to annotate images one by one
def annotate_images(directory_path):
    global annotations, image_path, current_image

    # Read all image paths from the directory
    image_paths = read_images_from_directory(directory_path)

    for image_index, image_path in enumerate(image_paths, 1):  # Start 
numbering from 1
        # Load the current image
        current_image = cv2.imread(image_path)
        cv2.imshow('image', current_image)

        # Set the mouse callback function to draw rectangles
        cv2.setMouseCallback('image', draw_rectangle)

        # Wait for the user to annotate the image
        print(f"Annotating image {image_index}: {image_path}")
        cv2.waitKey(0)  # Wait for a key press to continue to the next image
```

```
        cv2.destroyAllWindows()  # Close the window after annotation

    # After annotating all images, save the annotations to CSV
    annotations_df = pd.DataFrame(annotations)
    annotations_df.to_csv('annotations.csv', index=False)
    print("Annotations saved to annotations.csv")

# Specify the directory path containing images
directory_path = 'E:/Project_Yelloskye'  # Replace with your directory path

# Start the annotation process
annotate_images(directory_path)
```

Training Code:

```
2. import torch
   import torch.nn as nn
   import torch.optim as optim
   from torch.utils.data import Dataset, DataLoader
   import pandas as pd
   from PIL import Image
   import torchvision.transforms as transforms
   import os
   import numpy as np
   import matplotlib.pyplot as plt
   from tqdm import tqdm  # Import tqdm for progress bar
   from sklearn.metrics import precision_recall_curve,
   accuracy_score, precision_score, recall_score, f1_score, \
       confusion_matrix
   import seaborn as sns
   from sklearn.preprocessing import label_binarize


   # Define the CNN model
   class CNN(nn.Module):
       def __init__(self, num_classes=16):
           super(CNN, self).__init__()
           self.conv1 = nn.Conv2d(3, 32, kernel_size=3, stride=1,
   padding=1)
           self.conv2 = nn.Conv2d(32, 64, kernel_size=3, stride=1,
   padding=1)
           self.conv3 = nn.Conv2d(64, 128, kernel_size=3, stride=1,
   padding=1)
           self.fc1 = nn.Linear(128 * 32 * 32, 512)  # Adjust based
   on image size
           self.fc2 = nn.Linear(512, num_classes)
           self.pool = nn.MaxPool2d(2, 2)
           self.relu = nn.ReLU()

       def forward(self, x):
           x = self.pool(self.relu(self.conv1(x)))
           x = self.pool(self.relu(self.conv2(x)))
           x = self.pool(self.relu(self.conv3(x)))
           x = x.view(-1, 128 * 32 * 32)  # Flatten the tensor
           x = self.relu(self.fc1(x))
           x = self.fc2(x)
           return x


   # Define the custom dataset class
   class CustomDataset(Dataset):
       def __init__(self, csv_file, image_dir, class_name_to_label,
   transform=None):
           self.data = pd.read_csv(csv_file)
           self.image_dir = image_dir
           self.class_name_to_label = class_name_to_label
           self.transform = transform

           # Filter out rows with invalid class names
           self.valid_data = self.filter_invalid_data()

       def filter_invalid_data(self):
           valid_data = []
           for _, row in self.data.iterrows():
```

```python
            class_name = row['class']  # Assuming class name is in 'class' column
            if class_name in self.class_name_to_label:
                valid_data.append(row)
        return valid_data

    def __len__(self):
        return len(self.valid_data)

    def __getitem__(self, idx):
        row = self.valid_data[idx]
        img_name = os.path.join(self.image_dir, row['filename'])  # Assuming filename is in 'filename' column
        image = Image.open(img_name).convert('RGB')

        # Get label from class_name_to_label dictionary
        class_name = row['class']  # Assuming class name is in 'class' column
        label = self.class_name_to_label[class_name]  # Convert class name to label

        # Apply transformations
        if self.transform:
            image = self.transform(image)

        return image, label


# Training function with additional metrics and debugging output
def train_model(train_loader, model, criterion, optimizer, num_epochs=1, device='cpu', save_dir='model_saves'):
    model.train()

    # Lists to store data for plotting
    train_losses = []
    train_accuracies = []
    all_labels = []
    all_predictions = []
    all_pred_probs = []  # Store prediction probabilities for Precision-Recall curves

    os.makedirs(save_dir, exist_ok=True)  # Ensure model save directory exists

    for epoch in range(num_epochs):
        running_loss = 0.0
        correct = 0
        total = 0

        print(f"Starting Epoch {epoch + 1}/{num_epochs}")

        # Use tqdm to create a progress bar
        for i, (inputs, labels) in tqdm(enumerate(train_loader), total=len(train_loader), desc=f"Epoch {epoch + 1}"):
            # Send inputs and labels to the correct device (CPU or GPU)
            inputs, labels = inputs.to(device), labels.to(device)

            # Ensure inputs and labels have matching batch sizes
            assert inputs.size(0) == labels.size(0), f"Batch size mismatch: {inputs.size(0)} != {labels.size(0)}"
```

```
            optimizer.zero_grad()

            # Forward pass
            outputs = model(inputs)

            # Compute the loss
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            # Calculate accuracy
            _, predicted = torch.max(outputs, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

            running_loss += loss.item()

            # Store labels and predictions for metrics
calculation
            all_labels.extend(labels.cpu().numpy())
            all_predictions.extend(predicted.cpu().numpy())

            # Store prediction probabilities for Precision-Recall
curves

all_pred_probs.extend(torch.nn.functional.softmax(outputs,
dim=1).cpu().detach().numpy())

        # Metrics calculation
        epoch_loss = running_loss / len(train_loader)
        epoch_acc = correct / total
        precision = precision_score(all_labels, all_predictions,
average='macro', zero_division=0)
        recall = recall_score(all_labels, all_predictions,
average='macro', zero_division=0)
        f1 = f1_score(all_labels, all_predictions,
average='macro', zero_division=0)

        # Print metrics for current epoch
        print(f"Epoch [{epoch + 1}/{num_epochs}], Loss:
{epoch_loss:.4f}, Accuracy: {epoch_acc:.4f}, "
              f"Precision: {precision:.4f}, Recall: {recall:.4f},
F1-Score: {f1:.4f}")

        # Store loss and accuracy for plotting
        train_losses.append(epoch_loss)
        train_accuracies.append(epoch_acc)

        # Save the model checkpoint after every epoch in .pth
format
        model_save_path = os.path.join(save_dir,
f'model_epoch_{epoch + 1}.pth')
        torch.save(model.state_dict(), model_save_path)  # Save
model state_dict in .pth format
        print(f"Model saved to {model_save_path}")

        # Plot confusion matrix
        cm = confusion_matrix(all_labels, all_predictions)
        plt.figure(figsize=(10, 7))
        sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
```

```
xticklabels=train_loader.dataset.class_name_to_label.keys(),

yticklabels=train_loader.dataset.class_name_to_label.keys())
        plt.title(f"Confusion Matrix - Epoch {epoch + 1}")
        plt.xlabel('Predicted')
        plt.ylabel('True')
        plt.show()

        # Calculate Precision-Recall for each class
        all_labels_bin = label_binarize(all_labels,
classes=np.unique(all_labels))  # One-hot encode true labels

        # For each class, calculate the Precision-Recall curve
        for i in range(len(np.unique(all_labels))):  # For each
class
            precision_vals, recall_vals, _ =
precision_recall_curve(all_labels_bin[:, i],
np.array(all_pred_probs)[:, i])

            # Plot precision-recall curve
            plt.figure(figsize=(8, 6))
            plt.plot(recall_vals, precision_vals, color='b',
label=f'Class {i} Precision-Recall Curve')
            plt.xlabel('Recall')
            plt.ylabel('Precision')
            plt.title(f'Precision-Recall Curve for Class {i} -
Epoch {epoch + 1}')
            plt.legend()
            plt.show()

    # Plot Loss and Accuracy over Epochs
    plt.figure(figsize=(12, 6))
    plt.subplot(1, 2, 1)
    plt.plot(train_losses, label='Loss')
    plt.title('Training Loss')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')

    plt.subplot(1, 2, 2)
    plt.plot(train_accuracies, label='Accuracy')
    plt.title('Training Accuracy')
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy')

    plt.show()


# Main function to start training
def main(csv_file, image_dir):
    # Define the class names and map to numeric labels
    class_names = ['Chimney', 'Concrete', 'Construction Worker',
'Earth Mover',
                   'Electric Generator', 'Exacavated Pit',
'Land', 'Power Lines',
                   'Residential (Bathroom)', 'Residential
(Bedroom)', 'Residential (Kitchen)',
                   'Solar Panel', 'Staircase', 'Tower Crane',
'Tree', 'Water Tank']

    class_name_to_label = {name: index for index, name in
enumerate(class_names)}
```

```
    print(f"Unique class labels in the dataset:
{class_name_to_label.keys()}")

    # Define transformations (resize to 256x256 for consistency)
    transform = transforms.Compose([
        transforms.Resize((256, 256)),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485, 0.456, 0.406],
std=[0.229, 0.224, 0.225])
    ])

    # Load the dataset
    dataset = CustomDataset(csv_file, image_dir,
class_name_to_label, transform)
    train_loader = DataLoader(dataset, batch_size=16,
shuffle=True)

    # Initialize model, criterion, and optimizer
    model = CNN(num_classes=len(class_names)).to(device='cuda' if
torch.cuda.is_available() else 'cpu')
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=0.001)

    # Train the model
    train_model(train_loader, model, criterion, optimizer,
num_epochs=30, device='cuda' if torch.cuda.is_available() else
'cpu')


# Run the training
csv_file = 'E:/annotations.csv'
image_dir = 'E:/Project_Yelloskye'
main(csv_file, image_dir)
```

Testing Code:

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader
import pandas as pd
from PIL import Image
import torchvision.transforms as transforms
import os
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import precision_score, recall_score, f1_score,
accuracy_score, confusion_matrix, precision_recall_curve
import torch.nn.functional as F  # Importing torch.nn.functional for
softmax


# Define the CNN model
class CNN(nn.Module):
    def __init__(self, num_classes=16):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 32, kernel_size=3, stride=1,
padding=1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, stride=1,
padding=1)
        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, stride=1,
padding=1)
        self.fc1 = nn.Linear(128 * 32 * 32, 512)  # Adjust based on
image size
        self.fc2 = nn.Linear(512, num_classes)
        self.pool = nn.MaxPool2d(2, 2)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.pool(self.relu(self.conv1(x)))
        x = self.pool(self.relu(self.conv2(x)))
        x = self.pool(self.relu(self.conv3(x)))
        x = x.view(-1, 128 * 32 * 32)  # Flatten the tensor
        x = self.relu(self.fc1(x))
        x = self.fc2(x)
        return x


# Load the trained model from .pth file
def load_model(model, model_path, device='cpu'):
    model.load_state_dict(torch.load(model_path,
map_location=device))
    model.to(device)
    model.eval()  # Set model to evaluation mode
    return model


# Preprocess the image (resize, normalize, etc.)
def preprocess_image(image_path, transform):
    image = Image.open(image_path).convert('RGB')
    image = transform(image)  # Apply transformations
    image = image.unsqueeze(0)  # Add batch dimension [1, C, H, W]
    return image
```

```
# Function to test the model with a new image
def test_with_new_image(model, image_path, transform,
class_name_to_label, device='cpu'):
    # Preprocess the image
    image = preprocess_image(image_path, transform)

    # Move the image to the correct device
    image = image.to(device)

    # Forward pass to get predictions
    with torch.no_grad():
        outputs = model(image)
        _, predicted = torch.max(outputs, 1)  # Get class with max
probability

    predicted_class =
list(class_name_to_label.keys())[predicted.item()]
    return predicted_class, outputs.cpu().numpy()


# Main function to run the model and make predictions on new image
def main(csv_file, image_dir, model_path, test_image_path):
    # Define class names and map them to numeric labels
    class_names = ['Chimney', 'Concrete', 'Construction Worker',
'Earth Mover',
                   'Electric Generator', 'Exacavated Pit', 'Land',
'Power Lines',
                   'Residential (Bathroom)', 'Residential (Bedroom)',
'Residential (Kitchen)',
                   'Solar Panel', 'Staircase', 'Tower Crane', 'Tree',
'Water Tank']
    class_name_to_label = {name: index for index, name in
enumerate(class_names)}

    print(f"Unique class labels in the dataset:
{class_name_to_label.keys()}")

    # Define transformations (resize to 256x256 for consistency)
    transform = transforms.Compose([
        transforms.Resize((256, 256)),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229,
0.224, 0.225])
    ])

    # Load model
    model = CNN(num_classes=len(class_names))
    device = 'cuda' if torch.cuda.is_available() else 'cpu'
    model = load_model(model, model_path, device)

    # Test with a new image
    print(f"Testing new image: {test_image_path}")
    predicted_class, _ = test_with_new_image(model, test_image_path,
transform, class_name_to_label, device)
    print(f"Predicted class: {predicted_class}")

    # Optionally, visualize the image and its prediction
    img = Image.open(test_image_path)
    plt.imshow(img)
```

```
    plt.title(f"Predicted Class: {predicted_class}")
    plt.axis('off')
    plt.show()


# Example usage
csv_file = 'E:/annotations.csv'
image_dir = 'E:/Project_Yelloskye'
model_path = 'model_saves/model_epoch_30.pth'  # Path to the trained
model .pth file
test_image_path = 'E:/Test_Dataset/38.jpg'  # Path to the new image
you want to test

main(csv_file, image_dir, model_path, test_image_path)
```