UNIVERSITA' DEGLI STUDI DI CATANIA
DIPARTIMENTO DI MATEMATICA E INFORMATICA
CORSO DI LAUREA IN DATA SCIENCE

Classifying Small Images with Custom CNN and Faster R-CNN Architectures

Negar Ialeh

Barbara Micalizzi

1 Introduction

The aim of this project is to develop and assess an image classification and object detection system capable of distinguishing between small images across multiple categories. A key constraint of the project is the avoidance of pre-trained neural networks, encouraging the design and implementation of custom models from scratch. This approach ensures full control over the architecture and training process, offering a more transparent view of model behaviour and performance.

The dataset consists of eight distinct classes, each represented by a collection of images with unique backgrounds for training, and a set of 800 test images that all share the same background. This difference in background between training and testing phases introduces a notable challenge, requiring the model to focus on class-defining features while ignoring background noise.

To handle the dataset efficiently in a cloud-based environment such as Google Colab, local datasets were mounted and extracted directly, preventing memory overload and ensuring smooth access. Along with the image files, a CSV file containing bounding box coordinates for the training images was provided, which supports the object detection component of the project.

Two model types were explored to address the classification task: a Faster R-CNN detector for object localization and a custom Convolutional Neural Network (CNN) for image classification.

These models were trained and evaluated with the goal of accurately recognizing and distinguishing between the classes in the test dataset.

The following sections describe the data pre-processing pipeline, model architectures, training procedures, and evaluation metrics, offering a comprehensive analysis of the methods and results obtained throughout the project.

2 Model Description

This project explores two custom-built models for image classification and object detection: a Convolutional Neural Network (CNN) and a Faster R-CNN. Both models were implemented using PyTorch and adapted to the dataset's characteristics to ensure optimal performance without relying on pre-trained weights.

Faster R-CNN Architecture

The detection model is based on a custom Faster R-CNN architecture with a **ResNet-50** backbone, adapted to extract features suitable for object detection.

The model was built using `torchvision.models.detection.fasterrcnn_resnet50_fpn`, modified with the following specifications:

- **Anchor Generator**: An **anchor** is a predefined box (template) with a specific **size** and **aspect ratio**.
  The model uses it as a **starting point** to predict the actual bounding boxes of objects in an image.

  `sizes= ((32,), (64,), (128,), (256,), (512,))`

  This sets **five different scales** for the anchors, from small to large:

1. 32 pixels
2. 64 pixels
3. 128 pixels
4. 256 pixels
5. 512 pixels

   `aspect_ratios= ((0.5, 1.0, 2.0),) * 5`

   For **each scale**, three different **aspect ratios** are defined:

   a) **0.5** = a **tall and narrow** rectangle
   b) **1.0**= a **square**
   c) **2.0** = a **wide and short** rectangle

The * 5 means these aspect ratios are repeated for all five scales.

- **Region Proposal Network (RPN)**: Generates candidate region proposals from feature maps.
- **RoI Align**: Ensures precise feature alignment for each region proposal.
- **Box Head and Classifier**: Classify and refine the bounding boxes. Modification of the model's ROI Head.

  in_features = model.roi_heads.box_predictor.cls_score.in_features

  This extracts the **number of input features** going into the classification head (cls_score) of the ROI (Region of Interest) head.

  model.roi_heads.box_predictor = FastRCNNPredictor(in_features, 2)

  This **replaces the existing box predictor** with a new one:

  - in_features: the number of input features
  - 2: the number of classes (**0 = background** (no object), **1 = object)**

For bounding box prediction, the **Smooth L1 Loss** is used, defined as:

$$\mathcal{L}_{reg}(x) = \begin{cases} 0.5x^2 & \text{if } |x| < 1 \\ |x| - 0.5 & \text{otherwise} \end{cases}$$

The classification component of Faster R-CNN uses the standard cross-entropy loss. The optimizer was again **Adam**, with a learning rate of 1e-4.

Overall, the two models were selected to reflect complementary perspectives on the task: Faster R-CNN for localization and detection of the object and the CNN for pure classification. The architectural diversity ensures a thorough exploration of strategies suitable for image recognition tasks from scratch.

CNN Architecture

The CNN model was designed to perform multi-class classification. It consists of three convolutional blocks; each composed of the following sequence of layers:

- **Convolutional Layer**: Applies 2D convolution with increasing filter depth across layers (e.g., 16, 32, 64 filters) and a fixed kernel size (typically 3×3).
- **Batch Normalization**: Normalizes the activations to accelerate training and improve stability.
- **ReLU Activation**: Introduces non-linearity using the Rectified Linear Unit function $f(x) = max\ (0, x)$.
- **Max Pooling**: Downsamples the spatial dimensions (2×2 pooling), reducing computational load and capturing translation-invariant features.

After the convolutional blocks, the output is flattened and passed through a fully connected layer with 128 units and ReLU activation, followed by a dropout layer with a 30% drop rate, and finally a fully connected output layer that maps to the number of classes for classification.

The model was trained using the **Cross-Entropy Loss**, defined as:

$$\mathrm{LOSS} = -\sum_{\mathrm{i}=1}^{\mathrm{c}} \mathrm{y_i} \log(\hat{\mathrm{y}}_\mathrm{i})$$

where C is the number of classes, y_i is the ground truth, and $\hat{y}_i$ is the predicted probability.

The optimization strategy employed **Adam** with an initial learning rate of 1e-4 and default momentum parameters.

3 Dataset

Data Source:
The dataset used in this project was provided as part of the **AI@UNICT2025 Challenge** and is publicly available on **Kaggle**. It contains annotated images designed for object detection and classification tasks.

Dataset Composition and Size:
The dataset consists of two main parts:

- **Training Set:** Approximately 1600 annotated images belonging to **8 distinct classes**. Each image includes a bounding box annotation specifying the object's location within the image. Notably, each class is associated with a **unique background**, which helps distinguish them in the training phase.
- **Test Set:** Comprises **800 unannotated images**, all featuring the **same background**, regardless of class. This uniformity introduces a potential challenge for models trained on the background-dependent training data.

Annotations:
The training annotations are provided in a CSV file, which contains:

- The image file name
- Bounding box coordinates (xmin, ymin, xmax, ymax)
- Class label (from 0 to 7)

Preprocessing Steps:
Given the mismatch in background distribution between training and test sets, preprocessing played a critical role in this project. The following strategies were implemented:

- **Background Removal:**
  To address the inconsistency in backgrounds, backgrounds were removed, and bounding boxes were used to crop out only the object from each image. This step was applied to both the training and test sets, removing background bias and ensuring the model focuses solely on object features.
- **Resizing:**
  All images (cropped or otherwise) were resized to **224×224 pixels**. This fixed input size is required for the convolutional neural networks and helps reduce computational load while maintaining sufficient detail.

- **Data Augmentation:**
  To improve the generalization capability of the classifier and prevent overfitting, several augmentation techniques were applied to the training images:
  - **Random horizontal flipping**
  - **Small rotations (±10°)**
  - **Color jitter (brightness and contrast)**
  - **Random translations**

Dataset Splitting:
The annotated dataset was split into:

Training Set (80%) – Used to learn the model parameters.

Validation Set (20%) – Used to evaluate model performance during training and tune hyperparameters.

Test Set – Reserved for final model evaluation on unseen data.

This preprocessing pipeline ensures that the models are trained on data that closely resembles the test distribution in structure (i.e., objects-only), while also being diverse enough to learn class-relevant features effectively.

4 Training Process

The training process was designed to promote effective learning while minimizing overfitting, through careful dataset management, augmentation strategies, and appropriate loss functions and optimizers. The process is divided into two main components: the object detector and the image classifier.

4.1 Training Setup

4.1.1 Detector

The object detection model is based on **Faster R-CNN with a ResNet-50 backbone**, trained from scratch. The `train_detector()` function is responsible for training a Faster R-CNN object detection model using PyTorch. The training process involves the following key steps:

1. **Dataset and DataLoader Initialization**
   A custom dataset is created by loading image paths and corresponding bounding boxes from a CSV file. A PyTorch DataLoader is used to iterate over the dataset in mini-batches. A custom collate_fn is provided to correctly batch variable-length annotations (i.e., varying numbers of bounding boxes per image).

2. **Model Setup**
   A pre-configured Faster R-CNN model is instantiated and moved to the appropriate device (GPU if available). The optimizer used is Adam with a learning rate of 0.0001.

3. **Training Loop**
   The model is set to training mode. For each epoch, the training data is looped through batch-by-batch. Images and target annotations are moved to the same device as the model. A forward pass is performed, and the model returns a dictionary containing multiple loss components (e.g., classification and regression losses). These losses are summed into a single scalar loss, which is used for backpropagation. The optimizer then updates the model parameters accordingly.

4. **Model Saving**
   After training is complete, the model's learned parameters are saved to a file for later use or evaluation.

The function returns the trained model instance.

During training, the Faster R-CNN model returns a dictionary containing multiple loss components. These losses correspond to different parts of the model and are designed to optimize both object classification and bounding box regression. The total loss is computed as the sum of the following components:

- **loss_classifier**:
  This loss measures how accurately the model classifies objects within the proposed regions. It is computed using cross-entropy loss based on the predicted class scores.
- **loss_box_reg**:
  This is the bounding box regression loss from the final detection head. It evaluates how well the predicted bounding boxes match the ground truth boxes, using smooth L1 loss.
- **loss_objectness**:
  This loss comes from the Region Proposal Network (RPN) and determines how confidently the model predicts whether a region contains an object or not.
- **loss_rpn_box_reg**:
  This is the bounding box regression loss from the RPN. It refines the coordinates of the candidate regions before they are passed to the classification head.

These individual losses are summed to produce the overall training loss, which is used to update the model parameters. Monitoring each component separately can provide insights into how different parts of the network are learning and help diagnose potential training issues.

4.1.2 Classifier

The image classification task is performed using a **custom-built Convolutional Neural Network (CNN)**. The dataset—containing filenames, bounding box coordinates, and class labels—is first read from a CSV file and then split into **80% training and 20% validation**.

To improve generalization, training images undergo data augmentation including **random horizontal flips, rotations**, and **brightness adjustments**. The validation set, by contrast, is only **resized and normalized** to maintain evaluation consistency and reproducibility.

4.2.2 Classifier

For each epoch:

- The model is set to training mode using model.train(), ensuring that layers like dropout and batch normalization behave appropriately during training.
- Each batch of input images and their corresponding labels is loaded from the training DataLoader and transferred to the selected computing device (CPU or GPU).

- A forward pass is performed: the model processes the images and outputs a tensor of class scores for each image in the batch.
- The CrossEntropyLoss function compares the predicted class scores with the true labels to calculate the classification loss for the batch.
- Gradients are reset with `optimizer.zero_grad()`, then backpropagation is performed using `loss.backward()` to compute gradients with respect to the model's parameters. The weights are then updated via `optimizer.step()` using the Adam optimizer.
- For performance tracking, the number of correct predictions is calculated by comparing the model's predicted classes (obtained via argmax) with the ground truth labels. The cumulative loss and accuracy for the epoch are updated accordingly.
- After all batches are processed, the average training loss and accuracy are calculated and printed. These metrics provide insight into the model's learning progress and help identify overfitting or underfitting during training.

The final classifier model is saved as **best_model_mixed(solutionv4).pth**.

4.3 Evaluation Metrics and Configuration

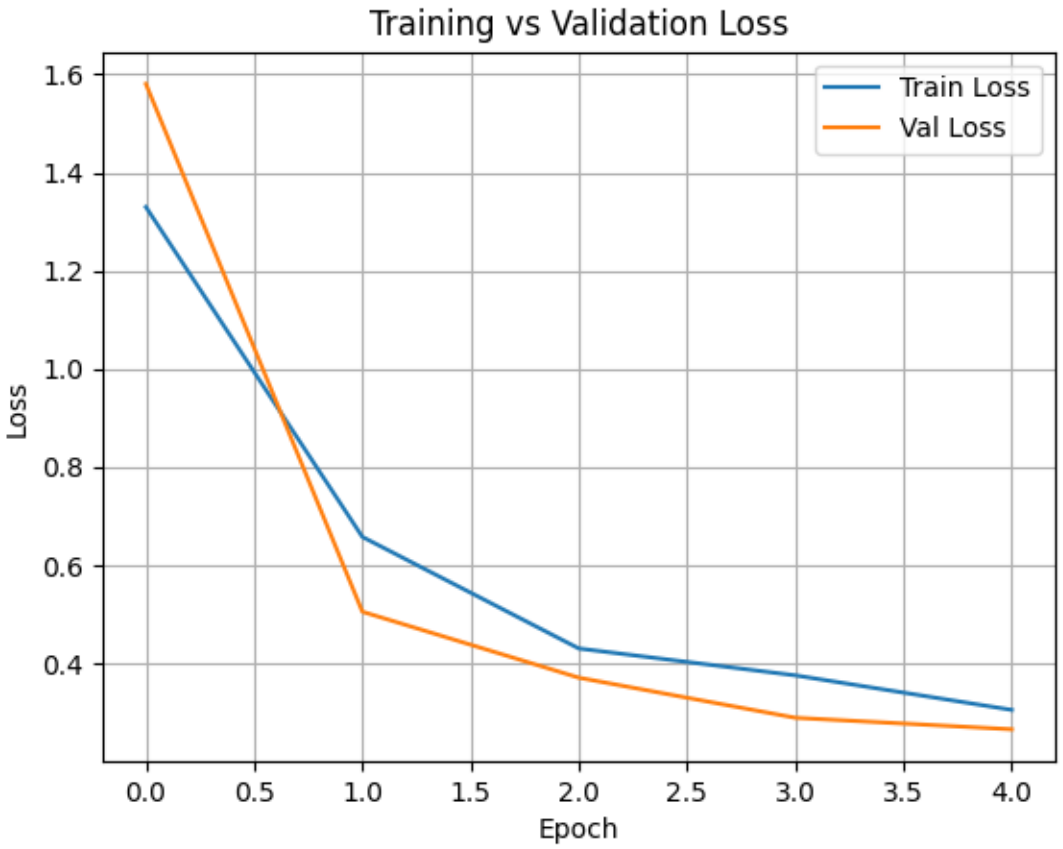
Evaluation – Object Detection (Faster R-CNN)
The evaluation of the object detection component was performed using the trained Faster R-CNN model. The model was set to evaluation mode to ensure deterministic behavior and disable training-specific layers such as dropout. Each test image was passed through the detector, which predicted a set of bounding boxes along with associated confidence scores and class labels. To simplify the pipeline, only the most confident bounding box per image was selected. If no object was detected, the image was skipped. The selected bounding boxes were then used to crop the detected regions from the original images. These cropped regions were saved and later used as input to the classification stage. This evaluation step verified the detector's ability to localize relevant regions in unseen data and served as a critical pre-processing step for the classification task.

Evaluation – Image Classification
After the detection and cropping phase, the resulting images were passed to the trained image classifier for evaluation. The classifier was also switched to evaluation mode, and predictions were made on a validation set using a standard cross-entropy loss function. For each batch of validation images, the model output class probabilities, from which the predicted class was extracted. The predicted labels were compared to the ground truth to compute accuracy.

Both the average validation loss and accuracy were calculated across the entire dataset. This evaluation allowed for a quantitative assessment of the classifier's generalization performance on unseen data and confirmed whether the model was effectively learning to distinguish between the defined categories.

5. Experimental result

Now, let's start to train the classification model and see the results.



Epoch	Training loss	Validation loss	Training accuracy	Validation ccuracy
1	1.33	1.58	49%	37%
2	0.65	0.50	74%	79%
3	0.43	0.37	80%	85%
4	0.37	0.28	84%	88%
5	0.30	0.26	90%	92%

As we can see, there is no problem; both losses are going down steadily as the accuracy rises.

5.1 Performance

Now it’s time to use the detector on the test images and run the classifier model on the output.

We can see some of the detector model outputs below:

The accuracy of the classification model on the test dataset is 72%, as recorded in the Kaggle competition.

6. Conclusion

This project successfully implemented a two-stage deep learning pipeline that combines object detection and image classification. A Faster R-CNN model was used to detect and localize relevant objects within input images, followed by a custom convolutional classifier that assigned each detected region to one of eight predefined categories.

Relying solely on a simple image classifier would not have been sufficient, as the original input images often contain complex scenes with varying backgrounds, scales, and irrelevant content. Without a detection phase, the classifier would be forced to make predictions based on the entire image, including areas that may not contain the object of interest. This would likely reduce classification accuracy and lead to unreliable predictions.

By introducing an object detector as a first step, the pipeline focuses the classification process only on the relevant cropped regions, significantly improving performance and robustness. The modular structure of the system also allows for greater flexibility, enabling independent optimization of the detection and classification stages.

Overall, the results demonstrate the effectiveness of combining object detection with classification in tasks that require both spatial localization and semantic understanding, providing a strong foundation for further development and real-world applications

Appendix

a. Remove background

i. training dataset

This function starts by ensures the output_folder exists and creates any necessary parent directories.

It loops through any folder existing in the input_folder and removes their background and saved them in a new folder.

```
def remove_background_from_folder(input_folder, output_folder):
    Path(output_folder).mkdir(parents=True, exist_ok=True)
    for subfolder in os.listdir(input_folder):
        input_subfolder_path = os.path.join(input_folder, subfolder)
        output_subfolder_path = os.path.join(output_folder, subfolder)
        if not os.path.isdir(input_subfolder_path):
            continue

        Path(output_subfolder_path).mkdir(parents=True, exist_ok=True)
        for filename in os.listdir(input_subfolder_path):
            if not filename.lower().endswith(('.png', '.jpg', '.jpeg', '.bmp',
'.gif')):
                continue

            input_path = os.path.join(input_subfolder_path, filename)
            output_filename = os.path.splitext(filename)[0] + '.jpg'
            output_path = os.path.join(output_subfolder_path, output_filename)

            try:
                input_image = Image.open(input_path)
                output_image = remove(input_image)
                output_image.save(output_path)
                print(f"Processed: {input_path} -> {output_path}")
            except Exception as e:
                print(f"Error processing {input_path}: {str(e)}")

input_folder = "/content/d2_2025/train"
output_folder = "/content/d2_2025/train_no_background"
remove_background_from_folder(input_folder, output_folder)
```

ii. test dataset

```
def remove_background_from_folder(input_folder, output_folder):
    Path(output_folder).mkdir(parents=True, exist_ok=True)

    for filename in os.listdir(input_folder):
        if not filename.lower().endswith(('.png', '.jpg', '.jpeg', '.bmp',
'.gif')):
            continue

        input_path = os.path.join(input_folder, filename)
        output_filename = os.path.splitext(filename)[0] + '.jpg'
        output_path = os.path.join(output_folder, output_filename)

        try:
            input_image = Image.open(input_path)
            output_image = remove(input_image)
            output_image.save(output_path)
            print(f"Processed: {input_path} -> {output_path}")
        except Exception as e:
            print(f"Error processing {input_path}: {str(e)}")
input_folder = "test"
output_folder = "test_noback"
remove_background_from_folder(input_folder, output_folder)
```

b. DetectorDataset

```
class DetectorDataset(Dataset):
    def __init__(self, csv_path, images_dir):
        df = pd.read_csv(csv_path)
        self.images_dir = images_dir
        self.items = df.values.tolist()

    def __getitem__(self, idx):
        row = self.items[idx]
        img_path = os.path.join(self.images_dir, row[0])
        img = Image.open(img_path).convert("RGB")
        box = torch.tensor([[row[1], row[2], row[3], row[4]]],
dtype=torch.float32)
        label = torch.tensor([1], dtype=torch.int64)
        return TF.to_tensor(img), {"boxes": box, "labels": label}

    def __len__(self):
        return len(self.items)
```

The DetectorDataset loads images and their corresponding bounding boxes from a CSV file, prepares them in a format required by object detection model, FasterRCNN, and returns the image converted to a PyTorch tensora and a dictionary containing:

"boxes": The bounding box tensor

"labels": The class label tensor

c. get_detector()

Used in predicting bounding boxes for the test dataset. defines and returns a custom Faster R-CNN object detection model using a ResNet-50 backbone and custom anchor settings.

Without anchors, the network would have to predict absolute box coordinates, which is significantly more challenging.

```
def get_detector():
    anchor_generator = AnchorGenerator(
    sizes=((32,), (64,), (128,), (256,), (512,)),
    aspect_ratios=((0.5, 1.0, 2.0),) * 5
    )
    model = fasterrcnn_resnet50_fpn(pretrained=False,
rpn_anchor_generator=anchor_generator)
    in_features = model.roi_heads.box_predictor.cls_score.in_features
    model.roi_heads.box_predictor = FastRCNNPredictor(in_features, 2)
    return model
```

d. train_detector()

This function trains the object detector model (Faster R-CNN) using the training dataset , and then saves the trained model to disk.

```
def train_detector():
    dataset = DetectorDataset(CSV_PATH, IMAGES_DIR)
    dataloader = DataLoader(dataset, BATCH_SIZE, shuffle=True,
collate_fn=lambda x: tuple(zip(*x)))
    model = get_detector().to(DEVICE)
    optimizer = torch.optim.Adam(model.parameters(), lr=1e-4)

    model.train()
    for epoch in range(EPOCHS):
        for imgs, targets in tqdm(dataloader, desc=f"Detector Epoch
```

```
{epoch+1}"):
            imgs = [img.to(DEVICE) for img in imgs]
            targets = [{k: v.to(DEVICE) for k, v in t.items()} for t in
targets]
            loss_dict = model(imgs, targets)
            losses = sum(loss for loss in loss_dict.values())

            optimizer.zero_grad()
            losses.backward()
            optimizer.step()
```

```
def detect_and_crop(model, test_dir, output_dir):
    if os.path.exists(output_dir):
        shutil.rmtree(output_dir)
    os.makedirs(output_dir)

    model.eval()
    model.to(DEVICE)

    transform = transforms.Compose([
        transforms.Resize((IMG_SIZE, IMG_SIZE)),
        transforms.ToTensor()
    ])

    filenames = sorted(os.listdir(test_dir))
    for fname in tqdm(filenames, desc="Detecting test set"):
        img_path = os.path.join(test_dir, fname)
        img = Image.open(img_path).convert("RGB")
        img_tensor = F.to_tensor(img).to(DEVICE)

        with torch.no_grad():
            pred = model([img_tensor])[0]
            if len(pred['boxes']) == 0:
```

```
                continue
            box = pred['boxes'][0].cpu().int().tolist()
            crop = img.crop((box[0], box[1], box[2], box[3]))
            crop = transform(crop)
            base_name = os.path.splitext(fname)[0]
            save_path = os.path.join(output_dir, f"{base_name}.jpg")
            transforms.ToPILImage()(crop).save(save_path)
```

f. ClassifierDataset(Dataset)

To crop object regions from images (based on bounding box coordinates in a CSV), apply transforms, and return image–label pairs for classification.

```
class ClassifierDataset(Dataset):
    def __init__(self, csv_path, images_dir, transform):
        df = pd.read_csv(csv_path)
        self.images_dir = images_dir
        self.items = df.values.tolist()
        self.transform = transform

    def __getitem__(self, idx):
        row = self.items[idx]
        img_path = os.path.join(self.images_dir, row[0])
        img = Image.open(img_path).convert("RGB")
        box = [int(v) for v in row[1:5]]
        label = int(row[5])
        img = img.crop((box[0], box[1], box[2], box[3]))
        img = self.transform(img)
        return img, label

    def __len__(self):
        return len(self.items)
```

g. SimpleClassifier(nn.Module)

This is a simple convolutional neural network (CNN) built using PyTorch's nn.Module.

```
class SimpleClassifier(nn.Module):
    def __init__(self, num_classes=8):
        super().__init__()
        self.model = nn.Sequential(
            nn.Conv2d(3, 16, 3, padding=1),
            nn.BatchNorm2d(16),
            nn.ReLU(),
            nn.MaxPool2d(2),

            nn.Conv2d(16, 32, 3, padding=1),
            nn.BatchNorm2d(32),
            nn.ReLU(),
            nn.MaxPool2d(2),

            nn.Conv2d(32, 64, 3, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.MaxPool2d(2),

            nn.Flatten(),
            nn.Linear(64 * (IMG_SIZE // 8) * (IMG_SIZE // 8), 128),
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.Linear(128, num_classes)
        )
```

h. train_classifier()

Trains the SimpleClassifier neural network on cropped object images.

```
def train_classifier():
    base_df = pd.read_csv(CSV_PATH)
    unique_images = base_df['image'].unique()
    train_imgs, val_imgs = train_test_split(
        unique_images, test_size=0.2, random_state=42
```

```
train_df = base_df[base_df['image'].isin(train_imgs)]
    val_df = base_df[base_df['image'].isin(val_imgs)]
    train_df.to_csv("train_temp.csv", index=False)
    val_df.to_csv("val_temp.csv", index=False)

    train_transform = transforms.Compose([
        transforms.Resize((IMG_SIZE, IMG_SIZE)),
        transforms.RandomHorizontalFlip(),
```

```
            total += labels.size(0)
            correct += predicted.eq(labels).sum().item()
        train_losses.append(running_loss / len(train_loader))
        print(f"[Train] Epoch {epoch+1}: Loss={train_losses[-1]:.4f},
Acc={correct/total:.4f}")

        model.eval()
        val_loss = 0.0
        val_correct, val_total = 0, 0
        with torch.no_grad():
            for imgs, labels in val_loader:
                imgs, labels = imgs.to(DEVICE), labels.to(DEVICE)
                outputs = model(imgs)
                loss = criterion(outputs, labels)
                val_loss += loss.item()
                _, predicted = outputs.max(1)
                val_total += labels.size(0)
                val_correct += predicted.eq(labels).sum().item()
        val_losses.append(val_loss / len(val_loader))
        val_accuracies.append(val_correct/val_total)
        print(f"[Val] Epoch {epoch+1}: Loss={val_losses[-
1]:.4f},Acc={val_accuracies[-1]:.4f}")

    plt.plot(train_losses, label="Train Loss")
    plt.plot(val_losses, label="Val Loss")
    plt.xlabel("Epoch")
    plt.ylabel("Loss")
    plt.legend()
    plt.title("Training vs Validation Loss")
    plt.grid(True)
    plt.savefig("/content/loss_curve.png")
    plt.show()

    torch.save(model.state_dict(), MODEL_PATH_class)
    return model
```

i. generate_submission(classifier_model)

This function generates and save the prediction result of the test data set.

```
def generate_submission(classifier_model):
    transform = transforms.Compose([
        transforms.Resize((IMG_SIZE, IMG_SIZE)),
        transforms.ToTensor()
    ])

    classifier_model.eval()
    classifier_model.to(DEVICE)

    results = []
    filenames = sorted(os.listdir("/content/test_crops"))
    for fname in filenames:
        path = os.path.join("test_crops", fname)
        img = Image.open(path).convert("RGB")
        img = transform(img).unsqueeze(0).to(DEVICE)

        with torch.no_grad():
            output = classifier_model(img)
            pred = torch.argmax(output, dim=1).item()

        results.append((fname, pred))

    df = pd.DataFrame(results, columns=["Id", "Category"])
    df = df.drop_duplicates(subset="Id")
    df.to_csv(SUBMISSION_PATH, index=False)
```