AIN4311 Special Topics in AI

Kaan Yalman 2104371

Contents

1 Introduction

1.1 Motivation

The main motivation of this work is to build an AI model that can automatically classify bean leaves into three categories: **healthy**, **bean rust**, or **angular leaf spot**. The purpose is to support early disease detection in agriculture, helping farmers and researchers quickly identify infected plants and take preventive measures.

To achieve this, I used the pretrained model `google/vit-base-patch16-224-in21k`, a Vision Transformer (ViT) model developed by Google. The model was fine-tuned using a bean leaf image dataset to adapt it specifically for this classification task. Several preprocessing and fine-tuning techniques were applied to improve the model's accuracy and generalization, including image normalization, data augmentation, and learning rate optimization.

1.2 Dataset

The dataset used in this project is the **Beans Dataset**, which contains images of bean leaves belonging to three classes:

- **Healthy beans**
- **Bean rust**
- **Angular leaf spot**

Each image is associated with a corresponding label indicating the disease condition of the bean leaf. The dataset is organized into three subsets:

- **Training set:** 1034 samples
- **Validation set:** 133 samples
- **Test set:** 128 samples

This structure ensures that the model can learn effectively from the training data and generalize well to unseen examples during validation and testing.

```
DatasetDict({
    train: Dataset({
        features: ['image_file_path', 'image', 'labels'],
        num_rows: 1034
    })
    validation: Dataset({
        features: ['image_file_path', 'image', 'labels'],
        num_rows: 133
    })
    test: Dataset({
        features: ['image_file_path', 'image', 'labels'],
        num_rows: 128
    })
})
```

Figure 1: Sample images from the Beans Dataset showing healthy, rust, and angular leaf spot beans.

1.3 Model

In this project, I used the **Vision Transformer (ViT)** model from Hugging Face, specifically the pretrained model `google/vit-base-patch16-224-in21k`. This model was originally trained on the large ImageNet-21k dataset and is well-suited for image classification tasks.

I fine-tuned the pretrained ViT model on the Beans Dataset to classify images into three categories: healthy, bean rust, and angular leaf spot. The implementation and training were done using the Hugging Face `transformers` library and the PyTorch framework.

2 Implementation

2.1 Requirements

This project was implemented in **Google Colab**, which provides a Python environment with GPU support to accelerate model training. For most experiments, I used a **T4 GPU**. However, since I have Colab Pro, I was sometimes able to use an **A100 GPU**, which is one of the most powerful GPUs available and significantly reduces training time.

The following Python packages and tools were required:

- **datasets** — for loading and handling the Beans Dataset
- **transformers** — for using and fine-tuning the Vision Transformer model
- **evaluate** — for computing evaluation metrics
- **torch** and **torchvision** — for deep learning and image processing
- **scikit-learn** — for calculating the confusion matrix and accuracy
- **opencv-python Pillow** — for image manipulations

The packages were installed using the following commands in a Colab notebook:

```
!pip install datasets transformers evaluate torch torchvision
!pip install transformers[torch]
!git clone https://huggingface.co/spaces/avsarasan/imageclassification
```

These installations ensured that all necessary dependencies for model training, evaluation, and data handling were available.

2.2 Setup

In this section, the environment and dataset were prepared for fine-tuning the Vision Transformer model. The following Python libraries were imported:

```
import torch
import numpy as np
from datasets import load_dataset
from transformers import AutoImageProcessor, AutoModelForImageClassification, TrainingArguments, Tra
from torchvision.transforms import RandomResizedCrop, Compose, Normalize, ToTensor, InterpolationMod
!pip install evaluate
import evaluate
import matplotlib.pyplot as plt
import cv2
from PIL import Image
```

The `beans` dataset was loaded directly from the Hugging Face Datasets library:

```
dataset = load_dataset("beans")
```

Next, the labels were extracted, and mappings were created between label names and their numeric IDs:

```
labels = dataset["train"].features["labels"].names
num_classes = len(labels)

label2id = {label: str(i) for i, label in enumerate(labels)}
id2label = {str(i): label for i, label in enumerate(labels)}
```

An example image from each class was visualized using Matplotlib:

```
first_images = {label: None for label in labels}

for example in dataset["train"]:
    label = labels[example["labels"]]
    if first_images[label] is None:
        first_images[label] = example
    if all(v is not None for v in first_images.values()):
        break

plt.figure(figsize=(12, 4))
for i, (label, example) in enumerate(first_images.items()):
    plt.subplot(1, 3, i+1)
    plt.imshow(example["image"])
    plt.axis("off")
    title = label.replace("_"," ").capitalize()
    plt.title(f"Label: {title}")
plt.show()
```

Figure 2: Visualization of one example from each class in the Beans Dataset: healthy, bean rust, and angular leaf spot.

2.3 Training Process

The training process involved significant data preprocessing and augmentation to improve model robustness and prevent overfitting.

First, the image processor's normalization values and target size were extracted. These values are essential for ensuring the input data matches what the model expects.

```
image_processor = AutoImageProcessor.from_pretrained(model_checkpoint)
normalize = Normalize(mean=image_processor.image_mean, std=image_processor.image_std)
size = image_processor.size["height"]
```

For the **training dataset**, a combination of augmentation methods was used to increase accuracy and reduce loss. After testing various methods (like `CenterCrop`, `RandomAffine`, and blurring), the following combination yielded the best results.

The `RandomResizedCrop` function randomly crops a part of the image, rescales it (e.g., to 4/3 or 3/4 of its size), and then resizes it to the model's required input size (224x224). This increases the variety of samples and prevents the model from overfitting. For the resizing interpolation, `InterpolationMode.LANCZOS` was used, as Lanczos interpolation is one of the best methods for high-quality image resizing.

Additionally, `RandomHorizontalFlip` is applied with a 50

```
train_transforms = Compose([
    RandomResizedCrop(
        size,
        interpolation=InterpolationMode.LANCZOS
    ),
    RandomHorizontalFlip(),
```

```
    ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2, hue=0.1),
    ToTensor(),
    normalize
])
```

For the **evaluation dataset**, augmentations are not used. This is essential for detecting the model's pure performance on real-life data. The evaluation set should reflect the kind of images the model will see in production.

Therefore, the evaluation transforms only resize and crop the image. It is fundamental to preserve the aspect ratio during this process; otherwise, the image will look stretched horizontally or vertically, which can degrade performance. A custom `resize_lanczos` function was used to ensure high-quality resizing, and `CenterCrop` was applied to get the final 224x224 image.

```
def resize_lanczos(image, size):
  if not isinstance(image, np.ndarray):
    image = np.array(image)
  resize_image = cv2.resize(image, (size,size), interpolation=cv2.INTER_LANCZOS4)
  return Image.fromarray(resize_image)

aspect_ratio = int((256/224) * size)
eval_transforms=Compose([
    lambda x: resize_lanczos(x, aspect_ratio),
    CenterCrop(size),
    ToTensor(),
    normalize
])
```

These transforms were applied to the datasets using `.with_transform()`. For each sample in a batch, the image is first converted to RGB, and then the corresponding transformation (training or evaluation) is applied. The resulting tensor is stored in the `"pixel_values"` key.

```
def preprocess_train(example_batch):
  example_batch["pixel_values"]=[train_transforms(image.convert("RGB"))
  for image in example_batch["image"]]
  return example_batch

def preprocess_eval(example_batch):
  example_batch["pixel_values"]=[eval_transforms(image.convert("RGB"))
  for image in example_batch["image"]]
  return example_batch

train_ds = dataset['train'].with_transform(preprocess_train)
eval_ds = dataset['validation'].with_transform(preprocess_eval)
test_ds = dataset['test'].with_transform(preprocess_eval)
```

A `collate_fn` is used by the `Trainer` to batch examples together. This function stacks the individual image tensors (from `"pixel_values"`) into a single batch tensor with the shape `(batch_size, channels, height, width)`. It also converts the list of labels into a tensor.

```
def collate_fn(examples):
  pixel_values=torch.stack([example["pixel_values"]for example in examples])
  labels=torch.tensor([example["labels"] for example in examples])
  return {"pixel_values":pixel_values, "labels":labels}
```

To compute metrics during evaluation, the `evaluate` library from Hugging Face was used. The `compute_metrics` function takes the `eval_pred` (which contains model predictions and true labels) and calculates the accuracy. It first finds the predicted class by taking the `argmax` of the prediction logits (the index with the maximum value) and then compares these predictions to the true labels.

```
accuracy_metric=evaluate.load("accuracy")
def compute_metrics(eval_pred):
```

```
  predictions, labels = eval_pred
  predictions=np.argmax(predictions, axis=1)
  return accuracy_metric.compute(predictions=predictions, references=labels)
```

Finally, the `TrainingArguments` were defined to control the training process, setting parameters like the number of epochs, batch size, warmup steps, logging strategy, and evaluation strategy. The `Trainer` class was then initialized, bringing together the model, training arguments, datasets, image processor (tokenizer), collate function, and metrics function.

```
training_args = TrainingArguments(
    output_dir="./vit_finetuned_beans",
    num_train_epochs=3,
    per_device_train_batch_size=16,
    per_device_eval_batch_size=16,
    warmup_steps=500,
    weight_decay=0.01,
    logging_dir='./logs',
    logging_steps=100,
    eval_strategy="epoch",
    save_strategy="epoch",
    load_best_model_at_end=True,
    report_to="none",
    remove_unused_columns=False
)

trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_ds,
    eval_dataset=eval_ds,
    tokenizer=image_processor,
    data_collator=collate_fn,
    compute_metrics=compute_metrics,
)

print("Starting Fine-Tuning...")
trainer.train()
```

2.4 Results

After the training, the model was evaluated on the validation set (`eval_ds`).

```
results = trainer.evaluate(eval_ds)
print(f"Final Evaluation Results: {results}")
```

The final validation results showed high performance:

```
 [9/9 00:01]
Final Evaluation Results: {
    'eval_loss': 0.1874169409275055,
    'eval_accuracy': 0.9849624060150376,
    'eval_runtime': 2.3373,
    'eval_samples_per_second': 56.903,
    'eval_steps_per_second': 3.851,
    'epoch': 3.0
}
```

2.5 Test Set Evaluation

To confirm the model's generalization, it was evaluated on the held-out **test dataset**. The predictions were used to generate a confusion matrix to visualize the model's performance on each class.

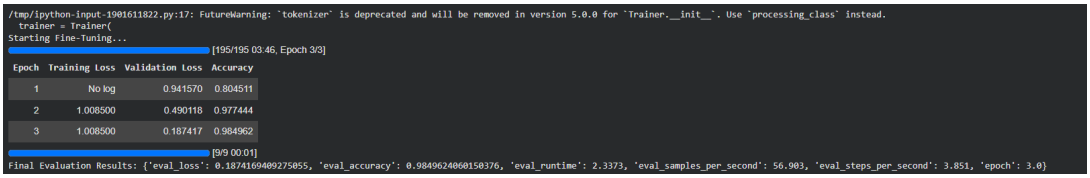

Figure 3: Training and Validation Accuracy/Loss curves (placeholder for accuracy image).

```
predictions = trainer.predict(test_ds)
y_true = predictions.label_ids
y_pred = np.argmax(predictions.predictions, axis=1)

BEANS_LABELS = ['angular_leaf_spot', 'bean_rust', 'healthy']
cm = confusion_matrix(y_true, y_pred, labels=[0, 1, 2])

disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=BEANS_LABELS)
disp.plot(cmap=plt.cm.Blues)
plt.title("Confusion Matrix")
plt.show()
```

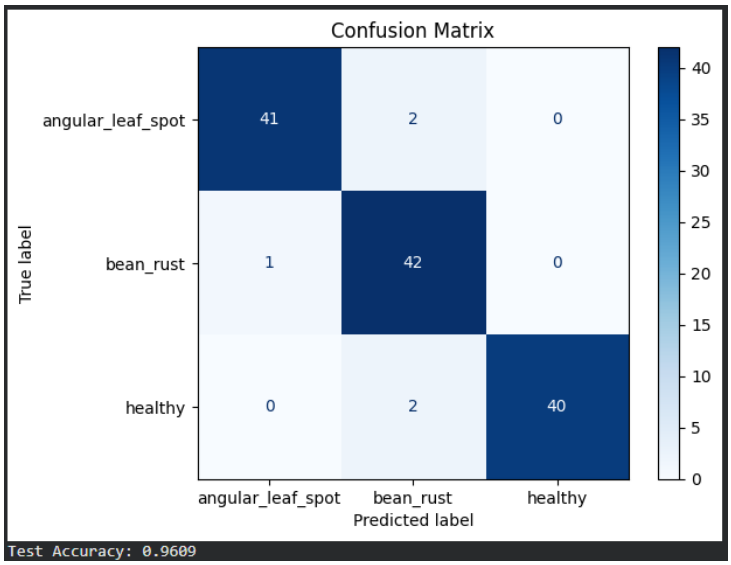

Figure 4: Confusion Matrix on the Test Set (placeholder for CM image).

The final accuracy on the test set was calculated:

```
acc = accuracy_score(y_true, y_pred)
print(f"Test Accuracy: {acc:.4f}")
```

The resulting test accuracy confirmed the model's high performance.

2.6 Model Saving

After successful training and evaluation, the fine-tuned model and its corresponding image processor were saved to disk for future use.

```
trainer.save_model("./vit_finetuned_beans")
image_processor.save_pretrained("./vit_finetuned_beans")
```

2.7 Discussion

The choice of data augmentation and preprocessing methods was critical to the model's success.

- **Interpolation Method:** The **Lanczos** interpolation method was chosen for resizing as it provides high-quality results for down-sampling.
- **Training Augmentation:** The combination of `RandomResizedCrop`, `RandomHorizontalFlip`, and `ColorJitter` was effective in preventing overfitting and improving generalization.
- **Evaluation Preprocessing:** No augmentation was used during evaluation to measure the model's pure performance on unseen data.
- **Aspect Ratio:** Preserving the aspect ratio during evaluation resizing was fundamental to prevent image distortion, which could confuse the model.

3 Inference Pipeline

To use the model in a real-life scenario, a Hugging Face `pipeline` was created. This encapsulates all the steps required for inference: loading the model and processor, preprocessing an image, passing it to the model, and post-processing the results to return a human-readable label.

First, the saved model and image processor are loaded from the directory.

```
from transformers import ViTForImageClassification, ViTImageProcessor, pipeline
from PIL import Image
from IPython.display import display
import torch

MODEL_DIR = "./vit_finetuned_beans"
BEANS_LABELS=['angular_leaf_spot', 'bean_rust', 'healthy']

model = ViTForImageClassification.from_pretrained(
    MODEL_DIR,
    label2id = {label:i for i, label in enumerate(BEANS_LABELS)},
    id2label = {i:label for i, label in enumerate(BEANS_LABELS)}
  )

image_processor = ViTImageProcessor.from_pretrained(MODEL_DIR)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)
```

Next, the `pipeline` is initialized for image classification.

```
classifier = pipeline(
    task="image-classification",
    model=model,
    feature_extractor=image_processor,
    device=device
)
```

Finally, the pipeline is tested on a single image from the test set.

```
example = dataset["test"][0]
image = example["image"]

print("TEST IMAGE:")
display(image)

results = classifier(image)

print("\n\n--- CLASSIFICATION RESULT ---")
top_result = results[0]

top_result_label = top_result["label"].replace("_", " ").upper()
```

```
print(f"Prediction: **{top_result_label}**")
print(f"Confidence: {top_result['score']:.4f}")

print("\nAll Results:")
for result in results:
    label = result["label"].replace("_", " ").capitalize()
    print(f"- {label:<20}: {result['score']:.4f}")
```

The pipeline correctly identifies the image class and provides a confidence score, demonstrating its readiness for practical application.

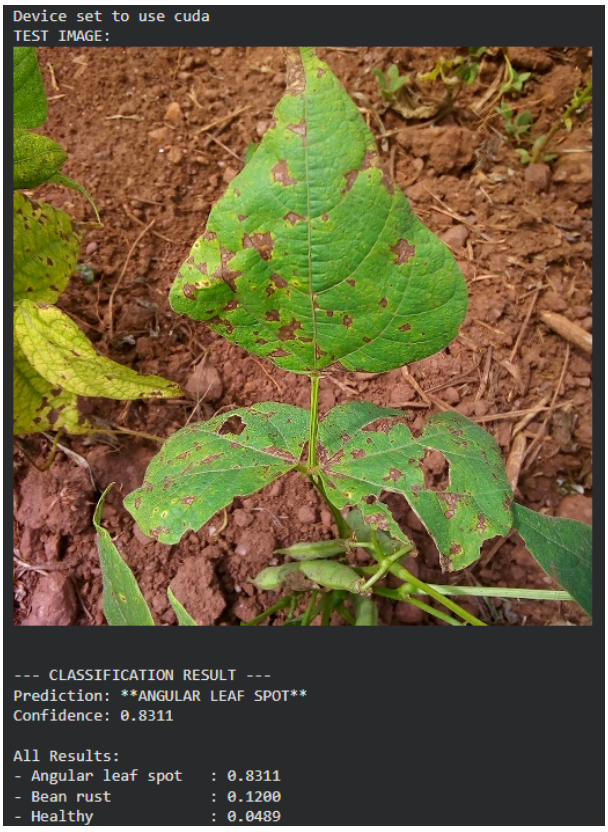

Figure 5: A single test image used for pipeline inference (placeholder).

4 Conclusion

In the provided code, instead of using the dataset's validation set, the training set was split into training and validation subsets. This approach can be problematic because when the model sees more samples during training, it better understands and retains the distinguishing features. Therefore, in this project, I used the original validation set provided in the dataset to ensure a fair evaluation.

To better understand the dataset and the problem, I visualized samples from each class. Plotting these images helped identify patterns and gain insights into the characteristics of healthy leaves, bean rust, and angular leaf spot.

During preprocessing, instead of using the simple `BILINEAR` resizing in `RandomResizedCrop`, I chose the **Lanczos** interpolation method, which is more powerful and produces higher-quality resized images. In addition to resizing, I applied augmentations such as horizontal flipping and `ColorJitter` to improve the model's robustness.

Selecting the best combination of augmentations was challenging. I experimented with horizontal flipping, color adjustments, blurring, and cropping (center vs. random). After extensive evaluation, I concluded that the chosen combination—RandomResizedCrop with Lanczos, horizontal flipping, and color jitter—provided the best preprocessing for this dataset.

After training, the model was tested on the held-out test set, and a confusion matrix was plotted to analyze its performance. The results indicate that with more data, the model could better distinguish between classes, especially for subtle differences between angular leaf spot and rust. The confusion matrix shows that the model sometimes confuses angular leaf spot with rust and rust with healthy leaves. Although not explored in this work, using techniques such as k-fold cross-validation could improve accuracy by exposing the model to a larger variety of samples during training.

Overall, the fine-tuned Vision Transformer achieved high accuracy and demonstrates the potential for automated disease detection in bean leaves. Further improvements could be obtained by expanding the dataset or applying additional ensemble or cross-validation strategies.

5 Google Colab Notebook

The code and implementation for this project can be accessed on Google Colab at the following link: Open Colab Notebook