ISYE 6740 - Spring 2024
Classification of Foliar Diseases in Apple Trees
Proposal

Zimeiyi Wang
GTID: 903955209

Problem Statement

Crop protection and precision agriculture stand as essential pillars ensuring the sustainability and productivity of agricultural systems amidst the escalating global food demand and environmental challenges. Crop protection strategies are designed to shield crops from the adverse impacts of pests, diseases, and weeds, all of which pose significant threats to yield and quality if left unaddressed. Historically, crop protection heavily relied on widespread chemical applications, raising concerns about environmental contamination, the development of resistance in pests and pathogens, and adverse effects on human health.

In contrast, precision agriculture harnesses state-of-the-art technologies like remote sensing, data analytics, and artificial intelligence to optimize resource utilization and decision-making processes at the field level. By enabling farmers to fine-tune management practices with exceptional precision, precision agriculture minimizes input wastage, maximizes yield potential, and mitigates environmental impacts.

In light of this backdrop, this project places emphasis on early disease detection, a process traditionally reliant on manual scouting by crop experts, characterized by its time-consuming nature, high costs, and susceptibility to errors. Inaccurate or delayed diagnoses can lead to excessive or insufficient chemical usage, resulting in increased production costs and heightened environmental and health risks.

In recent years, the integration of digital imaging and machine learning has showcased significant promise in accelerating plant disease diagnosis. This innovative approach combines human expertise with machine learning algorithms to identify patterns and relationships in visual data for grouping and classification. Typically, images are collected along with metadata, which are then annotated by experts to train machine learning models. Once trained, these models enable automatic identification of unknown images.

This project centers around an open dataset comprising images of apple foliar disease symptoms observed during the 2019 growing season from commercially grown cultivars in an unsprayed apple orchard at Cornell AgriTech in Geneva, New York, USA. The objective is to train machine learning models capable of:

- Classifying images from the testing dataset into different disease categories or healthy leaves
- Distinguishing between multiple diseases, sometimes occurring concurrently on a single leaf, while considering depth perception factors such as angle, light, shade, and the physiological age of the leaf

Data

The expert-annotated pilot dataset of apple leaves was sourced from a Kaggle competition known as the Plant Pathology Challenge (https://www.kaggle.com/c/plant-pathology-2020-fgvc7), comprising 3642 meticulously curated RGB images. These images exhibit:

- healthy apple leaves
- apple rust (characterized by early symptoms manifesting as small, light yellow spots on leaves, which later expand and transition into bright orange patches

- apple scab (initially appearing as black or olive-brown lesions protruding from the leaf's adaxial surface, subsequently evolving into chlorotic sporulating lesions
- leaves with complex diseases

Photos were captured using a Canon Rebel T5i DSLR and smartphones, under diverse conditions encompassing illumination, angle, surface variations, noise, and differing maturity stages of the leaves.

On the official website, the dataset has been split equally into training (1821 images) and testing set (1821 images). However, I only have access to the labels of the training images because those of the testing set were kept by the competition organizer for evaluating participants' code performance.

Figure 1: Examples of the dataset. **Top-left**: a healthy leaf; **top-right**: a leaf with multiple dieases (rust and scab); **bottom-left**: a leaf with apple rust; **bottom-right**: a leave with apple scab.

In the training dataset, the label frequency distribution is shown in Figure 2 and Figure 3.

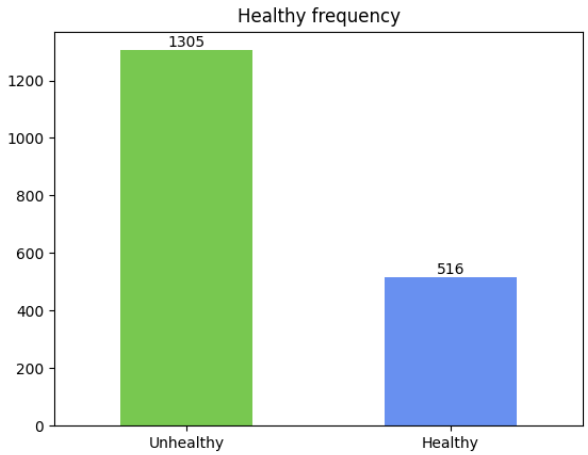

Figure 2: Distribution of healthy and unhealthy leaves

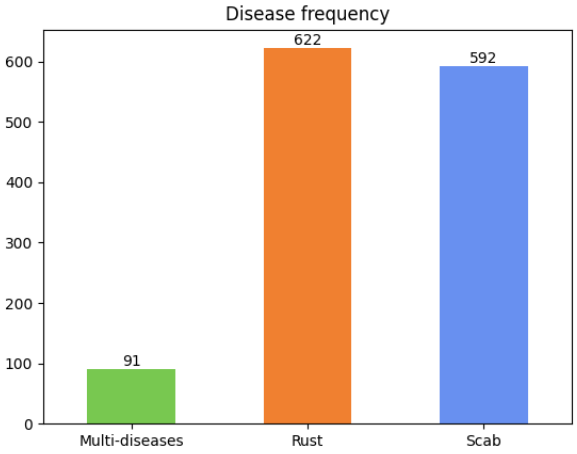

Figure 3: Distribution of foliar diseases

Method

I built a Convolutional Neural Network (CNN) to classify images into 4 classes.

Data Preprocessing

Preprocessing images for classification with a CNN involves several steps to ensure that the images are appropriately prepared for input into the model.

- Resizing: Since raw images in the data set have various dimensions, the project resized them to 256x256 pixels.
- Normalizing: The project normalized the pixel values of the images to a range between 0 and 1, by dividing the pixel values by 255. This could help the model converge faster during training and improves its stability.
- Splitting: We only have access to labels in the training set. For this project, we split the training into training, validation and testing sets with 70:15:15 ratio.

0.1 Image Augmentation

Due to the limited number of training examples, a high-capacity model, such as a CNN, can quickly overfit to the training set. Data augmentation is proven to be effective against overfitting. During each training step, I randomly flip, translate and rotate each input image independently. See Table 4 for a few examples:

Figure 4: Training examples with data augmentation. Each image is randomly flipped, translated and rotated.

Model architecture

Because I have limited computing power, I use a small CNN with 4 convolutional layers to reduce the resolution of the feature map, then use 2 fully connected layers and a softmax activation at the end to predict probabilities over 4 classes. Figure 5 illustrates the architecture design.

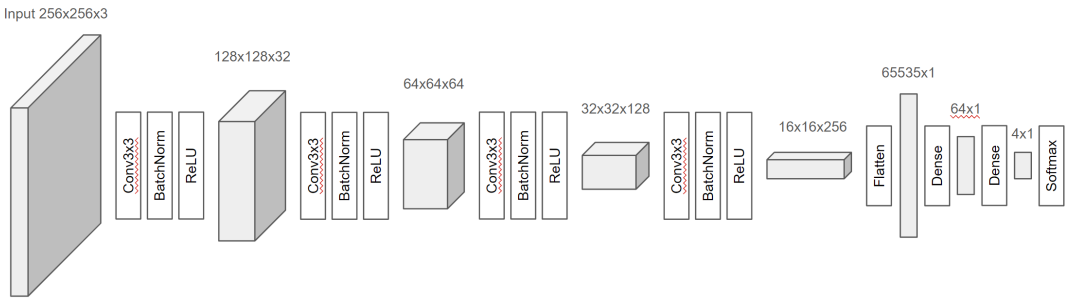

Figure 5: The model architecture.

Convolutional layers: The CNN begins with a series of convolutional layers, which serve as feature extractors. I followed the standard practice of reducing the resolution by half (using strided convolution) while doubling number of channels for each layer so that it can extract rich semantic feature at different resolutions. I also found that batch normalization layers are necessary for fast convergence. Lastly, a Rectified Linear Unit (ReLU) layer is used after the batch norm layer

Fully connected layers: Once the final feature map is extracted by convolutional layers, the model flattens then into a 1 dimensional vector and feeds it to 2 fully connected layers to model interactions between all features, and to produce the 4-class logits.

Output layer: The final layer of the CNN model utilizes a softmax activation function to normalize unbounded logits into probability distributions over the disease categories. The softmax function is defined as:

$$y_i = \frac{e^{x_i}}{\sum_{j=k}^{K} e^{x_j}} \tag{1}$$

Implementation

I implemented the training pipeline in Tensorflow 2.1.16. The pipeline includes multiple programs:

Data splitting: because I only have access to the training set, I manually split them into training, validation and testing set. This is done by first shuffling all rows in `label.csv` file, them splitting and writing each split into separate files `train.csv`, `val.csv` and `test.csv`. See `split_data.py` for the implementation.

Data processing: I implemented functions to load images based on `image_id` in the CSV file, normalize image values from [0, 255] to [0.0, 1.0] and pair the image with label. This function is called by `tf.data.Dataset` to process rows in the CSV into images and labels. See **data.py** for details.

Training: I followed the official Tensorflow documentation to implement a custom training loop. I tried to use the default training loop provided by Keras (i.e. `model.fit()`, but found it's not convenient for debugging. The pseudo-code is listed below:

```
# The training loop is based on the official documentation:
# https://www.tensorflow.org/guide/keras/writing_a_training_loop_from_scratch

# Construct the tf dataset. This loops over (image path, label)
# pairs parsed from the CSV file, load image from the image path
# and apply normalization.
```

```
train_dataset = tf.data.Dataset.from_tensor_slice((image_path, label))
train_datset = train_dataset.map(data.process_data).shuffle().batch(batch_size).
    repeat()

# Outer loop over epoch.
for epoch in range(num_epochs):

  # Inner loop over steps.
  for step, (x_train, y_train) in enumerate(train_dataset):

    # Run forward pass and compute gradients.
    with tf.gradient_tape() as tape:
      y_pred = model(x_train)
      train_loss = loss_fn(y_train, y_pred)

    gradients = tape.gradient(train_loss, model.trainable_weights)

    # Apply gradients to model parameters.
    optimizer.apply_gradients(zip(grads, model.trainable_weights))
```

Note that I do not load all images upfront to avoid out-of-memory. Instead, I loop over **image file paths**, and only load images of the current batch. In my experiment, this is necessary to use 256x256 resolution or above.

To monitor the training progress, I also run a validation loop at certain steps (e.g. once every 200 training steps) to obtain the up-to-date model performance. I use `CategoricalAccuracy` metric from the Keras library to keep track of the accuracy over the entire validation set.

Finally, I use `Tensorboard` to monitor the training process in real time and to easily compare different experiments. I created summary writers for training and validation separately so that we can compare the their losses and accuracy to monitor overfitting.

Please refer to `train.py` for the full implementation.

Testing: the training program saves model checkpoints periodically. I wrote a separate program to load a saved model checkpoint and run evaluation over the testing set. In addition to testing accuracy, the testing program also reports the confusion matrix and accuracy breakdowns for each class. These additional metrics can help us to understand the model quality.

Evaluation

All experiments are run on a laptop with AMD 5800X CPU and Nvidia 3050Ti GPU. We use batch size 64 and train for 200 epochs with the Adam optimizer.

Following the standard practice in deep learning, we decay the learning rate over the course of the training so that the model can converge. I used the cosine decay from the Keras library with the initial learning rate of 0.02. The learning schedule is shown in Figure 6.

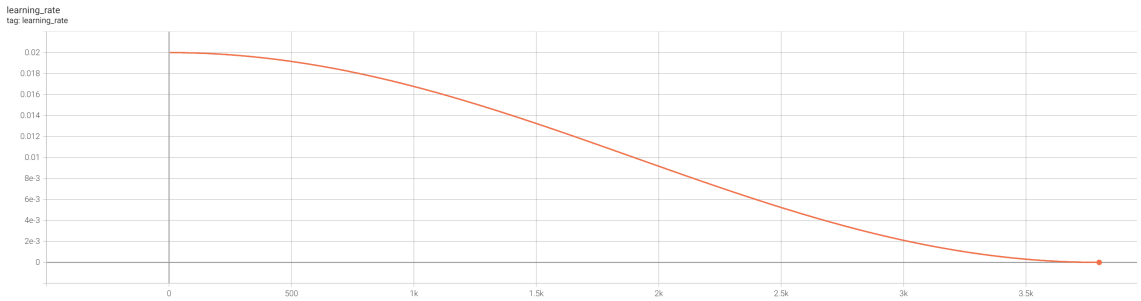

Figure 6: The learning rate decay.

Test accuracy

I selected the model with the best validation accuracy and evaluated its performance on a separate test set with 276 examples. The testing program is implemented in test.py.

The model achieves 96.5% training accuracy, 92.3% validation accuracy and 90.5% testing accuracy. The result suggests that a small CNN with only 4 convolutional layers is able to classify diseases in apple tree leaves with high accuracy and low overfitting.

Table 1 shows the confusion matrix as the number of examples and accuracy breakdowns for each label class. It is apparent that the model does not perform well on examples with multiple diseases. This can be explained by 2 factors: (1) the number of examples with multiple diseases is very low compared with other classes. (2) images with multiple diseases can be confused with images with one of these disease.

Label \ Prediction	Healthy	Multiple	rust	Scab	Accuracy
Healthy	70	0	0	7	90.9%
Multiple	3	2	5	4	14.2%
Rust	2	0	70	0	97.2%
Scab	5	0	0	75	93.7%

Table 1: Number of examples for each (label class, predicted class) pair. The last column shows the accuracy for each label class.

Ablation study

To verify the effectiveness of method, especially data augmentation and batch normalization, I trained models without them and evaluated their accuracy. The result is summarized in Table 2:

	Train accuracy	Val accuracy	Test accuracy
Baseline	96.5%	92.3%	90.5%
No augmentation	99.8%	44.7%	48.9%
No batch norm	36.4%	34.4%	35.1%

Table 2: Ablation study.

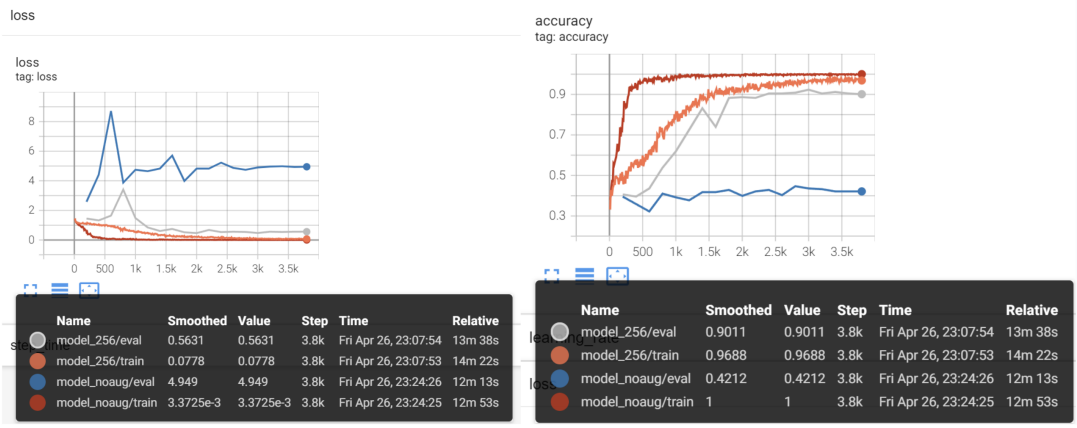

Figure 7: Loss and accuracy curves with and without data augmentation.

From these results we can observe:

- Without data augmentation, the training accuracy quickly converges to almost 100%, however both validation and testing accuracy are low, indicating severe overfitting. Figure 7 shows how the loss and accuracy curves evolve for the training and validation set. Notice that the training accuracy converges slower with data augmentation, but the gap between train and validation is much smaller than without augmentation.
- Batch normalization layers are critical for the training to work.

Failure examples

The trained model reported some misclassified cases, especially when distinguishing leaves with multiple diseases from those with single disease, and healthy leaves from ones with single disease.

- Mixing up multiple diseases with single disease: The model has difficulties to identify complex patterns. Multiple diseases can change how a leaf looks in complicated ways, it is challenging for the model to catch all the different signs and understand them correctly. It tends to focus too much on the most obvious symptoms but miss other subtle hints.

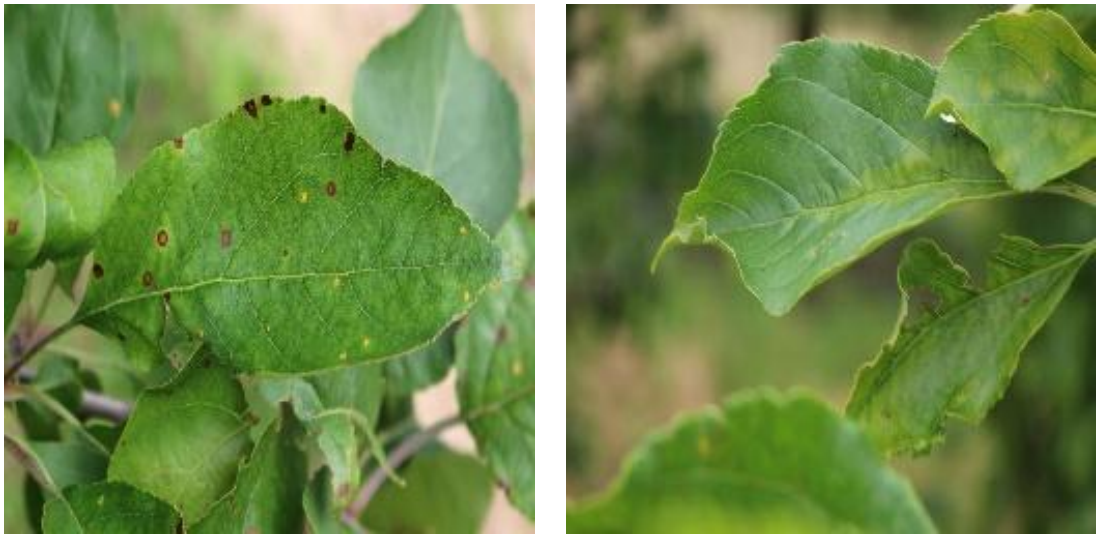

Figure 8: Misclassification example. **Left:** Misclassified multiple diseases as rust; **Right:** Misclassified multiple diseases as scab

- Mixing up healthy leaf with single disease: This might happen because it is hard for the model to tell the difference between a normal leaf and one that is just starting to show early symptoms of disease. This also could be attributed to environmental factors that affect the clarity of disease symptoms. In the field, natural conditions could be too unpredictable to capture high-quality images. The angle, distance and lighting when taking pictures may make the symptoms less obvious or make natural pigments look like unhealthy colors.

Figure 9: Misclassification example. **Left:** Misclassified healthy as scab; **Right:** Misclassified rust as healthy

To improve the model's performance, at the data collection stage, we can enrich the training data set with more images showing different status of leaves, especially instances of multiple diseases (Currently there are only 91 images in the data set. This imbalance might impact the learning process). It is also helpful to introduce quality control procedures to ensure the stability of image characteristics and minimize the impact of environmental variability.

When refining the model architecture, we can take the background information like environment and the plant's growth stage into consideration, while introducing more advanced feature extraction techniques to help it pick up the small differences.

In real-world practice, misclassification of foliar diseases underscores the emerging automatic system for precision farming, possibly leading to inappropriate treatments, wasted resources or even the further spread of disease. Apart from technical refinement, there is the need for interdisciplinary collaboration between data scientists, agronomists, and agricultural producers, who can incorporate their domain expertise into model development.