**INFERENCE DOCUMENT FOR DEEP LEARNING CT 3**

**MANJUSHA PA2312049010030**

**Question:** Assume you are a Data Scientist tasked with solving challenges in the agricultural industry, particularly focusing on the detection of diseases in crops. You have been given the responsibility to build a model that can classify images of beans into various categories to assist farmers in identifying disease-infected crops. You will apply transfer learning using TensorFlow and MobileNet to achieve this goal**.**

**Goal :** We are tasked with developing a transfer learning model for a real-world agricultural problem. Specifically, we are working on a bean classification problem, where we will be using a pre-trained MobileNet model to classify images of beans into different classes. The goal is to deploy the model in a user-friendly application to assist farmers in identifying different types of beans.

**DATASET LOADING**

To begin working with the TensorFlow Beans dataset and prepare it for a machine learning model, we need to load the dataset, inspect its structure, and determine any preprocessing steps required. The tensorflow\_datasets package allows us to easily load datasets like the Beans dataset, and then we can perform necessary steps to prepare the data for training and evaluation.

We will load the dataset using the code provided, which uses tensorflow\_datasets (tfds) to load both the training and test splits of the dataset. Additionally, we will retrieve the dataset's metadata (ds\_info), which provides useful information about the dataset such as the number of classes, features, and the dataset size**.**

The ds\_info object contains several important details about the dataset. Here are the key pieces of information we can extract from it:

Number of classes: The dataset has a set number of classes (in this case, different types of bean diseases).

Feature details: This tells us what the data consists of (e.g., images and labels).

Class names: The names of the classes, which represent different types of diseases or conditions in beans.

The image size is **500x500**, which is large and might not be optimal for many deep learning models that expect a specific input size (e.g., **224x224** for models like MobileNet).

The class names indicate a classification problem with **3 classes**.

The images are color images (3 channels)

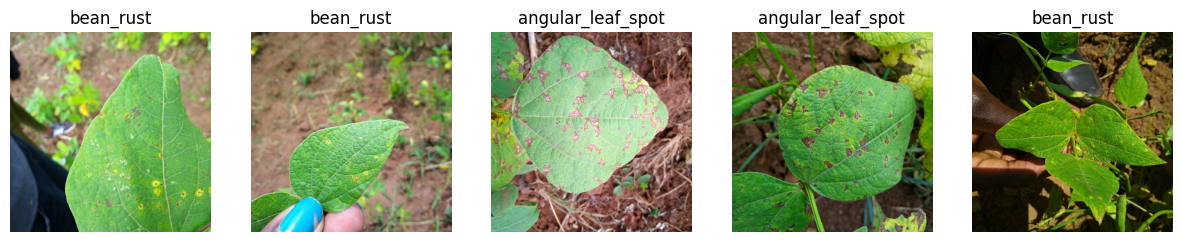
**angular\_leaf\_spot**

**bean\_rust**

**healthy**

The label value of **1** in your sample corresponds to **'bean\_rust'**.

**DATA VISUALIZATION**



Above we can see 5 images with 3 leaves with bean\_rust disease and 2 with angular leaf spot disease.

**Angular Leaf Spot**: Images show beans with irregular spots or lesions on the leaves. The affected areas appear darker or have a distinct color pattern.

**Bean Rust**: These images display orange or reddish spots on the leaves, often with a powdery appearance, which is typical of rust diseases.

**Healthy Beans**: Even though the image is not shown, these leaf Images will likely show healthy beans without significant discoloration or damage to the leaves.

We can observe differences in leaf texture, color, and size, which are crucial features for distinguishing between healthy and diseased plants.

**DATA PREPROCESSING/RESIZING**

MobileNet is designed to be a lightweight model suitable for mobile and edge devices, with the following input requirements:

* Fixed Input Size: MobileNet expects images to be resized to a fixed 224x224 resolution. This is because the architecture was originally trained with this specific input size. Resizing ensures that the input images match the model's expected shape.
* Efficiency: The model is optimized to process images of a certain size efficiently. Using a larger image size would unnecessarily increase the computational load, while smaller sizes might not contain enough information for accurate predictions.
* Transfer Learning: Since MobileNet is often used in transfer learning (i.e., using a pre-trained model on new tasks), resizing the images ensures that the features learned from large, diverse datasets (like ImageNet) are still relevant and effective for your task. The original weights are optimized for 224x224 images, and resizing the new images to this size helps the model make predictions effectively.

Resizing the images to 224x224 pixels is a crucial preprocessing step when using the MobileNet model. It ensures compatibility with the model’s architecture, maintains computational efficiency, and preserves the benefits of transfer learning by aligning the input size with the model's pre-trained weights. This step is necessary for achieving optimal performance in image classification tasks.

**MODEL SETUP FOR TRANSFER LEARNING**

To apply transfer learning with MobileNet on the Beans dataset, we'll use the pre-trained MobileNet model, which has been trained on ImageNet. The goal is to leverage the learned features (such as textures, edges, shapes, etc.) from ImageNet and adapt them to our bean diseases identification task.

**Model: "sequential"**

┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━┓

┃ **Layer (type)** ┃ **Output Shape** ┃ **Param #** ┃

┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━┩

│ mobilenetv2\_1.00\_224 (Functional) │ (None, 7, 7, 1280) │ 2,257,984 │

├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤

│ global\_average\_pooling2d │ (None, 1280) │ 0 │

│ (GlobalAveragePooling2D) │ │ │

├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤

│ dense (Dense) │ (None, 256) │ 327,936 │

├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤

│ dropout (Dropout) │ (None, 256) │ 0 │

├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤

│ dense\_1 (Dense) │ (None, 3) │ 771 │

└──────────────────────────────────────┴─────────────────────────────┴─────────────────┘

**Total params:** 2,586,691 (9.87 MB)

**Trainable params:** 328,707 (1.25 MB)

**Non-trainable params:** 2,257,984 (8.61 MB)

Base Model: MobileNetV2:

* Output Shape: The output of the MobileNetV2 base model is (None, 7, 7, 1280). This represents a feature map with a size of 7x7 and 1280 channels (feature maps). This output is generated by the MobileNetV2 layers after the convolutional operations, before the final classification layer.
* Trainable Parameters: The base model has 2,257,984 parameters (non-trainable), which are frozen, meaning they will not be updated during training.

Global Average Pooling Layer:

* After the base MobileNetV2 model, you apply GlobalAveragePooling2D, which reduces the 3D feature map (7, 7, 1280) to a 1D vector of length 1280. This step aggregates the spatial information in the feature map, making it ready for the fully connected layers.
* Output Shape: (None, 1280).

Fully Connected (Dense) Layer:

* Dense Layer with 256 units and ReLU activation adds a fully connected layer for learning high-level features.
* Output Shape: (None, 256).

Dropout Layer:

* Dropout (0.5): This layer is used to regularize the model by randomly setting 50% of the input values to zero during training. This helps prevent overfitting and encourages the model to generalize better.
* Output Shape: (None, 256).

Final Dense Layer:

* The final Dense layer with 3 units and softmax activation is used for classification into three classes: angular\_leaf\_spot, bean\_rust, and healthy. This outputs the class probabilities.

Output Shape: (None, 3)

* Total Parameters: The model has a total of 2,586,691 parameters, out of which 328,707 are trainable (these correspond to the custom layers you added: the Dense layers and Dropout).
* Non-Trainable Parameters: The MobileNetV2 base model has 2,257,984 non-trainable parameters, which are the pre-trained weights from ImageNet.
* Efficiency: The model has a relatively small number of trainable parameters compared to the base model, making it computationally efficient for training. This is because the MobileNetV2 base model's weights are frozen, and only the custom layers are being trained.

Transfer Learning: Since you're leveraging the pre-trained weights from ImageNet, the model will already have learned a lot of useful features related to visual patterns, which helps the model perform well on your task (bean disease classification) even with a smaller dataset.

Fine Tuning

After training the custom layers (i.e., the new dense layers and the output layer), we choose to fine-tune some of the deeper layers of the pre-trained MobileNetV2 model.

Why fine-tune later layers? The deeper layers in the model capture more complex and abstract features, such as specific shapes or combinations of features. These patterns may need to be adapted to the Beans dataset, which contains different kinds of images compared to ImageNet.

How to fine-tune later layers? Fine-tuning involves unfreezing some of the layers of the base model (MobileNetV2). Typically, we fine-tune only the top few layers because they are more specific to the task at hand.

**TRAINING THE MODEL**

When training a machine learning model, overfitting occurs when the model starts to memorize the training data rather than learning generalizable patterns. To prevent overfitting, we can use EarlyStopping, a regularization technique that monitors the model's performance on a validation set during training. If the model's performance on the validation set does not improve for a certain number of epochs, the training process is stopped early. This helps in preventing the model from overfitting and saves computational resources by halting the training process once the model reaches its best performance.

Here are the key parameters involved in training the model and incorporating early stopping:

1. Batch Size(32): Batch size determines the number of training samples used in one forward and backward pass. Batch sizes typically range from 16 to 128, depending on the dataset and available memory. Here we start with a batch size of 32, which strikes a balance between training time and model convergence. This means that the model will process 32 images at a time before updating its weights
2. Learning Rate(0.0001): The learning rate controls how much the model’s weights are updated in response to the estimated error during training. Learning rates are often set in the range of 0.0001 to 0.01.We are using a learning rate of 0.0001 for the Adam optimizer, which is a commonly used starting point. A smaller learning rate prevents large updates to the model’s weights, which is important when fine-tuning pre-trained models like MobileNetV2.
3. Epochs(50): An epoch is one full pass through the entire training dataset. Initially, we set epochs = 50 for the first round of training. If the model performance improves continuously, we can increase this value. But we’ll rely on early stopping to stop training early if needed.
4. Patience **(**5**)**: If the model does not improve for 3 consecutive epochs (as measured by validation loss), training will stop, thus preventing unnecessary epochs and overfitting.
5. Early Stopping Parameters: This is the number of epochs to wait for an improvement in the validation loss before stopping. A typical value is 3-5 epochs. We monitor the validation loss or validation accuracy to determine if the model’s performance has stopped improving. Restore Best Weights: We’ll set this to True so that the model reverts to the weights from the epoch that had the best performance.

Role of Early Stopping:

Early stopping serves as a regularization technique to prevent overfitting. Here's why it's crucial for your training:

* Prevents Overfitting: If the model starts to overfit the training data (i.e., the training loss continues to decrease, but the validation loss starts to increase), early stopping halts training to avoid this problem.
* Saves Resources: It saves computational resources by stopping training when no further improvement is observed, which is particularly useful when training on large datasets or complex models.
* Restores Best Weights: By restoring the best weights, the model is guaranteed to use the version of the model that performed best on the validation set, ensuring optimal performance.

**MODEL EVALUATION**

To evaluate the trained model on the test dataset and assess its performance in classifying new bean images, we will calculate the key metrics such as accuracy, precision, and recall.

Based on the evaluation of your model on the test dataset

Test Accuracy: 81.25%

* This indicates that the model was able to correctly classify 81.25% of the test images. This is a strong result, suggesting that the model performs well at classifying new unseen data.

Precision: 0.88

* Precision tells us that when the model predicts a class, it is correct 88% of the time. This is a high value, indicating that the model does a good job of minimizing false positives. It is especially important when predicting a disease, as false positives can lead to unnecessary interventions.

Recall: 0.81

* Recall represents the percentage of actual positive instances (diseased leaves) that were correctly identified by the model. A recall of 81.25% is good, but there's still some room for improvement. The model may be missing some instances of certain diseases or misclassifying healthy leaves as diseased ones.

F1-Score: 0.84

* The F1-score is the harmonic mean of precision and recall, and a value of 0.84 suggests a balanced trade-off between the two. It indicates the model is fairly strong in both minimizing false positives (precision) and false negatives (recall).

Confusion Matrix

[[37 0 6]

[ 1 25 17]

[ 0 0 42]]

Class 0 (angular\_leaf\_spot): The model performs very well with this class, achieving 37 correct predictions and only 6 misclassifications as healthy. This suggests that the model has learned to distinguish this disease from healthy leaves effectively.

Class 1 (bean\_rust): There is some difficulty with bean rust predictions. There are 17 misclassifications of this disease as healthy leaves, which indicates that the model may be confusing healthy leaves with bean rust. However, the 1 misclassification as angular leaf spot is relatively small, showing that it is reasonably good at distinguishing this class from others. The recall of 0.81 reflects this issue, as it means some instances of bean rust were missed by the model.

Class 2 (healthy): The model performed perfectly with the healthy class, with 42 correct predictions and no misclassifications. This suggests that healthy leaves are easy to distinguish from the diseased ones in the dataset.

The model performs strongly overall with 81.25% accuracy, a good precision of 0.88, and a decent recall of 0.81. The main area for improvement lies in the bean rust class, where some healthy leaves are misclassified as bean rust.

**MODEL SAVING AND REUSABILITY**

model.save('my\_bean\_disease\_classifier.keras')

This saves the model's architecture, optimizer state, and learned weights to a file named my\_bean\_disease\_classifier.keras. It’s important to note that this saved model can now be reused and loaded in the future without needing to retrain it.

**The saved .keras file** contains all the information necessary to restore the model, including the model architecture (layers and how they connect),the learned weights

,the optimizer settings (if any),the training configuration (for future fine-tuning).

Steps to Load and Reuse the Model

Once the model has been saved, it can easily be loaded and reused for prediction, evaluation, or further fine-tuning. Here’s how you can load and reuse the model:

1. Load the Model in a New Environment

You can load the saved model in a different environment or by other team members by using the tensorflow.keras.models.load\_model() function. This will restore both the architecture and the weights of the model.

2. Use the Loaded Model for Prediction or Evaluation

Once the model is loaded, you can use it for predictions or evaluate it on new data, just as you would with the original model

3. Fine-Tune the Loaded Model

If you want to continue training or fine-tuning the loaded model, you can do so by compiling the model again and continuing the training process.

Why Use the .keras Format?

* Portability: The .keras format is native to TensorFlow and can be easily shared across different environments and machines, ensuring compatibility.
* Efficiency: It saves both the model’s architecture and weights in a compact and efficient way, making it suitable for deployment and future use.
* Flexibility: The .keras format can be loaded and used directly for inference or further training.

Explanation of the Steps:

1. Saving the Model:

Why Save the Model? Saving the model after training allows you to preserve the learned weights, architecture, and configurations. This ensures that you don’t have to retrain the model from scratch each time you want to deploy it or make predictions, saving both time and resources.

The .keras format is a self-contained format that stores everything in a single file, making it ideal for sharing, storing, and deploying.

1. Loading the Model:

After saving, you can load the model in any environment with TensorFlow installed. This is helpful for transferring the model to production systems, collaborating with other team members, or performing inference on new datasets.

The load\_model() function will automatically reconstruct the model architecture and restore the learned weights, so you don't need to rebuild the model manually.

1. Use in Different Environments:

When loading the model in a different environment, it’s essential to ensure that the TensorFlow version is compatible. For example, if your team uses different versions of TensorFlow, make sure the version is consistent or the model was saved in a way that ensures backward compatibility.

1. File Management: You might want to save the model in a directory or cloud storage (like AWS S3, GCP Cloud Storage, or Google Drive) to allow access from different environments or team members.
2. Model Versioning: When deploying multiple models or versions, it's a good idea to version your saved models (e.g., model\_v1.keras, model\_v2.keras) to keep track of changes and improvements.
3. Dependencies: If you plan to load the model in a different environment, make sure that the necessary libraries (e.g., TensorFlow, Keras) and dependencies are installed

Summary

* Saving the model: Use model.save('model\_name.keras') to save the trained model in the .keras format.
* Loading the model: Use load\_model('model\_name.keras') to load the saved model into a new environment.
* Reuse: Once loaded, the model can be used for predictions, evaluation, or further fine-tuning.

This process ensures that your model can be easily transferred and reused, making it suitable for deployment in production systems or sharing with team members for collaboration.

**DEPLOYING A STREAMLIT APPLICATION**

Creating a Streamlit application that allows farmers to upload images of beans and get predictions on the presence of diseases can be done using the trained model.

We'll create a simple Streamlit application where the user can upload an image, and the model will classify it into one of the three classes: angular\_leaf\_spot, bean\_rust, or healthy.

How the Streamlit Application Works:

1. Upload Image: The user uploads an image file via the st.file\_uploader() widget. The file type is restricted to .jpg, .png, and .jpeg.
2. Preprocess the Image: The uploaded image is resized to 224x224 pixels because MobileNet requires this input size. The image is converted into a NumPy array, normalized to a range of [0, 1], and reshaped to add a batch dimension, which is required for model input.
3. Make Prediction: The pre-processed image is passed to the trained model for prediction. The model outputs a set of probabilities for each class, and we use np.argmax() to find the class with the highest probability. The predicted class and its corresponding confidence score (percentage) are displayed.
4. Display Predictions: The application shows the predicted class label and confidence score. Optionally, it also displays the probabilities for all the classes to give more detailed information about the model’s confidence.

Key Components

**file\_uploader**: Allows users to upload an image.

**Image.open**: Used to open the uploaded image file.

**predict()**: Uses the trained model to make predictions based on the uploaded image.

To Run the Application:

1. Save the above code into a Python script file, for example, bean\_disease\_classifier.py.
2. In the same directory, make sure you have the saved model file (my\_bean\_disease\_classifier.keras).
3. In the terminal, navigate to the directory where the script is saved and run:
4. bash
5. Copy code
6. streamlit run bean\_disease\_classifier.py
7. This will launch the Streamlit app in your browser, allowing you to upload an image and get predictions.

This Streamlit application makes it easy for farmers or agricultural experts to use machine learning to diagnose bean leaf diseases based on image input. Below is the streamlit interface identifying Angular leaf spot disease.

