Project Development Phase
Model Building and Training

Date	19 February 2026
Team ID	LTVIP2026TMIDS77295
Project Name	Smart Sorting: Transfer Learning for Identifying Rotten Fruits and Vegetables.

Model Building

1. Choice of Architecture

The model is built using transfer learning on top of a pre-trained MobileNetV2 convolutional neural network. MobileNetV2 is chosen because it is lightweight, optimized for image classification, and already trained on a large dataset (ImageNet), so it has learned rich low-level features like edges, textures, and shapes that transfer well to fruit images.

- The base MobileNetV2 is loaded with pre-trained weights and without its original classification head.
- The base layers are initially frozen, so their weights are not changed during the first training phase. This helps avoid destroying useful pre-learned features and speeds up convergence on a small to medium-sized dataset.

2. Custom Classification Head

A custom classification head is added on top of the MobileNetV2 base to adapt it to the binary classification problem (Fresh vs Rotten):

- Global average pooling layer to reduce the spatial feature maps into a single vector.
- One or more dense (fully connected) layers with ReLU activation to learn task-specific patterns.
- Dropout (optional) to reduce overfitting by randomly deactivating neurons during training.
- Final dense layer with 1 neuron and sigmoid activation (or 2 neurons with softmax) to output the probability of the image being fresh or rotten.

The entire model is compiled with:

- Loss function: Binary cross-entropy (for two classes).
- Optimizer: Adam optimizer with an appropriate learning rate (e.g., 1e-4).
- Metrics: Accuracy as the primary metric, with optional precision, recall, or F1 score in analysis.

3. Input Preprocessing

Before feeding images into the model, they are preprocessed to match MobileNetV2 requirements:

- Images are resized to a fixed resolution (e.g., 224×224).
- Converted to RGB and normalized (pixel values scaled to a suitable range, such as $[0, 1]$ or the specific preprocessing function provided by MobileNetV2).

- Data is loaded using generators (like ImageDataGenerator.flow_from_directory) that read images from train and test/validation directories and apply augmentation in real time.

Model Training

1. Training Configuration

The model is trained on the cleaned and split dataset:

- Training images in dataset/train/fresh and dataset/train/rotten.
- Validation or test images in dataset/test/fresh and dataset/test/rotten.
- Typical hyperparameters:
  - Batch size: e.g., 16 or 32.
  - Epochs: e.g., 10–30 for initial training, adjusted based on convergence.
  - Learning rate: small value (e.g., 1e-4) to ensure stable training.

The training loop uses:

- model.fit() with training and validation generators.
- Callbacks like ModelCheckpoint (to save best weights) and EarlyStopping (to stop when validation loss stops improving) for better generalization.

2. Data Augmentation During Training

To improve robustness and performance, data augmentation is applied on the fly to training images:

- Random rotations, horizontal flips, zoom, width/height shifts, and brightness changes.
- This artificially increases data variety and helps the model generalize to different real-world conditions (angles, lighting, background).
- Augmentation is applied only to the training generator, not to validation/test data.

3. Initial Training Phase (Frozen Base)

In the first phase:

- All layers of the MobileNetV2 base are frozen; only the newly added classification head is trainable.
- The model is trained for several epochs until training and validation accuracy stabilize.
- This phase lets the top layers learn to map MobileNetV2 features to the fresh/rotten decision without disturbing the pre-trained weights.

4. Fine-Tuning Phase (Optional, for Higher Performance)

For improved performance, a second fine-tuning phase can be carried out:

- Unfreeze the upper few layers of the MobileNetV2 base while keeping lower layers (which capture generic features) frozen.
- Re-compile the model with a lower learning rate (e.g., 1e-5) to prevent large, destabilizing weight updates.
- Train for additional epochs while monitoring validation loss and accuracy.

This fine-tuning allows the model to adapt deeper feature representations specifically to the fruit freshness detection task and can significantly boost validation accuracy.

5. Saving the Trained Model

After training:

- The best performing weights (based on validation accuracy/loss) are saved to a file such as model/healthy_vs_rotten.h5.
- This file is later loaded by the Flask backend (predict.py) for real-time predictions without retraining.

Performance Evaluation (Under Performance Testing)

Once training is complete, performance testing focuses on how well the model generalizes:

- Accuracy on the validation/test set: fraction of correctly classified images.
- Confusion matrix: counts of true positives, true negatives, false positives, and false negatives for Fresh and Rotten, revealing where the model makes mistakes.
- Classification report: precision, recall, and F1 score per class to understand performance especially on the minority class.

These metrics are used to:

- Decide whether the model is good enough for deployment.
- Analyze failure cases (e.g., rotten fruits misclassified as fresh) and guide dataset improvements or further fine-tuning.