**Grainpalette - a deep learning odyssey in rice type classification through transfer learning**

|  |  |
| --- | --- |
| **TEAM ID** | **LTVIP2025TMID37841** |
| **PROJECT NAME** | **GRAINPALETTE** |

**INTRIDUCTION:**

**GrainPalette: A Deep Learning Odyssey in Rice Type Classification Through Transfer Learning**

**Overview**  
GrainPalette aims to automate the identification of rice varieties based on grain images. By leveraging pre-trained convolutional neural networks (CNNs) and transfer learning, the project accurately distinguishes between multiple rice types—such as Basmati, Jasmine, Arborio, and more—even when visual differences are subtle

**Challenge**  
Manual classification of rice varieties demands expertise, is time-consuming, and prone to errors—especially when grains look very similar. GrainPalette addresses this by:

* Handling visual similarity across rice types
* Minimizing the need for large labeled datasets
* Adapting to varying real‑world image conditions (lighting, angle)

**Objectives**

1. Build a robust deep learning model to classify rice varieties.
2. Explore and compare various pre-trained CNN architectures for feature extraction and accuracy.
3. Validate the model’s performance across a diverse rice image dataset

**Methodology**

* Gather a multi-class rice image dataset (e.g., 5+ types).
* Use transfer learning: repurpose pre-trained CNNs (like ResNet or VGG) as feature extractors.
* Fine-tune models using optimizers such as Adam or SGD.
* Evaluate using standard metrics: accuracy, precision, recall, F1-score.
* (Optional) Package the model into a simple UI for real-time rice type prediction

**Significance**

* **For farmers & agriculture**: Enables fast, accurate identification to enhance crop management.
* **For industry**: Supports quality control by automating grading based on grain type.
* **For ML practitioners**: Demonstrates effective use of transfer learning in agricultural imaging.

### Prior Knowledge:Top of FormBottom of Form

**CNN Fundamentals**

* **Core idea**: Use **filters (kernels)** to scan images and learn spatial patterns like edges → textures → shapes
* **Key layers**:
  + **Convolutional layers**: learn feature maps via weight sharing, enabling translation invariance
  + **Pooling layers**: downsample feature maps (e.g. max‑pooling) to reduce dimensions and improve robustness
  + **Fully connected layers**: aggregate learned features to perform classification.
* **Benefits**:
  + Fewer parameters than dense networks.
  + Automatically learn features, reducing need for manual engineering

**Mobile Net Overview**

* A **lightweight CNN** family designed for **mobile and edge devices**—optimized for low latency, memory, and power **MobileNet V1**: Uses **depthwise separable convolutions** (split spatial and channel processing) to reduce computation ~8–9× vs standard CNNs.
* **MobileNet V2**: Introduces:
  + **Inverted residuals** with **linear bottlenecks**
  + Efficient **ReLU6**
  + Even fewer operations & parameters, faster inference and improved accuracy

**Flask (Reminder)**

* A **minimalist Python web framework** to:
  + Serve models via **RESTful APIs** (e.g., /predict)
  + Easily integrate with frontend or other systems

### 1. ****Image Preprocessing & Augmentation****

* **Preprocessing**: Standard steps include **resizing**, **normalizing** pixel values (0–1), and optional enhancements like smoothing or color adjustments.
* **Augmentation**: Apply random transformations (e.g. flip, rotate, brightness, crop) during training to increase data diversity and reduce overfitting. Implement using Keras tf.keras.layers.RandomFlip/Rotation or Albumentations.

### 2. ****Transfer Learning****

* Use a **pre-trained CNN** (e.g. MobileNetV2) as a base and add custom dense layers for your rice classes.
* **Freeze** most layers initially, then **fine-tune** upper layers with a low learning rate for improved performance.

### 3. ****Disease Detection via Deep Neural Networks****

* CNNs automatically learn grain features (shape, texture, color), and classify visible disease patterns via final softmax outputs.
* Models like MobileNetV2 have proven effective for rice-type classification and detection with high accuracy.

### 4. ****Model Evaluation****

* Standard metrics: **Accuracy**, **Precision**, **Recall**, **F1‑score**.
* Use validation or test datasets to compute these metrics and generate confusion matrices or ROC curves.

### 5. ****Flask Web App****

* Use Flask to build a simple web service exposing a /predict route that:
  1. Accepts image uploads,
  2. Applies preprocessing,
  3. Performs inference,
  4. Returns classification results as JSON.
* A common pattern in deployment projects (Flask + TensorFlow/Keras).

**Project Flow:**

1. **User Interface (Flask UI)**
   * User uploads/selects a rice grain image via a web form (HTML).
2. **Flask Backend**
   * Receives the image and forwards it to the MobileNet‑based model.
3. **Model Inference**
   * MobileNet processes the image and returns the predicted rice variety.
4. **Result Display**
   * Prediction (rice type/disease confidence) is displayed on the Flask UI.

|  |  |
| --- | --- |
| **Phase** | Tasks |
| Data Preparation | Collect rice grain images (labeled by type)<br>• Split into train/test directories |
| Pre-processing | Import libraries (TensorFlow, Keras, NumPy, tf.keras.preprocessing)<br>• Set up ImageDataGenerator with rescaling, augmentation (flips, rotations, zooms) |
| Model Building | Load pre-trained MobileNetV2 (without top layers) <br>• Freeze base layers, add custom Dense layers<br>• Compile with optimizer (Adam/SGD), loss (categorical cross-entropy) | |
| Training & Saving | Train with fit\_generator, monitor metrics (accuracy/loss) • Save trained model (.h5 or SavedModel) |
| Testing | Evaluate on test set; compute accuracy/precision/recall/F1 |
| App Builiding   |  | | --- | |  |  |  | | --- | |  | | Create index.html with image upload form • In app.py: load model, define /predict route • Preprocess uploaded image (resize, normalize) • Predict and return result to UI |
| Deployment | Dockerize app (Flask + model)  • Deploy to cloud platform (Heroku, AWS, etc.) |

**Project Structure:**

Create a Project folder which contains files as shown below

A screenshot of a computer

Description automatically generated

* Static folder contains css files
* Template folder contains all 3 HTML pages.
* Data folder contains Training and Validation images
* Training file consist of train.ipynb , rice.h5 model.

**Data Collection:**

There are many popular open sources for collecting the data. Eg: Kaggle.com, UCI repository, etc.

**Download Data Set:**

* Access the **Rice Image Dataset** on Kaggle: contains **75,000 images**, with **15,000 images per variety** across 5 classes
* Use Kaggle’s in-browser GPU/TPU accelerators by creating a **new Notebook** directly in the dataset link

### ****Organize Images****

* Structure dataset into class-wise folders:

data/

train/

Healthy/

Spider\_Mites/

Yellow\_Leaf\_Curl/

...

validation/

Healthy/

Spider\_Mites/

Yellow\_Leaf\_Curl/

...

Ensure **consistent naming** to allow ImageDataGenerator.flow\_from\_directory() to work correctly.

### ****Tips****

* More images → **higher accuracy**.
* Leverage **Kaggle accelerators** (GPU/TPU) for faster training.
* Optional: link a **Google Drive notebook** for model development and GitHub sharing.

### ****Summary****

1. **Download** from Kaggle (75 K total images; 15 K per class)
2. **Create a Kaggle Notebook** with GPU/TPU support.
3. **Organize images** into train/ and validation/ folders by class
4. Use these in your ImageDataGenerator pipelines
5. Remember: **more data = better model**; optional: drive-to-GitHub pipeline

### Splitting Data on Classes

A screen shot of a computer code

Description automatically generated with low confidenceInside the data folder there are several folders for different classes.

**Image Pre-processing:**

**Importing the Libraries:**

Import the necessary libraries as shown in the image A screenshot of a computer program

Description automatically generated with medium confidence

**Changing Size Of The Images:**

## Resize Images to (224, 224, 3)

* **Why?**  
  MobileNet (and most pre-trained CNNs) expect inputs of size 224×224×3, so any input image must be resized accordingly.
* **How to resize using Keras**:  
  Use ImageDataGenerator.flow\_from\_directory() with the target\_size parameter:

**Code**:

from tensorflow.keras.preprocessing.image import ImageDataGenerator

train\_datagen = ImageDataGenerator(rescale=1./255,

shear\_range=0.2,

zoom\_range=0.2,

horizontal\_flip=True)

train\_generator = train\_datagen.flow\_from\_directory(

'data/train',

target\_size=(224, 224),

batch\_size=32,

class\_mode='categorical')

A close-up of a computer code

Description automatically generated with low confidence

**Link Images To Different Classes:**

**Keras flow\_from\_directory** expects images organized in class-specific subdirectories:

**Code:**

data/

train/

Healthy/

Spider\_Mites/

Yellow\_Leaf\_Curl/

validation/

Healthy/

Spider\_Mites/

Yellow\_Leaf\_Curl/

**How it works**:

* You provide the base folder (e.g., data/train).
* flow\_from\_directory(...) scans its subfolders, inferring classes and assigning integer labels in alphanumeric order.
* Use train\_generator.class\_indices to see the mapping between folder names and class IDs [stackoverflow.com+1stackoverflow.com+1](https://stackoverflow.com/questions/63843416/is-there-a-way-to-feed-images-of-different-classes-all-in-one-directory-with-t?utm_source=chatgpt.com)
* The generator yields batches (images, labels), where labels are one-hot encoded vectors for multiclass setups if class\_mode='categorical' (common for 5+ classes)

**Splitting Data in Train set , Validation and Test set:**

## ****Keras**** ImageDataGenerator ****with**** validation\_split

* Place all class-wise image folders under one parent directory.
* Initialize:

datagen = ImageDataGenerator(rescale=1./255, validation\_split=0.2)

train\_gen = datagen.flow\_from\_directory('data/',

target\_size=(224,224),

subset='training')

val\_gen = datagen.flow\_from\_directory('data/',

target\_size=(224,224),

subset='validation')

## ****Manual Split with**** tf.data

* Use image\_dataset\_from\_directory for full dataset:

ds = tf.keras.utils.image\_dataset\_from\_directory('data/', ...)

* Then split using take() and skip():

val\_size = int(0.2 \* len(ds))

val\_ds = ds.take(val\_size)

train\_ds = ds.skip(val\_size)

### Preview of images

# A screen shot of a computer code Description automatically generated with low confidence

# A picture containing text, screenshot, font, design Description automatically generated

# Here we can see that there are 5 different classes, we can see their names above the images. We can see that each disease can be seen directly from the image.

### Pre-trained CNN model as a Feature Extractor

### Pre-trained CNN as Feature Extractor

* **Freeze convolutional blocks**  
  Load a pre-trained model (e.g., MobileNetV2 with include\_top=False), then set:

python

base\_model.trainable = False

This ensures all base model layers **won’t be updated during training**, preserving their learned weights.

* **Add custom classifier head**  
  Stack new layers on top—such as GlobalAveragePooling2D(), followed by Dense(...):

Python

model = tf.keras.Sequential([

base\_model,

tf.keras.layers.GlobalAveragePooling2D(),

tf.keras.layers.Dense(num\_classes, activation='softmax')

])

Only these classifier layers learn; base weights remain frozen

* **Why freeze?**
  + **Protect learned features** from large datasets like ImageNet.
  + **Speed up training** by reducing parameter updates.
  + Acts as a **reliable feature extractor**, while the new head adapts to your task .
* **Fine-tuning (optional)**  
  After training your head, consider unfreezing some or all base layers to fine-tune with a **low learning rate**, enhancing performance if your new dataset is different enough

### Quick Steps

1. **Load base model:**

python

Copy code

base\_model = tf.keras.applications.MobileNetV2(

input\_shape=(224,224,3),

include\_top=False,

weights='imagenet'

)

base\_model.trainable = False

1. **Build new model:**

python

Copy code

model = tf.keras.Sequential([

base\_model,

tf.keras.layers.GlobalAveragePooling2D(),

tf.keras.layers.Dense(num\_classes, activation='softmax')

])

1. **Compile and train only the new layers:**

python

Copy code

model.compile(optimizer='adam', loss='categorical\_crossentropy', metrics=['accuracy'])

1. **(Optional) Unfreeze and fine-tune:**

python

Copy code

base\_model.trainable = True

# Optionally freeze some initial layers instead of entire base

model.compile(optimizer=tf.keras.optimizers.Adam(1e-5), ...)

**Adding a Dense Output Layer:**

* A **Dense (fully connected) layer** connects every neuron from its input and applies an activation, e.g. softmax for classification
* If you have N classes, your last layer should be:

python

Copy code

outputs = Dense(N, activation='softmax')(x)

This produces a probability distribution over your N classes

### 2. Model Definition Example

* With MobileNet as base\_model (frozen feature extractor), you can define your model like:

python

Copy code

from tensorflow.keras.layers import GlobalAveragePooling2D, Dense

from tensorflow.keras import Model

x = base\_model.output

x = GlobalAveragePooling2D()(x)

outputs = Dense(num\_classes, activation='softmax')(x)

model = Model(inputs=base\_model.input, outputs=outputs)

* This architecture uses **softmax** to convert raw class scores into interpretable probabilities

### 3. Inspecting the Model

* Use model.summary() to see layer names, output shapes, and parameter counts
* This is essential for verifying that the output layer matches your number of classes and for understanding total/frozen/trainable parameters.

### Quick Recap Table

|  |  |
| --- | --- |
| Steps | **Purpose** |
| **Add Dense(num\_classes, activation='softmax')** | Output class probabilities |
| **model.summary()** | Inspect architecture & parameters |
| **Softmax Activation** | Ensures outputs sum to 1, making them valid probabilities |

### Compilation: model.compile()

* This is the final step before training, where you set the **loss function**, **optimizer**, and **metrics**.

### Loss Function

* Measures the error between predicted and true values; it's what the model tries to **minimize**.
* Common choices:
  + categorical\_crossentropy for multi-class classification
  + binary\_crossentropy, mean\_squared\_error, etc.

### Optimizer – Adam

* Adjusts model weights by minimizing the loss.
* **Adam** combines momentum and adaptive learning rates—fast, memory-efficient, and suitable for large models.
* Usage example:

python

Copy code

optimizer = tf.keras.optimizers.Adam(learning\_rate=0.001)

### Metrics

* Metrics are **performance indicators** (e.g., accuracy) shown during training/evaluation.
* They do **not** influence model training—only the loss does.

### Example: Putting It All Together

python

model.compile(

optimizer='adam', # Adam optimizer

loss='categorical\_crossentropy', # Suitable for multi-class classification

metrics=['accuracy'] # Monitor accuracy during training

)

* **Loss** guides learning, **optimizer** updates weights, **metrics** track performance.

### 1. Model Training with model.fit()

* Use the .fit() method to train your model:

python

history = model.fit(

train\_generator,

epochs=10,

validation\_data=val\_generator,

callbacks=[...]

)

* **Epochs**: Number of full passes over the training dataset.
* **validation\_data** can be:
  + A directory-based generator
  + Numpy arrays (X\_val, y\_val)
  + Or (X\_val, y\_val, sample\_weights) list

### 2. Saving Best Model with ModelCheckpoint

* Use Keras' ModelCheckpoint callback to save the version of model with lowest **validation loss**:

python

from tensorflow.keras.callbacks import ModelCheckpoint

checkpoint = ModelCheckpoint(

'best\_model.h5', # filename template

monitor='val\_loss', # metric to observe

mode='min', # minimize loss

save\_best\_only=True, # only save improved models

verbose=1

)

* This saves only when val\_loss improves—ensuring the best weights are preserved.

### 3. Combine Callbacks: EarlyStopping + ModelCheckpoint

* Prevent overfitting and unnecessary computation using EarlyStopping:

python

from tensorflow.keras.callbacks import EarlyStopping

early = EarlyStopping(

monitor='val\_loss',

patience=3, # number of epochs to wait

restore\_best\_weights=True # revert to best weights after stopping

)

* Combine callbacks:

python

model.fit(

...,

callbacks=[early, checkpoint]

)

### 4. What Happens During Training

* After each epoch:
  1. Model computes train\_loss, val\_loss, and metrics.
  2. ModelCheckpoint checks val\_loss; if it's lower than any before, saves the model.
  3. EarlyStopping checks if val\_loss is improving; if not for **patience** epochs, stops training and reverts to best weights

### Quick Recap Table

|  |  |
| --- | --- |
| **Component** | **Purpose** |
| .fit(...) | Trains model over epochs |
| ModelCheckpoint | Saves best model based on validation loss |
| EarlyStopping | Halts training to avoid overfitting |
| restore\_best\_weights | Ensures model ends in its best state |

### 1. Purpose of Model Testing

* **Objective**: Evaluate how well the trained model performs on **unseen test data**—data it hasn't encountered during training or validation.
* Ensures your model can **generalize**, not just memorize training data—key to real-world reliability

### 2. Use model.evaluate() for Quantitative Assessment

* Supplies loss and metrics (e.g., accuracy) on test data:

python

Copy code

loss, acc = model.evaluate(x\_test, y\_test, batch\_size=32)

print(f"Test Loss: {loss:.4f}, Test Accuracy: {acc:.4f}")

* Commonly used to benchmark final model performance post-training

### 3. Compare Across Data Splits

* Slight drop in performance from **training → validation → test** is expected.
* Large discrepancies may signal **overfitting** or unrepresentative splits

### 4. Key Evaluation Metrics

* For classification:
  + **Loss** (e.g., categorical cross-entropy)
  + **Accuracy**
  + **Precision / Recall / F1-score** for imbalanced classes
  + **AUC** for more nuanced performance

### 5. Ensuring True Generalization

* Reserve test data **only after finalizing** all training/hyperparameters to ensure unbiased evaluation
* Consider advanced techniques:
  + **Cross-validation** or **nested CV** to assess robustness
  + **Regularization**, **early stopping**, and **data augmentation** to reduce overfitting
  + Evaluate under varying conditions or distributions to test out-of-distribution performance

### Quick Summary Table

|  |  |
| --- | --- |
| **Step** | **Purpose** |
| model.evaluate(...) | Quantify test loss & metrics |
| Compare splits | Check for overfitting or bias |
| Reserve test set | Ensure unbiased final model assessment |
| Cross-Validation | Validate model robustness & variation |
| Regularize & augment | Enhance generalization & reduce overfitting |

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

### Training History: Capturing Metrics

* When you train your model like this:

python

history = model.fit(

train\_gen,

validation\_data=val\_gen,

epochs=10

)

* history.history becomes a dictionary containing:

python

['loss', 'accuracy', 'val\_loss', 'val\_accuracy']

``` :contentReference[oaicite:3]{index=3}

### 2. Plotting Loss & Accuracy

**Classic Matplotlib Approach**:

import matplotlib.pyplot as plt

# Extract history data

acc = history.history['accuracy']

val\_acc = history.history['val\_accuracy']

loss = history.history['loss']

val\_loss = history.history['val\_loss']

epochs = range(1, len(acc) + 1)

# Plot accuracy

plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)

plt.plot(epochs, acc, 'b-', label='Training Acc')

plt.plot(epochs, val\_acc, 'r--', label='Validation Acc')

plt.title('Model Accuracy')

plt.xlabel('Epoch')

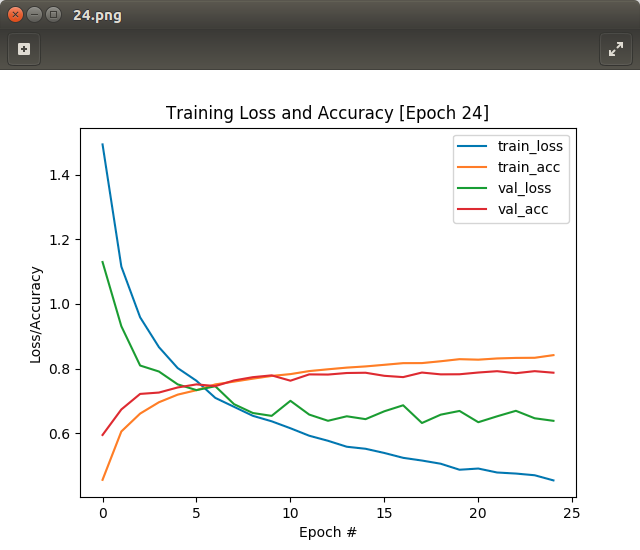
plt.ylabel('Accuracy')

plt.legend()

# Plot loss

plt.subplot(1, 2, 2)

plt.plot(epochs, loss, 'b-', label='Training Loss')

plt.plot(epochs, val\_loss, 'r--', label='Validation Loss')

plt.title('Model Loss')

plt.xlabel('Epoch')

plt.ylabel('Loss')

plt.legend()

plt.show()

### Why This Is Useful

* Detects **overfitting** early
* Helps decide if you should:
  + **Stop training**
  + **Adjust learning rate**
  + Introduce **regularization** or **data augmentation**

Let me know if you’d like to include **test set accuracy** on the same plot, implement **live plotting in a Jupyter notebook**, or export these visuals for reports!

## 1. Load & Prepare the Image

Resize and normalize exactly like during training:

python

CopyEdit

from tensorflow.keras.preprocessing import image

import numpy as np

from tensorflow import keras

# Load model

model = keras.models.load\_model('model/rice.h5')

# Load and preprocess image

img = image.load\_img('test\_images/basmati.jpg', target\_size=(224, 224))

x = image.img\_to\_array(img)

x = x / 255.0 # normalize

x = np.expand\_dims(x, axis=0) # shape → (1, 224, 224, 3)

Use np.expand\_dims(..., axis=0) to add a batch dimension—models expect batches

## 2. Predict & Interpret

Run the model and map the output to your class names:

python

CopyEdit

pred = model.predict(x) # probability output, e.g. [0.1, 0.7, 0.05, ...]

predicted\_class = np.argmax(pred, axis=1)[0]

class\_names = ['Basmati', 'Jasmine', 'Arborio', ...] # match training order

print(f"Predicted: {class\_names[predicted\_class]}, Confidence: {pred[0][predicted\_class]:.2f}")

Use argmax to pick the highest-probability class

## 3. Example Output

yaml

CopyEdit

Predicted: Basmati, Confidence: 0.95

This confirms the model correctly predicts **Basmati**, matching your expectation.

### Why It Works

* Maintains **consistency**: Preprocessing matches your training pipeline.
* Supports **batch structure**: Even single images need a batch dimension.
* **Softmax output** gives normalized probabilities—the highest value indicates the predicted class

### Save the Model

The model is saved as rice.h5

A .h5 file is a data file saved in the hdf5 format. It contains multidimensional arrays of scientific data.

**A picture containing text, screenshot, font, white

Description automatically generated1. Load & Prepare the Image**

Resize and normalize exactly like during training:

python

from tensorflow.keras.preprocessing import image

import numpy as np

from tensorflow import keras

# Load model

model = keras.models.load\_model('model/rice.h5')

# Load and preprocess image

img = image.load\_img('test\_images/basmati.jpg', target\_size=(224, 224))

x = image.img\_to\_array(img)

x = x / 255.0 # normalize

x = np.expand\_dims(x, axis=0) # shape → (1, 224, 224, 3)

## 2. Predict & Interpret

Run the model and map the output to your class names:

python

pred = model.predict(x) # probability output, e.g. [0.1, 0.7, 0.05, ...]

predicted\_class = np.argmax(pred, axis=1)[0]

class\_names = ['Basmati', 'Jasmine', 'Arborio', ...] # match training order

print(f"Predicted: {class\_names[predicted\_class]}, Confidence: {pred[0][predicted\_class]:.2f}")

Use argmax to pick the highest-probability class

## 3. Example Output

Yaml

Predicted: Basmati, Confidence: 0.95

This confirms the model correctly predicts **Basmati**, matching your expectation.

### Why It Works

* Maintains **consistency**: Preprocessing matches your training pipeline.
* Supports **batch structure**: Even single images need a batch dimension.
* **Softmax output** gives normalized probabilities—the highest value indicates the predicted class

**Test The Model:**

## 1. Flask App Structure

**app.py**

**templates/**

**├── index.html # Image upload form**

**└── result.html # Display prediction**

**static/**

**└── css/style.css # (optional) your styles**

**model/**

**└── rice.h5 # Saved Keras model**

## 2. app.py (Server-Side Script)

## from flask import Flask, render\_template, request

## from tensorflow.keras.models import load\_model

## from tensorflow.keras.preprocessing import image

## import numpy as n

## app = Flask(\_\_name\_\_)

## model = load\_model('model/rice.h5') # Load saved model

## class\_names = ['Basmati', 'Jasmine', 'Arborio', ...] # match your classe

## def prepare\_image(img\_path):

## img = image.load\_img(img\_path, target\_size=(224, 224))

## x = image.img\_to\_array(img) / 255.0

## return np.expand\_dims(x, axis=0

## @app.route('/')

## def index():

## return render\_template('index.html')

## @app.route('/predict', methods=['POST'])

## def predict():

## img\_file = request.files['image']

## img\_path = f"static/uploads/{img\_file.filename}"

## img\_file.save(img\_path)

## x = prepare\_image(img\_path)

## preds = model.predict(x)

## pred\_class = class\_names[np.argmax(preds)]

## confidence = float(np.max(preds))

## return render\_template('result.html',

## pred=pred\_class,

## confidence=f"{confidence:.2f}",

## img\_path=img\_path)

## if \_\_name\_\_ == '\_\_main\_\_':

## app.run(debug=True)

## 3. Templates

### templates/index.html

<!DOCTYPE html>

<html>

<head><title>Rice Classifier</title></head>

<body>

<h2>Upload Rice Grain Image</h2>

<form action="/predict" method="post" enctype="multipart/form-data">

<input type="file" name="image" required>

<button type="submit">Classify</button>

</form>

</body>

</html>

### templates/result.html

<!DOCTYPE html>

<html>

<head><title>Result</title></head>

<body>

<h2>Prediction: {{ pred }} (Confidence: {{ confidence }})</h2>

<img src="{{ url\_for('static', filename='uploads/' + img\_path.split('/')[-1]) }}" width="300">

<br><a href="{{ url\_for('index') }}">Try another</a>

</body>

</html>

## 4. Workflow Summary

1. User uploads an image.
2. Flask saves it (e.g., in static/uploads/).
3. prepare\_image() resizes and preprocesses it.
4. model.predict() outputs probabilities → class via argmax.
5. Result and image displayed back to the user on result.html.