Train.py

# scripts/train.py

import os

import sys

import datetime

# allow imports from project root

sys.path.append(os.path.abspath(os.path.join(os.path.dirname(\_\_file\_\_), '..')))

import numpy as np

import tensorflow as tf

from config import Config

from dataset\_loader import load\_dataset

from model\_builder import build\_fruit\_classifier

from loss\_metrics import get\_losses, get\_metrics

from tensorflow.keras.callbacks import TensorBoard, ModelCheckpoint, EarlyStopping

# -------------------------

# GPU memory growth (avoid grabbing all VRAM)

# -------------------------

gpus = tf.config.experimental.list\_physical\_devices('GPU')

if gpus:

    try:

        for gpu in gpus:

            tf.config.experimental.set\_memory\_growth(gpu, True)

        print(f"✅ GPU(s) available: {[gpu.name for gpu in gpus]}")

    except RuntimeError as e:

        print(f"❌ GPU error: {e}")

else:

    print("⚠️ No GPU found. Training on CPU.")

# -------------------------

# Overall accuracy callback

# -------------------------

class OverallAccuracyCallback(tf.keras.callbacks.Callback):

    """

    Computes an aggregated accuracy across all heads after each epoch.

    For each batch it computes (total\_correct\_labels / total\_labels)

    across all heads, and averages across batches.

    """

    def \_\_init\_\_(self, train\_data, val\_data):

        super().\_\_init\_\_()

        self.train\_data = train\_data

        self.val\_data = val\_data

    def \_preds\_to\_dict(self, preds):

        if isinstance(preds, dict):

            return preds

        if isinstance(preds, (list, tuple)):

            return dict(zip(self.model.output\_names, preds))

        return {self.model.output\_names[0]: preds}

    def \_batch\_overall\_fraction(self, batch\_x, batch\_y):

        preds = self.model.predict(batch\_x, verbose=0)

        preds = self.\_preds\_to\_dict(preds)

        correct = 0

        total = 0

        for head\_name in self.model.output\_names:

            y\_pred = preds.get(head\_name)

            y\_true = batch\_y.get(head\_name)

            if y\_pred is None or y\_true is None:

                continue

            if tf.is\_tensor(y\_true):

                y\_true = y\_true.numpy()

            if tf.is\_tensor(y\_pred):

                y\_pred = y\_pred.numpy()

            if y\_pred.ndim > 1 and y\_pred.shape[1] > 1:

                y\_pred\_classes = np.argmax(y\_pred, axis=1)

                if y\_true.ndim > 1 and y\_true.shape[1] > 1:

                    y\_true\_classes = np.argmax(y\_true, axis=1)

                else:

                    y\_true\_classes = np.ravel(y\_true).astype(int)

            else:

                y\_pred\_classes = (np.ravel(y\_pred) > 0.5).astype(int)

                y\_true\_classes = np.ravel(y\_true).astype(int)

            if len(y\_pred\_classes) != len(y\_true\_classes):

                m = min(len(y\_pred\_classes), len(y\_true\_classes))

                y\_pred\_classes = y\_pred\_classes[:m]

                y\_true\_classes = y\_true\_classes[:m]

            correct += np.sum(y\_pred\_classes == y\_true\_classes)

            total += len(y\_true\_classes)

        return correct, total

    def \_dataset\_overall\_accuracy(self, dataset):

        fractions = []

        for batch\_x, batch\_y in dataset:

            try:

                correct, total = self.\_batch\_overall\_fraction(batch\_x, batch\_y)

                if total > 0:

                    fractions.append(correct / total)

            except Exception as e:

                print(f"⚠️ Skipping batch in overall-acc calc due to error: {e}")

                continue

        return float(np.mean(fractions)) if fractions else 0.0

    def on\_epoch\_end(self, epoch, logs=None):

        overall\_train\_acc = self.\_dataset\_overall\_accuracy(self.train\_data)

        overall\_val\_acc = self.\_dataset\_overall\_accuracy(self.val\_data)

        print(f"\n📊 Epoch {epoch+1} — Overall Train Accuracy: {overall\_train\_acc:.4f} | Overall Val Accuracy: {overall\_val\_acc:.4f}")

# -------------------------

# Main

# -------------------------

def main():

    train\_ds, val\_ds, test\_ds = load\_dataset(Config)

    # Build model with frozen base

    model = build\_fruit\_classifier(Config.input\_shape, base\_trainable=False)

    model.compile(

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

        loss=get\_losses(),

        metrics=get\_metrics()

    )

    epochs = Config.epochs

    freeze\_epochs = epochs // 2

    finetune\_epochs = epochs - freeze\_epochs

    # Callbacks (reuse yours, example below)

    callbacks = [

        TensorBoard(log\_dir=os.path.join(Config.LOG\_DIR, datetime.datetime.now().strftime("%Y%m%d-%H%M%S"))),

        ModelCheckpoint(

            filepath=os.path.join(Config.MODEL\_SAVE\_PATH, 'model\_best.keras'),

            save\_best\_only=True,

            monitor='val\_loss',

            mode='min',

            verbose=1

        ),

        EarlyStopping(

            monitor='val\_loss',

            patience=Config.patience,

            restore\_best\_weights=True,

            verbose=1

        ),

        OverallAccuracyCallback(train\_ds, val\_ds),

    ]

    print(f"🚀 Training with frozen base for {freeze\_epochs} epochs...")

    model.fit(

        train\_ds,

        validation\_data=val\_ds,

        epochs=freeze\_epochs,

        callbacks=callbacks,

        verbose=1

    )

    print("🚀 Unfreezing base model for fine-tuning...")

    model.set\_base\_trainable(True)

    # Recompile with smaller LR

    model.compile(

        optimizer=tf.keras.optimizers.Adam(learning\_rate=Config.learning\_rate / 10),

        loss=get\_losses(),

        metrics=get\_metrics()

    )

    print(f"🚀 Fine-tuning for {finetune\_epochs} epochs...")

    model.fit(

        train\_ds,

        validation\_data=val\_ds,

        epochs=finetune\_epochs,

        callbacks=callbacks,

        verbose=1

    )

    # Save final model

    final\_path = os.path.join(Config.MODEL\_SAVE\_PATH, 'model\_final.keras')

    model.save(final\_path)

    print(f"✅ Model saved to {final\_path}")

if \_\_name\_\_ == "\_\_main\_\_":

    main()

model\_builder.py

import tensorflow as tf

from tensorflow.keras import layers, Input

from tensorflow.keras.applications import MobileNetV2

def build\_fruit\_classifier(input\_shape=(224,224,3), base\_trainable=False):

    inputs = Input(shape=input\_shape)

    # Pass your 'inputs' explicitly as input\_tensor to MobileNetV2

    base\_model = MobileNetV2(include\_top=False,

                             input\_tensor=inputs,

                             weights='imagenet',

                             pooling=None)  # keep pooling None to access last conv

    base\_model.trainable = base\_trainable

    # Access last conv output (connected to inputs)

    last\_conv\_output = base\_model.get\_layer('Conv\_1').output  # This is connected to inputs

    # Add global avg pooling and heads

    x = layers.GlobalAveragePooling2D(name='global\_avg\_pool')(last\_conv\_output)

    x = layers.Dropout(0.3)(x)

    out\_is\_organic = layers.Dense(1, activation='sigmoid', name='is\_organic')(x)

    out\_quality\_grade = layers.Dense(3, activation='softmax', name='quality\_grade')(x)

    out\_size = layers.Dense(3, activation='softmax', name='size')(x)

    out\_shininess = layers.Dense(1, activation='sigmoid', name='shininess')(x)

    out\_darkspots = layers.Dense(1, activation='sigmoid', name='darkspots')(x)

    out\_shape\_irregularity = layers.Dense(3, activation='softmax', name='shape\_irregularity')(x)

    # Build model with all outputs including last conv layer for Grad-CAM

    model = tf.keras.Model(

        inputs=inputs,

        outputs={

            'last\_conv\_output': last\_conv\_output,

            'is\_organic': out\_is\_organic,

            'quality\_grade': out\_quality\_grade,

            'size': out\_size,

            'shininess': out\_shininess,

            'darkspots': out\_darkspots,

            'shape\_irregularity': out\_shape\_irregularity

        }

    )

    def set\_base\_trainable(trainable):

        base\_model.trainable = trainable

        print(f"Base model trainable set to {trainable}")

    model.set\_base\_trainable = set\_base\_trainable

    return model

dataset\_loader.py

import sys

import os

import pandas as pd

import tensorflow as tf

from sklearn.model\_selection import train\_test\_split

from config import Config

# Allow importing from project root

sys.path.append(os.path.abspath(os.path.join(os.path.dirname(\_\_file\_\_), '..')))

AUTOTUNE = tf.data.AUTOTUNE

def preprocess\_image(image\_path):

    image = tf.io.read\_file(image\_path)

    image = tf.image.decode\_jpeg(image, channels=3)

    image = tf.image.resize(image, Config.IMAGE\_SIZE)

    image = image / 255.0

    return image

def augment\_image(image):

    image = tf.image.random\_flip\_left\_right(image)

    image = tf.image.random\_brightness(image, max\_delta=0.1)

    image = tf.image.random\_contrast(image, lower=0.9, upper=1.1)

    image = tf.image.random\_saturation(image, lower=0.9, upper=1.1)

    return image

def preprocess\_and\_augment(image\_path, augment=True):

    image = preprocess\_image(image\_path)

    if augment:

        image = augment\_image(image)

    return image

def make\_full\_path(path, config):

    """Safely build absolute paths from CSV values."""

    path = str(path).replace("\\", "/").strip()

    # Already absolute path → return as-is

    if os.path.isabs(path):

        return os.path.abspath(path)

    # Normalize IMAGE\_DIR

    image\_dir\_norm = config.IMAGE\_DIR.replace("\\", "/").strip()

    # Avoid double-joining if path already contains IMAGE\_DIR

    if path.startswith(image\_dir\_norm) or path.startswith(os.path.basename(image\_dir\_norm)):

        return os.path.abspath(path)

    return os.path.abspath(os.path.join(config.IMAGE\_DIR, path))

def load\_dataset(config, test\_split=0.2, val\_split=0.1, shuffle=True, seed=42):

    df = pd.read\_csv(config.CSV\_PATH)

    # Drop rows with missing required columns

    df = df.dropna(subset=config.HEADS + ['image\_path'])

    # Fix all image paths

    df["image\_path"] = df["image\_path"].apply(lambda p: make\_full\_path(p, config))

    # Show debug info

    print(f"🔍 First 5 image paths from CSV after processing:\n{df['image\_path'].head().tolist()}")

    print(f"📂 IMAGE\_DIR from config: {config.IMAGE\_DIR}")

    # Filter missing files

    missing\_files = df[~df["image\_path"].apply(os.path.exists)]

    if not missing\_files.empty:

        print(f"⚠️ Skipping {len(missing\_files)} missing images...")

        print("Example missing files:", missing\_files["image\_path"].head().tolist())

        df = df[df["image\_path"].apply(os.path.exists)]

    if df.empty:

        raise ValueError("❌ No valid images found. Check your CSV paths and IMAGE\_DIR.")

    # Split into train/val/test

    train\_val\_df, test\_df = train\_test\_split(df, test\_size=test\_split, random\_state=seed)

    train\_df, val\_df = train\_test\_split(train\_val\_df, test\_size=val\_split, random\_state=seed)

    return (

        df\_to\_dataset(train\_df, shuffle, config.BATCH\_SIZE),

        df\_to\_dataset(val\_df, shuffle=False, batch\_size=config.BATCH\_SIZE),

        df\_to\_dataset(test\_df, shuffle=False, batch\_size=config.BATCH\_SIZE)

    )

def df\_to\_dataset(df, shuffle=True, batch\_size=8):

    def encode\_label(row):

        return {

            'is\_organic': tf.cast(row[0], tf.float32),

            'quality\_grade': tf.one\_hot(tf.cast(row[1], tf.int32), 3),

            'size': tf.one\_hot(tf.cast(row[2], tf.int32), 3),

            'shininess': tf.cast(row[3], tf.float32),

            'darkspots': tf.cast(row[4], tf.float32),

            'shape\_irregularity': tf.one\_hot(tf.cast(row[5], tf.int32), 3),

        }

    image\_paths = df['image\_path'].values.astype(str)

    label\_data = df[['is\_organic', 'quality\_grade', 'size', 'shininess', 'darkspots', 'shape\_irregularity']].values

    image\_ds = tf.data.Dataset.from\_tensor\_slices(image\_paths)

    label\_ds = tf.data.Dataset.from\_tensor\_slices(label\_data)

    # Zip images and labels together

    ds = tf.data.Dataset.zip((image\_ds, label\_ds))

    # Shuffle before batching (optional)

    if shuffle:

        ds = ds.shuffle(buffer\_size=len(df))

    # Map preprocessing and encoding

    def process\_path\_and\_label(image\_path, label\_row):

        image = preprocess\_image(image\_path)

        if shuffle:  # augment only during training

            image = augment\_image(image)

        label = encode\_label(label\_row)

        return image, label

    ds = ds.map(process\_path\_and\_label, num\_parallel\_calls=AUTOTUNE)

    ds = ds.batch(batch\_size).prefetch(AUTOTUNE)

    return ds

loss\_metrics.py

# loss\_metrics.py

import tensorflow as tf

def get\_losses():

    """

    Returns a dictionary mapping each output head to its corresponding loss function.

    """

    return {

        'is\_organic': tf.keras.losses.BinaryCrossentropy(),

        'quality\_grade': tf.keras.losses.CategoricalCrossentropy(),

        'size': tf.keras.losses.CategoricalCrossentropy(),

        'shininess': tf.keras.losses.BinaryCrossentropy(),

        'darkspots': tf.keras.losses.BinaryCrossentropy(),

        'shape\_irregularity': tf.keras.losses.CategoricalCrossentropy()

    }

def get\_metrics():

    """

    Returns a dictionary mapping each output head to a list of metrics.

    Includes accuracy and AUC for each head.

    """

    return {

        'is\_organic': [

            tf.keras.metrics.BinaryAccuracy(name='accuracy'),

            tf.keras.metrics.AUC(name='auc')

        ],

        'quality\_grade': [

            tf.keras.metrics.CategoricalAccuracy(name='accuracy'),

            tf.keras.metrics.AUC(name='auc', multi\_label=True)

        ],

        'size': [

            tf.keras.metrics.CategoricalAccuracy(name='accuracy'),

            tf.keras.metrics.AUC(name='auc', multi\_label=True)

        ],

        'shininess': [

            tf.keras.metrics.BinaryAccuracy(name='accuracy'),

            tf.keras.metrics.AUC(name='auc')

        ],

        'darkspots': [

            tf.keras.metrics.BinaryAccuracy(name='accuracy'),

            tf.keras.metrics.AUC(name='auc')

        ],

        'shape\_irregularity': [

            tf.keras.metrics.CategoricalAccuracy(name='accuracy'),

            tf.keras.metrics.AUC(name='auc', multi\_label=True)

        ]

    }

Gradcam\_multi.py

import tensorflow as tf

import numpy as np

import cv2

def compute\_gradcam(model, processed\_tensor, head\_name):

    model\_input = model.input

    last\_conv\_layer = model.get\_layer('Conv\_1')

    last\_conv\_output = last\_conv\_layer.output

    head\_output = model.get\_layer(head\_name).output

    grad\_model = tf.keras.Model(inputs=model\_input, outputs=[last\_conv\_output, head\_output])

    with tf.GradientTape() as tape:

        conv\_outputs, predictions = grad\_model(processed\_tensor)

        tape.watch(conv\_outputs)

        if len(predictions.shape) == 2:

            class\_idx = tf.argmax(predictions[0])

            target = predictions[:, class\_idx]

        else:

            target = predictions[:, 0]

    grads = tape.gradient(target, conv\_outputs)

    pooled\_grads = tf.reduce\_mean(grads, axis=(0, 1, 2))

    conv\_outputs = conv\_outputs[0]

    heatmap = tf.reduce\_sum(tf.multiply(pooled\_grads, conv\_outputs), axis=-1)

    heatmap = tf.maximum(heatmap, 0) / (tf.reduce\_max(heatmap) + 1e-10)

    return heatmap.numpy()

def overlay\_heatmap(heatmap, image, alpha=0.4, colormap=cv2.COLORMAP\_JET):

    heatmap = cv2.resize(heatmap, (image.shape[1], image.shape[0]))

    heatmap = np.uint8(255 \* heatmap)

    heatmap\_color = cv2.applyColorMap(heatmap, colormap)

    overlayed\_img = heatmap\_color \* alpha + image

    overlayed\_img = np.clip(overlayed\_img, 0, 255).astype(np.uint8)

    return overlayed\_img

def generate\_gradcam\_explanations(model, processed\_tensor, class\_head, quality\_head):

    class\_heatmap = compute\_gradcam(model, processed\_tensor, class\_head)

    quality\_heatmap = compute\_gradcam(model, processed\_tensor, quality\_head)

    # Convert processed tensor (normalized) back to uint8 image for overlay

    img\_for\_overlay = (processed\_tensor[0].numpy() \* 255).astype(np.uint8)

    class\_overlay = overlay\_heatmap(class\_heatmap, img\_for\_overlay)

    quality\_overlay = overlay\_heatmap(quality\_heatmap, img\_for\_overlay)

    classification\_text = f"Grad-CAM for head: {class\_head}"

    quality\_text = f"Grad-CAM for head: {quality\_head}"

    return class\_overlay, quality\_overlay, classification\_text, quality\_text

predict.py

import os

import numpy as np

import tensorflow as tf

import matplotlib.pyplot as plt

from gradcam\_multi import generate\_gradcam\_explanations

from config import Config

def load\_and\_preprocess(image\_path):

    original\_img = tf.keras.utils.load\_img(image\_path, target\_size=(224, 224))

    img\_array = tf.keras.utils.img\_to\_array(original\_img)

    img\_array = np.expand\_dims(img\_array, axis=0) / 255.0

    processed\_tensor = tf.convert\_to\_tensor(img\_array, dtype=tf.float32)

    return original\_img, processed\_tensor

def interpret\_predictions(preds):

    # preds keys: 'is\_organic', 'quality\_grade', 'size', 'shininess', 'darkspots', 'shape\_irregularity'

    # Each is a numpy array of shape (1, classes) or (1, 1)

    # 1) Organic / Inorganic with confidence

    is\_organic\_prob = preds['is\_organic'][0][0]

    label = "Organic" if is\_organic\_prob > 0.5 else "Inorganic"

    confidence = is\_organic\_prob if is\_organic\_prob > 0.5 else 1 - is\_organic\_prob

    confidence\_pct = confidence \* 100

    # 2) Quality grade (3 classes)

    qg\_idx = np.argmax(preds['quality\_grade'][0])

    quality\_map = {0: "Bad", 1: "Medium", 2: "Good"}

    quality\_label = quality\_map.get(qg\_idx, "Unknown")

    # 3) Size (3 classes)

    size\_idx = np.argmax(preds['size'][0])

    size\_map = {0: "Small", 1: "Medium", 2: "Big"}

    size\_label = size\_map.get(size\_idx, "Unknown")

    # 4) Shininess (binary)

    shiny\_prob = preds['shininess'][0][0]

    shiny\_label = "Shiny" if shiny\_prob > 0.5 else "Dull"

    # 5) Darkspots (binary)

    darkspot\_prob = preds['darkspots'][0][0]

    darkspot\_label = "Yes" if darkspot\_prob > 0.5 else "No"

    # 6) Shape irregularity (3 classes)

    shape\_idx = np.argmax(preds['shape\_irregularity'][0])

    shape\_map = {0: "Normal", 1: "Some irregularity", 2: "Lots of irregularity"}

    shape\_label = shape\_map.get(shape\_idx, "Unknown")

    # Compose text for prediction summary

    pred\_text = (

        f"PREDICTION\n"

        f"- Label: {label}\n"

        f"- Confidence: {confidence\_pct:.1f}%\n"

        f"- Quality: {quality\_label}\n"

        f"- Size: {size\_label}\n"

        f"- Shine: {shiny\_label}\n"

        f"- Dark Spots: {darkspot\_label}\n"

        f"- Shape: {shape\_label}"

    )

    # You can add extra detail or rules for classification and quality notes

    classification\_notes = []

    if shape\_idx == 2:

        classification\_notes.append("Shape asymmetry detected")

    if shiny\_label == "Dull":

        classification\_notes.append("Dull shine")

    if darkspot\_label == "Yes":

        classification\_notes.append("Dark spots detected")

    if len(classification\_notes) == 0:

        classification\_notes.append("No obvious defects")

    quality\_notes = []

    if quality\_label == "Good":

        quality\_notes.append("Smooth, glossy surface")

        quality\_notes.append("No dark spots detected")

    elif quality\_label == "Medium":

        quality\_notes.append("Some wrinkling visible")

    else:

        quality\_notes.append("Poor surface quality")

    class\_text = "CLASSIFICATION\n" + "\n".join(f"- {line}" for line in classification\_notes)

    quality\_text = "QUALITY\n" + "\n".join(f"- {line}" for line in quality\_notes)

    return pred\_text, class\_text, quality\_text

def plot\_results(original\_img, class\_overlay, quality\_overlay, pred\_text, class\_text, quality\_text):

    fig, axs = plt.subplots(3, 3, figsize=(15, 10))

    # Show images in the top row:

    axs[0, 0].imshow(original\_img)

    axs[0, 0].axis('off')

    axs[0, 0].set\_title("Original Image")

    axs[0, 1].imshow(class\_overlay)

    axs[0, 1].axis('off')

    axs[0, 1].set\_title("Classification Grad-CAM")

    axs[0, 2].imshow(quality\_overlay)

    axs[0, 2].axis('off')

    axs[0, 2].set\_title("Quality Grad-CAM")

    # Text below images in the middle row, aligned left and wrapped

    axs[1, 0].text(0.3, 0.8, pred\_text, fontsize=11, va='center', ha='left', wrap=True)

    axs[1, 0].axis('off')

    axs[1, 1].text(0.3, 0.8, class\_text, fontsize=11, va='center', ha='left', wrap=True)

    axs[1, 1].axis('off')

    axs[1, 2].text(0.3, 0.8, quality\_text, fontsize=11, va='center', ha='left', wrap=True)

    axs[1, 2].axis('off')

    # Hide the bottom row of subplots completely (optional)

    for ax in axs[2, :]:

        ax.axis('off')

    plt.tight\_layout()

    plt.show()

def predict\_and\_explain(image\_path, model):

    original\_img, processed\_tensor = load\_and\_preprocess(image\_path)

    preds = model(processed\_tensor)

    preds = {k: v.numpy() for k, v in preds.items()}

    # Select heads for Grad-CAM visualization

    class\_head = 'is\_organic'

    quality\_head = 'quality\_grade'

    class\_overlay, quality\_overlay, classification\_text, quality\_text = generate\_gradcam\_explanations(

    model, processed\_tensor, class\_head, quality\_head

    )

    pred\_text, class\_text, quality\_text = interpret\_predictions(preds)

    return original\_img, class\_overlay, quality\_overlay, pred\_text, class\_text, quality\_text

if \_\_name\_\_ == "\_\_main\_\_":

    test\_image\_path = os.path.join(Config.TEST\_IMAGES\_DIR, "test2.jpg")

    model = tf.keras.models.load\_model(os.path.join(Config.MODEL\_SAVE\_PATH, "model\_final.keras"))

    print("Model loaded.")

    print(f"Processing: {test\_image\_path}")

    results = predict\_and\_explain(test\_image\_path, model)

    plot\_results(\*results)

config.py

import os

class Config:

    # Root directory of your project

    ROOT\_DIR = "D:/plants-app/fruit-classify-quality-detector"

    # Paths

    IMAGE\_DIR = os.path.join(ROOT\_DIR, "data", "images")  # Where images are stored

    CSV\_PATH = os.path.join(ROOT\_DIR, "data", "labels.csv")  # Your label file

    MODEL\_SAVE\_PATH = os.path.join(ROOT\_DIR, "outputs", "model\_weights")

    GRADCAM\_SAVE\_PATH = os.path.join(ROOT\_DIR, "outputs", "predictions", "gradcam")

    TEST\_IMAGES\_DIR = os.path.join(ROOT\_DIR, "data", "test\_images")  # Directory for test images

    LOG\_DIR = os.path.join(ROOT\_DIR, "outputs", "logs")

    # Input image size for the model

    IMAGE\_SIZE = (224, 224)

    BATCH\_SIZE = 8

    input\_shape = IMAGE\_SIZE + (3,)  # Input shape for the model

    learning\_rate = 1e-3

    epochs = 20

    patience = 5

    # Fruit classes (optional if you want to classify fruit type too)

    FRUIT\_CLASSES = [

        "mango", "banana", "apple", "orange", "guava",

        "pomegranate", "grapes", "watermelon", "papaya", "strawberries"

    ]

    # Label Mappings

    IS\_ORGANIC = {0: "Inorganic", 1: "Organic"}

    QUALITY\_GRADE = {0: "Bad", 1: "Mid", 2: "Good"}

    SIZE = {0: "Small", 1: "Mid", 2: "Large"}

    SHININESS = {0: "Dull", 1: "Shiny"}

    DARKSPOTS = {0: "None", 1: "Yes"}

    SHAPE\_IRREGULARITY = {0: "None", 1: "Some", 2: "Lots"}

    NOTES = ["batch\_single", "batch\_double"]

    # Multi-output heads

    HEADS = [

        "is\_organic",

        "quality\_grade",

        "size",

        "shininess",

        "darkspots",

        "shape\_irregularity"

    ]

    # Mapping of head to number of classes

    HEAD\_OUTPUT\_DIM = {

        "is\_organic": 2,

        "quality\_grade": 3,

        "size": 3,

        "shininess": 2,

        "darkspots": 2,

        "shape\_irregularity": 3

    }

    # For reverse mapping predictions → readable labels

    LABEL\_MAPS = {

        "is\_organic": IS\_ORGANIC,

        "quality\_grade": QUALITY\_GRADE,

        "size": SIZE,

        "shininess": SHININESS,

        "darkspots": DARKSPOTS,

        "shape\_irregularity": SHAPE\_IRREGULARITY

    }