**Fruit Classification & Quality Grading with YOLO + Grad‑CAM**

**Author:** Chaudhry Ahmed Ali Khan  
**License:** Apache License 2.0  
**Repo:** <https://github.com/Ahmed-ali0005/fruit-classify-quality-detector.git>

**Table of Contents**

1. Purpose & High‑Level Overview
2. System & Environment Requirements
3. Project Structure (Authoritative)
4. Data & Annotations
5. End‑to‑End Workflow
6. File‑by‑File Deep Dive
   * config.py
   * dataset\_loader.py
   * model\_builder.py
   * loss\_metrics.py
   * train.py
   * gradcam\_multi.py
   * predict.py (prediction + explanations)
   * yolo\_train.py
   * test\_yolo.py (detection + per‑box classification stats)
7. YOLO Dataset Config (data.yaml)
8. Training: How To
9. Inference: How To
10. Troubleshooting & Common Pitfalls
11. Accuracy & Robustness Improvement Playbook
12. Deployment Notes
13. Licensing & Attribution
14. Future Work

**Purpose & High‑Level Overview**

This repository provides a full pipeline to:

* **Detect** fruits in images using **YOLOv8** (Ultralytics).
* **Classify & grade** each detected fruit with a **multi‑head CNN** (MobileNetV2 backbone in TensorFlow/Keras) that predicts:
  + is\_organic (Organic vs Inorganic)
  + quality\_grade (Bad, Mid, Good)
  + size (Small, Mid, Large)
  + shininess (Dull, Shiny)
  + darkspots (None, Yes)
  + shape\_irregularity (None, Some, Lots)
* **Explain** predictions with **Grad‑CAM** overlays per head.

The architecture supports **one‑image → many detections → per‑crop classification → aggregated stats + explanations**.

**System & Environment Requirements**

**OS**: Windows 10/11 (primary), Linux and macOS should work with path adjustments.  
**CPU**: Intel Core i7 (minimum).  
**RAM**: 16 GB (minimum).  
**Disk**: 8 GB free (minimum; dataset size may require more).  
**GPU**: NVIDIA GPU with CUDA (recommended for both YOLO and TensorFlow training/inference).

**Python**: 3.10.x recommended.  
**Key Python libs**: TensorFlow 2.10.x (as per your environment), Ultralytics YOLO, OpenCV, NumPy, Pandas, scikit‑learn, Matplotlib, Pillow.

⚠️ **Windows note:** You already enabled GPU memory growth in train.py. Keep NVIDIA drivers + CUDA/cuDNN aligned with your TensorFlow build.

**Project Structure (Authoritative)**

fruit-classify-quality-detector/

├── LICENSE

├── README.md

├── requirements.txt

├── scripts/

│ ├── train.py # Train the multi-head classifier

│ ├── model\_builder.py # Build MobileNetV2-based multi-output model

│ ├── loss\_metrics.py # Losses & metrics dicts per head

│ ├── dataset\_loader.py # TF data pipeline + CSV parsing

│ ├── gradcam\_multi.py # Grad-CAM utilities

│ ├── predict.py # Predict + Grad-CAM visualization end-to-end

│ ├── config.py # (Recommended single source of truth for Config)

│ └── (optional) \_\_init\_\_.py

├── yolo\_train.py # YOLOv8 training (outside scripts)

├── test\_yolo.py # Detect with YOLO + crop classify stats

├── datasetYolo/ # YOLO dataset (Ultralytics format)

│ ├── data.yaml

│ ├── train/

│ │ ├── images/\*.jpg|png

│ │ └── labels/\*.txt

│ └── valid/

│ ├── images/\*.jpg|png

│ └── labels/\*.txt

├── data/

│ ├── images/ # Classifier images (if not using YOLO crops)

│ ├── test\_images/ # for quick testing

│ └── labels.csv # Classifier labels (see schema below)

└── outputs/

├── logs/

├── model\_weights/

│ ├── model\_best.keras

│ └── model\_final.keras

└── predictions/

└── gradcam/

**Import consistency:** In the current code, some files import Config as from config import Config and others as from scripts.config import Config. The **canonical** location should be **repo root** (config.py) and all imports should be from config import Config. Update imports in test\_yolo.py and anywhere else to avoid confusion.

**Data & Annotations**

**1) Classifier CSV (data/labels.csv)**

**Required columns:**

* image\_path: path to the image file (absolute or relative to data/images/).
* is\_organic: 0 or 1.
* 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).

**2) YOLO dataset (datasetYolo/)**

* Standard Ultralytics structure with images/ and labels/ under train/ and valid/.
* Label files are in YOLO TXT format (one line per object: cls x\_center y\_center width height normalized to [0,1]).
* Created using RoboFlow annotations on each individual image.

**End‑to‑End Workflow**

1. **Train YOLO (yolo\_train.py)** → produces weights (e.g., runs/detect/train\*/weights/best.pt).
2. **Train classifier (scripts/train.py)** on labeled single‑fruit images (or on YOLO crops if you generate them).
3. **Run detection + classification**:
   * Use test\_yolo.py to detect, crop, classify each box and aggregate per‑image stats (as a test to check if the yolo model is performing correctly).
   * Use predict.py to generate **Grad‑CAM explanations** and visual reports.

**File‑by‑File Deep Dive**

Below is a detailed walkthrough of each file. Explanations are grouped by logical blocks and, where useful, line‑by‑line commentary.

**1) config.py**

**Purpose:** Centralize paths, constants, label maps, and hyperparameters.

Key contents:

* **Root & paths**: ROOT\_DIR, IMAGE\_DIR, CSV\_PATH, MODEL\_SAVE\_PATH, GRADCAM\_SAVE\_PATH, TEST\_IMAGES\_DIR, LOG\_DIR, YOLO\_MODEL\_PATH.
* **Model input**: IMAGE\_SIZE = (224, 224) and input\_shape = (224, 224, 3).
* **Training**: BATCH\_SIZE, learning\_rate, epochs, patience.
* **Heads and label maps**: HEADS, HEAD\_OUTPUT\_DIM, and readable LABEL\_MAPS.

**Why it matters:** Keeping this as the **single source of truth** reduces errors and makes experiments reproducible.

**Tip:** If you change image size or add a head (e.g., color\_damage), update HEADS, HEAD\_OUTPUT\_DIM, CSV schema, and training code.

**2) dataset\_loader.py**

**Purpose:** Build efficient TensorFlow Datasets from the CSV.

Important functions:

* preprocess\_image(image\_path)
  + Reads bytes → decodes JPEG → resizes to Config.IMAGE\_SIZE → normalizes to [0,1].
* augment\_image(image)
  + Random left/right flip, brightness/contrast/saturation jitter (mild, to avoid label drift).
* preprocess\_and\_augment(image\_path, augment=True)
  + Convenience wrapper to combine the above.
* make\_full\_path(path, config)
  + Normalizes relative/absolute paths; guards against double‑joining IMAGE\_DIR.
* load\_dataset(config, test\_split=0.2, val\_split=0.1, ...)
  + Loads CSV with pandas, cleans rows (drops NAs), repairs paths, filters missing files.
  + Splits into train/val/test via sklearn.model\_selection.train\_test\_split.
  + Returns three Datasets: train, val, test (batched, prefetched).
* df\_to\_dataset(df, shuffle=True, batch\_size=8)
  + Builds paired datasets (image\_path, labels).
  + encode\_label(row) emits a **dict** per head:
    - Binary heads: tf.float32 scalars (0/1).
    - Multi‑class heads: one‑hot vectors of sizes 3.
  + Uses map to turn (image\_path, label\_row) → (image\_tensor, dict\_of\_labels).

**Notable details:**

* Uses AUTOTUNE prefetch for throughput.
* Applies augmentation only when shuffle=True (i.e., for training).
* Emits labels as a **dict** with keys exactly matching model output names.

**Pitfalls:** Ensure your CSV columns exactly match the keys in encode\_label and the heads in model\_builder.py.

**3) model\_builder.py**

**Purpose:** Define the **multi‑output** classifier on top of MobileNetV2.

Key steps:

1. inputs = Input(shape=input\_shape) — explicit input tensor.
2. base\_model = MobileNetV2(include\_top=False, input\_tensor=inputs, weights='imagenet', pooling=None)
   * include\_top=False to get feature maps.
   * input\_tensor=inputs ensures the graph is connected to inputs.
   * pooling=None so we can access the last conv feature map (Conv\_1) for Grad‑CAM.
3. base\_model.trainable = base\_trainable — start frozen for transfer learning.
4. last\_conv\_output = base\_model.get\_layer('Conv\_1').output — this is the feature map used by Grad‑CAM.
5. x = GlobalAveragePooling2D()(last\_conv\_output) → Dropout(0.3)
6. **Heads** (names must match the CSV/loader):
   * is\_organic: Dense(1, activation='sigmoid')
   * quality\_grade: Dense(3, activation='softmax')
   * size: Dense(3, activation='softmax')
   * shininess: Dense(1, activation='sigmoid')
   * darkspots: Dense(1, activation='sigmoid')
   * shape\_irregularity: Dense(3, activation='softmax')
7. The model outputs a **dict** including 'last\_conv\_output' for Grad‑CAM convenience.
8. Adds helper method set\_base\_trainable(True|False) to unfreeze for fine‑tuning.

**Why dict outputs?**

* Training uses only the heads; 'last\_conv\_output' is extra (ignored if not in loss dict). This is handy for Grad‑CAM without rebuilding models.

**Caveat:** When compiling, provide losses/metrics **only** for the heads (not for last\_conv\_output). Keras will ignore outputs not in the loss dict by default.

**4) loss\_metrics.py**

**Purpose:** Central registry of losses & metrics per head.

* **Losses:**
  + Binary heads → BinaryCrossentropy()
  + Multi‑class heads → CategoricalCrossentropy()
* **Metrics:**
  + Binary heads → BinaryAccuracy, AUC
  + Multi‑class heads → CategoricalAccuracy, AUC(multi\_label=True)

**Why AUC on multi‑class?** It provides threshold‑independent view; you may also add TopKCategoricalAccuracy(k=1) if you prefer top‑1 accuracy.

**Tuning ideas:** Use **class weights** if classes are imbalanced; or **Focal Loss** for hard negatives.

**5) train.py**

**Purpose:** Orchestrate classifier training with freeze → fine‑tune schedule.

**Top section:**

* Adds repo root to sys.path so from config import Config works.
* GPU setup: lists GPUs and enables set\_memory\_growth (prevents TensorFlow from pre‑allocating all VRAM). Prints available GPUs.

**Callback OverallAccuracyCallback:**

* Computes an **aggregate accuracy** across all heads at epoch end.
* Internally calls model.predict on batches, converts predictions to class labels (sigmoid > 0.5 or argmax), compares to ground truth, and averages across batches.

**Main flow:**

1. train\_ds, val\_ds, test\_ds = load\_dataset(Config)
2. model = build\_fruit\_classifier(Config.input\_shape, base\_trainable=False)
3. compile(...) with Adam(lr), get\_losses(), get\_metrics()
4. Determine freeze\_epochs vs finetune\_epochs (here: half & half).
5. **Callbacks**:
   * TensorBoard (logs per timestamp)
   * ModelCheckpoint(model\_best.keras) (monitors val\_loss)
   * EarlyStopping with patience=Config.patience, restores best weights
   * OverallAccuracyCallback
6. **Phase 1 (frozen)**: model.fit(..., epochs=freeze\_epochs)
7. Unfreeze base: model.set\_base\_trainable(True)
8. Recompile with **lower LR**: learning\_rate / 10
9. **Phase 2 (fine‑tune)**: model.fit(..., epochs=finetune\_epochs)
10. Save model\_final.keras to outputs/model\_weights/.

**Why two‑phase training?** Transfer learning works best when you stabilize the new heads first, then gently fine‑tune the backbone with a smaller LR.

**6) gradcam\_multi.py**

**Purpose:** Compute Grad‑CAM heatmaps per head and overlay on images.

Core functions:

* compute\_gradcam(model, processed\_tensor, head\_name)
  + Builds a sub‑model that outputs (last\_conv\_output, head\_output).
  + With tf.GradientTape, computes gradients of the **target** score w.r.t. conv features.
    - For multi‑class heads: takes argmax logit.
    - For binary heads: uses the single logit (index 0).
  + Averages gradients over H×W to get channel weights (α\_k).
  + Weighted sum of channels → ReLU → normalize to [0,1] → heatmap.
* overlay\_heatmap(heatmap, image, alpha=0.4, colormap=cv2.COLORMAP\_JET)
  + Resizes heatmap to image, colorizes, blends with alpha.
* generate\_gradcam\_explanations(model, processed\_tensor, class\_head, quality\_head)
  + Computes two heatmaps (e.g., is\_organic and quality\_grade).
  + Returns overlay images + text labels for the heads.

**Important assumptions:**

* The last conv layer is named 'Conv\_1' (true for MobileNetV2). If you change the backbone, update this layer name.

**7) predict.py (prediction + explanations)**

**Purpose:** One‑shot explainable prediction on a single image (or callable from a loop).

Highlights:

* load\_and\_preprocess(image\_path)
  + Loads PIL image → resizes to 224×224 → converts to float tensor batch [1,H,W,3] in [0,1].
* interpret\_predictions(preds)
  + Converts raw model outputs to human‑readable strings and confidences.
  + Maps argmax indices to labels for multi‑class heads; thresholds binary heads at 0.5.
* predict\_and\_explain(image\_path, classifier\_model, yolo\_model)
  + Runs YOLO on the **original image**; draws boxes; for each box:
    - Crops the region, resizes to 224×224, feeds classifier.
    - Aggregates per‑box stats (shiny, darkspots, size breakdown, irregular shapes).
  + Also runs classifier on the **full image** to compute **Grad‑CAM** overlays for class\_head and quality\_head.
  + Returns: annotated image, class overlay, quality overlay, text panels.
* plot\_results(...)
  + Nicely displays Original + Class Grad‑CAM + Quality Grad‑CAM with text panels.

**Notes:**

* For per‑box classification you are using **crops** from YOLO detections — this is the right integration pattern.
* For Grad‑CAM you currently compute on the full image; you can optionally compute per‑box Grad‑CAMs on the crop images for tighter explanations.

**8) yolo\_train.py**

**Purpose:** Train YOLOv8 on your datasetYolo/ with stable hyperparameters for modest GPUs.

Key points:

* Loads yolov8n.pt (nano) as the base model.
* Trains with:
  + epochs=100, imgsz=640, batch=8, optimizer='AdamW', lr0=1e-4
  + amp=False (mixed precision off on Windows to avoid NaNs)
  + workers=0 (Windows dataloader stability)
  + augment=True, close\_mosaic=0
* Saves final model to outputs/model\_weights/yolov8\_fruit\_final.pt (Ultralytics also saves best.pt under runs/detect/.../weights/).

**Adjustments to consider:**

* Use model = YOLO('yolov8s.pt') for higher accuracy if VRAM allows.
* Increase imgsz to 768 or 896 if small fruits are missed.

**9) test\_yolo.py (detection + per‑box classification stats)**

**Purpose:**

* Load YOLO detector and Keras classifier.
* For each image in Config.TEST\_IMAGES\_DIR:
  + Detect boxes → crop → classify → aggregate counts (shine, darkspots, irregular size/shape).
  + Draw boxes and print per‑image summary to console; show window with cv2.imshow.

Key functions:

* load\_classifier() → loads model\_final.keras from MODEL\_SAVE\_PATH.
* preprocess\_crop(crop) → resize to 224×224 and normalize.
* interpret\_predictions(preds) → same mapping logic as elsewhere.
* Main loop: iterates test images, runs YOLO, iterates results, crops, classifies, draws, and aggregates.

**Windows UI note:** cv2.imshow requires a display; press **ESC** to quit.

**YOLO Dataset Config (data.yaml)**

Place under datasetYolo/data.yaml. Example template:

path: ../datasetYolo # or absolute path

train: train/images

val: valid/images

defaults:

imgsz: 640

batch: 8

epochs: 100

workers: 0

names:

0: mango

1: banana

2: apple

3: orange

4: guava

5: pomegranate

6: grapes

7: watermelon

8: papaya

9: strawberries

Make sure class order matches your label TXT files.

**Training: How To**

**1) Install dependencies**

pip install -r requirements\_full.txt

**2) Train YOLO**

python yolo\_train.py

# After training, note: runs/detect/train\*/weights/best.pt

**3) Train classifier (two‑phase)**

python scripts/train.py

# Logs -> outputs/logs/

# Weights -> outputs/model\_weights/model\_best.keras and model\_final.keras

**Inference: How To**

**1) Detection + per‑box classification roll‑up (to test detection model integrated with classification)**

python test\_yolo.py

* Expects YOLO weights path inside the script (update to your runs/detect/.../weights/best.pt).
* Reads images from Config.TEST\_IMAGES\_DIR.

**2) Single image with Grad‑CAM visual report**

python predict.py

* Update test\_image\_path to a file under data/test\_images/.
* Outputs three panels: Original, Class Grad‑CAM, Quality Grad‑CAM with text overlays.

**Troubleshooting & Common Pitfalls**

* **Import path mixups (config vs scripts.config)**: Standardize to from config import Config with config.py at repo root.
* **CSV paths invalid**: dataset\_loader.py prints the first few repaired paths and warns about missing files. Fix image\_path values or IMAGE\_DIR.
* **TensorFlow GPU memory errors**: You already set set\_memory\_growth; reduce batch size if needed.
* **AUC on multi‑class**: If you see shape errors, consider using AUC(multi\_label=True, num\_labels=3) or remove AUC for simplicity.
* **Ultralytics workers on Windows**: keep workers=0 for stability.
* **OpenCV windows don’t show**: Avoid running in headless terminals; or save images instead of imshow.

**Accuracy & Robustness Improvement Playbook**

**Data**

* **Balance labels** per head; oversample minority classes or use **class weights**.
* **Hard‑example mining**: curate images with glare, occlusion, small fruits.
* **Resolution**: Increase base resolution to 256–288 for the classifier if VRAM allows.

**Augmentation**

* Classifier: add **RandomRotation(±10°)**, **RandomZoom**, **Gaussian noise**, **Cutout** (TensorFlow Addons) while avoiding label leakage.
* YOLO: increase hsv\_\* augmentation, mosaic/mixup (Ultralytics handles).

**Architecture**

* Switch backbone to **EfficientNetB0/B1** for better accuracy at similar cost.
* Add a small **SE (Squeeze‑Excitation)** block before heads.
* Use **Dropout** tuning (0.2–0.5) and **L2 weight decay** on Dense layers.

**Losses & Optimization**

* Use **Focal Loss** for binary heads (shiny/darkspots) prone to imbalance.
* Apply **label smoothing** (e.g., 0.05) on multi‑class heads.
* Tune LR schedule: **Cosine decay** or **OneCycle**.
* Use **early stopping** per‑head metric if certain heads overfit.

**Training Strategy**

* **Longer warmup** with frozen base (e.g., 5–10 epochs), then fine‑tune more layers gradually (unfreeze last N blocks first).
* **Mixed precision** on capable GPUs (TensorFlow: tf.keras.mixed\_precision.set\_global\_policy('mixed\_float16')), monitor stability.

**Post‑processing**

* For per‑image aggregation, consider **majority vote** or **probability averaging** across multiple crops/augmentations (test‑time augmentation).

**Evaluation**

* Track **per‑head confusion matrices**, ROC/PR curves.
* Use a **validation set per fruit type** to detect class‑conditional failures.

**Deployment Notes**

* Save models in **SavedModel** or .keras; for mobile/edge, convert to **TF‑Lite** and quantize (dynamic range or int8 with a small calibration set).
* For a REST API, wrap YOLO + classifier in a server (FastAPI/Flask). Batch requests for throughput.
* Consider **Dockerizing** with CUDA base images for reproducibility.

**Licensing & Attribution**

* Project is licensed under **Apache 2.0**. Keep the LICENSE file at repo root and include copyright notices in source files (already present).
* If you include third‑party assets (datasets, pretrained weights), include their licenses in a NOTICE file where required.

**Future Work**

* **Semi‑supervised learning** to leverage unlabeled data.
* **Active learning** loop: suggest uncertain samples for labeling to improve weak heads (e.g., darkspots).
* **Per‑box Grad‑CAM report**: generate one Grad‑CAM image per detected fruit crop.

**Appendix A — CLI Snippets (Ready to Copy)**

**Train classifier:**

python scripts/train.py \

--epochs 20 \

--batch-size 8

*(You can wire argparse if you want flags; currently constants live in config.py.)*

**Train YOLO:**

python yolo\_train.py

**Run YOLO + per‑box classifier stats:**

python test\_yolo.py

**Run single‑image prediction with Grad‑CAM:**

python predict.py

**Appendix B — Notes on Consistency & Paths**

* Prefer from config import Config everywhere.
* Keep Config.IMAGE\_DIR, Config.CSV\_PATH, and labels.csv:image\_path consistent.
* Keep YOLO data.yaml class names aligned with Config.FRUIT\_CLASSES (order matters for display).