ChatGPT

Crop and Weed Detection Using YOLOv8

Abstract: This project uses the Ultralytics YOLOv8 framework to build an object detector that distinguishes crops from weeds. We collected a custom dataset of agricultural images, annotated each image in the YOLO format, and trained a YOLOv8n (nano) model on this data using Python. We then implemented inference scripts for images, videos, and live webcam input, as well as a simple Tkinter GUI to run detection interactively. The trained model outputs detected bounding boxes labeled "crop" or "weed," and we save the predictions (including box coordinates and confidences) to CSV files for further analysis. This documentation details the objectives, tools, dataset structure, training process, and inference code, with clear explanations and examples.

Project Objectives

- **Detect and classify** objects in agricultural images as *crop* or *weed* using a YOLOv8 detector.
- **Build and train** a custom object detection model with two classes (crop, weed) in the YOLOv8 framework.
- **Prepare and structure** a dataset in the required YOLO format, including creating a YAML configuration file.
- **Implement inference scripts** to run detection on images, video files, and live webcam streams.
- **Create a user-friendly GUI** with Tkinter for running inference without command-line operations.
- **Save detection results** (bounding boxes, class labels, confidence scores) to CSV for record-keeping and analysis.
- **Document challenges** encountered (e.g. file path issues, API changes) and their solutions.
- **Present final results** and evaluation outputs in a clear way.

Tools and Technologies Used

- **Ultralytics YOLOv8:** A state-of-the-art object detection framework built on PyTorch. We use the YOLOv8 Python API to load models, train on custom data, and run inference [1] [2].
- **Python:** The implementation language. Key libraries include:
- **PyTorch/Torch:** Underlies YOLOv8 for model training and inference. By default, YOLOv8 uses GPU if available, otherwise CPU [3].
- **OpenCV (cv2):** Used for handling video input/output and image preprocessing in some scripts.
- **PIL/Pillow:** For image loading and manipulation if needed.
- **Pandas/CSV:** To store inference results (e.g. detected box coordinates and labels) in tabular CSV files.
- **Tkinter:** A standard GUI toolkit in Python, used to build a simple interface (`gui_inference.py`) for loading images and running detection with buttons and display elements.
- **Ultralytics Data Utilities:** Built-in dataset converters and format checkers (e.g. converting COCO to YOLO format, verifying annotation directories) [1].
- **YOLOv8 Pretrained Weights:** We start from the `yolov8n.pt` pretrained weights (a lightweight model) and fine-tune it on our dataset.

Dataset Description

Our dataset consists of color images of agricultural fields, annotated with two classes: **crop** and **weed**. Each image has a corresponding label file in YOLO format. The YOLO annotation format requires one `.txt` label file per image, where each line is:

```
<object_class> <x_center> <y_center> <width> <height>
```

with all coordinates normalized (0–1) by image width/height [4]. The `<object_class>` is an integer starting at 0. For example, if class 0 is "crop" and class 1 is "weed", a line like `1 0.45 0.6 0.1 0.2` would label a weed at the given position. A classes file (or YAML) defines this mapping from index to name [5] [4].

The dataset is split into *train* and *val* sets. A typical folder structure is:

```
agri_data/
 ├── images/
 │    ├── train/
 │    └── val/
 ├── labels/
 │    ├── train/
 │    └── val/
 └── data.yaml
```

Here, `images/train/` contains all training images, and `labels/train/` contains the corresponding `.txt` annotation files. (Similarly for `val/`.) The `data.yaml` file configures the dataset for YOLOv8. An example `data.yaml` looks like:

```
train: agri_data/images/train
val: agri_data/images/val
nc: 2
names: [ "crop", "weed" ]
```

where `nc` is the number of classes and `names` lists their names. (Ultralytics requires such a YAML config to point to the train/val image directories and class info [2].) The `classes.txt` file (optional) can also list class names line by line, but YOLOv8 uses the names in the YAML.

To illustrate, below is the project directory structure (highlighting dataset folders):

Figure: Dataset folder structure. Images and labels are organized under `agri_data/` *in YOLO format, with a* `data.yaml` *config file listing paths and classes.*

The YAML file `data.yaml` ties this structure together. It tells YOLOv8 where to find the train/val images and how many classes are in the data [2]. For example:

```
path: agri_data
train: images/train
val: images/val
nc: 2
names: [ "crop", "weed" ]
```

This tells YOLOv8 to look under `agri_data/images/train` for training images, `agri_data/images/val` for validation, with 2 classes named “crop” and “weed”. (Any additional keys like `test:` or full file paths can also be included if needed.)

In summary, **two classes** are defined (“crop” and “weed”), with indices 0 and 1. Each image has a matching `.txt` annotation (same base filename) in the `labels/` directories. This follows the standard YOLO format [4]. The complete structure and YAML config ensure the training script can load the data correctly.

Training Process

The training is handled by a Python script (`train.py`) using the Ultralytics YOLO API. We start from the pretrained YOLOv8n weights (`yolov8n.pt`) and fine-tune on our dataset. Below is an example training snippet:

```
from ultralytics import YOLO

# Load the pretrained YOLOv8n model
model = YOLO("yolov8n.pt")

# Train the model on our custom dataset
results = model.train(
    data="agri_data/data.yaml",  # path to dataset config YAML
    epochs=50,                   # number of training epochs
    imgsz=640,                   # image size (pixels)
    batch=16,                    # batch size (if CPU, set small)
    device="cpu",                # force training on CPU (Ultralytics auto-
chooses GPU/CPU)
    save=True                    # save weights and training logs
)
```

- `model = YOLO("yolov8n.pt")`: Loads the YOLOv8n model (pretrained on COCO). This initializes the network architecture and loads weights. (Loading a `.pt` checkpoint is the recommended way to start training [1].)
- `data="agri_data/data.yaml"`: Points to the YAML dataset configuration. This file provides training/validation image paths and class info [2].
- `epochs=50`: Specifies 50 passes over the training set. Adjusting this changes how long the model trains.

- `imgsz=640`: All images are resized to 640×640 for training. Higher values can improve accuracy at cost of speed.
- `batch=16`: Number of images per batch. (On CPU, a small batch like 1 or 2 might be needed; on GPU, larger batches are possible.)
- `device="cpu"`: Forces training on CPU. By default, YOLOv8 uses GPU 0 if available, else CPU [3]. (If training on a GPU machine, omit this or set `device=0`.)
- `save=True`: Ensures that training checkpoints and final weights are saved. After training, you'll find the best and last model weights under `runs/detect/train/weights/`.

During training, YOLOv8 prints metrics (loss, precision, recall, mAP) each epoch. It saves the best model (highest validation mAP) to `runs/detect/train/weights/best.pt` and also a final epoch model to `last.pt`. These output folders (`runs/detect/train/`) follow Ultralytics' default structure.

All training hyperparameters have sensible defaults (see Ultralytics docs for details on learning rate, optimizer, etc. [2]). In practice, the critical parts are the dataset path (`data`) and number of epochs.

Inference Scripts

After training, we perform inference using the trained model. We wrote several scripts for different sources:

1. Image Inference (`inference.py`)

To run detection on a single image or a folder of images, we use the YOLO predict mode. Example usage:

```
from ultralytics import YOLO

# Load the trained model (best.pt from training)
model = YOLO("runs/detect/train/weights/best.pt")

# Run inference on an image file
results = model.predict(source="test_images/sample1.jpg",  # input image
                        conf=0.25,                         # confidence threshold
                        save=True,                         # save output image
                        save_txt=False)                    # also save labels
to .txt if True
```

- `source="test_images/sample1.jpg"`: path to the input image.
- `conf=0.25`: only detections with confidence $\geq$ 0.25 are kept.
- `save=True`: writes an annotated image (with boxes and labels) to disk.
- The annotated image is saved under a `runs/detect/predict/exp` directory by default (e.g. `runs/detect/predict/exp/sample1.jpg`), created by Ultralytics.
- The `results` object returned is a list (or generator) of `Results` objects. Each `result` contains `.boxes` (bounding boxes), `.masks`, etc.

YOLOv8's `predict` automatically handles opening the image, running the model, and plotting boxes. For example, calling `model.predict(..., save=True)` will overlay detection boxes on the image and save it [6]. The results can be accessed programmatically; for instance:

```
result = results[0]          # since we ran on one image
df = result.boxes.data.cpu().numpy()  # get raw box array if needed
result.show()                # display in a window
result.save(filename="output.jpg")  # explicitly save to filename [7].
```

2. Video File Inference (`inferencevideo.py`)

For video files (e.g. `input.mp4`), we similarly call predict:

```
results = model.predict(source="videos/input.mp4",  # input video path
                        conf=0.25,
                        save=True)
```

This processes the video frame by frame and writes an annotated video file (e.g. `runs/detect/predict/exp/input.mp4`). By default, YOLOv8 saves videos in MP4 or AVI format at the `runs/detect/predict/exp` path. (You can control the save format via the `yaml` or predictor arguments if needed.) The inference is done with streaming under the hood, so it doesn't load the whole video into memory.

3. Webcam Inference (`inferencewebcam.py`)

To run live detection from a webcam, we use source `0` (the default camera) with `stream=True`. Example:

```
results = model.predict(source=0,           # use first webcam
                        conf=0.25,
                        show=True,           # show live video window
                        stream=True)
# stream=True yields a generator of results

# Process each frame in a loop
for result in results:
    # result is for one frame; show it or save it
    result.show()  # displays the frame with boxes
    # Optionally, break after some frames or on keypress
```

Setting `stream=True` makes YOLOv8 return a frame-by-frame generator, which is more memory-efficient for live video [8]. The loop continuously updates the display window. (Without `stream`, the entire result sequence would be returned at once, which is not ideal for live streams.)

4. Tkinter GUI (`gui_inference.py`)

We built a simple Tkinter interface to allow users to run image detection without the command line. The GUI includes: - A button to **browse for an image** (opens a file dialog). - A button to **run detection** on the selected image. - A `Label` or `Canvas` to display the chosen image and the result. - Code behind the scenes calls the same `model.predict(...)` as above, then updates the UI with the annotated image.

Example (simplified) code snippet for the GUI button:

```
from tkinter import filedialog, Label, Button, Tk
from ultralytics import YOLO

model = YOLO("runs/detect/train/weights/best.pt")

def run_detection():
    img_path = filedialog.askopenfilename()  # let user pick an image file
    if img_path:
        results = model.predict(source=img_path, save=True)
        # Display the annotated image in the GUI
        result = results[0]
        annotated = result.orig_img  # NumPy array of image with boxes
        photo = ImageTk.PhotoImage(Image.fromarray(annotated))
        image_label.config(image=photo)
        image_label.image = photo

root = Tk()
detect_button = Button(root, text="Open Image and Detect",
command=run_detection)
detect_button.pack()
image_label = Label(root)
image_label.pack()
root.mainloop()
```

This GUI code uses `filedialog` to select an image, then runs the same YOLOv8 inference and shows the output. (Error handling and layout code omitted for brevity.) No external citations are needed here.

5. Saving Results to CSV

After inference, we often want a CSV record of all detections (bounding box coordinates, class labels, confidences). YOLOv8 makes this easy via the `Results` object. For example:

```
results = model.predict(source="test_images/sample1.jpg", save=True)
result = results[0]
df = result.boxes.data.cpu().numpy()  # raw array [x1, y1, x2, y2, conf, cls]
# Alternatively, convert to Pandas
```

```
result_df = result.boxes.xyxyn  # normalized xyxy in tensor form
# Or use built-in conversion:
res_csv = results.to_csv("detection_output.csv")  # convert all results to CSV
```
[9]

The `results.to_csv()` method (or `result.to_df()`) will produce a CSV text with all detections. As noted in the Ultralytics reference, the results object can be exported to a pandas DataFrame or CSV format [9]. We ensure we write these outputs after every inference. For example, we might append each frame's results to a CSV when processing video or webcam streams.

Code Snippets Explained

Below we summarize key code segments:

- **Training snippet (`train.py`):**

```
from ultralytics import YOLO
model = YOLO("yolov8n.pt")                              # load YOLOv8n model
results = model.train(data="agri_data/data.yaml",       # dataset config
                      epochs=50,
                      imgsz=640,
                      device="cpu")                     # use CPU (default
is GPU if available)
```

This builds and trains the model on our two-class dataset [1]. The `data` parameter points to the YAML config, which contains paths and class names [2].

- **Inference on an image (`inference.py`):**

```
from ultralytics import YOLO
model = YOLO("runs/detect/train/weights/best.pt")    # load the trained
weights
results = model.predict(source="path/to/image.jpg",  # input image
                        conf=0.3,
                        save=True)                  # save annotated image
```

This runs detection and saves `image.jpg` with bounding boxes. The results can be viewed or programmatically parsed [6].

- **Inference on a video (`inferencevideo.py`):**

```
results = model.predict(source="videos/input.mp4", save=True, conf=0.3)
```

After running, check `runs/detect/predict/exp/input.mp4` for the output video with detections overlaid.

- **Webcam inference (`inferencewebcam.py`):**

```
results = model.predict(source=0, stream=True, show=True, conf=0.3)
for res in results:
    res.show()  # display each frame
```

This opens the webcam and shows detections live (pressing a key or closing the window ends the loop).

- **GUI interaction (`gui_inference.py`):** code is as shown above. The core idea is that clicking the button triggers `model.predict` on a chosen image and updates a Tkinter image widget with the result.

- **Dataset YAML snippet:**
The `data.yaml` file for two classes might look like:

```
path: agri_data
train: images/train
val: images/val
nc: 2
names:
  0: crop
  1: weed
```

This follows the Ultralytics format, where `train:` and `val:` give image directories relative to `path:`, and `names` maps each class index to a label [2].

File and Folder Structure

Expected project layout (relative paths):

```
project_root/
 ├─ agri_data/
 │    ├─ images/
 │    │    ├─ train/
 │    │    └─ val/
 │    ├─ labels/
 │    │    ├─ train/
 │    │    └─ val/
 │    └─ data.yaml
```

```
├─ classes.txt              # (optional) "crop\nweed"
├─ train.py
├─ inference.py
├─ inferencevideo.py
├─ inferencewebcam.py
├─ gui_inference.py
└─ yolov8n.pt               # pretrained model weights
```

- **Training outputs:** After training, Ultralytics saves logs and weights under `runs/detect/train/`. For example, the best model is `runs/detect/train/weights/best.pt`, and a final weights file is `last.pt`.
- **Inference outputs:** By default, predictions are saved under `runs/detect/predict/`. Each run gets a new `exp` folder. For example, running `model.predict(..., save=True)` might produce `runs/detect/predict/exp/image1.jpg` or `runs/detect/predict/exp/input.mp4`.

Challenges Faced

During development, we encountered and addressed several issues:

- **File path errors:** It's crucial that the YAML file paths match the actual folders. A common error was mis-pointing `train:` or `val:` paths. We fixed this by using absolute paths or ensuring the YAML `path:` is correct (YOLOv8 reads paths relative to `path:`) [2].
- **Inference not saving:** Originally, some outputs weren't saved because we missed `save=True`. Remember to use `save=True` (and appropriate video writer settings) to write files. Also ensure write permissions on the save directory.
- `save_path` **parameter:** The Ultralytics API has a `save_path` argument in predictor utilities. If specifying it manually, ensure it's a `Path` object or valid string, and that directories exist.
- **Changing API (to_pandas vs to_df):** In earlier YOLO versions one used `result.pandas().xyxy` or `result.to_pandas()`. In YOLOv8, the recommended way is `results.to_df()` or `results.to_csv()` [9], so we switched to `to_csv()` to export. (This resolved errors like `'Results' object has no attribute to_pandas`.)
- **Performance on CPU:** Training on CPU is very slow. We reduced batch sizes (e.g. `batch=1`) and image size (320 or 640) to avoid crashes due to memory.
- **Model path errors:** When loading the model for inference, the exact path to the trained weights mattered. We had to double-check that we loaded the correct file (`best.pt` vs `last.pt` vs the original `yolov8n.pt`).

By referring to the Ultralytics documentation, we resolved these issues. For example, the docs clarify how to set `device="cpu"` if no GPU is available [3], and how to export results with `to_csv()` [9].

Final Results

After training for 50 epochs, the YOLOv8 model learned to distinguish crops and weeds. The training log shows convergence of loss and gradually improving mAP on the validation set. We qualitatively verify the model on test images: for instance, when running inference on a field image, the model correctly draws boxes around weeds labeled as "weed" and around crop stalks labeled as "crop".

Example outputs are saved under the `runs/detect/predict/` folder. For instance, after detecting on `sample1.jpg`, one sees `runs/detect/predict/exp/sample1.jpg` with bounding boxes and labels. All detected boxes (coordinates and labels) are recorded in our CSV files. These results indicate that the model successfully identifies weeds (often smaller plants in the background) and crops (larger foreground plants) in the images.

Example Result Path

- Annotated image: `runs/detect/predict/exp/sample1.jpg`
- Inference CSV: `detection_output.csv`

(Here "exp" is the timestamped experiment folder created by YOLOv8.)

Conclusion

In this project, we built a complete pipeline for detecting crops vs weeds using the Ultralytics YOLOv8 framework. We collected and annotated a custom dataset, trained a lightweight YOLOv8n model on CPU, and implemented multiple inference modes. The system uses standard YOLO data formats (with a `data.yaml` configuration) and integrates with Python tools like Tkinter and OpenCV for a user-friendly interface. Results (bounding boxes and CSV logs) are automatically saved, making it easy to review detections. Overall, YOLOv8 proved effective at this binary classification task. With further tuning (e.g. more data or longer training), the accuracy can be improved, but the current model already provides a solid proof-of-concept for automated crop/weed detection.

Sources: Project implementation was guided by the Ultralytics YOLOv8 documentation on custom training and inference [1] [10], which explains dataset structure and training commands. The YOLOv8 API reference was used for result handling (e.g., saving outputs) [7] [9]. The YOLO label format is standard (one label file per image in `class x_center y_center width height` format) [4]. These sources ensure correct usage of the tools and clarifications of parameters.

---

[1] [4] [10] Object Detection Datasets Overview - Ultralytics YOLO Docs
https://docs.ultralytics.com/datasets/detect/

[2] Configuration - Ultralytics YOLO Docs
https://docs.ultralytics.com/usage/cfg/

[3] Model Training with Ultralytics YOLO - Ultralytics YOLO Docs
https://docs.ultralytics.com/modes/train/

[5] How to Create a Dataset for Object Detection using the YOLO Labeling Format | Cogniflow Documentation
https://docs.cogniflow.ai/en/article/how-to-create-a-dataset-for-object-detection-using-the-yolo-labeling-format-1tahk19/

[6] [8] Python Usage - Ultralytics YOLO Docs
https://docs.ultralytics.com/usage/python/

[7] Model Prediction with Ultralytics YOLO - Ultralytics YOLO Docs
https://docs.ultralytics.com/modes/predict/

9 Reference for ultralytics/engine/results.py - Ultralytics YOLO Docs
https://docs.ultralytics.com/reference/engine/results/