PokeWar Hackathon CV Report

September 23, 2025

Abstract

We present a reproducible pipeline that converts COCO-format annotations into YOLO format, trains a YOLO object detector on a large Pokémon dataset (15.9k images), and runs inference to produce bounding boxes and center coordinates for downstream targeting. This report explains our design choices, key implementation details, robustness fixes, and the rationale for selecting YOLO and the `yolo12x` model.

1. **Load COCO JSON:** COCO stores separate lists of images, annotations, and categories. Grouping annotations by image makes it straightforward to write per-image label files (YOLO format).

2. **Class ID mapping:** COCO category IDs commonly start at 1. YOLO labels require zero-based class IDs (0..nc-1), so we mapped:
$$\text{yolo\_cls} = \text{coco\_category\_id} - 1.$$

3. **Train/validation split:** We used an 80/20 split with a fixed random seed for reproducibility and to monitor validation loss, mAP, etc.

4. **COCO → YOLO conversion:** COCO bboxes are $[x_{min}, y_{min}, w, h]$ in pixel units. YOLO input expects normalized center format $[x_c^{norm}, y_c^{norm}, w^{norm}, h^{norm}]$. The conversion formulas we used:
$$x_c^{norm} = \frac{x_{min} + w/2}{W}, \qquad y_c^{norm} = \frac{y_{min} + h/2}{H}$$
$$w^{norm} = \frac{w}{W}, \qquad h^{norm} = \frac{h}{H}$$
where W, H are the image width and height respectively.

5. **Training:** Initialized from a general pretrained checkpoint (ImageNet/COCO-style), then trained with Ultralytics' `model.train(...)` interface. We used validation metrics to select the best checkpoint.

6. **Inference & CSV export:** We loaded the best checkpoint, run `model.predict(...)` per test image, read `res.boxes` (class and `xywh`) and wrote rows to a CSV.

1 Why `yolo12x` over `yolo11` (model selection)

- **Model capacity:** `yolo12x` has higher capacity (deeper/wider) which helps distinguish visually-similar sprites and learn features from a large dataset (15k images).
- **Accuracy vs Speed:** `yolo12x` typically yields higher mAP at the cost of slower training and inference and higher GPU memory. `yolo11` (or `yolo11n/s`) is better for rapid iteration or constrained deployment.
- **Data scale justification:** Given a sufficiently large training corpus (we have 15k images, many augmented), higher-capacity models can be trained effectively without severe overfitting.

2 Appendix: Useful Kaggle-ready commands & snippets

2.1 Conversion helper (COCO $\rightarrow$ YOLO label)

```
# python snippet (concept)
def coco_to_yolo(bbox, W, H):
    x_min, y_min, w, h = bbox
    x_c = (x_min + w/2) / W
    y_c = (y_min + h/2) / H
    return x_c, y_c, w/W, h/H
```

2.2 data.yaml example

```
train: /kaggle/working/yolo_pokemon/images/train
val: /kaggle/working/yolo_pokemon/images/val
nc: 4
names: ['Pikachu','Charizard','Bulbasaur','Mewtwo']
```

2.3 Training (Ultralytics)

```
from ultralytics import YOLO
model = YOLO("yolo12x.pt")
model.train(data="/kaggle/working/yolo_pokemon/data.yaml",
           epochs=50, imgsz=640, batch=8)
```

2.4 Inference & CSV export

```
results = model.predict(img_path, conf=0.5, save=False)
res = results[0]
for box in res.boxes:
    cls = int(box.cls[0].item())
    x_c, y_c, w, h = box.xywh[0].tolist()
    # write row to CSV
```