Training Deploying and Optimizing AI models with Ultralytics YOLO and TensorRT

By Andrew Jimenez, Bodi Samdan & Elvis Los

Introduction

Under the direction of Mr. Jack Tsao, our team was tasked with developing a comprehensive Standard Operating Procedure (SOP) for the full lifecycle management of AI models, including training, deployment, and optimization. This initiative builds upon foundational work by previous interns, with Andrew Jimenez spearheading the initial research phase. Andrew conducted an extensive evaluation of existing workflows and modernized the pipeline by integrating industry-standard tools such as Ultralytics' YOLOv11 framework.

This document represents a collaborative effort, combining preliminary research with systematic refinements to create a unified protocol for AI/ML operations. The SOP emphasizes reproducibility, performance optimization, and seamless integration with NVIDIA's edge computing ecosystem.

YOLO, A Brief History

YOLO (You Only Look Once), a popular object detection and image segmentation model, was developed by Joseph Redmon and Ali Farhadi at the University of Washington. Launched in 2015, YOLO quickly gained popularity for its high speed and accuracy.

The model we're using in this SOP:

YOLO11: Ultralytics' latest YOLO models delivering state-of-the-art (SOTA) performance across multiple tasks, including object detection, segmentation, pose estimation, tracking, and classification, leverage capabilities across diverse AI applications and domains.

Documentation & Technical Specifications

The following resources and tools were used in the development of this SOP:

Hardware Environment

- NVIDIA Jetson AGX Orin Developer Kit
- JetPack SDK 6.0

Software Stack

- Operating System: Ubuntu 22.04 LTS (NVIDIA-optimized distribution)
- Python: 3.10.12
- CUDA: 12.2.140
- cuDNN: 8.9.4.25
- TensorRT: 8.6.2.3

Core Frameworks

- Ultralytics YOLOv11 (Object Detection Framework)
- NVIDIA DeepStream SDK (Streaming Analytics Toolkit)

Key Documentation

- Ultralytics Documentation
    - YOLOv11 implementation guides
    - Model export/optimization protocols
- NVIDIA DeepStream Dev Guide
    - Edge deployment configurations
    - Hardware acceleration workflows
- NVIDIA DeepStream GitHub

Training on labeled Data with Ultralytics YOLO

NOTE: If you're working with a pre-labeled dataset, follow the steps below. However, if you're using a dataset without labels (such as one you created yourself), please refer to our alternative guide here.

Step 1. Download and Organize Your Dataset

Download a pre-labeled dataset:
Download a free public dataset (like a mask-detection dataset). After downloading (often as a ZIP file), create a dedicated project directory and unzip the dataset .
https://public.roboflow.com/classification/rock-paper-scissors

```
# Make a directory for the project
mkdir -p ~/[ProjectName]
# Go into the project
cd ~/[ProjectName]
# Unzip files of the dataset in your current folder with `.`
unzip ~/Downloads/YourDataset.zip .
```

Example:

```
mkdir -p ~/mask_detection_project
cd ~/mask_detection_project
unzip ~/Downloads/YourDataset.zip .
```

Dataset Folder Structure:

Ensure your dataset folder follows a structure similar to this (YOLO format):

```
[ProjectName]/
├── data.yaml
├── train/
│   ├── images/
│   └── labels/
├── valid/
│   ├── images/
│   └── labels/
```

```
└── test/ (optional)
    ├── images/
    └── labels/
```

Train vs. Validation Split:

Split your images and corresponding label files into training and validation sets. A common split is 80% training and 20% validation.

1. *Note:* The `data.yaml` file is provided by Roboflow or created manually to define the dataset paths and classes.

Step 2. Edit the data.yaml file

Open the `data.yaml` file in a text editor and update the paths to point to the correct directories. Using absolute paths is recommended to avoid confusion with default directories. For example, if your dataset is in `~/mask_detection_project`, edit **data.yaml** as follows:

```
train: /PathToYourProject/ProjectName/train/images
val: /PathToYourProject/ProjectName/valid/images
test: /PathToYourProject/ProjectName/test/images  # optional

nc: 2 # amount of classes in the dataset
names: ['yourFirstClass', 'yourSecondClass'] # the class names in the
dataset
```

Example:

```
train: /home/ubuntu/mask_detection_project/train/images
val: /home/ubuntu/mask_detection_project/valid/images
test: /home/ubuntu/mask_detection_project/test/images   # optional

nc: 2
names: ['mask', 'no-mask']
```

Make sure:

- The paths exactly match your directory structure.
- The number of classes (`nc`) and class names (`names`) match your dataset.

Step 3. Train Your Model

We can use the CLI or a Python file to fine-tune a model. We will be using the pre-trained yolov11 model.

```
cd ~/mask_detection_project
yolo detect train data=data.yaml model=yolo11n.pt epochs=100 imgsz=416
```

Explanation:

- **data=data.yaml:** Uses your dataset configuration.
- **model=yolov11n.pt:** Uses the pre-trained YOLOv11 nano model (you can also use a different model if desired).
- **epochs=100:** Sets the number of training epochs.
- **imgsz=416:** Resizes images to 416×416 during training (this should match your dataset's preprocessing).

During training, checkpoints (including the best model) will be saved (typically in a folder like `runs/detect/trainX/`).

Tip: When working with NVIDIA-powered machines (Jetson, …) running YOLO, you can achieve better performance by following these steps in the documentation:

https://docs.ultralytics.com/guides/nvidia-jetson/#best-practices-when-using-nvidia-jetson

Step 4. Validate Your Model

After training, you need to validate your model to get performance metrics. You must validate the best model in your last train folder. (You can find this model by searching for the `best.pt` file.

This file is stored in the `/ProjectName/runs/detect/trainX/weights` folder.

```
yolo detect val model=runs/detect/trainX/weights/best.pt data=data.yaml
imgsz=416
```

Example:

```
yolo detect val model=runs/detect/train2/weights/best.pt data=data.yaml
imgsz=416
```

Step 5. Testing on a Live Camera Feed

Note: Make sure your camera works well with AI vision models (it should have a high enough FPS rate + performance should be high). If you encounter low frames during testing on a live camera feed click here.

Step 5a. When Running on NVIDIA Devices

Because we're using an NVIDIA device, we **shouldn't** try to run your model with the Yolo detect predict command. We should run your models with **DeepStream**:

NVIDIA DeepStream ***is a powerful SDK made for NVIDIA devices that lets you use GPU-accelerated technology to develop end-to-end streaming vision AI pipelines.***
Cloning the Github Repository makes this possible

1. Go into your project root folder.
2. Clone GitHub repository that's built on top of an NVIDIA-provided plugin that makes sure that you can use YOLO11 models with DeepStream: https://github.com/sh-aidev/deepstream-yolo11
3. Follow the steps in the quickstart/readme with this guide and an explanation of what all the steps do.
1. Clone the repository in your project folder.

```
cd ~/[PathToYourProject]/
git clone https://github.com/sh-aidev/deepstream-yolo11.git

cd deepstream-yolov11
cd src

# Download the model weights (skip this part)
```

2. **Copy** your `best.pt` model into the `deepstream-yolov11/src/models` folder.

3. Convert/export your `best.pt` model to an onnx model (3x faster) with the script.

```
# Convert the model weights to onnx
python3 scripts/export_yolo11.py -w models/[ModelName.pt]
```

4. Change the path in the config file so it uses this model.

```
# Change the model path in the config_infer_primary_yoloV11.txt file to the
onnx file generated as
```

```
# onnx-file=models/[ModelName.pt]

# EXTRA STEP: Change config_infer_primary_yoloV11.txt variables.

network_mode=2 # instead of 0 (because we're working on a Jetson device)

# Set the CUDA_VER according to your DeepStream version
export CUDA_VER=12.2
```

5. Here, a library file is created `libnvdsinfer_custom_impl_Yolo.so` with the `MakerFile` and the other functions in the `/deepstream-yolo11/src/nvdsinfer_custom_impl_Yolo` folder. This is a file referenced in the `config_infer_primary_yoloV11.txt` file for the variable `custom-lib-path.` This variable is used when you add your custom model (yolo) to run with DeepStream.

```
# Compile the YOLO plugin
make -C nvdsinfer_custom_impl_Yolo clean && make -C
nvdsinfer_custom_impl_Yolo
```

6. Run Deepstream Application

```
deepstream-app -c deepstream_app_config.txt
```

Some documentation we found explains some things a bit more:

- https://docs.nvidia.com/metropolis/deepstream/dev-guide/text/DS_using_custom_model.html
- https://forums.developer.nvidia.com/t/implement-custom-yolo-or-any-custom-model-in-deepstream-using-python/235144

Step 5b. When Running on non-NVIDIA Devices

Export to a TensorRT Engine (5x faster)

After you've trained your model (and have a checkpoint like `best.pt`), export it with:

```
yolo export model=runs/detect/trainX/weights/best.pt format=engine
```

- Replace `trainX` with the folder name where your training run was saved.
- This command converts your PyTorch model to a TensorRT engine (e.g. `best.engine`) optimized for faster inference.

Optional: If your hardware supports FP16, you can enable half-precision by adding `half=True`:

```
yolo export model=runs/detect/trainX/weights/best.pt format=engine
half=True
```

Run Inference Using the TensorRT Engine

Once exported, use the engine file to run inference on your live camera feed:

```
yolo detect predict model=runs/detect/trainX/weights/best.engine source=0
imgsz=416 conf=0.25 show=true
```

- Make sure to replace `trainX` with your actual training run folder.
- `source=0` tells YOLO to use your default camera.
- `imgsz=416` and `conf=0.25` should match your training settings (adjust if needed).

Test it on a Live Camera Feed

Finally, to see how your model performs on a live feed, run the inference (predict) command using your best checkpoint:

```
yolo detect predict model=runs/detect/trainX/weights/best.pt source=0
imgsz=416 conf=0.25 show=true
```

Explanation:

- **source=0:** Uses the default camera (usually `/dev/video0` on Linux).
- **conf=0.25:** Sets the confidence threshold for displaying detections (adjust as needed).
- **show=true:** Displays the annotated camera feed in a window.

Step 6. Optimizing camera for better frames

If your FPS while running on a live-camera feed is too slow, try the following things.

Locate your camera using the following command:

```
ubuntu@localhost:~$ v4l2-ctl --list-devices
NVIDIA Tegra Video Input Device (platform:tegra-camrtc-ca):
        /dev/media0

AnkerWork C310 Webcam (usb-3610000.usb-3.3):
        /dev/video0
        /dev/video1
        /dev/media1
```

Once you know where your camera is located, look at the specs on different formats: (--device=”path/to/your/camera”)

```
ubuntu@localhost:~$ v4l2-ctl --device=/dev/video0 --list-formats-ext
ioctl: VIDIOC_ENUM_FMT
        Type: Video Capture

        [0]: 'YUYV' (YUYV 4:2:2)
                Size: Discrete 1920x1080
                        Interval: Discrete 0.017s (60.000 fps)
                        Interval: Discrete 0.033s (30.000 fps)
                Size: Discrete 1280x720
                        Interval: Discrete 0.017s (60.000 fps)
                        Interval: Discrete 0.033s (30.000 fps)
                Size: Discrete 960x640
                        Interval: Discrete 0.017s (60.000 fps)
                        Interval: Discrete 0.033s (30.000 fps)
```

Here you will get a list of resolutions where your camera will have the following fps. Try and choose one according to your needs.

Use GStreamer to try and create a pipeline to inspect your camera using the following command:

```
ubuntu@localhost:~$ gst-launch-1.0 v4l2src device=/dev/video0 ! videoconvert ! autovideosink
```

If this works, it confirms that the v4l2src plugin is being used by your USB-camera.

Now, choose the resolution where your camera runs at a stable fps and use MJPEG if possible, as it can improve frame rates.

When using Deepstream, adjust the app-config file to match your camera resolution.

Training on Unlabeled Data with Ultralytics YOLO

https://labelstud.io/

When you don't have pre-labeled data, you'll need to annotate your images before you can train an object detection model. This guide will walk you through the following steps:

1. **Collecting Your Data**
2. **Annotating Your Images**
3. **Organizing Your Dataset**
4. **Creating/Editing the Data Configuration File**
5. **Training the Model**

Step 1. Collecting Your Data

Gather the images you want to use for training. These might be captured from a camera, downloaded from the web, or collected from your projects. Save these images in a dedicated folder, for example:

```
mkdir -p ~/my_custom_dataset/images
```

Step 2. Annotating Your Images

Since your data is unlabeled, you need to annotate it. There are several tools available for this purpose:

- **LableStudio**: A popular open-source web-based interface
- **Roboflow**: Offers a web-based interface for annotation and dataset management.

Repeat for all images in your dataset.

Step 3. Train Your Model

Click here to go to step 3 to train your model (same workflow for now as labeled dataset).

Optimizing your AI models with YOLO

Source: https://www.youtube.com/watch?v=q7LwPoM7tSQ&ab_channel=Ultralytics

Effective model optimization requires a strategic analysis of key YOLO performance metrics. This guide details essential metrics, diagnostic tools, optimization strategies, and best practices for refining YOLO-based computer vision models.

Core Performance Metrics

1. **mAP (Mean Average Precision)**
    - Primary indicator of model accuracy across classes (IOU thresholds 0.5-0.95)
2. **Precision/Recall Balance**
    - High Precision: Indicates few false positives.
    - High Recall: Indicates few missed detections.
    - F1 Score: The harmonic mean of precision and recall, providing a balanced performance measure.
3. **IoU (Intersection over Union)**
    - Measures bounding box localization accuracy (predicted vs. ground truth)
4. **Speed Metrics**
    - Frames-per-second (FPS): A critical metric for real-time applications that reflects the speed at which the model processes images.

Diagnostic Tools

- **Confusion Matrix**: Identifies class-specific performance gaps
- **PR Curves**: Visualizes precision-recall trade-offs across confidence thresholds
- **Validation Visualizations**: Inspect predicted vs. actual bounding boxes

Optimization Strategies

Scenario	Diagnosis	Action
Low Precision/High Recall	Excessive false positives	Tighten confidence thresholds
Adequate mAP/Poor IoU	Bounding box localization errors	Refine annotation quality or augment box data
Class-specific mAP drops	Data imbalance or rare classes	Apply class weights or targeted data augmentation

Best Practices

- Validate metrics against visual predictions (avoid "100% precision" traps)
- Export models to hardware-optimized formats (TensorRT/ONNX) after tuning
- Prioritize speed-accuracy tradeoffs based on application requirement

Comparison between running on YOLO vs running on Deepstream

This is tested on the Jetson AGX Orin™.

Because deepstream is optimized for devices running on NVIDIA, you have:

- A higher framerate
- Better object detection
- Etc

TODO: We should test Yolo on non-nvidia devices

Installing Ultralytics and DeepStream with Docker for Jetson Devices

1. Follow this documentation: https://docs.nvidia.com/metropolis/deepstream/dev-guide/text/DS_docker_containers.html
2. Pull latest deepstream docker image for jetson devices (as off 10/03/2025): docker pull nvcr.io/nvidia/deepstream:7.1-samples-multiarch