Internship/Training Project Report

On

COMPARISON OF TRAINING AND INFERENCE OF DIFFERENT VARIANTS OF LATEST YOLO ALGORITHM

At

Research Centre Imarat
Defence Research and Development Organization
Vigyanakancha, Hyderabad-500069

Under the Guidance of
Dr. J.V. Satyanarayana, Sc 'G' of DEAIS

CONTENTS

ABSTRACT

Object detection is a cornerstone of computer vision, enabling a broad spectrum of applications in areas such as autonomous driving, surveillance, healthcare, and robotics. This project delves into advanced object detection methodologies, with a particular focus on the YOLO (You Only Look Once) algorithm and its latest iteration, YOLOv11. Unlike traditional approaches, YOLO employs a single-pass detection framework that integrates object classification and localization into a unified neural network, significantly improving real-time performance and computational efficiency. The algorithm's evolutionary advancements from YOLOv1 to YOLOv11 are critically examined, emphasizing key innovations such as CSPDarknet backbones, Mish activations, anchor-free detection, and multi-scale feature aggregation techniques. These improvements collectively enhance the algorithm's ability to handle complex scenarios involving small objects, intricate backgrounds, and diverse lighting conditions.

This report provides a comprehensive comparative analysis of the training and inference performance of YOLOv11 algorithm variants, including Nano, Small, Medium, Large, and Extra Large (X). Each variant is meticulously evaluated to highlight its unique capabilities, strengths, and trade-offs. For training performance, the report examines critical metrics such as model architecture, training loss trends, computational requirements, and overall training time. These factors offer an in-depth understanding of the efficiency, scalability, and practicality of each variant during the model development phase.

For inference performance, a rigorous evaluation is conducted using metrics including the confusion matrix, F1 scores, precision values (average precision and mean average precision), recall values, and inference speed. These metrics provide valuable insights into the real-world applicability and deployment feasibility of each YOLOv11 variant, enabling a nuanced understanding of their operational trade-offs.

The experimental framework leverages the resources of Google Colab, utilizing NVIDIA Tesla T4 GPUs to maintain a consistent hardware and software environment. This ensures the reliability, reproducibility, and comparability of results across all experiments. The analysis reveals that the Nano variant, optimized for edge devices, achieves remarkable speed, processing up to 80 frames per second (FPS), making it an excellent choice for real-time applications requiring low latency. On the other hand, the Extra Large (X) variant excels in precision and recall metrics, making it suitable for high-accuracy tasks in domains where computational resources and latency constraints are less critical.

This detailed study underscores the versatility and adaptability of YOLOv11 across a wide range of computational environments and application scenarios. By analyzing the trade-offs between performance, computational efficiency, and resource utilization, the findings provide a robust framework for selecting the most appropriate YOLOv11 variant for specific use cases. Furthermore, this work contributes to advancing the state-of-the-art in object detection, bridging the gap between speed and accuracy, and laying the foundation for its application in dynamic and real-world environments.

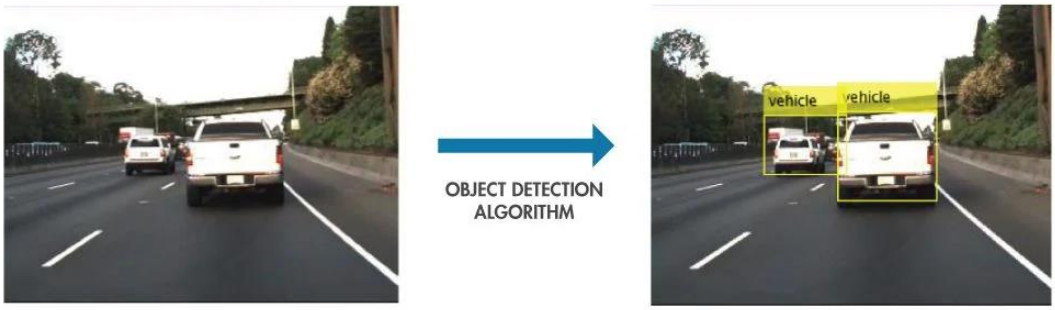
INTRODUCTION

Object Detection

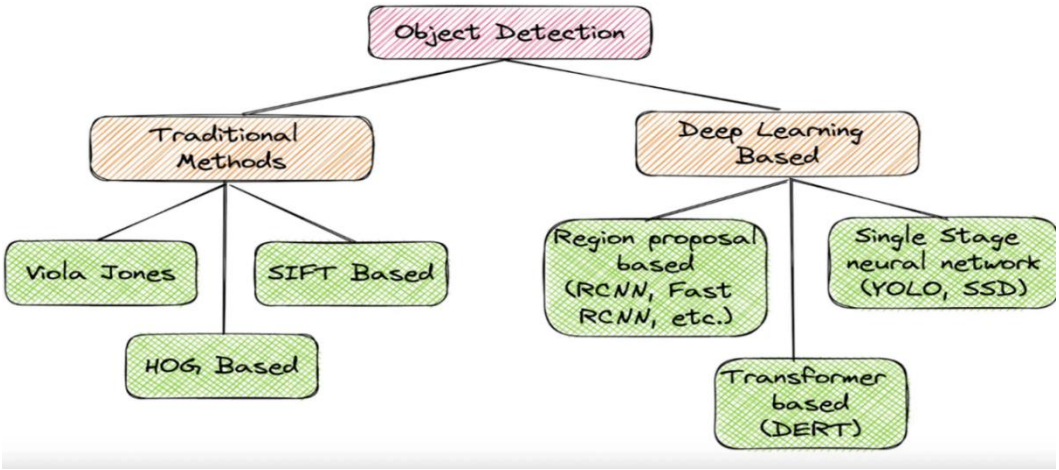
Object detection is a crucial task in computer vision that involves identifying and localizing objects within images or video frames. This task has gained significant attention due to its wide range of applications, including autonomous driving, surveillance, healthcare, and robotics. Recent advancements in deep learning, particularly *Convolutional Neural Networks (CNNs)*, have markedly improved the performance and accuracy of object detection techniques.

Key Components of Object Detection:

1. **Class Label Prediction**: Object detection systems must categorize detected objects into predefined classes, such as humans, animals, cars, or buildings. Robust models are essential for distinguishing between different object categories under varying conditions like changes in lighting, orientation, and occlusion.
2. **Localization**: In addition to classifying objects, detection models must also accurately locate each object within the image. This is typically achieved by predicting bounding boxes that enclose the objects, defined by coordinates representing the top-left and bottom-right corners of the box.



- **Approaches used in Object Detection**



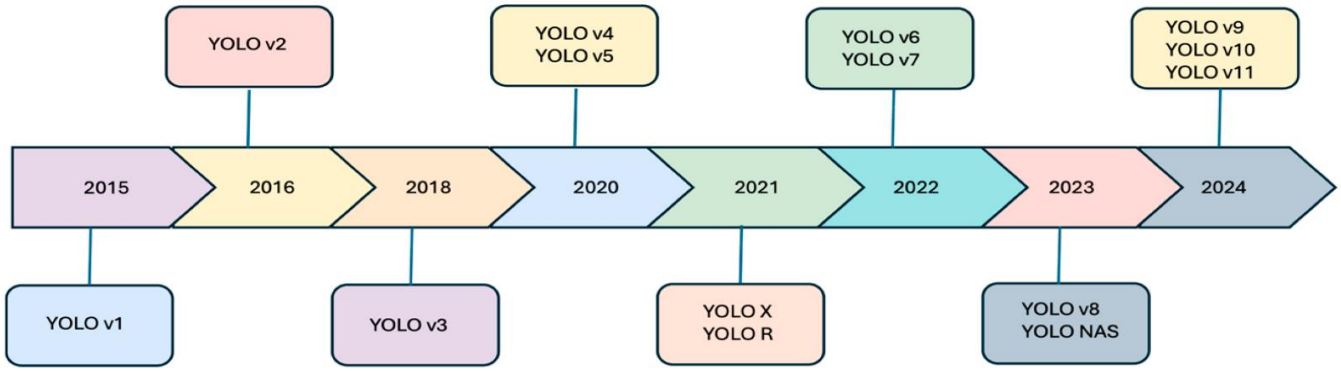
YOLO

You Only Look Once (YOLO) proposes using an end-to-end *neural network* that makes predictions of bounding boxes and class probabilities all at once. It differs from the approach taken by previous object detection algorithms, which repurposed classifiers to perform detection.

Following a fundamentally different approach to object detection, YOLO achieved state-of-the-art results, beating other real-time object detection algorithms by a large margin. While algorithms like Faster RCNN work by detecting possible regions of interest using the Region Proposal Network and then performing recognition on those regions separately, YOLO performs all of its predictions with the help of a single fully connected layer.

Methods that use Region Proposal Networks perform multiple iterations for the same image, while YOLO gets away with a single iteration. Several new versions of the same model have been proposed since the initial release of YOLO in 2015, each building on and improving its predecessor. Here's a timeline showcasing YOLO's development in recent years.

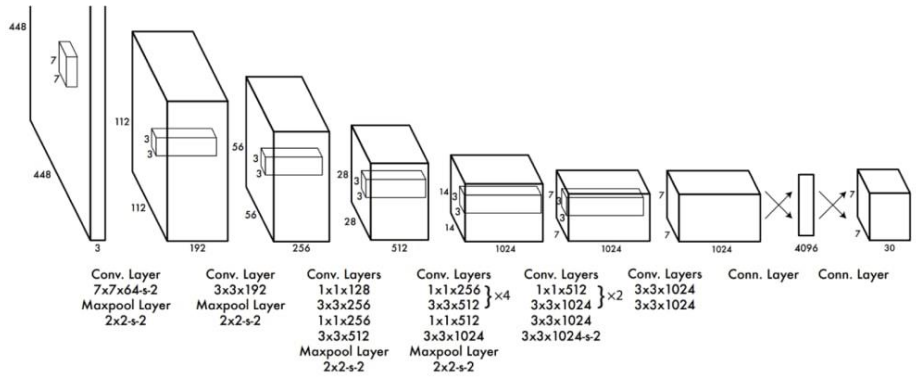
Evolution of YOLO



- **YOLOv1**: Introduced real-time object detection using a single neural network and grid-based predictions.
- **YOLOv2 (YOLO9000)**: Enhanced accuracy and speed with batch normalization, higher resolution input images, and anchor boxes.
- **YOLOv3**: Improved small object detection with a deeper network (Darknet-53) and multi-scale predictions.
- **YOLOv4**: Boosted performance using CSPDarknet53, Mish activation, and PANet for feature aggregation.
- **YOLOv5**: Known for ease of use, fast deployment, and employing smaller, faster models.
- **YOLOv7**: Improved memory efficiency and gradient propagation with E-ELAN computational block, enhancing learning power.
- **YOLOv8**: Brought innovations in detection heads and anchor-free approaches, further enhancing performance metrics.
- **YOLOv9**: Continued advancements in model efficiency and accuracy, integrating new optimization techniques.
- **YOLOv10**: Revolutionizes object detection with dual assignments eliminating NMS, enhancing accuracy, and reducing computational overhead.
- **YOLOv11**: Adopted transformer-based backbone and dynamic head design, advancing accuracy and real-time efficiency.

YOLO Architecture

The YOLO Algorithm takes an **image** as input and then uses a simple deep convolutional neural network to detect objects in the image. The architecture of the CNN model that forms the backbone of YOLO is shown below.



The Architecture. Our detection network has 24 convolutional layers followed by 2 fully connected layers. Alternating 1×1 convolutional layers reduce the features space from preceding layers. We pretrain the convolutional layers on the ImageNet classification task at half the resolution (224×224 input image) and then double the resolution for detection.

- The first *20 convolution layers* of the model are pre-trained using **ImageNet** by plugging in a temporary average pooling and fully connected layer. Then, this pre-trained model is converted to perform detection since previous research showcased that adding convolution and connected layers to a pre-trained network improves performance. YOLO's final fully connected layer predicts both class probabilities and bounding box coordinates.

- YOLO divides an input image into an **S × S** grid. If the centre of an object falls into a grid cell, that grid cell is responsible for detecting that object. Each grid cell predicts B bounding boxes and confidence scores for those boxes. These confidence scores reflect how confident the model is that the box contains an object and how accurate it thinks the predicted box is.

- YOLO predicts multiple bounding boxes per grid cell. At training time, we only want one bounding box predictor to be responsible for each object. YOLO assigns one predictor to be "responsible" for predicting an object based on which prediction has the highest current **IOU** with the ground truth. This leads to specialization between the bounding box predictors. Each predictor gets better at forecasting certain sizes, aspect ratios, or classes of objects, improving the overall recall score.

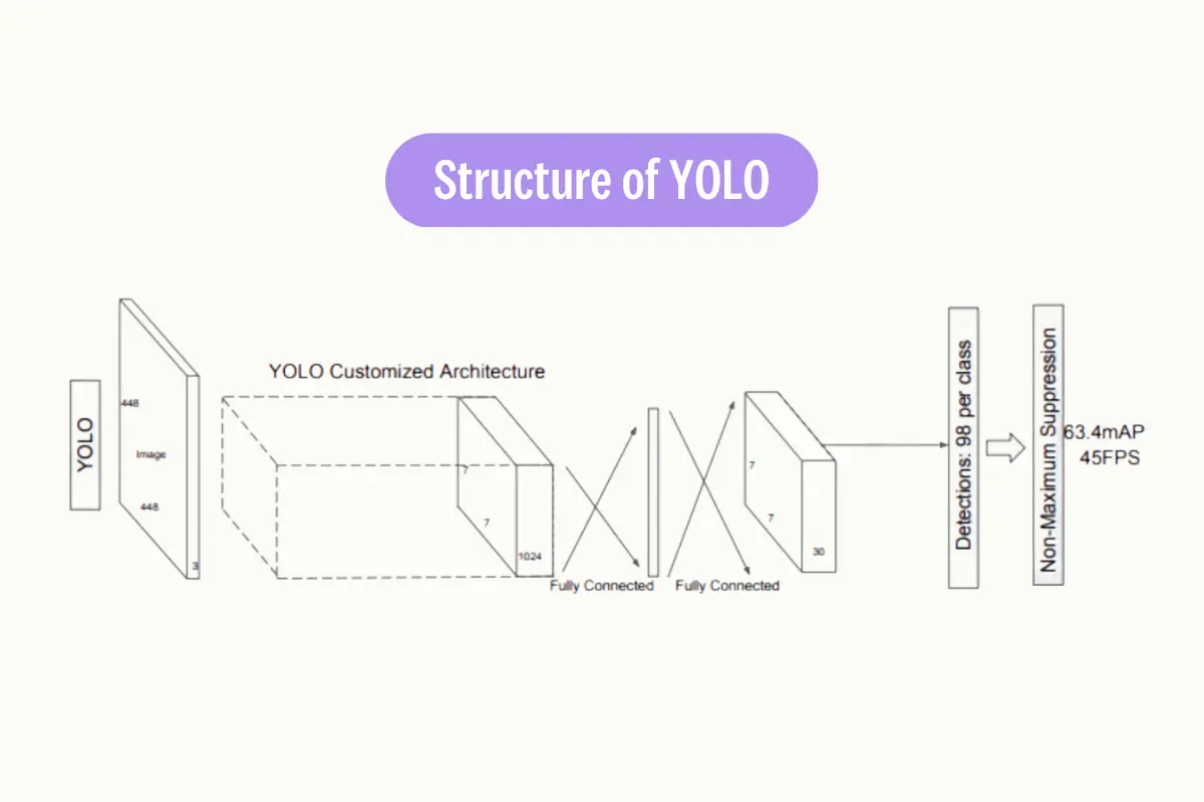
One key technique used in the YOLO models is **non-maximum suppression (NMS)**. NMS is a post-processing step that is used to improve the accuracy and efficiency of object detection. In object detection, it is common for multiple bounding boxes to be generated for a single object in an image. These bounding boxes may overlap or be located at different positions, but they all represent the same object. NMS is used to identify and remove redundant or incorrect bounding boxes and to output a single bounding box for each object in the image.

Working of YOLO

The basic idea behind YOLO is to divide the input image into a grid of cells and, for each cell, predict the probability of the presence of an object and the bounding box coordinates of the object. The process of YOLO can be broken down into several steps:

1. Input image is passed through a CNN to extract features from the image.
2. The features are then passed through a series of fully connected layers, which predict class probabilities and bounding box coordinates.
3. The image is divided into a grid of cells, and each cell is responsible for predicting a set of bounding boxes and class probabilities.
4. The output of the network is a set of bounding boxes and class probabilities for each cell.
5. The bounding boxes are then filtered using a post-processing algorithm called non-max suppression to remove overlapping boxes and choose the box with the highest probability.
6. The final output is a set of predicted bounding boxes and class labels for each object in the image.

One of the key advantages of YOLO is that it processes the entire image in one pass, making it faster and more efficient than two-stage object detectors such as R-CNN and its variants.



Comparison Between YOLO, R-CNN & Fast R-CNN

R-CNN	Fast RCNN	YOLO
Based on classification	Based on classification	Based on regression
Uses 2000 conv Nets for each region	Uses single deep convent	Uses single convolution network for entire image
Uses selective search algorithm	Uses selective search algorithm	---
Needs 49 seconds to test one image	Needs 2.3 seconds to test one image	Needs less than 2 second to test on one image
Slow, cannot be implemented real time	Faster than RCNN	Can run real time
Uses SVM for classification	Uses softmax for classification	Uses regression for classification
Produces bounding boxes	Produces bounding Box regression head and classification head	Produces bounding box, prediction contextual concurrently
Can find small objects	Can find small objects	Struggles to find small objects that appear in groups

- ## Why should I use a YOLO model?

YOLO models are popular for a variety of reasons.

- First, YOLO models are fast. This enables you to use YOLO models to process video feeds at a high frames-per-second rate. This is useful if you are planning to run live inference on a video camera to track something that changes quickly (i.e. the position of a ball on a football court, or the position of a package on a conveyor belt).
- Second, YOLO models continue to lead the way in terms of accuracy. The YOLOv7 model, the latest in the family of models as of November 2022, has state-of-the-art performance when measured against the MS COCO object detection dataset. While there are other great models out there, YOLO has a strong reputation for its accuracy.
- Third, the YOLO family of models are open source, which has created a vast community of YOLO enthusiasts who are actively working with and discussing these models. This means that there is no shortage of information on the web about the how and why behind these models, and getting help is not difficult if you reach out to the computer vision community.
- Combining all of these three factors, it is clear why so many engineers use YOLO to power their computer vision models.

METHODOLOGY

- **Environment**: Used Google Colab Notebook and utilized its NVIDIA Tesla T4 GPU in all the variants.
- **Dataset**: Dataset is taken from roboflow and can be downloaded using following code compatible with yolo11.

```
!pip install roboflow

from roboflow import Roboflow
rf = Roboflow(api_key="YOUR_API_KEY")
project = rf.workspace("ardi-csjyk").project("ppe-hfjoc")
version = project.version(2)
dataset = version.download("yolov11")
```

The dataset is downloaded into the PPE-2 folder, organized in the following structure:

```
PPE-2
+---data.yaml
|
+---test
|   +---images
|   +---labels
+---train
|   +---images
|   +---labels
+---valid
    +---images
    +---labels
```

Content of data.yaml file:-

```
train: ../train/images
val: ../valid/images
test: ../test/images

nc: 6
names: ['glasses', 'gloves', 'helmet', 'mask', 'safety-shoes', 'vest']

roboflow:
  workspace: ardi-csjyk
  project: ppe-hfjoc
  version: 2
  license: MIT
  url: https://universe.roboflow.com/ardi-csjyk/ppe-hfjoc/dataset/2
```

This means that the trained model can detect six classes of objects: glasses, gloves, helmets, masks, safety shoes, and vests.

- **Libraries/Dependencies**: YOLO models are available in the ultralytics library, which can be installed using the following command:

```
!pip install ultralytics
```

We can use YOLO algorithm by simply importing it from ultralytics library:-

```
from ultralytics import YOLO
```

- **Model Loading**: Different variants of YOLO object detection models can be utilized by loading their pre-trained weights using the sample code below.

  Using pre-trained weights is an efficient way to accelerate the training process. These weights are derived from models trained on extensive datasets (e.g., COCO), providing a strong foundation for the model. Through transfer learning, these pre-trained models can be adapted to new, related tasks. Fine-tuning involves starting with pre-trained weights and further training the model on a specific dataset. This approach leads to faster training times and improved performance, as the model benefits from an initial understanding of fundamental features.

```
model = YOLO('yolo-VARIANT_NAME.pt')
```

- **Training Set**: The training set is the primary dataset used for teaching the model. The model observes and learns patterns from this data to make predictions or informed decisions.

  The model can be trained using the following code:

```
train_results = model.train(
    data = '/content/PPE-2/data.yaml',
    epochs = 30,
    batch = 12,
    imgsz = 640
)
```

  Training Hyperparameters:-

  - **Learning Rate (lr0)**: Determines the step size at each iteration while moving towards a minimum in the loss function. By default, its value is 0.01.
  - **epochs**: Total number of training epochs. Each epoch represents a full pass over the entire dataset. Adjusting this value can affect training duration and model performance. By default, it is 100.
  - **batch**: The number of training samples processed before the model's internal parameters are updated. By default, it is 16.
  - **imgsz**: Determines the target image size for training. All images are resized to this dimension before being fed into the model. Affects model accuracy and computational complexity. By default, it is 640.
  - **Optimizer**: Determines the choice of optimizer for training. Options include SGD, Adam, AdamW, NAdam, RAdam, RMSProp etc., or auto for automatic selection based on model configuration. Affects convergence speed and stability. By default, it is set to auto. In our case, AdamW is used.

- **Training Metrics:**
  - **Losses**: In the context of training an object detection model, loss functions play a critical role. Loss functions quantify the discrepancy between the predicted bounding boxes and the ground truth annotations, providing a measure of how well the model is learning during training. Common loss components in object detection include:

- **box_loss**: Box loss measures the error in predicting the coordinates of bounding boxes. It encourages the model to adjust the predicted bounding boxes to align with the ground truth boxes.
- **cls_loss**: Class loss quantifies the error in predicting the object class for each bounding box. It ensures that the model accurately identifies the object's category.
- **dfl_loss**: Evaluates the quality of the predicted probability distributions for bounding box coordinates. This component ensures that the model accurately predicts the localization of object boundaries by focusing on the distributional aspects of the predictions.

- **Load the trained model:** After training, our trained model is saved as *best.pt*. It can be used by loading the model:-

```
trained_model = YOLO('/content/runs/detect/train/weights/best.pt')
```

- **Testing Set:** This dataset is independent of the training set but has a somewhat similar type of probability distribution of classes and is used as a benchmark to evaluate the model, used only after the training of the model is complete.
  The model can be evaluated using following code:-

```
metrics = trained_model.val(
    data = '/content/PPE-2/data.yaml',
    split = 'test'
)
```

  The model can be used for detecting objects by:-

```
predictions = trained_model.predict(
    source = '/content/PPE-2/test/images',
    save = True
)
```

- **Testing/Inference Metrics:-**
  - **Intersection over Union (IoU)**: Intersection over Union (IoU) is a crucial metric in object detection, providing a quantitative measure of the degree of overlap between a ground truth (gt) bounding box and a predicted (pd) bounding box generated by the object detector. This metric is fundamental for assessing the accuracy of object detection models and is used to define key terms such as True Positive (TP), False Positive (FP), and False Negative (FN).
    IoU is computed by taking the ratio of the area of intersection between the gt and pd bounding boxes to the area of their union. The IoU value ranges from 0 to 1, where 0 indicates no overlap between the two boxes, and 1 represents a perfect match or complete overlap.

$$\text{IoU} = \frac{\text{area}(gt \cap pd)}{\text{area}(gt \cup pd)}$$

- **Definition of terms**:
    - **True Positive (TP)** — Correct detection made by the model.
    - **False Positive (FP)** — Incorrect detection made by the detector.
    - **False Negative (FN)** — A Ground-truth missed (not detected) by the object detector.
    - **True Negative (TN)** —This is the background region correctly not detected by the model. This metric is not used in object detection because such regions are not explicitly annotated when preparing the annotations.

- **Confusion Matrix**: The confusion matrix provides a detailed view of the outcomes, showcasing the counts of true positives, true negatives, false positives, and false negatives for each class.
- **Normalized Confusion Matrix**: This visualization is a normalized version of the confusion matrix. It represents the data in proportions rather than raw counts. This format makes it simpler to compare the performance across classes.
- **F1 Score**: The F1 Score is the harmonic mean of precision and recall, providing a balanced assessment of a model's performance while considering both false positives and false negatives.
- **Mean Average Precision (mAP):** mAP extends the concept of AP by calculating the average AP values across multiple object classes. It looks at the precision of detecting each object class, averages these scores, and gives an overall number that shows how accurately the model can identify and classify objects.
  Let's focus on two specific mAP metrics:
    - **mAP@0.5**: Measures the average precision at a single IoU (Intersection over Union) threshold of 0.5. This metric checks if the model can correctly find objects with a looser accuracy requirement. It focuses on whether the object is roughly in the right place, not needing perfect placement. It helps see if the model is generally good at spotting objects.
    - **mAP@0.5-0.95**: Averages the mAP values calculated at multiple IoU thresholds, from 0.5 to 0.95 in 0.05 increments. This metric is more detailed and strict. It gives a fuller picture of how accurately the model can find objects at different levels of strictness and is especially useful for applications that need precise object detection.
- **Precision**: Precision is the degree of exactness of the model in identifying only relevant objects. It is the ratio of TPs over all detections made by the model. Precision Curve is a graphical representation of precision values at different thresholds. This curve helps in understanding how precision varies as the threshold changes.
- **Recall**: Recall measures the ability of the model to detect all ground truths— proposition of TPs among all ground truths. Recall Curve is a graph illustrating how the recall values change across different thresholds.

$$P = \frac{\text{TP}}{\text{TP} + \text{FP}} = \frac{TP}{\text{all detections}}$$

$$R = \frac{\text{TP}}{\text{TP} + \text{FN}} = \frac{TP}{\text{all ground-truths}}$$

- **Precision-Recall Curve**: An integral visualization for any classification problem, this curve showcases the trade-offs between precision and recall at varied thresholds. It becomes especially significant when dealing with imbalanced classes. Average Precision (AP) computes the area under the precision-recall curve, providing a single value that encapsulates the model's precision and recall performance.
- **Model Speed**: Below timing breakdown provides insights into the computational speed of the YOLO model pipeline when processing a single image. Here's what each component means:
  1. **Preprocess**: The time taken to prepare the input image for the model. This involves tasks such as:-
     - Resizing the image to the required input size (e.g., 640x640).
     - Normalizing pixel values.
     - Converting the image to the appropriate tensor format.

     Importance: Efficient preprocessing ensures that the model can handle inputs quickly, minimizing latency.

  2. **Inference**: The time the model takes to pass the image through its layers and make predictions. This is the core computation time for the neural network. Lower inference time is crucial for real-time applications, such as object detection in videos or live streams.
  3. **Postprocess**: The time taken to process the raw output of the model into human-readable or usable results. This includes:
     - Applying non-maximum suppression (NMS) to remove redundant bounding boxes.
     - Decoding class probabilities and bounding box coordinates.
     - Formatting results for output.

     Importance: Faster postprocessing ensures the predictions are ready for use with minimal delay.

DIFFERENT VARIANTS OF YOLO11

1. n-variant (nano):

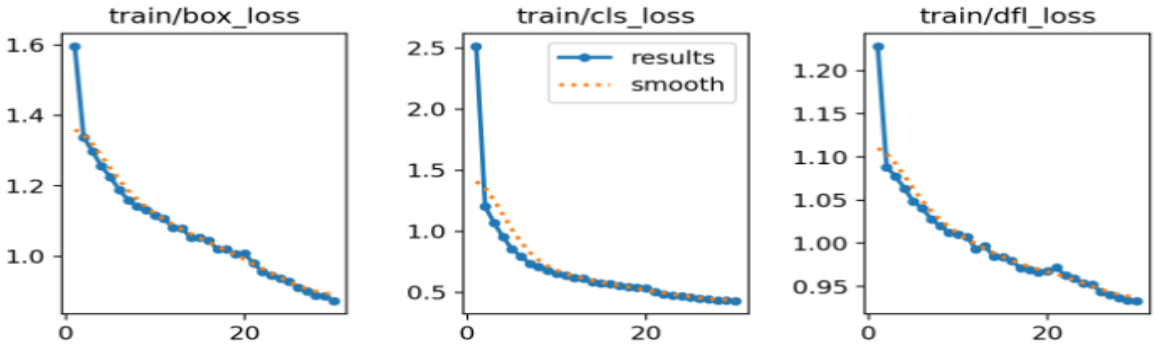
It is mainly designed for edge devices with low computational power; prioritizes speed over accuracy. It contains:-

- **319 layers**: This indicates that the model consists of 319 layers, which might include convolutional layers, normalization layers, activation functions, and other operations.
- **2,591,010 parameters**: These are the total trainable parameters in the model. Parameters are the weights and biases of the neural network, which are learned during training. This small number of parameters indicates the model is lightweight and optimized for use in environments with limited computational resources.
- **2,590,994 gradients**: Gradients are the derivatives of the loss function with respect to the parameters, used during backpropagation to update the model's parameters. The number of gradients is typically the same as the number of trainable parameters (or nearly identical), as each parameter has a corresponding gradient.

- **6.4 GFLOPs**: GFLOPs (Giga Floating Point Operations per Second**)** measure the computational complexity of a model, indicating the number of operations needed for a single forward pass. Lower GFLOPs mean higher efficiency, ideal for real-time use or resource-limited devices.
  - Nano variant can be loaded by the following code:-

```
model = YOLO('yolo11n.pt')
```

- # Training Metrics:
  - **Losses during 30 epochs**



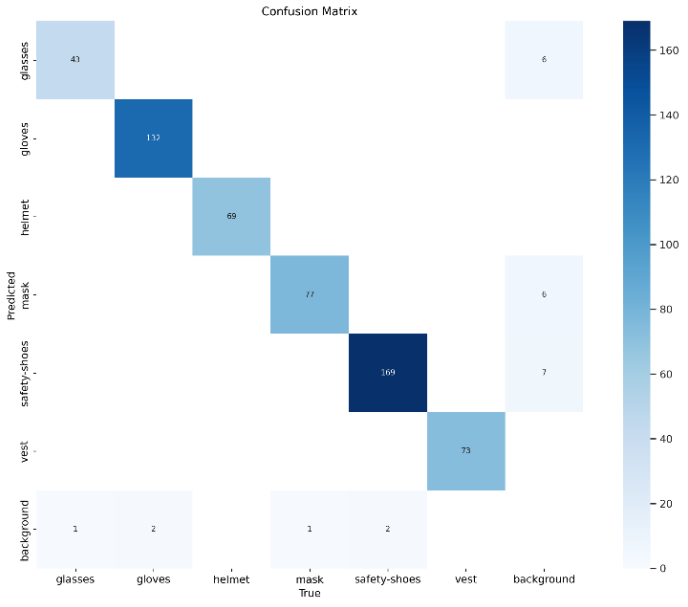
After 30 epochs, final losses were:-

box_loss	cls_loss	dfl_loss
0.8725	0.4279	0.933

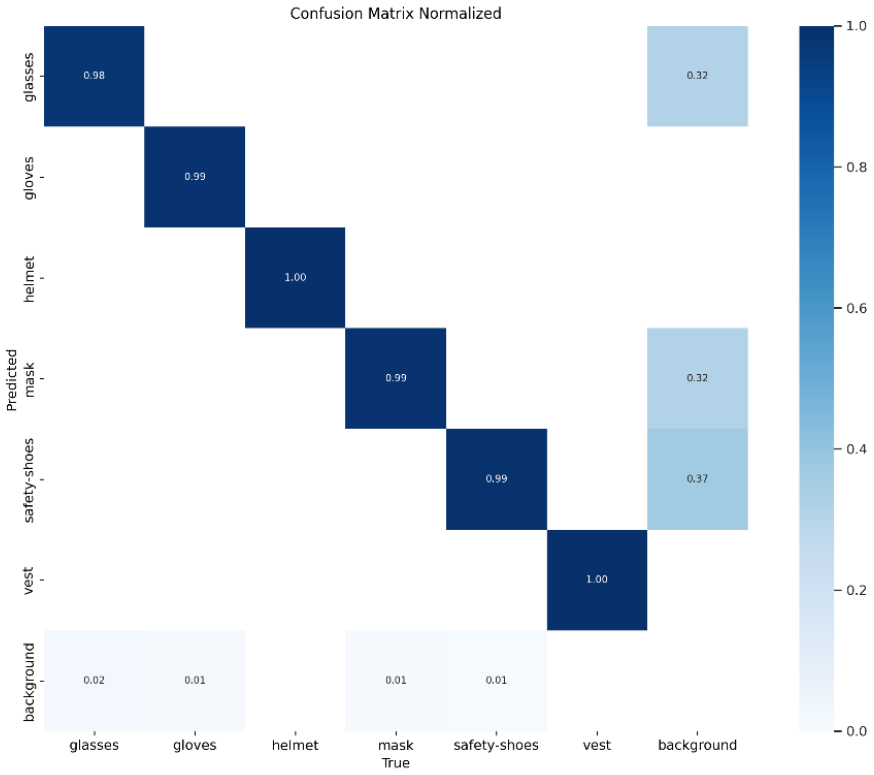
  - **GPU Memory Usage per Epoch**: Approximately 1.8 GB of GPU memory was utilized during each epoch.
  - **Time Taken per Epoch**: Each epoch required approximately 45–50 seconds for training.
  - **Total Training Time**: The entire training process took 0.42 hours, equivalent to 25 minutes and 12 seconds.
- # Testing/Inference Metrics:

On testing our trained model on the testing dataset, we got the following results:-

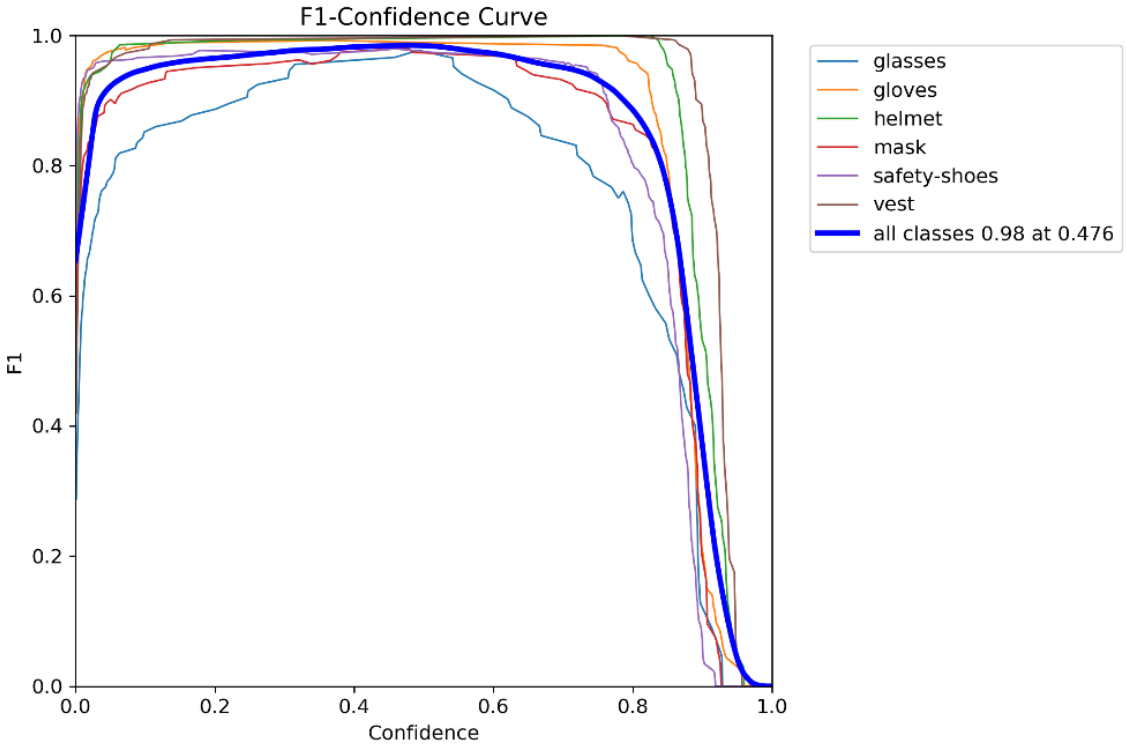
  - **Confusion Matrix**:



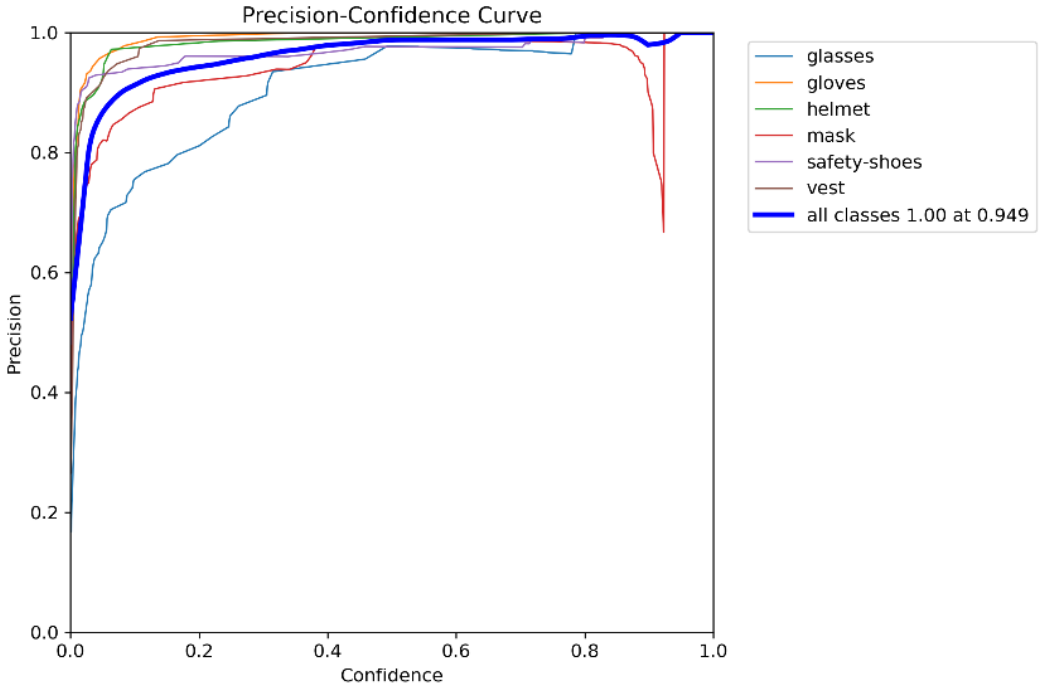
- **Normalized Confusion Matrix**:



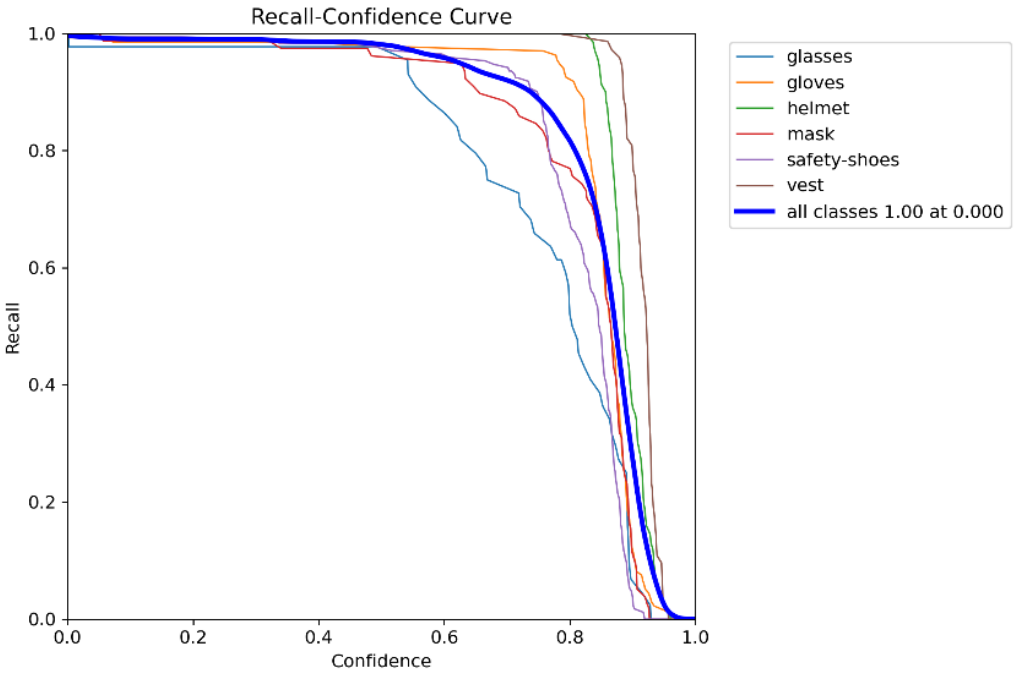
- **F1 Curve**:



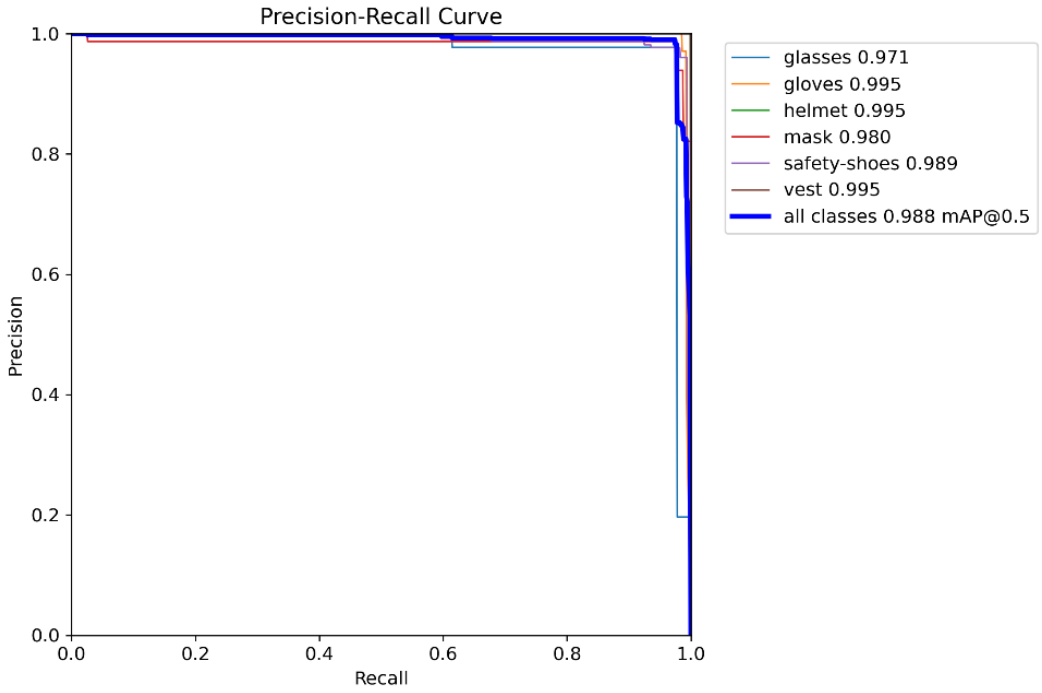
- **P Curve**:



- **R Curve**:



- **PR Curve**:



- **Class wise precisions on test dataset**:

Class	Images	Instances	Box(P	R	mAP50	mAP50-95):
all	120	569	0.985	0.985	0.988	0.716
glasses	44	44	0.963	0.977	0.971	0.55
gloves	73	134	1	0.98	0.995	0.718
helmet	69	69	0.991	1	0.995	0.778
mask	78	78	0.986	0.974	0.98	0.694
safety-shoes	85	171	0.977	0.98	0.989	0.706
vest	73	73	0.993	1	0.995	0.847

- **Speed of the model**:-
  - 0.7ms preprocess per image
  - 7.1ms inference per image
  - 4.7ms postprocess per image

  Total Time Per Image (Latency):-

  Sum: 0.7 + 7.1 + 4.7 = 12.5 ms
  This means the model processes one image in approximately 12.5 milliseconds, translating to 80 FPS (frames per second).

2. s-variant (small)

It balances speed and accuracy for lightweight applications. It contains:-

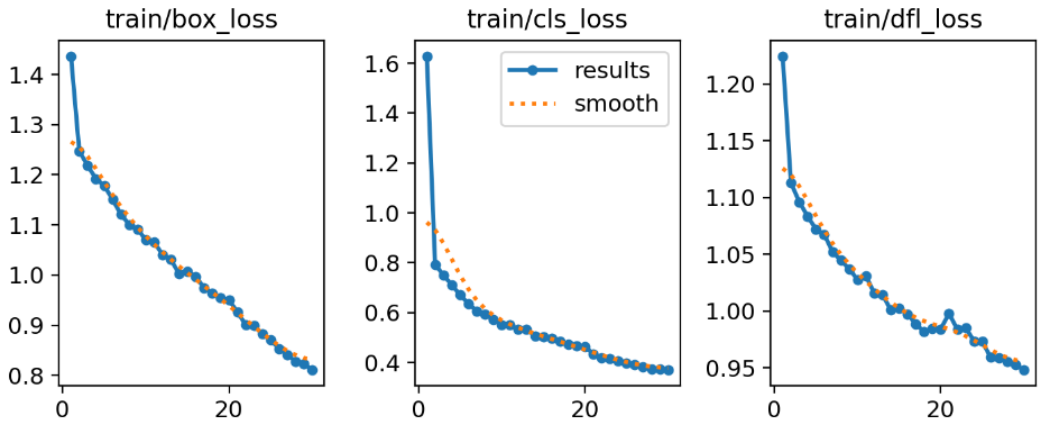
- 319 layers: Number of layers are same as that of nano variant.
- 9,430,114 parameters: Number of parameters are more than that of nano variant.
- 9,430,098 gradients: Number of gradients are also more than nano variant.
- 21.6 GFLOPs: It is more than nano variant, this means that the computational complexity of small variant is higher than that of nano variant.

    - Small variant can be used/loaded by the following code:-

```
model = YOLO("yolo11s.pt")
```

○ Training Metrics:

- Losses during 30 epochs



After 30 epochs, final losses are:-

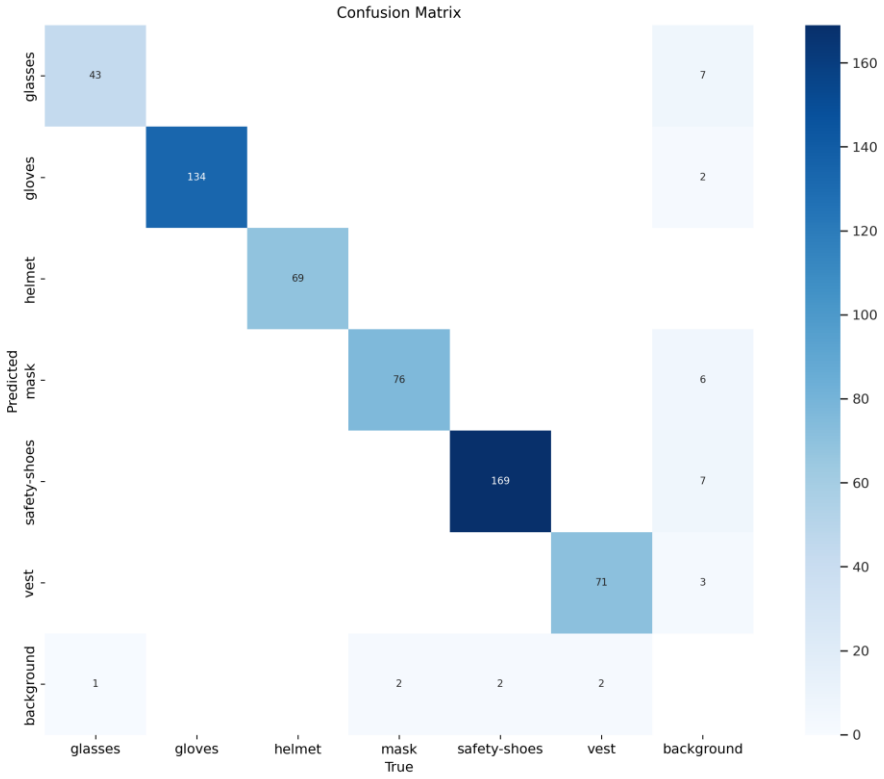
box_loss	cls_loss	dfl_loss
0.8107	0.3697	0.948

- **GPU Memory Usage per Epoch**: Approximately 3.3 GB of GPU memory was utilized during each epoch.
- **Time Taken per Epoch**: Each epoch required approximately 50–55 seconds for training.
- **Total Training Time**: The entire training process took 0.47 hours, equivalent to 28 minutes and 12 seconds.

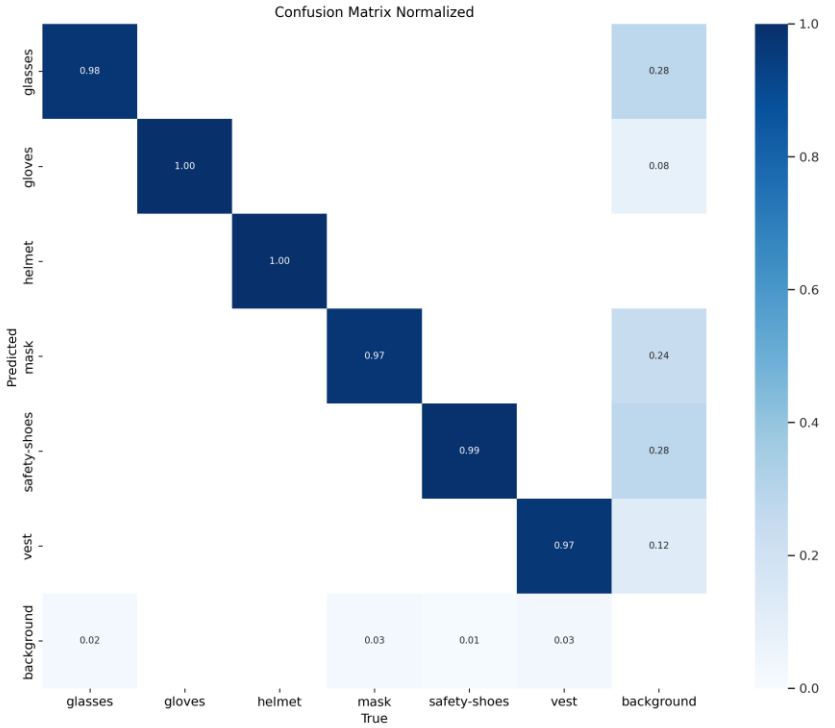
• Testing/Inference Metrics:

On testing our trained model on the testing dataset, we got the following results:-

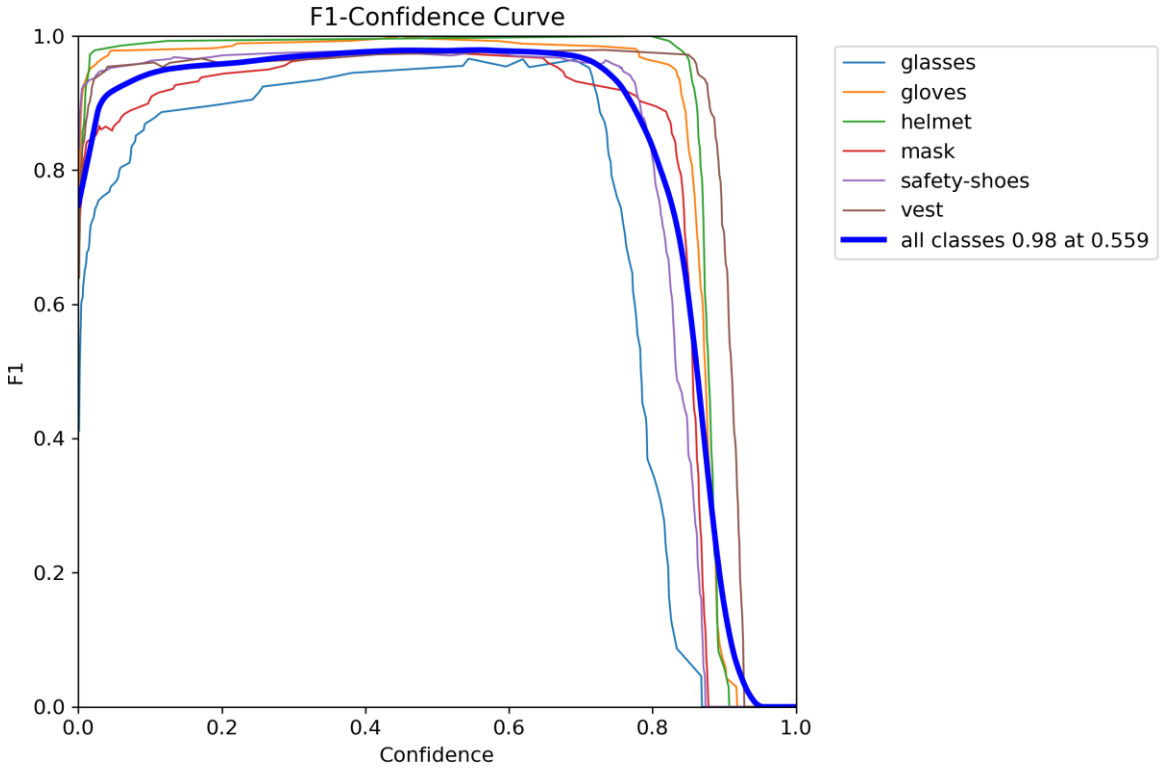
- **Confusion Matrix**:



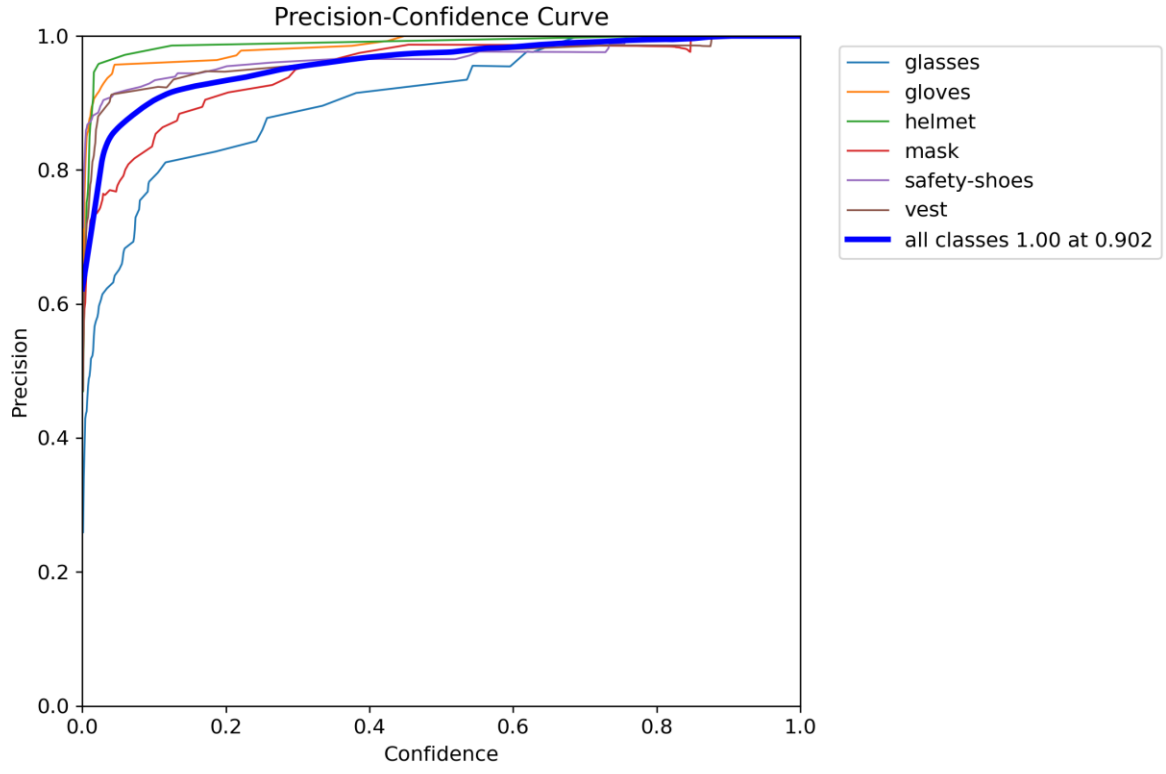
- **Normalized Confusion Matrix**:



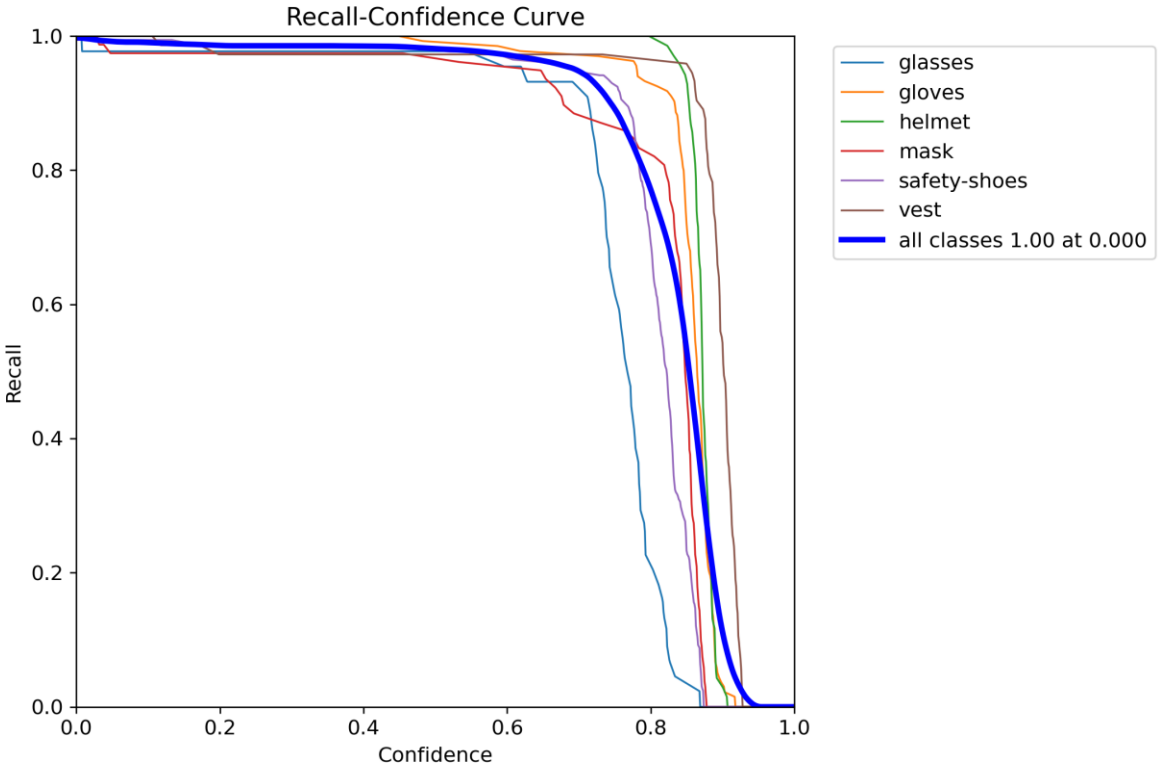
- **F1 Curve**:



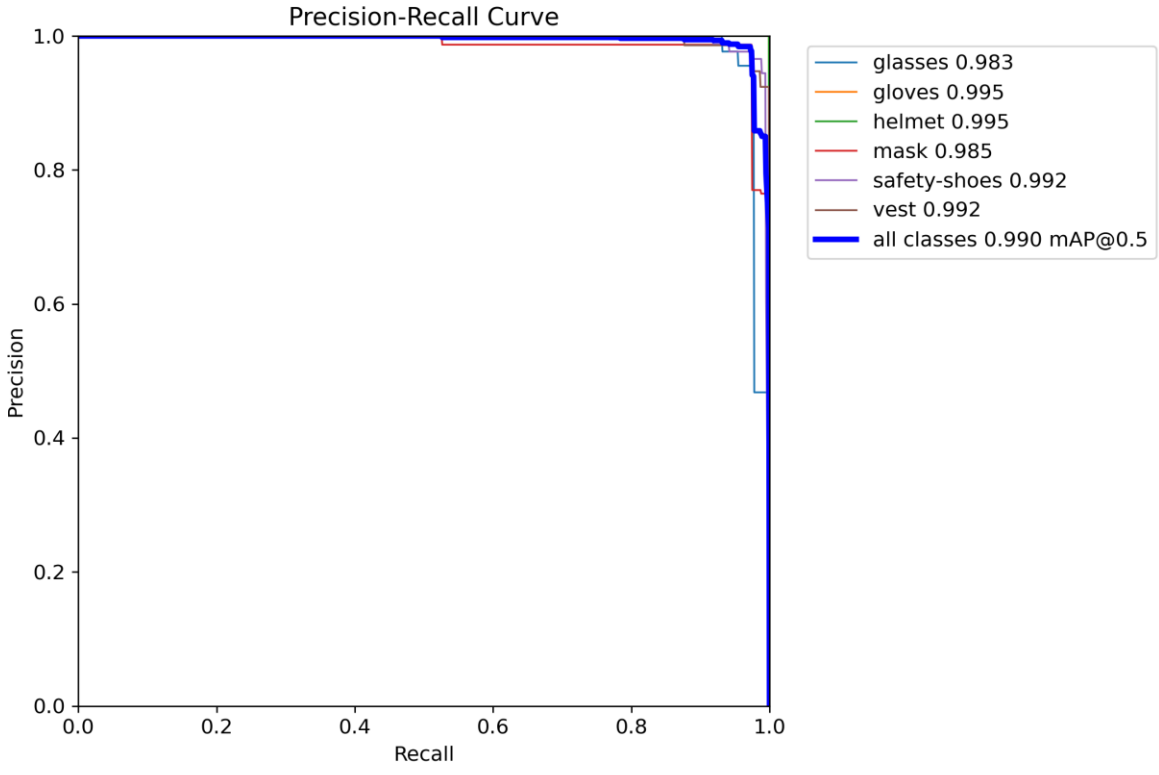
- **P Curve**:



- **R Curve**:



- **PR Curve**:



- **Class wise precisions on test dataset**:

Class	Images	Instances	Box(P	R	mAP50	mAP50-95):
all	120	569	0.977	0.98	0.99	0.742
glasses	44	44	0.934	0.977	0.983	0.589
gloves	73	134	1	0.989	0.995	0.746
helmet	69	69	0.994	1	0.995	0.803
mask	78	78	0.987	0.962	0.985	0.73
safety-shoes	85	171	0.968	0.977	0.992	0.715
vest	73	73	0.977	0.973	0.992	0.868

- **Speed of the model**:-
  - 0.4ms preprocess per image
  - 9.4ms inference per image
  - 3.0ms postprocess per image

  Total Time Per Image (Latency):-

  Sum: 0.4 + 9.4 + 3.0 = 12.8 ms
  This means the model processes one image in approximately 12.8 milliseconds, translating to 78.125 FPS (frames per second).

3. m-variant (medium)

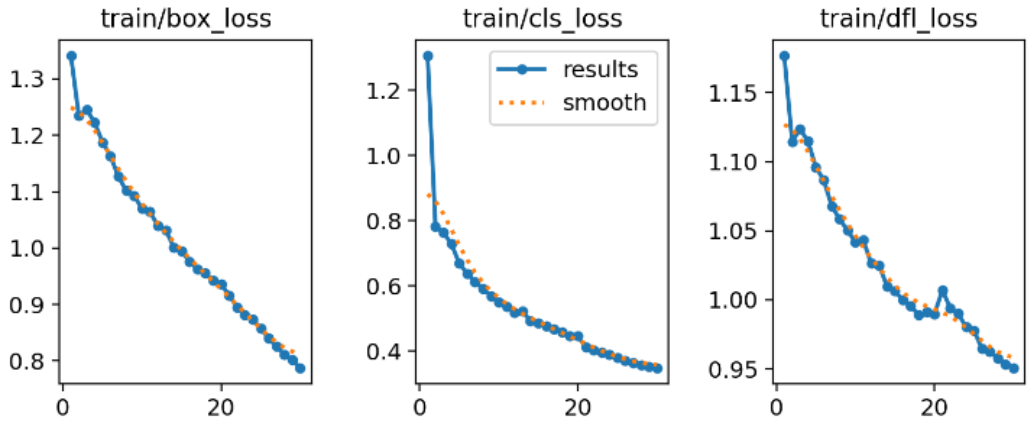
It aims for moderate computational requirements while delivering higher accuracy. It contains:-

- 409 layers: Number of layers are more than both the nano and small variants.
- 20,057,634 parameters: Number of parameters are more than that of small variant.
- 20,057,618gradients: Number of gradients are also more than small variant.
- 68.2 GFLOPs: It is more than small variant, meaning even more computational complexity than small variant.

  - Medium variant can be used/loaded by the following code:-

```
model = YOLO('yolo11m.pt')
```

- **Training Metrics**:
  - **Losses during 30 epochs**



After 30 epochs, final losses are:-

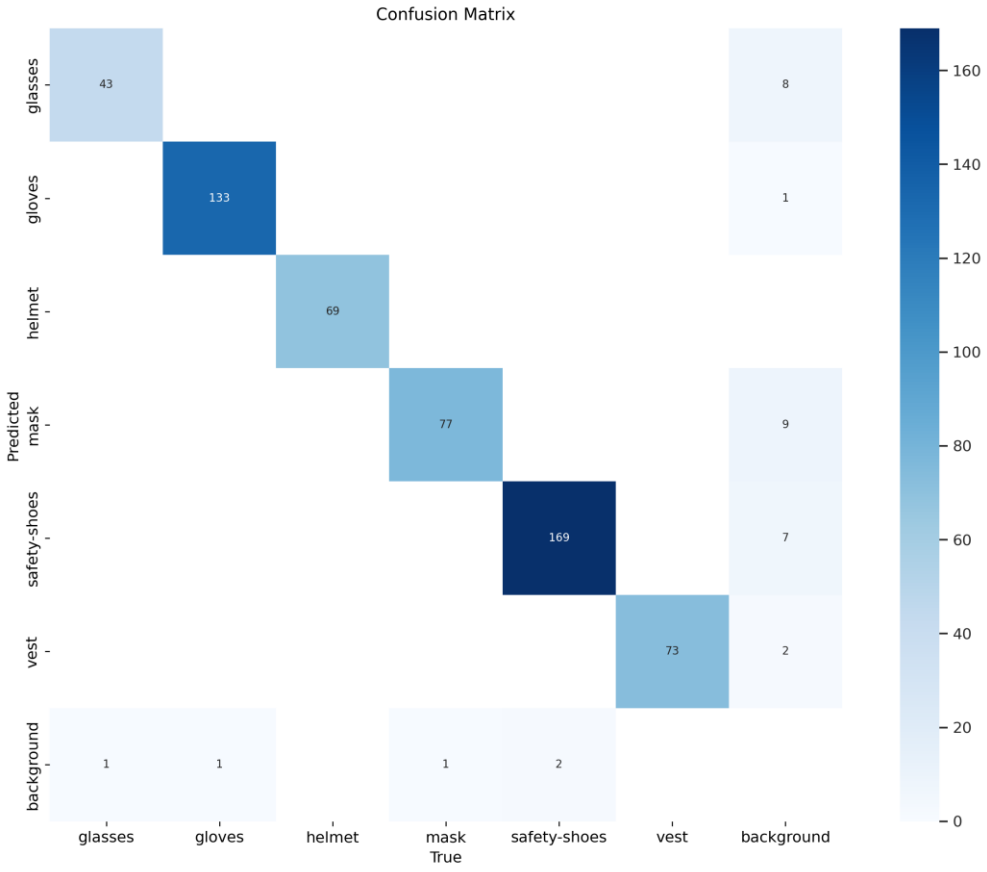
box_loss	cls_loss	dfl_loss
0.7869	0.3472	0.9505

- **GPU Memory Usage per Epoch**: Approximately 6.64 GB of GPU memory was utilized during each epoch.
- **Time Taken per Epoch**: Each epoch required approximately 1 minute and 20–25 seconds for training.
- **Total Training Time**: The entire training process took 0.732 hours, equivalent to approximately 44 minutes.

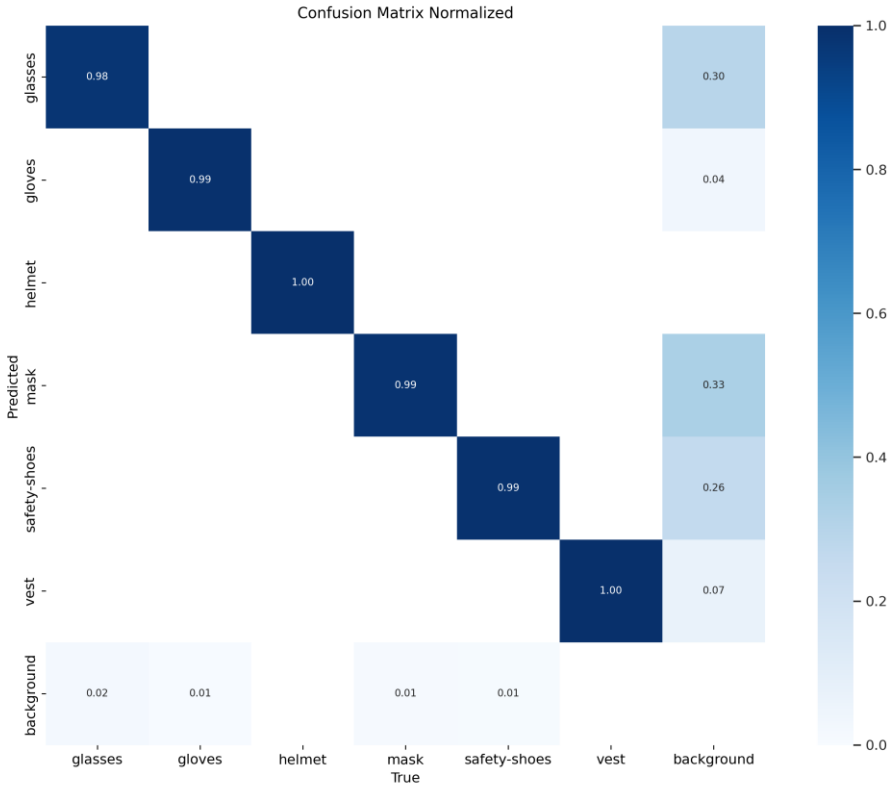
- **Testing/Inference Metrics**:

On testing our trained model on the testing dataset, we got the following results:-

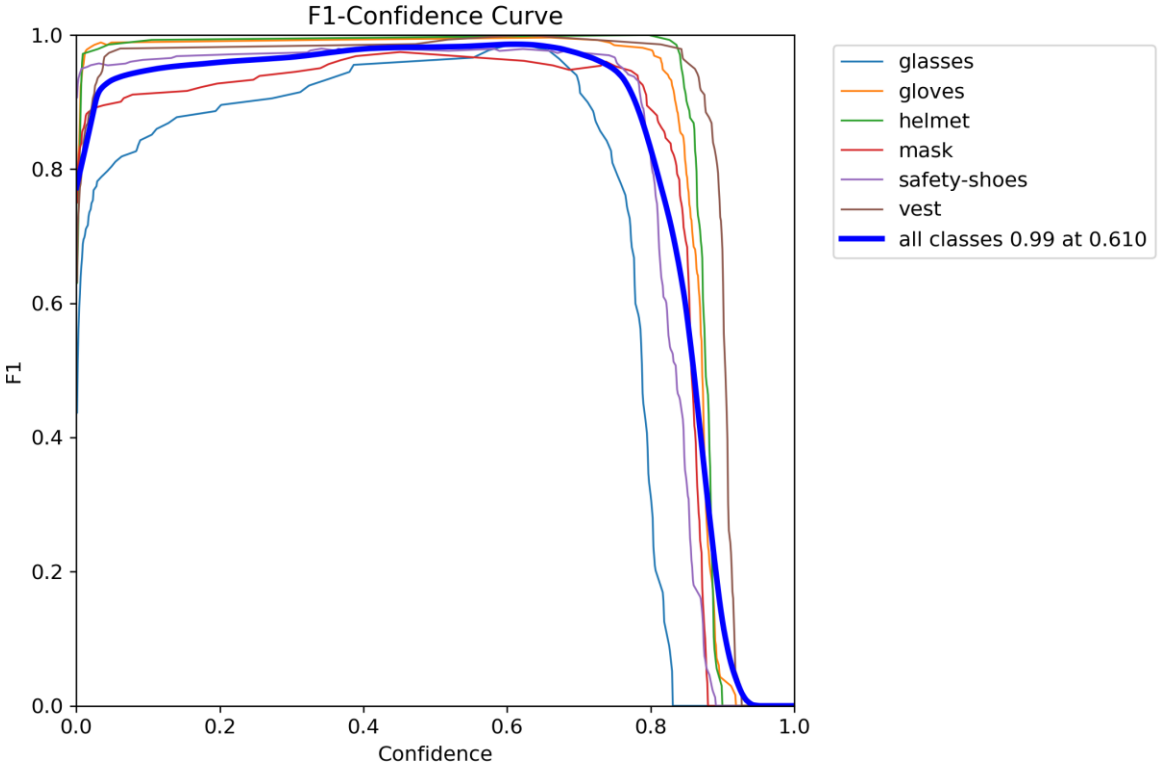
- **Confusion Matrix**:



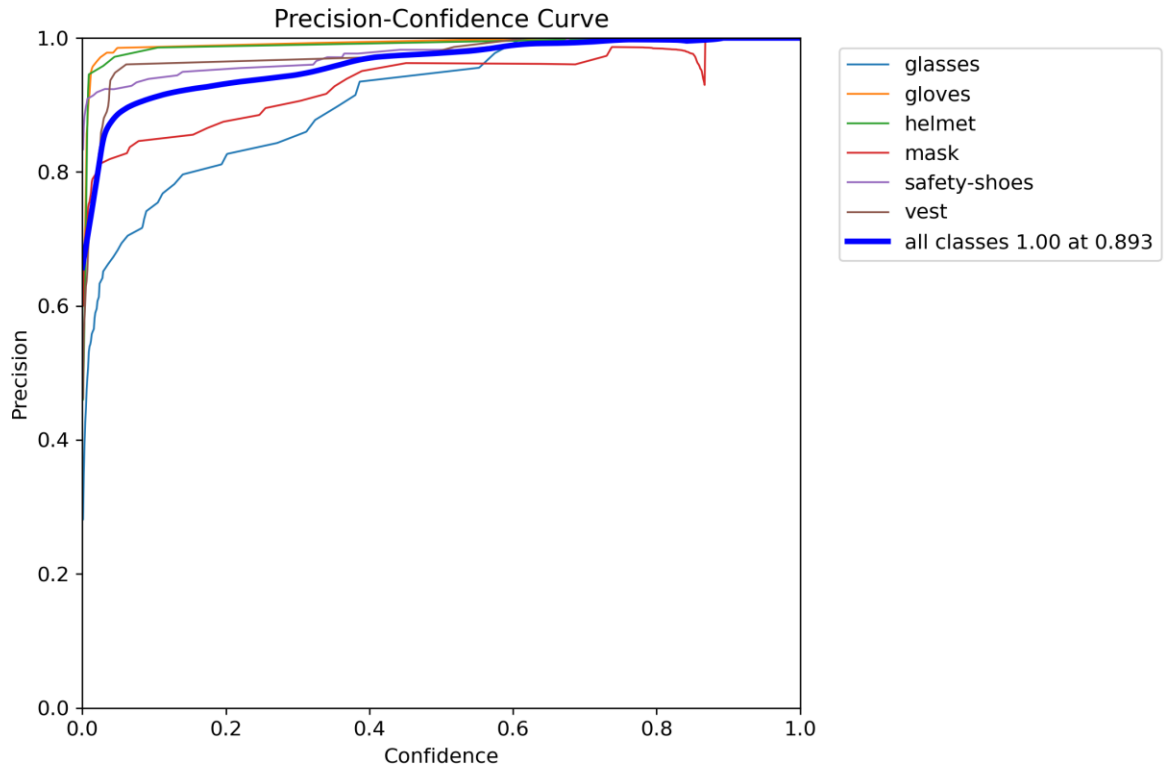
- **Normalized Confusion Matrix**:



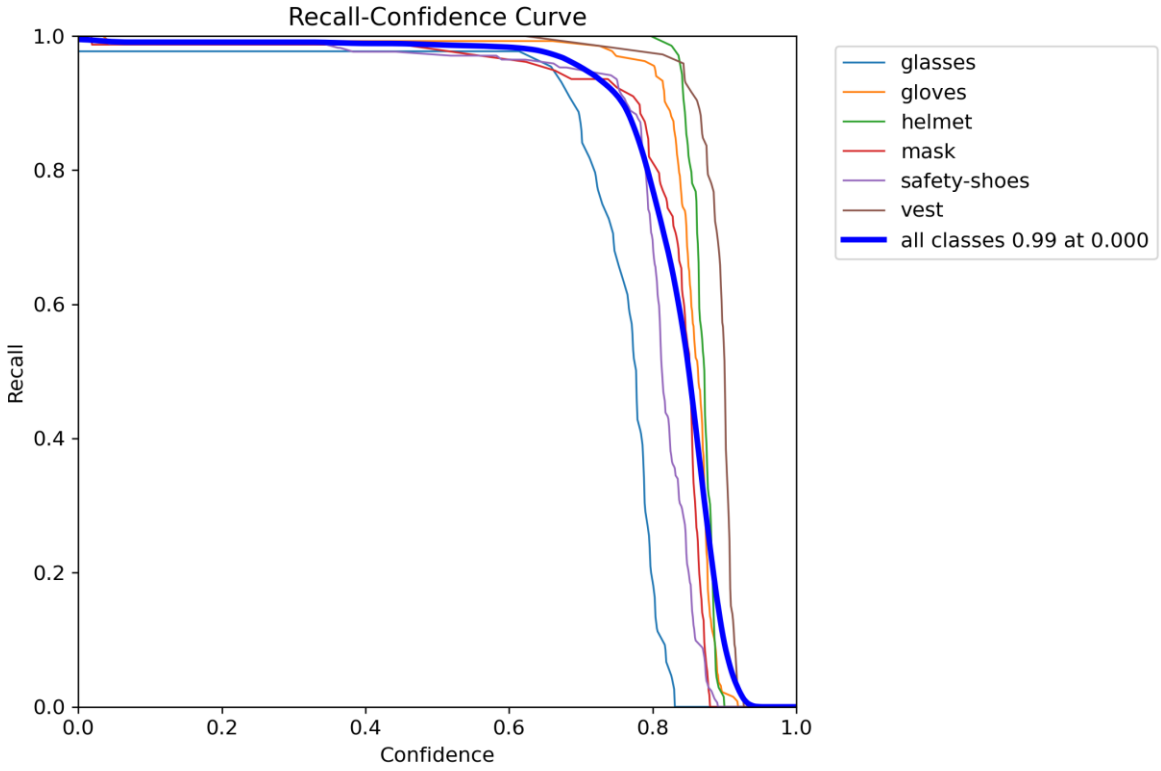
- **F1 Curve**:



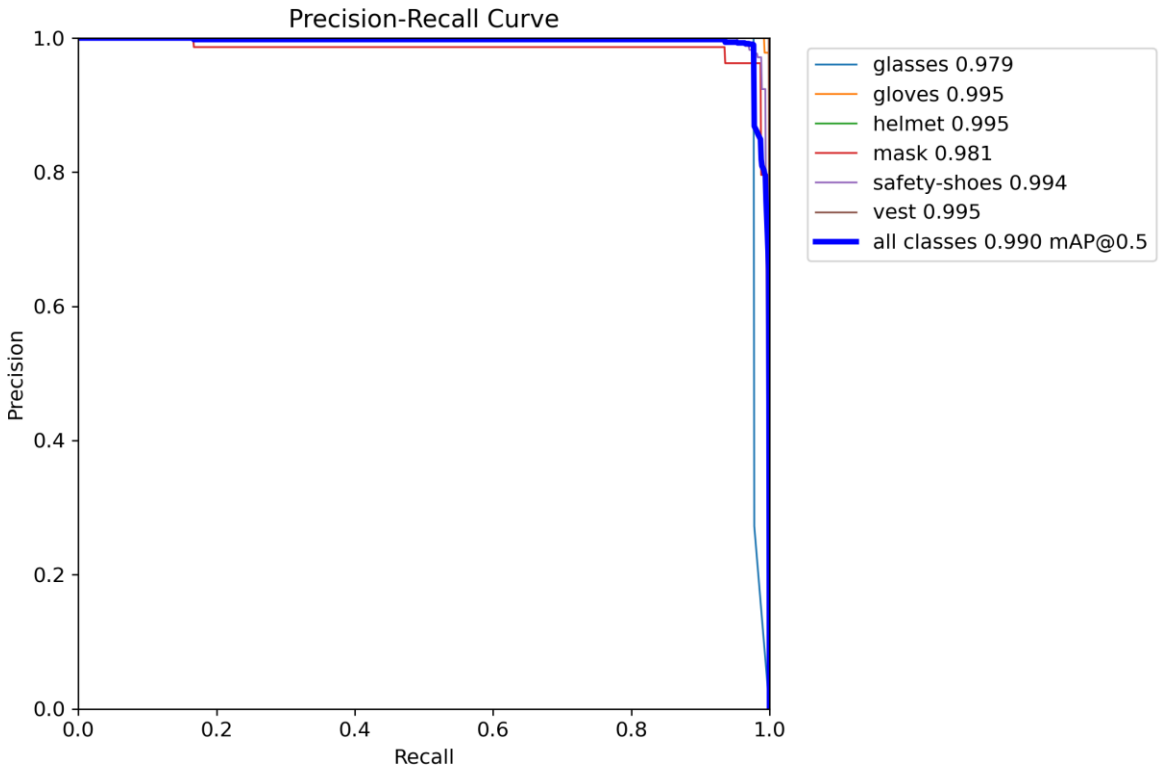
- **P Curve**:



- **R Curve**:



- **PR Curve**:



- **Class wise precisions on test dataset**:

Class	Images	Instances	Box(P	R	mAP50	mAP50-95):
all	120	569	0.99	0.983	0.99	0.748
glasses	44	44	0.996	0.977	0.979	0.601
gloves	73	134	0.999	0.993	0.995	0.737
helmet	69	69	0.996	1	0.995	0.811
mask	78	78	0.962	0.964	0.981	0.73
safety-shoes	85	171	0.991	0.965	0.994	0.735
vest	73	73	0.998	1	0.995	0.874

- **Speed of the model**:-
  - 0.5 ms preprocess per image
  - 20.4 ms inference per image
  - 4.4 ms postprocess per image

  Total Time Per Image (Latency):-

  Sum: 0.5 + 20.4 + 4.4 = 25.4 ms
  This means the model processes one image in approximately 25.4 milliseconds, translating to around 39.37 FPS (frames per second).

4. l-variant (large)

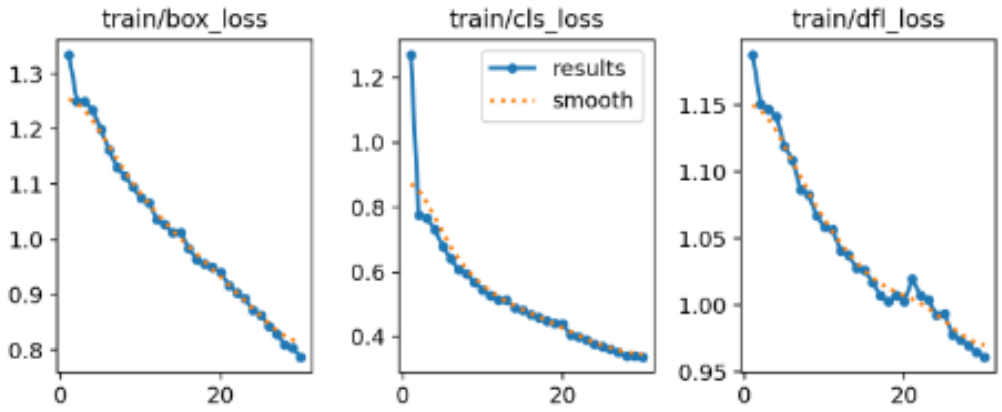
It is suited for high-accuracy tasks on powerful systems. It contains:-

- 631 layers: Number of layers are more than the medium variant.
- 25,315,106 parameters: Number of parameters are more than that of medium variant.
- 25,315,090 gradients: Number of gradients are also more than medium variant.
- 87.3 GFLOPs: It is more than the medium variant, meaning even more computational complexity medium variant.

  - Large variant can be used/loaded by the following code:-

```
model = YOLO("yolo11l.pt")
```

- **Training Metrics**:
  - **Losses during 30 epochs**



After 30 epochs, final losses are:-

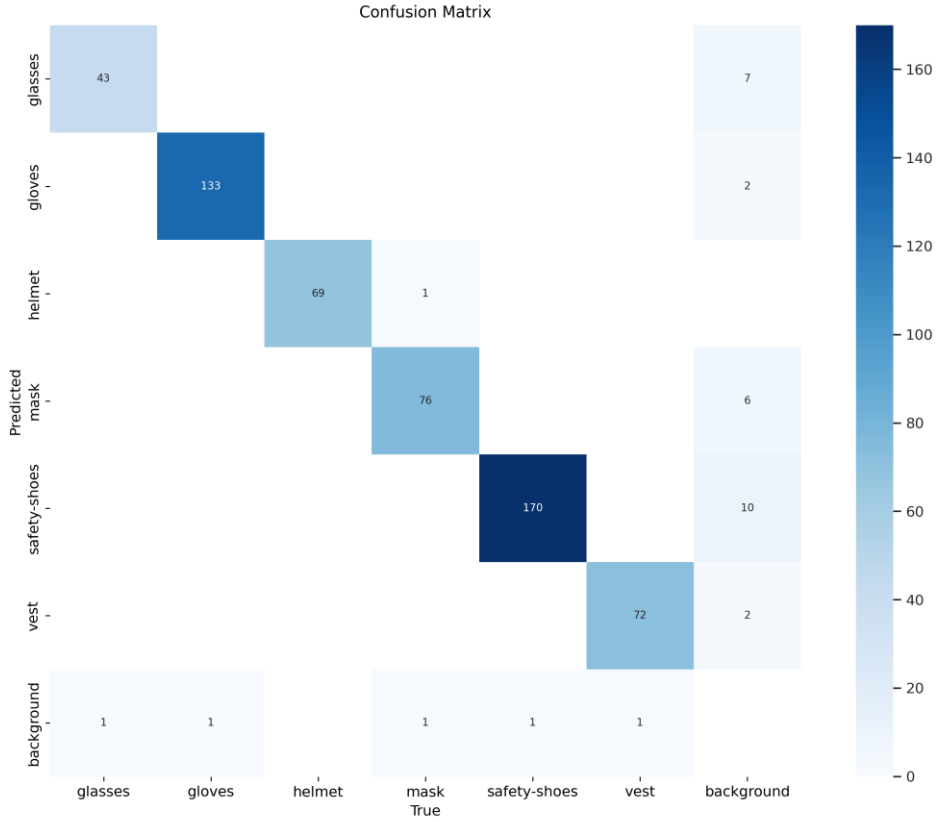
box_loss	cls_loss	dfl_loss
0.7867	0.3369	0.9608

- **GPU Memory Usage per Epoch**: Approximately, 8.38 GB of GPU memory was utilized during each epoch.
- **Time Taken per Epoch**: Each epoch required approximately 1 minute and 40 seconds for training.
- **Total Training Time**: The entire training process took 0.913 hours, equivalent to approximately 55 minutes.

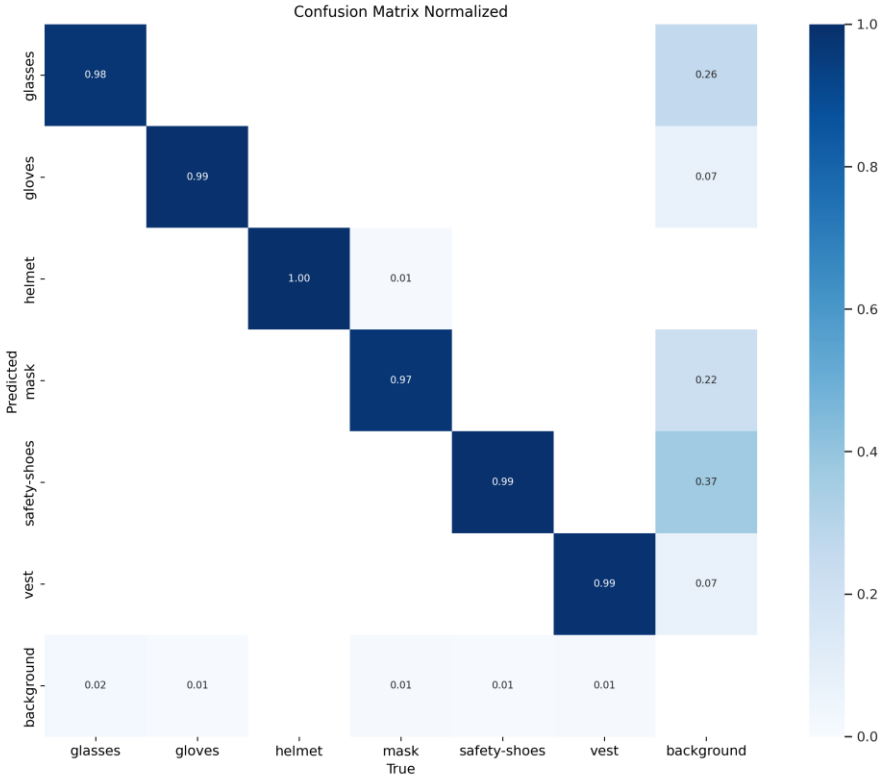
- **Testing/Inference Metrics**:

On testing our trained model on the testing dataset, we got the following results:-

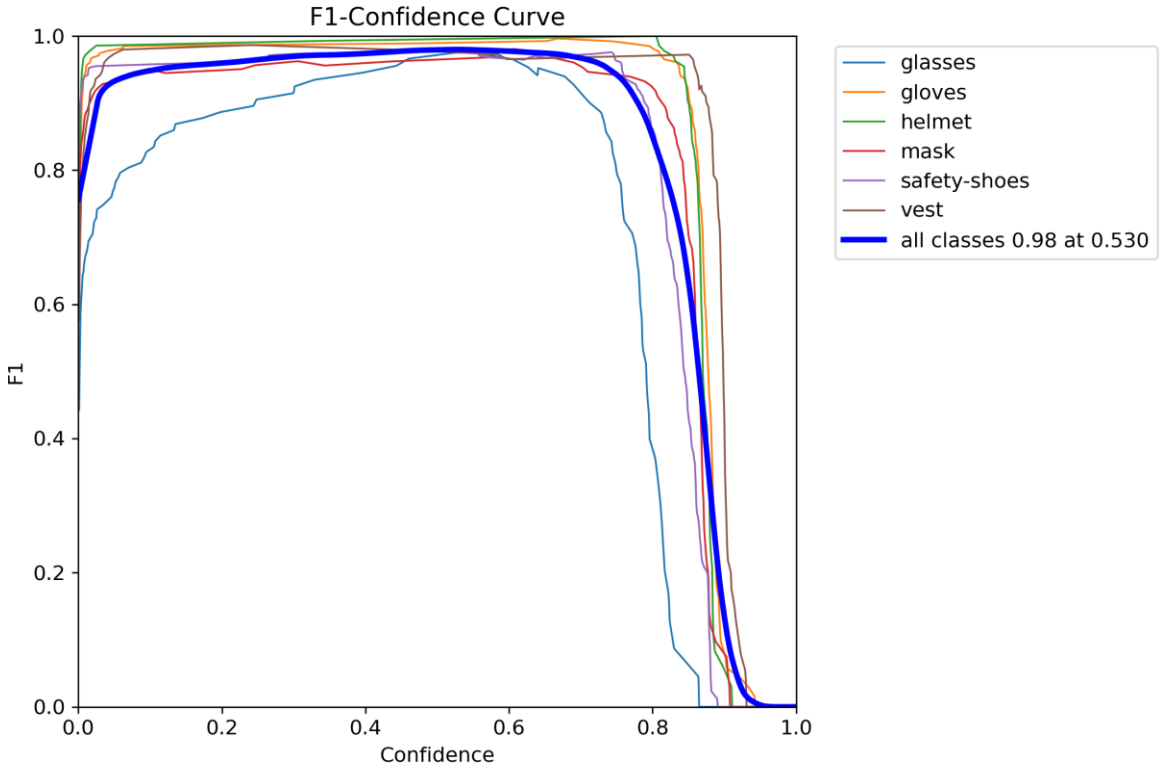
- **Confusion Matrix**:



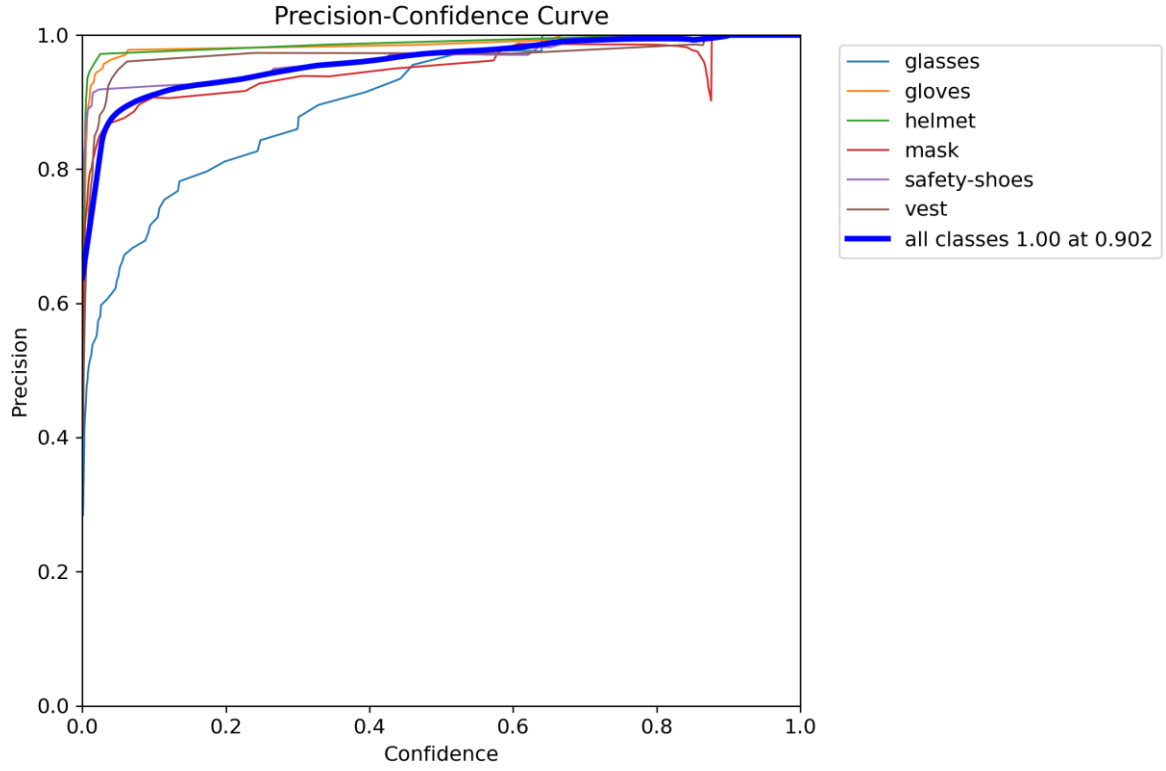
- **Normalized Confusion Matrix**:



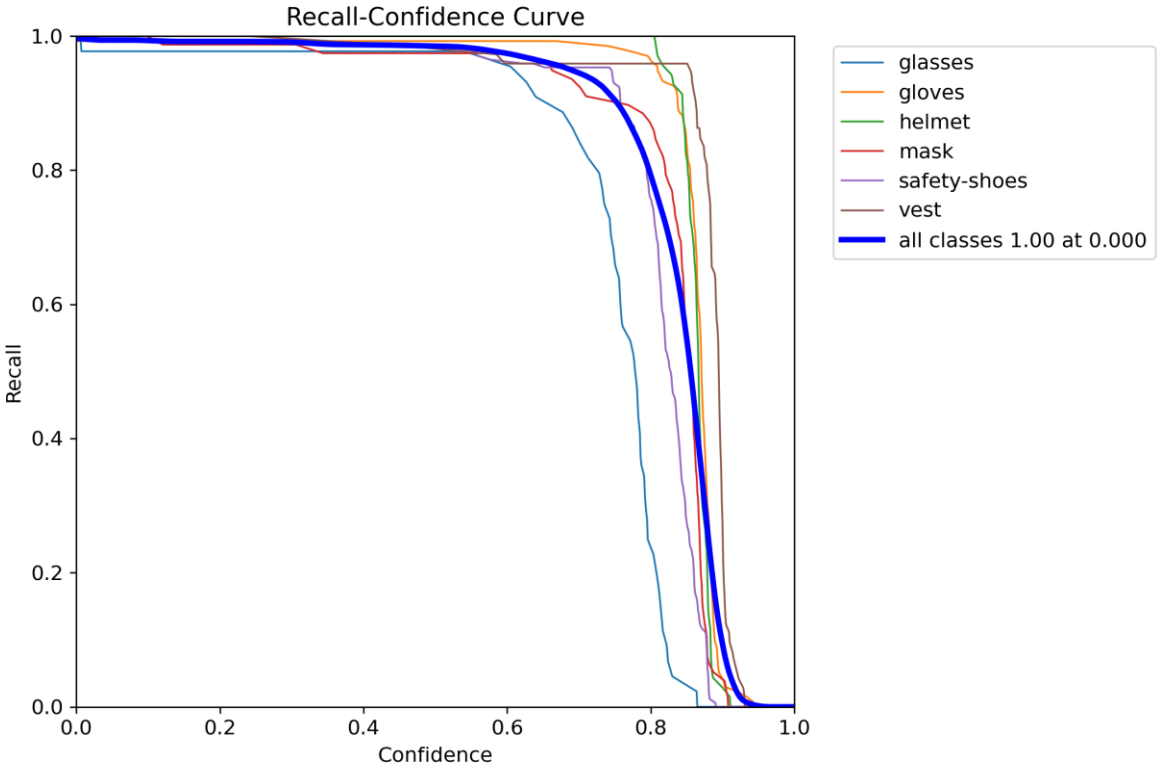
- **F1 Curve**:



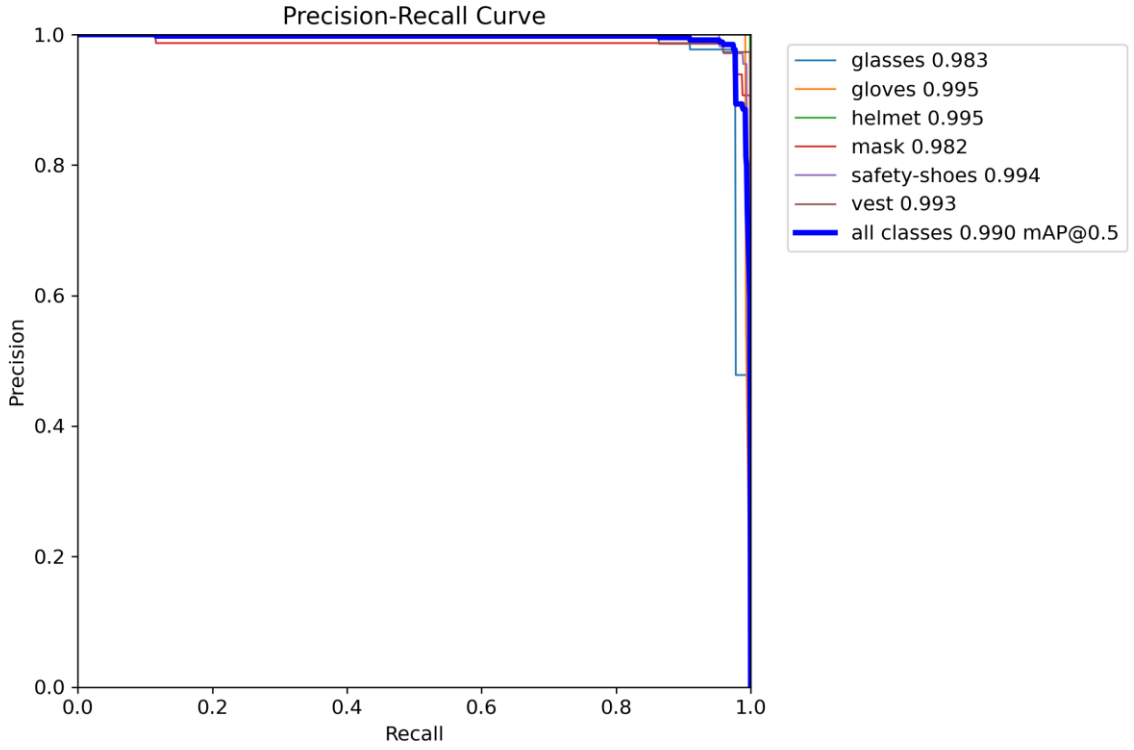
- **P Curve**:



- **R Curve**:



- **PR Curve**:



- **Class wise precisions on test dataset**:

Class	Images	Instances	Box(P	R	mAP50	mAP50-95):
all	120	569	0.975	0.985	0.99	0.745
glasses	44	44	0.97	0.977	0.983	0.607
gloves	73	134	0.987	0.993	0.995	0.747
helmet	69	69	0.991	1	0.995	0.8
mask	78	78	0.957	0.974	0.982	0.726
safety-shoes	85	171	0.971	0.984	0.994	0.73
vest	73	73	0.973	0.98	0.993	0.861

- **Speed of the model**:-
  - 0.5 ms preprocess per image
  - 27.1 ms inference per image
  - 3.7 ms postprocess per image

  Total Time Per Image (Latency):-

  Sum: 0.5 + 27.1 + 3.7 = 31.3 ms
  This means the model processes one image in approximately 31.3 milliseconds, translating to around 32 FPS (frames per second).

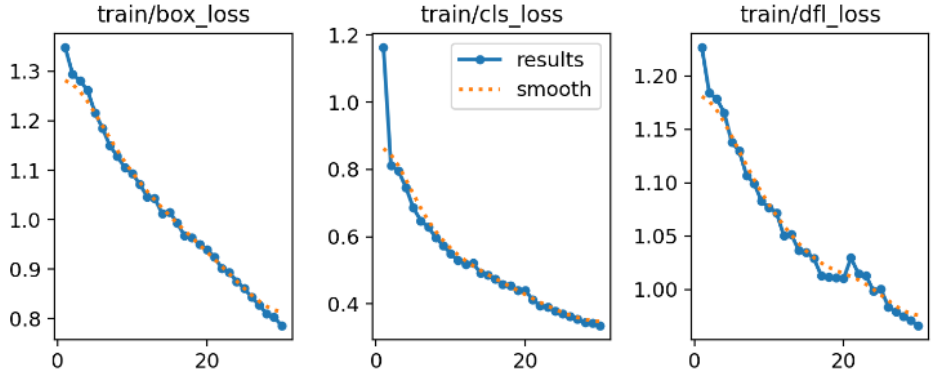
5. x-variant (extra-large)

It targets maximum accuracy, often requiring extensive computational resources. It contains:-

- 631 layers: Number of layers are same as that of large variant.
- 56,880,706 parameters: The number of parameters exceeds that of the large variant, specifically amounting to more than double the previous count.
- 56,880,690 gradients: Number of gradients are also more than the double as that of large variant.
- 195.5 GFLOPs: This is the highest among all variants, indicating the greatest computational complexity.
  - Extra-large variant can be used/loaded by the following code:-

```
model = YOLO("yolo11x.pt")
```

- **Training Metrics**:
  - **Losses during 30 epochs**



After 30 epochs, final losses are:-

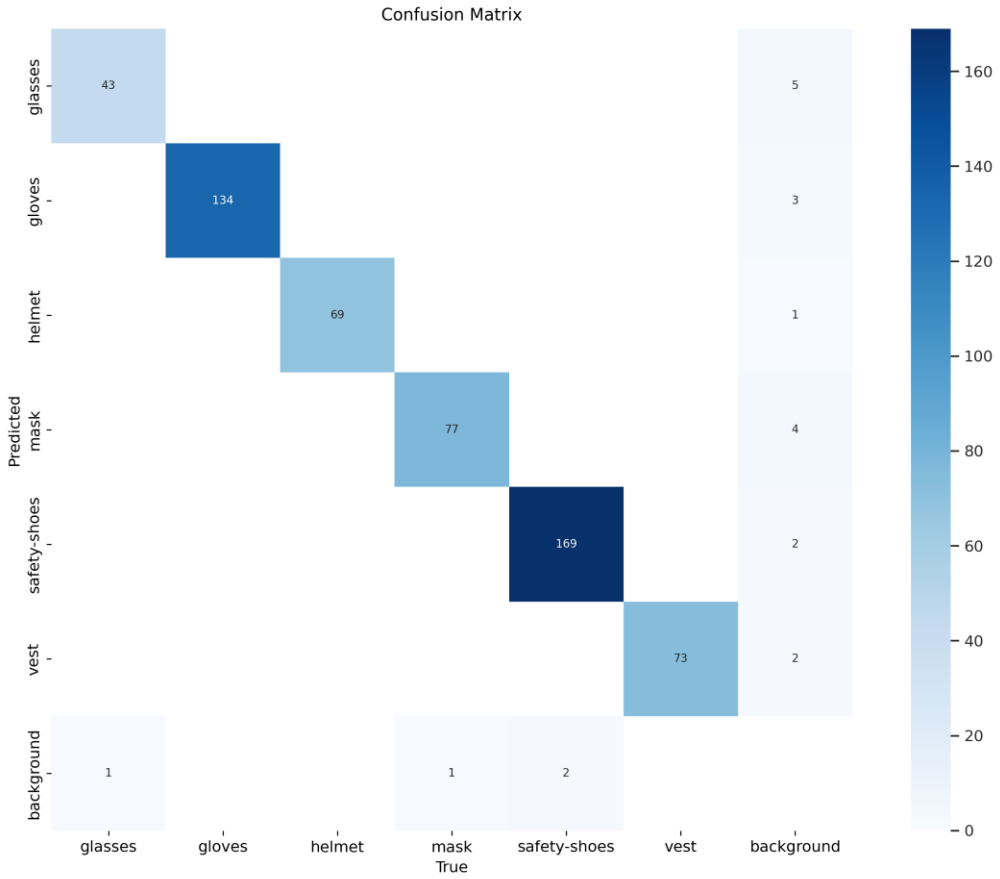
box_loss	cls_loss	dfl_loss
0.7853	0.3348	0.9659

- **GPU Memory Usage per Epoch**: Approximately, 13 GB of GPU memory was utilized during each epoch.
- **Time Taken per Epoch**: Each epoch required approximately 2 minutes and 45 seconds for training.
- **Total Training Time**: The entire training process took 1.515 hours, equivalent to approximately 1 hour and 31 minutes.

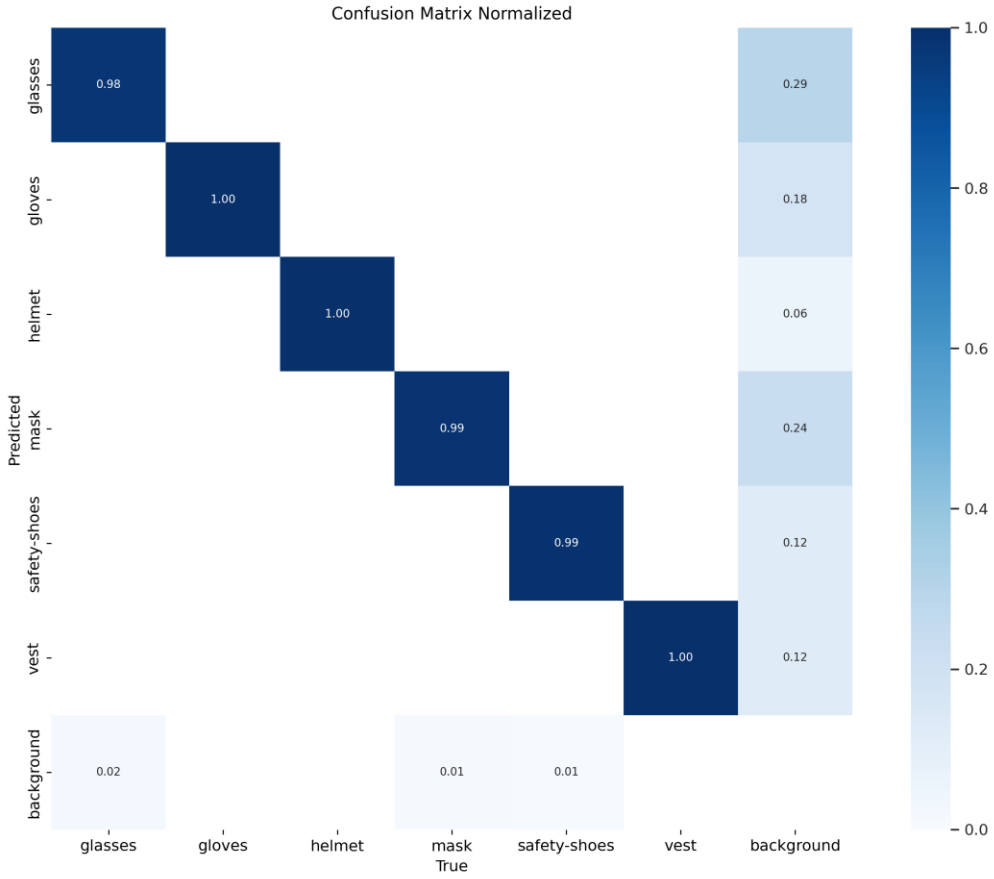
- **Testing/Inference Metrics**:

On testing our trained model on the testing dataset, we got the following results:-

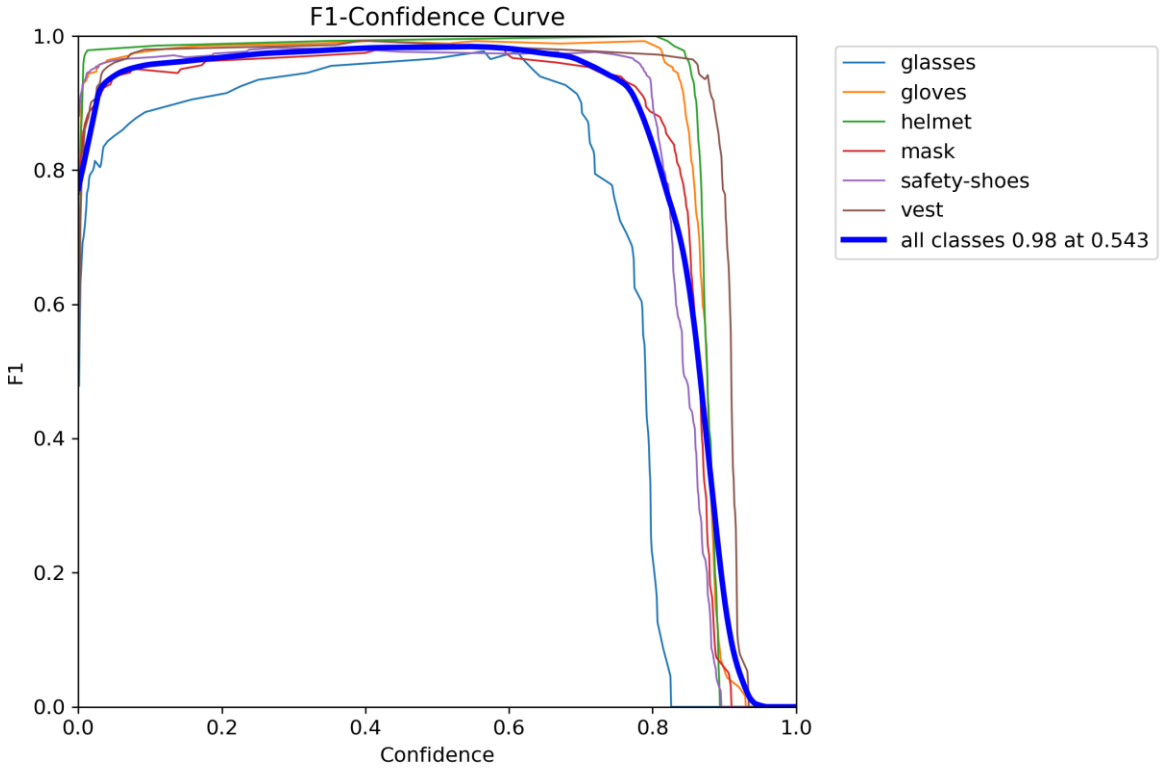
- **Confusion Matrix**:



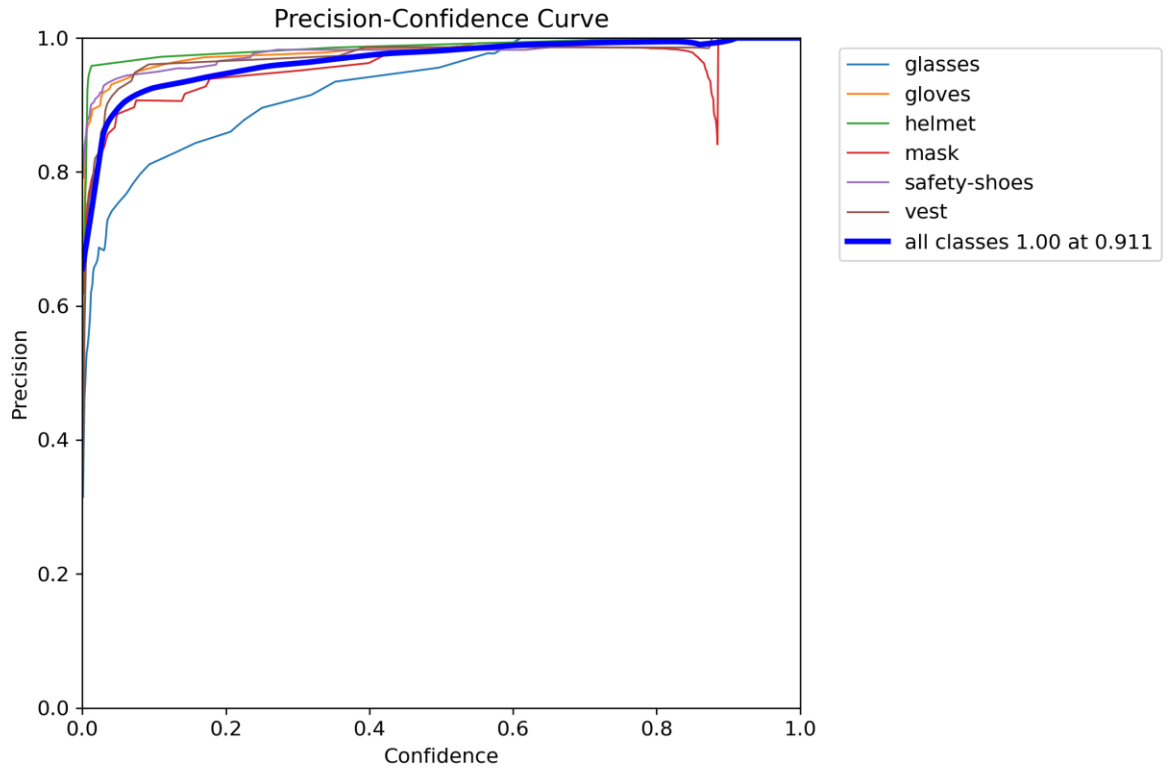
- **Normalized Confusion Matrix**:



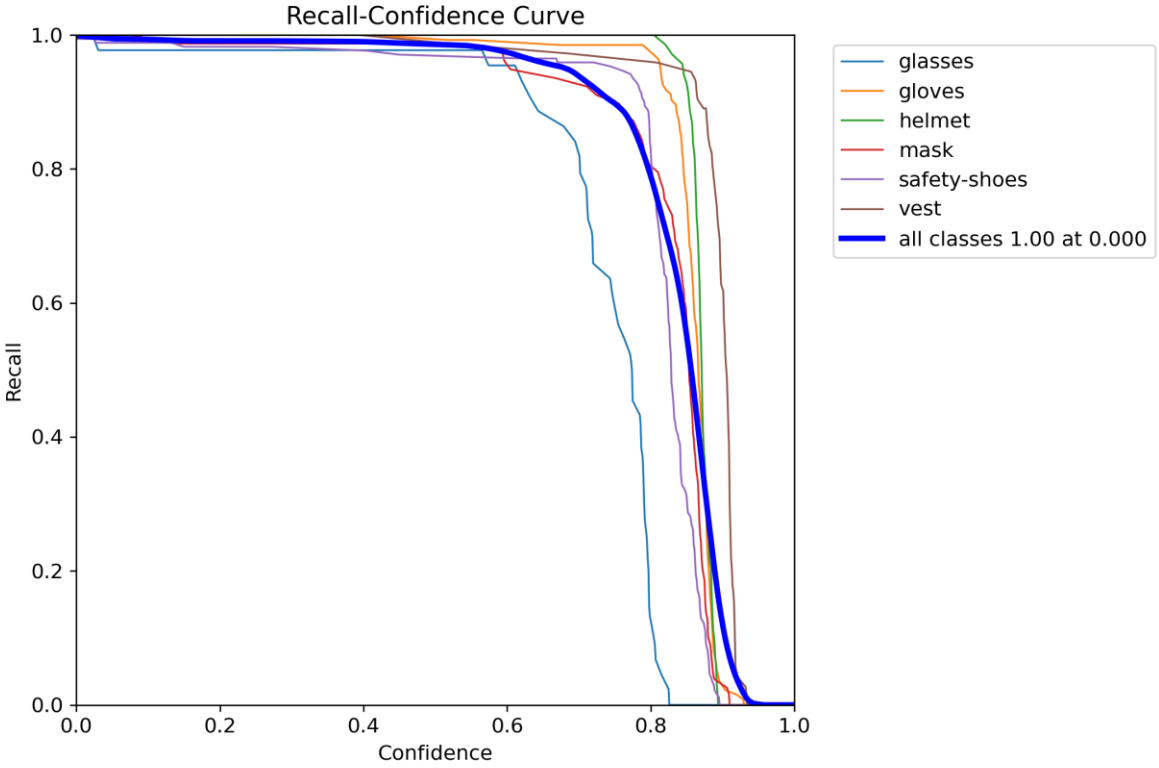
- **F1 Curve**:



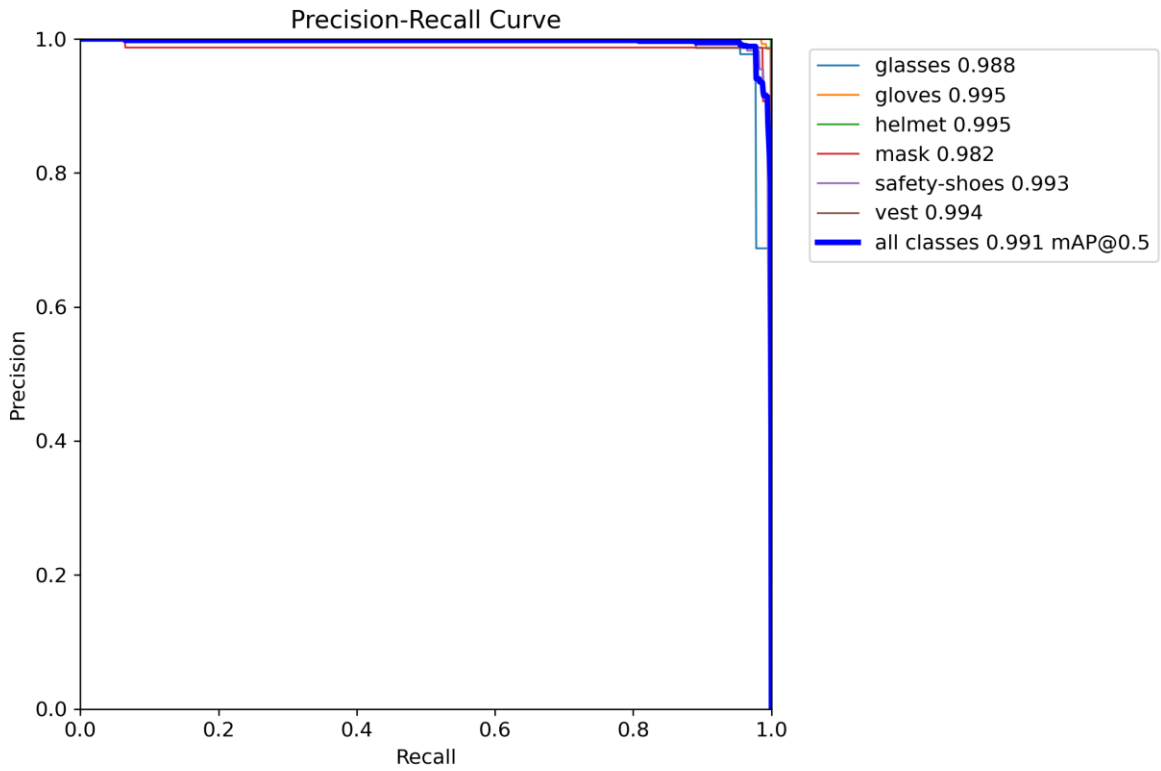
- **P Curve**:



- **R Curve**:



- **PR Curve**:



- **Class wise precisions on test dataset**:

Class	Images	Instances	Box(P	R	mAP50	mAP50-95):
all	120	569	0.983	0.985	0.991	0.746
glasses	44	44	0.963	0.977	0.988	0.617
gloves	73	134	0.986	0.993	0.995	0.748
helmet	69	69	0.991	1	0.995	0.8
mask	78	78	0.987	0.987	0.982	0.725
safety-shoes	85	171	0.982	0.968	0.993	0.726
vest	73	73	0.986	0.987	0.994	0.86

- **Speed of the model**:-
  - 0.3 ms preprocess per image
  - 48.6 ms inference per image
  - 3.0 ms postprocess per image

Total Time Per Image (Latency):-

Sum: 0.3 + 48.6 + 3.0 = 51.9 ms
This means the model processes one image in approximately 51.9 milliseconds, translating to around 19.27 FPS (frames per second).

COMPARISON OF DIFFERENT VARIANTS OF YOLO11

- ## ARCHITECTURE COMPARISON

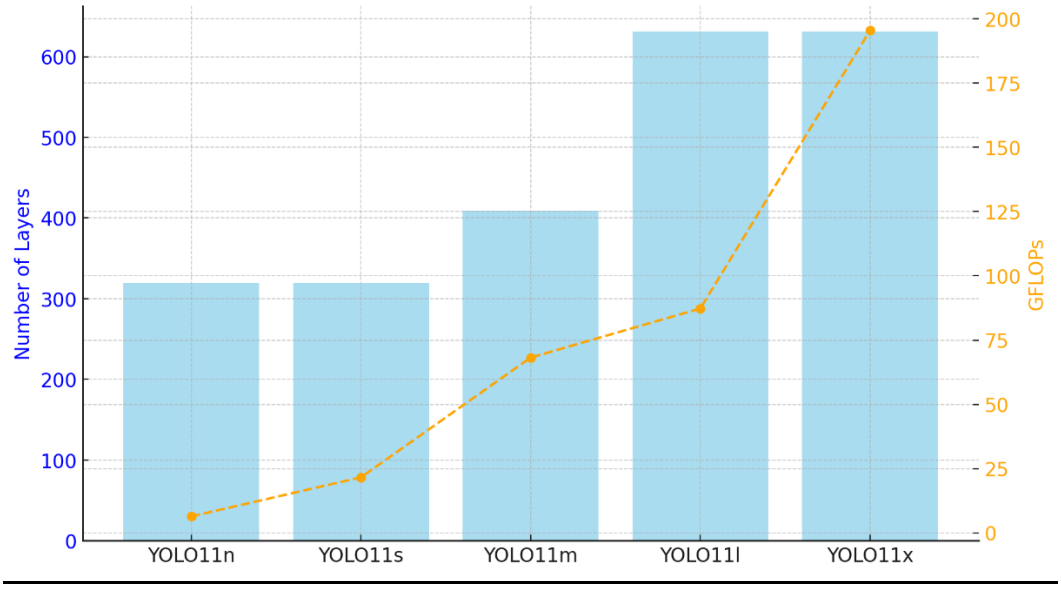
Here's the data tabulated for better clarity:

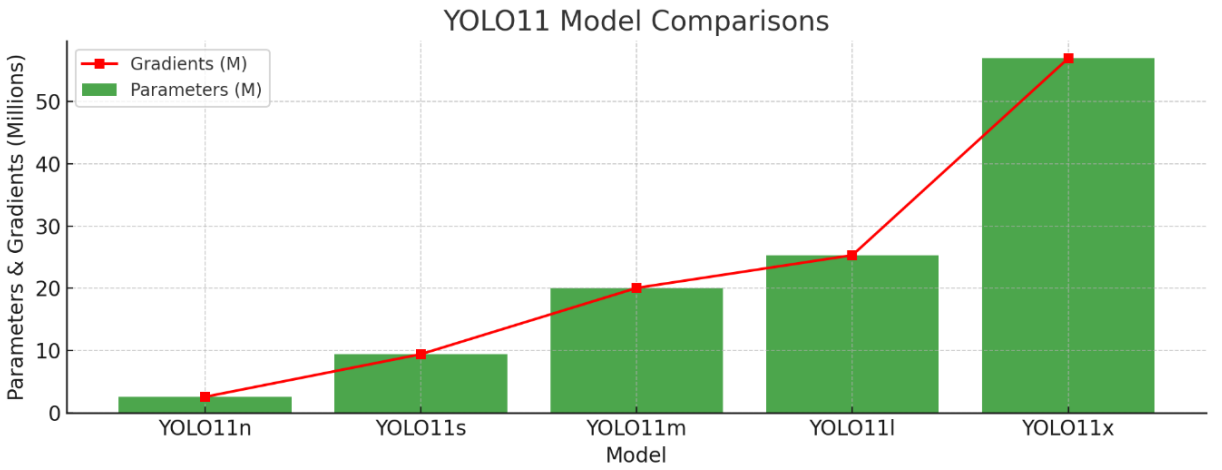
Model	Layers	Parameters (M)	Gradients (M)	GFLOPs
YOLO11n	319	2.59	2.59	6.4
YOLO11s	319	9.43	9.43	21.6
YOLO11m	409	20.06	20.06	68.2
YOLO11l	631	25.32	25.32	87.3
YOLO11x	631	56.88	56.88	195.5

Key:

- **Parameters (M):** Total parameters in millions.
- **Gradients (M):** Trainable gradients in millions.
- **GFLOPs:** Giga Floating Point Operations per second, indicating computational cost.

- ## VISUALIZATIONS





The visualizations illustrate the differences among the YOLO11 model variants:

1. **First Chart:**

- The number of layers (bars) grows with the model size, peaking with YOLO11l and YOLO11x.
- The GFLOPs (orange line) increase significantly with model size, reflecting computational demands.

2. **Second Chart:**

- Parameters (green bars) and gradients (red line) are closely matched, rising sharply as the model size increases.
- YOLO11x has the highest values, indicating its large capacity and computational complexity.

Key Observations:

- As model size increases (n → x), layers and parameters grow substantially, with a corresponding rise in GFLOPs.
- The computational demand (GFLOPs) grows faster than the number of layers, parameters, and gradients, reflecting increased model complexity.

- **TRAINING COMPARISON**

  - **LOSSES COMPARISON**

Variant	Box Loss	Cls Loss	Dfl Loss	Key Insights
Nano (n)	Starts at 1.6, reduces to 0.8725.	Starts at 2.5, declines sharply to 0.4279.	Starts at 1.2, decreases steadily to 0.933.	A lightweight model with slower convergence and higher final losses.
Small (s)	Starts at 1.4, reduces to 0.8107.	Starts at 1.6, converges quickly to 0.3697.	Starts at 1.2, shows smoother convergence, ending at 0.948.	Balances performance and efficiency. Faster convergence and lower losses than Nano.
Medium (m)	Starts at 1.3, reduces to 0.7869.	Starts at 1.2, drops sharply to 0.3472.	Starts at 1.2, stabilizes at 0.9505.	Offers strong overall performance with low final losses.
Large (l)	Starts at 1.3, reduces to 0.7867.	Starts at 1.1, drops quickly to 0.3369.	Starts at 1.15, decreases steadily to 0.9608.	Combines high accuracy with moderate computational cost.
Extra-Large (x)	Starts at 1.3, reduces to 0.7853.	Starts at 1.2, declines sharply to 0.3348.	Starts at 1.2, converges smoothly to 0.9659.	Delivers the best accuracy across all metrics.

From the above data, we can infer the following:-

1. The Box Loss decreases across all YOLO variants, with significant improvements observed when moving from Nano and Small to Medium, indicating better object localization. However, the loss stabilizes at similar levels for larger models like Large and Extra-Large, suggesting diminishing returns in localization precision as model complexity increases. This trend implies that smaller models are suitable for tasks with moderate localization requirements, while larger models excel in applications where precise bounding box placement is critical.
2. Class Loss decreases significantly with increasing model size, with the sharpest improvements seen in Medium, Large, and Extra-Large variants. This reflects the larger models' ability to learn complex decision boundaries, resulting in better object classification. For tasks requiring high classification accuracy, larger models are more suitable, whereas smaller models may suffice for simpler tasks but may struggle with complex multi-class detection scenarios due to higher final Class Loss values.
3. The increasing Distribution Focal Loss (DFL Loss) across Nano to Extra-Large YOLO variants reflects the growing complexity of their predicted probability distributions for bounding box coordinates. Larger models, with greater capacity, capture finer details and subtle variations in localization, resulting in higher DFL loss. This trend indicates that larger models prioritize precision and intricate patterns, while smaller models focus on efficiency and simplicity. The higher DFL loss in larger models is not a sign of poorer performance but rather their ability to model detailed distributions, making them suitable for tasks requiring high localization accuracy, though at the cost of increased computational demands.

○ GPU Memory Usage Comparison

The GPU memory usage for YOLOv11 variants scales significantly with model size, from nano (~1.8 GB) to x-large (~13 GB) per epoch, reflecting the growing complexity and parameter count in larger models. The memory growth is not linear; the increase from nano to small (~83%) and small to medium (~101%) is more pronounced compared to the jump from large to x-large (~55%). This trend highlights the steep resource demands of larger variants, making nano and small ideal for resource-constrained systems, while medium offers

a balanced trade-off between memory efficiency (~6.64 GB) and potential performance gains. Larger variants (large and x-large) demand significant GPU capacity, suitable only for high-performance setups. These insights suggest careful consideration of resource availability, deployment constraints, and accuracy-performance trade-offs when choosing a variant, emphasizing the need for further evaluation of inference memory usage and accuracy to guide optimal deployment decisions.

○ Training Time Comparison

Variant	Training Time Per Epoch	Total Training Time	Approximate Total Time (hh:mm:ss)
Nano (n)	45–50 seconds	0.42 hours	25 minutes 12 seconds
Small (s)	50–55 seconds	0.47 hours	28 minutes 12 seconds
Medium (m)	1 min 20–25 seconds	0.732 hours	~44 minutes
Large (l)	1 min 40 seconds	0.913 hours	~55 minutes
Extra Large (x)	2 min 45 seconds	1.515 hours	~1 hour 31 minutes

- **Key Observations:**
  - Training time scales with model size, with **Extra Large** taking **3.6x longer** than **Nano**.
  - **Nano and Small** are ideal for quick training and resource-limited environments.
  - **Medium, Large, and Extra Large** prioritize performance and are suited for tasks requiring higher accuracy.

• TESTING/INFERENCE COMPARISON

○ CONFUSION MATRIX COMPARISON

1. True Positives:
   - The Extra Large model generally achieves slightly better true positive rates (e.g., Gloves = 134, Mask = 77, Vest = 73).
   - The Nano and Small models are more prone to misclassifications in challenging classes.
2. Misclassifications:
   - The Extra Large model has the least number of misclassifications overall, with notable improvement in handling Safety Shoes and Mask.
3. General Trends:
   - As the model size increases (from Nano → Extra Large), performance improves slightly in terms of accuracy but at the cost of higher computational requirements.

○ F1 CURVE COMPARISON

Variant	F1 Score	Confidence Threshold
Nano (n)	0.98	0.476
Small (s)	0.98	0.559
Medium (m)	0.99	0.610
Large (l)	0.98	0.530
Extra Large (x)	0.98	0.543

Key Trends:

1. **F1 Score Saturation**: Performance plateaus at an F1 score of 0.98 for most variants, with the medium variant achieving the highest score of 0.99.
2. **Confidence Thresholds**: Larger models require higher confidence thresholds for peak performance, while the nano variant performs well at lower thresholds (0.476).
3. **Efficiency vs. Performance**: The nano variant prioritizes speed and efficiency, whereas the medium variant balances performance and confidence.
4. **Diminishing Returns**: Beyond the medium variant, increasing model size offers negligible performance improvement, indicating diminishing returns with added complexity.

○ PRECISION-CONFIDENCE CURVE COMPARISON

Variant	Precision	Confidence Threshold
Nano	1.00	0.949
Small	1.00	0.902
Medium	1.00	0.893
Large	1.00	0.902
Extra Large	1.00	0.911

Key Trends

1. **Confidence Thresholds Impact Detection:**

- **Higher thresholds** (e.g., Nano at 0.949) are more selective, detecting fewer objects but ensuring only high-confidence predictions.
- **Lower thresholds** (e.g., Medium at 0.893) detect more objects by including lower-confidence predictions while still maintaining accuracy.

2. **Variant-Specific Behavior:**

- **Nano**: Most conservative and selective.
- **Medium**: Most inclusive, detecting the highest number of objects.
- **Small & Large**: Similar performance, offering a balance of inclusivity and precision.
- **Extra Large**: Balances confidence and recall slightly better than Small and Large.

○ CLASS WISE PRECISION COMPARISON

These comparisons are at IoU=0.7 :-

Class	Nano	Small	Medium	Large	Extra Large
All	0.985	0.977	0.990	0.975	0.983
Glasses	0.963	0.934	0.996	0.970	0.963
Gloves	1.000	1.000	0.999	0.987	0.986
Helmet	0.991	0.994	0.996	0.991	0.991
Mask	0.986	0.987	0.962	0.957	0.987
Safety Shoes	0.977	0.968	0.991	0.971	0.982
Vest	0.993	0.977	0.998	0.973	0.986

Key Trends:

1. **Medium variant consistently outperforms other variants** in terms of overall precision, making it the best choice for general-purpose detection.
2. **Nano and small variants perform exceptionally well for gloves detection**, achieving perfect or near-perfect precision (1.000).
3. **Small variant shows a notable drop in precision for glasses detection**, indicating difficulty in detecting smaller or less prominent objects.
4. **Extra-large variant offers stable and balanced precision across all classes**, slightly behind the medium variant but more reliable than the large variant.
5. **Large variant shows slightly lower precision for certain classes**, particularly for mask and vest, despite being a more complex model.
6. **Helmets and gloves are detected consistently well by all variants**, indicating these are easier classes for the models to handle.
7. **Safety shoes detection is more precise with medium and extra-large variants**, suggesting these models are better suited for detecting complex or cluttered objects.

○ CLASS WISE RECALL COMPARISON

These comparisons are at IoU=0.7 :-

Class	Nano	Small	Medium	Large	Extra-Large
All	0.985	0.980	0.983	0.985	0.985
Glasses	0.977	0.977	0.977	0.977	0.977
Gloves	0.980	0.989	0.993	0.993	0.993
Helmet	1.000	1.000	1.000	1.000	1.000
Mask	0.974	0.962	0.964	0.974	0.987
Safety-Shoes	0.980	0.977	0.965	0.984	0.968
Vest	1.000	0.973	1.000	0.980	0.987

Key Trends:

1. **Consistent Recall for Helmet and Glasses**: Recall for *helmet* remains perfect (1.0) across all variants, while *glasses* recall stays constant at **0.977**, indicating no impact of model size on their detection.
2. **Improvement in Gloves and Mask with Larger Models**: Recall for *gloves* and *mask* increases with larger model variants, peaking at 0.993 and 0.987, respectively, in the extra-large variant.
3. **Mixed Performance for Safety-Shoes**: Recall for *safety-shoes* varies across models, with the large variant achieving the highest recall (0.984).
4. **Nano and Medium Variants Excel in Vest Detection**: Both nano and medium variants achieve perfect recall (1.0) for *vest*, indicating that smaller models can be highly effective.
5. **General Trend of Increasing Recall with Larger Models**: Larger models tend to provide higher recall overall, especially for complex objects like *gloves* and *mask*, but smaller models remain competitive for simpler classes.

○ CLASS WISE mAP50 COMPARISON

Class	Nano	Small	Medium	Large	Extra Large
All	0.988	0.990	0.990	0.990	0.991
Glasses	0.971	0.983	0.979	0.983	0.988
Gloves	0.995	0.995	0.995	0.995	0.995
Helmet	0.995	0.995	0.995	0.995	0.995
Mask	0.980	0.985	0.981	0.982	0.982
Safety-shoes	0.989	0.992	0.994	0.994	0.993
Vest	0.995	0.992	0.995	0.993	0.994

Key Trends:

1. **Overall Improvement:**
   mAP50 improves slightly across model sizes, from Nano (0.988) to Extra Large (0.991), reflecting better overall performance with larger models.
2. **Consistent Class-wise Performance:**
   Gloves and *Helmet* maintain a perfect mAP50 (0.995) across all model variants, indicating strong and consistent detection for these classes.
3. **Improved Detection of Glasses:**
   The mAP50 for Glasses significantly improves, starting from 0.971 (Nano) and reaching 0.988 (Extra Large), highlighting better performance for challenging objects in larger models.
4. **Minimal Gains for Mask Detection:**
   Detection of *Mask* shows only slight improvement, with mAP50 increasing marginally from 0.980 (Nano) to 0.982 (Large and Extra Large).
5. **Peak Performance in Safety-shoes:**
   Safety-shoes achieve the highest mAP50 (0.994) in Medium and Large models, slightly outperforming the Nano model (0.989).
6. **Fluctuations in Vest Detection:**
   The *Vest* class exhibits variability, with Nano reaching the highest mAP50 (0.995), a dip in the Small model (0.992), and recovery in larger models (0.994 for Extra Large).

○ CLASS WISE mAP50-95 COMPARISON

Class	Nano	Small	Medium	Large	Extra Large
All	0.716	0.742	0.748	0.745	0.746
Glasses	0.55	0.589	0.601	0.607	0.617
Gloves	0.718	0.746	0.737	0.747	0.748
Helmet	0.778	0.803	0.811	0.8	0.8
Mask	0.694	0.73	0.73	0.726	0.725
Safety-Shoes	0.706	0.715	0.735	0.73	0.726
Vest	0.847	0.868	0.874	0.861	0.86

Key Trends:

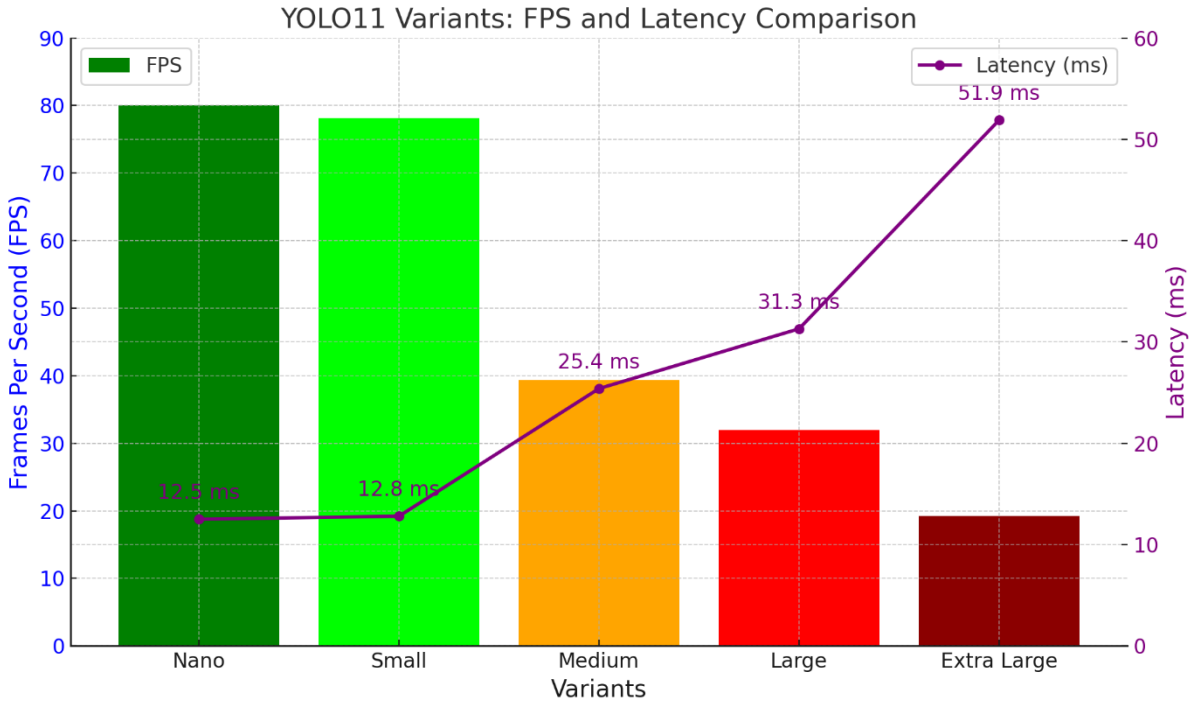
1. **Model Size vs. Performance**:
   - As model size increases (Nano → Extra Large), overall performance improves, with the **Medium variant** achieving the best balance (mAP50-95: 0.748). Larger models show diminishing returns for most classes.
2. **Class Difficulty**:
   - **Glasses** are the hardest to detect, with the lowest scores across all variants.
   - **Vest** is the easiest, consistently achieving the highest mAP50-95.
3. **Performance Trends by Class**:
   - **Glasses**: Steady improvement with larger models, peaking at **Extra Large (0.617)**.
   - **Helmet**: Best performance with the **Medium variant (0.811)**.
   - **Other Classes** (Gloves, Mask, Safety-Shoes, Vest): Moderate gains across variants, with most stabilizing at the **Medium** or **Large size**.
4. **Optimal Variant**:

- The **Medium variant** offers the best trade-off between size and accuracy, performing the highest across most classes.

○ SPEED COMPARISON

Variant	Preprocess (ms)	Inference (ms)	Postprocess (ms)	Total Latency (ms)	FPS
Nano	0.7	7.1	4.7	12.5	**80**
Small	0.4	9.4	3.0	12.8	**78.13**
Medium	0.5	20.4	4.4	25.4	**39.37**
Large	0.5	27.1	3.7	31.3	**32**
Extra Large	0.3	48.6	3.0	51.9	**19.27**

VISUALIZATION



Key Observations:

1. **Nano** is the fastest, achieving 80 FPS, making it suitable for applications requiring high-speed processing.
2. **Small** is close in performance to Nano, with slightly lower FPS but still very efficient.
3. **Medium** sees a noticeable increase in latency, halving the FPS compared to Nano, but provides better accuracy and performance for more complex tasks.

4. **Large** continues the trend of increased latency and lower FPS, suitable for more demanding workloads.
5. **Extra Large** has the highest latency, processing fewer frames per second (19.27 FPS), but is likely optimized for scenarios requiring the highest accuracy and model capacity.

In general, the choice of variant depends on the balance between processing speed (FPS) and the desired model performance or accuracy.

• CONCLUSION

1. **Nano and Small**:
    - Designed for ultra-lightweight deployments, these variants excel in scenarios requiring high-speed processing and minimal computational resources.
    - Ideal for real-time applications such as video streaming, robotics, and embedded systems where quick decision-making is crucial.
    - Their compact size makes them particularly well-suited for edge devices with limited hardware capabilities.
2. **Medium**:
    - Strikes an optimal balance between speed and accuracy, making it a versatile choice for a wide range of applications.
    - Suitable for moderately complex tasks, such as object detection in moderately dynamic environments or systems that need a balance of real-time performance and reliable precision.
    - Commonly used in industrial automation, traffic monitoring, and augmented reality systems.
3. **Large and Extra Large**:
    - Designed for tasks where achieving high precision is a priority, even if it comes at the cost of slower processing times.
    - Best suited for offline or post-processing scenarios, such as detailed image analysis, medical imaging, and high-resolution video analytics.
    - Leveraging higher computational resources, these variants provide superior accuracy for tasks requiring meticulous attention to detail.