INTELLIGENT TRAVEL PLATFORM

HORUS EYE

PROBLEM STATEMENT

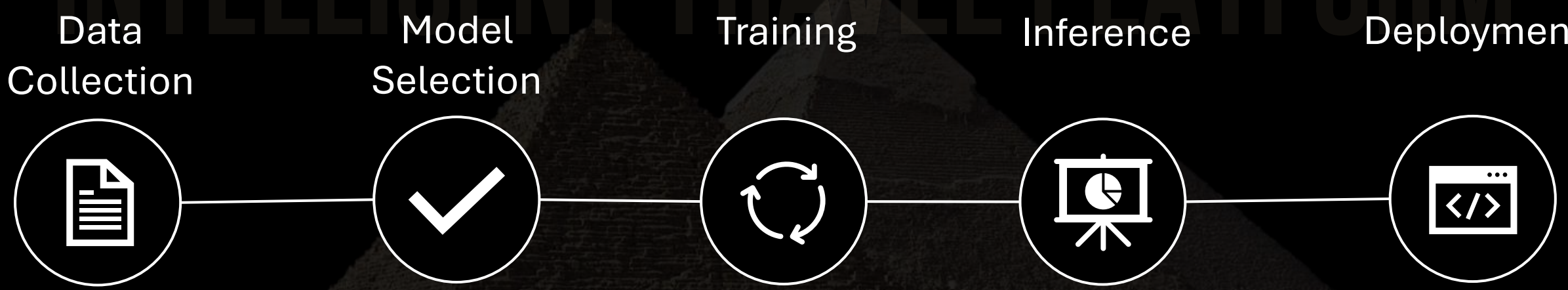
Traditional guidebooks and online resources often fail to provide the real-time assistance and personalized insights necessary for tourists to fully appreciate and explore the wonders of Egypt. This lack of effective guidance and support presents several key challenges:

- **Inefficient Landmark Identification:** Tourists without prior knowledge or proper guidance may struggle to identify significant landmarks. This can result in missed opportunities in exploration of Egypt's rich cultural heritage.
- **Fragmented Booking Experience**: Booking accommodations in Egypt can be a disjointed process, requiring travelers to navigate multiple websites and platforms. This fragmentation often leads to confusion, and suboptimal choices for accommodations.
- **Language Barrier:** As the trend of self-guided tourism grows, the language barrier becomes a significant challenge. Tourists often struggle with communication, which can lead to difficulties in understanding local customs, navigating through cities, and accessing services. This barrier results in missed opportunities, wasted time, and diminished travel experience.

PROBLEM STATEMENT

to address these challenges, our project takes the advantage of artificial intelligence power providing a modern, smart solution for the entire tourism experience in Egypt. By using AI, we offer efficient landmark identification, a seamless booking process, overcoming language barriers, ensuring that tourists do not miss any opportunities and can fully engage with and enjoy their travel experiences.

PROJECT WORKFLOW



DATA COLLECTION

Data is the keyword of the entire process although it was the grand challenge we faced. Our data is gathered from Kaggle Ancient Egyptian Landmarks dataset.

The main Challenge was associated with the data is being raw data:

Unlabeled data (Lack of annotations)	Lack of preprocessing	Class Imbalance

UNLABELED DATA (LACK OF ANNOTATIONS)

- Annotating data involves adding labels or metadata to raw data, enabling machines to understand and process the information effectively. In our project level context, the dataset contains raw data without any metadata.
- **Annotation Challenges:**
  - **Large Dataset Size as the data annotations were handled manually:**

    Annotating the data was the major challenge as it's a time-consuming process to annotate more than 3000 examples.
  - **Suitable Annotation Tool:** Identifying a user-friendly annotation tool capable of exporting data in the required YOLOv7 PyTorch format, preferably be available free of cost posed another challenge.

HOW DO WE OVERCOME THE CHALLENGES

Selecting suitable annotating tool:

- After extensive research, we identified the Roboflow online annotation tool as the most suitable option for our needs. Its user-friendly interface, compatibility with YOLOv7 PyTorch format, in addition to the free plan for users offers wide range of exporting formats.

The data were preprocesses using roboflow:

- Resized to 640x640.
- Greyscaled, Grayscale conversion reduces the image data to a single channel, this helps in scenarios of the colors is not important in object detection, improving performance of the model.

Dealing with class imbalance using Augmentation

MODEL SELECTION

A selection criteria was followed to select the best algorithm for object detection, the following factors are critical in guiding the selection process:

- Accuracy
- Speed
- Computational and hardware requirements
- Complexity and model size
- Development and Community support.

MODEL SELECTION

Based on these factors several models were used to select the best option for object detection task. Each model was assigned to each member of machine learning team.

Object detection models:

- SSD
- R-CNNs (Fast & Mask RCNN)
- YOLO (v5 & v7)

SDD (SINGLE SHOT MULTIBOX DETECTOR)

Overview:

- **Architecture**: A Single Shot Detector (SSD) is an innovative object detection algorithm in computer vision. It stands out for its ability to swiftly and accurately detect and locate objects within images or video frames. What sets SSD apart is its capacity to accomplish this in a single pass of a deep neural network, making it exceptionally efficient and ideal for real-time applications.

Key Features:

- **Single shot**: SSD performs object detection in a single pass through the network, making it faster and more efficient compared to two-stage models.

SDD (SINGLE SHOT MULTIBOX DETECTOR)

- **MultiBox**: SSD uses a set of default bounding boxes of various scales and aspect ratios at multiple locations in the input image, predicting adjustments to these boxes to accurately locate objects.
- **Multi-Scale Detection**: SSD operates on multiple feature maps with different resolutions, allowing it to detect objects of various sizes by making predictions at different scales to capture varying levels of granularity.

Challenges and Limitations of SSD

- Small Object Detection: SSD struggles with detecting tiny objects, as anchor boxes may not effectively represent their size and shape within feature pyramids, leading to accuracy challenges.

SDD (SINGLE SHOT MULTIBOX DETECTOR)

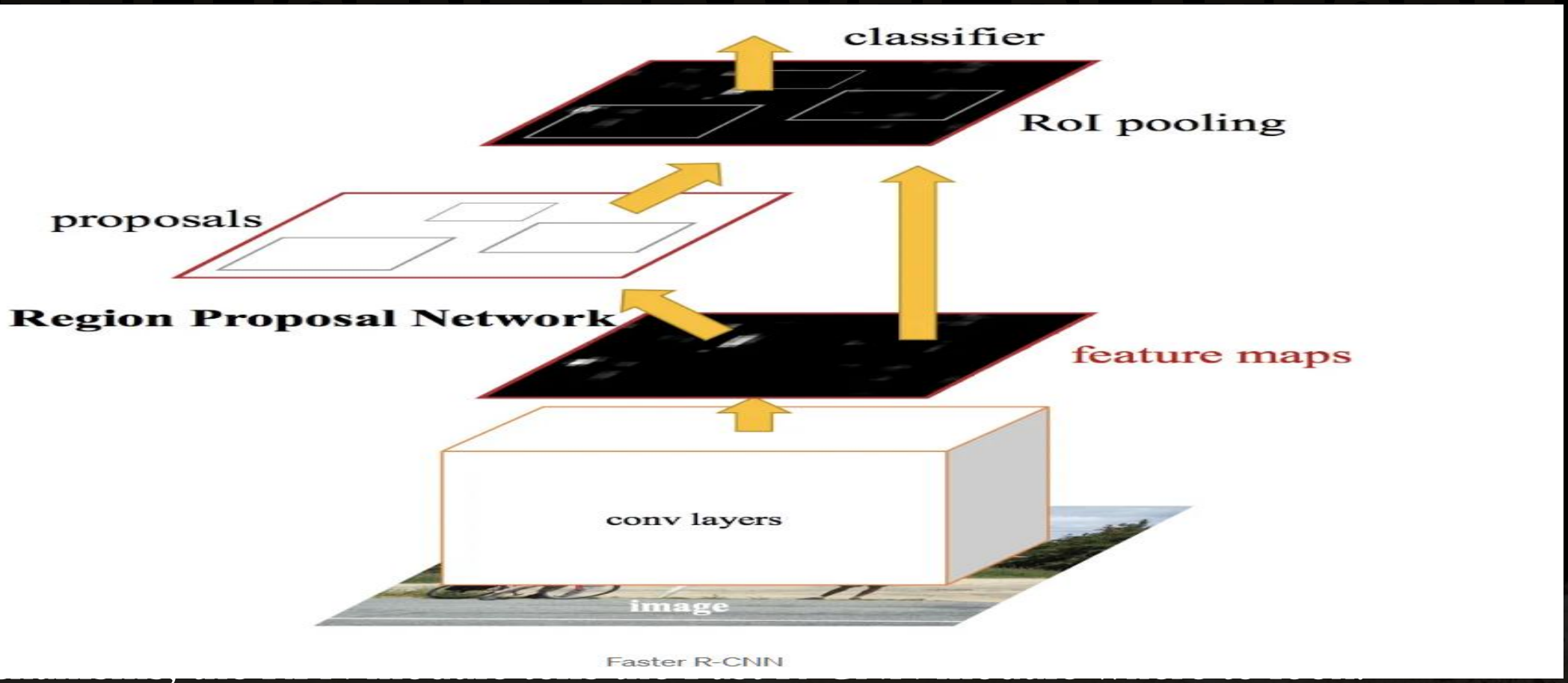
-**Complex Backgrounds:** SSDs can produce false positives or misclassify objects in complex or cluttered backgrounds due to confusing visual information.

- Trade-off between Speed and Accuracy: SSD excels in speed but may require sacrificing some accuracy. In precision-critical applications, alternatives might be preferred for higher accuracy.

- Customization Overhead: Fine-tuning SSDs for specific applications can be labor-intensive and resource-consuming, requiring deep learning expertise for effective customization and optimization.

RCNNS (FASTER & MASK)

1- Fast RCNN



Faster R-CNN

RCNNS (FASTER & MASK)

RPN vs. Selective Search:

- R-CNN and Fast R-CNN use selective search for region proposals, while in Faster R-CNN eliminates selective search by allowing the network to learn region proposals directly via the RPN, greatly improving speed and efficiency

Drawbacks of Faster R-CNN

Speed: Faster R-CNN is not truly real-time, as it typically processes images at a speed lower than 10 FPS on high-end GPUs

Complexity: multiple stages (RPN and detector), which increases the overall computational load and implementation complexity.

Training time :Faster R-CNN requires a substantial amount of time to train due to its intricate architecture and the need to balance the RPN and detection network.

RCNNS (FASTER & MASK)

Implementation Challenges for testing faster R-CNN on our dataset:

Complexity of the Model: Faster R-CNN's architecture, is inherently complex.

Training Time: Training model on our custom dataset was extremely time-consuming with non-satisfactory results.

Dataset Issues:

- Custom dataset constraint The complexity of adapting the Faster R-CNN implementation to handle our custom dataset added to the challenge.
- Parsing annotations from our custom dataset required substantial modifications to the training and inference scripts. This process was prone to errors and inconsistencies, further complicating the implementation.

RCNNS (FASTER & MASK)

Using API to train:

```
Accumulating evaluation results...
DONE (t=0.20s).
IoU metric: bbox
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.340
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.593
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.376
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = -1.000
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = -1.000
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.340
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.448
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.493
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.493
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = -1.000
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = -1.000
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.493
SAVING PLOTS COMPLETE...
SAVING PLOTS COMPLETE...
SAVING PLOTS COMPLETE...
SAVING PLOTS COMPLETE...
SAVING PLOTS COMPLETE...
SAVING PLOTS COMPLETE...
```

RCNNS (FASTER & MASK)

2- Mask RCNN

It is a deep learning model that combines object detection and instance segmentation. It is an extension of the Faster R-CNN architecture.

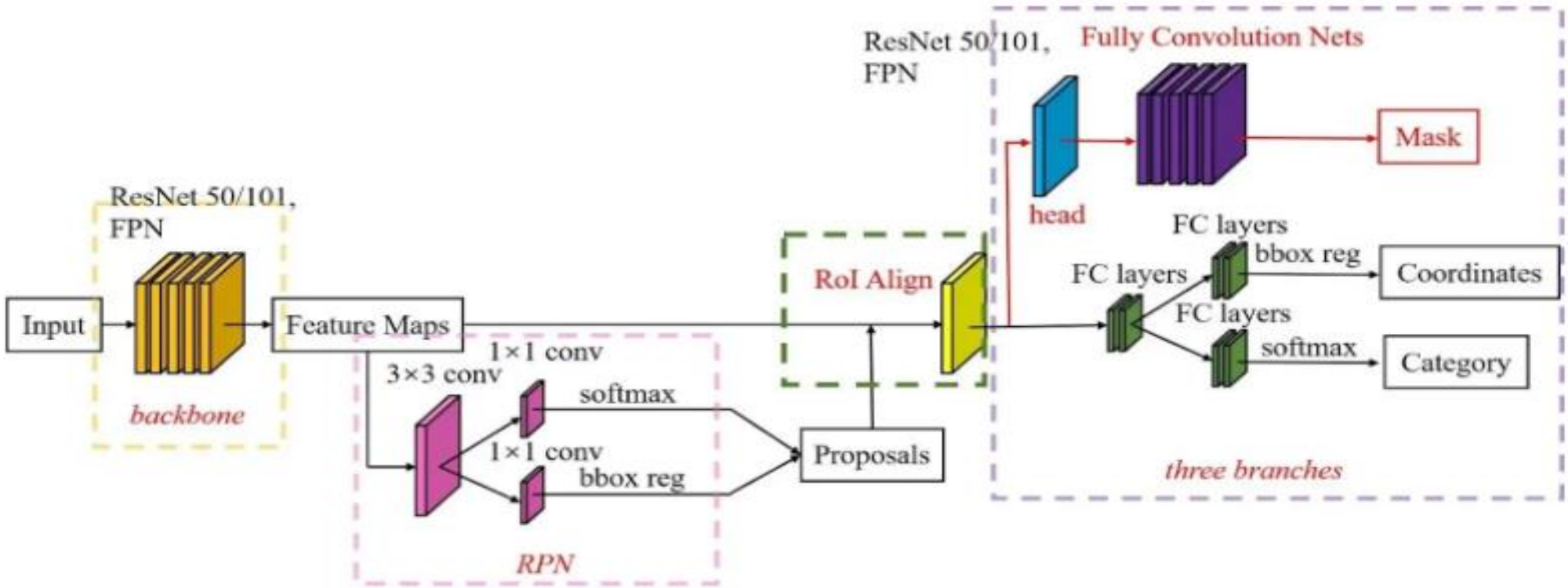
It extends Faster R-CNN by adding a "mask head" branch, enabling pixel-wise instance segmentation for precise boundaries

Key Innovations:

- ROIAlign: Uses bilinear interpolation during pooling to ensure accurate spatial information capture, improving segmentation accuracy.
- Feature Pyramid Network (FPN): Constructs a multi-scale feature pyramid for better object detection and segmentation across various sizes.

RCNNS (FASTER & MASK)

Mask R-CNN Architecture:



The Mask R-CNN framework for instance segmentation. Source

RCNNS (FASTER & MASK)

Mask R-CNN Limitations

1. Computational Complexity: Training and inference are resource-intensive, especially for high-resolution images or large datasets.

2. Small-Object Segmentation: Struggles with accurately segmenting very small objects due to limited pixel information.

3. Data Requirements: Requires a large amount of annotated data, which can be time-consuming and expensive to obtain.

4. Limited Generalization: The model's ability to generalize to unseen categories is limited, particularly with scarce data.

RCNNS (FASTER & MASK)

```
create_coco_tf_reco 100%[===================>]   9.93K  --.-KB/s    in 0s

2024-06-05 10:45:04 (56.4 MB/s) - 'create_coco_tf_record.py' saved [10171/10171]

2024-06-05 10:45:05.149404: I tensorflow/core/platform/cpu_feature_guard.cc:210] This TensorFlow binary is optimized to use ava
To enable the following instructions: AVX2 FMA, in other operations, rebuild TensorFlow with the appropriate compiler flags.
2024-06-05 10:45:06.166011: W tensorflow/compiler/tf2tensorrt/utils/py_utils.cc:38] TF-TRT Warning: Could not find TensorRT
2024-06-05 10:45:07.580132: I external/local_xla/xla/stream_executor/cuda/cuda_executor.cc:998] successful NUMA node read from
2024-06-05 10:45:07.641859: I external/local_xla/xla/stream_executor/cuda/cuda_executor.cc:998] successful NUMA node read from
2024-06-05 10:45:07.642233: I external/local_xla/xla/stream_executor/cuda/cuda_executor.cc:998] successful NUMA node read from
INFO:tensorflow:Found groundtruth annotations. Building annotations index.
I0605 10:45:08.182826 138577848553472 create_coco_tf_record.py:213] Found groundtruth annotations. Building annotations index.
INFO:tensorflow:3 images are missing annotations.
I0605 10:45:08.184632 138577848553472 create_coco_tf_record.py:226] 3 images are missing annotations.
INFO:tensorflow:On image 0 of 2353
I0605 10:45:08.185090 138577848553472 create_coco_tf_record.py:232] On image 0 of 2353
Traceback (most recent call last):
  File "/content/create_coco_tf_record.py", line 266, in <module>
    tf.app.run()
  File "/usr/local/lib/python3.10/dist-packages/tensorflow/python/platform/app.py", line 36, in run
    _run(main=main, argv=argv, flags_parser=_parse_flags_tolerate_undef)
  File "/usr/local/lib/python3.10/dist-packages/absl/app.py", line 308, in run
    _run_main(main, args)
  File "/usr/local/lib/python3.10/dist-packages/absl/app.py", line 254, in _run_main
    sys.exit(main(argv))
  File "/content/create_coco_tf_record.py", line 253, in main
    _create_tf_record_from_coco_annotations(
  File "/content/create_coco_tf_record.py", line 234, in _create_tf_record_from_coco_annotations
    _, tf_example, num_annotations_skipped = create_tf_example(
  File "/content/create_coco_tf_record.py", line 147, in create_tf_example
    run_len_encoding = mask.frPyObjects(object_annotations['segmentation'],
  File "pycocotools/_mask.pyx", line 294, in pycocotools._mask.frPyObjects
IndexError: list index out of range
```

YOLO (YOU ONLY LOOK ONCE) V5 & V7

1- YOLOv5:

- YOLOv5, part of the You Only Look Once (YOLO) family, is widely used for object detection.

- It processes images in a single pass, ensuring fast and efficient object detection.

Overall Architecture:

- Backbone: CSPDarknet53 for feature extraction.

- Neck: Combines FPN and PAN methods for

- Head: Generates predictions with bounding boxes, class confidence, and object coordinates.

YOLO (YOU ONLY LOOK ONCE) V5 & V7

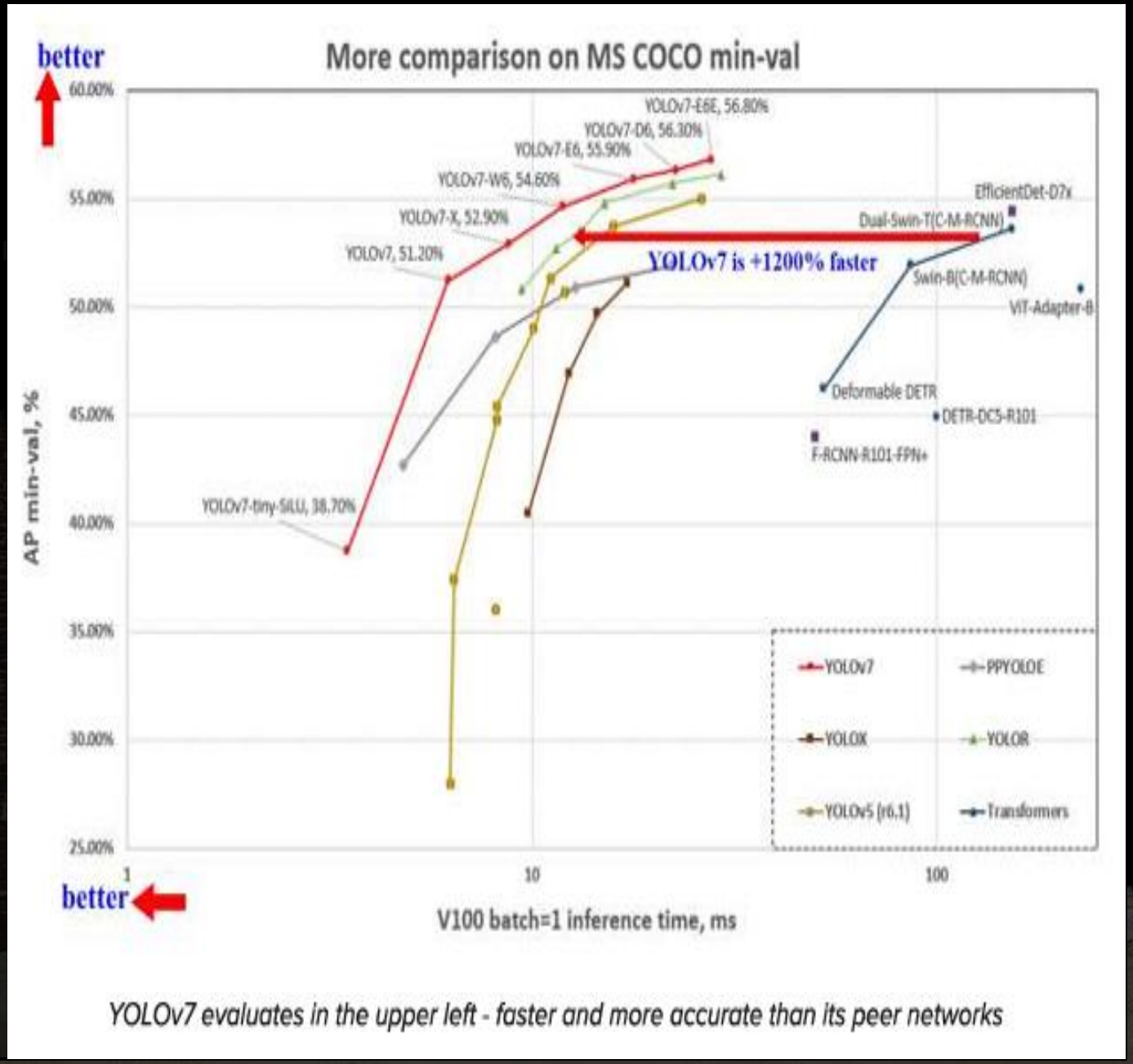
Drawbacks and Test Results of YOLOv5:

```
                 Class     Images  Instances          P          R      mAP50   mAP50-95: 100% 4/4 [00:36<
                   all        114        113    0.00199      0.562     0.0159     0.0046
              Akhenaten        114          2    0.00132        0.5      0.124     0.0372
Bent-pyramid-for-senefru        114         13     0.0029      0.692     0.0269    0.00915
Colossal-Statue-of-Ramesses-II        114          1          0          0          0          0
    Colossoi-of-Memnon        114         11    0.00352      0.818      0.015    0.00483
Goddess-Isis-with-her-child        114          5          0          0          0          0
             Hatshepsut        114          4   0.000357       0.25   0.000537   0.000215
        Hatshepsut-face        114          2      0.001        0.5    0.00126   0.000251
         Khafre-Pyramid        114         17    0.00366      0.941     0.0383    0.00933
   Mask-of-Tutankhamun        114         13    0.00676      0.923     0.0474     0.0142
              Nefertiti        114          2    0.00207          1    0.00703    0.00102
     Pyramid_of_Djoser        114          4    0.00478        0.5    0.00783    0.00306
               Ramessum        114          2   0.000682        0.5   0.000682   6.82e-05
Ramses-II-Red-Granite-Statue        114          2          0          0          0          0
  Statue-of-King-Zoser        114          2          0          0          0          0
Statue-of-Tutankhamun-with-Ankhesenamun        114          2    0.00126        0.5    0.00161   0.000525
Temple_of_Isis_in_Philae        114          1          0          0          0          0
    Temple_of_Kom_Ombo        114          1    0.00127          1    0.00258    0.00074
The Great Temple of Ramesses -        114          7    0.00208          1     0.0219    0.00601
amenhotep-iii-and-tiye        114          6    0.00383      0.833    0.00445   0.000952
      menkaure-pyramid        114          3    0.00345          1     0.0201    0.00654
                sphinx        114         13    0.00288      0.846     0.0138    0.00256
Results saved to runs/train/yolov5s_results
CPU times: user 1min 46s, sys: 13.4 s, total: 2min
Wall time: 3h 44s
```

YOLO (YOU ONLY LOOK ONCE) V5 & V7

Versions comparison:

Yolo v7 is faster and more accurate than other versions.



Why Yolo v7?

- YOLOv7 is a state-of-the-art real-time object detection
- Accuracy, Speed , Hardware Requirements , Efficiency.
- overall accuracy is 56.8% AP and specifically we got 51.4% AP.

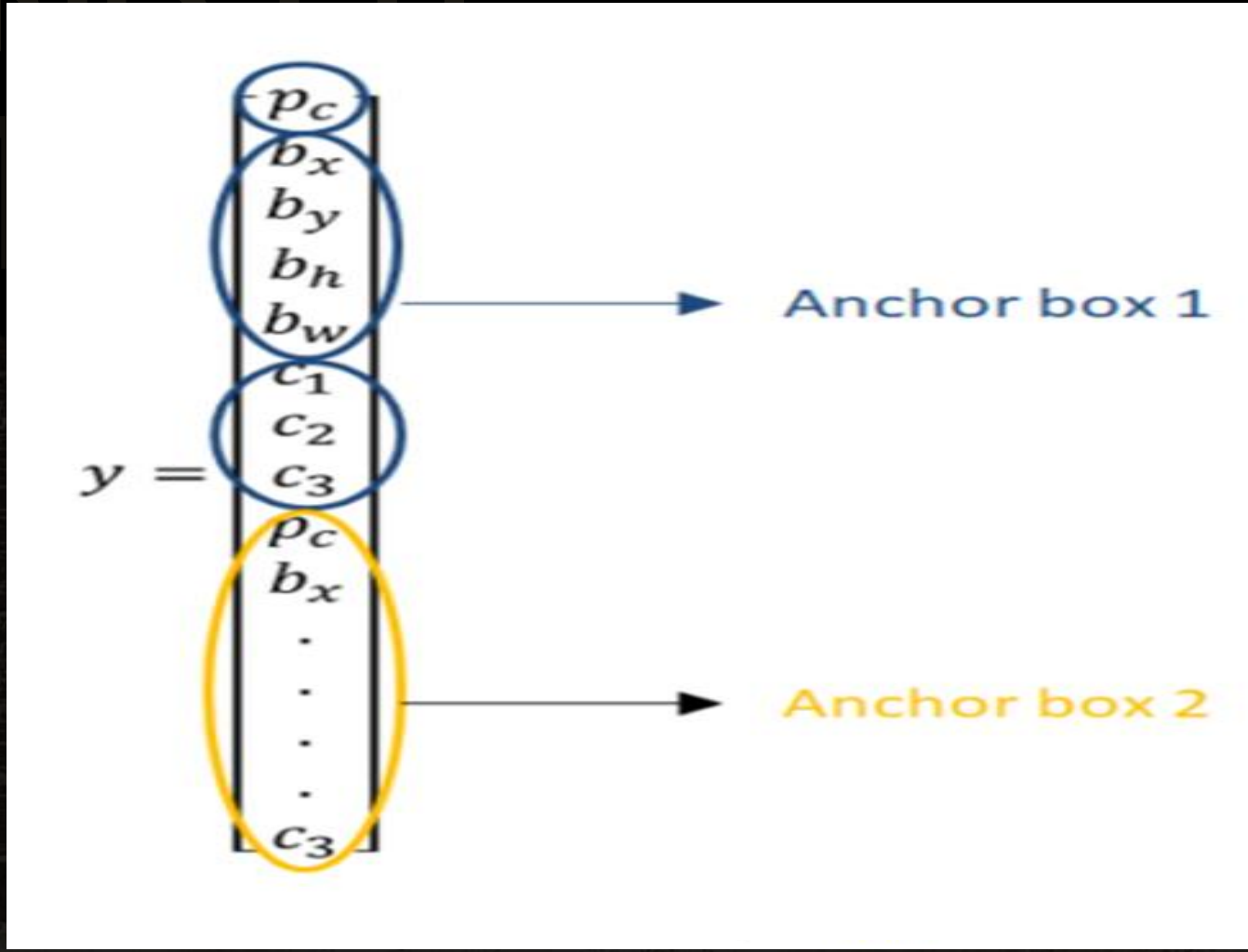
Table 2: Comparison of state-of-the-art real-time object detectors.

Model	#Param.	FLOPs	Size	FPS	AP^{test} / AP^{val}	AP^{test}_{50}	AP^{test}_{75}	AP^{test}_{S}	AP^{test}_{M}	AP^{test}_{L}
YOLOX-S [21]	9.0M	26.8G	640	102	40.5% / 40.5%	-	-	-	-	-
YOLOX-M [21]	25.3M	73.8G	640	81	47.2% / 46.9%	-	-	-	-	-
YOLOX-L [21]	54.2M	155.6G	640	69	50.1% / 49.7%	-	-	-	-	-
YOLOX-X [21]	99.1M	281.9G	640	58	51.5% / 51.1%	-	-	-	-	-
PPYOLOE-S [85]	7.9M	17.4G	640	208	43.1% / 42.7%	60.5%	46.6%	23.2%	46.4%	56.9%
PPYOLOE-M [85]	23.4M	49.9G	640	123	48.9% / 48.6%	66.5%	53.0%	28.6%	52.9%	63.8%
PPYOLOE-L [85]	52.2M	110.1G	640	78	51.4% / 50.9%	68.9%	55.6%	31.4%	55.3%	66.1%
PPYOLOE-X [85]	98.4M	206.6G	640	45	52.2% / 51.9%	69.9%	56.5%	33.3%	56.3%	66.4%
YOLOv5-N (r6.1) [23]	1.9M	4.5G	640	159	- / 28.0%	-	-	-	-	-
YOLOv5-S (r6.1) [23]	7.2M	16.5G	640	156	- / 37.4%	-	-	-	-	-
YOLOv5-M (r6.1) [23]	21.2M	49.0G	640	122	- / 45.4%	-	-	-	-	-
YOLOv5-L (r6.1) [23]	46.5M	109.1G	640	99	- / 49.0%	-	-	-	-	-
YOLOv5-X (r6.1) [23]	86.7M	205.7G	640	83	- / 50.7%	-	-	-	-	-
YOLOR-CSP [81]	52.9M	120.4G	640	106	51.1% / 50.8%	69.6%	55.7%	31.7%	55.3%	64.7%
YOLOR-CSP-X [81]	96.9M	226.8G	640	87	53.0% / 52.7%	71.4%	57.9%	33.7%	57.1%	66.8%
YOLOv7-tiny-SiLU	6.2M	13.8G	640	286	38.7% / 38.7%	56.7%	41.7%	18.8%	42.4%	51.9%
YOLOv7	36.9M	104.7G	640	161	51.4% / 51.2%	69.7%	55.9%	31.8%	55.5%	65.0%
YOLOv7-X	71.3M	189.9G	640	114	53.1% / 52.9%	71.2%	57.8%	33.8%	57.1%	67.4%
YOLOv5-N6 (r6.1) [23]	3.2M	18.4G	1280	123	- / 36.0%	-	-	-	-	-
YOLOv5-S6 (r6.1) [23]	12.6M	67.2G	1280	122	- / 44.8%	-	-	-	-	-
YOLOv5-M6 (r6.1) [23]	35.7M	200.0G	1280	90	- / 51.3%	-	-	-	-	-
YOLOv5-L6 (r6.1) [23]	76.8M	445.6G	1280	63	- / 53.7%	-	-	-	-	-
YOLOv5-X6 (r6.1) [23]	140.7M	839.2G	1280	38	- / 55.0%	-	-	-	-	-
YOLOR-P6 [81]	37.2M	325.6G	1280	76	53.9% / 53.5%	71.4%	58.9%	36.1%	57.7%	65.6%
YOLOR-W6 [81]	79.8G	453.2G	1280	66	55.2% / 54.8%	72.7%	60.5%	37.7%	59.1%	67.1%
YOLOR-E6 [81]	115.8M	683.2G	1280	45	55.8% / 55.7%	73.4%	61.1%	38.4%	59.7%	67.7%
YOLOR-D6 [81]	151.7M	935.6G	1280	34	56.5% / 56.1%	74.1%	61.9%	38.9%	60.4%	68.7%
YOLOv7-W6	70.4M	360.0G	1280	84	54.9% / 54.6%	72.6%	60.1%	37.3%	58.7%	67.1%
YOLOv7-E6	97.2M	515.2G	1280	56	56.0% / 55.9%	73.5%	61.2%	38.0%	59.9%	68.4%
YOLOv7-D6	154.7M	806.8G	1280	44	56.6% / 56.3%	74.0%	61.8%	38.8%	60.1%	69.5%
YOLOv7-E6E	151.7M	843.2G	1280	36	56.8% / 56.8%	74.4%	62.1%	39.3%	60.5%	69.0%

[1] Our FLOPs is calaculated by rectangle input resolution like 640 × 640 or 1280 × 1280.
[2] Our inference time is estimated by using letterbox resize input image to make its long side equals to 640 or 1280.

YOLO V7 ARCHITECTURE

How YOLOv7 works?
Image split into **grids SxS.**
For each cell there is **label=ground truth**
3.Label Encoding for each cell.
4.Apply Anchor boxes at each cell.
5. Find The output from training our model.
6. Find **IOU** between ground truth and predicted bounding box And take highest IOU.
7. Find **loss function** for coordinates , classes and confidence
and assign **hyper parameter** (lambda).
8. Find AP and **mAP**



3- MODEL TRAINING

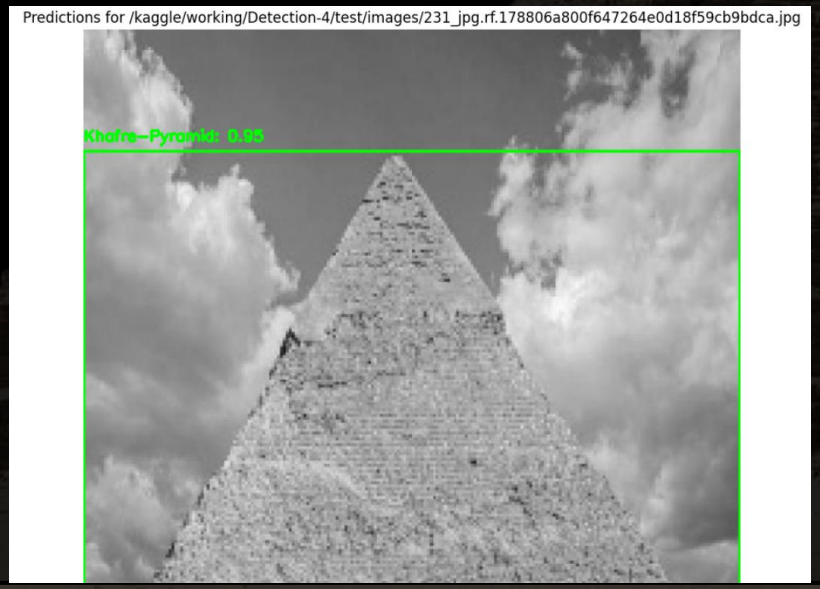
1. Prepare environment and use **GPU**
2. Download YOLOv7 repository .
3. folder for our dataset.
4. Get dataset from roboflow.
5. Download COCO starting checkpoint**.**
6. Begin training with **30 epochs.**
7. Get Results**:**
8. evaluation.

Class	Images	Labels	P	R	mAP@.5	mAP@.5:.95: 100% 21/21 [00:15<00:00, 1.39it/s]
all	654	658	0.614	0.584	0.643	0.345
Akhenaten	654	18	0.325	0.214	0.204	0.0713
Bent-pyramid-for-senefru	654	55	0.753	0.927	0.937	0.73
Colossal-Statue-of-Ramesses-II	654	22	0.844	0.492	0.684	0.315
Colossoi-of-Memnon	654	52	0.511	0.865	0.846	0.533
Goddess-Isis-with-her-child	654	15	0.737	0.933	0.918	0.328
Hatshepsut	654	53	0.691	0.415	0.546	0.182
Hatshepsut-face	654	19	0.737	0.444	0.628	0.365
Khafre-Pyramid	654	80	0.574	0.912	0.778	0.525
Mask-of-Tutankhamun	654	43	0.752	1	0.98	0.636
Nefertiti	654	20	0.436	0.7	0.61	0.266
Pyramid_of_Djoser	654	13	0.786	0.923	0.929	0.623
Ramessum	654	36	0.54	0.522	0.567	0.248
Ramses-II-Red-Granite-Statue	654	18	0.62	0.667	0.649	0.422
Statue-of-King-Zoser	654	22	0.655	0.545	0.587	0.259
Statue-of-Tutankhamun-with-Ankhesenamun	654	16	0.714	0.16	0.59	0.231
Temple_of_Isis_in_Philae	654	8	0.664	0.5	0.701	0.288
Temple_of_Kom_Ombo	654	17	0	0	0.129	0.0358
The-Great-Temple-of-Ramesses-II	654	45	0.716	0.689	0.792	0.443
amenhotep-iii-and-tiye	654	16	0.46	0.375	0.479	0.244
bust-of-ramesses-ii	654	12	1	0	0.216	0.114
menkaure-pyramid	654	24	0.288	0.917	0.693	0.496
sphinx	654	54	0.701	0.648	0.689	0.239

4- INFERENCE

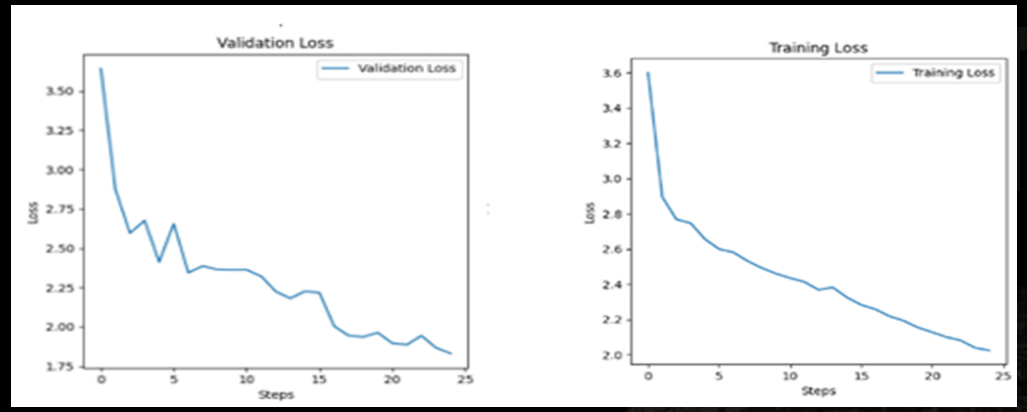

Colossoi-of-Memnon (Colossoi-of-Memnon 0.97


Bent-pyramid-for-senefru 0.94

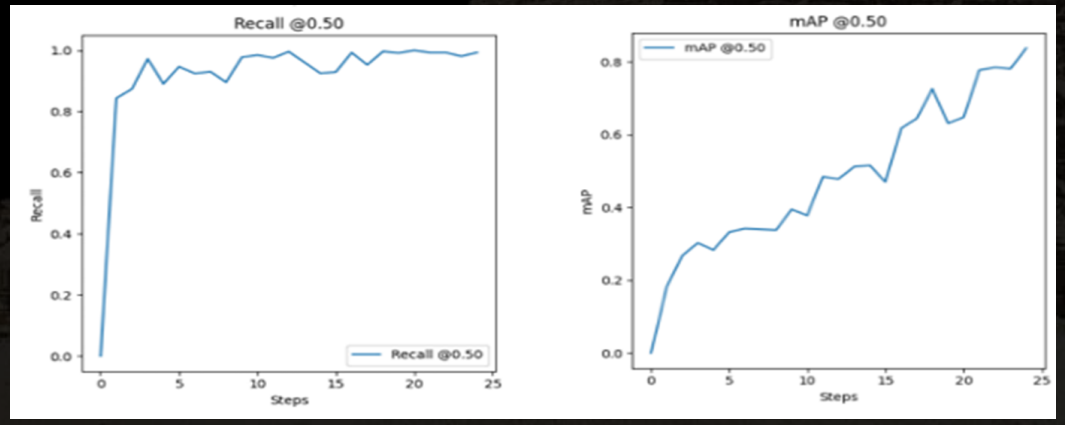

Predictions for /kaggle/working/Detection-4/test/images/231_jpg.rf.178806a800f647264e0d18f59cb9bdca.jpg
Khafre-Pyramid: 0.85

4- INFERENCE

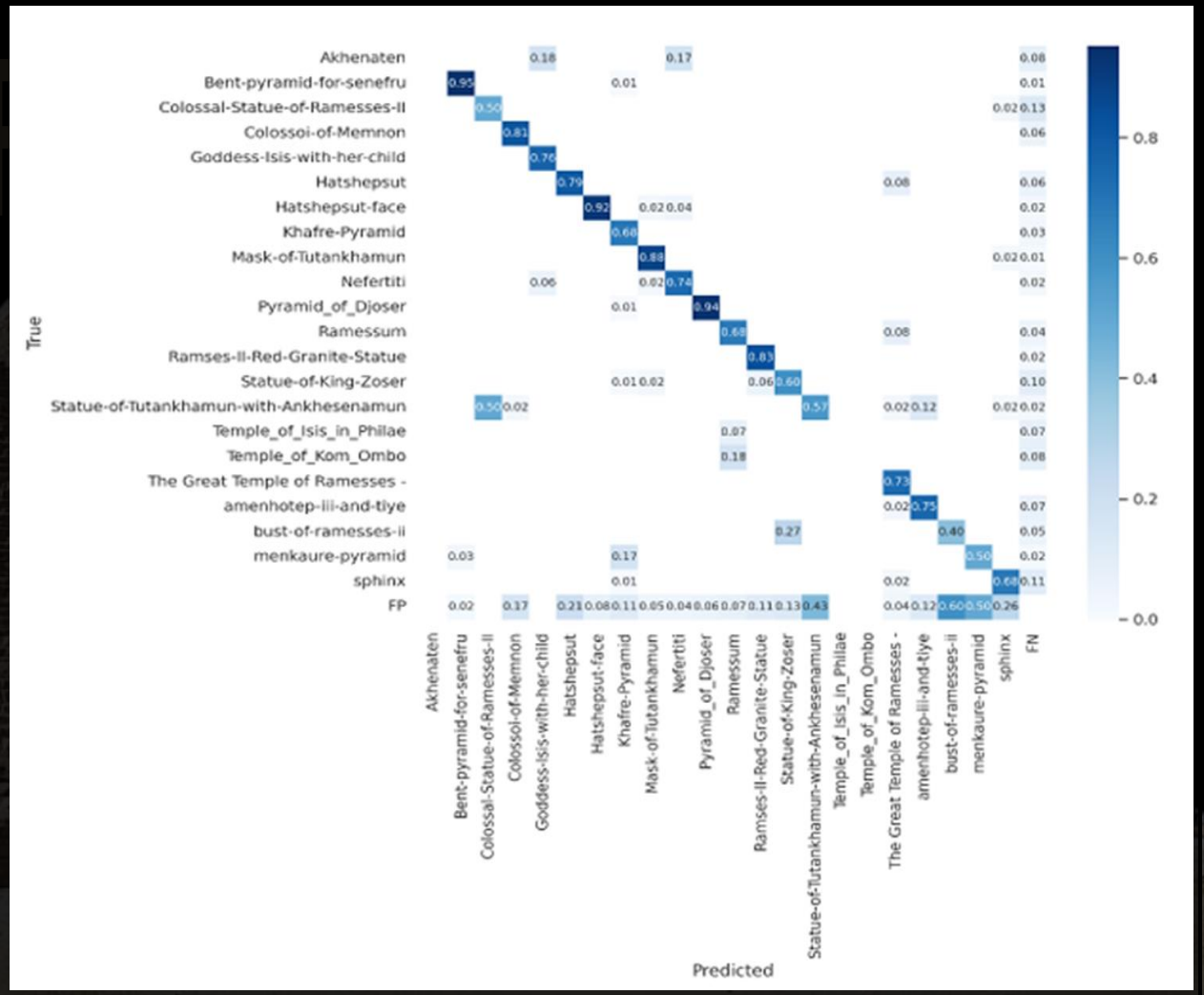
Validation loss & Training loss



Recall & mAP



Confusion Matrix



5- DEPLOYMENT

Deployment Process:

1. Setting Up the Environment:

- Created a virtual environment to manage dependencies and ensure compatibility.
- Installed necessary libraries and frameworks such as Flask, PyTorch, and OpenCV.

2. Flask Application:

- Developed a Flask web application to handle HTTP requests and perform inference using the YOLOv7 model.
- Encapsulated the core functionality in the `app.py` script, which contains the logic for loading the model and processing incoming requests.

3. Key Components of Deployment:

- **Model Loading:** The model is loaded using the `attempt_load` function from the YOLOv7 implementation. The path to the model weights is specified, and the model is loaded onto the CPU.
- **Handling Inference Requests:** The `/predict` endpoint accepts POST requests with an image file. The image is read, decoded, resized to 640x640 pixels, converted to a PyTorch tensor, processed by the model, and returns a response indicating successful inference.

5- DEPLOYMENT

Testing the Deployment

To ensure our deployed Flask application functions correctly, we conducted tests by sending images to the server and verifying the responses.

1. Setting Up the Test Script:

- Used the `requests` library to send HTTP requests to the Flask server.
- Utilized OpenCV (cv2) to read, resize, and preprocess test images.

2. Preparing and Encoding the Test Image:

- Selected a test image from our dataset.
- Resized the image to 640x640 pixels and converted it from BGR to RGB format.
- Encoded the image as JPEG and converted it to bytes for transmission.

3. Sending the Image to the Flask Server:

- Sent the encoded image bytes as a POST request to the `/predict` endpoint.
- Included error handling to manage any exceptions during the request.

5- DEPLOYMENT

4. Handling the Response:

- Parsed the result from JSON if the server responded with status code 200.
- Displayed the status code and error message if an error occurred.

Key Points:

- URL Configuration: Used `http://127.0.0.1:5000/predict` as the local address of the Flask server.

- Image Preparation: Ensured the image was resized to 640x640 pixels and converted to RGB format.

- Image Encoding: Encoded the image as a JPEG and converted it to bytes for the POST request.

- POST Request: Sent image bytes to the Flask server and handled the response to check for success or errors.

By following these steps, we successfully tested the deployment of our object detection model using Flask, ensuring it can receive images, perform inference, and return results.**

SOFTWARE PART (FRONT-END 'REACT')

The idea of Horus Eye:

The eye of horus is an ancient Egyption symbol that represents protection, healing and rejuvenation. It is often depicted as the eye of a falcon , with markings resembling a human eye. The symbol is believed to bring good fortune and ward off evil.

In our message of our project to see with different eyes that you can be protected, healed and rejuvenated

The design:

The design is spired from the ancient Egypt special the Horus Eye methodology

The challenges:

- Hard to visualize the design
- Integration
- The proper UX

MEET OUR TEAM

RAWAN SOUDI
ML TEAM
-TEAM LEADER-

SARAH SAMEH
ML TEAM

FATMA SAEED
ML TEAM

MEET OUR TEAM

ANAS SAEED
ML TEAM

AMIR SAMIR
SOFTWARE TEAM
FRONT END

FADY OSAMA
SOFTWARE TEAM
-BACK END-