Deep learning based Object Detection

Harsh Shah, Lalithambal Swaminathan

Wilfrid Laurier University
Waterloo, Canada

{shah5610, swam5140}@mylaurier.ca

ABSTRACT

In this project, we are using deep learning-based methodologies to perform object detection on construction safety workers dataset. As we were more inclined towards accuracy and not on speed for this work, we selected Faster R-CNN as the base algorithm for the project and successfully achieved good detection results with mAP value of 64% for the hard hat workers dataset.

1. INTRODUCTION

Computer vision is a field of artificial intelligence that teaches computers to read and understand the visual world. Using digital images, videos from cameras, and deep learning models, machines can accurately recognize and distinguish objects and then react to what they see. Object detection has been an active area of research for several decades. A wide range of applications, notable technological inventions are a couple of the key factors for the significant advancement of object detectors in recent years. The domain of computer vision embodies several recognition problems, like image classification, object detection, semantic segmentation, and instance segmentation. The task of the object localization is to localize that object of interest with a bounding box, while object detection identifies multiple object categories along with the exact locations of all those objects and shows detected objects via bounding boxes.

1.1 Motivation

Traditional object detection techniques were used to detect and classify the generic as well as salient object categories which in compare to today’s advancement were complex in design, had weak performance with large memory footprints. With the recent developments in deep learning technology, the object detection algorithms more or less have a contribution of CNN (Convolutional Neural Network) and its properties. Considering CNN's excellent ability to extract features, we decided to go with deep learning-based techniques and use the same to develop a safety helmet detection system.

1.2 Project Overview

In this project, based on the previous studies on computer vision-based object detection, we develop a deep learning-based method for the real-time detection of safety helmets at the construction site. The detection of safety helmets automatically worn by construction workers at the construction site and timely warning of workers without helmets can widely avoid accidents caused by ignorance of workers without wearing safety helmets. The CNN used in the current work is trained using the TensorFlow framework. The current work involves a deep learning-based safety helmet detection model and a safety helmet image dataset for further research. The project work allows detecting the helmets and improving safety management.

1.3 Problem Definition

Construction is a high-risk industry where construction workers tend to get hurt or die during work. It is necessary to oversee the safe protective equipment wearing condition of the construction workers at the construction site. Safety helmets can bear and disperse the hit of falling objects and ease the damage of workers falling from heights. Due to negligence and less awareness of their safety, construction workers often ignore safety helmets. So, to provide a solution by designing a system that detects the helmets and also individuals without wearing a helmet can help in maintaining safety regulations with automation involvement.

1.4 Objective

- The main aim of this project is to detect the helmet from the hard hat workers image dataset with reasonably higher mAP values and higher confidence threshold of detected bounding boxes.

1.5 Data Source

- Source: ROBOFLOW Open-Source dataset, Hard Hat Workers [1]
- Description: "The Hard Hat dataset is an object detection dataset of workers in workplace settings that require a hard hat. Annotations also include examples of just "person" and "head," for when an individual may be present without a hard hat."
- Shared by: Northeastern University - China (April 2020)
- Total Images: 7041

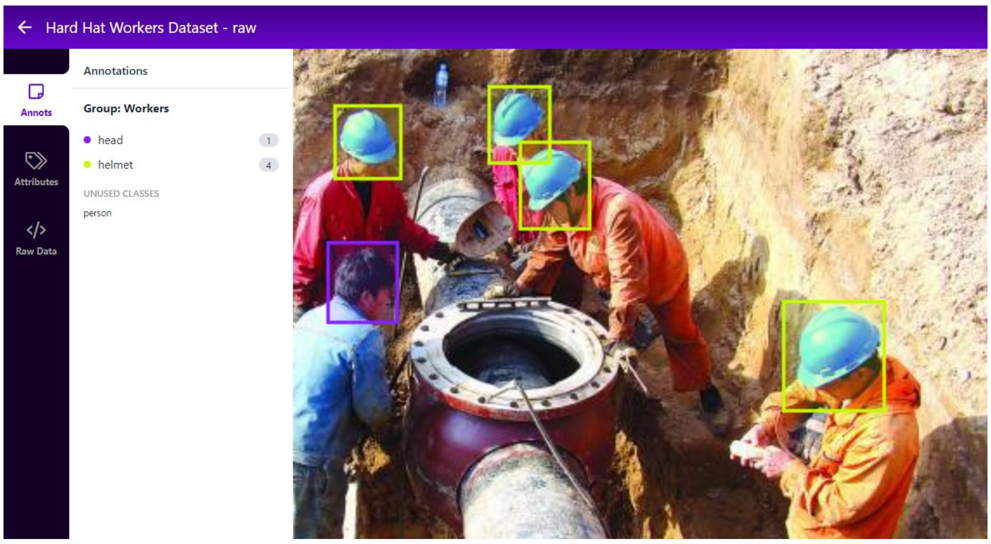


Fig. 1. – Hard Hat Workers dataset sample with visualized annotations [1]

1.6 Hardware and Tools

- Framework: Tensorflow Object Detection API
- Platform: Google Colab (with free GPU)
- Programming Language: Python 3.x
- Libraries:
    - Tensorflow_gpu 1.15, tf_slim, Cython
    - Pillow, Matplotlib
    - Pycocotools

1.7 Evaluation Metric

For evaluating the performance of object detectors, there are mainly three criteria [2]:

- Precision
- Recall
- Detection speed in Frames Per Second (FPS)

As in our work, only image dataset is used and no video data is there, **Average Precision (AP)** is preferred for Object detection which is derived from precision and recall and originally introduced in VOC2007. Intersection over Union (IoU) is used to measure the object localization accuracy.

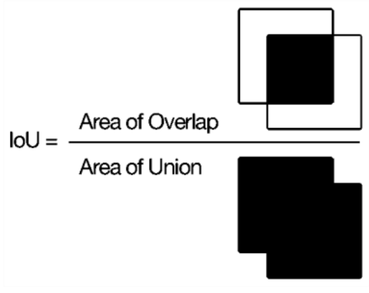


Fig. 2. IoU Equation

IoU threshold is predefined with a value, say 0.5. If IoU between the predicted box and the ground truth box is greater than the set threshold, the object will be identified as “successfully detected”, otherwise identified as “miss”. As per our research, **mAP@0.5IoU** has been a standard metric for detection problems so we will be using this mainly to evaluate our performance.

2. ALGORITHM

2.1 Faster R-CNN

To address the issues caused by conventional techniques like selective search and edge boxes that use visual signals and are difficult for detectors to learn the data-driven approach, Faster R-CNN was proposed shortly after the release of Fast R-CNN. A fully convolutional layer called Region Proposal Network (RPN) which creates proposal sets on each feature map location by working on random images and is used in Faster R-CNN. The derived feature map yields feature vectors that are fed into the classification layer and then into a bounding box regression layer for localization leading to Object detection.

This model consists of two modules where the first one is a fully convolutional network that proposes regions, and the second is the Faster R-CNN detector that uses the proposed regions, and these two modules form the unified network for Object detection. The region proposal network (RPN) is for generating region proposals and a network using these proposals to detect objects.

RPN takes an input image of any size and provides a set of rectangular object proposals as output where each proposal has an object-ness score. A small network that takes as the input of an n × n window of the feature map slid over the output convolutional feature map. This feature is given as an input into two fully connected layers out of which one is a box classification layer, and another is a box regression layer. This typical architecture is implemented with n x n convolutional layer followed by 1x1 convolutional layers.

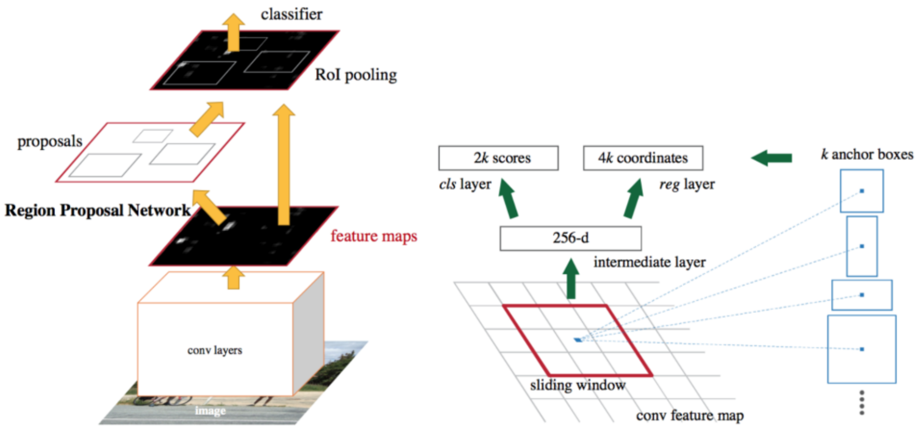


Fig. 3. Left side: Faster R-CNN Architecture, Right side: Region Proposal Network [2]

Anchors: Multiple region proposals are simultaneously predicted at each sliding-window location, and k denotes the maximum number of possible proposals for each location.

ROI Pooling: As the Regional Proposal Network produces proposed regions of different sized convolutional feature maps, Region of Interest Pooling handles the problem to work with features of different sizes by reducing the feature maps into equal sizes. ROI pooling splits the input feature map into a decided number of roughly equal regions, followed by Max pooling on each region.

The main difference between Fast R-CNN and Faster R-CNN is that the latter uses a selective search to generate region proposals. The time cost of generating region proposals is much smaller in RPN than selective search when RPN shares the most computation with the object detection network.

2.2 Inception V2

Inception network was complex, developed to increase the performance when compared to other CNN classifiers. Inception v2 was developed to reduce 'representational bottleneck' which occurs when the dimension of the input image is reduced drastically causing loss of information.

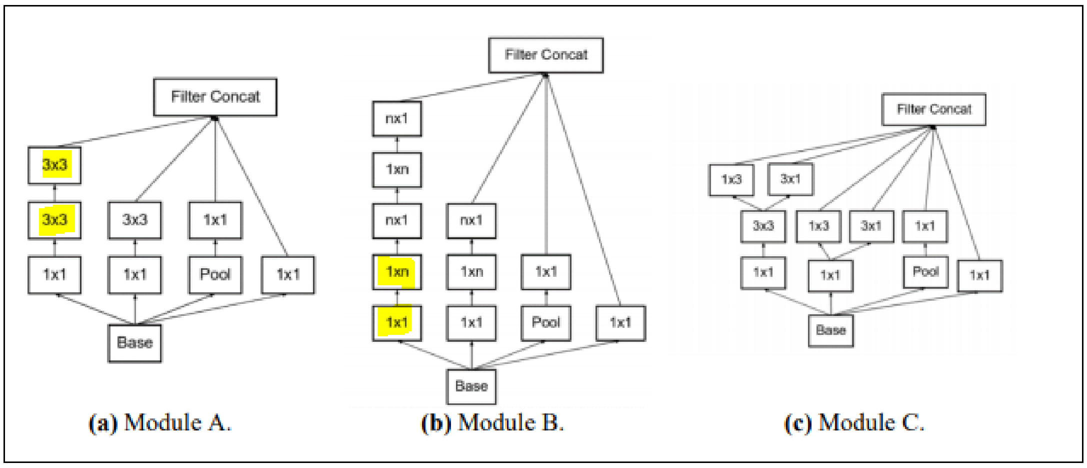


Fig. 4 – Module Inception V2 [3]

In this version, a 5x5 convolution is factorized to two 3x3 convolution layers to increase the computational speed and the former is 2.78 times expensive than the latter convolution operation. And also combining 1 x n and n x 1 convolutions by factorizing convolutions of filter size n x n is found to be 33% cheaper than a 3x3 convolution.

In order to remove the representational bottleneck, filter banks present in the module were made wider instead of having it deeper. Having the deeper filter would result in an extreme reduction in dimension leading to loss of information.

3. PROJECT WORKFLOW

In this section, we are going to explain the steps that we performed to implement the system. In below Fig. 5 the abstract workflow of our implementation is shown.

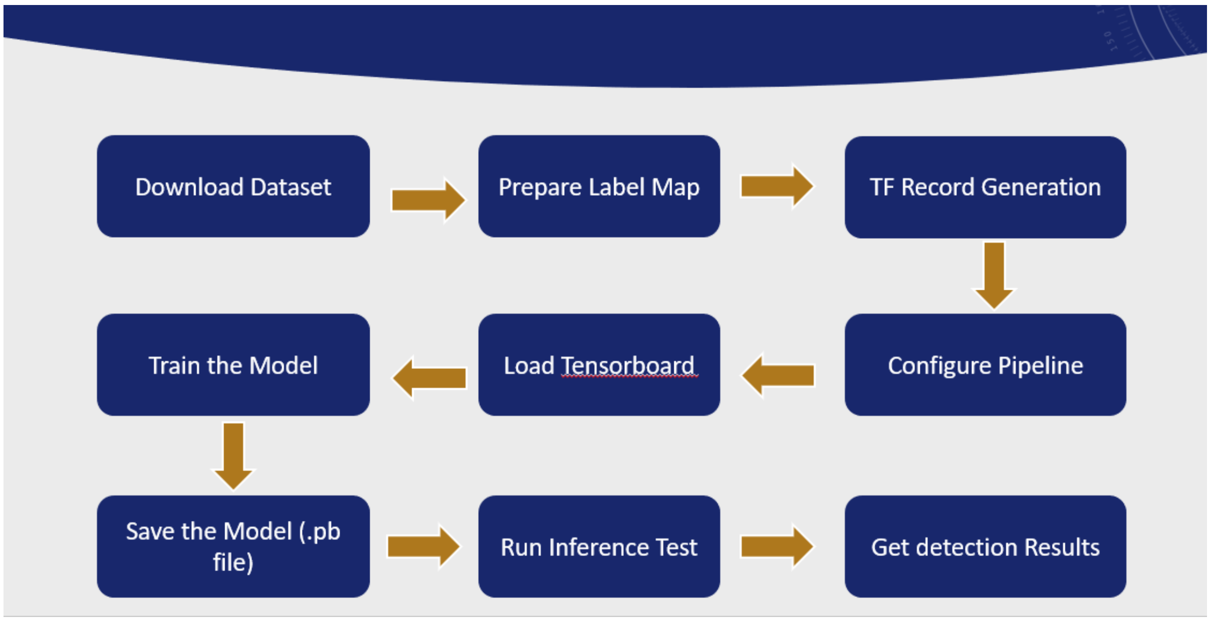
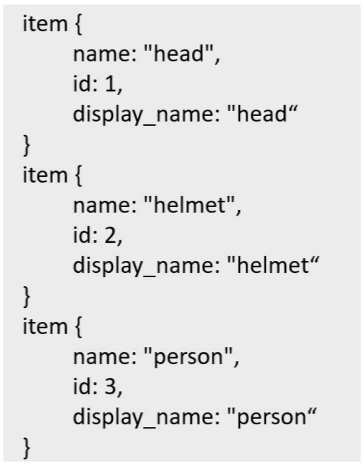


Fig. 5 - Project Workflow

In the above Fig. 5, as we can see that after downloading the dataset, preparing label map is the next step. In our work, as we have 3 classes to detect, the label map is prepared as below in Fig. 6.

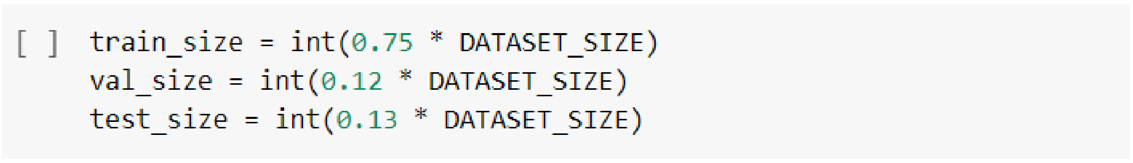


```
item {
      name: "head",
      id: 1,
      display_name: "head“
}
item {
      name: "helmet",
      id: 2,
      display_name: "helmet“
}
item {
      name: "person",
      id: 3,
      display_name: "person“
}
```

Fig 6. - Label Map

A. Preparing Train Test Data

In TensorFlow object detection framework, to train the model, input data has to be in TFRECORD (.record) format. For our work, the source has already provided (.record) files for training as well as testing. From source, total of 7041 images were provided.



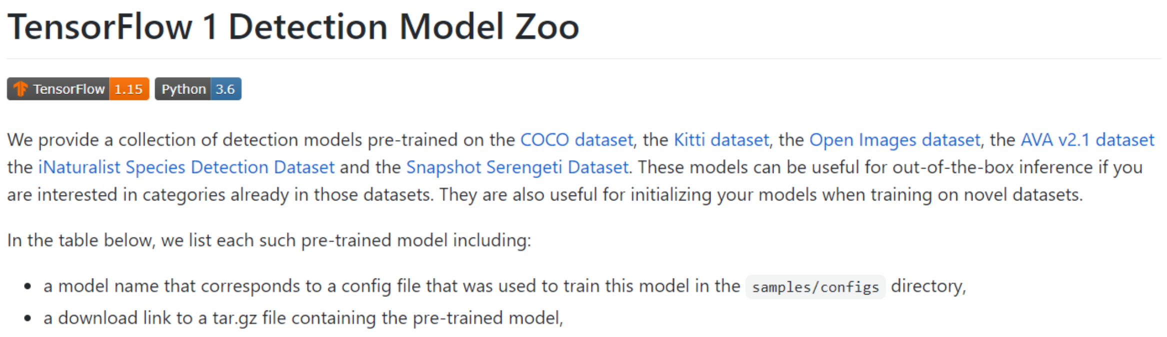
```
[ ]  train_size = int(0.75 * DATASET_SIZE)
     val_size = int(0.12 * DATASET_SIZE)
     test_size = int(0.13 * DATASET_SIZE)
```

Fig. 7 - Train | Validation | Test Split

As showm in above Fig. 7, we have used 75%-12%-13% split for train test and validation data.

B. Configuring Pipeline

In this project we are using the pre-trained model which is provided by TensorFlow 1 Model Zoo.



TensorFlow 1 Detection Model Zoo

TensorFlow 1.15 Python 3.6

We provide a collection of detection models pre-trained on the COCO dataset, the Kitti dataset, the Open Images dataset, the AVA v2.1 dataset the iNaturalist Species Detection Dataset and the Snapshot Serengeti Dataset. These models can be useful for out-of-the-box inference if you are interested in categories already in those datasets. They are also useful for initializing your models when training on novel datasets.

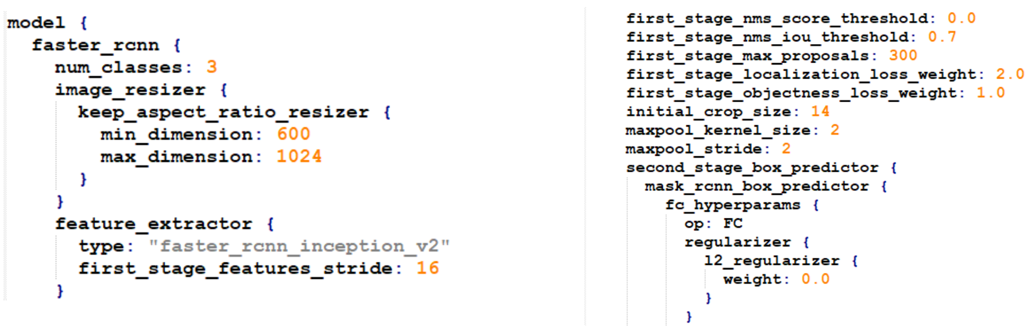
In the table below, we list each such pre-trained model including:

- a model name that corresponds to a config file that was used to train this model in the `samples/configs` directory,
- a download link to a tar.gz file containing the pre-trained model,

Fig 8. TensorFlow Detection Model Zoo official website snapshot [4]

A pre-trained model is basically trained on a large-scale dataset which consist of a more generic object categories, and is saved for other people to utilize it for a specific task. This phenomenon is generally termed as ‘transfer learning’ in which developers does not have to build the model from scratch which is a cumbersome and required large memory footprint. To avoid these problems, customized the configurations and using these pre-trained weights help train the model faster and also achieve overall better performance.

To customize the configurations of the parameters, we did some research and analysis and understood few parameters and their importance. These details are summarized in Table 1 with the final values of those parameters that we selected to fine tune our model.



```
model {
  faster_rcnn {
    num_classes: 3
    image_resizer {
      keep_aspect_ratio_resizer {
        min_dimension: 600
        max_dimension: 1024
      }
    }
    feature_extractor {
      type: "faster_rcnn_inception_v2"
      first_stage_features_stride: 16
    }
```

```
first_stage_nms_score_threshold: 0.0
first_stage_nms_iou_threshold: 0.7
first_stage_max_proposals: 300
first_stage_localization_loss_weight: 2.0
first_stage_objectness_loss_weight: 1.0
initial_crop_size: 14
maxpool_kernel_size: 2
maxpool_stride: 2
second_stage_box_predictor {
  mask_rcnn_box_predictor {
    fc_hyperparams {
      op: FC
      regularizer {
        l2_regularizer {
          weight: 0.0
        }
      }
```

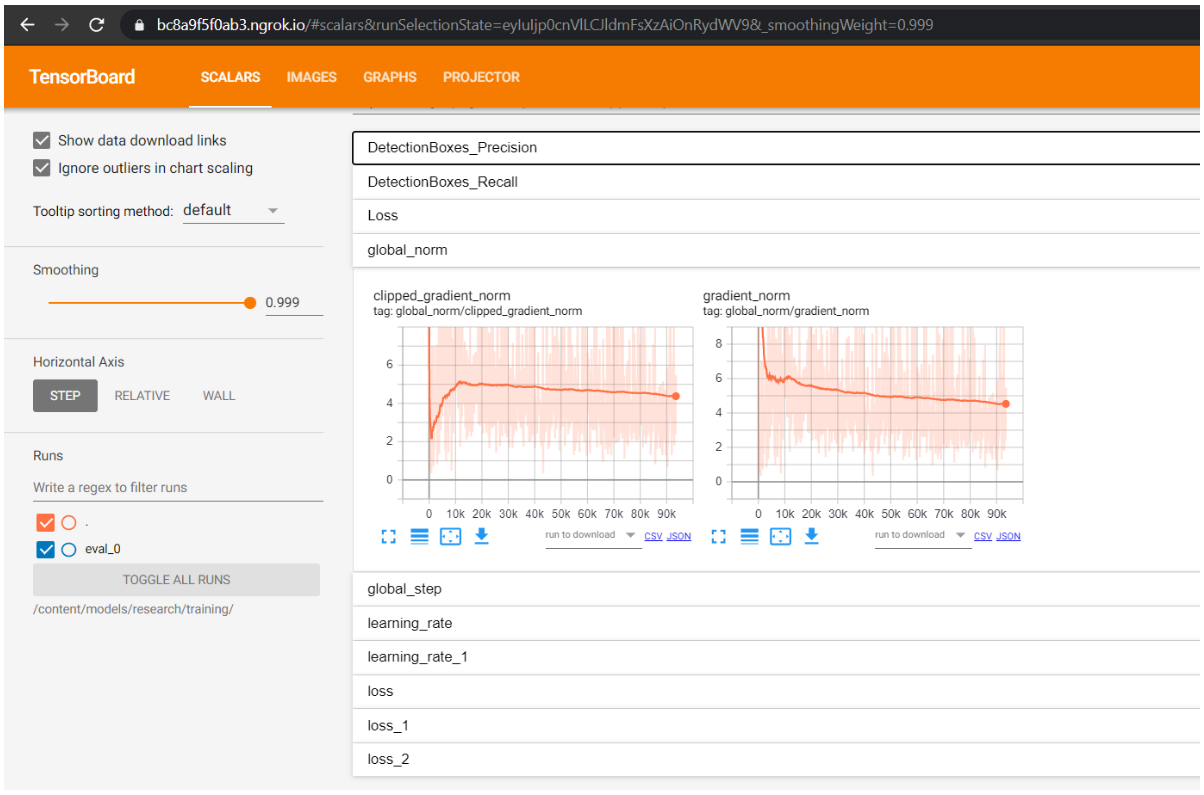
Fig 9. Our customized faster_rcnn_inception_v2_pets configuration file []

TABLE 1 HYPERPARAMETER SELECTION

Parameter	Description	Final Value
num_classes	Total number of classes	3
iou_threshold	"to measure the overlap of a predicted versus actual bounding box for an object"	0.6
batch_size	This is different from epoch and defines the number of samples to process before updating internal model parameters.	1 (Due to restricted GPU usage)
Initial_learning_rate	Defines how quickly the model can be adjusted to problem.	0.002
num_steps	Total number of training steps before finishing it.	120000
metrics_set	Selection of benchmark metric for specific purpose. Few other available options are: pascal_voc_detection_metrics open_images_metrics	coco_detection_metrics

C. Load TensorBoard

In the domain of machine learning, to be able to measure the performance is vital as it helps in improving the system in next iteration. As per the official documentation, "TensorBoard is a tool for providing the measurements and visualizations needed during the machine learning workflow. It enables tracking experiment metrics like loss and accuracy, visualizing the model graph, projecting embeddings to a lower dimensional space, and much more."


bc8a9f5f0ab3.ngrok.io/#scalars&runSelectionState=eyluljp0cnVlLCJldmFsXzAiOnRydWV9&_smoothingWeight=0.999

TensorBoard SCALARS IMAGES GRAPHS PROJECTOR

☑ Show data download links
☑ Ignore outliers in chart scaling
Tooltip sorting method: default

Smoothing 0.999

Horizontal Axis: STEP RELATIVE WALL

Runs
Write a regex to filter runs
☑ .
☑ eval_0
TOGGLE ALL RUNS
/content/models/research/training/

DetectionBoxes_Precision
DetectionBoxes_Recall
Loss
global_norm

clipped_gradient_norm
tag: global_norm/clipped_gradient_norm

gradient_norm
tag: global_norm/gradient_norm

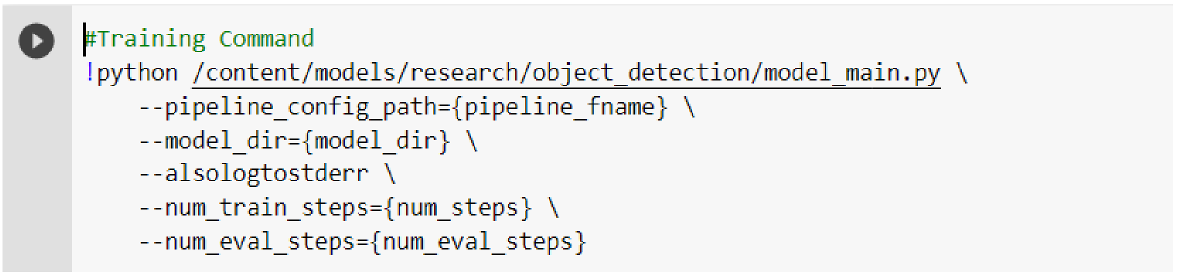
global_step
learning_rate
learning_rate_1
loss
loss_1
loss_2

Fig. 10. – Sample TensorBoard Snapshot

As per the official documentation, “the ***Scalars Dashboard*** shows how the loss and metrics change with every epoch.” We have utilized this dashboard mainly to track the training speed, learning rate, and other scalar values.”

D. Train the Model

Now, after completing all the pre-steps to finally train the model below is the command shown in Fig. 11. As we are going to utilize the free GPU with limited memory and storage provided by Google, it will take several hours to get some good results. While this training is in process, TensorBoard gives insight into this learning task.



```
#Training Command
!python /content/models/research/object_detection/model_main.py \
    --pipeline_config_path={pipeline_fname} \
    --model_dir={model_dir} \
    --alsologtostderr \
    --num_train_steps={num_steps} \
    --num_eval_steps={num_eval_steps}
```

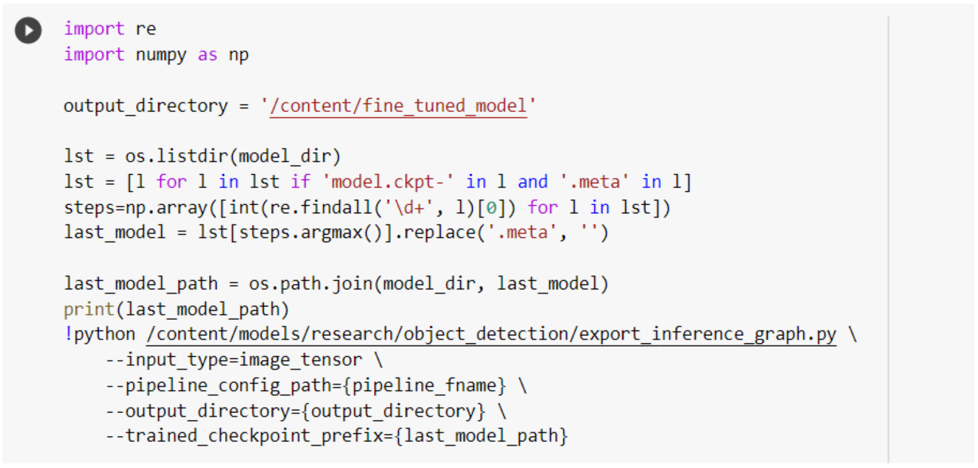
Fig. 11. Training Command Code Snippet

Passing arguments explanations:

- pipeline_config_path = this is the path to the custom configuration file which consist of parameters.
- model_dir = this is the output model directory.
- alsologtostderr = this will send the logs to STDERR “standard file”.
- num_train_steps = total training steps
- num_eval_steps = total evaluation steps

E. Save Fine-Tuned Model

After the training is completed, the model is saved or exported by following code.

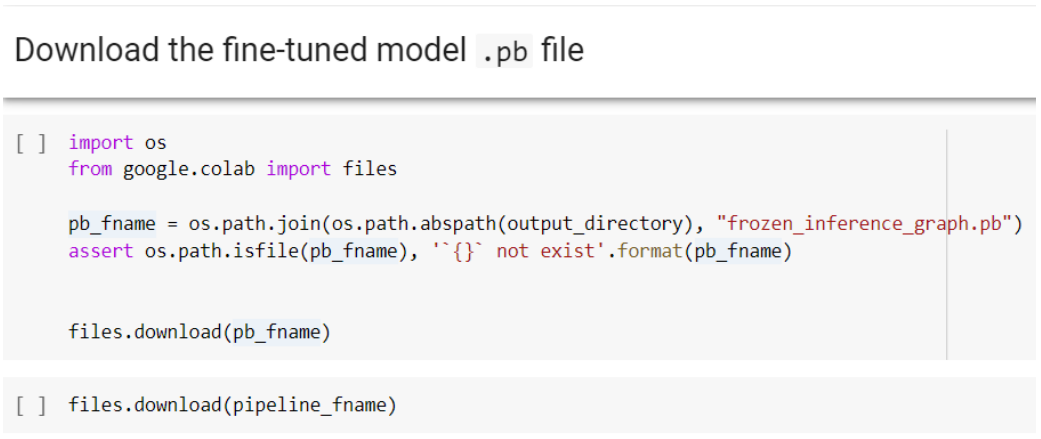


```
import re
import numpy as np

output_directory = '/content/fine_tuned_model'

lst = os.listdir(model_dir)
lst = [l for l in lst if 'model.ckpt-' in l and '.meta' in l]
steps=np.array([int(re.findall('\d+', l)[0]) for l in lst])
last_model = lst[steps.argmax()].replace('.meta', '')

last_model_path = os.path.join(model_dir, last_model)
print(last_model_path)
!python /content/models/research/object_detection/export_inference_graph.py \
    --input_type=image_tensor \
    --pipeline_config_path={pipeline_fname} \
    --output_directory={output_directory} \
    --trained_checkpoint_prefix={last_model_path}
```

Fig. 12. Save fine tuned model

Here, after setting up the output directory to save frozen_inference_graph.pb, the last checkpoint model is picked up by last_model_path variable. Then by using the above command of tensorflow API, inference graph is saved to output directory.

After saving the model, we have written code to download the configuration file and saved model for that particular training iteration to further run the inference test with it and check detection results.



Download the fine-tuned model `.pb` file

```
[ ]  import os
     from google.colab import files

     pb_fname = os.path.join(os.path.abspath(output_directory), "frozen_inference_graph.pb")
     assert os.path.isfile(pb_fname), '`{}` not exist'.format(pb_fname)


     files.download(pb_fname)
```

```
[ ]  files.download(pipeline_fname)
```

Fig. 13. Dowload fine tuned model code snippet

F. Evaluation/Inference Test

Now, to check the detection results on test data we first loaded the test data and the saved model in drive and then after mounting that drive data, we did inference test.

To perform this, TensorFlow detection graph, labels, test images as NumPy arrays are to be loaded. In the final step, these images with bounding boxes and confidence threshold are plotted using matplotlib library.

Following were the outputs:

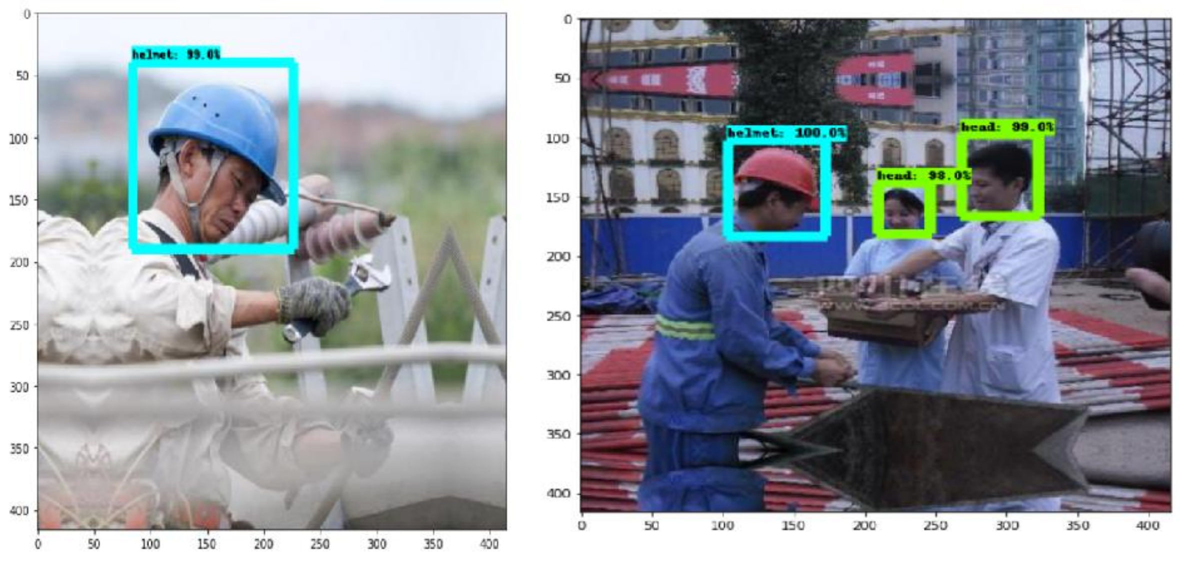


Fig. 14. Detection results I with bounding boxes.
Blue for helmet (labelled as helmet) and
Green for no-helmet (labelled as head)

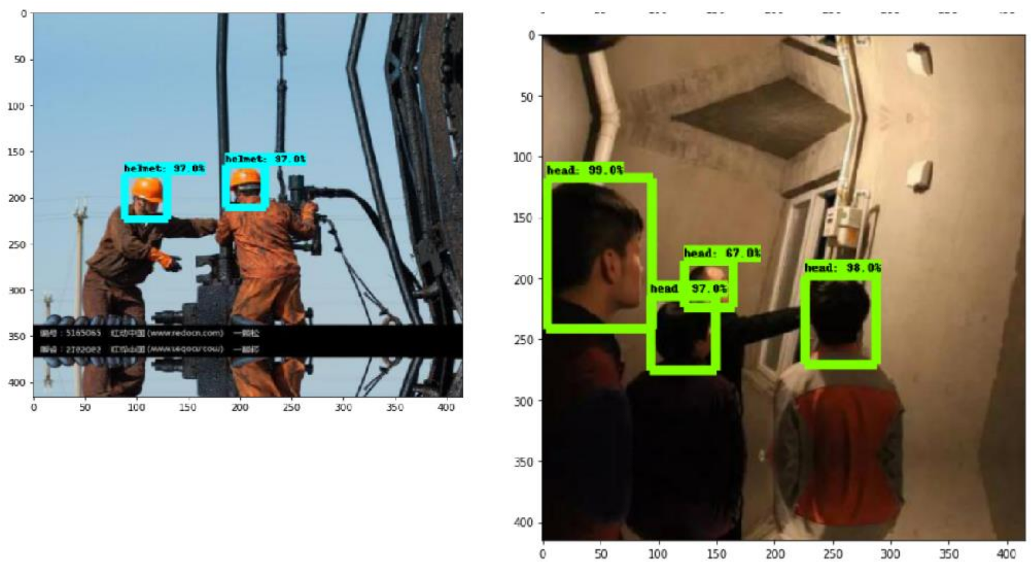


Fig. 15. Detection results II

4. PERFORMANCE

In this section, we are going to show the mAP, AR(Average Recall) values and validation loss.

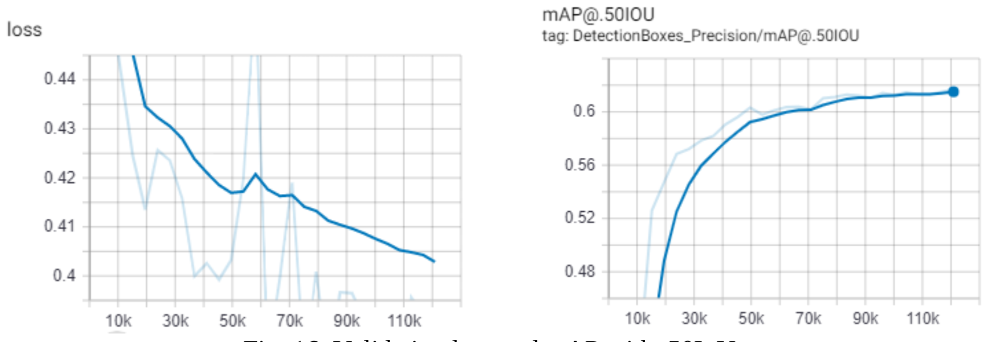


Fig. 16. Validation loss and mAP with .50IoU

The validation loss after training steps of 120000 was 0.40 and with that we got 64% of mAP for IoU >= 0.5 for detection boxes as can be seen in Fig. 16.

In Fig. 17, average recall values are shown for different object sizes and as per our observation, these values are somewhat similar which means model has generalized the learning and so different sizes of helmet (pixel wise coverage) in the image should be detected with similar confidence threshold and these observations are supported in the detection results figures, Fig. 14,15.

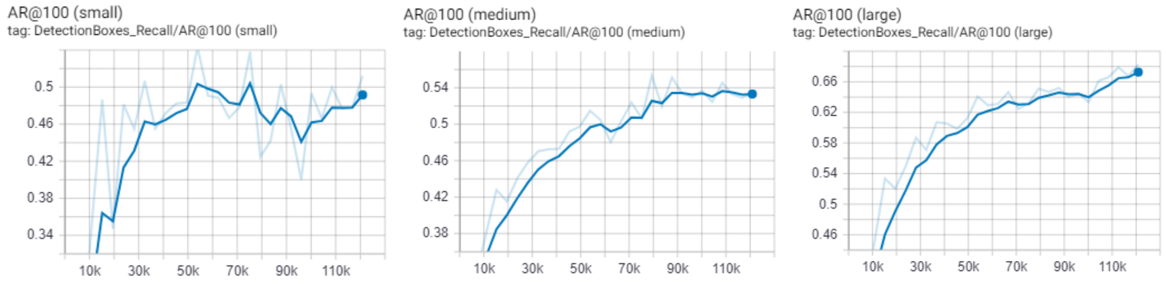


Fig. 17. Average Recall Graph

5. CONCLUSION

In this project, we are able to detect the helmets of the construction workers and the individuals without helmet with high confidence values of bounding boxes in Hard hat workers dataset. Yet the results obtained cannot be comparatively studied with other researches or projects as there was no study done on the algorithm with the current utilised dataset. The mAP values can be further increased by tuning more configuration parameters with a dedicated GPU based training. As per our analysis, if a real time detection via webcam feature has to be included for future directions, then the current backbone architecture would not be a good choice as in terms of speed there are faster backbone architectures available which can mostly overcome current backbone architecture's performance.

REFERENCES

[1] https://public.roboflow.com/object-detection/hard-hat-workers/2

[2] S. Ren, K. He, R. Girshick, and J. Sun. Faster R-CNN: Towards real-time object detection with region proposal networks. In NIPS, 2015.

[3] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision." In CVPR, 2016.

[4] https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/tf1_detection_zoo.md

[5] Zhengxia Zou, Zhenwei Shi, Member, IEEE, Yuhong Guo, and Jieping Ye,"Object Detection in 20 Years: A Survey", arXiv:1905.05055v

[6] https://www.tensorflow.org/resources/models-datasets

[7] https://towardsdatascience.com/how-to-train-a-tensorflow-face-object-detection-model-3599dcd0c26f