

# CHAPTER 5

## IMPLEMENTATION AND RESULTS

### 5.1. Experiment Setup

Pre-trained SSD MobileNet V2 320x320, obtained from the Tensorflow 2 detection model zoo, is the model used in this research. The model was trained on 27 May 2021, using a pro version of Google research collab with a RAM of 27.3 GB and a Tensorflow version of 2.5.0. Before this training, installing Tensorflow object detection API is needed to obtain files such as `model_main_tf2.py` and `generate_tfrecord.py`.

Next, the testing for the trained model was done using NVIDIA Jetson Nano Developer Kit - B01 New Revision that has RAM of 4 GB. The operating system for this device is Linux4Tegra which is based on Ubuntu 18.04. It has a GPU of NVIDIA Maxwell and 128 CUDA cores with a CPU of ARM A57, four cores, and 1.43 GHz. Next, is to install Tensorflow version 2.10.0 and Python version 3.7 before running the code.

The placement of the hardware in this research was done by placing it on a table, in a room with good lighting and a good Wifi network. The camera was placed on top of the table in the corner of the room and the picture is taken parallel to the camera level / eye-level

### 5.2. Collecting Data

The dataset has 853 images and it's saved into two different folders, one being training and the other for testing with a ratio of 80:20. The total number of images for training is 683 and for testing is 170, this division was done manually.

### 5.3. Pre-processing Data

After the dataset is ready, the next step is to preprocess it, which starts with image labeling. The image labeling was done by running a Python command "`python labelImg.py`" which will launch the LabelImg tool. Figure 5.1 shows the image labeling process. LabelImg is an open-source image labeling application that uses QT for its graphical interface and written in Python.



**Figure 5.1** Image Labeling for Dataset

Since the result of the labeling is a file with the extension .XML, it needs to be converted to .CSV by running “!python xml\_to\_csv.py --type train”. The function to convert in xml\_to\_csv.py code is written below :

```
1. def xml_to_csv(img_files, xml_files):
2.     xml_list = []
```

Lines 3 to 5 are to import the XML data by reading from a file.

```
3.     for i, xml_file in enumerate(xml_files):
4.         tree = ET.parse(xml_file)
5.         root = tree.getroot()
```

Then, lines 6 and 7 are to iterate over each node of the tree to obtain each element, its attribute, and all of its sub-elements in order to construct the dataframe. The elements consist of the filename, size, and object. The size element has sub-elements of width and height, whereas the object has sub-elements of name and bndbox which has other sub-elements of xmin, ymin, xmax, and ymax.

```
6.         for member in root.findall('object'):
7.             value = (root.find('filename').text,
                       int(root.find('size').find('width').text),
                       int(root.find('size').find('height').text),
                       member[0].text,
                       int(member.find("bndbox").find('xmin').text),
                       int(member.find("bndbox").find('ymin').text),
                       int(member.find("bndbox").find('xmax').text),
                       int(member.find("bndbox").find('ymax').text)
                       )
8.             xml_list.append(value)
```

A set of data from each iteration that can be thought as an observation in a pandas DataFrame will be returned in lines 9 to 11.

```
9.     column_name = ['filename', 'width', 'height', 'class', 'xmin',
                     'ymin', 'xmax', 'ymax']
10.    xml_df = pd.DataFrame(xml_list, columns=column_name)
```

```
11. return xml_df
```

The previous step's result is train\_labels.csv, and it can be seen on Figure 5.2 :

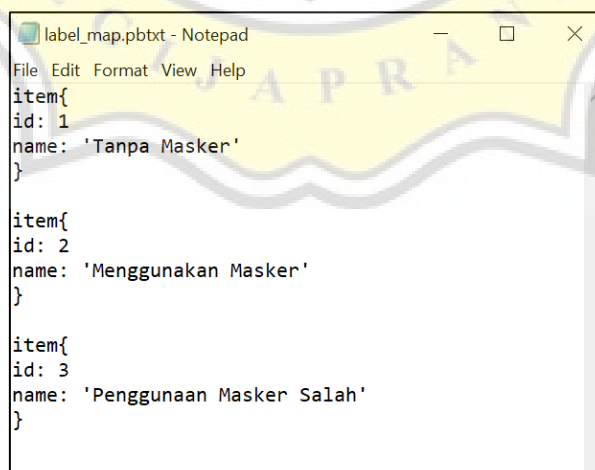
	A	B	C	D	E	F	G	H	I
1	filename	width	height	class	xmin	ymin	xmax	ymax	
2	makssskksss0.png	512	366	without_mask	79	105	109	142	
3	makssskksss0.png	512	366	with_mask	185	100	226	144	
4	makssskksss0.png	512	366	without_mask	325	90	360	141	
5	makssskksss1.png	400	156	with_mask	321	34	354	69	
6	makssskksss1.png	400	156	with_mask	224	38	261	73	
7	makssskksss1.png	400	156	with_mask	299	58	315	81	
8	makssskksss1.png	400	156	with_mask	143	74	174	115	
9	makssskksss1.png	400	156	with_mask	74	69	95	99	
10	makssskksss1.png	400	156	with_mask	191	67	221	93	
11	makssskksss1.png	400	156	with_mask	21	73	44	93	
12	makssskksss1.png	400	156	with_mask	369	70	398	99	
13	makssskksss1.png	400	156	without_mask	83	56	111	89	
14	makssskksss10.png	301	400	with_mask	98	267	194	383	
15	makssskksss100.png	400	226	with_mask	189	30	245	88	
16	makssskksss100.png	400	226	with_mask	387	54	400	75	
17	makssskksss100.png	400	226	with_mask	118	87	163	126	
18	makssskksss101.png	301	400	with_mask	48	284	164	400	

Figure 5.2 Sample Output of train\_labels.csv

Next, to recap the classes, the label map file is required. To generate a label map, run the command “python generate\_labelmap.py”. The code to generate\_labelmap.py is written below:

```
1. with open("label_map.pbt.txt", "w") as f:
2.     for idx, label in enumerate(df["class"].unique()):
3.         idx+=1
4.         f.write("item{\n")
5.         f.write("id: %d\n" % (idx))
6.         f.write("name: '" + label + "'\n")
7.         f.write("}\n\n")
8. print("DONE")
```

The previous step's result can be seen on Figure 5.3. It is saved as label\_map.pbt.txt file.



```
label_map.pbt.txt - Notepad
File Edit Format View Help
item{
id: 1
name: 'Tanpa Masker'
}

item{
id: 2
name: 'Menggunakan Masker'
}

item{
id: 3
name: 'Penggunaan Masker Salah'
}
```

Figure 5.3 Content of label\_map.pbt.txt

The last step for the preprocessing is to create the TFRecord using generate\_tfrecord.py which is provided by Tensorflow. The command “!python generate\_tfrecord.py --csv\_input train\_labels.csv --image\_dir train --labelmap\_dir label\_map.pbtxt --output\_path train.tfrecord” is used to create TFRecord files.

On the code, an adjustment needs to be made, which is the function of class\_text\_to\_int on lines 1 to 9. It should reflect the label map, since the label map has three classes, the function should also have three classes with matching return value and id on the previous label map.

```
1. def class_text_to_int(row_label):
2.     if row_label == 'without_mask':
3.         return 1
4.     elif row_label == 'with_mask':
5.         return 2
6.     elif row_label == 'mask_wearred_incorrect':
7.         return 3
8.     else:
9.         return 0
```

#### 5.4. Training Model

After the dataset is ready, the next step is to prepare the model that will be trained by doing some basic configuration to the existing hyperparameters config to the needs of this research and saving the file in “.config” format. The configuration for this model was made by changing the number of classes (num\_classes), the batch size (batch\_size), training steps, and set the path to the downloaded model checkpoint, the checkpoint type, the path to label\_map, TFRecord file with the training and testing data. The hyperparameter used for the batch size was 16 and the training steps of 120000 with a learning rate of 0.008. Training using other hyperparameters has been carried out, but due to hardware limitations, it is not possible to do another testing, so the research continues to use these hyperparameters. The following setup is for the "SSD MobileNet V2 320x320" model:

On line 3, the num\_classes is set to 3 because the dataset has 3 classes, with\_mask, mask\_wearred\_incorrect, and without\_mask.

```
1. model {
2.     ssd {
3.         num_classes: 3
```

On lines 4 to 9, there's a function to resize the image and it is set to 300 so it can reduce the training time.

```
4.     image_resizer {
5.         fixed_shape_resizer {
6.             height: 300
7.             width: 300
```

```
8.     }
9.     }
```

Next is to configure the `train_config`. On line 10, the `batch_size` (integer, must be divisible by 2) dictate the number of images to be fed into memory while training. Since the GPU has 27.3 GB available, the `batch_size` is set to 16. The number of training steps is also adjusted to 120000 on lines 26 and 35 with warmup steps of 1000 on line 28 to improve the model performance.

```
10. train_config {
11.   batch_size: 16
12.   data_augmentation_options {
13.     random_horizontal_flip {
14.     }
15.   }
16.   data_augmentation_options {
17.     ssd_random_crop {
18.     }
19.   }
20.   sync_replicas: true
21.   optimizer {
22.     momentum_optimizer {
23.       learning_rate {
24.         cosine_decay_learning_rate {
25.           learning_rate_base: 0.008
26.           total_steps: 120000
27.           warmup_learning_rate: 0.0001
28.           warmup_steps: 1000
29.         }
30.       }
31.       momentum_optimizer_value: 0.9
32.     }
33.     use_moving_average: false
34.   }
35.   num_steps: 120000
```

The path of the SSD model is pointed with the `fine_tune_checkpoint`. This ensures that the training of the model is not from scratch.

```
36.   fine_tune_checkpoint: "ssd_mobilenet_v2_320x320_coco17_tpu-8/cp/ckpt-0"
```

Its type (`fine_tune_checkpoint_type`) is set to “detection” from “classification” since this training is to train the model for object detection.

```
37.   fine_tune_checkpoint_type: "detection"
38. }
```

For training data, the train input reader must point to the label map path and the path to the TFRecords.

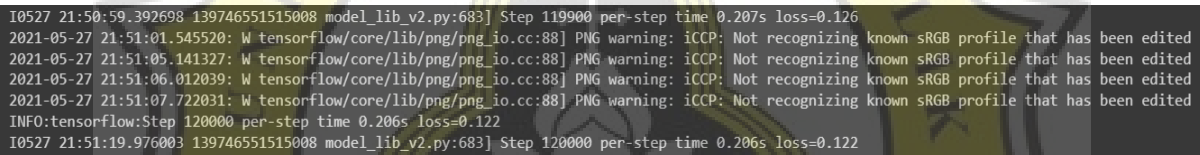
```
39. train_input_reader {
40.   label_map_path: "/content/dataset/label_map.pbtxt"
41.   tf_record_input_reader {
42.     input_path: "/content/dataset/train.record"
43.   }
```

```
44. }
```

The `eval_input_reader` is similar to `train_input_reader`, but for the test data for testing the model.

```
45. eval_input_reader {
46.   label_map_path: "/content/dataset/label_map.pbtxt"
47.   shuffle: false
48.   num_epochs: 1
49.   tf_record_input_reader {
50.     input_path: "/content/dataset/test.record"
51.   }
52. }
```

Other than the above, the hyperparameter used for the training is the same as the one that is provided by Tensorflow. To start training the model for this research use `model_main_tf2.py` which is provided by Tensorflow, so the command “!python /model\_main\_tf2.py --alsologtostderr --model\_dir=dirOfModel --pipeline\_config\_path=file.config” is run. Figure 5.4 displays the result of this training phase, which is a log of train performance:



```
I0527 21:50:59.392698 139746551515008 model_lib_v2.py:683] Step 119900 per-step time 0.207s loss=0.126
2021-05-27 21:51:01.545520: W tensorflow/core/lib/png/png_io.cc:88] PNG warning: iCCP: Not recognizing known sRGB profile that has been edited
2021-05-27 21:51:05.141327: W tensorflow/core/lib/png/png_io.cc:88] PNG warning: iCCP: Not recognizing known sRGB profile that has been edited
2021-05-27 21:51:06.012039: W tensorflow/core/lib/png/png_io.cc:88] PNG warning: iCCP: Not recognizing known sRGB profile that has been edited
2021-05-27 21:51:07.722031: W tensorflow/core/lib/png/png_io.cc:88] PNG warning: iCCP: Not recognizing known sRGB profile that has been edited
INFO:tensorflow:Step 120000 per-step time 0.206s loss=0.122
I0527 21:51:19.976003 139746551515008 model_lib_v2.py:683] Step 120000 per-step time 0.206s loss=0.122
```

**Figure 5.4** The Output of Training Steps

## 5.5. Hardware

The camera is using camera module IMX219 because it's mainly designed for the NVIDIA Jetson Nano. For the SD card, the SanDisk Extreme Pro 64GB MicroSD is used because it is intended for SD cameras capable of recording Full HD, 3D, and 4K video with raw and burst photos with a capacity of 64GB, that can handle 170MB/s read speeds and write rates of up to 90 MB/s. The device also utilizes the TL-WN722N USB Wifi Adapter, which enables the hardware to establish wireless network access on the computer and to access a high-speed Internet connection, with wireless speeds of up to 150Mbps.

When the model is ready, the next step is to construct the hardware based on the hardware design in the previous chapter. The result of the construction looks can be seen on Figure 5.5.



**Figure 5.5** Hardware Result

This device is placed on top of the table so it is parallel with the object for better detection purposes, since the parallel position gives clearer images.

## 5.6. Implementation

After the model is trained and exported, as well as the hardware is ready, the next step is to implement the trained model into the system that is developed using Python language. The code for the system is already adjusted to the existing hardware and is written below :

Lines 1 to 9 are related to a database where lines 1 to 6 are to connect and lines 8 and 9 are to input the data into a database.

```

1. db = mysql.connector.connect (
2.   host="localhost",
3.   user="root",
4.   password="",
5.   database="db_pkm2"
6. )
7. eksekusiDb = db.cursor()
8. sql = "INSERT INTO tblDataPelanggar (time, pelanggar_masker, \
9. pelanggar_jarak, bukti_ss) VALUES (%s, %s, %s, %s)"

```

Lines 10 to 13 are related to load label map.

```

10. PATH_TO_LABELS = 'label_map.pbtxt'
11. category_index =
12. label_map_util.create_category_index_from_labelmap(PATH_TO_LABELS, \
13. use_display_name=True)

```

And line 14 is to load the trained model.

```

14. detection_model = tf.saved_model.load("inference_graph/saved_model")

```

Line 15 to 18 are the function to detect the center of the bounding box of the object detected.

```

15. def deteksi_center(ymin, xmin, ymax, xmax, w, h):
16.     cx = ((xmin+xmax)/2)*w

```

```

17.     cy = ((ymin+ymax)/2)*h
18.     return int(cx),int(cy)

```

Line 19 to 40 are the function for calling the model and cleaning up the outputs. Line 21 converts the input to a tensor since it must be a tensor, and the model expects a batch of images, so an axis is added on line 22, and the inference is run on line 24.

```

19. def run_inference_for_single_image(model, image):
20.     image = np.asarray(image)
21.     input_tensor = tf.convert_to_tensor(image)
22.     input_tensor = input_tensor[tf.newaxis,...]
23.     model_fn = model.signatures['serving_default']
24.     output_dict = model_fn(input_tensor)

```

All outputs are batch tensors, so on lines 25 to 28, it is transformed to numpy arrays, and the batch dimension is eliminated by taking index[0]. On line 29, the detection\_classes should be ints.

```

25.     num_detections = int(output_dict.pop('num_detections'))
26.     output_dict = {key:value[0, :num_detections].numpy()
27.                    for key,value in output_dict.items()}
28.     output_dict['num_detections'] = num_detections
29.     output_dict['detection_classes'] = \
output_dict['detection_classes'].astype(np.int64)

```

Lines 30 to 39 are to handle models with masks. The box mask is reframed to the image size on lines 31 to 39.

```

30.     if 'detection_masks' in output_dict:
31.         detection_masks_reframed =
32.             utils_ops.reframe_box_masks_to_image_masks( \
33.                 output_dict['detection_masks'], \
34.                 output_dict['detection_boxes'], \
35.                 image.shape[0], image.shape[1])
36.         detection_masks_reframed = \
tf.cast(detection_masks_reframed>= \
37.         0.5, tf.uint8)
38.         output_dict['detection_masks_reframed'] = \
39.             detection_masks_reframed.numpy()
40.     return output_dict

```

Lines 41 to 78 are the procedures to execute on each test image and display the outcomes. First, the variables to store the data of the violation with the default of 0 is prepared on lines 42 and 43.

```

41. def show_inference(model, image_np):
42.     totalMelanggarJarak = 0
43.     totalMelanggarMasker = 0
44.     output_dict = run_inference_for_single_image(model, image_np)

```

And the result of the detection is visualized on lines 45 to 54.

```

45.     hasil = vis_util.visualize_boxes_and_labels_on_image_array(
46.         image_np,
47.         output_dict['detection_boxes'],
48.         output_dict['detection_classes'],

```



```

49.         output_dict['detection_scores'],
50.         category_index,
51.         instance_masks=output_dict.get('detection_masks_reframed', \
52.         None),
53.         use_normalized_coordinates=True,
54.         line_thickness=8)

```

Lines 55 to 78 are to detect the distance violation.

```

55.     box_05 = \
56.     output_dict['detection_boxes'] \
57.     [output_dict['detection_scores']>=0.5]
58.     h, w, c = hasil.shape
59.     rekapKoordinat = []

```

Lines 60 to 63 are to get the coordinate of the object and lines 63 to 66 are to give the bounding box, a dot (circle of red color) in the middle to mark the center.

```

60.     for index in range(len(box_05)):
61.         ymin, xmin, ymax, xmax = box_05[index]
62.         rekapKoordinat.append(deteksi_center(ymin, xmin, ymax, \
63.         xmax, w, h))
64.     for i in range(len(rekapKoordinat)):
65.         hasil = cv2.circle(hasil, rekapKoordinat[i], 5, (255, 0, 0), -
66.         5)

```

Next, lines 67 to 78 are to get the distance between objects and calculate it. On lines 72 to 74, the Euclidean Distance formula is used to compute the distance. If the distance is less than the threshold on line 75, the value distance violator is added on line 78.

```

67.     rekapJarak = {}
68.     if len(rekapKoordinat)>1:
69.         for i in range(len(rekapKoordinat)-1):
70.             rekapJarak[i] = {}
71.             for j in range(i+1, len(rekapKoordinat)):
72.                 rekapJarak[i][j] = \
73.                 (np.linalg.norm(np.array(rekapKoordinat[i]) - \
74.                 np.array(rekapKoordinat[j])))
75.                 if rekapJarak[i][j] < 300:
76.                     hasil = cv2.line(hasil, rekapKoordinat[i], \
77.                     rekapKoordinat[j], (255, 0, 0), 2)
78.                     totalMelanggarJarak+=1

```

And on lines 79 to 83, if there's a violation, the buzzer will make a sound. Line 83 is to return the value of the violator.

```

79.     if totalMelanggarJarak!=0 or totalMelanggarMasker!=0:
80.         mixer.init()
81.         mixer.music.load('alert.ogg')
82.         mixer.music.play()
83.     return(hasil, totalMelanggarJarak, totalMelanggarMasker)

```

Then, line 84 is to create a video capture object, which would help stream or display the video, and line 85 is to get the current date and time. Line 86 is to loop for 24 hours. Line 87 is to read the video capture and then it's converted to RGB on line 88 so it can be an input

for the model on line 89. Line 90 is to store the image result and it's converted back to RGB on line 91.

```

84. cap = cv2.VideoCapture(2)
85. terakhir = datetime.datetime.now()
86. while 1:
87.     _, img = cap.read()
88.     img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
89.     inferencehasil = show_inference(detection_model, img)
90.     final_img = inferencehasil[0]
91.     final_img = cv2.cvtColor(final_img, cv2.COLOR_RGB2BGR)

```

Lines 92 to 101 are to show the total of the distance violation. The color of the text depends on line 92, if the violation is more than 1, then the color will follow lines 93 to 95.

```

92.     if inferencehasil[1] != 0:
93.         final_img = cv2.putText(final_img, "Total Kasus Jarak = " + \
94.             str(inferencehasil[1]), org=(20,460), fontFace= \
95.             cv2.FONT_HERSHEY_SIMPLEX, fontScale=1, color=(0,0,255), \
96.             thickness=3, lineType=cv2.LINE_AA)
97.     else:
98.         final_img = cv2.putText(final_img, "Total Kasus Jarak = " + \
99.             str(inferencehasil[1]), org=(20,460), fontFace= \
100.            cv2.FONT_HERSHEY_SIMPLEX, fontScale=1, color=(0,255,0), \
101.            thickness=3, lineType=cv2.LINE_AA)

```

Then on lines 102 to 111, the data is stored in the database.

```

102.         valueku = \
103.             (datetime.datetime.now(), inferencehasil[2],
104.             inferencehasil[1], respon["data"]["display_url"])
105.         eksekusiDb.execute(sql, valueku)
106.         db.commit()
107.     else:
108.         valueku = (datetime.datetime.now(), inferencehasil[2], \
109.             inferencehasil[1], "-")
110.         eksekusiDb.execute(sql, valueku)
111.         db.commit()

```

## 5.7. Evaluation

The test was carried out using ten images given in front of the camera. Some images were taken from Google pictures and some were from camera video. The test was carried out on 16 October 2022 at 8 p.m. that took place on Anak Panah Kopi, Gajahmada Street Number 91. Each image was taken at eye level (directly in front of the camera) in a lighting room. Table 5.1 shows the results from the testing.

**Table 5.1.** Testing Result

No.	Actual		Predicted		TP	TN	FP	FN
	Violator	Non-Violator	Violator	Non-Violator				
1	0	2	0	2	0	2	0	0
2	2	0	2	0	2	0	0	0

3	6	0	5	1	5	0	0	1
4	2	0	2	0	2	0	0	0
5	2	0	2	0	2	0	0	0
6	2	0	2	0	2	0	0	0
7	3	0	3	0	3	0	0	0
8	2	0	2	0	2	0	0	0
9	2	0	0	2	0	0	0	2
10	2	0	2	0	2	0	0	0

From the tests carried out in Table 5.1, it was found that the system accuracy value is 88%, with a precision value of 100%, and a recall value of 87%. These results were calculated according to the number of objects detected between actual and detected by the MobileNet V2 320x320 SSD using a confusion matrix.



**Figure 5.6** Detection Using Images (a) No Violator (b) With Violators (c) With Violators but One is Undetected



(a)

(b)

**Figure 5.7** Detection Using Camera Video (a) Violators but One Undetected (b) Violators Detected

Figure 5.6 shows the detection results using photos and Figure 5.7 shows the detection results using camera video (real-time). On Figure 5.6(a) the detection is shown with no violator, Figure 5.6(b) shows the detection with violator, and Figure 5.6(c) shows the undetected violator. Figure 5.7(a) shows the detection result with an undetected violator and Figure 5.7(b) shows the detection with fully detected violators.

In Figure 5.6(c) there is a detection error due to the person's face being taken in side profiles, which can sometimes be detected and sometimes not by the camera, and another error in Figure 5.7(a) due to the backlighting. This shows that the object of detection can't be detected when there is a backlight and for better detection, it is best when the person's face is facing forward.

## 5.8. Discussion

Taking the experiment results in Figure 5.6 and Figure 5.7(b), the object was detected with a fairly high accuracy of 88%, this indicates that the SSD MobileNet V2 320x320 method is capable of detecting objects within a room lighting. In Figure 5.7(a), it can be seen that not all objects can be detected, because the backlighting made the object became less clear for detection. Another limitation of this research is only detecting objects that appear directly parallel in front of the camera because of the short cable connection between the device and the laptop and the limited space to put the camera.