# ASSIGNMENT 3 / MINI PROJECT

# VISUAL RECOGNITION(VR)

---

Name of Group Members:-
HUSSNAIN ASHRAF (MT2021055)
KUNAL (MT2021067)

# REPORT OF ASSIGNMENT – 3a

Implementation of AlexNet Convolutional Neural Network(CNN) architecture on CIFAR -10 using TensorFlow and Keras.

**AlexNet Implementation and Discussion**:-

AlexNet is the name of a convolutional neural network which has had a large impact on the field of machine learning, specifically in the application of deep learning to machine vision.

It consisted of 11×11, 5×5,3×3, convolutions, max pooling, dropout, data augmentation, ReLU activations, SGD with momentum. It attached ReLU activations after every convolutional and fully-connected layer.

**Some Key Points of AlexNet:-**

**1.** Relu activation function is used instead of Tanh to add non-linearity. It accelerates the speed by 6 times at the same accuracy.

**2.** Use dropout instead of regularization to deal with overfitting. However, the training time is doubled with the dropout rate of 0.5.

**3.** Overlap pooling to reduce the size of the network. It reduces the top-1 and top-5 error rates by 0.4% and 0.3%, respectively.

**Dataset:-**

We are using the CIFAR-10 dataset which contains 60,000 color images, each with dimensions *32x32px*. The content of the images within the dataset is sampled from 10 classes.

These 10 classes are depicted as follows:-

1. **Airplane**
2. **automobile**
3. **Bird**
4. **Cat**
5. **Deer**
6. **Dog**
7. **Frog**
8. **Horse**
9. **Ship**
10. **Truck**

The CIFAR dataset is partitioned into 50,000 training data and 10,000 test data by default. The last partition of the dataset we require is the validation data.so here we have taken some small amount of training ,validation and test data because our training time was taking so much time.

The validation data is obtained by taking the last some amount of images within the training data.

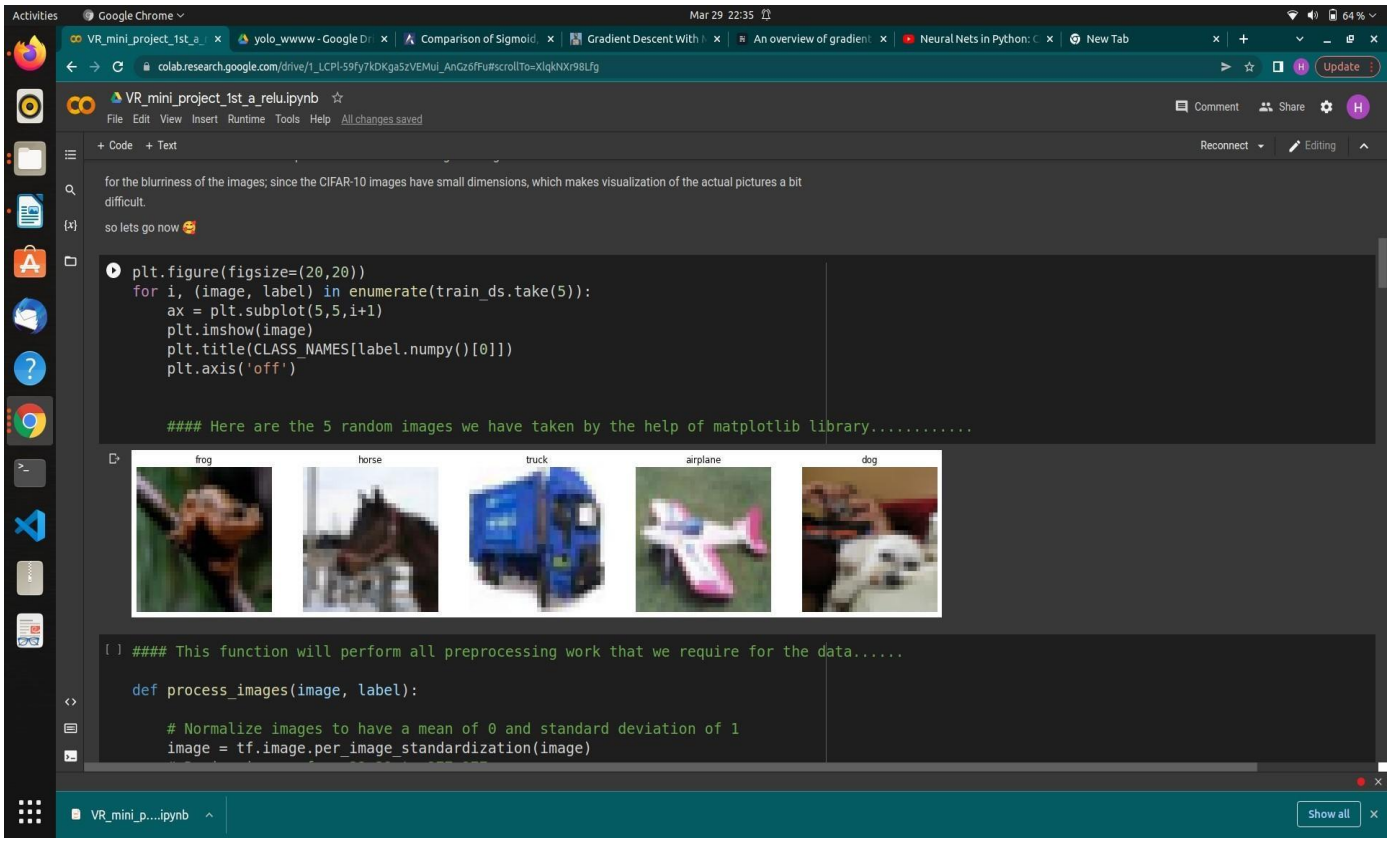
## Preprocessing:-

Preprocessing in any machine learning is associated with the transformation of data from one form to another.

Usually, preprocessing is conducted to ensure the data utilized is within an appropriate format.

First, let's visualize the images within the CIFAR-10 dataset.The code picture below uses the Matplotlib library to present the pixel information of the data from five training images into actual images. There is also an **indicator of the class** each **depicted content within the images belongs** to.

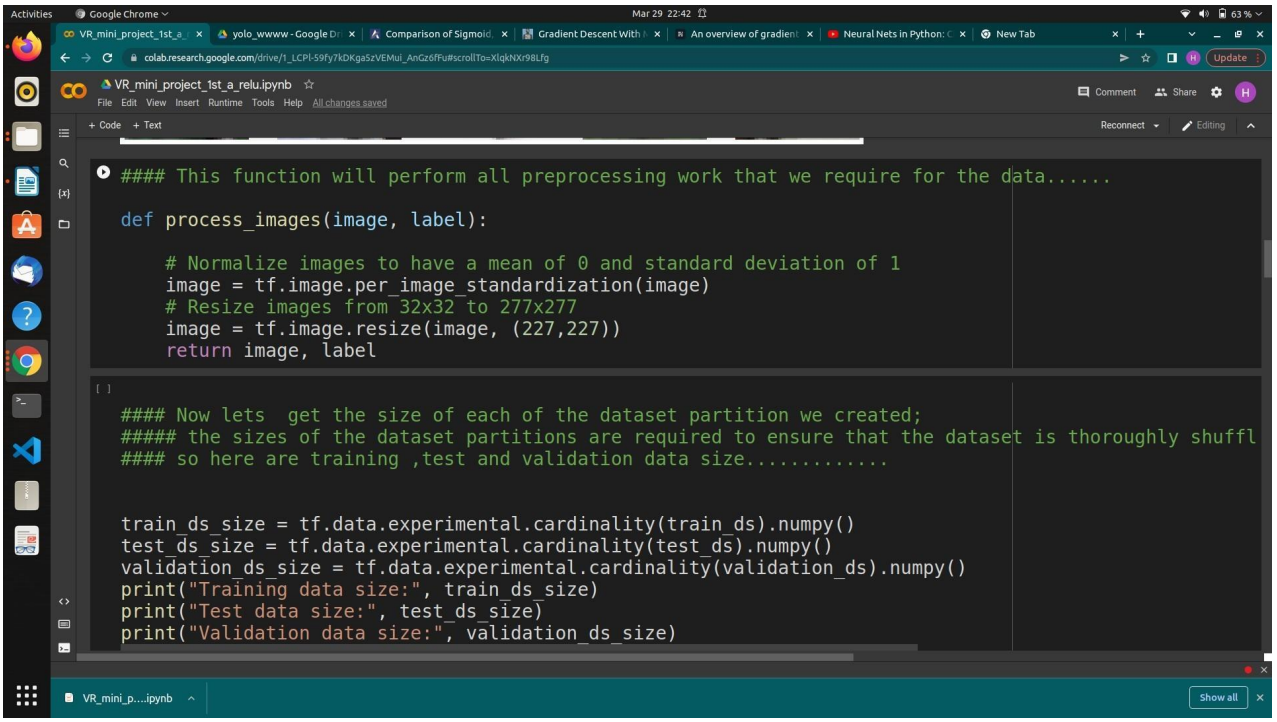
Here the blurriness of the images are coming in the CIFAR-10 images because of having small dimensions, which makes visualization of the actual pictures a bit difficult.



The primary preprocessing transformations which we have done on the data are:

- **Normalizing and standardizing the images.**

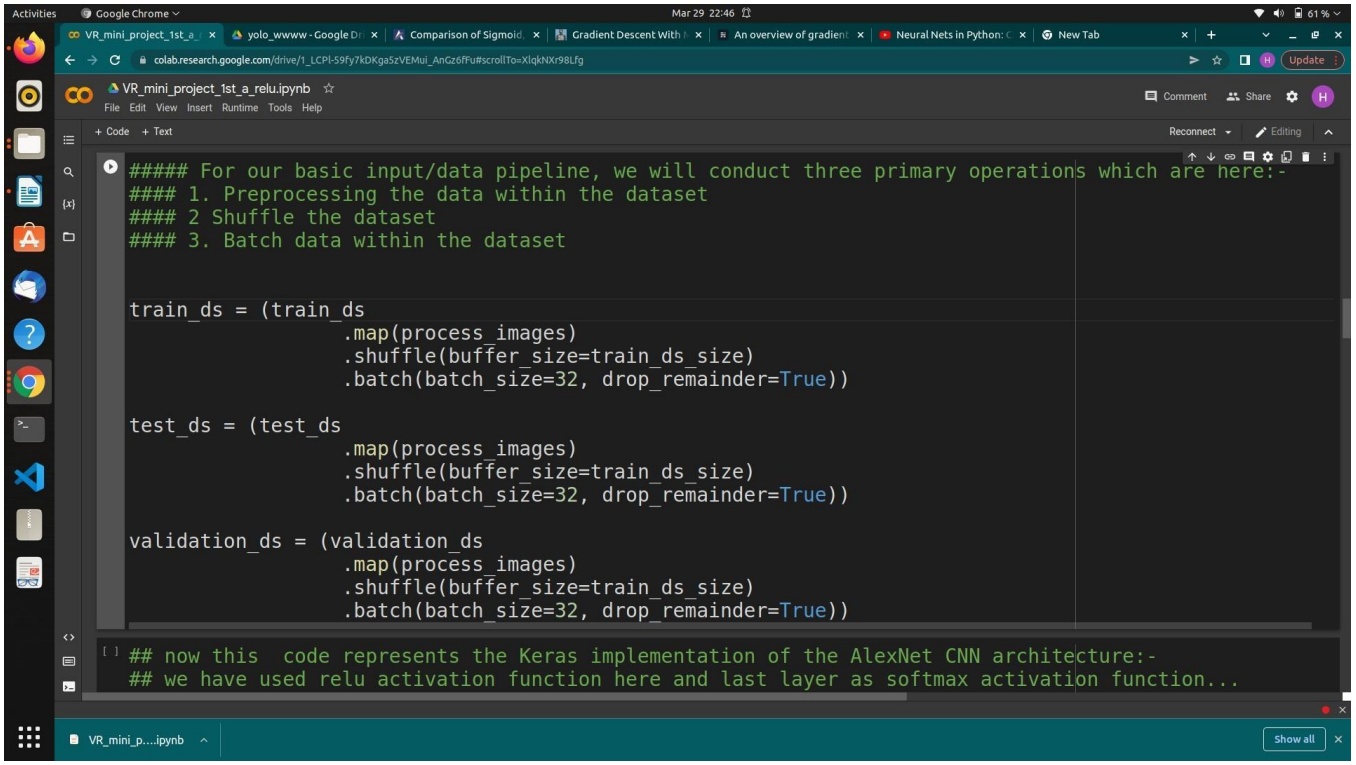
- **Resizing of the images from *32x32* to *227x227*. The AlexNet network input expects a 227x227 image.**




For our basic input/data , we will do three primary operations.

1. Preprocessing the data within the dataset
2. Shuffle the dataset
3. Batch data within the dataset

These what we have done after preprocessing that is shuffling and batching the data are attached in the below picture.




## Model Implementation:-

Here are the types of layers the **AlexNet CNN** architecture is composed of, along with a brief description:-

**Convolutional layer**: A convolution is a mathematical term that describes a dot product multiplication between two sets of elements. Within deep learning the convolution operation acts on the filters/kernels and image data array within the convolutional layer. Therefore a convolutional layer is simply a layer the houses the convolution operation that occurs between the filters and the images passed through a convolutional neural network.

**Batch Normalisation layer**: Batch Normalization is a technique that mitigates the effect of unstable *gradients*

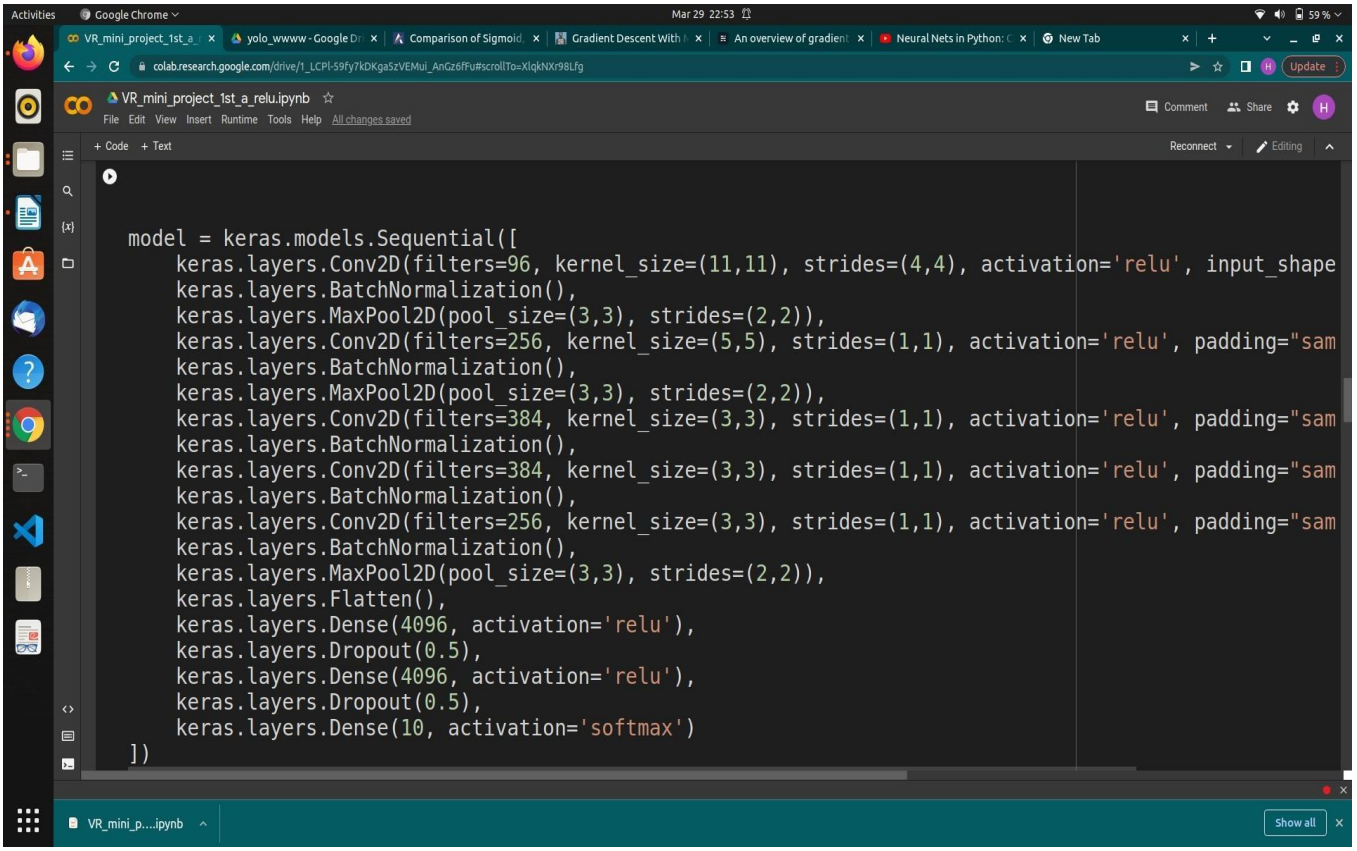
within a neural network through the introduction of an additional layer that performs operations on the inputs from the previous layer. The operations standardize and normalize the input values, after that the input values are transformed through scaling and shifting operations.

**MaxPooling layer**: Max pooling is a variant of sub- sampling where the maximum pixel value of pixels that fall within the receptive field of a unit within a sub- sampling layer is taken as the output. The max-pooling operation below has a window of 2x2 and slides across the input data, outputting an average of the pixels within the receptive field of the kernel.

**Flatten layer**: Takes an input shape and flattens the input image data into a one-dimensional array.

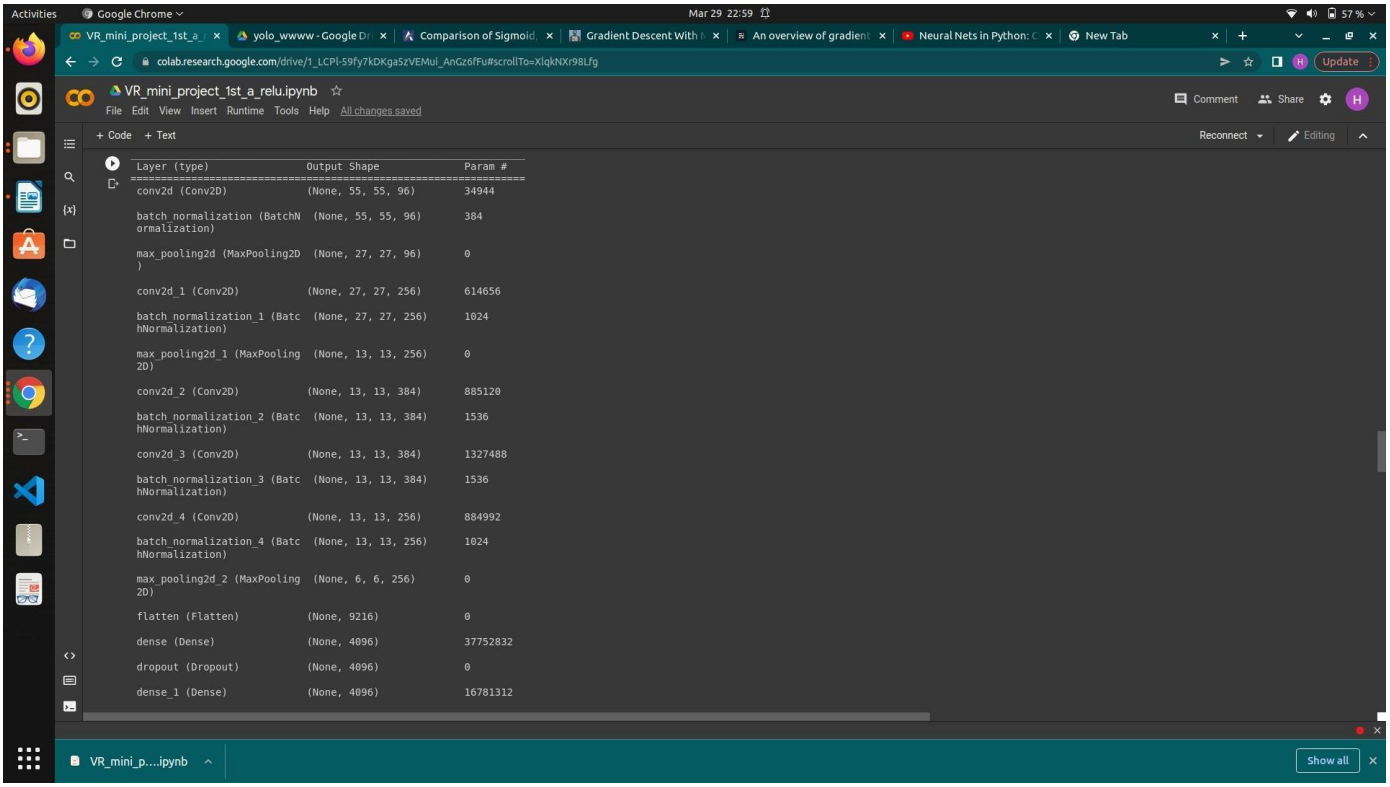
**Dense Layer**: A dense layer has an embedded number of arbitrary units/neurons within. Each neuron is a perceptron.

The code picture represents the Keras implementation of the AlexNet CNN architecture:-



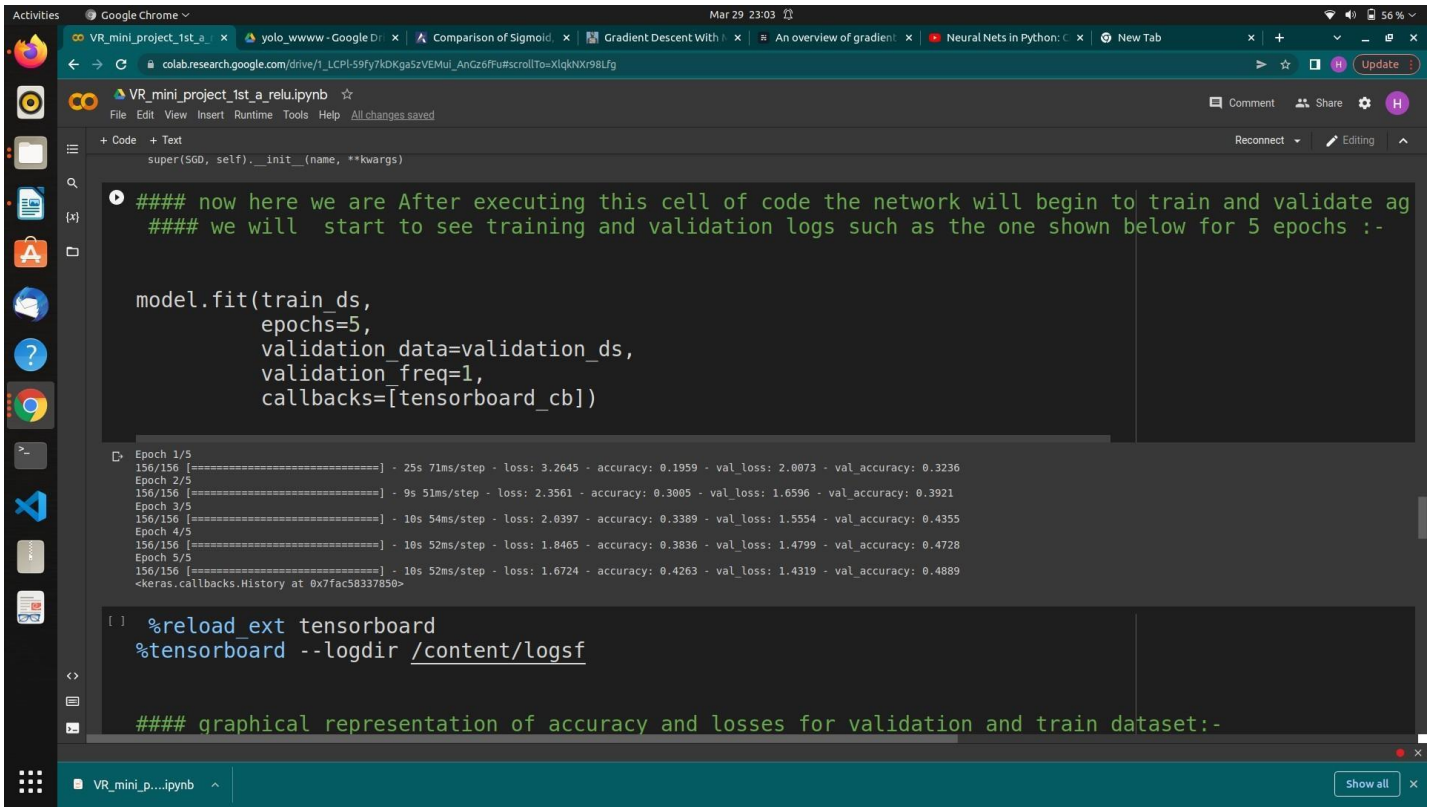
```
model = keras.models.Sequential([
    keras.layers.Conv2D(filters=96, kernel_size=(11,11), strides=(4,4), activation='relu', input_shape
    keras.layers.BatchNormalization(),
    keras.layers.MaxPool2D(pool_size=(3,3), strides=(2,2)),
    keras.layers.Conv2D(filters=256, kernel_size=(5,5), strides=(1,1), activation='relu', padding="sam
    keras.layers.BatchNormalization(),
    keras.layers.MaxPool2D(pool_size=(3,3), strides=(2,2)),
    keras.layers.Conv2D(filters=384, kernel_size=(3,3), strides=(1,1), activation='relu', padding="sam
    keras.layers.BatchNormalization(),
    keras.layers.Conv2D(filters=384, kernel_size=(3,3), strides=(1,1), activation='relu', padding="sam
    keras.layers.BatchNormalization(),
    keras.layers.Conv2D(filters=256, kernel_size=(3,3), strides=(1,1), activation='relu', padding="sam
    keras.layers.BatchNormalization(),
    keras.layers.MaxPool2D(pool_size=(3,3), strides=(2,2)),
    keras.layers.Flatten(),
    keras.layers.Dense(4096, activation='relu'),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(4096, activation='relu'),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(10, activation='softmax')
])
```

Here are the model summary visualization for more insight about all the layers of AlexNet CNN architecture:-



```
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 55, 55, 96)        34944

batch_normalization (BatchN  (None, 55, 55, 96)        384
ormalization)

max_pooling2d (MaxPooling2D  (None, 27, 27, 96)        0
)

conv2d_1 (Conv2D)            (None, 27, 27, 256)       614656

batch_normalization_1 (Batc  (None, 27, 27, 256)       1024
hNormalization)

max_pooling2d_1 (MaxPooling  (None, 13, 13, 256)       0
2D)

conv2d_2 (Conv2D)            (None, 13, 13, 384)       885120

batch_normalization_2 (Batc  (None, 13, 13, 384)       1536
hNormalization)

conv2d_3 (Conv2D)            (None, 13, 13, 384)       1327488

batch_normalization_3 (Batc  (None, 13, 13, 384)       1536
hNormalization)

conv2d_4 (Conv2D)            (None, 13, 13, 256)       884992

batch_normalization_4 (Batc  (None, 13, 13, 256)       1024
hNormalization)

max_pooling2d_2 (MaxPooling  (None, 6, 6, 256)         0
2D)

flatten (Flatten)            (None, 9216)              0

dense (Dense)                (None, 4096)              37752832

dropout (Dropout)            (None, 4096)              0

dense_1 (Dense)              (None, 4096)              16781312
```

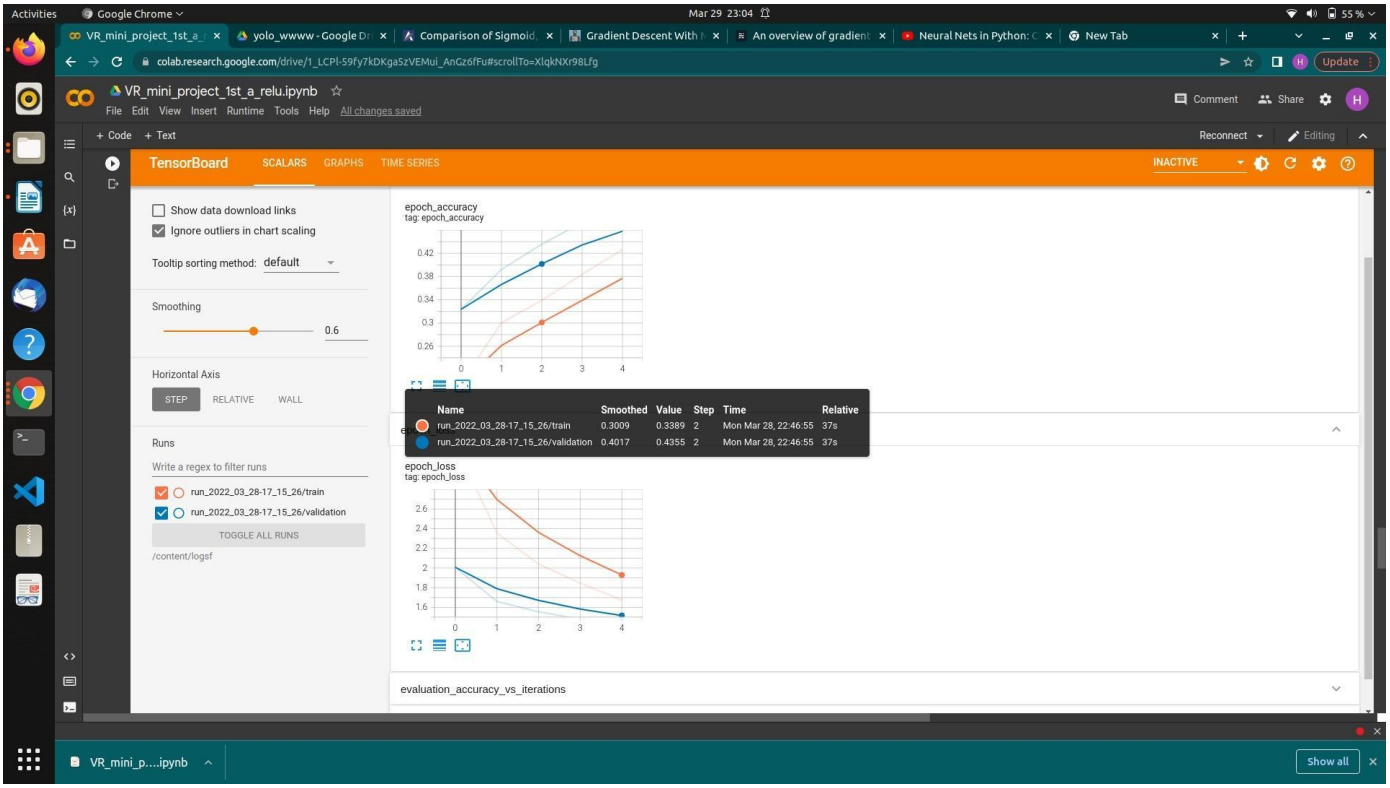
Now here we are **training the AlexNet on CIFAR-10 dataset for 5 epochs** only we haven't taken much epochs as it was taking more time and also GPU access of colab is limited:-



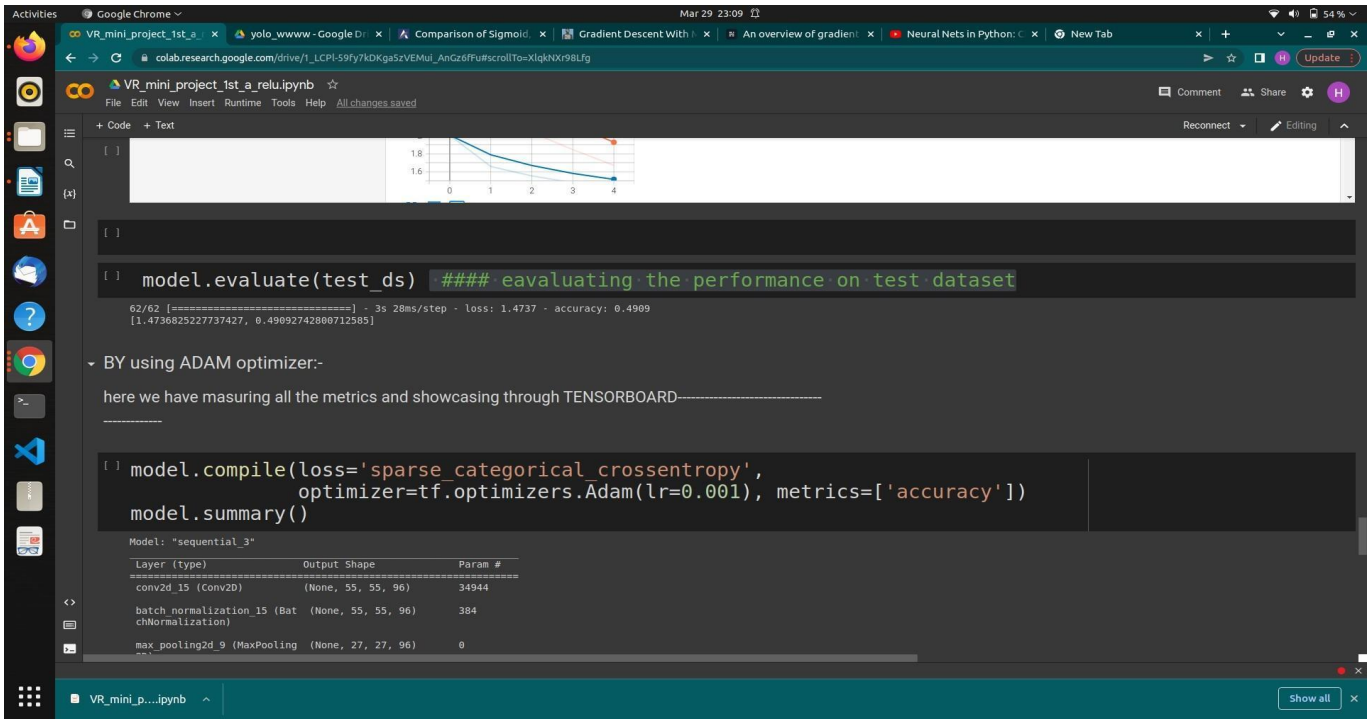

Here we have used **ReLu as activation function and SGD optimization algorithm and using Learning rate =0.001** for calculating metrics that is loss and accuracy.

Now here we are using **TENSORBOARD** which is a tool that provides a suite of visualization and monitoring the metrics on

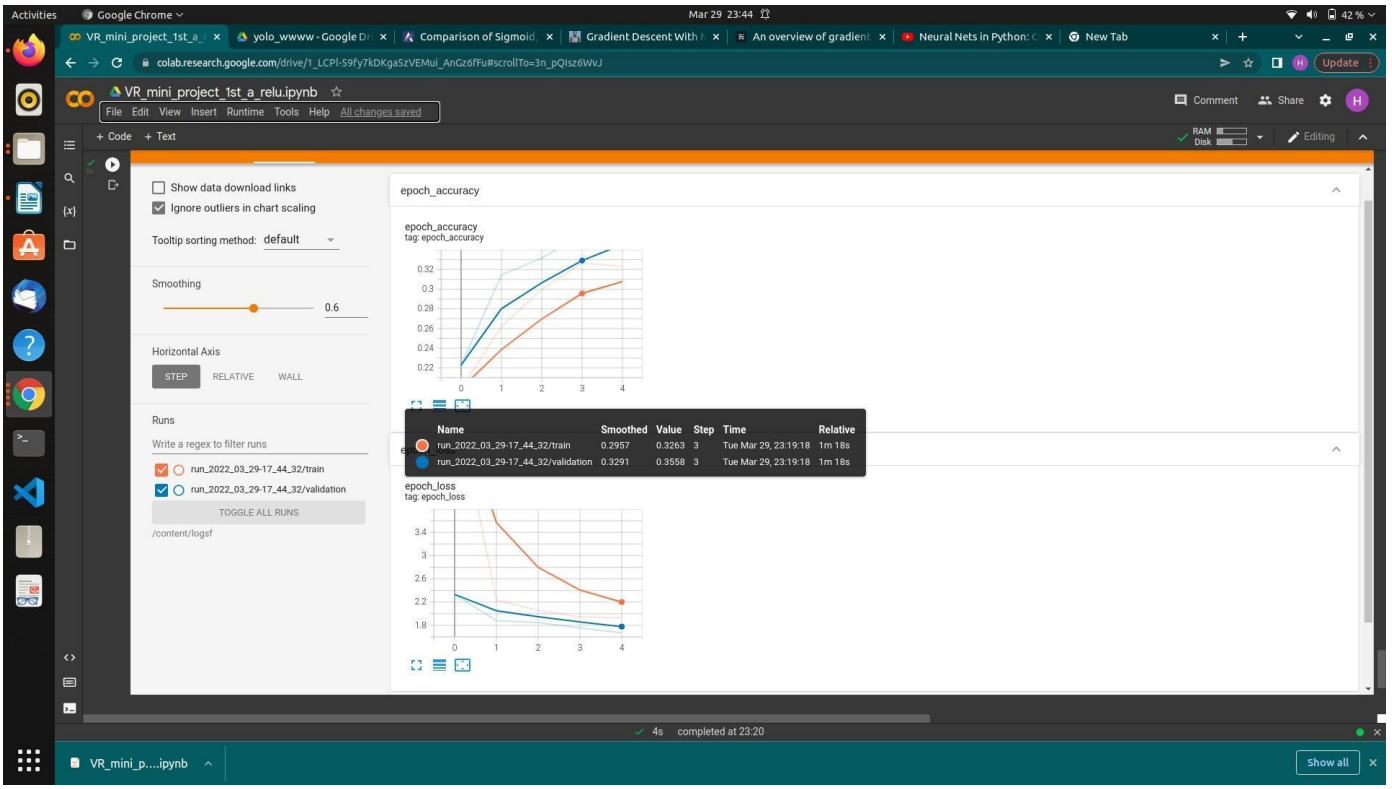
train and validation datasets. For the work we'll be utilizing TensorBoard to monitor the progress of the training of the model on the datasets.

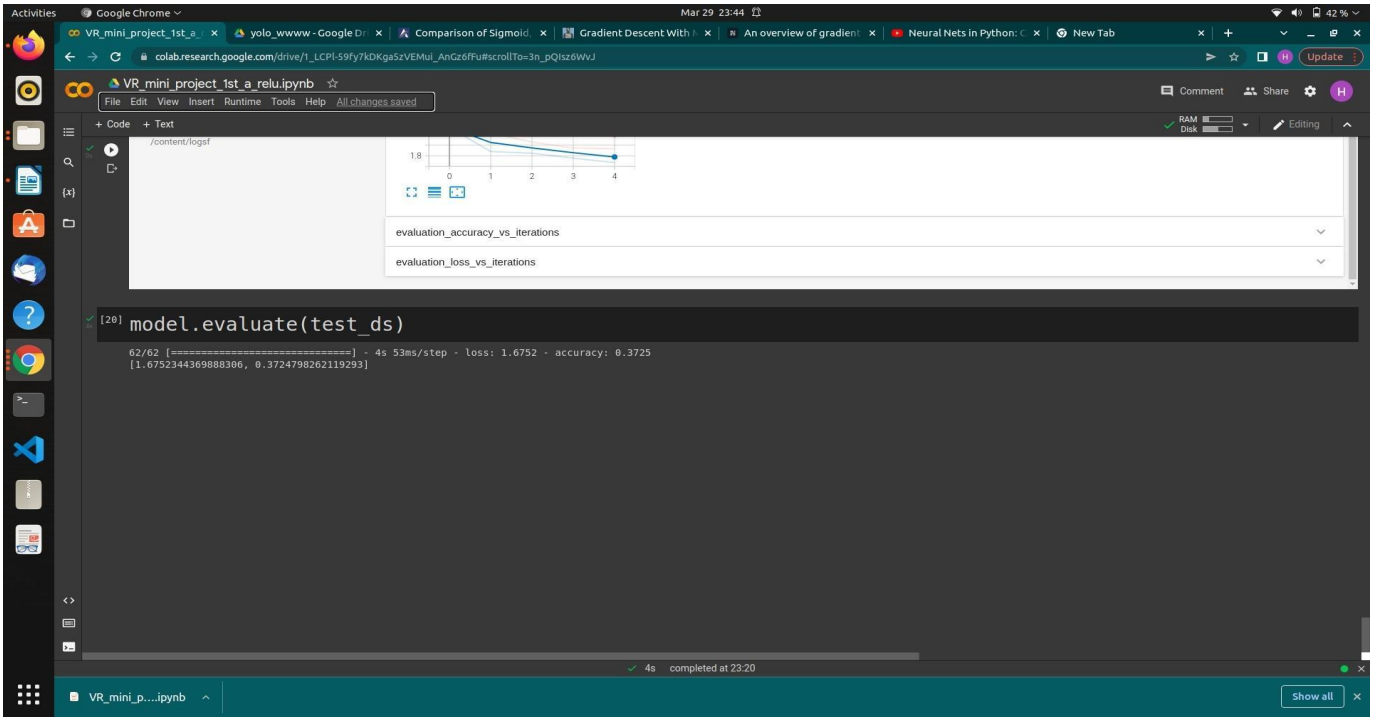



## Now we are evaluating our trained model on test datasets:-

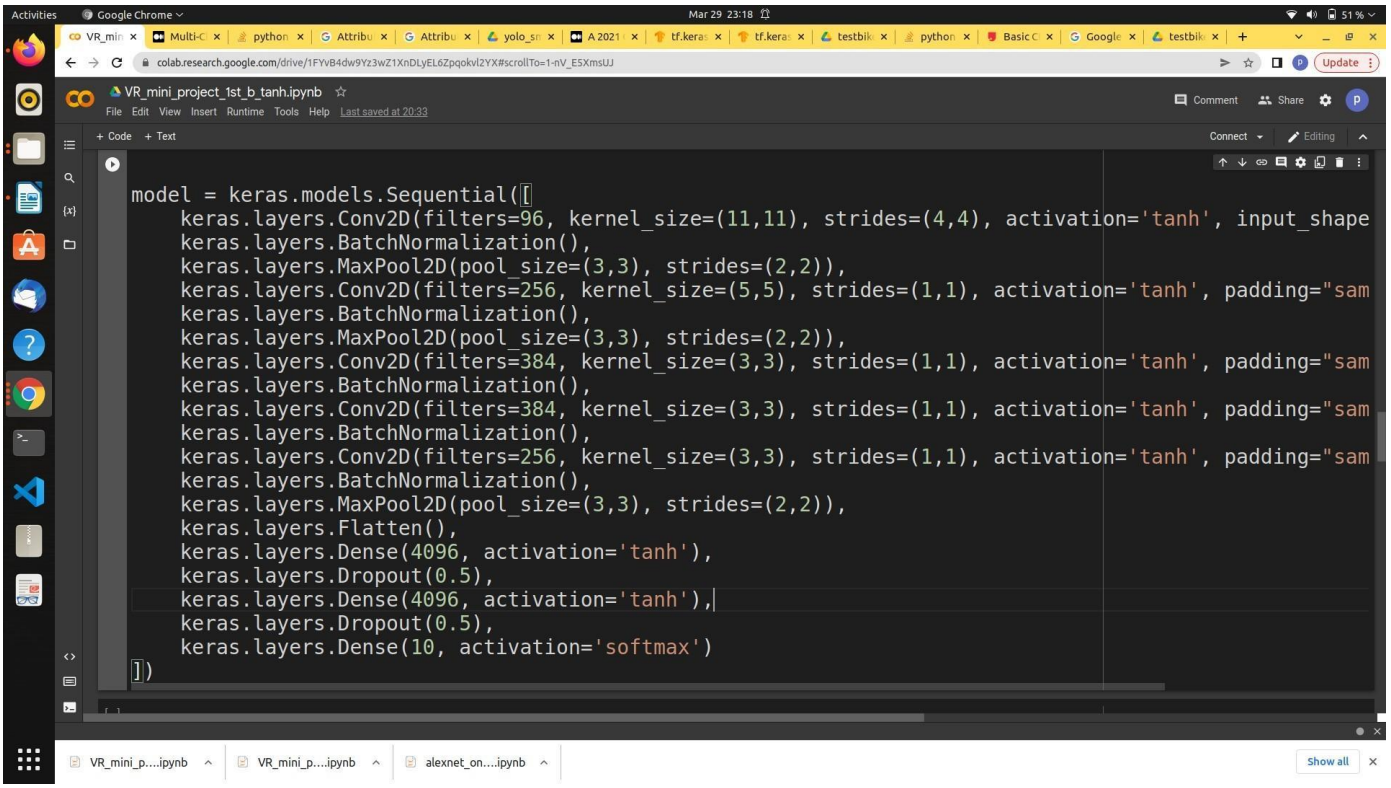


Now metrics visualization and prediction of loss and accuracy on test dataset **by using Adam optimizer and with same learning rate as 0.001:-**

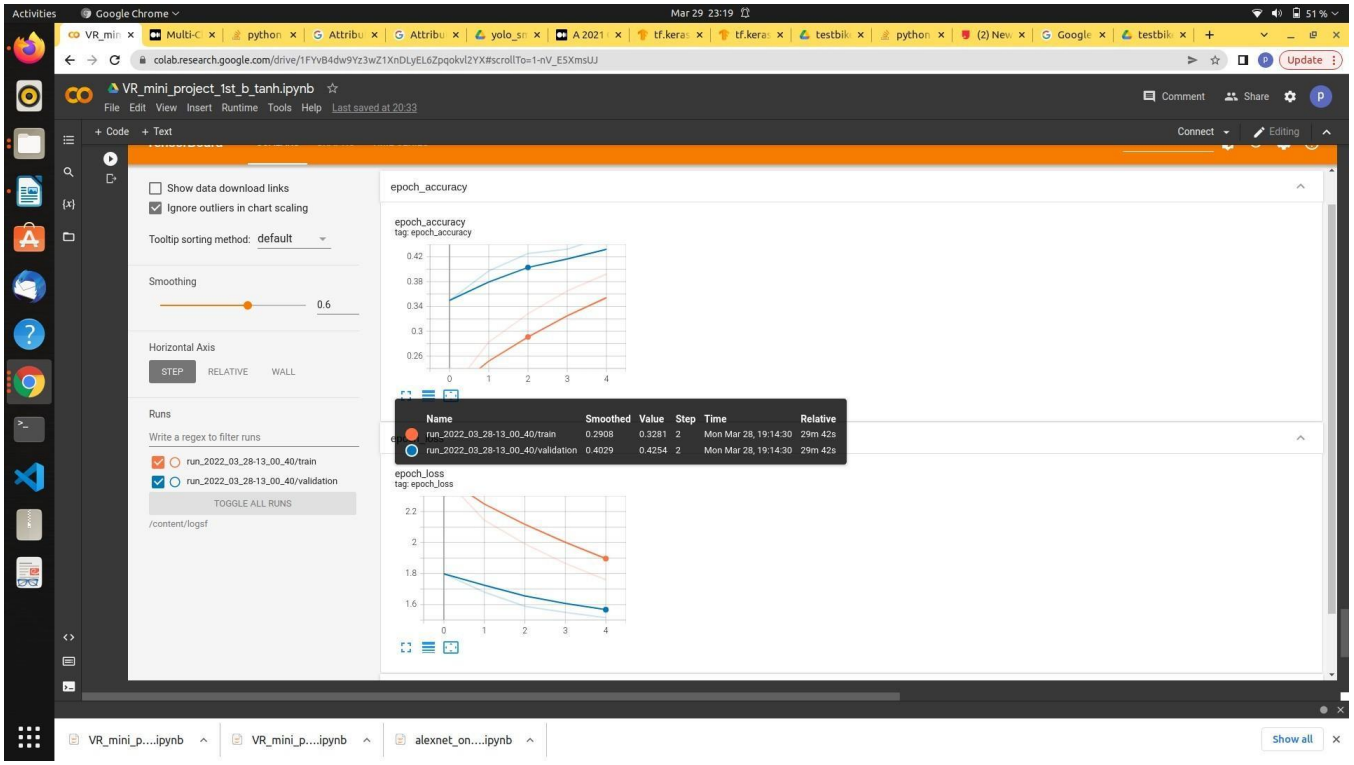


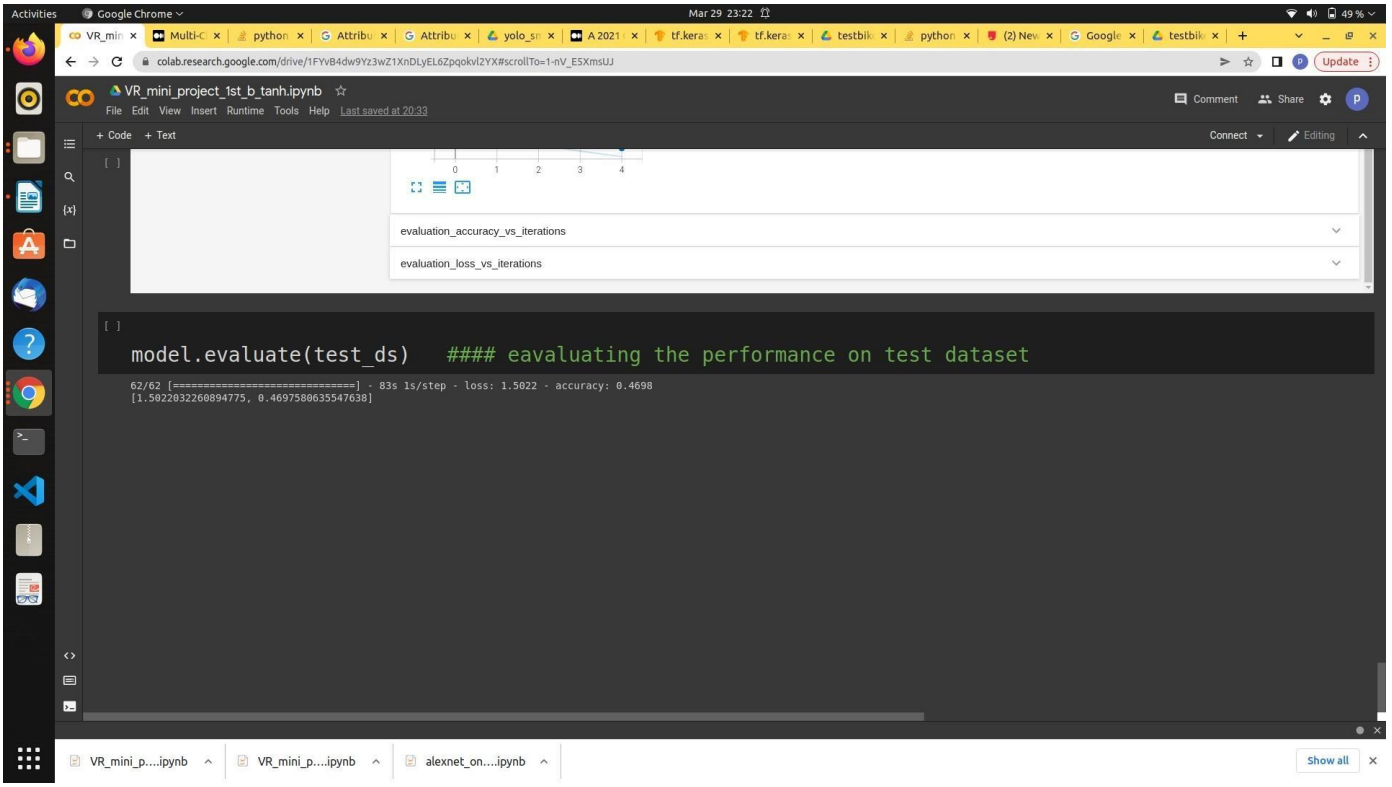
## By using tanh as activation function and we are using softmax activation function for last layer



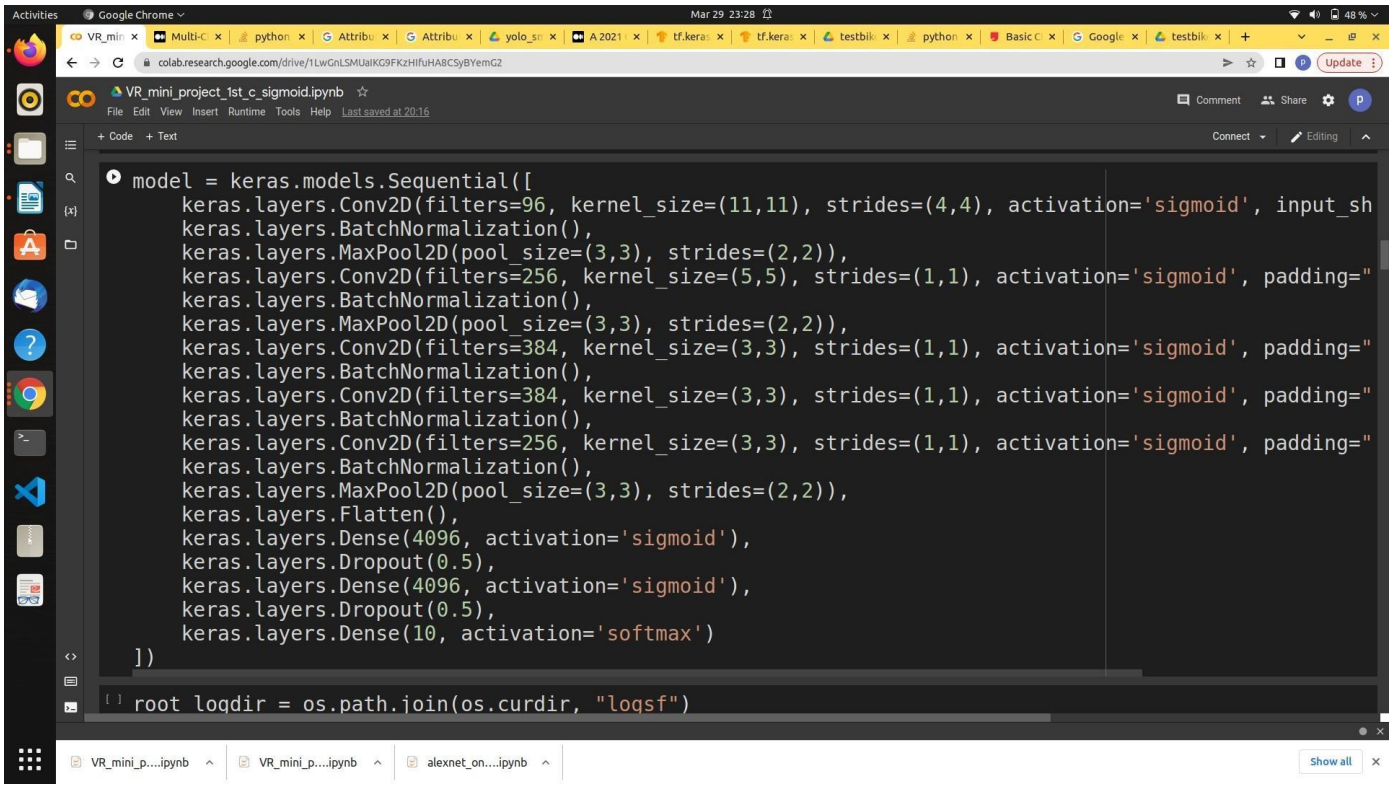
## Loss and Accuracy graph using Tensorboard:-



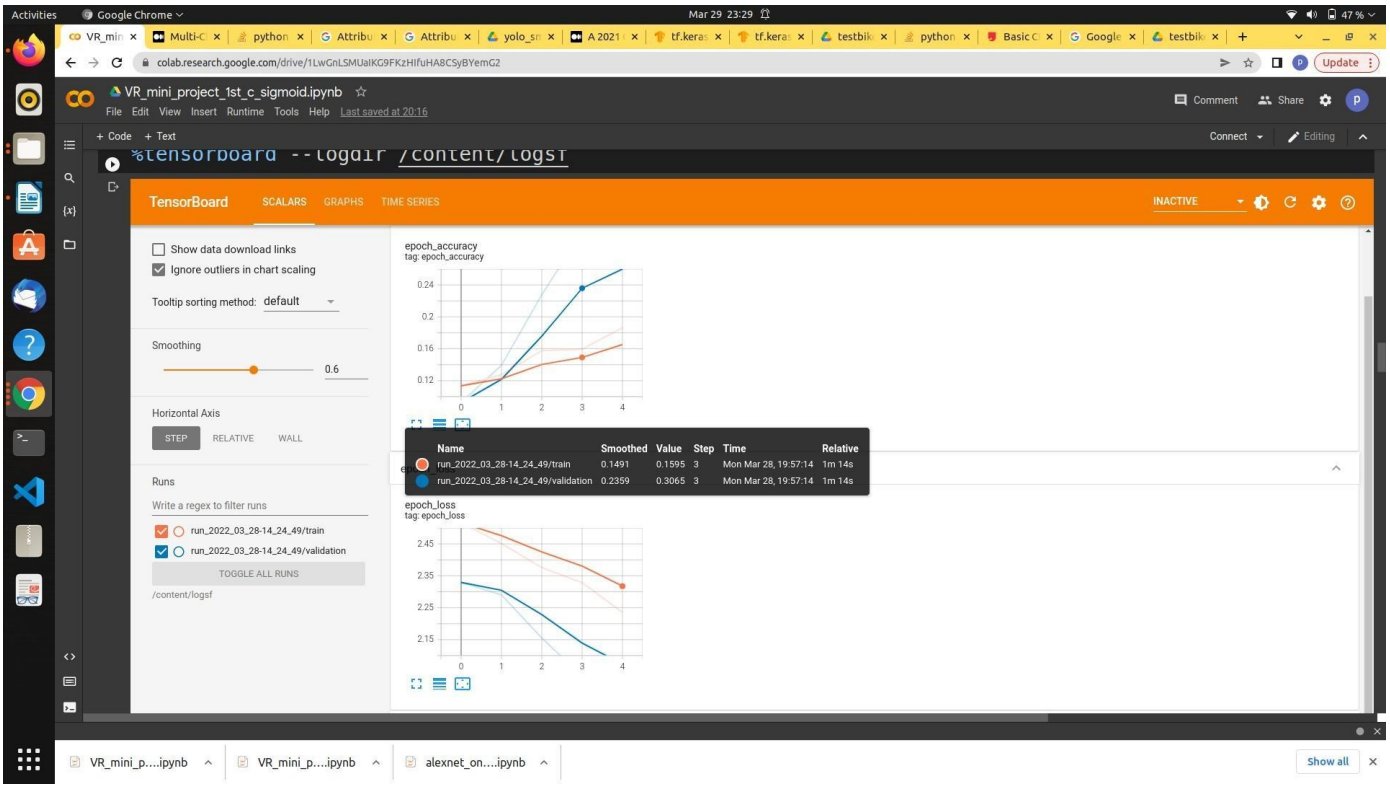
# Evaluating the performance on test datasets:-



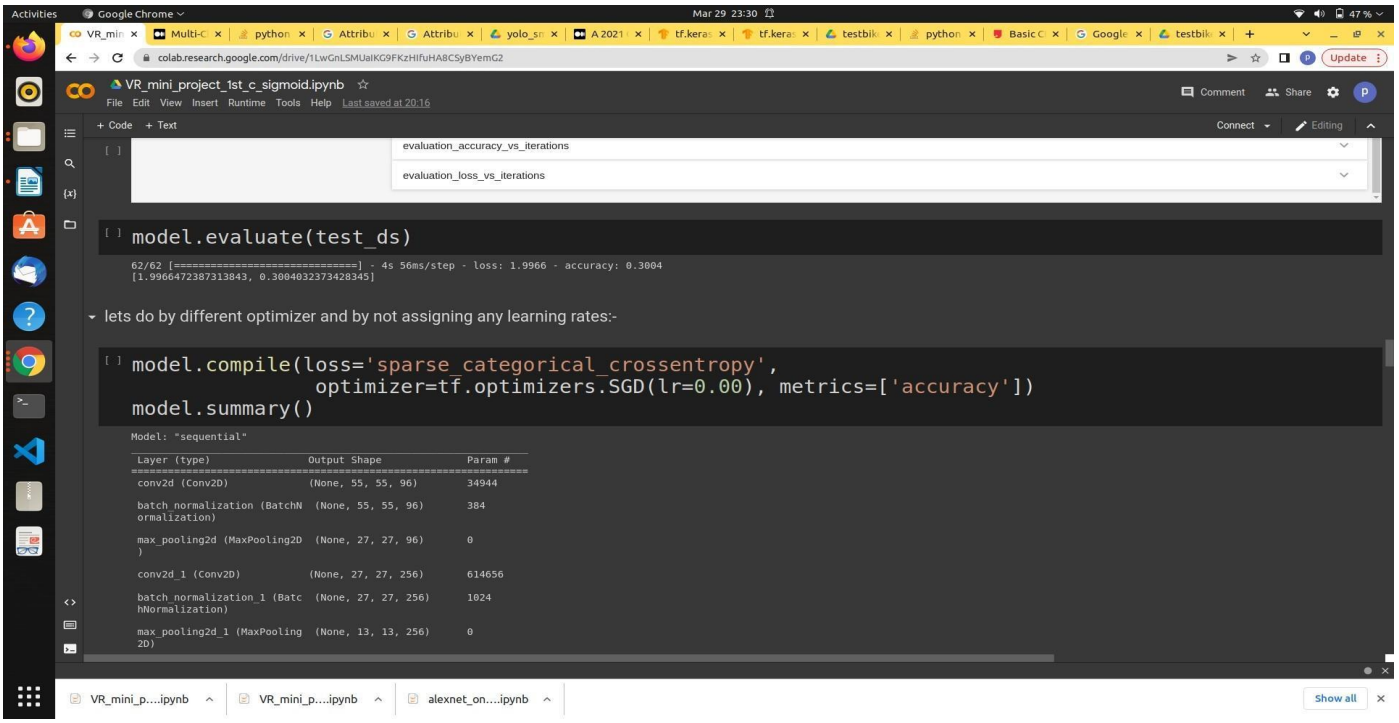
## By using Sigmoid as activation function and we are using softmax activation function for last layer:-



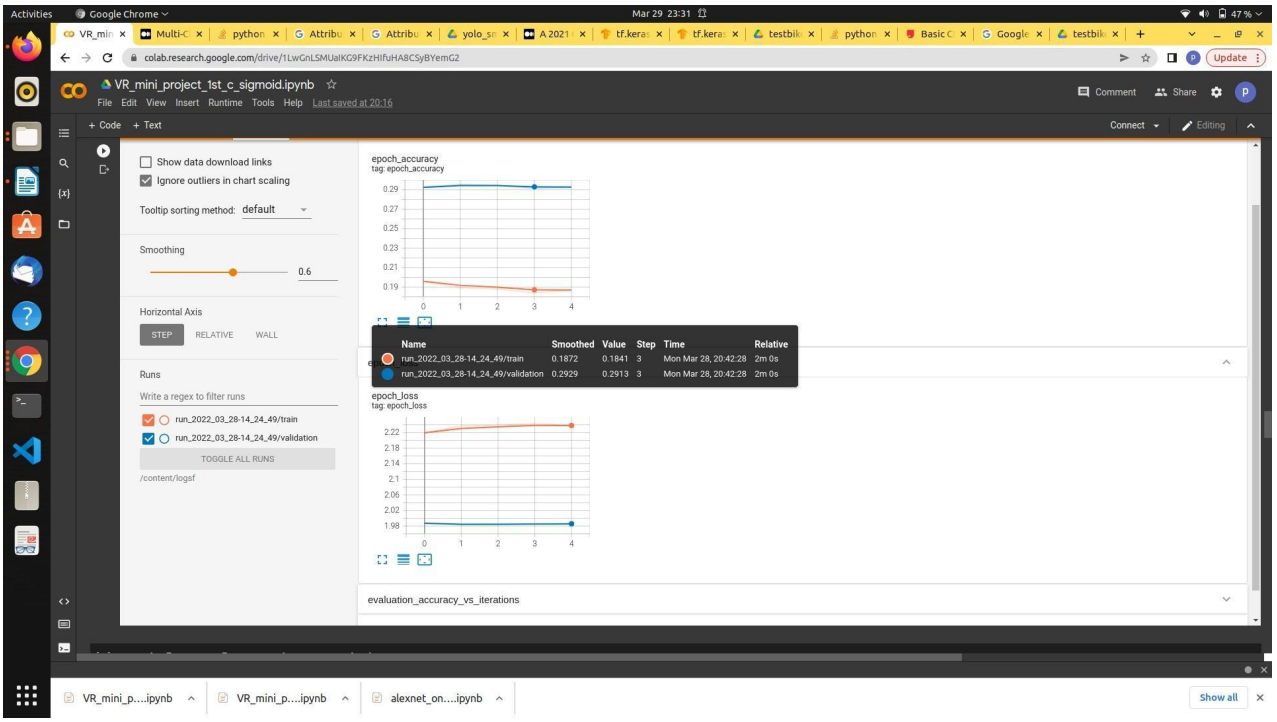
## Loss and Accuracy graph using Tensorboard:-



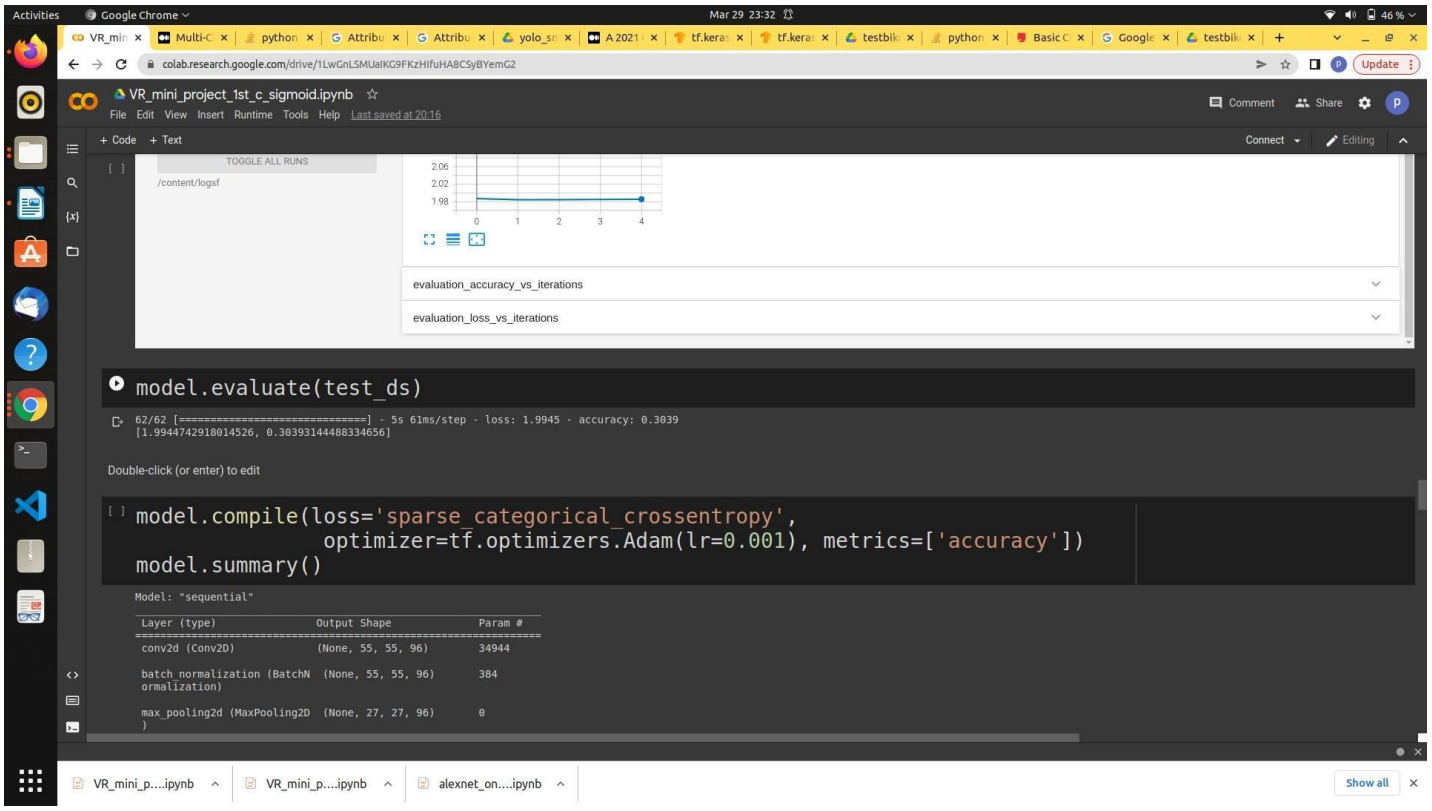

## Evaluating the performance on test datasets:-



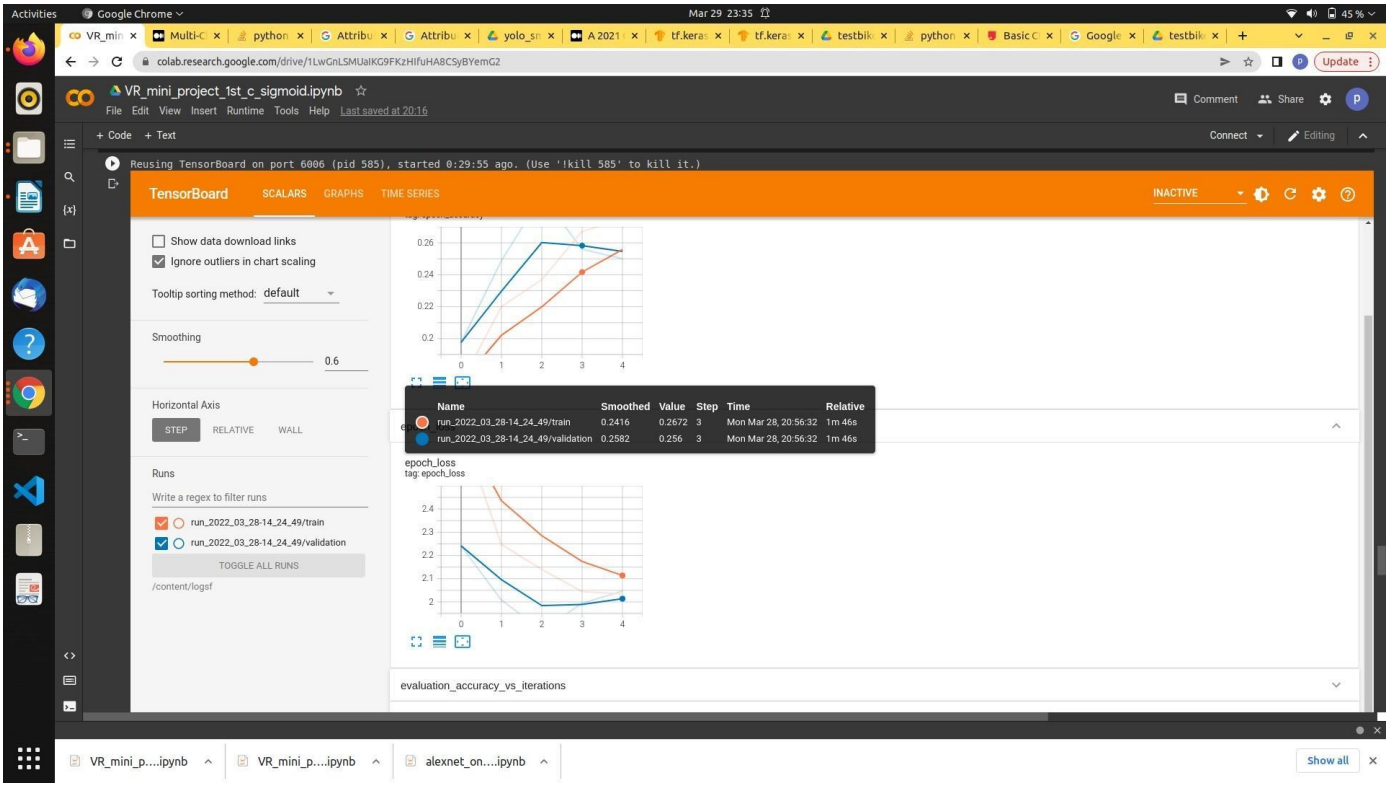

## Using SGD optimizer without any learning rates the graph of loss and accuracy on train and validation datasets :-



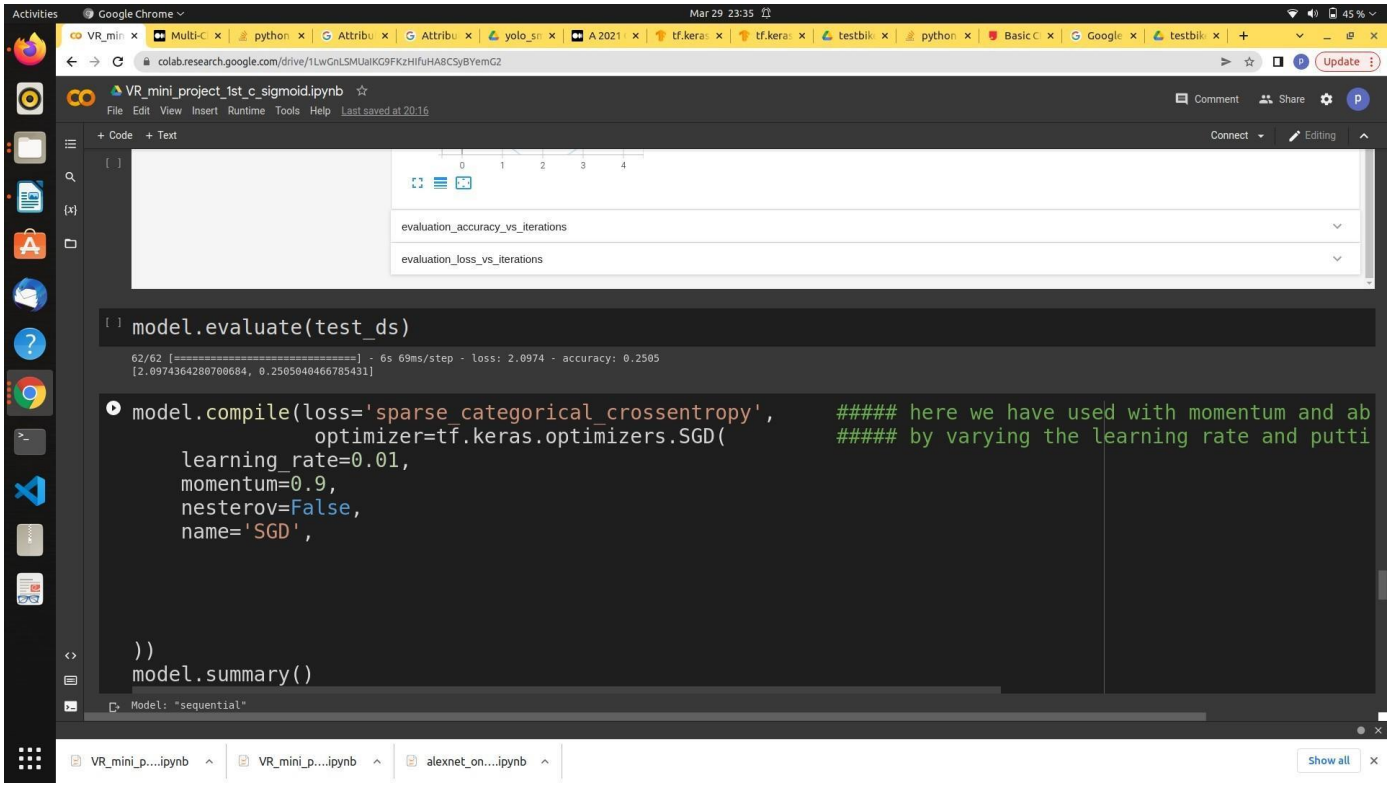
## Visualization of accuracy and loss on test datasets:-



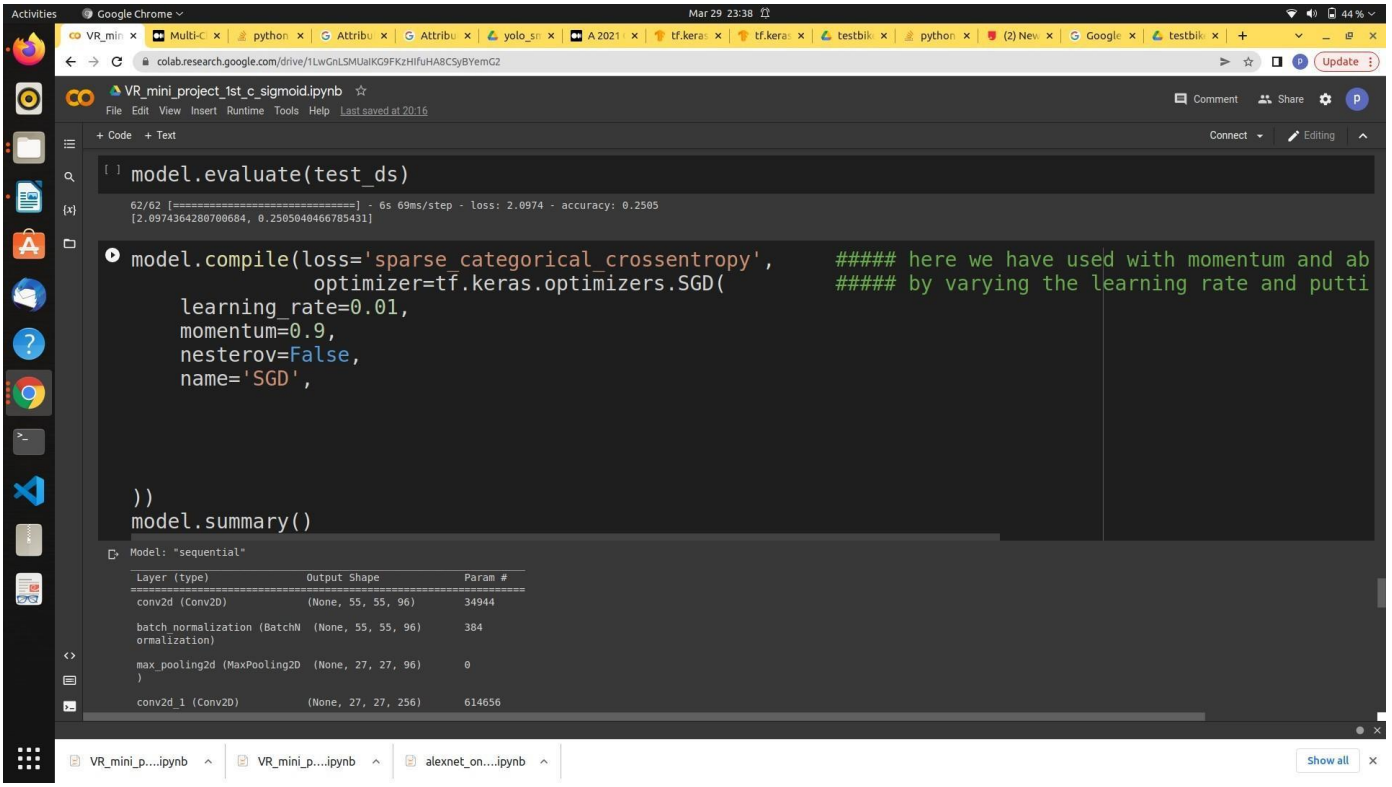

## By using Adam optimizer and Learning rate as .001 without momentum:-

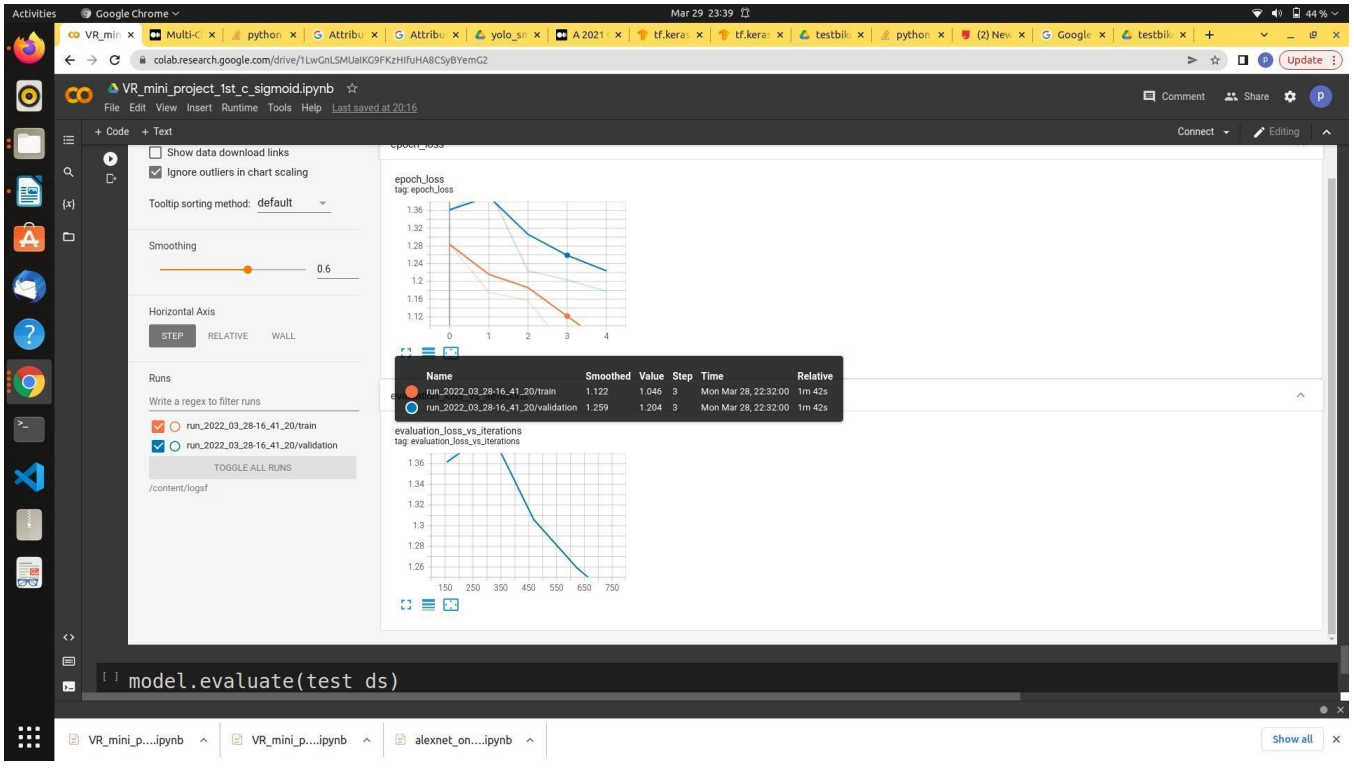


# Visualization of accuracy and loss on test datasets:-

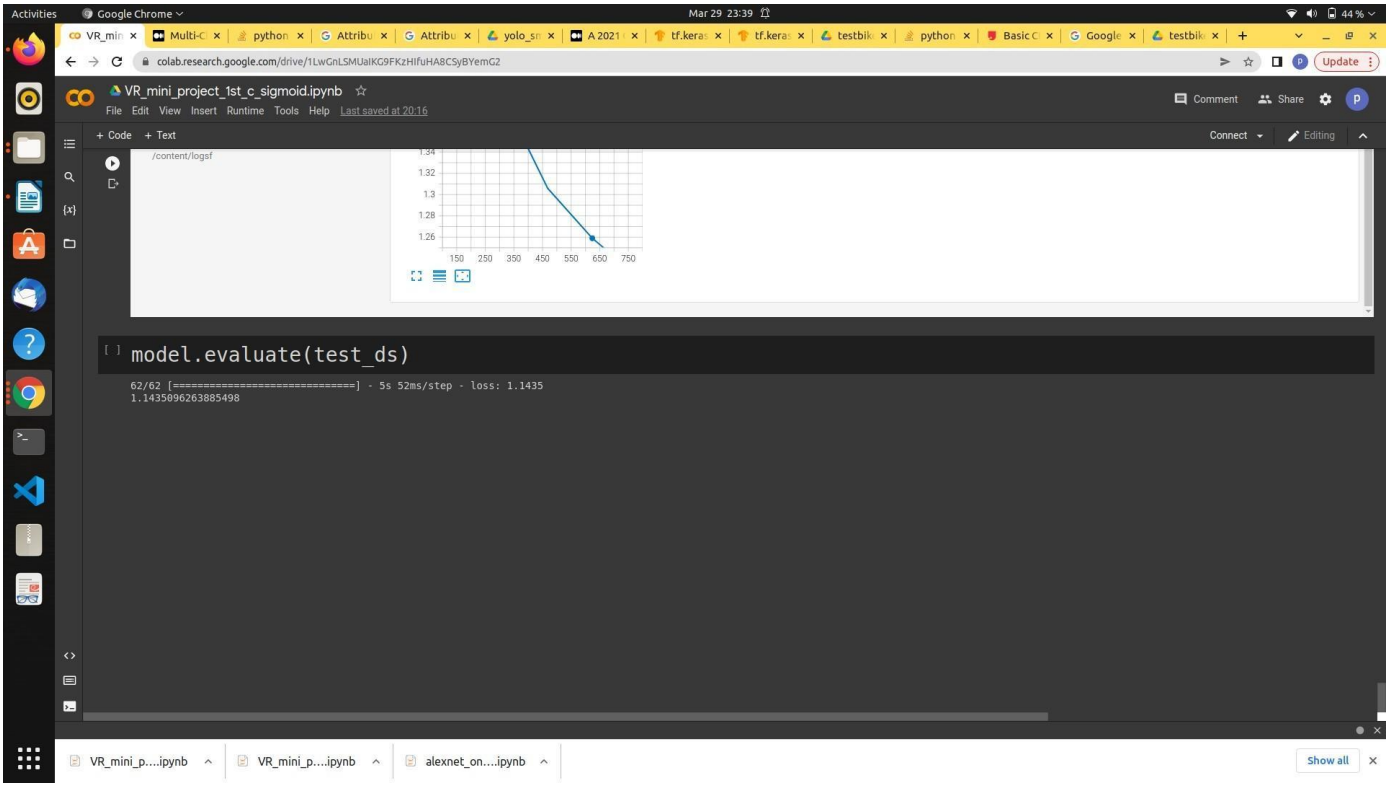


## By using SGD optimizer and Learning rate as .001 with momentum:-



# Visualization of accuracy and loss on test datsets:-



# Report of Assignment 3b

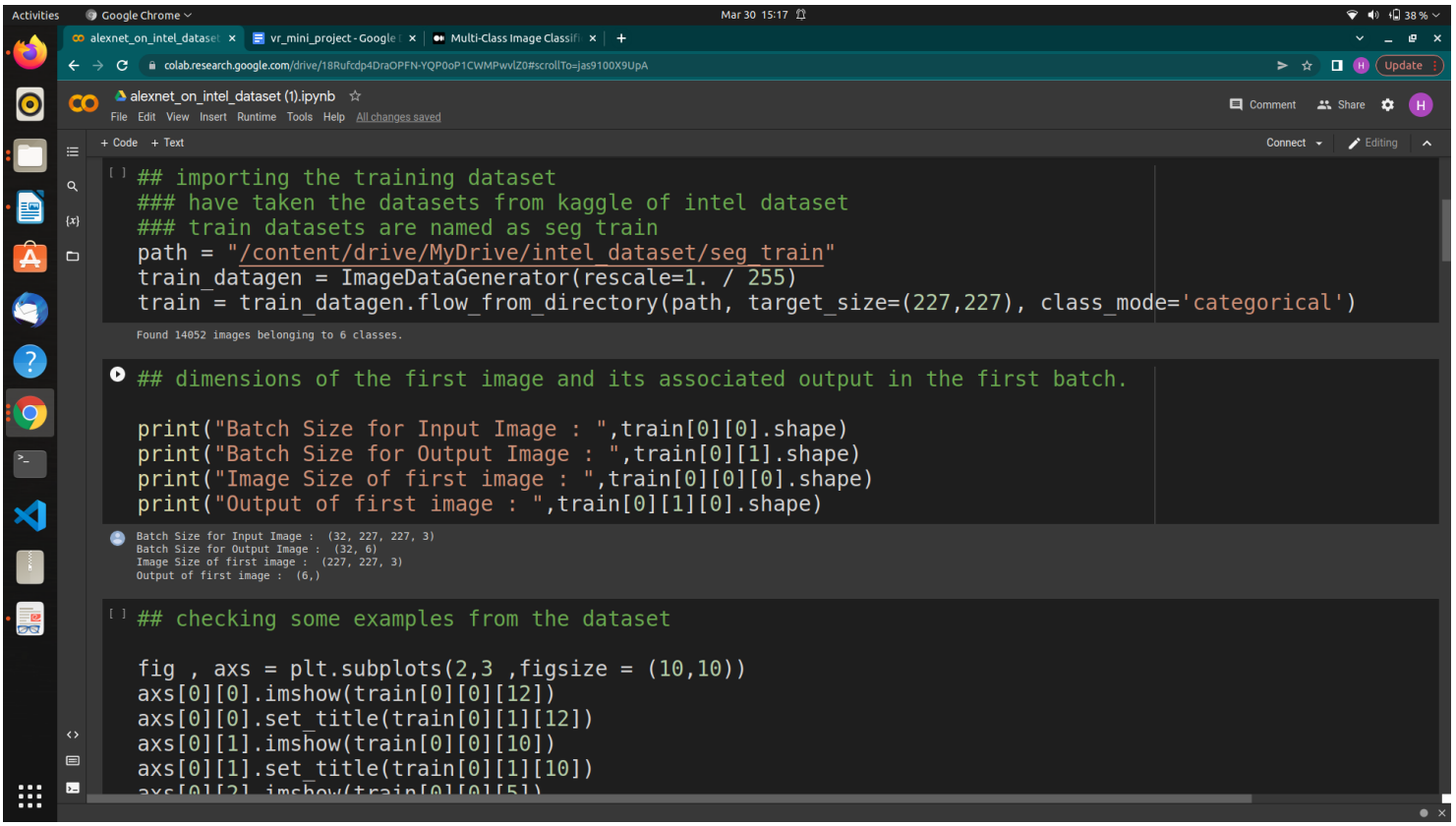
## AlexNet implementation on Intel Datasets using Tensorflow and keras:-

**Keras is an API for python, built over Tensorflow 2.0,which is scalable and adapt to deployment capabilities of Tensorflow.**

**we are using Intel Datasets taken from kaggle:-**

This Data contains around 25k images of size 150x150 distributed under 6 categories,namely : **'buildings' , 'forest' , 'glacier' , 'mountain' , 'sea' , 'street'** . There are 14K images in  training set, 3K in validation set and 7K in the Prediction set.

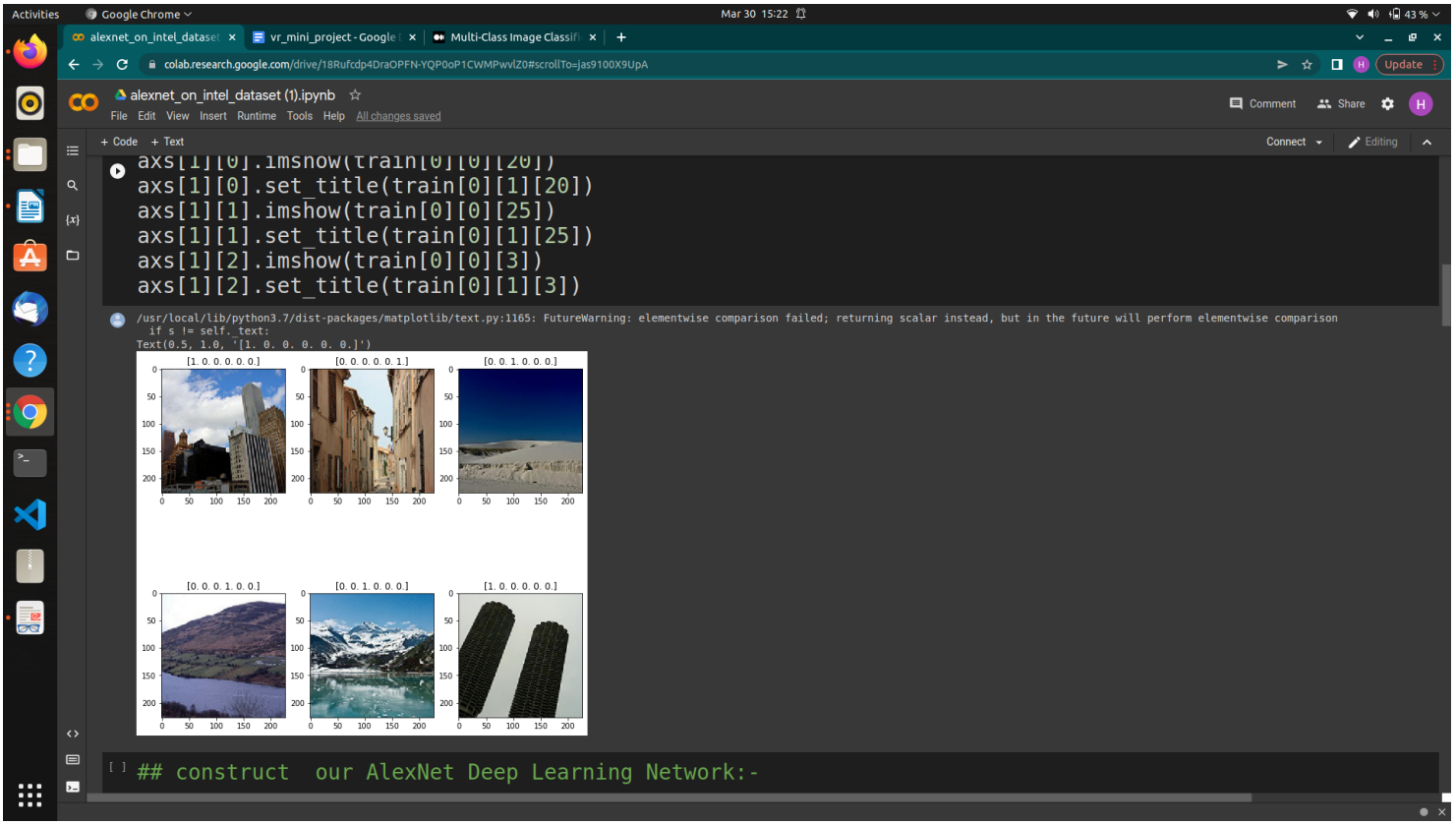
we will import the dataset that is training dataset named as seg_train  now as shown below :-



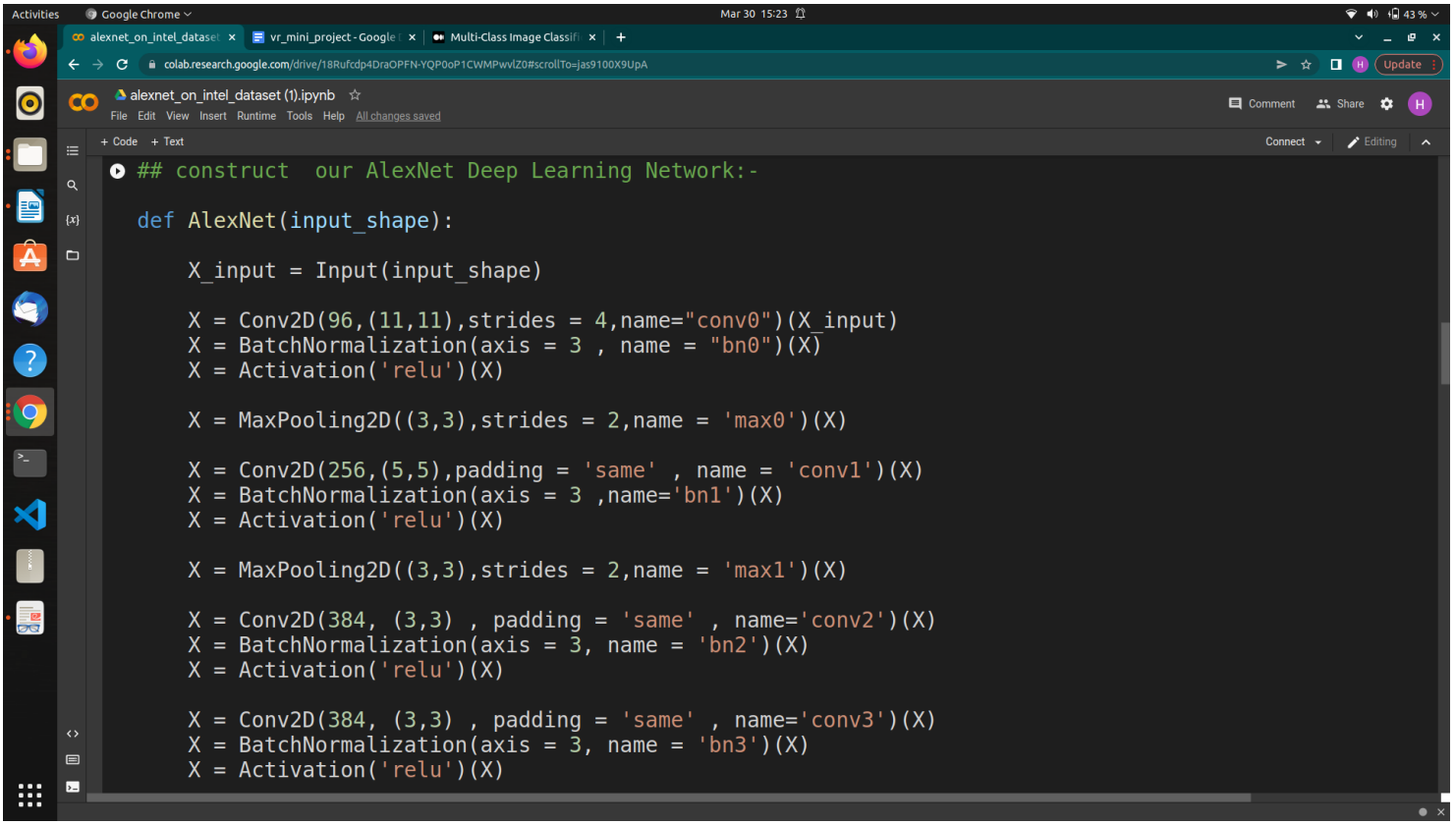

Now as a part of preprocessing , the input size for AlexNet is 227x227x3 and so we will change the target size to (227,227). The by default Batch Size is 32. Lets see the type of train and train_datagen.

Next let us check the dimensions of the first image and its associated output in the first batch which is shown above in the screenshots.

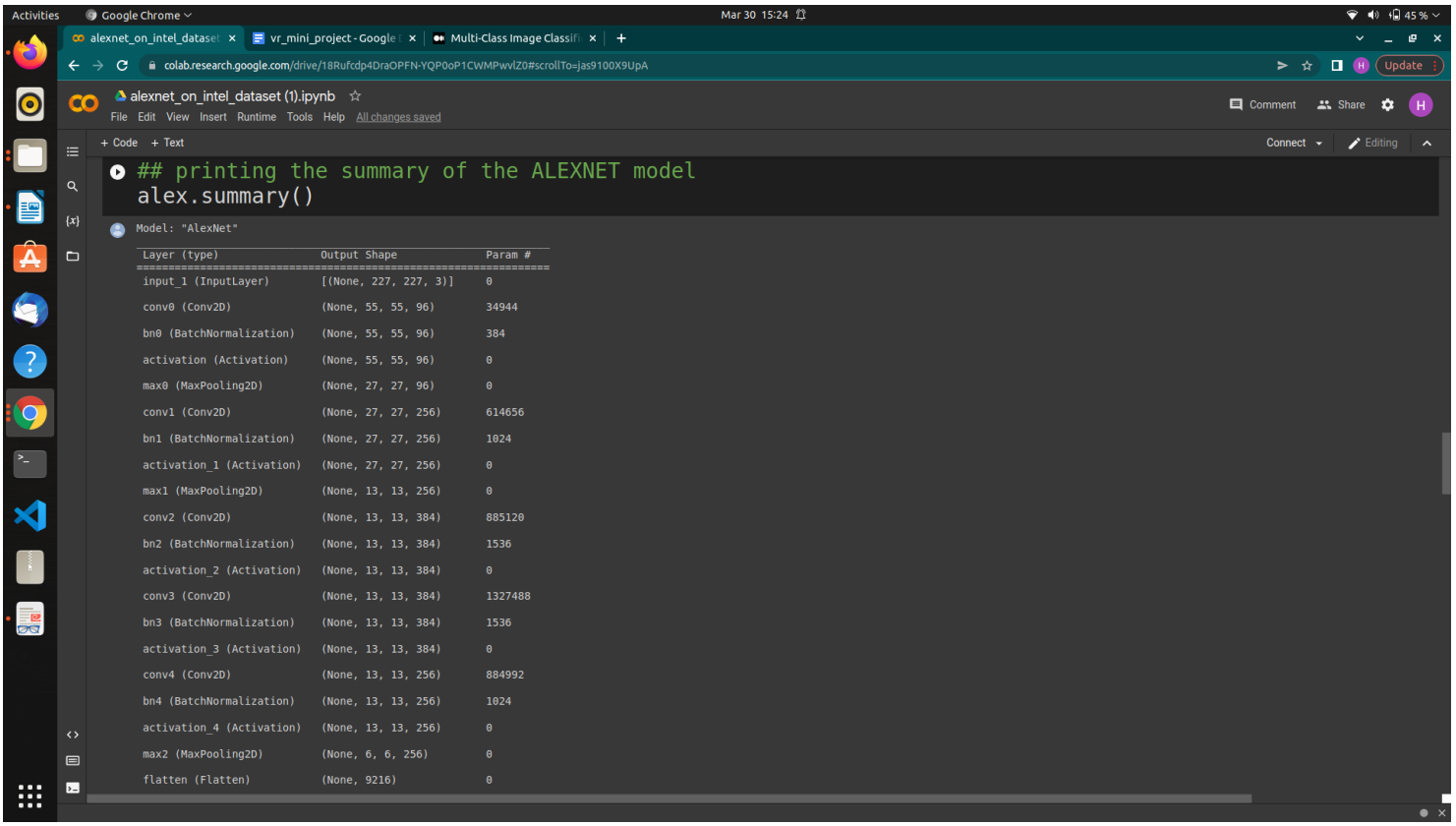
check out some Examples from our  Dataset :



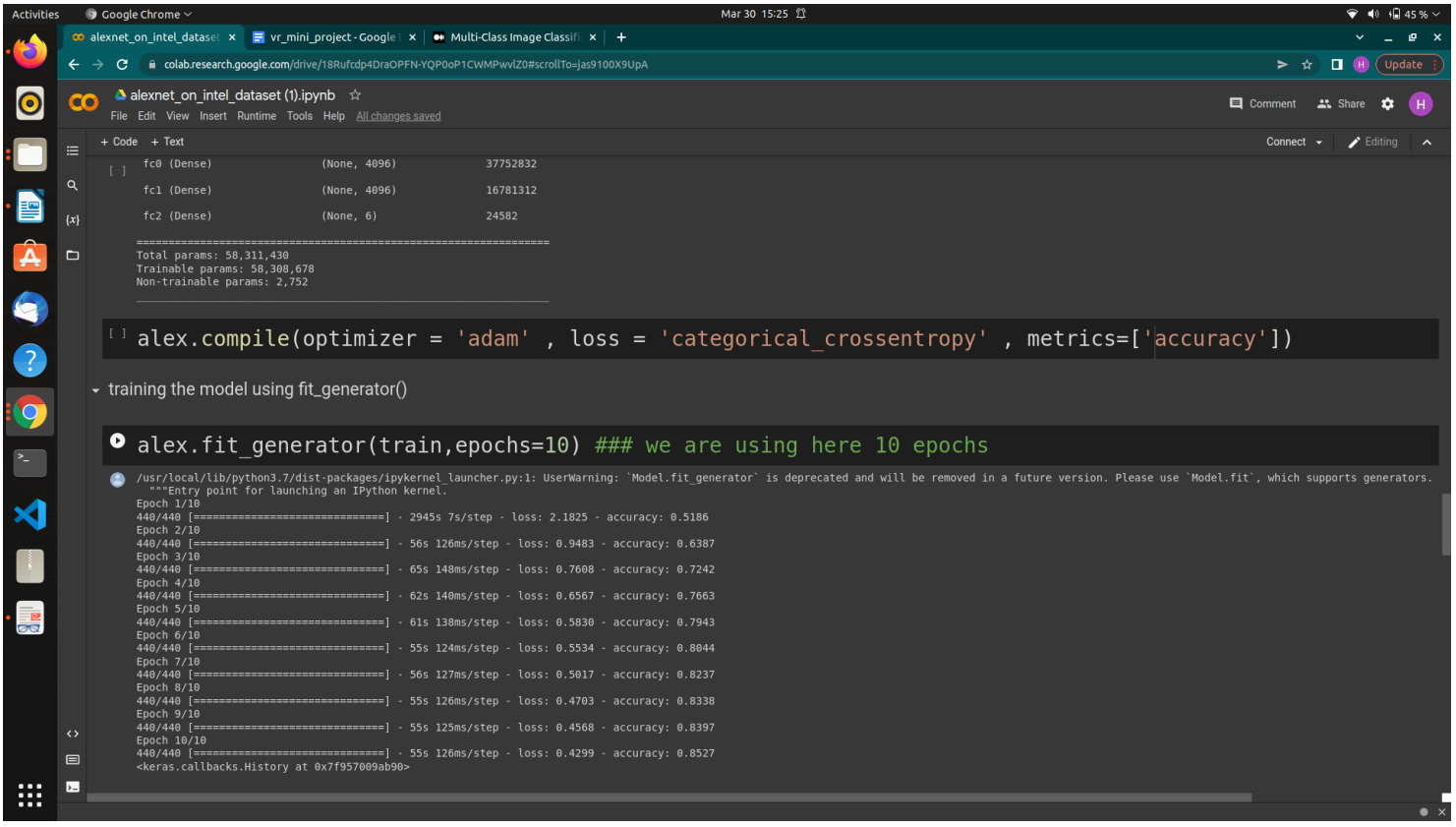
**Now let's start the construction of the Model. The following code will construct our AlexNet Deep Learning Network :**



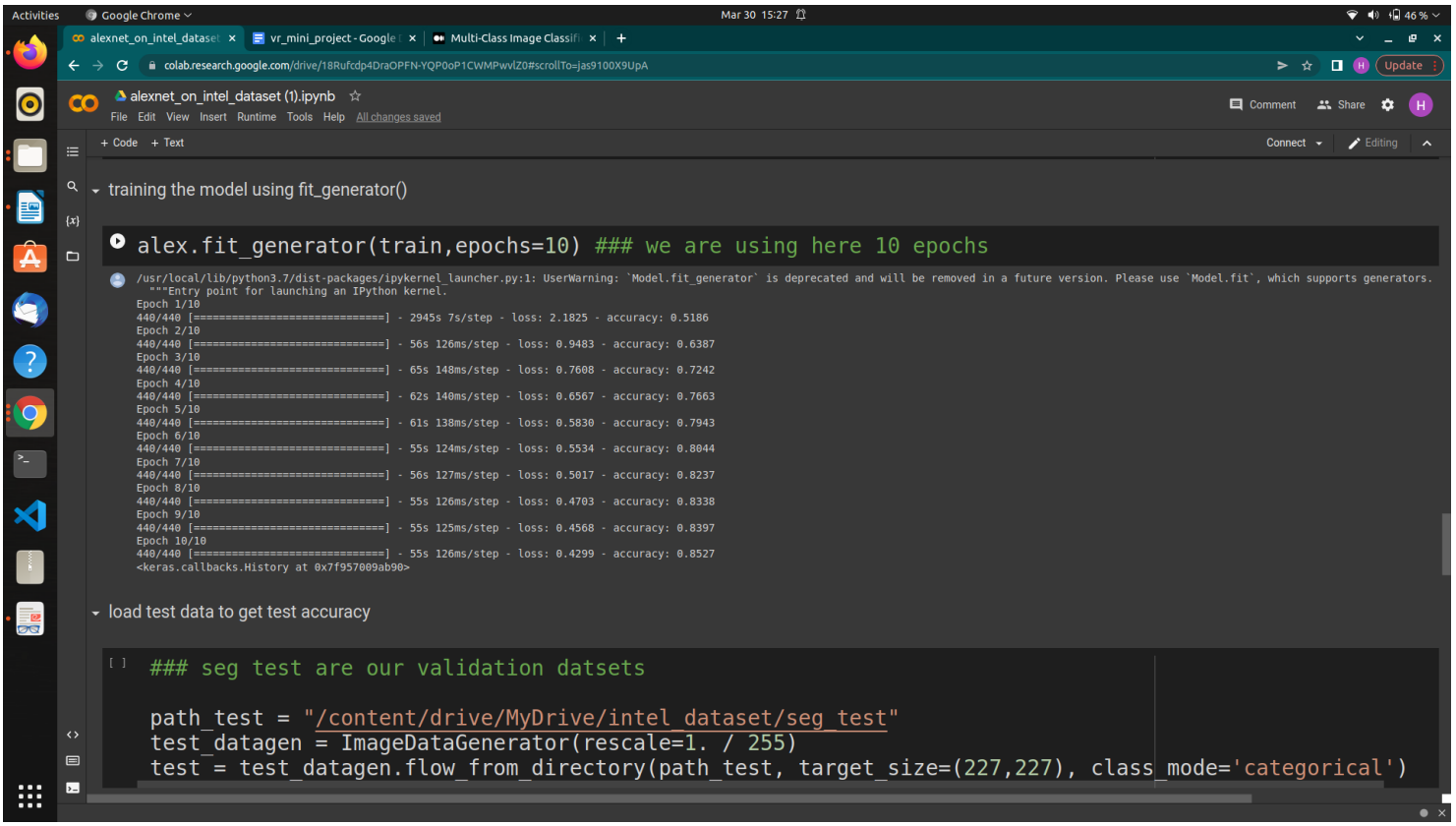

**we are showing the model summary here:-**




# Now we will compile the model using “adam” optimizer and choosing loss as categorical_crossentropy , with accuracy metrics.

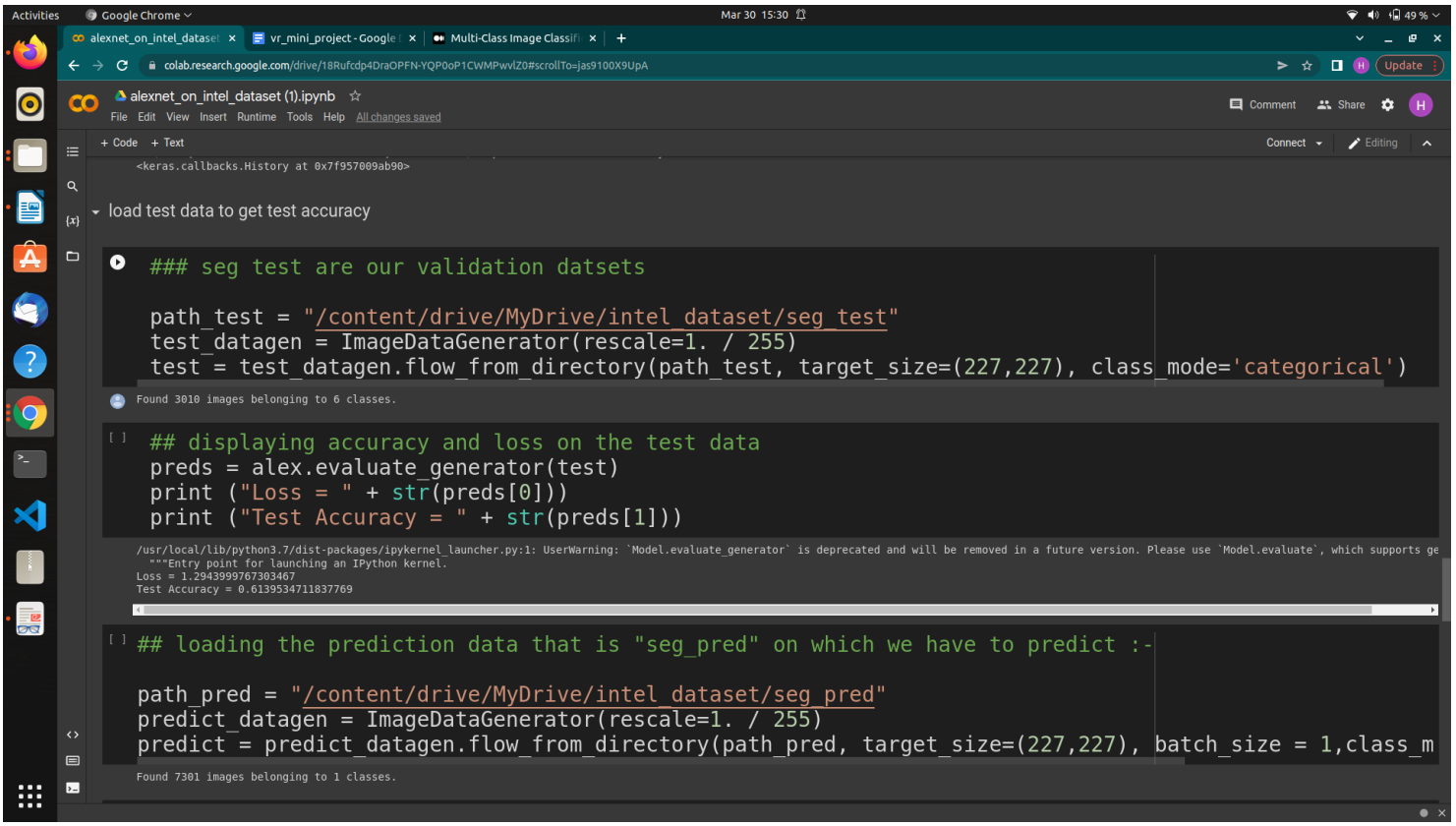


**Now we will train the model using fit_generator with the following command with 10 epochs  :**

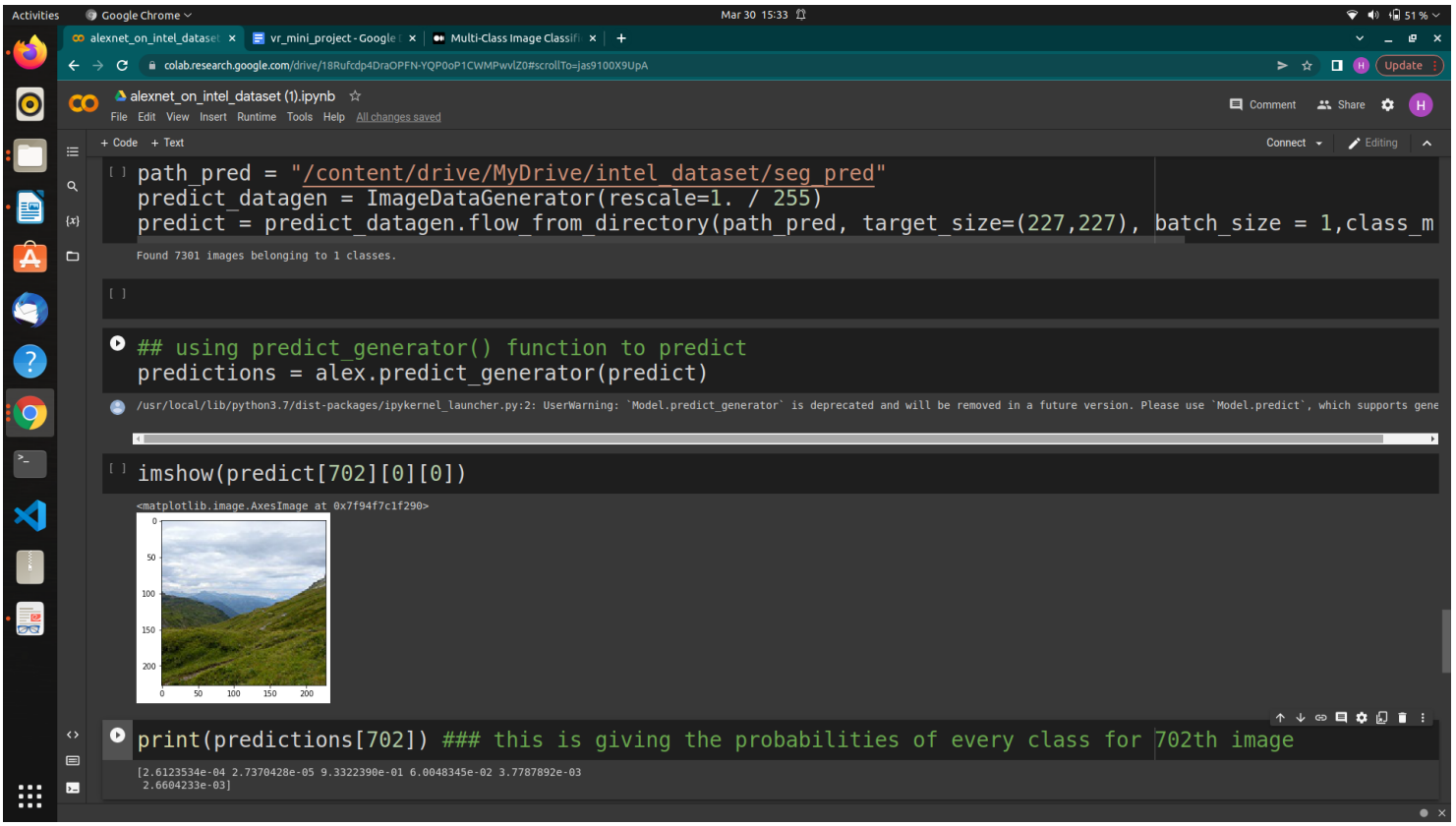



**and got final accuracy of 85% and losses of  42%**

## Now we will load validation data to get test accuracy :-

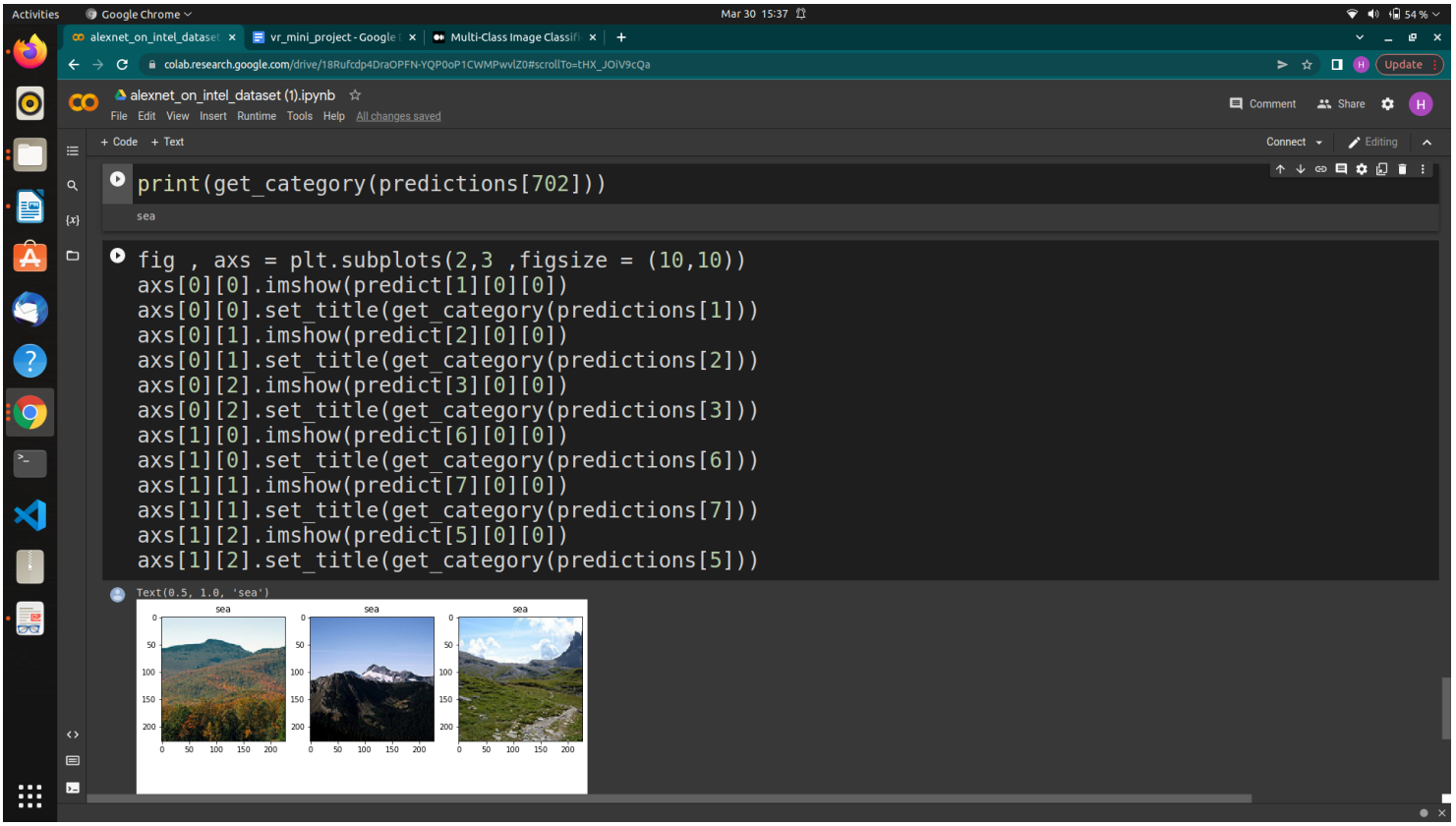



**We got a test accuracy of 61.3% Now we will run the model over prediction Images to predict the image.**




This is the output of our model, since we used softmax at last layer , the model is returning the probabilities for each category for this particular image input. As seen above, the model is predicting the image as ‘sea’ with a probability of 0.933223

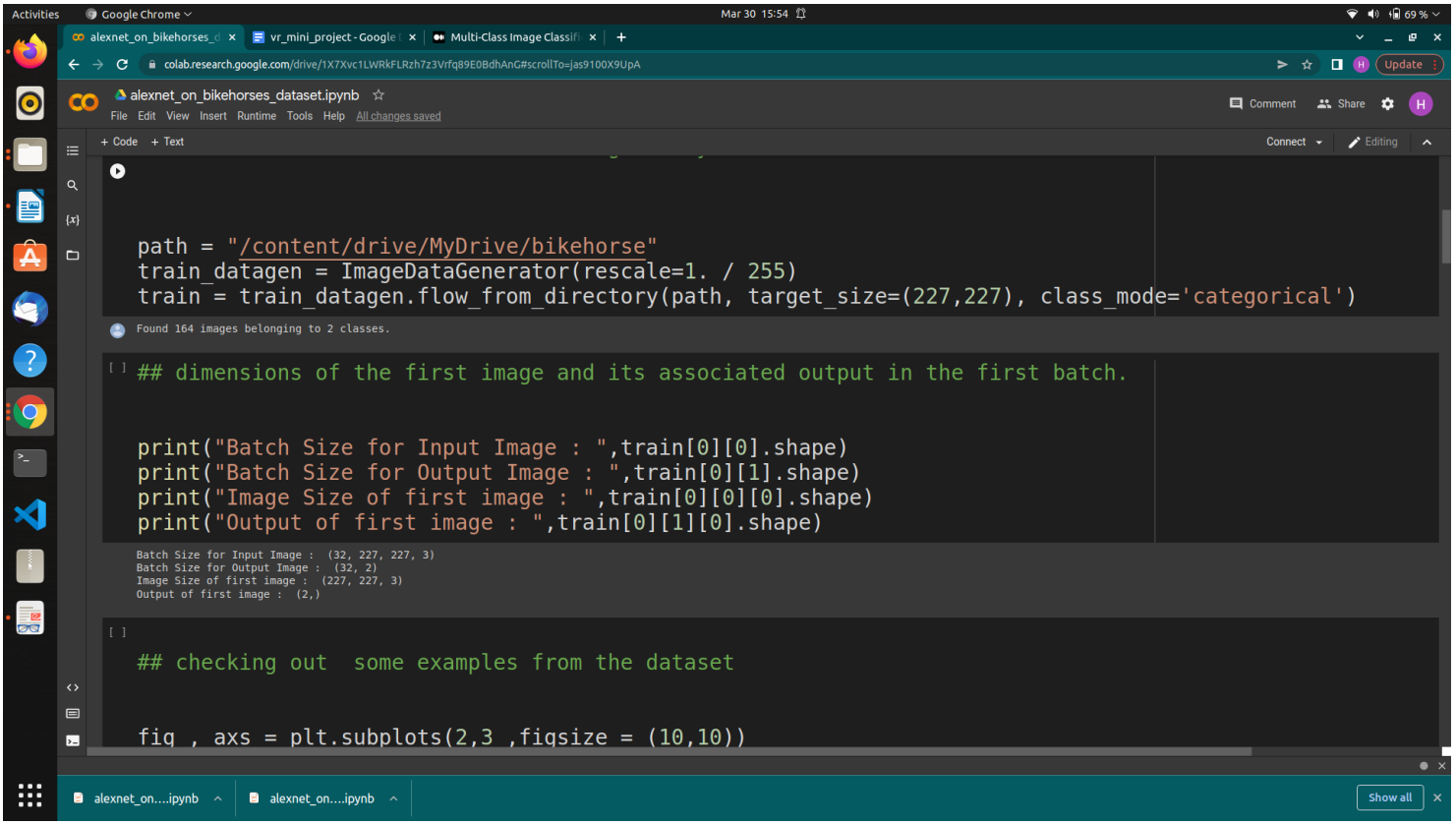
Now we don’t want to have this to be our output format, so we will make a function that will give us the category to which the Input Image, predicted by the model will belong to.



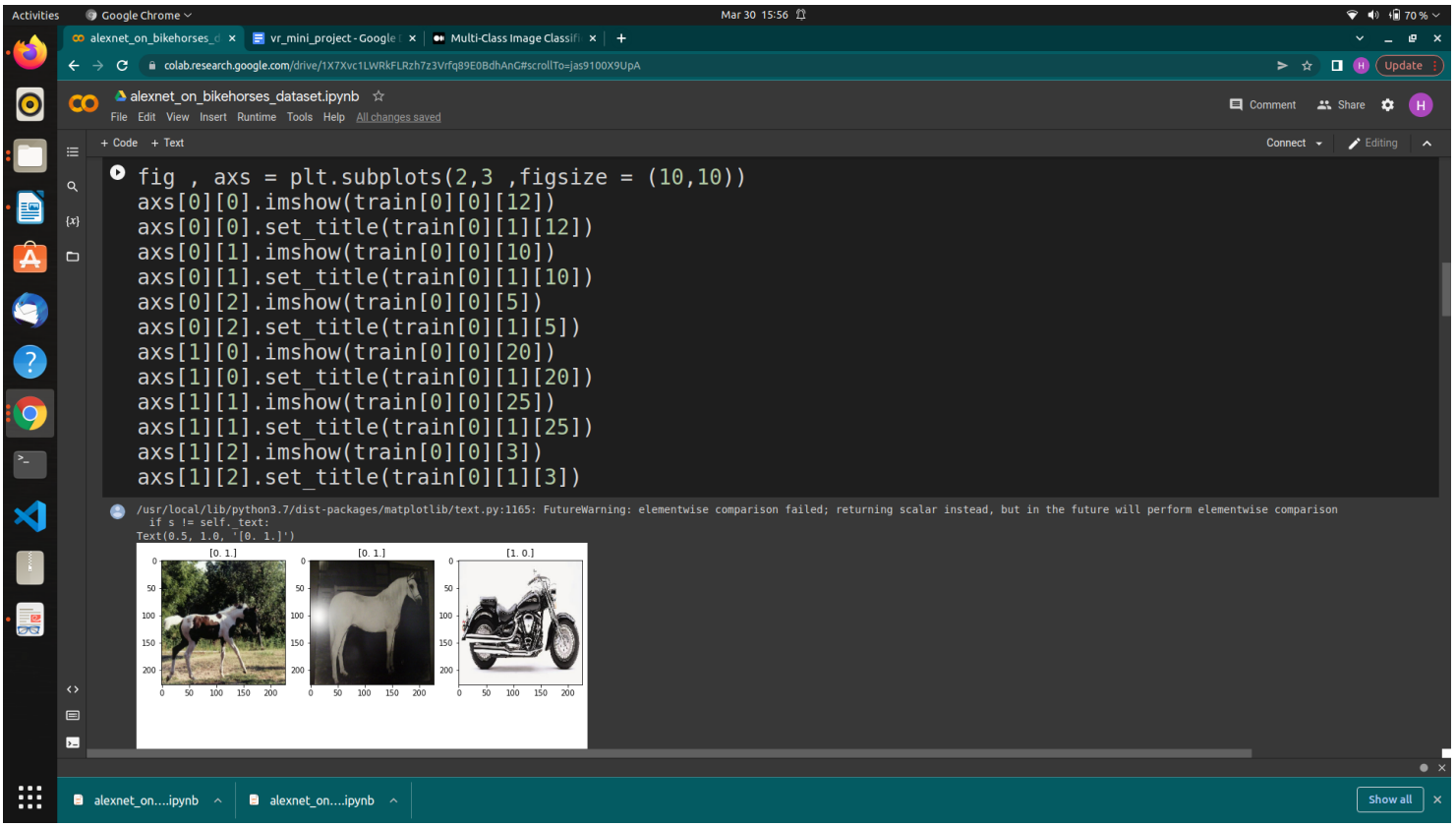

**Since we have limited access of GPU on collab so can’t able to run more epochs on such a huge dataset, so training accuracy is not that much and also  some prediction is going wrong.**

# ALEXNET ON BIKE VS HORSE DATASET

Now we have implemented same AlexNet CNN  on 2 classes datasets given by sir that's on Horse and Bike datasets:-
Uploading the datasets to train the model upon and checking the batch size after changing the target size which are by default:-



# checking out some examples here from our train datasets of bikes and horses:-




## Training the model using fit_generator with the following command with 10 epochs :

```
alex.compile(optimizer = 'adam' , loss = 'categorical_crossentropy' , metrics=['accuracy'])
```

```
erator(train,epochs=10)    ### we are using here 10 epochs have checked through increasing epochs too
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:1: UserWarning: `Model.fit_generator` is deprecated and will be removed in a future version. Please use `Model.fit`, which supports generators.
  """Entry point for launching an IPython kernel.
Epoch 1/10
6/6 [==============================] - 1s 228ms/step - loss: 0.0209 - accuracy: 0.9939
Epoch 2/10
6/6 [==============================] - 1s 228ms/step - loss: 0.0216 - accuracy: 0.9939
Epoch 3/10
6/6 [==============================] - 1s 189ms/step - loss: 0.0158 - accuracy: 0.9939
Epoch 4/10
6/6 [==============================] - 1s 188ms/step - loss: 0.0330 - accuracy: 0.9878
Epoch 5/10
6/6 [==============================] - 1s 184ms/step - loss: 0.0682 - accuracy: 0.9695
Epoch 6/10
6/6 [==============================] - 1s 202ms/step - loss: 0.0971 - accuracy: 0.9634
Epoch 7/10
6/6 [==============================] - 1s 185ms/step - loss: 0.0505 - accuracy: 0.9756
Epoch 8/10
6/6 [==============================] - 1s 200ms/step - loss: 0.0371 - accuracy: 0.9878
Epoch 9/10
6/6 [==============================] - 1s 187ms/step - loss: 0.0387 - accuracy: 0.9878
Epoch 10/10
6/6 [==============================] - 1s 193ms/step - loss: 0.0376 - accuracy: 0.9817
<keras.callbacks.History at 0x7f748ef4bdd0>
```

```
# path_test = '/content/drive/MyDrive/test'
# test_datagen = ImageDataGenerator(rescale=1. / 255)
# test = test_datagen.flow_from_directory(path_test, target_size=(227,227), class_mode='categorical')
```

```
Found 0 images belonging to 0 classes.
```

```
# preds = alex.evaluate_generator(test)
```

**This is the output of our model, since we used softmax at last layer , the model is returning the probabilities for each category for this particular image input. As seen above, the model is predicting the image as 'Horse' with a probability of 0.552**

```
## loading the prediction data that is "seg_pred" on which we have to predict :

path_test = "/content/drive/MyDrive/test"
predict_datagen = ImageDataGenerator(rescale=1. / 255)
predict = predict_datagen.flow_from_directory(path_test, target_size=(227,227), batch_size = 1,class_m
```

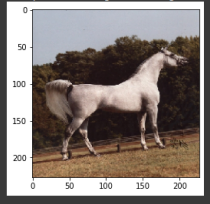
```
Found 15 images belonging to 1 classes.
```

```
## using predict_generator() function to predict
predictions = alex.predict_generator(predict)
```

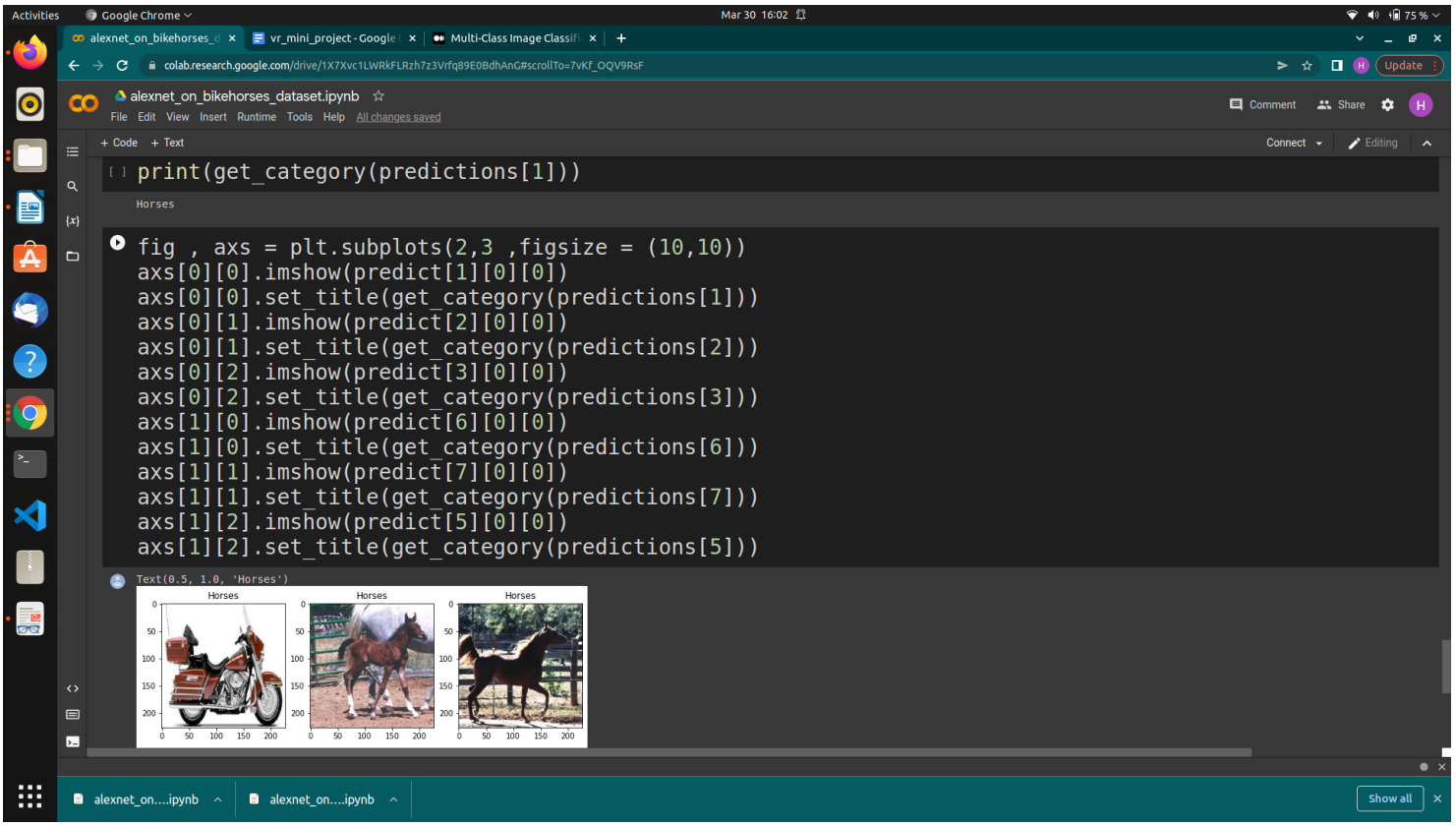
```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:1: UserWarning: `Model.predict_generator` is deprecated and will be removed in a future version. Please use `Model.predict`, which supports gene
  """Entry point for launching an IPython kernel.
```

```
imshow(predict[4][0][0])
```

```
<matplotlib.image.AxesImage at 0x7f748e42bf50>
```



so here are the some prediction of our model on prediction datasets:-




## Conclusion:-

since we have very small dataset given by sir so i have trained my model by varying the epochs from 300 epochs to 50 epochs but prediction accuracy was coming very good o train dataset but not able to predict all data of test datasets properly .

so lastly concluded my prediction after training by giving 100 epochs on training datasets.

# Report of Assignment 3c

## AUTO DETECTION USING YOLO

## What is YOLO?

**YOLO is an abbreviation for the term ‘You Only Look Once’. This is an algorithm that detects and recognizes various objects in a picture (in real-time). Object detection in YOLO is done as a regression problem and provides the class probabilities of the detected images.**

**It takes the entire image in a single instance and predicts the bounding box coordinates and class probabilities for these boxes. The biggest advantage of using YOLO is its superb speed, it's incredibly fast and can process 45 frames per second. YOLO also understands generalized object representation.**

**It is based on regression, instead of selecting the interesting part of an Image, it predicts classes and bounding boxes for the whole image in one run of the Algorithm.**

**Object detection consists of various approaches such as fast R-CNN, Retina-Net, and Single-Shot MultiBox Detector (SSD). Although these approaches have solved the challenges of data limitation and modeling in object detection, they are not able to detect objects in a single algorithm run. YOLO algorithm has gained popularity because of its superior performance over the aforementioned object detection techniques.**

**The YOLO algorithm employs convolutional neural networks (CNN) to detect objects in real-time. As the name suggests, the algorithm requires only a single forward propagation through a neural network to detect objects.**

**This means that prediction in the entire image is done in a single algorithm run. The CNN is used to predict various class probabilities and bounding boxes simultaneously.**

**The YOLO algorithm consists of various variants. Some of the common ones include tiny YOLO and YOLOv3.**

# Why is the YOLO algorithm important?

YOLO algorithm is important because of the following reasons:

- **Speed:** This algorithm improves the speed of detection because it can predict objects in real-time.
- **High accuracy:** YOLO is a predictive technique that provides accurate results with minimal background errors.
- **Learning capabilities:** The algorithm has excellent learning capabilities that enable it to learn the representations of objects and apply them in object detection.

# How the YOLO algorithm works

**YOLO algorithm works using the following three techniques:**

- **Residual blocks**
- **Bounding box regression**
- **Intersection Over Union (IOU)**

## Residual blocks

First, the image is divided into various grids. Each grid has a dimension of S x S. The following image shows how an input image is divided into grids.


In the image above, there are many grid cells of equal dimension. Every grid cell will detect objects that appear within them. For example, if an object center appears within a certain grid cell, then this cell will be responsible for detecting it.

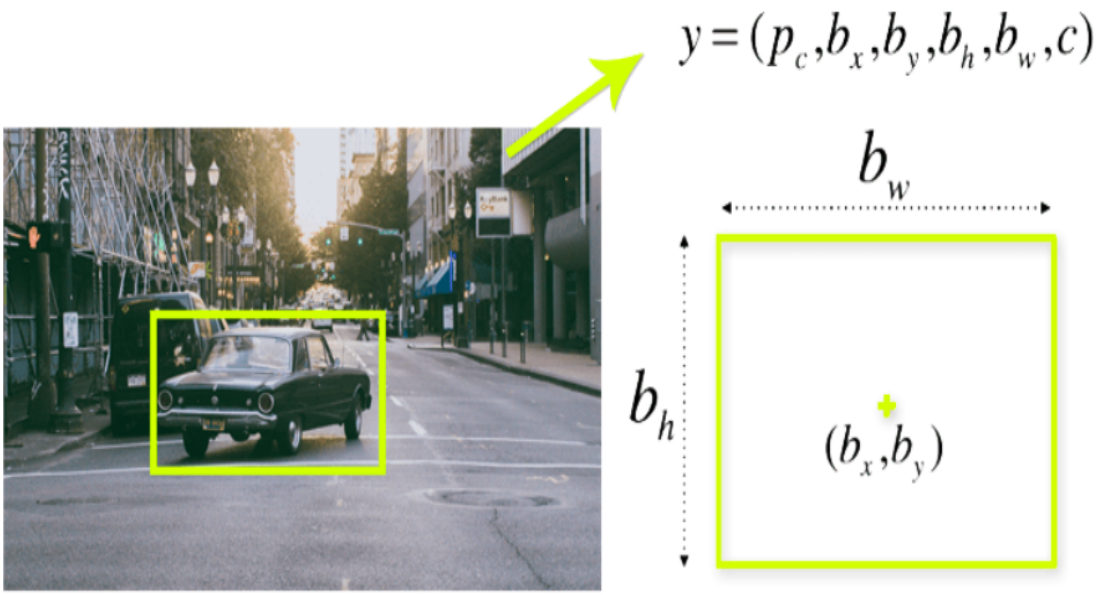
## Bounding box regression

A bounding box is an outline that highlights an object in an image.

Every bounding box in the image consists of the following attributes:

- Width (bw)
- Height (bh)
- Class (for example, person, car, traffic light, etc.)- This is represented by the letter c.
- Bounding box center (bx,by)

The following image shows an example of a bounding box. The bounding box has been represented by a yellow outline.



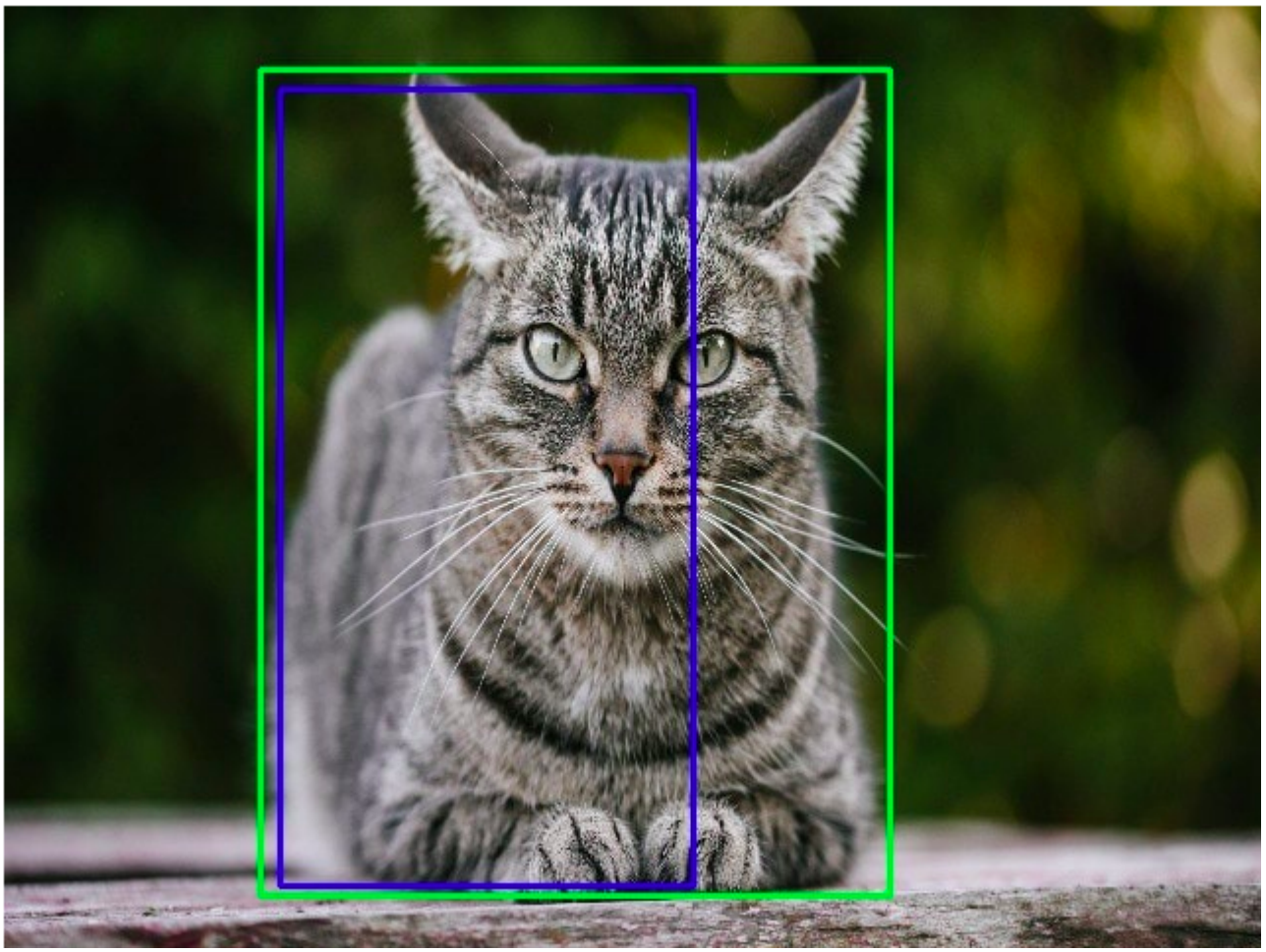

YOLO uses a single bounding box regression to predict the height, width, center, and class of objects. In the image above, represents the probability of an object appearing in the bounding box.

## Intersection over union (IOU)

Intersection over union (IOU) is a phenomenon in object detection that describes how boxes overlap. YOLO uses IOU to provide an output box that surrounds the objects perfectly.

Each grid cell is responsible for predicting the bounding boxes and their confidence scores. The IOU is equal to 1 if the predicted bounding box is the same as the real box. This mechanism eliminates bounding boxes that are not equal to the real box.

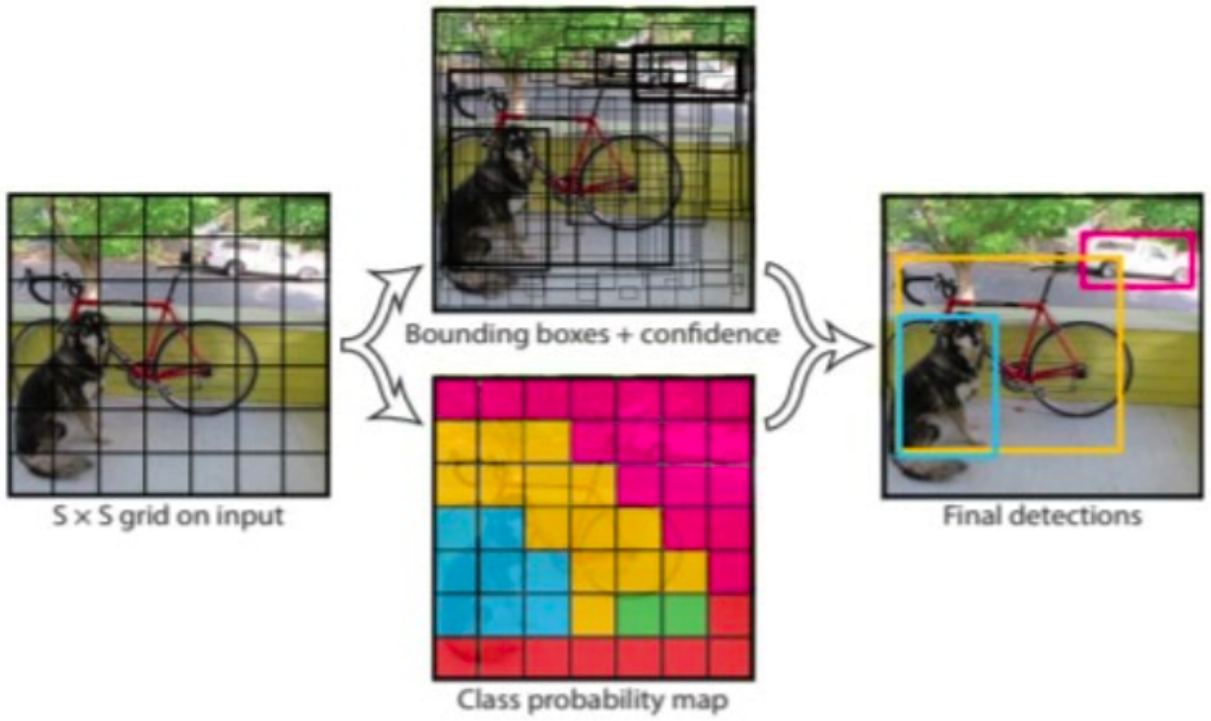
The following image provides a simple example of how IOU works.



**In the image above, there are two bounding boxes, one in green and the other one in blue. The blue box is the predicted box while the green box is the real box. YOLO ensures that the two bounding boxes are equal.**

## Combination of the three techniques

The following image shows how the three techniques are applied to produce the final detection results.




First, the image is divided into grid cells. Each grid cell forecasts B bounding boxes and provides their confidence scores. The cells predict the class probabilities to establish the class of each object.

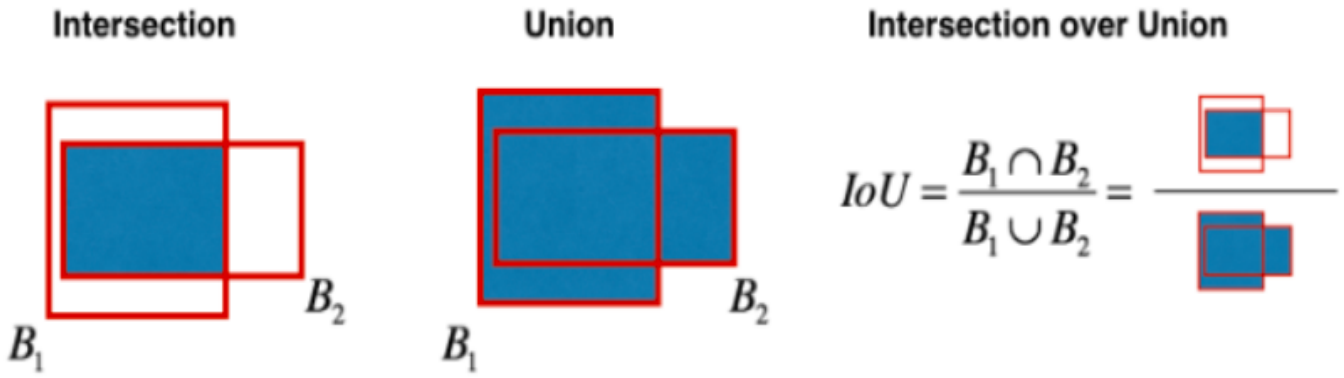
For example, we can notice at least three classes of objects: a car, a dog, and a bicycle. All the predictions are made simultaneously using a single

**convolutional neural network.**

**This shows the before and after of predicting the class probabilities for each grid cell. After predicting the class probabilities, the next step is Non-max suppression. It helps the algorithm to get rid of the unnecessary anchor boxes, like you can see in the figure below, there are numerous anchor boxes calculated based on the class probabilities.**

Anchor boxes

To resolve this problem Non-max suppression eliminates the bounding boxes that are very close by performing the IoU (Intersection over Union) with the one having the highest class probability among them.



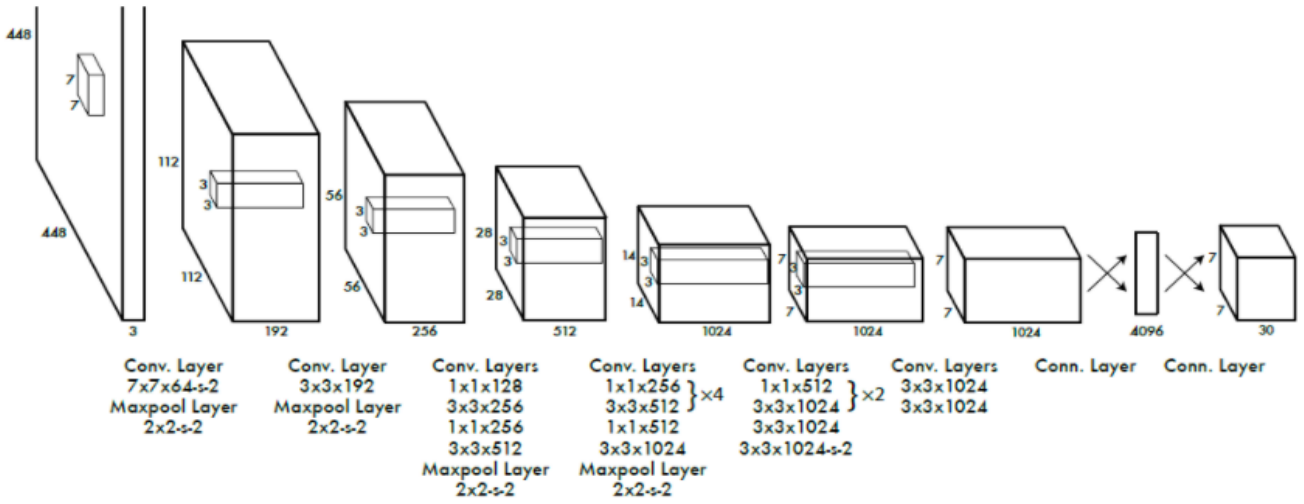

IoU operation

It calculates the value of IoU for all the bounding boxes respective to the one having the highest class probability, it then rejects the bounding boxes whose value of IoU is greater than a threshold. It signifies that those two bounding boxes are covering the same object but the other one has a low probability for the same, thus it is eliminated.
Once done, the algorithm finds the bounding box with next highest class probabilities and does the same process, it is done until we are left with all the different bounding boxes.


Before and After of Non-max suppression

**After this, almost all of our work is done, the algorithm finally outputs the required vector showing the details of the bounding box of the respective class. The overall architecture of the algorithm can be viewed below :**

# Architecture of YOLOV3




YOLO Architecture, source: You Only Look Once: Unified, Real-Time Object detection

Still improvements are being made in the algorithm. We currently have four generations of the YOLO Algorithm from v1 to v4, along with a slightly small version of it YOLO-tiny, it is specifically designed to achieve a incredibly high speed of 220fps.

# Training YOLOv3 object detection on AUTO dataset

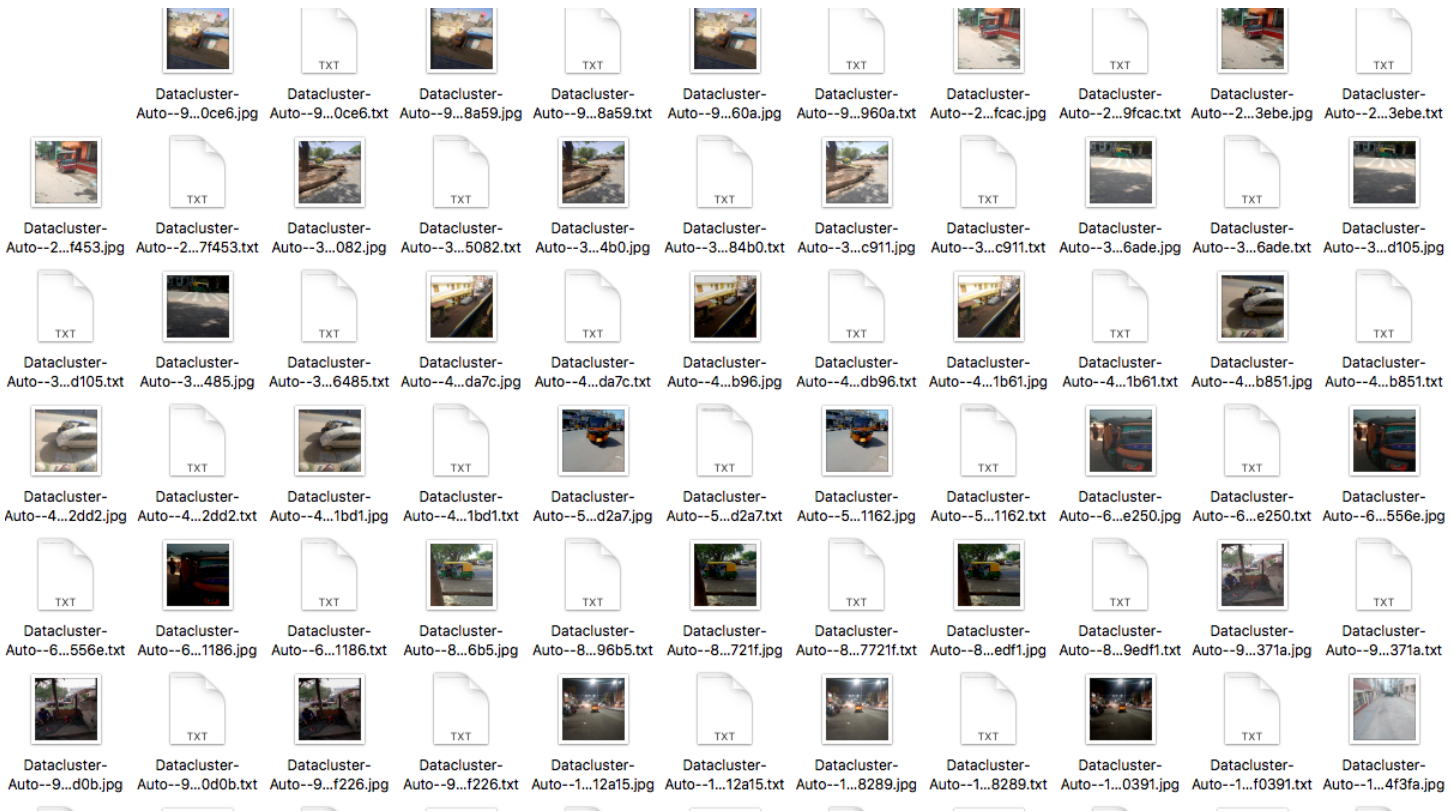
To train a YOLOv3 object detection model we'll training an auto detection model on our own dataset of auto using Colab and darknet

**Our Data:**
given Our dataset of 150 unannotated auto images.

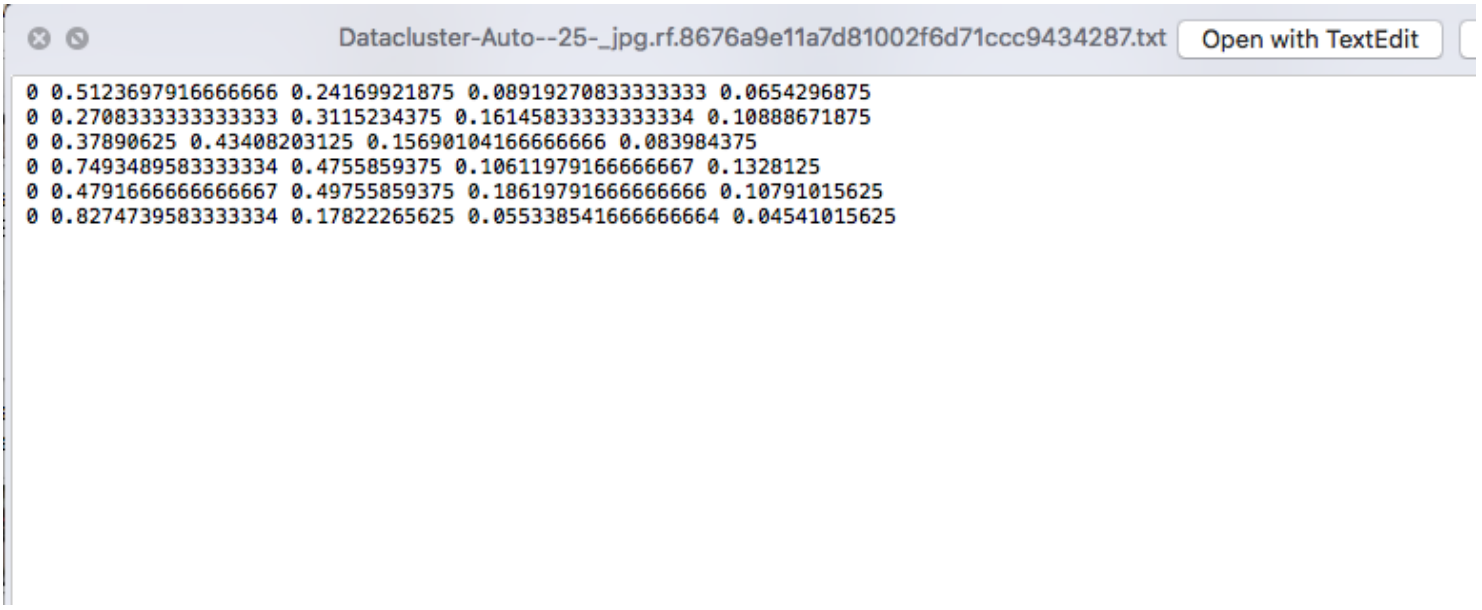
**ANNOTATING ALL THE IMAGES**

Firstly, we have annotated all the images using roboflow software and converted the annotated file into the yolov3 format as shown below:

the image and its annotated txt file is organized at the same place as shown below:

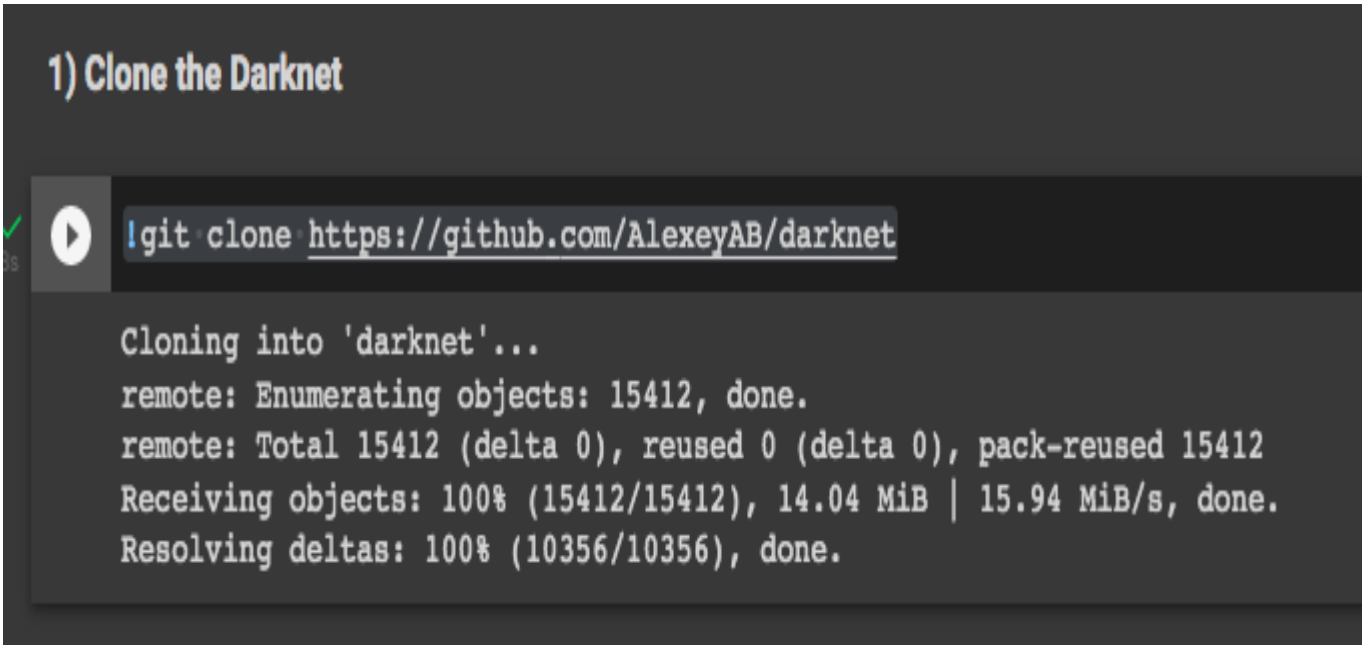



a typical .txt annotated file for an image :



## cloning the darknet repository

*!git clone https://github.com/AlexeyAB/darknet*




# Darknet

**Darknet: An open source platform for neural networks in C**

**Darknet is a high performance open source framework for the implementation of neural networks. Written in C and CUDA, it can be integrated with CPUs and GPUs.**
**Advanced implementations of deep neural networks can be done using Darknet. These implementations include You Only Look Once (YOLO) for real-time object detection, ImageNet classification, recurrent neural networks (RNNs), and many others**

## configuring the darknet network:

3) Configure Darknet network for training YOLO V3

```
[ ] !cp cfg/yolov3.cfg cfg/yolov3_training.cfg
```

```
!sed -i 's/batch=1/batch=64/' cfg/yolov3_training.cfg
!sed -i 's/subdivisions=1/subdivisions=16/' cfg/yolov3_training.cfg
!sed -i 's/max_batches = 500200/max_batches = 6000/' cfg/yolov3_training.cfg
!sed -i '610 s@classes=80@classes=1@' cfg/yolov3_training.cfg
!sed -i '696 s@classes=80@classes=1@' cfg/yolov3_training.cfg
!sed -i '783 s@classes=80@classes=1@' cfg/yolov3_training.cfg
!sed -i '603 s@filters=255@filters=18@' cfg/yolov3_training.cfg
!sed -i '689 s@filters=255@filters=18@' cfg/yolov3_training.cfg
!sed -i '776 s@filters=255@filters=18@' cfg/yolov3_training.cfg
```

```
[ ] # Create folder on google drive so that we can save there the weights
!mkdir "/content/drive/MyDrive/yolo_v3"
```

```
mkdir: cannot create directory '/content/drive/MyDrive/yolo_v3': File exists
```

```
[ ] !echo "auto" > data/obj.names
!echo -e 'classes= 1\ntrain  = data/train.txt\nvalid  = data/test.txt\nnames = data/obj.names\nbackup = /content/drive/MyDrive/yolo_v3' > data/obj.data
!mkdir data/obj
```

```
mkdir: cannot create directory 'data/obj': File exists
```

## Downloading the pretrained weights of the YOLO :-

```
# Download weights darknet model 53
!wget https://pjreddie.com/media/files/darknet53.conv.74
```

```
--2022-03-28 13:26:39--  https://pjreddie.com/media/files/darknet53.conv.74
Resolving pjreddie.com (pjreddie.com)... 128.208.4.108
Connecting to pjreddie.com (pjreddie.com)|128.208.4.108|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 162482580 (155M) [application/octet-stream]
Saving to: 'darknet53.conv.74.2'

darknet53.conv.74.2 100%[==================>] 154.96M  21.0MB/s    in 7.8s

2022-03-28 13:26:47 (19.9 MB/s) - 'darknet53.conv.74.2' saved [162482580/162482580]
```

## We're going to convert the class index on the .txt files. As we're working with only one

class, it's supposed to be class 0. If the index is different from 0 then we're going to change it.

```python
import glob
import os
import re

txt_file_paths = glob.glob(r"data/obj/train/*.txt")
for i, file_path in enumerate(txt_file_paths):
    # get image size
    with open(file_path, "r") as f_o:
        lines = f_o.readlines()

        text_converted = []
        for line in lines:
            print(line)
            numbers = re.findall("[0-9.]+", line)
            print(numbers)
            if numbers:

                # Define coordinates
                text = "{} {} {} {} {}".format(0, numbers[1], numbers[2], numbers[3], numbers[4])
                text_converted.append(text)
                print(i, file_path)
                print(text)
        # Write file
        with open(file_path, 'w') as fp:
            for item in text_converted:
                fp.writelines("%s\n" % item)
```

```
92 data/obj/train/Datacluster-Auto--65-_jpg.rf.33a7b2023cc41d77c9d6ac79a30b8b1f.txt
0 0.505859375 0.255859375 0.09114583333333333 0.05078125
0 0.294921875 0.5517578125 0.033203125 0.02099609375

['0', '0.294921875', '0.5517578125', '0.033203125', '0.02099609375']
93 data/obj/train/Datacluster-Auto--43-_jpg.rf.f033008d31f044395a793c367a22e581.txt
```

## Making the list of names all the images :

```python
import glob
images_list = glob.glob("data/obj/train/*.jpg")
print(len(images_list))
print(images_list)
```

```
782
['data/obj/train/Datacluster-Auto--33-_jpg.rf.0b4800f8738b5510bbbfab898cb91bb3.jpg', 'data/obj/train/Datacluster-Auto--46-_jpg.rf.c4d40aa79cd458764ec6dce44
```

## creating the train.txt file for training the model:

```python
#Create training.txt file
file = open("data/train.txt", "w")
file.write("\n".join(images_list))
file.close()
```

# Training the model for 300 epochs :

```
# Start the training
!./darknet detector train data/obj.data cfg/yolov3_training.cfg darknet53.conv.74 -dont_show -epochs 300
```

```
v3 (mse loss, Normalizer: (iou: 0.75, obj: 1.00, cls: 1.00) Region 82 Avg (IOU: 0.610298), count: 3, class_loss = 1.760342, iou_loss = 0.223981, total_loss =
v3 (mse loss, Normalizer: (iou: 0.75, obj: 1.00, cls: 1.00) Region 94 Avg (IOU: 0.565712), count: 1, class_loss = 1.136141, iou_loss = 0.107346, total_loss =
v3 (mse loss, Normalizer: (iou: 0.75, obj: 1.00, cls: 1.00) Region 106 Avg (IOU: 0.000000), count: 1, class_loss = 0.805603, iou_loss = 0.000000, total_loss
 total_bbox = 14268, rewritten_bbox = 0.000000 %
v3 (mse loss, Normalizer: (iou: 0.75, obj: 1.00, cls: 1.00) Region 82 Avg (IOU: 0.396995), count: 4, class_loss = 2.124478, iou_loss = 0.772548, total_loss =
v3 (mse loss, Normalizer: (iou: 0.75, obj: 1.00, cls: 1.00) Region 94 Avg (IOU: 0.296509), count: 1, class_loss = 1.104689, iou_loss = 0.805412, total_loss =
v3 (mse loss, Normalizer: (iou: 0.75, obj: 1.00, cls: 1.00) Region 106 Avg (IOU: 0.000000), count: 1, class_loss = 0.806219, iou_loss = 0.000000, total_loss
 total_bbox = 14273, rewritten_bbox = 0.000000 %
v3 (mse loss, Normalizer: (iou: 0.75, obj: 1.00, cls: 1.00) Region 82 Avg (IOU: 0.516060), count: 3, class_loss = 1.774929, iou_loss = 0.755288, total_loss =
v3 (mse loss, Normalizer: (iou: 0.75, obj: 1.00, cls: 1.00) Region 94 Avg (IOU: 0.257452), count: 1, class_loss = 1.163162, iou_loss = 0.743407, total_loss =
v3 (mse loss, Normalizer: (iou: 0.75, obj: 1.00, cls: 1.00) Region 106 Avg (IOU: 0.000000), count: 1, class_loss = 0.838266, iou_loss = 0.000000, total_loss
 total_bbox = 14277, rewritten_bbox = 0.000000 %
v3 (mse loss, Normalizer: (iou: 0.75, obj: 1.00, cls: 1.00) Region 82 Avg (IOU: 0.562737), count: 4, class_loss = 2.010544, iou_loss = 0.831665, total_loss =
v3 (mse loss, Normalizer: (iou: 0.75, obj: 1.00, cls: 1.00) Region 94 Avg (IOU: 0.000000), count: 1, class_loss = 0.864711, iou_loss = 0.000000, total_loss =
v3 (mse loss, Normalizer: (iou: 0.75, obj: 1.00, cls: 1.00) Region 106 Avg (IOU: 0.000000), count: 1, class_loss = 0.826429, iou_loss = 0.000000, total_loss
 total_bbox = 14281, rewritten_bbox = 0.000000 %
v3 (mse loss, Normalizer: (iou: 0.75, obj: 1.00, cls: 1.00) Region 82 Avg (IOU: 0.608213), count: 4, class_loss = 1.981104, iou_loss = 0.442960, total_loss =
v3 (mse loss, Normalizer: (iou: 0.75, obj: 1.00, cls: 1.00) Region 94 Avg (IOU: 0.000000), count: 1, class_loss = 0.852525, iou_loss = 0.000000, total_loss =
v3 (mse loss, Normalizer: (iou: 0.75, obj: 1.00, cls: 1.00) Region 106 Avg (IOU: 0.000000), count: 1, class_loss = 0.809039, iou_loss = 0.000000, total_loss
 total_bbox = 14285, rewritten_bbox = 0.000000 %
v3 (mse loss, Normalizer: (iou: 0.75, obj: 1.00, cls: 1.00) Region 82 Avg (IOU: 0.750019), count: 2, class_loss = 1.548289, iou_loss = 0.229079, total_loss =
v3 (mse loss, Normalizer: (iou: 0.75, obj: 1.00, cls: 1.00) Region 94 Avg (IOU: 0.500924), count: 2, class_loss = 1.420185, iou_loss = 0.498734, total_loss =
v3 (mse loss, Normalizer: (iou: 0.75, obj: 1.00, cls: 1.00) Region 106 Avg (IOU: 0.000000), count: 1, class_loss = 0.821955, iou_loss = 0.000000, total_loss
 total_bbox = 14289, rewritten_bbox = 0.000000 %
v3 (mse loss, Normalizer: (iou: 0.75, obj: 1.00, cls: 1.00) Region 82 Avg (IOU: 0.260090), count: 4, class_loss = 2.078038, iou_loss = 1.769656, total_loss =
v3 (mse loss, Normalizer: (iou: 0.75, obj: 1.00, cls: 1.00) Region 94 Avg (IOU: 0.000000), count: 1, class_loss = 0.842986, iou_loss = 0.000000, total_loss =
v3 (mse loss, Normalizer: (iou: 0.75, obj: 1.00, cls: 1.00) Region 106 Avg (IOU: 0.000000), count: 1, class_loss = 0.795839, iou_loss = 0.000000, total_loss
 total_bbox = 14293, rewritten_bbox = 0.000000 %
v3 (mse loss, Normalizer: (iou: 0.75, obj: 1.00, cls: 1.00) Region 82 Avg (IOU: 0.369633), count: 4, class_loss = 2.020746, iou_loss = 1.091243, total_loss =
v3 (mse loss, Normalizer: (iou: 0.75, obj: 1.00, cls: 1.00) Region 94 Avg (IOU: 0.000000), count: 1, class_loss = 0.863868, iou_loss = 0.000000, total_loss =
v3 (mse loss, Normalizer: (iou: 0.75, obj: 1.00, cls: 1.00) Region 106 Avg (IOU: 0.000000), count: 1, class_loss = 0.802748, iou_loss = 0.000000, total_loss
 total_bbox = 14297, rewritten_bbox = 0.000000 %
```

# Testing the model:

## loading the yolo weights:
## the YOLO trained weights are saved as yolov3_training_last.weights which we have used to test our model

testing

```
[ ] # plot ALL results in test directory (NOTE: adjust figsize as you please)
    # for img in np_images:
    #     plt.figure(figsize=(8, 6))
    #     plt.imshow(img)
```

```
[ ] ## testing
```

```
import cv2
import numpy as np
import glob
import random
import matplotlib.pyplot as plt
```

```
[ ] from google.colab.patches import cv2_imshow
    # cv2_imshow(img)
```

```
[ ] # Load Yolo
    net = cv2.dnn.readNet("/content/drive/MyDrive/yolo_v3/yolov3_training_last.weights", "/content/drive/MyDrive/yolo_v3/yolov3_testing.cfg")

    # Name custom object
    classes = ["auto"]

    # Images path
    images_path = glob.glob(r"/content/drive/MyDrive/yolo_v3/testimages/*.jpg")
```

test image paths for testing our model:

```
images_path
```

```
['/content/drive/MyDrive/yolo_v3/testimages/Datacluster Auto (7).jpg',
 '/content/drive/MyDrive/yolo_v3/testimages/Datacluster Auto (79).jpg',
 '/content/drive/MyDrive/yolo_v3/testimages/Datacluster Auto (28).jpg',
 '/content/drive/MyDrive/yolo_v3/testimages/Datacluster Auto (27).jpg',
 '/content/drive/MyDrive/yolo_v3/testimages/Datacluster Auto (24).jpg',
 '/content/drive/MyDrive/yolo_v3/testimages/Datacluster Auto (23).jpg',
 '/content/drive/MyDrive/yolo_v3/testimages/Datacluster Auto (10).jpg',
 '/content/drive/MyDrive/yolo_v3/testimages/Datacluster Auto (101).jpg']
```

loading images for detection:

```
# Loading image
img = cv2.imread(img_path)
img = cv2.resize(img, None, fx=0.8, fy=0.8)
height, width, channels = img.shape
```

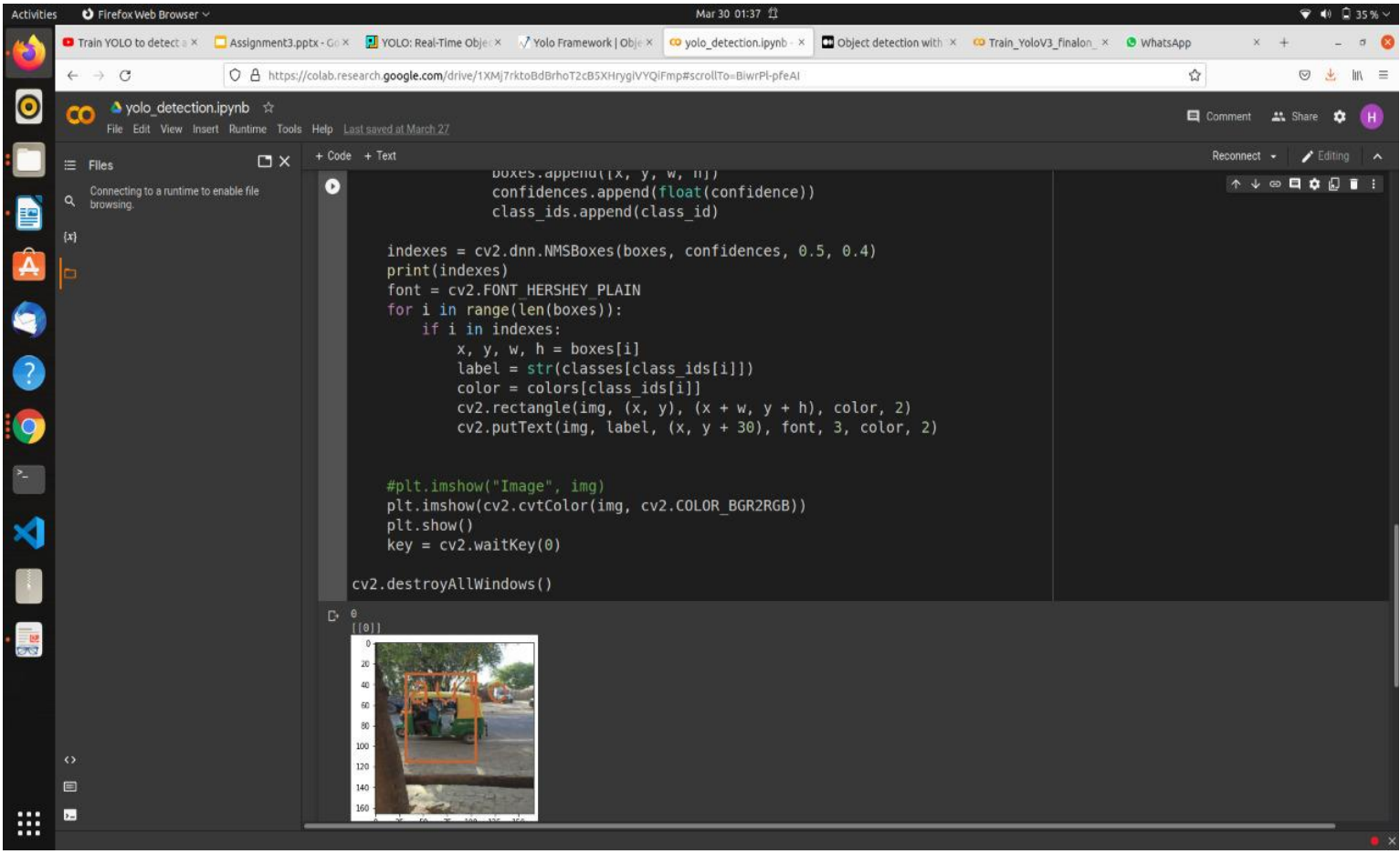
detecting objects and showing information on the screen:

```
# Detecting objects
blob = cv2.dnn.blobFromImage(img, 0.00392, (416, 416), (0, 0, 0), True, crop=False)

net.setInput(blob)
outs = net.forward(output_layers)

# Showing informations on the screen
class_ids = []
confidences = []
boxes = []
for out in outs:
    for detection in out:
        scores = detection[5:]
        class_id = np.argmax(scores)
        confidence = scores[class_id]
        if confidence > 0.3:
            # Object detected
            print(class_id)
            center_x = int(detection[0] * width)
            center_y = int(detection[1] * height)
            w = int(detection[2] * width)
            h = int(detection[3] * height)
```

inserting the rectangular co-ordinates in the image for each auto detected:

```
            h = int(detection[3] * height)

            # Rectangle coordinates
            x = int(center_x - w / 2)
            y = int(center_y - h / 2)

            boxes.append([x, y, w, h])
            confidences.append(float(confidence))
            class_ids.append(class_id)
```

displaying the output:




# Differences between YOLOV3, YOLOV4 and YOLOV5

| | YOLOv3 | YOLOv4 | YOLOv5 |
|---|---|---|---|
| Neural Network Type | Fully convolution | Fully convolution | Fully convolution |
| Backbone Feature Extractor | Darknet-53 | CSPDarknet53 | CSPDarknet53 |
| Loss Function | Binary cross entropy | Binary cross entropy | Binary cross entropy and Logits loss function |
| Neck | FPN | SSP and PANet | PANet |
| Head | YOLO layer | YOLO layer | YOLO layer |

**The main differences between YOLOv3, YOLOv4, and YOLOv5 architecture is that YOLOv3 uses Darknet53 backbone. YOLOv4 architecture uses CSPdarknet53 as a backbone and YOLOv5 uses Focus structure with CSPdarknet53 as a backbone. The Focus layer is first introduced in YOLOv5**

---

# Auto detection using YOL0 V5

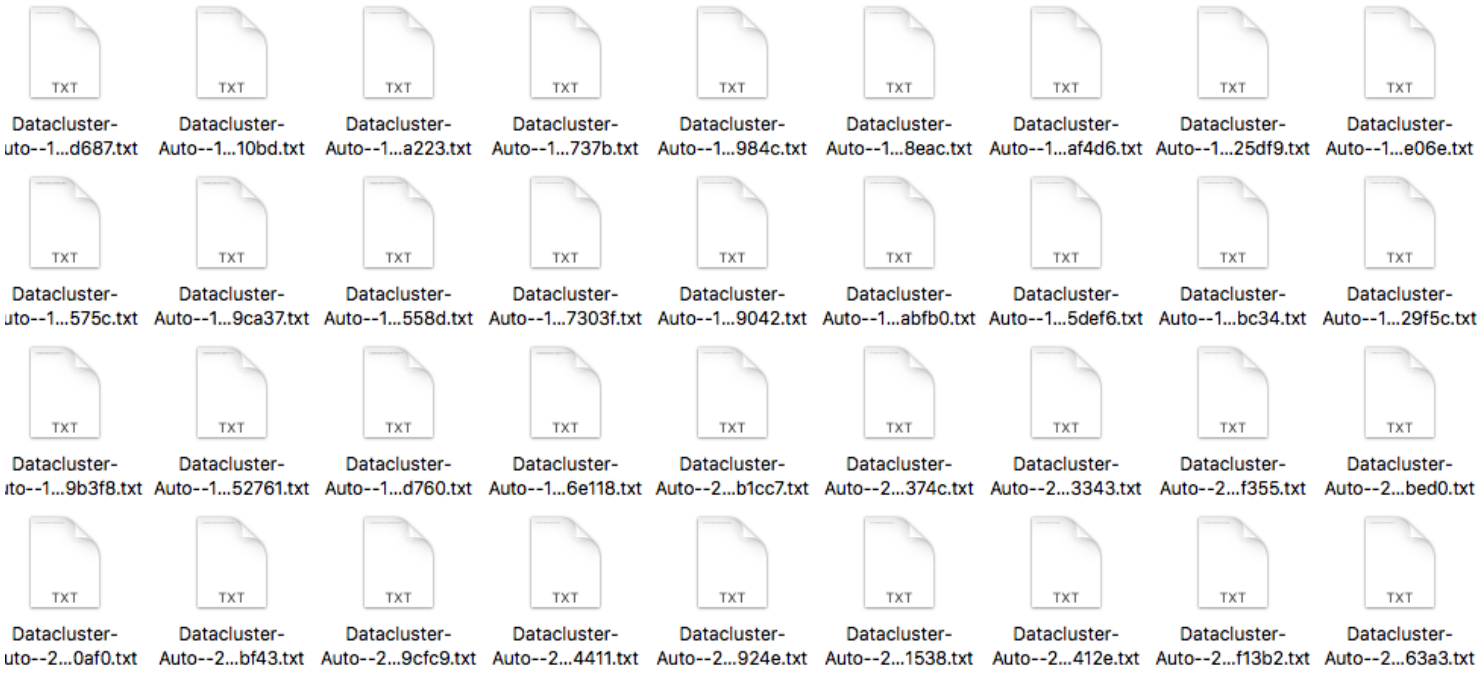
To train a YOLOv5 object detection model we'll training an auto detection model on our own dataset of auto using Colab and darknet

## Our Data:

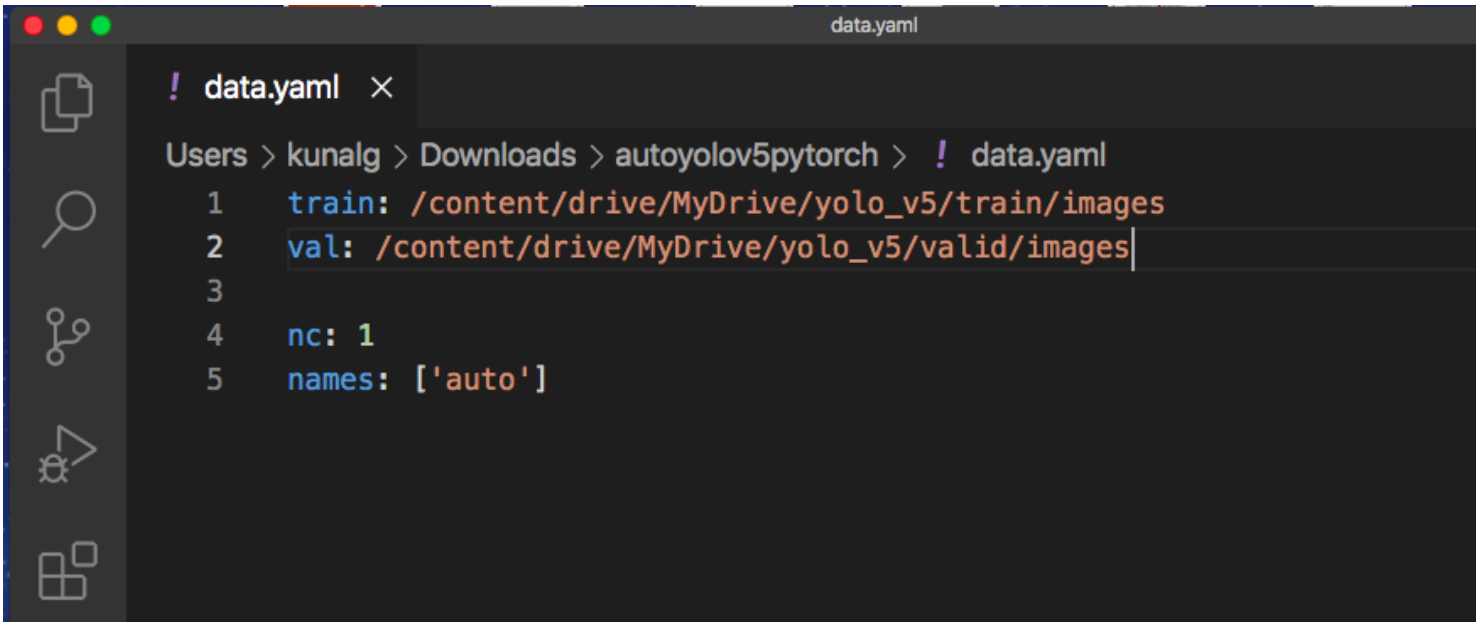
given Our dataset of 150 unannotated auto images.

## ANNOTATING THE IMAGES IN YOLOV5 FORMAT

Separating the images annotation .txt file in one place:




giving the path of train and test images in the yaml file :




we have taken the help from roboflow site for annotating our images on yolov5 dataset.

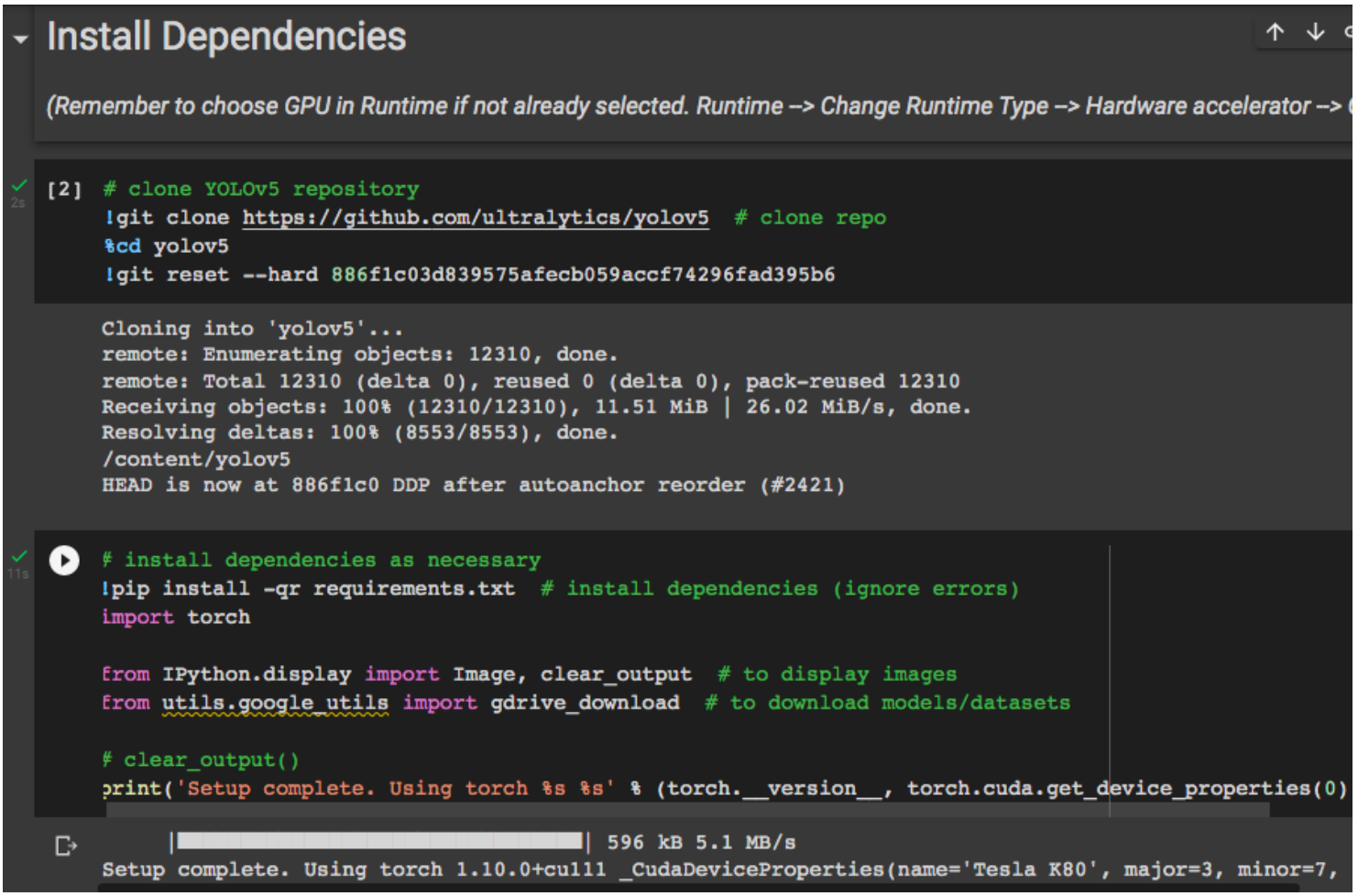
# YOLOv5 training

To train our detector we take the following steps:

* Install YOLOv5 dependencies
* Download custom YOLOv5 object detection data
* Write our YOLOv5 Training configuration
* Run YOLOv5 training
* Evaluate YOLOv5 performance
* Visualize YOLOv5 training data
* Run YOLOv5 inference on test images
* Export saved YOLOv5 weights for future inference

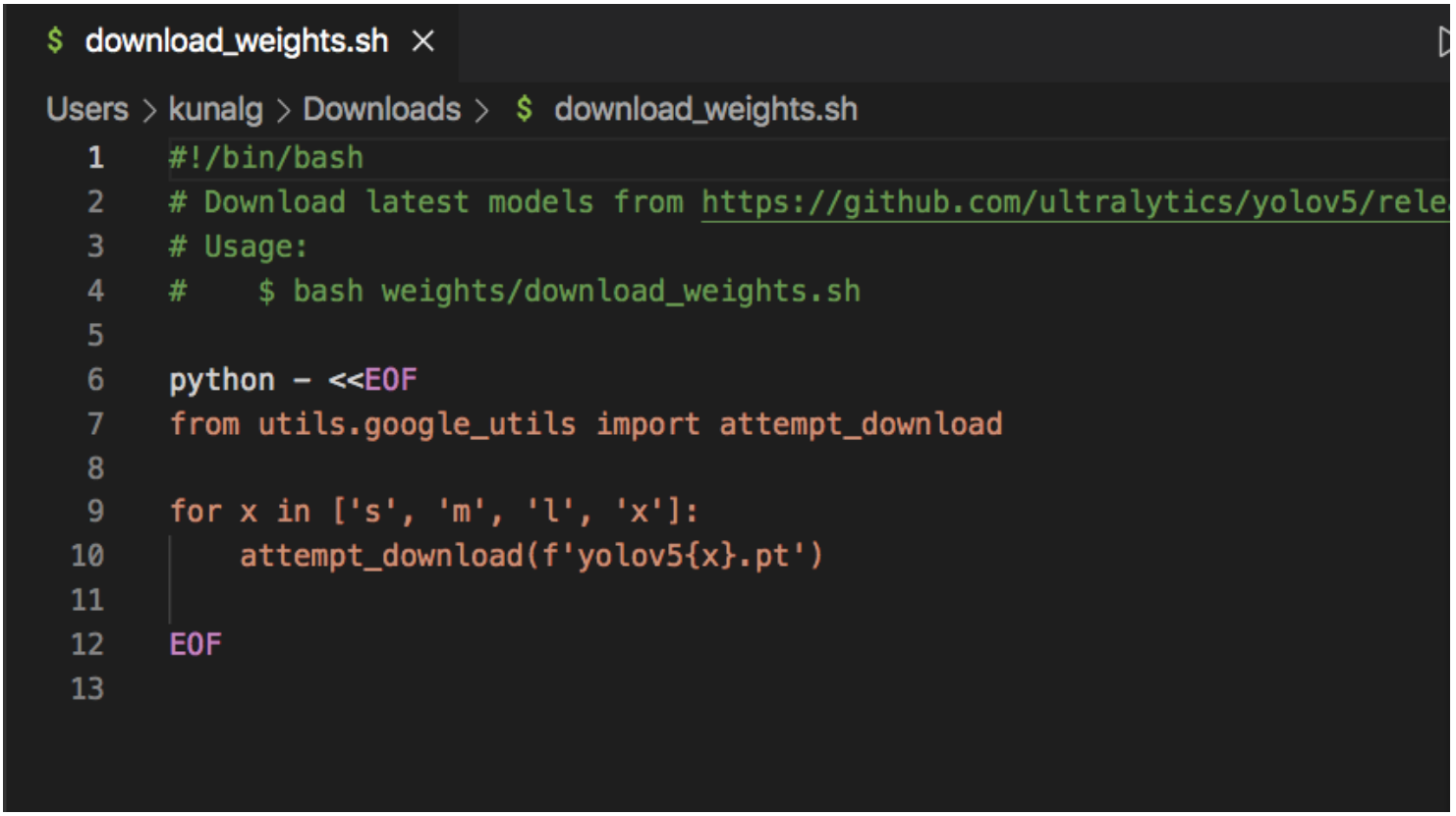
# INSTALLING YOLOV5  DEPENDENCIES

we have cloned the github repository and imported torch



## Download custom YOLOv5 object detection data

running a python script to automatically download yolov5 weights:



# Defining Model Configuration and Architecture

We will write a yaml script that defines the parameters for our model like the number of classes, anchors, and each layer.

```
[38] # define number of classes based on YAML
     import yaml
     with open("/content/gdrive/MyDrive/yolo_v5/data.yaml", 'r') as stream:
         num_classes = str(yaml.safe_load(stream)['nc'])
```

```
#this is the model configuration we will use for our tutorial
%cat /content/yolov5/models/yolov5s.yaml
```

```
# parameters
nc: 80  # number of classes
depth_multiple: 0.33  # model depth multiple
width_multiple: 0.50  # layer channel multiple

# anchors
anchors:
  - [10,13, 16,30, 33,23]  # P3/8
  - [30,61, 62,45, 59,119]  # P4/16
  - [116,90, 156,198, 373,326]  # P5/32

# YOLOv5 backbone
backbone:
  # [from, number, module, args]
  [[-1, 1, Focus, [64, 3]],  # 0-P1/2
   [-1, 1, Conv, [128, 3, 2]],  # 1-P2/4
   [-1, 3, C3, [128]],
```

# customizing the yaml file of yolov5 :

[6]
```
#customize iPython writefile so we can write variables
from IPython.core.magic import register_line_cell_magic

@register_line_cell_magic
def writetemplate(line, cell):
    with open(line, 'w') as f:
        f.write(cell.format(**globals()))
```

```
%%writetemplate /content/yolov5/models/custom_yolov5s.yaml

# parameters
nc: {num_classes}  # number of classes
depth_multiple: 0.33  # model depth multiple
width_multiple: 0.50  # layer channel multiple

# anchors
anchors:
  - [10,13, 16,30, 33,23]  # P3/8
  - [30,61, 62,45, 59,119]  # P4/16
  - [116,90, 156,198, 373,326]  # P5/32

# YOLOv5 backbone
backbone:
  # [from, number, module, args]
  [[-1, 1, Focus, [64, 3]],  # 0-P1/2
   [-1, 1, Conv, [128, 3, 2]],  # 1-P2/4
   [-1, 3, BottleneckCSP, [128]],
   [-1, 1, Conv, [256, 3, 2]],  # 3-P3/8
   [-1, 9, BottleneckCSP, [256]],
```

```
# YOLOv5 head
head:
  [[-1, 1, Conv, [512, 1, 1]],
   [-1, 1, nn.Upsample, [None, 2, 'nearest']],
   [[-1, 6], 1, Concat, [1]],  # cat backbone P4
   [-1, 3, BottleneckCSP, [512, False]],  # 13

   [-1, 1, Conv, [256, 1, 1]],
   [-1, 1, nn.Upsample, [None, 2, 'nearest']],
   [[-1, 4], 1, Concat, [1]],  # cat backbone P3
   [-1, 3, BottleneckCSP, [256, False]],  # 17 (P3/8-small)

   [-1, 1, Conv, [256, 3, 2]],
   [[-1, 14], 1, Concat, [1]],  # cat head P4
   [-1, 3, BottleneckCSP, [512, False]],  # 20 (P4/16-medium)

   [-1, 1, Conv, [512, 3, 2]],
   [[-1, 10], 1, Concat, [1]],  # cat head P5
   [-1, 3, BottleneckCSP, [1024, False]],  # 23 (P5/32-large)

   [[17, 20, 23], 1, Detect, [nc, anchors]],  # Detect(P3, P4, P5)
  ]
```

## Training Custom YOLOv5 Detector

Here, we are able to pass a number of arguments:
- **img:** define input image size
- **batch:** determine batch size
- **epochs:** define the number of training epochs.
- **data:** set the path to our yaml file
- **cfg:** specify our model configuration
- **weights:** specify a custom path to weights.
- **name:** result names
- **nosave:** only save the final checkpoint
- **cache:** cache images for faster training

```
from google.colab import drive
drive.mount('/content/drive')
```

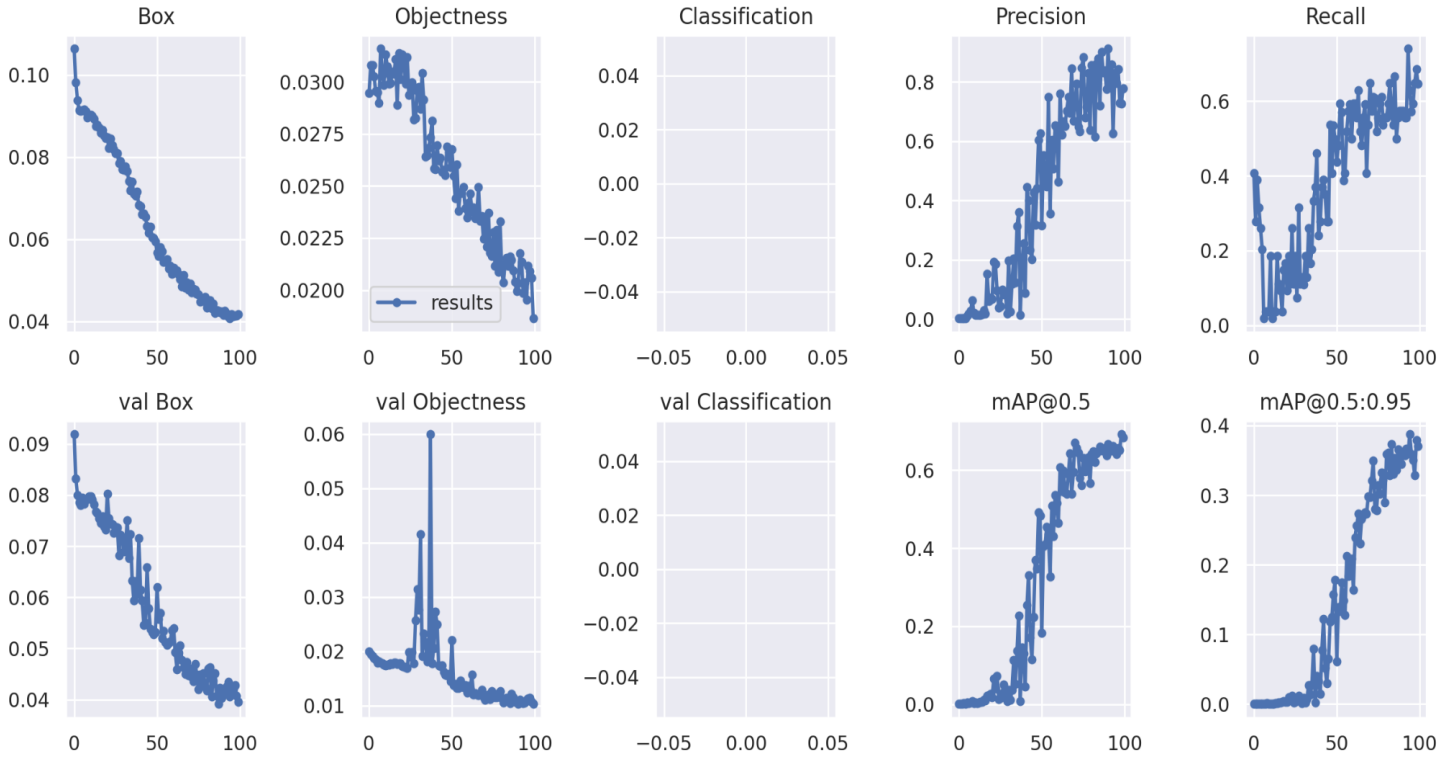
```
Mounted at /content/drive
```

+ Code + Text

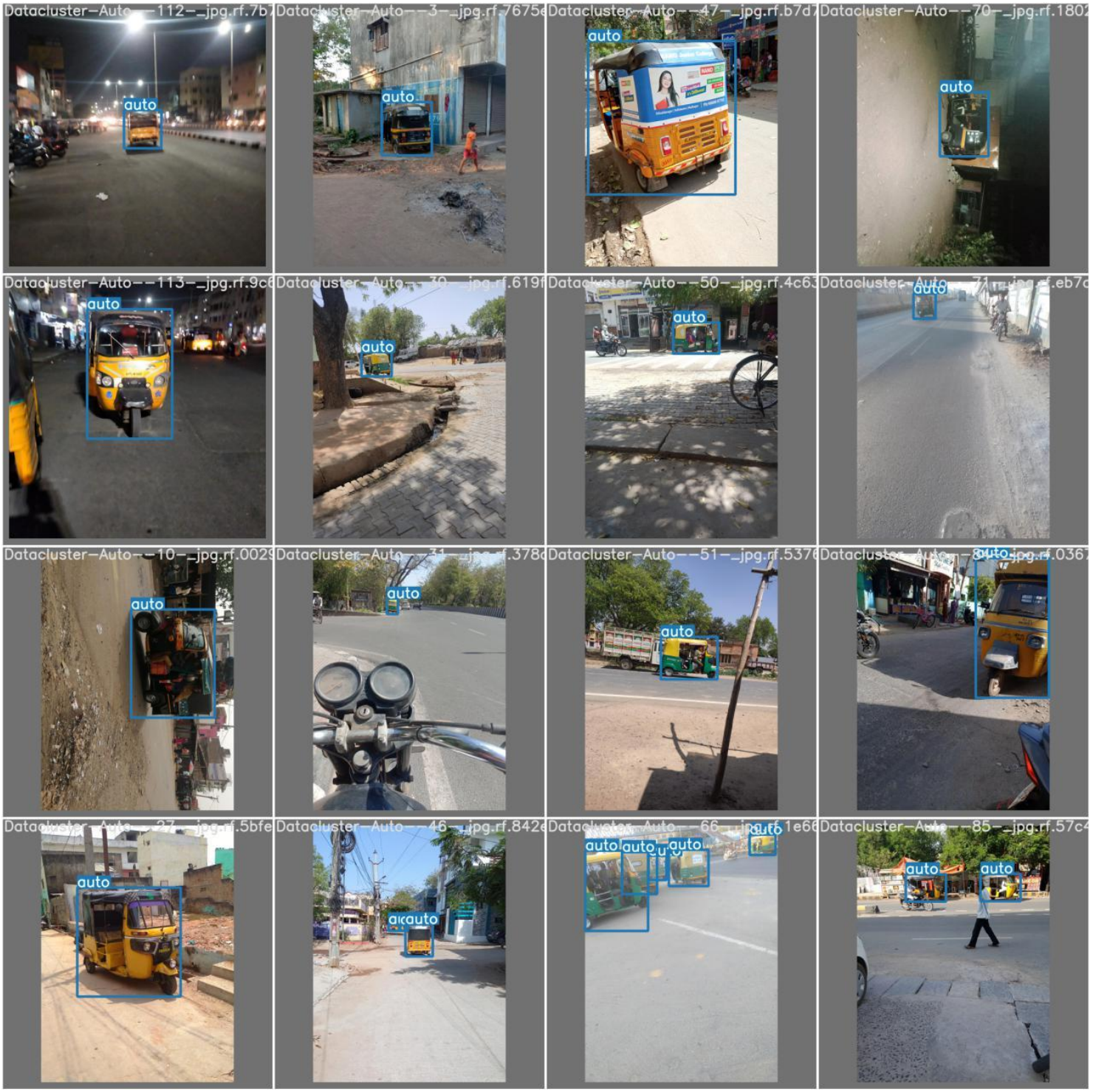
```
[23] # train yolov5s on custom data for 100 epochs
     # time its performance
     %%time
     %cd /content/yolov5/
     !python train.py --img 516 --batch 16 --epochs 100 --data "/content/drive/MyDrive/yolo_v5/data.yaml" --cfg ./models/c

     /content/yolov5
     github: ⚠️ WARNING: code is out of date by 934 commits. Use 'git pull' to update or 'git clone https://github.com/ul
     YOLOv5 v4.0-126-g886f1c0 torch 1.10.0+cu111 CUDA:0 (Tesla K80, 11441.1875MB)

     Namespace(adam=False, batch_size=16, bucket='', cache_images=True, cfg='./models/custom_yolov5s.yaml', data='/conten
     wandb: Install Weights & Biases for YOLOv5 logging with 'pip install wandb' (recommended)
     Start Tensorboard with "tensorboard --logdir runs/train", view at http://localhost:6006/
     hyperparameters: lr0=0.01, lrf=0.2, momentum=0.937, weight_decay=0.0005, warmup_epochs=3.0, warmup_momentum=0.8, warm

                     from  n    params  module                                  arguments
       0                -1  1      3520  models.common.Focus                     [3, 32, 3]
       1                -1  1     18560  models.common.Conv                      [32, 64, 3, 2]
       2                -1  1     19904  models.common.BottleneckCSP             [64, 64, 1]
       3                -1  1     73984  models.common.Conv                      [64, 128, 3, 2]
       4                -1  1    161152  models.common.BottleneckCSP             [128, 128, 3]
       5                -1  1    295424  models.common.Conv                      [128, 256, 3, 2]
       6                -1  1    641792  models.common.BottleneckCSP             [256, 256, 3]
       7                -1  1   1180672  models.common.Conv                      [256, 512, 3, 2]
```

```
WARNING: --img-size 516 must be multiple of max stride 32, updating to 544
train: Scanning '/content/drive/MyDrive/yolo_v5/train/labels.cache' for images and labels... 428 found, 0 missing,
train: Caching images (0.3GB): 100% 428/428 [00:16<00:00, 26.65it/s]
val: Scanning '/content/drive/MyDrive/yolo_v5/valid/labels.cache' for images and labels... 42 found, 0 missing, 0 e
val: Caching images (0.0GB): 100% 42/42 [00:02<00:00, 14.92it/s]
Plotting labels...

autoanchor: Analyzing anchors... anchors/target = 5.04, Best Possible Recall (BPR) = 1.0000
Image sizes 544 train, 544 test
Using 2 dataloader workers
Logging results to runs/train/yolov5s_results4
Starting training for 100 epochs...

     Epoch   gpu_mem       box       obj       cls     total   targets  img_size
      0/99     1.82G    0.1064   0.02947         0    0.1358        41       544: 100% 27/27 [00:43<00:00,  1.60s/i
                Class      Images     Targets           P           R      mAP@.5  mAP@.5:.95: 100% 2/2 [00:09<00:00
                  all          42          54     0.00175       0.407     0.00073    0.000114

     Epoch   gpu_mem       box       obj       cls     total   targets  img_size
      1/99     2.25G   0.09821   0.03081         0     0.129        29       544: 100% 27/27 [00:23<00:00,  1.14it/
                Class      Images     Targets           P           R      mAP@.5  mAP@.5:.95: 100% 2/2 [00:01<00:00
                  all          42          54     0.00239       0.278     0.00151    0.000239

     Epoch   gpu_mem       box       obj       cls     total   targets  img_size
      2/99     2.25G   0.09386   0.03081         0    0.1247        31       544: 100% 27/27 [00:23<00:00,  1.14it/
                Class      Images     Targets           P           R      mAP@.5  mAP@.5:.95: 100% 2/2 [00:01<00:00
                  all          42          54     0.00235       0.389     0.00171    0.000301
```

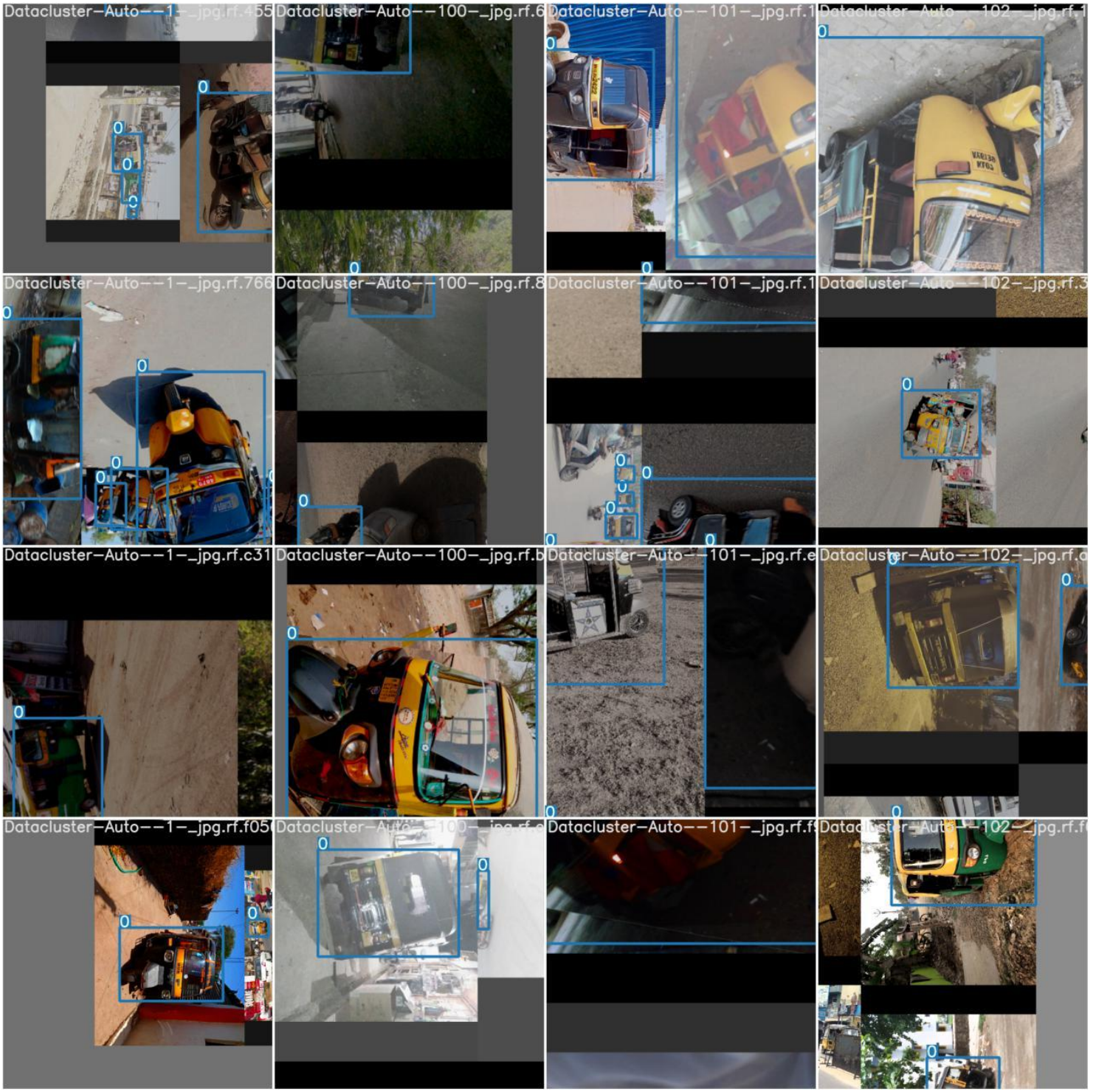
# Evaluate Custom YOLOv5 Detector Performance through tensorboard



# Visualizing our training model data with labels and augmentation



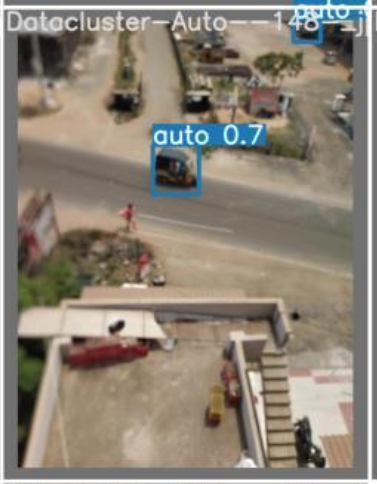
# training images augmentation by the yolov5 algorithm



# testing data :


auto 0.5
auto 0.7
auto 0.4


Datacluster-Auto--67-_jpg.rf.2bd54d50d99


auto 0.7

auto 0.6
Datacluster-Auto--129-_jpg.rf.bf03138dae

auto 0.7

auto 0.8

auto 0.3

auto 0.7
Datacluster-Auto--93-_jpg.rf.23b5b2fc587

# Running inferences with trained weights

Run inference with a pretrained checkpoint on contents of `test/images` folder downloaded from Roboflow.

```
[28] # trained weights are saved by default in our weights folder
     %ls runs/

     detect/  train/
```

```
%ls /content/yolov5/runs/train/yolov5s_results4/weights
```

```
best.pt  last.pt
```

```
[31] 007 second inference time. That is 140 FPS on a TESLA P100!


     /content/yolov5/runs/train/yolov5s_results4/weights/best.pt --img 84 --conf 0.4 --source "/content/drive/MyDrive/yolo

     /content/yolov5
     Namespace(agnostic_nms=False, augment=False, classes=None, conf_thres=0.4, device='', exist_ok=False, img_size=84, ic
     YOLOv5 v4.0-126-g886f1c0 torch 1.10.0+cu111 CUDA:0 (Tesla K80, 11441.1875MB)

     Fusing layers...
     /usr/local/lib/python3.7/dist-packages/torch/functional.py:445: UserWarning: torch.meshgrid: in an upcoming release,
       return _VF.meshgrid(tensors, **kwargs)  # type: ignore[attr-defined]
     Model Summary: 232 layers, 7246518 parameters, 0 gradients, 16.8 GFLOPS
     WARNING: --img-size 84 must be multiple of max stride 32, updating to 96
```

# Displaying inferences on the test data:



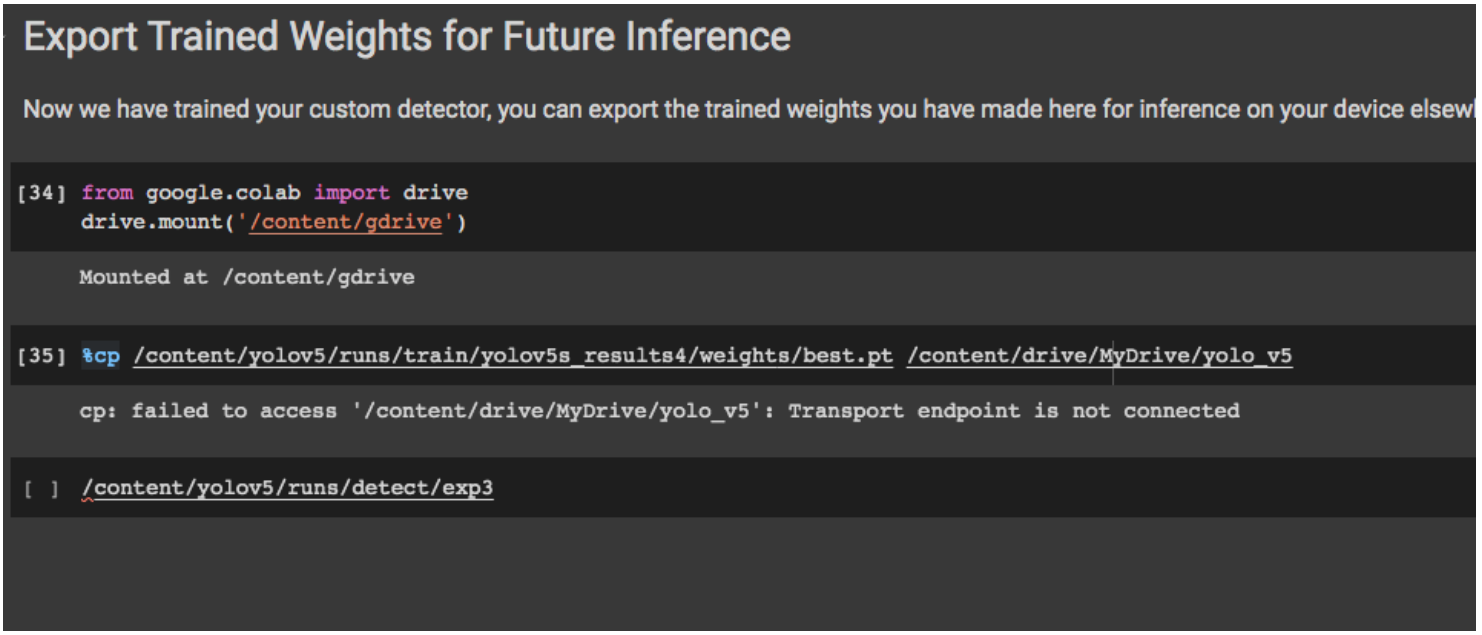
auto 0.63


auto 0.47
DECORE
CAPITAL
इन्दौर सीट
कवर डेकोरेश
M.B. CHINESE
FOOD STALL


auto 0.62


auto 0.53

# *Exporting trained weights for future references*



# Applications of YOLO

**YOLO algorithm can be applied in the following fields:**

- Autonomous driving: YOLO algorithm can be used in autonomous cars to detect objects around cars such as vehicles, people, and parking signals. Object detection in autonomous cars is done to avoid collision since no human driver is controlling the car.

- Wildlife: This algorithm is used to detect various types of animals in forests. This type of detection is used by wildlife rangers and journalists to identify animals in videos (both recorded and real-time) and images. Some of the animals that can be detected include giraffes, elephants, and bears.

Security: YOLO can also be used in security systems to enforce security in an area. Let's assume that people have been restricted from passing through a certain area for security reasons. If someone passes through the restricted area, the YOLO algorithm will detect him/her, which will require the security personnel to take further action.

# Where YOLO detection can not be used

## Disadvantages of YOLO:

- **Comparatively low recall and more localization error compared to Faster R_CNN.**
- **Struggles to detect close objects because each grid can propose only 2 bounding boxes.**
- **Struggles to detect small objects.**