VISUAL RECOGNITION
Mini Project 1

Samaksh Dhingra (IMT2019075)
Ankit Agrawal (IMT2019010)
Lovejeet Singh Parihar (IMT2019048)

March 30, 2022

A. Task 3a: Activation Functions, Momentum and CNNs

Comparing Tanh, ReLU and Sigmoid

Some of the most popular activation functions used in Neural Networks (and CNNs also) are:

1. Tanh activation function: The Tanh activation function is a popular activation function which is continuous and differentiable in nature. Its expression is given by:

$$Tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Where e is the Euler's number and approximately equals 2.718.

The plot of the above function is also shown below:

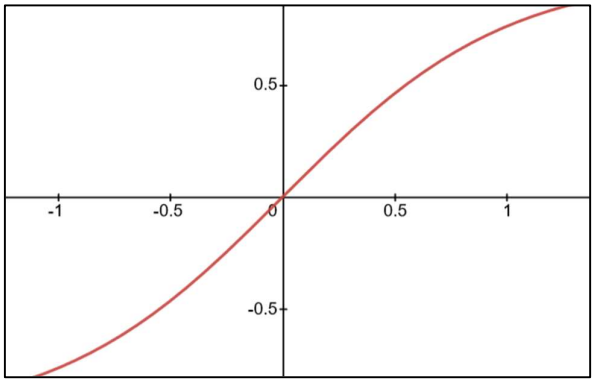


Fig. 1: The Tanh activation function

The Tanh function is 0 at $x = 0$ and its values range from -1 to 1.

2. ReLU activation function: The ReLU activation function is one of the most widely used activation functions while training Neural Networks. Here, ReLU stands for Rectified Linear Unit and its expression is given by:

$$ReLU(x) = \max(0, x)$$

Clearly, the above function is discontinuous at $x = 0$. This is also clearly visible when we plot the graph of the ReLU activation function as shown in the next page…

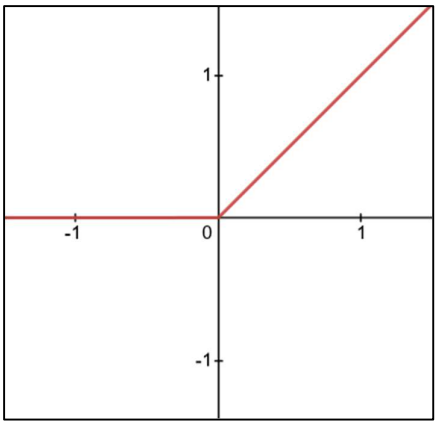


Fig. 2: The ReLU activation function

3. Sigmoid activation function: The Sigmoid activation function was a popular choice for the activation function but has now been replaced by ReLU and other activation functions. Its expression is:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Where e is the Euler's number and approximately equals 2.718.

The plot of the sigmoid activation function is also shown below:

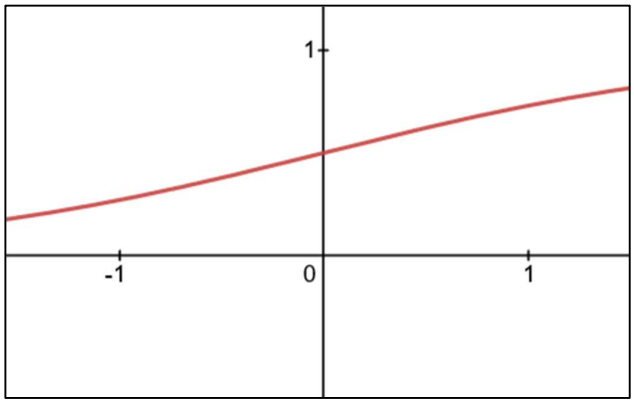


Fig. 3: The Sigmoid activation function

Now, let us see how these activation functions stack up against each other when used in a Convolutional Neural Network (or CNN).

In order to test these activation functions, a CNN was created which contains 2 Convolutional Layers and 2 Fully Connected Layers. The activation functions under test were used in between these layers. Finally, these activation functions

were compared on the basis of two metrics: **the training time** and the **classification performance** on the CIFAR-10 dataset.

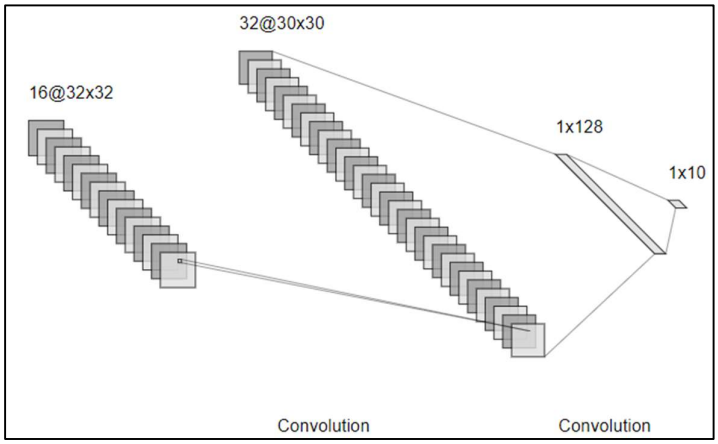
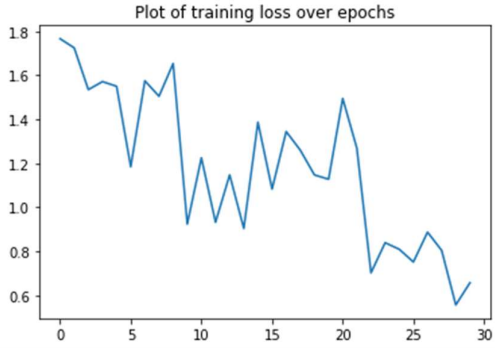
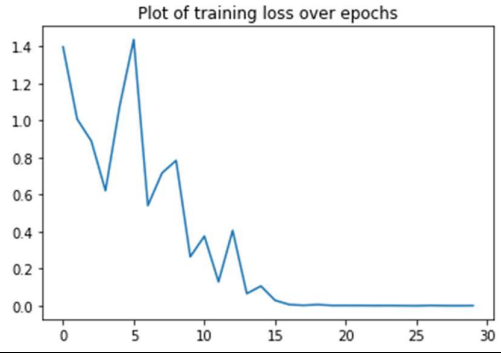
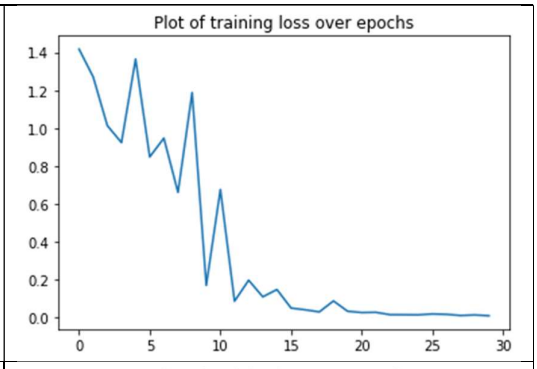
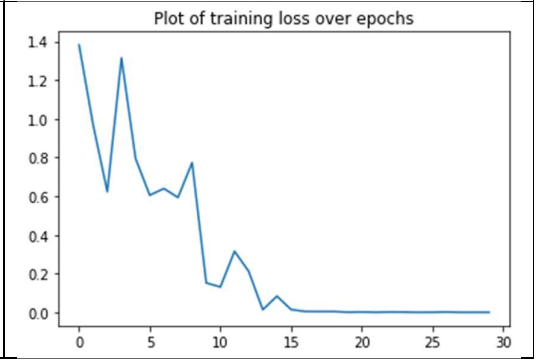


Fig. 4: The CNN architecture. The activation functions are present between each convolution and full connected layers.

The results of obtained are presented in the table below:

Activation Function	Training Time (in sec.)	Accuracy on Test Dataset	Plot showing Loss v/s Epoch count on Train Dataset
Sigmoid	588.94	61.86 %	
ReLU	590.67	66.30 %	

Tanh	588.67	66.01 %	
Leaky ReLU	588.24	67.39 %	

The training was conducted over **30** epochs with a batch size of **32** and the learning rate being **0.01**. Also, training was done over GPUs that are available on Kaggle.

As shown in the table above, we additionally also tested the Leaky ReLU activation function, whose expression is given by:

$$\text{LeakyRELU}(x) = \begin{cases} x, & \text{if } x \geq 0 \\ \text{negative\_slope} \times x, & \text{otherwise} \end{cases}$$

In our case, the value of negative_slope was **0.01** (which is the default value set by PyTorch).

Based on the results obtained, we can draw in the following conclusions:

1. The training time is almost the same for all the activation functions. This shows that the choice of activation functions does not affect the training time, which should hold true intuitively because we require only constant

amount of time to calculate the value of the activation function for a given input.

2. The plot showing the training loss when the Sigmoid activation function is used is pretty shaky in nature. This shows that we might need more epochs in order for our model to converge when using the Sigmoid activation function.

3. Leaky ReLU gives the best results, followed by ReLU and Tanh activation functions. One of the advantages of using Leaky ReLU over ReLU is that it helps to overcome the vanishing gradient problem, which might help the model train better.

4. It is also worth noting that models using ReLU and Leaky ReLU converge after approximately 15 epochs, while model using Tanh converges after approximately 20 epochs, while the model using Sigmoid activation function has not yet converged.

So, in practice, one would like to go with either the Leaky ReLU or the ReLU activation function while creating and training the model.

Learning with Momentum and Adaptive Learning Rates

We would like to make our models learn faster and better. So, how do we do that? One of the many ways of doing that is to use Momentum and Adaptive Learning Rates while training our model. These terms are explained below:

Momentum: It is an extension of the traditional Stochastic Gradient Descent algorithm. It allows the search to build up inertia in a direction in the search space and overcome noisy gradients and flat spots of the search space [1].

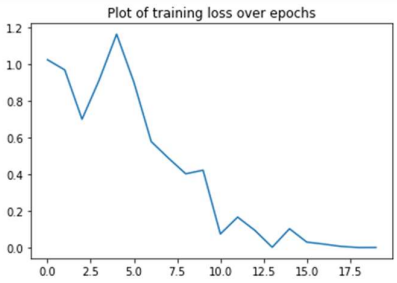
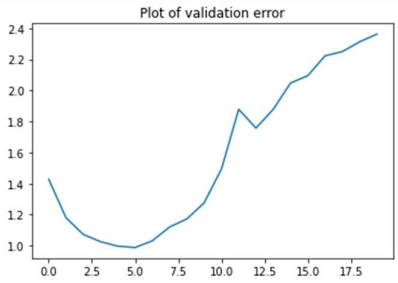
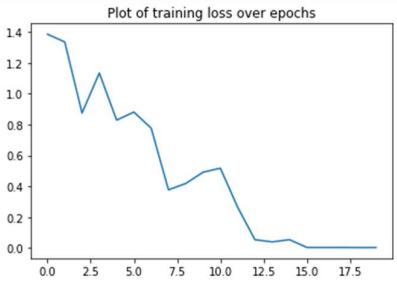
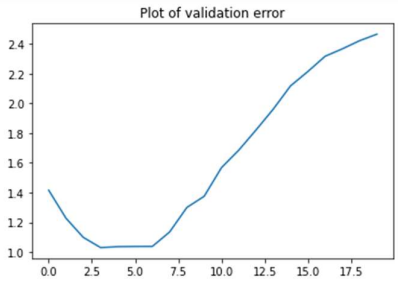
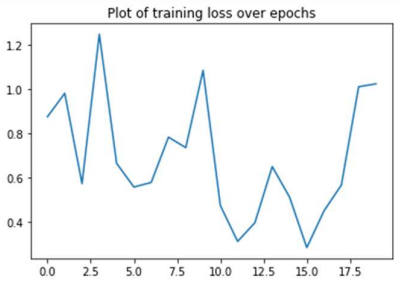
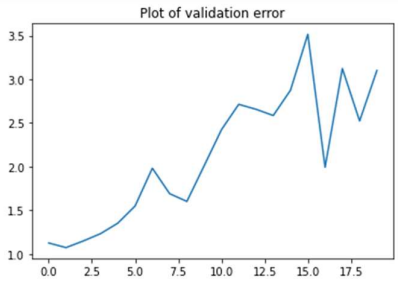
Adaptive Learning Rates: Adaptive Learning Rates enable the training algorithms to adjust the speed of learning of a model. This can be very useful particularly when we would like to avoid slow learning. Some examples of

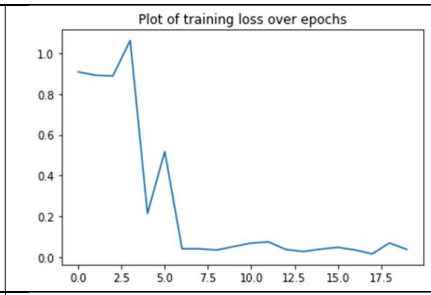
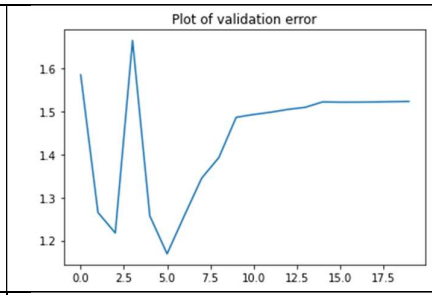
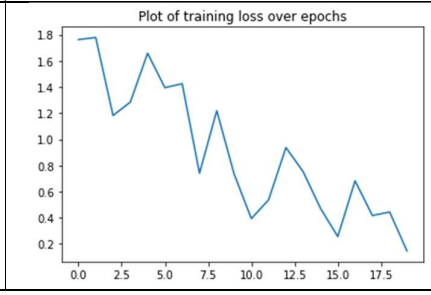
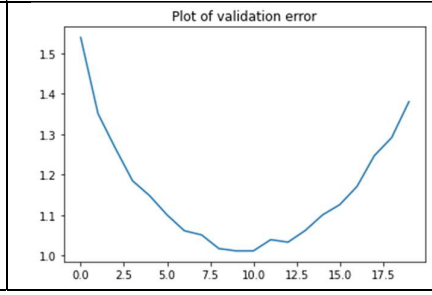
[1] https://machinelearningmastery.com/gradient-descent-with-momentum-from-scratch/

optimization algorithms that implement adaptive learning rates are **AdaGrad, Adam** and the **RMSprop** algorithm.

In order to compare the performances with and without momentum, we used a CNN that is very similar to the one shown in Fig. 4, except with the use of a Max-Pooling layer after CNN Layer 2 (having kernel size = 2 and stride = 2). Further, the optimization process was done for a total of 20 epochs.

Also, in order to better see how the model improves with different momentum values, we used the test dataset as the validation dataset. The results found are summarized in the table below:

Momentum value	Optimizer Used	Training Loss v/s Epoch Count Plot	Validation Loss v/s Epoch Count Plot
0.0	SGD		
0.2	SGD		
0.9	SGD		

0.0	SGD with Adaptive Learning Rate Adjustment		
-	Adam		

The table showing the training time and the final testing error (after **20** epochs) is as follows:

Momentum value	Optimizer Used	Training Time (in sec.)	Accuracy after **20** epochs
0.0	SGD	301.86	66.76 %
0.2	SGD	308.87	65.91 %
0.9	SGD	314.72	57.28 %
0.0	SGD with Adaptive Learning Rate Adjustment	286.01	68.65 %
-	Adam	288.79	64.79 %

Based on the results obtained, we can draw the following conclusions:

1. Higher value of momentum tends the model to learn quickly. However, this can be problematic as we might make too large jumps and miss desired minima in the loss function. This is evident from the comparison between SGD with momentum **0.2** and SGD with momentum **0.9**, where the former converges slowly but steadily.

2. The Adam optimizer converges relatively slowly as compared to other optimizers, however the way it converges is more stable as compared to other optimizers.

3. We can say that for reaching the optimal model, different optimizers (with different momentum values) take different epochs and hence different amount of time to reach the optimal model.

4. The best way to use momentum is to set the momentum value not too low (and not too high either) and adjust it based on how the model is getting trained to get the best performance.

Learning with Momentum and Adaptive Learning Rates

The CNN architecture that we tested out had the following structure:

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1          [-1, 32, 30, 30]             896
         LeakyReLU-2          [-1, 32, 30, 30]               0
            Conv2d-3          [-1, 64, 28, 28]          18,496
         LeakyReLU-4          [-1, 64, 28, 28]               0
         MaxPool2d-5          [-1, 64, 14, 14]               0
            Conv2d-6          [-1, 32, 12, 12]          18,464
         LeakyReLU-7          [-1, 32, 12, 12]               0
            Conv2d-8          [-1, 16, 10, 10]           4,624
         LeakyReLU-9          [-1, 16, 10, 10]               0
        MaxPool2d-10            [-1, 16, 9, 9]               0
           Linear-11                 [-1, 128]         166,016
        LeakyReLU-12                 [-1, 128]               0
           Linear-13                  [-1, 10]           1,290
================================================================
Total params: 209,786
Trainable params: 209,786
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.01
Forward/backward pass size (MB): 1.41
Params size (MB): 0.80
Estimated Total Size (MB): 2.22
----------------------------------------------------------------
```

Fig. 5: Summary of the first CNN model used.
Summary generated using the torchsummary module

Our CNN model has a total of 4 Convolution layers with 2 Fully Connected layers. Since we don't have many Fully connected layers, the size of the model is small (~2.22 MB). In order to train the CNN, we followed the below steps:

1. We augmented the dataset to create more images. By augmentation, the final training dataset containing around 3,00,000 images. Augmenting images included:
    a. Flipping images horizontally
    b. Randomly rotating images

    The data augmentation was done with the help of the albumentations module.

2. Using a batch size of 128, epochs to be 10 and learning rate to be 0.001, the model was trained with the help of Adam optimizer. The learning rate was also decreased when a certain number of epochs were completed to slow down the learning and make convergence better.

3. The final accuracy obtained was 73.26 %, which is around 5 % better than the results described earlier.

A plot showing the training loss (across epochs) is shown below:

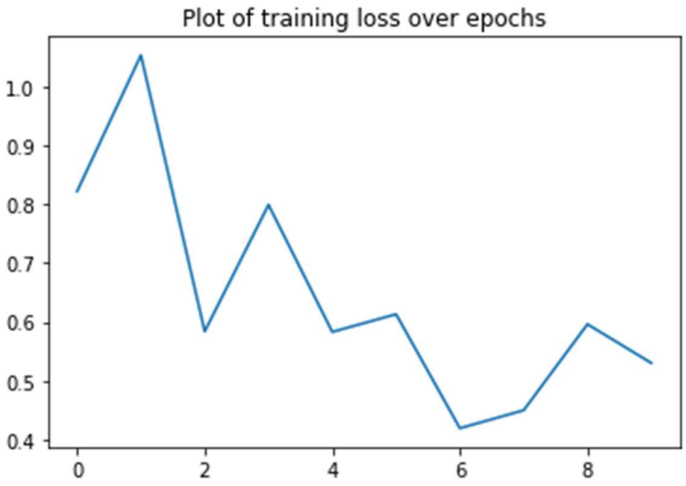


Fig. 6: Plot of training loss over epochs

References used: https://blog.paperspace.com/writing-cnns-from-scratch-in-pytorch/
https://albumentations.ai/docs/getting_started/image_augmentation/

Part 3B

CNN as a feature detecter

Modules used:-

- Pytorch
- numpy
- os

Pre trained CNNs, with last fully connected layer removed, were used to get features out from the pictures. The extracted features and labels were then fed into classification models of SVC, Logistic regression and K-Nearest Neigbours to get predictions.

Classification 1 - Bike vs Horse Dataset

- In this dataset, there were 80 bike images and 99 horse images.
- The train-test split had 80 percent as train data and 20 percent as test data.
- The pretrained CNN used for this dataset was Resnet18. After removing the last layer, it generated 512 features for each picture.

- The following models were used for classification and the following accuracies were obtained:-
    - Linear SVC - Accuracy of 100%
    - Logistic Regression - Accuracy of 100%
    - K-nearest neighbours - With k = 5, Accuracy of 100%

Classification 2 - Airplane vs Car vs Cat vs Dog Dataset

- In this dataset, there were 727 airplane images, 968 car images, 885 cat images and 702 dog images.
- The train-test split had 80 percent as train data and 20 percent as test data.
- The pretrained CNN used for this dataset was Resnet101. After removing the last layer, it generated 2048 features for each picture.
- The following models were used for classification and the following accuracies were obtained:-
    - Linear SVC - Accuracy of 98.63%
    - Logistic Regression - Accuracy of 98.63%
    - K-nearest neighbours - With k = 15, Accuracy of 97.42%

Part C. Auto Detection Using YOLO Method (Feature Learning)

Task: Build an "Auto" detector by training YOLO model on a custom dataset with "Auto" images.

Given: Dataset containing 157 "Auto" images.

Implementation

Google Colab Environment (Runtime with GPU) is used for this part.

a. Adding more images to given dataset

For rigorous training and testing, more images are added to the current given dataset. Procedure followed for addition of images, so that maximum edge cases get covered, is as follows:-

Some images from given dataset are chosen randomly. 2 copies are made for each of the chosen image. Orientation of image is changed for one copy and colour brightness for the other copy.

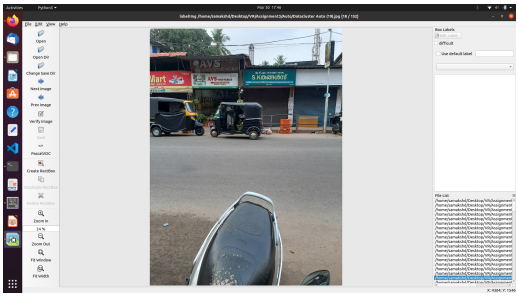
After following this procedure, we got a set of 355 images in our dataset.

b. Annotating the dataset obtained

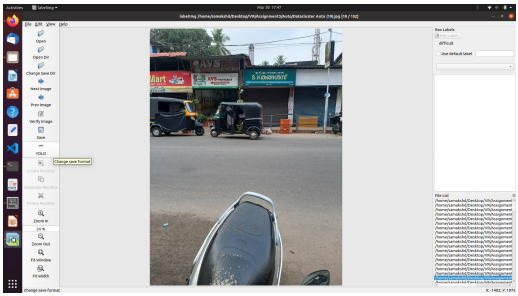
YOLO requires annotated dataset for working on it. Given dataset in annotated in YOLO annotation format, i.e. text file for each image with each line containing information (center x, center y, width, height) of bounding box(es) around auto(s) in the image.

We annotated the dataset manually using tool in linux called 'LabelImg'. Following is the procedure followed for annotation of an image:-

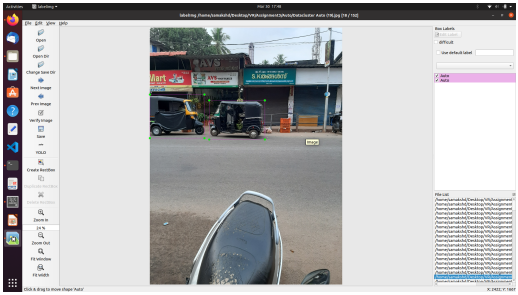
(a) Load the image in 'LabelImg'



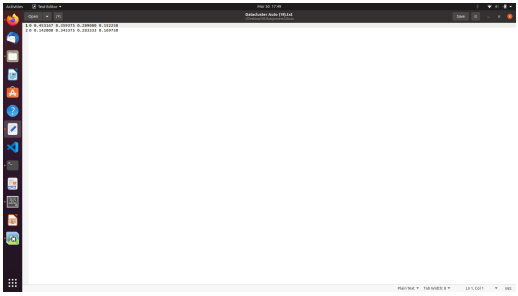
(b) Select the output as "YOLO" formatting



(c) Draw tight bounding boxes around all the autos in the given image



(d) Save the annotation text file for that image



This procedure is repeated for all the images in the given dataset and their annotation text files are stored.

c. Setting up YOLOv5

There are multiple methods to use YOLOv5:-

(a) Loading it from Pytorch

(b) Cloning its Github Repository

We have used 2nd method for this assignment. All the requirements for it are installed using 'pip'.

d. Loading the Annotated Dataset

The given dataset is loaded and stored as list of 'images' and 'annotations'.

e. Splitting the Dataset in Train-Validation-Test Sets

The list of images and annotations are splitted into training, validation and testing sets. These sets are then used to split the given full dataset into training, validation and testing dataset directories. Basically these directories are required by YOLO for training, validation and testing.

f. Training YOLOv5 Model on the given dataset

YOLOv5 Model is trained with respective flag values (will be discussed in Experimentation section). Mean Average Precision of validation set is calculated during each Epoch.

g. Testing trained model on Test Images

Detection

For all the images in testing set, bounded box for autos are detected using the YOLO trained model.

Metrics

Mean Average Precision is calculated for the Test Images.

Experimentation Results and Observations

Mean Accuracy Precision (mAP) on Validation Set (Last Epoch): 93.7%

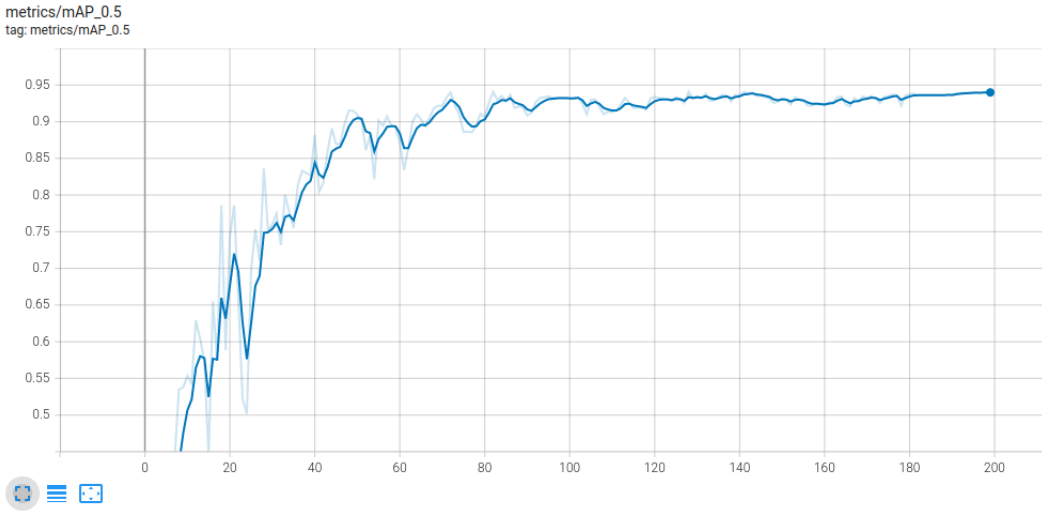
Mean Accuracy Precision (mAP) on Test Set (Best): 90.8%

After Experimentation, Flags for training the YOLO Model on Training Set were kept as follows:-

- img-size: 500 (500 × 500)
- batch size: 64
- Number of Epochs: 200
- Weights: yolov5s.pt

Experiment: Number of Epochs for training YOLO Model
Observations: Increasing the number of Epochs for training gave much better results in terms of Mean Accuracy Precision.

Mean Average Precision on Validation Set V.S. Number of Epochs



Performance of YOLO based Auto Detector

Where YOLO fails (Limitations of YOLO)

- YOLO imposes strong spatial constraints on bounding box predictions since each grid cell only predicts two boxes and can only have one class. This spatial constraint limits the number of nearby objects that our model can predict. You can clearly see in one of the test images detected by our model, the two autos situated very close to a detected auto (0.96) are failed to be detected.

- Our model struggles with small objects that appear in groups. You can clearly see in one of the test images detected by our model, the autos situated as small images and in groups are failed to be recognized properly.

- Since our model learns to predict bounding boxes from data, it struggles to generalize to objects in new or unusual aspect ratios or configurations. You can clearly see in one of the test images detected by our model, detection accuracy for the auto in an unusual aspect ratio and orientation is fairly less.

Where YOLO works well (Advantages of YOLO)

After seeing the overall performance of YOLO on Test Images, we can say that it works pretty well.

- Speed of YOLO is its biggest advantange. It gave accuracy of 90% with approximately 300 images in training set within only half an hour.
- Network understands generalized object representation. This allowed them to train the network on real world images. You can clearly see in one of the test images detected by our model, which is quite blur but still our model predicted it with fairly well accuracy.

References

YOLO Documentation:-
https://docs.ultralytics.com/

Implementation:-
https://blog.paperspace.com/train-yolov5-custom-data/

YOLOv5 Github Repository:-
https://github.com/ultralytics/yolov5

LabelImg (For annotating dataset images):-
https://pypi.org/project/labelImg/

Benefits and Limitations of YOLO
https://towardsdatascience.com/yolo-you-only-look-once-real-time-object-de