# AI825 Term 2 2021-22 Visual Recognition

# Graded Assignment 3 / Mini Project

# CNN & Auto Detection


## Team Members

*Adrij Sharma – IMT2019004*
*Rohan Thakkar – IMT2019071*

# Problem 1
# Image Classification on CIFAR-10 dataset using Convolutional Neural Network (CNN)

## Introduction to CNN & CIFAR

CNN model was learnt on CIFAR-10 dataset for image classification using pytorch library of python. Activation functions such as ReLU, tanh and sigmoid were tried with various combinations of convolutional layers and fully connected layers. In the pytorch documentations a good example of training a classifier has been shown.

## About CIFAR10 Dataset

The CIFAR-10 dataset consists of 60000 32X32 color images. These 6000 images are from 10 classes with 6000 images in each class. The 10 classes in the dataset are airplane, automobile, bird, cat, deer, dog, frog, horse, ship and truck. For our classification purpose, these 60000 images are divided into train and test set with 50000 images in the train set and 10000 images in the test set.

## About Convolutional Neural Network (CNN)

CNN is a deep learning algorithm which is most commonly applied to analyzing visual imagery. While in traditional methods the image features are hard engineered, CNN provides us the liberty to give input as the image as it is and learn the features/characteristics of the image on its own. CNN are composed of multiple layers of artificial neurons. The output of one layer of CNN is fed as the input to the next one. The first layer of CNN usually detects basic features such as horizontal or vertical edges. As you move deeper the layer starts detecting the higher-level features such as objects, faces. The layers in CNN can be convolutional layers, pooling layers and fully-connected layers.

## Approach Explained in Detail:

Firstly, we decided on an architecture wherein we analyzed the original Alexnet. Since the original Alexnet neural network works on 256x256 image, we had to reduce the convolutional layer sized since the inputs where of 32x32. My architecture is as follows initially.

- 2x2 Convolutional layer, stride = 2, and no of filters = 96.
- Maxpool layer of 2x2 size with stride = 2
- 2x2 Convolutional Layer, stride = 1, no of filters = 256
- Maxpool layer of 2x2 size with stride = 1
- Linear Layer with output of size 1000
- Linear layer with output size 120
- Linear layer with output size 84
- Linear layer with output size 10
- Softmax layer

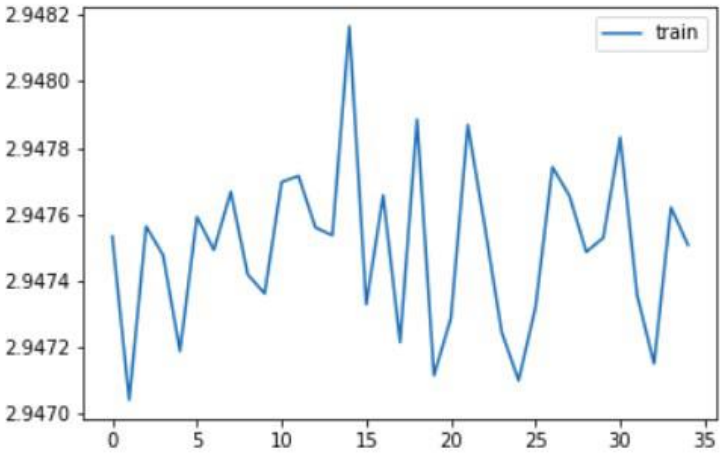
All layers have relu activation layers in between them. This has been trained with SGD optimizer and cross entropy loss. we have trained this on Kaggle notebook on GPU with batch size of 4 and for 20 epochs. It gave an accuracy of 69% on the test data of CIFAR10 dataset on 10000 images. The training time was around 15 mins on the training data.

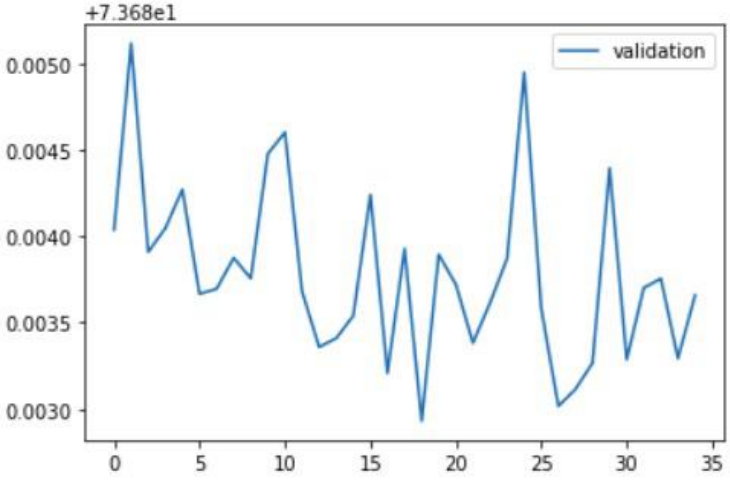
Now, in the coming days we am going to work with different activations and optimizers and measure the training time and classification performance. So, now we started to work with different activations first. we tried all three activations ReLU, Sigmoid and tanh activations. The results are below.

we compared the classification time vs the performance of different activations. For each activation we have plotted loss curves. Loss curves were very helpful here because they helped me understand when the neural network was overfitting or underfitting. So, for all my different experiments we trained my convolutional neural network for 50 epochs to understand when the neural network started overfitting. The classification time has been taken to be epochs on x-axis and loss on the y-axis. To plot the curves we have used matplotlib and losses for both train and validation sets have been accumulated in a list to plot in the end.

## Sigmoid with SGD and Momentum

This activation has the formula – g(x) = 1/(1 + e^(-x))
The loss curves for both training and validation sets are.

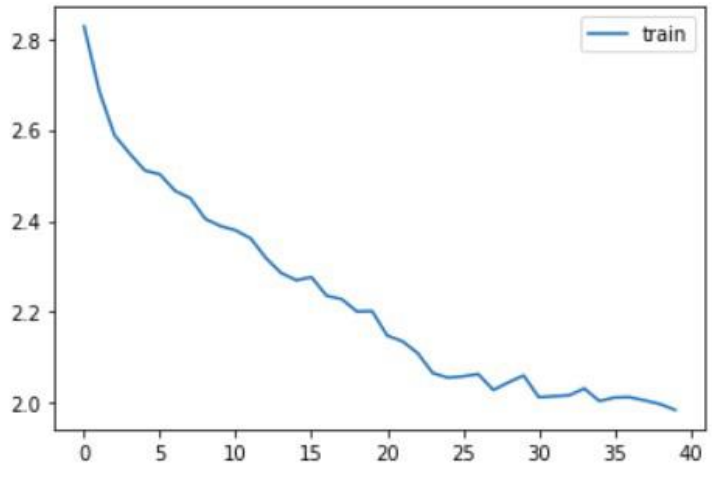


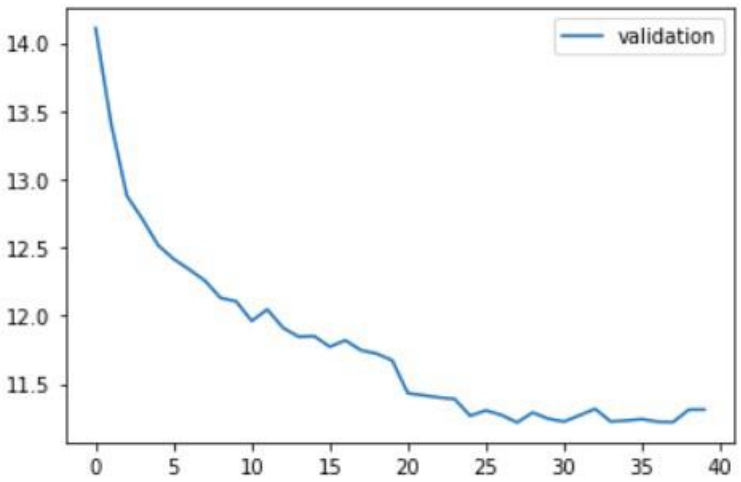




The neural network fails to converge to the global minimum. The accuracy on the 10000 test set images of the CIFAR10 dataset was 10%. Since the accuracy was low, we felt that it would take a very long time to converge so we did not train further.

## ReLU with SGD and Momentum

This activation has the formula –> g(x) = max(0, x)
The loss curves for both training and validation sets are.

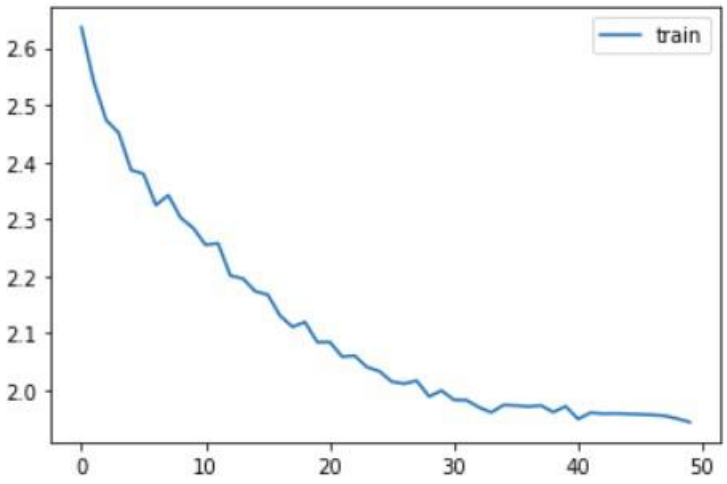


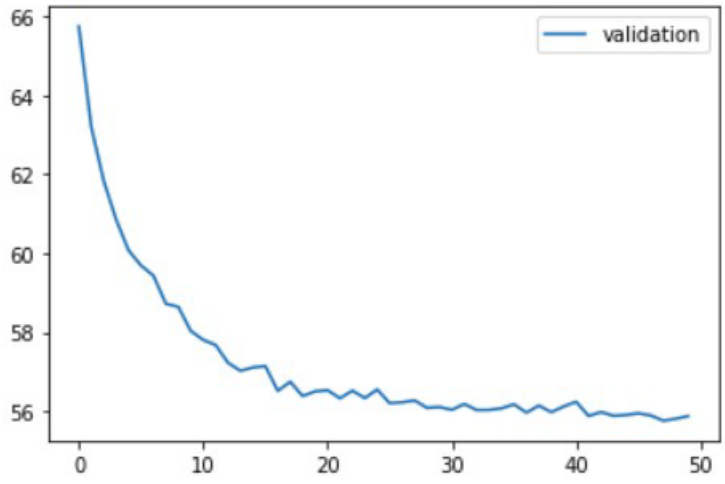



Here the ReLU activated neural network works very well. The neural network converges at 37 epochs and then starts overfitting. So, we stopped here. The accuracy on the validation set was 70%.

## tanh with SGD and Momentum

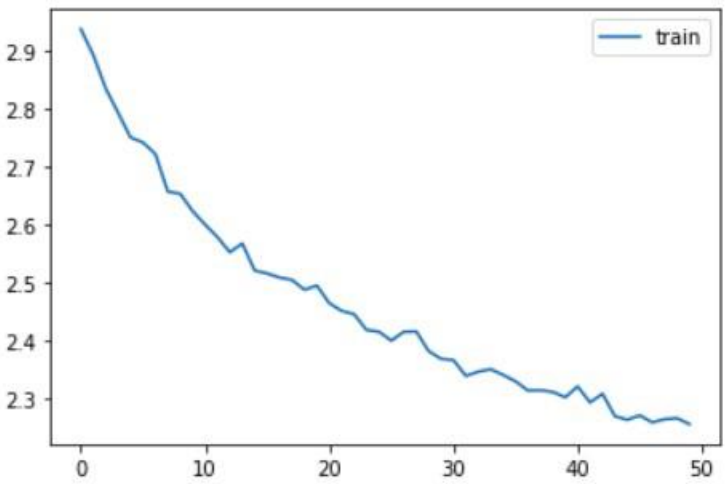
This activation has the formula – g(x) = 2/(1+ e^(-2x)) – 1
The loss curves for both training and validation sets are.

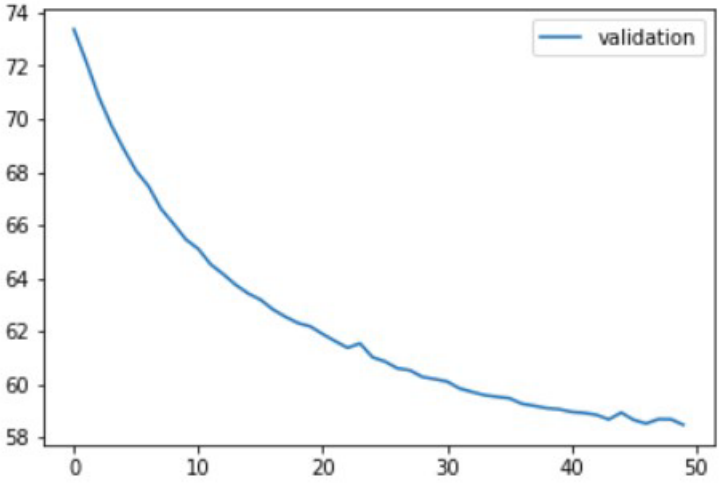



Here tanh activated neural network converges at 48 epochs after which it starts overfitting. The neural network gives an accuracy of **71**% on the test set. Though the neural network gives better performance than ReLU, it is slow in converging and requires more time to train. Now, we finalised that we would work with tanh or ReLU activation functions because Sigmoid activation was giving very low accuracy. In all the above different training methods, we have used SGD with momentum and experimented with different values. The best value was around 0.85. Also, the learning rate in the neural networks were lr = 0.001.

## tanh with SGD and Momentum = 0



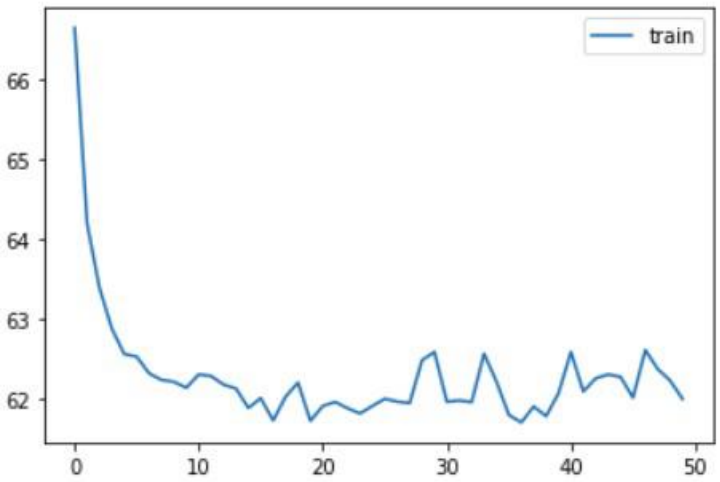
Here it is very clear that the neural network does not converge after 50 epochs also. Though, we do not have the image here, we trained the neural network for 80 epochs when it converged and the accuracy was **70%.** we tried this for ReLU also and it took a longer time to converge and gave an accuray of **68%.**

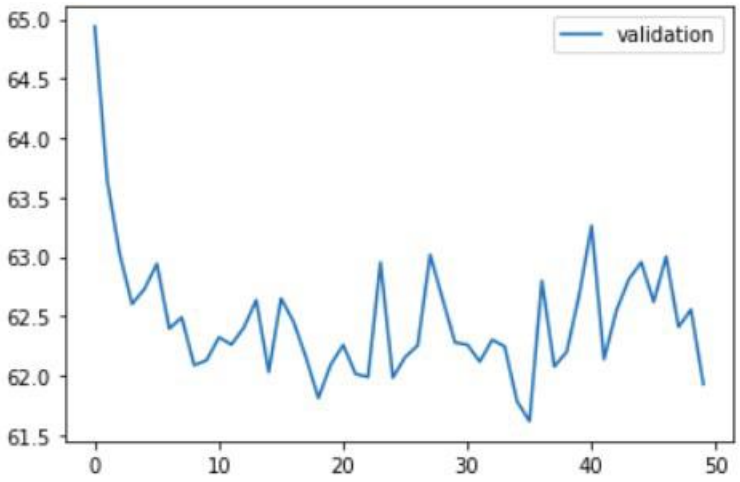
## tanh with Adam Optimizer for adaptive learning rate of different parameters

Adam optimizer is an optimizing algorithm which is a combination of SGD with momentum and RMSProp. Instead of taking the current gradient it takes a exponential moving average of last n gradients which is specified by a beta parameter which is a tuple of beta1 and beta2. It calculates the mean and the variance with an exponential moving average formula.

mt = (beta1) * (m(t-1)) + (1 - beta1) * (gt)
vt = (beta2) * (vt-1) + (1-beta2) * (gt^2)

Here beta1 and beta2 are generally 0.9 and 0.99 meaning that the average is being taken on last 10 values and last 100 values respectively. we also had the same values. The learning rate was 0.001. we played with different values of learning rates and momentum but the default values were working good.But the Adam optimizer was very noisy and did not work very well. The loss curves are below with tanh activation.



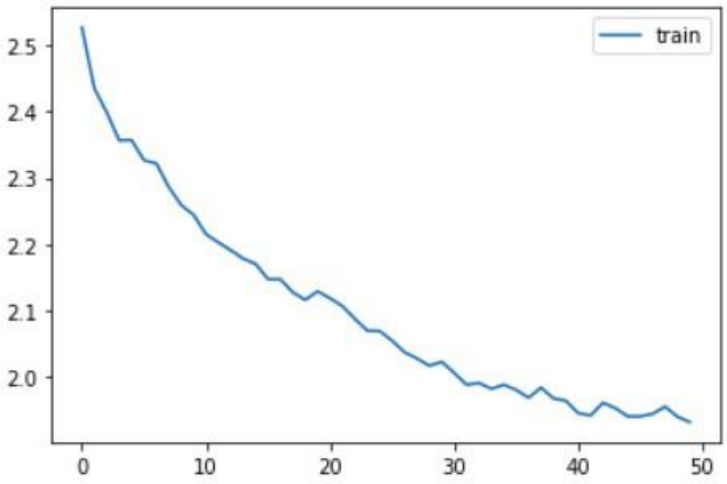

As we can see the convergence is very noisy. we trained more but the Adam optimizer was not giving good accuracies even at 70 epochs. The maximum accuracy on test set that we was able to achieve was only **52%** which was significantly less than SGD with momentum. So, now we switched back to SGD with momentum for optimization. Before batch normalization, we modified my architecture and added a fully connected layer and reduced the number of filters in the convolutional layers.

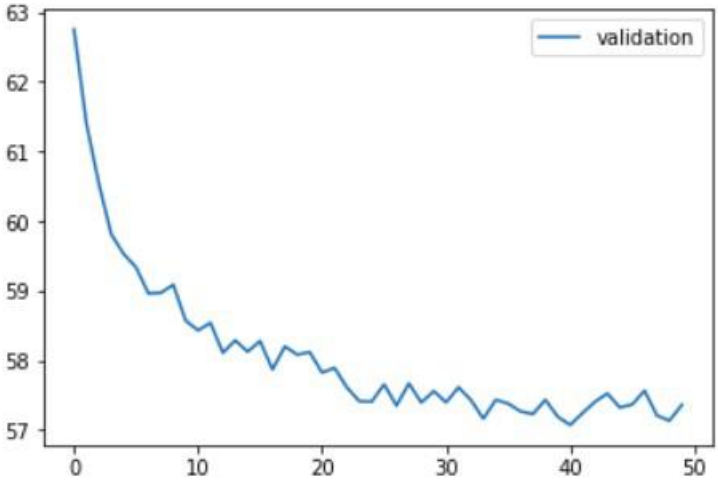
## Batch Normalization

Batch Normalization is an extra layers in neural networks that perform the function of standardization. When there are many parameters and filters, the gradients are noisy and hence, this normalizes the inputs. There are two parameters called gamma and beta that this learns if affine parameters is set to True in pytorch library. Otherwise it works with default values. But for me learnable parameters was giving good accuracies.

## tanh with Batch Normalization

we observed that batch normalization was not working well with fully connected layers. Firstly, we had done batch normalization after every layer of the neural network but this reduced the accuracy on test set to **53%.** So, we only applied batch normalization on the convolutional layers and it was converging fast. Though, the accuracies dropped a little but the training was fast.

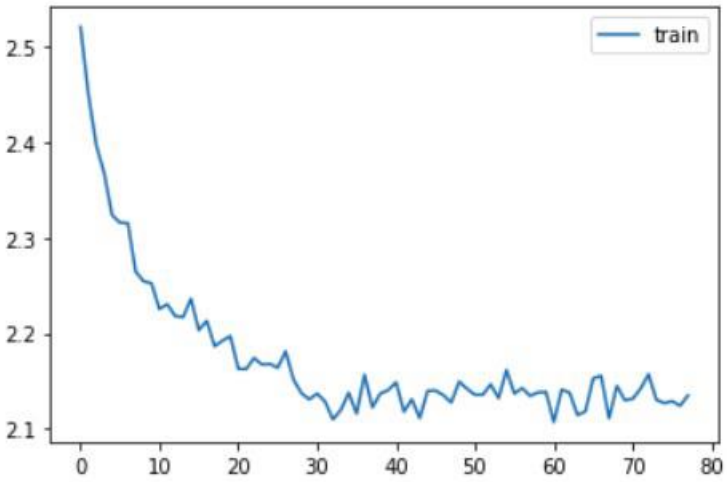


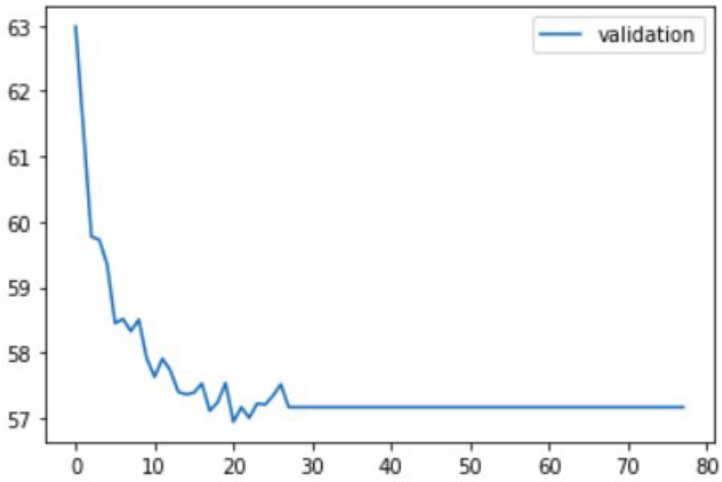




Here the CNN converged at 40 epochs as opposed to 48 epochs before after which is overfits. But the accuracy fell to about **66%.**

## ReLU with BN

Here the CNN converges at 20 epochs as opposed to 37 epochs before. The accuracy on the test set was **66%.**




Finally, we played more with different architectures and my recommended architecture would be the following –

- 2x2 Convolutional layer, stride = 2, and no of filters = 50.
- Maxpool layer of 2x2 size with stride = 2
- Bath Normalization Layer
- 2x2 Convolutional Layer, stride = 1, no of filters = 150
- Maxpool layer of 2x2 size with stride = 1

- Batch Normalization Layer
- Linear Layer with output of size 1000
- Linear layer with output size 120
- Linear layer with output size 84
- Linear layer with output size 10 because there are 10 classes.
- Softmax layer

All layers have tanh activations above. we also tried combining different activations together. Combining tanh and relu also gave fairly good accuracy. we had applied tanh on the linear layer and relu on convolutional layers which gave an accuracy of about 70% with batch normalization.

All the above training was done using GPU on Kaggle.

## Tabulated Results (with Different Networks used):

**2 CN + 3 FC**

**ReLU**

| **Description** | **Time to train** | **Accuracy Score (%)** |
|---|---|---|
| SGD without momentum | ~ 9 minutes | 54 |
| SGD + momentum = 0.3 | ~ 7 minutes | 54 |
| SGD + momentum = 0.8 | ~ 9.5 minutes | 65 |
| SGD + momentum = 0.8 + Batch Normalisation | ~ 8.5 minutes | 69 |

Tanh

| **Description** | **Time to train** | **Accuracy Score (%)** |
|---|---|---|
| SGD without momentum | ~ 9 minutes | 50 |
| SGD + momentum = 0.8 | ~ 9 minutes | 60 |
| SGD + momentum = 0.8 + Batch Normalisation | ~ 9 minutes | 65 |

**Sigmoid**

| **Description** | **Time to train** | **Accuracy Score (%)** |
|---|---|---|
| SGD + momentum = 0.8 + Batch Normalisation | ~ 9 minutes | 20 |

**3CN + 3FC**

- ReLU
- SGD
- Momemtum = 0.8
- Batch Normlization
- Took about 70 minutes to train
- Accuracy score = 70%

**4CN + 3FC**

- ReLU
- SGD

- Momemtum = 0.8
- Batch Normlization
- Took more than 80 minutes to train
- Accuracy score = 72%

**5CN + 3FC**

- ReLU
- SGD
- Momemtum = 0.8
- Batch Normlization
- Took more than 70 minutes to train
- Accuracy score = 72%

## Observations:

- The ReLU activation function outperforms the tanh and sigmoid activation functions.
- The results of Stochastic Gradient Descent with momentum are better than those of SGD without momentum.
- Batch normalization improves the categorization accuracy marginally.
- As the number of network layers or parameters increases, so does the training time.

# Problem 2:

# CNN as a feature extractor (Fruits & Vegetables dataset)

## Fruits & Vegetables Dataset

The dataset name was fruit and vegetable dataset which was in total 2 GB size. The dataset had different sets for train and test. It contained 36 categories of fruits and vegetables in total. The sized of the image were 224x224. we extracted the data using os.listdir function of os module. Link to the dataset: Fruits Vegetables Dataset Kaggle.

## Approach:

Pretrained Alexnet from pytorch has been used. Alexnet works well with images which have a specific mean and standard deviation. So, these preprocessing steps were done to bring the images to the required format. Alexnet has an architecture in which the final fully connected layer gives an output of size 1000 vector. So, we took all the images of the dataset seperated as train and test images and ran Alexnet on them to get 1000 size feature vectors. Next, we passed these images through a logistic regression classifier. We will work with different models, seeing which models give better score. Then we converted these images to numpy arrays.

Next, we had to prepare this data to be run on a GPU and alexet from pytorch. So, firstly we created a Dataset object from pytorch which has 2 important functions __len__ and __getitem__ functions. We converted both the training and testing sets to these pytorch type datasets. Then, we used a DataLoader object from pytorch to prepare this data to be passed into batches in the model. we used a batch size of 16 and used a collate_fn to generate batches. In the collate_fn named generate_batch we, firstly transformed the images to a form which is best for alexnet model. For this we used a transform object using transforms. Compose from pytorch.

All the images were first resized to 256x256 and then normalized with a mean = [0.485,.456,.406] and std = [0.229,0.224,0.225]. Images which are centered around this work best with alexnet model as mentioned in the pytorch documentation.

Next, we loaded a pretrained alexnet model from pytorch's alexnet neural network with pretrained parameter set to True. Now, we had to fine tune the alexnet model for my dataset. The output of the alexnet neural network is a 1000 size vector output. We added two more layers to this alexnet model.

- Linear Layer of size 36 due to 36 categories of fruits and vegetables
- Softmax Layer.

For all the layers, we first set require_grad = False. Then we set require_grad = True for last 3 linear layers in this model. This is to freeze the early convolutional and linear layers to avoid full training. This method is called transfer learning. Next, we fine trained this model on the train set. After training the model we removed the last linear and softmax layer. Next with zero_grad we ran the model on the train set and the test set. This gave me 1000 size vectors for the train and test sets. We converted these feature vectors to pandas dataframe and saved these to Kaggle to create a new dataset. We also saved the model.

Next, we opened another notebook and loaded these feature vectors of the train and test sets.
We ran two models primarily on these.

- SVM with linear kernel which gave an accuracy of 30.49%
- Logistic Regression model which gave an accuracy of 29.978%

We ran KNN model but it took very long time to run and did not give good accuracies.

In the above models we have got low accuracies which we particularly feel is die to the large number of categories and less images per category. We had approximately 3000 images and 36 categories. So, there were roughly, 100 images per category. The alexnet model could not be fine-tuned properly to learn each categories' features and also we feel the traditional models could not do it. But we learnt, hot to fune-tune models on a particular dataset and how CNNs can be used as feature extractors also.

In the same python notebook, we just changed the directory path for training and predicting on the Bike and Horse dataset. The accuracy obtained hereby was 100%. This is because the number of images is very less.

# Problem 3:

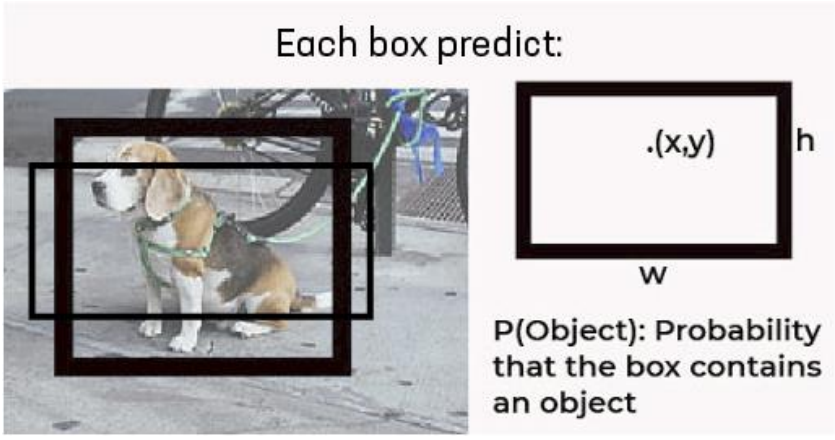
# Auto Detection using Faster RCNN and YOLOv5

## About YOLO

This architecture divides the image into a grid of *S*S* size. If the centre of the bounding box of the object is in that grid, then this grid is responsible for detecting that object. Each grid predicts bounding boxes with their confidence score. Each confidence score shows how accurate it is that the bounding that predicts contains an object and how precise it predicts the bounding box coordinates wrt. ground truth prediction.

At test time we multiply the conditional class probabilities and the individual box confidence predictions. We define our confidence score as follows:

Note, the confidence score should be *0* when there is no object exists in the grid. If there is an object present in the image the confidence score should be equal to IoU between ground truth and predicted boxes. Each bounding box consists of 5 predictions: *(x, y, w, h)* and confidence score. The *(x, y)* coordinates represent the centre of the box relative to the bounds of the grid cell. The *h, w* coordinates represents height, width of bounding box relative to *(x, y)*. The confidence score represents the presence of an object in the bounding box.

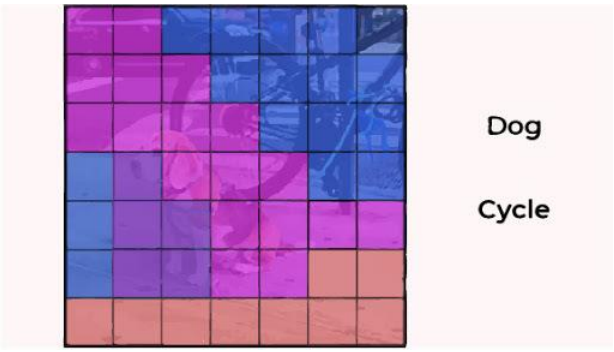



*YOLO single Grid Bounding box-Box*

This results into combination of bounding boxes from each grid like this.

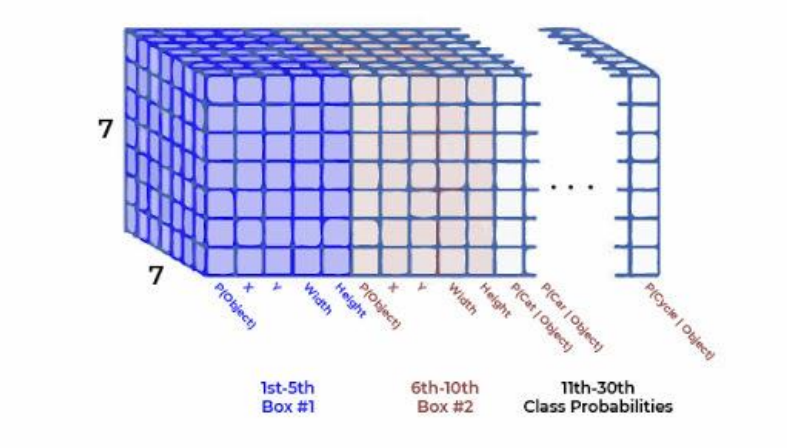
*YOLO bounding box Combination*

Each grid also predict C conditional class probability, Pr (Classi | Object).

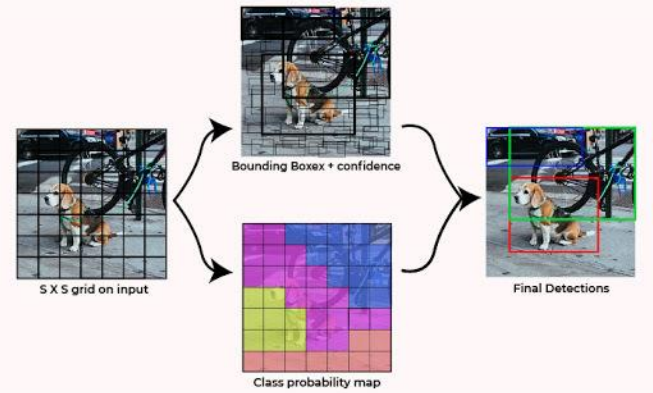



*YOLO conditional probability map*

This probability were conditional based on the presence of an object in grid cell. Regardless the number of boxes each grid cell predicts only one set of class probabilities. These prediction are encoded in the 3D tensor of size S * S * (5*B +C).




*YOLO output feature map*




*YOLO test Result*

Now, we multiply the conditional class probabilities and the individual box confidence predictions, which gives us class-specific confidence scores for each box. These scores encode both the probability of that class appearing in the box and how well the predicted box fits the object. Then after we apply non-maximal suppression to predict generate the final result of the input

YOLO is very fast at the test time because it uses only a single CNN architecture to predict results and class is defined in such a way that it treats classification as a regression problem.

**Benefits of YOLO:**

- Process frames at the rate of *45 fps* (larger network) to *150 fps*(smaller network) which is better than real-time.
- The network is able to generalize the image better.

**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.

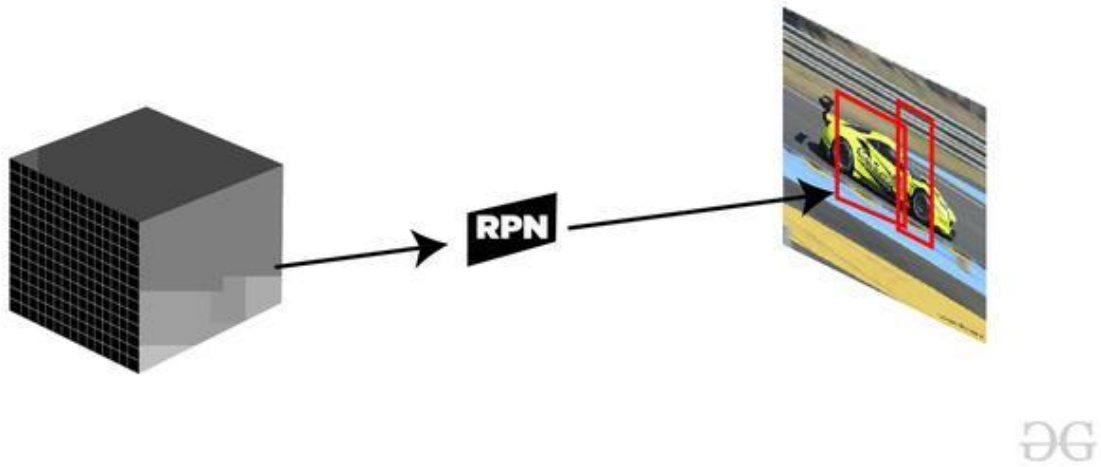
## About Faster RCNN

Faster R-CNN architecture contains 2 networks:

- Region Proposal Network (RPN)
- Object Detection Network

**Region Proposal Network (RPN):**

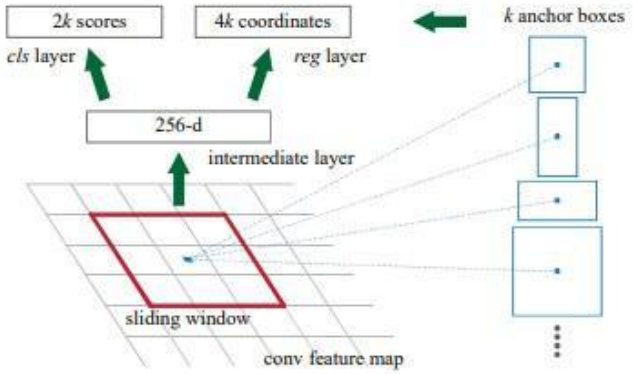
This region proposal network takes convolution feature map that is generated by the backbone layer as input and outputs the anchors generated by sliding window convolution applied on the input feature map.




*Function of RPN (Feature Map to Region Proposal)*

**Anchors:**

For each sliding window, the network generates the maximum number of k- anchor boxes. By the default the value of *k=9* (*3* scales of *(128*128, 256*256 and 512*512)* and *3* aspect ratio of *(1:1, 1:2 and 2:1))* for each of different sliding position in image. Therefore, for a convolution feature map of *W * H*, we get *N = W* H* k* anchor boxes. These region proposals then passed into an intermediate layer of *3*3* convolution and *1* padding and *256 (for ZF)* or *512 (for VGG-16 )* output channels. The output generated from this layer is passed into two layers of *1*1* convolution, the classification layer, and the regression layer. the regression layer has *4*N (W * H * (4*k))* output parameters (denoting the coordinates of bounding boxes) and the classification layer has *2*N (W * H * (2*k))* output parameters (denoting the probability of object or not object).



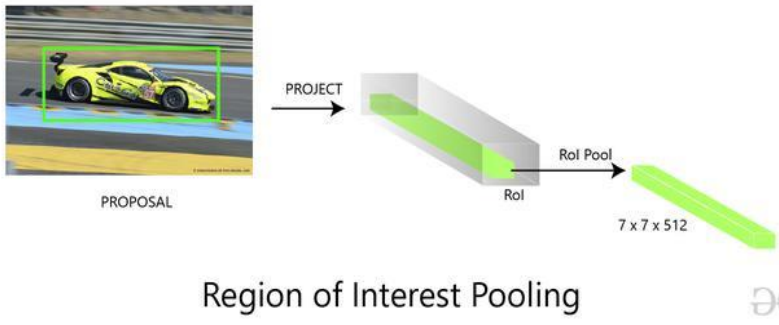

**Training and Loss Function (RPN):**

First of all, we remove all the cross-boundary anchors, so, that they do not increase the loss function. For a typical *1000*600* image, there are roughly *20000(~ 60*40*9)* anchors. If we remove the cross-boundary anchors then there are roughly *6000* anchors left per image. The paper also uses Non-Maximum Suppression based on their classification and IoU. Here they use a fixed IoU of *0.7*. This also reduces the number of anchors to *2000*. The advantage of using Non-Maximum suppression that it also doesn't hurt accuracy as well. RPN can be trained end to end by using backpropagation and stochastic gradient descent. It generates each mini-batch from the anchors of a single image. It does not train loss function on each anchor instead it selects *256* random anchors with positive and negative sample s in the ratio of *1:1*. If an image contains <128 positives then it uses more negative samples. For training RPNs, First, we need to assign binary class label (weather the concerned anchor contains an object or background). In the faster R-CNN paper, the author uses two conditions to assign a positive label to an anchor. These are:

- those anchors which have the highest Intersection-over-Union (IoU) with a ground-truth box, or
- an anchor that has an IoU overlap higher than 0.7 with any ground-truth box. Negative label to those which has IoU overlap is <0.3 for all ground truth boxes. Those anchors which does not have either positive or negative label does not contribute to training. Now Loss function is defined as follows:

**RoI pooling :**

We take the output generated from region proposal as input and passed into the RoI pooling layer, this RoI pooling layer has the same function as it performed in Fast R-CNN, to make different sizes region proposals generated from RPN into a fixed-size feature map. We have discussed RoI pooling in this article in great detail. This RoI pooling layer generates the output of size *(7*7*D)* (where *D =256 for ZF and 512 of VGG-16*).



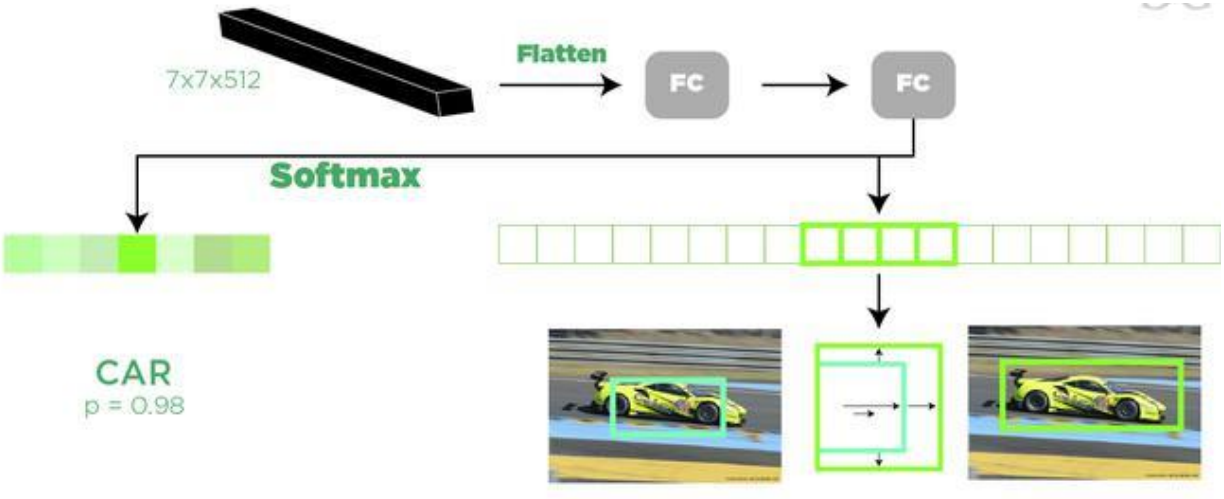

Region of Interest Pooling

**Softmax and Bounding Box Regression Layer:**

The feature map of size *(7 * 7 * D)* generated in RoI pooling are then sent to two fully connected layers, these fully connected layers flatten the feature maps and then send the output into two parallel fully connected layer each with the different task assigned to them:

The first layer is a softmax layer of *N+1* output parameters (*N* is the number of class labels and background ) that predicts the objects in the region proposal. The second layer is a bounding box regression layer that has *4* N* output parameters. This layer regresses the bounding box location of the object in the image.

*Softmax Classifier and Bounding Box Regressor*



## YOLO v5 vs Faster RCNN

| | YOLO v5 | Faster RCNN |
|---|---|---|
| Inference Speed | ✓ | |
| Detection of small or far away objects | ✓ | |

| | | |
|---|---|---|
| Inference Speed | ✓ | |
| Detection of small or far away objects | ✓ | |
| Little to no overlapping boxes | ✓ | |
| Missed Objects | ✗ | ✗ |

| | YOLO v5 | Faster RCNN |
|---|---|---|
| Inference Speed | ✓ | |
| Detection of small or far away objects | ✓ | |
| Little to no overlapping boxes | ✓ | |
| Missed Objects | ✗ | ✗ |
| Detection of Crowded objects | ✓ | ✓ |

The final comparison b/w the two models shows that YOLO v5 has a clear advantage in terms of run speed. The small YOLO v5 model runs about 2.5 times faster while managing better performance in detecting smaller objects. The results are also cleaner with little to no overlapping boxes. Ultralytics have done a fabulous job on their YOLO v5 open sourcing a model that is easy to train and run inference on.

**Approach:**

First we annotated all the images and made boxes around the autos in all the images. The box details got saved as a separate .txt file for each image. These label.txt files were used by the model. Annotated images helped in training, testing and detection using YOLO and faster RCNN.

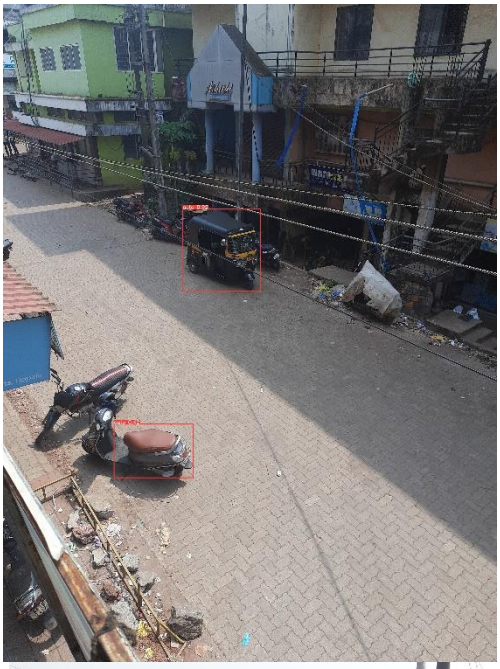
**YOLO Parameters:**

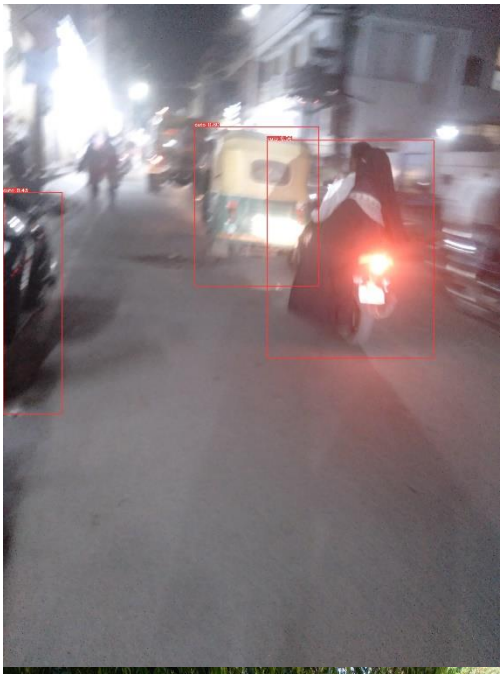
Epoch: An epoch is a term used in machine learning and **indicates the number of passes of the entire training dataset the machine learning algorithm has completed**. Datasets are usually grouped into batches (especially when the amount of data is very large). We have computed the outpits for 3 different epoch values – 50, 100, 150.

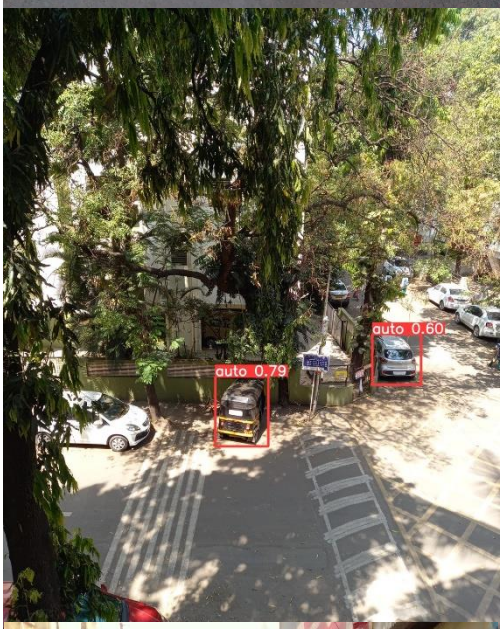
Batch Size: Batch size is **the number of images you process before updating the network weights** (i.e. a minibatch in other work) The batch size is divided by subdivisions when the network is loaded, this is how many images can fit into your GPU at once. We have used a batch size of 32.

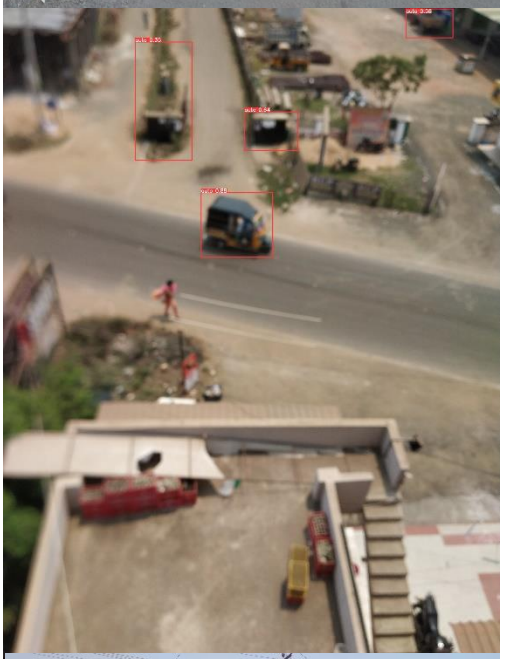
Img: Size of image to be trained. We have used 640.

## YOLO Results: Epoch = 50

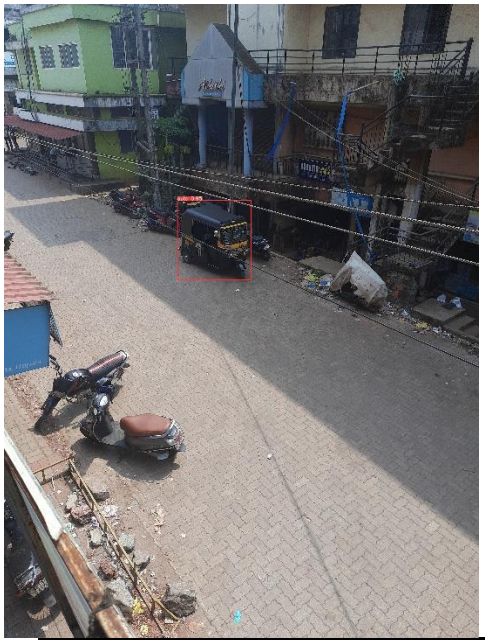


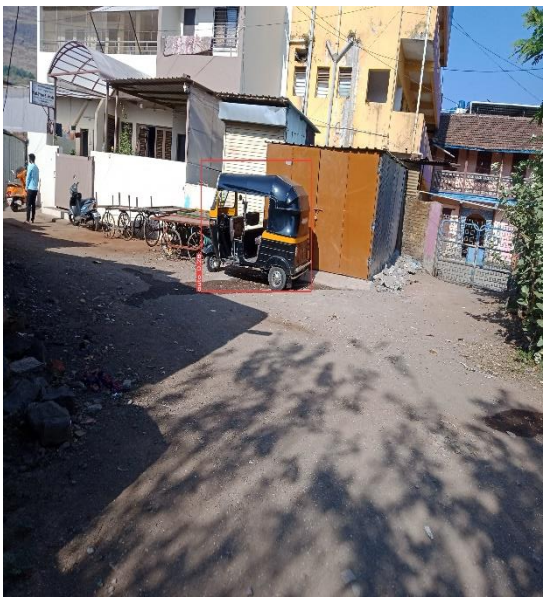




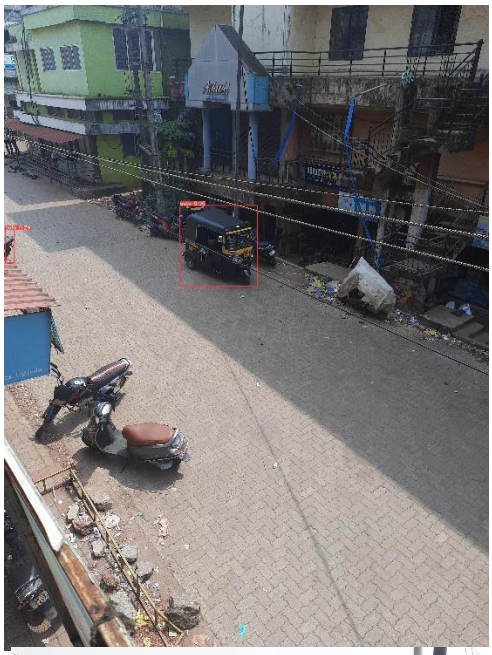



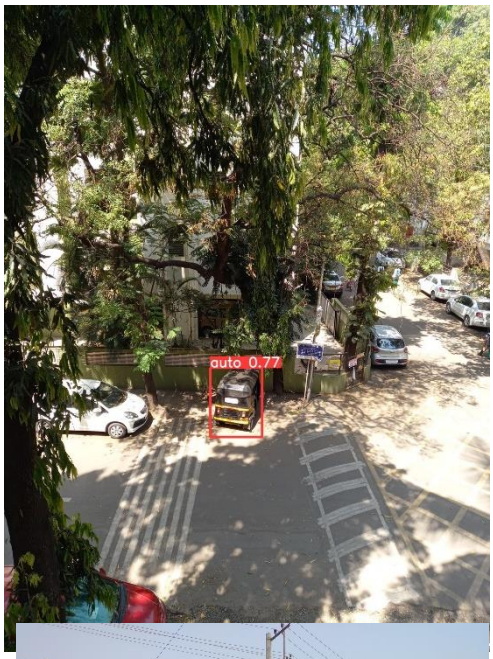
# YOLO Results: Epoch = 100

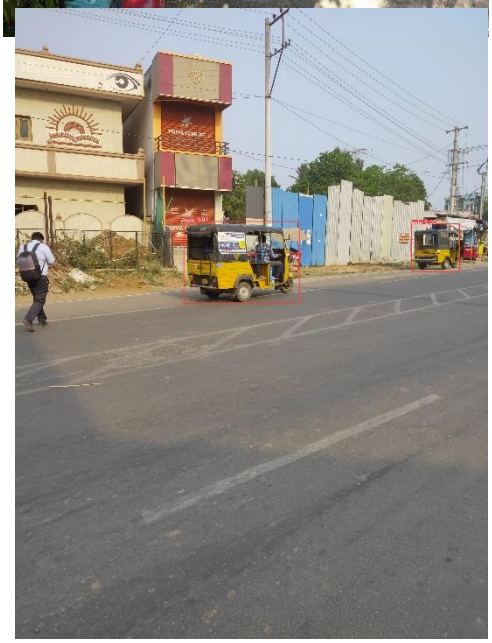




# YOLO Results: Epoch = 150



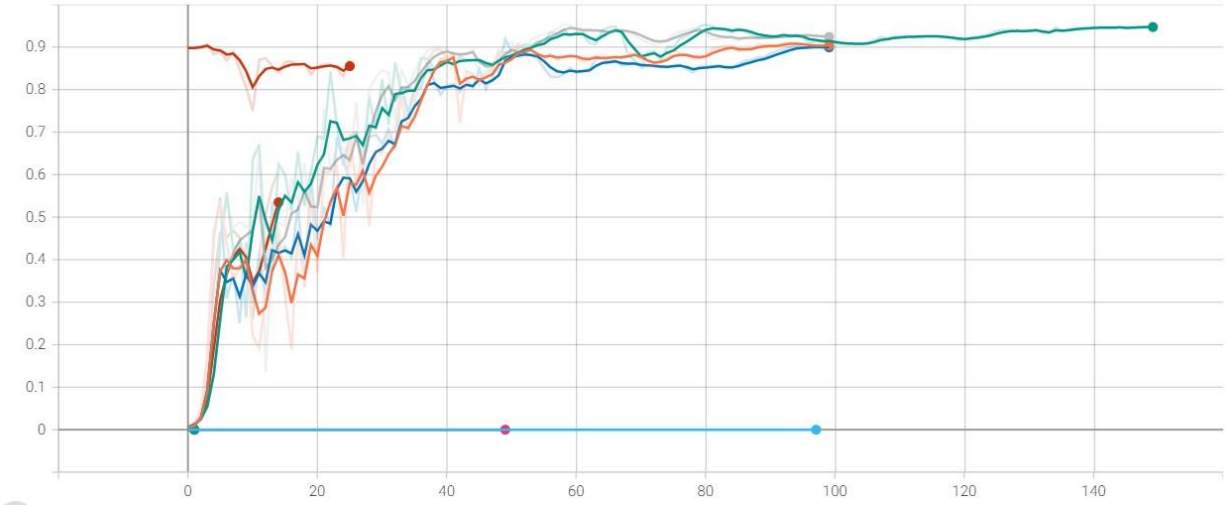


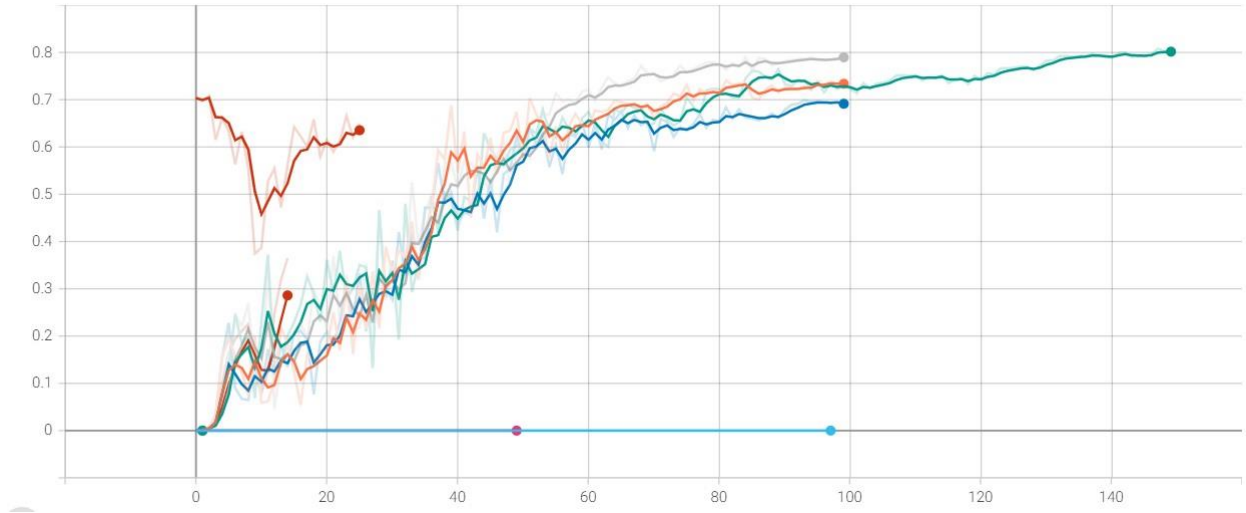
# YOLO Graphs (Build over 22 experiments):

## Metrics

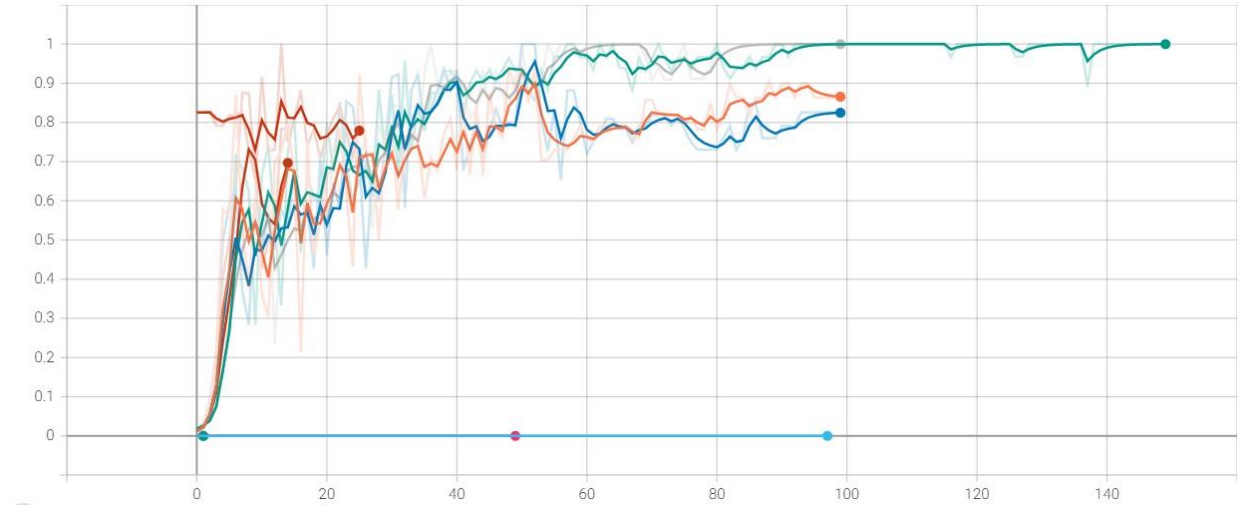
metrics/mAP_0.5
tag: metrics/mAP_0.5

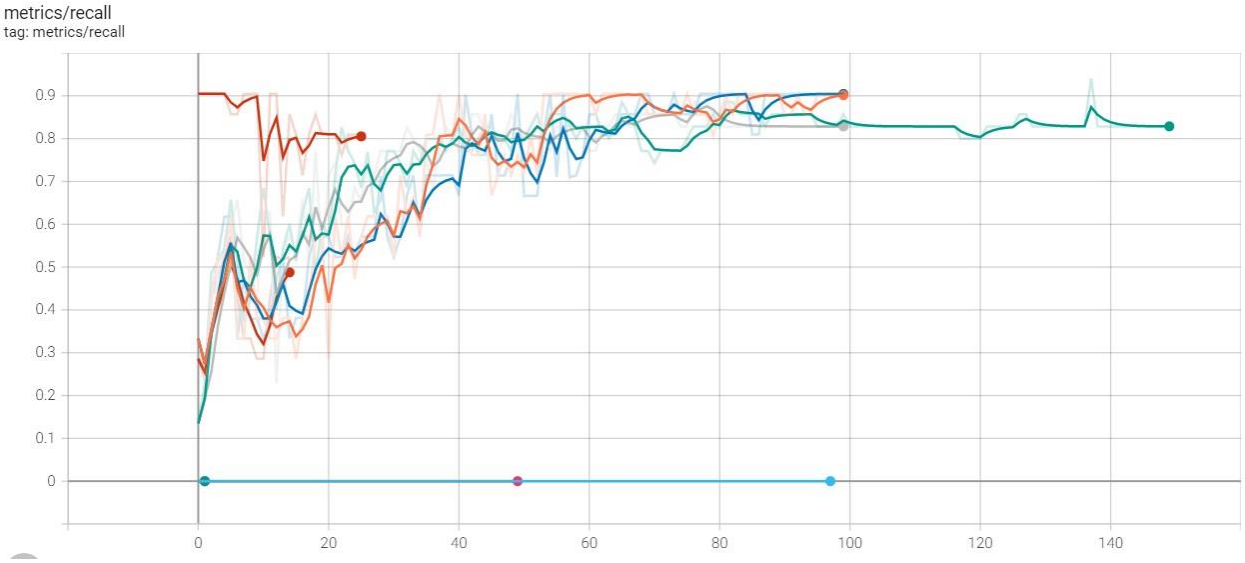



metrics/mAP_0.5:0.95
tag: metrics/mAP_0.5:0.95

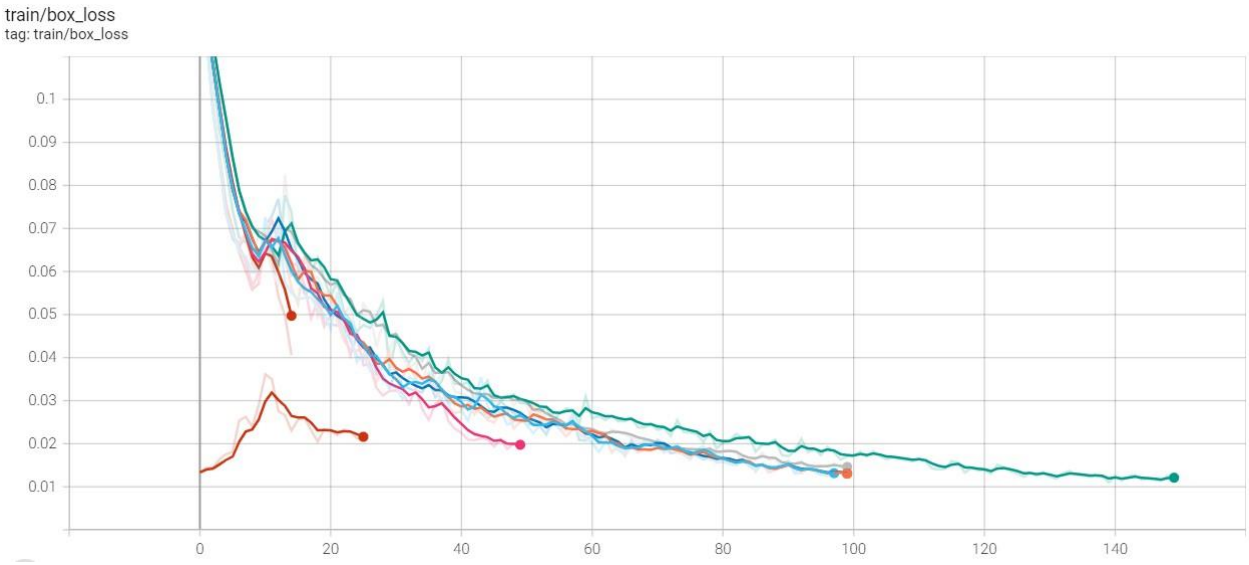


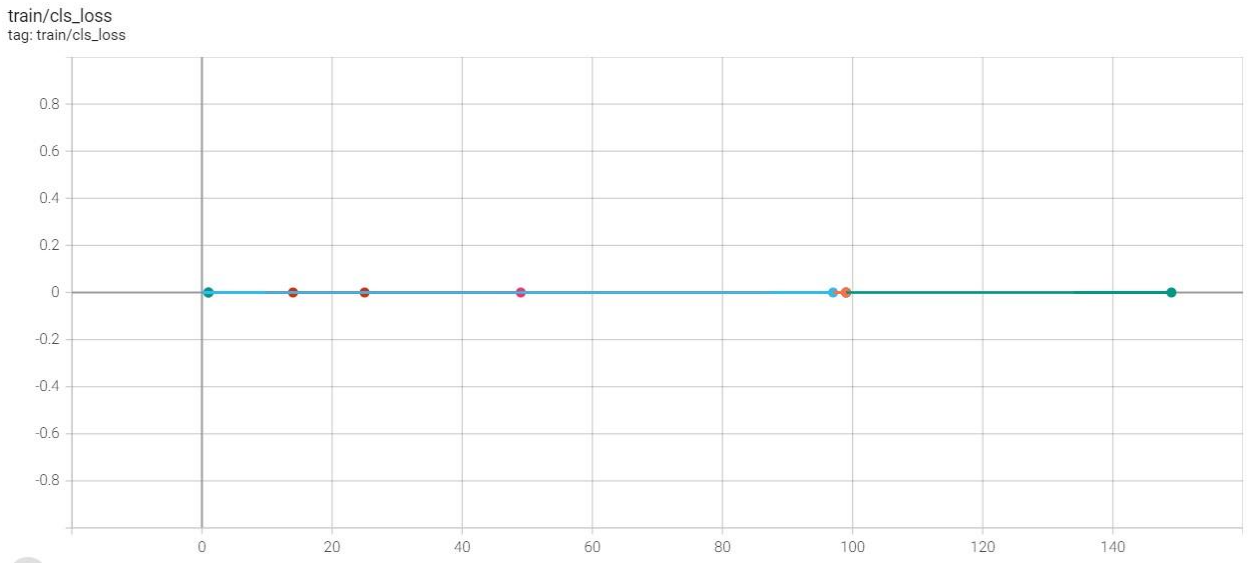

metrics/precision
tag: metrics/precision



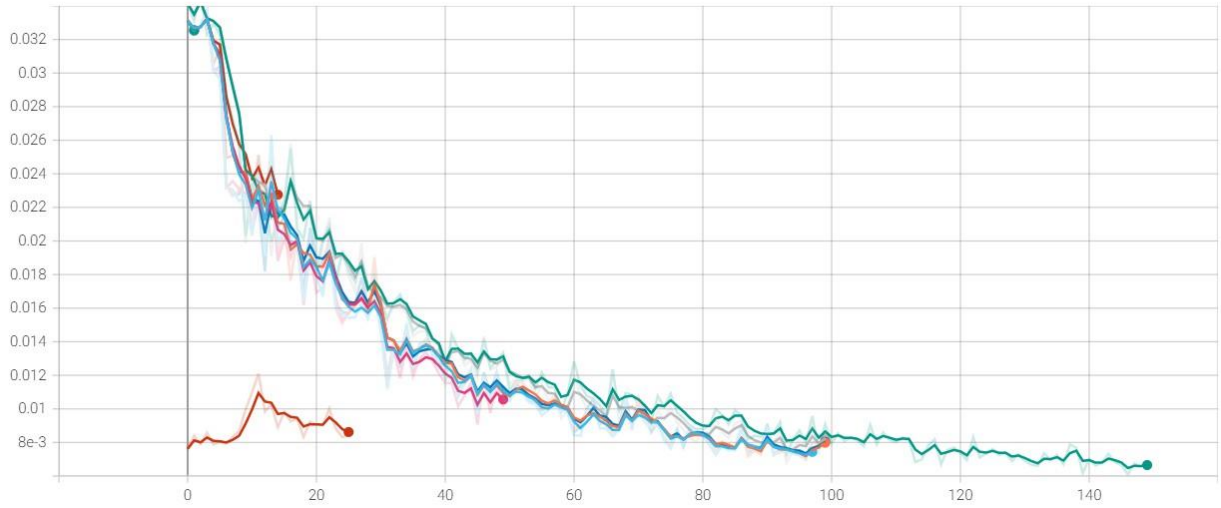

metrics/recall
tag: metrics/recall
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
20
40
60
80
100
120
140

# Training


train/box_loss
tag: train/box_loss
0.1
0.09
0.08
0.07
0.06
0.05
0.04
0.03
0.02
0.01
0
20
40
60
80
100
120
140

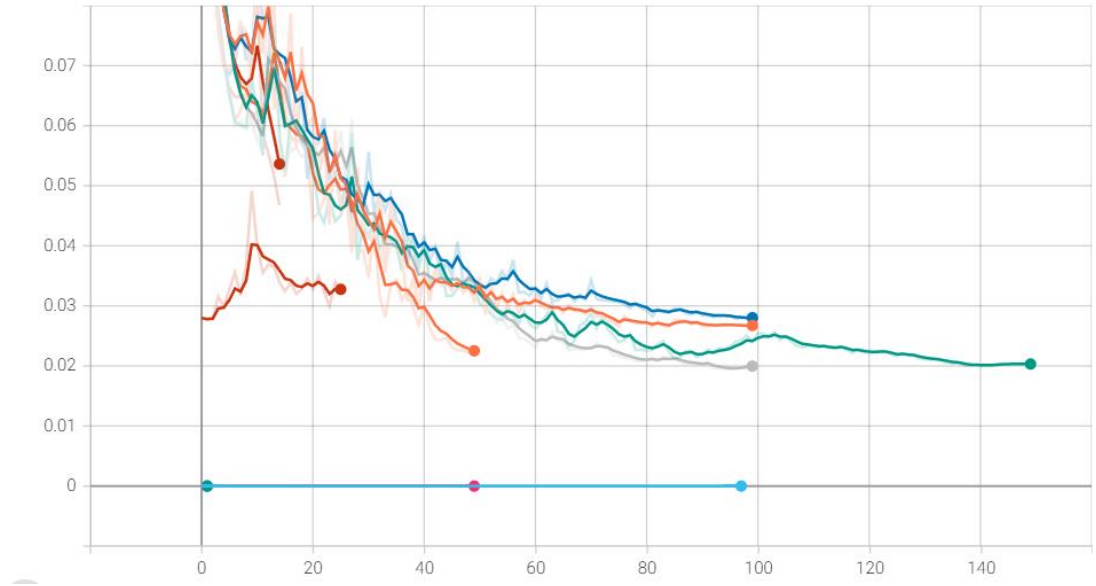

train/cls_loss
tag: train/cls_loss
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
0
20
40
60
80
100
120
140

train/obj_loss
tag: train/obj_loss

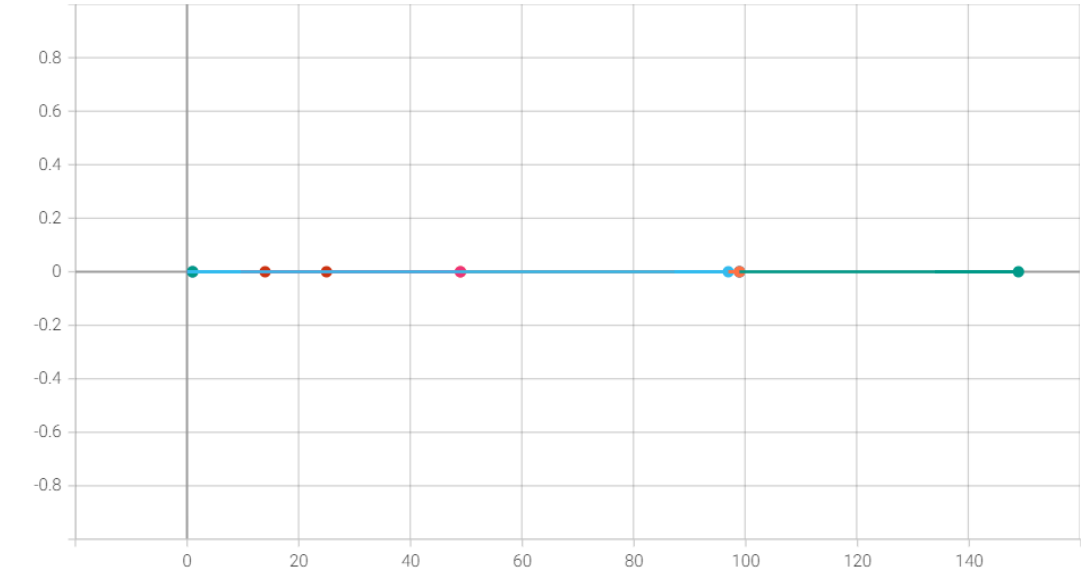


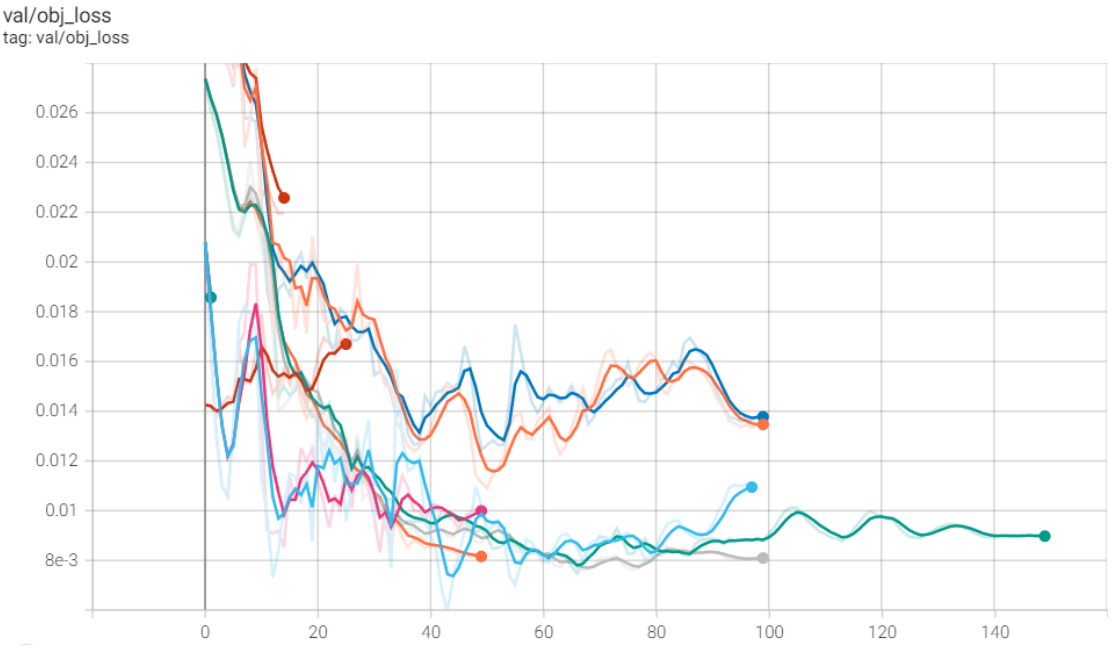

# Validation

val/box_loss
tag: val/box_loss




val/cls_loss
tag: val/cls_loss



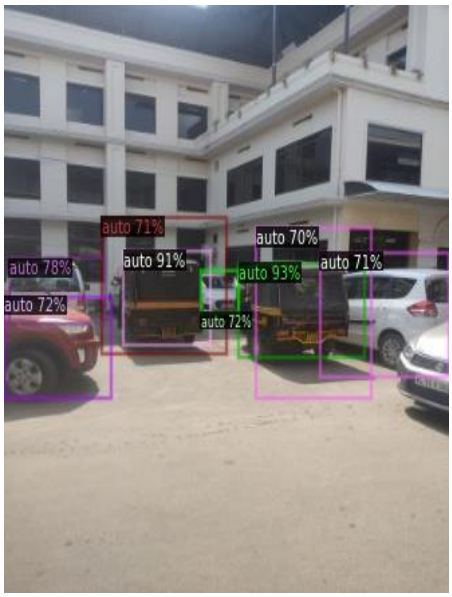

val/obj_loss
tag: val/obj_loss
0.026
0.024
0.022
0.02
0.018
0.016
0.014
0.012
0.01
8e-3
0
20
40
60
80
100
120
140

## Faster RCNN Parameters:

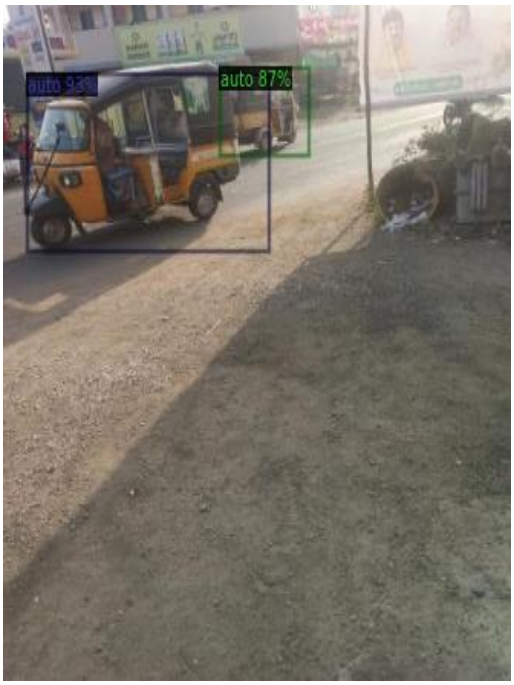
- DATALOADER.NUM_WORKERS = 4
- MODEL.WEIGHTS =
- model_zoo.get_checkpoint_url("COCO-Detection/faster_rcnn_X_101_32x8d_FPN_3x.yaml") # Let training initialize from model zoo
- SOLVER.IMS_PER_BATCH = 4
- SOLVER.BASE_LR = 0.001
- SOLVER.WARMUP_ITERS = 1000
- SOLVER.MAX_ITER = 150 #adjust up if val mAP is still rising, adjust down if overfit
- SOLVER.STEPS = (1000, 1500)
- SOLVER.GAMMA = 0.05
- MODEL.ROI_HEADS.BATCH_SIZE_PER_IMAGE = 64
- MODEL.ROI_HEADS.NUM_CLASSES = 2 #your number of classes + 1
- TEST.EVAL_PERIOD = 500

## Faster RCNN Results: Iterations = 150













# Faster RCNN Results: Iteration = 300









## Faster RCNN Results: Iteration = 450



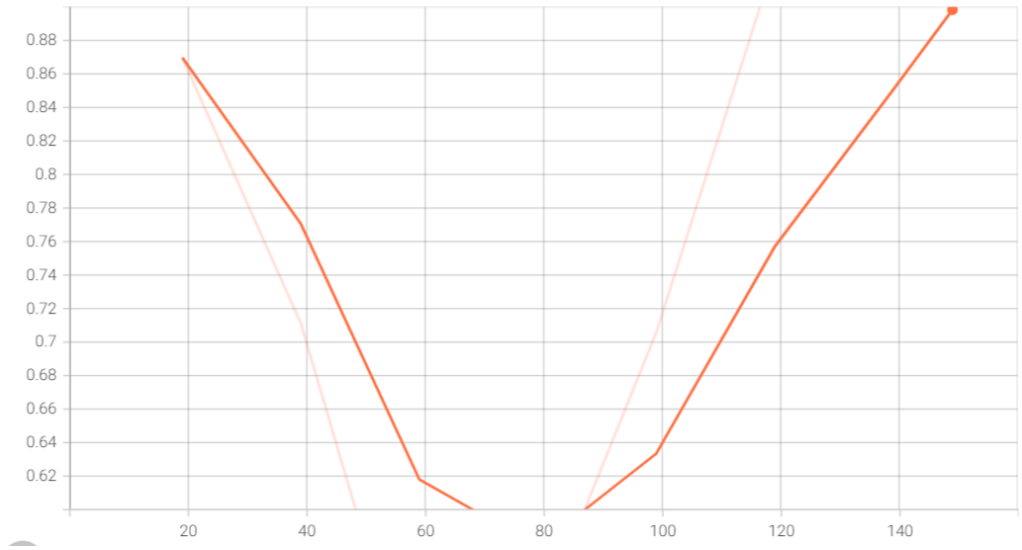




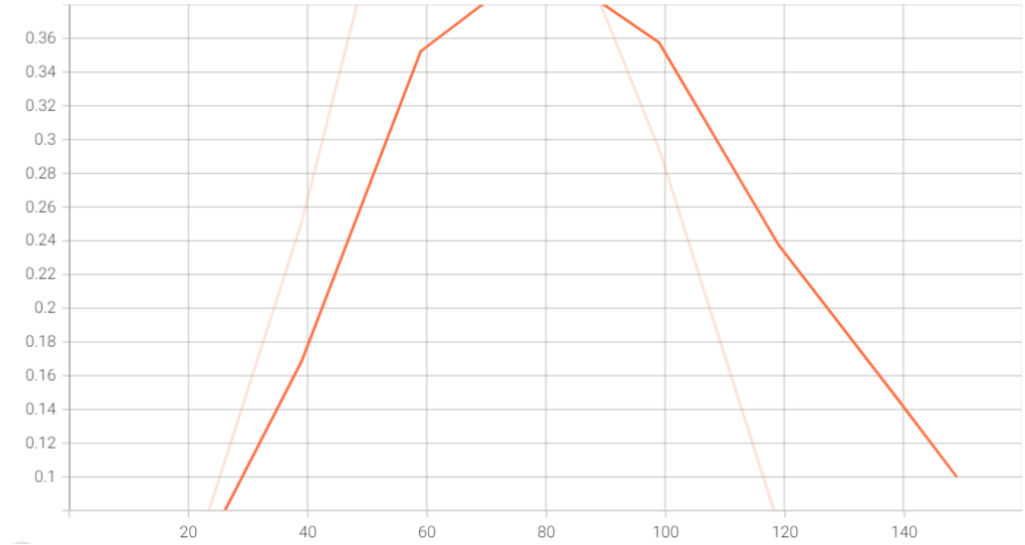


# Faster RCNN Graphs (Iterations = 150)

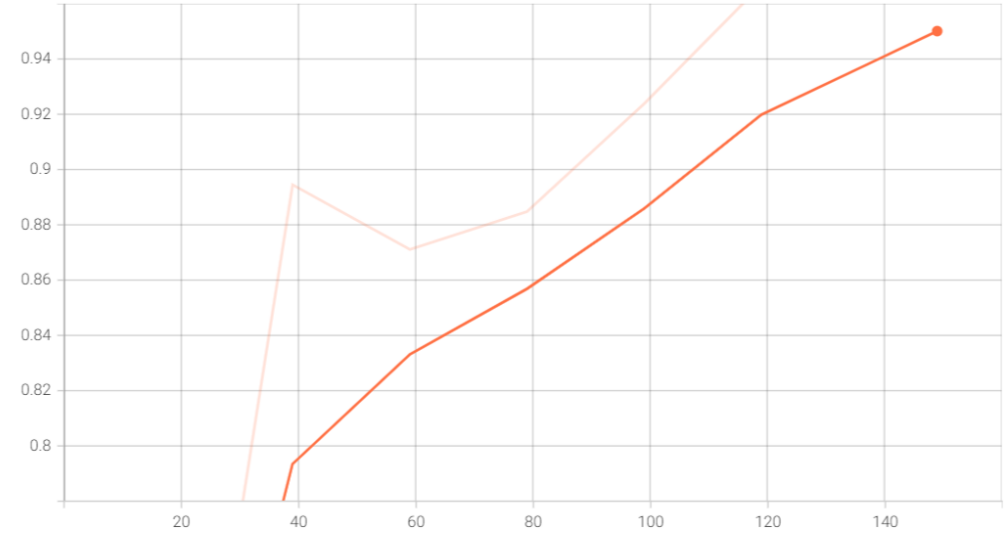
fast_rcnn/fg_cls_accuracy
tag: fast_rcnn/fg_cls_accuracy




fast_rcnn/false_negative
tag: fast_rcnn/false_negative




fast_rcnn/cls_accuracy
tag: fast_rcnn/cls_accuracy



# Observations:

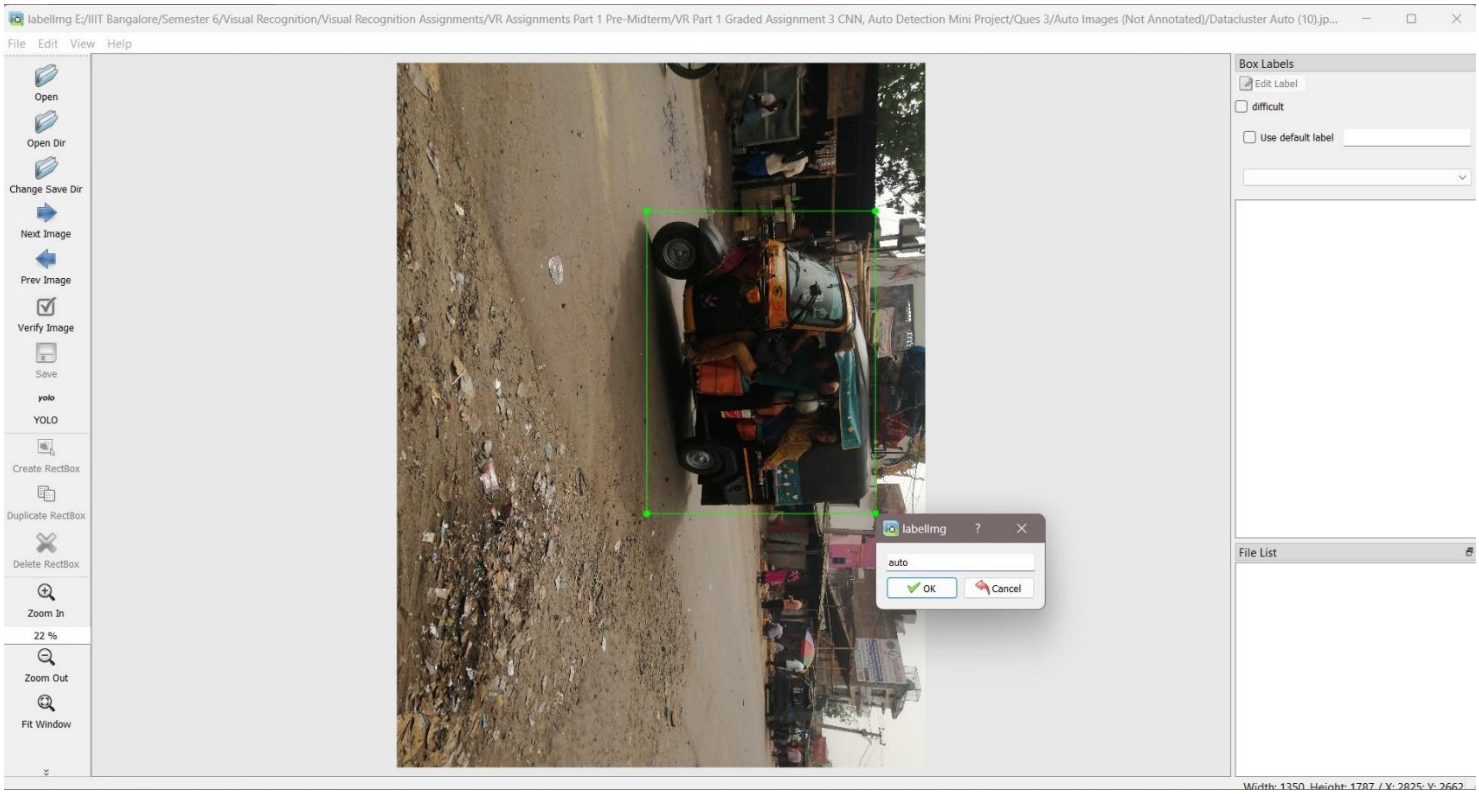
## Cases of Failures:

- If the image is very blurred then the auto detection is not accurate.
- If the image contained autos which are very close to each other and are more or less overlapping, then all the autos will not get detected properly.
- In some images, where there are cars, trucks then these are also getting detected as autos. This is not true for all the cars or trucks in the image.
- If the auto is hidden or partially covered by any other object then the detection is not accurate.

## Cases of Success:

- If there is a single auto in a clean and clear image, then it is getting detected with 100% accuracy.
- If there are multiple autos in the image and they are well distanced from each other, and the image is clear too then all the autos in the image get detected properly.
- Autos from the back are getting detected better than the autos from the front.

# Tool used for image annotation – Label Img



# Resources

Documents & Articles:

- https://pytorch.org/hub/pytorch_vision_alexnet/
- https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html
- https://pytorch.org/vision/0.8/datasets.html
- https://towardsdatascience.com/batch-normalization-explainedalgorithm-breakdown-23d2794511c
- https://towardsdatascience.com/adam-latest-trends-in-deep learningoptimization-6be9a291375c

Videos:

- https://www.youtube.com/watch?v=f2ccs2xziLk
- https://www.youtube.com/watch?v=2nR2e4J4ZaI&t=439s&ab_channel=KarndeepSingh
- https://youtu.be/yGMZOD44GrI
- https://youtu.be/ag3DLKsl2vk
- https://youtu.be/b59xfUZZqJE
- https://www.youtube.com/watch?v=2nR2e4J4ZaI&t=439s&ab_channel=KarndeepSingh

Repositories

- https://zhenye-na.github.io/2018/10/07/pytorch-resnet-cifar100.html
- https://github.com/vijendra1125/Custom-Faster-RCNN-Using-Tensorfow-Object-Detection-API
- https://www.kaggle.com/kritikseth/fruit-and-vegetable-imagerecognition