Virtual Recognition

Mini Project 1

- By

Dhamodhar Reddy (IMT2019026)

Vignesh Bondugula (IMT2019092)

Tarun Kumar (IMT2019011)

Contents

Convolution Neural Networks on CIFAR10

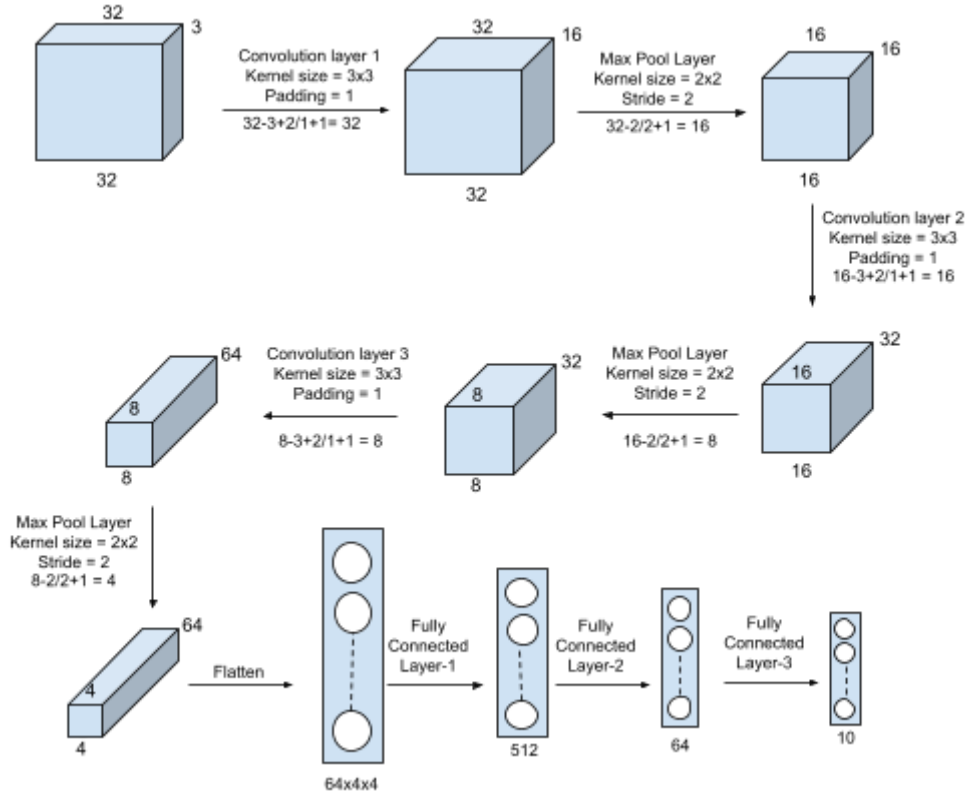
1.1 Introduction

In this part of the project we have built a deep neural network and tried to play around with it on the CIFAR10 dataset by varying hyperparameters, activation functions etc. We compared the results obtained in various cases based on two metrics. Firstly, the time taken to train the model. Secondly, the accuracy obtained through the classification. Then we made a few observations and recommended efficient architecture.

The CIFAR 10 dataset contains images of airplanes, horses, trucks, automobiles, ships, dogs, birds, frogs, cats and deer. It contains 60,000 images of 32x32 dimensions with 6000 images per class. Out of which 50,000 images are used for training and 10,000 are for testing.

1.2 CNN Model

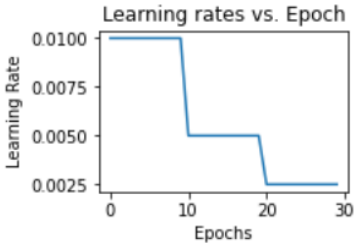
We constructed a medium deep neural network with 3 convolutional layers and 3 fully connected layers. The channels for the first layer are set as 3 and 16, second layer as 16 and 32 and the last convolutional layer as 32 and 64. We have also added max pool layers to down sample the image representation so that we do not go out of memory. We added dropout layers to reduce overfitting. The pictorial representation of the CNN model we have taken look like,



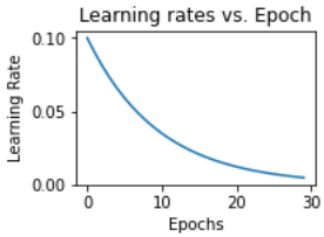
1.3 Implementation

We have used the pytorch module to build our CNN model. We followed the below steps,

- Firstly, we imported the CIFAR10 dataset from torchvision.datasets. We normalized the data and converted them into tensors. We have taken 8000 images from the train set for validation.
- We then constructed the network explained above using torch.nn and then added it into the CUDA device.
- Then we trained the model using the train data calculating losses and accuracies at each epoch. Then we validated and tested the model.
- We visualized the results using some plots.
- Then we tried to train by changing different parameters such as learning rate, momentum. We also tried making learning rates adaptive. We tried two different adaptive learning rates, Exponential and StepLR.



Exponential Learning Rate decay



StepLR Learning Rate decay

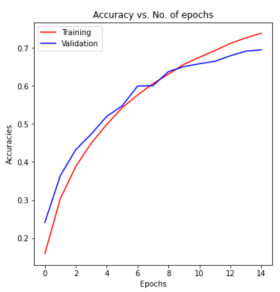
1.4 Results and analysis

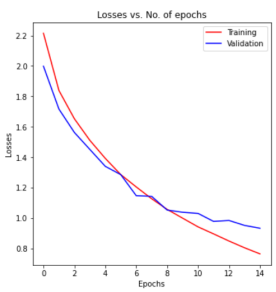
We have observed the following results in different scenarios,

Activation Function	With Momentum and Fixed Learning Rate	Without Momentum and Fixed Learning Rate	With momentum and Adaptive Learning Rate		Without momentum and adaptive Learning Rate	
Relu	Time taken = 177.73sec Accuracy = 68% Epochs = 15 Momentum = 0.9 Learning Rate = 0.005	Time taken = 359.66 Accuracy = 58% Epochs = 30 Learning Rate =0.001	**Exponential decay** Time taken = 184.20 Accuracy = 53% Epochs = 15 Initial_Learning Rate = 0.1	**StepLearning Rate decay** Time taken = 186.61 Accuracy = 69% Epochs = 15 Initial_Learning Rate = 0.01	**Exponential decay** Time taken = 356.99 Accuracy = 72% Epochs = 30 Initial_Learning Rate =0.1	**StepLearning Rate decay** Time taken = 361.22 Accuracy = 51% Epochs = 30 Initial_Learning Rate =0.01
Tanh	Time taken = 181.92sec Accuracy = 66% Epochs = 15 Momentum = 0.9 Learning Rate = 0.005	Time taken = 363.78 Accuracy = 34% Epochs = 30 Learning Rate =0.001	**Exponential decay** Time taken = 182.75 Accuracy = 48% Epochs = 15 Initial_Learning Rate = 0.1	**StepLearning Rate decay** Time taken = 183.34 Accuracy = 69% Epochs = 15 Initial_Learning Rate =0.01	**Exponential decay** Time taken = 363.77 Accuracy = 69% Epochs = 30 Initial_Learning Rate =0.1	**StepLearning Rate decay** Time taken = 391.43 Accuracy = 60% Epochs = 30 Initial_Learning Rate =0.01
Sigmoid	Time taken = 193.27sec Accuracy = 9% Epochs = 15 Momentum = 0.9 Learning Rate = 0.005	Time taken = 369.04 Accuracy = 10% Epochs = 30 Learning Rate =0.001	**Exponential decay** Time taken = 184.27 Accuracy = 10% Epochs = 15 Initial_Learning Rate =0.1	**StepLearning Rate decay** Time taken = 197.68 Accuracy = 9% Epochs = 15 Initial_Learning Rate =0.01	**Exponential decay** Time taken = 358.23 Accuracy = 9% Epochs = 30 Initial_Learning Rate =0.1	**StepLearning Rate decay** Time taken = 365.23 Accuracy = 10% Epochs = 30 Initial_Learning Rate =0.01

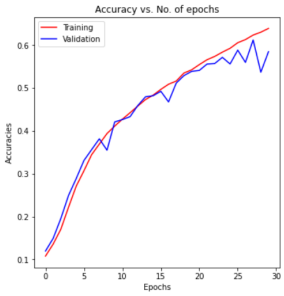
*Relu activation without momentum and exponential decay of learning rate gave best results with 72% accuracy.

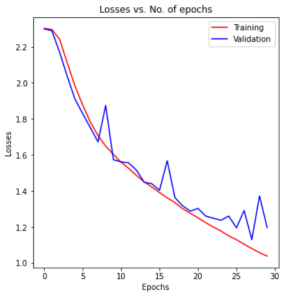
The plots in each scenario look like,



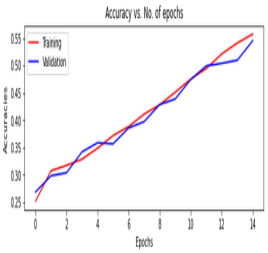


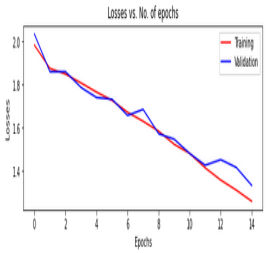
Relu activation with momentum and fixed learning rate



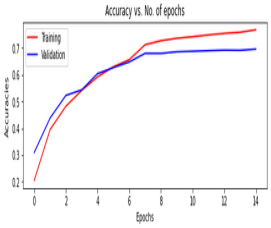


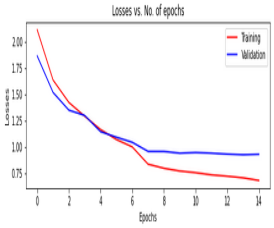
Relu activation without momentum and fixed learning rate



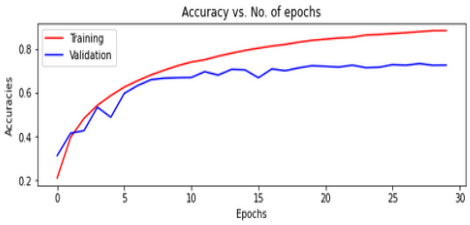


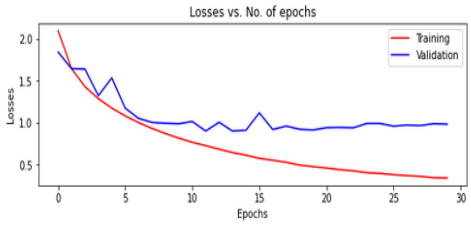
Relu activation with momentum and adaptive learning rate(E)



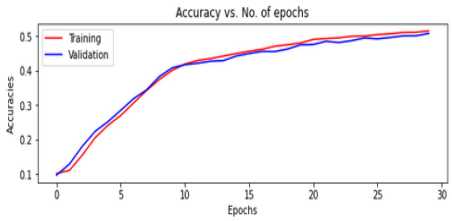


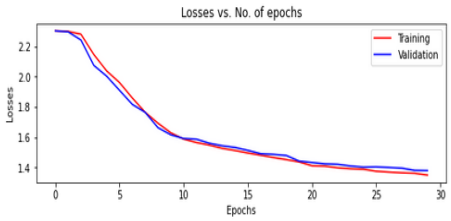
Relu activation with momentum and adaptive learning rate(S)



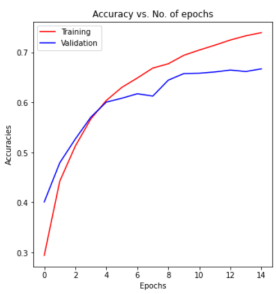


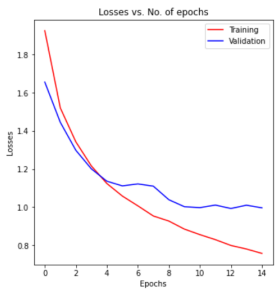
Relu activation without momentum and adaptive learning rate(E)



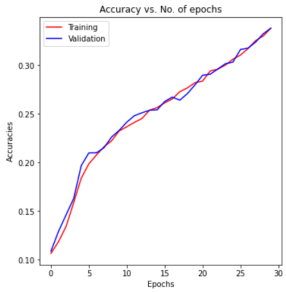


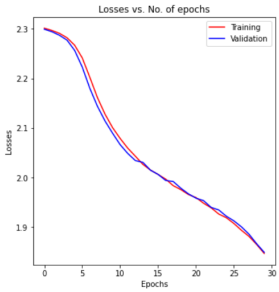
Relu activation without momentum and adaptive learning rate(S)



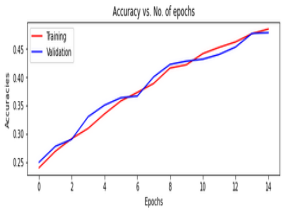


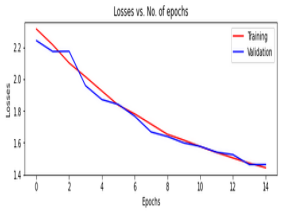
Tanh activation with momentum and fixed learning rate



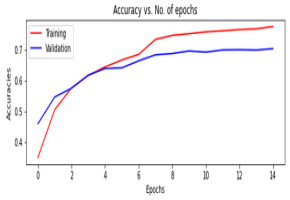


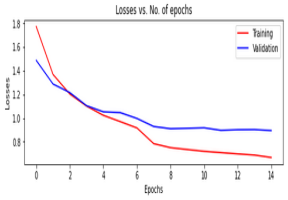
Tanh activation without momentum and fixed learning rate



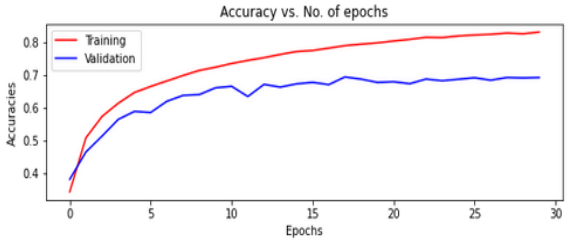


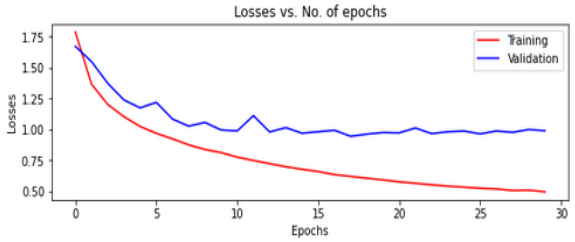
Tanh activation with momentum and adaptive learning rate(E)



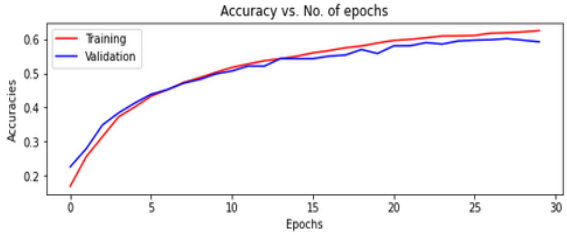


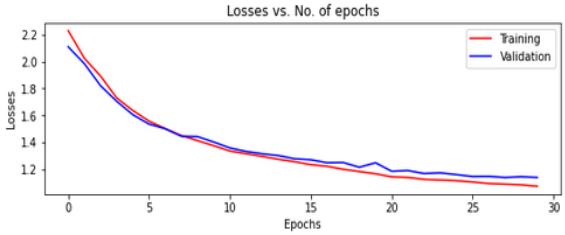
Tanh activation with momentum and adaptive learning rate(S)



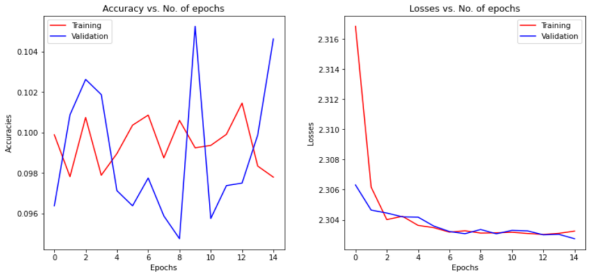


Tanh activation without momentum and adaptive learning rate(E)

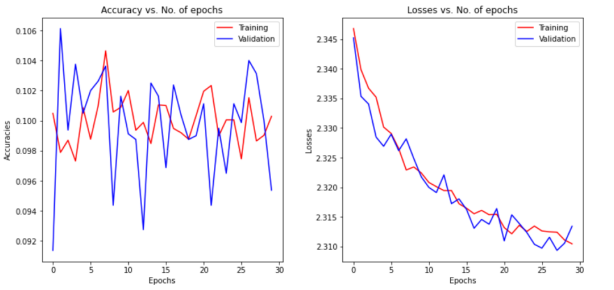




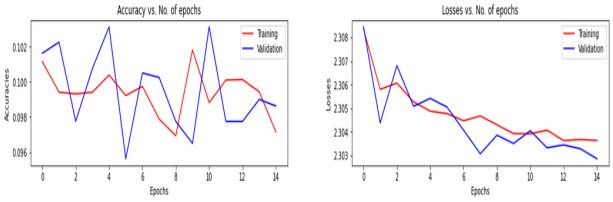
Tanh activation without momentum and adaptive learning rate(S)



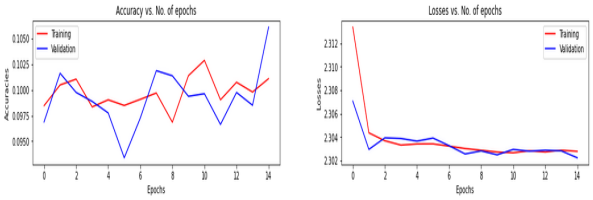
Sigmoid activation with momentum and fixed learning rate



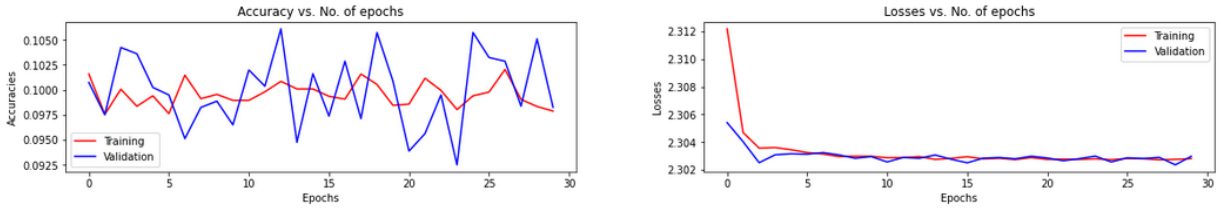
Sigmoid activation without momentum and fixed learning rate



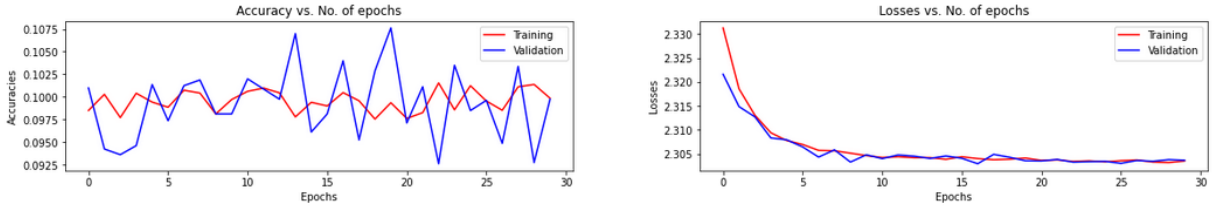
Sigmoid activation with momentum and adaptive learning rate(E)



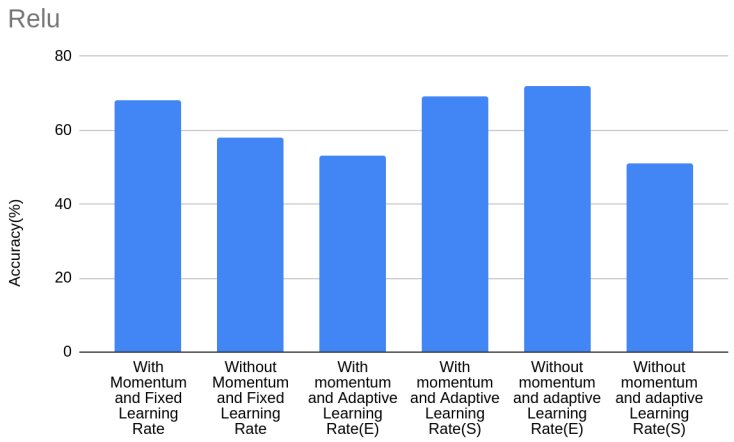
Sigmoid activation with momentum and adaptive learning rate(S)



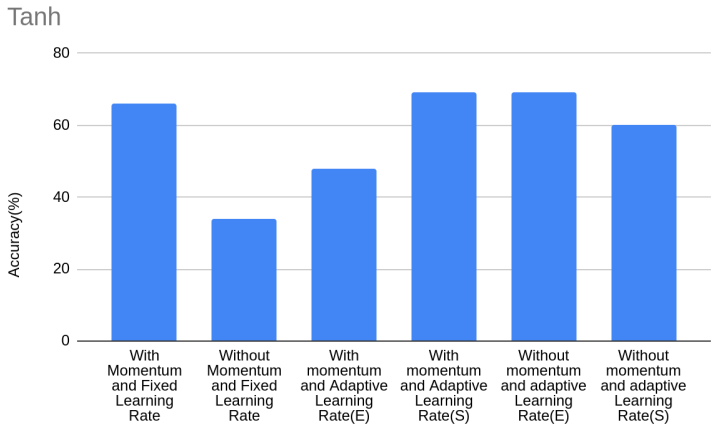
Sigmoid activation without momentum and adaptive learning rate(E)



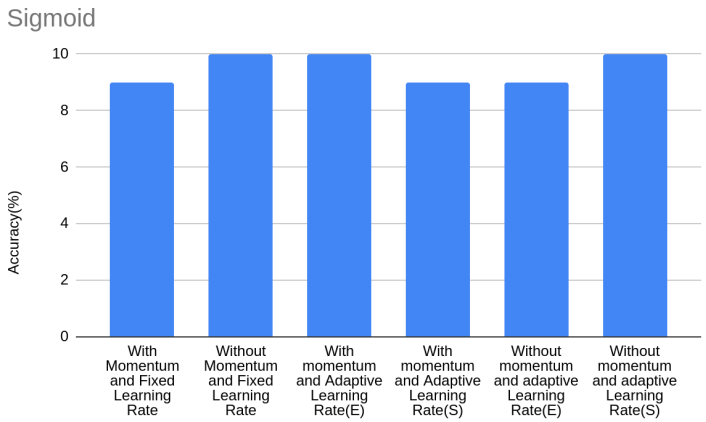
Sigmoid activation without momentum and adaptive learning rate(S)



Relu gave the best results in case of without momentum and exponential adaptive learning rate. Overall all the results were great.



Tanh gave the best results in case of momentum and exponential/step adaptive learning rate. Overall the results were not as great as Relu.



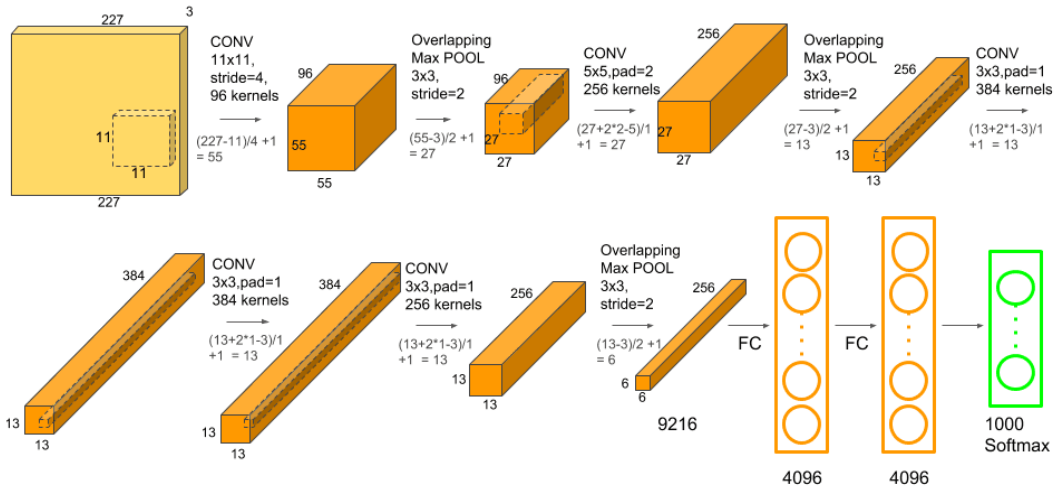
Sigmoid did not give proper results. Accuracies for all scenarios were less than or equal to 10. This gave worse results when compared to Tanh and Relu.

We observed that when there is non zero momentum, the loss is decreasing at a faster rate. So we would recommend using a lower learning rate when momentum is non zero. Without momentum the decrease in loss is slower, hence learning rate can be increased a bit to make the process faster.
Overall we would say it is better to use relu as the activation function as it gave better results. We would also recommend using adaptive learning rates methods with momentum. Exponential decay is a better choice for adaptive learning rate. It is better to avoid using sigmoid as an activation function. Even the time taken by the models with activation functions tanh or sigmoid was higher than the models with activation function relu. This makes relu more desirable.

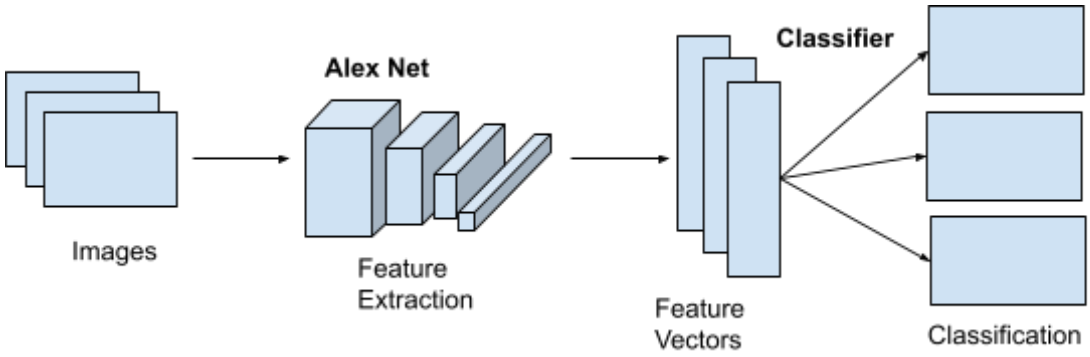
AlexNet CNN as Extractor

2.0 Introduction

Alexnet is a convolutional neural network architecture. This architecture consists of 6 convolutional layers and 3 fully connected layers. In this part of assignment, we use the alex net CNN to extract features and then use different models on them. We have tried different models such as Logistic Regression, SVM. The architecture looks like,



Alex Net Architecture



Alex Net as a Feature Extractor

2.1 Brain Tumor Detection (Kaggle Dataset Link)

2.1.1 About the Dataset

The brain tumor detection dataset consists of images of 4 different kinds of brain tumors namely glioma, meningiom, pituitary and no tumor. Each class has around 800 train images and 100 test images.

2.1.2 Implementation

We applied the alexnet pretrained model imported from pytorch to the training images and extracted the last layer from the CNN network which consists of 1000 outputs. Considering these 1000 outputs as features of each image, we trained different classification models on those features extracted from the AlexNet.

2.1.3 Different Models Used

2.1.3.1 Logistic Regression

Applying logistic regression on the features produced the following results :

Classification Report :

	precision	recall	f1-score	support
0.0	0.88	0.22	0.35	100
1.0	0.65	0.97	0.77	115
2.0	0.77	1.00	0.87	105
3.0	0.93	0.77	0.84	74
accuracy			0.75	394
macro avg	0.81	0.74	0.71	394
weighted avg	0.79	0.75	0.71	394

Confusion Matrix :

```
[[ 22  48  26   4]
 [  1 111   3   0]
 [  0   0 105   0]
 [  2  13   2  57]]
```

2.1.3.2 Support Vector Machines

Applying support vector classifiers on the features produced the following results :

Classification Report :

	precision	recall	f1-score	support
0.0	0.97	0.28	0.43	100
1.0	0.67	0.98	0.80	115
2.0	0.74	1.00	0.85	105
3.0	0.98	0.74	0.85	74
accuracy			0.76	394
macro avg	0.84	0.75	0.73	394
weighted avg	0.82	0.76	0.73	394

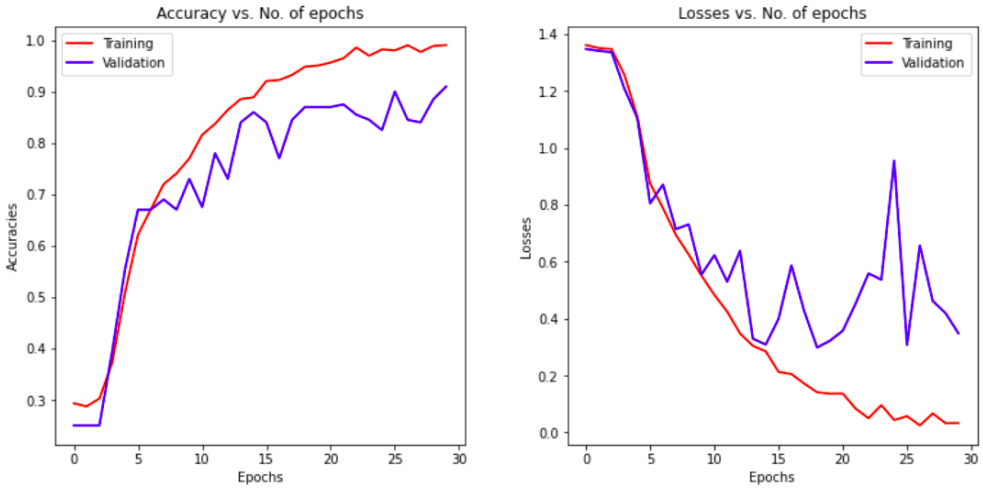
Confusion Matrix :

```
[[ 28  43  28   1]
 [  1 113   1   0]
 [  0   0 105   0]
 [  0  12   7  55]]
```

2.1.3.3 Softmax Activation

Applying softmax activation on a new layer added to the last layer, produced the following results:

Graphs:



Accuracy of the network on the test images: 70 %

2.2 Horse vs Bike Classification

2.2.1 Implementation

Similar to the previous brain tumor detection, we extracted the last layer of the pretrained AlexNet convolutional network from pytorch and considering those as features we applied different classification models.

2.2.2 Different Models Used

2.2.2.1 Logistic Regression

Applying logistic regression on the features produced the following results :
(100% Accuracy)

Classification Report :

	precision	recall	f1-score	support
0.0	1.00	1.00	1.00	13
1.0	1.00	1.00	1.00	17
accuracy			1.00	30
macro avg	1.00	1.00	1.00	30
weighted avg	1.00	1.00	1.00	30

Confusion Matrix :

[[13 0]
[0 17]]

2.2.2.2 Support Vector Machines

Applying support vector classifiers on the features produced the following results : (100% Accuracy)

Classification Report :

	precision	recall	f1-score	support
0.0	1.00	1.00	1.00	13
1.0	1.00	1.00	1.00	17
accuracy			1.00	30
macro avg	1.00	1.00	1.00	30
weighted avg	1.00	1.00	1.00	30

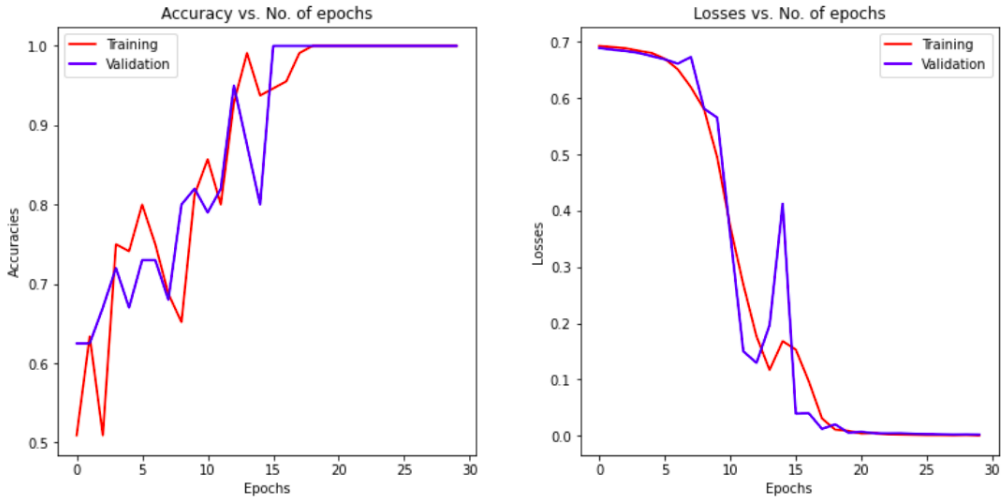
Confusion Matrix :

[[13 0]
[0 17]]

2.2.2.3 Softmax Activation

Applying softmax activation on a new layer added to the last layer, produced the following results: (100% Accuracy)

Graphs:

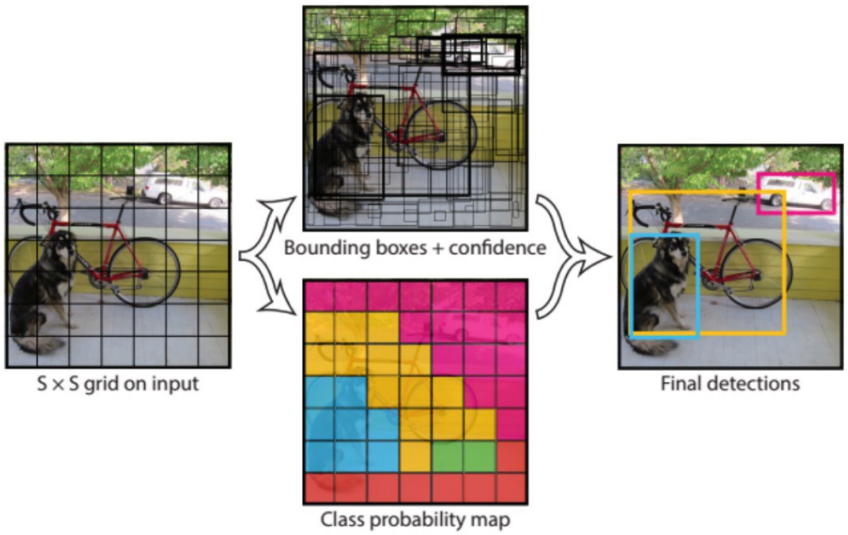


Accuracy of the network on the test images: 100 %

YOLO for Image Detection

3.1 Introduction

You only look once(YOLO) is a state-of-the-art real time object detection system. Other detection systems like fast r-cnn repurpose classifiers or localizers to perform detection. They apply the model to an image at multiple locations and scales. High scoring regions of the image are considered detections. YOLO uses a totally different approach. Yolo applies a single neural network to the full image. This network divides the image into regions and predicts bounding boxes and probabilities for each region. These bounding boxes are weighted by the predicted probabilities. YOLO also makes predictions with a single network evaluation. Hence, it is much faster than any other models.

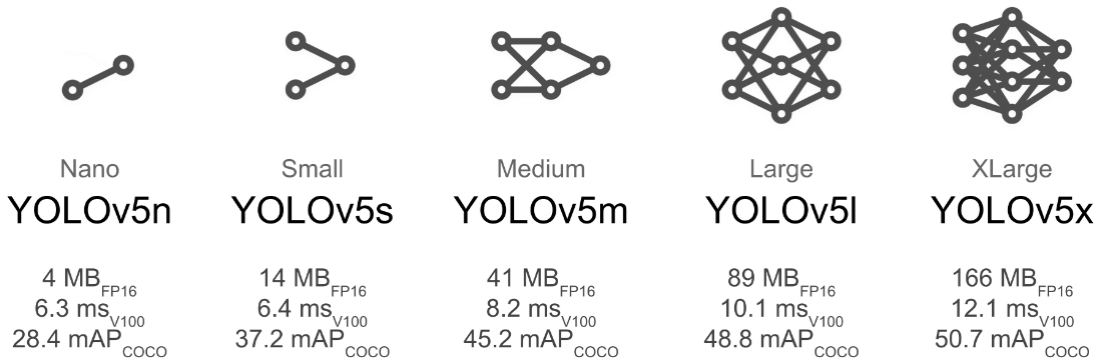


In this part of the project we have used the yolov5 module on github and trained the model on a dataset of images of Indian autos. Then we tested on a few images which contain autos and tried to detect them using the model we trained. We tried to analyze and find out the types of images for which we got accurate predictions.

3.2 Dataset and YOLO Model

Since the dataset given in the LMS did not consist of annotations, we tried to find similar datasets along with its annotations on the web to save some time. We found this dataset on the web. The full dataset consists of 1000 images from Indian roads, with arbitrary perspectives. There were around 800 images with bounding box annotations of auto rickshaws for training and validation along with 200 test images.

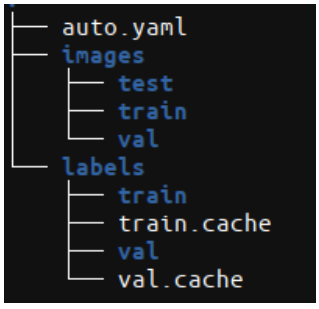
There were models of different complexities that were present to choose from in the pretrained YOLO network, ranging from nano (smallest level of complexity) to extra large (highest level of complexity).



We tried training with yolo v5m (medium network), but due to the CUDA memory constraint, we were unable to train our data with that model. Hence, we shifted to yolo v5s (small network). Even for yolo v5s network, we were unable to train the model due the same error as mentioned above. We realized the default num workers present in the yolo network was 24 which put heavy load on the GPU and hence the network was not being trained. We changed the number of workers parameters to 8. We fixed the image resolution at 640x640 and batch size to 4. For these parameters we were finally able to run the model and train the network weights.

3.3 Implementation

The annotations provided by the dataset were all in a single file. We later separated each annotation for each image respectively. Also the annotations given in the dataset consisted of four x y coordinates of the corners of the box. But to train the YOLO model we need to change it to 'width, height, center' format for every annotation and then normalize it for each image. We created individual text files for each image which stored this annotation information as the YOLO model assumes that the files are organized in a certain structure as shown below.



We found that the dataset also contained a few incorrect annotations, which we removed with help of python code.

We need to create a .yaml file which consists of the number of classes along with their names that we are trying to classify and also the path of each train and validation image folders.

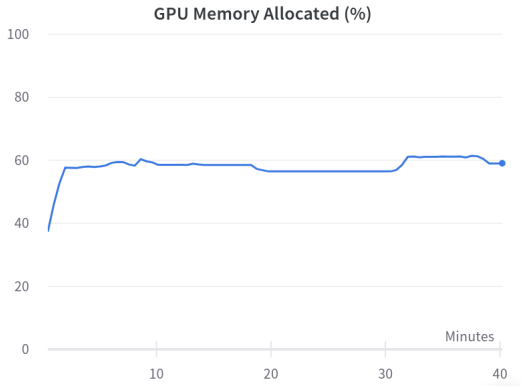
```yaml
path: ../data/
train: ../data/images/train/
val: ../data/images/val/

nc: 1

names: ['auto']
```

Now that we are done with image organization we then need to download the yolo model from the open source (github) and then install all the requirements that are needed to ensure smooth training of the yolo model. As discussed above, we finally used the yolo v5s model with mentioned parameters for training.

We used wandb, a python library, to keep track of the hyperparameters, system metrics, predictions and to compare models in real time while the model is still training. We also made sure that we reached the max GPU utilization on our local systems with the help of the charts provided on wandb. (wandb)



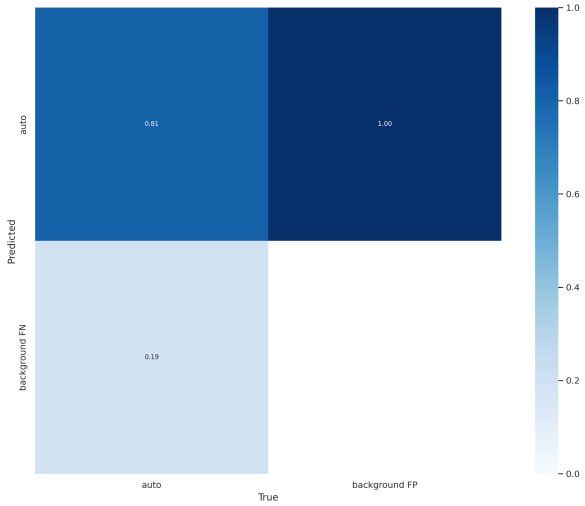
3.4 Results and Analysis

- Final Losses and Results obtained are as followed,

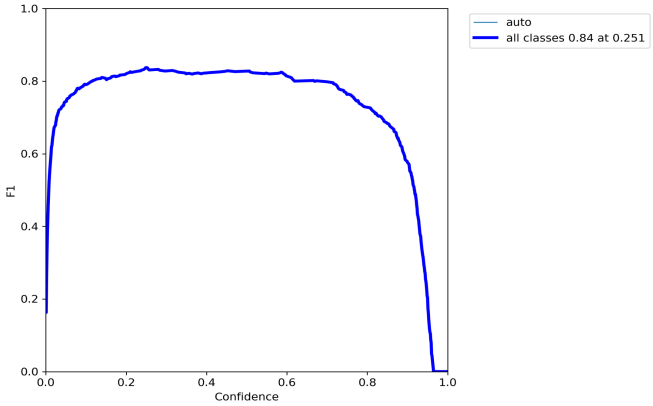
train/box_loss	train/obj_loss	train/cls_loss	metrics/precision	metrics/recall	metrics/mAP_0.5
0.023745	0.021277	0	0.88829	0.78113	0.80958

metrics/mAP_0.5:0.95	val/box_loss	val/obj_loss	val/cls_loss	x/lr0	x/lr1	x/lr2
0.59848	0.031393	0.02368	0	0.00076	0.00076	0.00076

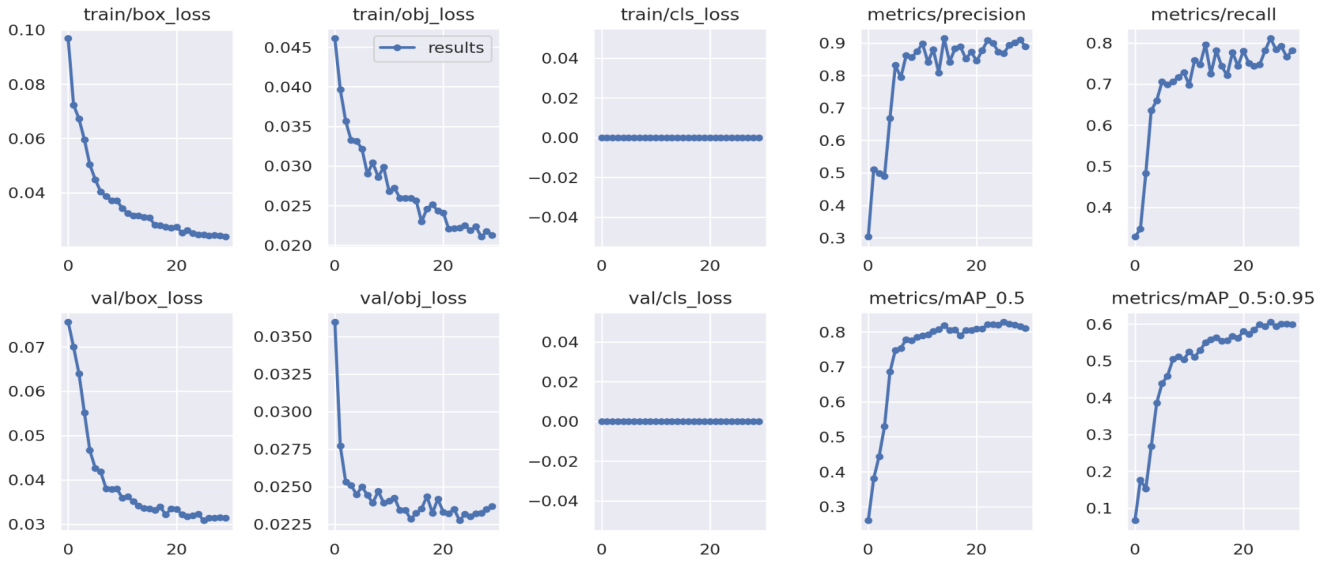
- Confusion Matrix obtained is,



- F1 Curve obtained is,



- Few Graphs that are obtained:



- Detected Images

Few of the detected validation images are as follows:

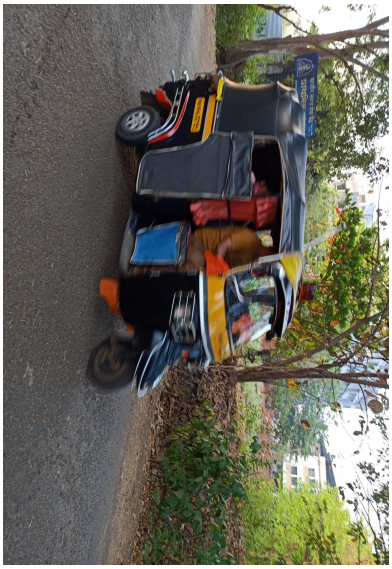
Labeled Validation Images

Images after detection with trained YOLO network

- We also tested the trained YOLO network on the dataset that was provided on the LMS

The images in where the model detected the autos accurately :

- The images where the network struggled to detected the autos :


- One important observation we made was when the pictures of the autos are rotated, the model struggled to detect those.
- For up right images the network worked very well. From the observation the network was correctly detecting almost every time. The only time when the network failed to detect was when the auto was in a rotated position.
- The solution we found to overcome this problem was to increase the number of rotated auto images in the training dataset and then feed it to the network. Due to time constraints, we were unable to rotate the images and write new annotations to the rotated images.

- At first we trained the yolo model using only 200 images. This model was not able to distinguish between auto and other vehicles. Later on when we increased the number of trained images the model slowly started to distinguish between an auto and any other vehicle.
- The model worked well even for images where there was occlusion.


- Fews autos were detected even in blurry or low resolution images