CV Assignment-4

By:- Kwanit Gupta (B19EE046)

Q1. **Perform image classification using CNN on the MNIST dataset. Follow the standard train and test split. Design an 8-layer CNN network (choose your architecture, e.g., filter size, number of channels, padding, activations, etc.). Perform the following tasks:**

1. Show the calculation of output filter size at each layer of CNN.

2. Calculate the number of parameters in your CNN. Calculation steps should be clearly shown in the report.

3. Report the following on test data: (should be implemented from scratch)

a. Confusion matrix
b. Overall and classwise accuracy.
c. ROC curve. (you can choose one class as positive and the rest classes as negative)

4. Report loss curve during training.

5. Replace your CNN with resnet18 and compare it with all metrics given in part 3. Comment on the final performance of your CNN and resnet18.

Solution:-

Input size: 128 x 128 x 1

Layer 1: Conv2d(1, 4, kernel_size=5)

Output size: 124 x 124 x 4 (applying formula [(W - K + 2P) / S] + 1)
Activation map size: 4 x 124 x 124
Number of parameters: (5 x 5 x 1 x 4) + 4 = 104 (applying formula [K*K*Input_Filters*Output_Filters) + Output_Filters

Layer 2: Pool2d(2)

Output size: 62 x 62 x 4 (applying formula [(W - F) / S] + 1)
Activation map size: 4 x 62 x 62
Number of parameters: 0

Layer 3: Conv2d(4, 5, kernel_size=5)

Output size: 58 x 58 x 5 (applying formula [(W - K + 2P) / S] + 1)
Activation map size: 5 x 58 x 58
Number of parameters: (5 x 5 x 4 x 5) + 5 = 505

Layer 4: Pool2d(2)

Output size: 29 x 29 x 5 (applying formula [(W - F) / S] + 1)
Activation map size: 5 x 29 x 29
Number of parameters: 0

Layer 5: Conv2d(5, 6, kernel_size=4)

Output size: 26 x 26 x 6 (applying formula [(W - K + 2P) / S] + 1)
Activation map size: 6 x 26 x 26
Number of parameters: (4 x 4 x 5 x 6) + 6 = 966

Layer 6: Pool2d(2)

Output size: 13 x 13 x 6 (applying formula [(W - F) / S] + 1)
Activation map size: 6 x 13 x 13
Number of parameters: 0

Layer 7: Flatten()

Output size: 1014

Layer 8: Linear(1014, 512)

Output size: 512
Number of parameters: (1014 x 512) + 512 = 519,680

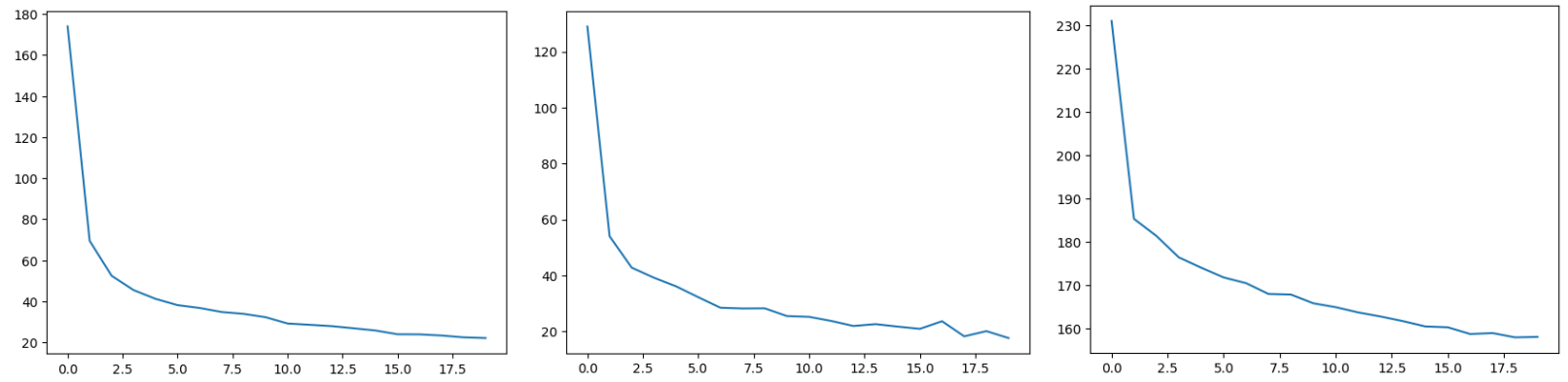
Layer 9: Linear(512, 10)

Output size: 10
Number of parameters: (512 x 10) + 10 = 5,130
Total number of parameters in the model: 526,305

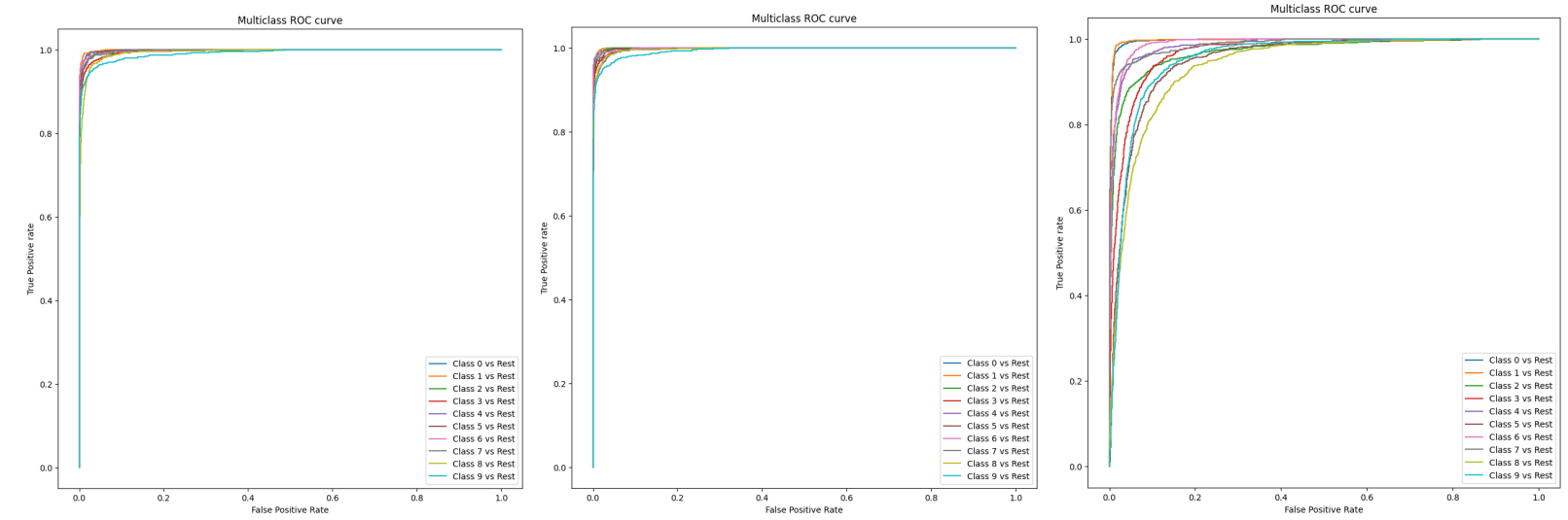
Conv2d layer 1: (5 x 5 x 1 x 4) + 4 = 104
Conv2d layer 3: (5 x 5 x 4 x 5) + 5 = 505
Conv2d layer 5: (4 x 4 x 5 x 6) + 6 = 966
Linear layer 8: (1014 x 512) + 512 = 519,680
Linear layer 9: (512 x 10) + 10 = 5,130
So, the total number of parameters in the CNN is 526,305.

For the above question, I utilized 3 different variants of pooling (MaxPool, AvgPool, and PowerPool). Following are the results (loss curve, confusion matrix, classification report, and ROC-Curves)

Loss v/s Epoch Curves (applied for 20 epochs) for (a) Max Pool (b) Power Pool (c) Avg Pool



Multi-Class ROC Curves for (a) Max Pool (b) Power Pool (c) Avg Pool



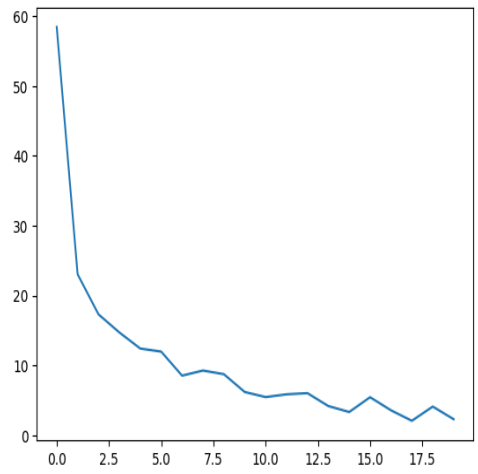
Confusion Matrix, Overall and Class-Wise Accuracy/Metrics for (a) Max Pool (b) Power Pool (c) Avg Poo

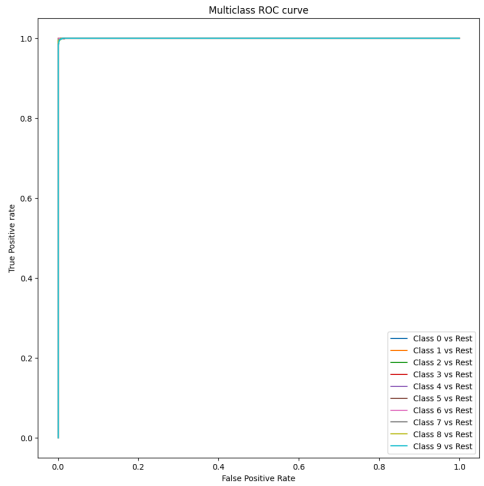
```
[[201   0   5   6   1   2  10   0  12   3]
 [  0 104   5   0   1   0   4   2   0   2]
 [  0   3 236   8   1   1   2   9   5   1]
 [  1   0   0 198   0   6   0   0   0   3]
 [  0   2   5   0 202   0   6   1   0   9]
 [  1   0   0  13   0 106   2   0   2   8]
 [  3   1   1   1   0   8 163   0   2   0]
 [  3   0   8   8   3   0   0 243   5  12]
 [  0   1   3   2   0   0   3   1 176   1]
 [  3   0   1   6   6   1   0   4   4 202]]

Overall Accuracy =  97.67 %

Class-Wise Accuracies =  [[98.87755102 99.38325991 97.28682171 95.64356436 98.77800407 97.98206278
  97.18162839 98.3463035  96.91991786 96.13478692]]
              precision    recall  f1-score   support

           0       0.99      0.96      0.97      1008
           1       0.99      0.99      0.99      1142
           2       0.97      0.97      0.97      1034
           3       0.96      0.99      0.97       976
           4       0.99      0.98      0.98       993
           5       0.98      0.97      0.98       900
           6       0.97      0.98      0.98       947
           7       0.98      0.96      0.97      1050
           8       0.97      0.99      0.98       955
           9       0.96      0.97      0.97       995

    accuracy                           0.98     10000
   macro avg       0.98      0.98      0.98     10000
weighted avg       0.98      0.98      0.98     10000
```

```
[[210   0   2   0   4   2   7   1   8   3]
 [  0 110   3   0   2   0   4   3   1   3]
 [  0   1 251   4   0   0   1   3   2   0]
 [  0   0   0 230   0   3   1   2   1   1]
 [  0   0   0   0 171   0   1   0   0   1]
 [  0   0   0   4   0 114   1   0   0   3]
 [  0   0   0   0   5   3 171   0   1   1]
 [  1   0   5   0   3   1   0 249   0   7]
 [  1   0   3   4   5   1   4   1 191   8]
 [  0   0   0   0  24   0   0   1   2 214]]

Overall Accuracy =  98.47 %

Class-Wise Accuracies =  [[99.79591837 99.91189427 98.74031008 98.81188119 95.62118126 98.87892377
  98.01670146 98.92996109 98.45995893 97.32408325]]
              precision    recall  f1-score   support

           0       1.00      0.97      0.99      1005
           1       1.00      0.99      0.99      1150
           2       0.99      0.99      0.99      1030
           3       0.99      0.99      0.99      1006
           4       0.96      1.00      0.98       941
           5       0.99      0.99      0.99       890
           6       0.98      0.99      0.98       949
           7       0.99      0.98      0.99      1034
           8       0.98      0.97      0.98       986
           9       0.97      0.97      0.97      1009

    accuracy                           0.98     10000
   macro avg       0.98      0.98      0.98     10000
weighted avg       0.98      0.98      0.98     10000
```

```
[[199   0   9   5   1  11   8   2  11  13]
 [  0  93  11   1   2   3   3   9  10   7]
 [  1   5 169  32   9   8  11  28   9   1]
 [  1   1   6 125   2  31   1   3  13   9]
 [  1   1   8   1 160   9   6  13  10  47]
 [  5   1   4  29   0  17  28   2  27   4]
 [  3   3   9   3   7   8 129   0   8   0]
 [  1   1   7  10   1   6   1 169   8  18]
 [  1   6  37  30   9  25   3   2 103  10]
 [  0   0   4   6  23   6   0  32   7 132]]

Overall Accuracy =  92.32 %

Class-Wise Accuracies =  [[98.67346939 98.41409692 90.79457364 88.41584158 94.50101833 88.0044843
  93.63256785 91.14785992 89.42505133 89.19722498]]
              precision    recall  f1-score   support

           0       0.99      0.94      0.96      1027
           1       0.98      0.96      0.97      1163
           2       0.91      0.90      0.90      1041
           3       0.88      0.93      0.91       960
           4       0.95      0.91      0.93      1024
           5       0.88      0.89      0.88       885
           6       0.94      0.96      0.95       938
           7       0.91      0.95      0.93       990
           8       0.89      0.88      0.89       994
           9       0.89      0.92      0.91       978

    accuracy                           0.92     10000
   macro avg       0.92      0.92      0.92     10000
weighted avg       0.92      0.92      0.92     10000
```

For the case of Resnet18, following are the required comparisons:-






```
100%|██████████| 10000/10000 [00:00<00:00, 248804.94it/s]
[[210   0   1   0   0   0   3   0   2   0]
 [  0 109   1   0   0   0   2   3   0   0]
 [  0   0   2   0   1   0   0   4   2   0]
 [  0   1   1 238   0   1   0   0   1   1]
 [  0   0   0   0 209   0   0   0   0   4]
 [  0   0   0   4   0 122   1   0   0   2]
 [  1   0   0   0   0   1 182   0   0   0]
 [  1   1   3   0   0   0   0 253   0   2]
 [  0   0   0   0   0   0   2   0 200   3]
 [  0   0   0   0   4   0   0   0   1 229]]

Overall Accuracy =  99.46 %

Class-Wise Accuracies =  [[99.79591837 99.82378855 99.41860465 99.6039604  99.49083503 99.77578475
  99.16492693 99.31906615 99.38398357 98.81070367]]
              precision    recall  f1-score   support

           0       1.00      0.99      1.00       984
           1       1.00      0.99      1.00      1139
           2       0.99      0.99      0.99      1033
           3       1.00      1.00      1.00      1011
           4       0.99      1.00      1.00       981
           5       1.00      0.99      0.99       897
           6       0.99      1.00      0.99       952
           7       0.99      0.99      0.99      1028
           8       0.99      0.99      0.99       973
           9       0.99      1.00      0.99      1002

    accuracy                           0.99     10000
   macro avg       0.99      0.99      0.99     10000
weighted avg       0.99      0.99      0.99     10000
```

ResNet18 has a initial convolutional layer with a kernel size of 7x7, padding of 3 and stride of 2, which produces an output of size 64x64x64.

Then there is a max pooling layer with kernel size 3x3, stride of 2 which reduces the output size to 32x32x64.

The residual blocks are composed of two convolutional layers with kernel size 3x3, padding of 1 and stride of 1. The number of filters starts at 64 and doubles after each downsampling stage. There are 4 stages in total, each with 2 residual blocks. So, the output size and number of filters in each stage are:

Stage 1: output size 32x32x64, 2 residual blocks, 64 filters
Stage 2: output size 16x16x128, 2 residual blocks, 128 filters
Stage 3: output size 8x8x256, 2 residual blocks, 256 filters
Stage 4: output size 4x4x512, 2 residual blocks, 512 filters
The global average pooling layer takes the output of the last residual block and computes the average of each channel, resulting in a feature map of size 512x1x1.

The final linear layer has 512 input features and 10 output features, one for each class, resulting in a weight matrix of size 10x512 and a bias vector of size 10.

To compute the total number of parameters, we need to count the parameters in each layer and add them up. The number of parameters in a convolutional layer is the sum of the number of weights (kernel size times number of input and output channels) and biases (one per output channel). The number of parameters in a linear layer is the sum of the number of weights (input size times output size) and biases (one per output channel).

So, the total number of parameters in ResNet18 with final linear layer adjusted for predicting 10 classes is:

Conv1: 7x7x3x64 + 64 = 9,472
Layer1: 3x3x64x64x2 + 64x2 = 37,248
Layer2: 3x3x64x128x2 + 128x2 = 148,480
Layer3: 3x3x128x256x2 + 256x2 = 594,176
Layer4: 3x3x256x512x2 + 512x2 = 2,359,808
FC: 512x10 + 10 = 5,130
Total: 3,154,314 parameters.

ResNet18 is a deep residual neural network architecture that has several advantages over the 9-layer CNN architecture, which could explain why it performed better. Here are some reasons:

1. Residual connections: ResNet18 uses residual connections, which allow gradients to flow through the network more easily, making it easier for the network to learn. This leads to better performance than architectures without residual connections, such as the 9-layer CNN.

2. Deeper architecture: ResNet18 has a deeper architecture with 18 layers, compared to the 9-layer CNN. This deeper architecture allows the network to learn more complex features, making it more accurate.

3. Pretrained weights: ResNet18 is often trained on a large dataset, such as ImageNet, and the weights are then fine-tuned for the specific task at hand. This pretraining helps to initialize the network with useful features, which can help the network learn more efficiently.

4. Global average pooling: ResNet18 uses global average pooling instead of fully connected layers at the end of the network. This reduces the number of parameters and can help prevent overfitting.

5. Batch normalization: ResNet18 uses batch normalization, which helps to normalize the input to each layer and can help prevent overfitting. Batch normalization can also help the network converge more quickly during training.

6. Skip connections: ResNet18 uses skip connections to connect layers that are not adjacent to each other, which helps to mitigate the vanishing gradient problem. This allows gradients to flow more easily through the network and can lead to better performance.

Q2. **Download the Flicker8k dataset [images, captions]. Implement an encoder-decoder architecture for Image Captioning. For the encoder and decoder, you can use resnet/densnet/VGG and LSTM/RNN/GRU respectivly. Perform the following tasks:**

1. Split the dataset into train and test sets appropriately. You can further split the train set for validation. Train your model on the train set. Report loss curve during training.

2. Choose an existing evaluation metric or propose your metric to evaluate your model. Specify the reason behind your selection/proposal of the metric. Report the final results on the test set.

Solution:-

For the comparative purpose of above, I utilized the following pretrained-CNNs as encoder, with 2-Layered LSTM as decoders (since it was taking lots of time, I did comparisons on 10 epochs, but for Resnet34, I took it till 50 epochs to see convergence):-

1. Squeezenet1_0: It's a lightweight CNN architecture that uses fire modules (combination of 1x1 and 3x3 convolutions) to reduce the number of parameters. It has a small model size and provides comparable accuracy to larger models.

2. Resnet_34: ResNet is a CNN architecture that uses residual blocks to address the vanishing gradient problem in deeper networks. ResNet-34 is a variant of the ResNet architecture with 34 layers, which has shown excellent performance in image classification tasks.

3. Swin_tiny: Swin Transformer is a new vision Transformer architecture that uses hierarchical windows of feature maps to efficiently capture information at different scales. Swin_tiny is a smaller version of the Swin Transformer, with a reduced number of layers and model size.

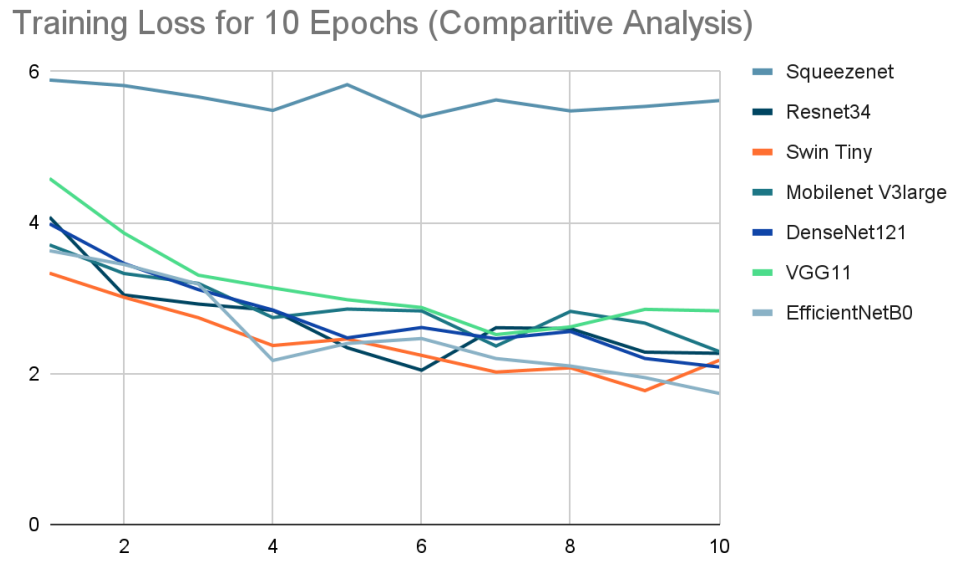
4. MobileNet_v3_large: MobileNet is a lightweight CNN architecture designed for mobile devices. MobileNet_v3_large is the latest version of MobileNet, which uses a combination of depthwise separable convolutions and linear bottlenecks to achieve high accuracy with low latency.

5. Densenet_121: DenseNet is a CNN architecture that uses dense blocks, where each layer receives feature maps from all preceding layers, to encourage feature reuse and reduce the number of parameters. Densenet_121 is a variant of the DenseNet architecture with 121 layers.

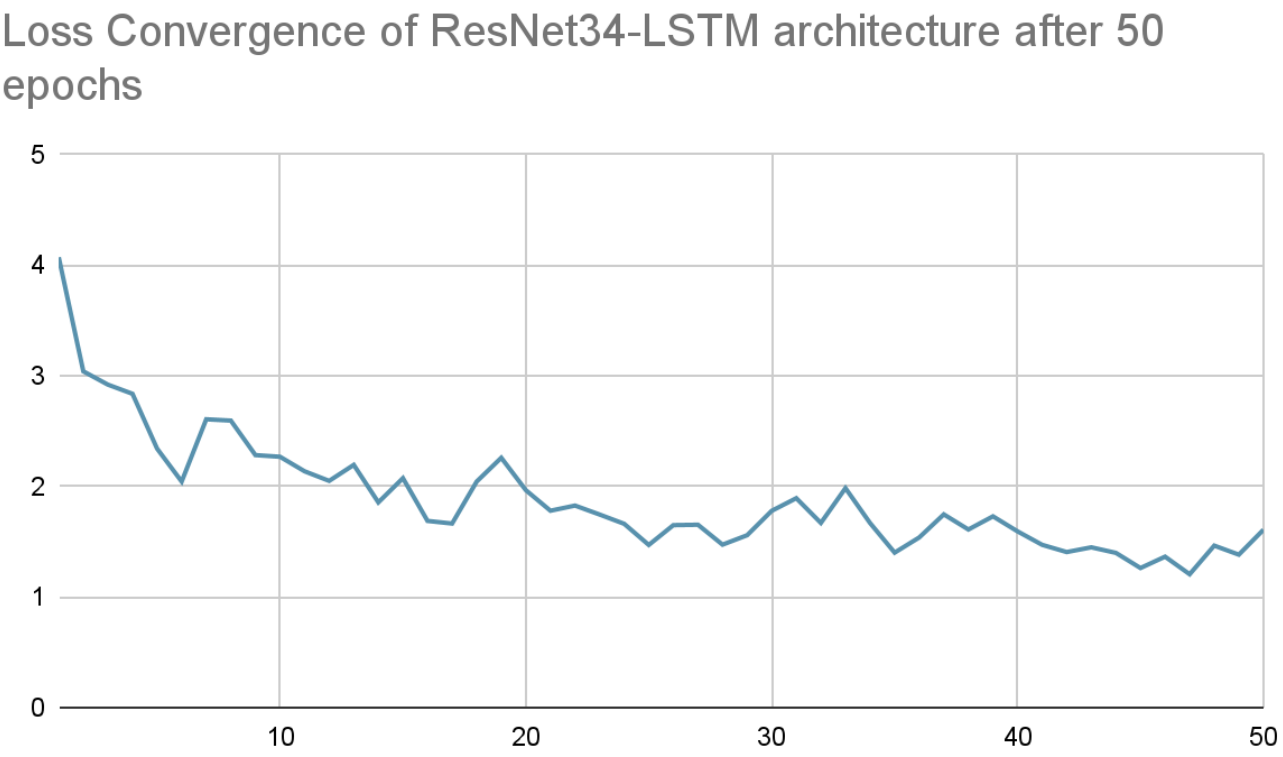
6. VGG_11: VGG is a CNN architecture that uses a series of 3x3 convolutions and max-pooling layers to gradually reduce the spatial resolution of the input. VGG_11 is a variant of the VGG architecture with 11 layers, which has shown good performance on image classification tasks.

7. EfficientNet_B0: EfficientNet is a family of CNN architectures that uses a compound scaling method to optimize the model size, depth, and width for a given resource budget. EfficientNet_B0 is the baseline model in the EfficientNet family, which has shown state-of-the-art performance on image classification tasks while maintaining a small model size.

Following are the average batch-wise losses after 10 epochs for each model architecture:-



Following is the Loss convergence for Resnet34 after taking 50 epochs:-



Due to late submission as well as time constraints because of end semester-based evaluations, I only focused on loss convergence graphs (sorry for the inconvenience). Apart from that, I also tested on an image prompt, and the following were the responses:-

1. trees bookcase trees bookcase trees bookcase trees bookcase trees bookcase trees bookcase trees bookcase trees bookcase trees bookcase trees bookcase trees

2. nine ractrack nine ractrack nine ractrack nine ractrack nine ractrack nine ractrack nine ractrack nine ractrack nine ractrack nine
3. sit up climbing climbing climbing climbing climbing climbing climbing climbing climbing climbing climbing climbing climbing climbing climbing climbing climbing climbing

4. beach bicycle chaps running chainmail doing running chainmail doing running chainmail doing running chainmail doing running chainmail doing running

5. beach of pink of pink of pink of pink of pink of pink of pink of pink of pink

6. passing mantle shallow woman devices he white from denim toy palying photograph white from denim toy palying photograph white

7. brick old old old old old old old old old old old old old old old old old old

Q3. **Use the dataset from Assignment 3 (Q. 4). Train YOLO object detection model (any version) on the train set. Compute the AP for the test set and compare the result with the HOG detector. Show some visual results and compare both of the methods.**

Solution:-

Since the dataset was scarce, I utilized Roboflow's services to augment the dataset, resize it and introduce 2 different augmentations (flip and blur).

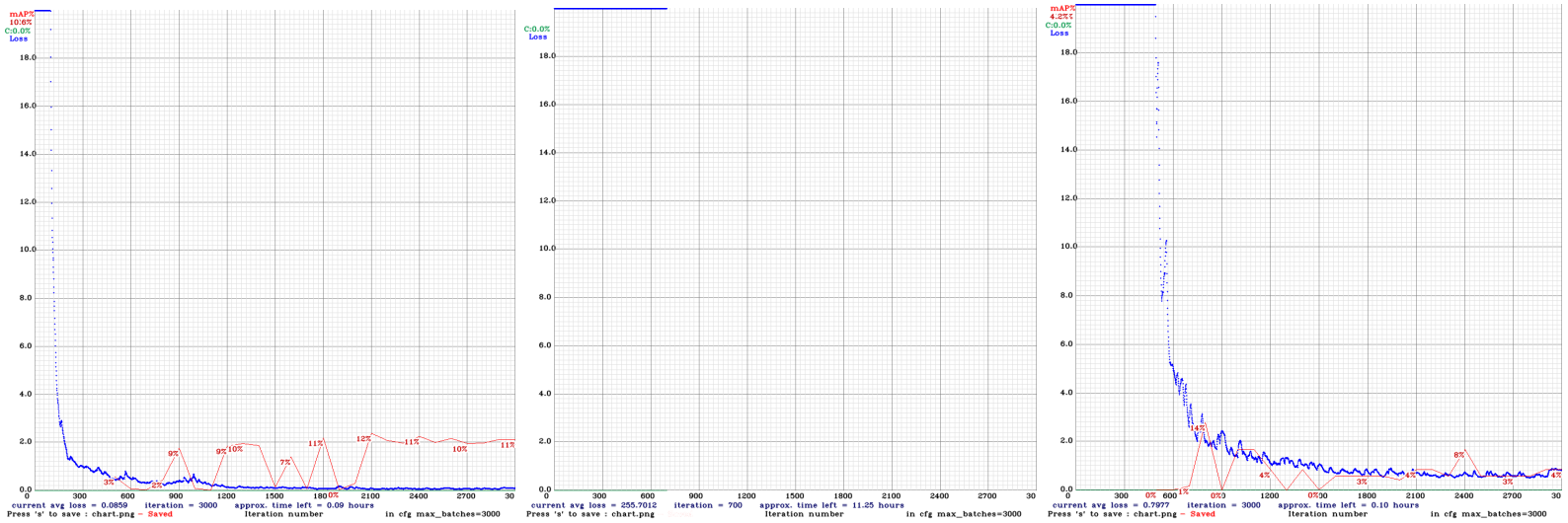
Apart from this, I made a comparative analysis on 3 different variants of Yolo:-

1. YOLOv3-Tiny: It is a lightweight version of YOLOv3, which is designed to have fewer parameters and faster processing time. It has a simplified architecture with fewer layers, making it suitable for deployment on mobile and embedded devices.

2. YOLOv3: It is an object detection model that uses a single neural network to predict bounding boxes and class probabilities for multiple objects in an image. YOLOv3 introduces several improvements over previous versions, including the use of a feature pyramid network and anchor boxes for better accuracy.

3. YOLOv2-Tiny: This is the lightweight version of YOLOv2, which has a simpler architecture and fewer parameters than the original YOLOv2. It uses a single-scale feature map and anchor boxes to predict object locations and class probabilities, making it faster and more efficient.

Following were the mAP graphs alongside confidence and loss for these variations (Leftmost Yolov2-Tiny, then Yolov3 and Rightmost Yolov3-Tiny):-



I might not be able to attach the predicted boxes for Yolo variants as of now, because I am going through final revision for CV end-semester (Sorry for being too late), but initial boxes were way better than SVM+HOG detector, because of the following reasons:-

1. Speed: YOLO variants use a single neural network to detect objects in an image, making it faster than the traditional SVM+HOG detector.

2. Accuracy: YOLO variants achieve higher accuracy than SVM+HOG detectors because they use a deep learning-based approach to detect objects.

3. Robustness: YOLO variants are more robust to different lighting conditions, camera angles, and object sizes, making them suitable for real-world applications.

4. End-to-end learning: YOLO variants learn to detect objects end-to-end, which means that they learn to extract features and classify objects simultaneously, making the detection process more efficient.

5. Object tracking: YOLO variants can track objects across frames, which makes them suitable for real-time video analysis.

6. Flexibility: YOLO variants can detect multiple objects of different classes in a single image, making them flexible for various use cases. In contrast, SVM+HOG detector is limited to detecting a single object class at a time.