# UNIVERSITY OF TEHRAN

**COLLEGE OF ENGINEERING**

**DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING**

# NEURAL NETWORK & DEEP LEARNING

MINI PROJECT#1

MOHAMMAD HEYDARI

810197494

SIAVASH SHAMS

810197644

**UNDER SUPERVISION OF:**

DR. AHMAD KALHOR

SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING

UNIVERSITY OF TEHRAN

***Apr. 2022***

# 1 CONTENTS

## 2 QUESTION #1: CNN(CLASSIFICATION)

In this part we intend to implement a Convolutional Neural Network in case of image classification and feature extraction.

In this section We have used different convolutional layers such as max-pooling layer, batch-normalization layer and many others to satisfy our goal which is all about increasing the classifier accuracy.

As we know the most important part of our implementation is about our pipeline architecture, so further we have provided a usable figure which properly visualizes all we need throughout this part of project:

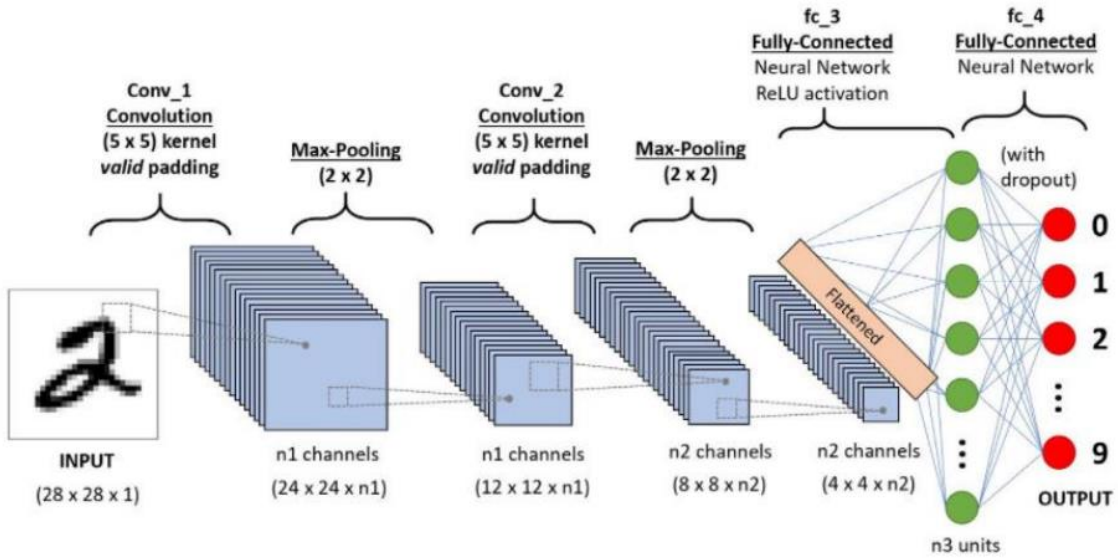



**Figure1.** CNN Classifier Architecture

**As you can see, we are dealing with three main layers which abbreviated below:**

1- 3D-Convolutional Layers in case of feature extraction.

2- Max-Pooling layers in case of image size reduction.

3- Fully-Connected Layers in case of Classification task.

**Pipeline Procedure:**

In the first step we pass the input image to the first Convolution layer for extracting the features and also for image size reduction, afterward we repeat filtering task one more time to obtain the size-reducing form of input image, furthermore we have implemented the flatten layer which gives us all feature information in case of a suitable vector for passing through fully-connected layers topology.

Finally, we have used softmax activation-function to do multi-class classification task as well as it could be.

Now after a brief contextualization we are dealing with four separated implementation to satisfy all of question requirements.

As of question description, we have reported figure of Error and Accuracy variation among epochs for both train and validation data in two separated curves.

As of a supplementary approach we have also reported the training-time, confusion-matrix and also test final accuracy in each case.

## 2.1 IMPLEMENTING JUST USING CONVOLUTIONAL-LAYERS

**Further you can see the training process:**

```
Epoch 1/20
625/625 [==============================] - 48s 75ms/step - loss: 1.5993 - accuracy: 0.4534 - val_loss: 1.2390 - val_accuracy: 0.5579
Epoch 2/20
625/625 [==============================] - 47s 75ms/step - loss: 1.0590 - accuracy: 0.6248 - val_loss: 1.0107 - val_accuracy: 0.6493
Epoch 3/20
625/625 [==============================] - 48s 77ms/step - loss: 0.8322 - accuracy: 0.7077 - val_loss: 0.9190 - val_accuracy: 0.6820
Epoch 4/20
625/625 [==============================] - 46s 74ms/step - loss: 0.6085 - accuracy: 0.7878 - val_loss: 0.9835 - val_accuracy: 0.6763
Epoch 5/20
625/625 [==============================] - 46s 74ms/step - loss: 0.3239 - accuracy: 0.8878 - val_loss: 1.1720 - val_accuracy: 0.6753
Epoch 6/20
625/625 [==============================] - 46s 74ms/step - loss: 0.1463 - accuracy: 0.9500 - val_loss: 1.6779 - val_accuracy: 0.6495
Epoch 7/20
625/625 [==============================] - 46s 73ms/step - loss: 0.1057 - accuracy: 0.9643 - val_loss: 2.0934 - val_accuracy: 0.6602
Epoch 8/20
625/625 [==============================] - 48s 77ms/step - loss: 0.0835 - accuracy: 0.9718 - val_loss: 1.9129 - val_accuracy: 0.6571
Epoch 9/20
625/625 [==============================] - 46s 74ms/step - loss: 0.0645 - accuracy: 0.9784 - val_loss: 2.1368 - val_accuracy: 0.6586
Epoch 10/20
625/625 [==============================] - 48s 77ms/step - loss: 0.0704 - accuracy: 0.9761 - val_loss: 2.1459 - val_accuracy: 0.6575
Epoch 11/20
625/625 [==============================] - 48s 77ms/step - loss: 0.0571 - accuracy: 0.9816 - val_loss: 2.3041 - val_accuracy: 0.6549
Epoch 12/20
625/625 [==============================] - 48s 77ms/step - loss: 0.0619 - accuracy: 0.9807 - val_loss: 2.2597 - val_accuracy: 0.6583
Epoch 13/20
625/625 [==============================] - 46s 74ms/step - loss: 0.0392 - accuracy: 0.9867 - val_loss: 2.5389 - val_accuracy: 0.6404
Epoch 14/20
625/625 [==============================] - 48s 77ms/step - loss: 0.0548 - accuracy: 0.9829 - val_loss: 2.3591 - val_accuracy: 0.6540
Epoch 15/20
625/625 [==============================] - 48s 77ms/step - loss: 0.0437 - accuracy: 0.9863 - val_loss: 2.6697 - val_accuracy: 0.6285
Epoch 16/20
625/625 [==============================] - 48s 77ms/step - loss: 0.0401 - accuracy: 0.9872 - val_loss: 2.6943 - val_accuracy: 0.6400
Epoch 17/20
625/625 [==============================] - 48s 77ms/step - loss: 0.0450 - accuracy: 0.9851 - val_loss: 2.9846 - val_accuracy: 0.6362
Epoch 18/20
625/625 [==============================] - 46s 74ms/step - loss: 0.0437 - accuracy: 0.9864 - val_loss: 2.9165 - val_accuracy: 0.6454
Epoch 19/20
625/625 [==============================] - 48s 77ms/step - loss: 0.0385 - accuracy: 0.9880 - val_loss: 3.1239 - val_accuracy: 0.6185
Epoch 20/20
625/625 [==============================] - 49s 78ms/step - loss: 0.0290 - accuracy: 0.9912 - val_loss: 3.4342 - val_accuracy: 0.6426
> 64.256
```

As we can see the accuracy on the train-data is a suitable value.

```
Training time: 947.9825377464294s
test loss:  3.4342384338378906
test acc:  64.25642371177673
confusion matrix:

 [[713  36  31  19  16  10  10  33  98  34]
 [ 47 755  13  14   7   7   2  16  36 103]
 [139  28 334  93  65 128  55 102  39  17]
 [ 88  38  63 331  42 216  56  99  26  41]
 [ 87  31  66 124 296  89  55 203  20  29]
 [ 46  12  39 141  28 554  35 115  16  14]
 [ 35  54  46 124  31  79 533  40  30  28]
 [ 41   6  29  39  28  86   5 738   8  20]
 [149  48  17  17   7   7   6  23 675  51]
 [ 60 139   5  20  10  22   2  56  44 642]]
```

**Figure2.** Test Results

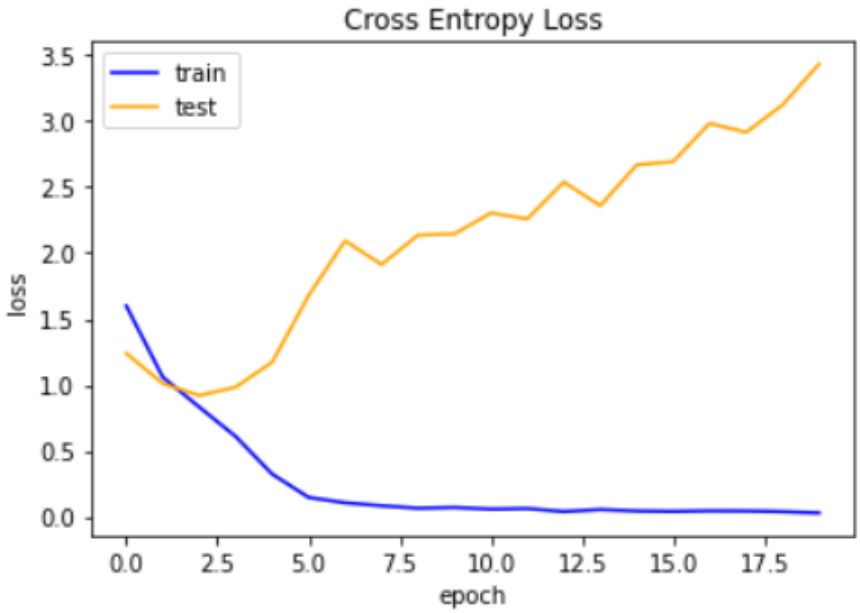



**Figure3.** Cross Entropy Loss

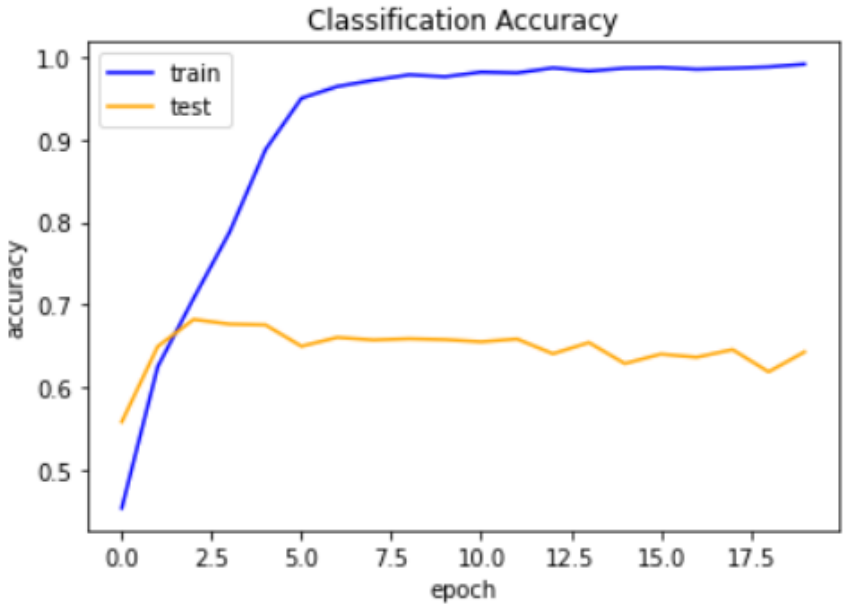



**Figure4.** Classification Accuracy

## 2.2 ADDING MAX-POOLING & BATCH-NORMALIZATION LAYERS

**Batch-Normalization:**

Batch normalization scales layers outputs to have mean 0 and variance 1. The outputs are scaled such a way to train the network faster. It also reduces problems due to poor parameter initialization.

Specifically, batch normalization normalizes the output of a previous layer by subtracting the batch mean and dividing by the batch standard deviation.

This is much similar to feature scaling which is done to speed up the learning process and converge to a solution.

If the distribution of the inputs to every layer is the same, the network is efficient. Batch normalization standardizes the distribution of layer inputs to combat the internal covariance shift. It controls the amount by which the hidden units shift.

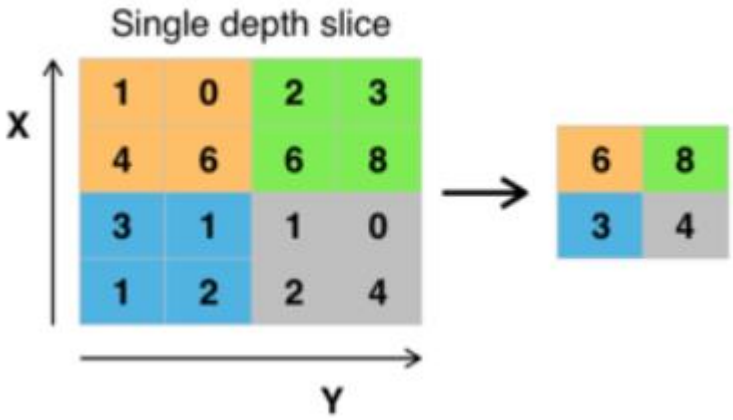
**Pooling:**

Pooling is nothing other than down sampling of an image. The most common pooling layer filter is of size 2x2, which discards three forth of the activations. Role of pooling layer is to reduce the resolution of the feature map but retaining features of the map required for classification through translational and rotational invariants. In addition to spatial invariance robustness, pooling will reduce the computation cost by a great deal.

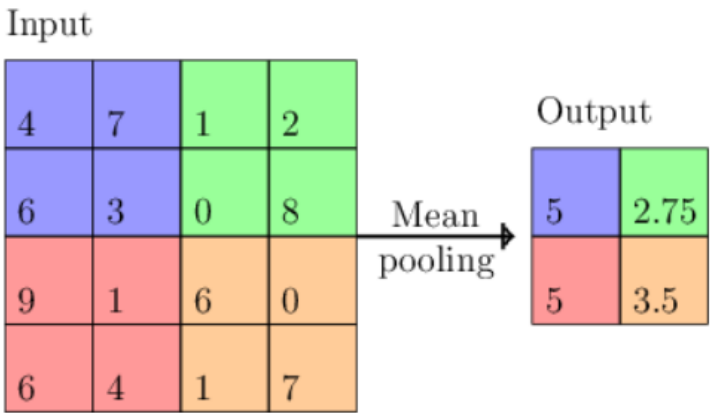
- Backpropagation is used for training of pooling operation
- It again helps the processor to process things faster.

There are many pooling techniques. They are as follows:

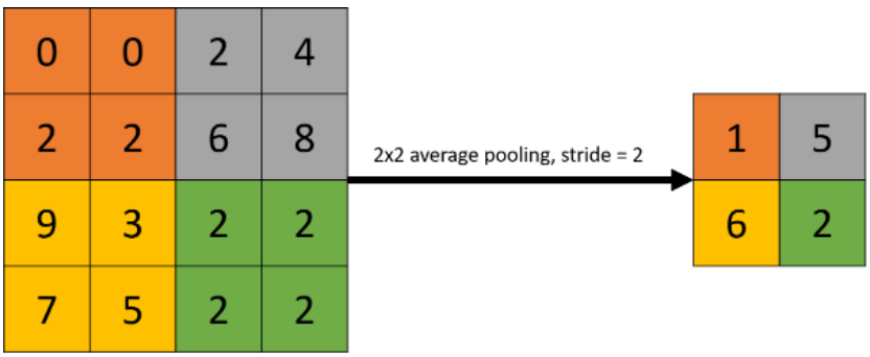
**Max pooling** where we take largest of the pixel values of a segment.




**Mean pooling** where we take largest of the pixel values of a segment.




**Avg-pooling** where we take largest of the pixel values of a segment.



**Further you can see the training process in this section:**

As we can see, 10 epochs are suitable for training.

```
Epoch 1/20
625/625 [==============================] - 14s 21ms/step - loss: 1.2726 - accuracy: 0.5549 - val_loss: 1.1316 - val_accuracy: 0.6202
Epoch 2/20
625/625 [==============================] - 12s 19ms/step - loss: 0.8566 - accuracy: 0.7004 - val_loss: 1.0320 - val_accuracy: 0.6497
Epoch 3/20
625/625 [==============================] - 13s 20ms/step - loss: 0.6464 - accuracy: 0.7755 - val_loss: 0.9796 - val_accuracy: 0.6710
Epoch 4/20
625/625 [==============================] - 12s 19ms/step - loss: 0.4468 - accuracy: 0.8493 - val_loss: 1.0827 - val_accuracy: 0.6623
Epoch 5/20
625/625 [==============================] - 13s 20ms/step - loss: 0.2760 - accuracy: 0.9128 - val_loss: 1.1940 - val_accuracy: 0.6630
Epoch 6/20
625/625 [==============================] - 13s 20ms/step - loss: 0.1587 - accuracy: 0.9526 - val_loss: 1.2941 - val_accuracy: 0.6676
Epoch 7/20
625/625 [==============================] - 13s 20ms/step - loss: 0.1034 - accuracy: 0.9713 - val_loss: 1.5259 - val_accuracy: 0.6633
Epoch 8/20
625/625 [==============================] - 12s 20ms/step - loss: 0.0848 - accuracy: 0.9740 - val_loss: 1.5379 - val_accuracy: 0.6618
Epoch 9/20
625/625 [==============================] - 12s 20ms/step - loss: 0.0822 - accuracy: 0.9740 - val_loss: 1.5767 - val_accuracy: 0.6553
Epoch 10/20
625/625 [==============================] - 12s 19ms/step - loss: 0.0694 - accuracy: 0.9774 - val_loss: 1.6583 - val_accuracy: 0.6535
Epoch 11/20
625/625 [==============================] - 12s 20ms/step - loss: 0.0613 - accuracy: 0.9802 - val_loss: 1.6181 - val_accuracy: 0.6565
Epoch 12/20
625/625 [==============================] - 12s 19ms/step - loss: 0.0593 - accuracy: 0.9803 - val_loss: 1.8839 - val_accuracy: 0.6495
Epoch 13/20
625/625 [==============================] - 12s 19ms/step - loss: 0.0501 - accuracy: 0.9837 - val_loss: 1.9413 - val_accuracy: 0.6555
Epoch 14/20
625/625 [==============================] - 12s 19ms/step - loss: 0.0405 - accuracy: 0.9866 - val_loss: 1.9635 - val_accuracy: 0.6658
Epoch 15/20
625/625 [==============================] - 12s 19ms/step - loss: 0.0454 - accuracy: 0.9856 - val_loss: 2.0227 - val_accuracy: 0.6484
Epoch 16/20
625/625 [==============================] - 12s 19ms/step - loss: 0.0423 - accuracy: 0.9861 - val_loss: 1.9324 - val_accuracy: 0.6627
Epoch 17/20
625/625 [==============================] - 12s 19ms/step - loss: 0.0328 - accuracy: 0.9895 - val_loss: 1.8632 - val_accuracy: 0.6700
Epoch 18/20
625/625 [==============================] - 12s 19ms/step - loss: 0.0406 - accuracy: 0.9873 - val_loss: 1.9531 - val_accuracy: 0.6629
Epoch 19/20
625/625 [==============================] - 12s 20ms/step - loss: 0.0365 - accuracy: 0.9870 - val_loss: 1.9569 - val_accuracy: 0.6705
Epoch 20/20
625/625 [==============================] - 12s 19ms/step - loss: 0.0366 - accuracy: 0.9876 - val_loss: 1.9948 - val_accuracy: 0.6750
> 67.497
```

As we can see the accuracy on the train-data is a suitable value.

```
Training time: 246.03012609481812s
test loss:  1.9947682619094849
test acc:  67.49674677848816
confusion matrix:

 [[376 256   8  14   0  19   5 101 183  38]
 [ 26 802   2   3   0  12   1  21  85  48]
 [ 87 145  85  61   0 238  54 163 142  25]
 [ 53 149  24 145   1 299  12 100 183  34]
 [ 39 172  12  37   1 257  80 245 132  25]
 [ 28  95  13  83   0 510  14 116 122  19]
 [ 29 178  14  45   0 230 266  58 162  18]
 [ 32 118  14  20   0  79   3 603  74  57]
 [ 42 266   0   4   0   7   0  34 597  50]
 [ 32 312   3   7   0   9   2  49 121 465]]
```

**Figure5.** Test Results

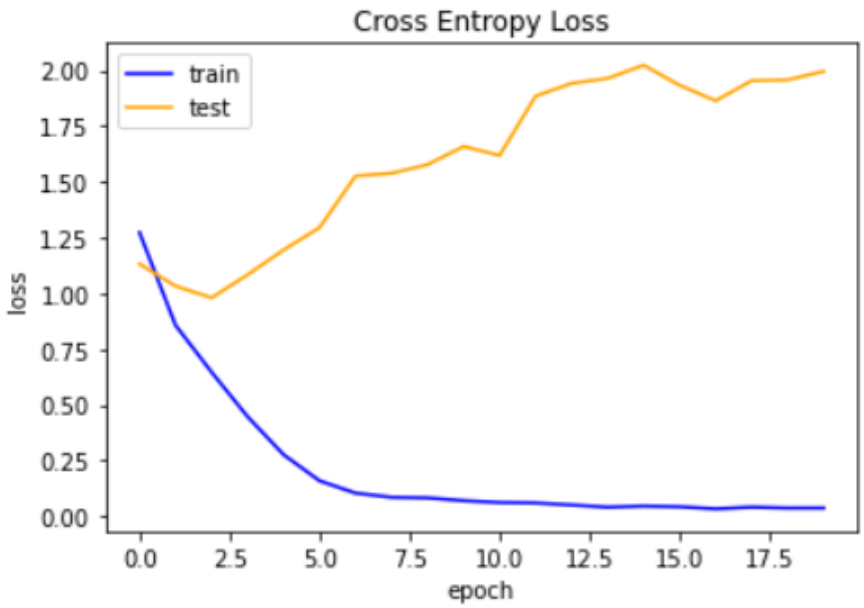



**Figure6.** Cross Entropy Loss

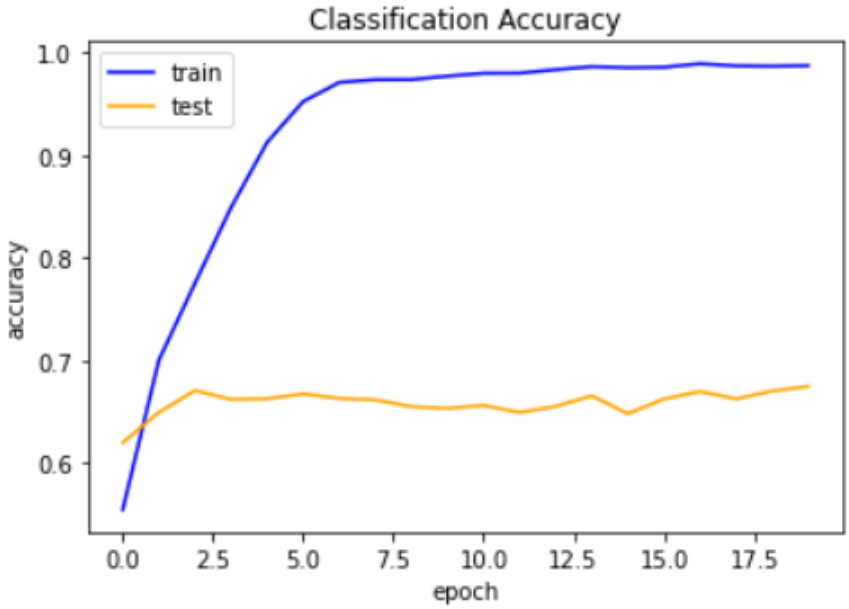



**Figure7.** Classification Accuracy

## 2.3 ADDING DROPOUT LAYER AND ANALYSING RESULTS

**Why we use dropout?**

Dropout is a regularization technique for reducing overfitting in artificial neural networks by preventing complex co-adaptations on training data. It is an efficient way of performing model averaging with neural networks. The term dilution refers to the thinning of the weights

## Further you can see the training process in this section:

```
Epoch 1/60
625/625 [==============================] - 20s 29ms/step - loss: 1.5263 - accuracy: 0.4545 - val_loss: 1.3354 - val_accuracy: 0.5302
Epoch 2/60
625/625 [==============================] - 17s 28ms/step - loss: 1.1479 - accuracy: 0.5906 - val_loss: 1.2705 - val_accuracy: 0.5707
Epoch 3/60
625/625 [==============================] - 17s 28ms/step - loss: 0.9663 - accuracy: 0.6562 - val_loss: 1.2749 - val_accuracy: 0.5781
Epoch 4/60
625/625 [==============================] - 18s 28ms/step - loss: 0.8425 - accuracy: 0.7042 - val_loss: 0.8262 - val_accuracy: 0.7129
Epoch 5/60
625/625 [==============================] - 18s 28ms/step - loss: 0.7588 - accuracy: 0.7329 - val_loss: 0.7219 - val_accuracy: 0.7446
Epoch 6/60
625/625 [==============================] - 18s 29ms/step - loss: 0.6958 - accuracy: 0.7560 - val_loss: 0.6497 - val_accuracy: 0.7749
Epoch 7/60
625/625 [==============================] - 18s 28ms/step - loss: 0.6483 - accuracy: 0.7700 - val_loss: 0.5964 - val_accuracy: 0.7935
Epoch 8/60
625/625 [==============================] - 18s 28ms/step - loss: 0.5992 - accuracy: 0.7871 - val_loss: 0.6394 - val_accuracy: 0.7802
Epoch 9/60
625/625 [==============================] - 18s 28ms/step - loss: 0.5612 - accuracy: 0.8005 - val_loss: 0.6022 - val_accuracy: 0.7964
Epoch 10/60
625/625 [==============================] - 17s 28ms/step - loss: 0.5247 - accuracy: 0.8157 - val_loss: 0.5436 - val_accuracy: 0.8136
Epoch 11/60
625/625 [==============================] - 17s 28ms/step - loss: 0.4941 - accuracy: 0.8253 - val_loss: 0.5757 - val_accuracy: 0.8022
Epoch 12/60
625/625 [==============================] - 18s 28ms/step - loss: 0.4662 - accuracy: 0.8334 - val_loss: 0.5474 - val_accuracy: 0.8141
Epoch 13/60
625/625 [==============================] - 18s 28ms/step - loss: 0.4488 - accuracy: 0.8419 - val_loss: 0.5503 - val_accuracy: 0.8116
Epoch 14/60
625/625 [==============================] - 18s 28ms/step - loss: 0.4238 - accuracy: 0.8492 - val_loss: 0.5323 - val_accuracy: 0.8209
Epoch 15/60
625/625 [==============================] - 18s 28ms/step - loss: 0.4101 - accuracy: 0.8545 - val_loss: 0.5127 - val_accuracy: 0.8263
Epoch 16/60
625/625 [==============================] - 18s 28ms/step - loss: 0.3947 - accuracy: 0.8603 - val_loss: 0.5000 - val_accuracy: 0.8316
Epoch 17/60
625/625 [==============================] - 18s 28ms/step - loss: 0.3762 - accuracy: 0.8661 - val_loss: 0.5317 - val_accuracy: 0.8231
Epoch 18/60
625/625 [==============================] - 18s 28ms/step - loss: 0.3585 - accuracy: 0.8715 - val_loss: 0.5096 - val_accuracy: 0.8344
Epoch 19/60
625/625 [==============================] - 18s 28ms/step - loss: 0.3435 - accuracy: 0.8793 - val_loss: 0.5251 - val_accuracy: 0.8258
Epoch 20/60
625/625 [==============================] - 18s 28ms/step - loss: 0.3319 - accuracy: 0.8808 - val_loss: 0.4934 - val_accuracy: 0.8412
Epoch 21/60
625/625 [==============================] - 18s 28ms/step - loss: 0.3184 - accuracy: 0.8871 - val_loss: 0.4809 - val_accuracy: 0.8404
Epoch 22/60
625/625 [==============================] - 18s 28ms/step - loss: 0.3106 - accuracy: 0.8891 - val_loss: 0.5351 - val_accuracy: 0.8297
Epoch 23/60
625/625 [==============================] - 17s 28ms/step - loss: 0.3032 - accuracy: 0.8917 - val_loss: 0.5800 - val_accuracy: 0.8171
Epoch 24/60
625/625 [==============================] - 17s 28ms/step - loss: 0.2924 - accuracy: 0.8933 - val_loss: 0.4957 - val_accuracy: 0.8421
Epoch 25/60
625/625 [==============================] - 18s 29ms/step - loss: 0.2792 - accuracy: 0.9004 - val_loss: 0.5304 - val_accuracy: 0.8360
Epoch 26/60
625/625 [==============================] - 19s 30ms/step - loss: 0.2755 - accuracy: 0.9027 - val_loss: 0.5587 - val_accuracy: 0.8266
Epoch 27/60
625/625 [==============================] - 18s 30ms/step - loss: 0.2685 - accuracy: 0.9032 - val_loss: 0.4944 - val_accuracy: 0.8476
Epoch 28/60
625/625 [==============================] - 18s 30ms/step - loss: 0.2593 - accuracy: 0.9078 - val_loss: 0.4982 - val_accuracy: 0.8447
Epoch 29/60
625/625 [==============================] - 18s 29ms/step - loss: 0.2488 - accuracy: 0.9110 - val_loss: 0.5049 - val_accuracy: 0.8416
Epoch 30/60
625/625 [==============================] - 18s 29ms/step - loss: 0.2518 - accuracy: 0.9105 - val_loss: 0.5257 - val_accuracy: 0.8388
Epoch 31/60
625/625 [==============================] - 18s 29ms/step - loss: 0.2391 - accuracy: 0.9133 - val_loss: 0.4906 - val_accuracy: 0.8510
Epoch 32/60
625/625 [==============================] - 18s 29ms/step - loss: 0.2357 - accuracy: 0.9148 - val_loss: 0.5185 - val_accuracy: 0.8461
Epoch 33/60
625/625 [==============================] - 18s 28ms/step - loss: 0.2312 - accuracy: 0.9179 - val_loss: 0.5006 - val_accuracy: 0.8440
Epoch 34/60
625/625 [==============================] - 18s 28ms/step - loss: 0.2217 - accuracy: 0.9196 - val_loss: 0.5637 - val_accuracy: 0.8377
Epoch 35/60
625/625 [==============================] - 17s 28ms/step - loss: 0.2188 - accuracy: 0.9215 - val_loss: 0.5227 - val_accuracy: 0.8434
Epoch 36/60
625/625 [==============================] - 18s 28ms/step - loss: 0.2098 - accuracy: 0.9242 - val_loss: 0.5122 - val_accuracy: 0.8462
Epoch 37/60
625/625 [==============================] - 18s 28ms/step - loss: 0.2069 - accuracy: 0.9262 - val_loss: 0.5318 - val_accuracy: 0.8478
Epoch 38/60
625/625 [==============================] - 18s 29ms/step - loss: 0.2116 - accuracy: 0.9242 - val_loss: 0.5265 - val_accuracy: 0.8445
Epoch 39/60
625/625 [==============================] - 18s 28ms/step - loss: 0.1992 - accuracy: 0.9287 - val_loss: 0.5287 - val_accuracy: 0.8512
Epoch 40/60
625/625 [==============================] - 18s 28ms/step - loss: 0.1992 - accuracy: 0.9282 - val_loss: 0.5531 - val_accuracy: 0.8423
Epoch 41/60
625/625 [==============================] - 18s 28ms/step - loss: 0.1864 - accuracy: 0.9326 - val_loss: 0.5482 - val_accuracy: 0.8415
```

```
Epoch 42/60
625/625 [==============================] - 18s 28ms/step - loss: 0.1960 - accuracy: 0.9308 - val_loss: 0.5229 - val_accuracy: 0.8534
Epoch 43/60
625/625 [==============================] - 18s 28ms/step - loss: 0.1887 - accuracy: 0.9330 - val_loss: 0.5929 - val_accuracy: 0.8360
Epoch 44/60
625/625 [==============================] - 18s 29ms/step - loss: 0.1829 - accuracy: 0.9347 - val_loss: 0.5652 - val_accuracy: 0.8421
Epoch 45/60
625/625 [==============================] - 18s 29ms/step - loss: 0.1832 - accuracy: 0.9344 - val_loss: 0.5283 - val_accuracy: 0.8542
Epoch 46/60
625/625 [==============================] - 18s 29ms/step - loss: 0.1763 - accuracy: 0.9376 - val_loss: 0.5963 - val_accuracy: 0.8393
Epoch 47/60
625/625 [==============================] - 18s 28ms/step - loss: 0.1788 - accuracy: 0.9363 - val_loss: 0.5665 - val_accuracy: 0.8432
Epoch 48/60
625/625 [==============================] - 17s 28ms/step - loss: 0.1748 - accuracy: 0.9374 - val_loss: 0.5472 - val_accuracy: 0.8486
Epoch 49/60
625/625 [==============================] - 17s 28ms/step - loss: 0.1719 - accuracy: 0.9380 - val_loss: 0.6177 - val_accuracy: 0.8354
Epoch 50/60
625/625 [==============================] - 18s 28ms/step - loss: 0.1678 - accuracy: 0.9401 - val_loss: 0.5152 - val_accuracy: 0.8540
Epoch 51/60
625/625 [==============================] - 18s 28ms/step - loss: 0.1659 - accuracy: 0.9409 - val_loss: 0.5503 - val_accuracy: 0.8493
Epoch 52/60
625/625 [==============================] - 17s 28ms/step - loss: 0.1696 - accuracy: 0.9387 - val_loss: 0.5490 - val_accuracy: 0.8507
Epoch 53/60
625/625 [==============================] - 18s 28ms/step - loss: 0.1611 - accuracy: 0.9427 - val_loss: 0.5841 - val_accuracy: 0.8369
Epoch 54/60
625/625 [==============================] - 18s 28ms/step - loss: 0.1643 - accuracy: 0.9421 - val_loss: 0.5457 - val_accuracy: 0.8539
Epoch 55/60
625/625 [==============================] - 18s 29ms/step - loss: 0.1610 - accuracy: 0.9422 - val_loss: 0.5333 - val_accuracy: 0.8535
Epoch 56/60
625/625 [==============================] - 18s 29ms/step - loss: 0.1562 - accuracy: 0.9445 - val_loss: 0.5998 - val_accuracy: 0.8436
Epoch 57/60
625/625 [==============================] - 18s 29ms/step - loss: 0.1558 - accuracy: 0.9448 - val_loss: 0.5676 - val_accuracy: 0.8458
Epoch 58/60
625/625 [==============================] - 18s 28ms/step - loss: 0.1533 - accuracy: 0.9443 - val_loss: 0.5926 - val_accuracy: 0.8407
Epoch 59/60
625/625 [==============================] - 18s 29ms/step - loss: 0.1484 - accuracy: 0.9467 - val_loss: 0.6024 - val_accuracy: 0.8374
Epoch 60/60
625/625 [==============================] - 18s 29ms/step - loss: 0.1509 - accuracy: 0.9469 - val_loss: 0.5532 - val_accuracy: 0.8502
> 85.019
```

As we can see the accuracy on the train-data is a suitable value.

```
Training time: 1103.7988185882568s
test loss:  0.5531874895095825
test acc:  85.01850366592407
confusion matrix:

 [[487 354  36   0   5   0   0   3   7 108]
 [  9 958   2   0   0   0   0   1   0  30]
 [161 548 157   7   6   7   0  36   7  71]
 [136 532  21  58   9  67   1  86   3  87]
 [118 558  29   5  52  20   0 121   4  93]
 [ 37 537  14  18   4 202   0 125   1  62]
 [113 678   4  13  10   7   1  71  13  90]
 [ 51 461  16   7   2  10   0 386   0  67]
 [149 437   7   2   1   0   0   1 191 212]
 [ 22 507   1   4   2   3   0   7   2 452]]
```

**Figure8.** Test Results

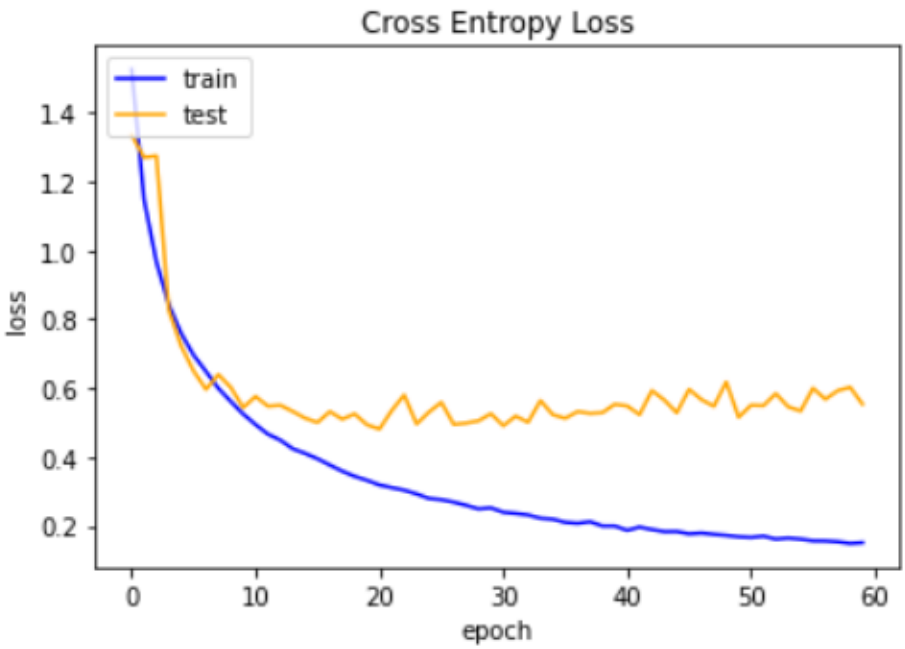



**Figure9.** Cross Entropy Loss

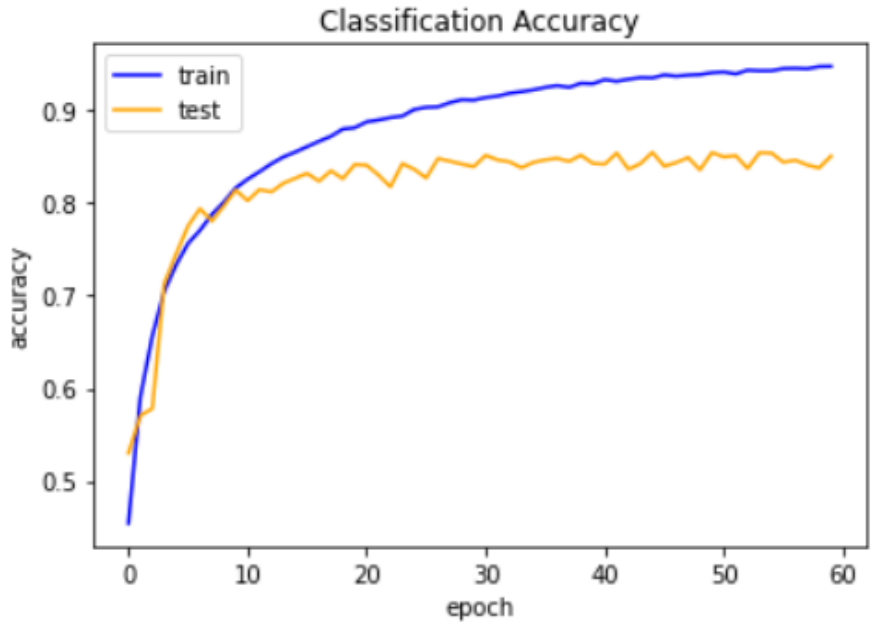



**Figure10.** Classification Accuracy

## 2.4 EARLY STOPPING APPROACH IN NEURAL NETWORKS

A problem with training neural networks is in the choice of the number of training epochs to use. Too many epochs can lead to overfitting of the training dataset, whereas too few may result in an underfit model.

Early stopping is a method that allows you to specify an arbitrarily large number of training epochs and stop training once the model performance stops improving on the validation dataset.

**Monitoring Performance: (metrics can be used)**

The performance of the model must be monitored during training.

This requires the choice of a dataset that is used to evaluate the model and a metric used to evaluate the model.

It is common to split the training dataset and use a subset, such as 30%, as a validation dataset used to monitor performance of the model during training. This validation set is not used to train the model. It is also common to use the loss on a validation dataset as the metric to monitor, although you may also use prediction error in the case of regression, or accuracy in the case of classification.

The loss of the model on the training dataset will also be available as part of the training procedure, and additional metrics may also be calculated and monitored on the training dataset.

Performance of the model is evaluated on the validation set at the end of each epoch, which adds an additional computational cost during training. This can be reduced by evaluating the model less frequently, such as every 2, 5, or 10 training epochs.

**Some of the most important metrics can be monitored during training:**

1) Validation-Loss (most common usage)

2) Training-loss (as mentioned above)

3) Precision

4) Recall

5) F-measure

**Implementation Early-Stopping:**

in this section we are going to implement Early-Stopping on the best model which obtained in the previous section.

Further you can see that the training process has been stopped at the special number of epochs to prevent overfitting:

```
Epoch 1/60
625/625 [==============================] - 20s 30ms/step - loss: 1.5356 - accuracy: 0.4478 - val_loss: 1.3643 - val_accuracy: 0.5171
Epoch 2/60
625/625 [==============================] - 18s 29ms/step - loss: 1.1485 - accuracy: 0.5902 - val_loss: 0.9521 - val_accuracy: 0.6587
Epoch 3/60
625/625 [==============================] - 18s 28ms/step - loss: 0.9504 - accuracy: 0.6640 - val_loss: 0.8136 - val_accuracy: 0.7173
Epoch 4/60
625/625 [==============================] - 18s 28ms/step - loss: 0.8186 - accuracy: 0.7107 - val_loss: 0.7579 - val_accuracy: 0.7327
Epoch 5/60
625/625 [==============================] - 18s 29ms/step - loss: 0.7463 - accuracy: 0.7362 - val_loss: 0.7399 - val_accuracy: 0.7399
Epoch 6/60
625/625 [==============================] - 18s 28ms/step - loss: 0.6836 - accuracy: 0.7600 - val_loss: 0.6501 - val_accuracy: 0.7760
Epoch 7/60
625/625 [==============================] - 18s 29ms/step - loss: 0.6326 - accuracy: 0.7761 - val_loss: 0.6111 - val_accuracy: 0.7918
Epoch 8/60
625/625 [==============================] - 18s 29ms/step - loss: 0.5877 - accuracy: 0.7938 - val_loss: 0.6139 - val_accuracy: 0.7829
Epoch 9/60
625/625 [==============================] - 18s 29ms/step - loss: 0.5470 - accuracy: 0.8068 - val_loss: 0.6410 - val_accuracy: 0.7813
Epoch 10/60
625/625 [==============================] - 18s 29ms/step - loss: 0.5215 - accuracy: 0.8170 - val_loss: 0.6021 - val_accuracy: 0.7977
Epoch 11/60
625/625 [==============================] - 18s 28ms/step - loss: 0.4852 - accuracy: 0.8298 - val_loss: 0.5102 - val_accuracy: 0.8240
Epoch 12/60
625/625 [==============================] - 18s 29ms/step - loss: 0.4674 - accuracy: 0.8342 - val_loss: 0.5712 - val_accuracy: 0.8088
Epoch 13/60
625/625 [==============================] - 18s 29ms/step - loss: 0.4405 - accuracy: 0.8421 - val_loss: 0.5338 - val_accuracy: 0.8227
Epoch 14/60
625/625 [==============================] - 18s 28ms/step - loss: 0.4256 - accuracy: 0.8481 - val_loss: 0.5252 - val_accuracy: 0.8220
Epoch 15/60
625/625 [==============================] - 18s 28ms/step - loss: 0.4023 - accuracy: 0.8571 - val_loss: 0.5292 - val_accuracy: 0.8233
Epoch 16/60
625/625 [==============================] - 18s 29ms/step - loss: 0.3840 - accuracy: 0.8651 - val_loss: 0.5007 - val_accuracy: 0.8318
Epoch 17/60
625/625 [==============================] - 18s 29ms/step - loss: 0.3680 - accuracy: 0.8700 - val_loss: 0.4972 - val_accuracy: 0.8370
Epoch 18/60
625/625 [==============================] - 18s 29ms/step - loss: 0.3483 - accuracy: 0.8761 - val_loss: 0.4984 - val_accuracy: 0.8367
Epoch 19/60
625/625 [==============================] - 18s 29ms/step - loss: 0.3429 - accuracy: 0.8777 - val_loss: 0.5252 - val_accuracy: 0.8297
Epoch 20/60
625/625 [==============================] - 18s 29ms/step - loss: 0.3244 - accuracy: 0.8842 - val_loss: 0.4735 - val_accuracy: 0.8418
Epoch 21/60
625/625 [==============================] - 18s 29ms/step - loss: 0.3136 - accuracy: 0.8869 - val_loss: 0.4937 - val_accuracy: 0.8420
Epoch 22/60
625/625 [==============================] - 18s 28ms/step - loss: 0.3041 - accuracy: 0.8907 - val_loss: 0.5083 - val_accuracy: 0.8367
Epoch 23/60
625/625 [==============================] - 18s 29ms/step - loss: 0.2964 - accuracy: 0.8925 - val_loss: 0.4987 - val_accuracy: 0.8423
Epoch 24/60
625/625 [==============================] - 18s 29ms/step - loss: 0.2860 - accuracy: 0.8971 - val_loss: 0.5099 - val_accuracy: 0.8392
```

```
Epoch 25/60
624/625 [============================>.] - ETA: 0s - loss: 0.2698 - accuracy: 0.9033Restoring model weights from the end of the best epoch: 20.
625/625 [==============================] - 18s 29ms/step - loss: 0.2700 - accuracy: 0.9032 - val_loss: 0.4854 - val_accuracy: 0.8459
Epoch 25: early stopping
/usr/local/lib/python3.7/dist-packages/sklearn/metrics/_classification.py:1318: UndefinedMetricWarning: Precision is ill-defined and being set tc
  _warn_prf(average, modifier, msg_start, len(result))
> 84.178
```

As you can see Epoch 25 is the early stopping epoch to prevent overfitting and you can see that we get a same accuracy with preventing the overfit.

**The train and validation error have been attached below:**

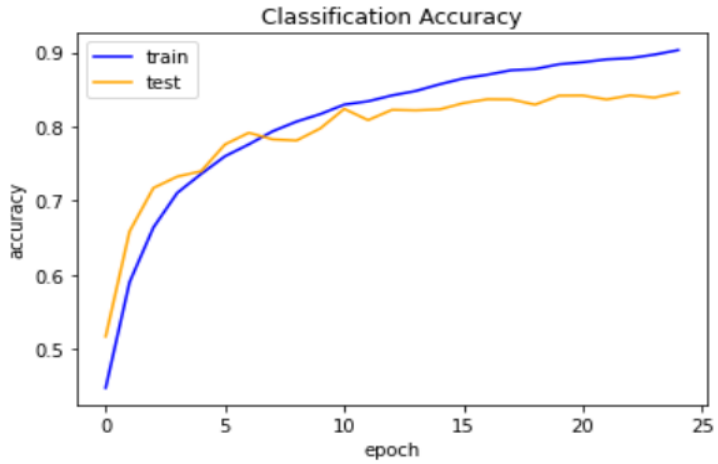



**Figure11.** Classification Accuracy

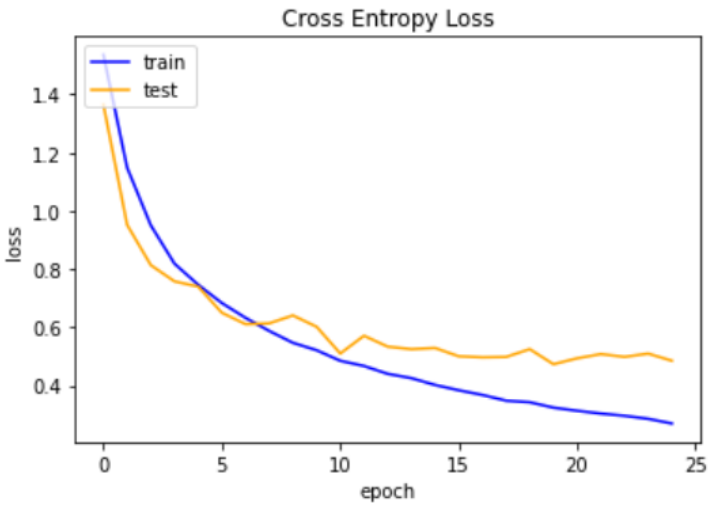



**Figure12.** Cross Entropy Loss

# 3 QUESTION #2: TRANSFER LEARNING

## 3.1 VGG19

### Architecture:

Visual Geometric Group or VGG is a CNN architecture that was introduced 2 years after AlexNet in 2014. The main reason for introducing this model was to see the effect of depth on accuracy while training models for image classification/recognition.

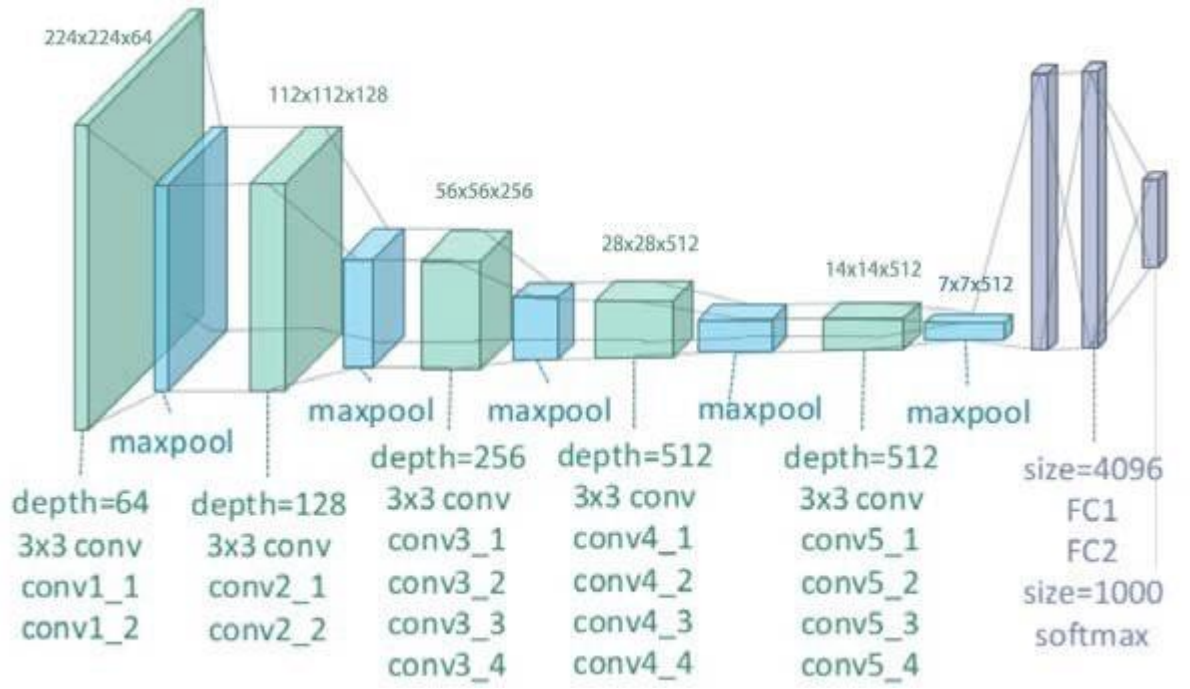



**Figure13.** VGG-19 Architecture

2Conv — 1Maxpool — 2Conv — 1Maxpool — 4Conv — 1Maxpool — 4Conv — 1Maxpool — 4Conv — 1Maxpool — 1FC — 1FC — 1FC

VGG-19 architectures, due to its depth is slow to train and produce models of very large size. Though the architectures we see here are different, we can create a simple template to perform transfer learning from these models with few lines of code.

### Advantages:

- VGG19 brought with it a massive improvement in accuracy and an improvement in speed as well. This was primarily because of improving the depth of the model and also introducing pretrained models.
- The increase in the number of layers with smaller kernels saw an increase in non-linearity which is always a positive in deep learning.

- VGG19 brought with it various architectures built on the similar concept. This gives more options to us as to which architecture could best fit our application.

### Disadvantages:

- One major disadvantage that they found was that this model experiences the vanishing gradient problem. This wasn't the case with any of the other models. The vanishing gradient problem was solved with the ResNet architecture.
- VGG19 is slower than the newer ResNet architecture that introduced the concept of residual learning which was another major breakthrough.

### Preprocessing:

For VGG19, we need to convert the input images from RGB to BGR, then zero-center each color channel with respect to the ImageNet dataset without scaling as preprocessing.

## 3.2 TRANSFER LEARNING

Transfer Learning is a machine learning method where we reuse a pre-trained model as the starting point for a model on a new task. To put it simply—a model trained on one task is repurposed on a second, related task as an optimization that allows rapid progress when modeling the second task.

The key idea here is to leverage the pre-trained model's weighted layers to extract features, but not update the model's weights during training with new data for the new task.

The pre-trained models are trained on a large and general enough dataset and will effectively serve as a generic model of the visual world.

## 3.3 VGG19 PRE-TRAINED BY IMAGENET

We used VGG19 Architecture pre-trained by ImageNet dataset which has more than 14 million images. ImageNet contains more than 20,000 categories with a typical category, such as "balloon", "strawberry" and "animals", consisting of several hundred images.

We pre-processed a picture of a bald eagle shown in Figure14 And the fed it to the pre-trained VGG19 model to see the results.

**Figure14.** Photo of bald eagle used for testing the network

```
bald_eagle (99.81%)
kite (0.16%)
vulture (0.02%)
```

**Figure15.** Top 3 classes with highest probabilities

## 3.4 TRANSFER LEARNING AND VGG19

To solidify these concepts, let's walk you through a concrete end-to-end transfer learning & fine-tuning example. We will load the VGG19 model, pre-trained on ImageNet, and use it on the Kaggle "cats vs. dogs" classification dataset.

Some random images of the dataset are shown in figure16


**Figure16.** Nine random pictures of the dataset

As for preprocessing we reshaped the images to 224*224 and converted them from RGB to BGR, then zero-centered each color channel with respect to the ImageNet dataset without scaling.

For transfer learning we froze the weights of all convolutional layers of the model and replaced the last 3 fully connected layers with one fully connected layer with 1 neuron (since we have 2 classes).

Then we trained the network on the dataset.

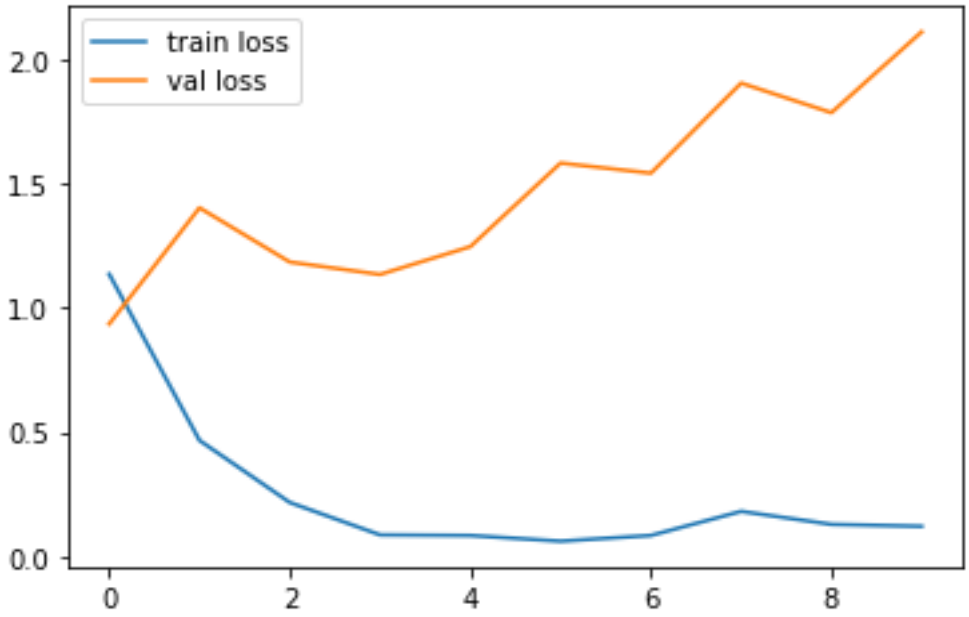



**Figure17.** Training and validation loss for each epoch

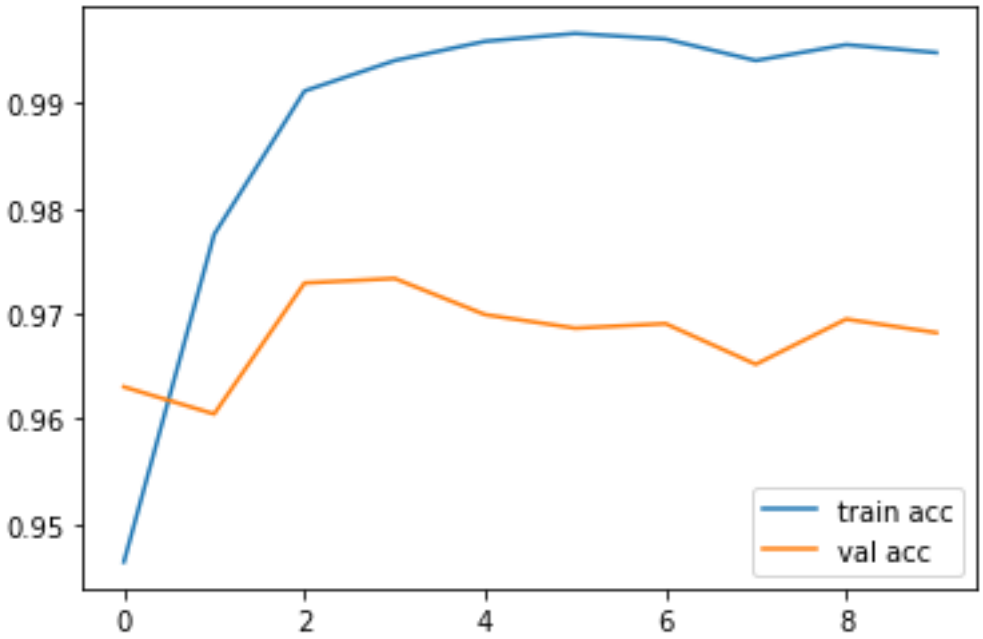



**Figure18.** Training and validation accuracy for each epoch

By looking at the accuracy and loss plots we can see that 3 to 4 epoch is enough for training our model and more epochs will cause overfitting.

```
Accuracy on test data is: 0.9729148753224419
```

**Figure19.** Accuracy on test data

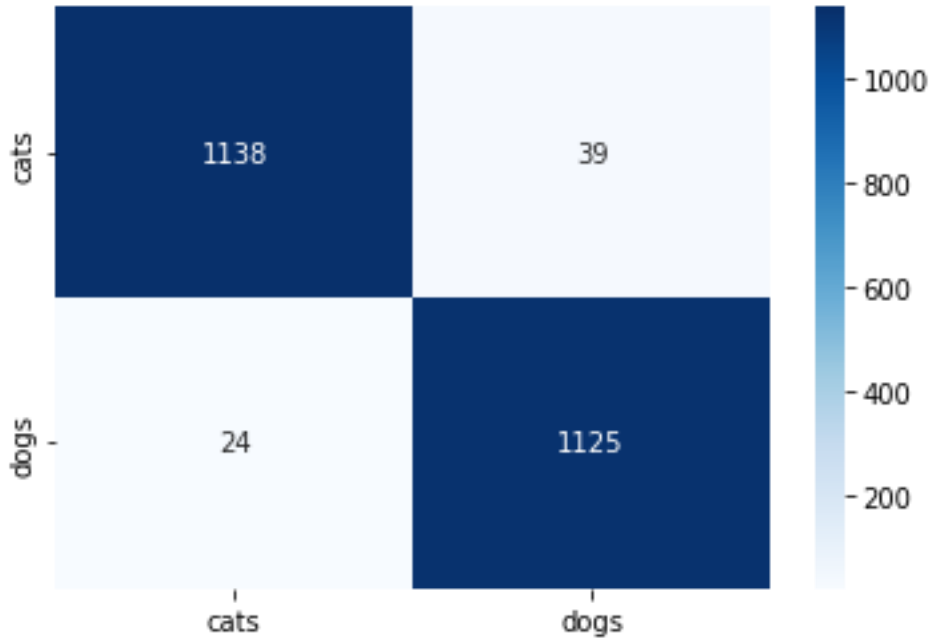



**Figure20.** Confusion matrix for test data

# 4 QUESTION #3: SEGMENTATION

In this part we intend to implement one of the most common application of neural network which is image segmentation.

Image segmentation is a method in which a digital image is broken down into various subgroups called Image segments which helps in reducing the complexity of the image to make further processing or analysis of the image simpler. Segmentation in easy words is assigning labels to pixels. All picture elements or pixels belonging to the same category have a common label assigned to them. For example: Let's take a problem where the picture has to be provided as input for object detection. Rather than processing the whole image, the detector can be inputted with a region selected by a segmentation algorithm. This will prevent the detector from processing the whole image thereby reducing inference time.

Further, we are going to analyse two of the most important models in case of image segmentation.

## 4.1 DEEPLAB

The first model we have implemented called DEEPLAB which is a state-of-the-art semantic segmentation model having encoder-decoder architecture. The encoder consisting of pretrained CNN model is used to get encoded feature maps of the input image, and the decoder reconstructs output, from the essential information extracted by encoder, using up sampling.

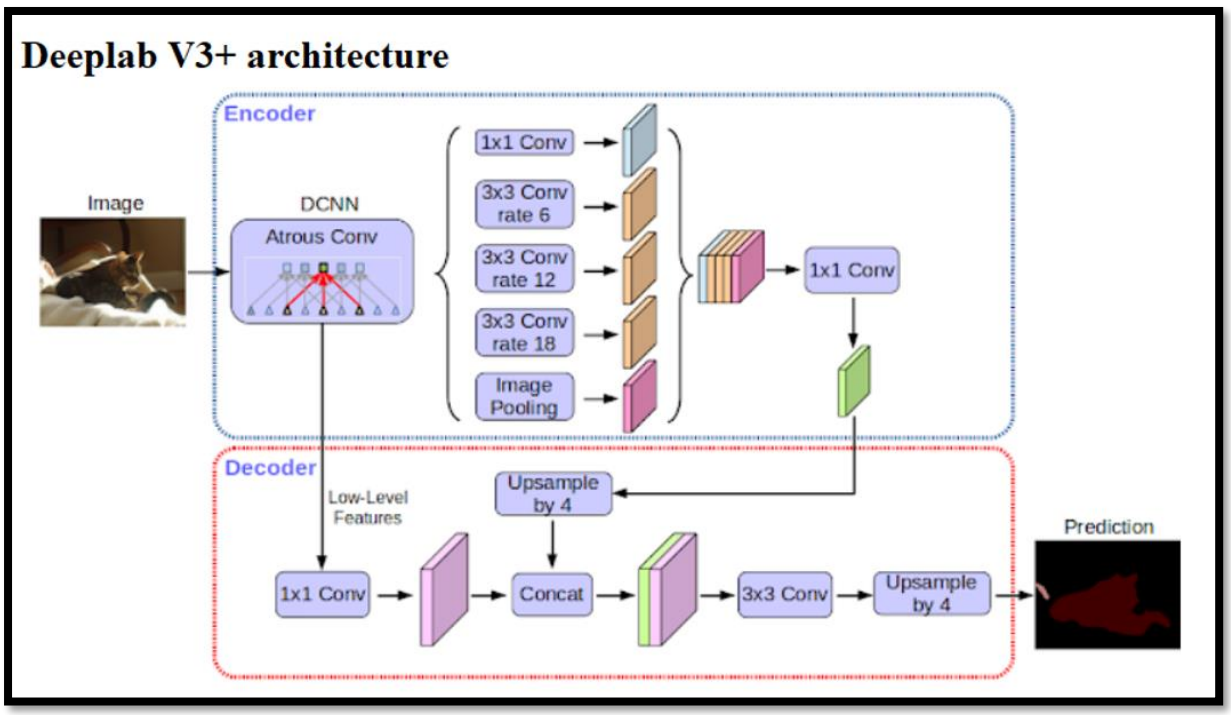



**Figure21.** Architecture *of Deeplab V3*

## 4.2 FCN

The second model we have implemented called FCN.

One of the ways to do semantic segmentation is to use a Fully Convolutional Network (FCN) i.e., we stack a bunch of convolutional layers in an encoder-decoder fashion. The encoder down samples the image using strided convolution giving a compressed feature representation of the image, and the decoder up-samples the image using methods like transpose convolution to give the segmented output.

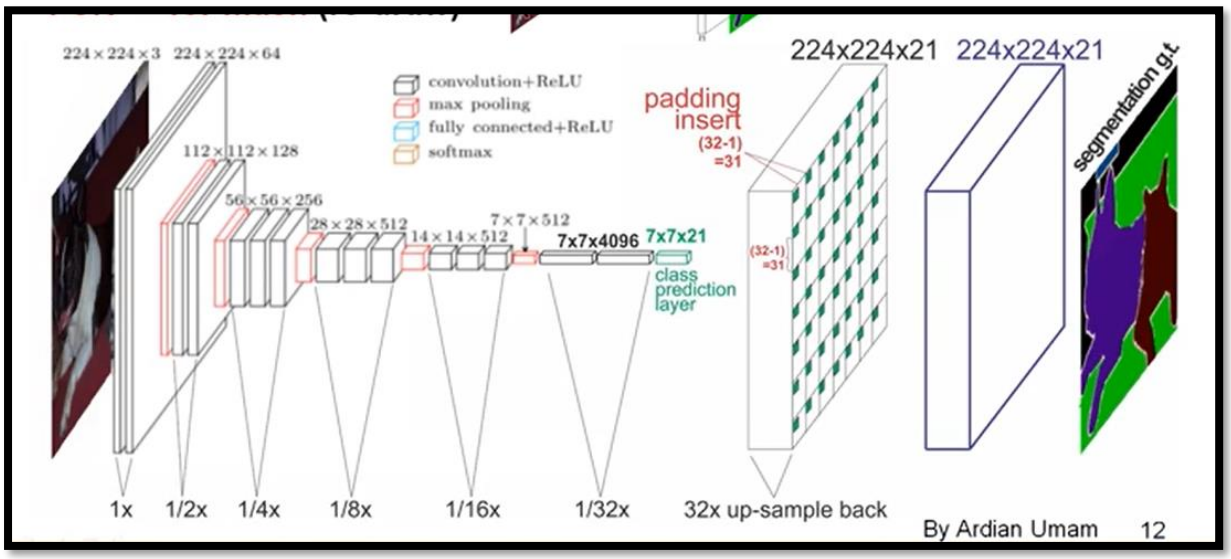



**Figure22.** Architecture *of FCN*

## 4.3 QUICK REVIEW ON PERFORMANCE & ARCHITECTURE OF BOTH MODELS

**DeepLab V3 (2017):**

First introduce the following four methods to solve multi-scale problems:

- Image pyramid: Scale the image to different scales, share models and parameters, input different sizes, and then merge and output;
- Encoder-decoder: Encoding and decoding network, the decoding part gradually merges the encoded information;
- Context module: add additional modules at the end, such as CRF
- Spatial pyramid pooling: Add spatial pyramid pooling to the last layer

Now we express some of important notes about the architecture:

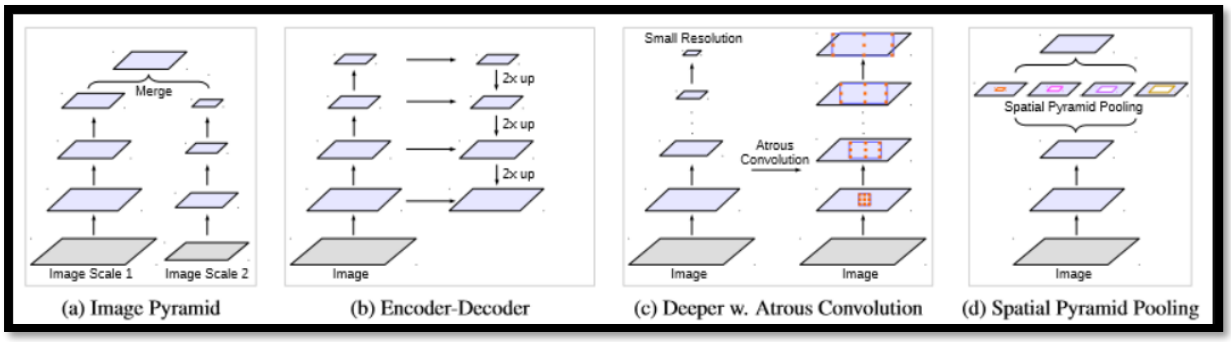



**Figure23.** rate value representation

As the rate value increases, the effective weight decreases. When the rate value is large, the hole convolution is equivalent to a 1x1 convolution, because only the middle weight of the convolution kernel is valid, so that global information cannot be obtained. Therefore, certain modifications have been made on the basis of V2: (a) ASPP includes 1x1 convolution and three hole convolutions with rates of (6, 12, 18); (b) using global pooling on the last feature , And connect a 256-channel 1x1 convolution and BN, and then bilinearly up-sampling. After concat, pass a 256-channel 1x1 convolution and BN layer.

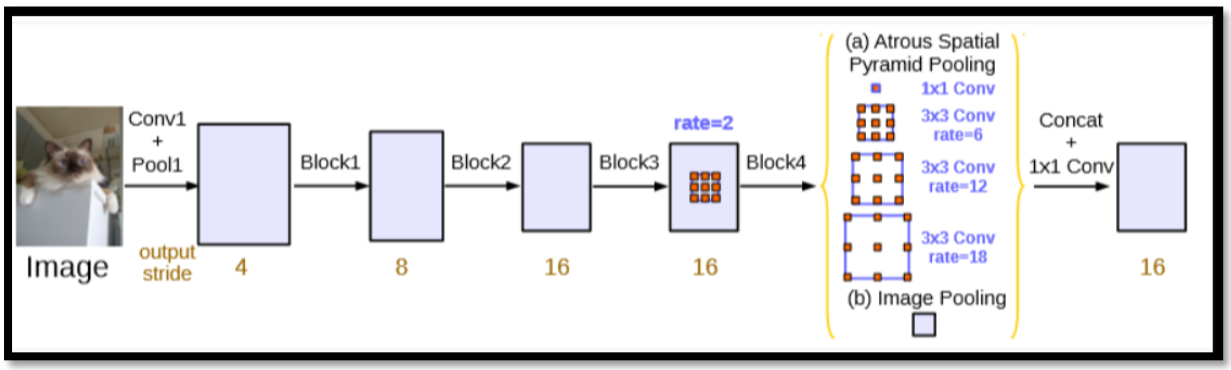



**Figure24.** Atrous Spatial Pyramid Pooling Module

**Brief summary:**

- Using ResNet's 4 blocks to extract features
- The brown numbers at the bottom of the figure represent the ratio of the original input image spatial resolution to the output resolution

- The original text says that if output_stride=8, the rate should be doubled, which means that if the ASPP structure is used one block in advance, the rate value should be changed accordingly (Original: Note that the rates are doubled when output stride = 8.)

**FCN (2014):**

FCN is the basis of semantic segmentation, and many subsequent segmentation methods are developed on the basis of FCN.

**Brief summary:**

FCN turns the fully connected layer of some classification networks (such as VGG) into a fully convolutional layer, obtains a low-resolution feature map and then uses deconvolution to up sample to the original image size

Pooling can obtain advanced features and also cause resolution reduction and loss of spatial information, so FCN combines features with different roughness (mentioned in the notes) to refine the segmentation results

## 4.4 RESULTS OF IMPLEMENTATION

We have used three separated images as of our test-input, one of them from the one-class category, another one from two-classes category and the final from the multi-class category.

First of all, we have used **transform command** and also **preprocess command** to do some the important preprocessing steps such as **suitable-scaling** for the input image, **normalizing** and **resizing the input images**.

Afterwards, we call the both models to import the correspond weights for doing appropriately segmentation task and at the final we have passed the input images to the models and get the results **which are abbreviated below for both Deeplab and FCN approaches:**

**DeepLab V3:**

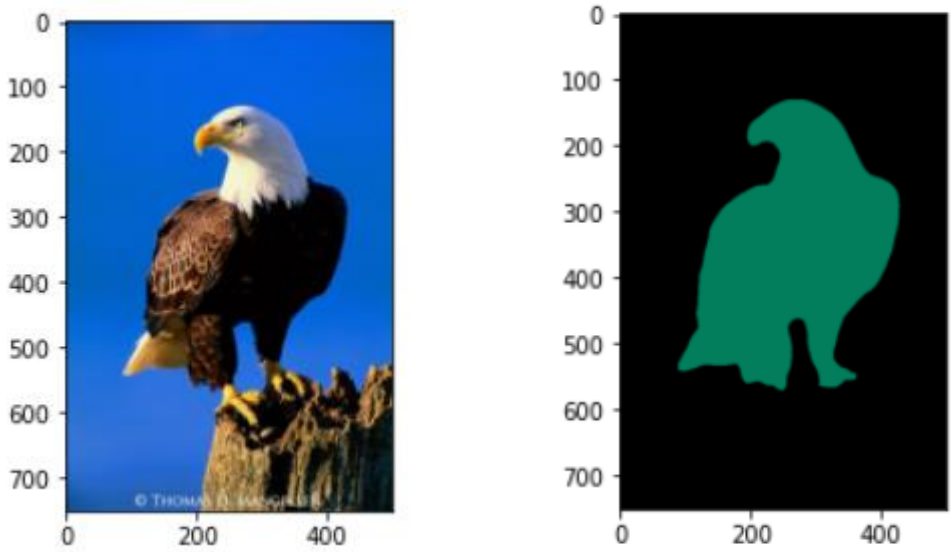



**Figure25.** One-Class Segmentation

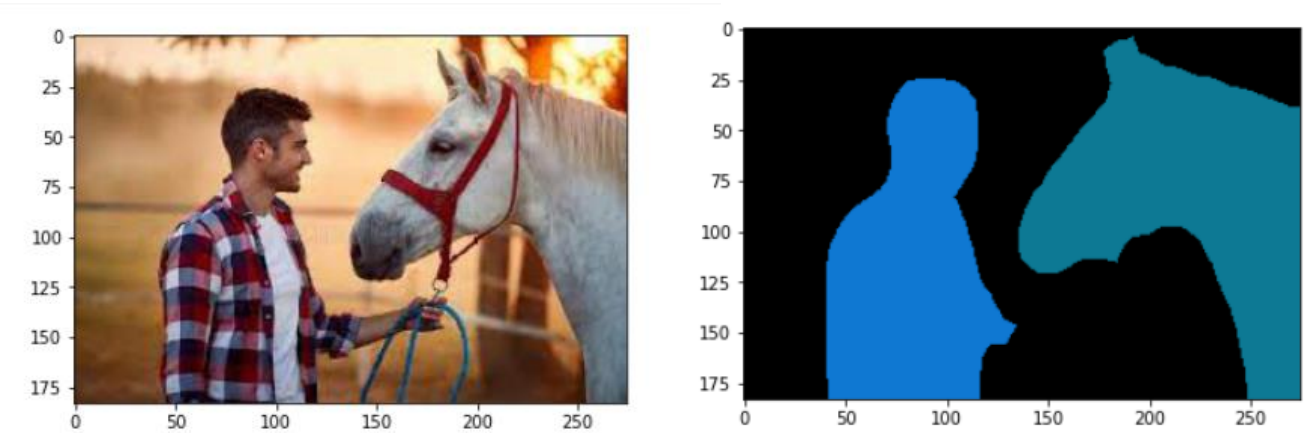



**Figure 26.** Two-Classes Segmentation

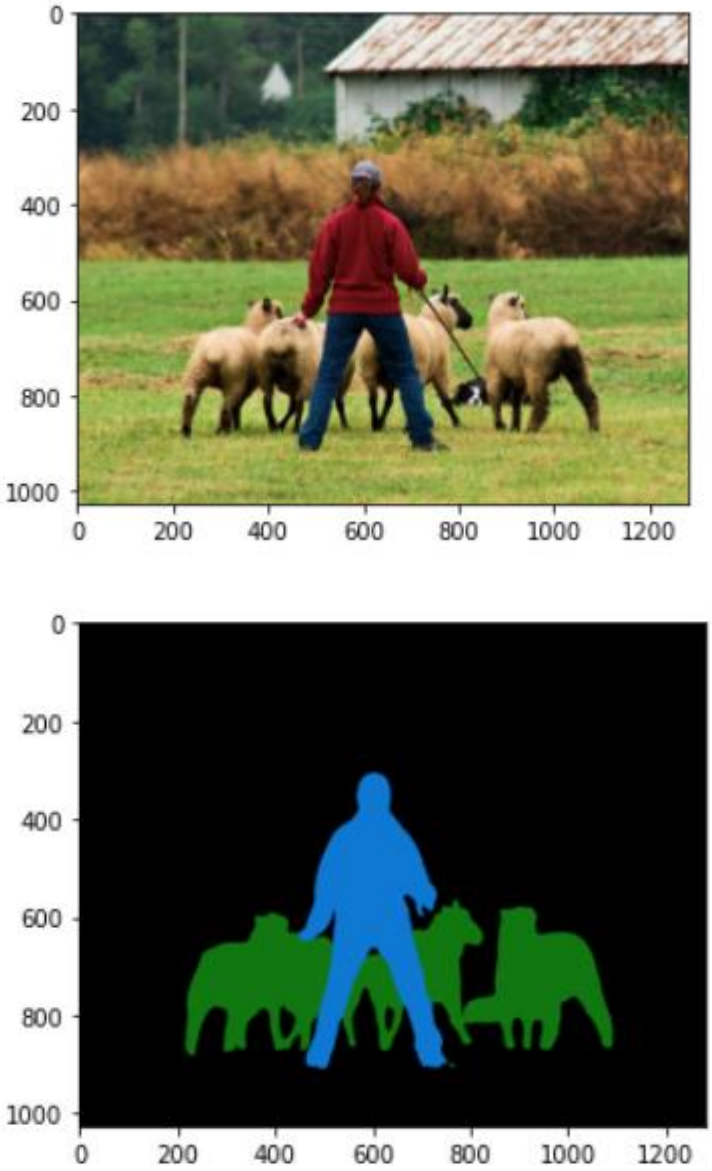



**Figure27.** Multi-Classes Segmentation

**FCN:**

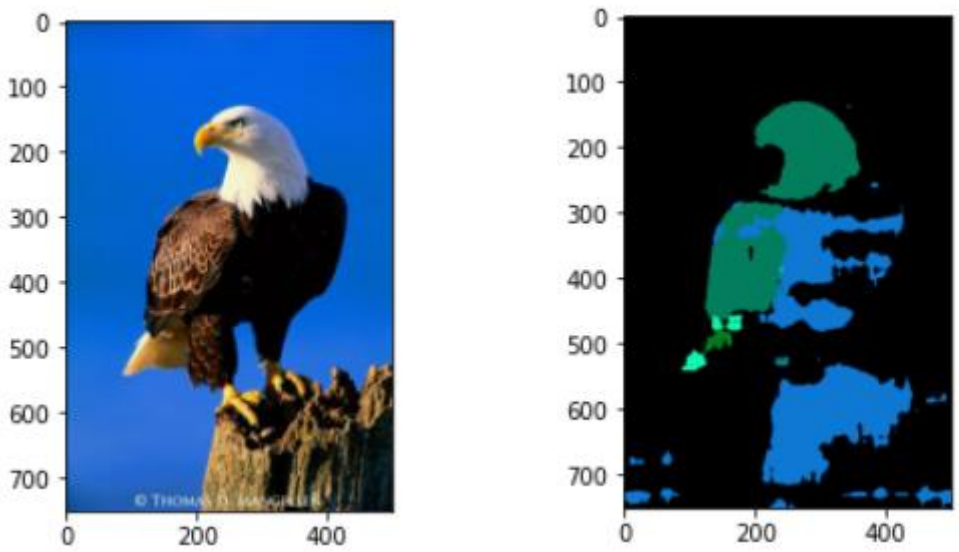



**Figure28.** One-Class Segmentation

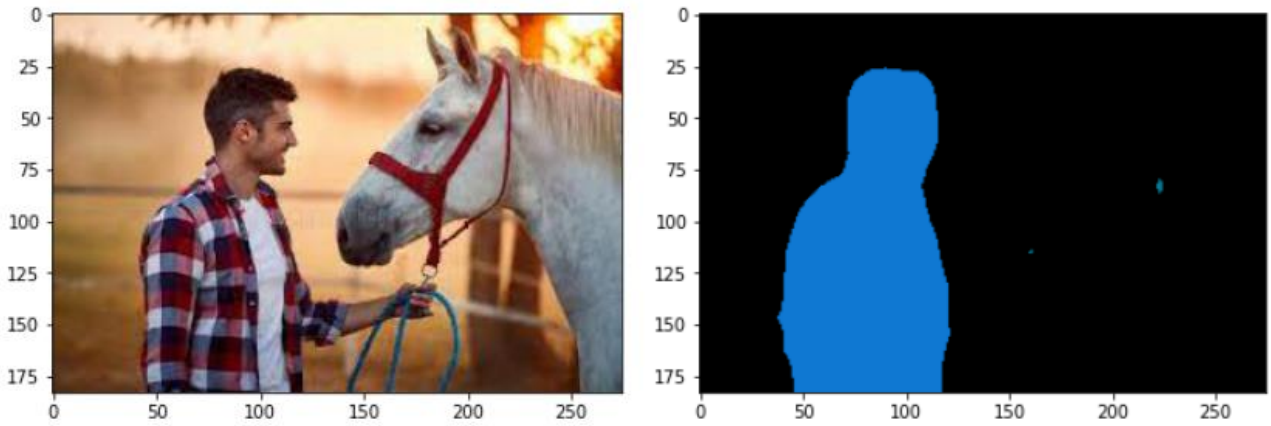



**Figure29:** Two-Classes Segmentation

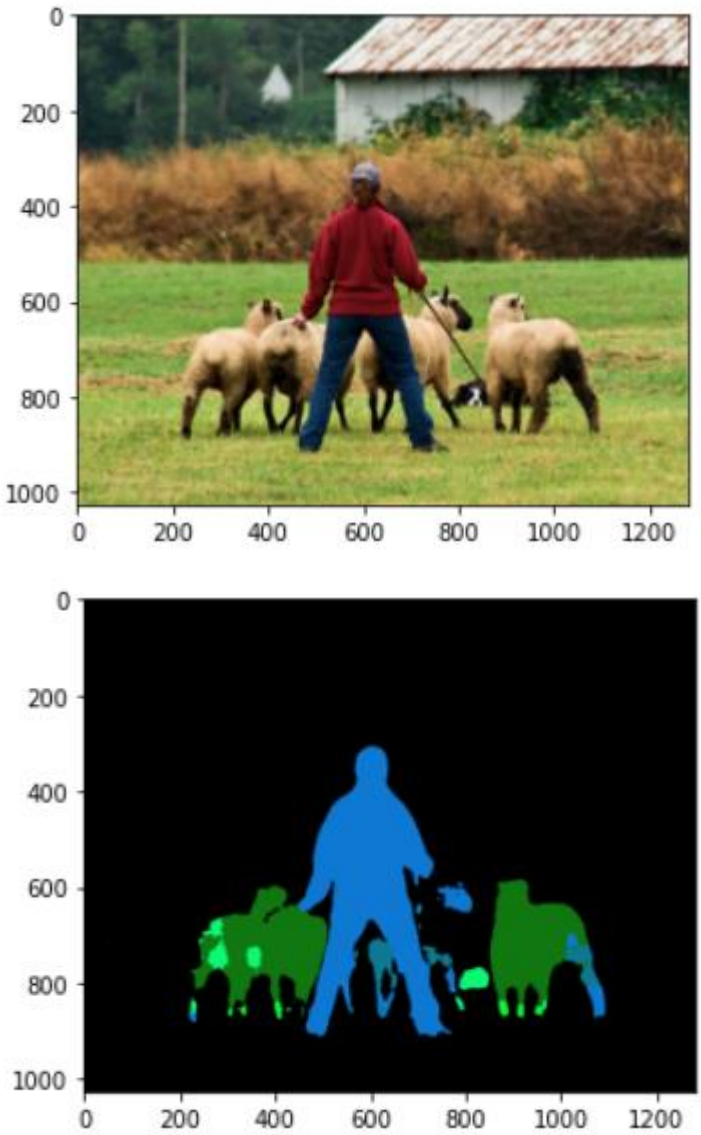



**Figure30:** Multi-Classes Segmentation

**Analysis:**

As we expected, **the DeepLab V3** acts **pretty much better** in comparison to the **FCN models.**

Because according to the model's architecture, DeepLab uses **more down-sampling module** which helps the model to extract more detailed features from the input image and then It leads to have a better classification accuracy and consequently have a **better power to recognize** every separated classes.

One more thing to mention here is about encoding image information module in **DeepLab V3 architecture** which let us to get a **better rate** in **task of encoding information** such as **edges** and **floors** detection which leads to have a better strength for **classifying different classes.**

# 5 QUESTION #4: OBJECT DETECTION

## 5.1 YOLOv2 Vs YOLOv1

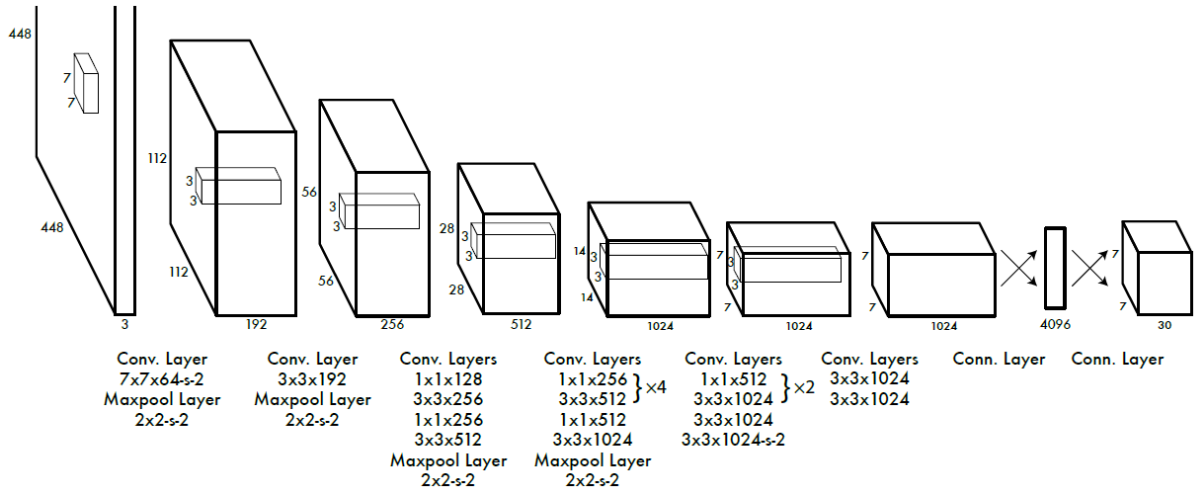



**Figure31.** YOLOv1 architecture

The changes from YOLOv2 to YOLO v1:

- **Batch Normalization:** it normalizes the input layer by altering slightly and scaling the activations. Batch normalization decreases the shift in unit value in the hidden layer and by doing so it improves the stability of the neural network.
- **Higher Resolution Classifier**: the input size in YOLO v2 has been increased from 224*224 to 448*448.
- **Anchor Boxes**: one of the most notable changes which can visible in YOLO v2 is the introduction the anchor boxes. YOLO v2 does classification and prediction in a single framework. These anchor boxes are responsible for predicting bounding box and this anchor boxes are designed for a given dataset by using clustering (k-means clustering)
- **Fine-Grained Features**: one of the main issued that has to be addressed in the YOLO v1 is that detection of smaller objects on the image. This has been resolved in the YOLO v2 divides the image into 13*13 grid cells which is smaller when compared to its previous version. This enables the yolo v2 to identify or localize the smaller objects in the image and also effective with the larger objects.
- **Multi-Scale Training**: on YOLO v1 has a weakness detecting objects with different input sizes which says that if YOLO is trained with small images of a particular object. it has issues detecting the same object on image of bigger size.
- **Darknet 19**: YOLO v2 uses Darknet 19 architecture with 19 convolutional layers and 5 max pooling layers and a softmax layer for classification objects. The architecture of the Darknet 19 has been shown below. Darknet is a neural network framework written in Clanguage and CUDA. It's really fast in object detection which is very important for predicting in real-time.

**Table1.** Darknet 19 architecture

| Type | Filters | Size/Stride | Output |
|---|---|---|---|
| Convolutional | 32 | 3 × 3 | 224 × 224 |
| Maxpool | | 2 × 2/2 | 112 × 112 |
| Convolutional | 64 | 3 × 3 | 112 × 112 |
| Maxpool | | 2 × 2/2 | 56 × 56 |
| Convolutional | 128 | 3 × 3 | 56 × 56 |
| Convolutional | 64 | 1 × 1 | 56 × 56 |
| Convolutional | 128 | 3 × 3 | 56 × 56 |
| Maxpool | | 2 × 2/2 | 28 × 28 |
| Convolutional | 256 | 3 × 3 | 28 × 28 |
| Convolutional | 128 | 1 × 1 | 28 × 28 |
| Convolutional | 256 | 3 × 3 | 28 × 28 |
| Maxpool | | 2 × 2/2 | 14 × 14 |
| Convolutional | 512 | 3 × 3 | 14 × 14 |
| Convolutional | 256 | 1 × 1 | 14 × 14 |
| Convolutional | 512 | 3 × 3 | 14 × 14 |
| Convolutional | 256 | 1 × 1 | 14 × 14 |
| Convolutional | 512 | 3 × 3 | 14 × 14 |
| Maxpool | | 2 × 2/2 | 7 × 7 |
| Convolutional | 1024 | 3 × 3 | 7 × 7 |
| Convolutional | 512 | 1 × 1 | 7 × 7 |
| Convolutional | 1024 | 3 × 3 | 7 × 7 |
| Convolutional | 512 | 1 × 1 | 7 × 7 |
| Convolutional | 1024 | 3 × 3 | 7 × 7 |
| Convolutional | 1000 | 1 × 1 | 7 × 7 |
| Avgpool | | Global | 1000 |
| Softmax | | | |

YOLOv2 has seen a great improvement in detecting smaller objects with much more accuracy which it lacked in its predecessor version.

## 5.2 YOLOv5 AND YOLOv4 ADVANCES

YOLO4. uses Cross Stage Partial Network (CSPNet) in Darknet, creating a new feature extractor backbone called CSPDarknet53. The convolution architecture is based on modified DenseNet. It transfers a copy of feature map from the base layer to the next layer through dense block. The advantages of using DenseNet include the diminishing gradient vanishing problems, boosting backpropagation, removal of the computational bottleneck, and improved learning.

However, YOLOv5 is different from the previous releases. It utilizes PyTorch instead of Darknet. It utilizes CSPDarknet53 as backbone. This backbone solves the repetitive gradient information in large backbones and integrates gradient change into feature map that reduces the inference speed, increases accuracy, and reduces the model size by decreasing the parameters.

**Table2.** Comparison between different versions of YOLO

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

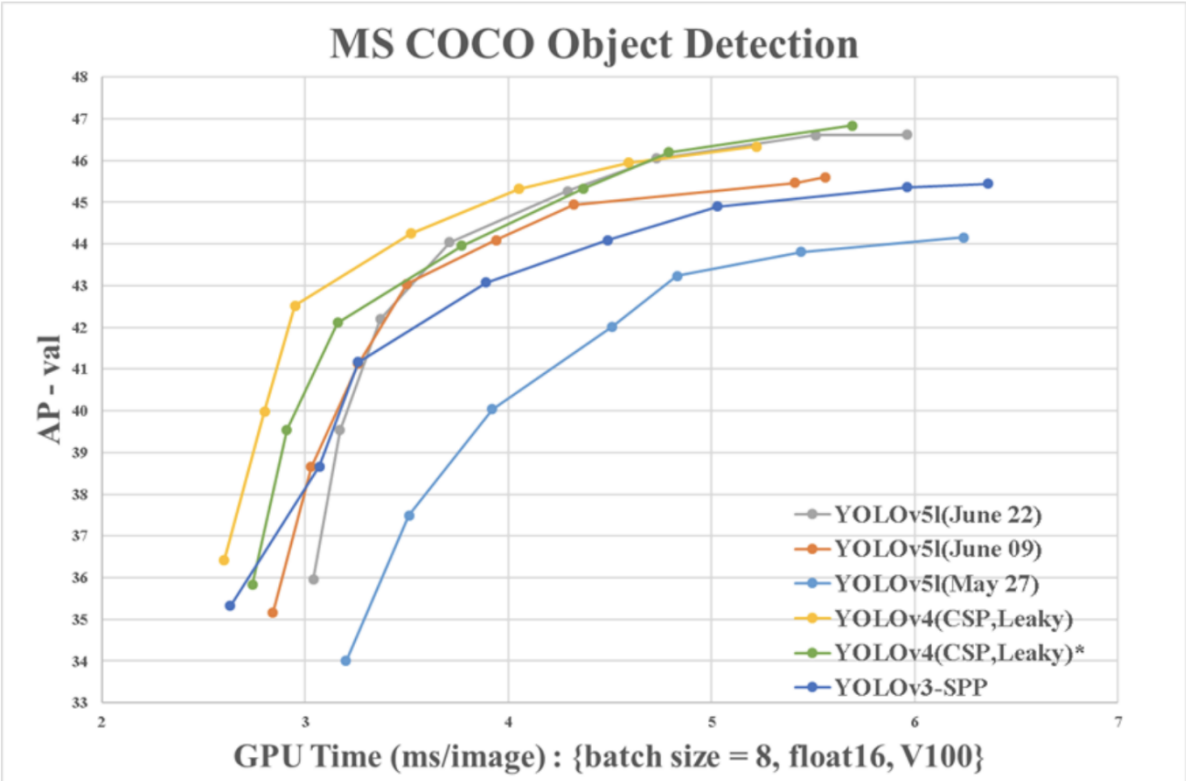
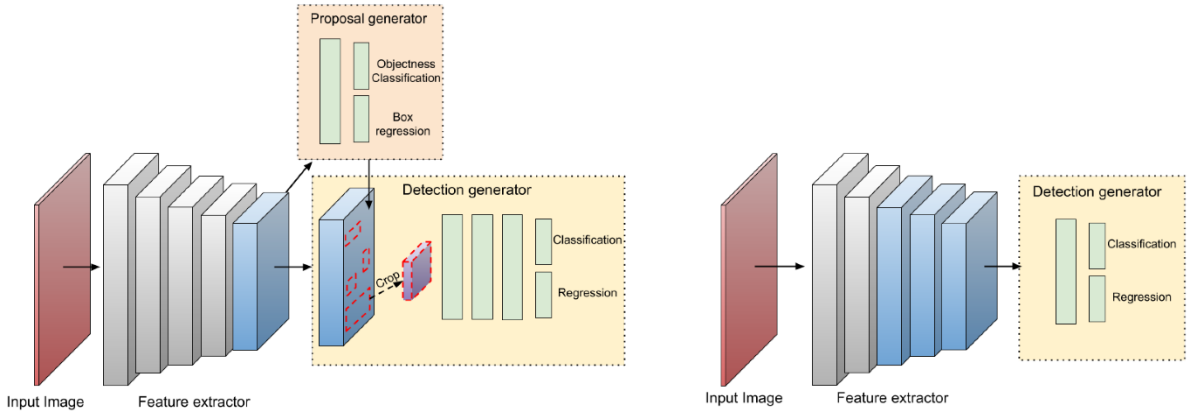



**Figure32.** Average precision against GPU Time for different versions of YOLO

## 5.3 ONE STAGE VS TWO STAGE OBJECT DETECTION

One-stage detectors have high inference speeds and two-stage detectors have high localization and recognition accuracy. The two stages of a two-stage detector can be divided by a RoI (Region of Interest) Pooling layer. One of the prominent two-stage object detectors is Faster R-CNN. It has the first stage called RPN, a Region Proposal Network to predict candidate bounding boxes. In the second stage, features are by RoI pooling operation from each candidate box for the following classification and bounding-box regression tasks. In contrast, a one-stage detector predicts bounding boxes in a single-step without using region proposals. It leverages the help of a grid box and anchors to localize the region of detection in the image and constraint the shape of the object.




(a) Two-stage Faster R-CNN (b) One-stage RetinaNet

**Figure33.** One stage vs two stage object detection

Some of one-stage object detection methods are: YOLO, EfficientDet, RefineDet

And for two-stage object detection methods we can name: Fast R-CNN, PANet

## 5.4 YOLOV5 FOR BOCCE BALLS

We trained our YOLOv5 model for the Booceball dataset which has 6 objects to detect; Balls with 5 different colors and vlines.

Training time: 1hour 28minutes on Tesla K80 GPU and 12GB of RAM, 100 epochs

First, we upload the dataset with a .yaml file on the RoboFlow website and then use the roboflow library in python with the API obtained from the website to load the data in the Colab environment. Then we git clone the YOLOv5 repository in Github and build our model (anchors, backbone, head). Then we train our model on the dataset for 100 epochs using 16 batch sizes. After training, we will show precision and accuracy graphs for validation and training data. Then we will show an augmented (the augmentation procedure is explained in the "README.roboflow.txt") image from the training dataset and a ground truth image with labels from the validation dataset along with the image with the predicted labels by the model. Then we

will run the "detect.py" from the YOLOv5 repository to perform object detection on the test dataset and show the results.

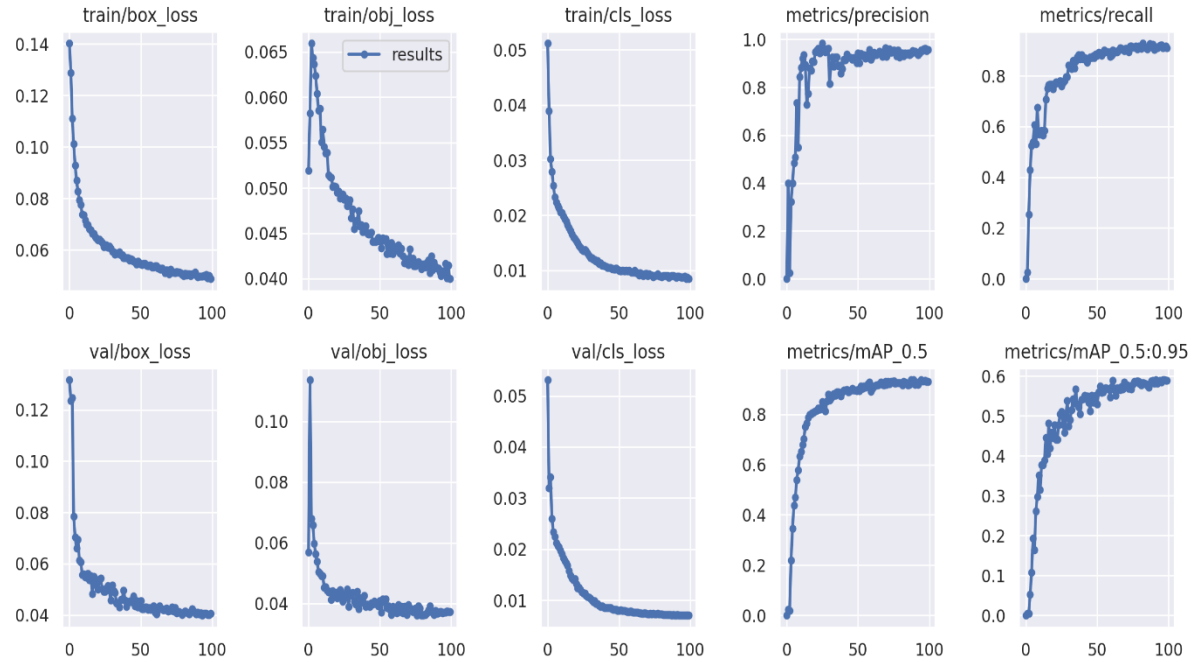



**Figure34.** Training and validation metrics against number of epochs

**Figure35.** Some of augmented images used for training with labels

**Figure36.** Some validation images with their ground truth labels

**Figure37.** Some validation images with predicted labels

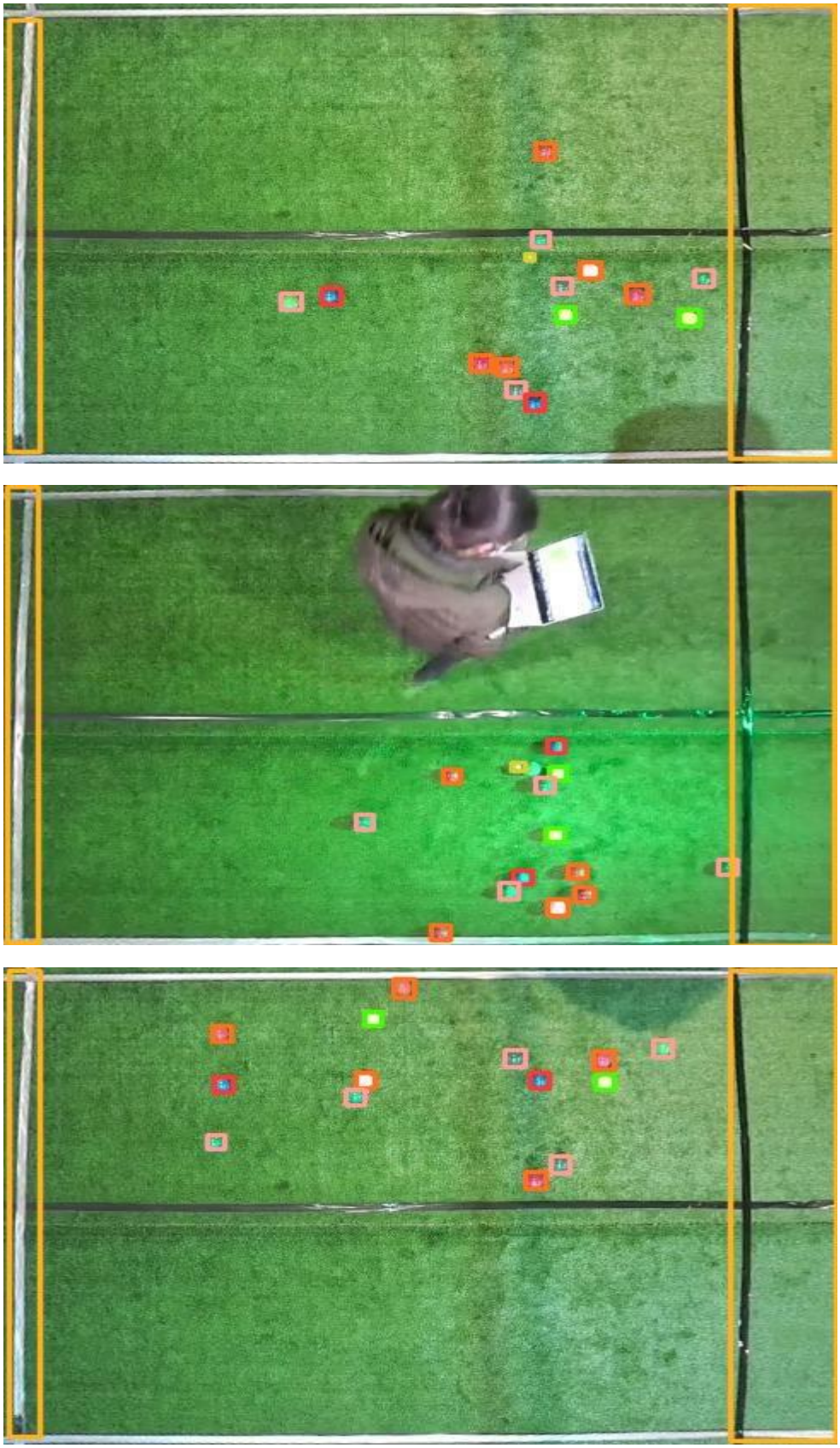


**Figure38.** Some test images with predicted labels

# 6 REFERENCES

[1] https://koushik1102.medium.com/transfer-learning-with-vgg16-and-vgg19-the-simpler-way-ad4eec1e2997

[2] https://keras.io/api/applications/vgg/

[3] https://tejasmohanayyar.medium.com/a-practical-experiment-for-comparing-lenet-alexnet-vgg-and-resnet-models-with-their-advantages-d932fb7c7d17

[4] Nepal U, Eslamiat H. Comparing YOLOv3, YOLOv4 and YOLOv5 for Autonomous Landing Spot Detection in Faulty UAVs. Sensors. 2022; 22(2):464. https://doi.org/10.3390/s22020464

[5] https://towardsdatascience.com/yolo-v4-or-yolo-v5-or-pp-yolo-dad8e40f7109

[6] https://blog.roboflow.com/yolov5-improvements-and-evaluation/

[7] https://machinelearningmastery.com/how-to-stop-training-deep-neural-networks-at-the-right-time-using-early-stopping/

[8] https://towardsdatascience.com/the-evolution-of-deeplab-for-semantic-segmentation-95082b025571

[9] https://www.cambridge.org/core/journals/apsipa-transactions-on-signal-and-information-processing/article/dsnet-an-efficient-cnn-for-road-scene-segmentation/7F5F21DD5DA0625F3BB96C9846550904

[10]https://www.researchgate.net/publication/326785340_Segmentation_model_based_on_convolutional_neural_networks_for_extracting_vegetation_from_Gaofen-2_images/figures?lo=1

[11] https://blog.actorsfit.com/a?ID=01600-ac03ed5c-eb66-45fd-bcbc-ce8c4dacb915

[12] https://towardsdatascience.com/image-segmentation-part-1-9f3db1ac1c50

[13] https://www.jeremyjordan.me/semantic-segmentation/

[14] https://en.wikipedia.org/wiki/ImageNet

[15] https://deeplearning.cms.waikato.ac.nz/user-guide/class-maps/IMAGENET/