Visual Learning & Recognition Assignment 1: Object Classification with TensorFlow!

Name: Bhavan Jasani
Andrew ID: bjasani

Note: For testing the codes, keep all python files and the “VOCdevkit” directory as well as “vgg_16.ckpt” in the same folder

Task 0

To run code type: python 00_mnist.py

Analysis: Pretty simple no issues and no hyper parameter tuning required

Q 0.1: What test accuracy does your model get?

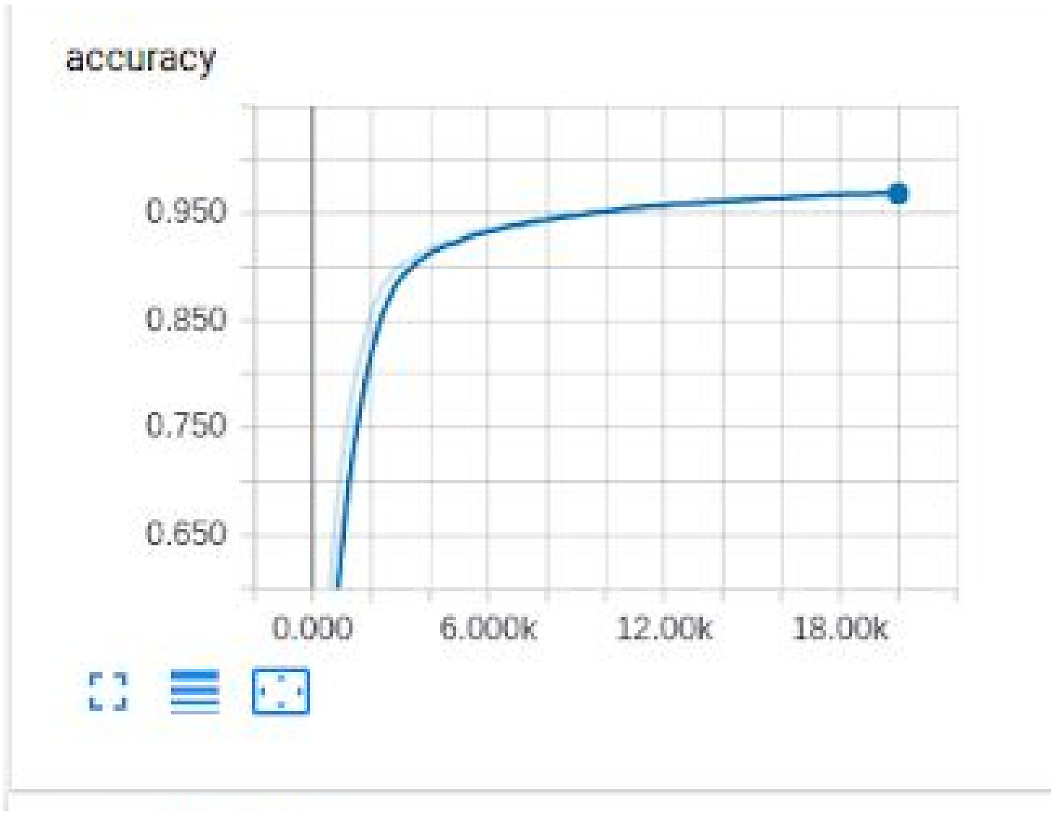
After 20000 iterations:
Training Loss: 0.098742664,
Test Accuracy: 0.968

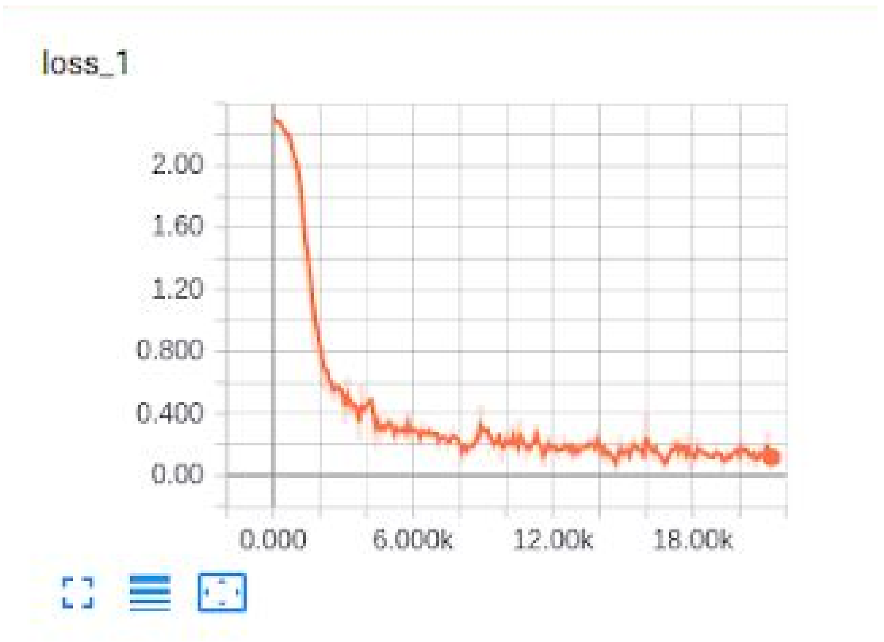
Q 0.2: What happens if you train for more iterations (30000)?

After 35000 iterations:
Training Loss: 0.06859733
Test Accuracy: 0.9782

The test accuracy still increases for training further beyond 30,000 iterations. Probably after running even more the test accuracy might go down because of overfitting.

Q 0.3: Make the training loss and validation accuracy curve.

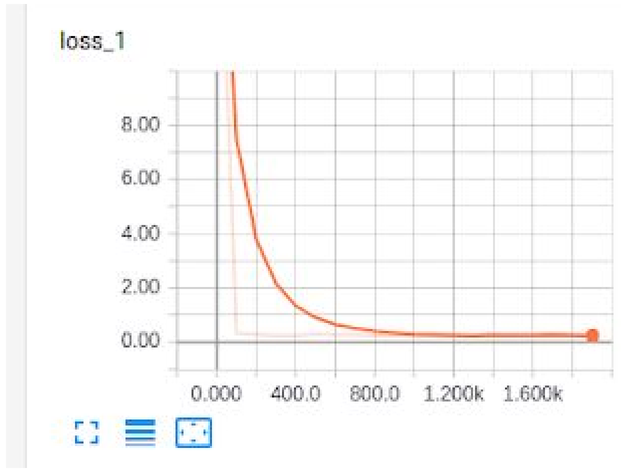


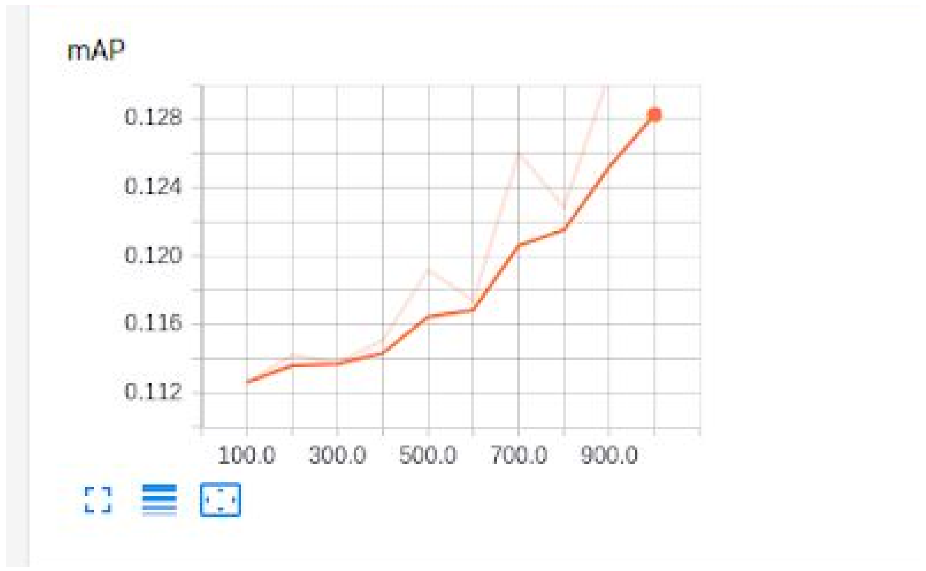


Task 1

To run code type: python 01_pascal.py VOCdevkit

Analysis: Pretty simple no issues and no hyper parameter tuning required



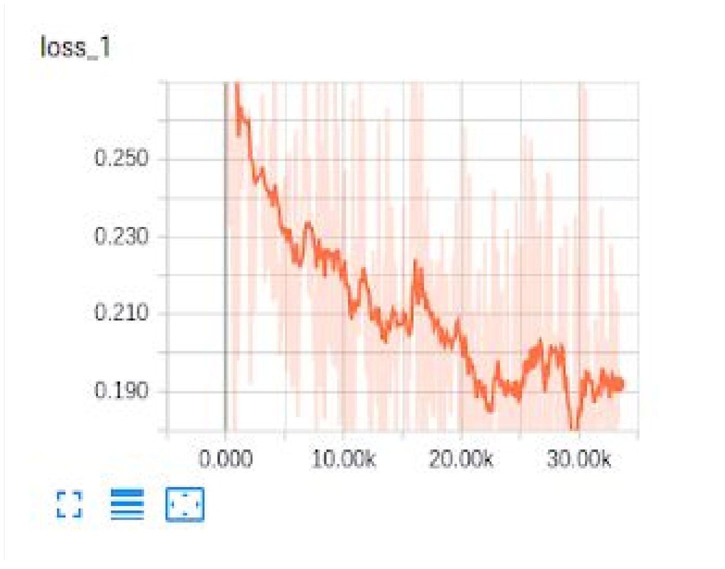


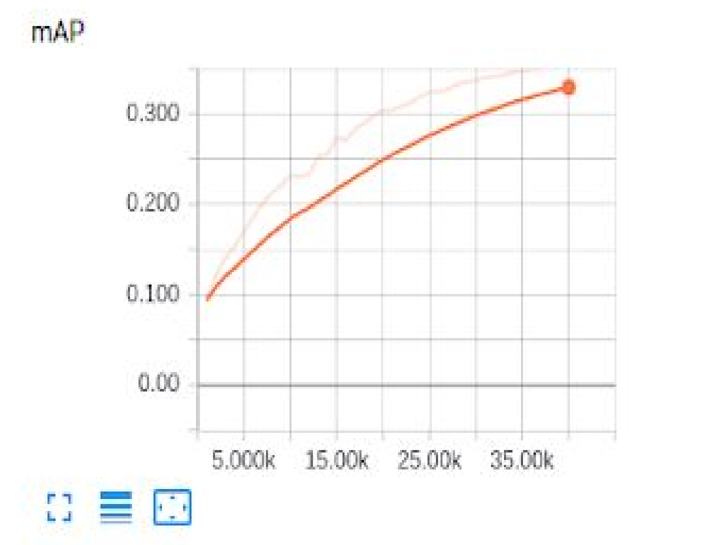
Loss for final step: 0.16898814
Obtained mAP: 0.132809426336

Task 2

To run code type: python 02_pascal_alexnet.py VOCdevkit

Analysis: Pretty simple no issues and no hyper parameter tuning required





Loss for final step: 0.19311388

Obtained mAP : 0.354166004843

Task 3

To run code type: python 03_pascal_vgg16.py VOCdevkit

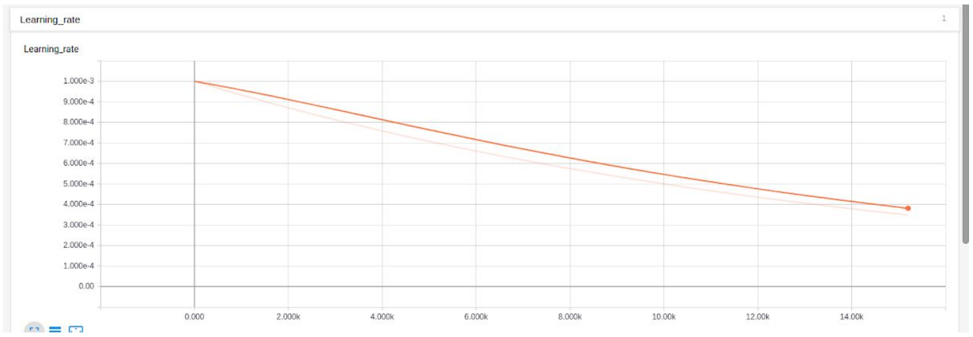
Analysis: This one required a lot of debugging, initially with the parameters mentioned in the question, my training loss was decreasing but so was mAP and after some debugging I was able to manage to make mAP increase but, then it started to saturate at 0.06 which is a pretty low value. And finally I found that the Gaussian weight initialization of conv2d layers was the root cause of this erratic behaviour. I tried removing the Gaussian initialization, so as to have the default tensorflow initialization of filters (it’s based on fan in - fan out, similar to Xavier initialization) and it worked like a charm, my mAP started to increase.

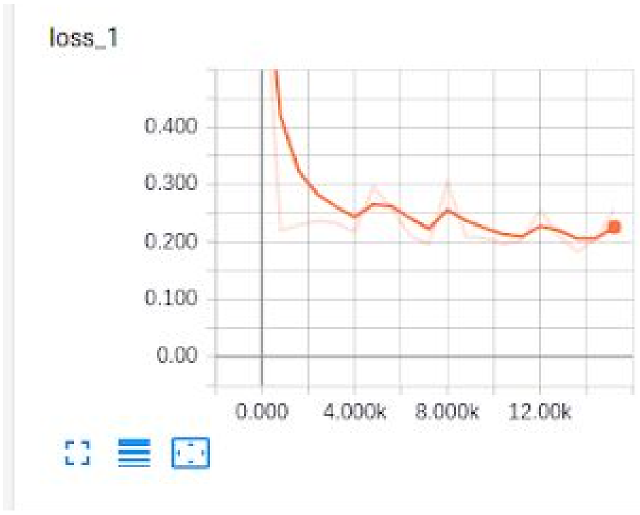
I had used a lot of AWS credits for this and some other questions and so I didn’t train for full 40k iterations rather only for 16k (around 2 hours) and so my mAP is not that great, coming to 0.275. Had I trained for more iterations and a bit more of hyper parameter tuning I would have had far better test accuracy.

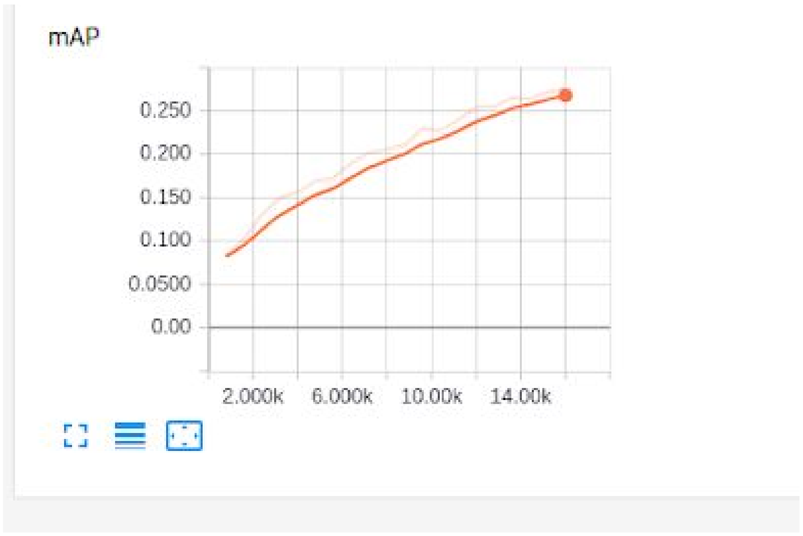
Also for task 4 I found using 1/10th learning rate (I had experimented this learning rate tuning with task 4 because task 4 requires lesser training time) of that mentioned in question got me about 35% improvement in test accuracy, and I would expect a similar thing here if I had kept a learning rate of 0.01 I would have gotten far better performance, but I couldn’t because of lack of AWS credits.

Hyper – parameters used:
Starting learning rate = 0.001
Batch size = 10
Momentum = 0.9
No. of iterations: 16,000
Weight initialization = Tensorflow default
Bias initialization = zero initialization

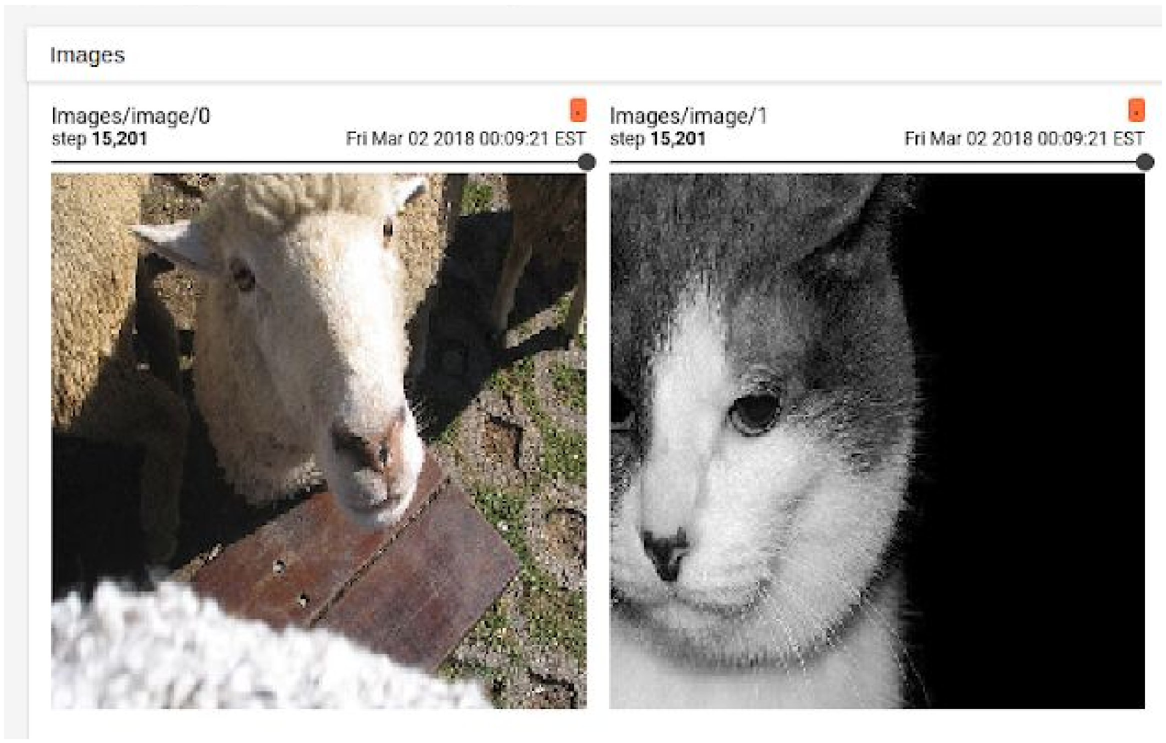
Loss for final step: 0.16858448
Obtained mAP: 0.275229643958


Learning_rate
Learning_rate
1.000e-3
9.000e-4
8.000e-4
7.000e-4
6.000e-4
5.000e-4
4.000e-4
3.000e-4
2.000e-4
1.000e-4
0.00
0.000
2.000k
4.000k
6.000k
8.000k
10.00k
12.00k
14.00k


loss_1
0.400
0.300
0.200
0.100
0.00
0.000
4.000k
8.000k
12.00k

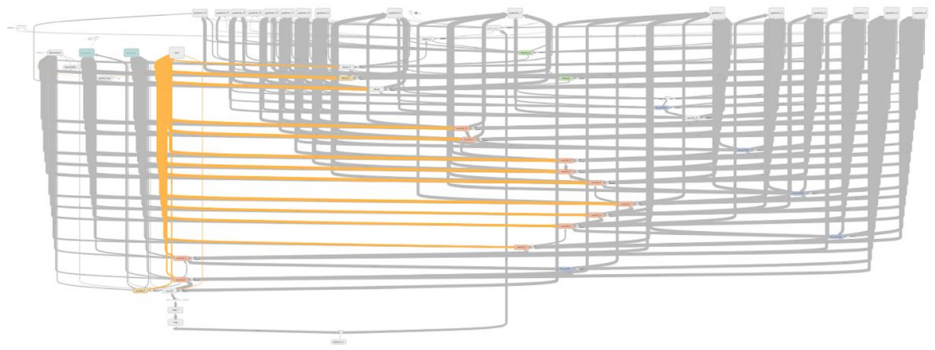

mAP
0.250
0.200
0.150
0.100
0.0500
0.00
2.000k
6.000k
10.00k
14.00k

Train images:

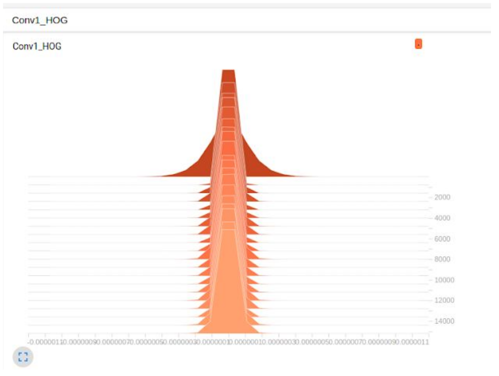


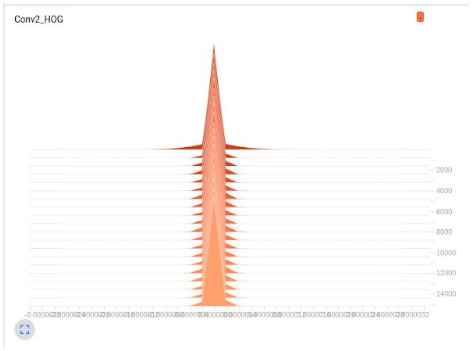
Network graph:

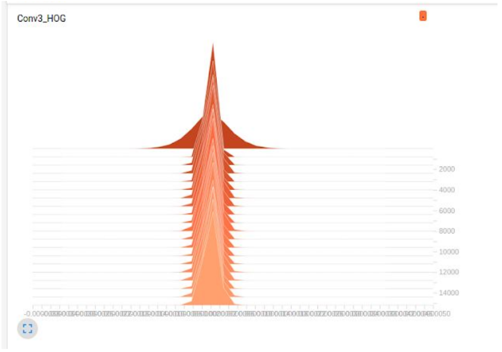
Note: This is really difficult to fit in a screen, this is my best shot at it.

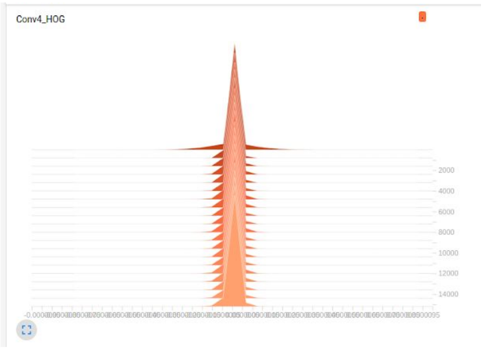


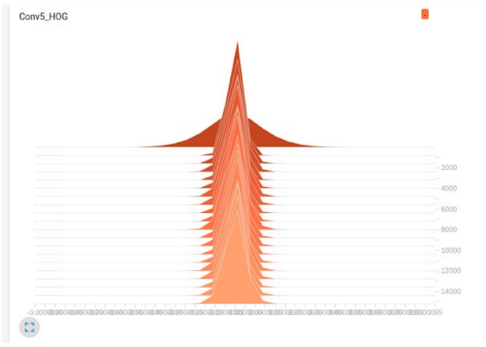
HOG for various filters:

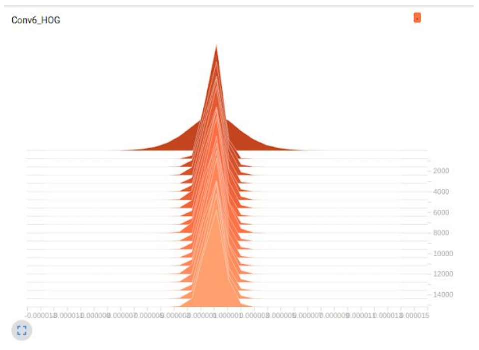


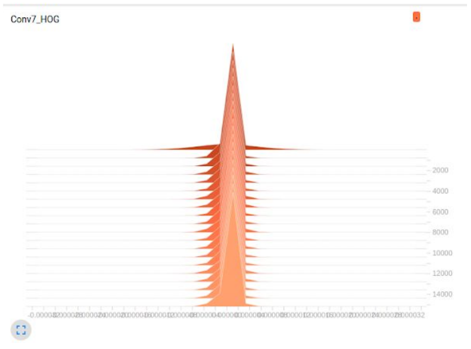


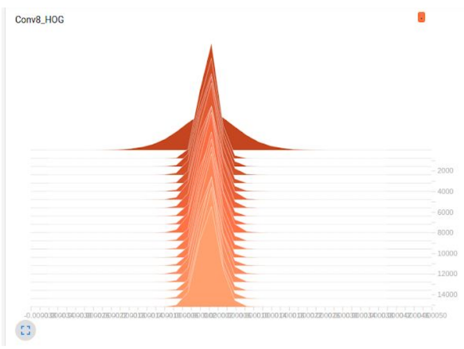


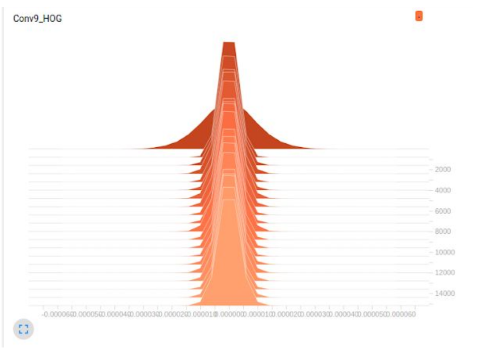


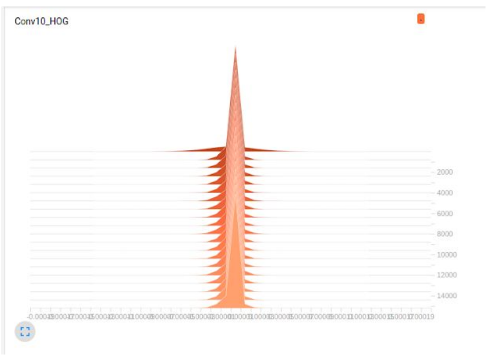


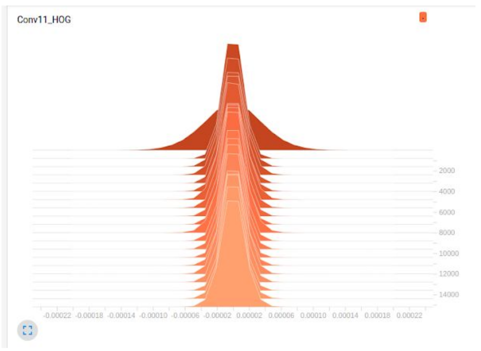


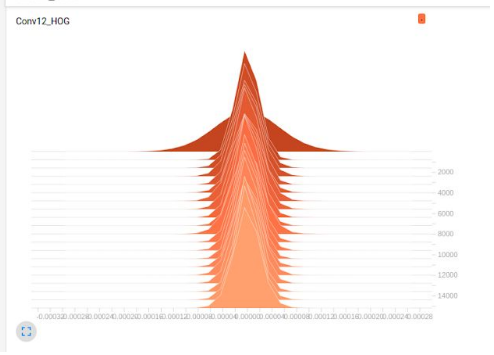


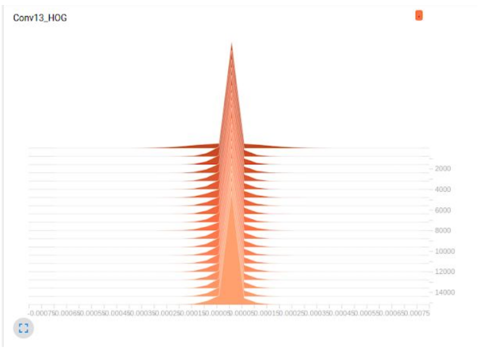


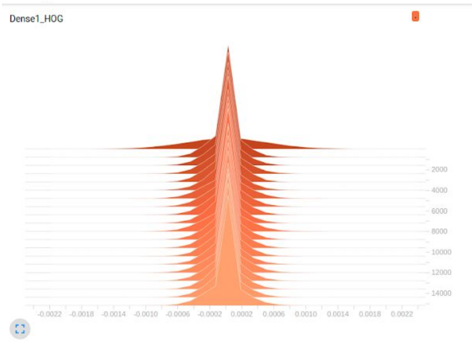

Conv9_HOG
2000
4000
6000
8000
10000
12000
14000
-0.000060
-0.000050
-0.000040
-0.000030
-0.000020
-0.000010
0.000000
0.000010
0.000020
0.000030
0.000040
0.000050
0.000060

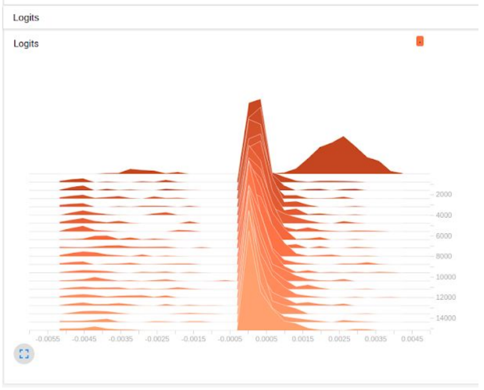

Conv10_HOG
2000
4000
6000
8000
10000
12000
14000
0.000019

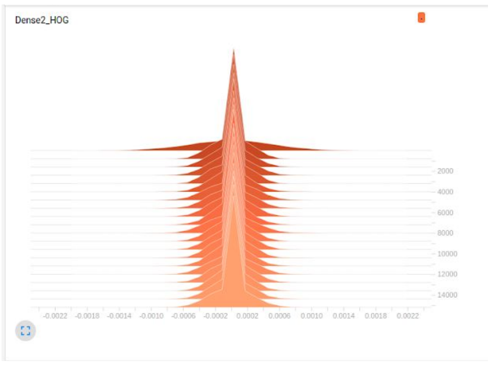

Conv11_HOG
2000
4000
6000
8000
10000
12000
14000
-0.00022
-0.00018
-0.00014
-0.00010
-0.00006
-0.00002
0.00002
0.00006
0.00010
0.00014
0.00018
0.00022


Conv12_HOG
2000
4000
6000
8000
10000
12000
14000
-0.00032
-0.00028
-0.00024
-0.00020
-0.00016
-0.00012
-0.00008
-0.00004
0.00000
0.00004
0.00008
0.00012
0.00016
0.00020
0.00024
0.00028


Conv13_HOG
2000
4000
6000
8000
10000
12000
14000
-0.00075
-0.00065
-0.00055
-0.00045
-0.00035
-0.00025
-0.00015
-0.00005
0.00005
0.00015
0.00025
0.00035
0.00045
0.00055
0.00065
0.00075


Dense1_HOG
2000
4000
6000
8000
10000
12000
14000
-0.0022
-0.0018
-0.0014
-0.0010
-0.0006
-0.0002
0.0002
0.0006
0.0010
0.0014
0.0018
0.0022


Logits
Logits
2000
4000
6000
8000
10000
12000
14000
-0.0055
-0.0045
-0.0035
-0.0025
-0.0015
-0.0005
0.0005
0.0015
0.0025
0.0035
0.0045


Dense2_HOG
2000
4000
6000
8000
10000
12000
14000
-0.0022
-0.0018
-0.0014
-0.0010
-0.0006
-0.0002
0.0002
0.0006
0.0010
0.0014
0.0018
0.0022

Task 4

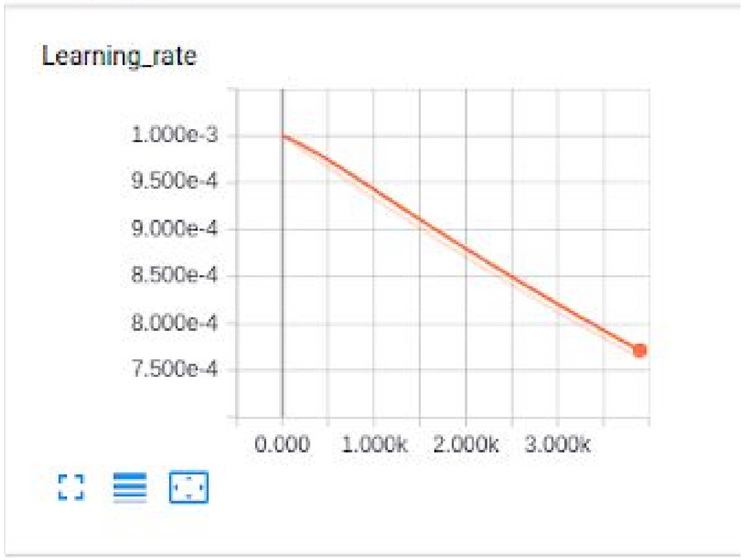
To run code type: python 04_pascal_vgg16_finetune.py VOCdevkit

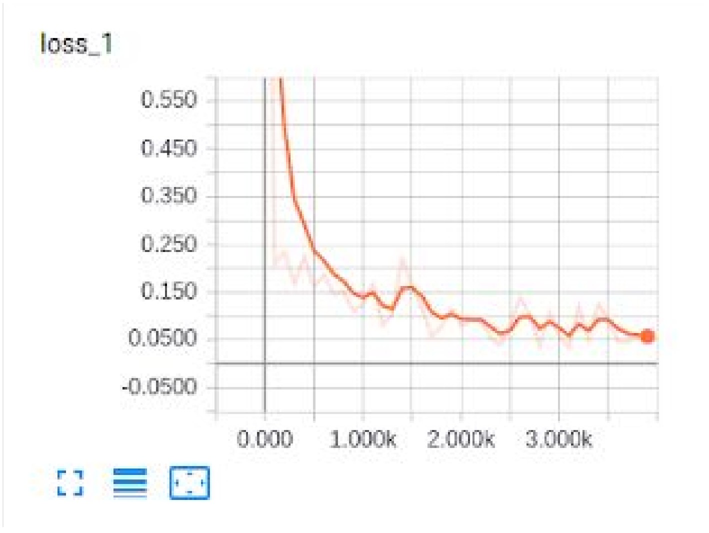
Analysis: I tried using the specified Gaussian initialization but that resulted in mAP either decreasing in few experiments or it being saturating to a very small value of around 0.06 in few experiments. After some experiments I found that keeping default weight initialization and a learning rate same of 0.001 instead of 0.0001 (as mentioned in question) works far better, I was getting more then around 35% improvement in test accuracy for lr = 0.001 over lr = 0.0001.

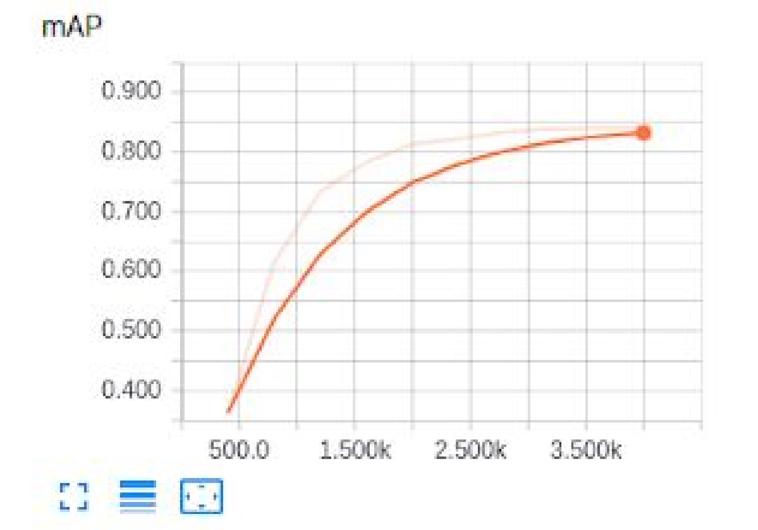
Hyper parameter:
Starting learning rate = 0.001
Batch size = 10
Momentum = 0.9
No. of iterations = 4,000
Weight initialization = Tensorflow default
Bias initialization = zero initialization

Loss for final step: 0.03603913

Obtained mAP: 0.840560359791




loss_1
0.550
0.450
0.350
0.250
0.150
0.0500
-0.0500
0.000
1.000k
2.000k
3.000k


mAP
0.900
0.800
0.700
0.600
0.500
0.400
500.0
1.500k
2.500k
3.500k

Task 5

To run code type:

```
python 05_Alex.py VOCdevkit
python 05_VGG.py VOCdevkit
```

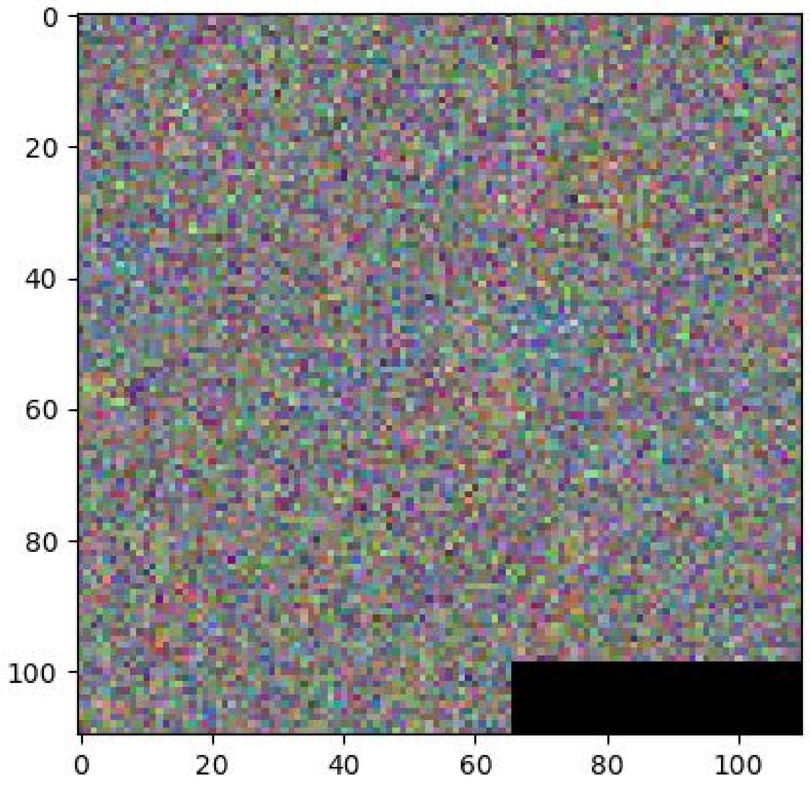
That will generate and store the numpy data required for further analysis in the same directory and then after that run:

```
python Task5_local_Alex.py
python Task5_local_VGG.py
```

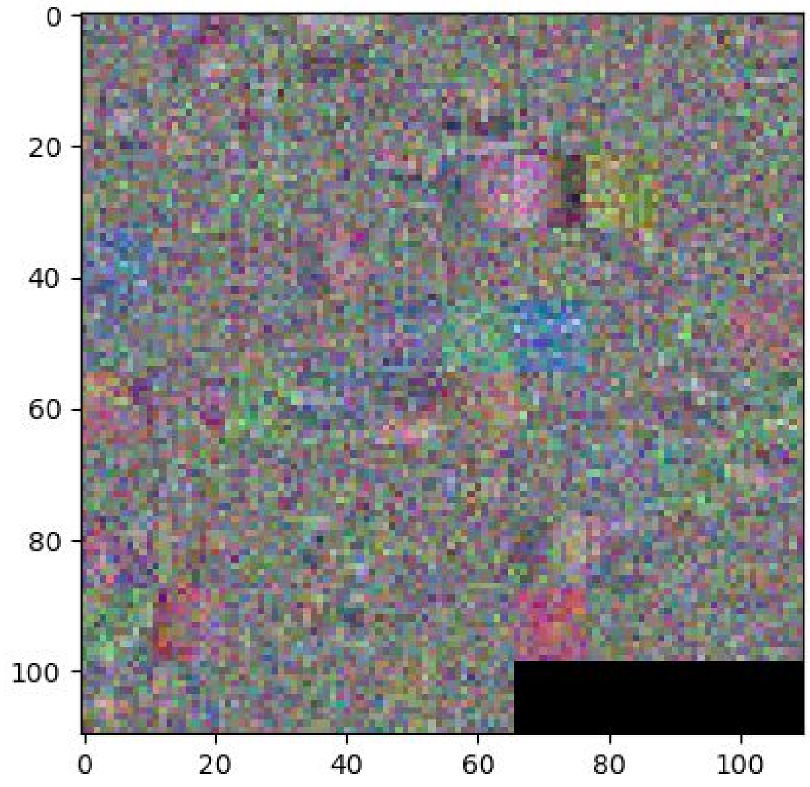
Note: In the above 2 files if it throws error then change “data_dir” variable at top in the code to point to “VOCdevkit” folder

Conv-1 filters:

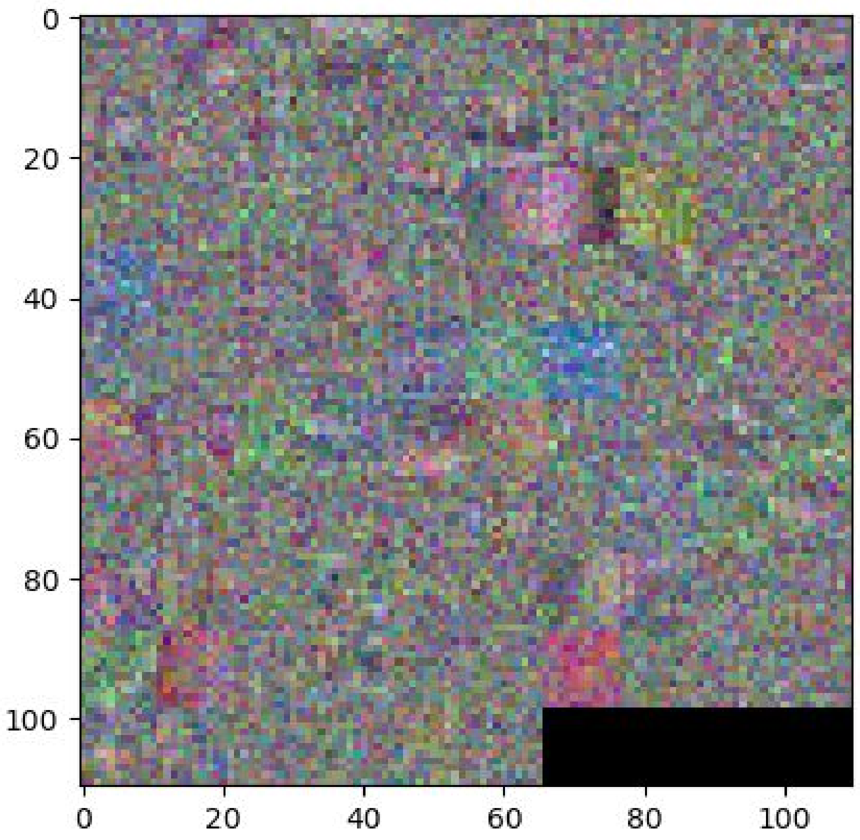
At the start of training (just random weights)



Half way through training

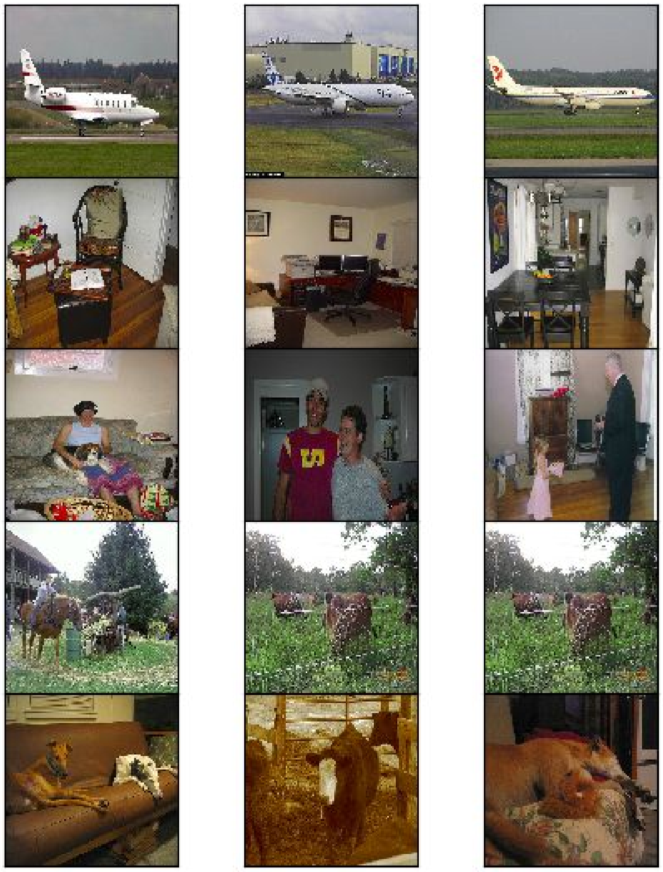


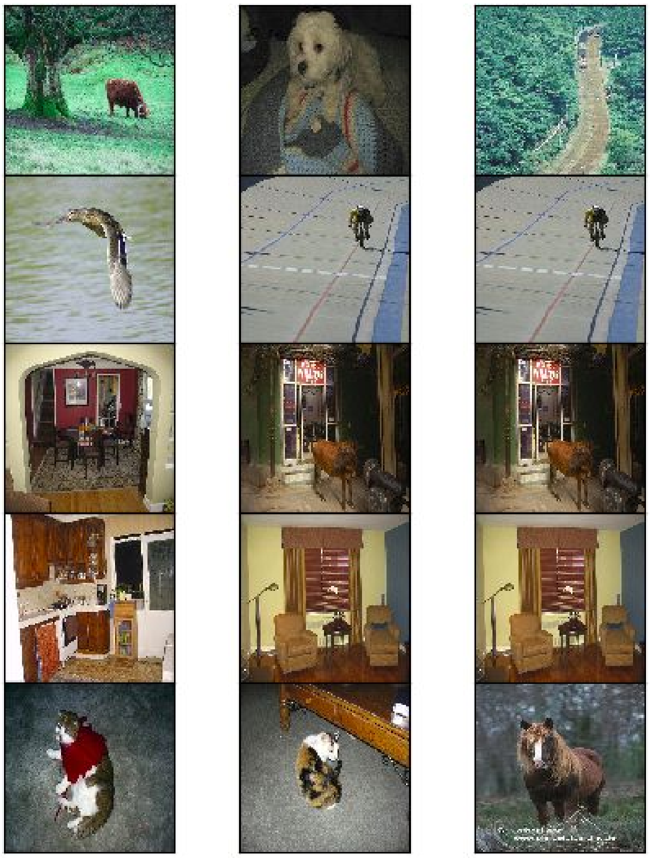
At the end of training time



Nearest neighbors - Alex Net

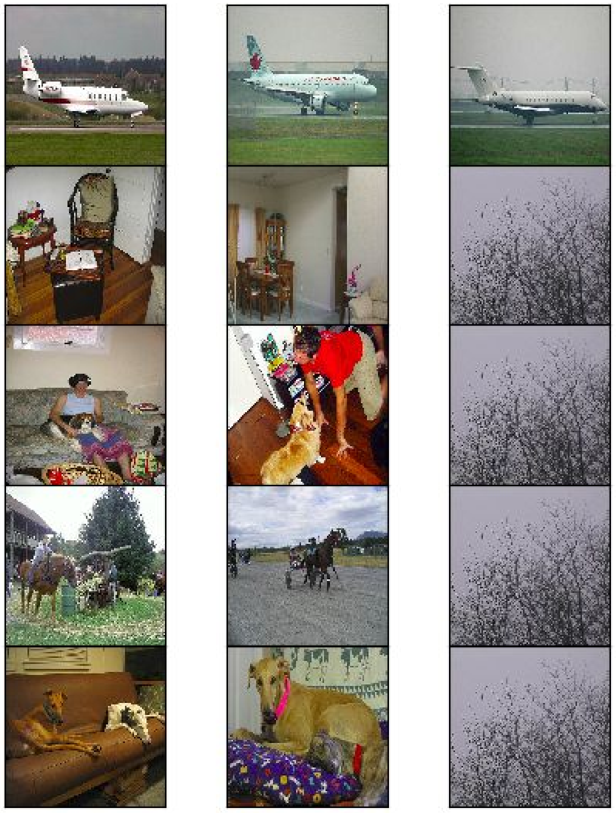
For below images:
1st column - test images
2nd column - fc7 features based nearest neighbour
3rd column - pool5 features based nearest neighbour

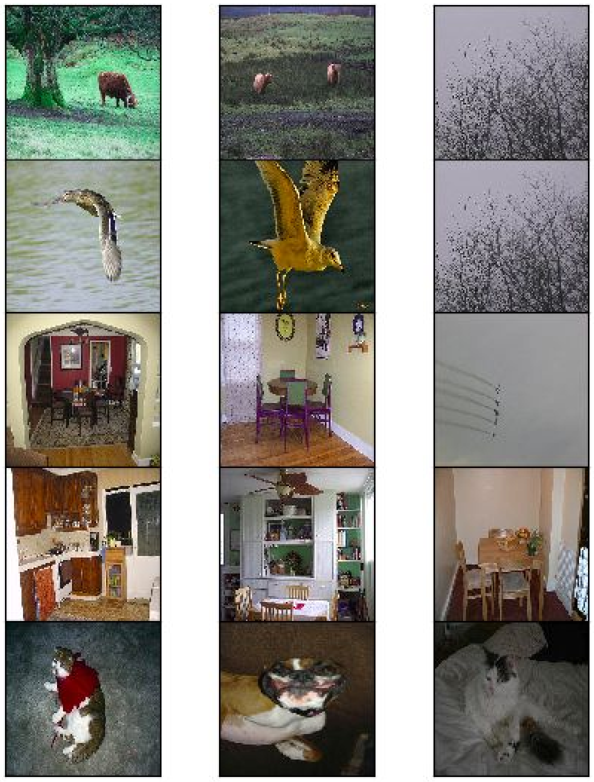




Nearest neighbors - VGG16 (finetune)

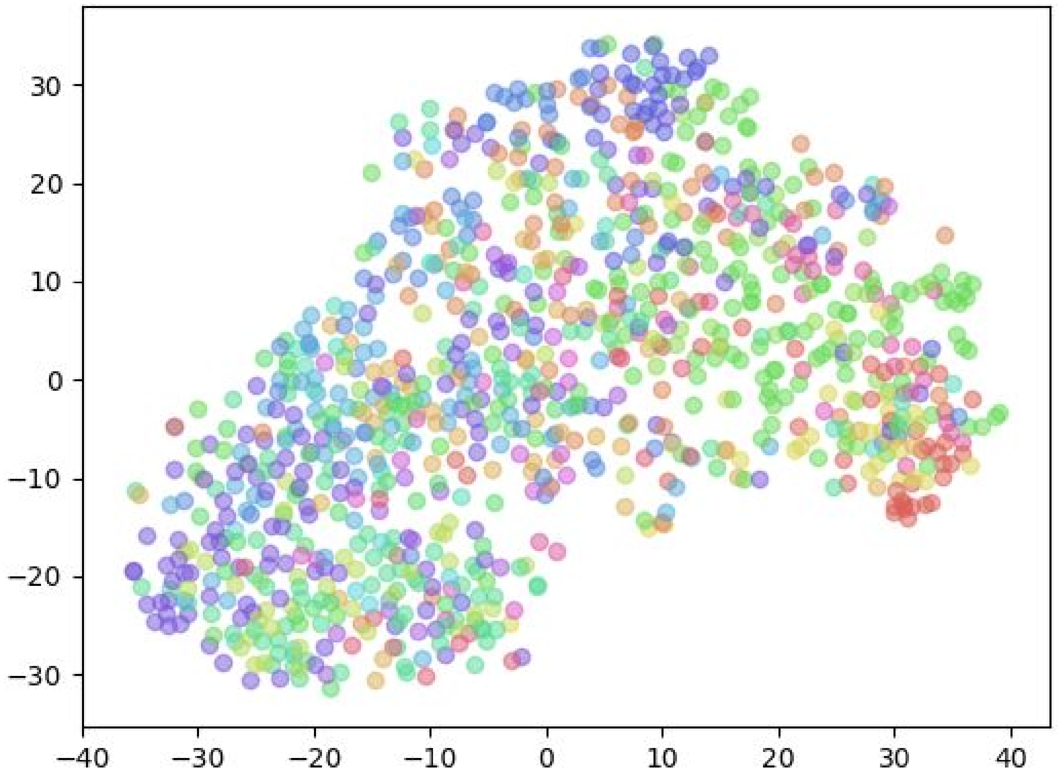
For below images:
1st column - test images
2nd column - fc7 features based nearest neighbour
3rd column - pool5 features based nearest neighbour

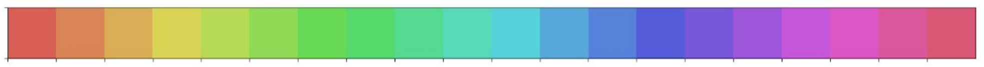




tSNE visualization of intermediate features

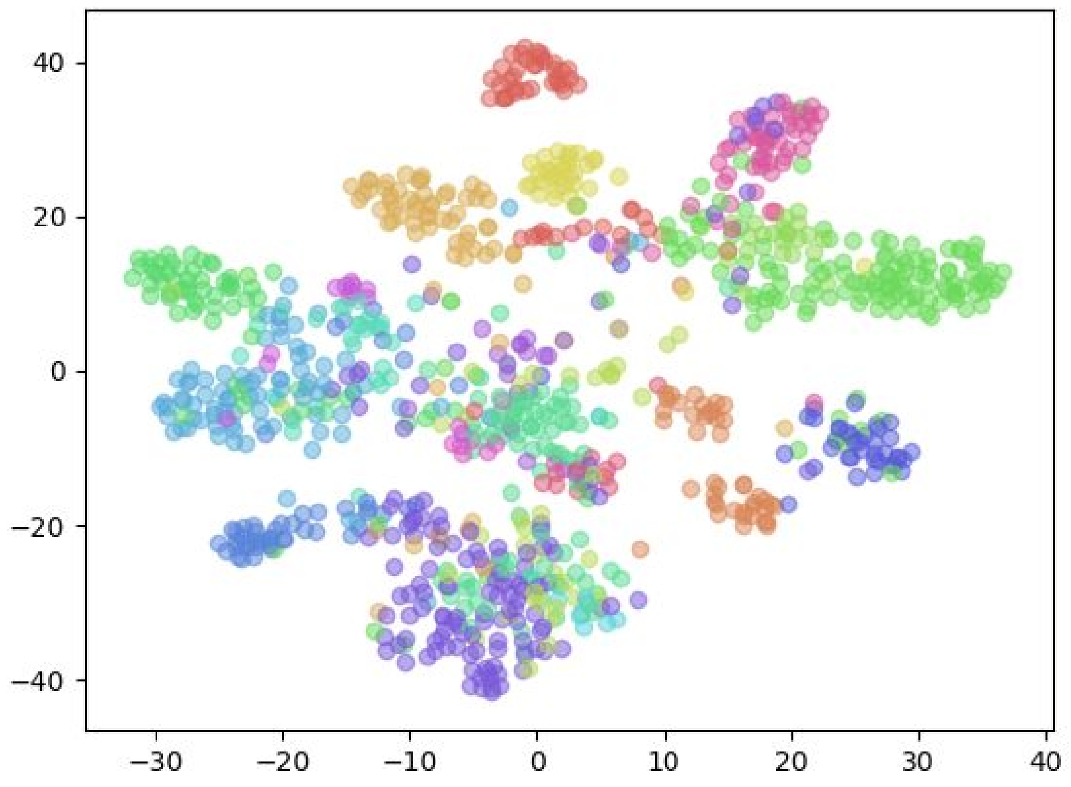
tSNE AlexNet:

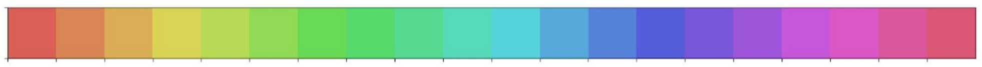




Scale: Leftmost color (red) is for class label = 1 i.e. aeroplane and it increase accordingly, so the righmot color (pink) is for class label =20 i.e. tvmonitor

tSNE VGG16(fine tune):





Scale: Leftmost color (red) is for class label = 1 i.e. aeroplane and it increase accordingly, so the righmot color (pink) is for class label =20 i.e. tvmonitor

Analysis: Features in VGG are very distinct in comparison with AlexNet as VGG is a better model.

Are some classes harder?

Per class mAP:

Class	Alexnet (scratch)	VGG16 (scratch)	VGG16 (fine tune)
aeroplane	0.596861458437	0.542845329478	0.960708122297
bicycle	0.287227927409	0.164497321268	0.926972363219
bird	0.238195003773	0.155321655149	0.938779260698
boat	0.326339412943	0.280099958961	0.916447606412
bottle	0.142603924724	0.129549453687	0.513327199762
bus	0.243904583987	0.154085124387	0.844761691743
car	0.585960637008	0.520279762372	0.929894365391
cat	0.305050311513	0.2359448984	0.913513285219
chair	0.351835094951	0.264455563915	0.640927963317
cow	0.164199664061	0.133854909007	0.798490644384
diningtable	0.29515567674	0.253781719755	0.773744752736
dog	0.275946640313	0.235048815378	0.903726223078
horse	0.616345309011	0.483887435765	0.919367302815
motorbike	0.469836205744	0.254425541053	0.890891714028
person	0.767944630768	0.69565179938	0.961002641829
pottedplant	0.179718453396	0.113876948706	0.657868758866
sheep	0.209802021552	0.204473434133	0.802228669266
sofa	0.292424769787	0.190744915488	0.74999013264
train	0.472935053056	0.356673034362	0.957100398443
tvmonitor	0.26103331768	0.135095258504	0.811464099681

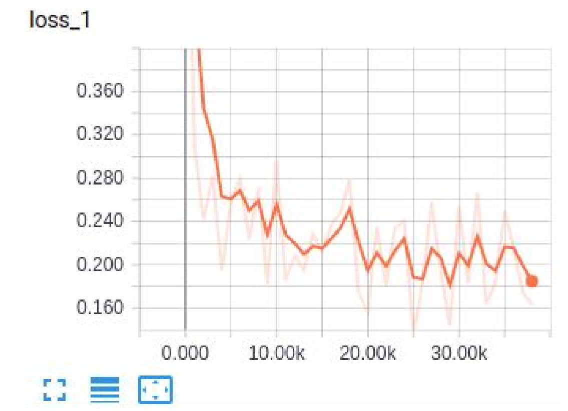
Analysis: As can be seen above some classes are difficult over others because VOC contains less images of those classes. All the classes are not uniform in number. For example images of bottles are very less and hence it has low accuracy, in fact the least. Also bottles are smaller in size and so inherently it’s difficult to find them. And also some classes are difficult to classify because they have lots of variations like view point and huge deformation.

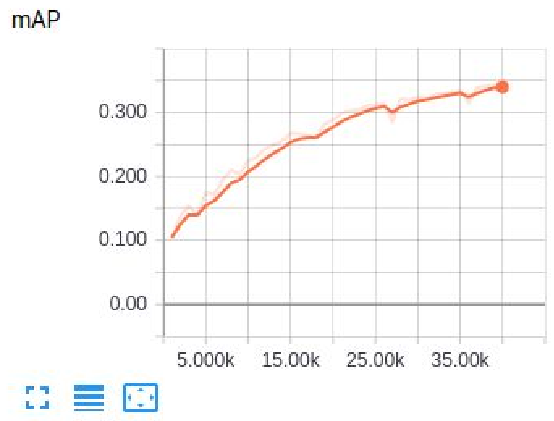
Also training on ImageNet improves the accuracy for all classes but some classes improve more then others. For example tv monitor’s accuracy is lesser than dining table in VOC but with Imagenet pretraining it becomes the other way, possibly because ImageNet has lot more examples of tv monitor.

Task 6

To run code type:
python 06_Alex.py VOCdevkit
python 06_VGG.py VOCdevkit

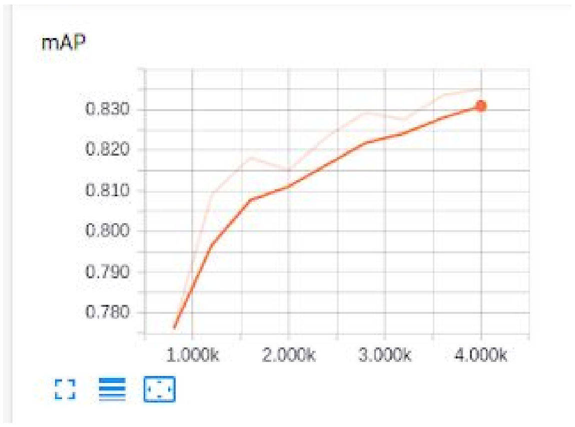
Mix- up on Alex Net

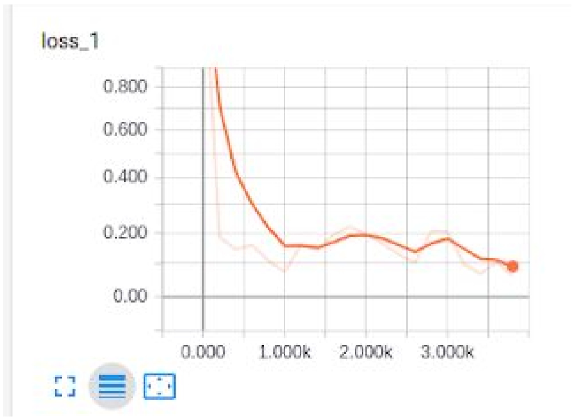




	Alexnet with mix-up	Original Alexnet
Loss for final step:	0.18235916	0.19311388
Obtained mAP :	0.342542216116	0.354166004843

Mix- up on VGG(fine tune)





	VGG with mix-up	Original VGG
Loss for final step	0.073412366	0.03603913
Obtained mAP	0.835166481009	0.840560359791

Analysis: Keeping all parameters same the above 2 tables compare Alexnet and VGG with and without using mix-up. As can be seen in both the cases mix-up leads to slightly lower test accuracy although it should be the opposite case.

One possible reason is, according to the paper performance improvement with mix-up would happen when training is done for large no. of epochs, which I didn’t do here. So probably if both models were trained even more then probably mix-up might result in better test accuracy. And that makes sense also because in mix-up you are essentially giving data which is a combination of 2 images (in a stochastic way) and so essentially the network would take lot more iterations to learn this distribution.