Name: Animesh Dhagat [adhagat] **SUB: VLR-Assignment-1**
Date: March 07, 2020

Task 0:
0.1.

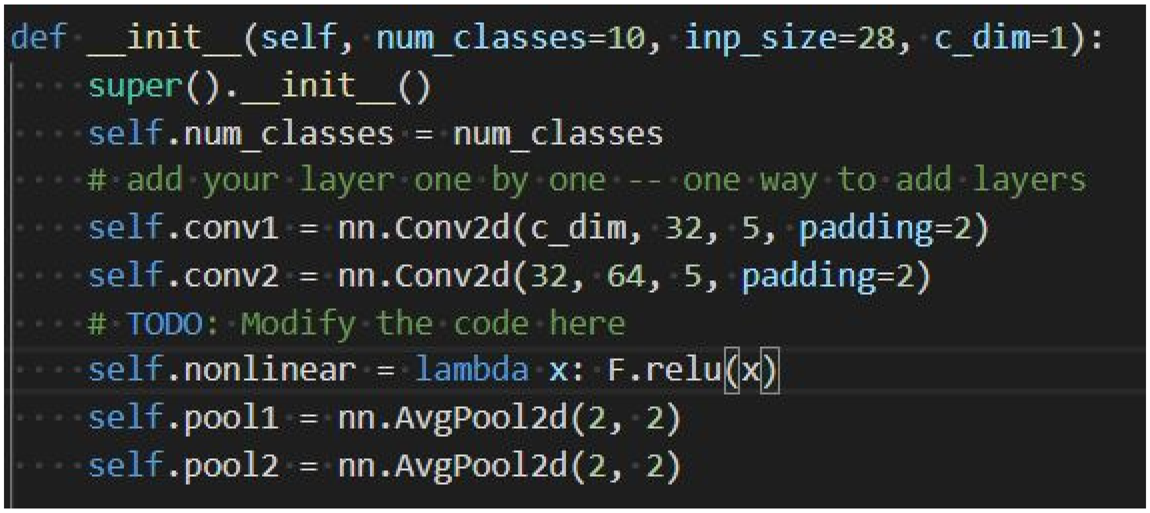
- **Number of flat_dim?**
  3136
- **Hyperparameters modified?**
  Learning Rate - Set it to 0.001

0.2. Trainable parameters in each layer?

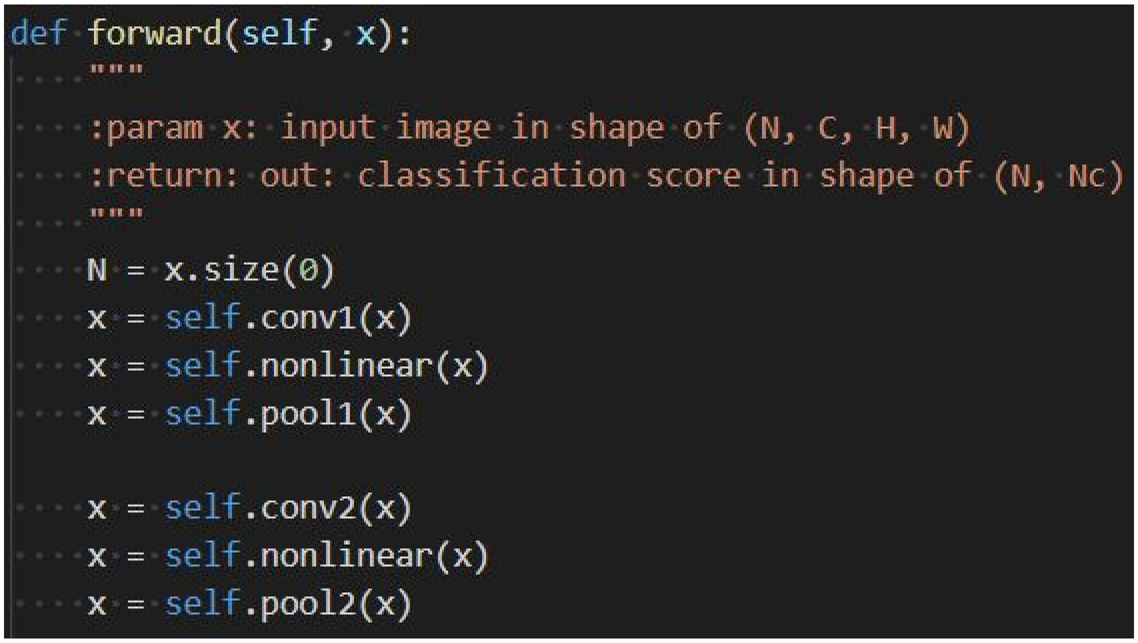
- Conv1: [(5*5*1) + 1]*32 = 832
- Conv2: [(5*5*32) + 1]*64 = 51264
- FC1: [3136*128 + 128] = 401536
- FC2: [128*10 + 10] = 1290
- Total Parameters = Conv1 + Conv2 + FC1 + FC2
  = 454922

0.3.

- **Which non-linearity was added?**
  ReLU non-linearity
- **Where did you modify the code to add this non-linearity?**



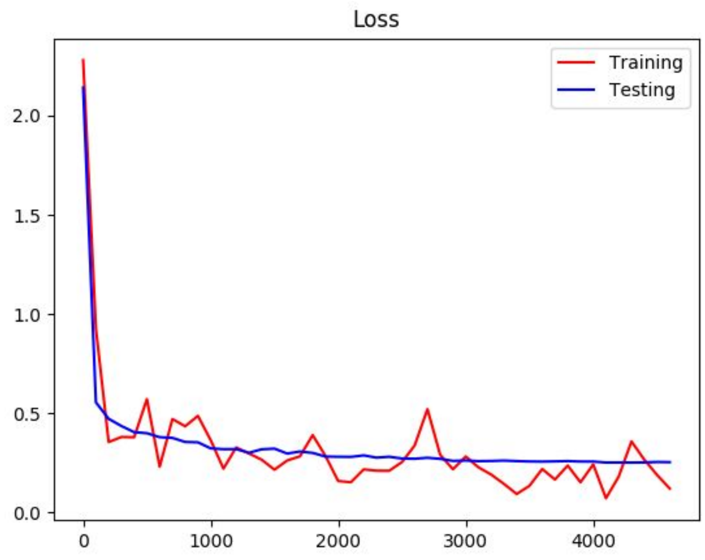
```
def __init__(self, num_classes=10, inp_size=28, c_dim=1):
    super().__init__()
    self.num_classes = num_classes
    # add your layer one by one -- one way to add layers
    self.conv1 = nn.Conv2d(c_dim, 32, 5, padding=2)
    self.conv2 = nn.Conv2d(32, 64, 5, padding=2)
    # TODO: Modify the code here
    self.nonlinear = lambda x: F.relu(x)
    self.pool1 = nn.AvgPool2d(2, 2)
    self.pool2 = nn.AvgPool2d(2, 2)
```



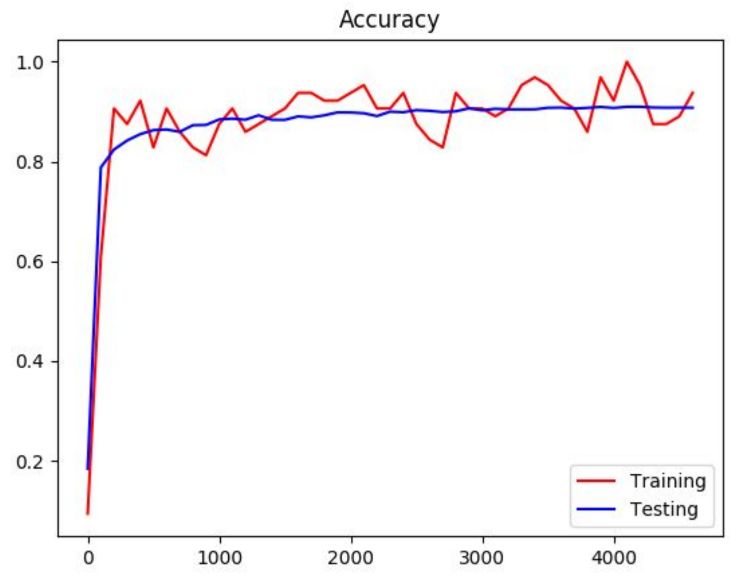
```
def forward(self, x):
    """
    :param x: input image in shape of (N, C, H, W)
    :return: out: classification score in shape of (N, Nc)
    """
    N = x.size(0)
    x = self.conv1(x)
    x = self.nonlinear(x)
    x = self.pool1(x)

    x = self.conv2(x)
    x = self.nonlinear(x)
    x = self.pool2(x)
```

- **Why were the results good even without the non-linearity?**
  Non-linearity helps model a response that varies non-linearly with the variables. However, since in this case, the number of classes are fewer (10), and the images are also simple (28x28 having only 1 grayscale channel), the response to this data can be modelled through a linear decision boundary to a reasonable extent.

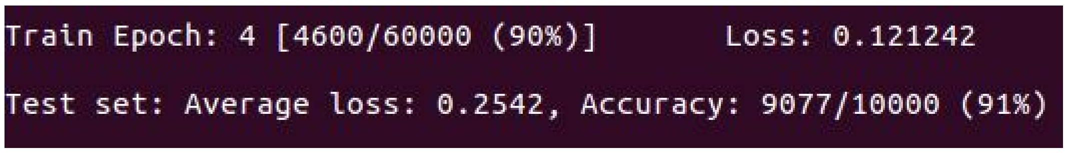
- **Loss and Accuracy curves - train & test for 5 epochs.**
  Loss:



Accuracy:



After 5 epochs:



```
Train Epoch: 4 [4600/60000 (90%)]       Loss: 0.121242

Test set: Average loss: 0.2542, Accuracy: 9077/10000 (91%)
```

Task 1:

1.2. What data augmentations were applied?

- **Training:**
    - **Resizing image to larger size**
    - **Random crops of canonical size on the resized image**
    - **Random Horizontal flips**
    - **Random vertical flips reduced the mAP, hence not applied**
- **Testing:**
    - **Resizing image to a larger size**
    - **Center crops to canonical size**

1.3. Describe how to compute AP for each class.

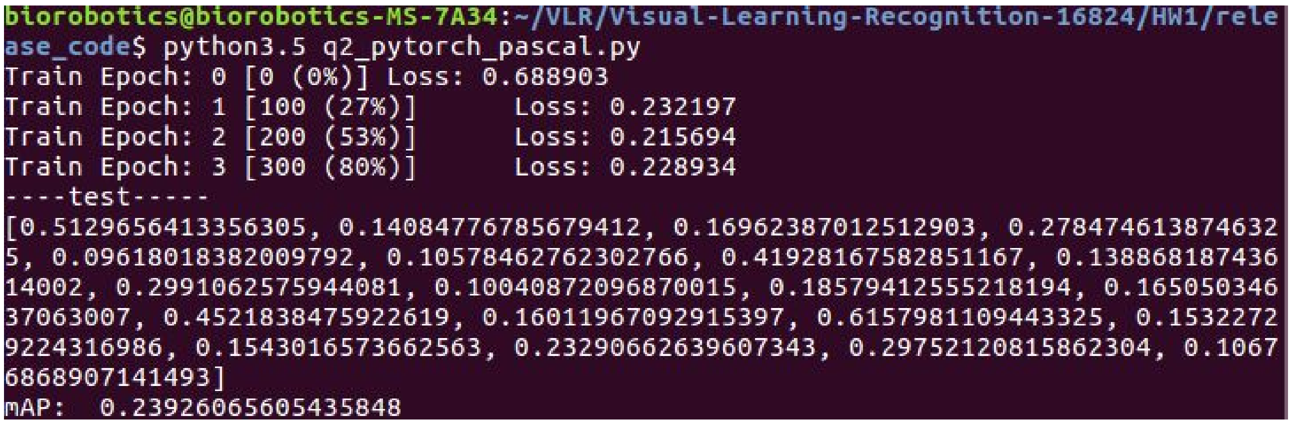
- **Example:**

Actual | Predicted	**Positive**	**Negative**
Positive	**True Positive**	**False Negative**
Negative	**False Positive**	**True Negative**

  Accuracy = num correctly predictions / total num samples
  Precision = TP / [TP + FP]
  Recall = TP / [TP + FN]

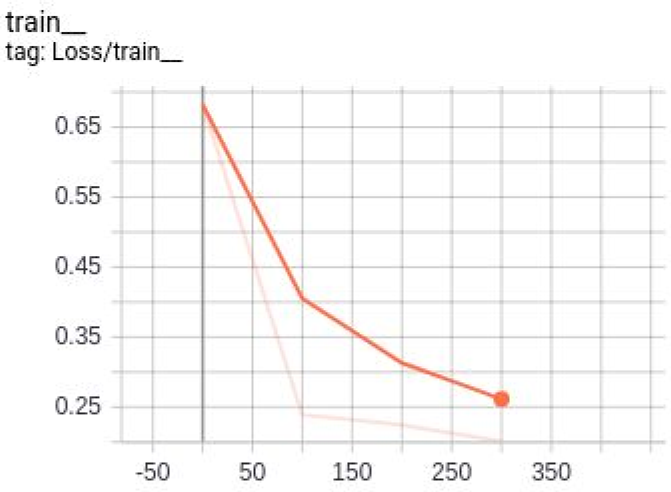
- The above confusion matrix is representative of the type of matrix formed in our case, except that we have 20 classes so the matrix would be 20 x 20.
- The prediction made by our network is stored in the corresponding position in the matrix.
  - A prediction is True Positive (TP) when the prediction is true for a class, and it is actually that class.
  - A prediction is False positive (FP) when the prediction is true for a class, but actually it is not that class.
  - A prediction is False Negative (FN) when the prediction is false for a class, but actually it is that class.
  - A prediction is True Negative (TN) when the prediction is false for a class, and it is actually not that class.
- Precision for each class is computed using the formula mentioned above.
- For each class, the precision values are averaged over to compute the Average Precision(AP).

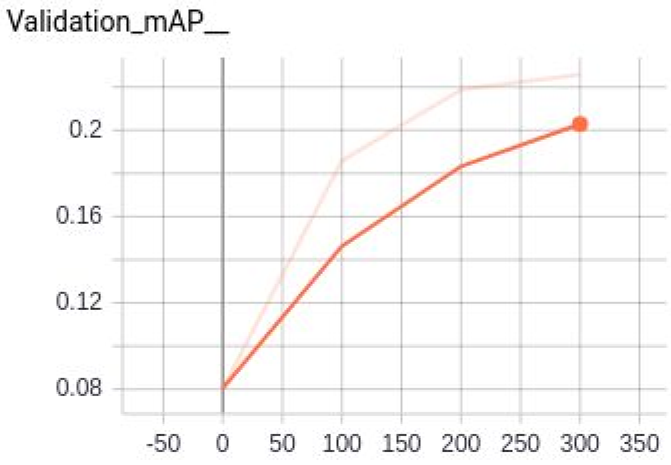
1.4. mAP on test set after 5 training epochs.



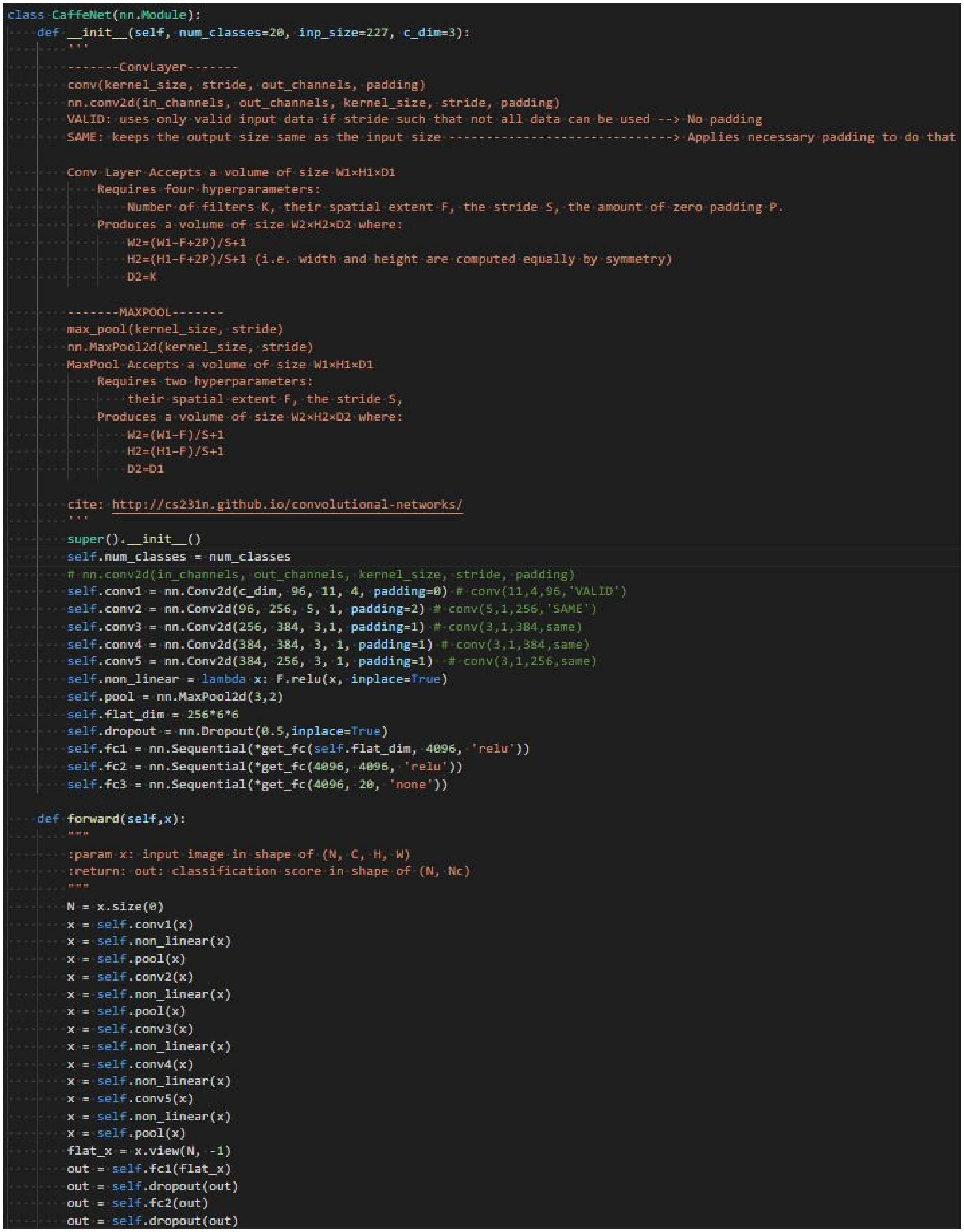
```
biorobotics@biorobotics-MS-7A34:~/VLR/Visual-Learning-Recognition-16824/HW1/rele
ase_code$ python3.5 q2_pytorch_pascal.py
Train Epoch: 0 [0 (0%)] Loss: 0.688903
Train Epoch: 1 [100 (27%)]      Loss: 0.232197
Train Epoch: 2 [200 (53%)]      Loss: 0.215694
Train Epoch: 3 [300 (80%)]      Loss: 0.228934
----test-----
[0.5129656413356305, 0.14084776785679412, 0.16962387012512903, 0.278474613874632
5, 0.09618018382009792, 0.10578462762302766, 0.41928167582851167, 0.138868187436
14002, 0.2991062575944081, 0.10040872096870015, 0.18579412555218194, 0.165050346
37063007, 0.4521838475922619, 0.16011967092915397, 0.6157981109443325, 0.1532272
9224316986, 0.1543016573662563, 0.23290662639607343, 0.29752120815862304, 0.1067
6868907141493]
mAP:  0.23926065605435848
```

1.5. Learning curves of testing mAP & training loss for 5 epochs



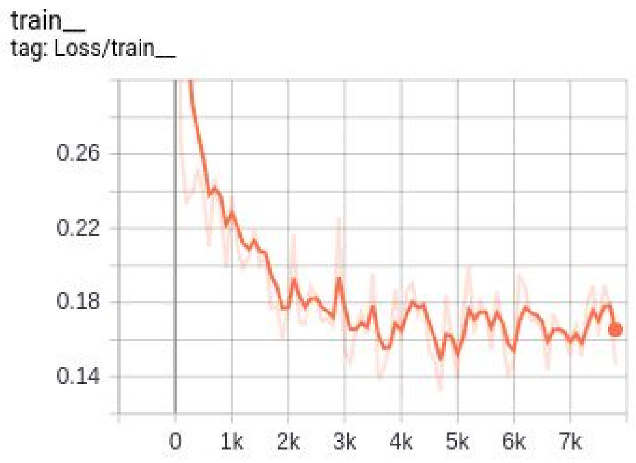


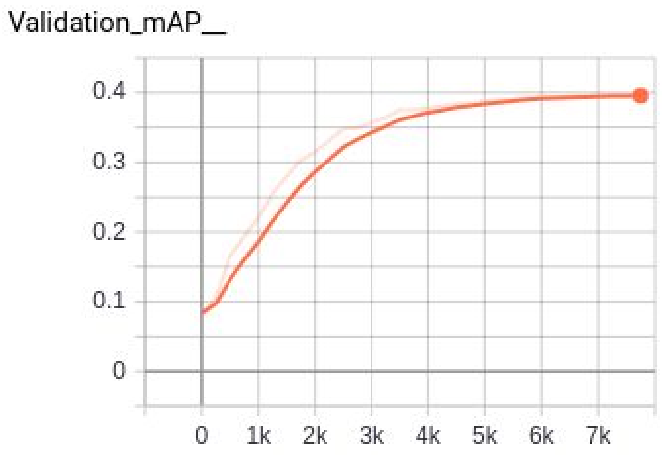
Task 2:
2.1. Screenshot of CaffeNet model



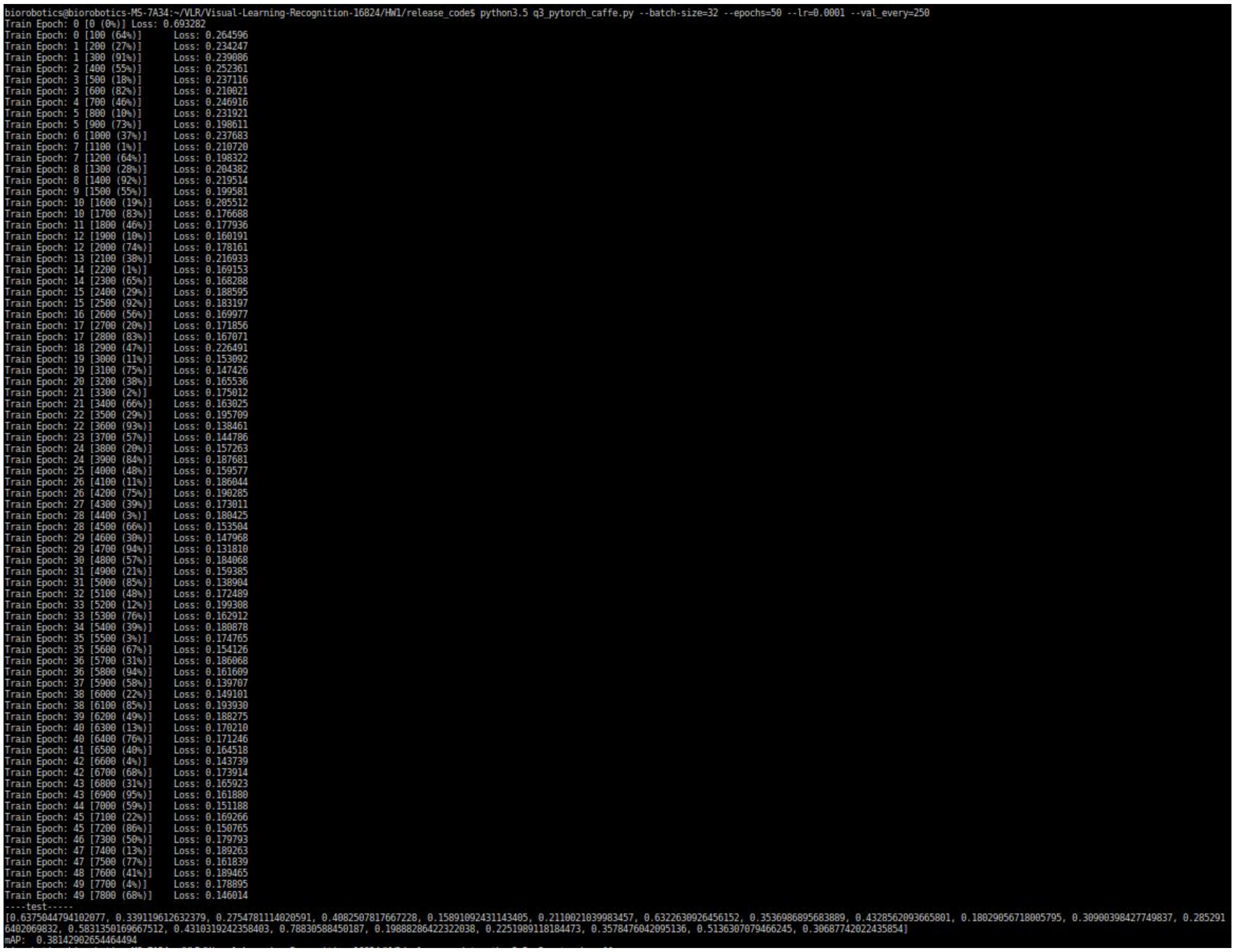
```python
class CaffeNet(nn.Module):
    def __init__(self, num_classes=20, inp_size=227, c_dim=3):
        '''
        -------ConvLayer-------
        conv(kernel_size, stride, out_channels, padding)
        nn.conv2d(in_channels, out_channels, kernel_size, stride, padding)
        VALID: uses only valid input data if stride such that not all data can be used --> No padding
        SAME: keeps the output size same as the input size -----------------------------> Applies necessary padding to do that

        Conv Layer Accepts a volume of size W1×H1×D1
            Requires four hyperparameters:
                Number of filters K, their spatial extent F, the stride S, the amount of zero padding P.
            Produces a volume of size W2×H2×D2 where:
                W2=(W1−F+2P)/S+1
                H2=(H1−F+2P)/S+1 (i.e. width and height are computed equally by symmetry)
                D2=K

        -------MAXPOOL-------
        max_pool(kernel_size, stride)
        nn.MaxPool2d(kernel_size, stride)
        MaxPool Accepts a volume of size W1×H1×D1
            Requires two hyperparameters:
                their spatial extent F, the stride S,
            Produces a volume of size W2×H2×D2 where:
                W2=(W1−F)/S+1
                H2=(H1−F)/S+1
                D2=D1

        cite: http://cs231n.github.io/convolutional-networks/
        '''
        super().__init__()
        self.num_classes = num_classes
        # nn.conv2d(in_channels, out_channels, kernel_size, stride, padding)
        self.conv1 = nn.Conv2d(c_dim, 96, 11, 4, padding=0) # conv(11,4,96,'VALID')
        self.conv2 = nn.Conv2d(96, 256, 5, 1, padding=2) # conv(5,1,256,'SAME')
        self.conv3 = nn.Conv2d(256, 384, 3,1, padding=1) # conv(3,1,384,same)
        self.conv4 = nn.Conv2d(384, 384, 3, 1, padding=1) # conv(3,1,384,same)
        self.conv5 = nn.Conv2d(384, 256, 3, 1, padding=1)  # conv(3,1,256,same)
        self.non_linear = lambda x: F.relu(x, inplace=True)
        self.pool = nn.MaxPool2d(3,2)
        self.flat_dim = 256*6*6
        self.dropout = nn.Dropout(0.5,inplace=True)
        self.fc1 = nn.Sequential(*get_fc(self.flat_dim, 4096, 'relu'))
        self.fc2 = nn.Sequential(*get_fc(4096, 4096, 'relu'))
        self.fc3 = nn.Sequential(*get_fc(4096, 20, 'none'))

    def forward(self,x):
        """
        :param x: input image in shape of (N, C, H, W)
        :return: out: classification score in shape of (N, Nc)
        """
        N = x.size(0)
        x = self.conv1(x)
        x = self.non_linear(x)
        x = self.pool(x)
        x = self.conv2(x)
        x = self.non_linear(x)
        x = self.pool(x)
        x = self.conv3(x)
        x = self.non_linear(x)
        x = self.conv4(x)
        x = self.non_linear(x)
        x = self.conv5(x)
        x = self.non_linear(x)
        x = self.pool(x)
        flat_x = x.view(N, -1)
        out = self.fc1(flat_x)
        out = self.dropout(out)
        out = self.fc2(out)
        out = self.dropout(out)
```

2.3. mAP and training loss for 50 epochs. Final mAP.



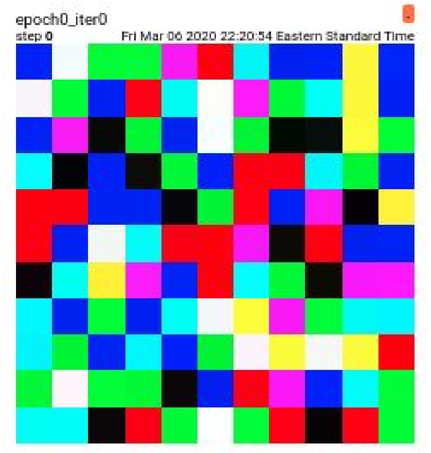


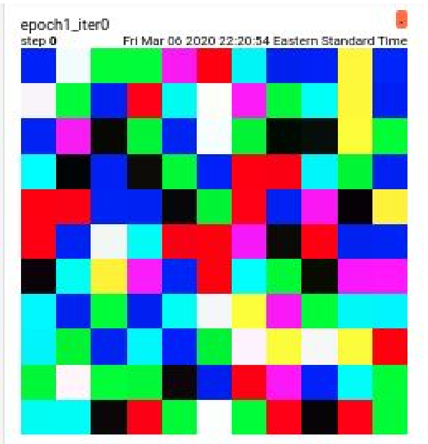
Final mAP: 0.3814

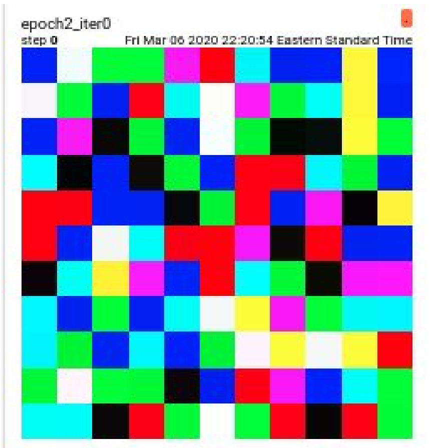


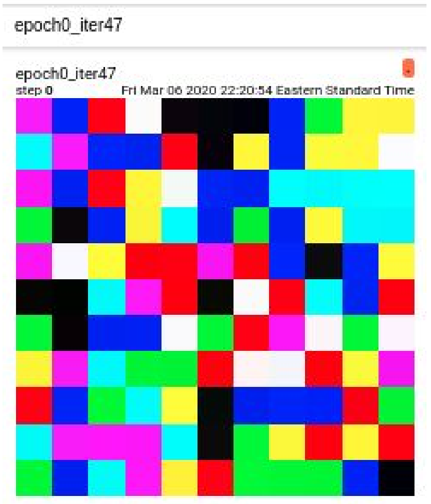
```
biorobotics@biorobotics-MS-7A34:~/VLR/Visual-Learning-Recognition-16824/HW1/release_code$ python3.5 q3_pytorch_caffe.py --batch-size=32 --epochs=50 --lr=0.0001 --val_every=250
Train Epoch: 0 [0 (0%)] Loss: 0.693282
Train Epoch: 0 [100 (64%)]      Loss: 0.264596
Train Epoch: 1 [200 (27%)]      Loss: 0.234247
Train Epoch: 1 [300 (91%)]      Loss: 0.239086
Train Epoch: 2 [400 (55%)]      Loss: 0.252361
Train Epoch: 3 [500 (18%)]      Loss: 0.237116
Train Epoch: 3 [600 (82%)]      Loss: 0.210021
Train Epoch: 4 [700 (46%)]      Loss: 0.246916
Train Epoch: 5 [800 (10%)]      Loss: 0.231921
Train Epoch: 5 [900 (73%)]      Loss: 0.198611
Train Epoch: 6 [1000 (37%)]     Loss: 0.237683
Train Epoch: 7 [1100 (1%)]      Loss: 0.210720
Train Epoch: 7 [1200 (64%)]     Loss: 0.198322
Train Epoch: 8 [1300 (28%)]     Loss: 0.204382
Train Epoch: 8 [1400 (92%)]     Loss: 0.219514
Train Epoch: 9 [1500 (55%)]     Loss: 0.199581
Train Epoch: 10 [1600 (19%)]    Loss: 0.205512
Train Epoch: 10 [1700 (83%)]    Loss: 0.176688
Train Epoch: 11 [1800 (46%)]    Loss: 0.177936
Train Epoch: 12 [1900 (10%)]    Loss: 0.160191
Train Epoch: 12 [2000 (74%)]    Loss: 0.178161
Train Epoch: 13 [2100 (38%)]    Loss: 0.216933
Train Epoch: 14 [2200 (1%)]     Loss: 0.169153
Train Epoch: 14 [2300 (65%)]    Loss: 0.168288
Train Epoch: 15 [2400 (29%)]    Loss: 0.188595
Train Epoch: 15 [2500 (92%)]    Loss: 0.183197
Train Epoch: 16 [2600 (56%)]    Loss: 0.169977
Train Epoch: 17 [2700 (20%)]    Loss: 0.171856
Train Epoch: 17 [2800 (83%)]    Loss: 0.167071
Train Epoch: 18 [2900 (47%)]    Loss: 0.226491
Train Epoch: 19 [3000 (11%)]    Loss: 0.153092
Train Epoch: 19 [3100 (75%)]    Loss: 0.147426
Train Epoch: 20 [3200 (38%)]    Loss: 0.165536
Train Epoch: 21 [3300 (2%)]     Loss: 0.175012
Train Epoch: 21 [3400 (66%)]    Loss: 0.163025
Train Epoch: 22 [3500 (29%)]    Loss: 0.195709
Train Epoch: 22 [3600 (93%)]    Loss: 0.138461
Train Epoch: 23 [3700 (57%)]    Loss: 0.144786
Train Epoch: 24 [3800 (20%)]    Loss: 0.157263
Train Epoch: 24 [3900 (84%)]    Loss: 0.187681
Train Epoch: 25 [4000 (48%)]    Loss: 0.159577
Train Epoch: 26 [4100 (11%)]    Loss: 0.186044
Train Epoch: 26 [4200 (75%)]    Loss: 0.190285
Train Epoch: 27 [4300 (39%)]    Loss: 0.173011
Train Epoch: 28 [4400 (3%)]     Loss: 0.180425
Train Epoch: 28 [4500 (66%)]    Loss: 0.153504
Train Epoch: 29 [4600 (30%)]    Loss: 0.147968
Train Epoch: 29 [4700 (94%)]    Loss: 0.131810
Train Epoch: 30 [4800 (57%)]    Loss: 0.184068
Train Epoch: 31 [4900 (21%)]    Loss: 0.159385
Train Epoch: 31 [5000 (85%)]    Loss: 0.138904
Train Epoch: 32 [5100 (48%)]    Loss: 0.172489
Train Epoch: 33 [5200 (12%)]    Loss: 0.199308
Train Epoch: 33 [5300 (76%)]    Loss: 0.162912
Train Epoch: 34 [5400 (39%)]    Loss: 0.180878
Train Epoch: 35 [5500 (3%)]     Loss: 0.174765
Train Epoch: 35 [5600 (67%)]    Loss: 0.154126
Train Epoch: 36 [5700 (31%)]    Loss: 0.186068
Train Epoch: 36 [5800 (94%)]    Loss: 0.161609
Train Epoch: 37 [5900 (58%)]    Loss: 0.139707
Train Epoch: 38 [6000 (22%)]    Loss: 0.149101
Train Epoch: 38 [6100 (85%)]    Loss: 0.193930
Train Epoch: 39 [6200 (49%)]    Loss: 0.188275
Train Epoch: 40 [6300 (13%)]    Loss: 0.170210
Train Epoch: 40 [6400 (76%)]    Loss: 0.171246
Train Epoch: 41 [6500 (40%)]    Loss: 0.164518
Train Epoch: 42 [6600 (4%)]     Loss: 0.143739
Train Epoch: 42 [6700 (68%)]    Loss: 0.173914
Train Epoch: 43 [6800 (31%)]    Loss: 0.165923
Train Epoch: 43 [6900 (95%)]    Loss: 0.161880
Train Epoch: 44 [7000 (59%)]    Loss: 0.151188
Train Epoch: 45 [7100 (22%)]    Loss: 0.169266
Train Epoch: 45 [7200 (86%)]    Loss: 0.150765
Train Epoch: 46 [7300 (50%)]    Loss: 0.179793
Train Epoch: 47 [7400 (13%)]    Loss: 0.189263
Train Epoch: 47 [7500 (77%)]    Loss: 0.161839
Train Epoch: 48 [7600 (41%)]    Loss: 0.189465
Train Epoch: 49 [7700 (4%)]     Loss: 0.178895
Train Epoch: 49 [7800 (68%)]    Loss: 0.146014
----test-----
[0.6375044794102077, 0.339119612632379, 0.2754781114020591, 0.4082507817667228, 0.15891092431143405, 0.2110021039983457, 0.6322630926456152, 0.3536986895683889, 0.4328562093665801, 0.18029056718005795, 0.30900398427749837, 0.285291
6402069832, 0.5831350169667512, 0.4310319242358403, 0.78830588450187, 0.19888286422322038, 0.2251989118184473, 0.3578476042095136, 0.5136307079466245, 0.30687742022435854]
mAP:  0.38142902654464494
```

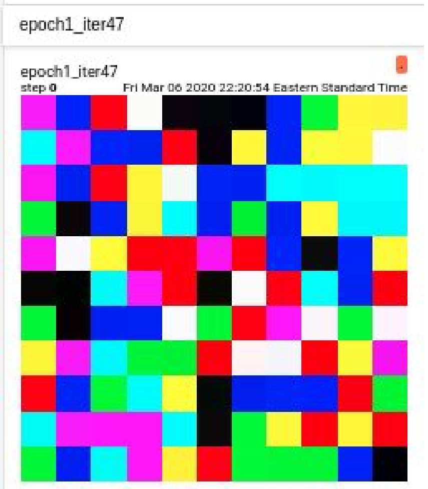
2.4 Visualizing Conv-1 filters.

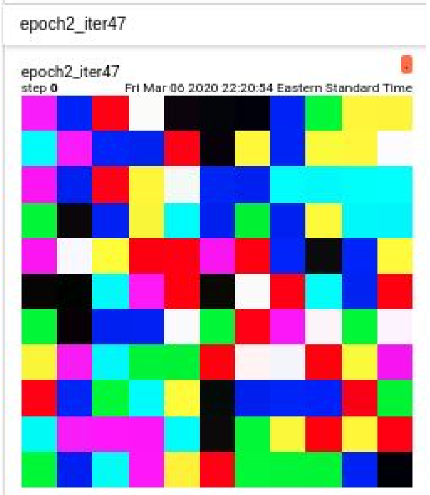


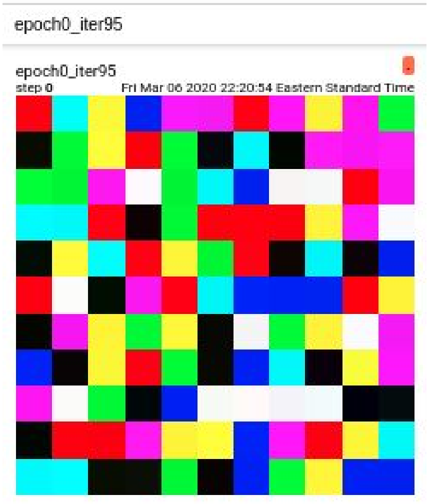


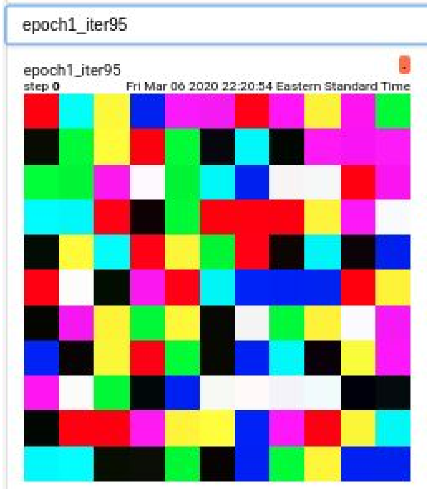


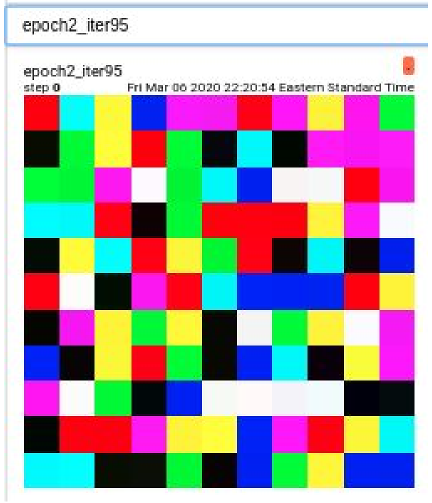




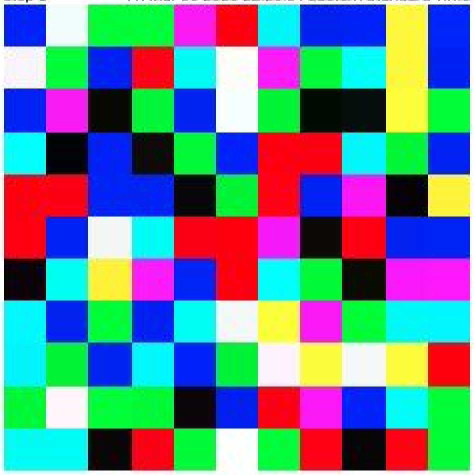




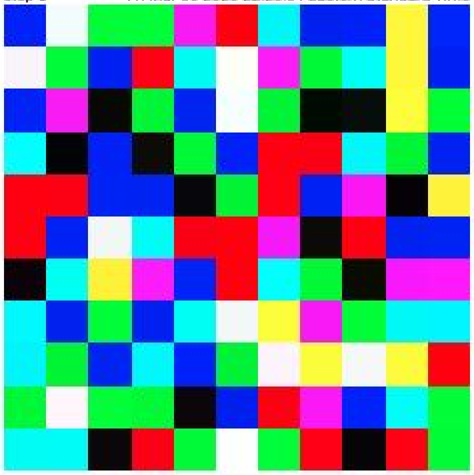




epoch3_iter0
step **0** Fri Mar 06 2020 22:20:54 Eastern Standard Time

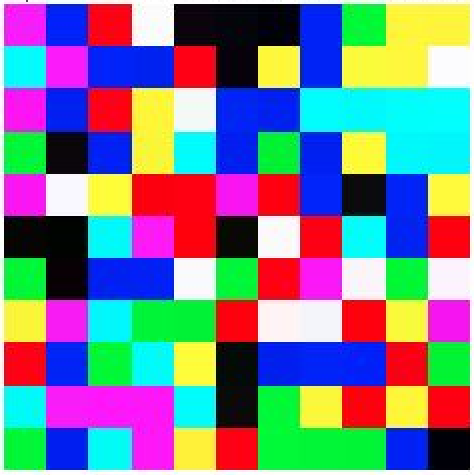


epoch4_iter0
step **0** Fri Mar 06 2020 22:20:54 Eastern Standard Time



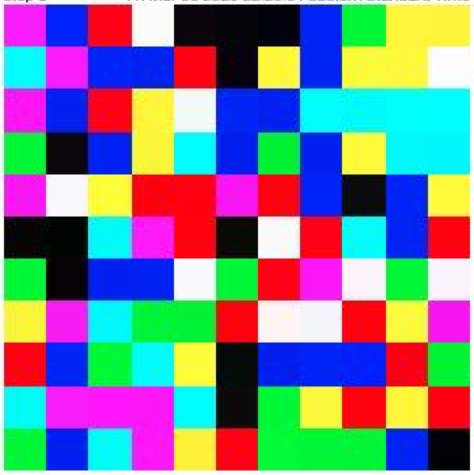
epoch3_iter47

epoch3_iter47
step **0** Fri Mar 06 2020 22:20:54 Eastern Standard Time



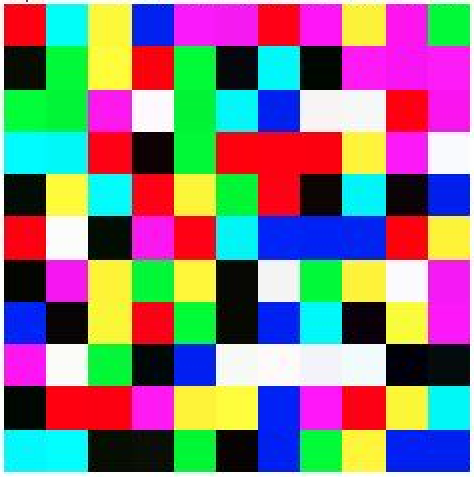
epoch4_iter47

epoch4_iter47
step **0** Fri Mar 06 2020 22:20:54 Eastern Standard Time



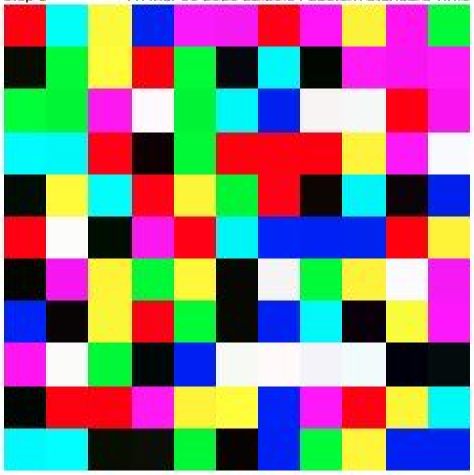
epoch3_iter95

epoch3_iter95
step **0** Fri Mar 06 2020 22:20:54 Eastern Standard Time



epoch4_iter95

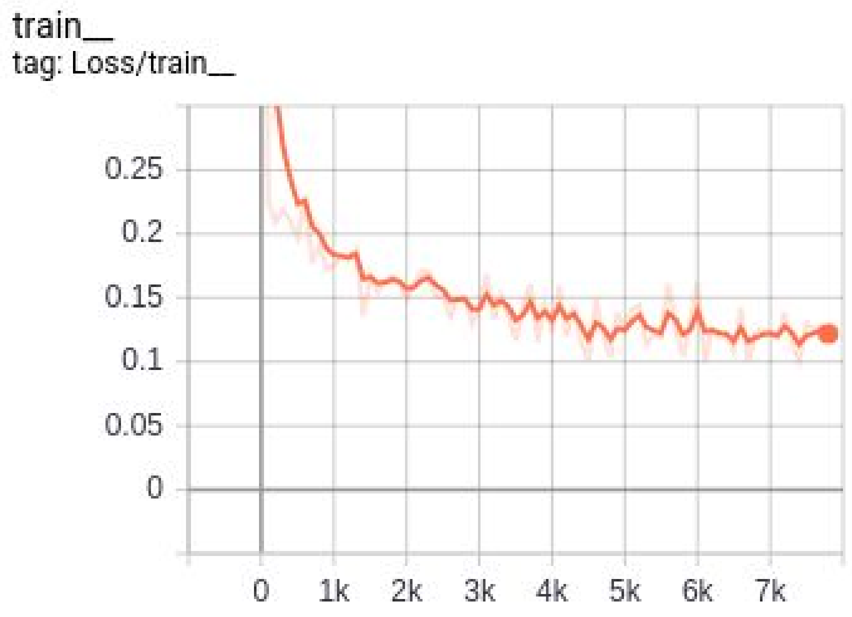
epoch4_iter95
step **0** Fri Mar 06 2020 22:20:54 Eastern Standard Time



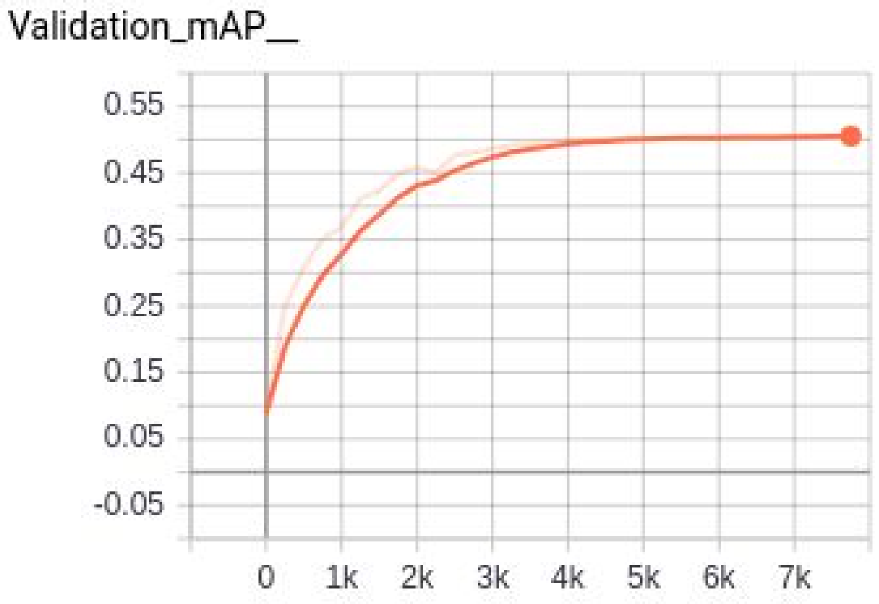
Task 3:

3.2. Screenshots:

- **Training loss**



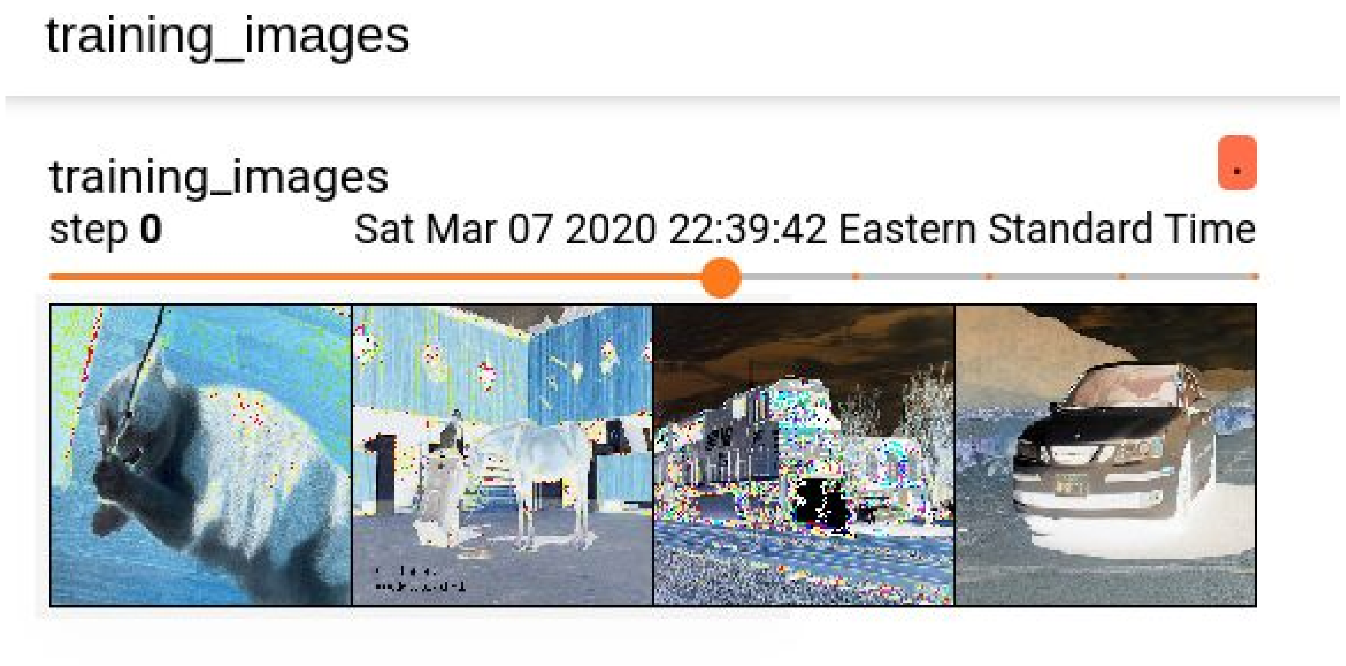
- **Testing mAP**



- **Learning rate**

  --lr=0.0001, --gamma=0.9

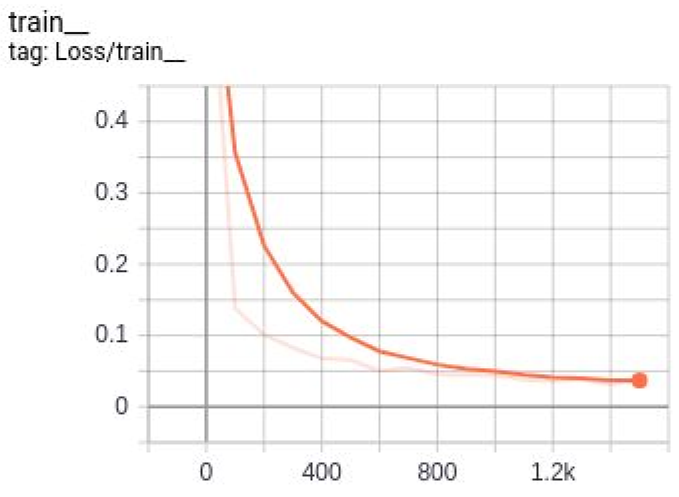
- **3 examples of training images from TensorBoard in different iterations**



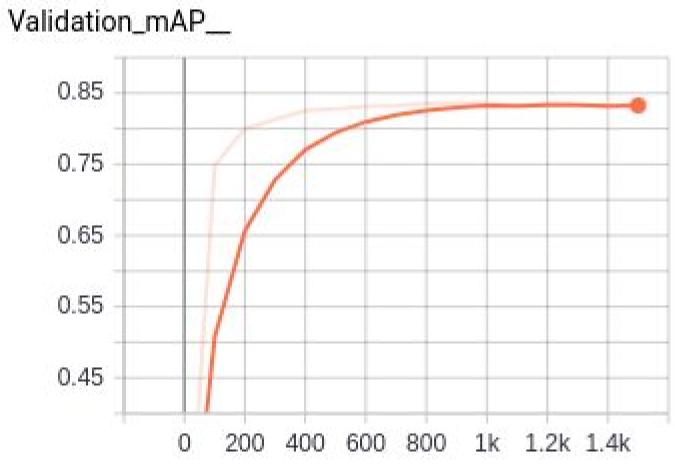
Task 4:
4.1. How to load weights in the model if the weight-names do not match?

4.2. Learning Curves:

- **Training Loss**



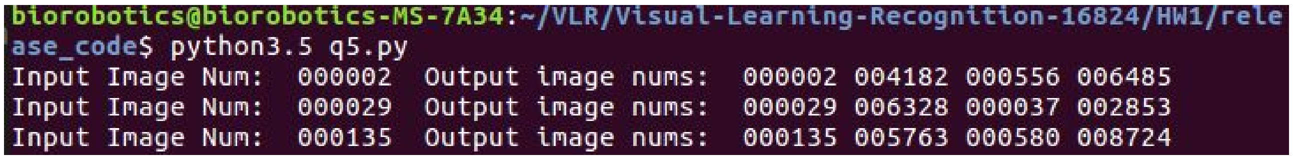
- **Testing mAP**



Task 5:

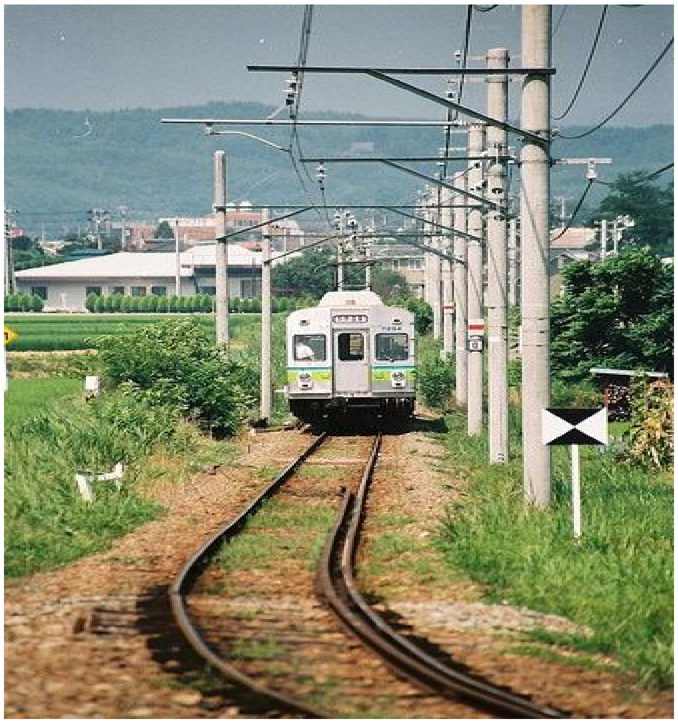
5.1. 3 nearest neighbors of 3 images from different classes

ResNet:



```
biorobotics@biorobotics-MS-7A34:~/VLR/Visual-Learning-Recognition-16824/HW1/rele
ase_code$ python3.5 q5.py
Input Image Num:  000002  Output image nums:  000002 004182 000556 006485
Input Image Num:  000029  Output image nums:  000029 006328 000037 002853
Input Image Num:  000135  Output image nums:  000135 005763 000580 008724
```

Input 1:



Outputs:

Input 2:

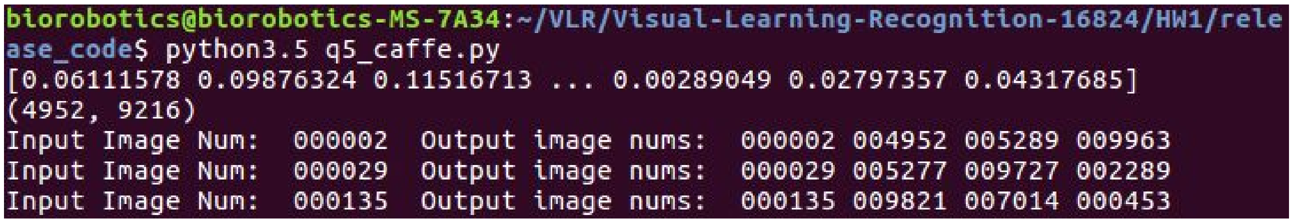
Outputs:

Input 3:

ÉCOLE de
RFECTIONNEMENT
MAGAZINE
COMPTE TOURS
TOTA
ANDROS

Outputs:

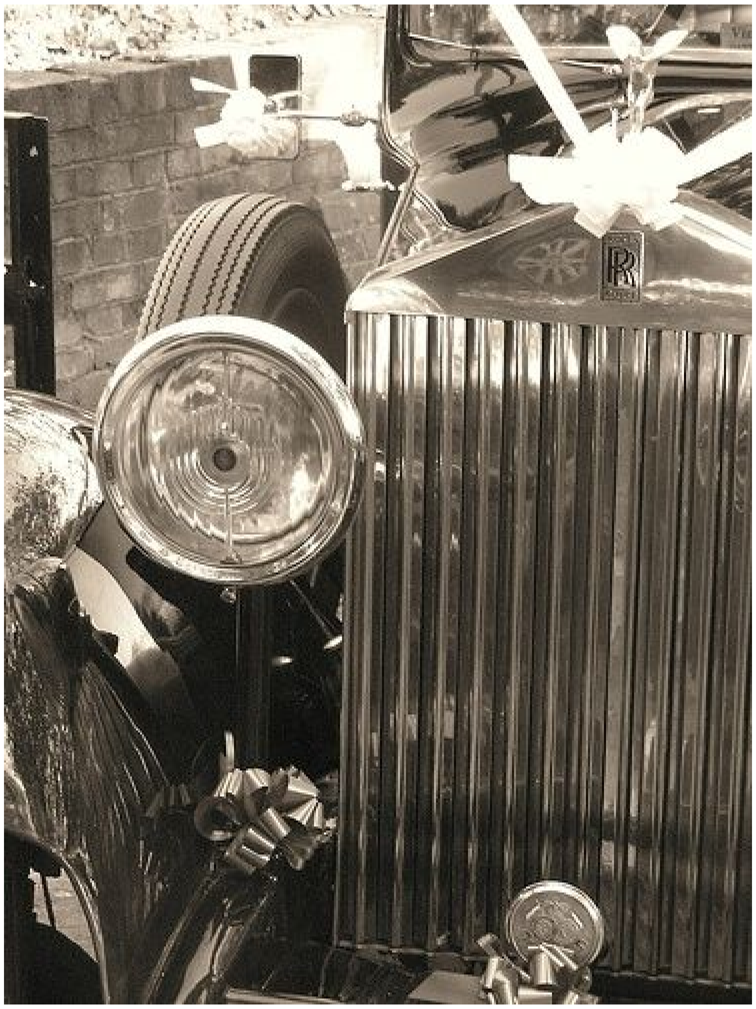
CaffeNet:



```
biorobotics@biorobotics-MS-7A34:~/VLR/Visual-Learning-Recognition-16824/HW1/rele
ase_code$ python3.5 q5_caffe.py
[0.06111578 0.09876324 0.11516713 ... 0.00289049 0.02797357 0.04317685]
(4952, 9216)
Input Image Num:  000002  Output image nums:  000002 004952 005289 009963
Input Image Num:  000029  Output image nums:  000029 005277 009727 002289
Input Image Num:  000135  Output image nums:  000135 009821 007014 000453
```

Input 1:

Outputs:



Input 2:

Outputs:

Input 3:

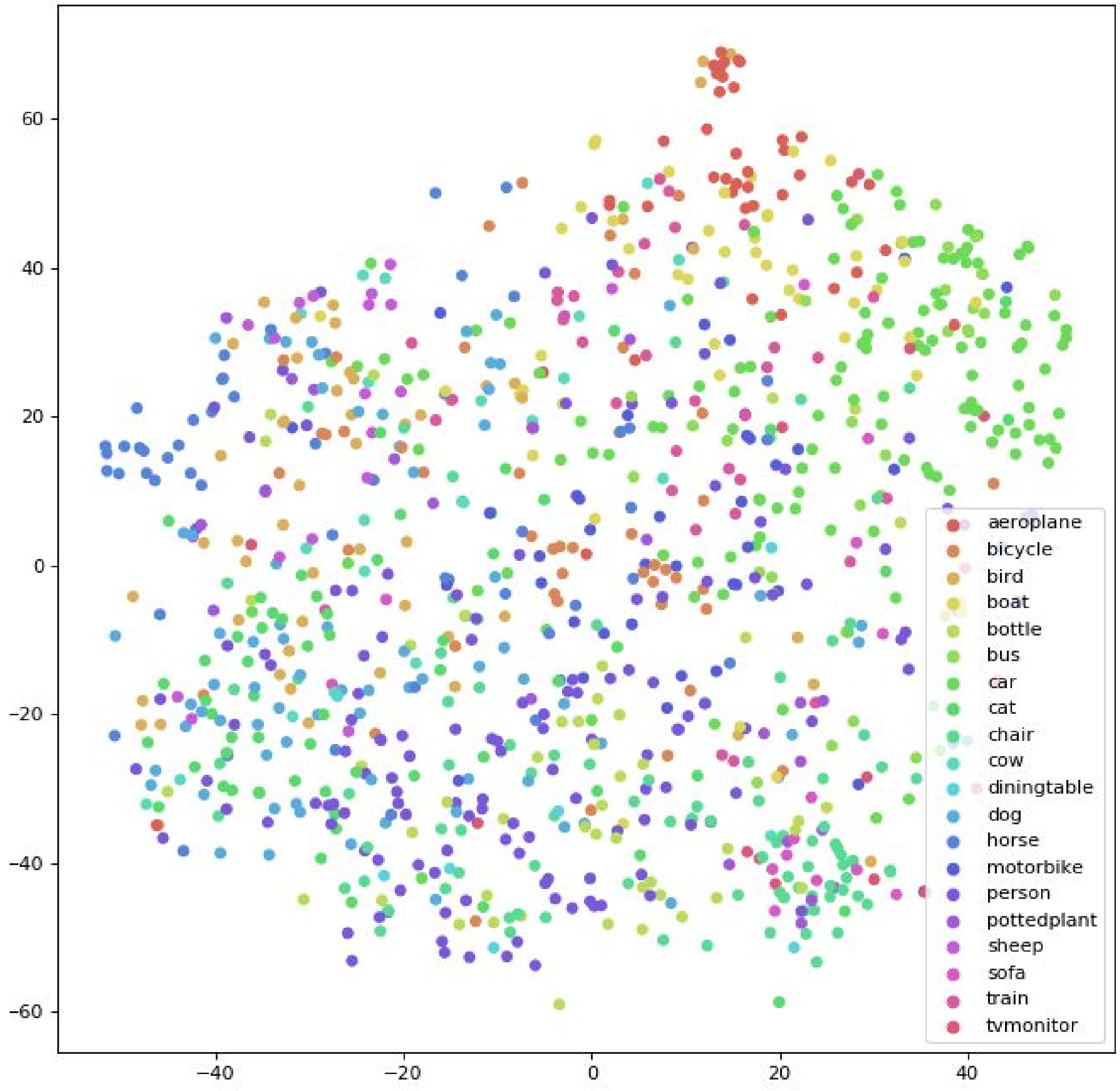
ÉCOLE de
RFECTIONNEMENT
MAGAZINE
COMPTE TOURS
TOTA
ANDROS

Outputs:

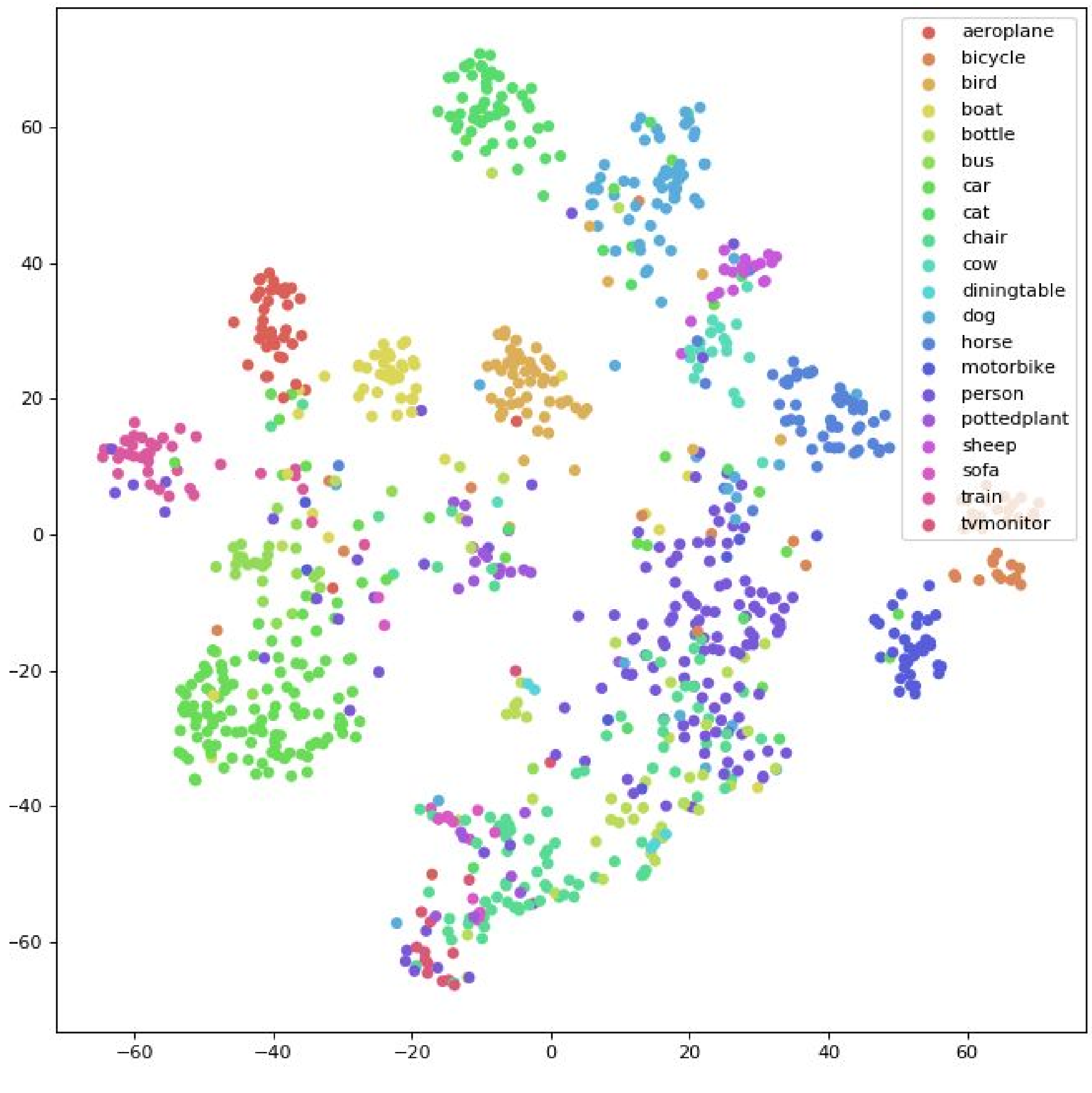
From the nearest-neighbor comparison between ResNet (pretrained on image_net) and caffe_net trained from scratch, it is evident that ResNet is benefiting greatly from the initialization from ImageNet.
All neighbors computed on the features of the ResNet model belong to the same class.
Whereas, for CaffeNet (trained from scratch), only the first 2 neighbors (including itself) belong to the same class in most cases. However, the other neighbors somehow resemble the input image and it is observable why the network mis-classifies those images - they either bear similar backgrounds, or due to their RGB information. An interesting exception happened when testing on the car (input 3). All nearest neighbors computed are from the correct category - car! This can be attributed to the high number of training examples of cars in the dataset which allowed the model to be able to tell it's features apart from the rest of the classes.

5.2. t-SNE visualization of intermediate features

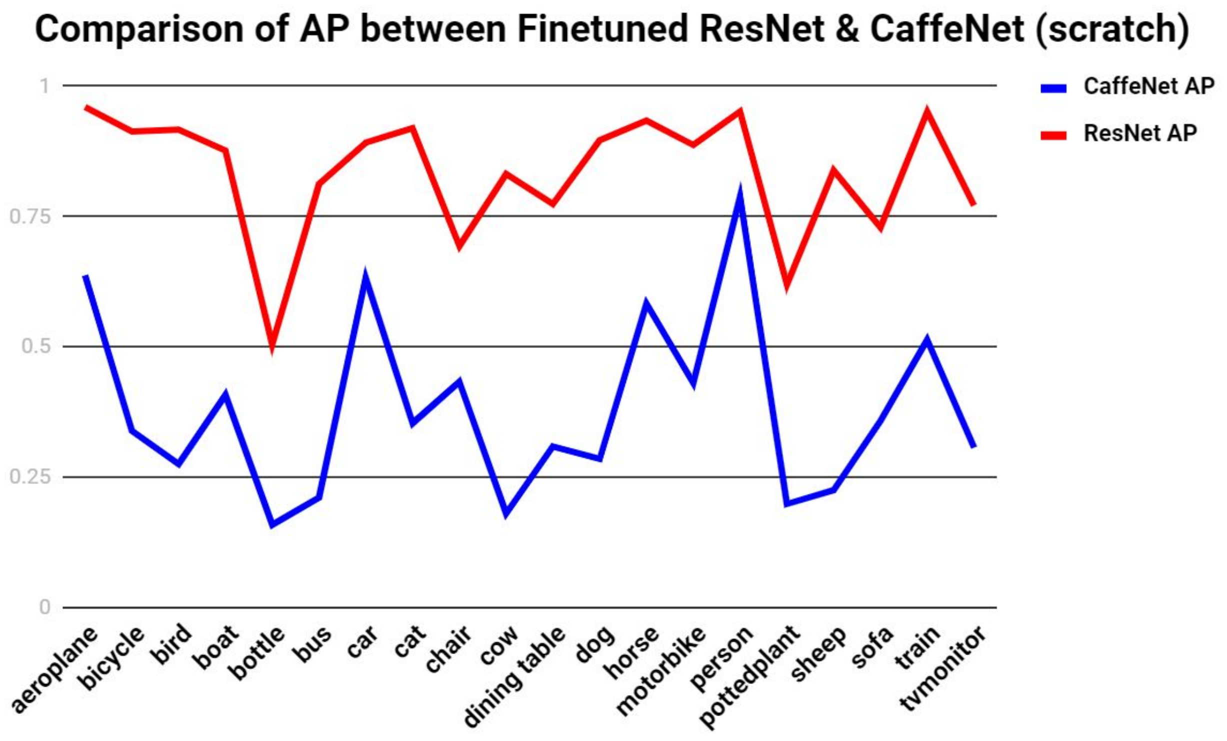
- CaffeNet



- ResNet


aeroplane
bicycle
bird
boat
bottle
bus
car
cat
chair
cow
diningtable
dog
horse
motorbike
person
pottedplant
sheep
sofa
train
tvmonitor
60
40
20
0
-20
-40
-60
−60
−40
−20
0
20
40
60

5.3. Why are some classes harder between CaffeNet and ResNet? Does pre-training (of ResNet) contribute to larger gains as opposed to CaffeNet?



From the above comparison, it is evident that ResNet finetuned has a high per-class performance than CaffeNet trained from scratch.
From the perspective of VOC Dataset, the performance on those classes is better which have higher number of examples - like person, car, train, etc.
On the other hand, classes like pottedplant, cow, sheep, have fewer examples and thus the AP on these classes is also lesser.
Due to pre-training, the ‘train’, ‘motorbike’, ‘cow’, ‘cat’, ‘horse’ classes see higher AP gains than most others. This can be attributed to the features from ImageNet dataset which would have higher number of examples from these classes. ImageNet’s pretrained weights provide a good initialization to the network which can now be finetuned on the VOC Dataset and thus we see higher gains.

--

Collaborated with: Aaditya Saraiya [asaraiya] on Q2, Q5