Task 0: Fashion MNIST classification in Pytorch (10 points)

The goal of this task is to get you familiar with Pytorch (https://pytorch.org/), teach you to debug your models, and give you a general understanding of deep learning and computer vision work-flows.

Fashion MNIST (https://github.com/zalandoresearch/fashion-mnist) is a dataset of Zalando's (https://jobs.zalando.com/tech/) article images — consisting of 70,000 grayscale images in 10 categories. Each example is a 28x28 grayscale image, associated with a label from 10 classes. 'Fashion- MNIST' is intended to serve as a direct **drop-in replacement** for the original MNIST (http://yann.lecun.com/exdb/mnist/) dataset — often used as the "Hello, World" of machine learning programs for computer vision. It shares the same image size and structure of training and testing splits. We will use 60,000 images to train the network and 10,000 images to evaluate how accurately the network learned to classify images.

In [1]:
```
# installation directions can be found on pytorch's webpage
import torch
import torch.nn as nn
import torch.nn.functional as F
from torchvision import datasets, transforms
import matplotlib.pyplot as plt
%matplotlib inline

# import our network module from simple_cnn.py
from simple_cnn import SimpleCNN              # be sure to modify or you may h
ave to restart kernel!
```

Usually you'll parse arguments using `argparse` (or similar library) but we can simply use a stand-in object for ipython notebooks. Furthermore, PyTorch can do computations on NVidia `GPU` s or on normal `CPU` s. You can configure the setting using the `device` variable.

In [2]:
```
class ARGS(object):
    # input batch size for training
    batch_size = 64
    # input batch size for testing
    test_batch_size=1000
    # number of epochs to train for
    epochs = 14
    # learning rate
    lr = 1.0
    # Learning rate step gamma
    gamma = 0.7
    # how many batches to wait before logging training status
    log_every = 100
    # how many batches to wait before evaluating model
    val_every = 100
    # set true if using GPU during training
    use_cuda = True

args = ARGS()
device = torch.device("cuda" if args.use_cuda else "cpu")
```

We define some basic testing and training code. The testing code prints out the average test loss and the training code (`main`) plots train/test losses and returns the final model.

In [3]:

```
def test(model, device, test_loader):
    """Evaluate model on test dataset."""
    model.eval()
    test_loss = 0
    correct = 0
    with torch.no_grad():
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)
            output = model(data)
            test_loss += F.cross_entropy(output, target, reduction='sum').item
()  # sum up batch loss
            pred = output.argmax(dim=1, keepdim=True)  # get the index of the
 max log-probability
            correct += pred.eq(target.view_as(pred)).sum().item()

    test_loss /= len(test_loader.dataset)
    # print("TEST ACCURACY: ", 100. * correct / len(test_loader.dataset))
    print('\nTest set: Average loss: {:.4f}, Accuracy: {}/{} ({:.0f}%)\n'.form
at(
        test_loss, correct, len(test_loader.dataset),
        100. * correct / len(test_loader.dataset)))

    return test_loss, correct / len(test_loader.dataset)


def main():
    # 1. load dataset and build dataloader
    train_loader = torch.utils.data.DataLoader(
        datasets.FashionMNIST('../data', train=True, download=True,
                       transform=transforms.Compose([
                           transforms.ToTensor(),
                           transforms.Normalize((0.1307,), (0.3081,))
                       ])),
        batch_size=args.batch_size, shuffle=True)
    test_loader = torch.utils.data.DataLoader(
        datasets.FashionMNIST('../data', train=False, transform=transforms.Com
pose([
                           transforms.ToTensor(),
                           transforms.Normalize((0.1307,), (0.3081,))
                       ])),
        batch_size=args.test_batch_size, shuffle=True)

    # 2. define the model, and optimizer.
    model = SimpleCNN().to(device)
    model.train()
    optimizer = torch.optim.Adam(model.parameters(), lr=args.lr)

    scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=1, gamma=
args.gamma)
    cnt = 0
    train_log = {'iter': [], 'loss': [], 'accuracy': []}
    test_log = {'iter': [], 'loss': [], 'accuracy': []}
    for epoch in range(args.epochs):
        for batch_idx, (data, target) in enumerate(train_loader):
            # Get a batch of data
            data, target = data.to(device), target.to(device)
```

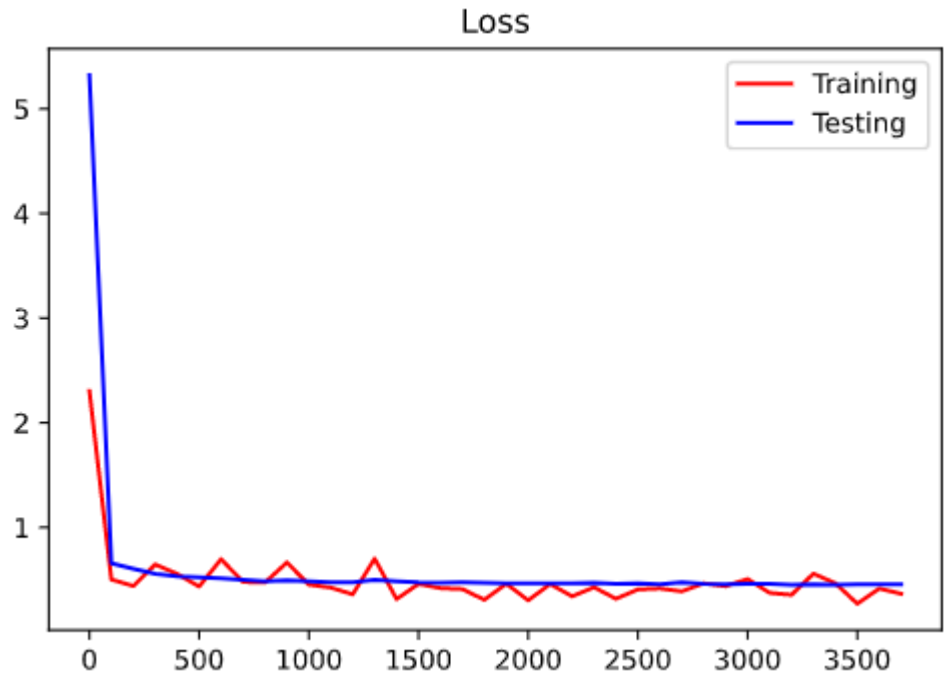
```
            optimizer.zero_grad()
            # print("Size of data epoch data input: ",data.size())
            # Forward pass
            output = model(data)
            # Calculate the loss
            loss = F.cross_entropy(output, target)
            # Calculate gradient w.r.t the loss
            loss.backward()
            # Optimizer takes one step
            optimizer.step()
            # Log info
            if cnt % args.log_every == 0:
                print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format
(
                    epoch, cnt, len(train_loader.dataset),
                           100. * batch_idx / len(train_loader), loss.item()))
                train_log['iter'].append(cnt)
                train_log['loss'].append(loss)
                # TODO: calculate your train accuracy!
                mod_pred = torch.argmax(output,dim=1)
                correct = mod_pred==target
                correct = correct.long()
                correct = sum(correct)
                train_acc = correct.float()/args.batch_size
                train_log['accuracy'].append(train_acc)
                # print("TRAINING ACCURACY: ", train_acc)
            # Validation iteration
            if cnt % args.val_every == 0:
                test_loss, test_acc = test(model, device, test_loader)
                test_log['iter'].append(cnt)
                test_log['loss'].append(test_loss)
                test_log['accuracy'].append(test_acc)
                model.train()
            cnt += 1
        scheduler.step()
    fig = plt.figure()
    plt.plot(train_log['iter'], train_log['loss'], 'r', label='Training')
    plt.plot(test_log['iter'], test_log['loss'], 'b', label='Testing')
    plt.title('Loss')
    plt.legend()
    fig = plt.figure()
    plt.plot(train_log['iter'], train_log['accuracy'], 'r', label='Training')
    plt.plot(test_log['iter'], test_log['accuracy'], 'b', label='Testing')
    plt.title('Accuracy')
    plt.legend()
    plt.show()
    return model
```

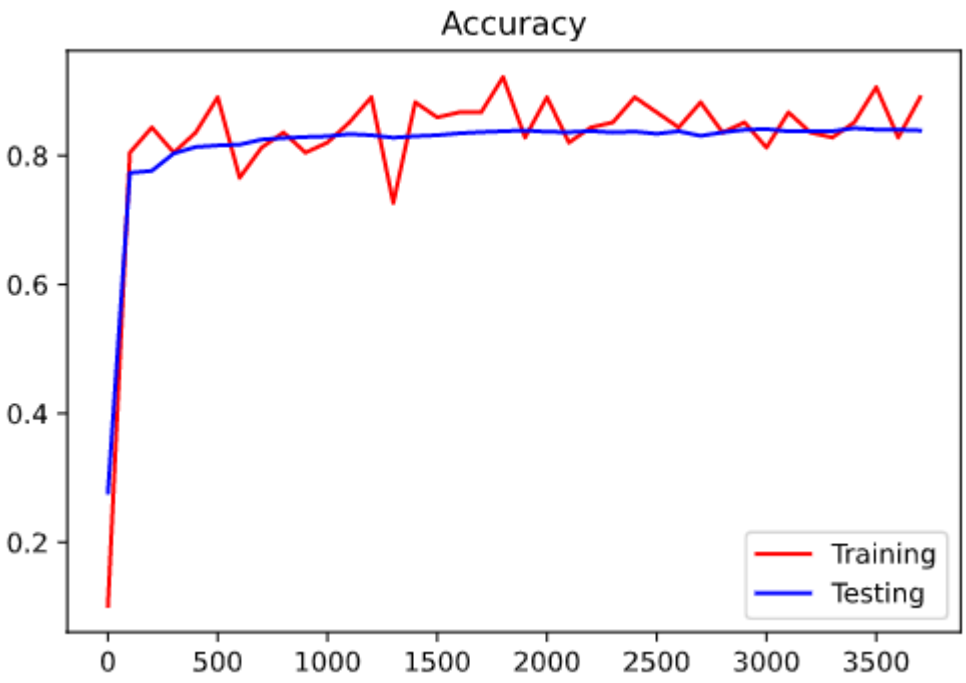
0.1 Bug Fix and Hyper-parameter search. (2pts)

Simply running `main` will result in a `RuntimeError` ! Check out `simple_cnn.py` and see if you can fix the bug. You may have to restart your ipython kernel for changes to reflect in the notebook. After that's done, be sure to fill in the TODOs in `main` .

Once you fix the bugs, you should be able to get a reasonable accuracy within 100 iterations just by tuning some hyper-parameter. Include the train/test plots of your best hyperparamter setting and comment on why you think these settings worked best. (you can complete this task on CPU)

YOUR ANSWER HERE





During the training period with the old hyper-parameters, one can notice that the accuracy of the test and training data was plateauing after a set number of epochs. As such, the number of epochs were reduced to 8. The batch size was increased to 128 to speed up the training period as well. The learning rate was significantly large so it was reduced down to 0.01. In the end, the accuracy of the training and testing sets were able to break the 80% mark.

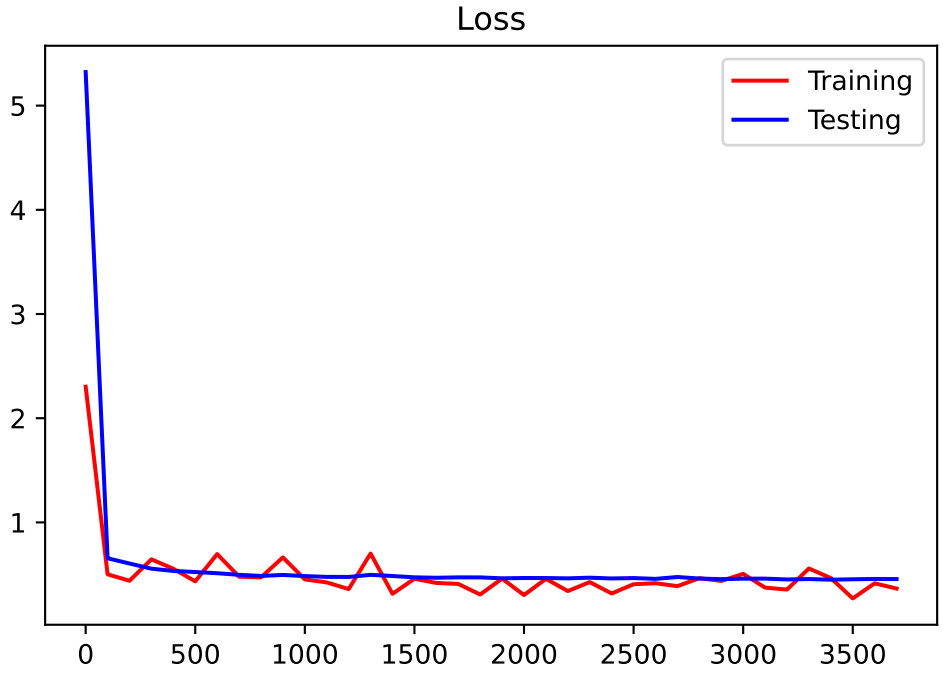
In [4]:

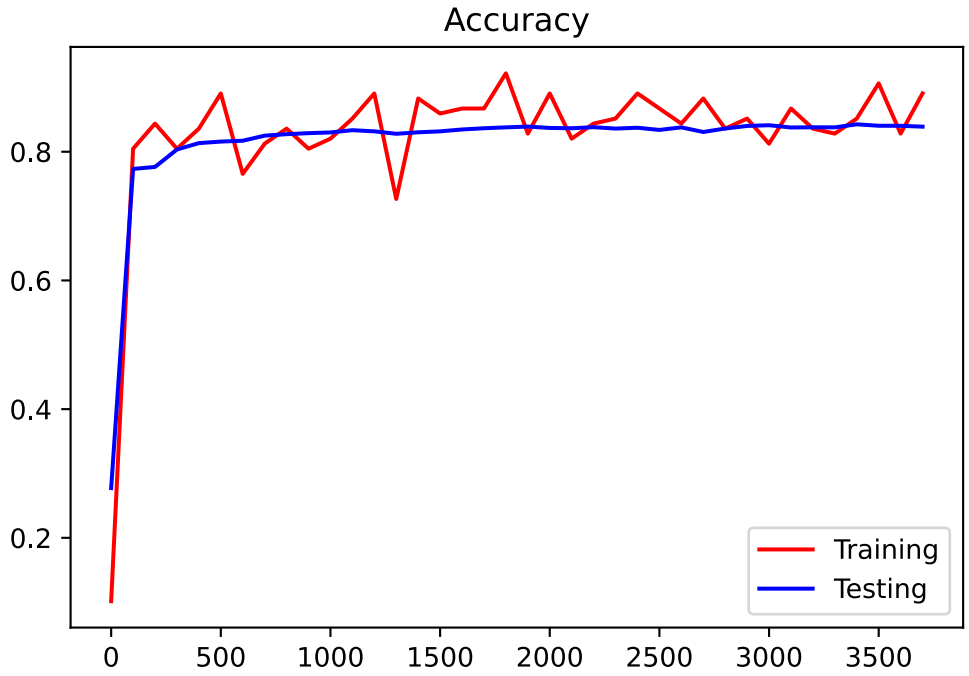
```
#### FEEL FREE TO MODIFY args VARIABLE HERE OR ABOVE ####
# args.gamma = float('inf')
args.lr = 0.01 # lowered
args.epochs = 8
args.batch_size = 128



# DON'T CHANGE
# prints out arguments and runs main
for attr in dir(args):
    if '__' not in attr and attr !='use_cuda':
        print('args.{} = {}'.format(attr, getattr(args, attr)))
print('\n\n')
model = main()
```

```
args.batch_size = 128
args.epochs = 8
args.gamma = 0.7
args.log_every = 100
args.lr = 0.01
args.test_batch_size = 1000
args.val_every = 100



Train Epoch: 0 [0/60000 (0%)]   Loss: 2.302593

Test set: Average loss: 5.3219, Accuracy: 2773/10000 (28%)

Train Epoch: 0 [100/60000 (21%)]        Loss: 0.501979

Test set: Average loss: 0.6558, Accuracy: 7733/10000 (77%)

Train Epoch: 0 [200/60000 (43%)]        Loss: 0.439888

Test set: Average loss: 0.6055, Accuracy: 7764/10000 (78%)

Train Epoch: 0 [300/60000 (64%)]        Loss: 0.645565

Test set: Average loss: 0.5559, Accuracy: 8035/10000 (80%)

Train Epoch: 0 [400/60000 (85%)]        Loss: 0.555263

Test set: Average loss: 0.5344, Accuracy: 8134/10000 (81%)

Train Epoch: 1 [500/60000 (7%)] Loss: 0.434446

Test set: Average loss: 0.5238, Accuracy: 8158/10000 (82%)

Train Epoch: 1 [600/60000 (28%)]        Loss: 0.696253

Test set: Average loss: 0.5123, Accuracy: 8171/10000 (82%)

Train Epoch: 1 [700/60000 (49%)]        Loss: 0.478521

Test set: Average loss: 0.4971, Accuracy: 8250/10000 (82%)

Train Epoch: 1 [800/60000 (71%)]        Loss: 0.472464

Test set: Average loss: 0.4861, Accuracy: 8273/10000 (83%)

Train Epoch: 1 [900/60000 (92%)]        Loss: 0.664430

Test set: Average loss: 0.4959, Accuracy: 8289/10000 (83%)

Train Epoch: 2 [1000/60000 (13%)]       Loss: 0.452996

Test set: Average loss: 0.4847, Accuracy: 8300/10000 (83%)

Train Epoch: 2 [1100/60000 (35%)]       Loss: 0.424224

Test set: Average loss: 0.4772, Accuracy: 8335/10000 (83%)
```

```
Train Epoch: 2 [1200/60000 (56%)]       Loss: 0.360117

Test set: Average loss: 0.4767, Accuracy: 8317/10000 (83%)

Train Epoch: 2 [1300/60000 (77%)]       Loss: 0.701172

Test set: Average loss: 0.4968, Accuracy: 8279/10000 (83%)

Train Epoch: 2 [1400/60000 (99%)]       Loss: 0.314604

Test set: Average loss: 0.4855, Accuracy: 8302/10000 (83%)

Train Epoch: 3 [1500/60000 (20%)]       Loss: 0.459535

Test set: Average loss: 0.4729, Accuracy: 8317/10000 (83%)

Train Epoch: 3 [1600/60000 (41%)]       Loss: 0.419015

Test set: Average loss: 0.4688, Accuracy: 8346/10000 (83%)

Train Epoch: 3 [1700/60000 (62%)]       Loss: 0.409575

Test set: Average loss: 0.4726, Accuracy: 8365/10000 (84%)

Train Epoch: 3 [1800/60000 (84%)]       Loss: 0.308018

Test set: Average loss: 0.4721, Accuracy: 8379/10000 (84%)

Train Epoch: 4 [1900/60000 (5%)]        Loss: 0.460715

Test set: Average loss: 0.4635, Accuracy: 8391/10000 (84%)

Train Epoch: 4 [2000/60000 (26%)]       Loss: 0.302965

Test set: Average loss: 0.4663, Accuracy: 8370/10000 (84%)

Train Epoch: 4 [2100/60000 (48%)]       Loss: 0.458157

Test set: Average loss: 0.4661, Accuracy: 8365/10000 (84%)

Train Epoch: 4 [2200/60000 (69%)]       Loss: 0.341236

Test set: Average loss: 0.4631, Accuracy: 8382/10000 (84%)

Train Epoch: 4 [2300/60000 (90%)]       Loss: 0.427276

Test set: Average loss: 0.4694, Accuracy: 8360/10000 (84%)

Train Epoch: 5 [2400/60000 (12%)]       Loss: 0.317789

Test set: Average loss: 0.4621, Accuracy: 8373/10000 (84%)

Train Epoch: 5 [2500/60000 (33%)]       Loss: 0.406668

Test set: Average loss: 0.4658, Accuracy: 8339/10000 (83%)
```

```
Train Epoch: 5 [2600/60000 (54%)]       Loss: 0.416200

Test set: Average loss: 0.4581, Accuracy: 8379/10000 (84%)

Train Epoch: 5 [2700/60000 (76%)]       Loss: 0.388463

Test set: Average loss: 0.4759, Accuracy: 8307/10000 (83%)

Train Epoch: 5 [2800/60000 (97%)]       Loss: 0.464706

Test set: Average loss: 0.4614, Accuracy: 8361/10000 (84%)

Train Epoch: 6 [2900/60000 (18%)]       Loss: 0.438095

Test set: Average loss: 0.4554, Accuracy: 8401/10000 (84%)

Train Epoch: 6 [3000/60000 (40%)]       Loss: 0.506237

Test set: Average loss: 0.4599, Accuracy: 8412/10000 (84%)

Train Epoch: 6 [3100/60000 (61%)]       Loss: 0.374118

Test set: Average loss: 0.4604, Accuracy: 8377/10000 (84%)

Train Epoch: 6 [3200/60000 (82%)]       Loss: 0.355285

Test set: Average loss: 0.4524, Accuracy: 8380/10000 (84%)

Train Epoch: 7 [3300/60000 (4%)]        Loss: 0.557764

Test set: Average loss: 0.4565, Accuracy: 8379/10000 (84%)

Train Epoch: 7 [3400/60000 (25%)]       Loss: 0.465712

Test set: Average loss: 0.4502, Accuracy: 8426/10000 (84%)

Train Epoch: 7 [3500/60000 (46%)]       Loss: 0.270384

Test set: Average loss: 0.4542, Accuracy: 8404/10000 (84%)

Train Epoch: 7 [3600/60000 (68%)]       Loss: 0.415405

Test set: Average loss: 0.4572, Accuracy: 8403/10000 (84%)

Train Epoch: 7 [3700/60000 (89%)]       Loss: 0.365243

Test set: Average loss: 0.4563, Accuracy: 8390/10000 (84%)
```





Play with parameters.(3pt)

How many trainable parameters does the trained model have?

Model has 454922 params

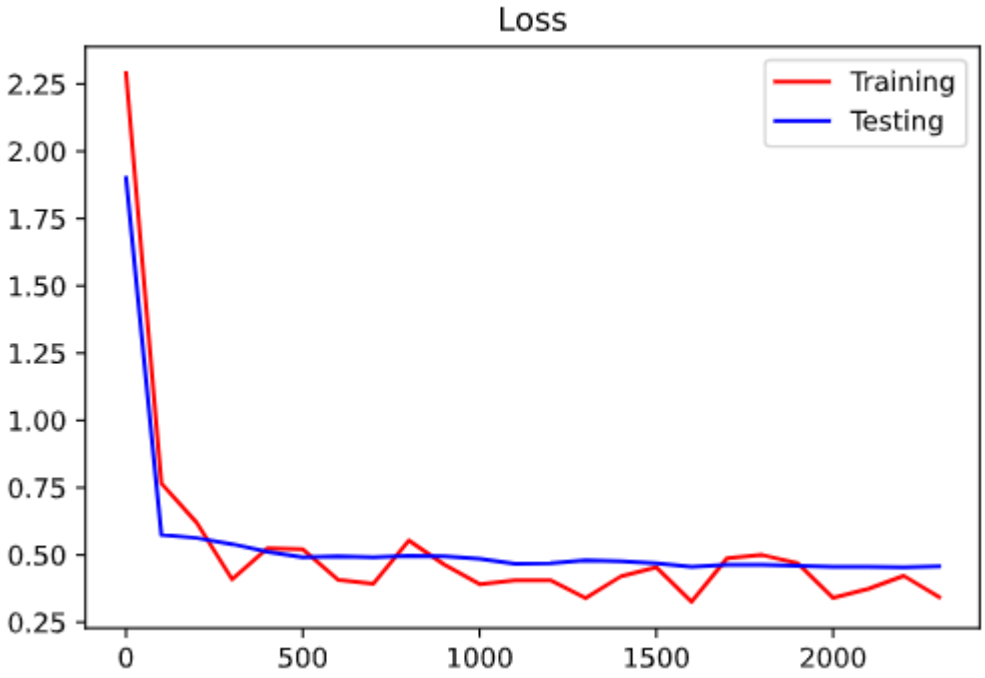
In [7]:
```
def param_count(model):
    try:
        numParam = sum(iterVal.numel() for iterVal in model.parameters() if it
erVal.requires_grad)
        return numParam
    except:
        raise NotImplementedError
print('Model has {} params'.format(param_count(model)))
```

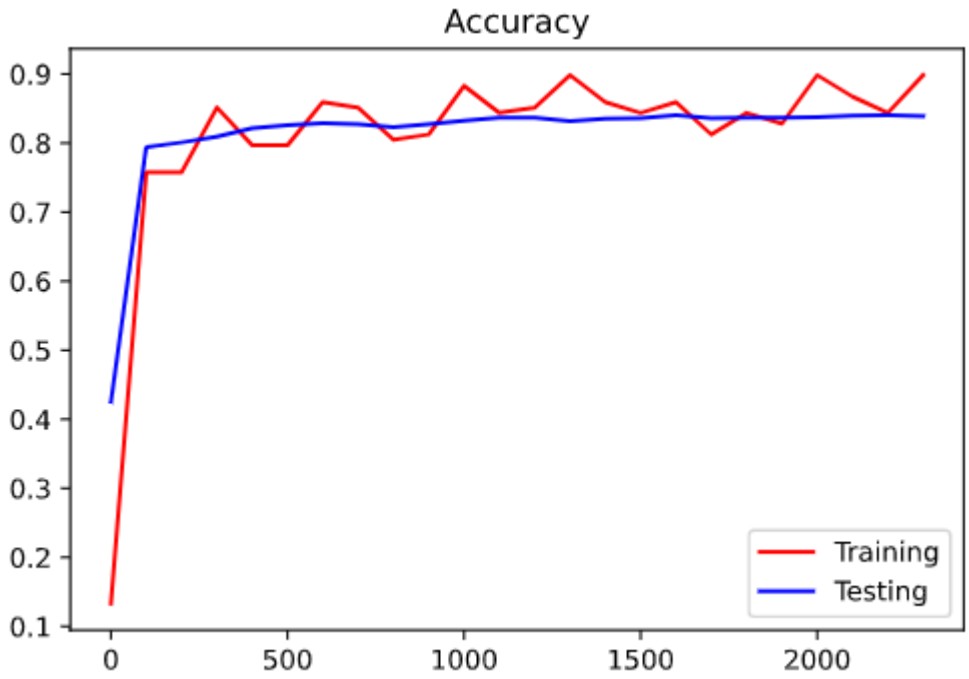
```
Model has 454922 params
```

Deep Linear Networks?!? (5pt)

Until this point, there are no non-linearities in the SimpleCNN! (Your TAs were just as surprised as you are at the results.) Your next task is to modify the code to add non-linear activation layers, and train your model in full scale. Make sure to add non-linearities at **every** applicable layer.

Compute the loss and accuracy curves on train and test sets after 5 epochs.

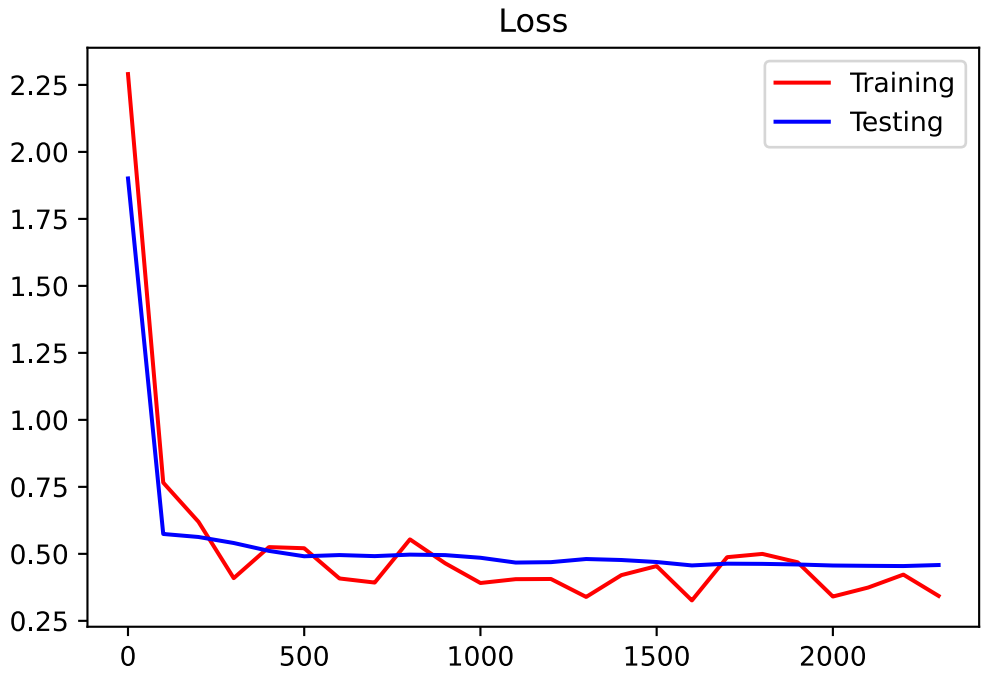


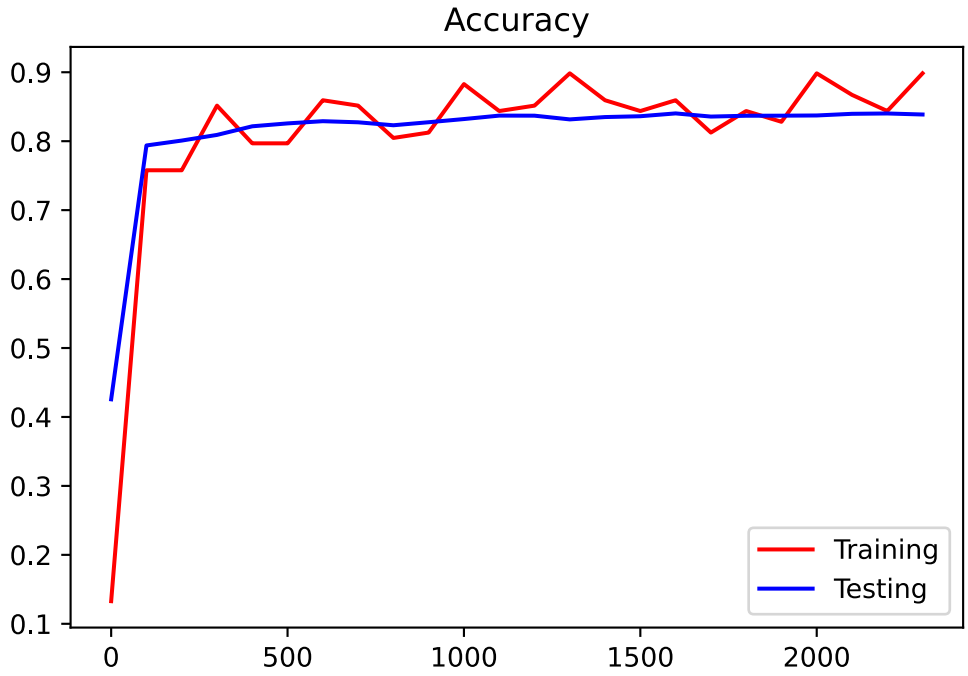


In [8]:
```
args.epochs = 5
args.lr = 1e-3
main()
```

```
Train Epoch: 0 [0/60000 (0%)]   Loss: 2.290475

Test set: Average loss: 1.9010, Accuracy: 4256/10000 (43%)

Train Epoch: 0 [100/60000 (21%)]        Loss: 0.765751

Test set: Average loss: 0.5742, Accuracy: 7938/10000 (79%)

Train Epoch: 0 [200/60000 (43%)]        Loss: 0.619610

Test set: Average loss: 0.5629, Accuracy: 8009/10000 (80%)

Train Epoch: 0 [300/60000 (64%)]        Loss: 0.409301

Test set: Average loss: 0.5404, Accuracy: 8091/10000 (81%)

Train Epoch: 0 [400/60000 (85%)]        Loss: 0.525194

Test set: Average loss: 0.5110, Accuracy: 8216/10000 (82%)

Train Epoch: 1 [500/60000 (7%)] Loss: 0.520956

Test set: Average loss: 0.4909, Accuracy: 8258/10000 (83%)

Train Epoch: 1 [600/60000 (28%)]        Loss: 0.408131

Test set: Average loss: 0.4955, Accuracy: 8290/10000 (83%)

Train Epoch: 1 [700/60000 (49%)]        Loss: 0.392827

Test set: Average loss: 0.4915, Accuracy: 8274/10000 (83%)

Train Epoch: 1 [800/60000 (71%)]        Loss: 0.554026

Test set: Average loss: 0.4971, Accuracy: 8231/10000 (82%)

Train Epoch: 1 [900/60000 (92%)]        Loss: 0.464746

Test set: Average loss: 0.4953, Accuracy: 8275/10000 (83%)

Train Epoch: 2 [1000/60000 (13%)]       Loss: 0.391170

Test set: Average loss: 0.4854, Accuracy: 8321/10000 (83%)

Train Epoch: 2 [1100/60000 (35%)]       Loss: 0.405492

Test set: Average loss: 0.4675, Accuracy: 8371/10000 (84%)

Train Epoch: 2 [1200/60000 (56%)]       Loss: 0.406192

Test set: Average loss: 0.4690, Accuracy: 8370/10000 (84%)

Train Epoch: 2 [1300/60000 (77%)]       Loss: 0.339169

Test set: Average loss: 0.4807, Accuracy: 8316/10000 (83%)

Train Epoch: 2 [1400/60000 (99%)]       Loss: 0.420615
```

```
Test set: Average loss: 0.4770, Accuracy: 8350/10000 (84%)

Train Epoch: 3 [1500/60000 (20%)]       Loss: 0.454342

Test set: Average loss: 0.4695, Accuracy: 8362/10000 (84%)

Train Epoch: 3 [1600/60000 (41%)]       Loss: 0.326145

Test set: Average loss: 0.4568, Accuracy: 8403/10000 (84%)

Train Epoch: 3 [1700/60000 (62%)]       Loss: 0.487575

Test set: Average loss: 0.4633, Accuracy: 8357/10000 (84%)

Train Epoch: 3 [1800/60000 (84%)]       Loss: 0.499794

Test set: Average loss: 0.4626, Accuracy: 8369/10000 (84%)

Train Epoch: 4 [1900/60000 (5%)]        Loss: 0.468573

Test set: Average loss: 0.4603, Accuracy: 8370/10000 (84%)

Train Epoch: 4 [2000/60000 (26%)]       Loss: 0.340706

Test set: Average loss: 0.4565, Accuracy: 8373/10000 (84%)

Train Epoch: 4 [2100/60000 (48%)]       Loss: 0.374197

Test set: Average loss: 0.4553, Accuracy: 8397/10000 (84%)

Train Epoch: 4 [2200/60000 (69%)]       Loss: 0.422526

Test set: Average loss: 0.4544, Accuracy: 8402/10000 (84%)

Train Epoch: 4 [2300/60000 (90%)]       Loss: 0.342963

Test set: Average loss: 0.4583, Accuracy: 8387/10000 (84%)
```





Out[8]:
```
SimpleCNN(
  (conv1): Conv2d(1, 32, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
  (conv2): Conv2d(32, 64, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
  (pool1): AvgPool2d(kernel_size=2, stride=2, padding=0)
  (pool2): AvgPool2d(kernel_size=2, stride=2, padding=0)
  (fc1): Sequential(
    (0): Linear(in_features=3136, out_features=128, bias=True)
  )
  (fc2): Sequential(
    (0): Linear(in_features=128, out_features=10, bias=True)
  )
)
```

Where did you add your non-linearities?

YOUR ANSWER HERE

I added ReLU activation layers after each of the two convolution layers. I also added a ReLU activation after the first fully-connected layer. I did not apply a softmax at the second fully-connected layer because the cross_entropy loss function defined in main() contains the softmax function.

Provide some insights on why the results was fairly good even without activation layers. (**2** pts)

YOUR ANSWER HERE

The contents of the objects in the images of the dataset were presented in a standardized way. There was no occulsion or varying configurations of most of the objects in each class. For this reason, a model can generalize the dataset with good performance throught the use of linear filters and logistic regression lines. The classification for shoes, for examples, can be adequately characterized through a set of logistic regression lines along the edges of the shoe. The same thing may be said for jackets and pants. Differentiations between classes may be significant enough to be separated via logistic regression rather than a nonlinear function.

Another reason may be that the images of the same classification reside within the same sections of the frames compared to other class images. For example, trouser images usually occupy a rectangular area around the center of the image frame. Ankle boots usually occupy the lower-right diagonal half of the image. Given a linear network model with no activation functions, straight regression lines can be used to encompass common firing zones for certain classes. From this thought, nonlinear activations are not needed for the success of the model.

Q1: Simple CNN network for PASCAL multi-label classification (20 points)

Now let's try to recognize some natural images. We provided some starter code for this task. The following steps will guide you through the process.

1.1 Setup the dataset

We start by modifying the code to read images from the PASCAL 2007 dataset. The important thing to note is that PASCAL can have multiple objects present in the same image. Hence, this is a multi-label classification problem, and will have to be tackled slightly differently.

First, download the data. `cd` to a location where you can store 0.5GB of images. Then run:

```
wget http://host.robots.ox.ac.uk/pascal/VOC/voc2007/VOCtrainval_06-Nov-2007.tar
tar -xf VOCtrainval_06-Nov-2007.tar

wget http://host.robots.ox.ac.uk/pascal/VOC/voc2007/VOCtest_06-Nov-2007.tar
tar -xf VOCtest_06-Nov-2007.tar
cd VOCdevkit/VOC2007/
```

1.2 Write a dataloader with data augmentation (5 pts)

Dataloader The first step is to write a pytorch data loader (https://pytorch.org/tutorials/beginner/data_loading_tutorial.html) which loads this PASCAL data. Complete the functions `preload_anno` and `__getitem__` in `voc_dataset.py` .

- **Hint**: Refer to the `README` in `VOCdevkit` to understand the structure and labeling.
- **Hint** : As the function docstring says, `__getitem__` takes as input the index, and returns a tuple - `(image, label, weight)` . The labels should be 1s for each object that is present in the image, and weights should be 1 for each label in the image, except those labeled as ambiguous (use the `difficult` attribute). All other values should be 0. For simplicity, resize all images to a canonical size.)

Data Augmentation Modify `__getitem__` to randomly *augment* each datapoint. Please describe what data augmentation you implement.

- **Hint**: Since we are training a model from scratch on this small dataset, it is important to perform basic data augmentation to avoid overfitting. Add random crops and left-right flips when training, and do a center crop when testing, etc. As for natural images, another common practice is to subtract the mean values of RGB images from ImageNet dataset. The mean values for RGB images are: `[123.68, 116.78, 103.94]` – sometimes, rescaling to `[-1, 1]` suffices.

Note: You should use data in 'trainval' for training and 'test' for testing, since PASCAL is a small dataset.

DESCRIBE YOUR AUGMENTATION PIPELINE HERE**

Train Augmentations:

The following augmentations were performed on the training set:

- Rescaling to 28 x 28
- Random horizontal flipping
- Normalization based on a mean of (0.485, 0.456, 0.406) and a standard deviation of (0.229, 0.224, 0.225)

Test Augmentations:

The following augmentations were performed on the test set:

- Rescaling to 28 x 28
- Random horizontal flipping
- Normalization based on a mean of (0.485, 0.456, 0.406) and a standard deviation of (0.229, 0.224, 0.225)

1.3 Measure Performance (5 pts)

To evaluate the trained model, we will use a standard metric for multi-label evaluation - mean average precision (mAP) (http://scikit-learn.org/stable/modules/generated/sklearn.metrics.average_precision_score.html). Please implement `eval_dataset_map` in `utils.py` - this function will evaluate a model's map score using a given dataset object. You will need to make predictions on the given dataset with the model and call `compute_ap` to get average precision.

Please describe how to compute AP for each class(not mAP). **YOUR ANSWER HERE**

The average precision (AP) is computed by gettings the ground truth, non-difficult classes and the correct predition classes. The predicition class get a small decimal subtracted from it (perhaps for normalization). Then an average precision score is calculated between the two vectors and given a value for each class and stacked on top of each of as a vector. At the end of the function, the AP vector is returned out.

1.4 Let's Start Training! (5 pts)

Write the code for training and testing for multi-label classification in `trainer.py` . To start, you'll use the same model you used for Fashion MNIST (bad idea, but let's give it a shot).

Initialize a fresh model and optimizer. Then run your training code for 5 epochs and print the mAP on test set.

In [1]:
```
import torch
import trainer
from utils import ARGS
from simple_cnn import SimpleCNN
from voc_dataset import VOCDataset

# create hyperparameter argument class
args = ARGS(epochs=5)
print(args)
```

```
args.batch_size = 32
args.device = cuda
args.epochs = 5
args.gamma = 0.7
args.log_every = 100
args.lr = 1.0
args.save_at_end = False
args.save_freq = 10
args.test_batch_size = 1000
args.val_every = 100
```

In [ ]:
```
# initializes (your) naiive model
model = SimpleCNN(num_classes=len(VOCDataset.CLASS_NAMES), inp_size=64, c_dim=
3)
# initializes Adam optimizer and simple StepLR scheduler
optimizer = torch.optim.Adam(model.parameters(), lr=args.lr)
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=1, gamma=args
.gamma)
# trains model using your training code and reports test map
test_ap, test_map = trainer.train(args, model, optimizer, scheduler)
print('test map:', test_map)
```

TensorBoard (https://www.tensorflow.org/guide/summaries_and_tensorboard) is an awesome visualization tool. It was firstly integrated in TensorFlow (https://www.tensorflow.org/) (~possibly the only useful tool TensorFlow provides~). It can be used to visualize training losses, network weights and other parameters.

To use TensorBoard in Pytorch, there are two options: TensorBoard in Pytorch (https://pytorch.org/docs/stable/tensorboard.html) (for Pytorch >= 1.1.0) or TensorBoardX (https://github.com/lanpa/tensorboardX) - a third party library. Add code in `trainer.py` to visualize the testing MAP and training loss in Tensorboard. *You may have to reload the kernel for these changes to take effect*

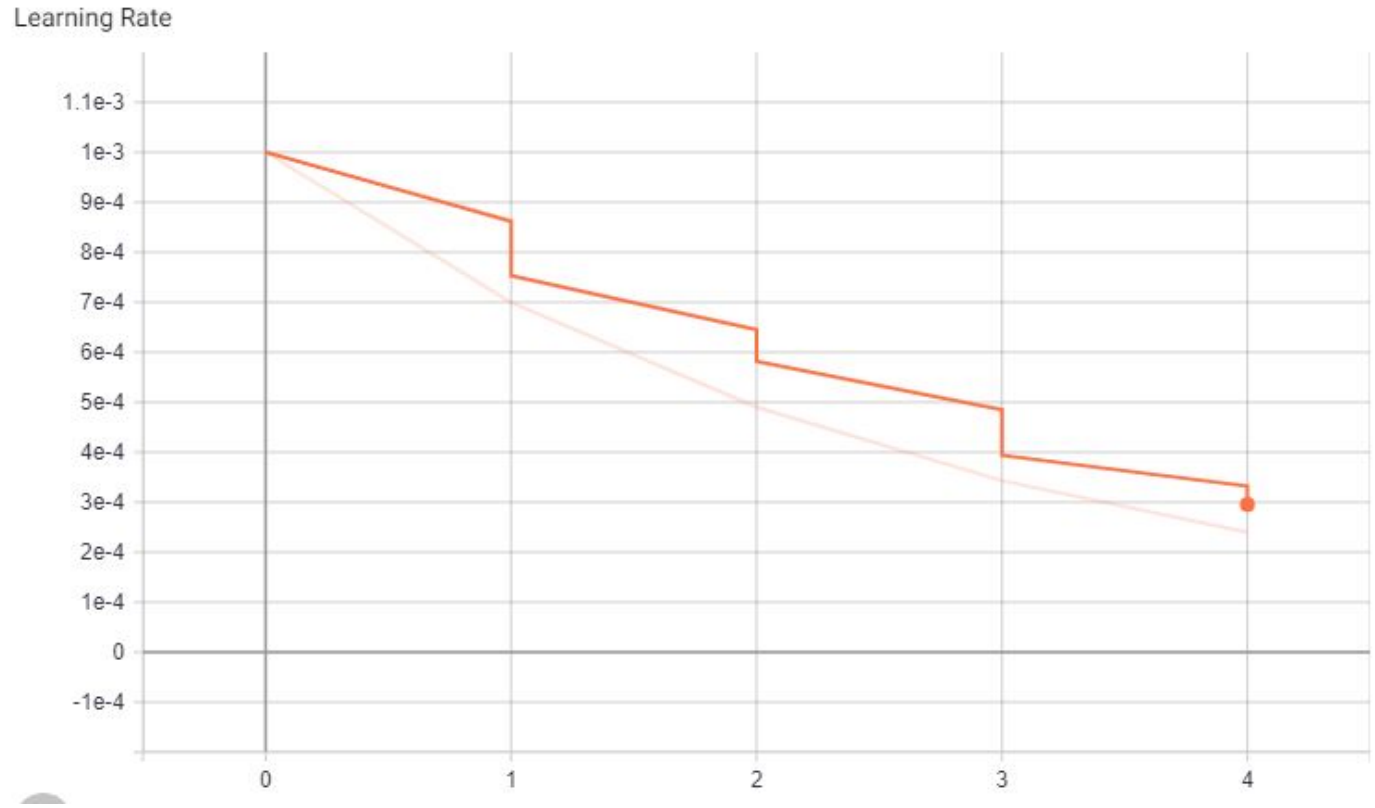
Show clear screenshots of the learning curves of testing MAP and training loss for 5 epochs (batch size=20, learning rate=0.001). Please evaluate your model to calculate the MAP on the testing dataset every 100 iterations.

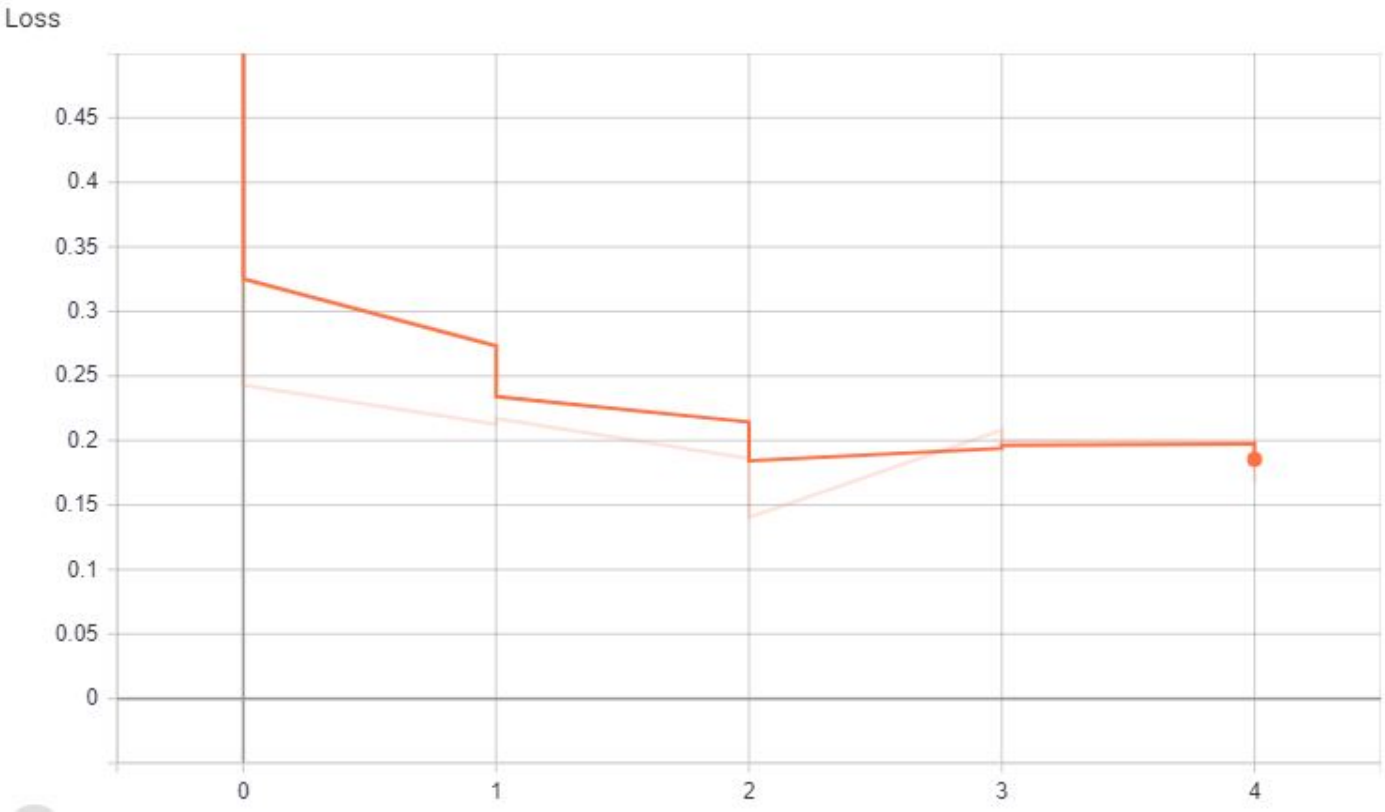
In [2]:
```
args = ARGS(epochs=5, batch_size=20, lr=0.001)
model = SimpleCNN(num_classes=len(VOCDataset.CLASS_NAMES), inp_size=64, c_dim=
3)
optimizer = torch.optim.Adam(model.parameters(), lr=args.lr)
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=1, gamma=args
.gamma)
if __name__ == "__main__":
    test_ap, test_map = trainer.train(args, model, optimizer, scheduler)
    print('test map:', test_map)
```

```
Train Epoch: 0 [0 (0%)] Loss: 0.694451 | mAP: 0.076841
Train Epoch: 0 [100 (40%)]      Loss: 0.240711 | mAP: 0.123666
Train Epoch: 0 [200 (80%)]      Loss: 0.242937 | mAP: 0.163159
Train Epoch: 1 [300 (20%)]      Loss: 0.212498 | mAP: 0.173409
Train Epoch: 1 [400 (59%)]      Loss: 0.211122 | mAP: 0.193135
Train Epoch: 1 [500 (99%)]      Loss: 0.216955 | mAP: 0.204963
Train Epoch: 2 [600 (39%)]      Loss: 0.186005 | mAP: 0.218065
Train Epoch: 2 [700 (79%)]      Loss: 0.140446 | mAP: 0.224668
Train Epoch: 3 [800 (19%)]      Loss: 0.208131 | mAP: 0.232051
Train Epoch: 3 [900 (59%)]      Loss: 0.194046 | mAP: 0.234823
Train Epoch: 3 [1000 (98%)]     Loss: 0.199973 | mAP: 0.238088
Train Epoch: 4 [1100 (38%)]     Loss: 0.199175 | mAP: 0.241002
Train Epoch: 4 [1200 (78%)]     Loss: 0.167286 | mAP: 0.246116
test map: 0.24888071523104266
```

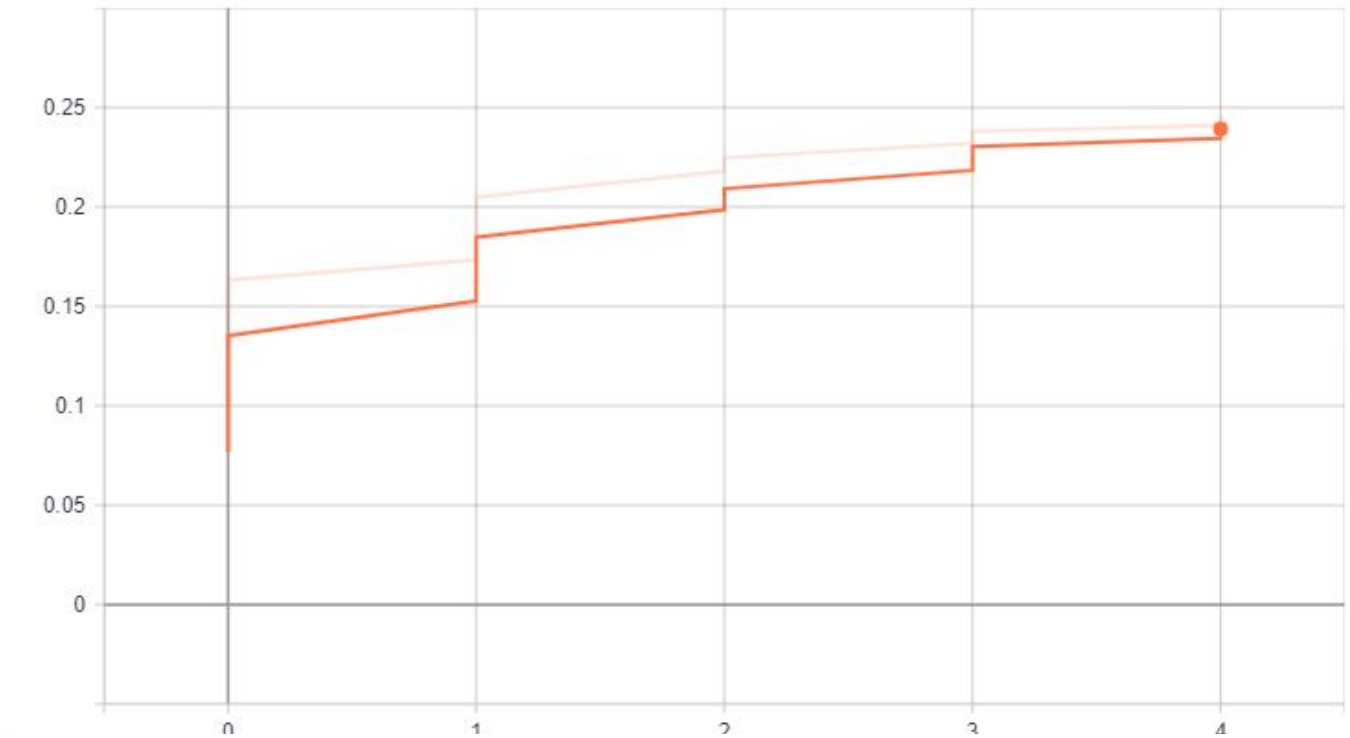
INSERT YOUR TENSORBOARD SCREENSHOTS HERE

The figures below display the learning rate, training loss, and training map, respectively:





mAP



In [ ]:

Q2: Lets go deeper! CaffeNet for PASCAL classification (20 pts)

Note: You are encouraged to reuse code from the previous task. Finish Q1 if you haven't already!

As you might have seen, the performance of the SimpleCNN model was pretty low for PASCAL. This is expected as PASCAL is much more complex than FASHION MNIST, and we need a much beefier model to handle it.

In this task we will be constructing a variant of the AlexNet (https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf) architecture, known as CaffeNet. If you are familiar with Caffe, a prototxt of the network is available here (https://github.com/BVLC/caffe/blob/master/models/bvlc_reference_caffenet/train_val.prototxt). A visualization of the network is available here (http://ethereon.github.io/netscope/#/preset/caffenet).

2.1 Build CaffeNet (5 pts)

Here is the exact model we want to build. In this task, `torchvision.models.xxx()` is NOT allowed. Define your own CaffeNet! We use the following operator notation for the architecture:

1. Convolution: A convolution with kernel size k, stride s, output channels n, padding p is represented as $conv(k, s, n, p)$.
2. Max Pooling: A max pool operation with kernel size k, stride s as $maxpool(k, s)$.
3. Fully connected: For n output units, $FC(n)$.
4. ReLU: For rectified linear non-linearity $relu()$

```
ARCHITECTURE:
-> image
-> conv(11, 4, 96, ’VALID’)
-> relu()
-> max_pool(3, 2)
-> conv(5, 1, 256, 'SAME')
-> relu()
-> max_pool(3, 2)
-> conv(3, 1, 384, 'SAME')
-> relu()
-> conv(3, 1, 384, 'SAME')
-> relu()
-> conv(3, 1, 256, ’SAME’)
-> relu()
-> max_pool(3, 2)
-> flatten()
-> fully_connected(4096)
-> relu()
-> dropout(0.5)
-> fully_connected(4096)
-> relu()
-> dropout(0.5)
-> fully_connected(20)
```

In [ ]:

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import matplotlib.pyplot as plt
# %matplotlib inline

import trainer
from utils import ARGS
from simple_cnn import SimpleCNN
from voc_dataset import VOCDataset


def get_fc(inp_dim, out_dim, non_linear='relu'):
    """
    Mid-level API. It is useful to customize your own for large code repo.
    :param inp_dim: int, intput dimension
    :param out_dim: int, output dimension
    :param non_linear: str, 'relu', 'softmax'
    :return: list of layers [FC(inp_dim, out_dim), (non linear layer)]
    """
    layers = []
    layers.append(nn.Linear(inp_dim, out_dim))
    if non_linear == 'relu':
        layers.append(nn.ReLU())
    elif non_linear == 'softmax':
        layers.append(nn.Softmax(dim=1))
    elif non_linear == 'none':
        pass
    else:
        raise NotImplementedError
    return layers

class CaffeNet(nn.Module):
    def __init__(self):
        super().__init__()
        c_dim = 3
        self.conv1 = nn.Conv2d(c_dim,96,11,4,padding=0) # valid padding
        self.pool1 = nn.MaxPool2d(3,2)
        self.conv2 = nn.Conv2d(96, 256, 5,padding=2) # same padding
        self.pool2 = nn.MaxPool2d(3,2)
        self.conv3 = nn.Conv2d(256,384,3,padding=1) # same padding
        self.conv4 = nn.Conv2d(384,384,3,padding=1) # same padding
        self.conv5 = nn.Conv2d(384,256,3,padding=1) # same padding
        self.pool3 = nn.MaxPool2d(3,2)
        self.flat_dim = 5*5*256 # replace with the actual value
        self.fc1 = nn.Sequential(*get_fc(self.flat_dim, 4096, 'relu'))
        self.dropout1 = nn.Dropout(p=0.5)
        self.fc2 = nn.Sequential(*get_fc(4096, 4096, 'relu'))
        self.dropout2 = nn.Dropout(p=0.5)
        self.fc3 = nn.Sequential(*get_fc(4096, 20, 'none'))

        self.nonlinear = lambda x: torch.clamp(x,0)

    def forward(self, x):
        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)

        x = self.conv3(x)
        x = self.nonlinear(x)
        x = self.conv4(x)
        x = self.nonlinear(x)
        x = self.conv5(x)
        x = self.nonlinear(x)
        x = self.pool3(x)
        x = x.view(N, self.flat_dim) # flatten the array

        out = self.fc1(x)
        out = self.nonlinear(out)
        out = self.dropout1(out)
        out = self.fc2(out)
        out = self.nonlinear(out)
        out = self.dropout2(out)
        out = self.fc3(out)

        return out
```

2.2 Save the Model (5 pts)

Finish code stubs for saving the model periodically into `trainer.py` . **You will need these models later**

2.3 Train and Test (5pts)

Show clear screenshots of testing MAP and training loss for 50 epochs. Please evaluate your model to calculate the MAP on the testing dataset every 250 iterations. Use the following hyperparamters:

- batch_size=32
- Adam optimizer with lr=0.0001

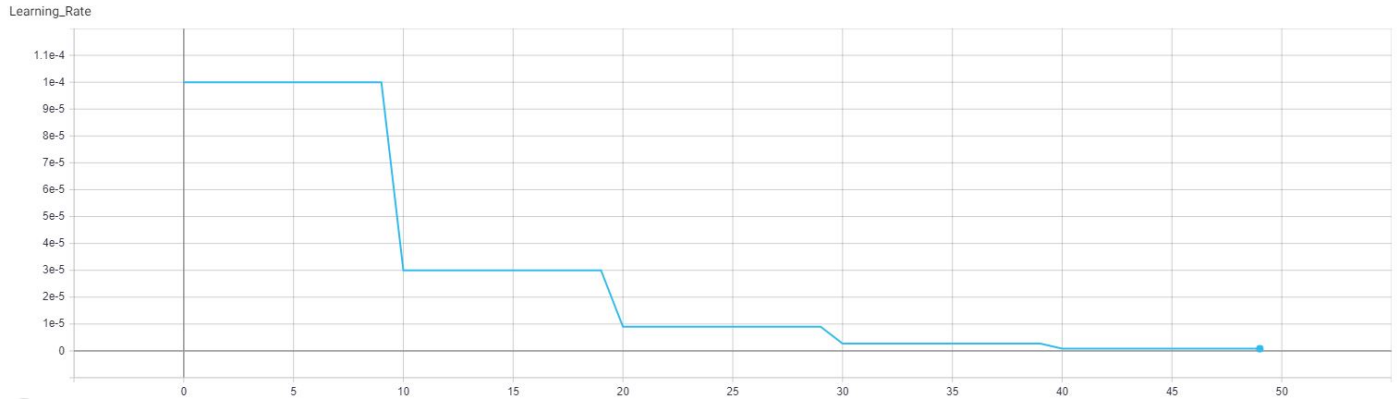
NOTE: SAVE AT LEAST 5 EVENLY SPACED CHECKPOINTS DURING TRAINING (1 at end)

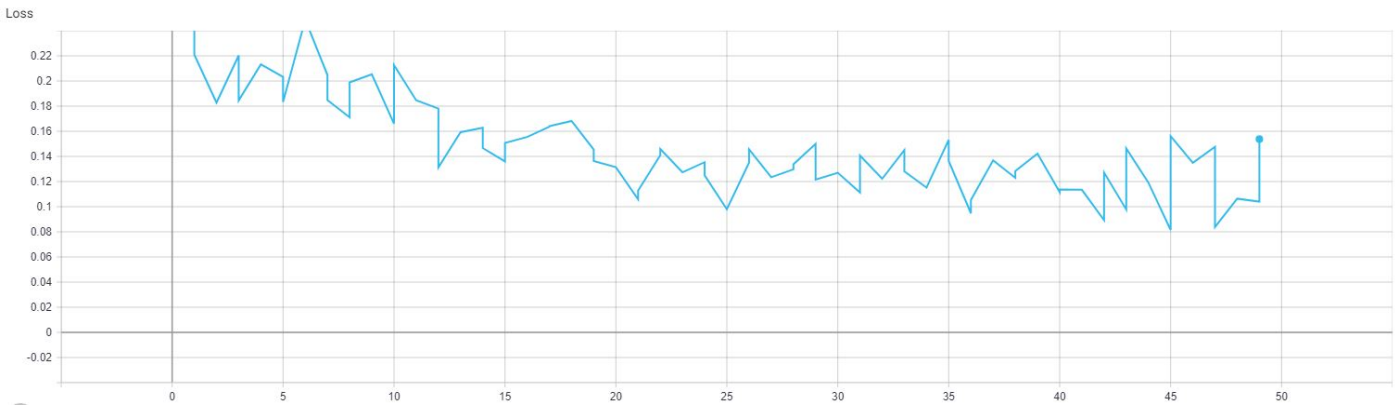
In [ ]:
```
def xavier_normal_init(m):
    if(type(m)==nn.Conv1d or type(m)==nn.Conv2d or type(m)==nn.Linear):
        torch.nn.init.xavier_normal_(m.weight.data)
        if(m.bias is not None):
            torch.nn.init.xavier_normal_(m.weight.data)
```

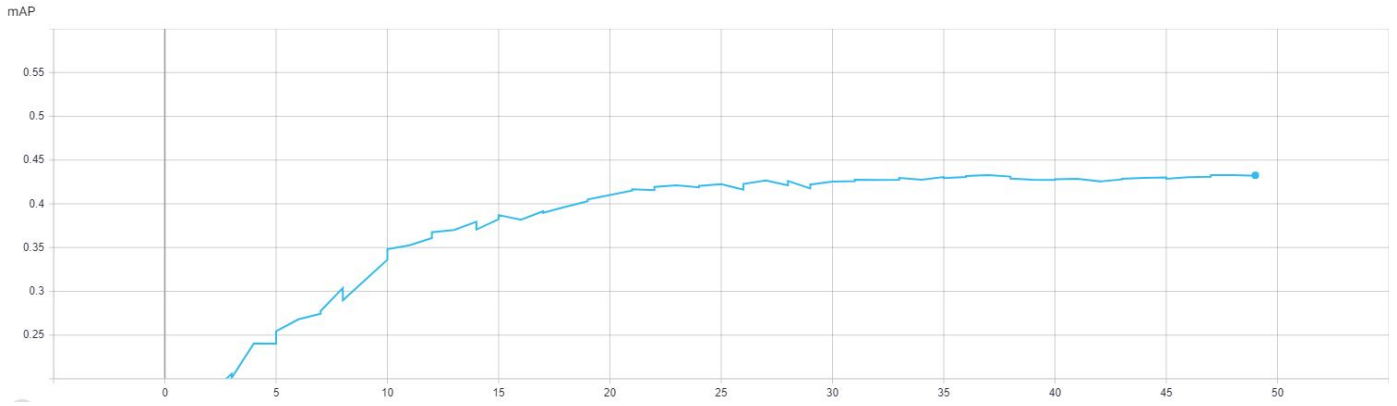
In [ ]:
```
args = ARGS(batch_size = 32, epochs=50, lr = 0.0001)
args.gamma = 0.3
weightDecay = 5e-5
model = CaffeNet()
model.apply(xavier_normal_init)
optimizer = torch.optim.Adam(model.parameters(), lr = args.lr,weight_decay=wei
ghtDecay) # ,weight_decay=weightDecay
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=10, gamma=arg
s.gamma)
if __name__ == '__main__':
    test_ap, test_map = trainer.train(args, model, optimizer, scheduler)
    print('test map:', test_map)
```

INSERT YOUR TENSORBOARD SCREENSHOTS HERE

The figures below display the learning rate, loss, and map, respectively, during training.



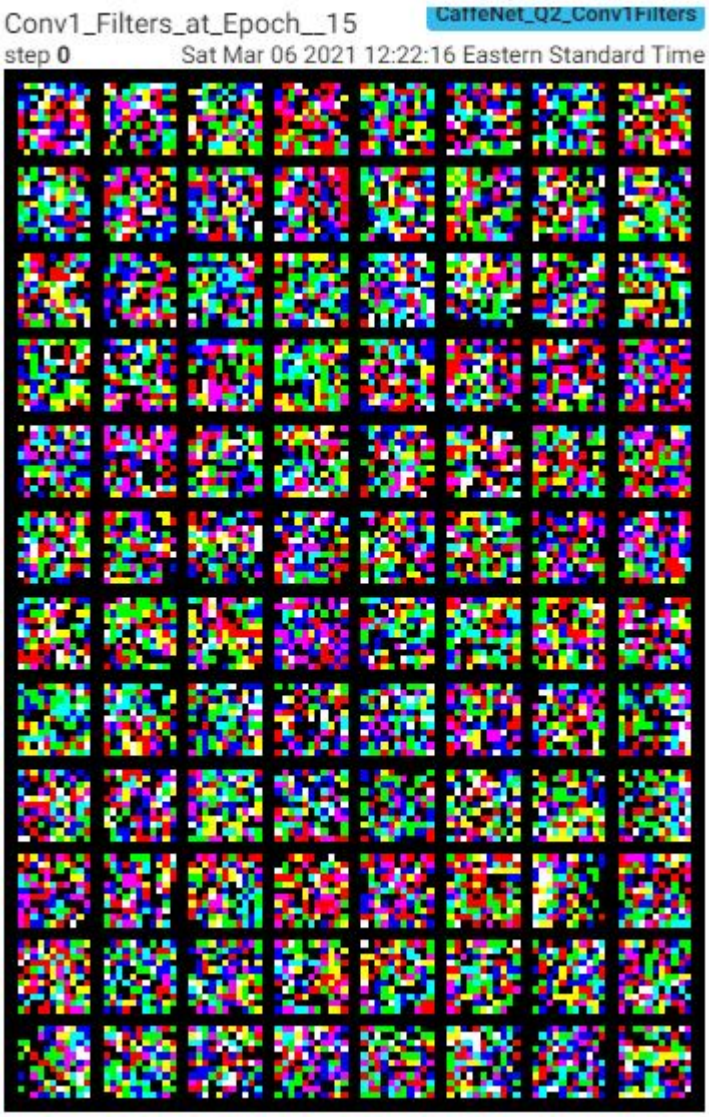




2.4 Visualizing: Conv-1 filters (5pts)

Extract and compare the conv1 filters, at different stages of the training (at least from 3 different iterations). Show at least 5 filters.

The following two image grids below show the Conv1 layer filters at epochs 15, 30, and 45, respectively:

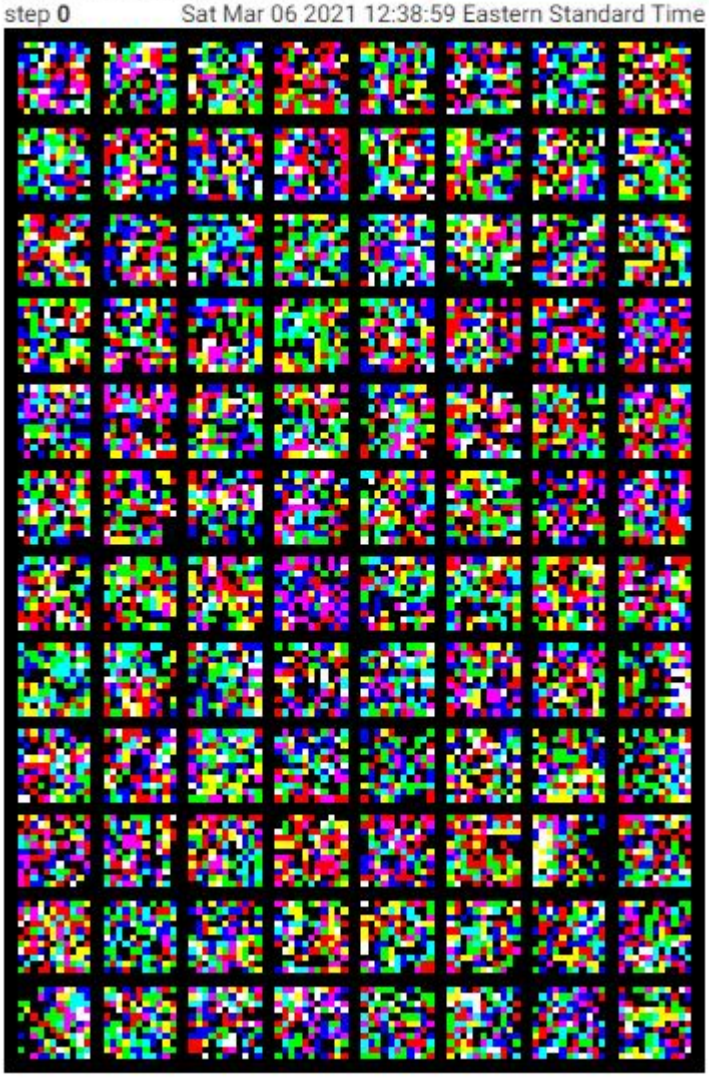


Conv1_Filters_at_Epoch__30

CaffeNet_Q2_Conv1Filters

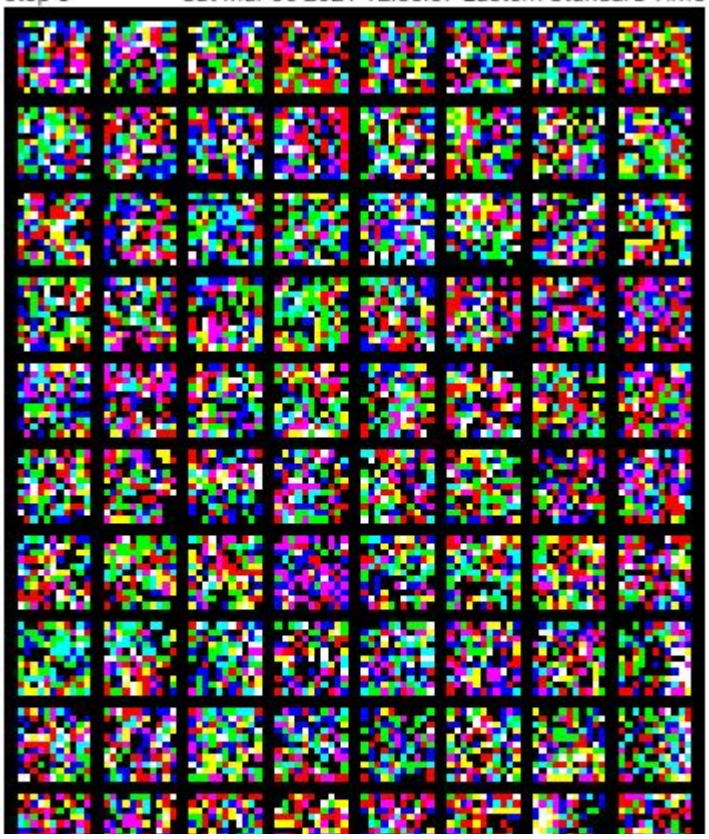
step **0**

Sat Mar 06 2021 12:38:59 Eastern Standard Time



Conv1_Filters_at_Epoch__45 CaffeNet_Q2_Conv1Filters

step **0** Sat Mar 06 2021 12:55:07 Eastern Standard Time



Appendex A: Training Epoch Information

The following log shown below displays the batch iteration, loss, and mAP calculation at various points during the training process:

Train Epoch: 0 [0 (0%)] Loss: 0.694260 | mAP: 0.079919
Train Epoch: 0 [100 (64%)] Loss: 0.252534 | mAP: 0.106128
Train Epoch: 1 [200 (27%)] Loss: 0.245160 | mAP: 0.125702
Train Epoch: 1 [300 (91%)] Loss: 0.220970 | mAP: 0.154655
Train Epoch: 2 [400 (55%)] Loss: 0.182752 | mAP: 0.181409
Train Epoch: 3 [500 (18%)] Loss: 0.220406 | mAP: 0.205849
Train Epoch: 3 [600 (82%)] Loss: 0.184534 | mAP: 0.200986
Train Epoch: 4 [700 (46%)] Loss: 0.213212 | mAP: 0.240417
Train Epoch: 5 [800 (10%)] Loss: 0.203305 | mAP: 0.240243
Train Epoch: 5 [900 (73%)] Loss: 0.183403 | mAP: 0.254320
Train Epoch: 6 [1000 (37%)] Loss: 0.248874 | mAP: 0.267995
Train Epoch: 7 [1100 (1%)] Loss: 0.204796 | mAP: 0.274374
Train Epoch: 7 [1200 (64%)] Loss: 0.184777 | mAP: 0.277550
Train Epoch: 8 [1300 (28%)] Loss: 0.171141 | mAP: 0.303782
Train Epoch: 8 [1400 (92%)] Loss: 0.198896 | mAP: 0.289614
Train Epoch: 9 [1500 (55%)] Loss: 0.205384 | mAP: 0.312908
Train Epoch: 10 [1600 (19%)] Loss: 0.166079 | mAP: 0.336100
Train Epoch: 10 [1700 (83%)] Loss: 0.212687 | mAP: 0.348000
Train Epoch: 11 [1800 (46%)] Loss: 0.184672 | mAP: 0.352580
Train Epoch: 12 [1900 (10%)] Loss: 0.178017 | mAP: 0.360864
Train Epoch: 12 [2000 (74%)] Loss: 0.131287 | mAP: 0.367583
Train Epoch: 13 [2100 (38%)] Loss: 0.159262 | mAP: 0.370199
Train Epoch: 14 [2200 (1%)] Loss: 0.162815 | mAP: 0.379512
Train Epoch: 14 [2300 (65%)] Loss: 0.146647 | mAP: 0.370575
Train Epoch: 15 [2400 (29%)] Loss: 0.136011 | mAP: 0.382428
Train Epoch: 15 [2500 (92%)] Loss: 0.150734 | mAP: 0.387179
Train Epoch: 16 [2600 (56%)] Loss: 0.155456 | mAP: 0.381900
Train Epoch: 17 [2700 (20%)] Loss: 0.163627 | mAP: 0.391537
Train Epoch: 17 [2800 (83%)] Loss: 0.164096 | mAP: 0.389797
Train Epoch: 18 [2900 (47%)] Loss: 0.168196 | mAP: 0.396490
Train Epoch: 19 [3000 (11%)] Loss: 0.145444 | mAP: 0.402965
Train Epoch: 19 [3100 (75%)] Loss: 0.136387 | mAP: 0.404979
Train Epoch: 20 [3200 (38%)] Loss: 0.131393 | mAP: 0.410043
Train Epoch: 21 [3300 (2%)] Loss: 0.105851 | mAP: 0.415120
Train Epoch: 21 [3400 (66%)] Loss: 0.112801 | mAP: 0.416542
Train Epoch: 22 [3500 (29%)] Loss: 0.140922 | mAP: 0.415565
Train Epoch: 22 [3600 (93%)] Loss: 0.145867 | mAP: 0.419292
Train Epoch: 23 [3700 (57%)] Loss: 0.127371 | mAP: 0.421104
Train Epoch: 24 [3800 (20%)] Loss: 0.135417 | mAP: 0.418907
Train Epoch: 24 [3900 (84%)] Loss: 0.124882 | mAP: 0.420355
Train Epoch: 25 [4000 (48%)] Loss: 0.097941 | mAP: 0.422405
Train Epoch: 26 [4100 (11%)] Loss: 0.134959 | mAP: 0.416246
Train Epoch: 26 [4200 (75%)] Loss: 0.145727 | mAP: 0.422733

Train Epoch: 27 [4300 (39%)] Loss: 0.123460 | mAP: 0.426791
Train Epoch: 28 [4400 (3%)] Loss: 0.129733 | mAP: 0.421078
Train Epoch: 28 [4500 (66%)] Loss: 0.133712 | mAP: 0.426055
Train Epoch: 29 [4600 (30%)] Loss: 0.149955 | mAP: 0.417618
Train Epoch: 29 [4700 (94%)] Loss: 0.121479 | mAP: 0.421980
Train Epoch: 30 [4800 (57%)] Loss: 0.127036 | mAP: 0.425303
Train Epoch: 31 [4900 (21%)] Loss: 0.111305 | mAP: 0.425996
Train Epoch: 31 [5000 (85%)] Loss: 0.140682 | mAP: 0.427547
Train Epoch: 32 [5100 (48%)] Loss: 0.122193 | mAP: 0.427262
Train Epoch: 33 [5200 (12%)] Loss: 0.145076 | mAP: 0.427588
Train Epoch: 33 [5300 (76%)] Loss: 0.128083 | mAP: 0.429591
Train Epoch: 34 [5400 (39%)] Loss: 0.115151 | mAP: 0.427573
Train Epoch: 35 [5500 (3%)] Loss: 0.153220 | mAP: 0.430578
Train Epoch: 35 [5600 (67%)] Loss: 0.136315 | mAP: 0.429386
Train Epoch: 36 [5700 (31%)] Loss: 0.094817 | mAP: 0.430634
Train Epoch: 36 [5800 (94%)] Loss: 0.105419 | mAP: 0.431691
Train Epoch: 37 [5900 (58%)] Loss: 0.136911 | mAP: 0.432678
Train Epoch: 38 [6000 (22%)] Loss: 0.122926 | mAP: 0.431091
Train Epoch: 38 [6100 (85%)] Loss: 0.128397 | mAP: 0.428878
Train Epoch: 39 [6200 (49%)] Loss: 0.142264 | mAP: 0.427402
Train Epoch: 40 [6300 (13%)] Loss: 0.111605 | mAP: 0.427158
Train Epoch: 40 [6400 (76%)] Loss: 0.113659 | mAP: 0.428018
Train Epoch: 41 [6500 (40%)] Loss: 0.113470 | mAP: 0.428675
Train Epoch: 42 [6600 (4%)] Loss: 0.089273 | mAP: 0.425716
Train Epoch: 42 [6700 (68%)] Loss: 0.127156 | mAP: 0.425413
Train Epoch: 43 [6800 (31%)] Loss: 0.097778 | mAP: 0.427899
Train Epoch: 43 [6900 (95%)] Loss: 0.146353 | mAP: 0.428698
Train Epoch: 44 [7000 (59%)] Loss: 0.119124 | mAP: 0.429632
Train Epoch: 45 [7100 (22%)] Loss: 0.081511 | mAP: 0.430262
Train Epoch: 45 [7200 (86%)] Loss: 0.156132 | mAP: 0.428628
Train Epoch: 46 [7300 (50%)] Loss: 0.134881 | mAP: 0.430437
Train Epoch: 47 [7400 (13%)] Loss: 0.147624 | mAP: 0.430969
Train Epoch: 47 [7500 (77%)] Loss: 0.083810 | mAP: 0.432749
Train Epoch: 48 [7600 (41%)] Loss: 0.106350 | mAP: 0.432845
Train Epoch: 49 [7700 (4%)] Loss: 0.104012 | mAP: 0.431998
Train Epoch: 49 [7800 (68%)] Loss: 0.153784 | mAP: 0.432565
test map: 0.41263213323264036

Q3: Even deeper! Resnet18 for PASCAL classification (15 pts)

Hopefully we all got much better accuracy with the deeper model! Since 2012, much deeper architectures have been proposed. ResNet (https://arxiv.org/abs/1512.03385) is one of the popular ones. In this task, we attempt to further improve the performance with the “very deep” ResNet-18 architecture.

3.1 Build ResNet-18 (1 pts)

Write a network modules for the Resnet-18 architecture (refer to the original paper). You can use `torchvision.models` for this section, so it should be very easy!

In [ ]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torchvision import models
import matplotlib.pyplot as plt
%matplotlib inline

import trainer
from utils import ARGS
from simple_cnn import SimpleCNN
from voc_dataset import VOCDataset
```

3.2 Add Tensorboard Summaries (6 pts)

You should've already written tensorboard summary generation code into `trainer.py` from q1. However, you probably just added the most basic summary features. Please implement the more advanced summaries listed here:

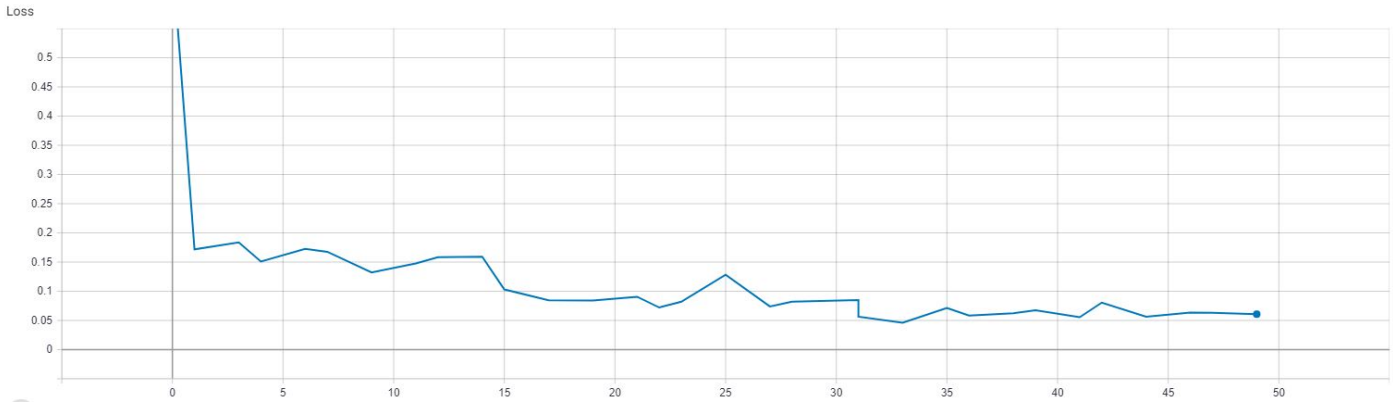
- training loss (should be done)
- testing MAP curves (should be done)
- learning rate
- histogram of gradients

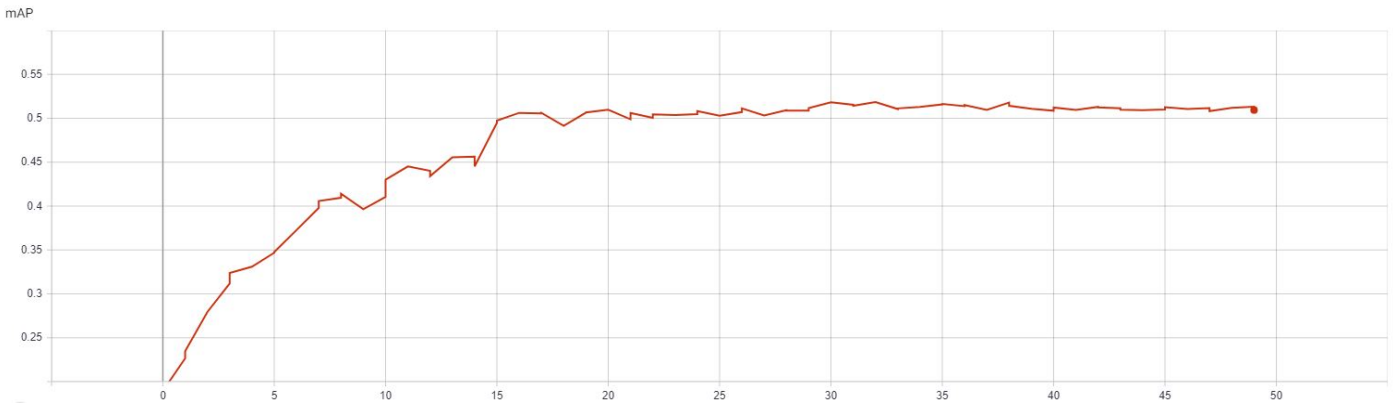
3.3 Train and Test (8 pts)

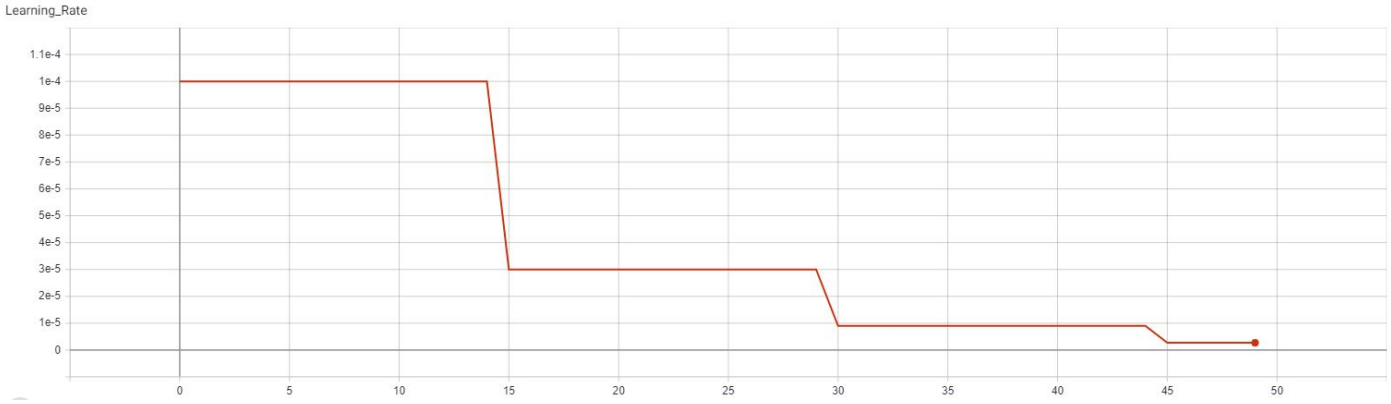
Use the same hyperparameter settings from Task 2, and train the model for 50 epochs. Report tensorboard screenshots for *all* of the summaries listed above (for image summaries show screenshots at $n \geq 3$ iterations)

REMEMBER TO SAVE A MODEL AT THE END OF TRAINING

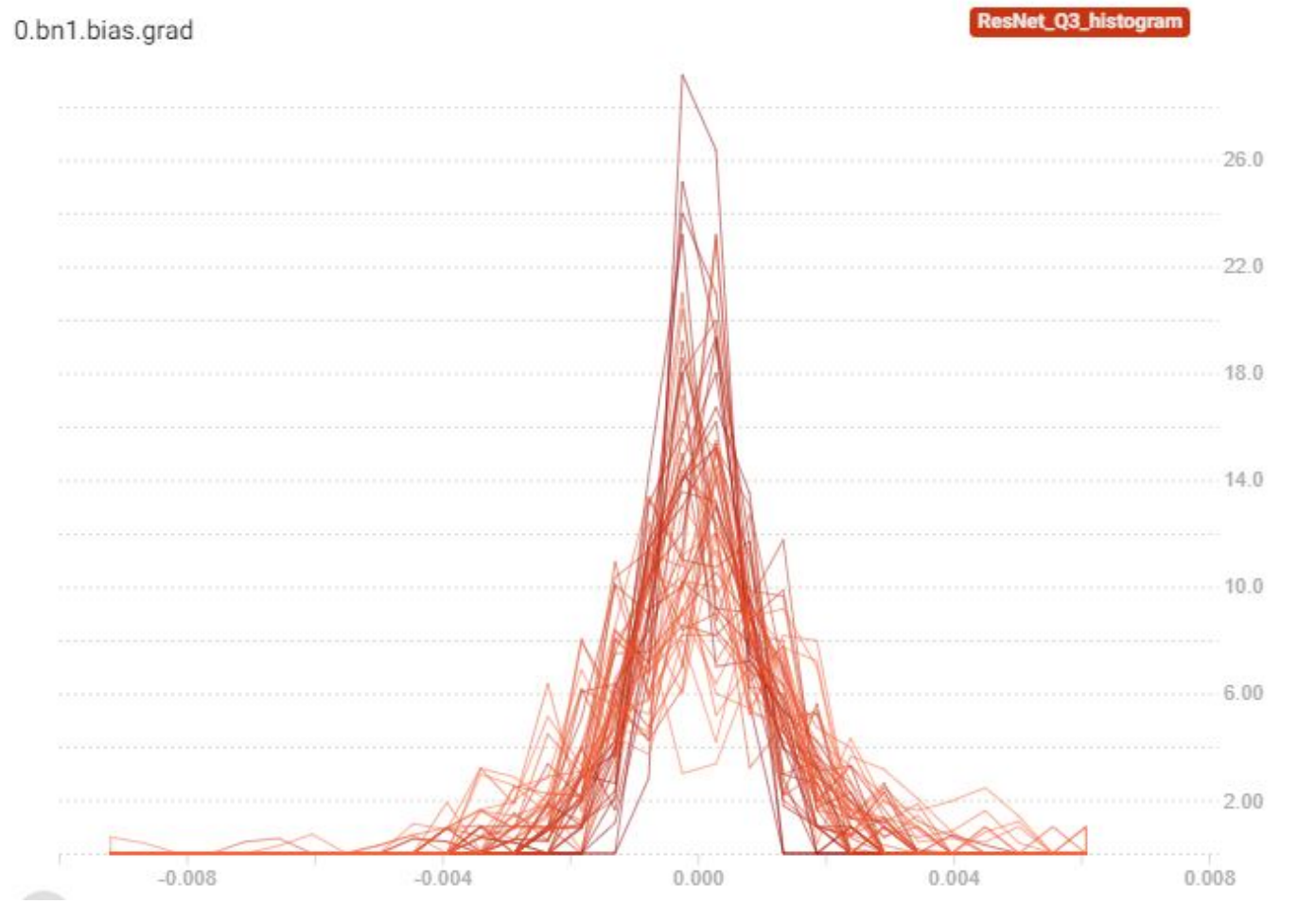
The figure shown below display the training loss, training mAP, learning rate, and histogram of gradients as displayed through TensorBoard.





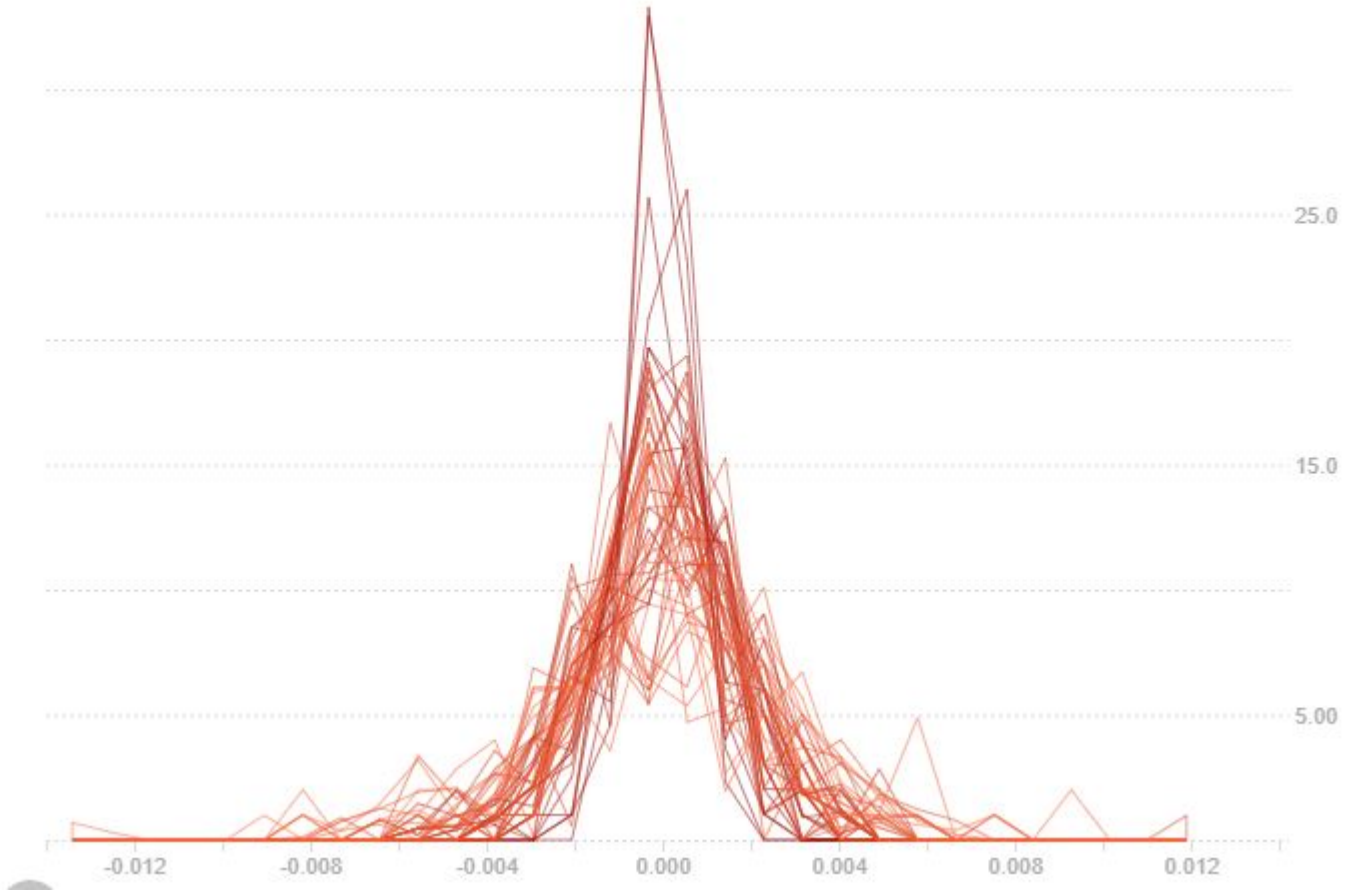


The following 4 images below showcase histogram of gradients for the first two layers:



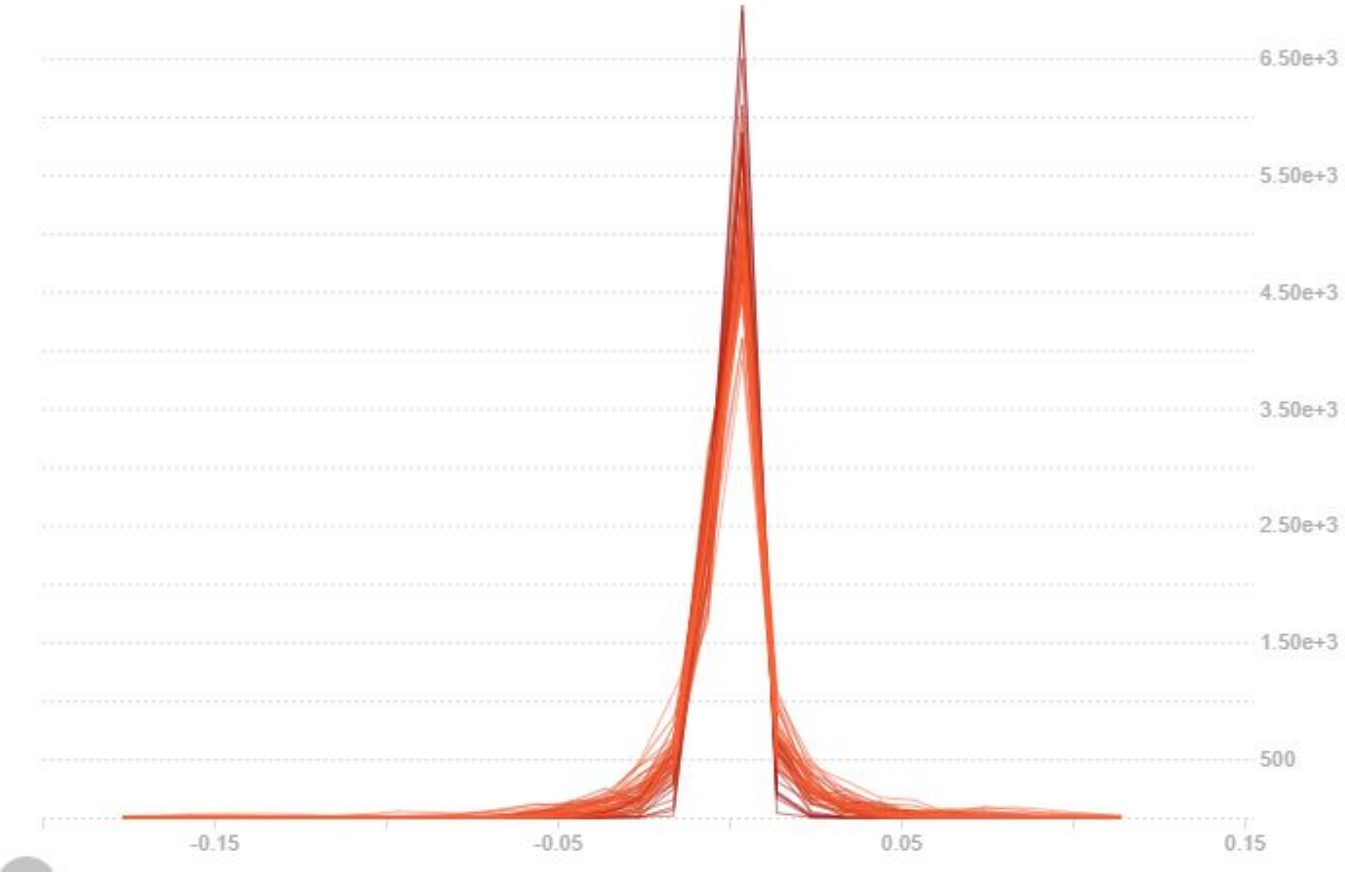
0.bn1.weight.grad

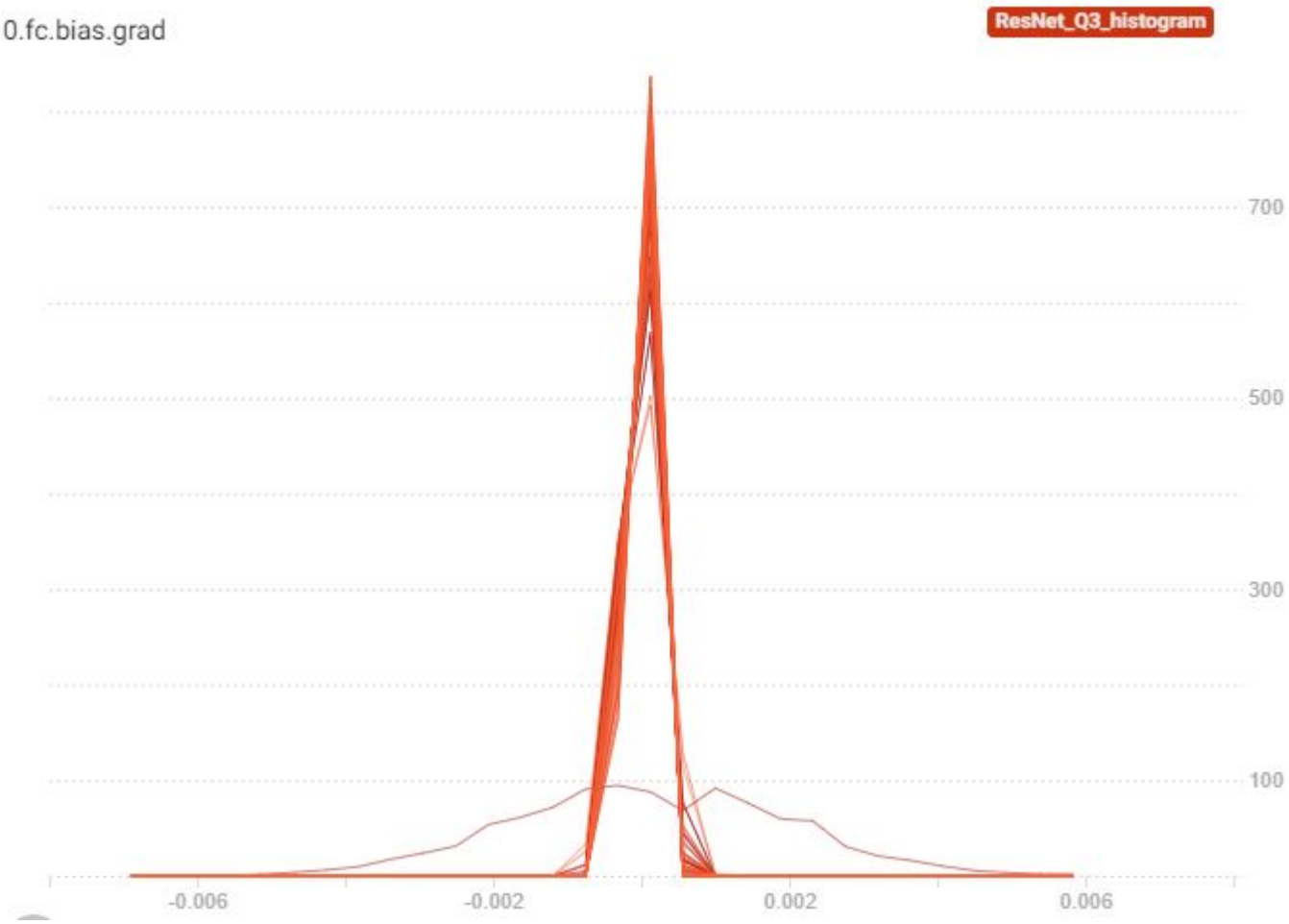
ResNet_Q3_histogram



0.conv1.weight.grad

ResNet_Q3_histogram

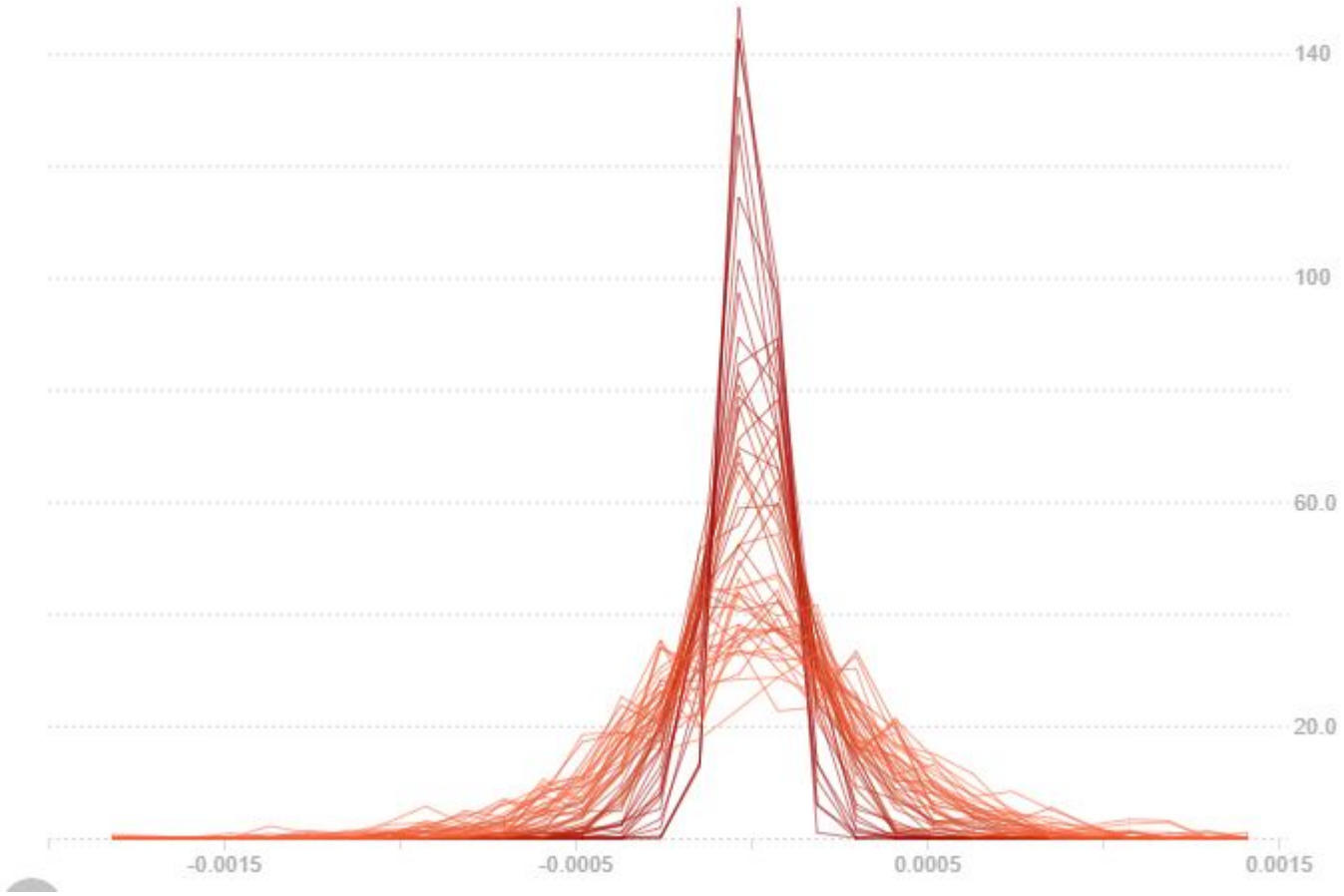




The following 4 images below showcase histogram of gradients for the middle two layers:

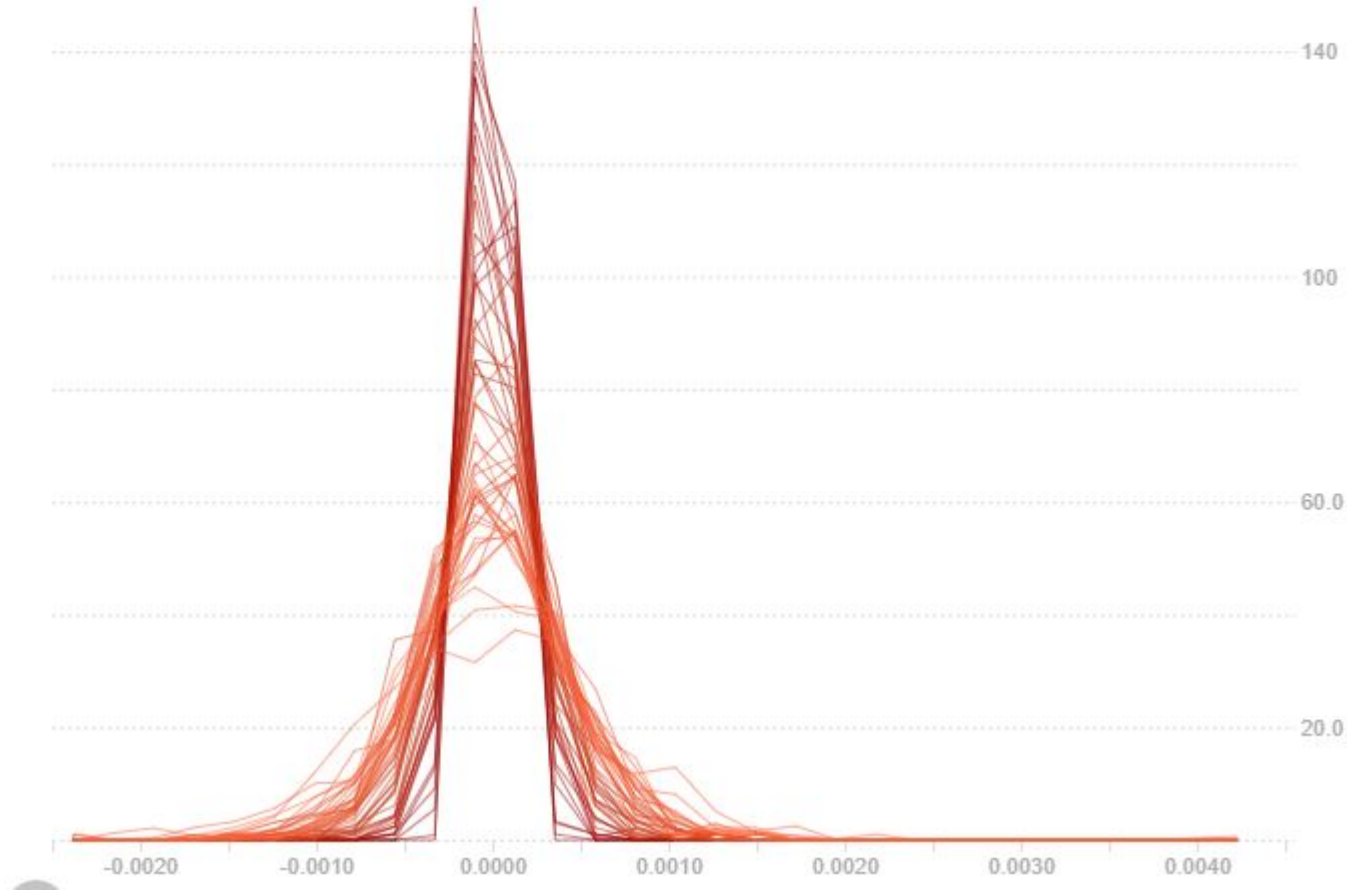
0.layer3.0.bn2.bias.grad

ResNet_Q3_histogram



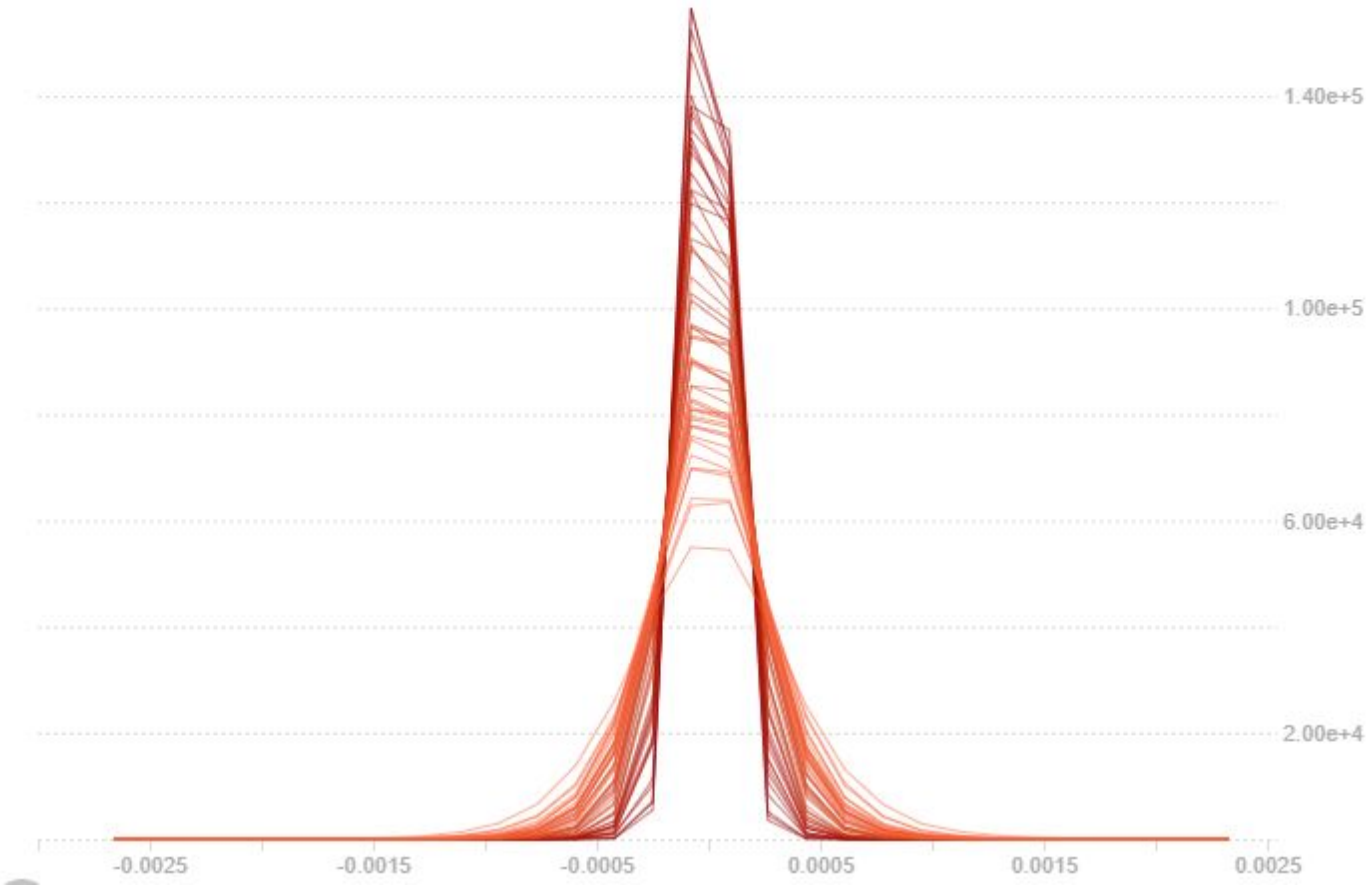
0.layer3.0.bn2.weight.grad

ResNet_Q3_histogram



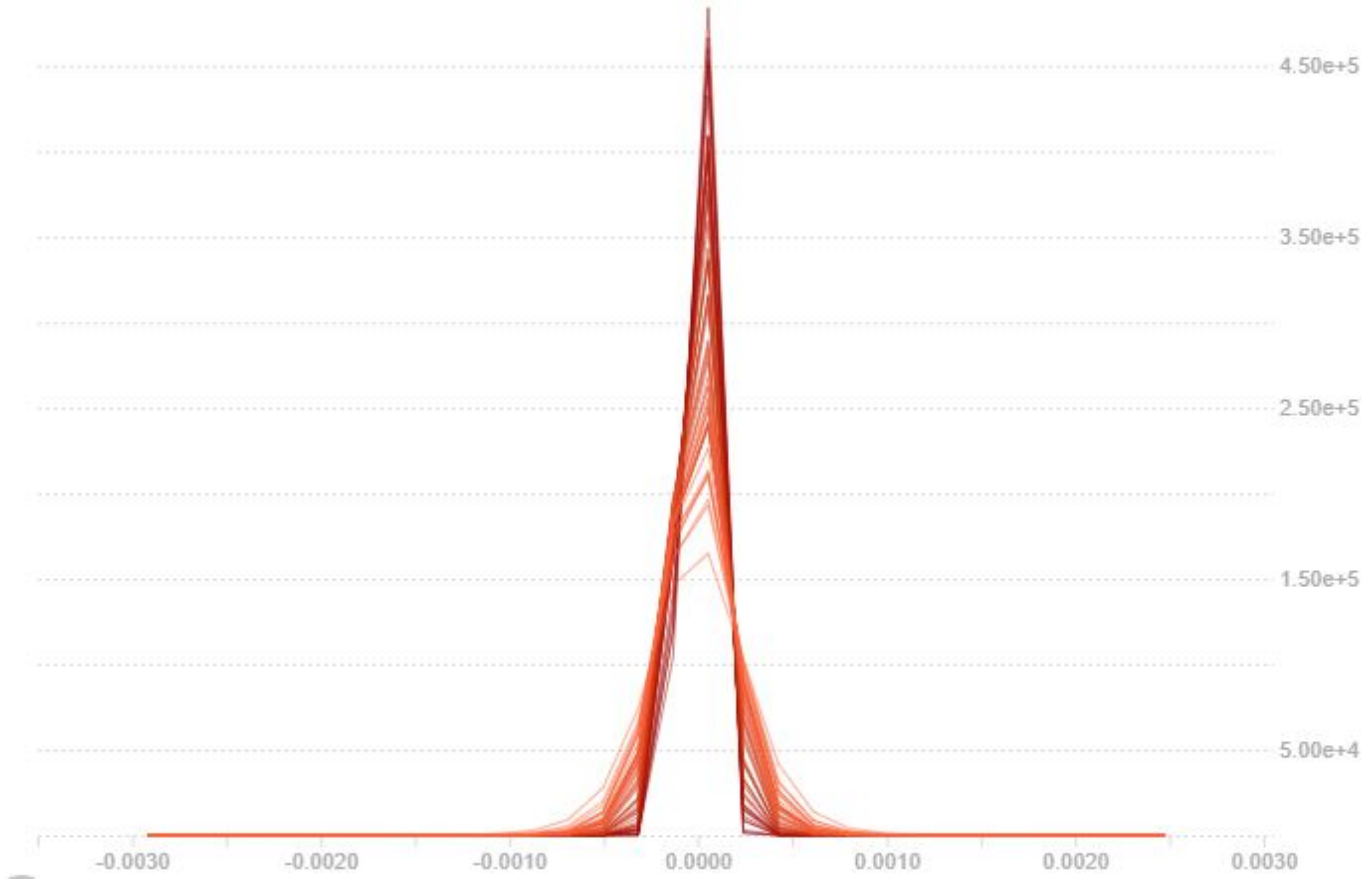
0.layer3.0.conv1.weight.grad

ResNet_Q3_histogram

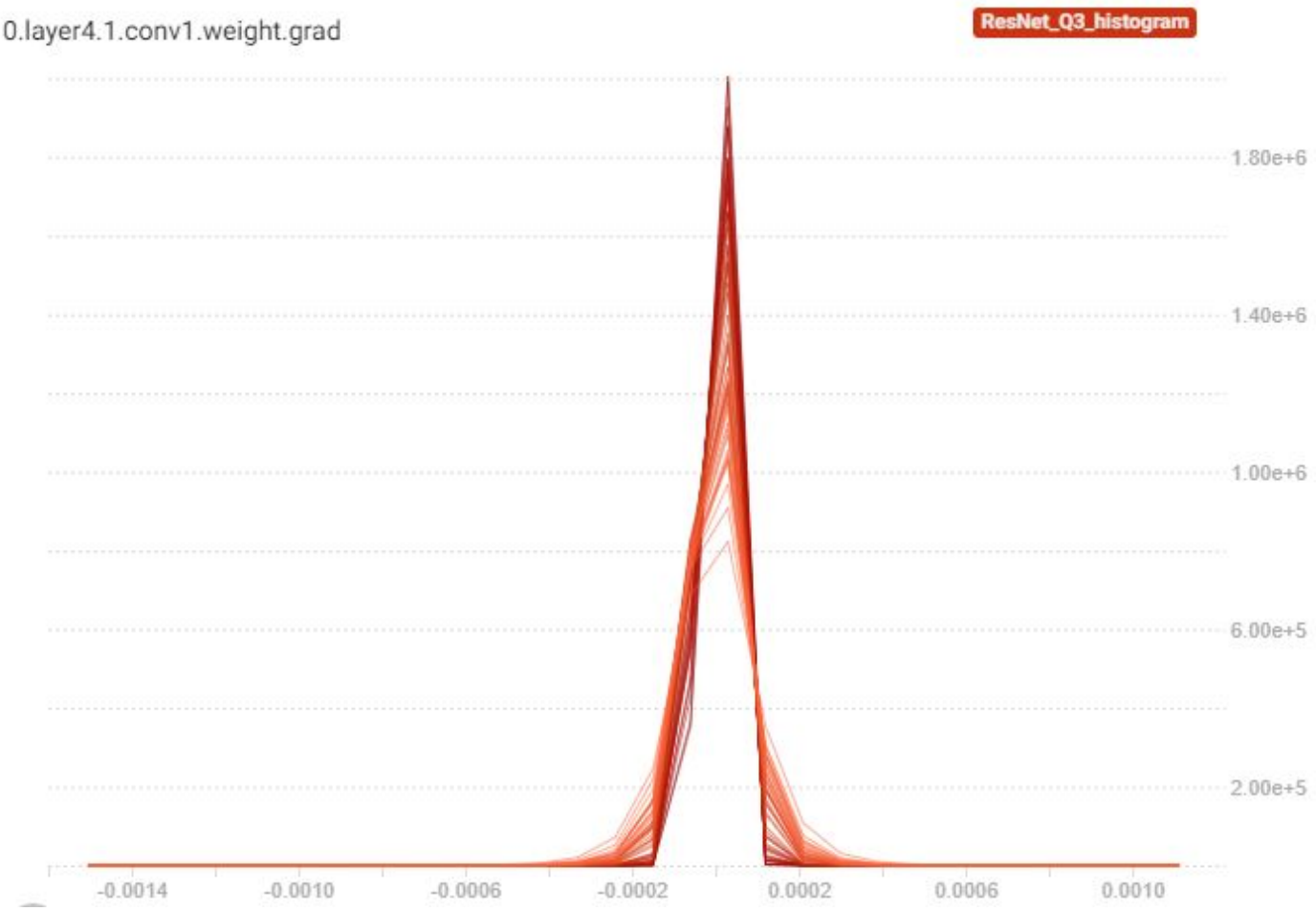


0.layer3.0.conv2.weight.grad

ResNet_Q3_histogram

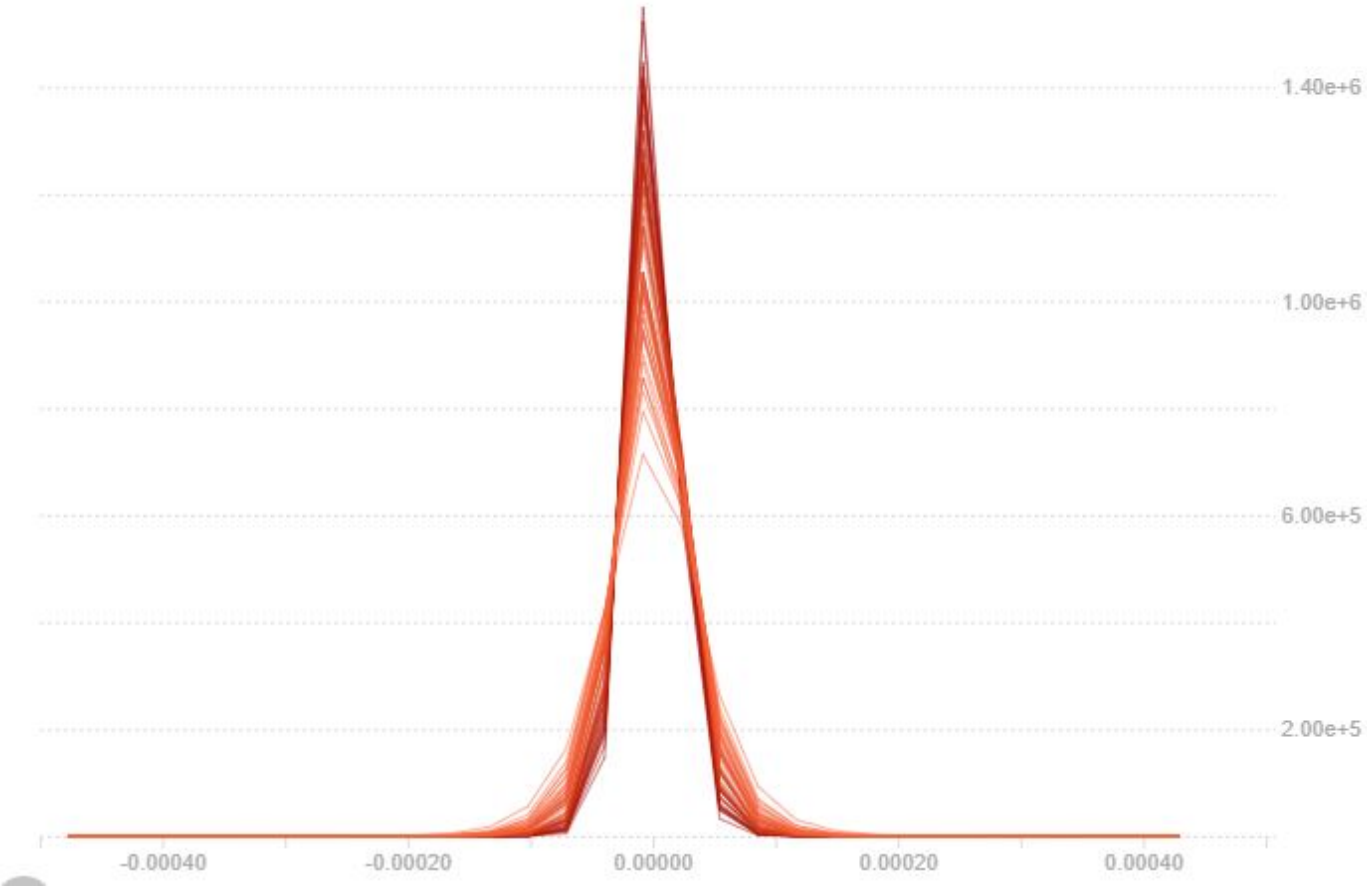


The following 4 images below showcase histogram of gradients for the last two layers:



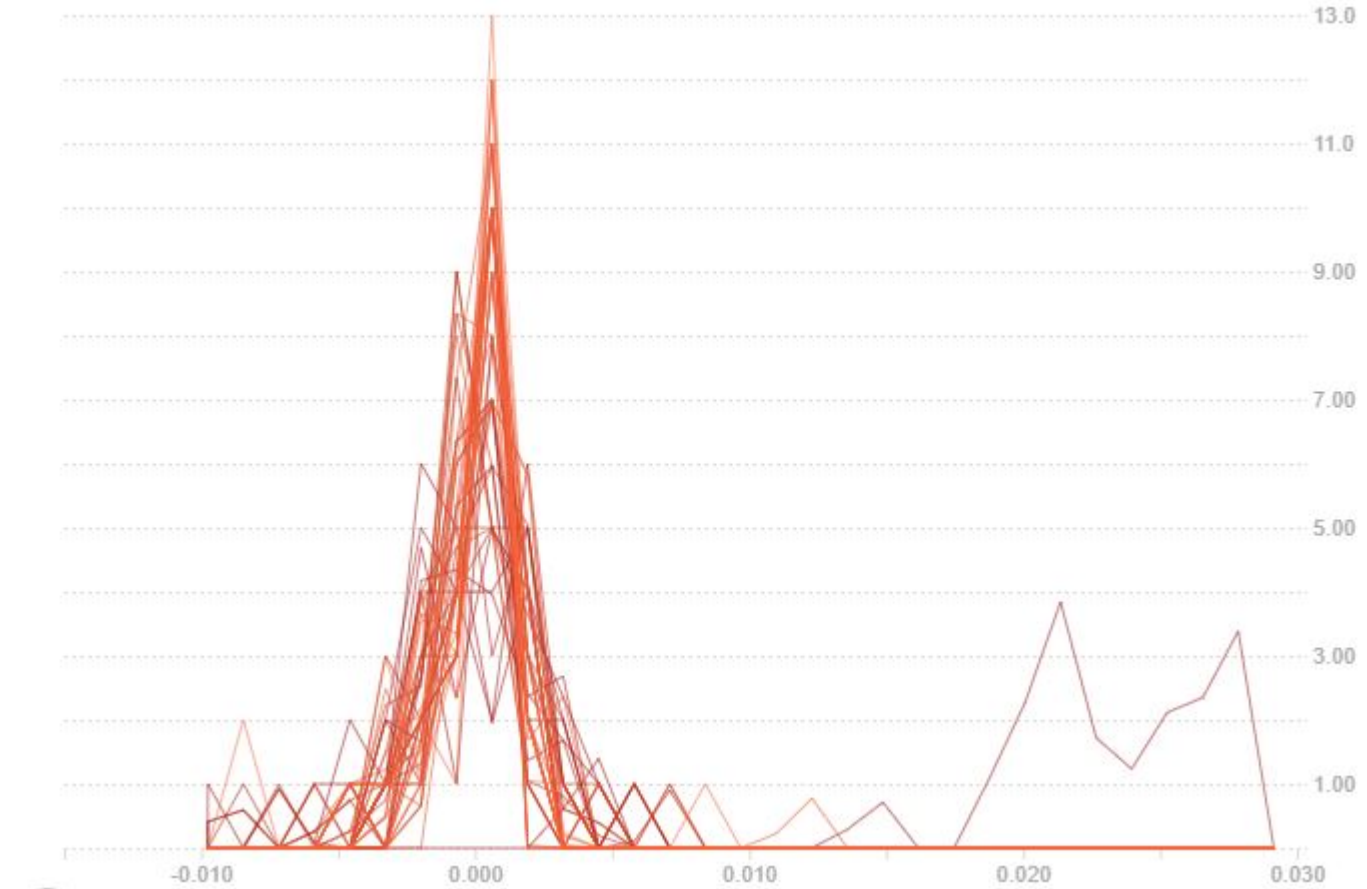
0.layer4.1.conv2.weight.grad

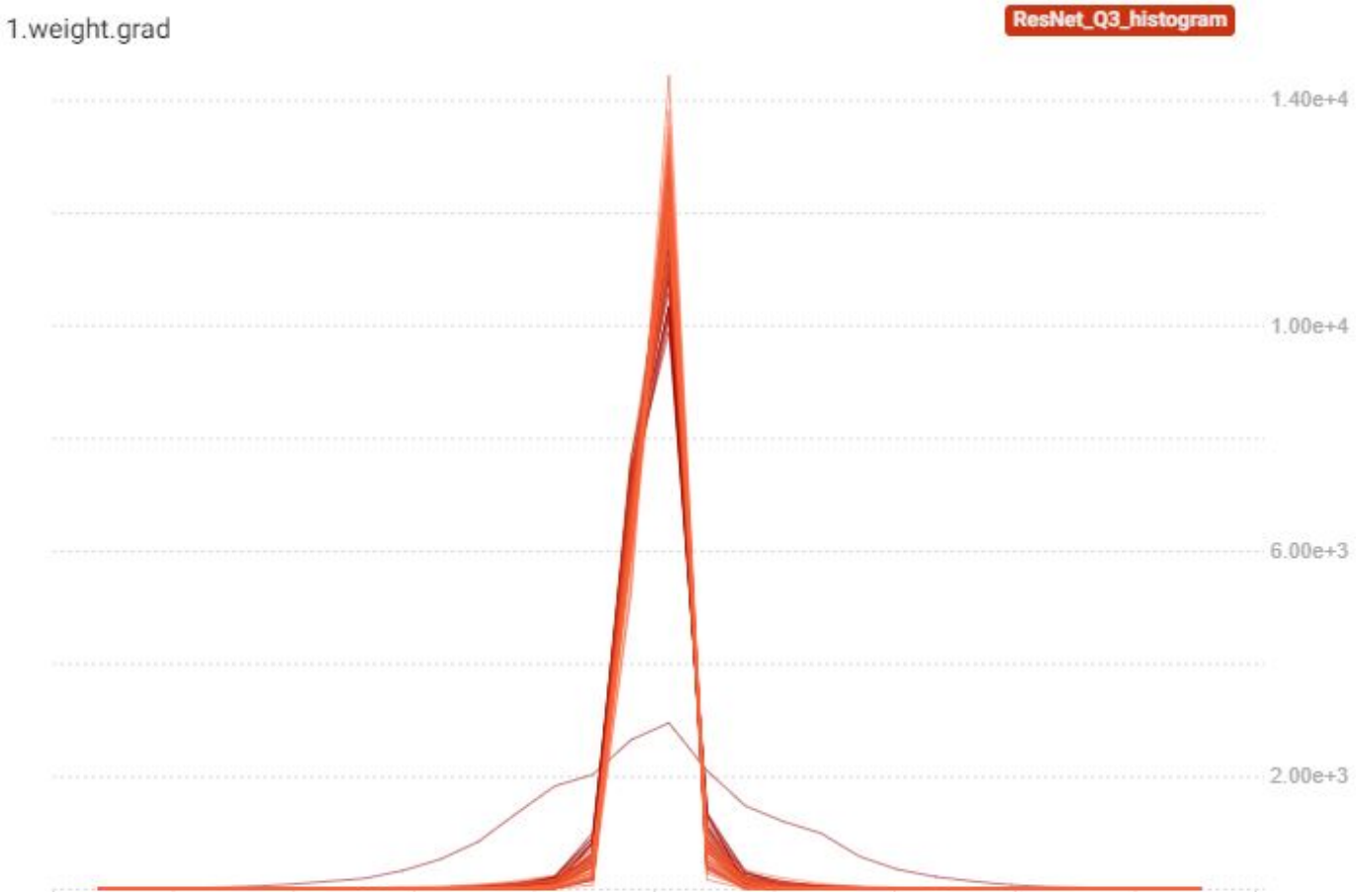
ResNet_Q3_histogram



1.bias.grad

ResNet_Q3_histogram





In [ ]:
```
args = ARGS(batch_size = 32, epochs=50, lr = 0.0001)
args.gamma = 0.3
modelres = models.resnet18(pretrained=False)
model= nn.Sequential(modelres,nn.Linear(1000,20,bias=True))
optimizer = torch.optim.Adam(model.parameters(), lr = args.lr)
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=15, gamma=arg
s.gamma)
if __name__ == '__main__':
    test_ap, test_map = trainer.train(args, model, optimizer, scheduler)
    print('test map:', test_map)
```

Appendex A: Training Epoch Information

The following log shown below displays the batch iteration, loss, and mAP calculation at various points during the training process:

Train Epoch: 0 [0 (0%)] Loss: 0.720259 | mAP: 0.073370
Train Epoch: 0 [100 (64%)] Loss: 0.215044 | mAP: 0.189455
Train Epoch: 1 [200 (27%)] Loss: 0.216712 | mAP: 0.226419
Train Epoch: 1 [300 (91%)] Loss: 0.208612 | mAP: 0.234548
Train Epoch: 2 [400 (55%)] Loss: 0.181140 | mAP: 0.279389
Train Epoch: 3 [500 (18%)] Loss: 0.202289 | mAP: 0.311690
Train Epoch: 3 [600 (82%)] Loss: 0.197823 | mAP: 0.323791
Train Epoch: 4 [700 (46%)] Loss: 0.184834 | mAP: 0.330959
Train Epoch: 5 [800 (10%)] Loss: 0.185951 | mAP: 0.346840
Train Epoch: 5 [900 (73%)] Loss: 0.165875 | mAP: 0.347633
Train Epoch: 6 [1000 (37%)] Loss: 0.165044 | mAP: 0.372633
Train Epoch: 7 [1100 (1%)] Loss: 0.156291 | mAP: 0.397830
Train Epoch: 7 [1200 (64%)] Loss: 0.149558 | mAP: 0.405792
Train Epoch: 8 [1300 (28%)] Loss: 0.155647 | mAP: 0.409463
Train Epoch: 8 [1400 (92%)] Loss: 0.146398 | mAP: 0.414038
Train Epoch: 9 [1500 (55%)] Loss: 0.153696 | mAP: 0.396369
Train Epoch: 10 [1600 (19%)] Loss: 0.137831 | mAP: 0.410398
Train Epoch: 10 [1700 (83%)] Loss: 0.167851 | mAP: 0.430070
Train Epoch: 11 [1800 (46%)] Loss: 0.130406 | mAP: 0.445091
Train Epoch: 12 [1900 (10%)] Loss: 0.152793 | mAP: 0.440219
Train Epoch: 12 [2000 (74%)] Loss: 0.140506 | mAP: 0.434110
Train Epoch: 13 [2100 (38%)] Loss: 0.140575 | mAP: 0.455477
Train Epoch: 14 [2200 (1%)] Loss: 0.118644 | mAP: 0.456327
Train Epoch: 14 [2300 (65%)] Loss: 0.138294 | mAP: 0.445303
Train Epoch: 15 [2400 (29%)] Loss: 0.103462 | mAP: 0.494774
Train Epoch: 15 [2500 (92%)] Loss: 0.114315 | mAP: 0.497309
Train Epoch: 16 [2600 (56%)] Loss: 0.105307 | mAP: 0.506209
Train Epoch: 17 [2700 (20%)] Loss: 0.126236 | mAP: 0.505680
Train Epoch: 17 [2800 (83%)] Loss: 0.111384 | mAP: 0.506179
Train Epoch: 18 [2900 (47%)] Loss: 0.089492 | mAP: 0.491313
Train Epoch: 19 [3000 (11%)] Loss: 0.129798 | mAP: 0.506378
Train Epoch: 19 [3100 (75%)] Loss: 0.124896 | mAP: 0.506707
Train Epoch: 20 [3200 (38%)] Loss: 0.097039 | mAP: 0.509701
Train Epoch: 21 [3300 (2%)] Loss: 0.069134 | mAP: 0.498853
Train Epoch: 21 [3400 (66%)] Loss: 0.109458 | mAP: 0.506017
Train Epoch: 22 [3500 (29%)] Loss: 0.070703 | mAP: 0.500423
Train Epoch: 22 [3600 (93%)] Loss: 0.074366 | mAP: 0.504471
Train Epoch: 23 [3700 (57%)] Loss: 0.092572 | mAP: 0.503754
Train Epoch: 24 [3800 (20%)] Loss: 0.100229 | mAP: 0.504639
Train Epoch: 24 [3900 (84%)] Loss: 0.089912 | mAP: 0.508232
Train Epoch: 25 [4000 (48%)] Loss: 0.083635 | mAP: 0.502918
Train Epoch: 26 [4100 (11%)] Loss: 0.083434 | mAP: 0.507066
Train Epoch: 26 [4200 (75%)] Loss: 0.070950 | mAP: 0.511265

Train Epoch: 27 [4300 (39%)] Loss: 0.073017 | mAP: 0.503205
Train Epoch: 28 [4400 (3%)] Loss: 0.074057 | mAP: 0.509178
Train Epoch: 28 [4500 (66%)] Loss: 0.089060 | mAP: 0.508615
Train Epoch: 29 [4600 (30%)] Loss: 0.081091 | mAP: 0.508820
Train Epoch: 29 [4700 (94%)] Loss: 0.082942 | mAP: 0.511548
Train Epoch: 30 [4800 (57%)] Loss: 0.063460 | mAP: 0.518176
Train Epoch: 31 [4900 (21%)] Loss: 0.073838 | mAP: 0.515324
Train Epoch: 31 [5000 (85%)] Loss: 0.062788 | mAP: 0.514318
Train Epoch: 32 [5100 (48%)] Loss: 0.067382 | mAP: 0.518381
Train Epoch: 33 [5200 (12%)] Loss: 0.063792 | mAP: 0.510551
Train Epoch: 33 [5300 (76%)] Loss: 0.058528 | mAP: 0.511280
Train Epoch: 34 [5400 (39%)] Loss: 0.080845 | mAP: 0.512993
Train Epoch: 35 [5500 (3%)] Loss: 0.071587 | mAP: 0.515748
Train Epoch: 35 [5600 (67%)] Loss: 0.066521 | mAP: 0.516310
Train Epoch: 36 [5700 (31%)] Loss: 0.038676 | mAP: 0.513884
Train Epoch: 36 [5800 (94%)] Loss: 0.109776 | mAP: 0.514998
Train Epoch: 37 [5900 (58%)] Loss: 0.056390 | mAP: 0.509673
Train Epoch: 38 [6000 (22%)] Loss: 0.057494 | mAP: 0.517916
Train Epoch: 38 [6100 (85%)] Loss: 0.048027 | mAP: 0.514223
Train Epoch: 39 [6200 (49%)] Loss: 0.062421 | mAP: 0.510778
Train Epoch: 40 [6300 (13%)] Loss: 0.064487 | mAP: 0.508616
Train Epoch: 40 [6400 (76%)] Loss: 0.060632 | mAP: 0.512278
Train Epoch: 41 [6500 (40%)] Loss: 0.076664 | mAP: 0.509637
Train Epoch: 42 [6600 (4%)] Loss: 0.053598 | mAP: 0.513044
Train Epoch: 42 [6700 (68%)] Loss: 0.054771 | mAP: 0.512391
Train Epoch: 43 [6800 (31%)] Loss: 0.044373 | mAP: 0.511334
Train Epoch: 43 [6900 (95%)] Loss: 0.071721 | mAP: 0.509659
Train Epoch: 44 [7000 (59%)] Loss: 0.068732 | mAP: 0.509379
Train Epoch: 45 [7100 (22%)] Loss: 0.108448 | mAP: 0.510053
Train Epoch: 45 [7200 (86%)] Loss: 0.069663 | mAP: 0.512628
Train Epoch: 46 [7300 (50%)] Loss: 0.054835 | mAP: 0.510637
Train Epoch: 47 [7400 (13%)] Loss: 0.069490 | mAP: 0.511583
Train Epoch: 47 [7500 (77%)] Loss: 0.059993 | mAP: 0.508102
Train Epoch: 48 [7600 (41%)] Loss: 0.058032 | mAP: 0.512003
Train Epoch: 49 [7700 (4%)] Loss: 0.056961 | mAP: 0.513160
Train Epoch: 49 [7800 (68%)] Loss: 0.047862 | mAP: 0.509374
test map: 0.5124332392148173

Q4 Shoulders of Giants (15 points)

As we have already seen, deep networks can sometimes be hard to optimize. Often times they heavily overfit on small training sets. Many approaches have been proposed to counter this, eg, Krahenbuhl et al. (ICLR'16) (http://arxiv.org/pdf/1511.06856.pdf), self-supervised learning, etc. However, the most effective approach remains pre-training the network on large, well-labeled supervised datasets such as ImageNet.

While training on the full ImageNet data is beyond the scope of this assignment, people have already trained many popular/standard models and released them online. In this task, we will initialize a ResNet-18 model with pre-trained ImageNet weights (from `torchvision`), and finetune the network for PASCAL classification.

4.1 Load Pre-trained Model (7 pts)\

Load the pre-trained weights up to the second last layer, and initialize last weights and biases from scratch.

The model loading mechanism is based on names of the weights. It is easy to load pretrained models from `torchvision.models`, even when your model uses different names for weights. Please briefly explain how to load the weights correctly if the names do not match (hint (https://discuss.pytorch.org/t/loading-weights-from-pretrained-model-with-different-module-names/11841)).

YOUR ANSWER HERE

If you are loading pretrained parameters into a model and the names of the parameters do not match, you can still iterate through the layers of the model and load those weights. First, when you lod your pretrained parameters, turn those parameters into a list. You will be iterating through the list later. Build a for loop to iterate through your new models parameters. At each iteration of the for loop, set the layer_name and weights of the new model to be equal to the name and parameters of the pretrained model (which was turned into a list prior to the for loop). Iterate through the layers of the new model until the end. You should have a pretrained model-imported new model now.

```
pre_trained_model=torch.load("Path to the .pth file")
new=list(pre_trained.items())

my_model_kvpair=mymodel.state_dict()
count=0
for key,value in my_model_kvpair.item():
    layer_name,weights=new[count]
    mymodel_kvpair[key]=weights
    count+=1
```

In [ ]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torchvision import models
import matplotlib.pyplot as plt
%matplotlib inline

import trainer
from utils import ARGS
from simple_cnn import SimpleCNN
from voc_dataset import VOCDataset


# Pre-trained weights up to second-to-last layer
# final layers should be initialized from scratcH!
class PretrainedResNet(nn.Module):
    def __init__(self):
        super().__init__()
        # load resnet model
        self.modelres = models.resnet18(pretrained = True)
        for params in self.modelres.parameters():
            params.requires_grad = False

        self.model= nn.Sequential(self.modelres,nn.Linear(1000,20,bias=True))

    def forward(self, x):
        return self.model(x)
```

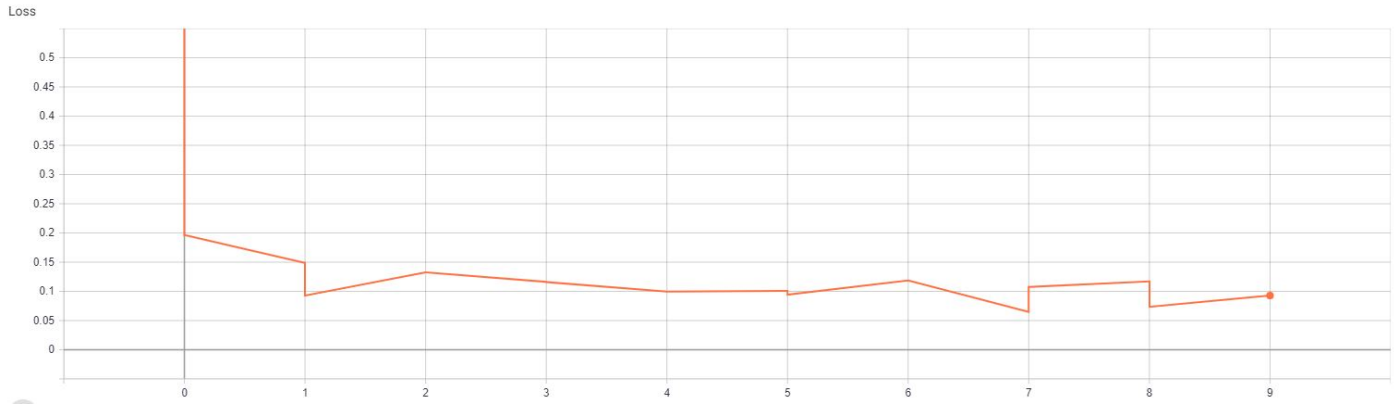
Use similar hyperparameter setup as in the scratch case. Show the learning curves (training loss, testing MAP) for 10 epochs. Please evaluate your model to calculate the MAP on the testing dataset every 100 iterations.

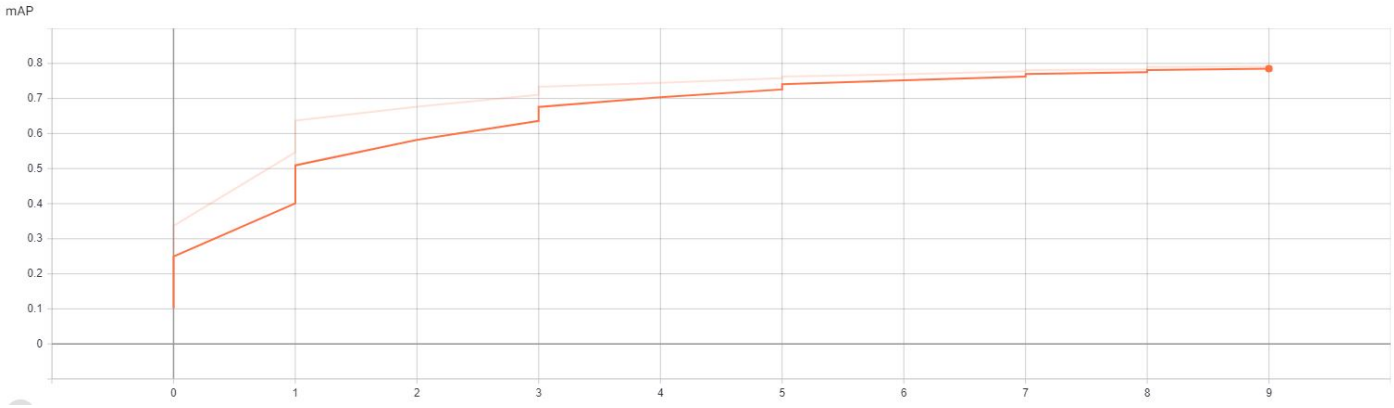
REMEMBER TO SAVE MODEL AT END OF TRAINING

In [ ]:
```
args = ARGS(batch_size = 32, epochs=10, lr = 0.0001)
args.gamma = 0.5
weightDecay = 5e-5
model = PretrainedResNet()
optimizer = torch.optim.Adam(model.parameters(), lr = args.lr, weight_decay =
weightDecay)
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=15, gamma=arg
s.gamma)
if __name__ == '__main__':
    test_ap, test_map = trainer.train(args, model, optimizer, scheduler)
    print('test map:', test_map)
```

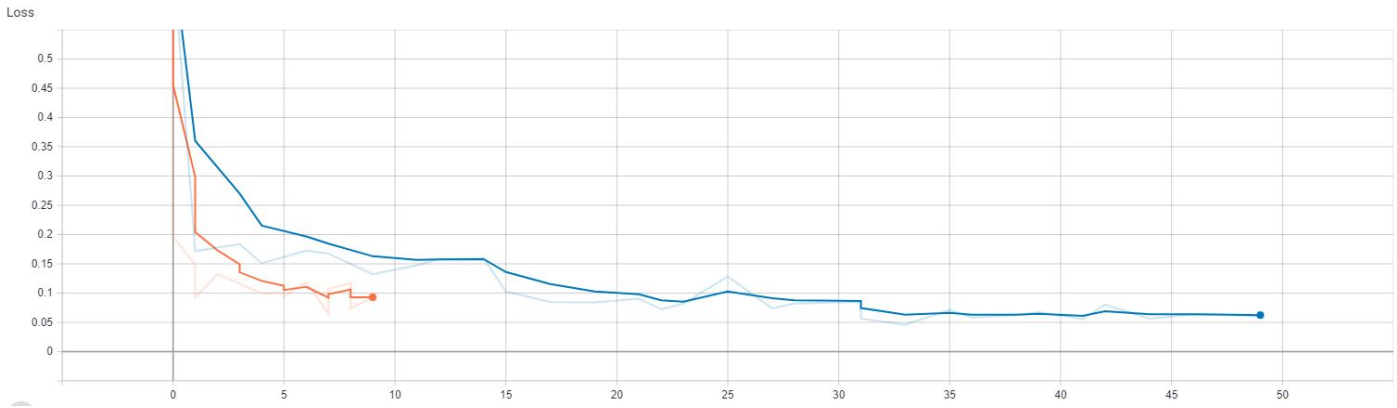
YOUR TB SCREENSHOTS HERE

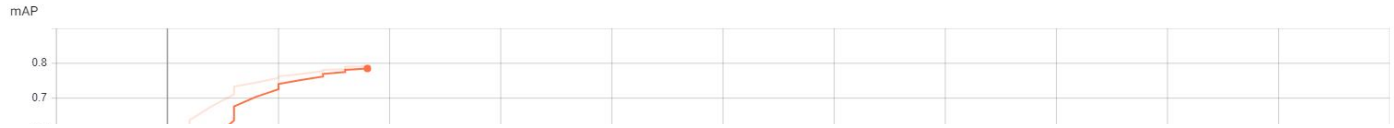
The following two figures shown below display the training loss and training mAP for the ResNet finetuned model.





The following two figures shown below compared the finetuned ResNet model with the ResNet model that was trained from scratch. The curves belonging to the transfer learning ResNet model are colored in light blue.





Appendex A: Training Epoch Information

The following log shown below displays the batch iteration, loss, and mAP calculation at various points during the training process:

Train Epoch: 0 [0 (0%)] Loss: 0.904385 | mAP: 0.112697
Train Epoch: 0 [100 (64%)] Loss: 0.246686 | mAP: 0.318172
Train Epoch: 1 [200 (27%)] Loss: 0.145483 | mAP: 0.512955
Train Epoch: 1 [300 (91%)] Loss: 0.148025 | mAP: 0.619263
Train Epoch: 2 [400 (55%)] Loss: 0.127986 | mAP: 0.674601
Train Epoch: 3 [500 (18%)] Loss: 0.116205 | mAP: 0.710900
Train Epoch: 3 [600 (82%)] Loss: 0.118125 | mAP: 0.730045
Train Epoch: 4 [700 (46%)] Loss: 0.111438 | mAP: 0.739866
Train Epoch: 5 [800 (10%)] Loss: 0.120211 | mAP: 0.755548
Train Epoch: 5 [900 (73%)] Loss: 0.099371 | mAP: 0.761425
Train Epoch: 6 [1000 (37%)] Loss: 0.103049 | mAP: 0.767812
Train Epoch: 7 [1100 (1%)] Loss: 0.095842 | mAP: 0.778130
Train Epoch: 7 [1200 (64%)] Loss: 0.111486 | mAP: 0.776830
Train Epoch: 8 [1300 (28%)] Loss: 0.082349 | mAP: 0.783933
Train Epoch: 8 [1400 (92%)] Loss: 0.100563 | mAP: 0.788158
Train Epoch: 9 [1500 (55%)] Loss: 0.095881 | mAP: 0.792471
test map: 0.7933945926532152

Q5: Analysis (20 points)

By now you should know how to train networks from scratch or using from pre-trained models. You should also understand the relative performance in either scenarios. Needless to say, the performance of these models is stronger than previous non-deep architectures used until 2012. However, final performance is not the only metric we care about. It is important to get some intuition of what these models are really learning. Lets try some standard techniques.

FEEL FREE TO WRITE UTIL CODE IN ANOTHER FILE AND IMPORT IN THIS NOTEBOOK FOR EASE OF READABILITY

5.1 Nearest Neighbors (7 pts)

Pick 3 images from PASCAL test set from different classes, and compute 4 nearest neighbors of those images over the test set. You should use and compare the following feature representations for the nearest neighbors:

1. fc7 features from the ResNet (finetuned from ImageNet)
2. pool5 features from the CaffeNet (trained from scratch)

CaffeNet 4 Nearest Neighbors

In [ ]:

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torchvision import models
import matplotlib.pyplot as plt
%matplotlib inline

import trainer
from utils import ARGS
from simple_cnn import SimpleCNN
from voc_dataset import VOCDataset

import os
import numpy as np

#load CaffeNet model
def get_fc(inp_dim, out_dim, non_linear='relu'):
    """
    Mid-level API. It is useful to customize your own for large code repo.
    :param inp_dim: int, intput dimension
    :param out_dim: int, output dimension
    :param non_linear: str, 'relu', 'softmax'
    :return: list of layers [FC(inp_dim, out_dim), (non linear layer)]
    """
    layers = []
    layers.append(nn.Linear(inp_dim, out_dim))
    if non_linear == 'relu':
        layers.append(nn.ReLU())
    elif non_linear == 'softmax':
        layers.append(nn.Softmax(dim=1))
    elif non_linear == 'none':
        pass
    else:
        raise NotImplementedError
    return layers

class CaffeNet(nn.Module):
    def __init__(self):
        super().__init__()
        c_dim = 3
        self.conv1 = nn.Conv2d(c_dim,96,11,4,padding=0) # valid padding
        self.pool1 = nn.MaxPool2d(3,2)
        self.conv2 = nn.Conv2d(96, 256, 5,padding=2) # same padding
        self.pool2 = nn.MaxPool2d(3,2)
        self.conv3 = nn.Conv2d(256,384,3,padding=1) # same padding
        self.conv4 = nn.Conv2d(384,384,3,padding=1) # same padding
        self.conv5 = nn.Conv2d(384,256,3,padding=1) # same padding
        self.pool3 = nn.MaxPool2d(3,2)
        self.flat_dim = 5*5*256 # replace with the actual value
        self.fc1 = nn.Sequential(*get_fc(self.flat_dim, 4096, 'relu'))
        self.dropout1 = nn.Dropout(p=0.5)
        self.fc2 = nn.Sequential(*get_fc(4096, 4096, 'relu'))
        self.dropout2 = nn.Dropout(p=0.5)
        self.fc3 = nn.Sequential(*get_fc(4096, 20, 'none'))

        self.nonlinear = lambda x: torch.clamp(x,0)
```

```
    def forward(self, x):
        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)

        x = self.conv3(x)
        x = self.nonlinear(x)
        x = self.conv4(x)
        x = self.nonlinear(x)
        x = self.conv5(x)
        x = self.nonlinear(x)
        x = self.pool3(x)
        x = x.view(N, self.flat_dim) # flatten the array

        out = self.fc1(x)
        out = self.nonlinear(out)
        out = self.dropout1(out)
        out = self.fc2(out)
        out = self.nonlinear(out)
        out = self.dropout2(out)
        out = self.fc3(out)

        return x
```

In [ ]:

```
import utils
from sklearn.neighbors import KNeighborsClassifier

args = ARGS(batch_size = 32, use_cuda = True)

if __name__ == "__main__":
    modelCaffe = CaffeNet()
    torchLoadCaffe = torch.load('saved_models/CaffeNet-50.pth')
    modelCaffe.load_state_dict(torchLoadCaffe['model_state_dict'])
    modelCaffe = modelCaffe.to(args.device)
    modelCaffe.eval()
    testfindex, testOutput, testTarget = trainer.train_output_CaffeNet(modelCa
ffe, args)

    # build k- nearest neighbors
    clf = KNeighborsClassifier(n_neighbors = 4)
    clf.fit(testOutput, testfindex)
```

In [ ]:
```
vocTest = VOCDataset(split='test',size=64)
# how big are train and test sets?
print("Test Set is: ", testOutput.shape)

# select three random images from the test set
num = 3
indexArr = []
sampleTestOutputArr = np.ones((1,6400))
sampleTestfindexArr = np.ones((1,1))

for i in range(num):
    randNum = int(np.random.rand()*len(testOutput))
    if randNum not in indexArr:
        indexArr.append(randNum)
        sampleTestOutputArr = np.concatenate((sampleTestOutputArr,testOutput[r
andNum][np.newaxis,:]))
        sampleTestfindexArr = np.concatenate((sampleTestfindexArr,testfindex[r
andNum][np.newaxis,:]))

sampleTestOutputArr = sampleTestOutputArr[1:,:]
sampleTestfindexArr = sampleTestfindexArr[1:,:]

print("Size of Test Sample Output is: ", sampleTestOutputArr.shape)
print("Size of Test Sample Output is: ", sampleTestfindexArr.shape)
```

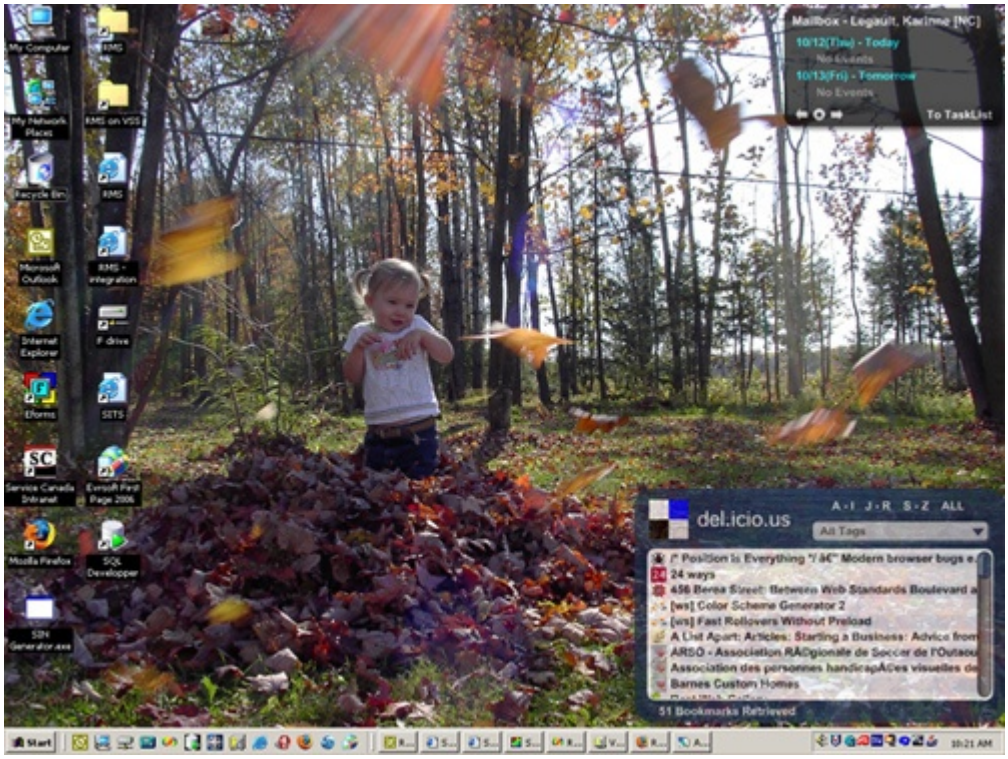
In [ ]:
```
# choose four values between 1 and 5011
testPred = clf.kneighbors(sampleTestOutputArr, return_distance=False)
print("The input is: ", np.transpose(sampleTestfindexArr))
# print("The prediction is: ",testPred)

numVal = 3

neighborImageIndex = np.zeros((numVal,4))

for i in range(numVal):
    for j in range(4):
        neighborImageIndex[i,j] = vocTest.index_list[testPred[i,j]]


print("The prediction is: ", neighborImageIndex)
```

Sample prediction output:

```
The input is:  [['008515' '008131' '006491']]
The prediction is:  [[8515. 2016.  562. 3659.]
 [8131. 8356. 3707. 3514.]
 [6491. 1089. 5673. 5048.]]
```

For the input image shown below ('008515.jpg'):

The follow 4 neighboring images were given:

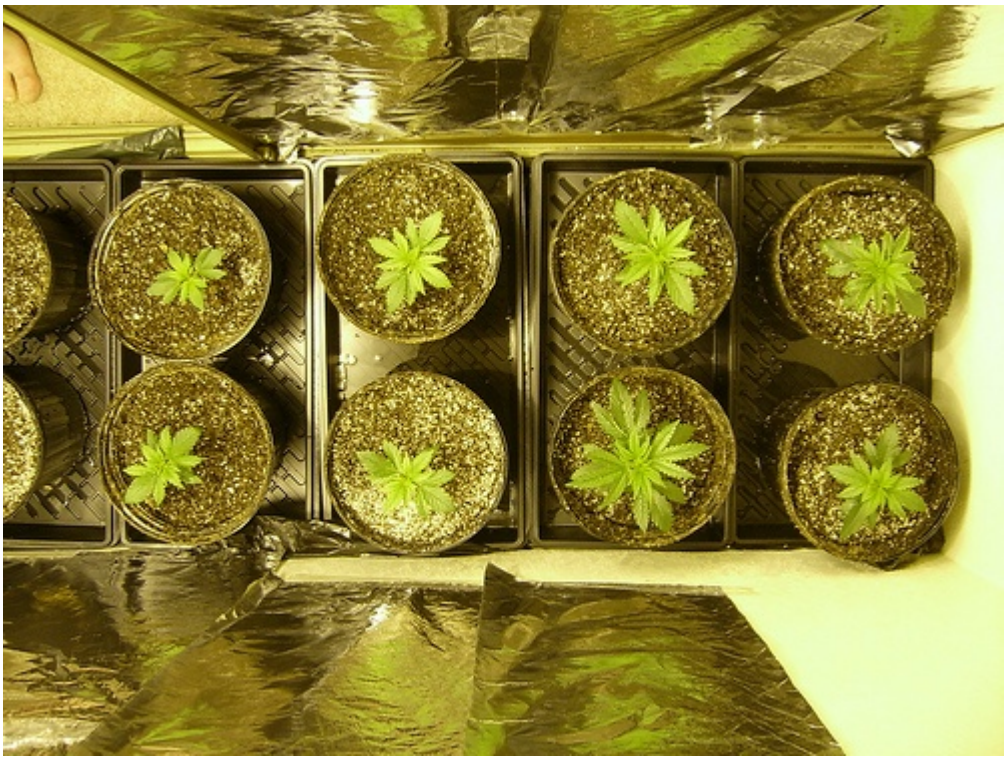


For the input image shown below ('008131.jpg'):

The follow 4 neighboring images were given:

For the input image shown below ('006491.jpg'):

The follow 4 neighboring images were given:



ResNet 4 Nearest Neighbors Section

In [ ]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torchvision import models
import matplotlib.pyplot as plt
%matplotlib inline

import trainer
from utils import ARGS
from simple_cnn import SimpleCNN
from voc_dataset import VOCDataset

#
import os
import numpy as np
```

In [ ]:
```
# Pre-trained weights up to second-to-last layer
# final layers should be initialized from scratcH!
class PretrainedResNet(nn.Module):
    def __init__(self):
        super().__init__()
        # load resnet model
        self.modelres = models.resnet18(pretrained = True)
        for params in self.modelres.parameters():
            params.requires_grad = False

        self.model= nn.Sequential(self.modelres,nn.Linear(1000,20,bias=True))

    def forward(self, x):
        return self.model(x)
```

In [ ]:
```
import utils
from sklearn.neighbors import KNeighborsClassifier

args = ARGS(batch_size = 16, use_cuda = True)

# Create ResNet model
if __name__ == '__main__':
    m = PretrainedResNet()
    model = m.model
    model = torch.load('saved_models/PreTrainedResNet-Model.pth')
    #model.load_state_dict(torch.load('q4_resnet_pretrained_statedict.pth'))
    model = model.to(args.device)
    testfindex, testOutput, testTarget = trainer.train_output_ResNet(model, ar
gs)

    clf = KNeighborsClassifier(n_neighbors = 4)
    clf.fit(testOutput, testfindex)
```

In [ ]:
```
vocTest = VOCDataset(split='test',size=64)

# how big are train and test sets?
print("Test Set is: ", testOutput.shape)

# select three random images from the test set
num = 3
indexArr = []
sampleTestOutputArr = np.ones((1,512))
sampleTestfindexArr = np.ones((1,1))

for i in range(num):
    randNum = int(np.random.rand()*len(testOutput))
    if randNum not in indexArr:
        indexArr.append(randNum)
        sampleTestOutputArr = np.concatenate((sampleTestOutputArr,testOutput[r
andNum][np.newaxis,:]))
        sampleTestfindexArr = np.concatenate((sampleTestfindexArr,testfindex[r
andNum][np.newaxis,:]))

sampleTestOutputArr = sampleTestOutputArr[1:,:]
sampleTestfindexArr = sampleTestfindexArr[1:,:]

print("Size of Test Sample Output is: ", sampleTestOutputArr.shape)
print("Size of Test Sample Output is: ", sampleTestfindexArr.shape)
```

In [ ]:

```
# choose four values between 1 and 5011
testPred = clf.kneighbors(sampleTestOutputArr, return_distance=False)
print("The input is: ", np.transpose(sampleTestfindexArr))
# print("The prediction is: ",testPred)

numVal = 3

neighborImageIndex = np.zeros((numVal,4))

for i in range(numVal):
    for j in range(4):
        neighborImageIndex[i,j] = vocTest.index_list[testPred[i,j]]


print("The prediction is: ", neighborImageIndex)
```

Sample prediction output:

```
The input is:  [['000517' '003326' '001374']]
The prediction is:  [[ 517. 6194.  205. 5196.]
 [3326. 3246. 3977. 1000.]
 [1374. 9521. 9632. 6422.]]
```

For the input image shown below ('000517.jpg'):

The follow 4 neighboring images were given:

For the input image shown below ('003326.jpg'):

The follow 4 neighboring images were given:


swell

For the input image shown below ('001374.jpg'):

The follow 4 neighboring images were given:

5.2 t-SNE visualization of intermediate features (7pts)

We can also visualize how the feature representations specialize for different classes. Take 1000 random images from the test set of PASCAL, and extract caffenet (scratch) fc7 features from those images. Compute a 2D t-SNE projection of the features, and plot them with each feature color coded by the GT class of the corresponding image. If multiple objects are active in that image, compute the color as the ”mean” color of the different classes active in that image. Legend the graph with the colors for each object class.

CaffeNet t-SNE

In [ ]:

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torchvision import models
import matplotlib.pyplot as plt
%matplotlib inline

import trainer
from utils import ARGS
from simple_cnn import SimpleCNN
from voc_dataset import VOCDataset

import os
import numpy as np

#load CaffeNet model
def get_fc(inp_dim, out_dim, non_linear='relu'):
    """
    Mid-level API. It is useful to customize your own for large code repo.
    :param inp_dim: int, intput dimension
    :param out_dim: int, output dimension
    :param non_linear: str, 'relu', 'softmax'
    :return: list of layers [FC(inp_dim, out_dim), (non linear layer)]
    """
    layers = []
    layers.append(nn.Linear(inp_dim, out_dim))
    if non_linear == 'relu':
        layers.append(nn.ReLU())
    elif non_linear == 'softmax':
        layers.append(nn.Softmax(dim=1))
    elif non_linear == 'none':
        pass
    else:
        raise NotImplementedError
    return layers

class CaffeNet(nn.Module):
    def __init__(self):
        super().__init__()
        c_dim = 3
        self.conv1 = nn.Conv2d(c_dim,96,11,4,padding=0) # valid padding
        self.pool1 = nn.MaxPool2d(3,2)
        self.conv2 = nn.Conv2d(96, 256, 5,padding=2) # same padding
        self.pool2 = nn.MaxPool2d(3,2)
        self.conv3 = nn.Conv2d(256,384,3,padding=1) # same padding
        self.conv4 = nn.Conv2d(384,384,3,padding=1) # same padding
        self.conv5 = nn.Conv2d(384,256,3,padding=1) # same padding
        self.pool3 = nn.MaxPool2d(3,2)
        self.flat_dim = 5*5*256 # replace with the actual value
        self.fc1 = nn.Sequential(*get_fc(self.flat_dim, 4096, 'relu'))
        self.dropout1 = nn.Dropout(p=0.5)
        self.fc2 = nn.Sequential(*get_fc(4096, 4096, 'relu'))
        self.dropout2 = nn.Dropout(p=0.5)
        self.fc3 = nn.Sequential(*get_fc(4096, 20, 'none'))

        self.nonlinear = lambda x: torch.clamp(x,0)
```

```
    def forward(self, x):
        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)

        x = self.conv3(x)
        x = self.nonlinear(x)
        x = self.conv4(x)
        x = self.nonlinear(x)
        x = self.conv5(x)
        x = self.nonlinear(x)
        x = self.pool3(x)
        x = x.view(N, self.flat_dim) # flatten the array

        out = self.fc1(x)
        out = self.nonlinear(out)
        out = self.dropout1(out)
        out = self.fc2(out)
        out1 = out
        out = self.nonlinear(out)
        out = self.dropout2(out)
        out = self.fc3(out)

        return out1
```

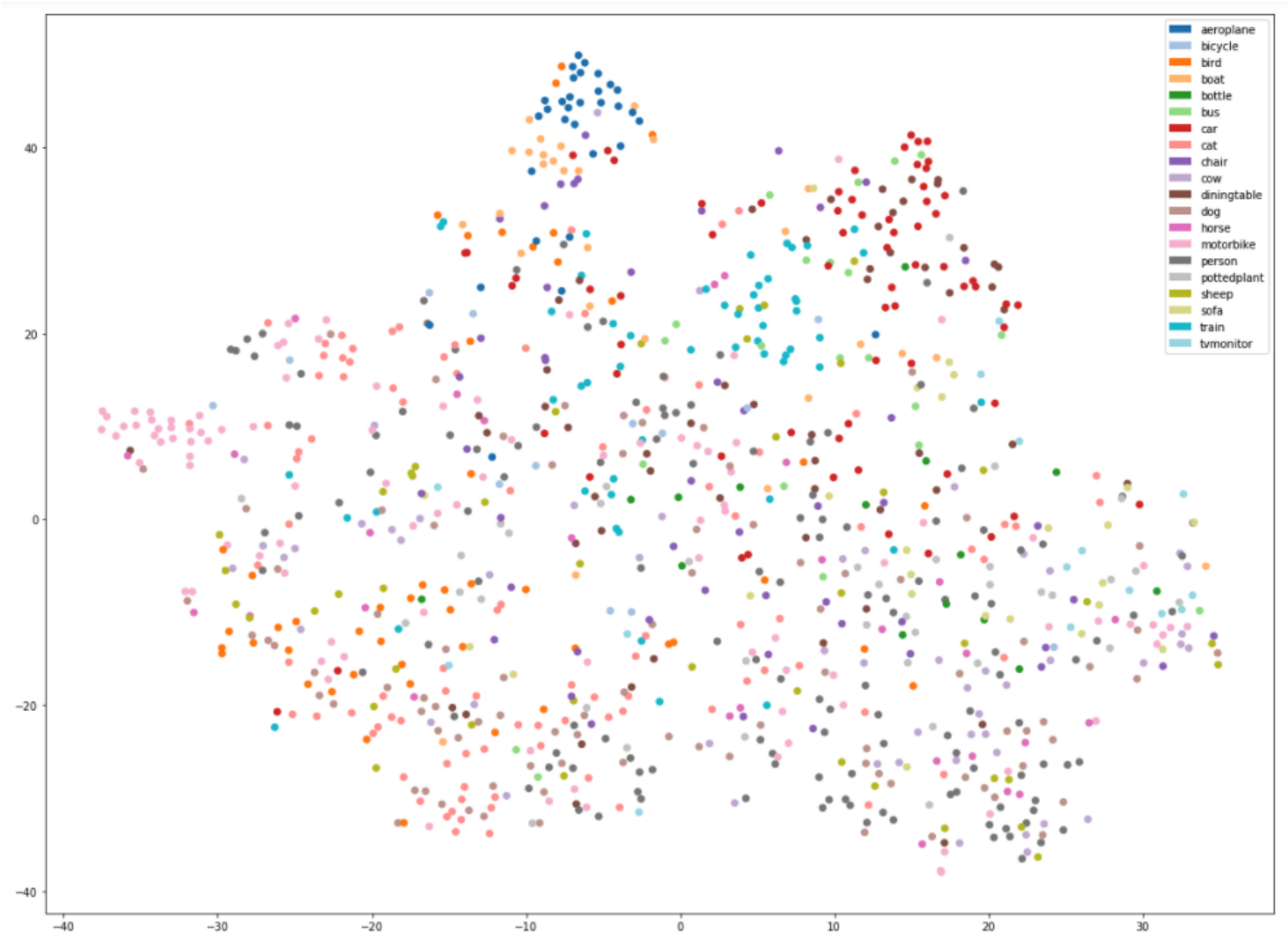
In [ ]:
```
if __name__ == '__main__':
    modelCaffe = CaffeNet()
    torchLoadCaffe = torch.load('saved_models/CaffeNet-50.pth')
    modelCaffe.load_state_dict(torchLoadCaffe['model_state_dict'])
    modelCaffe = modelCaffe.to(args.device)
    modelCaffe.eval()
    testfindex, testOutput, testTarget = trainer.train_output_CaffeNet(modelCa
ffe, args)
```

In [ ]:
```
# define "color" averaging function
def avg_color_label(labels):
    oneHotSum = np.sum(labels,axis=1)
    indexHotSum = np.sum(labels*np.arange(1,21)[np.newaxis,:],axis=1)

    returnVect = indexHotSum/oneHotSum


    return returnVect[:,np.newaxis]
```

In [ ]:
```
import numpy as np
from sklearn.manifold import TSNE

# select 1000 random images from the test set
num = 1000
indexArr = []
sneTestOutputArr = np.ones((1,6400))
sneTestTargetArr = np.ones((1,20))

while len(indexArr) < 1000:
    randNum = int(np.random.rand()*len(testOutput))
    if randNum not in indexArr:
        indexArr.append(randNum)
        sneTestOutputArr = np.concatenate((sneTestOutputArr,testOutput[randNum
][np.newaxis,:]))
        sneTestTargetArr = np.concatenate((sneTestTargetArr,testTarget[randNum
][np.newaxis,:]))
    i += 1

sneTestOutputArr = sneTestOutputArr[1:,:]
sneTestTargetArr = sneTestTargetArr[1:,:]

x_inbedded = TSNE(n_components=2).fit_transform(sneTestOutputArr)
```

In [ ]:
```
import matplotlib

meanLabels = avg_color_label(sneTestTargetArr)
#meanLabels = meanLabels/np.amax(meanLabels)
meanLabels = np.squeeze(meanLabels,axis=1)
plt.figure(figsize=(20,15))
cmap = plt.get_cmap('tab20')
recs = []
legend = ['aeroplane', 'bicycle', 'bird', 'boat', 'bottle', 'bus', 'car',
                  'cat', 'chair', 'cow', 'diningtable', 'dog', 'horse', 'moto
rbike',
                  'person', 'pottedplant', 'sheep', 'sofa', 'train', 'tvmonit
or']
plt.scatter(x_inbedded[:,0],x_inbedded[:,1],c = meanLabels, cmap = plt.get_cma
p('tab20'))
for i in range(0,len(meanLabels)):
    recs.append(matplotlib.patches.Rectangle((0,0),1,1,fc=cmap(i)))
plt.legend(recs,legend,loc=1)
```

The output plot is shown as the following figure below:



5.3 Are some classes harder? (6pts)

Show the per-class performance of your caffenet (scratch) and ResNet (finetuned) models. Try to explain, by observing examples from the dataset, why some classes are harder or easier than the others (consider the easiest and hardest class). Do some classes see large gains due to pre-training? Can you explain why that might happen?

YOUR ANSWER HERE

CaffeNet and PreTrained ResNet

The class that had the highest AP performance was the "person" while the class that had the lowest performance was the "bottle".

There can be many reasons as to why the "person" performs the best. Two images of the class are shown below for reference. One reason could be that the features of a person are the most distinct from the other classes in the dataset such as "cow" or "aeroplane". In addition, a nondifficult image of a person in the dataset exposes a lot of features such as the upright position, arms, legs, facial features, colored attire, which can help to differentiate it from the other images in the dataset during training.

Two images of the class "bottle" are shown below for reference. Compared to the features of a human, there a significantly less features of a bottle for the model to detect. In some cases, the background may make the profile harder to detect, as shown in the image below where a close-up picture was taken of some wine bottles in a dark room. In addition, features of bottles may be more and more difficult to detect as the object is further away from the camera. For example, the label of a bottle may be easier to extract as a feature when the camera is close-up, but when the bottle is a couple of meters away from the camera, then the label would be difficult for the model to spot.


Farmer's
Daughter
2004

TreTrained ResNet vs From-Scratch ResNet

For reference, the per-class performance of the From-Scratch ResNet model is shown below:

aeroplane: 0.695663796029232
bicycle: 0.5622480632184709
bird: 0.41245039413080287
boat: 0.5710383505097775
bottle: 0.15982507088326053
bus: 0.5238514422134894
car: 0.7453615983966118
cat: 0.46971015685819406
chair: 0.4283246799656059
cow: 0.28834160996828456
diningtable: 0.40786228993246554
dog: 0.3652161054748284
horse: 0.7533615316892681
motorbike: 0.6526196860641374
person: 0.8447806695750432
pottedplant: 0.2523940858888379
sheep: 0.38764743196135676

sofa: 0.3606603351942273
train: 0.7182136028211066
tvmonitor: 0.4197671608743151

Though most classes saw some level of performance gain by using the transfer learning approach, the classes that saw the most significant gain were the cow, dog, and sheep classes. The gain percentage was abou 3-5% in the pretrained model. One reason for this specific gain is attributed to that the model was getting better at differentiating between the three classes. As seen through some iteration with the 4 nearest neighbors model, cows, dogs, and sheeps were frequently predicted between each other, probably due to the similar features and shape between classes. Using a transfer learning approached allow the model to run through another set of iterations to differentiate between the classes. Freezing the previous layers also allowed the model to focus on training on layer of the model, which proved to be successful in gaining performance compared to its From-Scratch counterpart.

CaffeNet Class Performance Analysis

In [ ]:

```
# redefine CaffeNet without the pool5 output
class CaffeNet(nn.Module):
    def __init__(self):
        super().__init__()
        c_dim = 3
        self.conv1 = nn.Conv2d(c_dim,96,11,4,padding=0) # valid padding
        self.pool1 = nn.MaxPool2d(3,2)
        self.conv2 = nn.Conv2d(96, 256, 5,padding=2) # same padding
        self.pool2 = nn.MaxPool2d(3,2)
        self.conv3 = nn.Conv2d(256,384,3,padding=1) # same padding
        self.conv4 = nn.Conv2d(384,384,3,padding=1) # same padding
        self.conv5 = nn.Conv2d(384,256,3,padding=1) # same padding
        self.pool3 = nn.MaxPool2d(3,2)
        self.flat_dim = 5*5*256 # replace with the actual value
        self.fc1 = nn.Sequential(*get_fc(self.flat_dim, 4096, 'relu'))
        self.dropout1 = nn.Dropout(p=0.5)
        self.fc2 = nn.Sequential(*get_fc(4096, 4096, 'relu'))
        self.dropout2 = nn.Dropout(p=0.5)
        self.fc3 = nn.Sequential(*get_fc(4096, 20, 'none'))

        self.nonlinear = lambda x: torch.clamp(x,0)

    def forward(self, x):
        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)

        x = self.conv3(x)
        x = self.nonlinear(x)
        x = self.conv4(x)
        x = self.nonlinear(x)
        x = self.conv5(x)
        x = self.nonlinear(x)
        x = self.pool3(x)
        x = x.view(N, self.flat_dim) # flatten the array

        out = self.fc1(x)
        out = self.nonlinear(out)
        out = self.dropout1(out)
        out = self.fc2(out)
        out = self.nonlinear(out)
        out = self.dropout2(out)
        out = self.fc3(out)

        return out
```

In [ ]:
```
# output performance for each class

if __name__ == '__main__':
    modelCaffe = CaffeNet()
    torchLoadCaffe = torch.load('saved_models/CaffeNet-50.pth')
    modelCaffe.load_state_dict(torchLoadCaffe['model_state_dict'])
    modelCaffe = modelCaffe.to(args.device)
    modelCaffe.eval()

    test_loader = utils.get_data_loader('voc', train=False, batch_size=args.te
st_batch_size, split='test')
    ap, map = utils.eval_dataset_map(modelCaffe, args.device, test_loader)


    # output values from CaffeNet, compare with the target values
    classNames = ['aeroplane', 'bicycle', 'bird', 'boat', 'bottle', 'bus', 'ca
r',
                          'cat', 'chair', 'cow', 'diningtable', 'dog', 'horse',
'motorbike',
                          'person', 'pottedplant', 'sheep', 'sofa', 'train', 'tvm
onitor']

    print("Accuracy Precision of CaffeNet Among Individual Classes: ")
    for i in range(len(classNames)):
        print("{}: {}".format(classNames[i],ap[i]))
```

Accuracy Precision of CaffeNet Among Individual Classes:
aeroplane: 0.6581140150639725
bicycle: 0.4561887044476487
bird: 0.35825532448861536
boat: 0.5089389243663608
bottle: 0.16395719924468152
bus: 0.3739061674684568
car: 0.6706960925944321
cat: 0.38321888683805233
chair: 0.39942472719290306
cow: 0.2317710070030363
diningtable: 0.29885085506558173
dog: 0.3274386268412313
horse: 0.6648294880873716
motorbike: 0.5684288971068936
person: 0.8086725636680325
pottedplant: 0.23318258506171002
sheep: 0.25812258618853845
sofa: 0.3496855596933779
train: 0.5634974187463156
tvmonitor: 0.36742932492025593

mAP:

0.4322304477043734

ResNet Class Performance Analysis

In [ ]:
```
if __name__ == '__main__':
    modelRes.eval()
    test_loader = utils.get_data_loader('voc', train=False, batch_size=args.te
st_batch_size, split='test')
    ap, map = utils.eval_dataset_map(modelRes, args.device, test_loader)


    # output values from CaffeNet, compare with the target values
    classNames = ['aeroplane', 'bicycle', 'bird', 'boat', 'bottle', 'bus', 'ca
r',
                          'cat', 'chair', 'cow', 'diningtable', 'dog', 'horse',
'motorbike',
                          'person', 'pottedplant', 'sheep', 'sofa', 'train', 'tvm
onitor']

    print("Accuracy Precision of PreTrained ResNet Among Individual Classes: "
)
    for i in range(len(classNames)):
        print("{}: {}".format(classNames[i],ap[i]))
```

Accuracy Precision of PreTrained ResNet Among Individual Classes:
aeroplane: 0.7213222117720576
bicycle: 0.5694538600068458
bird: 0.4367706344884813
boat: 0.5773203950369964
bottle: 0.1818182675529317
bus: 0.5025989148818082
car: 0.7382160645945922
cat: 0.4581863265778226
chair: 0.43791839747272837
cow: 0.32130702694874635
diningtable: 0.4435263229389825
dog: 0.4043789661300173
horse: 0.7495443845590672
motorbike: 0.6876729041809619
person: 0.854338868879449
pottedplant: 0.2733504707389632
sheep: 0.4169879431619927
sofa: 0.3385033253895445
train: 0.6906721981250176
tvmonitor: 0.46132828158342576

mAP:

0.5132607882510216