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

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

Fashion MNIST is a dataset of Zalando's 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 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.

Prerequisites:

- Install conda and create a conda environment to manage all the packages for the homeworks.
- Install the following packages in your conda environment:
  - jupyterlab and get familiar with basic operations on jupyter notebook.
  - Pytorch
  - matplotlib
  - tensorboard
  - imageio
  - sklearn

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 hav
```

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 = False

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()
            pred = output.argmax(dim=1, keepdim=True)  # get the index of the ma
            correct += pred.eq(target.view_as(pred)).sum().item()

    test_loss /= len(test_loader.dataset)

    print('\nTest set: Average loss: {:.4f}, Accuracy: {}/{} ({:.0f}%)\n'.format
        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.Compo
                            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=ar
    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()
            # 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, args.batch_size * batch_idx, len(train_loader.dataset
                            100. * batch_idx / len(train_loader), loss.item()))
                train_log['iter'].append(cnt)
                train_log['loss'].append(loss.cpu().detach().numpy())
                # TODO: q0.1 calculate your train accuracy!
                pred = output.argmax(dim=1, keepdim=True)  # get the index of th
                correct = pred.eq(target.view_as(pred)).sum().item()
                train_acc = correct / data.size()[0]
                train_log['accuracy'].append(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`!

- (1 pt) Check out `TODO: q0.1` in `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.
- (1 pt) Fill in the `TODO: q0.1` in the `main()` function above to calculated the train accuracy.

Once you fix the bugs, you should be able to get a reasonable accuracy (>80%) 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)

The default hyperparameters actually worked pretty well, but since the loss was fluctuating near the end I decreased gamma. I hoped that this would give a more monotonic increase in performance and it sort of did.

In [61]:

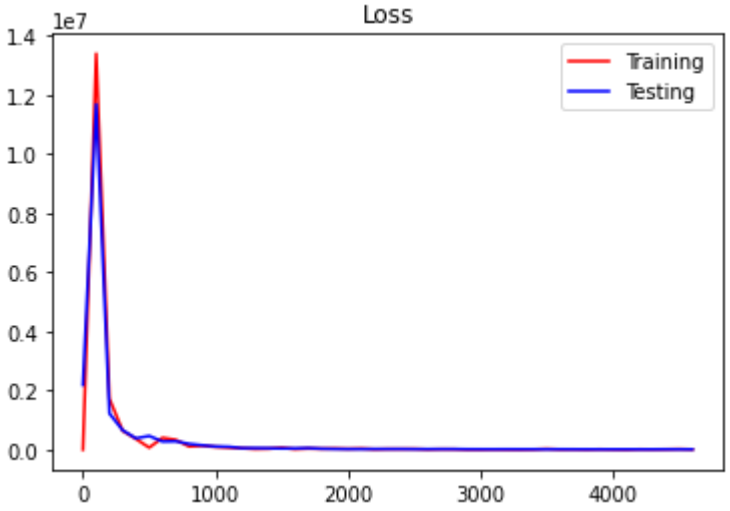
```
#### FEEL FREE TO MODIFY args VARIABLE HERE OR ABOVE ####
args.lr = 1.0
args.gamma = 0.3
args.epochs = 5
args.batch_size = 64
args.log_every = 100
args.val_every = 100


# 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()
```

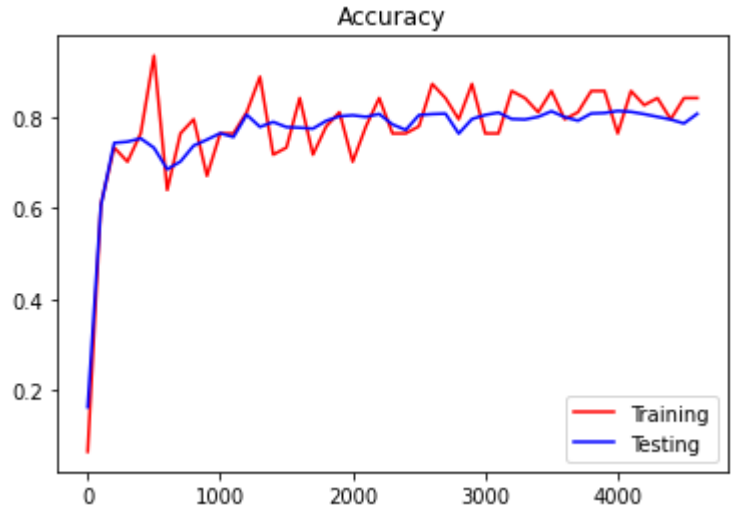
```
args.batch_size = 64
args.epochs = 5
args.gamma = 0.3
args.log_every = 100
args.lr = 1.0
args.test_batch_size = 1000
args.val_every = 100



Train Epoch: 0 [0/60000 (0%)]   Loss: 2.318530

Test set: Average loss: 2189613.8240, Accuracy: 1609/10000 (16%)

Train Epoch: 0 [6400/60000 (11%)]       Loss: 13386178.000000

Test set: Average loss: 11673823.4368, Accuracy: 6067/10000 (61%)

Train Epoch: 0 [12800/60000 (21%)]      Loss: 1719941.000000

Test set: Average loss: 1224248.9856, Accuracy: 7446/10000 (74%)

Train Epoch: 0 [19200/60000 (32%)]      Loss: 636349.812500

Test set: Average loss: 655507.6864, Accuracy: 7473/10000 (75%)

Train Epoch: 0 [25600/60000 (43%)]      Loss: 363832.406250
```

```
Test set: Average loss: 383846.1024, Accuracy: 7546/10000 (75%)

Train Epoch: 0 [32000/60000 (53%)]      Loss: 69688.148438

Test set: Average loss: 464885.6224, Accuracy: 7339/10000 (73%)

Train Epoch: 0 [38400/60000 (64%)]      Loss: 401374.250000

Test set: Average loss: 277241.0608, Accuracy: 6864/10000 (69%)

Train Epoch: 0 [44800/60000 (75%)]      Loss: 334800.968750

Test set: Average loss: 283876.8528, Accuracy: 7035/10000 (70%)

Train Epoch: 0 [51200/60000 (85%)]      Loss: 109731.390625

Test set: Average loss: 196316.7792, Accuracy: 7387/10000 (74%)

Train Epoch: 0 [57600/60000 (96%)]      Loss: 133407.562500

Test set: Average loss: 148866.3368, Accuracy: 7518/10000 (75%)

Train Epoch: 1 [3968/60000 (7%)]        Loss: 93748.500000

Test set: Average loss: 109034.2176, Accuracy: 7660/10000 (77%)

Train Epoch: 1 [10368/60000 (17%)]      Loss: 70121.148438

Test set: Average loss: 98638.0328, Accuracy: 7576/10000 (76%)

Train Epoch: 1 [16768/60000 (28%)]      Loss: 49642.414062

Test set: Average loss: 53517.4368, Accuracy: 8069/10000 (81%)

Train Epoch: 1 [23168/60000 (39%)]      Loss: 23262.658203

Test set: Average loss: 55805.2404, Accuracy: 7800/10000 (78%)

Train Epoch: 1 [29568/60000 (49%)]      Loss: 32026.714844

Test set: Average loss: 50338.3504, Accuracy: 7909/10000 (79%)

Train Epoch: 1 [35968/60000 (60%)]      Loss: 65620.710938

Test set: Average loss: 43007.6904, Accuracy: 7796/10000 (78%)

Train Epoch: 1 [42368/60000 (71%)]      Loss: 18781.445312

Test set: Average loss: 44356.6788, Accuracy: 7782/10000 (78%)

Train Epoch: 1 [48768/60000 (81%)]      Loss: 39106.691406

Test set: Average loss: 55442.0960, Accuracy: 7762/10000 (78%)

Train Epoch: 1 [55168/60000 (92%)]      Loss: 30028.488281

Test set: Average loss: 34690.6118, Accuracy: 7935/10000 (79%)

Train Epoch: 2 [1536/60000 (3%)]        Loss: 36932.484375

Test set: Average loss: 25635.6020, Accuracy: 8035/10000 (80%)

Train Epoch: 2 [7936/60000 (13%)]       Loss: 26927.953125

Test set: Average loss: 21735.3656, Accuracy: 8057/10000 (81%)
```

```
Train Epoch: 2 [14336/60000 (24%)]      Loss: 36507.179688

Test set: Average loss: 21324.9974, Accuracy: 8022/10000 (80%)

Train Epoch: 2 [20736/60000 (35%)]      Loss: 14878.640625

Test set: Average loss: 22970.1110, Accuracy: 8086/10000 (81%)

Train Epoch: 2 [27136/60000 (45%)]      Loss: 22453.406250

Test set: Average loss: 23450.3830, Accuracy: 7855/10000 (79%)

Train Epoch: 2 [33536/60000 (56%)]      Loss: 22506.996094

Test set: Average loss: 23897.4890, Accuracy: 7733/10000 (77%)

Train Epoch: 2 [39936/60000 (67%)]      Loss: 24659.277344

Test set: Average loss: 20044.7288, Accuracy: 8061/10000 (81%)

Train Epoch: 2 [46336/60000 (77%)]      Loss: 10991.498047

Test set: Average loss: 18907.8482, Accuracy: 8084/10000 (81%)

Train Epoch: 2 [52736/60000 (88%)]      Loss: 17623.554688

Test set: Average loss: 21019.6019, Accuracy: 8094/10000 (81%)

Train Epoch: 2 [59136/60000 (99%)]      Loss: 15955.556641

Test set: Average loss: 21142.9298, Accuracy: 7653/10000 (77%)

Train Epoch: 3 [5504/60000 (9%)]        Loss: 5275.152344

Test set: Average loss: 15503.4052, Accuracy: 7974/10000 (80%)

Train Epoch: 3 [11904/60000 (20%)]      Loss: 5469.032227

Test set: Average loss: 14751.4059, Accuracy: 8065/10000 (81%)

Train Epoch: 3 [18304/60000 (30%)]      Loss: 9776.177734

Test set: Average loss: 14135.3546, Accuracy: 8117/10000 (81%)

Train Epoch: 3 [24704/60000 (41%)]      Loss: 6045.632324

Test set: Average loss: 15295.0161, Accuracy: 7975/10000 (80%)

Train Epoch: 3 [31104/60000 (52%)]      Loss: 4615.315430

Test set: Average loss: 14284.6782, Accuracy: 7965/10000 (80%)

Train Epoch: 3 [37504/60000 (62%)]      Loss: 9129.966797

Test set: Average loss: 14465.0921, Accuracy: 8023/10000 (80%)

Train Epoch: 3 [43904/60000 (73%)]      Loss: 25879.271484

Test set: Average loss: 14599.9800, Accuracy: 8146/10000 (81%)

Train Epoch: 3 [50304/60000 (84%)]      Loss: 9197.474609

Test set: Average loss: 14643.8591, Accuracy: 8017/10000 (80%)
```

```
Train Epoch: 3 [56704/60000 (94%)]      Loss: 12123.690430

Test set: Average loss: 12946.9164, Accuracy: 7938/10000 (79%)

Train Epoch: 4 [3072/60000 (5%)]        Loss: 6255.207031

Test set: Average loss: 12342.8591, Accuracy: 8097/10000 (81%)

Train Epoch: 4 [9472/60000 (16%)]       Loss: 13667.074219

Test set: Average loss: 11970.7257, Accuracy: 8108/10000 (81%)

Train Epoch: 4 [15872/60000 (26%)]      Loss: 8054.487305

Test set: Average loss: 11673.4703, Accuracy: 8153/10000 (82%)

Train Epoch: 4 [22272/60000 (37%)]      Loss: 3587.880127

Test set: Average loss: 11830.4021, Accuracy: 8133/10000 (81%)

Train Epoch: 4 [28672/60000 (48%)]      Loss: 13462.431641

Test set: Average loss: 11558.0515, Accuracy: 8083/10000 (81%)

Train Epoch: 4 [35072/60000 (58%)]      Loss: 10829.637695

Test set: Average loss: 11407.6982, Accuracy: 8023/10000 (80%)

Train Epoch: 4 [41472/60000 (69%)]      Loss: 12933.336914

Test set: Average loss: 13602.6295, Accuracy: 7964/10000 (80%)

Train Epoch: 4 [47872/60000 (80%)]      Loss: 20772.351562

Test set: Average loss: 13215.7049, Accuracy: 7877/10000 (79%)

Train Epoch: 4 [54272/60000 (90%)]      Loss: 10423.455078

Test set: Average loss: 11629.9123, Accuracy: 8090/10000 (81%)
```





0.2 Play with parameters.(3pt)

How many trainable parameters does the trained model have? The answer needs to depend on the input `model` - outputting a constant number will not get any credits. Hint: Find out how to use model.parameters() in PyTorch.

In [64]:
```
def param_count(model):
    total_params = 0
    for param in model.parameters():
        prod = 1
        for d in param.data.size():
            prod *= d
        total_params += prod
    return total_params

print('Model has {} params'.format(param_count(model)))
```

```
Model has 454922 params
```

0.3 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 `simple_cnn.py` to add non-linear activation layers, and train your model in full scale. Make sure to add non-linearities at **every** applicable layer.

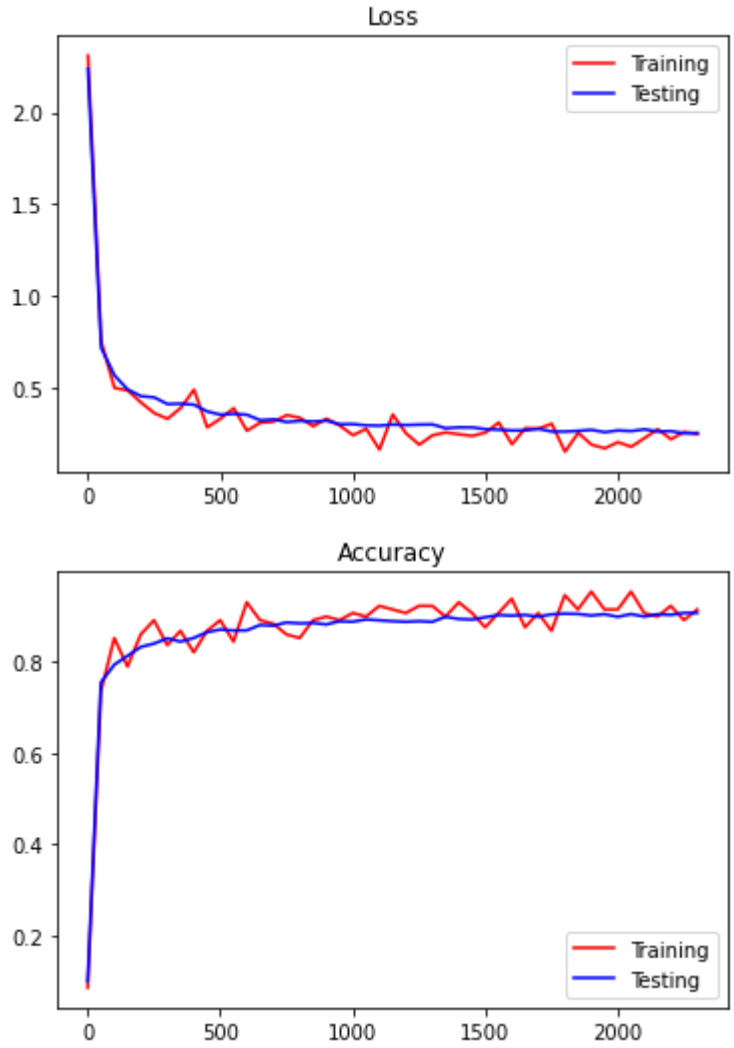
Compute the loss and accuracy curves on train and test sets after 5 epochs. The accuracy should be around 90% or higher.

In [4]:
```
args.epochs = 5
args.lr = 0.001
args.gamma = 0.8
args.batch_size = 128
args.val_every = 50
args.log_every = 50
main()
```

```
Train Epoch: 0 [0/60000 (0%)]   Loss: 2.307297

Test set: Average loss: 2.2358, Accuracy: 1002/10000 (10%)

Train Epoch: 0 [6400/60000 (11%)]       Loss: 0.756638

Test set: Average loss: 0.7218, Accuracy: 7543/10000 (75%)

Train Epoch: 0 [12800/60000 (21%)]      Loss: 0.504415

Test set: Average loss: 0.5724, Accuracy: 7931/10000 (79%)

Train Epoch: 0 [19200/60000 (32%)]      Loss: 0.490513

Test set: Average loss: 0.4954, Accuracy: 8123/10000 (81%)

Train Epoch: 0 [25600/60000 (43%)]      Loss: 0.425394

Test set: Average loss: 0.4596, Accuracy: 8320/10000 (83%)

Train Epoch: 0 [32000/60000 (53%)]      Loss: 0.367680

Test set: Average loss: 0.4529, Accuracy: 8393/10000 (84%)

Train Epoch: 0 [38400/60000 (64%)]      Loss: 0.336352

Test set: Average loss: 0.4166, Accuracy: 8508/10000 (85%)

Train Epoch: 0 [44800/60000 (75%)]      Loss: 0.395595

Test set: Average loss: 0.4188, Accuracy: 8436/10000 (84%)

Train Epoch: 0 [51200/60000 (85%)]      Loss: 0.494616

Test set: Average loss: 0.4132, Accuracy: 8516/10000 (85%)

Train Epoch: 0 [57600/60000 (96%)]      Loss: 0.290082

Test set: Average loss: 0.3765, Accuracy: 8643/10000 (86%)

Train Epoch: 1 [3968/60000 (7%)]        Loss: 0.337396

Test set: Average loss: 0.3589, Accuracy: 8701/10000 (87%)

Train Epoch: 1 [10368/60000 (17%)]      Loss: 0.394273

Test set: Average loss: 0.3638, Accuracy: 8681/10000 (87%)

Train Epoch: 1 [16768/60000 (28%)]      Loss: 0.271810

Test set: Average loss: 0.3584, Accuracy: 8682/10000 (87%)

Train Epoch: 1 [23168/60000 (39%)]      Loss: 0.315684

Test set: Average loss: 0.3289, Accuracy: 8800/10000 (88%)

Train Epoch: 1 [29568/60000 (49%)]      Loss: 0.320857

Test set: Average loss: 0.3336, Accuracy: 8786/10000 (88%)

Train Epoch: 1 [35968/60000 (60%)]      Loss: 0.356442

Test set: Average loss: 0.3190, Accuracy: 8855/10000 (89%)

Train Epoch: 1 [42368/60000 (71%)]      Loss: 0.342018
```

```
Test set: Average loss: 0.3255, Accuracy: 8836/10000 (88%)

Train Epoch: 1 [48768/60000 (81%)]      Loss: 0.296039

Test set: Average loss: 0.3214, Accuracy: 8845/10000 (88%)

Train Epoch: 1 [55168/60000 (92%)]      Loss: 0.336877

Test set: Average loss: 0.3265, Accuracy: 8813/10000 (88%)

Train Epoch: 2 [1536/60000 (3%)]        Loss: 0.299994

Test set: Average loss: 0.3071, Accuracy: 8882/10000 (89%)

Train Epoch: 2 [7936/60000 (13%)]       Loss: 0.247622

Test set: Average loss: 0.3079, Accuracy: 8872/10000 (89%)

Train Epoch: 2 [14336/60000 (24%)]      Loss: 0.282088

Test set: Average loss: 0.3009, Accuracy: 8923/10000 (89%)

Train Epoch: 2 [20736/60000 (35%)]      Loss: 0.169858

Test set: Average loss: 0.2994, Accuracy: 8903/10000 (89%)

Train Epoch: 2 [27136/60000 (45%)]      Loss: 0.359936

Test set: Average loss: 0.3058, Accuracy: 8884/10000 (89%)

Train Epoch: 2 [33536/60000 (56%)]      Loss: 0.257947

Test set: Average loss: 0.3027, Accuracy: 8871/10000 (89%)

Train Epoch: 2 [39936/60000 (67%)]      Loss: 0.196254

Test set: Average loss: 0.3051, Accuracy: 8884/10000 (89%)

Train Epoch: 2 [46336/60000 (77%)]      Loss: 0.247057

Test set: Average loss: 0.3061, Accuracy: 8869/10000 (89%)

Train Epoch: 2 [52736/60000 (88%)]      Loss: 0.262468

Test set: Average loss: 0.2856, Accuracy: 8981/10000 (90%)

Train Epoch: 2 [59136/60000 (99%)]      Loss: 0.252602

Test set: Average loss: 0.2900, Accuracy: 8932/10000 (89%)

Train Epoch: 3 [5504/60000 (9%)]        Loss: 0.244024

Test set: Average loss: 0.2899, Accuracy: 8920/10000 (89%)

Train Epoch: 3 [11904/60000 (20%)]      Loss: 0.261676

Test set: Average loss: 0.2804, Accuracy: 8970/10000 (90%)

Train Epoch: 3 [18304/60000 (30%)]      Loss: 0.316545

Test set: Average loss: 0.2775, Accuracy: 9019/10000 (90%)

Train Epoch: 3 [24704/60000 (41%)]      Loss: 0.198511
```

```
Test set: Average loss: 0.2731, Accuracy: 9005/10000 (90%)

Train Epoch: 3 [31104/60000 (52%)]      Loss: 0.285187

Test set: Average loss: 0.2735, Accuracy: 9021/10000 (90%)

Train Epoch: 3 [37504/60000 (62%)]      Loss: 0.283321

Test set: Average loss: 0.2821, Accuracy: 8983/10000 (90%)

Train Epoch: 3 [43904/60000 (73%)]      Loss: 0.309603

Test set: Average loss: 0.2675, Accuracy: 9034/10000 (90%)

Train Epoch: 3 [50304/60000 (84%)]      Loss: 0.158129

Test set: Average loss: 0.2672, Accuracy: 9049/10000 (90%)

Train Epoch: 3 [56704/60000 (94%)]      Loss: 0.260526

Test set: Average loss: 0.2711, Accuracy: 9043/10000 (90%)

Train Epoch: 4 [3072/60000 (5%)]        Loss: 0.196330

Test set: Average loss: 0.2764, Accuracy: 9008/10000 (90%)

Train Epoch: 4 [9472/60000 (16%)]       Loss: 0.176387

Test set: Average loss: 0.2639, Accuracy: 9037/10000 (90%)

Train Epoch: 4 [15872/60000 (26%)]      Loss: 0.209738

Test set: Average loss: 0.2735, Accuracy: 8979/10000 (90%)

Train Epoch: 4 [22272/60000 (37%)]      Loss: 0.184776

Test set: Average loss: 0.2698, Accuracy: 9039/10000 (90%)

Train Epoch: 4 [28672/60000 (48%)]      Loss: 0.231746

Test set: Average loss: 0.2793, Accuracy: 8985/10000 (90%)

Train Epoch: 4 [35072/60000 (58%)]      Loss: 0.280187

Test set: Average loss: 0.2678, Accuracy: 9035/10000 (90%)

Train Epoch: 4 [41472/60000 (69%)]      Loss: 0.225850

Test set: Average loss: 0.2694, Accuracy: 9023/10000 (90%)

Train Epoch: 4 [47872/60000 (80%)]      Loss: 0.265149

Test set: Average loss: 0.2580, Accuracy: 9065/10000 (91%)

Train Epoch: 4 [54272/60000 (90%)]      Loss: 0.253430

Test set: Average loss: 0.2599, Accuracy: 9065/10000 (91%)
```



Out[4]:
```
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)
    (1): ReLU()
  )
  (fc2): Sequential(
    (0): Linear(in_features=128, out_features=10, bias=True)
  )
)
```

Where did you add your non-linearities?

I kept the position of the nonlinearities in the given code, so right after the conv layers and before the pool layers (ReLU). I also added a ReLU nonlinearity between the two fully connected layers. I didn't include a softmax at the end because one is already applied in the cross entropy loss.

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

A necessary condition for this performance is that there exist fairly good linear separators that separate each class and the rest for 80+% of the data. The cross entropy loss applies a softmax so the linear output is fine in this scenario. Also, as Professor Pathak discussed in class, the negative gradient of cross entropy just corresponds to increasing the logit of the correct class so the objective of minimizing cross entropy is perfectly fine.

In [ ]:

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 which loads this PASCAL data. Browse the folders and files under `VOCdevkit` to understand the structure and labeling. Complete the functions `preload_anno` and `__getitem__` in `voc_dataset.py` according to the following instructions and the instructions in the code. More information about the dataset can be found here. We will use data in 'trainval' for training and 'test' for testing.

- `preload_anno` : This function will be called when the dataloader is initialized. We will load the annotations under folder `Annotations` . Each .xml file in the `Annotations` folder corresponds to the image with the same name under `JPEGImages` . In this function, we need to load `label` and `weight` vectors for each image according to the .xml file.
- The labels should be 0 by default. Assign 1 for each class label in the .xml file. For example, in 000001.xml, the label vector should have 1s at the class indices correspond to 'dog' and 'person'. The rest of the vector should be 0.
- The weights should be 1 by defatul. For each class label in the image, if 'difficult'=1 (which means it is ambiguous), we will assign 0 for the weight vector at this class index. This weight will be used when we calculate the test performance. We will not consider the ambiguous labels during testing.

- `__getitem__` : This function will be called when the dataloader is called during training. It takes as input the index, and returns a tuple - `(image, label, weight)` . You need to load the image from the `JPEGImages` folder and load the corresponding label and weight using `self.anno_list` .

Data Augmentation Modify `__getitem__` to randomly *augment* each datapoint using TORCHVISION.TRANSFORMS. Make sure the data augmentation is only used for training data (based on self.split). Please describe what data augmentation you implement.

- Before any augmentation, resize all the images based on `self.size` .
- **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]` . You may also rescale the images to `[-1, 1]` . There is no "correct" answer here! Feel free to search online about the data augmentation methods people usually use.

DESCRIBE YOUR AUGMENTATION PIPELINE HERE**

Train Augmentations:

My train augmentations consisted of normalization, where the mean and std were from the ImageNet dataset, random horizontal flipping, random cropping, brightness variation, and random rotation.

Test Augmentations:

My test augmentations consisted of just the same normalization as in the train augmentations.

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). 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 is an estimate of the area under the precision-recall curve for a single class.

1.4 Let's Start Training! (5 pts)

Fill out the loss function for multi-label classification in `trainer.py` and start training. In this question, you will use the model that you finished in the previous question (with proper non-linearities).

Initialize a fresh model and optimizer. Then run your training code for 5 epochs and print the mAP on test set. The resulting mAP should be around 0.24. Make sure to tune the hyperparameters.

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

# create hyperparameter argument class
# Use image size of 64x64 in Q1. We will use a default size of 224x224 for the r
args = ARGS(epochs=5, inp_size=64, batch_size=20, lr=0.001, gamma=0.7, log_every
print(args)
```

```
args.batch_size = 20
args.device = cuda
args.epochs = 5
args.gamma = 0.7
args.inp_size = 64
args.log_every = 100
args.lr = 0.001
args.save_at_end = True
args.save_freq = 10
args.step_size = 1
args.test_batch_size = 1000
args.val_every = 100
```

In [2]:
```
# initializes the 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=args.step_size,
# 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)
```

```
Train Epoch: 0 [0 (0%)] Loss: 0.678817
0.07707119113333445
Train Epoch: 0 [100 (40%)]      Loss: 0.237034
0.1561614832181566
Train Epoch: 0 [200 (80%)]      Loss: 0.202147
0.1796810966876655
Train Epoch: 1 [300 (20%)]      Loss: 0.186375
0.19904037898820232
Train Epoch: 1 [400 (59%)]      Loss: 0.226133
0.21088154400180642
Train Epoch: 1 [500 (99%)]      Loss: 0.197145
0.2248062588520447
Train Epoch: 2 [600 (39%)]      Loss: 0.209637
0.23705273588627337
Train Epoch: 2 [700 (79%)]      Loss: 0.129279
0.24052953987531023
Train Epoch: 3 [800 (19%)]      Loss: 0.215028
0.24946973122634647
```

```
Train Epoch: 3 [900 (59%)]      Loss: 0.195588
0.2513845313579141
Train Epoch: 3 [1000 (98%)]     Loss: 0.226564
0.25634061037505435
Train Epoch: 4 [1100 (38%)]     Loss: 0.204269
0.2620287092032275
Train Epoch: 4 [1200 (78%)]     Loss: 0.185575
0.26824479423976694
test map: 0.265469733014256
```

TensorBoard is an awesome visualization tool. It was firstly integrated in TensorFlow. 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 (for Pytorch >= 1.1.0) or TensorBoardX - a third party library. Following these links to 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.*

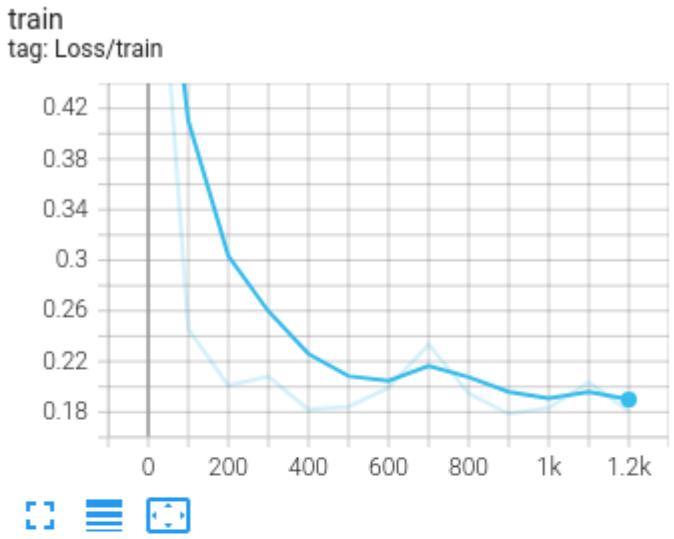
Show clear screenshots of the learning curves of testing MAP and training loss for 5 epochs (batch size=20, learning rate=0.001). Please evaluate your model to calculate the MAP on the testing dataset every 100 iterations.

In [2]:

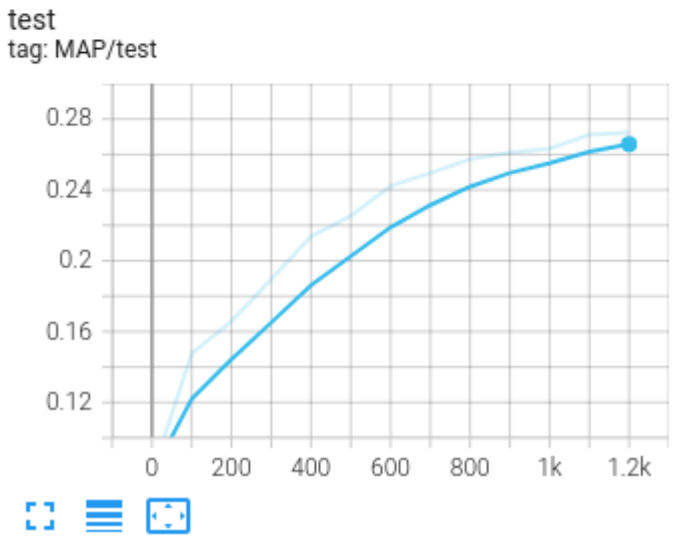
```
args = ARGS(epochs=5, batch_size=20, lr=0.001, inp_size=64)
model = SimpleCNN(num_classes=len(VOCDataset.CLASS_NAMES), inp_size=64, c_dim=3)
optimizer = torch.optim.Adam(model.parameters(), lr=args.lr)
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=args.step_size,
test_ap, test_map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

```
Train Epoch: 0 [0 (0%)] Loss: 0.684341
0.07935421723358299
Train Epoch: 0 [100 (40%)]      Loss: 0.245733
0.14766045432429703
Train Epoch: 0 [200 (80%)]      Loss: 0.201044
0.16583498376725417
Train Epoch: 1 [300 (20%)]      Loss: 0.208218
0.1895160060769484
Train Epoch: 1 [400 (59%)]      Loss: 0.181977
0.21374501495370785
Train Epoch: 1 [500 (99%)]      Loss: 0.184173
0.22523008009212736
Train Epoch: 2 [600 (39%)]      Loss: 0.199220
0.24227686280080346
Train Epoch: 2 [700 (79%)]      Loss: 0.233587
0.2495561492268977
Train Epoch: 3 [800 (19%)]      Loss: 0.194873
0.25738166321231637
Train Epoch: 3 [900 (59%)]      Loss: 0.178514
0.2609400041308615
Train Epoch: 3 [1000 (98%)]     Loss: 0.183231
0.2633717690589613
Train Epoch: 4 [1100 (38%)]     Loss: 0.203642
0.2711326789735352
Train Epoch: 4 [1200 (78%)]     Loss: 0.180814
0.27232201110650467
test map: 0.2787274906527557
```

Loss


train
tag: Loss/train
0.42
0.38
0.34
0.3
0.26
0.22
0.18
0
200
400
600
800
1k
1.2k

MAP


test
tag: MAP/test
0.28
0.24
0.2
0.16
0.12
0
200
400
600
800
1k
1.2k

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 architecture, known as CaffeNet. If you are familiar with Caffe, a prototxt of the network is available here. A visualization of the network is available here.

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 [1]:

```
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


class CaffeNet(nn.Module):
    def __init__(self, num_classes=20, inp_size=224, c_dim=3):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 96, 11, stride=4, padding="valid")
        self.pool1 = nn.MaxPool2d(3, 2)
        self.conv2 = nn.Conv2d(96, 256, 5, stride=1, padding="same")
        self.pool2 = nn.MaxPool2d(3, 2)
        self.conv3 = nn.Conv2d(256, 384, 3, stride=1, padding="same")
        self.conv4 = nn.Conv2d(384, 384, 3, stride=1, padding="same")
        self.conv5 = nn.Conv2d(384, 256, 3, stride=1, padding="same")
        self.pool5 = nn.MaxPool2d(3, 2)

        self.fc6 = nn.Linear(6400, 4096)
        self.dropout6 = nn.Dropout(0.5)
        self.fc7 = nn.Linear(4096, 4096)
        self.dropout7 = nn.Dropout(0.5)

        self.fc8 = nn.Linear(4096, 20)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = self.forward_features(x)
        x = F.relu(x)
        x = self.dropout7(x)
        x = self.fc8(x)
        x = self.sigmoid(x)
        return x

    def forward_features(self, x):
        x = self.conv1(x)
        x = F.relu(x)
        x = self.pool1(x)
        x = self.conv2(x)
        x = F.relu(x)
        x = self.pool2(x)
        x = self.conv3(x)
        x = F.relu(x)
        x = self.conv4(x)
        x = F.relu(x)
        x = self.conv5(x)
        x = F.relu(x)
        x = self.pool5(x)
        x = torch.flatten(x, start_dim=1)
        x = self.fc6(x)
        x = F.relu(x)
        x = self.dropout6(x)
```

```
        x = self.fc7(x)
        return x
```

2.2 Save the Model (5 pts)

Fill out `save_model()` in `trainer.py` to save the checkpoints of the model periodically. **You will need these models later.**

2.3 Train and Test (5pts)

Show clear screenshots of testing MAP and training loss for 50 epochs. The final MAP should be at least around 0.4. 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

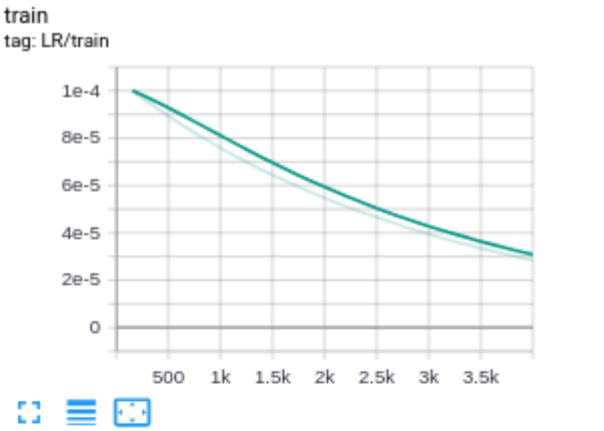
NOTE: SAVE AT LEAST 5 EVENLY SPACED CHECKPOINTS DURING TRAINING (1 at end)

In [2]:
```
args = ARGS(epochs=50, batch_size=32, lr=0.0001, gamma=0.95, log_every=250, val_
model = CaffeNet()
model.to()
optimizer = torch.optim.Adam(model.parameters(), lr=args.lr)
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, args.step_size, args.gamm
test_ap, test_map = trainer.train(args, model, optimizer, scheduler, model_name=
print('test map:', test_map)
```

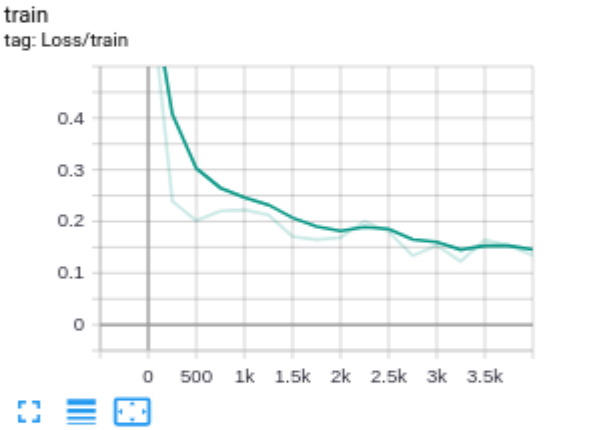
```
Train Epoch: 0 [0 (0%)] Loss: 0.691711
0.0784589675504003
Train Epoch: 1 [250 (59%)]      Loss: 0.224166
0.10554606550987836
Train Epoch: 3 [500 (18%)]      Loss: 0.199720
0.14652089394284556
Train Epoch: 4 [750 (78%)]      Loss: 0.213377
0.19086541337939167
Train Epoch: 6 [1000 (37%)]     Loss: 0.238671
0.23338250653705117
Train Epoch: 7 [1250 (96%)]     Loss: 0.214446
0.26570668064757674
Train Epoch: 9 [1500 (55%)]     Loss: 0.153335
0.2923314497099425
Train Epoch: 11 [1750 (15%)]    Loss: 0.175858
0.308810695803064
Train Epoch: 12 [2000 (74%)]    Loss: 0.169708
0.3196483782617757
Train Epoch: 14 [2250 (33%)]    Loss: 0.173147
0.3370750674621268
Train Epoch: 15 [2500 (92%)]    Loss: 0.161965
0.35716942658721074
Train Epoch: 17 [2750 (52%)]    Loss: 0.165715
0.3639815168932926
Train Epoch: 19 [3000 (11%)]    Loss: 0.173318
0.38216185900794974
Train Epoch: 20 [3250 (70%)]    Loss: 0.138881
0.3888815212315434
```

```
Train Epoch: 22 [3500 (29%)]    Loss: 0.137587
0.3928481051776763
Train Epoch: 23 [3750 (89%)]    Loss: 0.148573
0.39949266595229715
Train Epoch: 25 [4000 (48%)]    Loss: 0.137879
0.39972811492123744
Train Epoch: 27 [4250 (7%)]     Loss: 0.098056
0.4064225808801778
Train Epoch: 28 [4500 (66%)]    Loss: 0.116812
0.4088195563979279
Train Epoch: 30 [4750 (25%)]    Loss: 0.076597
0.41125484357178294
Train Epoch: 31 [5000 (85%)]    Loss: 0.099131
0.4094467073612118
Train Epoch: 33 [5250 (44%)]    Loss: 0.090959
0.4144100163660437
Train Epoch: 35 [5500 (3%)]     Loss: 0.128714
0.4176852975023948
Train Epoch: 36 [5750 (62%)]    Loss: 0.088461
0.41256288513875916
Train Epoch: 38 [6000 (22%)]    Loss: 0.095840
0.4143234768097222
Train Epoch: 39 [6250 (81%)]    Loss: 0.100404
0.4172378392438299
Train Epoch: 41 [6500 (40%)]    Loss: 0.088979
0.4173704266719646
Train Epoch: 42 [6750 (99%)]    Loss: 0.085091
0.4169819595791916
Train Epoch: 44 [7000 (59%)]    Loss: 0.099109
0.41489510974535426
Train Epoch: 46 [7250 (18%)]    Loss: 0.083472
0.4143678614497602
Train Epoch: 47 [7500 (77%)]    Loss: 0.054491
0.4154436251311065
Train Epoch: 49 [7750 (36%)]    Loss: 0.060693
0.414235890796882
test map: 0.4035523247570797
```

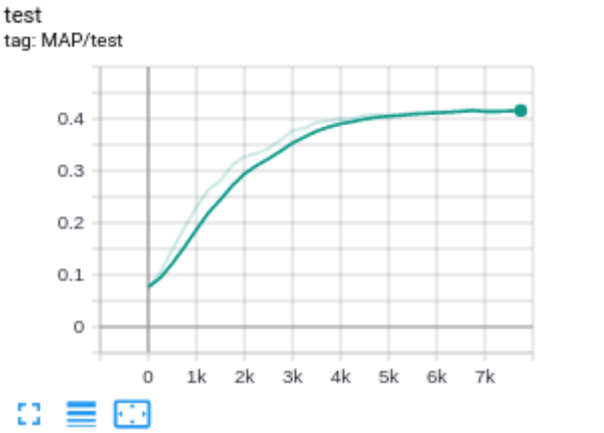
LR



Loss



MAP



2.4 Visualizing: Conv-1 filters (5pts)

Extract and compare the weights of conv1 filters at different stages of the training (at least from 5 different epochs).

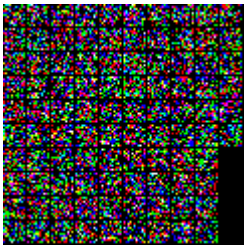
- Write a function to load your model checkpoints.
- Get the weights for conv1 from the loaded model.
- Visualize the weights using the following vis() function.

Sometimes the filters all look very random and may not change too much across epochs. Don't worry! You will get full credits as long as the code is correct.

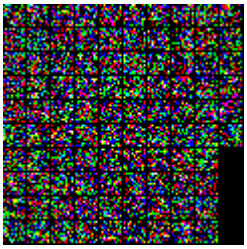
In [3]:
```
import numpy as np
from PIL import Image

# This function plots all the filters in one image.
def vis(conv1, name):
    assert type(conv1) == np.ndarray
    assert conv1.shape == (11, 11, 3, 96)
    im = np.zeros((120, 120, 3))
    step_size = 12
    column = 0
    row = 0
    for k in range(conv1.shape[3]):
        this_filter = conv1[:, :, :, k]
        im[column*step_size:column*step_size+11, row*step_size:row*step_size+11,
        column = column + 1
        if column == 10:
            column = 0
            row = row + 1
    image = Image.fromarray(np.uint8((im-np.mean(im))/np.std(im)))
    #image.show()
    image.save("figures/{}.png".format(name))
    return image
```

In [5]:
```
# Write your code here to get the conv1 filters for each epoch
path_format = "models/checkpoint-caffenet_scratch-epoch{}.pth"
conv1s= []
for i in range(1, 11):
    model_path = path_format.format(i * 5)
    state_dict = torch.load(model_path)
    conv1s.append(state_dict['conv1.weight'].cpu().permute(2, 3, 1, 0).numpy())

# For each epoch, use vis() to visualize the filters.
# Before passing the weights into vis(), make sure it is an numpy array with sha
# You may need torch.permute to reorganize the dimensions.
for i, conv1 in enumerate(conv1s):
    vis(conv1, i)
```

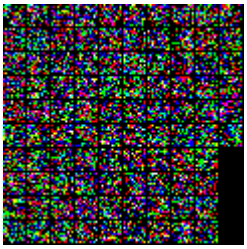
epoch 0 conv1 filters:



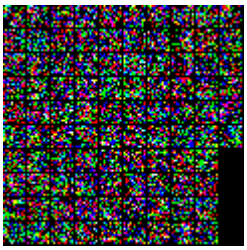
epoch 10 conv1 filters:



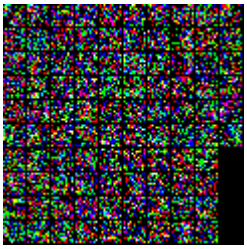
epoch 20 conv1 filters:



epoch 30 conv1 filters:



epoch 40 conv1 filters:



In [ ]:

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 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! Do not load the pretrained weights for this question. We will get to that in the next question.

In [1]:
```
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


# you could write the whole class....
# or one line :D
ResNet = models.resnet18
```

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. Tune hyperparameters properly to get mAP around 0.5. Report tensorboard screenshots for *all* of the

summaries listed above (for image summaries show screenshots at $n \geq 3$ iterations). For the histograms, include the screenshots of the gradients of layer1.1.conv1.weight and layer4.0.bn2.bias.

REMEMBER TO SAVE A MODEL AT THE END OF TRAINING

In [2]:

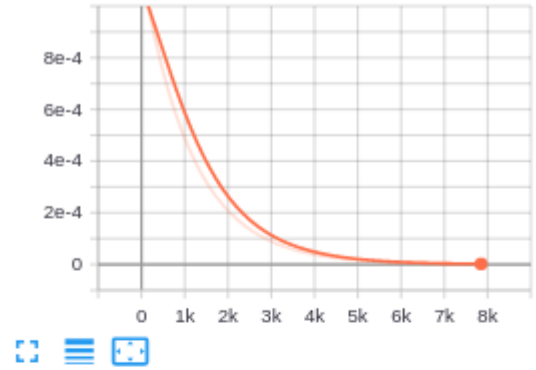
```
args = ARGS(lr=0.001, gamma=0.875, epochs=50, log_every=250, val_every=250, test
model = nn.Sequential(ResNet(), nn.Sigmoid())
model[0].fc = nn.Linear(512, 20)
optimizer = torch.optim.Adam(model.parameters(), lr=args.lr)
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, args.step_size, gamma=arg
test_ap, test_map = trainer.train(args, model, optimizer, scheduler, model_name=
print('test map:', test_map)
```

```
Train Epoch: 0 [0 (0%)] Loss: 0.688458
0.08056013123039628
Train Epoch: 1 [250 (59%)]      Loss: 0.207289
0.19369932969348108
Train Epoch: 3 [500 (18%)]      Loss: 0.165179
0.24129273070398694
Train Epoch: 4 [750 (78%)]      Loss: 0.166618
0.2914690526842999
Train Epoch: 6 [1000 (37%)]     Loss: 0.180516
0.32696730589792555
Train Epoch: 7 [1250 (96%)]     Loss: 0.181079
0.33650263841865513
Train Epoch: 9 [1500 (55%)]     Loss: 0.137413
0.3896580987865251
Train Epoch: 11 [1750 (15%)]    Loss: 0.145514
0.41077113781918184
Train Epoch: 12 [2000 (74%)]    Loss: 0.163915
0.4309862924880952
Train Epoch: 14 [2250 (33%)]    Loss: 0.137167
0.43199099463757645
Train Epoch: 15 [2500 (92%)]    Loss: 0.123929
0.45671356130161955
Train Epoch: 17 [2750 (52%)]    Loss: 0.135618
0.46101131060291795
Train Epoch: 19 [3000 (11%)]    Loss: 0.101740
0.48304875963039784
Train Epoch: 20 [3250 (70%)]    Loss: 0.105733
0.4789194153899964
Train Epoch: 22 [3500 (29%)]    Loss: 0.100793
0.48592370297543336
Train Epoch: 23 [3750 (89%)]    Loss: 0.086307
0.48686607085233746
Train Epoch: 25 [4000 (48%)]    Loss: 0.122521
0.49475154572567054
Train Epoch: 27 [4250 (7%)]     Loss: 0.097966
0.4941897346300742
Train Epoch: 28 [4500 (66%)]    Loss: 0.109268
0.49501151668474
Train Epoch: 30 [4750 (25%)]    Loss: 0.104902
0.4953016367471147
Train Epoch: 31 [5000 (85%)]    Loss: 0.086078
0.4958042838856202
Train Epoch: 33 [5250 (44%)]    Loss: 0.085340
0.49785620074452286
Train Epoch: 35 [5500 (3%)]     Loss: 0.119642
0.49991952186519917
Train Epoch: 36 [5750 (62%)]    Loss: 0.114841
```

```
0.5015002885558572
Train Epoch: 38 [6000 (22%)]    Loss: 0.084887
0.49734695843173177
Train Epoch: 39 [6250 (81%)]    Loss: 0.104603
0.4984315046008737
Train Epoch: 41 [6500 (40%)]    Loss: 0.100967
0.5004873684450372
Train Epoch: 42 [6750 (99%)]    Loss: 0.123334
0.5004997066357303
Train Epoch: 44 [7000 (59%)]    Loss: 0.084665
0.500199641332797
Train Epoch: 46 [7250 (18%)]    Loss: 0.081049
0.5022052840910909
Train Epoch: 47 [7500 (77%)]    Loss: 0.083421
0.5006379439974482
Train Epoch: 49 [7750 (36%)]    Loss: 0.085745
0.501648418937741
test map: 0.5105096170848368
```

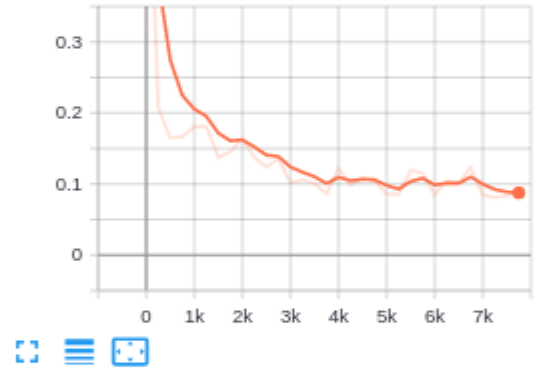
LR

train
tag: LR/train



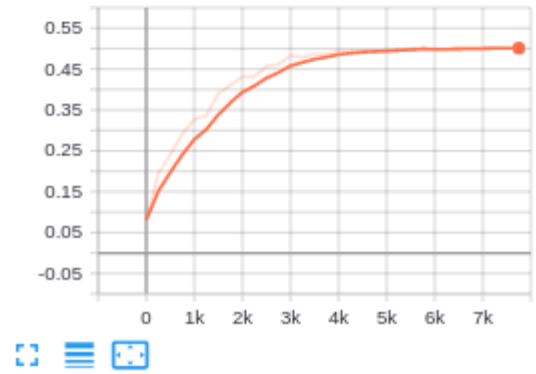
Loss

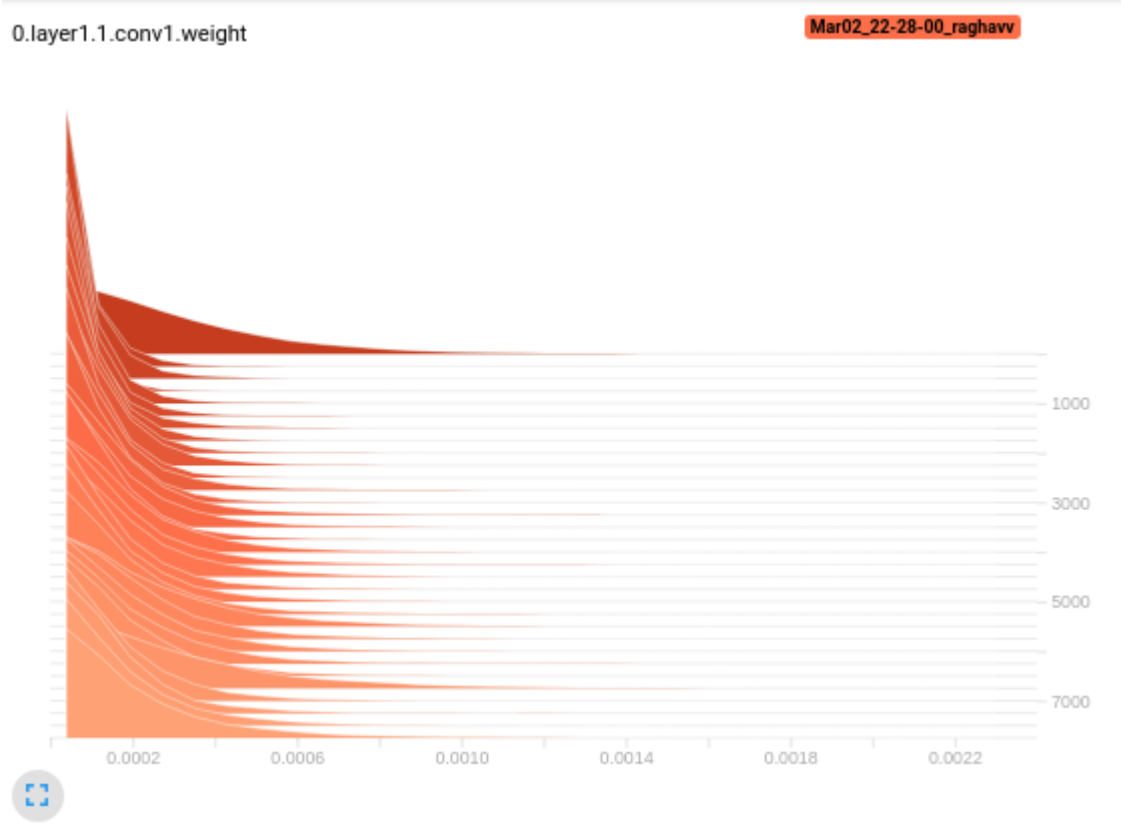
train
tag: Loss/train

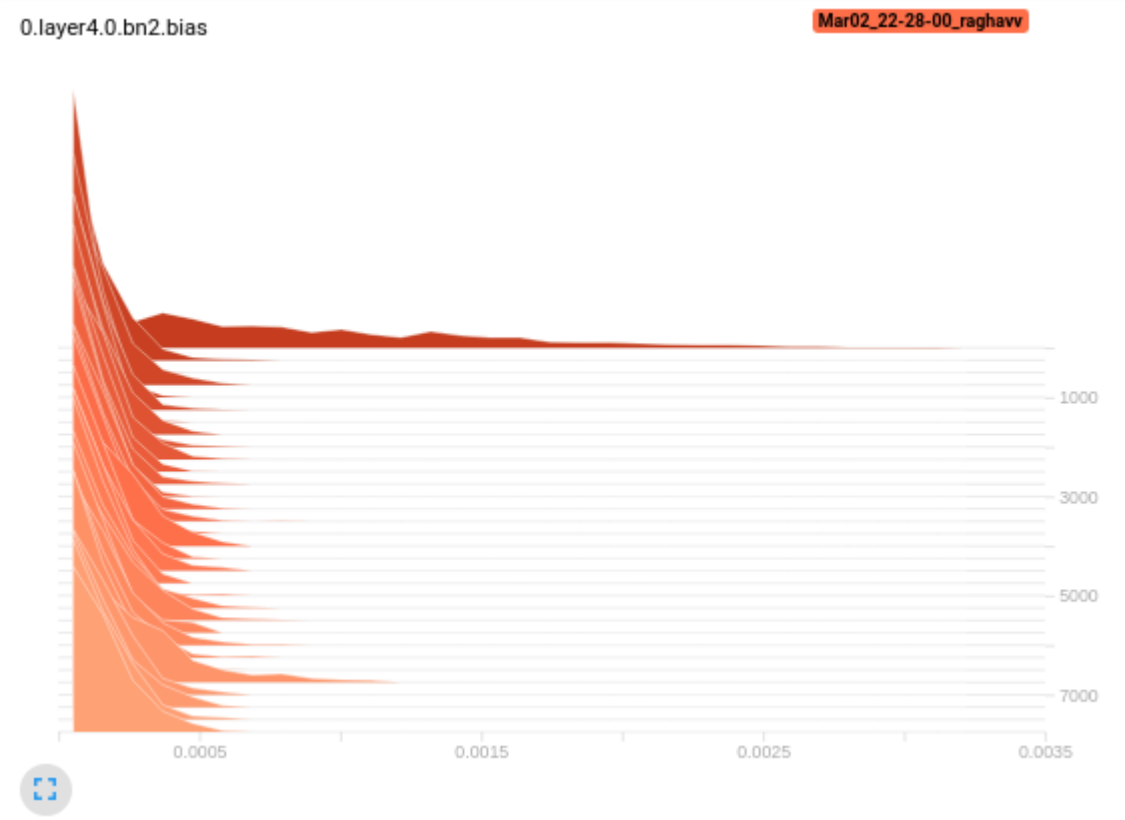


MAP

test
tag: MAP/test




0.layer1.1.conv1.weight
Mar02_22-28-00_raghavv
1000
3000
5000
7000
0.0002
0.0006
0.0010
0.0014
0.0018
0.0022


0.layer4.0.bn2.bias
Mar02_22-28-00_raghavv
1000
3000
5000
7000
0.0005
0.0015
0.0025
0.0035

In [ ]:

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), 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 layer from scratch (the very last layer that outputs the classes).

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).

You simply match the names of the layers in the saved state dict to the element names in your current network that you want the saved weights to be loaded to

In [8]:

```
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__()
        self.pretrained = models.resnet18(pretrained=True)
        self.pretrained.fc = nn.Linear(512, 20)
        self.sig = nn.Sigmoid()

    def forward(self, x):
        logits = self.pretrained(x)
        return self.sig(logits)
```

```
    def forward_features(self, x):
        x = self.pretrained.conv1(x)
        x = self.pretrained.bn1(x)
        x = self.pretrained.relu(x)
        x = self.pretrained.maxpool(x)
        x = self.pretrained.layer1(x)
        x = self.pretrained.layer2(x)
        x = self.pretrained.layer3(x)
        x = self.pretrained.layer4(x)
        x = self.pretrained.avgpool(x)
        return x
```

Train the model with a similar hyperparameter setup as in the scratch case. No need to freeze the loaded weights. 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. Also feel free to tune the hyperparameters to improve performance.

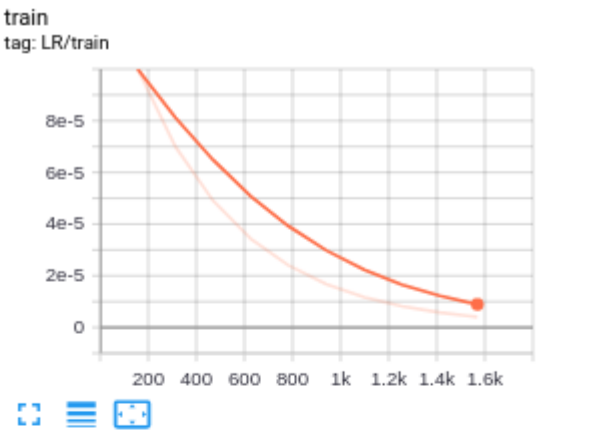
REMEMBER TO SAVE MODEL AT END OF TRAINING

In [4]:

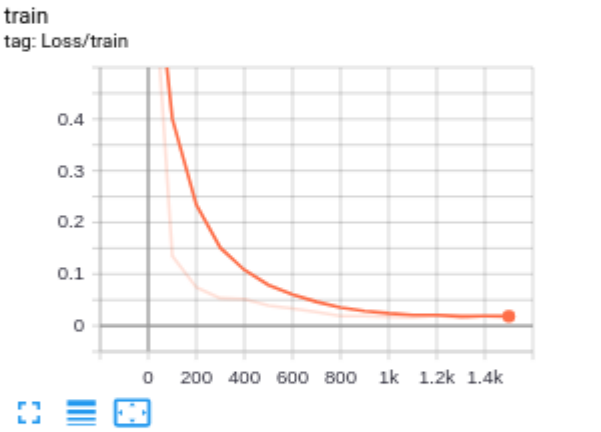
```
args = ARGS(lr=0.0001, gamma=0.70, epochs=10, log_every=100, val_every=100, test
model = PretrainedResNet()
optimizer = torch.optim.Adam(model.parameters(), lr=args.lr)
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, args.step_size, gamma=arg
test_ap, test_map = trainer.train(args, model, optimizer, scheduler, model_name=
print('test map:', test_map)
```

```
Train Epoch: 0 [0 (0%)] Loss: 0.842778
0.08321969562038846
Train Epoch: 0 [100 (64%)]      Loss: 0.135048
0.7654220687696534
Train Epoch: 1 [200 (27%)]      Loss: 0.074368
0.8144729261841819
Train Epoch: 1 [300 (91%)]      Loss: 0.053133
0.8298551614179456
Train Epoch: 2 [400 (55%)]      Loss: 0.051945
0.8380146812108149
Train Epoch: 3 [500 (18%)]      Loss: 0.039175
0.8426671177530591
Train Epoch: 3 [600 (82%)]      Loss: 0.033336
0.843094513190023
Train Epoch: 4 [700 (46%)]      Loss: 0.026231
0.8433836791993151
Train Epoch: 5 [800 (10%)]      Loss: 0.018942
0.8433562364868843
Train Epoch: 5 [900 (73%)]      Loss: 0.018844
0.8425709525151369
Train Epoch: 6 [1000 (37%)]     Loss: 0.017491
0.843212941537508
Train Epoch: 7 [1100 (1%)]      Loss: 0.016047
0.8434736132642418
Train Epoch: 7 [1200 (64%)]     Loss: 0.019922
0.8422612685897362
Train Epoch: 8 [1300 (28%)]     Loss: 0.014855
0.8428208279619934
Train Epoch: 8 [1400 (92%)]     Loss: 0.019463
0.8427232915613354
Train Epoch: 9 [1500 (55%)]     Loss: 0.017669
0.8423146312464406
test map: 0.8425921600704926
```

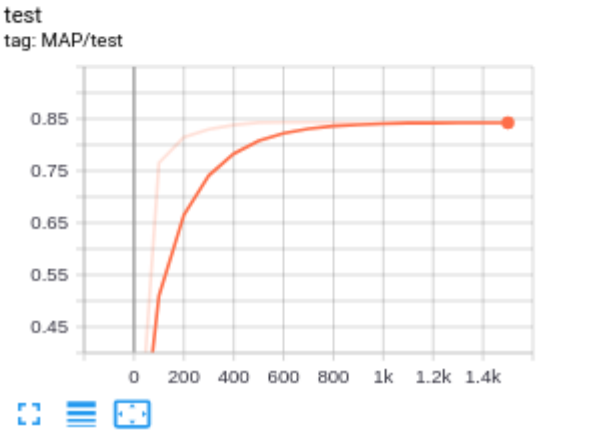
LR



Loss



MAP



In [ ]:

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 over the entire test set for each of them. You should compare the following feature representations to find the nearest neighbors:

1. The features before the final fc layer from the ResNet (finetuned from ImageNet). It is the features right before the final class label output.
2. pool5 features from the CaffeNet (trained from scratch)

You may use the this nearest neighbor function. Plot the raw images of the ones you picked and their nearest neighbors.

In [ ]:

```
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import DataLoader
from torchvision import models
from sklearn.neighbors import NearestNeighbors
import matplotlib.pyplot as plt
%matplotlib inline

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

# Load all the test images. Pick 3 indices.
dataset = VOCDataset(split='test', size=224, return_idx=True)
loader = DataLoader(dataset, batch_size=256, num_workers=4)

# Calculate the features for all the test images.
class CaffeNet(nn.Module):
    def __init__(self, num_classes=20, inp_size=224, c_dim=3):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 96, 11, stride=4, padding="valid")
        self.pool1 = nn.MaxPool2d(3, 2)
        self.conv2 = nn.Conv2d(96, 256, 5, stride=1, padding="same")
```

```
        self.pool2 = nn.MaxPool2d(3, 2)
        self.conv3 = nn.Conv2d(256, 384, 3, stride=1, padding="same")
        self.conv4 = nn.Conv2d(384, 384, 3, stride=1, padding="same")
        self.conv5 = nn.Conv2d(384, 256, 3, stride=1, padding="same")
        self.pool5 = nn.MaxPool2d(3, 2)

        self.fc6 = nn.Linear(6400, 4096)
        self.dropout6 = nn.Dropout(0.5)
        self.fc7 = nn.Linear(4096, 4096)
        self.dropout7 = nn.Dropout(0.5)

        self.fc8 = nn.Linear(4096, 20)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = self.forward_features(x)
        x = F.relu(x)
        x = self.dropout7(x)
        x = self.fc8(x)
        x = self.sigmoid(x)
        return x

    def forward_features(self, x):
        x = self.conv1(x)
        x = F.relu(x)
        x = self.pool1(x)
        x = self.conv2(x)
        x = F.relu(x)
        x = self.pool2(x)
        x = self.conv3(x)
        x = F.relu(x)
        x = self.conv4(x)
        x = F.relu(x)
        x = self.conv5(x)
        x = F.relu(x)
        x = self.pool5(x)
        x = torch.flatten(x, start_dim=1)
        x = self.fc6(x)
        x = F.relu(x)
        x = self.dropout6(x)
        x = self.fc7(x)
        return x

class PretrainedResNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.pretrained = models.resnet18(pretrained=True)
        self.pretrained.fc = nn.Linear(512, 20)
        self.sig = nn.Sigmoid()

    def forward(self, x):
        logits = self.pretrained(x)
        return self.sig(logits)

    def forward_features(self, x):
        batch_size = x.shape[0]
        x = self.pretrained.conv1(x)
        x = self.pretrained.bn1(x)
        x = self.pretrained.relu(x)
        x = self.pretrained.maxpool(x)
```

```
        x = self.pretrained.layer1(x)
        x = self.pretrained.layer2(x)
        x = self.pretrained.layer3(x)
        x = self.pretrained.layer4(x)
        x = self.pretrained.avgpool(x)
        return x.view(batch_size, -1)


caffenet = CaffeNet()
caffenet.load_state_dict(torch.load("models/checkpoint-caffenet_scratch-epoch50.
caffenet.eval()
resnet = PretrainedResNet()
resnet.load_state_dict(torch.load("models/checkpoint-resnet_finetuned-epoch10.pt
resnet.eval()

selected_indices = np.random.choice(len(dataset), size=3)
images = []
targets = []
caffenet_features = []
resnet_features = []
indices = []

# Fine the nearest neighbors for the 3 images you picked.
with torch.no_grad():
    for data, target, wgt, ind in loader:
        size = data.shape[0]
        caffenet_features.append(caffenet.forward_features(data))
        resnet_features.append(resnet.forward_features(data))
        targets.append(target)
        indices.append(ind)

print("Done with features")

caffenet_features = torch.cat(caffenet_features, dim=0).cpu().numpy()
resnet_features = torch.cat(resnet_features, dim=0).cpu().numpy()
targets = torch.cat(targets, dim=0)
indices = torch.cat(indices, dim=0)
```

In [9]:

```
# Plot the images and their neighbors.
caffenet_neighbors = NearestNeighbors(n_neighbors=5, algorithm='ball_tree', n_jo
_, caffenet_indices = caffenet_neighbors.kneighbors(caffenet_features)
caffenet_k_indices = caffenet_indices[selected_indices]
resnet_neighbors = NearestNeighbors(n_neighbors=5, algorithm='ball_tree', n_jobs
_, resnet_indices = resnet_neighbors.kneighbors(resnet_features)
resnet_k_indices = resnet_indices[selected_indices]

print(caffenet_k_indices)
print(resnet_k_indices)
```

```
[[2956 3117 4714 3798 2979]
 [ 532 2155 4315 3774 1458]
 [2162  101 1434 4903 2020]]
[[2956 2890 1059 2979 1557]
 [ 532 3931  634 4678   20]
 [2162 1697  591 2084  300]]
```

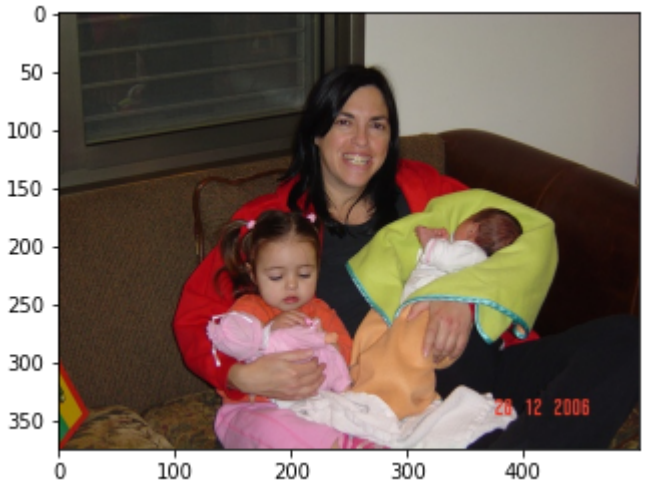
In [13]:

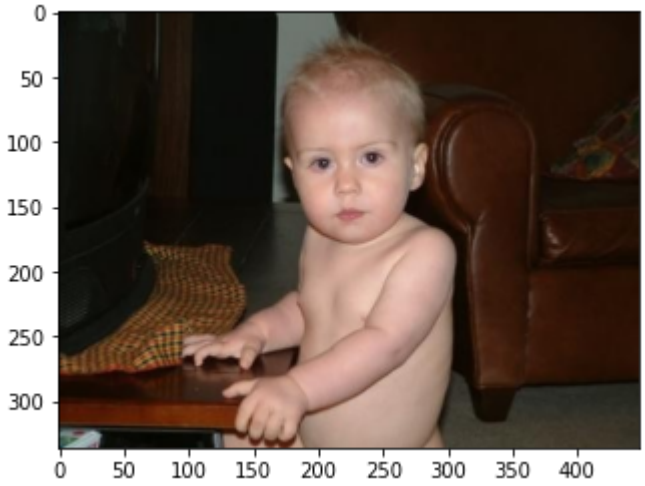
```
indices = indices.flatten()
for i in range(3):
```

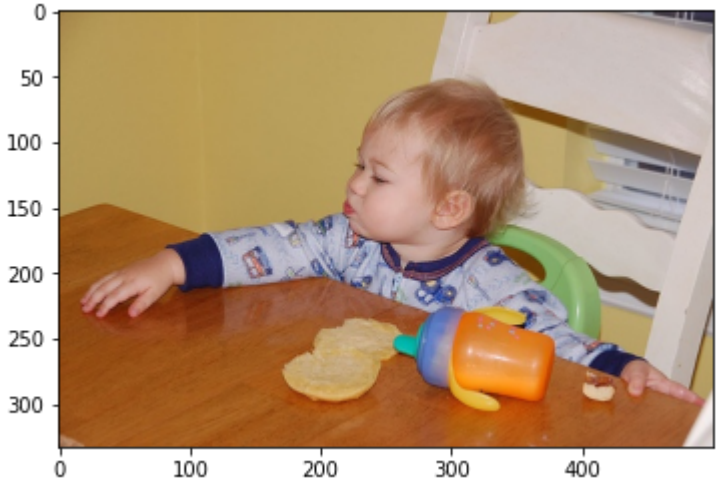
```
    print("Selected Image: {}".format(i + 1))
    plt.imshow(dataset.get_image(indices[selected_indices[i]]))
    plt.show()
    print("CaffeNet Nearest Neighbors:")
    for j in range(1, 5):
        plt.imshow(dataset.get_image(indices[caffenet_k_indices[i, j]]))
        plt.show()
    print("ResNet Nearest Neighbors:")
    for j in range(1, 5):
        plt.imshow(dataset.get_image(indices[resnet_k_indices[i, j]]))
        plt.show()
```

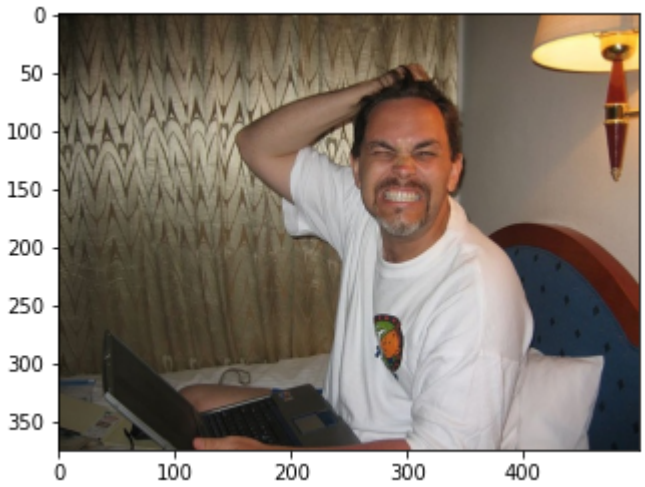
```
Selected Image: 1
```


```
CaffeNet Nearest Neighbors:
```

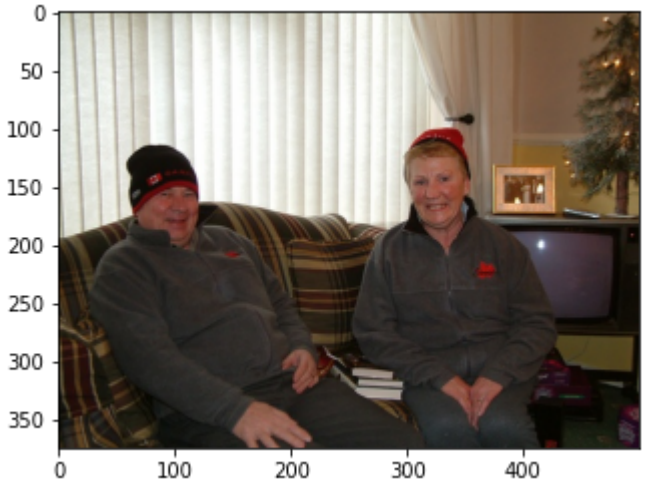




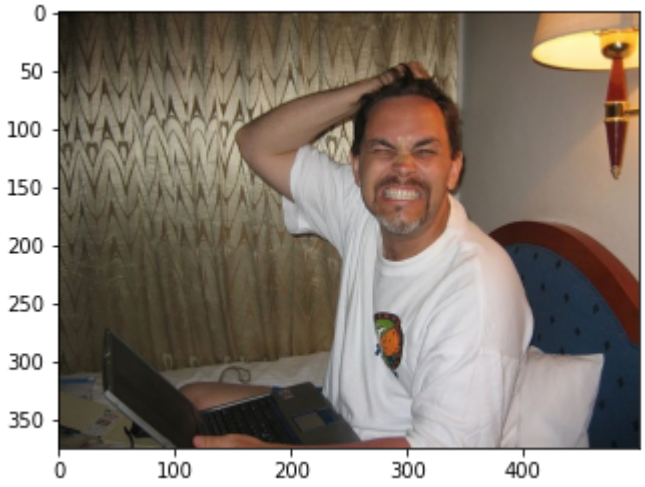




ResNet Nearest Neighbors:

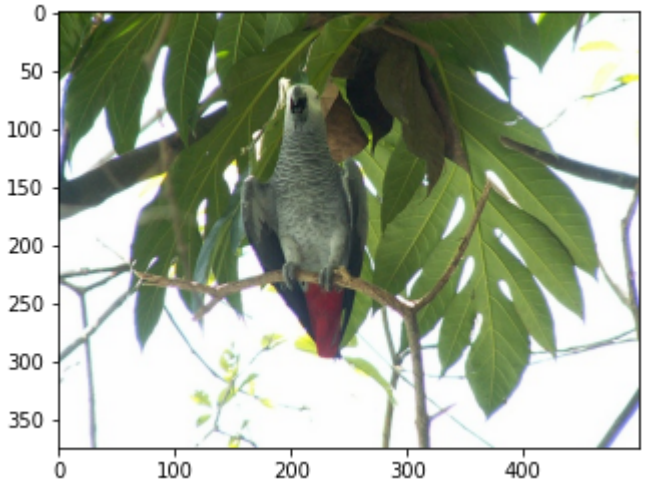

0
50
100
150
200
250
300
350
0
100
200
300
400

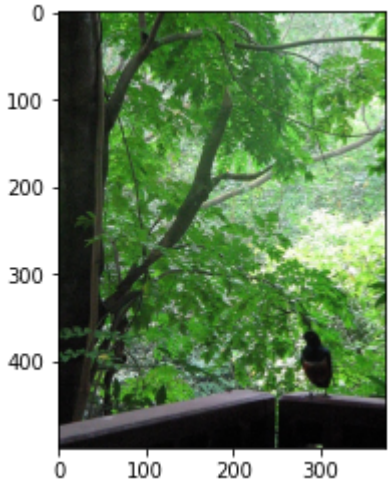

0
50
100
150
200
250
300
350
0
100
200
300
400


0
50
100
150
200
250
300
350
0
100
200
300
400

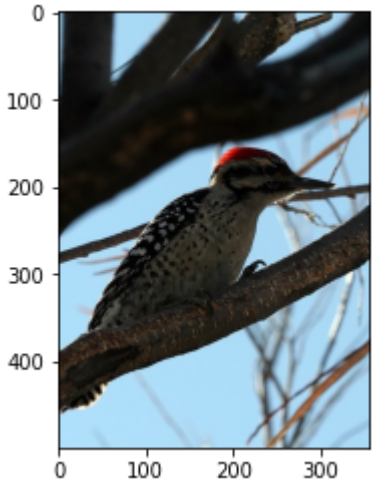
```
Selected Image: 2
```

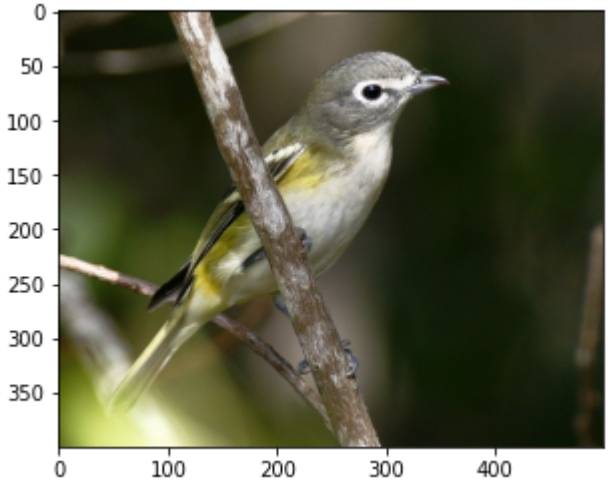
```
CaffeNet Nearest Neighbors:
```





ResNet Nearest Neighbors:

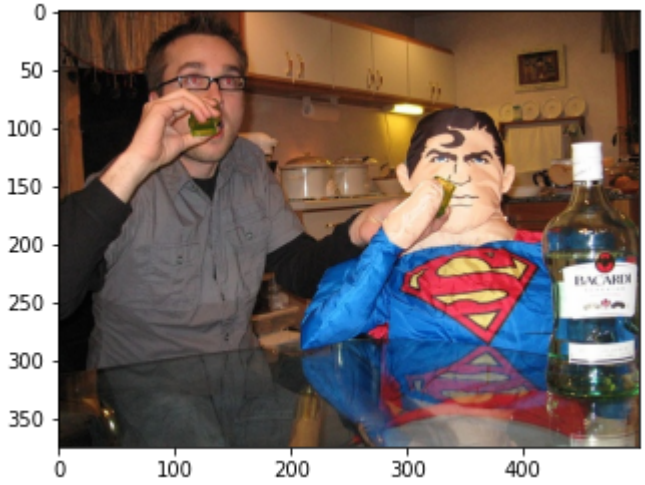

0
100
200
300
400
0
100
200
300

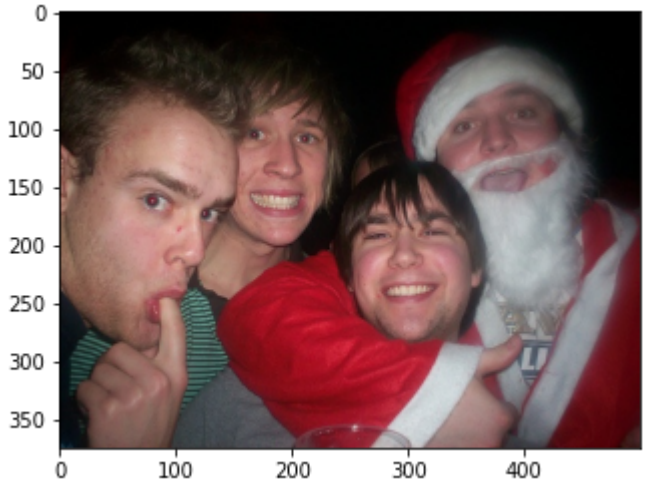

0
50
100
150
200
250
300
350
0
100
200
300
400

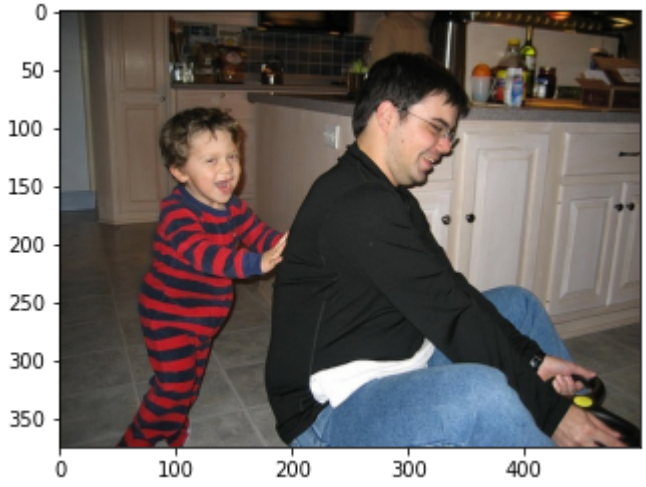

0
50
100
150
200
250
300
0
100
200
300
400

Selected Image: 3

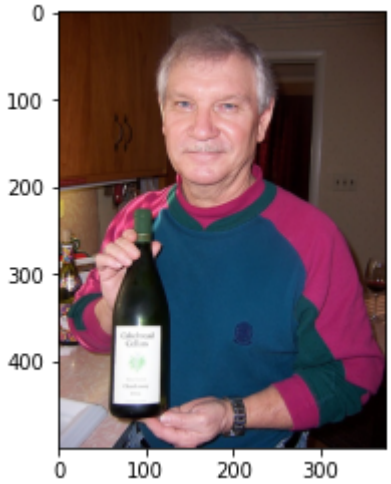
CaffeNet Nearest Neighbors:

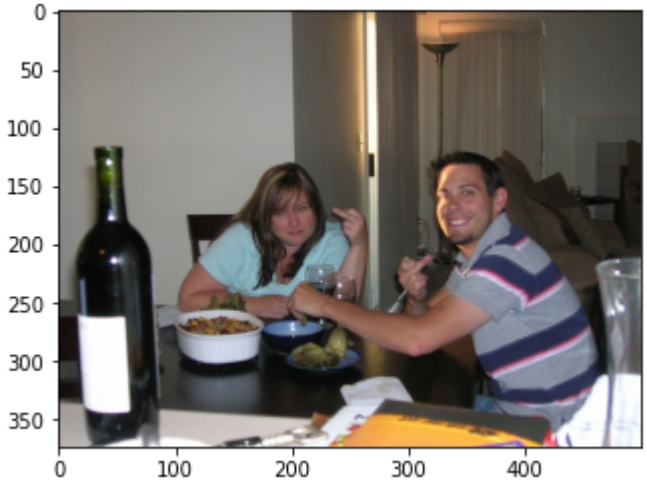




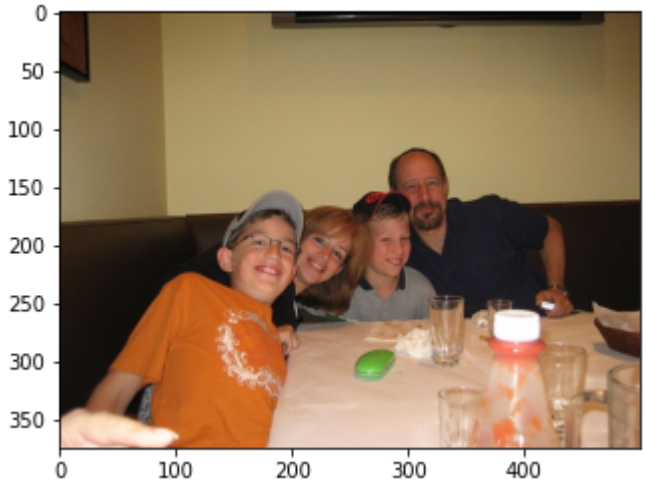


ResNet Nearest Neighbors:


0
100
200
300
400
0
100
200
300


0
50
100
150
200
250
300
350
0
100
200
300
400

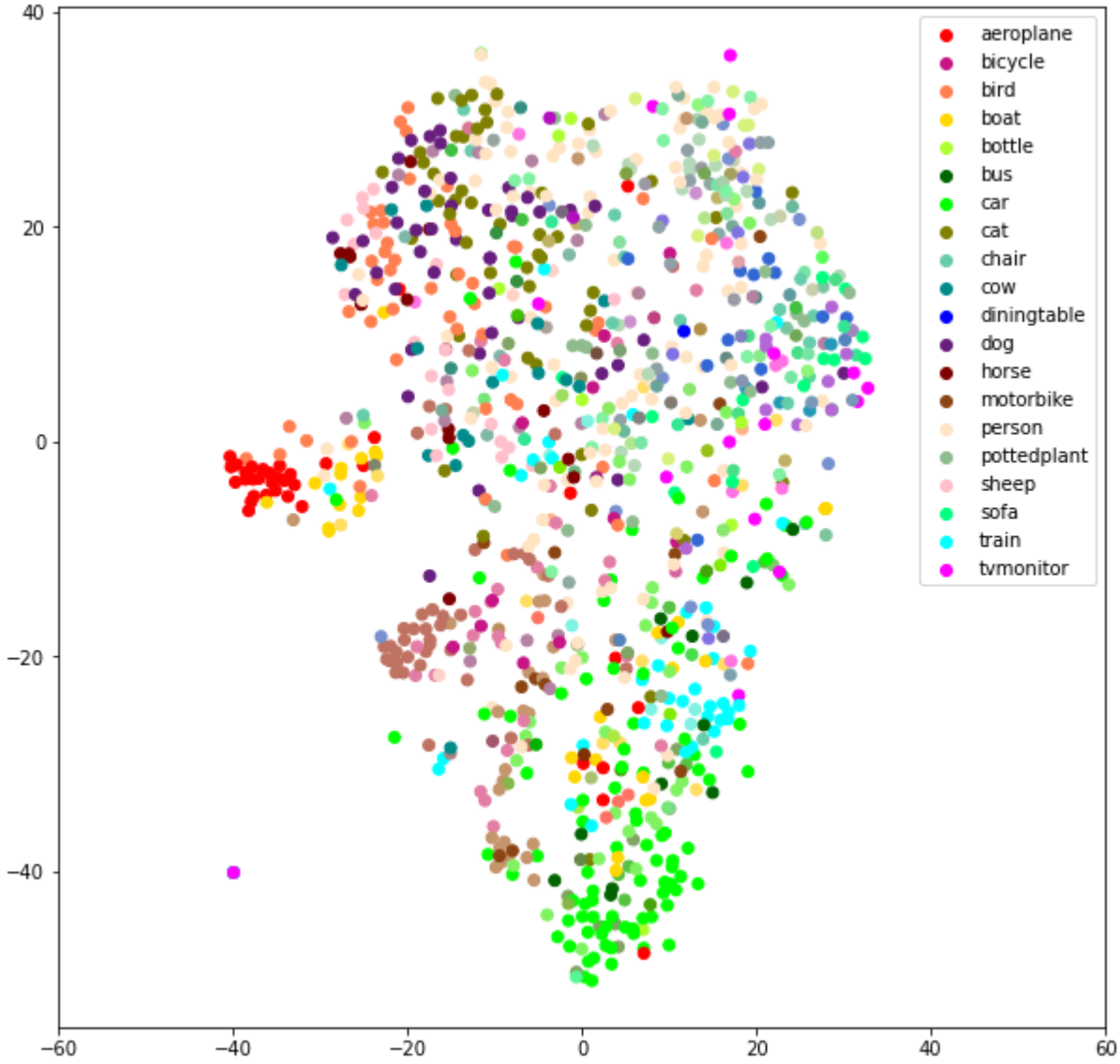

0
50
100
150
200
250
300
0
100
200
300
400



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.

In [28]:
```
from sklearn.manifold import TSNE

features = caffenet_features[:1000]
targets = targets[:1000]

embedding = TSNE(n_components=2, init='random').fit_transform(features)
class_colors = np.array([[255, 0, 0], [199, 21, 133], [255, 127, 80], [255, 215,
embedding_colors = []
for target in targets.cpu().numpy():
    nonzero = np.nonzero(target)[0]
    c = np.zeros((3,))
    for i in nonzero:
        c += class_colors[i]
    c /= np.sum(target)
    embedding_colors.append(c)
fig = plt.figure(figsize=(10, 10))
plt.scatter(embedding[:, 0], embedding[:, 1], c=embedding_colors)
for i in range(20):
    plt.scatter([-40], [-40], color=class_colors[i], label=dataset.CLASS_NAMES[i
plt.xlim(-60, 60)
plt.legend()
plt.show()
```



5.3 Are some classes harder? (6pts)

Show the per-class performance of your caffenet (scratch) and ResNet (finetuned) models. This is an open-ended question and you may use any performance metric that makes sense. 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?

Looking at the average precision across both networks, the person class has the highest AP for both. I believe that this is a function of the large class imbalance that exists in the VOC dataset. From the VOC paper, we can see that the person class constitutes more than 40% of the total detections in the trainval set.

I also looked at the bottle class in particular because it has the lowests AP for the CaffeNet. I saw that there was a good amount of intraclass variation within the bottle colors, labels, shapes. In addition, some bottles are some level of translucent which could cause further ambiguity in the

features associated with the class. Even on the pretrained ResNet, the bottle class has the lowest AP, leading me to believe it is just a difficult object to classify correctly.

On the other hand, something like cow which is fairly easy to identify, but has a low AP on CaffeNet due to a small number of training samples, sees a large increase do to pretraining. Its AP on CaffeNet is significantly lower than Dining Table, TV Monitor, and Chair, but with pretraining it gets a higher AP.

In [7]:

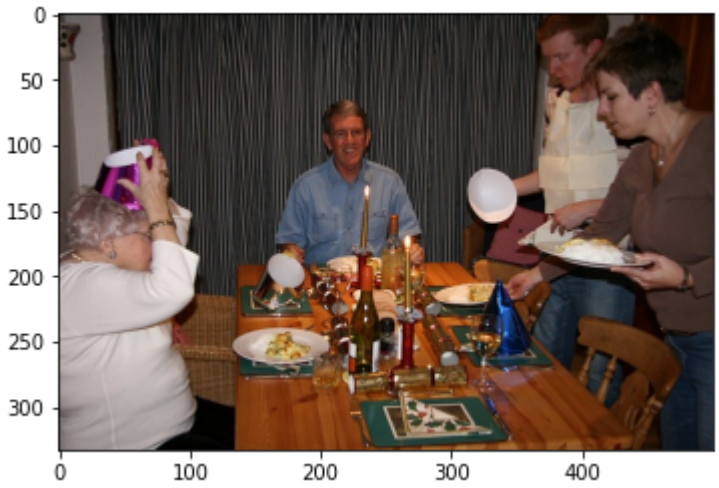
```
from utils import eval_dataset_map

dataset = VOCDataset(split='test', size=224, return_idx=False)
loader = DataLoader(dataset, batch_size=256, num_workers=4)

device = torch.device("cuda:0")
caffenet.to(device)
resnet.to(device)

AP, mAP = eval_dataset_map(caffenet, device, loader)
print("CaffeNet mAP: {}".format(mAP))
for i, cl in enumerate(dataset.CLASS_NAMES):
    print("{} AP: {}".format(cl, AP[i]))

print("\n")

AP, mAP = eval_dataset_map(resnet, device, loader)
print("ResNet: {}".format(mAP))
for i, cl in enumerate(dataset.CLASS_NAMES):
    print("{} AP: {}".format(cl, AP[i]))
```

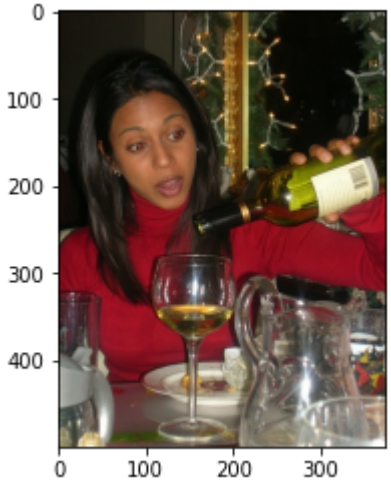
```
CaffeNet mAP: 0.41472539350365956
aeroplane AP: 0.6065504026749763
bicycle AP: 0.41801974981382123
bird AP: 0.3611896311683254
boat AP: 0.4849899090288321
bottle AP: 0.15331735760720447
bus AP: 0.37967164871846043
car AP: 0.6574895040158589
cat AP: 0.3818925820701358
chair AP: 0.3531412948386369
cow AP: 0.21356614445691155
diningtable AP: 0.2978809862687926
dog AP: 0.30000526286964446
horse AP: 0.6713839272249968
motorbike AP: 0.5460125968488998
person AP: 0.7818792689454563
pottedplant AP: 0.19232334763930692
sheep AP: 0.27520590825510505
sofa AP: 0.30421640841885306
train AP: 0.552619860721478
tvmonitor AP: 0.36315207848749625


ResNet: 0.8543920568846026
aeroplane AP: 0.9585709949751373
bicycle AP: 0.9346132642884324
bird AP: 0.9199991863050734
boat AP: 0.9178930703185753
bottle AP: 0.5686803123738275
bus AP: 0.8556680428725884
```

```
car AP: 0.9386909914376396
cat AP: 0.9135866189157451
chair AP: 0.7112527588037817
cow AP: 0.8186023131387906
diningtable AP: 0.7920118251635698
dog AP: 0.905742171434507
horse AP: 0.9297262597235341
motorbike AP: 0.8997696414856904
person AP: 0.9660581094044649
pottedplant AP: 0.7025028558638028
sheep AP: 0.8101225863293035
sofa AP: 0.7871556389816792
train AP: 0.9646186011264887
tvmonitor AP: 0.7925758947494211
```

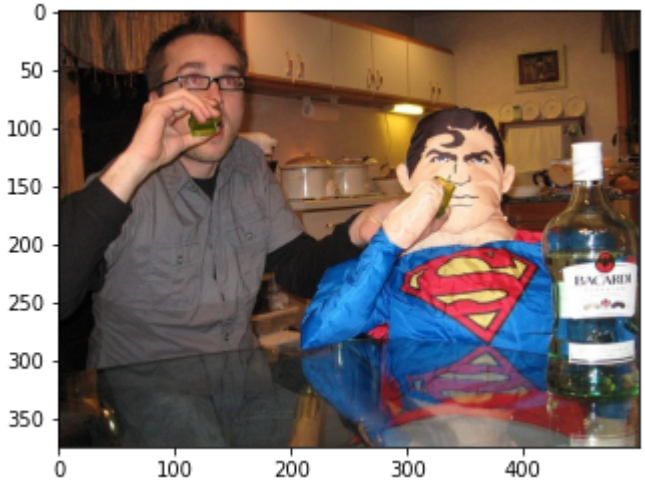
In [5]:

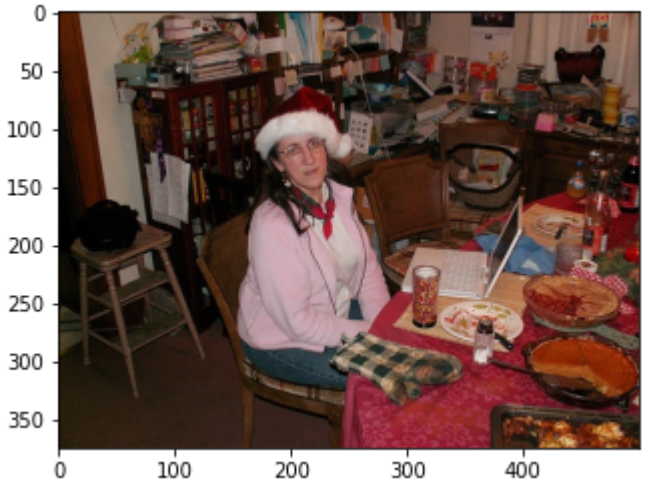
```
i = 0
idx = 0
while i < 10:
    _, target, wgt = dataset.__getitem__(idx)
    if target[4] and wgt[4]:
        image = dataset.get_image(idx)
        plt.imshow(image)
        plt.show()
        i += 1
    idx += 1
```

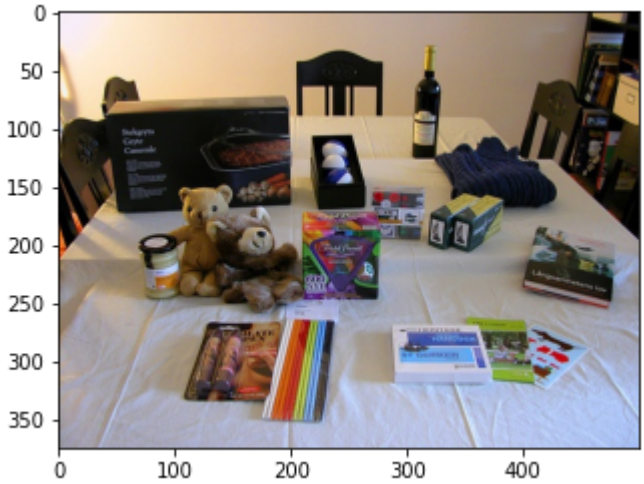





0
50
100
150
200
250
300
350
0
100
200
300
400

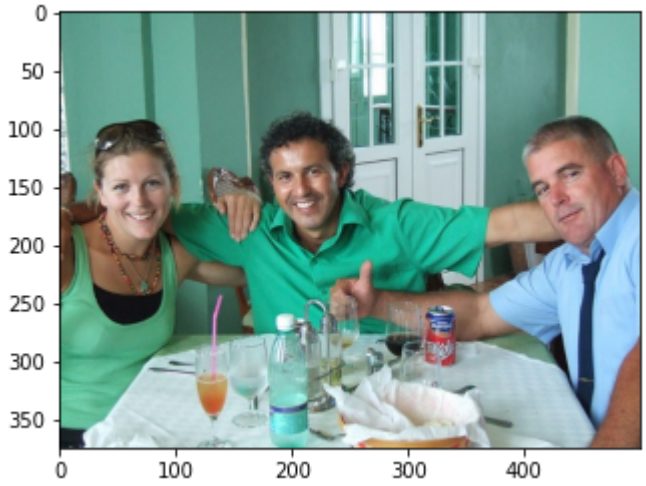

0
50
100
150
200
250
300
350
0
100
200
300
400


0
50
100
150
200
250
300
350
0
100
200
300
400


0
50
100
150
200
250
300
350
0
100
200
300
400


0
100
200
300
400
0
100
200
300


0
50
100
150
200
250
300
350
0
100
200
300
400


0
50
100
150
200
250
300
350
0
100
200
300
400


0
50
100
150
200
250
300
350
0
100
200
300
400

In [ ]: