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 = 256
    # input batch size for testing
    test_batch_size=256
    # number of epochs to train for
    epochs = 15
```

```
    # learning rate
    lr = .001
    # 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
    step_size = 3

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

In [3]:

```
def calculate_accuracy(output, target):
    predicted = torch.argmax(output, dim=1)
    accuracy = torch.sum(predicted == target) / target.shape[0]
    return accuracy
```

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 [4]:

```
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, cnt, 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!
                train_acc = calculate_accuracy(output, target)
                train_log['accuracy'].append(train_acc.cpu().detach().numpy())

            # 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)

YOUR ANSWER HERE

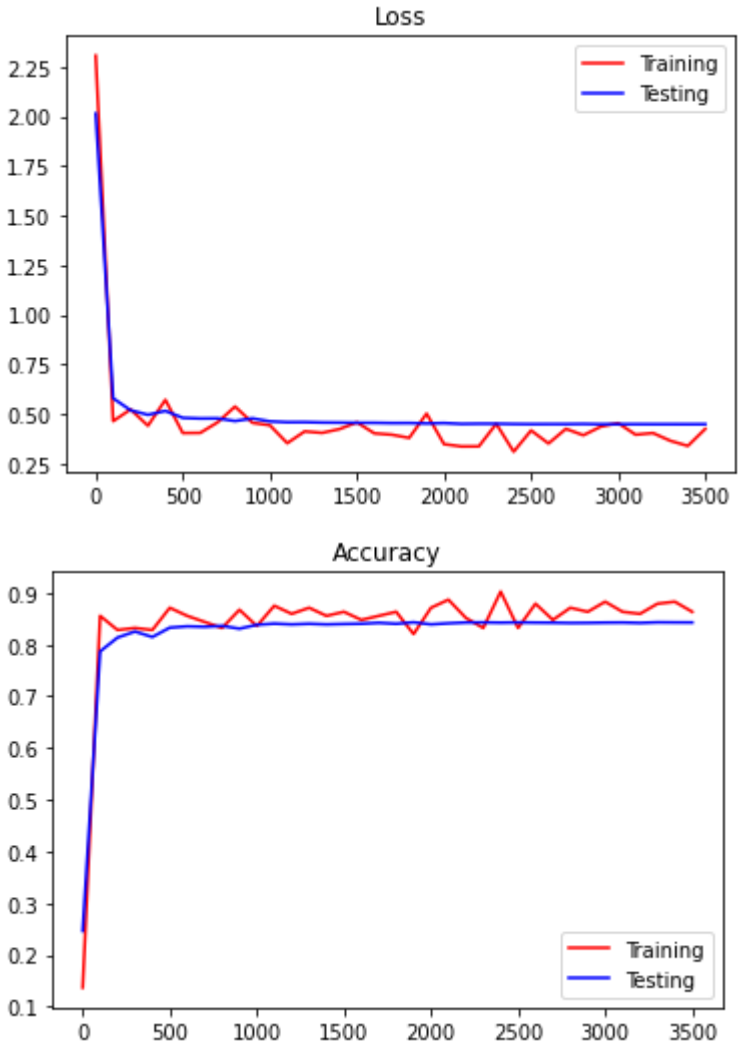
In [14]:
```
#### FEEL FREE TO MODIFY args VARIABLE HERE OR ABOVE ####
# args.gamma = float('inf')
args.epochs = 15

# 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 = 256
args.epochs = 15
args.gamma = 0.7
args.log_every = 100
args.lr = 0.001
args.step_size = 3
args.test_batch_size = 256
args.val_every = 100



Train Epoch: 0 [0/60000 (0%)]   Loss: 2.306719

Test set: Average loss: 2.0144, Accuracy: 2466/10000 (25%)

Train Epoch: 0 [100/60000 (43%)]        Loss: 0.462602

Test set: Average loss: 0.5782, Accuracy: 7868/10000 (79%)

Train Epoch: 0 [200/60000 (85%)]        Loss: 0.521841

Test set: Average loss: 0.5176, Accuracy: 8135/10000 (81%)

Train Epoch: 1 [300/60000 (28%)]        Loss: 0.439650

Test set: Average loss: 0.4948, Accuracy: 8252/10000 (83%)

Train Epoch: 1 [400/60000 (70%)]        Loss: 0.570406
```

```
Test set: Average loss: 0.5143, Accuracy: 8146/10000 (81%)

Train Epoch: 2 [500/60000 (13%)]        Loss: 0.402762

Test set: Average loss: 0.4793, Accuracy: 8327/10000 (83%)

Train Epoch: 2 [600/60000 (55%)]        Loss: 0.403454

Test set: Average loss: 0.4762, Accuracy: 8352/10000 (84%)

Train Epoch: 2 [700/60000 (98%)]        Loss: 0.456762

Test set: Average loss: 0.4767, Accuracy: 8343/10000 (83%)

Train Epoch: 3 [800/60000 (40%)]        Loss: 0.536255

Test set: Average loss: 0.4634, Accuracy: 8361/10000 (84%)

Train Epoch: 3 [900/60000 (83%)]        Loss: 0.454320

Test set: Average loss: 0.4766, Accuracy: 8307/10000 (83%)

Train Epoch: 4 [1000/60000 (26%)]       Loss: 0.442033

Test set: Average loss: 0.4613, Accuracy: 8382/10000 (84%)

Train Epoch: 4 [1100/60000 (68%)]       Loss: 0.351706

Test set: Average loss: 0.4575, Accuracy: 8405/10000 (84%)

Train Epoch: 5 [1200/60000 (11%)]       Loss: 0.410969

Test set: Average loss: 0.4578, Accuracy: 8390/10000 (84%)

Train Epoch: 5 [1300/60000 (53%)]       Loss: 0.403648

Test set: Average loss: 0.4559, Accuracy: 8402/10000 (84%)

Train Epoch: 5 [1400/60000 (96%)]       Loss: 0.422745

Test set: Average loss: 0.4553, Accuracy: 8390/10000 (84%)

Train Epoch: 6 [1500/60000 (38%)]       Loss: 0.456432

Test set: Average loss: 0.4544, Accuracy: 8398/10000 (84%)

Train Epoch: 6 [1600/60000 (81%)]       Loss: 0.401154

Test set: Average loss: 0.4542, Accuracy: 8403/10000 (84%)

Train Epoch: 7 [1700/60000 (23%)]       Loss: 0.394411

Test set: Average loss: 0.4530, Accuracy: 8420/10000 (84%)

Train Epoch: 7 [1800/60000 (66%)]       Loss: 0.378188

Test set: Average loss: 0.4532, Accuracy: 8402/10000 (84%)

Train Epoch: 8 [1900/60000 (9%)]        Loss: 0.500772
```

```
Test set: Average loss: 0.4512, Accuracy: 8430/10000 (84%)

Train Epoch: 8 [2000/60000 (51%)]       Loss: 0.346715

Test set: Average loss: 0.4529, Accuracy: 8391/10000 (84%)

Train Epoch: 8 [2100/60000 (94%)]       Loss: 0.335110

Test set: Average loss: 0.4489, Accuracy: 8411/10000 (84%)

Train Epoch: 9 [2200/60000 (36%)]       Loss: 0.335404

Test set: Average loss: 0.4498, Accuracy: 8426/10000 (84%)

Train Epoch: 9 [2300/60000 (79%)]       Loss: 0.449421

Test set: Average loss: 0.4497, Accuracy: 8428/10000 (84%)

Train Epoch: 10 [2400/60000 (21%)]      Loss: 0.308626

Test set: Average loss: 0.4484, Accuracy: 8423/10000 (84%)

Train Epoch: 10 [2500/60000 (64%)]      Loss: 0.415795

Test set: Average loss: 0.4478, Accuracy: 8428/10000 (84%)

Train Epoch: 11 [2600/60000 (6%)]       Loss: 0.350154

Test set: Average loss: 0.4480, Accuracy: 8424/10000 (84%)

Train Epoch: 11 [2700/60000 (49%)]      Loss: 0.423942

Test set: Average loss: 0.4480, Accuracy: 8422/10000 (84%)

Train Epoch: 11 [2800/60000 (91%)]      Loss: 0.391641

Test set: Average loss: 0.4487, Accuracy: 8417/10000 (84%)

Train Epoch: 12 [2900/60000 (34%)]      Loss: 0.435023

Test set: Average loss: 0.4473, Accuracy: 8419/10000 (84%)

Train Epoch: 12 [3000/60000 (77%)]      Loss: 0.452686

Test set: Average loss: 0.4475, Accuracy: 8424/10000 (84%)

Train Epoch: 13 [3100/60000 (19%)]      Loss: 0.394567

Test set: Average loss: 0.4476, Accuracy: 8426/10000 (84%)

Train Epoch: 13 [3200/60000 (62%)]      Loss: 0.402607

Test set: Average loss: 0.4474, Accuracy: 8418/10000 (84%)

Train Epoch: 14 [3300/60000 (4%)]       Loss: 0.363531

Test set: Average loss: 0.4474, Accuracy: 8430/10000 (84%)

Train Epoch: 14 [3400/60000 (47%)]      Loss: 0.337694
```

```
Test set: Average loss: 0.4473, Accuracy: 8428/10000 (84%)

Train Epoch: 14 [3500/60000 (89%)]      Loss: 0.423512

Test set: Average loss: 0.4474, Accuracy: 8427/10000 (84%)
```



These hyperparamters worked because lr = 0.001 is the optimal for initial epochs which is decreased by factor of 0.7 in every 3 epochs. So the accuracy and loss saturates just after initial 2-3 epochs. Further improvement only possible with non-linear layers.

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 [15]:

```
def param_count(model):
    sum = 0
    for layers in model.parameters():
        if layers.requires_grad:
            sum += torch.numel(layers)
```

```
    return sum
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.

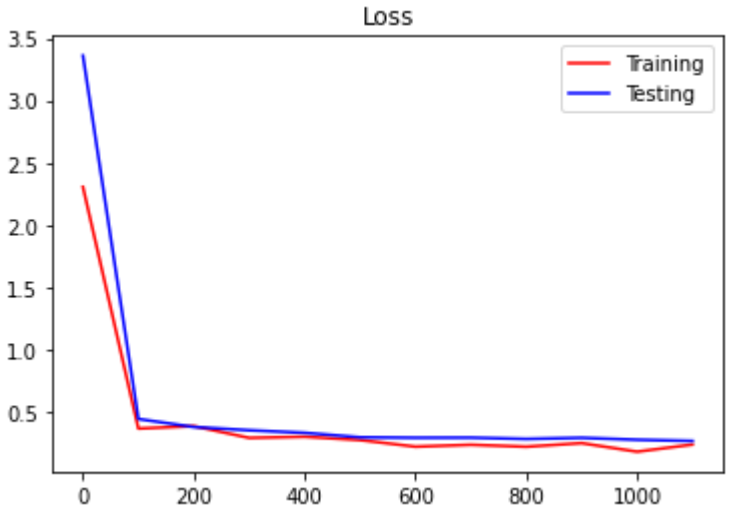
Compute the loss and accuracy curves on train and test sets after 5 epochs. The accuracy should be around 90% or higher.

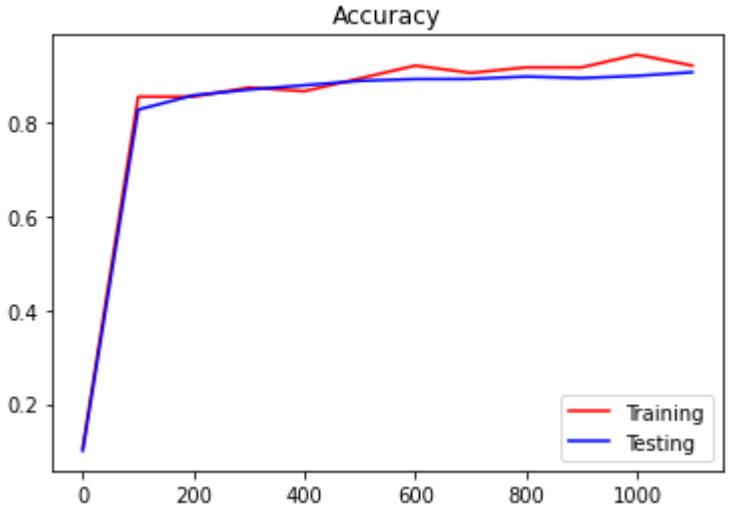
In [5]:

```
args.epochs = 5
args.lr = 0.01
args.step_size = 2

main()
```

```
/home/akshay/anaconda3/envs/materials2/lib/python3.9/site-packages/torchvision/datasets/mnist.py:498: UserWarning: The given NumPy array is not writeable, and PyTorch does not support non-writeable tensors. This means you can write to the underlying (supposedly non-writeable) NumPy array using the tensor. You may want to copy the array to protect its data or make it writeable before converting it to a tensor. This type of warning will be suppressed for the rest of this program. (Triggered internally at  /opt/conda/conda-bld/pytorch_1631630797748/work/torch/csrc/utils/tensor_numpy.cpp:180.)
  return torch.from_numpy(parsed.astype(m[2], copy=False)).view(*s)
Train Epoch: 0 [0/60000 (0%)]   Loss: 2.306187

Test set: Average loss: 3.3595, Accuracy: 1000/10000 (10%)

Train Epoch: 0 [100/60000 (43%)]        Loss: 0.368842

Test set: Average loss: 0.4455, Accuracy: 8279/10000 (83%)

Train Epoch: 0 [200/60000 (85%)]        Loss: 0.388116

Test set: Average loss: 0.3812, Accuracy: 8585/10000 (86%)

Train Epoch: 1 [300/60000 (28%)]        Loss: 0.293764

Test set: Average loss: 0.3550, Accuracy: 8706/10000 (87%)

Train Epoch: 1 [400/60000 (70%)]        Loss: 0.303087

Test set: Average loss: 0.3345, Accuracy: 8799/10000 (88%)

Train Epoch: 2 [500/60000 (13%)]        Loss: 0.276343

Test set: Average loss: 0.2985, Accuracy: 8893/10000 (89%)

Train Epoch: 2 [600/60000 (55%)]        Loss: 0.223603

Test set: Average loss: 0.2951, Accuracy: 8931/10000 (89%)
```

```
Train Epoch: 2 [700/60000 (98%)]	Loss: 0.238678

Test set: Average loss: 0.2957, Accuracy: 8932/10000 (89%)

Train Epoch: 3 [800/60000 (40%)]	Loss: 0.223235

Test set: Average loss: 0.2856, Accuracy: 8985/10000 (90%)

Train Epoch: 3 [900/60000 (83%)]	Loss: 0.251770

Test set: Average loss: 0.2946, Accuracy: 8951/10000 (90%)

Train Epoch: 4 [1000/60000 (26%)]	Loss: 0.183225

Test set: Average loss: 0.2789, Accuracy: 9000/10000 (90%)

Train Epoch: 4 [1100/60000 (68%)]	Loss: 0.240385

Test set: Average loss: 0.2693, Accuracy: 9077/10000 (91%)
```





Out[5]:

```
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))
  (nonlinear): ReLU()
  (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?

YOUR ANSWER HERE I placed relu non linearity after before pooling of every cnn and after first fcn

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

YOUR ANSWER HERE Without non-linearity the output is just a linear matrix multiplication of input, so it can learn linear decision boundaries, so the maximum accuracy without non-linearity is when data are linearly separated. As that gives a good accuracy the data must be linearly separable to a good extend, and classes are distinct.

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: Resize(): resize the image to size req for simple_cnn it is 64, for caffenet, resnet it is 224 RandomCrop: crops random portions of image(for some models, caffenet) RandomHorizontalFlip(p=0.5): flips the image horizontally with a probability of 0.5 (every 1 in 2) ToTensor(): converts PIL image to tensor(0-1) Normalize(mean = [0.485, 0.456, 0.406], std = [0.229, 0.224, 0.225])]): normalize with imagenet mean and std.

Test Augmentations: Resize(): same as train CenterCrop(): crops from the center, so basically image is unchanged ToTensor(): same as train Normalize(mean = [0.485, 0.456, 0.406], std = [0.229, 0.224, 0.225])]): same as train

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**

For each class, valid ground truths and valid predictions are selected. Valid data are those which has difficulty as 1, that means the corresponding class is not ambigous. Then recall and precision are calculated and precision is calculated as a weighted average of recall and returned for each 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 [3]:
```
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, lr=0.001, batch_size=64, gamma=.5, use_cuda=T
            log_every=10, val_every=10, step_size=3)
print(args)
```

```
args.batch_size = 64
args.device = cuda
args.epochs = 5
args.gamma = 0.5
args.inp_size = 64
args.log_every = 10
args.lr = 0.001
args.save_at_end = False
args.save_freq = -1
args.step_size = 3
args.test_batch_size = 256
args.val_every = 10
```

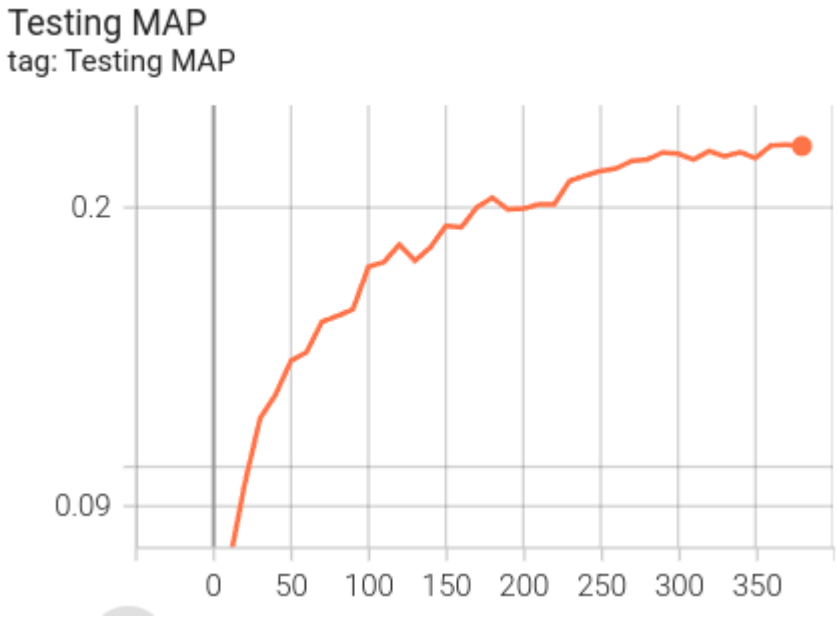
In [4]:
```
# initializes the model
model = SimpleCNN(num_classes=len(VOCDataset.CLASS_NAMES), inp_size=64, c_dim=3)
# initializes Adam optimizer and simple StepLR scheduler
optimizer = torch.optim.Adam(model.parameters(), lr=args.lr)
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=args.step_size,
#scheduler = torch.optim.lr_scheduler.ExponentialLR(optimizer, gamma=args.gamma)
# trains model using your training code and reports test map
test_ap, test_map = trainer.train(args, model, optimizer, scheduler, "simple_cnn
print('test map:', test_map)
```

```
Train Epoch: 0 [0 (0%)] Loss: 0.687220
Train Epoch: 0 [10 (13%)]       Loss: 0.273261
Train Epoch: 0 [20 (25%)]       Loss: 0.244007
Train Epoch: 0 [30 (38%)]       Loss: 0.240804
Train Epoch: 0 [40 (51%)]       Loss: 0.233130
Train Epoch: 0 [50 (63%)]       Loss: 0.245571
Train Epoch: 0 [60 (76%)]       Loss: 0.228997
Train Epoch: 0 [70 (89%)]       Loss: 0.242867
Train Epoch: 1 [80 (1%)]        Loss: 0.215927
Train Epoch: 1 [90 (14%)]       Loss: 0.227102
Train Epoch: 1 [100 (27%)]      Loss: 0.232885
```

```
Train Epoch: 1 [110 (39%)]      Loss: 0.198880
Train Epoch: 1 [120 (52%)]      Loss: 0.221349
Train Epoch: 1 [130 (65%)]      Loss: 0.224932
Train Epoch: 1 [140 (77%)]      Loss: 0.213901
Train Epoch: 1 [150 (90%)]      Loss: 0.225216
Train Epoch: 2 [160 (3%)]       Loss: 0.206045
Train Epoch: 2 [170 (15%)]      Loss: 0.208174
Train Epoch: 2 [180 (28%)]      Loss: 0.240340
Train Epoch: 2 [190 (41%)]      Loss: 0.247462
Train Epoch: 2 [200 (53%)]      Loss: 0.208283
Train Epoch: 2 [210 (66%)]      Loss: 0.213972
Train Epoch: 2 [220 (78%)]      Loss: 0.223647
Train Epoch: 2 [230 (91%)]      Loss: 0.205570
Train Epoch: 3 [240 (4%)]       Loss: 0.204614
Train Epoch: 3 [250 (16%)]      Loss: 0.186102
Train Epoch: 3 [260 (29%)]      Loss: 0.203506
Train Epoch: 3 [270 (42%)]      Loss: 0.220028
Train Epoch: 3 [280 (54%)]      Loss: 0.209237
Train Epoch: 3 [290 (67%)]      Loss: 0.190342
Train Epoch: 3 [300 (80%)]      Loss: 0.212667
Train Epoch: 3 [310 (92%)]      Loss: 0.196571
Train Epoch: 4 [320 (5%)]       Loss: 0.184751
Train Epoch: 4 [330 (18%)]      Loss: 0.179340
Train Epoch: 4 [340 (30%)]      Loss: 0.197176
Train Epoch: 4 [350 (43%)]      Loss: 0.218006
Train Epoch: 4 [360 (56%)]      Loss: 0.211968
Train Epoch: 4 [370 (68%)]      Loss: 0.186364
Train Epoch: 4 [380 (81%)]      Loss: 0.182761
Train Epoch: 4 [390 (94%)]      Loss: 0.199606
test map: 0.23708395534703194
```

Results of tuned hyperparameters

Testing MAP vs iteration number



Training loss vs iteration number

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.

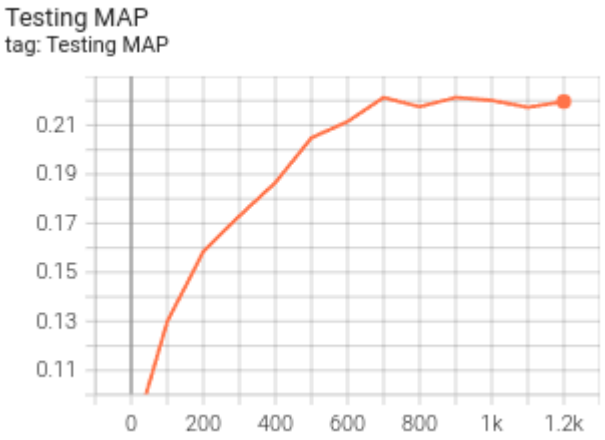
To use TensorBoard in Pytorch, there are two options: TensorBoard in Pytorch (for Pytorch >= 1.1.0) or TensorBoardX - a third party library. Following these links to add code in `trainer.py` to visualize the testing MAP and training loss in Tensorboard. *You may have to reload the kernel for these changes 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 [7]:
```
args = ARGS(epochs=5, batch_size=20, lr=0.001, inp_size=64, val_every=100, step_
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: 0.705692
Train Epoch: 0 [100 (40%)]      Loss: 0.237026
Train Epoch: 0 [200 (80%)]      Loss: 0.266471
Train Epoch: 1 [300 (20%)]      Loss: 0.241846
Train Epoch: 1 [400 (59%)]      Loss: 0.249883
Train Epoch: 1 [500 (99%)]      Loss: 0.219120
Train Epoch: 2 [600 (39%)]      Loss: 0.166754
Train Epoch: 2 [700 (79%)]      Loss: 0.192238
Train Epoch: 3 [800 (19%)]      Loss: 0.215409
Train Epoch: 3 [900 (59%)]      Loss: 0.211660
Train Epoch: 3 [1000 (98%)]     Loss: 0.199148
Train Epoch: 4 [1100 (38%)]     Loss: 0.160793
Train Epoch: 4 [1200 (78%)]     Loss: 0.207072
test map: 0.22557861191616416
```

INSERT YOUR TENSORBOARD SCREENSHOTS HERE Testing MAP vs iteration number



Training loss vs iteration number

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 [3]:

```
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(c_dim, 96, 11, stride=4)
        self.conv2 = nn.Conv2d(96, 256, 5, stride=1, padding=(2,2))
        self.conv3 = nn.Conv2d(256, 384, 3, stride=1, padding=(1,1))
        self.conv4 = nn.Conv2d(384, 384, 3, 1, 1)
        self.conv5 = nn.Conv2d(384, 256, 3, 1, 1)
        self.fcn1 = nn.Linear(5*5*256, 4096)
        self.fcn2 = nn.Linear(4096, 4096)
        self.fcn3 = nn.Linear(4096, num_classes)
        self.regular = nn.Dropout(0.5)
        self.activation = nn.ReLU()
        self.pooling = nn.MaxPool2d(3, 2)

    def forward(self, x):
        out1 = self.pooling(self.activation(self.conv1(x)))
        out2 = self.pooling(self.activation(self.conv2(out1)))
        out3 = self.activation(self.conv3(out2))
        out4 = self.activation(self.conv4(out3))
        out5 = self.pooling(self.activation(self.conv5(out4)))
        out5 = out5.reshape(out5.shape[0], -1)
        out6 = self.regular(self.activation(self.fcn1(out5)))
        out7 = self.regular(self.activation(self.fcn2(out6)))
        out8 = self.fcn3(out7)
        return out8

    #this is for q5 to return pool5
    def forward_analysis_pool5(self, x):
        out1 = self.pooling(self.activation(self.conv1(x)))
        out2 = self.pooling(self.activation(self.conv2(out1)))
        out3 = self.activation(self.conv3(out2))
        out4 = self.activation(self.conv4(out3))
        out5 = self.pooling(self.activation(self.conv5(out4)))
        out5 = out5.reshape(out5.shape[0], -1)
        return out5

    #this is for q5 to return fc7
    def forward_analysis_fc7(self, x):
        out1 = self.pooling(self.activation(self.conv1(x)))
        out2 = self.pooling(self.activation(self.conv2(out1)))
        out3 = self.activation(self.conv3(out2))
        out4 = self.activation(self.conv4(out3))
        out5 = self.pooling(self.activation(self.conv5(out4)))
        out5 = out5.reshape(out5.shape[0], -1)
        out6 = self.regular(self.activation(self.fcn1(out5)))
```

```
        out7 = self.regular(self.activation(self.fcn2(out6)))
        return out7
```

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)

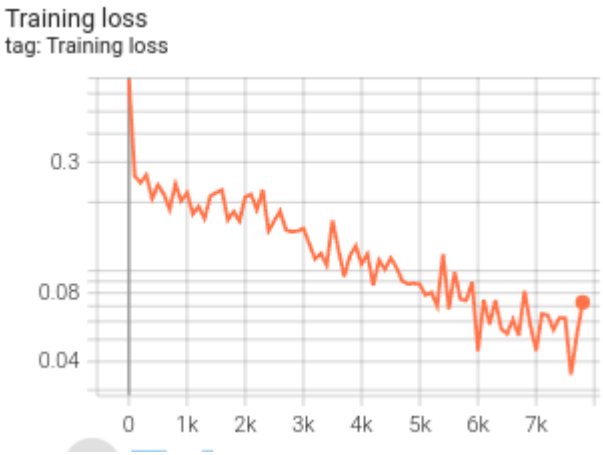
In [4]:

```
args = ARGS(batch_size=32, epochs=50,lr=0.0001, save_at_end=True, \
            save_freq=10, use_cuda=True, val_every=250, step_size=20, gamma=0.2)
model = CaffeNet()
optimizer = torch.optim.Adam(model.parameters(), args.lr)
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, args.step_size, args.gamm
#scheduler = torch.optim.lr_scheduler.ExponentialLR(optimizer, args.gamma)
test_ap, test_map = trainer.train(args, model, optimizer, scheduler, "caffenet")
print('test map:', test_map)
```

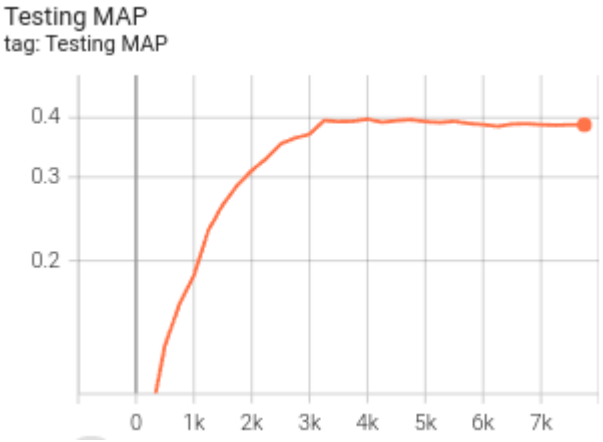
```
Train Epoch: 0 [0 (0%)] Loss: 0.680610
Train Epoch: 0 [100 (64%)]      Loss: 0.245385
Train Epoch: 1 [200 (27%)]      Loss: 0.214580
Train Epoch: 1 [300 (91%)]      Loss: 0.195987
Train Epoch: 2 [400 (55%)]      Loss: 0.216594
Train Epoch: 3 [500 (18%)]      Loss: 0.192287
Train Epoch: 3 [600 (82%)]      Loss: 0.215970
Train Epoch: 4 [700 (46%)]      Loss: 0.222798
Train Epoch: 5 [800 (10%)]      Loss: 0.195831
Train Epoch: 5 [900 (73%)]      Loss: 0.181027
Train Epoch: 6 [1000 (37%)]     Loss: 0.191647
Train Epoch: 7 [1100 (1%)]      Loss: 0.202245
Train Epoch: 7 [1200 (64%)]     Loss: 0.174693
Train Epoch: 8 [1300 (28%)]     Loss: 0.217424
Train Epoch: 8 [1400 (92%)]     Loss: 0.172403
Train Epoch: 9 [1500 (55%)]     Loss: 0.192192
Train Epoch: 10 [1600 (19%)]    Loss: 0.204493
Train Epoch: 10 [1700 (83%)]    Loss: 0.161235
Train Epoch: 11 [1800 (46%)]    Loss: 0.163877
Train Epoch: 12 [1900 (10%)]    Loss: 0.126206
Train Epoch: 12 [2000 (74%)]    Loss: 0.163106
Train Epoch: 13 [2100 (38%)]    Loss: 0.172990
Train Epoch: 14 [2200 (1%)]     Loss: 0.125785
Train Epoch: 14 [2300 (65%)]    Loss: 0.173278
Train Epoch: 15 [2400 (29%)]    Loss: 0.183991
```

```
Train Epoch: 15 [2500 (92%)]    Loss: 0.117062
Train Epoch: 16 [2600 (56%)]    Loss: 0.163846
Train Epoch: 17 [2700 (20%)]    Loss: 0.134695
Train Epoch: 17 [2800 (83%)]    Loss: 0.118578
Train Epoch: 18 [2900 (47%)]    Loss: 0.115297
Train Epoch: 19 [3000 (11%)]    Loss: 0.091669
Train Epoch: 19 [3100 (75%)]    Loss: 0.107353
Train Epoch: 20 [3200 (38%)]    Loss: 0.099787
Train Epoch: 21 [3300 (2%)]     Loss: 0.107063
Train Epoch: 21 [3400 (66%)]    Loss: 0.100759
Train Epoch: 22 [3500 (29%)]    Loss: 0.097955
Train Epoch: 22 [3600 (93%)]    Loss: 0.097192
Train Epoch: 23 [3700 (57%)]    Loss: 0.057685
Train Epoch: 24 [3800 (20%)]    Loss: 0.086066
Train Epoch: 24 [3900 (84%)]    Loss: 0.083796
Train Epoch: 25 [4000 (48%)]    Loss: 0.095654
Train Epoch: 26 [4100 (11%)]    Loss: 0.103063
Train Epoch: 26 [4200 (75%)]    Loss: 0.102911
Train Epoch: 27 [4300 (39%)]    Loss: 0.083479
Train Epoch: 28 [4400 (3%)]     Loss: 0.079015
Train Epoch: 28 [4500 (66%)]    Loss: 0.097977
Train Epoch: 29 [4600 (30%)]    Loss: 0.078154
Train Epoch: 29 [4700 (94%)]    Loss: 0.080005
Train Epoch: 30 [4800 (57%)]    Loss: 0.080714
Train Epoch: 31 [4900 (21%)]    Loss: 0.070169
Train Epoch: 31 [5000 (85%)]    Loss: 0.093791
Train Epoch: 32 [5100 (48%)]    Loss: 0.067551
Train Epoch: 33 [5200 (12%)]    Loss: 0.081107
Train Epoch: 33 [5300 (76%)]    Loss: 0.060450
Train Epoch: 34 [5400 (39%)]    Loss: 0.062620
Train Epoch: 35 [5500 (3%)]     Loss: 0.072301
Train Epoch: 35 [5600 (67%)]    Loss: 0.066797
Train Epoch: 36 [5700 (31%)]    Loss: 0.066436
Train Epoch: 36 [5800 (94%)]    Loss: 0.065772
Train Epoch: 37 [5900 (58%)]    Loss: 0.063835
Train Epoch: 38 [6000 (22%)]    Loss: 0.056233
Train Epoch: 38 [6100 (85%)]    Loss: 0.051230
Train Epoch: 39 [6200 (49%)]    Loss: 0.049546
Train Epoch: 40 [6300 (13%)]    Loss: 0.055738
Train Epoch: 40 [6400 (76%)]    Loss: 0.034720
Train Epoch: 41 [6500 (40%)]    Loss: 0.066654
Train Epoch: 42 [6600 (4%)]     Loss: 0.078433
Train Epoch: 42 [6700 (68%)]    Loss: 0.027333
Train Epoch: 43 [6800 (31%)]    Loss: 0.047634
Train Epoch: 43 [6900 (95%)]    Loss: 0.031249
Train Epoch: 44 [7000 (59%)]    Loss: 0.030353
Train Epoch: 45 [7100 (22%)]    Loss: 0.061614
Train Epoch: 45 [7200 (86%)]    Loss: 0.038374
Train Epoch: 46 [7300 (50%)]    Loss: 0.031651
Train Epoch: 47 [7400 (13%)]    Loss: 0.038640
Train Epoch: 47 [7500 (77%)]    Loss: 0.052842
Train Epoch: 48 [7600 (41%)]    Loss: 0.043696
Train Epoch: 49 [7700 (4%)]     Loss: 0.033748
Train Epoch: 49 [7800 (68%)]    Loss: 0.034967
test map: 0.37978047054875347
```

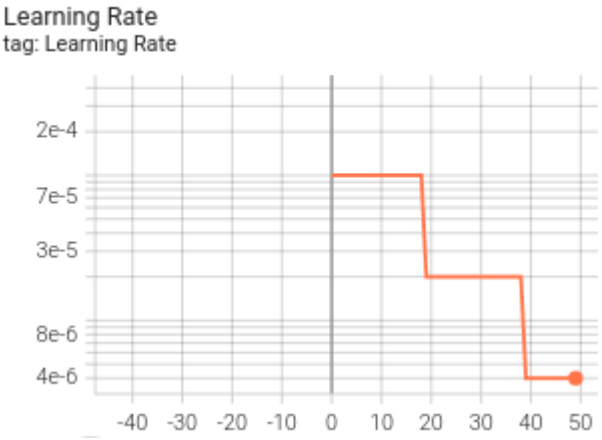
INSERT YOUR TENSORBOARD SCREENSHOTS HERE MAP vs Iteration number



Train loss vs iteration number



Test MAP vs iteration



LR VS iteration

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).

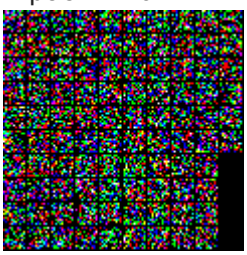
- Write a function to load your model checkpoints.
- Get the weights for conv1 from the loaded model.
- Visualize the weights using the following vis() function.

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.

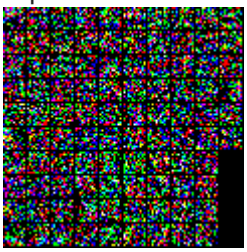
In [5]:
```
import numpy as np
from PIL import Image

# This function plots all the filters in one image.
def vis(conv1):
    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)))
    display(image)
    return image
```

In [6]:
```
# Write your code here to get the conv1 filters for each epoch
conv1s = [np.random.randn(11, 11, 3, 96)]
model = model.to('cpu')
for epoch in range(10, 51, 10):
    print("Epoch:", epoch)
    filename = 'checkpoint-{}-epoch{}.pth'.format('caffenet', epoch)
    model.load_state_dict(torch.load(filename))
    conv1 = model.conv1.weight
    conv1 = (torch.permute(conv1, (2, 3, 1, 0))).detach().numpy()
    vis(conv1)
# 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.
```

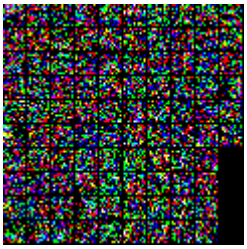
Epoch: 10



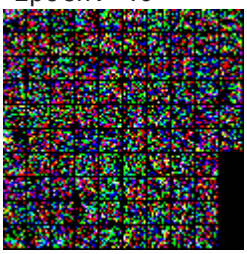
Epoch: 20



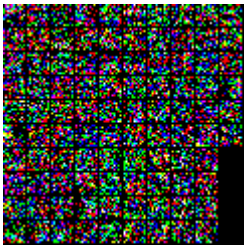
Epoch: 30



Epoch: 40



Epoch: 50



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



#Actually i am writing resnet in resnet.py and importing, so declared below
```

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.

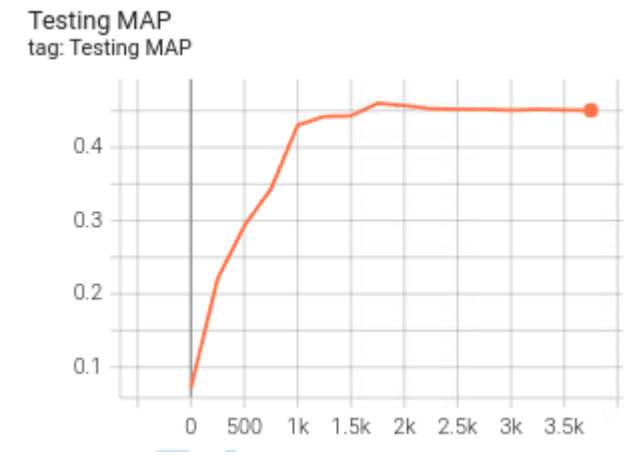
REMEMBER TO SAVE A MODEL AT THE END OF TRAINING

In [4]:

```
args = ARGS(batch_size=64, epochs=50, lr=0.001, save_at_end=True, \
            save_freq=50, use_cuda=True, val_every=250, gamma=0.2, step_size=10)
model = ResNet()
optimizer = torch.optim.Adam(model.parameters(), args.lr)
#scheduler = torch.optim.lr_scheduler.MultiStepLR(optimizer, [20], args.gamma)
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, args.step_size, args.gamm
test_ap, test_map = trainer.train(args, model, optimizer, scheduler, 'resnet18')
print('test map:', test_map)
```

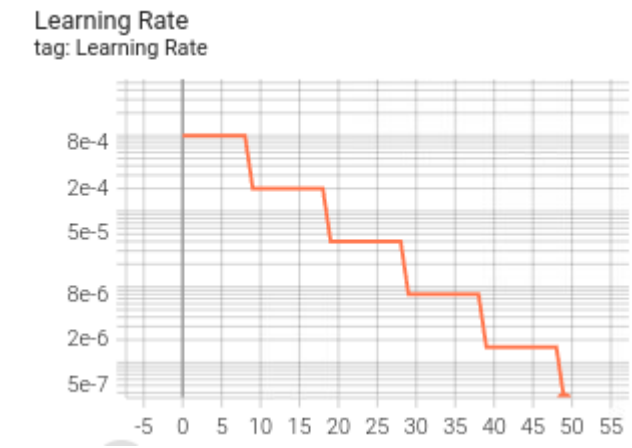
```
Train Epoch: 0 [0 (0%)] Loss: 0.716312
Train Epoch: 1 [100 (27%)]      Loss: 0.204812
Train Epoch: 2 [200 (53%)]      Loss: 0.216067
Train Epoch: 3 [300 (80%)]      Loss: 0.217491
Train Epoch: 5 [400 (6%)]       Loss: 0.196257
Train Epoch: 6 [500 (33%)]      Loss: 0.177712
Train Epoch: 7 [600 (59%)]      Loss: 0.180155
Train Epoch: 8 [700 (86%)]      Loss: 0.167347
Train Epoch: 10 [800 (13%)]     Loss: 0.167981
Train Epoch: 11 [900 (39%)]     Loss: 0.150647
Train Epoch: 12 [1000 (66%)]    Loss: 0.159790
Train Epoch: 13 [1100 (92%)]    Loss: 0.142800
Train Epoch: 15 [1200 (19%)]    Loss: 0.125673
Train Epoch: 16 [1300 (46%)]    Loss: 0.124705
Train Epoch: 17 [1400 (72%)]    Loss: 0.108823
Train Epoch: 18 [1500 (99%)]    Loss: 0.111886
Train Epoch: 20 [1600 (25%)]    Loss: 0.097949
Train Epoch: 21 [1700 (52%)]    Loss: 0.085734
Train Epoch: 22 [1800 (78%)]    Loss: 0.076886
Train Epoch: 24 [1900 (5%)]     Loss: 0.097521
Train Epoch: 25 [2000 (32%)]    Loss: 0.095540
Train Epoch: 26 [2100 (58%)]    Loss: 0.082524
Train Epoch: 27 [2200 (85%)]    Loss: 0.098191
Train Epoch: 29 [2300 (11%)]    Loss: 0.069079
Train Epoch: 30 [2400 (38%)]    Loss: 0.069418
Train Epoch: 31 [2500 (65%)]    Loss: 0.066674
Train Epoch: 32 [2600 (91%)]    Loss: 0.076110
Train Epoch: 34 [2700 (18%)]    Loss: 0.074669
Train Epoch: 35 [2800 (44%)]    Loss: 0.062976
Train Epoch: 36 [2900 (71%)]    Loss: 0.072554
Train Epoch: 37 [3000 (97%)]    Loss: 0.075666
Train Epoch: 39 [3100 (24%)]    Loss: 0.059815
Train Epoch: 40 [3200 (51%)]    Loss: 0.060860
Train Epoch: 41 [3300 (77%)]    Loss: 0.057131
Train Epoch: 43 [3400 (4%)]     Loss: 0.069644
Train Epoch: 44 [3500 (30%)]    Loss: 0.070085
Train Epoch: 45 [3600 (57%)]    Loss: 0.063951
Train Epoch: 46 [3700 (84%)]    Loss: 0.070152
Train Epoch: 48 [3800 (10%)]    Loss: 0.058336
Train Epoch: 49 [3900 (37%)]    Loss: 0.062096
test map: 0.44941460449423615
```

MAP vs iteration

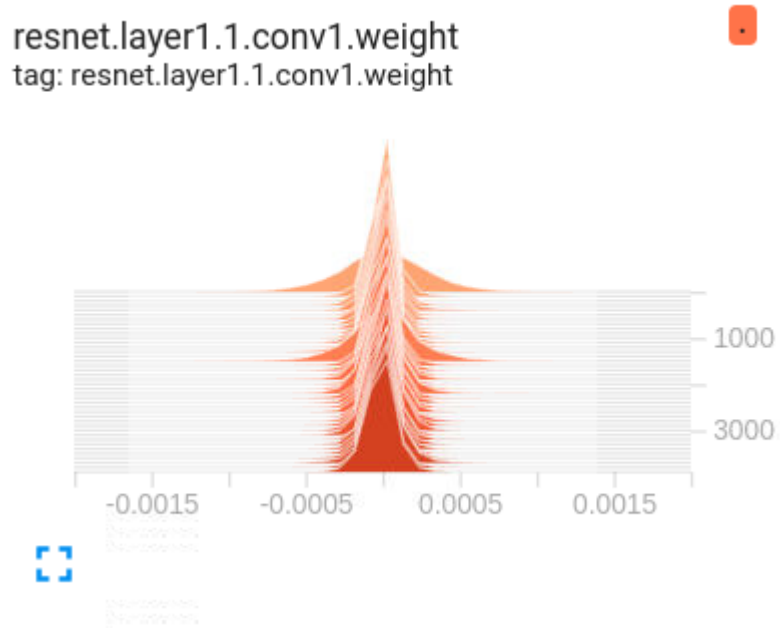


train loss vs iteration

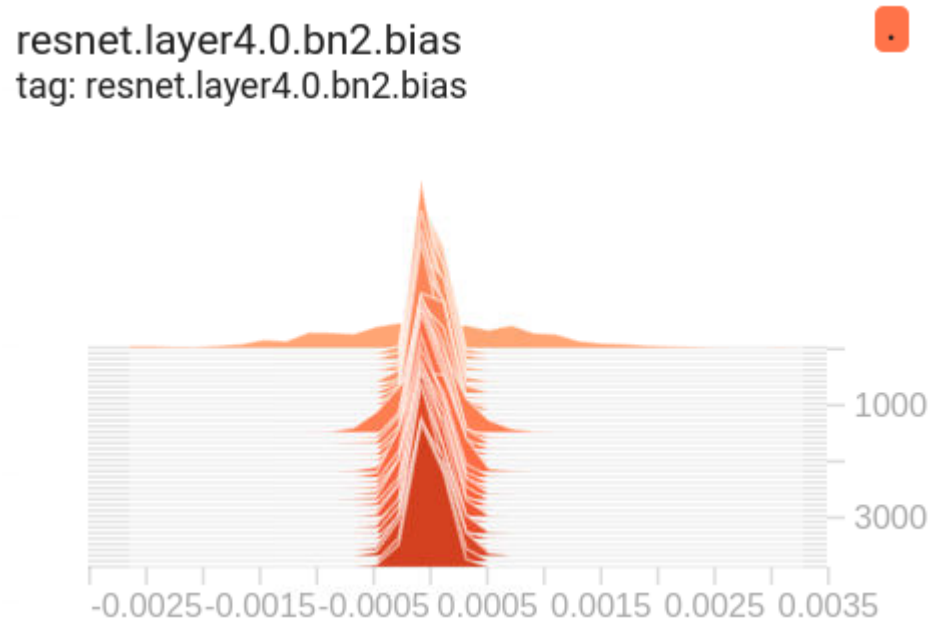
lr vs epoch



layer1.conv1.1.weight



layer4.bn2.0.bias



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).

YOUR ANSWER HERE

First I loaded the resnet with pretrained=True, then i itertated through the state_dict of pretrained resnet and my resnet(with last layer Linear(512, 20)) and assigned the weights of pretrained model to my resnet until the fcn layer is reached.

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
from resnet import ResNet

# 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__()
        org_resnet = models.resnet18(True)
        pretrained_weights = org_resnet.state_dict()
        self.resnet = ResNet()
        new = list(pretrained_weights.items())
```

```
        my_resnet_kvpair = self.resnet.state_dict()
        count  = 0
        for key, value in my_resnet_kvpair.items():
            layer_name, weights = new[count]
            if layer_name == 'fc.weight':
                break
            my_resnet_kvpair[key] = weights
            count += 1
        self.resnet.load_state_dict(my_resnet_kvpair)

    def forward(self, x):
        return self.resnet(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

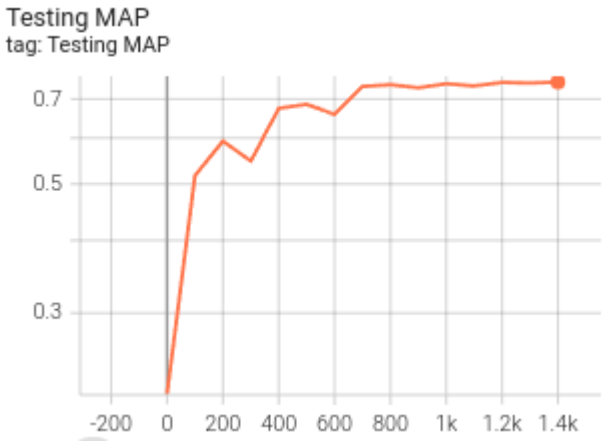
In [2]:

```
args = ARGS(batch_size=32, epochs=10, lr=0.001, save_at_end=True, \
            save_freq=10, use_cuda=True, val_every=100, gamma=0.5, step_size=2,
model = PretrainedResNet()
optimizer = torch.optim.Adam(model.parameters(), args.lr)
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, args.step_size, args.gamm
test_ap, test_map = trainer.train(args, model, optimizer, scheduler, 'resnet18_p
print('test map:', test_map)
```

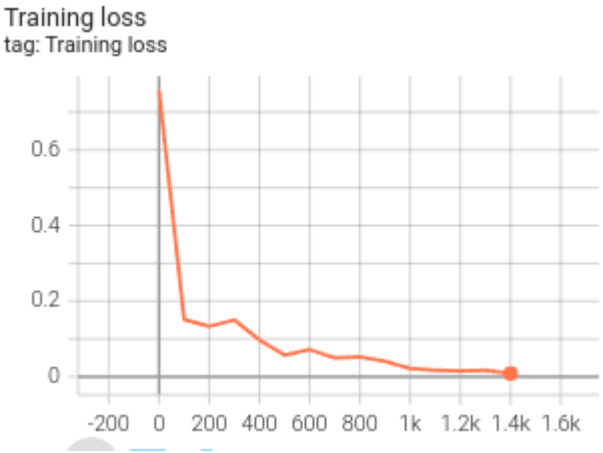
```
Train Epoch: 0 [0 (0%)] Loss: 0.756878
Train Epoch: 0 [100 (64%)]      Loss: 0.150994
Train Epoch: 1 [200 (27%)]      Loss: 0.133344
Train Epoch: 1 [300 (91%)]      Loss: 0.150230
Train Epoch: 2 [400 (55%)]      Loss: 0.097301
Train Epoch: 3 [500 (18%)]      Loss: 0.057413
Train Epoch: 3 [600 (82%)]      Loss: 0.071729
Train Epoch: 4 [700 (46%)]      Loss: 0.049825
Train Epoch: 5 [800 (10%)]      Loss: 0.052487
Train Epoch: 5 [900 (73%)]      Loss: 0.040857
Train Epoch: 6 [1000 (37%)]     Loss: 0.021818
Train Epoch: 7 [1100 (1%)]      Loss: 0.017224
Train Epoch: 7 [1200 (64%)]     Loss: 0.015167
Train Epoch: 8 [1300 (28%)]     Loss: 0.017226
Train Epoch: 8 [1400 (92%)]     Loss: 0.008600
Train Epoch: 9 [1500 (55%)]     Loss: 0.007446
test map: 0.7485116851892839
```

YOUR TENSORBOARD SCREENSHOTS HERE

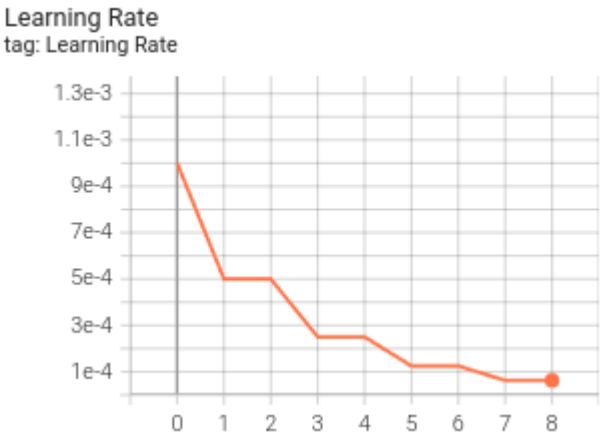
Test map vs iteration



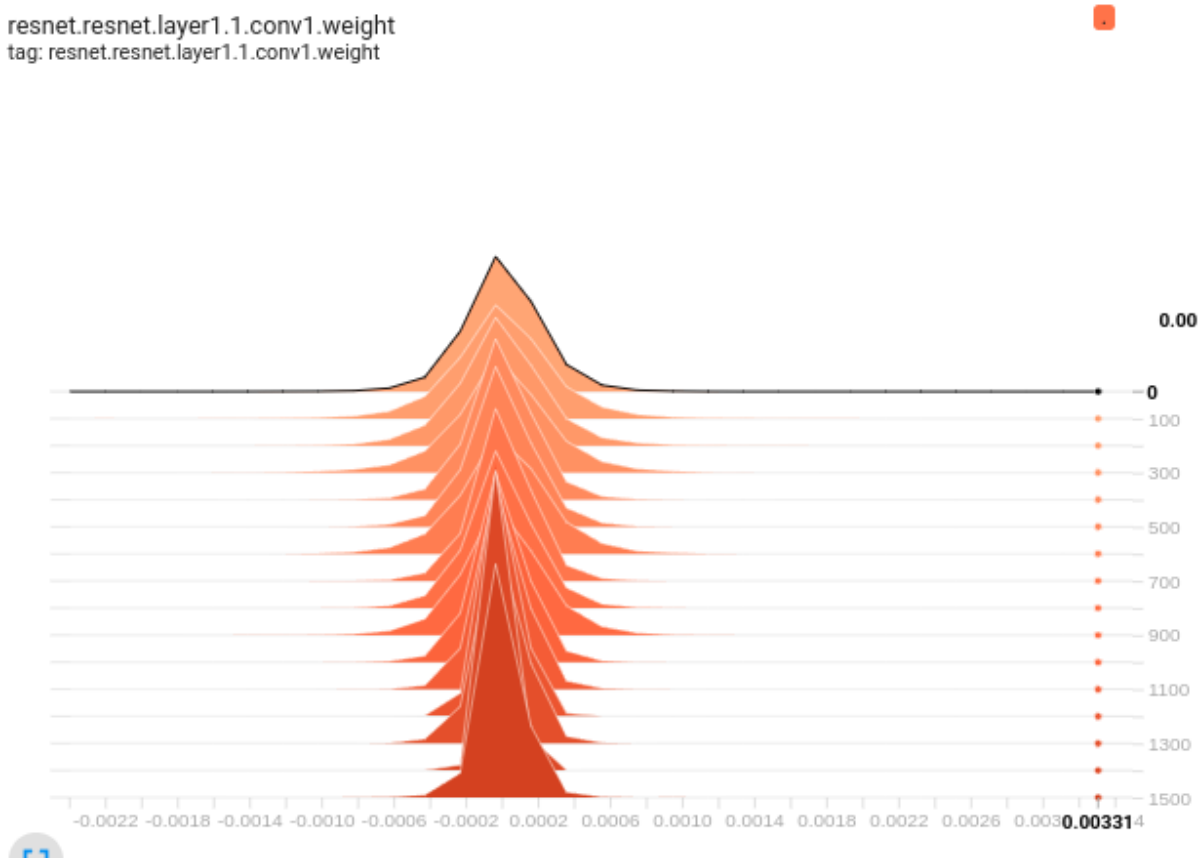
Train loss vs iteration



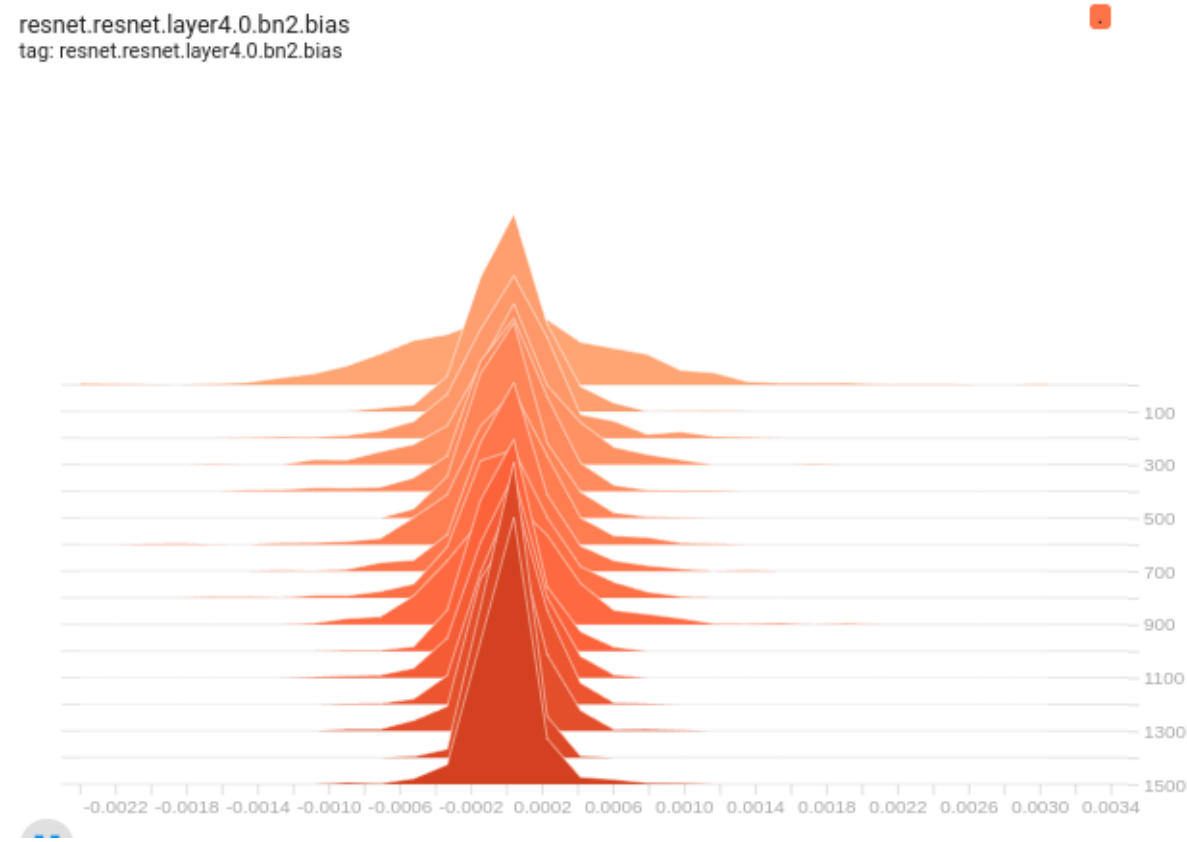
lr vs iteration



layer1.1.conv.1.weight



layer4.0.bn2.bias



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 [1]:
```
import torch
import torch.nn as nn
from torch.utils.data import DataLoader
import torch.nn.functional as F
from torchvision import models
import matplotlib.pyplot as plt
%matplotlib inline
from sklearn.metrics import multilabel_confusion_matrix
import numpy as np
from sklearn.neighbors import NearestNeighbors

import trainer
from utils import ARGS
from simple_cnn import SimpleCNN
from voc_dataset import VOCDataset
import nbimporter
from q5_utils import generate_rand, calculate_features, find_neighbors, plot_met
                     show_image, find_caffenet_features, plot_features, test_met
from sklearn.manifold import TSNE
```

In [2]:
```
# Load all the test images. Pick 3 indices.
test_loader = VOCDataset('test', 224)
req_three_idx = generate_rand(test_loader.index_list, test_loader.anno_list)
```

```
total_resnet_features = np.zeros((0, 512), dtype=np.float32)
total_caffenet_features = np.zeros((0, 6400), dtype=np.float32)
batch_size = 256

# Calculate the features for all the test images.
total_resnet_features, total_caffenet_features = calculate_features(test_loader)

# Fine the nearest neighbors for the 3 images you picked.
resnet_neighbor_indices = find_neighbors(total_resnet_features, req_three_idx)
caffenet_neighbor_indices = find_neighbors(total_caffenet_features, req_three_ic
```

ResNet Neighbours

In [3]:

```
# Plot the images and their neighbors.
show_image(resnet_neighbor_indices, test_loader)
```

```
 --- NEW CLASS ---
Original Image
Image classes are
bottle
person
```

```
Neigbour:  1
Image classes are
chair
diningtable
person
```

```
Neigbour:  2
Image classes are
person
```

Neigbour: 3
Image classes are
person
pottedplant

Neigbour: 4
Image classes are
chair
diningtable
person

Neigbour: 5

 --- NEW CLASS ---
Original Image
Image classes are
aeroplane
bus

Neigbour: 1
Image classes are
aeroplane
person

Neigbour: 2
Image classes are
boat

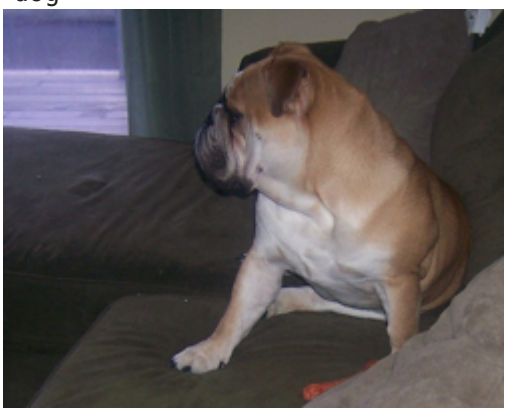
Neigbour: 3
Image classes are
aeroplane

Neigbour: 4
Image classes are
aeroplane

Neigbour: 5

 --- NEW CLASS ---
Original Image
Image classes are
dog

Neigbour: 1
Image classes are
dog



Neigbour: 2
Image classes are
dog
sofa

Neigbour: 3

```
Image classes are
dog
```

```
Neigbour:  4
Image classes are
dog
sofa
```

```
Neigbour:  5
```

CaffeNet Neighbors

In [4]:
```
show_image(caffenet_neighbor_indices, test_loader)
```

```
 --- NEW CLASS ---
Original Image
Image classes are
bottle
person
```

```
Neigbour:  1
Image classes are
bottle
person
```

```
Neigbour:  2
Image classes are
person
```

```
Neigbour:  3
Image classes are
chair
dog
person
```

```
Neigbour:  4
Image classes are
person
```

```
Neigbour:  5
```

```
 --- NEW CLASS ---
Original Image
Image classes are
aeroplane
bus
```

```
Neigbour:  1
Image classes are
aeroplane
person
```

```
Neigbour:  2
Image classes are
boat
```

```
Neigbour:  3
Image classes are
aeroplane
```

Neigbour: 4
Image classes are
boat

Neigbour: 5

 --- NEW CLASS ---
Original Image
Image classes are
dog

Neigbour: 1
Image classes are
person
sofa

Neigbour: 2
Image classes are

```
bottle
diningtable
person
```

```
Neigbour:  3
Image classes are
person
sofa
```

```
Neigbour:  4
Image classes are
chair
person
tvmonitor
```

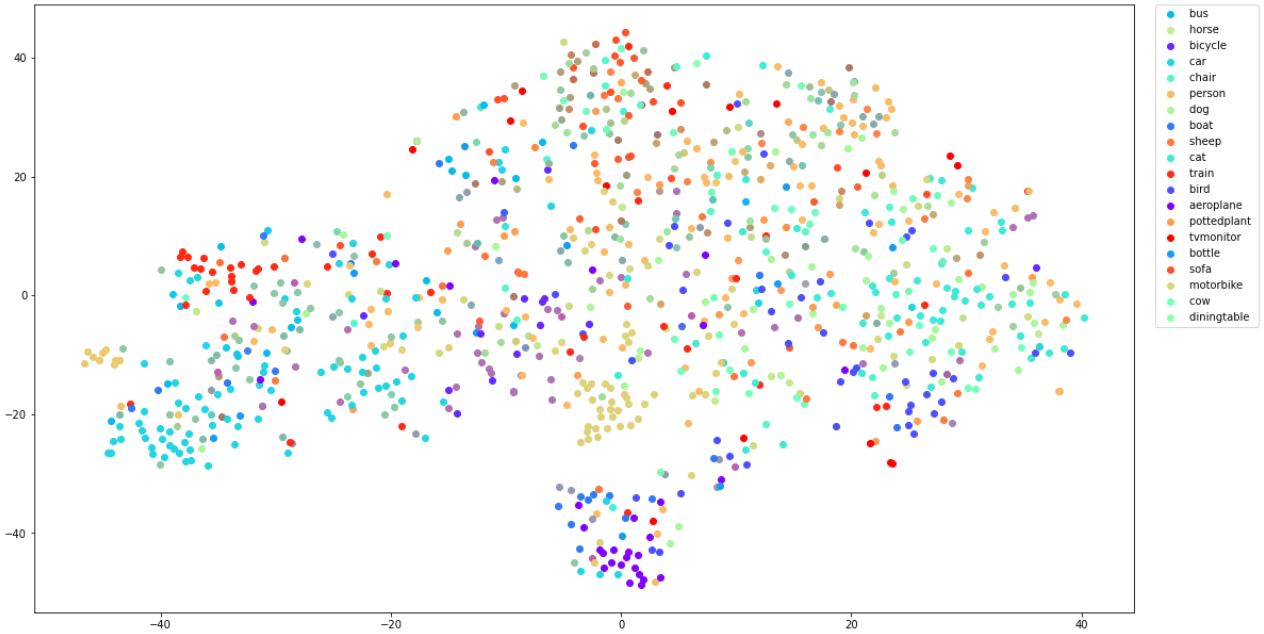
```
Neigbour:  5
```

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 [5]:
```
# plot t-SNE here
caffenet_features, label = find_caffenet_features(test_loader)
tsne_projection = TSNE(2).fit_transform(caffenet_features)
plot_features(tsne_projection, label, test_loader)
```



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?

YOUR ANSWER HERE

For Finetuned Resnet

In [4]:
```
test_loader = VOCDataset('test', 224)
resnet_gt, resnet_pred = test_metrics(test_loader, 'resnet')
confusion_metric_resnet = multilabel_confusion_matrix(resnet_gt, resnet_pred)
resnet_report = {}

for i, class_name in enumerate(test_loader.CLASS_NAMES):
    precision = confusion_metric_resnet[i][1][1] / (confusion_metric_resnet[i][1
    recall = confusion_metric_resnet[i][1][1] / (confusion_metric_resnet[i][1][1
    f1 = 2 * precision * recall / (precision + recall)
    resnet_report[class_name] = [precision, recall, f1]
```

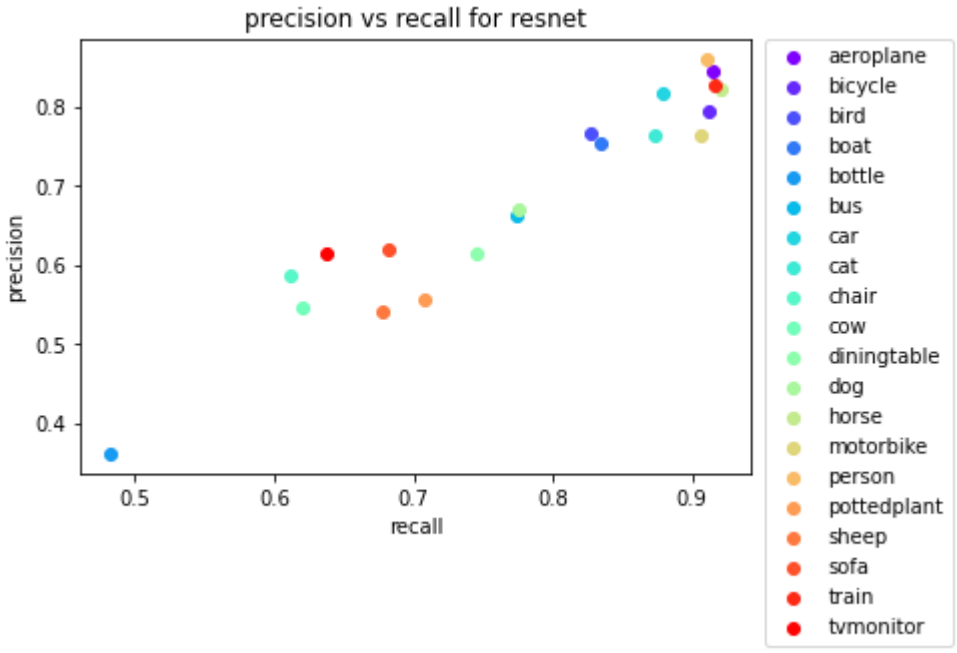
In [5]:
```
resnet_precision_order = sorted(resnet_report.items(), key=lambda x: x[1][0])
```

```
resnet_recall_order = sorted(resnet_report.items(), key=lambda x: x[1][1])
resnet_f1_order = sorted(resnet_report.items(), key=lambda x: x[1][2])
for precision, recall, f1 in zip(resnet_precision_order, resnet_recall_order, re
    print(precision[0], recall[0], f1[0])
```

```
bottle bottle bottle
sheep chair sheep
pottedplant cow cow
cow tvmonitor pottedplant
diningtable sheep chair
chair sofa tvmonitor
sofa pottedplant diningtable
bus diningtable sofa
dog bus bus
tvmonitor dog dog
motorbike bird boat
cat boat motorbike
boat cat cat
bicycle car bird
bird motorbike bicycle
horse person car
train bicycle horse
car aeroplane train
aeroplane train aeroplane
person horse person
```

In [6]:

```
plot_metrics(resnet_report)
```

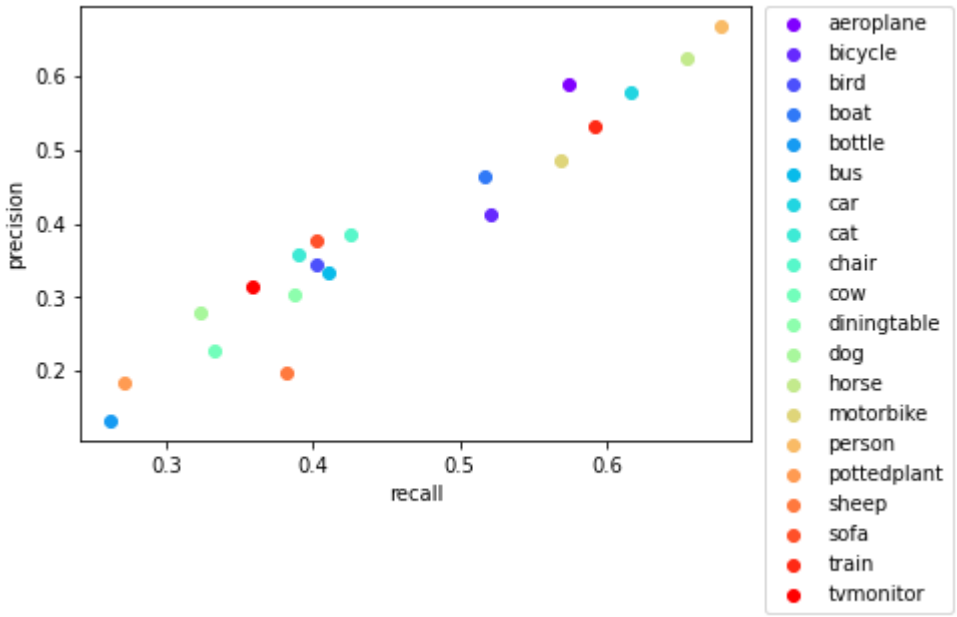


For Caffenet scratch

In [7]:

```
test_loader = VOCDataset('test', 224)
caffenet_report = {}
resnet_gt, resnet_pred = test_metrics(test_loader, 'caffenet')
confusion_metric_caffenet = multilabel_confusion_matrix(resnet_gt, resnet_pred)
for i, class_name in enumerate(test_loader.CLASS_NAMES):
    precision = confusion_metric_caffenet[i][1][1] / (confusion_metric_caffenet[
```

```
    recall = confusion_metric_caffenet[i][1][1] / (confusion_metric_caffenet[i][
    f1 = 2 * precision * recall / (precision + recall)
    caffenet_report[class_name] = [precision, recall, f1]
caffenet_report
plot_metrics(caffenet_report, False)
```



In [8]:

```
caffenet_precision_order = sorted(caffenet_report.items(), key= lambda x: x[1][0
caffenet_recall_order = sorted(caffenet_report.items(), key=lambda x: x[1][1])
caffenet_f1_order = sorted(caffenet_report.items(), key=lambda x: x[1][2])
for precision, recall, f1 in zip(caffenet_precision_order, caffenet_recall_order
    print(precision[0], recall[0], f1[0])
```

```
bottle bottle bottle
sheep pottedplant pottedplant
pottedplant dog sheep
cow cow cow
dog tvmonitor dog
diningtable sheep diningtable
tvmonitor diningtable tvmonitor
bus cat bus
bird sofa bird
cat bird cat
bicycle bus sofa
chair chair chair
sofa boat bicycle
boat bicycle boat
motorbike motorbike motorbike
train aeroplane train
car train car
horse car aeroplane
aeroplane horse horse
person person person
```

Observations

Hardest class: Bottle

Easiest class: Person

From the sorted class list of both resnet and caffenet we can see that the most hard classes are bottle, sheep, cow and pottedplant. The easiest classes are person, aeroplane and train. The pretarining with resnet improves the precision of bottle from 0.1875 to 0.2625. So pretraining helps. Every class (recall, precision, f1-score) increases due to pre training. This is beacuse the imagenet has similar datas in large numbers to the pretrained model is already good in classifying these objects.

For bottle

In [11]:
```
resnet_report['bottle'], caffenet_report['bottle']
```

Out[11]:
```
([0.2875, 0.4825174825174825, 0.3603133159268929], [0.0875, 0.2625, 0.13125])
```

For bottle, most of the data have some other classes in them like person holding a bottle, bottle near dining table, or sofa. So the network activates the other more prominant objects in the image, thereby giving a small output for bottle class. In case of cow, dog, sheep the general features of the animals are captured making it hard to classify them.

For person

In case of person class, the features (like edges, curvature) is very different from most other classes in the dataset. So probability of not detecting person and classyfing non-person as person is really low.

In [10]:
```
resnet_report['person'], caffenet_report['person']
```

Out[10]:
```
([0.8164043872198379, 0.9106382978723404, 0.8609504651747548],
 [0.6609442060085837, 0.6774193548387096, 0.6690803765387401])
```

In [ ]:
```

```

Q6: Improve Performance (20 pts)

Many techniques have been proposed in the literature to improve classification performance for deep networks. In this section, we try to use a recently proposed technique called mixup. The main idea is to augment the training set with linear combinations of images and labels. Read through the paper and modify your model to implement mixup. Report your performance, along with training/test curves, and comparison with baseline in the report.

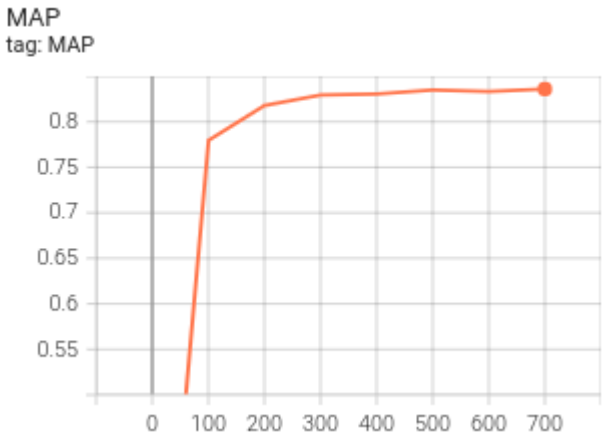
In [1]:
```
# implement mixup regularization here and show performance
import torch
from utils import ARGS
import nbimporter
from q6_utils import train
from q4_imagenet_finetune_pascal import PretrainedResNet
from resnet import ResNet
from q2_caffenet_pascal import CaffeNet
```

Pre-trained resnet

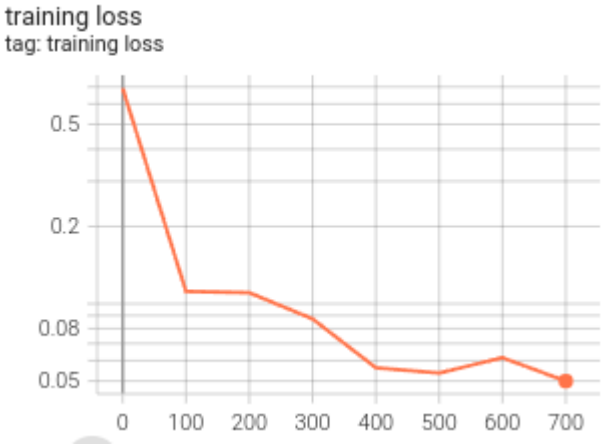
In [2]:
```
args = ARGS(batch_size=64, epochs=10, lr=0.0001, save_at_end=True, \
            save_freq=10, use_cuda=True, val_every=100, gamma=0.5, step_size=2,
model = PretrainedResNet()
optimizer = torch.optim.Adam(model.parameters(), args.lr)
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, args.step_size, args.gamm
test_ap, test_map = train(args, model, optimizer, scheduler, 'resnet18_pretraine
print('test map:', test_map)
```

```
Train Epoch: 0 [0 (0%)] Loss: 0.691422
Train Epoch: 1 [100 (27%)]      Loss: 0.111653
Train Epoch: 2 [200 (53%)]      Loss: 0.110398
Train Epoch: 3 [300 (80%)]      Loss: 0.087299
Train Epoch: 5 [400 (6%)]       Loss: 0.056296
Train Epoch: 6 [500 (33%)]      Loss: 0.053700
Train Epoch: 7 [600 (59%)]      Loss: 0.061616
Train Epoch: 8 [700 (86%)]      Loss: 0.049949
test map: 0.8358738446542814
```

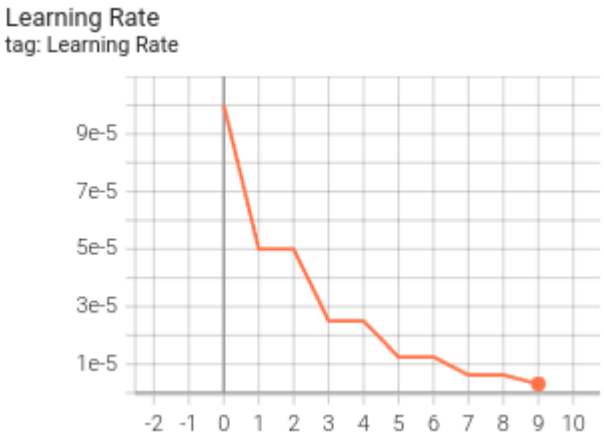
Test map vs iteration



Training loss vs iteration



lr vs iteration



final map without mixup = 0.7485, with mixup 0.8387

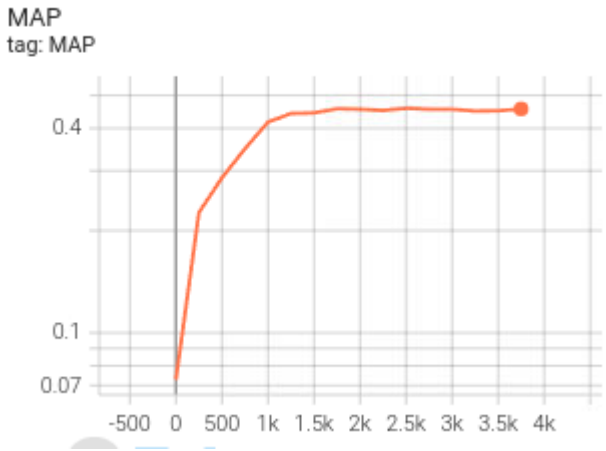
Resnet scratch

In [2]:
```
args = ARGS(batch_size=64, epochs=50, lr=0.001, save_at_end=True, \
            save_freq=50, use_cuda=True, val_every=250, gamma=0.2, step_size=10)
```

```
model = ResNet()
optimizer = torch.optim.Adam(model.parameters(), args.lr)
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, args.step_size, args.gamm
test_ap, test_map = train(args, model, optimizer, scheduler, 'resnet_scratch_mix
print('test map:', test_map)
```

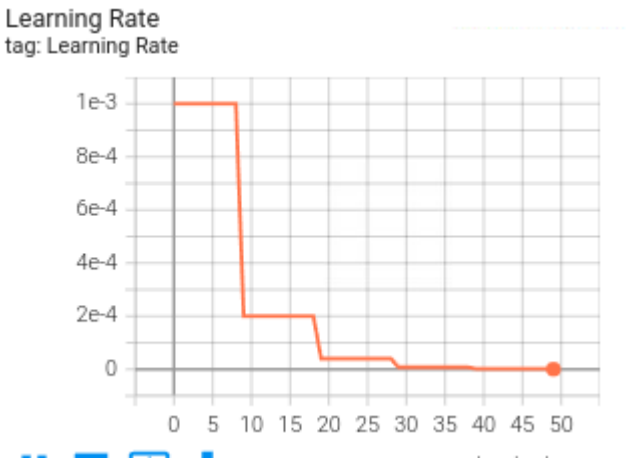
```
Train Epoch: 0 [0 (0%)] Loss: 0.768416
Train Epoch: 1 [100 (27%)]      Loss: 0.222269
Train Epoch: 2 [200 (53%)]      Loss: 0.224068
Train Epoch: 3 [300 (80%)]      Loss: 0.211722
Train Epoch: 5 [400 (6%)]       Loss: 0.217208
Train Epoch: 6 [500 (33%)]      Loss: 0.194764
Train Epoch: 7 [600 (59%)]      Loss: 0.223388
Train Epoch: 8 [700 (86%)]      Loss: 0.247940
Train Epoch: 10 [800 (13%)]     Loss: 0.162076
Train Epoch: 11 [900 (39%)]     Loss: 0.176556
Train Epoch: 12 [1000 (66%)]    Loss: 0.230983
Train Epoch: 13 [1100 (92%)]    Loss: 0.141811
Train Epoch: 15 [1200 (19%)]    Loss: 0.155055
Train Epoch: 16 [1300 (46%)]    Loss: 0.236883
Train Epoch: 17 [1400 (72%)]    Loss: 0.194826
Train Epoch: 18 [1500 (99%)]    Loss: 0.148982
Train Epoch: 20 [1600 (25%)]    Loss: 0.137758
Train Epoch: 21 [1700 (52%)]    Loss: 0.109818
Train Epoch: 22 [1800 (78%)]    Loss: 0.224005
Train Epoch: 24 [1900 (5%)]     Loss: 0.122144
Train Epoch: 25 [2000 (32%)]    Loss: 0.196529
Train Epoch: 26 [2100 (58%)]    Loss: 0.108749
Train Epoch: 27 [2200 (85%)]    Loss: 0.109581
Train Epoch: 29 [2300 (11%)]    Loss: 0.097486
Train Epoch: 30 [2400 (38%)]    Loss: 0.095131
Train Epoch: 31 [2500 (65%)]    Loss: 0.102778
Train Epoch: 32 [2600 (91%)]    Loss: 0.092377
Train Epoch: 34 [2700 (18%)]    Loss: 0.090251
Train Epoch: 35 [2800 (44%)]    Loss: 0.095165
Train Epoch: 36 [2900 (71%)]    Loss: 0.081474
Train Epoch: 37 [3000 (97%)]    Loss: 0.141284
Train Epoch: 39 [3100 (24%)]    Loss: 0.079245
Train Epoch: 40 [3200 (51%)]    Loss: 0.078353
Train Epoch: 41 [3300 (77%)]    Loss: 0.094300
Train Epoch: 43 [3400 (4%)]     Loss: 0.100915
Train Epoch: 44 [3500 (30%)]    Loss: 0.181455
Train Epoch: 45 [3600 (57%)]    Loss: 0.092917
Train Epoch: 46 [3700 (84%)]    Loss: 0.091736
Train Epoch: 48 [3800 (10%)]    Loss: 0.106035
Train Epoch: 49 [3900 (37%)]    Loss: 0.121508
test map: 0.4573248418379056
```

test map vs iteration



train loss vs iteration

learning rate vs epoch



Caffenet scratch

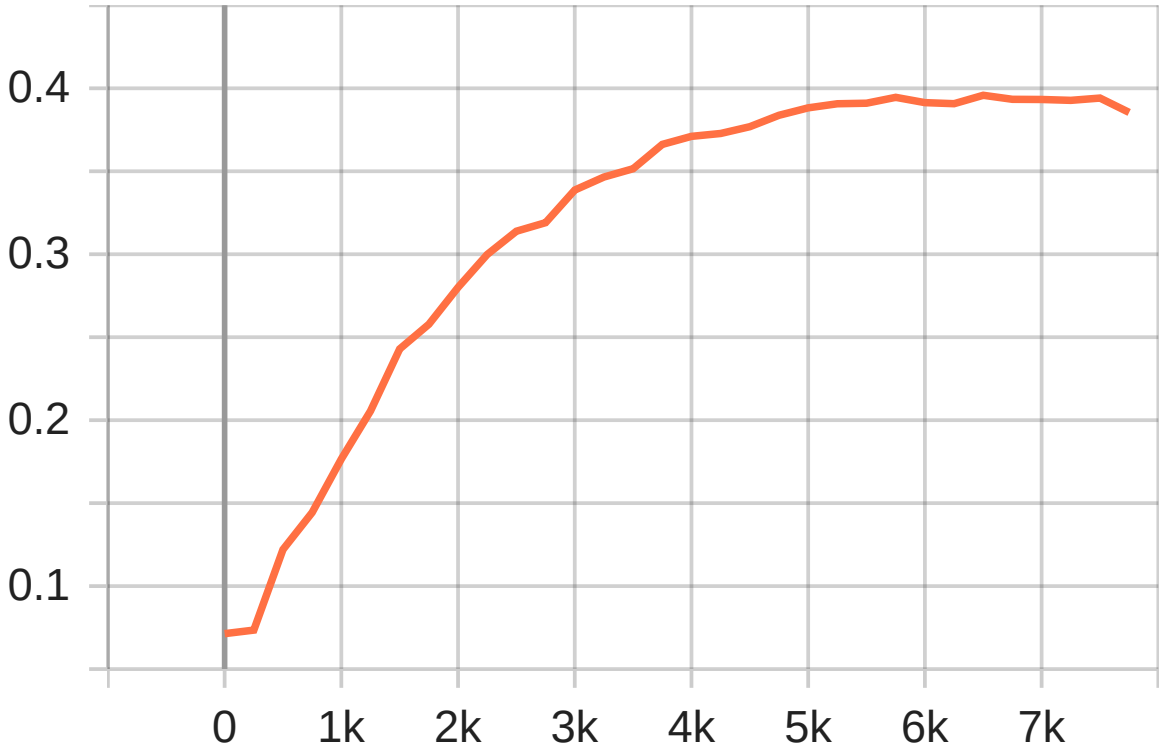
In [2]:
```
args = ARGS(batch_size=32, epochs=50,lr=0.0001, save_at_end=True, \
            save_freq=20, use_cuda=True, val_every=250, step_size=5, gamma=0.8)
model = CaffeNet()
```

```
optimizer = torch.optim.Adam(model.parameters(), args.lr)
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, args.step_size, args.gamm
test_ap, test_map = train(args, model, optimizer, scheduler, 'caffenet_mixup')
print('test map:', test_map)
```

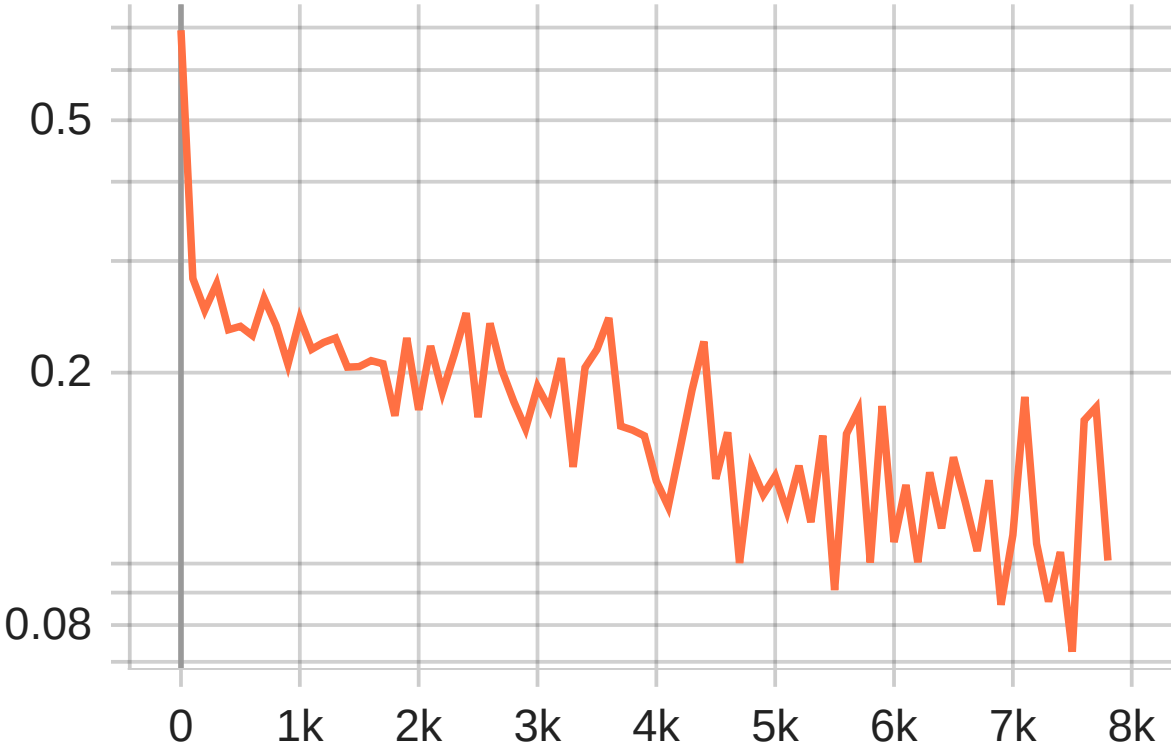
```
Train Epoch: 0 [0 (0%)] Loss: 0.693204
Train Epoch: 0 [100 (64%)]      Loss: 0.281435
Train Epoch: 1 [200 (27%)]      Loss: 0.250984
Train Epoch: 1 [300 (91%)]      Loss: 0.276004
Train Epoch: 2 [400 (55%)]      Loss: 0.233724
Train Epoch: 3 [500 (18%)]      Loss: 0.236591
Train Epoch: 3 [600 (82%)]      Loss: 0.228916
Train Epoch: 4 [700 (46%)]      Loss: 0.261812
Train Epoch: 5 [800 (10%)]      Loss: 0.237398
Train Epoch: 5 [900 (73%)]      Loss: 0.206168
Train Epoch: 6 [1000 (37%)]     Loss: 0.243423
Train Epoch: 7 [1100 (1%)]      Loss: 0.217701
Train Epoch: 7 [1200 (64%)]     Loss: 0.223154
Train Epoch: 8 [1300 (28%)]     Loss: 0.226751
Train Epoch: 8 [1400 (92%)]     Loss: 0.204041
Train Epoch: 9 [1500 (55%)]     Loss: 0.204400
Train Epoch: 10 [1600 (19%)]    Loss: 0.208899
Train Epoch: 10 [1700 (83%)]    Loss: 0.206577
Train Epoch: 11 [1800 (46%)]    Loss: 0.171006
Train Epoch: 12 [1900 (10%)]    Loss: 0.226879
Train Epoch: 12 [2000 (74%)]    Loss: 0.174719
Train Epoch: 13 [2100 (38%)]    Loss: 0.220328
Train Epoch: 14 [2200 (1%)]     Loss: 0.186300
Train Epoch: 14 [2300 (65%)]    Loss: 0.213836
Train Epoch: 15 [2400 (29%)]    Loss: 0.248321
Train Epoch: 15 [2500 (92%)]    Loss: 0.170090
Train Epoch: 16 [2600 (56%)]    Loss: 0.239223
Train Epoch: 17 [2700 (20%)]    Loss: 0.201813
Train Epoch: 17 [2800 (83%)]    Loss: 0.180104
Train Epoch: 18 [2900 (47%)]    Loss: 0.163188
Train Epoch: 19 [3000 (11%)]    Loss: 0.190065
Train Epoch: 19 [3100 (75%)]    Loss: 0.175428
Train Epoch: 20 [3200 (38%)]    Loss: 0.210763
Train Epoch: 21 [3300 (2%)]     Loss: 0.142008
Train Epoch: 21 [3400 (66%)]    Loss: 0.203719
Train Epoch: 22 [3500 (29%)]    Loss: 0.217518
Train Epoch: 22 [3600 (93%)]    Loss: 0.243951
Train Epoch: 23 [3700 (57%)]    Loss: 0.164711
Train Epoch: 24 [3800 (20%)]    Loss: 0.162352
Train Epoch: 24 [3900 (84%)]    Loss: 0.158962
Train Epoch: 25 [4000 (48%)]    Loss: 0.134920
Train Epoch: 26 [4100 (11%)]    Loss: 0.123040
Train Epoch: 26 [4200 (75%)]    Loss: 0.151778
Train Epoch: 27 [4300 (39%)]    Loss: 0.187629
Train Epoch: 28 [4400 (3%)]     Loss: 0.223925
Train Epoch: 28 [4500 (66%)]    Loss: 0.136009
Train Epoch: 29 [4600 (30%)]    Loss: 0.161034
Train Epoch: 29 [4700 (94%)]    Loss: 0.100226
Train Epoch: 30 [4800 (57%)]    Loss: 0.142090
Train Epoch: 31 [4900 (21%)]    Loss: 0.128444
Train Epoch: 31 [5000 (85%)]    Loss: 0.137434
Train Epoch: 32 [5100 (48%)]    Loss: 0.120914
Train Epoch: 33 [5200 (12%)]    Loss: 0.142646
Train Epoch: 33 [5300 (76%)]    Loss: 0.116207
Train Epoch: 34 [5400 (39%)]    Loss: 0.159174
```

```
Train Epoch: 35 [5500 (3%)]     Loss: 0.090823
Train Epoch: 35 [5600 (67%)]    Loss: 0.160194
Train Epoch: 36 [5700 (31%)]    Loss: 0.174657
Train Epoch: 36 [5800 (94%)]    Loss: 0.100387
Train Epoch: 37 [5900 (58%)]    Loss: 0.177173
Train Epoch: 38 [6000 (22%)]    Loss: 0.108121
Train Epoch: 38 [6100 (85%)]    Loss: 0.133032
Train Epoch: 39 [6200 (49%)]    Loss: 0.100478
Train Epoch: 40 [6300 (13%)]    Loss: 0.139267
Train Epoch: 40 [6400 (76%)]    Loss: 0.113654
Train Epoch: 41 [6500 (40%)]    Loss: 0.147110
Train Epoch: 42 [6600 (4%)]     Loss: 0.124898
Train Epoch: 42 [6700 (68%)]    Loss: 0.104651
Train Epoch: 43 [6800 (31%)]    Loss: 0.135332
Train Epoch: 43 [6900 (95%)]    Loss: 0.086103
Train Epoch: 44 [7000 (59%)]    Loss: 0.110926
Train Epoch: 45 [7100 (22%)]    Loss: 0.183165
Train Epoch: 45 [7200 (86%)]    Loss: 0.107485
Train Epoch: 46 [7300 (50%)]    Loss: 0.087114
Train Epoch: 47 [7400 (13%)]    Loss: 0.104271
Train Epoch: 47 [7500 (77%)]    Loss: 0.072625
Train Epoch: 48 [7600 (41%)]    Loss: 0.168238
Train Epoch: 49 [7700 (4%)]     Loss: 0.176388
Train Epoch: 49 [7800 (68%)]    Loss: 0.101117
test map: 0.3799856437694381
```

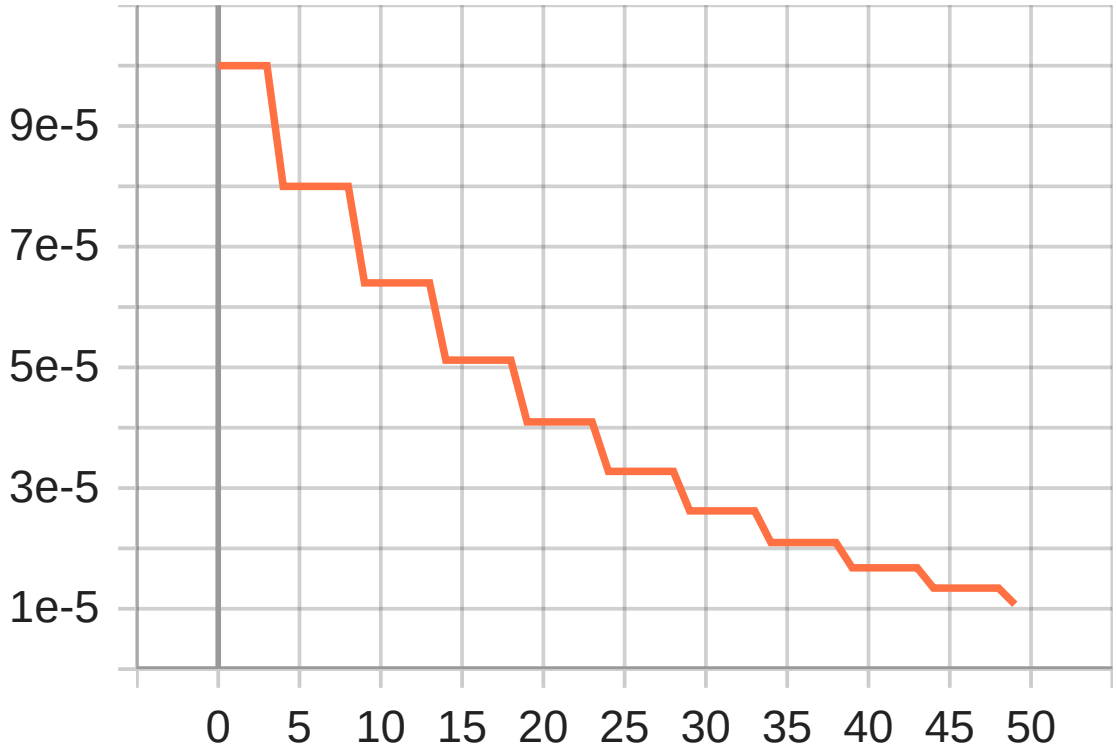
test map vs iter



train loss vs iter


0.5
0.2
0.08
0
1k
2k
3k
4k
5k
6k
7k
8k

lr vs iter


9e-5
7e-5
5e-5
3e-5
1e-5
0
5
10
15
20
25
30
35
40
45
50

In [ ]: