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.

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 = 0.001
    # 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

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, int(cnt), len(train_loader.dataset),
                           100. * batch_idx / len(train_loader), loss.item()))
                train_log['iter'].append(cnt)
                train_log['loss'].append(loss)
                # TODO: calculate your train accuracy!
                pred = output.argmax(dim=1, keepdim=True)
                correct = pred.eq(target.view_as(pred)).sum().item()
                train_acc = correct / data.shape[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`! Check out `simple_cnn.py` and see if you can fix the bug. You may have to restart your ipython kernel for changes to reflect in the notebook. After that's done, be sure to fill in the TODOs in `main`.

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

YOUR ANSWER HERE

The default learning rate 1.0 is too large, which leads to really poor performance. Therefore, I adjust the learning rate to 0.001, while keeping other hyperparameters the same.

In [6]:

```
#### FEEL FREE TO MODIFY args VARIABLE HERE OR ABOVE ####
# args.gamma = float('inf')
```

```
# 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 = 14
args.gamma = 0.7
args.log_every = 100
args.lr = 0.001
args.test_batch_size = 1000
args.val_every = 100


Train Epoch: 0 [0/60000 (0%)]   Loss: 2.340903

Test set: Average loss: 1.9903, Accuracy: 2961/10000 (30%)

Train Epoch: 0 [100/60000 (11%)]        Loss: 0.758049

Test set: Average loss: 0.6521, Accuracy: 7775/10000 (78%)

Train Epoch: 0 [200/60000 (21%)]        Loss: 0.524290

Test set: Average loss: 0.5717, Accuracy: 8009/10000 (80%)

Train Epoch: 0 [300/60000 (32%)]        Loss: 0.736438

Test set: Average loss: 0.5850, Accuracy: 7871/10000 (79%)

Train Epoch: 0 [400/60000 (43%)]        Loss: 0.608861

Test set: Average loss: 0.5354, Accuracy: 8112/10000 (81%)

Train Epoch: 0 [500/60000 (53%)]        Loss: 0.313856

Test set: Average loss: 0.5296, Accuracy: 8100/10000 (81%)

Train Epoch: 0 [600/60000 (64%)]        Loss: 0.413285

Test set: Average loss: 0.5572, Accuracy: 8045/10000 (80%)

Train Epoch: 0 [700/60000 (75%)]        Loss: 0.831430

Test set: Average loss: 0.5582, Accuracy: 8024/10000 (80%)

Train Epoch: 0 [800/60000 (85%)]        Loss: 0.346836

Test set: Average loss: 0.5065, Accuracy: 8211/10000 (82%)

Train Epoch: 0 [900/60000 (96%)]        Loss: 0.445019

Test set: Average loss: 0.5052, Accuracy: 8268/10000 (83%)

Train Epoch: 1 [1000/60000 (7%)]        Loss: 0.476827

Test set: Average loss: 0.4961, Accuracy: 8298/10000 (83%)
```

```
Train Epoch: 1 [1100/60000 (17%)]	Loss: 0.414920

Test set: Average loss: 0.5021, Accuracy: 8196/10000 (82%)

Train Epoch: 1 [1200/60000 (28%)]	Loss: 0.420798

Test set: Average loss: 0.4852, Accuracy: 8274/10000 (83%)

Train Epoch: 1 [1300/60000 (39%)]	Loss: 0.526706

Test set: Average loss: 0.4748, Accuracy: 8336/10000 (83%)

Train Epoch: 1 [1400/60000 (49%)]	Loss: 0.528484

Test set: Average loss: 0.4880, Accuracy: 8286/10000 (83%)

Train Epoch: 1 [1500/60000 (60%)]	Loss: 0.637834

Test set: Average loss: 0.5089, Accuracy: 8207/10000 (82%)

Train Epoch: 1 [1600/60000 (71%)]	Loss: 0.583493

Test set: Average loss: 0.4926, Accuracy: 8271/10000 (83%)

Train Epoch: 1 [1700/60000 (81%)]	Loss: 0.383371

Test set: Average loss: 0.4743, Accuracy: 8322/10000 (83%)

Train Epoch: 1 [1800/60000 (92%)]	Loss: 0.266437

Test set: Average loss: 0.4777, Accuracy: 8321/10000 (83%)

Train Epoch: 2 [1900/60000 (3%)]	Loss: 0.474151

Test set: Average loss: 0.4697, Accuracy: 8346/10000 (83%)

Train Epoch: 2 [2000/60000 (13%)]	Loss: 0.390083

Test set: Average loss: 0.4741, Accuracy: 8335/10000 (83%)

Train Epoch: 2 [2100/60000 (24%)]	Loss: 0.521146

Test set: Average loss: 0.4716, Accuracy: 8334/10000 (83%)

Train Epoch: 2 [2200/60000 (35%)]	Loss: 0.660169

Test set: Average loss: 0.4696, Accuracy: 8359/10000 (84%)

Train Epoch: 2 [2300/60000 (45%)]	Loss: 0.408397

Test set: Average loss: 0.4719, Accuracy: 8315/10000 (83%)

Train Epoch: 2 [2400/60000 (56%)]	Loss: 0.265255

Test set: Average loss: 0.4687, Accuracy: 8373/10000 (84%)

Train Epoch: 2 [2500/60000 (67%)]	Loss: 0.361958

Test set: Average loss: 0.4748, Accuracy: 8330/10000 (83%)

Train Epoch: 2 [2600/60000 (77%)]	Loss: 0.312788

Test set: Average loss: 0.4664, Accuracy: 8337/10000 (83%)

Train Epoch: 2 [2700/60000 (88%)]	Loss: 0.315269
```

```
Test set: Average loss: 0.4669, Accuracy: 8396/10000 (84%)

Train Epoch: 2 [2800/60000 (99%)]       Loss: 0.364829

Test set: Average loss: 0.4650, Accuracy: 8351/10000 (84%)

Train Epoch: 3 [2900/60000 (9%)]        Loss: 0.549056

Test set: Average loss: 0.4641, Accuracy: 8349/10000 (83%)

Train Epoch: 3 [3000/60000 (20%)]       Loss: 0.572824

Test set: Average loss: 0.4615, Accuracy: 8371/10000 (84%)

Train Epoch: 3 [3100/60000 (30%)]       Loss: 0.318392

Test set: Average loss: 0.4577, Accuracy: 8375/10000 (84%)

Train Epoch: 3 [3200/60000 (41%)]       Loss: 0.523411

Test set: Average loss: 0.4632, Accuracy: 8343/10000 (83%)

Train Epoch: 3 [3300/60000 (52%)]       Loss: 0.241377

Test set: Average loss: 0.4577, Accuracy: 8334/10000 (83%)

Train Epoch: 3 [3400/60000 (62%)]       Loss: 0.237253

Test set: Average loss: 0.4594, Accuracy: 8393/10000 (84%)

Train Epoch: 3 [3500/60000 (73%)]       Loss: 0.319804

Test set: Average loss: 0.4649, Accuracy: 8391/10000 (84%)

Train Epoch: 3 [3600/60000 (84%)]       Loss: 0.435473

Test set: Average loss: 0.4648, Accuracy: 8340/10000 (83%)

Train Epoch: 3 [3700/60000 (94%)]       Loss: 0.386885

Test set: Average loss: 0.4521, Accuracy: 8429/10000 (84%)

Train Epoch: 4 [3800/60000 (5%)]        Loss: 0.538492

Test set: Average loss: 0.4497, Accuracy: 8425/10000 (84%)

Train Epoch: 4 [3900/60000 (16%)]       Loss: 0.325613

Test set: Average loss: 0.4525, Accuracy: 8406/10000 (84%)

Train Epoch: 4 [4000/60000 (26%)]       Loss: 0.467162

Test set: Average loss: 0.4493, Accuracy: 8419/10000 (84%)

Train Epoch: 4 [4100/60000 (37%)]       Loss: 0.432622

Test set: Average loss: 0.4506, Accuracy: 8398/10000 (84%)

Train Epoch: 4 [4200/60000 (48%)]       Loss: 0.384308

Test set: Average loss: 0.4536, Accuracy: 8392/10000 (84%)

Train Epoch: 4 [4300/60000 (58%)]       Loss: 0.311331
```

```
Test set: Average loss: 0.4561, Accuracy: 8392/10000 (84%)

Train Epoch: 4 [4400/60000 (69%)]       Loss: 0.398237

Test set: Average loss: 0.4507, Accuracy: 8418/10000 (84%)

Train Epoch: 4 [4500/60000 (80%)]       Loss: 0.379100

Test set: Average loss: 0.4511, Accuracy: 8411/10000 (84%)

Train Epoch: 4 [4600/60000 (90%)]       Loss: 0.402143

Test set: Average loss: 0.4496, Accuracy: 8422/10000 (84%)

Train Epoch: 5 [4700/60000 (1%)]        Loss: 0.511020

Test set: Average loss: 0.4491, Accuracy: 8402/10000 (84%)

Train Epoch: 5 [4800/60000 (12%)]       Loss: 0.495833

Test set: Average loss: 0.4474, Accuracy: 8415/10000 (84%)

Train Epoch: 5 [4900/60000 (22%)]       Loss: 0.772290

Test set: Average loss: 0.4501, Accuracy: 8431/10000 (84%)

Train Epoch: 5 [5000/60000 (33%)]       Loss: 0.478323

Test set: Average loss: 0.4497, Accuracy: 8417/10000 (84%)

Train Epoch: 5 [5100/60000 (44%)]       Loss: 0.516549

Test set: Average loss: 0.4454, Accuracy: 8416/10000 (84%)

Train Epoch: 5 [5200/60000 (54%)]       Loss: 0.323361

Test set: Average loss: 0.4437, Accuracy: 8442/10000 (84%)

Train Epoch: 5 [5300/60000 (65%)]       Loss: 0.396120

Test set: Average loss: 0.4475, Accuracy: 8429/10000 (84%)

Train Epoch: 5 [5400/60000 (76%)]       Loss: 0.575883

Test set: Average loss: 0.4449, Accuracy: 8441/10000 (84%)

Train Epoch: 5 [5500/60000 (86%)]       Loss: 0.359784

Test set: Average loss: 0.4464, Accuracy: 8430/10000 (84%)

Train Epoch: 5 [5600/60000 (97%)]       Loss: 0.219373

Test set: Average loss: 0.4547, Accuracy: 8373/10000 (84%)

Train Epoch: 6 [5700/60000 (8%)]        Loss: 0.426067

Test set: Average loss: 0.4466, Accuracy: 8423/10000 (84%)

Train Epoch: 6 [5800/60000 (18%)]       Loss: 0.261763

Test set: Average loss: 0.4468, Accuracy: 8428/10000 (84%)

Train Epoch: 6 [5900/60000 (29%)]       Loss: 0.344103

Test set: Average loss: 0.4446, Accuracy: 8440/10000 (84%)
```

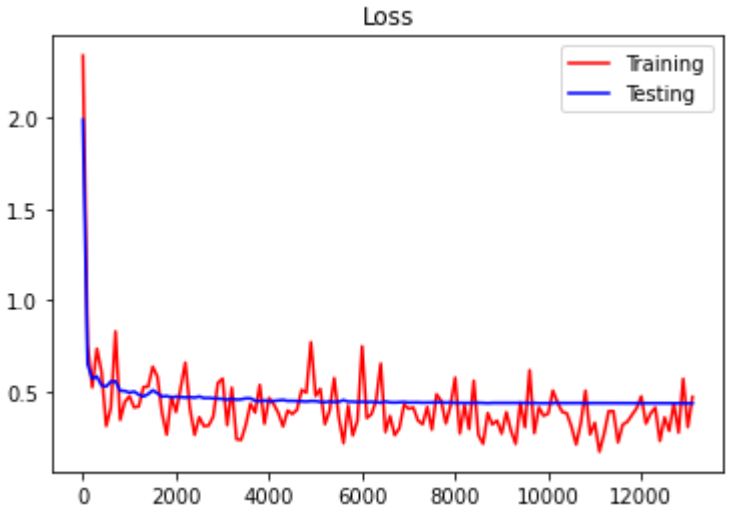
```
Train Epoch: 6 [6000/60000 (40%)]	Loss: 0.750165

Test set: Average loss: 0.4462, Accuracy: 8433/10000 (84%)

Train Epoch: 6 [6100/60000 (50%)]	Loss: 0.357264

Test set: Average loss: 0.4439, Accuracy: 8441/10000 (84%)

Train Epoch: 6 [6200/60000 (61%)]	Loss: 0.377430

Test set: Average loss: 0.4458, Accuracy: 8440/10000 (84%)

Train Epoch: 6 [6300/60000 (72%)]	Loss: 0.451706

Test set: Average loss: 0.4435, Accuracy: 8424/10000 (84%)

Train Epoch: 6 [6400/60000 (82%)]	Loss: 0.655201

Test set: Average loss: 0.4410, Accuracy: 8440/10000 (84%)

Train Epoch: 6 [6500/60000 (93%)]	Loss: 0.280491

Test set: Average loss: 0.4473, Accuracy: 8424/10000 (84%)

Train Epoch: 7 [6600/60000 (4%)]	Loss: 0.368149

Test set: Average loss: 0.4428, Accuracy: 8445/10000 (84%)

Train Epoch: 7 [6700/60000 (14%)]	Loss: 0.263288

Test set: Average loss: 0.4419, Accuracy: 8434/10000 (84%)

Train Epoch: 7 [6800/60000 (25%)]	Loss: 0.300995

Test set: Average loss: 0.4434, Accuracy: 8428/10000 (84%)

Train Epoch: 7 [6900/60000 (36%)]	Loss: 0.429817

Test set: Average loss: 0.4447, Accuracy: 8430/10000 (84%)

Train Epoch: 7 [7000/60000 (46%)]	Loss: 0.407572

Test set: Average loss: 0.4419, Accuracy: 8433/10000 (84%)

Train Epoch: 7 [7100/60000 (57%)]	Loss: 0.414460

Test set: Average loss: 0.4422, Accuracy: 8440/10000 (84%)

Train Epoch: 7 [7200/60000 (68%)]	Loss: 0.343657

Test set: Average loss: 0.4426, Accuracy: 8432/10000 (84%)

Train Epoch: 7 [7300/60000 (78%)]	Loss: 0.322952

Test set: Average loss: 0.4411, Accuracy: 8428/10000 (84%)

Train Epoch: 7 [7400/60000 (89%)]	Loss: 0.417411

Test set: Average loss: 0.4418, Accuracy: 8443/10000 (84%)

Train Epoch: 7 [7500/60000 (100%)]	Loss: 0.293951

Test set: Average loss: 0.4418, Accuracy: 8425/10000 (84%)
```

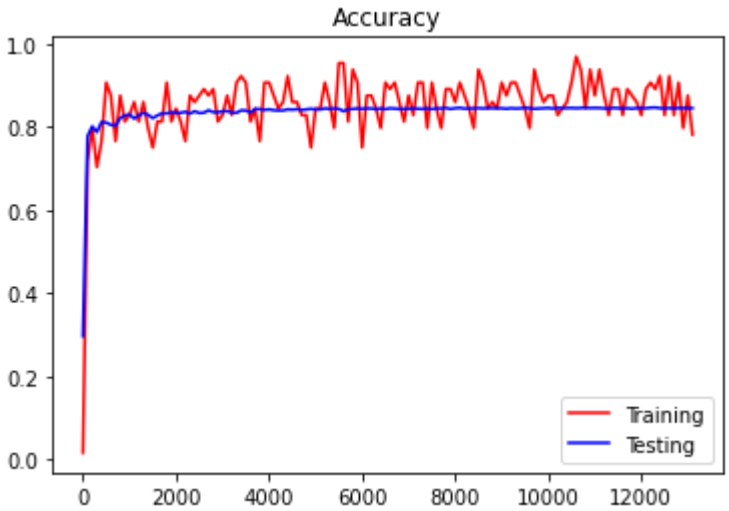
```
Train Epoch: 8 [7600/60000 (10%)]	Loss: 0.487157

Test set: Average loss: 0.4420, Accuracy: 8442/10000 (84%)

Train Epoch: 8 [7700/60000 (21%)]	Loss: 0.450039

Test set: Average loss: 0.4430, Accuracy: 8441/10000 (84%)

Train Epoch: 8 [7800/60000 (32%)]	Loss: 0.327558

Test set: Average loss: 0.4412, Accuracy: 8454/10000 (85%)

Train Epoch: 8 [7900/60000 (42%)]	Loss: 0.424175

Test set: Average loss: 0.4421, Accuracy: 8427/10000 (84%)

Train Epoch: 8 [8000/60000 (53%)]	Loss: 0.578022

Test set: Average loss: 0.4400, Accuracy: 8447/10000 (84%)

Train Epoch: 8 [8100/60000 (64%)]	Loss: 0.272798

Test set: Average loss: 0.4398, Accuracy: 8454/10000 (85%)

Train Epoch: 8 [8200/60000 (74%)]	Loss: 0.427282

Test set: Average loss: 0.4398, Accuracy: 8437/10000 (84%)

Train Epoch: 8 [8300/60000 (85%)]	Loss: 0.296832

Test set: Average loss: 0.4395, Accuracy: 8440/10000 (84%)

Train Epoch: 8 [8400/60000 (96%)]	Loss: 0.560962

Test set: Average loss: 0.4406, Accuracy: 8444/10000 (84%)

Train Epoch: 9 [8500/60000 (6%)]	Loss: 0.264197

Test set: Average loss: 0.4407, Accuracy: 8448/10000 (84%)

Train Epoch: 9 [8600/60000 (17%)]	Loss: 0.216328

Test set: Average loss: 0.4395, Accuracy: 8441/10000 (84%)

Train Epoch: 9 [8700/60000 (28%)]	Loss: 0.384126

Test set: Average loss: 0.4382, Accuracy: 8449/10000 (84%)

Train Epoch: 9 [8800/60000 (38%)]	Loss: 0.322103

Test set: Average loss: 0.4397, Accuracy: 8447/10000 (84%)

Train Epoch: 9 [8900/60000 (49%)]	Loss: 0.342752

Test set: Average loss: 0.4397, Accuracy: 8446/10000 (84%)

Train Epoch: 9 [9000/60000 (59%)]	Loss: 0.271790

Test set: Average loss: 0.4388, Accuracy: 8444/10000 (84%)

Train Epoch: 9 [9100/60000 (70%)]	Loss: 0.388907

Test set: Average loss: 0.4402, Accuracy: 8433/10000 (84%)

Train Epoch: 9 [9200/60000 (81%)]	Loss: 0.287224
```

```
Test set: Average loss: 0.4396, Accuracy: 8446/10000 (84%)

Train Epoch: 9 [9300/60000 (91%)]	Loss: 0.215781

Test set: Average loss: 0.4399, Accuracy: 8439/10000 (84%)

Train Epoch: 10 [9400/60000 (2%)]	Loss: 0.442570

Test set: Average loss: 0.4392, Accuracy: 8442/10000 (84%)

Train Epoch: 10 [9500/60000 (13%)]	Loss: 0.306275

Test set: Average loss: 0.4397, Accuracy: 8458/10000 (85%)

Train Epoch: 10 [9600/60000 (23%)]	Loss: 0.619452

Test set: Average loss: 0.4390, Accuracy: 8444/10000 (84%)

Train Epoch: 10 [9700/60000 (34%)]	Loss: 0.274386

Test set: Average loss: 0.4396, Accuracy: 8433/10000 (84%)

Train Epoch: 10 [9800/60000 (45%)]	Loss: 0.415355

Test set: Average loss: 0.4382, Accuracy: 8441/10000 (84%)

Train Epoch: 10 [9900/60000 (55%)]	Loss: 0.369623

Test set: Average loss: 0.4384, Accuracy: 8449/10000 (84%)

Train Epoch: 10 [10000/60000 (66%)]	Loss: 0.381140

Test set: Average loss: 0.4396, Accuracy: 8452/10000 (85%)

Train Epoch: 10 [10100/60000 (77%)]	Loss: 0.507195

Test set: Average loss: 0.4388, Accuracy: 8438/10000 (84%)

Train Epoch: 10 [10200/60000 (87%)]	Loss: 0.440882

Test set: Average loss: 0.4405, Accuracy: 8454/10000 (85%)

Train Epoch: 10 [10300/60000 (98%)]	Loss: 0.390745

Test set: Average loss: 0.4389, Accuracy: 8447/10000 (84%)

Train Epoch: 11 [10400/60000 (9%)]	Loss: 0.382395

Test set: Average loss: 0.4386, Accuracy: 8449/10000 (84%)

Train Epoch: 11 [10500/60000 (19%)]	Loss: 0.308836

Test set: Average loss: 0.4384, Accuracy: 8452/10000 (85%)

Train Epoch: 11 [10600/60000 (30%)]	Loss: 0.212496

Test set: Average loss: 0.4391, Accuracy: 8446/10000 (84%)

Train Epoch: 11 [10700/60000 (41%)]	Loss: 0.328970

Test set: Average loss: 0.4393, Accuracy: 8451/10000 (85%)

Train Epoch: 11 [10800/60000 (51%)]	Loss: 0.506715
```

```
Test set: Average loss: 0.4383, Accuracy: 8462/10000 (85%)

Train Epoch: 11 [10900/60000 (62%)]     Loss: 0.267738

Test set: Average loss: 0.4391, Accuracy: 8449/10000 (84%)

Train Epoch: 11 [11000/60000 (73%)]     Loss: 0.330896

Test set: Average loss: 0.4388, Accuracy: 8453/10000 (85%)

Train Epoch: 11 [11100/60000 (83%)]     Loss: 0.172803

Test set: Average loss: 0.4388, Accuracy: 8451/10000 (85%)

Train Epoch: 11 [11200/60000 (94%)]     Loss: 0.270212

Test set: Average loss: 0.4382, Accuracy: 8443/10000 (84%)

Train Epoch: 12 [11300/60000 (5%)]      Loss: 0.394662

Test set: Average loss: 0.4380, Accuracy: 8436/10000 (84%)

Train Epoch: 12 [11400/60000 (15%)]     Loss: 0.394579

Test set: Average loss: 0.4380, Accuracy: 8453/10000 (85%)

Train Epoch: 12 [11500/60000 (26%)]     Loss: 0.224089

Test set: Average loss: 0.4384, Accuracy: 8446/10000 (84%)

Train Epoch: 12 [11600/60000 (37%)]     Loss: 0.320641

Test set: Average loss: 0.4392, Accuracy: 8440/10000 (84%)

Train Epoch: 12 [11700/60000 (47%)]     Loss: 0.336827

Test set: Average loss: 0.4386, Accuracy: 8446/10000 (84%)

Train Epoch: 12 [11800/60000 (58%)]     Loss: 0.372964

Test set: Average loss: 0.4383, Accuracy: 8435/10000 (84%)

Train Epoch: 12 [11900/60000 (69%)]     Loss: 0.411211

Test set: Average loss: 0.4378, Accuracy: 8448/10000 (84%)

Train Epoch: 12 [12000/60000 (79%)]     Loss: 0.475940

Test set: Average loss: 0.4387, Accuracy: 8452/10000 (85%)

Train Epoch: 12 [12100/60000 (90%)]     Loss: 0.326505

Test set: Average loss: 0.4381, Accuracy: 8445/10000 (84%)

Train Epoch: 13 [12200/60000 (1%)]      Loss: 0.381781

Test set: Average loss: 0.4375, Accuracy: 8458/10000 (85%)

Train Epoch: 13 [12300/60000 (11%)]     Loss: 0.414065

Test set: Average loss: 0.4377, Accuracy: 8464/10000 (85%)

Train Epoch: 13 [12400/60000 (22%)]     Loss: 0.231819

Test set: Average loss: 0.4378, Accuracy: 8446/10000 (84%)
```

```
Train Epoch: 13 [12500/60000 (33%)]     Loss: 0.360525

Test set: Average loss: 0.4377, Accuracy: 8444/10000 (84%)

Train Epoch: 13 [12600/60000 (43%)]     Loss: 0.286705

Test set: Average loss: 0.4378, Accuracy: 8449/10000 (84%)

Train Epoch: 13 [12700/60000 (54%)]     Loss: 0.438446

Test set: Average loss: 0.4377, Accuracy: 8443/10000 (84%)

Train Epoch: 13 [12800/60000 (65%)]     Loss: 0.277531

Test set: Average loss: 0.4377, Accuracy: 8451/10000 (85%)

Train Epoch: 13 [12900/60000 (75%)]     Loss: 0.572079

Test set: Average loss: 0.4378, Accuracy: 8449/10000 (84%)

Train Epoch: 13 [13000/60000 (86%)]     Loss: 0.307816

Test set: Average loss: 0.4378, Accuracy: 8442/10000 (84%)

Train Epoch: 13 [13100/60000 (97%)]     Loss: 0.472870

Test set: Average loss: 0.4378, Accuracy: 8448/10000 (84%)
```





Play with parameters.(3pt)

How many trainable parameters does the trained model have?

In [7]:
```
def param_count(model):
    total_num_param = 0
    for param in model.parameters():
        num_param = 1
        for dim in param.shape:
            num_param *= dim
        total_num_param += num_param
    return total_num_param

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

```
Model has 454922 params
```

Deep Linear Networks?!? (5pt)

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

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

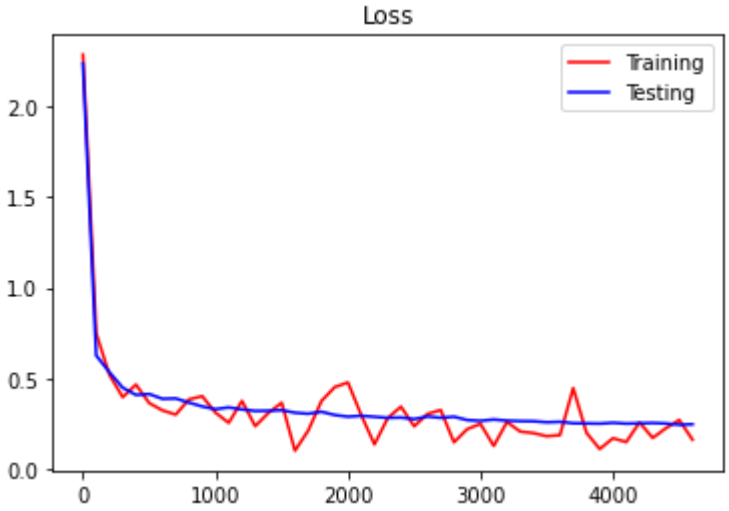
In [4]:
```
args.epochs = 5
main()
```

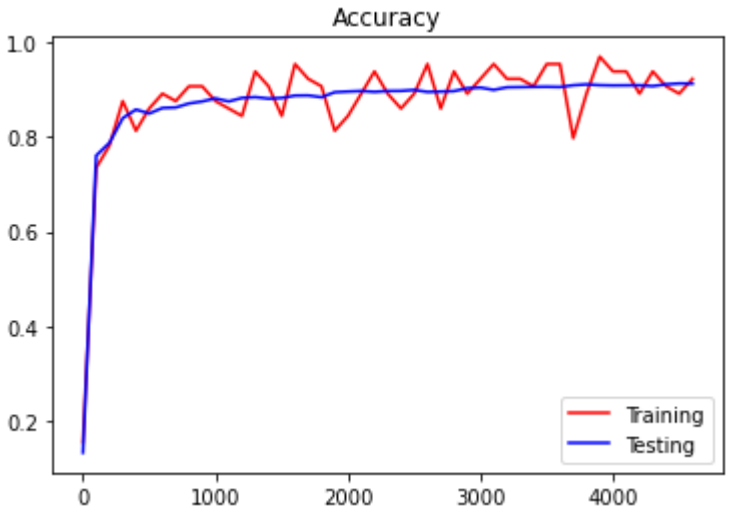
```
Train Epoch: 0 [0/60000 (0%)]   Loss: 2.288714

Test set: Average loss: 2.2396, Accuracy: 1340/10000 (13%)

Train Epoch: 0 [100/60000 (11%)]        Loss: 0.747727

Test set: Average loss: 0.6254, Accuracy: 7600/10000 (76%)

Train Epoch: 0 [200/60000 (21%)]        Loss: 0.518000

Test set: Average loss: 0.5323, Accuracy: 7870/10000 (79%)

Train Epoch: 0 [300/60000 (32%)]        Loss: 0.394593

Test set: Average loss: 0.4469, Accuracy: 8390/10000 (84%)

Train Epoch: 0 [400/60000 (43%)]        Loss: 0.464086

Test set: Average loss: 0.4070, Accuracy: 8572/10000 (86%)

Train Epoch: 0 [500/60000 (53%)]        Loss: 0.361940

Test set: Average loss: 0.4119, Accuracy: 8488/10000 (85%)

Train Epoch: 0 [600/60000 (64%)]        Loss: 0.321545

Test set: Average loss: 0.3854, Accuracy: 8605/10000 (86%)

Train Epoch: 0 [700/60000 (75%)]        Loss: 0.298111
```

```
Test set: Average loss: 0.3873, Accuracy: 8615/10000 (86%)

Train Epoch: 0 [800/60000 (85%)]	Loss: 0.382229

Test set: Average loss: 0.3647, Accuracy: 8700/10000 (87%)

Train Epoch: 0 [900/60000 (96%)]	Loss: 0.399932

Test set: Average loss: 0.3432, Accuracy: 8743/10000 (87%)

Train Epoch: 1 [1000/60000 (7%)]	Loss: 0.309426

Test set: Average loss: 0.3281, Accuracy: 8807/10000 (88%)

Train Epoch: 1 [1100/60000 (17%)]	Loss: 0.252525

Test set: Average loss: 0.3380, Accuracy: 8743/10000 (87%)

Train Epoch: 1 [1200/60000 (28%)]	Loss: 0.373907

Test set: Average loss: 0.3268, Accuracy: 8817/10000 (88%)

Train Epoch: 1 [1300/60000 (39%)]	Loss: 0.234380

Test set: Average loss: 0.3185, Accuracy: 8834/10000 (88%)

Train Epoch: 1 [1400/60000 (49%)]	Loss: 0.307808

Test set: Average loss: 0.3194, Accuracy: 8803/10000 (88%)

Train Epoch: 1 [1500/60000 (60%)]	Loss: 0.363554

Test set: Average loss: 0.3225, Accuracy: 8816/10000 (88%)

Train Epoch: 1 [1600/60000 (71%)]	Loss: 0.098626

Test set: Average loss: 0.3086, Accuracy: 8866/10000 (89%)

Train Epoch: 1 [1700/60000 (81%)]	Loss: 0.209873

Test set: Average loss: 0.3030, Accuracy: 8869/10000 (89%)

Train Epoch: 1 [1800/60000 (92%)]	Loss: 0.373239

Test set: Average loss: 0.3139, Accuracy: 8832/10000 (88%)

Train Epoch: 2 [1900/60000 (3%)]	Loss: 0.450348

Test set: Average loss: 0.2959, Accuracy: 8937/10000 (89%)

Train Epoch: 2 [2000/60000 (13%)]	Loss: 0.475501

Test set: Average loss: 0.2868, Accuracy: 8951/10000 (90%)

Train Epoch: 2 [2100/60000 (24%)]	Loss: 0.295849

Test set: Average loss: 0.2915, Accuracy: 8963/10000 (90%)

Train Epoch: 2 [2200/60000 (35%)]	Loss: 0.133682

Test set: Average loss: 0.2860, Accuracy: 8944/10000 (89%)

Train Epoch: 2 [2300/60000 (45%)]	Loss: 0.278902

Test set: Average loss: 0.2802, Accuracy: 8963/10000 (90%)
```

```
Train Epoch: 2 [2400/60000 (56%)]	Loss: 0.342095

Test set: Average loss: 0.2821, Accuracy: 8965/10000 (90%)

Train Epoch: 2 [2500/60000 (67%)]	Loss: 0.233407

Test set: Average loss: 0.2733, Accuracy: 8985/10000 (90%)

Train Epoch: 2 [2600/60000 (77%)]	Loss: 0.301706

Test set: Average loss: 0.2877, Accuracy: 8943/10000 (89%)

Train Epoch: 2 [2700/60000 (88%)]	Loss: 0.323473

Test set: Average loss: 0.2805, Accuracy: 8951/10000 (90%)

Train Epoch: 2 [2800/60000 (99%)]	Loss: 0.145058

Test set: Average loss: 0.2864, Accuracy: 8960/10000 (90%)

Train Epoch: 3 [2900/60000 (9%)]	Loss: 0.217979

Test set: Average loss: 0.2680, Accuracy: 9021/10000 (90%)

Train Epoch: 3 [3000/60000 (20%)]	Loss: 0.247084

Test set: Average loss: 0.2637, Accuracy: 9035/10000 (90%)

Train Epoch: 3 [3100/60000 (30%)]	Loss: 0.125018

Test set: Average loss: 0.2710, Accuracy: 8986/10000 (90%)

Train Epoch: 3 [3200/60000 (41%)]	Loss: 0.257429

Test set: Average loss: 0.2646, Accuracy: 9041/10000 (90%)

Train Epoch: 3 [3300/60000 (52%)]	Loss: 0.204306

Test set: Average loss: 0.2624, Accuracy: 9045/10000 (90%)

Train Epoch: 3 [3400/60000 (62%)]	Loss: 0.195959

Test set: Average loss: 0.2615, Accuracy: 9050/10000 (90%)

Train Epoch: 3 [3500/60000 (73%)]	Loss: 0.178723

Test set: Average loss: 0.2558, Accuracy: 9052/10000 (91%)

Train Epoch: 3 [3600/60000 (84%)]	Loss: 0.184792

Test set: Average loss: 0.2583, Accuracy: 9047/10000 (90%)

Train Epoch: 3 [3700/60000 (94%)]	Loss: 0.444782

Test set: Average loss: 0.2503, Accuracy: 9084/10000 (91%)

Train Epoch: 4 [3800/60000 (5%)]	Loss: 0.197771

Test set: Average loss: 0.2495, Accuracy: 9102/10000 (91%)

Train Epoch: 4 [3900/60000 (16%)]	Loss: 0.107926

Test set: Average loss: 0.2483, Accuracy: 9086/10000 (91%)
```

```
Train Epoch: 4 [4000/60000 (26%)]       Loss: 0.167688

Test set: Average loss: 0.2529, Accuracy: 9080/10000 (91%)

Train Epoch: 4 [4100/60000 (37%)]       Loss: 0.146352

Test set: Average loss: 0.2485, Accuracy: 9082/10000 (91%)

Train Epoch: 4 [4200/60000 (48%)]       Loss: 0.255102

Test set: Average loss: 0.2500, Accuracy: 9084/10000 (91%)

Train Epoch: 4 [4300/60000 (58%)]       Loss: 0.169296

Test set: Average loss: 0.2520, Accuracy: 9066/10000 (91%)

Train Epoch: 4 [4400/60000 (69%)]       Loss: 0.223094

Test set: Average loss: 0.2492, Accuracy: 9103/10000 (91%)

Train Epoch: 4 [4500/60000 (80%)]       Loss: 0.268666

Test set: Average loss: 0.2420, Accuracy: 9122/10000 (91%)

Train Epoch: 4 [4600/60000 (90%)]       Loss: 0.159400

Test set: Average loss: 0.2447, Accuracy: 9118/10000 (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))
  (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 applied ReLU after the first and second convolution layers but before the pooling layers. I also applied ReLU after the first fully-connected layer.

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

YOUR ANSWER HERE

The data could be separated with linear decision boundaries

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. Complete the functions `preload_anno` and `__getitem__` in `voc_dataset.py` .

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

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

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

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

DESCRIBE YOUR AUGMENTATION PIPELINE HERE**

Train Augmentations:

For train augmentations, I first use RandomResizedCrop to randomly scale the image and crop it, and then resize it to our desired size. Then, I filped the image randomly. Then, I transformed the image to tensor. Finally, I used the mean and std information from ImageNet to normalize the image tensor.

Test Augmentations:

For test augmentations, I first resized the image width and height to 1.05 times of our desired size. Then, I used center crop to crop out the center part of the image of our desired size. Then, I transformed the image to tensor. Finally, I used the mean and std information from ImageNet to normalize the image tensor.

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**

One way to compute AP is to first compute the precision and recall for each class. Then, for each class, we can plot the precision recall curve. Finally, we can compute the area under the precision recall curve as AP.

1.4 Let's Start Training! (5 pts)

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

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

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

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

```
args.batch_size = 64
args.device = cuda
args.epochs = 5
args.gamma = 0.7
args.log_every = 100
args.lr = 1.0
args.save_at_end = False
args.save_freq = -1
args.test_batch_size = 1000
args.val_every = 100
```

In [2]:

```
# initializes (your) naiive model
model = SimpleCNN(num_classes=len(VOCDataset.CLASS_NAMES), inp_size=64, c_dim=3)
# initializes Adam optimizer and simple StepLR scheduler
optimizer = torch.optim.Adam(model.parameters(), lr=args.lr)
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=1, gamma=args.g
# 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: 0.698940
test map (validation): 0.06304451297885003
Train Epoch: 1 [100 (27%)]      Loss: 0.241858
test map (validation): 0.06304451297885003
Train Epoch: 2 [200 (53%)]      Loss: 0.254361
test map (validation): 0.06304451297885003
Train Epoch: 3 [300 (80%)]      Loss: 0.247762
test map (validation): 0.06304451297885003
test map: 0.06304451297885003
```

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

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

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

In [3]:

```
args = ARGS(epochs=5, batch_size=20, lr=0.001, use_cuda=True, gamma=0.7)
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=2, gamma=args.g
test_ap, test_map = trainer.train(args, model, optimizer, scheduler)
print('test map:', test_map)
```

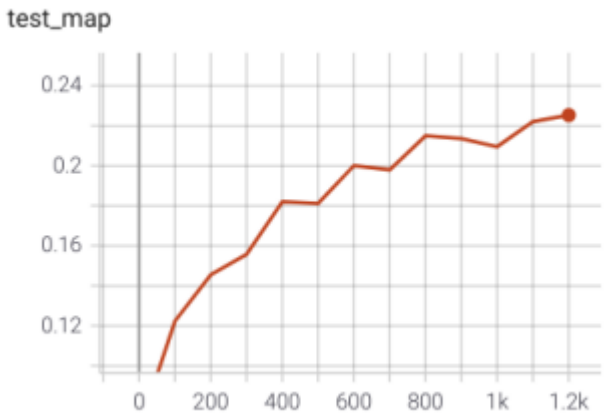
```
Train Epoch: 0 [0 (0%)] Loss: 0.688126
test map (validation): 0.06848793717607542
Train Epoch: 0 [100 (40%)]      Loss: 0.243655
test map (validation): 0.12235175558534847
```

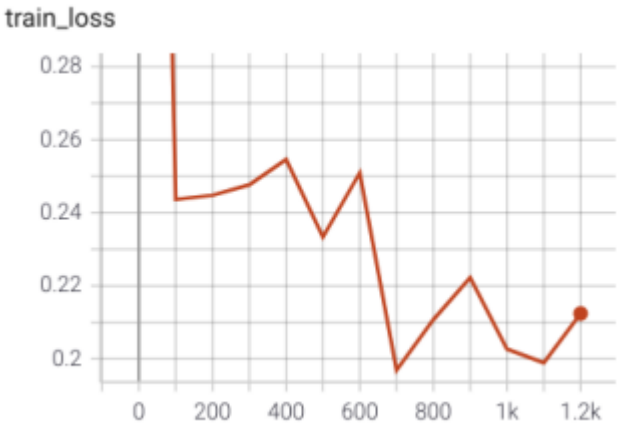
```
Train Epoch: 0 [200 (80%)]      Loss: 0.244782
test map (validation): 0.14538727048111316
Train Epoch: 1 [300 (20%)]      Loss: 0.247650
test map (validation): 0.15570737815700825
Train Epoch: 1 [400 (59%)]      Loss: 0.254594
test map (validation): 0.1819426870387927
Train Epoch: 1 [500 (99%)]      Loss: 0.233357
test map (validation): 0.1811027064544803
Train Epoch: 2 [600 (39%)]      Loss: 0.250921
test map (validation): 0.2000185282989106
Train Epoch: 2 [700 (79%)]      Loss: 0.196884
test map (validation): 0.19796246650691995
Train Epoch: 3 [800 (19%)]      Loss: 0.210737
test map (validation): 0.2149599569886854
Train Epoch: 3 [900 (59%)]      Loss: 0.222247
test map (validation): 0.21353854036398764
Train Epoch: 3 [1000 (98%)]     Loss: 0.202654
test map (validation): 0.20947277073477472
Train Epoch: 4 [1100 (38%)]     Loss: 0.198952
test map (validation): 0.2219808031021973
Train Epoch: 4 [1200 (78%)]     Loss: 0.212426
test map (validation): 0.22523818306601245
test map: 0.23750843674416466
```

INSERT YOUR TENSORBOARD SCREENSHOTS HERE

In [4]:

```
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
img = mpimg.imread('q1_1.png')
plt.imshow(img)
plt.axis("off")
plt.show()
img = mpimg.imread('q1_2.png')
plt.imshow(img)
plt.axis("off")
plt.show()
```





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 [2]:
```
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):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 96, kernel_size=11, stride=4, padding=0)
        self.conv2 = nn.Conv2d(96, 256, kernel_size=5, stride=1, padding=2)
        self.conv3 = nn.Conv2d(256, 384, kernel_size=3, stride=1, padding=1)
        self.conv4 = nn.Conv2d(384, 384, kernel_size=3, stride=1, padding=1)
        self.conv5 = nn.Conv2d(384, 256, kernel_size=3, stride=1, padding=1)
        self.relu = nn.ReLU()
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2)
        self.fc1 = nn.Linear(256 * 5 * 5, 4096)
        self.fc2 = nn.Linear(4096, 4096)
        self.fc3 = nn.Linear(4096, 20)
        self.dropout = nn.Dropout(p=0.5)

    def forward(self, x):
        x = self.conv1(x)
        x = self.relu(x)
        x = self.maxpool(x)
        x = self.conv2(x)
        x = self.relu(x)
        x = self.maxpool(x)
        x = self.conv3(x)
        x = self.relu(x)
        x = self.conv4(x)
        x = self.relu(x)
        x = self.conv5(x)
        x = self.relu(x)
        x = self.maxpool(x)

        flat_x = x.view(-1, 256 * 5 * 5)
        out = self.fc1(flat_x)
        out = self.relu(out)
        out = self.dropout(out)
        out = self.fc2(out)
        out = self.relu(out)
        out = self.dropout(out)
        out = self.fc3(out)
        return out
```

2.2 Save the Model (5 pts)

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

2.3 Train and Test (5pts)

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

- batch_size=32
- Adam optimizer with lr=0.0001

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

In [3]:
```
args = ARGS(epochs=50, batch_size=32, lr=0.0001,
            use_cuda=True, gamma=0.9, log_every=250,
            val_every=250,
            save_at_end=True, save_freq=10)

model = CaffeNet()
optimizer = torch.optim.Adam(model.parameters(), lr=args.lr)
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=20, gamma=args.
test_ap, test_map = trainer.train(args, model, optimizer, scheduler, model_name=
print('test map:', test_map)
```

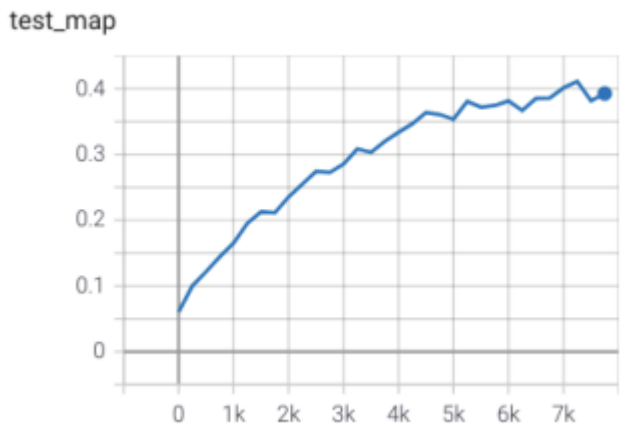
```
Train Epoch: 0 [0 (0%)] Loss: 0.694476
test map (validation): 0.06026704430732975
Train Epoch: 1 [250 (59%)]      Loss: 0.242136
test map (validation): 0.10017538170986677
Train Epoch: 3 [500 (18%)]      Loss: 0.234352
test map (validation): 0.12173328525411864
Train Epoch: 4 [750 (78%)]      Loss: 0.212129
test map (validation): 0.14463678261710222
Train Epoch: 6 [1000 (37%)]     Loss: 0.232532
test map (validation): 0.1651594803069962
Train Epoch: 7 [1250 (96%)]     Loss: 0.196818
test map (validation): 0.1953057074402645
Train Epoch: 9 [1500 (55%)]     Loss: 0.248760
test map (validation): 0.21267779149708468
Train Epoch: 11 [1750 (15%)]    Loss: 0.224964
test map (validation): 0.21123213918901224
Train Epoch: 12 [2000 (74%)]    Loss: 0.222756
test map (validation): 0.23541097185089468
Train Epoch: 14 [2250 (33%)]    Loss: 0.226058
test map (validation): 0.25486454152797117
Train Epoch: 15 [2500 (92%)]    Loss: 0.186564
test map (validation): 0.274279819867863
Train Epoch: 17 [2750 (52%)]    Loss: 0.155841
test map (validation): 0.2724542444365362
Train Epoch: 19 [3000 (11%)]    Loss: 0.198058
test map (validation): 0.2853462146121153
Train Epoch: 20 [3250 (70%)]    Loss: 0.178976
test map (validation): 0.30854936508317066
Train Epoch: 22 [3500 (29%)]    Loss: 0.203448
test map (validation): 0.30316554740376006
Train Epoch: 23 [3750 (89%)]    Loss: 0.195886
test map (validation): 0.3199196309063531
Train Epoch: 25 [4000 (48%)]    Loss: 0.194075
test map (validation): 0.33366805999028515
Train Epoch: 27 [4250 (7%)]     Loss: 0.143384
test map (validation): 0.3466667788712888
Train Epoch: 28 [4500 (66%)]    Loss: 0.186671
test map (validation): 0.3637284732819059
Train Epoch: 30 [4750 (25%)]    Loss: 0.162803
```

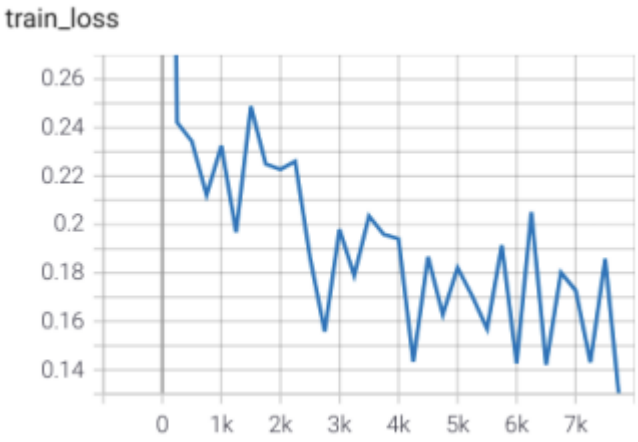
```
test map (validation): 0.3606202619299141
Train Epoch: 31 [5000 (85%)]    Loss: 0.182192
test map (validation): 0.3534624680574524
Train Epoch: 33 [5250 (44%)]    Loss: 0.170378
test map (validation): 0.38066717795304406
Train Epoch: 35 [5500 (3%)]     Loss: 0.156932
test map (validation): 0.3718773436524309
Train Epoch: 36 [5750 (62%)]    Loss: 0.191462
test map (validation): 0.37430348077314524
Train Epoch: 38 [6000 (22%)]    Loss: 0.142728
test map (validation): 0.3813758615387384
Train Epoch: 39 [6250 (81%)]    Loss: 0.205107
test map (validation): 0.3666792545891184
Train Epoch: 41 [6500 (40%)]    Loss: 0.142090
test map (validation): 0.38530953942992185
Train Epoch: 42 [6750 (99%)]    Loss: 0.180185
test map (validation): 0.38590299825467456
Train Epoch: 44 [7000 (59%)]    Loss: 0.172812
test map (validation): 0.40113172222384713
Train Epoch: 46 [7250 (18%)]    Loss: 0.143145
test map (validation): 0.41082291717067154
Train Epoch: 47 [7500 (77%)]    Loss: 0.185927
test map (validation): 0.3817210956129654
Train Epoch: 49 [7750 (36%)]    Loss: 0.125048
test map (validation): 0.3926093846828709
test map: 0.4000417992448294
```

INSERT YOUR TENSORBOARD SCREENSHOTS HERE

In [4]:

```
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
img = mpimg.imread('q2_1.png')
plt.imshow(img)
plt.axis("off")
plt.show()
img = mpimg.imread('q2_2.png')
plt.imshow(img)
plt.axis("off")
plt.show()
```





2.4 Visualizing: Conv-1 filters (5pts)

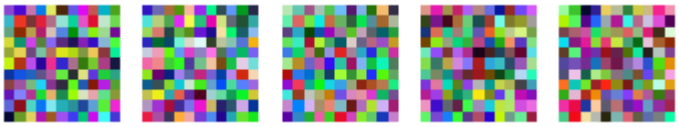
Extract and compare the conv1 filters, at different stages of the training (at least from 3 different iterations). Show at least 5 filters.

In [ ]:
```
# visualize below
```

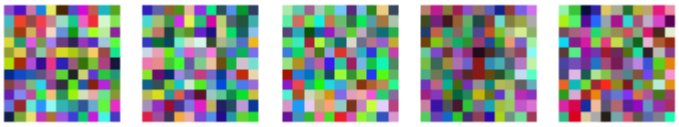
In [9]:
```
import numpy as np

model1, model2, model3 = CaffeNet(), CaffeNet(), CaffeNet()
model1.load_state_dict(torch.load("models/q2_0"))
model2.load_state_dict(torch.load("models/q2_30"))
model3.load_state_dict(torch.load("models/q2_50"))
model1.eval()
model2.eval()
model3.eval()

print("0 epoch")
weights1 = model1.conv1.weight.data.numpy()
fig1, ax1 = plt.subplots(1, 5)
for i in range(5):
    feature_map = np.moveaxis(weights1[i], 0, -1)
    feature_map = (feature_map - feature_map.min()) / (feature_map.max() - featu
    ax1[i].axis("off")
    ax1[i].imshow(feature_map)
plt.show()

print("30 epoch")
weights2 = model2.conv1.weight.data.numpy()
fig2, ax2 = plt.subplots(1, 5)
for i in range(5):
    feature_map = np.moveaxis(weights2[i], 0, -1)
    feature_map = (feature_map - feature_map.min()) / (feature_map.max() - featu
    ax2[i].axis("off")
    ax2[i].imshow(feature_map)
plt.show()

print("50 epoch")
weights3 = model3.conv1.weight.data.numpy()
fig3, ax3 = plt.subplots(1, 5)
for i in range(5):
```

```
    feature_map = np.moveaxis(weights3[i], 0, -1)
    feature_map = (feature_map - feature_map.min()) / (feature_map.max() - featu
    ax3[i].axis("off")
    ax3[i].imshow(feature_map)
plt.show()
```

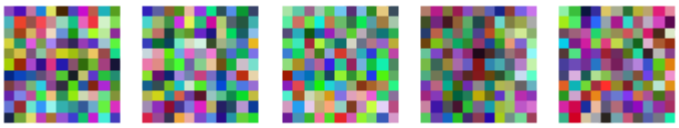
0 epoch



30 epoch



50 epoch



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!

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(pretrained=False)
num_features = ResNet.fc.in_features
ResNet.fc = nn.Linear(num_features, 20)
```

3.2 Add Tensorboard Summaries (6 pts)

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

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

3.3 Train and Test (8 pts)

Use the same hyperparameter settings from Task 2, and train the model for 50 epochs. Report tensorboard screenshots for *all* of the summaries listed above (for image summaries show

screenshots at $n \geq 3$ iterations)

REMEMBER TO SAVE A MODEL AT THE END OF TRAINING

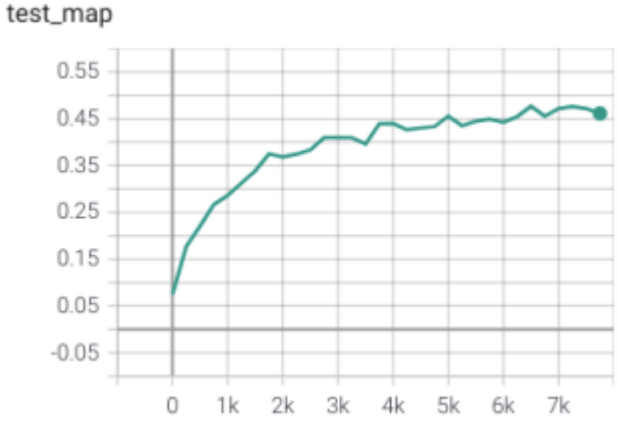
In [2]:
```
args = ARGS(epochs=50, batch_size=32, lr=0.0001,
            use_cuda=True, gamma=0.9, log_every=250,
            val_every=250, test_batch_size=16,
            save_at_end=True, save_freq=10)

model = ResNet
optimizer = torch.optim.Adam(model.parameters(), lr=args.lr)
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=20, gamma=args.
test_ap, test_map = trainer.train(args, model, optimizer, scheduler, model_name=
print('test map:', test_map)
```

```
Train Epoch: 0 [0 (0%)] Loss: 0.655183
test map (validation): 0.07534058953327674
Train Epoch: 1 [250 (59%)]      Loss: 0.218892
test map (validation): 0.177175142631832
Train Epoch: 3 [500 (18%)]      Loss: 0.197915
test map (validation): 0.2207908930041353
Train Epoch: 4 [750 (78%)]      Loss: 0.241303
test map (validation): 0.26692366200772927
Train Epoch: 6 [1000 (37%)]     Loss: 0.178215
test map (validation): 0.2862473268778083
Train Epoch: 7 [1250 (96%)]     Loss: 0.180843
test map (validation): 0.31267821891523273
Train Epoch: 9 [1500 (55%)]     Loss: 0.186540
test map (validation): 0.33858616131199876
Train Epoch: 11 [1750 (15%)]    Loss: 0.164025
test map (validation): 0.3749735587889866
Train Epoch: 12 [2000 (74%)]    Loss: 0.163688
test map (validation): 0.368346408887113
Train Epoch: 14 [2250 (33%)]    Loss: 0.178499
test map (validation): 0.37424829393373515
Train Epoch: 15 [2500 (92%)]    Loss: 0.160320
test map (validation): 0.38319528192067254
Train Epoch: 17 [2750 (52%)]    Loss: 0.177691
test map (validation): 0.40966181299588345
Train Epoch: 19 [3000 (11%)]    Loss: 0.146522
test map (validation): 0.41005308664679674
Train Epoch: 20 [3250 (70%)]    Loss: 0.180479
test map (validation): 0.40962363018467246
Train Epoch: 22 [3500 (29%)]    Loss: 0.180744
test map (validation): 0.3963020754254273
Train Epoch: 23 [3750 (89%)]    Loss: 0.169879
test map (validation): 0.43903465699617994
Train Epoch: 25 [4000 (48%)]    Loss: 0.148065
test map (validation): 0.43974724895163575
Train Epoch: 27 [4250 (7%)]     Loss: 0.144252
test map (validation): 0.42645977882349434
Train Epoch: 28 [4500 (66%)]    Loss: 0.174813
test map (validation): 0.43008582306028054
Train Epoch: 30 [4750 (25%)]    Loss: 0.167591
test map (validation): 0.43327790207520983
Train Epoch: 31 [5000 (85%)]    Loss: 0.167954
test map (validation): 0.4555399519840405
Train Epoch: 33 [5250 (44%)]    Loss: 0.147266
test map (validation): 0.4349755534439108
Train Epoch: 35 [5500 (3%)]     Loss: 0.146480
test map (validation): 0.44491227479833595
Train Epoch: 36 [5750 (62%)]    Loss: 0.135549
```

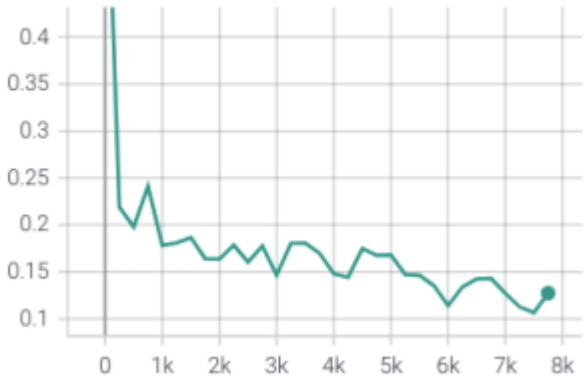
```
test map (validation): 0.44917670393986747
Train Epoch: 38 [6000 (22%)]    Loss: 0.114210
test map (validation): 0.4425242064902647
Train Epoch: 39 [6250 (81%)]    Loss: 0.134026
test map (validation): 0.45464507001943977
Train Epoch: 41 [6500 (40%)]    Loss: 0.142627
test map (validation): 0.4766089837877988
Train Epoch: 42 [6750 (99%)]    Loss: 0.143089
test map (validation): 0.4557379093441747
Train Epoch: 44 [7000 (59%)]    Loss: 0.127176
test map (validation): 0.47173339176828505
Train Epoch: 46 [7250 (18%)]    Loss: 0.112807
test map (validation): 0.47605732602367734
Train Epoch: 47 [7500 (77%)]    Loss: 0.106633
test map (validation): 0.47181862252117296
Train Epoch: 49 [7750 (36%)]    Loss: 0.127425
test map (validation): 0.4611959050584032
test map: 0.4734543259745834
```

In [4]:

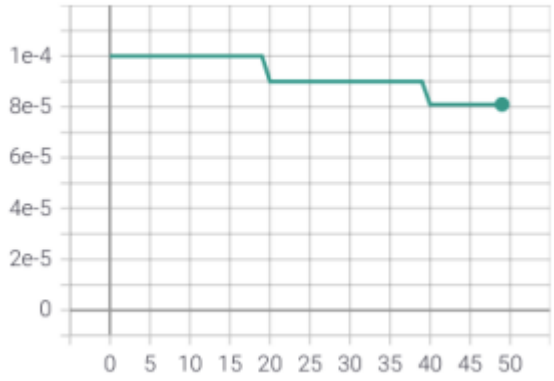
```
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
img = mpimg.imread('q3_1.png')
plt.imshow(img)
plt.axis("off")
plt.show()
img = mpimg.imread('q3_2.png')
plt.imshow(img)
plt.axis("off")
plt.show()
img = mpimg.imread('q3_3.png')
plt.imshow(img)
plt.axis("off")
plt.show()
img = mpimg.imread('q3_4.png')
plt.imshow(img)
plt.axis("off")
plt.show()
img = mpimg.imread('q3_5.png')
plt.imshow(img)
plt.axis("off")
plt.show()
img = mpimg.imread('q3_6.png')
plt.imshow(img)
plt.axis("off")
plt.show()
```



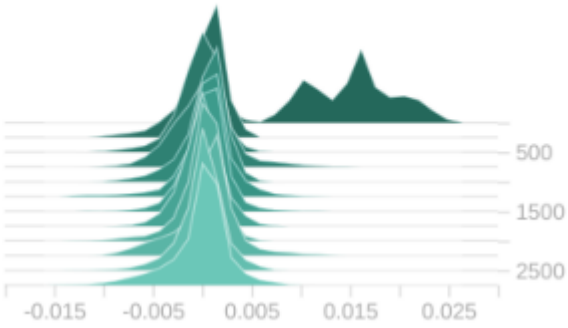
train_loss



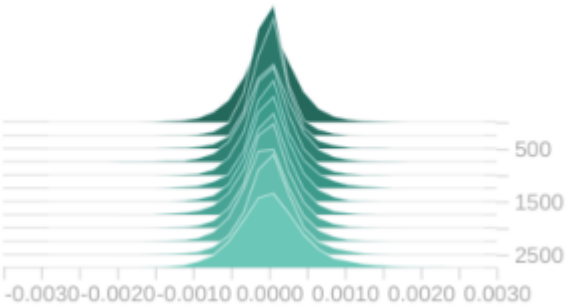
lr

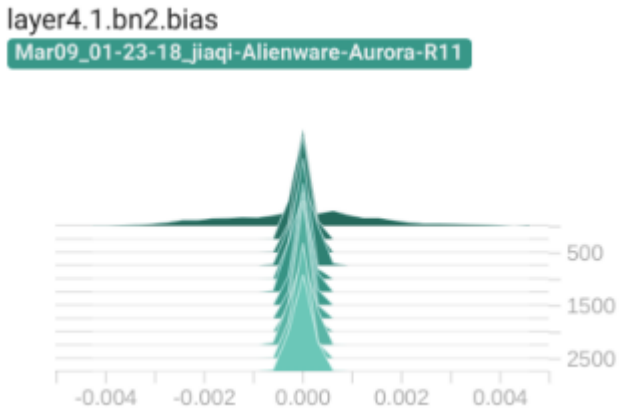


fc.weight Mar09_01-23-18_jiaqi-Alienware-Aurora-R11



layer1.1.conv1.weight
Mar09_01-23-18_jiaqi-Alienware-Aurora-R11




layer4.1.bn2.bias
Mar09_01-23-18_jiaqi-Alienware-Aurora-R11
500
1500
2500
-0.004
-0.002
0.000
0.002
0.004

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 weights and biases from scratch.

The model loading mechanism is based on names of the weights. It is easy to load pretrained models from `torchvision.models`, even when your model uses different names for weights. Please briefly explain how to load the weights correctly if the names do not match (hint).

YOUR ANSWER HERE

The state_dict of each model is ordered, so we can iterate through a model's state_dict() to retrieve each parameter's weights regardless of the parameter's name.

In [9]:

```
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.resnet = models.resnet18(pretrained=True)
        for param in self.resnet.parameters():
            param.requires_grad = False
        num_features = self.resnet.fc.in_features
        self.resnet.fc = nn.Linear(num_features, 20)
```

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

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

REMEMBER TO SAVE MODEL AT END OF TRAINING

In [10]:
```
args = ARGS(epochs=10, batch_size=32, lr=0.0001, use_cuda=True, gamma=1, test_ba
model = PretrainedResNet()
optimizer = torch.optim.Adam(model.parameters(), lr=args.lr)
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=1, gamma=args.g
test_ap, test_map = trainer.train(args, model, optimizer, scheduler, model_name=
print('test map:', test_map)
```

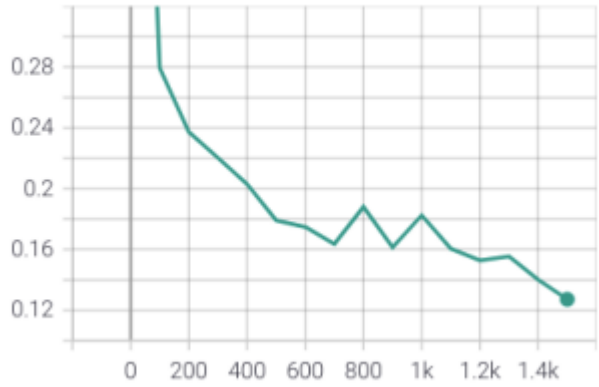
```
Train Epoch: 0 [0 (0%)] Loss: 0.734941
test map (validation): 0.08578939146440936
Train Epoch: 0 [100 (64%)]      Loss: 0.279321
test map (validation): 0.10951315533272668
Train Epoch: 1 [200 (27%)]      Loss: 0.237286
test map (validation): 0.16402468834730846
Train Epoch: 1 [300 (91%)]      Loss: 0.220260
test map (validation): 0.24599123433458195
Train Epoch: 2 [400 (55%)]      Loss: 0.203026
test map (validation): 0.341704939609551
Train Epoch: 3 [500 (18%)]      Loss: 0.179159
test map (validation): 0.4239706556001412
Train Epoch: 3 [600 (82%)]      Loss: 0.174807
test map (validation): 0.5018738518998134
Train Epoch: 4 [700 (46%)]      Loss: 0.163658
test map (validation): 0.5495332651726884
Train Epoch: 5 [800 (10%)]      Loss: 0.188048
test map (validation): 0.5940733983070333
Train Epoch: 5 [900 (73%)]      Loss: 0.161388
test map (validation): 0.6172160132346611
Train Epoch: 6 [1000 (37%)]     Loss: 0.182495
test map (validation): 0.6467297705499719
Train Epoch: 7 [1100 (1%)]      Loss: 0.160554
test map (validation): 0.6674625749247446
Train Epoch: 7 [1200 (64%)]     Loss: 0.152898
test map (validation): 0.6861990425439659
Train Epoch: 8 [1300 (28%)]     Loss: 0.155487
test map (validation): 0.7075196448683543
Train Epoch: 8 [1400 (92%)]     Loss: 0.140228
test map (validation): 0.7063816847721374
Train Epoch: 9 [1500 (55%)]     Loss: 0.127193
test map (validation): 0.719321845096174
test map: 0.727417615022198
```

YOUR TB SCREENSHOTS HERE

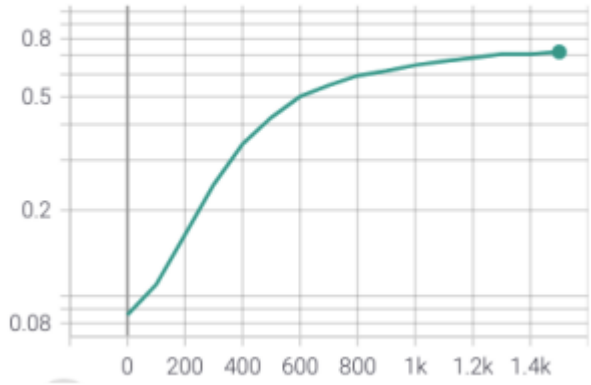
In [11]:
```
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
img = mpimg.imread('q4_1.png')
plt.imshow(img)
plt.axis("off")
plt.show()
img = mpimg.imread('q4_2.png')
plt.imshow(img)
```

```
plt.axis("off")
plt.show()
```

train_loss



test_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 of those images over the test set. You should use and compare the following feature representations for the nearest neighbors:

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

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
import utils
from utils import ARGS
from simple_cnn import SimpleCNN
from voc_dataset import VOCDataset
from sklearn.neighbors import NearestNeighbors
import numpy as np


class CaffeNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 96, kernel_size=11, stride=4, padding=0)
        self.conv2 = nn.Conv2d(96, 256, kernel_size=5, stride=1, padding=2)
        self.conv3 = nn.Conv2d(256, 384, kernel_size=3, stride=1, padding=1)
        self.conv4 = nn.Conv2d(384, 384, kernel_size=3, stride=1, padding=1)
        self.conv5 = nn.Conv2d(384, 256, kernel_size=3, stride=1, padding=1)
        self.relu = nn.ReLU()
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2)
        self.fc1 = nn.Linear(256 * 5 * 5, 4096)
        self.fc2 = nn.Linear(4096, 4096)
        self.fc3 = nn.Linear(4096, 20)
        self.dropout = nn.Dropout(p=0.5)
```

```
    def forward(self, x):
        x = self.conv1(x)
        x = self.relu(x)
        x = self.maxpool(x)
        x = self.conv2(x)
        x = self.relu(x)
        x = self.maxpool(x)
        x = self.conv3(x)
        x = self.relu(x)
        x = self.conv4(x)
        x = self.relu(x)
        x = self.conv5(x)
        x = self.relu(x)
        x = self.maxpool(x)

        flat_x = x.view(-1, 256 * 5 * 5)
        out = self.fc1(flat_x)
        out = self.relu(out)
        out = self.dropout(out)
        out = self.fc2(out)
        out = self.relu(out)
        out = self.dropout(out)
        out = self.fc3(out)
        return flat_x

resnet = models.resnet18(pretrained=True)
truncated_model = list(resnet.children())[:-1]
resnet = nn.Sequential(*truncated_model)
for p in resnet:
    p.requires_grad = False
```

In [2]:

```
# load images, calculate nearest neighbors, and plot
model = CaffeNet().to('cuda')
model.load_state_dict(torch.load("models/q2_50"))
model.eval()

test_loader = utils.get_data_loader('voc', train=False, batch_size=32, split='te
all_features = None
all_data = None

for batch_idx, (data, target, wgt) in enumerate(test_loader):
    # Get a batch of data
    data, target, wgt = data.to('cuda'), target.to('cuda'), wgt.to('cuda')
    output = model(data)
    if all_features is None:
        all_features = output.detach().cpu().numpy()
    else:
        all_features = np.concatenate((all_features, output.detach().cpu().numpy
    if all_data is None:
        all_data = data.detach().cpu().numpy()
    else:
        all_data = np.concatenate((all_data, data.detach().cpu().numpy()), axis=


nbrs = NearestNeighbors(n_neighbors=5, algorithm='ball_tree').fit(all_features)
distances0, indices0 = nbrs.kneighbors(all_features[0, :].reshape((1, -1)))
distances1, indices1 = nbrs.kneighbors(all_features[1, :].reshape((1, -1)))
distances2, indices2 = nbrs.kneighbors(all_features[2, :].reshape((1, -1)))
```

In [3]:
```
print("CaffeNet\n")

print("The leftmost image is the picked image, the other 4 is its nearest neighb
fig0, ax0 = plt.subplots(1, 5)
fig0.set_figwidth(15)
for i, index in enumerate(indices0[0]):
    image = np.moveaxis(all_data[index], 0, -1)
    image = (image - image.min()) / (image.max() - image.min())
    ax0[i].axis("off")
    ax0[i].imshow(image)
plt.show()

print("The leftmost image is the picked image, the other 4 is its nearest neighb
fig1, ax1 = plt.subplots(1, 5)
fig1.set_figwidth(15)
for i, index in enumerate(indices1[0]):
    image = np.moveaxis(all_data[index], 0, -1)
    image = (image - image.min()) / (image.max() - image.min())
    ax1[i].axis("off")
    ax1[i].imshow(image)
plt.show()

print("The leftmost image is the picked image, the other 4 is its nearest neighb
fig2, ax2 = plt.subplots(1, 5)
fig2.set_figwidth(15)
for i, index in enumerate(indices2[0]):
    image = np.moveaxis(all_data[index], 0, -1)
    image = (image - image.min()) / (image.max() - image.min())
    ax2[i].axis("off")
    ax2[i].imshow(image)
plt.show()
```

```
CaffeNet

The leftmost image is the picked image, the other 4 is its nearest neighbors
```

```
The leftmost image is the picked image, the other 4 is its nearest neighbors
```

```
The leftmost image is the picked image, the other 4 is its nearest neighbors
```

In [4]:
```
# load images, calculate nearest neighbors, and plot
model = resnet.to('cuda')
model.eval()

test_loader = utils.get_data_loader('voc', train=False, batch_size=32, split='te
all_features = None
all_data = None

for batch_idx, (data, target, wgt) in enumerate(test_loader):
    # Get a batch of data
    data, target, wgt = data.to('cuda'), target.to('cuda'), wgt.to('cuda')
    output = model(data)
    if all_features is None:
        all_features = output.detach().cpu().numpy().reshape((data.shape[0], -1)
    else:
        all_features = np.concatenate((all_features,
                                       output.detach().cpu().numpy().reshape((da
    if all_data is None:
        all_data = data.detach().cpu().numpy()
    else:
        all_data = np.concatenate((all_data, data.detach().cpu().numpy()), axis=


nbrs = NearestNeighbors(n_neighbors=5, algorithm='ball_tree').fit(all_features)
distances0, indices0 = nbrs.kneighbors(all_features[0, :].reshape((1, -1)))
distances1, indices1 = nbrs.kneighbors(all_features[1, :].reshape((1, -1)))
distances2, indices2 = nbrs.kneighbors(all_features[2, :].reshape((1, -1)))
```

In [5]:
```
print("ResNet\n")

print("The leftmost image is the picked image, the other 4 is its nearest neighb
fig0, ax0 = plt.subplots(1, 5)
fig0.set_figwidth(15)
for i, index in enumerate(indices0[0]):
    image = np.moveaxis(all_data[index], 0, -1)
    image = (image - image.min()) / (image.max() - image.min())
    ax0[i].axis("off")
    ax0[i].imshow(image)
plt.show()

print("The leftmost image is the picked image, the other 4 is its nearest neighb
fig1, ax1 = plt.subplots(1, 5)
fig1.set_figwidth(15)
for i, index in enumerate(indices1[0]):
    image = np.moveaxis(all_data[index], 0, -1)
    image = (image - image.min()) / (image.max() - image.min())
    ax1[i].axis("off")
    ax1[i].imshow(image)
plt.show()

print("The leftmost image is the picked image, the other 4 is its nearest neighb
fig2, ax2 = plt.subplots(1, 5)
fig2.set_figwidth(15)
for i, index in enumerate(indices2[0]):
    image = np.moveaxis(all_data[index], 0, -1)
    image = (image - image.min()) / (image.max() - image.min())
    ax2[i].axis("off")
    ax2[i].imshow(image)
plt.show()
```

ResNet

The leftmost image is the picked image, the other 4 is its nearest neighbors

The leftmost image is the picked image, the other 4 is its nearest neighbors

The leftmost image is the picked image, the other 4 is its nearest neighbors

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 [31]:

```
# plot t-SNE here
model = CaffeNet().to('cuda')
model.load_state_dict(torch.load("models/q2_50"))
model.eval()

test_loader = utils.get_data_loader('voc', train=False, batch_size=50, split='te
all_features = None
all_targets = None
counter = 0
for batch_idx, (data, target, wgt) in enumerate(test_loader):
    if counter == 20:
        break
    data, target, wgt = data.to('cuda'), target.to('cuda'), wgt.to('cuda')
    output = model(data)
    if all_features is None:
        all_features = output.detach().cpu().numpy()
    else:
        all_features = np.concatenate((all_features, output.detach().cpu().numpy
    if all_targets is None:
```

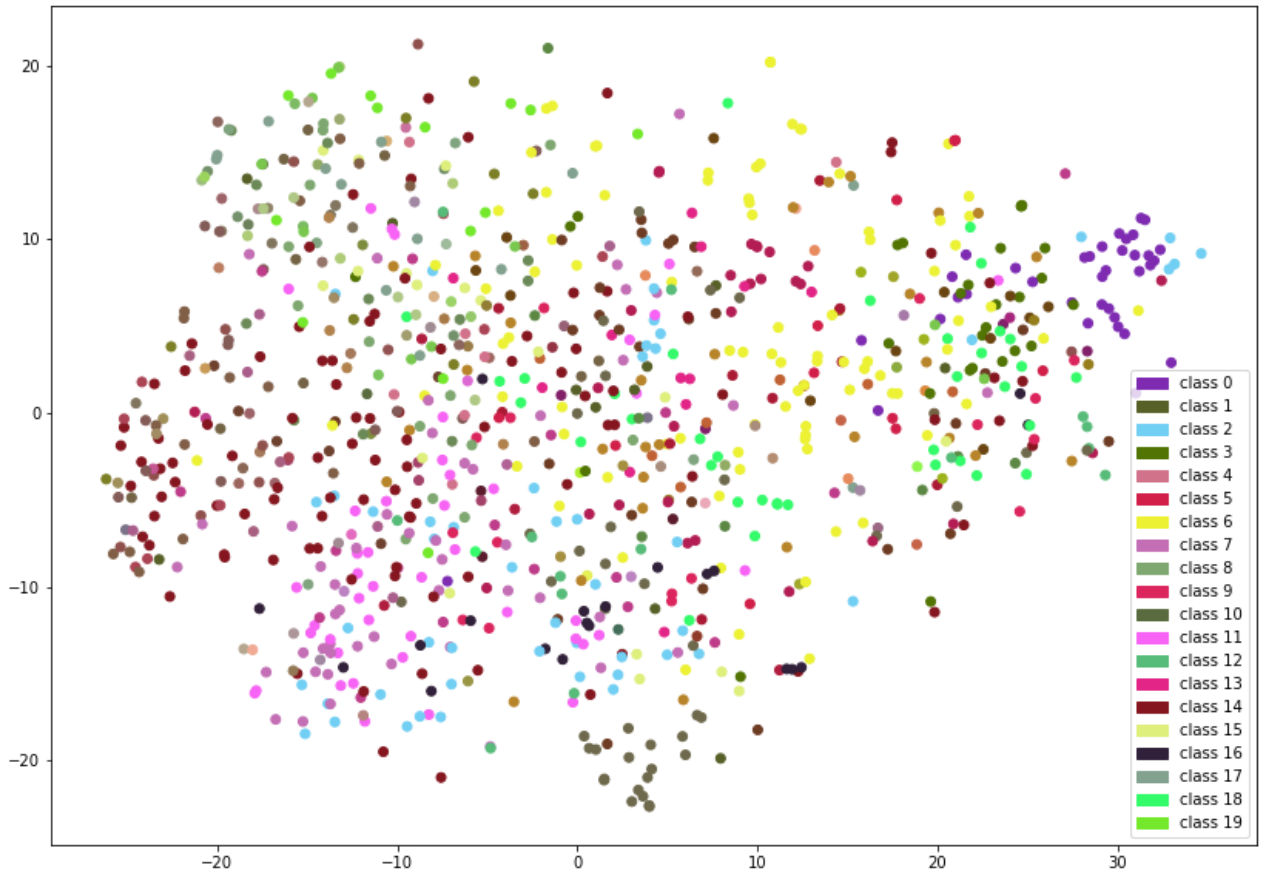
```
            all_targets = target.detach().cpu().numpy()
        else:
            all_targets = np.concatenate((all_targets, target.detach().cpu().numpy(
        counter += 1
```

In [16]:

```
import numpy as np
from sklearn.manifold import TSNE
import seaborn as sns

colors = []
for i in range(20):
    color = list(np.random.choice(range(256), size=3))
    colors.append(color)

features_embedded = TSNE(n_components=2).fit_transform(all_features)
```

In [51]:

```
c = np.array(colors)
mean_cs = []
for i in range(1000):
    multi_c = c[np.where(all_targets[i, :].astype(np.int)==1)]
    mean_c = np.mean(multi_c, axis=0)
    mean_cs.append(mean_c.astype(np.int))

plt.figure(figsize=(14, 10))
import matplotlib.patches as mpatches
handles = []
for i in range(c.shape[0]):
    patch = mpatches.Patch(color=c[i] / 255, label="class " + str(i))
    handles.append(patch)
plt.legend(handles=handles)

c = np.array(mean_cs)
plt.scatter(features_embedded[:, 0], features_embedded[:, 1], c=c/255)
plt.show()
```



5.3 Are some classes harder? (6pts)

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

YOUR ANSWER HERE

We can see that the airplane class is easy, since the airplanes are usually in the center of the image, and the background is distinguishable from the plane. There is very little room to misclassify. However, for the pottedplant class, the sizes and shapes of the plants are very different, so it is quite hard to classify this class.

Some classes, like class 16 (Sheep), see large gains due to pre-training (from 0.2281 to 0.8272). This might be becasue we don't really have a lot of sheep images in our dataset, but we have a lot of sheep images in the ImageNet dataset.

In [65]:
```
class CaffeNet_(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 96, kernel_size=11, stride=4, padding=0)
        self.conv2 = nn.Conv2d(96, 256, kernel_size=5, stride=1, padding=2)
        self.conv3 = nn.Conv2d(256, 384, kernel_size=3, stride=1, padding=1)
        self.conv4 = nn.Conv2d(384, 384, kernel_size=3, stride=1, padding=1)
        self.conv5 = nn.Conv2d(384, 256, kernel_size=3, stride=1, padding=1)
```

```
        self.relu = nn.ReLU()
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2)
        self.fc1 = nn.Linear(256 * 5 * 5, 4096)
        self.fc2 = nn.Linear(4096, 4096)
        self.fc3 = nn.Linear(4096, 20)
        self.dropout = nn.Dropout(p=0.5)

    def forward(self, x):
        x = self.conv1(x)
        x = self.relu(x)
        x = self.maxpool(x)
        x = self.conv2(x)
        x = self.relu(x)
        x = self.maxpool(x)
        x = self.conv3(x)
        x = self.relu(x)
        x = self.conv4(x)
        x = self.relu(x)
        x = self.conv5(x)
        x = self.relu(x)
        x = self.maxpool(x)

        flat_x = x.view(-1, 256 * 5 * 5)
        out = self.fc1(flat_x)
        out = self.relu(out)
        out = self.dropout(out)
        out = self.fc2(out)
        out = self.relu(out)
        out = self.dropout(out)
        out = self.fc3(out)
        return out


class PretrainedResNet_(nn.Module):
    def __init__(self):
        super().__init__()
        self.resnet = models.resnet18(pretrained=True)
        for param in self.resnet.parameters():
            param.requires_grad = False
        num_features = self.resnet.fc.in_features
        self.resnet.fc = nn.Linear(num_features, 20)

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

model1, model2 = CaffeNet_().to('cuda'), PretrainedResNet_().to('cuda')
model1.load_state_dict(torch.load("models/q2_50"))
model2.load_state_dict(torch.load("models/q4_10"))
model1.eval()
model2.eval()

test_loader = utils.get_data_loader('voc', train=False, batch_size=32, split='te
ap1, map1 = utils.eval_dataset_map(model1, 'cuda', test_loader)
ap2, map2 = utils.eval_dataset_map(model2, 'cuda', test_loader)
```

In [63]:

```
print("caffenet")
for i, ap in enumerate(ap1):
    print("class " + str(i) + ":", ap)
```

```
print("resnet")
for i, ap in enumerate(ap2):
    print("class " + str(i) + ":", ap)
```

```
caffenet
class 0: 0.6571594368211714
class 1: 0.4146887133450841
class 2: 0.3912622672351793
class 3: 0.4518777118809009
class 4: 0.08013075429068757
class 5: 0.38600263386583133
class 6: 0.6242862185129479
class 7: 0.41953464397269563
class 8: 0.28765640253403024
class 9: 0.1521166953334916
class 10: 0.27783947389837693
class 11: 0.3041652753368963
class 12: 0.6206574697871048
class 13: 0.4432827291334196
class 14: 0.7761796835850486
class 15: 0.23661580814430566
class 16: 0.22809223481213398
class 17: 0.3246888150772234
class 18: 0.5504456105185601
class 19: 0.37415340681149795
resnet
class 0: 0.9419235037056286
class 1: 0.795997299716794
class 2: 0.9084771841265284
class 3: 0.7871659758602396
class 4: 0.3088018903481415
class 5: 0.7404690874522255
class 6: 0.8812303927242877
class 7: 0.8967646592792092
class 8: 0.4907394713333879
class 9: 0.5324915842399371
class 10: 0.5192030102788457
class 11: 0.8199904353538435
class 12: 0.47276066935745226
class 13: 0.7351923130568266
class 14: 0.9139758021752549
class 15: 0.463156598686071
class 16: 0.8271949642121673
class 17: 0.6078466750511264
class 18: 0.9169913441676288
class 19: 0.7063906858914517
```

In [5]:

```
test_loader = utils.get_data_loader('voc', train=False, batch_size=1, split='tes
easy_images = []
hard_images = []
counter_easy, counter_hard = 0, 0
for batch_idx, (data, target, wgt) in enumerate(test_loader):
    data, target, wgt = data.to('cuda'), target.to('cuda'), wgt.to('cuda')
    if len(easy_images) < 5:
        if target[0, 0].item() == 1.:
            easy_images.append(data[0].cpu().detach().numpy())

    if len(hard_images) < 5:
        if target[0, 15].item() == 1.:
            hard_images.append(data[0].cpu().detach().numpy())

    if len(easy_images) >= 5 and len(hard_images) >= 5:
        break
```

In [8]:

```
print("Easy Examples")
fig4, ax4 = plt.subplots(1, 5)
fig4.set_figwidth(15)
for i in range(5):
    image = np.moveaxis(easy_images[i], 0, -1)
    image = (image - image.min()) / (image.max() - image.min())
    ax4[i].axis("off")
    ax4[i].imshow(image)
plt.show()

print("Hard Examples")
fig5, ax5 = plt.subplots(1, 5)
fig5.set_figwidth(15)
for i in range(5):
    image = np.moveaxis(hard_images[i], 0, -1)
    image = (image - image.min()) / (image.max() - image.min())
    ax5[i].axis("off")
    ax5[i].imshow(image)
plt.show()
```

```
Easy Examples
```

```
Hard Examples
```

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 [ ]:
```
# implement mixup regularization here and show performance
```