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

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

1.1 Setup the dataset

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

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

```
wget http://host.robots.ox.ac.uk/pascal/VOC/voc2007/VOCtrainval
_06-Nov-2007.tar
tar -xf VOCtrainval_06-Nov-2007.tar

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

1.2 Write a dataloader with data augmentation (5 pts)

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

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

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

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

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

DESCRIBE YOUR AUGMENTATION PIPELINE HERE**

Train Augmentations: Added random crop and random horizontal flip to all training images. The cropped size is set to 400. The images are normalized with the mean values as [123,106,255] and std as 1 for all channels.

Test Augmentations: Added center crop to all images. The cropped size is set to 400.

1.3 Measure Performance (5 pts)

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

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

1.4 Let's Start Training! (5 pts)

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

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

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

# create hyperparameter argument class
args = ARGS(epochs=5)
print(args)
# %env CUDA_DEVICE_ORDER=PCI_BUS_ID
# os.environ[“CUDA_VISIBLE_DEVICES”]=“1”
# %env CUDA_VISIBLE_DEVICES=3
# import notebook_util
# notebook_util.pick_gpu_lowest_memory()
import os
%env CUDA_VISIBLE_DEVICES=3
```

```
args.batch_size = 32
args.device = cuda
args.epochs = 5
args.gamma = 0.7
args.log_every = 100
args.lr = 1.0
args.save_at_end = True
args.save_dir = None
args.save_freq = -1
args.test_batch_size = 32
args.val_every = 100

env: CUDA_VISIBLE_DEVICES=3
```

In [ ]:
```
#initializes (your) naiive model
args = ARGS(epochs=5, batch_size=20, lr=0.001,gamma=0.95)
model = SimpleCNN(num_classes=len(VOCDataset.CLASS_NAMES), inp_size=64,
# 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=5, gamm
# 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: 0.689729
```

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

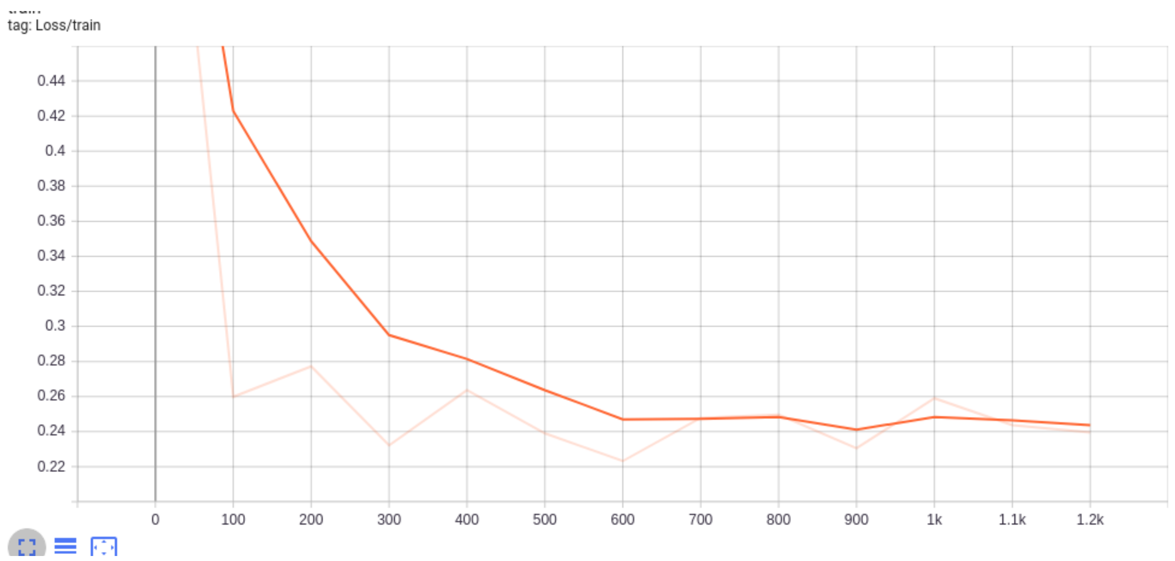
To use TensorBoard in Pytorch, there are two options: TensorBoard in Pytorch (https://pytorch.org/docs/stable/tensorboard.html) (for Pytorch >= 1.1.0) or TensorBoardX (https://github.com/lanpa/tensorboardX) - a third party library. Add code in `trainer.py` to visualize the testing MAP and training loss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

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

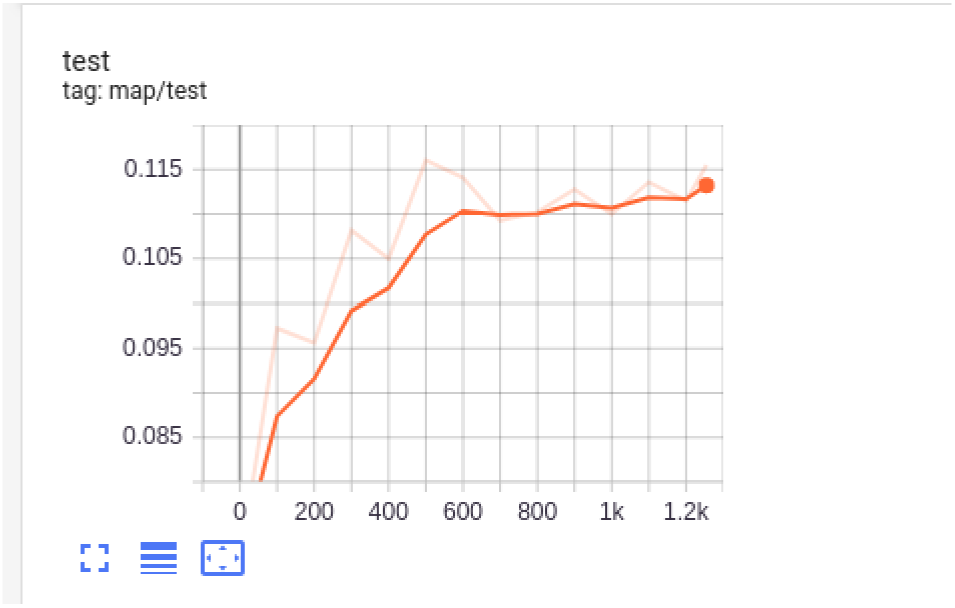
In [ ]:
```
args = ARGS(epochs=5, batch_size=20, lr=0.001,gamma=0.95)
model = SimpleCNN(num_classes=len(VOCDataset.CLASS_NAMES), inp_size=64,
optimizer = torch.optim.Adam(model.parameters(), lr=args.lr)
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=5, gamm
test_ap, test_map = trainer.train(args, model, optimizer, scheduler)
print('test map:', test_map)
```

```
Train Epoch: 0 [0 (0%)] Loss: 0.694538
Test Epoch: 0 [0 (0%)]  mAP: 0.070930
Train Epoch: 0 [100 (40%)]      Loss: 0.259850
Test Epoch: 0 [100 (40%)]       mAP: 0.097251
Train Epoch: 0 [200 (80%)]      Loss: 0.277247
Test Epoch: 0 [200 (80%)]       mAP: 0.095631
Train Epoch: 1 [300 (20%)]      Loss: 0.232117
Test Epoch: 1 [300 (20%)]       mAP: 0.108148
Train Epoch: 1 [400 (59%)]      Loss: 0.263631
Test Epoch: 1 [400 (59%)]       mAP: 0.104974
Train Epoch: 1 [500 (99%)]      Loss: 0.238991
Test Epoch: 1 [500 (99%)]       mAP: 0.116036
Train Epoch: 2 [600 (39%)]      Loss: 0.223213
Test Epoch: 2 [600 (39%)]       mAP: 0.114084
Train Epoch: 2 [700 (79%)]      Loss: 0.247888
Test Epoch: 2 [700 (79%)]       mAP: 0.109263
Train Epoch: 3 [800 (19%)]      Loss: 0.249655
Test Epoch: 3 [800 (19%)]       mAP: 0.110193
Train Epoch: 3 [900 (59%)]      Loss: 0.230486
Test Epoch: 3 [900 (59%)]       mAP: 0.112760
Train Epoch: 3 [1000 (98%)]     Loss: 0.259004
Test Epoch: 3 [1000 (98%)]      mAP: 0.110008
Train Epoch: 4 [1100 (38%)]     Loss: 0.243651
Test Epoch: 4 [1100 (38%)]      mAP: 0.113573
Train Epoch: 4 [1200 (78%)]     Loss: 0.239551
Test Epoch: 4 [1200 (78%)]      mAP: 0.111469
```

INSERT YOUR TENSORBOARD SCREENSHOTS HERE


tag: Loss/train
0.44
0.42
0.4
0.38
0.36
0.34
0.32
0.3
0.28
0.26
0.24
0.22
0
100
200
300
400
500
600
700
800
900
1k
1.1k
1.2k

)


test
tag: map/test
0.115
0.105
0.095
0.085
0
200
400
600
800
1k
1.2k

```
In [ ]:
```