```
from google.colab import drive
drive.mount('/content/gdrive')
import os
os.chdir("gdrive/My Drive/assignment3_p2")
```

```
    Drive already mounted at /content/gdrive; to attempt to forcibly remount, call
```

```
!pip install torch==1.5.1+cu101 torchvision==0.6.1+cu101 -f https://download.pytorc
```

```
    Looking in links: https://download.pytorch.org/whl/torch_stable.html
    Requirement already satisfied: torch==1.5.1+cu101 in /usr/local/lib/python3.6/
    Requirement already satisfied: torchvision==0.6.1+cu101 in /usr/local/lib/pyth
    Requirement already satisfied: numpy in /usr/local/lib/python3.6/dist-packages
    Requirement already satisfied: future in /usr/local/lib/python3.6/dist-package
    Requirement already satisfied: pillow>=4.1.1 in /usr/local/lib/python3.6/dist-
```

```
# import shutil
# shutil.copyfile("VOCtrainval_06-Nov-2007.tar", "/content/VOCtrainval_06-Nov-2007.
# !tar -xf "/content/VOCtrainval_06-Nov-2007.tar" -C "/content/"
# shutil.move("/content/VOCdevkit/", "/content/VOCdevkit_2007")

# shutil.copyfile("VOCtest_06-Nov-2007.tar", "/content/VOCtest_06-Nov-2007.tar")
# !tar -xf "/content/VOCtest_06-Nov-2007.tar" -C "/content/"
# shutil.move("/content/VOCdevkit/VOC2007", "/content/VOCdevkit_2007/VOC2007test")
```

```
import os
import random

import cv2
import numpy as np

import torch
from torch.utils.data import DataLoader
from torchvision import models

from resnet_yolo import resnet50
from yolo_loss import YoloLoss
from dataset import VocDetectorDataset
from eval_voc import evaluate
from predict import predict_image
from config import VOC_CLASSES, COLORS
from kaggle_submission import output_submission_csv
import matplotlib.pyplot as plt

%matplotlib inline
%load_ext autoreload
%autoreload 2
```

Assignment3 Part2: Yolo Detection

We provide you a Yolo Detection network implementation, which is not finished. You are asked to complete the implementation by writing the loss function.

What to do

You are asked to implement the loss function in `yolo_loss.py`. You can use `yolo_loss_debug_tool.ipynb` to help you debug.

What to submit

See the submission template for what to submit.

Initialization

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

```
# YOLO network hyperparameters
B = 2  # number of bounding box predictions per cell
S = 14  # width/height of network output grid (larger than 7x7 from paper since we
```

To implement Yolo we will rely on a pretrained classifier as the backbone for our detection network. PyTorch offers a variety of models which are pretrained on ImageNet in the [torchvision.models]() package. In particular, we will use the ResNet50 architecture as a base for our detector. This is different from the base architecture in the Yolo paper and also results in a different output grid size (14x14 instead of 7x7).

Models are typically pretrained on ImageNet since the dataset is very large (> 1million images) and widely used. The pretrained model provides a very useful weight initialization for our detector, so that the network is able to learn quickly and efficitively.

```
load_network_path = "best_detector.pth"
pretrained = True

# use to load a previously trained network
if load_network_path is not None:
    print('Loading saved network from {}'.format(load_network_path))
    net = resnet50().to(device)
    net.load_state_dict(torch.load(load_network_path))
else:
    print('Load pre-trained model')
    net = resnet50(pretrained=pretrained).to(device)
```

```
Load pre-trained model
```

```
learning_rate = 0.001
num_epochs = 50
```

```
num_epochs = 50
batch_size = 24

# Yolo loss component coefficients (as given in Yolo v1 paper)
lambda_coord = 5
lambda_noobj = 0.5


criterion = YoloLoss(S, B, lambda_coord, lambda_noobj)
optimizer = torch.optim.SGD(net.parameters(), lr=learning_rate, momentum=0.9, weigh
```

Reading Pascal Data

Since Pascal is a small dataset (5000 in train+val) we have combined the train and val splits to train our detector. This is not typically a good practice, but we will make an exception in this case to be able to get reasonable detection results with a comparatively small object detection dataset.

The train dataset loader also using a variety of data augmentation techniques including random shift, scaling, crop, and flips. Data augmentation is slightly more complicated for detection dataset since the bounding box annotations must be kept consistent through the transformations.

Since the output of the dector network we train is an SxSx(B*5+C), we use an encoder to convert the original bounding box coordinates into relative grid bounding box coordinates corresponding to the the expected output. We also use a decoder which allows us to convert the opposite direction into image coordinate bounding boxes.

```
file_root_train = 'VOCdevkit_2007/VOC2007/JPEGImages/'
# file_root_train = 'VOCdevkit_2007/VOC2007/'
annotation_file_train = 'voc2007.txt'

train_dataset = VocDetectorDataset(root_img_dir=file_root_train,dataset_file=annota
train_loader = DataLoader(train_dataset,batch_size=batch_size,shuffle=True,num_work
print('Loaded %d train images' % len(train_dataset))
```

```
Initializing dataset
Loaded 5011 train images
```

```
file_root_test = 'VOCdevkit_2007/VOC2007test/JPEGImages/'
# file_root_test = 'VOCdevkit_2007/VOC2007test/'
annotation_file_test = 'voc2007test.txt'

test_dataset = VocDetectorDataset(root_img_dir=file_root_test,dataset_file=annotati
test_loader = DataLoader(test_dataset,batch_size=batch_size,shuffle=False,num_worke
print('Loaded %d test images' % len(test_dataset))
```

```
Initializing dataset
Loaded 4950 test images
```

Train detector

```
best_test_loss = np.inf

for epoch in range(num_epochs):
    net.train()

    # Update learning rate late in training
    if epoch == 30 or epoch == 40:
        learning_rate /= 10.0

    for param_group in optimizer.param_groups:
        param_group['lr'] = learning_rate

    print('\n\nStarting epoch %d / %d' % (epoch + 1, num_epochs))
    print('Learning Rate for this epoch: {}'.format(learning_rate))

    total_loss = 0.

    for i, (images, target) in enumerate(train_loader):
        images, target = images.to(device), target.to(device)

        pred = net(images)
        loss = criterion(pred,target)
        total_loss += loss.item()

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        if (i+1) % 5 == 0:
            print('Epoch [%d/%d], Iter [%d/%d] Loss: %.4f, average_loss: %.4f'
                  % (epoch+1, num_epochs, i+1, len(train_loader), loss.item(), tota

    # evaluate the network on the test data
    with torch.no_grad():
        test_loss = 0.0
        net.eval()
        for i, (images, target) in enumerate(test_loader):
            images, target = images.to(device), target.to(device)

            pred = net(images)
            loss = criterion(pred,target)
            test_loss += loss.item()
        test_loss /= len(test_loader)

    if best_test_loss > test_loss:
        best_test_loss = test_loss
        print('Updating best test loss: %.5f' % best_test_loss)
        torch.save(net.state_dict(),'best_detector.pth')

    torch.save(net.state_dict(),'detector.pth')
```

```
Epoch [49/50], Iter [25/209] Loss: 2.7405, average_loss: 1.8703
Epoch [49/50], Iter [30/209] Loss: 1.8052, average_loss: 1.8367
Epoch [49/50], Iter [35/209] Loss: 1.7098, average_loss: 1.8035
Epoch [49/50], Iter [40/209] Loss: 2.1116, average_loss: 1.8153
Epoch [49/50], Iter [45/209] Loss: 1.5421, average_loss: 1.8089
Epoch [49/50], Iter [50/209] Loss: 1.5820, average_loss: 1.8156
Epoch [49/50], Iter [55/209] Loss: 1.4845, average_loss: 1.8108
Epoch [49/50], Iter [60/209] Loss: 1.5350, average_loss: 1.7857
Epoch [49/50], Iter [65/209] Loss: 1.5175, average_loss: 1.7843
Epoch [49/50], Iter [70/209] Loss: 1.7349, average_loss: 1.8115
Epoch [49/50], Iter [75/209] Loss: 1.5485, average_loss: 1.8092
Epoch [49/50], Iter [80/209] Loss: 2.1778, average_loss: 1.8102
Epoch [49/50], Iter [85/209] Loss: 1.5317, average_loss: 1.7903
Epoch [49/50], Iter [90/209] Loss: 1.3252, average_loss: 1.7835
Epoch [49/50], Iter [95/209] Loss: 1.8122, average_loss: 1.7691
Epoch [49/50], Iter [100/209] Loss: 1.5078, average_loss: 1.7702
Epoch [49/50], Iter [105/209] Loss: 1.8509, average_loss: 1.7685
Epoch [49/50], Iter [110/209] Loss: 1.3832, average_loss: 1.7582
Epoch [49/50], Iter [115/209] Loss: 1.7506, average_loss: 1.7499
Epoch [49/50], Iter [120/209] Loss: 1.6768, average_loss: 1.7441
Epoch [49/50], Iter [125/209] Loss: 1.6072, average_loss: 1.7410
Epoch [49/50], Iter [130/209] Loss: 2.0268, average_loss: 1.7428
Epoch [49/50], Iter [135/209] Loss: 1.9117, average_loss: 1.7487
Epoch [49/50], Iter [140/209] Loss: 1.6591, average_loss: 1.7587
Epoch [49/50], Iter [145/209] Loss: 1.5211, average_loss: 1.7558
Epoch [49/50], Iter [150/209] Loss: 1.3887, average_loss: 1.7503
Epoch [49/50], Iter [155/209] Loss: 2.2669, average_loss: 1.7555
Epoch [49/50], Iter [160/209] Loss: 2.9388, average_loss: 1.7683
Epoch [49/50], Iter [165/209] Loss: 1.1610, average_loss: 1.7649
Epoch [49/50], Iter [170/209] Loss: 1.7106, average_loss: 1.7637
Epoch [49/50], Iter [175/209] Loss: 1.4156, average_loss: 1.7598
Epoch [49/50], Iter [180/209] Loss: 1.2923, average_loss: 1.7529
Epoch [49/50], Iter [185/209] Loss: 1.7030, average_loss: 1.7475
Epoch [49/50], Iter [190/209] Loss: 1.4662, average_loss: 1.7451
Epoch [49/50], Iter [195/209] Loss: 1.3205, average_loss: 1.7494
Epoch [49/50], Iter [200/209] Loss: 1.8669, average_loss: 1.7493
Epoch [49/50], Iter [205/209] Loss: 2.7511, average_loss: 1.7479
Updating best test loss: 2.68434


Starting epoch 50 / 50
Learning Rate for this epoch: 1e-05
Epoch [50/50], Iter [5/209] Loss: 2.1002, average_loss: 1.8417
Epoch [50/50], Iter [10/209] Loss: 1.6859, average_loss: 1.7162
Epoch [50/50], Iter [15/209] Loss: 2.2123, average_loss: 1.7989
Epoch [50/50], Iter [20/209] Loss: 1.7928, average_loss: 1.7833
Epoch [50/50], Iter [25/209] Loss: 1.7774, average_loss: 1.7942
Epoch [50/50], Iter [30/209] Loss: 1.8237, average_loss: 1.7797
Epoch [50/50], Iter [35/209] Loss: 1.4604, average_loss: 1.7791
Epoch [50/50], Iter [40/209] Loss: 1.7047, average_loss: 1.7932
Epoch [50/50], Iter [45/209] Loss: 1.9074, average_loss: 1.7923
Epoch [50/50], Iter [50/209] Loss: 1.5433, average_loss: 1.7802
Epoch [50/50], Iter [55/209] Loss: 1.5468, average_loss: 1.7779
Epoch [50/50], Iter [60/209] Loss: 1.8126, average_loss: 1.7728
Epoch [50/50], Iter [65/209] Loss: 1.8309, average_loss: 1.7810
Epoch [50/50], Iter [70/209] Loss: 1.4800, average_loss: 1.7796
Epoch [50/50], Iter [75/209] Loss: 1.6934, average_loss: 1.7798
Epoch [50/50], Iter [80/209] Loss: 2.1115, average_loss: 1.7868
Epoch [50/50], Iter [85/209] Loss: 1.4839, average_loss: 1.7817
Epoch [50/50], Iter [90/209] Loss: 2.1335, average_loss: 1.7752
```

View example predictions

```
net.eval()

# select random image from test set
image_name = random.choice(test_dataset.fnames)
image = cv2.imread(os.path.join(file_root_test, image_name))
image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

print('predicting...')
result = predict_image(net, image_name, root_img_directory=file_root_test)
for left_up, right_bottom, class_name, _, prob in result:
    color = COLORS[VOC_CLASSES.index(class_name)]
    cv2.rectangle(image, left_up, right_bottom, color, 2)
    label = class_name + str(round(prob, 2))
    text_size, baseline = cv2.getTextSize(label, cv2.FONT_HERSHEY_SIMPLEX, 0.4, 1)
    p1 = (left_up[0], left_up[1] - text_size[1])
    cv2.rectangle(image, (p1[0] - 2 // 2, p1[1] - 2 - baseline), (p1[0] + text_size
                  color, -1)
    cv2.putText(image, label, (p1[0], p1[1] + baseline), cv2.FONT_HERSHEY_SIMPLEX,

plt.figure(figsize = (15,15))
plt.imshow(image)
```

```
predicting...
<matplotlib.image.AxesImage at 0x7f7322ef07b8>
```

Evaluate on Test

To evaluate detection results we use mAP (mean of average precision over each class)


```
test_aps = evaluate(net, test_dataset_file=annotation_file_test)
```

```
---Evaluate model on test samples---
100%|████████████████| 4950/4950 [02:01<00:00, 40.83it/s]
---class aeroplane ap 0.5207943965230863---
---class bicycle ap 0.6228150397293724---
---class bird ap 0.47934208875403206---
---class boat ap 0.2978074901244012---
---class bottle ap 0.22236373727318803---
---class bus ap 0.5987360983786492---
---class car ap 0.6360978513402967---
---class cat ap 0.7205577063582964---
---class chair ap 0.2959844818865669---
---class cow ap 0.5061604635134203---
---class diningtable ap 0.347764420696824---
---class dog ap 0.6541402977339184---
---class horse ap 0.679804655354225---
---class motorbike ap 0.5502349418425755---
---class person ap 0.5261194412093616---
---class pottedplant ap 0.18589932123662786---
---class sheep ap 0.422578885831798---
---class sofa ap 0.4954896891515419---
---class train ap 0.6998623727638267---
---class tvmonitor ap 0.4920808315756556---
---map 0.4977317105638832---
```

```
output_submission_csv('my_solution.csv', test_aps)
```