mamalis2_mp3_part2

November 2, 2020

[4]:
```
import os
import random
from predict import decoder

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
```

1 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.

1.1 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.

1.2 What to submit

See the submission template for what to submit.

1.3 Initialization

[5]:
```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

[6]:
```
# 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 use a different network)
```

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 effictively.

[7]:
```
# load_network_path = 'detector.pth'
load_network_path = None


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
```

[8]:
```
learning_rate = 0.001
num_epochs = 70
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,␣
 ↪weight_decay=5e-4)
```

```
# load_network_path = 'detector_list.pth'
# checkpoint = torch.load(load_network_path)
# net.load_state_dict(checkpoint['model_state_dict'])
# optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
# epoch = checkpoint['epoch']
# loss = checkpoint['loss']
```

1.4 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.

[10]:
```
file_root_train = 'VOCdevkit_2007/VOC2007/JPEGImages/'
annotation_file_train = 'voc2007.txt'

train_dataset =␣
 ↪VocDetectorDataset(root_img_dir=file_root_train,dataset_file=annotation_file_train,train=Tr
 ↪S=S)
train_loader =␣
 ↪DataLoader(train_dataset,batch_size=batch_size,shuffle=True,num_workers=0)
print('Loaded %d train images' % len(train_dataset))
```

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

[11]:
```
file_root_test = 'VOCdevkit_2007/VOC2007test/JPEGImages/'
annotation_file_test = 'voc2007test.txt'

test_dataset =␣
 ↪VocDetectorDataset(root_img_dir=file_root_test,dataset_file=annotation_file_test,train=Fals
 ↪S=S)
test_loader =␣
 ↪DataLoader(test_dataset,batch_size=batch_size,shuffle=False,num_workers=0)
print('Loaded %d test images' % len(test_dataset))
```

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

1.5 Train detector

[19]:
```
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(),␣
 ↪total_loss / (i+1)))

    # 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)
        print(test_loss)
    if best_test_loss > test_loss:
        best_test_loss = test_loss
        torch.save(net.state_dict(),'best_detector.pth')
        torch.save({'epoch': epoch, 'model_state_dict': net.state_dict(),␣
↪'optimizer_state_dict': optimizer.state_dict(), 'loss': best_total_loss,}, '.
↪/best_detector_list.pth')

    torch.save(net.state_dict(),'detector.pth')
    torch.save({'epoch': epoch, 'model_state_dict': net.state_dict(),␣
↪'optimizer_state_dict': optimizer.state_dict(), 'loss': test_loss,}, './
↪detector_list.pth')
```

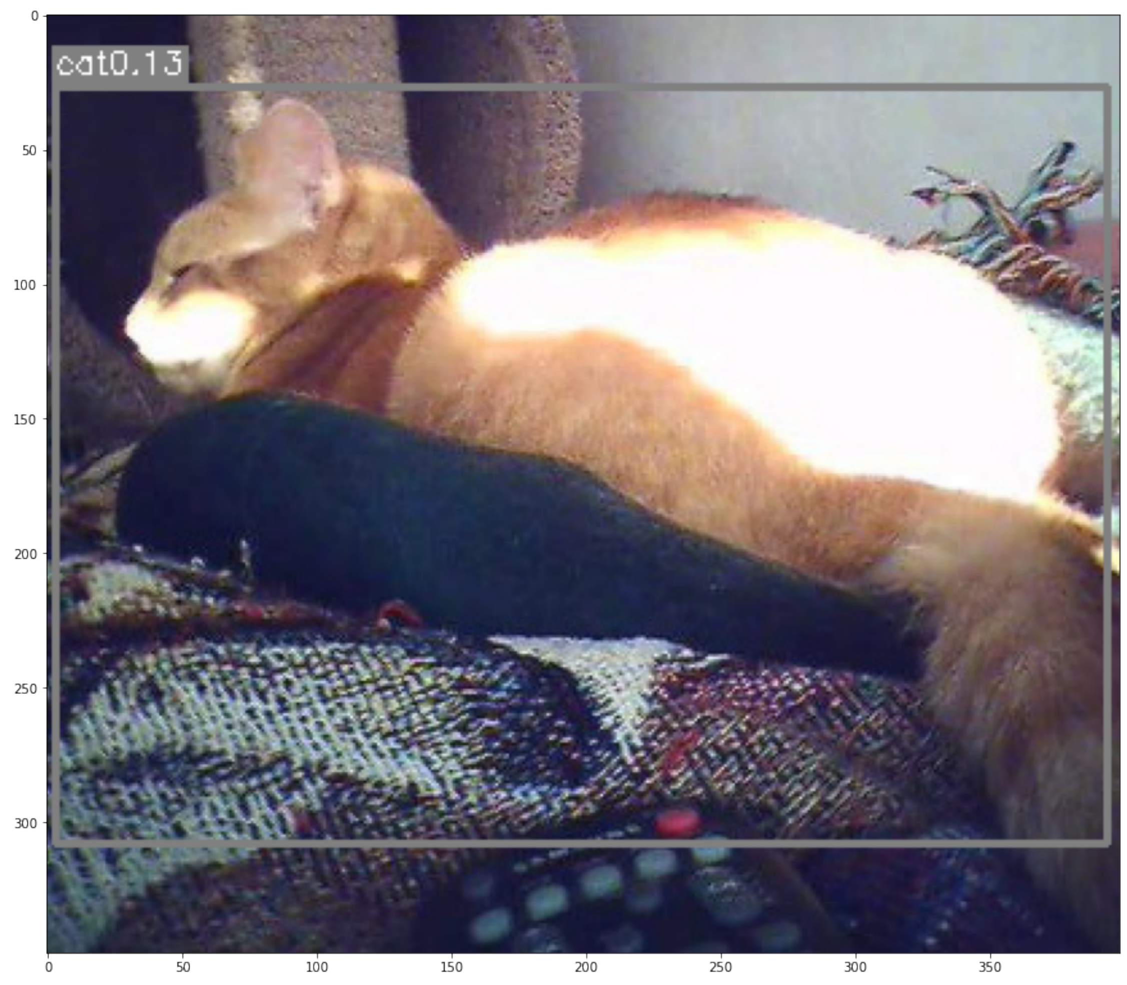
```
Starting epoch 70 / 70
Learning Rate for this epoch: 1e-05
Epoch [70/70], Iter [5/209] Loss: 1.7261, average_loss: 1.6416
Epoch [70/70], Iter [10/209] Loss: 1.6121, average_loss: 1.6269
Epoch [70/70], Iter [15/209] Loss: 1.1951, average_loss: 1.6363
Epoch [70/70], Iter [20/209] Loss: 2.0298, average_loss: 1.6720
Epoch [70/70], Iter [25/209] Loss: 1.8210, average_loss: 1.6957
Epoch [70/70], Iter [30/209] Loss: 1.3446, average_loss: 1.6887
Epoch [70/70], Iter [35/209] Loss: 1.9565, average_loss: 1.7009
Epoch [70/70], Iter [40/209] Loss: 1.4003, average_loss: 1.6963
Epoch [70/70], Iter [45/209] Loss: 1.6395, average_loss: 1.6808
Epoch [70/70], Iter [50/209] Loss: 1.6097, average_loss: 1.7035
Epoch [70/70], Iter [55/209] Loss: 1.6433, average_loss: 1.7034
Epoch [70/70], Iter [60/209] Loss: 1.8897, average_loss: 1.7054
Epoch [70/70], Iter [65/209] Loss: 1.3417, average_loss: 1.6921
Epoch [70/70], Iter [70/209] Loss: 1.6253, average_loss: 1.6968
Epoch [70/70], Iter [75/209] Loss: 1.5387, average_loss: 1.6986
Epoch [70/70], Iter [80/209] Loss: 1.9778, average_loss: 1.6940
Epoch [70/70], Iter [85/209] Loss: 1.4638, average_loss: 1.6888
Epoch [70/70], Iter [90/209] Loss: 1.8020, average_loss: 1.6858
Epoch [70/70], Iter [95/209] Loss: 1.4771, average_loss: 1.6807
Epoch [70/70], Iter [100/209] Loss: 2.0288, average_loss: 1.6780
Epoch [70/70], Iter [105/209] Loss: 1.6067, average_loss: 1.6878
Epoch [70/70], Iter [110/209] Loss: 1.4837, average_loss: 1.6984
Epoch [70/70], Iter [115/209] Loss: 1.6374, average_loss: 1.6933
Epoch [70/70], Iter [120/209] Loss: 2.1789, average_loss: 1.7144
Epoch [70/70], Iter [125/209] Loss: 1.8208, average_loss: 1.7099
Epoch [70/70], Iter [130/209] Loss: 1.9870, average_loss: 1.7153
Epoch [70/70], Iter [135/209] Loss: 1.3868, average_loss: 1.7071
Epoch [70/70], Iter [140/209] Loss: 1.2941, average_loss: 1.7115
Epoch [70/70], Iter [145/209] Loss: 1.6284, average_loss: 1.7095
Epoch [70/70], Iter [150/209] Loss: 1.7527, average_loss: 1.7147
Epoch [70/70], Iter [155/209] Loss: 2.0790, average_loss: 1.7161
Epoch [70/70], Iter [160/209] Loss: 1.4729, average_loss: 1.7211
Epoch [70/70], Iter [165/209] Loss: 1.4907, average_loss: 1.7190
Epoch [70/70], Iter [170/209] Loss: 2.0361, average_loss: 1.7167
Epoch [70/70], Iter [175/209] Loss: 1.3378, average_loss: 1.7165
Epoch [70/70], Iter [180/209] Loss: 1.6205, average_loss: 1.7167
Epoch [70/70], Iter [185/209] Loss: 2.0279, average_loss: 1.7176
Epoch [70/70], Iter [190/209] Loss: 1.8029, average_loss: 1.7227
Epoch [70/70], Iter [195/209] Loss: 1.7669, average_loss: 1.7248
Epoch [70/70], Iter [200/209] Loss: 1.6686, average_loss: 1.7272
Epoch [70/70], Iter [205/209] Loss: 2.4803, average_loss: 1.7266
2.6993180691907948
```

2 View example predictions

[24]:
```
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[0], p1[1] + text_size[1]),
                  color, -1)
    cv2.putText(image, label, (p1[0], p1[1] + baseline), cv2.
 ↪FONT_HERSHEY_SIMPLEX, 0.4, (255, 255, 255), 1, 8)

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

```
predicting...
```

[24]:
```
<matplotlib.image.AxesImage at 0x7feacb1b7e90>
```



2.1 Evaluate on Test

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

[15]:
```
test_aps = evaluate(net, test_dataset_file=annotation_file_test)
```

```
---Evaluate model on test samples---

100%|| 4950/4950 [05:49<00:00, 14.14it/s]

---class aeroplane ap 0.47323943134496954---
---class bicycle ap 0.6118454239884109---
---class bird ap 0.49220495515745577---
---class boat ap 0.275612365401917---
---class bottle ap 0.22744575269261746---
---class bus ap 0.6204233702754385---
---class car ap 0.6821270658273266---
---class cat ap 0.6913554521022629---
```

```
---class chair ap 0.27465563400474063---
---class cow ap 0.4998506892526715---
---class diningtable ap 0.35844816011964276---
---class dog ap 0.6460726121107241---
---class horse ap 0.6748813489465739---
---class motorbike ap 0.5433698269917407---
---class person ap 0.5350005936463598---
---class pottedplant ap 0.1812271113345213---
---class sheep ap 0.5118898916716628---
---class sofa ap 0.43936921638433359---
---class train ap 0.6474850558761931---
---class tvmonitor ap 0.47592137951110236---
---map 0.49312126683203344---
```

[16]:
```
output_submission_csv('my_solution.csv', test_aps)
```