R E P O R T

Submission : Object Detection

학 번: 2017310367

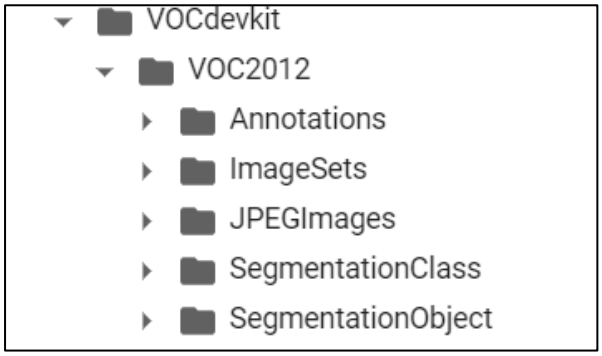
이 름: 정 다솔

제출일: 2021-6-6

성균관대학교
SUNGKYUNKWAN UNIVERSITY

Preprocessing the data

데이터는 Pascal VOC 2012를 사용하였다.



데이터의 구조는 다음과 같다. 이 중 JPEGImages와 Annotation 만을 사용하였다. 가장 먼저, 데이터가 어떠한 구성을 가지고 있는지 확인한다.

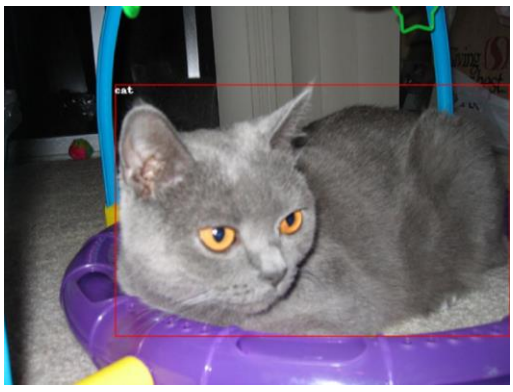
JPEGImages는 jpg 확장자로, 특정 이미지이다. Annotation는 xml로 이미지에 대한 정보를 담고 있다. Xml을 분석하여 주요 feature를 뽑아보면 다음과 같다.

```
plt.figure(figsize=(10,10))
plt.imshow(image)
plt.show()
plt.close()
```

```
Image properties
width : 375
height : 500
channels : 3

Objects Description
class : person
xmin : 130
ymin : 45.70000076293945
xmax : 375
ymax : 470
```

이러한 JPEGImages와 Annotation을 통해 다음과 같은 Object Detection 결과를 볼 수 있다.



데이터 학습을 위해 YOLO를 사용하기로 결정하고, 이를 위해 데이터 전처리에서는 데이터셋을 YOLO에서 사용하는 format으로 바꿔주는 convert2Yolo code를 참고하기로 했다.

일단, 가장 먼저 voc.names 파일이 필요하다. 이 파일 안에는 이미지의 class 종류가 담겨있어야 한다. 이에 for문을 돌며 image들의 name을 모두 꺼내 voc.names 파일에 저장하였다. 20개의 class를 가지고 있는 것을 확인할 수 있다. 이 voc.names 파일은 데이터셋과 함께 저장하여 VOC 데이터셋을 YOLO format으로 바꾸는 데에 사용한다.

준비가 다 되었다면 VOC 데이터셋을 YOLO format으로 바꾼다. pytorch의 dataset은 다음과 같다. 가장 먼저 init을 통해 변수들을 할당한다.

```
def __init__(self, root, class_path, transform, train=True, resize=448):
    self.root = root
    self.transform = transform
    self.train = train
    self.resize_factor = resize
    self.class_path = class_path
    self.data = self.cvtData()
    print(len(self.data))
```

그 후 데이터를 parsing 한다. 구현은 convert2Yolo를 참고했다. VOC data를 parsing한 후 yolo의 label 포맷으로 변경한다.

```
def cvtData(self):

    result = []
    voc = yoloVOC()

    yolo = yoloYOLO(os.path.abspath(self.class_path))
    flag, self.dict_data =voc.parse(os.path.join(self.root, self.LABEL_FOLDER))

    if flag == True:
        flag, data =yolo.generate(self.dict_data)

        keys = list(data.keys())
        keys = sorted(keys, key=lambda key: int(key.split("_")[-1]))

        for key in keys:
            contents = list(filter(None, data[key].split("\n")))
            target = []
            for i in range(len(contents)):
                tmp = contents[i]
                tmp = tmp.split(" ")
                for j in range(len(tmp)):
                    tmp[j] = float(tmp[j])
                target.append(tmp)

            result.append([os.path.join(self.root, self.IMAGE_FOLDER, "".join([key, self.IMG_EXTENSIONS])) : target])

        return result
```

그 후 getitem을 통해 result값을 하나하나 가져오며 image를 resize한 후 transform해준다. YOLO 논문에서는 이미지 크기로 448를 사용하지만 코랩 환경

문제로 112로 줄여 지정해주었다. 데이터를 학습하기 전에는 Colab pro를 사용하지 않아, 여러 부분에서 타협을 했다.

```
def __getitem__(self, index):

    key = list(self.data[index].keys())[0]
    img = Image.open(key).convert('RGB')
    current_shape = img.size
    img = img.resize((self.resize_factor, self.resize_factor))
    target = self.data[index][key]
    img = self.transform(img)

    return img, target, current_shape
```

그렇게 데이터셋을 VOC에서 YOLO 포맷으로 만든다.

```
transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

train_dataset = VOC(root='/content/data/VOCdevkit/VOC2012/',
                    class_path = '/content/data/VOCdevkit/VOC2012/voc.names',
                    transform=transform)
```

YOLO는 입력 이미지(input images)를 S x S grid로 나눈다. 따라서 output tensor의 사이즈는 S x S x (바운딩박스 * 5 + confidence score)이다. Loss를 계산하고 학습하려면 label 또한 output tensor와 동일한 사이즈로 만들어야 한다.

이에 detection_collate 함수로 batch를 받아, label의 사이즈를 맞춰준다.

```
def detection_collate(batch):
    targets = []
    imgs = []
    sizes = []

    for sample in batch:
        imgs.append(sample[0])
        sizes.append(sample[2])

        np_label = np.zeros((7, 7, 6), dtype=np.float32)
        for object in sample[1]:
            objectness = 1
            classes = object[0]
            x_ratio = object[1]
            y_ratio = object[2]
            w_ratio = object[3]
            h_ratio = object[4]

            scale_factor = (1 / 7)
            grid_x_index = int(x_ratio // scale_factor)
            grid_y_index = int(y_ratio // scale_factor)
            x_offset = (x_ratio / scale_factor) - grid_x_index
            y_offset = (y_ratio / scale_factor) - grid_y_index

            np_label[grid_x_index][grid_y_index] = np.array([objectness, x_offset, y_offset, w_ratio, h_ratio, classes])
```

그 후 train 데이터와 validate 데이터를 8:2의 비율로 나누어 주고, Dataloader를 통해 batch를 만들어준다.

```
[12] tr_idx, val_idx = train_test_split(list(range(len(train_dataset))), test_size=0.2, random_state=SEED)
```

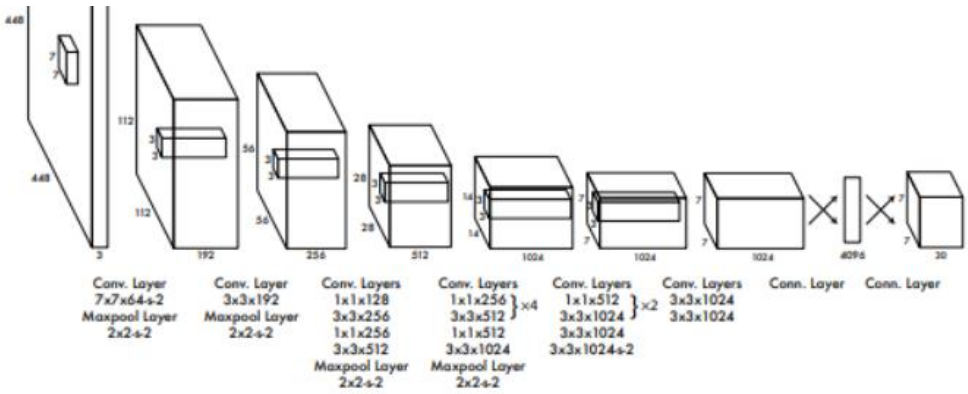
```
[13] tr_sampler = SubsetRandomSampler(tr_idx)
     val_sampler = SubsetRandomSampler(val_idx)
```

```
[14] import torch.utils.data as data

     train_loader = data.DataLoader(
         dataset=train_dataset,
         batch_size=3,
         sampler=tr_sampler,
         collate_fn=detection_collate,
         num_workers = 0)
     val_loader = data.DataLoader(
         dataset=train_dataset,
         batch_size=3,
         sampler=val_sampler,
         collate_fn=detection_collate,
         num_workers = 0)
```

Model architecture

사용한 모델은 YOLO이다. 다만 코랩 환경과 시간의 문제로 layer를 간소화하여 완벽한 YOLO라고 보기에는 어렵다. YOLO는 real-time에 약한 기존의 Object Detection 모델의 단점을 개선한 모델이다. 성능은 조금 떨어지더라도 빠른 fps를 가져 동영상에서의 Object Detection에 자주 사용된다. 이번 프로젝트는 동영상이 아니지만, 동영상 Object Detection에 관심이 많았기 때문에 이 모델을 선택했다. YOLO는 localization작업 후에 classification작업이 수행되는 다른 모델과는 달리, localization작업과 classification작업이 동시에 수행되어 빠른 fps를 가진다.
YOLO 또한 다른 Object Detection과 같이 CNN을 통해 feature map(grid cell)를 생성한다. YOLO 논문에서 Grid cell의 사이즈를 7로 설정해주었으므로 그대로 따랐다.



```
self.layer6 = nn.Sequential(
    nn.Conv2d(256, 512, kernel_size=3, stride=1, padding=0),
    nn.BatchNorm2d(512, momentum=0.01),
    nn.LeakyReLU(),
    nn.MaxPool2d(kernel_size=2, stride=2))
self.layer7 = nn.Sequential(
    nn.Conv2d(512, 512, kernel_size=1, stride=1, padding=0),
    nn.BatchNorm2d(512, momentum=0.01),
    nn.LeakyReLU())
self.layer8 = nn.Sequential(
    nn.Conv2d(512, 1024, kernel_size=3, stride=1, padding=1),
    nn.BatchNorm2d(1024, momentum=0.01),
    nn.LeakyReLU(),
    nn.MaxPool2d(kernel_size=2, stride=2))
self.layer9 = nn.Sequential(
    nn.Conv2d(1024, 1024, kernel_size=3, stride=1, padding=1),
    nn.BatchNorm2d(1024, momentum=0.01),
    nn.LeakyReLU())
```

```
self.fc1 = nn.Sequential(
    nn.Linear(3 * 3 * 1024, 4096),
    nn.LeakyReLU(),
    nn.Dropout(self.dropout_prop)
)

self.fc2 = nn.Sequential(
    nn.Linear(4096, 7 * 7 * ((5) + self.num_classes))
)
```

```
      nn.Conv2d(128, 256, kernel_size=3, stride=1, padding=1),
      nn.BatchNorm2d(256, momentum=0.01),
      nn.LeakyReLU())
  self.layer5 = nn.Sequential(
      nn.Conv2d(256, 256, kernel_size=1, stride=1, padding=1),
      nn.BatchNorm2d(256, momentum=0.01),
      nn.LeakyReLU())
```

YOLO 논문과 같이 7x7의 Grid cell로 시작하였다. YOLO는 총 24개의 Convolutional layers과 2개의 Fully connected layers으로 구성되어 있다. 그러나 이번 프로젝트에서는 시간과 환경의 제약으로 인해 YOLO에서 반복되는 구조를 제외하고, Fast YOLO와 유사하게 9개의 Convolutional layers만을 사용하였다. 또한 YOLO에서는 1x1 Reduction layer와 3x3 Convolutional layers을 결합하여 사용한다. 학습의 결과로 7x7x30(7x7x(2x5+20))의 결과를 갖는 데이터를 얻을 수 있다. 다만 이 프로젝트에서는 YOLO 논문처럼 2개의 bounding box가 아닌 1개의 bounding box만을 생성했기 때문에 데이터의 사이즈는 7x7x25(7x7x(5+20))이다. output 데이터에는 bounding box에 대한 정보와 bounding box안에 object가 포함되어 있을 확률인 confidence score 정보가 담겨있다. Output 데이터를 이용하여 bounding box에서 가장 높은 confidence score을 갖는 class를 선택한다. 만약 score값이 0이 넘는다면 object로 간주한다.

$$
\begin{aligned}
&\lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left[(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 \right] \\
&+ \lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left[\left(\sqrt{w_i} - \sqrt{\hat{w}_i} \right)^2 + \left(\sqrt{h_i} - \sqrt{\hat{h}_i} \right)^2 \right] \\
&+ \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left(C_i - \hat{C}_i \right)^2 \\
&+ \lambda_{\text{noobj}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{noobj}} \left(C_i - \hat{C}_i \right)^2 \\
&+ \sum_{i=0}^{S^2} \mathbb{1}_{i}^{\text{obj}} \sum_{c \in \text{classes}} (p_i(c) - \hat{p}_i(c))^2 \qquad (3)
\end{aligned}
$$

그 후 Loss function을 구한다. YOLO의 Loss를 계산하는 식은 다음과 같다. Output 데이터는 bounding box의 좌표값인 x, y, w, h와 object의 확률인 c를 포함한다. Loss는 이 값들을 슬라이싱하고 이를 정답과 비교하여 계산할 수 있다. One-hot encoding을 통해 각 label의 class별로 loss를 계산한다. 이렇게 각각 loss를 구한 후 total_loss로 합치고, 이를 반환한다. 조금 더 자세하게 설명하면,

1. Object가 존재하는 Grid cell의 x, y의 loss 계산
2. Object가 존재하는 Grid cell의 w, h의 loss 계산
3. Object가 존재하는 Grid cell의 c의 loss 계산
4. Object가 존재하지 않는 Grid cell의 c의 loss 계산
5. Class 확률 loss 계산

그 후 1~5까지 모두 더해준다.

```
no_obj_label = torch.neg(torch.add(obj_label, -1))

obj_coord_loss = lambda_coord * torch.sum(obj_label *
                 (torch.pow(x_offset_output - x_offset_label, 2) +
                  torch.pow(y_offset_output - y_offset_label, 2)))

obj_size_loss = lambda_coord * torch.sum(obj_label *
                (torch.pow(width_output - torch.sqrt(width_label), 2) +
                 torch.pow(height_output - torch.sqrt(height_label), 2)))

object_cls_map = obj_label.unsqueeze(-1)


for i in range(num_cls - 1):
    object_cls_map = torch.cat((object_cls_map, obj_label.unsqueeze(-1)), 3)

obj_class_loss = torch.sum(object_cls_map * torch.pow(class_output - class_label, 2))

no_obj_loss = lambda_noobj * torch.sum(no_obj_label * torch.pow(obj_output - obj_label, 2))
obj_loss = torch.sum(obj_label * torch.pow(obj_output - obj_label, 2))

total_loss = (obj_coord_loss + obj_size_loss + no_obj_loss + obj_loss + obj_class_loss)
total_loss = total_loss / b
```

lambda_coord는 x, y, h, w에 대한 loss과 다른 loss들의 균형을 맞춰주고, lambda_noobj는 object가 있는 Grid cell과 없는 Grid cell의 균형을 맞춰준다. One-hot encoding을 통해 1~5까지의 loss를 계산한다. 이렇게 각각 loss를 구한 후 loss들의 합을 반환한다.

Process and results of training

```
optimizer = torch.optim.Adam(model.parameters(), lr=0.01, weight_decay=1e-5)
scheduler = torch.optim.lr_scheduler.ExponentialLR(optimizer, gamma=0.95)
```

그 후 학습을 진행한다. Train data로 학습을 시킨 후 validation data로 검증을 하였다. Optimizer는 Adam을 사용하였으며 learning rate는 0.01을 주었다. Scheduler의 gamma값에는 0.95를 주었다.

```
for epoch in range(epochs):

    train_loss = 0
    for i, (images, labels, sizes) in enumerate(train_loader):

        images = images.to(device)
        labels = labels.to(device)
        optimizer.zero_grad()
        outputs = model(images).to(device)

        # Loss 계산
        train_loss = loss_yolo(outputs, labels, device.type)
        train_loss.backward()
        optimizer.step()


    valid_loss = 0
    with torch.no_grad():
        for i, (images, labels, sizes) in enumerate(val_loader):
            images = images.to(device)
            labels = labels.to(device)
            outputs = model(images).to(device)
            val_loss = loss_yolo(outputs, labels, device.type)

    print(
        'epoch: [{}/{}], lr: {}, train_loss: {:.4f}, val_loss: {:.4f}'
        .format(epoch + 1, epochs, ([param_group['lr'] for param_group in optimizer.param_groups])[0], train_loss, val_loss))
```

Batch size는 20이며, 총 20번의 epoch동안 학습하였다. 그 결과는 다음과 같다.

```
epoch: [1/20], lr: 0.01, train_loss: 20.1739, val_loss: 1157.5538
epoch: [2/20], lr: 0.01, train_loss: 13.7877, val_loss: 100.2710
epoch: [3/20], lr: 0.01, train_loss: 12.6009, val_loss: 10.3725
epoch: [4/20], lr: 0.01, train_loss: 9.3500, val_loss: 10.2538
epoch: [5/20], lr: 0.01, train_loss: 8.2326, val_loss: 30.1056
epoch: [6/20], lr: 0.01, train_loss: 4.9897, val_loss: 3.3702
epoch: [7/20], lr: 0.01, train_loss: 4.8530, val_loss: 192.6311
epoch: [8/20], lr: 0.01, train_loss: 9.6027, val_loss: 6.2644
epoch: [9/20], lr: 0.01, train_loss: 5.7163, val_loss: 5.1407
epoch: [10/20], lr: 0.01, train_loss: 4.7761, val_loss: 8.0794
epoch: [11/20], lr: 0.01, train_loss: 6.2644, val_loss: 5.4658
epoch: [12/20], lr: 0.01, train_loss: 61.1468, val_loss: 81.9163
epoch: [13/20], lr: 0.01, train_loss: 12.7003, val_loss: 28.2273
epoch: [14/20], lr: 0.01, train_loss: 7.5471, val_loss: 18.5594
epoch: [15/20], lr: 0.01, train_loss: 8.5272, val_loss: 8.4357
epoch: [16/20], lr: 0.01, train_loss: 15.1778, val_loss: 18.5068
epoch: [17/20], lr: 0.01, train_loss: 8.7969, val_loss: 9.9220
epoch: [18/20], lr: 0.01, train_loss: 6.1472, val_loss: 7.6366
epoch: [19/20], lr: 0.01, train_loss: 9.0222, val_loss: 5.9237
epoch: [20/20], lr: 0.01, train_loss: 5.1753, val_loss: 6.6699
```

Reference

1. You Only Look Once: Unified, Real-Time Object Detection/ Joseph Redmon/University of Washington

2. 모두의 연구소 - 모두를 위한 Object Detection(Object Detection for All) https://deepbaksuvision.github.io/Modu_ObjectDetection/