Artificial Intelligence Project Final Report

-2017313135 권동민-

Training

데이터 전처리

데이터 전처리는 다음과 같은 단계로 진행했다.

- Annotation 파싱
- 이미지 resizing
- Box resizing
- Custom dataset 만들기
- Train과 validation set 나누기
- 각각 Dataloader로 배치 합치기

Annotation 파싱

전체 데이터셋 이미지들에 따라, 각각 xml 파일을 순회하며

데이터를 파싱했다. 이때, 전역변수인 _classes, _idx2size, _idxmap, _class2idx를 설명하자면, 파싱을 하면서 새로운 class가 등장하면 _classes에 추가해주었다. _idx2size에는 각 index별 이미지의 size가 저장되며, _idxmap에는 각 인덱스 이미지별로 {class : boxsize}의 map을 생성한다. _idxmap[i]["class_name"]식으로 접근할 수 있다.

```
Now start parsing xml files
------------------------------------------------------------


image 0, image size: [486, 500, 3]
    class : person, box_size: [174, 101, 349, 351]
image 1000, image size: [500, 375, 3]
    class : person, box_size: [261, 1, 500, 234]
    class : sofa, box_size: [1, 1, 500, 375]
image 2000, image size: [500, 333, 3]
    class : diningtable, box_size: [299, 156, 500, 312]
    class : chair, box_size: [396, 162, 451, 269]
    class : sofa, box_size: [26, 134, 102, 204]
```

이처럼 1000번째 index에 있는 이미지마다 파싱한 결과를 출력하도록 해주었다. 결과를 확인해보면 각 이미지별로 size와 class별 box_offset이 저장돼 있는 것을 알 수 있다.

전체 이미지가 파싱된 후의 결과를 보면

```
---------- result parsing ----------

classes : ['person', 'aeroplane', 'tvmonitor', 'train', 'boat', 'dog', 'chair', 'bird', 'bicycle', 'bottle', 'sheep
iningtable', 'horse', 'motorbike', 'sofa', 'cow', 'car', 'cat', 'bus', 'pottedplant']

The number of classes : 20
The number of total img : 17125
```

총 20개의 클래스가 존재하며, 전체 데이터셋이 17125개임을 알 수 있다.

이미지 resizing

추후 model에서 pretrained model을 사용할 것이므로 resnet 공식 홈페이지를 참고하여 이미지 크기를 맞춰주고, transform을 설정해주었다.

이미지 크기는 우선 모든 이미지 데이터를 256,256으로 설정해준 뒤, transform에서 RandomCrop을 통해 최종적으로 224, 224 크기를 맞춰주었다.

```
Start Image Resizing
------------------------------------------------------------


[1000/17125] Image Resized
[2000/17125] Image Resized
[3000/17125] Image Resized
[4000/17125] Image Resized
[5000/17125] Image Resized
```

Box resizing

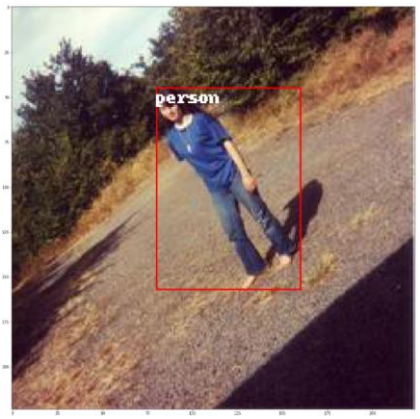
이미지를 resizing하면 문제가 되는 것이, 객체가 존재하는 Box의 offset값도 달라져야 한다는 것이다. 따라서 이미지의 가로, 세로가 줄어드는 비율에 따라 offset 값을 조정해주었다. Width_ratio = 224 / (원본 이미지의 가로size), height_ratio = 224 / (원본 이미지의 세로size) 이런 식으로 비율을 구해서, 각 가로와 세로에 해당하는 offset에 곱해주었다.

```
width_ratio = 224 / _idx2size[i][0]
height_ratio = 224 / _idx2size[i][1]
```

```
for key, box in _idx2map[i].items():

    _boxinfo = box
    _boxinfo[0] = int(_boxinfo[0] * width_ratio)
    _boxinfo[1] = int(_boxinfo[1] * height_ratio)
    _boxinfo[2] = int(_boxinfo[2] * width_ratio)
    _boxinfo[3] = int(_boxinfo[3] * height_ratio)
```

이후 테스트를 위해 resizing된 그림과 box를 출력해보면,

```
'''
image = Image.open('./JPEGImages_resized###2007_000027.jpg').convert("RGB")
draw = ImageDraw.Draw(image)

for key, box in _idx2map[0].items():
    draw.rectangle(((box[0], box[1]), (box[2], box[3])), outline="red")
    draw.text((box[0], box[1]), key)

plt.figure(figsize=(25,20))
plt.imshow(image)
plt.show()
plt.close()
'''
```



이처럼 제대로 박스 안에 객체가 들어가 있음을 확인할 수 있었다.

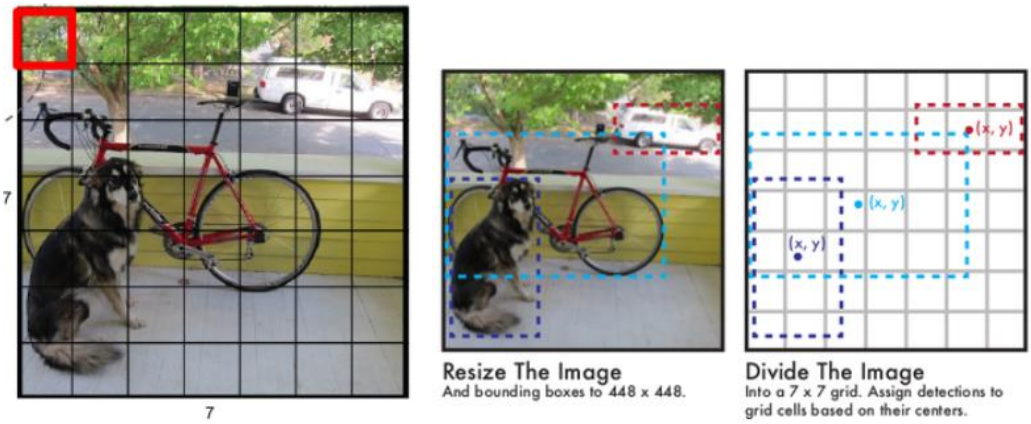
Custom dataset 만들기

여기에서 가장 중요한 것이, 각 인덱스별로 input tensor와 output tensor의 형태를 어떻게 해야 하는 지이다. 이후 model에서 학습을 진행할 때 어떻게 예측 값을 만드냐가 중요한 관건이다. 본인은 Yolo 모델을 기반으로 학습을 진행할 것이기에 input은 이미지, output은 7*7*25로 맞춰주었다.

Output에 대해 잠시 살펴보면, 현재 데이터가 각 이미지별로 클래스들과 해당하는 box offset을 가지고 있다. 따라서 이 정보를 바탕으로 output값을 만들어야 한다. 간단하게 구조를 살펴보면

7x7은 전체 이미지 224 x 224를 7x7의 cell들로 나누었을 때, 어느 지점에 이미지가 존재하는지를 나타내는 차원이다. 이후 각 grid cell마다 클래스 정보와 box 크기, 객체 존재여부 등의 정보를 담는데, 이게 25차원이 된다. (객체존재여부, x위치, y위치, 가로비율, 세로비율, 클래스 20개) 이때, 30차원이나 그 이상의 차원으로 진행할 수도 있는데, (이때에는 각 grid cell마다 2~3개의 box를 예측한다) 본인은 그냥 하나의 box만 예측하도록 진행하였다.

이때, 클래스는 one-hot encoding을 사용해서 표현한다. 그림으로 살펴보면



위와같이 그림을 7x7로 나누고, 객체 위치와 x, y, w, h값, class의 one-hot encoding 정보를 담는 output tensor를 만들고 이를 이미지 input과 함께 custom dataset의 __getitem__의 결과로 만들어주었다.

모든 데이터의 처리는 customDS의 __init__ 파트에서 진행하였다.

```
np_label = np.zeros((7, 7, 25), dtype=np.float32)

for key, box in idx2map[i].items():
    xoff = int((box[0]+box[2])/2)
    yoff = int((box[1]+box[3])/2)
    width = box[2] - box[0]
    height = box[3] - box[1]

    x_idx = xoff//32
    y_idx = yoff//32

    x_ratio = (xoff%32)/32
    y_ratio = (yoff%32)/32
```

우선 새로운 label을 정의하고, xoff와 yoff를 계산한다. 이때 (xoff, yoff)는 기존에 가지고 있던 (xmin, ymin), (xmax, ymax)의 중심 좌표이다. 이후, 박스의 width와 height를 계산했다.

이미지의 크기가 224x224이므로 7로 나누면 32가 되므로, 중심좌표가 존재하는 gridcell을 알아내기 위해 중심좌표인 (xoff, yoff)를 32로 나누어 각각의 index값을 계산해 주었다.(x, y) 또한 32로 나누었을 때의 나머지 offset을 통해 해당 grid cell 내에서의 비율을 계산했다. 즉 "중심좌표 (xoff, yoff)가 존재하는 grid cell 내에서의 x offset비율과 y offset 비율 w, h"를 구한 것이다.

```
class_onehot = np.zeros(20, dtype=np.float32)
class_onehot[_class2idx[key]] = 1
```

또한 class를 20개로 나눈 뒤 해당하는 클래스의 index로 매핑했다. 이처럼 one-hot encoding을 진행하고, 기존에 구한 값들과 하나로 합쳐 25차원의 tensor로 만들어주었다.

```
                np_label[x_idx][y_idx] = np.concatenate((np.array([1, x_ratio, y_ratio, width/224, height/224]),
class_onehot))
```

이렇게 만들어진 7x7x25 차원의 텐서를 y_data라는 list에 각 인덱스별로 넣어주었다. 즉 index별로 idx2files에는 annotation 파일 이름이, y_data는 각 annotation 파일별로 만든 output tensor가 들어있는 것이다. 이에 idx2files의 확장자만 jpg로 바꿔주면 결국 input과 output을 만들어낼 수 있는 것이다.

```
def __getitem__(self, index):
    target = self.y_data[index]
    x_file = self.idx2files[index].split(".")[0]+".jpg"

    image = Image.open(os.path.join(self.root, x_file)).convert('RGB')
    if self.transform is not None:
        image = self.transform(image)

    return image, target
```

Train과 validation set 나누기

```
split = int(np.floor(0.7 * len(totalDS)))
split2 = int(np.floor(0.9 * len(totalDS)))
trainIdx, validIdx, testIdx = idxList[:split], idxList[split:split2], idxList[split2:]

train_sampler = SubsetRandomSampler(trainIdx)
valid_sampler = SubsetRandomSampler(validIdx)
test_sampler = SubsetRandomSampler(testIdx)
```

데이터셋을 나누기 위해 SubsetRandomSampler를 사용했다. 여기에서는 그냥 전체 데이터의 70%를 training set으로, 20%를 validation set으로, 나머지 10%를 test data set으로 설정했다.

각각 Dataloader로 배치 합치기

```
train_loader = DataLoader(
    dataset=totalDS,
    batch_size=b_size,
    num_workers=0,
    sampler=train_sampler
)

val_loader = DataLoader(
    dataset=totalDS,
    batch_size=b_size,
    num_workers=0,
    sampler=valid_sampler
)

test_loader = DataLoader(
    dataset=totalDS,
    batch_size=b_size,
    num_workers=0,
    sampler=test_sampler
)

return train_loader, val_loader, test_loader
```

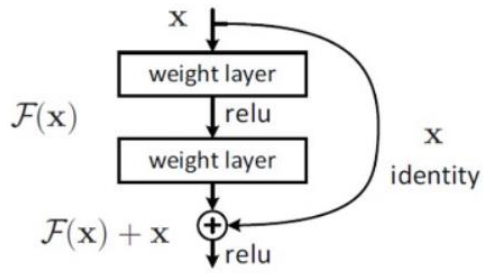
위와 같이 각 sampler를 train_loader와 val_loader에 설정하여 set을 나눴다.

Model 구현

Model은 Yolo의 구조를 참고해서 만들었다. Object detection은 크게 세 단계로 구분되는데, 이는 다음과 같다.

- Backbone
- Neck
- Head

각각에 대해 설명하자면, Backbone이 바로 feature extractor 부분이다. 이미지에서 특징을 추출하는 단계라 볼 수 있다. 이에 일반적으로 pretrained model을 사용한다. 본인은 Resnet을 사용했는데, 일반적인 CNN처럼 x -> H(x) = y가 되는 것이 아니라 x -> H(x) = x인 구조를 만드는 것이 목표인 네트워크이다. 즉 H(x) – x를 최소화한 구조로 이 값을 통해 학습하는 것을 residual learning이라 한다. Resnet은 skip connection을 사용해 gradient vanishing문제를 해결하였다. 이는



위 그림과 같은 구조로 F(x) + x -> H(x) = x가 되도록 하여 최소 1의 기울기를 갖도록 만든 것이다.

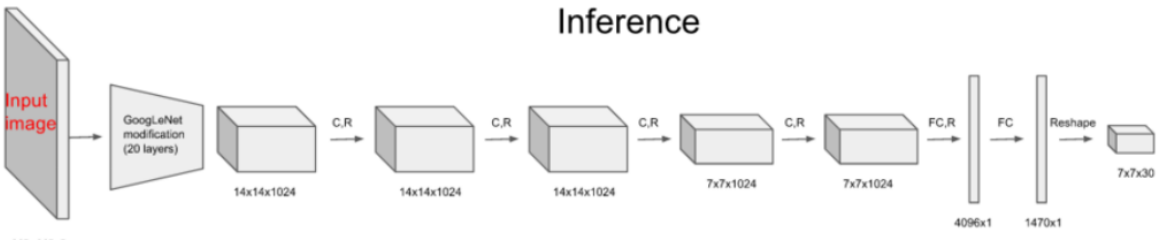
Resnet의 input은 위에서 언급한 것처럼 224, 224 크기의 이미지이고, transform 정규화에서 transforms.Normalize((0.485, 0.456, 0.406), (0.229, 0.224, 0.225)) 이와 같은 값을 가진다.

Neck은 이렇게 backbone들에서 만들어지는 feature을 각 stage별로 추출하는 부

분이다. 즉, augmentation 역할을 수행한다. 여기에는 SSD, FPN, STDN 등 여러가지 방식이 존재하는데, 각각 stage별로 feature map의 크기를 다르게 만든다는 특징이 있다. 이렇게 neck을 거쳐 학습하게 되면, 서로 다른 크기의 object를 검출하는데 효과적으로 작용한다.

마지막으로 Head부분은 실질적으로 detection을 수행하는 부분으로 object detector라고 불리기도 한다. 이 부분이 아까 언급했던 output값을 만드는 부분이다. Neck에서 뽑은 feature map들을 (7 x 7 x 25) tensor로 바꿔주는 역할을 한다.

필자가 구현한 구조는, 굉장히 단순하게 feature map을 뽑는 resnet과 detection을 수행하는 Fully Connected Layer로 구성된다. 일반적으로 resnet을 조금 변형하여 중간까지 사용하고 여기에 4~5개의 Convolution Layer를 중첩하여 FC에 연결하지만, 필자는 단순하게 추가적인 Convolution layer를 사용하지 않았고, resnet에서 나온 output을 바로 FC에 연결했다.



이는 일반적인 구조로, pretrained network이후 convolution layer를 더 쌓는 것을 알 수 있다.

하지만 학습이 너무 단순해질 우려에 Fully Connected layer를 하나 더 쌓았을 뿐이다. 이때 활성화 함수로는

Leaky ReLU를 사용했으며, Fully Connected Layer사이에 dropout값은 0.3으로 하였다.

Model 학습

Data loader, Loss function과 Optimizer를 설정해 준다.

```
train_loader, val_loader = makeValRandom()
model = Obj_Detect().to(device)
optimizer = optim.Adam(model.parameters(), lr=lrate)
```

우선 optimizer부터 설명하자면, Adam optimzer는 여러 가지 최적화 알고리즘중 하나로, 단순 gradient descent에서 Momentum과 Rmsprop이라는 개념을 적용시킨 것이다. Momentum은 가중치를 갱신할 때, 현재 batch만 반영하는 것이 아닌, 이전 batch의 학습결과도 반영하는 것이다. Rmsprop은 가중치 gradient를 누적시키는 것뿐만 아니라 최신의 gradient의 반영률을 높이는 방법이다.

이제 가장 중요한 loss function을 설명해보면, Yolo의 loss function은 다음 그림과 같이 구성돼있다.

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

각각 살펴보면, 총 5개로 구성이 되어 있다.

첫번째는 object가 존재할 때 box의 loss값이다. 대부분의 grid에 object가 존재하지 않기에, 해당 오브젝트의 값이 존재할 때 일반적으로 5를 곱해줘서 loss값을 증폭시켜준다.

```
box_loss = 5 * torch.sum(hasobj_target * (torch.pow(x_predict - x_target, 2) + torch.pow(y_predict - y_target, 2)))
```

두번째는 마찬가지로 object가 존재할 때 w, h의 값이다. 하지만 이때 최대값과

최소값의 차이를 줄이기 위해(표준편차를 줄인다) sqrt를 취해준다. 하지만 실제 코드에서는 target 값에만 sqrt를 취해주었다. (둘 다 sqrt를 취했을 때 학습이 제대로 이루어지지 않는다는 내용이 있었다.)

```
size_loss = 5 * torch.sum(hasobj_target * (torch.pow(w_predict - torch.sqrt(w_target), 2) + torch.pow(h_predict - torch.sqrt(h_target), 2)))
```

세번째는 confidence score라고도 불리며, object가 존재할 확률의 차이를 계산하는 것이다. IOU와 같은 개념이라고 보면 된다.

```
hasobj_loss = torch.sum(hasobj_target * torch.pow(hasobj_predict - hasobj_target, 2))
```

네번째는 세번째와 반대로 object가 존재하지 않을 확률의 차이를 계산하는데, 이번에는 반대로 0.5를 곱해 loss를 억제한다.

```
noobj_loss = 0.5 * torch.sum(nohasobj_target * torch.pow(hasobj_predict - hasobj_target, 2))
```

마지막으로는, 단순히 클래스 예측 값의 차이를 계산한 것이다.

```
cls_map = hasobj_target.unsqueeze(-1)

for i in range(19):
    cls_map = torch.cat((cls_map, hasobj_target.unsqueeze(-1)), 3)

class_loss = torch.sum(cls_map * torch.pow(class_predict - class_target, 2))
```

이렇게 optimizer와 loss function을 정의하고 학습을 진행하였다.

```
model.train()
for i, (img, target) in enumerate(train_loader):

    img, target = img.to(device), target.to(device)

    hypothesis = model(img)
    loss = loss_func(hypothesis, target)

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    trn_loss += loss.item()
    if((i % 10) == 0):
        print('Epoch [{}/{}], Step [{}/{}], Loss: {:.4f}'.format(epoch + 1, epochs, i + 1, total_step, loss.item()))
```

이때 각 step(mini batch) 마다 각각의 loss값을 출력해 주었다. 그리고 모든 mini-batch 학습이 끝나면, 각 epoch마다 validation loss를 계산해서 출력해주었다.

```
Epoch [18/20], Step [231/240], Loss: 3.3316

Evaluating validation loss in this epoch

    Valid - total_conf_score_loss: 64.91 total batchsize : 3425
      Epoch [18/20], Loss: 225.5492, avg_confidence_score_loss: 0.01895

Epoch [19/20], Step [1/240], Loss: 3.5133
Epoch [19/20], Step [11/240], Loss: 3.8379
```

이렇게 validation loss가 epoch마다 줄다가 어느 순간부터 다시 늘어난다면 overfitting의 확률이 높으므로, early stopping을 통해 학습을 중단해야 한다.

이를 모니터링 하기 위해 tensorboard를 사용하여 loss값의 변화를 확인해보았다.

이 그래프를 살펴보면 valid_loss가 대략 epoch 15까지는 꾸준히 감소하다가 이후 다시 증가세를 보이는 것을 알 수 있다. 따라서 epoch 15에서 early stopping을 해주는 것이 좋다.

이에, 코드상에 epoch을 20까지로 설정하여 학습했지만, epoch 15에서 모델을 저장했다.

Test

객체를 인식해서 box로 나타내는 Test코드는 구현하지 못해, loss값과 상정한 score로만 성능을 측정했습니다.

```
Start testing
-----------------------------------------------

TEST RESULT : Total_Loss: 120.1459, avg_confidence_score_loss: 0.01871
```

여기에서 avg_score_loss는 total_loss에 포함된 Loss값 중 세번째인 hasobj_loss값의 평균이다. 이 값이 적을수록 실제 값 이랑 가깝다고 할 수 있다.

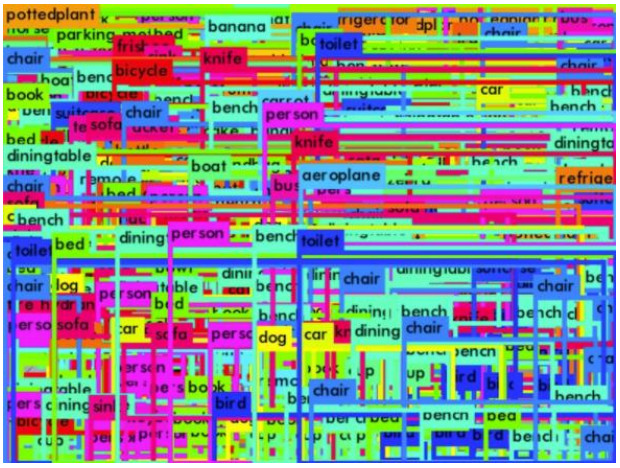
다만 어떤 식으로 모델을 통과한 image가 객체를 판별할 수 있는지 간단하게 설명하도록 하겠습니다.

우선 test model에 이미지를 넣기 위해 맨 앞에 배치 사이즈인 1을 추가해주어야 한다. 이후 model을 통과시킨다면 결과적으로 (1, 7, 7, 25) 차원을 가지는 tensor가 생길 것이다. 이 tensor를 가지고 객체를 어떻게 인식할 수 있는지가 가장 중요한 문제이다.

7x7은 grid cell 평면의 index를 담고 있으며, 나머지 25차원 값을 통해 box 위치를 판단할 수 있다. 우선 25차원 중 첫번째 차원은 sigmoid를 통과한 값으로, 객체가 해당 cell에 있는지 없는지 판단하는 기준이 된다.

```
out[:, :, :, 0] = torch.sigmoid(out[:, :, :, 0])
```

대부분의 값이 0이 아닐 것이므로 만약 threshold를 지정하지 않는다면, 아래와 같이 모든 box들이 출력될 것이다.



따라서 threshold값을 대략 0.2~0.3정도로 지정하여 box를 출력할 grid cell 위치를 정한다.

이후 grid cell에 출력될 박스의 크기와 정확한 위치를 결정해야 하는데, 이는 25차원중(1부터 시작이라고 한다면) 2~5 차원에 저장 돼있는 값을 읽어와서 데이터 전 처리 방법의 반대로 적용하면 된다. (가령 box의 width 값을 구하기 위해서는 224 / w처럼 구하면 된다. x값은 x_index * 32 + 32 * ratio를 하면 될 것이다.)

마지막으로 해당 box가 어떤 클래스인지 판단해야 하는데, 가장 기본적으로 max 값을 가지는 class를 선택하면 될 것이다.

사실 이렇게 box를 결정하는 것에는 많은 방법이 사용된다. Box의 중복을 제거하기 위한 NMS 등이 추가적으로 사용된다. (각 box의 IOU값에 따라 같은 위치에 있는 비율이 높다면 지워주는 식으로)