YOLO v1 논문 리뷰 및 모델 구현

KDT

송태인 고예성 박현식 서승수 조윤재 탁성대

CONTENTS

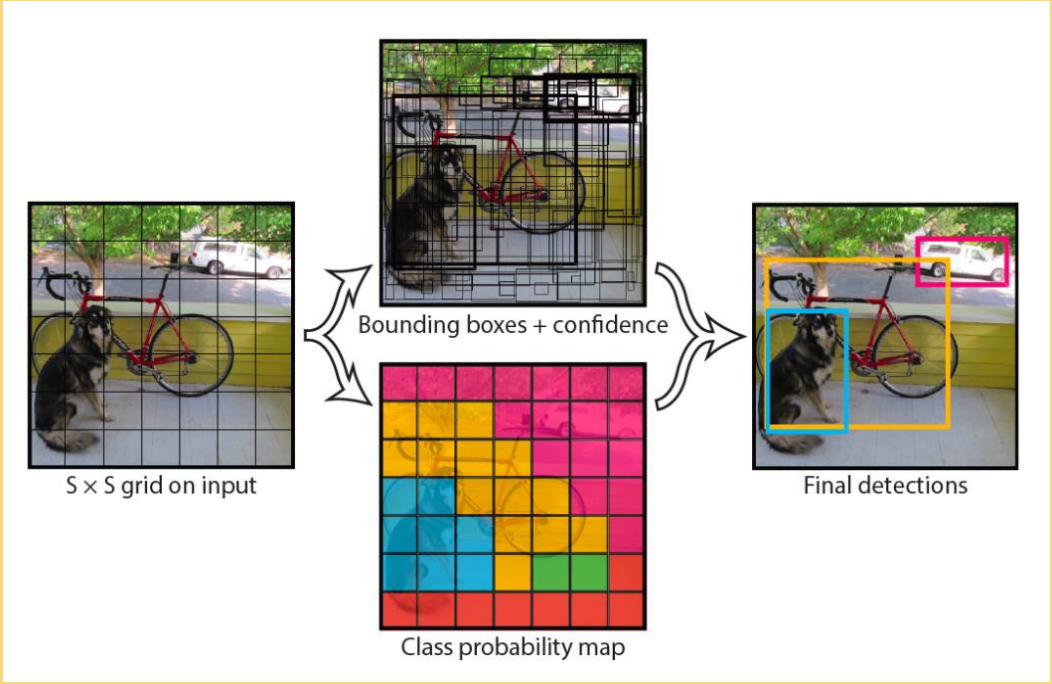
Part 01

YOLO v1 모델

모델 개요 및 특징

1. YOLO v1 설명

모델 개요

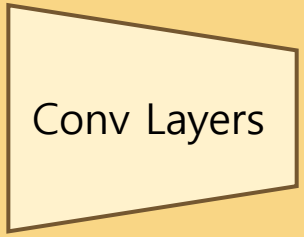


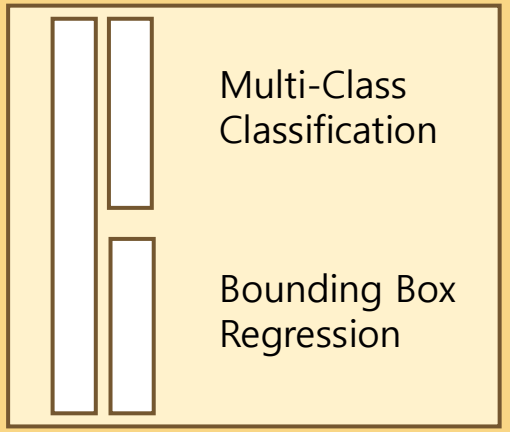
- YOLOv1는 2015년에 시작
- 이전에 존재하던 2-stage 방식 R-CNN, Fast R-CNN와 달리 YOLO는 1-stage방식으로 Localization과 Classification을 하나의 문제로 정의
- 이미지를 지정한 grid로 나누고, 각 grid cell이 한번에 bounding box와 class 정보라는 2가지 정답을 도출

1-1. One-stage vs Two-stage Detector

One-Stage Detector

- Loacalization & Classification 동시에 수행
- 이미지 내 모든 위치를 Object의 잠재영역으로 보고 각 후보 영역에 대해 Class를 예측



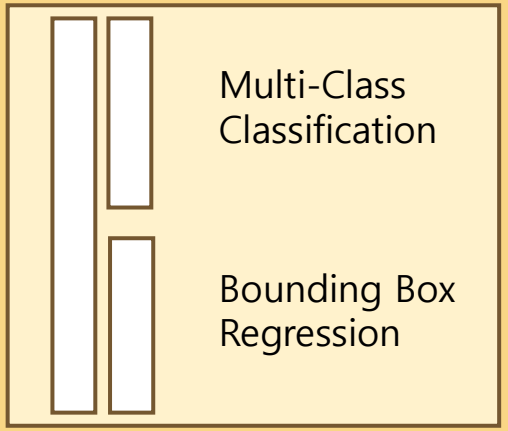


For each Grid cell

Two-Stage Detector

- Localization -> classification 순차적으로 수행
- 후보 Object 위치 제안 후 Object Class 예측

For each Proposed region



1. YOLO v1 설명

모델 장점

1. 속도가 매우 빠르다

real-time에서 다른 모델들에 비해

mAP가 두 배 이상 나옴

2. 예측 시 이미지 전체를 추론

이미지 전체 이용하여 class와 객체 출현에 대한

contextual information까지 이용

3. 객체의 일반화 할 수 있는 표현을 학습

특징 있는 부분만 추출하여 학습하는 것이 아닌

일반화 할 수 있는 표현도 학습할 수 있음

Real-Time Detectors	Train	mAP	FPS
100Hz DPM [31]	2007	16.0	100
30Hz DPM [31]	2007	26.1	30
Fast YOLO	2007+2012	52.7	**155**
YOLO	2007+2012	**63.4**	45
Less Than Real-Time			
Fastest DPM [38]	2007	30.4	15
R-CNN Minus R [20]	2007	53.5	6
Fast R-CNN [14]	2007+2012	70.0	0.5
Faster R-CNN VGG-16[28]	2007+2012	73.2	7
Faster R-CNN ZF [28]	2007+2012	62.1	18
YOLO VGG-16	2007+2012	66.4	21

Table 1: Real-Time Systems on PASCAL VOC 2007. Comparing the performance and speed of fast detectors. Fast YOLO is the fastest detector on record for PASCAL VOC detection and is still twice as accurate as any other real-time detector. YOLO is 10 mAP more accurate than the fast version while still well above real-time in speed.

Part 02

YOLO v1 논문 리뷰

You Look Only Once in 2016

3. YOLO v1 논문 리뷰

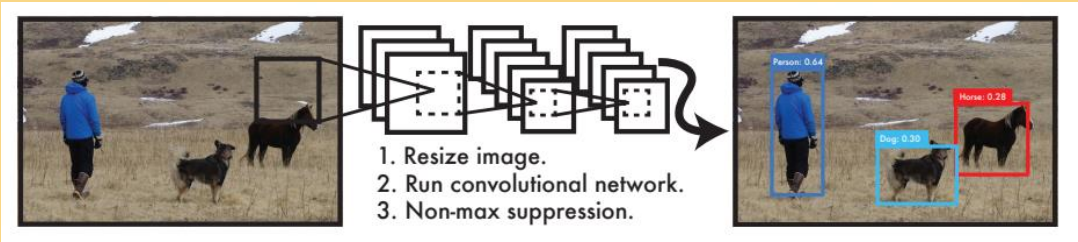
You Only Look Once:

Unified Real-Time Object Detection

CPVR, 2016

https://arxiv.org/pdf/1506.02640

3-1. Main Contribution



1) Object detection을 Regression problem으로 관점 전환

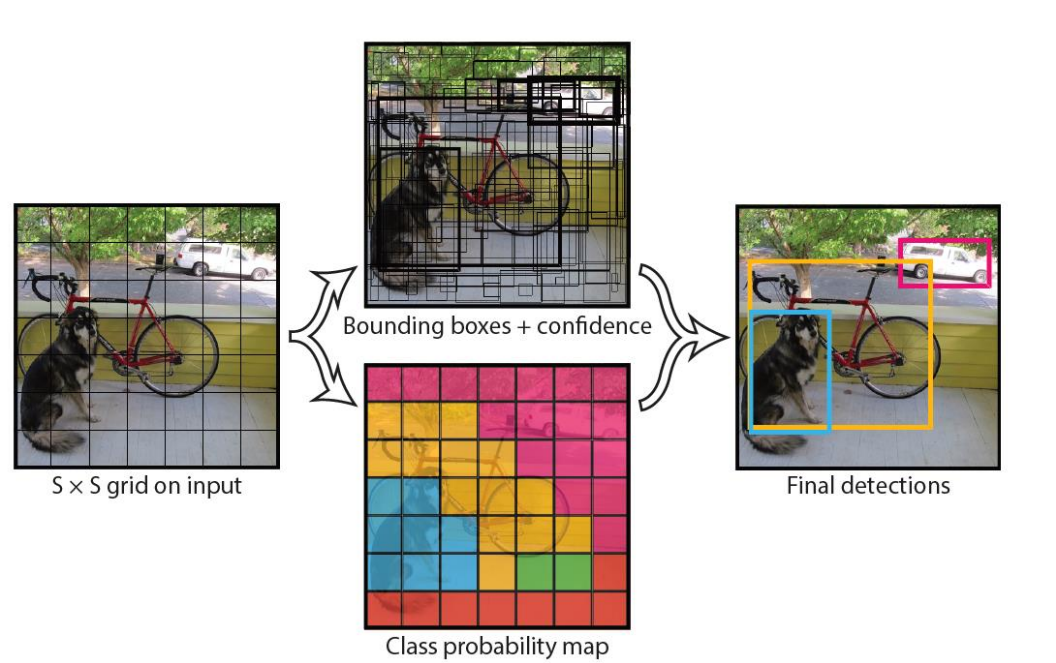
2) Unified Architecture

➞ 하나의 신경망으로 Classification & Localization 예측

3) 기존 DPM, RCNN 모델보다 속도 개선

4) 여러 도메인에서 Object detection 가능

3-2. Unified Detection



S : grid size
B : bounding boxes
C : Class probalilties

- YOLO v1은 별도의 region proposals를 사용하지 않고 전체 이미지를 입력하여 사용
- Object의 중심이 grid cell에 있으면 grid cell은 **object를 탐지**
- 각 grid cell은 B개의 **Bounding box와 confidence score를 예측**
- 각 Bounding box는 **5개의 예측 값**으로 구성

  [xc, yc, w, h, confidence]

  ※ Confidence Score은 예측된 bounding box와 진짜 박스 사이의 IoU

- 각 **grid cell은 class 확률을 예측**
- grid cell은 bounding box에도 영향을 미치고 class 확률에도 영향을 미침

  예측값 = S x S x(B * 5 + C)) 크기의 tensor

  ※ 논문에서는 7 x 7 x 30 tensor

- Non-max suppression을 거쳐 최종 bounding box를 select

3-3. Network Design - GoogleNet

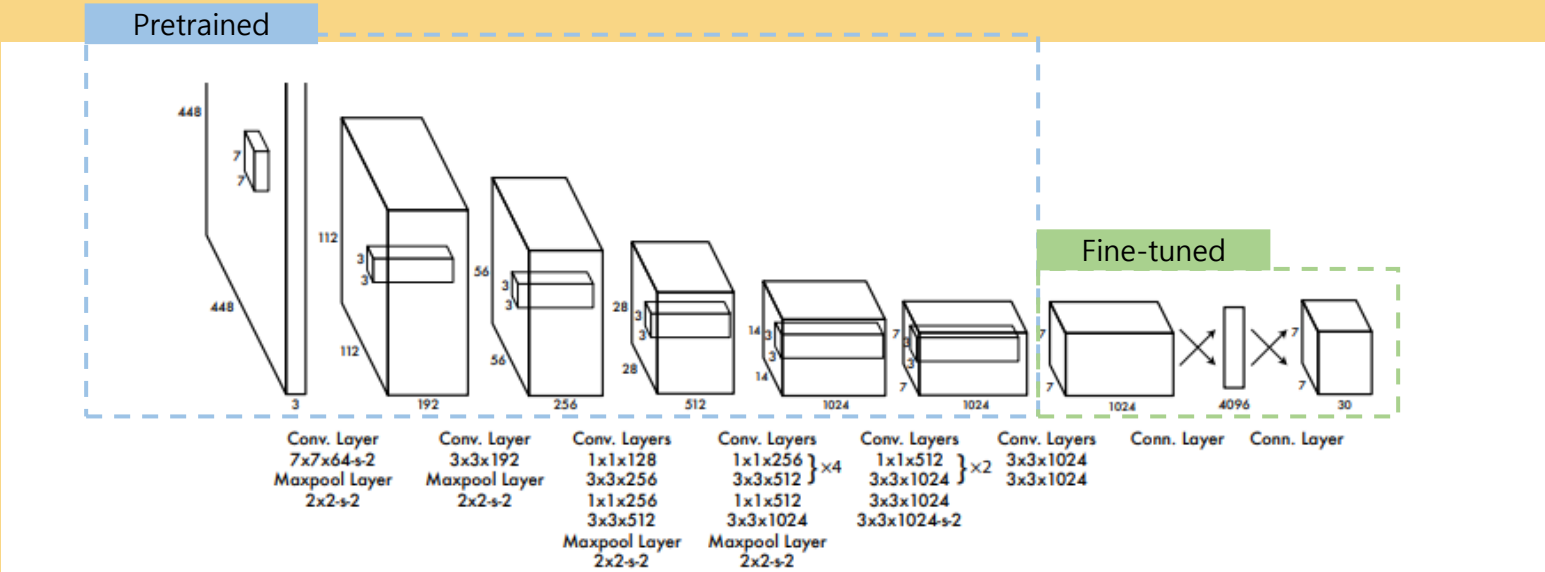


Figure 3: The Architecture. Our detection network has 24 convolutional layers followed by 2 fully connected layers. Alternating 1×1 convolutional layers reduce the features space from preceding layers. We pretrain the convolutional layers on the ImageNet classification task at half the resolution (224×224 input image) and then double the resolution for detection.

- 24개의 Convolution layer와 2개의 fully connected layer
- GoogleNet에 사용한 inception model 대신 1 x 1 convolution layer 사용

3-4. Training Stage – Loss Function

- λcoord 와 λnoobj는 손실에 대한 상수값(가중치)
- 통상적으로 λcoord =5(높은 가중치), λnoobj=0.5(낮은 가중치)를 사용

Localization Loss

$$\lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left[(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 \right]$$

$$+ \lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left[\left(\sqrt{w_i} - \sqrt{\hat{w}_i} \right)^2 + \left(\sqrt{h_i} - \sqrt{\hat{h}_i} \right)^2 \right]$$

- bounding box의 정답 좌표랑 예측 좌표 차이를 제곱하여 error를 계산함.
- 이 수식은 i번째 grid에서 j번째 객체를 포함한 bounding box를 의미 + 5배의 패널티를 부과
- 너비와 높이에 대한 error, 너비와 높이에도 5배의 패널티를 부여

Confidence Loss

$$+ \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left(C_i - \hat{C}_i \right)^2$$

$$+ \lambda_{\text{noobj}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{noobj}} \left(C_i - \hat{C}_i \right)^2$$

- 객체를 포함한 바운딩 박스에 대한 confidence error
- 객체를 포함하지 않은 바운딩 박스에 대한 confidence error

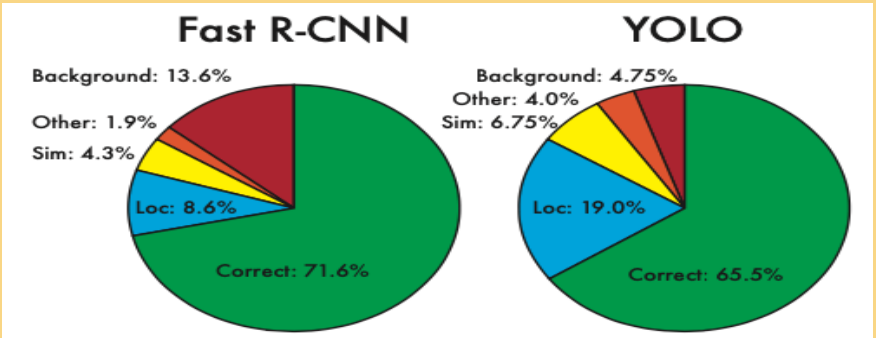
Classification Loss

$$+ \sum_{i=0}^{S^2} \mathbb{1}_{i}^{\text{obj}} \sum_{c \in \text{classes}} \left(p_i(c) - \hat{p}_i(c) \right)^2 \quad (3)$$

- p(c) 클래스 확률에 대한 classification error
- 객체를 포함한 바운딩박스에 대해서만 계산

3-5. 결과

Real-Time Detectors	Train	mAP	FPS
100Hz DPM [31]	2007	16.0	100
30Hz DPM [31]	2007	26.1	30
Fast YOLO	2007+2012	52.7	**155**
YOLO	2007+2012	**63.4**	45
Less Than Real-Time			
Fastest DPM [38]	2007	30.4	15
R-CNN Minus R [20]	2007	53.5	6
Fast R-CNN [14]	2007+2012	70.0	0.5
Faster R-CNN VGG-16[28]	2007+2012	73.2	7
Faster R-CNN ZF [28]	2007+2012	62.1	18
YOLO VGG-16	2007+2012	66.4	21



	mAP	Combined	Gain
Fast R-CNN	71.8	-	-
Fast R-CNN (2007 data)	**66.9**	72.4	.6
Fast R-CNN (VGG-M)	59.2	72.4	.6
Fast R-CNN (CaffeNet)	57.1	72.1	.3
YOLO	63.4	**75.0**	**3.2**

- Dataset: PASCAL VOC 2007
- 속도: Fast YOLO >> Yolo >> DPM, RCNN계열의 속도
- 성능: Faster-RCNN > Fast-RCNN > Yolo >> DPM

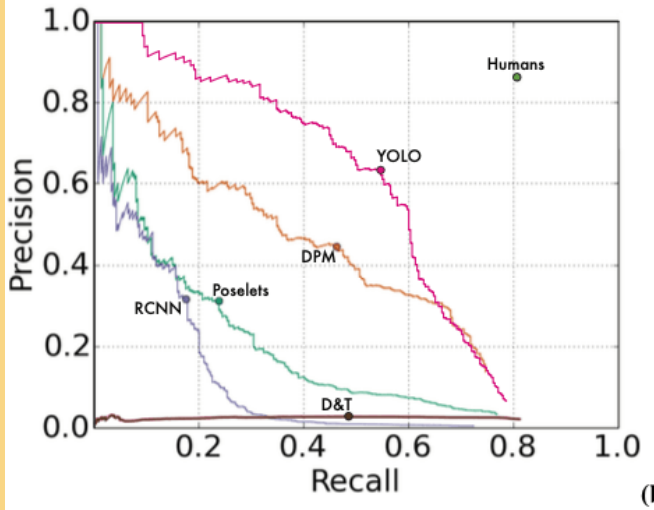
※ mAP : Mean Average Precision

- Fast RCNN 보다 Background Error는 감소

  ➞ false positive를 감소
- Fast RCNN + YOLO: mAP를 3.2% 향상

※ false positive : 배경에 물체가 없는데 있다고 하는 에러

3-5. 결과



(a) Picasso Dataset precision-recall curves.

	VOC 2007	Picasso		People-Art
	AP	AP	Best F_1	AP
YOLO	**59.2**	**53.3**	**0.590**	**45**
R-CNN	54.2	10.4	0.226	26
DPM	43.2	37.8	0.458	32
Poselets [2]	36.5	17.8	0.271	
D&T [4]	-	1.9	0.051	

(b) Quantitative results on the VOC 2007, Picasso, and People-Art Datasets. The Picasso Dataset evaluates on both AP and best F_1 score.

Figure 5: Generalization results on Picasso and People-Art datasets.

- Dataset: Picasso Dataset, People-Art Dataset
- Task: Person Detection

  (하나의 사람에 대해서만 탐지하므로 성능지표는 AP)
- R-CNN과 DPM 모델에 비해 Person Detection Task의 성능지표 폭 감소가 작음
- 다양한 도메인에서도 Detection 성능을 보임

Part 03

코드 구현

Bus & Truck Object Detection

4. YOLO v1 실습

실습 개요

- 데이터셋 - Bus & Trcuck classification 이미지 데이터
- 데이터 증강 – albumentations 라이브러리 이용
- 환경 – Google Colab, T4 GPU
- Resnet18 모델을 backbone으로 사용, Yolo v1 알고리즘을 도입
- 학습 에폭은 10으로 설정
- 테스트 데이터는 도로의 차량 동영상을 사용

4-1. Loss Function

```python
class YOLOv1Loss(nn.Module):
    def __init__(self, num_classes, S=7, B=2):
        super(YOLOv1Loss, self).__init__()
        self.num_classes = num_classes
        self.S = S  # Grid 사이즈
        self.B = B  # Bounding Box 개수

        self.lambda_coord = 5.0
        self.lambda_noobj = 0.5

        self.mse_loss = nn.MSELoss()
        self.bce_loss = nn.BCEWithLogitsLoss()
```

코드 설명

- 손실함수 클래스 구축
- S: grid cell 사이즈
- B: Bounding Box 개수
- Num_classes = 2(Bus & Truck)
- 가중치 설정

4-1. Loss Function

```python
def forward(self, predictions, targets):
    """
    predictions: 모델의 출력. (batch_size, S, S, (B * 5) + num_classes)
    targets: 실제 타겟. (batch_size, S, S, (B * 5) + num_classes)
    """

    target_bbox = targets[..., :self.B * 5]
    target_class = targets[..., self.B * 5:]

    pred_bbox = predictions[..., :self.B * 5]
    pred_class = predictions[..., self.B * 5:]

    target_conf = target_bbox[..., 4::5]
    target_xy = target_bbox[..., :2]
    target_wh = target_bbox[..., 2:4]

    pred_conf = pred_bbox[..., 4::5]
    pred_xy = pred_bbox[..., :2]
    pred_wh = pred_bbox[..., 2:4]

    loss_conf = self.bce_loss(pred_conf, target_conf)

    object_mask = target_conf > 0
    loss_obj = self.mse_loss(pred_xy[object_mask], target_xy[object_mask]) + \
               self.mse_loss(pred_wh[object_mask], target_wh[object_mask])

    no_object_mask = target_conf == 0
    loss_noobj = self.lambda_noobj * self.bce_loss(pred_conf[no_object_mask], target_conf[no_object_mask])

    loss_class = self.bce_loss(pred_class, target_class)

    total_loss = loss_conf + loss_obj + loss_noobj + loss_class
    return total_loss
```

① ② ③ ④ ⑤ ⑥ ⑦ ⑧

코드 설명

① 타겟의 차원 중 Bounding Box 정보와 클래스 정보를 나눠서 추출

② 모델의 출력도 동일하게 처리

③ Bounding Box 정보에서 confidence와 클래스 정보를 분리

④ Confidence loss 계산

⑤ Objectness loss 계산

(실제 객체가 있는 Bounding Box에 대해서만 계산)

⑥ No-objectness loss 계산

(실제 객체가 없는 Bounding Box에 대해서만 계산)

⑦ Class loss 계산

⑧ 전체 손실 계산

4-2. Model

```
resnet18 = torchvision.models.resnet18(pretrained=True)

backbone_layers = list(resnet18.children())[:-2]
backbone = nn.Sequential(*backbone_layers)
```

```
self.layer19 = nn.Sequential(
    nn.Conv2d(1024, 512, kernel_size=1, stride=1, padding=0),
    nn.BatchNorm2d(512, momentum=0.01),
    nn.LeakyReLU())
self.layer20 = nn.Sequential(
    nn.Conv2d(512, 1024, kernel_size=3, stride=1, padding=1),
    nn.BatchNorm2d(1024, momentum=0.01),
    nn.LeakyReLU())
self.layer21 = nn.Sequential(
    nn.Conv2d(1024, 1024, kernel_size=3, stride=1, padding=1),
    nn.BatchNorm2d(1024, momentum=0.01),
    nn.LeakyReLU())
self.layer22 = nn.Sequential(
    nn.Conv2d(1024, 1024, kernel_size=3, stride=2, padding=1),
    nn.BatchNorm2d(1024, momentum=0.01),
    nn.LeakyReLU())
# LAYER 6
self.layer23 = nn.Sequential(
    nn.Conv2d(1024, 1024, kernel_size=3, stride=1, padding=1),
    nn.BatchNorm2d(1024, momentum=0.01),
    nn.LeakyReLU())
self.layer24 = 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(7 * 7 * 1024, 4096),
    nn.LeakyReLU()
)
self.fc2 = nn.Sequential(
    nn.Linear(4096, 7 * 7 * ((5) + self.num_classes))
)
```

코드 설명

- Backbone: Resnet18 사용
- Convolutional Layer 1~24까지 적층과정을 거침
- 활성화함수: LeakyReLU() 사용
- 마지막 부분: Fully Connected 레이어로, 최종적으로 객체의 경계 상자와 클래스에 대한 예측을 수행

  fc1: 중간 Fully Connected Layer

  fc2: 최종 출력 생성 Layer
- 출력 차원은 (7 * 7 * ((5) + num_classes))로 설정되어, 각 그리드 셀마다 5개의 박스 정보와 클래스 예측이 이루어짐

※ num_classes는 2로 설정

4-2. Model

```
class YOLO(torch.nn.Module):
    def __init__(self, VGG16):
        super(YOLO, self).__init__()
        self.backbone = VGG16
```

```
self.conv = nn.Sequential(
    nn.Conv2d(in_channels = 512,out_channels = 1024, kernel_size = 3, padding = 1),
    nn.LeakyReLU(),
    nn.Conv2d(in_channels = 1024,out_channels = 1024, kernel_size = 3, padding = 1),
    nn.LeakyReLU(),
    nn.MaxPool2d(2),
    nn.Conv2d(in_channels = 1024,out_channels = 1024, kernel_size = 3, padding = 1),
    nn.LeakyReLU(),
    nn.Conv2d(in_channels = 1024,out_channels = 1024, kernel_size = 3, padding = 1),
    nn.LeakyReLU(),
    nn.Flatten()
)
self.linear = nn.Sequential(
    nn.Linear(7*7*1024, 4096),
    nn.LeakyReLU(),
    nn.Dropout(),
    nn.Linear(4096, 1470)
)

# 가중치 초기화
for m in self.conv.modules():
    if isinstance(m, nn.Conv2d) :
        nn.init.normal_(m.weight, mean=0, std=0.01)

for m in self.linear.modules():
  if isinstance(m, nn.Linear) :
    nn.init.normal_(m.weight, mean=0, std=0.01)
```

코드 설명

- Backbone: VGG16
- 입력채널 및 출력채널, 커널 및 패딩의 크기를 parameter로 받아 Convolutional Layer, Leaky ReLU를 번갈아가며 적층함
- 마지막 부분: Flatten으로 평탄화 과정을 거친 후,
- Convolutional 레이어와 Linear 레이어의 가중치를 초기화함.
- 초기화된 값은 평균이 0이고 표준 편차가 0.01인 정규 분포에서 추출함.

→ 가중치 초기화는 초기 학습율과 학습 과정의 안정성을 향상시키는 데 도움이 됨.

4-3. Train

```
BATCH_SIZE = 64
EPOCH = 10

Train_Dataset = YOLO_PASCAL_VOC(path2data, year='2007', image_set='train', download=True)
Test_Dataset = YOLO_PASCAL_VOC(path2data, year='2007', image_set='test', download=True)

data_loader = torch.utils.data.DataLoader(dataset=Train_Dataset,
                                          batch_size=BATCH_SIZE,
                                          shuffle=True,
                                          drop_last=True)

VGGNet = torch.hub.load('pytorch/vision:v0.10.0', 'vgg16', pretrained=True)

for i in range(len(VGGNet.features[:-1])) :
    if type(VGGNet.features[i]) == type(nn.Conv2d(64,64,3)) :
        VGGNet.features[i].weight.requires_grad = False
        VGGNet.features[i].bias.requires_grad = False
        VGGNet.features[i].padding = 1

YOLO_model =  YOLO(VGGNet.features[:-1]).to(device) # Create YOLO model
optimizer = torch.optim.SGD(YOLO_model.parameters(), lr = 0.01, momentum = 0.9, weight_decay=0.0005)

YOLO_model = train_YOLO(YOLO_model, yolo_multitask_loss, optimizer, EPOCH, data_loader, device)
```

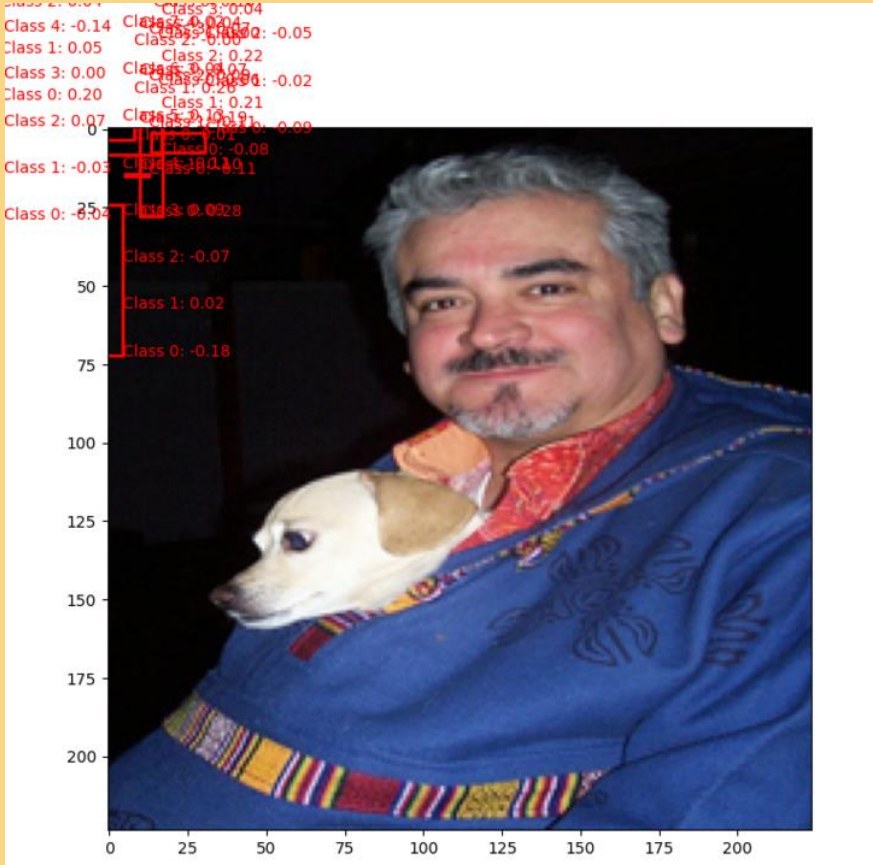
코드 설명

- PASCAL VOC2007 데이터셋 사용
- 사전학습된 VGG16 모델을 불러옴
- 마지막 layer에 Yolo v1 모델 적용
- Optimizer는 SGD를 이용하며 lr=0.01, momentum=0.9, weight_decay = 0.0005 등의 parameter를 이용함.
- Batch_size는 64, Epoch는 10으로 설정한 후 학습을 진행.

4-4. Test

```
from tqdm import tqdm
results = []
with torch.no_grad():
    for images, targets in tqdm(data_loader, desc="Testing", leave=False):
        images = images.to(device)
        detections = YOLO_model(images)
        results.extend(detections)
```

- Tqdm으로 테스트 시각화
- 테스트 데이터셋 사용
- 클래스에 대한 분류 문제에서 음수의 결과
- 바운딩박스 좌표 오류 발생



Part 04

결론

느낀점 & 개선사항

4. 결론

느낀점

- One-Stage의 검출방법으로 속도 면에서 Fast R-CNN보다 빠르다는 것을 논문을 통해 확인할 수 있었다.
- 논문에서는 Backbone을 VGG16로 사용했으나 현재는 더욱 성능이 좋은 모델이 많이 나와 있고, YOLO 버전이 계속 갱신 중이기 때문에 객체 검출이 더욱 쉬워졌다는 것을 확인할 수 있었다.
- 직접 구현을 시도해보았으나 input tensor와 target tensor의 차원 통일이 원활하게 이루어지지 못해 만족스러운 결과가 나오지 않아 아쉬웠다.
- 하지만 YOLO 모델의 기반이 되는 YOLO V1에 대해 심도 있게 공부해볼 수 있는 기회가 되어 좋았다.
- 구현을 위해 조사하는 과정에서 많은 개발자들이 Github에 업로드한 모델을 공부하며 다양한 방식으로 모델 구축이 가능하다는 사실을 알 수 있었다.

4. 결론

개선사항

- Input tensor와 Target tensor의 차원 통일 필요
- Dataset의 원활한 전처리 과정 필요
- Loss function의 정확한 개념 이해에 따른 total loss value의 산출 필요
- Numpy의 ndarray와 Pytorch의 tensor Data type 통일 필요

그럼 이만

감사합니다