TensorFlow, PyTorch를 이용한 YOLO 구현

CUAI 4기 컴퓨터비전 1팀
강민기(소프트웨어학부), 김민규(전자전기공학부), 김태윤(소프트웨어학부), 이승연(소프트웨어학부)

[요약] 컴퓨터비전은 딥러닝 알고리즘이 적용되며 빠른 속도로 증가하였다. 그 중 객체탐지 분야에서 딥러닝 알고리즘을 사용해 빠른 속도와 높은 정확도로 객체를 검출하는 YOLO는 최근까지도 개량되고 있고 다양한 객체탐지 모델에게 큰 영향을 미치고 있다. 우리는 이러한 YOLO를 읽는 것에 그치지 않고 텐서플로우와 파이토치를 이용해 직접 구현, 테스트하여 객체탐지 모델이 어떻게 만들어지는지 깊게 이해하는 시간을 가졌다.

1. 서 론

컴퓨터 비전(Computer Vision)은 딥러닝 알고리즘이 적용되며 빠른 속도로 증가하였다.
특히 CNN(Convolutional Neural Network)을 이용해 이미지에 있는 객체의 클래스를 분류(Classificatoin)하는 AlexNet[1]이 등장하며 딥러닝 알고리즘을 이용한 컴퓨터 비전의 발전이 더욱 가속되었다.
AlexNet의 등장 이후 CNN을 이용해 클래스 분류 외에도 객체탐지(Object Detection)를 수행하는 인공지능 모델을 만들려는 시도가 있었다. 수많은 논문들이 나왔고 그 중 객체가 존재할 영역을 먼저 검출한 후(stage 1), 검출한 영역의 위치와 영역에 있는 객체의 종류를 판단(stage 2)하는 2-Stage Detector인 R-CNN[2]이 추후 등장할 모델들에게 큰 영향을 미칠 정도로 좋은 성능을 보여줬다. 허나 R-CNN[2]은 성능 향상에 한계가 있고 처리 속도가 느리다는 단점이 있었고 이를 개선한 Fast R-CNN[3], Faster R-CNN[4]이 등장하며 속도와 성능이 개선되었지만 가장 개선된 모델이라고 할 수 있는 Faster R-CNN의 처리속도가 7 fps밖에 안되어 실시간 객체 탐지를 하기에는 부족했다. 그래서 속도를 올리기 위한 많은 시도가 있었으나 속도가 올라가면 성능이 그만큼 떨어지는 trade-off가 계속해서 발생했다. 그러나 사람들은 포기하지 않았고, 노력 끝에 Joseph Redmon, Santosh Divvala, Ross Girshick, Ali Farhadi가 높은 성능과 빠른 속도를 가진 YOLO[5]를 만들어냈다. YOLO는 2-Stage Detector와는 달리 이미지의 특성을 추출하는 Backbone과 추출된 정보를 가지고 객체의 위치와 정보를 알아내는 Head가 합쳐져 하나의 통합된 네트워크로 구성된 1-Stage Detector 모델이며 전체 이미지를 7 X 7 = 49개의 구역으로 나눈 뒤 각 구역당 2개의 Bounding box로 해당 구역에 **중심좌표가 있는 객체**를 탐지한다. Bounding box는 (x,y,w,h,confidence score)의 양식을 가지고 있는데 여기서 x, y는 객체의 중심 좌표, w, h는 객체의 너비와 높이, confidence score는 Bounding box가 객체를 얼마나 잘 표현했나 신뢰도를 나타내는 수치를 말한다. 그리고 각 구역에서 중점을 둔 객체가 어떤 종류인지 C개의 값을 출력해 종류별 예측 스코어를 표현한 class score가 있는데 C는 모델을 학습시키는데 사용한 데이터셋에 있는 객체의 종류를 말한다. 저자는 YOLO를 훈련시킬 때 PASCAL VOC 2007 dataset을 사용했는데 여기에 20종류의 객체가 있기 때문에 C=20으로 설정, 최종 출력값 사이즈가 (7,7,5X2+ 20)=(7,7,30)이 되었다. YOLO는 63.4의 mAP를 기록하면서 처리 속도가 45 fps나 되어 실시간으로 객체를 정확히 검출할 수 있는 모델이라고 할 수 있고 지금도 계속해서 개선되어 YOLO v5[6]까지 등장하였다. 그리고 YOLO의 객체검출 방식에 영향을 받은 수많은 모델들이 나오고 있으니 현재까지 객체검출 분야에 많은 영향을 끼치고 있다고 할 수 있다. 우리는 이러한 YOLO를 자세히 알아보는게 큰 도움이 될 것이라 판단해 이를 구현해보기로 했다. 쿠아이 컴퓨터비전 1팀의 멤버들이 각자 텐서플로우(TensorFlow)나 파이토치(PyTorch)를 사용해 모델을 구현했으며 코드는 이곳에서 확인할 수 있다.

2. 본 론

2-1. 텐서플로우(TensorFlow)를 이용한 구현

원래 YOLO는 Backbone으로 DarkNet[7]을 사용했으나 TensorFlow로 구현된 DarkNet을 찾을 수 없어 Backbone으로 VGG-16[8]과 DenseNet-121[10]을 사용했다. 왜냐하면 텐서플로우의 keras.applications 모듈에서 VGG-16과 DenseNet-121을 구현한 모델을 지원해줬기 때문이다. DenseNet-121을 사용한 YOLO는 입력 이미지의 사이즈를 유지했지만 VGG-16을 사용한 YOLO는 224 X 224로 줄어들었다. 그리고 YOLO에서는 데이터셋을 구성할 때 데이터 증강(Data Augmentation)을 이용했지만 우리는 데이터 증강을 할 때 이미지 속 객체들의 정보(Labeled data)도 같이 변환시켜야 하는 과정을 어떻게 구현해야 할지 방법을 찾지 못해 구현하지 못했다. 구현 단계는 데이터셋 전처리, 모델 설계, 훈련, 테스트로 이루어졌고 Backbone에 따라 구현 방식을 조금씩 다르게 했다.

1) 데이터셋 전처리

YOLO 본문에 나온 것과 같이 PASCAL VOC 2007 dataset을 사용했다. PASCAL은 객체탐지용

모델을 훈련시킬 때 사용되는 주요 데이터셋 중 하나로 train데이터셋과 test데이터셋으로 구성되어 있다.

VGG-16 : [11]의 방식을 참조했다. OpenCV 모듈을 이용해 각 데이터셋에 있는 사진들을 넘파이(NumPy) 배열 형태의 데이터로 가공하였고 이미지 속 객체들의 정보는 xml 파일이었기 때문에 xmltodict 모듈을 사용해 리스트 형식의 데이터로 변환했다. 이 과정을 통해 데이터셋에 존재하는 객체들을 알아냈고 이를 이용해 각 xml파일에 들어있는 객체 정보를 추출, 학습용 데이터셋과 테스트용 데이터셋을 텐서(Tensor)형식으로 만들었다. 검증용 데이터셋은 학습용 데이터셋을 최대한 보존하기 위해 테스트용 데이터셋의 일부를 떼서 만들었다.

DenseNet-121 : [12]의 방식을 참조했다. 데이터셋에 있는 객체들을 미리 알아내 리스트 형식으로 만들었다. 그리고 OepnCV 모듈로 이미지 파일을 넘파이 배열로 만들고 xml.etree.ElementTree모듈을 사용해 xml파일에서 이미지 속 객체들의 정보를 추출해 학습용, 훈련용 데이터셋을 만들었다. Input과 Output을 batch로 return해주는 generator클래스를 만들고 배치 사이즈 단위로 데이터를 x_train, y_trian, x_val, y_val에 넣어주었다.

2)모델 설계

텐서플로우의 tf.keras.Model 클래스로 모델을 선언 후 tf.keras.layers 모듈에 있는 CNN, FCN 등의 레이어들을 모델에 하나씩 추가하는 방식으로 구현했다.

VGG-16 : 우선 Tensorflow에 있는 VGG-16을 Backbone으로 불러온다. 이 때 VGG-16은 ImagetNet[9]으로 사전훈련된 모델이며 최종 출력값의 사이즈가 (14, 14)가 되는 CNN까지만 불러온다. 사전훈련된 VGG-16을 불러왔으면 여기에 YOLO에 나온내용대로 Head에 해당하는 CNN과 Fully Connected Neural Network(FCN)를 연결한 뒤 마지막에 reshape레이어를 추가해 논문에 나온 대로 (7, 7, 30) 사이즈의 텐서가 출력되게끔 만들었다. 로스 함수(Loss Function)는 텐서플로우에서 정한대로 (y_true, y_pred)를 받아 계산하는 함수로 만들었다. 우선 (y_true, y_pred)에서 box, confidence score, class score 데이터를 분리했다. 그리고 예측값인 y_pred에서 얻은 2개의 Bounding box 중 라벨값 y_true에 있는 ground truth box와 IoU(Intersection over Union)가 더 높은 것을 이용해 box 위치에 대한 로스를 구했고 두개의 Bounding box를 모두 이용해 confidence score에 대한 로스도 구했다(**그림1**). 그리고 예측한 클래스 예측값과 라벨 데이터로 클래에 대한 로스(**그림2**)를 구한 뒤 로스들을 더해 최종 로스를 획득한다.

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

그림 1. 손실 함수에서 Bounding box에 대한 로스(Localization loss)와 Confidence score에 대한 로스(Confidence loss). λ_coord = 5, λ_noobj = 0.5다.

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

그림 2. YOLO의 loss function에서 class score에 대한 로스(classification loss)

여기서 유의할 점은, **이미지 속 실제 객체가 해당 구역에 중심좌표를 두고 있지 않으면 로스를 0으로 처리한다는 것**이다. obj, noobj가 들어있는 기호가 이를 구현한 것이다. 식에 대한 자세한 설명은 논문에서 확인할 수 있다.

DenseNet-121 : Tensorflow에 있는 DenseNet-121를 Backbone으로 불러온다. Trainable 값을 False로 맞추어 네트워크를 동결시킨다. 최종 출력의 사이즈가 (14,14)이기에 VGG16을 이용한 구현에서와 동일하게 CNN, FCN를 쌓고 마지막에 커스텀 레이어를 추가해 (7, 7, 30)사이즈의 텐서가 나오도록 하는 Head를 구현했다. 손실함수는 keras backend 모듈을 사용하는 [14]의 코드를 참조했다. 세부 구조는 VGG16을 Backbone으로 한 구현과 유사하다.

3)훈련

Backbone에 관계없이 논문에 나온 대로 YOLO를 훈련시켰다. 텐서플로우에 있는 callback기능을 이용해 학습률(learning rate)를 조정했고 텐서플로우의 fit() 메소드를 이용해 학습용 데이터셋으로 모델을 학습시켰다. 매 에포크마다 검증용 데이터셋으로 검증 로스를 평가했으며 텐서플로우의 ModelCheckpoint를 이용해 검증 로스가 이전 에포크에 비해 줄어들 때마다 학습된 모델의 가중치를 .h5파일로 저장했다. 검증 로스가 떨어질 때만 가중치를 저장하니 검증 로스가 가장 낮은 모델을 최종 모델로 사용할 수 있었으며 가중치를 계속해서 저장한 덕분에 중간에 훈련이 잘못된 방향으로 진행되어 문제가 발생해도 잘되던 시점부터 다시 훈련을 진행할 수 있게 되었다.

4) 테스트

VGG-16을 기반으로 만든 YOLO를 학습시킨 후 가지고있는 데이터셋을 전부 이용해 어떻게 객체를 탐지했는지 확인해봤다. **그림 1**은 학습용 데이터셋으로 테스트한 결과, **그림 2**는 테스트용 데이터셋

으로 테스트한 결과다. 학습용 데이터에서는 객체 위치와 정보가 정확히 나타났지만 테스트용 데이터에서는 객체의 위치와 정보를 정확히 알아내지 못하는 경우가 많았고 심하면 하나도 검출하지 못할 때가 많았다. 이러한 오버피팅(Over Fitting)을 보이는 이유는 데이터셋을 구성할 때 데이터 증강을 하지 못한게 큰 지분을 차지하였다고 생각한다.

그림 3. 학습용 데이터셋으로 테스트한 결과중 하나, 제대로 검출했다.

그림 4. 테스트용 데이터셋으로 테스트한 결과중 하나. 사람과 말이 있으나 사람만 검출했고 검출 영역도 정확하지 않다는걸 확인할 수 있다.

2-2. 파이토치(PyTorch)를 이용한 구현

파이토치를 통해 구현해보는 것이 처음이었기 때문에 [13]에 이미 구현된 코드를 보며 각주를 달고 이해하는 활동을 하였다. 구현단계는 데이터셋 전처리, 모델 설계, 손실 함수, 훈련 및 테스트로 이루어지며 다음과 같다.

1)데이터셋 전처리

데이터셋은 PASCAL VOC 2007 dataset을 사용했다. 이미지 파일은 파이썬의 이미지 처리 모듈인 Pillow을 사용했고 이미지 속 객체 정보는 xml파일을 csv파일로 변환 후 이미지에 들어있는 객체 정보를 추출하는 방법을 사용해 데이터셋을 만들었다.

2)모델 설계

Backbone의 변경 없이 DarkNet + Head 구조를 사용했고 신경망을 생성하는 패키지인 torch.nn을 사용했다. 코드의 가독성을 위해 CNN, BN(Batch nomalization), LeakyReLU로 이루어진 CNNBlock로 YOLO에 존재하는 CNN 레이어들을 모두 구현했고 마지막에 FCN을 연결하여 YOLO를 완성시켰다.

3)손실 함수

nn.Module을 상속받은 YoloLoss를 손실함수로 사용했다. 텐서플로우로 구현할 때와는 달리 IoU계산, loss계산을 메서드로 구현했다. 예측한 두 개의 bounding box 중 Ground_truth_box와 IoU 값이 더 큰 box를 bestbox에 담아 Localization loss, Confidence loss를 계산했다.

4)훈련 및 테스트

텐서플로우로 학습시킬 때와 같은 설정을 하여 파이토치로 구현된 YOLO 학습을 시도해봤다. 그러나 학습을 시키는 과정에서 runtimeError: DataLoader worker (pid(s) 31892, 33392) exited unexpectedly 라는 오류를 맞이했다. 이는 학습을 시키는 컴퓨터의 GPU 메모리가 부족함 등의 이유로 발생하는 오류이며 적절한 num_workers로 설정하면 해결되는 오류임을 알아냈다. 우리는 [15]를 참고해 num_workers를 0으로 변경 후 다시 학습을 시도했으나 여전히 같은 오류가 발생하였기에 모델을 학습하지 못했다. 또 다른 오류는 파이썬 모듈 import 오류다. Utils.py에 학습을 위해 구현해놓은 메스드를 사용 하기위해 Utils를 import하려고 했으나 오류가 발생했다. 이러한 오류가 발생한 이유는 파이썬 설치와 환경변수 및 경로에서 문제가 발생한 것이라 생각된다. 이를 해결하기 위해 가상환경을 새로 만들어 처음부터 설정을 다시 해 훈련을 다시 시도했으나 같은 오류가 발생해 결국 YOLO를 학습시킬 수 없었다.

3. 결 론

YOLO는 객체탐지를 수행하는 모델 중 대표적인 모델로 연급될 정도로 수많은 객체탐지 모델에게 영향을 준 모델이다. 이러한 YOLO를 텐서플로우, 파이토치를 이용해 직접 구현하며 객체탐지가 어떻게 이뤄지는지 확인하는 과정은 우리들이 향후 객체탐지 분야를 공부하거나 객체탐지 외에도 다양한 분야를 공부할 큰 도움이 될 것으로 기대된다.

참고 문헌

1. A. Krizhevsky, I. Sutskever, and G. Hinton, "Imagenet classification with deep convolutional neural networks", NIPS, 2012.
2.R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation", CVPR, 2014.
3. R. Girshick, "Fast R-CNN," IEEE International Conference on Computer Vision (ICCV), 2015.
4. S. Ren, K. He, R. Girshick, and J. Sun, "Faster R-CNN: Towards real-time object detection with

region proposal networks", NIPS, 2015.
5. J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, Real-time Object Detection", arXiv preprint arXiv:1506.02640, 2015.
6. github, https://github.com/ultralytics/yolov5
7. Darknet: Open source neural networks in c, http://pjreddie.com/darknet
8. K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," ICLR, 2015.
9. O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge", IJCV, 2015.
10. G.Huang,Z.Liu,K.Q.Weinberger,andL.Maaten, "Densely connected convolutional networks", CVPR, 2017.
11. github, https://github.com/Kanghee-Lee/Faster-RCNN_TF-RPN-
12. Vivek Maskara, https://www.maskaravivek.com/post/yolov1/
13. github, https://github.com/aladdinpersson/Machine-Learning-Collection
14.github. https://github.com/JY-112553/yolov1-keras-voc/blob/master/yolo/yolo.py
15. Flssh, https://wsshin.tistory.com/18?category=943067