Chapter8.

2021210088 허지혜

Training Faster R-CNN-based custom object detectors

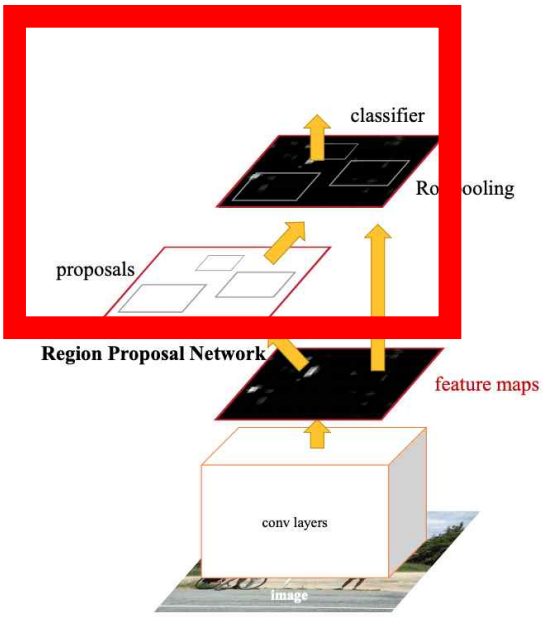
	R-CNN	Fast R-CNN
연산량	224 x 224 x 2000	600 x 1000 x 1
ROI 추출 방식	이미지에서 ROI 추출	이미지에서 RoI 추출, feature map에 RoI projection
학습 방식	Classifier, regressor, CNN 세개를 따로 학습시켜야함, CNN은 Pretrain AlexNet을 transfer learning	한번에 학습시키는 end-to-end 방식 (multi task loss 사용)

Fast R-CNN의 단점은 Region Proposal이 모형 안이 아니라 밖에 위치하여 bottleneck을 일으킨다. 또한, Region Proposal을 구하는 과정이 gpu가 아닌 cpu를 이용하여 계산해왔다.

Faster R-CNN은 지금까지의 Selective Search를 사용하여 계산해왔던 Region Proposal 단계를 Neural Network와 **융합**시켜 진정한 end-to-end Object Detection 모형을 제시한 모형이다. 이를 통해 모든 과정을 gpu에서 계산할 수 있게 되었다. 이 모형은 PASCAL VOC DATASET에서 5FPS 속도를 내며 78.8%의 성능을 내었다.

* end-to-end : 종단간이라고 불리며 처음부터 끝까지 파이프라인 네트워크 없이 데이터에 의존하여 한 번에 처리한다는 의미이다.

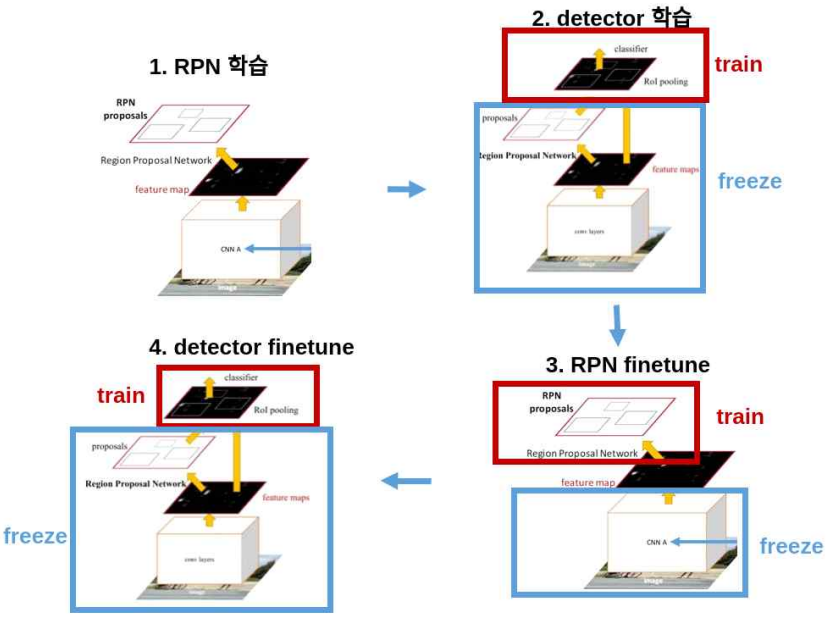
위 사진은 Faster R-CNN의 Network Architecture이다. Faster R-CNN의 과정은 다음과 같이 나타낼 수 있다.



출처. https://yeomko.tistory.com/17

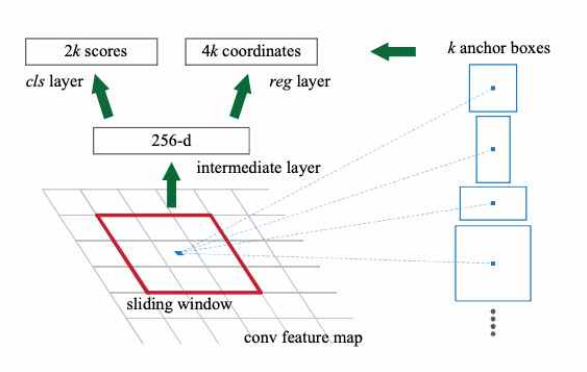
Faster R-CNN = RPN + Fast R-CNN

1. ImageNet pretrained 모형을 불러온 다음, RPN을 학습시킨다.
2. 1.에서 학습시킨 RPN에서 CNN을 제외한 Region Proposal Layer만 가져와 이를 활용하여 Fast R-CNN을 학습시킨다. 이때 처음 Feature map을 추출하는 CNN은 fine-tunning시킨다.
3. 2.에서 학습시킨 Fast R-CNN과 RPN을 불러온 다음 다음 weight는 고정하고 RPN에 해당하는 Layer만 fine-tunning시킨다. 여기 과정부터 convolution weight를 공유한다.
4. 마지막으로 CNN, RPN을 고정시킨 채, Fast R-CNN에 해당하는 Layer를 fine-tunning시킨다.



- 핵심 아이디어

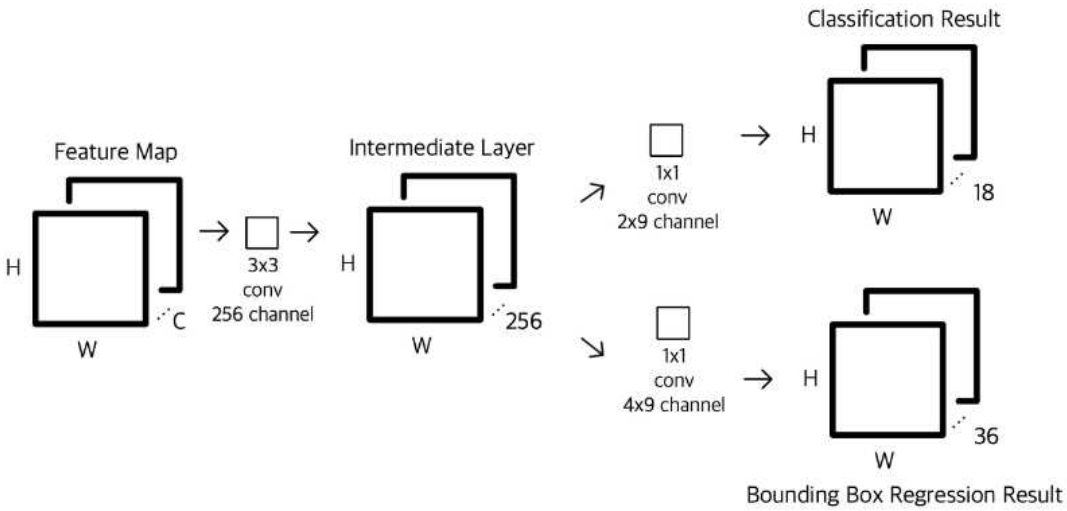
RPN(Region Proposal Network)



출처. https://yeomko.tistory.com/17

Faster R-CNN은 Fast R-CNN의 Selective Search 과정을 제거하고 위 그림의 **Region Proposal Network(RPN)**을 이용하여 RoI(Region of Inter???)를 계산한다. 이 과정을 통해 RoI 계산의 정확도를 올릴 수 있게 되었다.

RPN은 CNN을 기반으로 둔 모형으로, 위 CNN을 통해 Region Proposal을 생성하고 Object Detection도 같이 수행한다. 위 사진을 직관적으로 풀어둔 사진을 가져오면 다음과 같다.



순서는 다음과 같다.

1. INPUT : CNN을 통해 뽑아낸 Feature map을 입력으로 받는다. (H,W,C)
2. Intermediate Layer : Feature map에 3x3 Convolution을 256 or 512 Channel만큼 수행한다. 이때 padding size를 1로 설정하여 H,W가 보존될 수 있게 한다. (H,W,256 or 512)
3. (H,W,256 or 512)를 입력으로 받아 Classification과 Bounding box Regression

을 계산한다. 계산할 때는 Fully Connected Layer가 아니라 Fully Convolution Network의 특징을 가진다. 이는 입력 크기에 상관없이 동작할 수 있도록 하기 위함이다.

Classification : 1x1 Convolution을 (2(Object이다, 아니다)x9(Anchor Box 개수)) 만큼 수행해준다. 따라서 (H,W,18)이 출력값이 된다. H,W는 Feature map의 좌표를 의미하고 18 channel은 해당 좌쵸를 앵커로 삼아 k개의 앵커 좌표들에 대한 예측을 수행하게 한다. 그 후 reshape와 softmax를 이용하여 확률값을 얻는다.

Bounding Box Regression : 1x1 Convolution을 (4(Bounding box 개수)x 9(Anchor Box 개수)) 만큼 수행해준다.

4. 앞서 얻은 값들로 RoI를 계산한다.

Classification을 통해 얻은 물체일 확률을 정렬한 다음 높은 순서대로 k개의 앵커만 추려낸다. 그 후 k개의 앵커에 각각 Bounding box Regression을 적용한다. 그다음 NMS를 적용해 RoI를 구해준다.

이러한 과정들을 통해 얻어진 RoI를 이용하여 다시 첫 Feature map에 project한 다음 RoI Pooling을 적용한 뒤 Classification과 Regression을 하면 된다.

Loss Function

Faster R-CNN의 Loss를 구하는 함수는 Fast R-CNN과 마찬가지로 Multi-Task Loss이다. 식을 나타내면 다음과 같다.

$$L(\{p_i\},\{t_i\}) = \frac{1}{N_{cls}}\sum_i L_{cls}(p_i, p_i^*) + \lambda\frac{1}{N_{reg}}\sum_i p_i^* L_{reg}(t_i, t_i^*).$$

I : 엥커

p_i : Classification을 통해 얻은 해당 앵커가 Object일 확률(p_i* : 정답)

t_i : Bounding box Regression을 통해 얻은 box vector(t_i* : 정답)

Lambda : 가중치를 조절, 논문에서는 10 설정

Classification Loss : Log Loss이용, Regression Loss : smoothL1 Loss 이용

특이점) Ncls, Nreg를 나누어준다. 이는 각각 mini-batch size인데 논문에서는 256이다. 이때 Nreg는 앵커개수라 대략 2400개(256*9)이다.

- R-CNN, Fast R-CNN, Faster R-CNN의 흐름

http://www.aitimes.kr/news/articleView.html?idxno=12087

객체 탐지(Object Detection)은 컴퓨터 비전(Computer Vision)기술 중 하나로 단순 분류 문제에서 벗어나 세부사항을 쉽게 알 수 있다. 가령, 의료 이미지에서 질병 식별 및 질병의 구체적 위치를 찾아주거나 자율주행에서 보행자와 신호등 구별을 해준다. 이러한 객체 인식은 여러 가지 방법론이 존재한다. 단일 단계 방식(One stage detector)의 객체 탐지 알고리즘과 이단계 방식(two stage detector)의 객체 탐지 알고리즘이 있다. 이들 중 우리는 이단계 방식에 있는 모형에 대하여 공부중이다. 이단계 방식은 R-CNN 계열의 모형을 대표 모형으로 나타내는데 3가지 모형을 정리해보자.

① **R-CNN**

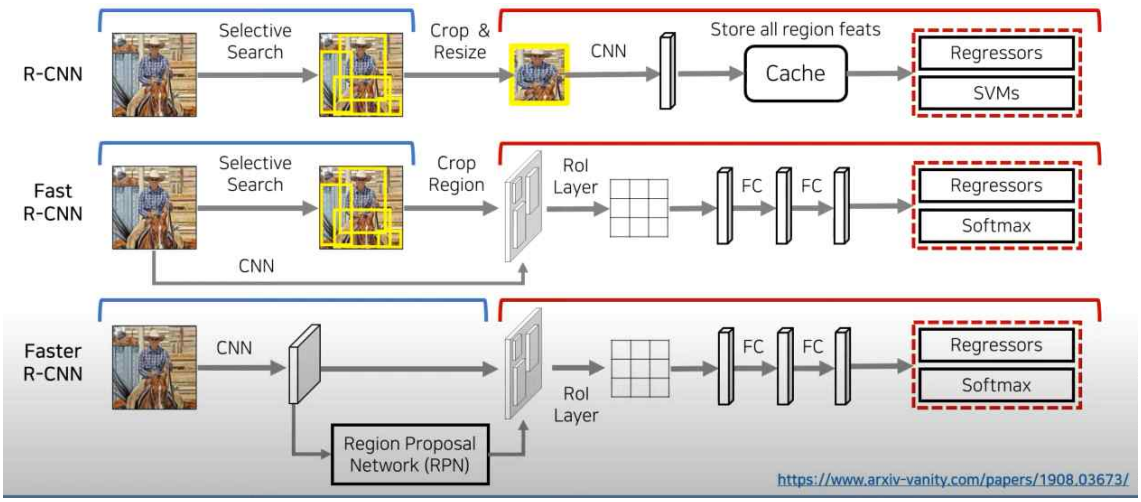
단계를 간단하게 설명하자면 RoI(Selective Search)를 추출한 뒤, CNN을 거치고 SVM, Bbox regression을 수행하여 detection한다.

② **Fast R-CNN**

먼저, 전체 Image를 CNN에 통과시킨 후, RoI(Selective Search)를 추출한 후 projection하기 위해 RoI Pooling layer를 지난다. 그 후, Softmax, Bbox regression을 수행하여 detection한다.

③ **Faster R-CNN**

CNN을 지나 Feature map 추출 후, RoI(Region Proposal Network)를 추출한다. 그 후 RoI Pooling layer를 지나 고정 크기로 조정 후 Softmax, Bbox regression을 수행하여 detection한다. 이때 대부분의 순서는 Fast R-CNN과 동일하다.



코드

- 패키지 불러오기

```
!pip install -q --upgrade selectivesearch torch_snippets
from torch_snippets import *
import selectivesearch
from torchvision import transforms, models, datasets
from torch_snippets import Report
from torchvision.ops import nms
#device = 'cuda' if torch.cuda.is_available() else 'cpu'
device='cpu'
```

- 데이터 불러오기

• 데이터 설명

데이터는 kaggle에서 다운받았다. R-CNN과 마찬가지로 image가 들어있는 **images 폴더**와 bounding box 좌표, image_id, image_path 등이 들어있는 **df.csv 파일**이 있다.
먼저, df.csv 파일을 살펴보자.

```
IMAGE_ROOT = './data/images/images'
DF_RAW = pd.read_csv('./data/df.csv')
print(DF_RAW.head())
```

df.csv 파일이 가지고 있는 컬럼은 21개가 존재하지만 주로 쓰는 열은 6개 정도이다.

'ImageID, LabelName, XMin, XMax, YMin, YMax'

df.csv 파일이 가지고 있는 행은 24062개이다. 같은 ImageID를 가진 행들이 존재한다.

```
label2target = {l:t+1 for t,l in enumerate(DF_RAW['LabelName'].unique())}
label2target['background'] = 0
target2label = {t:l for l,t in label2target.items()}
background_class = label2target['background']
num_classes = len(label2target)
```

```
def preprocess_image(img):
    img = torch.tensor(img).permute(2,0,1)
    return img.to(device).float()
```

- OpenDataset class 정의

```
class OpenDataset(torch.utils.data.Dataset):
    w, h = 224, 224
    def __init__(self, df, image_dir=IMAGE_ROOT):
        self.image_dir = image_dir
```

```
        self.files = glob.glob(self.image_dir+'/*')
        self.df = df
        self.image_infos = df.ImageID.unique()
    def __getitem__(self, ix):
        # load images and masks
        image_id = self.image_infos[ix]
        img_path = find(image_id, self.files)
        img = Image.open(img_path).convert("RGB")
        img = np.array(img.resize((self.w, self.h), resample=Image.BILINEAR))/255.
        data = df[df['ImageID'] == image_id]
        labels = data['LabelName'].values.tolist()
        data = data[['XMin','YMin','XMax','YMax']].values
        data[:,[0,2]] *= self.w
        data[:,[1,3]] *= self.h
        boxes = data.astype(np.uint32).tolist() # convert to absolute coordinates
        # torch FRCNN expects ground truths as a dictionary of tensors
        target = {}
        target["boxes"] = torch.Tensor(boxes).float()
        target["labels"] = torch.Tensor([label2target[i] for i in labels]).long()
        img = preprocess_image(img)
        return img, target
    def collate_fn(self, batch):
        return tuple(zip(*batch))

    def __len__(self):
        return len(self.image_infos)
```

OpenImages class를 통하여 얻을 수 있는 값은 4가지이다.

◆ image : image의 numpy값
◆ boxes : bounding box 좌표값
◆ classes : Truck or Bus
◆ image_path : 이미지 경로

얻은 값을 시각화하기 위해 torch_snippets의 show 함수를 이용하였다.
위 [[40, 36, 184, 105], [158, 58, 255, 93]]는 bounding box 좌표값이다.

- Data Loader 정의

```
from sklearn.model_selection import train_test_split
trn_ids, val_ids = train_test_split(df.ImageID.unique(), test_size=0.1,random_state=
```

```
99)
trn_df, val_df = df[df['ImageID'].isin(trn_ids)], df[df['ImageID'].isin(val_ids)]
len(trn_df), len(val_df)

train_ds = OpenDataset(trn_df)
test_ds = OpenDataset(val_df)

train_loader = DataLoader(train_ds, batch_size=4, collate_fn=train_ds.collate_fn,
drop_last=True)
test_loader = DataLoader(test_ds, batch_size=4, collate_fn=test_ds.collate_fn,
drop_last=True)
```

- pretrained weight 불러오기

torchvision.models.detection에서는 Faster R-CNN API를 제공하고 있어 쉽게 구현이 가능하다.

torchvision.models.detection.fasterrcnn_resnet50_fpn

-> COCO 데이터셋을 ResNet50으로 pretrained한 모델이다.

num_classes는 background를 추가해야 하기 때문에 n+1개의 값이 들어간다.

```
import torchvision
from torchvision.models.detection.faster_rcnn import FastRCNNPredictor

device = 'cuda' if torch.cuda.is_available() else 'cpu'

def get_model():
    model = torchvision.models.detection.fasterrcnn_resnet50_fpn
(pretrained=True)
    in_features = model.roi_heads.box_predictor.cls_score.in_features
    model.roi_heads.box_predictor =
FastRCNNPredictor(in_features, num_classes)
    return model
```

- train & validation 함수 정의

```
# Defining training and validation functions for a single batch
def train_batch(inputs, model, optimizer):
    model.train()
    input, targets = inputs
    input = list(image.to(device) for image in input)
    targets = [{k: v.to(device) for k, v in t.items()} for t in targets]
    optimizer.zero_grad()
    losses = model(input, targets)
```

```
    loss = sum(loss for loss in losses.values())
    loss.backward()
    optimizer.step()
    return loss, losses

@torch.no_grad() # this will disable gradient computation in the function below
def validate_batch(inputs, model):
    model.train() # to obtain the losses, model needs to be in train mode only.
# #Note that here we are not defining the model's forward method
#and hence need to work per the way the model class is defined
    input, targets = inputs
    input = list(image.to(device) for image in input)
    targets = [{k: v.to(device) for k, v in t.items()} for t in targets]

    optimizer.zero_grad()
    losses = model(input, targets)
    loss = sum(loss for loss in losses.values())
    return loss, losses
```

- **모형 정의**

```
model = get_model().to(device)
optimizer = torch.optim.SGD(model.parameters(), lr=0.005,
                            momentum=0.9, weight_decay=0.0005)
n_epochs = 5
log = Report(n_epochs)
```

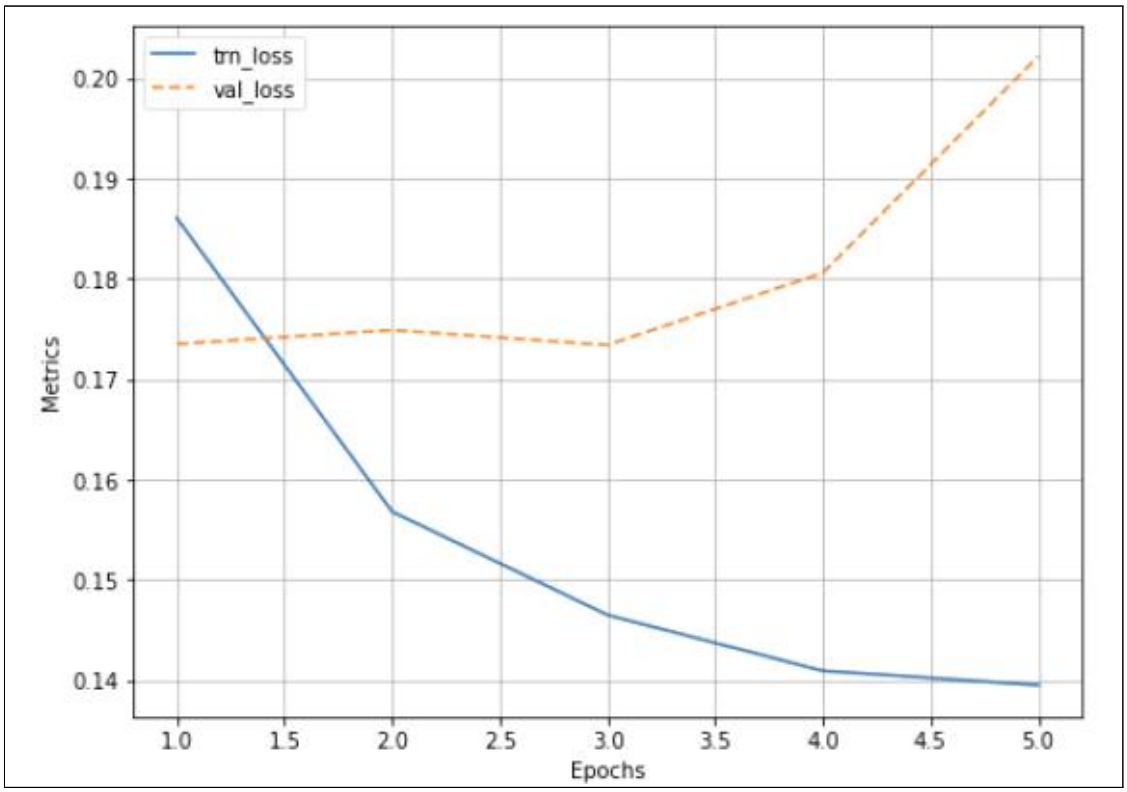
- **학습 시키기**

```
for epoch in range(n_epochs):
    _n = len(train_loader)
    for ix, inputs in enumerate(train_loader):
        loss, losses = train_batch(inputs, model, optimizer)
        loc_loss, regr_loss, loss_objectness, loss_rpn_box_reg = \
            [losses[k] for k
in ['loss_classifier','loss_box_reg','loss_objectness','loss_rpn_box_reg']]
        pos = (epoch + (ix+1)/_n)
        log.record(pos, trn_loss=loss.item(), trn_loc_loss=loc_loss.item(),
                   trn_regr_loss=regr_loss.item(),
trn_objectness_loss=loss_objectness.item(),
                   trn_rpn_box_reg_loss=loss_rpn_box_reg.item(), end='\r')
```

```
    _n = len(test_loader)
    for ix,inputs in enumerate(test_loader):
        loss, losses = validate_batch(inputs, model)
        loc_loss, regr_loss, loss_objectness, loss_rpn_box_reg = \
          [losses[k] for k in
['loss_classifier','loss_box_reg','loss_objectness','loss_rpn_box_reg']]
        pos = (epoch + (ix+1)/_n)
        log.record(pos, val_loss=loss.item(), val_loc_loss=loc_loss.item(),
                   val_regr_loss=regr_loss.item(),
val_objectness_loss=loss_objectness.item(),
                   val_rpn_box_reg_loss=loss_rpn_box_reg.item(), end='\r')
    if (epoch+1)%(n_epochs//5)==0: log.report_avgs(epoch+1)

log.plot_epochs(['trn_loss','val_loss'])
```

```
EPOCH: 1.000  trn_loss: 0.186  trn_loc_loss: 0.077  trn_regr_loss: 0.084
trn_objectness_loss: 0.017  trn_rpn_box_reg_loss: 0.008  val_loss: 0.174
val_loc_loss: 0.070  val_regr_loss: 0.080  val_objectness_loss: 0.014
val_rpn_box_reg_loss: 0.009  (779.76s - 3119.05s remaining)
EPOCH: 2.000  trn_loss: 0.157  trn_loc_loss: 0.065  trn_regr_loss: 0.075
trn_objectness_loss: 0.010  trn_rpn_box_reg_loss: 0.007  val_loss: 0.175
val_loc_loss: 0.071  val_regr_loss: 0.080  val_objectness_loss: 0.015
val_rpn_box_reg_loss: 0.009  (1541.89s - 2312.84s remaining)
EPOCH: 3.000  trn_loss: 0.146  trn_loc_loss: 0.060  trn_regr_loss: 0.071
trn_objectness_loss: 0.009  trn_rpn_box_reg_loss: 0.007  val_loss: 0.173
val_loc_loss: 0.069  val_regr_loss: 0.080  val_objectness_loss: 0.016
val_rpn_box_reg_loss: 0.009  (2304.00s - 1536.00s remaining)
EPOCH: 4.000  trn_loss: 0.141  trn_loc_loss: 0.057  trn_regr_loss: 0.070
trn_objectness_loss: 0.008  trn_rpn_box_reg_loss: 0.007  val_loss: 0.181
val_loc_loss: 0.073  val_regr_loss: 0.083  val_objectness_loss: 0.016
val_rpn_box_reg_loss: 0.009  (3066.69s - 766.67s remaining)
EPOCH: 5.000  trn_loss: 0.139  trn_loc_loss: 0.056  trn_regr_loss: 0.069
trn_objectness_loss: 0.007  trn_rpn_box_reg_loss: 0.006  val_loss: 0.202
val_loc_loss: 0.089  val_regr_loss: 0.084  val_objectness_loss: 0.020
val_rpn_box_reg_loss: 0.009  (3830.10s - 0.00s remaining)
```



- **test**

```
from torchvision.ops import nms
def decode_output(output):
    'convert tensors to numpy arrays'
    bbs = output['boxes'].cpu().detach().numpy().astype(np.uint16)
    labels = np.array([target2label[i] for i in
 output['labels'].cpu().detach().numpy()])
    confs = output['scores'].cpu().detach().numpy()
    ixs = nms(torch.tensor(bbs.astype(np.float32)), torch.tensor(confs), 0.05)
    bbs, confs, labels = [tensor[ixs] for tensor in [bbs, confs, labels]]

    if len(ixs) == 1:
        bbs, confs, labels = [np.array([tensor]) for tensor in [bbs, confs, labels]]
    return bbs.tolist(), confs.tolist(), labels.tolist()
```

```
model.eval()
for ix, (images, targets) in enumerate(test_loader):
    if ix==3: break
    images = [im for im in images]
    outputs = model(images)
    for ix, output in enumerate(outputs):
        bbs, confs, labels = decode_output(output)
        info = [f'{l}@{c:.2f}' for l,c in zip(labels, confs)]
        show(images[ix].cpu().permute(1,2,0), bbs=bbs, texts=labels, sz=5)
```