YOLO 부분만

1. main.py

argparse()를 통해 커맨드 라인에 입력된 인자들을 파싱

1. -C 또는 --camera: 카메라 또는 비디오 파일의 경로를 지정. 기본값은 source
2. --detection_input_size: 검출 모델에 입력되는 이미지의 크기를 지정.
   정사각형 형태이며, 32로 나누어 떨어져야 함. 기본값은 384입니다.
3. --pose_input_size: pose 모델에 입력되는 이미지 크기 지정.

높이, 너비를 쉼표로 구분해 입력, 32로 나누어 떨어져야 함. (기본값 '224x160')

4. --pose_backbone: SPPE FastPose 모델의 backbone 지정.(기본값 'resnet50')
5. --show_detected: 검출된 객체들의 경계 상자를 화면에 출력. (기본값 False)
6. --show_skeleton: 검출된 객체들의 스켈레톤 포즈 화면에 출력. (기본값 True)
7. --save_out: 디스플레이 화면을 비디오 파일로 저장. 저장할 파일 경로를 입력.
8. --device: 모델을 실행할 디바이스를 선택. (기본값 'cuda')

(※ 32는 TinyYOLOv3가 입력 이미지를 32배수 크기의 그리드로 나누기 때문)

이 중 YOLO는 2번 항목인 –detection_input_size 사용

2. DetectLoader.py

class TinyYOLOv3_onecls(inp_dets, device=device)

'oncls'는 one class로 해당 모델에서는 특정 클래스인 사람 객체만 검출

파라미터: device

기본값 cuda 사용

파라미터: inp_dets(int)

검출 모델에 입력되는 이미지의 크기 - 384

def __init__ :

input_size: (int)입력 이미지의 크기, 32로 나누어 떨어져야 함. Default: 416,

config_file: (str) YOLO 모델 구조 설정 파일의 경로(yolov3-tiny-onecls.cfg)

weight_file: (str) 사전 학습된 가중치 파일의 경로(best-model.pth)

nms: (float) Non-Maximum Suppression(비최대 억제) 중첩 임계값= 0.2

conf_thres: (float) 예측된 bbox의 최소 확률 임계값 = 0.45

device: (str) cuda로 선택

self.model에서 config_file을 사용한 Darknet으로 정의

def detect(self, image, need_resize=True, expand_bb=5):

RGB 이미지를 입력받아 객체를 검출하는 함수

파라미터: need_resize

need_resize가 True이면 input size로 크기 조정, 현재 True로 되어 있음

파라미터: expand_bb = 5

bbox를 이미지 경계를 벗어나게 확장시키는 값으로, bbox가 실체 물체의 경계까지 포함되도록 함

-> 검출된 객체 주위에 여유 공간이 있도록 확장하여 이미지 경계를 벗어나는 부분의 객체를 놓치

지 않도록 함

핵심

```
# 모델을 통해 bounding box detection 수행
detected = self.model(image.to(self.device))# 이미지를 모델에 통과시켜 detected에 저장
```

이후, **non_max_suppression** 함수를 통해 중복되는 bbox를 제거하고 conf_thes 이하의 bbox 제거하고 **detected** 변수에 저장한다.

return

detected 변수에서 추출하여

bounding box의

1.top
2.left
3.bottom
4.right
5.bbox_score
6.class_score
7.class
의 7가지 결과를 반환한다.

3. Models.py

def create_modules

모듈 생성 함수로 nn.ModuleList 형태로 모듈 리스트를 만들고 반환

module_def을 인자로 받아와, parsing. 이 함수는 Darknet 클래스 및 더 나아가 TinyYOLOv3 클래스에서 config_file을 이용해 hyperparameter와 레이어를 분리한다.

config 파일에 "convolutional", "maxpool", "upsample", "route", "shortcut", "yolo"에 대하여 각각의 레이어가 어떤 기능을 해야하는지 정의하고 있다. Darknet에서 다시 이를 분리해 순서에 맞게 정의된다.

class Upsample

특성 맵의 크기를 늘리는 클래스

특성 맵의 크기를 늘리기 위해 nearest라고 하는 최근접 이웃 보간 방식을 사용

이를 위해 F.interpolate 함수를 사용하는데, 이미지를 확대하거나 축소할 때 사용되며, 이미지를 이루는 픽셀 값들을 새로운 이미지의 크기에 맞게 보간한다. 입력 이미지에서 새로운 이미지로 보간되는 각 픽셀은 가장 가까운 픽셀 값으로 결정. 즉, 각 픽셀은 입력 이미지에서 가장 가까운 픽셀 값으로 결정. 계산이 간단하고 빠르지만, 업샘플링 과정에서 이미지의 선명도가 떨어짐

class EmptyLayer

"route"와 "shortcut layer"를 사용할 때 쓰이며 말 그대로 아무 기능도 없는 layer

이 두 layer는 별도의 학습 파라미터를 갖지 않기에, 이러한 layer를 구현할 때 nn.Module을 상속하여 새로운 layer를 정의하는 것은 비효율적이다. 따라서 EmptyLayer 클래스를 정의하여 이 두 layer의 연산을 처리하도록 한다. EmptyLayer는 forward 함수를 가지고 있지 않기 때문에, 단순히 입력을 전달. 이렇게 하면 nn.Module을 상속하여 클래스를 정의하는 것보다 메모리 사용량과 계산량을 줄일 수 있다.

class Darknet

참고로 YOLO 모델은 self.module_list로 구성됨

0. self.module_defs
1. self.hyperparams, self.module_list = create_modules(self.module_defs)
2. self.yolo_layers
3. self.img_size
4. self.seen
5. self.header_info

0) `self.module_defs = parse_model_config(config_path)`

self.modul_def는 Darknet을 이용하고 config_file을 사용해 네트워크 구조 정의

(config_File에는 24개의 layer로 구성되어있는 것을 확인)

아래는 config 파일에 적혀있는 layer로, 레이어의 진행 순서가 아니라 단순히 표기

0[convolutional] # 1[maxpool] # 2[convolutional] # 3[maxpool]

4[convolutional] # 5[maxpool] # 6[convolutional] # 7[maxpool]

8[convolutional] # 9[maxpool] # 10[convolutional] # 11[maxpool]

12[convolutional]

###########

13[convolutional] # 14[convolutional] # 15[convolutional]

16[yolo] # 17[route] # 18[convolutional] # 19[upsample]

20[route] # 21[convolutional] # 22[convolutional] # 23 [yolo]

1) `self.hyperparams, self.module_list = create_modules(self.module_defs)`

self.hyperparam과 self.module_list는 create_module 함수에서 self.module_defs를 사용해 정의

아래는 config 파일에 있는 hyperparameter

[net]

# Testing
batch=1
subdivisions=1
# Training
# batch=64
# subdivisions=2
width=416
height=416
channels=3
momentum=0.9
decay=0.0005
angle=0
saturation = 1.5
exposure = 1.5
hue=.1
learning_rate=0.001
burn_in=1000
max_batches = 500200
policy=steps
steps=400000,450000

scales=.1,.1

2) self.yolo_layer는

```
self.yolo_layers = [layer[0] for layer in self.module_list if hasattr(layer[0], "metrics")]
```

metrics 속성을 가진 layer를 찾아 yolo_layers 리스트에 저장

(여기서 metrics는 모델이 학습하면서 사용하는 지표를 의미)

※ hasattr()는 첫 번째 인자 '객체', 두 번째 인자 '속성이름'을 전달하여 self.module_list 안에 해당 객체가 있다면 True 반환 아니면 False를 반환한다.

3) self.img_size

img_size는 416으로 정의

4) self.seen

seen은 현재까지 학습된 이미지의 수로 현재 0으로 되어있으며, 새로운 학습을 시작할 때 0으로 초기화

5) self.header_info

```
self.header_info = np.array([0, 0, 0, self.seen, 0], dtype=np.int32)
```

모델 가중치 파일을 저장할 때 파일의 헤더 정보를 저장하기 위해 사용

def forward(self, x, targets=None):

입력 이미지 x와 정답 라벨 targets로 YOLO 출력값 반환

(layer_output은 layer 출력값 저장, yolo_outputs은 scale 출력값 저장)

layer_outputs, yolo_outputs = [], []

이후,

self.module_defs에서 각 type들을 살펴봄

1. 만약 module_def["type"]이 **"convolutional", "upsample", "maxpool"** 안에 있다면

    x = module(x) 코드 실행

여기서 module은 해당 layer의 클래스 객체를 나타내며, 이 객체의 __call__() 메서드를 호출하여 입력 데이터 x를 해당 layer에 적용.

이렇게 하면, 입력 데이터 x가 각 layer를 통과하면서 출력 데이터가 생성.

생성된 출력 데이터 x를 다시 다음 레이어의 입력으로 사용할 수 있다.

2. elif module_def["type"]이 **"route"** 라면

    이전 레이어의 출력값을 연결(concatenate)하여 현재 레이어의 입력값으로 사용

이전 레이어의 출력값들을 가져와서 concatenate하는 이유는, 모델의 feature map을 다양한 scale에서 결합하면서 더욱 복잡한 특징들을 추출하기 위해

3. elif module_def["type"]이 "**shortcut**" 이라면

    이전 layer의 출력값과 더하는 연산을 수행하는 layer 생성

    x = layer_outputs[-1] + layer_outputs[layer_i]로 나타냄

x = layer_outputs[-1] + layer_outputs[layer_i]는 이전 layer의 출력값 layer_outputs[-1] 과 layer_i 인덱스를 가진 이전 layer의 출력값 layer_outputs[layer_i] 을 더함.

이 값을 현재 shortcut layer의 출력값 x로 지정

4. elif module_def["type"]이 "**yolo**" 라면

    x, layer_loss = module[0](x, targets, img_dim)

    loss += layer_loss

    yolo_outputs.append(x)

따라서 YOLO layer에서는 입력값으로 feature map(x), ground truth인 라벨(targets), 이미지의 크기(img_dim)를 받아들이며, 출력값으로 bounding box와 클래스 확률을 출력. 이때, 출력값과

ground truth 간의 손실(loss)도 계산되고, 이러한 YOLO layer의 출력값은 yolo_outputs 리스트에 추가된다.

이후 모든 yolo layer의 출력을 연결하고 이를 return

def load_darknet_weights(self, weights_path):

Darknet 모델에서 사전 학습된 weights 파일을 parsing하고 해당 가중치를 모델의 각 layer에 로드

def save_darknet_weights(self, path, cutoff=-1):

darknet 형식의 가중치를 저장하는 함수

def load_pretrain_to_custom_class(self, weights_pth_path):

미리 학습된 PyTorch 모델의 가중치(weights)를 커스텀 클래스의 가중치로 로드하는 함수