# **6D pose estimation**

# **Data Specification**

* 여기에 원본 image,
* 축 추가 image,
* return 할 6d, label, size 혹은 거리 데이터 등을 적어주세요!

예시입니다! 원본 사진은 빨간 선이 없는 사진입니다



[{'name': 0,

'location': [1.2527949714576911, 0.47768698546240124, 7.904661829579003], 'quaternion': Quaternion([-2.91814613e-02, 2.86069184e-01, -4.09498438e-04,

9.57764427e-01]),

'cuboid2d': array([[314.05579947, 255.39048837],

[291.53795008, 254.82506029],

[287.61068136, 288.84466116],

[313.49556449, 286.68493733],

[304.99883248, 254.2942211 ],

[279.20310049, 253.99572291],

[278.35610772, 289.57658675],

[305.49198092, 288.41620929],

[296.1298292 , 271.55448321]]),

'projected\_points': array([[317.90463713, 255.92488736],

[289.14532779, 256.44855284],

[288.88706044, 286.27961838],

[317.83001299, 287.7904503 ],

[305.66671206, 253.94840439],

[274.68675514, 254.66721715],

[274.29566558, 287.70492408],

[305.48781609, 289.49994358],

[296.57295804, 271.47034787]]),

'score': 0.3046929}]



[{'name': 0, 'location': [-2.070444752568392, 0.291737852791006, 4.1978190183183095], 'quaternion': Quaternion([-0.03019635, -0.90213811, 0.01245618, 0.43020909]), 'cuboid2d': array([(87.02926207511882, 245.16912519828298),

(97.7725255330067, 242.92835730517308),

(93.33088727797552, 308.52217114297366),

(90.42851402997303, 306.1350122675931), None,

(170.79382222274592, 239.87374667299346),

(173.5446518195461, 303.77830467101046), None,

(134.2033336518966, 274.24735975397215)], dtype=object), 'projected\_points': array([[ 89.41756809, 245.69795084],

[ 98.39417615, 238.6856662 ],

[ 99.42178337, 312.03853745],

[ 90.1550953 , 305.73587212],

[151.10298007, 246.83260011],

[170.24660278, 241.09633874],

[171.96551031, 303.21652503],

[152.35828408, 299.13056862],

[129.73589611, 273.79135546]]), 'score': 0.1739532}]

# **8/6 상의**

추론 과정

DOPE network로 belief map(큐보이드 점 히트맵), affinity map 출력

출력된 결과물을 가지고 pnp 알고리즘으로 후처리 -> 2D사물의 좌표를 3D 좌표로.

* DOPE 원코드 링크: <https://github.com/NVlabs/Deep_Object_Pose>
* 학습
  + scripts/train.py
* 추론
  + src/dope/inference/
  + 추론에 필요한 코드들
* hit map 이란?
  + object 가 있을걸로 예측되는 대표점 하나가 밝게 빛나는 사진(행렬) x 9

# **8/7 윤성, 영재, 제환 회의**

unreal engine editor 에서 3d 모델 부를때, fbx, obj 도 가능.

다음주 목요일(13일) 에 1차적인 결과 들고 미팅 갖기.

# **8/17 DOPE 매뉴얼**

**환경**

원본 깃허브(<https://github.com/NVlabs/Deep_Object_Pose>)의 소스코드를 기본으로 하나, 깃허브에서 제시된 환경 구축에 문제가 있어 전체적으로 버전을 높여 구축하였음. opencv-python의 경우 버전 3 이하를 사용할 것

깃허브에 제시된 필요 패키지명 및 버전:

python==2.7

pyrr==0.9.2

torch==0.4.0

numpy==1.14.2

scipy==1.1.0

opencv\_python==3.4.1.15

Pillow==5.3.0

torchvision==0.2.1

현재 사용 패키지명 및 버전:

python==3.8.3

pyrr==0.10.3

torch==1.5.0

numpy==1.18.1

scipy==1.4.1

opencv\_python==3.4.9.33

Pillow==7.1.2

torchvision==0.6.0

버전을 높이는 과정에서 발생한 몇몇 버전 문제 및 소스코드 자체의 문제(특히 학습 코드 script/train.py 에 “It is provided as a convenience for researchers, but it is otherwise unsupported.”로 명시됨에 따라 발생)를 해결하고자 코드의 몇몇 부분이 수정되었음. 이는 추론 과정에서도 이슈가 있으며, 중요 부분만 아래에서 명시함.

**학습**

기존 학습 소스코드는 scripts/train.py 이나, data augmentation 과정 중 하나인 cropping에 문제가 발생하여 cropping을 제거한 train\_no\_cropping.py 코드를 만들어 사용하였음. (원인불명)

학습 소스코드에서 다른 참고하는 다른 소스코드는 없음. 모두 코드 내에 내에서

학습 데이터의 형식은 논문에서 사용한 FAT 데이터(<https://research.nvidia.com/publication/2018-06_Falling-Things>)와 같음.

학습 명령어:

train\_no\_cropping.py [-h] [--data DATA] [--datatest DATATEST] [--object OBJECT] [--workers WORKERS] [--batchsize BATCHSIZE] [--imagesize IMAGESIZE] [--lr LR] [--noise NOISE] [--net NET] [--namefile NAMEFILE] [--manualseed MANUALSEED] [--epochs EPOCHS] [--loginterval LOGINTERVAL] [--gpuids GPUIDS [GPUIDS ...]] [--outf OUTF] [--sigma SIGMA] [--save] [--pretrained PRETRAINED] [--nbupdates NBUPDATES] [--datasize DATASIZE] [--option OPTION]

일반적으로 다음과 같이 사용(예시):

python train\_no\_crop.py --data ../turtlebot --outf turtle\_no\_crop --batchsize 8 --epochs 10 --imagesize 512

--data 학습 데이터의 위치

--outf 학습 결과(통계 및 가중치) 저장경로

--imagesize 학습 데이터의 height 값

이어서 학습하고자 할 경우 --net 옵션에 저장된 가중치 경로를 입력할 것.

**추론**

DOPE의 이론적 추론 방법은 다음과 같다.

1. 원본 이미지 입력가 DOPE Network에 입력되면 두 벡터 belief map (물체의 위치 추론과 관련. 총 9개의 히트맵으로, 8개의 cuboid 꼭지점 추론과 1개의 중앙 점 추론에 대응), affinity map (물체의 방향 추론과 관련)을 출력한다.

2. 1의 belief map과 affinity map을 입력받아 설정한 threshold에 따라 map들의 각 요소들을 3에서 사용할지 폐기할지 정한다.

3. 2에서 살아남은 요소들을 이용하여 PnP 알고리즘으로 위치(그림 1 참조)와 방향(Quaternion, xyzw 순)을 추론한다. 살아남은 요소들이 충분히 많지 않은 경우 감지한 물체가 없는 것으로 판단한다.

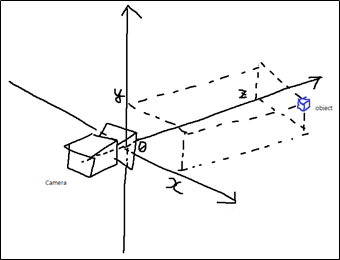


그림 1. DOPE의 좌표계

추론 과정 1번은 DOPE 네트워크를 사용하는 과정이며, 2번 및 3번은 후처리 과정임.

DOPE의 추론 과정에 사용되는 소스코드와 관련 과정은 다음과 같다.

nodes/camera

> ROS 노드: 로봇의 camera 노드. 사진을 실시간으로 dope 노드로전송한다.

nodes/dope

> ROS 노드: dope 노드. camera 노드로부터 전달받은 이미지를 실시간으로 DOPE 네트워크 및 후처리를 거쳐 타겟 물체의 위치와 방향을 출력한다. 이 때 아래의 세 코드를 참조한다.

src/dope/inference/detector.py

> DOPE 네트워크 클래스를 담고 있으며 threshold가 적용된다.

src/dope/inference/cuboid\_pnp\_solver.py

> 후처리의 PnP 알고리즘 관련 코드. 아래의 cuboid.py 코드를 참조한다. PnP 알고리즘엔 3가지가 있으며 이 코드에서 설정한다.

src/dope/inference/cuboid.py

> cuboid 클래스.

원본 논문 및 원본 소스코드 위의 추론 과정 3번에서는 PnP 알고리즘을 적용하기 위해 살아남아야 할 belief map의 최소 개수를 4개로 설명하고 있으나, 실제 코드에서는 최소 6개를 필요로 하여 수정하였음.

실시간으로 로봇의 DOPE 추론을 위해서는 ROS 상에서 camera 노드 및 dope 노드를 활성화해야 하나, ROS 환경 적용은 아직은 필요하지 않아 시행해보지 않았음. 대신 추론이 잘 되는지 확인하기 위하여 src/dope/inference/infer.py 코드를 만들어 확인하였음.

[infer.py 소스코드 설명]

해당 코드는 몇 장의 테스트 이미지에 대한 추론 결과를 파일로 출력한다. (1. cuboid가 그려진 추론 결과 이미지 (\*.png), 2. 추론 결과 (results), 3. 변환 결과 (BMRR에서 사용, 무시해도 됨))

입력 파라미터는 없고 소스코드 내부에서 수정한다. 따라서 그냥 실행하면 된다.

소스 코드 내의 SETTINGS 부분에서 다음을 설정한다.

weight\_path 학습된 dope\_network의 가중치 경로

infer\_save\_path 추론 결과를 저장할 폴더 이름

sample\_image\_paths 테스트 이미지의 경로를 담을 리스트

sample\_gt\_paths 테스트 이미지에 해당하는 정답 경로를 담을 리스트.

cuboid\_dimension\* 학습데이터에 있는 정보로, 추론할 물체를 감쌀 cuboid의 크기 정보

config\_detect.\* 후처리에 사용할 threshold 값들(thresh\_angle, thresh\_map, thresh\_points 3가지가 중요해 보임)

camera\_matrix 테스트 이미지의 카메라 정보 행렬. PnP 알고리즘에 해당 정보가 사용된다. (학습 데이터에도 이 정보가 포함되어 있으므로, 테스트 이미지의 카메라 정보가 학습 데이터의 카메라 정보가 동일한 경우 학습 데이터에 있는 카메라 정보와 똑같이 설정하면 된다)

pnp\_solver\* 추론할 물체의 이름과 추론할 물체에 대응하는 cuboid\_dimension\*을 파라미터로 넣어준다.

이후 INFERENCE 부분에서 추론 및 파일 출력을 실행한다.

src/dope/inference/turtle\_temp와 src/dope/inference/div\_temp에 추론 결과 예시를 넣어놓았다. 전자는 학습 데이터 자체에 문제가, 후자는 10에폭만으로 나타난 결과이기에 성능이 썩 좋지는 않다.

**Inference.py 코드 상세 설명 (08/31)**

src/dope/inference/inference.py 코드는 단순히 내부 동작을 파악하기 위해 제가 새로 작성한 코드입니다.

처음의 두 함수 find\_object\_poses, find\_objects는 기존 src/dope/inference/detector.py 코드의 ObjectDetector 클래스에서 가져온 것입니다.

find\_objects 함수에서 DopeNet에서 나온 belief maps와 affinities를 통해 objects를 추론하고, 그 결과를 가지고 find\_object\_poses에서 pnp 알고리즘을 통해 각 object의 pose를 추론합니다.

클래스 Draw는 nodes/dope 코드의 클래스 Draw를 복사한 것입니다.

해당 클래스는 큐브를 그려주기 위한 부분입니다.

이후 SETTINGS 부분이 나오게 됩니다.

weight\_path에 학습한 weight 경로를 할당합니다.

infer\_save\_path에 추론 결과를 저장할 폴더 경로를 할당합니다.

그런 다음 for문을 돌면서 리스트 sample\_image\_paths와 .sample\_gt\_paths에 각각 입력 이미지와 그에 대응하는 정답 (json 파일)을 append합니다.

이후 큐브의 크기에 대한 정보를 cuboid\_dimension 변수에 할당하게 됩니다. 각 물체마다 큐브의 크기가 다른데, 이는 추론하는 것이 아닌 이미 알고 있는 정보입니다. 해당 정보는 학습 데이터의 object\_settings.json 파일에 있으며, 그대로 복사하여 할당하였습니다. 여러 종류의 물체를 사용할 경우 그 수에 맞게 변수 이름 뒤에 1, 2 등을 붙여 할당했습니다.

각도변환 정보는 무시하셔도 됩니다.

그 다음 tresholds 값을 설정하게 됩니다. 각각이 의미하는 정보는 저도 모릅니다. 다만 thresh\_angle, thresh\_map, thresh\_points가 중요하며, 주석으로 기존의 값을 표시해 놓았습니다.

그 뒤에 transform 정보는 코드 내부에 있던 걸 그냥 복붙한 것입니다. (그대로 두시면 됩니다)

그 다음 net에 DopeNet을 할당합니다.

이후 camera\_matrix를 할당하게 되는데, 원래 dope는 실시간으로 카메라 상으로 돌리기 때문에 카메라의 초점 거리와 같은 정보가 필요합니다. 그러나 inference 코드는 그냥 이미지를 한개씩 받아서 추론을 보기 위한 과정이기 때문에, 입력되는 이미지 상에서의 카메라 정보를 할당하면 됩니다. 코드에 쓰여있는 정보는, 추론해볼 이미지로 학습 데이터를 사용했기 때문에 학습 데이터의 \_camera\_settings.json 파일에 나온 바와 같습니다.

그 다음 pnp\_solver를 할당합니다. pnp 알고리즘에 이용되는 것으로, 물체의 이름(**이름은 아무렇게나 설정하면 됩니다. 실제 ROS 노드를 통해 돌리면 알아서 할당될 것들이나, 여기서는 추론이 내부적으로 어떻게 돌아가는지를 수동적으로 확인하기 위해서 일일이 하드코딩 하는 것입니다.**)과 앞서 할당한 cuboid\_diumension을 각각 파라미터로 받습니다.

dist\_coeffs는 코드 내부에 있던 걸 그냥 복붙한 것입니다. (역시 그대로 두시면 됩니다)

그 후 pnp\_solver 각각에 알맞는 camera\_matrix를 할당하고, dist\_coeffs를 할당해줍니다.

마지막의 plt figure 사이즈는 히트맵을 출력하기 위해 시도한 흔적으로, 무시하셔도 됩니다.

그 다음엔 INFERENCE 부분에서 SETTING에 따라 실제로 추론을 수행합니다. for문을 돌면서 sample\_image\_paths에서 이미지를 하나씩 가져오며 추론합니다. 많은 코드들이 코드 내부적인 부분들을 그대로 복붙한 것이기에, 아주 상세한 과정은 저도 조사해 봐야 알고, 현재는 모릅니다.

주석이 쳐져있는 부분들은 히트맵을 출력하기 위한 시도였거나 디버깅의 흔적으로 무시하셔도 됩니다.

# Find objects from network output 부분의 if 문은, 추론하려는 물체의 종류가 여러 개일 경우 SETTINGS에서 설정한 pnp\_solver를 추론할 입력 이미지별로 따로 할당하기 위함입니다.

위치 정확도와 각도 정확도는 성능을 보기 위해 직접 작성한 것입니다. 각도 정확도의 경우 euler로 변환하는 부분은 터틀봇에 특화된 부분이므로 크게 의미두지 않으셔도 됩니다.

그다음 큐브를 그리고 이미지를 저장합니다.

(z, x, pitch) -> (x, y,m theta)로 변환하여 posexxx.txt 형태로 저장하는 부분 또한 터틀봇을 위한 것으로 무시하셔도 됩니다.

마지막에는 평균 오차를 출력합니다.

**ros로 실행 guide**

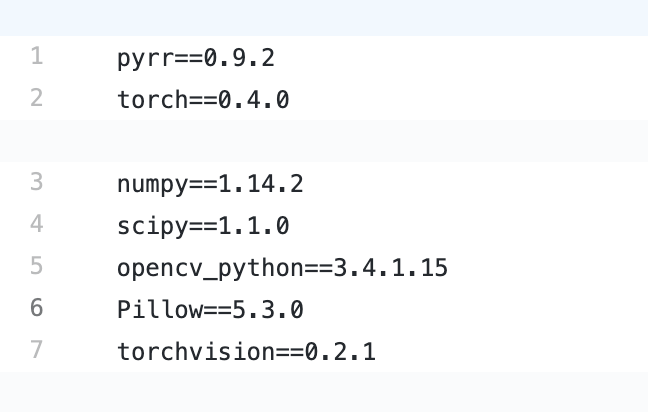
재인씨께 들은거 문서화 합니다.

실행해본 패키지 버전

cuda 10.0

pytorch 1.0

이건 dope requirement.txt(아래사진) 을 참고.



config 파일 설정

설명 보고 맞추면됨..

realsense rectified 로 쓰는게 좋은듯

weight 는 각각 물체별로 다르게 줘야할.. 수도?

object 마다 size, color 등 설정

roslaunch dope.launch 돌리면 됨.

rostopic : image\_topic/dope/pose\_xxxx(object명) 으로 6d 얻을 수 있음.