Multi-view self-supervised deep learning framework for solving 6D pose estimation problem

Li Zhidong 12011506
Ren Shize 12012818
Jiang Meng 12011603
Zeng Yuqi 12011811
Zhao Xuda 12010325

***Abstract*—Automatic logistics is a very hot topic recently,and picking is a significant part to implement this technology. To study further, we searched the past project of a famous competition: Amazon Picking Challenge (APC). From the projects we chose a Multi-view self-supervised learning for our project. We combined the methods in DenseFusion: 6D Object Pose Estimation by Iterative Dense Fusion to complete estimating 6D poses of known objects from RGB-D images.our method outperforms previous approaches in several datasets, and is significantly more robust against occlusions.We also completed A complete deep learning algorithm and used ROBOSUITE to perform the whole process simulation of the grasping process of the manipulator.This provides a basis for us to use this technique in real scenarios.In the future we can create larger datasets and use images and videos to refine our project and train different data models.**

We completed the training of the linemod dataset through 60 training sessions, with an accuracy of 0.9535. We also visualized the 6D pose estimation and re evaluated the advantages and disadvantages of several pose estimation methods for different object recognition. The average of the evaluation results for each method is as follows, PoseCNN-0.68,PoseCNN+Multiview-0.75,PoseCNN+ICP-0.87,densefusion-0.92,which highlighting the excellence of densefusion.

I. Introduction

In the era of globalization and information age, logistics is becoming increasingly complex and fast, posing significant challenges to the warehousing industry. Therefore, the automation of warehousing systems is crucial. One way to improve the quality of stored materials is by combining intelligent warehouse image recognition systems with RFID technology.[7]

Accurate acquisition of the 6D pose of an object is particularly important. We draw inspiration from the Amazon Picking Challenge (APC), where they utilize a fully convolutional neural network to segment and label multiple views of a scene. They fit pre-scanned 3D object models to the resulting segmentation to determine the 6D object pose.[11] We also refer to the method of Article DenseFusion.

DenseFusion is a heterogeneous architecture that processes the two data sources individually and uses a novel dense fusion network to extract pixel-wise dense feature embedding, from which the pose is estimated.[9] In another paper, they introduce a Normalized Object Coordinate Space (NOCS) to directly infer the correspondence from observed pixels to this shared object representation (NOCS) along with other object information[5].We also refer to methods such as DeepDPose[3],Pix2Pose[6], and PoseNet[10].

To overcome limitations in cluttered scenes and real-time applications, we choose the DenseFusion method: "6D Object Pose Estimation by Iterative Dense Fusion." We train the model using deep learning and computer vision techniques and utilize Robosuite to visualize the object grasping process. We create a new grasping environment in Robosuite, modify the "sample" function to determine object positions, and simulate real-life scenarios. Using the aforementioned learning algorithms, we obtain a 6D coordinate for each object to be grasped and moved. Finally, the program outputs a figure of the robot's trajectory.

Regarding the 6D pose estimation algorithm, the first step involves training the model, which consists of training the DenseFusion model and the iterative refinement model. We visualize the results to assess the model's performance in both real and simulated environments. Furthermore, we compare DenseFusion with other methods, utilizing the YCB dataset and a toolbox to plot the 3D accuracy threshold curve.

In this paper, we propose a novel approach for estimating 6D poses of known objects from RGB-D images and we have completed a comprehensive deep learning process. Our method outperforms previous approaches in various datasets and exhibits greater robustness against occlusions.

II. Method

1) A brief introduction to DenseFusion:

DenseFusion is a network proposed by Li Feifei et al. in 2019 for 6D pose estimation. It is an end-to-end structure that takes RGB-D data as input and predicts the 6D pose of an object as output. The main contribution lies in proposing a pixel-level dense fusion approach for integrating color and geometric features.[9]

In fact, it uses the traditional network structure for processing rgb-d data, only replacing the depth map with pointnet. The prediction results of the first network posenet did not perform well on objects with occlusion, so a new network was trained with the same structure, but the input was changed to the output of the first network.

The network consists of two stages:

Fusion of color and depth information from RGB-D images, followed by learning-based mapping to another feature space. Pose prediction is performed using this feature vector. Iterative pose refinement using a cyclic iterative optimization method.

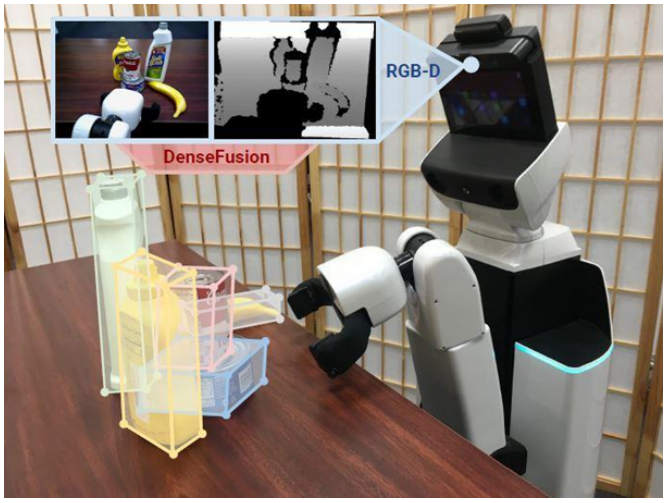

Fig. 1. An end-to-end deep network model for 6D pose estimation from RGB-D data, which performs fast and accurate predictions for real-time applications such as robot grasping and manipulation.

Related work:

Pose from RGB images. Classical methods rely on detecting and matching key points with known object models[2]. Newer methods address the challenge by learning to predict the 2D key points[1] and solve the poses by PnP[4]. Though prevail in speed-demanding tasks, these methods become unreliable given low-texture[8] or low-resolution inputs. Other methods propose to directly estimate objects pose from images using CNN-based architectures. Many such methods focus on orientation estimation. Our method leverages both image and 3D data to estimate object poses in 3D in an end-to-end architecture.

Pose from depth / point cloud. Recent studies have proposed to directly tackle the 3D object detection problem in discretized 3D voxel spaces. These methods are often prohibitively expensive: takes nearly 20 seconds for each frame. More recent 3D deep learning architectures have enabled methods that directly performs 6D pose estimation on 3D point cloud data.

Pose from RGB-D data. Classical approaches extract 3D features from the input RGB-D data and perform correspondence grouping and hypothesis verification. However, these features are either hardcoded or learned by optimizing surrogate objectives such as reconstruction instead of the true objective of 6D pose estimation. Newer methods such as PoseCNN directly estimates 6D poses from image data. Further fuses the depth input as an additional channel to a CNN-based architecture. However, these approaches rely on expensive post-processing steps to make full use of 3D input.

Our method is most related to PointFusion, in which geometric and appearance information are fused in a heterogeneous architecture. We show that our novel local feature fusion scheme significantly outperforms PointFusion's naive fusion-by-concatenation method. In addition, we use a novel iterative refinement method to further improve the pose estimation.

The previously considered FCN+ICP method faced issues during the reproduction stage due to the use of an outdated CUDA version in the original paper.[11]

2) DenseFusion's simple usage:

Dataset Selection:

The YCB dataset has a large file size of 265GB and was only used for evaluating the results in the later stages of the experiment.[10]The YCB Video dataset was created based on the YCB dataset, and 21 objects were selected from the YCB dataset (reason for selecting these 21 objects: high quality 3D models and good visibility in depth). Select 3-9 objects from 21 to create a real indoor scene, and then use RGBD cameras to take physical photos to create 92 videos. All videos in the entire dataset contain 133827 frames. Finally, a semi-automatic method was used for 6D pose annotation.

The Linemod dataset is a widely used object recognition dataset, which includes 15 objects, including bottles, cups, televisions, keyboards, etc., with multiple perspectives of RGB D images and 3D models. These images and models can be used to train and test object recognition algorithms

The characteristic of the Linemod dataset is that each object has a unique color and texture, making it easy to recognize in the image. In addition, this dataset also provides true pose annotation of objects, that is, their position and direction in 3D space, making it suitable for research on object pose estimation.

We trained our model using the provided Linemod dataset (5.8GB). For creating our own dataset, we utilized the Object-DatasetTools [12]

(https://github.com/F2Wang/ObjectDatasetTools.git)tool.

Thanks for providing us with the depth camera D435i.

The "data" folder consists of 13 sub-folders, each corresponding to a specific category. Here is the content within each category:

1. depth(directory): Depth maps.

2. mask(directory): Masks of the target objects, representing the segmentation results.

3. rgb(directory): RGB images.

4. gt.yml(file): Contains the rotation matrix, translation matrix, standard bounding box of the target object, and the class label of the object for each captured image.

5. info.yml(file): Intrinsic parameters of the camera used for capturing each image, along with the scale factor for depth.

6. test.txt(file): Predefined test dataset split.

7. train.txt(file): Predefined training dataset split.

The "models" folder contains PLY files, where each file represents point cloud information. The point cloud data in these files is based on the reference image "linemod/-Linemod_preprocessed/data/xx/rgb/0000.png." With this reference image's point cloud data (including the complete point cloud of the target object), we can calculate the corresponding

point cloud data for other captured images using their camera parameters (rotation and translation matrices).

The "models_info.yml" file contains information about the radius, starting values, and size ranges along the x, y, and z axes for each target point cloud model.

During the training of the model, the most accurate data is used to train the model. Therefore, during both training and validation, the "mask" folder containing the most accurate masks (located at "Linemod_preprocessed/data/mask") is used as the segmentation labels.

During testing, it is desired to simulate real-world scenarios as closely as possible. In the actual application, DenseFusion network predicts the pose of segmented objects. Therefore, an additional segmentation network is required in practical applications. The "segnet_results" folder contains the images segmented by the semantic segmentation network. This segmentation output is intended to be closer to real-world scenarios, and it differs slightly from the standard masks used during training and validation.

Obtain our Linemod images:

It is important to note that the resolution is 640x480, and the coordinates need to be converted from meters to millimeters (by timing 1000).

The training process consists of two parts:

(i) training the DenseFusion model and (ii) training the iterative refinement model. In this code, the training of the DenseFusion model will be conducted first. Once the average test distance results (ADD for non-symmetric objects and ADD-S for symmetric objects) are below a certain margin (refine_margin), the training of the iterative refinement model will start automatically, and the DenseFusion model will be fixed. You can modify this margin to obtain better DenseFusion results without performing refinement, but it will be inferior to the final results achieved after iterative refinement.

Before training, I conducted a validation using the provided models from the linemod_preprocessed dataset, and the success rate was 0.9539.

We performed two training sessions for pose refinement using their pre-trained models without refinement. The training was conducted for a total of 30 epochs, divided into two sessions (epochs 31-61 and 61-91). The refine_margin, which determines when the training of the pose refinement model starts (i.e., when the average distance is below the refine_margin), was set to 0.013 (default) for the first session and 0.008 for the second session.

During the first training session, the average distance (Avg dis) did not fall below the refine_margin. As a result, the first trained model obtained was for pose refinement, not pose estimation.

During the second training session, both the pose estimation model and the pose refinement model were loaded simultaneously. This means that the training process incorporated both models to refine the pose estimation.

3) Environment creation in Robosuite:

The general step of create a new grasping environment is:

1. Encapsulate the environment into a class,

2. Register the environment at__init__.py,

3. Create a demo to instantiate the environment.

An environment includes several necessary functions:

1. def reward(self, action): This function defines the reward for the task.

2. def staged_rewards(self): Helper function to calculate staged rewards based on current physical states.

3. def _load_model(self): Loads an xml model, puts it in self.model, including customize and initialize arena and objects.

4. def _setup_references(self): Sets up references to important components.

5. def _reset_internal(self): Resets simulation internal configurations.

Object setting:

In the environment file, we can create models by import XML package. Also we can customize our own material by import CustomMaterial package.

```
from robosuite.models.objects import BoxObject
from robosuite.utils.mjcf_utils import
    CustomMaterial
```

For example, we can customize a material named "redwood" by these code:

```
redwood = CustomMaterial(
    texture="WoodRed",
    tex_name="redwood",
    mat_name="redwood_mat",
    tex_attrib=tex_attrib,
    mat_attrib=mat_attrib,
)
```

And we can create a cube called "cubeA" with size = [0.02, 0.02, 0.02] using the redwood material:

```
self.cubeA = BoxObject(
    name="cubeA",
    size_min=[0.02, 0.02, 0.02],
    size_max=[0.02, 0.02, 0.02],
    rgba=[1, 0, 0, 1],
    material=redwood,
)
```

For example, we created an environment with three cubes on a table arena, between which there are a small red cube, a large green cube and a large blue cube. We need to create an array to represent the cubes.

```
cubes = [self.cubeA, self.cubeB, self.cubeC]
```

Register and initialize the environment:

After creating a new environment, we must register the environment at __init__.py. We can add a "import" function into it. For example, if we create a new environment called "Env1", then we can use the function below: from robosuite.environments.manipulation.Env1 import Env1 The registered environment could be initiated in another python

file. By using function “env = suite.make()”, we can initiate the environment and render it.

Generally, we can adjust the environment by these factors below:

1. env_name: Change the registered environment.
2. robots: Change the registered robot (arm).
3. render_camera: Change the position of the camera that renders the environment and display.

4) Manipulation in Robosuite:: Set the position for the 3 objects created in the environment:

```
positions = np.array(
[[x1, y1, z1], [x2, y2, z2], [x3, y3, z3]])
```

Set 2 PD controllers: PD_Controller_Open for the gripper open and the PD_Controller_Closed for the gripper closed:

```
def PD_Controller_Open(target, eef_pos,
    last_eff_pos):
   p = 8
   d = 7
   action = -np.ones(4)
   target = ori_pos + target
   action[0:3] = (target - eef_pos) * p -
       abs(eef_pos - last_eff_pos) * d
   return action

def PD_Controller_Close(target, eef_pos,
    last_eff_pos):
   p = 8
   d = 7
   action = np.ones(4)
   target = ori_pos + target
   action[0:3] = (target - eef_pos) * p -
       abs(eef_pos - last_eff_pos) * d
   return action
```

Set 5 steps for the robot arm panda manipulation, repeat if there is other objects:

Step1: move to target object
Step2: close gripper and grasp target object
Step3: move to the end point
Step4: open gripper
Step5: repeat the above process

```
##Step1
for i in range(50):
   line = [x1 + 0.1, y1, -0.15]
   eef_pos = obs['robot0_eef_pos']
   action = PD_Controller_Open(line, eef_pos,
       last_eff_pos)
   track = np.append(track, eef_pos - ori_pos)
   obs, reward, done, info = env.step(action)
   last_eff_pos = eef_pos
   env.render()
   #env.robots.grasp = not env.robots.grasp
##Step2
for i in range(50):
   line = [x1 + 0.1, y1, -0.2]
   eef_pos = obs['robot0_eef_pos']
   action = PD_Controller_Close(line,
       eef_pos, last_eff_pos)
   track = np.append(track, eef_pos - ori_pos)
   obs, reward, done, info = env.step(action)
   last_eff_pos = eef_pos
   env.render()
##Step3
for i in range(50):
   line = [-0.1, 0.25, 0.2]
   eef_pos = obs['robot0_eef_pos']
   action = PD_Controller_Close(line,
       eef_pos, last_eff_pos)
   track = np.append(track, eef_pos - ori_pos)
   obs, reward, done, info = env.step(action)
   last_eff_pos = eef_pos
   env.render()
##Step4 Open
for i in range(50):
   line = [-0.1, 0.25, 0.2]
   eef_pos = obs['robot0_eef_pos']
   action = PD_Controller_Open(line, eef_pos,
       last_eff_pos)
   track = np.append(track, eef_pos - ori_pos)
   obs, reward, done, info = env.step(action)
   last_eff_pos = eef_pos
   env.render()
```

Output a figure of robot track:

```
track = track.reshape([600, 3])
plt.figure(0)
plt.scatter(track[:, 1], track[:, 2])
plt.show()
```

III. Experiment

1) Compare the model provided by densefusion with the one we trained ourselves:

The validation results are as follows, and they are very close to the success rate obtained from the validation of the model provided by the Linemod_preprocessed dataset.

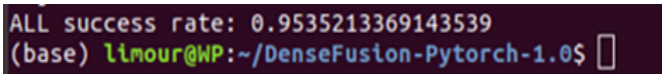

Fig. 2. Evaluation results

Final training duration: 60 epochs (30 + 30), with a total training time of 31 hours.

Fig. 3. Success rate

The first training only requires loading the dataset to be used for training, and the previously trained pose estimation model can be directly used for further training. Due to the small epoch setting, the Avg dis value can be changed to be lower than refine_ margin. Through 60 training sessions, the pass rate was increased from 0.8062 to 0.9535, which is close to the 0.9538 obtained from the original 500 training sessions. This proves that the set epoch and other related parameters are reasonable, and also proves that the given no refine model has certain universality and can be used for training other item datasets.

Visualization:

The visualization process involves outputting the predicted results, which helps in understanding how the model performs in real-world scenarios and simulation environments. One important function in the 'draw_linemod.py' script is 'mask_to_bbox'. This function takes a binary mask image 'mask' as input, representing the object's location. It converts the binary mask to an unsigned 8-bit integer type and uses the 'findContours' function from OpenCV to find the contours in the mask image. Using the obtained contours, it calculates and returns the minimum bounding rectangle [x, y, w, h] that encloses the object. Here, x and y are the coordinates of the top-left corner of the rectangle, and w and h represent the width and height of the rectangle.

Another function, 'get_bbox', takes a list of bounding boxes 'bbox' as input, where each bounding box is represented as [x, y, w, h]. This function adjusts the coordinates of the bounding boxes to fit within the image's dimensions. Specifically, it restricts the coordinates of the top-left and bottom-right corners of the bounding boxes to stay within the image boundaries, preventing them from going beyond the image borders. Finally, it returns the adjusted bounding box coordinates [y_min, y_max, x_min, x_max].

These functions are used to extract the object's bounding boxes from binary mask images and adjust the bounding box coordinates to fit within the image boundaries. Here is an example showcasing a reasonably predicted object annotation:

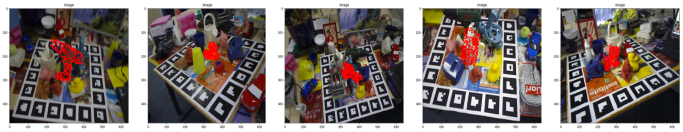

Fig. 4. Results without attitude refinement in the original paper

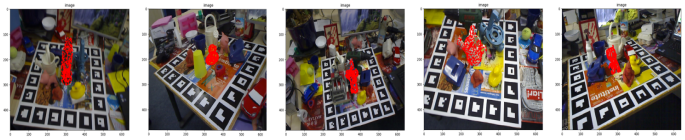

Fig. 5. The visualization results after attitude refinement(In order to show the pose estimation results intuitively, we use a new visualization method)

2) Compare densefusion with several other methods:

YCB-VIDEO Usage:

After completing the basic tasks, we downloaded the 265GB YCB dataset and used the toolbox to plot the 3D accuracy threshold curve for DenseFusion. The results are saved in the "plots" folder (see attachment). The toolbox also provides the "show_pose_results.m" script to visualize the pose results specifically for the YCB dataset.

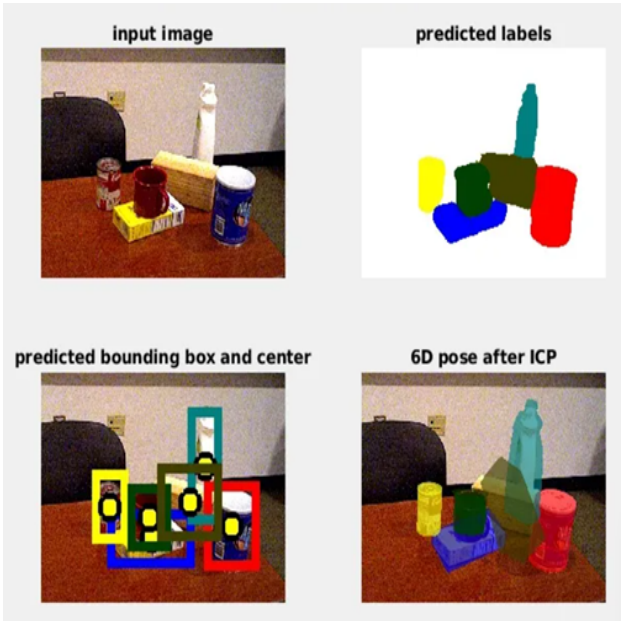

Fig. 6. Picture of toolbox position estimation

Finally, we are evaluated and analyzed densefusion and other commonly used methods using 21 items, and the results are as follows:(The data involved in the picture can be seen in the attachment)

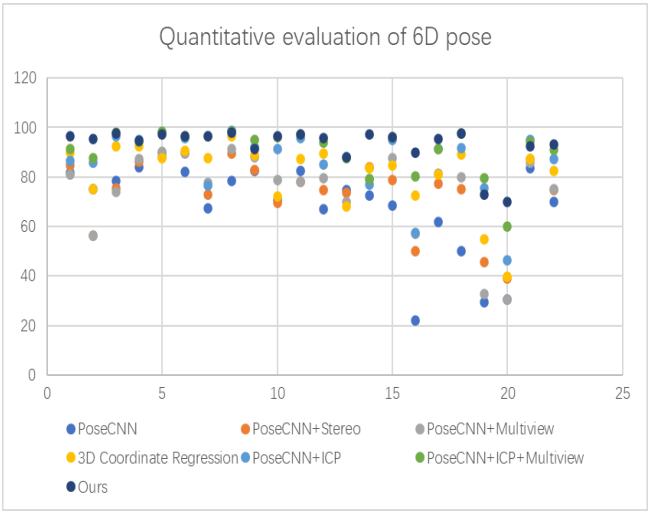

Fig. 7. Quantitative evaluation of 6D pose

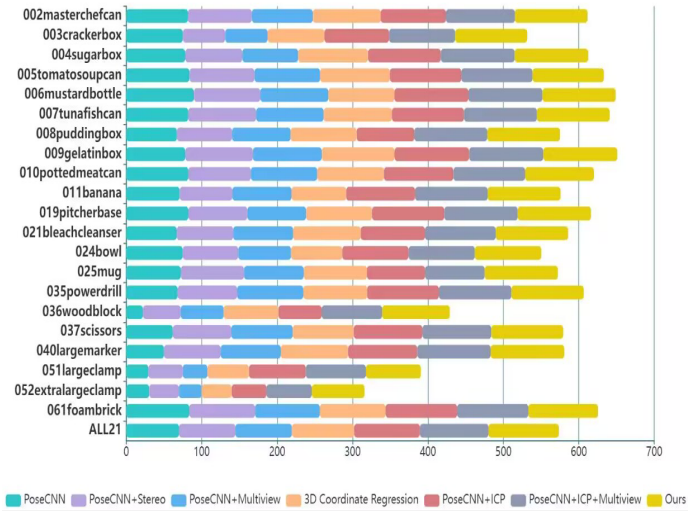

Fig. 8. Evaluation results of different pose estimation methods for different objects

The main advantage of our dense fusion method is its robustness towards occlusions. To quantify the effect of occlusion on final performance, we calculate the visible surface ratio of each object instance (further detail available in supplementary material). Then we calculate how the accuracy changes with extent of occlusion. As shown in Fig, the performances of PoseCNN+Stereo and PoseCNN+ICP degrade significantly as the occlusion increases. In contrast, none of our methods experiences notable performance drop. In particular, the performance of ours only decrease by 2% overall.

As we can see, PoseCNN+ICP and PoseCNN+Stereo fail to estimate the correct pose of the extra-large clamp due to heavy occlusion, whereas our method remains robust. Another challenging case is the clamp in the middle row due to poor segmentation. Our approach localizes the clamp from only the visible part of the object and effectively reduces the dependency on accurate segmentation result.

However, the overall complexity of the YCB scenario is not sufficient, and the accuracy of item identification in chaotic backgrounds such as construction sites and warehouses still needs further evaluation.

3) Placing and grasping tests in Robosuite:

We can run the initiation file to display the rendered environment. For example, I created an environment with three cubes on a table arena. Between the cubes there are a small red cube, a large green cube and a huge blue cube, and the render camera is “frontview”.

Here is the image generated:

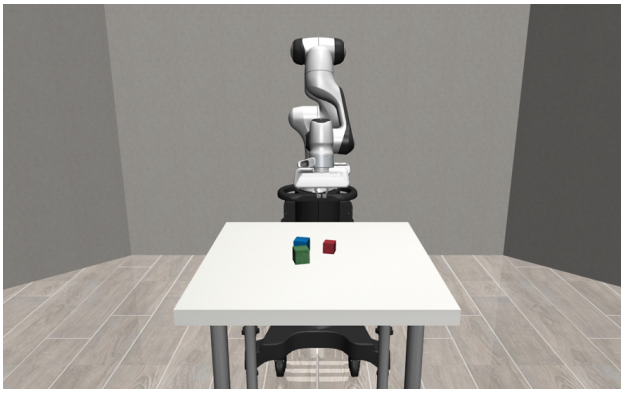

Fig. 9. Three cubes on tablearena in random with frontview

And here is the image generated by using “agentview”:

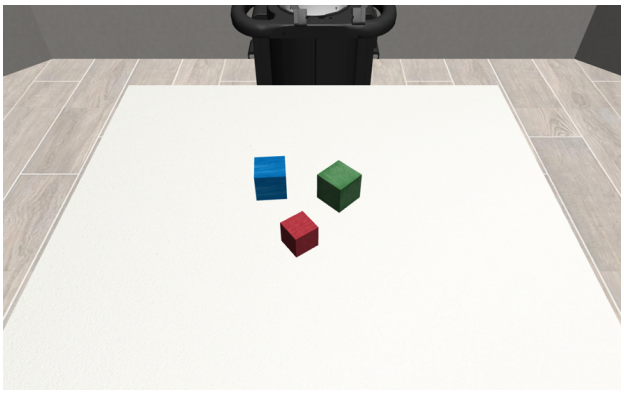

Fig. 10. Three cubes on tablearena in random with agentview

Determine the position of objects:

The position of the cubes is random, but we can change it by rewriting the function “sample”:

To determine the position of the cubes, we have another step to do: rewrite the “sample” function, for example, we set the cubes at position [0, -0.08, 1], [0, 0, 1], [0, 0.08, 1]:

```
def sample(self, fixtures=None,
    reference=None, on_top=True):
   placed_objects = {}
   for i, obj in
       enumerate(self.mujoco_objects):
      placed_objects[obj.name] =
          (self.positions[i], np.array([0.5,
          -0.5, 0.5, -0.5]), obj)
   return placed_objects
positions = np.array(
   [[0, -0.08, 1], [0, 0, 1], [0, 0.08, 1]])
```

The position of the cubes changes from random to a determined place, and here they are placed at [0, -0.08, 1], [0, 0, 1], [0, 0.08, 1]. Thus, we can simulate the situation of the real world.

Manipulate the robot arm:

In this section, we use Robosuite to simulate the robot arm to grasp objects. With the learning algorithms mentioned before, we can get a 6D coordinate for every object to be

grasped and moved. Because the gripper is simple and robust, we only need 3D coordinates to simulate the whole grab action. In this example, we only need x and y coordinates because every object is on the table.

Step1: move to target object
Step2: close gripper and grasp target object
Step3: move to the end point
Step4: open gripper
Step5: repeat the above process

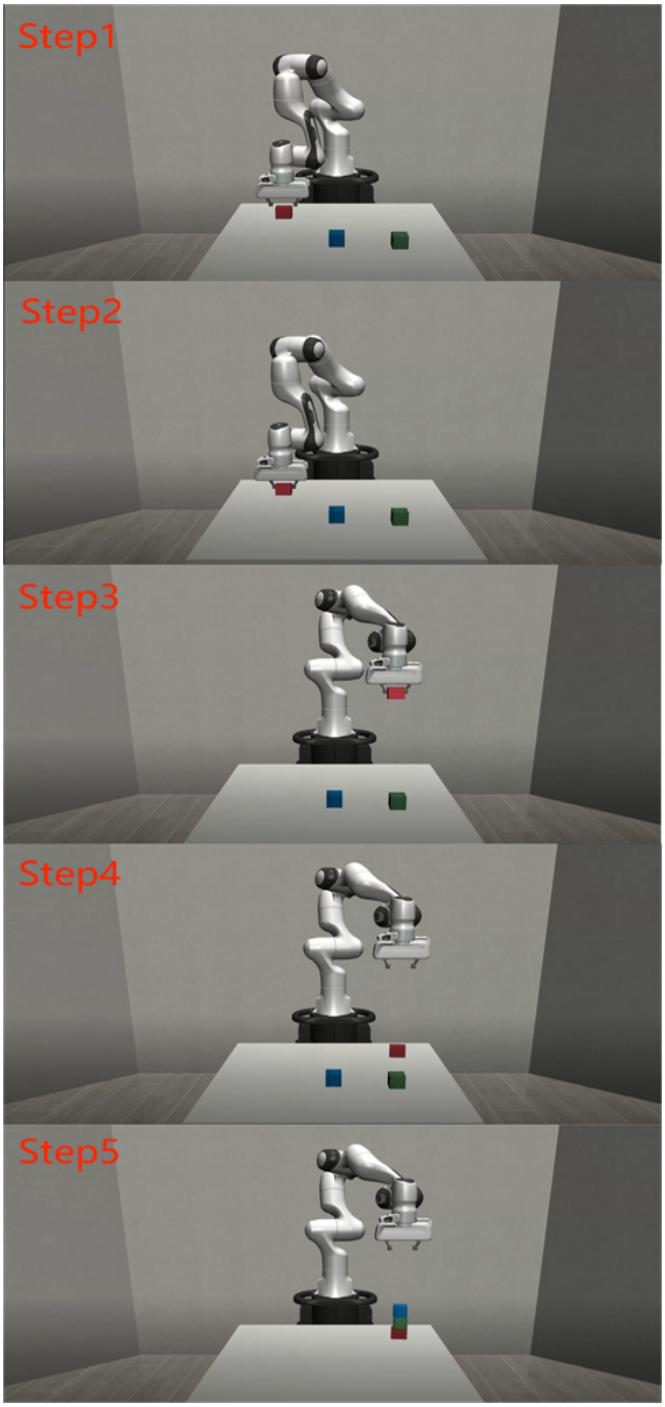

Fig. 11. Steps of manipulation the robot arm

In one picture, these steps work like the figure below:

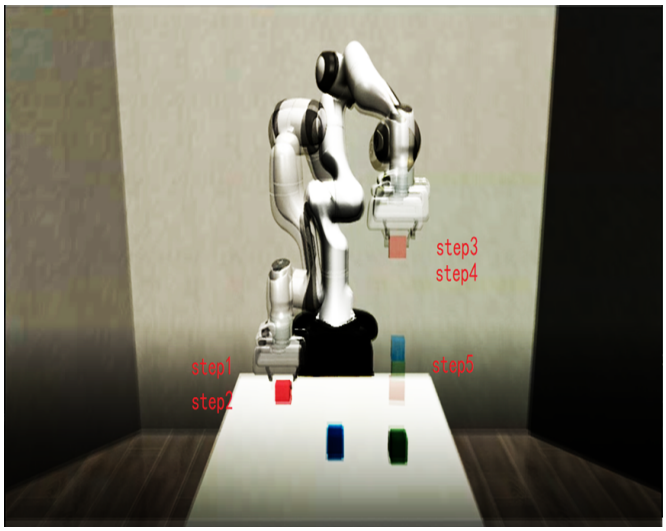

Fig. 12. Steps of manipulation the robot arm in one picture

After all above steps, the program will output a figure of the robot track.

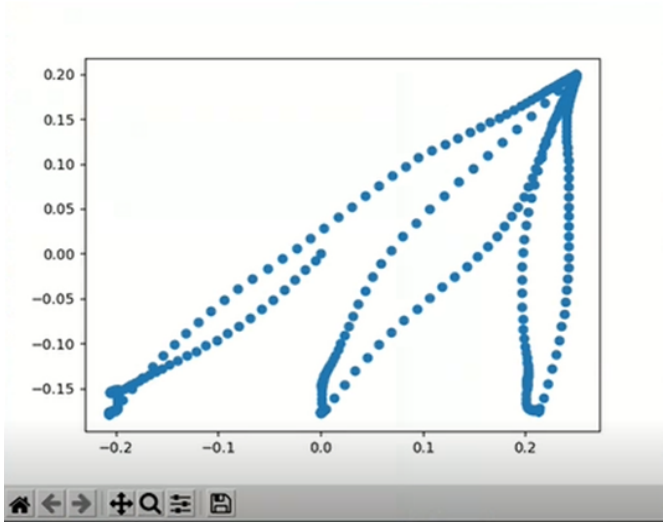

Fig. 13. The output figure of the robot track

Other: grasp irregular object

Besides, we also try some other irregular objects, including some food and composite. If the gripper is suitable for grasping the objects, the objects can be manipulated as expected. (like fig A) However, if the gripper cannot hold the objects, the simulation would fail like fig B and fig E.

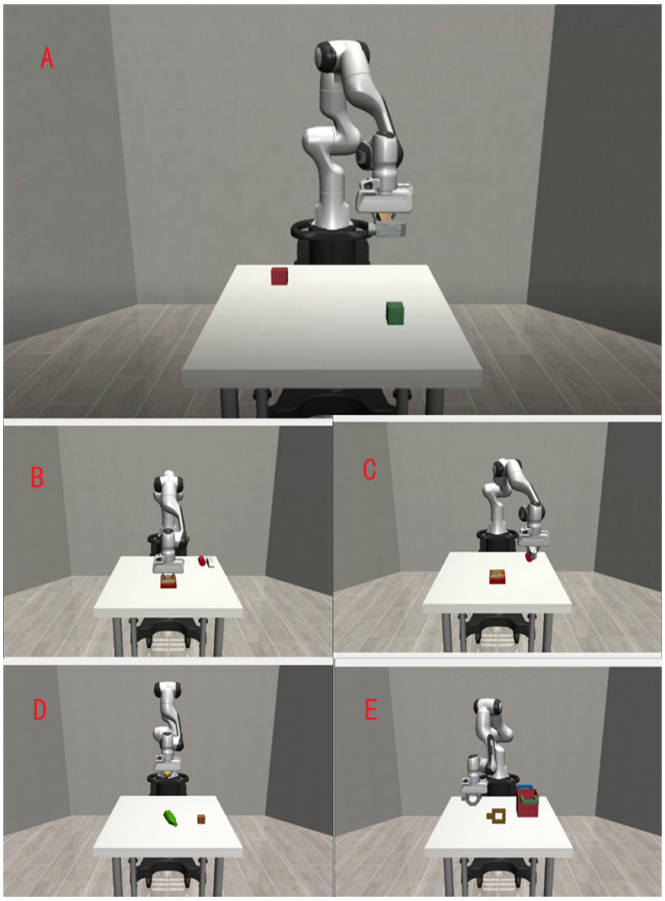

Fig. 14. Steps of grasping irregular object

IV. Conclusion

We used a novel approach to estimating 6D poses of known objects from RGB-D images. Our approach fuses a dense representation of features that include color and depth information based on the confidence of their predictions. With this dense fusion approach, our method outperforms previous approaches in several datasets, and is significantly more robust against occlusions.

We completed a complete deep learning process, from dataset selection, dataset production, adversarial algorithm selection, to environment configuration and training, and finally visualization processing. Simultaneously experiencing the convenience of many CV tools. For project expansion, we can create larger datasets of our own and train models for different scenarios through images and videos. And I hope to have the opportunity to estimate the 6D posture and apply it to real scenarios, such as rescue robots, industrial robotic arms, etc.

Limitations:1.The computing power of the graphics card was insufficient. In the early stage, only a 1650 graphics card could be used for training. Later, CUDA environment of 30 series graphics cards was configured. Linemod and ycb, as well as self-collected data obtained through depth cameras using Object Dataset Tools for validation. For the linemod dataset, the original paper (densefusion) conducted 500 training iterations, taking into account time constraints (each training session requiring 40 minutes). 2. Without hardware support, you can only use simulation environment to capture and demonstrate. Later, it is possible to train its own data set completely and grasp it on the real robot arm.

Acknowledgments

References

[1] Eric Brachmann, Alexander Krull, Frank Michel, Stefan Gumhold, Jamie Shotton, and Carsten Rother. Learning 6d object pose estimation using 3d object coordinates. *Springer International Publishing*, 2014.

[2] A. Collet, M. Martinez, and S. S. Srinivasa. The moped framework: Object recognition and pose estimation for manipulation. *The International Journal of Robotics Research*, 30(10):1284–1306, 2011.

[3] T. T. Do, M. Cai, T. Pham, and I. Reid. Deep-6dpose: Recovering 6d object pose from a single rgb image, 2018.

[4] M. A. Fischler and R. C. Bolles. Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography. *Communications of the ACM*, 24(6):381–395, 1981.

[5] W. He, S. Sridhar, J. Huang, J. Valentin, and L. J. Guibas. Normalized object coordinate space for category-level 6d object pose and size estimation. *IEEE*, 2019.

[6] K. Park, T. Patten, and M. Vincze. Pix2pose: Pixel-wise coordinate regression of objects for 6d pose estimation, 2019.

[7] S Saha, N. K Bambha, and S. S Bhattacharyya. Design and implementation of embedded computer vision systems based on particle filters. *Computer Vision and Image Understanding*, 114(11):1203–1214, 2010.

[8] M. Schwarz, H. Schulz, and S. Behnke. Rgb-d object recognition and pose estimation based on pre-trained convolutional neural network features. In *IEEE International Conference on Robotics Automation*, pages 1329–1335, 2015.

[9] C. Wang, D. Xu, Y. Zhu, R Martín-Martín, C. Lu, L. Fei-Fei, and S. Savarese. Densefusion: 6d object pose estimation by iterative dense fusion. 2019.

[10] Y. Xiang, T. Schmidt, V. Narayanan, and D. Fox. Posecnn: A convolutional neural network for 6d object pose estimation in cluttered scenes. 2017.

[11] A. Zeng, K. T. Yu, S. Song, D. Suo, and J. Xiao. Multi-view self-supervised deep learning for 6d pose estimation in the amazon picking challenge. *IEEE*, 2017.

[12] Tielin Zhang, Yang Yang, Yi Zeng, and Yuxuan Zhao. Cognitive template-clustering improved linemod for efficient multi-object pose estimation. *Cognitive Computation*, 12(4):834–843, 2020.