Technische Universität Berlin

Industry Grade Networks and Clouds
Sec. TEL 18-1
Ernst-Reuter-Platz 7
10587 Berlin

Master Thesis

Implementation of an automated pipeline for random keypoint detection and evaluation for visual object localization on synthetic and real data

Sourabh Raj
Matriculation Number: 397371

Chair
Industry Grade Networks Clouds

Supervised by
Prof. Dr.-Ing. Jens Lambrecht

Second Supervisor
Prof. Dr.-Ing. Sebastian Möller

Hereby I declare that the thesis submitted is my own, unaided work, completed without any unpermitted external help. Only the sources and resources listed were used.

The independent and unaided completion of the thesis is affirmed by affidavit:

Berlin, 03.01.2020 .....

(Signature)

Abstract

In this paper, an automated pipeline has been devised for synthetic data generation and then training and evaluating keypoint detector to estimate the final pose. An efficient and effective keypoint detection method is an essence of pose-estimation and recent developments in the field of neural networks have made it achievable. However, collecting or generating training images for convolutional neural networks is still a bottleneck and so, in this paper, an application has been setup for generating synthetic data to train CNN for keypoint detection and then evaluating the generated keypoints for pose estimation. In this thesis, CAD-Object based method has been used to generate synthetic images using the Unity-3D tool and further Perspective-n-point to estimate the pose of a calibrated camera.

Zusammenfassung

In diesem Arbeit wurde eine automatisierte Pipeline für die Generierung synthetischer Daten und die anschließende Schulung und Auswertung des Schlüsselpunktdetektors zur Abschätzung der endgültigen Pose entwickelt. Ein effizientes und effektives Schlüsselpunkterkennungsverfahren ist eine Essenz der Posenschätzung, und die jüngsten Entwicklungen auf dem Gebiet der neuronalen Netze haben es erreichbar gemacht. Das Sammeln oder Generieren von Trainingsbildern für neuronale Faltungsnetze ist jedoch immer noch ein Engpass. In diesem Arbeit wurde eine Anwendung zum Generieren von synthetischen Daten zum Trainieren von CNN für die Erkennung von Schlüsselpunkten und zum anschließenden Auswerten der generierten Schlüsselpunkte für die Posenschätzung eingerichtet. In dieser Arbeit wurde eine CAD-Objekt-basierte Methode verwendet, um synthetische Bilder mit dem Unity-3D-Werkzeug und einem weiteren Perspective-n-point zu erzeugen, um die Position einer kalibrierten Kamera abzuschätzen.

Contents

1 Introduction

1.1 Motivation

Pose estimation and visual object localization is the key to robot-based automation in many industries as it enables a robot to interact with or manipulate other objects. The task of pose estimation has been receiving a lot of attention from other fields as well, like the medical field where, for example, a precise pose calculation is needed for a robot to operate a human. Lately, many steps have been taken to popularize this field like, there was a challenge organized by Amazon [Ama17], where robots had to pick objects from a bin. To perform these tasks, the convolutional neural network(CNN) based models are getting popular these days. However, finding a suitable application-specific model to perform end-to-end pose estimation is still a challenge. It takes a lot of effort to find appropriate data to train the neural network and to evaluate the results before using a model into the application. Motivated to tackle these challenges, an application to provide a pipeline for generating synthetic image data, based on CAD models and then training and evaluating CNN for keypoint detection and PnP for object pose estimation has been developed in this thesis.

The recent advancements in the field of CNN have made the task of detecting keypoints easier and more precise but, training CNN is still a challenge because of the unavailability of desired training data. This was another factor that motivated the implementation of this thesis. To cater to the issue of gathering the real annotated data, the approach of using precise synthetic data [TPA+18] to train the neural network has been experimented.

1.2 Scope

The overall goal of this scientific implementation is to develop a pipeline for generating synthetic data to train CNN-based keypoint detector and then to evaluate PnP methods for object localization on synthetic as well as real data. The scope of this thesis is to experiment with the feasibility of such a pipeline to be developed and then to measure the performance of underlying modules for their precision and accuracy.

Multiple tools and technologies like Unity-3D, OpenCVSharp, Python, have been used and based on these, components like CNN, PnP, have been developed to achieve the automation. Tasks that are being performed as part of the scope of this thesis can be categorized into two high-level categories, figure 1.1 shows the overall scope of the application.

- Generating Synthetic Data for training CNN,
- Performing 2D-3D mapping using PnP to extract extrinsic parameters

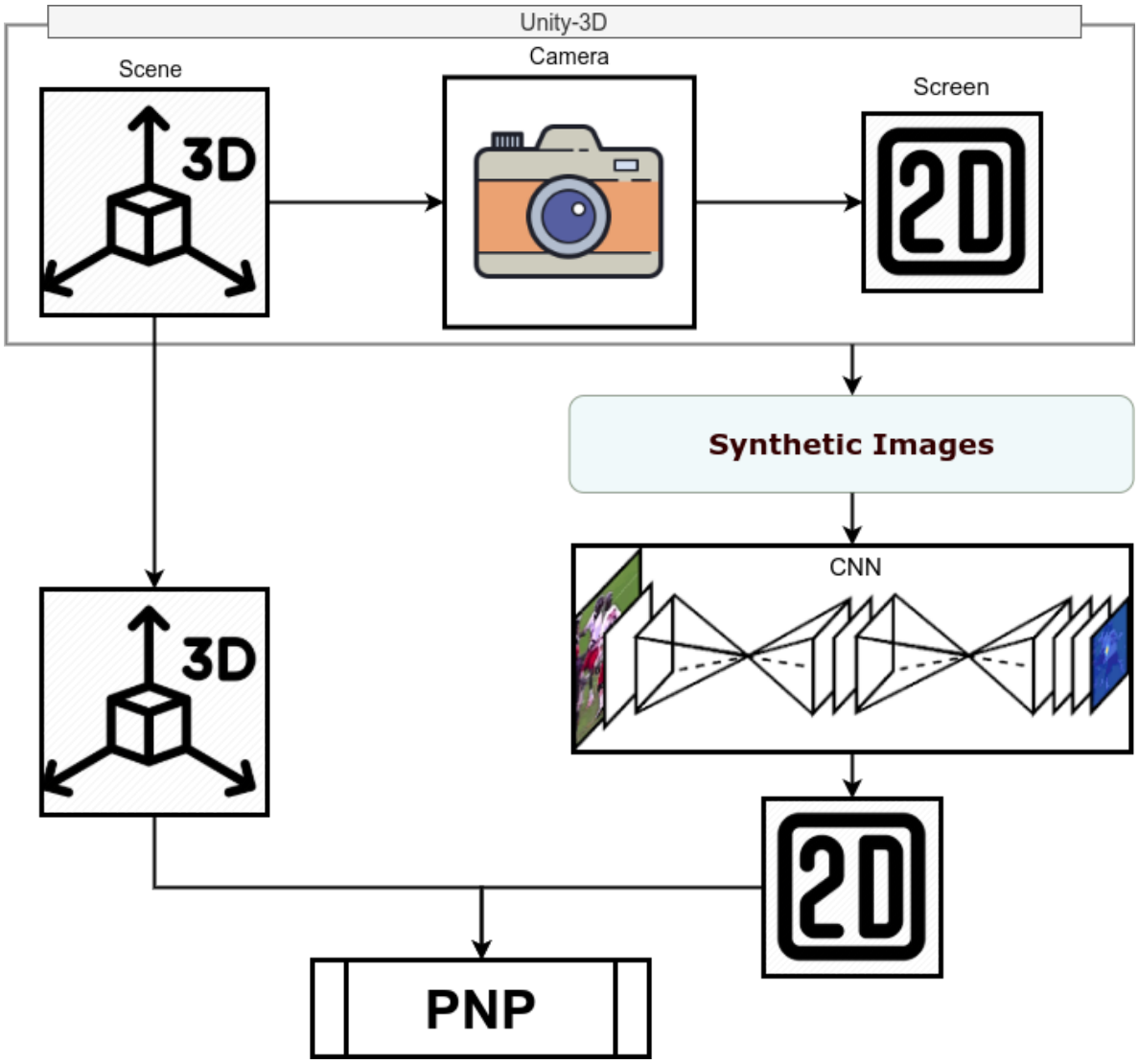


Figure 1.1: Overall scope of the application

1.3 Outline

This thesis paper is outlined in below chapters.

Chapter 2 : Fundamentals and Related work contains the details of fundamental concepts and components used in the thesis and in the end, it compares the approach of this thesis with some of the contemporary works.

Chapter 3 : Technical approach section will give details about the main modules of this thesis along with the implemented technical flows/tasks that have been used to achieve the goal of object localization.

Chapter 4 : Implementation section contains the details of tools and software used in this thesis along with the steps to project and code set up.

Chapter 5 : Performance evaluation part compares the results of overall performance of the modules of this application.

Chapter 6 : Conclusion This section summarizes the paper along with presenting the challenges faced during the implementation of this thesis and the scope of future work.

2 Fundamentals and Related Work

2.1 Overview

2.1.1 Synthetic data in Machine Learning

Recently, Convolutional Neural Networks have been proven to be a very efficient tool in the field of image processing because of its high accuracy and robustness. However, CNN does possess some challenges like designing an efficient CNN architecture and training them. Fortunately, researchers have come up with various architectures that already handle many problems effectively but, training a CNN is still considered as a heavy task and also it is kind of a black box in terms of hidden layer feature mapping. But the biggest of all these challenges is to collect annotated image data to train the CNN.

In the field of CNN, some research papers have employed existing techniques like using structure from motion [ÖVBS17][Wik11c] to collect annotated image data, where training labels (camera poses) are automatically generating from a video of the scene, or gathering data from the internet made available by some organizations. However, some researchers have also used synthetically generated image data to train the model and the results were quite comparable. Another approach is to use mixing synthetic images with real-world images to further enhance the performance of CNN [TPA+18].

The benefit of using Synthetic data is that they are easier to generate, thousands of images can be generated within the timespan of 2 to 3 hours, and it does not require any additional setup. One of the best tools these days to create synthetic data is Unity-3D as it allows the user to manipulate the scene dynamically.

Unity-3D

Unity-3D [Uni19f] is a graphical tool which allows developing end-to-end Camera-based application. It has many inbuilt functions to support 3D-2D mapping which enables users to develop a simulation environment consisting of 3D and 2D world scenarios. It provides a synthetic way to create a scene depicting the real world which can be manipulated and transformed to generate images that closely relate to real images.

This tool is easy to use and has the benefit of being able to be integrated with multiple platforms. It provides users with an interactive interface like drag-drop, key, and clicks, to control and modify the objects in the scene and screen. Unity uses C# as a scripting language which is an object-oriented programming language and has its own benefits. It has implementation of some native methods to support the integration of other programming languages like C, C++, JAVA with C# in the form of DLL/SO, [Uni19b] files. There are plenty of already developed APIs are available to be integrated with Unity, which can easily be found for download on Unityâs asset store. Unity-3D tool is

officially available for Windows and MacOS operating systems however for Linux based OS, it can be used as Unity-editor through Unity-hub.

2.1.2 Object pose estimation

Object pose estimation refers to the computation of position and orientation (6 DoF in total) that fully define the posture of an object in space. In the field of robotics, pose estimation is the most crucial part as it allows robots to interact with other static or mobile objects/robots. So far various approaches exist using fiducial markers and RGB-D cameras to estimate the pose of a non-rigid robot, however lately some RGB camera-based approaches are also getting popular. The basic concept behind using the RGB camera-based approach is to estimate the calibrated camera pose by making use of an object's 3D coordinates and corresponding 2D projections. In this paper, the results of pose estimation using RGB camera has been evaluated on synthetic and real data.

2.2 Contemporary work

Using synthetic data to train neural networks is a field less explored however, some of the organizations are taking lead in this area. Recently a paper has been published by the researcher of Nvidia [TPA+18] about using synthetic data to train a deep neural network and it's success entices some hope for this field to be explored more in coming days.

However, when it comes to localizing visual objects using real data, there are various published papers available with different architectures and procedures. Some of the papers related to pose estimation and their applications are mentioned below along with the highlighted difference from the approach used in this thesis.

1. ***PoseCNN : A Convolutional Neural Network for 6D Object Pose Estimation in Cluttered Scenes [XSNF17]***

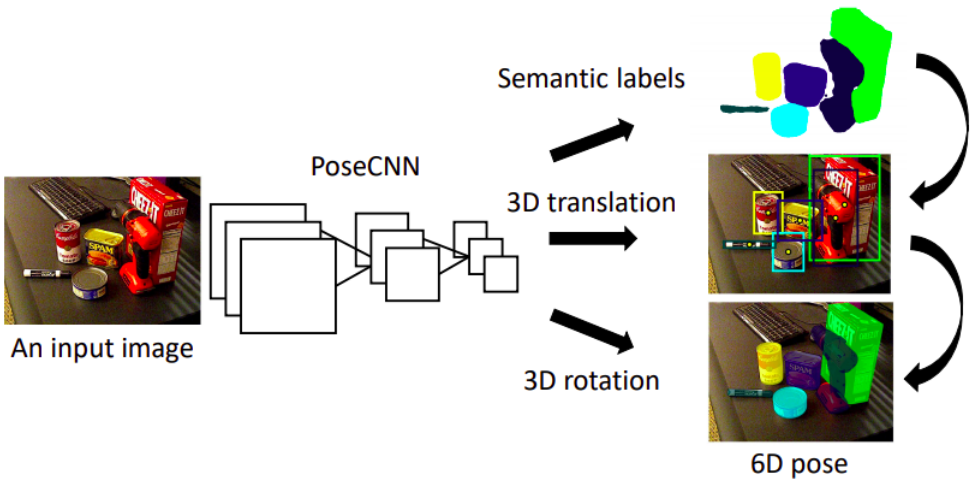


Figure 2.1: PoseCNN architecture. Source [XSNF17].

PoseCNN is very successful in estimating the 6D pose of a known object but it relies on depth calculation of the center of the object from the camera. It regresses over the 3D translation and 3D rotation of the object to predict the final pose of the camera. The architecture is shown in figure 2.1.

Although, PoseCNN is robust to occlusion and symmetry it differs from this thesis in terms of requiring depth measurement. The approach proposed in this thesis does not rely on any depth data or initialization.

2. ***6-DoF Object Pose from Semantic Keypoints [PZC+17]***

This paper has presented an approach to predict the 6-DOF pose using a single RGB image. Technically this approach, predicting 2D keypoints in an image and then estimating pose, is very similar to the one presented in this thesis. However, in the proposed paper they are fitting an optimization problem to estimate the full 6-DOF pose but the application implemented in this thesis has relied on the accuracy of CNN prediction and PnP method of pose estimation. The comparison between PnP and the ICP optimization has been detailed and resulted later in this thesis, section 5.2.3.

3. ***IPose: Instance-Aware 6D Pose Estimation of Partly Occluded Objects [JMP+17]***

In this paper, they have proposed a way to estimate the 6 DOF pose of an occluded known rigid object from the input image. In their implementation firstly they are using image segmentation for object localization, then they map the object's position with each pixel and lastly to predict the 6D pose, they are using ICP and Ransack based geometry optimization.

The approach above differs from the one implemented in this thesis as rather than emphasizing on iterative methods to estimate the pose, the approach in this paper relies on the accuracy of detected keypoints and PnP. Also, in IPose Object Coordinate Regression is a very heavy task as it tries to regress pixel-wise object coordinates, on the contrary, the approach in this paper simply detects the keypoints and then estimate the pose using PnP.

4. ***DOPE: Deep Object Pose Estimation for Semantic Robotic Grasping of Household Objects [TTS+18]***

This paper is research from Nvidea. In this paper, they have proposed a way to train a one-shot deep neural network on synthetic data to estimate the pose of household items in a clutter. They have tried to bridge the reality gap between the synthetic data and real-world data to make the neural network operate

correctly in real-world scenarios. The same approach has been employed in this thesis to train the network and then to estimate the 6-DOF pose, however, the CNN architecture being used in these papers is different as DOPE mainly focuses on household items but this thesis emphasizes on the robot interactions.

Above different existing papers for estimating pose based on real and synthetic data have been compared however, most of these papers suggest that accuracy of pose estimation relies upon the detected keypoints given an image. Below some of the architecture of already available convolutional neural networks for keypoint/skeleton detection have been compared with the architecture used in this thesis.

1. ***Introduction to Camera Pose Estimation with Deep Learning [SF19]***

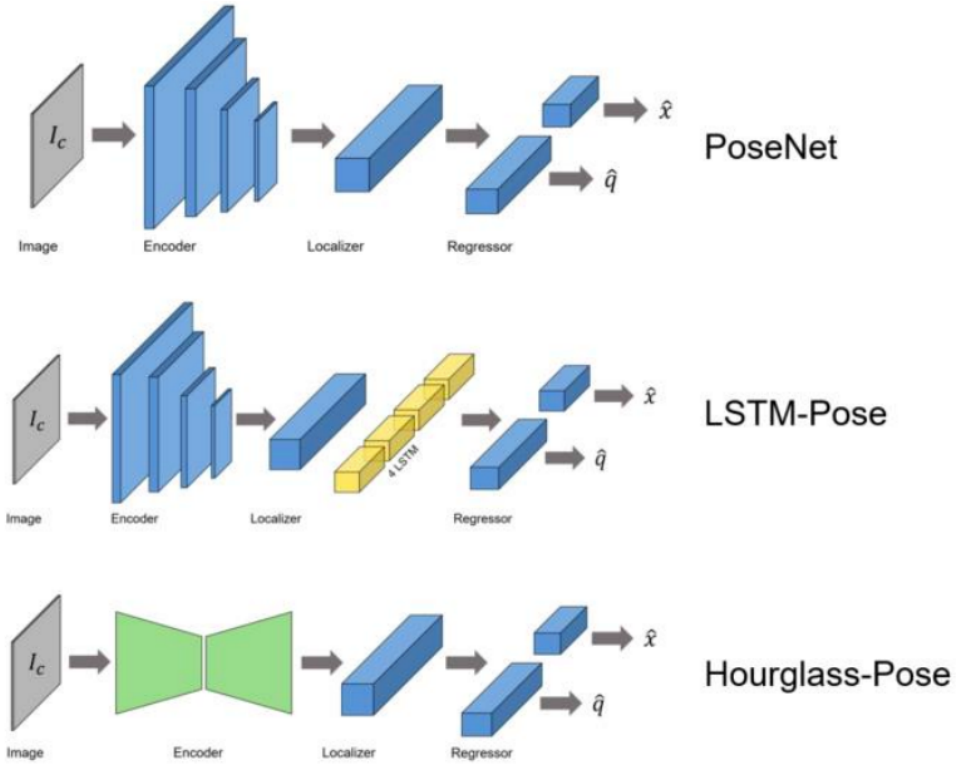


Figure 2.2: Example modifications to PoseNet architecture proposed in the paper. Source [SF19].

In this paper, they have proposed a variety of deep learning based approaches to estimate the camera pose. As part of this paper, they have proposed an extended neural network architecture to reduce the error of PoseNet [KGC15]. Some of these architectures are shown in figure 2.2. They have also presented the results of some of the networks in the paper which inspired us to use Hourglass architecture in this thesis.

In this thesis a stacked Hourgalss architecture has been used inspired by the paper 'Stacked Hourglass Networks for Human Pose Estimation' [NYD16] with multiple

different settings, details are in section 5.2.3

2. ***PoseFix: Model-agnostic General Human Pose Refinement Network [MCL18]***

   They have proposed a pose refinement network to estimate the skeleton pose of Humans. An approach of localizing semantic keypoints of the human body has been employed in their paper which is quite successful in estimating the pose of humans. In this thesis, the same architecture, Hourglass net, and same settings, Cross entropy with softmax + L1 loss, have been experimented, results are in section 5.2.3.

3 Technical approach

While the implementation of this thesis, plenty of tools and methods were experimented. Some proved to be useful and some could not be included because of the lack of their integration capabilities with existing tools. An overall pipeline was needed to be developed and to realize this a sequence of actions were to be performed: hand annotating the first image for ground truth, then generating further images based on the first frame and then training the CNN to detect keypoints later to perform PnP to estimate the pose.

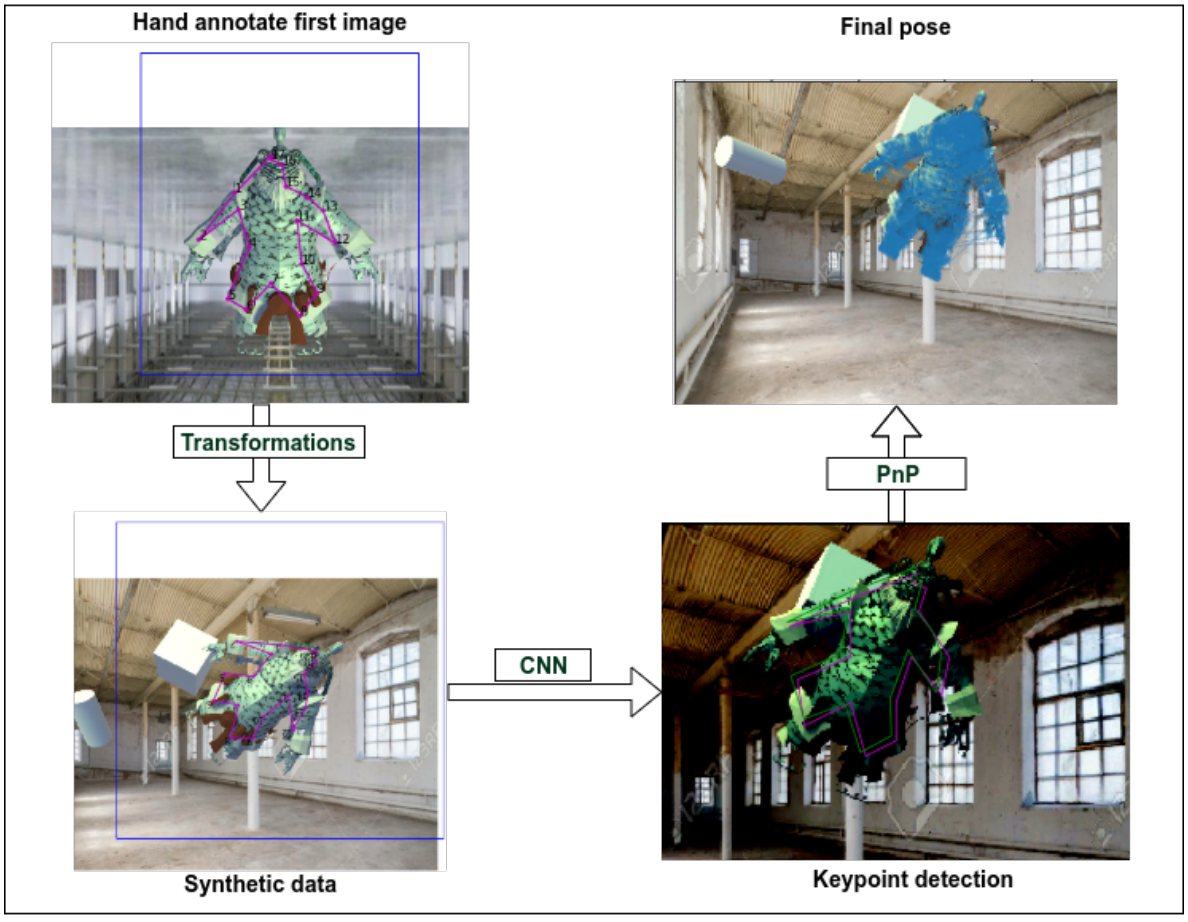


Figure 3.1: Overall pipeline.

Figure 3.1, shows the overall pipeline of the approach used in this thesis. The first picture shows a hand-annotated image, then by transforming the first image, synthetic data is generated along with transformed annotations. Which then passed to CNN for training and then using detected keypoints by the CNN and object's 3D coordinates,

PnP generates the final pose.

Below is a walkthrough of the technical modules and methods used to implement the application along with the technical flows to achieve end-to-end object localization.

3.1 Modules

3.1.1 Convolutional Neural Network

A convolutional neural network is Deep Neural Network based implementation most commonly used in the field of Computer Vision to process images. It generally consists of Convolutional (linear mathematical operation) layers, max pool layers as hidden layers and fully connected layer. These kinds of networks are specialized in extracting features information from the image and so very effective when performing tasks related to the feature of pixels.

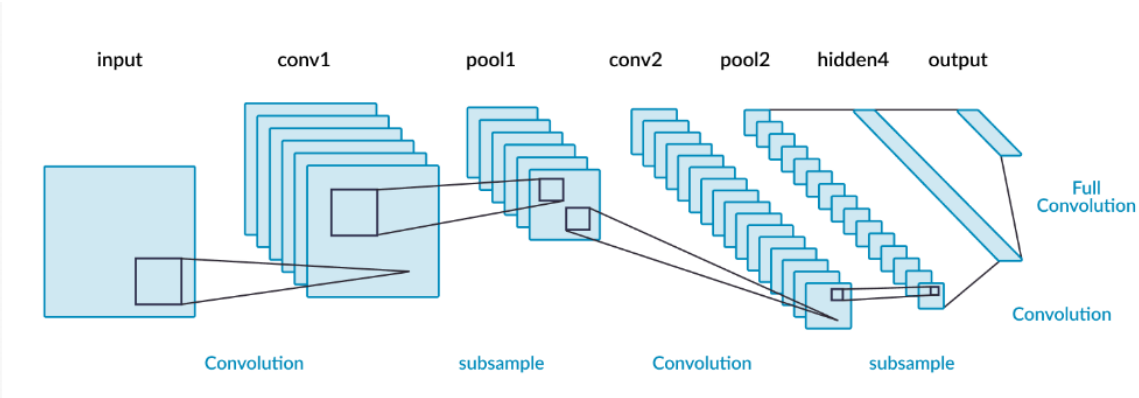


Figure 3.2: Architecture of CNN LeNet-5. Source [mis18]

In earlier days, the hand-designed filters were the tools to extract features from an image, Ie Sobel operator [Wik11b], etc but the advent of CNN made this process obsolete as CNN learns these filters dynamically to optimize the feature mapping process. Internally a CNN uses a convolution filter, which it optimizes using the input and output connected to neurons by performing forward and backward pass. The pooling layer is generally used to reduce the dimension of the feature map and so to reduce the complexity of the model. Although the basic layers of CNN remain the same, their architecture varies based on the task's requirement.

CNN is one of the main components of this implementation. It has been used to detect 2D interest points of a robot/object given an RGB image. in this application. Figure 3.3, shows the flow of, how Unity-3D scene transforms to introduce variation in captured images of the CAD-object to generate synthetic data and then pass it to CNN for training. Python programming language has been used to write neural networks and it interacts with Unity-3D over a network socket to enable end-to-end pose estimation pipeline. Various architectures of CNN have been experimented as part of this thesis to achieve improved precision and efficiency (evaluation results are published in section 3.1.2).

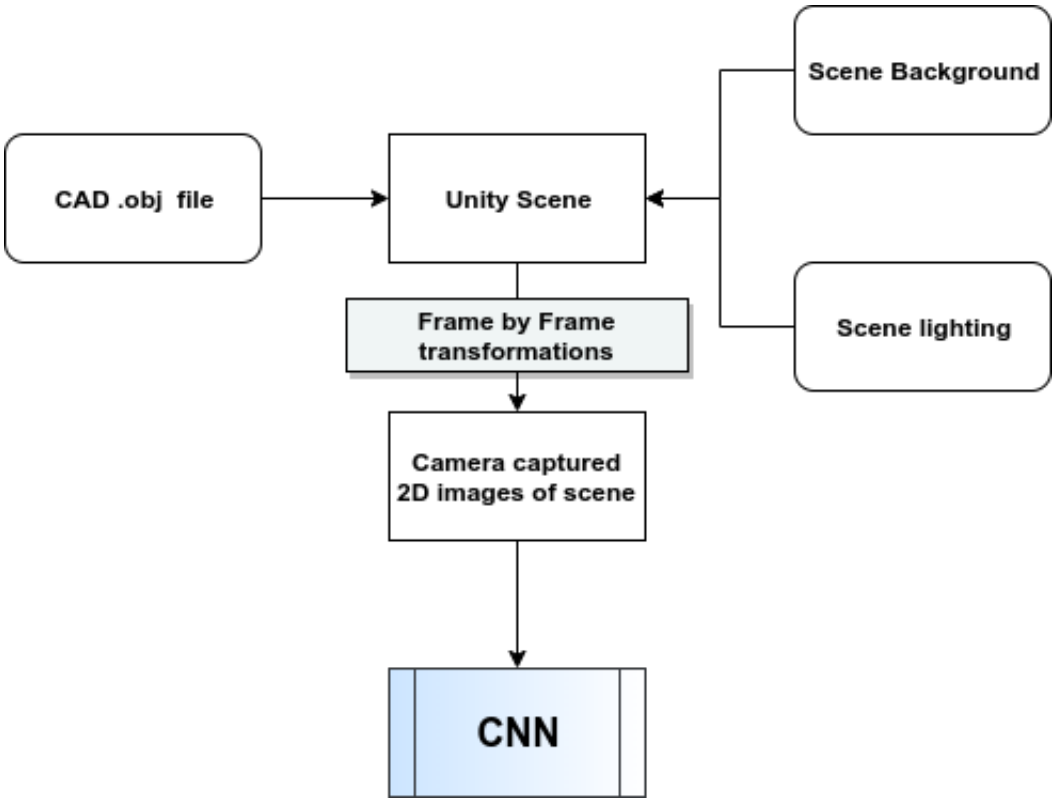


Figure 3.3: Flow to feeding synthetic training data to CNN

3.1.2 Perspective-n-point

Perspective-n-point is a method to estimate 6 degrees-of-freedom(DOF) pose of the calibrated camera given 3D coordinates of an object and their corresponding projected 2D points in an image.6-DOF of camera pose is defined by 3D rotation (roll, pitch, yaw) and 3D translation in correspondence to the world.

$$s\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & \gamma & u_0 \\ 0 & f_y & v_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}.$$

Figure 3.4: Formula of PnP. Source [Wik11a]

PnP provides multiple methods to solve the problem of pose estimation, for example, P3P, EPnP, etc. The most common solution is P3P for n = 3, where data is considered noise-free. Other methods assume some noise to be present in the data and so they provide solutions for cases for n $\geq$ 3.

PnP works on the assumption that the intrinsic parameters of the camera are already known: the focal lengths of the camera, aspect ratio, and the principle points. Figure 3.4 shows the formula of Pnp: the left-hand side is the scaling factor S and 2D points, the right-hand side is the camera matrix (intrinsic parameters), rotation-translation matrix (extrinsic parameters), and 3D points. This formulation is written in C# programming language within Unity-3D using the OpenCVSharp as it uses implementations from both OpenCV and Unity-3D. PnP's goal in this application is to localize the object by calculating the Rotation and translation by using the detected 2D keypoints with 3D coordinates of the object. Figure 3.5 shows the process of PnP using 3D points and detected corresponding 2D keypoints.

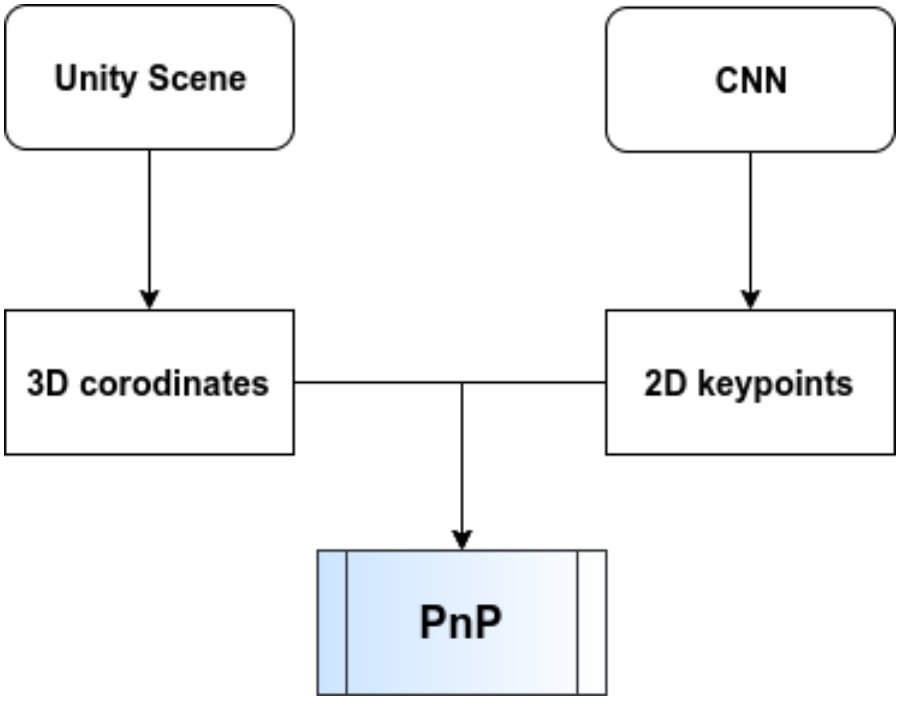


Figure 3.5: PnP process

3.2 Tasks

This overall application has been broken down into three technical flows to achieve the end to end automation. These flows have been separated on the basis of their actions performed, marking first annotated data, generating synthetic data and training CNN and, then performing PnP.

Generate

This flow is used to mark the initial hand-annotated data points, 2D data points corresponding to 3D coordinates of objects, to generate further training image data with ground truth 2D keypoint annotations.

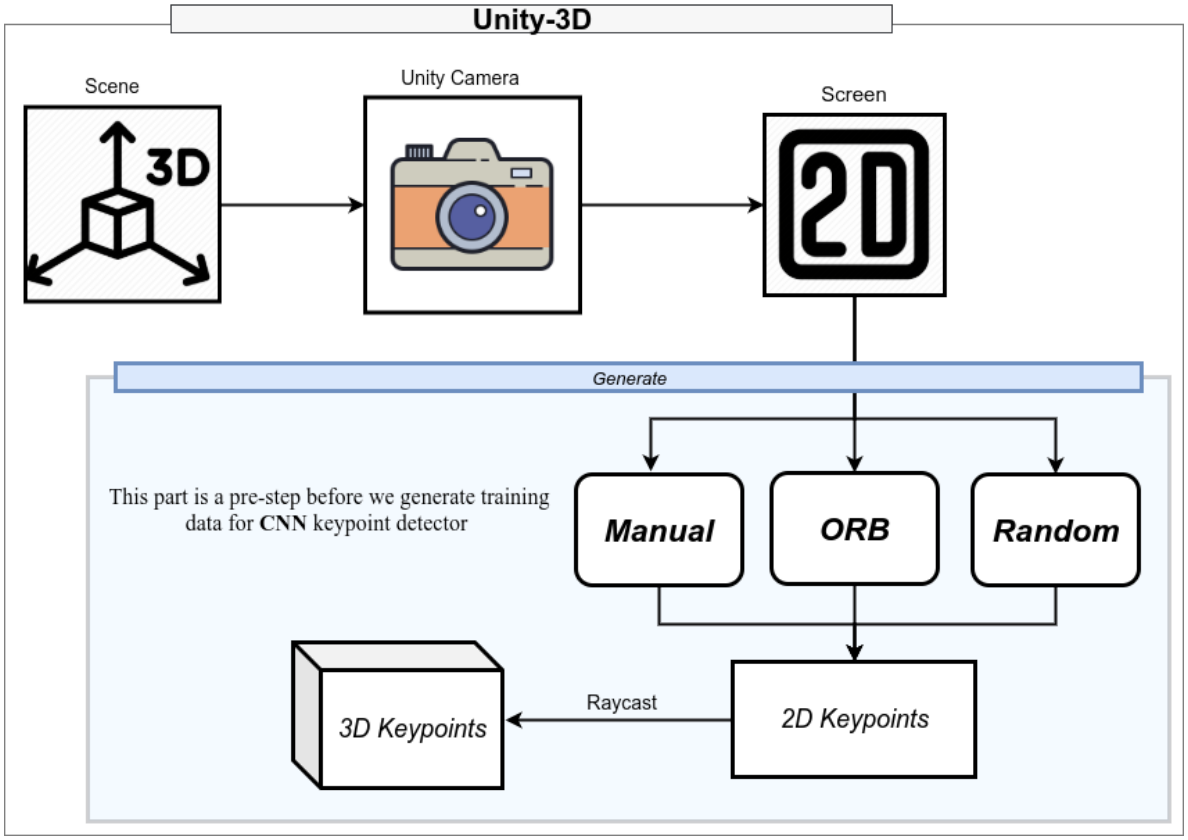


Figure 3.6: Generate task flow

This is the first and mandatory step if CNN training is pending as to train the network, images with different visual settings are needed with respective 2D ground-truth keypoints. To generate these training images and ground-truth points, a set of starting 2D points is needed. Once the starting set of 2D points are marked, either by using ORB keypoint detector, [RRKB11], or by manually marking the keypoints using mouse interface, then Unity's *Raycast* [Uni19e] method is used to find the corresponding initial 3D coordinates in the scene. Finding these 3D points helps generate further training data as these points then can be rotated, translated and scaled in conformity with the 3D scene transformations to generate more images with ground truth keypoints. This overall flow is shown in 3.6.

There are multiple ways provided to generate the initial 2D data points.

- **Manual:** We have given mouse click interaction where one can click on the image and select initial keypoints manually as per one's convenience.
- **ORB/SURF:** In the initial step one can opt for ORB/SURF keypoint generator, implemented by OpenCV, to generate initial keypoints.
- **Random:** In this option, some random keypoints are thrown on the object.

Control

This flow is to generate the training images for CNN based on the 2D keypoints and corresponding 3D world points generated in Generate flow, section 3.2.

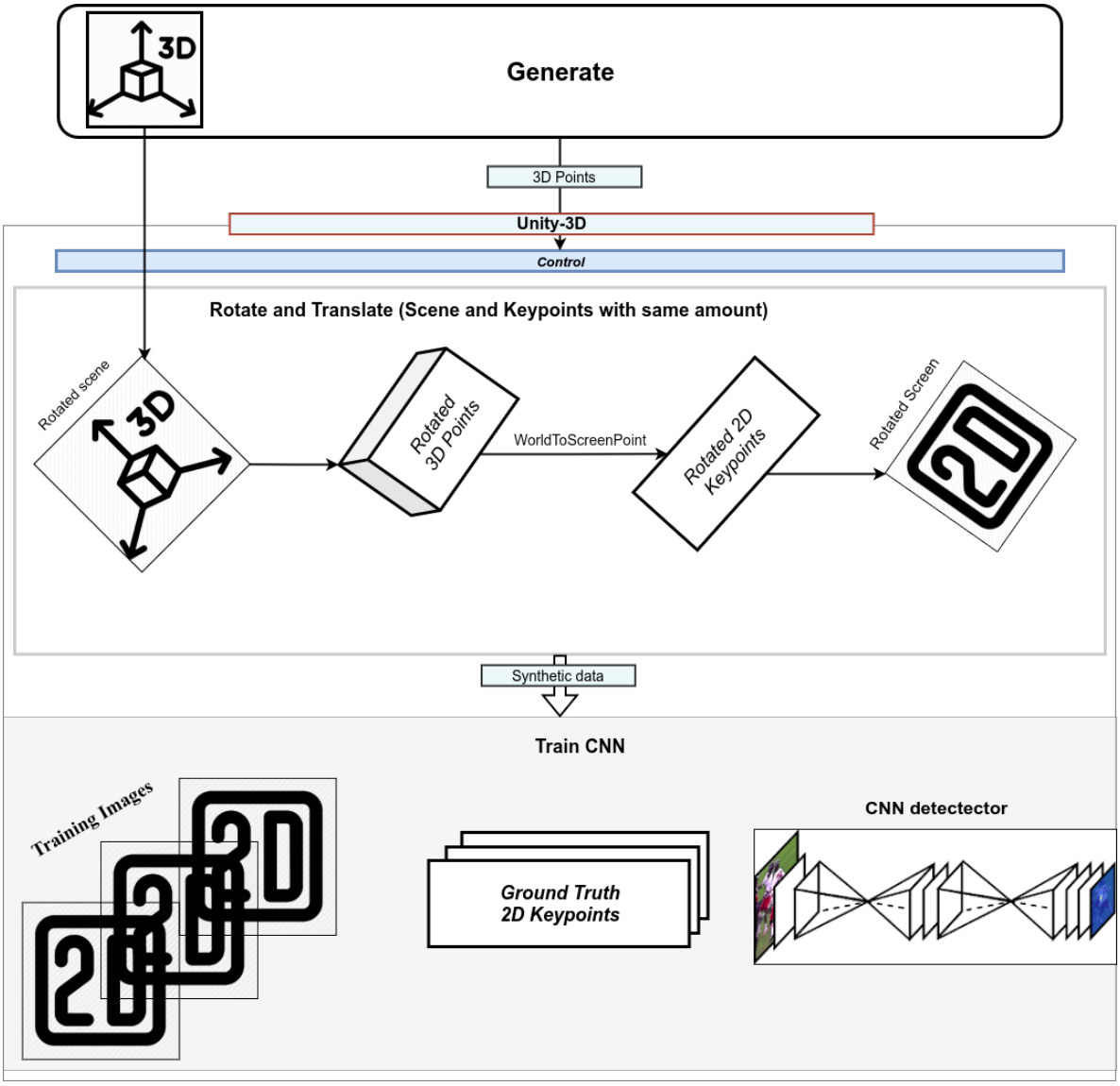


Figure 3.7: Control task flow.

In this flow, already generated initial 3D points are picked and then rotated, translated and scaled along with the scene with the same factor. These transformed scenes generate images with variations and then the transformed 3D points help find the ground truth keypoints in these generated images by using Unity's *WorldToScreenPoint* [Uni19d] method, by mapping the world point to image point. Once transformed images and transformed ground truth 2D points are generated, they are then used to train CNN. Figure 3.7 shows the flow to Rotate, translate and change the background of images to generate synthetic image data and then train the CNN with generated data.

There are two ways to train the CNN with generated synthetic image data.

- **Synchronously:** Python is integrated with Unity-3D over the network socket which allows training the CNN per frame as scene changes in Unity. This action is computationally heavy.
- **Asynchronously:** One can generate as many numbers of images for training offline and save them to a location which can later be used to train the CNN separately.

Convert

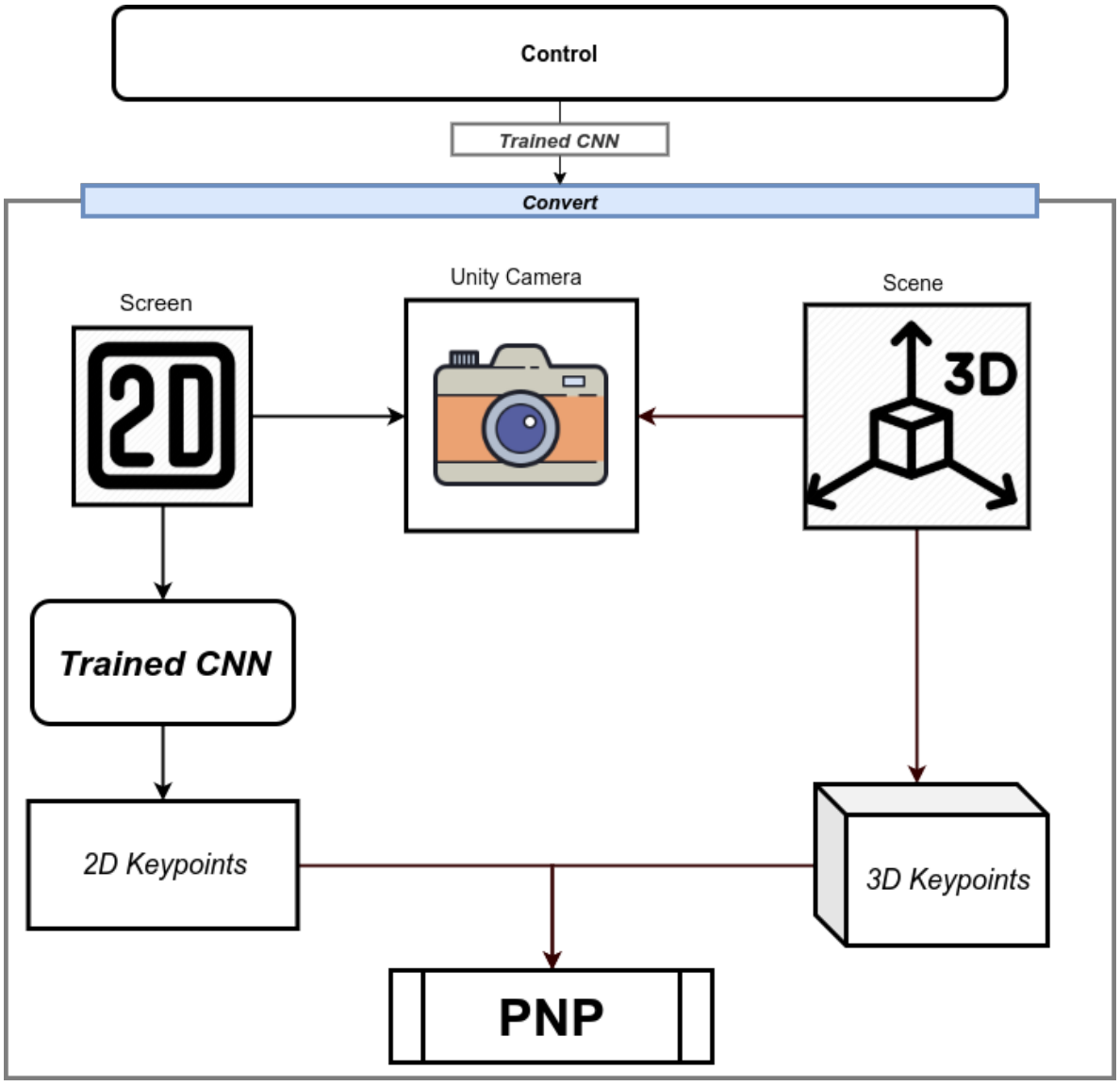


Figure 3.8: Convert task flow.

This flow is to detect keypoints using the already trained CNN in Control, section 3.2. This flow represents the end result of the whole implementation as in this flow, trained CNNs and PnP are used to estimate the pose of Camera in real-time.

While training the network trained model is being saved at a physical location and then by using the interface, written to communicate with the trained model through a network socket, the real-time images are passed to a trained model to predict the keypoints. Once detected keypoints are sent back to Unity, perspective-n-point is performed to calculate the desired 3D rotation and 3D translation of the camera (extrinsic parameters) to localize the object. 3.8 shows how a trained CNN is used to detect keypoints from the image and use PNP to visualize object localization using detected keypoints and 3D object coordinates.

4 Implementation

4.1 Environment

The following platform and software have been used.

Platform:

- Linux based operating system UBUNTU 19.0

Tools and software::

- Unity Hub 2.1.2
- Unity Editor 2019.3.0a6
- OpenCVSharp 4
- Python 3.7

Within Python, try to use the latest version of APIs like Numpy, Matplotlib, Pandas, PIL, Pytorch.

4.2 Tools and Technologies

This thesis has been developed in three folds technical steps.

- ***Synthetic data and PnP:*** This was the first implemented part in which Unity-3D used to generate the synthetic data. To perform this, Unity-3D needed to be installed, which was done using Unity-hub and then Unity-3D was tested for integration with CAD(.obj file) objects, [uni19g]. Once Unity was setup, OpenCV needed to be integrated to perform PnP and additionally to detect initial key-points. This was done using OpenCVSharp [shi19a], unity's extension of OpenCV.
- ***CNN:*** Developing, training, and testing CNN was done separately with the help of generated synthetic data. All these implementations were written using Python 3.7 and then trained and evaluated on GPU provided by Google-Colab [Goo19].
- ***Connecting CNN and PnP:*** As, CNN and PnP modules were developed separately, a framework to connect these both components was needed. To achieve this connectivity NetMQ [Net19] was used, which gives socket-based connectivity to exchange messages and files over TCP protocol.

Below are the details of all the tools used to develop this pipeline.

Unity-3D

As the scope of this thesis is to generate synthetic data to train the convolutional neural network for visual object localization, a CAD-object based approach seemed to suit the requirement and so to develop such an application where interaction with such CAD-objects in real-time, manually or via code, was possible. Unity-3D fits this profile quite well as it gives a very easy way to map 2D images with 3D objects through screen and scene interfaces in tandem. To provide the variation in generated images, objects needed to be transformed and to do so the depth of the object from the camera was needed to be calculated, which Unity could achieve by using its inbuilt function *rayCast* (Code snippet 1) (this depth calculation was only needed to calculate the synthetic data and not for final pose estimate). Also, Unity has method *WorldToScreenPoint* which calculates the image points corresponding to 3D world point, which was used to generate ground truth points corresponding to transformed 3D points (code snippet 3). These qualities of Unity-3D allowed the generation of synthetic images with dynamic transformations, rotation, translation, and background variations. A simple C# script is all that was needed to manipulate the scene to generate 2D images from 3D objects with such transformations.

Unity-3D also proved to be very efficient when needed to perform PnP and other OpenCV-based tasks as it gives a very convenient way to integrate outside tools to be used within Unity. In this application, Unity-3D is being used to perform the below tasks.

- It provides synthetic image data to train CNNs based on CAD objects manipulation, figure 3.3.
- Perform PnP to localize objects using detected 2D keypoints from CNN and 3D coordinates of Object, 3.5 (Code snippet 5, 6, 7, 8).

There are some other tools also available in the market for generating synthetic data, like Blender. Unity-3D and Blender, both tools enables the interaction with the 3D object using a graphical interface. However, Unity-3D is generally preferred over blender as the former has the capability to integrate with Hololens and other Machine Learning based plugins.

OpenCV

OpenCV is a library of programming functions that have been proven to be a very efficient tool for Image processing and computer vision. It has many inbuilt functions that make image manipulation easy and efficient. In this application, OpenCV is being used as a plugin, OpenCVSharp using *.so* file, 4.2, within unity and its inbuilt functions are used to perform below actions.

- To generate initial keypoints using ORB Keypoint detector (Code snippet 4).

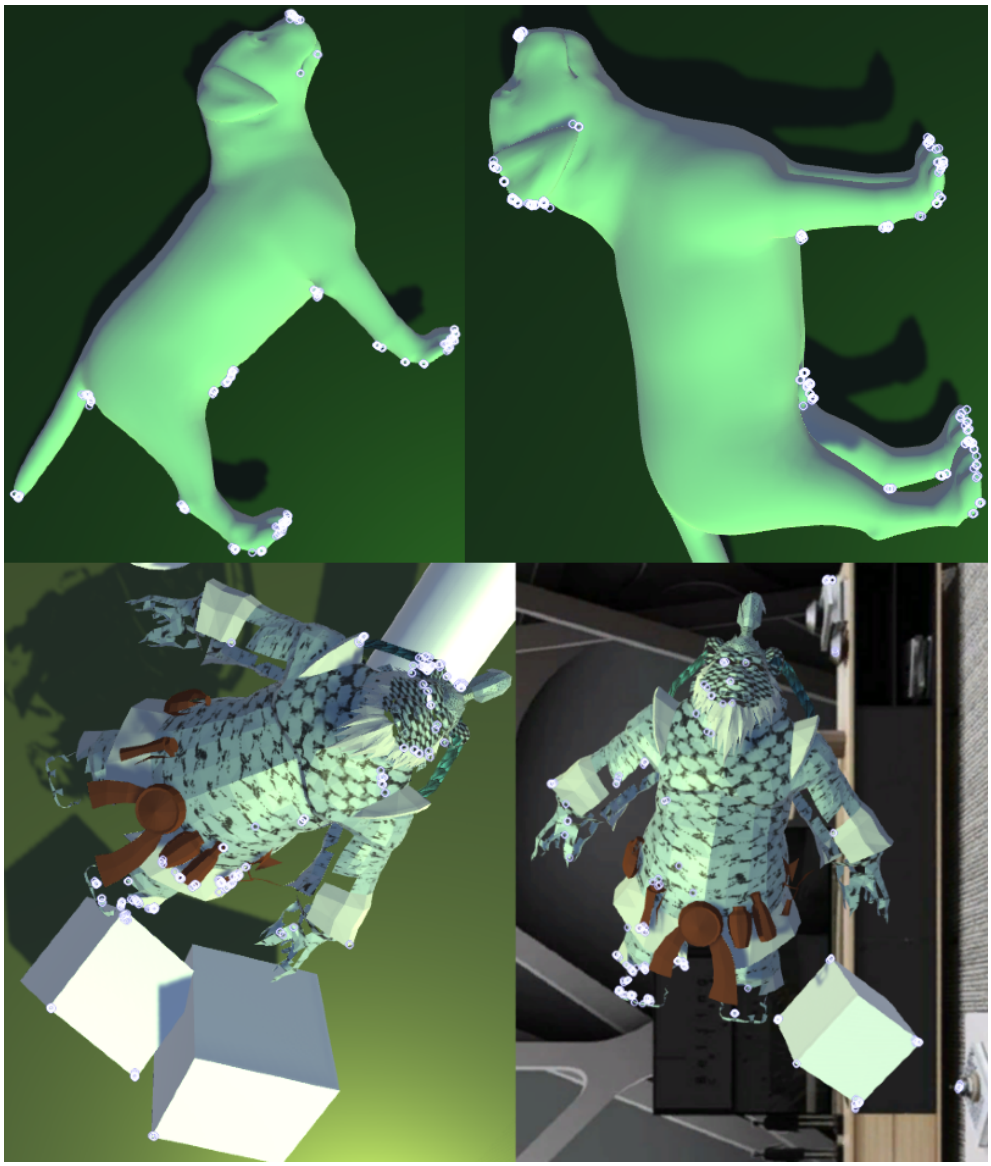


Figure 4.1: Keypoints detected by ORB in a cluttered scene.

- To perform PnP mapping to localize the object using 3D and 2D points 3.5 Code snippet 6).

As mentioned above, OpenCV has many inbuilt functions for image manipulation and processing like SIFT, SURF, ORB, FAST, etc. for keypoint detection but they perform poorly with cluttered and occluded scenes, as it can be seen in 4.1 that the keypoints are also present on the edges on cubes. On the contrary, CNN provides this kind of robustness and symmetry and can be proved more trustworthy than OpenCV's functions when PnP needed to be performed by mapping 2D-3D. The progress in the field of CNN and computational resources has helped CNN evolve to be more precise and hence more popular. However, these OpenCV based methods can indeed be used as pre-steps for neural networks.

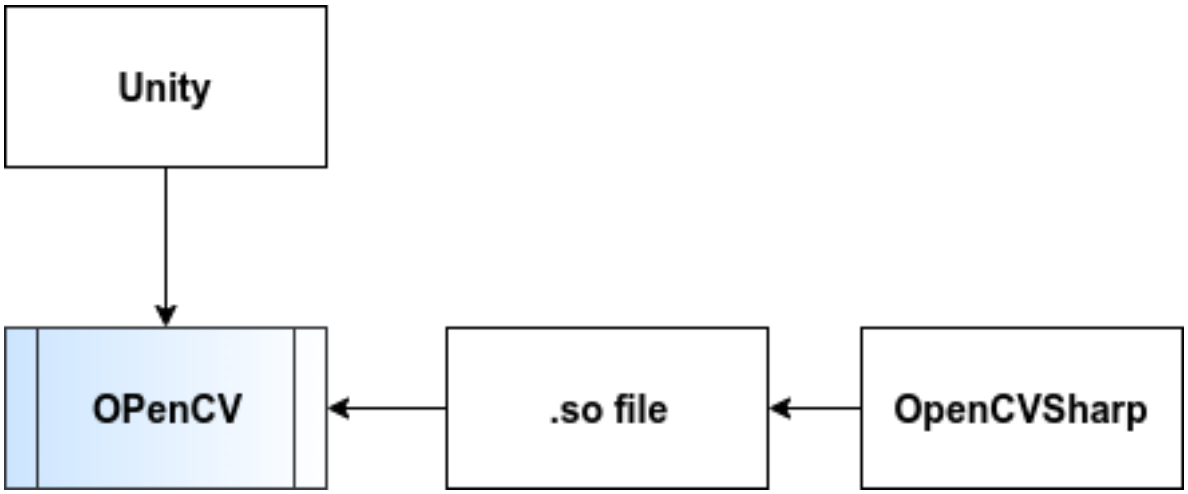


Figure 4.2: OpenCV inside Unity-3D

Python

CNN is the main component of this application and python is a language that provides plenty of ways to implement that. Python is the best-suited language out there for developing neural networks as first, it is an interpreter based language and so it is fast and second, it supports plenty of APIs that make the implementation easy and efficient. The main API which is used to write CNN in this application is Pytorch it has many benefits over its rivals like, Tensorflow. Pytorch has been used as this is a bit easier to implement than Tensorflow and also it makes sure to utilize the computation resources efficiently to speed up the training and predictions. However, there might be one benefit of using TensorFlow over PyTorch is that now TensorflowSharp gives an integrated API to use TensorFlow directly inside Unity-3D which could be proven more efficient.

As Unity-3D and Python are two very different entities and for the purpose of creating the pipeline to estimate pose, they needed to work in tandem. To achieve so, a messaging library was needed to make Unity-3D communicate with python code over the network and for this purpose, NetMQ has been used. Unity-3D and python are exchanging data in below scenarios. (code snippet 9, 10)

- Once when training of CNN is done synchronously with Unity-3D scene change. This happens when *realTime* option is opted in Unity. In this case Unity passes images and ground-truth points to python at every frame to train the CNN. However, using this method restricts CNN to train for only one epoch, which can hamper it's accuracy.

- Second when images are passed to trained CNN for prediction. In this case images are transferred over the socket to trained CNN to get the predicted 2D keypoints.

To make these settings work one has to be careful to start the Python side server before using Unity-3D or else it will not produce desired results.

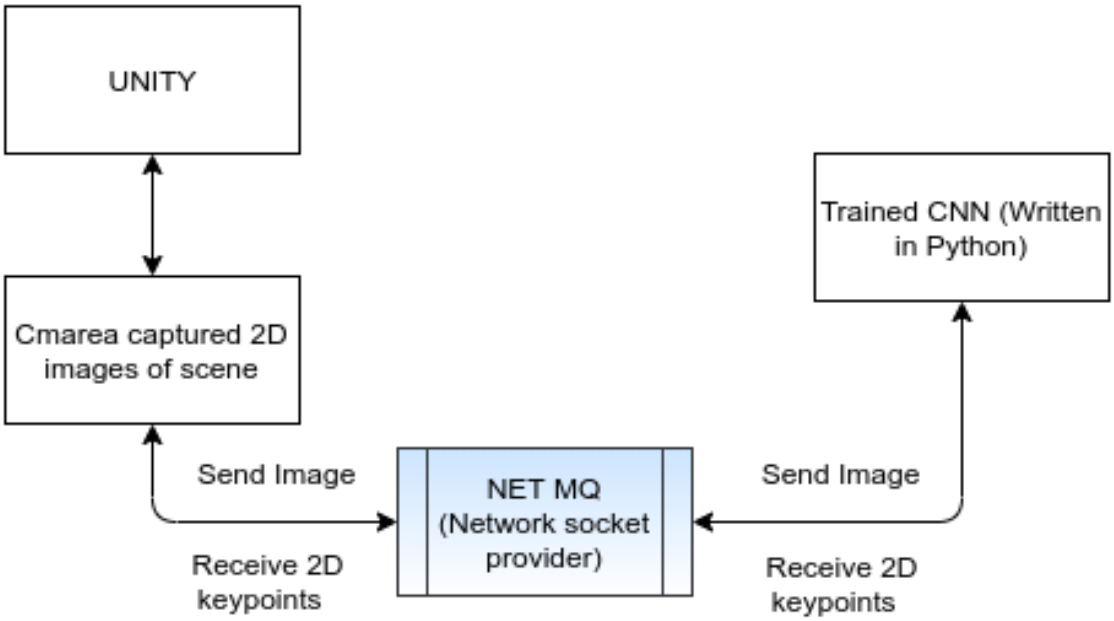


Figure 4.3: Exchange of data between Unity and Python over network socket.

4.3 Project and Code Setup

To set up this project on the local machine couple of tools need to be installed first. These installations vastly depend on the platform that is being used, in this case, Ubuntu 19.04.

All the installations steps below are based on the assumption that the operating system is Linux based and a graphics card is available on the system.

- ***Unity-3D:*** Unity-3D application is not available for Linux based platforms it's editor version can be used using Unity-hub.
  - Install Nvidea graphics card driver. [nix18]
  - Download and install Unity Hub [Uni19a].
  - Open UnityHub and create a new project in Unity Hub and then import the asset downloaded from Github.
- ***OpenCvSharp:*** To use OpenCVSharp in Unity-3D, a git hub project [shi19b] need to be downloaded and imported in Unity. Follow the document attached with this installation [shi19a].
  - Move 'rsp' and '.so' files to Asset folder. [Uni19c].
  - The files of OpenCV now can be used by importing the OpenCV namespace in Unity files. [Uni19c].
- ***Python:***: Python can be used independently and so one can use any online available tutorial to install in on the local machine. One way to install it is through

Conda [Dig18]. Once Conda is installed, related APIs need to be installed.

- Numpy,
- Matplotlib,
- Pandas,
- PIL,
- Cuda,
- Pytorch

Jupyter notebook is by default installed in Conda but if it is not there install it separately in Conda.

- ***Code:*** The code is present on GitHub.
  - Unity-3D/C# code can be downloaded from *Unity-GitHub*. Using Asset folder inside Unity is enough to import all the related scenes and settings.
  - Python Code is available at location, *Python-GitHUb* along with saved trained model checkpoints and output images. A separate Ipynb file is written to be used directly on Google-Colab along with data in zip format.

Below are the project structures of Unity and Python used while developing this application.

Unity-3D:

Figures 4.4 shows the project scene structure used in Unity-3D and 4.5 shows the asset folder structure used while development. Some files in the asset folder should be used as it like, 'rsp' and 'so' files however other folders can be changed as per the convenience. Please select *Physical camera* in Camera.

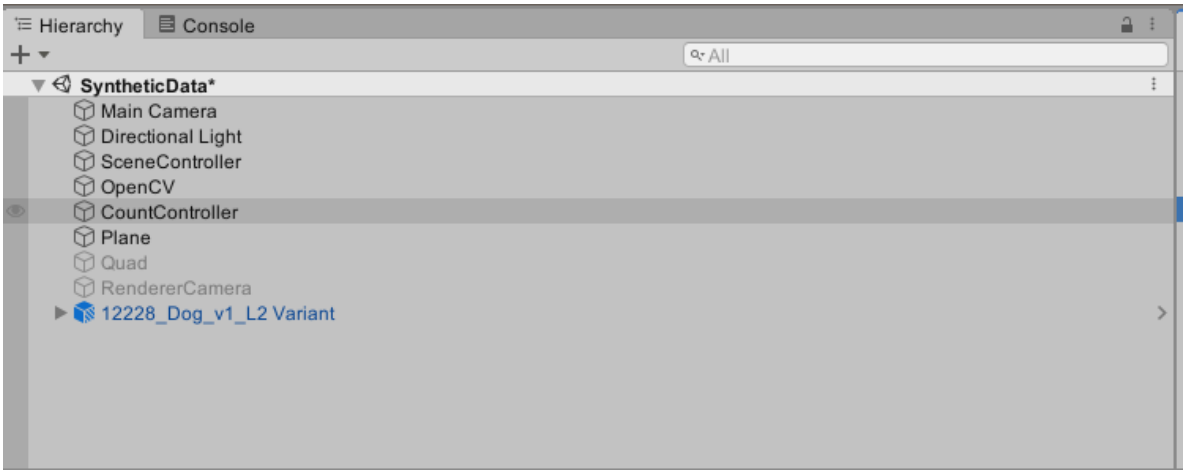


Figure 4.4: Project structure in Unity-3D.

Python:

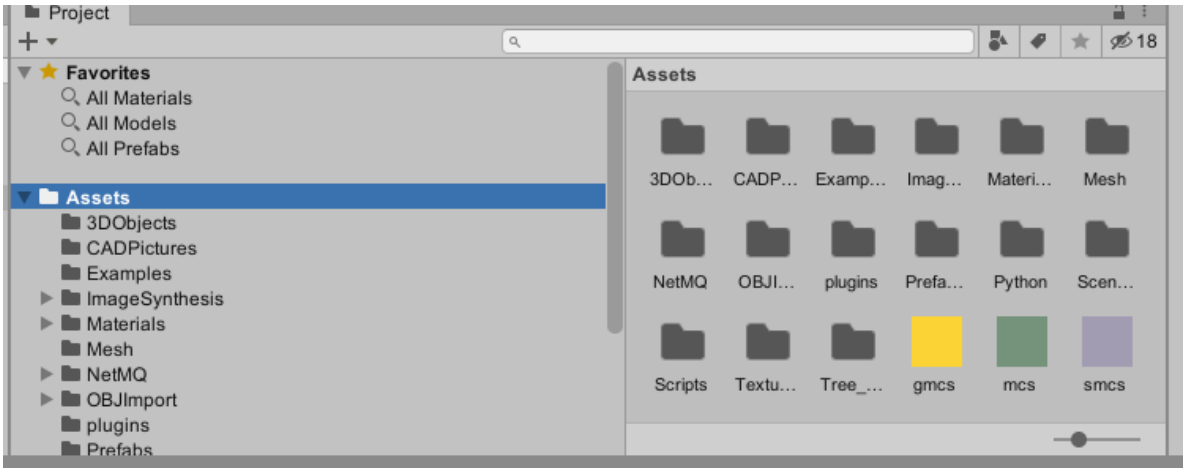


Figure 4.5: Asset Structure in Unity-3D.

Figure 4.6 shows the project structure used in python.

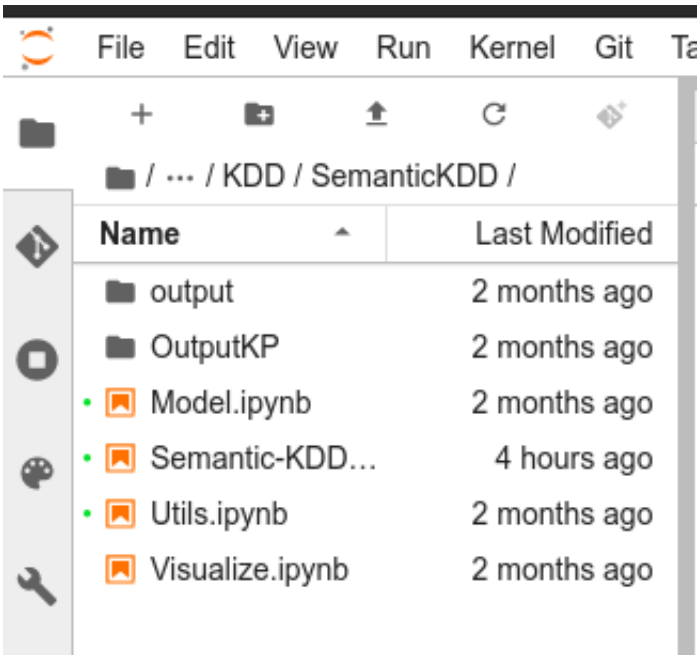


Figure 4.6: Python project structure.

5 Performance evaluation

5.1 Datasets

The synthetic data, for training and evaluation purpose has been generated using Unity-3D, then further real data has also been used for evaluation of final pose estimation. In this thesis, *domain randomization* (placing the foreground objects within virtual environments consisting of various distractor objects in front of a random background) has been used to generate synthetic data for training purpose.

The results in this thesis have been benchmarked against multiple datasets. The Training has been done on synthetic data and then the estimated pose has been compared on synthetic as well as real data [HMB+18]. The CAD models have been collected from different sources, keeping in mind below points.

- Models should be textured and resemble the real data set closely.
- Images should have enough variations.
  - Scene's background, color and light, occlusion and incompleteness
  - Object's rotation and translation

Data from Table 5.1 has been used to measure the performance of CNNs and dataset from table 5.2 has been used for final pose evaluation on synthetic and real data.

Dataset	**#Train**	**#Test**	**#KPs**	**Source**
Dog	*100*	*20*	18	Synthetic dataset
Panda	*100*	*20*	18	Synthetic dataset
Canister	*100*	*20*	18	Real dataset [PZC+17]

Table 5.1: Datasets for CNN evaluation

Dataset	**BOP Object**	**Epochs**	**#Train**	**#Test**	**#KPs**	**size**
Homebrew	4 (Buffalo)	*300*	*1000*	*100*	51	640 × 480
	32 (Car)	*300*	*500*	*50*	45	640 × 480
TYOL	9 (Floater)	*300*	*500*	*50*	39	640 × 480
	3 (Cup)	*300*	*100*	*20*	50	640 × 480
Linmod	9 (Duck)	*300*	*100*	*20*	48	640 × 480

Table 5.2: Datasets for final pose estimation [HMB+18]

5.2 Convolutional Neural Networks

As part of CNN implementation, multiple architectures with different settings have been experimented seeking more precision and efficiency. These architectures differ in their layer count and the usage of Loss functions and Optimizers. We first measured the performance of different architectures on synthetic data and then evaluated the results on real data using the best performing CNN net.

Below are some pre-processing steps performed before feeding the data to CNN:

- Images has been segmented (bounding-box detection) before passing them to keypoint detector for which Unity's mesh method has been used in this thesis, but one can also opt for available segmentation-based algorithms(YOLO etc.) to do so.
- Images haves been cropped and padded before they are fed to CNN.
- Images have *not* been Normalized before training/testing, as the pixel distribution of synthetic and real data are different and so network trained on raw images performed better.

5.2.1 Resnet 52/101/152

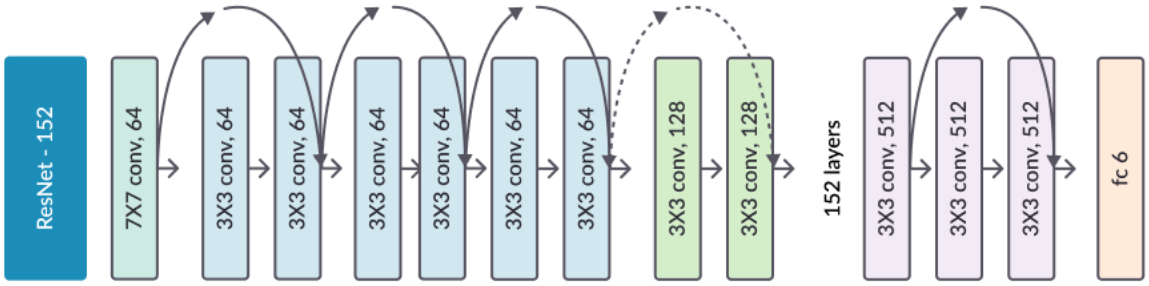


Figure 5.1: Architecture of Resnet-152. Source [HPL+18]

Resnet is a convolutional neural network 52/101/152 layers deep and is very efficient when used to classify images. A CNN using Resnet-52 as the backbone network has been evaluated to perform keypoint detection with input settings listed in 5.3.

Loss function	Optimizer	Input
MSE loss	Adam Optimizer	Greyscale images

Table 5.3: Input setting of Resnet-52 based CNN

This network is using MSE loss function with Adam optimizer.

$$\textbf{MSE}(\textit{Mean Squared Error}) : \frac{1}{n}\Sigma_{i=1}^{n}\left(y_i - \tilde{y_i}\right)^2 \tag{5.1}$$

5.2.2 Hourglass

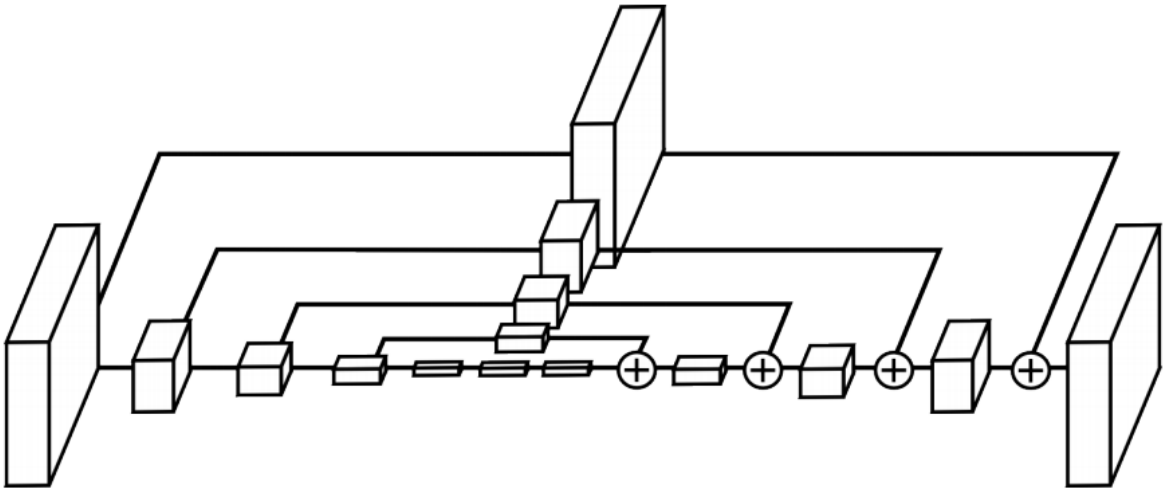


Figure 5.2: Single hourglass module. Source [NYD16]

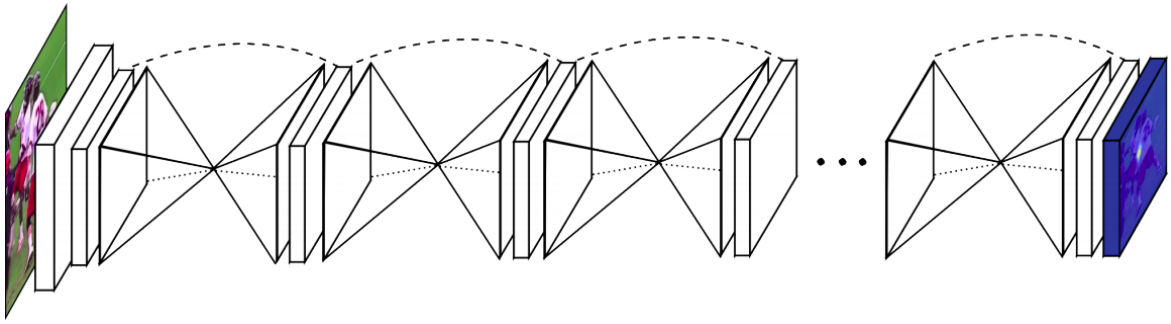


Figure 5.3: Stacked hourglass modules. Source [NYD16]

This idea of using stacked hourglass architecture has been employed from the paper 'Stacked Hourglass Networks for Human Pose Estimation' [NYD16]. These kinds of architectures are the most successful when the input and output image relates to each other. This network for pose estimation consists of multiple stacked hourglass modules that allow for repeated bottom-up, top-down inference, figure 5.3. An illustration of a single hourglass module shown in 5.2. Each box in the figure corresponds to a residual module as seen in figure 5.2. The number of features is consistent across the whole hourglass. These pictures are taken from the paper stacked hourglass for pose estimation [NYD16].

This architecture have been evaluated with two different settings, listed in table 5.4.

1. *Keypoint detection*: In this method an RGB image is fed to the Stacked hourglass network network and it learns to detect hetmaps of keypoints by optimizing MSE loss on predicted heatmap and ground truth heatmap. This setting inspired from Semantic KDD [PZC+17] [NYD16].

No.	Loss function	Optimizer	Input
#1	MSE loss	Adam	Raw/Normalized images
#2	CE with Softmax + L1 loss	Adam	Raw images

Table 5.4: Input setting

2. *Keypoint filtering*: In this setting an image stacked with heatmap of initial keypoints is fed to the Hourglass network and then it learns to predict output heatmap by optimizing the Cross entropy with Softmax + L1 loss function. This setting is inspired from PoseFix implementation [MCL18].

$$\mathbf{CE}(CrossEntropy): \ -\sum_{n}^{N}\sum_{k}^{c} t_k^n \ln y_k^n \tag{5.2}$$

$$\mathbf{Softmax}: \ \frac{\exp(\phi_k)}{\sum_j^c \exp(\phi_j)}, \tag{5.3}$$

$$\mathbf{L1}: \ \left\| \bar{P}_x - \tilde{P}_x \right\|_1 \tag{5.4}$$

After training the network for #300 epochs with both the settings, keypoint filtering did not seem to work well enough as keypoint detection. For detecting keypoints, the Hourglass network is giving more accurate results than simple Resent-52 based architecture.

5.2.3 Results

Networks have been evaluated over the datasets mentioned in table 5.1 and then the best network has been used for evaluating the pose on dataset from table 5.2.

Model	Training error			Generalization error		
	Dog	**Panda**	**Canister**	**Dog**	**Panda**	**Canister**
Resnet-52	0.38	1.29	NA	0.31	1.26	NA
Hourglass$_{\#1}$	3.5×10^{-5}	9.1×10^{-5}	7.8×10^{-5}	38.3×10^{-5}	75.1×10^{-5}	72.5×10^{-5}
Hourglass$_{\#2}$	2.9	3.7	NA	1.73	1.76	NA

Table 5.5: Training and Test error of different networks on synthetic data.

Table 5.5 lists the results on #20 images of networks after training them on #100 synthetics and real images for #300 epochs on Google-Colab. Based on these results, it can be seen that the best results have been generated by the Hourglass network with MSE-loss, a Keypoint-detection technique. A Keypoint-filtering technique inspired from [MCL18], Hourglass network with Cross-Entropy with softmax+L1 loss function, has also been evaluated, but it did not seem to work well. "Hourglass network with MSE loss ($Hourgalss_{\#1}$)" function has produced best results on both type of datasets. All

these networks have been able to predict the result given an image within seconds (0.3 to 0.8 seconds).

OKS	$\exp\left(-\frac{d_i^2}{2s^2k_i^2}\right)$	d_i : L2 norm between GT and Prediction s : sqrt(boundig box area/image area) $k_i : \mathbb{E}\left[d_i^2/s^2\right]$, per-keypoint standard deviation with respect to object scale s
PCK	$bool\left(\frac{d_i \leq 0.05 * diagonal}{n}\right)$	d_i : Euclidean distance between GT and Prediction diagonal : Diagonal of the bounding box n : Number of keypoints
IoU	$area\left(\bar{B} \cap \tilde{B}\right) / area\left(\bar{B} \cup \tilde{B}\right)$	IOU of mask area between GT and Prediction
$\mathbf{AP}_{th}$	$\text{auc}(R, P)$	auc : area under curve th : threshold *0.5* R : Recall $\left(\frac{TP}{TP+FN}\right)$ P : Precision $\left(\frac{TP}{TP+FP}\right)$

Table 5.6: Functions used to evaluate detected keypoint accuracy. [LMB+14] [HMO16]

Based on the results from tables 5.5, the best network was chosen to evaluate the final pose on datasets from table 5.2. The $Hourglass_{\#1}$ network is trained on Nvidia GPU to estimate the final pose, and the results are listed in table 5.7 on the basis of evaluation functions mentioned in table 5.6.

In order to get better results on Real dataset following steps have been performed.

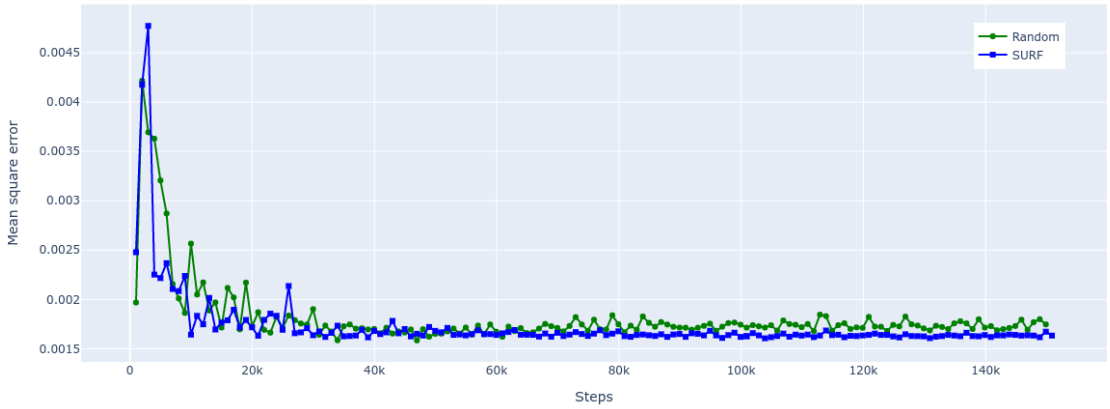
- Avoid normalizing the images before training/testing. Normalizing images were making 75% of pixels of real data shift to value 0 and so the results were better on un-normalized images.
- Based on cross-validation on the real dataset, below setting gives the best results.
  - #Stacks = 1, more complicated stack layers tend to overfit on real dataset.
  - #Residual-blocks = 1 & Learning rate = 2.5e-4

Only the best and relevant results have been published in the table 5.7. Figure 5.4(b) shows the Mean average accuracy (MaP) of PCK, table 5.6, dependent upon the Standard deviation of d_i of GT and Prediction. As you can see in figure 5.4(b), the pixel deviation for Synthetic data is almost *0* which signifies that the network is able to perform better on synthetic dataset however, real dataset has high pixel deviations , although both the graphs from 5.4 suggests that *SURF* keypoint distributor works better then the *Random* distribution of keypoints.

Data	**Method**	**Type**	**OKS**	**PCK**	**IOU**
			$AP_{0.5}$	$AP_{0.5}$	$AP_{0.5}$
Buffalo$^{(1000)}$	SURF	Real	**0.062**	**0.39**	**0.00**
		Synth	0.75	**0.99**	**0.92**
	Random	Real	0.025	0.18	0.00
		Synth	**0.81**	0.98	0.87

Table 5.7: keypoint accuracy score of $Hourglass_{\#1}$ on mixed dataset.

[a]



[b]0.1

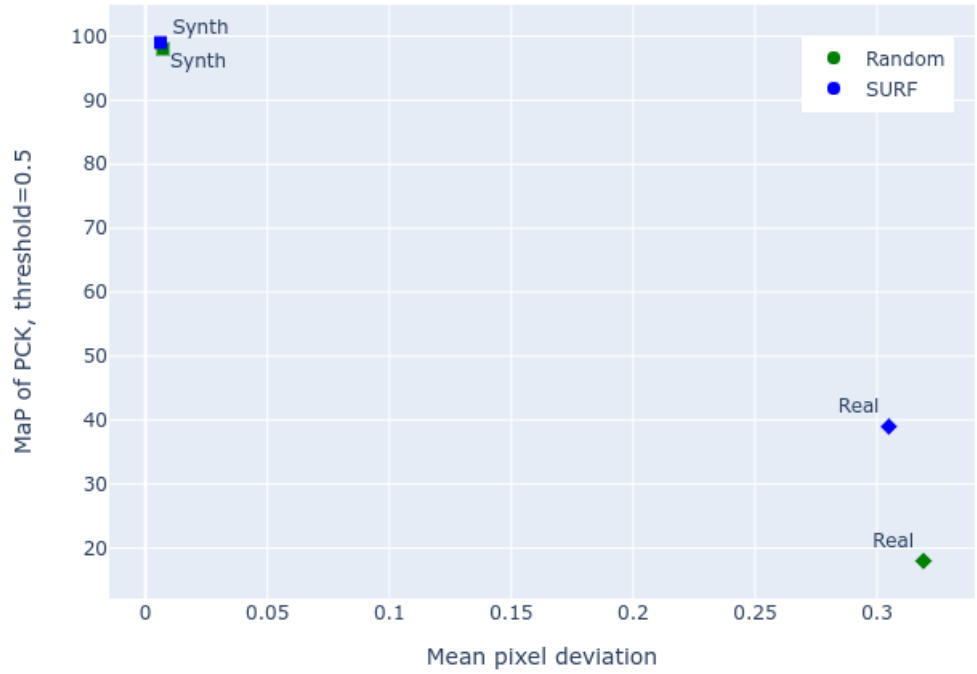


Figure 5.4: (a): Validation (MSE) error on real dataset, (b): Predicted keypoint precision dependent on keypoint related pixel deviation.)

5.3 Perspective-n-Point

Once CNN returns detected 2D keypoints PNP mapping is performed using detected 2D keypoints and the 3D coordinates to estimate the pose of a monocular camera. The intrinsic parameters of the camera are already known in this case. PnP has multiple implementations based on the value of *n*.

- ***E-PNP*:** This is a non-iterative method that works on the assumption that all 3D points can be expressed as the weighted sum of four virtual points. These control points are further solved for the solution. This method solves the general problem of PnP for n $\geq$ 4.[LMNF08].
- ***U-PNP*:** This approach provides the capability to handle multiple properties simultaneously, which makes it more accurate and faster than existing closed-form solutions [PAM13].
- ***DLS*:** This method is for computing the solution for PnP in general case (n $\geq$ 3). They minimize a nonlinear-least-squares cost function based on camera measurement equation. The key advantage of this method is scalability since the order of the polynomial system that they solve is independent of the number of points [HR11].
- ***Iterative*:** This function minimizes the pose by iteratively minimizing the reprojection error, the sum of squared distances, between the 3d coordinates and the corresponding 2D keypoint[Puj07].

For first phase evaluation, only on Synthetic images, PnP methods have been used along with an ICP based approach inspired from [PZC+17]. This evalaution is based on the functions [HMO16] listed in table 5.8.

ADD(*Average Distance of ModelPoints*)	$\text{avg}\left\|\bar{P}_x - \tilde{P}_x\right\|_2$
ADI (*Average Distance to closest ModelPoint*)	$\text{avg}\ \min\left\|\bar{P_{x1}} - \tilde{P_{x2}}\right\|_2$
ACPD (*Average closest point distance*)	$\min\ \text{avg}\ \left\|\bar{P}_x - \tilde{P}_x\right\|_2$
MCPD (*Maximum closest point distance*)	$\min\ \max\ \left\|\bar{P}_x - \tilde{P}_x\right\|_2$

Table 5.8: Functions used to evaluate pose.

The results are shown in table 5.9. These results are generated by performing PnP and ICP on different datasets (table 5.2) based on the predictions by Hourglass network with MSE loss (as per results from result section 5.2.3). Based on the data in the table,

it can be said that the Iterative method of PnP is producing the best results, with less ACPD and MCPD error and high IoU area, PnP's other method seems to work better on real data(Canister) and best pose estimations results are on Panda dataset. Some of the resulting projected points are shown in figure 6.3. P3P's mapping was not accurate because of the noise present in data and so it's not included for evaluation. *Others* has been used to signify EPNP, UPNP, and DLS as the results of these 3 methods were exactly same and so they were replaced by single name.

Dataset	Method	ADD	ADI	ACPD	MCPD	IoU
Dog	*Others*	0.3083	0.0072	0.0546	0.1346	0.3451
	Iterative	0.2987	0.0083	**0.0512**	**0.1306**	**0.3897**
	ICP	0.3185	0.0155	0.0611	0.1070	0.3256
Panda	*Others*	0.0561	0.0018	0.0087	0.0203	0.7450
	Iterative	0.0259	0.0013	**0.0039**	**0.0076**	**0.8448**
	ICP	0.0399	0.0020	0.0062	0.0107	0.7844
Canister	*Others*	0.0676	0.0064	0.0041	0.0079	0.8115
	Iterative	0.783	0.0208	**0.0036**	**0.0075**	**0.8186**
	ICP	0.9950	0.0637	0.0039	0.0084	0.7102

Table 5.9: Scores [HMO16] of PnP methods on a different dataset. *others* signifies (EPNP, UPNP, DLS)

A further evaluation on the dataset from table 5.2 has been performed to measure the results of network trained with synthetic data on real world data. The results have been then measured on both synthetic and real world data for comparison purpose, using the functions mentioned in table 5.11 and the results are listed in tables 5.10. The final drawn meshes are shown in figure 6.3 and 6.4 and the results are compared in graphs 5.5.

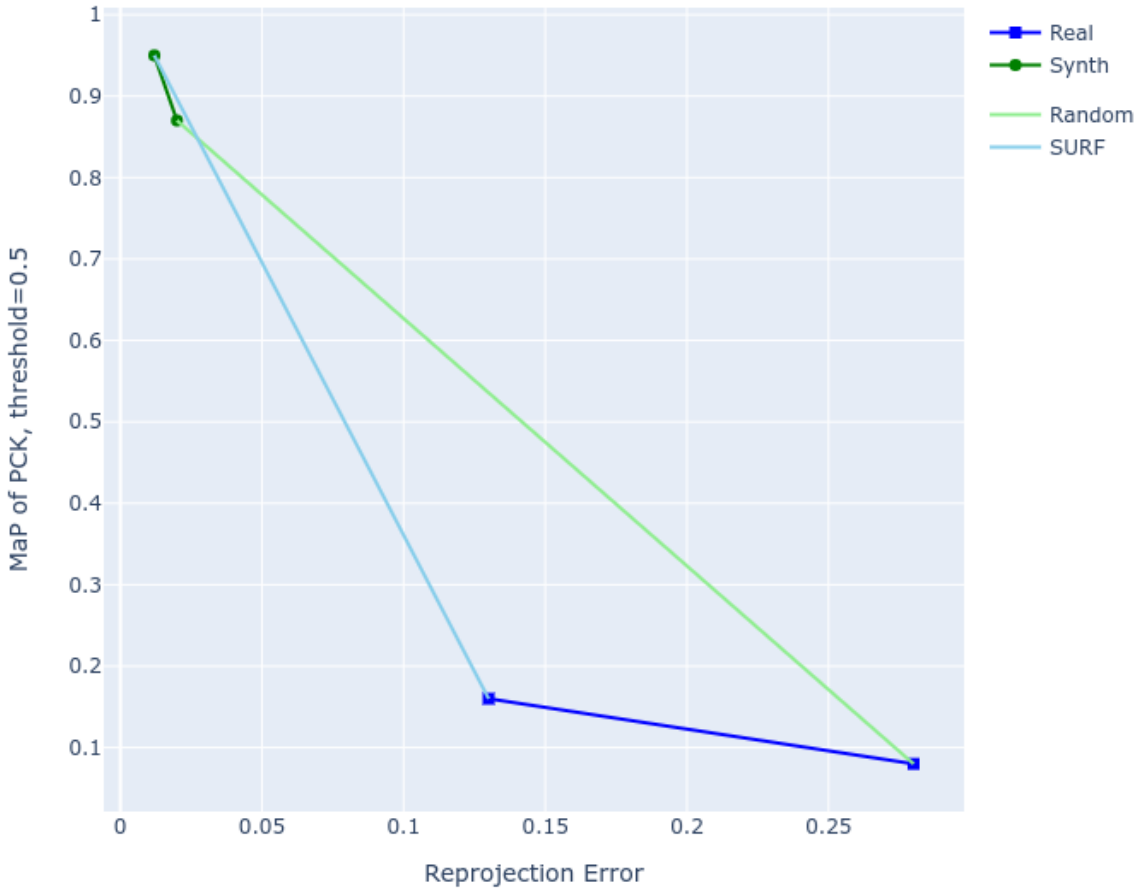
Dataset	KP	PnP	Type	RErr	TErr	RPErr	IOU	OKS	PCK	IOU
								$AP_{0.5}$	$AP_{0.5}$	$AP_{0.5}$
Buffalo$^{(1000)}$	*SURF*	*O*	R	**2.17**	**352.44**	**0.13**	**0.08**	**0.012**	**0.16**	**0.0**
			S	1.57	928.32	0.012	0.42	0.33	0.95	0.35
		I	R	2.48	1366.87	0.15	0.07	0.010	0.15	0.0
			S	**1.57**	**28.42**	**0.0009**	**0.63**	**0.91**	**0.99**	**0.81**
	Random	*O*	R	2.27	1148.55	0.28	0.05	0.010	0.076	0.0
			S	1.57	967.30	0.02	0.37	0.26	0.87	0.38
		I	R	2.69	1732.97	0.29	0.05	0.010	0.08	0.0
			S	1.57	29.46	0.0016	0.57	0.87	0.98	0.80

Table 5.10: Results on Synthetic and Real data, **O** signifies PnP methods *(EPNP, UPNP, DLS)*, **I** signifies PnP Method *Iterative*, **R** signifies *Real* dataset and **S** signifies *Synthetic* dataset.

RError	$\arccos\left(\left(\mathrm{Tr}\left(R\tilde{R}^{-1}\right)/2\right)-1\right)$	R : Ground truth Rotation $\tilde{R}$: Estimated Rotation after solving PnP using predicted keypoints
TError	$\left\|\bar{T}-\tilde{T}\right\|_2$	T : Ground truth Translation $\tilde{T}$: Estimated Translation after solving PnP using predicted keypoints
RPError	$\left(\left\|\bar{P}-\tilde{P}\right\|_2\right)^2$	P : Ground truth Points $\tilde{P}$: Re-projection of predicted points
IoU	$area\left(\bar{B}\cap\tilde{B}\right)$ / $area\left(\bar{B}\cup\tilde{B}\right)$	IOU of mask area between GT and Re-projection
OKS	$\exp\left(-\frac{d_i^2}{2s^2k_i^2}\right)$	d_i : L2 norm between GT and Re-projection s : sqrt(boundig box area/image area) $\mathrm{k}_i : \mathbb{E}\left[d_i^2/s^2\right]$, per-keypoint standard deviation with respect to object scale s
PCK	$bool\left(\frac{d_i \leq 0.05 * diagonal}{n}\right)$	d_i : Euclidean distance between GT and Re-projection diagonal : Diagonal of the bounding box n : Number of keypoints
$\mathbf{AP}_{th}$	$\mathrm{auc}(R,P)$	auc : area under curve R : Recall $\left(\frac{TP}{TP+FN}\right)$ P : Precision $\left(\frac{TP}{TP+FP}\right)$ th : Threshold value (0.50)

Table 5.11: Functions used to evaluate keypoints. [LMB+14] [HMO16]

[a]



[b]

Figure 5.5: (a) : Reprojected keypoint precision against Reprojection error, (b) : Translation error comparing best results of Random vs SURF.

6 Conclusion

6.1 Summary

This application uses the PnP method to estimate the pose of the calibrated camera using 2D keypoints detected by CNN and 3D object points fetched using forwarding kinematics. The end-to-end pipeline comprises of many parts like creating synthetic data, training CNN, performing PnP and to achieve this, multiple tools and technologies have been used.

To develop a pipeline to estimate pose given an image, the compatibility and feasibility of various tools and software were tested, like TensorflowSharp could not be used as it's current version was not compatible with Unity. However, some of the tools did not only integrate properly but also proved to be very efficient like OpenCVSharp. This Unity's plugin was used for PnP mapping within Unity and using this plugin helped keep the technology stack clean. To make Unity to communicate with CNN, written in Python, NetMQ socket interface was used as it was compatible with both Unity and Python. The internal implementation of this pipeline is segregated in different flows, Generate, Control and Convert, each performing specific tasks.

As part of this thesis, the results of different CNN nets and PnP methods have been evaluated for their accuracy and efficiency on synthetic and real data. While producing synthetic data, it was being taken care that enough variations were introduced in images to bridge the reality gap. When CNN was evaluated for semantic keypoint detection, Hourglass CNN with MSE loss function produced quite precise results on both synthetic and real data. Using these detected keypoints when the pose was estimated using PnP, the results varied significantly, object to object. Also, different methods of PnP produced different results and the best results were seen using the Iterative method. On the gas canister dataset, PnP's Iterative method even performed better than ICP. All the CNN training has been done on 'Google Colab' and trained for #300 epochs. Further improvements in the results can be hoped to be seen given these networks are trained longer.

6.2 Challenges

The challenges faced during the implementation of this application were mainly technical and can be summarized in the below points.

- ***Linux based Operating system:*** The biggest challenge while implementing this application was the non-compatibility of tools with the operating system. For ex-

ample, Unity-3D is not very compatible with a Linux based operating system.

Some other challenges were also faced while integrating external plugins with Unity-3D. Unity allows external plugins to be integrated with, in the form of DLL file but these files are not compatible with Linux based operating system. To tackle this problem a lot of settings needed to be done on the local machine..

- ***Computational capabilities of the system:*** The available machines and computational resources were not efficient enough to perform some of the heavy tasks, like training CNN or using graphics for Unity. Running Unity-3D and Python simultaneously on the local system was making it crash again and again. Another problem was to train CNN, as it generally takes a lot of time and computation power, it simply could not be done on the local system. Google-Colab proved to be a very useful resource to perform the training tasks however even 'Google-colab' has some drawbacks like it does not allow a network to be trained for more than 10 hours.
- ***Finding CAD (.obj) files:*** Finding CAD files of desired objects was a little difficult as, they were either paid or not compatible with Unity-3D directly (Needed to be explicitly converted before using).

6.3 Future work

One of the main tasks to be done as part of future work is to improve the code's integrability with AR-Based devices as a service to be used with real data and also to make this application work better in a dynamic environment. Some technical changes could also be done to make the performance of this application even better. For example, OpenCV's extension OpenVSharp can directly be used from Unity's asset store, it is paid but it is better performing. Also, using TensorflowSharp as an integration utility to access CNN models directly inside the Unity engine. This can reduce the problem of network latency in the code and further enhance the performance.

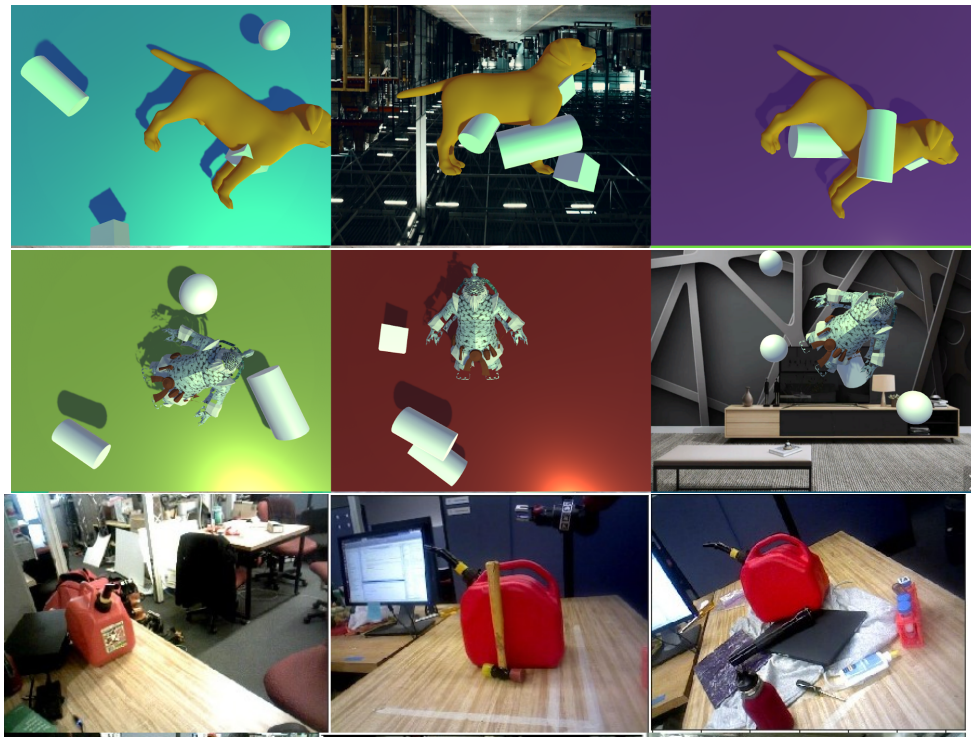


Figure 6.1: Synthetic and Real data for CNN evaluation

Figure 6.2: Synthetic and Real data for Pose evaluation (left column is Synthetic data and right column is Real data)

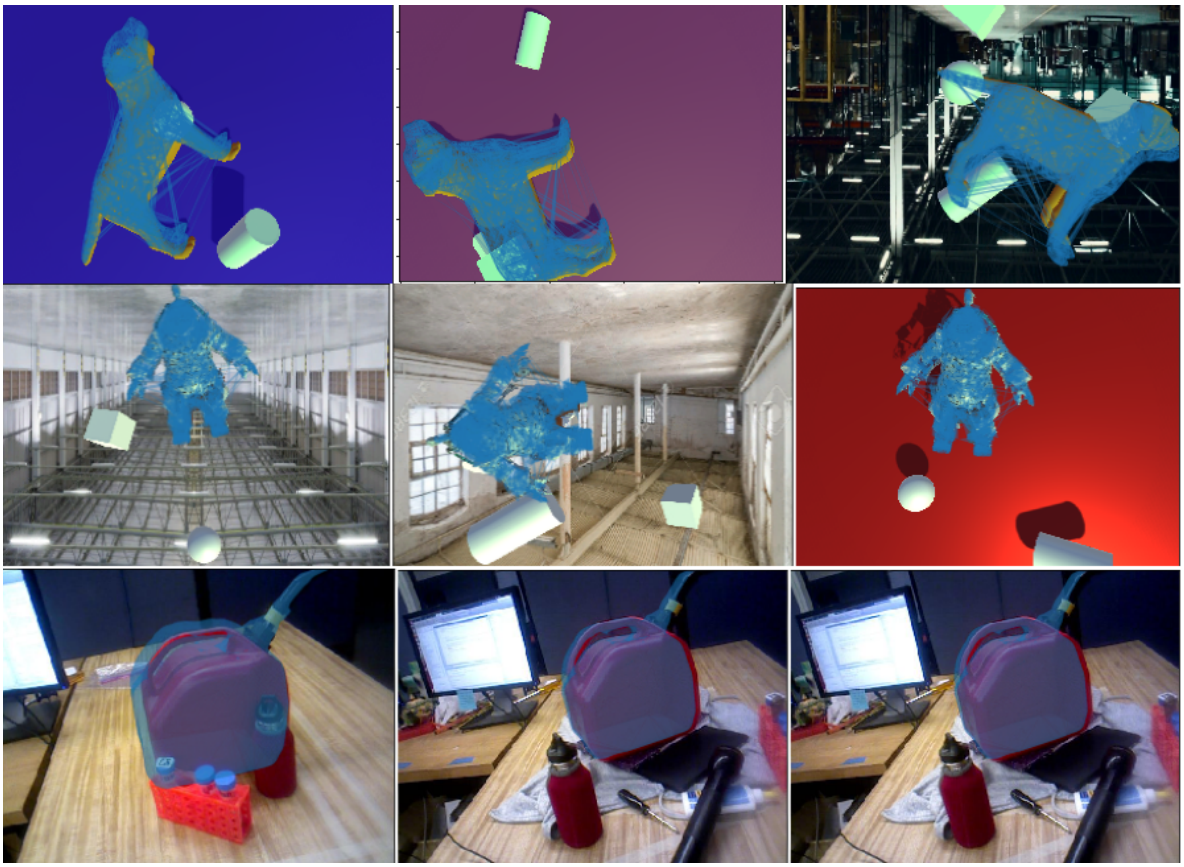


Figure 6.3: Reprojected mesh after PnP.

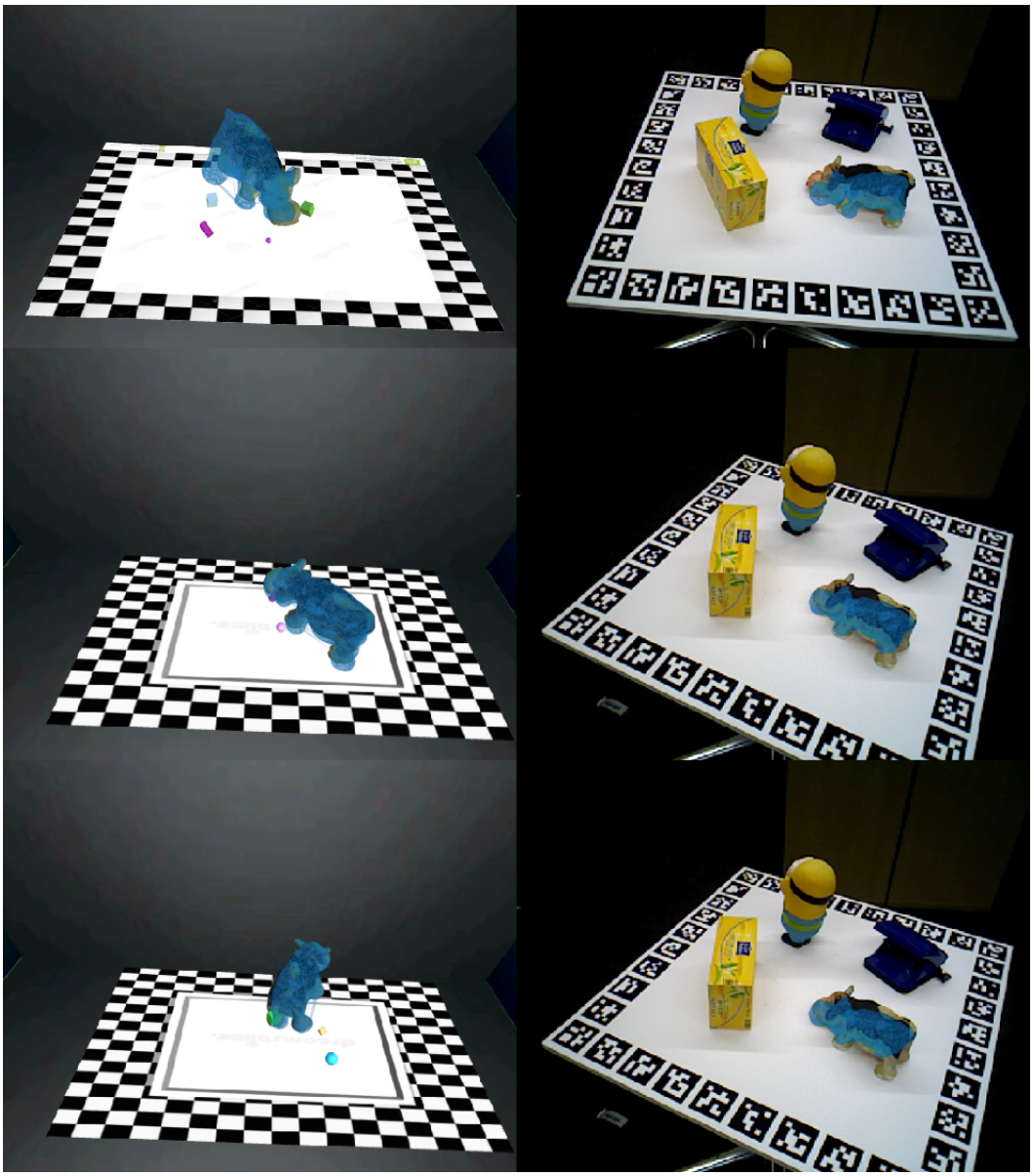


Figure 6.4: Synthetic and Real data for Pose evaluation (left column is Synthetic data and right column is Real data)

References

[Ama17] Amazon robot research project: *Amazon picking challenge*, 2017. [Online; accessed December-2019].

[Dig18] Digitalocean, Lisa Tagliaferri: *How To Install Anaconda on Ubuntu 18.04 [Quickstart]*, 2018. [Online; accessed July-2019].

[Goo19] Google research: *Google Colaboratory*, 2019. [Online; accessed September till December-2019].

[HMB+18] HodaÅ, TomÃ¡Å¡, Frank Michel, Eric Brachmann, Wadim Kehl, Anders Glent Buch, Dirk Kraft, Bertram Drost, Joel Vidal, Stephan Ihrke, Xenophon Zabulis and et al.: *BOP: Benchmark for 6D Object Pose Estimation*. Lecture Notes in Computer Science, page 19â35, 2018.

[HMO16] Hodan, Tomás, Juan E. Sala Matas and Stepán Obdrzálek: *On Evaluation of 6D Object Pose Estimation*. In *ECCV Workshops*, 2016.

[HPL+18] Han, Seung, Gyeong Park, Woohyung Lim, Myoung Kim, Jung-Im Na, Ilwoo Park and Sung Chang: *Deep neural networks show an equivalent and often superior performance to dermatologists in onychomycosis diagnosis: Automatic construction of onychomycosis datasets by region-based convolutional deep neural network*. PLOS ONE, 13:e0191493, 01 2018.

[HR11] Hesch, J. A. and S. I. Roumeliotis: *A Direct Least-Squares (DLS) method for PnP*. In *2011 International Conference on Computer Vision*, pages 383–390, Nov 2011.

[JMP+17] Jafari, Omid Hosseini, Siva Karthik Mustikovela, Karl Pertsch, Eric Brachmann and Carsten Rother: *The Best of Both Worlds: Learning Geometry-based 6D Object Pose Estimation*. CoRR, abs/1712.01924, 2017.

[KGC15] Kendall, Alex, Matthew Grimes and Roberto Cipolla: *Convolutional networks for real-time 6-DOF camera relocalization*. CoRR, abs/1505.07427, 2015.

[LMB+14] Lin, Tsung-Yi, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr DollÃ¡r and C. Lawrence Zitnick: *Microsoft COCO: Common Objects in Context*. Lecture Notes in Computer Science, page 740â755, 2014.

[LMNF08] Lepetit, Vincent, Francesc Moreno-Noguer and Pascal Fua: *EPnP: An Accurate O(n) Solution to the PnP Problem*. International Journal of Computer Vision, 81:155–166, 2008.

[MCL18] Moon, Gyeongsik, Ju Yong Chang and Kyoung Mu Lee: *PoseFix: Model-agnostic General Human Pose Refinement Network.* CoRR, abs/1812.03595, 2018.

[mis18] missinglink.ai: *Convolutional Neural Network Architecture: Forging Pathways to the Future*, 2018. [Online; accessed November-2019].

[Net19] NetMQ: *NetMQ*, 2019. [Online; accessed October-2019].

[nix18] nixCraft: *Ubuntu Linux Install Nvidia Driver (Latest Proprietary Driver)*, 2018. [Online; accessed April-2019].

[NYD16] Newell, Alejandro, Kaiyu Yang and Jia Deng: *Stacked Hourglass Networks for Human Pose Estimation.* CoRR, abs/1603.06937, 2016.

[ÖVBS17] Özyesil, Onur, Vladislav Voroninski, Ronen Basri and Amit Singer: *A Survey on Structure from Motion.* CoRR, abs/1701.08493, 2017.

[PAM13] Penate-Sanchez, A., J. Andrade-Cetto and F. Moreno-Noguer: *Exhaustive Linearization for Robust Camera Pose and Focal Length Estimation.* IEEE Transactions on Pattern Analysis and Machine Intelligence, 35(10):2387–2400, Oct 2013.

[Puj07] Pujol, Jose: *The solution of nonlinear inverse problems and the Levenberg-Marquardt method.* GEOPHYSICS, 72(4):W1–W16, 2007.

[PZC+17] Pavlakos, Georgios, Xiaowei Zhou, Aaron Chan, Konstantinos G. Derpanis and Kostas Daniilidis: *6-DoF Object Pose from Semantic Keypoints.* CoRR, abs/1703.04670, 2017.

[RRKB11] Rublee, E., V. Rabaud, K. Konolige and G. Bradski: *ORB: An efficient alternative to SIFT or SURF.* In *2011 International Conference on Computer Vision*, pages 2564–2571, Nov 2011.

[SF19] Shavit, Yoli and Ron Ferens: *Introduction to Camera Pose Estimation with Deep Learning.* CoRR, abs/1907.05272, 2019.

[shi19a] shimat: *.NET Framework wrapper for OpenCV*, 2019. [Online; accessed July-2019].

[shi19b] shimat: *A sandcastle Documented class library*, 2019. [Online; accessed July-2019].

[TPA+18] Tremblay, Jonathan, Aayush Prakash, David Acuna, Mark Brophy, Varun Jampani, Cem Anil, Thang To, Eric Cameracci, Shaad Boochoon and Stan Birchfield: *Training Deep Networks with Synthetic Data: Bridging the Reality Gap by Domain Randomization.* CoRR, abs/1804.06516, 2018.

[TTS+18] Tremblay, Jonathan, Thang To, Balakumar Sundaralingam, Yu Xiang, Dieter Fox and Stan Birchfield: *Deep Object Pose Estimation for Semantic Robotic Grasping of Household Objects.* CoRR, abs/1809.10790, 2018.

[Uni19a] UNITY: *Installing the Unity Hub*, 2019. [Online; accessed April-2019].

[Uni19b] UNITY: *Native plug-ins*, 2019. [Online; accessed May/June-2019].

[Uni19c] UNITY: *Platform dependent compilation*, 2019. [Online; accessed July-2019].

[Uni19d] UNITY: *Unity Documentation/Camera.WorldToScreenPoint*, 2019. [Online; accessed June/July-2019].

[Uni19e] UNITY: *Unity Documentation/Physics.Raycast*, 2019. [Online; accessed June/July-2019].

[Uni19f] UNITY: *Unity User Manual (2019.2)*, 2019. [Online; accessed April-2019].

[uni19g] UNITYASSETSTORE, DUMMIESMAN: *Runtime OBJ Importer*, 2019. [Online; accessed May/Septemmber/October-2019].

[Wik11a] WIKIPEDIA CONTRIBUTORS: *Perspective-n-Point*, 2011. [Online; accessed August/October-2019].

[Wik11b] WIKIPEDIA CONTRIBUTORS: *Sobel operator*, 2011. [Online; accessed April-2019].

[Wik11c] WIKIPEDIA CONTRIBUTORS: *Structure from motion*, 2011. [Online; accessed May-2019].

[XSNF17] XIANG, YU, TANNER SCHMIDT, VENKATRAMAN NARAYANAN and DIETER FOX: *PoseCNN: A Convolutional Neural Network for 6D Object Pose Estimation in Cluttered Scenes*. CoRR, abs/1711.00199, 2017.

Annex

```
if (Input.GetMouseButtonDown(0))
{
    Vector3 mFar = new Vector3(Input.mousePosition.x, Input.mousePosition.y, Camera.main.farClipP
    Vector3 mNear = new Vector3(Input.mousePosition.x, Input.mousePosition.y, Camera.main.nearCli

    Vector3 mousePosF = Camera.main.ScreenToWorldPoint(mFar);
    Vector3 mousePosN = Camera.main.ScreenToWorldPoint(mNear);

    Debug.DrawRay(mousePosN, mousePosF - mousePosN, Color.green);

    RaycastHit hit;

    if (Physics.Raycast(mousePosN, mousePosF - mousePosN, out hit))
    {
        //Debug.Log($"hit.point {hit.point}  WorldToScreenPoint(hit.point) {Camera.main.WorldToSc
        Debug.Log($"Point3D {hit.point}  :  Point2D {Input.mousePosition} ");
        tw.WriteLine("(" + Input.mousePosition.x + ", " + Input.mousePosition.y + ") : " + (hit.p
    }
    tw.Flush();
}
```

Listing 1: Raycast on click of mouse

```
 private Vector3[] transformCADObject()
{
    // position
    float newX, newY, newZ;
    newX = UnityEngine.Random.Range(-5.0f, 5.0f);
    newY = UnityEngine.Random.Range(0.0f, -5.0f);
    newZ = UnityEngine.Random.Range(0.0f, 5.0f);

    Vector3 newPosition = new Vector3(newX, newY, newZ);
    // Rotation
    var newRot = Quaternion.Euler(0, UnityEngine.Random.Range(-180, 180), 0);
    //Debug.Log($"initPos  {initPos}");
    cadObj.transform.position = initPos - newPosition;
    cadObj.transform.rotation = newRot * initRot;

    Vector3[] transformedObjPoints = new Vector3[kpLen];
    int i = 0;
    foreach (var p3D in objPts)
    {
        //Vector3 obj = newRot * (new Vector3(p3D.X, p3D.Y, p3D.Z) - cadObj.transform.position) +
        Vector3 obj = new Vector3(p3D.X, p3D.Y, p3D.Z);
        // Apply same rotation to points as object
        obj = newRot * obj;
        // Apply same transaltion to points as object
        obj = obj + initPos - newPosition;
        //Debug.Log($"obj after {obj}");
```

```
        transformedObjPoints[i] = obj;
        i++;
    }
    Debug.Log($"PoBJECT TRANSFORME ");
    synth.OnSceneChange();
    return transformedObjPoints;
}
```

Listing 2: Transforming 3D object and 3D points.

```
foreach (var kp3D in transformed3dKeypoints)
{
    // In python, CNN, the Y—axis, is inverted and so height — Y
    Vector3 pnpKeyPoints = openCV.project3DPoints(kp3D, cam);
    pnpKpWriter.WriteLine(pnpKeyPoints.x + "," + (height — pnpKeyPoints.y));

    Vector3 gdKeyPoints = cam.WorldToScreenPoint(kp3D);
    gtKeyPoints.AppendLine(gdKeyPoints.x + "," + (height — gdKeyPoints.y));
    gdKpWriter.WriteLine(gdKeyPoints.x + "," + (height — gdKeyPoints.y));
}
```

Listing 3: Converting world point to image points

```
public KeyPoint[] getKeyPoints(Mat camMat, int nKeyPoints)
    {

        orb = ORB.Create(nKeyPoints);
        KeyPoint[] keyPoints = orb.Detect(camMat);

        return keyPoints;
    } gdKpWriter.WriteLine(gdKeyPoints.x + "," + (height — gdKeyPoints.y));
}
```

Listing 4: ORB keypoint detector.

```
public double[,] getCameraMatrix(Camera cam)
{

    var f = cam.focalLength;
    var sensorSize = cam.sensorSize;

    var sx = sensorSize.x;
    var sy = sensorSize.y;
    var width = cam.pixelWidth;
    var height = cam.pixelHeight;

    //Debug.Log(" f " + f + "  sensor " + sensorSize + " : " + sx + " width/height " + width + "
:  " + height);

    double[,] cameraMatrix = new double[3, 3]
    {
        { (width*f)/sx, 0, width/2 },
        { 0, (height*f)/sy, height/2 },
        { 0, 0, 1}
    };
    Debug.Log($"cameraMatrix {(width*f)/sx}  {width/2} {(height*f)/sy} {height/2} ");
```

```
    return cameraMatrix;
}
```

Listing 5: Getting camera matrix

```
public void pnp(Point3f[] objPts, Point2f[] imgPts, Camera cam, SolvePnPFlags type)
{

    Vector3 rot = new Vector3();

    Debug.Log($" PNP   : {objPts.Length} : {imgPts.Length}");
    using (var objPtsMat = new Mat(objPts.Length, 1, MatType.CV_32FC3, objPts))
    using (var imgPtsMat = new Mat(imgPts.Length, 1, MatType.CV_32FC2, imgPts))
    using (var cameraMatrixMat = new Mat(3, 3, MatType.CV_64FC1, getCameraMatrix(cam)))
    using (var distMat = Mat.Zeros(5, 0, MatType.CV_64FC1))
    using (var rvecMat = new Mat())
    using (var tvecMat = new Mat())
    {

        Cv2.SolvePnP(objPtsMat, imgPtsMat, cameraMatrixMat, distMat, rvecMat, tvecMat, flags: typ
        //Debug.Log("Solved " + rvecMat + "  :  "+tvecMat);

        Mat rot_matrix = Mat.Zeros(3, 3, MatType.CV_64FC1);
        Mat tr_matrix = Mat.Zeros(3, 1, MatType.CV_64FC1);
        Mat proj_matrix = Mat.Zeros(3, 4, MatType.CV_64FC1);

        rot = new Vector3(rvecMat.At<float>(0), rvecMat.At<float>(1), rvecMat.At<float>(2));

        Cv2.Rodrigues(rvecMat, rot_matrix);
        tr_matrix = tvecMat;

        proj_matrix.Set<double>(0, 0, rot_matrix.At<double>(0, 0));
        proj_matrix.Set<double>(0, 1, rot_matrix.At<double>(0, 1));
        proj_matrix.Set<double>(0, 2, rot_matrix.At<double>(0, 2));

        proj_matrix.Set<double>(1, 0, rot_matrix.At<double>(1, 0));
        proj_matrix.Set<double>(1, 1, rot_matrix.At<double>(1, 1));
        proj_matrix.Set<double>(1, 2, rot_matrix.At<double>(1, 2));

        proj_matrix.Set<double>(2, 0, rot_matrix.At<double>(2, 0));
        proj_matrix.Set<double>(2, 1, rot_matrix.At<double>(2, 1));
        proj_matrix.Set<double>(2, 2, rot_matrix.At<double>(2, 2));

        proj_matrix.Set<double>(0, 3, tr_matrix.At<double>(0));
        proj_matrix.Set<double>(1, 3, tr_matrix.At<double>(1));
        proj_matrix.Set<double>(2, 3, tr_matrix.At<double>(2));

        rot = eulerAngle(rot_matrix);


        Debug.Log($"rot1 {cam.transform.eulerAngles} rot2 { rot * Mathf.Rad2Deg}");
        Debug.Log($"tr1: {cam.transform.position} tr2: {tr_matrix.At<double>(0)}, {tr_matrix.At<d
        projectionMatrix = proj_matrix;
    }
}
```

Listing 6: Solving PnP to get Projection matrix

```
public Vector3 project3DPoints(Vector3 obj, Camera cam)
{

    Mat point2d_vec = new Mat(4, 1, MatType.CV_64FC1);

    Mat cameraMatrixMat = new Mat(3, 3, MatType.CV_64FC1, getCameraMatrix(cam));

    Mat point3d_vec = new Mat(4, 1, MatType.CV_64FC1);
    point3d_vec.Set<double>(0, obj.x);
    point3d_vec.Set<double>(1, obj.y);
    point3d_vec.Set<double>(2, obj.z);
    point3d_vec.Set<double>(3, 1);

    point2d_vec = cameraMatrixMat * projectionMatrix * point3d_vec;

    var X2D = point2d_vec.At<double>(0) / point2d_vec.At<double>(2);
    var Y2D = point2d_vec.At<double>(1) / point2d_vec.At<double>(2);
    Vector3 pnpPoints = new Vector3((float)X2D, (float)Y2D, 0.0f);

    return pnpPoints;
}
```

Listing 7: Getting reprojected 2D points using PnP projection matrix

```
var mesh = cadObj.GetComponent<MeshCollider>().sharedMesh;
var vertices3D = mesh.vertices;
var normal3D = mesh.triangles;

TextWriter vertices = new StreamWriter("captures/GroundTruth/image_groundtruth_img—vertices.txt")
for (int i = 0; i < vertices3D.Length; i++)
    {
        Vector3 v =  cadObj.transform.rotation * vertices3D[i];
        v = Vector3.Scale(v , cadObj.transform.localScale);

        vertices.WriteLine(v.x + ", " + v.y + ", " +v.z);
    }
Debug.Log($"vertices saved");
vertices.Flush();

TextWriter faces = new StreamWriter("captures/GroundTruth/image_groundtruth_img—faces.txt");
for (int i = 0; i < normal3D.Length; i=i+3)
    {
        faces.WriteLine(normal3D[i] + ", " + normal3D[i+1] + ", " +normal3D[i+2]);
    }
faces.Flush();
```

Listing 8: Unity method to generate 3D vertices and faces of CAD object

```
public void start(int port)
    {
        ServerEndpoint = $"tcp://localhost:{port}";

        ForceDotNet.Force();
        Debug.Log($"C: Connecting to server...{ServerEndpoint}");

        client = new RequestSocket();
        client.Connect(ServerEndpoint);
```

```
    }

    // Update is called once per frame
    public void send(string fileName)
    {
        client.SendFrame(fileName);
        Debug.Log($"C: Frame sent...{fileName}");
    }

    public void sendMore(string fileName,string gtKeyPoints, string keypoints, bool control)
    {
        if(control){
            client.SendMoreFrame(fileName).SendMoreFrame(gtKeyPoints).SendMoreFrame(keypoints).Se
        }else{
            client.SendMoreFrame(fileName).SendMoreFrame(gtKeyPoints).SendFrame(keypoints);
        }
        Debug.Log($"C: Frame sent...{fileName}");
    }

    public string recieve()
    {
        string message = client.ReceiveFrameString();
        Debug.Log($"C: Frame recieved...{message}");
        return message;
    }

    public void close()
    {
        client.Disconnect(ServerEndpoint);
        client.Close();
        Debug.Log($"C: Frame disconnected...{ServerEndpoint}");
    }
```

Listing 9: Unity class to connect with python over network

```
from NeuralNetwork import NNFacade, NNConnector

def main():

    context = zmq.Context()
    socket = context.socket(zmq.REP)
    socket.bind("tcp://*:5000")

    print("Socket is bound...")

    nnFacade = NNFacade()

    while True:
        #  Wait for next request from client
        message = socket.recv_multipart()
        print("Request received... : %s" % message[0])

        retMessage = nnFacade.execute(message)
        #  Do some 'work'.
        #  Try reducing sleep time to 0.01 to see how blazingly fast it communicates
        #  In the real world usage, you just need to replace time.sleep() with
        #  whatever work you want python to do, maybe a machine learning task?
        time.sleep(0.1)
```

```
        #  Send reply back to client
        #  In the real world usage, after you finish your work, send your output here
        socket.send_string(retMessage)
        print("Request returned...: %s" % message[0])
```

Listing 10: Server class to provide socket endpoint to connect with Unity

```
class NNFacade():

    def __init__(self):
        pass

    def execute(self, message, draw=False):

        train = True if len(message) == 4 else False

        image = NNConnector.fileToImage(message[0])
        gtKeyPoints = NNConnector.byteToKPs(message[1])
        cvKeyPoints = NNConnector.byteToKPs(message[2])

        test_pts = Detector(image, gtKeyPoints, cvKeyPoints, train)
        return np.array2string(test_pts(draw))



class NNConnector():

    def byteToImage(dimensions, encodedImage):
        dims  = np.array(dimensions.decode().split(','), dtype=int)
        image = Image.frombytes('RGBA',(dims[0], dims[1]),encodedImage, 'raw')
        return image

    def fileToImage(filename):
        image = mpimg.imread(filename)
        return image

    def byteToKPs(keypoints, decodeType=None):

        if decodeType is None:
            keypoints = keypoints.decode()
        else:
            keypoints = keypoints.decode(decodeType)

        KPs = keypoints.split("\n")
        KPs = KPs if KPs[-1] else KPs[:-1]

        finalKPs = np.zeros((len(KPs), 2))
        for i, kp in enumerate(KPs):
            finalKPs[i] = np.array(kp.split(','))

        return finalKPs
```

Listing 11: Facade class of python.

```
class Normalize(object):
    """Convert a color image to grayscale and normalize the color range to [0,1]."""
```

```
    def __call__(self, sample):
        image, key_pts = sample['image'], sample['keypoints']

        image_copy = np.copy(image)
        key_pts_copy = np.copy(key_pts)

        # convert image to grayscale
        image_copy = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)

        # scale color range from [0, 255] to [0, 1]
        image_copy=  image_copy/255.0

        # scale keypoints to be centered around 0 with a range of [-1, 1]
        # mean = 100, sqrt = 50, so, pts should be (pts - 100)/50
        key_pts_copy = (key_pts_copy - 100)/50.0


        return {'image': image_copy, 'keypoints': key_pts_copy}


class Rescale(object):
    """Rescale the image in a sample to a given size.

    Args:
        output_size (tuple or int): Desired output size. If tuple, output is
            matched to output_size. If int, smaller of image edges is matched
            to output_size keeping aspect ratio the same.
    """

    def __init__(self, output_size):
        assert isinstance(output_size, (int, tuple))
        self.output_size = output_size

    def __call__(self, sample):
        image, key_pts = sample['image'], sample['keypoints']

        h, w = image.shape[:2]
        if isinstance(self.output_size, int):
            if h > w:
                new_h, new_w = self.output_size * h / w, self.output_size
            else:
                new_h, new_w = self.output_size, self.output_size * w / h
        else:
            new_h, new_w = self.output_size

        new_h, new_w = int(new_h), int(new_w)

        img = cv2.resize(image, (new_w, new_h))
        # scale the pts, too
        key_pts = key_pts * [new_w / w, new_h / h]

        return {'image': img, 'keypoints': key_pts}


class RandomCrop(object):
    """Crop randomly the image in a sample.

    Args:
```

```
        output_size (tuple or int): Desired output size. If int, square crop
            is made.
    """

    def __init__(self, output_size):
        assert isinstance(output_size, (int, tuple))
        if isinstance(output_size, int):
            self.output_size = (output_size, output_size)
        else:
            assert len(output_size) == 2
            self.output_size = output_size

    def __call__(self, sample):
        image, key_pts = sample['image'], sample['keypoints']

        h, w = image.shape[:2]
        new_h, new_w = self.output_size

        top = np.random.randint(0, h - new_h)
        left = np.random.randint(0, w - new_w)

        image = image[top: top + new_h,
                      left: left + new_w]

        key_pts = key_pts - [left, top]

        return {'image': image, 'keypoints': key_pts}
```

Listing 12: Image augumentation.

```
ADD = lin.norm(dataGT - dataPnP)
ADI = np.min([lin.norm(dataGT[i] - dataPnP[i]) for i in range(0, dataGT.shape[0])])
ACPD = np.mean([lin.norm(dataGT[i] - dataPnP[i]) for i in range(0, dataGT.shape[0])])
MCPD = np.max([lin.norm(dataGT[i] - dataPnP[i]) for i in range(0, dataGT.shape[0])])

IOU = np.divide(p1.intersection(p2).area, p1.union(p2).area)
```

Listing 13: PnP Score calculation.