ENGG*6100 Machine Vision

Assignment 3

Spencer Walls

Introduction

Problem: Given the magnitude of the online shopping sector in today's society, there is substantial interest in the automation of numerous operations in a standard online vendor warehouse. Such activities may consist of the transportation and packing/filling of numerous orders (which may include multiple items) within the warehouse. In order for robots to successfully complete these objectives, it is often required that they be able to diligently approximate the pose of the object (product) that they are dealing with. This can be achieved through the utilization of pose estimation algorithms, which can enable the robot to successfully pick-up/manipulate a wide array objects.

Dataset: The dataset is comprised of 10,368 2.5D images of multiple objects with varying degrees of clutter and from different viewpoints. The specifics of the dataset, which was published by the Rutgers University team in 2016, are as follows: 24 objects of interest; 12 bin locations per object; 3 clutter states per bin; 3 mapping positions per clutter state; and 4 frames per mapping position. Most importantly, the images are complete with hand-annotated 6-DOF poses. There is a YAML file available for each image which contains the transformation matrix between the camera and the ground truth pose of the object. 3D CAD mesh models for each object, which can assist in the training of machine learning and object detection algorithms, are also included in the dataset (Rennie et al., 2016).

Requirements: Design and implement a pose estimation algorithm for detecting and estimating the pose for a minimum of five of the objects in the dataset. Report your results in terms of translational and rotational error. Any existing pose estimation algorithm/model may be used as long as sufficient knowledge is demonstrated as to how the techniques work and the benefits/limitations of using them with respect to this dataset. Submit a report that summarizes your experiments, results, and findings via a discussion.

Solution: Two approaches to the present problem were implemented. The first approach involves traditional computer vision methods such as image processing and feature detection/matching, and finally pose estimation via the Coplanar POSIT algorithm. This approach unfortunately was wholly unsuccessful in terms of its ability to accurately approximate the pose of the five objects considered. Thus, a second approach, namely a Convolutional Neural Network (CNN) for regression, was implemented. This deep learning approach estimated the pose of the objects with acceptable accuracy, however has several limitations that must be noted. This assignment is therefore divided into two parts, Part 1 and Part 2, pertaining to the first approach and second approach, respectively. Considerably more weight is given to the second approach (CNN for regression) since it was the method that produced adequate results.

Part 1

Solution:

The first solution implemented to solve the given problem is multi-faceted and involves numerous stages. Concisely, these may be given as follows:

Step 1. Preprocess the image (in effort to remove redundant information).

Step 2. Detect the object (identify its coordinates in a 2D environment using feature detection).

Step 3. Pose estimation using Coplanar POSIT algorithm.

This algorithm is delineated in detail below. The "cheezit_big_original" object is used as an example when visualising the steps of the algorithm.

Algorithm description:

Step 1. Preprocess the image (in effort to remove redundant information).

This step involves acquiring the region of interest (ROI) through utilization of the mask that is provided in the dataset. First, the ROI is identified by computing the masked image. This is executed through use of the following command in OpenCV (Bradski, 2000):

```
mask_img = cv2.bitwise_and(img, img, mask = mask)
```

Using the "cheezit_big_original" image from the dataset as an example, executing this command results in the following masked image:

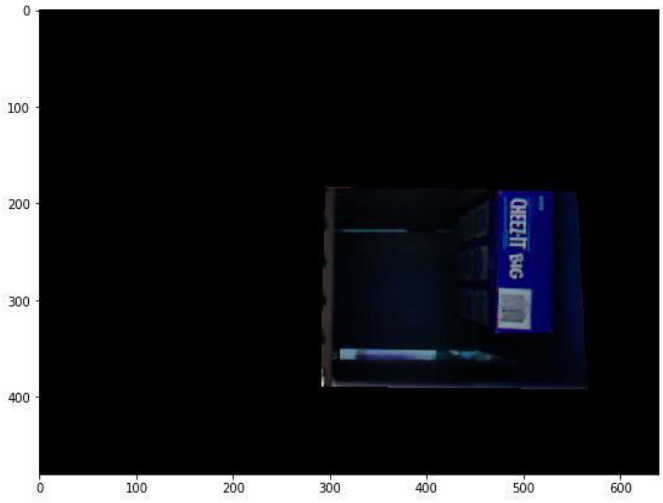


Figure 1 - Masked version of "cheezit_big_original" image

The ROI is then isolated and cropped, and the mask is removed, through use of the following commands:

```
img, contours, hierarchy = cv2.findContours(thresh, cv2.RETR_LIST, cv2.CHAIN_APPROX_SIMPLE)

img_final = cv2.minAreaRect(contours[0])
```

Executing these command results in the following image:

Figure 2 - Cropped ROI of cheezit_big_original image

Step 2. Detect the object (identify its coordinates in a 2D environment using feature detection).

This step pertains to the detection of the object by matching the object's features with that of the training images. This involves attempting to match as many points as possible between the testing and training images, and then employing Lowe's ratio test to determine the best matches. The selected algorithm for this task is Feature Matching + Homography, using Scale-invariant Feature Transform (SIFT) and OpenCV's Fast Approximate Nearest Neighbor Search (FLANN) library. The main commands used to implement this algorithm are as follows (Bradski, 2000):

```
M, mask = cv2.findHomography(src_pts, dst_pts, cv2.RANSAC,5.0)

matchesMask = mask.ravel().tolist()
```

The Random Sample Consensus (RANSAC) algorithm may then be used to address errors that may be present, by separating the matches that provide correct estimation (inliers) from the erroneous estimates (outliers). The number of minimum matches is set to 10, meaning that in order to consider the object successfully detected, a bare minimum of 10 matches must be present. If enough matches are discovered,

the locations of the matched key points are then extracted from both the train and test images. The perspective transformation can then be computed by inputting four of these points that are shared between the training and testing images to OpenCV's perspectiveTransform() function. This results in a 3 x 3 transformation matrix of the object of interest. This matrix is employed to transform the corners of the test image to corresponding points in the train image, and OpenCV's cv2.polylines() and drawMatches() functions are used to visualize the feature matching between the two images (Bradski, 2000):

```
img2 = cv2.polylines(img2,[np.int32(dst)],True,255,3, cv2.LINE_AA)
```

```
img3 = cv2.drawMatches(img1,kp1,img2,kp2,good,None,**draw_params)
```

Executing these commands results in the following image:

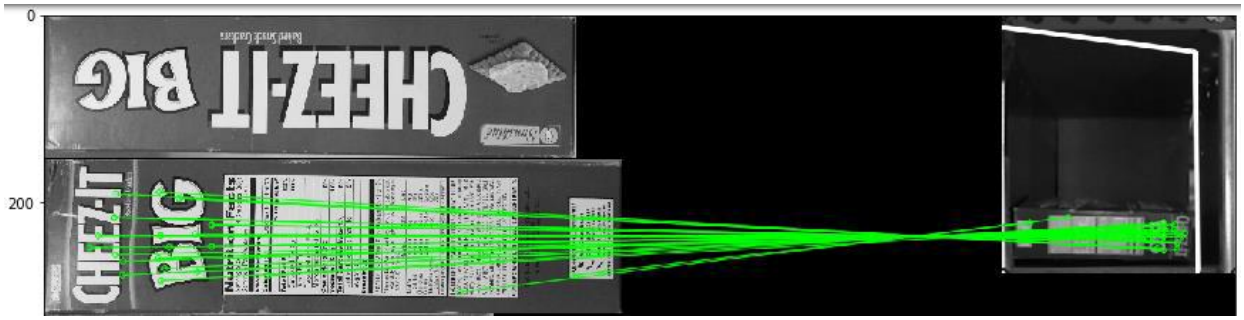


Figure 3 - Feature matching using SIFT

This object detection method was implemented on 5 of the objects as per assignment requirements, processing a total of 10 images for each object (50 images in total). The results of the object detection using Feature Detection + Homography for each of the 5 objects are presented in the Table 1, whereby the number of images for which the object was successfully detected is given out of 10.

Table 1 - Results of object detection using SIFT

Object	**Number of images for which the object was successfully detected (out of 10 images)**
cheezit_big_original	8/10
crayola_64_ct	6/10
expo_dry_erase_board_eraser	5/10
feline_greenies_dental_treats	6/10
genuine_joe_plastic_stir_sticks	7/10

Step 3. Pose estimation using Coplanar POSIT algorithm.

The Coplanar POSIT algorithm was implemented to estimate the pose of the object. POSIT stands for Pose from Orthography and Scaling with Iteration. This algorithm requires a minimum of four points on the object in addition to the focal length of the camera. An important distinction is that between the original POSIT algorithm (DeMenthon and Davis, 1995) and the Coplanar POSIT algorithm (Oberkampf et al., 1996), whereby the former demands that the original four points on the object are not all on the same plane, and the latter allows the four points to be on the same plane. In the present assignment, the Coplanar POSIT algorithm was implemented using the AForge.NET framework (AForget.NET, 2011). Given four object points and the focal length as input, the Coplanar POSIT algorithm iteratively estimates the transformation matrix (pose) of a given object whilst trying to minimize the error with respect to the ground truth values of the pose. Example output generated by the Coplanar POSIT algorithm with respect to the "cheezit_big_original" image displayed in Step 1 is given in Figure 4.

0.9979	-0.002	0.0654	0.0051
-0.0217	0.8677	0.4967	1.7044
-0.0577	-0.497	0.8658	-6.5715
0	0	0	1

Rotation matrix			Translation vector
-0.1344	0.0360	0.9903	0.0020
0.0839	-0.9953	0.0476	0.0834
0.9874	0.0895	0.1308	0.6936

Figure 4 - Pose estimation of cheezit_big_original image. Left: output of Coplanar POSIT algorithm for pose estimation using AForge.NET framework (note that the first three columns comprise the rotation matrix, whereas the last column on the right comprises the translation vector; the bottom row can be ignored). Right: ground truth pose of the object consisting of the rotation matrix and translation vector. This algorithm clearly provides unsatisfactory results; therefore, another approach was implemented and is presented as the proposed algorithm to solve this problem (Part 2).

The approach to pose estimation that is outlined in this part of the assignment unfortunately yielded inaccurate and unacceptable results when trying to estimate the pose of the 5 objects considered. Figure 4 is an example of one such unacceptable output. The values of the pose estimate are no where close to the ground truth values. This was the case for every image that was passed to the algorithm. After five images for each object were passed to the algorithm to no avail, it was concluded that the algorithm is not the right choice for solving the given problem. As a result, a completely different approach was chosen to be implemented subsequently. The next part of this assignment (Part 2) delineates an approach that is orders of magnitude more optimal than the present approach, simply because it produces reasonably accurate pose estimates for all five of the objects considered.

Part 2

"*There is also an influx of new results in the area of machine learning that can potentially be applied for the problem of pose estimation for warehouse picking and it would be interesting to see the quality of such solutions given the available dataset*" – Rutgers University Team (Rennie et al., 2016).

Solution:

The second approach employed to solve the given problem is substantially different from the first approach. In addition, the second approach was very successful compared to the first, as it actually provides reasonably accurate pose estimations for the five objects considered. This approach involves a Convolutional Neural Network (CNN) that is employed specifically for regression. Most research involving the CNN algorithm pertains to classification; however, it is also a feasible candidate for regression problems, especially when the input pertains to digital imagery and the output pertains to continuous numerical variables. Such characteristics indeed define the present problem of pose estimation. Concisely, the steps taken to implement the algorithm may be summarized as follows:

Step 1. Data preprocessing.
Step 2. Train/test split and normalization.
Step 3. Training phase.
Step 4. Testing phase.
Step 5. Processing CNN output.

A detailed description of these steps, which were carried out in Python, is provided below.

Algorithm description:

Step 1. Data preprocessing.

Prior to the implementation of the CNN algorithm, the dataset needs to be processed. This preprocessing specifically pertains to the output of the algorithm, i.e. the transformation matrices. In their original state, the YAML files provided by the Rutgers University team are unsuitable to use as target output to train the deep learning model. Thus, instead of preprocessing the images (input data) as was necessary for the first approach employed in this report, the second approach demands the preprocessing of the output. All of the YAML files need to be iteratively parsed and concurrently the relevant information for the problem, i.e. the rotation matrices and translation vectors, needs to be extracted. Therefore, a directory was created that contained every YAML file for the five objects being considered (cheezit_big_original; crayola_64_ct; expo_dry_erase_board_eraser; feline_greenies_dental_treats; and

genuine_joe_plastic_stir_sticks), and a Python program was written which parsed the files and extracted the relevant information, which was then saved to a CSV file, i.e. the "object_poses.csv" file that is included in the assignment submission. Please see this file if you wish to inspect the format of the ground truth pose data. The Python program can be found in **Appendix A** (and was submitted as "read_ymls.py" in the assignment submission).

Once the output dataset has been converted into a format that is suitable for training a CNN, it may be imported using the following command:

```
dataset = pd.read_csv("object_poses.csv").values
```

The next step that needs to be taken is the importing of the actual images. All of the images of the five objects that are available in the APC dataset were used. Table 2 shows the total number of images that were used as input data and how images of each of the five objects comprise this number.

Table 2 - Distribution of input images between the five objects

Object	**Number of Images**
cheezit_big_original	420
crayola_64_ct	420
expo_dry_erase_board_eraser	432
feline_greenies_dental_treats	420
genuine_joe_plastic_stir_sticks	432
Total	2124

A total of 2124 images were included in the input dataset for the algorithm. Naturally, the output dataset also contains 2124 pose estimates, each of which pertains to the relevant object within each respective image. It is important to note that no separating of the images was done with respect to the presence of clutter, in contrast to what was done by the Rutgers University team (Rennie et al., 2016). All of the images of the five objects were simply combined into one comprehensive input dataset.

Step 2. Train/test split and normalization.

The CNN must undergo both a training and a testing phase. Therefore, the input and output data must be split into a training set and a testing set. A train/test split of 75/25 was implemented, i.e. 75% of the data was used for training the algorithm and the remaining 25% was used for testing (on unseen inputs). Thus, a total of 1593 images (and corresponding pose estimates) were used for the training set and the remaining 531 were used for the test set. The test dataset was separated into five different datasets, i.e.

one for each of the five objects. It is very important to note that the algorithm was trained only one time, whereby it was trained to estimate the poses of the five different objects all at once. The training dataset thus was not partitioned with respect to the five objects. This allows for a much more flexible algorithm than would be the case if the algorithm was trained on each object individually. In the testing phase, the CNN was deployed to predict the five objects separately, which allows for results to be reported for each individual object. Training on all objects at once and testing on individual objects allows for the CNN to be quite robust; in its current state, it has the capacity to approximate the poses for all five objects without needing to be trained on each object separately.

The train_test_split() function of the sklearn.metrics class was used to allocate separate training and testing datasets for every object, via the following five commands:

```
(cheeseYtrain, cheeseYtest, cheeseXtrain, cheeseXtest) = train_test_split(cheeseY, cheeseX, test_size=0.25)
(crayolaYtrain, crayolaYtest, crayolaXtrain, crayolaXtest) = train_test_split(crayolaY, crayolaX, test_size=0.25)
(expoYtrain, expoYtest, expoXtrain, expoXtest) = train_test_split(expoY, expoX, test_size=0.25)
(felineYtrain, felineYtest, felineXtrain, felineXtest) = train_test_split(felineY, felineX, test_size=0.25)
(genuineYtrain, genuineYtest, genuineXtrain, genuineXtest) = train_test_split(genuineY, genuineX, test_size=0.25)
```

The testing datasets are kept separate because in the testing phase the algorithm is tested on one object at a time. However, since the algorithm is trained only once – on all five objects at the same time – the training datasets for each object need to be concatenated into one final training set, which collectively contains the training sets of all five objects. This is done through execution of the following commands, which combine all of the training sets for the input images and all of the training sets for the target output:

```
total_Xtrain = cheeseXtrain + crayolaXtrain + expoXtrain + felineXtrain + genuineXtrain

total_Ytrain = np.zeros((1593,12))
total_Ytrain[0:315,:] = cheeseYtrain
total_Ytrain[315:630,:] = crayolaYtrain
total_Ytrain[630:954,:] = expoYtrain
total_Ytrain[954:1269,:] = felineYtrain
total_Ytrain[1269:1593] = genuineYtrain
```

It is also important to “shuffle” the training set so to speak (so that the order in which the CNN receives the input images is completely random), which can be done by again using the train_test_split() function, this time setting the test_size parameter to zero so the data is simply shuffled around in a random-fashion:

```
(final_Ytrain, NULL1, final_Xtrain, NULL2) = train_test_split(total_Ytrain, total_Xtrain, test_size=0)
```

Lastly, in order for the algorithm to function properly, i.e. return acceptable output in a reasonable amount of time, and also to avoid overfitting, it is necessary to rescale the input images. This is done simply by dividing each pixel value by 255, which rescales each pixel to a value between 0 and 1. This was done for both the training and testing input images, via the following commands:

```
final_Xtrain_scaled = (np.array(final_Xtrain))/255

cheeseXtest_scaled = (np.array(cheeseXtest))/255
crayolaXtest_scaled = (np.array(crayolaXtest))/255
expoXtest_scaled = (np.array(expoXtest))/255
felineXtest_scaled = (np.array(felineXtest))/255
genuineXtest_scaled = (np.array(genuineXtest))/255
```

Step 3. Training phase.

After Steps 1 and 2, the input images can be passed to the CNN and the training phase can commence. Python code for the CNN is provided below; however, the specific details of every function are not discussed in detail. The specifics of the most important steps are provided in **Appendix B**. This program may be found in the "CNN_pose_estimation.py" file included with the assignment submission.

```
# Initialize CNN
regressor = Sequential()

# Step 1 - Convolution
regressor.add(Convolution2D(filters = 32, kernel_size =  3, strides = 3, input_shape = (480, 640, 3),
activation = 'relu'))

# Step 2 - Pooling
regressor.add(MaxPooling2D(pool_size = (2,2)))

# Step 3 - Add second Conv and Pooling layers
regressor.add(Convolution2D(32, 3, 3, activation = 'relu'))
regressor.add(MaxPooling2D(pool_size = (2,2)))

# Step 4 - Flattening
regressor.add(Flatten())

# Step 5 - Full connection
regressor.add(Dense(units = 128, activation = 'relu'))
regressor.add(Dense(units = 12, activation = 'linear'))

# Step 6 - Compile
regressor.compile(optimizer = 'adam', loss = 'MSE', metrics = ['MAE'])

# Step 7 - Fit CNN to the images
regressor.fit(final_Xtrain_scaled, final_Ytrain, batch_size=10, epochs=10,verbose=1)
```

It is important to note that the CNN is being used for regression rather than classification; thus, the algorithm is a *regressor* rather than a *classifier*. Thus, a couple model parameters need to be changed from what they are typically set to, in order for the CNN output to be predictions of continuous variables that are reasonably accurate.

Firstly, the final activation function in the output layer of the fully-connected section of the network needs to be a *linear activation function*:

$$linear\ activation\ function\colon\ f(x) = x$$

This allows for the output of the CNN to be continuous numerical values, rather than probabilities between 0 and 1. Since the output of the algorithm pertains to the transformation matrix which is a 3 x 4 matrix (including both the rotation matrix and translation vector), the CNN contains a total of 12 output neurons, i.e. one for each variable of the transformation matrix.

Secondly, the objective function of the CNN needs to be one that is suitable for regression. The mean-squared-error function was therefore selected to be the objective function that the algorithm minimizes:

$$MSE = \frac{1}{n}\sum_{i=0}^{n}(y_i - \widehat{y_i})^2$$

where y_i = the ground truth values; $\widehat{y_i}$ = the predicted values; and n = number of observations.

Other than these two fundamental parameter changes that need to be made, the model architecture of the CNN for regression is very similar to that of a CNN used for classification.

When training the CNN, the number of epochs was set to 10, meaning that the full training dataset (all 1593 images) is passed through the network a total of 10 times. It should be noted that network contains both two convolutional and two pooling (max pooling) layers, in effort to optimize performance and promote feature learning. The activation function that is used in both the convolutional layers as well as the hidden layer of the fully-connected network is the rectified linear unit (ReLU) activation function (Jianxin, 2017):

$$ReLU\colon f(x) = \max(0, x)$$

A more detailed description of the theory and dynamics of the CNN model may be found in **Appendix C**.

Step 4. Testing phase.

After the model has been sufficiently trained for 10 epochs, the testing phase can commence. This involves testing the model for a total of five times, i.e. once for each individual object. In this stage, all testing images pertaining to the given object (either 105 or 108, depending on the object) are passed to the CNN, which then outputs a 1 x 12 vector for each image that contains the values of the rotation matrix followed by the values of the translation vector. The output of the CNN is thus in the format presented in Table 3, which shows the 1 x 12 vector predicted by the algorithm for one of the "cheezit_big_original" object images and compares this with the ground truth pose values.

Table 3 – Example comparison of ground truth pose with that predicted by CNN

	Translation vector			**Rotation vector**								
Ground truth pose	-0.021	0.081	0.691	-0.227	0.045	0.973	0.090	-0.994	0.067	0.970	0.102	0.222
CNN pose prediction	-0.026	0.077	0.686	-0.262	0.036	0.960	0.083	-1.004	0.060	0.949	0.097	0.186

The testing phase of the CNN is initiated by means of the single following command:

```
y_pred = regressor.predict(cheeseXtest_scaled, batch_size=None, verbose=2, steps=None)
```

In the above command, the argument cheeseXtest_scaled pertains to the scaled version of the test images for the "cheezit_big_original" object. Thus, the output of this command would be a 105 x 12 matrix that corresponds to predictions of the ground truth pose estimates contained in cheeseYtest from Step 2. The output is therefore pose estimates for 105 images of the given object. In order to alter this command to predict the poses of a different object, the cheeseXtest_scaled argument simply needs to be replaced with the identifier of another object, for example crayolaXtest_scaled for the "crayola_64_ct" object. The output would then be a 105 x 12 matrix that contains the pose predictions (for the crayola_64_ct object) for the 105 test images in crayolaXtest_scaled. After this command has been executed for all five objects, all of the requisite output data has been acquired. This is comprised of pose predictions pertaining to the five objects of interest, which collectively span 531 testing images.

Step 5. Processing CNN output.

Once the testing phase is complete, further data processing is necessary to sufficiently evaluate the accuracy of the CNN's pose estimates. Firstly, the pose estimates, which as raw output are contained within a 1 x 12 vector, need to be separated into 1 x 9 vectors for the rotation estimates and 1 x 3 vectors for the translation estimates. This can be done by executing the following commands:

```
ypred_rot = y_pred[:,3:12]
ypred_tran = y_pred[:,0:3]
ytest_rot = cheeseYtest[:,3:12]
ytest_tran = cheeseYtest[:,0:3]
```

In the above block of commands, the object of interest is again the "cheezit_big_original" object. In order to properly partition the pose estimation data into rotation and translation data for another object, cheeseYtest would simply need to be replaced by the testing data identifier for that object, for example crayolaYtest for the "crayola_64_ct" object. In order to display results in the desired format, the 1 x 9 rotation vectors generated by the above commands need to be converted into rotation angles in degrees. This needs to be done for both the pose estimates as well as the ground truth pose values, in order to properly compare these and accurately measure the error that is present. This can be done by implementing the following algorithm:

```
# Convert rotation matrix to angles in degrees
ypred_rotdeg = np.zeros((105,3))
for i in range(0,105):
        temp1 = ypred_rot[i,:].reshape(3,3)
        temp2 = rotationMatrixToEulerAngles(temp1)
        for j in range(0,3):
            ypred_rotdeg[i,j] = math.degrees(temp2[j])
```

The above algorithm converts the original 1 x 9 vector of rotation values into a 1 x 3 vector of angles in degrees, specifically for the predicted pose values which are denoted by the identifier ypred_rot. In order to implement the algorithm for the ground truth pose values, all that needs to be done is to replace ypred_rot with the identifier for the ground truth poses, i.e. ytest_rot (both of which were declared above). The code for the rotationMatrixToEulerAngles() function used in the above algorithm is given in **Appendix D**. Once the rotation values of the transformation matrix have been converted into angles in degrees, the Euclidean distance (L2 distance) may be calculated between the ground truth pose values and the CNN pose estimates for both the rotation and translation vectors. This is done through execution of the following two algorithms:

```
# Calculate L2 distance for rotation matrix #
L2_rot = np.zeros((105,1))
for i in range(0,105):
    for j in range(0,3):
        L2_rot[i] += ((ypred_rotdeg[i,j]) - (ytest_rotdeg[i,j]))**2
    L2_rot[i] = np.sqrt(L2_rot[i])

# Calculate L2 distance for translation matrix #
L2_tran = np.zeros((105,1))
```

```
for i in range(0,105):
    for j in range(0,3):
        L2_tran[i] += ((ypred_tran[i,j]) - (ytest_tran[i,j]))**2
    L2_tran[i] = np.sqrt(L2_tran[i])
```

The first of these algorithms computes the Euclidean distance between the rotation estimates (ypred_rotdeg) and the ground truth rotation values (ytest_rotdeg), while the second algorithm computes the Euclidean distance between the translation estimates (ypred_tran) and the ground truth translation values (ytest_tran). Once the necessary Euclidean distances have been calculated, a plot for each object may be created with the translational error on the x-axis and the rotational error on the y-axis, similar to the results displayed in Rennie et al. (2016). This can be done through execution of the following commands.

```
plt.scatter(L2_tran, L2_rot, s = 20)
plt.xlabel("Translation Dist (m)")
plt.ylabel("Rotational Dist (deg)")
plt.xticks([0.05,0.1,0.15,0.2,0.25,0.3,0.35,0.4])
plt.yticks([45,90,135,180])
axes = plt.axes()
axes.grid()
plt.suptitle("[INSERT OBJECT NAME]")
plt.axis(xmin=0,ymin=0,ymax=180)
```

The CNN may be further evaluated by computing the mean Euclidean distance as well as the mean absolute error (MAE), for both rotation and translation. This last step is executed by way of the following simple commands:

```
L2_rot_mean = L2_rot.mean()
L2_tran_mean = L2_tran.mean()

MAE_rot = mean_absolute_error(ypred_rotdeg, ytest_rotdeg)
MAE_tran = mean_absolute_error(ypred_tran, ytest_tran)
```

Results:

The CNN performed reasonably well when evaluated using the same performance metric (L2 distance) that was used in Rennie et al. (2016). Figure 5 displays scatter plots of rotational error in degrees vs. translational error in metres for the five objects of interest. Figure 5 demonstrates that the CNN algorithm produces acceptable pose estimates for the five objects. Rotational error is typically less than 45 degrees, ranging roughly between 0 and 30 degrees for all objects (with the exception of a few outliers).

Furthermore, translational error is usually less than 0.1 metres, roughly ranging between 0 and 0.08 metres for all objects. Table 4 presents the mean absolute error and mean Euclidean distance between the pose estimates produced by the CNN and the ground truth pose values for rotation and translation.

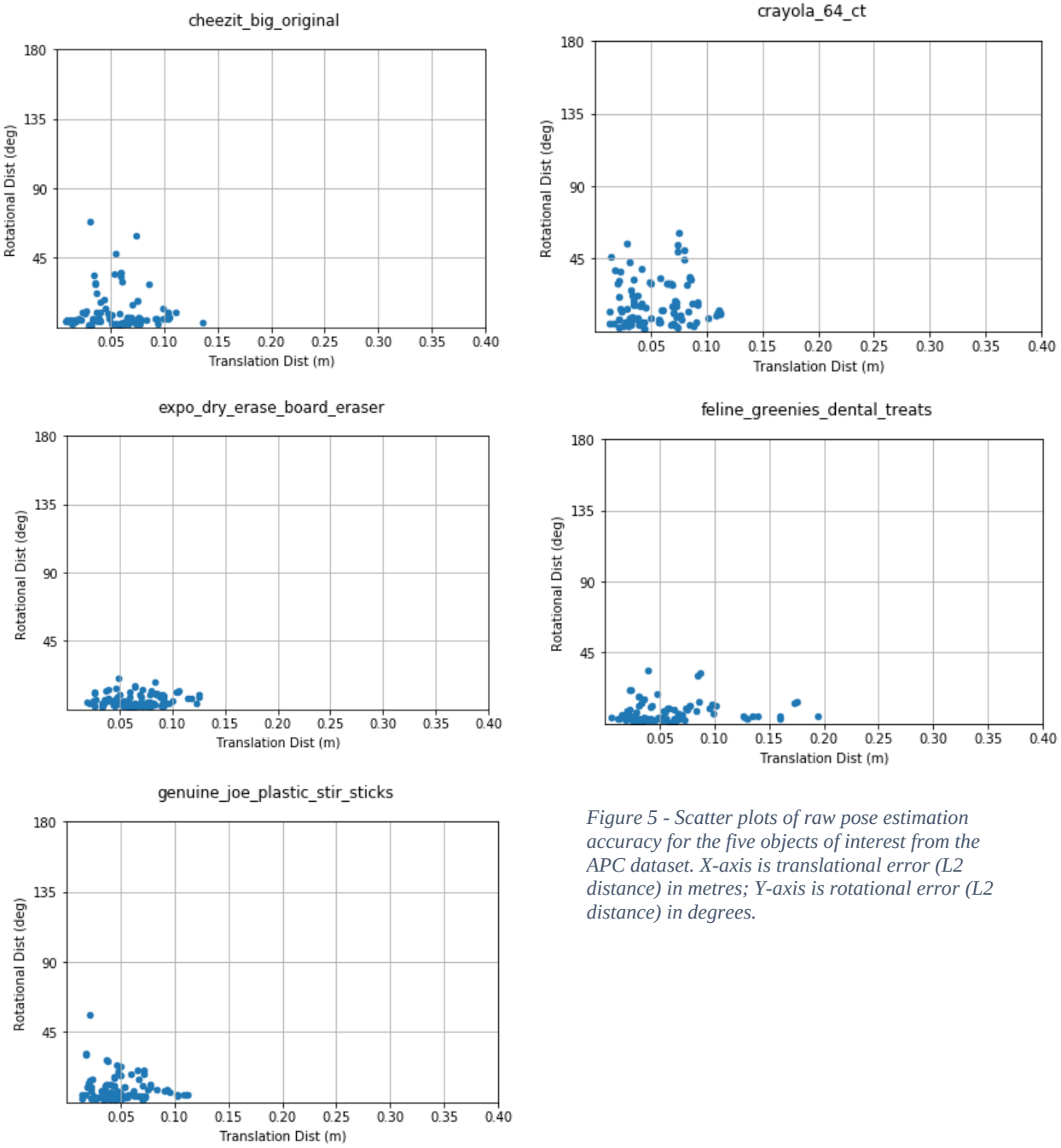


Figure 5 - Scatter plots of raw pose estimation accuracy for the five objects of interest from the APC dataset. X-axis is translational error (L2 distance) in metres; Y-axis is rotational error (L2 distance) in degrees.

Table 4 - Rotational and translational error for the five objects given as both mean L2 distance and mean absolute error

	Rotational error (degrees)		**Translational error (metres)**	
Object	**Mean L2 distance**	**MAE**	**Mean L2 distance**	**MAE**
cheezit_big_original	31.56	13.53	0.06	0.02
crayola_64_ct	39.51	15.27	0.06	0.03
expo_dry_erase_board_eraser	6.06	2.93	0.07	0.03
feline_greenies_dental_treats	40.12	14.51	0.06	0.03
genuine_joe_plastic_stir_sticks	51.06	18.66	0.05	0.03

Discussion:

The CNN boasts adequate performance for the pose estimation problem with respect to the five objects considered from the APC dataset. Table 4 demonstrates that the mean L2 distance ranged from 6.06 to 51.06 degrees and 0.05 to 0.07 metres, for rotational and translational error, respectively. Furthermore, the mean absolute error ranged from 2.93 to 18.66 degrees and 0.02 to 0.03 metres, for rotational and translational error, respectively. These results are far superior to those generated by the first approach (outlined in Part 1), which produced completely unreasonable and inaccurate pose estimates. When displayed on scatter plots, the results produced by the CNN model look quite similar to those displayed in Figure 4 of Rennie et al. (2016), in terms of rotational and translational error for individual objects.

Out of the five objects, the CNN most accurately estimated the pose of the expo_dry_erase_board_eraser object. This is namely attributed to the very small rotational error present, with a mean L2 distance of 6.06 degrees and mean absolute error of 2.93 degrees. In regard to translational error, however, the accuracy of the CNN was relatively consistent across all five objects. The algorithm produced the least accurate pose estimates (on average) for the genuine_joe_plastic_stir_sticks object, with a mean L2 distance of 51.06 degrees and a mean absolute error of 18.66 degrees for rotational error. In terms of translational error, the pose estimate accuracy of the genuine_joe_plastic_stir_sticks object was the same as the other objects. Table 3 demonstrates that when comparing algorithm performance with respect to the pose estimates of individual objects, the rotational error varies much more greatly than the translational error. Indeed, the mean absolute error for translational error was 0.03 metres for all objects except the cheezit_big_original object, for which the mean absolute error for translational error was 0.02 metres. On average, the algorithm's estimations of the translation vectors were not only more consistent but also more accurate than that of the rotation matrices.

Although the present CNN implementation boasts impressive results, the limitations of this approach should be noted. The main limitation of the algorithm is that it will only produce accurate pose estimates for the objects that it has been trained on. The algorithm will thus only produce accurate pose estimates for the cheezit_big_original, crayola_64_ct, expo_dry_erase_board_eraser, feline_greenies_dental_treats, and genuine_joe_plastic_stir_sticks objects. If the algorithm were to be tested on any of the other objects present in the APC dataset, its performance would likely degrade by an unacceptable amount. The present CNN algorithm that has been developed is therefore only suitable for predicting the pose of the five objects considered. However, if the algorithm is trained on all 25 objects of the APC dataset, there is no reason for it to not be able to produce accurate pose estimates for all 25 objects.

It should also be noted that in order to accurately estimate the pose of a given object, the algorithm must be trained on hundreds of images of the object in a wide variety of poses. This limitation has the potential to be quite problematic; for example, if a warehouse requires a robot to manipulate thousands of different objects, significant computational resources would be required to train a CNN that can accurately approximate the pose of all these objects. For such applications, the approach outlined here may be infeasible. Contrarily, the Coplanar POSIT algorithm that was outlined in Part 1 does not suffer from this same limitation; however, the POSIT implementation documented in this report produced unsatisfactory results. In theory, the POSIT algorithm does not need to be trained on numerous images of the relevant object prior to implementation in order to be successful, which is of course a huge advantage of the approach. Overall, the main limitation of the CNN algorithm is indeed the substantial amount of training that needs to be executed prior to any practical implementation.

Another minor limitation of the algorithm is that this implementation did not account for the differences between images with significant clutter, minor clutter, and no clutter. This adds some ambiguity to the results of the model, simply because it is not possible to determine if the algorithm performs better (or worse) when it estimates the pose of an object that is surrounded by some or no clutter. Nevertheless, this limitation is relatively minor, given that the overall performance of the algorithm was acceptable in terms of pose estimation for every object.

In conclusion, the main benefit of CNN approach is that it indeed produces serviceable pose estimates. Although the algorithm must be trained on the relevant objects prior to implementation, it boasts impressive results by using a regression framework to map input images of objects to accurate pose estimates. Overall, if the approach outlined in Part 1 were to be successful, it would be the recommended approach to take. This is due to the fact that the first approach does not require the abundance of object-specific training that the CNN requires prior to implementation (or the extensive computational resources

to accomplish this). Nevertheless, since the POSIT approach served to be infeasible for the present problem, the CNN model is the recommended approach to take when solving this problem.

References

Baldassi, C., Borgs, C., Chayes, J., Ingrosso, A., Lucibello, C., Saglietti, L., & Zecchina, R. (2016). Unreasonable Effectiveness of Learning Neural Networks: From Accessible States and Robust Ensembles to Basic Algorithmic Schemes. *PNAS.* https://doi.org/10.1073/pnas.1608103113

Bradski, G. (2000). The OpenCV library. *Dr. Dobb's Journal of Software Tools.*

C. Rennie, R. Shome, K. E. Bekris, and A. F. D. Souza, "A dataset for improved RGBD-based object detection and pose estimation for warehouse pick-and-place," in IEEE International Conference on Robotics and Automation (ICRA), 2016, (Stockholm, Sweden), 2016.

D. DeMenthon and L. S. Davis. Model-based object pose in 25 lines of code. International Journal of Computer Vision, 15:123–141, 1995.

D. Oberkampf, D. DeMenthon, and L. Davis, "Iterative pose estimation using coplanar feature points," Computer Vision and Image Understanding, vol. 63, pp. 495–511, 1996.

Francois Chollet et al. (2015). Keras documentation. Retrieved from: https://keras.io/

G. Bradski. The OpenCV library. *Dr. Dobb's Journal of Software Tools.* 2000.

Kumar, M., Raghuwanshi, N. S., Singh, R. (2011). Artificial neural networks approach in evapotranspiration modeling: A review. *Irrigation Science*, *29(*1), 11–25. https://doi.org/10.1007/s00271-010-0230-8

Tufféry, S. (2007) Data mining et statistique décisionnelle. L'intelligence de données, Modèles linéaire, Régression logistique, Réseaux de neurones, Scoring et Text mining. Edition TECHNIP. Paris.

Other sources: http://www.aforgenet.com/articles/posit/

Appendix A – Python program for parsing the YAML files and extracting the transformation matrices, which are then saved to a CSV file (in which each observation is a 1 x 12 vector).

```
import sys
from pathlib import Path
import csv
import ruamel.yaml

result = [['observation', 'rotation', 'translation']]
flatres = ["observation,tran1,tran2,tran3,rot1,rot2,rot3,rot4,rot5,rot6,rot7,rot8,rot9".split(',')]
yaml = ruamel.yaml.YAML()

for idx, file_name in enumerate(Path('.').glob('*.yml')):
  txt = file_name.read_text()
  if txt.startswith('%YAML:1.0'):
    txt = txt.replace('%YAML:1.0', "", 1).lstrip()
  data1 = yaml.load(txt)
  result.append([
   idx+1,
   data1['object_rotation_wrt_camera']['data'],
   data1['object_translation_wrt_camera'],
  ])
  row = [idx+1]
  row.extend(data1['object_translation_wrt_camera'])
  row.extend(data1['object_rotation_wrt_camera']['data'])
  flatres.append(row)

writer = csv.writer(sys.stdout)
writer.writerows(result)
print('---------')
writer = csv.writer(sys.stdout)
writer.writerows(flatres)
```

Appendix B – Detailed description of Python program for CNN.

The most fundamental steps that were taken to implement the CNN are documented below. The Python programming language was selected for the algorithm's implementation due to the presence of many useful libraries for scientific computing and particularly deep learning. The code is presented in blue to distinguish it from the step descriptions.

Step 1. Import necessary libraries and packages

```
from keras.models import Sequential
from keras.layers import Convolution2D
from keras.layers import MaxPooling2D
from keras.layers import Flatten
from keras.layers import Dense
```

The library that needs to be imported is the *keras* library which is an open-source neural network library (Francois Chollet et al., 2015). Python is an Object-Oriented programming language and therefore keras is considered to be a class. From the keras.models class, the Sequential subclass must be imported which is used to initialize the neural network; from the keras.layers class, Convolution2D, Maxpooling2D, Flatten, and Dense subclasses need to be imported. The names of these subclasses are self-explanatory and will be further elucidated in subsequent steps.

Step 2. Initialize the CNN

```
regressor = Sequential()
```

This step simply instantiates an object of the Sequential subclass, which is assigned "regressor" as its identifier. This essentially states that a neural network model is going to be created using keras, and that the model is going to be called "regressor".

Step 3. Create the Convolutional layer

```
regressor.add(Convolution2D(filters = 32, kernel_size =  3, strides = 3, input_shape = (480, 640, 3), activation = 'relu'))
```

This step creates the convolutional layer of the CNN. The filters argument indicates the number of feature maps that are going to be created, i.e. 32; the kernel_size argument indicates the size of the feature detectors that are going to be applied to the image to create the feature maps, i.e. 3 x 3 pixel feature detectors will be applied to the original image to create the feature maps; the strides argument indicates how many pixels the feature detector will slide in between each feature detection step; the input_shape argument specifies the size of the input image; and lastly, the activation argument specifies the activation

function that the feature maps are passed through prior to leaving the convolutional layer, which is the ReLU activation function.

Step 4. Create the Pooling layer

```
regressor.add(MaxPooling2D(pool_size = (2,2)))
```

This step creates the pooling layer, i.e. the max pooling layer in this case. The pool_size argument specifies the size of the filter that slides across the image and extracts the maximum value. Thus, a 2 x 2 pixel filter slides across the image, whereby every 4 pixel area is translated into one pixel by choosing the maximum value within this 4 pixel area to represent the area as one individual pixel. This essentially results in a pooling layer that is ¼ the size of the convolutional layer which reduces the dimensionality of the data and thus makes the image easier to process.

Step 5. Add a second convolutional layer and pooling layer

```
regressor.add(Convolution2D(filters = 32, kernel_size =  3, strides = 3, activation = 'relu'))

regressor.add(MaxPooling2D(pool_size = (2,2)))
```

This step simply adds a second convolutional layer as well as a second pooling layer. It can be seen here that these statements are the exact same as the statements in Steps 3 and 4. Adding these additional layers is necessary to improve the performance of the model by further augmenting the model's capacity to understand the data through *feature learning*.

Step 6. Flattening

```
regressor.add(Flatten())
```

This step simply flattens the output generated by the previous layer; this means that the previous 2D representation of the image is translated into a 1D vector that can be passed to the fully connected neural network in the next step.

Step 7. Add the fully connected neural network

```
regressor.add(Dense(units = 128, activation = 'relu'))
regressor.add(Dense(units = 12, activation = linear))
```

This step creates the fully connected neural network, i.e. it adds a hidden layer and an output layer via the Dense subclass of keras. The units parameter here indicates the number of neurons. The hidden layer therefore contains 128 neurons (this number was determined by trail-and-error); the output layer naturally

contains 12 neurons corresponding to the 12 different continuous values contained within the transformation matrices. The activation parameter once again pertains to the activation function implemented. The ReLU activation function was used for the hidden layer; the output layer employs a linear activation function in order to generate continuous-value predictions for each of the 12 values within the transformation matrices.

Step 8. Compile the model

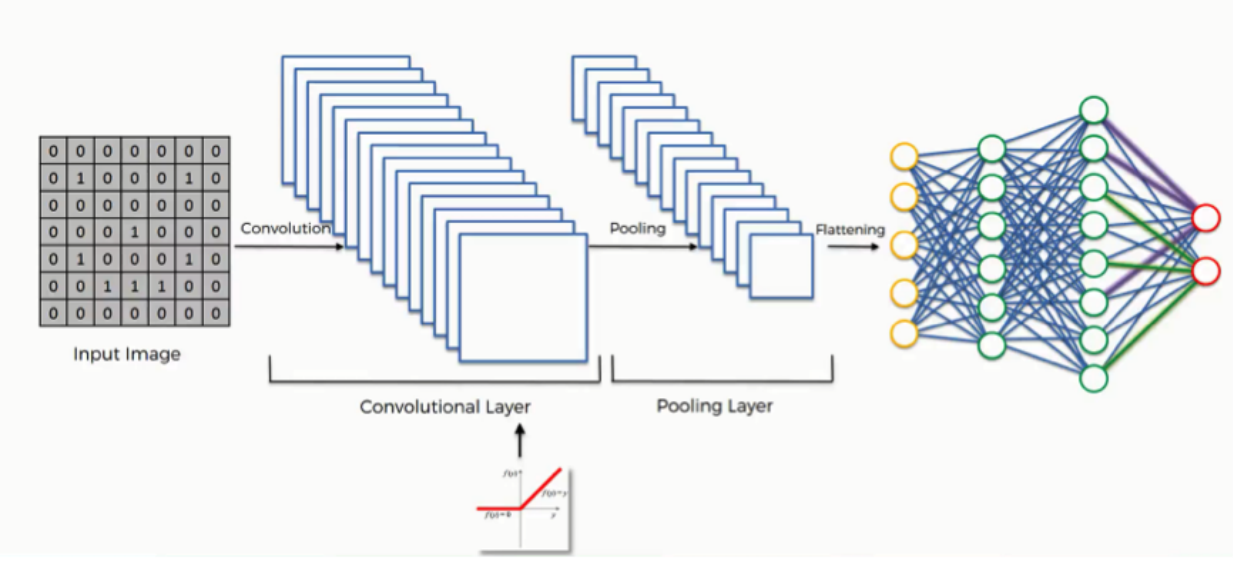
```
regressor.compile(optimizer = 'adam', loss = ‘MSE’, metrics = ['MAE'])
```

This step essentially brings the model together by specifying the optimization/learning algorithm (via the optimizer argument) to be employed as well as the error function (via the loss argument). The optimization/learning algorithm essentially optimizes the error function by minimizing it, therefore improving the accuracy of the model. Lastly, the final parameter specified by this statement, i.e. metrics, specifies the metric that will be used to measure the model’s performance; the *mean absolute error* metric was selected accordingly.

Appendix C – Detailed description of the theory behind the CNN approach.

The deep learning approach that was employed is the Convolutional Neural Network (CNN). The CNN is a certain type of Artificial Neural Network (ANN). An ANN model is essentially a nonlinear statistical technique that is inspired by and loosely modelled after the learning capabilities of a biological brain. ANNs are simplified mathematical models of biological neural networks which are comprised of thoroughly interconnected processing nodes (neurons) that are arranged in layers and connected via connection weights (Kumar et al., 2011). The weights are iteratively tuned by stochastic-gradient-based heuristic processes over a loss function, which serves to be a computationally demanding task. Nevertheless, provided that sufficient data are available, these models of computation have a profound ability to learn and generalize through the processing of training examples, recognizing patterns present in the behaviour of a system (Baldassi et al., 2016). The models learn, through their profound generalization capacity, from examples that contain consistent outputs relative to different combinations of inputs (Tufféry, 2007). These sophisticated computational models not only can reproduce the results of training examples but more importantly can predict the results of new and unseen examples with exceptional accuracy.

CNNs are a particular type of ANN that are widely used in computer vision tasks involving image classification and object detection. The CNN is loosely based off of the functionality of the visual cortex in a biological brain (Jianxin, 2017). The basic architecture of a CNN is given below in Figure 1.



source: *https://www.superdatascience.com/blogs/convolutional-neural-networks-cnn-summary*

The algorithm essentially involves an input image which is first inputted into what's called a *convolutional layer*. In the convolutional layer, the individual image is scanned by many different *feature detectors* (which is analogous to the concept of an image template in image processing), each of which is designed to detect a certain feature that might be present in the image. As each feature detector separately scans the input image, the mathematical operation of convolution is performed between the image and the feature detector. The output of this operation is a *feature map* which is the result of the convolution between the input image and the specific feature detector. As can be seen in Figure 1, the convolutional layer is comprised of many different feature maps. The output of the convolutional layer is then naturally the input to the pooling layer, after being put through the rectified linear unit (ReLU) activation function to eliminate any possible linearity in the feature maps (Jianxin, 2017).:

$$ReLU: \ f(x) = \max(0, x)$$

Multiple types of pooling are possible; in the present assignment, *max pooling* was employed. This involves scanning each feature map that was produced in the previous layer by a 3 x 3 grid, whereby the maximum value of these 9 cells is extracted and placed in a new smaller feature map as the representative of this 3 x 3 grid. The result is a compressed version of the original feature map; as can be seen in Figure 1, the pooling layer simply consists of the feature maps from the convolutional layer after max pooling has been applied to them. This helps reduce the dimensionality of the image which ultimately leads to more efficient image processing by the CNN. After the pooling layer, the pooled images are then flattened (transformed into a vector) and inputted into a regular feed-forward neural network, whereby each individual pixel is represented by a neuron in the network (Jianxin, 2017).

After the image is inputted pixel by pixel into the fully connected neural network, it then goes through one or more hidden layers (The above diagram displays two hidden layers; however, the present assignment actually only utilized one hidden layer). The neurons of the hidden layer contain the aforementioned ReLU activation function. The present assignment included one hidden layer that was comprised of 128 *hidden neurons.* The hidden layers essentially automate the process of feature selection through *feature learning*. In contrast to classic machine learning algorithms which typically demand hand-crafted features, a deep learning model such as a CNN does not require this; instead, the hidden layers work as a replacement for this process. After the data has been processed through the hidden layers, the output is finally inputted to the output layer, which contains a number of neurons equal to the number of continuous values being predicted (for regression). Note that in the above diagram, only two output neurons are present; a more accurate visualization of the present implementation would of course contain 12 output neurons. Each of the 12 output neurons contains a linear activation function, the output of which is a prediction for the value of the transformation matrix that the given neuron represents.

Appendix D – Python code for the rotationMatrixToEulerAngles() function used in the "L2dist_compute.py" program.

```
def rotationMatrixToEulerAngles(R):
    sy = math.sqrt(R[0,0] * R[0,0] +  R[1,0] * R[1,0])
    singular = sy < 1e-6
    if  not singular :
        x = math.atan2(R[2,1] , R[2,2])
        y = math.atan2(-R[2,0], sy)
        z = math.atan2(R[1,0], R[0,0])
    else :
        x = math.atan2(-R[1,2], R[1,1])
        y = math.atan2(-R[2,0], sy)
        z = 0
    return np.array([x, y, z])
```