Rheinische
Friedrich-Wilhelms-
Universität Bonn

Institute for Computer Science
Department VI
Autonomous Intelligent Systems

Rheinische Friedrich-Wilhelms-Universität Bonn

Master Thesis

Convolutional 6D Object Pose Estimation

Author:
Catherine Elisabeth Capellen

First Examiner:
Prof. Dr. Sven Behnke

Second Examiner:
Prof. Dr. Christian Bauckhage

Advisor:
Max Schwarz

Date: January 10, 2019

Declaration

I hereby declare that I am the sole author of this thesis and that none other than the specified sources and aids have been used. Passages and figures quoted from other works have been marked with appropriate mention of the source.

Place, Date

Signature

Abstract

6D Pose estimation describes the prediction of translation, rotation and object category of rigid objects in an RGB or RGB-D image. This information is important for the ability of robots to interact with objects in their environment.

PoseCNN [46] is a recent, partially convolutional, network architecture, that separates the estimation of the translation and rotation and shows state-of-the-art results on the OCCLUDED (Linemod) dataset. The authors of PoseCNN introduce the YCB dataset, a significantly larger dataset than the previous benchmarks, that is challenging due to occlusions, clutter and symmetric objects.

In this thesis we developed two fully-convolutional architectures, ConvPoseCNN and CoordPoseCNN, based on the PoseCNN architecture and compare them to PoseCNN on the YCB dataset. CoordPoseCNN estimates the visible object coordinates pixel-wise and recovers the pose by solving the PnP problem. ConvPoseCNN replaces the fully-connected rotation estimation branch of PoseCNN with a convolutional architecture that produces pixel-wise quaternion predictions. For the resulting quaternion predictions we evaluated different averaging and clustering strategies and experimented with weighting the predictions. ConvPoseCNN reaches comparable results to PoseCNN on the YCB dataset and yields advantages in the performance and the number of parameters.

Contents

1 Introduction

Given images of rigid objects, 6D Object Pose Estimation describes the problem of finding their object class, their position and their rotation. Recent research focuses on increasingly more difficult datasets with multiple objects, cluttered environments and partially occluded objects. For the rotation estimation, symmetric objects are challenging. A specific use-case where these challenges might occur, are robots as they could be used in a warehouse. Here the objects are known in advance and the environment might contain many objects, which occlude each other. This problem setting has received more attention in the recent years, for example in the Amazon Picking Challenge of 2015 and 2016 and in the Amazon Robotic Challenge of 2017 and 2018 (see Fig.1.1), where robots had to pick objects from a bin filled with many objects. Generally, robots trying to grasp or manipulate objects will benefit from a better pose estimation, which can be used in many applications including household helpers or assembly robots.

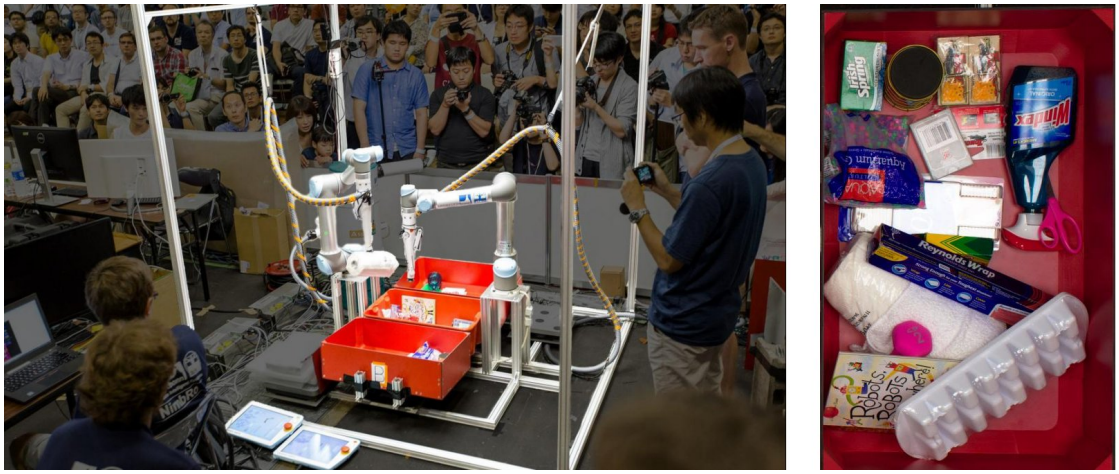


Figure 1.1: Example from the Amazon Robotic Challenge 2017. Pictures taken from [37].

While many pose estimation methods rely on RGB and depth data, recently more works consider the task of only using the RGB information. As Rad and Lepetit [32] point out, depth cameras might fail outside or with specular objects

Figure 1.2: A Region of Interest crop that contains more than one instance of the same object class and might lead to confusion for the rotation estimation.

and depth sensors consume more energy. Also, RGB cameras are already available on many devices.

Deep learning methods have shown great improvement for the solutions of many kind of problems that deal with images, pose estimation is no exception. Recently, PoseCNN [46] has delivered state of the art performance on their YCB dataset. PoseCNN decouples the problem of pose estimation into estimating the translation and rotation separately. It uses features given from the convolutional layers from VGG16 and then three different branches: Two fully convolutional branches estimate a semantic segmentation and center directions for every pixel of the image. The third branch consists of a RoI pooling and a fully-connected architecture which regresses each region of interest to a quaternion describing the rotation.

In this thesis we developed different fully-convolutional architectures evolved from PoseCNN. Our architectures are fully convolutional and have no fully connected layers. One of them uses the segmentation and translation estimation as in PoseCNN and changes the rotation estimation to a pixel-wise prediction of quaternions. We then infer a final quaternion. Another architecture predicts pixel-wise 3D object coordinate correspondences. Finding the pose is then equivalent to solving the Perspective-n-Point (PnP) problem.

One motivation for this is speed. Convolutional layers have far less parameters than fully connected layers and are faster to train. Also, it removes the necessity of RoI Pooling, which is a time-consuming procedure. Apart from that, our proposed changes unify the architecture and make it more parallel: PoseCNN first predicts the translation and the region of interests and then, using the regions, the rotation estimation. Our architecture can perform estimate the rotation independent of the translation estimation. We compare the results of our architectures to PoseCNN on the YCB dataset.

We also expect possible improvements for cases where multiple objects of the same class lie close enough, that they will be in the same RoI. Since the RoI gets no further information on which object it is supposed to predict the rotation of, it could possibly lead to confusion. An example is shown in Figure 1.2. However, since the YCB dataset does not feature any images with multiple instances, this hypothesis can not be verified on this dataset.

First we will introduce the algorithms and techniques in Chapter 2 that are used or compared against in this thesis. Afterwards, in Chapter 3, we briefly introduce some important Convolutional Neural Network architectures and then give an overview over the related work for 6D pose estimation. In Chapter 4 we describe the architecture of PoseCNN in detail, followed by our own architectures in Chapter 5. Then, in Chapter 6 we introduce in detail the YCB dataset, as well as, other used datasets and describe our experiments. At the same time we report our performance against PoseCNN. Finally, Chapter 7 summarizes our results and concludes this thesis.

2 Methods

Our work builds upon many different methods from Computer Vision and Deep Learning and consists of different problem settings. In this chapter we will give a brief introduction to the basic concepts that we use in our thesis.

2.1 Convolutional Neural Networks

Multilayer networks consist of layers with typically non-linear processing units called neurons and weighted connections which pass on the results of the previous layers. Usually these layers are fully connected, meaning that all neurons in one layer are connected to all neurons of the following layer. Neural networks try to learn a function where the internal parameters of the function are the weights of the model [22].

Convolutional Neural Networks (CNNs) are multilayer neural networks, which were developed for array-like input as images. These networks contain convolutional layers, which are inspired by the mathematical operation of the same name: For an image I, a small $m \times n$-dimensional array K, called kernel or filter, the convolution, $K * I$, of K and I at indices i and j is given by:

$$(K * I)(i, j) = \sum_{m_i=1}^{m} \sum_{n_i=1}^{n} I(i - m_i, j - n_i) K(m_i, n_i). \tag{2.1}$$

Convolutions are widely used in computer vision. The operation convolutes the image with a filter of a usually small size and produces another image as an output. Important to understand is that the value of a specific pixel, which is calculated by the operation, only takes information from a patch of the input image around this pixel position. The size of the patch corresponds to the size of the filter. The response will be large if the patch of the image around this pixel contains a feature which is similar to the weights inside the filter. Convolutions are interesting for processing images, since they look at local features. This reflects that pixels that are close together are more likely to form relevant connections than pixels which are far away from each other [12]. Most frameworks implement cross correlation

instead of the original convolution, which is defined as

$$(K * I)(i, j) = \sum_{m_i=1}^{m} \sum_{n_i=1}^{n} I(i + m_i, j + n_i) K(m_i, n_i). \tag{2.2}$$

This flips the filter, but in practice otherwise makes no difference.

Before the advent of CNNs, filters were hand designed to extract specific information from images, as for example the edges. A CNN will learn its own filters. A convolutional layer may contain a lot of feature maps, where each feature map performs a convolution on the previous layer. To achieve this, the neurons from a feature map will only be connected to a small, local patch from the previous layer. The size of the patch corresponds to the size of the convolutional filter. These connections share the same weights for all the neurons of the same feature map. Performing a forward pass is then the same as performing a convolution on the output of the last layer, where the weights are the values of the filter. Another advantage is that it might often not matter for a specific feature, where in the image it will appear. Since the filter is applied to every pixel of the input image, the whole image will be checked for this feature without the need to learn this feature for all the positions separately [12, 22].

There are four key ideas in CNNs: "Local connections, shared weights, pooling and the use of many layers" [12]. The first two ideas are reflected in the feature maps. The third idea is realized by pooling layers. Each neuron from the pooling layer is only connected to one patch of neighboring neurons from the layer before. It will take a summary from its inputs (for example the maximum or the average value) and pass this on. Pooling layers can also be used to reduce the dimension of the feature maps. An example is shown in Fig 2.1. A typical architecture of a CNN usually combines convolutional layers and pooling layers and ends with one or more fully-connected layers. Convolutional layers are usually followed by non-linear activation functions. The convolutional layers are supposed to identify features and the pooling layers are useful for downsampling. They can also make the network more robust to small translations in the input data [12, 22].

An advantage of convolutional layers is that they are in comparison usually much faster than using a fully-connected layer, since they have less parameters. This reduces both the amount of operations needed to compute the output of this layer as well as the size of the model in memory [12, 22].

The Rectified linear unit (ReLU), defined as $f(z) = \max(z, 0)$, is a popular choice as a non-linearity, since it enables deep networks to often learn faster [22].

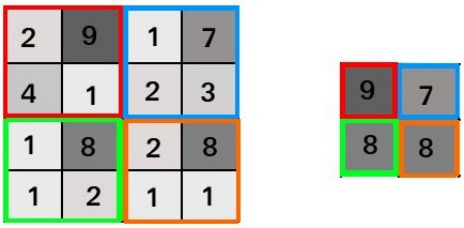


Figure 2.1: An example for Max pooling, for a 2-dimensional feature map. For each patch the maximum value is passed on.

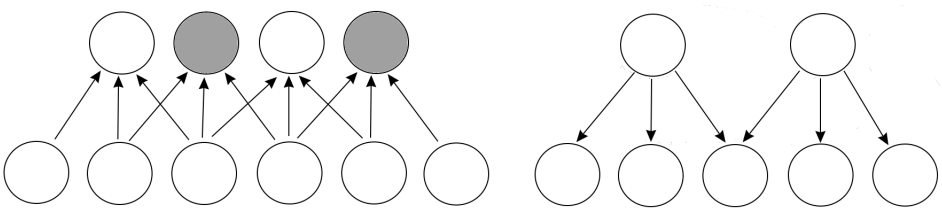


Figure 2.2: Left: example of a 2-dimensional convolution with kernel size 3. If the grey coloured neurons are removed it is a convolution with stride 2, meaning the kernel moves over the image skipping every second pixel. Right: example of a 2-dimensional deconvolution, which is equivalent to the backward pass of the convolution with stride 2.

2.1.1 Upsampling

While pooling and strided convolutions reduce the size of the feature maps, upsampling is the process of increasing their size. One possibility is using transposed convolution, sometimes inaccurately also called deconvolution, for this task. As Long, Shelhamer, and Darrell [25] describe it, this is essentially the same process as backpropagating the gradient through a regular convolutional layer. An example of that is shown in Figure 2.2.

2.1.2 Region of Interest Pooling

Some architectures predict bounding boxes called Region of Interests (RoIs) containing interesting parts of the image and then crop these parts to use them for the further prediction. The cropping does not need to happen on the image itself, but can also happen on the features. Since the layers of the neural network have a fixed input size it is necessary to change the size of the crop to this input size. A RoI Pooling Layer takes as input feature maps, and a RoI of size (h, w) and reduces the feature maps to a size of (H, W). It crops the rectangle given by the

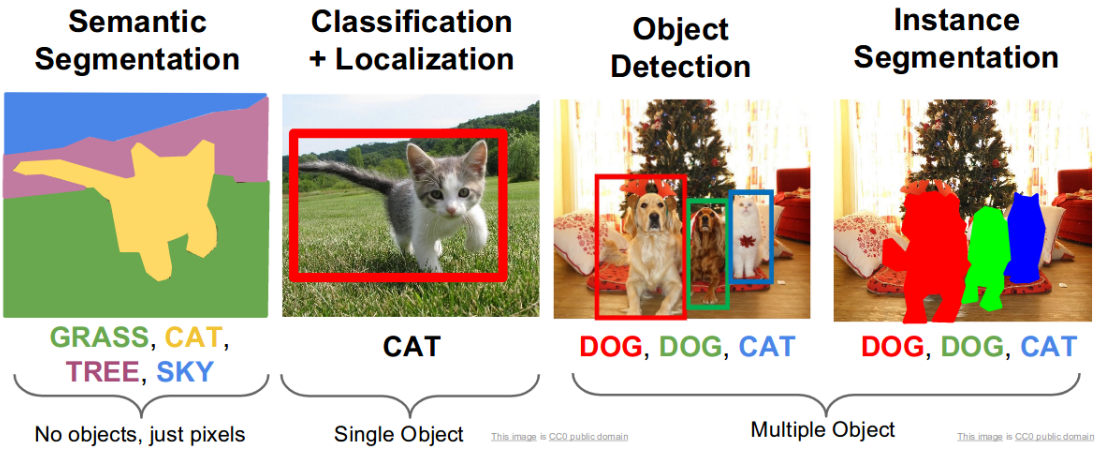


Figure 2.3: Different problem settings for finding objects in images. Image taken from `http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture12.pdf`.

RoI from each feature map and divides it into a $H \times W$ grid of roughly equal sizes and then pools over the grid, such that each grid cell is reduced to one value. This is done for all feature maps separately [10]. If the RoI is smaller than the input size, the RoIPooling Layer will upsample it.

2.1.3 Important Applications of CNNs

Classification describes the task of giving some input an object label. In multi-label classification, different items appearing in the same image should be recognized. That can be combined with finding the bounding boxes of these objects. Semantic segmentation describes the task of assigning a class label to each pixel of an input image. Figure 2.3 shows an example of different problem settings related to semantic segmentation and object detection. CNNs are state-of-the-art for these problem settings. The architectures we use in this thesis solve semantic segmentation as a sub-problem. The semantic segmentation determines the object class, but also helps in identifying which pixels are relevant for determining the rotation and translation.

2.2 Training Deep Neural Networks

Training Deep Neural Networks is a challenge because of the big amount of parameters and often long training times which makes tuning of the often numerous

hyperparameters infeasible. Additionally, regularization methods can reduce overfitting effects, that are induced by the large model capacities. Therefore we will shortly present some methods used for training and regularizing neural network models.

2.2.1 Stochastic Gradient Descent and Momentum

Gradient descent is used for training the network. It calculates the gradient of some loss function that describes the error for every parameter of the model and then updates the weights along this gradient, such that they change towards a smaller error. It is usually used with backpropagation, an efficient method for computing the gradient with regard to the single parameters. The learning rate determines how large the step along the gradient should be. It is a very important parameter with regard to the models performance [12]. Stochastic gradient descent (SGD) is an adaption from the gradient descent algorithm. While the classical gradient descent method uses the gradient of the loss calculated over all training examples to update the weights once, stochastic gradient descent approximates the whole training loss by taking the loss of a mini-batch of examples. Stochastic gradient descent is therefore suited to be used with large training sets. The term batch size refers to the number of training examples used for each approximation.

Momentum is a method used to speed up learning. It saves gradients from the past updating steps and adds a fraction of them to the current gradient. Therefore, the direction and strength of the former gradients is partially kept. This might be compared to a momentum of a moving body, which does not disappear when a new directional impulse is given. The influence of the former gradients decreases exponentially and a momentum parameter α can further scale this exponential decrease. So for a current gradient θ and a current velocity v, given learning rate ϵ and momentum parameter α, as well as new gradient estimate g, the velocity is updated by $v = \alpha v - \epsilon g$ and then the new gradient is $\theta = \theta + v$ [12].

2.2.2 Transfer Learning

Transfer learning aims to take learned knowledge from one dataset and use it for a different domain. For neural networks this knowledge can be interpreted as the weights which a particular architecture learned. A model can be trained first with the different dataset. This is called pre-training. Many frameworks offer models which are already trained on some dataset. If the architecture of the models is not the same, then the weights of some layers can be transferred to parts of the other architecture that are the same. In the fine-tuning step the network is then trained

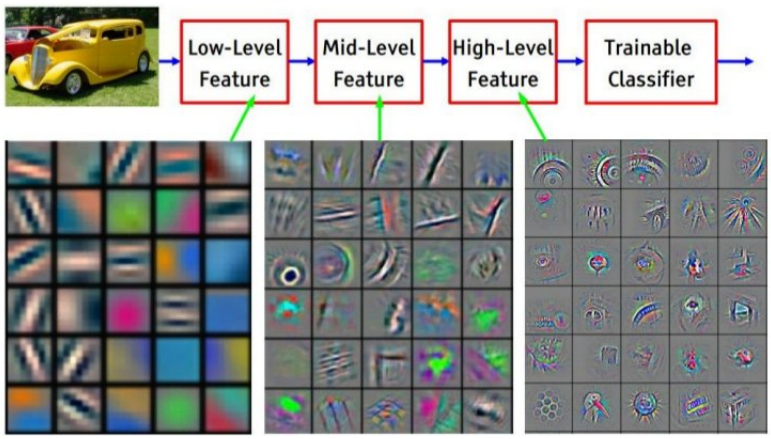


Figure 2.4: The visualized layers of a deep convolutional architecture. Image taken from Yann LeCun's tutorial `http://www.iro.umontreal.ca/~bengioy/talks/DL-Tutorial-NIPS2015.pdf`, based on Zeiler and Fergus [48]. The low-level features contain edge filters or recognize colours, which can be useful features for most images.

with the new data. Transfer Learning has been shown to reduce the generalization error [12].

Transfer learning is used as a method to intelligently initialize models, especially in cases, where there is not much training data available. For CNNs it is very easy to see, why this initialization might work, even when the application are different: When visualizing trained networks the early layers show that they learnt very easy features as for example edge or colour detectors [12]. Even if the domains are very different, the knowledge represented in this kernels would be useful for many applications dealing with photos. An example for that is shown in Figure 2.4.

2.2.3 Dropout

Dropout is a regularization method. It is used with mini-batch based training methods as SGD. In each training step units are effectively taken out of the network at random for forward pass, backpropagation and weight update by multiplying their output with 0. The probability that a neuron is present in any step is independent of the other units and other steps and is a fixed hyperparameter. The probability for including input or hidden units might differ though. Dropout simulates training not only the present network architecture, but an ensemble of all the architectures that one can obtain by removing units. During training not all the features, that are predicted by the other units, are available which encourages the network to learn more robust predictions, that do not rely on all the input

features. Dropout has shown good regularizing results at a low computational cost in cases with sufficient training data. [12]

2.3 Hough Voting

The Hough transform was originally described as a way to find lines in images. For this, points of the image vote on lines they might be part of. The voting is performed in Hough space, which spans the parameters describing possible lines.

For example to find lines the normal parameterization of lines can be used:

$$x \cos\theta + y \sin\theta = \rho. \tag{2.3}$$

Then the Hough space has as one axis θ and as the other ρ, and a line can be represented as one point in the Hough space. If θ is restricted to $[0, \pi]$, then the parameterization for a line is unique. Each point in the image votes in Hough space for all lines it can be part of. All the points that lie on one line should therefore give a vote to the point representing this line in Hough space.

To count the votes and determine the maximum, the Hough space is discretized, usually in a grid. The finer the quantization is, the more accurate the result can be. On the other hand it increases the computation time and if only few voting points are available, it might make finding the accurate maximum more difficult. If only one entity should be detected, then the bin with the most votes is selected. Multiple objects can be detected at the same time by selecting bins as a detection that are above a certain threshold. The Hough transform can be generalized to find other objects as curves or circles. [8]

We use the Hough transform to find the object centers. For this the Hough space is the pixel space and we do not need a reparametrization. Our use of the Hough transform is explained in detail in Section 4.2.

2.4 Camera Projections

Since we need some parts of this in the following, we will quickly define some terms related to how the 3D scene is transformed to the 2D image.

We assume a 3D scene that might contain objects. Initially the scene including the objects are given in world coordinates and somewhere in this scene there is the camera. Taking the camera as the origin of its own coordinate system, the scene can be transformed from world to the camera coordinate system by a translation T and rotation Ω. If the camera is assumed to be at the center,

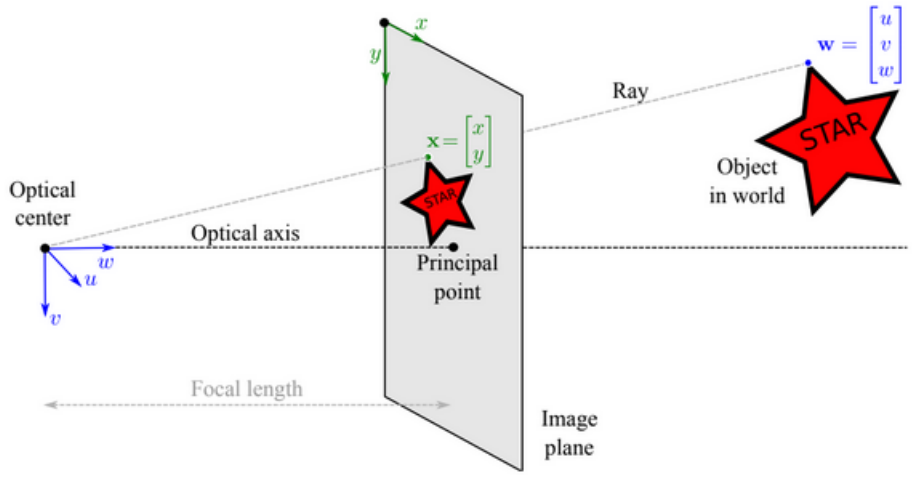


Figure 2.5: A depiction of the intrinsic camera parameters. Image taken from Prince [31].

then the world coordinates and camera coordinates are the same. The position and orientation of the camera are called the extrinsic camera parameters. Then the scene given in the camera coordinate system is projected onto the 2D image plane. This transformation depends on the assumed camera model as well as the intrinsic camera parameters: the focal lengths of the camera (f_x, f_y) and the principal point (c_x, c_y). In our case we assume the pinhole model and therefore a perspective transform is used. The whole transform of a 3D point (u, v, w) to the 2D image points can then be calculated in homogeneous coordinates as follows:

$$\lambda \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \omega_{11} & \omega_{12} & \omega_{13} & t_x \\ \omega_{21} & \omega_{22} & \omega_{23} & t_y \\ \omega_{31} & \omega_{32} & \omega_{33} & t_z \end{pmatrix} \begin{pmatrix} u & v & w & 1 \end{pmatrix} \tag{2.4}$$

The first matrix contains the intrinsic camera coordinates, the second the extrinsic camera coordinates. The first matrix projects the The variable λ arises from the conversion of homogeneous coordinates to Cartesian coordinates [31]. This is illustrated in Figure **??**.

2.5 RANSAC

RANdom SAmple Consesus (RANSAC) [9] is an approach to find a hypothesis given measurements. It randomly selects k measurements and fits a hypothesis that fits these measurements. For all measurements it is tested, whether they fit

the hypothesis or not. If they do, they are counted as inliers. This is repeated S times. Afterwards, the hypothesis with the highest amount of inliers is selected. This hypothesis can be used, but often the hypothesis is refined by fitting it to all the inliers [39].

2.6 Perspective-n-Point

Given a set of n 2D image and 3D world coordinate correspondences and the intrinsic camera matrix, the Perspective-n-Point (PnP) problem describes the problem of finding the pose consisting of translation and rotation . Classically the pose of the camera itself should be determined. For $n = 3$ (the P3P problem) there usually exist four possible solutions, therefore four points are in general sufficient for estimating the pose [23].

Different solvers for the PnP problem exist, that solve the problem if more correspondences than necessary are given. Another way of dealing with too many correspondences is to wrap a PnP solver into a RANSAC loop, which makes the procedure more robust to outliers.

2.7 Iterative Closest Point

The Iterative Closest Point algorithm (ICP) determines a transformation for two point clouds that transforms one of them, called reading, to match the other one, called reference, more closely. ICP then iteratively improves the matching from the initial pose of the reading. In each step it finds associations between the reading and reference and then calculates a transformation that minimizes the distance between the point clouds. The initial pose of the reading has a big influence on the overall performance of ICP. ICP might also do some preprocessing of the point clouds, since it is sensitive to outliers. It can also determine the number of iterations with a stopping criterion.

There exist many ICP variants, which use different methods to perform the individual steps [30].

2.8 Quaternions and Rotations

Quaternions are popular for representing 3D rotations. While theoretically one only needs three dimensions to represent any possible 3D rotation, there are instabilities representing them this way. A famous example is the gimbal lock when using the Euler rotation representation [42].

A quaternion, as defined by Hamilton, is expressed as the sum

$$q = q_w + q_x i + q_y j + q_z k$$

and

$$i^2 = j^2 = k^2 = ijk = -1 ij = k, jk = i, ki = jji = -k, kj = i, ik = -j$$

where $q_w, q_x, q_y, q_z \in \mathbb{R}$. So you can express the quaternion by the tuple (q_w, q_x, q_y, q_z) [42].

The addition of two quaternions q and p is defined as

$$p + q = (p_w + q_w) + (p_x + q_x)i + (p_y + q_y)j + (p_z + q_z)k. \tag{2.5}$$

The norm is defined as $|q| = \sqrt{q * q}$. A unit quaternion has norm 1 [42].

Any rotation in $\mathbb{R}^3$ around axis $u = (u_1, u_2, u_3)$ by θ degrees, can be expressed as a the unit quaternion

$$q = \cos\frac{\theta}{2} + u_1 \sin\frac{\theta}{2} + u_2 \sin\frac{\theta}{2} + u_3 \sin\frac{\theta}{2}. \tag{2.6}$$

In addition to avoiding the gimbal lock, representing rotations as quaternions is popular because the representation is continuous [39].

2.8.1 Quaternion Symmetry

Since we are not interested in the direction of the rotation, but in the final result, expressing rotations by axis and angle leads to two different ways in which you can express the same rotation and therefore also two different unit quaternions. For example rotating an object by 90 degrees around axis $(0, 0, 1)$ is equivalent to rotating it around the axis $(0, 0, -1)$ by 270 degrees. So the quaternions $(0.7071, 0, 0, 0.7071)$ and $(-0.7071, 0, 0, -0.7071)$ express the same final object orientation.

For two quaternions p and q, where $q = \cos\frac{\theta_q}{2} + u_x \sin\frac{\theta_q}{2} + u_y \sin\frac{\theta_q}{2} + u_z \sin\frac{\theta_q}{2}$ and $p = \cos\frac{\theta_p}{2} + u_x \sin\frac{\theta_p}{2} + u_y \sin\frac{\theta_p}{2} + u_z \sin\frac{\theta_p}{2}$,
if $-q = p$, then $\theta_p = 360° - \theta_q$ and $v_i = -u_i$ for $i = x, y, z$. Proof:

$$\theta_p = 2\arccos(p_w) = 2\arccos(-q_w) = 2(180° - \arccos(q_w)) = \\ 360° - 2\arccos(q_w) = 360° - \theta_w, \tag{2.7}$$

using $\arccos(-x) = 180° - \arccos(x)$.

And for $i = 1, 2, 3$:

$$v_i = \frac{p_i}{\sin(\frac{\theta_p}{2})} = \frac{-q_i}{\sin(\frac{360^\circ - \theta_q}{2})} = \frac{-q_i}{\sin(180^\circ - \frac{\theta_q}{2})} = \frac{-q_i}{\sin(90^\circ + 90^\circ - \frac{\theta_q}{2})}$$
$$= \frac{-q_i}{\cos(90^\circ - \frac{\theta_q}{2})} = \frac{-q_i}{\sin(\frac{\theta_q}{2})} = -u_i \quad (2.8)$$

Using $\sin(90^\circ + x) = \cos(x)$ and $\cos(90^\circ - x) = \sin(x)$.

This is exactly the case of the rotation axis pointing the opposite direction and the rotation around the other direction, mentioned above.

Apart from the $q, -q$ symmetry the rotation representation through unit quaternions is unique [39].

In order to unify the quaternion representation, we force the ground truth quaternions to all have a positive q_w, by multiplying the quaternion by -1 if q_w is negative.

Averaging by Markley et al. [27]

As Markley et al. [27] point out in their paper it is not so obvious how to average quaternions. One reason is the symmetry of q and $-q$ since replacing any q with $-q$ changes the average, though the same rotations are present. Also an average does not have to be a unit quaternion and then has to be normalized to be interpreted as a rotation.

They argue that what they want is an average of the rotations and that for this they can use the rotation matrices. So they define the quaternion $\bar{q}$ as the average that minimizes the squared Frobenius norm between its corresponding rotation matrix $A(q)$ and the corresponding rotation matrices $A(q_i)$ of all the quaternions that should be averaged:

$$\bar{q} = \arg\min_{q \in \mathbb{S}^3} \sum_{i=1}^{n} w_i ||A(q) - A(q_i)||_F^2.$$

w_i are weights and $\mathbb{S}^3$ is the unit 3-sphere. This average is then indifferent to whether q or $-q$ is averaged since their rotation matrices are the same. Also, by definition, $\bar{q}$ is a unit quaternion.

We will directly introduce the algorithm for the weighted case. The algorithm for determining the average quaternion, as layed out by Markley et al. [27] derivations,

is the following: For a set of quaternions $\{q_1, ...q_n\}$, $n \in \mathbb{N}$ and weights $\{w_1, ...w_n\}$:

$$M = \sum_{i=1}^{n} w_i q_i q_i^T \tag{2.9}$$

Then $\bar{q}$ is given by normalizing the eigenvector corresponding to the maximum eigenvalue of M.

3 Related Work

In this chapter we will first briefly describe some deep learning methods, that are used in the following, and then try to give an overview over the state-of-the-art in 6D pose estimation. While pose estimation is applied in many scenarios, for known or unknown, rigid or non-rigid objects, we will focus on giving an overview over the literature that is relevant for our task of developing a predictor that can handle clutter and symmetry for known and rigid objects.

3.1 Convolutional Neural Networks

Convolutional neural network architectures have shown great performance on several tasks related to image understanding: For classification, VGG16 and ResNet have produced state-of-the-art results for the ImageNet dataset [36]. YOLO [33] is a state-of-the art real-time detection system that also uses convolutions. It predicts the object class and bounding boxes for multiple objects in the image and can also handle multiple instances of the same class. R-CNN [11] predicts region proposals and the corresponding object label, it is also extendable to predicting the segmentation. Mask R-CNN [13] performs object instance segmentation.

3.2 6D Object Pose Estimation

6D object pose estimation was a problem that at some point was almost considered solved. However, the detection systems were only able to achieve good results for textured objects. The research focused on finding features that improved the robustness to changes in illumination. Later, more methods were developed that focused on objects with less or no texture.

With application in robotics becoming interesting that needed methods that are robust to occlusions and clutter in the scene, new datasets representing these challenges emerged: Datasets popular in the more recent works are the LINEMOD dataset [16] and OCCLUDED (Linemod) [20]. LINEMOD contains 15 objects, some of them textureless and symmetric. For each object there are around 1000 images, where the object was manually annotated. So only one object per image is

annotated. OCCLUDED is a subset of LINEMOD where the annotations for the other objects were added to 1214 images, so multiple objects have to be detected. All 1214 images were taken from the same video and the dataset contains only 8 different objects. Both datasets contain occlusions. Both datasets are small for training deep architectures. The YCB dataset [46] is a more recent dataset, that contains textureless and symmetric objects, clutter and very strong occlusions. There are multiple objects in each frame. In all datasets at most one object of each possible object class is present. YCB contains 133,827 pictures, so far more than the other datasets, making it more suitable for training deep neural networks. Since we work with the YCB dataset, we give a more detailed description in Section 6.1.1.

In the following we will present some works on object detection sorted by their general approach.

3.2.1 Sparse Feature-based Methods and Template Methods

In the past mostly feature-based or template based methods were used. The feature-based methods find features in the image that correspond to known object points and then calculate the pose. Usually only a few well-recognizable features are extracted for each object, therefore some also refer to them as sparse feature-based or keypoint-based models. Feature-based methods as for example by Lowe [26] and Wagner et al. [43] are popular and successful for determining the pose of highly-textured objects. A disadvantage of these methods is that they generally do not work well for non-textured objects.

Template-based methods try to match a rotated object to the image and decide the pose by the best fit. Two noteworthy template-based approaches were published by Hinterstoisser et al. [15] and Hinterstoisser et al. [16]. Though working well for non-textured objects, template-based approaches generally do not work well if part of the object is occluded.

3.2.2 Using Depth

With the development of cheaper depth sensors, methods using the depth modality became popular: Wong et al. [44] uses a CNN to predict a pixel-wise segmentation of the RGB input image and selects the depth map of the pixels belonging to some object. This point cloud is then matched to the object model using the iterative closest point algorithm (ICP). Hinterstoisser et al. [17] only uses the depth information to match pixel locations to object points. They extend the method of Drost et al. [7] to find object - image point pairs and use Hough voting

to determine the final pose.

Apart from methods for which the depth is a necessary input, all methods can be refined if depth data is available by performing ICP on the final prediction.

3.2.3 Deep Learning Approaches

With the rise of deep learning methods showing success for different problem settings, models inspired or extending these have been increasingly used. Some methods use established architectures to solve subproblems. For example Jafari et al. [18], Xiang et al. [46], and Rad and Lepetit [32] present methods that use a network for segmentation as part of their pipelines. Other works, as Do et al. [5], use instance segmentation. What separates these methods from the sparse feature based ones is, that they do not extract the features, but work on the images directly and learn some internal features. A lot of the recent methods can be divided into two groups: Methods that predict some sort of 2D image, 3D object coordinates correspondences and then solve the arising PnP problem for recovering the object pose, and methods that directly regress to some pose representation.

Predicting Pixel, Object Coordinate Correspondences

Brachmann et al. [2] uses random forests to jointly predict pixel wise 3D object coordinates and class labels for an RGB-D image. They sample pose hypotheses inside a RANSAC algorithm and evaluate the best pose using a energy function that is based on the reprojection error of the rendered pose hypothesis. Brachmann et al. [3] extend this work with a similar joint prediction of object coordinates and labels distributions. They improve the method by utilizing the uncertainty of the predictions. They introduce also a new energy function that does not need the depth information and therefore enables the method to work without depth information.

Jafari et al. [18] introduce a pipeline that divides the problem into subproblems: It uses an instance segmentation network to find the pixel belonging to each object, then an encoder-decoder network to densely map pixel to 3D object surface points. The instance segmentation is used to select the 3D object surface points that belong to each object. For RGB-D data the pose is then determined by sampling a fixed number of hypotheses, each based on three 3D points. The most promising scores are selected using a scoring function and then refined by using ICP. The pose with the lowest ICP fitting error is selected. Afterwards another refinement based on rendering is used. For RGB data the RANSAC method from Brachmann et al. [3] is used. Some qualitative results are shown in Fig. 3.1.

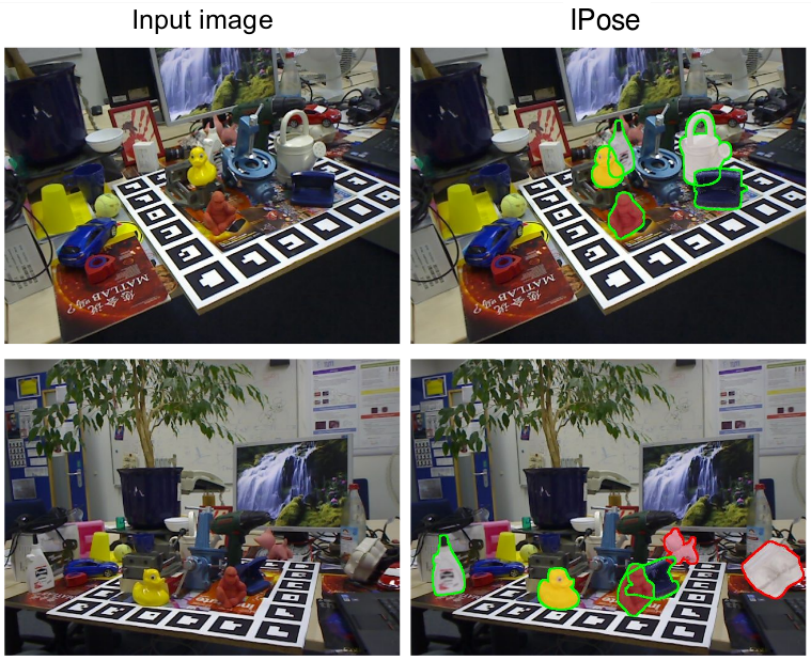


Figure 3.1: Results from IPose [19] on the OCCLUDED (Linemod) dataset.

Also there have been works, that instead of densely estimating the correspondences for all pixel, estimate the 2D points corresponding to the 3D object bounding boxes [41, 40].

Tremblay et al. [41] use exclusively synthetic data to train a VGG19 based fully convolutional architecture which predicts belief maps and the 2D image correspondences of the 3D bounding boxes. PnP is used for extracting translation and rotation. The belief nets are used to determine which objects are present.

Tekin, Sinha, and Fua [40] introduce a deep CNN architecture based on an improved version of YOLO [33], that directly predicts the 2D-3D bounding box correspondences. Translation and rotation are then recovered with PnP.

Rad and Lepetit [32] use a network architecture, called BB8, that extends VGG [38] to predict a segmentation. The centroid of an object segmentation is regarded as the 2D center of the object. Then another CNN, that is also based on VGG, takes crops centered on the 2D center of the original image to predict the 2D projections of the 3D bounding boxes and solves for the pose with PnP. The method handles symmetric objects by limiting the possible rotations to a fixed range during training. At run-time a separate classifier predicts the range the rotation is in. The prediction is followed by an optional refinement step that also uses only colour and no depth information. While BB8 achieves state-of-the-art results on different datasets, Jafari et al. [18] exclude it from their own comparison,

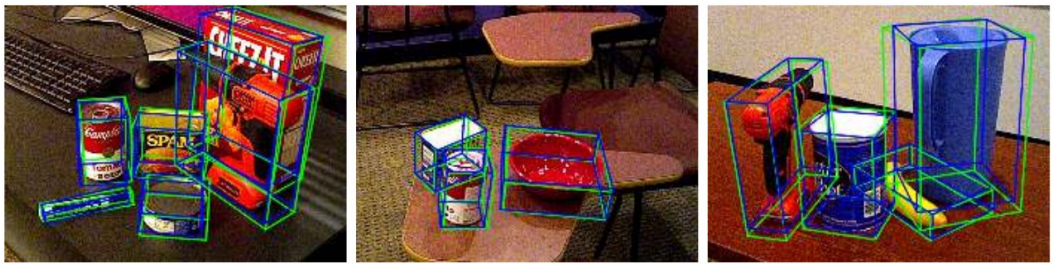


Figure 3.2: Qualitative results taken from Oberweger, Rad, and Lepetit [28] on the YCB dataset [46]. The ground truth 3D bounding box is drawn in green, the prediction in blue.

because its results rely on ground truth crops.

Oberweger, Rad, and Lepetit [28] also predict the projection of the 3D bounding box. They try to tackle occlusion by using small object patches for independent predictions. These predict a heatmap of possible corner locations based on that patch. Then the heatmap from the patches are added and the maximum is selected as the corner position. If patches are ambiguous, they are trained to also show that ambiguity in their prediction. If a patch cannot give any valuable information, for example because it is occluded, the heatmap is trained to be uniform. Some qualitative results are shown in Fig. 3.2.

Direct Pose Regression

Another group of methods uses either fully connected or convolutional networks for regressing directly to some representation of the pose. For many of the methods only parts of the pipelines are trainable, but there also has been work aimed at developing end-to-end trainable methods. While the methods predicting object coordinates predict translation and rotation jointly, directly regressing methods separate the estimation of translation and rotation.

Do et al. [5] use Mask R-CNN [13] for instance segmentation and add a pose estimation branch to it. They decouple the estimation of translation and rotation. For rotation they estimate Lie parameters. No post refinement is needed and the architecture is end to end trainable.

Xiang et al. [46] predict the pose using the convolutional features given by VGG16 [38]. These features are used to estimate a segmentation and pixel-wise translation. The 2D object center is then inferred in a Hough voting and a simultaneous prediction of RoIs is used to crop the VGG16 features and regress the rotation with a fully-connected architecture. They introduce Shapeloss, a loss function that avoids penalizing symmetric objects. Our work builds upon

PoseCNN, therefore there will be a more thorough description in Chapter 4.

A recent method that uses the same feature extraction set up as PoseCNN is Silhonet by Billings and Johnson-Roberson [1]. It uses an RCNN [11] to predict Region of Interests (RoIs) and then crops the features to that regions. These regions are concatenated with additional 12 rendered viewpoints of the recognized object. From this the unoccluded silhouette of the object is predicted. Finally a ResNet-18 [14] architecture predicts quaternions from the unoccluded silhouette representation. For the training, object silhouettes are matched to quaternions thus eliminating the problem of shape symmetry. As PoseCNN it does not require depth data. It shows better performance than PoseCNN for estimating the rotation, however, the translation is not estimated and therefore not the same measures are applied, which makes comparison difficult. Also the texture of objects is not considered, so objects that are symmetric in shape but not in texture cannot be oriented due to their texture.

3.2.4 Refining the Result

Apart from predicting the pose from the RGB or RGB-D data, there are several refinement techniques for improving the poses after determining an initial pose. For some of the methods this refinement step is based on the prediction and therefore related to the method. Others, such as ICP, can be used with any initial estimation.

Li et al. [24] introduce a refinement technique that improves the estimation only using the original RGB input. It takes an image and an initial pose estimate and then iteratively improves the pose by zooming in on objects and matching a rendered image against the original image. For this it generates a foreground mask for both original image and rendered image and feeds them with the two masks into a FlowNetSimple [6] architecture. It uses the Point Matching Loss, a modified version of the geometric reprojection loss to jointly regress translation and rotation. The authors show improvement on the initial estimate of both PoseCNN [46] and Faster RCNN [34].

3.2.5 Our Approach

Our own work incorporates some of the previously listed related work: As many of the newer methods we will make our predictions only using the RGB image. Also we do not use any post-refinement. Our models are fully-convolutional and we produce a dense prediction. From the presented work only few have a strategy to handle symmetric object poses. We will experiment using the Shapeloss for our

models. While many of the works use semantic segmentation and a few use pixel-wise object coordinate predictions, none of the works we saw has tried a pixel-wise rotation estimation.

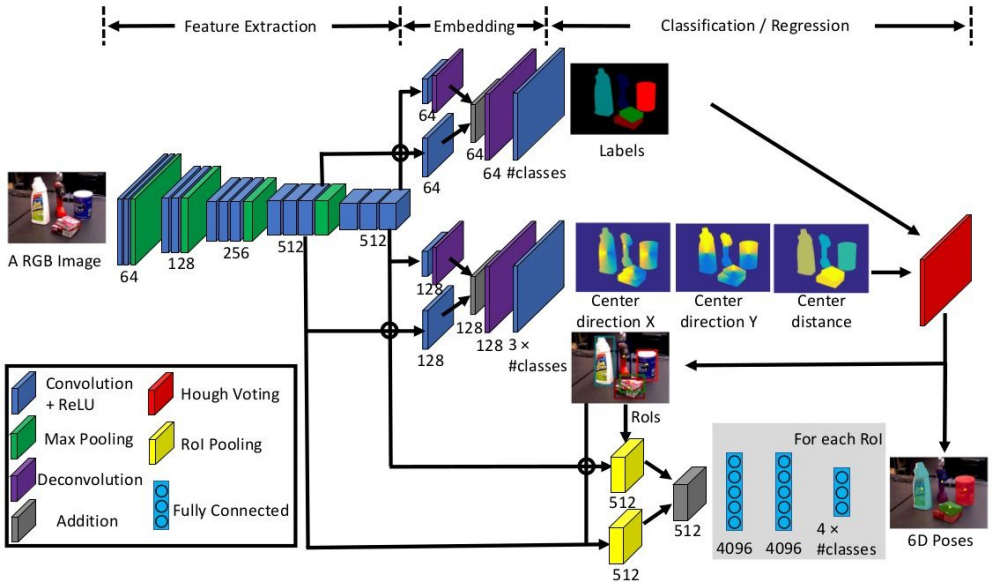


Figure 4.1: The architecture of PoseCNN. Graphic taken from [46].

4 PoseCNN

In this chapter we will describe the 6D Pose Estimation architecture PoseCNN, as introduced by Xiang et al. [46], which is the base for our own architectures and our most important baseline for evaluation. Figure 4.1 shows an overview of its architecture. There one can see the main parts of PoseCNN: It uses the convolutional backbone of VGG16 [38] to extract features. These features are then processed by two convolutional parts: The segmentation branch, that predicts the pixel-wise semantic segmentation, and the vertex branch, which predicts a pixel-wise estimation of the center direction and center depth. Both results are combined to vote for object centers in a Hough transform layer. The Hough layer also predicts bounding boxes for the detected objects. These bounding boxes are used to crop the extracted features which are then pooled and fed into a fully-connected neural network architecture. This fully-connected part predicts quaternions for all bounding boxes.

Input to the network are RGB images. The network makes its prediction without

using any depth information. Only afterwards the depth information is used to refine the pose with ICP.

In the following we will introduce the different parts of the network more closely. We will present them as they were presented in the original paper. However, we have found some differences to their published code, which we will discuss in Section 6.3.

4.1 Segmentation

The semantic branch of PoseCNN is inspired by the architecture of Long, Shelhamer, and Darrell [25]. The features of two different parts of the VGG backbone are passed through convolutions and added together. Transposed convolutions are used to upsample the smaller feature maps to the same size and again to transform the added feature maps to the original image size. This way the prediction has a direct relation to each pixel of the input image.

4.2 3D Translation Estimation

The second branch of the network, called vertex branch, is used to estimate the 3D translation in the camera coordinate system: $T = (T_x, T_y, T_z)$. For each pixel of the image it predicts the direction towards its corresponding object center (c_x, c_y), which is the projection of T onto the image. Each pixel also estimates T_z. It uses an architecture similar to the semantic branch, but increases the number of feature maps, since it is not only predicting (#classes· image size) but (3·#classes· image size) values. Since the network estimates the directions and depth for each possible object class, the labels from the semantic branch are needed to determine which information is relevant.

The Hough voting layer then determines where the center (c_x, c_y) is: It operates separately for every class. For each pixel p that belongs to this class, all pixel that lay in the predicted center direction of p receive a vote in the Hough space. So for each pixel of the class one line is added to the Hough space. Where there intersect many lines, the voting space reaches a local maximum. Then either the maximum is selected as the estimated center or non-maximum suppression is applied to find multiple object centers. The voting space has the same size as the image, so centers can only be found inside the image.

The Hough layer determines inlier for each center, which are pixels whose estimated direction is close to the selected center. For this the cosine similarity between the direction n that is estimated at point p and the direction from p to

the determined center x is used:

$$s(p, x) = \frac{\langle n, (x - p) \rangle}{||n|| \cdot ||(x - p)||}.$$

If this similarity s is bigger than a chosen threshold s^*, then the point p is considered an inlier. For each center the depth is determined by taking the inliers associated to one center and averaging the depths they predict.

Now with an estimate for the 2D image center (c_x, c_y) and the depth T_z, one can determine T_x and T_y using the projection equation, based on the pinhole camera model.

$$\begin{pmatrix} c_x \\ c_y \end{pmatrix} = \begin{pmatrix} f_x \frac{T_x}{T_z} + p_x \\ f_y \frac{T_y}{T_z} + p_y \end{pmatrix}$$

and therefore

$$\begin{pmatrix} T_x \\ T_y \end{pmatrix} = \begin{pmatrix} (c_x - p_x) \frac{T_z}{f_x} \\ (c_y - p_y) \frac{T_z}{f_y} \end{pmatrix}$$

where f_x and f_y are the focal lengths of the camera and (p_x, p_y) is the principal point.

The network regresses to the normalized center directions (n_x, n_y):

$$\left(n_x = \frac{c_x - x}{||c - p||}, n_y = \frac{c_y - y}{||c - p||} \right)$$

for a pixel $p = (x, y)^T$ and center $c = (c_x, c_y)^T$. Xiang et al. [46] state that they experimentally verified that this is easier to train than regressing to the unnormalized variant.

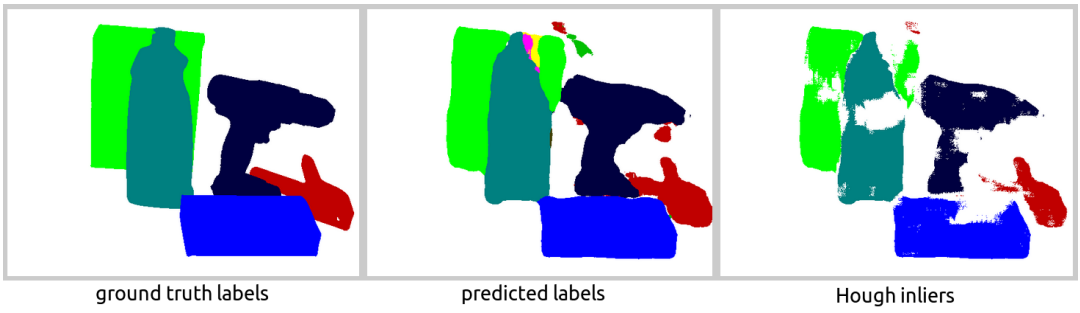


Figure 4.2: Images from our implementation of the Hough voting. The left picture shows the ground truth segmentation, the middle shows the predicted segmentation and the right picture the Hough inliers.

The Hough layer additionally predicts Regions of Interests (RoIs), which are bounding boxes containing part of the image where the wanted object is present. For this the smallest rectangle that includes the inliers of each object is used.

Figure 4.2 shows an example from our implementation of the inliers returned from the Hough layer. The Hough layer removes small patches of wrong prediction, as the pink and yellow one. Since the cosine similarity is used for the selection, pixels that are far away have a higher tolerance on how much their prediction can differ from the center direction, leading to the holes in the middle of the objects.

4.3 Rotation Estimation

The rotation branch consists of a fully-connected architecture. The RoIs predicted by the translation estimation part are used to crop and pool the features given from VGG16. Like the other branches it uses features from two different feature extraction branch layers. These are passed through two different RoI pooling layers and then added. The result is then put through three fully-connected layers (FCs). For each of these RoIs and each possible class a quaternion is predicted. With the information from the Hough layer, the correct quaternion can then be selected. The published code shows that the first two FCs are followed by ReLU activations and the final layer is followed by a tanh activation, however, both the direct output and the tanh activation are returned. We got the best results, when training with tanh but using the quaternions before the nonlinearity for testing.

4.4 Training Losses

The network learns the segmentation, vertex prediction and rotation somewhat separately and uses different losses for training the different parts. The Smooth L_1 Loss is defined as follows:

$$loss(x, y) = \frac{1}{n} \sum_i z_i \tag{4.1}$$

where

$$z_i = \begin{cases} 0.5(x_i - y_i)^2, & \text{if} |x_i - y_i| < 1 \\ |x_i - y_i| - 0.5, & \text{otherwise} \end{cases}$$

It is less sensitive to outliers than using a L_2 loss [34]. For training the semantic branch a softmax cross entropy loss is used.

For training the rotation Xiang et al. [46] use the ShapeMatch Loss. This loss

consists of two losses: The point-wise loss (Ploss) and the shape loss (Sloss). The ShapeMatch loss takes as input whether an object is symmetric or not and returns the Sloss for a symmetric object and the Ploss if not. Given a set of 3D points $\mathbb{M}$, where m = $|\mathbb{M}|$ and $R(q)$ and $R(\tilde{q})$ are the rotation matrices from the ground truth quaternion and the estimated quaternion, the Ploss and Sloss are defined as follows:

$$\text{Ploss}(\tilde{q}, q) = \frac{1}{2m} \sum_{x \in \mathbb{M}} ||R(\tilde{q})x - R(q)x||^2, \tag{4.2}$$

$$\text{Sloss}(\tilde{q}, q) = \frac{1}{2m} \sum_{x_1 \in \mathbb{M}} \min_{x_2 \in \mathbb{M}} ||R(\tilde{q})x_1 - R(q)x_2||^2, \tag{4.3}$$

Similar to ICP, it uses the distance between any point of the 3D model in the estimated orientation and the closest point on the 3D model of the ground truth orientation. That means that the Sloss does not penalize a rotation for a symmetric object that is not the ground truth rotation but leads to an equivalent shape.

4.5 Evaluation Metrics

For the final evaluation Xiang et al. [46] use the average distance metric, ADD, and the average distance metric using the closest points, ADD-S, as stated by Xiang et al. [46]:

$$\text{ADD} = \frac{1}{m} \sum_{x \in \mathbb{M}} ||(Rx + T) - (\tilde{R}x + \tilde{T})||, \tag{4.4}$$

$$\text{ADD-S} = \frac{1}{m} \sum_{x_1 \in \mathbb{M}} \min_{x_2 \in \mathbb{M}} ||(Rx_1 + T) - (\tilde{R}x_2 + \tilde{T})||, \tag{4.5}$$

where $\tilde{R}$ and $\tilde{T}$ are the estimated rotation and translation.

For the final evaluation the Area Under the Curve (AUC) for these losses is considered. Here they consider the curve which shows for a range of average distance threshold (ADD) values, how many percent of the objects fall below that threshold. They cut off the maximum error at 0.1. Also unrecognized objects are penetalized with an ADD of 0.1. An example is shown in Figure 4.3. We call AUC P the area under the curve of ADD and AUC S the area under the curve for ADD-S. An example of some of the relevant loss functions is shown in Figure 4.4.

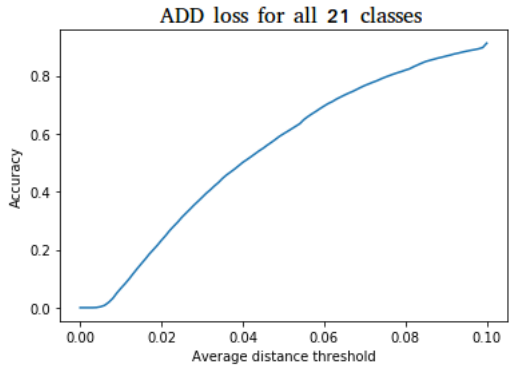


Figure 4.3: An example for the curve evaluating the ADD metric, with which AUC P is calculated.

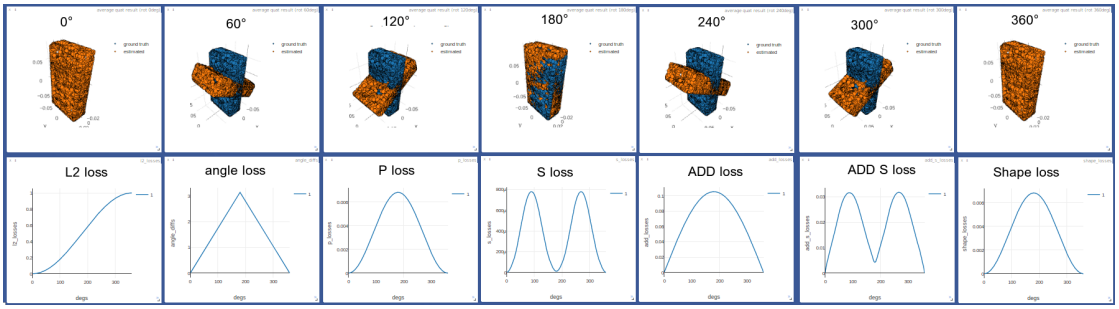


Figure 4.4: Different losses shown for the example rotation of a sugar box. The blue box stays, while the orange one is rotated by 360 degrees around one axis. The L2, angle, P, S, ADD, ADD-S and Shape loss are shown for this rotation.

4.6 Training of PoseCNN

The training of PoseCNN starts with a pretraining. The weights of a VGG16 network trained on ImageNet are transferred to the convolutional layers of the feature extraction part and the fully connected layers. The weights are optimized using Stochastic Gradient Descent with momentum (0.9) and a learning rate of 0.001. First, the segmentation and translation parts are trained with the cross entropy and the Smooth L1 losses as described above. The losses are calculated directly on the network prediction before it enters the Hough layer. The network training then optimizes the loss $L = L_{seg} + L_{trans}$. Then also the rotation is trained. For this the Shape loss is calculated and added to the other losses. The training then optimizes $L = L_{seg} + L_{trans} + L_{rot}$. While the rotation estimation uses the RoIs predicted from the Hough layer, no gradient is backpropagated through the Hough layer.

5 Fully Convolutional Architectures

Our goal was to replace the RoI pooling and fully-connected parts for the rotation estimation of PoseCNN with a fully convolutional design. For this we investigated two different approaches: The first, as PoseCNN, estimates quaternions. The second one estimates 2D pixel, 3D object coordinate correspondences. Both these architectures estimate this information pixel-wise.

5.1 ConvPoseCNN

As a possible improvement we implemented a network that predicts the quaternions pixel-wise. For this we use the architecture shown in Figure5.1, which we

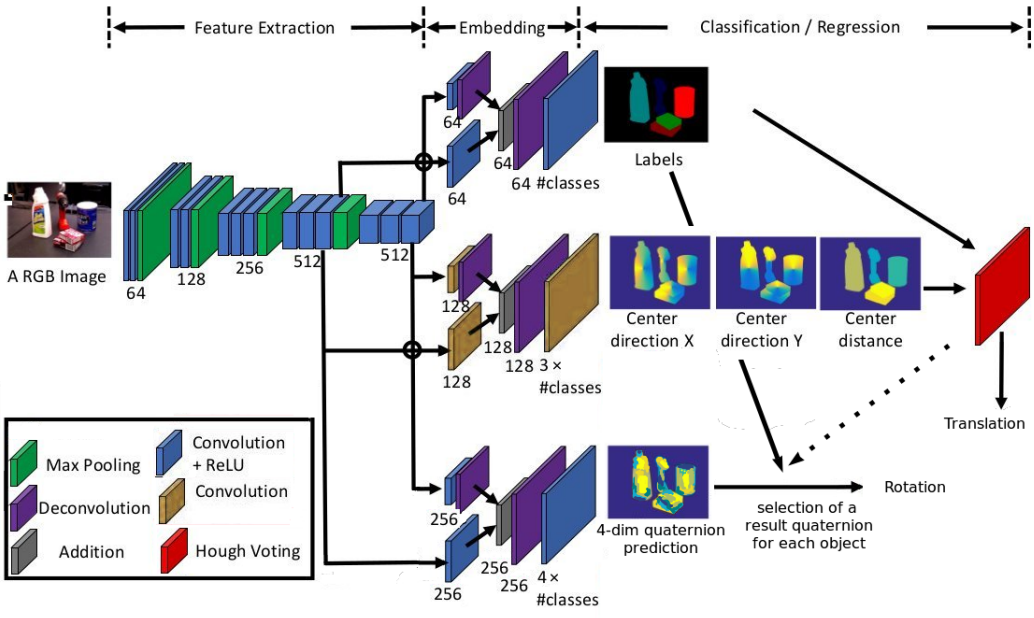


Figure 5.1: Our ConvPoseCNN architecture for estimating the quaternions pixel-wise. Image adapted from [46]. The selection of the quaternions for each object is done using the segmentation labels. However, for the case of detecting multiple instances of the same object class, the Hough inliers can be used.

will call ConvPoseCNN (short for convolutional PoseCNN). It uses PoseCNN as a base, but the former RoI pooling and fully connected layers of the PoseCNN rotation estimation branch are replaced by an architecture similar to the segmentation and translation estimation branch. The segmentation estimation branch has 64 feature maps to estimate one value, which was increased in the design of PoseCNN to 128 for the vertex branch, since it estimates three values. Since we are predicting four values for each pixel, we increase the number of feature maps to 256.

5.1.1 Training ConvPoseCNN

For training ConvPoseCNN we generally follow the same approach as for PoseCNN: We use SGD with momentum on the combined loss $L = L_{\text{seg}} + L_{\text{trans}} + L_{\text{rot}}$. We calculate the rotation loss pixel-wise using the ground truth segmentation. For a pixel-wise loss, the proposed Shape loss from PoseCNN is too slow. We use either the L2 loss or the Qloss. For two quaternions $\bar{q}$ and q the L2 loss is defined as

$$\text{L2}(\bar{q}, q) = \sum_{i=1}^{4} (\bar{q}_i - q_i)^2. \tag{5.1}$$

The Qloss was introduced by Billings and Johnson-Roberson [1] and is designed to handle the quaternion symmetry. For two quaternions $\bar{q}$ and q it is defined as

$$\text{Qloss}(\bar{q}, q) = \log(\epsilon + 1 - |\bar{q} \cdot q|). \tag{5.2}$$

ϵ is a small constant used for stability.

5.1.2 Averaging or Clustering Quaternions

Since we estimate quaternions pixel-wise, we obtain many quaternions for each object in the image and need to find a way to determine a final quaternion.

First we need to identify which quaternions belong to which object. For this we can use the predicted segmentation. If multiple instances of one object can occur one could use the Hough inliers. Then we need to determine a result quaternion for each object based on the quaternions belonging to one object:

First it is possible to calculate a simple average of quaternions $\bar{q} = \frac{1}{n} \sum_{i=1}^{n} q_i$. As we explained in Section 2.8.1 it is not an accurate average of the quaternions though. For this Markley's average is better suited.

Additionally, when considering the shape of some objects, we expect that some parts of the object are more important for determining the correct rotation than

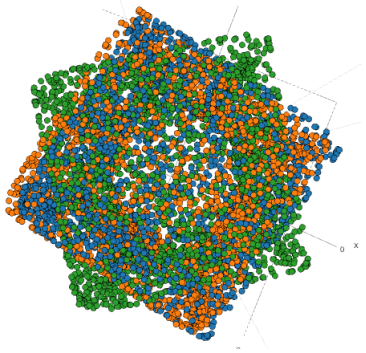


Figure 5.2: Example of rotating an objects point cloud by three quaternions: One that expresses no rotation (blue points), one that rotates the points by 90° (orange points) and their average (green points). While both quaternions represent a similar shape for the symmetric object, the averaged quaternion would not represent that shape.

others. For example looking at a non-textured cup, most of the cup could be turned in either direction, but the handle shows the additional information that at least a human needs to determine the rotation. Therefore, we will also experiment with weighting the quaternions by a confidence. For this we try different weighting sources. One of them is the norm of the quaternions before they get normalized. Another possibility is using the segmentation confidence.

Also depending on the object it might not be a good idea to simply average the quaternions. For example if we have an object that is symmetric by 180°, then it is quite possible that the network would predict quaternions that correspond to both likely rotations. If you average these quaternions though, you get a result somewhere in between, as you can see in the example in Figure 5.2. Because of this we also investigate methods that will try to cluster the quaternions, such that, ideally, it results in an average of one potential rotation and not an average between multiple valid rotations.

One method for finding cluster is using a RANSAC algorithm. This algorithm repeatedly chooses a random quaternion $\bar{q}$ from all quaternions $\{q_1, ..., q_n\}$ and then counts quaternions that are close by as inliers I. As a measure of closeness different metrics d can be used. Also the distance threshold t and the number of repetitions r are parameters that need to be chosen. A quaternion quaternion q_i is then an inlier if $d(q_i, \bar{q}) < t$. The quaternion that has the highest number of inliers, $\sum_{q_i \in I} 1$, is considered the cluster center.

We also use a modified version of this approach that considers weights. For this we count the inliers with their weight instead of simply by number. So for quaternions $\{q_1, ..., q_n\}$ and weights $\{w_1, ..., w_n\}$ and inliers I, the $\bar{q}$ with the highest

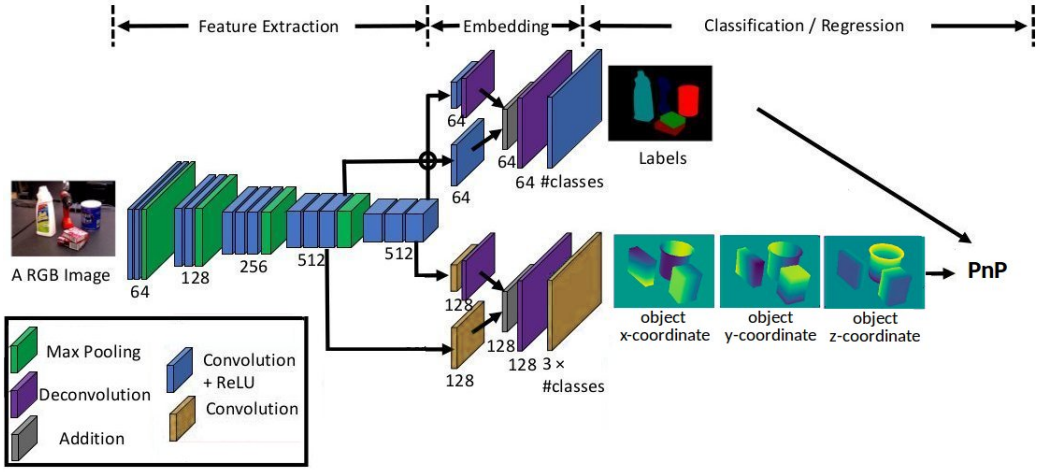


Figure 5.3: The architecture of CoordPoseCNN. The direction prediction is replaced by estimating the object coordinates and the rotation branch is dropped. Image adapted from [46].

sum $\sum_{q_i \in I} w_i$ is selected. Additionally, the initial quaternion $\bar{q}$ in each iteration is not chosen uniformly at random, but with a probability proportional to its weight.

5.1.3 Training with an Average Quaternion

It is also possible to first average the predicted quaternions for an object and then calculate the loss on the average. We then train with this loss. This makes it possible to use for example the Shape loss. We used the simple and weighted simple average for this, since the PyTorch autograd engine is not able to compute the gradient of Markley's average.

5.2 CoordPoseCNN

We also tried another fully convolutional architecture derived from PoseCNN that aims to estimate the 3D object coordinate - 2D pixel correspondences for each pixel of the input image. The architecture is shown in Figure 5.3. Also Xiang et al. [46] shortly mention such experiments in their paper, with the same architecture, which they used as a base-line. However, they do not describe these experiments in detail and refer to Brachmann et al. [3] on how they recovered the poses from the correspondences. Brachmann et al. [3] also predict object coordinates for each pixel and a probability for belonging to each possible class. They sample pixels and their class according to the predicted distributions for all classes and

then perform RANSAC for each object class separately. Their RANSAC approach takes four pixels belonging to the same class and calculates inliers by solving the PnP problem and calculating the reprojection error.

We use PnP for recovering the final pose for each object separately by selecting the relevant pixel with the predicted labels. We investigated different PnP solvers offered by OpenCV [4]. One method for solving the PnP problem is the EPnP approach, as proposed by [23]. It is a non-iterative approach that runs in $O(n)$ for all $n \geq 4$. The approach reduces the n 3D points to four so called virtual control points and then estimates their 2D image correspondences.

OpenCV also offers an approach called ITERATIVE, that works by minimizing the sum of squared distances of the reprojection error. For this it uses the Levenberg-Marquardt optimization [35]. Both algorithms can be wrapped into a RANSAC loop.

5.3 Training CoordPoseCNN

For training we only have two losses, the segmentation loss and the coordinate loss, so we minimized the joint loss $L = L_{\text{seg}} + L_{\text{coor}}$. For the segmentation we used again the Cross Entropy loss and for the coordinates the SmoothL1 loss, defined in Section 5.1.1. We used the same optimizer and learning rate as PoseCNN.

6 Experiments

In this chapter we will describe our experiments first on recreating the PoseCNN architecture and then comparing it to our own architectures. Additionally, we provide insights into the models training process and design choices and report details on the used datasets.

6.1 Datasets

Both PoseCNN and our own architectures use a VGG16 architecture pretrained on the ImageNet dataset. ImageNet [36] contains around 21 million images for image classification from 1000 different object classes. Our models use a VGG16 pretrained on ImageNet.

SUN2012 and ObjectNet3D are used as backgrounds for the synthetic data SUN2012 [47] contains 16,873 images of different scenes containing a wide range of objects. The dataset was designed as an object detection benchmark. It is part of a bigger scene recognition benchmark. Therefore the dataset has a lot of variety. There are indoor and outdoor scenes, rigid and non-rigid objects and small and big objects, as for example mountains.

ObjectNet3D [45] contains 90,127 images covering 100 different scene categories. It is a database that was made for object category detection of rigid objects. It contains indoor and outdoor scenes. ObjectNet3D also contains some objects that are similar, to the objects in the YCB dataset, for example scissors and pens.

6.1.1 The YCB Dataset

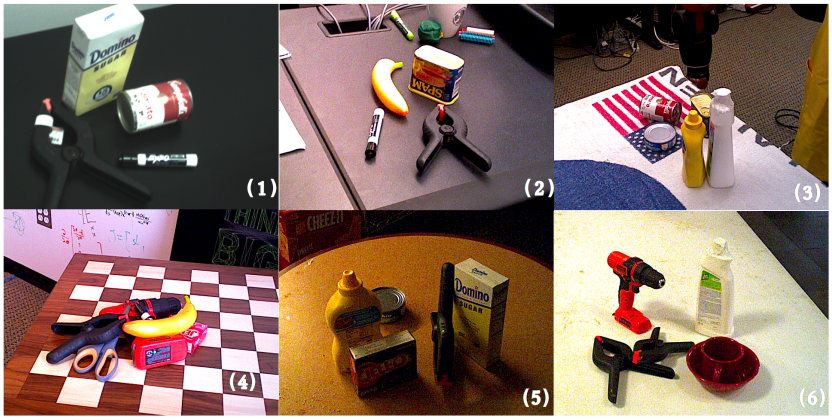
The YCB dataset, created by Xiang et al. [46], contains RGB and depth images of 21 objects. There are 133,936 images extracted from 92 videos, which are separated into 113,198 training images from 80 videos and 20738 validation images from 12 separate videos. Among the validation videos there are 2950 keyframe images, that are used as the test set. The objects include symmetric objects and objects with strong texture or almost unicoloured surfaces. The objects of the dataset are shown in Figure 6.1 and a sample of images is shown in Figure 6.2.

Figure 6.1: The 21 objects of the YCB dataset. Image taken from [46].

The data is not entirely hand annotated: Only the first frame was annotated and the rest of the frames inferred by estimating and using the camera trajectory throughout the video. Although the poses are again refined, the annotations of the data are therefore not perfect. The images contain among their meta-data the objects that are present in the images, their poses and 2D centers. However, there are images, where objects are completely or almost completely occluded. Our experiments checking the training set showed that around 0.6 percent of the objects listed as being present in the images are completely occluded. Additionally there are around 0.03 percent images which are present in less than 25 pixels.

In some images the 2D center of the object lies outside the image, so the Hough layer of the PoseCNN architecture can not detect the correct center. We also checked the distribution of classes in the training set. It is shown in Figure 6.3. The classes are more or less equally distributed, only the bowl appears notably less often. Also the full or strong occlusions are quite equally distributed, so we do not expect any classes to be underrepresented in the training set. Each image contains on average 4.4 objects.

Additionally to the frames extracted from the videos there are around 80.000 synthetic images containing the same objects. These synthetic images are not physically realistic and contain hovering objects. The objects appear more or less equally often. However there are a lot of completely or very occluded objects:



(a) training set images

(b) synthetic images

(c) keyframe images

Figure 6.2: Images from the YCB dataset [46]. The training set shows varying lighting conditions, distracting backgrounds, clutter, unfocused images and severe occlusions. The test set (keyframes) also shows challenging scenes.

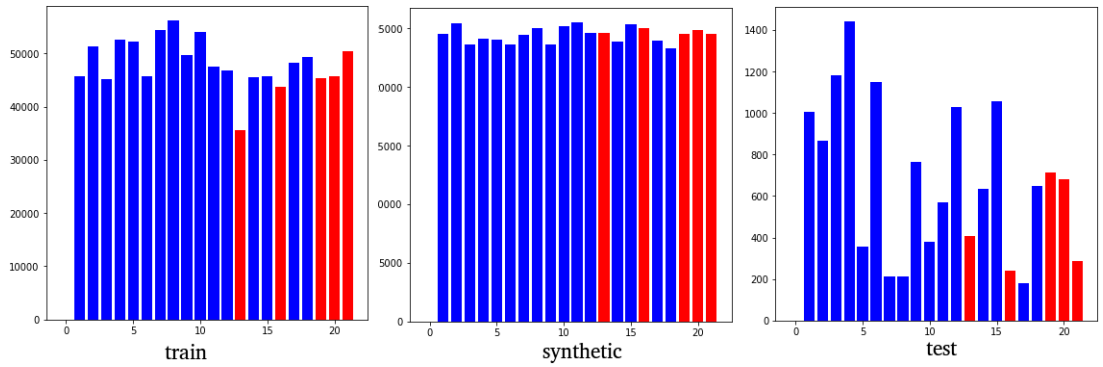


Figure 6.3: The number of appearances of each object class in the training, synthetic and test set. For all sets the number of actual appearances is counted, so the complete occlusions, that are still listed in the ground truth data, are not counted. The symmetric classes are coloured red. The class appearing the least in the training is class 13, the bowl. The other classes are in the same order as in the following evaluations e.g. in Table 6.1.

Around 3.8 percent of objects are completely occluded and around 4.1 percent are smaller than 25 pixels. The synthetic set contains on average 6.4 objects per image.

The images shown in Figure 6.2 are rendered onto images from the SUN2012 [47] and ObjectNet3D [45] backgrounds. The synthetic images saved in the YCB dataset are without background though. We explain this further in Section 6.3, where we discuss the differences we found in the published code, which were not mentioned in the paper.

For the test set the hand-annotated keyframes are used. Therefore, the data has no completely occluded objects. However, occlusions can still be very strong as can be seen in Figure 6.2 (c2): Behind the mustard bottle, there is a soup can, that is almost not visible.

For each object the dataset contains a point model with 2620 points each, and a mesh file. The dataset also contains label images, poses and two-dimensional images of the object, where the three channels correspond to the 3D object coordinate, that can be seen at that pixel.

The images contain multiple relevant objects in the same images, as well as sometimes uninteresting objects and distracting background. Notably each object appears at most once in each image. Therefore, this dataset is not able to test the performance for the case of multiple instances of the same class.

A particular challenge we found in this dataset are the two clamp objects, that appear to almost only differ in size. Also their reference rotation in the dataset is

different. Therefore confusing the clamps will lead to a wrong rotation estimation. In Figure 6.2, in images A6 and C4, both clamps can be seen side by side. Since the architectures work without depth information, that would help determine the scale of the clamps, the context is therefore important for determining the correct clamp and the correct depth of the clamps.

Evaluating on YCB

Since we want to compare to PoseCNN, we use the same set up for comparison and also tried their code for evaluation as described in Section 6.2. However, the number of times each object appears in the test set is quite unbalanced as Figure 6.3 shows. Therefore, the final average over the AUC losses is far more dependent on the performance of certain objects. Furthermore, the two different losses are of different relevance for different objects: Deciding which loss is important for which object is, of course, dependent on the use-case. When the task is to simply grab an object, then only the shape matters. Symmetric objects with texture (as for example the soup can or the tuna can) are then also symmetric. On the other hand there might be use cases where it is important which way the soup can is facing. For our evaluation we consider the losses with our expectations of what the network should learn: For the average of the AUC S we consider only the untextured, symmetric objects, which are the same that the PoseCNN paper labeled as symmetric. For the average of the AUC P we use all other objects. This is the same division as was used for PoseCNN to train the symmetric objects with the Sloss and others with the Ploss. The average is taken over the class averages, so the number of occurrences in the test set is irrelevant. This way of averaging the AUC Losses made it easier for us to see, whether our models were actually improving. In the following evaluations we call it SymC (short for symmetry dependent and class-wise). Also Oberweger, Rad, and Lepetit [28] use a similar final score to compare to PoseCNN. They report only the AUC S for the symmetric objects and only the AUC P for the non-symmetric objects. They then take only one average over all the classes. In the following we will report our results using the following abbreviations: [46] total, for the values reported in the PoseCNN paper, [28] average, for their reported average and SymC. We will also report values for which we consider only the rotation estimation (so the translation is perfect). These averages are then called [28] ROT and SymC ROT.

Another thing we had to consider is the detection sensitivity. The way Xiang et al. [46] evaluate, there is no penalty for detecting objects that do not appear in the image. The Hough layer implemented by Xiang et al. [46] has a detection threshold that only returns a detection if the maximum vote is above a certain

threshold. We do the same for all experiments.

6.2 Implementation

We implemented our experiments using PyTorch [29], a deep learning library for Python.

PyTorch has an automatic differentiation module that computes the gradients for most functions and operations that PyTorch offers.

Getting a quick implementation for the Hough transform was crucial especially for using it in the training of PoseCNN. We used Numba [21], a compiler for Python code, that can speed up NumPy code. This significantly outperformed our approaches to implement it in PyTorch. Since no gradient is backpropagated through the Hough layer, we could use NumPy and Numba instead of PyTorch.

Our models ran on different NVIDIA GPUs, which had 12GB memory. We report time measurements on the fastest one in Section 6.7.

For the pretrained VGG16 weights we used the weights provided by PyTorch.

We also used the evaluation framework published in the YCB toolbox[1], which was published by Xiang et al. [46] to verify our own implementation of the evaluation. This is discussed in the next Section.

6.3 Re-implementing PoseCNN

The PoseCNN paper reports the results for their model on the YCB dataset. The authors additionally published the saved predictions from their model in the YCB toolbox. This toolbox also provides a framework for evaluating these saved results. Using the toolbox to evaluate their saved model produces results that are similar, but not exactly the same as were published in the paper. They might be from a different trained model than the one that produced the results reported in the paper. Thus we had to choose which values we want to use for the baseline. Oberweger, Rad, and Lepetit [28] compared to values from the saved model. They again get slightly different values than the toolbox evaluation, as did we, when implementing our own evaluation. We show the results of all sources side by side in Table 6.1. Our own evaluation framework gets similar numbers as their MATLAB code, but not exactly the same, which also is the case for Oberweger, Rad, and Lepetit [28]. For our baseline we use the results saved in the toolbox, but evaluate with our own evaluation framework since then we are able to compare also with other metrics that were not reported in the paper.

[1] `https://github.com/yuxng/YCB_Video_toolbox`

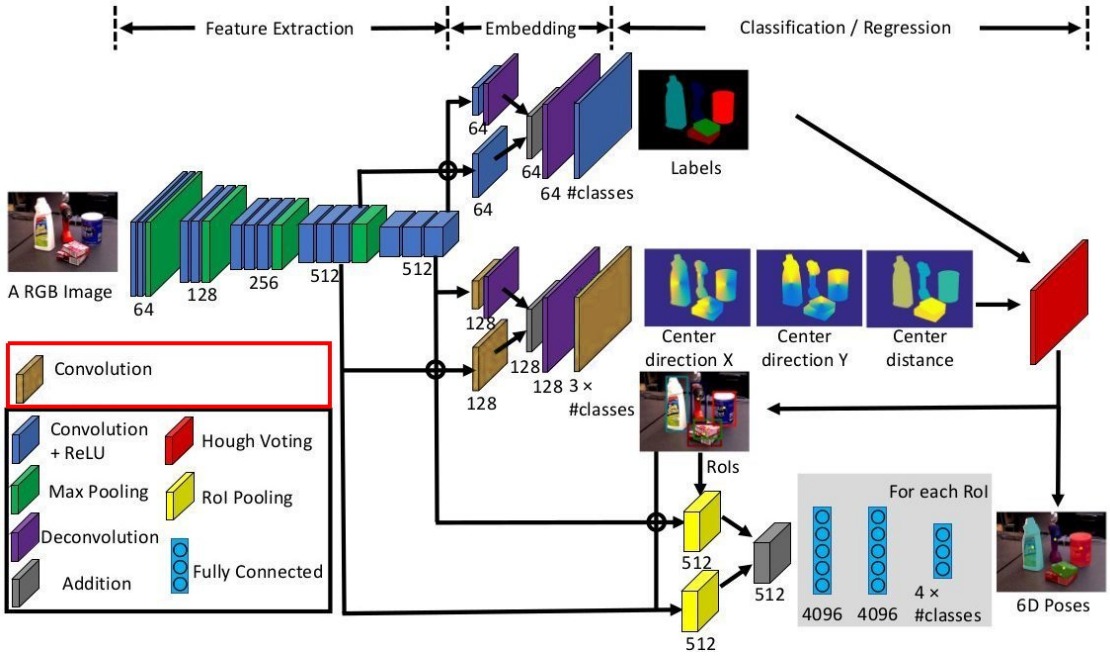


Figure 6.4: The PoseCNN architecture based on the code published by Xiang et al. [46] (image adapted from [46]). The original image did not contain convolutions without ReLu activations.

We implemented PoseCNN [46] as a baseline. For this we followed the description from the paper, apart from some differences we found in the published code[2]. However, the code shown in the repository has also been changed since the PoseCNN paper was published.

While the schematic depicting PoseCNN from the original paper shows every convolution to be followed by a ReLU, some layers explicitly do not have any activation function in their published code. We show the difference in Figure 6.4.

Also, we noticed that instead of predicting the depth directly, the published code is predicting the logarithm of the depth value, which is scaled back during the Hough voting. The fully-connected layers are followed by ReLUs, except for the last layer which returns both the output before and after using tanh. The implemented code also contains dropout in all the branches.

The YCB dataset contains 80.000 synthetic images and the paper states that they used 80.000 synthetic images for training, however, their code shows a possibility to render synthetic images online, during training, and they add the rendered objects on top of images from the SUN2012 [47] and ObjectNet3D [45] datasets, which was not reported in their paper. During training they use a synthetic image

[2] `https://github.com/yuxng/PoseCNN/`

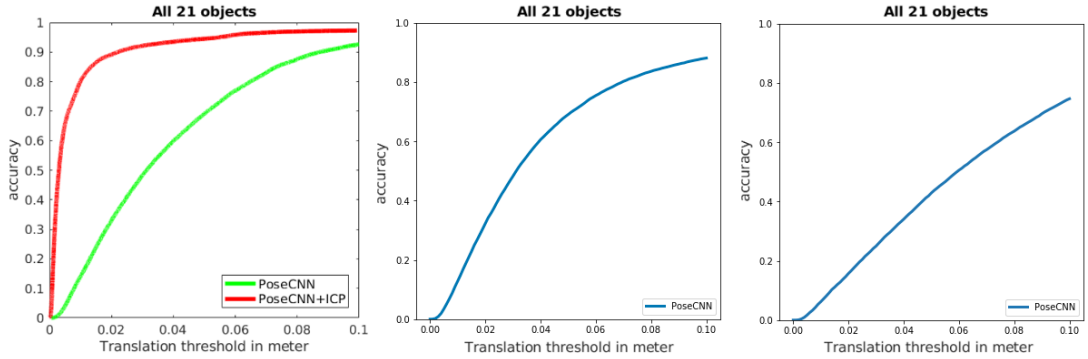


Figure 6.5: Left: The results for the translation from the original PoseCNN paper (image taken from [46]); middle: the results we achieved using the changed architecture, predicting the logarithm of the depth and scaling the depth by a factor of 100; right: our results when using the logarithm, but no depth scaling. We do not use ICP refinement for our results.

with a probability of 80%. If a synthetic image should be used, it is rendered and added to a random background. We do the same, but do not render new images, but use the ones provided in the dataset.

Training the network to achieve results similar to their paper turned out be a challenge. PoseCNN first trains segmentation and translation. We trained the network with training set and synthetic dataset for both the logarithm of the depth and without for at least three epochs each, checking during the training using the validation set and finally testing on the keyframes. For both options the translation error of the network was too high to reach the results of the paper. Especially the error of the depth prediction was very high, which caused the an overall high translation error, since the translation of the other dimensions is recovered using the predicted depth. We then tried scaling the depth to improve the estimate, which for a scaling factor of 100 gave results, that closely resemble the depicted performance in the PoseCNN paper. We show the evaluation curves in Figure 6.5. We used the model trained with segmentation and vertex loss after 40,000 iterations, as was indicated in an earlier version of the PoseCNN paper. Training these losses first, enables the network to train with its predicted RoIs immediately.

Another detail, that was not mentioned in the paper, but turned out to be important for the overall performance, was the initialization of the transposed convolution layers. The layers are initialized to perform a class-wise bilinear upsampling. Weights of connections between different classes are set to zero.

As the PoseCNN implementation, we chose a batch size of 2. The Shapeloss is extremely small compared to the other losses and training with a scaled loss

showed much better performance. This was true also for trials where the depth of the vertex loss was not scaled.

We also noticed that the Shapeloss had trouble learning. If we use it unscaled, even for toy examples, the proposed learning rate is too small. Therefore we also scaled the rotation loss. Additionally, we experimented with decreasing the learning rate, as in the published code. However, the resulting learning rate is very small and the network stops improving. We tried both training with and without tanh, as well as using tanh or not at test time. We got the best results when using tanh for training and not for testing.

Trying the possibilities above, the best result we got is shown in Table 6.2. The total, as Xiang et al. [46] reports them, that our re-implementation achieves, is similar to the original PoseCNN. Our model is worse at predicting the rotation though, as can be seen with the angle loss. The translation, on the other hand, is good enough, that the final AUC values are still similar.

We also evaluated the translation and segmentation class-wise, shown in Table 6.3. The segmentation of the toolbox model is better than of our implementation, our implementation has a smaller translation error. The potted meat can, the woodblock and the extra large clamp have a large translation error. The clamp objects have a small segmentation Intersection over Union (IoU) in both models.

	PoseCNN Paper		YCB toolbox		[28]	Our evaluation	
class	AUC P	AUC S	AUC P	AUC S	AUC	AUC P	AUC S
master_chef_can	50.9	84.0	50.18	83.92	50.1	50.08	83.72
cracker_box	51.7	76.9	53.07	76.86	52.9	52.94	76.56
sugar_box	68.6	84.3	68.42	84.25	68.3	68.33	83.95
tomato_soup_can	66.0	80.9	66.16	81.00	66.1	66.11	80.90
mustard_bottle	79.9	90.2	80.99	90.43	80.8	80.84	90.64
tuna_fish_can	70.4	87.9	70.65	88.05	70.6	70.56	88.05
pudding_box	62.9	79.0	62.70	79.05	62.2	62.22	78.72
gelatin_box	75.2	87.1	75.21	87.22	74.8	74.86	85.73
potted_meat_can	59.6	78.5	59.50	78.5	59.5	59.40	79.51
banana	72.3	85.9	72.33	85.99	72.1	72.16	86.24
pitcher_base	52.5	76.8	53.27	76.98	53.1	53.11	78.08
bleach_cleanser	50.5	71.9	50.31	71.56	50.2	50.22	72.81
bowl	6.5	69.7	3.33	69.61	69.8	3.10	70.31
mug	57.7	78.0	58.54	78.16	58.4	58.39	78.22
power_drill	55.1	72.8	55.31	72.70	55.2	55.21	72.91
wood_block	31.8	65.8	26.61	64.33	61.8	26.19	62.43
scissors	35.8	56.2	35.82	56.88	35.3	35.27	57.48
large_marker	58.0	71.4	58.27	71.73	58.1	58.11	70.98
large_clamp	25.0	49.9	24.59	50.16	50.1	24.47	51.05
extra_large_clamp	15.8	47.0	16.06	44.11	46.5	15.97	46.15
foam_brick	40.4	87.8	40.24	87.99	85.9	39.90	86.46
[46] total	**53.7**	**75.9**	53.71	75.81		53.70	76.12
SymC average	60.44	64.04	60.67	63.24		60.49	63.28
average by [28]	61.3		61.28		**61.0**	61.15	

Table 6.1: Comparison of different PoseCNN result values: First, from the published paper [46], second, the published model from the YCB toolbox evaluated with the toolbox, then, Oberweger, Rad, and Lepetit [28] values calculated from the model, as well as our own evaluation of the model. The different averages are from the different papers and are explained in Section 6.1.1. The average that was reported in the respective papers is bold. The coloured objects are symmetric.

	Our implementation			YCB toolbox model		
	AUC P	AUC S	angle	AUC P	AUC S	angle
002_master_chef_can	53.84	89.19	61.74	50.08	83.72	50.71
003_cracker_box	55.80	81.01	26.76	52.94	76.56	19.69
004_sugar_box	63.30	80.95	18.17	68.33	83.95	9.29
005_tomato_soup_can	57.49	80.87	59.40	66.11	80.90	23.17
006_mustard_bottle	62.79	88.33	58.97	80.84	90.64	9.94
007_tuna_fish_can	67.60	90.99	66.55	70.56	88.05	32.80
008_pudding_box	60.64	79.24	32.77	62.22	78.72	10.20
009_gelatin_box	87.26	93.22	12.63	74.86	85.73	5.25
010_potted_meat_can	55.39	75.92	68.85	59.40	79.51	31.24
011_banana	54.36	76.49	35.44	72.16	86.24	15.48
019_pitcher_base	64.47	84.63	17.48	53.11	78.08	11.98
021_bleach_cleanser	43.47	73.43	53.74	50.22	72.81	20.85
024_bowl	20.95	69.50	108.69	3.09	70.31	130.54
025_mug	60.19	80.85	19.88	58.39	78.22	19.44
035_power_drill	64.60	82.42	21.28	55.21	72.91	9.91
036_wood_block	0.13	17.54	109.87	26.19	62.43	23.63
037_scissors	58.65	77.61	53.10	35.27	57.48	43.98
040_large_marker	59.36	72.43	94.49	58.11	70.98	92.44
051_large_clamp	22.13	58.70	97.09	24.47	51.05	97.89
052_extra_large_clamp	4.57	61.50	136.49	15.97	46.15	126.82
061_foam_brick	43.96	87.63	130.49	39.90	86.46	160.37
[46] total	52.20	78.22		53.71	76.12	
SymC	60.58	58.97	43.83	60.49	63.28	25.40
[28] average	60.19			61.15		

Table 6.2: Our re-implementation vs the PoseCNN results, both evaluated using our own framework. The results from PoseCNN shown are from the model in the YCB toolbox. The angle error is given in degrees. The SymC average of the angle is the class-wise average of the angles of the non-symmetric objects. The symmetric objects are coloured.

	Our implementation		YCB toolbox model	
	translation	SEG IoU	translation	SEG IoU
002_master_chef_can	0.0224	0.8842	0.0329	0.8836
003_cracker_box	0.0284	0.8805	0.0402	0.9063
004_sugar_box	0.0331	0.8905	0.0306	0.9371
005_tomato_soup_can	0.0559	0.8715	0.0582	0.8782
006_mustard_bottle	0.0169	0.9262	0.0172	0.9426
007_tuna_fish_can	0.0180	0.9001	0.0241	0.9217
008_pudding_box	0.0333	0.6387	0.0369	0.7790
009_gelatin_box	0.0107	0.9246	0.0249	0.9082
010_potted_meat_can	0.1297	0.7590	0.0524	0.8537
011_banana	0.0341	0.8669	0.0243	0.9096
019_pitcher_base	0.0270	0.9513	0.0443	0.9574
021_bleach_cleanser	0.0463	0.8697	0.0486	0.8919
024_bowl	0.0559	0.8611	0.0523	0.9129
025_mug	0.0376	0.8809	0.0400	0.8769
035_power_drill	0.0284	0.8806	0.0459	0.8798
036_wood_block	0.1802	0.5556	0.0634	0.8331
037_scissors	0.0343	0.6593	0.0640	0.6643
040_large_marker	0.0356	0.6532	0.0389	0.7027
051_large_clamp	0.0880	0.4718	0.1149	0.4722
052_extra_large_clamp	0.1179	0.4381	0.1958	0.3528
061_foam_brick	0.0266	0.8723	0.0248	0.9011
class-wise average	0.0505	0.7922	0.0512	0.8269

Table 6.3: An evaluation of the class-wise segmentation, measured as the IoU (SEG IoU) and of the translation estimation performance of the YCB toolbox model of PoseCNN and our re-implementation.

6.4 Estimating Quaternions Pixel-wise

Developing our model was an incremental process, since some model behaviour surprised us and inspired new ideas. Therefore we will first show some of the insights we gained when trying our model and how we used them to improve it. Lastly, we will compare different versions to each other and to PoseCNN.

6.4.1 Training ConvPoseCNN

For training the pixel-wise model we used the architecture described in Section 5.1. While PoseCNN needs to pretrain segmentation and translation, since they use the RoIs for training, ConvPoseCNN can be trained directly. We trained one model with the L2 loss as the rotation loss and one with the Qloss, in the following we call these two trained models ConvPoseCNN L2 and ConvPoseCNN Qloss. As for PoseCNN, we used SGD with momentum (0.9) and a learning rate of 0.001 and then trained for around 300.000 iterations with batchsize 2 on the training and synthetic data set. We used the validation set to select a good model and tested on the keyframes. As for PoseCNN we had to scale the depth by 100, since the translation estimation of the networks are the same. We did not scale any of the other losses.

6.4.2 Visualizing the Weighted Quaternions

In order to understand which averaging or clustering methods might be successful in determining a final quaternion from the pixel-wise estimations, we first visualized the pixel-wise predictions. For this we used PCA (reducing to three dimensions) as well as a rotation-axis representation. Both showed similar results therefore we will show only the rotation-axis images.

For the rotation-axis representation, we mapped each quaternion (q_w, q_x, q_y, q_z) to a 3D unit ball. q_x, q_y, q_z are represented as the direction of a vector with the direction of q_x, q_y, q_z on the x, y and z coordinate axis. q_w is shown in the length of the vector. We plotted the endpoint of this vector for each quaternion using the visualization framework visdom[3].

Looking at the visualizations showed us that against our intuition the quaternion that we were trying to estimate was not the center of any quaternion cluster that our network predicted. Instead in almost all seen samples it is lying on the border or outside the area, where the predictions lie. Figure 6.6 shows an example visualization.

[3] `https://github.com/facebookresearch/visdom`

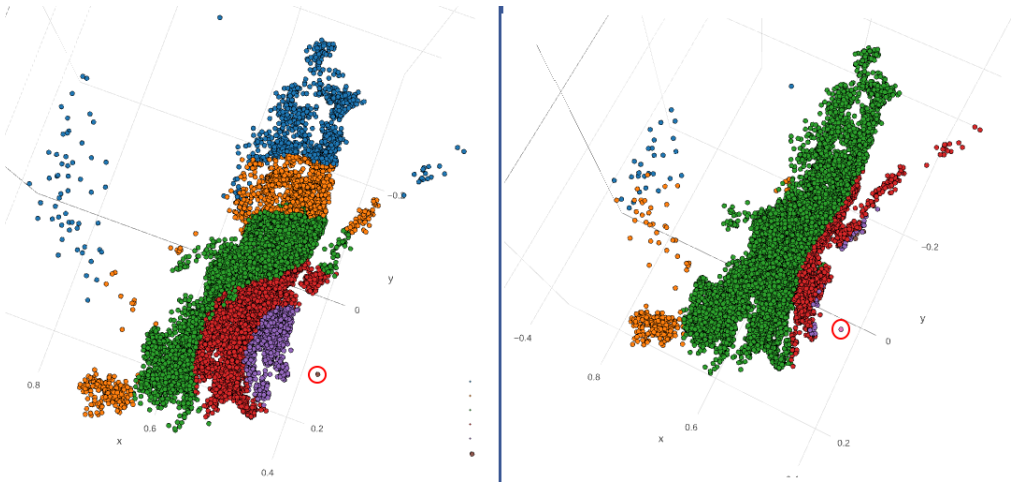


Figure 6.6: The quaternions predicted for a bleach cleanser by CoordPoseCNN L2, visualized using the rotation axis representation. The left image shows the quaternions coloured by the angle between them and the ground truth quaternion The lowest error is less than 20 degrees (purple), then 30 (red), 40 (green), 50 (orange) and below 100 (blue). The right image shows the same quaternions coloured by the Sloss. The order of the colours with regard to the error size is the same as before. The ground truth quaternion is circled in red.

ConvPoseCNN L2	[46] total		SymC		SymC ROT	
	AUC P	AUC S	AUC P	AUC S	AUC P	AUC S
PoseCNN	53.71	76.12	60.49	63.28	86.90	87.01
simple avg.	55.76	78.70	61.25	58.38	76.67	82.08
norm weighted avg.	56.51	78.91	62.3	58.42	78.41	82.37
[27] avg.	56.59	78.86	62.16	**58.54**	78.04	82.28
[27] norm weighted avg	**57.13**	**79.01**	**62.99**	58.52	**79.42**	**82.53**
[27] segm. weighted avg	56.63	78.87	62.23	58.54	78.22	82.28

Table 6.4: Results of the different averaging strategies, defined in Section 6.1.1, for ConvPoseCNN L2. The best results, excluding PoseCNN, are in bold.

ConvPoseCNN Qloss	[46] total		SymC		SymC ROT	
	AUC P	AUC S	AUC P	AUC S	AUC P	AUC S
PoseCNN	53.71	76.12	60.49	63.28	86.90	87.01
simple avg.	55.90	76.47	63.12	52.96	85.04	84.50
norm weighted avg.	56.53	76.74	64.06	53.23	86.64	85.67
[27] avg.	56.57	76.69	63.51	53.53	85.70	85.62
[27] norm weighted avg.	**56.98**	**76.90**	**64.28**	**53.69**	**87.05**	**86.39**
[27] segm. weighted avg.	56.67	76.74	63.64	53.64	85.94	85.97

Table 6.5: Results of the different averaging strategies, defined in Section 6.1.1, for ConvPoseCNN Qloss. The best results, excluding PoseCNN, are in bold.

6.4.3 Comparison of Different Averaging Strategies

First, we evaluated some of the simple averaging strategies described in Section 2.8. We show the results for the model trained with the L2 loss in Table 6.4 using the AUC averages as described in Section 6.1.1. As weights we use either the norm of the quaternions before normalizing, which we also call their confidence, or the segmentation confidence.

Markley's averaging method [27] performs better than the simple average and the norm weights improve both averages. The results for the non-symmetric objects are a bit better in comparison to PoseCNN. Using the AUC S from the paper, the methods also all perform better than PoseCNN. However, as the SymC average shows, are the poses of the symmetric objects not estimated better than by PoseCNN. Also the norm weights do not seem to work as well for the symmetric objects as for the non-symmetric objects. When looking at the SymC AUCs calculated using only the rotation, we see a big difference to PoseCNN. The rotation estimation of the ConvPoseCNN is worse than of PoseCNN, however, the translation must be better, as we confirm in Section 6.4.7. The comparison between the SymC losses with and without rotation also shows, that there is a difference between the translation estimation of symmetric objects and non-symmetric objects.

We show the results for the model trained with the Qloss in Table 6.5. The results are better for the non-symmetric objects and worse for the symmetric ones, compared to the L2 trained model, when considering both rotation and translation. For only the rotation, the Qloss model performs better than the L2 model and very similar to PoseCNN. Markley's method weighted by the confidence, produces the best results.

ConvPoseCNN L2	[46] total		SymC		SymC ROT	
	AUC P	AUC S	AUC P	AUC S	AUC P	AUC S
PoseCNN	53.71	76.12	60.49	63.28	86.90	87.01
pruned(0)	57.13	79.01	62.99	58.52	79.42	82.53
pruned(0.5)	57.43	79.14	63.52	58.55	80.34	82.91
pruned(0.75)	**57.43**	79.19	**63.59**	58.60	**80.48**	83.16
pruned(0.85)	57.39	79.21	63.53	58.68	80.42	83.39
pruned(0.9)	57.37	**79.23**	63.48	58.74	80.34	83.53
pruned(0.95)	57.39	79.21	63.53	58.68	80.42	83.39
most confident	57.11	79.22	63.17	**59.01**	79.80	**84.11**

Table 6.6: Results of the averaging strategies using the norm weighting for the ConvPoseCNN L2. All, except the most confident one, were averaged using Markley's [27] method. The best results, excluding PoseCNN, are in bold.

Removing Unconfident Quaternions

Since the weighting with the confidences worked so well we increased their effect for the averaging. For this we used the following strategies:

- pruned(x): The quaternions are sorted by their confidences and the x percent less confident ones are removed
- most confident: Only the most confident quaternion is taken.

The remaining quaternions are averaged using Markley's weighted averaging method. The results for the L2 model are shown in Table 6.6. Removing less confident quaternions shows an improvement for both losses. The symmetric objects show a clear improvement when there are less predictions, which might mean, that the predictions span multiple possible shape equivalent possibilities. The non-symmetric objects perform better if there are around 25% of quaternions left, however, the performance does not change much when up to 95% are removed.

The Qloss trained model, shown in Table 6.7, similarily improves when unconfident quaternions are removed. Here also the non-symmetric objects perform better when more quaternions are removed, the preformance difference between the different pruning rates is smaller as for the L2 model.

6.4.4 Comparison with Clustering Strategies

For clustering we used both RANSAC strategies, described in 5.1.2. We used the angle loss as a distance metric, since it can be calculated fast but unlike for the L2 metric there are no problems with the quaternion symmetry.

ConvPoseCNN Qloss	[46] total		SymC		SymC ROT	
	AUC P	AUC S	AUC P	AUC S	AUC P	AUC S
PoseCNN	53.71	76.12	60.49	63.28	86.90	87.01
pruned(0)	56.98	76.90	64.28	53.69	87.05	86.39
pruned(0.5)	57.07	76.96	64.49	53.76	87.47	86.76
pruned(0.75)	**57.12**	77.03	64.61	53.89	87.69	87.16
pruned(0.8)	57.11	77.04	64.64	53.91	87.73	87.21
pruned(0.9)	57.07	**77.06**	**64.65**	53.96	**87.77**	87.31
pruned(0.95)	57.00	**77.06**	64.62	**53.98**	87.72	**87.35**
most confident	56.72	77.04	64.54	53.94	87.56	87.27

Table 6.7: Results of the averaging strategies using the norm weighting for the ConvPoseCNN Qloss. All, except the most confident one, were averaged using Markley's [27] method. The best results, excluding PoseCNN, are in bold.

The results for the L2 trained model are shown in Table 6.8. The number in the brackets indicates the angle threshold in radian. For all we used 50 iterations. For comparison the best performing averaging strategies are also listed. Symmetric and non-symmetric objects perform best, according to only the rotation, with an inlier threshold of 0.2 radians (around 11.5 degrees). Even then they are slightly worse than the best averaging strategies. The weighted RANSAC variant performs generally a bit better than the non-weighted one for the same inlier thresholds.

6.4.5 Confidence Weighting

Since weighting the predicted quaternions with their confidence gave good results, we experimented with learning confidence weights. For this we used a slightly changed architecture, where a confidence weight is learned with each quaternion. So instead of returning four times the number of classes for each pixel, the network now returns five times the number of classes. The quaternions are normalized as before and the fifth value is used as the confidence weight.

Generally, the training of this model suffered from instabilities. We used the already trained model from the previous experiments as a starting point. In order to train the confidence weights we calculated the weighted average of the predicted quaternions with regard to the learnt weights for each object and then calculated the loss with that quaternion. For the average we used the simple weighted average, since PyTorch's autograd module can not calculate the gradient for Markley's average.

Doing this, we got the results shown in Table 6.10. The model has better results

ConvPoseCNN L2	[46] total		SymC		SymC ROT	
	AUC P	AUC S	AUC P	AUC S	AUC P	AUC S
PoseCNN	53.71	76.12	60.49	63.28	86.90	87.01
RANSAC(0.1)	57.18	79.16	63.26	58.69	80.01	83.39
RANSAC(0.2)	57.36	79.20	63.31	58.76	80.11	83.59
RANSAC(0.3)	57.27	79.20	63.14	58.80	79.80	83.48
RANSAC(0.4)	57.00	79.13	62.70	**58.91**	79.05	83.55
weighted RANSAC(0.1)	57.27	79.20	63.39	58.73	80.25	83.53
weighted RANSAC(0.2)	**57.42**	**79.26**	**63.48**	58.85	**80.39**	**83.72**
weighted RANSAC(0.3)	57.38	79.24	63.38	58.83	80.15	83.58
pruned(0.75)	**57.43**	79.19	**63.59**	58.60	**80.48**	83.16
most confident	57.11	79.22	63.17	**59.01**	79.80	**84.11**

Table 6.8: Results for different RANSAC clustering strategies for the L2 trained ConvPoseCNN and the best performing averaging methods for comparison. The best results excluding the PoseCNN results are marked bold.

ConvPoseCNN Qloss	[46] total		SymC		SymC ROT	
	AUC P	AUC S	AUC P	AUC S	AUC P	AUC S
posecnn	53.71	76.12	60.49	63.28	86.90	87.01
RANSAC(0.1)	57.14	**77.08**	64.59	53.98	87.81	87.83
RANSAC(0.2)	57.05	77.05	64.42	54.03	87.26	**87.94**
RANSAC(0.3)	56.79	76.97	64.07	54.06	86.35	87.81
RANSAC(0.4)	56.45	76.89	63.62	**54.09**	85.34	87.59
weighted RANSAC(0.05)	57.11	77.07	64.74	53.91	**88.04**	87.54
weighted RANSAC(0.1)	**57.16**	**77.08**	**64.75**	53.95	88.02	87.60
weighted RANSAC(0.2)	57.04	77.05	64.59	54.02	87.52	87.68
weighted RANSAC(0.3)	56.82	76.99	64.32	54.02	86.80	87.47
weighted RANSAC(0.4)	56.59	76.91	64.00	53.98	86.05	87.18
pruned(0.9)	57.07	77.06	64.65	53.96	87.77	87.31
pruned(0.95)	57.00	77.06	64.62	53.98	87.72	87.35

Table 6.9: Results for different RANSAC clustering strategies for the Qloss trained ConvPoseCNN and the best performing averaging methods for comparison. The best results excluding the PoseCNN results are marked bold.

	[46] total		SymC		SymC ROT	
	AUC P	AUC S	AUC P	AUC S	AUC P	AUC S
[27] avg.	56.59	78.86	62.16	**58.54**	78.04	82.28
[27] weighted avg.	57.13	**79.01**	62.99	58.52	**79.42**	82.53
Conf [27] avg.	57.45	78.95	64.11	55.82	78.53	**83.37**
Conf [27] weighted avg.	**57.48**	78.97	**64.12**	55.83	78.54	83.31

Table 6.10: The results of ConvPoseCNN L2, at the top of the table, and the results of the network adapted to learn a weighting, below (Conf). The second network was initialized with the weights of the first one.

for the AUC P, but worse for AUC S than ConvPoseCNN. The weighting by the learnt confidences does not improve the average much. The improvement using the confidence weights in the original ConvPoseCNN is greater, therefore we conclude that the other weighting leads to a better performance.

6.4.6 Training with the Shapeloss

Since computing the Shapeloss for all quaternions is too time consuming for training, we had to calculated an average quaternion first and then calculate the Shapeloss to the ground truth quaternion. We use the simple average. Using Markley's average would probably produce better results, but PyTorch's autograd engine is not able to automatically differentiate the operation, therefore, using it for training would require calculating and implementing it.

As for the training of PoseCNN it was necessary to weigh the loss, so $L = L_{seg} + L_{trans} + \alpha L_{rot}$. The best results for ConvPoseCNN Shapeloss were achieved using $\alpha = 10$. They are shown in Table 6.11. For the Shapeloss trained model, the norm weights improve the averages, but pruning does not. Since the network is trained with an average, it is likely that the quaternions are encouraged to lie around the quaternion, so that the average is close to the estimate. Supporting this idea is that pruning the less confident quaternions does not improve the results. Also, RANSAC produces worse estimates than the averaging methods, but improves the higher the distance threshold is. If the distance threshold is large, then RANSAC is similar to an average.

6.4.7 Influence of Translation and Segmentation

Even though our aim was to improve the rotation estimation of PoseCNN, the results show that this can not be isolated from the translation estimation. Scaling the losses seems to create a trade off between translation and rotation estimation

ConvPoseCNN Shapeloss	[46] total		SymC		SymC ROT	
	AUC P	AUC S	AUC P	AUC S	AUC P	AUC S
PoseCNN	53.71	76.12	60.49	63.28	86.90	87.01
simple avg.	53.11	78.55	59.66	56.33	72.71	87.01
simple weighted avg.	55.11	79.08	62.29	56.41	76.64	87.15
[27] avg.	54.27	78.94	61.21	56.35	74.89	87.03
[27] norm weighted avg.	**55.54**	79.27	**62.77**	56.42	**77.54**	**87.16**
[27] seg. weighted avg.	54.44	79.00	61.36	56.40	75.16	87.10
pruned(50)	55.33	**79.29**	62.40	56.56	77.11	87.12
pruned(0.75)	54.62	79.09	61.55	**56.57**	75.71	86.88
pruned(0.85)	53.86	78.85	60.67	56.52	74.41	86.47
pruned(0.9)	53.23	78.66	60.00	56.47	73.43	86.14
RANSAC(0.2)	49.44	77.65	56.09	56.33	67.97	86.43
RANSAC(0.3)	50.47	77.92	57.30	56.41	69.62	86.66
RANSAC(0.4)	51.19	78.09	58.17	56.40	70.85	86.72
weighted RANSAC(0.2)	49.56	77.73	56.29	56.33	68.35	86.36
weighted RANSAC(0.3)	50.54	77.91	57.33	56.41	69.82	86.58
weighted RANSAC(0.4)	51.33	78.13	58.16	56.45	70.99	86.76

Table 6.11: The results for ConvPoseCNN Shapeloss. The methods are the same as used for the other evaluations. The pruning methods use Markley's [27] average. The best results excluding the PoseCNN results are marked bold.

	[28]	[28] ROT	translation	SEG IoU
PoseCNN	61.15	86.93	0.0520	0.8369
PoseCNN (reimplementation)	60.19	78.42	0.0484	0.8136
ConvPoseCNN Qloss	62.18	87.92	0.0565	0.7725
ConvPoseCNN Shapeloss	61.26	79.83	0.0455	0.8038
ConvPoseCNN(L2)	62.38	81.18	0.0411	0.8044
Ground truth semantic segmentation used				
PoseCNN (re-implementation)	62.08	79.75	0.0345	1
ConvPoseCNN Qloss	63.57	88.72	0.0386	1
ConvPoseCNN Shapeloss	63.23	80.11	0.0316	1
ConvPoseCNN L2	64.52	81.82	0.0314	1

Table 6.12: The average translation error, the segmentation IoU and the AUC losses, averaged as by Oberweger, Rad, and Lepetit [28] for different models. The AUC results were achieved using weighted RANSAC 0.1 for ConvPoseCNN Qloss, Markley's norm weighted average for ConvPoseCNN Shapeloss and weighted RANSAC(0.2) for ConvPoseCNN L2.

performance. While overall our models reach similar AUC values, some perform better for the rotation and some better for the rotation even though, the translation estimation branch is the same for all of these networks. We want to look at the performance of the model with regard to translation and segmentation more closely. For this we report the average translation error and the segmentation IoU for all models in Table 6.12. They show that there is a strong influence of the translation estimation on the AUC losses. However, for the models with a better translation estimation, the orientation estimation is worse. We conclude that finding a proper balancing between translation and orientation estimation is important but difficult to achieve. Also, the segmentation performance affects the results.

We also included results in Table 6.12 that were produced by evaluating using the ground truth semantic segmentation, in order to evaluate how much our model's performance could improve by the segmentation performance alone. If the segmentation is perfect, then the rotation and the translation estimation of all models improves. Even the re-implemented PoseCNN improves its rotation, therefore the RoIs must have improved by the better translation and inlier estimation.

	[46] total		SymC		SymC ROT	
	AUC P	AUC S	AUC P	AUC S	AUC P	AUC S
ITERATIVE	0.07	0.18	0.06	0.15	24.31	80.38
ITERATIVE RANSAC	2.18	4.85	2.33	1.71	33.96	80.65
EPNP	7.20	16.09	7.71	6.08	**64.21**	**83.71**
EPNP RANSAC	**9.19**	**21.38**	**9.47**	**9.55**	58.89	82.70

Table 6.13: Results of the CoordPoseCNN for different PNP methods.

6.5 Estimating Object Coordinates

We trained the CoordPoseCNN as described in Section 5.2. Again scaling the loss was essential for the model to learn. A scaling of 100 for the loss of the the object coordinate prediction, L_{coor}, gave the best results. We trained the network for around 300,000 iterations and then tried different PnP solvers offered by the OpenCV implementation, described in Section 2.6. Doing this we got the results shown in Table 6.13. As expected, this approach performs badly on the symmetric, textureless objects. But it also performs worse than the pixel-wise quaternion estimation in general.

6.5.1 Adding External Translation

From the previous results you can see that CoordPoseCNN does not estimate the translation well. Though not reported in the table, we calculated the translation error for the dimensions separately and could see that especially the depth prediction error is high. The results of the rotation are also not that good. It is possible to combine the estimated rotation with a better translation estimation. For this we used the translation estimation of PoseCNN and the EPNP method for RANSAC since it has the best results for the rotation only AUCs. Using these two regressors together improved the result to a SymC AUC P loss of 53.19 and a SymC AUC S loss 58.99. That is worse for AUC P than for the ConvPoseCNN L2 and Qloss models. AUC S on the other hand surprisingly reaches the performance of ConvPoseCNN L2.

6.6 Comparison to PoseCNN

We compare the best models from our previous evaluation more closely with PoseCNN. Some models are better in translation and some in rotation, as we noticed before. We chose the best ConvPoseCNN model by the [28] average and

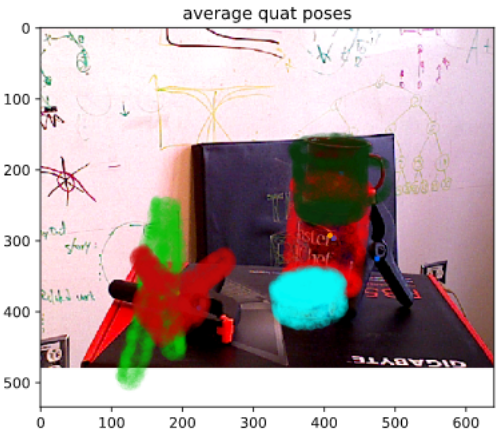


Figure 6.7: An example where ConvPoseCNN confuses the clamps.

report the class-wise results in Table 6.16. The table shows well some of the difficulties that we noticed in the dataset: The two clamps are easy to confuse. Because of this the segmentation is bad and rotation and translation errors are high. When looking at visualizations of the prediction for images with both clamps we can often see that the center of one clamp is close to that of the other, because the network can not distinguish between the two different classes, which leads to a bad translation. An example of that is shown in Figure 6.7.

Another object that has very bad results is the woodblock. Here the segmentation is not much worse than the average, but the results for translation and AUC S are far worse than average. We also show some qualitative results of the ConvPoseCNN L2 model in Figure 6.8. The detailed results for our baseline PoseCNN are shown in Table 6.17.

The weak performance of the ConvPoseCNN for some of the classes might indicate some weakness of the ConvPoseCNN, however, also our implementation of PoseCNN shows a much higher translation error for the woodblock class than the PoseCNN model from the toolbox. Here also the segmentation IoU is below average. The bad results for this class are therefore likely caused by the training process and not due to the ConvPoseCNN architecture itself.

6.7 Time Comparisons

We timed our models on a NVIDIA GTX 1080 Ti with 11 gigabytes of memory. Table 6.14 lists the training times for the different models, as well as the model sizes when saved. The training of the ConvPoseCNNs is faster and the models are

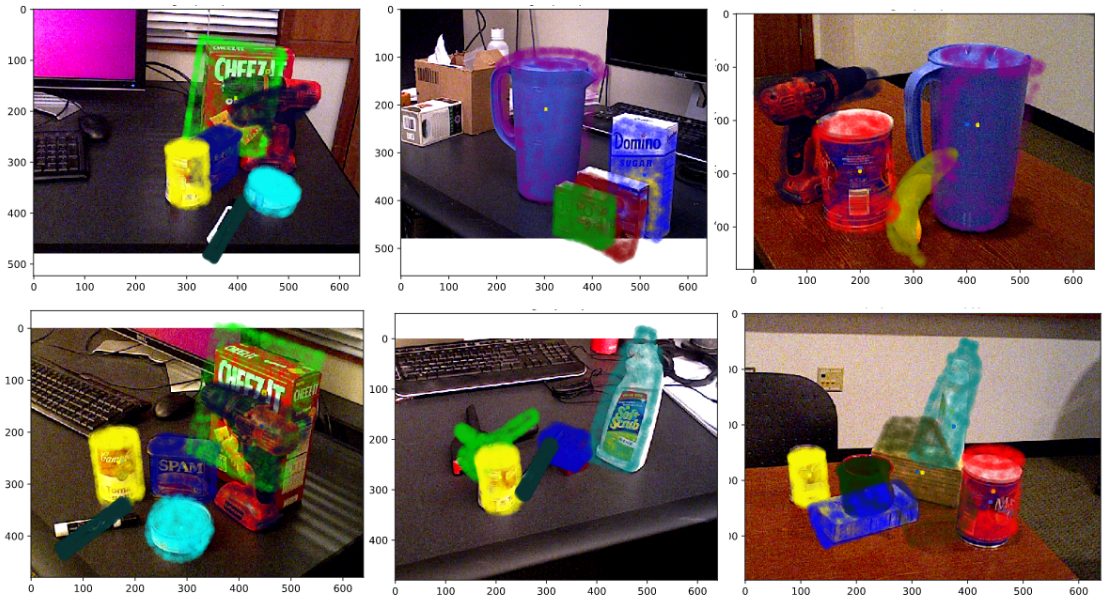


Figure 6.8: Some qualitative results of ConvPoseCNN L2 using pruned(0.75).

	it/s	model size
PoseCNN	1.18	1.1 GB
ConvPoseCNN L2	2.09	308,9 MB
ConvPoseCNN Qloss	2.09	308,9 MB
ConvPoseCNN Shapeloss	1.99	308,9 MB
CoordPoseCNN	3.89	164 MB

Table 6.14: Training times for our implementation of PoseCNN, ConvPoseCNN and CoordPoseCNN reported in iterations per second. The values are produced using a batch size of 2 and averaging over 400 iterations.

		time total in seconds	it/s
PoseCNN		557	5.29
ConvPoseCNN	simple average	538	5.48
	Markley	572	5.16
	Markley norm weighted	572	5.16
	Pruned(0.75)	586	5.03
	RANSAC	838	3.52
	weighted RANSAC	2908	1.01

Table 6.15: The total time measured on the test set with 2949 images and the resulting number of iterations per second. The batch size is 1.

smaller compared to PoseCNN, CoordPoseCNN is even smaller and faster.

The speed of the ConvPoseCNN models at test time depend on the method used to determine the final quaternion. The times on the test set are shown in Table 6.15. A ConvPoseCNN L2 model was used to measure the time for the ConvPoseCNN model. For the averaging methods the times do not differ much from PoseCNN. PoseCNN takes longer to produce the output, but then does not need to perform any other step. For ConvPoseCNN the simple averaging method is the fastest, followed by the other averaging methods. RANSAC is slow. The averaging methods are implemented in PyTorch since we also use them for training. They could possibly be sped up for testing though by using a different implementation.

class	AUC P	AUC S	angle	translation	SEG IoU
002_master_chef_can	62.32	89.55	48.90	0.0193	0.88
003_cracker_box	66.69	83.78	11.55	0.0293	0.89
004_sugar_box	67.19	82.51	10.43	0.0315	0.90
005_tomato_soup_can	75.52	88.05	29.25	0.0450	0.87
006_mustard_bottle	83.79	92.59	17.67	0.0110	0.93
007_tuna_fish_can	60.98	83.67	38.88	0.0322	0.90
008_pudding_box	62.17	76.31	17.54	0.0527	0.64
009_gelatin_box	83.84	92.92	22.53	0.0098	0.93
010_potted_meat_can	65.86	85.92	38.85	0.0368	0.83
011_banana	37.74	76.30	76.08	0.0275	0.82
019_pitcher_base	62.19	84.63	19.79	0.0288	0.95
021_bleach_cleanser	55.14	76.92	38.59	0.0377	0.83
024_bowl	3.55	66.41	123.61	0.0619	0.82
025_mug	45.83	72.05	23.03	0.0489	0.77
035_power_drill	76.47	88.26	13.08	0.0193	0.88
036_wood_block	0.12	25.90	105.10	0.1195	0.71
037_scissors	56.42	79.01	62.43	0.0362	0.68
040_large_marker	55.26	70.19	75.98	0.0358	0.63
051_large_clamp	29.73	58.21	112.26	0.0948	0.42
052_extra_large_clamp	21.99	54.43	84.86	0.1056	0.35
061_foam_brick	51.80	88.02	103.80	0.0241	0.87
[46] total	57.43	79.19			
SymC	63.59	58.60	34.04	0.0432	0.79
[28] average	62.40				

Table 6.16: The detailed results for the ConvPoseCNN(L2) pruned(0.75) model, that has the best score as defined by Oberweger, Rad, and Lepetit [28] of all ConvPoseCNN models. The averages stated for the segmentation IoU and the translation error are a class-wise average. For the angle loss average the angle losses of the symmetric objects were not considered.

class	AUC P	AUC S	angle	translation	segm. IoU
002_master_chef_can	50.08	83.72	50.71	0.0329	0.88
003_cracker_box	52.94	76.56	19.69	0.0402	0.91
004_sugar_box	68.33	83.95	9.29	0.0306	0.94
005_tomato_soup_can	66.11	80.90	23.17	0.0582	0.88
006_mustard_bottle	80.84	90.64	9.94	0.0172	0.94
007_tuna_fish_can	70.56	88.05	32.80	0.0241	0.92
008_pudding_box	62.22	78.72	10.20	0.0369	0.78
009_gelatin_box	74.86	85.73	5.25	0.0249	0.91
010_potted_meat_can	59.40	79.51	31.24	0.0524	0.85
011_banana	72.16	86.24	15.48	0.0243	0.91
019_pitcher_base	53.11	78.08	11.98	0.0443	0.96
021_bleach_cleanser	50.22	72.81	20.85	0.0486	0.89
024_bowl	3.09	70.31	130.54	0.0523	0.91
025_mug	58.39	78.22	19.44	0.0400	0.88
035_power_drill	55.21	72.91	9.91	0.0459	0.88
036_wood_block	26.19	62.43	23.63	0.0634	0.83
037_scissors	35.27	57.48	43.98	0.0640	0.66
040_large_marker	58.11	70.98	92.44	0.0389	0.70
051_large_clamp	24.47	51.05	97.89	0.1149	0.47
052_extra_large_clamp	15.97	46.15	126.82	0.1958	0.35
061_foam_brick	39.90	86.46	160.37	0.0248	0.90
[46] total	53.71	76.12			
SymC	60.49	63.28	25.40	0.0512	0.83
[28] average	61.15				

Table 6.17: The detailed evaluation of our PoseCNN baseline. The values are produced using the model from the YCB toolbox and our evaluation framework. The averages stated for the segmentation IoU and the translation error are a class-wise average. For the angle loss average the angle losses of the symmetric objects were not considered.

7 Conclusions

In this thesis we developed two fully-convolutional architectures: ConvPoseCNN and CoordPoseCNN based on the PoseCNN [46] architecture and compared them on the YCB dataset [46]. CoordPoseCNN estimates the visible object coordinates pixel-wise and recovers the pose using PnP. The results show a large error of the depth prediction. Also the rotation estimation performs worse than PoseCNN.

ConvPoseCNN replaces the fully-connected rotation estimation branch of the PoseCNN model with a convolutional architecture that produces pixel-wise quaternion predictions. We experimented with using the L2 loss, the Qloss and Shape loss for regressing the orientation. For the resulting quaternion predictions we evaluated different averaging and clustering strategies and experimented with weighting the predictions. ConvPoseCNN reaches comparable results to PoseCNN for multiple training and averaging or clustering strategies. The model trained with L2 has a lower translation estimation error than PoseCNN, but a higher error considering the rotation, the model trained with Qloss reaches similar values for the rotation estimation than PoseCNN, but has a higher overall translation error.

Additionally, we re-implemented PoseCNN and evaluated the performance of the three internal branches, segmentation, translation, and rotation, separately. Even though ConvPoseCNN changes only the rotation estimation, we noticed a trade off effect between the translation and rotation estimation performance, that is caused by different loss scaling. Also, improving the segmentation would improve the whole performance further. We shifted the emphasis of the model more towards the estimation of the translation, when we scaled the depth. All in all, training PoseCNN and our own models is difficult and the parameters need to be selected carefully.

ConvPoseCNN and CoordPoseCNN are faster to train and use less memory than our own implementation of PoseCNN.

A contribution of PoseCNN is the training with Shape loss to improve the rotation prediction of symmetric objects. We also used Shape loss for training, but the results did not outperform the other models for the symmetric or the non-symmetric objects. A possible explanation is that we train with the average of the predicted quaternions. If the quaternions are divided between different possible rotations, then the average of these shape equivalent estimations might not be

shape equivalent. Therefore, the quaternions are encouraged during training to focus on one possible rotation. That removes some of the benefits of the Shape loss.

For future work it could be interesting to investigate methods that train pixel-wise with the Shape loss, or to find other methods that enable the network to train with multiple correct rotations.

We believe that ConvPoseCNN can be easily altered for the detection of multiple object instances by using the Hough inliers to select the quaternions belonging to each instance. This might make it more robust in cases where the instances are so close to each other that they lie in each others bounding boxes. If in the future such a dataset arises, it would be interesting to confirm this hypothesis.

A problem of ConvPoseCNN, as well as PoseCNN, is the balancing of the three losses. Finding a way to remove this problem would likely improve the performance of the networks and reduce the amount of work needed to train the regressor.

Bibliography

[1] Gideon Billings and Matthew Johnson-Roberson. "SilhoNet: an RGB method for 3D object pose estimation and grasp planning". In: *Arxiv preprint arxiv:1809.06893* (2018).

[2] Eric Brachmann, Alexander Krull, Frank Michel, Stefan Gumhold, Jamie Shotton, and Carsten Rother. "Learning 6D object pose estimation using 3D object coordinates". In: *European conference on computer vision*. Springer. 2014, pp. 536–551.

[3] Eric Brachmann, Frank Michel, Alexander Krull, Michael Ying Yang, Stefan Gumhold, et al. "Uncertainty-driven 6D pose estimation of objects and scenes from a single RGB image". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 3364–3372.

[4] G. Bradski. "The OpenCV Library". In: *Dr. dobb's journal of software tools* (2000).

[5] Thanh-Toan Do, Ming Cai, Trung Pham, and Ian D. Reid. "Deep-6DPose: recovering 6D object pose from a single RGB image". In: *CoRR* abs/1802.10367 (2018). arXiv: 1802.10367. URL: http://arxiv.org/abs/1802.10367.

[6] Alexey Dosovitskiy, Philipp Fischer, Eddy Ilg, Philip Hausser, Caner Hazirbas, Vladimir Golkov, Patrick Van Der Smagt, Daniel Cremers, and Thomas Brox. "Flownet: learning optical flow with convolutional networks". In: *Proceedings of the IEEE international conference on computer vision*. 2015, pp. 2758–2766.

[7] Bertram Drost, Markus Ulrich, Nassir Navab, and Slobodan Ilic. "Model globally, match locally: efficient and robust 3D object recognition". In: *Computer vision and pattern recognition (CVPR), 2010 IEEE conference on*. Ieee. 2010, pp. 998–1005.

[8] Richard O Duda and Peter E Hart. *Use of the hough transformation to detect lines and curves in pictures*. Tech. rep. SRI International Menlo Park CA artificial intelligence center, 1971.

[9] Martin A Fischler and Robert C Bolles. "Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography". In: *Communications of the ACM* 24.6 (1981), pp. 381–395.

[10] Ross Girshick. "Fast R-CNN". In: *Arxiv preprint arxiv:1504.08083* (2015).

[11] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. "Rich feature hierarchies for accurate object detection and semantic segmentation". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2014, pp. 580–587.

[12] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. `http://www.deeplearningbook.org`. MIT Press, 2016.

[13] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross B. Girshick. "Mask R-CNN". In: *CoRR* abs/1703.06870 (2017). arXiv: `1703.06870`. URL: `http://arxiv.org/abs/1703.06870`.

[14] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[15] Stefan Hinterstoisser, Cedric Cagniart, Slobodan Ilic, Peter Sturm, Nassir Navab, Pascal Fua, and Vincent Lepetit. "Gradient response maps for real-time detection of textureless objects". In: *IEEE transactions on pattern analysis and machine intelligence* 34.5 (2012), pp. 876–888.

[16] Stefan Hinterstoisser, Vincent Lepetit, Slobodan Ilic, Stefan Holzer, Gary Bradski, Kurt Konolige, and Nassir Navab. "Model based training, detection and pose estimation of texture-less 3d objects in heavily cluttered scenes". In: *Asian conference on computer vision*. Springer. 2012, pp. 548–562.

[17] Stefan Hinterstoisser, Vincent Lepetit, Naresh Rajkumar, and Kurt Konolige. "Going further with point pair features". In: *European conference on computer vision*. Springer. 2016, pp. 834–848.

[18] Omid Hosseini Jafari, Siva Karthik Mustikovela, Karl Pertsch, Eric Brachmann, and Carsten Rother. "IPose: instance-aware 6D pose estimation of partly occluded objects". In: *CoRR abs/1712.01924* (2017).

[19] Omid Hosseini Jafari, Siva Karthik Mustikovela, Karl Pertsch, Eric Brachmann, and Carsten Rother. "The best of both worlds: learning geometry-based 6D object pose estimation". In: *CoRR* abs/1712.01924 (2017). arXiv: `1712.01924`. URL: `http://arxiv.org/abs/1712.01924`.

[20] Alexander Krull, Eric Brachmann, Frank Michel, Michael Ying Yang, Stefan Gumhold, and Carsten Rother. "Learning analysis-by-synthesis for 6D pose estimation in RGB-D images". In: *Proceedings of the IEEE international conference on computer vision*. 2015, pp. 954–962.

[21] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. "Numba: a LLVM-based python JIT compiler". In: *Proceedings of the second workshop on the LLVM compiler infrastructure in hpc*. LLVM '15. Austin, Texas: ACM, 2015, 7:1–7:6. ISBN: 978-1-4503-4005-2. URL: `http://doi.acm.org/10.1145/2833157.2833162`.

[22] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. "Deep learning". In: *Nature* 521.7553 (2015), pp. 436–444.

[23] Vincent Lepetit, Francesc Moreno-Noguer, and P Fua. "EPnP: efficient perspective-n-point camera pose estimation". In: *International journal of computer vision* 81.2 (2009), pp. 155–166.

[24] Yi Li, Gu Wang, Xiangyang Ji, Yu Xiang, and Dieter Fox. "DeepIM: deep iterative matching for 6D pose estimation". In: *Arxiv preprint arxiv:1804.00175* (2018).

[25] Jonathan Long, Evan Shelhamer, and Trevor Darrell. "Fully convolutional networks for semantic segmentation". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2015, pp. 3431–3440.

[26] David G Lowe. "Distinctive image features from scale-invariant keypoints". In: *International journal of computer vision* 60.2 (2004), pp. 91–110.

[27] F Landis Markley, Yang Cheng, John L Crassidis, and Yaakov Oshman. "Quaternion averaging". In: (2007).

[28] Markus Oberweger, Mahdi Rad, and Vincent Lepetit. "Making deep heatmaps robust to partial occlusions for 3D object pose estimation". In: *CoRR* abs/1804.03959 (2018). arXiv: 1804.03959. URL: http://arxiv.org/abs/1804.03959.

[29] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. "Automatic differentiation in pytorch". In: *NIPS-W*. 2017.

[30] François Pomerleau, Francis Colas, Roland Siegwart, and Stéphane Magnenat. "Comparing ICP variants on real-world data sets". In: *Autonomous robots* 34.3 (2013), pp. 133–148.

[31] Simon JD Prince. *Computer vision: models, learning, and inference*. Cambridge University Press, 2012.

[32] Mahdi Rad and Vincent Lepetit. "BB8: a scalable, accurate, robust to partial occlusion method for predicting the 3D poses of challenging objects without using depth". In: *2017 IEEE international conference on computer vision (ICCV)* (Oct. 2017). URL: http://dx.doi.org/10.1109/ICCV.2017.413.

[33] Joseph Redmon and Ali Farhadi. "YOLO9000: better, faster, stronger". In: *Arxiv preprint* (2017).

[34] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. "Faster R-CNN: towards real-time object detection with region proposal networks". In: *Advances in neural information processing systems*. 2015, pp. 91–99.

[35] Sam Roweis. "Levenberg-marquardt optimization". In: *Notes, university of toronto* (1996).

[36] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. "ImageNet Large Scale Visual Recognition Challenge". In: *International journal of computer vision (IJCV)* 115.3 (2015), pp. 211–252.

[37] Max Schwarz, Christian Lenz, Germán Martın Garcıa, Seongyong Koo, Arul Selvam Periyasamy, Michael Schreiber, and Sven Behnke. "Fast object learning and dual-arm coordination for cluttered stowing, picking, and packing". In: *IEEE international conference on robotics and automation (ICRA)*. 2018.

[38] Karen Simonyan and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition". In: *Arxiv preprint arxiv:1409.1556* (2014).

[39] Richard Szeliski. *Computer vision: algorithms and applications*. Springer Science & Business Media, 2010.

[40] Bugra Tekin, Sudipta N. Sinha, and Pascal Fua. "Real-time seamless single shot 6D object pose prediction". In: *CoRR* abs/1711.08848 (2017). arXiv: 1711.08848. URL: http://arxiv.org/abs/1711.08848.

[41] Jonathan Tremblay, Thang To, Balakumar Sundaralingam, Yu Xiang, Dieter Fox, and Stan Birchfield. *Deep object pose estimation for semantic robotic grasping of household objects*. 2018. arXiv: 1809.10790 [cs.RO].

[42] John Vince. *Quaternions for computer graphics*. Springer Science & Business Media, 2011.

[43] Daniel Wagner, Gerhard Reitmayr, Alessandro Mulloni, Tom Drummond, and Dieter Schmalstieg. "Pose tracking from natural features on mobile phones". In: *Proceedings of the 7th ieee/acm international symposium on mixed and augmented reality*. IEEE Computer Society. 2008, pp. 125–134.

[44] Jay M Wong, Vincent Kee, Tiffany Le, Syler Wagner, Gian-Luca Mariottini, Abraham Schneider, Lei Hamilton, Rahul Chipalkatty, Mitchell Hebert, David Johnson, et al. "SegICP: integrated deep semantic segmentation and pose estimation". In: *Arxiv preprint arxiv:1703.01661* (2017).

[45] Yu Xiang, Wonhui Kim, Wei Chen, Jingwei Ji, Christopher Choy, Hao Su, Roozbeh Mottaghi, Leonidas Guibas, and Silvio Savarese. "ObjectNet3D: a large scale database for 3D object recognition". In: *European conference computer vision (ECCV)*. 2016.

[46] Yu Xiang, Tanner Schmidt, Venkatraman Narayanan, and Dieter Fox. "PoseCNN: a convolutional neural network for 6D object pose estimation in cluttered scenes". In: *Arxiv preprint arxiv:1711.00199v2* (2018).

[47] Jianxiong Xiao, James Hays, Krista A Ehinger, Aude Oliva, and Antonio Torralba. "Sun database: large-scale scene recognition from abbey to zoo". In: *Computer vision and pattern recognition (CVPR), 2010 IEEE conference on*. IEEE. 2010, pp. 3485–3492.

[48] Matthew D Zeiler and Rob Fergus. "Visualizing and understanding convolutional networks". In: *European conference on computer vision*. Springer. 2014, pp. 818–833.