Freie Universität Berlin

Bachelorarbeit am Institut für Informatik der Freien Universität Berlin

Pose Estimation of a Cuboid in a Controlled Environment with Deep Convolutional Neural Networks for an Industrial Application

Amélie Froessl
Matrikelnummer: 4855540
amelief@zedat.fu-berlin.de

Betreuer:	Dr. Cristian Grozea
Erstgutachter:	Prof. Dr. Raúl Rojas
Zweitgutachter:	Prof. Dr. Tim Landgraf

Berlin October 5, 2017

Abstract

Object pose estimation is a common computer vision problem. The recent trend in augmented reality applications, as well as applying deep convolutional neural networks for computer vision problems was the original motivation for this thesis. Specifically, this thesis tries to explore the possibilities of using a deep convolutional neural network trained on rendered training data to estimate the pose of a cube in an industrial application.

The mentioned industrial application is to be realized in the context of a project at the Fraunhofer FOKUS. In an industrial like setting, the location and rotation of a cube on a workbench is to be predicted by a trained neural network given an RGB image. With the exact pose of the object an augmented reality experience for a user can be realized to interact with the cube. More importantly, a robot within the workspace can use this information to further process the cube.

The trained convolutional neural network in this thesis generalizes well over the pose of a cuboid when tested on rendered images. Some approaches are discussed as an outlook to bridging the gap between predicting on rendered images and real images at the end of this thesis.

Eidesstattliche Erklärung

Ich versichere hiermit an Eides statt, dass diese Arbeit von niemand anderem als meiner Person verfasst worden ist. Alle verwendeten Hilfsmittel wie Berichte, Bücher, Internetseiten oder ähnliches sind im Literaturverzeichnis angegeben, Zitate aus fremden Arbeiten sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

October 5, 2017

Amélie Froessl

Acknowledgements

I would like to thank Dr. Cristian Grozea at the Fraunhofer FOKUS for supervising me during my thesis. He patiently explained many concepts to me which lead to my deeper understanding of the theory. The completion of this thesis would not have been possible without him.

I would also like to thank Prof. Dr. Raúl Rojas of the Freie University of Berlin for agreeing to be my thesis advisor.

Lastly, I would like to express my gratitude to my family for their continuous support throughout my studies, especially my parents for proofreading my thesis and their valuable comments.

Contents

1 Introduction **1**
1.1 Problem Statement . . . 1
1.2 Solution Approach . . . 2

2 Related Work **3**
2.1 Hara et al. . . . 3
2.2 M. Rad and V. Lepetit . . . 4

3 Methods Used **6**
3.1 Neural Network Architectures . . . 6
3.2 Network Training . . . 9
3.3 Difficulties of Rotational Characteristics . . . 11

4 Familiarization with the Problem **12**

5 Designing the Network Architecture **16**
5.1 Initializing the Convolutional Neural Network Architecture . . . 16
5.2 Rotation of a Cube . . . 17
5.3 Realistic Scenario . . . 20
5.4 Tuning for Rotational Accuracy . . . 23

6 Final Network Architecture and Results **26**

7 Outlook and Conclusion **31**

References **33**

1 Introduction

1.1 Problem Statement

A deep convolutional neural network (DCNN) is proposed in this thesis to extract the pose of a cube to be used in a specific application at the Fraunhofer FOKUS's Visual Computing (VISCOM) business unit[1]. To give some context, and to specify the problem domain further, a possible use case to clarify the motivation of this thesis is described.

A user as well as a robot interact with a cube, positioned on a workbench within a predefined area. The robot's task is to smooth out the edges of the cube. The user chooses an edge by pointing to it on the cube. With the position of the user's finger as well as the cube's current pose, the edge will be highlighted with projectors. The robot then smoothes out the indicated edge.

The scene will have industrial style controlled lighting. Slight shadows in arbitrary directions can be expected, however no outdoor or extreme lighting conditions will be present. The cube's exact shape and size will be known (CAD information available). Its surface is a non-glossy, mate surface. Multiple calibrated cameras will be present to record the scene, however for this thesis only one RGB image was used. Also, the cameras are regular RGB cameras without depth sensors. Only the cube will be located within the predefined scene, therefore the object itself will not be occluded by other objects. Also, no background clutter will be seen in the image.

[1]https://www.fokus.fraunhofer.de/go/viscom

1.2 Solution Approach

The work in this thesis explores the possibility of using machine learning for the given problem, in the light of the recent trend for using DCNNs for solving computer vision tasks.

The proposed approach is to construct a single DCNN using regression, without data preprocessing and use supervised training with synthetically created RGB images modeling the cube in the previously defined setting. The trained network is then to be used to predict the pose of the cube from a real image captured in the actual industrial environment.

To do this, we can exploit some advantages of our simple and controlled scene for the task. A prerequisite for deep convolutional neural networks, with many parameters, is a big amount of data to train on. Creating synthetic images for such a simple scene is a relatively easy task if compared to settings with no control over lighting, background, multiple or complex objects, etc. No random object occlusion or clutter needs to be accounted for either. Such simple scenes do not take long to render, therefore a big amount of data can be created in a relatively short amount of time. Also, manually creating data would risk measurement inaccuracies that would reflect in the predictions of the network.

In order to achieve good predictions of an object's pose, a good fit must be found between training data, their labels and the network architecture. In this case, a good fit would be characterized by a network, which is able to extract the necessary features from the training data needed to make a good generalization of an object's location and rotation. The difficulty lies in constructing the architecture of the network and the labeled training data.

To define a structure and process for the thesis, simpler versions of the problem were first experimented with to gain a better feel for the network behavior. This was done by training simple networks on small black and white images at first. Later, tested hyperparameters for vision tasks were introduced for an initial network architecture as well as training images created using Blender [2]. This network architecture was then tuned to predict the pose of a cuboid seen in synthetic images created with Blender.

2 Related Work

The most recent machine learning approaches for object pose estimation utilize the benefits of additional information from depth sensors. This is especially useful for featureless objects, similar to the objects in this thesis. Unfortunately, no depth cameras will be available in the foreseen application. Therefore, the approaches described in the paper published by Hara et al. *'Designing Deep Convolutional Neural Networks for Continuous Object Orientation Estimation'* [5] and the approach in M. Rad and V. Lepetit 's paper *'BB8: A Scalable, Accurate, Robust to Partial Occlusion Method for Predicting the 3D Poses of Challenging Objects without Using Depth'* [10] are the ones most drawn from for inspiration for this thesis.

2.1 Hara et al.

Designing Deep Convolutional Neural Networks for Continuous Object Orientation Estimation

Hara et al.'s paper [5] describes a solution to estimate the continuous estimation of an arbitrary object using DCNNs. The definition of orientation for an object in the paper is identical to the definition of rotation for the cube around the z-axis used in this thesis. Although the paper only focuses on the orientation of an object whereas this thesis considers the rotation and location of an object, the network training is similar in respects to the label representation and the training method used. The paper describes three different approaches, two of which use regression and one which uses classification to estimate orientation.

The first approach described in the paper is similar to the approach in this thesis in multiple ways. The network uses regression to prediction the orientation of the object. Also, the function used for optimization is the L2-loss function, which calculates the squared difference between the predictions and the ground truth of the orientation, similar to the mean squared error used in this thesis. Not only is the optimization method similar but more importantly the label representation as coordinates on the unit circle instead of angular measure for the orientation is equivalent to the representation of rotation in this thesis.

However, it has its differences as well. In the paper only orientation is estimated, not the location of the object. Furthermore, the paper describes using only available real-world images from an open source dataset to train their networks, whereas this thesis uses specifically generated training data. The disadvantage of such real-world image datasets is the limited amount of labeled training data available. They solve this problem by using a pre-trained network for an image classification task, and fine-tuning it for the orientation task. Therefore, they can use the advantages of a deep network without having to create the immense amount of data needed to train such a network.

The important part from the paper which is used in this thesis is the usage of the coordinates on a unit circle for representation for rotation, having the advantage of being a continuous representation for orientation.

2.2 M. Rad and V. Lepetit

BB8: A Scalable, Accurate, Robust to Partial Occlusion Method for Predicting the 3D Poses of Challenging Objects without Using Depth

Unlike many recent state of the art approaches of pose estimation using machine learning, M. Rad and V. Lepetit 's paper [10] describes a solution which solves object localization and pose prediction of an object, without depth information.

To do this, the problem is divided into two separate steps, solving localization and then solving rotation. They first use two networks to estimate the location of the object. Then, the projection of the objects bounding box corners is estimated. Finally, they obtain the 3D coordinates of the bounding box by utilizing a perspective n-point algorithm. The reason for the bigger effort in object localization is due to the fact that they also allow for partial occlusion with other objects.

The problem addressed in the paper also faced in this thesis, is the problem of rotational symmetry. They focus on objects within the T-LESS dataset [6], which are mostly manufactured objects with rotational symmetry. Their solution restricts the range of rotation to the angle of rotational symmetry, so that the network is only trained on images in which a difference in label translates to a difference in the image. However, the image of an object rotated close to $0°$ still resembles the image of the same object rotated close to β degrees for an object with rotational symmetry which repeats every β degrees. Therefore, they split the region between $0°$ and β into half and use a classification network to predict in which region the rotation is. If the rotation falls into the second region, the input is mirrored.

A cube also has rotational symmetry, which allows to use the same method of restricting the rotation in the approach taken in this thesis. However, instead of using degrees the final configuration of the network in this thesis predicts the angle using the coordinates on a unit circle as a representation of rotation, as described in the previously discussed paper published by Hara et al.

In order to improve the accuracy of the estimated pose, M. Rad and V. Lepetit are using an additional network. This network gets as input the original image segment, as well as a color render or binary mask of the object from the already estimated pose and trains to optimize the difference in pose of the two images.

While fine tuning the network for this thesis, when the accuracy of the output was not in the desired range, the approach of using a second network predicting the difference in pose was tried as well. Simultaneously other more promising possibilities were explored therefore this approach was not further followed.

3 Methods Used

The explanations, figures and equations in this section are mostly derived from Micheal Nielson's Book *'Neural Networks and Deep Learning'* [9] and the lecture notes accompanying the Stanford CS class *'CS231n Convolutional Neural Networks for Visual Recognition'*[13].

3.1 Neural Network Architectures

Neural networks are models for data processing consisting of interconnected elements, also called neurons, arranged in layers. Typically, there exists an input layer, an arbitrary number of hidden layers and an output layer, figure (1).

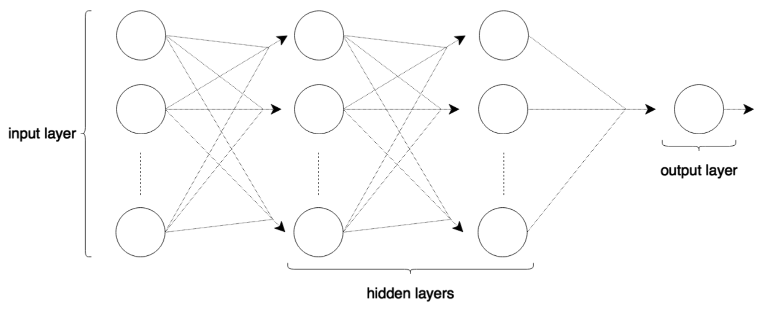


Figure 1: A simplified example of a network architecture.

Figure (2) shows us a closer view of a neuron in the hidden layers of the network. Each neuron has multiple input values $a_1 ... a_j$, weights w_{ij} multiplied with these values, a bias b_i added to the dot product of the weights and input values and lastly an activation function σ, equation (1).

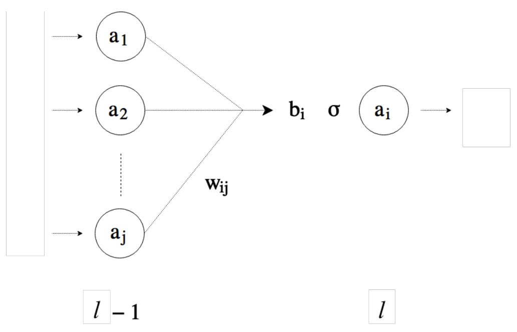


Figure 2: A neuron with input values $a_1 ... a_j$, an output value a_i weights w_{ij}, a bias b_i, an activation function σ and layer index l

$$a_i^l = \sigma(\sum^j w_{ij}^l a_j^{l-1} + b_i^l) \tag{1}$$

The weights and biases are the learnable parameters of the layers which are updated during the network training process. The weights will determine the importance of a given input and the bias will determines a threshold for the artificial neuron to activate. The activation function adds the elementwise non-linearity to the network.

A fully connected layer is a layer connecting each neuron in layer $l-1$ to each neuron in layer l. If all layers in a network are fully connected the network is called a dense network. A very large amount of parameters would be needed to process an image with a dense network, given the many pixels an image has. To avoid this, convolutional neural network architectures make use of convolutional and pooling layers to reduce the amount of parameters needed to process the input.

The convolutional layers implement a method similar to many image processing tasks, which convolve a filter in the form of a kernel over the image, to extract certain features within an image. An example of a filter used for image processing tasks is the Sobel filter [11], which is used to extracts the edges within an image. The difference being that the kernel for the Sobel filter is predefined whereas the values of the kernel for the convolutional layer correspond to that layer's learnable parameters and must be learned during the training process.

A single kernel has dimensions (i x j), which slides over the input (n x m x d), with a stride, or step size k, always returning the dot product of the values in the kernel and the values of the image. The yellow square in figure (3) represents the kernel sliding over the image input with stride k. The result is called a feature map. Multiple kernels in each convolutional layer, allow to extract multiple feature maps from a single input.

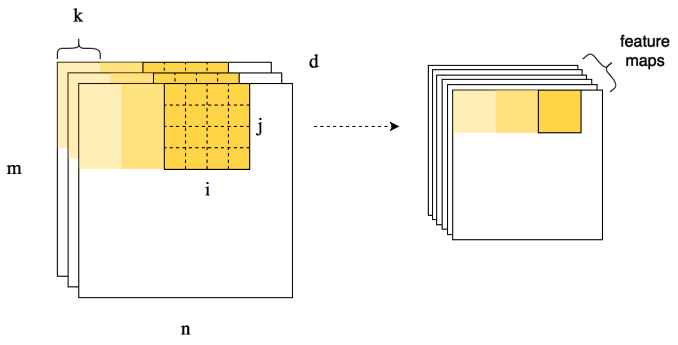


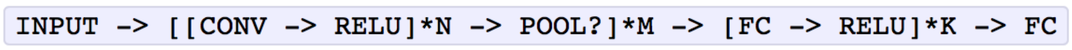
Figure 3: An illustration of convolution

Pooling layers are similar to convolutional layers in the way that they slide a kernel with certain dimensions and stride over the input. However, these layers do not have any learnable parameters, they simply reduce the amount of data needing to be processed by the next layer.

The difficulty with constructing a network from scratch is picking the right architecture or hyperparameters for the network. Hyperparameters are all the variables set before training such as the type of layers within a network, the amount of layers and their dimensions, the activation functions used between layers, the batch sizes for training, etc. Hyperparameter tuning refers to experimenting with different variables for the hyperparameters, such that a network is able to infer the function relating the inputs to their target outputs.

The consensus about hyperparameter tuning in the scientific community is that it is still much more an art rather than an exact science. This is due to the complexity of the networks through the huge amount of trainable parameters within the network and the vast amount of possibilities to choosing hyperparameters. Therefore, guidelines suggested by the Stanford course on 'Convolutional Neural Networks for Visual Recognition' [13] were used for constructing the convolutional neural network in this thesis. The course discusses many aspects of neural networks, therefore only the relevant ones used in this thesis will be shortly reviewed.

The course lists a couple common convolutional network architectures implementing a version of the layer pattern seen in figure (4).



```
INPUT -> [[CONV -> RELU]*N -> POOL?]*M -> [FC -> RELU]*K -> FC
```

Figure 4: A general layer pattern seen often in convolutional neural networks. Source: `https://goo.gl/uka6ZR`

This layer pattern describes an architecture of an input layer, M sets of layers following a pattern of convolution, activation with the rectified linear unit function (ReLU) and possibly followed by a pooling layer. Towards the end, the pattern changes to K sets of fully connected layers and ReLU activations followed by a single fully connected layer. The architecture in this thesis only uses this pattern with one convolutional layer in each set (N set to 1). However, the number of sets (M) varied throughout the process of finding the final network architecture. Also, the number of fully connected layers (K) varied until the final number was found for the end architecture.

An example of an exact convolutional architecture in the course increases the amount of feature maps extracted at each following convolutional layer by two. Even though this is not explicitly mentioned in the course, it is often seen for known convolutional architectures and used as a point of reference in this thesis.

They also suggest using ReLU, figure (5), as an element-wise activation function between the layers.

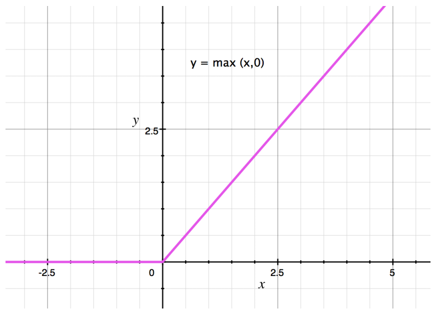


Figure 5: Rectified Linear Unit activation function

The pooling layers are recommended to have a kernel of size (2x2) with a stride of 2. The reason for the small kernel size mentioned in the course is that larger sizes result in too much data loss. The type of pooling used is 'maxpooling', which selects the maximum value for each region it covers.

The recommended kernel size for the convolutional filter is a kernel size of of (3x3) and a stride of 1. In order to have more control over the output dimensions of a convolutional layer, the course recommends using zero-padding.

Knowing that the hyperparameters recommended in the course are used in other convolutional networks solving computer vision tasks, helped narrow down hyperparameters which needed tuning.

3.2 Network Training

The goal is to train a deep convolutional neural network such that it can predict continuous values representing the 3D pose of an object seen in a 2D image. The network in this thesis was implemented using Keras [3] with a Tensorflow [1] backend and trained using NVIDIA's GeForce GTX 1080[2].

[2]https://goo.gl/DB9WHT

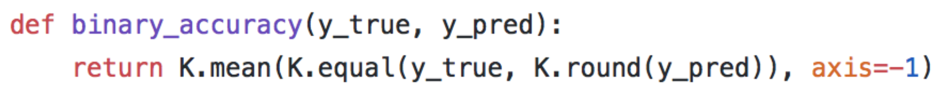
The 'Adam' optimizer [8], a method for stochastic optimization, was used to train the network. The aim of the training process is for the network to find a mapping relating the input values (images) to their corresponding labels (pose), by minimizing a loss function, such as the mean squared error used in this thesis, equation (2).

$$C = 1/n \sum_{i=1}^{n} (y_i - a_i)^2 \tag{2}$$

The optimization algorithm implements an iterative process to minimizing the loss function. In each iteration, predictions a_i are made by the current state of the network for each image in a batch of training data consisting of n images. The error is then calculated between the predictions a_i and the desired labels y_i, or target outputs, with the loss function. Lastly, the state of the network is updated, by taking a step in the negative direction of the gradient at the point on the loss function representing the earlier calculated error.

Considering that the inputs and the target outputs are constants to the loss function and the weights and biases of the network are variables, each weight and bias must be updated by taking a step in the negative direction of their respective gradient in order to take a step towards the minimum of the loss function. Backpropagation is the method used to calculating the gradient, or the partial derivative of the loss function in respects to each weight and bias of the network.

The quality of the network is monitored during and after training in order to tell if the network is approaching a minimum or has found a minimum loss. The metric used for evaluation of the network's quality used in this thesis, was Keras' 'binary accuracy' metric, figure (6). This metric calculates the percentage of the correctly predicted values over a batch of training images. This is done by rounding the network's predictions and comparing it to the target outputs. K.equal returns the element-wise equality between two tensors as boolean values and K.mean casts these values to floats and returns the mean of that tensor[3]. A regression model generally will not predict precisely the same target outputs, therefore the discretization produced by rounding enables one to check for perfect identity and count the amount of inputs predicted closely enough.



```
def binary_accuracy(y_true, y_pred):
    return K.mean(K.equal(y_true, K.round(y_pred)), axis=-1)
```

Figure 6: Keras' binary accuracy metric.
Source: `https://goo.gl/4fP9Y8`

[3]https://goo.gl/gygZ2v

3.3 Difficulties of Rotational Characteristics

When training a neural network to predict the pose of an object from an image, the training data with their corresponding labels should be unambiguous. Two characteristics of rotation hinder this unambiguity, which had to be considered when creating training data. One of these characteristics is the rotational symmetry of an object and the other is the linear representation of degrees.

Rotational symmetry of an object refers to the characteristic of an object, if when rotated a certain number of degrees β it looks exactly the same as it did before the rotation. For example, a square exhibits this behavior when rotated in intervals of 90°. A cube has a rotational symmetry for the same interval, but for the rotation around the x,y and z-axis. A network will have difficulties learning a representation of rotation if our training data consists of identical images labeled with different rotations.

The linear representation of degrees handicaps the network training as well, not because different labels are used to describe two indistinguishable images, but the differences in value of the labels do not correspond to the visual difference of the image. An image of an object rotated at 0° and one rotated at 395° are visually more similar, but their label difference is bigger than two images, one of which shows the object rotated at 0° and one rotated at 180°. For objects with rotational symmetry this not only applies to the full rotational range, but each rotational range comprised between multiples of β.

4 Familiarization with the Problem

There are no set instructions or predefined processes to designing a neural network. Many convolutional neural networks in literature are trained for object classification or detection, not many examples of convolutional neural networks predicting object pose estimation without using additional depth information exist. Under these circumstances, first experiments with simpler networks, problem segments and simpler images were done to observe initial network behavior when training to predict pose estimation from an RGB image. Simplifying the problem, the network and predicting on smaller images reduces the amount of learnable parameters ergo reducing training time making it possible to quickly prototype and test. The following subsections describe experiments done on simpler networks and with small black and white images of squares created using the OpenCV library [7].

The simplest definition for location used, was the location of a randomly located white pixel in an image of dimensions (1x8), figure (7).

Figure 7: An enlarged image of 8 pixels with 1 random white pixel

Theoretically, a single neuron with 8 input values and 1 output value could be used to predict the location of the white pixel. The inputs would be the values of the pixels, 0 for black and 1 for white and the weights connecting each input to the output would be the index of the given input, figure (8).

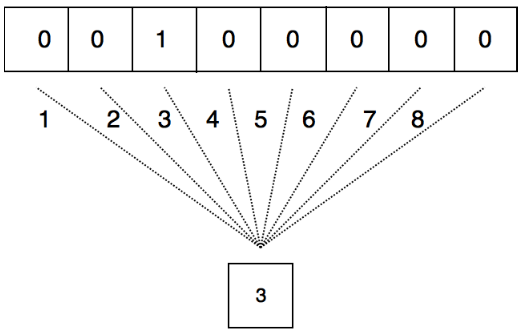


Figure 8: A theoretical artificial neuron consisting of 8 input values, 1 output value and the corresponding weights to predict the location of a randomly located white pixel. In this image, the white pixel is located at index 3.

Trials with a dense network reflecting the artificial neuron in figure (8) consisting of only one input and one output layer with 8 input values and 1 output value showed no positive results. Thereupon, the network capacity was increased by adding two hidden layers with 4 and 2 values respectively. The increased capacity produced a network capable of correctly predicting the pixel location after a couple of training iterations, for which the network was fed 10 images in each iteration.

The next couple of scenarios describe tests using images of dimensions (90x90), a significant increase to the earlier (8x1) dimension. As discussed earlier dense networks are not ideal for processing images. To adjust the network to better handle the bigger images, the dense network was exchanged for a shallow convolutional neural network. The network consisted of an input layer reflecting the image dimensions, pairs of convolutional and pooling layers (either one or two) followed by one or two dense layers. For the experiments described for the rest of section (4) a wide variety of hyperparameters were experimented with, but since these experiments were only used to gain a feeling for the network's behavior, the exact hyperparameters of the network and exact results are omitted for this section. A more detailed and tested network architecture, constructed using the guidelines discussed in section (3.1) will be introduces in the next section.

Images depicting a randomly located and scaled square, such as the ones in figure (9) were used as training data to get a feeling for a network's performance with an increased complexity of location. The network was trained to predict three labels representing the location of the square, instead of one label. The first two labels were the x and y coordinates of the square's center and the third was the scale of the square.

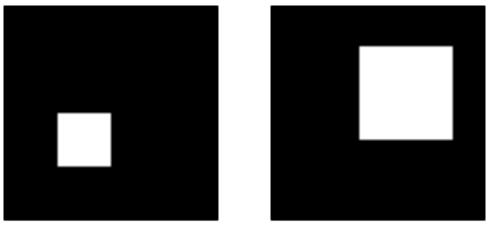


Figure 9: Sample images of a square centered on a randomly located pixel (x,y) and randomly scaled.

Again, the x and y coordinates were randomly chosen to be a pixel in the image and the scale was randomly chosen from a predefined range of scale. The reason the network was trained to predict the x,y coordinates as well as scale was to model more closely the location of an object in a coordinate system with three axes, figure (10). The x,y coordinates of the square's center correspond to the location of the square in the x,y-plane of the coordinate system and the square's scale correspond to the translation of the square along the z-axis in the coordinate system, considering the z-axis is perpendicular to the image plane.

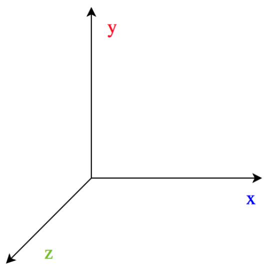


Figure 10: The coordinate system with x,y and z-axes

To examine the network's behavior when predicting the rotation of an object, simple experiments were done using images with the same dimensions (90x90) but depicting a rotated square, figure (11). For these trials the network was trained to predict this square's rotation, which was always located perfectly in the center of the image. The rotation was restricted to angles between 0° and 90°, to avoid confusing the network given a square's rotational symmetry, as discussed in section (3.3).

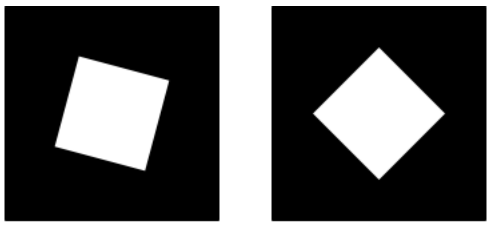
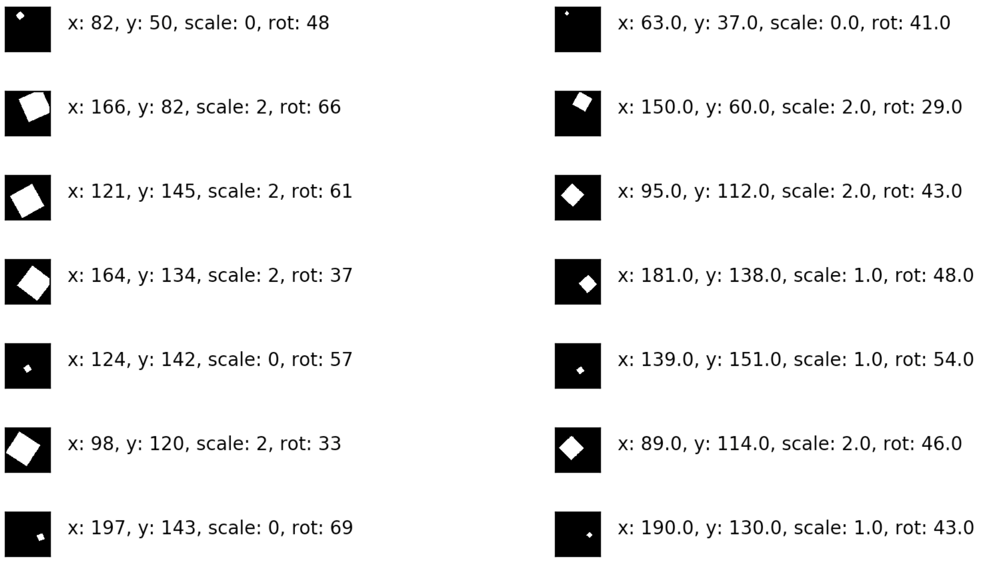


Figure 11: Sample images of a randomly rotated square

Some simple tests were also done to see how well a network performs when trained on images with a slightly off-center square compared to one always located perfectly in the center of the image. Training the network to predict the rotation for a square slightly off-center resulted in less accurate predictions than when trained on images with a square located perfectly in the center of the image. This was to be expected since this is a more complex problem.

The combined problem of rotation and location was experimented with by training the network to predict the location and rotation of a square for images such as the ones in the left column of figure (12). The x, y coordinates, the scale and the rotation of the square were chosen as in the previous scenarios. The square in the image was rotated randomly between 0° and 90°, centered on a randomly located pixel with x,y position and randomly scaled. The figure visually illustrates the performance of the network.

The original images with the target outputs x, y, scale and rotation are seen in the left column of the figure. The x, y, scale and rotation values next to the images on the right are the corresponding predictions of the network for the original images in the left column. The images in the right column are then drawn using the predictions to give a rough visual indication of the networks performance. The predicted values are rounded to the nearest integer, however the images are drawn using the unrounded predictions of the network.



x: 82, y: 50, scale: 0, rot: 48	x: 63.0, y: 37.0, scale: 0.0, rot: 41.0
x: 166, y: 82, scale: 2, rot: 66	x: 150.0, y: 60.0, scale: 2.0, rot: 29.0
x: 121, y: 145, scale: 2, rot: 61	x: 95.0, y: 112.0, scale: 2.0, rot: 43.0
x: 164, y: 134, scale: 2, rot: 37	x: 181.0, y: 138.0, scale: 1.0, rot: 48.0
x: 124, y: 142, scale: 0, rot: 57	x: 139.0, y: 151.0, scale: 1.0, rot: 54.0
x: 98, y: 120, scale: 2, rot: 33	x: 89.0, y: 114.0, scale: 2.0, rot: 46.0
x: 197, y: 143, scale: 0, rot: 69	x: 190.0, y: 130.0, scale: 1.0, rot: 43.0

Figure 12: The left column shows the original images with their corresponding ground truth labels. The right column shows the predicted numerical values and the redrawn images of squares using the predictions.

5 Designing the Network Architecture

The experiments described in the previous section helped gain a better understanding of a network's behavior, by predicting a square's pose, a simplified version of predicting a cube's pose. However, for the application an estimation of a cube's pose is needed. To start the process of tuning the network to be usable in the end application, two changes were made from the previous simple experiments. First, training images were no longer created on the fly with the OpenCV library, but rendered using Blender [2], an open source 3D rendering software. Second, a network architecture was constructed using the Stanford guidelines discussed in section (3.1), as an initial starting point.

Subsection (5.1) introduces the convolutional neural network architecture used and improved upon throughout the following subsections, eventually leading up to the final network architecture. The succeeding subsection (5.2) discusses important design decisions made while training the network to predict the cube's rotation. Section (5.3) then focuses on narrowing down the problem to more closely fit for the application. Tuning the network to more accurately predict a cube's pose is covered in section (5.4) and lastly, the final network architecture is summarized in section (6). The process of constructing and tuning the convolutional neural network is illustrated through key design decisions and how they were driven by the results obtained from experiments.

5.1 Initializing the Convolutional Neural Network Architecture

A rough outline of the initial network architecture serves to better demonstrate the changes discussed in the next section leading to the final network architecture. Hyperparameters for the architecture initially set and held fixed were taken from the Stanford course guidelines, discussed in section (3.1). These included the general layer pattern, the specifications for the convolutional and pooling layers and the activation function. Hyperparameters independently chosen from the course and slightly changed throughout the process of constructing the final network were number of layers, input image dimensions, exact number of feature maps extracted at each layer, number of training iterations as well as batch sizes during training.

Starting off the network architecture consisted of an input layer whose dimensions reflected the initial image size (144x128), 4 sets of layers each set consisting of a convolution layer, ReLU activation and a pooling layer and finally two trailing dense layers. The first convolutional layer extracted 8 feature maps and the number of feature maps increased by 2 for each following convolutional layer (8,16, 32, 64). The first dense layer comprised 32 neurons and the last dense layer reflected the dimension of the label vector, for example 3 neurons if the network was to predict the translation of the object along the x,y- and z-axis. They next sections cover key milestones in the evolution of the hyperparameters and training data used for the final network architecture. Changes made which had smaller effects are omitted for simplicity.

5.2 Rotation of a Cube

While experimenting with the simpler versions of the problem it proved easier to find a network configuration which could confidently predict the location of a square than one predicting its rotation. Network predictions for the rotation of a square seemed to fluctuate more heavily under different hyperparameter settings versus the predictions of location. These facts and the fact that rotation of an object in 3D is more complex given the increased number of possible axes to rotate around, were the reasons for first tackling the rotation of a cube in 3D and later adding the location.

In our earlier scenarios of training on images of a square, the origin of our coordinate system was the upper left corner of the image and the x and y coordinates corresponded to the index of the pixel in relation to the width and height of the image. In this scenario the width and height of the image no longer translate directly to the x- and y-axis. Instead, the origin of the coordinate system is located at the center of the cube and the image becomes the projection of the cube onto the image plane. The object can be rotated around each axis.

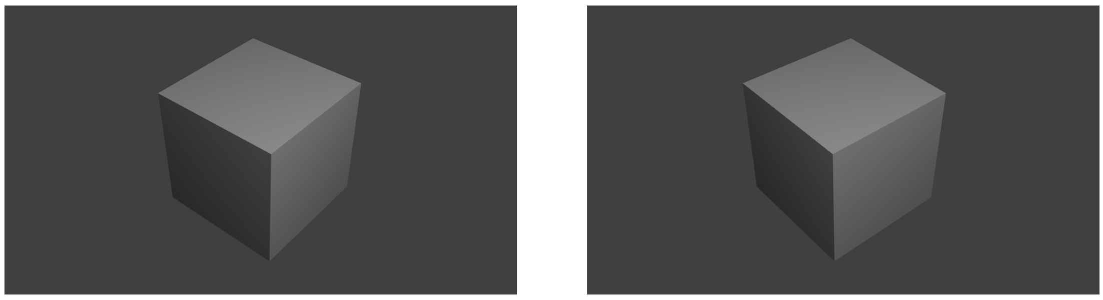


Figure 13: Rendered images of a cube rotated around its x- and y-axes. Rotation around y-axis in the left image differs from the rotation in the right image by 10°

Figure (13) shows an example of one of the images used at this stage. Images were rendered, such that each image represents a different rotation of the cube around the x,y- and z-axis in the range of 0°- 45°, with an interval of 5°. The restricted range of rotation was to avoid confusing the network, given the rotational symmetry of a cube as discussed in section (3.3).

The problem encountered with this configuration was the unpredictable behavior of the network. Depending on the training session the network performed very differently. After some training sessions the learned representation of rotation was much better than after others. In this case the representation of rotation was the rotation of the cube around the x,y- and z-axis. Analyzing the histograms below help to explain the problem. Each histogram shows an evaluation of a trained network after a training session. The evaluation is done over 1000 test images, in which the angle difference is calculated as the difference between the original label and the predicted label for each image.

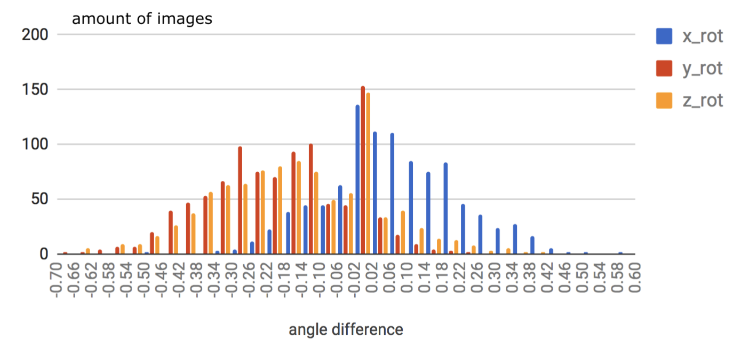


Figure 14: The network learns to predict the rotation around all axes

Figure (14) shows a lucky training session, in which the network learns to predict the rotation around all axes.

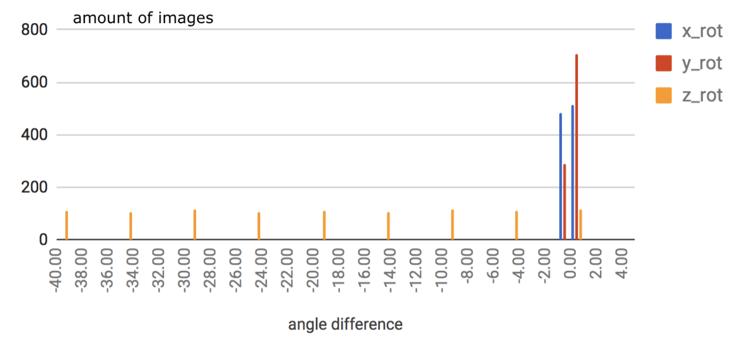


Figure 15: The network learns to predict the rotation around the x- and y-axes

Figure (15) shows the evaluation after a different training session. The network only accurately predicts the cube's rotation around the x-axis and the y-axis. The evenly spaced angle differences for the rotation about the z-axis reflect the angle intervals at which the test images were rendered.

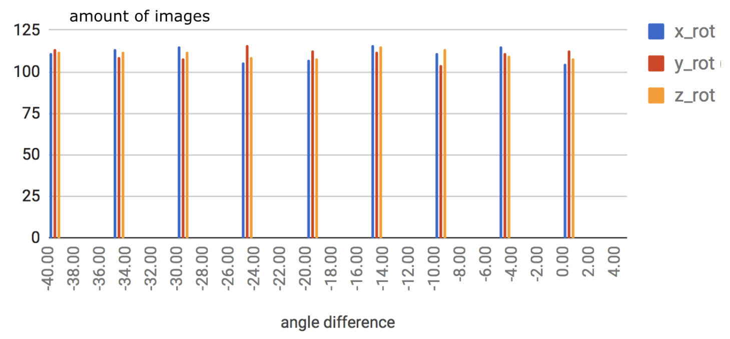


Figure 16: The network is not able to predict any rotation

Figure (16) shows a very 'unlucky' training session, for which the network learns no representation whatsoever.

This poses a problem, since the training is not reliable. When using the network configuration in production, running multiple sessions to train the network multiple times, waiting for a 'lucky' session in which all rotations are learned is not efficient.

The best explanation for the observed behavior was that the small amount of feature maps extracted in the first convolutional layer creating a type of bottleneck for the network during training. Offering the network more possibilities of learning low level features by having the first convolutional layer extract double the original amount of feature maps showed immediate improvement. Given the time it takes to run a training session, this was only evaluated over two sets of 10 training sessions. To keep the layer pattern defined by the Stanford course in which the extracted number of features increase by 2 for each preceding convolutional layer, another 2 sets of 10 training sessions were done with extracting (16,32,64,128) feature maps at the respective layers. Since this did not show a difference to the previous configuration of (16,16,32,64) extracted feature maps, this was the configuration used.

Also at this stage, the representation of rotation was changed from using degrees to using points on a unit circle $(\cos\alpha,\ \sin\alpha)$ as a representation of angle, figure (17), as described in the paper by Hara et al. [5]. Originally the reason for this was that at this stage, also cube-like objects without rotational symmetry were considered. Using this representation solved the problem of the linear representation of degrees mentioned in section (3.3).

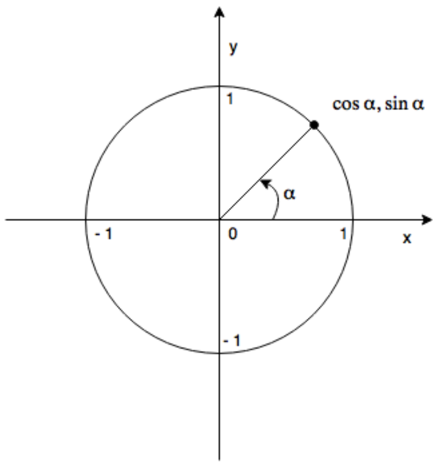


Figure 17: Rotation α represented as coordinates $(\cos\alpha,\ \sin\alpha)$ on a united circle

Restricting the range of rotation for an object with rotational symmetry, translates to using only a portion of the values on the unit circle, no longer being a continuous representation. However, this does balance out the representation of rotation and location for all labels, further discussed in the next section, also it offers the possibility for expanding the usage of the network to objects without rotational symmetry. Those were the reasons it remained the representation of rotation.

5.3 Realistic Scenario

In experiments for which the training data was created by rotating the object around and translating the object along all three axes, the network predictions of the object's pose were very far off. Realizing that this solved a more general problem than needed, the decision was made to introduce some restrictions to better fit the actual real-world scenario.

Unfortunately, the delivery of a cube for which this network should predict, was delayed and therefore a surrogate cuboid object had to be used. Figure (18) shows the surrogate object and a rendered model of the object.

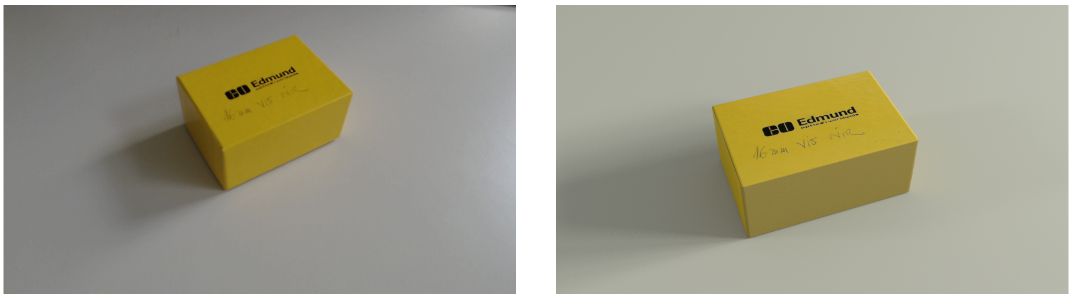


Figure 18: The left image is a real image of the surrogate object and the right image is a rendered image using the left image as a reference.

The consequence of using synthetic data is that the network might train well on training data, but might not be robust enough to the slight differences of the environment in the end application. To model for variations of lighting and shadows in our industrial setting, the lighting was randomized for each rendered image.

The following section uses the labels (x, y, $\cos\alpha$, $\sin\alpha$) to define the location (x, y) and rotation ($\cos\alpha$, $\sin\alpha$) of the cube. Analyzing the workbench in the industrial setting, explains why the labels were chosen this way. Figure (19) shows a diagram of the workbench. The triangle represents the camera, viewing an origin O of our coordinate system at an angle β and distance d. The blue field indicates the field of view for the camera.

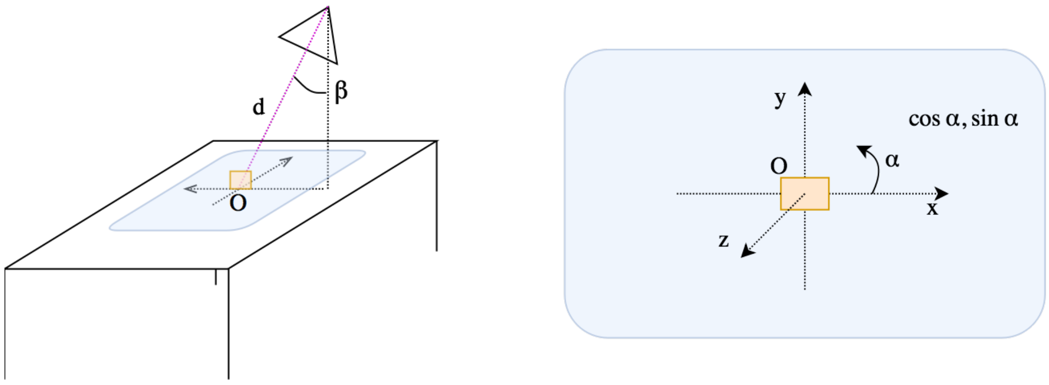


Figure 19: On the left the workbench with an object (orange box) and camera (triangle) is shown and on the right the view of the coordinate system from the camera is shown.

A camera with known viewing angle β and distance d to the origin O as well as working with a cuboid object, helps simplify the pose problem. Even though the surrogate object had 3 unique faces, a cube has only 1. Also, a cuboid object cannot stand alone on its edge or corner, it can only stand on its faces. To avoid object symmetry and to take into account the rules of physics, the decision was made to train the network to predict the rotation of the object only around its z-axis versus training the network to predict the rotation around its x,y- and z-axis.

Since this cuboid has a rotational symmetry for intervals of 180°, if disregarding the writing on the top, the images were rendered with a rotation of the cuboid for the range 0° - 180° in an interval of 5° around the z-axis. For a cube this can later be adjusted to using a restricted range of 0° - 90°. As described in the earlier section, the representation used for the rotation are the coordinates on a unit circle $(\cos\alpha,\ \sin\alpha)$. The values are scaled to range between 0 - 200 as labels for the rotation when training the network. To evaluate the quality of the network's predictions on the original range of values, these are then scaled back down and converted back to degrees using the *atan2* function.

To represent the full range of location for the object on the table, we only need to indicate a translation along the x- and y-axis of the coordinate system centered at O. The units for the coordinate system used were cm. The images were rendered by translating the cuboid along the x- and y-axis in a range of -2.0cm - 2.0cm in an interval of 0.5cm. These values were also scaled to range between 0 - 200 for training and scaled back down when evaluating the quality of the network's predictions.

Having half of the labels represent rotation and half of the labels represent the location of the object as well as the equal range of possible values for each label, helps avoid a biased network training. For example, that learning a representation of location does not overpower learning a representation for rotation, or the translation along the x-axis does not overpower the translation along the y-axis. Considering that the mean squared error calculates a combined loss for all labels, an equal range and equal representation establish an equal weighing of the different components for the calculated error.

The representation introduced in this section simplifies the problem in that the network only has to predict a subset of all possible ways the cuboid can move in three-dimensional space. It no longer has to predict the translation of the cuboid along the z-axis since it is always flat on the table and the distance of the camera to the table is known. It also only has to predict the rotation of the cuboid about its z-axis. The reduced dimensionality of the total descriptors for an image, reduce the complexity of the function needed to describe the pose of the object within the image. This also means, less amount of total training images is needed to cover the range of possibilities, as well as potentially less amount of training time.

5.4 Tuning for Rotational Accuracy

Recalling the application for which the predictions are needed, the exact position and rotation of an edge on a cube, makes the precision of the prediction extremely important. At this point, the network was learning to represent the rotation and location of the object, however with an average error of ~20° for rotation and ~0.05 cm for the translation along the x- and y-axes respectively. For the robot in the foreseen application to smooth out the edge perfectly, a significant decrease in error had to be obtained for rotation. The following section uses histograms and standard deviation to illustrate the impact of changes made to improve the network's predictions.

The calculated accuracy mentioned in this section refers to Keras' binary accuracy discussed in section (3.2). This accuracy was calculated during training over a small set of images every couple of iterations as an indication of the network's learning progress during a training session. The labels ranged between 0 - 200 as discussed in section (5.3). The histograms and standard deviations in this section were obtained by letting the trained network predict over a test set of 1000 images, scaling the label values back, converting them back to the original degrees and eventually calculating the difference between the recalculated original degrees and the recalculated network predictions. This was an evaluation of the network's quality for the actual range of labels.

Even though the calculated accuracy was relatively high towards the end of different training sessions (~0.95), the evaluation showed an error of ~20°. By increasing the number of iterations, the calculated accuracy fluctuated between ~0.95 and 1.0. To gain more control over the network's state towards the end of training, a validation set was introduced. The database of training images was split into 70% training set and 30% validation set. Each 20 iterations the accuracy was calculated over 350 images from the validation set. Once the rounded calculated accuracy for the validation set reached 1.0 training would end, leading to a standard deviation of 14.66°.

Experiments in which the network had to predict both location and rotation always had less accurate results than experiments in predicting only one of the two. Intuitively, this makes sense since the combined problem is more complex. This sparked the idea of splitting the last three layers (Convolution, Pooling and Dense) up into two branches, such that each branch could potentially specialize in learning the rotation and translation separately. (Figure (23) in section (6) shows a graphical representation of the branches.) The learnable parameters in the two branches would then no longer have to be shared to encoded location and rotation simultaneously. Training a network with split layers led to a decreased standard deviation of 8.6°.

The question remains if the splitting of the layers is the reason for the increased accuracy. Since splitting up the layers, means calculating the accuracy over rotation and translation separately, an accuracy of 1.0 for rotation and 1.0 for location is an overall higher accuracy, than a combined accuracy of 1.0. Nevertheless, the histograms in figure (20) show a narrower range for the error of the predictions on rotation.

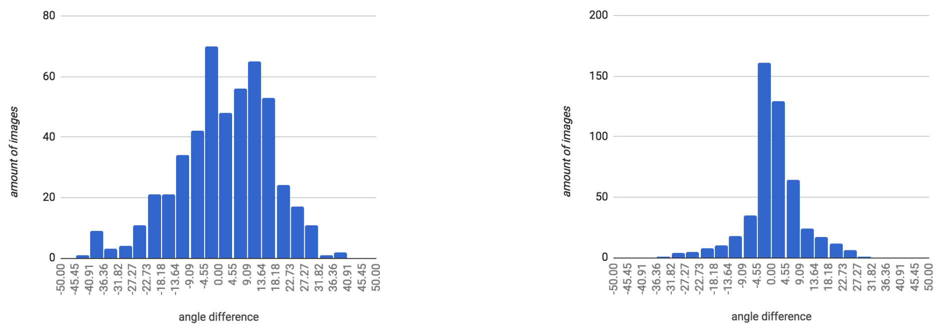


Figure 20: Histograms comparing predictions of rotation for a network without split layers (left) and one with split layers (right).

A standard deviation of 8.6° is still not precise enough for the application. Exploiting the advantage of creating unlimited training images, the interval of rotation, when rendering our training images, was decreased from 5° to 1° and the interval for translation of the cuboid along the x-and y-axis was decreased from 0.5cm to 0.1cm for rendering the training images. Even though the predictions made for the translation of the cuboid were very accurate, the smaller interval for translation was also changed, to keep an equal balance between the two. Figure (21) shows the improved rotation.

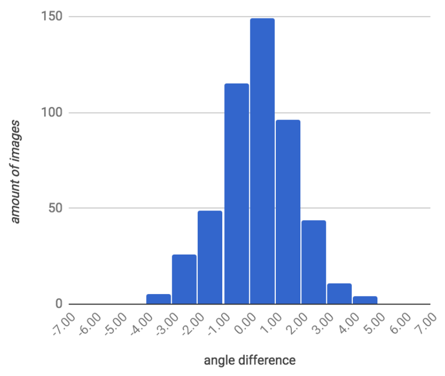
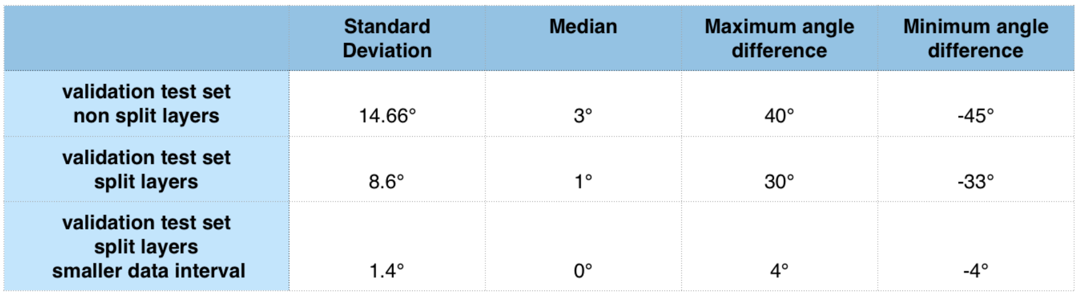


Figure 21: Histograms showing the evaluation of a network trained on images with smaller rendering interval.

A standard deviation of 1.4° for rotation is an acceptable result for the application. At this point, the standard deviation for translation along the x-axis is 0.01cm and 0.02cm for translation along the y-axis. A slight improvement to the ones at the beginning of the section, however since these are cm units the deviation is already barely visually noticeable, the improvement was appreciated, but not necessary for the application. Figure (22) summarizes the standard deviations discussed in this section. The median, maximum and minimum angle differences are given for further insite.



	Standard Deviation	Median	Maximum angle difference	Minimum angle difference
validation test set non split layers	14.66°	3°	40°	-45°
validation test set split layers	8.6°	1°	30°	-33°
validation test set split layers smaller data interval	1.4°	0°	4°	-4°

Figure 22: Calculations used as a means of comparison obtained from the three histograms within this section.

6 Final Network Architecture and Results

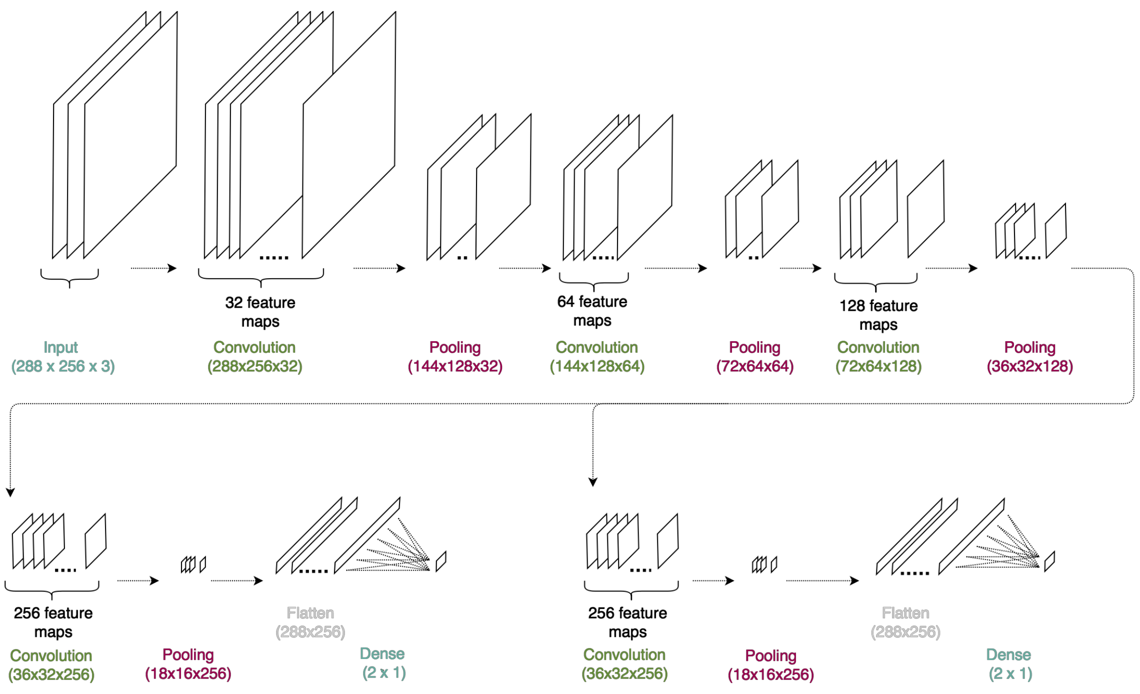


Figure 23: Final network architecture

The final network architecture is a single deep convolutional neural network using regression to estimate the location and rotation of a cuboid. Figure (23) shows the graphical representation of the final network architecture. It consists of multiple different layers and splits after the third pooling layer into two branches of the network, resulting in two different output layers, one for rotation and one for location.

All pooling layers have a kernel with dimension (2x2) and stride 2. Meaning each pooling layer will bring the data dimension down by 75%. All convolutional layers have a kernel with dimensions (3x3) and a stride of 1. A zero-padding around the image makes sure that the width and height of the feature maps extracted at each convolutional layer are the same as the width and height of the input to the convolutional layer. The first convolutional layer extracts 32 feature maps and the amount of feature maps extracted at each convolutional layer is double the amount extracted in the previous convolutional layer. All biases in the network are initialized to zero and the weights are initialized using the Glorot-uniform distribution [4]. The optimizer used is the Adam optimizer, mentioned in section (3.2).

The network predicts the location of the surrogate cuboid on the table and the its rotation around its z-axis, which is perpendicular to the table. The rotation is predicted as scaled values for the points on a unit circle representing the angle of rotation ($\cos\alpha$, $\sin\alpha$) and the location is predicted as scaled coordinates of the cuboid on x,y-plane of the coordinate system, represented by the table. Training images are rendered for all combinations of the cuboid rotated at an angle of 0° - 180° in 1° intervals, and translated along the x-axis and the y-axis in a range of -2.0cm - 2.0cm in 0.1cm intervals. Each image is randomly lit.

Training is done over ~ 3000 iterations with batch sizes of 10, ending once an accuracy of 1.0 for both rotation and location is reached for a validation set. Validation is done every 20 iterations over a set of 350 before unseen images. This amount was chosen as the maximum amount, before having memory problems.

Figure (24) shows the graph for the calculated binary accuracy and figure (25) shows the graph for the loss calculated over the validation set during a training session. A declining loss and an increasing accuracy indicate a network capable of minimizing the error for its predictions and learning an increasingly better representation for the pose of the object.

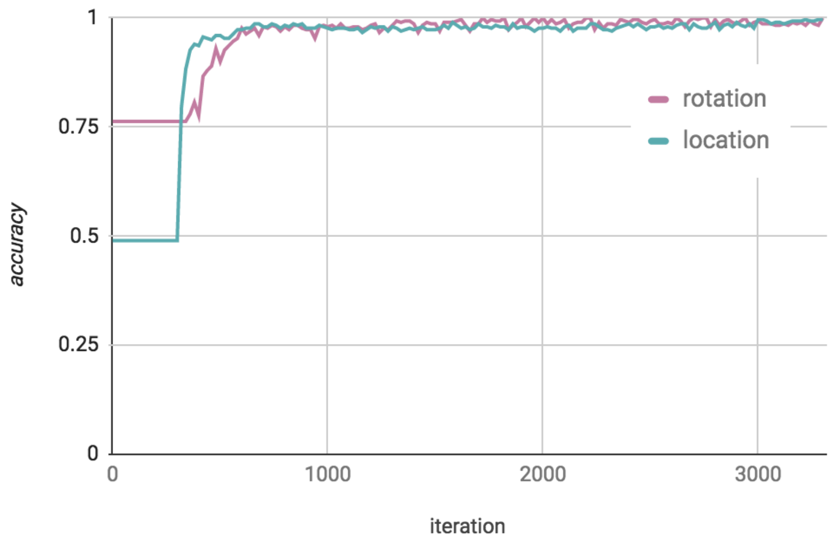


Figure 24: The accuracy calculated over a validation set during training

The steep incline in accuracy and steep decline in loss reflect the fast learning of the network in early iterations. The then gradual incline in accuracy and gradual decline in loss reflect the increased time the network needs to fine tune its parameters. A possible explanation for this are the scaled labels. The effect of the scaled label is an initial larger calculated loss which leads to taking bigger steps towards the minimum of the loss function during optimization. Having the same effect as a higher learning rate.

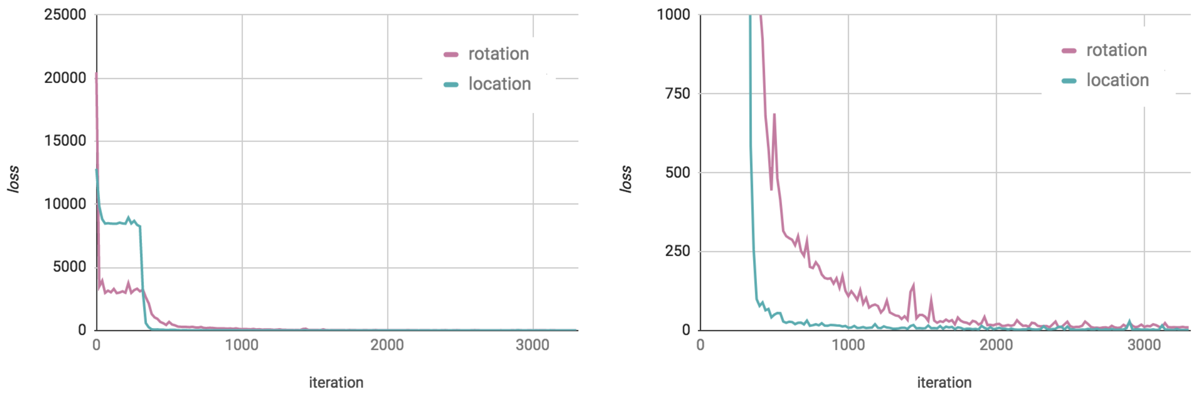


Figure 25: The loss calculated over a validation set during training. The left graph shows the complete range of loss and the right graph shows the loss for the same training session however only for values in the range 0 - 1000.

Figures (26) and (27) show the evaluation of the trained network after the previously mentioned training session. The evaluation of the trained network is done with the same method as explained in section (5.4). The histograms show a network able to generalize well when evaluated over a test set of images.

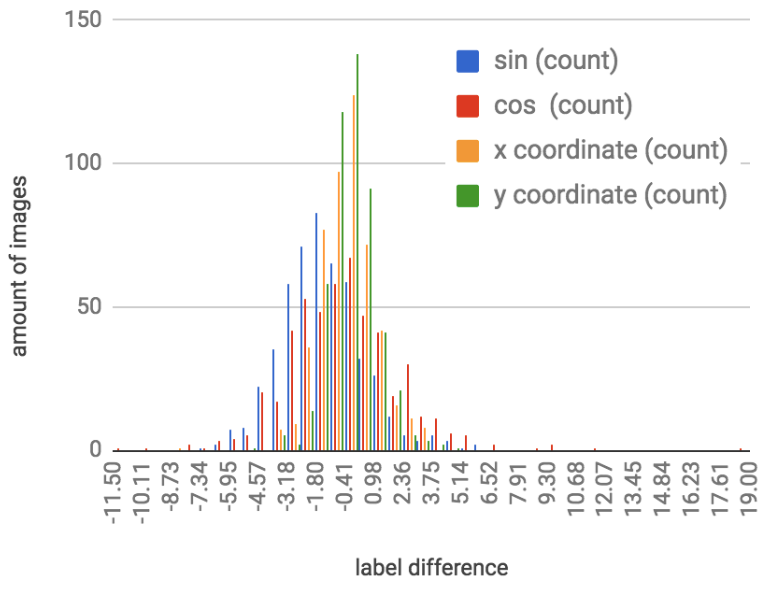


Figure 26: Histograms showing the evaluation of the trained network for the scaled labels ranging from 0 - 200.

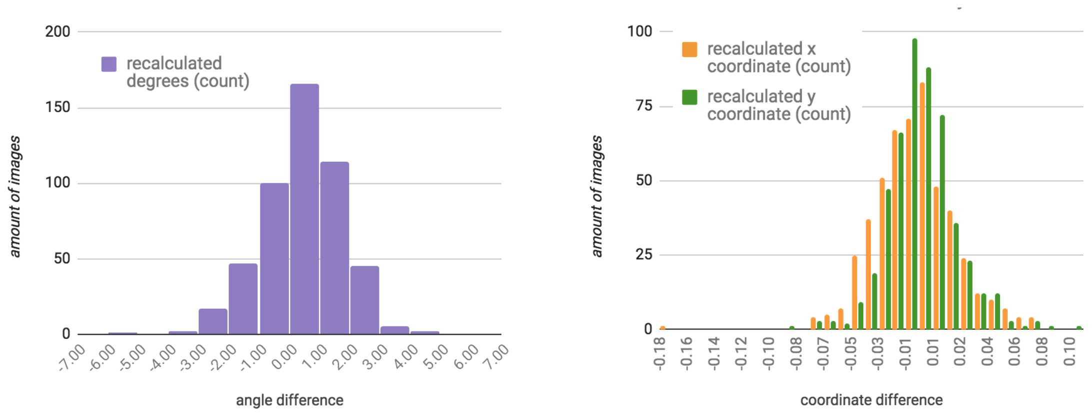


Figure 27: Histograms showing the evaluation of the trained network for recalculated labels. The left histogram shows the evaluation for the degrees ranging from 0 - 180. The right histogram shows the evaluation for the location, values ranging from -2.0 - 2.0

Images in figure (29) and figure (30) show the visual accuracy for predictions of this network configuration. The image on which the network makes a prediction is shown and rendered on top of it is a blue wireframe using the predicted values.

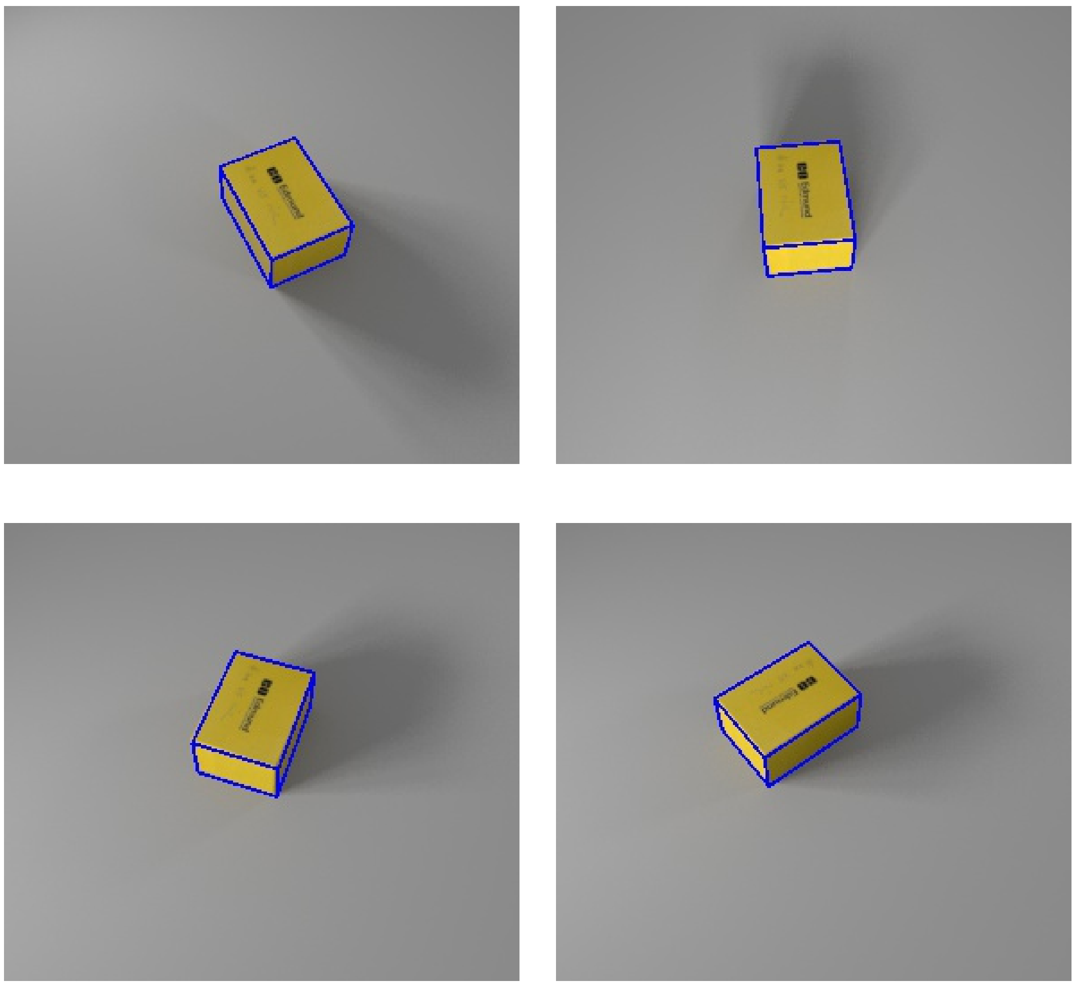


Figure 29: Predicted poses are rendered as a wireframe and are seen as the blue outlines on top of the original image, for which the prediction was made by the network.

Figure (30) exemplifies a worse prediction of the network on a training image. The angle of the prediction is slightly off, however the predicted x and y coordinates seem pretty accurate.

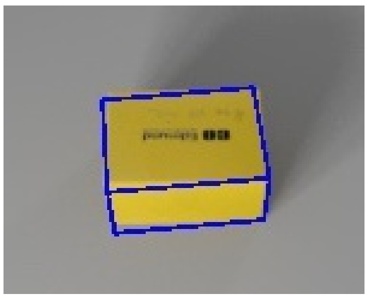


Figure 30: An example of a less accurate prediction.

7 Outlook and Conclusion

The network in this thesis could only ever be evaluated theoretically over a test set of rendered images, due to the delayed arrival of the actual cube and not having the actual setup in the industrial environment. The next step for using this network for the application described in section (1.1), would be testing the network's ability to predict the pose of the actual cube from real images taken in the actual industrial setup.

Despite the fact that rendering images with Blender allowed creating an unlimited amount of training data, early tests show their disadvantage when predicting the pose of an object taken from the real world. Figure (31), shows the results of these tests done by letting the network predict the pose of the real cuboid object used as reference for the rendered training data in an uncalibrated environment.

The camera's position used to take the images is highly approximate. Therefore, the predictions might have been more accurate than the rendering of the blue wireframe visually indicate.

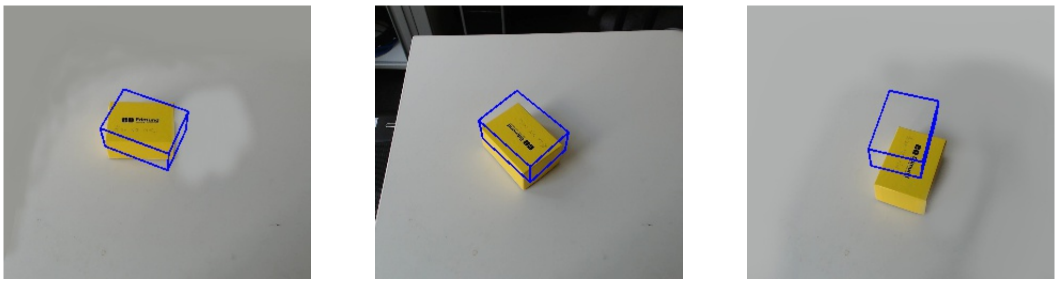


Figure 31: Early tests of predictions on real world images.

A possible explanation is that variations in the image, seemingly unimportant through visual comparison have a greater effect on the predictions. These variations could include lighting, color differences in RGB space, blur or noise amongst many other variations. It seems that the training images did not sufficiently account for these, throwing off the network, when seeing such outliers.

If the missing real world noise in the training set is a possible explanation, then the paper published by Tobin et al. [12] which explores modeling for domain randomization to bridge the 'reality gap', might offer a solution. Their approach of increasing randomization in rendered training images, through random lighting positions and strengths, random textures and randomizing the cameras field of view increases the chances of better predictions on real-world images. Tobin et al. use upwards of 1,000 different textures, to give an idea.

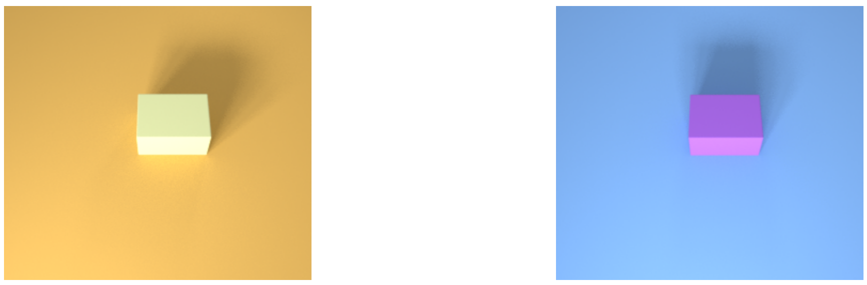


Figure 32: A possible randomization of color for the training images used in this thesis.

Another possible solution for bridging the gap between predicting on real-world images and synthetic images is the approach described in the paper published by M. Rad and V. Lepetit discussed in section (2). Their attempt to fine tune the accuracy of the estimation made by the network is using a second network. The second network is trained to predict a difference in pose. Its input are two images, the first being the original image for which the first network has made a prediction and the second image is rendered using the network's predictions for the first image. The target output is the difference in pose. This approach was implemented and partially tested in this thesis until simultaneous experiments described in section (5.4) showed more promising results. However, after observing the network's predictions for the real world images, this might be a solution worth revisiting.

In conclusion, the deep convolutional neural network described in this thesis predicts the location and rotation of a cuboid on a workbench. The network is trained and evaluated using rendered images modeling the environment for a specific industrial application. The network architecture was initialized using tested hyperparameters and later fine tuned to fit the specific problem. Evaluations show that the network is able to generalize well over the cuboid's pose in rendered images.

References

[1] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems.* Software available from tensorflow.org. 2015. URL: `https://www.tensorflow.org/`.

[2] Blender Foundation. *Blender.* Version 2.79. URL: `https://www.blender.org`.

[3] François Chollet et al. *Keras.* `https://github.com/fchollet/keras`. 2015.

[4] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics.* Ed. by Yee Whye Teh and Mike Titterington. Vol. 9. Proceedings of Machine Learning Research. Chia Laguna Resort, Sardinia, Italy: PMLR, 2010, pp. 249–256. URL: `http://proceedings.mlr.press/v9/glorot10a.html`.

[5] Kota Hara, Raviteja Vemulapalli, and Rama Chellappa. "Designing Deep Convolutional Neural Networks for Continuous Object Orientation Estimation". In: *arXiv preprint arXiv:1702.01499* (2017).

[6] Tomáš Hodan et al. "T-LESS: An RGB-D Dataset for 6D Pose Estimation of Texture-less Objects". In: *Applications of Computer Vision (WACV), 2017 IEEE Winter Conference on.* IEEE. 2017, pp. 880–888.

[7] Itseez. *Open Source Computer Vision Library.* `https://github.com/itseez/opencv`. 2015.

[8] Diederik Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

[9] Michael A Nielsen. *Neural networks and Deep Learning.* Determination Press USA, 2015.

[10] 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: *arXiv preprint arXiv:1703.10896* (2017).

[11] Irvin Sobel. "An isotropic 3× 3 image gradient operator". In: *Machine vision for three-dimensional scenes* (1990), pp. 376–379.

[12] Josh Tobin et al. "Domain Randomization for Transferring Deep Neural Networks from Simulation to the Real World". In: *arXiv preprint arXiv:1703.06907* (2017).

[13] Stanford University. *CS231n Convolutional Neural Networks for Visual Recognition.* University Lecture Notes. 2017. URL: `http://cs231n.github.io`.