Universitetet
i Stavanger

FACULTY OF SCIENCE AND TECHNOLOGY

MASTER'S THESIS

Study program/specialization: Industrial Automation and Signal Processing	Spring semester, 2016 Open / Confidential: Open
Author: Jonatan Sjølund Dyrstad	Jonatan Sjølund Dyrstad (signature author)

Instructor: Kjersti Engan
Supervisor(s): Kjersti Engan (UiS), John Reidar Mathiassen (SINTEF)

Title of Master's Thesis: Training convolutional neural networks in virtual reality for grasp detection from 3D images

Norwegian title: Trening av nevrale konvolusjonsnettverk i et virtuelt miljø for gripevektorestimering med 3D-bilder

Subject headings:
Deep learning, convolutional neural network, virtual reality, robotics, synthetic data generation

ECTS: 30

Pages: 84
+ attachments/other: 104

Stavanger, 15/6-2016
Date/year

Training convolutional neural networks in virtual reality for grasp detection from 3D images

by
Jonatan Sjølund Dyrstad

Universitetet
i Stavanger

A master thesis in Industrial Automation and Signal Processing
submitted to
The Department of Electrical Engineering and Computer Science
University of Stavanger
June 2016

Abstract

The focus of this project has been on training convolutional neural networks for grasp detection with synthetic data. Convolutional neural networks have had great success on a wide variety of computer vision tasks, but they require large amounts of labelled training data, which currently is non existent for grasp detection tasks. In this thesis, a novel approach for generating large amounts of synthetic data for grasp detection is proposed. By working solely with depth images, realistic looking data can be generated with 3D models in a virtual environment. It is proposed to use simulated physics to ensure that the generated depth images captures objects in natural poses. Additionally, the use of heuristics for choosing the best grip vectors for the objects in relation to their environment is proposed, to serve as the labels for the generated depth images.

A virtual environment for synthetic depth image generation was created and a convolutional neural network was trained on the generated data. The results show that neural networks can find good grasps from the synthetic depth images for three different types of objects in cluttered scenes.

A novel way of creating real world data sets for grasping using a head mounted display and tracked hand controllers is also proposed. The results show that this may enable easy and fast labelling of real data which can be performed without training by non-technical people.

Preface

The work done in this project concludes a five year masters program in automation and signal processing at the University of Stavanger. The problem formulation for the thesis was developed in cooperation with SINTEF Fisheries and Aquaculture as part of an ongoing project called *The Humanoid Robotics Roadmap 2030*, which aims at developing robotic technology for future use in the industry.

I would like to thank my supervisors, Kjersti Engan at UiS for valuable inputs on structure and writing style and John Reidar Mathiassen at SINTEF for the many hours spent discussing the problems encountered along the way, as well as for specific inputs on how to improve the final drafts. I would also like to thank Peter Leupi for spending his time reading different versions of the thesis and, as always, saying it like it is when something doesn't meet his high standards for clarity and precision. Lastly, I want to thank Åsmund Pedersen Hugo for proofreading the final draft. With his borderline compulsive emphasis on eloquent wording, he is truly the ideal man for the job.

Work on this project has been great fun, and I am looking forward to continuing the work in cooperation with SINTEF. We will try to release a polished and easy to use version of the developed system for data generation in the near future.

A movie summarizing the work done in this project can be viewed at:

`https://www.youtube.com/watch?v=JitH6b9i5pQ&feature=youtu.be`

Contents

List of Figures

Chapter 1

Introduction

We are now entering a new era of robotics, one in which manufacturers are developing robots that are something closer to collaborators or colleagues than tools. Currently, humanoid robots are employed at hotels [1] and even ABB's industrial robots have changed color from the iconic orange, designed to keep people away, to a more subtle and friendly graphite white[2]. However, in today's industry many laborious and repetitive tasks are still performed by human workers. This is especially true in the food industry where biological variations in the handled material makes automation challenging. As rapid progress is made in several technological areas, a possibility of creating more flexible robotic systems capable of matching human performance at simple, yet varied tasks seems to emerge.

There is a need for a robot that is capable of handling a wide variety of materials subject to occlusion, deformation and changes in environmental conditions. Such a system needs to be capable of differentiating between objects and finding suitable ways to grip each one so that the robot can handle it according to the given task. This problem of *grasp detection* has been the focus of this project.

Deep artificial neural networks (DNNs) have had great success in a variety of computer vision tasks in recent years. They are approaching human performance on image classification and object detection tasks in competitions such as the large scale visual recognition challenge [3]. At the same time, depth cameras such as the Shapecrafter and Microsoft Kinect 2 have become cheaper and precise enough for use in the industry. The system proposed in this project is a DNN which takes a depth image as input. The output of the network is the type of object detected in the depth image (if any) along with a grip point and grip vector for the detected object. An example of use for this system could be sorting of fish based on some criteria, e.g. size, and this is depicted in figure 1.1.

One of the drawbacks of DNNs is the large amounts of data required to train them. The proposed system utilizes a developed virtual environment to meet these data requirements. In virtual reality (VR), the network can be exposed to thousands of example tasks in a short period of time. If the experience gained in VR is transferable to the real world, this could reduce the need for on site training of a robot dramatically.

The VR-environment used for the experiments presented in this project is based on the Unity game engine [4]. Thousands of objects can be instantiated and using Unity's built-in physics engine they can be dropped or moved around to land in natural ways in any scene. Synthetic depth images are rendered of the objects and saved as training examples along with the grasps for the objects in their current pose.

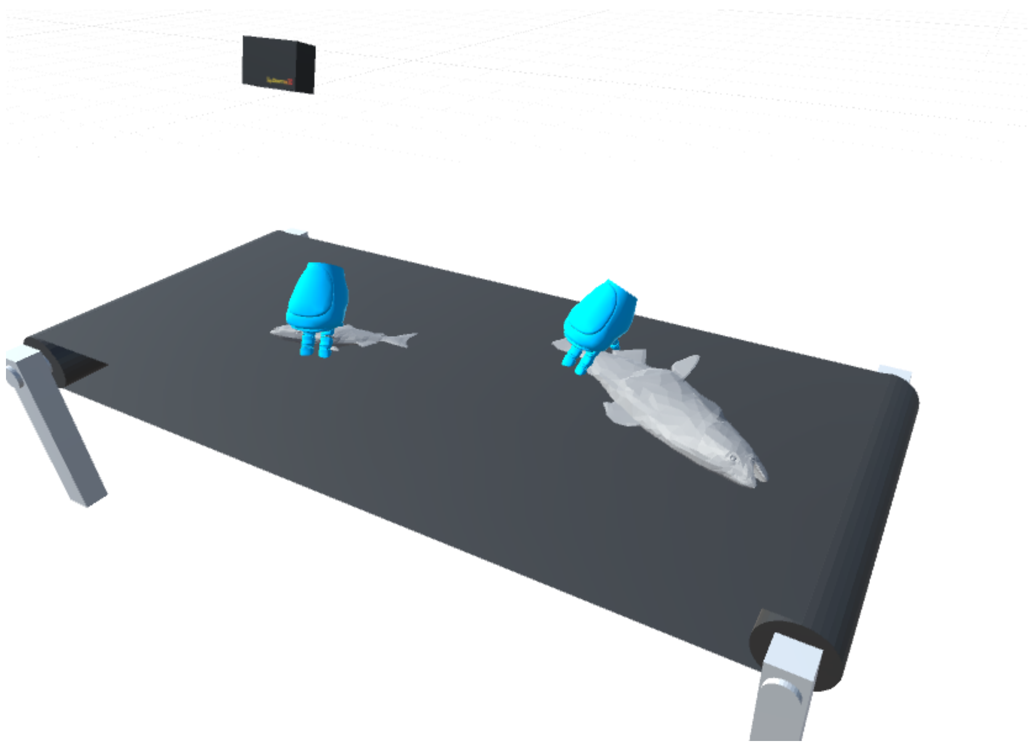


Figure 1.1: An example of use for the system proposed in this project. The input to the neural network is a depth image captured by a depth camera (upper left corner). The output of the system is a grip point and grip vector for each object illustrated by the overlaid end effectors in blue.

If robots are to become good ”colleagues”, they need to be flexible and easy to communicate with. Ideally, a robot should be able to step in on any part of a production line, much like a human worker would, and perform a task after a short period of training. A worker who is familiar with the task at hand should be able to communicate it directly to the robot, without the need for an engineer or programmer to ”translate” it into machine language. Such a system for easy communication and re-purposing of robots is also proposed in this thesis.

With the developed human-machine interface a human ”trainer” can teach the robot how to grasp never seen before objects by showing it in virtual reality. A point cloud captured by the depth camera is loaded in VR and the trainer can enter and walk around in this environment naturally using a head mounted display (HMD). Using a set of tracked hand controllers, the trainer can place precise grip vectors on objects, and thereby generate a data set of real world examples in an easy and intuitive way. A real world pseudo example where this system could be used on a production line, would be if a robot needed to be re-purposed from sorting apples and oranges to sorting bananas and cucumbers.

The focus of this project has been on investigating if deep neural networks can extract enough information about types of objects and their pose from synthetic depth images to estimate good grasps. A considerable effort has been made in creating a good VR environment for producing realistic synthetic data, and as a good interface between man and machine.

In future work more research will be done into how well experiences in VR transfer to the real world and the realism of the synthetically created depth images will be increased based on what was learned in this project.

1.1 Related work

Detection of robotic grasps is an active area of research and different approaches has been made by different research teams. At Google, training of a large convolutional neural network is done on monocular images for learning of hand-eye coordination for grasping [5]. The use of monocular images alone, forces the network to observe the spatial relationship between the gripper and the objects in the scene, thus learning hand-eye coordination. The network is used to servo the gripper in real time and the result of a grasp attempt creates a labelled example which is used to train and improve the neural network. A set of 14 robots were used to gather over 800,000 grasp attempts over the course of two months with variations in camera positions and hardware (see figure 1.2). The combined "experience" from all robots is used to improve the behaviour of all, thus enabling much faster learning than could be achieved with only one robot.

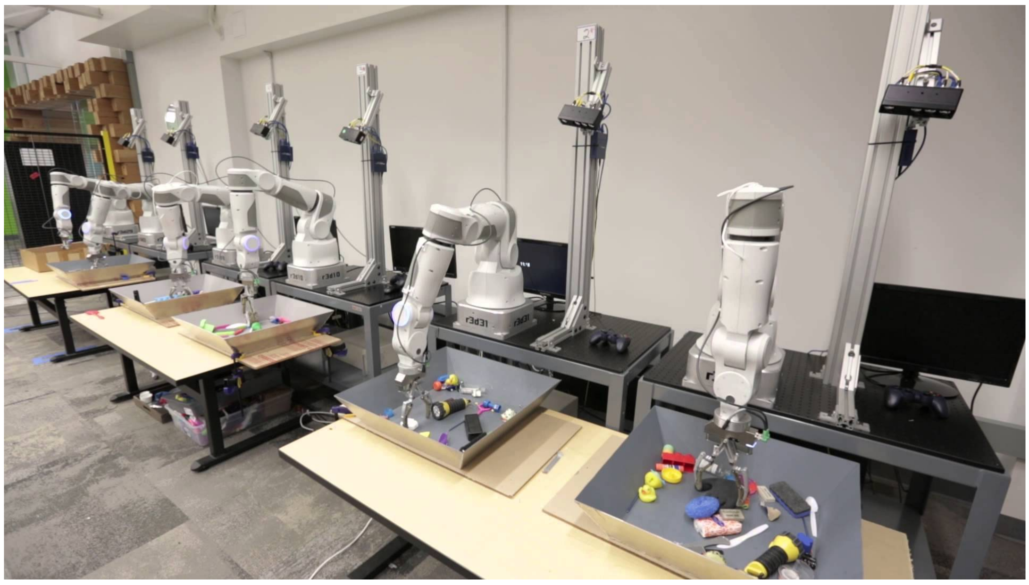


Figure 1.2: Some of the robots used at Google during training [6].

The robots at Google have been shown to not only learn ways of grasping objects, but also to learn intelligent behaviours such as isolating objects from a group before grasping.

In a paper from 2015, Lenz et al. at Cornell University focused on the use of deep learning for grasp detection from RGB-D images [7]. They used the extended version of the Cornell grasping data set consisting of 1035 images of 280 graspable objects and found that good grasps could be achieved by combining RGB, depth and surface normal information. The data set is labelled with rectangles indicating good and bad grasps in the 2D image plane and the full 3D configuration of an end effector is inferred from the surface normals at that point in the image, see figure 1.3.

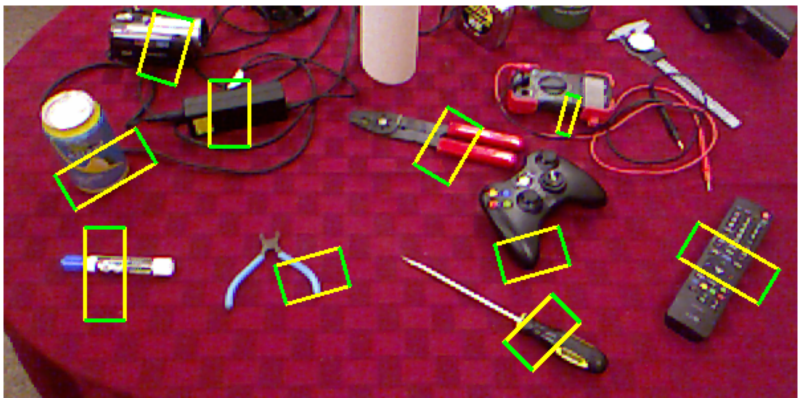


Figure 1.3: The predicted grasps output from the system trained by Lenz et al.[7].

A grasp was considered a success if the intersection of two bounding boxes over their union scored at least 25 % and they showed that deep learning outperforms even well designed hand engineered features with this metric. With training on RGB, depth and surface normals they got an accuracy of 93.7 % and using only depth images they got 92.4 %. Because of the limited amount of available training data, they pre-trained the hidden layers in the network using a *sparse auto encoder*.

In general, large amounts of labelled data always increase the performance of deep neural networks. Unsupervised training methods and *transfer learning*, i.e. pre-training a network on some similar data for a different purpose, before transferring the parameters to the real network can reduce this need.

Some work has been done, suggesting that computer generated images with precise labels can increase the performance of deep learning algorithms. In 2015, Wood et al. outperformed the current state of the art systems for gaze estimation by utilizing synthetic images of eyes (see figure 1.4)[8]. They rendered a large amount of images of eyes with different lighting conditions and *environment maps* to create a very realistic data set of RGB images. With this approach, they created over 11,000 images with perfect labelling.

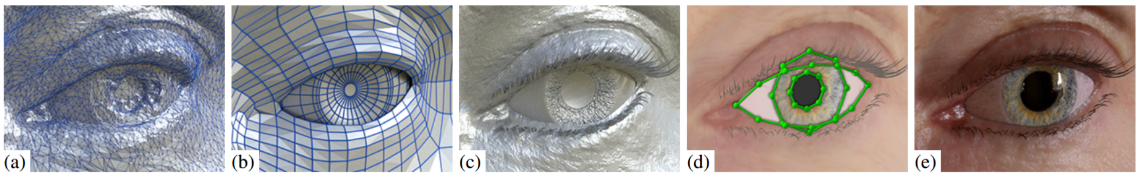


Figure 1.4: The rendering pipeline for generation of the photo realistic images used by Wood et al. for gaze estimation[8].

In 2011, Shotton et. al created hundreds of thousands of synthetic depth images with 3D models and a set of recorded human motions to create a human pose estimator for the Microsoft Kinect sensor [9]. Later, rendered depth images from 3D models have also been used with great success on object recognition and detection tasks as well as for object pose estimation, see figure 1.5 [10, 11]. In 2015, Zheng et. al found that adding more synthetic renderings into the training set helps performance on classification tasks in the real world[10]. They also found that using many models increase the accuracy, but at some point if unusual shapes are introduced, the performance decreases.

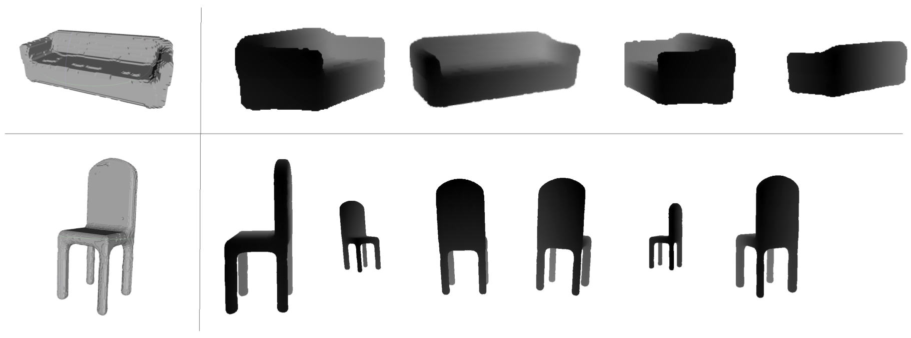


Figure 1.5: Some synthetically created depth images generated by Zheng et. al used to train a system for object detection and pose estimation [10]

In this project, the focus has been on grasp detection, rather than visual servoing of a physical robot. Proposed is the use of large amounts of synthetic data to train a deep neural network for this purpose. Based on the findings of Lenz et al. we suggest the use of depth images alone to predict robotic grasps, because this greatly simplifies the process of generating realistic looking synthetic data. A novel approach to ensure realism in the synthetic data is proposed by using simulated physics and heuristics for choosing the best grasp for an object in a certain pose as label in the training set.

Chapter 2

Background theory

2.1 Artificial neural networks

In the field of machine learning the term *artificial neural network* or simply *neural network* (NN) refers to a way of approximating mathematical functions loosely inspired by the biology of the brain. Neural networks are made up of *neurons* that connect to other neurons through weighted connections as shown in figure 2.1. The illustrated neuron takes two inputs x_1 and x_2 and has two weighted connections w_1 and w_2, one from each input to the output. The output computed by the neuron is given by $y = g(x_1w_1 + x_2w_2)$, where the function $g(\cdot)$ is called the *activation function.*[1] There are many different activation functions to choose from, whereof the traditional (but now less common) sigmoid function is defined as:

$$g(z) = \sigma(z) = \frac{1}{1 + e^{-z}}$$

This particular activation function has the effect of limiting the output of a neuron to a number between 0 and 1. The sigmoid and its derivative are shown in figure 2.2

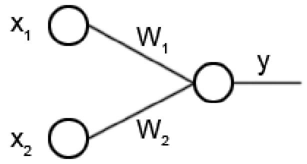


Figure 2.1: Illustration of an artificial neuron with two inputs

Given some inputs $\{x_1, x_2, ...x_n\}$, a neural net can be used to compute a desired output $\{y_1, y_2, ...y_m\}$ by adjusting the weights. Consider the problem of computing the logical AND and OR operations.

$$y_1 = x_1 \wedge x_2 = \begin{cases} 1, & \text{if } x_1 = x_2 = 1 \\ 0, & \text{otherwise} \end{cases}$$

[1]The activation function $g(\cdot)$ is sometimes referred to as *the non-linearity.* It is important that the function is non-linear for the network to be able to learn more complex functions, see A.2

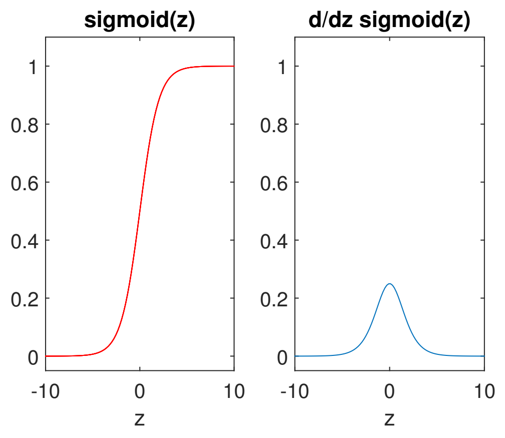


Figure 2.2: The sigmoid function (left) and its derivative (right)

$$y_2 = x_1 \vee x_2 = \begin{cases} 0, & \text{if } x_1 = x_2 = 0 \\ 1, & \text{otherwise} \end{cases}$$

If we assume binary inputs and use a simple threshold at 0.5 as activation function, we can compute these logical operations with a small neural network with $n = 2$ inputs and $m = 2$ outputs as shown in figure 2.3.

$$thresh(z) = \begin{cases} 1, & \text{if } z >= 0.5 \\ 0, & \text{otherwise} \end{cases}$$

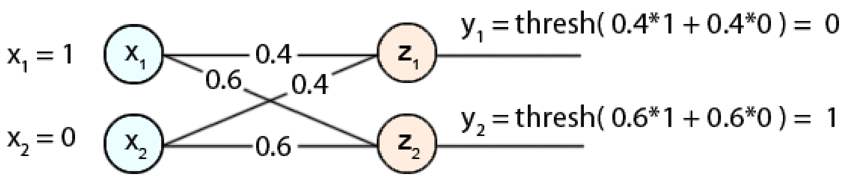

x1 = 1
x2 = 0
x1
x2
0.4
0.6
0.4
0.6
z1
z2
y1 = thresh(0.4*1 + 0.4*0) = 0
y2 = thresh(0.6*1 + 0.6*0) = 1

Figure 2.3: Two neurons computing the logical AND (y_1) and OR(y_2) functions

The network in 2.3 has two neurons in the *top layer*, the first (y_1) is computing the AND-function and the second (y_2) the OR function. The network also has two inputs in the *input layer*, x_1 and x_2. The weights on the connections between z_1 and the inputs are both 0.4 and the weights on the connections between z_2 and the inputs are both 0.6. For real world problems, it is common to add another input to the input layer $x_0 = 1$, called *the bias unit*. The need for such a bias becomes apparent if we consider a regression problem with one input, as illustrated in figure 2.4. A good fit to this example data can be found with a linear model $y = w_b + w_x x$ with $w_b = 2$ and $w_x = 0.5$, where w_b is the weight on the connection between the bias and the output neuron.

In general, neural networks can be used to approximate highly complex, non-linear functions of the input. This is done by organizing neurons into *layers* as shown in figure 2.5. The input features, organized in the *feature vector* $\mathbf{x}$, are used to compute a new feature vector for the following layer in the network. Many of today's neural networks consists of tens or hundreds of layers, and have given rise

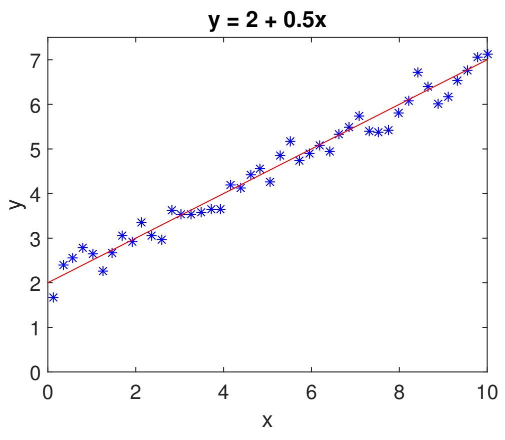


Figure 2.4: A linear fit to some data, where the bias unit shifts the prediction line, so that it does not pass through the origin.

to the term *deep learning*. These systems are used in a wide variety of applications, among them, face detection and speech recognition systems and in self-driving cars.

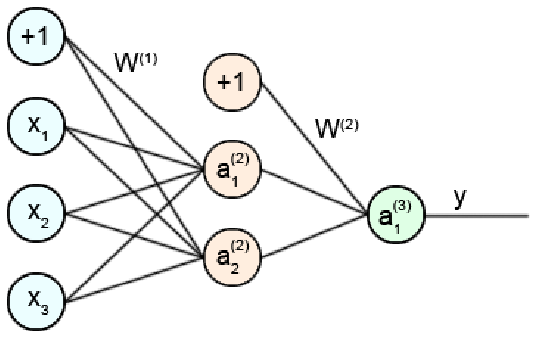


Figure 2.5: The neural network model

The neural network in 2.5 has one *hidden layer* between the input and output layer. A bias unit has been added to both the input and the hidden layer, as the hidden layer serves as an input to the top layer. Here the activation of neuron i in layer l is referred to as $a_i^{(l)}$ and is given by:

$$a_i^{(l)} = g(z_i^{(l)})$$

where

$$z_i^{(l+1)} = w_{i1}^{(l)} a_0^{(l)} + w_{i2}^{(l)} a_1^{(l)} + \cdots + w_{i(n+1)}^{(l)} a_n^{(l)}$$

The vector $\mathbf{a^{(l)}}$ denotes the activations in layer l. For convenience, we redefine the input vector $\mathbf{x}$ as the first activation map in the network

$$\mathbf{a^{(1)}} = \begin{bmatrix} 1 \\ x_1 \\ \vdots \\ x_n \end{bmatrix} \in \mathbb{R}^{n+1}$$

For a layer with n inputs and m outputs we define the weight matrix

$$\mathbf{W^{(l)}} = \begin{bmatrix} w_{11}^{(l)} & w_{12}^{(l)} & \cdots & w_{1(n+1)}^{(l)} \\ w_{21}^{(l)} & w_{22}^{(l)} & \cdots & w_{2(n+1)}^{(l)} \\ \cdots & \cdots & \ddots & \vdots \\ w_{m1}^{(l)} & w_{m2}^{(l)} & \cdots & w_{m(n+1)}^{(l)} \end{bmatrix} \in \mathbb{R}^{mx(n+1)}$$

The mapping from input to output through a layer is called *forward propagation*, and this can be expressed compactly as:

$$\mathbf{a^{(l+1)}} = g(\mathbf{W^{(l)}a^{(l)}})$$

2.1.1 Backpropagation

Neural networks are initialized with random weights on the connections. For a network to compute something interesting, it needs to be *trained* to do so. This is done by minimizing a cost function with respect to the weights in the network. Consider the *quadratic cost function* or *mean squared error*:

$$J(\mathbf{x}; \mathbf{W}) = \frac{1}{2}(\mathbf{y} - \mathbf{a^{(l)}})^2$$

The cost J is a function of the input $\mathbf{x}$, given the current weights in the network $\mathbf{W}$. We have a desired output from the network $\mathbf{y}$, called the *target vector*, for this particular $\mathbf{x}$. The target vector is a $k \times 1$ vector, where k is the number of outputs in the top layer (k is equal to 2 for the network in 2.3).

The cost function quantifies the error in the network by comparing the output to the target vector. The way each weight in the network affects the cost is given by the partial derivatives of the cost function with respect to the weights. These partial derivatives, or *the gradient* can be used to update the weights in order to minimize the cost. This is done with an optimization algorithm called *gradient descent* and the weights are updated by taking a step of length η in the direction of steepest descent as defined by the gradient.

$$\mathbf{W} \mapsto \mathbf{W} - \eta \frac{\delta J}{\delta \mathbf{W}}$$

The gradient matrix for the weights in layer l is given by

$$\frac{\delta J}{\delta \mathbf{W^{(l)}}} = \begin{bmatrix} \frac{\delta J}{\delta w_{11}^{(l)}} & \frac{\delta J}{\delta w_{12}^{(l)}} & \cdots & \frac{\delta J}{\delta w_{1(n+1)}^{(l)}} \\ \frac{\delta J}{\delta w_{21}^{(l)}} & \frac{\delta J}{\delta w_{22}^{(l)}} & \cdots & \frac{\delta J}{\delta w_{2(n+1)}^{(l)}} \\ \cdots & \cdots & \ddots & \vdots \\ \frac{\delta J}{\delta w_{m1}^{(l)}} & \frac{\delta J}{\delta w_{m2}^{(l)}} & \cdots & \frac{\delta J}{\delta w_{m(n+1)}^{(l)}} \end{bmatrix} \in \mathbb{R}^{m\times(n+1)}$$

It is not apparent how the weights in the hidden layers of a network affect the output. Each neuron connects to every other neuron in the layer above it, and these neurons may in turn connect to a new set of neurons. To find the gradients wrt. the weights in all layers we use the *backpropagation algorithm*[12]. The algorithm works by backpropagating error-terms, δ's, from the top layer, backwards through the layers in the network. This is done by applying the chain rule.

$$\frac{\delta}{\delta \mathbf{W^{(l)}}} J(\mathbf{x};\mathbf{W}) = \frac{\delta \mathbf{z^{(l+1)}}}{\delta \mathbf{W^{(l)}}} \frac{\delta \mathbf{a^{(l+1)}}}{\delta \mathbf{z^{(l+1)}}} \frac{\delta}{\delta \mathbf{a^{(l+1)}}} J(\mathbf{x};\mathbf{W})$$

For a network with L layers, we start by backpropagating through the squared error cost function and obtaining the first error-term:

$$\frac{\delta}{\delta a_j^{(L)}} J(\mathbf{x};\mathbf{W}) = a_j^{(L)} - y_j = \delta_j^{(L,costfunction)}$$

Thereafter the error is propagated further back through the activation function, in this case the sigmoid:

$$\frac{\delta a_j^{(L)}}{\delta z_j^{(L)}} = \frac{\delta}{\delta z_j^{(L)}} \frac{1}{1+e^{-z_j^{(L)}}} = \frac{e^{z_j^{(L)}}}{(e^{z_j^{(L)}}+1)^2} = \frac{1}{1+e^{-z_j^{(L)}}} \left(1 - \frac{1}{1+e^{-z_j^{(L)}}}\right)$$

$$\Rightarrow$$

$$\delta_j^{(L,activation)} = \sigma(z)(1-\sigma(z))$$

Finally we obtain the gradient of z wrt. the weights:

$$\frac{\delta z_j^{(L)}}{\delta w_{jk}^{(L-1)}} = a_j^{(L-1)}$$

The gradient of the cost function wrt. a weight in the top layer of a network can now be expressed as

$$\frac{\delta J}{\delta w_{jk}^{(L-1)}} = a_j^{(L-1)} \delta_j^{(L,activation)} \delta_j^{(L,costfunction)}$$

For networks with more than one layer we need to find the gradient of the cost function with respect to the lower level weights as-well. This is done by propagating the error-terms further down the network through the weighted connections in the layers above.

$$\frac{\delta J}{\delta w_{jk}^{(L-2)}} = \frac{\delta z_j^{(L-1)}}{\delta w_{jk}^{(L-2)}} \delta_j^{(L-1,activation)} \frac{\delta z_j^{L}}{\delta a_{jk}^{(L-1)}} \delta_j^{(L,activation)} \delta_j^{(L,costfunction)}$$

where

$$\frac{\delta z_j^{L}}{\delta a_j^{(L-1)}} = w_{jk}^{(L-1)}$$

Neural networks can become very large, and when they do it is practical to think of them as *computational graphs.* The computational graph is built up of many small units, where each unit is responsible for computing an output based on the inputs to the unit and the gradient of the output wrt. the inputs. Consider the unit in figure 2.6, which computes the output of a neuron with two inputs and no activation function. During forward propagation, this unit computes the output z as a function of a and w and also the "local gradient" of the output wrt. the inputs. These local computations are also done by the other units above this particular unit in the computational graph, and at the very top of the network some unit is computing the gradient of the cost J wrt. its inputs. During backpropagation, this gradient gets propagated down through the units and eventually ends up at the output of the unit in figure 2.6. With the local gradients already computed, this unit simply applies the chain rule to the incoming gradient and passes it further down the network.

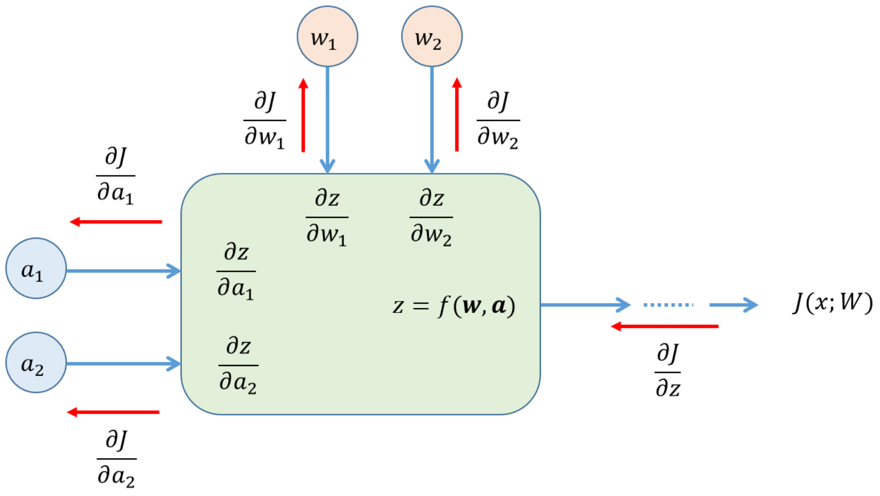

w_1
w_2
$\frac{\partial J}{\partial w_1}$
$\frac{\partial J}{\partial w_2}$
$\frac{\partial J}{\partial a_1}$
$\frac{\partial z}{\partial w_1}$
$\frac{\partial z}{\partial w_2}$
a_1
$\frac{\partial z}{\partial a_1}$
$z = f(\mathbf{w}, \mathbf{a})$
$J(x; W)$
a_2
$\frac{\partial z}{\partial a_2}$
$\frac{\partial J}{\partial z}$
$\frac{\partial J}{\partial a_2}$

Figure 2.6: A unit in a computational graph

A unit defined with a *forward* and *backward pass* can be used in a computational graph. However, one has to be mindful of the effect a unit has on the flow of gradients, especially when designing deep networks. Consider the *tanh* activation function in figure 2.7. This function limits the output of a neuron to a number between -1 and 1. If the input to the tanh is less than about -4 or greater than 4, the gradient of the function is effectively zero. When this happens the neuron is said to be *saturated*. If neurons in a layer near the top of the network gets saturated, it will block the gradient from flowing further down the graph and the lower level weights will not get updated. This is the called the problem of *vanishing gradients*.

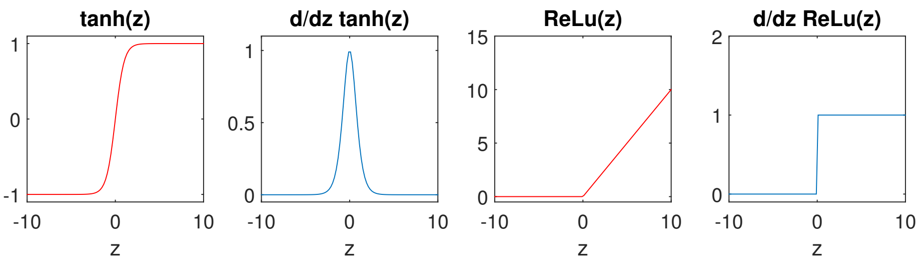


Figure 2.7: The tanh, its derivative and the ReLu and its derivative

By plotting the the histograms of the neurons in each hidden layer we can visualize how our input data is being forwardpropagated through the network. Consider an example network with ten layers taken from [13]. In this example, unit gaussian input data was given to a network with weights initialized randomly, but with values that were too large. If the weights in a layer are initialized with too large values, the tanh is likely to get saturated and we may get a histogram that looks something like 2.8. Every histogram shows the number of neurons (y-axis) with a given value (x-axis) in a specific layer. As is shown, almost all neurons in every layer are either -1 or 1.

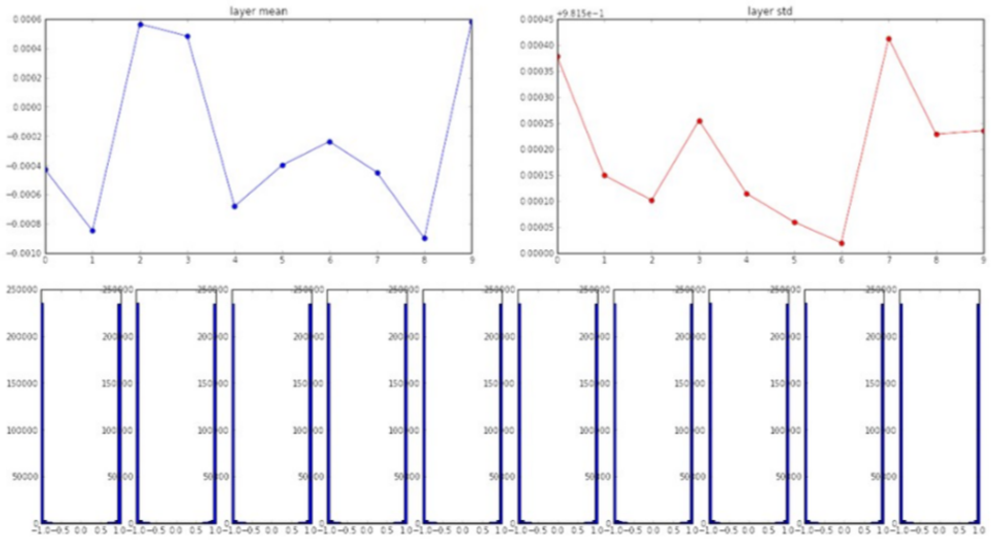


Figure 2.8: The mean (upper left), standard deviation (upper right) and histograms (bottom) of the layers in a neural network with unit gaussian input data, when the weights are initialized with values that are too large [13].

As figure 2.8 shows, too large weights will lead to saturated neurons throughout the network and no gradients will flow. Similarly, if the weights are initialized with values that are too small, the problem of vanishing gradients persists. In figure 2.9, we can see that the standard deviation for the activations in each layer goes towards zero as we forwardpropagate up through the network. Even though the the tanh is as far from saturated as can be, no gradient will flow through the network. For the top layers, this is because the gradient of $z_j^{(l)}$ wrt. $w_{jk}^{(l-1)}$ is equal to the activation $a_j^{(l-1)}$, as was shown earlier in this chapter. The weights of the top layers will therefore not be updated before the activations from the lower layers have a higher absolute value. However, because the derivative of $z_j^{(l)}$ wrt. $a_j^{(l-1)}$ is given by the weights $w_{jk}^{(l-1)}$, which are small, the gradients flowing from the top will go towards zero before reaching the lower layers. Therefore, the cost for a poorly initialized network might not change at all with training.

To combat the vanishing gradient problem some initialization strategies have been proposed. In 2010, Glorot et al. proposed an initialization scheme called the *Xavier initialization* which works well for networks with tanh activations, and in 2015, He et. al expanded on this work for ReLu units [14, 15]. They propose an initialization that keeps information flowing during forward propagation by setting the weights so that the variance of the output from all layers stay the same. Similarly, to keep the gradients flowing during backpropagation, the weights need to be set so that the partial derivative of the cost wrt. the activations remains unchanged. The histograms for the layers after forward propagating unit gaussian data through a network with tanh activations initialized with the Xavier method is shown in figure 2.10.

The goal of the Xavier initialization is to make the input to all layers distributed

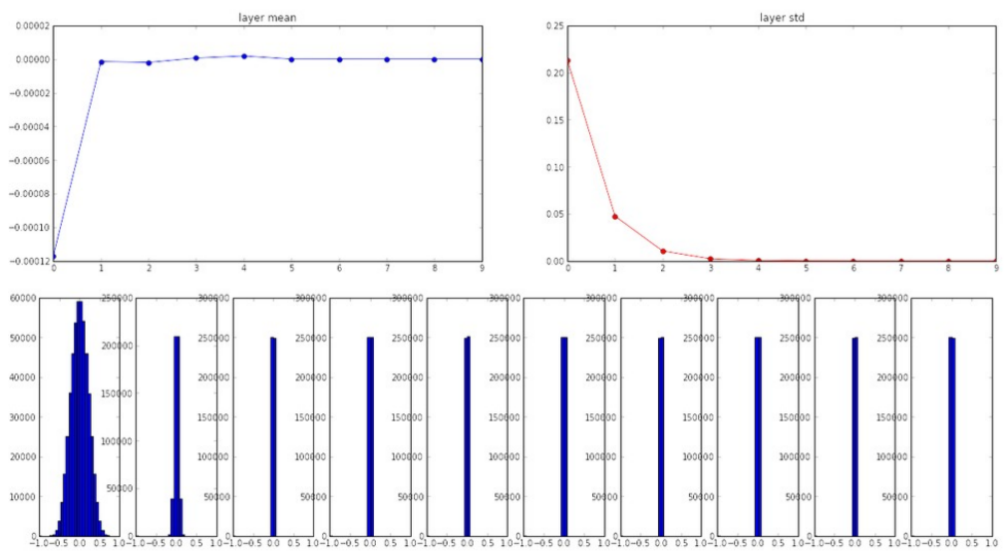


Figure 2.9: The mean (upper left), standard deviation (upper right) and histograms (bottom) of the layers in a neural network with unit gaussian input data, when the weights are initialized with values that are too small [13].

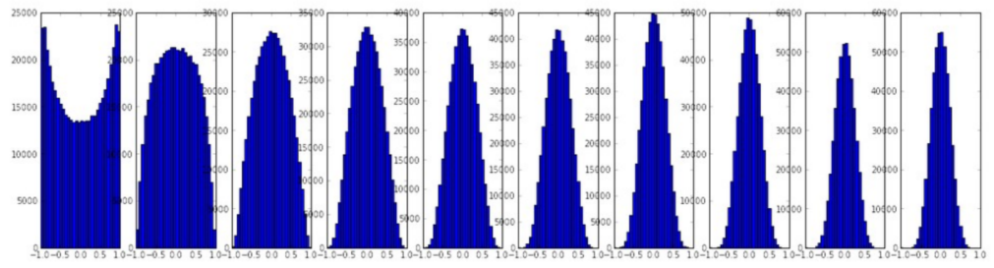


Figure 2.10: The histograms of the layers in a neural network with unit gaussian input data and weights initialized with the Xavier method [13].

as a unit gaussian. Another way of achieving this is to normalize the input to each layer with a *batch normalization layer*[16]. For each input to a layer, the mean and variance is computed across all (or a subset of) the data in the data set. The new input to the layer is computed by subtracting this empirical mean and dividing by the variance.

$$\hat{x}_j = \frac{x_j - E\{x_j\}}{\sqrt{Var\{x_j\}}}$$

These layers are differentiable and can therefore be placed after any layer in the network and gradients will be propagated through them. However, this introduces a constraint that might not be desirable, e.g. for a sigmoid layer the inputs are constrained to the linear region of the activation function. To solve this problem the network is given the chance to undo the batch-norm layer by introducing two learnable parameters.

$$y_j = \gamma_j \hat{x}_j + \beta_j$$

We can see that the layer has no effect if the parameters are learned so that $\gamma_j = \sqrt{Var\{x_j\}}$ and $\beta_j = E\{x_j\}$. These layers improves the flow of gradients through the network and thus reduces the need for good initialization schemes. They also accelerate learning in general.

In 2012, Krizhevsky et al. showed that a four-layer convolutional neural network with *rectified linear units* (ReLus) converged six times faster compared to the same network with tanh activations [17]. The ReLu and its derivative are shown in figure 2.7. The ReLu does not saturate, and in the active region, the gradient is equal to one so that gradients from the top flow unchanged through it. If the ReLu receives an input that is less than zero however, the derivative is zero and no gradient will flow. This means that a ReLu neuron that outputs a value of zero for all examples in the data set never will be updated. These *dead ReLus* may occur if the step size is too high during training and the weights accidentally get updated so that the input to the neuron is never positive again.

Neural networks can be used for both regression and classification problems. In image classification tasks, the network is trained to recognize certain objects, such as cars, motor cycles, horses etc. The target vector for an image of a motor cycle given to a network designed to recognize 4 classes, would look like this:

$$\mathbf{y} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \Rightarrow \begin{matrix} \text{horse} \\ \text{motor cycle} \\ \text{car} \\ \text{none} \end{matrix}$$

Each training example in the data set has a corresponding target vector where only one entry has a value of 1, the others are 0. The most common activation function for the top layer in a classification network is a modified version of the sigmoid called the *softmax*.

$$g(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}$$

Like the sigmoid, the softmax squashes the outputs to a number between 0 and 1, in addition to upholding an output vector norm of 1. The predicted class for an image is given by the entry in the output vector with the largest value.

$$\mathbf{a^{(L)}} = \begin{bmatrix} 0.025 \\ 0.850 \\ 0.100 \\ 0.025 \end{bmatrix} \Rightarrow \text{motor cycle}$$

The most common cost function for training classification networks is the categorical cross entropy function. For a single neuron with n inputs and a sigmoid activation function it is defined as:

$$J_{CE} = -\frac{1}{n}\sum_{x}(yln(a) + (1-y)ln(1-a))$$

where, as before, $a = \sigma(\mathbf{Wx})$, and $\mathbf{W}$ is a row-vector of length n. It can be shown[18] that the derivative of the cost wrt. the weights is given by:

$$\frac{\delta J_{CE}}{\delta w_j} = \frac{1}{n}\sum_{x} x_j(\sigma(z) - y)$$

Intuitively, we see that larger errors in the output (differences between the the network output, $\sigma(z)$, and the target, y) yields larger gradients. This is not the case for the squared error cost function. As was shown earlier in this chapter, the derivative of the squared error cost function wrt. a weight is given as:

$$\frac{\delta J_{SqE}}{\delta w_j} = (a - y)\sigma'(z)x_j$$

The derivative of the sigmoid function, $\sigma'(z)$, in the gradient is undesirable, because it never takes on a value larger than 0.25, and thus inhibits the flow of gradients and slows down the learning process (see figure 2.2). However, for an output layer with a linear activation function, the gradient of the squared error cost wrt. the weights is given by:

$$\frac{\delta J_{SqE}}{\delta w_j} = (a - y)x_j$$

Linear activation functions are used when neural networks are trained on target vectors consisting of real valued numbers. The squared error cost is therefore well suited for regression tasks.

2.1.2 Training neural networks

When training neural networks, the data sets are commonly split into three sets: A training set, cross validation set and test set. Each set consists of m_{set} example input vectors, each of length n, and m_{set} corresponding target vectors of length K. The different sets are not necessarily of equal size.

$$\{(\mathbf{x}^{(1)}, \mathbf{y}^{(1)}), (\mathbf{x}^{(2)}, \mathbf{y}^{(2)}), \cdots, (\mathbf{x}^{(m_{set})}, \mathbf{y}^{(m_{set})})\}, x \in \mathbb{R}^n, y \in \mathbb{R}^K$$

The training set is used to update the weights while training the network. The cross validation (CV) set is used while training to see how well the network generalizes to data not seen during training. The examples in the CV-set are forward propagated and the cost is calculated, but no weight updates are done. If the cost on the CV-set is much higher than the cost on the training set, this is a sign of *overfitting*, and the network is only good at separating the specific examples in the training set e.g. those images of cars and horses, not necessarily images of cars and horses in general. The test set is only used to see how well the designed network is performing on data never seen before. While the CV set can be used to tune parameters of the network, the test set cannot.

As described in 2.1.1, neural networks can be trained with gradient descent, with the update rule:

$$\mathbf{W} \mapsto \mathbf{W} - \eta \nabla_{\mathbf{W}} J(\mathbf{W}; x^{(i)}, y^{(i)})$$

For regular *batch gradient descent*, the gradients from all training examples in the training set are summed up before one step with *step size*[2] η is taken in the direction of steepest descent. This becomes impractical for data sets of the sizes common in deep learning. In order to take one small step in the direction of the gradient one would have to propagate the many thousands of examples forward and backward

[2]Also called learning rate (lr)

through the network. In practice therefore, the data sets are split up into *mini-batches* randomly sampled from the training set and the weights are updated with the gradients found from these few examples. This is called *mini-batch stochastic gradient descent* (SGD) and typical batch sizes vary from 32-256 examples and are largely determined by the capacity of the GPU (more about this in 2.1.4).

Because of the stochastic sampling from the data set, SGD can keep the optimization from getting stuck in local minima, but for the same reason SGD can converge rather slowly. An illustration of how SGD steps in the direction of a gradient is shown in figure 2.11.

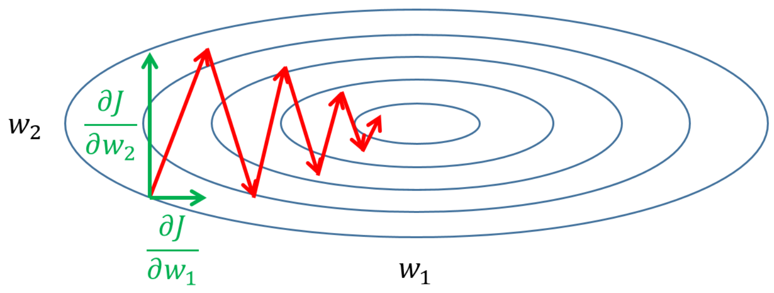

w_2
$\frac{\partial J}{\partial w_2}$
$\frac{\partial J}{\partial w_1}$
w_1

Figure 2.11: SGD oscillating down the gradient spanned by the parameters w_1 and w_2. Each red arrow illustrates one step of SGD.

For SGD to converge faster it is common to add *momentum* to the update rule[12].

$$\mathbf{v} \mapsto \gamma\mathbf{v} + \eta\nabla_{\mathbf{W}}J(\mathbf{W}; x^{(i)}, y^{(i)}), \gamma \in [0, 1]$$

$$\mathbf{W} \mapsto \mathbf{W} - \mathbf{v}$$

The weights are updated with a combination of the gradient and the previous update. Intuitively, by looking at figure 2.11, we can see that the $\frac{\delta J}{\delta w_2}$ terms will sum to zero, while the $\frac{\delta J}{\delta w_1}$ terms will sum up in the direction of the minimum. This ensures that each step is taken in a more direct path towards the minimum. The hyper parameter γ is a number between 0 and 1 (usually set to about 0.9) and it ensures convergence by acting like friction on the surface. A modified version of the momentum update, which ensures even faster convergence is called *nesterov momentum* (NAV). We know that the weight update will be a sum of the momentum vector and the gradient vector. NAV works by taking a one step look-ahead based on the momentum and evaluating the gradient at this point before taking a step, as shown in figure 2.12. The derivation of nesterov momentum is appended in A.3.

An important part of training neural networks consists of preventing overfitting of the training data. This is characterized by low errors on the training set, but large errors on the test and CV sets, i.e. data never seen before. Neural networks are capable of learning highly complex non-linear functions of the input, and for this reason they may be able to distinguish the individual training examples in the training set from each other, rather than capturing the concepts they represent.

There are many ways of *regularizing* neural networks, e.i. combating overfitting. One of them is called dropout[19]. Dropout is based on the idea that many separately trained models combined, generally improves machine learning methods. The different models should be trained on different subsets of the data or have different architectures. Dropout works by removing a randomly selected subset of the neurons

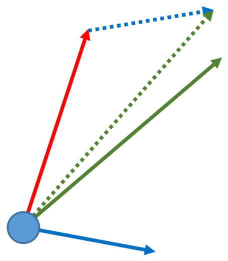


Figure 2.12: Gradient vector (blue), momentum vector (red), momentum update vector (green), NAV gradient vector (dashed blue), NAV update vector (green dashed)

during training as shown in figure 2.13. For each forward pass the probability of a neuron being present is set by the parameter ρ. During backpropagation, only the weights contributing to the output are updated. Thus, training a neural network with dropout can be viewed as training many "thinned" versions of the networks.

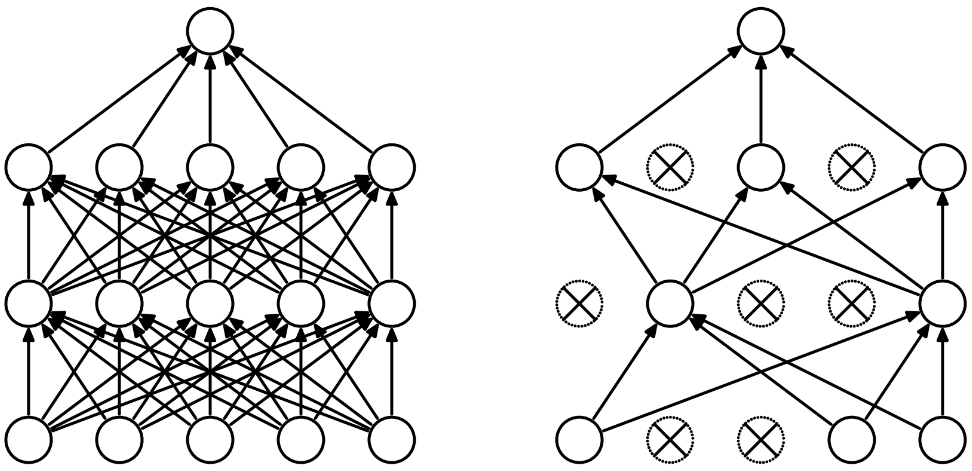


Figure 2.13: An ordinary neural network (left) and the same model with some randomly selected neurons dropped during training (right) [19]

At test time, the prediction of the network is given by averaging the prediction from each one of the thinned networks. A simple approximation of this average is given by weighing each weight with the constant ρ as shown in figure 2.14. Intuitively, dropout forces each neuron to become more robust with regard to its inputs and to learn more useful features on its own, as it cannot co-adapt with other neurons as easily.

Batch normalization, as described in 2.1.1 also works as a regularizer. This is because the input to a neuron is a function of the other examples that also happens to be present in the mini-batch. Because the examples that are in each mini-batch are randomly drawn, this introduces noise which has a regularizing effect that reduces the need for dropout.

In general the need for regularization is there because we have a finite amount of training data. To illustrate the effect of big data sets, a neural network was trained on the MNIST data set of handwritten digits[3][20]. This is a classification problem

[3]This data set was used as a bench mark for many years, but has become less common because

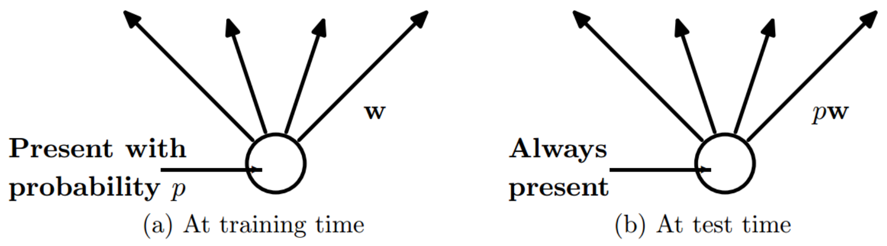


Figure 2.14: During training, each neuron has a probability ρ of being dropped. At test time, all neurons are active and the weights on the connections are set to ρw [19].

where the goal is to separate the digits 0-9 in ten different classes. As is shown in figure 2.15, the training and cross validation costs follows each other closely when the entire training set of 50 000 examples is used. In fact the CV cost is lower because dropout was used during training with randomness introduced for the training set, but not for the CV set. When the number of training examples is reduced to 10 000, the costs diverge more at the end, but the CV cost keeps decreasing with time. With only 500 examples, however, the CV cost turns around and starts increasing and we have a case of severe overfitting.

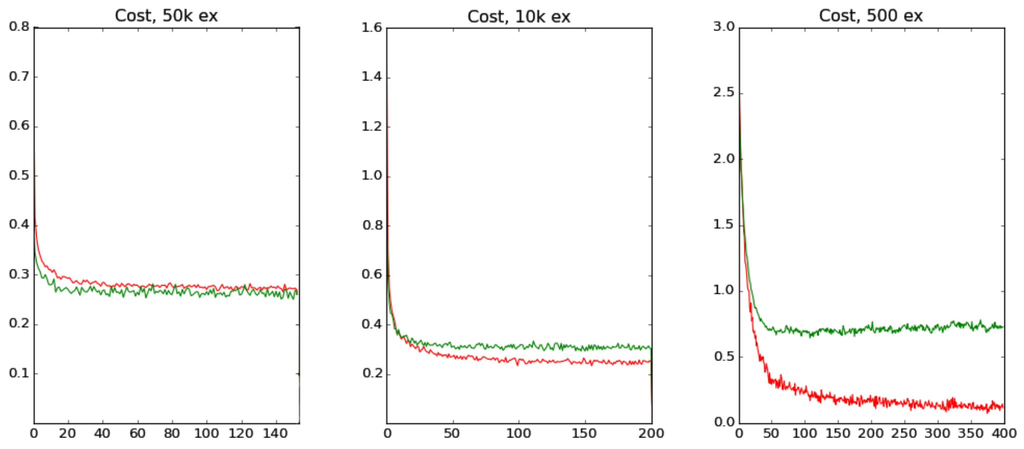


Figure 2.15: The effects of different training set sizes. Cost vs. epocs during training for 50k (left), 10k (middle) and 500 examples (right). The training cost is plotted in red and the CV cost is green.

To meet the data needs of deep learning it is common to create more training examples by modifying the examples in the training set. For images this could mean rotating them by some amount, mirroring and translating (cropping). In addition, some stochastic noise is often added to the image before it is fed to the network.

it is deemed to be too easy for convolutional NNs. With error rates as low as 0.23 %, it is now regarded by many as the "hello world" example for deep learning.

2.1.3 Neural networks for images

The layers used in the neural networks described this far have taken one dimensional vectors as inputs. These types of layers are called *dense layers* of *fully connected layers.* Networks consisting of only fully connected layers can be used with images by unrolling the image matrix into a vector before feeding it to the network. However, much better results can be achieved with *convolutional neural networks* (CNNs).

Regular fully connected neural networks are unpractical to apply on images because of the need for a large number of parameters. Consider an RGB image of size $28 \times 28 \times 3$. This image could be unrolled to a 1D vector of length 2352 and used as input to the neural network. Every neuron in the second layer of the network would then have 2353 weights (accounting for the bias) and we would like to have many such neurons in the layer. It is clear that this does not scale well to larger images and deeper nets. Convolutional neural networks reduces the number of parameters needed by having the neurons in a layer share weights.

CNNs are inspired by Hubel and Wiesel's work on the visual cortex of the cat, where they showed that specific neurons in the brain reacts to specific features within the cat's field of view[21]. I.e. one neuron fires when an image of a horizontal line is presented and another fires when a slightly angled line is presented. By assuming that these *edge detectors* are useful feature extractors in the entire image, that is, detection of e.g. horizontal lines is interesting in all parts of the image, this can be implemented with *convolution*[22].

2D filters can be designed to react strongly on certain edges in a grayscale image. An example is the 3×3 Sobel operator for detection of vertical edges:

$$\mathbf{G} = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

By filtering an image with the kernel $\mathbf{G}$ the result is an image with high pixel values in areas corresponding to sharp vertical edges in the original image, see figure 2.16. Each pixel in the resulting image can be viewed as a neuron looking for a vertical edge in a specific position in the input image. Because convolution is a differential operation, the filter can be initialized randomly and learned through back propagation. For the example image in figure 2.16 with dimensions 256×256 the resulting image or *activation map* is of size 254×254. This means that we have 64516 neurons in the second layer, with only 9 weights!

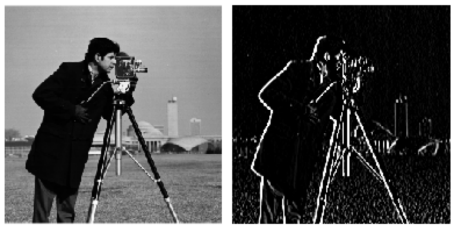


Figure 2.16: A test image convolved with the Sobel vertical edge detector.

In general, the input to a CNN is a volume with dimensions $M_i \times M_j \times D$. By convolving this volume with N_f filters with dimensions $N_i \times N_j \times D$ we get an activation map with dimensions $N_f \times (M_i - N_i + 1) \times (M_j - N_j + 1)$. Each "slice" in the activation map can be viewed as an image, where each pixel (i.e. neuron) is a function of a filter and a subset of the pixels in the original image. The subset of pixels in the input image which a neuron in the activation map is ”looking at” is called *the receptive field* of this neuron. The activation map may in turn be used as an input to another layer in the network as shown in figure 2.17.

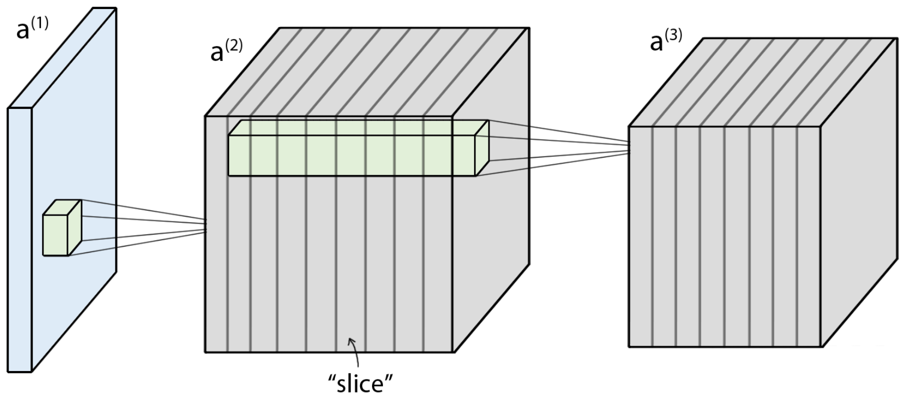


Figure 2.17: All neurons in a slice in the activation map $a^{(2)}$ are the result of filtering a part of the input image $a^{(1)}$ with the same filter. Thus, the depth of the activation map is equal to the number of filters in the layer. The resulting activation map can in turn be used as an input to the next convolution layer. The depth of the filter in layer two is given by the number of filters in the previous layer.

As the networks become deeper, neurons in higher layers starts to combine the lower layer edge detectors into more complex feature extractors. If a classifier is trained to recognize faces, some neurons will get excited when they see eyes and some when they see eyebrows, mouths, noses etc. The receptive field of the neurons in the higher layers is larger than the receptive field for the neurons in the lower layers. This is because every pixel in an activation map is the result of filtering several pixels in the previous activation map.

To introduce robustness to deformation and translation in the image *max pooling layers* are commonly used. Max pooling is a form of non-linear down sampling which reduces the size of the activation maps. It works by dividing an activation map up into non-overlapping regions and discarding all but the highest value in each region as shown in figure 2.18. It works over the spatial dimensions and leaves the depth of the volume intact.

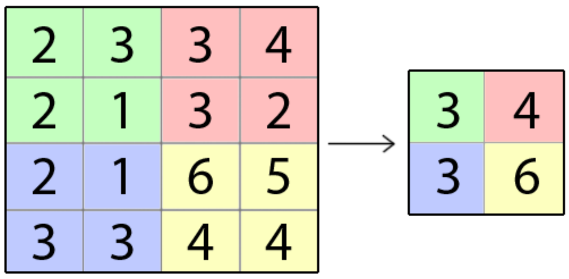


Figure 2.18: An example of 2 × 2 max pooling

Consider the example of recognizing a face in an image. Intuitively it does not matter if two eyes are detected with 30 pixels or 31 pixels in between them, the image should still be classified as a face. Max pooling provides invariance to such translations, but at the same time, higher layers lose information about precise spatial positions in the image. There are other forms of pooling and subsampling, but max pooling have been shown to yield superior results on object detection tasks in practice [23, 24].

In classification tasks, ordinary dense layers are often used as top layers. However, the classification networks can be expanded and used on the more general object detection problem by swapping the fully connected layers with 1×1 convolution layers [25]. An example of this is shown in figure 2.19. This is possible because the dot product of two vectors essentially is the same as the valid convolution of the vectors[4].

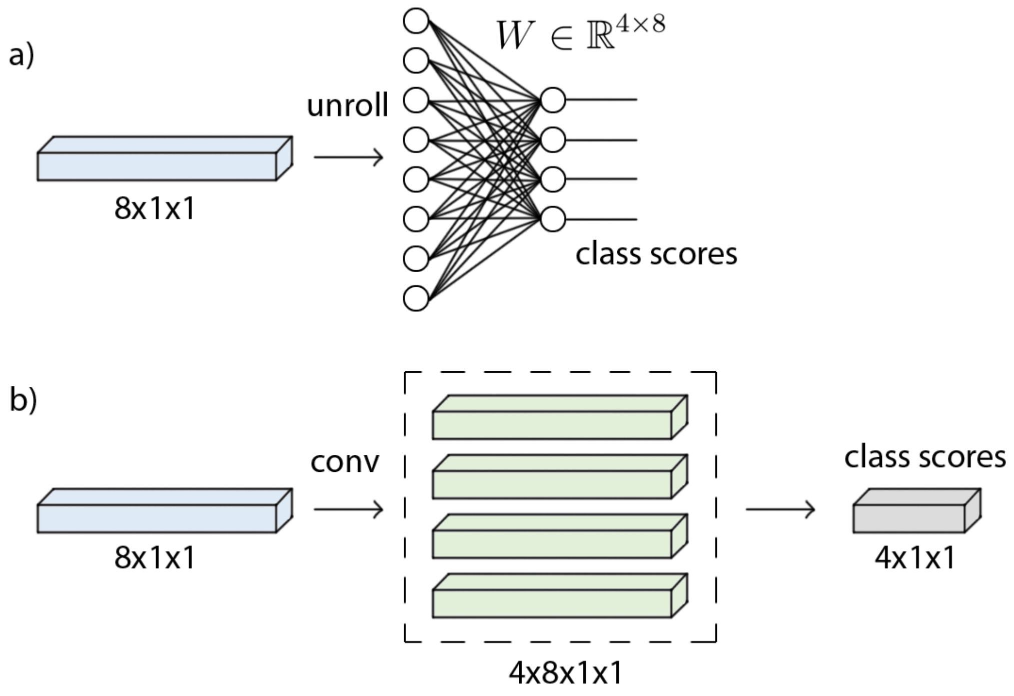


Figure 2.19: a) Unrolling an activation map into a vector before adding a dense layer. b) The equivalent network architecture realized with 1 by 1 convolutions.

The advantage of networks consisting only of convolutional layers is that they can be applied to images of (almost) arbitrary size. A 1×1 convolution can be viewed as a *sliding dense layer*, and if the input to this layer has a 1×1 spatial dimension, the output will simply be a set of class scores. However, if the input has a larger spatial dimension, e.g. 2×2, the output will be also have a spatial dimension of 2×2 and a depth corresponding to the number of classes. The output can thus be viewed as a set of *probability surfaces*, where each pixel in the output corresponds to a different receptive field in the input image. This is shown in figure 2.20.

CNNs reduces the number of parameters in the networks dramatically, which in turn reduces the need for data. However, recent work has shown that deeper

[4]In practice, because of the definition of convolution, one of the vectors will have to be flipped before convolving. The valid cross correlation of the two vectors is identical to the dot product. When networks are trained directly with 1×1 convolutions this is not a concern, it is only an issue if a fully connected layer has to be converted to a convolution layer post hoc.

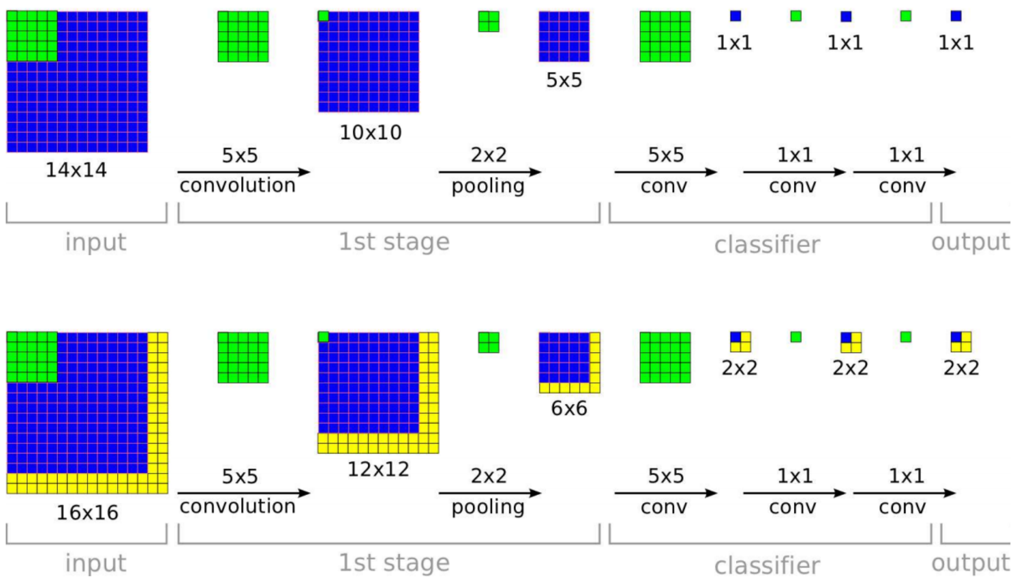


Figure 2.20: Neural networks with sliding classifiers as proposed by Sermanet et. al in [25].

networks outperform shallow nets, and among previous winners of the ImageNet challenge, we have AlexNet from 2012 and the VGG net from 2014 which have a total of 61 and 138 million parameters respectively [26, 17]. Some winners have gotten away with fewer parameters, like GoogLeNet in 2014 with only 4 million parameters [27].

In any case, the need for large amounts of data is still an issue, and neural networks are far from able to capture the essence of an object, say, a tomato, after seeing just one example, as humans can. The popular ImageNet classification challenge with 1k classes consists of 1.28 million training images and 50k evaluation images.

In practical applications it is often unfeasible to gather such large amounts of labelled data. One way of solving this problem is to pre-train the model on some other labelled data before fine tuning it on a smaller task spesific data set. This is called *transfer learning*. For instance, if a network is to be used on color images, the network could be pre-trained on the ImageNet data set before training is begun on the smaller data set. If the data set available is very small, the lower level filters can be fixed, and only the top layers fine-tuned with the task spesific data.

Neural networks can also be pre-trained *unsupervised* with the use of an *autoencoder*. This is a training technique that can reduce the need for data by ensuring that the the filters in every layer extracts useful information before training on the labelled data set is begun. This is achieved by training a network to replicate its input on the output with some constraints on the hidden layer. With an autoencoder, the label corresponding to an image is the same image itself and virtually infinite amounts of data can be gathered by simply taking a lot of images or downloading relevant images from the web. Some experiments with unsupervised pre-training has shown that it might help the the networks avoid local minima [28]. However, recent progress in backpropagation of errors in deep networks with good initialization schemes, rectified linear units and batch-norm, has reduced the need for unsupervised training if one has enough data.

2.1.4 Libraries and GPUs

As discussed in chapter 2.1.1, neural networks are best viewed as computational graphs, where each unit in the graph needs to have defined a forward and backward pass. There exists several frameworks for construction of such graphs with pre-defined units. One such framework is called Theano. Theano is a library for symbolic differentiation in python. With Theano, one can simply define the forward pass of the entire network and a cost function, and thereafter find the gradient of the cost wrt. the weights with a function call. This greatly simplifies prototyping with complex neural networks. Another framework for even simpler prototyping is called Lasagne. Lasagne is built on Theano and comes with pre-defined layers such as convolution, batch-norm and pooling, as well as different regularization techniques and cost functions. The graphs constructed in Theano and Lasagne can be optimized and compiled to run on the GPU for faster training.

2.2 Virtual reality

The goal of virtual reality (VR) is to create immersive digital environments and replace the old ways of interacting with machines with keyboard's and mice. The field has enjoyed a renewed interest in the last couple of years largely because of two new head mounted displays (HMDs), the Oculus Rift and the HTC Vive. Technological advancements, driven to some degree by smart phones, has lead to cheaper mass produced OLED displays and accelerometers, making it possible to produce HMDs of good quality for a wider audience. The head mounted displays combined with motion tracking and hand controllers provide an intuitive way of communication with machines. Figure 2.21 shows a player wielding a bow and arrow with two controllers, while defending a castle from intruders with the HTC Vive. With VR head sets it is possible to walk around in any digitally created world and interact with objects in an intuitive way by reaching out and touching them.

Figure 2.21: A player defending a castle with a bow and arrow in "The Lab" developed by Valve Corporation with the HTC Vive. What the player sees has replaced the real background using a green screen [29].

Game development in general has become easier for everyone, thanks to free game engines such as Unity [4] and Unreal [30]. These engines support VR, and this opens up for the possibility of creating and exploring one's own virtual worlds. When developing games with game engines such as these, one can make use of the built-in physics, and spend more time on the actual gameplay and design of the game.

Chapter 3

Deep learning for grip point and grip vector estimation

To detect objects and find good grip points and grip vectors for each one, a convolutional neural network (CNN) is proposed. The input to the network is a depth image, and the system can be broken into three parts:

1. An object classifier/detector
2. A grip point estimator
3. A grip vector estimator

A depth image of arbitrary size is presented to the network and enters the object detector sub system as depicted in figure 3.1 a). The output of this network is a set of probability surfaces, one for each class the network has been trained on. E.g. one surface for recognizing hammers and another for recognizing knives. Every pixel in a given probability surface corresponds to a set of pixels in the input image i.e. the receptive field. If a neuron in one of the probability surfaces detects an object of interest, the receptive field of this neuron in the input image is fed to the grip vector sub system in fig. 3.1 b) and the grip point detector in fig. 3.1 c).

The grip vector sub system returns 6 numbers defining two 3D-vectors, dubbed the *palm vector* and the *pointer vector*. Together these two vectors define the orientation of an end effector, as shown in figure 3.2. The grip point detector returns a 3D vector defining the x, y and z- position of the grip point in camera coordinates.

To meet the data demands for CNNs, the network is trained on synthetic data created in a realistic virtual environment. After training, the whole network, or only the top-layers, may be fine-tuned with real data on site with an easy to use interface. This is discussed in chapter 4.

3.1 Depth images as input

As input to the network, a depth image from the Shapecrafter depth camera is proposed. Every pixel in a depth image can be viewed as an x, y, z-coordinate given in camera coordinates. This is not true for RGB images. If a good grasp is found in a color image, the task of mapping the pixels in the image to a precise position in the world still remains. The Shapecrafter camera outputs color information as well and

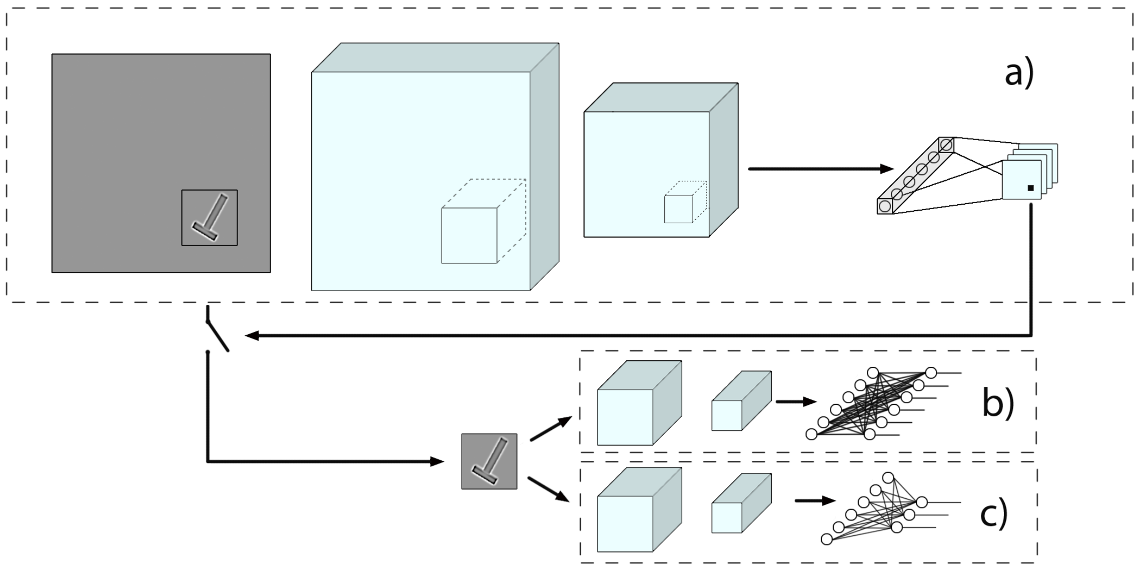


Figure 3.1: A depiction of the proposed system for object detection (a), grip vector estimation (b) and grip point estimation (c)

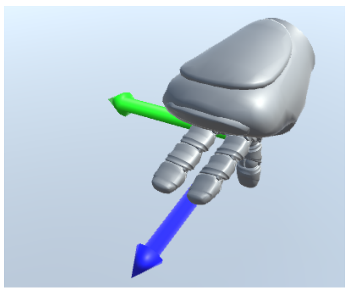


Figure 3.2: Two vectors define the end effector rotation, the palm vector (green) and the pointer vector (blue).

the proposed system can be expanded to utilize colors information also by changing the depth of the input volume from one "image" to four. I.e. one additional image per color channel and one for the depth image.

In reality, the pixels in the depth image do not correspond to real world coordinates directly. They are subject to distortion in the lens and perspective effects. Additionally, "3D-shadows" appear in the image because the camera works by projecting and recognizing patterned light from a projector that is offset by some amount from the camera. This is shown in figure 3.3. Most of the perspective and distortion effects can be eliminated by calibrating the camera and getting the point cloud, instead of the depth image, directly from the camera. The 3D-shadows are most often small and can possibly be eliminated by filling and low pass filtering the image.

Using only depth images without color makes it easier to create realistic looking synthetic data. Crucially, this eliminates the need for realistic textures, reflections, lighting etc. Only the geometry of the objects matter and this is comparatively easy to get right in a virtual environment.

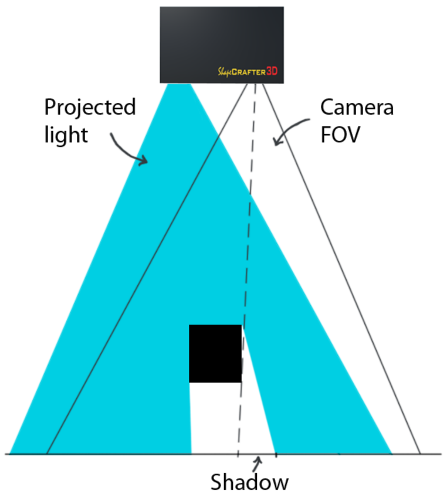


Figure 3.3: 3D shadows appear in areas the camera in the Shapecrafter can see, but the projector cannot.

3.2 Neural networks

The system for grip detection was as mentioned split into three sub systems. This was done to make experimenting easier. By decoupling each system and training them separately, it is easier to understand how pooling layers and number of filters per layer affect classification, grip point and vector estimation differently. However, decoupling of the grip point and grip vector estimator might not be ideal in a real world system because of their strong dependence on each other. Combining the grip point and grip vector network into one system is straight forward and can be done by defining a network with nine real valued outputs, instead of two separate networks with six and three outputs for the vector and point networks respectively.

The object detector was designed to recognize specific objects rather than as a binary classifier recognizing ”graspable” and ”not graspable” objects. Down the line this enables a robot to take different actions with different objects, e.g sort knives and hammers by putting knives in a box to the left and hammers in a box to the right.

The grip point and grip vector estimators are also class specific, meaning that a different set of weights is used to estimate grips for hammers, knives and berries. This does not scale well, and might not be necessary in practice. If the purpose of a robot is simply to move things, and precise gripping is unimportant in itself, then a single grip point/vector estimator might be plausible. Several estimators for specific types of objects, such as ”long thin objects”, ”round objects” etc. might also work, where the output with the most certain grip vector gets used.

3.2.1 Object detector

The object detector is trained as a classifier. Three different looking objects were chosen for recognition, a hammer, a knife and a strawberry. Therefore, the output of the classifier is a softmax layer with four outputs, one for each class and one for ”no object detected”. The input to the classifier is an image of size 100×100. Many different architectures were tested with different combinations of convolution, pooling, batch-norm and dense layers. As activation function, rectified linear units was chosen and dropout and L2-regularization was used to prevent overfitting.

Depth images have large mean values, which the classifier needs to be invariant to. Two pre-processing techniques are proposed to achieve this invariance: Zero mean- and unit variance normalization, and filtering of the image with a band pass filter.

3.2.2 Grip point and grip vector estimation

The output of the grip point and vector estimators are real valued numbers. For the grip point estimator, the output is a three dimensional vector defining a grasp's position in camera coordinates. This network is trained directly with regression and the squared error cost function.

The purpose of the grip vector estimator is to estimate the orientation of a grasp. The output is cyclic, e.i. an object rotated 360° around an axis should have the same grip vector as the object when it is not rotated.

A common way of defining 3D orientation is by the use of Euler angles. However, training directly on three values in degrees is not a good solution because the orientation is dependent on the order of the rotations and the discontinuity from 359° to 0° would be very hard for the network to capture. A better solution would be to use quaternions. However, because any rotation can be described by two quaternions, i.e. $q = -q$, the rotation was instead defined with two unit vectors. This ensures that the rotation is described uniquely and unambiguously[1] as seen in figure 3.2.

Training these vectors with the squared error cost is possible, but it does not emphasize the true goal, which is minimizing the angle between the vectors and their target vectors. Instead a cost function based on the dot product of vectors was defined, where $\hat{y}$ is the estimated vector and y is the target :

$$J(\mathbf{x};\mathbf{W}) = 1 - \frac{dot(\hat{y}, y)}{\sqrt{dot(\hat{y}, \hat{y}) * dot(y, y)}}$$

Unlike the classifier sub system, the grip point/vector estimator is dependent on the distance from the camera. Therefore no pre-processing was done on the images, instead the input is sent through a batch-norm layer before the first convolution layer. As described in chapter 2.1.1, with batch-norm the empirical mean of the whole data set is subtracted from every image, thus no depth information is lost.

[1]Training directly on the rotation matrix was also considered, but it is redundant because one of the axes is given by the two others.

3.2.3 Future improvements

The sub systems developed in this project were designed in order to ease testing of object detection, grip point and grip vector estimation separately. As argued above, the grip point and grip vector estimators could most likely be combined to one system. Additionally, the filters learned in the lower layers (and possibly also the higher layers) could probably be used for both classification and grip point/vector regression, and should therefore be shared. A depiction of the system with shared weights in the bottom layer and a combined grip point and grip estimator is shown in figure 3.4.

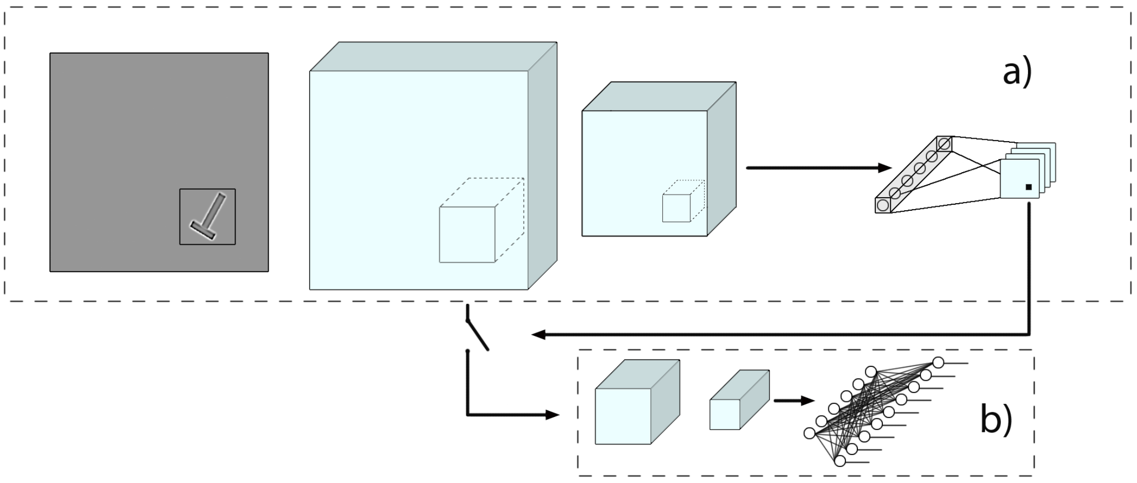


Figure 3.4: A depiction of the proposed system with weight sharing in the first layer and a combined grip point and grip vector sub system.

3.3 Pre-training in VR

To meet the data requirements for deep learning, the networks were trained on synthetic data created in a virtual environment. The virtual environment outputs precisely labelled depth images, and can be used to create large data sets in a short period of time. In future work, the parameters trained in VR will be fine-tuned with data from the real world.

If the parameters trained in VR are to be successfully transferred to a real world system, the statistics of the synthetic data has to be very similar to the statistics of the real data. If the synthetic data is good enough, the cost functions with both types of data, should be approximately equal, $J(\mathbf{X}_{\text{VR}}; \mathbf{W}) \approx J(\mathbf{X}_{\text{real world}}; \mathbf{W})$. Minimizing the cost with synthetic data should therefore yield weights that gets us close to a minimum in the cost with real data, as illustrated in figure 3.5. If the network is close to the "correct" minimum in $J(\mathbf{X}_{\text{real world}}; \mathbf{W})$ after training on synthetic data, relatively few real world examples could be used to fine-tune the network.

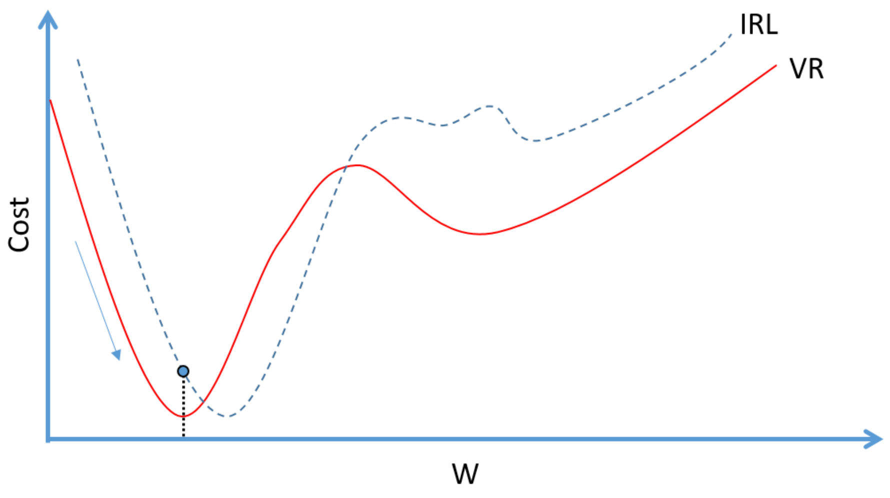


Figure 3.5: Illustrated cost functions wrt. the weights for real world data (IRL) and synthetic data (VR). If the functions are sufficiently similar to each other, the real world minimum will be close to the synthetic minimum. SGD can be performed on synthetic data till convergence, and later on real data to reach a good optimum.

The network could also be pre-trained using an autoencoder. Some initial experiments with autoencoders were done, but they were not used in the finished system due to the successful generation of large amounts of realistic looking synthetic data. As discussed in chapter 2.1.3, good initialization and training schemes have largely eliminated the need for unsupervised pre-training.

Chapter 4

VR environment for synthetic data generation and as an interface between man and machine

To meet the large data requirements for deep learning, a virtual environment for synthetic generation of data was created. The environment is based on the Unity game engine and in it, it is possible to instantiate objects with a set of pre-defined grip vectors and create depth images of these objects using a virtual depth camera. The objects are instantiated with random rotation, at a random distance from the ground. After instantiation these objects fall to the ground and land naturally with simulated physics. The best of the pre-defined grip vectors is chosen, and each object is scanned with the virtual camera. This is shown in figure 4.1. The output of the system is a set of depth images, with corresponding strings of information for each image. The information in the string is: The class of the object that is depicted, the grip point in camera coordinates and the grip vector in camera coordinates.

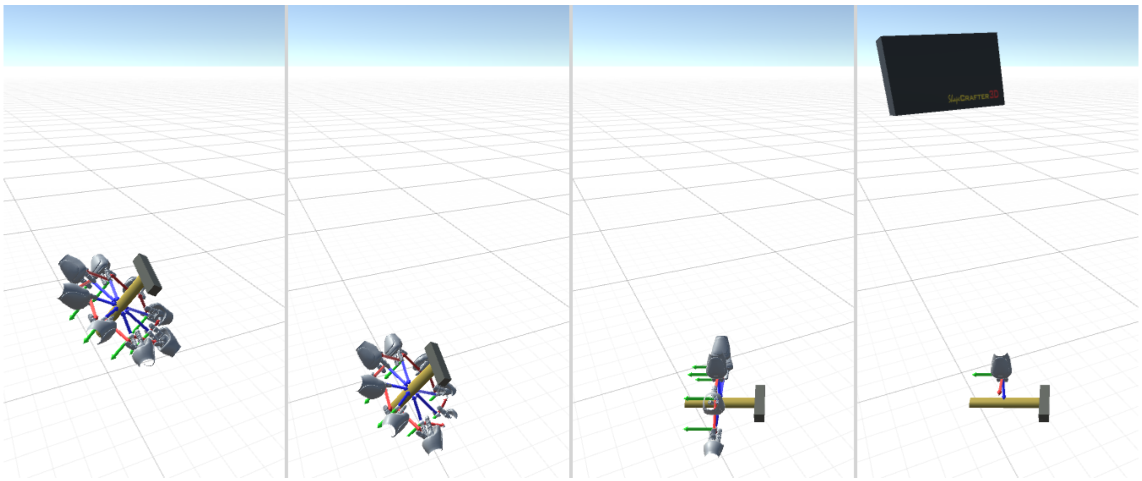


Figure 4.1: Instantiating and scanning a hammer in the virtual environment. First the hammer is instantiated in mid air. Thereafter it falls to the ground and comes to rest. The most suited grip vector for the current pose is chosen and the hammer is scanned with the virtual depth camera.

The virtual environment can also be used to create data sets from real data. A point cloud from the Shapecrafter depth camera is loaded, and the user can navigate the environment with the HTC Vive virtual reality head set. It is possible to place grip vectors on objects with the HTC Vive hand controllers and save the resulting vectors to file. This system makes it intuitive to train neural networks, and the user simply feels that he is showing a robot how to grip an object. However, because of long loading times and some other difficulties, it is not yet practical to create data sets this way, and this will be improved in future work.

The synthetic data created with the virtual environment was used to train the entire neural network model described in chapter 3. In future work, real data will be used to test how well the features trained in VR transfer to the real world and to fine tune the weights in the network.

4.1 Virtual environment for creation of synthetic data

The goal of the virtual environment is to create synthetic data which is as similar to the real data from the Shapecrafter as possible. Three key elements that are especially important in order to make such a system work well are

1. Realistic objects, scenery and physics for simulation
2. Realistic outputs from the virtual camera comparable to the outputs of the real camera
3. Precise labelling of data

By training robots in virtual reality, they can be exposed to an enormous amount of different situations, including scenarios that are unlikely or rare in real life.

4.1.1 Objects, scenery and simulation

To test the capability of the CNN for grip vector estimation, three objects were chosen, a hammer, a knife and a strawberry. A 3D-model for each object was created, and for every one, a set of possible grip vectors were defined. The 3D models are shown in figure 4.2. The hammer and the knife should be gripped by the shaft and the berry by the stem. The objects and their grip vectors are shown in figure 4.3. The objects were chosen on the intuition that the orientation of a hammer would be easy to recognize, a knife quite easy, and a berry rather difficult. A set of objects with shapes similar to the objects of interest were also designed, these were to have the label ”Nothing” in the data set and are shown in figure 4.4.

Each object in the virtual environment has a *transform* component added to it. The transform stores the position, rotation and scale of the object in world coordinates. The shape of the object is defined by a 3D-model, or *mesh*, and this mesh is what is seen by the virtual camera in the scene.

For physics simulation, the built-in physics engine in Unity was used. Unity lets you add components such as *rigidbodys* and *colliders* to objects. The rigidbody component defines the physical properties of the object, like its mass, drag, angular drag and if it should be affected by gravity. The collider component defines how the object should behave when it collides with other colliders in the scene. The colliders

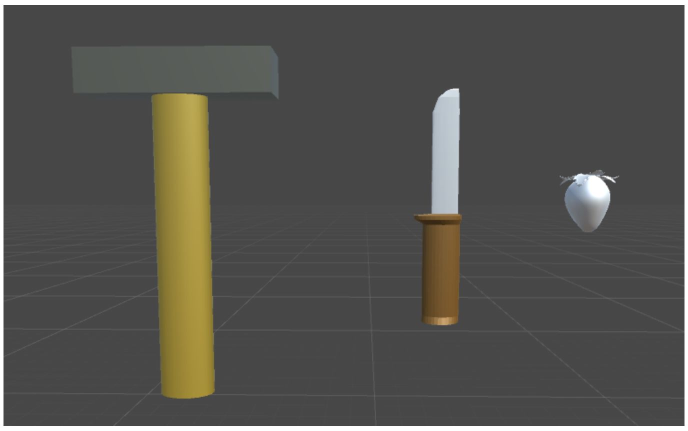
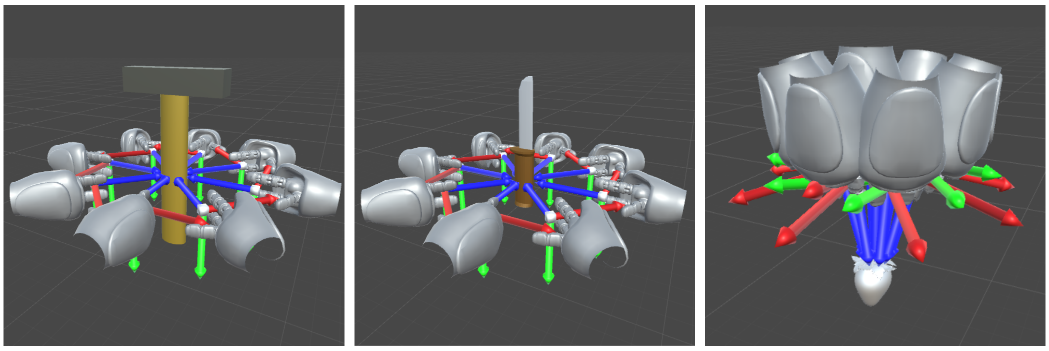


Figure 4.2: The 3D-models of the objects used in the virtual environment.



(a) Hammer with vectors (b) Knife with vectors (c) Berry with vectors

Figure 4.3: The pre-defined vectors for the designed objects. Each vector is illustrated with a 3D-model of the NAO robot's hand, and a coordinate system.

for the objects used in this project were made with a *low-poly*[1] version of the visible mesh in order to speed up the program. It is possible to add *collider material* to different objects as well, defining things like friction and bounciness, but realistic tuning of these parameters was not deemed necessary to produce realistic looking depth images of the given objects.

Each object has its own *material*, defining the appearance of the objects surface. Because only depth images were produced, no attention was paid to creating realistic looking materials with textures etc. and for the same reason lighting conditions were not considered.

Each instantiated object in the scene has a C#-script added to it with information about the object and a tunable *random scaling factor*. Upon instantiation the objects original size, as defined by the 3D-model, is adjusted according to this scaling factor. The object is stretched or squeezed in the x-, y-, z-dimension independently, deforming the mesh, thus creating a more realistic data set. The scaling factor was set the largest for the strawberry object to simulate biological variations.

The scenery works just like the objects described above, with the exception that it is not affected by gravity. Two scenes were used for data set generation, one of which consisted of a simple plane and nothing else. The other was littered with boxes in different rotations, designed to make prominent edges in the depth images.

[1]Meaning "low number of polygons", where polygon refers to a "face" in the 3D-model.

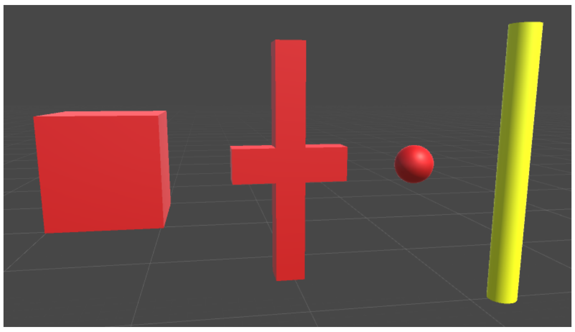


Figure 4.4: The 3D-models of the four different kinds of "Nothing" the network was trained on.

The first scene was used to create *the easy data set*, and the second to create *the hard data set*. Some objects dropped in both scenes are showed in figure 4.5.

4.1.2 Virtual depth camera

The sensor that will be used in the real world system is the depth camera Shapecrafter. In order to create a realistic synthetic data set, the virtual environment needs a virtual depth camera with similar outputs to the Shapecrafter.

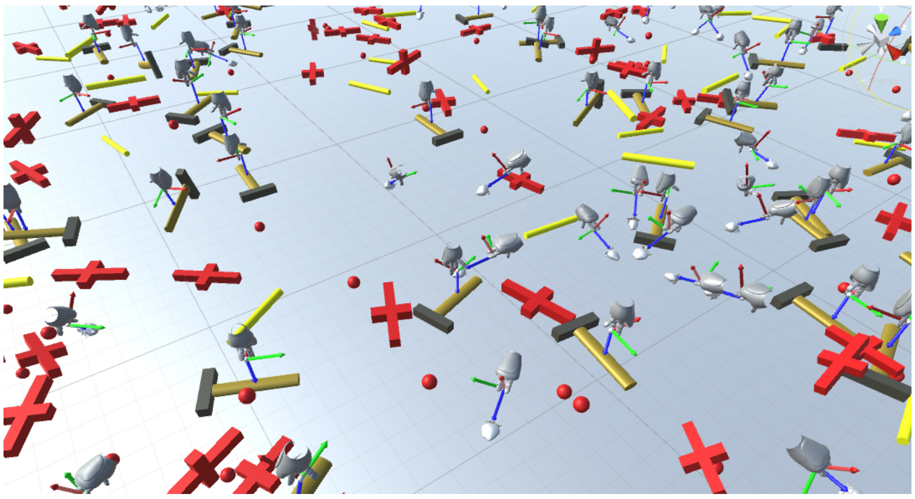
The Shapecrafter has a lateral resolution of 1920×1200 pixels and a depth resolution of $\frac{1}{10000}$ of the image width (see Shapecrafter factsheet in A.1). At a distance of 1000 mm from the camera we get an image width of 709 mm, height of 445 mm and a depth resolution of 0.07 mm. This gives $\frac{1920\text{pixels}}{709\text{mm}} \approx 2.7\frac{\text{pixels}}{\text{mm}}$ in both horizontal and vertical direction. This resolution is not needed to differentiate between objects of the sizes discussed here. Visual inspection of data from the virtual Shapecrafter indicated that a resolution of about $1/2\frac{\text{pixel}}{\text{mm}}$ would be sufficient. This resolution was chosen and used for all data generation in this project. This means that the real Shapecrafter can be used with a field of view of 3840 mm × 2400 mm, with a depth resolution of $\frac{3840}{10000} \approx 0.38$ mm. This might not be feasible due to light conditions etc., and if this is the case, the Shapecrafter image would simply have to be re-sized before it is fed to the network.

The virtual depth camera was created using one of Unity's virtual cameras. In Unity, virtual cameras define the way in which the player sees the world. The camera has a field of view (FOV) and a near and far *clipping plane*, and together they draw out a box in the scene as shown in figure 4.6. Everything inside this box will be rendered to the screen when the game is running.

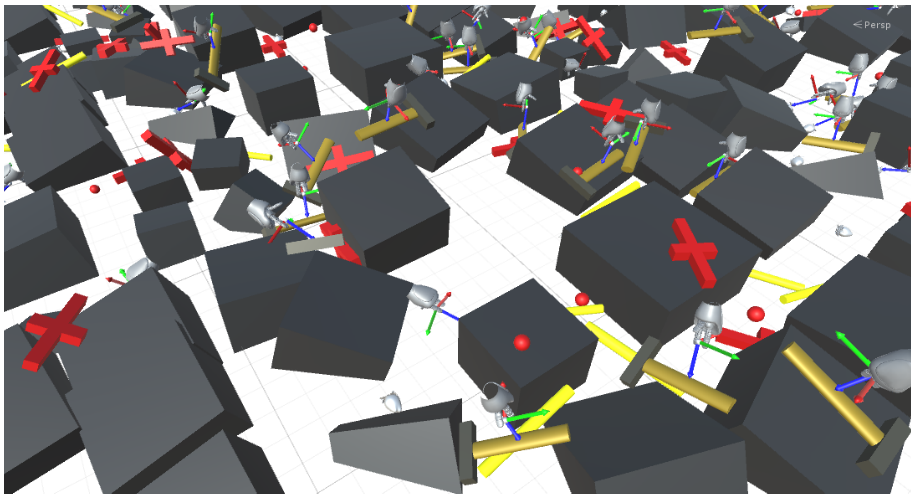
The virtual cameras in unity have a *depth texture*, which holds information about the distance to different objects in the scene. This texture is typically used when creating effects like fog which limits the visibility of objects as a function of how far away from the camera they are. This information can not be accessed directly from regular C# scripts, only through *shaders*.

How objects in the environment are rendered to the screen is defined by shaders. Shaders contain the algorithms for calculating the color of the rendered pixels as they appear on the screen. As an example, the shader for an object with a reflective surface would have to do some math based on the normal vectors of the mesh and the angle of the incoming light.

In order to access the depth texture needed for the virtual camera, a custom



(a) Some objects lying on the simple plane used for the easy data set.



(b) Some objects lying in the scene used for the hard data set.

Figure 4.5: The two scenes with all the designed objects instantiated randomly. This is not exactly what it looked like when the data sets were created. During data creation only one class was instantiated at the time, the reason for this is explained in 4.1.4.

shader was written and added to the camera object. The shader script, written in the *C for Graphics* language is appended in B.1.6. This shader renders the depth texture to a *render texture* which in turn is accessible from the depth camera's C# script, which handles writing of training examples to file.

The virtual scanning process, and the result of a scan is shown in figure 4.7. The grip vector for this particular example is also visible in the FOV of the camera. The grip vector does not appear in the rendered depth image because it is added to another *rendering layer* which is made invisible to the virtual camera.

For the virtual Shapecrafter, orthographic projection was chosen. The output data can therefore be viewed as ideal xyz-coordinates with no distortion. For the real depth images coming from the Shapecrafter, this is not the case. The real images are subject to lense distortion and perspective effects, as well as noise and "3D-shadows". The main issue with the raw depth images is probably the perspective effects, namely that the apparent size of the objects vary with the distance to the

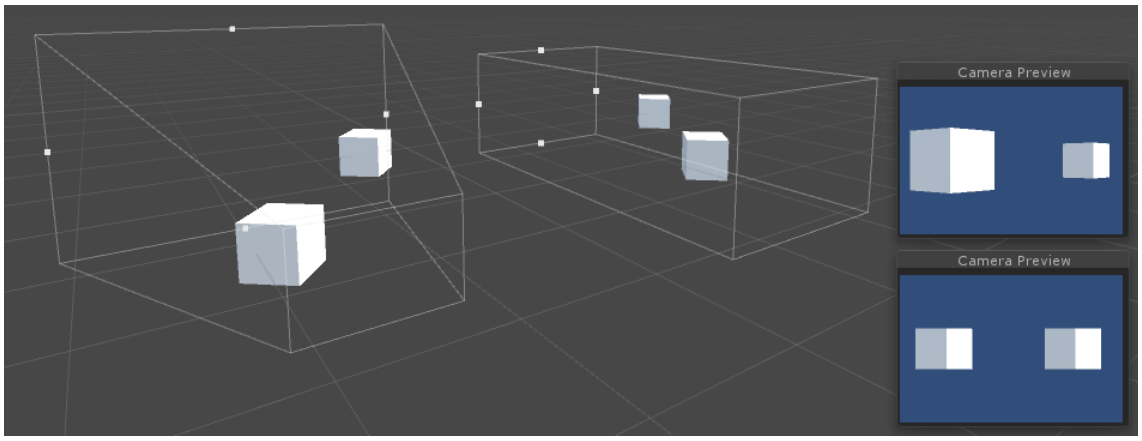


Figure 4.6: A perspective camera (left) and an orthographic camera (right) in the Unity editor. To the right, the rendered results of the perspective (top) and orthographic (bottom) projection.

camera. The neural networks in this project are trained directly on the orthographic images from the virtual camera, and have therefore only been exposed to the objects with one apparent size. In practice this means that if classification of objects were to be done directly on depth images from the Shapecrafter it would only will work at a given distance from the camera. As a preemptive measure every object is scaled randomly at instantiation when generating data, as discussed in chapter 4.1.1. Combined with the invariance to distortions introduced by the pooling layers in the CNN as explained in 2.1.3, it is deemed likely that the network would work within a reasonable range from the camera, even on the raw images. Additionally, the effects of perspective becomes smaller with greater distance from the camera, and as discussed earlier, the resolution of the Shapecrafter allows for quite large distances between it and the objects.

Orthographic projection has the desirable effect of eliminating perspective effects, and by using this projection method, the neural networks are given information about sizes and distances that are inherent to the scene and objects directly. As mentioned in chapter 3.1, there is a possibility of getting the data from the Shapecrafter as a set of calibrated xyz-coordinates with most of the undesirable effects eliminated. At runtime, the point cloud can be re-projected orthographically before it is fed to the network. Alternatively, the synthetically generated data could be re-projected with perspective effects and the network retrained on the new data, but this is probably the lesser option.

4.1.3 Heuristics for choosing the best grip vector

All three objects that were designed to be recognized by the system were defined with multiple grip vectors as shown in figure 4.3. When an object is scanned, one of the vectors is chosen and added to the data set as the correct grip vector for the object instance. For a vector to be chosen, it has to be deemed valid. For a vector to be valid it should be reachable by a robot, that is, no objects should be in the way of the vector and it should not point down into the ground etc. Each vector is defined as an empty object with cylindrical colliders, see figure 4.8. These colliders act as triggers, and do not stop objects from passing through each other. The vectors do

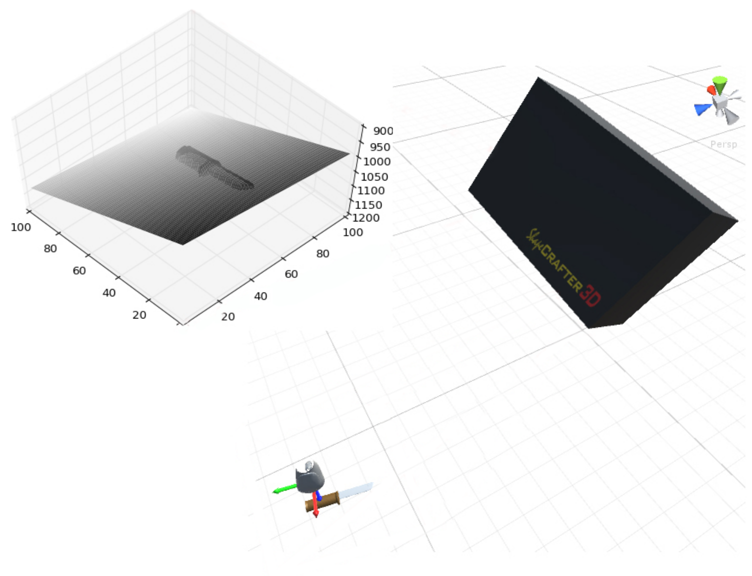


Figure 4.7: Scanning a knife lying on the ground in Unity with the virtual Shapecrafter. In the upper left corner, the raw data from the scan is plotted as a surface in python.

also have a mesh of a coordinate system and the hand of the NAO robot attached to them for debugging, but they are without physics components and do not get rendered by the depth camera.

When a vector is chosen, all the colliders are checked to see if they are colliding with the floor or other objects and therefore are not reachable. The vector objects that are colliding are then ”destroyed” and eliminated from the selection process. Out of the remaining vectors, the one that is closest to the surface normal in the scene is chosen (the one that is pointing closest to straight up). If all grip vectors for an object instance are destroyed, the example is saved with a grip vector reading ”NaN”.

4.1.4 Precise labelling of data

An advantage of the VR environment is the certainty and precision that can be ensured for the labels and vectors in the data sets. The positions and orientations of all objects in the scene are known through the transform component. Before the data collection is started, the best grip vector for each object is found, and the simulated physics is turned off.

A script called the *data collector* is handling the creation of the synthetic data set. The data collector contains a list of all the objects instantiated in the scene and has access to the transforms of all objects in the list. It also has access to the transform of the virtual depth camera. The data collector goes through its list of objects from top to bottom, moving the camera to a position above each object, taking an image, and saving the image and its corresponding grip point and grip vector to file. The data collection process is showed in figure 4.9. When moving the camera, it is first moved to the position of the object it is supposed to scan (as defined by its transform), and offset by one meter in world y direction (up). Thereafter it is rotated around the position of the object it is aiming at, by

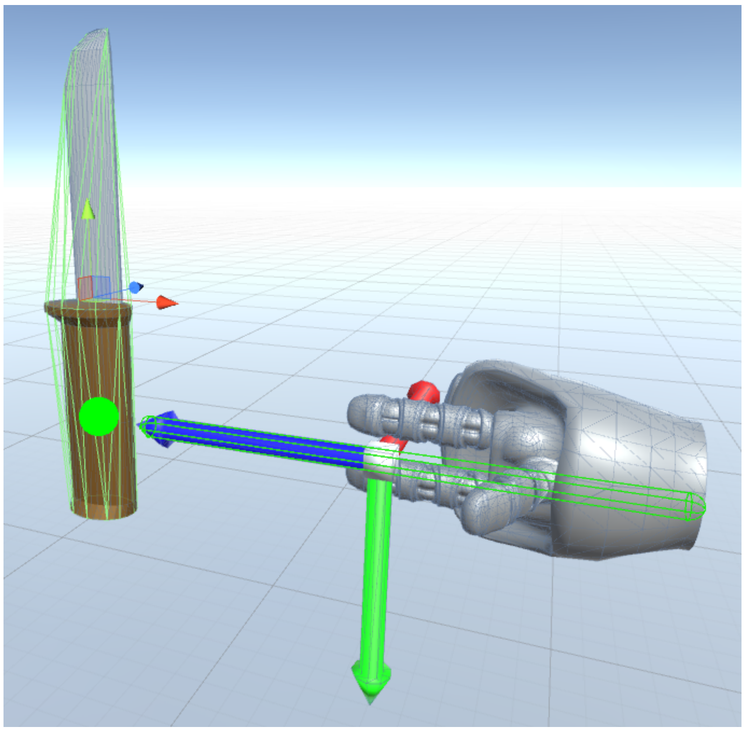


Figure 4.8: The knife object and a grip vector child object. The collider for the knife is shown as a green grid around the knife, and the collider for the grip vector is shown as a green cylinder over the axis model and hand. The grip point is defined at the position of the green ball, six cm below the knife's transform indicated by the thin axes near the middle of the knife.

some random amount (limits set by user). The random rotation makes sure that the examples and scenery are viewed from different angles. Finally, the camera is moved by some random amount in all directions to get the object out of the center of the image, and to add randomness in the distance to the camera.

When the camera is in position above the object, an image is rendered and saved, and the grip point and grip vector are obtained from the object of interest's grip point and grip vector *child objects*. These 6 *degrees of freedom* (DOF) are transformed from world to camera coordinates. This is done by getting the world to local matrix from the camera's transform and the local to world matrix from the grip vector's transform.

As described, the virtual camera's position above an object when an image is taken is defined relative to the scanned object's transform component. Because the grip point for each object is at a different distance from the respective transform, this leads to class dependent differences in the statistics of the target vectors. E.g. The grip point for the hammer is defined at the same position as the hammer's transform, but the grip point for the knife is defined at the shaft, about six centimeters from the knife's transform, see figure 4.8. For all examples in the generated data sets, the random movement of the camera was set with a limit such that an object's transform never was more than ± 5 cm from the image center for the x and y-axes, and at a distance of 100 ± 5 cm in the z-direction. Because the grip point for the knife is offset by 6 cm from the transform, it can be as much as 11 cm from the middle of the frame, whilst the hammer's grip point can only be 5 cm from the middle.

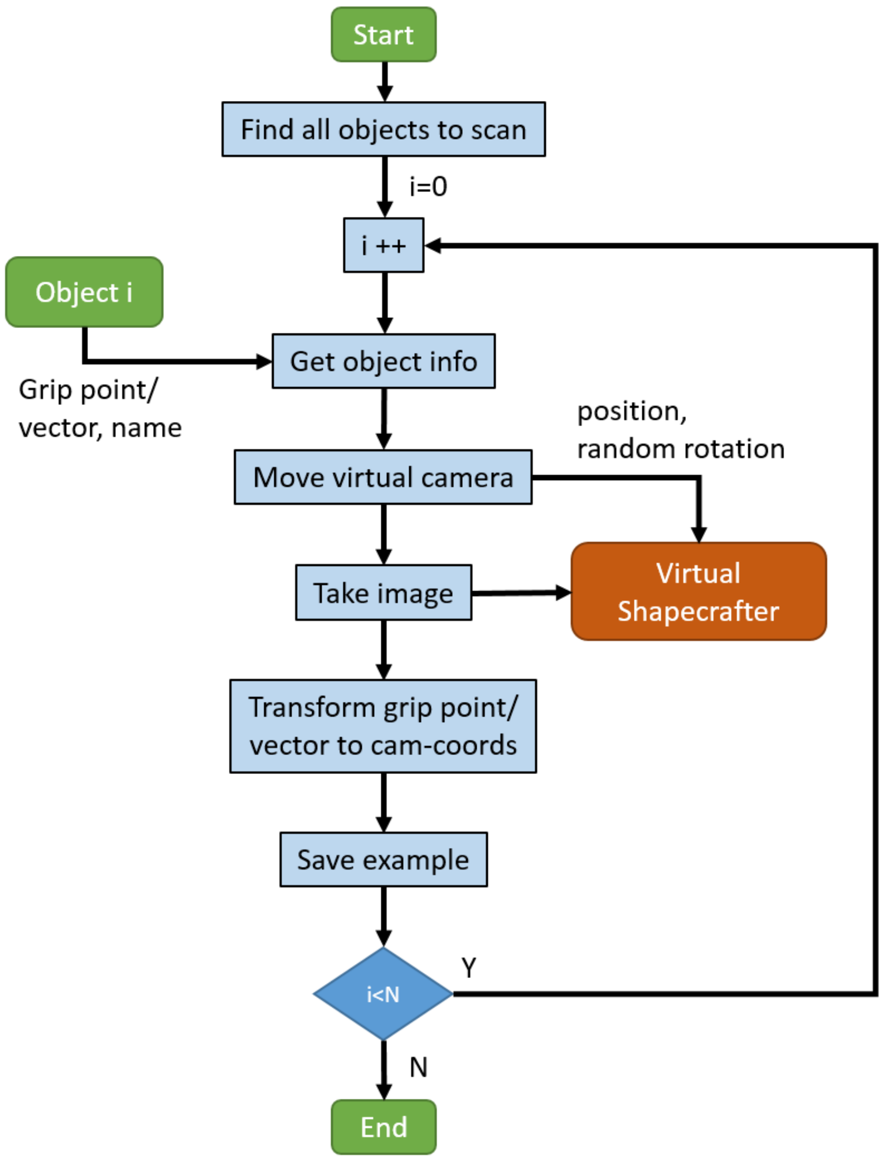


Figure 4.9: Float diagram for the data collector object

Segmentation truth maps

An interesting topic in computer vision is called *instance segmentation*. This is related to object detection, but instead of just detecting objects, each instance of a class is segmented. Even though segmentation is not done in this project, the truth maps for training of segmentation networks is output as an image file when data is generated. This was done with future work in mind and because it might be of general interest to the computer vision community as manual segmentation of real images is very tedious work.

An interesting thing to note about the synthetic ground truth images is that they don't have to be affected by occlusion (see figure 4.10). It might therefore be possible to use them to train networks that can "imagine" parts of an object that is occluded.

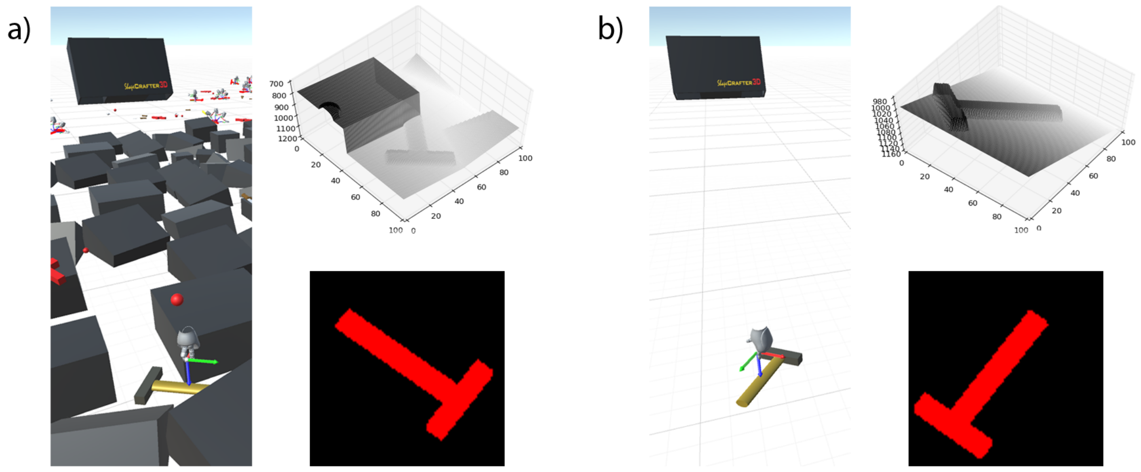


Figure 4.10: A scanned object and the generated depth map and segmentation truth map for an occluded object (a) and a clearly visible object (b). Although the hammer is occluded in example (a), the truth map shows the hammer in its entirety.

4.1.5 Summary of pipeline for synthetic data generation

The written source code for the virtual environment can be viewed in B.1. In summary, the synthetic data generation can be described like this:

1. Instantiate a number of objects of each class with random position and rotation
2. Scale the objects randomly
3. Let the objects fall and settle
4. Turn off physics
5. Choose a vector for each object in their current pose
6. Locate all objects with "object to scan"-script and add to list in data collector
7. For all objects in the list:
   - Move the camera into position above the object, pointing directly at it
   - Rotate the camera randomly around the objects transform
   - Move the camera randomly in all directions by a small amount
   - Render depth image
   - Render segmentation ground truth image
   - Save images to file
   - Append the information string with the label, grip point and grip vector for the object just scanned
8. Save information string to file

4.2 VR as an interface between man and machine

The developed VR environment can be used to label real world data. A point cloud from the Shapecrafter is imported in Unity and presented to the user. Using the HTC Vive head mounted display, the user can walk up to the scanned scene as he would in the real world and look at it from any angle. With the hand controller visible to the user as a semi transparent "ghost hand", the user can place grip vectors naturally by reaching into the point cloud and pulling the trigger on the controller, see figure 4.11.

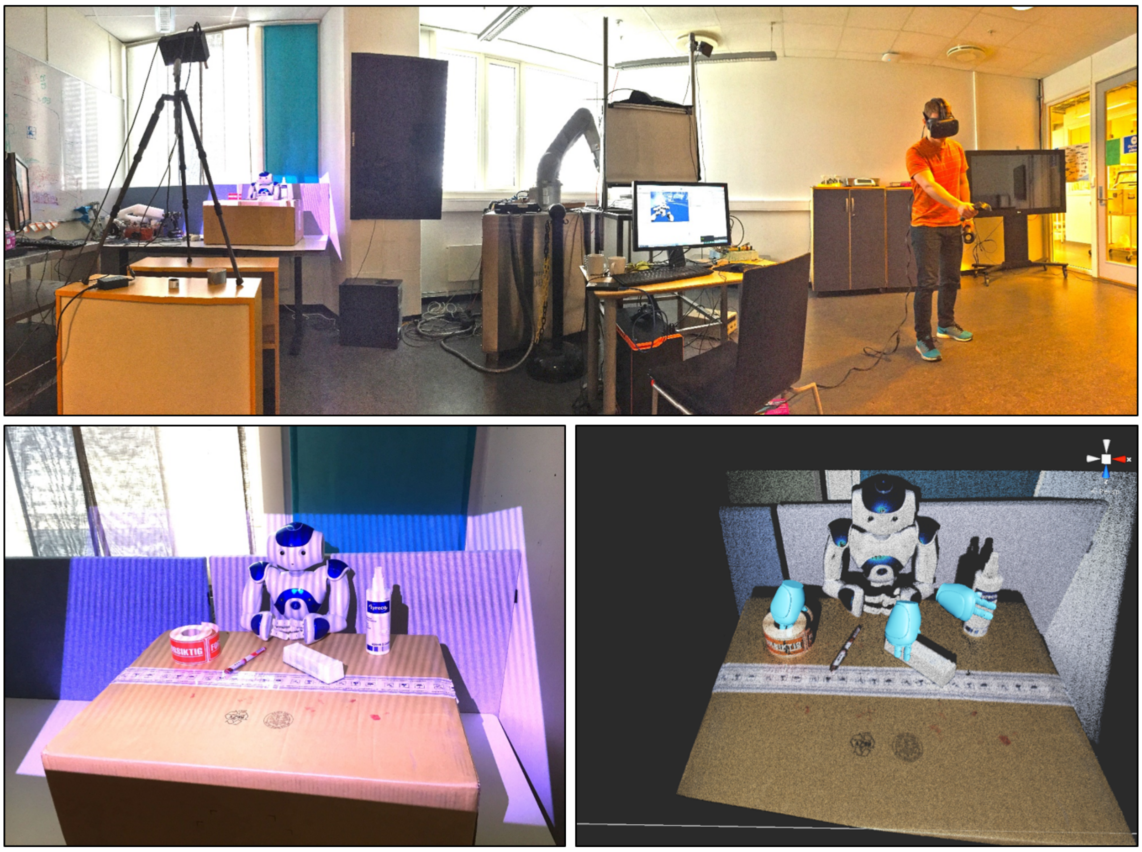


Figure 4.11: Top: To the left the Shapecrafter is scanning the scene and creating a point cloud. To the right the user is placing grip vectors in VR using the HTC Vive. Bottom left: The scene scanned with the Shapecrafter. Bottom right: The point cloud as it appears in VR, with grip vectors (blue) placed with the hand controllers.

Working with point clouds on a traditional screen is difficult because the cloud needs to be projected in two dimensions before it is presented to the user. This makes it difficult to judge distances within the cloud, and placement of a precise grip vector often requires excessive rotation of the cloud with respect to the projection plane. Additionally, the control of a grip vector in three spatial dimensions with three rotations is quite difficult.

Two attempts at creating a system for easy data labelling with a 2D screen were made. One using a 3D mouse and the other using an Xbox 360 controller. In both cases, placement of a single vector took about 20 seconds, and significantly more if the user was unfamiliar with the controls. Such a long lag time between deciding where to place the grasp, and successful execution of the task is perceived as very frustrating. Typically the user feels discouraged already after a few examples and the effort and focus needed to place each vector makes labelling of a few hundred examples this way seem like a daunting task.

A real data set with some hundred labelled examples is needed if the neural networks are to be fine-tuned before they are tested on real data. The test set should also consist of a few hundred examples if the networks (especially the grip vector estimator for all poses) shall be tested satisfactory.

One of the goals of this project is to create a system that can be expanded in a way such that human workers can show robots what to do without involving a programmer. Loading point clouds in Unity provides an intuitive way of labelling data and a human worker with little or no technological background can show the robot how to grip an object simply by doing it himself with his virtual hand. "Under the hood", the user is really constructing a data set for the neural networks to train on[2]. This system is easily expandable and could also be used to create data sets of paths with way-points in camera coordinates instead of only end effector positions. This would be unthinkable with the 3D-mouse or Xbox 360 controller.

Currently, the system is only working as a proof of concept. Point clouds can be loaded in Unity using a commercially available plug-in [31] and the functionality for the head mounted display is provided by the Valve Corporation [32]. A colored point cloud from the Shapecrafter is loaded as an .xyzrgb file and converted to a binary file before it is viewed in Unity. Every point cloud needs to be loaded from disk and converted separately and this takes about 15 seconds pr. example. In cooperation with the developers of the Shapecrafter some effort has been made in creating a live feed from the camera directly into Unity and this will be expanded upon in future work.

[2]The training could be done *online*, that is, the weights could be updated with one step of SGD for each new example or they could be gathered into a data set before training is begun.

Chapter 5

Experiments and results

The neural networks trained in this project were implemented in python using the Theano and Lasagne libraries. All neural nets were trained solely on synthetic data and the training was done with stochastic batch gradient descent with Nesterov momentum on an NVIDIA Titan X GPU. Testing of the different sub systems described in chapter 3 were done separately and the results for the systems are also presented separately. Several architectures were tested for all subsystems in order to compare the best architectures for classification, grip point and grip vector estimation. A full system test was also performed, with the neural network estimating grip vectors live in the VR environment.

5.1 Generating data

Data sets for training and testing of the networks were generated using the developed virtual environment. Two separate data sets were created: the *easy* and the *hard data set*. For the easy set, 2000 examples of a class (e.g. hammer) were instantiated and dropped on a plain surface with an area of 0.25 km^2. These instances were scanned with the virtual camera and the resulting depth images were saved to disk along with the position and orientation information for the objects' grip vectors as described in 4.1.4. This process was done separately for the classes, hammer, knife and strawberry and for the four look alike classes showed in figure 4.4. In addition 2000 depth images of the plain surface were created with randomness in the distance to and rotation of the virtual camera The result is a training set of 16 000 examples. By instantiating relatively few objects over such a large surface the likelihood of any two objects landing in close vicinity of each other is minimized. Therefore, almost all the rendered depth images captured a single object lying naturally on a plain background. By creating the data for each class separately it is ensured that if an image does contain two objects, they are of the same class. Thus, any ambiguities as to what label a training example should have are avoided. The cross validation set was created in the same way and during training, 5000 examples were randomly chosen and used.

The hard data set was created in the same manner as the easy set, but the objects were dropped on the more difficult background surface, as shown in figure 4.5b. The dropped objects were the same, with the exception that the box-look-alike-object was not used, due to its similarity to the background, and also no images were rendered of the background alone. 5000 objects were instantiated of each class, creating a training set of 30 000 examples. The prior probability of the "Nothing" class is therefore 50 % for the hard data set and 62.5 % for the easy data set. During all training and testing, including the live testing, random noise was added to the images, amounting to an uncertainty in the depth direction of 1 mm, which is much higher than the uncertainty for the real sensor.

Two different pre-processing steps were tested for the classification network, whereof one consisted of making the data zero mean and unit variance. The mean was calculated for each image separately and subtracted. The standard deviation was calculated for the training set only, giving a single scaling factor. Every image in both the training set and the cross validation set was divided by the standard deviation for the training set. The classifier was also tested with band pass filtered input images. This was done in an attempt to eliminate the effect of the DC-value and tilt of the background plane. The filtering was implemented by introducing a bottom layer in the CNN with only one filter with fixed weights.

No dedicated pre-processing was done for the grip point and grip vector estimators, but the input is passed through a batch-norm layer before it reached the first convolution layer.

5.1.1 Comparison to real data

The generated synthetic images can be subjectively compared to real data by visualizing the real and synthetic images as surface plots. The raw output from the Shapecrafter and a similar looking synthetic example is shown in figure 5.1. The shown Shapecrafter image is a naive orthographic projection of the point cloud, where each point is simply rounded to the nearest mm in the spatial dimensions (x, y) and used to index a matrix where the lowest z-value for each x-, y-position is set as the pixel value.

As discussed in chapter 4.1.2, the real images have a much higher resolution than the synthetic ones and the real images will also have "3D-shadows", meaning that areas of the image are without value, i.e. the pixels are defined as *NaNs*. The naive projection of the image leads to somewhat larger 3D-shadows than necessary, and ideally the cloud should be rotated before it is projected.

A simple way of filling the holes in the image is to replace the NaN-values with the mean value of the image[1]. This was done in MATLAB, before the image was resized to resemble the synthetic data with MATLAB's *imresize* function. After filling the NaNs, no other form for pre-processing was done on the image, other than what is done by the imresize function's bulit-in anti-aliasing functionality. The result is shown in figure 5.2. The resulting image looks quite similar to the synthetic one, keeping in mind that the real hammer is longer and has a shaft and head shape that is different from the virtual hammer. Some artifacts are visible in the real image where the NaNs were before, and these are nonexistent in the synthetic image. Low pass filtering and other pre-processing steps might be needed and in that case, the pre-processing should also be done on the synthetic images during training.

In the real world, different objects have different *specular properties*, that is, they reflect light differently. For scenes consisting of objects with both high and low reflectance, this leads to more NaNs in the image than is the case for the image

[1]More sophisticated ways of filling the holes are possible, such as using a local mean value rather then the global mean.

shown here. The reflections are most often not a result of the ambient lighting, but rather the emitted light from the Shapecrafter itself. With the Shapecrafter, multiple images can be taken with different aperture sizes to counteract this, but this can in turn be problematic if there is much movement in the scene.

From visual inspection of the images it is believed that the synthetic data does have enough resemblance to the real world data to ensure at least some transferability of the neural network parameters trained in VR to the real world. The lack of labelled real world examples has prevented the testing of this hypothesis and will be the focus of future work.

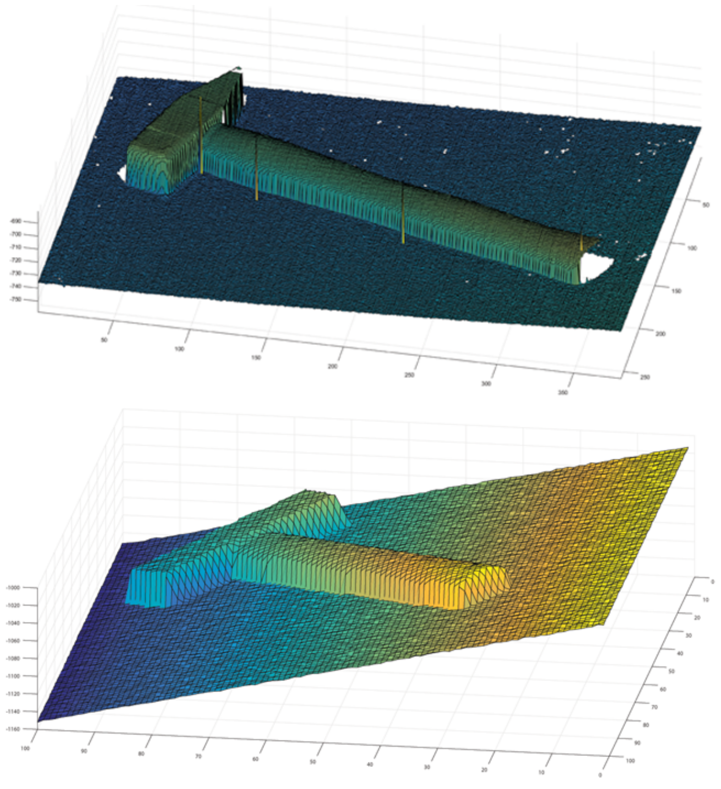


Figure 5.1: Top: A real depth image from the Shapecrafter. Bottom: A synthetic image from the virtual environment with added stochastic noise.

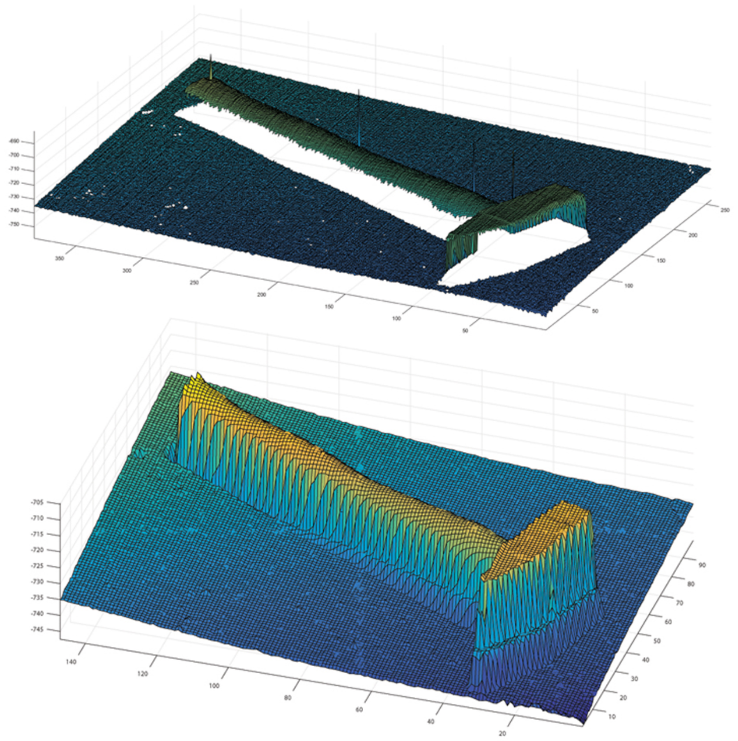


Figure 5.2: The real depth image before (top) and after (bottom) processing. This is the same image as shown in 5.1, viewed from another angle.

5.2 Architectures

Many different architectures were tested with different numbers of convolution and pooling layers for all sub systems. In order to narrow down the search for good architectures, the filter dimension was fixed to 3×3, pooling to 2×2 and drop out was used with $\rho = 0.5$ for the dense layers in all experiments.

Of special interest were the pooling layers, which give invariance to translation and this is important for good classification. At the same time, precise spatial information is lost, which could be needed for precise grip point estimation.

It turns out that the three classes in the synthetic data set were very easily distinguishable and good results were found for a wide variety of CNNs. Some of the training curves for the classifier trained on the hard data set are shown in figure 5.3.

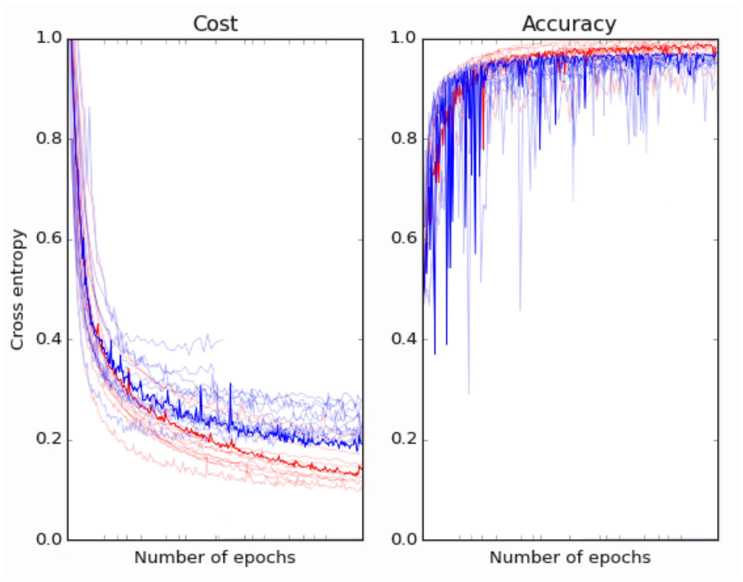


Figure 5.3: Left: Cost vs. epochs. Right: Accuracy vs. epochs. Red lines are for the training data, blue for the CV-data.

As figure 5.3 shows, there are differences in the costs for the different classifiers, but the accuracy is very high in all cases.

The classification architecture which got the best results when training on the easy data set with zero mean/unit variance inputs and band pass filtered inputs respectively, are shown in figure 5.4. As is shown, the best network with band pass filtered inputs has fewer parameters and is not as deep as the network with the zero mean/unit variance inputs.

The best architectures found for both types of pre-processing when training on the hard data set are shown in figure 5.5. In this case, both architectures have three pooling layers, indicating that more invariance in the spatial dimensions is needed when classifying the hard data set than what is needed for the easy data set. The networks for both types of pre-processing have equal depth, but the number of parameters is lower in the network with band pass filtered inputs, than in the network with zero mean/unit variance inputs.

The best architecture for the grip point estimator is shown in figure 5.6. That the number of pooling layers is the same as for the classifier network is of particular

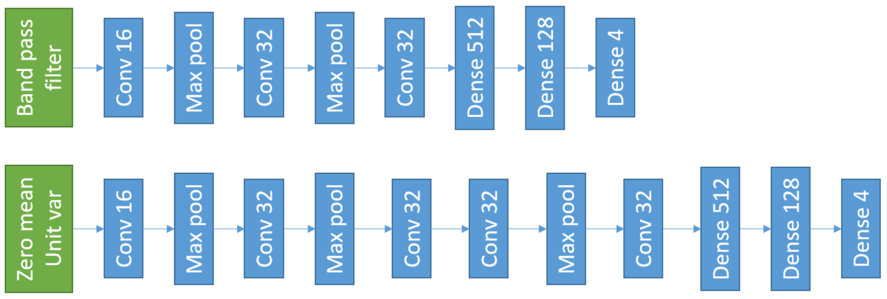


Figure 5.4: The network architectures for the easy data set with both types of pre-processing. Batch-norm layeres are not shown, they succeed each of the conv-layers.

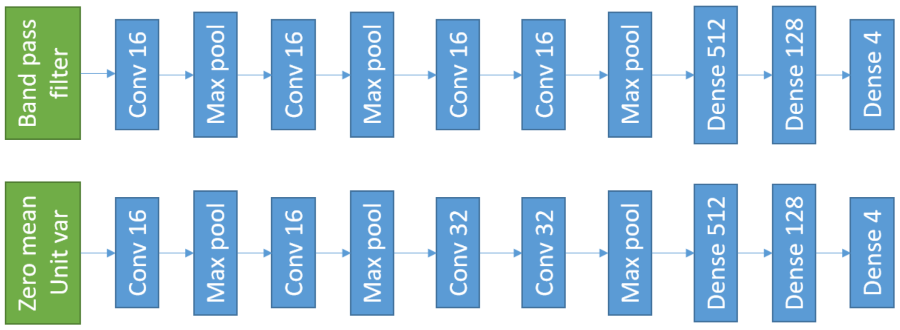


Figure 5.5: Architectures for the hard data set with both the band pass filtered, and the unit variance and zero mean inputs. Batch-norm layeres are not shown, they succeed each of the conv-layers.

interest. As will be presented in section 5.4.1, the grip point estimator has an average error of 4 mm in the spatial dimensions for the most difficult class. This indicates that enough spatial information is preserved in the higher layers of the classification network to predict grip points with errors of 4 mm or less in the spatial dimensions.

As described in chapter 4.1.2, the resolution set for the virtual camera is 2 mm/pixel. With three pooling layers, we get an uncertainty in spatial position of 8 pixels, which is equivalent to 16 mm in the virtual environment (or real world). Presumably, by looking at the whole object, the grip point estimator still manages to get errors of 4 mm or less in the x-, y-plane.

The best results for the grip vector estimator was found with the same architecture that was used for grip point estimation, shown in figure 5.6.

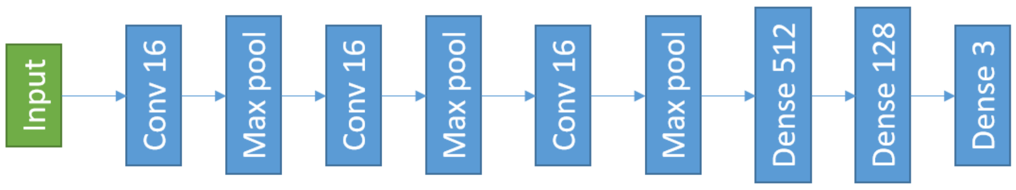


Figure 5.6: The best architecture for the grip point estimator. Batch-norm layers succeed each of the conv-layers and also the input layer.

5.3 Classifier

The classifier subsystem was tested on both the hard and easy data sets. Additionally, testing was done with both band pass filtered and zero mean and unit variance input data.

The results for the easy data set are very good overall, and labelling the data proved to be an easy task for many CNN architectures. The confusion matrix for the CV set with zero mean and unit variance inputs is shown in table 5.2 and the confusion matrix for the CV set with band pass filtered inputs is shown in table 5.1. Some of the mislabelled examples are shown in figure 5.7. Most of the erroneously labelled examples are special cases, e.g. the hammer standing on its head, and the berry lying upside down with its stem hidden. One notable example is the ordinary looking knife erroneously classified as a berry.

	Nothing	Hammer	Knife	Strawberry
Nothing	3124	1	5	0
Hammer	17	599	0	0
Knife	2	0	640	0
Strawberry	2	0	0	610

Table 5.1: Confusion matrix for classifier with band pass filtered inputs from the easy test set.

	Nothing	Hammer	Knife	Strawberry
Nothing	3124	3	3	0
Hammer	1	614	1	0
Knife	0	0	641	1
Strawberry	6	0	0	606

Table 5.2: Confusion matrix for classifier with zero mean and unit variance inputs from the easy test set.

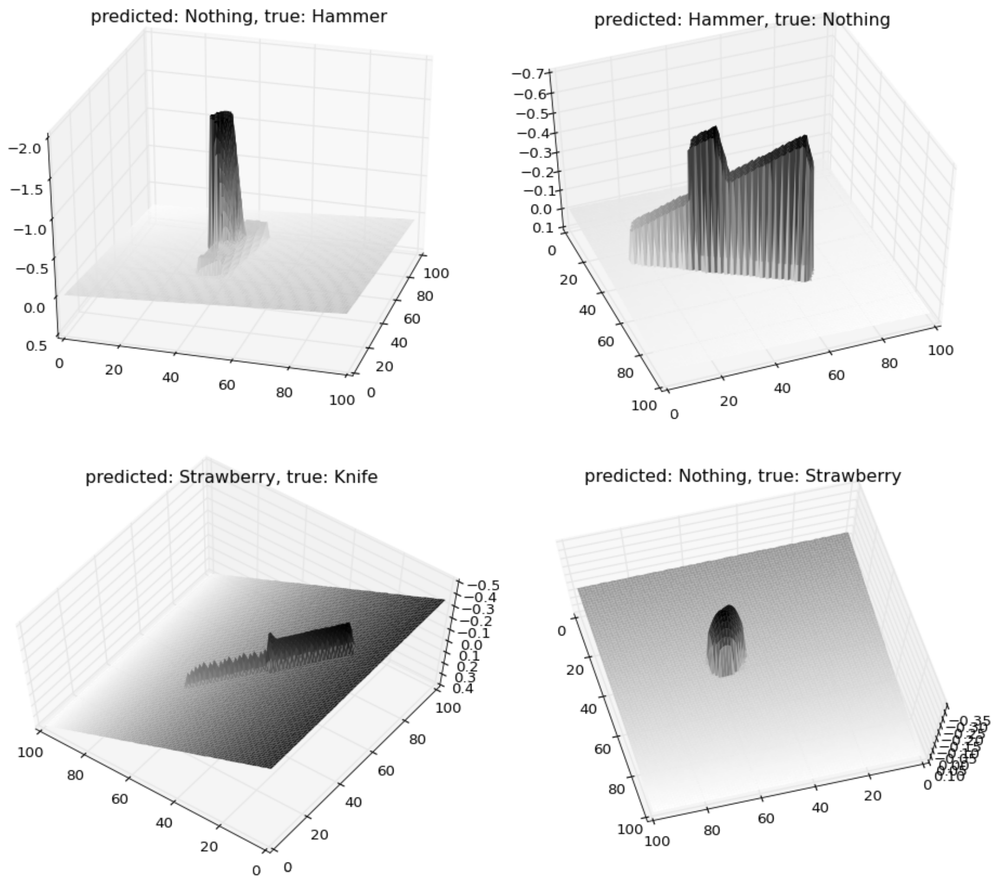


Figure 5.7: Mislabelled examples from the easy data set. Upper Left: Predicted Nothing, true label Hammer. Upper Right: Predicted Hammer, true label Nothing. Down left: Predicted Strawberry, true label Knife. Down right: Predicted Nothing, true label Strawberry

The best results on the hard data set were found with zero mean and unit variance inputs, as was also the case on the easy data set. The confusion matrices for the best classifiers with both types of pre-processing are shown in tables 5.3 and 5.4.

	Nothing	Hammer	Knife	Strawberry
Nothing	2978	10	6	6
Hammer	46	952	2	0
Knife	17	0	975	8
Strawberry	24	0	3	973

Table 5.3: Confusion matrix for classifier with zero mean and unit variance input

	Nothing	Hammer	Knife	Strawberry
Nothing	2961	6	21	12
Hammer	55	941	4	0
Knife	30	0	968	2
Strawberry	42	0	38	920

Table 5.4: Confusion matrix for classifier with band pass filtered inputs

As can be seen from the confusion matrices, the overall performance is good. For the network with bandpass filtered inputs, strawberries get mislabelled the most, with an error rate of 8.0 %. For the network with zero mean and unit variance inputs, the hammer class is most often mislabelled, with an error rate of 4.8 %. The cost and accuracy plots for both types of pre-processing is shown in figure 5.8 and some of the mislabelled examples are shown in figure 5.9.

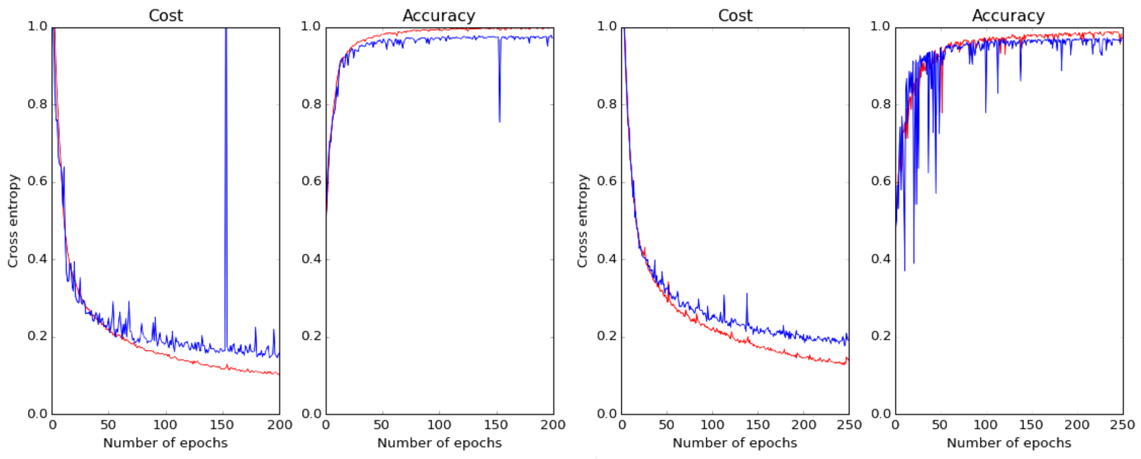


Figure 5.8: Cost and accuracy vs. epochs of training. Left: Zero mean and unit variance inputs. Right: Band pass filtered inputs.

Some of the mislabelled examples are difficult to classify, even for humans, when they are viewed directly as images and not as surface plots. As was also the case for the easy data set, the strawberry class is sometimes confused with the knife class. As shown in figure 5.9 d), some ordinary looking strawberries are labelled as a knife. Because of the way the berries roll on the surface they seem to be more prone to occlusion, than for instance the hammer class is, resulting in errors such as the one in 5.9 f). The same is true for the knife class. As we can see in figure 5.9 a) some quite understandable mistakes are also made. If a human labeller hadn't known ahead of time that there are no such looking hammers in the data set, it is possible that the labeller would have made the same mistake.

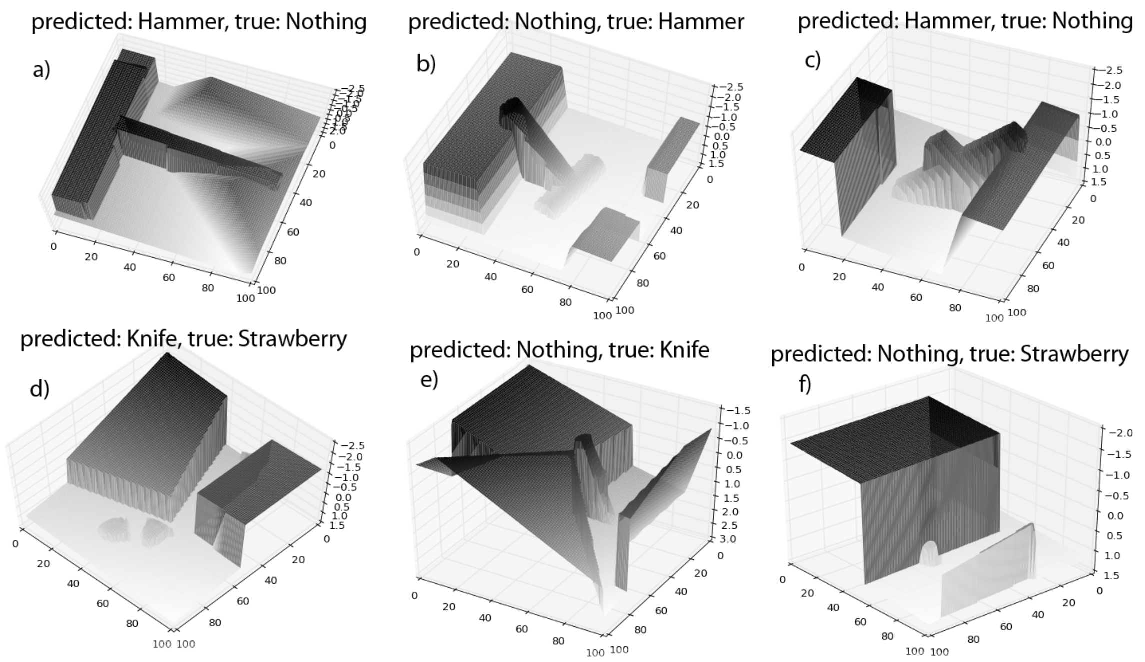


Figure 5.9: Mislabelled examples from the hard data set.

The classifier trained on the easy data set was also tested on the hard data set and vice versa. The results are very poor in both cases and confusion matrices are shown in tables 5.5 and 5.6. The classifier trained on the easy set predicts that close to 82 % of the examples in the hard set contains nothing. This is not surprising due to the fact that the background in the hard set looks quite like one of the nothing-objects the classifier was trained on. In addition, the prior probability of the nothing class was 62.5 % in the easy set, which the classifier was trained on, far higher than the other classes, making "nothing" the natural default choice.

	Nothing	Hammer	Knife	Strawberry
Nothing	2898	64	27	11
Hammer	715	282	3	0
Knife	639	94	238	29
Strawberry	663	17	7	313

Table 5.5: Confusion matrix for the classifier trained on the easy set, tested on the hard set.

	Nothing	Hammer	Knife	Strawberry
Nothing	1244	43	965	878
Hammer	100	196	271	49
Knife	6	0	607	29
Strawberry	0	0	33	579

Table 5.6: Confusion matrix for the classifier trained on the hard set, tested on the easy set.

The classifier trained on the hard data set classifies most of the knives and berries in the easy data set correctly. At the same time however, it classifies more than half of the hammers and nothing-objects as either knives or berries. Every example in the easy data set contains much fewer edges in total, than do the images the classifier was exposed to during training on the hard data set. On average, the examples in the hard data set with the least edges are probably the examples containing berries and knives. These objects often roll down slopes and become less prominent in the image. This is a possible explanation for the large bias toward knives and strawberries for the classifier trained on the hard set.

To test if the network could learn to distinguish the examples in both data sets simultaneously if it was exposed to both types of examples during training, the 30 000 examples from the hard data set were combined with the 16 000 from the easy data set to form one large training set consisting of 46 000 examples. The 16 000 CV-examples from the easy set were also combined with the 6 000 CV-examples from the hard set and 10 000 images was randomly chosen for cross validation during training. The architecture with the best result on the zero mean/unit variance inputs for the hard data set (bottom in figure 5.5) was chosen, and retrained using the new data set. The training progression is shown in figure 5.10 and the confusion matrix for classification of all 22 000 CV/test examples is shown in table 5.7.

As the results show, the re-trained network is not only capable of distinguishing the objects from both the easy and the hard data sets at the same time, it does a better job than before.

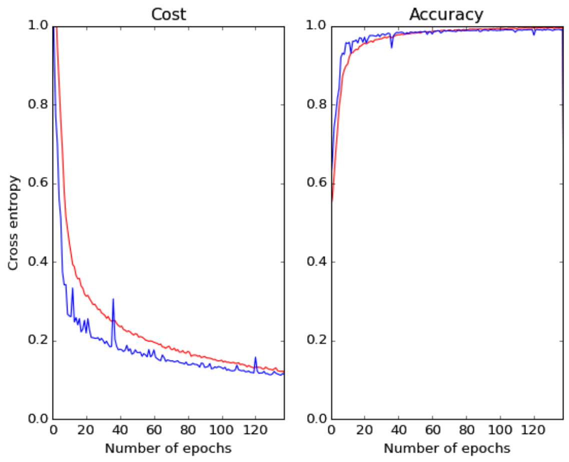


Figure 5.10: Cost and accuracy vs. epochs for training on the combined data set.

	Nothing	Hammer	Knife	Strawberry
Nothing	12956	11	28	5
Hammer	67	2924	9	0
Knife	12	0	2981	7
Strawberry	54	0	6	2940

Table 5.7: Confusion matrix for the classifier trained on both data sets.

The classifier was also tested as an object detector by splitting a larger rendered image of size 500×500 into a stack of 100×100 images. Without any further training, every image was classified with a network trained on the easy data set and the resulting probability vectors were reshaped into into four probability surfaces, each the size of the original image. The resulting decision boundaries for the different classes are shown in figure 5.11.

No thorough tests were done, but this single test demonstrates the plausibility of the system. The decision boundaries are noisy, and would probably need filtering in a real world system. However, keeping in mind that the network never saw different classes within the same receptive field during training, the results are promising. The decision boundaries are quite sharp and the object detector goes from close to 100 % certainty of one class to 100 % certainty of another in a few millimeters. This is probably a result of the systematic way, in which randomness in translation is introduced in the synthetically created data (as described in 4.1.4, the object is only allowed to be ± 5 cm off-center in the image). Smoother decision boundaries could be achieved with different training procedures, but low-pass filtering of the existing probability surfaces may suffice in a real world application.

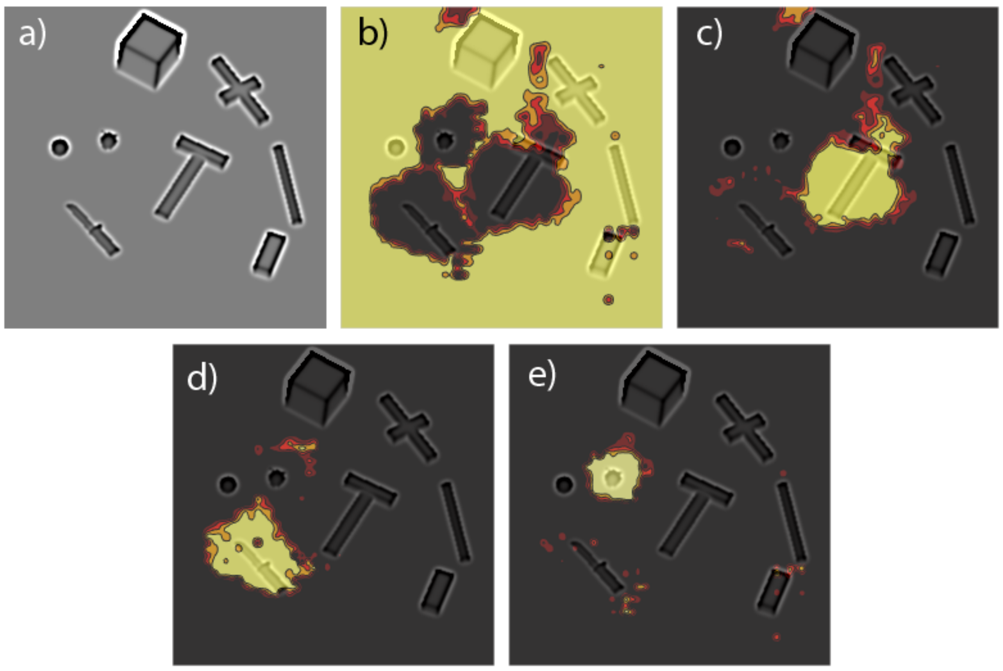


Figure 5.11: Probability surfaces overlaid the input image. Yellow indicates certainty of detected class. a) Original image, b) Nothing detector, c) Hammer detector d) Knife detector e) Berry detector. (The input image was filtered for better illustration).

5.4 Grip point and grip vector estimation

The grip point and grip vector estimators were trained as separate networks. The grip point estimator outputs the x, y, z-coordinates of the proposed grip and the vector estimator outputs six values, defining the rotation of the grip. Both networks have a linear activation function in the top layer and the grip point estimator was trained with the L2-loss function and the vector estimator with the dot-loss proposed in chapter 3.2.2.

As described in chapter 4.1.3, not all objects have valid grip vectors (E.g. berries lying on their heads or with their stems close to some other objects). In both the easy, hard and combined data sets, these examples were ignored. That is, the networks were only trained with images and corresponding vectors, there is no "do not grip" option. The number of examples of each class in the easy and hard data sets after removal of invalid examples are shown in table 5.8.

Object	Training hard	Training easy	CV hard	CV easy
Hammer	4980	5000	998	5000
Knife	4951	5000	988	5000
Berry	3515	4688	689	4737

Table 5.8: Number of examples in the different data sets.

5.4.1 Grip point estimation

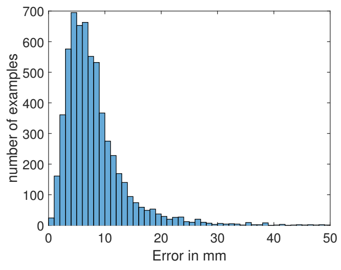
The grip point estimator was trained on the combined training set consisting of examples from both the hard and easy data sets. A value of 1000 mm was subtracted from the z-value in the target vectors to account for the distance to the camera. The average errors after training are shown in table 5.9 and the histograms of the errors are shown in figure 5.12.

Object	Avg. error tot.	Avg. error x	Avg. error y	Avg. error z
Hammer	8.1 mm	3.9 mm	4.0 mm	4.3 mm
Knife	7.7 mm	3.6 mm	3.5 mm	4.2 mm
Berry	4.9 mm	2.2 mm	2.3 mm	2.8 mm

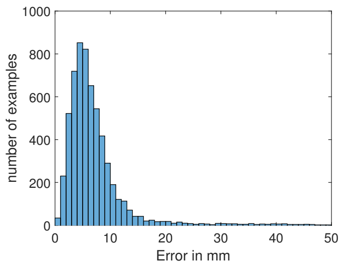
Table 5.9: Average errors for the grip point estimator on the combined CV-set.

As is shown, detecting the grip point for the hammer seems to be most difficult. A possible reason for this might be that the shaft of the hammer is very homogeneous looking. The grip point for the hammer is somewhat arbitrarily defined as the middle of the hammers transform component (close to the middle of the hammers shaft). Additionally, the hammer tends to land in more challenging positions than the knife and especially the berry object.

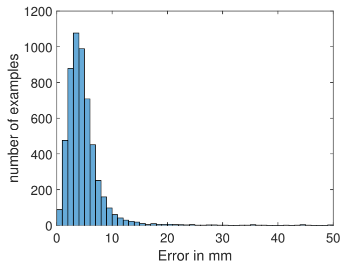
As described in chapter 4.1.4, the variance in the target vectors is largest for the knife, and smallest for the hammer. This does not seem to affect the grip point estimator much.



(a) Hammer



(b) Knife



(c) Berry

Figure 5.12: Histogram of errors for the estimated grip points on the combined CV set. Error in millimeters on the x-axis, and number of examples on the y-axis.

5.4.2 Grip vector regression

The sub system for finding grip vectors was trained on two target vectors dubbed the *palm vector* and the *point vector* as shown in figure 5.13. The vectors define the rotation of an end effector, are of unit length and are given in camera coordinates.

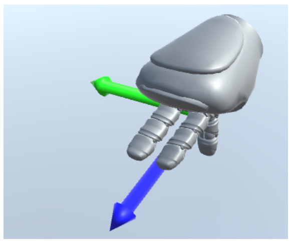


Figure 5.13: The palm (green) and point (blue) vectors that define the rotation of the end effector.

The target vectors for the different classes in the easy CV-set are visualized in figure 5.14. Each point can be viewed as the tip of either the palm or point vector pointing away from the origin. The top row shows the targets for the point vectors and the bottom row the targets for the palm vector. Most of the hammers and almost all of the knives in the easy data set are lying flat on the surface. The target vectors for the pointer are therefore all pointing in the same direction as seen in 5.14 a) and b) (down in world coordinates, up in camera coordinates). Similarly, the palm vectors draw out a ring in space as seen in 5.14 d) and e) because of the rotation of the objects on the plane. In 5.14 d) some palm targets are also pointing down for cases where the hammer landed standing on its head. The spread in the clouds in figure 5.14 a) and b) and the width of the rings in d) and e) are mostly due to the random rotation of the virtual camera.

The predicted vectors corresponding to the targets in figure 5.14 are shown in figure 5.15. The same plots for training done on the hard and combined data sets are shown in figure 5.16, 5.17, 5.18 and 5.19. As is clearly visible from the predicted palm vectors (the d) e) and f) plots in figures 5.15, 5.17 and 5.19) the network is not able to capture rotations out of the mean image projection plane well. The visualized rings are much narrower than the corresponding rings in the plot with the target vectors. One notable exception is the predicted palm vectors for the hammer in the hard and combined data sets in figures 5.17 and 5.19 d). As is seen from the corresponding target vector manifolds (especially the one for the hard data set in figure 5.16 d)), there are quite a few training examples on a larger part of the sphere, which undoubtedly helps training a lot.

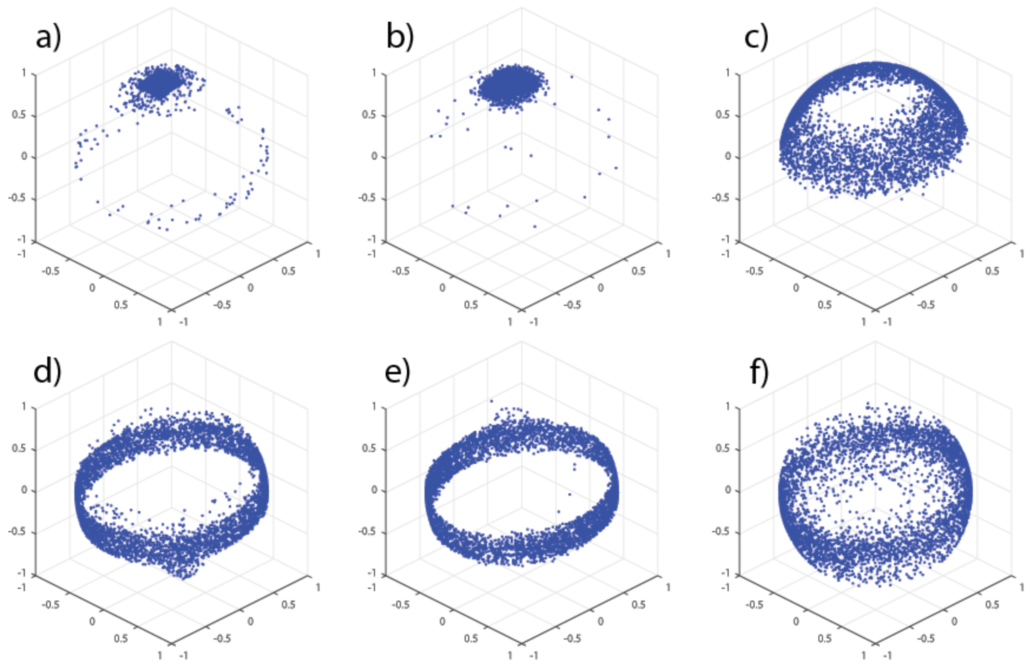


Figure 5.14: The target vectors for the different classes visualized with a scatter plot. Each point represents one training example in the easy set. a) Hammer point vector targets, b) Knife point vector targets, c) Berry point vector targets, d) Hammer palm vector targets, e) Knife palm vector targets, f) Berry palm vector targets.

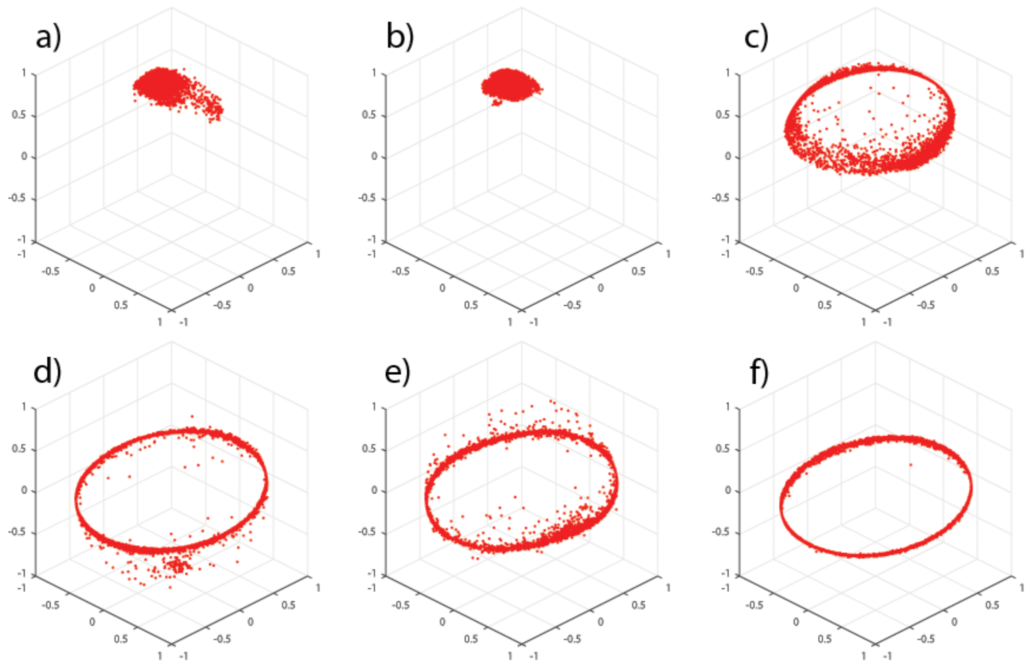


Figure 5.15: The predicted grip vectors for the targets in 5.14. a) Hammer pointer, b) Knife pointer, c) Berry pointer, d) Hammer palm, e) Knife palm, f) Berry palm.

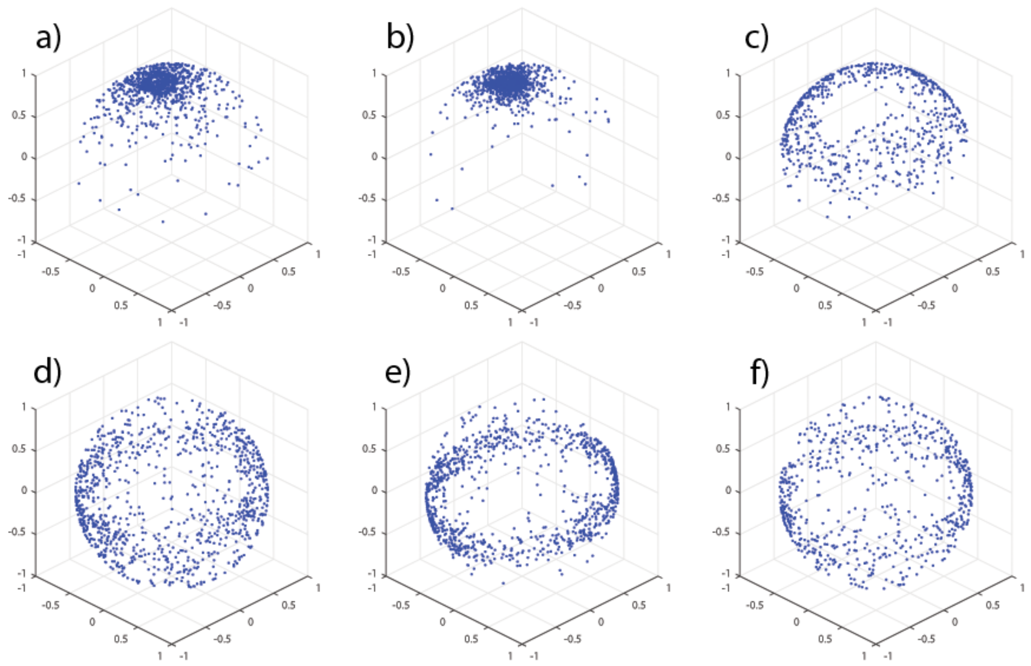


Figure 5.16: The target vectors for the different classes visualized with a scatter plot. Each point represents one training example in the hard set. a) Hammer point vector targets, b) Knife point vector targets, c) Berry point vector targets, d) Hammer palm vector targets, e) Knife palm vector targets, f) Berry palm vector targets.

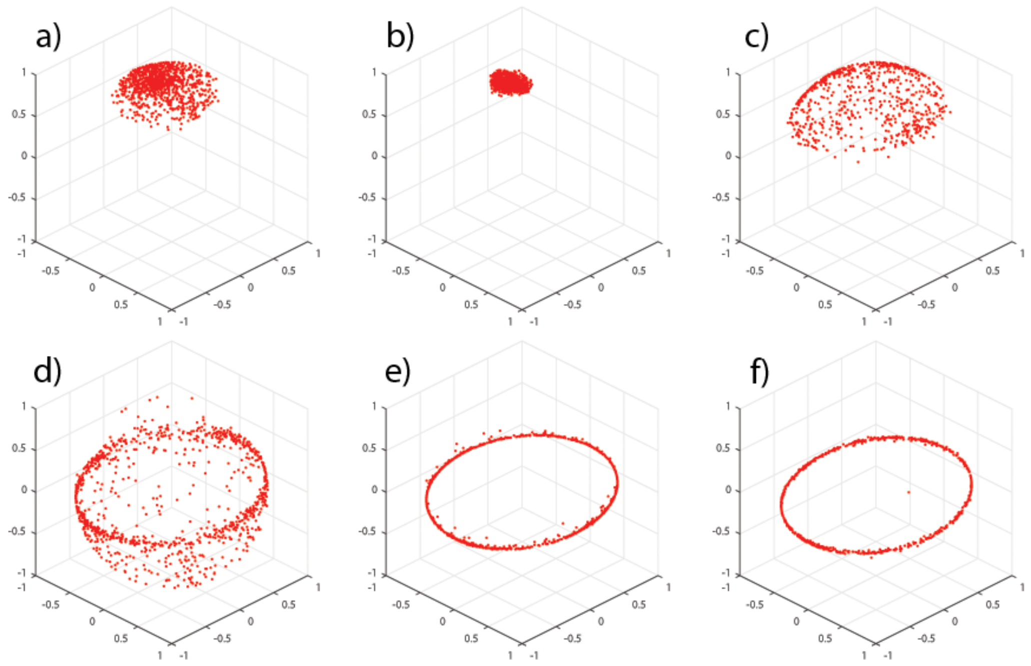


Figure 5.17: The predicted grip vectors for the targets in 5.16. a) Hammer pointer, b) Knife pointer, c) Berry pointer, d) Hammer palm, e) Knife palm, f) Berry palm.

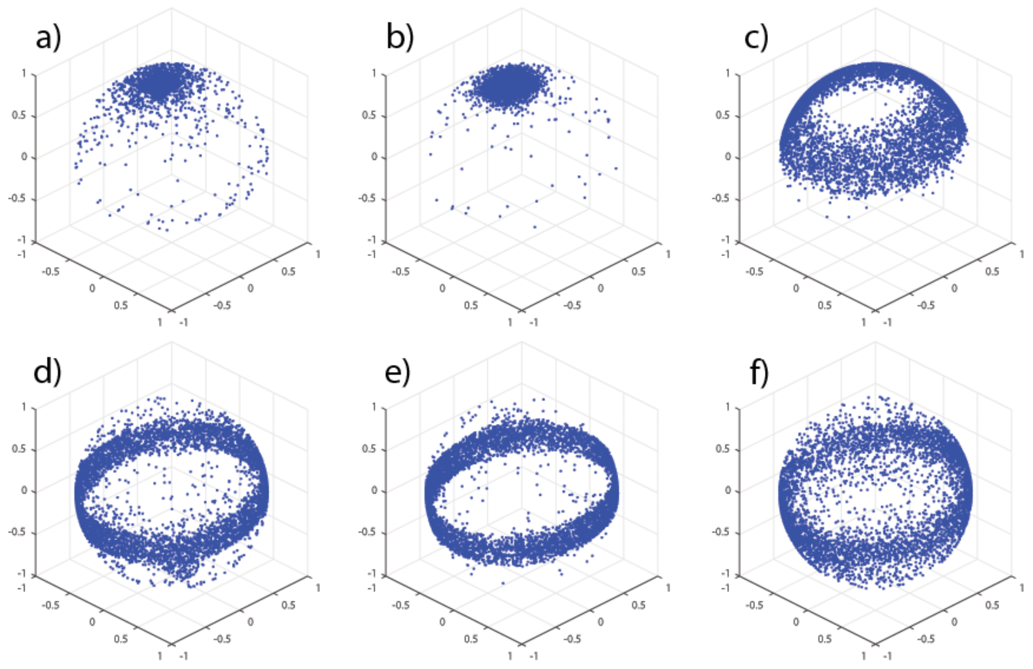


Figure 5.18: The target vectors for the different classes visualized with a scatter plot. Each point represents one training example in the combined set. a) Hammer point vector targets, b) Knife point vector targets, c) Berry point vector targets, d) Hammer palm vector targets, e) Knife palm vector targets, f) Berry palm vector targets.

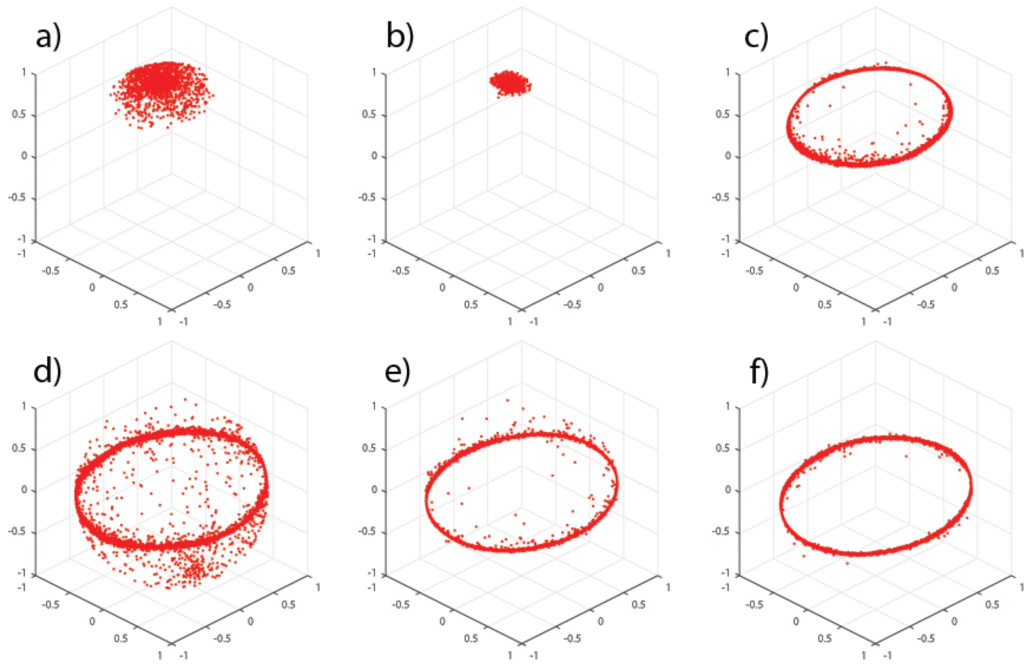


Figure 5.19: The predicted grip vectors for the targets in 5.18. a) Hammer pointer, b) Knife pointer, c) Berry pointer, d) Hammer palm, e) Knife palm, f) Berry palm.

The average errors in degrees for each vector is shown in tables 5.10, 5.11 and 5.12. The average error gotten by guessing the empirical mean of the data set is also shown to highlight that a small error not necessarily means that the network has learned something interesting. The histograms of the errors are shown in figures 5.20, 5.21 and 5.22.

Object	Vector	avg. error	avg. error guessing empirical mean vector
Hammer	pointer	6.1°	9.9°
	palm	8.6°	89.4°
Knife	pointer	6.5°	9.4°
	palm	23.9°	89.1°
Berry	pointer	20.3°	57.1°
	palm	19.4°	86.7°

Table 5.10: Average errors for the different vectors in the easy CV-set.

Object	Vector	avg. error	avg. error guessing empirical mean vector
Hammer	pointer	16.3°	20.5°
	palm	21.1°	85.8°
Knife	pointer	13.1°	14.5°
	palm	25.1°	85.9°
Berry	pointer	46.6°	56.8°
	palm	29.9°	87.9°

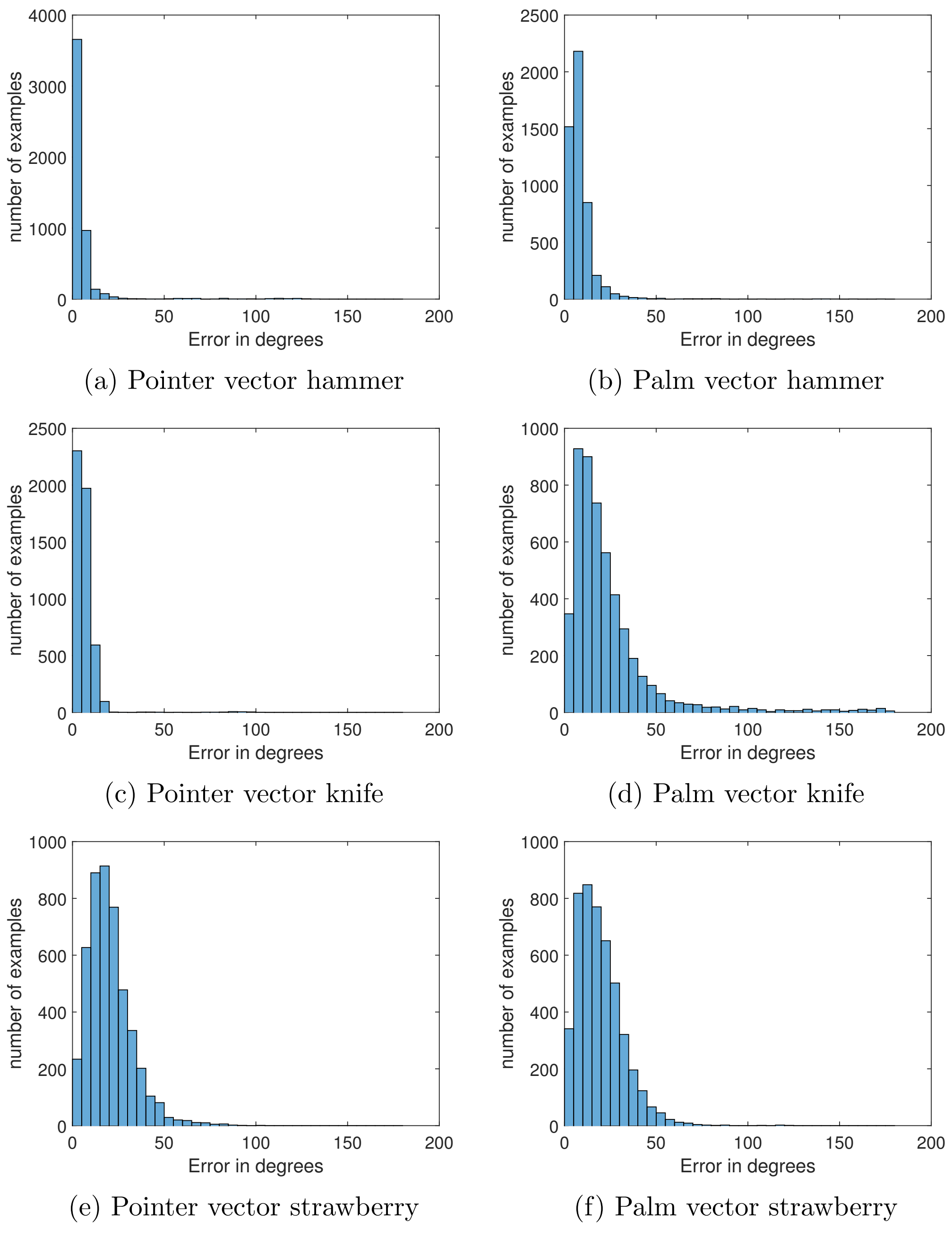
Table 5.11: Average errors for the different vectors in the hard CV-set.

Object	Vector	avg. error	avg. error guessing empirical mean vector
Hammer	pointer	8.6°	11.7°
	palm	13.9°	89.3°
Knife	pointer	9.7°	10.2°
	palm	16.3°	88.6°
Berry	pointer	13.4°	57.0°
	palm	19.7°	87.0°

Table 5.12: Average errors for the different vectors in the combined CV-set.

The networks are good at recognizing rotations around the camera's z-axis. For the hammer and knife objects, this corresponds to locating good palm vectors. For the berry object, which should be gripped by the stem, the rotation around the z-axis is captured by both the pointer and the palm vector. Rotation around the camera's x- and y-axis, seems to be more difficult to capture. Some of the rotation does seem to be captured by the pointer vector and it is very likely that the performance can be improved significantly by babysitting the learning process more carefully.

The results on both vectors are poorest for the berry object. It is quite possible that it is most difficult to capture the orientation of this object out of the three that were tested, but some of the errors are certainly the result of bad labelling of the data. The object has a finite set of vectors associated with it at instantiation and one of them is chosen when a training example is generated (see figure 4.3c). As described in chapter 4.1.3 the grip vector with the pointer component closest to pointing in the world-up-direction is chosen if it is not colliding. This results in a rather arbitrary palm vector which will be on average parallel to the world plane. Therefore some of the predictions might actually be more reasonable than the true target vector.

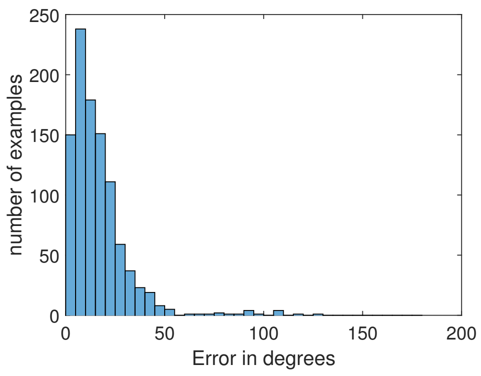


(a) Pointer vector hammer (b) Palm vector hammer

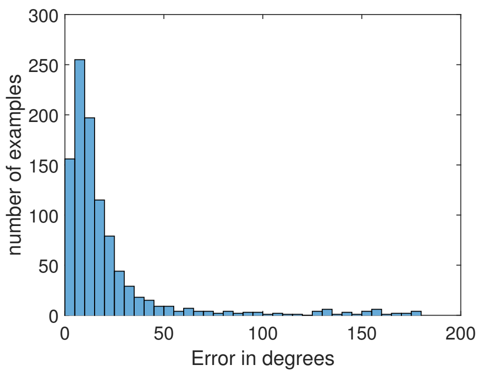
(c) Pointer vector knife (d) Palm vector knife

(e) Pointer vector strawberry (f) Palm vector strawberry

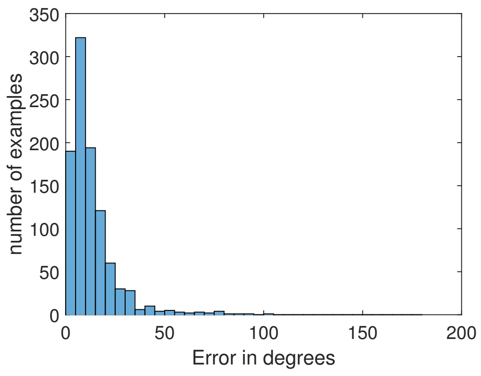
Figure 5.20: Histograms for vector estimates on the easy data set.



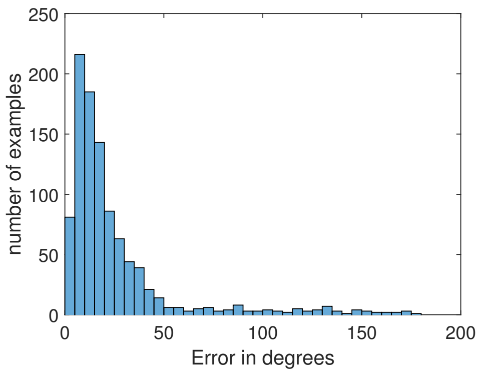
(a) Pointer vector hammer



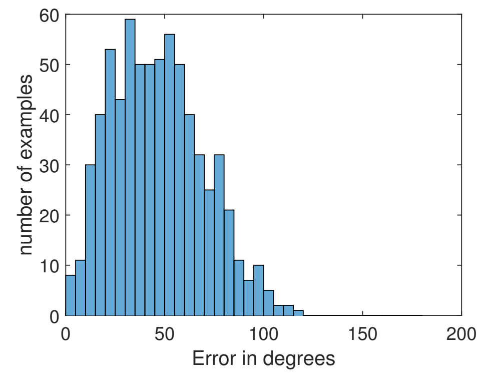
(b) Palm vector hammer



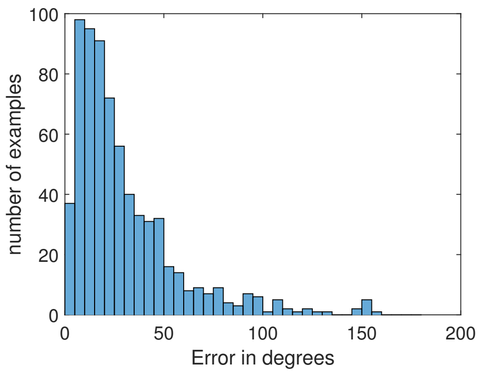
(c) Pointer vector knife



(d) Palm vector knife

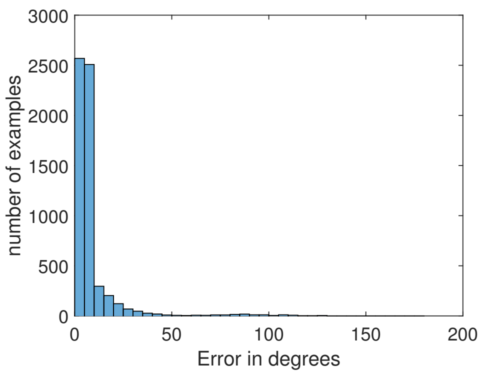


(e) Pointer vector strawberry

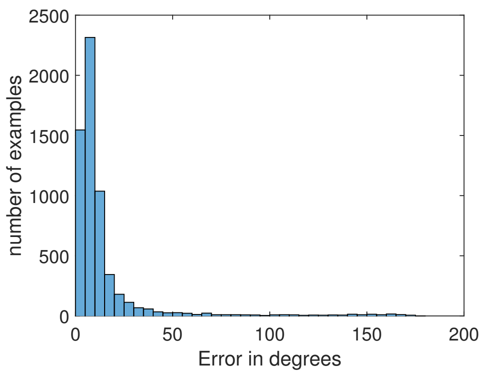


(f) Palm vector strawberry

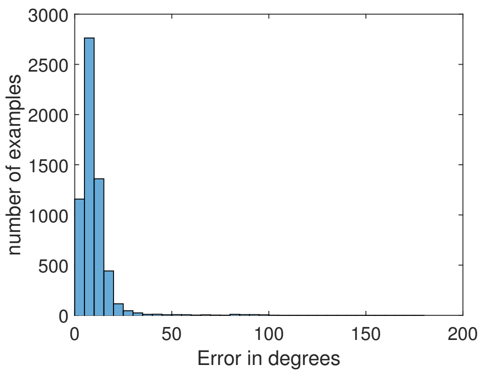
Figure 5.21: Histograms for vector estimates on the hard data set.



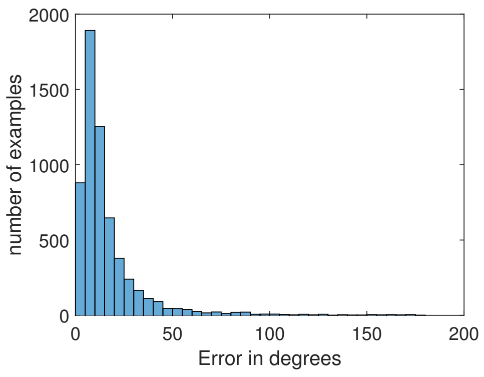
(a) Pointer vector hammer



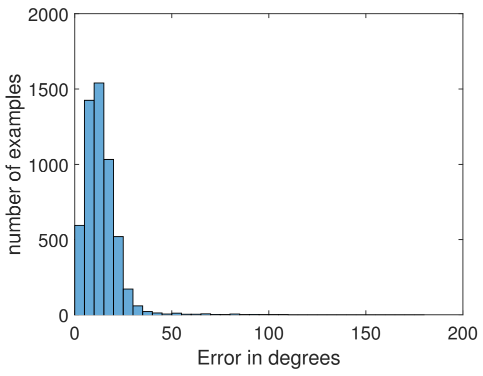
(b) Palm vector hammer



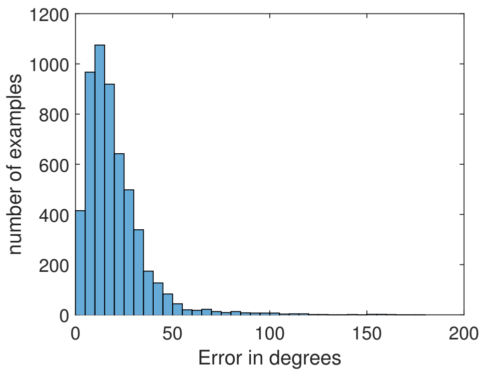
(c) Pointer vector knife



(d) Palm vector knife



(e) Pointer vector strawberry



(f) Palm vector strawberry

Figure 5.22: Histograms for vector estimates on the combined data set.

5.5 Live testing in the virtual environment

After training, the full system for classification, grip point and grip vector estimation was tested live in the virtual environment. The same objects were instantiated as during training data generation, only this time without their pre-defined grip vectors. The virtual camera can be moved around freely and depth images generated and sent to python. When python receives an image, noise is added before the image is classified and a class name, grip point and grip vector is returned if an object is recognized.

Experiments with the live system has shown that the neural network is very good at finding grip points and grip vectors for all three objects in the most common poses. When the objects are lying in typical poses on the plain background, good grasps are found virtually every time. This is true even for some of the less common poses, such as the hammer in figure 5.23.

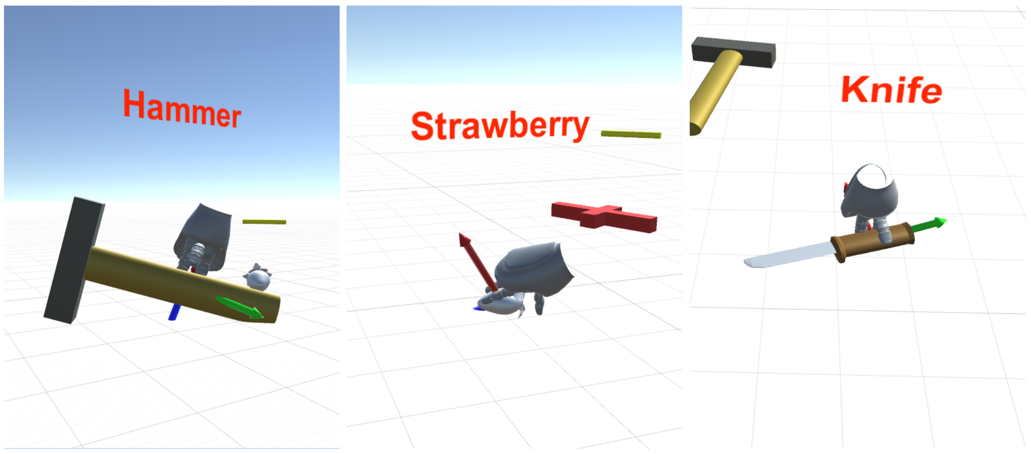


Figure 5.23: The objects with predicted grasps in poses that are common in the training set.

The results are also very good over all for more challenging cases with clutter and occlusion present in the images. The classifier handles this very well, but there are somewhat more errors in the grip point and grip vector estimates for unusual poses. Nevertheless, many of the proposed grasps are probably "good enough" for successful grasping to be achieved in the real world, see figure 5.24.

For very unusual poses, such as the knife standing straight up balancing on its shaft, the grip point and vector estimates are often completely off. However, the classifier works surprisingly well and it rarely has trouble unless there are several objects gathered within the field of view partially overlapping. Currently, the classifier only has four options to chose from and is likely to miss more when the number of classes is increased. It is also easy to fool by introducing objects never seen before in training. A round-like object, yet with many edges was created by rotating seven cubes around the same coordinate in world space. As is shown in figure 5.25, this objects is recognized both as a strawberry and a knife when it is re-sized to match those objects in size. The classifier is 98 % certain that the object is a berry in the image to the left and estimates that it is a 57 % probability for it being a knife and 43 % probability that it is nothing in the image to the right.

All subsystems have trouble when an object is further from the center of the image or the distance to the camera is different than it was in any of the examples

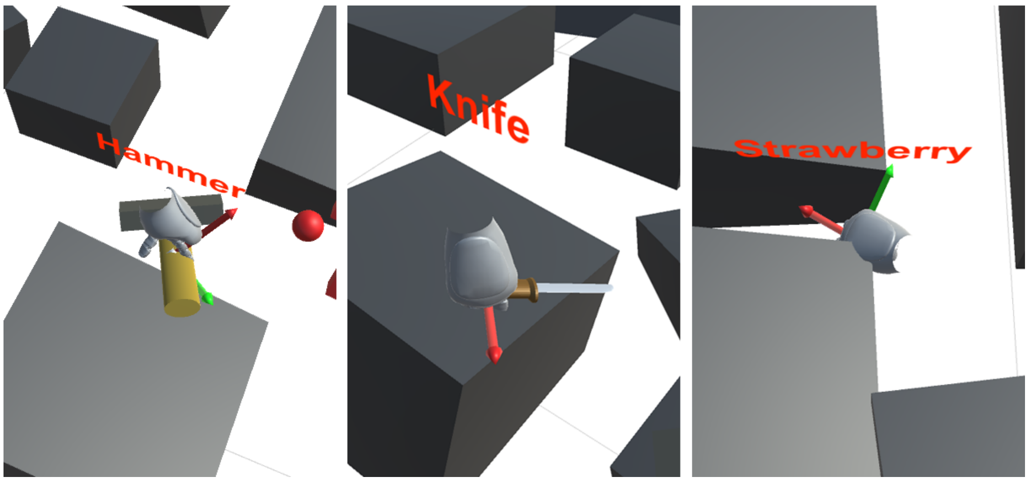


Figure 5.24: Predicted grasps for the objects in less common poses with clutter present in the scene.

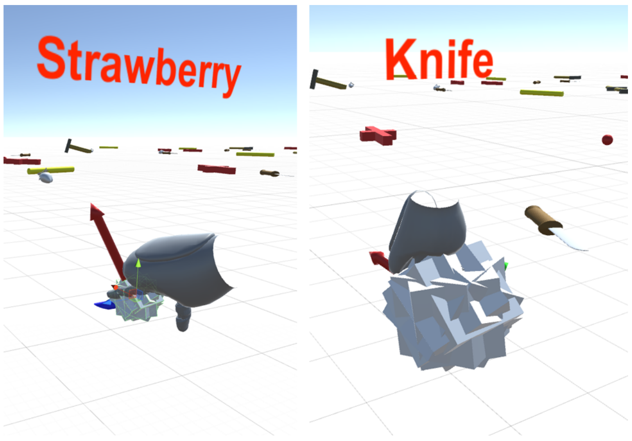


Figure 5.25: The cube objects misclassified as a berry and a knife.

in the training set. This is illustrated in figure 5.26, where the classifier refuses to output a grip point that is more than 1050 mm from the camera. This is for the most part an issue in the depth direction, because if the object is far enough from the center in the x-, y-plane, no object is detected and no grip vector is given.

Testing has shown that the grip point estimator is capable of predicting grip points that are occluded for the camera. Some predicted grasps for a partially occluded knife object is shown in figure 5.27 and the field of view for the virtual Shapecrafter corresponding to each grasp is shown in figure 5.28. Grip points seem to be harder to find for the occluded knife than the other classes, possibly because of the knife's more symmetric shape. As is seen in 5.27 and 5.28 c) the grip point can be placed on the blade of the knife, in this case when the entire shaft is occluded. It is also clear that the grip vectors are off by quite a bit and this tends to happen when there are sharp edges not associated with the object close to the grip point. However, the virtual hands indicating the grasp angle above the grip points seems to show that successful grasps can be achieved, even if the vectors are not 100 % accurate.

In some cases the grip point and grip vector can be completely off. In figure 5.29 the depth sensor was aimed so that the strawberry was approximately centered in

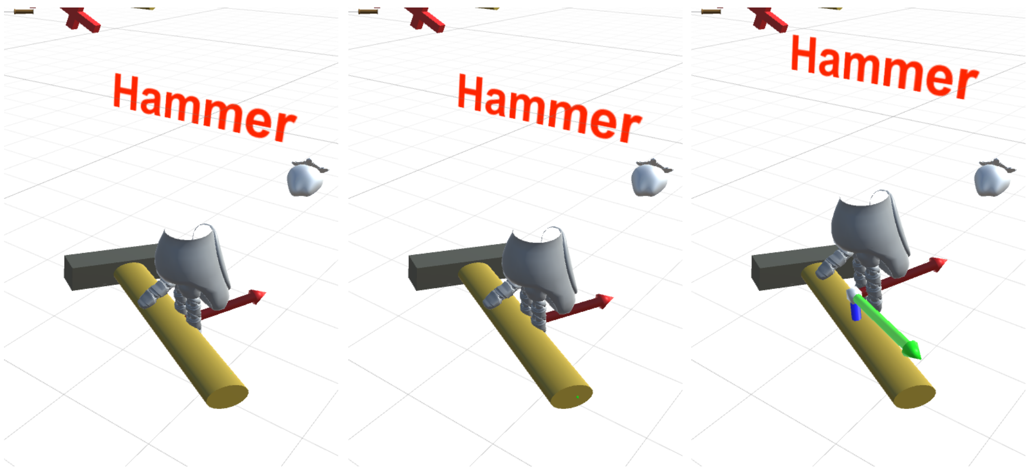


Figure 5.26: Estimated grasps for the same object with increasing distance to the camera. First image: 1000 mm, second: 1050 mm, third: 1100 mm.

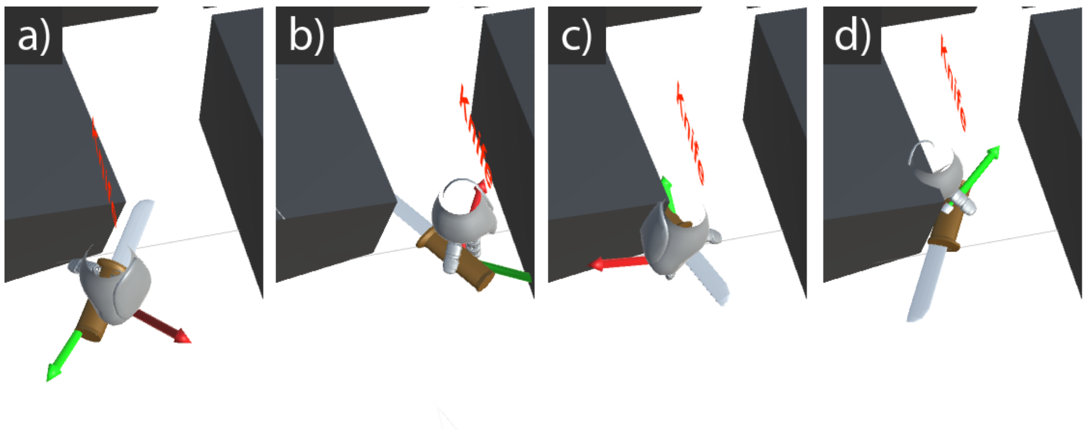


Figure 5.27: Some predicted grasps for the occluded knife object.

the frame. However, the image was classified as a hammer, and thus the system tried to find a suitable hammer grasp for a hammer that was further than the ±5 cm limit from the center of the frame. This shows that a precise object detector is important if the following two sub systems shall be expected to perform well.

It was a concern that the networks would learn specific areas in the scene as berries and knives because the different objects tend to land in different places in the environment. This has not turned out to be an issue for the classifier. However, the environment, in addition to the object itself, is clearly considered by the network when grip vectors are estimated. As an example, the grip vector for the berry in figure 5.24 seems to be more a result of the narrow space between the boxes than the berry itself. Meaning, when the berry was rotated manually and a grip vector was estimated again, approximately the same grip vector was given.

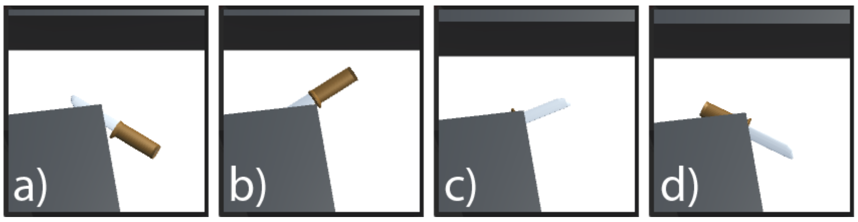


Figure 5.28: The FOV's for the depth camera corresponding to the images in figure 5.27

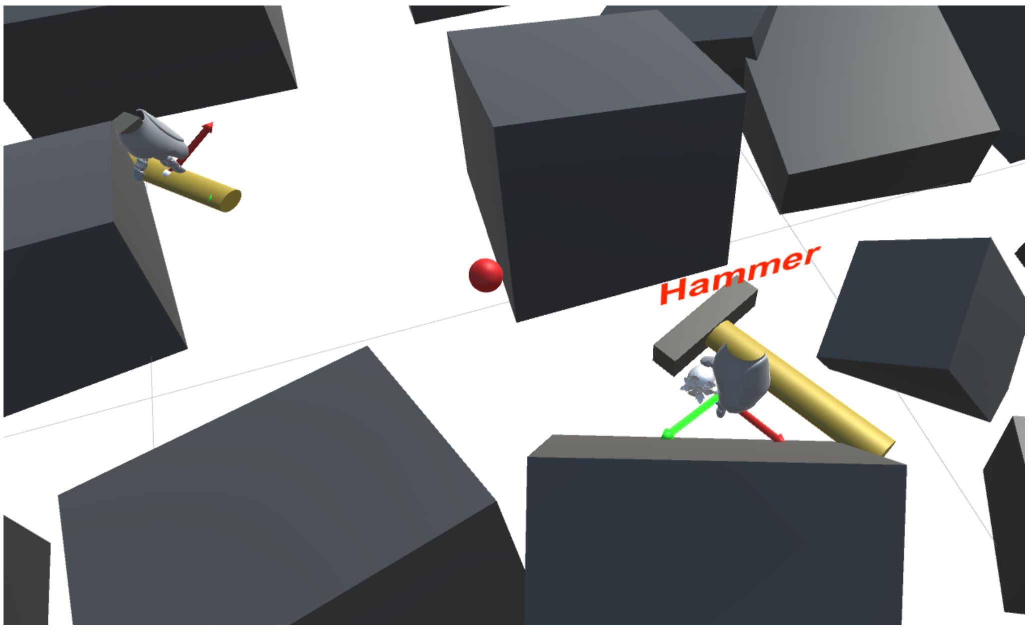


Figure 5.29: A bad grip estimate as a result of bad classification.

Chapter 6

Discussion

In this thesis it has been shown that convolutional neural networks can be used with good results to estimate grasps in a virtual environment solely from synthetic depth images. The motivation for this experiment is that if something works in a realistic virtual environment it is also likely to work in the real world. By working with depth images alone, as opposed to RGB-D images, it is easier to create synthetic data that meets the demand for high realism. A method for generation of large amounts of realistic looking synthetic depth images has been developed. This makes it possible to train deep neural networks from scratch with task specific labelled data.

A system for labelling of real world data which in turn enables easy communication with and repurposing of robots using a head mounted display is also proposed in this thesis.

6.1 The virtual environment

Proposed is the use of simulated physics in a virtual environment for generation of large amounts of realistic looking synthetic depth images. The physics simulation ensures that the synthetic images are of objects in natural poses and positions, which is vital for data that is to be used for grasp detection. The physics simulation also ensures that the data set contains images of objects with different types of realistic occlusion and clutter.

Through instantiation of objects with pre-defined grip vectors and grip points, precise labelling can be achieved for a wide variety of computer vision tasks including grasp detection, classification/object detection and instance segmentation. It is easy to introduce new objects in the VR environment by providing a 3D model, defining grip vectors and setting deformation and random scaling limits. By training in VR, the neural network can be exposed to many thousands of different scenarios, including ones that only rarely occur in the real world.

The ultimate goal of the virtual environment is to produce data that is so similar to real data that it can serve almost as a complete replacement. To achieve this goal, realistic looking depth images needs to be rendered of the virtual scenes and the virtual depth camera used in the VR-environment was created to resemble the real Shapecrafter depth camera. Visual comparison of generated and real depth images show that the images from both sources are similar, but not identical. The data is similar enough for it to be likely that the parameters from the neural networks trained on synthetic data are transferable to real world scenarios.

In total, the virtual environment works well and as expected. However, there are some drawbacks with the system and areas that can be improved.

If a network shall be trained task specifically in VR, there is a need for 3D models with pre-defined grip vectors. This might present a hurdle if no models exist and some objects can take on a wide variety of shapes (e.g. chairs), creating a need for several models. There are large amounts of free 3D models available online which can be used, but grip vectors will have to be placed manually. However, once the grip vectors have been placed, a virtually infinite number of training examples can be created. Training a grasp detection network *task agnostically* might reduce the need for models as long as the models that are used are representative for most other types of objects.

The heuristics used in this project for choosing the best grip vector is oversimplified and needs to be improved. Specifically, the best grip vector for an object is not only given by the object itself and its surroundings, but also the robot's relative position to the object and possibly also the robot's pose. The small number of pre-defined grip vectors is probably also less than optimal, and a new way of defining vectors in general, perhaps as volumes of legal positions might give better results.

A fundamental drawback with the whole system for synthetic data generation is that it does not expand easily to include rendering of realistic looking color images. This does obviously limit the capability of the system, i.e. it can not be used to sort *m&m's*. However, color information might be more important for object detection than it is for grip estimation and it could therefore be possible to use a network trained on existing color image data sets in tandem with the grasp detector network.

The current ways in which the objects in the VR environment are deformed on instantiation is not enough to create realistic looking data for non-rigid objects. Better ways of deforming the objects realistically in contact with the environment are needed, e.g. like a fish is bent when it is put in a small bucket. These types of realistic physics interactions are common in the games of today, and can be modelled in the virtual environment.

The virtual depth camera can be improved in several ways to better approximate the real depth camera. Specifically, the real depth images have "3D shadows" because of the way the light pattern emitting projector is off-set with respect to the camera in the Shapecrafter. These shadows are not present in the synthetic data. Additionally, in the real world too much of the emitted light from the projector tends to be reflected back to the camera from areas in the scene with surface normals at a certain angles to the projector and camera. Because these effects are mostly a result of the projector and camera positions and the objects in the scene and their surface normals, they can be modelled in the VR-environment. However, different objects reflect light differently and some simplifications will have to be made.

6.1.1 Future work

The possibilities for expansion of the virtual environment are huge. The function of the virtual environment is to create synthetic data, which can replace the need for real world data. If this is going to be achieved the synthetic data should be as similar as possible to real data and the emphasis of future work should therefore be on improving the system in areas that gets us closer to that goal with the least amount of effort.

Large improvements in the output from the virtual depth camera can be achieved rather quickly with simulated 3D-shadows and reflections. This will increase the quality of all generated data, regardless of application and is therefore a priority. Additionally, more realistic objects and sceneries will be introduced. Realistic deformation of 3D models in contact with the environment is also important for handling of biological material and will also be explored in future work.

Further down the line, efforts will be made in "gamification" of the entire pipeline for synthetic data generation. The finished system can then be released as an executable which can enable people to create their own data sets. Plans for creation of a large realistic data set for classification, object detection, instance segmentation and grasp detection which can be released online are also being made.

6.2 Neural networks

Three separate neural networks have been trained in this project for object recognition, grip point and grip vector estimation respectively. The grip point refers to the position of an end-effector and the grip vector refers to the end-effectors rotation. All networks have yielded good performance and shown that convolutional neural networks can be used for grasp detection in the created virtual environment. The lack of labelled real world data has prevented testing of transfer learning from virtual reality to a real world application, and this will be tested in future work.

The predicted grasps have been found to not only be a result of the respective object in its current pose, but also the surrounding environment in which it lies. This means that the network can learn to avoid obstacles in the scene if good heuristics for choosing grip vectors are used in the virtual environment.

All sub systems have shown some robustness to occlusion and clutter and especially the classifier, which gets almost everything right in the validation data set. The grip point estimator has also shown that it is to some degree able to estimate grips that are hidden for the camera behind other objects.

None of the sub systems perform well when exposed to examples that are generated somewhat differently than the ones in the training set. For instance, the grip point estimator never predicts anything to be further than 1050 mm away from the depth camera, which was the limit set during generation of the training sets. Similarly, the network can learn to recognize objects in a cluttered scene very well, but fail when given the same objects in an uncluttered scene. This highlights the importance of exposing the network to large amounts of varied data during training.

The grip vector estimator is sometimes fooled by prominent edges in the image that are a part of the surrounding environment and not related to the pose of the object in question. This could perhaps be prevented if the background were to be successfully segmented out with an instance segmentation network.

Good results for all sub systems were achieved for many different architectures and it became clear that the generated data set was too simple to enable proper testing of different architectures. The architectures presented here are therefore quite possibly only well fit for the given data sets and not for grasp detection from depth images in general. The classes in the data set turned out to be very easily separable, and the clutter in the environment does only counteract this to some degree. The scenery and objects in the data sets are too ideal-looking with sharp edges and no texture. The networks will have to be retrained on more realistic data if the features learnt in VR shall have any hope of being transferred to a real world application successfully.

6.2.1 Future work

In future work, training and testing of the grasp detector should be done on much larger, more realistic synthetic data sets, with many more classes. This will enable good testing of architectures and testing of transfer learning from virtual reality to the real world.

It would also be interesting to investigate the possibility of replacing the object detector with an instance segmentation network. Because the ground truth maps needed for training of such networks already are created by the virtual environment, testing of this is currently possible.

6.3 Interface between man and machine

The use of virtual reality as an intuitive way of communicating with robots is also proposed. With a head mounted display and by loading a point cloud from the depth camera in VR, a user can step into the robots shoes and see the world, literally, through its eyes. A set of tracked hand controllers appear to the user as two robotic hands and these can be used to show the robot how different objects should be grasped.

Walking around and placing grip vectors in this way has proven to be very intuitive. To the user, no other means of interacting with the world is apparent than through the use of his or hers virtual hands and the user is able to place grip vectors quickly without the need for instructions of any kind.

This environment can be used for creation of real world labelled data, which in turn can be used to test the transferability of the features learned by the neural networks in VR. The real data can also be used to fine-tune the network trained in VR and to retrain the top layers in the neural networks if a robot needs to be repurposed to perform some new, never seen before task.

The technology for the head mounted display is very new, and the hardware needed to create the system only just arrived in May. This did not leave enough time to fully develop the system, label real data and test the transferability of the features learned in VR. The current state of the system is that point clouds can be loaded one by one and users can walk around and place grip vectors in the scene with the hand controllers. This is effortless, and very good frame rate is achieved with the point clouds from the Shapecrafter (2.3 million points). Functionality for transferring live images from the Shapecrafter is needed before the system is practical to use for generation of substantial amounts of real data.

Ultimately, the goal is that this system can be used to train a robot to perform a task never seen before, on site in a factory, by showing it a few times how it is done. For this to be possible the robot will need to be pre-trained extensively, both in the virtual environment, and on some real data, so that it comes with a good set of feature extractors in the lower layers of the neural network. Only the top layers with fewer parameters will then need to be retrained on site, thus reducing the need for data dramatically. There is still a long way to go before this is realised, and among the things that needs to be figured out for this to become a reality, is how the network can be trained online and automatically in an efficient manner while labelling is performed.

6.3.1 Future work

Future work will mainly revolve around creating a good interface between the virtual environment and the depth camera in cooperation with the developers of the Shapecrafter. There are also exciting possibilities for expansion of the system, where it can be used not only to place grasps, but entire paths for a robot to learn. Another possible application of the system is in teleoperation of robots.

6.4 Overall

This project has demonstrated the feasibility of the proposed system for grasp detection with deep neural networks from depth images. Large amounts of realistic looking synthetic depth images can be generated with the use of simulated physics in a virtual environment, and deep neural networks can be trained to predict good grasps based on these images. Additionally, a system for intuitive communication with robots using a head mounted display has been presented. This may enable more flexible robots, which are not restricted to performing tasks they have seen before, but can be repurposed without the need for reprogramming after a period of training on site.

There are several ways in which the whole system can be improved and expanded upon, and this will be the focus of future work. Testing of the neural networks trained in VR on real data will need to be done before the feasibility of the presented methodology for training of neural networks in VR for real world application can be securely confirmed. No fundamental weakness with the method has been exposed in the experiments conducted in this project and the work has served as a good first step towards a fully functional system.

Chapter 7

Conclusion

A novel approach for generation of synthetic data for grasp detection has been proposed in this thesis. With the use of simulated physics, large amounts of synthetic depth images of objects in realistic poses is generated. With heuristics for choosing the best grip vector for an object in its current pose, a data set with precise labels which are a function of the object in relation to its environment is created.

A convolutional neural network has been trained and shown to be able to find good grasps from the synthetic depth images in an easy data set. The estimated grasps are not only object-, but also environment dependent.

Future work will focus on the improvement of the virtual environment and generation of a large, realistic data set for a specific grasping task. This will in turn enable testing of transfer learning from the virtual environment to the real world.

A novel way of creating real world data sets for grasping using a head mounted display and tracked hand controllers is also proposed. In future work it will be explored how this can enable easy repurposing of robots without the need for reprogramming, by retraining the top layers in a neural networks with new data.

Bibliography

[1] The Henn na Hotel. *Henn na Hotel General Concept.* 2016. URL: http://www.h-n-h.jp/en/concept/.

[2] ABB robotics. *ABB introduces design and color change for a new era of robotics.* 2016. URL: http://www.abb.com/cawp/seitp202/.

[3] Olga Russakovsky et al. "ImageNet Large Scale Visual Recognition Challenge". In: *International Journal of Computer Vision (IJCV)* 115.3 (2015), pp. 211–252. DOI: 10.1007/s11263-015-0816-y.

[4] Unity. *Unity3D game engine.* 2016. URL: https://unity3d.com/unity.

[5] Sergey Levine et al. "Learning Hand-Eye Coordination for Robotic Grasping with Deep Learning and Large-Scale Data Collection". In: *CoRR* abs/1603.02199 (2016). URL: http://arxiv.org/abs/1603.02199.

[6] Sergey Levine. *Deep Learning for Robots: Learning from Large-Scale Interaction.* 2016. URL: https://research.googleblog.com/2016/03/deep-learning-for-robots-learning-from.html.

[7] A. Saxena I. Lenz H. Lee. "Deep learning for detecting robotic grasps". In: *The International Journal of Robotix Research* (2015).

[8] Erroll Wood et al. "Rendering of Eyes for Eye-Shape Registration and Gaze Estimation". In: *CoRR* abs/1505.05916 (2015). URL: http://arxiv.org/abs/1505.05916.

[9] Jamie Shotton et al. "Real-Time Human Pose Recognition in Parts from a Single Depth Image". In: *CVPR.* IEEE, June 2011. URL: http://research.microsoft.com/apps/pubs/default.aspx?id=145347.

[10] Shuai Zheng et al. "Object Proposal Estimation in Depth Images using Compact 3D Shape Manifolds". In: *German Conference on Pattern Recognition (GCPR).* 2015.

[11] Saurabh Gupta et al. "Aligning 3D Models to RGB-D Images of Cluttered Scenes". In: *Computer Vision and Pattern Recognition (CVPR).* 2015.

[12] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. "Learning representations by back-propagating errors". In: *Nature* 323 (1986). DOI: doi:10.1038/323533a0.

[13] Andrej Karpathy, Justin Johnson, and Fei-Fei Li. *CS231n Convolutional Neural Networks for Visual Recognition Lecture Notes.* 2016. URL: http://cs231n.stanford.edu/syllabus.html.

[14] Xavier Glorot and Yoshua Bengio. “Understanding the difficulty of training deep feedforward neural networks”. In: *International conference on artificial intelligence and statistics* (2010), pp. 249–256.

[15] Kaiming He et al. “Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification”. In: *CoRR* abs/1502.01852 (2015). URL: `http://arxiv.org/abs/1502.01852`.

[16] Sergey Ioffe and Christian Szegedy. “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift”. In: *CoRR* abs/1502.03167 (2015). URL: `http://arxiv.org/abs/1502.03167`.

[17] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. “ImageNet Classification with Deep Convolutional Neural Networks”. In: *Advances in Neural Information Processing Systems* 25 (2012).

[18] Michael A. Nielsen. *Neural Networks and Deep Learning.* Determination Press, 2015.

[19] Nitish Srivastava et al. “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”. In: *Journal of Machine Learning Research* 15 (2014).

[20] Yann LeCun, Corinna Cortes, and Cristopher J.C. Burges. *The MNIST database of handwritten digits.* 2016. URL: `http://yann.lecun.com/exdb/mnist/`.

[21] D.H.Hubel and T.N.Wiesel. “Receptive fields of single neurones in the cat’s striate cortex”. In: *J. Physiol* 148 (1959), pp. 574–591.

[22] Yann LeCun et al. “Gradient-Based Learning Applied to Document Recognition”. In: *Proc of the IEEE* (1998).

[23] Andreas Müller Dominik Scherer and Sven Behnke. “Evaluation of Pooling Operations in Convolutional Architectures for Object Recognition”. In: *20th International Conference on Artificial Neural Networks* (2010).

[24] Jawad Nagi et al. “Max-Pooling Convolutional Neural Networks for Vision-based Hand Gesture Recognition”. In: *2011 IEEE International Conference on Signal and Image Processing Applications* (2011).

[25] Pierre Sermanet et al. “OverFeat: Integrated Recognition, Localization and Detection using Convolutional Networks”. In: *CoRR* abs/1312.6229 (2013). URL: `http://arxiv.org/abs/1312.6229`.

[26] Karen Simonyan and Andrew Zisserman. “Very Deep Convolutional Networks for Large-Scale Image Recognition”. In: *CoRR* abs/1409.1556 (2014). URL: `http://arxiv.org/abs/1409.1556`.

[27] Christian Szegedy et al. “Going Deeper with Convolutions”. In: *CoRR* abs/1409.4842 (2014). URL: `http://arxiv.org/abs/1409.4842`.

[28] Dumitru Erhan et al. “Why Does Unsupervised Pre-training Help Deep Learning”. In: *Journal of Machine Learning Research* (2010), pp. 625–660.

[29] Valve. *Virtual Reality - SteamVR featuring the HTC Vive.* 2016. URL: `https://www.youtube.com/watch?v=qYfNzhLXYGc`.

[30] Epic Games. *Unreal game engine.* 2016. URL: `https://www.unrealengine.com/`.

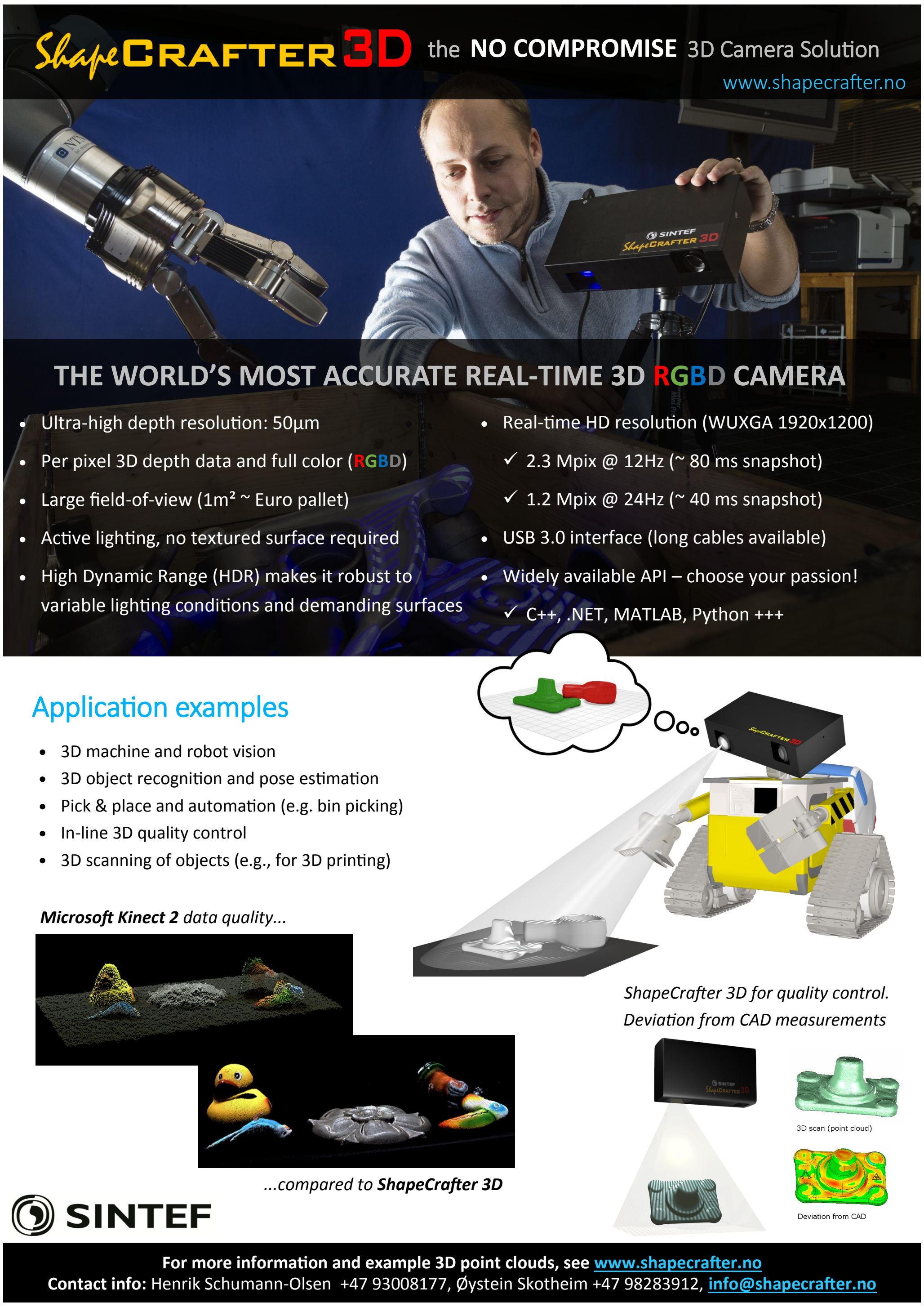
[31] Mgear. *Point Cloud Viewer and Tools*. 2016. URL: `https://www.assetstore.unity3d.com/en/#!/content/16019`.

[32] Valve Corporation. *SteamVR Plugin*. 2016. URL: `https://www.assetstore.unity3d.com/en/#!/content/32647`.

[33] Ilya Sutskever et al. "On the importance of initialization and momentum in deep learning". In: *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*. Ed. by Sanjoy Dasgupta and David Mcallester. Vol. 28. 3. JMLR Workshop and Conference Proceedings, May 2013, pp. 1139–1147. URL: `http://jmlr.org/proceedings/papers/v28/sutskever13.pdf`.

Appendix A

A.1 Shapecrafter fact sheet



ShapeCRAFTER 3D the **NO COMPROMISE** 3D Camera Solution

www.shapecrafter.no

THE WORLD'S MOST ACCURATE REAL-TIME 3D RGBD CAMERA

- Ultra-high depth resolution: 50µm
- Per pixel 3D depth data and full color (RGBD)
- Large field-of-view (1m² ~ Euro pallet)
- Active lighting, no textured surface required
- High Dynamic Range (HDR) makes it robust to variable lighting conditions and demanding surfaces
- Real-time HD resolution (WUXGA 1920x1200)
  - ✓ 2.3 Mpix @ 12Hz (~ 80 ms snapshot)
  - ✓ 1.2 Mpix @ 24Hz (~ 40 ms snapshot)
- USB 3.0 interface (long cables available)
- Widely available API – choose your passion!
  - ✓ C++, .NET, MATLAB, Python +++

Application examples

- 3D machine and robot vision
- 3D object recognition and pose estimation
- Pick & place and automation (e.g. bin picking)
- In-line 3D quality control
- 3D scanning of objects (e.g., for 3D printing)

***Microsoft Kinect 2** data quality...*

*...compared to **ShapeCrafter 3D***

ShapeCrafter 3D for quality control. Deviation from CAD measurements

For more information and example 3D point clouds, see **www.shapecrafter.no**
Contact info: Henrik Schumann-Olsen +47 93008177, Øystein Skotheim +47 98283912, **info@shapecrafter.no**

A.2 Importance of non-linear activation function

The activation function $f(\cdot)$ is sometimes referred to as "the non-linearity". The non-linearity is important because without it any neural network could be simplified to a less capable one-layer model:

$$\mathbf{a}^{(3)} = f(\mathbf{W}^{(2)}\mathbf{a}^{(2)}) = \mathbf{W}^{(2)}\mathbf{a}^{(2)}$$
$$\mathbf{a}^{(2)} = f(\mathbf{W}^{(1)}\mathbf{x}) = \mathbf{W}^{(1)}\mathbf{x}$$
$$\Rightarrow$$
$$\mathbf{a}^{(3)} = \mathbf{W}^{(2)}\mathbf{W}^{(1)}\mathbf{x} = \mathbf{W}\mathbf{x}$$

A.3 Nesterov momentum

Derivation of Nesterov's momentum by Sutskever et. al [33].

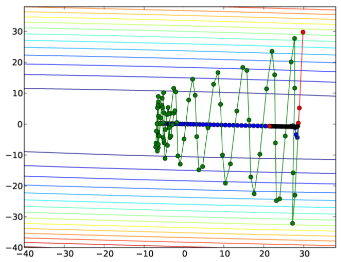


Figure 2. The trajectories of CM, NAG, and SGD are shown. Although the value of the momentum is identical for both experiments, CM exhibits oscillations along the high-curvature directions, while NAG exhibits no such oscillations. The global minimizer of the objective is at (0,0). The red curve shows gradient descent with the same learning rate as NAG and CM, the blue curve shows NAG, and the green curve shows CM. See section 2 of the paper.

Appendix

A.1 Derivation of Nesterov's Accelerated Gradient as a Momentum Method

Nesterov's accelerated gradient is an iterative algorithm that was originally derived for non-stochastic gradients. It is initialized by setting $k = 0$, $a_0 = 1$, $\theta_{-1} = y_0$, y_0 to an arbitrary parameter setting, z to an arbitrary parameter setting, and $\varepsilon_{-1} = \|y_0 - z\|/\|\nabla f(y_0) - \nabla f(z)\|$. It then repeatedly updates its parameters with the following equations:

$$\varepsilon_t = 2^{-i}\varepsilon_{t-1} \tag{6}$$

(here i is the smallest positive integer for which

$$f(y_t) - f(y_t - 2^{-i}\varepsilon_{t-1}\nabla f(y_t)) \geq 2^{-i}\varepsilon_{t-1}\frac{\|\nabla f(y_t)\|^2}{2})$$

$$\theta_t = y_t - \varepsilon_t \nabla f(y_t) \tag{7}$$

$$a_{t+1} = \left(1 + \sqrt{4a_t^2 + 1}\right)/2 \tag{8}$$

$$y_{t+1} = \theta_t + (a_t - 1)(\theta_t - \theta_{t-1})/a_{t+1} \tag{9}$$

The above presentation is relatively opaque and could be difficult to understand, so we will rewrite these equations in a more intuitive manner.

The learning rate ε_t is adapted to always be smaller than the reciprocal of the "observed" Lipshitz coefficient of ∇f around the trajectory of the optimization. Alternatively, if the Lipshitz coefficient of the derivative is known to be equal to L, then setting $\varepsilon_t = 1/L$ for all t is sufficient to obtain the same theoretical guarantees. This method for choosing the learning rate assumes that f is not noisy, and will result in too-large learning rates if the objective is stochastic.

To understand the sequence a_t, we note that the function $x \to \left(1 + \sqrt{4x^2 + 1}\right)/2$ quickly approaches $x \to x + 0.5$ from below as $x \to \infty$, so $a_t \approx (t + 4)/2$ for large t, and thus $(a_t - 1)/a_{t+1}$ (from eq. 9) behaves like $1 - 3/(t + 5)$.

Finally, if we define

$$v_t \equiv \theta_t - \theta_{t-1} \tag{10}$$

$$\mu_t \equiv (a_t - 1)/a_{t+1} \tag{11}$$

then the combination of eqs. 9 and 11 implies:

$$y_t = \theta_{t-1} + \mu_{t-1}v_{t-1}$$

which can be used to rewrite eq. 7 as follows:

$$\theta_t = \theta_{t-1} + \mu_{t-1}v_{t-1} - \varepsilon_{t-1}\nabla f(\theta_{t-1} + \mu_{t-1}v_{t-1}) \tag{12}$$

$$v_t = \mu_{t-1}v_{t-1} - \varepsilon_{t-1}\nabla f(\theta_{t-1} + \mu_{t-1}v_{t-1}) \tag{13}$$

where eq. 13 is a consequence of eq. 10. Alternatively:

$$v_t = \mu_{t-1}v_{t-1} - \varepsilon_{t-1}\nabla f(\theta_{t-1} + \mu_{t-1}v_{t-1}) \tag{14}$$

$$\theta_t = \theta_{t-1} + v_t \tag{15}$$

where $\mu_t \approx 1 - 3/(t + 5)$. (Nesterov, 1983) shows that if f is a convex function with an L-Lipshitz continuous derivative, then the above method satisfies the following:

$$f(\theta_t) - f(\theta^*) \leq \frac{4L\|\theta_{-1} - \theta^*\|^2}{(t + 2)^2} \tag{16}$$

To understand the quadratic speedup obtained by the momentum, consider applying momentum to a linear function. In this case, the i-th step of the momentum method will be of distance proportional to i; therefore N steps could traverse a quadratically longer distance: $1 + 2 + \cdots + N = O(N^2)$.

A.2 Details for Theorem 2.1

We will first formulate and prove a result which establishes the well known fact that first-order methods such CM and NAG are invariant to orthonormal transformations (i.e. rotations) such as U. In particular, we will will show that the sequence of iterates obtained by applying NAG and CM to the reparameterized quadratic p, is given by U times sequence of iterates obtained by applying NAG and CM to the original quadratic q. Note that the only fact we use about U at this stage is that it is orthonormal/unitary, not that it diagonalizes q.

Appendix B

B.1 Source code for the virtual environment

B.1.1 Main function

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class MainScript : MonoBehaviour {

public List<GameObject> AllObjects;
public string DataCollectionState = "nada";
public bool SwichStateOnNextUpdate = false;
public GameObject ShapeCrafterObj;
public string DataSetPath;
private DataCollector dataCollector;

void Start(){
DepthCamera ShapeCrafter = ShapeCrafterObj.
            GetComponent<DepthCamera> ();
ShapeCrafter.image_dir = DataSetPath;
dataCollector = this.GetComponent<DataCollector> ();
dataCollector.TrainingLabelsPath = DataSetPath +
            "\\" + "Labels.txt";
}


void Update () {

dataCollector.CollectData ();

// Initiate Scan
if (Input.GetButtonDown("Jump")==true) {
// Selecting Gripvectors and freezing physics
PrepareScene ();
DataCollectionState = "ScanEachObject";
dataCollector.DataCollectionState = DataCollectionState;
}
}


void PrepareScene(){
// Find all objects in scene
GameObject[] Objs = GameObject.
                FindGameObjectsWithTag ("HasNotBeenScanned");
for (int i = 0; i < Objs.Length; i++) {
AllObjects.Add (Objs[i]);
}

for (int i = 0; i < AllObjects.Count; i++) {

// Decide on grip vectors for all objects
GripVectors gvs = AllObjects [i].
            GetComponentInChildren<GripVectors> ();
if (gvs != null) {
```

```
gvs.ChooseGripVector ();
}

// Disable rigidbodies to avoid further movement
AllObjects [i].GetComponent<Rigidbody> ().
            isKinematic = true;
}
}
}
```

B.1.2 Script added to all scanable objects

```
using UnityEngine;
using System.Collections;

public class ObjectToPick : MonoBehaviour {
public bool HasBeenScanned;
public Vector3 GripVector;
public string name;
public float RandomScalingFactor = 0;

void Start() {
float randScale = Random.Range (1.0f-RandomScalingFactor,
                            1.0f+RandomScalingFactor);
this.transform.localScale =
       Vector3.Scale (this.transform.localScale,
       new Vector3(randScale,randScale,randScale));
}

}
```

B.1.3 Instantiation handler

```
using UnityEngine;
using System.Collections;

public class Instantiation : MonoBehaviour {

public int NumberOfObjects;
public float DropArea;
public Vector2 DropHeight;
public GameObject[] OriginalObject;
GameObject[] ObjectClone;

void Awake(){
ObjectClone = new GameObject[NumberOfObjects];
Vector3 pos;
for (int j = 0; j < OriginalObject.Length; j++) {
for (int i = 0; i < NumberOfObjects; i++) {
pos = new Vector3 (Random.Range (-DropArea / 2.0f,
                   DropArea / 2.0f),
                   Random.Range (DropHeight[0],
                   DropHeight[1]),
                   Random.Range (-DropArea / 2.0f,
                   DropArea / 2.0f));
ObjectClone [i] = Instantiate (OriginalObject [j],
        pos, Random.rotation) as GameObject;
ObjectClone [i].tag = "HasNotBeenScanned";
}
}
}
}
```

B.1.4 Data Collector object

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class DataCollector : MonoBehaviour {

public Camera VirtualShapeCrafterCamera;
public GameObject VirtualShapeCrafter;
public float ExamplesGenerated;
public string DataCollectionState;
public bool SaveData;
public float DistanceToFocalPoint;
public List<GameObject> ObjectsNotScanned;
public Vector2 RotationLimitsTilt;
public Vector2 RotationLimitsPan;
public Vector2 RotationLimitsZ;
public Vector3 RandomnessInCameraAim;

public string TrainingLabels;
public string TrainingLabelsPath;


public float CameraStepSize;
public float CameraHeight;
Vector4 MovementBoundingBox;
public float CameraLimitMinimumX;
public float CameraLimitMaximumX;
public float CameraLimitMinimumZ;
public float CameraLimitMaximumZ;


void Start () {
TrainingLabels = "";
ExamplesGenerated = 0.0f;
VirtualShapeCrafter = GameObject.Find ("VirtualShapeCrafter");
VirtualShapeCrafterCamera = VirtualShapeCrafter.
                                GetComponent<Camera> ();
MovementBoundingBox = new Vector4 ( CameraLimitMinimumX,
                                    CameraLimitMinimumZ,
                                    CameraLimitMaximumX,
                                    CameraLimitMaximumZ);
SaveData = false;
GameObject[] Objs =
        GameObject.FindGameObjectsWithTag ("HasNotBeenScanned");

for (int i = 0; i < Objs.Length; i++) {
ObjectsNotScanned.Add (Objs[i]);
}

}

public static Quaternion QuaternionFromMatrix(Matrix4x4 m){
return Quaternion.LookRotation (m.GetColumn (2), m.GetColumn (1));
}

// Main Loop for data collection
public void CollectData () {

switch (DataCollectionState) {
case "ScanEntireScene":

Vector3 CurrentShapecrafterPosition = VirtualShapeCrafter.
                                            transform.
                                            position;
Vector3 DistanceToMove = new Vector3 (0.0f, 0.0f, 0.0f);

if (CurrentShapecrafterPosition[0]<MovementBoundingBox[2])
{
DistanceToMove =
        new Vector3 (CameraStepSize, 0.0f, 0.0f);
} else {
MoveCameraToPosition (
        new Vector3(CameraLimitMinimumX,
                    CameraHeight,
                    CurrentShapecrafterPosition[2]
                    -CameraStepSize));
}
ExamplesGenerated += 1;
MoveCamera (DistanceToMove);
SaveData = true;
break;
case "ScanEachObject":
if (ObjectsNotScanned.Count == 0) {
Debug.Log ("Done scanning");
SaveData = false;
DataCollectionState = "DoNotScan";
} else {
VirtualShapeCrafter.transform.position =
        ObjectsNotScanned [0].transform.
            position + new Vector3(
                        0.0f,
                        DistanceToFocalPoint,
                        0.0f);

VirtualShapeCrafter.transform.
        LookAt (ObjectsNotScanned [0].transform);
VirtualShapeCrafter.transform.
        RotateAround (ObjectsNotScanned [0].
            transform.position,
            Vector3.right,
            Random.Range(RotationLimitsTilt[0],
                RotationLimitsTilt[1]));
VirtualShapeCrafter.transform.
        RotateAround (ObjectsNotScanned [0].
            transform.position,
            Vector3.forward,
            Random.Range(RotationLimitsPan[0],
            RotationLimitsPan[1]));
VirtualShapeCrafter.transform.position +=
    new Vector3(Random.Range(
        -RandomnessInCameraAim[0],
        RandomnessInCameraAim[0]),
        Random.Range(-RandomnessInCameraAim[1],
        RandomnessInCameraAim[1]),
        Random.Range(-RandomnessInCameraAim[2],
        RandomnessInCameraAim[2]));

string ObjectName = ObjectsNotScanned [0].
        GetComponent<ObjectToPick>().name;
string BestGripVector;
string PositionCamCoords;
if (ObjectsNotScanned [0].
    GetComponentInChildren<GripVectors> () != null) {

try {
// World rotation from vector object
Matrix4x4 Transform_World_Grip = ObjectsNotScanned [0].transform.
            FindChild ("Gripvectors").GetChild(0).
            transform.localToWorldMatrix;


// World position from grip point object
Vector3 WorldPos = ObjectsNotScanned [0].transform.FindChild ("GripPoint").position;
Vector4 GripPointPosition = new Vector4 (WorldPos [0], WorldPos [1], WorldPos [2], 1);
Transform_World_Grip.SetColumn (3, GripPointPosition);

Matrix4x4 Transform_Cam_Grip = VirtualShapeCrafter.transform.
        worldToLocalMatrix*Transform_World_Grip;

Vector3 pos = new Vector3(Transform_Cam_Grip.GetColumn(3)[0],
            Transform_Cam_Grip.GetColumn(3)[1],
            Transform_Cam_Grip.GetColumn(3)[2]);

Vector3 bgv_up = ObjectsNotScanned [0].transform.FindChild ("Gripvectors").
        GetChild(0).transform.up;
Vector4 bgv_up_4 = new Vector4(bgv_up[0], bgv_up[1], bgv_up[2], 0.0f);
bgv_up = VirtualShapeCrafter.transform.worldToLocalMatrix*bgv_up_4;

Vector3 bgv_forward = ObjectsNotScanned [0].transform.FindChild ("Gripvectors").
    GetChild(0).transform.forward;
Vector4 bgv_forward_4 = new Vector4(bgv_forward[0], bgv_forward[1], bgv_forward[2], 0.0f);
bgv_forward = VirtualShapeCrafter.transform.worldToLocalMatrix*bgv_forward_4;

BestGripVector = bgv_up[0].ToString () + "," + bgv_up[1].ToString () +
    "," + bgv_up[2].ToString () + "," + bgv_forward[0].ToString () +
    "," + bgv_forward[1].ToString () + "," + bgv_forward[2].ToString ();

PositionCamCoords = pos[0].ToString()+ "," + pos[1].ToString () + "," + pos[2].ToString ();
Debug.Log ("Euler: " +BestGripVector);
} catch {
BestGripVector = "nan,nan,nan";
PositionCamCoords = "nan,nan,nan";
}

} else {
BestGripVector = "nan,nan,nan";
PositionCamCoords = "nan,nan,nan";
}

TrainingLabels += ObjectName + "," + PositionCamCoords + "," + BestGripVector + "\r\n";
System.IO.File.WriteAllText (TrainingLabelsPath, TrainingLabels);

ObjectsNotScanned [0].tag = "Untagged";
Debug.Log ("Objects left: " + ObjectsNotScanned.Count);
ObjectsNotScanned.RemoveAt (0);

//Debug.Log ("Scanning each object");
SaveData = true;
}
break;
case "DoNotScan":
Debug.Log ("Not scanning objects");
SaveData = false;
break;
default:
Debug.Log ("Error in Data Collection State");
break;
}
}

void MoveCamera (Vector3 DistanceToMove){
VirtualShapeCrafter.transform.position += DistanceToMove;
}

void MoveCameraToPosition( Vector3 Position){
VirtualShapeCrafter.transform.position = Position;
}
}
```

B.1.5 Virtual depth camera object

```
using UnityEngine;
using System.Collections;
using System.IO;
using System.Xml.Serialization;

//[ExecuteInEditMode]
public class DepthCamera : MonoBehaviour {
public RenderTexture DepthTexture;
public RenderTexture GroundTruthTexture;
public Material mat;
public Camera DepthCam;
public Camera ColorCam;
public string image_dir;
public int ExampleNum;
public int WidthResolution;
public int HeightResolution;
public GameObject MainObject;

void Start () {
ExampleNum = 0;
DepthCam = GetComponent<Camera> ();
DepthCam.depthTextureMode = DepthTextureMode.Depth;
DepthCam.depth = 24;
ColorCam = GameObject.FindGameObjectWithTag ("GroundTruthCamera").GetComponent<Camera>();
InitColorCam (ColorCam, DepthCam);

}


void InitColorCam(Camera ColorCam , Camera DepthCam){
ColorCam.orthographicSize = DepthCam.orthographicSize;
ColorCam.farClipPlane = DepthCam.farClipPlane;
ColorCam.nearClipPlane = DepthCam.nearClipPlane;
ColorCam.aspect = DepthCam.aspect;

}


void SaveExamples(){
string DepthName = "training_example_" + ExampleNum + "_depth";
string TruthName = "training_example_" + ExampleNum + "_truth";
SaveDepthTexture (DepthTexture, DepthName);
SaveTruthTexture (GroundTruthTexture, TruthName);
ExampleNum += 1;

}


void SaveDepthTexture(RenderTexture rt, string file_name){
RenderTexture.active = rt;
Texture2D DepthImage = new Texture2D (WidthResolution, HeightResolution,
                                        TextureFormat.RGBAFloat, false);

//Texture2D DepthImage = new Texture2D (WidthResolution, HeightResolution,
                                        TextureFormat.RGB24, false);
DepthImage.ReadPixels (new Rect (0, 0, WidthResolution, HeightResolution), 0, 0);
DepthImage.Apply ();


byte[] to_save = DepthImage.EncodeToPNG ();
Object.Destroy (DepthImage);
string path = image_dir + "\\" + file_name +".png";
File.WriteAllBytes (path, to_save);
string my_path = image_dir + "\\" + file_name + ".txt";
SaveImageAsTxt(DepthImage, my_path);
}


void SaveTruthTexture(RenderTexture rt, string file_name){
RenderTexture.active = rt;
Texture2D DepthImage = new Texture2D (WidthResolution, HeightResolution,
                                        TextureFormat.RGB24, false);
DepthImage.ReadPixels (new Rect (0, 0, WidthResolution, HeightResolution), 0, 0);
DepthImage.Apply ();

byte[] to_save = DepthImage.EncodeToPNG ();
Object.Destroy (DepthImage);
string path = image_dir + "\\" + file_name +".png";
File.WriteAllBytes (path, to_save);
}


void SaveImageAsTxt(Texture2D image, string path){

string str = "";
for (int i = 0; i < image.height; i++) {
for (int j = 0; j < image.width; j++) {
str = str + (DepthCam.nearClipPlane+image.GetPixel(j,i)[0]*DepthCam.farClipPlane) + " ";
}
//str.Remove (str.Length - 2,2);
str = str + "\n";
}
System.IO.File.WriteAllText (path, str);

}


public void OnRenderImage(RenderTexture source, RenderTexture destination){
if (MainObject.GetComponent<DataCollector> ().SaveData) {
Graphics.Blit (source, destination, mat);
SaveExamples ();
}

}
}
```

B.1.6 Shader for the virtual depth camera

```
Shader "Custom/DepthShader"
{
SubShader{
Tags { "RenderType"="Opaque"}

Pass{
CGPROGRAM
#pragma vertex vert
#pragma fragment frag
#include "UnityCG.cginc"


// The depth texture for the virtual Shapecrafter
sampler2D _CameraDepthTexture;

struct v2f{
float4 pos : SV_POSITION;
float4 scrPos : TEXCOORD1;
};


v2f vert(appdata_base v) {
v2f o;
o.pos = mul(UNITY_MATRIX_MVP, v.vertex);
o.scrPos=ComputeScreenPos(o.pos);
return o;
}

float4 frag(v2f i) : COLOR {
float depthValue = tex2Dproj(_CameraDepthTexture,
                        UNITY_PROJ_COORD(i.scrPos)).r;

// Orthographic projection is used in all experiments,
// and for perspective projection this line can be
// changed to:

// float depthValue = Linear01Depth(tex2Dproj(
                        _CameraDepthTexture,
                        UNITY_PROJ_COORD(i.scrPos)).r);

float4 depth;

// Store depth values in all channels
depth.r = depthValue;
depth.g = depthValue;
depth.b = depthValue;
depth.a = 1;
return depth;
}
ENDCG
}
}
}
```

Appendix C

C.1 Python code for neural networks

C.1.1 Training the classification network

```
import lasagne
import theano
import theano.tensor as T
import numpy as np
from DataSetHandling import LabelNumberToClassVector
import pickle
import numpy as np
from DataSetHandling import subtractMeanPixel, unitVariancePixel
from lasagne.nonlinearities import leaky_rectify, softmax, tanh, linear, rectify
from lasagne_functions import *

np.set_printoptions(threshold=np.nan)
np.set_printoptions(suppress=True)

# Options
drive = 'F:\\'
training_path = '.pkl'
cv_path = '.pkl'
save_params_path = ''
save_logs_path = ''
save_number = 40
num_epochs = 300
mean_unit_mode = True


load_parameters = False
params_to_load = '.pkl'
batch_size = 128
noisiness = 1

data_set, data_set_cv, labels_num, labels_num_cv = load_data_cPickle(training_path, cv_path,
                                                              num_ex_training='all', num_ex_cv='all')

print 'Standard deviation in training set: ' + str(np.std(data_set))
print 'Standard deviation in cv set: ' + str(np.std(data_set_cv))



if mean_unit_mode==True:
    data_set = subtractMeanPixel(data_set)
    data_set_cv = subtractMeanPixel(data_set_cv)
    std_training_data = 192.691# np.std(data_set)

print 'Training variance: ' + str(np.var(data_set/np.std(data_set))) + ', CV variance: ' +  \
            str(np.var(data_set_cv/np.std(data_set)))

training_data = split_data_in_batches(data_set, labels_num, batch_size)


# create Theano variables for input and target minibatch
X = T.tensor4('X')
Y = T.ivector('y')
```

```
if mean_unit_mode==True:
    pre_process_L2 = lasagne.layers.InputLayer((None, 1, 100, 100), X)
    print 'Mean and unit variance mode ON...'
else:
    print 'Band pass filter mode...'
    f_size = 11
    pp_filter = np.reshape(gaussian_filter((f_size,f_size), 1) -
        gaussian_filter((f_size, f_size), 2),
        newshape=(1, 1, f_size, f_size)).astype('float32')

    pre_process_L1 = lasagne.layers.InputLayer((None, 1, 100, 100), X)
    pre_process_L2 = lasagne.layers.Conv2DLayer(pre_process_L1, 1, (11, 11),
        nonlinearity=linear, W=pp_filter)
    #pre_process_fn = theano.function([input_var],
        T.clip(lasagne.layers.get_output(pre_process_net), -8.0, 8.0))

network = lasagne.layers.Conv2DLayer(pre_process_L2, 16, (3, 3), nonlinearity=rectify,
        W=lasagne.init.HeUniform())
network = lasagne.layers.batch_norm(network)
network = lasagne.layers.MaxPool2DLayer(network, pool_size=(2,2))

network = lasagne.layers.Conv2DLayer(network, 16, (3, 3), nonlinearity=rectify,
        W=lasagne.init.HeUniform())
network = lasagne.layers.batch_norm(network)
network = lasagne.layers.MaxPool2DLayer(network, pool_size=(2,2))


network = lasagne.layers.Conv2DLayer(network, 32, (3, 3), nonlinearity=rectify,
        W=lasagne.init.HeUniform())
network = lasagne.layers.batch_norm(network)

network = lasagne.layers.Conv2DLayer(network, 32, (3, 3), nonlinearity=rectify,
        W=lasagne.init.HeUniform())
network = lasagne.layers.batch_norm(network)
network = lasagne.layers.MaxPool2DLayer(network, pool_size=(2,2))

network = lasagne.layers.dropout(network, 0.5)
network = lasagne.layers.DenseLayer(network, 1024, nonlinearity=rectify,
        W=lasagne.init.HeUniform())
network = lasagne.layers.batch_norm(network)

network = lasagne.layers.dropout(network, 0.5)
network = lasagne.layers.DenseLayer(network, 256, nonlinearity=rectify,
        W=lasagne.init.HeUniform())
network = lasagne.layers.batch_norm(network)

network = lasagne.layers.dropout(network, 0.5)
network = lasagne.layers.DenseLayer(network, 4, nonlinearity=softmax)


if load_parameters == True:
    loaded_parameters = load_network_parameters(params_to_load)
    lasagne.layers.set_all_param_values(network,loaded_parameters)

# Loss functions
reg_param = 1e-4
prediction = lasagne.layers.get_output(network)
loss = lasagne.objectives.categorical_crossentropy(prediction, Y)
loss = loss.mean() +  reg_param * lasagne.regularization.regularize_network_params(network,
        lasagne.regularization.l2)

prediction_cv = lasagne.layers.get_output(network, deterministic=True)
loss_cv = lasagne.objectives.categorical_crossentropy(prediction_cv, Y)
loss_cv = loss_cv.mean() + reg_param * lasagne.regularization.regularize_network_params(network,
        lasagne.regularization.l2)


# Theano update rules
params = lasagne.layers.get_all_params(network, trainable=True)
updates = lasagne.updates.nesterov_momentum(loss, params, learning_rate=0.005, momentum=0.9)
```

```
# Compiling functions
train_function = theano.function([X, Y], loss, updates=updates)
test_function = theano.function([X, Y], loss_cv)


# Compiling deterministic functions for testing
test_prediction = lasagne.layers.get_output(network, deterministic=True)
predict_functionn = theano.function([X], T.argmax(test_prediction, axis=1))


# Initialize logs to pickle
cost_log = np.zeros((num_epochs)); cost_log_cv = np.zeros((num_epochs))
acc_log = np.zeros((num_epochs)); acc_log_cv = np.zeros((num_epochs))

save_parameters(lasagne.layers.get_all_param_values(network), save_params_path)



for epoch in range(num_epochs):

    idx = np.random.permutation(len(labels_num))
    labels_num_n = labels_num[idx]
    data_set_n = data_set[idx,:,:,:] +
        (np.random.random(data_set.shape).astype('float32'))*noisiness
    if mean_unit_mode==True:
        data_set_n = data_set_n/std_training_data
    training_data = split_data_in_batches(data_set_n, labels_num_n, batch_size)
    #vectors = vectors[idx,:]
    #training_data = split_data_with_truth_ims_in_batches(data_set,
                                                          labels_num,
                                                          truth_ims,
                                                          batch_size)

    data_set_cv_n = data_set_cv +
            (np.random.random(data_set_cv.shape).astype('float32'))*noisiness
    if mean_unit_mode==True:
        data_set_cv_n = data_set_cv_n/std_training_data

    print 'Training variance: ' + str(np.var(data_set_n)) + ', CV variance: ' +
            str(np.var(data_set_cv_n))

    # Finding and printing confusion matrices
    cm, cm_p = confusion_matrix(predict_functionn(data_set_cv_n),labels_num_cv)
    print cm
    print cm_p*100
    loss = 0
    acc = 0
    for input_batch, target_batch in training_data:
        new_loss = train_function(input_batch, target_batch)
        print 'Batch loss: ' + str(new_loss)
        loss += new_loss
        acc += calculate_accuracy(predict_functionn(input_batch), target_batch)

    loss_cv = test_function(data_set_cv_n, labels_num_cv)

    acc = acc/float(len(training_data))
    acc_cv = calculate_accuracy(predict_functionn(data_set_cv_n), labels_num_cv)
    print 'Save_number ' + str(save_number)
    print("Epoch %d: Loss %g" % (epoch + 1, loss / len(training_data)) + ",
            Loss_cv %g" % (loss_cv)) + ' acc: ' + str(acc) + ' acc_cv ' + str(acc_cv)


    cost_log[epoch] = loss / len(training_data);    acc_log[epoch] = acc
    cost_log_cv[epoch] = loss_cv;                   acc_log_cv[epoch] = acc_cv

    save_parameters([cost_log, acc_log, cost_log_cv, acc_log_cv],save_logs_path)
    save_parameters(lasagne.layers.get_all_param_values(network), save_params_path)


# Printing predictions and true labels
print str(predict_functionn(data_set_cv_n))
print str(labels_num_cv)
```

C.1.2 Training the grip point/vector network

```
import lasagne
import theano
import theano.tensor as T
import numpy as np
from DataSetHandling import LabelNumberToClassVector
import pickle
import numpy as np
from DataSetHandling import subtractMeanPixel, unitVariancePixel
from lasagne.nonlinearities import leaky_rectify, rectify, softmax, tanh, linear
from lasagne_functions import *

np.set_printoptions(threshold=np.nan)
np.set_printoptions(suppress=True)

# <Options>
directory = 'F:\\\\Datasets\\Vectors\\'
training_object = 'Hammer'
test_number = 'X'
num_epochs = 1000
load_parameters = True
params_to_load = directory + 'params_grip_vector_Hammer.pkl'
batch_size = 128
noisiness = 1
train_point_estimator = False
training_path = directory + 'Pickled\\' + training_object + '_Hard.pkl'
cv_path = directory + 'Pickled\\' + training_object + '_Hard_cv.pkl'

# </Options>


save_name = 'grip_vector_for_' + training_object + '_test_' + test_number

if train_point_estimator:
    dot_loss = False
else:
    dot_loss = True

# Training set
complete_dataset, labels_num, complete_truth_ims, vectors =
        pickle.load(open(training_path, "rb"))
data_set = np.reshape(complete_dataset,
    (complete_dataset.shape[0], 1,
    complete_dataset.shape[1],
    complete_dataset.shape[2])).astype(dtype=theano.config.floatX)

# CV set
complete_dataset_cv, labels_num_cv, complete_truth_ims_cv, vectors_cv =
    pickle.load(open(cv_path, "rb"))
data_set_cv = np.reshape(complete_dataset_cv,
    (complete_dataset_cv.shape[0], 1,
        complete_dataset_cv.shape[1],
        complete_dataset_cv.shape[2])).astype(dtype=theano.config.floatX)

print 'Shape full set: ' + str(data_set.shape) + ', shape full cv: ' +
    str(data_set_cv.shape) + ', shape labels: ' + str(labels_num.shape)+
    ', shape labels_cv: ' + str(labels_num_cv.shape)

print 'Full training vectors shape: ' + str(vectors.shape) +
    ', CV vectors shape: ' + str(vectors_cv.shape)

if train_point_estimator:
    use_vector_values = [0,3]
    num_outputs = 3
else:
    use_vector_values = [3,9]
    num_outputs = 6

print data_set.shape
vectors = vectors[:,use_vector_values[0]:use_vector_values[1]]
vectors_cv = vectors_cv[:,use_vector_values[0]:use_vector_values[1]]
```

```
# Theano
X = T.tensor4('X')
Y = T.matrix('y')

network = lasagne.layers.InputLayer((None, 1, 100, 100), X)
network = lasagne.layers.batch_norm(network)
network = lasagne.layers.Conv2DLayer(network, 16, (3, 3), nonlinearity=rectify,
        W=lasagne.init.HeUniform())
network = lasagne.layers.batch_norm(network)
network = lasagne.layers.MaxPool2DLayer(network, pool_size=(2,2))

network = lasagne.layers.Conv2DLayer(network, 16, (3, 3), nonlinearity=rectify,
        W=lasagne.init.HeUniform())
network = lasagne.layers.batch_norm(network)
network = lasagne.layers.MaxPool2DLayer(network, pool_size=(2,2))

network = lasagne.layers.Conv2DLayer(network, 16, (3, 3), nonlinearity=rectify,
        W=lasagne.init.HeUniform())
network = lasagne.layers.batch_norm(network)
network = lasagne.layers.MaxPool2DLayer(network, pool_size=(2,2))

network = lasagne.layers.dropout(network, 0.5)
network = lasagne.layers.DenseLayer(network, 512, nonlinearity=rectify,
        W=lasagne.init.HeUniform())
network = lasagne.layers.batch_norm(network)

network = lasagne.layers.dropout(network, 0.5)
network = lasagne.layers.DenseLayer(network, 128, nonlinearity=rectify,
        W=lasagne.init.HeUniform())
network = lasagne.layers.batch_norm(network)
network = lasagne.layers.dropout(network, 0.5)
network = lasagne.layers.DenseLayer(network, num_outputs,
        nonlinearity=linear,W=lasagne.init.GlorotUniform())

# Option of loading pre-trained parameters
if load_parameters == True:
    loaded_parameters = load_network_parameters(params_to_load)
    lasagne.layers.set_all_param_values(network,loaded_parameters)

prediction = lasagne.layers.get_output(network)
prediction_deterministic = lasagne.layers.get_output(network, deterministic=True)

# Dot-loss function for grip vector estimation and L2 loss for grip point regression
if dot_loss:
    loss = T.mean(1-(T.batched_dot(prediction, Y)/T.sqrt(T.batched_dot(Y, Y) *
        T.batched_dot(prediction, prediction))))
    loss_deterministic = T.mean(1-(T.batched_dot(prediction_deterministic, Y)/
        T.sqrt(T.batched_dot(Y, Y) *
            T.batched_dot(prediction_deterministic, prediction_deterministic))))
else:
    loss = lasagne.objectives.squared_error(prediction, Y)
    loss_deterministic = lasagne.objectives.squared_error(prediction_deterministic, Y)

reg_value = 1e-4
loss = loss.mean() + reg_value * lasagne.regularization.regularize_network_params(network,
    lasagne.regularization.l2)
loss_deterministic = loss_deterministic.mean() + reg_value *
    lasagne.regularization.regularize_network_params(network,
        lasagne.regularization.l2)

# Update functions for symbolic differentiation with Theano
params = lasagne.layers.get_all_params(network, trainable=True)
updates = lasagne.updates.nesterov_momentum(loss, params, learning_rate=0.01, momentum=0.9)


# Compiling functions
training_function = theano.function([X, Y], loss, updates=updates)
testing_functionn = theano.function([X, Y], loss_deterministic)

test_prediction = lasagne.layers.get_output(network, deterministic=True)
prediction_function = theano.function([X], test_prediction)

# Initialize logs to pickle
```

```
cost_log = np.zeros((num_epochs)); cost_log_cv = np.zeros((num_epochs))
acc_log = np.zeros((num_epochs)); acc_log_cv = np.zeros((num_epochs))

# Training loop
for epoch in range(num_epochs):
    print save_name

    # Shuffle data
    idx = np.random.permutation(len(labels_num))
    labels_num_n = labels_num[idx]
    data_set_n = data_set[idx,:,:,:]
    vectors_n = vectors[idx,:]

    # Add noise
    data_set_n = data_set_n +
        (np.random.random(data_set_n.shape).astype('float32'))*noisiness

    # Split into training batches
    training_data = split_data_with_vectors_in_batches(data_set_n,
                                                       labels_num_n,
                                                       vectors_n,
                                                       batch_size)

    data_set_cv_n = data_set_cv +
            (np.random.random(data_set_cv.shape).astype('float32'))*noisiness

    loss = 0
    for input_batch, target_batch in training_data:
        input_batch_n = input_batch+
            (np.random.random(input_batch.shape).astype('float32'))*noisiness

        new_loss = training_function(input_batch_n, target_batch)
        print 'Batch loss: ' + str(new_loss)
        loss += new_loss

    loss_cv = testing_functionn(data_set_cv_n, vectors_cv)
    print("Epoch %d: Loss %g" % (epoch + 1, loss / len(training_data)) +
            ", Loss_cv %g" % (loss_cv))

    cost_log[epoch] = loss / len(training_data)
    cost_log_cv[epoch] = loss_cv

    # Save parameters
    save_parameters([cost_log, acc_log, cost_log_cv, acc_log_cv], directory +
        'log_lasagne_' + str(save_name) + '.pkl')
    save_parameters(lasagne.layers.get_all_param_values(network), directory +
        'params_' + str(save_name) + '.pkl')

    pred = prediction_function(data_set_cv_n)
    np.savetxt(directory +save_name+ '_predicted.txt', pred, delimiter=',')
    np.savetxt(directory +save_name+ '_true.txt', vectors_cv, delimiter=',')


# Saving predictions on the CV set
data_set_cv_n = data_set_cv + (np.random.random(data_set_cv.shape).astype('float32'))*noisiness
pred = prediction_function(data_set_cv_n)
np.savetxt(directory +save_name+ '_predicted.txt', pred, delimiter=',')
np.savetxt(directory +save_name+ '_true.txt', vectors_cv, delimiter=',')
print 'Saved'
```