HTBLuVA Wiener Neustadt
Höhere Lehranstalt für Informatik

DIPLOMARBEIT

Localisation via ML Methods

Ausgeführt im Schuljahr 2019/20 von:

Algorithm
Ida Hönigmann 5AHIF

System Engineering
Peter Kain 5AHIF

Betreuer:

MMag. Dr. Michael Stifter

Wiener Neustadt, am March 29, 2019/20

Abgabevermerk: Übernommen von:

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst und keine anderen als die im Literaturverzeichnis angegeben Quellen und Hilfsmittel verwendet habe. Insbesondere versichere ich, dass ich alle wörtlichen und sinngemäßen Übernahmen aus anderen Werken als solche kenntlich gemacht habe.

Wiener Neustadt am March 29, 2019/20

Verfasser / Verfasserinnen:

Ida HÖNIGMANN Peter KAIN

Contents

Acknowledgement

The authors are very thankful for the help during the development of this work.

They especially would like to thank MMag. Dr. Michael Stifter for reviewing this work and suggesting creative solutions to many problems encountered.

A thank you also goes to all peers helping with mental support and for sharing a work place, without ever arguing.

Lastly the authors would like to say thank you to family, friends and the cats Bambi, Louis and Fritzi for always listening.

Kurzfassung

Das Thema dieser Diplomarbeit ist die Distanzbestimmung mit Hilfe eines neuronalen Netzwerks. In vorherigen Arbeiten hatten die Autoren die Aufgabe, die Entfernung zu Objekten in der Umgebung einer Drohne zu erkennen. Dafür konnten sie eine Tiefenkamera verwenden. Allerdings ist es nicht möglich, die Tiefenkamera auf der Drohne zu befestigen. Die Idee dieser Arbeit ist es, die Tiefenkamera wegzulassen und nur die Kamera auf der Drohne zu verwenden. Um das zu verwirklichen simulieren die Autoren die menschlichen Augen, indem sie zwei Bilder vom selben Objekt nehmen, die aus leicht unterschiedlichen Positionen stammen, und daraus die Distanz bestimmen, ähnlich dem menschlichen Gehirn.

Um das neuronale Netzwerk zu trainieren verwendeten die Autoren Blender, ein Programm, das Modellierung von dreidimensionalen Szenen ermöglicht. So modellierten die Authoren Szenen, die unterschiedliche Objekte beinhalten. Mit Hilfe eines Python Skripts wurden dann wiederholt zwei Kameras in der Szene platziert und aus deren Sicht die zwei Bilder des Objektes gerendert. So konnten die Autoren ausreichend Testdaten in der limitierten Zeit generieren, um das neuronale Netzwerk zu trainieren.

Die Autoren verwenden TensorFlow, um das Neuronale Netzwerk zu verwirklichen. Allerdings entschieden sie sich aber auch dazu, eine Eigenimplementierung in C++ zu schreiben um die Vorteile von TensorFlow deutlich zu machen. Diese Eigenimplementierung konnte sich in einfachen Anwendungsfällen zwar gegen Tensorflow behaupten oder teilweise sogar TensorFlow übertreffen, aber für komplexe Aufgaben, wie diese Diplomarbeit, war sie nicht ausreichend entwickelt. Zusammengefasst ist Tensorflow besser getestet und funktioniert auch für komplexe Anwendungsfälle verlässlich gut.

Im Laufe dieser Arbeit wurden zwei Experimente durchgeführt, die das Generalisieren von neuronalen Netzwerken in diesem Anwendungsgebiet untersuchen. Das erste Experiment beschäftigt sich mit der Generalisierung zu unterschiedlichen Objekten. Das Zweite mit der Möglichkeit nach einem Lernvorgang mit ausschließlich computergenerierten Bildern, Bilder, die von einer realen Kamera stammen, zu verarbeiten.

Diese Diplomarbeit zeigt, dass es sehr schwer ist, Wirklichkeit mit computergenerierten Bildern nachzustellen. Die zurückgelieferten Distanzen für computergenerierte Bilder waren mit rund 85-prozentiger Genauigkeit akzeptabel, aber das neuronale Netzwerk, welches nur mit computergenerierten Bildern trainiert wurde, hatte Schwierigkeiten mit realen Bildern.

Abstract

This diploma thesis deals with distance estimation realised via a neural network. In previous work the authors were challenged to detect objects in the surrounding of a drone, which they realised with a depth camera. The problem of this approach was the difficulty of mounting the camera on the drone, which proved impossible. The general idea of this work is therefore to omit the depth camera and only use the pre-existing camera of the drone. The authors decide to simulate the human eyes by taking two pictures of the same object from slightly different positions. The distance to some object can then be determined from these two images, similarly a human brain.

In order to train the neural network the authors used Blender, a program which allows users to model three dimensional scenes. The authors used this functionality to model scenes containing multiple different objects. With the help of a Python script two cameras were repeatedly placed in the scene and images of the objects from two different points of view were created. This way the authors could generate enough training data to train the neural network in the limited time frame.

The authors used TensorFlow to implement the neural network. Additionally an implementation of a neural network was written in C++ from scratch, which shows the advantages of using TensorFlow. The neural network written by the authors proved successful and even outperformed TensorFlow in simple use cases, for more complex ones, as this diploma thesis, the C++ implementation was not developed enough. Summarized TensorFlow is well tested and works very reliably in complex use cases.

During this work the authors performed two experiments, testing if the neural network is able to generalise in the described use case. The first experiment deals with generalising to other objects. The second one explores the possibility of the neural network generalising after having been trained on computer-generated images only, to process images taken by a real camera.

The results of this diploma thesis are, that it is very difficult to represent reality with computer-generated images. The retrieved distances for computer-generated images, which are around 85% accurate, were acceptable. However, the neural network, trained exclusively with computer-generated images, had problems with real images.

Chapter 1

Introduction

Author: Ida Hönigmann

Robots are getting more and more mobile. While a few years ago their usage was mostly limited to aid factory automation, robots have found widespread adoption in a multitude of industries, such as self driving cars and autonomous delivery drones. A challenge frequently encountered is navigating in unknown environments, which either requires the robot to sense specific characteristics of its surroundings or to communicate with some external system.

The problem of navigation has been looked at from many different angles. One popular approach in mobile robotics is to use the GPS, an external positioning system. To determine the position of a robot with GPS, it has to establish communication with a minimum of four satellites. The exact position of each satellite and its current time is broadcast by the satellites. By measuring the time needed for the signal to reach the robot, the position can be calculated up to three meters accurately.

However, in some cases positioning a robot using external positioning methods is not possible. In the case of the GPS this can be due to obstacles interfering with the radio signals send by the satellites, for example occurring inside a building. In comparison this work focuses on a system that can navigate in outdoor as well as in indoor environments.

1.1 Goal

The goal of this diploma thesis is to implement a system which can localize a robot using no other sensors than a camera.

This limitation was purposely chosen as the result of this thesis will find application by future robotics students at the HTBLuVA (Technical Secondary College) Wiener Neustadt and many robot systems used for educational purposes are only poorly equipped with sensors that detect the robot's environment. One sensor used in educational robotics is the either already equipped, or easily mountable camera.

As part of this thesis the authors not only want to implement an easy to use API for future robotic students, but to also show the possibilities and advantages of machine learning in localisation.

In order to accomplish precise localisation in various different surroundings, the authors

plan on implementing a neural network. The neural network should take images, taken by the camera, as an input, and outputs the relative distance to any object shown in the images. By using machine learning the system should be less dependent on a specific situation or setup in comparison to different camera based localisation methods. For example the localisation should work on objects varying in size and shape, as well as in different situations of lighting.

1.2 Motivation

In July 2019 the two authors of this work participated in the aerial tournament at the Global Conference on Educational Robotics held in Norman, Oklahoma. One of the two main challenges encountered at this tournament was landing a drone next to some randomly placed object, which colour, shape and size was known in advance.

The second challenge the participants at the tournament faced was flying from one side of randomly placed cardboard boxes to the other. The cardboard boxes, representing a mountain, are placed in one of various configurations, one example can be seen in figure 1.1.

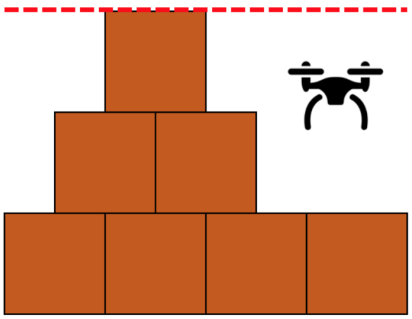


Figure 1.1: Seven cardboard boxes, representing a mountain, are placed in a random configuration. The team scores points if the drone passes the mountain while staying under the height limit indicated by the red dotted line.

The drones used at this aerial tournament have a front-facing camera mounted, while lacking any other sensor that can be used for detection of obstacles and game items. Therefore the participants needed to determine the distance to the object and the cardboard boxes using only the camera. At the Global Conference on Educational Robotics the authors of this work decided to detect the object based on its colour, but had to invest quite some time tweaking the values to get the localisation working correctly. Therefore the authors want to research and implement a method that is more robust than the colour based one.

1.3 Outlook and Perspective

The objective of this work is to create a system which uses machine learning methods in localising objects. After having trained the system, it should reliably return the x, y and z distances to an object, shown in two pictures taken from different angles. If this task turns

out to be too complex the system should be simplified by only returning one output number corresponding to distance from the drone to the object.

It is planned that the distance will be measured from the second camera position to the centre of object, as seen in Figure 1.2.

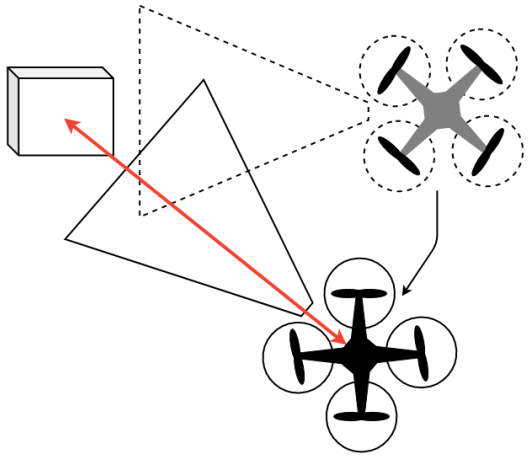


Figure 1.2: A drone tries to gather the data needed for the localisation of an object. After taking the first picture (grey drone with dotted lines) the drone flies to a second position (black drone with solid lines) to take another picture from a different point of view. The red line indicates the distance vector to be returned.

1.4 Equipment

Testing of the functionality of the localisation is to be performed on a Parrot Bebop 2. This drone was chosen as it is available to the authors during their time of research on this subject. This drone offers 1080p full HD video, which is believed be more than sufficient for this project.

The Parrot Bebop 2 weighs 500g. The drone was not designed to carry any additional weight, therefore mounting a stereo camera, a LIDAR sensor or any other method of localisation using additional hardware is not recommended.

A wide-angle 14 megapixel fish-eye lens is mounted as a camera on the Parrot Bebop 2[1]. It films 180 degrees vertically and horizontally and returns a 16:9 section which can be selected by specifying the vertical and horizontal angle of the virtual 16:9 camera. While the video resolution is limited to 1920 x 1080 pixels at 30 fps, the photo resolution is 4096 x 3072 pixels.

A picture of the Parrot Bebop 2 can be seen in Figure 1.3.

[1] **parrotBebop2**.

Figure 1.3: The drone chosen for testing of the localisation system is a Parrot Bebop 2. It consists of four propellers each attached to a motor, a camera in the front, a battery located on the rear, some processors and a plastic frame to hold everything together.

Though only tested on a Parrot Bebop 2 drone the distance estimation using this method generalises to various other robotic systems, as the drones will only be used to take images, that will be evaluated by a neural network.

Chapter 2

Study of Literature

Author:

2.1 Different Approaches to the Problem

Localising a robot in an unknown surrounding can be achieved by a multitude of different sensors. To name a few stereo cameras, LIDAR systems, ultrasonic sensors and time-of-flight cameras all measure the distance to the objects located around the robot. The data gathered by these sensors can then be used for localization and to aid navigation of the robot.

2.2 Stereo Camera

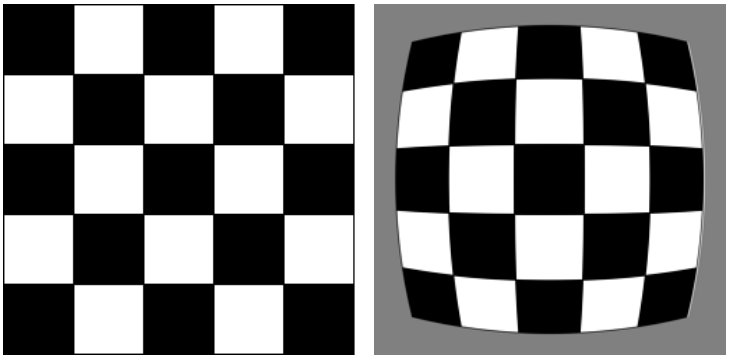
The challenge of sensing distances to various objects has been solved using stereo vision cameras. Computer stereo vision systems use two horizontally displaced cameras to take two images which then are both processed together to calculate depth information of the images. This process can be rather complicated as the distortions of the images have to be undone, before the two images can be projected onto a common plane, a disparity map can be created by comparing the two images and a 3d point cloud can be generated from this comparison[1]. In most robotics applications this point cloud is then filtered in search of some object, which distance was sought-after.

2.2.1 Distortion

One of the distortions that have to be undone before the images can be processed any further is called barrel distortion.

When the lens used by the camera has a lower magnification at the corners and the sides of the image than at the centre, barrel distortion can occur. This distortion can be visualized as seen in Figure 2.1.

[1] **Bradski˙Learning˙OpenCV**.



(a) Original image. (b) Distorted image.

Figure 2.1: The left shows the original image composed of straight horizontal and vertical lines. On the right image the effect of the barrel distortion can be seen. The distortion causes the lines to curve toward the outside of the image, causing the lines to appear in a barrel like shape.

Gribbon et.al.[2] propose equations, calculating the pixel values in the final image based on the pixel values in the original image.

A different distortion is called tangential distortion. It displaces points along the tangent of a circle placed centrally on the image as seen in Figure 2.2.

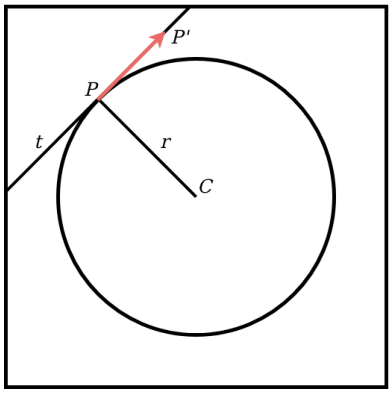


Figure 2.2: A point P is distorted along the tangent t of a circle placed at the middle of the image C with a radius r to a point P'. Distortions of this form are called tangential distortions.

The radius of the circle in Figure 2.2 depends on the point P. It can be calculated as the length between P and C. The length of the vector PP' is not uniform for all points and therefore depends on point P.

[2] **Gribbon˙Barrel˙Distortion˙Correction˙Algorithm**.

2.2.2 Image Rectification

Image Rectification projects multiple images taken from different points of view onto a common plane.

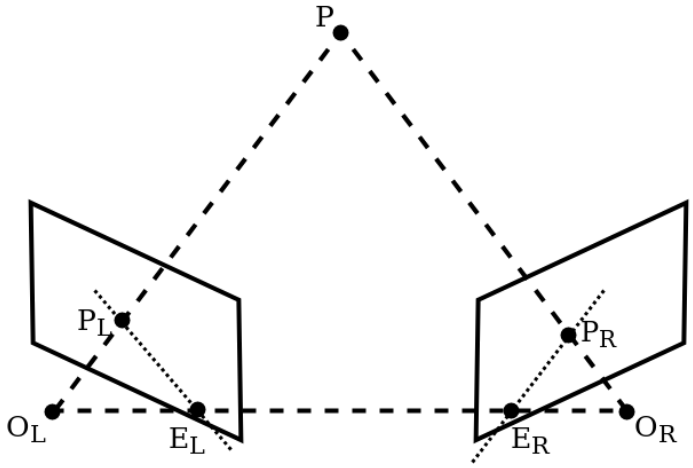


Figure 2.3: Two images containing some point P are taken from the two points O_L and O_R. Point P is projected in the image planes as points P_L and P_R. E_L and E_R depict the epipoles.

Chan et. al. propose an image rectification algorithm[3], which follows the following sequence of events:

1. At least seven matching points visible on both images are found.
2. The fundamental matrix (as well as the epipoles) are estimated.
3. The common region is identified (using epipolar geometry constraints).
4. The epipolar line is transferred and the Bresenham algorithm[4] is used to extract pixel values.
5. The rectified image is resampled.

2.2.3 Disparity Map

A disparity map represents the depth of an image. It can be calculated by comparing the position of pixels in two images made for example by a stereo camera. Objects, that are closer, have a larger difference in their relative position on the two image, than objects that are farther away.

One approach, which focuses not solely on the quality of the disparity, but additionally on the time needed for computation of the result, is described in the work of Mühlmann et.al[5].

[3] **Chen˙New˙Image˙Rectification˙Algorithm**.
[4] **Bresenham˙Linear˙Algorithm˙For˙Incremental˙Digital˙Display˙Of˙Circular˙Arcs**.
[5] **Muehlmann˙Calculating˙Dense˙Disparity˙Maps˙from˙Color˙Stereo˙Images**.

2.2.4 3D Point Cloud

To generate a 3D point cloud out of the disparity map a two dimensional image has to be converted into a three dimensional space. This is done by extracting the information about the depth for each pixel from the disparity map. In our use case the resulting 3d point cloud then has to be filtered in search of the object, which distance to was sought after.

2.3 LIDAR

A lidar is a sensor that is used to collect data about the environment. The sensor sends out light in a particular direction. Then the time needed for the light to return to the lidar is measured and the distance is calculated from that. By sending and receiving the light from multiple different directions, data points from the entire two or three dimensional surrounding can be detected.

One example of a use case for a lidar sensor is demonstrated in the paper on Lidar-based teach-and-repeat of mobile robot trajectores of Spunk et al.[6]. The objective of their work is to teach a robot some trajectory by demonstration and then have the robot repeat the same trajectory as closely as possible. Their use of a lidar sensor enables them to follow the given trajectory with an accuracy of a few millimetres.

2.4 Structure from Motion

Similar to a stereo camera structure from motion estimates the three dimensional structure of the surrounding from the information of two dimensional images. The difference lies in the number of images typically used. While stereo cameras use two images, structure from motion commonly takes more images to reconstruct the three dimensional scene.

Structure from Motion can be used whenever it is necessary to take multiple images. This can be the case when one image can not include all of the relevant data that should be processed, for example when modelling terrain.

[6] **Sprunk˙Lidar-based˙teach-and-repeat˙of˙mobile˙robot˙trajectories**.

2.5 Depth Maps from Structured Light

By taking an image of an specially illuminated scene the structured light approach is able to calculate a depth map. Scharstein et al. present an approach that constructs high accuracy depth maps without the need to calibrate the light sources.[7] To get information on specific distances to objects additional data, such as a different sensor measuring the distance to some (other) point in the image, is needed.

2.6 Human Depth Perception

In medicine depth perception is defined as the ability to see three dimensional space and judge distance accurately. Humans achieve depth perception by combining information from multiple depth cues. A depth cue is a feature of the environment or the person itself, that can be used to judge distance to objects. Depth cues can be classified in either of two categories: binocular and monocular, where binocular depth cues require two eyes and monocular only require one eye to work.

Which depth cue is used by humans in which situation was researched by Schrater et al.[8].

binocular depth cues	
retinal disparity	Because the two eyes of a human person are separated by a few centimetres each eye perceives the environment from a slightly different point of view. The vertical displacement of objects on the retina are used to determine distance and can be calculated as described by Cormack et al.[a].
convergence	To focus on objects farther away the eyes have to be less crossed than when focusing on near objects. It is debatable whether the information on how crossed the eyes look is used by all humans to gather information on depth. Richards et al. designed a test and found that approximately two thirds of the population can use convergence as a depth cue.[b]

Table 2.1: A short overview of different binocular depth cues used by humans to see three dimensional space and judge distance.

[a] **Cormack˙The˙computation˙of˙retinal˙disparity**.
[b] **Richards˙Convergence˙as˙a˙cue˙to˙depth**.

[7] **Scharstein˙High-accuracy˙stereo˙depth˙maps˙using˙structured˙light**.
[8] **Schrater˙How˙optimal˙depth˙cue˙integration˙depends˙on˙the˙task**.

monocular depth cues	
accommodation	As the eye focusses on an object the lens changes its thickness. The difference in thickness may be used to determine distance similar to convergence.
light and shadow	The position on which a shadow of an object falls can give information on its position in three dimensional space. This phenomenon can be observed in Fig. 2.4.
linear perspective	Parallel lines in the environment, such as railroad tracks, appear to converge in one point. Therefore when shown an image containing converging lines, such as Fig. 2.4, the two dimensional picture conveys depth. Yonas et al. performed an experiment, showing at what point in their life infants develop the ability to use linear perspective.[a]
texture gradients	According to Ganong[b] the retina takes up two thirds of the inner side of the eyeball. It is covered with cone cells, which are able to detect light. Since the retina only contains a finite number of cone cells, which are placed some distance apart, the eye can only detect its surrounding at a limited resolution. Therefore objects farther away seem smoother as fine details can not be detected as easily.
overlap	Overlapping is a depth cue humans develop early in their life. It is described as getting information on the order of objects sorted by their distance by looking at the way objects hide parts of other objects. Hagen tested sensitivity of overlapping as a depth cue in three, five and seven year old children and found that all where able to perceive depth based on this information.[c]

Table 2.2: A short overview of different monocular depth cues used by humans to see three dimensional space and judge distance.

[a] **Yonas˙Infants˙distance˙perception˙from˙linear˙perspective˙and˙texture˙gradients**.
[b] **Ganong˙Review˙of˙Medical˙Physiology**.
[c] **Hagen˙Development˙of˙ability˙to˙perceive˙and˙produce˙pictorial˙depth˙cue˙of˙overlapping**.

monocular depth cues	
aerial perspective	Very far distanced objects may appear hazy in certain weather conditions. The distance to these objects can be estimated by the haziness at which they appear.
relative motion	When the observer changes position the position at which objects are sensed change. The distance between the two positions as viewed by the observer are used to determine the how far the object is away. Closer objects move further while farther objects move less when the observer moves. This depth cue is called relative motion or motion parallax. According to Rogers et al. humans can reliably reconstruct distance from relative motion alone.[a].

Table 2.3: A short overview of different monocular depth cues used by humans to see three dimensional space and judge distance.

[a] **Rogers˙Motion˙parallax˙as˙an˙independent˙cue˙for˙depth˙perception**.

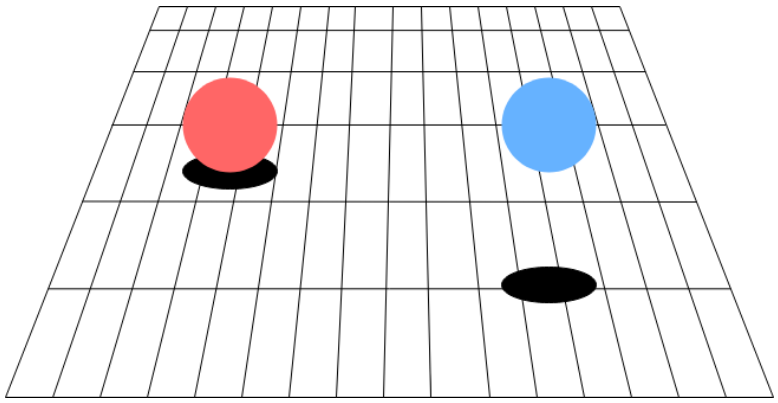


Figure 2.4: A two dimensional image conveying depth using linear perspective. The blue sphere seems closer than the red one, because of its shadow. Meanwhile the red sphere appears bigger as it is farther away and has the same relative size in the two dimensional image.

2.6.1 Depth Sensation

Since it is unclear if and to what extent animals and infants can determine depth and sense their three dimensional environment the term depth perception is not used. It has been shown however that at least some animals can sense depth and therefore experience depth sensation.

One experiment which tests depth sensation is called visual cliff.[9] The task of the test subjects is to identify a sharp drop over which they must move to reach a desired destination. The setup of the experiment can be found in Fig. 2.5. It has been shown that infants along with all tested animals stop as soon as they have reached the cliff. Therefore they must

[9] **Gibson˙The˙visual˙cliff**.

experience some sort of depth sensation.

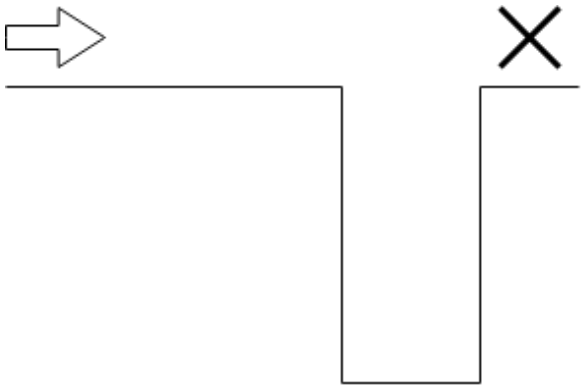


Figure 2.5: Setup of the visual cliff experiment. The arrow marks the starting point of the test subject. At the position of the X a desirable object is placed. Therefore the test subject starts moving toward the goal. If the subject has depth sensation it will stop before the cliff, while subjects without depth sensation won't be able to detect the cliff and therefore won't stop.

Chapter 3

Methodology

Author:

3.1 Challenges in the Use of Stereo Cameras

Since many drones used in educational robotics can only carry a limited amount of weight it is not possible to attach a stereo camera to such a robot. Instead the functionality of a stereo camera on such a drone system can be mimicked by taking the first image, flying to a second position, located horizontally next to the first one and taking the second image. This process in not as precise as a stereo camera, where the two lenses are always positioned at an exact interval from one another. Therefore the output of this system might not work as reliably. Additionally other factors, such as differences in the two images due to some time passing between the taking of the images influence and mostly worsen the accuracy.

Therefore the authors try to approach this challenge with the help of machine learning. Neural networks can be taught to work with different changes in the environment and still return results with superior quality as opposed to a conventional implementation.

3.2 Generating Data

Neural Networks require huge amounts of data to work reliably. Because of the author's limited time frame this test data will be generated with the help of Blender 2.8. Blender is a free program for designing and animating 3D objects, which also supports scripting with Python to add or remove objects from a scene. The authors will use this capability to generate the huge amounts of test data needed from the perspectives of the two cameras, which point to a specific object in the scene. This enables the authors to create enough images to train a neural network, since shooting the amount of pictures needed by hand would take too long to consider.

3.2.1 Blender

Besides 3D-modelling Blender enables the user to perform various different actions, such as laying out scenes, UV-Editing, shading, animating and rendering. Additionally scenes can be modified by executing Python scripts. Figure 3.1 shows the interface of Blender 2.8 with the default file loaded.

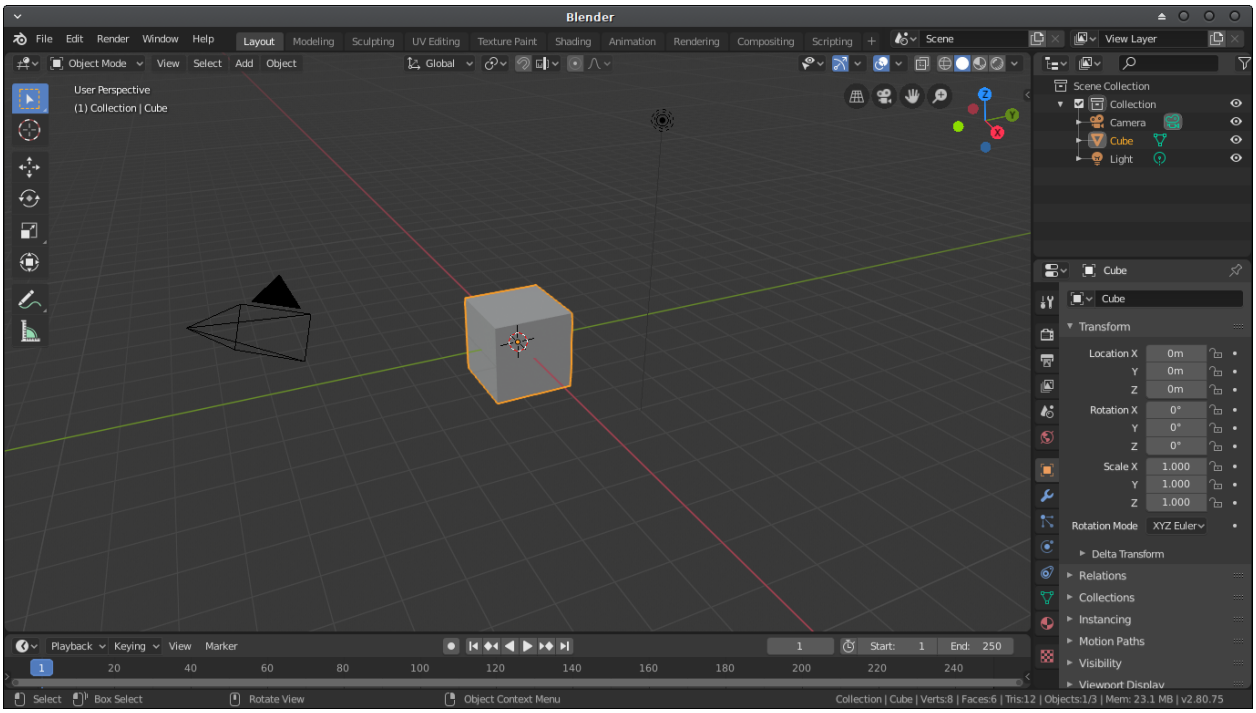


Figure 3.1: Blender interface after startup. The default scene features a grey cube, a camera (left of the cube) and a light source (black dot and circle positioned on the top right of the cube).

The authors decided to extensively use the scripting function in their work. One example of a Python script, that can be executed in Blender is the following code:

```python
import bpy

# Selects all cubes and deletes them
bpy.ops.object.select_all(action='DESELECT')
bpy.ops.object.select_by_type(type='MESH')
bpy.ops.object.delete()

# Adds a new cube
bpy.ops.mesh.primitive_cube_add(size=3, enter_editmode=False, location=(4, 2, 0))

# Adds a new material representing the colour red
bpy.ops.material.new()
material = bpy.data.materials[-1]
material.name = 'Red'
material.diffuse_color = (0.8, 0.1, 0.1, 1)

# Apply material onto the newly created cube object
bpy.context.active_object.data.materials.append(material)
```

This code first clears the scene from other meshes (running the script twice would otherwise place the new cube inside the old cube). Then a new mesh in form of a cube is added at the given location. Next colour is added to the new cube by applying a material. A material is a specification of how the surface of the object should look. Advanced materials can represent raw or reflective surfaces, however the authors decided to keep it simple. The chosen material represents a red surface (the red/green/blue/alpha channels ranging from 0.0 (for 0) to 1.0 (for 255) are specified). Lastly the created material is applied to the object. The resulting images is shown below:

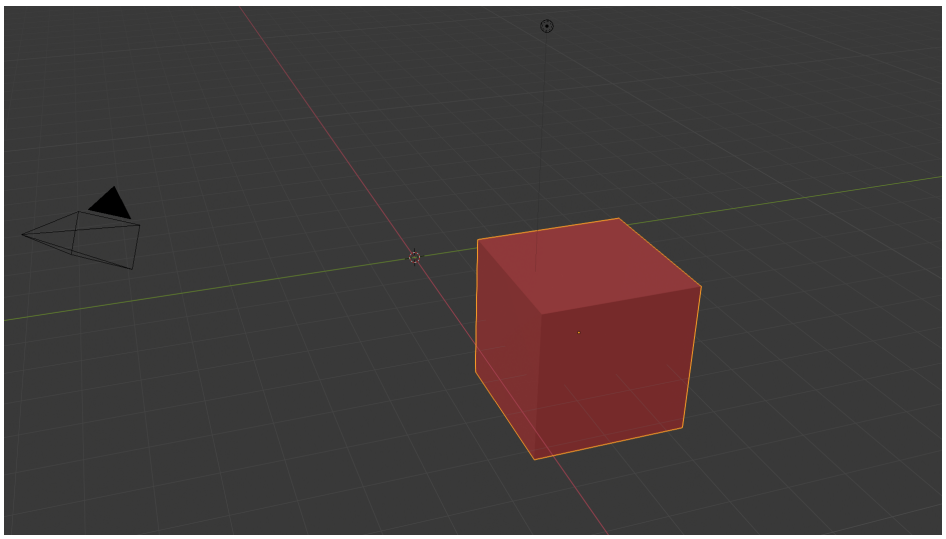


Figure 3.2: The result when running above code. After the default cube is deleted, a new one is added with a red material and translated via the 'location' argument.

Using similar Python scripts the authors will generate the image data necessary as well as perform the calculation of the distance which will be specified as the correct values to train the neural network on.

3.3 Image Preprocessing

After the test images have been rendered with the help of Blender some image preprocessing is required. For example the machine learning component of this project should take two images as an input. To simplify the input the two images will be placed next to each other to form a new image twice as wide as the original images. The concatenation of two example images is visualized in Figure 3.3. Other preprocessing measures that have to be taken are downscaling the image, as to not have too many weights in the input layer of the neural network.

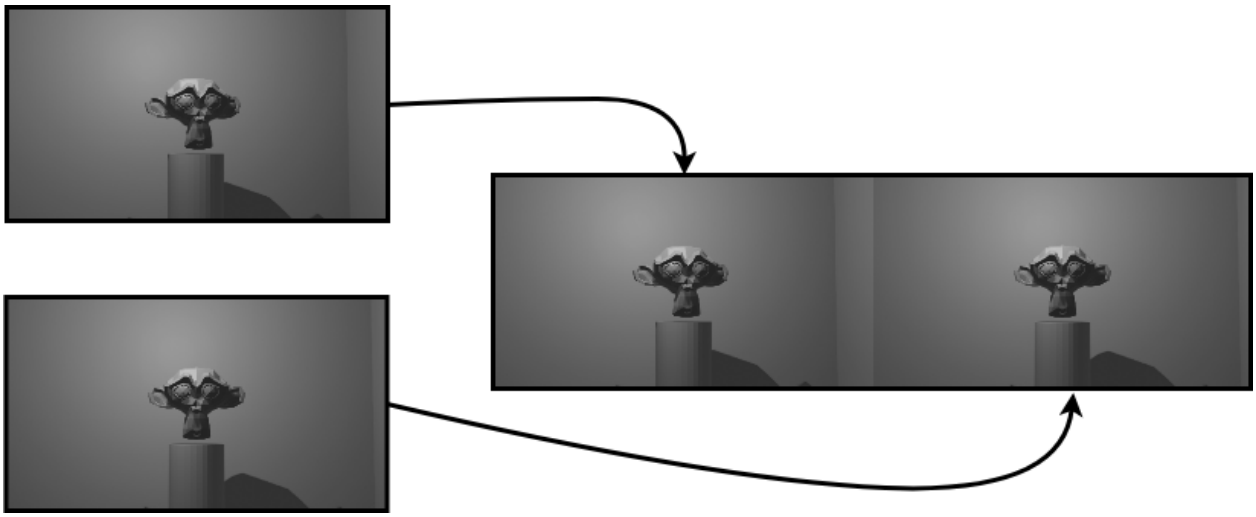


Figure 3.3: Two greyscale renders showing the same object from two different camera positions in Blender are merged into one picture which can be feed into the neural network component.

Additionally, the authors decided to experiment if manipulating these test images further would result in differences of distance perception by the neural network or if they would speed up or slow down the learning phase. These manipulations are done using OpenCV and Python3. OpenCV provides many image manipulation tools. For this project the authors chose the following methods of image manipulation:

Name	Description
Greyscale	A greyscale image is an image with only one value for the red, green and blue colour channels, resulting in different shades of grey instead of usual colours.
Resolution	Resolution refers to the count of pixels in each dimension (width and height).
Cropping	When cropping an image an unwanted part located at the peripheral areas of the image is removed.
Saturated	Saturated images feature stronger colours, which makes them easier to distinguish from another.
Brightness	Brightening images can make colours harder to distinguish from another. Additionally it can lead to the same problems encountered in overexposed images, such as part of the image being completely white and therefore not providing any information.

3.4 Neural Network

A neural network is a tool used in machine learning. It consists of nodes, each receiving some input values as well as some weights associated to each input value and outputs some output value. The calculation returning the output value is relatively simple. Many of these nodes form what is called a layer. By connecting the nodes of consecutive layers the neural network can perform more complex tasks. A vanilla neural network is formed of multiple layers, where the input of each node, except the input nodes, is calculated from all output values of all nodes in the previous layer. Figure 3.4 depicts a visualization of a vanilla neural network. By manipulating the weights associated to each input value the network can learn to solve a given task.

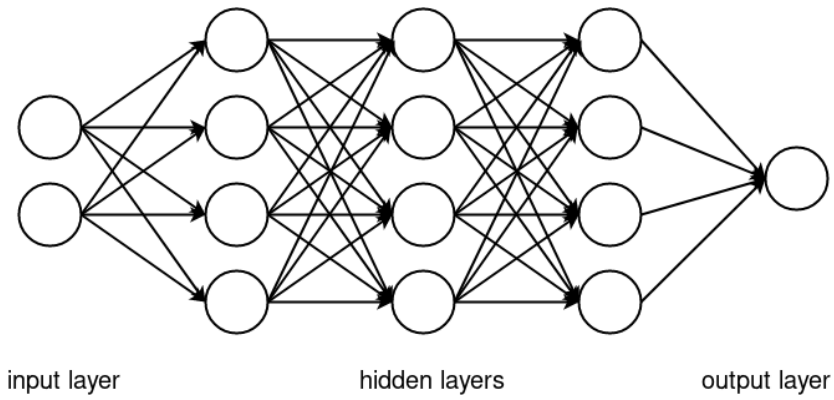


Figure 3.4: Depiction af a vanilla neural network. Each circle represents a node and the arrows show the flow of information. All nodes on the same level are collectively called a layer.

3.4.1 Convolutional Neural Network

Neural networks need huge amounts of data to optimize the values of all weights, in such a way that the neural network is able to evaluate unseen data with a reasonable accuracy. Convolutional neural networks simplify the optimization of the weights by stating that some weights are shared between multiple connections of nodes. This results in fewer weights having to be optimized. Additionally convolutional neural networks perform similar actions in multiple parts of the input data which especially makes sense when working with images, e.g. search for edges in all sections of an image.

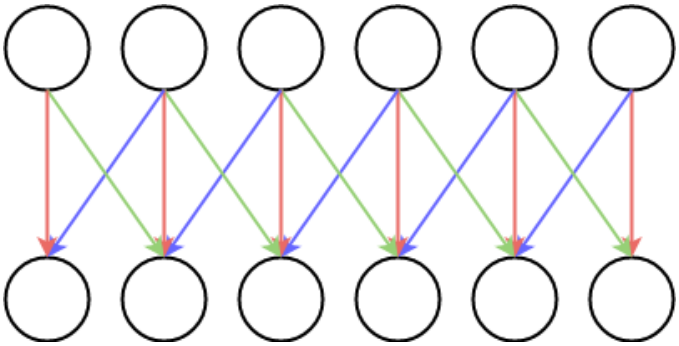


Figure 3.5: Segment of a convolutional neural network. In comparison to a vanilla neural network each node is only connected to its closest neighbours. Additionally all arrows connected to another node in the same relative position (same colour) have the same weight attached.

3.4.2 Loss Function

A loss function is used in optimization problems to determine the loss or cost of some operation. Therefore a low loss is desired. Some commonly used loss functions commonly are:

Mean Squared Error

$$mse = \frac{1}{n}\sum_{i=1}^{n}(x_i - \hat{x}_i)^2$$

where n is the number of elements, x is the correct value and $\hat{x}$ is the estimated value. Squaring the difference of the correct and estimated value is necessary, as negative and positive differences could otherwise annul each other.

Mean Absolute Error

$$mae = \frac{1}{n}\sum_{i=1}^{n}|\hat{x}_i - x_i|$$

where n is the number of elements, x is the correct value and $\hat{x}$ is the estimated value. In comparison to the mean squared error, this method uses the absolute value of the difference.

Mean Absolute Percentage Error

$$mape = \frac{100\%}{n} \sum_{i=1}^{n} \left| \frac{x_i - \hat{x}_i}{x_i} \right|$$

where n is the number of elements, x is the correct value and $\hat{x}$ is the estimated value. The mean absolute percentage error returns the average percentage by which the estimated value differs from the actual one.

Mean Squared Logarithmic Error

$$msle = \frac{1}{n} \sum_{i=1}^{n} log(\hat{x}_i + 1) - log(x_i + 1)$$

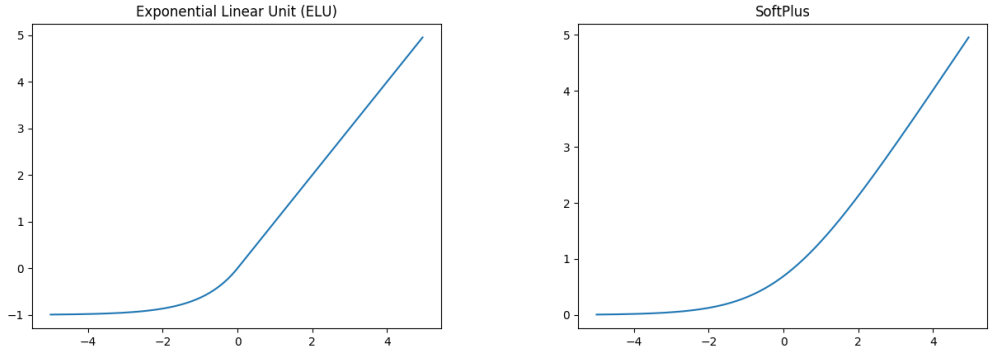
where n is the number of elements, x is the correct value and $\hat{x}$ is the estimated value.

Logcosh

Kullback Leibler Divergence

3.4.3 Activation Function

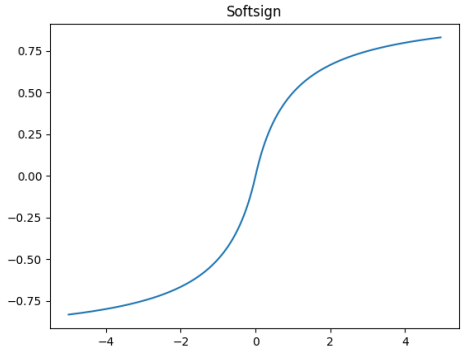
The activation function defines the output of a node based on the input values it receives. Some of the most common activation functions are displayed in Fig. 3.6 and Fig. 3.7.



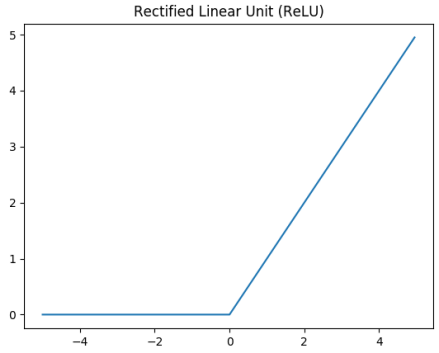
(a) The Exponential Linear Unit function is defined as $f(\alpha, x) = \begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$.

(b) The SoftPlus function is is defined as $f(x) = ln(1 + e^x)$.

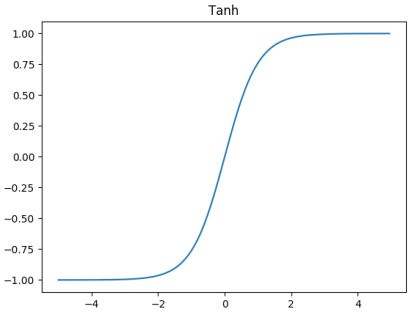
Figure 3.6: An excerpt of the most common activation functions used in machine learning. The rest can be found in Figure 3.7. Non-linear functions allow the neural network to solve more complex tasks. All activation functions are plotted from -4 to 4 as the most interesting features of these functions often occur around 0.



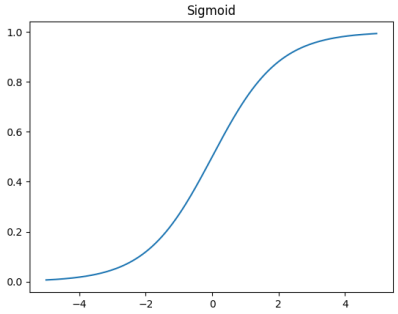
(a) Softsign is defined as $f(x) = \frac{x}{1+|x|}$.



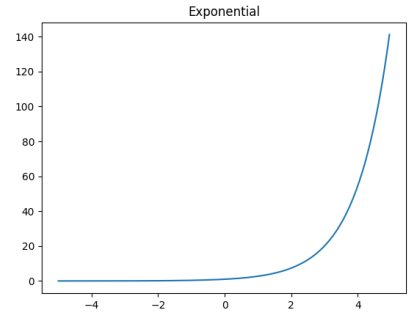
(b) Rectified Linear Unit can be described as $f(x) = \begin{cases} x & x \geq 0 \\ 0 & x < 0 \end{cases}$



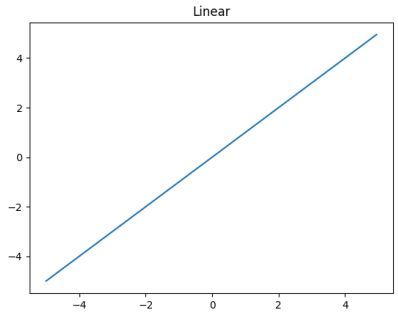
(c) The Tanh activation function is defined as $f(x) = tanh(x)$.



(d) The Sigmoid function is defines as $f(x) = \frac{1}{1+e^{-x}}$.



(e) The Exponential activation function is defined as $f(x) = e^x$.

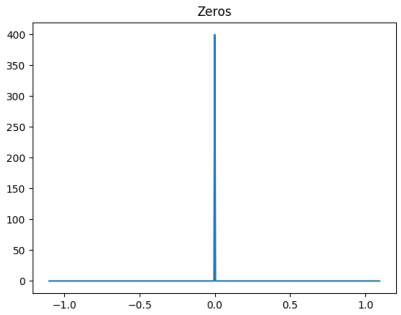


(f) The Linear activation function, also called identity function, is defined as $f(x) = x$.

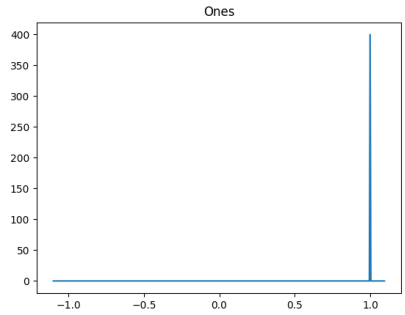
Figure 3.7: Other common activation functions used in machine learning. Once again all activation functions are plotted from -4 to 4 for the same reason as in Fig. 3.6.

3.4.4 Initializer

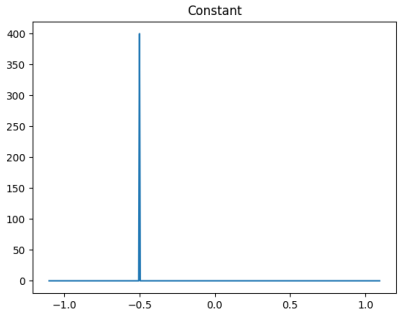
In neural networks the initial weights of all layers have to be set before training can begin. These weights can either be set to some constant value or initialized by random values from some distribution. Some basic initializer distributions are plotted in Fig 3.8 and Fig 3.9.



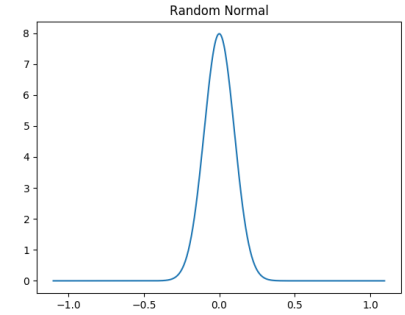
(a) All output values of this function are zero.



(b) All output values of this function are one.

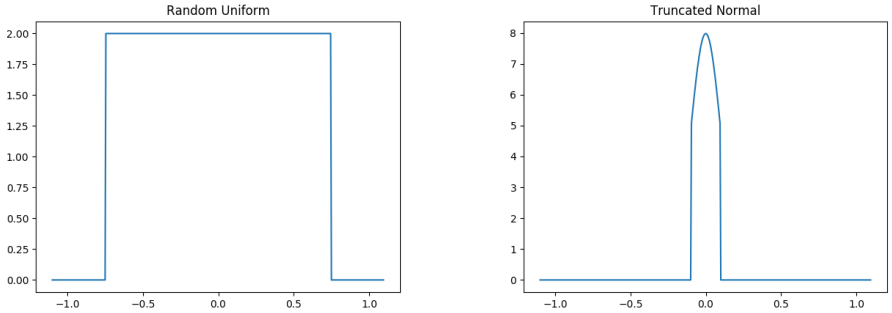


(c) All output values of this function are one constant value. In this example 0.5 was chosen as the output value.



(d) A random normal distribution where the mean is set to 0 and the standard deviation is set to 0.05. Values closer to the mean value are more likely.

Figure 3.8: These are some of the most basic distribution most initializer functions are based on.



(a) In a random uniform function all values between the set minimum and maximum value are equally likely.

(b) A distribution based on the normal distribution, where all values outside of two standard deviations away from the mean are redrawn.

Figure 3.9: More basic distribution initializer functions of weights and biases are often based on.

3.4.5 Optimizer

The objective of an optimizer is to optimize the loss function (find a local minimum). Some of the most commonly used optimizers are:

- Stochastic gradient descent (SGD)
- Root Mean Square Propagation (RMSProp)
- Adaptive Gradient Algorithm (Adagrad)
- Adadelta
- Adaptive Moment Estimation (Adam)
- Adamax
- Nesterov Adam (Nadam)

More information on the most commonly used optimizer SGD can be found in "Pattern Recognition and Machine Learning" by Christopfer M. Hishop[1].

3.4.6 Overfitting and Underfitting

Overfitting occurs if the data sample is replicated too closely. One example of overfitting can be seen in Figure 3.10. The function used is a high degree polynomial function, which has a very good fit on the noisy data originating from a linear correlation, but when testing this model on other data points, especially far from zero on the x-axis it fails to represent the original relation.

[1] **Bishop˙Pattern˙Recognition˙and˙Machine˙Learning**.

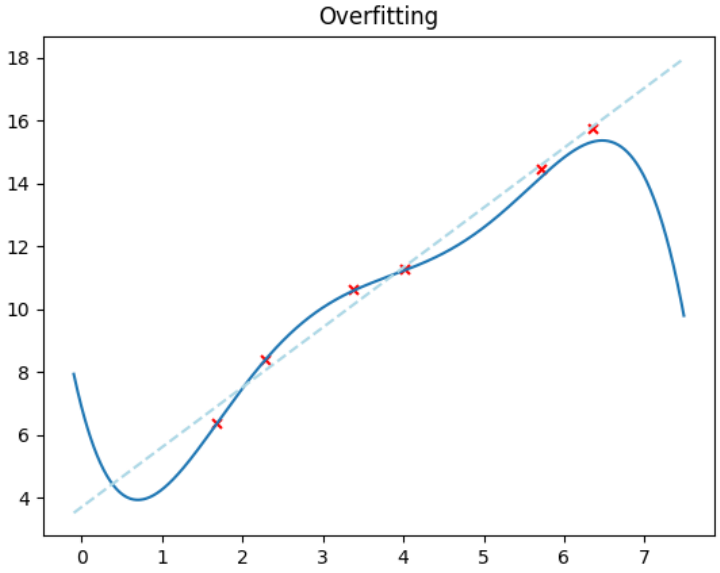


Figure 3.10: Noisy data points originating in a linear correlation (red crosses) are fitted using a high degree polynomial function (blue line), resulting in overfitting. An linear fit (light blue dashed line) has a worse fit on the given data sample, but gives better results when extrapolating.

The goal is to find a function, that has just enough complexity to fit the data points. The opposite of overfitting is called underfitting. When underfitting a more complex problem is approximated using too simple methods, again resulting in poor results.

3.5 Neural Network Implementations

The authors decided to use a software framework called TensorFlow for the first implementation of the neural network. This has the following two advantages: using Tensorflow allows for a low effort proof of concept and it makes testing out different configurations (e.g. number of hidden layers or filters in image preprocessing) of the neural network easier.

After it has been shown that the challenge of detecting the distance to an object can be solved using machine learning, the authors plan on implementing a neural network in C++ on their own. The knowledge gained in the TensorFlow implementation will be used in the C++ implementation, which hopefully will make the work less time consuming.

3.6 TensorFlow

As machine learning has gained popularity in recent years the demand for applicable frameworks grew. One of the most popular is called TensorFlow. It was developed by Google for internal use and was published under the Apache License 2.0 on the 9$^{\text{th}}$ of November 2015.

TensorFlow supports APIs for Python, C, C++, Go, Java, JavaScript and Swift. Due to its popularity third party APIs for C#, R, Scala, Rust and many more were developed.

Its use cases reach from categorizing handwritten digits to YouTube video recommendations, one of the many applications Google use it for.

Tom Hope et al. describe TensorFlow as a software framework for numerical computations based on dataflow graphs[2].

3.6.1 Computation graph

To compute a value using TensorFlow a computation graph has to be constructed. In this graph each node corresponds to an operation, such as subtraction or division. By connecting these nodes via edges the output of one node can be fed as input into another node. One example of such a computation graph can be seen in Figure 3.11.

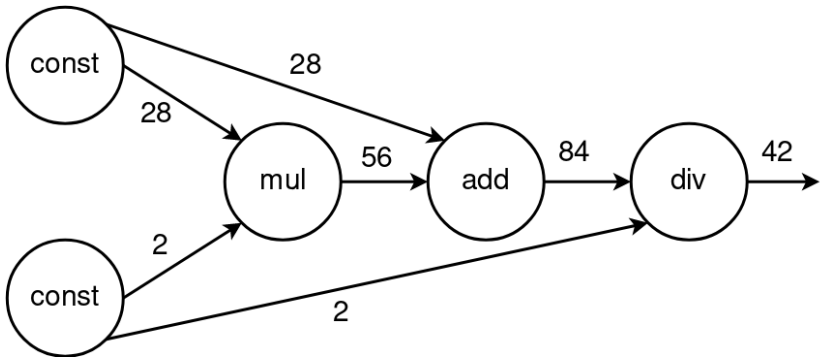


Figure 3.11: Each node represents an operation, where *const* stands for a constant value, *add* for addition, *mul* for multiplication and *div* for division. Edges, represented by arrows, connect nodes. The information shared between the nodes is described by the numbers written next to the edges. This computational graph calculates the result of the arithmetic expression $(28 * 2) + 28/2$.

The implementation of the computation graph, shown in Figure 3.11, in Python could look as follows:

[2] **Hope˙Learning˙TensorFlow**.

```
1  import tensorflow as tf
2
3  a = tf.constant(28)
4  b = tf.constant(2)
5
6  c = tf.multiply(a, b)
7  d = tf.add(a, c)
8  e = tf.divide(d, b)
9
10 with tf.Session() as sess:
11     out = sess.run(e)
12
13 print(out)
```

The first line specifies that the TensorFlow functionality should be imported. Line 3 and 4 define the two constant values and assigns them the values 28 and 2 respectively. In line 6 to 8 the other nodes of the graph are specified. E.g. in line 6 a new node, named c, is created and the output of node a and node b are connected as its input. To perform the calculation described by the graph a new session is created in line 10. Finally the output of the graph (node e) is specified in line 11, the result is calculated and printed in line 13.

TensorFlow allows for another way of specifying a graph with these arithmetic operations:

```
1  import tensorflow as tf
2
3  a = tf.constant(28)
4  b = tf.constant(2)
5
6  e = (a * b + a) / b
7
8  with tf.Session() as sess:
9      out = sess.run(e)
10
11 print(out)
```

This code is equivalent to the first one, but uses syntactic sugar to shorten line 6 to 8 in the first code block into line 6. At this point it should be noted that while it might look like it line 6 does not calculate anything. It simply describes how the computational graph should look. The answer (42) is calculated in the session in line 9.

Using the same principle, but different functions TensorFlow allows the creation of complex neural networks in a simple fashion.

3.6.2 Alternatives to Tensorflow

TensorFlow offers some abstraction libraries, such as Keras. As some code snippets are used very frequently and with only slight variation when implementing a Neural Network, Keras offers these code blocks as predefined functions, making development easier for "standard" use cases.

For example Keras provides several different layers, that can be used to put together a neural network. Table 3.1 describes just some of the available layers.

Name	**Description**
Dense	Regular densely connected layer
Activation	An activation function is applied to the output
Flatten	Reduces the shape (e.g. flattens the input to be one-dimensional)
Conv1D/2D	Convolution layer, as described in Section 3.4.1
MaxPooling1D/2D	Resizes the tensor by calculating the max of each group
AveragePooling1D/2D	Resizes the tensor by calculating the average of each group
LocallyConnected1D/2D	A node is only connected to few other nodes (with unshared weights)
SimpleRNN	Fully-connected RNN where the output is to be fed back to input
LSTM	Long short-term memory layer

Table 3.1: Short description of layers available in Keras, that can be used to assemble a neural network, with little effort. A more detailed description of these layers can be found in the keras documentation[a]

[a] **kerasDocumentation**.

Another feature Keras offers is image preprocessing. In comparison to the authors use of this phrase, Keras offers transformations of the image that alter the image only slightly, which prevents overfitting, when using only a small input data set of images. Examples of the transformations offered are: rotating the image, shifting the image along the vertical or horizontal axis, changing the brightness of the image, zooming, flipping the image vertically or horizontally and rescaling the image.

Other available machine learning frameworks are:

- PyTorch
- Caffe
- Microsoft Cognitive toolkit (CNTK)
- Caffe2
- MXNet

- Torch
- Theano
- DeepLearning4j

These frameworks were not chosen as the authors already had a basic understanding of Tensorflow and Keras. Additionally Keras and Tensorflow are open source (available on GitHub at `https://github.com/tensorflow/tensorflow` and `https://github.com/keras-team/keras`), widely used and free of charge.

Chapter 4

Implementation

Author:

4.1 Generating test data

Good test data is of utmost importance in machine learning. The system can only know information that is depicted in the training data, which is why it is important to include as many aspects of the problem as possible in this data.

Since machine learning needs a lot of data in order to solve the given task it can be tiresome to generate and label all this data by hand. Therefore the authors decided to simulate the objects and the camera using a computer graphics modelling software called Blender.

Blender allows for relatively easy generation of training data by providing a Python API. With this API almost anything that can be done using the blender user interface can also be done using Python.

For generating data the first step for the authors was to model a scene, as seen in Figure 4.1. This scene would contain a building split into multiple rooms. Each of the rooms is home to one object, which was rendered using two cameras, representing two points of view. The first camera is positioned relative to the object, whereas the second camera is positioned relative to the first camera, guaranteeing a different view on the object. Lastly, the two renders formed one input for the neural network.

For the test objects the authors used two objects: a monkey head, which already exists in Blender and a self-modelled object, representing a vase with some decorating sticks in it. Both objects can be seen in Figure 4.2.

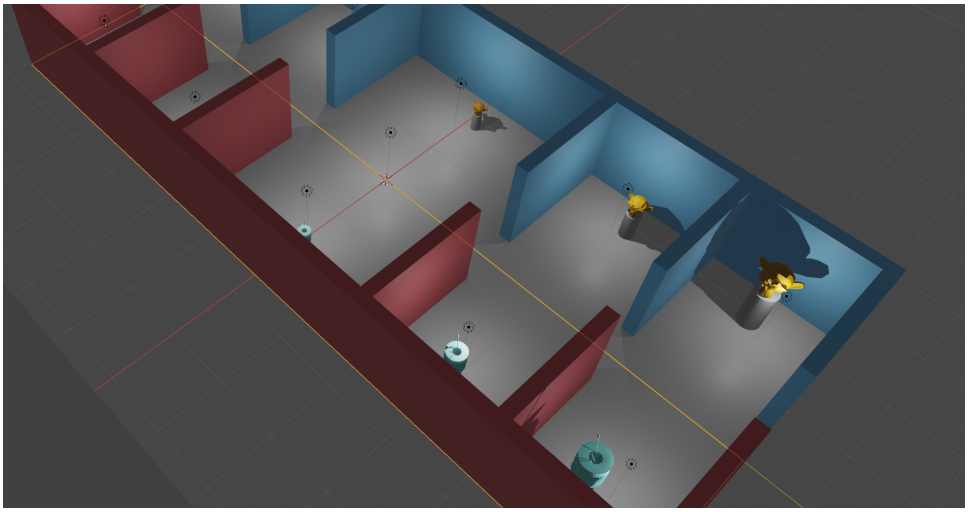
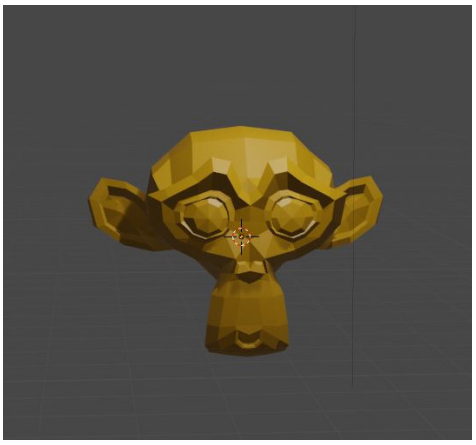
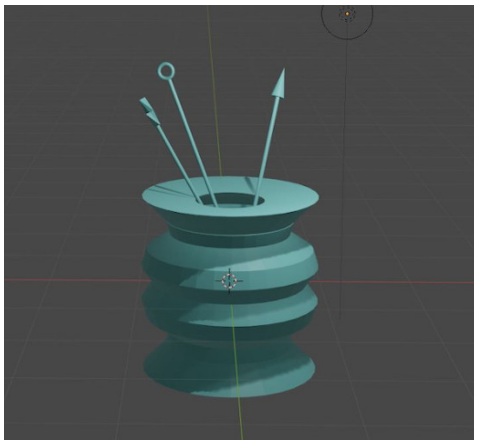


Figure 4.1: The test scene for generating data. Each object is placed in a room and is separated by walls.



(a) One of the test objects - a monkey head.



(b) One of the test objects - a decorative vase.

Figure 4.2: The two test objects as displayed in the Blender interface.

4.1.1 Generating the Render-Pairs

The following code generates the two needed renders for one of the test objects:

```
# Generate for 1 object
def generate_1(renderpath, infopath, target_name, iterations):
target_loc = bpy.data.objects[target_name].location

XMIN = target_loc.x - 0.6 if target_loc.x > 0 else target_loc.x + 0.6
XMAX = target_loc.x - 8.6 if target_loc.x > 0 else target_loc.x + 8.6
YMIN = target_loc.y - 1.85
YMAX = target_loc.y + 1.85
ZMIN = 0.75
ZMAX = 2

RENDERPATH = renderpath + target_name
INFOPATH = infopath + target_name

RENDERFILEPATH = RENDERPATH + "/{}-{}.jpg"
INFOFILEPATH = INFOPATH + "/{}-{}.txt"

make_dirs(RENDERPATH, INFOPATH)

for i in range(0, iterations):
# Calculate values
X = random.uniform(XMIN, XMAX)
Y1 = random.uniform(YMIN, YMAX)
Y2 = Y1 + 1 if (Y1 - target_loc.y) < 0 else Y1 - 1
Z = random.uniform(ZMIN, ZMAX)

# Add first camera
add_camera(X, Y1, Z, target_name)
write_dist_info(INFOFILEPATH, target_name, i, 1)
render(RENDERFILEPATH, i, 1)
del_cameras()

# Add second camera
add_camera(X, Y2, Z, target_name)
write_dist_info(INFOFILEPATH, target_name, i, 2)
render(RENDERFILEPATH, i, 2)
del_cameras()
```

This code can seem overwhelming at first, so the authors decided to explain each part of it.

The first code block

```
target_loc = bpy.data.objects[target_name].location

XMIN = target_loc.x - 0.6 if target_loc.x > 0 else target_loc.x + 0.6
XMAX = target_loc.x - 8.6 if target_loc.x > 0 else target_loc.x + 8.6
YMIN = target_loc.y - 1.85
YMAX = target_loc.y + 1.85
ZMIN = 0.75
ZMAX = 2
```

stores the location of the given target, or object, in the variable 'target_loc'. Relative to this location the min/max values for the X/Y/Z coordinates are being calculated. These coordinates are used for the camera placement, so the camera is placed relatively to the object. 'YMIN', for example means, that the camera's Y value can be no further than 1.85m to the left of the object. 'YMAX' clamps this value so there is a border on the right side as well.

```
# Calculate values
X = random.uniform(XMIN, XMAX)
Y1 = random.uniform(YMIN, YMAX)
Y2 = Y1 + 1 if (Y1 - target_loc.y) < 0 else Y1 - 1
Z = random.uniform(ZMIN, ZMAX)
```

This code block does exactly what was described before. A random value for X/Y/Z is chosen. Notice that the Y value is split in two, where 'Y1' uses 'YMIN' and 'YMAX' and 'Y2' uses 'Y1' for calculation. This is to guarantee a different angle to the object, because camera one will use 'Y1' and the second camera will use 'Y2'.

```
# Add first camera
add_camera(X, Y1, Z, target_name)
write_dist_info(INFOFILEPATH, target_name, i, 1)
render(RENDERFILEPATH, i, 1)
del_cameras()

# Add second camera
add_camera(X, Y2, Z, target_name)
write_dist_info(INFOFILEPATH, target_name, i, 2)
render(RENDERFILEPATH, i, 2)
del_cameras()
```

This last code block is mostly the same for each camera. The first camera uses 'Y1' for its placement, centered on 'target_name' (the object). Then, from the location of the first camera, the distance to the object is calculated and written to a file. After the calculation of the actual distance blender renders an image from the first camera's point of view, after

which the first camera is being deleted. This is the same for the second camera, except 'Y2' is used for its placement.

4.2 OpenCV

OpenCV is a framework for image manipulation. Some of its use cases are changing the colour spectrum, filtering the image by colour and cropping images. The authors use OpenCV to test whether there are differences between filters for the images in the training data, for example greyscale images compared to coloured images. An example render, which OpenCV gets as an input can be seen in Figure 4.3.

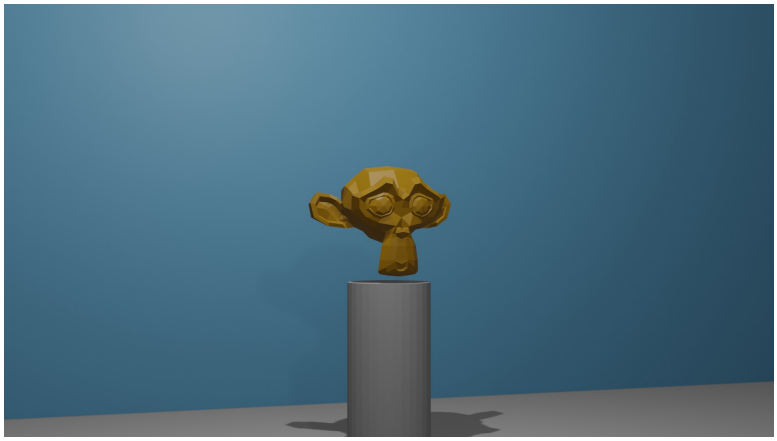


Figure 4.3: One of two original renders produced by Blender, both depicting the same object from different points of view.

4.2.1 Greyscale

Converting an image into greyscale can easily be achieved by the following OpenCV code:

```
1 import cv2
2
3 image = cv2.imread('path/to/image')
4 image_greyscale = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
5 cv2.imwrite('path/for/saving/greyscale/image', image_greyscale)
```

This code first reads the image into 'image'. Then it converts the color of 'image' into a greyscale format and stores the result into 'image_greyscale', which is then written to the specified path. The output generated by this code is depicted in Figure 4.4.

The advantage of using greyscale images in neural networks is the simplification of the input layer. In greyscale images each pixel can be represented by a single decimal value between 0 and 1. This enables the first layer of the neural network to be two dimensional

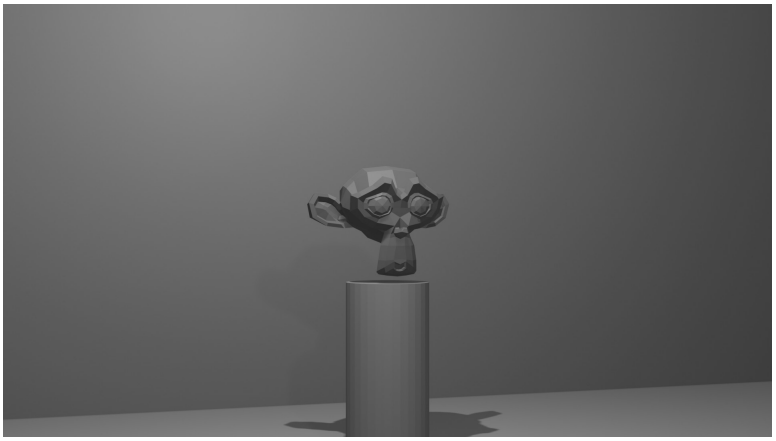


Figure 4.4: The greyscale image produced by OpenCV.

instead of the three dimensional counterpart, where each pixel is represented by the three decimal values for the red, green and blue colour channels.

4.2.2 Resolution

By reducing the resolution of an image the density of pixels is lessened. In this process information, that can not be regained, is lost. The OpenCV code for downscaling the images used by the authors is the following:

```
import cv2

image = cv2.imread('path/to/image')

scale_percent = 10 # percent of original size
width = int(image.shape[1] * scale_percent / 100)
height = int(image.shape[0] * scale_percent / 100)
dim = (width, height)

downscaled = cv2.resize(image, dim)
cv2.imwrite('{}/{}'.format(newdir_path, filename), downscaled)
```

OpenCV provides a resize function, which takes an image and the new dimensions of the image as an argument and outputs the resized image. To make sure the aspect ratio stays the same new image dimensions are calculated as a percentage of the original ones.

The output image of this code can be found in Figure 4.5.

Downscaling the images before they are passed into the neural network can be profitable, because the number of weights in the first layer of the network is reduced. This can advance the learning speed.

Figure 4.5: Image with lower resolution than the original one. Due to the fact that the image is displayed in the same size as Figure 4.3, the pixel size in this image appears larger.

4.2.3 Cropping

Cropping an image can remove unnecessary or unwanted parts of an image by simply cutting off areas. This is often wanted in photography to only keep what is interesting in a photo and to shift the view of the viewer to specific areas. However, the authors guess that cropping an image will worsen performance of the neural network, because information of the relative sizes is partly lost. It compares to zooming into the image.

Such a cropped image produced by the following code can be found in Figure 4.6.

```
1  import cv2
2
3  image = cv2.imread('path/to/image')
4
5  crop_margin_percent = 5
6  crop_margin_width = int(image.shape[1] * crop_margin_percent / 100)
7  crop_margin_height = int(image.shape[0] * crop_margin_percent / 100)
8
9  crop_img = image[crop_margin_width:-crop_margin_width, crop_margin_height:-
       crop_margin_height]
10 cv2.imwrite('{}/{}'.format(newdir_path, filename), crop_img)
```

Figure 4.6: In this image five percent of each side was removed, therefore the object appears closer if the image is displayed in the same size. Some information positioned in the outer areas of the image are lost during the cropping process.

4.2.4 Saturated

A saturated image means stronger colours, making them more distinguishable from each other. If an image is not saturated enough, colours appear as "washed out" and differences in colour are difficult to determine. Therefore, in order to help the neural network, the authors decided to also test with saturated versions of these images.

```
1  import cv2
2
3  image = cv2.imread('path/to/image')
4
5  hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV).astype('float32')
6  (h, s, v) = cv2.split(hsv)
7  s *= 1.5
8  s = np.clip(s, 0, 255)
9  hsv = cv2.merge([h, s, v])
10 saturated = cv2.cvtColor(hsv.astype('uint8'), cv2.COLOR_HSV2BGR)
11 cv2.imwrite('{}/{}'.format(newdir_path, filename), saturated)
```

To saturate an image using OpenCV it has to be converted into the HSV colour representation. The HSV representation specifies each colour as a hue, a saturation and a value. Therefore changing the saturation in this model is relatively easy, as the saturation value of each pixel can simply be multiplied by a constant. Before the image can be converted back into the bgr colour representation the saturation values are restricted between 0 and 255, the lowest and highest saturation possible. The result achieved by this code can be seen in Figure 4.7.

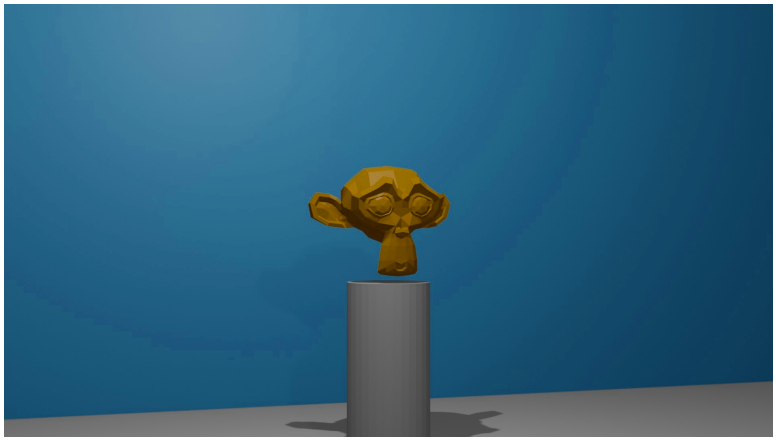


Figure 4.7: A saturated version of Fig. 4.3.

4.2.5 Brightness

Because the neural network should also work in different environment, where other methods could have problems, the authors also tested with overly bright images. This makes it very hard to notice dark areas, such as shadows, to help with distances between objects and scaling of objects.

```
import cv2

image = cv2.imread('path/to/image')

alpha = 1 # contrast
beta = 60 # brightness
bright_img = cv2.convertScaleAbs(image, alpha=alpha, beta=beta)

cv2.imwrite('{}/{}'.format(newdir_path, filename), bright_img)
```

Figure 4.8: The original image (Figure 4.3) modified by the brightening code. The image appears more washed out, as all colours are move similar due to them having moved closer to white.

4.3 Neural Network

4.3.1 Setting up the Neural Network

Using Python and TensorFlow enabled the authors to exchange the components of the neural network without much effort. This enabled a fast development of a working prove of concept. The structure of the neural network depicted in Figure 4.9 was the first one tested and already proved effective.

In the first test run only 50 greyscale images where used. This encourages overfitting and was only intended to test if the neural network was syntactically set up correctly.

As an optimizer the authors chose stochastic gradient descent as the other optimizers available in TensorFlow are only variations of this optimizer.

4.3.2 Problems Encountered: Normalizing

At this point in time the first problem arose. The accuracy when training the neural network always was equal to one, meaning that the distance to the object in all images was estimated correctly to the last millimetre, which does not seem plausible. Even if the neural network was overfitting, the accuracy should at least in the beginning have a value less than one.

It appeared later that this was due to the fact, that the true as well as the estimated output values were not normalized. In a neural network the output of each node depends on the activation function used by this layer. A sigmoid activation function, as used by the authors in the last layer, reduces the output values to numbers between zero and one. As all true values (the distance to the object outputted by Blender) where above one they where simply converted to a value of one. The neural network was trained to only output one, which it easily managed to do, and therefore received an accuracy of one.

Normalizing the distances was a simple task as the maximum distance in the generated images was 10 meters. Therefore simply dividing by the maximum possible value reduced

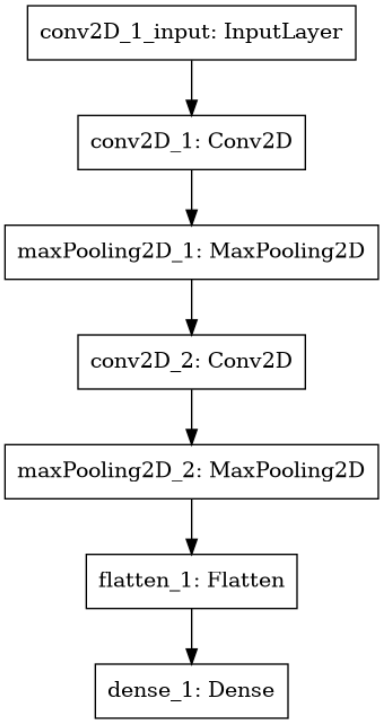


Figure 4.9: A sequential model of a neural network. The input layer receives the greyscale images. The image is then processed by two convolutional layers, each followed by a max pooling layer reducing the width in one dimension and therefore reducing the number of weights. The last layer, a dense layer, outputs the number representing the normalized distance the neural network estimated. The input of this dense layer is a unrolled version of the output of the last max pooling layer. This is achieved by the flatten layer.

the range of the values into the normalized form. This however had the consequence that the accuracy of the neural network was zero, however long it was trained.

4.3.3 Problems Encountered: Accuracy and Metrics

At this point it is noteworthy that in machine learning one can distinguish between classification and regression problems. In classification problems the output value can only be one of a discrete set of values. For example image classification, where a machine learning algorithm labels images as either depicting a cat or a dog. In this example two possible output values are defined. Regression problems however are identified by the output value being able to take any continuous value. The distance estimation problem the described in this thesis is a regression problem, as the distance to the object can be any value between zero and ten.

Accuracy is a system of measurement which describes how often the machine learning algorithm "guessed" the correct output. If the output differs in the last decimal place (in our case less than a nano meter) the accuracy does not get increased. Therefore accuracy is only used in classification problems, where the accuracy gets increased for example for correctly

guessing that an image depicts a cat (the output matching the actual value exactly). The equivalent used in regression problems is called metric. The metric chosen by the authors is the mean absolute percentage error. It represents a difference between the actual and the output value as a percentage. The lower this percentage the better the performance of the neural network.

4.3.4 Problems Encountered: Number of Training Images

After these modifications the output seemed plausible enough to increase the number of training images from 50 to 10000. Up to this point all images where loaded and converted into tensors before the actual training began. This posed no problem when loading 50 images, however loading 10000 images was not possible with the limited RAM installed on the machine. Since the process was running on a Linux machine, the kernel killed the process due to resource starvation.

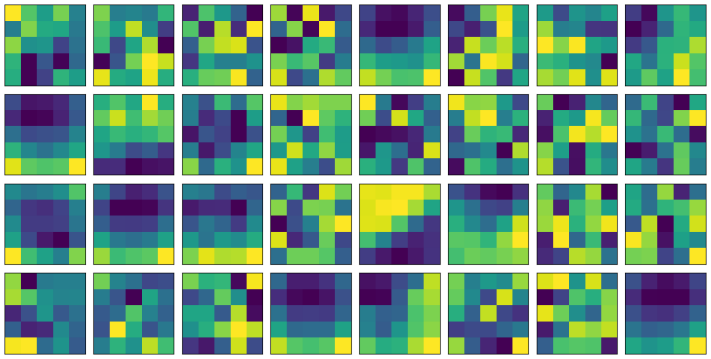
The most obvious solution was to split the images into smaller sections and only load and process one of these batches at a time. Additionally the authors chose to store the computed weights after each of these batches, enabling the user to stop the training at any given point in time and still have the relatively new weights stored. When starting the program again, the weights are loaded and training can resume.

4.3.5 First Results

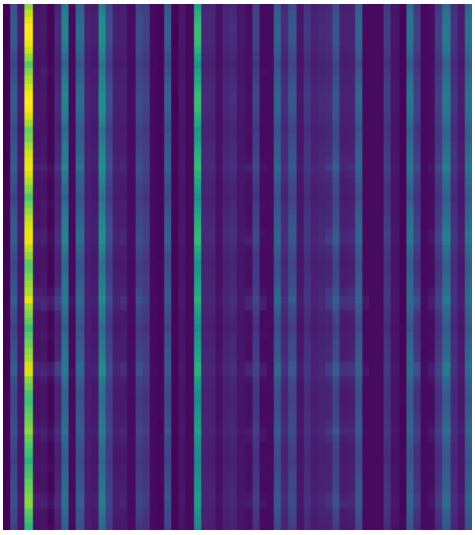
The first serious training run was performed on 2500 binocular images of a single object. Therefore it was expected, that the neural network would overfit and not work on any other objects. The setup of layers chosen was the one depicted in Fig. 4.9.

While training the loss sank to a value fluctuating between 0.21 and 0.27 while the mean absolute percentage error decreased to values fluctuating between 11.1% and 16.5%.

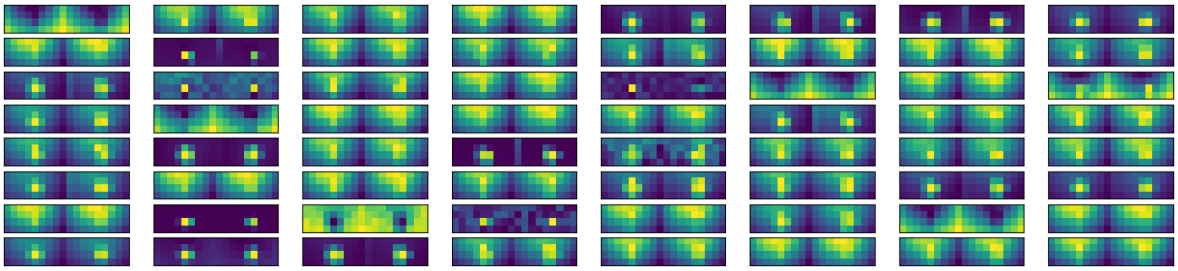
The weights of this trained neural network are visualized in Fig. 4.10 and Fig. 4.11. While interpreting weights of a neural network is a vague science, some features of these weights are worth mentioning. For example a few of the matrices shown in Subfig. 4.10a have a dark blue blob appear in the upper middle part, while the bottom left and right pixels show the brightest colours. Since brighter colours mean more influence of the input value received, the authors guess, that the size of objects could be determined by these convolution windows. These convolution windows could for example react to the size of the mount of the object, which is located in the lower part of the image. The second convolutional layer (Subfig. 4.11a) is hard to interpret, as the tensors are already quite abstract at this point. The dense layer (Subfig. 4.10b), however, gives a low weight to most input values, only using some for the calculation of the final output. Even more interesting are the horizontal stripes appearing in the weights of the dense layer. To get a better understanding of why these stripes appear the authors decided to group the weights of this layer into the sizes of the output of the last convolutional layer. The resulting image can be found in Subfig. 4.10c. From this it is apparent, that the neural network is able to work out that the input image consists of two separate renders. Most of these units focus on the two middle part of the two images separately, some on a larger a region, while other only on the exact place where the object is located. Only few units focus on the section where the shadow of the object is located.



(a) The already trained weights of the first convolutional layer. Each of the 5x5 pixel matrices displays the weights of one layer of the convolution window.

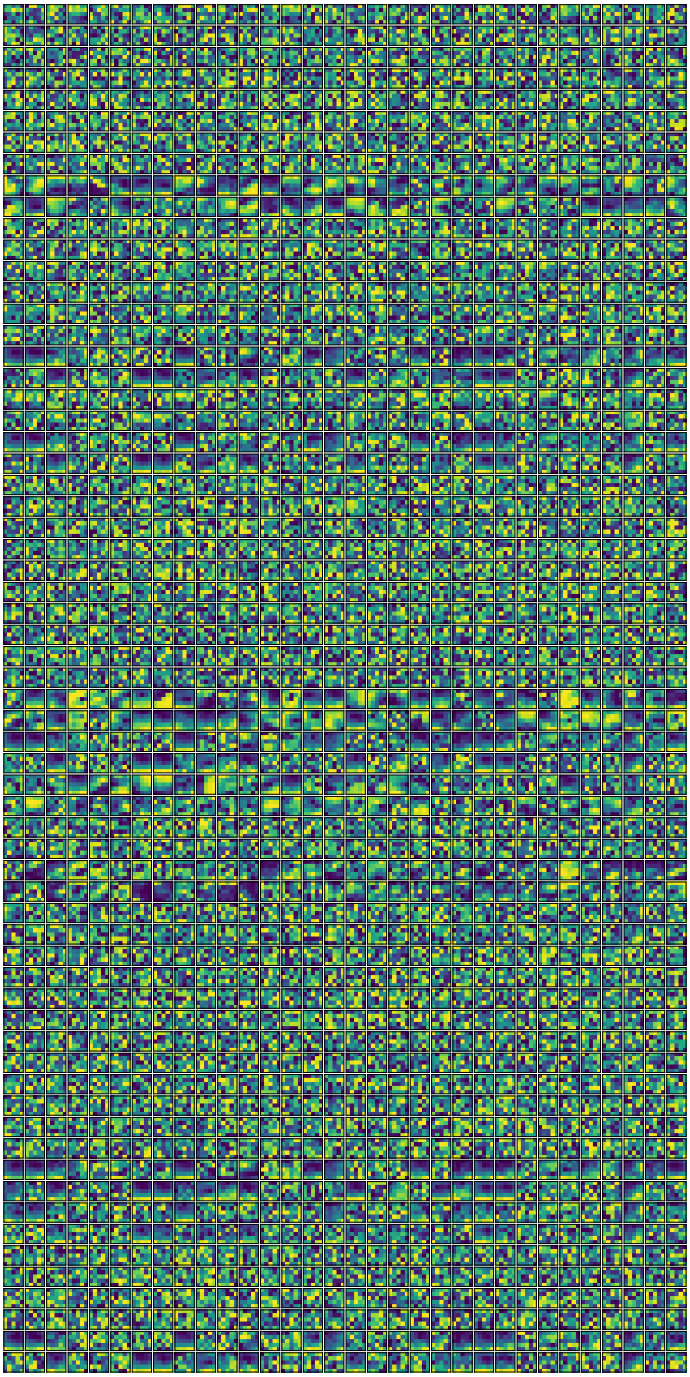


(b) The trained weights of the dense layer, which outputs the estimated result. Horizontal stripes appear because the one dimensional weight vector is displayed as a 72x64 matrix for convenience.



(c) The same weights as in Subfig. 4.10b regrouped into 64 4x18 units.

Figure 4.10: In all subplots the highest value occurring in a unit of pixels is displayed as a yellow colour and the lowest value occurring is a dark blue colour. All other values are shown as the corresponding colours ranging from yellow over green to dark blue.



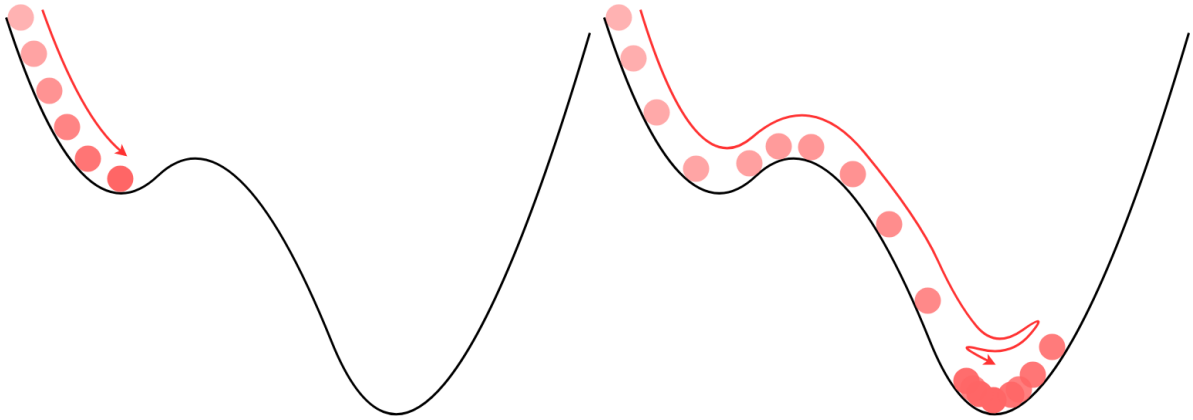
(a) The trained weights of the second convolutional layer. Again each of the 5x5 pixel matrices displays the weights of one units that make up the second convolution window.

Figure 4.11: As in Fig. 4.10 all subplots display the values ranging from yellow (high value) to dark blue (low value).

4.3.6 Modifications to the Neural Network

Since the authors wanted to reach an accuracy of ±10% and the first test only performed with an accuracy of around ±14% possible modifications to the neural network where tested. One of these modifications is the number of epochs, which control how often all images in the input data are used to train the neural network with. E.g. setting the number of epochs to three makes the fit method go through all input images three times. As it turned out increasing the number of epochs does not have any measurable affects on the performance. Therefore the value was set to a count of two epochs.

The next change to the structure of the neural network was the introduction of a momentum term. A momentum term helps to avoid getting stuck in a local, but not the lowest, minimum. One can imagine the optimizer as a mechanism which lets a sphere "roll" down a surface to find the minimum value. A two dimensional version of this is visualized in Fig. 4.12. When no momentum term is defined training behaves as if the sphere had no inertia, which can lead to it getting stuck in a valley, which is not the lowest reachable point. When inertia is added the change in the position of the sphere depends not only on the gradient of the curve in the current position of the sphere, but also on the last few gradients the sphere "rolled" along. Therefore the sphere in Subfig. 4.12b manages to find a lower minimum of the curve.



(a) Visualization of the optimization without a momentum term. The red sphere gets stuck in a local minimum, but fails to find the lowest possible point reachable.

(b) Visualization of the optimization with a momentum term defined. Here the red sphere has enough inertia to "roll" over the small bump, which enables it to reach the lower local minimum.

Figure 4.12: Two figures showing representations of optimization functions. In both cases the optimisation is complete when the red sphere reached a local minimum. The difference in the figures is the use of a momentum term.

In the case of the Neural Network described in the previous sections the addition of a momentum term sadly did not produce better results, as the mean absolute percentage error once again reached a value of around 14% before fluctuating by the same divergence as before.

Since the modification of the neural network using a momentum term was not sufficient the authors decided to test whether changing the number and type of layers would change

the results. At first one of the convolutional layers was swapped for a dense layer. This led to a remarkable decrease in the accuracy reached after training on only 500 images. Therefore the authors decided to test if increasing the number of convolutional layers would yield the opposite results. The resulting layer setup of the neural network can be found in Fig. 4.13.

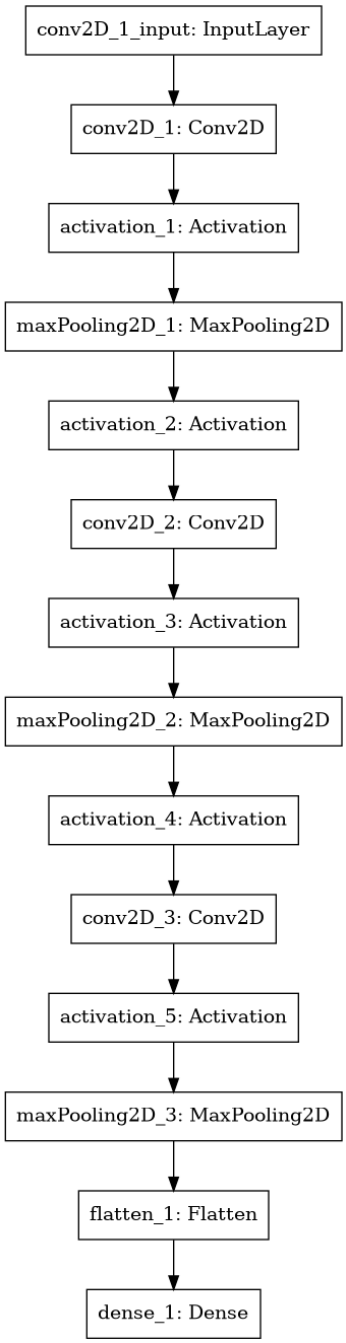


Figure 4.13: The layer structure from Fig. 4.9 modified to include three convolutional layer.

This new setup, having 156739 trainable parameters, is too large to train on the laptop used to this point. Therefore the training from this point on was performed on a PC satisfying the requirements for a neural network this size. The details of this process are described in Experiment 1 (Chapter 5).

4.4 C++ Implementation

The next step for the authors was implementing a neural network in C++ from scratch. This implementation would then be compared to Tensorflow by solving the same task with a neural network. The reason for this comparison is for the authors to see, how much a neural network can be optimized in order to achieve machine learning faster. Therefore the C++ implementation will not be heavily optimized, but rather just usable for any kind of machine learning with neural networks in order to see the difference.

4.4.1 Structure

The C++ implementation consists of three main parts: One major part is the neural network itself. The second part is for measuring time and the third part is for all math, such as some activation functions. Because the second and the third part should be self-explanatory the authors will just focus on the first and also largest part - the neural network.

The Neural Network in C++

The neural network is designed to be a sequence of layers, which consist of several nodes. There are three types of layers:

1. Input Layers
2. Hidden Layers
3. Output Layers

There can only be one input layer and one output layer. The main difference between the three types is that input layers have no connection to them, since they represent the input. Output Layers have no connection from them, because the are the final output. Hidden layers have both connections to and from them. Connections are achieved in the Node class via two lists (in C++ as a std::vector): One list stores all weights and the second list stores the neighbours of the next layer.
So to conclude: A node references the nth neighbour via the nth weight. The connection of layers happens in the NeuralNetwork class with a call to NeuralNetwork::create(). Before this call is made, the layers have to be added with NeuralNetwork::add_layer() first. As parameters this method requires:

1. The number of nodes in the layer
2. The number of connections per node to the next layer's nodes, which is regarded to as the dimension of the layer
3. The activation function, which also dictates the function used for error calculation
4. The type of layer

If the dimension is one then every node has exactly one neighbour in the next layer (except for nodes in the output layer, of course. In this case the dimension is always 0). If the dimension is the number of nodes in the next layer, we speak of a dense layer, where every node of the added layer is connected to every node of the next layer.

The Application class

With the neural network implementation done the first major part is finished, but we are still missing the learning process. In order to abstract that there exists the Application class. The authors tried to make it as simple and user-friendly as possible, so all needed to train your created neural network is creating an instance of this class, which means calling the constructor. The constructor needs a few things:

1. The reference to the created neural network
2. The inputs for the neural network
3. The expected outputs for the neural network
4. The number of epochs (The number of times it should train)
5. The learning rate it should train with
6. (Optional) A logging stream (can be a filestream or stdout, for example)
7. (Optional) (Bool) More verbose output (increases training time)

The inputs are used to initialize the nodes in the input layer. With help of the hidden layers the goal of the training procedure is to have an output layer holding the values of the outputs. Putting it all together - the following code achieves learning bitwise AND with 500 epochs and a learning rate of 0.15 in 2 milliseconds (Tested on a Ryzen 7 2700X on Windows 10, Compiled via Visual Studio 2019 with /O2 and 64 Bit):

```cpp
#include <vector>
#include <fstream>

#include "NeuralNetwork.h"
#include "Timer.h"

using namespace std;

int main()
{
    // Define Structure
    nn::core::NeuralNetwork neural_net;
    neural_net.add_layer(2, 2, nn::math::Function::Sigmoid(), nn::core::LayerType::
        Input);
    neural_net.add_layer(2, 1, nn::math::Function::Sigmoid(), nn::core::LayerType::
        Hidden);
    neural_net.add_layer(1, 1, nn::math::Function::Relu(), nn::core::LayerType::
        Output);
    neural_net.create();

    // Expected inputs/outputs
    vector<vector<float>> input_sets{{1.f, 1.f}, {1.f, 0.f}, {0.f, 1.f}, {0.f, 0.f
        }};
    vector<vector<float>> output_sets{{1.f}, {0.f}, {0.f}, {0.f}};

    ofstream log{"test_and.log"};
    nn::benchmark::Timer t{"AND learning"};

    // Learn!
    nn::core::Application and_training{neural_net, input_sets, output_sets, 500,
        0.15, log};
}
```

4.4.2 Implementation details

Because the neural network in C++ is written from scratch, the authors had to implement fundamental features for the neural network to work. This section describes those features in greater detail.

Error Backpropagation

Finding and adjusting errors is a very important part of a neural network, because this is the component which enables the neural network to learn and evolve. Error Backpropagation is called like it is, because it does its work from back to front of a neural network, or from the output layer to the input layer. So, the explanation will start at the output layer as well.

Calculating the errors

Our first goal is to find deviations of the neural network's output from expected outputs. In order to do this we first calculate the difference of the two outputs and then multiply that difference by the first derivative of the activation function applied to the output node's value of the neural network. This calculation is implemented like this:

```
// Calculate error of the output layer
for (std::size_t node{0}; node != nn.layers.back().size; ++node)
{
    float output_node_value{nn.layers.back().nodes[node].value};
    float error{outputs[curr_training_set][node] - output_node_value};
    delta.front().push_back(error * nn.layers.back().f.derivative(output_node_value)
        );
}
```

For the hidden layers the steps are similar, but we also have to take the weights into account: For each node of the current layer its weights to nodes of the next layer are multiplied with the error of the node in the next layer. All of these products are summed and again multiplied by the first derivative of the activation function applied to the hidden node's value. This can be difficult to understand by text alone, so you will find the relevant code again below:

```
// Calculate errors of the hidden layers
for (std::size_t layer{nn.layers.size() - 2}; layer != 0; --layer)
{
    delta.push_front(std::vector<float>());
    for (unsigned node{0}; node != nn.layers[layer].size; ++node)
    {
        float error{0.f};
        for (std::size_t next_layer_node{0}; next_layer_node != nn.layers[layer +
            1].dim; ++next_layer_node)
            error += delta[1][next_layer_node] * nn.layers[layer].nodes[node].weights
                [next_layer_node];

        delta.front().push_back(error * nn.layers[layer].f.derivative(nn.layers[
            layer].nodes[node].value));
    }
}
```

Adjusting the Neural Network

Now that we have calculated the errors we have to adjust the neural network so that the errors approach 0, which is also where the learning rate comes into play. We traverse the network from back to front again. First we have to calculate the bias for each node. For this we loop through the previous layer's dimension and then calculate the bias as the sum of the

error of each previous layer's node times the learning rate. Now we can adjust the weights. This is done by adding the product of the value of the previous layer's nodes, the error of the current layer's nodes and the learning rate to the weight of the previous layer's nodes. This is again better understood by looking at the code:

```
// Adjust neural network
for (std::size_t layer{nn.layers.size() - 1}; layer != 0; --layer)
{
    for (unsigned node{0}; node != nn.layers[layer].size; ++node)
        nn.layers[layer].nodes[node].bias += delta[layer - 1][node] * learning_rate;

    for (unsigned node{0}; node != nn.layers[layer - 1].dim; ++node)
        for (auto& prev_layer_node : nn.layers[layer - 1].nodes)
            prev_layer_node.weights[node] += prev_layer_node.value * delta[layer -
                1][node] * learning_rate;
}
```

Keep in mind that we don't store errors for the input layer, hence "delta[layer - 1]" actually refers to the current layer instead of the previous layer.

Chapter 5

Experiment 1

Author:

The first experiment tests whether the trained neural network from Section 4.3 generalises to other objects. Since it is impossible to train the neural network for all existing objects, this experiment tests if the neural network can, by learning from one object, determine the distance to other objects as well.

5.1 Setup and Environment

The addition of the third convolutional layer (see Fig. 4.13) allowed to solve problems of a higher complexity, as compared to before. For this experiment the neural network was further developed to attain a colour image as an input and output the corresponding x-, y- and z-coordinate. These changes dramatically increased the number of parameters and therefore the need for RAM and computation power.

The best system available at that time was equipped with a NVIDIA 2080 TI, which has 11 GB of VRAM (or "video random access memory"). To also fully take advantage of the CUDA cores (parallel processors for graphical computing), the authors decided to use Tensorflow with the GPU only. These changes resulted in much greater performance than before (10s to train with ten images on the previous configuration opposed to 0.32s on the current one). One disadvantage of this solution is, that only the GPU is used for computing, which means that only the VRAM can be used. But, since the performance was so much better, it was worth it to lower the batch size from 40 to 20 images at a time, which, after testing, is the optimal value and uses about 10.8 of the 11 GB of VRAM available.

5.2 Sequence of Events

To train the neural network, the authors chose the monkey head object, as depicted in Figure 4.2. This neural network is trained exclusively on the monkey head until the mean error is confirmed to be less than 10%. For training the authors use 500 distinct images each for monkey heads of 5 different sizes, totalling in 2500 images.

After the neural network has been trained and reached a respectable accuracy of around 85%, the authors will then test the neural network with images of another object. This object can again be seen in Figure 4.2. Because we are not training with these images, only 50 images for each of the 5 sizes are evaluated. First the training aspect (changing the weights of the neural network according to the new input) will be disabled, allowing the authors to figure out the accuracy of the neural network when viewing new images for the first time, without changing the neural network itself.

Then the training aspect will be enabled again and the time and effort needed to reach an accuracy of under 15% when retraining exclusively for the new object with the already pre-trained weights will be measured.

5.3 Results

After training the neural network with the monkey head object, the data predicted by the neural network for the vase object differed from the actual values as the following table depicts:

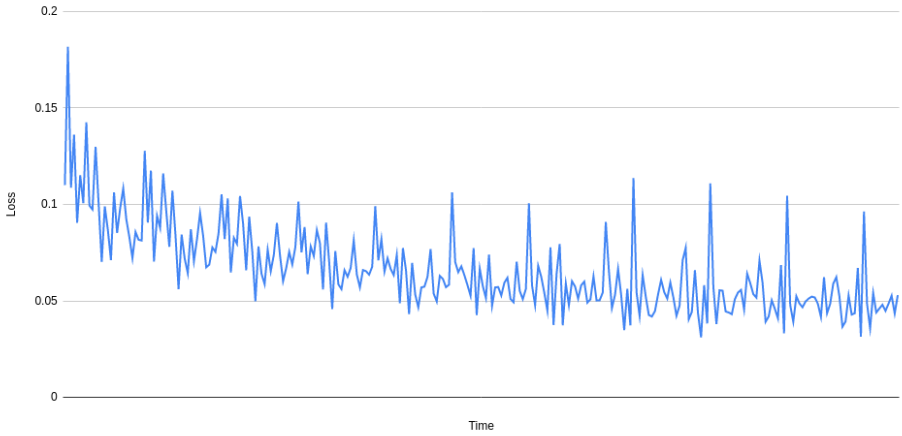
		Difference in meters
average error	x	3.023
	y	0.443
	z	3.18
	average	**2.215**
minimum error	x	0.011
	y	0.011
	z	2.259
maximum error	x	6.577
	y	1.224
	z	7.132

Table 5.1: Aggregated predictions of the neural network for all vase images.

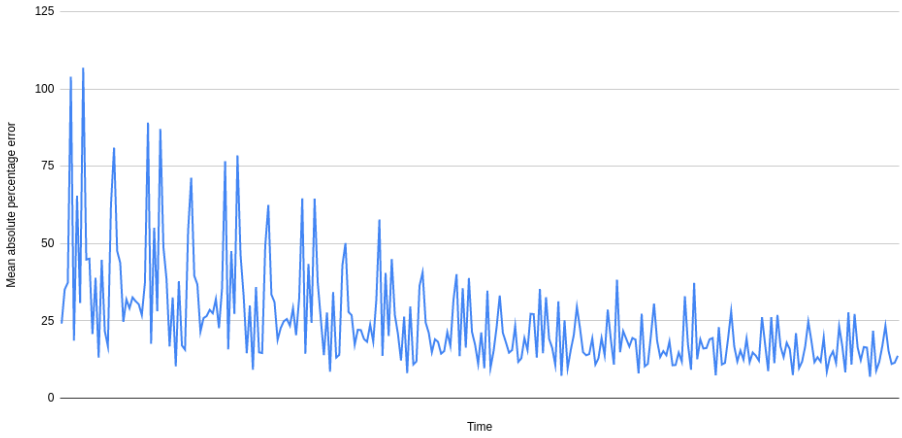
So overall this resulted in an average error of 2.215 meters.

The time needed to train the neural network with the vase images, which resulted in the desired accuracy of more than 85%, was 40.62 seconds, training with 1220 images of the vase object.

The history of the loss values during the training on the vase images can be seen in Fig. 5.1. This function is decaying exponentially, but it can be seen that the horizontal asymptote is relatively high. The same is true for the mean absolute percentage error over time. This means that there will always be some error and the maximum reachable accuracy is not 100%. This was expected, as not even humans are able to judge distance with 100% accuracy. Additionally the loss periodically jumps to a high value. This can be explained by a "faulty" image in the data, as depicted in Figure 5.2, which is used as training data periodically as the images are not used in a random, but rather in a sequential order.



(a) Loss over time when training with the vase images.



(b) Mean absolute percentage error over time when training with the vase images

Figure 5.1: Both, the loss and the mean absolute percentage error, decrease over time which means the predictions on vase images are getting more and more accurate.

5.4 Lessons Learned

Looking back the most time consuming part of the Tensorflow implementation was modifications to the already existing neural network of Section 4.3. This could have been avoided if the authors would have developed the prototype with the new hardware, where this experiment was run on, but this hardware was unavailable at the time of working on Section 4.3, so in the future the authors will try to perform all preparations, including developing the prototype, on the same machine used in experiments.

Training a neural network with many different objects before "showing" it a new one could have helped improving the accuracy in this experiment. Therefore the authors recommend everyone to train on images of various objects, when the final image is not known in advance.

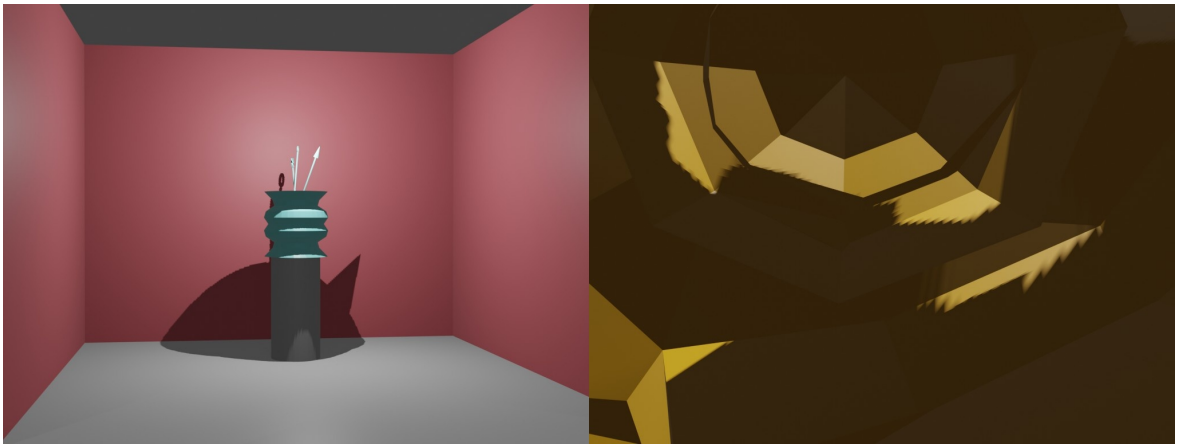


Figure 5.2: One of the occasional faulty images. This is caused by the camera view intersecting with some objects also present in the Blender scene such as the monkey head object or the wall.

If however, the object is known, as in the situation described in Section 1.2, training can be done specifically for the situation.

Chapter 6

Experiment 2

Author:

The second experiment is performed for the reason of testing the neural network with images of real objects. This experiment is mandatory for testing the capabilities of the neural network when it comes to real images.

6.1 Preparation

The first step of the authors was choosing a suitable object. In an effort to be time efficient once again the monkey head was chosen. This had the advantage that 3D-printing the object was easy, as all the authors had to do was exporting the existing model from Blender, and, even better, the trained weights of this object were already existing. Next the 3D model of the head was modified to include a cylindrical hole at the bottom, be able to easily mount the object. The last step was printing the object with a 3D printer. For this a Ultimaker Original and pink PLA (Polylactic acid) filament was used.

6.2 Setup and Environment

The setup consisted of a Parrot Bebop 2 and the 3D printed monkey head put on a stick to simulate the height of the head in the images produced by Blender. The authors chose a mostly white background, consisting of a table and a painted wooden board leaned on the table.

The authors were not specific about the lighting, because this experiment was to be performed as realistic as possible. For example in drone competitions the lighting often can not be influenced.

The monkey head was placed on the table facing away from the white wooden board. The drone was positioned at a distance of approximately 14cm from the object. The drone's camera was looking directly towards the object. After taking the first image the drone was moved to a second position, again about 14cm away and looking at the object. This resulted in the drone rotating slightly to the left to keep the object centred.

The images were taken with Parrot SA's FreeFlight Pro app.

6.3 Sequence of Events

The two images taken with the drone's camera from the different positions then had to be modified in order to be recognized by our neural network. The main differences between these images and the Blender generated ones were the different aspect ratio and a distortion, caused by to the way the camera operates. This can be seen in Figure 6.1. The authors wanted to make the images in (a) look like the images in (b), so modifications were necessary. The following modifications were executed with the program GIMP (GNU Image Manipulation Program), as the authors had previous experience with it:

First, the authors had to rotate the images to the left by 90 degrees, since the original images were saved in landscape. Then the authors needed to modify the aspect ratio, were cropping the image came into play. After cropping the image the final step was to get rid of the distortion. This is was relatively easy as GIMP already has a built-in distortion filter. After reversing the existing distortion the result can be seen in (c), which can then be used by our neural network to predict the distance to the drone.

Lastly the authors feed the final, undistorted images into the Neural Network, with the trained monkey head weights loaded.

6.4 Results

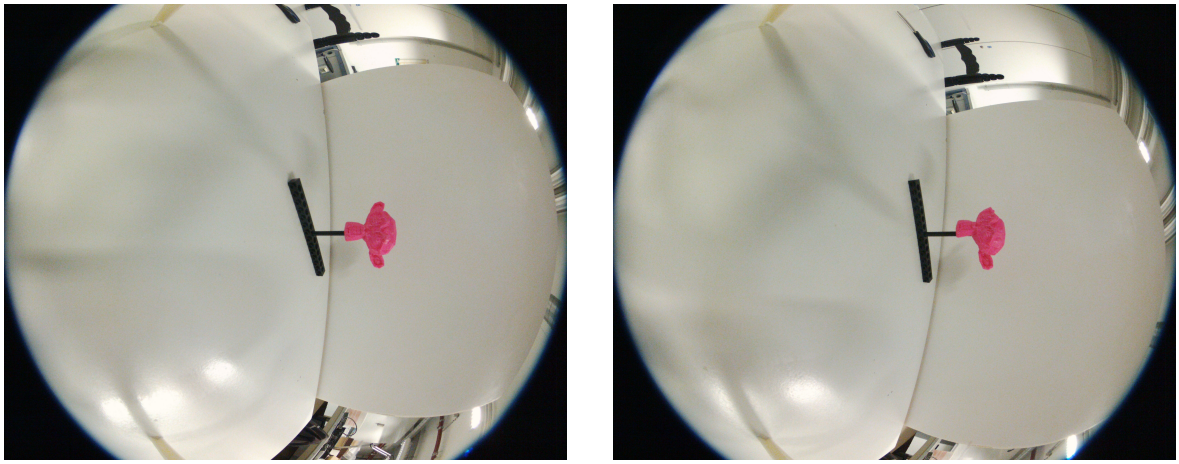
Because the preparation of the images made with the Bebop Parrot 2 camera was time consuming only one pair of images was tested. The result of the prediction of this image pair was the following:

```
images 0 to 1:
prediction - [3.451700 0.146270 -1.12407] actual - [0.130000 0.030000 0.020000]
```

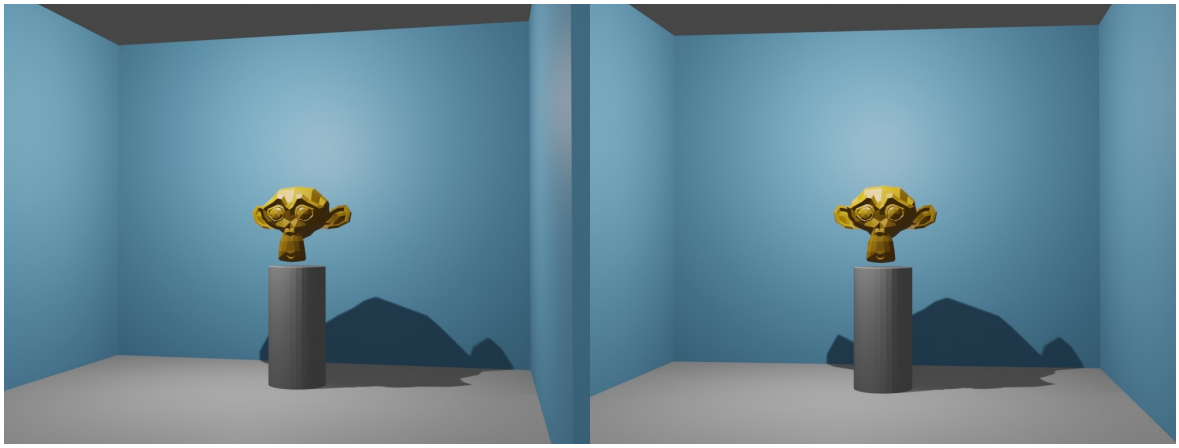
The fact that the prediction is off by more than one meter shows that the neural network was not able to generalise to real images. This could have multiple reasons:

- the units of the scenes in Blender could not be compatible with real world units this easily,
- the complexity of estimating real world distances could be a more complex problem, than the limited layer neural network build by the authors can solve,
- the different colours of the real objects, compared to the ones in Blender, could cause confusion,
- the neural network could have been trained for too long (overfitting) or
- the missing shadow, due to the different lighting situation could have caused the problem.

It is also possible that a completely different circumstance caused the problem, but the authors could not test any hypothesis due to the limited time frame of this work.



(a) The original images of the 3D printed monkey head produced by the Parrot Bebop 2.



(b) An example training image generated with the help of Blender.

(c) The two modified images of the 3D printed monkey head taken with the Parrot Bebop 2.

Figure 6.1: Image pairs of the monkey head.

Chapter 7

Conclusion

Author:

To conclude this diploma thesis, two things became apparent: First, it does not seem possible to train a neural network with computer-generated images only and then estimate distances of real images. Second, even when evaluating distances depicted by computer-generated images, there is still some error involved. How can one fix these issues and make the neural networks more reliable for this use case?

7.1 Possible Improvements

7.1.1 Input data

In order to estimate distances of real images there currently seem to be two choices: Either taking pictures of the object by hand, or modelling a more realistic scene in Blender and take pictures the same way the authors did.

Clearly the first choice has a drawback, which is the effort needed. Taking pictures by hand is something the authors wanted to avoid from the beginning, because it is just too time consuming. One would need at least 1000 pictures of the object and would have to measure the distances to the object in the required accuracy for every picture taken.

Modelling the scene with Blender clearly seems more reasonable, because it is something one can get done with some effort. Additionally the object and background can easily be changed to accommodate to a new situation. But some effort is still present and one will need some advanced modelling skills to convincingly create a realistic scene. There also is a factor of insecurity: Before adequate testing one can not be sure if this method is working.

7.1.2 Improving the Accuracy

Now for the second fact, the always present errors in the retrieved distance from the neural network: This is something that cannot really be fixed. Neural networks are not 100% accurate. This can be explained by the fact that not even humans are perfect at estimating distances. The author's neural network managed to estimate distances with an accuracy of two decimal places after all the modification described in this thesis. Humans generally can

not judge distance with only their eyes and brain with this high accuracy. This is because we do not really measure distance, rather we have a feeling of how distant something is. For example when picking up an object rather than estimating the distance and then moving our arm to this distance, we intuitively know how we have to move our muscles to reach the desired position. Since the idea of neural networks is to train similar to the human brain, the authors do not think that small but existing errors can be eliminated.

However, this does not mean that the always present error cannot be reduced: Both experiments use a neural network which outputs the distances to the object in x, y and z components. This can easily be simplified by just using the length of this three-dimensional vector, foregoing the information of the x, y and z coordinates and just using the length of the vector as distance to the object. This should increase the accuracy of the neural network, as the complexity is reduced. Similarly using images, which have been further downscaled simplify the problem at hand.

The other approach to improve the accuracy is to modify the structure of the neural network and the rest of the setup to cope with more complex problems. This could be achieved by adding more layers, increasing the number of trainable parameters; using different kinds of layers in the neural network; generating more input images, to decrease overfitting or using images of various objects on different backgrounds in the training data.

There is another approach: Staying minimalistic when it comes to input data. As seen in Section 4.3 when only training with one object and the same background, the neural network's error was 14%, which is acceptable. In a tournament situation you may not need to train for multiple objects and backgrounds, because you may only have that one obstacle you normally would have to setup an external camera system for, and evaluate the images retrieved by the camera with a script you wrote, as was the author's case. In this case training the neural network for this one obstacle and the same background can yield results which compare to Section 4.3, so you could expect an accuracy of around 85%, which is not that bad, since you do not want to steer close to the obstacle anyway.

7.1.3 Switching from TensorFlow to a C++ implementation

Since the workings of a neural network are not dependent on the programming language used, the theoretical part of this diploma thesis can be applied to self implemented neural networks as well. The C++ implementation of a neural network written by the authors will be made available to all future robotic students and can be extended to include all features necessary to perform the required estimations.

The missing features are for the most part in the development of the different layers described in Section 4.3.6.

Moving from a TensorFlow implementation to a C++ one can give an advantage in competitions, because you are not bound to a specific api probably everyone is using and you can expand the C++ API by adding features or optimizations specifically for the given challenge.

7.2 Outlook

7.2.1 Usability of this method

Still some problems encountered during this work are unsolved, but the authors think that the main challenge of using a neural network for this task was thoroughly explained in this thesis. Therefore the remaining work needed to get a working prototype running on a drone are limited to getting more realistic training data and wrapping the code in a few lines of instructions to the drone itself. In pseudo code these few lines looks like the following:

```
fly toward the general direction where the object is suspected to lie
turn toward the suspected position of the object
take the first image
fly some given distance to the side
again turn toward the suspected position of the object
take the second image
merge both images into one and perform image modifications as wished
send merged image to neural network
fly toward the given output of the neural network
now the drone should be approximately at the position of the object
```

The authors conclude that future robotic students will be able to use the knowledge presented in this thesis in upcoming drone competitions, if similar challenges to the ones described in Section 1.2 are posed.

7.2.2 Possibilities of further development

The authors suggest possibilities of further development of the developed code and the idea of this thesis:

Distance estimation in mono vision cameras with the help of a neural network

According to the visual clues used by humans to judge distance, described in Section **??**, humans are able to use a multitude of monocular depth cues. The possibility of developing a working program, which estimates distance based on monocular depth cues and a neural network seems possible.

Comparison between the work of this thesis and a stereo camera

One way to determine if the principles described in this thesis are sufficient to get acceptable results is the comparison with a stereo camera. One could set up both methods next to each other and explore the differences in the results.

Generating realistic Blender scenes

One of the missing parts, to get the neural network to work in the use case described in section 1.2, is the training with realistic images. The situation of having some given object and a unknown background has to be modelled in Blender and then feed into the neural network. Afterwards one could test if this is an improvement compared to experiment 2.

Writing a ROS (robot operating system) node from the existing code

Lastly the authors suggest all persons wanting to use the methods described in this thesis

to integrate them into a ROS node. This makes the communication to the drone more easy and completely removes the problem of the image modifications needed to perform the second experiment. Capsuling the distance estimation into a node has the following advantages: debugging the code is easy as all messages send and received by the node can be printed, the node can easily be replaced by some different new code and the calculation can be distributed onto many different computers.

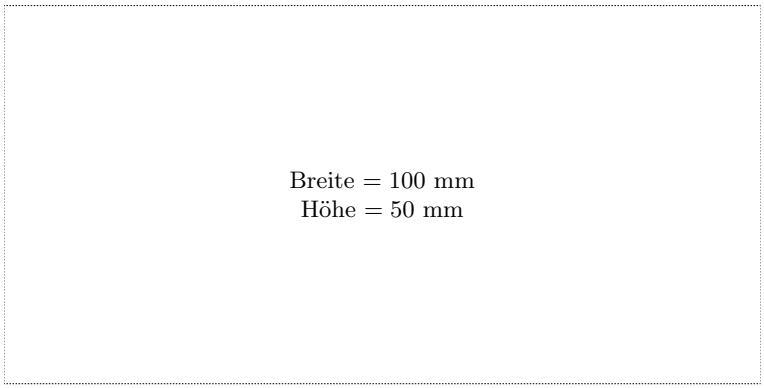
7.3 Final thoughts

All in all the authors hope to have given all readers of this thesis a deeper understanding of neural networks, their limits when it comes to processing images in search of depth information and the reasons behind computer-aided training data generation. They hope that the reader can use the obtained knowledge to get started in the development of neural networks more easily or, if they already have previous experience with this topic, deepen their insight.

Index

Messbox zur Druckkontrolle

— Druckgröße kontrollieren! —



— Diese Seite nach dem Druck entfernen! —