UNIVERSITÄT KOBLENZ · LANDAU

Fachbereich 4: Informatik

Temporal tracking of objects utilizing deep learning

Masterarbeit

zur Erlangung des Grades Master of Science (M.Sc.)
im Studiengang Computervisualistik

vorgelegt von

Marcel Pohl

Erstgutachter:	Prof. Dr.-Ing. Stefan Müller (Institut für Computervisualistik, AG Computergraphik)
Zweitgutachter:	M.Sc. Nils Höhner (Institut für Computervisualistik, AG Computergraphik)

Koblenz, im Mai 2019

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☒	☐

Koblenz, 4.5.2019 M. Pohl

(Ort, Datum) (Unterschrift)

Institut für Computervisualistik
AG Computergraphik
Prof. Dr. Stefan Müller
Postfach 20 16 02
56 016 Koblenz
Tel.: 0261-287-2727
Fax: 0261-287-2735
E-Mail: stefanm@uni-koblenz.de

UNIVERSITÄT KOBLENZ · LANDAU
Fachbereich 4: Informatik

Topic of the master thesis
Marcel Pohl
(Mat. Nr. 213 200 275)

Topic: Temporal tracking of objects utilizing deep learning

Tracking is one of the key aspects and challenges when developing Augmented Reality applications. It is essential for determining the position and viewing direction of the viewer, represented by the virtual camera. Tracking is also used to estimate the position and orientation of real objects, for exmaple to augment them with virtual pieces of information or interaction with virtual objects. For tracking there exist two different approaches. Tracking by detection searches every frame the camera transmits for the object to be tracked and – if found – computes its pose from scratch. Temporal tracking on the other hand computes the relative change in pose between two adjacent frames. For this, the pose of the previous frame has to be known.

The topic of this thesis will be the implementation and analysis of a temporal tracker which uses neural networks for pose estimation. The focus will be on analyzing the influence of the architecture of the neural network, particularly on the performance and precision of the tracking. A tracker is to be built, utilizing a neural network for pose estimation. The training of the neural network to recognize a certain object requires training data. To generate real world training data is costly and time consuming. For this reason it will be generated automatically by a 3D rendering pipeline.

With regards to content, the focus of this work will be:

1. Research on temporal tracking and neural networks
2. Familiarization with convolutional neural networks (CNN)
3. Generating training and test data
4. Creating a neural network and tracker
5. Analysis, evaluation and documentation of the results

Koblenz, 04.10.2018

M. Pohl — [illegible]

– Marcel Pohl – — – Prof. Dr. Stefan Müller –

Zusammenfassung

Tracking ist ein zentraler Bestandteil vieler moderner technischer Anwendungen, insbesondere in den Bereichen autonome Systeme und Augmented Reality. Für Tracking gibt es viele unterschiedliche Ansätze. Ein erst seit kurzem verfolgter ist die Verwendung von Neuronalen Netzen. Im Rahmen dieser Masterarbeit wird eine eine Anwendung erstellt, welche für das Tracking ein Neuronales Netz verwendet. Dazu gehört ebenfalls die Erstellung von Trainingsdaten, sowie die Erstellung des Neuronalen Netzes und dessen Training.

Anschließend wird die Verwendung von Neuronalen Netzen für Tracking analysiert und ausgewertet. Hierunter fallen verschiedene Aspekte. Es wird für eine unterschiedliche Anzahl an Freiheitsgraden geprüft wie gut das Tracking funktioniert und wie viel Performance dieser Ansatz kostet. Des Weiteren wird die Menge der benötigten Trainingsdaten untersucht, der Einfluss der Architektur des Netzwerks und wie wichtig das Vorhandensein von Tiefendaten für die Funktion des Trackings ist. Dies soll einen Einblick ermöglichen wie relevant dieser Ansatz für den Einsatz in zukünftigen Produkten sein könnte.

Abstract

Tracking is an integral part of many modern applications, especially in areas like autonomous systems and Augmented Reality. For performing tracking there are a wide array of approaches. One that has become a subject of research just recently is the utilization of Neural Networks. In the scope of this master thesis an application will be developed which uses such a Neural Network for the tracking process. This also requires the creation of training data as well as the creation and training of a Neural Network.

Subsequently the usage of Neural Networks for tracking will be analyzed and evaluated. This includes several aspects. The quality of the tracking for different degrees of freedom will be checked as well as the the impact of the Neural Network on the applications performance. Additionally the amount of required training data is investigated, the influence of the network architecture and the importance of providing depth data as part of the networks input. This should provide an insight into how relevant this approach could be for its adoption in future products.

Contents

1 Introduction

1.1 Motivation

Computing power has been increasing steadily over the last decades. Especially during recent years, this has resulted in a manifold of new mobile devices. Offering plenty of computing power with little power consumption, those enable a whole range of new applications. Augmented Reality (AR) is one of the technologies which have benefited greatly from this development. Devices like Microsoft Hololens or Google Glass offer us a glimpse of what the future has to offer. While this technology is not ready for consumers yet, it shows great potential and may change the way we work and perceive the world around us. At the time of writing this, Microsoft just announced the Hololens 2, which has been improved in many key aspects. It is more lightweight, smaller, has a greater field of view, offers more processing power and improved tracking.

Having a closer look at the software, tracking is a notably hard problem to solve. There are two different cases one might want to consider in the context of tracking for Augmented Reality. For one thing, the camera has to be positioned correctly in the environment. With access to RGB-D sensors, simultaneous localization and mapping (SLAM) techniques can be utilized to perform this task. As the Hololens demonstrates, this process works quite well already. The device is capable of setting up a room by scanning and thereafter navigating inside it. This allows one to place virtual objects within the physical environment and have them stick to their assigned locations. The second case is tracking of physical objects, for example to make them interact with virtual ones or to overlay information. There are various algorithms like the ICP and particle filters, but they come with limitations, such as having high computation costs, needing hand crafted features and not being robust to occlusions.

During recent years, machine learning has been another popular subject of research. While there are many different algorithms like support vector machines and decision trees, one type has been a particularly hot topic and has drawn a lot of attention: Artifical Neural Networks (ANN) and working on top of that, Deep Learning algorithms. While Neural Networks are best known to be used for Artificial Intelligence like Googles Alpha Go (beat a human Go world champion in 2016) or Alpha Star (beat a human professional Starcraft 2 player), they have also proven to perform well in computer vision. Utilizing Convolutional Neural Networks (CNN), they can be trained for tasks such as the recognition of characters or complex objects. By now there are large image databases like MNIST (character recognition) and CIFAR (complex objects) which can be used for training and testing the accuracy of ANNs. There are even competitions held like the Imagenet Large Scale Visual Recognition Challenge.

The focus of this thesis will be to combine the aforementioned topics and implement an object tracker which uses a Convolutional Neural Network for pose estimation. The tracker will work temporal, which means it is not going to calculate the pose from scratch every frame, but calculates the relative movement of the object between the current frame and the one before. To train the Neural Network, training data has to be generated. As this task is costly and time consuming when performed in a real world scenario, a geometric rendering pipeline is used to generate RGB-D training images. Finally the results will be analyzed and the possibilities and problems of this approach will be discussed.

1.2 Structure

This thesis consists of six chapters. At first related works are presented and discussed, showing different approaches to tracking and how they evolved over time. This is followed by defining the goals for this thesis in the next chapter. At this point the concept for creating training data, training a Neural Network and the tracking process are designed as well. The last part of the chapter defines technologies and frameworks used in the scope of this thesis. Chapter 4 provides the foundations for Machine Learning are provided. This is started by giving general definitions which are essential to all types of Machine Learning and then providing an overview of Neural Networks, which also includes a more detailed view at the Convolutional Neural Networks used in the application. The next chapter details the design of the object tracker, which consists of three stand alone applications required to cover the different tasks. Here the generation of training data, the training of the Neural Network and the final object tracker are explained. Chapter 6.1 focuses on analyzing and evaluating the different aspects of the tracking applications. It begins with checking how well the tracking process works for different degrees of freedom, followed by evaluating the influence of different aspects like the amount of training data, the network architecture and the usage of depth data. The thesis is finished by drawing a conclusion.

2 Related Works

Object tracking has been researched extensively for decades and is already applied to many practical applications. Examples for common tracking algorithms are the Kalman Filter [1], Optical Flow [2] and Particle filter [3]. Since many tracking algorithms work with 2D tracking only, the focus will be on works related to 3D tracking as well as Neural Networks.

Tracking of objects in three dimensional space is a highly geometric problem, which can be solved with algorithms like the ICP [4]. While the ICP was intended for registering 3D objects and finds its application for example in SLAM algorithms, it can also be used for tracking and pose estimation [5][6]. Given an object that has moved in space, this movement can be represented as two instances of this object placed at the start and end positions. The ICP will try to match their shape and return a transformation vector. This vector contains the translation and rotation which can be applied to either object instance in way that both instances will end up on top of each other. This process requires distinct and large enough objects to work with, and is computationally expensive. To track smaller objects with high accuracy in real time, more sophisticated methods are necessary. Possibilities include algorithms which utilize traditional image processing methods[7] and particle filters [8], both depending on handcrafted features.

To simplify and automate the process of finding feasible features, recently the use of data-driven methods has been proposed. These try to leverage Machine Learning algorithms to find optimal feature sets to enable fast and robust tracking. An example is the use of Random Forests [9] in the 3D temporal tracker by Tan et al. [10]. Only using depth data, they can calculate the displacement between points of the object and the current depth frame and feed a random subset of those values into a decision tree to estimate the change in pose. The random forest is trained with the object from various viewpoints. Occlusion handling is done in two ways. For once, the selection of random points makes the tracker robust against holes in the depth image. To decrease the impact of actual occlusions, the object on the depth image is divided into two regions and only one of them is used for random sampling. With this purely being a temporal tracker, Akkaladevi et al. [11] expanded on this approach and added a component to perform a randomized global object localization (RANGO). It is used to initialize the object location, but also for re-initialization in case the tracker loses the object.

With increasing popularity of Neural Networks, particularly Convolutional Neural Networks (CNN) which perform very well in computer vision [12], they were picked up for tracking. Gan et al. [13] used the ability of CNNs to recognize objects to perform temporal 2D tracking of the bounding box of an object, but not the actual pose of it in 3D space. Others solved a variety of 3D geometric problems like estimating the camera pose from a single RGB image [14], object pose estimation from a single RGB image [15] or deriving a transformation between two input images [16].

Oberweger et al. [17] made use of CNNs for hand pose estimation, combining the output of three networks. In their approach they utilize depth images only. They first feed a recorded depth image into a CNN to predict an initial estimate of the pose. This estimated pose is used as input for a second CNN which generates a synthesized image. Both the original and synthesized image are fed into a third CNN which derives the differences resulting in a pose update. The update is applied and the process is repeated to improve the pose. This feedback loop ensures that the mistakes made by a CNN trained for hand pose estimation can be corrected. In context of object tracking it is important to mention the difference in nature of hand-pose estimation. Every joint of a hand is tracked separately, hence the resulting hand pose actually is a combination of many single joint poses.

Garon et al. [18] expand upon this idea to create a temporal 6 degrees of freedom object tracker. While the concept of feeding a CNN with two images to obtain a pose update is the same, there are some major differences. Instead of just depth data, they make use of a RGB-D image. They also require just one CNN in their method, the synthesized image is generated using a geometric rendering pipeline. The pose estimation loop consists of two steps. First the synthesized image is generated from the previously estimated pose. Afterwards this image and the current RGB-D frame are fed into the CNN to obtain the change in pose. Finally, the pose can be updated accordingly and the process is started over for the next input. Their experiments showed that this setup resulted in a robust tracker which was also able to handle occlusions when the network was trained accordingly.

3 Requirements and Concept

3.1 Goals

This thesis aims at creating an application which is able to track objects in three dimensional space and estimate the objects pose. Input will be the initial pose of the object and a sequence of images in which the object is to be tracked. The initial pose estimation will not be part of this work. The output will be the tracking result in form of the relative pose changes every frame as well as an optional 3D representation of the object rendered onto the tracked position. To keep the scope of this work manageable, at first only two degrees of freedom will be considered, moving the object along the x and y axes. Additional degrees of freedom may be added later on in case there is enough time left to do so.

The main difference compared to other tracking applications will be the usage of a Neural Network which is responsible for the tracking part. The goal is to make the process of manually finding and designing features for object tracking obsolete. Instead, those features should be learned automatically by the Neural Network. A more robust tracking, especially when the object is partially occluded, would be a desirable result of this process. The Neural Network will be trained in advance to recognize the object it will have to track as well as the changes to its position. The tracker itself will only need to load the trained Network and feed it with the necessary inputs to receive the change in pose.

Furthermore the influence the design of the Neural Network has on the tracking process will be analyzed. There are many different types of Neural Networks, some of them will be discussed later in chapter 4 about Machine Learning. As certain types have been proven to be very efficient for specific areas of application, this work will make use of one which is well suited for image processing. The design of this specific network type will be analyzed, which includes different aspects like the type of layers used, the number of layers, their size and the way they are connected to each other.

3.2 Concept

The project will consist of three standalone applications. The first will generate training images, the second will use this data to train the Neural Network and the last will be the actual tracker.

The training image generator will be used to generate a large training data set. Looking at popular datasets available on the internet, the MNIST dataset [19] - containing hand written numbers from zero to nine - includes 60,000 samples for training and 10,000 for testing. CIFAR-10 [20], a dataset for object classification, contains 50,000 samples for training and

10,000 for testing. Garon et al. [18], which this work is based on, generated 250,000 training images. Taking these numbers in consideration, the minimum amount of training images generated for this project should be around 50,000. The output for every sample will consist of three files, two images and a text or binary.

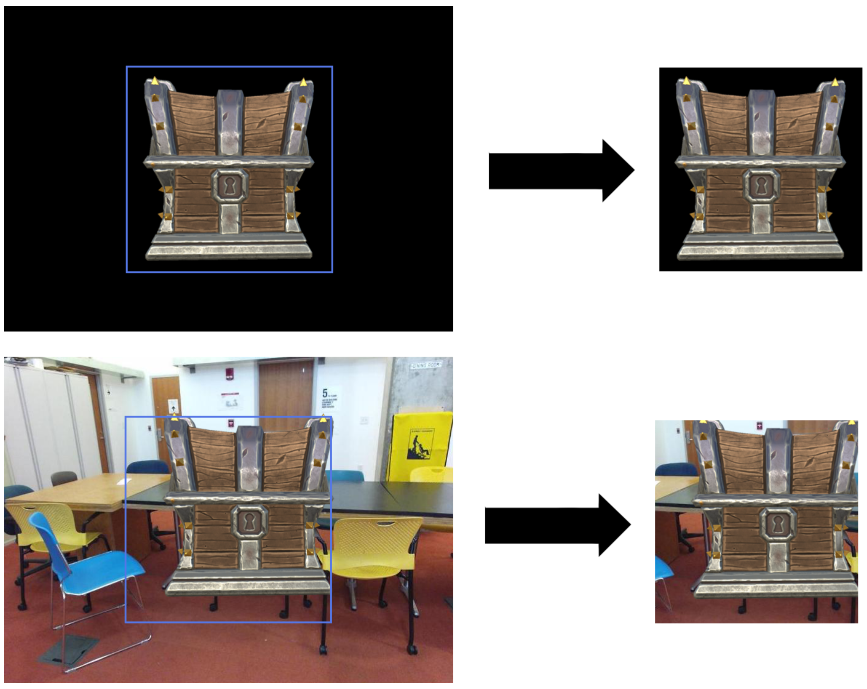


Figure 1: Mockups for the image generator. On the left is the full render, on the right the normalized output. Top: The image shows the object in the previous position. Bottom: The image shows the object in its current position. The background for this image is taken from [21].

One image will be a render of the tracked object in its previous position, using a geometric rendering pipeline. There will be nothing else added, only the object on a plain background. It is important to note that the previous position will be the one which was *predicted* during the last iteration of the tracker. The second image will be a representation of the current frame a camera would have recorded. It is the object rendered in its current position, this time with a background. The second image can either be fully rendered or the object can be composed onto an existing recorded image. Garon et al. [18] render their object onto images taken from the SUN RGB-D dataset [21]. Both images will contain RGB-D data and will be normalized to a fixed square size. To achieve this a square bounding box is drawn around the object in the image representing the previous object position on the empty background. The selected area is then cut out and scaled to fit a fixed size defined for all images. In the second image representing the camera image showing the object in its current position, the bounding box will be placed at the same coordinates having the same size, the process of cutting and merging is repeated. This results in two output images, both of the same size. One showing the object in its previous position, perfectly centered, the other showing the object in its current position, maybe partially outside the selected area. Mockups for the two images can be found in figure 1.

The text or binary file will contain the transformation vector which was applied to get from the previous to the current object position. In the first step, this will be the x and y translation values, which can be extended with other degrees of freedom (z and rotation) later on. The images will serve as input for the Neural Network, which should derive the change in pose from these. The text or binary file will act as a label against which the output of the network can be compared to.

The second application will take care of several tasks. First a Neural Network will be created. Since they have proven to perform well at image processing, a Convolutional Neural Network (CNN, see section 4.2.4 for more details) will be used. The application will define the networks structure, including layers and their parameters, activation functions and the connections between layers. Once this is completed, the training data will be loaded. After this the training process is started. It will feed batches of training data into the network, compute the results, determine the error and adjust the network parameters accordingly. Once the training process is concluded, the model (which is the trained instance of the Neural Network) will be saved. The training application should also be able to load an existing model and continue its training.

Finally, the last application will combine the aspects of image rendering and Neural Networks to perform the actual tracking of objects. It first has to be initialized before it can begin the tracking process. The trained model of the CNN has to be loaded and the geometric rendering pipeline set up. The source of the input images has to be defined as well. This can either be a sequence of prerecorded images or a camera feed, both however need to provide RGB-D images. For testing purposes this work will use a prerecorded sequence, which will also provide the initial pose of the object to track. In the main loop of the program the following steps are performed:

- First the pose of the previous frame (or initial pose for the first iteration) will be used to render a 3D representation of the object in the last pose it was seen. Like in the training image generator, nothing else will be added to the image. The same steps to normalize the image to the desired size have to be performed as well.
- The rendered image as well as the input image of the image sequence will serve as input for the CNN. The network is run and the change in pose calculated.

- The output of the Neural Network is then used to transform the previous pose into the current.

The result of the pose estimation can be visualized. A 3D representation, only shaded in a single color to have a distinct representation, will be rendered onto the current frame using the new pose and displayed in the viewport of the application. A successful estimation should cover the original object. The main loop then continues to process the next input frame. A mockup for this can be found in figure 2.

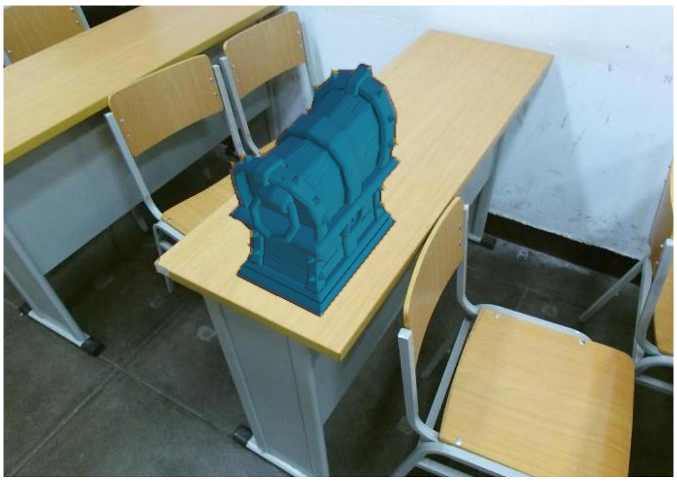


Figure 2: Mockup for the graphical output of the tracker. The tracked object is rendered on the estimated current position, which should cover the actual object in the image. The background for this image is taken from [21].

3.3 Technologies and Frameworks

This project has two main aspects to focus on: rendering of images and tracking. The tracking is done by a Neural Network and besides the training, it is only required in the final tracker application. The image rendering is used at several stages in the project. First of all the training data has to be generated. As it is too time-consuming to record actual real world data for training, these images are computed using a geometric rendering pipeline. In the final tracker the geometric rendering pipeline is required in two places, to generate additional input images for the Neural Network (will be discussed in 5.3) and to render a representation of the tracked object.

3.3.1 Platform

The platform of choice will be a desktop PC with a CUDA enabled graphics card. This will ensure fast training and computation times utilizing the

GPU acceleration built into the Machine Learning frameworks. All the dependencies for the project like CUDA, Machine Learning frameworks and the rendering pipeline work on both Linux and Windows. Since Windows has been my primary operating system and it is easy to set up the dependencies, it will be the OS of choice for this project.

The Machine Learning framework and the geometric rendering pipeline will both be used in the tracker application of this project. For this reason it is required that they use the same programming language. However, while most Machine Learning frameworks use Python as their primary or only programming language, the language of choice for computer graphics is C++. Some of the bigger Machine Learning frameworks provide a C++ API as well, hence C++ was chosen to be the preferred language for this project unless no framework would work well with it. Which framework was chosen will be the subject of the next section.

The 3D rendering pipeline will be created using OpenGL. The advantage of OpenGL over DirectX is the availability on many different platforms and most notably, that I have been working with it for years. This also means that while the applications will be developed on Windows, they can still be compiled and used on Linux or other platforms if desired. Additional dependencies are Assimp, GLFW, GLM, GLAD and OpenCV. GLFW, GLM and GLAD all provide an easy-to-use setup for OpenGL. Assimp is used to load 3D models and OpenCV to load and save images and do image processing tasks.

3.3.2 Machine Learning Framework

At the time of writing, there are three machine learning frameworks which are well known, well supported and have large communities around them. They are TensorFlow, PyTorch and Caffee.

TensorFlow is the biggest of the three. It is an open source project developed and supported by Google and was released in 2015. TensorFlow supports CPU as well as GPU computing. It is used in Googles own applications, mostly for image and speech recognition. While this usage is mostly hidden in the application itself and not much talked about, the Alpha applications have caused big news headlines. As of now there are two Alpha projects: AlphaGo and AlphaStar. The first one beat a world champion in the game Go, the latter a professional player of Starcraft 2. Officially TensorFlow provides a Python as well as a C++ API. While the Python API worked well, unfortunately I was not able to get the C++ API to work in an acceptable way. The examples provided by the TensorFlow team only show working with the C++ API within the TensorFlow project structure, using Googles internal build tool Bazel. This would mean a massive overhead for the project compared to using libraries which can be included into my own project structure. Including the required libraries for the geometric rendering pipeline would not be easy either, as it meant to include those additional dependencies into the existing, large TensorFlow project structure. While there were people exploring the possibility to build libraries and binaries on your own from the TensorFlow project files, unfortunately those articles were outdated and did not work anymore with the current TensorFlow versions.

Caffee is a framework developed by the Vision and Learning Center of Berkeley University in 2014. While its roots lie within research projects, it is suitable for industrial projects too. Like TensorFlow it offers APIs for C++ and Python. Later the Facebook Research team published Caffee2, which was merged with PyTorch in 2018. For this reason I did not do a more extensive evaluation of this framework and instead focused on the last framework, PyTorch.

PyTorch is based on Torch, which was released in 2002 under an open source license. It was developed by the Facebook AI research team and was released in 2016. As mentioned above, it was merged with the Caffee2 framework in 2018, combining both projects into one big framework. While previously the API was available for Python only, just recently a C++ API has been released as well, providing the same interface the Python one offers. Like TensorFlow it has CPU and GPU enabled computing, but additionally it also offers libraries and binaries for use in your own projects. This makes it a lightweight and easy-to-include alternative to TensorFlow.

Comparing TensorFlow and PyTorch there are just minor differences. Both projects have been growing rapidly during the last years, with TensorFlow having the bigger community and being used in more projects. While the focus of TensorFlow lies on production use, PyTorch has seen more research usage. Both offer all the functionality needed in this project, are CPU and GPU enabled and are comparable in speed. Since PyTorch offers not only a C++ API but also easy integration by providing libraries and binaries, the decision was made to use this framework for the project.

4 Machine Learning

In this thesis, Machine Learning (ML), particularly Neural Networks, will be used for tracking. For this reason this chapter will give a short introduction into those topics. First some general definitions in the context of Machine Learning are explained, afterwards there will be a look at Neural Networks. Detailed explanations on the topics in this chapter can be found in [22] and [23].

4.1 General definitions

Machine Learning is used to perform a variety of *Tasks*, usually ones that are "too difficult to solve with fixed programs written and designed by human beings" [23]. Generally speaking, a task consists of processing a *sample* given to the ML Algorithm, analyzing its *features* and delivering a result in the context the developer expects. Samples are usually represented by a vector $\mathbf{x} \in \mathbb{R}^n$, with every entry $\mathbf{x}_i$ being a feature. Common tasks can be:

- **Classification**: The sample is to be assigned to one or more provided classes. Usually this process is described as a function $f : \mathbb{R}^n \rightarrow 1, \ldots, k$. A common output is one or more tuples assigning the sample to one or more classes, but it is also possible to provide the probability distribution over all classes. Examples for classification is text recognition or object classification.
- **Clustering**: This task is similar to classification. The given sample is to be assigned to one or more clusters. Usually clusters are not provided by the designer of the algorithm, but are learned through finding similarities between samples used for training. Hence the algorithm attempts to find "classes" which distinctly group the samples.
- **Regression**: The algorithm is tasked with predicting numerical values for a given sample. This process can be described as a function $f : \mathbb{R}^n \rightarrow \mathbb{R}$, which clearly illustrates the similarity and difference compared to a classification task. Tracking, the task at hand for this thesis, is a problem of regression as well.

To solve a given task, a ML algorithm has to be chosen first. After its implementation, a **model** has to be trained. A model is a set of parameters the algorithm calculates during the training process, which is later used to determine the output for a given input. The model is trained using an entire **dataset**, which is a collection of many samples, sometimes also refered to as **data points**. A dataset usually contains at least tens or hundreds of thousands of samples to cover a wide variety of possible in- and outputs.

Most Machine Learning algorithms, to be more precise their training process, can be categorized as **supervised** or **unsupervised**. **Supervised learning algorithms** use labeled samples for training. Classification is an example for this kind of algorithms. The training data consists of samples which contain the input, like the image of a number, as well as the class the algorithm is supposed to assign to it. With this information available, the ML algorithm can compute a result for a sample input using its current set of parameters, calculate the error between computed and intended result and adjust the parameters accordingly. For **Unsupervised learning algorithms**, only the input values are provided, no additional information on the expected outcome. The algorithm has to find the parameters which fit the input data best on its own. This is usually used for tasks like clustering, for which a general structure of the data is to be found and no specific classes are necessary or even known.

During the training process, the algorithm attempts to fit the model to the training data. The goal is to find a model which predicts all the training data correctly when fed with the training data, but still is able to predict unknown samples correctly. The ability of a model to fit a variety of functions is called its **capacity**. The higher the capacity of a model, the greater the variety of functions it can fit.

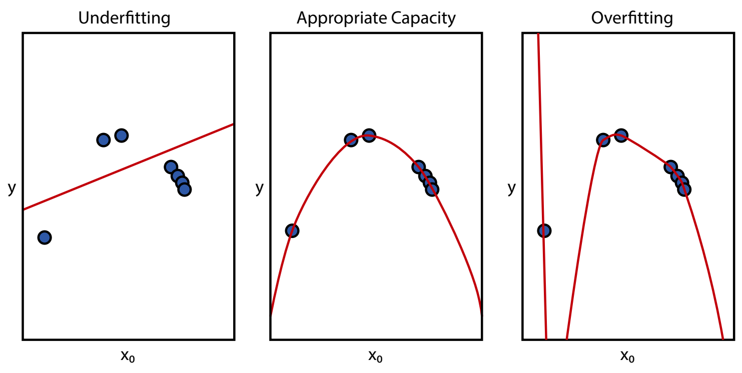


Figure 3: Underfitting, overfitting and the appropriate capacity of a model [23]

Two problems which can occur during training are overfitting and underfitting. In case of **underfitting**, the algorithm is unable to find parameters for the model which result in low error values on the training dataset. As a result, the algorithm will not be able to score a high amount of correctly predicted results even on the training data. Most common causes for this problem are an insufficient number of parameters (low capacity of the model), a training dataset which is too small or a lack of variety within the training data. **Overfitting** is the opposite of underfitting. The trained model performs well on the training data, but fails to predict new samples correctly. It is fit too tightly to the training data and does not allow for deviations in data it has never seen before. The causes can be similar to underfitting, like a training dataset which is too small or has too little variety. In this case however, the number of parameters and features used for the model may be too high (high capacity), and it may help to reduce the complexity of the model. The concepts of underfitting, overfitting and a properly fit model are illustrated in figure 3.

4.2 Neural Networks

Neural Networks is an area of Machine Learning that has gained a lot of popularity recently. While the basic idea and structure is the same for any network that is built, there are a wide variety of concepts and designs, focusing on different tasks. This section will discuss the basic concepts of Neural Networks and present the architecture of Convolutional Neural Networks, which will be used for the tracker.

4.2.1 Neurons

A **Neuron** - or perceptron - is the smallest unit of a Neural Network, every network is built upon this concept. The basic idea of a neuron is to take an arbitrary number of inputs and return a result in one output. A schematic of this idea can be found in figure 4.

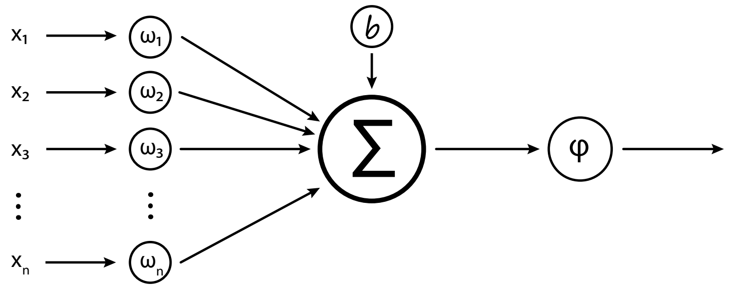


Figure 4: The structure of a neuron

In a basic neuron the inputs are first weighted and then summed up. A bias can be added onto the result, which adds a precondition for activation of the unit, for example in case the input data has a general bias. As a final step, an activation function φ is applied which determines the output of the neuron. Before training, the weights are initialized with random values and will get adjusted during the training process.

There are a great variety of **activation functions**. As their name implies, they determine the conditions for a neuron to activate and influence the output as well. A few examples for common and often used activation functions are:

- **Binary**: The simplest type of activation function. When defining the Neural Network, a threshold is assigned. When the sum of weighted inputs exceeds the threshold, the neuron will output 1, otherwise it will output 0.

$$f(x) = \begin{cases} 0 & \text{for } x < \theta \\ 1 & \text{for } x \geq \theta \end{cases}$$

- **Sigmoid**: The sigmoid activation function clamps the input values between 0 and 1. Positive values will result in an output of > 0.5, and negative ones in an output of < 0.5.

$$f(x) = \frac{1}{1 + e^{-x}}$$

- **Tanh**: The tanh activation function is very similar to the sigmoid function. As the name implies, it applies the tanh function to the input. This results in the input values being clamped between -1 and 1.

$$f(x) = tanh(x) = \frac{2}{1 + e^{-2x}} - 1$$

- **ReLU**: The Rectified Linear Unit (ReLU) activation function clamps negative values to 0 and output positive values as is. There are slightly altered versions like the leaky or parametric ReLU, which do not clamp negative values strictly to 0 and provide a slope for negative values as well.

$$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$$

Which activation function to be used depends on the task at hand. For certain tasks which have been researched extensively already, there are best practice activation functions that work well. For less researched subjects, ReLU usually is a good starting from. It provides a good behavior for training, which I will discuss later in section 4.2.3.

This basic concept can be extended by adding more components to it, for example to create a recurrent, memory or kernel unit [24]. Recurrent and memory units are self explanatory, they either receive their own output as input for one or more iterations or store a number of values from past executions. The kernel unit will be interesting in context of tracking, as it applies an n-dimensional filter to the input like traditional filtering done for image processing. It is one of the main building blocks for Convolutional Neural Networks.

4.2.2 Structure of a Network

A Neural Network is composed of many connected neurons, which are organized in layers. It has one input and one output layer, as well as a number of hidden layers. The **input layer** is connected to the inputs and the number of neurons it contains corresponds to the number of features of the sample. The **output layer** can have any number of outputs, it usually is a lot smaller than the input layer and every neuron of it returns a desired feature value. The number and size of **hidden layers** can vary greatly, the same goes for their behavior, which calculations they perform and how they are interconnected. An example for a Neural Network is provided in figure 5

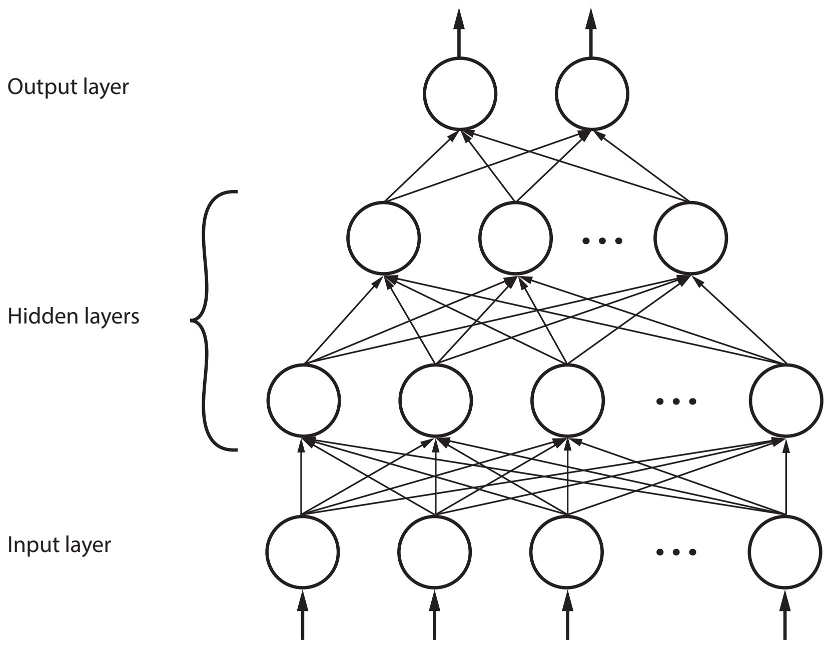


Figure 5: Example for a Neural Network

The neurons on a layer are not interconnected, but they connect to neurons of the previous and next layer, with the exception of the input and output layers, which only connect to the first or last hidden layer respectively. In the example a Neural Network is shown which consists exclusively of fully connected layers. In a fully connected layer, every neuron of a layer is connected to every neuron of the other layers.

There are many different architectures [24], the simplest one is a Feed Forward Network. It is composed of an input and an output layer and has just one hidden layer in between. The neurons are fully connected. This can be extended to a Deep Feed Forward Network, the only difference being there can be more than one hidden layer. Among other network architectures which can also utilize recurrent and memory units, the most important ones for this thesis will be Convolutional Neural Networks, since they are used in image processing. They will be discussed later in section 4.2.4.

4.2.3 Training a Neural Network

Training a Neural Network is done in several steps. Once a network has been modeled, its parameters are initialized with random values. Then training data is fed into it, calculating a (most likely incorrect) output. For this step, the neurons of each layer process their weighted inputs, apply the activation function and output the result to the next layer. this process is repeated for every layer until the output layer is reached. A **cost function** is used to compute the error between the computed and expected values. Commonly used cost functions are Sum of Squared Differences, Cross-Entropy and Exponential cost.

Once the training data is processed and the error calculated, the actual training of the network can take place. Most commonly a process called **Backpropagation** is used to accomplish this. The error value is processed by the network in reverse, starting at the output and ending at the input layer. At every node, its calculations are performed in reverse and the gradient between input and output values is determined. A part of this gradient, depending on the **learning rate**, is added to the input weight to incorporate the error. The goal is to improve the input weights so future computations will be closer to the desired result. Since a neuron can have multiple inputs, this process has to be repeated for each incoming connection until all weights have been adjusted.

The activation function chosen has a big impact on the learning process as it will impact the gradient. Learning progress can only be made when the gradient is unequal to zero, as zero means there is no change in value. Considering the Sigmoid function, it has high gradients around the origin, but their value decreases quickly towards negative or positive infinity. This means for larger values, be it positive or negative, when computing the gradient it will result in a zero or near zero value. A percentage of this value, the learning rate, defines how much of the error will influence the weights of the current neuron. If the gradient is zero or near zero, the influence of the error is minimal to none on this neuron. Especially when using tanh and sigmoid activation functions the learning process can get stuck because of this. The ReLU activation function improves on this behavior by passing on positive values and setting negative values to zero. However, for negative values there is the same problem. For this reason there are variants of the ReLU, like leaky ReLU, which allow for a small amount to pass through, even if it is a negative value. This prevents the training process from getting stuck because even if small, the gradient can never be zero.

To train a Neural Network a big amount of data is necessary. Those training datasets usually contain tens to hundreds of thousands of samples. This allows for a wide variety of input data to cover as many scenarios as possible. Since the process of Backpropagation is computationally expensive, the training is usually done in batches. Batch sizes vary on network size, architecture and the problem at hand, common sizes are around 50 to 100 samples. The output of the Neural Network is computed for all samples in the batch, the resulting error averaged and then a Backpropagation pass done.

4.2.4 Convolutional Neural Networks

Convolutional Neural Networks (CNN) are designed with image processing in mind. They combine convolution layers, which apply kernels to the input vectors in the same way traditional image processing does, and fully connected layers to solve tasks like object recognition. The convolution layers will do feature detection, like corners and edges, the fully connected layers interpret those features, for example do an image classification. What makes CNN really stand apart from traditional image recognition algorithms is, that not only the fully connected layers are trained, but the convolution layers as well. Hence they will adjust automatically to fit the task at hand, which makes CNN a powerful tool.

The feature detection part of a CNN usually is comprised of three types of operations: convolution layers, pooling layers and the activation function. Unlike Neural Networks which use fully connected layers and usually work on one dimensional vectors, they work on surfaces or volumes. Since in this thesis the input always is an image, the highest dimension a volume can have is 4, which either means a multi channel or single 2D image.

When setting up convolution layers, instead of defining the number of neurons of a layer, the filter size and number of filters is specified. The number of neurons is automatically determined by the size of the input image. Each neuron is only connected to the inputs which are close enough to be inside the specified filter size as well as every corresponding depth layer of the input (figure 6, right). A convolution is performed on this window which results in a single value for the part of the image the filter covers. One filter results in a 2D output vector (figure 6, left).

For convolution layers two additional considerations have to be made, the treatment of borders and sharing of parameters. The **treatment of borders** is done using **zero-padding**, a process which adds inputs around the image borders with a value of zero. There are three cases of zero-padding worth mentioning. One is adding no padding at all, a kernel with size k will only be applied to positions which are fully within the image dimensions. This results in the image shrinking in size by $k-1$ pixels whenever

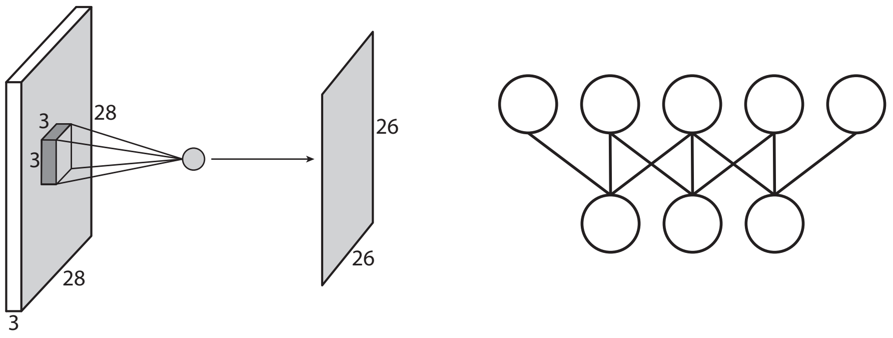


Figure 6: A single convolution layer. Left: Schematic of how a filter is applied to a 28x28 pixel sized image with 3 channels. Right: Neuron connections to the input for a single channel image with a filter size of 3.

a convolution layer is applied. This is called a **valid convolution**. The second setting adds $(k-1)/2$ pixels around the border, the center of the kernel will visit every pixel of the image and the output will be of the same size. This is called **same convolution**. The last case adds a zero padding of $k-1$ pixels, which means the center of the kernel will leave the image to the extend the filter window and image overlap by just one row of pixels. This results in the image growing by $k-1$ pixels every time a convolution layer is applied and is called **full convolution**. An illustration of these cases can be found in figure 7. Usually either valid or same convolution is used.

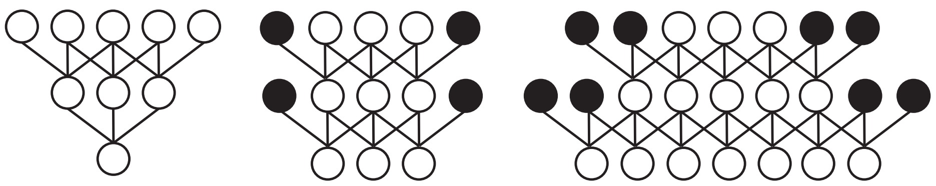


Figure 7: Three cases of zero-padding (added padding in black). Left: Valid, the image shrinks. Middle: Same, the image size remains constant. Right: Full, the image size grows.

In fully connected layers, their parameters are usually stored locally for each neuron. In convolution layers it is possible to share them between kernels. This results in less memory usage. There are several options for parameter sharing. In **unshared convolution**, no sharing is used at all. No matter at which image position a kernel is applied, it has its own local weights that get trained and used for calculation later. This is useful when it is known that certain features will only appear in certain areas of the image, for example when doing face recognition, but needs a lot of memory. **Traditional convolution** is the opposite of unshared convolution, as it shares the same parameters across all kernel applications. This approach will recognize the same feature no matter where it occurs in the image and uses very little memory. Finally **tiled convolution** combines both previous approaches. A set of kernels is defined which are rotated through as the filter moves through space. Immediately neighboring locations will have different filters, but the memory requirements are a lot less than for a locally connected layer. A comparison of all methods of parameter sharing can be found in figure 8.

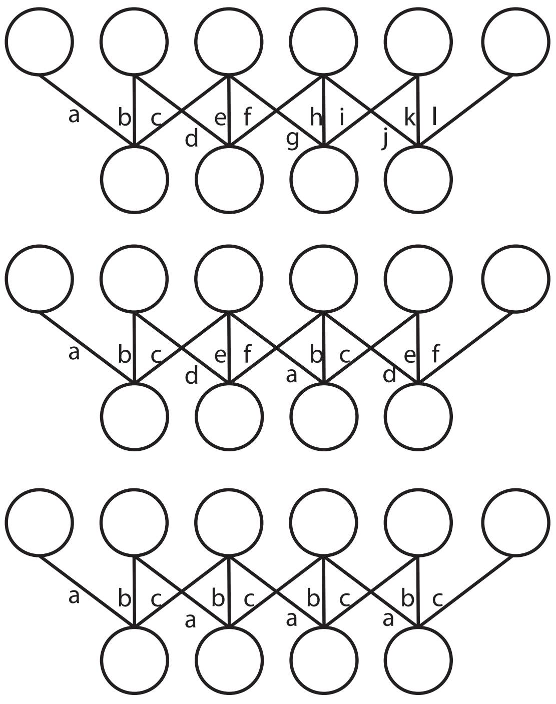


Figure 8: Parameter sharing options between kernels. Top: Fully locally connected, no sharing. Middle: Tiled sharing. Bottom: Traditional convolution

Pooling layers have the purpose to reduce the output size, hence simplifying it. This reduces the number of parameters that need to be learned. A common method of doing this is max pooling. A pooling window size is defined, for example 2x2. Furthermore a stride parameter can be added, which determines how fast the window moves across the input. This window is then moved across the input vector with the defined step size and the maximum value in it is taken as output for this layer. An example for this operation can be found in figure 9.

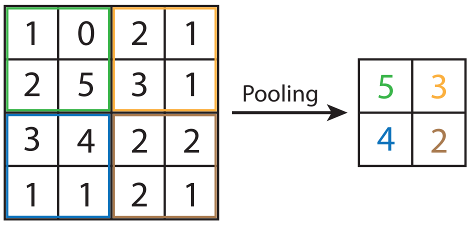


Figure 9: A pooling operation with window size of 2x2 and step size of 2.

The final step of the feature detection is applying an activation function. This works exactly like already discussed in 4.2.1. When for example using the ReLU activation function, this removes all negative outputs and leaves positive ones untouched, which is ideal for image processing. Depending on the activation function, the two steps of pooling and activation function can be swapped in their order.

The second part of a CNN, the feature interpretation, is done by a fully connected network. To be able to use the output of convolution layers in a fully connected layer, it has to be flattened first. This operation converts the output of the previous convolution layers into a 1D vector, which can be fed into a fully connected network. This part works exactly as discussed earlier in this chapter. An example for a complete CNN can be found in figure 10.

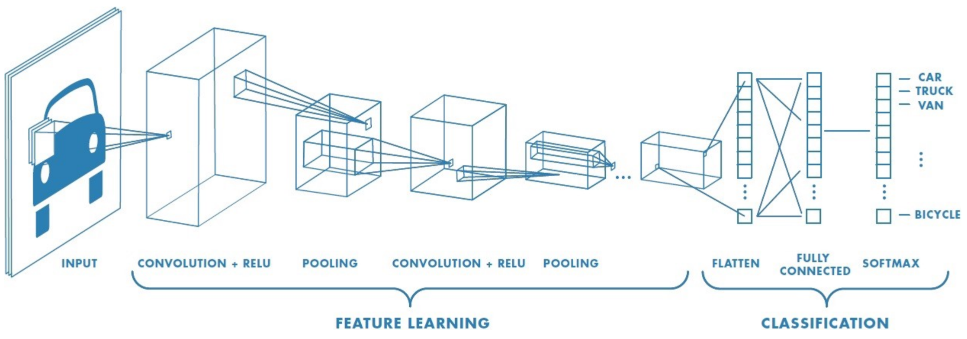


Figure 10: Example for a complete Convolutional Neural Network [25]

5 The Object Tracker

In this chapter the object tracker is presented. The project is divided into three separate applications, which will be discussed in the following order: Up first is the generation of training data to train the Neural Network that is responsible for the tracking. After this I will explain the training process and finally the object tracker application itself.

5.1 Generating Training Data

The process of generating the training data has been adopted from the one Garon and Lalonde use in their Deep 6-DoF tracker [18]. The application generates samples consisting of three files each. An image of the current (observed) position, an image of the previous position and a binary which contains the transformation vector which was applied to get from previous to current position. While at first only training data for object translation in x and y direction is necessary, the application supports generation of data for full six degrees of freedom.

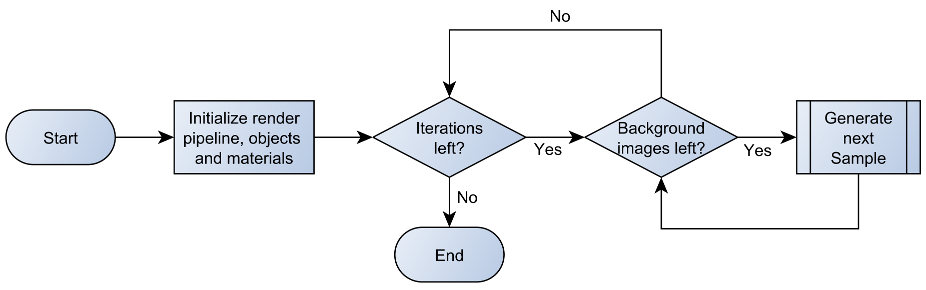


Figure 11: Flowchart: Generate training data - main loop

Figure 11 shows the main loop of the application. As a first step the OpenGL render pipeline is initialized. Then the 3D objects, materials as well as the shaders are loaded and prepared for use. The sample creation is done in a double loop. The inner loop iterates over a directory in which the background images are stored, generating a sample for every image in that directory. The outer loop makes it possible to re-use the same background images several times in case not enough background images are available. It is important that for every background image its depth data is available. Instead of background images a full 3D scene can be used as well. In this case just one loop is necessary, calling the sample generation routine the desired amount of times.

The image generation is fairly linear with a few options that can extend it, shown in Figure 12. First, if RGB-D background images are used, those

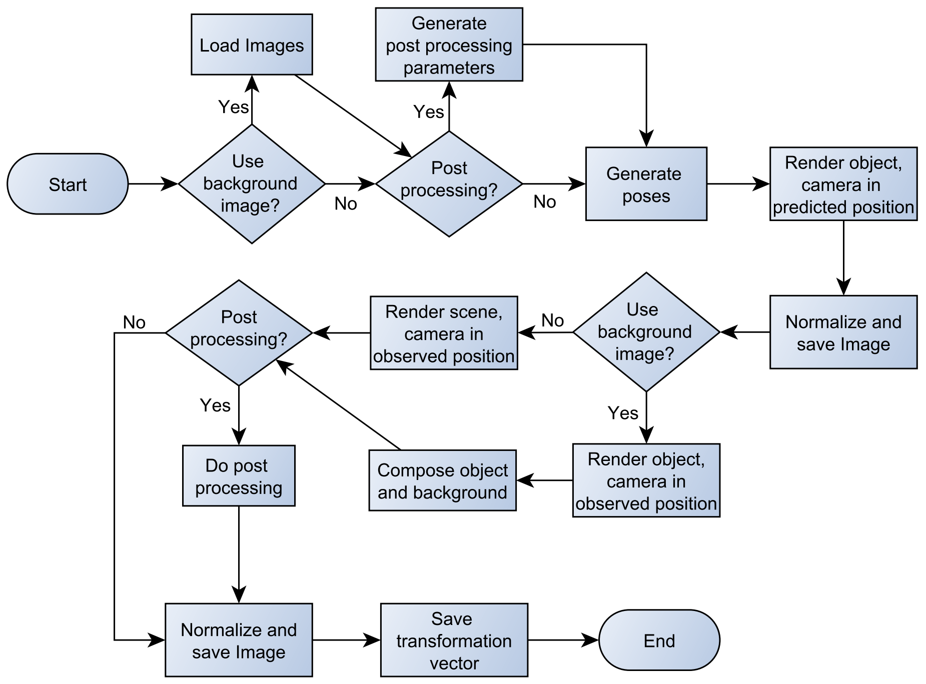

Load Images
Generate post processing parameters
Yes
Yes
Start
Use background image?
No
Post processing?
No
Generate poses
Render object, camera in predicted position
Normalize and save Image
Use background image?
No
Render scene, camera in observed position
Post processing?
No
Yes
Yes
Do post processing
Compose object and background
Render object, camera in observed position
Normalize and save Image
Save transformation vector
End

Figure 12: Flowchart: Generate training data - sample generation

are loaded. In case of the SUN RGB-D dataset [21], the information is distributed over two images. One contains the RGB image, the other holds the depth information. Both images are combined to a single RGB-D image later in this pipeline using a shader.

It is possible to perform postprocessing on the image representing the current position of the object, also called observed image. In the final tracker, this image can be provided by different sources. In my case it will be an in application rendered image sequence to keep it simple for this thesis. Since all images are generated, there is no postprocessing necessary. However, it is also possible to provide the tracker with a recorded image sequence or real time video feed. Since the sensors and lenses of cameras can not produce a perfect representation of the real world, it is necessary to consider these factors when generating the samples so the Neural Network can be trained with those impurities. In the postprocessing stage, the image can be enhanced with blur, color shift and noise. Their values can be changed, the default ones are taken from [18]. Blur will be a 3×3 mean filter. The color shift is applied in HSV color space to the hue and luminosity channels with a random value of $h \sim U(-.05, .05)$, where U is a uniform distribution. The noise will use a gaussian distribution $N(0, \sigma)$ with $\sigma \sim U(0, 2)$. A noise texture will be generated which will be used by the postprocessing shader. The blur is applied to the RGB and depth channel, the color shift and noise only to the RGB channels.

Next the poses for both images are generated. They represent the actual pose found in the current input image, also called *observed pose* $\mathbf{p}_{obs}$, as well as the pose which was predicted during the last iteration of the tracker, called *predicted pose* $\mathbf{p}_{pred}$.

The object to track needs to be placed in the center of the world coordinate system and the camera is placed on a random pose around it. To do this a random position (θ, ϕ) on a sphere surrounding the object is sampled. The sampler uses a uniform distribution U, which results in $\theta \sim U(-180°, 180°)$ and $\phi = \cos^{-}1(2x - 1)$ with $x \sim U(0, 1)$. The sphere radius, which represents the distance between camera and object, is sampled the same way, using a uniform range which represents the use case for the application best. Having the distance and sphere coordinates, the camera position and rotation can be calculated. Additionally, a camera roll angle is sampled with $\gamma \sim U(-180°, 180°)$. All of these values are now used to determine the camera matrix for the observed camera pose $\mathbf{p}_{obs}$.

To obtain the previous camera pose, a random displacement vector is sampled. For a full 6-DoF tracker it contains a small translation $t_{x,y,z}$, uniformly sampled for a range appropriate for the current application, and a rotation $r_{x,y,z}$ which is uniformly sampled between $-10°$ and $10°$ for every rotation angle. The final displacement vector will be $\mathbf{d} = [t_x, t_y, t_z, r_x, r_y, r_z]$. The limited version of the tracker, which at first tracks only x and y direction, will use only that part of the displacement vector, resulting in $\mathbf{d} = [t_x, t_y]$. The *inverse* of this displacement vector is now applied to $\mathbf{p}_{obs}$ to obtain $\mathbf{p}_{pred}$.

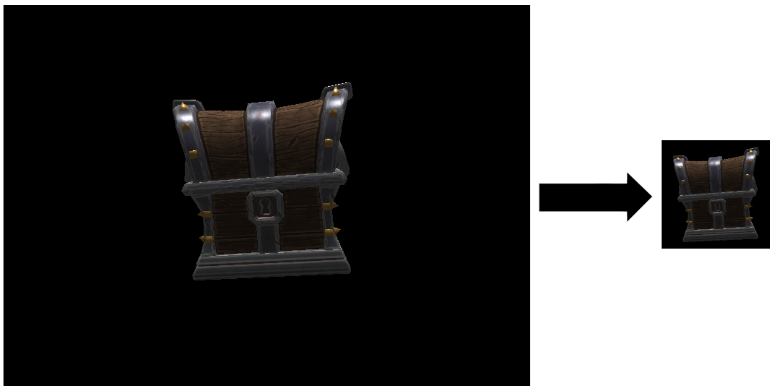


Figure 13: Predicted image generation steps. Rendering the object followed by normalization.

Now the *predicted image* i_{pred} is rendered using a textured version of the objected to be tracked. In addition to a diffuse color texture an ambient occlusion texture is used as well. The object is placed at the center of the world coordinate system and the camera at the previously obtained p_{pred}.

Two light sources are used, one directional white light and one ambient light. The directional light always points downward with regard to the camera viewing direction. The background is black, which is initialized when clearing the frame buffers. The bounding box for the object in the image is determined. To allow for the object movement with respect to the observed image, its size is increased by 15%. The image is cropped to the size of the bounding box and resized to fit a 150 $\times$ 150 pixel image size. Additionally, the depth pixels are shifted. For this the mean depth value of the object is computed, the difference between that mean and 128 is calculated and the result is added to the depth values of the image. This step ensures that, no matter how close or far the object is from the camera, the input images for the Neural Network will always have the same starting depth, resulting in the same relative depth movement. The image is then saved. The depth image is stored in the alpha channel of the image to eliminate the need of saving two separate images. Figure 13 shows the output of the steps.

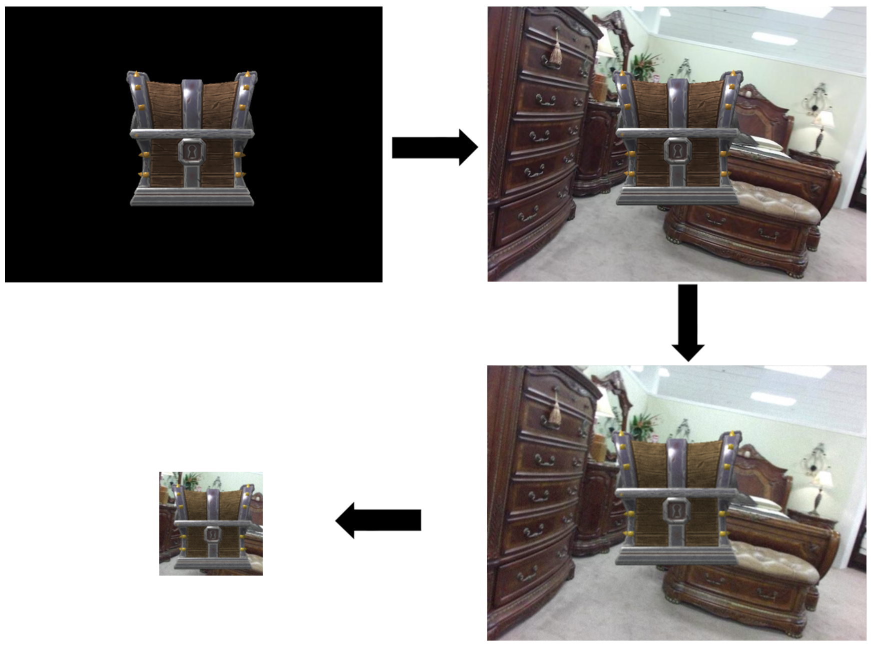


Figure 14: Observed image generation steps. Rendering the object, composition onto background, postprocessing and normalizing.

Next the *observed image* i_{obs} is created. The camera is positioned at p_{obs}. In case a complete virtual scene is used, it is rendered with the object. The image only needs to be cropped to the bounding box obtained in the previous step, resized, normalized in depth and saved. If a background images is used, the object is rendered in the same manner it was done for i_{pred}. Instead of pointing downwards, the light source direction is sampled uniformly on a sphere using the same process explained earlier to determine the camera position. Afterwards two additional render passes are applied. First the object is composed onto the background image. Then the postprocessing can be applied to the result. The image is also finished by cropping it to the bounding box, resizing, normalizing the depth and saving it. Figure 14 shows the output of each step and figure 15 shows the composed depth image.

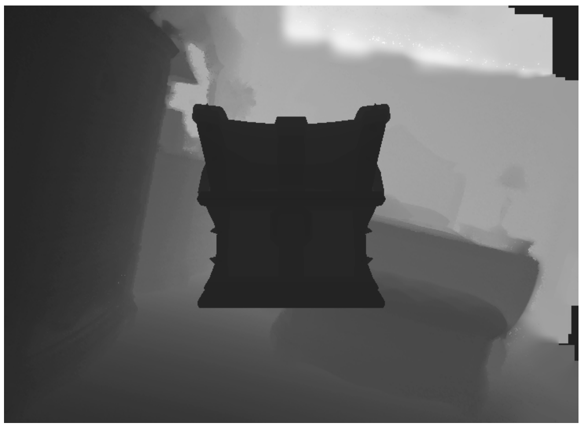


Figure 15: Depth for the observed image.

As a last step, the displacement vector **d** is saved as a binary file, which concludes the sample generation.

5.2 Training the Neural Network

This part of the project is about setting up and training the Neural Network which will be used later on to do the tracking. This is done using PyTorch as discussed earlier. A n-dimensional vector is also called a *tensor* in PyTorch, hence I will use both terms interchangeably. The initial network is structured the same way Garon and Lalonde have defined [18]. The structure of it is visualized in figure 16.

As discussed earlier, a Convolutional Neural Network (CNN) is used for tracking. The network takes two inputs, the predicted image which is generated from the previously estimated pose and the observed image, which is the current video frame. The images are normalized, using the method described before in section 5.1. Both are passed through a convolution layer, which is configured to have 24 filters with a 5x5 window size. Then a max pool operation is performed, which reduces the image size to half and an ELU activation function is applied. The ELU activation function is defined as follows [26]:

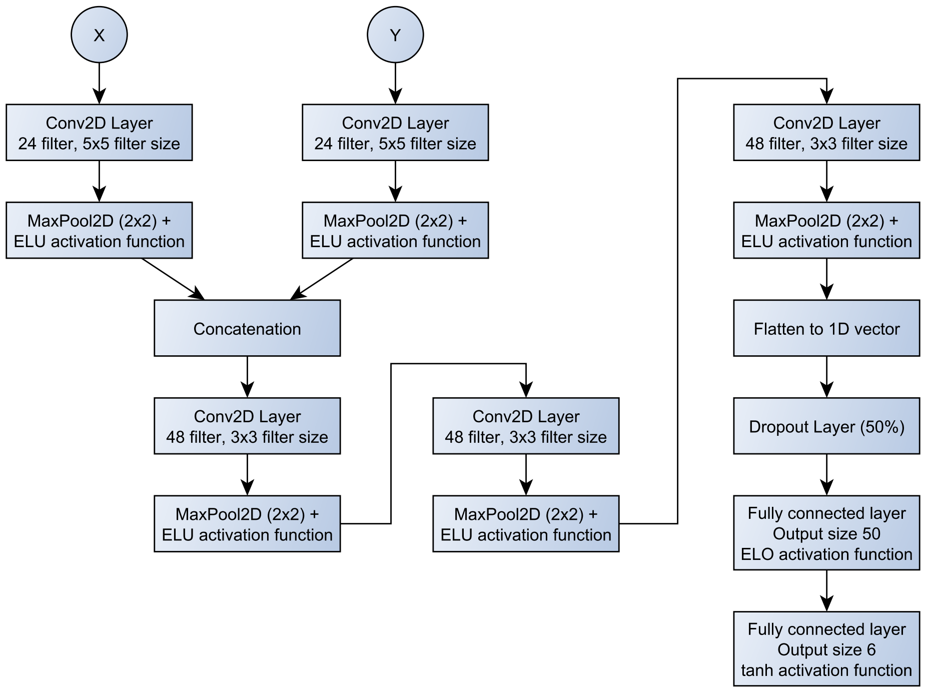


Figure 16: The structure of the Convolutional Neural Network.

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha(\exp(x) - 1) & \text{if } x \leq 0 \end{cases}$$

While the already discussed ReLU activation function clamps negative values to a minimum of 0, the ELU activation function saturates at a minimum of -1. The advantage of ELU over ReLU is that values are not just clamped, but they converge against -1 in negative direction. This ensures that the learning process will not get stuck for negative values (the gradient is 0 for all negative values in ReLU).

The separate layers are then concatenated and will get processed at the same time by every layer coming up. The combined image is put through three more convolution layers, all using 48 filters with a size of 3×3, each followed by another max pool and ELU activation operation. The result is a three dimensional vector of size 7 * 16 * 48 (image height * image width * channels) which has to be flattened for further processing. This will convert it into a one dimensional vector of length 5,376 which can be processed by fully connected layers.

As the next step a dropout layer with a dropout rate of 50% is added. A dropout layer is only active during the training process. A given number of inputs, in this case half of them, will be randomly selected for every execution of the network. These inputs will not be passed on to the next layer following the dropout layer. This helps to prevent overfitting of the network.

Two fully connected layers form the end of the CNN. The first will reduce the input to an output size of 50 and uses the ELU activation function. The second reduces these inputs to the network output size, which is 6 for a full 6-DoF tracker and 2 for the simpler version tracking only two directions. It is the only layer in the network which uses not the ELU but a tanh activation function.

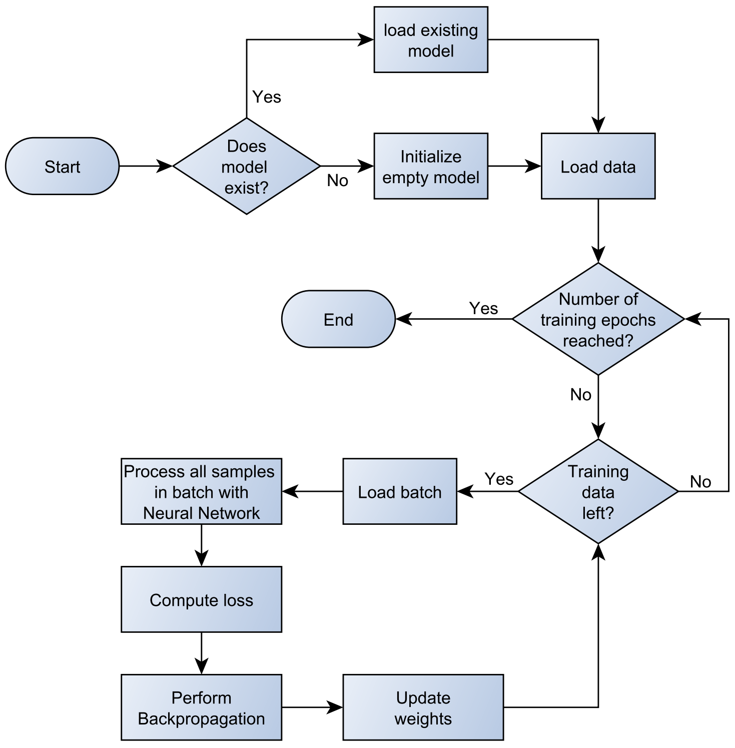


Figure 17: Flowchart: Training the Convolutional Neural Network

The actual training process is shown in figure 17. First the CNN is prepared. In case an already trained model is found it is loaded and used for further training. If not a new model is created and the weights initialized with random values. Then it is transferred to the GPU. The next step is to load the training data. The samples are read from a given directory. While the binary displacement vector can be moved to a PyTorch tensor without any additional processing, the images which are read using OpenCV have to be reformatted. The OpenCV Mat class into which the images are loaded stores them in a per pixel fashion. However, the PyTorch tensor requires a per channel storage. For this reason the image has to be split into its channels and then concatenated correctly before moving it to the tensor. Once this process is done the training data is moved to the GPU as well.

Now the training process starts. The training data is processed in batches with a batch size of 64. This means 64 samples are loaded and processed by the Neural Network before adjusting its weights. The average loss for all 64 samples is computed using the mean squared error. For this average the backpropagation is conducted, which computes the gradient for the loss at all neurons in the network. Doing this for a batch of samples instead of every sample decreases the computation time significantly, but it comes at the cost of accuracy. This loss is usually compensated for by using a high number of samples as well as using them several times during a training process. Finally the weights are updated. This done using an Adam optimizer.

Processing all samples of a dataset once is called an epoch. The whole training process is now repeated several times until a defined number of epochs is reached. As mentioned employing several epochs compensates for using batches and averaging the results. This is possible because the data loaders provide the batches with randomized samples.

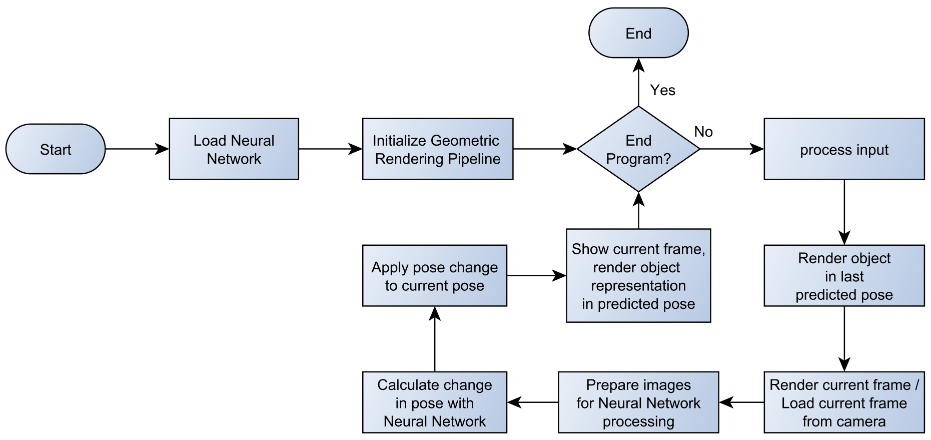
After the training is concluded the trained model is saved. It can either be further refined by training it with other datasets or used in the tracker application.

5.3 The Object Tracking Application

The tracking application combines both main elements of the previous applications, the geometric rendering pipeline and the Neural Network. It will load and generate the data required to feed to the Neural Network and then processes it to acquire the change in pose. Figure 18 provides an overview of the tracking application.

First the core elements of the application are prepared before entering the main loop. This step consists of initializing the Neural Network as well as the geometric rendering pipeline. The Neural Network has to be defined to match the structure of the one that was trained. Then an object of this Network is created and the trained parameters are loaded from the file they were saved to. The initialization of the Neural Network is finalized by transferring it onto the GPU if CUDA is available, otherwise the CPU is used for calculations. Afterwards all components of the geometric rendering pipeline are initialized, like creating the output window, loading shaders, objects and materials and creating cameras, lights and frame buffer objects (FBOs). Since this is a temporal tracker it is also important to initialize the predicted pose at this point, so the tracker has a pose to start with.

The main loop begins with creating the inputs for the Neural Network. First the predicted image is rendered. Similar to generating the training



End
Yes
Start
Load Neural Network
Initialize Geometric Rendering Pipeline
End Program?
No
process input
Apply pose change to current pose
Show current frame, render object representation in predicted pose
Render object in last predicted pose
Calculate change in pose with Neural Network
Prepare images for Neural Network processing
Render current frame / Load current frame from camera

Figure 18: Flowchart: The object tracker

data this means to render just the tracked object using the predicted pose. The result is stored in a FBO for later use. Next the current frame is created. In the scope of this thesis it is generated on the fly by loading a RGB-D image from the SUN RGB-D dataset [21] and rendering the object on top of it using the current pose. This step can easily be replaced by loading the data from any other source, like a video, image sequence or camera input. Both images are then processed to create the actual input required to feed the Neural Network. The bounding box for the object is determined and both images cropped to its size. The depth values of both images are normalized using the same process described in 5.1.

After preparing the images, they are converted to tensors and uploaded to the GPU. The Neural Network is then executed using those tensors as inputs. It returns a single tensor which represents the displacement vector. It needs to be transferred back to the CPU before the values can be read. The transformations are then applied to the previous predicted pose to obtain the new one.

Finally the output is rendered, which is a two step process. First the unchanged version of the previously rendered or loaded current frame is drawn using a screen filling quad. Then the object is optionally rendered on top of this image, using the new predicted pose and a simple, single color shader. Both outputs are shown in figure 19.

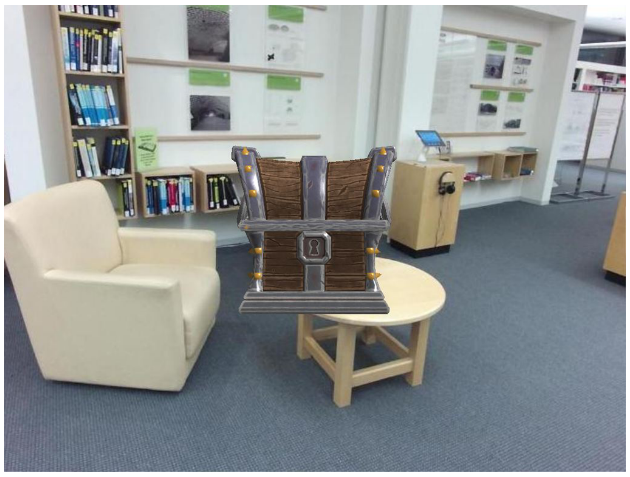
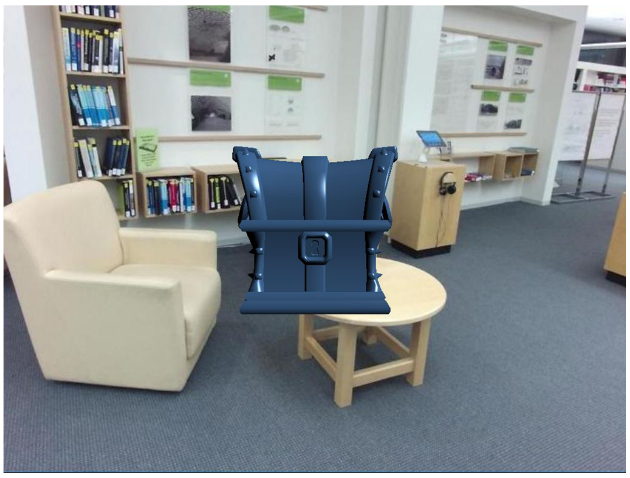



Figure 19: Top: Unchanged output of the current video frame. Bottom: Output of the current video frame with the object rendered on top of it using the last predicted pose.

6 Analysis and Evaluation

In this section I will investigate various aspects in the context of this object tracker and its usage of Neural Networks. First there will be a general analysis of the applications the way they were developed, focusing on the quality of the results as well as the performance of the different parts. After this I will have a look at how changes in network architecture affect training times and the tracking results and performance. The section will be concluded by checking the importance of the depth data included in the Neural Network input.

The system used for training the Neural Network and running the tracker is a Intel Core i7 4790k with 16GB RAM and a nVidia GeForce GTX 1080Ti GPU with 11GB memory. Training the network and processing the data while tracking is both done on the GPU as this results in significantly increased computation speed.

6.1 Tracker Performance Evaluation

The tracker has been built to support up to six degrees of freedom, three translation and three rotation axes. The only adjustment necessary to switch between different degrees of freedom is to change the number of outputs the Neural Network has. Of course the training data has to be generated accordingly and in the tracking application all outputs have to be applied to the predicted pose. Evaluation of the tracking was done in several steps, starting with tracking x and y translations to the pose (hereafter referred to as 2-DoF tracking), then tracking of all 3D translation axes (3-DoF tracking) and finally full 3D tracking with all translation and rotation axes (6-DoF tracking).

For training the 2-DoF tracking network 50,000 training samples were generated. In each sample the change in pose is a translation along the x and y axes by a random value between -0.2 and 0.2 units. The training was done in three iterations, each training the Neural Network using all 50,000 samples for a total of 30 epochs. Considering the amount of images and their size, it would not be possible to keep all of them stored in the memory of the graphics card. Hence the training data was split into sets of 10,000. This resulted in the diagram showing 50 epochs per iteration, which corresponds to 10 epochs for each set. With each iteration the learning rate of the Neural Network was decreased to refine the learning process, starting at a rate of 0.005 and ending at 5E-5. After each iteration, the trained network - also called model - was tested inside the tracking application to check the tracking performance. The total training duration for all epochs was about 57 minutes.

After the first iteration the object tracking was already working. When rendering the object on top of the input frame using the predicted pose,

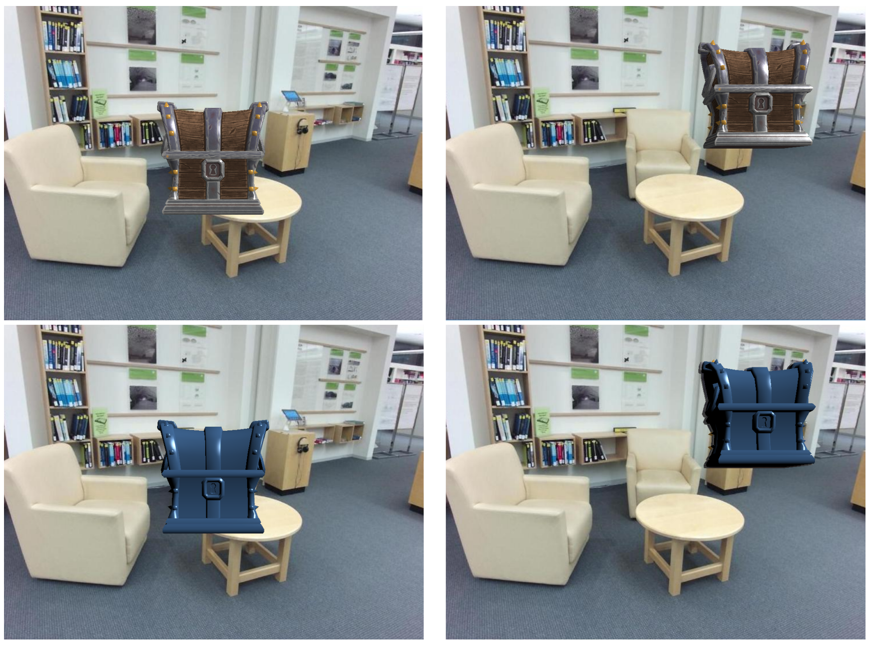


Figure 20: The upper images show the input frames, the lower ones show the object in its predicted pose rendered on top. Left side: The object in its starting position. Right side: The object was moved.

it remained centered on the object as it was displayed on the input frame (figure 20). However, when the pose was not changing there was a noticeable amount of jitter, indicating that more training was necessary. The next iteration resulted in a visible decrease of the jitter to a very subtle level, but was not able to eliminate it completely. The last iteration did have no further visible changes. Further training might decrease the jitter more, but given the nature of Neural Networks and the low floating point numbers that cause this problem it seems unlikely to disappear completely. It can be solved by introducing an epsilon value which ignores small values in pose changes around 0. This results in a stable resting position with the tracking still working.

The graph in figure 21 shows the change in error value over the time of training and confirms the observations. During the first iteration (epochs 1-50) the error value decreases rapidly, but has not reached its lowest point yet. The graph smoothes out over the second and third iteration (epochs 51-150) and reaches a constantly low error value.

Overall, the tracking limited to x and y axes works very well. The digital representation rendered on top of the input frame sticks to the middle of the object. Even rapid movement of the camera / object does not result in losing the tracked object.

In the next step the z axis was added to the Neural Network to enable

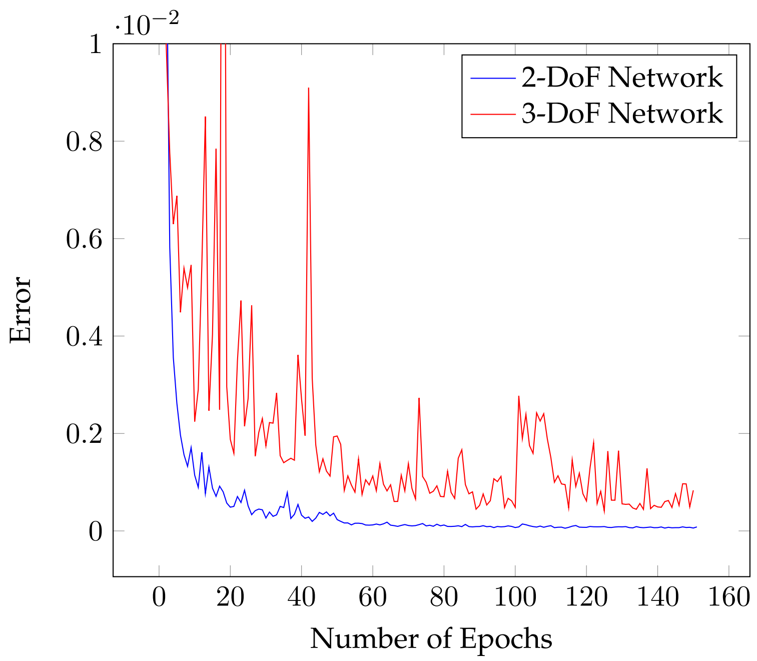


Figure 21: Error rate over time during training for 2 and 3-DoF Networks.

3-DoF tracking. As already stated, this only required to change the number of outputs of the Neural Network from two to three. The training process was identical to the 2-DoF network, using the same amount of samples, iterations and epochs. Like when training the 2-DoF network, after every iteration the performance of the network was checked. The training process took about 58 minutes and therefore just one minute longer than the 2-DoF network. This may be connected to the additional output, but since the difference is very small there is also the possibility that it is just a variance.

The first attempt of training the network did not have any normalization of the depth data yet. While the translation along the x and y axes still was working fine, there was no change in pose when translating along the z axis. After adding depth normalization, which shifts the depth values to the center of their range of values, the translation along the z axis was working too. This normalization process ensures a relative change of depth values similar to the normalization done when cropping the input images to the bounding box of the object. With the depth normalization in place the results are similar to the 2-DoF network. After the first iteration significant jitter is visible, decreasing to be more subtle with the second and third iteration. Figure 22 shows the output of the tracker with the object now being translated along all three axes.

Like for the 2-DoF network, the graph in figure 21 shows the change in error value over the time of training. It is by far not as smooth and takes

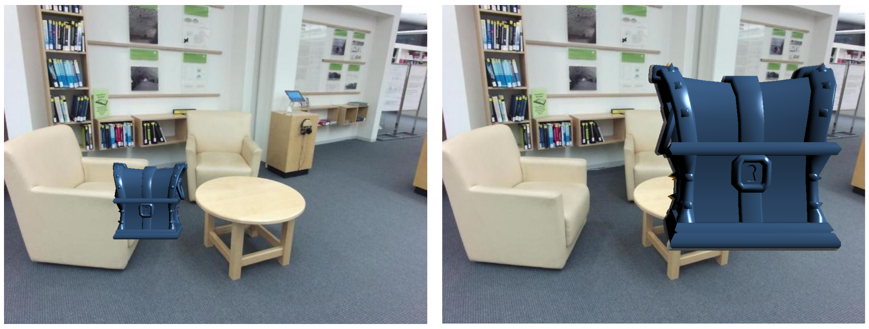


Figure 22: The tracker showing the object in two different positions, the predicted pose rendered on top of the input frame.

longer to get to a low error value, but this had no visible influence on the object tracking task. The graph also shows that the baseline of the error value is higher in general and most likely will remain so even if it would smooth out completely. This is to be expected, since the mean squared error function is used to compute the value. Adding another value, even when very small, will increase the total unless it reaches 0. Looking at the error value in isolation it seems to indicate more training is necessary to bring it to the same smoothness the 2-DoF network achieved. However, considering the performance in the tracking application the Neural Network has been trained sufficiently.

While the tracking itself is working, there are two problems when it comes to extreme poses. When the object is very close to the camera a lot of jitter appears. In the opposite case, when the object is far away from the camera, it has increased jitter at first and with further increasing the distance the tracking is lost and it starts to drift off. In the first case the object may be too close to the camera. This could result in the bounding box, which is even larger than the object, leaving the bounds of the image, causing undefined areas. These undefined areas could cause the problems when trying to estimate the pose with the Neural Network. The problem might be solved when limiting the bounding box to the image size even when parts of the object are cut off and adjusting the training data accordingly to include this situation. However when the object gets too close to the camera too many details of the shape and depth data will get lost, which will impose a limit on how close objects can be tracked either way.

As for the second case, neither the size of the bounding box nor the lower resolution shape should have a significant impact on the tracking performance since both parameters are normalized. It is more likely that the trained differences in depth between the shape of the object and the background get too small to be registered anymore. Similar to the closeup case, increasing the amount of samples by including far away positions may increase the distance an object can be tracked at. But this will have a limit as well, a point at which the depth values of the object and background are too similar to gain any meaningful information. This also might give a hint at the importance of the depth data for the tracking process in general, which is discussed later in section 6.3.2.

Aside from the two cases mentioned the 3-DoF network performs the tracking task just as well as the 2-DoF network. The pose sticks to the object and rapid movement is no problem.

Finally the network was adjusted for 6-DoF tracking by increasing the amount of outputs to 6. Since these additional dimensions add a lot of possibilities to the pose changes, the number of training samples was increased to 150,000. The training rate is kept at the same range and decreases from 0.005 to 5E-5. The net was trained for about four hours. The graph in figure 23 shows the development of the error during the training process.

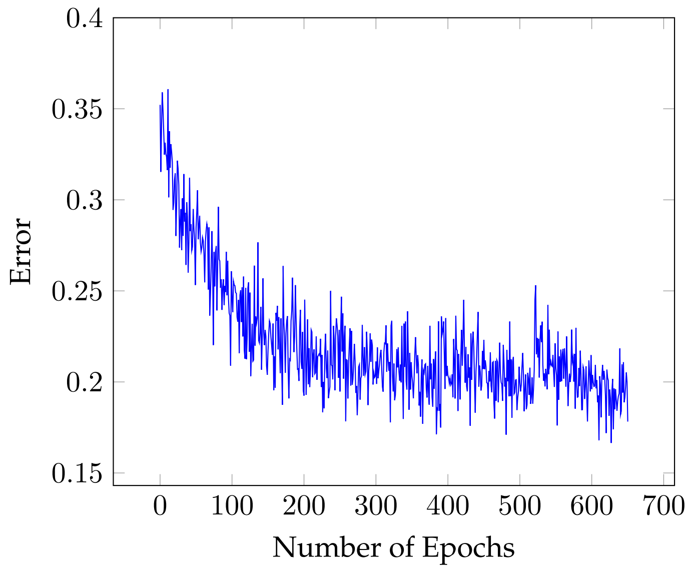


Figure 23: Error rate over time during training for the 6-DoF Network.

The error has decreased noticeably, but the curve does not get smooth. It is also quite a bit higher than the lower degrees of freedom nets. The lowest error value reached is about 0.1679, but jumps up to over 0.2 regularly even at the end of training. Compared to that the 3-DoF network stays constantly below an error value of 0.001 and the 2-Dof is even lower at less than 0.0001, which is a very big difference, even given the additional outputs.

Looking at the error value it is to be expected that the tracker will not work properly, and this is confirmed when using the network in the tracking application. When applying only the translation output of the network to the predicted pose the tracking still roughly works, but it has a lot of jittering with very big jumps. When adding the rotation output to the new predicted pose the object starts to randomly turn and when moving the object tracking is lost completely.

Identifying the problem is not an easy task, given it could be situated in any of the applications and training a Neural Network is a blackbox process. It would need more extensive research and gathering experience to solve, for which the amount of time available to write this thesis is not enough. While I tried to stay true to the information given by [18], not all required details are provided, which adds to the difficulty. For this reason I will forgo trying to get the 6-DoF tracking to work in favor of analyzing Neural Networks for tracking more thoroughly by using the 2-Dof and 3-DoF networks.

To close this section I have a short look at the performance during tracking. Rendering the predicted pose with the geometric render pipeline takes about 0.1ms, which makes it barely a factor considering the the time needed by the rest of the application. However, copying the image from GPU to CPU, calculating the bounding box and performing the postprocessing (depth shift, cropping) takes 10ms which is a lot in comparison. But since rendering the object in its predicted pose is a mandatory step of this algorithm this has to be done and cannot be further optimized. In my case the observed image is not loaded from an image sequence but generated during runtime too. While the rendering process itself does not add much with 0.15ms, copying and postprocessing the image needs another 7ms. When using a pre-rendered image sequence this could be reduced by about 6ms, since the processing only takes about 1ms with the GPU to CPU transfer taking up most of the time. For the last part, predicting the pose by processing the input images with the Neural Network and applying the resulting displacement vector to the predicted pose takes about 6ms. This process also has no further room for optimization. Changing the number of outputs for different degrees of freedom does not affect the computation time by a significant degree, it stays at around 6ms. This was expected since the majority of the network did not change. Adding the times up it takes about 23ms every frame to perform these operations. Some general operations needed to complete a 3D rendering application will add another 2ms totaling in 25ms for a fully rendered frame, which will result in a steady rate of about 40fps.

6.2 Impact of the Amount of Training Data

To check the impact the amount of data used for training has on the performance of the Neural Network, the same 2-DoF network was trained with a different number of samples for the same amount of epochs. Figure 24 shows two graphs which picture the development of the error rate over time. The graph was split into two for clarity and the 50,000 sample error rate was provided in both for reference.

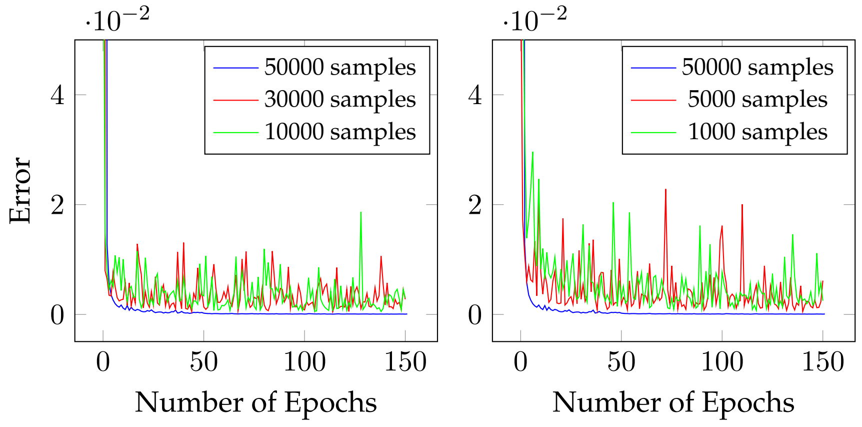


Figure 24: Error rate over time during training for a 2-DoF network with different amount of training samples. Both graphs include the 50000 samples for comparison.

The error rate still decreases over training, but it does not get as low anymore as before. Also it is noticeable that with decreasing the amount of samples, the error rate keeps jumping up and down and does not get to a steady point compared to when using a high amount of samples.

Since the amount of samples decreased and the number of epochs remained the same, this also meant the training time decreased as well. While the original 50,000 samples took about 57 minutes as already mentioned earlier, the consecutive training procedures took about 55, 55, 26 and 5 minutes. The 30,000 and 10,000 sample size passes took about as long as the 50,000 samples, which is caused by the way the training is done. The data is split into sets of 10 which are trained for 10 epochs each. After all samples were used once the training rate is reduced and the process repeated. In this context, each of the 50,000 samples was used for 3 sets, while each of the 30,000 was used for 5 and of the 10,000 for 15 sets. The 5000 and 1000 sample passes still used 15 sets, hence the reduced training time.

The tracking works in all cases still surprisingly well, even when using only 1000 samples. Idle tracking has the usual jitter to it but it does not grow significantly worse by reducing the amount of samples. Noticeable changes only occur when moving the object. With providing less samples for training, the predicted pose starts to lag behind the actual one and when moving away from the center of the screen tracking can get lost. Also when moving the camera rapidly the tracking might not recognize all movements, but catches up again within a few frames. As expected this decline in tracking quality is most obvious when training with only 1000 samples. When decreasing the amount of samples further it is expected that the tracking will not work anymore very soon.

Reducing the training time instead of the amount of samples or a combination of both will most likely have a bigger impact on the performance of the network than reducing the number of samples alone. The decrease in training time happened automatically in the previous example when decreasing the amount of samples low enough while not increasing the number of epochs. Looking at the graphs of the error rate over time which were provided previously, all show a big initial decrease in error rate and suggest that a short training time will have the biggest effect on the performance.

6.3 The Impact of the Neural Network Architecture

With the Neural Network being the central component of the tracking application, it is worth investigating further. While the architecture used until now was provided by Garon et al. [18], in this section changes will be applied to it like adding and removing layers or adjusting parameters. The goal is to provide an insight into which adjustments could be made to improve tracking performance and how they impact training and execution time.

6.3.1 Layers and Parameters

First it is checked whether the size of the Neural Network can be further reduced and whether processing speed can potentially be gained. By doing so it is to be expected that the stability of the tracking will suffer once a minimum has been reached, which should also show in the error rate during training. Since the original net was meant to support six degrees of freedom, there may be some room to perform those changes. The 2-DoF network will be used for this purpose since its error rate got the lowest and smoothest, which will make changes to the error rate the most obvious. As a reminder, the original structure of the Neural Network can be found in figure 16 in chapter 5.2 for reference. The training process uses the 50,000 samples generated for the 2-DoF network and 150 epochs with different training rates.

At first the preprocessing of the two input tensors with one convolution layer each is removed. The rest of the network remains unchanged. The question is how much the network relies on finding features in the input images separately before concatenating them and treating both inputs as one. On starting the training process, it already becomes clear that this change would break the network completely. The error was very high and did not go down at all. It was completed within 59 minutes, so it took about the same time to train as the unchanged 2-DoF network, even though two layers were missing. When testing the network tracking was lost immediately, even when the object remained idle.

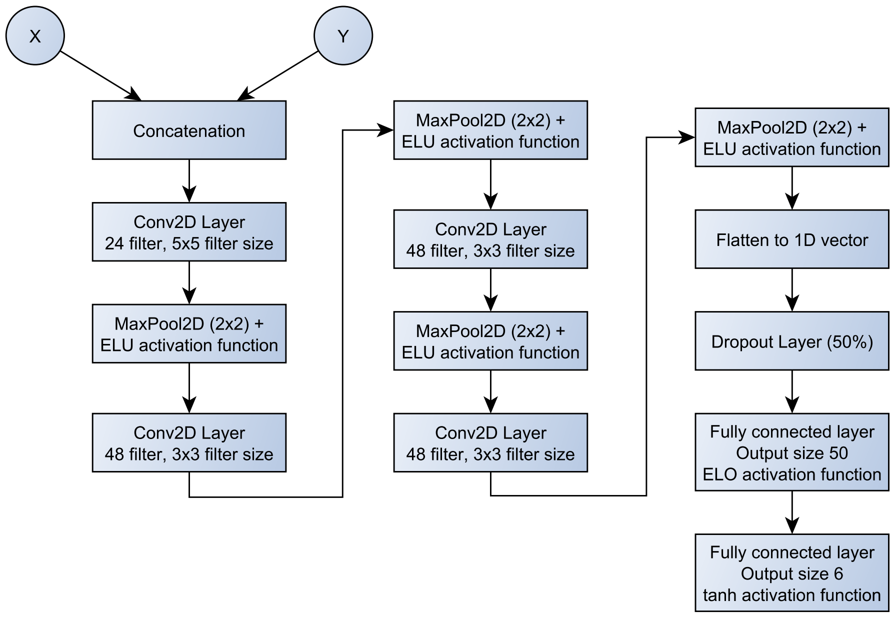


Figure 25: The altered structure of the Neural Network. The additional convolution layer which was applied to each input was combined and moved to take place immediately after the concatenation.

To counteract the removal of the two layers, an additional convolution layer was added after the concatenation. It used the same parameters of the convolution layers applied to the input before the concatenation took place and hence just moved this step. It still is a small decrease in the networks size as only one convolution layer will be used instead of two. The training process again took about as long as the original network with a total of 56 minutes. This time the error rate ended up almost identical with the original network and the tracking test confirmed it was working just as well. The execution time during runtime did not change either, the original and reduced network both took about 3-4ms in average to compute. Besides being able to move this processing step to another place in the pipeline, it is not too surprising there is no or a barely noticeable performance gain. The same amount of data has to be processed, in one case this is done separately and in the other combined. The adjusted network is shown in figure 25 and the error rate can be found in the left graph of figure 26.

Next the influence of the last three convolution layers is investigated, the ones that are processed after the concatenation in the original network. For this first one, then two of them are disabled. With one less the network needs a bit longer to reach a low error value but otherwise, nothing changed. It takes the same amount to train and process the data. This is shown in the left graph in figure 26. The tracking works just as good as

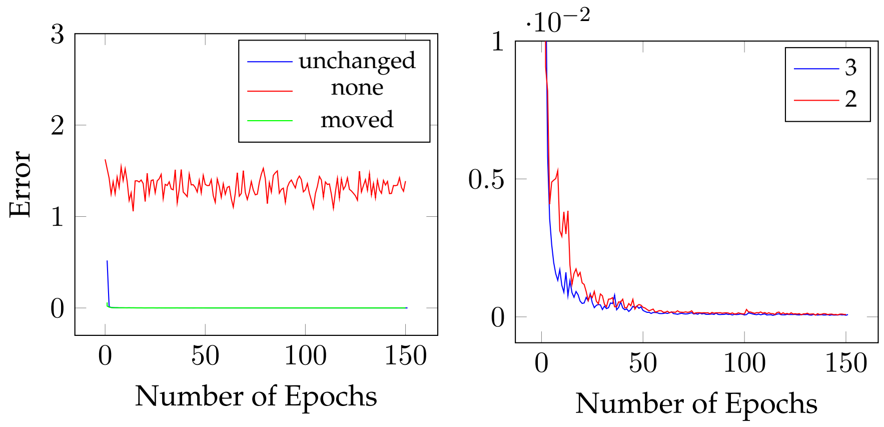


Figure 26: Error rate over time during training for a 2-DoF network with different architectures. Left: Different preprocessing options. Right: Changing the number of main convolution layers.

the original when used in the tracking application. When deactivating two of the convolution layers it has the same effect removing the preprocessing layers had, which is why it is not shown anymore on the graph.

The last part of the original network that has not been touched yet are the two fully connected layers at the end. Since one is the output layer which cannot be removed and the other is the only one connecting the convolution layers and the output layer it makes no sense to remove this one either. So this part remains unchanged.

The tests show that it is possible to reduce the size of the network without losing its capability to track the object. However, there seems to be no noticeable advantage in execution time when training the network or processing the input data. On the contrary, when changing the network to track a higher number of degrees of freedom, further condensing its structure may result in poor tracking results. Hence the network appears to be in its optimal form already and it might be better to add additional features on top to improve the tracking performance.

Adding more convolution layers to the network will most likely not improve the tracking quality. When the input images have been processed by all existing convolution layers their size already is down from 150×150 to 7×7 pixels. Reducing the size even further will make it hard for the fully connected layers to register meaningful features. For this reason I will not pursue this approach. Instead fully connected layers will be added after the convolution layers, which reduce the information output by the convolution layers more slowly. Two cases were considered. The first was adding a fully connected layer with 512 to 50 neurons, the second added another 1024 to 512 neurons on top. Both variants were trained for about one hour with 50,000 samples and uses a 3-DoF network. The choice fell upon this over the 2-DoF network since its error rate has not been as consistent and low and if the additional layers would add any improvement, it should be more visible than by using the 2-DoF which had a decent and stable error rate already.

Just one more layer did not change too much. Looking at the left graph in figure 27 shows the error rate is relatively similar to the one of the original network. It takes a bit longer to reach its low point, and it also got some additional spikes. The tracking works as expected, but did not improve. Adding a second layer did have the same effect as removing the second convolution layer or removing the preprocessing layer. Since the error rate is high and it is similar to behavior encountered and pictured before, it is not included in this graph to not distort the other curves.

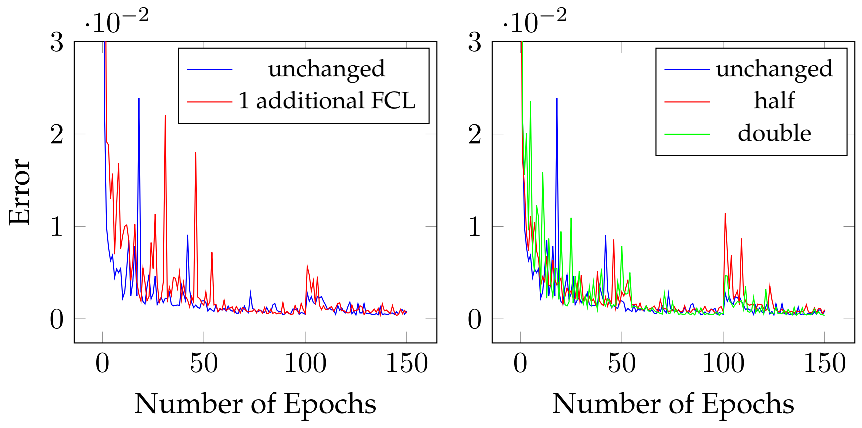


Figure 27: Error rate over time during training for a 3-DoF network with different architectures. Left: Different fully connected layers (FCL). Right: Different number of filters for convolution layers.

The last property tested in this section was changing the number of filters for the convolution layers. The number of filters represents the number of different kernels applied to the image on this convolution layer. Instead of telling the layer which kernels to use, for example a Sobel filter, the network learns them on its own. Decreasing or increasing the number of kernels will reduce or increase the number of features the network is able to learn on a single convolution layer, which may influence the tracking process as well. Since it takes a long time to train a network, two settings will be tried. Cutting the number of filters per layer in half in one and doubling them in the other case. The resulting error rate graph can be found on the right in figure 27.

The total training time was reduced a little bit to about 55 minutes from 58 minutes for half the number of filters, while it increased to about 66 minutes when doubling it. The error rate was not influenced significantly. Halving the number of filters seem to make the result a bit more unstable while the higher number of filters doing the opposite. When testing the trained networks in the tracking application, tracking worked well in all cases. Unless reducing the number of filters even more, changing it on its own without other changes does not seem to improve the tracking results anymore.

6.3.2 The Importance of including the Depth Data

While investigating different degrees of freedom and network architectures, the question arose how important the addition of the depth data is for the tracking process. For this reason the 2-DoF network was altered to receive pure RGB input in one instance and only depth data in another. The network was trained with 50,000 samples for 150 epochs (30 epochs per sample). The samples used for training were the same generated for the normal 2-DoF network. Of those either the RGB channels or the depth values were taken as input for the network. The training process took about 46 minutes for training the RGB only network and about 26.5 minutes for the depth only one. While a reduction in training time was to be expected since the first layer of the Neural Network had to process less data, it was a surprise to see it go down that much. For comparison, the RGB-D version of the 2-DoF network took about 57 minutes to train. The development of the error rate while training is shown in figure 28.

The error rate clearly shows that as soon as either RGB or depth data are missing it becomes significantly more unstable. This means either of those two pieces of information cannot be omitted for the tracking to be fully working. However, even though the depth channel provides only a third of the data compared to the RGB channels combined, it seems to get lower and more consistent. In section 6.1, when discussing the problems the 3-DoF network was having with extreme poses, it was assumed that the depth data might be more important than the RGB data. This graph seems to support that assumption.

Next the trained networks were put to the test in the tracking application and the assumptions made when only looking at the error rate were confirmed. The RGB and the depth data alone, each in its own scenario, did not perform very well on their own. When using the depth only network there was a noticeable increase of jitter in the resting position of the tracked object. Moving the object around resulted in the predicted pose lagging behind and the tracking lost the object very easily. Yet the tracking was working to some degree. When doing slower movements that did not get too close to the borders of the window they were recognized and the pose updated accordingly. On the other hand the RGB network did not work at all. Even in the resting position the predicted pose was jumping around the actual object position and when moving it. The best case one could hope for was that the pose would adjust slightly in the direction the object was

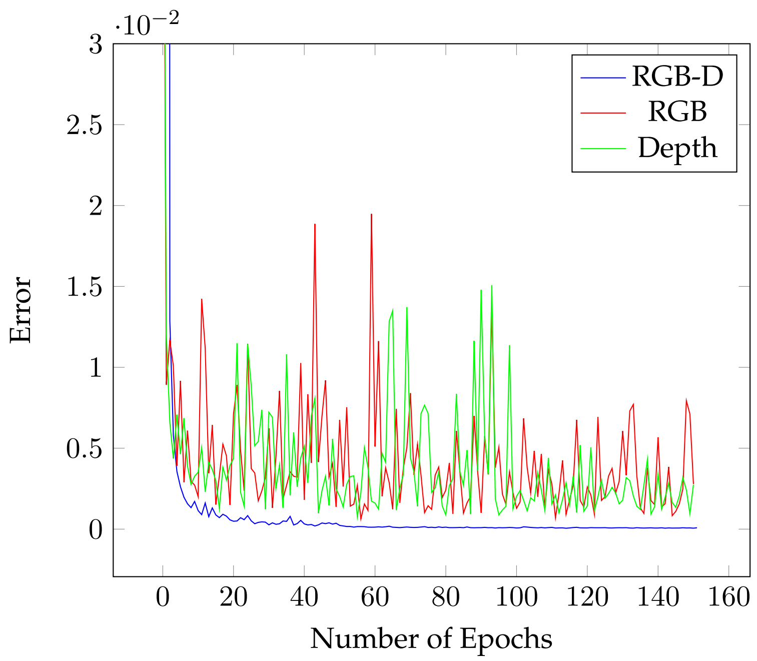


Figure 28: Error rate over time during training for 2-DoF networks which use only the RGB or Depth channels of an image. The RGB-D results are additionally provided for reference.

moved, but in general it was still stuck at the location the object started.

These results show the importance of having both, the RGB image data and the depth information for the tracking to work. However, it also confirmed the importance of the depth data within this process, especially when compared to the plain RGB image.

6.3.3 The usage of CPU and GPU

Until now every task using the Neural Network was computed on the GPU. For both cases, training and processing tracking data, the network as well as the data had to be transferred to the GPU before the computation could be done. The result had to be transferred back to the CPU for further processing. This process takes up additional time, hence the question arises whether it is feasible to perform computations on CPU only.

By design Neural Networks are highly parallel. While the layers depend on the input they receive from the previous one, the neurons on each layer are independent from each other. For this reason it is possible to compute all of them in parallel. It is also not necessary to wait for a layer to fully complete all of its computations. Once the required inputs for a neuron on the next layer are available, its computation can be started. Even modern CPUs commonly only have four to eight cores, the more expensive ones and server processors can have up to 64 at the time of writing. While their cores are slower in raw computation speed, GPUs offer a lot of them, easily outperforming a CPU when doing highly parallelizied tasks. The nVidia GTX 1080Ti which was used for computation offers 3584 cores. A training task that took about 10 minutes to finish on the CPU was done in about 30 seconds on the GPU. This shows the CPU is not an option when it comes to training a Neural Network.

On the other hand this might change when processing just one sample with a fully trained network. Two factors come to mind when considering this case. First there is no backpropagation pass needed when not actively training the network. As a reminder, the backpropagation pass is necessary to update the weights of a Neural Network. Since in this step the computation direction of the network is reversed, which requires the computation of derivatives, it is very expensive and time consuming. For this reason even when training on the GPU the training data is usually processed in mini batches, which means a small amount of samples is computed by the network (e.g. 64) and the backpropagation is performed only once for the average error of the batch. Secondly, not all devices, especially mobile ones, that might be used for production purposes may provide a dedicated GPU which offers general purpose computation capabilities.

The Neural Network is transferred to the GPU just once at the beginning of the application, which makes this step a non factor for the actual tracking process. All steps necessary for tracking, which include transferring the inputs to the GPU, processing them by the network and transferring the output back to the CPU, take up a total of about 6ms when using the 3-DoF network. This is now changed to be done by the CPU only. Compared to the GPU computation this task now takes about 57ms, which is significantly worse. Computing the network on a CPU does not seem to be an option, even when only computing for one input without backpropagation.

7 Conclusion

In the scope of this thesis a temporal object tracker was successfully implemented which utilizes Neural Networks for the tracking process. To use it for tracking, a Convolutional Neural Network which takes two images as inputs - the object in its previous pose and the current frame - was defined and trained with generated training data. The tracker performs in real time and enables tracking up to three degrees of freedom, the translation of the object along the x, y and z axes.

Currently rotating the object, which would add another three degrees of freedom for tracking, is not possible. According to Garon et al. [18] the Convolutional Neural Network which was implemented should be able to track rotation as well. This can also be seen in videos they provide of the tracking process. Neural Networks are a complex topic in on their own and by design they are more like a blackbox than a system in which problems can be easily tracked. This makes it particularly difficult to find and identify possible sources of error, especially when combining them with another notoriously difficult topic in tracking. Therefor getting the tracker to register full six degrees of freedom is a subject for future research.

The analysis of the tracker and the Neural Network, especially the change in network architecture, provided an insight in how the tracker works and how it could be further optimized. It requires only a relatively small amount of data when tracking two degrees of freedom, but including additional degrees of freedom will need an appropriately increased number of samples. The Neural Network which was implemented for this tracker seems to be very optimized already, as simple changes like removing or adding layers, reorganizing them or changing parameters did not result in visible improvements in speed or tracking performance. However, this does not conclude that there are no more improvements that can be made. It will require more time, extensive research and testing to find ways to optimize tracking speed and quality. An important insight gained was on the significance of providing the depth data with the images. It was shown that by processing the depth data on its own, the tracking process was rudimentary working. On the contrary the tracking was not working at all when using the RGB data only, indicating the supportive character of this data.

The implementation of the tracker certainly has room for optimization. When looking at the tracking process as a whole, first transferring the rendered image of the object in its previous pose to the CPU, performing some post processing and then pushing it back to the GPU as input for the Neural Network seems like a waste of computing time. For future iterations it should be checked whether it is possible to never transfer this data and perform all computations necessary on the GPU directly.

Considering real world applications for tracking like Augmented Reality and autonomous systems, this kind of systems usually are very limited in the computing power they can provide. They also may not have a dedicated processor which can be used to outsource the highly parallel computation task of a Neural Network. As shown earlier, the computations necessary for tracking can be done by a CPU without losing too much performance. Hence a subject for future research could be whether this algorithm can be used on mobile or embedded systems to realize real time tracking. It would also be interesting to compare the performance of different algorithms to see how this approach fares against others.

Everything considered, using a Neural Network for tracking is an interesting and promising approach, especially when taking all the recent advances in Machine Learning into account. It renders tedious steps like manually crafting tracking features obsolete and replaces them with an automated process. Besides a higher number of degrees of freedom there are other aspects covered in papers which I did not have the time to have a look at, robustness against occlusions being one of them. This subject has a lot of potential and offers many research possibilities.

References

[1] R. E. Kalman, "A new approach to linear filtering and prediction problems," *Transactions of the ASME–Journal of Basic Engineering*, vol. 82, no. Series D, pp. 35–45, 1960.

[2] B. K. Horn and B. G. Rhunck, "Determining optical flow," *ARTIFICIAL INTELLIGENCE*, vol. 17, pp. 185–203, 1981.

[3] M. K. Pitt and N. Shephard, "Filtering via simulation: auxiliary particle filters," *Journal of the American Statistical Association*, vol. 94, pp. 590–599, 1999.

[4] P. J. Besl and N. D. McKay, "A method for registration of 3-d shapes," *IEEE Transactions on pattern analysis and machine intelligence*, vol. 14, no. 2, 1992.

[5] A. Aldoma, F. Tombari, J. Prankl, A. Richtsfeld, L. D. Stefano, and M. Vincze, "Multimodal cue integration through hypotheses verification for rgb-d object recognition and 6dof pose estimation," *IEEE International Conference on Robotics and Automation*, pp. 2104–2111, 2013.

[6] R. A. Newcombe, S. Izadi, O. Hilliges, D. Molyneaux, A. J. D. D. Kim, P. Kohi, J. Shotton, S. Hodges, and A. Fitzgibbon, "Kinectfusion: Real-time dense surface mapping and tracking," *IEEE International Symposium on Mixed and Augmented Reality*, pp. 127–136, 2011.

[7] Y. Yoon, G. Desouza, and A. Kak, "Real time tracking and pose estimation for industrial objects using geometric features," *Proceedings - IEEE International Conference on Robotics and Automation*, vol. 3, pp. 3473 – 3478, 2003.

[8] C. Choi and H. I. Christensen, "Rgb-d object tracking: A particle filter approach on gpu," *IEEE International Conference on Intelligent Robots and Systems*, pp. 1084–1091, 2013.

[9] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001.

[10] D. Joseph Tan, F. Tombari, S. Ilic, and N. Navab, "A versatile learning-based 3d temporal tracker: Scalable, robust, online," in *The IEEE International Conference on Computer Vision (ICCV)*, December 2015.

[11] S. Akkaladevi, M. Ankerl, C. Heindl, and A. Pichler, "Tracking multiple rigid symmetric and non-symmetric objects in real-time using depth data.," *IEEE International Conference on Robotics and Automation*, pp. 5644–5649, 2016.

[12] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, "Handwritten digit recognition with a back-propagation network.," in *Advances in Neural Information Processing Systems 2*, Denver: Morgan Kaufman, 1990.

[13] Q. Gan, Q. Guo, Z. Zhang, and K. Cho, "First step toward model-free, anonymous object tracking with recurrent neural networks," *CoRR*, 2015.

[14] A. Kendall, M. Grimes, and R. Cipolla, "Posenet: A convolutional network for real-time 6-dof camera relocalization.," *IEEE International Conference on Computer Vision*, pp. 2938–2964, 2015.

[15] E. Brachmann, F. Michel, A. Krull, M. Y. Yang, and S. Gumhold, "Uncertainty-driven 6d pose estimation of objects and scenes from a single rgb image.," *IEEE Conference on Computer Vision and Pattern Recognition*, pp. 3364–3372, 2016.

[16] D. DeTone, T. Malisiewicz, and A. Rabinovich, "Deep image homography estimation," *CoRR*, 2016.

[17] M. Oberweger, P. Wohlhart, and V. Lepetit, "Training a feedback loop for hand pose estimation," *CoRR*, vol. abs/1609.09698, 2016.

[18] M. Garon and J.-F. Lalonde, "Deep 6-dof tracking," *IEEE Transactions on ComputerGraphics and Visualization*, vol. 23, no. 11, 2017.

[19] Y. LeCun, C. Cortes, and C. J. Burges, "The mnist database." `http://yann.lecun.com/exdb/mnist/`. (Version: 19.03.2019).

[20] A. Krizhevsky, V. Nair, and G. Hinton, "The cifar-10 dataset." `https://www.cs.toronto.edu/~kriz/cifar.html`. (Version: 19.03.2019).

[21] P. V. . R. Labs, "Sunrgb-d 3d object detection challenge." `http://rgbd.cs.princeton.edu/challenge.html`. (Version: 19.03.2019).

[22] D. M. Skapura, *Building Neural Networks*. ACM Press, 1996.

[23] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[24] A. Tch, "The mostly complete chart of neural networks, explained." `https://towardsdatascience.com/the-mostly-complete-chart-of-neural-networks-explained-3fb6f2367464`. (Version: 19.03.2019).

[25] The MathWorks Inc., "Introducing deep learning with matlab." `https://www.mathworks.com/content/dam/mathworks/`

`tag-team/Objects/d/80879v00_Deep_Learning_ebook.pdf`. (Version: 19.03.2019).

[26] D.-A. Clevert, T. Unterthiner, and S. Hochreiter, “Fast and accurate deep network learning by exponential linear units (elus),” *arXiv preprint*, 2015.