Learning Visual Odometry with Recurrent Neural Networks

Master Thesis

Adrian Wälchli

University of Bern

January 2018

Abstract

In computer vision, *Visual Odometry* is the problem of recovering the camera motion from a video. It is related to *Structure from Motion*, the problem of reconstructing the 3D geometry from a collection of images. Decades of research in these areas have brought successful algorithms that are used in applications like autonomous navigation, motion capture, augmented reality and others. Despite the success of these prior works in real-world environments, their robustness is highly dependent on manual calibration and the magnitude of noise present in the images in form of, e.g., non-Lambertian surfaces, dynamic motion and other forms of ambiguity. This thesis explores an alternative approach to the Visual Odometry problem via *Deep Learning*, that is, a specific form of machine learning with artificial neural networks. It describes and focuses on the implementation of a recent work that proposes the use of *Recurrent Neural Networks* to learn dependencies over time due to the sequential nature of the input. Together with a convolutional neural network that extracts motion features from the input stream, the recurrent part accumulates knowledge from the past to make camera pose estimations at each point in time. An analysis on the performance of this system is carried out on real and synthetic data. The evaluation covers several ways of training the network as well as the impact and limitations of the recurrent connection for Visual Odometry.

Acknowledgements

I would like to express my deeply felt gratitude to the people who have supported me throughout my master studies and this thesis. In particular, I would like to thank

- my supervisor Prof. Paolo Favaro for his continuous support and guidance,
- the PhD students of the Computer Vision Group, that is, Simon, Givi, Meiguang, Xiaochen, Qiyang, Mehdi and Attila,
- my brothers, parents and grandparents,
- and a bunch of other great friends and colleagues that, in some form or another, contributed to my wellbeing during my master studies: Armin, Susanne, Florian, Raoul, Amanda, René, Lars, Peter, Siavash.

Contents

Chapter 1

Introduction

1.1 Motivation

Building and programming a robot that interacts with and learns from its environment is challenging. Usually the robot is designed to solve a specific task that requires some form of interaction with objects and the environment they occupy. Some robots require navigation to complete their task, e.g., a self-driving car that has to maneuver into a parking space. The robot has to determine its position and orientation in the world with respect to some coordinate system. Furthermore, it may be required to detect and prevent imminent collisions, which requires sensing the distance to nearby objects. One possible way to tackle the navigation problem is to use one or more cameras mounted to the body of the robot. In order for the robot to determine its location, it has to intelligently analyze the motion in the input image sequence coming from the video camera. To avoid collisions or allow interaction, the 3D structure of the surrounding world and objects has to be reconstructed from the camera input or measured by other sensors.

Visual Odometry (VO) (Nistér et al. [2004]) addresses the problem of recovering the ego-motion from a video input in real-time. *Structure from Motion (SfM)*, an extension of VO, additionally recovers the 3D structure from the video input. Although these methods have many practical applications not only in robotics, they have some limitations. First of all, both VO and SfM rely on robust feature detection and matching between images in order to accurately estimate motion. The algorithms to extract these informations from images are hand-designed and tuned to work well in specific domains and environments. They often rely on strong assumptions about the observed scene, lighting conditions or type of motions involved. These assumptions are made to avoid possible ambiguities arising from, e.g., dynamic motion in the scene or non-Lambertian surfaces. Furthermore, classic VO and SfM do not leverage high-level information about the image content that could be used to eliminate and resolve these conflicts.

Quickly after birth, we humans learn to develop a complex understanding of the three-dimensional world around us. The emphasis here is on *learning*. There is no external force or teacher that predetermines how the network of neurons in our brain develops and grows. It is the continuous loop of information retrieval

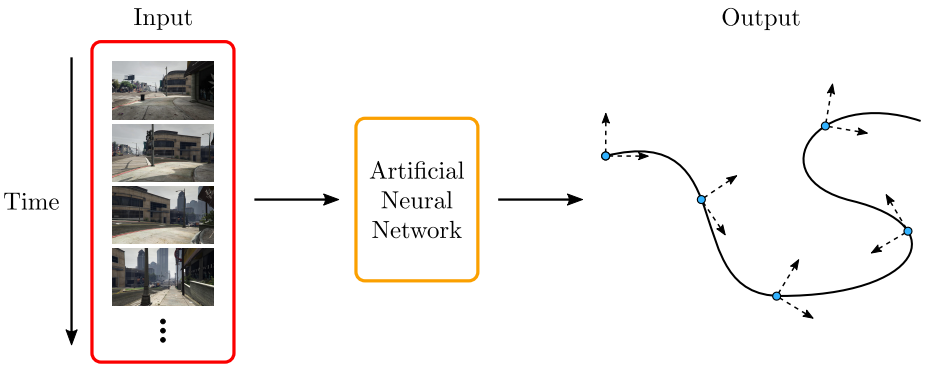


Figure 1.1: Overview of the Visual Odometry problem for Deep Learning. The neural network learns to map the input video (left) to the desired output sequence of poses (right) that describe the path of the camera.

(e.g. seeing, feeling) and action (e.g. body movement) that determines reward or penalty (e.g. walking or falling) which then influences the development of connections in the brain. Recent advances in Deep Learning have shown that machines are able to learn and identify patterns in data and use them to solve complex problems that also humans are faced with everyday, e.g., recognizing a persons face or solving puzzle games. Deep Learning has had a significant impact on computer vision and is under rapid development, becoming more popular every year.

The potential and recent successes of Deep Learning motivate an alternative approach to the classic VO and SfM algorithms. Learning a powerful representation of camera motion from a large dataset of image sequences could help overcome some of the aforementioned challenges in SfM as well as reduce the computational cost to provide real-time mobile applications.

1.2 Visual Odometry by Deep Learning

The problem of Visual Odometry is simple to formulate. Given a sequence of temporally ordered images, i.e. a video, the objective is to compute the camera pose at each frame. The camera pose is the location and orientation of the camera in the world coordinate system. It describes the change of coordinates from the local- to the world reference frame in form of a rotation and translation, each having three degrees of freedom.

There are three main problems that need to be addressed by the proposed solution for VO. First, one needs to specify a suitable reference frame and representation for the camera poses. Second, the system has to be designed in a way that it can handle a video input of variable length and frame rate. Third, it should be robust to noise in form of scene dynamics, lighting changes, camera shake or blur. Furthermore, the system must support any camera model without the need for manual calibration.

In this thesis, we explore a Deep Learning approach to the VO problem as illustrated in figure 1.1. It involves training an artificial neural network on a large number of videos in a supervised fashion. Supervised learning requires

that the true camera poses are known during the learning phase. In order for the network to learn a useful mapping from input to output, it is crucial that the collected data has accurate pose labels for each video frame. The first two of the aforementioned problems are addressed by the architectural choices for the neural network, and the robustness depends, for the most part, on the quality and quantity of the available data for learning.

1.3 Challenges

Training a neural network requires a large dataset with realistic data. Collecting the video data is time consuming, but the challenge lies in the collection of the ground truth, i.e., the camera pose. The accelerometers and gyroscopes in consumer cameras are not accurate enough to capture the pose of the device over a long period of time as they experience large measurement drift. GPS may be used to determine the horizontal location within a few meters depending on the signal strength, but it is not suited for indoor navigation. An alternative way to obtain precise camera pose is by generating it synthetically. However, the challenge here is to generate photo-realistic data with much variation.

Another challenge lies in the design of the architecture of the neural network. It may be problematic to properly separate the motion into rotation and translation, as small rotational changes could be confused with translation. Furthermore, it is unclear how one should design the system to be able to resolve ambiguity. For example, the projection of an arbitrary object is the same when the object is moved twice as far away from the camera and its physical size is doubled. This is known as the *scale ambiguity* and it is inherent to the projective camera model. The *aperture problem* is another example of ambiguity where the global motion of an object can not be determined from local motions in the limited receptive field (aperture) of the camera. Finally, a more serious problem is *scene motion* or *dynamic motion*. Even though object motions usually result in local movement in the image plane, it still remains a challenge to design a system that can properly separate them from the more global camera motion.

1.4 Contribution

The contributions of this thesis are the following.

- A review of prior work on SfM and VO with the focus on Deep Learning oriented methods.
- A large and diverse synthetic dataset of labeled videos for Visual Odometry captured from the video game Grand Theft Auto V.
- An implementation of a VO model using a recurrent neural network architecture as presented in the paper by Wang et al. [2017]. The code is written in Python with the deep learning framework PyTorch[1].
- A study of the models capabilities and limitations with an analysis of hyperparameters, different pose representations, performance metrics and the impact of the recurrent network itself.

[1] `http://pytorch.org`

The proposed system is completely end-to-end. It does not require any external calibration and runs in real time after training is completed.

1.5 Structure of the Thesis

The remainder of this thesis is structured as follows:

- Chapter 2 presents the prior works done on Structure from Motion and Visual Odometry. Both the classical methods as well as the recent approaches with Deep Learning are discussed.
- Chapter 3 contains all the mathematical background needed for the rest of the thesis. Moreover, it reviews the concepts of machine learning, feedforward- and recurrent neural networks.
- Chapter 4 is a detailed description of the proposed deep learning model for Visual Odometry.
- Chapter 5 documents the experiments and results on real and synthetic datasets. It also shows the strengths and weaknesses of the system under various conditions.
- Chapter 6 concludes the thesis with a discussion of the results and improvements for future work.

Chapter 2

Prior Work

We humans have always examined the world around us. There is a certain urge to explore, build and shape the environment we live in. One thing that we have learned long ago is that every object, place, person changes its shape and appearance, the way it behaves and feels like. Sooner or later, everything ages, breaks and vanishes as it contributes to the increase of entropy in our universe. Often we wish to stop this process and capture a moment in time, forever and unchanged, so that we can go back and re-experience. Of course, with the technology we have today it is impossible to instantly capture and store every piece of information about a place or an object on a subatomic level. The complexity of such a measurement process is simply too high. However, we do have devices that allow us to record a sparse subset of all the information. One such device is the digital camera. It captures the light in the same fashion as the human eye. The camera has an image sensor that accumulates the energy from light rays entering through the aperture. The image that forms on the sensor is a perspective projection of the 3D world onto a 2D plane. We humans have two eyes that allow us to recover depth information by exploiting the displacement in the 2D projections of each eye. The same applies to a pair of cameras displaced along a horizontal baseline. A single 3D point in front of the cameras projects to a different 2D location on the two sensors. The resulting shift is called *disparity*. It is inversely proportional to depth, i.e., a small disparity is the result of a point far away from the camera. Therefore, a stereo image pair allows us to get the depth values for each pixel.

Although the depth reconstruction from stereo vision has many practical applications, it does not give a full 3D geometry of the observed scene, since a camera can only capture the visible parts. It is therefore not possible to reconstruct the 3D of surfaces that are occluded or outside the field of view. In order to obtain a full 3D reconstruction of a static object or scene, images have to be taken from many positions and angles. Through the motion of the camera, in a continuous fashion new 2D measurements of points are obtained that would otherwise be occluded from a single viewing direction. The technique of recovering the 3D geometry from multiple images is called *Structure from Motion (SfM)*. In the typical SfM pipeline the input is an ordered or unordered list of images and the output consists of the camera poses and the 3D point cloud. The typical SfM pipeline can be summarized in three basic steps:

1. Feature detection, extraction and matching
2. Camera motion estimation
3. Recovery of the 3D structure

This chapter presents the prior work that focuses on one or multiple of these steps. Although the topic of this thesis is Visual Odometry, the discussion of prior works includes recovery of 3D structure to give a better overview since the two research areas are closely related.

2.1 Established Methods

2.1.1 Feature Extraction and Matching

All SfM algorithms depend on a robust feature detection and matching to solve the correspondence problem between two or more images. It is usually the first step in the SfM pipeline. A pair of 3D coordinates, one in each camera view, is called *corresponding* if they depict the projection of the same 3D point. In this case, the feature description in the local area of these keypoints matches. A reasonable feature descriptor should be invariant to rotation, scale, illumination changes and noise. One of the descriptors that addresses these concerns is the *Scale Invariant Feature Transform (SIFT)* introduced by Lowe [1999]. In the first step, the feature detection, SIFT identifies patterns that are distinctive from their neighborhood by finding edges, corners and blobs in the image. The second step transforms these patterns (in form of image patches) into a scale-invariant representation by convolving with multiple Difference of Gaussian (DoG) kernels at different scales of the image (via down-sampling). At this stage, one of the candidate scales is selected based on contrast and edge response. In this way, the same pattern in two different images at different scales will be transformed to the same scale in the feature space and therefore invariance is satisfied. In the third step, rotation invariance is achieved by computing a dominant orientation of the patch based on local gradients. The patch is then further subdivided into subregions where a histogram of gradients (HOG) is computed. All of this information is then encoded into a 128-dimensional vector describing a single feature.

The extraction of these features works the same for every image, and the features descriptors can be compared directly to find correspondence candidates. Although SIFT has proven to be very robust, the computation of the features is expensive, especially the scale-invariant part. Subsequent works have proposed improved variants of SIFT such as PCA-SIFT (Ke and Sukthankar [2004]) or SURF (Bay et al. [2006]). Despite the successes of these methods in many applications, e.g. object recognition or panorama stitching, they rely on hand-crafted functions that are empirically derived. Furthermore, SIFT is defined based on very low-level image characteristics, which is contrary to the workings of the human brain, the most-sophisticated system for vision we know.

As we will see in section 2.2, neural networks can be designed to automatically learn useful features without the need for manual intervention and tuning. These networks are built in an intuitive way. It is often easier and faster to train a neural network instead of manually adapting features to a new domain.

2.1.2 Structure and Motion from Two Images

Early works on SfM focused their study on the geometry between two camera views only. An important algorithm that emerged from these works is the *The Eight Point Algorithm*, introduced by Longuet-Higgins [1981]. It makes use of the epipolar geometry between a pair of views in order to compute relative translation and orientation of the cameras. The relationship between the two camera views is described by the so-called *essential matrix*. It is defined as

$$\boldsymbol{E} = [\boldsymbol{t}]_{\times}\boldsymbol{R}, \tag{2.1}$$

where $[\boldsymbol{t}]_{\times}$ is the matrix that computes the cross-product between $\boldsymbol{t}$ and an arbitrary vector. The essential matrix describes the relationship between corresponding points in the image planes of the two cameras. When $\boldsymbol{R}$ and $\boldsymbol{t}$ are unknown, it is possible to compute the essential matrix using known corresponding points. The epipolar constraint is formulated as

$$\hat{\boldsymbol{x}}_1^{\top}\boldsymbol{E}\,\hat{\boldsymbol{x}}_0 = 0, \tag{2.2}$$

where $\hat{\boldsymbol{x}}_0$ and $\hat{\boldsymbol{x}}_1$ are the corresponding points in image one and two respectively. The unknown elements of the essential matrix can be determined using this system of equations. It requires eight equations for eight elements of the matrix and one element has to be fixed because any scaling of the true matrix satisfies the epipolar constraint. After estimating $\boldsymbol{E}$, the rotation and translation can be uniquely recovered from it up to an unknown scaling of the translation.

In practice, there are a few problems that need to be addressed. First, the corresponding points are usually not known and must be found using a feature detection and matching technique (SfM step 1), e.g. SIFT. This then can lead to outliers, which are incorrectly assigned correspondences. In this case, RANSAC (Fischler and Bolles [1981]) is applied to select the best eight points among many possible choices. Second, the selected coordinates are usually not exact and contain noise that lead to a rank three estimate of $\boldsymbol{E}$ instead of rank two. The singular value decomposition is used to discard the component with the smallest singular value. Third, in some cases the camera calibration matrices $\boldsymbol{K}_0$ and $\boldsymbol{K}_1$ are unknown. This means that the normalized coordinates $\hat{\boldsymbol{x}}_j = \boldsymbol{K}_j^{-1}\boldsymbol{x}_j$ are no longer available and only the image coordinates $\boldsymbol{x}_j$ are known. This results in the so called *fundamental matrix* $\boldsymbol{F}$ in the epipolar constraint:

$$\hat{\boldsymbol{x}}_1^{\top}\boldsymbol{E}\hat{\boldsymbol{x}}_0 = \boldsymbol{x}_1^{\top}\boldsymbol{K}_1^{-\top}\boldsymbol{E}\boldsymbol{K}_0^{-1}\boldsymbol{x}_0 = \boldsymbol{x}_1^{\top}\boldsymbol{F}\boldsymbol{x}_0 = 0. \tag{2.3}$$

The same way as for the essential matrix, $\boldsymbol{F}$ can be computed using the Eight Point Algorithm although the fundamental matrix is less useful than the essential matrix because it is not anymore possible to uniquely recover $\boldsymbol{R}$ and $\boldsymbol{t}$ in the uncalibrated case. Nonetheless, $\boldsymbol{F}$ defines the epipolar constraint and therefore can be used to find more correspondences in the image pair.

The 3D geometry can be reconstructed (up to scale) by triangulation using the essential matrix to determine two lines for corresponding projections. Ideally, the two lines passing through the center of projection and the point on the image plane intersect in 3D space. In practice, the measurement- and numerical errors in the estimation of the essential inhibits an intersection of these lines. In this case it is possible to estimate the 3D point to be the midpoint of the shortest segment connecting the two lines.

2.1.3 Multi-View Structure from Motion

So far, we have seen a method for estimating the relative camera pose and 3D structure based on two images by solving the epipolar constraint with matched feature points. The next step is the reconstruction from multiple camera views. The factorization method can be used to recover motion and 3D shape simultaneously for all cameras (step 2 and 3). A first method was introduced by Tomasi and Kanade [1992] that works with an orthographic camera model. It uses the so called *measurement matrix* $\boldsymbol{W} \in \mathbb{R}^{2n \times m}$ which is defined as

$$\boldsymbol{W} = \begin{bmatrix} x_{11} & \cdots & x_{1m} \\ \vdots & \ddots & \vdots \\ x_{n1} & \cdots & x_{nm} \\ y_{11} & \cdots & y_{1m} \\ \vdots & \ddots & \vdots \\ y_{n1} & \cdots & y_{nm} \end{bmatrix}. \tag{2.4}$$

It contains the 2D coordinates of the m orthographically projected 3D points for each of the n cameras. Under the assumption that the origin of global coordinate system is located at the centroid of the point cloud, the matrix W can be factorized into

$$\boldsymbol{W} = \boldsymbol{M}\boldsymbol{S}, \tag{2.5}$$

where $\boldsymbol{M} \in \mathbb{R}^{2n \times 3}$ contains the 2×3 projection matrices of each camera and $\boldsymbol{S} \in \mathbb{R}^{3 \times m}$ are the points of the observed 3D shape. As described by Tomasi and Kanade [1992], this factorization can be achieved using the singular value decomposition (SVD) as $\boldsymbol{W} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^\top$ and deriving $\boldsymbol{M}$ and $\boldsymbol{S}$ from $\boldsymbol{U}$, $\boldsymbol{V}$ and $\boldsymbol{\Sigma}$.

This simple factorization method has some disadvantages. First, it can not be applied to the commonly used perspective projection model, although it is approximated by the orthographic model for distant objects. One reason for this is the ambiguity that comes with the projective camera model. Applying a projective distortion $\boldsymbol{Q} \in \mathbb{R}^{4 \times 4}$ to the scene and inverting it by multiplying the inverse transformation to the camera matrix, the projection is exactly the same, i.e.,

$$\boldsymbol{P}\boldsymbol{x} = (\boldsymbol{P}\boldsymbol{Q}^{-1})(\boldsymbol{Q}\boldsymbol{x}). \tag{2.6}$$

Without any constraints on the projection matrices, the factorization would be ambiguous.

The works of Sturm and Triggs [1996] and Christy and Horaud [1996] have extended the factorization method for perspective projection using an iterative process by first recovering the projective depth using the fundamental matrices between successive pairs of images. Second, in order for the factorization method to work, all points of the shape must be visible in every frame, i.e., to obtain the measurement matrix one needs to find the correspondences across all frames. This is a serious limitation makes the factorization method less usable in practice.

An alternative method to solve multi-view SfM is the *bundle adjustment* technique. It is the most general technique for simultaneously recovering 3D shape and camera pose, but it is also computationally expensive. The bundle adjustment technique aims to minimize the re-projection error of the unknown

3D points and camera matrices. The re-projection error is formulated as the euclidean distance between the known image coordinates and the projection of the unknown points using the unknown camera matrices.

More formally, let $\{\boldsymbol{C}_j\}_{j=1}^{m}$ be the (unknown) camera matrices that encode location, rotation and intrinsic parameters, let $\{\boldsymbol{p}_i\}_{i=1}^{n}$ denote the (unknown) 3D points and let $\boldsymbol{x}_{ij}$ be the (known) observed 2D coordinates of the point $\boldsymbol{p}_i$ in camera j. Then, the objective of bundle adjustment is defined as the optimization problem

$$\min_{\{\boldsymbol{p}_i\},\{\boldsymbol{C}_j\}} \quad \sum_{i=1}^{n}\sum_{j=1}^{m} v(i,j)\left\|P(\boldsymbol{C}_j,\boldsymbol{p}_i)-\boldsymbol{x}_{ij}\right\|_2, \tag{2.7}$$

where $v(i,j)$ denotes the visibility of point i in camera j and P is the projection operation with homogeneous division. This optimization problem is non-convex and as explained by Özyeşil et al. [2017] naïve optimization algorithms achieve only a poor local minimum. One approach to find a better local minimum is to initialize using an other SfM algorithm such as the factorization method. Since bundle adjustment is very computationally expensive for many cameras, it is often used as a refinement step.

2.2 Learning-Based Methods

Recently, it has been shown that Convolutional Neural Networks (CNN) are very effective at detecting and extracting features in images. Many computer vision problems like image classification have benefited from the newly introduced CNN architectures like AlexNet (Krizhevsky et al. [2012]). A CNN applies a set of convolution filters to the input image, followed by a non-linear operation that acts as a thresholding. Patterns in the input that "match" the filters have a high response (activation), and these activations will again be convolved with new filters, layer after layer. The key is that the convolution filters are not fixed, but they are learned through many examples of data. Due to the limited kernel size of the convolutions, the receptive field increases with each layer, representing a growing abstraction level that is key to understanding complex spatial relationships in images. Network architectures like VGGNet (Simonyan and Zisserman [2014]), GoogLeNet (Szegedy et al. [2015]) or ResNet (He et al. [2016]) have demonstrated that greater depth (more layers) has a positive impact on the learned features and performance for image classification and other tasks.

2.2.1 Learning Camera Motion

The PoseNet, as proposed by Kendall et al. [2015], is a CNN that performs regression for the location and orientation of the camera given a single image as input. It outputs a 7D vector that describes the camera pose $\boldsymbol{p} = [\boldsymbol{x}, \boldsymbol{q}]^\top$ with a location $\boldsymbol{x}$ and orientation $\boldsymbol{q}$ as a quaternion. The architecture is a modified GoogleNet with 23 layers which was trained on a classification task. They replace the last layers to perform regression instead of classification.

Because CNNs require a large amount of training data, the authors apply transfer learning by pre-training the network on a different task with large datasets such as *ImageNet* and *Places*. Pose regression is performed using the

pre-trained network and their own dataset called *Cambridge Landmarks* with five scenes of camera motion in large scale outdoor areas for which ground truth pose is available. A challenge that comes with this dataset is the clutter in form of moving pedestrians and cars. Also, due to the long trajectories, weather and lighting conditions change a lot. This is contrary to the aforementioned classic SfM approaches where it is always assumed that the scene is static, without moving objects.

In their experiments they find that the system is robust to large spatial distance between camera samples. Also, from the visualization of the features they observe that the network does not only produce high response outputs for high level features, but it is also sensitive to large textureless regions where SIFT-based approaches typically fail.

For the most part, the objective of PoseNet is very different from the objective of this thesis. It estimates the camera parameters only based on one input image and in an absolute coordinate system. This means that network is specifically trained for a small number of scenes, e.g., the Cambridge landmarks, and it is not intended to generalize to other scenes and settings. Nonetheless, the work of Kendall et al. shows that transfer-learning can be used for pose estimation when a large labeled dataset for training is not available and that the pre-trained network can aid to learn pose information despite being trained for pose-invariant outputs.

Melekhov et al. [2017] propose a CNN architecture to estimate the relative pose between two camera images. Similar to PoseNet Kendall et al. [2015], they estimate the 7D pose vector with a network composed of two main parts. The first is the *representation part*. It consists of two identical branches with shared weights of five convolutional layers with ReLUs and Spatial Pyramid Pooling (SPP) at the end. Each branch processes one of the input images. The second part is the *regression part*. Regression is performed on the output of the representation part using two fully-connected layers to estimate the relative pose. The weights for the Siamese network are initialized from a pre-trained AlexNet on the *ImageNet* and *Places* datasets. Melekhov et al. apply SPP in the representation part in order to overcome the limitation in the input size due to the fully-connected layers. This allows for a larger size of input images for the network to be able to extract more structural information that helps reconstructing the pose. The dataset used for training is a crowd-sourced image collection with ground truth pose from a SfM pipeline based on local feature matching. For testing and comparison with other works they use the *DTU* dataset where the exact pose is obtained from a robotic arm that moves the camera.

The difference to PoseNet is that the estimated camera pose is relative, and the aim is to learn a representation for motion between frames instead of absolute localization. This is a more general approach, with a clear separation of concerns, i.e., the feature extraction (representation part) and pose estimation (regression part). It could be directly applied for Visual Odometry by computing the relative pose between subsequent pairs of images in the video. In fact, Mohanty et al. [2016] have exactly followed this idea. Their model has a very similar architecture as Melekhov et al.. What stands out is their experiment where prior knowledge is supplied to the CNN by an additional input of features from the FAST corner detector (Rosten and Drummond [2006]). They find that the CNN learns to extract similar features to FAST that work better compared

to their initial experiments using a pre-trained AlexNet.

Wang et al. [2017] add a new component to the past deep learning approaches for VO, the recurrent neural network (RNN). The RNN differs from a regular neural network in the sense that it is designed to learn dependencies over a variable amount of time. This makes it suitable for problems that involve a sequence of inputs with temporal dependency, as in a video. The idea is that the information learned in the past (e.g. from earlier video frames) provides the *context* for the present input. Wang et al. also use a pre-trained CNN architecture for feature extraction, followed by the RNN for pose regression. However, as opposed to previous works, the CNN is not initialized with weights from motion-invariant tasks. It is a modified version of FlowNet (Dosovitskiy et al. [2015]) pre-trained on the synthetic *Flying Chairs* dataset. Although this data does not include camera motion, Wang et al. show that it still serves as a good initialization for the VO problem. Their paper was released concurrently to this thesis and both share the same ideas, hence this thesis can be seen as a study and supplement to the original work.

There is an alternative approach by the same authors (Clark et al. [2017]) that tackles the problem of Visual-*Inertial* Odometry. The interesting part of their architecture called VINet is that they provide pose measurements from an inertial measurement unit (IMU) as input to the network, in addition to the video. The IMU is a physical device that consists of accelerometers and gyroscopes which are used to derive position and orientation of the device. The measurement accuracy of the IMU suffers from accumulated error over time, so-called "drift". Therefore the input to VINet is not the true camera pose, but a noisy measurement of it. Clark et al. use two separate RNNs, one for the visual signal (the video input) and one for the pose stream (from the IMU). The idea behind this setup is that the combined input of both the visual features and the noisy pose measurements improve the networks ability to output a more accurate ego-motion estimate. In theory, the additional pose input should help the network handle scene dynamics and the recovery of scale. Although the work demonstrates that RNNs are useful for combining information of multiple streams of data, the aim of this thesis is pure VO with only the video input as resource for ego-motion.

In summary, we can conclude that there are two main trends. One is the use of pre-trained networks for feature extraction such as AlexNet or FlowNet to accelerate training on small datasets. Second, RNNs have shown to be very powerful for solving sequence-to-sequence problems, e.g. in the text domain, speech recognition or machine translation. They are not so well-studied in the context of image- and video domain and present many challenges and questions for future work.

2.2.2 Learning Structure and Motion

The previous subsection has given an overview of recent works for learning ego-motion from video. In theory, all the presented networks have to learn a weak representation of 3D structure even though they are never asked to output depth. This is because motion and depth are two highly coupled quantities that both contribute to the optical flow in the observed image sequence.

Vijayanarasimhan et al. [2017] implement a deep learning approach to SfM that supports various types of supervision. Their architecture consists of two

subnetworks, one for motion and one for structure. The former takes a pair of consecutive video frames as input and computes a fixed number of segmentation masks for moving objects. Using these masks, each pixel is assigned to an object-specific transformation by a learned rotation and a translation. In addition, camera rotation and translation are regressed using the features from intermediate layers. The second subnetwork estimates depth from a single input image (the left image of the pair in the motion network). The output is a cloud of 3D points, one for each pixel coordinate in the input image.

In total, the network estimates relative camera motion, per-object transformations with motion masks, and a point cloud. Using the camera and object transforms, the point cloud can be re-projected into the two camera views to obtain the optical flow. This then presents two options for supervision. One is self-supervision with a brightness-constancy criterion between the two input frames using the optical flow. The other option is to supervise all components individually, i.e. camera motion, depth and optical flow. Their results show typical failure cases in the self-supervised setting where shadows and occlusions break the brightness-consistency criterion, which hinders the network to properly separate object and camera motion. However, the self-supervised approach still opens the possibility to use it as an initialization process for a supervised training on smaller datasets.

In a concurrent work, Zhou et al. [2017] go a step further and do completely unsupervised training for camera pose and depth. In contrast to Vijayanarasimhan et al. they do not explicitly model the scene dynamics and assume that camera intrinsics are known in advance. Both works have in common that the depth estimation is only from a single view. It is unclear how well-developed the representation is that the network learns since in both works, the subnetwork for single-view depth is separated from the motion network. With a single image as input, the depth estimation is an ill-posed problem.

Chapter 3

Foundations

This chapter presents the necessary theoretical background for the thesis. The first part is about the representation of the camera pose, which is ultimately the output of a system for Visual Odometry. In the second part, it follows a very brief review of Deep Learning. In particular, the recurrent neural network, an important building block for the Visual Odometry model in this thesis, is introduced. For a more comprehensive insight into Deep Learning, the reader is advised to consult Goodfellow et al. [2016].

3.1 Transforming Camera Pose

The camera pose, also known as the *extrinsic parameters*, is a combination of position and orientation. Together, they define a coordinate transformation from the camera coordinate frame to the world coordinate frame. Such a transformation is commonly represented by a translation vector $\boldsymbol{t} \in \mathbb{R}^3$ and a rotation matrix $\boldsymbol{R} \in SO(3)$. Together, they describe a rigid transformation which can also be written as a 4×4 matrix of the form

$$\boldsymbol{T} = \begin{bmatrix} \boldsymbol{R} & \boldsymbol{t} \\ 0 & 1 \end{bmatrix} \in SE(3). \tag{3.1}$$

In Structure from Motion (SfM), there is not only one camera, but many cameras with different transformations. This is also the case in Visual Odometry (VO), but the sequence of transformations is ordered according to the path of the camera motion. Specifically, we deal with a sequence of rotations $\boldsymbol{R}_1, \ldots, \boldsymbol{R}_n$ and translations $\boldsymbol{t}_1, \ldots, \boldsymbol{t}_n$. Depending on how the poses are obtained, they are all relative to a common (world) coordinate frame. Sometimes it is necessary or useful to have all transformations relative to a new coordinate system. This can be achieved by applying the transformations

$$\begin{aligned} \boldsymbol{R}_{i \to j} &= \boldsymbol{R}_j^\top \boldsymbol{R}_i \\ \boldsymbol{t}_{i \to j} &= \boldsymbol{R}_j^\top (\boldsymbol{t}_i - \boldsymbol{t}_j) \end{aligned} \tag{3.2}$$

to get the new relative rotation $\boldsymbol{R}_{i \to j}$ and relative translation $\boldsymbol{t}_{i \to j}$. The arrow notation $i \to j$ denotes that the i-th transformation is in coordinates of system j. Note that for $i = j$ we obtain $\boldsymbol{R}_{i \to j} = \boldsymbol{I}$ and $\boldsymbol{t}_{i \to j} = \boldsymbol{0}$.

Representation	Constraint	Problems	Identity
Euler angles	None	Gimbal lock	Ambiguous
Axis-angle	Unit-norm axis	Missing algebraic structure	Ambiguous
Matrix	Must be in $SO(3)$	Numerical instability	Unique
Unit quaternion	4D unit sphere	Less intuitive	Unique

Table 3.1: A comparison of representations for rotations.

The formulas in equation 3.2 can be applied to every transformation in the sequence for $i = 1, \ldots, n$ and having j fixed, e.g. $j = 1$ would make all transformations relative to the first camera coordinate system. In the remainder of this thesis we refer to the camera pose in this format as the *global pose*, where the first pose in the sequence always represents the coordinate system of all other poses. An alternative choice is $j = i - 1$. We refer to this special setting as the *incremental pose*, since all transformations are relative to the previous video frame, i.e. we have $\boldsymbol{R}_{i \to i-1}$ and $\boldsymbol{t}_{i \to i-1}$ for $i = 2, \ldots, n$.

3.2 Representations for Pose

So far, we have seen the camera pose represented as a 4×4 transformation matrix with a rotational and translational part (equation 3.1). In a system for VO we could, for example, aim to compute the rotation and translation separately and then plug them together into a transformation matrix. For the translation, the system would need to estimate the three components in X-, Y- and Z-direction. However, it is unclear how the system should be designed to output the elements of a rotation matrix, since it is constrained to be in $SO(3)$, which is the group of matrices with determinant equal to one and for which the transpose is equal to the inverse. A constraint like this is difficult to enforce directly.

Actually, it is redundant to describe rotations with nine numbers when in fact there are only three degrees of freedom for a rotation in 3D, i.e., two degrees are needed for the orientation of the axis and one for the angle of rotation around the axis. It is also not straightforward to interpolate rotations in $SO(3)$, or to find the best approximation for a matrix that lies outside of $SO(3)$. The later is definitely useful when numerical errors occur in computations involving the rotation.

What follows is a discussion and comparison of three other popular representations used to describe rotations. A brief overview these is also shown in table 3.1.

Euler Angles The Euler angles represent a rotation with three numbers $\alpha, \beta, \gamma \in [0, 2\pi)$, which are also known as *pitch*, *roll* and *yaw*. There are many different conventions and definitions of these three angles and how they are applied. In this thesis, we apply the rotations in the following order. The initial canonical coordinate frame shall be denoted by (x, y, z).

1. The coordinate frame is rotated around the x-axis with angle α. This results in a new coordinate frame (x', y', z').

2. Next, the system is rotated around the y'-axis by angle β. The new coordinate frame is (x'', y'', z'').

3. Finally, we perform a rotation around axis z'' by angle γ to obtain the desired orientation.

Altogether these steps can be formalized by a multiplication of three matrices that define the rotations around the different axes. The result is a rotation matrix

$$\boldsymbol{R}(\alpha, \beta, \gamma) = \boldsymbol{R}^{\gamma}_{z''} \boldsymbol{R}^{\beta}_{y'} \boldsymbol{R}^{\alpha}_{x} \tag{3.3}$$

defined by the Euler angles. Although the above description of how the rotations are applied is more or less intuitive, the computation of rotation matrices around arbitrary axes in equation 3.3 is a bit complicated. Fortunately, it can be shown that

$$\boldsymbol{R}(\alpha, \beta, \gamma) = \boldsymbol{R}^{\alpha}_{x} \, \boldsymbol{R}^{\beta}_{y} \, \boldsymbol{R}^{\gamma}_{z}, \tag{3.4}$$

where

$$\begin{aligned}
\boldsymbol{R}^{\alpha}_{x} &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\alpha) & -\sin(\alpha) \\ 0 & \sin(\alpha) & \cos(\alpha) \end{bmatrix} \\
\boldsymbol{R}^{\beta}_{y} &= \begin{bmatrix} \cos(\beta) & 0 & \sin(\beta) \\ 0 & 1 & 0 \\ -\sin(\beta) & 0 & \cos(\beta) \end{bmatrix} \\
\boldsymbol{R}^{\gamma}_{z} &= \begin{bmatrix} \cos(\gamma) & -\sin(\gamma) & 0 \\ \sin(\gamma) & \cos(\gamma) & 0 \\ 0 & 0 & 1 \end{bmatrix}
\end{aligned} \tag{3.5}$$

are simple rotations around the axes of the initial (canonical) coordinate system (x, y, z). Further, it can be shown that this representation covers all 3D rotations, and in reverse, every rotation matrix $\boldsymbol{R} \in SO(3)$ can be decomposed into the form in equation 3.4, e.g. to extract the Euler angles.

Unfortunately, the Euler angle representation comes with some limitations. There are special configurations of angles that can lead to a loss in the degrees of freedom for the rotation. For example, if we fix $\beta = \frac{\pi}{2}$, any choice of values for the other angles results in a matrix where the first row is always the same. Therefore, one degree of freedom for rotation is lost. This unavoidable problem is also known as *gimbal lock* and it is inherent to the way Euler angles are defined. A second limitation worth mentioning is the fact that interpolation between two rotations in Euler form is not intuitive and quite complicated. It is better to convert to a different representation and perform the interpolation there, e.g., with quaternions.

Axis-Angle Every rotation in three-dimensional space can be defined by an angle $\theta \in \mathbb{R}$ and a vector $\boldsymbol{a} \in \mathbb{R}^3$ for the axis of rotation. We denote it by the tuple $(\boldsymbol{a}, \theta)$. The axis can be thought of as the normal vector on the plane in which the rotation takes place. The axis itself has three degrees of freedom, but for the rotation, only two of these matter. Therefore the axis is often assumed to be normalized, i.e. $\|\boldsymbol{a}\| = 1$. There are two main limitations to the axis-angle, which often make it only usable as an intermediate representation.

One is the missing algebraic structure to directly concatenate several rotations. Second, the representation is ambiguous. We have that $(\boldsymbol{a}, \theta)$ represents the same rotation as $(-\boldsymbol{a}, -\theta)$ for all possible axes and angles, and the identity rotation can be represented by any axis-angle $(\boldsymbol{a}, 2\pi k)$ for $k \in \mathbb{Z}$.

This ambiguity and the unit-norm axis constraint make the axis-angle representation less useful for this work. Especially for supervised training of artificial neural networks, it is best to eliminate as many ambiguities as possible to avoid confusion.

Unit Quaternions Quaternions are four-dimensional numbers. They are an extension of the complex numbers. Formally, a quaternion is defined as $\boldsymbol{q} = w + \boldsymbol{i}x + \boldsymbol{j}y + \boldsymbol{k}z$ where $w, x, y, z \in \mathbb{R}$ and $\boldsymbol{i}, \boldsymbol{j}$ and $\boldsymbol{k}$ are the basis elements of the quaternion space. The quaternion can also be written as a tuple $\boldsymbol{q} = (w, x, y, z) \in \mathbb{R}^4$ by introducing the notation

$$\mathbf{1} = \begin{bmatrix}1\\0\\0\\0\end{bmatrix}, \quad \boldsymbol{i} = \begin{bmatrix}0\\1\\0\\0\end{bmatrix}, \quad \boldsymbol{j} = \begin{bmatrix}0\\0\\1\\0\end{bmatrix}, \quad \boldsymbol{k} = \begin{bmatrix}0\\0\\0\\1\end{bmatrix}. \tag{3.6}$$

But quaternions are not just tuples of numbers. They are equipped with a separate addition and multiplication. Addition is simply carried out elementwise as for vectors, whereas multiplication makes use of the distributive rule and the property

$$\boldsymbol{i}^2 = \boldsymbol{j}^2 = \boldsymbol{k}^2 = \boldsymbol{ijk} = -\mathbf{1}. \tag{3.7}$$

Furthermore, the norm of a quaternion is defined as

$$\|\boldsymbol{q}\| = \sqrt{w^2 + x^2 + y^2 + z^2}. \tag{3.8}$$

One particularly interesting subset of the quaternions are those with unit norm, i.e., $\|\boldsymbol{q}\| = 1$. They are called *unit quaternions*. The quaternions of this form describe 3D rotations and can be parameterized by

$$\boldsymbol{q} = \cos\left(\tfrac{\theta}{2}\right) + \sin\left(\tfrac{\theta}{2}\right)(a_1\boldsymbol{i} + a_2\boldsymbol{j} + a_3\boldsymbol{k}), \tag{3.9}$$

where $\theta \in \mathbb{R}$ is the angle of rotation around a unit-length axis $\boldsymbol{a} = (a_1, a_2, a_3) \in \mathbb{R}^3$. The parameterization makes use of the axis-angle representation, however, quaternions are more powerful in the sense that rotations can be concatenated via multiplication, i.e., $\boldsymbol{q}_2\boldsymbol{q}_1$ represents the compound rotation of $\boldsymbol{q}_1$ followed by $\boldsymbol{q}_2$. One drawback of the quaternion is that it is a double cover of the space of rotations, meaning that $\boldsymbol{q}$ and $-\boldsymbol{q}$ represent the same rotation.

How well a neural network can be trained to output a rotation may depend on the choice of the representation and its constraints. An analysis on this matter is done in chapter 5. We now continue with a brief review on artificial neural networks and machine learning.

3.3 Deep Feedforward Neural Networks

A *feedforward neural network* is a function mapping an input $\boldsymbol{x}$ to an output $\boldsymbol{y} = f(\boldsymbol{x})$. For instance, $\boldsymbol{y} \in \{\text{dog}, \text{cat}\}$ could be the output of the function

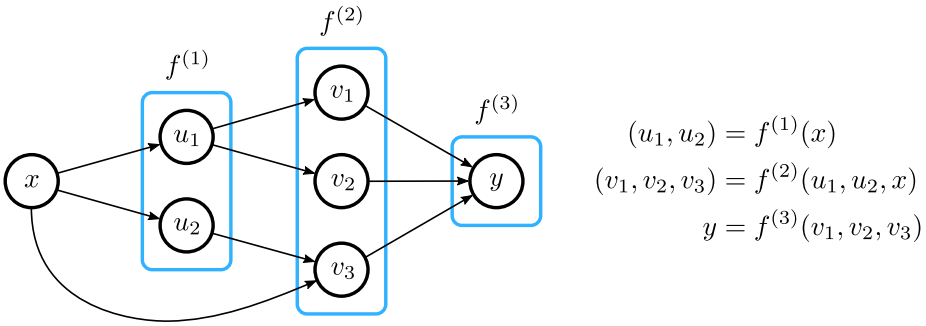


Figure 3.1: Example of a feedforward neural network represented by the computational graph. Each layer (blue) operates on the outputs of previous layers, where x is the input and y is the output.

deciding whether an image $\boldsymbol{x}$ contains a dog or a cat. The reason it is called a *deep network* is because f is often a long chain of function compositions, e.g., $f = f^{(n)} \circ \ldots \circ f^{(2)} \circ f^{(1)}$. The functions $f^{(1)}, \ldots, f^{(n)}$ are called *layers* and n is the *depth* of the network. Each layer performs a computation on the output of the previous layer, hence the name *feedforward*. A feedforward network is best visualized as a graph[1], the so-called *computational graph*. Figure 3.1 shows an example of a small feedforward network. A node in the graph represents an output- or input variable to one of the functions $f^{(i)}$, and the collection of all output nodes for one function is called a *layer*. The edges between nodes represent information flow, i.e., which outputs of layer j depend on the outputs of layer i.

Feedforward networks build the foundation of deep learning. Why? Because in general, f is unknown and must be learned to solve a specific task. The practical approach to finding a suitable f is to consider a family of functions $f_{\boldsymbol{w}}$ defined by the *parameters* $\boldsymbol{w}$, also called *weights*. For example, the layer i in the network might be a linear function $f^{(i)}(\boldsymbol{x}) = \boldsymbol{W}\boldsymbol{x}$ where the parameters of this layer are the elements of the matrix $\boldsymbol{W}$.

The collective of all functions in the family is called the *model*. To solve the task, e.g., classifying dogs and cats, one has to *train* the model. In other words, one has to search through the space of all parameters $\boldsymbol{w}$ to find the ones that perform the best. The performance is measured with the *loss function* $\mathcal{L}(f_{\boldsymbol{w}}(\boldsymbol{x}), \boldsymbol{y}) \in \mathbb{R}$ that compares the prediction $\hat{\boldsymbol{y}} = f_{\boldsymbol{w}}(\boldsymbol{x})$ with the *ground truth* $\boldsymbol{y}$ and gives a score how close they are. The ground truth comes from the dataset, which is the most important resource of knowledge for training a model. The dataset contains many samples $\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(N)}$ each with a ground truth label $\boldsymbol{y}^{(1)}, \ldots, \boldsymbol{y}^{(N)}$, e.g., each dog image in the dataset has a label "dog" and each image with a cat has the label "cat".

The loss function is not only used to evaluate the performance of the model, but to find optimal parameters for the model as part of the training. In general, the objective is to minimize the loss function on the data, that is

$$\boldsymbol{w}^* = \underset{\boldsymbol{w}}{\operatorname{argmin}} \sum_{i=1}^{N} \mathcal{L}(f_{\boldsymbol{w}}(\boldsymbol{x}^{(i)}), \boldsymbol{y}^{(i)}). \tag{3.10}$$

[1]More precisely, it is a DAG (directed acyclic graph). It does not allow loops.

The resulting function $f_{\boldsymbol{w}^*}$ is a solution that best satisfies the loss function on all the data. How can $\boldsymbol{w}^*$ be found? A common way to minimize the loss is to use gradient descent. It is one of the most general purpose optimization algorithms. Gradient descent can find a local minimum of a function F by iteratively stepping in the opposite direction of the gradient, which can be described by the update rule

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \lambda \frac{\partial F(\boldsymbol{w})}{\partial \boldsymbol{w}}. \tag{3.11}$$

The parameters $\boldsymbol{w}$ approach a local minimum when a suitable learning rate λ is chosen. In the context of deep learning, F is the loss function evaluated over the training data. More precisely, the update rule is

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \lambda \frac{\partial}{\partial \boldsymbol{w}} \sum_{i=1}^{N} \mathcal{L}(f_{\boldsymbol{w}}(\boldsymbol{x}^{(i)}), \boldsymbol{y}^{(i)}). \tag{3.12}$$

One parameter update with gradient descent requires the model to be evaluated for the entire dataset. This is a very expensive operation and in practice it is most often unfeasible. *Stochastic* gradient descent is a modification that performs the update for each sample, which means that the sum in equation 3.12 is dropped and the update rule is simply

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \lambda \frac{\partial \mathcal{L}(f_{\boldsymbol{w}}(\boldsymbol{x}^{(i)}), \boldsymbol{y}^{(i)})}{\partial \boldsymbol{w}}. \tag{3.13}$$

The following steps describe a typical training procedure of a feedforward network.

1. Define model $f_{\boldsymbol{w}}$
2. Initialize parameters $\boldsymbol{w}$
3. Loop over training samples $\boldsymbol{x}^{(i)}$
   (a) Forward operation: $\hat{\boldsymbol{y}} = f_{\boldsymbol{w}}(\boldsymbol{x}^{(i)})$
   (b) Compute loss $\mathcal{L}(\hat{\boldsymbol{y}}, \boldsymbol{y}^{(i)})$
   (c) Backward operation: Backpropagate the gradient of the loss
   (d) Update parameters

The algorithm above trains the model using each sample from the dataset once. A complete run over the dataset is called one *epoch*. To improve the fit of the model to the data further, one can iterate for multiple epochs.

After training it is a good practice to evaluate the model on new data, the test set. This way we can measure how well the model generalizes. It is important that the model performs well on new data because in practice we are often limited in how much data we can collect for training. A model that does not generalize well is not very useful.

Some remarks on the above summary for feedforward neural networks: First, training with samples that have a ground truth label is called *supervised learning*. It is also possible to learn from data where labels are not available. This is called *unsupervised learning*. However, the focus of this thesis lies in the supervised approach. Second, since the feedforward network is a composition of

functions, gradient computation involves the chain rule for differentiation. The *backpropagation* algorithm is an efficient implementation for the chain rule that avoids computing the same subexpressions multiple times. It is the preferred method to compute gradients in a neural network.

3.4 Convolutional Neural Networks

A layer in a feedforward network can theoretically depend on all the outputs of the previous layer. In this case, it is called a *fully connected* layer. An example is the linear layer $f(\boldsymbol{x}) = \boldsymbol{W}\boldsymbol{x}$ with parameters $\boldsymbol{W} \in \mathbb{R}^{m \times n}$ that connects n inputs $\boldsymbol{x} \in \mathbb{R}^n$ with m outputs. An operation like this has a high computational cost, especially when the input is large (e.g. an image). The *convolutional neural network (CNN)* is a special type of feedforward network that has layers where the output nodes are connected only to a few neighboring nodes from the previous layer. The operation on this neighborhood is linear (or affine), and all outputs share the same weight matrix, the *kernel*. The number of connections in the neighborhood is defined by the *kernel size*. The convolution kernel is applied by sliding it over the input. This makes it independent of the input size, which is a very convenient when the inputs to the network are images of varying size/resolution.

Formally, the 2D convolution at location (i, j) on an input $\boldsymbol{X}$ with C channels is defined as

$$\boldsymbol{Y}(i,j) = \sum_{c,n,m} \boldsymbol{X}(c, i \cdot s - n, j \cdot s - m)\boldsymbol{K}(c,n,m), \tag{3.14}$$

where $\boldsymbol{K} \in \mathbb{R}^{C \times N \times M}$ is the kernel with size $N \times M$. Usually, this size is chosen to be quadratic, i.e. $N = M$. Note that the convolution is only applied in the spatial dimensions, and that the weights for each channel can be different. The variable $s \in \mathbb{N}$ in equation 3.14 is the *stride*. A stride greater than one means that outputs are "skipped", i.e., the spatial size shrinks by the factor $\frac{1}{s}$.

There are a number of advantages a convolution layer has over the fully connected layer. The convolution can be applied on arbitrary input sizes, it is computationally efficient due to the limited number of connections to a neighborhood (sparsity), the same kernel is applied to the whole input (weight sharing) and therefore can "detect" a pattern in an arbitrary location (shift-invariance). Lastly, a convolution layer usually applies multiple kernels on the same input, resulting in an output $\boldsymbol{Y}(l, i, j)$ with multiple channels. The slices indexed by l are called *feature maps*.

3.5 Recurrent Neural Networks

So far, we have looked at the feedforward network, which is a function that maps an input $\boldsymbol{x}$ to an output $\boldsymbol{y} = f(\boldsymbol{x})$. It will always compute the same output for the same input, independently of the input-output pairs it has seen before. Now consider a sequence $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T$ of inputs, e.g., a sentence. How can we learn to predict an attribute that depends on the whole sequence, e.g., the sentiment of the sentence? One way is to concatenate the sequence to one vector and use the feedforward network. However, this will limit the sequence

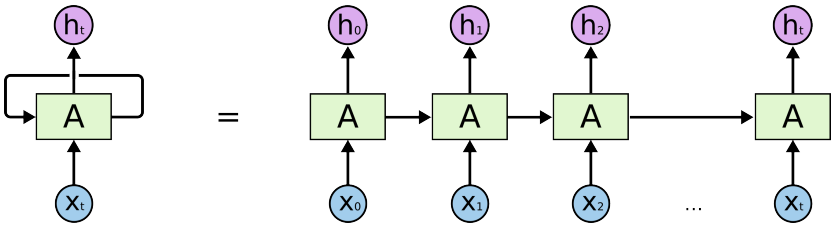


Figure 3.2: Unfolding the RNN. Left: The RNN as a cyclic graph. Right: Unfolded RNN for a number of timesteps. Image courtesy Christopher Olah (colah.github.io).

length to a fixed size and potentially requires more training data. How do we model predictions for a variable sequence length? And what if we also want the output to be a sequence? What if the output sequence needs to have a different length than the input sequence, such as in machine translation?

This is where the *recurrent neural network (RNN)* comes into play. It is a function that maps each vector $\boldsymbol{x}_t \in \mathbb{R}^n$ from the sequence to a hidden state

$$\boldsymbol{h}_t = R(\boldsymbol{x}_t, \boldsymbol{h}_{t-1}), \qquad t = 1, \ldots, T \tag{3.15}$$

with the hidden state $\boldsymbol{h}_{t-1} \in \mathbb{R}^d$ carried over from the previous prediction as a second input, where d is called the *hidden size* of the RNN. The hidden state represents the accumulated knowledge from all previous inputs. This way, the output at each time step t depends on the inputs from $\boldsymbol{x}_1$ to $\boldsymbol{x}_{t-1}$. Figure 3.2 shows the RNN as a looping component. When unfolded, one can see the information flow from one state to the next. Note that it is possible to feed arbitrary sizes of sequences to the RNN.

The vanilla implementation of an RNN uses an affine transformation followed by a non-linear function:

$$\boldsymbol{h}_t = \tanh\left(\boldsymbol{W}\begin{bmatrix}\boldsymbol{x}_t \\ \boldsymbol{h}_{t-1}\end{bmatrix} + \boldsymbol{b}\right) \tag{3.16}$$

The weight matrix $\boldsymbol{W}$ has dimensions $d \times (n + d)$. Sometimes it is desirable to have an output that has a different size than the hidden state, e.g., as for classification. In this case one can apply a second affine layer

$$\boldsymbol{y}_t = \boldsymbol{V}\boldsymbol{h}_t + \boldsymbol{c} \tag{3.17}$$

to obtain the prediction $\boldsymbol{y}_t$. Otherwise we define the output as $\boldsymbol{y}_t = \boldsymbol{h}_t$.

Training an RNN is not much different from a feedforward network. The weights $\boldsymbol{W}, \boldsymbol{V}, \boldsymbol{b}$ and $\boldsymbol{c}$ are updated in the same fashion as for feedforward networks using gradient descent on the loss function. Say we define a criterion $\mathcal{L}_t(\hat{\boldsymbol{y}}_t, \boldsymbol{y}_t)$ for the loss between prediction and ground-truth at each time step t. Then, the total loss for the sequence is simply

$$\mathcal{L}(\{\hat{\boldsymbol{y}}_1, \ldots \hat{\boldsymbol{y}}_T\}, \{\boldsymbol{y}_1, \ldots \boldsymbol{y}_T\}) = \sum_{t=1}^{T} \mathcal{L}_t(\hat{\boldsymbol{y}}_t, \boldsymbol{y}_t). \tag{3.18}$$

This loss is used to compute gradients for updating the model parameters similar to feedforward networks.

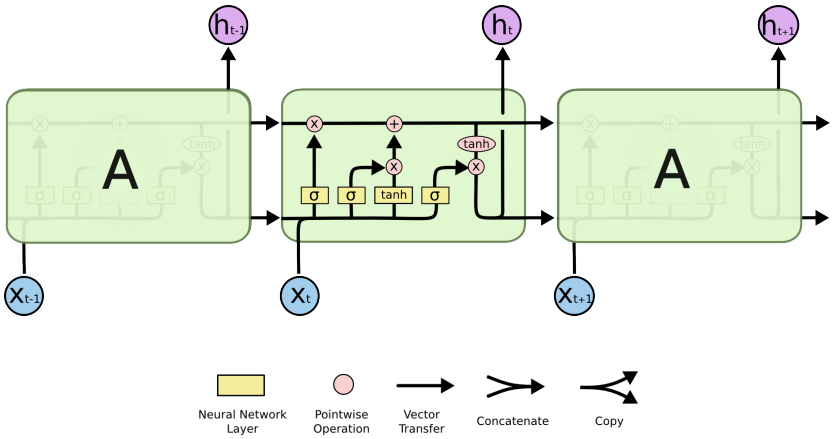


Figure 3.3: Unfolded LSTM cell. Image courtesy Christopher Olah (colah.github.io).

3.6 Gradient Computation for RNNs

The gradient computation for the RNN is almost the same as for the feedforward network and the backpropagation algorithm can be applied by unfolding the RNN as shown in figure 3.2. However, there is a problem with the gradient computation in the above version of the RNN (Pascanu et al. [2013], Bengio et al. [1994]). For long sequences, the gradient norm can become very small such that the parameter updates have no influence and therefore the RNN is not able to learn long-term dependencies. To see this, we can write down the full equation for the gradient:

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{w}} = \sum_{t=1}^{T} \frac{\partial \mathcal{L}_t}{\partial \boldsymbol{w}} = \sum_{t=1}^{T} \sum_{k=1}^{t} \frac{\partial \mathcal{L}_t}{\partial \boldsymbol{y}_t} \frac{\partial \boldsymbol{y}_t}{\partial \boldsymbol{h}_t} \left(\prod_{j=k+1}^{t} \frac{\partial \boldsymbol{h}_j}{\partial \boldsymbol{h}_{j-1}} \right) \frac{\partial \boldsymbol{h}_k}{\partial \boldsymbol{w}} \tag{3.19}$$

In the above equation, we have the product term that causes the problem when the sequence is very long. Note that the partial derivatives are Jacobian matrices when we take the derivatives of a vector-valued function with respect to a vector. Due to many multiplications, the gradient values can drop or increase exponentially.[2] The case of exploding values is less problematic since the gradient can be manually rescaled to a maximum norm. However, in the case of a vanishing gradient, the problem is more serious. A rescaling would be numerically unstable due to the norm being very close to zero.

3.7 The Long Short-Term Memory

The *Long Short-Term Memory (LSTM)* is a version of the RNN that was designed to overcome the vanishing gradient problem, and it is also used in this thesis. As shown in figure 3.3, it has four gates (in yellow) and an additional cell state variable that is carried over from one time step to the next. The

[2] A more complete analysis is provided in the work of Pascanu et al. [2013].

recurrence can be formulated as a function

$$(\boldsymbol{h}_t, \boldsymbol{c}_t) = R(\boldsymbol{x}_t, \boldsymbol{h}_{t-1}, \boldsymbol{c}_{t-1}), \qquad t = 1, \ldots, T. \tag{3.20}$$

More precisely, the hidden state and cell state are computed as follows:

$$\begin{aligned} \begin{bmatrix} \boldsymbol{i} \\ \boldsymbol{f} \\ \boldsymbol{o} \\ \boldsymbol{g} \end{bmatrix} &= \begin{bmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{bmatrix} \left(\boldsymbol{W} \begin{bmatrix} \boldsymbol{x}_t \\ \boldsymbol{h}_{t-1} \end{bmatrix} + \begin{bmatrix} \boldsymbol{b}_i \\ \boldsymbol{b}_f \\ \boldsymbol{b}_o \\ \boldsymbol{b}_g \end{bmatrix} \right) \\ \boldsymbol{c}_t &= \boldsymbol{i} \odot \boldsymbol{g} + \boldsymbol{f} \odot \boldsymbol{c}_{t-1} \\ \boldsymbol{h}_t &= \boldsymbol{o} \odot \tanh(\boldsymbol{c}_t) \end{aligned} \tag{3.21}$$

The matrix $\boldsymbol{W} \in \mathbb{R}^{4d \times (n+d)}$ contains the weights for the input- and hidden state transformations before each gate. It can be broken down into four parts:

$$\boldsymbol{W} = \begin{bmatrix} \boldsymbol{W}_i \\ \boldsymbol{W}_f \\ \boldsymbol{W}_o \\ \boldsymbol{W}_g \end{bmatrix} \tag{3.22}$$

Compared to the vanilla RNN (eq. 3.16), there are three additional gates that control information flow with a sigmoid activation. If the sigmoid output is close to one, information is let through, and if it is close zero, no information passes the gate.

The first is the input gate $\boldsymbol{i}$. It controls how much information should be collected from the current input and hidden state. The second is the forget gate $\boldsymbol{f}$, which will learn what information should be discarded from the previous cell state. Finally, the output gate $\boldsymbol{o}$ regulates the amount of information that goes to the output $\boldsymbol{h}_t$.

The main reason why the LSTM can counteract the gradient vanishing problem is because of the forget gate in the cell state update $\boldsymbol{c}_t = \boldsymbol{i} \odot \boldsymbol{g} + \boldsymbol{f} \odot \boldsymbol{c}_{t-1}$ in equation 3.21. Note that when we take the derivative of $\boldsymbol{c}_t$ with respect to $\boldsymbol{c}_{t-1}$, the only term that remains is the forget gate output. For each element in the cell state, the forget gate directly regulates the decay of gradient over time.

3.8 Training RNNs

There are several ways an RNN can be trained. First of all, one can distinguish the type of input-output relationship for RNNs.

Input to Output Mapping In section 3.5 it was assumed that for each time step t there is an input $\boldsymbol{x}_t$ and an output $\boldsymbol{h}_t$. This is the most general setup for an RNN and we refer to it as *many-to-many*. For some tasks, it might not be suitable to have an input at each time, or an output at each time. On the other hand, the RNN is not a feedforward network in the time axis and thus does not allow a modification to remove inputs or outputs at certain time steps. However, this does not mean that one has to use all outputs for computing loss and gradients. For example, if we wish to predict the sentiment of a sentence, we can feed each word to the RNN and and ignore all outputs except the last

one and use that for backpropagation of the gradient. We refer to this method as *many-to-one*. If there is only one input available, but the output must be a sequence (*one-to-many*), one possibility is to simply feed the same input for all time steps. An example for this method is image captioning, where the input is an image and the output is a description of the image in form of a sentence (Karpathy and Fei-Fei [2015]). Finally, the *one-to-one* setting is a combination of the two above.

Signaling Output Instead of ignoring certain outputs and only using the ones needed, an alternative is to use a special input symbol $\boldsymbol{e}$ to explicitly signal that an output should be written. This special token has to be encoded with the same dimensionality as the other inputs. Its numerical value can be fixed before training (as a hyperparameter) or it can be part of the model parameters and be learned. For example, in the sentiment analysis example from before, the token can be used as an end-of-sequence symbol to mark the end of a sentence. In general, the symbol does not necessarily have to be used at the end of a sequence. It can be inserted whenever an output is needed.

Truncated Backpropagation As discussed in section 3.6, gradient computation over long sequences can cause vanishing or exploding values that hinder the learning process during training. The LSTM was designed to addressed this issue, but backpropagation over very long sequences is still computationally expensive. As a compromise, in order to save memory and reduce computation, one can choose to backpropagate only a certain amount of time steps. This is called *truncated backpropagation through time (TBPTT)*. Say the sequences have a length of T and the number of time steps to backpropagate is $\tau \leq T$. To compute the gradient only back to $T - \tau$, we can replace the start indices for t and k in the sums of equation 3.19 with $T - \tau$. Obviously, this is only an approximation of the full gradient over all time steps. The RNN is not forced to learn dependencies back in time longer than τ steps and thus we can not expect it to do so. TBPTT is used for all experiments in chapter 5 as well as for the toy example in section 3.9.

A specific application of TBPTT is for training on one continuous data stream instead of many shorter sequences. For example, if the goal is to train an RNN to generate natural text, it needs to remember the context over multiple sentences. To do this, one can concatenate the data from all text documents to one very long sequence. During training, we can forward subsequences consecutively and backpropagate τ steps. After each backpropagation, the hidden state $\boldsymbol{h}_t$ (or $(\boldsymbol{h}_t, \boldsymbol{c}_t)$ in the case of LSTM) is carried over to the next subsequence as initial state. This way, there is always information available from the past even though the gradients are never propagated to the very beginning of the sequence. An application of this training procedure is presented later in chapter 5.

Ground Truth How ground truth is used depends heavily on the task at hand. In the aforementioned example for generating text, the RNN predicts the next word given the current word and the context (hidden state). Therefore, the ground truth data is the same as the input data except that it is shifted in time, e.g., $\boldsymbol{y}_t = \boldsymbol{x}_{t+1}$. In this example, not only do the input and output have the

same dimensionality (words are encoded as vectors), but the semantic meaning is also the same. This is of course not true for all applications of RNNs. In this thesis, the inputs are camera images (thousands of color values) and the outputs are poses (usually 6D or 7D vectors). The semantic gap is large since the pose is not directly encoded in the individual image, but in the motion between the frames. The pose is always relative to the first frame in the sequence. The RNN has to learn this relationship directly from the ground truth, not from the input.

3.9 Toy Example: Memorizing Digits

The intention of this toy experiment is to familiarize the reader with the LSTM. Although the following example has nothing to do with VO, it is important to understand the basic properties of the LSTM, the key building block that we will use later in a larger model for VO. The section gives a rough outline of how an LSTM model is defined, trained and evaluated. The main questions that are investigated are: Can the LSTM memorize the past and use this knowledge for future outputs? And how far back in time can it remember? What is the influence of the hyperparameters? These are the main questions that are investigated in this section.

The Task The goal is to train an LSTM that remembers the last m digits of a random sequence of binary digits $\boldsymbol{x}_t \in \{0, 1\}$. Thus, at time t we would like the output to be the digit $\boldsymbol{x}_{t-m}$ that was fed m time steps before. For example, here are two sequences of zeros and ones shifted by $m = 3$:

```
...01110101 10...
...00101110 10...
```

The first line is the input sequence and below is the desired output sequence. The LSTM reads the digits one by one to store them in memory and at the same time it outputs the digit from memory at time $t - 3$ as highlighted by the colors above. In order to tackle this machine learning problem, we will define three key components:

1. The model
2. The performance measure
3. The optimization procedure

The Model We begin by defining the model. The first part of it is the LSTM itself as it was defined in equations 3.20 and 3.21. Since the output can only be two numbers (zero or one), we will treat the problem as a classification task. To do this, we add an affine layer to shrink the size of the LSTM output $\boldsymbol{h}_t$ down to a two-element vector

$$\boldsymbol{z}_t = \boldsymbol{V}\boldsymbol{h}_t + \boldsymbol{c}. \tag{3.23}$$

The last layer is the *softmax* operation

$$\text{softmax}(\boldsymbol{x})_j = \frac{e^{x_j}}{\sum_{k=1}^{K} e^{x_k}}, \qquad j = 1, \ldots, K \tag{3.24}$$

Layer	Variable	Input size	Output size	Parameters
LSTM	$\boldsymbol{h}_t$	1	d	$4d(d+1)+4d$
Affine	$\boldsymbol{z}_t$	d	2	$2d+2$
Softmax	$\boldsymbol{p}_t$	2	2	0

Table 3.2: A simple model to memorize a binary sequence. The variable d is the hidden size of the LSTM.

which is applied to $\boldsymbol{z}_t$, where $K = 2$ in this example. The softmax layer forces the output to be a probability vector, which means that each entry is in the range $[0, 1]$ and all elements sum to one. Therefore, the output vector contains two probabilities,

$$\boldsymbol{p}_t = \begin{bmatrix} P(\boldsymbol{o}_t = 0 \mid \boldsymbol{x}_t, \boldsymbol{h}_{t-1}, \boldsymbol{c}_{t-1}) \\ P(\boldsymbol{o}_t = 1 \mid \boldsymbol{x}_t, \boldsymbol{h}_{t-1}, \boldsymbol{c}_{t-1}) \end{bmatrix}.$$

The output digit $\boldsymbol{o}_t$ must be chosen according to the highest probability. A summary of the full model is shown in table 3.2.

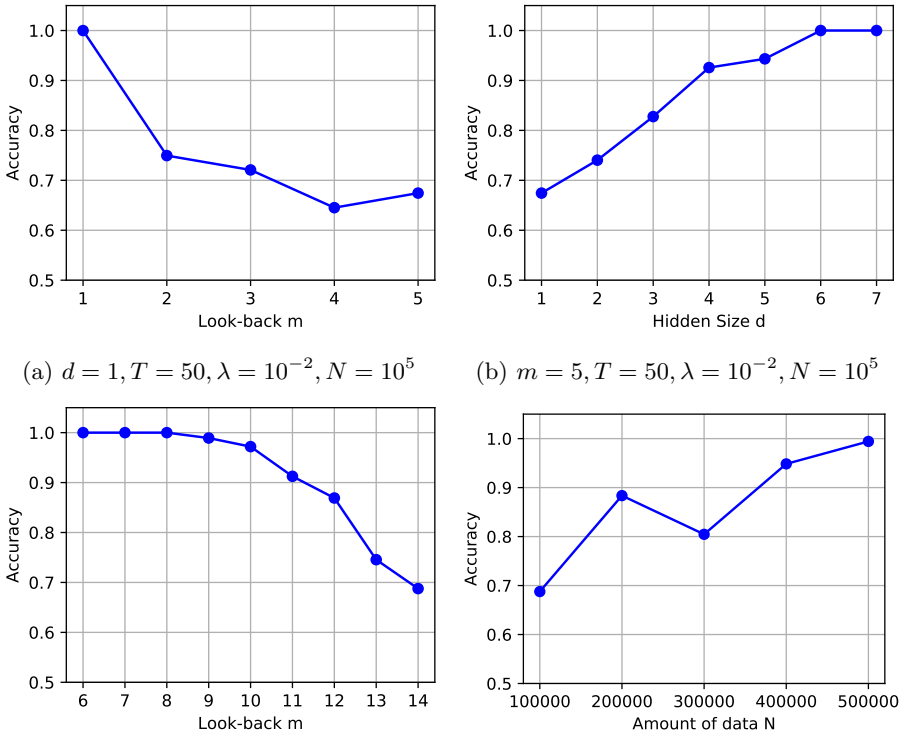
The Performance Measure Next, we have to define a suitable loss function. For classification with softmax as the last layer, it is preferable to use the negative log-likelihood

$$\mathcal{L}(\boldsymbol{x}, y) = -\log(\boldsymbol{x}_y) \tag{3.25}$$

as a loss function. Here, the first argument of $\mathcal{L}$ would be replaced with the output $\boldsymbol{p}_t$ of the network and the second argument is the ground truth label. Since we are predicting the digit from m time steps ago, the ground truth directly comes from the input sequence, and therefore the label y_t at time t is the input $\boldsymbol{x}_{t-m}$. By minimizing this loss, the probability for the correct class is maximized. As described in equation 3.18, the total loss for the entire sequence is the sum over the individual losses.

The Optimization Procedure Because this experiment is completely artificial, it is possible to generate as much data as needed. Instead of generating many smaller sequences, one can generate a single large sequence or sample the digits on-the-fly. In order to save memory and reduce computation, we apply TBPTT. Starting from the beginning of the sequence, T inputs are fed to the LSTM followed by the loss computation. The gradient computation is performed for exactly T steps back in time and the weights are updated accordingly. Continuing with this pattern, the next T inputs are fed for the next update and so on. In each step, the hidden state is carried over from the previous digit. This way, there is always information available from the past even though the gradients are never propagated to the very beginning of the sequence.

Results for Binary Sequences Now that task, model and optimization procedure are defined, the model can be trained and evaluated. For all experiments, we train the model and test on new unseen data of the same size by computing the accuracy, that is the relative frequency of correctly predicted digits. First, we train and evaluate on the simplest model possible: The LSTM has a hidden



(a) $d = 1, T = 50, \lambda = 10^{-2}, N = 10^5$ (b) $m = 5, T = 50, \lambda = 10^{-2}, N = 10^5$

(c) $d = 50, T = 50, \lambda = 10^{-3}, N = 10^5$ (d) $m = 14, d = 50, T = 50, \lambda = 10^{-3}$

Figure 3.4: Accuracy on the test dataset for varying hyperparameters while keeping others fixed. The parameters involved are the hidden size d, look-back m, BPTT T, training set size N and learning rate λ. (a) The hidden size is too small. (b) Effect of increasing the hidden size. (c) Only increasing the hidden size is not enough. (d) Increasing the amount of training data.

size of one ($d = 1$). The result is shown in figure 3.4a. We observe that the model can perfectly memorize the digit of one time step in the past ($m = 1$). However, when increasing the look-back the accuracy drops significantly. This problem can be fixed by increasing the hidden size as shown in 3.4b. Based on these observations, it seems that the hidden size d is related to the memory capacity of the network and must always be chosen higher than the look-back m. But this hypothesis does not hold true when we continue to increase m as demonstrated in 3.4c. Figure 3.4d shows that for a higher look-back, we also need significantly more training data.

Extension to Multiple Classes So far we have learned a binary classifier, but we can extend the task to go beyond that to make the problem harder. Instead of having only two symbols 0 and 1, we can choose each digit to be a number $\boldsymbol{x}_t \in \{1, \dots, C\}$. This is essentially a classification problem with C classes. The updated model is shown in table 3.3. It is trained and evaluated for each value $C \in \{1, \dots 15\}$. The blue line in figure 3.5 shows that the accuracy decreases the more classes are used in the sequence. This is due to the more

Layer	Variable	Input size	Output size	Parameters
LSTM	$\boldsymbol{h}_t$	1	d	$4d(d+1)+4d$
Affine	$\boldsymbol{z}_t$	d	C	$Cd+C$
Softmax	$\boldsymbol{p}_t$	C	C	0

Table 3.3: A simple model to memorize a sequence of numbers. This is a generalization of the binary model in table 3.2 where C was equal to two.

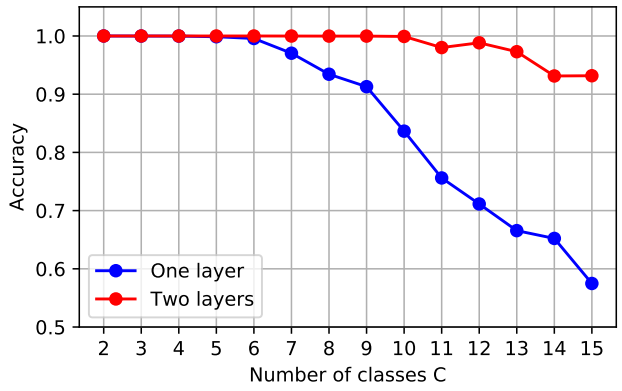


Figure 3.5: Memorizing multiple classes with an LSTM. Performance of single layer LSTM vs. stacked LSTM. Adding more layers helps deal with the higher complexity of the output. Hyperparameters: $m = 5, d = 50, T = 50, \lambda = 10^{-3}, N = 10^5$.

complex mapping between input and output. An LSTM with a single layer can not handle the complexity. As shown in the red line in figure 3.5, adding a second layer boosts the performance.

Chapter 4

Visual Odometry with RNN

4.1 Introduction

The bulk of this chapter and the experiments that follow in chapter 5 are based on the work of Wang et al. [2017]. At the time of writing this thesis, they have not released the source code of their implementation. Therefore, the aim of this chapter is to study and re-implement the ideas presented in their paper and to further expand on it. To reiterate, the task we try to solve in this thesis is Visual Odometry (VO). Given a video, the output of the system should be a sequence of poses describing the camera motion in the video. Wang et al. propose a Deep Learning approach that makes use of an RNN to handle a video of arbitrary length and learn dependencies over time. What follows is a detailed description of the proposed Deep Learning pipeline for VO.

4.2 Datasets

Every machine learning project starts with the collection and assessment of data. Data is what drives machine learning, and often it is also a limiting factor. Generally, the more data, the better the learning process. However, it is not always easy to get the desired amount and quality. For VO, the videos need to be recorded manually, which is very time consuming. Not only that, but in our supervised setting, accurate per-frame pose annotations are required. Therefore we have two main quality requirements for the data in VO: Accurate ground truth pose and realistic image quality. The following subsections describe the different datasets that are used in this thesis.

4.2.1 KITTI

KITTI from the Karlsruhe Institute of Technology (Geiger et al. [2013]) is a dataset that contains many videos captured from the roof of a driving car. Some example images are shown in figure 4.1. Each video frame is labeled with ground truth data such as camera pose and 3D points from a velodyne laser scanner. The camera pose was obtained by combining the data from GPS and an IMU (inertial measurement unit) that was mounted to the car. The dataset has around 43k stereo pairs of images with size 1226×370 pixels captured at a

Figure 4.1: Example images from the KITTI dataset. The pictures show wide open areas, narrow streets, reflections and moving cars. The videos were captured with a camera mounted on top of a car.

frame rate of about 10 fps. For the experiments in this thesis, only the images from the left camera are used. The dataset is divided into 22 sequences, each captured at a different location in the metropolitan area of Karlsruhe, Germany. For the public, the ground truth is only available for the first 11 sequences. The rest of the data is intended to be used for submissions to the KITTI Vision Benchmark Suite[1] online. In the experiments here, only the sequences 0 to 10 are used and divided into training- and test sets, which amounts to around 23k images.

4.2.2 VIPER

The VIPER dataset from Richter et al. [2017] contains a mix of car driving and walking sequences, but all data was generated synthetically from the video game Grand Theft Auto 5 (GTA V) for PC released in April 2015. With around 254k frames it is significantly larger that the KITTI dataset. There is a variety of ground truth data available, including camera pose, semantic class labels and 3D object bounding boxes. The camera poses have been directly extracted from the game engine and are therefore fully accurate, while other data such as the semantic labels were generated in a post-processing step. VIPER is subdivided into training, test, and validation sets with 134k, 70k, and 50k frames respectively. The splits contain diverse scenes at day and night across different scene types such as urban, suburban under various weather conditions. The videos were captured at a frame rate of 15 fps.

4.2.3 GTA V

The GTA V dataset, as presented in figure 4.2, was created specifically for this thesis shortly before the VIPER dataset by Richter et al. was made public. Third party "modding" tools[2] had to be used to extract the camera pose information because the game developers do not intend users to modify the game files and only provide binaries for executing the software. The tool allows one to inject a user defined C++ function to run code in a separate thread next to the game engine with access to variables such as the camera matrix, player-

[1] `http://www.cvlibs.net/datasets/kitti/eval_odometry.php`
[2] `http://www.dev-c.com/gtav/`

Figure 4.2: Example images from different sequences in the GTA V dataset which was recorded for this thesis. The images show wide open- as well as narrow places, reflections on cars and water, complex shadows from foliage and moving objects (people, cars).

and world properties and more. The tool[3] developed for this thesis streams the camera orientation and location to an output file with timestamps for each recorded pose. The video is recorded parallel to the pose with an external program called *NVIDIA ShadowPlay*. Although both the recording of pose and video start synchronously, the rate at which the pose is queried is higher than the frame rate of the recorded video. Hence, the poses written to the output file were interpolated using the timestamps to match the 30 fps video recording.

The dataset has a total of 470k frames of walking in first-person perspective. The sequences contain many factors of variation in form of weather changes (cloudy, sunny, night), dynamic motion of cars on the street or people on the sidewalk, different environments (urban, suburban, rural) and changes in speed (standing, walking, running). Although this dataset contains more video data compared to VIPER, it has only pose annotations.

4.2.4 Preprocessing

Each dataset contains long sequences/videos of thousands of frames over several minutes of recording. Due to the high memory footprint, it is unfeasible to load a complete sequence and feed it to the network. Therefore the sequences are cut into subsequences of smaller sizes. For most experiments here, the sequence size is 100 frames or less. Instead of creating a segmentation of the dataset, this method of extracting subsequences also allows to define an overlap between sequences. This is especially useful when the dataset is small, such as KITTI.

Since resolution and aspect ratio of the images are different between the datasets, they are first proportionally resized to a height of 320 pixels and then

[3] `https://github.com/awaelchli/GTA-V-Camera-Tracker`

the center region of 448×320 pixels is extracted. Fixing the input size across multiple datasets makes it possible to train the same network architecture for different data and make a more accurate comparison. Due to the cropping, the left and right boundaries of the image are removed. An alternative is to resize the images directly without regard to the aspect ratio. This leads to a loss in horizontal resolution which could impact the networks performance for horizontal motions, e.g., a rotation of the camera around the vertical axis. The alternative resizing method was not studied in this thesis, but the impact is expected to be minor.

The ground truth poses also require preprocessing. Each raw sequence has a corresponding text file that contains the 3×4 pose matrices for every frame. These poses are all relative to the coordinate system of the first frame in the sequence. In other words, the coordinate system of the first frame is the world coordinate system of all other frames in the sequence. Because the raw sequences are divided into subsequences, the ground truth poses need to be converted to poses that are relative to the first frame within each subsequence. The formulas in equation 3.2 from chapter 3 are applied here.

4.3 Encoding the Pose

Each dataset comes with per-frame pose annotations, where each pose is defined by a position and orientation. The position always has the same format: It is a vector $\boldsymbol{p} \in \mathbb{R}^3$ describing the location of the camera in three-dimensional space. As discussed before in chapter 3, the orientation is a rotation that can be represented in different ways, each having different benefits and use cases. Since all representations can be converted between each other, it does not matter in which format the training data is available. No matter which representation is chosen, the pose here is always a concatenation of position $\boldsymbol{p}$ and orientation $\boldsymbol{o}$, that is, $\boldsymbol{y} = [\boldsymbol{p}, \boldsymbol{o}]$.

In this work, we investigate two representations for the rotation: Euler angles and unit quaternions. In the case of Euler angles, the orientation is defined by the three angles $\boldsymbol{\varphi} = [\alpha, \beta, \gamma]$, and so the pose is represented as a vector $\boldsymbol{y} = [\boldsymbol{p}, \boldsymbol{\varphi}] \in \mathbb{R}^6$ with six degrees of freedom. When using unit quaternions, the pose is represented as $\boldsymbol{y} = [\boldsymbol{p}, \boldsymbol{q}] \in \mathbb{R}^7$ where $\boldsymbol{q}$ is a unit-norm quaternion determined by a four-dimensional vector. A comparison of how these representations affect the performance is shown in chapter 5.

4.4 The Model

The model is the key to solving the task. By adjusting the weights during the training phase, it learns a high-level representation of relevant information in the input that is needed to produce the desired output. In the case of Visual Odometry, we aim to learn a representation for motion. Specifically, the network has to be able to distinguish between dynamic motion in the scene and camera motion. These and other challenges mentioned in the Introduction chapter are addressed by the two main parts of the model described in the following two sections. The architecture of the full model is shown in figure 4.3.

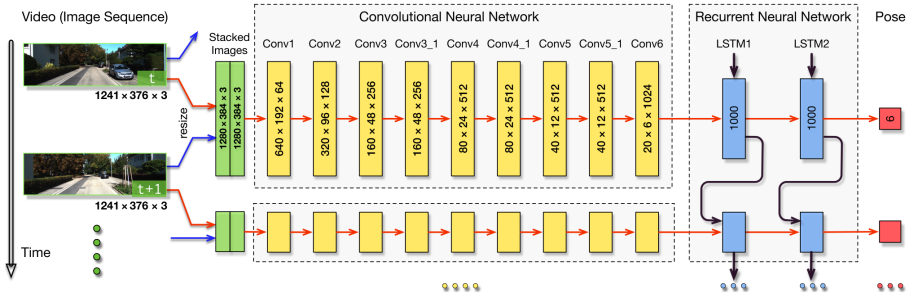


Figure 4.3: Main architecture for Visual Odometry. Green: Input of consecutive video frames. Yellow: Feature extraction layers. Blue: Pose estimation with recurrent connections. The figure was taken from Wang et al. [2017].

4.4.1 Part 1: Feature Extraction

The purpose of the first part of the network is to extract information about motion from two consecutive frames at time t and $t-1$. One way to describe motion between two images is by *optical flow*. It is defined as a vector-valued function $u(\boldsymbol{x}) \in \mathbb{R}^2$ at each point $\boldsymbol{x}$ in the image I_t describing the translation of the point in the next frame such that

$$I_t(\boldsymbol{x}) = I_{t+1}(\boldsymbol{x} + u(\boldsymbol{x})). \tag{4.1}$$

The property in equation 4.1 is called the *brightness constancy constraint*. It means that a particular point can move in the image plane but it does not change its color. In general, this constraint does not hold for every point, e.g., because of occlusion introduced by motion.

Dosovitskiy et al. [2015] describe a deep-learning approach for estimating optical flow. They propose and study two network architectures, *FlowNetS* and *FlowNetC*. The later is computing the inner product between patches of two input frames (correlation) followed by multiple layers of convolutions. FlowNetS on the other hand is the simple version that does not have the correlation layer. Instead, it is only made of convolution layers and ReLUs. The first half of FlowNetS are strided convolutions that gradually increase the number of feature maps. The second half, called refinement, consists of transposed convolution layers that, in reverse, increase the spatial dimension to form the high resolution optical flow maps. In addition, both FlowNet versions make use of skip-connections that allow the features from earlier layers to be added as input to layers in the second half by concatenating the tensors along the feature channel dimension.

In this thesis, the focus is on FlowNetS because it is the only publicly available implementation for PyTorch.[4] FlowNetS is also the choice in the work of Wang et al., which is the basis of the architecture described here. The FlowNetS implementation comes with pre-trained model weights. It is trained on a synthetic dataset of moving chairs rendered into images that serve as a static background. Although this data does not include camera motion, according to Wang et al. the model weights serve as a good initialization that leads to

[4]Pinard [2017] has the source code available on GitHub.

Layer	Kernel size	Stride	Padding	Channels
CONV 1	7×7	2	3	64
CONV 2	5×5	2	2	128
CONV 3	5×5	2	2	256
CONV 3.1	3×3	1	1	256
CONV 4	3×3	2	1	512
CONV 4.1	3×3	1	1	512
CONV 5	3×3	2	1	512
CONV 5.1	3×3	1	1	512
CONV 6	3×3	2	1	1024
CONV 6.1	3×3	1	1	1024

Table 4.1: Architecture of the feature extraction based on FlownetS. Each layer is a convolution followed by a rectified linear unit (ReLU).

faster convergence.

Estimating camera motion solely based on optical flow is not very robust since scene motion is mixed into the optical flow which can significantly vary in magnitude. In order to have a more abstract representation, only the layers in the first half up to "CONV 6.1" are used for feature extraction. All layers in the second half of FlowNetS are dropped. The exact configuration of the remaining layers is shown in table 4.1. Note that the convolutions are applied with a stride and therefore the spatial size is reduced from one layer to the next. At the same time, the number of feature maps (channels) increases up to 1024 in the last layer. Say the input images have a width and height of 448×320 pixels, for example. Then the output tensor contains 1024 feature maps of size 7×5. These activations are a coarse and abstract description of the motion seen in the input and do not directly imply the per-pixel optical flow. To emphasize again, we do not directly make use of optical flow since we are interested in separating camera motion from scene motion. This has to be done on a higher level of abstraction. The remaining task is now to take these coarse features and map them to the camera pose.

4.4.2 Part 2: Pose Estimation

The key component of the second part is the LSTM that maps the motion features to a hidden state. Due to the recurrent connections, the hidden state does not only encode the pose at the current time step, but also a history of poses from previous time steps. To be clear, it is not obvious that the LSTM will actually learn a representation for the past and use that information to estimate the pose because nothing is explicitly forcing the network to do so. The question whether or not the recurrent connections make a significant contribution will be addressed in chapter 5.

As opposed to convolution layers, the LSTM is defined in terms of matrix operations and hence the size of the input to the LSTM is not variable. This also implies that there is a limit on the size of the input images. As described in section 4.2.4, the images are rescaled to 448×320 pixels, which leads to a tensor of size $1024 \times 7 \times 5$ after feature extraction. This tensor is then reshaped

Layer	Input size	Output size
LSTM 1	35840	1000
LSTM 2	1000	1000
DROP	1000	1000
FC	1000	6 or 7

Table 4.2: Architecture of the recurrent part of the pose network. It consists of a two-layer LSTM with hidden size 1000, a dropout- and fully-connected layer.

to a vector with $1024 \cdot 7 \cdot 5 = 35840$ elements and fed to the LSTM as input. Although the LSTM itself applies multiple operations involving input, hidden state and gate outputs, it can be summarized as one single layer. Moreover, as shown in figure 4.3 we adopt the architecture of Wang et al. and stack two LSTM layers. Each of these recurrent layers has an associated hidden state of size $d = 1000$.

The recurrent layers are followed by dropout and a fully connected layer that reduces the output size of the LSTM to a 6D or 7D pose vector depending on the chosen representation. Table 4.2 summarizes the input- and output dimensions of all layers in the second component of the network. The dropout layer is a special layer that randomly overwrites outputs of the previous layer (in this case it is the LSTM) with zeros. The fraction of outputs that are kept is controlled by a probability $p \in [0, 1]$, and outputs are ignored with probability $1 - p$. By randomly discarding information, the network is forced to learn a redundant representation in order to solve the task. In practice, this leads to less overfitting and reduces the test error. The effect of dropout for VO is studied in chapter 5.

4.5 Training

Starting from the beginning of a video, the hidden state of the LSTM is initialized with zero. Pairs of frames are fed to the CNN that performs the feature extraction as shown in figure 4.4. At each time step t the LSTM outputs a pose estimate $\hat{\boldsymbol{y}}_t = [\hat{\boldsymbol{p}}_t, \hat{\boldsymbol{\varphi}}_t]$ and updates the hidden state $\boldsymbol{h}_t$. The quality of the estimated pose is determined by comparison against the ground truth pose $\boldsymbol{y}_t = [\boldsymbol{p}_t, \boldsymbol{\varphi}_t]$ with the loss function

$$\mathcal{L}_t(\hat{\boldsymbol{y}}_t, \boldsymbol{y}_t) = \|\hat{\boldsymbol{p}}_t - \boldsymbol{p}_t\|_2 + \beta\|\hat{\boldsymbol{\varphi}}_t - \boldsymbol{\varphi}_t\|_2. \tag{4.2}$$

As defined in equation 3.18, the full loss of all estimates in the sequence is the sum of the losses in the sequence, i.e., $\mathcal{L} = \sum_{t=1}^{T} \mathcal{L}_t$. The variable T denotes the number of time steps used for backpropagation. Usually, T is fixed before training, and often it is also the length of the input sequences. It is still possible to feed videos with length greater than T. As shown with a red arrow in figure 4.4, the hidden state can be carried over to the next part of the video instead of resetting it to zero. In chapter 5, Experiments And Results, we compare the two strategies of how the hidden state is used during training.

The loss function in equation 4.2 balances the squared error of position and Euler angles with a hyperparameter $\beta > 0$. The scale of positional change is usually higher than the angular changes for rotation, but in general, the

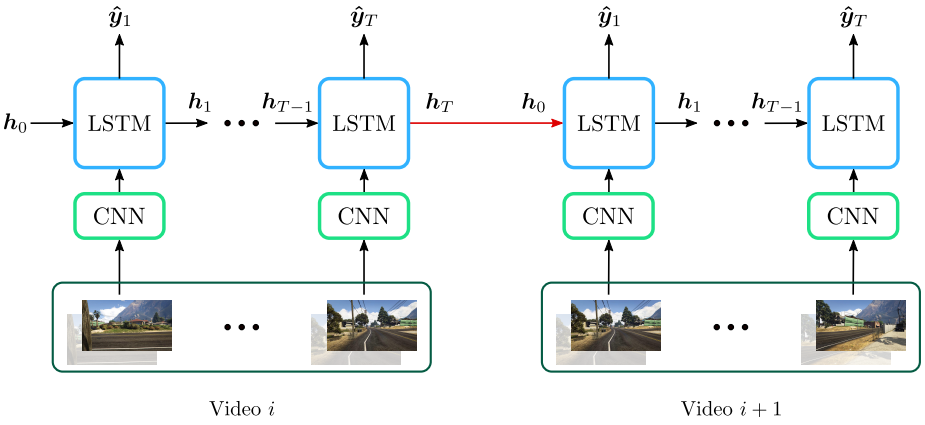


Figure 4.4: Forward operation in time using the combination of a CNN and RNN. When using truncated backpropagation, a long video is divided into multiple parts (i and $i+1$). Between video sequences, one can choose to either transfer the hidden state or reset it.

scale depends on the ground truth provided in the dataset, e.g., the unit of measurements used or the speed of the camera motion. Kendall and Cipolla [2017] find that β requires significant tuning to get acceptable results. Moreover, as an alternative they propose a loss based on the reprojection error of the 3D points that naturally balances rotation and translation and therefore the need for manual balancing is eliminated. However, this thesis does not explore a geometric loss function of this type. The balance value in equation 4.2 is manually chosen and set to $\beta = 10$ by default for training. For evaluation on the test set in all experiments that follow, we use $\beta = 1$. Furthermore, the two terms $\|\hat{\boldsymbol{p}}_t - \boldsymbol{p}_t\|_2^2$ and $\|\hat{\boldsymbol{\varphi}}_t - \boldsymbol{\varphi}_t\|_2^2$ in equation 4.2 are referred to as the translation- and rotation errors respectively.

Alternatively, for experiments where the rotations are represented by quaternions, the loss is defined as

$$\mathcal{L}_t(\hat{\boldsymbol{y}}_t, \boldsymbol{y}_t) = \|\hat{\boldsymbol{p}}_t - \boldsymbol{p}_t\|_2 + \beta(1 - (\hat{\boldsymbol{q}}_t \cdot \boldsymbol{q}_t)^2) \tag{4.3}$$

where $\cdot$ is the inner product of the two unit quaternions. The output quaternion of the network is not constraint to have unit-norm and thus, the output needs to be normalized in an additional step before applying the loss function. The inner product between the two unit quaternions is equal to the cosine of the angle between the two vectors representing the quaternions on the 4D unit sphere. The square in equation 4.3 is necessary to eliminate the sign, because the two quaternions $\boldsymbol{q}$ and $-\boldsymbol{q}$ represent the same rotation. Alternatively, the absolute value $|\hat{\boldsymbol{q}}_t \cdot \boldsymbol{q}_t|$ could also be used for this purpose.

The optimizer used to train the model is Adam (Kingma and Ba [2014]). Adam (Adaptive Moment Estimation) is an extension of the gradient descent method described in chapter 3. It uses adaptive learning rates for each individual parameter in the network. The learning rates are updated based on an exponentially decaying average of past gradients and squared gradients.

The Adam optimizer is applied independently to each part of the network with different initial learning rates $\lambda_1 = 10^{-4}$ and $\lambda_2 = 10^{-3}$. λ_1 is the learning

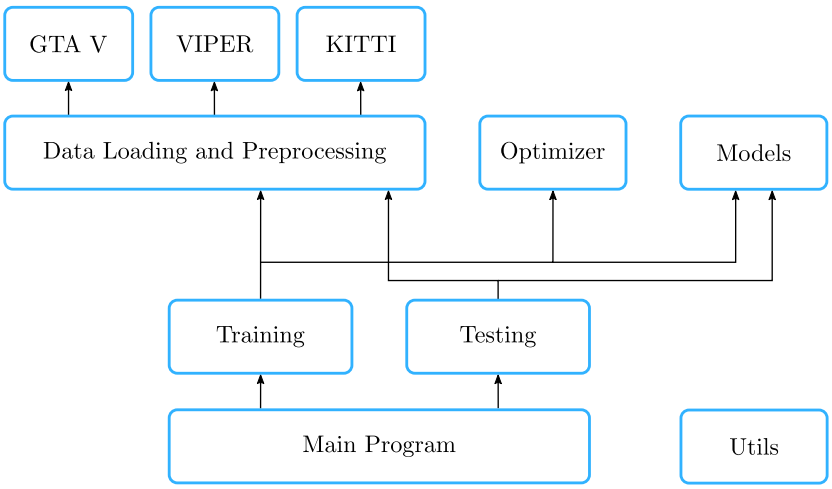


Figure 4.5: Structure of the implemented software. The main program is broken down into training and testing. Separate data loading classes for each dataset take care of converting the data into the desired format.

rate for the part that is initialized with weights from the pre-trained FlowNetS. It is chosen to be smaller than λ_2 because the transferred weights are expected to be already close to optimal values and can be fine-tuned with a smaller learning rate.

The choice of Adam over other optimizers that use adaptive learning rates (e.g. Adagrad, RMSProp, etc.) is rather arbitrary. The main reason why it was chosen in this thesis is to eliminate the need for manual scheduling of learning rate decay over time. Without reducing the learning rate, there is a high risk of an oscillation as the parameters approach a local minimum, and this would lead to a slower convergence.

4.6 Implementation

The model and training procedure described above is implemented in Python using the deep learning framework PyTorch[5]. This framework comes with a rich tensor library with GPU support for all tensor operations. PyTorch differs from other frameworks like TensorFlow, Caffe, or Theano in the sense that it builds the computational graph dynamically at runtime. This flexibility allows for dynamic control of computational flow while training or testing the network.

Figure 4.5 shows the structure of the code developed for this thesis. The main program is split into training and testing. It has the responsibility to parse and initialize parameters provided by the user, load and save checkpoints, and create a folder structure for output files like plots and log-files. The training and testing code accesses data loading classes and model classes depending on which experiment is conducted. Adding a new dataset only requires the implementation of a class for loading and converting the data to the desired format.

[5] `http://pytorch.org`

Chapter 5

Experiments And Results

This chapter presents all experiments conducted on the deep learning model that was introduced in chapter 4. The results shown here give useful insights that supplement the work of Wang et al. [2017].

5.1 Dataset Size and Dropout

We start by training the proposed model on the KITTI dataset on small subsequences of 25 frames. Initially, the sequences are divided without overlap, that is, each video frame is part of exactly one subsequence. In addition, to simulate a dataset with more subsequences, the overlap is set to 20 frames (80%). This means that for each subsequence there is another one that differs by 5 frames. Table 5.1 compares the test error for models trained with and without overlap. After the same number of epochs, the test error is about 13% lower for the model trained on overlapping sequences with a decrease of 21% for rotation and 12% for translation. The same experiment is repeated by training on sequences of 100 frames with overlap 20 vs. overlap 80. The translation error decreases by almost 46%, however, the rotation loss increases. A similar observation can be made when using dropout on the LSTM output (before the fully-connected layer), as seen in the second last row of table 5.1.

For the last experiment in table 5.1, the sequence length grows by an increment of one frame every epoch, starting from 20 frames and stopping at 100 frames after 80 epochs. As we can see, this method of training in combination with the dropout gives the best overall loss on the test set. A possible explanation for this observation is that the learning process is faster for short sequences because the LSTM requires a less complex memory mapping in order to remember the coordinate transforms from earlier time steps, including the first one. A gradual refinement of such a mapping may be less challenging for the optimization as opposed to learning it directly from long sequences.

Although the numbers in the table can be used to compare the different experiments, they are not very insightful by themselves. From the sum of squared differences alone, it is not possible to understand how the error behaves across the sequence. For better visualization, we can compute the error at regular distance intervals along the estimated path. Figure 5.1 shows translation and rotation errors for path distance on subsequences of the KITTI sequence 10. The

Experiments			Test error		
Length	Overlap	Dropout	Total	Rotation	Translation
25	0	0	19.0293	0.4981	18.5312
25	20	0	16.5471	0.3913	16.1558
100	20	0	529.9459	39.8967	490.0493
100	80	0	313.1278	47.8437	265.2841
100	5	0.5	376.7666	43.1578	333.6088
$20 \rightarrow 100$	5	0.5	289.8579	42.0427	247.8152

Table 5.1: Experiments on KITTI: Overlap and dropout. Shown is the test error on sequence 10 for different overlap and dropout during training. The evaluation on the test set is performed on sequences of the same size as seen during training, and without overlap. In the last row, the sequence length grows by one frame every epoch. Each model was trained with default parameters for 100 epochs, except the last one which was trained for only 80 epochs.

relative rotation error is the angle of rotation around the axis corresponding to the relative rotation between estimated- and ground truth pose.

Discussion From these experiments, we can conclude that sequence overlap and dropout have a positive impact on the generalization of translation estimates. Especially when training data is scarce, the overlap method can be utilized to artificially increase the dataset and reduce the error on the test data.

5.2 The Problem on Long Sequences

In the experiments before, the model was always tested on subsequences with the same length as during training. However, in a real-world application, we would like to test the model on sequences with an arbitrary number of frames, and potentially the test sequences are much longer than the ones seen during training. Table 5.2 shows an experiment where the sequence length between training set and test set changes. The problem: The model trained on sequences with 25 frames does not generalize to sequences of 100 frames. This is a very undesired result, after all, the recurrent part of the network was designed with the intention to handle arbitrary input sizes. From the results in table 5.2 it appears that the network is able to memorize the sequence length and therefore overfits to the specific length it is trained on. The attempt to counteract this behavior with sequences of random size failed. The network would still overfit to the average length of sequences seen during training. Figure 5.2 visualizes the camera path on a test sequence of 100 frames. One can observe that the estimated path quickly diverges after 25 frames. However, when trained on sequences with 100 frames, the estimation is more accurate.

Discussion The experiments in this section reveal a serious problem. One of the main purposes of the LSTM is to handle arbitrary input length, and yet, in the scenario we have so far it is not able to do so. In the general application of

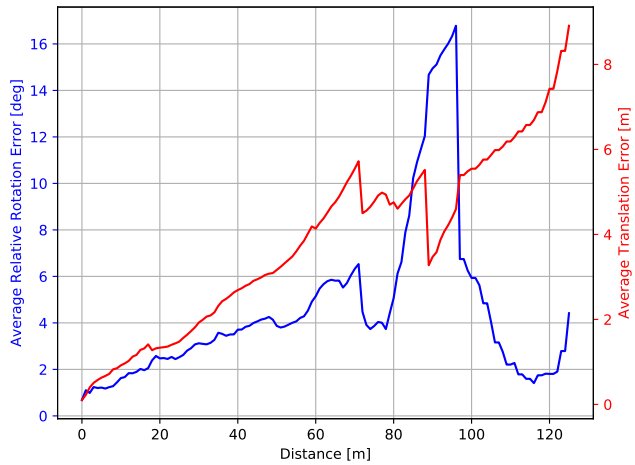


Figure 5.1: Experiments on KITTI: Average rotation- and translation error vs. path distance. The relative rotation- and translation error is evaluated at equal intervals along the subsequences of KITTI sequence 10. The errors at the same distance are averaged over all sequences of that length.

#Frames		**Test error**		
Training	Test	Total	Rotation	Translation
25	25	16.5471	0.3913	16.1558
25	100	1060.7528	37.6196	1023.1332
100	100	529.9459	39.8967	490.0493

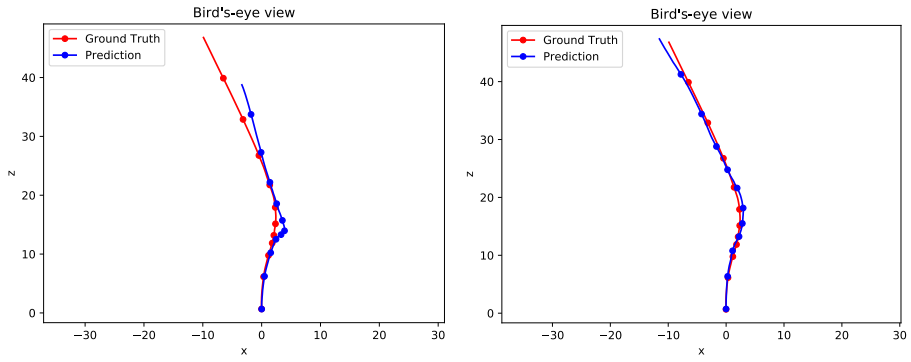
Table 5.2: Experiments on KITTI: Testing on longer sequences. When testing on sequences with more frames, the error becomes extremely high compared to the model trained on 100 frames.

VO, it is not realistic to assume a maximum length for a video and to train the model only for that length.

5.3 Training with Incremental Poses

One potential reason for why the model is overfitting to a certain length could be semantic difference between input and output. Since input at each time step is a pair of consecutive images in the video, the observed motion (by the LSTM) is relative to the current time step. On the other hand, the output pose is forced to be in the global coordinate system given by the very first video frame. This difference could potentially cause confusion and lead to the overfitting problem.

To address the issue, we convert the ground truth poses to incremental poses using the formula from equation 3.2 by setting $j = i - 1$. Table 5.3 shows the results of training with incremental pose. At test time, the output poses are converted back into the global coordinate system for evaluating the loss on the test set. This allows for a direct comparison with the previous results from



(a) (b)

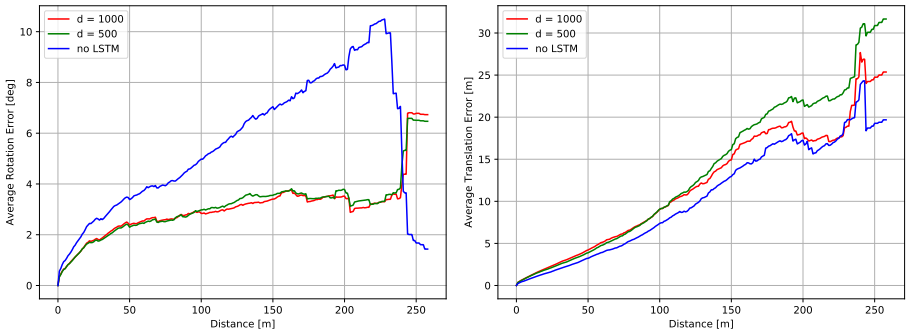
Figure 5.2: Training and testing on different sequence length. Two models tested on a KITTI subsequence of 100 frames when trained on sequences of (a) 25 frames and (b) 100 frames. The markers in the plot are shown every ten frames. The scale of the axes is in meters.

Experiments			**Test error**		
Length		Dropout	Total	Rotation	Translation
Training	Test				
25	25	0	22.0498	0.5642	21.4856
25	25	0.5	21.0171	0.4957	20.5214
25	100	0	335.5754	16.5476	319.0277
25	100	0.5	301.5090	22.9736	278.5354

Table 5.3: Experiments on KITTI: Training with incremental poses. Three variants are tested: LSTM, LSTM with dropout layer, and LSTM replaced with fully-connected layer. Shown is the loss on the test set with incremental poses converted to global poses.

table 5.2. From these results we can see that the error is significantly lower when testing on long sequences. Since the only difference in the experiments is the format of the pose, it can be concluded that training with incremental pose is the preferred way. In the paper of Wang et al. [2017], it is never explicitly stated which coordinate system is used. The results in this thesis suggest that Wang et al. have also used incremental poses in their experiments. Qualitative results for motion estimation on the KITTI test data are presented in figure 5.6. They show how the system handles scenes with moving objects, sharp turns and change of speed.

The new approach of training the network with incremental poses raises an important question. What exactly is the contribution of the LSTM? Since the pose is not global anymore, the LSTM is not forced to keep track of the accumulation of pose changes from the past frames. To investigate this question, we conduct an experiment where the LSTM is replaced with a fully-connected layer followed by a ReLU. The previous affine output layer is kept, making it a total



(a) Rotation (b) Translation

Figure 5.3: Comparison of the recurrent and non-recurrent models on the VIPER test dataset. The recurrent versions have an overall lower rotation error, but they are worse translation error compared to the non-recurrent model.

Experiments		**Incremental Pose**		**Global Pose**	
LSTM	Hidden size	Rotation	Translation	Rotation	Translation
✗	–	0.2162	7.3836	6.7071	241.1611
✓	1000	0.1625	7.4642	4.8312	285.5663
✓	500	0.1581	7.5345	4.3250	286.7348

Table 5.4: Experiments on VIPER: The effect of replacing the LSTM. All models are trained for 100 epochs and tested on sequences of 100 frames with both incremental- and global pose. The recurrent versions perform best in terms of rotation error, whereas the non-recurrent version has lower translation error.

of two affine layers that regress the pose from the CNN features. This eliminates the recurrence from the network, and all pose estimations are performed solely on consecutive frames. Each output is independent of the previous inputs, therefore it is simply a feedforward network. For these experiments, the VIPER dataset is used instead of KITTI. The table 5.4 compares the non-recurrent model with two recurrent models, one with a hidden size $d = 1000$ and one with $d = 500$. Both recurrent versions operate only with one LSTM layer as opposed to two as before. This should make for a fairer comparison. Alternatively, one could have chosen to insert two affine layers instead of one and compare with a two-layer LSTM. The test errors in table 5.4 show that the recurrent models perform better for pose, but worse for translation compared to the non-recurrent version. Additionally, in figure 5.3a we observe that the rotation error at long distances is significantly higher for the non-recurrent model.

Discussion In this second set of experiments, we have addressed the issue where the LSTM would overfit to a certain number of frames seen during training. When forced to output incremental poses, the model is able to estimate motion from videos of arbitrary sizes. However, it is still unclear if and to what amount the LSTM uses the hidden state that encodes the past. In fact, as we have shown, a feedforward network (without recurrence) is able to outperform

Hidden state	**Incremental Pose**		**Global Pose**	
	Rotation	Translation	Rotation	Translation
Keep	0.3097	8.5461	12.7551	426.7877
Reset	0.2319	5.7220	9.8600	262.7984

Table 5.5: Experiments on VIPER: State persistence vs. reset. Both models are identical and trained with with the same hyperparameters with the only difference being how the hidden state is handled.

the LSTM in positional error. Based on these observations, one might speculate that the recurrent connection hurts the performance. But this is not a very plausible, because during the optimization process the LSTM would learn to ignore the hidden state by setting the parameters of the cell gates accordingly, if this minimizes the loss. It is possible that the loss function in equation 4.2, a balance between euclidean error of translation and rotation, is not well-suited for VO. The balance factor β is chosen manually with a simple heuristic based on the fraction between expected translation- and rotation error. In a way, the balance factor tells the optimization algorithm how important the accuracy for rotation is compared to the position. It is certainly possible that the choice of β has a significant impact on the networks ability to properly separate rotation and translation from the input. As the scale of the motion between frames heavily depends on the dataset, it is not viable to systematically search through values for β for different datasets. A way to eliminate the balance factor is left for future work.

5.4 Pose Representation and State Transfer

For all experiments so far, the hidden state $\boldsymbol{h}_t$ of the LSTM is reset (re-initialized as zero) before the next video sample is fed because it is assumed that the videos in the dataset are shuffled. A state transfer from one video to the other would not make sense if the two are different. Also, in the case of global pose where each pose is relative to the first frame in the sequence, the hidden state can not be transfered as the coordinate system would change from one video to the next. However, when training on incremental motions between pairs of frames, it is possible to carry the state from one video to the other when the subsequences are kept in the same order as they occur in the original full video. In this case, backpropagation in time is still only performed over the frames in the subsequence, and not to the very beginning.

Table 5.5 compares the test errors of the two forms of training with hidden state. The experiment shows that resetting the state results in lower test errors for both rotation and translation. Additionally, as shown in figure 5.4, the convergence speed on the validation set is greater for the method that resets the state. The training error converges with the same speed for both methods. This experiment gives another indication that the recurrent connection might influence the learning process in a negative way. But the question remains why the optimization does not adjust the weights accordingly such that the hidden state vector is simply ignored.

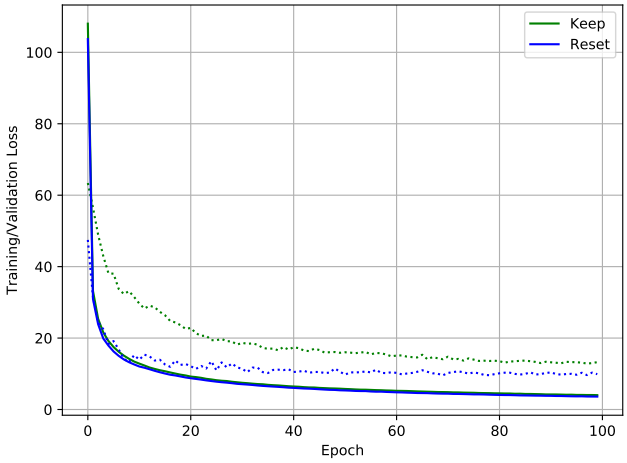
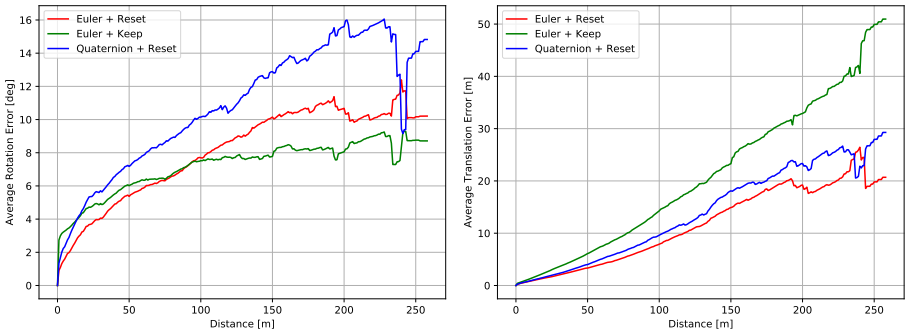


Figure 5.4: Convergence speed comparison: State persistence vs. reset. Two identical models are trained on the VIPER dataset. In one model (blue), the recurrent state is reset after each video sample. In the other case (green), the hidden state is carried over for the next video. The dotted lines show the validation error.

Further experiments show that it is unlikely that the pose representation is the problem. In figure 5.5, combinations of state management and pose representations are compared in terms of rotation- and translation error on VIPER. For the model where the rotations are represented by unit quaternions, the loss from equation 4.3 is used. In order to compare all versions, the rotations are converted to Euler angles at test time and evaluated using the Euclidean loss as before (eq. 4.2). From these results we can make two main observations. First, training with Euler angles (red) results in better performance compared to training with quaternions (blue), both for rotational- and translational error. Second, for the model trained with persistent hidden state, the rotation error is the lowest, but the translation error is highest among the others. Therefore, when comparing all three, there is no clear winner.

Discussion Why do Euler angles work better than quaternions? A possible explanation is that the complexity of the mapping for incremental pose changes is higher for the model where quaternions are used. Small rotational motions in the input result in small Euler angles, and on the other hand, large rotational motions imply larger Euler angles. The relationship between input rotation and output quaternion is more complex. As the unit-quaternion is parameterized by $\boldsymbol{q} = \cos\left(\frac{\theta}{2}\right) + \sin\left(\frac{\theta}{2}\right)(a_1\boldsymbol{i} + a_2\boldsymbol{j} + a_3\boldsymbol{k})$, the LSTM is forced to regress the four components that contain sine and cosine. A two-layer LSTM with sigmoid and hyperbolic tangent activation functions (eq. 3.21) might not be powerful enough to approximate the mapping of angle and axis in combination with the cosine- and sine components. Further investigation on this manner is a possibility for future work.



(a) Rotation (b) Translation

Figure 5.5: Experiments on VIPER: Euler vs. quaternion vs. hidden state transfer. In all experiments, the model was trained for 50 epochs on videos of 100 frames. No dropout is applied.

Dataset	**Rotation [deg]**		**Distance [m]**	
	Average	Maximum	Average	Maximum
GTA V 30 fps	0.9488	31.0659	0.1286	0.5440
GTA V 15 fps*	1.8976	62.1318	0.2572	1.0880
VIPER 15 fps	0.5284	15.6798	0.6792	6.6327

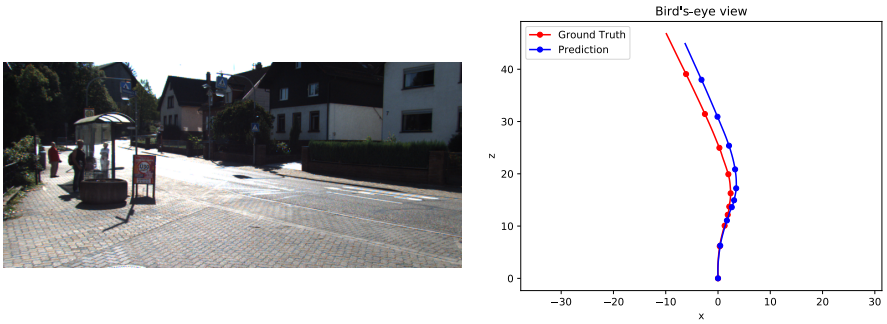
Table 5.6: Inter-frame motion comparison between the VIPER and GTA V datasets. The table shows the relative rotation angle and traveled distance between consecutive frames on the test set. Since the GTA V videos are captured with twice the frame rate as VIPER, the numbers are multiplied by two and shown in the second row (*) for direct comparison with VIPER.

5.5 Evaluation on the GTA V Dataset

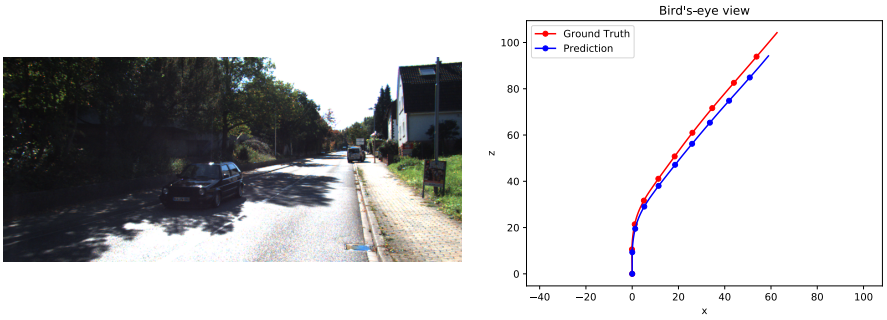
In contrast to the VIPER dataset where most videos are captured from a driving vehicle, in our dataset all motions are captured on foot. Additionally, the frame rate of 30 fps in GTA V is twice as high as in VIPER. The scale of motion between consecutive frames in the videos is compared in table 5.6. In VIPER, the camera is usually pointing in the forward driving direction, whereas in GTA V the camera is turned freely in all directions while walking and exploring the video game world. The average rotation angle in VIPER is lower compared to GTA V, since there are only a few rapid rotation movements in short periods of time. Although the traveled distance per frame in VIPER is longer, the freedom of motion in a car is limited as it can move forward or backward, but not to the side. Some qualitative results for motion estimation trained on VIPER are shown in figure 5.7. The Euler angle representation for pose is used in this setting. Even though the data is synthetic, the images in the figure show that the system is exposed to realistic and challenging scenarios, where multiple object (like cars) interact with the scene in different lighting and weather conditions.

The higher diversity of motion in GTA V makes the dataset more challenging for VO compared to VIPER and even KITTI. Qualitative results of the

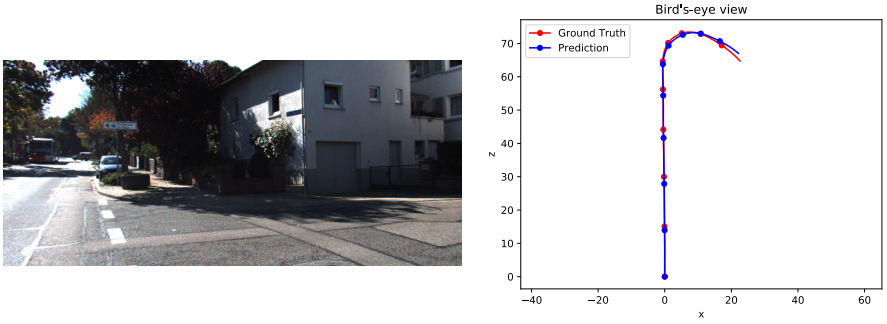
proposed VO applied to GTA V is shown in figures 5.8 and 5.9. In this case, the quaternion pose representation together with the loss from equation 4.3 was used for training. We observe that the estimations for the translation are not very impressive. The system can not handle motions with sharp turns (fig. 5.9d) or motions in wide open areas where structures like houses and mountains are far away.



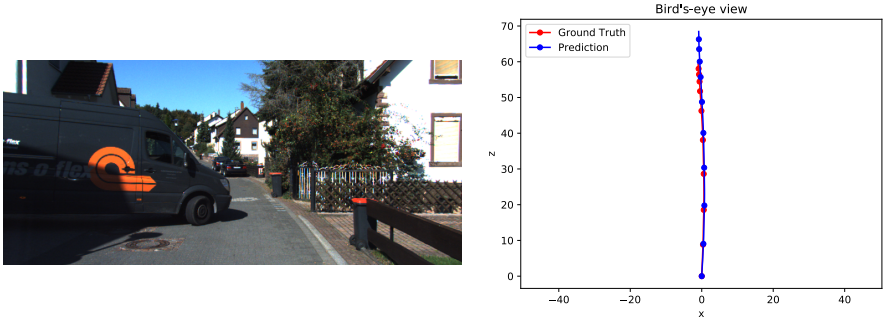
(a) Change of speed at intersection



(b) Dynamic motion from traffic

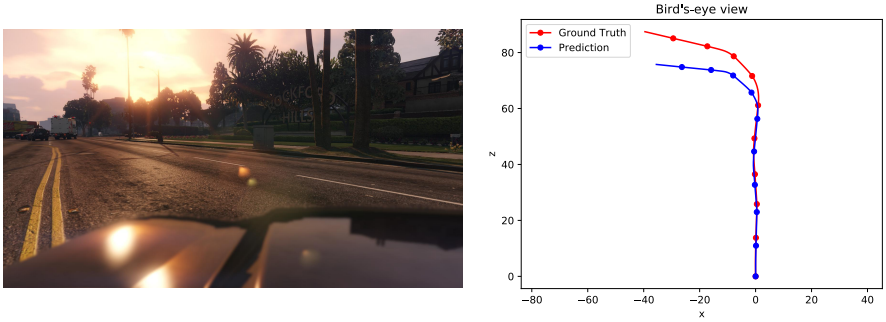


(c) Sharp right-hand bend

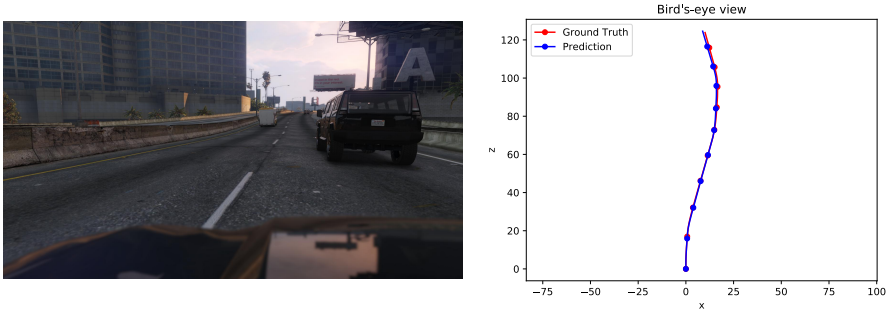


(d) Change of speed and scene motion in close proximity

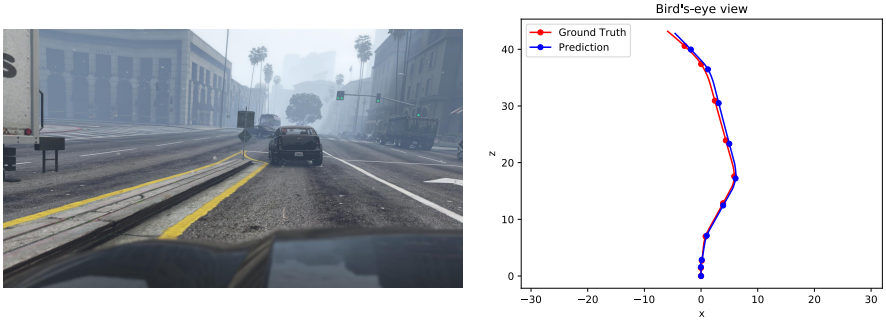
Figure 5.6: Qualitative results for motion estimation on the KITTI test sequence 10. Left column: Images from the subsequences showing a challenging scenario. Right column: Bird's-eye view of the estimated and true camera motion. The height dimension is omitted. Each subsequence is 100 frames long and a marker is shown for every 10th frame.



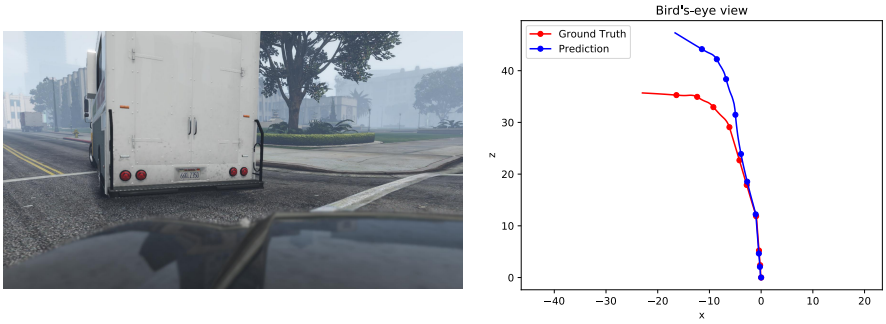
(a) Left turn, reflections from sun



(b) Passing maneuver on highway

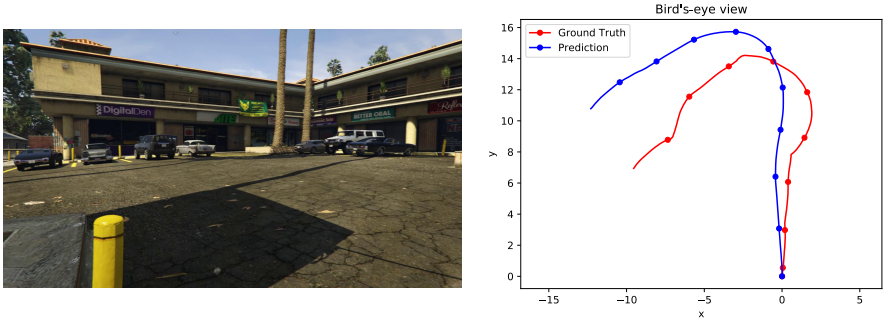


(c) Left turn on crowded intersection, light snow and fog

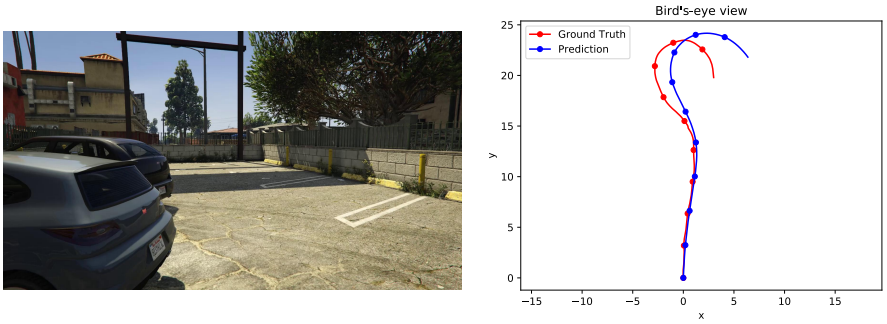


(d) Left turn on intersection following a truck in close proximity

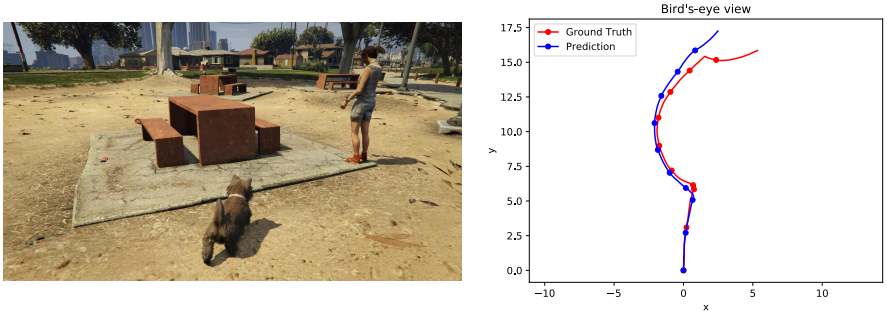
Figure 5.7: Qualitative results for motion estimation on the VIPER test set. Left column: An image from the video showing the scene type and motion scenario. Right column: Bird's-eye view of the estimated and true camera motion. The height dimension is omitted. Each video is 100 frames long and a marker is shown for every 10th frame.



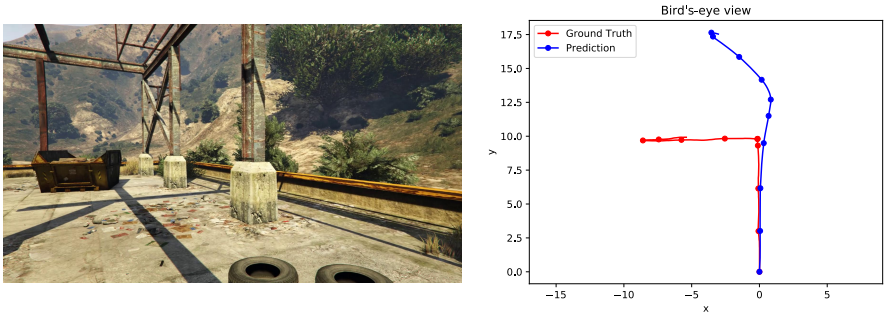
(a) Static scene, medium range



(b) Static scene, close range

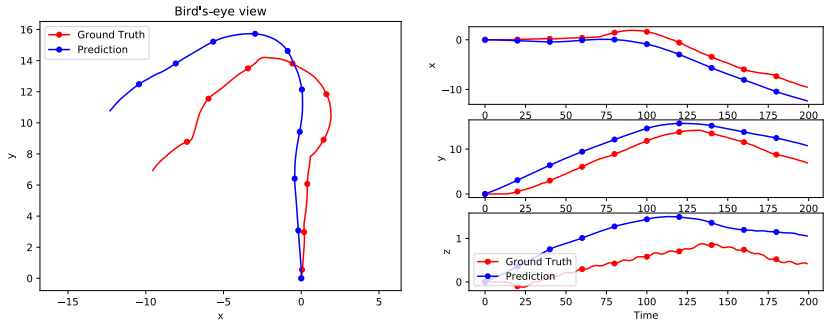


(c) Forward, stop, motion around fixed look-at point

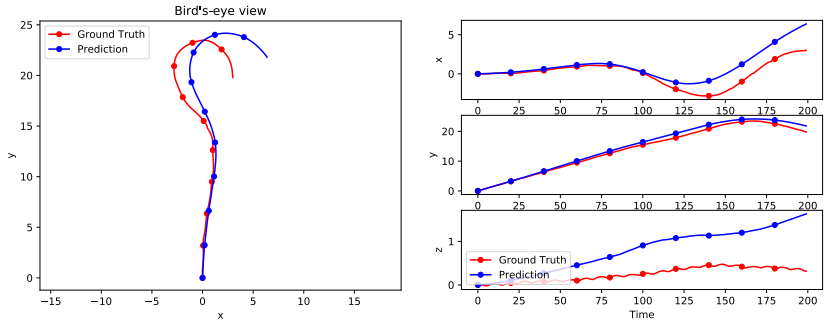


(d) Forward, stop, 90° turn, forward

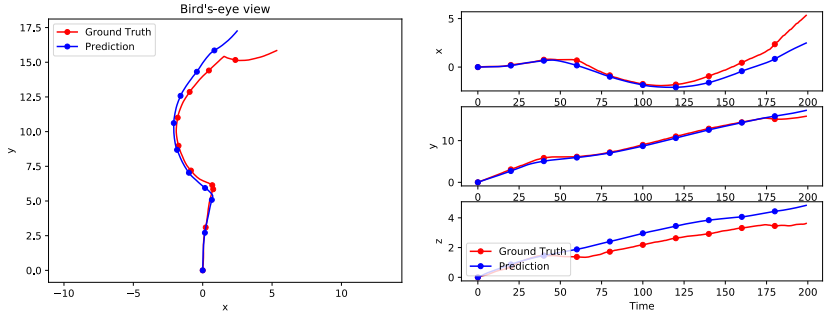
Figure 5.8: Qualitative results for motion estimation on the GTA V test set (part 1). Left column: The first frame of the 200 frames long video. Right column: Visualization of the estimated and true path from the bird's-eye perspective where the height dimension is omitted. The plots show a marker for every 20th frame.



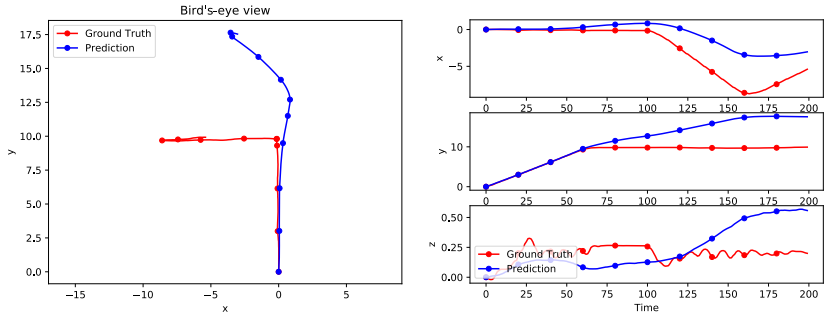
(a) Static scene, medium range



(b) Static scene, close range



(c) Forward, stop, motion around fixed look-at point



(d) Forward, stop, 90° turn, forward

Figure 5.9: Qualitative results for motion estimation on the GTA V test set (part 2). Left column: Visualization of the estimated and true path of the same videos from figure 5.8. Right column: Plot of each coordinate axis of the translation. The horizontal axis is the time (frame count). The plots show a marker for every 20th frame.

Chapter 6

Conclusion and Future Work

This thesis presented a deep learning model for Visual Odometry that makes use of recurrent connections in order to provide a context for each time step where camera location and orientation is estimated. The trained neural network can handle videos of arbitrary length and reconstructs the camera path in real-time. An analysis on real and synthetic data showed that training RNNs for the domain of videos is challenging. We have explored a variety of training schemes on small and large datasets. Sequence overlap, dropout and continuously increasing the sequence length (BPTT) seem to be effective on the small KITTI dataset. Furthermore, the thesis uncovers the problem with global pose and overfitting to the length of training sequences. Although we have solved this problem by converting the camera poses to incremental motions, the situation is rather confusing and counter-intuitive. At this point, we can only speculate on why the RNN does not generalize in a way that the camera pose is accumulated from the beginning of the video and over an arbitrary length. The fact that replacing the LSTM with a affine layer results in better performance indicates that the recurrent state might not be used to the benefit of pose regression.

These issues require further investigation and give plenty of room for future work. Estimating both global- and incremental pose at the same time would be a simple modification worth investigating. In terms of limitations, the loss function requires a manual balance between rotational- and positional error. The thesis did not explicitly explore the impact of the balance weight, but a solution that eliminates or automatically learns such a parameter could certainly have an impact on the convergence speed.

It would also be interesting to explore VO for stereo video input, as it could reduce the drift over time by providing double the amount of measurements for a single viewpoint. This would of course require a suitable dataset and potentially increase the memory footprint.

Furthermore, in the current model, it is unclear to what extent the recurrent network is capable of handling dynamic motion. A dataset with object motion annotations (e.g. in form of masks) would allow to specifically evaluate the impact of the recurrence in this regard. Also, the datasets could be further augmented by creating videos of different frame rate.

Another possibility for future work is to incorporate the new and improved FlowNet2 by Ilg et al. [2016]. The recently released PyTorch code base for the updated FlowNet could be integrated into the existing pipeline developed for this thesis. Although the initialization of the CNN with weights from FlowNet help the convergence speed, it is unclear how large of a benefit the new FlowNet2 weights would bring, since only the first ten layers were used and the decoder part was removed all together.

Lastly, there is room for improvement of the code base developed for this thesis. For example, the currently used library for conversion between different pose representations (Euler, quaternion, etc.) does not have a GPU back-end. It is therefore inefficient to convert poses on-the-fly as they have to be copied to CPU memory and back to the GPU. The few functions for pose conversion could be re-implemented with PyTorch, as it provides the necessary GPU support for its tensor operations.

List of Tables

List of Figures

Bibliography

H. Bay, T. Tuytelaars, and L. Van Gool. SURF: Speeded up robust features. *Computer vision–ECCV 2006*, pages 404–417, 2006.

Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2): 157–166, 1994.

S. Christy and R. Horaud. Euclidean shape and motion from multiple perspective views by affine iterations. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 18(11):1098–1104, 1996.

R. Clark, S. Wang, H. Wen, A. Markham, and N. Trigoni. VINet: Visual-inertial odometry as a sequence-to-sequence learning problem. pages 3995–4001, 2017.

A. Dosovitskiy, P. Fischer, E. Ilg, P. Hausser, C. Hazirbas, V. Golkov, P. van der Smagt, D. Cremers, and T. Brox. FlowNet: Learning optical flow with convolutional networks. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2758–2766, 2015.

M. A. Fischler and R. C. Bolles. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Communications of the ACM*, 24(6):381–395, 1981.

A. Geiger, P. Lenz, C. Stiller, and R. Urtasun. Vision meets robotics: The KITTI dataset. *The International Journal of Robotics Research*, 32(11):1231–1237, 2013.

I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT press, 2016.

K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

E. Ilg, N. Mayer, T. Saikia, M. Keuper, A. Dosovitskiy, and T. Brox. FlowNet 2.0: Evolution of optical flow estimation with deep networks. *arXiv preprint arXiv:1612.01925*, 2016.

A. Karpathy and L. Fei-Fei. Deep visual-semantic alignments for generating image descriptions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3128–3137, 2015.

Y. Ke and R. Sukthankar. PCA-SIFT: A more distinctive representation for local image descriptors. In *Computer Vision and Pattern Recognition, 2004. CVPR 2004. Proceedings of the 2004 IEEE Computer Society Conference on*, volume 2, pages II–II. IEEE, 2004.

A. Kendall and R. Cipolla. Geometric loss functions for camera pose regression with deep learning. *arXiv preprint arXiv:1704.00390*, 2017.

A. Kendall, M. Grimes, and R. Cipolla. PoseNet: A convolutional network for real-time 6-DOF camera relocalization. In *Proceedings of the IEEE international conference on computer vision*, pages 2938–2946, 2015.

D. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

A. Krizhevsky, I. Sutskever, and G. E. Hinton. ImageNet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

H. C. Longuet-Higgins. A computer algorithm for reconstructing a scene from two projections. *Nature*, 293(5828):133–135, 1981.

D. G. Lowe. Object recognition from local scale-invariant features. In *Computer vision, 1999. The proceedings of the seventh IEEE international conference on*, volume 2, pages 1150–1157. Ieee, 1999.

I. Melekhov, J. Kannala, and E. Rahtu. Relative camera pose estimation using convolutional neural networks. *arXiv preprint arXiv:1702.01381*, 2017.

V. Mohanty, S. Agrawal, S. Datta, A. Ghosh, V. D. Sharma, and D. Chakravarty. DeepVO: A deep learning approach for monocular visual odometry. *arXiv preprint arXiv:1611.06069*, 2016.

D. Nistér, O. Naroditsky, and J. Bergen. Visual odometry. In *Computer Vision and Pattern Recognition, 2004. CVPR 2004. Proceedings of the 2004 IEEE Computer Society Conference on*, volume 1, pages I–I. Ieee, 2004.

O. Özyeşil, V. Voroninski, R. Basri, and A. Singer. A survey of structure from motion*. *Acta Numerica*, 26:305–364, 2017.

R. Pascanu, T. Mikolov, and Y. Bengio. On the difficulty of training recurrent neural networks. In *International Conference on Machine Learning*, pages 1310–1318, 2013.

C. Pinard. Pytorch implementation of FlowNet, 2017. URL `https://github.com/ClementPinard/FlowNetPytorch`.

S. R. Richter, Z. Hayder, and V. Koltun. Playing for benchmarks. In *International Conference on Computer Vision (ICCV)*, 2017.

E. Rosten and T. Drummond. Machine learning for high-speed corner detection. *Computer Vision–ECCV 2006*, pages 430–443, 2006.

K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

P. Sturm and B. Triggs. A factorization based algorithm for multi-image projective structure and motion. *Computer Vision ECCV'96*, pages 709–720, 1996.

C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

C. Tomasi and T. Kanade. Shape and motion from image streams under orthography: a factorization method. *International Journal of Computer Vision*, 9 (2):137–154, 1992.

S. Vijayanarasimhan, S. Ricco, C. Schmid, R. Sukthankar, and K. Fragkiadaki. SfM-Net: Learning of structure and motion from video. *arXiv preprint arXiv:1704.07804*, 2017.

S. Wang, R. Clark, H. Wen, and N. Trigoni. DeepVO: Towards end-to-end visual odometry with deep recurrent convolutional neural networks. In *Robotics and Automation (ICRA), 2017 IEEE International Conference on*, pages 2043–2050. IEEE, 2017.

T. Zhou, M. Brown, N. Snavely, and D. G. Lowe. Unsupervised learning of depth and ego-motion from video. *arXiv preprint arXiv:1704.07813*, 2017.

Erklärung

gemäss Art. 28 Abs. 2 RSL 05

Name/Vorname: .....

Matrikelnummer: .....

Studiengang: ……

Bachelor ☐ Master ☐ Dissertation ☐

Titel der Arbeit: .....

.....

.....

LeiterIn der Arbeit: .....

.....

Ich erkläre hiermit, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Alle Stellen, die wörtlich oder sinngemäss aus Quellen entnommen wurden, habe ich als solche gekennzeichnet. Mir ist bekannt, dass andernfalls der Senat gemäss Artikel 36 Absatz 1 Buchstabe o des Gesetztes vom 5. September 1996 über die Universität zum Entzug des auf Grund dieser Arbeit verliehenen Titels berechtigt ist.

.....
Ort/Datum

.....
Unterschrift