DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Deep Learning based Sensor Fusion for 6-DoF Pose Estimation

Michael Sorg

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Deep Learning based Sensor Fusion for 6-DoF Pose Estimation

Deep Learning basierte Sensorfusion für 6-DoF Poseschätzung

Author:	Michael Sorg
Supervisor:	Prof. Gudrun Klinker, Ph.D
Advisors:	Adnane Jadid
	Christian Eichhorn
Submission Date:	15.04.2020

I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 15.04.2020 Michael Sorg

Abstract

In this thesis a deep learning approach for 6-DoF pose estimation is developed. In doing so a neural network model is trained on several sensor inputs (GPS, IMU and odometry) with the aim of predicting a 6D pose (latitude, longitude, altitude, pitch, roll and yaw).
During an architecture search with simulation data it turned out that a recurrent neural network (RNN) containing one gated recurrent unit (GRU) with 256 units performs best. This model archives a mean position accuracy of 3,5 meter while the mean pitch, roll and yaw error is below 1,5 °. By increasing the dataset size from 2 hours to 20 hours the results could be improved to 0,95 meter and 0,3 °.
By training on a real world dataset which contains 1,5 hours of driving time the model archived a mean position accuracy of 3,6 meter and a orientation accuracy of 0,05 °, 0,16 ° and 0,08 ° for pitch, roll and yaw.

Contents

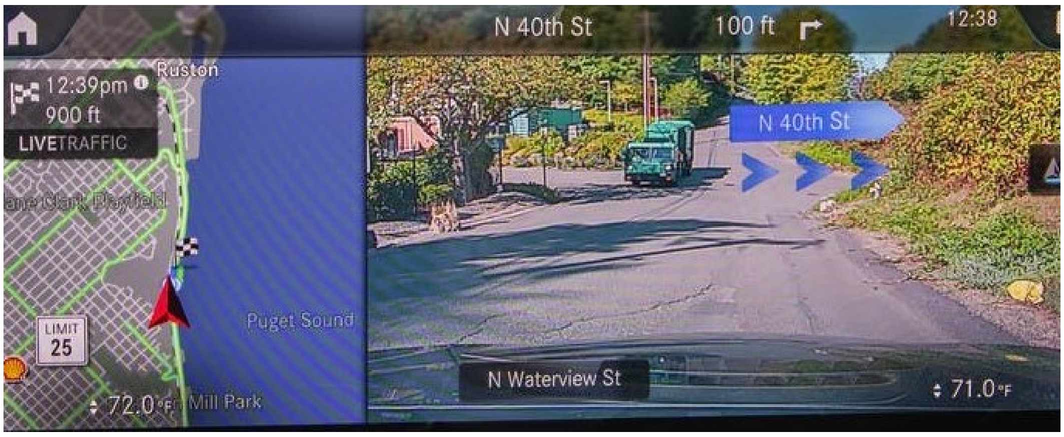
CHAPTER 1

Introduction

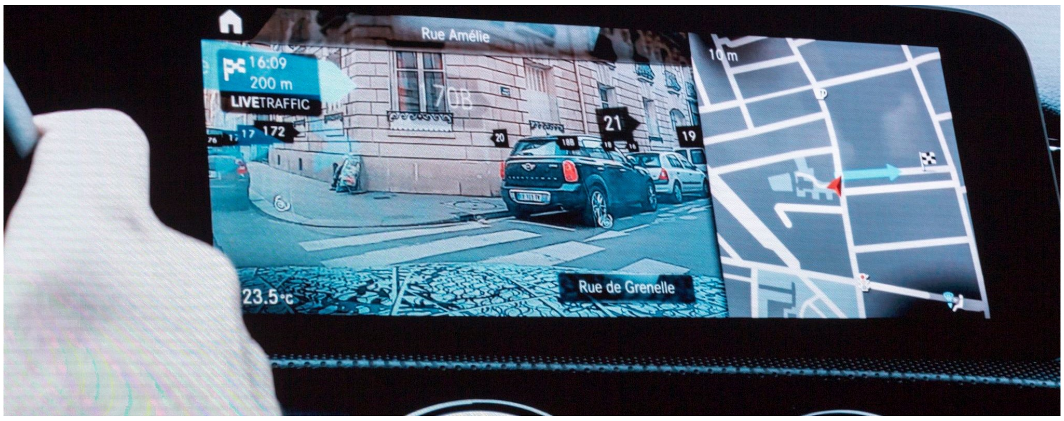
1.1 Motivation

Pose estimation and tracking is one of the most important tasks in the field of augmented reality. Modern navigation systems provide useful information to the user based on the current location [Rao+14a; Rao+14b]. For such an application, as it can be seen in figure 1.1, it is critical to estimate the current pose, meaning the position and orientation, of a vehicle very precisely. An inaccurate pose estimate would result in a mismatch between reality and the augmented livestream which leads to bad user experience. In the case of navigation systems this could even be dangerous if the driving directions are inaccurate or wrong.

Traditional methods are based on mathematical models and human engineered features. However, in a driving scenario these models become very complex and complicated. Furthermore all models are based on assumptions which do not describe the real world exactly. Another way of approaching this problem is the use of machine learning. Here one does not require prior knowledge. Instead the algorithm discovers useful features and pattern by itself without the need of human knowledge. Machine learning algorithms have shown tremendous success on different fields like computer vision, natural language processing and signal processing. Therefore the question arises if machine learning can be used for pose estimation.



(a)



(b)

Figure 1.1: Mercedes MBUX augmented reality navigation. A livestream of the front camera is augmented with additional information such as destination directions, house numbers or street names.

(a) `thedrive.com/tech/23816/daddy-mbux-testing-mercedes-benzs-new-infotainment-system-in-the-2019-a-class`
(b) `blog.mercedes-benz-passion.com/2018/02/neue-a-klasse-erstes-bild-der-augmented-reality-funktion`

There are some challenges for using augmented reality in vehicles:

1. *Sensor noise*. In most cases cheap sensors are used which provide inaccurate measurements with a lot of noise. Besides noise, sensors can contain other types of errors like drift, jitter or complete outages.

2. *Varying sensor update rates*. Different sensor types have different update rates. While a IMU sensor usually provides measurements at 100 Hz the rate is typically at 1-3 Hz for GPS receivers.

3. *Limited computation power*. Computational power is often very limited in vehicles since those platforms are optimized for low energy consumption and efficiency.

4. *Real-time requirements*. Augmented reality applications require real-time performance up to 60 frames per second.

1.2 Task Definition

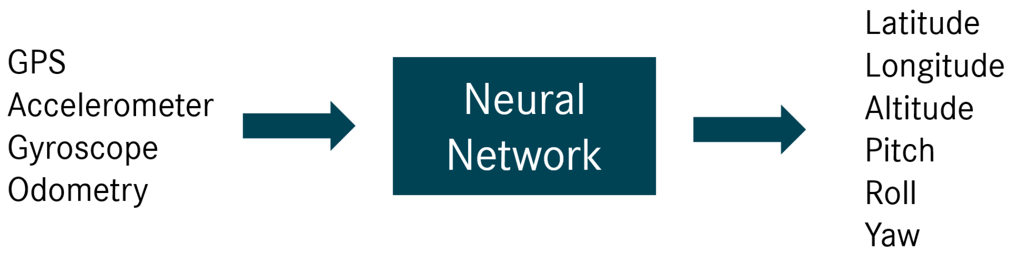


Figure 1.2: Schematic view of the approach to be developed. The input of the algorithm are different sensor measurements. The goal is to train a neural network which predicts a 6D pose estimate.

The aim of this work is it to evaluate if deep learning can be applied to 6-DoF pose estimation using sensor data. This involves the following tasks:

- The dataset contains different sensor types with different update rates which need to be synchronized.
- In order to find a suitable network an architecture search needs to be done which should include different suitable neural network types and layers.
- The resulting trained neural network should be executable in real-time (60 Hz).
- It is to be investigated whether pre-training on simulation data improve the performance

- The performance of the developed neural network should be evaluated quantitatively and qualitatively.

1.3 Overview

Chapter 2 gives an insight into the state of the art for pose estimation and shows related work.
Chapter 3 present the required background knowledge in order to understand this work. It gives an overview of different sensor types and focuses on neural networks.
In Chapter 4 simulation data was used in order to get first insights into the problem. This includes an architecture search in order to find out which architectures and models archive good results as well as the analysis and discussion of the results.
Similar, chapter 5 presents the results when using a real world dataset. Here again an architecture search is described. Furthermore the performance of the developed approach is compared to other algorithms.
Finally, chapter 6 summarizes the results. At the end some future work is presented in chapter 7.

CHAPTER 2

Related Work

Pose estimation is the task of determining the position and orientation of an object. To compute an accurate pose it is essential that different sensor data are combined with each other [Ang+13; KHS17; Lan+10]. *Sensor fusion* describes the process of combining multiple sensor data in order to compute a more accurate results than the sum of the individual sensors [Elm02; Sha17].

Modern vehicles often contain small and low-cost sensors which provide noisy measurements and relatively high errors [Roa08]. In order to use such sensors for automated driving and augmented reality it is critical to filter and fuse those measurements. By combining multiple sensor sources the overall result can be improved. Another benefit of sensor fusion is it that the pose can be predicted continuously even if sensor outages occur. Especially GPS outages are likely in city's or tunnels. By using IMU and odometry data a sensor fusion system is still able to compute a pose estimate in such a scenario.

The most common used approach is the *Kalman filter* [Kal60; HR05] and the *extended Kalman filter* [Sab06; Rib04] which can be used for non-linear processes. There a several publications for 6-DoF (six degrees of freedom) pose estimation and Kalman filters which include only IMU data [TYH13; TO17] as well as IMU and GPS data [YZL12; MWJ12; Qin+19].
However, there are several downsides of the Kalman filter [Roa08; CESN04; VO99]. One main limitation is that the Kalman filter requires an error model (noise model) for each sensor. Thus, it requires accurate prior information about the statistical sensor characteristics. It has been shown that sensor noise in non-linear and correlated over

time which makes it very hard to distinguish sensor data and noise based on a physical model [AKR06; Par04]. Additionally, estimating the error states of the sensors is a major problem [RAHS04]. Finally, in the extended Kalman Filter non-linear motion and sensor models are linearized which leads to inaccuracies.
Other traditional approaches for pose estimation are Bayesian Inference [REG14; BF10], Dempster-Shafer [Wu+02] reasoning or Analytical methods [BL11]. To summarize, these classical approaches often require prior information as well as domain-knowledge and human experience since they often contain handcrafted features and models [Nwe+18].

One promising way to overcome these issues is *deep learning* [GBC16] which had tremendous success in many fields like image and object recognition [He+16], machine translation [Sin+17b; Wu+16] and speech recognition [Dah+11] during the last years [Yan+15; Sch15]. It has been shown that deep learning can be applied to time-series data tasks like classification, forecasting and anomaly detection [Gam17; LKL14; Ism+19]. Deep learning is the approach of training an end-to-end neural network in a supervised manner. Popular deep learning models are *Convolutional Neural Networks (CNN)* [LeC+98; KSH12] and *Recurrent Neural Networks (RNN)* [ZSV14]. These models can learn complex and high-level features from raw sensor data [Zeb+18]. As a result, there is no need for human engineered and hand-crafted features or heuristics. Additionally, RNN's are suited for time series since they take temporal correlations into account [Wan+17]. Often used RNN cells are *long short-term memory (LSTM)* [HS97a] or *gated recurrent unit (GRU)* [Chu+14] which contain an internal memory in order to process sequential data and detect temporal patterns.

Most deep learning approaches for pose estimation so far are based on image and video data as inputs [Pen+19; TSF18; KGC15; Keh+16; TS14; Bel+15]. However, computer vision is computationally expensive since image data can be high dimensional. Especially in vehicles, where low-cost and low-power hardware is used, computational power is very limited which makes it hard to fulfill the real-time requirements. Additionally, visual pose estimation is sensitive to motion blur, occlusions and illumination changes [Ram+16].

Deep learning approaches which use only sensor data are often limited to classification and do not use IMU as well as GPS data [HHC17]. A lot of work has been done in the area of *human activity recognition* [SPG19; Wan+17; Nwe+18; Zhe+14; Lim+19]. In general, this tasks involves classification of IMU data into several categories (walking, running, drinking etc.). These works are often based on Recurrent neural networks with stacked LSTM layers [Hua+18; Wan+19; Zeb+18; Yan+15; HHP; KDD17; MP17b]. The performance of pure RNN networks can be improved by a combination of CNN and

RNN (also called *CRNN*) [Rue+18; HC16; Yao+16; Sin+17a; JY15]. In these networks CNN layers are applied beforehand in order to extract meaningful and useful features which are then processed by the RNN instead of feeding the recurrent layers with the raw sensor data. The idea behind this approach is that the CNN capture spatial relations and the RNN temporal relations [Wan+17]. Some publications indicate that 2D convolution works better than 1D convolution [HYC16; HC16; Yao+16; JY15; Jaf+19].

However, almost all mentioned publications above are used for classification tasks. Pose estimation on the contrary is a *regression* problem [Lat+19]. An interesting approach is *DeepSense* described in [Yao+16] since it can be used for classification and regression tasks. In this work the authors provide a CRNN architecture containing several convolution layers with two recurrent and one output layer on top. Additionally, the input data is transformed to the frequency domain by applying a Fourier transform. The authors claim that patterns are better visible and detectable in the frequency than in the time domain. However, the authors use only IMU data as input.
In [Ö19] an end-to-end CRNN is proposed for 6-DoF pose estimation based on IMU data. Once again, no GPS input is used which has some limitations. Since IMU sensor measurements are local it is not possible to determine the absolute pose of an object. To overcome this, one may feed an initial starting pose into the algorithm and perform a relative prediction with respect to this starting point. However, this approach has serious downsides. Firstly, the result is depends strongly on the accuracy of the initial pose. Secondly, even with small prediction errors at each timestep, those error will accumulate and lead to a inaccurate pose after short prediction intervals. Therefore it is important to use both, IMU and GPS data, in order to determine the absolute pose of a vehicle. To the best of the author's knowledge there exist no end-to-end deep learning approach for 6-DoF pose estimation which uses IMU as well as GPS data at the moment.

Another approach is the combination of deep learning and Kalman filter. In [Ram+16] a RNN predicts a pose from IMU data. Simultaneously, a visual tracker proposes a second pose based on camera images. Afterwards a Kalman filter merges both poses and outputs a final pose.
As mentioned before one main disadvantage of the Kalman filter is that one has to specify a sensor and error model. One way to overcome this issue is to learn those motion and noise models by a neural network which is called *Deep Kalman filter* [Hos18; Cos+17; KSS15].

CHAPTER 3

Background

3.1 Inertial Measurement Unit

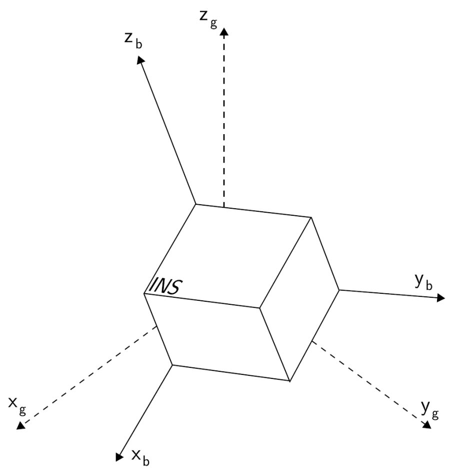
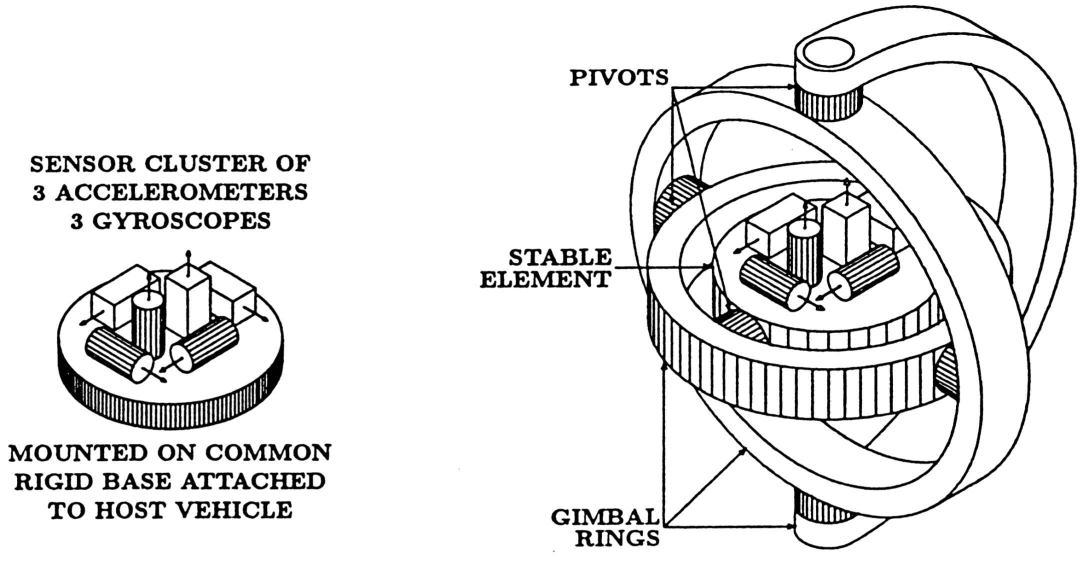


Figure 3.1: A IMU has it's own local (body) frame which is usally not the same as the world (global) frame [Woo07].

An *inertial measurement unit (IMU)* combines accelerometer, gyroscope and sometimes magnetometers into a single unit. Magnetometers are generally not used in the electromagnetic environment of vehicles, as the sensor measurements contain too much noise and are not reliable inside such systems. IMU's measure the (angular) velocity and can be used to determine the position and orientation of an object when no external reference, like GPS, is available (*dead reckoning*) [GWA07]. There are two different types of IMU's (see figure 3.2) [Woo07]:

- **Gimbaled or stable platform**: This systems compensate external rotations such that local frame is always aligned with the global frame. Stable platform system contain torque motors which compensate and cancel out external rotations.
- **Strapdown Systems**: Sensors are mounted on a fixed base. The outputs are therefore measured in the local body frame and not in the world frame. Position and orientation can be computed by integrating the accelerometer and gyroscope measurements which is also called *inertial navigation*. Strapdown Systems are cheaper, smaller and less complex which is why they are more commonly used.



(a) Strapdown (b) Gimbaled

Figure 3.2: Comparison between a strapdown and gimbaled IMU [GWA07].

Since a vehicle has six degrees of freedom (6DoF) its orientation (also called attitude) is described by the rotation around three axes (see figure 3.3). These three angle values are called roll (front-to-back axis), pitch (side-to-side axis) and yaw (vertical axis). These rotation axes a relative to the vehicle. This means that they move within the vehicle relative to the global earth frame.

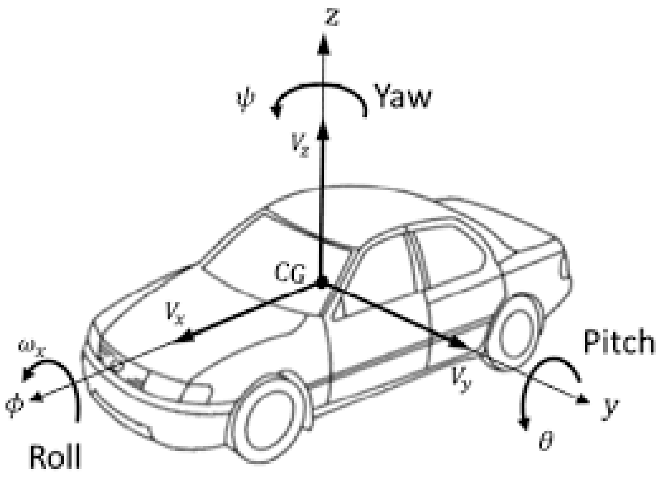


Figure 3.3: Roll, pitch and yaw in a vehicles [Kis+19].

3.2 Global Positioning System

Global Positioning System (GPS) is a system for military, civil and commercial applications [BDW95]. GPS calculates the position of a receiver with an accuracy of 5 to 15 meters if the receiver is visible by four or more satellites [Sha17]. GPS is cheap, fast, reliable and provides an absolute position in the world frame. However, position accuracy can be drastically reduced by obstacles like buildings, mountains, trees or bad weather conditions.
The position accuracy can be improved to 1 - 3 cm by using *differential GPS (DGPS)* [PE96]. DGPS uses a additional base station in order to compute differential corrections. The major drawback of DGPS is that it only works in a narrow area around a reference station and is more expensive.

A location on the earth is represented in a geographic coordinate system. To represent a position latitude, longitude and altitude (or elevation) are often used (see figure 3.4). Latitude describes the angle between a point and the equatorial plane. Latitude values range from -90° (South) to +90° (North). Longitude measure the angle to the east or west and range from -180° to +180°.

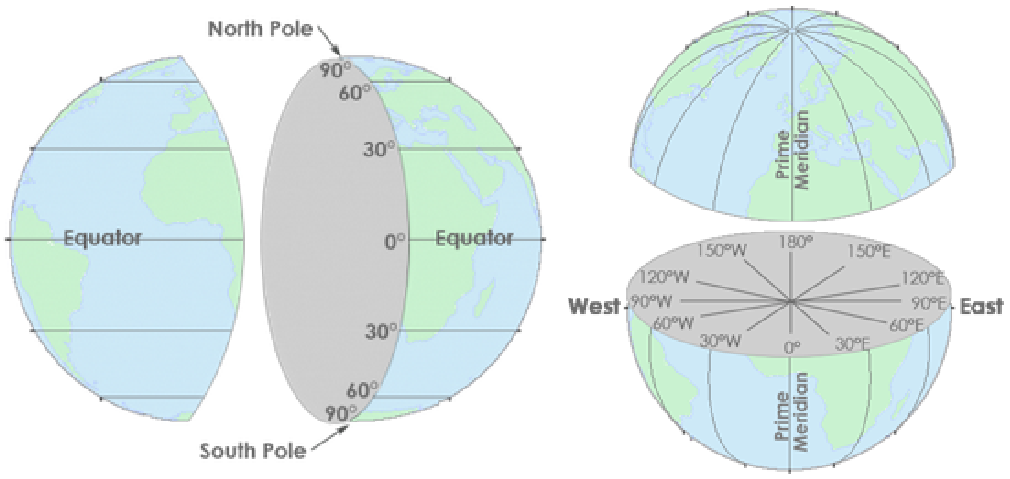


Figure 3.4: Latitude and longitude system.

`https://gisgeography.com/latitude-longitude-coordinates` (13.02.2020)

3.3 Artificial Neural Network

Artificial neural networks are inspired by the biological nervous systems and can be viewed as a simplified mathematical model of the human brain. Neural networks consist of several connected units which exchange signal among each other. By stacking multiple of those units neural networks are able to learn and process non-linear relationships.

3.3.1 Perceptron

The fundamental unit of a neural network is the so called *Perceptron* [Ros61] developed by Rosenblatt in 1958. The Perceptron is a binary classifier which takes an vector x as input an maps it to an output $f(x)$, where $\mathbf{w}$ are the weights and b the bias.

$$f(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{w} \cdot \mathbf{x} + b > 0 \\ 0 & \text{otherwise} \end{cases} \tag{3.1}$$

After multiplying each input value with the respective weight (dot product) a bias term is added in order to shift the decision boundary of the classifier. The activation function determines the output of the classifier and is the Heaviside step function for the Perceptron (see figure 3.5).

A major problem of the Perceptron is that it can only solve linear separable problems. Therefore, even the logical XOR problem is not solvable by a Perceptron [MP17a]. This

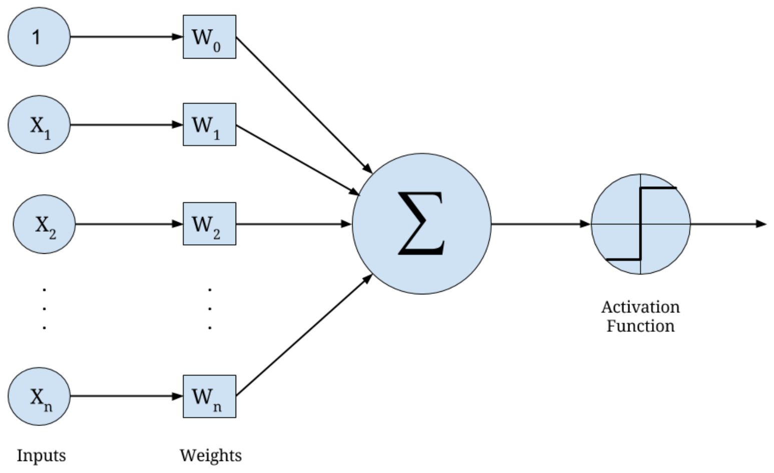


Figure 3.5: Perceptron Model [Sha17]. The Perceptron is a binary classifier with weights w which maps a set of input values to a binary output.

issue can be solved by stacking several Perceptrons.

3.3.2 Feedforward Neural Network

A *feedforward neural network* is contains several Perceptrons stacked into layers (see figure 3.6). It consists of an input layer, a variable number of hidden layers and an output layer. Thus it is also called *multilayer perceptron (MLP)*. A MLP is a universal function approximator and is able to learn complex functions including the XOR problem [HSW89].

A MLP has a variable number of outputs, depending on the number of units in the output layer. Additionally, this model can be used for classification and regression by choosing an appropriate activation and loss function.

In order to learn non-linear relationships it is necessary to use *non-linear activation functions*. By stacking those non-linearities a neural network can learn abstract and high-level concepts. The following are popular and often used activation functions:

- Rectified Linear Unit (ReLU):

$$\sigma(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & x \leq 0 \end{cases} \tag{3.2}$$

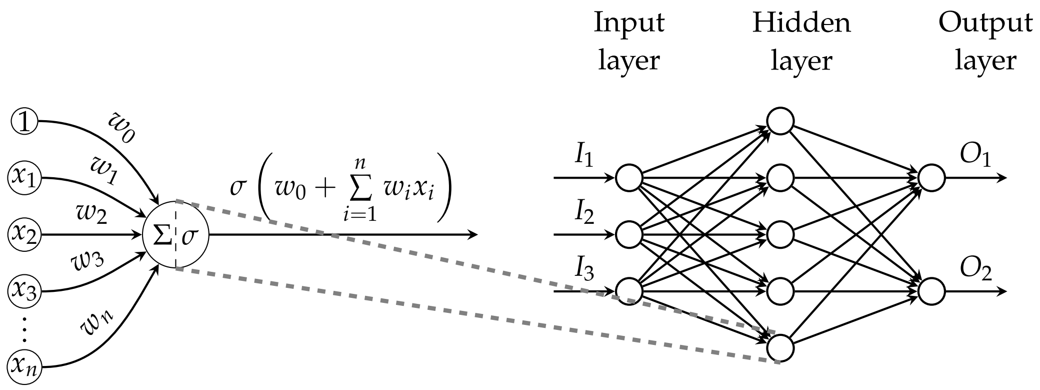


Figure 3.6: Model of a feed forward neural network (multilayer perceptron) with a single hidden layer where σ is the activation function. A MLP can consists of multiple hidden layers with different activation functions in every layer. In most cases MLP's are fully connected which means that the inputs of each unit are the outputs of all units in the previous layer.

- Hyperbolic Tangent (tanh):

$$\sigma(x) = \frac{2}{1 + e^{-2x}} - 1 \tag{3.3}$$

- Sigmoid:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{3.4}$$

3.3.3 Training and Optimization

In neural networks learning is achieved by modifying the weights. The goal of supervised learning is to minimize the deviation (error) between the target output value $\hat{y}$ and the actual output value y. Therefore a *cost function* or *loss function* is required which measures the error. Possible loss functions for regression problems are:

- mean squared error (MSE):

$$MSE = \frac{\sum_{i=1}^{N} (\hat{y}_i - y_i)^2}{N} \tag{3.5}$$

- root-mean-square error (RMSE):

$$RMSE = \sqrt{\frac{\sum_{t=1}^{N} (\hat{y}_t - y_t)^2}{N}} \tag{3.6}$$

The goal of the training process is to adjust the weights in such a way that the loss function is minimized. *Gradient descent* is a possible optimization method for finding such a minima. Here one iteratively follows the negative gradient of the loss function $\nabla L(w)$ by a small step size λ (learning rate). Hence we change the weights w at each timestep t in such a way that the loss function decreases fastest:

$$w_{t+1} = w_t - \lambda \nabla L(w_t) \tag{3.7}$$

Some drawbacks of gradients descent optimization are [GBC16]:

- There is no convergence guarantee. Training can be slow, especially if the loss function contains plateaus, saddle points or flat regions.
- It may converge to a local minima.
- Oscillation and zig-zagging may occur which may slow down training or even lead to divergence.

However, gradient descent is the most used optimization method for neural networks. Some of the mentioned issues above can be reduced if a more advanced optimization method is used which modifies and extends gradient descent and can speed up training [Rud16]. Popular choices are *Adam* [KB14] or *RMSprop* [TH12].
Within gradient descent the loss is computed with respect to all training examples before a weight update happens. When training on large datasets this can lead very quickly to memory issues and long training times. In order to speed up the learning process *stochastic gradient descent* (SGD) is used. Here the weights get updated after a batch of training examples (usually 64 to 512). Therefore stochastic gradient descent is only a approximation of the "true" gradient and not as accurate. However, in practice it is only feasible to use stochastic gradient descent since memory and time is limited.

Calculating the gradients with respect to each weight of the loss function is not trivial. Since a neural network can have millions of weights (parameters) and a complex loss function it is infeasible to calculate the gradients directly or analytically. For neural networks *Backpropagation* [RHW86] is a efficient way for for calculating the gradients of a neural network and updating weights.

In order to detect and prevent *overfitting* the available data is usually split into training, validation and test data [GB10; Zha+16]. Overfitting happens if the network memorizes training examples and does not generalize to new unseen data. One way to prevent overfitting is to use the *early stopping* method [Pre98]. In this case training is stopped, if the validation error increases (see figure 3.7).

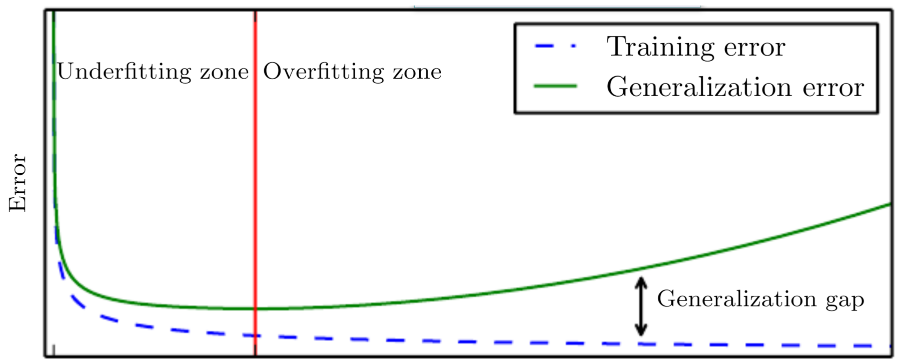


Figure 3.7: Typical plot of training and validation loss during training a neural network [GBC16]. The blue training loss decreases constantly. At some point the validation loss start increasing which is a indicator for overfitting.

Other techniques to prevent overfitting are:

- *Dropout* [Sri+14]: A certain percentage of the units are excluded which helps to reduce the generalization gap [Sri13].
- *DropConnect* [Wan+13]: Similar to Dropout. Here a certain percentage of weights are set to zero.
- *L1 / L2 Regularization* [Ng04]: Add a regularization or penalty term to the loss function that punishes large weights.

3.3.4 Convolutional Neural Networks

A *convolutional neural network (CNN)* [LeC+98] is able to learn and extract features from input data and have been successfully applied to a wide range of applications like image classification [He+16], human activity recognition [Yao+16] or natural language processing [Geh+17]. The core of a CNN is the so called (discrete) convolutional operation. This works by looping a filter (kernel) over the input data which can be seen in figure 3.8. Usally a convolutional layer contains multiple filters which weights are learned automatically by gradient descent. By stacking several convolutional layers followed by a non-linear activation function CNN's are able to learn abstract and high-level features [Yos+15; Qin+18].

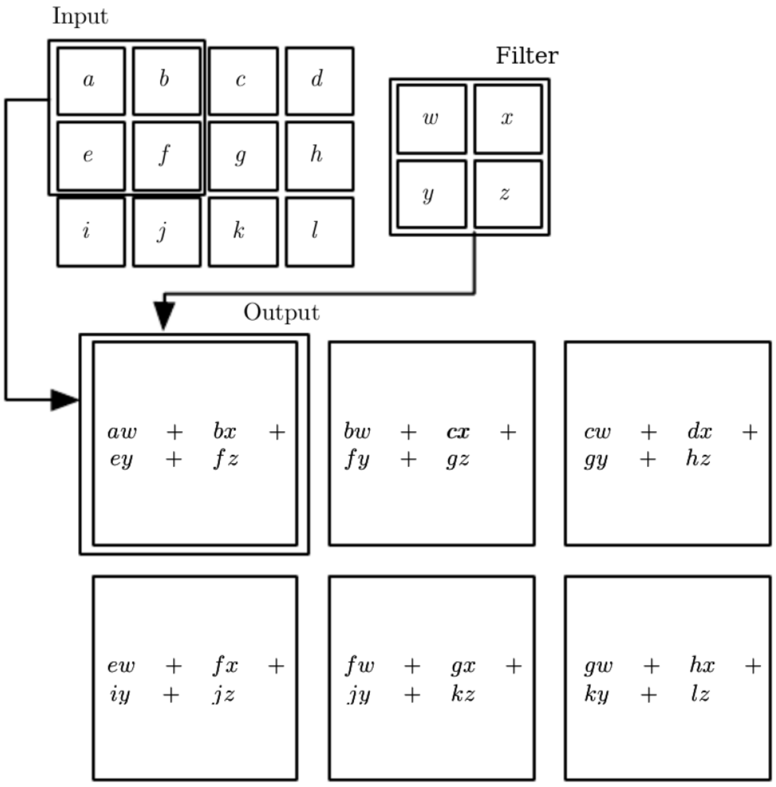


Figure 3.8: Example of a 2D convolution [GBC16]. Here a 2 x 2 filter is applied to a 3 x 4 input. The result of the operation is a 2 x 3 output.

3.3.5 Recurrent Neural Network

A *recurrent neural network (RNN)* has a feedback connection as it is shown in figure 3.9. Combined with their internal state (memory) RNN's are able to process sequential and time-series data [GSC99].

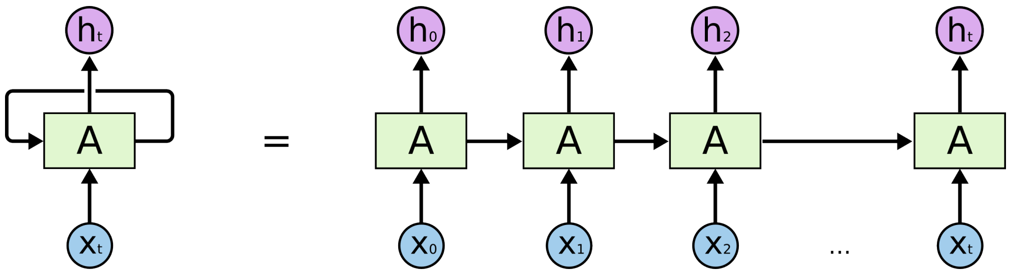


Figure 3.9: A recurrent neural network has a feedback connection and can be also be viewed as unrolled graph [Ola15]. Thus, the input at each time step is the previous internal state and the current input.

However, when the input sequence is long standard recurrent neural networks suffer from vanishing or exploding gradients. This can be avoided by using *long short-term memory (LSTM)* cells [HS97a]. Instead of using just a single activation function per cell, LSTM's are more advanced units (see figure 3.10). Each LSTM block contains 3 gates (input, output and forget gate) which modify and regulate the flow of information. As a result the gradients are more stable and the network can store and process information for long time intervals [HS97b].

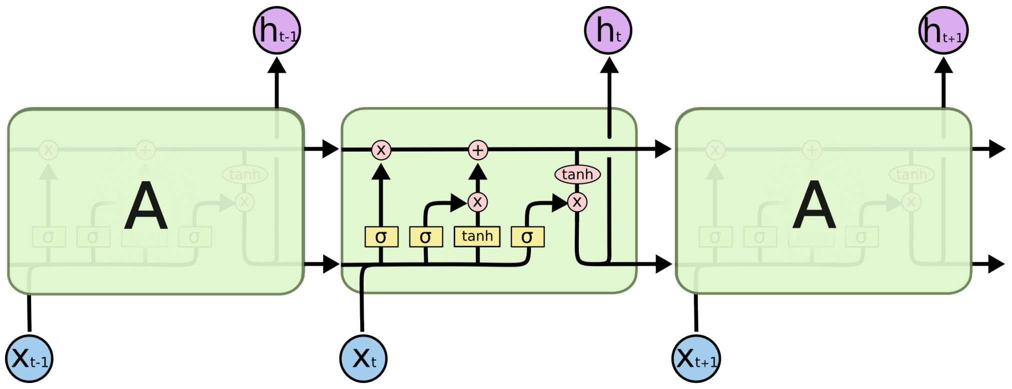
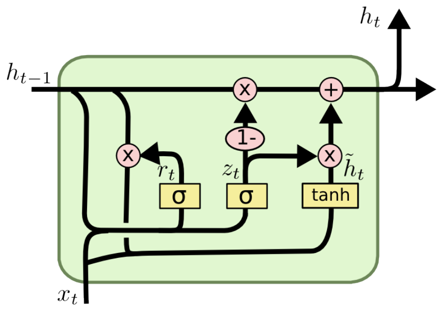


Figure 3.10: Model of a LSTM network [Ola15]. The yellow σ node represents the sigmoid function.

LSTM's forward pass can be described by the following equations where f, i and g are the forget, input and output gate. σ is the Sigmoid activation function, b the respective bias for each gate and c the internal cell state. The Hadamard product $*$ denotes the element-wise product. W and U are the trainable weight matrices.

$$f_t = \sigma(W_f x_t + U_f h_{t-1} + b_f) \tag{3.8}$$
$$i_t = \sigma(W_i x_t + U_i h_{t-1} + b_i) \tag{3.9}$$
$$o_t = \sigma(W_o x_t + U_o h_{t-1} + b_o) \tag{3.10}$$
$$c_t = f_t * c_{t-1} + i_t \circ tanh(W_c x_t + U_c h_{t-1} + b_c) \tag{3.11}$$
$$h_t = o_t * tanh(c_t) \tag{3.12}$$

An popular alternative to a LSTM cell is the *gated recurrent unit (GRU)* [Cho+14]. GRU's combine the forget and input gates into a single gate as well as some other minor changes (see figure 3.11). This makes the model simpler and requires less parameters which results in faster training time. However, it has been shown that LSTM's outperform GRU's in most cases [Chu+14; Bri+17].



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

Figure 3.11: Model of a GRU cell with corresponding equations [Ola15]. The operator $*$ denotes element-wise multiplication.

CHAPTER 4

Training on Simulation Data

In order to find out which models, architectures and techniques work for the task of pose estimation simulation data was used. Training on simulation data has several benefits:

- In reality, real world data contain typically both, more diverse and higher noise. With simulations, however, the sensor noise can be varied and freely selected. Thereby expensive hardware can be simulated in order to gain more insights and explore the limitations of the developed algorithm.
- Real sensors have different resolutions or update frequencies. While IMU sensors provide measurements at around 100 Hz GPS sensor on the other hand are often limited to 1 - 3 Hz. A simulation framework on the other hand is able to provide GPS measurements with 100 Hz.
- The measurements of different sensor are perfectly synchronized. Thus, at each timestep the measurements from all sensor as well as the ground truth data is available. In reality sensors measurements arrive unsynchronized. Additionally, sensor failure, outage and jitter occur in real world. By using simulation data one can generate high quality data such that additional data pre-processing steps are not needed.
- Data gathering is time consuming and cost intensive in real world. It also requires special hardware for monitoring the ground truth data which must be set up and configured properly.

To summarize, simulation data is used in order to reduce the complexity of the task. This allows to gain first insights and understandings about the presented problem.

4.1 Simulation Framework

The public available framework *GNSS-INS-SIM*[1] can be used for simulating a 6D pose with the corresponding sensor measurements. It provides IMU, odometer, magnetometer and GPS measurements as well as the associated ground truth values for each sensor. The ground truth reference trajectory can be freely defined with a so called motion profile (see table 4.1).

1	**ini lat**	**ini lon**	**ini alt**	**ini vx**	**ini vy**	**ini vz**	**ini yaw**	**ini pitch**	**ini roll**
2	31.9965	120.004	0	10	0	0	315	0	0
3	**command**	**yaw**	**pitch**	**roll**	**vx**	**vy**	**vz**	**duration**	**gps**
4	1	0	0	0	10	0	0	10	1
5	5	0	45	0	10	0	0	250	1

Table 4.1: Example motion file to generate the reference trajectory.

A motion profile starts with the initial position, velocity and orientation in line 2. From line 4 onwards the actual motion definition begins. There are several different command types. Type 1 defines the absolute velocity and orientation change rate. In this case we define that the vehicle accelerates with 10 $\frac{m}{s}$ for the next 10 seconds while the orientation does not change (line 4). Command type 5 sets the orientation change rate and the absolute velocity. In this example, during the next 250 seconds, we increase the pitch angle by 45 degree and set the velocity along the x axis to constant 10 $\frac{m}{s}$ (line 5).

After the trajectory has been defined one can set the sensor update frequencies and sensor accuracy. Unless otherwise stated, all sensor measurements (including GPS) are simulated at 100 Hz with medium accuracy. The error model (noise) of each sensor can be set freely. If not otherwise stated medium accuracy is used (see table 4.2).

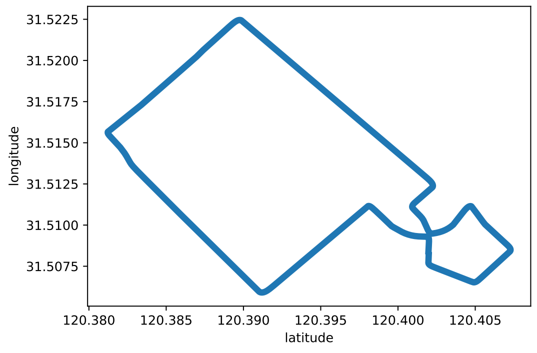
[1]`https://github.com/Aceinna/gnss-ins-sim`

error type	**low accuracy**	**medium accuracy**	**high accuracy**
gyro bias (deg/h)	0	0	0
gyro angle random walk (deg/rt-h)	0,75	0,25	2.0e-3
gyro bias instability (deg/hr)	10	3,5	0,1
accel. bias (m/s^2)	0	0	0
accel. velocity random walk (m/s)	0,05	0,03	2.5e-5
accel. bias instability (m/s^2)	2.0e-4	5.0e-5	3.6e-6
magnetometer std. noise (uT)	0,1	0,01	0.001
GPS position RMS error (m)	5	5	5
GPS vertical RMS error (m/s)	0,05	0,05	0,05

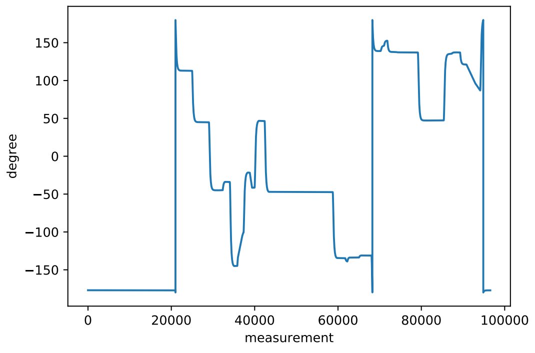
Table 4.2: Predefined sensor error model from the simulation framework.

4.2 Dataset Analysis

For training a motion profile was defined which resulted in the trajectory shown in figure 4.1. The dataset contains a round trip route with a driving time of 10 minutes per round. The data is rather simple and minimalistic, because it does not contain all possible driving scenarios and is quite short. Additionally, ground truth values for pitch, roll and altitude values do not change and are constant at zero. As stated previously, the aim is it to simply the problem and start at a lower complexity in order to find a good performing model.



(a) GPS coordinates

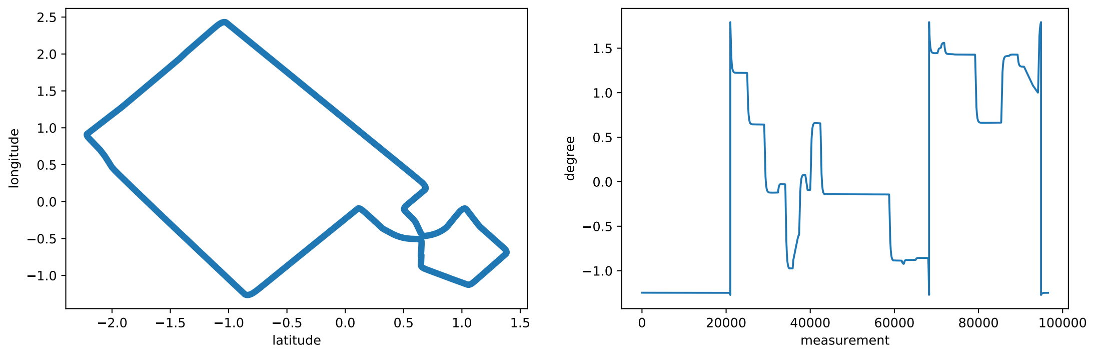


(b) yaw values

Figure 4.1: Dataset overview. The left plot shows the ground truth GPS coordinates. On the right the yaw angles are shown. Altitude, pitch and roll are constant at zero.

The sample rate of the sensor measurements was set to 100 Hz for IMU as well as the GPS sensor. Therefore no synchronization of the sensor measurements is needed.

4.3 Data Preprocessing



(a) GPS coordinates (b) yaw values

Figure 4.2: Dataset values after standard scaling has been applied.

The first pre-processing step is it to scale the different features. This is important, because the range of values differs strongly between the features since they have different units and scales. These differences may increase the difficulty of the problem. Especially for neural networks very high values can lead to large weights and vice versa. However, very small or large weights can make training and optimization unstable, lead to bad performance and generalization or can even cause divergence.
Very often used is the so called standard scaler. This scaling method will transform the data in such a way that the distribution will have a mean value of 0 and standard deviation 1. To archive this we subtract the mean value μ of the dataset and then divide by the standard deviation σ.

$$z = \frac{x-\mu}{\sigma} \tag{4.1}$$

$$\mu = \frac{1}{N}\sum_{i=1}^{N} x_i \tag{4.2}$$

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i-\mu)^2} \tag{4.3}$$

In this case we have a multivariate dataset with 9 features (3 for each sensor). Therefore we apply the scaling feature-wise (each column individually). The effect of this transformation can be seen in figure 4.2. The data has still the same shape, but the value range is changed compared to figure 4.1.

Predicting an accurate pose estimate from on single sensor measurement is clearly not manageable. Since this is a time series regression problem the network gets multiple sensor readings from the past as input. In order to generate the input data a sliding window approach is used (see figure 4.3). At each timestep the inputs are the last T sensor measurement from all sensors, where T is called the *window size* or *sequence length*. Afterwards the window is moved by 1 timestep such that we generate overlapping windows and generate as much data as possible.

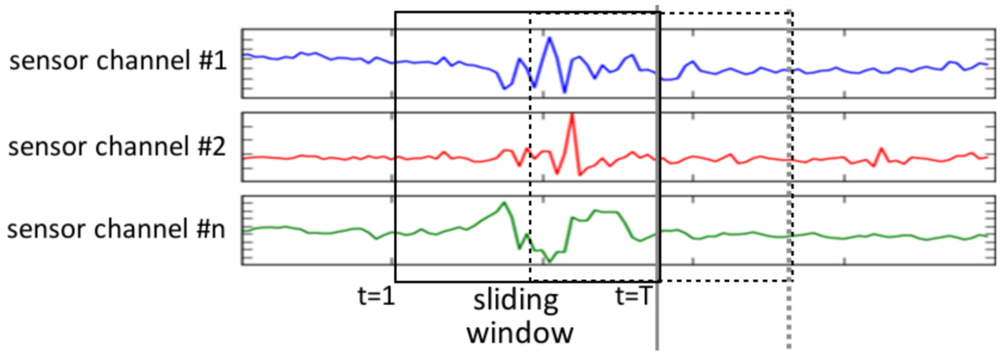


Figure 4.3: Sliding window approach for data preprocessing [OR16]. The length of the window is also called window size or sequence length.

The last preprocessing step is it to split the data into a training and test set. In this case 80% were used for training and 20% for testing.

4.4 Architecture search

In order to find a good performing model an architecture search was done. Here model architectures and parameters are systematically tested by a grid search. It is first investigated how good pure recurrent neural networks work before testing the combination of CNN and RNN.

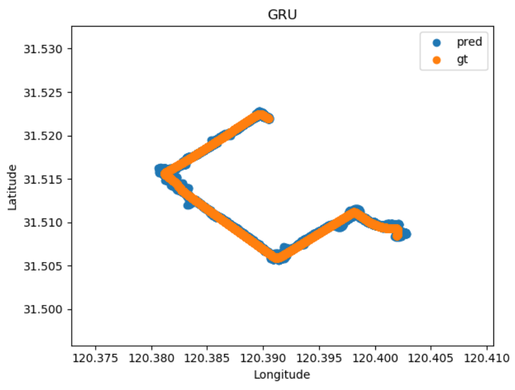
4.4.1 RNN

The most obvious architecture for time series modeling is a recurrent neural network. RNN's are specifically designed for processing time series since they have an internal memory.

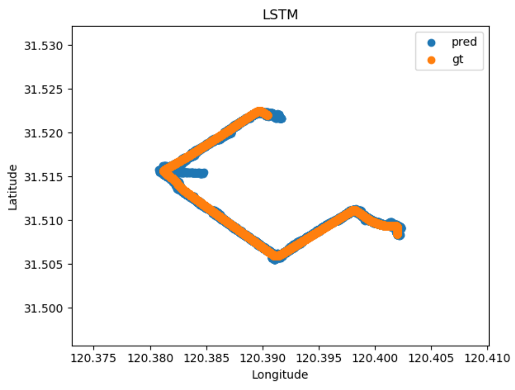
Comparison between GRU and LSTM

A first question is which RNN cell to use. The most common cell types are *long short-term memory (LSTM)* [HS97a] and *gated recurrent unit (GRU)* [Chu+14]. Typically LSTM's perform slightly better since they have a more complex internal structure with more parameters. However, GRU are very popular since they are simpler. Due to the fact the GRU has less parameters they allow deeper architectures and train faster which is why they are often preferred over LSTM.

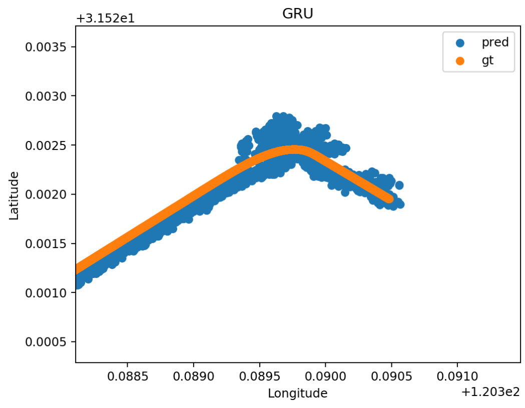
In multiple experiments with different parameters and number of layers it turned out that the quantitative difference between LSTM and GRU cells is negligible. Training and validation loss as well as other metrics such as mean position error are almost identical. However, there is a qualitative difference if one looks at the predictions shown in figure 4.4. It turns out that LSTM's sometimes has larger deviations and bigger jumps. The GRU predictions on the other hand are more stable, especially while making a turn. The mean average position error is almost identical, but qualitatively the GRU outputs are preferred. The jumps like in the LSTM outputs would lead to bad user experience. This is why the GRU predictions are preferred and in subsequent sections always GRU cells are used.



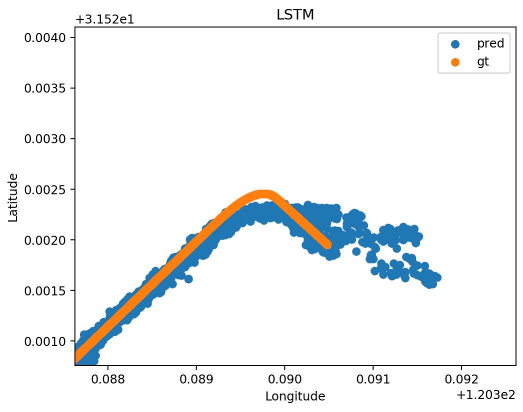
(a) GRU



(b) LSTM



(c) GRU



(d) LSTM

Figure 4.4: Comparison between GRU and LSTM predictions. The top shows the predictions and ground truth values. The second row shows an enlarged section of the upper row. On average the quantitative deviation is almost identical between GRU and LSTM. However, qualitatively the GRU outputs are better and preferred. The left hand side show GRU predictions which are more stable. On the right are LSTM outputs which sometimes contain huge deviations from the ground truth (gt).

Architecture search

In the context of this architecture search 2 hours of data was used. This amount of data is generated when you drive the above mentioned tour (section 4.2) 10 times. GPS and IMU data were simulated with an update rate of 100 Hz. Magnetometer sensor measurement were not used as inputs as they wouldn't be available in a real world car scenario. Each sensor (accelerometer, gyroscope and GPS sensor) produces 3 values which results in 9 input values.

The results for the architecture search are shown in table 4.3. The models listed are just a selection of the architectures actually tested in order to keep it clearly. The architecture *GRU(64), GRU(128)* for example means that the model has two hidden GRU layers with 64 and 128 units. The last layer is always a dense layer with 6 units since we predict 6 values. Deeper models with more than 3 layers were not possible since this leads to training problems such as vanishing and exploding gradients which is typical for training recurrent neural networks. The same holds for using a cell with more than 512 units. The following conclusions can be drawn from this architecture search:

- Increasing the window size (sequence length) seems to improve the performance. Going beyond a window size of 500 however does not seem to improve the result much more. Using a window size of more than 1000 (which equals to 10 seconds of data) was not possible due to training errors such as vanishing and exploding gradients.
- Pitch and roll estimates are constantly very good due to the fact that their ground truth values are always zero in this dataset. This shows that the network is in principle able to learn a target value very precisely.
- Deeper networks with 3 or more layers do not improve the performance. Networks with 1 or 2 layer outperform deeper models in most cases.
- The best models are GRU(64) and GRU(256). These models contain only one single hidden layer with 64 and 256 units. In the best case the model GRU(256) archives an average position error of 3,5 m and a mean yaw angle error of 1,5 $^\circ$.
- Regularization and Dropout did not improve the metrics. For most of the models training and testing loss was very close such that there was no generalization gap. Unsurprisingly, regularization and dropout are not needed in this case.

Architecture	**seq-len**	**position**	**yaw**	**pitch**	**roll**
GRU(64)	10	6,4	7,3	0,0016	0,0013
GRU(128)	10	6,9	4,7	0,0026	0,0028
GRU(256)	10	6,5	5,0	0,0037	0,0023
GRU(512)	10	6,8	5,0	0,002	0,0019
GRU(64),GRU(64)	10	7,2	7,0	0,0034	0,004
GRU(128),GRU(128)	10	7,6	4,1	0,002	0,002
GRU(256),GRU(256)	10	7,1	3,7	0,0031	0,0018
GRU(512),GRU(128)	10	7,6	3,8	0,0023	0,002
GRU(64),GRU(64),GRU(64)	10	7,9	5,6	0,0019	0,003
GRU(128),GRU(128),GRU(128)	10	8,7	4,1	0,0027	0,003
GRU(256),GRU(128),GRU(64)	10	6,8	3,7	0,0023	0,001
GRU(64)	100	4,8	3,5	0,0022	0,0017
GRU(128)	100	6,4	4,6	0,0028	0,0021
GRU(256)	100	4,8	2,2	0,0023	0,0028
GRU(512)	100	5,4	1,5	0,0021	0,0020
GRU(64),GRU(64)	100	6,3	2,9	0,0042	0,0017
GRU(128),GRU(128)	100	6,3	2,1	0,0032	0,0047
GRU(256),GRU(256)	100	8,7	1,6	0,0022	0,0029
GRU(512),GRU(128)	100	6,2	0,9	0,0030	0,0024
GRU(64),GRU(64),GRU(64)	100	6,0	1,6	0,0014	0,0035
GRU(128),GRU(128),GRU(128)	100	7,5	1,3	0,0024	0,0016
GRU(256),GRU(128),GRU(64)	100	6,7	1,1	0,0022	0,0028
GRU(64)	500	3,9	3,8	0,0023	0,0028
GRU(128)	500	4,0	2,6	0,0026	0,0022
GRU(256)	500	3,4	3,5	0,0020	0,0023
GRU(512)	500	5,4	1,9	0,0020	0,0084
GRU(64),GRU(64)	500	3,7	2,3	0,0039	0,0031
GRU(128),GRU(128)	500	6,3	1,8	0,0030	0,0020
GRU(256),GRU(256)	500	6,4	2,9	0,0055	0,0069
GRU(512),GRU(128)	500	6,6	1,6	0,0022	0,0032
GRU(64),GRU(64),GRU(64)	500	4,6	2,8	0,0030	0,0029
GRU(128),GRU(128),GRU(128)	500	4,7	1,5	0,0034	0,0031
GRU(256),GRU(128),GRU(64)	500	7,1	1,3	0,0036	0,0027
GRU(64)	1000	4,3	5,1	0,0025	0,0021
GRU(128)	1000	4,4	2,4	0,0030	0,0013

GRU(256)	**1000**	**3,5**	**1,5**	**0,0061**	**0,0030**
GRU(512)	1000	6,4	3,7	0,0042	0,0092
GRU(64),GRU(64)	1000	3,8	2,1	0,0025	0,0021
GRU(128),GRU(128)	1000	4,2	1,9	0,0030	0,0041
GRU(256),GRU(256)	1000	5,8	1,4	0,0043	0,0042
GRU(512),GRU(128)	1000	4,2	1,4	0,0030	0,0026
GRU(64),GRU(64),GRU(64)	1000	5,9	3,3	0,0040	0,0035
GRU(128),GRU(128),GRU(128)	1000	5,0	2,3	0,0025	0,0044
GRU(256),GRU(128),GRU(64)	1000	5,6	2,5	0,0039	0,0058

Table 4.3: Result of the architecture search for recurrent neural networks. Each model was trained for 5 epochs. The reported values in the columns position, pitch, roll and yaw are the mean absolute error of the test data. Position error is in meters and angle error in degrees. Column *seq-len* is the sequence length or window size and determines how many sensor measurements from the past the network gets as input. Data is simulated with 100 Hz. A sequence length of 1000 therefore means that the model gets the last 10 seconds as input. The architecture *GRU(64), GRU(128)* for example describes a model with two hidden GRU layers with 64 and 128 units followed by a dense layer with 6 units at the end.

4.4.2 CRNN

Convolutional neural networks are able to learn high level features from inputs. The idea behind a convolutional recurrent network (CRNN) is that the convolution layer extract meaningful patterns and features from the raw signal which are fed into a RNN cell. Thus, the convolution part is a feature extractor for noisy inputs while the RNN is able to learn temporal aspects. Several works have shown that this combination of CNN and RNN can increase the performance in the area of human activity recognition (see section 2). The question remains if this is also the case for pose estimation which is a regression problem.

Results for the CRNN architecture search are listed in table 4.4. The most important parameters of a convolution layer are the number of filters and the filter size. In this case it turned out that a filter size of 5 works best (compared to 3, 7, 10, 12 and 15). Therefore all the convolution layers below use a filter size of 5. The number of layers, number of filters and window length varies. From this search the following conclusions can be drawn:

- For a CRNN regularization and dropout improve the performance. Convolutions tend to overfit quite quickly even with a small amount of layers and filters. In order to reduce the generalization gap a L2 regularization of 0.001 was applied to each CNN layer. Additionally, a Dropout layer with a rate of 0,1 after each layer reduced the generalization gap further and resulted in better metrics.
- Again increasing the window length tend to increase the performance. In this case going from a window length of 500 to 1000 resulted in a considerable improvement in contrast to the previous RNN architecture search.
- Using a deep network with many convolutional layers did not lead to training errors such as with pure RNN's. However, using more than 3 convolution layers did not lead to any further improvement.
- Overall the metrics for the CRNN architectures (table 4.4) are not as good as for pure RNN (table 4.3). In most cases the position error is slightly above the pure RNN for a given window size. Especially yaw angles are often significant worse with errors up to 32 °.
- The best performing model is *CNN(32),CNN(32),GRU(256)* which reaches 4 meters position accuracy and 1,5 ° orientation error.
- Pooling layers [SMB10] and Batch Normalization layers [IS15] did not improve the result. Quite the contrary, using such layers resulted in poorer performance in most cases.

Architecture	**seq-len**	**position**	**yaw**	**pitch**	**roll**
CNN(16),CNN(16),GRU(64)	10	7,6	8,9	0,0028	0,0052
CNN(32),CNN(32),GRU(64)	10	8,4	6,1	0,0038	0,0032
CNN(16),CNN(16),GRU(256)	10	8,6	2,7	0,0036	0,0043
CNN(32),CNN(32),GRU(256)	10	9,5	4,6	0,0052	0,0032
CNN(16),CNN(16),GRU(64)	10	8,1	6,0	0,0016	0,0034
CNN(16),CNN(16),GRU(256)	10	8,8	3,0	0,0037	0,0020
CNN(32),CNN(32),GRU(256)	10	7,8	3,8	0,0032	0,0030
CNN(8),CNN(8),CNN(8),GRU(64)	10	7,4	8,37	0,0033	0,0034
CNN(16),CNN(16),GRU(64)	100	5,0	8,5	0,0031	0,0030
CNN(32),CNN(32),GRU(64)	100	6,6	6,4	0,0043	0,0044
CNN(16),CNN(16),GRU(256)	100	8,7	7,0	0,0060	0,0040
CNN(32),CNN(32),GRU(256)	100	6,6	22,0	0,0010	0,0075
CNN(16),CNN(16),GRU(64)	100	6,8	5,7	0,0024	0,0041

CNN(16),CNN(16),GRU(256)	100	9,6	4,6	0,0082	0,0043
CNN(32),CNN(32),GRU(256)	100	7,0	4,3	0,0033	0,0035
CNN(8),CNN(8),CNN(8),GRU(64)	100	6,5	12,0	0,0032	0,0053
CNN(16),CNN(16),GRU(64)	500	7,1	32,0	0,0068	0,0054
CNN(32),CNN(32),GRU(64)	500	7,2	10,2	0,0032	0,0034
CNN(16),CNN(16),GRU(256)	500	9,5	7,5	0,0049	0,0069
CNN(32),CNN(32),GRU(256)	**500**	**4,0**	**1,5**	**0,0018**	**0,0050**
CNN(16),CNN(16),GRU(64)	500	5,6	5,7	0,0026	0,0019
CNN(16),CNN(16),GRU(256)	500	8,4	3,0	0,0037	0,0087
CNN(32),CNN(32),GRU(256)	500	5,4	2,2	0,0044	0,0017
CNN(8),CNN(8),CNN(8),GRU(64)	500	6,7	13,0	0,0050	0,0048
CNN(16),CNN(16),GRU(64)	1000	5,5	5,9	0,0025	0,0034
CNN(32),CNN(32),GRU(64)	1000	5,6	5,1	0,0051	0,0071
CNN(16),CNN(16),GRU(256)	1000	6,2	2,8	0,0068	0,0053
CNN(32),CNN(32),GRU(256)	1000	7,7	2,3	0,0066	0,0065
CNN(16),CNN(16),GRU(64)	1000	6,6	4,3	0,0015	0,0029
CNN(16),CNN(16),GRU(256)	1000	5,7	1,3	0,0015	0,0061
CNN(32),CNN(32),GRU(256)	1000	5,0	1,4	0,0049	0,0016
CNN(8),CNN(8),CNN(8),GRU(64)	1000	9,1	13,5	0,0038	0,0053

Table 4.4: Result of the architecture search for convolutional recurrent neural networks (CRNN). The reported values in the columns position, pitch, roll and yaw are the mean absolute error over the test data. Position error is in meters and angle error in degrees. Column *seq-len* is the sequence length or window size and determines how many sensor measurements from the past the network gets as input. The architecture *CNN(32),CNN(32),GRU(256)* for example describes a model with two CNN layers which have 32 filters and a kernel size of 5 followed by a GRU cell with 256 units. At the end of each model is again a dense layer with 6 units.

4.4.3 Other Parameters

For the sake of completeness, more (hyper-)parameters are listed below:

- Adam optimizer with an initial learning rate of 0.001 worked best compared to other optimizer such as vanilla gradient descent or RMSProp.
- The network was trained on a NVIDIA Quadro P6000 GPU with 24 GB of memory. Training with a large batch size of 512 maximized training speed and didn't result in memory issues even with large window length of 1000.
- During preprocessing standard scaling worked best. Other scaling methods like MinMaxScaler, RobustScaler or PowerTransformer [2] led to poorer results.
- Usually training for more than 5 epochs did not improve the performance. Training for more than 10 epochs resulted in overfitting.
- At the final output layer no activation function is used. The last layer is a dense layer with 6 outputs and linear activation. Adding an non-linearity (like sigmoid, tanh or relu) in this layer led to bad results.
- Mean absolute error (MAE) as loss function yielded the best results. Other popular choices like mean squared error (MSE) or Hinge Loss resulted in poorer results.

Besides of CRNN's also pure CNN's were tested. A CNN means that the network contains only convolutional and dense layers. Therefore there are no RNN cells used in pure CNN's. However, this architecture showed the worst results. This makes sense, because CNN's are not able to process sequential time series data. Therefore they are not able to learn and extract temporal patterns which is very important for this problem.

[2] `https://scikit-learn.org/stable/auto_examples/preprocessing/plot_all_scaling.html` (10.01.2020)

4.5 Increasing Dataset Size

In the above mentioned architecture search the models were trained with roughly 2 hours of data. An interesting question is now what influence the amount of data has on the position accuracy. It is known that the power of deep learning evolves when large amounts of data are used. But how many hours of data do we need for this problem in order to use it in a real application?

Table 4.5 shows the results of increasing the dataset size. For this experiment the best model *GRU(256)* from the architecture search was used and trained for 5 epochs with varying amounts of training data.

Rounds	Driving Time	Position MAE	Yaw MAE
1	12 min	15 m	120 °
2	24 min	9 m	16 °
5	1 h	6 m	4,6 °
10	2 h	4 m	1,8 °
25	5 h	4 m	2,0 °
50	10 h	2,5 m	0,9 °
75	15 h	2 m	0,7 °
100	20 h	95 cm	0,3 °

Table 4.5: Effect on increasing the dataset size on the position accuracy. Rounds describes how often the presented round trip tour (section 4.2) was driven. In this case the model *GRU(256)* (one GRU layer with 256 units) was trained for 5 epochs.

As expected increasing the dataset size improves position and orientation accuracy clearly. Increasing the dataset size from 12 minutes to 20 hours improved the position accuracy from 15 m to 95 cm. Similar, yaw accuracy increased significantly. The increase is even stronger here (from 120 ° to 0,6 °).

However, the final position accuracy of 95 cm is still way too high in order to use it in an augmented navigation system. For the desired navigation application presented in section 1.1 a position accuracy of less then 20 cm is required. It could be the case that a deeper network architecture with more parameters and capacity would archive better results when training with more than 20 hours of data. Whether the position accuracy will fall below 20 cm with sufficient data cannot be predicted. But since the position accuracy does not seem to saturate there may still be room for further improvement when using even more data.

4.6 Analysis and Debugging of the Network

In the previous section it turned out that the presented models are able to learn something meaningful. However, 0,95 meter position accuracy in the best case is still far from being able to use it in a real world application. The aim of this section is it to analyze why the predictions are not as good and what could be the reasons for this.

4.6.1 Comparison between RNN and CRNN

The first question is it to identify if something is going wrong during the training process. The training loss and metrics are shown in table 4.6 for the best RNN and CRNN model.

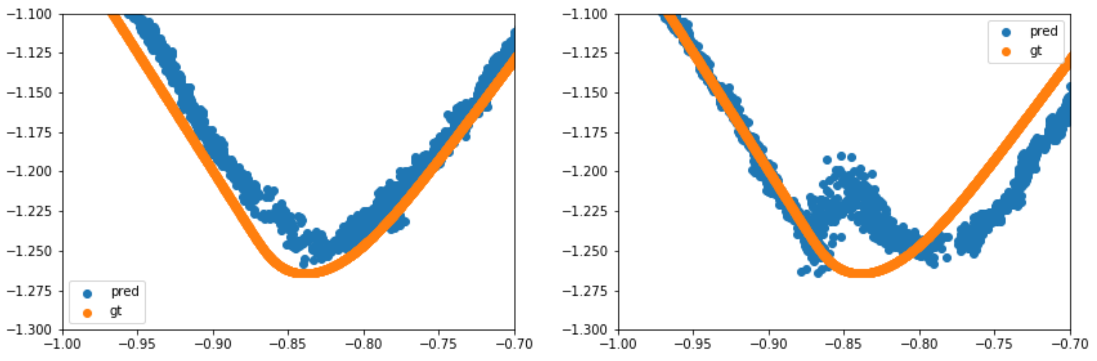
	1	2	3	4	5
RNN train loss	0.0768	0.0159	0.0086	0.0069	0.0053
RNN test loss	0.028	0.0124	0.0072	0.0075	0.0054
RNN position acc	11,5	8,4	6,5	4,3	3,7
RNN yaw acc	14,8	4,7	1,7	2,1	1,2
CRNN train loss	0.0943	0.025	0.00096	0.0142	0.0095
CRNN test loss	0.051	0.0139	0.009	0.0115	0.0075
CRNN position acc	28,8	10,1	6,5	7,4	4,6
CRNN yaw acc	24,5	4,5	2,4	3,7	2,1

Table 4.6: Loss and metric values during training. RNN is the best performing *GRU(256)* model from the architecture search. Similar, CRNN is the model *CNN(32),CNN(32),GRU(256)* which is the best CRNN architecture. Both models were trained for 5 epochs. Position accuracy values are in meters and calculated with respect to the test set. Yaw accuracy values are in degree.

For both networks the loss decreases very fast during the first 3 epochs and more or less converges afterwards. Also there is no indicator for overfitting. In both cases no generalization gap is visible since the test loss is not significantly higher than the training loss.
The loss for the RNN model is constantly lower than the CRNN loss which results in better position and orientation accuracy.

Overall there are no inconsistencies or oddities visible. They training process seems fine, because we have typical loss curves and the model is able to learn and improve it's performance during training.

If you just look at the numbers and compare the metrics, the difference between the RNN and CRNN does not seem too big. Although RNN metrics are better they still are in the same region. Quantitatively there is not much difference between RNN and CRNN. However, there is a qualitative difference as it can be seen in figure 4.5. If you look at the predictions of both models they RNN predictions are better and preferred, because they are much more stable. The CRNN predictions contain often high errors and major fluctuations which can result in kind of a messy output. Especially when not driving a straight line, for example a curve, the deviations from the ground truth trajectory can be quite big (see figure 4.5). The errors are much more stable and evenly distributed in the RNN case. This is why the RNN predictions are preferred over CRNN outputs.



(a) RNN (b) CRNN

Figure 4.5: Predictions and ground truth data for both models. In difficult driving situations CRNN outputs can become messy and unstable.

Something that stands out when you look at the predictions of both networks is a constant offset between predictions and ground truth (see figure 4.6). In most cases the predictions follow the ground truth closely and have the same shape and trajectory. However, the center of the predictions if often slightly off the ground truth values. This offset is not constant across the whole test set such that it could be easily manually removed. Usually this offset is only constant for a short period of time and then changes.

One guess was that this is due to the scaling method of the dataset. As described

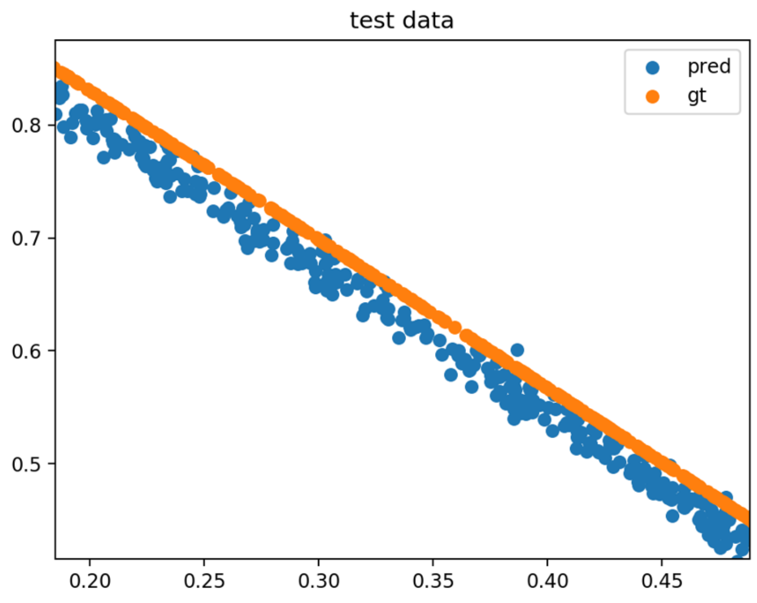


Figure 4.6: RNN and CRNN predictions often contain a constant offset which changes over time.

previously standard scaling was used in this case, which scales the data such that it has a mean of 0 and a variance of 1. However, switching to other scaling methods, like MinMaxScaler, RobustScaler or PowerTransformer, resulted in the same offset in the predictions or decreased the overall performance. Other methods that were used to fix this problem was using a different coordinate system (section 4.8.4). However, all experiments and techniques which were tested to eliminate this prediction offset were not successful.

4.6.2 Monitoring Weights and Gradients

To have closer look and identify possible training problems *Tensorboard* [3] can be used. Tensorflow is a visualization toolkit included in Tensorflow and can track and visualize losses and metrics. Furthermore, it can be also used to view and track weights and gradient changes over time. This enables one to have a closer look into what is going on during training time. The changing of the weights over time is shown in figure 4.7.

It can be seen that the weights surprisingly don't change much over time and across epochs. Most weight distributions look pretty much the same for every epoch. Addi-

[3] `https://github.com/tensorflow/tensorboard` (20.10.2020)

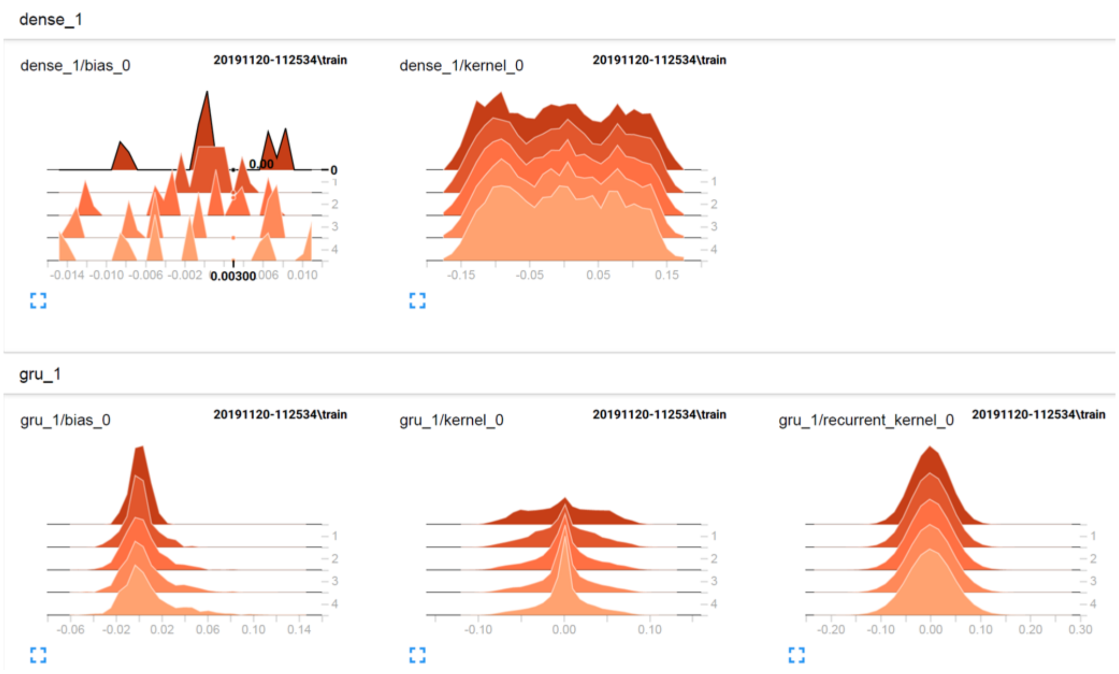


Figure 4.7: Tensorboard weight distribution during training for model *GRU(256)*. Here the model was trained for 5 epochs. The weight histograms are stacked vertically for each epoch. Layer *gru_1* is the only hidden layer and layer *dense_1* is the output layer.

tionally, the weights for the hidden GRU layer form a normal distribution where most of the weights are zero or very close to zero. This is a indicator that the model does not learn very well. Gradient changes are not shown in this figure.

To understand why this is the case it helps to look how the gradients change over time since they determine how much the weights will be adjusted. Surprisingly, the gradients look very similar to the weight distributions meaning that the gradients are most of the times zero or very small. Again, this vanishing gradients show that the model does not learn very much.

To overcome this vanishing gradients problem the following steps have been taken:

- Use a smaller model with less parameters.
- Use different RNN cells like LSTM or a CRNN architecture.
- Use a different optimizer instead of Adam (like SGD or RMSProp).

- Decrease the learning rate and use a learning rate decay.
- Use a different scaling method during preprocessing.

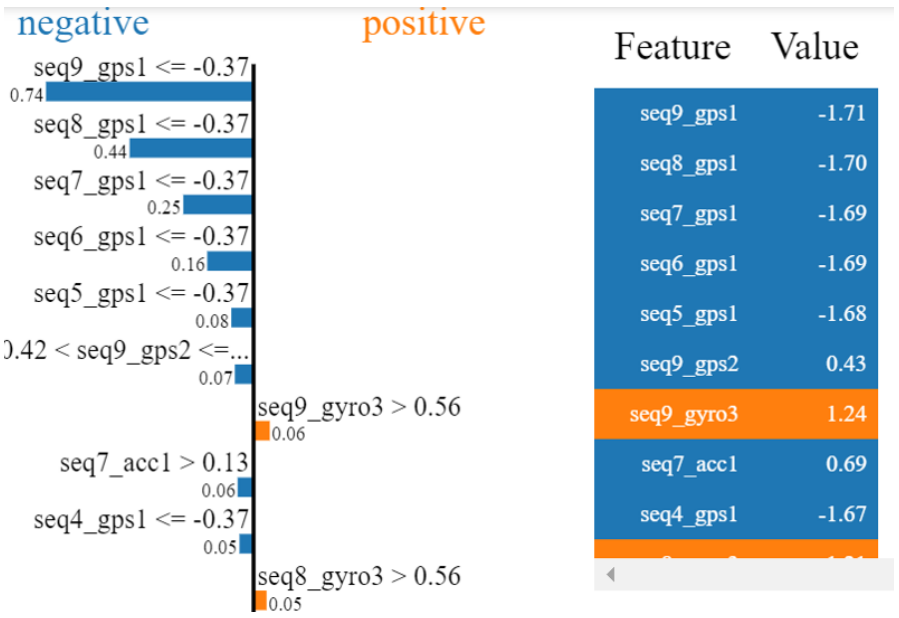
Unfortunately, all these actions have not solved the vanishing gradient problem. It remains unclear why the weights and gradients are so small.

4.6.3 Explaining Model Predictions

Traditional methods for pose estimation, like a Kalman filter, often contain human engineered features or models. Thus, this models can be understood, analyzed and debugged quite easily. Neural networks on the other hand are kind of a black box. Basically we don't know what and how the model learned exactly and why it comes to a specific decision or prediction. *Explainable artificial intelligence* (XAI) aims to tackle this problem. This involves techniques to open the black box such that the models and result can be understood [Arr+19].

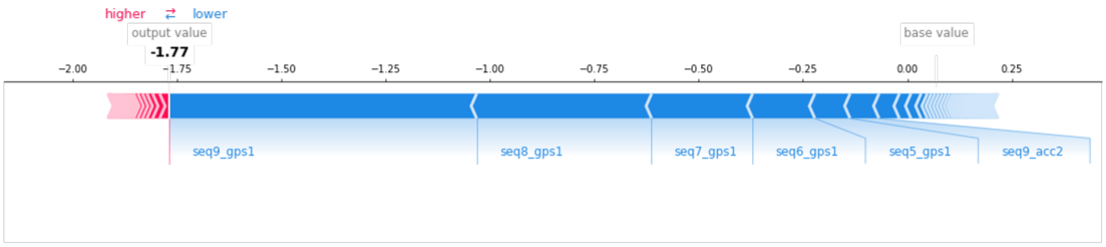
The frameworks *LIME* [RSG16] and *SHAP* [LL17] are examples for an explainable AI framework. Both can be used to explain the outputs of any machine learning classifier (see figure 4.8).
Both frameworks only support a single output. Therefore a models was trained on predicting the latitude values of the position based on GPS and IMU inputs. In order to keep things clear the model was trained with a window length of 10 meaning that it gets the last 10 measurements as input. Thus the models gets three values from the GPS sensor (gps1, gps2, gps3), accelerometer (acc1, acc2, acc3) and gyroscope (gyro1, gyro2, gyro3) at every time step (seq0,...,seq9).
One can clearly see in the output explanations that the model relies basically only on the GPS inputs for predicting the position. In other words, *the model does not include the IMU measurements for predicting the position*. This explains, why the position accuracy is as bad as presented in the previous sections. The GPS inputs are quite noisy (RMS error of 5 meters). Therefore it is obvious that the model can not predict the position accurate to the centimeter when relying only on the GPS inputs. In order to predict the position precisely the model should learn to include IMU measurements into its predictions. This would be even more important in a real dataset where GPS measurements come at 1 - 3 Hz and not with 100 Hz as in this simulated dataset.

But why the model does not learn to handle IMU measurements remains unclear. *LIME* and *SHAP* can explain what the model does at inference, but not why it does something or not. It might be the case that IMU data is just to complex and requires huge amounts of training data.



Feature	Value
seq9_gps1	-1.71
seq8_gps1	-1.70
seq7_gps1	-1.69
seq6_gps1	-1.69
seq5_gps1	-1.68
seq9_gps2	0.43
seq9_gyro3	1.24
seq7_acc1	0.69
seq4_gps1	-1.67

(a) LIME



(b) SHAP

Figure 4.8: Explainable AI frameworks such as *LIME* or *SHAP* can explain model predictions. In this case the model predicts the latitude value based on GPS and IMU data. The model was trained with a window length of 10 meaning that it gets the last 10 measurements from each sensor. The value *seq9_gps1* for example describes the last of the 10 input values from the gps sensor. It can be seen that the model uses basically only the GPS values for calculating a position estimate.

4.7 Comparison to other Algorithms

As described in the previous sections the position accuracy is not very good and it remains unclear why this is the case. Therefore the question arises if this is a specific deep learning issue. In order to answer this other (machine) learning algorithms were applied on the same dataset (see table 4.7).

Method	**Position MAE**
AutoML (H2O)	2,5 m
Linear Regression	3,4 m
Decision Tree	11,8 m
LastValue	4,5 m
Deep Learning (proposed approach)	3,5 m

Table 4.7: Position accuracy of other algorithms when training on 2 hours of data. In this case the best H2O model was a Gradient Boosting Machine (GBM). The algorithm LastValue acts as a baseline and only returns the last GPS sensor measurement as prediction.

Automated machine learning (AutoML) automates the training of a machine learning model. AutoML takes care of the whole process of training a model, from the raw dataset to a deployable model, without the user having to intervene or make decisions. Thus it allow also non exerts to use machine learning.
An example for an AutoML frame work is *H2O* [4]. H20 automates the process of model selection and training as well as parameter tuning. This involves testing a wide range of different algorithms (like random forests, gradient boosting machines, support vector machines, deep learning and more). The tuning of a model, or a ensemble of models, is done by a (random) grid search. H20 was used in this context because it allows one to test multiple algorithms at once via a single and easy to use interface.

In this case the best model trained by H20 was a *Gradient Boosting Machine (GBM)* which builds an ensemble of regression trees. This is done by iteratively growing multiple regression trees in parallel. This model archived a position accuracy of 2,5 meters. A standard decision tree model [5] archived only 11,8 meter position accuracy.

[4]`https://www.h2o.ai` (20.01.2020)
[5]`https://scikit-learn.org/stable/modules/preprocessing.html` (20.01.2020)

Training a linear regression model resulted in a position accuracy of 3,4 meters and is therefore just as good as the deep learning approach (3,5 meters). This is surprising because a linear regression is rather simple and was not specifically designed for processing time series like recurrent neural networks.
It is somewhat surprising that models like linear regression and GBM are in the same region although these models are not specifically designed for handling time series data like a RNN.

All the models mentioned above were trained on 2 hours of data. Unfortunately, it was not possible to use more data for the non deep learning algorithms. Using more data resulted either in extremely long computing time or very high memory usage since linear regression and decision trees don't scale well. Even with 256 GB of memory it is not possible to train these models with several hours of data.

On the other hand the memory usage neural networks is constant and independent of the dataset size since one only has to load the current batch into memory. Furthermore, computation time scales linear meaning that twice as much data only doubles training time. Therefore deep learning is the only algorithm which can be easily trained on large datasets. As described in section 4.5 the position accuracy can be improved to 95 cm when using up to 20 hours of training data. Therefore deep learning clearly outperforms the other algorithms, but only when training with sufficient data.

4.8 Further Experiments

In order to better understand and perhaps improve the result, a number of experiments were done. However, none of the following experiments brought a significant improvement of the result. For the sake of completeness they are nevertheless listed and briefly described.

4.8.1 Overfitting and Ground Truth as Input

One common approach of training neural networks is it to overfit on a small amount of data or even a single training example. By this procedure one can verify that the data preprocessing and training pipeline works. Additionally, overfitting shows that the network is in principle able to process the data and learn from it even though it is just memorizing the few training examples.

Surprisingly, in this case overfitting was not really possible. Even with training on very few training examples (10 to 100) and for many epochs (more than 20) the loss was not significantly lower than by training on the full dataset. Also the offset between training and validation loss (generalization gap) was not that large. The model archived a position accuracy of 1,8 meters which is surprisingly bad for this overfitting experiment.

One can take this overfitting experiment even further. In this case the model was again trained on a few data points. But this time the GPS sensor measurements were replaced with the ground truth position. This means that the inputs and target values are exactly the same. Very surprisingly the model only archived a position accuracy of 1,6 meters. When you consider that the network in this case only have to output the last GPS input in order to archive a position accuracy of almost zero the result is rather poor.

This overfitting experiments indicate that there is probably a more general problem when applying deep learning to this problem. It seems that the networks somehow is not really able to process this kind of GPS input data. It remains an open question why this is the case.

4.8.2 Low GPS Update Rates

In order to simplify the problem the GPS measurements were simulated with 100 Hz so far. In reality such high update rates are unrealistic. Especially low cost GPS senors often provide update rates of 1-3 Hz. In order to check if the network can handle different sensor resolutions and low GPS update rates the same dataset was simulated with 1 Hz GPS measurements. IMU measurements were again simulated with 100 Hz.

The results are shown in table 4.8. As one can see, a low resolution GPS sensor does not lead to poorer performance. The position accuracy is in this case 3,7 meter (compared to 3,5 meter with 100 Hz GPS). Surprisingly, the yaw accuracy is much better in this experiment (0,62 ° compared to 1,5 °).

Epoch	**Position**	**Yaw**
1	9,6 m	1,35 °
2	5,8 m	1,12 °
3	4,6 m	0,94 °
4	4,4 m	0,85 °
5	3,7 m	0,62 °

Table 4.8: Training the model GRU(256) with a GPS update rate of 1 Hz. IMU data was simulated at 100 Hz.

4.8.3 Relative Prediction

So far the model always got absolute coordinates (latitude and longitude) as inputs and also predicted the absolute position. In this experiment the model was trained on relative GPS inputs. This means that the GPS input at each timestep is the relative change to the previous timestep. Thus, the model gets the position change (deltas) as input and not absolute coordinates. As an example the absolute input sequence (47°, 48°, 48.5°, 48.2°) would be transformed to the relative input (0°, 1°, 0.5°, -0.3°).

Similar, the ground truth data was changed such that the model has to predict the position 1 second in the future relative to the current position (the first position from the input). Thus, the model has to *predict a translation and not a absolute position*. The aim of this experiment is it to find out if absolute coordinates as inputs and ground truth are a problem.

Table 4.9 shows the results for this experiment. In the best case the position error was 8,2 cm which is much lower than with absolute coordinates. However, one must keep in mind that this error sums up at each prediction step. Thus, the model outputs would be unreliable after a few predictions. In this case the model *GRU(256)* was trained to predict the pose 1 second in the future. Different time intervals resulted in similar results. Therefore we can conclude that absolute coordinate inputs and predictions doesn't seem to be a major problem.

Epoch	Position	Yaw
1	11 cm	0,20 °
2	10 cm	0,16 °
3	10 cm	0,17 °
4	9,7 cm	0,15 °
5	8,2 cm	0,16 °

Table 4.9: Prediction errors for the relative prediction experiment. Numbers show the mean absolute error (MAE).

4.8.4 Coordinate Transformation

The position inputs and targets so far were latitude and longitude. It could be the case that the network has problems handling latitude and longitude values. In order to test this the GPS measurements and ground truth positions were transformed into other coordinate systems. One of the coordinate systems tested is *ECEF (earth-centered, earth-fixed)* which is a Cartesian coordinate system where the origin (0,0,0) is defined as the center of the earth. Therefore positions in ECEF are represented as X, Y and Z coordinates (see figure 4.9).

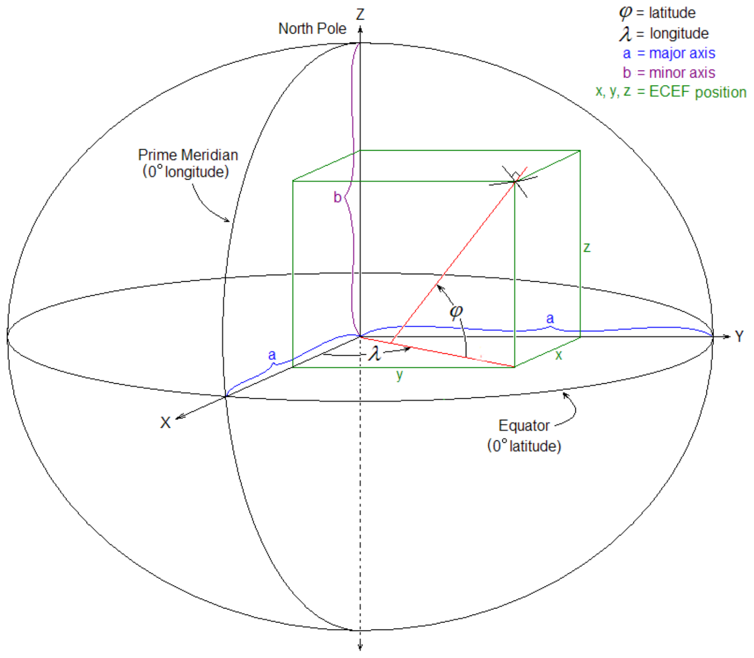


Figure 4.9: Comparison between different coordinate systems.

source: `https://upload.wikimedia.org/wikipedia/commons/8/88/Ecef.png` (23.01.2020)

One assumption was that this improves the prediction, because the IMU measurements are also represented in a Cartesian coordinate system. Therefore the model only has to learn handling Cartesian coordinates and not geodesic coordinates as well. However, the position accuracy was pretty much the same with ECEF coordinates. Another coordinate systems tested was *Universal Transverse Mercator (UTM)* which didn't improve the results either.

4.8.5 Bidirectional RNN Cells

Normal RNN cells contain one hidden layer. When a cell is called with a sequence of data a forward pass is computed where at each timestep the cell processes the current element from the sequence. A *bidirectional recurrent neural network (BRNN)* extends those standard cells with an additional second hidden layer. The difference between both hidden layers is that one layer processes the input sequence backwards (see figure 4.10). Thus it allows the cell to get information from the past and the future which in theory results in more accurate predictions.
Each of the two hidden layers in a BRNN compute one hidden state. In order to compute a final output for the layer those two internal states have to be merged or combined into one final output state. Possible ways of doing that is it to sum up both states or take the position wise average.

BRNN cells improved the performance of deep learning systems especially in the area of natural language processing which involves tasks like speech recognition or translation. In this case however the performance didn't improve when using bidirectional RNN cells.

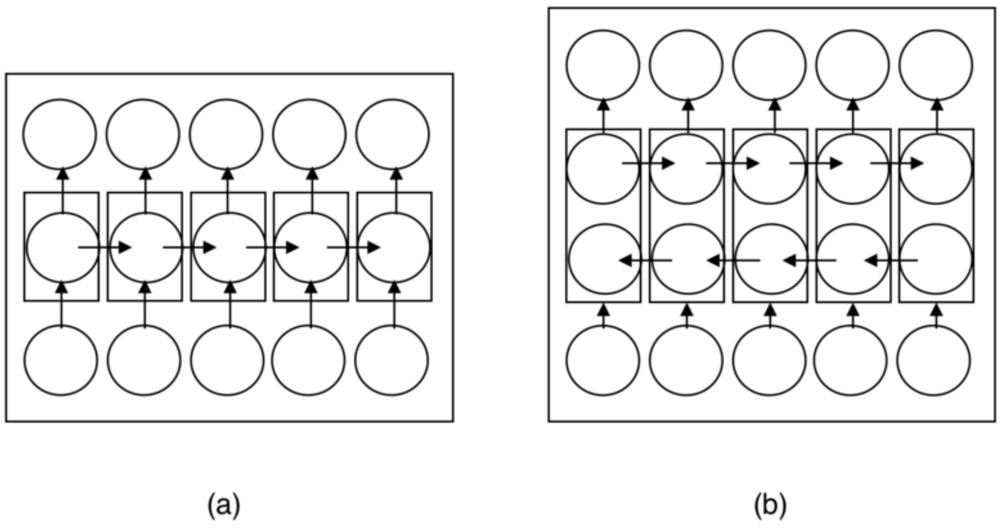


Figure 4.10: Comparison between unidirectional RNN (a) and bidirectional RNN (b). In the bidirectional case the sequence is additionally processed backwards.

source: `https://upload.wikimedia.org/wikipedia/commons/3/35/Structural_diagrams_of_unidirectional_and_bidirectional_recurrent_neural_networks.png` (22.01.2020)

4.8.6 Stateful RNN Cells

Usually the internal state of a RNN cell will be deleted after each training example. Therefore a series of predictions are independent of each other, because the "memory" of the network is cleared. Another way of training a RNN is so called stateful training. Here one does not delete the internal state after each training point. Therefore the model can keep it's internal state for the next prediction.

In theory this enables the model to learn long dependencies. In this work the window length was varied up to 1000 which corresponds to 1 second of data. If one does not use stateful training the model can only compute it's prediction based on this short time window. It could be the case that there is some useful information further in the past which would help to make a more accurate prediction. By using stateful RNN cells the model could in theory be able to exploit those long term dependencies.

Due to unknown reasons stateful training did not work at all in this case. During training the model did not learn and the loss did not improve or even diverged. None of the tested architectures was able to produce any useful results.

4.8.7 Using Existing Architectures

A lot of deep learning research is done in the area of human activity recognition (HAR) where one has to classify sensor data into categories. The idea behind this experiment was it to use an existing good performing HAR architecture and apply it to the pose estimation problem.

One interesting architecture is *DeepSense* [Yao+16] which archives state of the art performance in HAR classification tasks. The architecture is basically a CRNN with several convolutional layers in the beginning followed by two GRU layers (see figure 4.11). By replacing the output layer by a dense layer with 6 units this architecture can also be easily applied to regression problems. However, the architecture did not outperform the models tested in the architecture search. The implemented DeepSense model archived a position accuracy up to 6 meters.

Another good performing architecture is *SensorNet* [Jaf+19] which is a CNN architecture and does not include any RNN cells. A specialty of SensorNet is the usage of 2D convolutions instead of 1d convolution. In order to apply 2d2D convolution to time series and sensor data the inputs are put together into one image (see figure 4.12). By stacking each sensor output into a two dimensional image 2D convolution

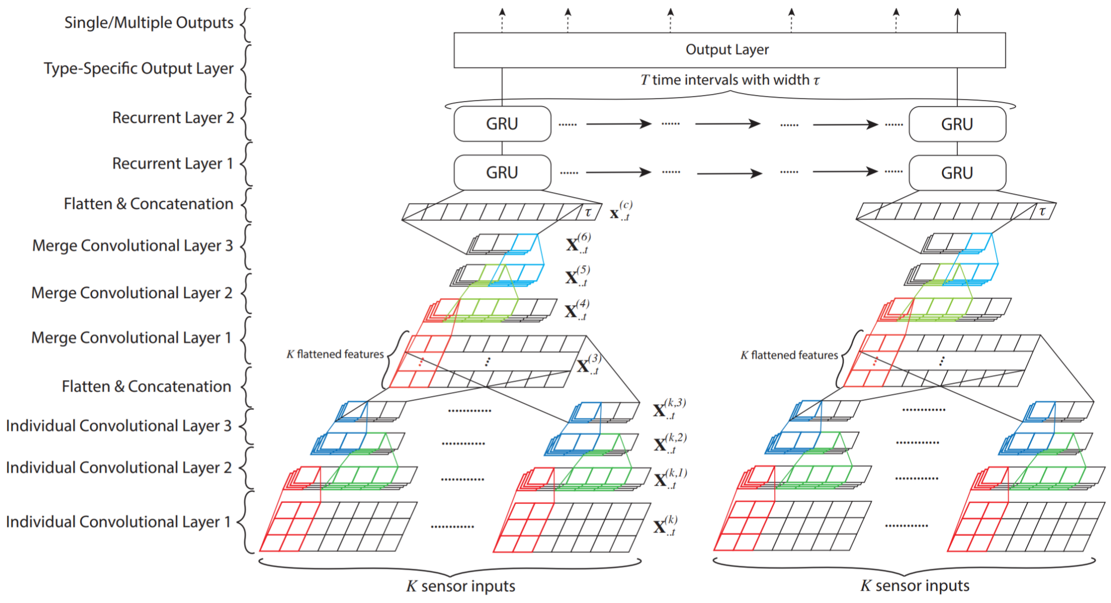

Single/Multiple Outputs
Type-Specific Output Layer
Recurrent Layer 2
Recurrent Layer 1
Flatten & Concatenation
Merge Convolutional Layer 3
Merge Convolutional Layer 2
Merge Convolutional Layer 1
Flatten & Concatenation
Individual Convolutional Layer 3
Individual Convolutional Layer 2
Individual Convolutional Layer 1
Output Layer
T time intervals with width τ
GRU
K flattened features
K sensor inputs

Figure 4.11: Architecture of DeepSense [Yao+16].

can be applied such as in normal pictures. There also exist several other works that show that 2D convolutions can improve the classification performance [HYC16; HC16; JY15]. Unfortunately, using the SensorNet also did not improve the results. SensorNet archived a Postion accuracy of 7 meters.

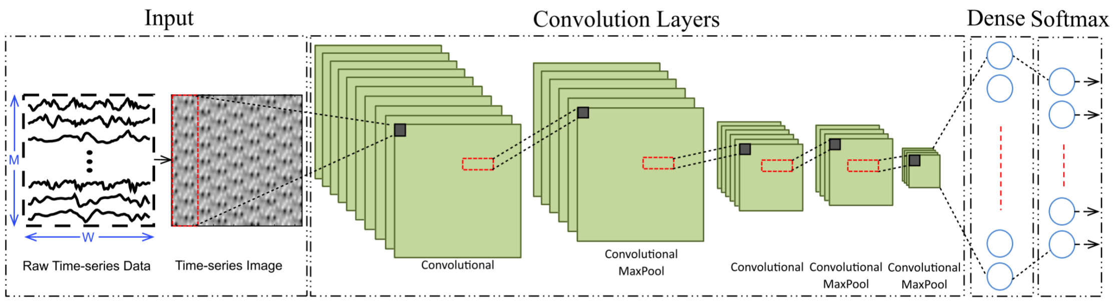


Figure 4.12: SensorNet preprocessing and architecture [Jaf+19].

To summarize, the performance couldn't be improved by using existing architectures. This is probably due to the fact that those architectures were developed only for IMU sensor data and classification tasks.

4.8.8 Noise Layer for Data Augmentation

One common problem of training neural networks is the lack of data. *Data augmentation* is a technique to artificially increase the dataset size. Data augmentation is heavily used in the computer vision area where images can be easily rotated, flipped or resized. This also makes the model more robust against noise and distortions in the input data. The goal of this experiment was is to apply data augmentation to sensor data. This was done by adding an additional noise layer at the beginning of the network. This layer adds every time a small noise to the sensor inputs. In this case a gaussian noise with different standard deviations (from 0,0001 to 0,1) was used.
However, adding a noise layer did not improve the performance. This is probably due to the fact that the sensor measurement already contain a lot of noise such that adding an additional artificial noise on top makes the problem much harder.

4.8.9 Position Prediction

So far the networks always was trained on predicting 6 values (latitude, longitude, altitude, pitch, roll and yaw). However, the position accuracy is not very good. In order to focus on an accurate position estimation the model has to predict only latitude and longitude in this experiment. Thus, the problem complexity is reduced and the optimizer can focus on learning the position. However, only predicting a 2D position did not improve the result.

4.8.10 Excluding IMU Inputs

In section 4.6.3 explainable AI frameworks showed that the model only relies on the GPS inputs for estimating the position. In order to verify this claim the model was trained only on GPS inputs in this experiment. Although the IMU inputs were excluded in this case the position accuracy did not change. This shows that the model really does not rely on the IMU measurements for predicting the current position. Why this is the case remains an open question. It could be the case that processing noisy IMU data is just to complex.

4.8.11 Different IMU Accuracy and Resolution

One benefit of using simulation data is it that the amount of noise in adjustable. So far the model was trained on medium accuracy IMU data (see section 4.1). In this experiment we train the model also on high accuracy IMU data in order to see if the noise of the IMU data is the problem. Table 4.10 show the results of training the model *GRU(256)* for 5 epochs on medium and high accuracy data. One can draw the following conclusions:

- High accuracy IMU data lead to more accurate orientation prediction. The yaw error is better across all epochs for high accuracy data. The final mean yaw error is only half as large at the end.
- The position error is not improved and reaches the same accuracy. However, the position error is smaller at the first epochs and converges more quickly.

Epoch	Position	Yaw
1	7,5 m	3,7 °
2	4,5 m	2,0 °
3	5,4 m	4,5 °
4	5,0 m	1,3 °
5	3,2 m	1,3 °

Medium Accuracy

Epoch	Position	Yaw
1	4,8 m	1,2 °
2	4,7 m	0,8 °
3	4,0 m	0,7 °
4	3,1 m	0,7 °
5	3,3 m	0,6 °

High Accuracy

Table 4.10: Training results for training on medium and high accuracy IMU data.

One can conclude that high accuracy IMU data reduce the orientation error, but the position error stays the same. Therefore it seems that the network is not able to include IMU data for the position estimation even if it contains less noise.

4.8.12 Fourier Transformation

There exists some works in the area of human activity recognition which claim that transforming the sensor inputs to the frequency domain improves the detection [Yao+16]. By applying a Fourier transform to the inputs it should make it easier to recognize and detect patterns. However, in this case this approach did not work at all. The model was not able to learn anything if Fourier transformed inputs were used. This is probably due to the fact that here we deal with a regression problem whereas human activity recognition is a classification problem.

4.8.13 Filtering as Preprocessing

Another experiment was it to filter the inputs as an additional preprocessing step instead of passing the raw sensor measurements to the neural network. To do this the author applied some well-known signal processing filters in order to clean the data and remove noise. A bunch of filters were tested including median, mean, Wiener and Hilbert filter [6]. But all these tested filters were not very effective and did not improve the result.

4.8.14 Implementation in PyTorch

The model was implemented with *Tensorflow 2.0* [7]. Since we can observe the previous mentioned issues the question arises if this is related to a bug or implementation error within Tensorflow. In order to check this the same preprocessing and training pipeline was implemented in *PyTorch* [8].

However, the results with the PyTorch implementation were pretty much the same. Loss, position accuracy and orientation accuracy were in the same range. Also the mentioned offset between predictions and ground truth was observable in this case.

[6]`https://docs.scipy.org/doc/scipy/reference/tutorial/signal.html` (17.12.2019)
[7]`https://tensorflow.org`
[8]`https://pytorch.org`

4.9 Summary

The previous sections can be summarized as followed:

- Predicting the orientation of a vehicle works well with a mean average error of around 1,5 °.
- The position estimation is significantly worse with an mean average error of about 95 centimeter in the best case.
- In order to archive a position accuracy of 0,95 meter a lot of training data (more than 20 hours of driving time) is needed.
- Recurrent neural networks (RNN) outperform convolutional neural networks (CRNN) both quantitatively and qualitatively.
- During training small weights and vanishing gradients could be observed but not resolved.
- The network does not include the IMU data for the position estimation which explains why the position error is that high. Why this is the case remains an open question.

CHAPTER 5

Training on Real Data

So far simulation data was used in order to get first insights and challenges of the problem. Now it will be tested how the developed approach and models perform on real world data.

Figure 5.1: OXTS RT3000 real time kinematic system.

source: `https://www.oxts.com/products/rt3000` (28.01.2020)

In order to gather ground truth data a professional and very accurate real time kinematic (RTK) system is needed. Here the *OXTS RT3000* (see figure 5.1) from the company Oxford Technical Solutions was used. This RTK system is able to precisely measure the position and orientation with a very high accuracy (see table 5.1). With 1 cm position accuracy and data output rates of up to 250 Hz this system allows one to gather high quality ground truth data.

Position accuracy	1 cm
Velocity accuracy	0,05 km/h
Roll/pitch accuracy	0,03 °
Heading accuracy	0,1 °
Track angle accuracy	0,07 °
Slip angle accuracy	0,15 °

Table 5.1: OXTS RT3000 accuracy values.

Since gathering real world data is time and cost intensive within the scope of this work the author was only able to record a dataset containing 1,5 hours of driving time. To the best of the authors knowledge this dataset is unique since there exists no public dataset which contains measurements from car sensors as well as highly accurate 6D ground truth data.

5.1 Dataset Analysis

The presented dataset contains the measurements from 5 sensors. The sensor measurements come at different update rates. All in all these 19 input features are available for training:

1. **GPS (8 values)** at 1 Hz: Besides the position (latitude, longitude and altitude) there is also the GPS heading and speed as well as 3 accuracy values included.
2. **Accelerometer (3 values)** at 100 Hz: X, Y and Z measurement from the accelerometer.
3. **Gyroscope (3 values)** at 100 Hz: X, Y and Z measurement from the gyroscope.
4. **Wheel ticks (8 values)** at 100 Hz: 4 RPM (revolutions per minute) values for each wheel and 4 values which indicate if a wheel spins backwards.

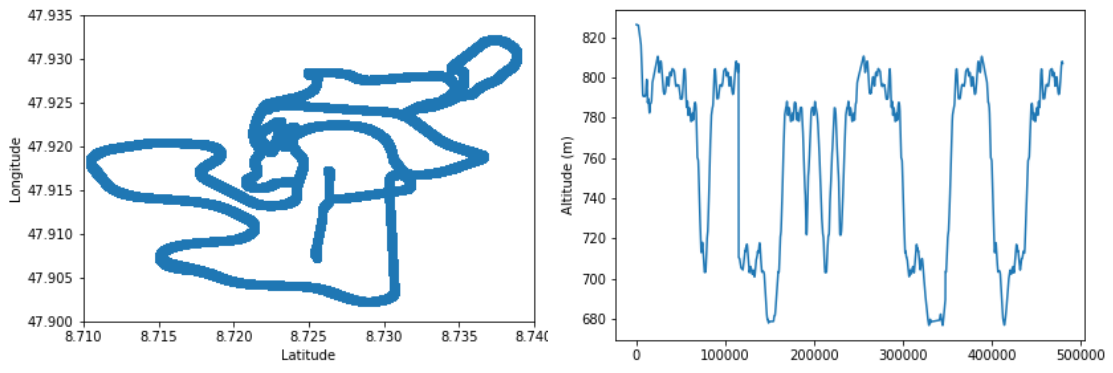


Figure 5.2: Position and Altitude values of the recorded dataset. This dataset contains 1,5 hours of driving time.

5. **Wheel angle (1 value)** at 10 Hz: Wheel angle of the front axle.

One thing that makes real time pose estimation very hard is the low update rate of the GPS sensor which only provides measurements at 1 Hz. Figure 5.3 shows the distribution of the time delay between two GPS measurements. The median value is 1 second which corresponds to the 1 Hz update rate. There are not many and no major outliers recognizable. The GPS measurements therefore contain no significant latency or jitter.

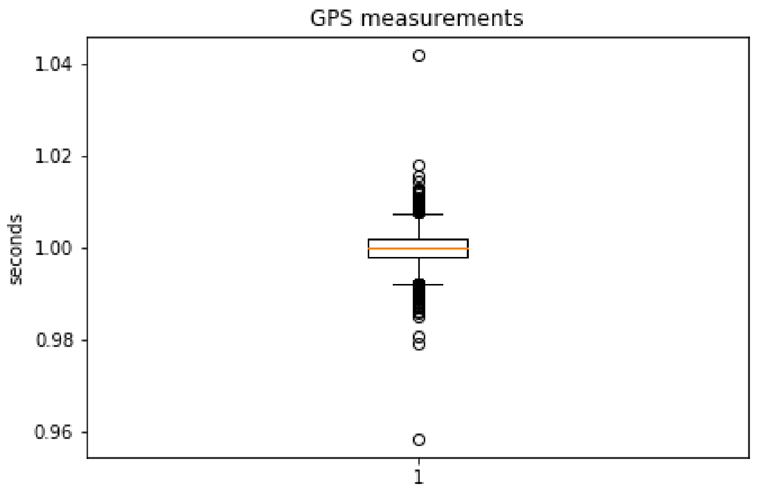


Figure 5.3: Boxplot of the time difference between two GPS measurements.

Since the dataset was mainly recorded on country roads and on a sunny day the GPS position accuracy is very good as it can be seen in figure 5.4. The median position error is at 2,1 meters in this case which is very good for such an low cost GPS sensor.

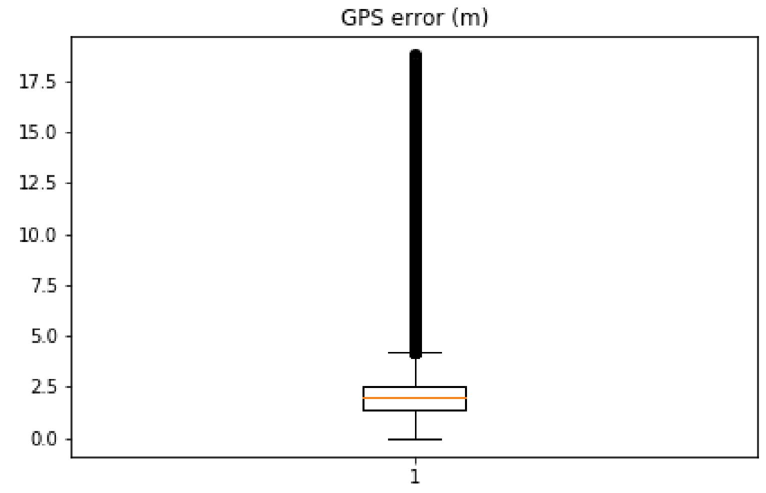


Figure 5.4: Position accuracy of the GPS measurements with respect to the ground truth position values.

Therefore we can conclude that the GPS measurements only come at 1 Hz but the quality and accuracy of the data is high.

5.2 Data Preprocessing

A small schematic excerpt of the dataset is shown in table 5.2. Since one part of the measurements come directly from the vehicle sensors and the other parts form the external OXTS system the data is unsynchronized. Each measurement gets a timestamp when it is received by the logging application.

Timestamp	**Sensor**	**x1**	**x2**	**x3**	**x4**	**x5**	**x6**	**x7**	**x8**
3	GPS	47.8956	9.4578	845.7	12,3	3,0	2	1	1
7	ACC	4,3	2,6	9,81	-	-	-	-	-
8	GYRO	1,24	0,007	0,05	-	-	-	-	-
10	WT	5,3	5,4	5,33	5,38	0	0	0	0
13	WA	8,7	-	-	-	-	-	-	-
15	GT	47.8970	9.4655	833.1	14	3	1	-	-

Table 5.2: Dataset overview. The measurements from different sensors arrive unsynchronized. The input sensors include GPS, accelerometer (ACC), gyroscope (GYRO), wheel ticks (WT) and wheel angle (WA). The ground truth data (GT) includes the position (latitude, longitude and altitude) as well as the orientation (pitch, roll and yaw). Depending on the sensor, each measurement contains a different number of values (x1 to x8).

In order to train the network one needs to generate training data where we have the input values together with the respective target values present at each training sample. Thus, one needs to synchronize the sensor measurements. This can be done by using nearest neighbor or linear interpolation. Both interpolation techniques were tested, but no significant difference between the two was found. Linear interpolation has the disadvantage that applying it during inference is not straightforward and requires additional computation.

5.3 Network Architecture Search

Similar to section 4.4.1 an architecture search was done on the real dataset. Compared to simulation data this collected dataset has some additional challenges which makes the problem probably harder:

- GPS measurements come only at 1 Hz. During simulation the GPS data was simulated at 100 Hz.
- Sensor measurements arrive unsynchronized.
- Real data typically contains more and larger noise.
- Real data can contain sensor outages, latency or jitter.

For the following the dataset was split into train and test data. In this case 80 % of the data were used for training and 20 % for testing.

5.3.1 RNN

Table 5.3 lists the results for recurrent neural networks. The models presented here are only a selection of all tested models. One can draw the following conclusions:

- Similar to the results with simulation data a sequence length of 500 to 1000 works best and improves the performance compared to shorter input lengths. With simulation data there was no significant position accuracy increase when using a sequence length of 1000 compared to 500. In this case however, a sequence length of 1000 improves the position accuracy quite a bit. Some models like *GRU(64)*, *GRU(256)* and *GRU(64),GRU(64),GRU(64)* archive a twice as accurate position estimate when using a sequence length of 1000 instead of 500. Again, using a window length higher than 1000 resulted in training errors.
- The orientation estimation works pretty well. The mean absolute error of pitch, roll and yaw is often below 0,3 °. There are no major discrepancies between pitch, roll and yaw. Although pitch values are slightly more accurate than roll and yaw.
- The best performing model is just as with simulation data *GRU(256)*. This model contains one GRU layer with 256 units and archives a position accuracy of 3,6 meter which is almost exactly the same as with simulation data. This is surprising, because real data come with the mentioned challenges above. The main difference between the simulation data and this real dataset is the GPS update rate (100 Hz compared to 1 Hz). This shows that the model is also able to do pose estimation with low sensor update rates.

Architecture	**seq-len**	**position**	**pitch**	**roll**	**yaw**
GRU(64)	100	9,8	0,17	0,24	0,50
GRU(128)	100	9,6	0,13	0,24	0,33
GRU(256)	100	8,1	0,086	0,23	0,14
GRU(64),GRU(64)	100	13,5	0,16	0,24	0,34
GRU(128),GRU(128)	100	9,9	0,1	0,24	0,22
GRU(256),GRU(256)	100	11,8	0,1	0,23	0,16
GRU(64),GRU(64),GRU(64)	100	14,3	0,13	0,24	0,28
GRU(128),GRU(128),GRU(128)	100	12,0	0,1	0,24	0,15
GRU(256),GRU(128),GRU(64)	100	13,1	0,12	0,24	0,13
GRU(64)	500	8,6	0,14	0,24	0,30
GRU(128)	500	7,5	0,11	0,24	0,17
GRU(256)	500	6,3	0,07	0,23	0,16
GRU(64),GRU(64)	500	9,7	0,16	0,24	0,34
GRU(128),GRU(128)	500	8,5	0,08	0,24	0,19
GRU(256),GRU(256)	500	10,9	0,07	0,23	0,15
GRU(64),GRU(64),GRU(64)	500	12,3	0,1	0,24	0,21
GRU(128),GRU(128),GRU(128)	500	9,3	0,07	0,24	0,17
GRU(256),GRU(128),GRU(64)	500	10,8	0,07	0,23	0,15
GRU(64)	1000	4,9	0,15	0,25	0,22
GRU(128)	1000	7,3	0,12	0,26	0,13
GRU(256)	**1000**	**3,6**	**0,12**	**0,22**	**0,15**
GRU(64),GRU(64)	1000	5,3	0,19	0,24	0,32
GRU(128),GRU(128)	1000	7,2	0,16	0,25	0,27
GRU(256),GRU(256)	1000	9,2	0,19	0,23	0,17
GRU(64),GRU(64),GRU(64)	1000	4,6	0,23	0,22	0,34
GRU(128),GRU(128),GRU(128)	1000	11,5	0,15	0,34	0,19
GRU(256),GRU(128),GRU(64)	1000	12,4	0,23	0,31	0,26

Table 5.3: Result of the architecture search for recurrent neural networks. Each model was trained for 5 epochs. The reported values in the columns position, pitch, roll and yaw are the mean absolute error of the test data. Position error is in meters and angle error in degrees. Column *seq-len* is the sequence length or window size and determines how many sensor measurements from the past the network gets as input. With a sequence length of 1000 therefore means that the model gets the last 10 seconds as input. The architecture *GRU(64), GRU(128)* for example describes a model with two hidden GRU layers with 64 and 128 units followed by a dense layer with 6 units at the end.

5.3.2 CRNN

The results of the architecture search with CRNN models is listed in table 5.4. One can conclude the following:

- Pitch, roll and yaw accuracy are basically identically to the RNN models. The mean absolute error is below 0,3 degree in most cases. Again, pitch estimates are slightly more accurate.
- The position accuracy is clearly worse than with RNN models. In the best case the model *CNN(16),CNN(16),GRU(256)* archived a position accuracy of only 8,1 meter.
- In this case the different sequence length had almost no influence on the position accuracy. Going from 500 to a window length of 1000 did not improve the result.

Architecture	**seq-len**	**position**	**pitch**	**roll**	**yaw**
CNN(16),CNN(16),GRU(64)	100	12,3	0,23	0,25	0,38
CNN(32),CNN(32),GRU(64)	100	11,8	0,22	0,25	0,36
CNN(16),CNN(16),GRU(256)	100	10,8	0,15	0,24	0,19
CNN(32),CNN(32),GRU(256)	100	12,3	0,15	0,24	0,22
CNN(16),CNN(16),GRU(64)	100	19,9	0,22	0,25	0,50
CNN(16),CNN(16),GRU(256)	100	12,7	0,16	0,24	0,18
CNN(32),CNN(32),GRU(256)	100	12,4	0,15	0,23	0,19
CNN(16),CNN(16),GRU(64)	500	10,9	0,17	0,26	0,28
CNN(32),CNN(32),GRU(64)	500	12,0	0,17	0,24	0,24
CNN(16),CNN(16),GRU(256)	**500**	**8,1**	**0,08**	**0,22**	**0,09**
CNN(32),CNN(32),GRU(256)	500	10,1	0,16	0,22	0,16

CNN(16),CNN(16),GRU(64)	500	17,9	0,22	0,28	0,37
CNN(16),CNN(16),GRU(256)	500	10,3	0,13	0,23	0,16
CNN(32),CNN(32),GRU(256)	500	10,6	0,12	0,24	0,17
CNN(16),CNN(16),GRU(64)	1000	10,0	0,15	0,25	0,24
CNN(32),CNN(32),GRU(64)	1000	10,2	0,16	0,24	0,24
CNN(16),CNN(16),GRU(256)	1000	9,1	0,12	0,18	0,19
CNN(32),CNN(32),GRU(256)	1000	9,4	0,17	0,27	0,20
CNN(16),CNN(16),GRU(64)	1000	13,9	0,19	0,23	0,21
CNN(16),CNN(16),GRU(256)	1000	9,1	0,14	0,20	0,19
CNN(32),CNN(32),GRU(256)	1000	9,5	0,13	0,19	0,16

Table 5.4: Result of the architecture search for convolutional recurrent neural networks. The reported values in the columns position, pitch, roll and yaw are the mean absolute error over the test data. Position error is in meters and angle error in degrees. Column *seq-len* is the sequence length or window size and determines how many sensor measurements from the past the network gets as input. The architecture *CNN(32),CNN(32),GRU(256)* for example describes a model with two CNN layers which have 32 filters and a kernel size of 5 followed by a GRU cell with 256 units. At the end of each model is again a dense layer with 6 units.

5.3.3 Other Parameters

The above listed models did not use any regularization or dropout. Although the dataset was quite small there was no overfitting visible. There were also several models tested with L1 and L2 regularization or dropout layers with different dropout rates. However, regularization did not improve the result.

Adam optimizer with an initial learning rate of 0.001 worked best (compared to SGD or RMSprop). Again, mean absolute error (MAE) as loss function worked better than mean squared error (MSE) or other related loss functions. Since the dataset is quite small the loss converged after 5 epochs in most cases. Training for more than 5 epochs did not improve the result. Somewhat surprisingly, no overfitting occurred when training for up to 307 epochs.

5.4 Comparison to other Algorithms

To put the approach into comparison other machine learning algorithms have been trained on the same data. The results are listed in table 5.5.
Again the AutoML framework H20 which tests multiple techniques (like decision trees, random forest, deep learning and some more) and optimizes those models automatically was used. In this case the best performing model was just as with simulation data a Gradient Boosting Machine (GBM). The GBM model archived a position accuracy of 2,4 meter.

It is surprising that linear regression and decision trees have a better position accuracy than the deep learning approach. As stated previously linear regression for example is not designed for processing sequences and time-series information. Recurrent neural networks on the contrast are specifically designed for this kind of problems. Therefore it is surprising that the deep learning approach is significantly worse for position prediction. It is quite astonishing that the baseline algorithm LastValue which simply returns the last GPS input as prediction archives a position accuracy of 2,5 meter and is therefore better than the deep learning approach. In other words, if the network would have learned to simple copy and return the last GPS measurement from the inputs the position accuracy would be higher.
However, one should keep in mind that the dataset is very small and it is known that deep learning begins to shine when training on larges amounts of data. As it can be seen in the previous chapter deep learning can archive a position accuracy of 1 meter or better with a sufficient amount of data.

Another point that is clearly visible is that predicting the orientation is way better with the proposed deep learning approach. The other tested algorithms archived a mean absolute error of more than 2 degree for pitch, roll and yaw. Deep learning on the other hand works better, because the mean absolute error is here below 0,2 degree for the orientation.

Method	**Position**	**Pitch**	**Roll**	**Yaw**
AutoML (H2O)	2,4 m	2,5 °	2,4 °	2,9 °
Linear Regression	1,7 m	3,2 °	2,1 °	3,6 °
Decision Tree	1,9 m	3,7 °	3,3 °	5,9 °
LastValue	2,5 m	-	-	-
Deep Learning (proposed approach)	3,6 m	0,08	0,22	0,10

Table 5.5: Position accuracy of other algorithms. The reported values in the columns position, pitch, roll and yaw are the mean absolute error over the test data. In this case the best H2O model was a Gradient Boosting Machine (GBM). The algorithm LastValue acts as a baseline and only returns the last GPS sensor measurement as prediction. To compare the algorithms, the performance of the best model *GRU(256)* from the architecture search above is also listed at the bottom of the table.

5.5 Further Experiments

5.5.1 Inference Time

As described in chapter 1 the trained network should fulfill real-time requirements. Therefore it is necessary that the trained model needs less than 16,6 milliseconds for one forward pass. Table 5.6 shows that this is possible when using a GPU. When computing the result on CPU the inference time increases drastically. Interestingly, there is a difference between the two deep learning frameworks Tensorflow and PyTorch. PyTorch is in the GPU as well in the CPU case much faster. The input sequence length has a big impact on the computation time. If one reduces the window length from 1000 to 500 the computation time almost halves.

	TF GPU	PyTorch GPU	TF CPU	PyTorch CPU
(1000, 19)	16 ms	12 ms	600 ms	100 ms
(500, 19)	10 ms	6 ms	320 ms	70 ms

Table 5.6: Inference time of the trained network for one forward pass with Tensorflow (version 2.0) and PyTorch (version 1.3). The were 2 input arrays tested which contain 500 respectively 1000 measurements (sequence length) and 19 features.

5.5.2 Pretraining on Simulation Data

Transfer learning is a technique where one trains a network on a similar task and then fine-tunes it on the actual task. This has been proven very successful in the computer vision and image classification area. Here one often faces the challenge that deep learning needs a lot of data but the actual dataset is very small and very specific. In such a case it can be useful to train the network first on a large image database even if it doesn't contain the actual classes and then use this network weights as a starting point for training on the actual task. The idea behind this is it that image classification contains some sub-tasks which are identical (or very similar) across all problems. Such a task could be a basic edge and shape detection. This low-level tasks can be learned on a different related dataset. Thus the model is already able to detect basic features and only needs to learn some task specific high level features which is possible with a small amount of data.

The idea behind this experiment is it to apply transfer learning to the pose estimation problem. Since one can generate huge amounts of simulation data this might be helpful to initialize and pretrain a model before training on a small real world dataset. Table 5.7 shows the result of using a network which was pretrained on 15 hours of simulation data before fine tuning on the real dataset.

Epoch	**Position**	**Pitch**	**Roll**	**Yaw**
1	11,1 m	0,29°	0,25°	0,81°
2	8,9 m	0,27°	0,24°	0,42°
3	9,4 m	0,16°	0,28°	0,31°
4	6,9 m	0,15°	0,24°	0,19°
5	6,5 m	0,11°	0,23°	0,16°
without pretraining	3,6 m	0,12°	0,22°	0,15°

Table 5.7: Pretraining results. In this case the network was trained on 15 hours of simulation data. Afterwards the network was fine-tuned on the real data for 5 epochs.

Surprisingly, pretraining on simulation data leads to poorer results. The position accuracy is twice as high (6,5 m compared to 3,5 m). Orientation accuracy is very similar, but did not improve either.
The author's hypothesis is that this is due to the difference between simulation and reality. The simulation data was simulated with a specific noise model while the real

measurement has different noise characteristics. Thus it could be the case that these two datasets are just fundamental different such that no synergy effects arise.

5.5.3 Odometry Impact

So far the results presented in this chapter where based on 9 input features (3 each for GPS, accelerometer and gyroscope). However, as described in section 5.1 this dataset contains also odometry sensor measurements. This includes one value for the wheel angle (of the front axle) and 4 wheel tick values (RPM values from the odometer) which are now also included in this experiment.

The assumption of the author was that this improves the position accuracy. Since GPS measurements only arrive every second the model needs to learn to interpolate between those GPS measurements. For this interpolation the model could use the IMU data. However, as described in the previous chapter the model does not include the IMU inputs for the position estimate and only relies on the GPS inputs. This may be because the IMU data is just to noisy and complex. Measuring odometry data on the other hand is straightforward and contains less noise. It was therefore expected that the wheel angle and wheel ticks (velocity) helps the model interpolating between GPS measurements and lead to a better position accuracy. Unfortunately, as it can be seen in table 5.8 this is not the case.

Epoch	**Position**	**Pitch**	**Roll**	**Yaw**
1	8,0 m	0,10°	0,24°	0,30°
2	6,7 m	0,08°	0,24°	0,15°
3	4,2 m	0,08°	0,23°	0,12°
4	4,1 m	0,06°	0,23°	0,09°
5	3,8 m	0,05°	0,16°	0,08°
without odometry	3,6 m	0,12°	0,22°	0,15°

Table 5.8: Results when odometry features are included during training.

The position accuracy is not affected if odometry inputs are included or not (3,8 m compared to 3,5 m). However, including odometry data improves the orientation accuracy quite a bit. Pitch and yaw are almost twice as good (0,05° and 0,15°). This it not surprising since the wheel angle input is obviously very helpful to determine the orientation.
It is interesting though that this shows that the network can do sensor fusion of multiple

sensors. The improvement of the orientation accuracy shows that the model is able to handle and make use of the odometry data. However, it remains an open question why the model does not include the odometry data for the position prediction.

5.5.4 Downsample Resolution

As described in section 5.2 the 1 Hz GPS signal is upscaled to 100 Hz by using linear interpolation. Another possibility during preprocessing is it to downscale all other inputs to 1 Hz which is done in this experiment. This has the benefit that in a input sequence (window length) of 100 there are now 100 GPS measurements included. So far by using a sequence length of 1000 and upsampling the data to 100 Hz there were only 10 GPS measurements included in the input window. The higher number of GPS measurements in the input data may help the model to do a more precise interpolation and could lead to a better position accuracy. In other words by downsampling the sensor measurements the timespan of the input data is much longer. Therefore the inputs reach back further into the past. It might be the case that looking back longer in time helps the model to predict the future more accurate even though the inputs are at a lower resolution.

However, as it can be seen in table 5.9 quite the contrary is the case. The position accuracy is significant worse and only reaches 13,3 meter. In this case the data was downscaled to 1 Hz and a sequence length of 100 was used. Other variants (downscaling to 1, 2, 10 or 50 Hz) and combinations (sequence lengths from 10 to 1000) were also tested. However, none of the test configurations led to a better results than upscaling the data to 100 Hz and using a sequence length of 1000.

Epoch	**Position**	**Pitch**	**Roll**	**Yaw**
1	25,6 m	0,50°	0,54°	1,31°
2	15,2 m	0,33°	0,53°	0,66°
3	13,8 m	0,24°	0,53°	0,50°
4	13,5 m	0,30°	0,52°	0,46°
5	13,3 m	0,20°	0,52°	0,28°
without odometry	3,6 m	0,12°	0,22°	0,15°

Table 5.9: Results of downsampling all sensor measurements to 1 Hz with an sequence length (window length) of 100.

CHAPTER 6

Conclusions

In the first part of this thesis simulation data was used to find a suitable network architecture. The dataset includes 9 input features coming from GPS, accelerometer and gyroscope. All sensor measurements are simulated with 100 Hz in order to simplify the problem. The architecture search reveals that recurrent neural networks (RNN) work better than a combination of convolutional and recurrent neural networks (CRNN). The best performing model contains one gated recurrent unit (GRU) layer with 256 units. The model archived a mean position accuracy of 3,5 meter and a mean orientation error of 1,5 $^\circ$ when training on 2 hours of data. By increasing the dataset size up to 20 hours the mean average error can be improved to 0,95 meter and 0,3 $^\circ$.
Experiments have shown that the network is also able to process sensor signals with different update rates. When the GPS signal was simulated with 1 Hz the performance did not decline significantly.
However, some problems have been identified. Firstly, during training gradients and weights changes are very small which indicates that the learning is not optimal. All attempts to fix this vanishing gradient problem were not successful. Secondly, the network basically only uses the GPS measurements for the position prediction and does not include IMU data in order to improve the result. It seems that the model is not able to fuse the IMU data with the GPS measurements at least for the position prediction. For the orientation estimation the network is able to do sensor fusion, because the performance gets worse if one of the three sensors is removed.

In the second part of this thesis a real dataset was used for training which includes 1,5 hours of driving time. This dataset contains 19 input features. Again an architecture search was done which confirms the results from the simulation data meaning that pure recurrent neural networks (RNN) with one GRU layer work best. In this case the model archived a mean position accuracy of 3,6 meter. Pitch, roll and yaw values were at 0,12 $^\circ$, 0,22 $^\circ$ and 0,15 $^\circ$.
Besides GPS and IMU in this case also odometry measurements (wheel velocity and wheel angle) are available. By including odometry inputs the orientation accuracy can be improved to 0,05 $^\circ$, 0,16 $^\circ$ and 0,08 $^\circ$. Again, this shows that the network is able to do sensor fusion for estimating the orientation. Why the model does not include IMU and odometry measurements for predicting the position remains an open question.

CHAPTER 7

Outlook

The most obvious and promising way to improve the performance is it to increase the dataset size. It is known that deep learning is very data hungry and begins to shine when training on large datasets. Experiments with the simulation data has shown that the position accuracy can be drastically improved by using 20 hours of training data. With a larger dataset it may be the case that the model can learn to handle the complex IMU data and fuse them together with GPS measurements in order to predict a more accurate position. However, how much the performance can be improved with huge amounts of data is not predictable. Since increasing the dataset size from 15 hours to 20 hours significantly improves the position accuracy from 2 meter to 0,95 meter it is likely that the performance does indeed improve further if even more data is used.

Another way to improve the results is it to increase the quality of the input data. Besides of using higher quality sensors this could be done by pre-filtering the raw data before passing it to the neural network in order to reduce noise. Classical and rather simple signal processing filtering methods did not improve the result, but it may be the case that more advanced techniques like the Kalman filter could indeed improve the performance.

One major drawback of the Kalman filter is it that one has to specify a error model for each sensor. Obviously the quality of this error models influences the prediction accuracy. However, the noise of IMU sensors is very complicated, non-linear, correlated over time and can differ from sensor to sensor. Therefore providing an accurate noise model is not as straightforward as it may sound. One possible way to overcome this issue is the usage of a so called *Deep Kalman filter* [Hos18]. Here one does not have to specify any error model since the sensor error is learned and estimated by a neural network.

Bibliography

[AKR06] W. T. Ang, P. K. Khosla, and C. N. Riviere. “Nonlinear regression model of a low-*g* MEMS accelerometer.” In: *IEEE Sensors Journal* 7.1 (2006), pp. 81–88.

[Ang+13] D. Anguita, A. Ghio, L. Oneto, X. Parra, and J. L. Reyes-Ortiz. “A public domain dataset for human activity recognition using smartphones.” In: *Esann*. 2013.

[Arr+19] B. Arrieta et al. “Explainable Artificial Intelligence (XAI): Concepts, Taxonomies, Opportunities and Challenges toward Responsible AI.” In: *arXiv preprint arXiv:1910.10045* (2019).

[BDW95] B. Barshan and H. F. Durrant-Whyte. “Inertial navigation systems for mobile robots.” In: *IEEE Transactions on Robotics and Automation* 11.3 (1995), pp. 328–342.

[Bel+15] V. Belagiannis, C. Rupprecht, G. Carneiro, and N. Navab. “Robust optimization for deep regression.” In: *Proceedings of the IEEE international conference on computer vision*. 2015, pp. 2830–2838.

[BF10] M. G. Blum and O. François. “Non-linear regression models for Approximate Bayesian Computation.” In: *Statistics and Computing* 20.1 (2010), pp. 63–73.

[BL11] J. B. Bancroft and G. Lachapelle. “Data fusion algorithms for multiple inertial measurement units.” In: *Sensors* 11.7 (2011), pp. 6771–6798.

[Bri+17] D. Britz, A. Goldie, M.-T. Luong, and Q. Le. “Massive exploration of neural machine translation architectures.” In: *arXiv preprint arXiv:1703.03906* (2017).

[CESN04] K.-W. Chiang, N El-Sheimy, and A Noureldin. "A new weights updating method for neural networks based INS/GPS integration architectures." In: *measurement science and technology* 15.10 (2004), pp. 2053–2061.

[Cho+14] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. "Learning phrase representations using RNN encoder-decoder for statistical machine translation." In: *arXiv preprint arXiv:1406.1078* (2014).

[Chu+14] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio. "Empirical evaluation of gated recurrent neural networks on sequence modeling." In: *arXiv preprint arXiv:1412.3555* (2014).

[Cos+17] H. Coskun, F. Achilles, R. DiPietro, N. Navab, and F. Tombari. "Long short-term memory kalman filters: Recurrent neural estimators for pose regularization." In: *Proceedings of the IEEE International Conference on Computer Vision*. 2017, pp. 5524–5532.

[Dah+11] G. E. Dahl, D. Yu, L. Deng, and A. Acero. "Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition." In: *IEEE Transactions on audio, speech, and language processing* 20.1 (2011), pp. 30–42.

[Elm02] W. Elmenreich. "An introduction to sensor fusion." In: *Austria: Vienna University Of Technology* February (2002), pp. 1–28. ISSN: 0952-813X.

[Gam17] J. C. B. Gamboa. "Deep Learning for Time-Series Analysis." In: (2017). arXiv: `1701.01887`.

[GB10] X. Glorot and Y. Bengio. "Understanding the difficulty of training deep feedforward neural networks." In: *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. 2010, pp. 249–256.

[GBC16] I. Goodfellow, Y. Bengio, and A. Courville. *Deep learning*. MIT press, 2016.

[Geh+17] J. Gehring, M. Auli, D. Grangier, D. Yarats, and Y. N. Dauphin. "Convolutional sequence to sequence learning." In: *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org. 2017, pp. 1243–1252.

[GSC99] F. A. Gers, J. Schmidhuber, and F. Cummins. "Learning to forget: Continual prediction with LSTM." In: (1999).

[GWA07] M. S. Grewal, L. R. Weill, and A. P. Andrews. *Global positioning systems, inertial navigation, and integration*. John Wiley & Sons, 2007.

[HC16] S. Ha and S. Choi. "Convolutional neural networks for human activity recognition using multiple accelerometer and gyroscope sensors." In: *Proceedings of the International Joint Conference on Neural Networks*. Vol. 2016-Octob. 2016, pp. 381–388. ISBN: 9781509006199. DOI: 10.1109/IJCNN.2016.7727224.

[He+16] 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*. 2016, pp. 770–778.

[HHC17] J. Huang, Z. Huang, and K. Chen. "Combining low-cost Inertial Measurement Unit (IMU) and deep learning algorithm for predicting vehicle attitude." In: *2017 IEEE Conference on Dependable and Secure Computing*. IEEE. 2017, pp. 237–239.

[HHP] N. Y. Hammerla, S. Halloran, and T. Plötz. *Deep, Convolutional, and Recurrent Models for Human Activity Recognition Using Wearables*. Tech. rep.

[Hos18] S. Hosseinyalamdary. "Deep Kalman filter: Simultaneous multi-sensor integration and modelling; A GNSS/IMU case study." In: *Sensors (Switzerland)* 18.5 (2018). ISSN: 14248220. DOI: 10.3390/s18051316.

[HR05] E. L. Haseltine and J. B. Rawlings. "Critical evaluation of extended Kalman filtering and moving-horizon estimation." In: *Industrial & engineering chemistry research* 44.8 (2005), pp. 2451–2460.

[HS97a] S. Hochreiter and J. Schmidhuber. "Long short-term memory." In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[HS97b] S. Hochreiter and J. Schmidhuber. "LSTM can solve hard long time lag problems." In: *Advances in neural information processing systems*. 1997, pp. 473–479.

[HSW89] K. Hornik, M. Stinchcombe, and H. White. "Multilayer feedforward networks are universal approximators." In: *Neural networks* 2.5 (1989), pp. 359–366.

[Hua+18] Y. Huang, M. Kaufmann, E. Aksan, M. J. Black, O. Hilliges, and G. Pons-Moll. "Deep inertial poser." In: *ACM Transactions on Graphics* 37.6 (2018), pp. 1–15. ISSN: 07300301. DOI: 10.1145/3272127.3275108.

[HYC16] S. Ha, J. M. Yun, and S. Choi. "Multi-modal Convolutional Neural Networks for Activity Recognition." In: *Proceedings - 2015 IEEE International Conference on Systems, Man, and Cybernetics, SMC 2015* (2016), pp. 3017–3022. DOI: 10.1109/SMC.2015.525.

[IS15] S. Ioffe and C. Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift." In: *arXiv preprint arXiv:1502.03167* (2015).

[Ism+19] H. Ismail Fawaz, G. Forestier, J. Weber, L. Idoumghar, and P. A. Muller. "Deep learning for time series classification: a review." In: *Data Mining and Knowledge Discovery* 33.4 (2019), pp. 917–963. ISSN: 1573756X. DOI: 10.1007/s10618-019-00619-1. arXiv: 1809.04356.

[Jaf+19] A. Jafari, A. Ganesan, C. S. K. Thalisetty, V. Sivasubramanian, T. Oates, and T. Mohsenin. "SensorNet: A Scalable and Low-Power Deep Convolutional Neural Network for Multimodal Data Classification." In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 66.1 (2019), pp. 274–287. ISSN: 15498328. DOI: 10.1109/TCSI.2018.2848647.

[JY15] W. Jiang and Z. Yin. "Human activity recognition using wearable sensors by deep convolutional neural networks." In: *Proceedings of the 23rd ACM international conference on Multimedia*. Acm. 2015, pp. 1307–1310.

[Kal60] R. E. Kalman. "A new approach to linear filtering and prediction problems." In: (1960).

[KB14] D. P. Kingma and J. Ba. "Adam: A method for stochastic optimization." In: *arXiv preprint arXiv:1412.6980* (2014).

[KDD17] K. Kyritsis, C. Diou, and A. Delopoulos. "Food Intake Detection from Inertial Sensors Using LSTM Networks." In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 10590 LNCS (2017), pp. 411–418. ISSN: 16113349. DOI: 10.1007/978-3-319-70742-6_39.

[Keh+16] W. Kehl, F. Milletari, F. Tombari, S. Ilic, and N. Navab. "Deep learning of local RGB-D patches for 3D object detection and 6D pose estimation." In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 9907 LNCS (2016), pp. 205–220. ISSN: 16113349. DOI: 10.1007/978-3-319-46487-9_13. arXiv: 1607.06038.

[KGC15] 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*. Vol. 2015 Inter. 2015, pp. 2938–2946. ISBN: 9781467383912. DOI: 10.1109/ICCV.2015.336.

[KHS17] M. Kok, J. D. Hol, and T. B. Schön. *Using inertial sensors for position and orientation estimation*. 2017. DOI: 10.1561/2000000094.

[Kis+19] M. Kissai, B. Monsuez, X. Mouton, D. Martinez, and A. Tapus. "Adaptive Robust Vehicle Motion Control for Future Over-Actuated Vehicles." In: *Machines* 7.2 (2019), p. 26.

[KSH12] A. Krizhevsky, I. Sutskever, and G. E. Hinton. "Imagenet classification with deep convolutional neural networks." In: *Advances in neural information processing systems*. 2012, pp. 1097–1105.

[KSS15] R. G. Krishnan, U. Shalit, and D. Sontag. *Deep Kalman Filters*. 2015. arXiv: `1511.05121 [stat.ML]`.

[Lan+10] N. D. Lane, E. Miluzzo, H. Lu, D. Peebles, T. Choudhury, and A. T. Campbell. "A survey of mobile phone sensing." In: *IEEE Communications magazine* 48.9 (2010), pp. 140–150.

[Lat+19] S. Lathuilière, P. Mesejo, X. Alameda-Pineda, and R. Horaud. "A comprehensive analysis of deep regression." In: *IEEE transactions on pattern analysis and machine intelligence* (2019).

[LeC+98] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. "Gradient-based learning applied to document recognition." In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[Lim+19] W. S. Lima, E. Souto, K. El-Khatib, R. Jalali, and J. Gama. "Human activity recognition using inertial sensors in a smartphone: An overview." In: *Sensors (Switzerland)* 19.14 (2019), pp. 14–16. ISSN: 14248220. DOI: `10.3390/s19143213`.

[LKL14] M. Längkvist, L. Karlsson, and A. Loutfi. "A review of unsupervised feature learning and deep learning for time-series modeling." In: *Pattern Recognition Letters* 42 (2014), pp. 11–24.

[LL17] S. M. Lundberg and S.-I. Lee. "A Unified Approach to Interpreting Model Predictions." In: *Advances in Neural Information Processing Systems 30*. Ed. by I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett. Curran Associates, Inc., 2017, pp. 4765–4774.

[MP17a] M. Minsky and S. A. Papert. *Perceptrons: An introduction to computational geometry*. MIT press, 2017.

[MP17b] A. Murad and J. Y. Pyun. "Deep recurrent neural networks for human activity recognition." In: *Sensors (Switzerland)* 17.11 (2017). ISSN: 14248220. DOI: `10.3390/s17112556`.

[MWJ12] R. van der Merwe, E. Wan, and S. Julier. "Sigma-Point Kalman Filters for Nonlinear Estimation and Sensor-Fusion: Applications to Integrated Navigation." In: August 2004 (2012). DOI: `10.2514/6.2004-5120`.

[Ng04] A. Y. Ng. "Feature selection, L 1 vs. L 2 regularization, and rotational invariance." In: *Proceedings of the twenty-first international conference on Machine learning*. ACM. 2004, p. 78.

[Nwe+18] H. F. Nweke, Y. W. Teh, M. A. Al-garadi, and U. R. Alo. *Deep learning algorithms for human activity recognition using mobile and wearable sensor networks: State of the art and research challenges*. 2018. DOI: `10.1016/j.eswa.2018.03.056`.

[Ö19] Özgür Akyazı. *IMU Sensor Fusion With Machine Learning*. Tech. rep. 2019.

[Ola15] C. Olah. "Understanding lstm networks, 2015." In: *URL http://colah.github.io/posts/2015-08-Understanding-LSTMs* (2015).

[OR16] F. J. Ordóñez and D. Roggen. "Deep convolutional and LSTM recurrent neural networks for multimodal wearable activity recognition." In: *Sensors (Switzerland)* 16.1 (2016). ISSN: 14248220. DOI: `10.3390/s16010115`.

[Par04] M. Park. "Error analysis and stochastic modeling of MEMS based inertial sensors for land vehicle navigation applications." In: (2004).

[PE96] B. W. Parkinson and P. K. Enge. "Differential gps." In: *Global Positioning System: Theory and applications.* 2 (1996), pp. 3–50.

[Pen+19] S. Peng, Y. Liu, Q. Huang, X. Zhou, and H. Bao. "PVNet: Pixel-wise Voting Network for 6DoF Pose Estimation." In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2019, pp. 4561–4570.

[Pre98] L. Prechelt. "Automatic early stopping using cross validation: quantifying the criteria." In: *Neural Networks* 11.4 (1998), pp. 761–767.

[Qin+18] Z. Qin, F. Yu, C. Liu, and X. Chen. "How convolutional neural network see the world-A survey of convolutional neural network visualization methods." In: *arXiv preprint arXiv:1804.11191* (2018).

[Qin+19] T. Qin, S. Cao, J. Pan, and S. Shen. "A General Optimization-based Framework for Global Pose Estimation with Multiple Sensors." In: (2019). arXiv: `1901.03642`.

[RAHS04] I. Rhee, M. F. Abdel-Hafez, and J. L. Speyer. "Observability of an integrated GPS/INS during maneuvers." In: *IEEE Transactions on Aerospace and Electronic systems* 40.2 (2004), pp. 526–535.

[Ram+16] J. R. Rambach, A. Tewari, A. Pagani, and D. Stricker. "Learning to fuse: A deep learning approach to visual-inertial camera pose estimation." In: *2016 IEEE International Symposium on Mixed and Augmented Reality (ISMAR)*. IEEE. 2016, pp. 71–76.

[Rao+14a] Q. Rao, C. Grünler, M. Hammori, and S. Chakrabort. "Design methods for augmented reality in-vehicle infotainment systems." In: *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE. 2014, pp. 1–6.

[Rao+14b] Q. Rao, T. Tropper, C. Grünler, M. Hammori, and S. Chakraborty. "AR-IVI—implementation of in-vehicle augmented reality." In: *2014 IEEE international symposium on mixed and augmented reality (ISMAR)*. IEEE. 2014, pp. 3–8.

[REG14] S.-u. REGELUNGSTECHNIK. "Sensor Fusion Algorithms for Robotics: Bayesian Inference vs. Cortical Circuits." In: (2014).

[RHW86] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. "Learning representations by back-propagating errors." In: *nature* 323.6088 (1986), pp. 533–536.

[Rib04] M. I. Ribeiro. "Kalman and extended kalman filters: Concept, derivation and properties." In: *Institute for Systems and Robotics* 43 (2004).

[Roa08] T.-h. Road. "Constructive MEMS / GPS Integration Scheme." In: *IEEE Transactions on Aerospace and Electronic Systems* 44.2 (2008), pp. 582–594.

[Ros61] F. Rosenblatt. *Principles of neurodynamics. perceptrons and the theory of brain mechanisms*. Tech. rep. Cornell Aeronautical Lab Inc Buffalo NY, 1961.

[RSG16] M. T. Ribeiro, S. Singh, and C. Guestrin. ""Why Should I Trust You?": Explaining the Predictions of Any Classifier." In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*. 2016, pp. 1135–1144.

[Rud16] S. Ruder. "An overview of gradient descent optimization algorithms." In: *arXiv preprint arXiv:1609.04747* (2016).

[Rue+18] F. M. Rueda, R. Grzeszick, G. A. Fink, S. Feldhorst, and M. Ten Hompel. "Convolutional neural networks for human activity recognition using body-worn sensors." In: *Informatics* 5.2 (2018), pp. 1–17. ISSN: 22279709. DOI: `10.3390/informatics5020026`.

[Sab06] A. M. Sabatini. "Quaternion-based extended Kalman filter for determining orientation by inertial and magnetic sensing." In: *IEEE Transactions on Biomedical Engineering* 53.7 (2006), pp. 1346–1356.

[Sch15] J. Schmidhuber. "Deep learning in neural networks: An overview." In: *Neural networks* 61 (2015), pp. 85–117.

[Sha17] Shaun M. Howard. "Deep Learning for sensor fusion." MA thesis. Case Western Reserve University, 2017.

[Sin+17a] M. S. Singh, V. Pondenkandath, B. Zhou, P. Lukowicz, and M. Liwickit. "Transforming sensor data to the image domain for deep learning—An application to footstep detection." In: *2017 International Joint Conference on Neural Networks (IJCNN)*. IEEE. 2017, pp. 2665–2672.

[Sin+17b] S. P. Singh, A. Kumar, H. Darbari, L. Singh, A. Rastogi, and S. Jain. "Machine translation using deep learning: An overview." In: *2017 International Conference on Computer, Communications and Electronics (Comptelix)*. IEEE. 2017, pp. 162–167.

[SMB10] D. Scherer, A. Müller, and S. Behnke. "Evaluation of pooling operations in convolutional architectures for object recognition." In: *International conference on artificial neural networks*. Springer. 2010, pp. 92–101.

[SPG19] S. H. Sánchez, R. F. Pozo, and L. A. H. Gómez. "Deep Neural Networks for Driver Identification Using Accelerometer Signals from Smartphones." In: *International Conference on Business Information Systems*. Springer. 2019, pp. 206–220.

[Sri13] N. Srivastava. "Improving neural networks with dropout." In: *University of Toronto* 182 (2013).

[Sri+14] N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. "Dropout: a simple way to prevent neural networks from overfitting." In: *Journal of Machine Learning Research* 15.1 (2014), pp. 1929–1958.

[TH12] T. Tieleman and G. Hinton. "Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude." In: *COURSERA: Neural networks for machine learning* 4.2 (2012), pp. 26–31.

[TO17] L. Tran and S. L. Obispo. *Data Fusion with 9 Degrees of Freedom Inertial Measurement Unit To Determine Object's Orientation*. Tech. rep. 2017.

[TS14] A. Toshev and C. Szegedy. "Deeppose: Human pose estimation via deep neural networks." In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2014, pp. 1653–1660.

[TSF18] B. Tekin, S. N. Sinha, and P. Fua. "Real-time seamless single shot 6d object pose prediction." In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2018, pp. 292–301.

[TYH13] A. Tamjidi, C. Ye, and S. Hong. "6-DOF pose estimation of a portable navigation aid for the visually impaired." In: *2013 IEEE International Symposium on Robotic and Sensors Environments (ROSE)*. IEEE. 2013, pp. 178–183.

[VO99] P. Vanicek and M. Omerbasic. “Does a navigation algorithm have to use a Kalman filter?” In: *Canadian aeronautics and space journal* 45.3 (1999), pp. 292–296.

[Wan+13] L. Wan, M. Zeiler, S. Zhang, Y. L. Cun, and R. Fergus. “Regularization of neural networks using dropconnect.” In: *Proceedings of the 30th international conference on machine learning (ICML-13)*. 2013, pp. 1058–1066.

[Wan+17] J. Wang, Y. Chen, S. Hao, X. Peng, and L. Hu. “Deep Learning for Sensor-based Activity Recognition: A Survey.” In: (2017). DOI: `10.1016/j.patrec.2018.02.010`. arXiv: `1707.03502`.

[Wan+19] J. Wang, Y. Chen, S. Hao, X. Peng, and L. Hu. “Deep learning for sensor-based activity recognition: A survey.” In: *Pattern Recognition Letters* 119 (2019), pp. 3–11. ISSN: 01678655. DOI: `10.1016/j.patrec.2018.02.010`.

[Woo07] O. J. Woodman. *An introduction to inertial navigation*. Tech. rep. University of Cambridge, Computer Laboratory, 2007.

[Wu+02] H. Wu, M. Siegel, R. Stiefelhagen, and J. Yang. “Sensor fusion using Dempster-Shafer theory [for context-aware HCI].” In: *IMTC/2002. Proceedings of the 19th IEEE Instrumentation and Measurement Technology Conference (IEEE Cat. No. 00CH37276)*. Vol. 1. IEEE. 2002, pp. 7–12.

[Wu+16] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, et al. “Google’s neural machine translation system: Bridging the gap between human and machine translation.” In: *arXiv preprint arXiv:1609.08144* (2016).

[Yan+15] J. Yang, M. N. Nguyen, P. P. San, X. L. Li, and S. Krishnaswamy. “Deep convolutional neural networks on multichannel time series for human activity recognition.” In: *Twenty-Fourth International Joint Conference on Artificial Intelligence*. 2015.

[Yao+16] S. Yao, S. Hu, Y. Zhao, A. Zhang, and T. Abdelzaher. “DeepSense: A Unified Deep Learning Framework for Time-Series Mobile Sensing Data Processing.” In: (2016). arXiv: `1611.01942`.

[Yos+15] J. Yosinski, J. Clune, A. Nguyen, T. Fuchs, and H. Lipson. “Understanding neural networks through deep visualization.” In: *arXiv preprint arXiv:1506.06579* (2015).

[YZL12] Y. Yang, J. Zhou, and O. Loffeld. “Quaternion-based Kalman Filtering on INS/GPS.” In: *Proc. IEEE Int. Conf. Inform. Fusion* (2012), pp. 511–518.

[Zeb+18] T. Zebin, M. Sperrin, N. Peek, and A. J. Casson. “Human activity recognition from inertial sensor time-series using batch normalized deep LSTM recurrent networks.” In: *Proceedings of the Annual International Conference of the IEEE Engineering in Medicine and Biology Society, EMBS*. Vol. 2018-July. Institute of Electrical and Electronics Engineers Inc., 2018, pp. 1–4. ISBN: 9781538636466. DOI: 10.1109/EMBC.2018.8513115.

[Zha+16] C. Zhang, S. Bengio, M. Hardt, B. Recht, and O. Vinyals. “Understanding deep learning requires rethinking generalization.” In: *arXiv preprint arXiv:1611.03530* (2016).

[Zhe+14] Y. Zheng, Q. Liu, E. Chen, Y. Ge, and J. L. Zhao. “Time series classification using multi-channels deep convolutional neural networks.” In: *International Conference on Web-Age Information Management*. Springer. 2014, pp. 298–310.

[ZSV14] W. Zaremba, I. Sutskever, and O. Vinyals. “Recurrent neural network regularization.” In: *arXiv preprint arXiv:1409.2329* (2014).