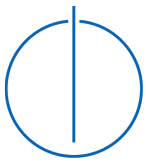
DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Application Project in
M. Sc. Data Engineering and Analytics

# IMU Sensor Fusion With Machine Learning

**Özgür Akyazı**



DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Application Project in
M. Sc. Data Engineering and Analytics

# IMU Sensor Fusion With Machine Learning

| | |
|---|---|
| Author: | Özgür Akyazı |
| Supervisor: | Dipl.-Inf. Univ. Adnane Jadid |
| Advisor: | Prof. Gudrun Klinker, Ph.D. |
| Submission Date: | 09.04.2019 |

I confirm that this application project in
m. sc. data engineering and analytics is my own work and I have documented all sources and material used.

Munich, 09.04.2019 Özgür Akyazı

# Contents

**Abstract**

Tracking is an important task in Augmented Reality and, variety of sensory data is used to achieve this. To extract more information about the scene, object or situation, these sensory data should be understood better in a sense, which is called Sensor Fusion. This field is well established since it has been studied for decades. Except one study, to the writer's knowledge, all studies use an analytical approach for solution. In this study, a fully automated deep learning architecture to fuse multiple-IMU data(acceleration and angular velocity) to get position and orientation is designed and evaluated.

# 1 Introduction

## 1.1 Motivation

Tracking is one of the key tasks in Augmented Reality(AR) and it means to determine the pose of an object, i.e. its position and orientation with respect to some coordinate system in real time. It is a major challenge for AR applications and has to be as precise, accurate and robust as possible in order to create the illusion that the virtual content is a part of the real world. An accurate tracking system is required for AR system because even a small tracking error may cause a noticeable misalignment between virtual and real objects.

There are different approaches to tracking, and hybrid tracking is one of them. Hybrid tracking techniques combine various sensor data into a merged data stream in order to enhance the quality of tracking data using a sensor fusion. Sensor fusion is combination of sensory data or data derived from sensory data such that the resulting information is in some sense better than the case where these sources were used individually. While it provides with better data, the fusion of multiple sensors increases the complexity of the tracking process because of complex manipulation or processing of data.

Tracking can be provided by a variety of sensors such as mechanical, optical and acoustic. Inertial measurement unit (IMU) is one of the mechanical sensors. IMU is an electronic device used for detection of the current object orientation. Usually it measures the acceleration and, angular velocity. Based on inertial principles, acceleration and angular velocity are measured always relative to inertial space. Such sensors mainly consist of at least two different types of sub-sensors, an accelerometer measuring linear acceleration and, a gyroscope measuring orientation and angular velocity.

## 1.2 Goals

In this project, we will be developing a sensor fusion Deep Learning model which fuses multiple IMUs using Deep Learning techniques. The fusion of 3 IMU data (which consists of acceleration and angular velocity) will be

experimented, i.e. whether it can describe the motion of the body mass or not. The main goal of this project is to obtain changes in position and orientation data using multiple IMUs, reliably. It is very well known that acceleration values read from an IMU is highly unstable and some misalignment between axes could exist. As the quality of them decrease, one needs to be more careful when using the information from those. By using Deep Learning, we aim to get a stable and reliable transformation in position and orientation, without any calibration, sensor registration and, error correction or modeling.

### 1.3 Structure of the document

In Section 2, basic topics about Machine Learning and Deep Learning will be introduced. Section 3 explains our project setup and the problem definition. Section 4 contains the previous work that has been done in this field and 6 contains our implementation and model details. Section 7 shows briefly shows the results from the experiments and Section 8 is conclusion.

## 2 Basic Knowledge

### 2.1 Sensor Fusion

Sensor Fusion is the combining of sensory data or data derived from sensory data such that the resulting information is in some sense better than would be possible when these sources were used individually, [1]. It is an important field where it has a lot of use cases in everyday life applications like smartwatches, phones and, Virtual Reality headsets. It has been studied for a long time, and there are very well known analytical solutions to it. Kalman Filter, Extended Kalman Filter, Bayesian Inference, Dempster-Shafer algorithm, Moving Horizon Estimation [9] are the most important ones of them.

### 2.2 Machine Learning

Machine learning is a paradigm that may refer to learning from past experience (which in this case is previous data) to improve future performance, and the sole focus of this field is automatic learning methods, [3]. Ultimate goal of this area is to make accurate predictions after the learning phase, so that various decision would be made by algorithms/models.

#### 2.2.1 Artificial Neural Networks

While Machine learning consists of mostly task-specific methods, its sub-field Artificial Neural Networks(ANN) has ability to work on data representations/ features by extracting characteristics of the input. ANN was, in a sense, inspired from biological neurons. The architecture of ANN usually consists of

grouped units whose outputs are connected to the ones in other, which carries information/data from one to another and, this groups are called as layers. As illustrated in Figure 1, the first(left most) layer is input layer, the last(right most) layer is output layer and, the layers other than those are called as hidden layers. This simple layer type is called as Dense or Fully Connected layers. Each unit at each layer is multiplied with corresponding weights and, all the input values for a unit in the next layer are summed. Then the resulting value is put in a nonlinear function, i.e. tanh, ReLU, sigmoid, which are called as activation functions and the activation function gives the network the ability to learn nonlinear features, otherwise the output of whole network would have been formulated by a matrix multiplication, which is a linear operation. The output value of activation function is the output of the unit and the whole process is repeated for the next layers until the output layer.

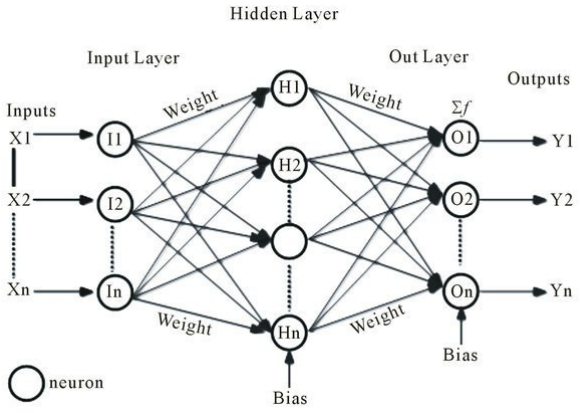



Figure 1: Artificial Neural Network illustration. Image from: [4]

### 2.2.2 Deep Learning

As the prices of the processor devices fall and capabilities of the GPUs increase, the number of hidden layers are increased a lot, and the resulting networks are called as Deep Neural Network(DNN), see Figure 2. As the experiments confirmed, deep architectures yield a better performance than the shallower ones, until a point. Also, supervised, semi-supervise and unsupervised learning is possible in Deep Learning.

- Supervised learning is learning by knowing both input, and its corresponding output. That is, all the input data has a known label.
- Unsupervised learning is learning from only the input data without any corresponding output, so the data set is not labeled.

- Semi-supervised learning is learning with partially labeled data, which could be thought of a mixture of supervised and unsupervised data.

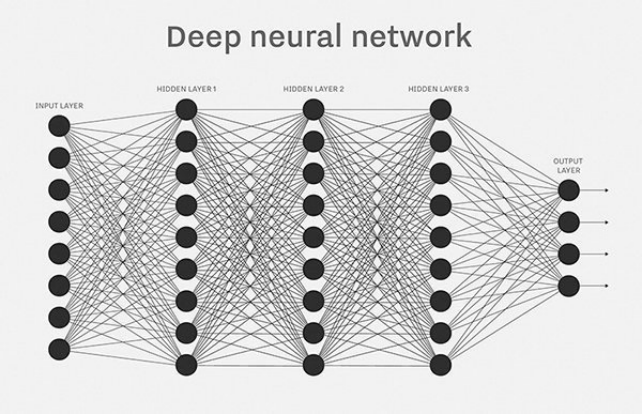



Figure 2: Deep Neural Network illustration. Image from: [5]

#### 2.2.2.1 Convolutional Neural Networks

A convolutional neural networks are basically DNNs and they are used mostly to recognize/extract features from the input data. A general structure of them could be seen in Figure 3.a. The important idea behind it is the convolution operation with kernel, which is applied by looping over the input data and multiplying the input with corresponding kernel values, which is illustrated in Figure 3.b. While it is mostly used on images, it has been used on other data types, like IMUs. By applying a number of convolutions in each layer and concatenating multiple layers, more abstract and high level features are extracted from the input, see Figure 4, which is then easier to be classified or processed by DNN.

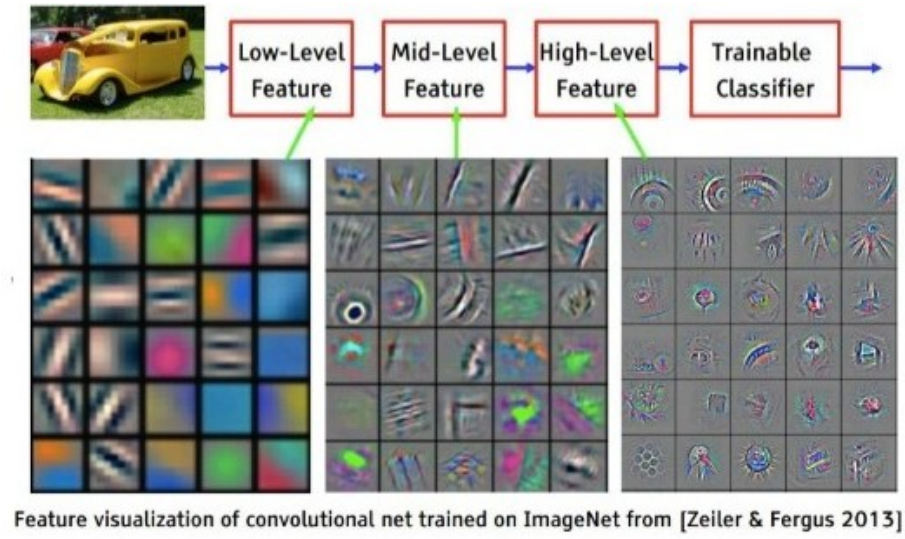
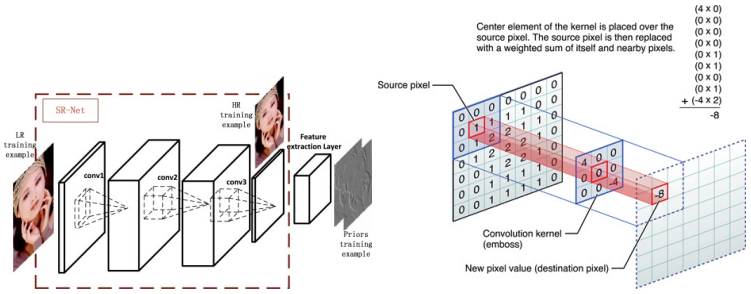



Figure 4: Convolutional Neural Network feature extraction illustration




(a) Convolutional Neural Network illustration. Source: [6]

(b) Convolution with a Kernel example. Source: [7]

Figure 3

#### 2.2.2.2 Recurrent Neural Networks

As the standard artificial neural networks only learn from the currently presented data, there is a need for the notion of time or sequence to learn, and it is met by the Recurrent Neural Networks(RNN). Basically, this structure provides the neural network with a memory from the previous inputs. There are different cell types of RNNs, some of which are:

- Basic Recurrent Unit
- Long Short-Term Memory

- Gated Recurrent Unit

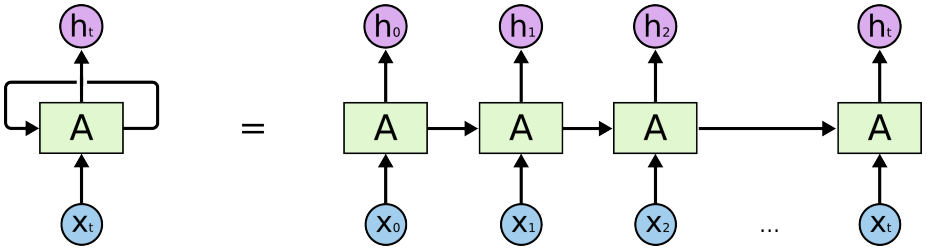



Figure 5: Unrolled RNN unit . Source: [8]

**2.2.2.2.1 Basic Recurrent Unit** A simple recursive unit could be seen in Figure 5. Since the state of the unit in time step $t$ is an input to the unit itself in time step $t+1$, the unit could be illustrated by unrolling it, on the right. However, later it is found that that simple structure is not able to extract long term dependencies(in terms of time or sequence).

**2.2.2.2.2 Long Short-Term Memory** Long Short-Term Memory (LSTM), is the solution for extracting long term dependencies in the data, [16] [17]. As seen in Figure 6, one LSTM unit could be unrolled and it has a complex gating structure, input, forget and output gates, they constitute the cell state, and it can store information for a long period. LSTMs contain memory blocks, not neurons as in ANNs, but they have the internal layer structure. These gates are all connected to the weights, as in the ANN units, they are learned during training. The basic use of them is to decide on how to learn from the new data, how (much) to forget from the already learned information and how to calculate the output, respectively.

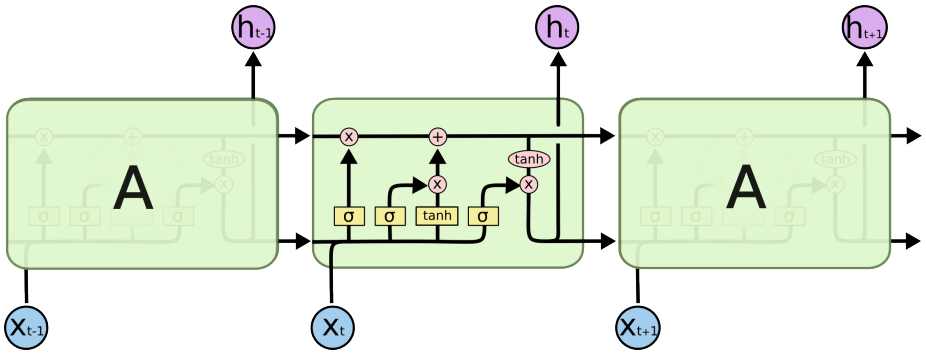



Figure 6: Unrolled LSTM. Source: [8]

# 3 Project Setup and Problem Definition

In our experiments, there will be a cube on which 3 IMUs and one visual target are attached to, where the visual target is tracked by visual tracker [23]. The constructed cube could be seen in Figure 7.

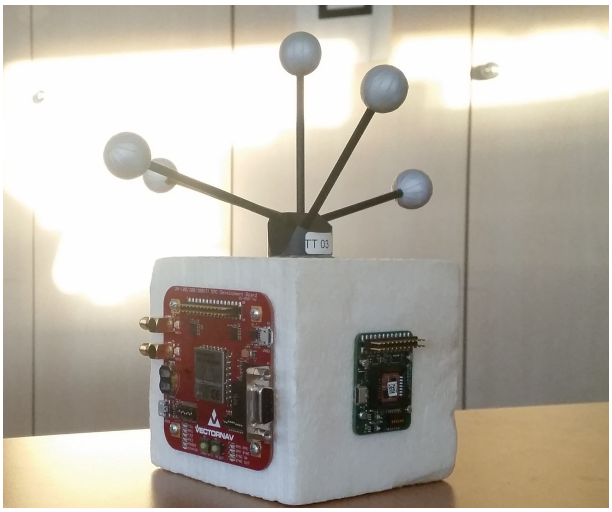


Figure 7: The cube used in the experiments. It has 3 IMUs attached onto it, and a visual target.

Acceleration data from one of the IMUs is a vector, which will be referred as $\boldsymbol{a}$, and angular velocity is also a vector, which will be referred as $\boldsymbol{\omega}$ and they could be defined as follows:

$$\boldsymbol{a} = \begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix} \quad \boldsymbol{\omega} = \begin{bmatrix} \omega_x \\ \omega_y \\ \omega_z \end{bmatrix} \tag{1}$$

where subscripts show the axis of the respective measurement. Also, data acquired from one IMU is $\boldsymbol{M}$, and defined as:

$$\boldsymbol{M} = \begin{bmatrix} \boldsymbol{a} \\ \boldsymbol{\omega} \end{bmatrix} \tag{2}$$

Since there will be multiple IMUs to be fused, $\boldsymbol{M}_1, \boldsymbol{M}_2, ..., \boldsymbol{M}_k$ will be referring to the respective sensor device data. The pose vector could be defined as $\boldsymbol{P}$ :

$$\boldsymbol{P} = \begin{bmatrix} U \\ W \end{bmatrix} \quad \boldsymbol{U} = \begin{bmatrix} Pos_x \\ Pos_y \\ Pos_z \end{bmatrix} \quad \boldsymbol{W} = \begin{bmatrix} q_1 \\ q_2 \\ q_3 \\ q_4 \end{bmatrix} \tag{3}$$

where the U is the position vector and the W is the orientation quaternion values. At the end, the purpose is to get final pose $\boldsymbol{P_f}$, using a few seconds data of multiple IMU data $\boldsymbol{M}_1, \boldsymbol{M}_2, ..., \boldsymbol{M}_k$ and the initial position $\boldsymbol{P_i}$. Later the network will be extended to get both final pose $\boldsymbol{P_f}$ and IMU data $\boldsymbol{M_f}$. The data for $\boldsymbol{P}$ will be obtained from the visual tracker. Because we will have both the input and the correct output(label), the type of learning will be supervised in our experiments. All predictions of $\boldsymbol{P_f}$ will be predicted using the function $f$:

$$\boldsymbol{P_f} = f(\boldsymbol{P_i}, (\boldsymbol{M_1}, \boldsymbol{M_2}, ..., \boldsymbol{M_k}) \times Nseconds) \tag{4}$$

From Machine Learning perspective, the goal is to predict $\boldsymbol{P_f}$ using the input $\boldsymbol{P_i}, (\boldsymbol{M_1}, \boldsymbol{M_2}, ..., \boldsymbol{M_k}) \times \boldsymbol{Nseconds}$

# 4 State of The Art

There are many sensor fusion algorithms in the literature as mentioned in 2.1, and Kalman Filter is a very important one. Kalman filter, as first presented in study [10], is a well-established method for the linear systems. However, this linearity restriction is not practical, and Extended Kalman Filter is a nonlinear solution, which achieves it by approximating a nonlinear function using Taylor series. Bayesian Inference is another well-established analytical method, which is based on statistical inference, [11]. Dempster-Shafer reasoning has also inference mechanisms and, sensor fusion using Dempster-Shafer has been studied by Huadong Wu et al. in [12]. Also, there are other analytical fusion methods [14].

Machine Learning and, ANNs are able learn from data without being explicitly programmed. The contribution of this paper is to fuse multiple IMU data to get the pose of the object which represents the motion of the body mass. In [15], Ahuja et al. try to improve precision of IMU using Machine Learning, however, the experiment is very restricted such that only normal walking on a treadmill is learned using Support Vector Regression. In [18], Kyritsis et al. also incorporated machine learning, namely Support Vector Machines and LSTM cells, however the features to be extracted are decided manually. Another point is that IMUs data is fused in order to classify application specific movements.

In terms of using Deep Learning techniques, a similar study is [19]. In this study, tracking consists of mainly two parts, one of them is inertial tracking and the other is visual tracking. Visual tracking uses natural features of objects to estimate camera pose, and inertial tracking is achieved via a deep network, which consists of 1 LSTM layer and following 3 DNN layers, again to estimate camera pose. While visual approach is used mostly and is accurate, its accuracy decreases as the movement of camera distorts the image quality. In those cases, an error detection system does not allow the visual tracker's result to pass to the next component, which is a Kalman filter. Kalman filter is used to fuse the

output of visual and inertial trackers and provide a smoothed output. Even though the experimental setup and components look similar, in this experiment fusion is still done via a Kalman filter, and LSTMs are used for camera pose tracking.

In our study, multiple, non-registered, rigidly attached IMUs will be fused to get the change in the pose of a fixed point around the object(in our case it is the visual target), using Deep Learning techniques. In the previous study, the IMU data is used only to assist the tracking, however, it is the main device to track the object in our case. Another, and the most, important difference is that all the process is learned by the neural network, unlike the other where the Kalman Filter and error detection components should be created manually and carefully. Because of the nature of the data we are using, acceleration and angular velocity, the exact location of the object can not be determined, but any position tracker device, i.e. a camera, could be added to the neural network easily.

# 5 Data Collection

In order to test and train the model, the data collection should be done first, and in our case this part was the most challenging and hard one. The best condition of data to be collected would be that all the IMU's and visual component start recording the data at the same time and all frequencies are equal to each other. The target frequency in this project is 50 Hz. Since all the electronic components used in this project are of different brands, each of them has a different API, so one should learn and implement them all to collect data. Following, the software components are explained.

## 5.1 Individual IMU APIs

### 5.1.1 Xsens MTi Sensor

One of the IMU components is from Xsens MT 1S-Dev [20]. To be able to fetch the data from the device using a custom code, one needs to include XsensDeviceApi.dll to code environment with some supplementary XDA source files for C++ wrapper, to be able to use it in a C++ project.

### 5.1.2 InertialSense IMU

Next IMU sensor is $\mu$IMU Development Kit from InertialSense [21]. This device is one of the high quality sensors used in this project. Although it is able to reach 1000 Hz, we cannot use it since not all other devices are able to do so.

### 5.1.3 VectorNav

Another IMU is from VectorNav [22], VN-100 Development Board. Maximum gyroscope frequency 256 Hz, and it is already placed onto a controller board. To use interact with the device, one should include "vn/sensors.h". After writing the desired asynchronous output frequency using the function writeAsyncDataOutputFrequency, the asynchronous data could be read with a constant frequency.

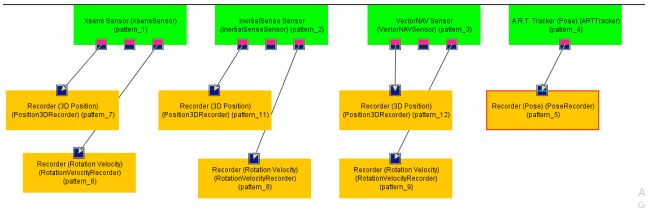
## 5.2 UbiTrack

Since we are using 3 different IMU sensors and 1 visual tracking from ART [23], we need an application to orchestrate all of these components. UbiTrack [24] is a solution for this, with some components already implemented. ART component is one of the implemented ones, so it could be used directly. However, for each IMU, an individual component has been written. According to the structure of the UbiTrack, one component should have a start, stop and startCapturing functions. For each IMU:

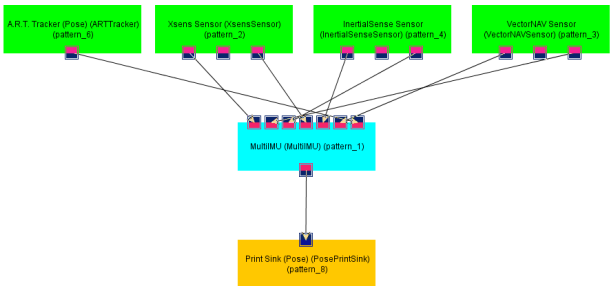
- The start function should be initiating the required port, connection to the IMU (device, in a general sense) and variables. At the end of this function, the startCapturing function is called using a different thread. That thread keeps running the while loop in there. This is an important point in the architecture since all the devices asynchronously.
- After terminating the parallel thread created by the start function, the stop function should be terminating connection to the device, removing garbage pointers and setting some required parameters.
- startCapturing should have a while loop to keep fetching data from the device and doing the required work with it.

The code for the components could be found in the attachment. These components for the pre-training data collection. In order to do a demo, live data needed and another component was written for that. It receives live data from all the IMUs, to predict the current location, and visual tracker, which is just for initiation and testing. After predicting the pose, the real and predicted location and pose is displayed in a simple representative visualization.

## 5.3 trackman



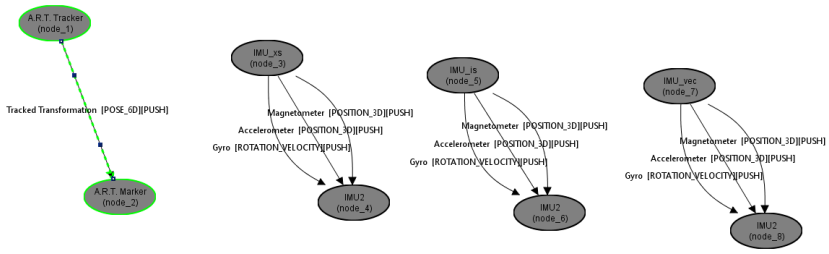

(a) Pre-training data collection data flow graph, screenshot from trackman



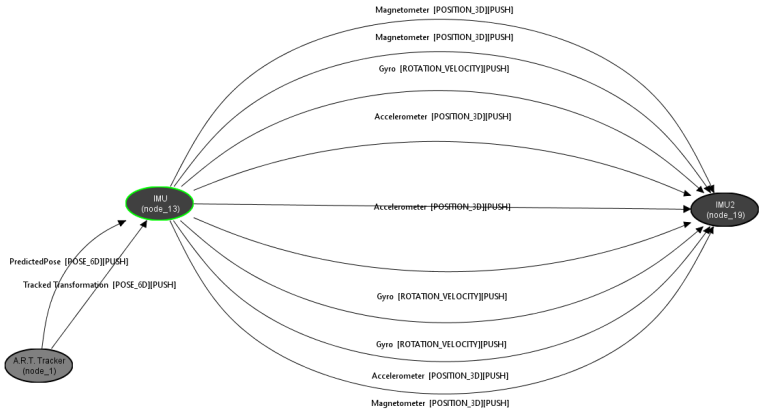

(b) Live demo data flow graph, screenshot from trackman

Figure 8: Data Flow Graphs

trackman [24] is a graphical planning and analysis tool used together with UbiTrack. In this project, data flow specification of spatial relation graph has been done using trackman. For two different applications, two different data flow graph has been created, one for pre-training data collection application, and the other is for live demo. In the Figure 8 data flow graphs, and in Figure 9 spatial relation graphs can be seen.




(a) Pre-training data collection spatial relationship graph, screenshot from trackman




(b) Live demo spatial relationship graph, screenshot from trackman

Figure 9: Spatial Relationship Graphs

Before collecting the data, the individual IMU software have been investigated and learned. Then, the components have been written according to the structure expected by the UbiTrack. Collected data from an IMU is written to two separate files, one of them is for acceleration, and the other is for angular velocity data, for each IMU. The acceleration files has the following format:

$$
\begin{array}{cccc}
 & & & \dots \\
timestamp1 & acc1_x & acc1_y & acc1_z \\
timestamp2 & acc2_x & acc2_y & acc2_z \\
 & & & \dots
\end{array}
$$

Each row corresponds to consecutive measurements and each row contains a local timestamp and acceleration data in 3 dimensions, separated by space.

Local timestamp means that the timestamp were read using the internal library functions of each IMU except the VectorNav. For that one, the timestamp is generated within from its UbiTrack component.

The same logic follows in the angular velocity data file. The format for that is:

$$\begin{array}{cccc} & & & \dots \\ timestamp1 & gyro1_x & gyro1_y & gyro1_z \\ timestamp2 & gyro2_x & gyro2_y & gyro2_z \\ & & & \dots \end{array}$$

For in $gyroi_u$, $i$ is an identifier of a measurement and $u$ is the axis of the measured angular velocity.

Another component is ART component, which we use it directly. The output format of ART visual tracker:

$$\begin{array}{ccccccccc} & & & & & & & & \dots \\ timestamp1 & quat1_a & quat1_b & quat1_c & quat1_d & pos1_x & pos1_y & pos1_z \\ timestamp2 & quat2_a & quat2_b & quat2_c & quat2_d & pos2_x & pos2_y & pos2_z \\ & & & & & & & & \dots \end{array}$$

Orientation is provided in the form of quaternions and the *quat* is its abbreviation, and position is abbreviated as *pos*.

## 5.4 Recordings

After getting all of components and data flow graphs(DFG files) ready, the data collection has started. In the created setup, each IMU is connected to the computer via a USB cable, and the visual tracker is via network. In order for visual tracker to work, the records should be taken in a very specific area, where the ART camera and the devices are setup. Also, with the cables connected to the computer, the area of the experiments is not larger than a small room. The records were taken for 4 days, each day approximately 2 hours long. Each day a random starting point and a sequence moves have been realized. During the second day, there had been some occlusions of the visual target, so at some points the visual tracker did not work. After the further investigations it has been understood that the target data (data from visual tracker) is misaligned with features data(data from IMUs). As will be mentioned later, misaligned data is a problem, and in this case it happened randomly, therefore the data from the second day is not used in the training process.

## 5.5 Issues

Data collection part is the most challenging part of this project. In order to feed the NN, a consistent size for the arrays is needed. This could be, for example, equal frequencies for the IMUs, so that at each time step an equal number of input data is provided, which is what we are trying to achieve in this project. However, the IMUs are of different brands and it took a lot of time to figure out how to put them in the same frequency, 50 Hz. And not all of them have the same dynamics in the code, for the case of Xsens and VectorNav we were able to specify an output frequency, but for InertialSense one needs to set the interval time between two measurements.

The problem resides even after putting the devices into the seemingly same frequency. Before starting the actual recordings, some small experiments about the consistency of the output data in terms of number of data points have been conducted. After collecting data for 10 minutes, the number of data points have been compared for each device. There were always some difference in the number of data points varying from 10 - 70, between all the devices. When these numbers are checked for the actual records used in the training, they are acceptable but not perfect. For the problematic recording, second day, the average number of difference in the number of data points of visual tracker with IMUs is 9500. This is possibly resulted from that visual tracker does not write any data to the record file in the case that it cannot detect any targets. And this has happened randomly during the experiment, which makes it really hard to recover. For other days' records, the difference with visual tracker is around 700 for day 1 out of around 264152 data points, 120 for day 3 out of around 284281 data points, and 340 for day 4 out of around 231630 data points. All these numbers are approximate, since the numbers of data points in an experiment is almost never equal for any IMU or visual tracker. Since this is a very important key point for this project, in the provided Jupyter Notebook, the output contains the exact number of data point differences, see Figure 10. As one might see from these examples, the difference is random for each record.

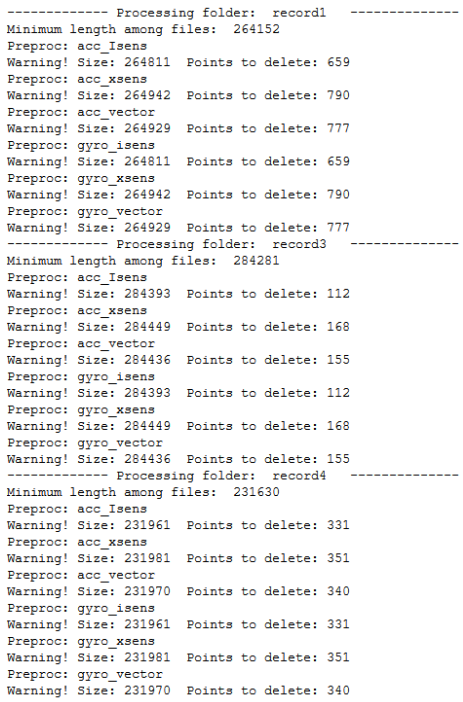



Figure 10: Jupyter Notebook output for the difference in the number of data points

# 6 Neural Network Design and Implementation

In order to design a working network, firstly the literature has been researched. Although there are not many examples of this task, one is able to understand the tools used from other relevant fields. Also, there are other tasks which consist of using acceleration data in neural networks, from which the nature of the data could be understood. In this sense, the study [25] has detailed explanations, where deep convolutional and recurrent networks used to process sensory data containing acceleration.

In the study [18] by Kritsis et al. , the LSTM layers are used with the IMU data, in order to extract long term dependencies, however, the features are extracted manually. Furthermore, Karpathy et al. [27] mentioned that at least 2 layers of LSTM layers would be useful. In order to classify eating gestures, Tara et al. [26] did benchmark neural network with the CNN, LSTM and ANN combinations, where the features are extracted by and data frequency variations are reduced with CNNs. Hence, after reading [18], [26], and [27], a rough design of the network has been shaped, and it is for sure that convolutional layers will be used in the network. However, the data at hand is not appropriate

to use as it is, so a preprocessing is needed.

## 6.1 Data Preprocessing

As mentioned in 5.5, configuring all devices to the same frequency is not enough to get equal number of data points from each device. In order to use the convolutional layers, we need some consistency in data. But as seen, the number of data points varies for each device and for each run, see Figure 10. The aim here is to get exactly same number of data points from each device to be able to put them in a matrix, so that convolutional layer could be fed with it. A simple solution would be to determine the minimum number got from a device, and cut the excessive ones. But not only the shape of the data is important, also the data points in the same index should be measurements of the same time, or at least approximately the same time. There are 3 simple approaches to remove excess data from each sensor and make the data as synchronized as possible. Here how it proceeds,

1. determine the minimum number of data points among IMUs and visual tracker data, say $size_{min}$.
2. For each sensor $X$ whose size is not equal to $size_{min}$, remove $size_{min} - size_X = diff_X$ points from sensor $X$.

First approach is to remove points from the beginning, i.e. remove first $diff_X$ points from $X$, simple illustration could be seen in Figure 11. This could eliminate the problems if the threads started by the UbiTrack have time differences. Random data visualization after this approach is applied could be seen in 14.a. As seen in the graph, it does not work, since there is a clear shift among the IMUs.

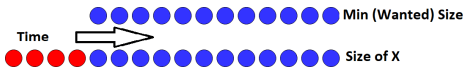



Figure 11: Remove from the beginning, data preprocessing approach illustration

Second approach is to remove data uniformly. For sensor $X$, whose size is $size_X$, and data point indices are $0, 1, 2...size_X - 1$, remove $diff_X$ data points with equally spaced indices, starting from 0 to $size_X - 1$, simple illustration is in Figure 12. A random data visualization after applying this could be seen in 14.b. This approach could have solved the problems of unequal frequencies, since three different sensors used this is a possible case. From the illustration, it could be deduced that this approach works better than the first approach, removing from the beginning.

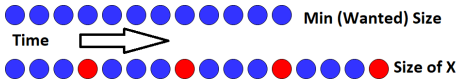



Figure 12: Remove uniformly, data preprocessing approach illustration

Third, the last, approach is to remove from the end. Simple illustration could be seen in Figure 13 Random data visualization is in 14.c. From further investigations, it is realized that removing from the end yields better results than the second approach.

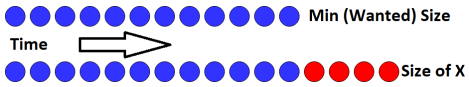
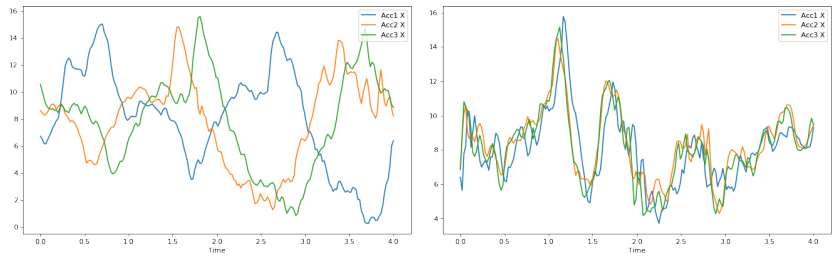


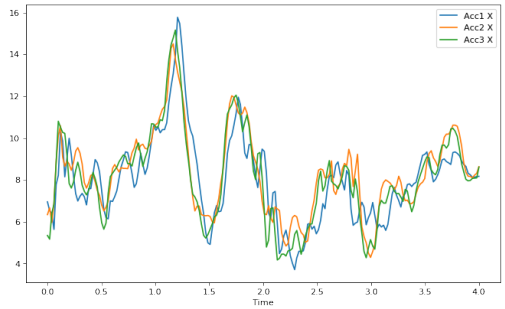

Figure 13: Remove from the end, data preprocessing approach illustration

Even though for the data which is in the beginning of the record they perform almost the same, from the synchronization point of view, in the end of the record, the third approach is better. Hence, the sensors are providing the data around similar frequencies, and they mostly start around the same time, however, when stopping the record, some of them take more time to do it, which is what could be understood from this set of data.




(a) Excess data is removed from the beginning and as seen the data is not synchronized at all.

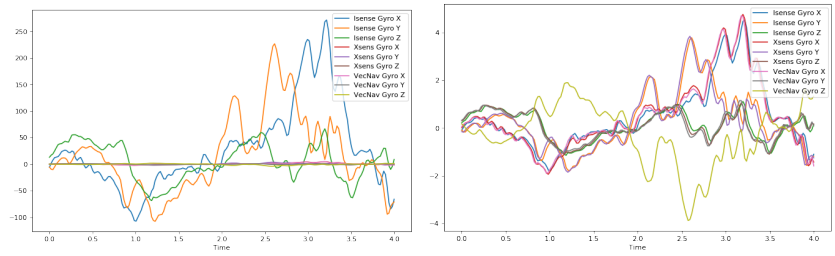
(b) Uniformly removed data. As seen it is better than removing from the beginning.




(c) Uniformly removed data. As seen it is better than removing from the beginning.

Figure 14: Data preprocessing random visualizations. The main goals here are to make the sensor data arrays' lengths equal to each other and to make them as synchronous as possible.

Later, it is found that InertialSense angular velocity values have a very different range than the other sensors' values. Then angular velocity values are scaled with 60, to get very close ranges with other IMUs. Before and after the scaling could be seen in Figure 15.




(a) Before scaling InertialSense angular velocity values.

(b) After scaling InertialSense angular velocity values.

Figure 15: InertialSense angular velocity scaling

Also a summary of the all the data at hand could be seen in Figure 16.

| | count | mean | std | min | 25% | 50% | 75% | max |
|---|---|---|---|---|---|---|---|---|
| **Isense Acc X** | 15589600.0 | 7.936959 | 2.649596 | -13.956030 | 7.127385 | 8.730714 | 9.539513 | 43.812809 |
| **Isense Acc Y** | 15589600.0 | 1.036869 | 3.611251 | -15.871825 | -1.304079 | 0.861210 | 3.351248 | 18.387049 |
| **Isense Acc Z** | 15589600.0 | 0.811349 | 3.798139 | -17.100693 | -1.877316 | 0.720712 | 3.387090 | 15.963652 |
| **Xsens Acc X** | 15589600.0 | 7.932414 | 2.596867 | -16.401899 | 7.120981 | 8.595177 | 9.469381 | 32.769958 |
| **Xsens Acc Y** | 15589600.0 | 0.444237 | 3.738038 | -18.202150 | -2.034191 | 0.204876 | 2.870667 | 21.169971 |
| **Xsens Acc Z** | 15589600.0 | 0.445715 | 3.801634 | -17.371794 | -2.258393 | 0.355857 | 3.013232 | 15.318937 |
| **VecNav Acc X** | 15589600.0 | 7.902559 | 2.676602 | -16.639000 | 7.072000 | 8.630000 | 9.492000 | 29.065001 |
| **VecNav Acc Y** | 15589600.0 | 0.691015 | 3.800145 | -18.719000 | -1.964000 | 0.644000 | 3.259000 | 16.687000 |
| **VecNav Acc Z** | 15589600.0 | -0.713929 | 3.702668 | -17.055000 | -3.197000 | -0.454000 | 1.719000 | 18.224001 |
| **Isense Gyro X** | 15589600.0 | -0.000372 | 0.835120 | -10.716225 | -0.124902 | 0.003549 | 0.131221 | 9.949092 |
| **Isense Gyro Y** | 15589600.0 | 0.005972 | 0.548251 | -6.166442 | -0.091622 | -0.000437 | 0.087691 | 7.160093 |
| **Isense Gyro Z** | 15589600.0 | -0.012678 | 0.587398 | -8.351933 | -0.102612 | -0.000269 | 0.090381 | 9.532391 |
| **Xsens Gyro X** | 15589600.0 | -0.001635 | 0.874962 | -11.169568 | -0.130244 | 0.002593 | 0.133372 | 10.139979 |
| **Xsens Gyro Y** | 15589600.0 | 0.008857 | 0.576427 | -6.547544 | -0.095376 | 0.001754 | 0.097265 | 7.373504 |
| **Xsens Gyro Z** | 15589600.0 | -0.008914 | 0.612053 | -8.788579 | -0.103346 | 0.004298 | 0.099737 | 9.708364 |
| **VecNav Gyro X** | 15589600.0 | -0.003724 | 0.872629 | -11.203337 | -0.133427 | 0.000271 | 0.132369 | 10.290122 |
| **VecNav Gyro Y** | 15589600.0 | -0.012821 | 0.608895 | -8.706632 | -0.105575 | 0.000204 | 0.094841 | 9.710053 |
| **VecNav Gyro Z** | 15589600.0 | -0.006010 | 0.578220 | -7.284298 | -0.093189 | 0.000200 | 0.097095 | 6.772167 |

Figure 16: Overall Data Summary

Before finding this miss-scaled data, initial 19 networks were already trained. Because of this, initial 19 model versions are useless, since the decisions are made based on only the InertialSense angular velocity data.

After getting all the data into the same size, we need to reshape the data

to get it ready for the training and testing. Our training procedure will be based on the time series estimation, since we have a time based data it would make sense. For each of the training points, there should be one initial position and orientation, IMU data from the initial position's time to target position's data time, and target position and orientation.In summary, predicting the target position and orientation from the initial position and orientation using a number of IMU data, and this is implemented as sliding window approach.

In our case, the window size is 4 seconds of data, considering that the frequency of the devices is 50 Hz, the number of data points in a window is 200, each data point containing 18 values, 6 sensory data(3 acceleration and 3 angular velocity) for 3 different IMU. Due to the scarcity of the data, data reuse is needed, so when sliding the window it is slid for 4/20 seconds = 200/20 data points = 10 data points(stride), hence each data point is used for 20 times, in different windows. In the Jupyter Notebook, this is done with the function named *create_train_data*.

## 6.2 Neural Network Design

In this section, the design choices will be done and justified. In order to compare different configurations, an initial network is created at the beginning. The first input of the neural network is in the size of 200x18, where 200 corresponds to 4 seconds of IMU data, for 3 IMU devices where each of them gives 3 acceleration and 3 angular velocity values, which makes 18 for each data point. This input is fed into convolutional part of the network, there are 4 convolutional layers and each of them has size of 64, with kernel size of 12. It should be noted that the convolutional layers here are not 2 dimensional, they are 1 dimensional convolutions since the all the data at hand is 1 dimensional, which is time, unlike the images which have 2 dimensions(excluding color channels). 4 layers of convolutions are followed by 2 LSTM layers, whose sizes are 128. At this point, the second input of the network, which is initial position( of size 1x7, 3 for position, 4 for orientation with quaternion), is concatenated with the LSTM's output. With this, it is expected that using the feature data and the initial position, the network will be predicting the final position and orientation. After concatenation, there are 3 dense layers whose size are 128, 128 and 7, respectively. At the end size 7 is mandatory because the final position and orientation data consists of 7 values. Another point is the initial regularization technique. It is chosen as batch normalization at the beginning, with the learning rate of $10^{-5}$. A visualization of the initial neural network could be seen in Figure 17.

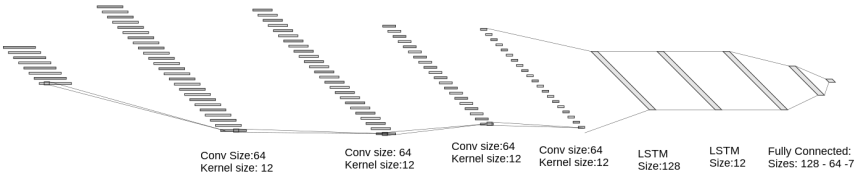



Figure 17: Initial Neural Network. Note that after each and every layer, a batch normalization layer then the activation function comes. For LSTM layers, activation function is hyperbolic tangent function, and for all the others, it is Rectified Linear Unit(ReLU).

Also the activation functions of all layers are ReLU (Rectified Linear Unit), except the LSTM layers which is tanh (hyperbolic tangent). The optimizer used is Adam, [29], which is the most common optimizer. There also other optimization algorithms like Momentum, RMSProp, Adagrad, Adadelta and Adamax [34].The default batch size is 128. As the size of batch is increased, the training time decreases but smaller batch sizes could lead to better models. To balance it, the size 128 is chosen. Also, 85% of the data is used as training and 15% as validation data.

### 6.2.1 Loss Function

The task done with the neural network could be defined as regression, by supervised learning. In order for the network to learn some information from the data, first it should have definition of what is good and what is bad to learn, what to learn and what not to. This is the use of the loss functions. In our case there are two common possible options, one of them is Mean Absolute Error(MAE) and the other is Mean Square Error(MSE). Formula of each is as follows:

$$MAE = 1/n \sum_{i=1}^{N} |y_j - \hat{y}_j|$$

$$MSE = 1/n \sum_{i=1}^{N} (y_j - \hat{y}_j)^2$$

where $y_j$ is the label(correct value/output), and $\hat{y}_j$ is the predicted value/output. Each of them has some advantageous and disadvantageous properties. Starting with MAE, it is more robust to outliers, since there is no square operation in it. In case of an outlier, all the unsigned distances from the label is averaged. However this is not the case for the MSE. In case of an outlier, with the square operation the loss gets very high, and the optimization is done using the loss that is unreasonably shifted by one point, at the expense of all other common

points. On the other hand, MSE's gradient is higher for larger values, and lower for the losses which are closer to 0. But MAE's gradient is same for either high or low values. Hence the with MAE the network will be learning always at the same speed, which would increase the training time.

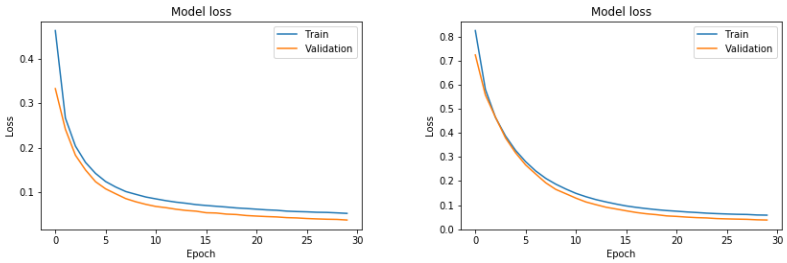
At the end, the choice depends on the data at hand. As could be seen in the data summary Figure 16 , there are not extreme outliers. After trying MAE and MSE in some models, it was seen that MSE actually results to better predictions than MAE. Therefore, using MSE would be a better choice in this project, so that the nice gradient properties of the MSE will be utilized and better predictions are obtained.

### 6.2.2 Convolutional Layers

At each step, we have 4 seconds of acceleration and angular velocity data from 3 IMUs, which is raw data with a lot of noise, as input. First, the variations should be reduced, using convolutional layers. This would also provide us with extracting high-level features from the raw data, as mentioned in studies [25] and [26]. As we have higher and more abstract features it is easier for recurrent and dense/fully connected layers to use those features. Just to decide on the number of convolutional layers and their size, there are several number of configurations to try. Utilizing the experience in the literature, the number of layers will be either 3 or 4. The experiment results for these 2 configurations could be found in Table 1, and the loss graphs in Figure 18.

| **# of Convolutional Layers** | **Best loss** |
|---|---|
| 3 | 0.0360 |
| 4 | 0.0377 |

Table 1: Number of Convolutional Layers Comparison, after Training for 30 Epochs




(a) Loss graph for 3 Convolutional Layers

(b) Loss graph for 4 Convolutional Layers

Figure 18

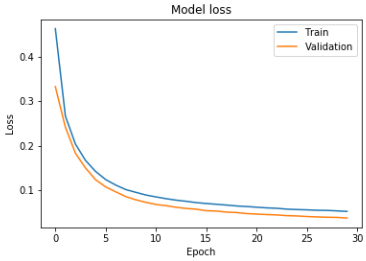
As seen in loss table, 3 layers has a smaller loss, so it performs (slightly) better than 4 layers. Also, 3 layers structures loss mostly start from around 0.4, although 4 layers has never started with a loss smaller than 0.6. From the complexity point of view, 3 layers structure would have a smaller runtime, since there is one layer missing, and there are less parameters to learn. At the end, 3 layers structure is adopted, for all these convincing reasons.

Another important parameter for convolutional layers is the size of the layer. As the size gets bigger, it will be able to represent more features in it, which will be used in the next layers. As the main purpose of the CNNs is to extract useful features for the later layers, a larger size would possibly be leading to better predictions. After several experiments, the loss graph and table for the most important ones could be find in Figure 19 and Table 2.

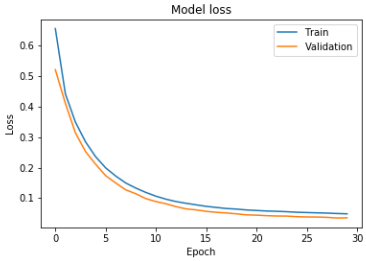
| Convolutional Layer Size | Best Loss | Time For 100 Predictions |
|---|---|---|
| 64 | 0.0360 | 6.78 secs |
| 128 | 0.0346 | 7.30 secs |
| 256 | 0.0491 | 9.89 secs |
| 512 | 0.0350 | 12.88 secs |

Table 2: Convolutional Layers' Size Comparison, after Training for 30 Epochs. Rightmost column is for the elapsed time for the 100 predictions in a CPU environment, whose unit is in seconds.

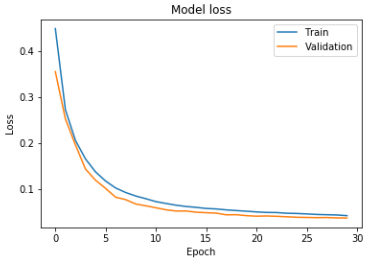
For these parameter, sizes of 128 and 512 are the main competing configurations. Multiple aspects of each should be considered carefully for this current project. First of all, the best losses from 30 epochs of training are almost equal to each other. Even though the size 128 configuration did start from a far worse starting point, it managed to finish training with a better loss.



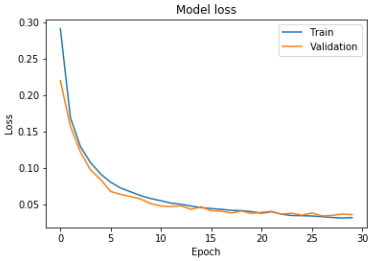

(a) Loss graph for Convolutional Layer size 64




(b) Loss graph for Convolutional Layer size 128




(c) Loss graph for Convolutional Layer size 256

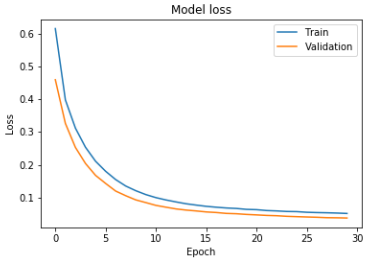



(d) Loss graph for Convolutional Layer size 512

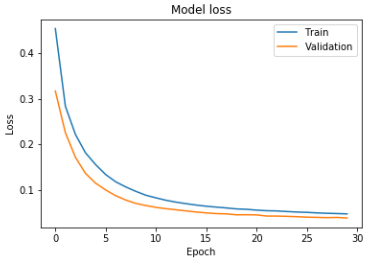
Figure 19

On the other hand, size 512 starts converging to its limit faster than 128, which makes it very fast to train. This can be seen in the Figure, after 10.th epoch the validation loss of size 512 starts to fluctuate, but size 128 keeps steadily decreasing. However, it comes with a con. As this application is based on real-time data, the output, ideally, should be produced as fast as possible. The time required for making 100 predictions for size 512 is 12.88 seconds, which almost doubles the time for size 128, 7.30 seconds. Since the difference between the best losses is not large enough to consider, for mostly the complexity reasons, the size is chosen as 128, so the size of 3 convolutional layers will be 128.

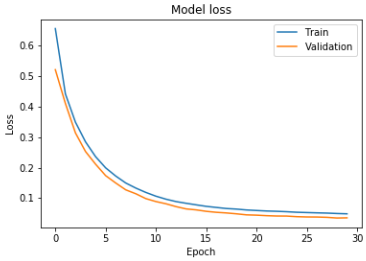
The last important parameter for the convolutional layers is the kernel size. Kernel size will be determining the receptive area of the kernel. As the kernel size gets bigger, its receptive area would be larger. As presented in the very popular convolutional neural networks i.e. [30], [31], the kernel sizes are mostly chosen as small for example 3x3, 5x5 or 7x7. In our experiments, kernel sizes of 4, 10, 12, 16 and 20 are tried. The loss graphs of them is in Figure 20 and the losses after 30 epochs is in Table 3.



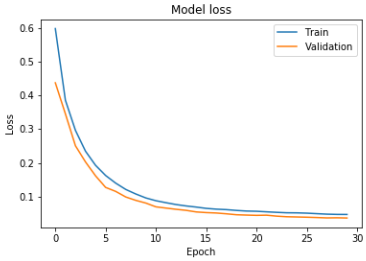

(a) Loss graph for kernel size 4



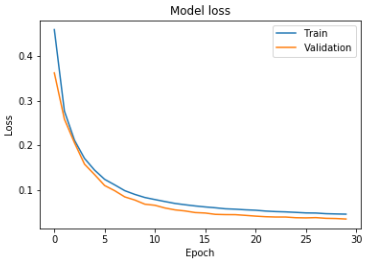

(b) Loss graph for kernel size 10




(c) Loss graph for kernel size 12




(d) Loss graph for kernel size 16




(e) Loss graph for kernel size 20

Figure 20

As prediction times are very close to each other, here the most important metric is the loss. The best candidates are the kernel size 12 and 20. For kernel size, the number of parameters to learn is 12x12x128 for each convolutional layer, but for kernel size 20 it is 20x20x128, and there are multiple layers. To keep the number of parameters as low as possible, kernel size 12 is chosen.

| Kernel Size | Best Loss | Time For 100 Predictions |
|---|---|---|
| 4 | 0.0375 | 7.66 secs |
| 10 | 0.0380 | 6.74 secs |
| 12 | 0.0346 | 6.52 secs |
| 16 | 0.0373 | 6.41 secs |
| 20 | 0.0354 | 6.42 secs |

Table 3: Kernel size comparison for Convolutional Layers, after Training for 30 Epochs. Rightmost column is for the elapsed time for the 100 predictions, its units is in seconds and the code was run on CPUs.

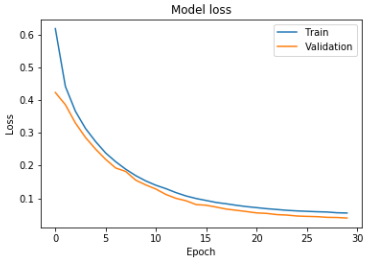
#### 6.2.3 Recurrent(LSTM) Layers

LSTM layers are presented in the network to utilize the temporal feature of the data,so that the network would have a notion of time. In the network, they are placed after the last convolutional layer, therefore they will be operating on high level data, the output of the CNN network. To obtain the best of its performance, there are some hyperparameters to be tuned. As in Convolutional layers, the size of each LSTM layer is important. As the number increases there will be more features to be learned, in theory, but also the prediction and training time increases. The experiments are run on 4 different sizes, which are 32, 64, 128, and 256. The table for the best losses after 30 epochs is in Table 4, and the loss graphs are in Figure 21.

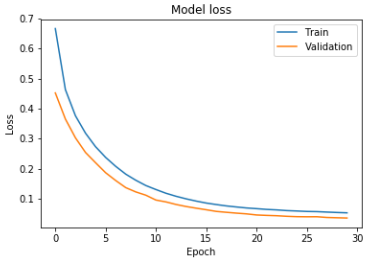
| LSTM Layer Size | Best Loss |
|---|---|
| 32 | 0.0352 |
| 64 | 0.0355 |
| 128 | 0.0354 |
| 256 | 0.0381 |

Table 4: LSTM Size Comparison, after Training for 30 Epochs.

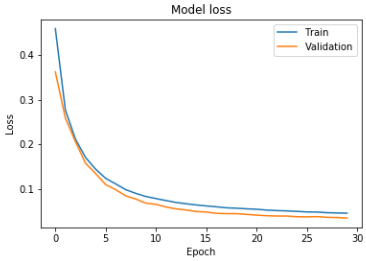
As in the convolutional layers' size, the largest size tried starts fluctuating after around epoch 10, which means it is getting closer to its limits, for the current learning rate, which is $10^{-5}$. Other 3 yields almost the same best loss, hence, the simplest solution is the size of 32.



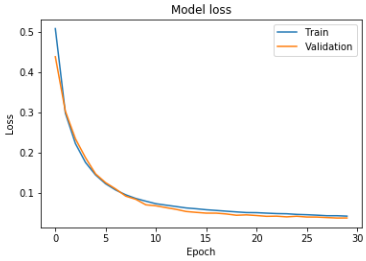

(a) Loss graph for LSTM size 32




(b) Loss graph for LSTM size 64




(c) Loss graph for LSTM size 128




(d) Loss graph for LSTM size 256

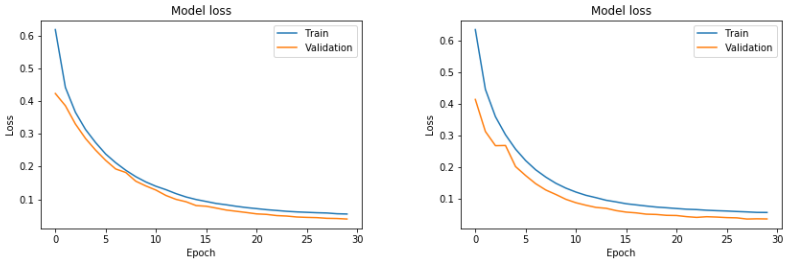
Figure 21

Second and the last important hyperparameter for the LSTM layers is the number of them. In the study [27], it is mentioned that at least 2 layers of recurrent units would be beneficial. For this reason, the depth of 2 and 3 are tried. The loss graphs is in Figure 22 and the best losses are in Table 5.

| Number of LSTM Layers | Best Loss |
|---|---|
| 2 | 0.0352 |
| 3 | 0.0355 |

Table 5: Number of LSTM Layers Comparison, after Training for 30 Epochs.

Graphs are almost identical, they start around same values and end nearly the same. To keep the structure simple, 2 LSTM layers are chosen, which also yields similar loss and behaviour as 3 layers.



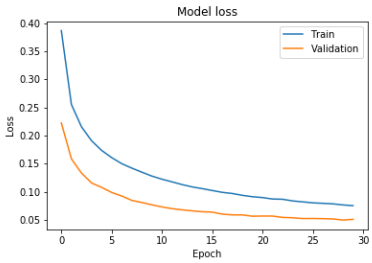

(a) Loss graph for 2 LSTM Layers (b) Loss graph for 3 LSTM Layers

Figure 22: Number of LSTM Layers Comparison, after Training for 30 Epochs.

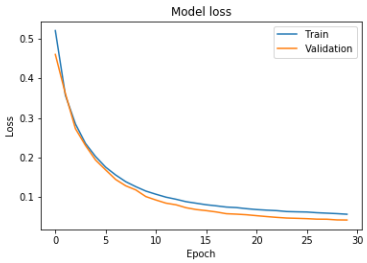
### 6.2.4 Dense (Fully Connected) Layers

After learning high level features and getting the temporal features from the data, now the last set of layers are dense or fully connected layers. Like the others, here the size and the number of them are important. However, in this layer there is an important limitation on the last layer. As the main goal is to predict the position and the orientation of the target, and the last layer is the output of the network, the label data(which consists of position and orientation) and the output should be of the same size. In the output of the visual tracker, the position is represented with 3 points, which are X, Y and Z position, and orientation is represented with 4 points, which corresponds to the quaternion of the object. Hence, the output layer should be of size 7.

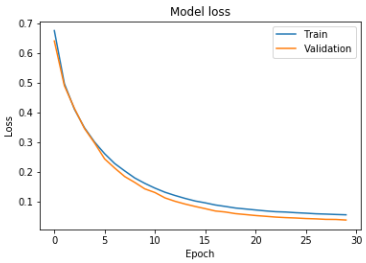
During the training phase, the output of the layer(predicted output) and the label of the current input(correct output) will be compared to each other, and the network will be learning from these mistakes. First the number of layers are decided. The loss graphs are in Figure 23 and the best losses are in Table 6. The validation loss for 1 dense layer starts from a very low point, but it does not improve a lot. From the best losses, it is clear that 3 layers yield the best loss.




(a) Loss graph for 1 Dense Layer




(b) Loss graph for 2 Dense Layers




(c) Loss graph for 3 Dense Layers

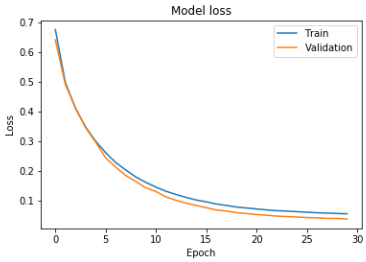
Figure 23: Number of Dense Layers Comparison, after Training for 30 Epochs.

The second hyperparameter for dense layers is the size of them. As the size increases, it would possibly be able to learn different features and map them into better results. On the other hand, more hidden units could make the network train very slowly, since more parameters will be presented. This dense layers are placed after the last LSTM layer, and the convention 128-64-7 means that there are 3 dense layers, the one after the last LSTM has size 128, and then 64 sized layer comes and then 7 sized.

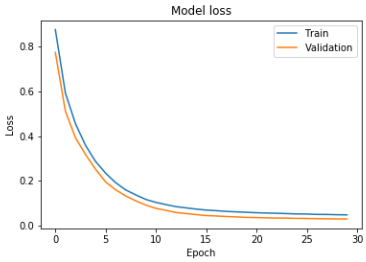
| Number of Dense Layers | Best Loss |
|---|---|
| 1 | 0.0511 |
| 2 | 0.0428 |
| 3 | 0.0390 |

Table 6: Number of Dense Layers Comparison, after Training for 30 Epochs.

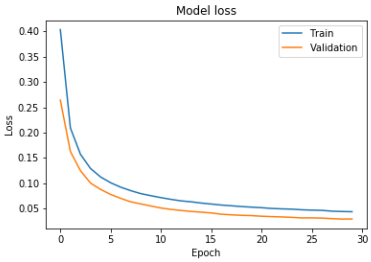
The loss graphs are in Figure 24 and the best losses are in the Table 7. In this part the decision is comparably easier to make, as the size increases the network learned more, so the size of 1024-1024-7 is chosen.



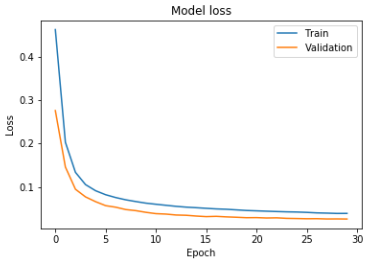

(a) Loss graph for
Dense layer sizes 128-64-7



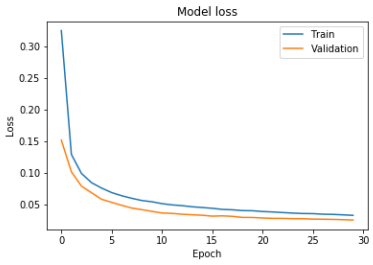

(b) Loss graph for
Dense layer sizes 256-128-7




(c) Loss graph for
Dense layer sizes 256-256-7




(d) Loss graph for
Dense layer sizes 512-512-7




(e) Loss graph for
Dense layer sizes 1024-1024-7

Figure 24: Comparison for Dense layer sizes, after training for 30 Epochs

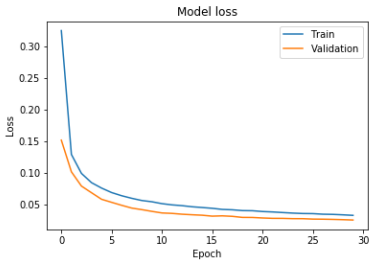
| Dense Layer Sizes | Best Loss |
|---|---|
| 128-64-7 | 0.0390 |
| 256-128-7 | 0.0313 |
| 256-256-7 | 0.0298 |
| 512-512-7 | 0.0257 |
| 1024-1024-7 | 0.0250 |

Table 7: Kernel size comparison for Convolutional Layers, after training for 30 Epochs. Rightmost column is for the elapsed time for the 100 predictions, its units is in seconds and the code was run on CPUs.

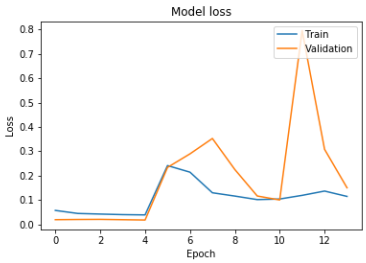
#### 6.2.5 Regularization Techniques

As the size or the depth of the neural networks grow, they are able to learn more. This learning is done on a dataset which is already at hand, however, the main goal is to perform good on a new data, i.e. classify the new object correctly or as in our case predict the new location and orientation as correct as possible. Hence, as the network learns better, there is risk of learning the training data too much, so that when a new data is presented it cannot perform good. This issue is a common one, and it is called overfitting. To overcome this, there are techniques, mainly called as regularization techniques. Until this part of the experiments, batch normalization(BN) is used after each single layer, except the output layer. Ioffe et al. [32] introduced batch normalization, it is suggested that the BN layer is used after the layer and before the activation function, which is the practice used in our network. Experiments are also done with dropout regularization, but the behaviour of the network changes a lot in this case. Note that, in case of dropout, the general order is as follows:
Layer $\rightarrow$ Activation $\rightarrow$ Dropout, unlike batch normalization. Thanks to the BN layers, data normalization is not needed, because it is already done after each layer, which is one of the merits of BN.

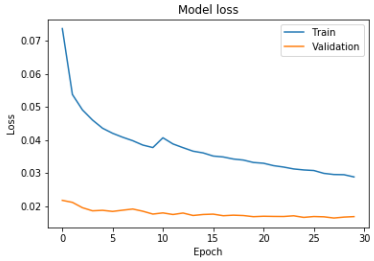
In the experiment where dropout is applied, the network did not learn anything at first, with a best loss around $10^8$. After the input data is normalized, even though the learning behaviour is not a regular one, the loss of 0.0183 is achieved, which could be seen in Figure 25.b. This is the best loss until this point, and it is achieved with only one trial using dropout. At this point, it is obvious that this loss jumps are caused by the learning rate, which will be discussed in the next parts. After small adjustment to the learning rate, which is $10^{-4}$, the dropout loss values behaved normally, with best loss of 0.0164. The loss graphs of dropout and batch normalization are in Figure 25 and the best losses are in Table 8.




(a) Loss graph for Batch Normalization Regularization




(b) Loss graph for Dropout with Learning Rate of $10^{-3}$




(c) Loss graph for Dropout with Learning Rate of $10^{-4}$

Figure 25: Regularization Technique Comparison with different learning rates

However, in this point, one could not decide that dropout is better for this application, since the learning rate is adjusted for it, but not for batch normalization. Therefore, in the next part, where learning rate is discussed, the experiments will be done for both of them.

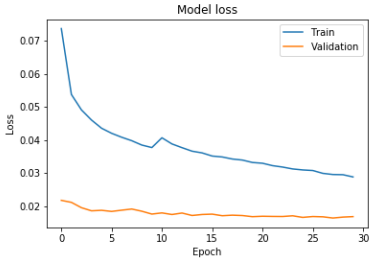
| Regularization Technique | Best Loss |
|---|---|
| Batch Normalization | 0.0250 |
| Dropout with learning rate 10−3 | 0.0183 |
| Dropout with learning rate 10−4 | 0.0164 |

Table 8: Regularization Technique Comparison, after Training for 30 Epochs.

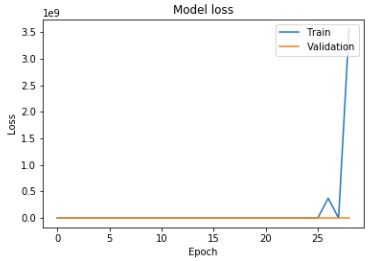
### 6.2.6 Learning Rate

For most, if not all, of the neural networks, learning rate is one of the most important hyperparameters. It could be described as the size of the step to take after each batch, with other words, how much to learn, how much to update the weights of the network after each batch. The learning with neural network is almost always a non-convex optimization, so the target function has a lot

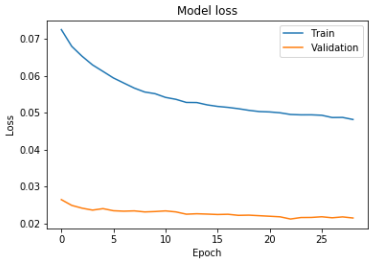
of local minimum and it could be varying a lot in small ranges. Hence, when the learning is set to a large number, the loss value would be behaving very unsteadily. When it is set to a small number, it would be learning very slowly, where the training takes a huge amount of time, without even knowing that that minima is a good one.



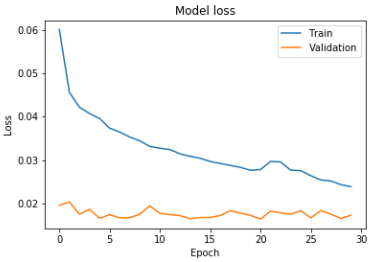

(a) Loss graph for Dropout with Learning Rate of $10^{-4}$




(b) Loss graph for Dropout with Learning Rate of $3 \times 10^{-4}$



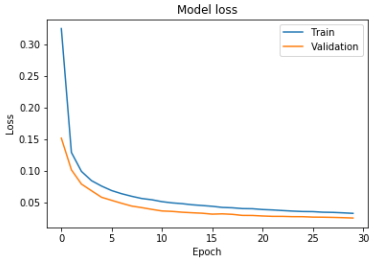

(c) Loss graph for Dropout with Learning Rate of $2 \times 10^{-4}$



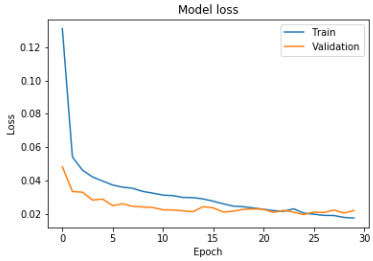

(d) Loss graph for Dropout with Learning Rate of $5 \times 10^{-4}$

Figure 26: Learning Rate Comparison for Dropout

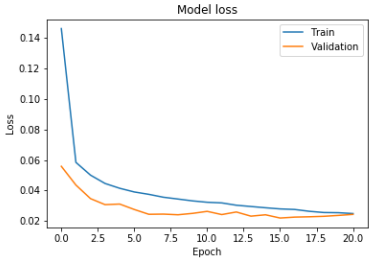
The loss graphs and the best losses could be found in Figures 26 and 27, and in Table 9, for the learning rate experiments.



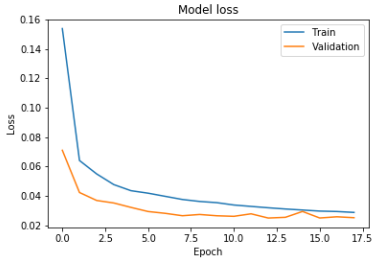

(a) Loss graph for Batch Normalization with Learning Rate of $10^{-5}$



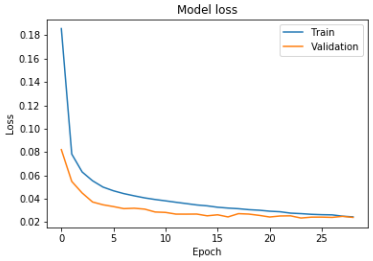

(b) Loss graph for Batch Normalization with Learning Rate of $10^{-4}$




(c) Loss graph for Batch Normalization with Learning Rate of $7 \times 10^{-5}$




(d) Loss graph for Batch Normalization with Learning Rate of $5 \times 10^{-5}$




(e) Loss graph for Batch Normalization with Learning Rate of $3 \times 10^{-5}$

Figure 27: Learning Rate Comparison for Batch Normalization

From the figures and the table, it is clear that Dropout starts training with a lower loss than the batch normalization approach in almost all experiments. Also, the final loss reached in dropout approach with learning rate $3 \times 10^{-4}$ is the best loss until this experiment. Based on these convincing results of dropout regularization, it will be used in the further models, instead of batch normalization.

| Regularization Technique | Learning Rate | Best Loss |
|---|---|---|
| Dropout | $10-4$ | 0.0164 |
| Dropout | $3 \times 10-4$ | 0.0162 |
| Dropout | $2 \times 10-4$ | 0.0212 |
| Dropout | $5 \times 10-4$ | 0.0168 |
| Batch Normalization | $5 \times 10-4$ | 0.0250 |
| Batch Normalization | $10-4$ | 0.0219 |
| Batch Normalization | $7 \times 10-5$ | 0.0219 |
| Batch Normalization | $5 \times 10-5$ | 0.0250 |
| Batch Normalization | $3 \times 10-5$ | 0.0234 |

Table 9: Regularization Technique Comparison

The final neural network model illustration could be find in Figure 28.

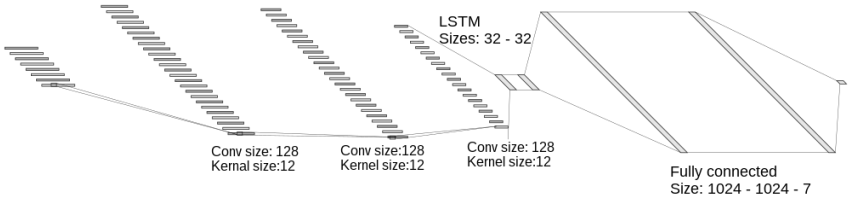



Figure 28: Final Neural Network architecture. Note that after each and every layer, first the corresponding activation function and then a dropout layer comes. For LSTM layers, the activation function is hyperbolic tangent function, and for all the others, it is Rectified Linear Unit(ReLU). These activations are followed by dropout layers.

Additionally, TensorBoard visualization of the final neural network is in Figure 29. TensorBoard is a suite of visualization tools for the TensorFlow applications [33].

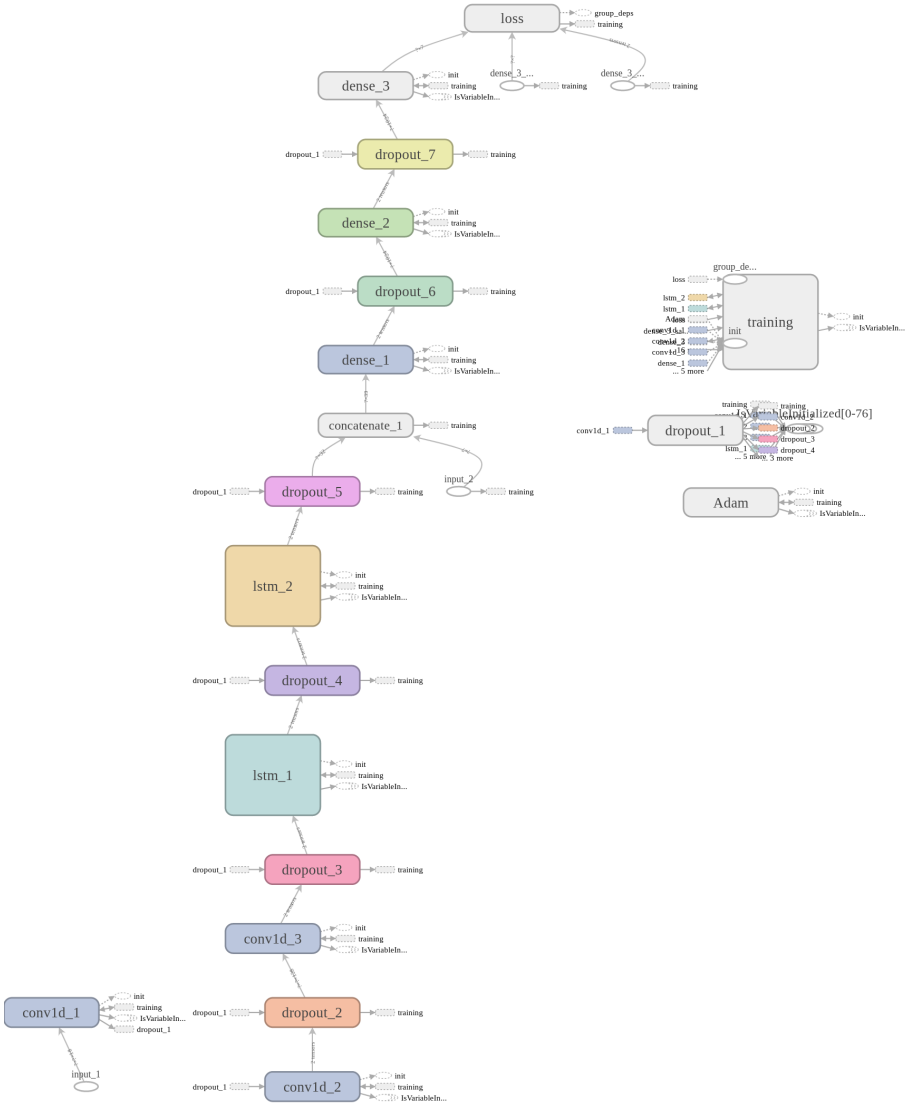



Figure 29: TensorBoard visualization of the final neural network

## 6.3 Implementation

The neural network implementation is done in Jupyter Notebook environment [35] using Keras Library [36].

It is observed that use of GPU instead of a CPU saves from long training time. While using a Dual-Core Intel i5-4210 CPU needs 5 minutes and 45

seconds for an epoch to train, Nvidia GeForce GTX 1080 needs only 55 seconds. So, the use of a GPU is highly encouraged.

### 6.4 Issues

UbiTrack creates different threads to read the data from each component/device, which means the time difference between very first data points of each IMU and visual tracker varies randomly. In Data Preprocessing 6.1, the main approach was to overlap the data from different sources onto each other, so that the peaks and bottoms would be at the same indices of the array. With each recording, the time difference will be changing, and possibly the data in same indices would be corresponding to different time for each device, even after preprocessing. However, the important point here is the difference in indices of the corresponding peaks and bottoms. Since the data is in discrete steps, the exact time difference is not the point of focus, but the difference in the number of indices is. In order to get better predictions, the number of recording sessions should be high so that the network learns as many indices difference between peaks and bottoms as possible, assuming that there will be a worst case scenario, i.e. maximum index difference is 100.

## 7 Results

Test data is extracted from last 5 minutes of each day's recording. Since we have 3 recordings (actually 4, but the second day cannot be used), in total we have 15 minutes of test data. The tests are conducted 1 minute each. To predict the next pose, last 4 seconds of IMU and label pose data is needed. Hence, when extracting the test data, 5 minutes + 4seconds data extracted, in order to get 5 minutes prediction. For each minute, starting from the first 4 seconds of IMU and pose data, the prediction is done, and the array keeping the initial pose data is appended with the prediction. Since the prediction in a step will be used as initial pose 4 seconds later, this structure is required.

The errors reported in this section are Mean Absolute Error(MAE), which is explained in 6.2.1. The reason is that MAE could be interpreted easier and the units of the position errors remains as meters.

The errors from each minute of each recording for the quaternions could be found in Table 10.

| Day Number | Minute | MAE Q1 | MAE Q2 | MAE Q3 | MAE Q4 |
|---|---|---|---|---|---|
| 1 | 1 | 0.29794178 | 0.47889321 | 0.54221289 | 0.19367892 |
| 1 | 2 | 0.2873525 | 0.39352458 | 0.38461821 | 0.19109086 |
| 1 | 3 | 0.30603866 | 0.37098453 | 0.25118557 | 0.10952405 |
| 1 | 4 | 0.28429213 | 0.40921022 | 0.33418614 | 0.19673632 |
| 1 | 5 | 0.24749812 | 0.52618677 | 0.32894294 | 0.13705228 |
| 3 | 1 | 0.24387641 | 0.52349242 | 0.49723168 | 0.11318289 |
| 3 | 2 | 0.20949533 | 0.50537554 | 0.5669253 | 0.13772507 |
| 3 | 3 | 0.18332603 | 0.64131045 | 0.69083356 | 0.15293999 |
| 3 | 4 | 0.20553715 | 0.44577435 | 0.56609571 | 0.16589802 |
| 3 | 5 | 0.27263279 | 0.469891 | 0.59259838 | 0.2110102 |
| 4 | 1 | 0.28712006 | 0.19753273 | 0.28266957 | 0.16927738 |
| 4 | 2 | 0.04945909 | 0.09094302 | 0.07779159 | 0.12041667 |
| 4 | 3 | 0.07588424 | 0.13071892 | 0.15345349 | 0.12953166 |
| 4 | 4 | 0.08677928 | 0.16520313 | 0.29060942 | 0.11231395 |
| 4 | 5 | 0.12565777 | 0.11478203 | 0.23180115 | 0.11637267 |

Table 10: Resulting Quaternion MAEs for the model which trained with all 3 days' data.

Again for the quaternions, the mean MAEs in the quaternion values could be seen in 11. The MAEs of different minutes within each day are averaged and the table shows the results. From the table it is understood that the model performs better on the data of day 4.th.

| Day Number | Mean MAE Q1 | Mean MAE Q2 | Mean MAE Q3 | Mean MAE Q4 |
|---|---|---|---|---|
| 1 | 0.28462464 | 0.43575986 | 0.36822915 | 0.16561649 |
| 3 | 0.22297354 | 0.51716875 | 0.58273693 | 0.15615124 |
| 4 | 0.12498009 | 0.13983596 | 0.20726504 | 0.12958247 |

Table 11: Resulting Quaternion Mean MAEs for the model which trained with all 3 days' data.

For the position, the MAEs could be found in the Table 12, whose unit is in meters.

| Day Number | Minute | MAE Pos X | MAE Pos Y | MAE Pos Z |
|---|---|---|---|---|
| 1 | 1 | 0.32185077 | 0.15752132 | 0.23592007 |
| 1 | 2 | 0.28327383 | 0.24320988 | 0.31408761 |
| 1 | 3 | 0.24034231 | 0.23204857 | 0.12944006 |
| 1 | 4 | 0.24209581 | 0.309488 | 0.13441538 |
| 1 | 5 | 0.23375713 | 0.15902535 | 0.0941516 |
| 3 | 1 | 0.32754326 | 0.07016635 | 0.06509805 |
| 3 | 2 | 0.21243546 | 0.16394965 | 0.05395144 |
| 3 | 3 | 0.21078694 | 0.27622665 | 0.10120114 |
| 3 | 4 | 0.30472198 | 0.38552358 | 0.17318112 |
| 3 | 5 | 0.26710652 | 0.1913575 | 0.15403332 |
| 4 | 1 | 0.12406341 | 0.25592646 | 0.09433397 |
| 4 | 2 | 0.08380599 | 0.07703061 | 0.11561494 |
| 4 | 3 | 0.09488921 | 0.16916791 | 0.15405664 |
| 4 | 4 | 0.13741003 | 0.14958843 | 0.12189978 |
| 4 | 5 | 0.14202988 | 0.17626911 | 0.20952724 |

Table 12: Resulting Position MAEs for the model which trained with all 3 days' data.

The mean MAEs within each day for the position is in the Table 13. Again the best results achieved is from the 4.th day.

| Day Number | MAE Pos X | MAE Pos Y | MAE Pos Z |
|---|---|---|---|
| 1 | 0.26426397 | 0.22025862 | 0.18160295 |
| 3 | 0.26451883 | 0.21744475 | 0.10949301 |
| 4 | 0.1164397 | 0.1655965 | 0.13908651 |

Table 13: Resulting Position mean MAEs for the model which trained with all 3 days' data.

As mentioned earlier, these tests are conducted with the data which was cut from the very end of 3 different recordings, and the model was trained with the data from 3 recordings. Also, the time differences among corresponding points from different devices for these recordings are not the same. Because of that, the model trained with the all of the records could perform worse than the model trained with one record.

To keep the training time lower, the data of the 4.th day is used during optimization of the neural network, since it showed the best synchronization characteristics. So, the 51 models trained during the Neural Network Design 6.2 could be tested to see whether using only one recording could increase the performance. In the Figure 30, the mean MAEs for the model trained only with

the data from the 4.th day could be seen.

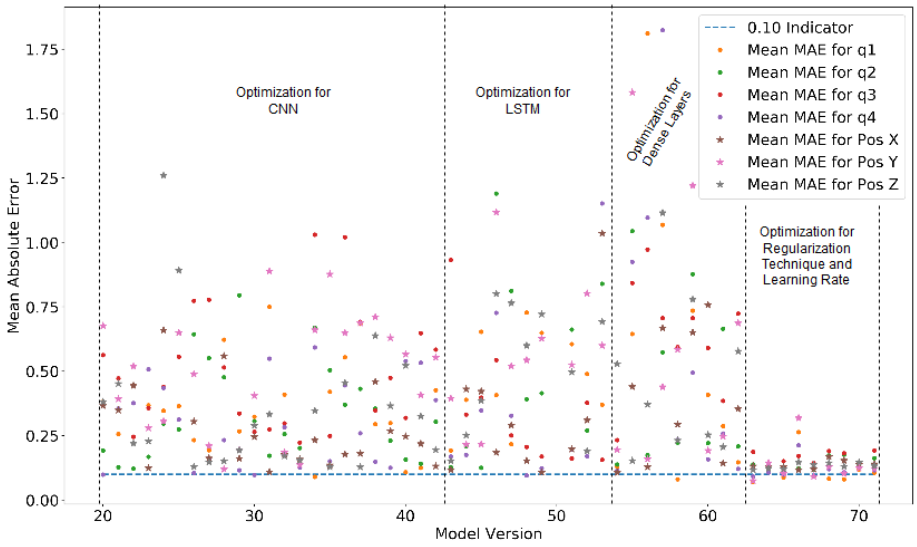



Figure 30: Mean MAEs for 51 Models Trained during 6.2 with only data from 4.th day. Note that the results for 5 minutes are averaged. Also, the model version 50 is a broken model so that no results are reported for it.

As seen in the figure, the last model versions, after x performs better than the others with similar errors on all values. Last 10 models' MAEs of the models in the Figure 30 could be seen in the Tables 14 and 15, for quaternion and position values respectively.

| Model Version | MAE Q1 | MAE Q2 | MAE Q3 | MAE Q4 |
|---|---|---|---|---|
| 62 | 0.14670297 | 0.20862615 | 0.72316642 | 0.12104922 |
| 63 | 0.07110358 | 0.13804635 | 0.18741067 | 0.09184004 |
| 64 | 0.11609026 | 0.11990964 | 0.13175562 | 0.10754768 |
| 65 | 0.08787334 | 0.12019154 | 0.15088795 | 0.09916175 |
| 66 | 0.26344772 | 0.12373371 | 0.17127091 | 0.21317618 |
| 67 | 0.09184921 | 0.12029804 | 0.14330653 | 0.09236645 |
| 68 | 0.08327096 | 0.17375036 | 0.19002712 | 0.12064425 |
| 69 | 0.08117492 | 0.17640153 | 0.18214565 | 0.10823182 |
| 70 | 0.11860842 | 0.14467068 | 0.13587602 | 0.13903005 |
| 71 | 0.10679877 | 0.16292754 | 0.19231899 | 0.11601716 |

Table 14: The Resulting Orientation MAEs for last 10 models in Figure 30.

When these are compared to the Tables 11 and 13, which was trained with

data from all 3 recordings, the results of the model version 63, 65 and 67 are better. But these models still cannot be used by the applications which require high precision.

| Model Version | MAE Pos X | MAE Pos Y | MAE Pos Z |
|---|---|---|---|
| 62 | 0.35449908 | 0.6874614 | 0.57736013 |
| 63 | 0.13229496 | 0.07595908 | 0.11826766 |
| 64 | 0.13009894 | 0.14485299 | 0.12379959 |
| 65 | 0.128669 | 0.10346915 | 0.12150998 |
| 66 | 0.11655888 | 0.31892743 | 0.14724584 |
| 67 | 0.12067505 | 0.0926576 | 0.13685473 |
| 68 | 0.16926609 | 0.12972437 | 0.14515569 |
| 69 | 0.15441113 | 0.10283495 | 0.13028951 |
| 70 | 0.13893357 | 0.1244476 | 0.14626394 |
| 71 | 0.14059806 | 0.12183204 | 0.13331454 |

Table 15: The Resulting Position MAEs for last 10 models in Figure 30.

The transition from 3 days' data to only 1 day's yielded a better result, which means transition from a more varying synchronization characteristics to less varying one helped the network at least a little. From this point, fully synchronous data could be tested to see whether it would improve the performance or not. With the fully synchronous data, the complexity of the network could be reduced, because the data would be of higher quality and the neural network does not need to deal with solving the synchronization characteristics of the data. So, with a simpler network, i.e. smaller layer size and/or less number of hidden layers, the same performance could be reached.

# 8 Conclusion

In this project, the main object was to determine the change in the pose using only the acceleration and orientation data from multiple IMUs. In this way, the IMUs will be fused to reach the target. Using neural networks would save us from sensor registration, calibration, error modelling, and coordinate system conversion.

To achieve this, first a comprehensive literature review is done. Although there are a lot of studies on sensor fusion, almost all of the approaches are analytical ones, where a number of manual work is required. Then the data is inspected for suitability for neural networks. It is found that the data is poorly synchronized, and to overcome it different preprocessing techniques are tested. Later, various configurations of neural networks have been tried, in order to fuse the sensory data more successfully. Here, it should be noted that there

are countless number of configurations for the neural network, where only a subset of it is tested. Even though the model configurations from this research cannot be used in the applications with high precision requirements, there is a strong possibility that the task would be achieved with neural networks using different network configurations and better synchronized data. Also, it should be noted that with this application, simple devices like IMUs would possibly have the potential of tracking, without the need for expensive visual tracking etc. setup.

At this point, it is important to know the nature of the task. There is no direct measurement of the position and the orientation, because the input data used is only acceleration and angular velocity. Therefore, the network will be predicting on top of its previous predictions, which means, at each step, error in the prediction will be accumulated for later use. To get a good output at a specific time, the network's error in the previous times should be very low, to avoid the error accumulation. As the time for the prediction gets longer, the final error will be higher. This would affect the application area of this solution, too. An example use case for this is to use the network as a backup in an application with visual or GPS tracking systems. In the case of non-reachable visual or GPS tracking, this application could be used for a small period of time.

From the experiments in this study, it is highly recommended for the data to be synchronized in the future studies. Also, there are still a lot of neural network configurations which needs to be experimented. For example, instead of having a window size of 4 seconds, narrower window sizes could be tried, along with different strides after each window(currently the stride is 10 data points). Maybe, a network without any window, which means a window size of 1 data points, hence, without any convolutional layers could perform better. Also, the initial pose is fed to the network after LSTM layers in the current design, but feeding the initial pose before the first LSTM layer is another possibility that could be investigated.

Another interesting point would be the comparison with analytical methods, using the same data from the same setup, so that the improvement of neural network usage could be better grasped.

# References


[1] Elmenreich, W. (2002). An introduction to sensor fusion. Austria: Vienna University Of Technology, (February), 1–28. https://doi.org/10.1073/pnas.1201800109

[2] Madgwick, S. O. H. (2010). An efficient orientation filter for inertial and inertial/magnetic sensor arrays. https://doi.org/10.1109/ICORR.2011.5975346

[3] Das, K., & Behera, R. N. (2017). A Survey on Machine Learning: Concept, Algorithms and Applications. International Journal of Innovative Research in Computer and Communication Engineering, 5(2). https://doi.org/10.15680/IJIRCCE.2017.0502001

[4] Monty, 'Python3 Codes', https://python3.codes/neural-network-python-part-1-sigmoid-function-gradient-descent-backpropagation. [accessed 22 November 2018]

[5] Kpkumar4u , 'Deep neural networks plays major role in finding obstacles near you', Trendy Techz, https://trendytechz.com/deep-neural-network-plays-major-role-finding-obstacles-near/. [accessed 22 November 2018]

[6] Deep Convolutional Neural Networks Projects and Research Topics, MTECH PROJECTS, http://www.mtechprojects.org/deep-convolutional-neural-networks-projects.html [accessed 22 November 2018]

[7] vImage Programming Guide, Documentation Archive https://developer.apple.com/library/archive/documentation/Performance/Conceptual/vImage/ConvolutionOperations/ConvolutionOperations.html [accessed 22 November 2018]

[8] Olah,Christopher "Understanding LSTM Networks", http://colah.github.io/posts/2015-08-Understanding-LSTMs/ [accessed 22 November 2018]

[9] Rawlings, E. L. H. and J. B. (2007). A Critical Evaluation of Extended Kalman Filtering and Moving Horizon Estimation. Development, 134(4), 635–646.

[10] Kalman, R. . (1992). A New Approach to Linear Filtering and Prediction Problems. Journal of Philosophical Logic, 21(2), 125–147. https://doi.org/10.1007/BF00248635

[11] Bergner, F. (1988). Sensor Fusion Algorithms for Robotics : Bayesian Inference vs . Cortical Circuits.

[12] Huadong Wu, Siegel, M., Stiefelhagen, R., & Jie Yang. (2002). Sensor fusion using Dempster-Shafer theory [for context-aware HCI]. IMTC/2002. Proceedings of the 19th IEEE Instrumentation and Measurement Technology Conference (IEEE Cat. No.00CH37276), 1(May), 7–12. https://doi.org/10.1109/IMTC.2002.1006807

[13] S. J. Julier and J. K. Uhlmann. A Ne w Extension of the Kalman Filter to Nonlinear Systems. In Pr oc. of Aer oSense: The 11th Int. Symp. on Aer ospace/Defence Sensing , Simulation and Contr ols. , 1997.

[14] Bancroft, J. B., & Lachapelle, G. (2011). Data fusion algorithms for multiple inertial measurement units. Sensors, 11(7), 6771–6798. https://doi.org/10.3390/s110706771

[15] Ahuja, S., Jirattigalachote, W., & Tosborvorn, A. (2011). Improving Accuracy of Inertial Measurement Units using Support Vector Regression. Cs229.Stanford.Edu.

[16] Sepp Hochreiter. (1997). Long Short Term Memory, 9(8), 1–32. https://doi.org/10.1162/neco.1997.9.8.1735

[17] Gers, F. A., Schmidhuber, J., & Cummins, F. (2000). Learning to forget: Continual prediction with LSTM. Neural Computation, 12(10), 2451–2471. https://doi.org/10.1162/089976600300015015

[18] Kyritsis, K., Diou, C., & Delopoulos, A. (2017). Food Intake Detection from Inertial Sensors Using LSTM Networks. Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 10590 LNCS, 411–418. https://doi.org/10.1007/978-3-319-70742-6_39

[19] Rambach, J. R., Tewari, A., Pagani, A., & Stricker, D. (2016). Learning to Fuse: A Deep Learning Approach to Visual-Inertial Camera Pose Estimation. In Proceedings of the 2016 IEEE International Symposium on Mixed and Augmented Reality, ISMAR 2016 (pp. 71–76). IEEE. https://doi.org/10.1109/ISMAR.2016.19

[20] Xsens, https://www.xsens.com/

[21] InertialSense , https://inertialsense.com/products/imu-sensors/

[22] VectorNav Technologies, https://www.vectornav.com/

[23] Advenced Realtime Tracking, ART, https://ar-tracking.com/

[24] UbiTrack and trackman, http://campar.in.tum.de/UbiTrack/WebHome

[25] Ordóñez, F. J., & Roggen, D. (2016). Deep convolutional and LSTM recurrent neural networks for multimodal wearable activity recognition. Sensors (Switzerland), 16(1). https://doi.org/10.3390/s16010115

[26] Tara N. Sainath, Oriol Vinyals, Andrew Senior, H. S. (2018). CONVOLUTIONAL, LONG SHORT-TERM MEMORY, FULLY CONNECTED DEEP NEURAL NETWORKS. Neurosurgery, 83(2), 166–179. https://doi.org/10.1093/neuros/nyx387

[27] Karpathy, A., Johnson, J., & Fei-Fei, L. (2015). Visualizing and Understanding Recurrent Networks, 1–12. https://doi.org/10.1007/978-3-319-10590-1_53

[28] Graves, A., Jaitly, N., & Mohamed, A. (2013). HYBRID SPEECH RECOGNITION WITH DEEP BIDIRECTIONAL LSTM, 273–278. https://doi.org/10.1111/j.1525-1314.1994.tb00042.x

[29] Kingma, D. P., & Ba, J. (2014). Adam: A Method for Stochastic Optimization, 1–15. Retrieved from http://arxiv.org/abs/1412.6980

[30] Simonyan, K., & Zisserman, A. (2014). Very Deep Convolutional Networks for Large-Scale Image Recognition, 1–14. https://doi.org/10.2146/ajhp170251

[31] He, K., Zhang, X., Ren, S., & Sun, J. (2015). Deep Residual Learning for Image Recognition. https://doi.org/10.1109/CVPR.2016.90

[32] Ioffe, S., & Szegedy, C. (2015). Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, 1134, 63–66. https://doi.org/10.1016/j.molstruc.2016.12.061

[33] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Rafal Jozefowicz, Yangqing Jia, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Mike Schuster, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[34] Ruder, S. (2016). An overview of gradient descent optimization algorithms, 1–14. Retrieved from http://arxiv.org/abs/1609.04747

[35] Kluyver, T., Ragan-kelley, B., & Pérez, F. (n.d.). Jupyter Notebooks — a publishing format for reproducible computational workflows.

[36] Keras, Chollet, Francois and others, 2015, https://keras.io