Bachelor Project

Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Control Engineering

Bicycle Transport Network Parameters Extraction Based on Mobile Phone Sensors

Jan Bednář

Supervisor: Ing. Drchal Jan, Ph.D.
Field of study: Cybernetics and Robotics
Subfield: Systems and Control
May 2016

Acknowledgements

I would like to thank my supervisor Ing. Jan Drchal, Ph.D. for his leadership, help and advices during the writing of this work.

I would like also to thank for access to computing and storage facilities owned by parties and projects contributing to the National Grid Infrastructure MetaCentrum, provided under the programme "Projects of Large Research, Development, and Innovations Infrastructures" (CESNET LM2015042), is greatly appreciated.

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Prague, 20. May 2016

.....

Abstract

The aim of this work is to develop a neural network model which could be used used for determination of a road surface using mobile sensors data. Mobile was attached to a bike to be able to measure vibration while cycling. I used measured data for a neural network training. I used mainly the *Python* language for the implementation of data processing and the neural network.

I have developed a neural network that can predict the surface with success rate of 81,7%. This model has following setting: 60 *LSTMs*, 200 sample sequence length, 50 ms resample.

The results of the work show up that the road surface can be predicted with usage of the mobile sensors. It can help to map surfaces in cities and the application can offer better roads options.

Keywords: Neural network, Surface detection, Android, Python, LSTM, PyBrain, Keras

Supervisor: Ing. Drchal Jan, Ph.D.

Abstrakt

Cílem této práce je vytvořit model neuronové sítě, který umožní identifikovat povrch (vybranou množinu povrchů) pomocí senzorů mobilního telefonu. Tento problém jsem řešil hlavně pomocí akcelerometru. Naměřená data jsem použil pro učení modelu neuronové sítě, který po naučení předpovídá projetý povrch.

Podařilo se mi vytvořit model, který předpovídá povrch s úspěšností 81,7%. Tento model je nastaven na: 60 *LSTMs*, 200 záznamů na sekvenci, 50 ms vzorkovací frekvence.

Výsledky této práce ukazují, že je možné pomocí mobilních senzorů s relativně dobrou úspěšností analyzovat povrch, po kterém cyklista jede. To může pomoci ke zmapování povrchů, po kterých cyklisté ve městě jezdí a zlepšit tak rozmanitost nabízených variant trasy.

Klíčová slova: Neuronová síť, Detekce povrchu, Android, Python, LSTM, PyBrain, Keras

Překlad názvu: Extrakce parametrů cyklodopravní sítě z dat ze senzorů mobilního telefonu

Contents

Figures

Tables

Chapter 1

Introduction

Nowadays smartphones and modern technology are moving the world. Thanks to these smart devices and using their oportunities with applications we could make our life easier. These devices contain wide range of various sensors that could be used for completely different purposes as both measuring life functions and measuring surroudings. Using smartphones while doing daily work as driving, shopping or training is becoming a part of modern lifestyle. I can find various Android applications for navigating and tracking cyclist route that can offer both GPS tracking and for example cyclist energy expenditure. It is important to create an Android application, that can offer something new to the cyclists. Having the Android application that can determine a surface from mobile sensors is good advantage. Cyclists could choose their route according to the surface preferences on the way.

The aim of this work is to create a neural network model which could be used to determine a surface from mobile sensors data. Neural network predicts surface according to learned dependecies between measured data and given surfaces. The creation of the model requires data measurement during cycling, data preparation for training and then using the data to train neural network. To be able to measure some data I have to create an application that will measure data on a phone and save them. Data has to be prepared before being used for training. Then I can train models with different parameters and find the best solution.

Chapter 2
Phone Sensors

2.1 Overview

Android devices have several different sensors for measuring orientation, motion, environmental conditions, etc. These sensors can be cleverly used in lots of applications. For our purpose we chose the position and the orientation as the best ones.

Most of the sensors could be used only as a support for the basic ones. From the list below, the main I will use, is an accelerometer. A GPS performs principally location provider and an audio is for my purposes used as a support but we can assume it to be a sensor.[6]

2.1.1 Accelerometer

An accelerometer is one of the most used sensor in general. It is really cheap and easy to use. Most of the devices contains the accelerometer. Its function can be described as measuring the acceleration force in ms^{-2} that is applied to the device on all axes *(x, y, z)*. This information can be used to get a phone tilt in axes or to measure shaking with a phone, etc.

I can describe an accelerometr coordinating system with a default phone orientation showed in the picture 2.1. The default orientation of the device is *(0,0,0)*. It means that the x axis is horizontal and points to the right, the y axis is vertical and points up and the z axis points toward the outside of the screen face. Accelerometer values also include the gravity force that is applied in axes according to the phone orientation. If the phone lies freely on straight ground, we can measure approximately 9,81 ms^{-2} value on the z axis. I will focus on accelerometer settings in the Android at 3.1.1.

An accelerometer is very sensitive to any movements. It is almost impossible to hold it without detecting any change in axes. Great advantage is that an accelerometr allows to measure fast-frequent vibration while the phone is attached to a bike.[7]

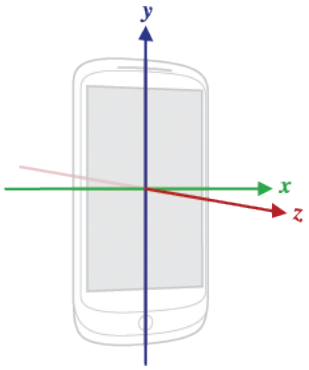

Figure 2.1: Coordinate system (relative to a device) that's used by the Sensor API [1]

2.1.2 Gyroscope

A gyroscope is used to measure the tilt and the position of a device just like the accelerometer, but the difference is in measured quantity. A gyroscope gives data in $rads^{-1}$ so the angular velocity is measured.[7]

But this sensor is quite expensive and devices with the gyroscope are much more expensive. This is one of the reasons why the gyroscope can't be a basic sensor for my task. Second reason is that the phone position (axes) on a bike is mostly changed along the axes so the gyroscope can't notice almost any change in value.

2.1.3 GPS

A GPS can measure the longtitude and the latitude of the device that can be used to locate the device, track the route, offer a route option, etc. I used it for tracking my routes (measurements) to be able to reconstruct the route. It doesn't have any special impact on the surface detection.

2.1.4 Audio

I have used audio records to record surroudings of the cycled route, but mainly for my own notes during the cycling, because the GPS is often not so accurate. Records may be used for detecting traffic intesity.

2.2 Conclusion

In summary the accelerometer appears to be the most suitable sensor for surface recognition. Both the accelerometer availability and given values are necessary for basic detection of surfaces. The gyroscope could be used as a support sensor for the accelerometer, but the problem is its availability in phones. There are some other sensors in mobile phones that might be usefull, but they occur very rarely. It is possible to do more specialized research about phone sensors to find out if there is any added value to this task.

Chapter 3

Android application

3.1 Description

To be able to do measurements I firstly had to create an Android application that records data from chosen sensors. The application is targeted on the Android 6.0 Marshmallow (SDK 23), but minimal running Android version is the 4.1 JellyBean (SDK 16). Below you can see the main screen of the application.

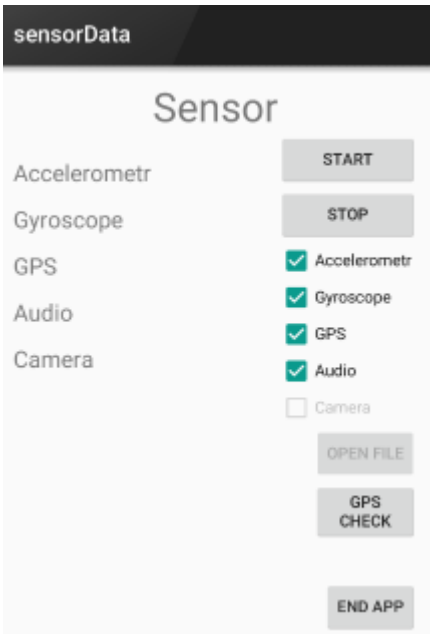

Figure 3.1: Application main window

3.1.1 Implementation

Accelerometer and Gyroscope

Android application development tools allows an access to sensors through the class *SensorManager*. This class allows to create an instance of a connection to chosen sensor. Firstly, it is recommended to check if the phone is equipped with the chosen sensor. For this purpose you can use the *PackageManager* class.

I have used the *SensorEventListener* to get the values from sensors. Instance of the listener has to implement two methods: *onAccuracyChanged()*, *onSensorChanged()*. In the method *onSensorChanged()* can be specified what happens if the sensor data changed. In the program can be created just one instance of *SensorEventListener*. Inside the method has to be specified the logic decision which sensor is calling method *onSensorChanged()*. The *SensorEventListener* has to be registered with relevant sensor to be able to listen to changes of sensor values.

While registering the sensor you can specify the delay between samples of recording data. I set this value to zero to get the data as fast as possible. Each phone has different sensor so it will give values on the average every 5 miliseconds. Firstly, I had problem with the delay between samples. At the beginning, I didn't set the delay value to zero and it was default set to 200 miliseconds which is too much.

GPS

A GPS tracking could be accessed with the instance of *LocationManager* class. Difference between sensors and location providers are that location providers, such a GPS, has to be turned on to use them.

Then a *onLocationChanged()* listener has to be implemented. The listener will be called when the GPS coordinates change.

The listener has to be added to the *requestLocationUpdate()* list of the *LocationManager* instance.

Audio

An audio is set with the *MediaReccorder* class. You have to create an instance of the class and then specify the save format and the encoding. There is no other special requirements for audio recording. More about audio records in the Android at [8].

3.1.2 Application Settings

On the top-right side of the screen there are two buttons to start and stop measuring.

Below there is a list of checkboxes to choose sensors that will be recorded. The camera is disabled because I have decided not to use it due to small

benefit.

In the bottom of the screen there are three buttons. The first one should open a directory where the data are saved, but it hasn't been implemented yet so it is unavailable. The second one checks if the GPS is enabled. The last button switches the application off.

3.1.3 Real time information

On the left side of the screen there is information about the real data obtained from sensors. Each box contains a time stamp, and a value from each axis. In case of the audio, there is information about the status (on/off) of chosen device.

3.2 How to save data?

Measured data are saved on a phone into a folder:

storage:\\Cycleplanner\month-day\hour-minute-second

The folder path could be for example:

storage:\\Cycleplanner\3-22\17-33-02

The application saves data from the sensors and the GPS in *.csv* file. Name of the file in the folder is *Accelerometer.csv* and *Gyroscope.csv* The format of each variant will be displayed below.

The audio record is saved in *.3gp* format. Name of the file in the folder is *Audio.3gp*. It is common format for mobile phones recording.

3.2.1 Sensors

Accelerometer

Time Stamp	X	Y	Z
2016_03_22_17_33_02_403	-0.051076304	0.03405094	0.017025948
2016_03_22_17_33_02_421	0.1021526	-0.11917782	0.017025948
2016_03_22_17_33_02_422	0.1021526	0.03405094	0.017025948
2016_03_22_17_33_02_449	-0.051076304	0.03405094	0.017025948

Table 3.1: Example of the accelerometer CSV file

Each row contains information about the time stamp and the actual values from the sensor. In case of the accelerometer values contains information about the acceleration in given axis. Data displayed in the table are raw. Data has to be processed before using. More about the sensor data processing in the chapter 5.1.3

Gyroscope

The gyroscope format is the same as the accelerometer. Just the values are different.

3.2.2 GPS

Time Stamp	Latitude	Longitude
2016_05_08_14_00_23_719	50.09209919	14.10519621
2016_05_08_14_00_24_458	50.09209978	14.10519448
2016_05_08_14_00_25_460	50.09209981	14.1051928

Table 3.2: Example of GPS file

Each row contains information about phone position in time. The GPS gives information almost every second. More about the GPS data processing at 5.1.3.

3.3 Conclusion

I have created the Android application for data measurement. This application isn't the main target of this work. I have studied the Android operating system, especially how to implement loading and saving the data for the sensors (*SensorManager* class), the GPS (*LocationManager* class) and the audio (*MediaRecorder* class). The application is just an example of recording/measuring and saving the data and could be modified and improved as required.

Chapter 4

Data Measurement

4.1 Description

Next important step is the practical measurement itself. I have cycled several cycleroutes, asphalt streets, cities pavements, brick streets and countryside roads. One measurement took about 2 hours.

I chose two phones for measurement. First was a "Vodafone Smart Prime 6". This phone has just the accelerometer so I had to find another for measuring with the gyroscope also. I used a mobile phone "LG G3".

4.2 Used sensors

I have performed measuring almost exclusively with the accelerometer. I have decided to use just one sensor because my methods haven't been tuned and I wanted to measure using the best method. At the end I decided to use also the gyroscope for measuring in order to pursue certain tests with the both sensors together.

4.3 Challenges

4.3.1 Phone holder

I had to choose a holder for the phone. I bought the holder "CONNECT IT M7" that simply allows to attach the phone to the bike. I had to solve problems with slipping of the holder which caused few errors in my data. The holder often slipped on the handlebar and zero value of axes changed. More about this problem in the chapter 4.3.2.

4.3.2 Phone holder placement

The phone position on the bike is essential for measurement. The phone could be placed both on the handlebar and a top tube of an axle. On the handlebar the phone could be placed both closer and further to the fulcrum which can add error during measurement caused by moves with the handlebar

by a biker.

I have had only the phone holder that could be attached only to the handlebar, so I tried to eliminate the cyclist error by mounting the holder as close as possible to the fulcrum. If the holder is moved from the default position during measurement, it could slightly affect the neural network learning.

I have decided to keep the holder in the fixed position on a bike. To eliminate the gravity force of the default position, I have made the average of 10 first values from the sensor and the algorithm constantly subtracts this average from following sensor values. It means that the algorithm will save just the differences from the average value.

4.3.3 Choice of surfaces

I have chosen following set of surfaces: *Asphalt*, *Bricks*, *Pavement* and *Countryside*. The *Asphalt* do not include only roads, but also pavement covered with asphalt. The *Pavement* means mainly interlocking pavement. The *Bricks* means cobblestones. The *Countryside* means bike roads. Set of surfaces is able to be extended for another ones.

4.3.4 Speed of measurement

It is important to remember, that the cyclist speed has significant influence to vibration measured with the phone. If the cyclist's speed is firstly 15 kmh^{-1} and then is increased up to 40 kmh^{-1} vibration will be recorded with approximately . It could be possible to find out how the speed of measuring influences neural network learning. However, I didn't consider this factor in this work and tried to measur data with average speed 15 kmh^{-1}.

4.4 Conclusion

Measuring is very tricky part of the whole work. If the data are not measured properly, it is impossible to create a functional model of a neural network. It is important to do a lot of measurements on different surfaces to gather a lot of objective data. I have tried to measure huge amount of data, but because of mistakes in the implementation and phone position fixing (4.3.2) it wasn't easy. It is possible to try much more positions of the phone on a bike to find out which is the best one for measurement.

As I mentioned in 4.3.3 the list of surfaces could be modified. For example the *Bricks* surface is probably the easiest one to recognize, but it is hard to find this surface in the reality.

Despite all kinds of problems and posibilites I measured enough usable data that will be used for neural network learning.

Chapter 5

Data Preparation

5.1 Overview

Next necessary processing step is to prepare the data for neural network learning. I have to load measured data to filter the useless ones, annotate the rest and save them for neural network learning.

I have used the *Python* as a programming language mainly for two reasons. Firstly, it is easy to code and read. Secondly, there are various libraries for both data preparation and network learning.

5.1.1 Loading data

I used the *Pandas* library for data loading. It simply allows to read *.csv* files with *read_csv()* method and prepares dataset that can be later easily used for the next steps. I also had to reformat the time stamp using method *to_datetime()* applied to chosen column.

At the beginning, I had problems with resampling, because of the format of miliseconds. The value of miliseconds smaller than 100, i.e. 0-99 was resampled into 000-990, i.e. one zero was added to the end. This error created mistakes in the dataset so it had to be removed. I changed the Android application to save miliseconds always with 3 digits.

I used the *Matplotlib* library for displaying measured data. It allows to display the data in a graph similar to graphs in the *Matlab*. Here is an example of graph with measured data.

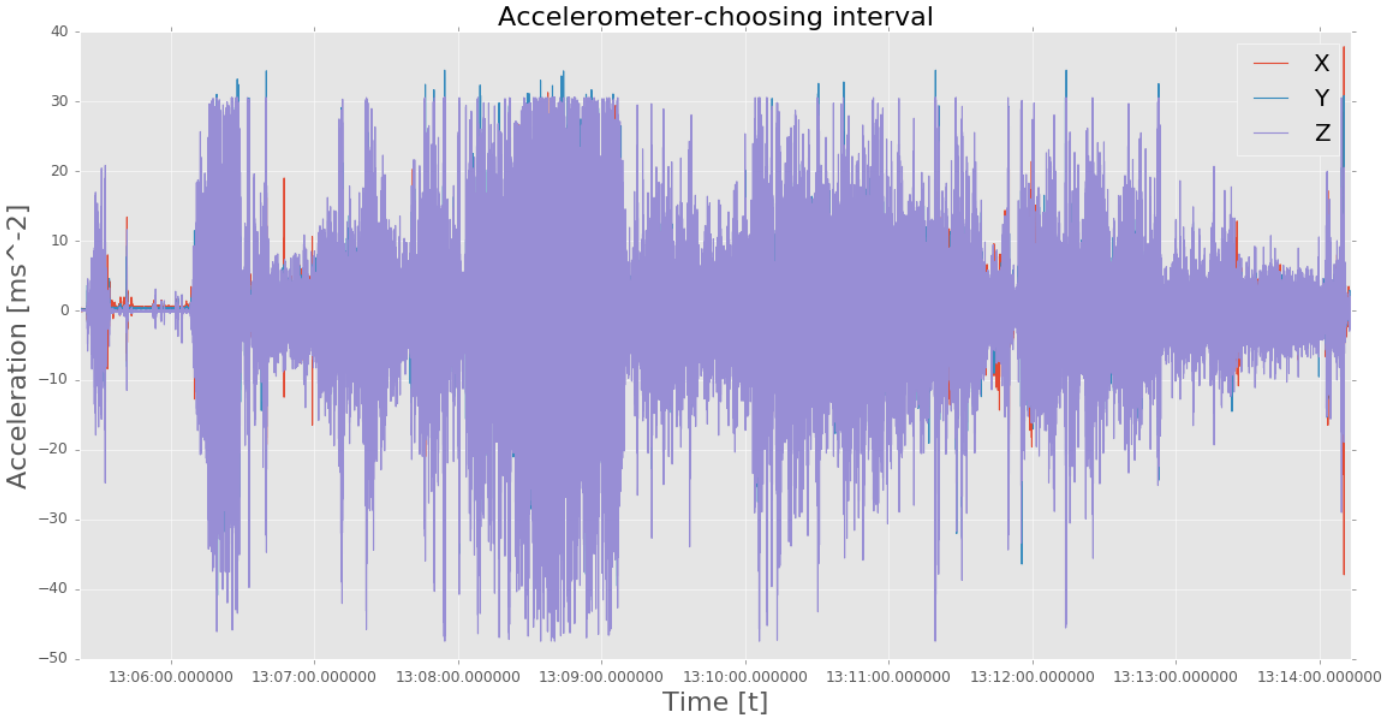

Figure 5.1: Displayed measured data with *Matplotlib* library

5.1.2 Annotation

To be able to learn neural network with the data, I have had to annotate measured data. That means to add an information to data about the surface from measurement. I solved this issue with mouse listeners that the *Matplotlib* library offers.

At the beginning, I did resampling right after annotation. Later I figured out that I needed different resampling time but it was impossible to change that because the original time information was lost. This is why I have decided to do resampling just right before training.

Interval mode

I added a listener to the graph, that reads a time stamp information from the position in the graph where user has clicked. On the clicked position there is created a blue line, that devides the data into two intervals. You can place any quantity of intervals you require. If user wants to exit interval settings, he has to click on left mouse button.

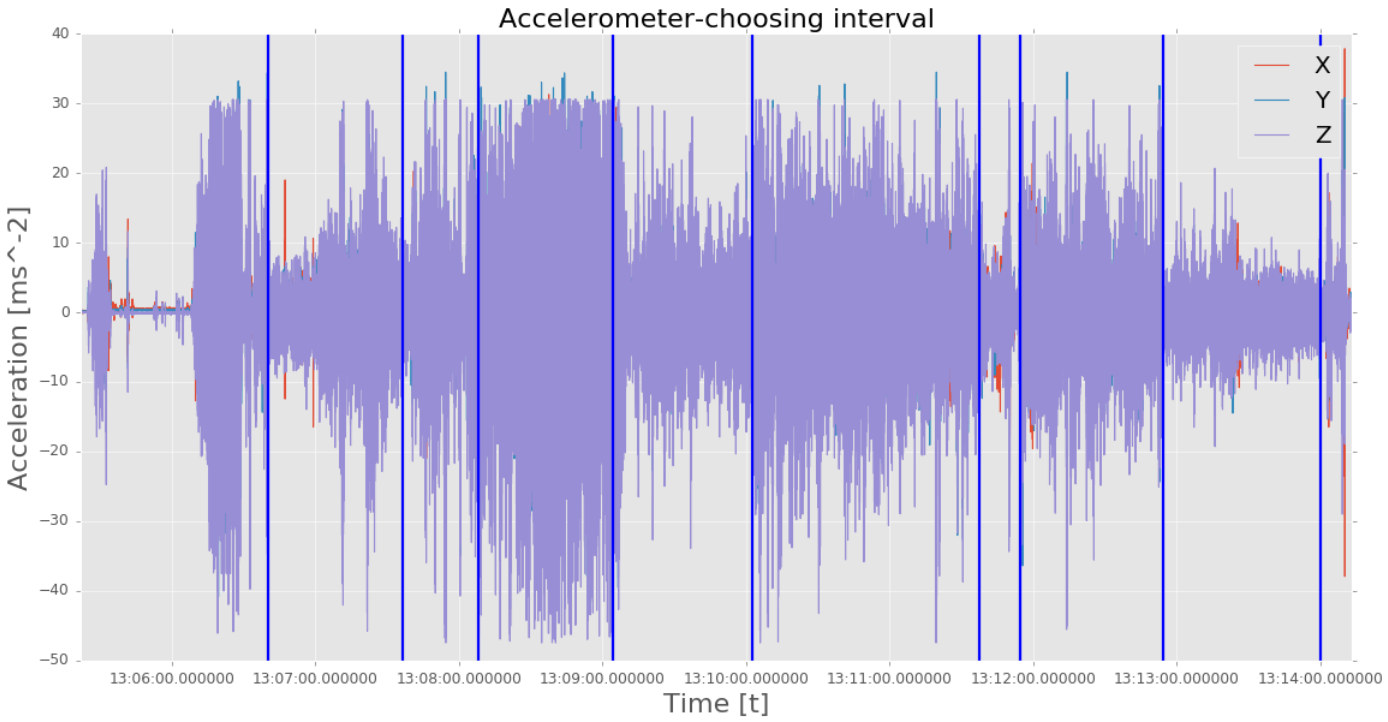

Figure 5.2: Created intervals

Surface mode

In this mode user assigns surface information to created intervals with each next right mouse click. The surface description is changed with every right click. Text is changed in given order: *Asphalt*, *Bricks*, *Pavement*, *Countryside*, *Mess*. Text is always displayed in the data interval which has the user clicked in. If the user wants to exit the surface mode and save intervals, he has to click on left mouse button. After the click new data will be loaded and displayed in a new graph or if it is a last data file, algorithm finishes.

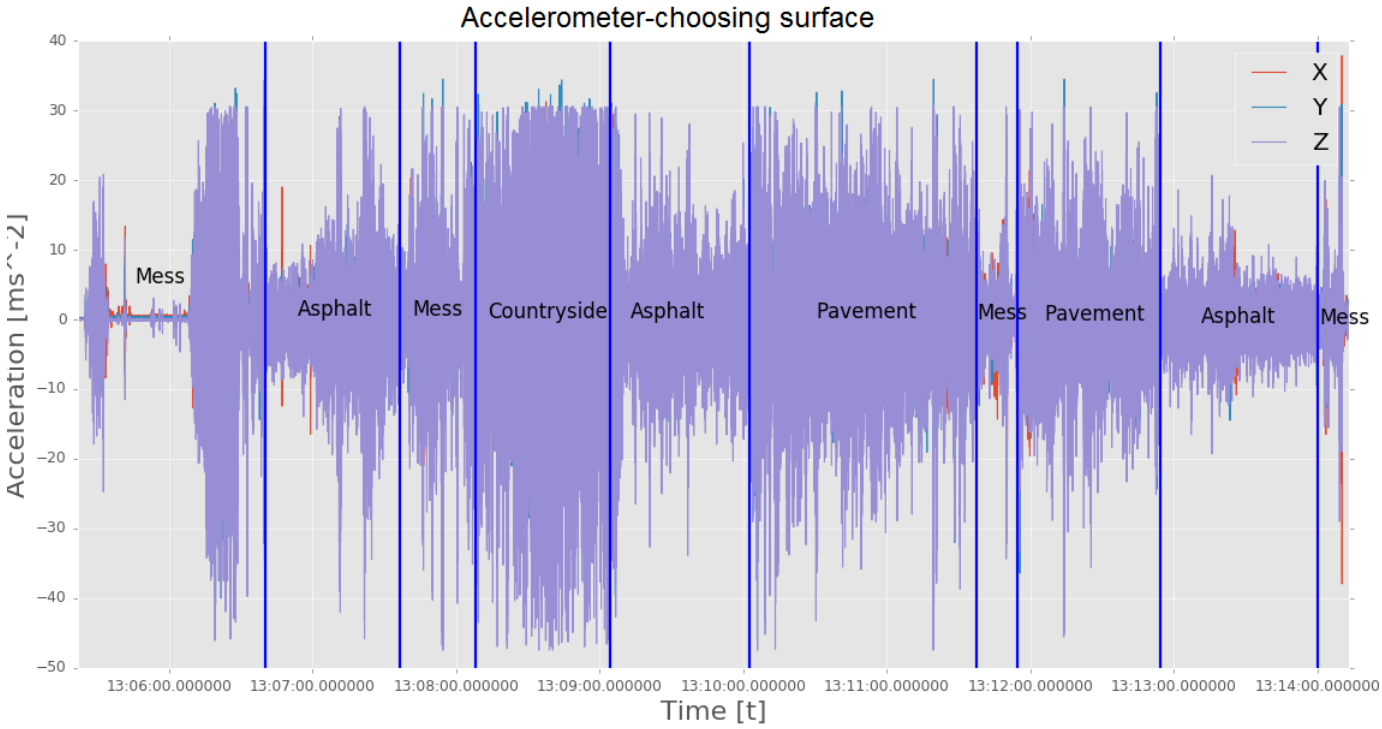

Figure 5.3: Added surfaces to intervals

Mess surface

I have created a new surface called *Mess*, which won't be used for training, but provides me with a space where I annotate all data I have acquired. I stopped cycling several times during measurement in order to have a rest for a moment. These intervals are useless but they are included in measured data.

5.1.3 Data save format

Accelerometer

After annotation, during which I also erased useless data, I added one column to the table with number that indicates surface of the interval. The *Asphalt* is indicated by zero, the *Bricks* is indicated by one, the *Pavement* is indicated by two, the *Countryside* is indicated by three and the *Mess* is indicated by four. As an example I show the table with the new column.

Time Stamp	X	Y	Z	Surface
2016-03-22 17:33:40.710	-4.341486	-2.724069	1.396085	0
2016-03-22 17:33:40.726	-2.655968	7.082580	-5.958901	0
2016-03-22 17:33:40.729	-3.422112	6.009979	-13.467118	0
2016-03-22 17:33:40.745	-1.123678	-0.732093	7.984928	0
2016-03-22 17:33:40.761	-2.502739	3.405087	-3.966926	0

Table 5.1: Example of Accelerometer CSV file

Annotated data are saved in a new *.csv* file that contains only one interval and name of this file is created with this pattern:

"DD-MM_hh-mm-ss_Surface_IntervalNumber.csv"

where:
DD - day of measurement
MM - month of measurement
hh - starting hour of measurement
mm - starting minute of measurement
ss - starting second of measurement
Surface - surface of interval
IntervalNumber - number assigned to the interval from original data set

Gyroscope

All procedures are identical with those described on accelerometer.

GPS

The GPS data aren't used for training, but I have used them to display route in the program *Google Earth*. To display the data in *Google Earth* they must be transformed into different format, I chose *.kml*. I have used [9] for transformation the data from *.csv* to *.klm* format.

5.2 Conclusion

In the data preparation I have filtered the useless data and have annotated the usefull samples. I have studied the *Python* language, development tools and libraries, for example *Pandas* and *Matplotlib*.

A human is able to distinguish the surface pattern just by a look at the graph. It means that if I can find the surface data pattern just by looking at the graph, the neural network model can learn to recognize each surface. This is key information because if there isn't any visible difference between the surface data pattern, the neural network model might have a problem with surface recognition.

Chapter 6

Neural Networks

6.1 Basic Overview

In informatics the neural network is a model that is inspired by the biological neural network such as the brain. It is provided with units called neurons similar as biological neurons. Each neuron can have more inputs, but has just one output. Neurons are connected with connections, that have some values. Generally these values (weights) describe how much important the connection between connected neurons is. All connection weights can be tuned according to success of leraning. It makes neural network adaptive to inputs and capable of learning.[10]

The type of neural network that we will be interested in is the recurrent neural network alias RNN. This network is different from the classic *Feedforward neural network*[11] with connections to previous neurons layer. These backwards connections create an internal state of the network. RNNs can use this state to process sequences of inputs dependent on time. Firstly, I'm going to explain the general artificial neuron, recurrent networks and then the used neuron [12].

6.1.1 Artificial neuron

There are several types of neurons. I will explain basic idea on a neuron unit called the perceptron created by Warren McCulloh and Walter Pitts.

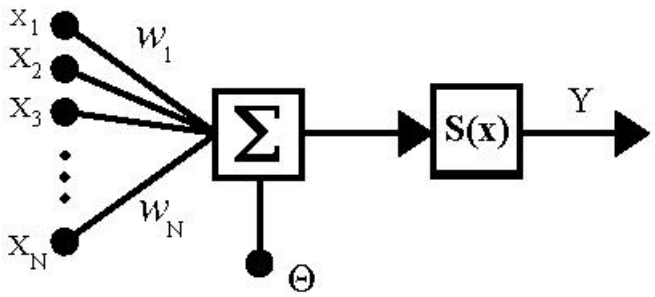

Figure 6.1: Perceptron model [2]

This model can be described with following equations:

$$Y = S(x)$$

$$x = \sum_{i=1}^{N}(w_i x_i) + \Theta$$

where:
x_i - neuron inputs
w_i - weights of connections
Θ - a treshold value
$S(x)$ - neuron activation function
Y - neuron output

In this model inputs x_i are mutiplied by weights w_i. If summary of mutiplied values reaches the threshold value Θ, the summary goes throug the activation function to output that leads to another neuron. [13]

6.1.2 Activation function

The activation function "activates" an input or an output to normalize value. Function could be discrete or continous in depending on the data. I will use just continous functions because I will work with continous flow of data. I will use especially the *sigmoid* and the *tanh*.

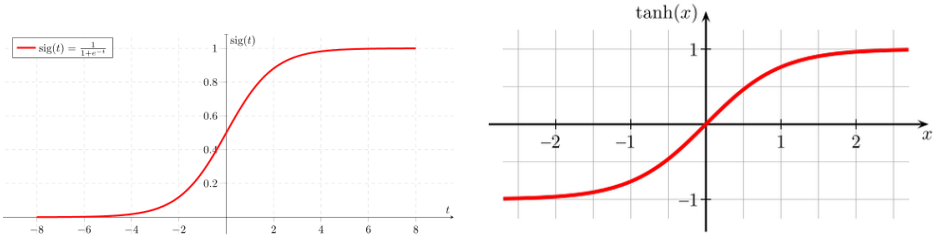

Figure 6.2: Activation functions for neural networks. Left - sigmoid, Right - tanh

These functions are useful to normalize values in the interval **(0,1)** or **(-1,1)**. I will describe their usage more in chapter 6.2.3.

6.1.3 Loss function

The loss function is type of a function which maps values of one or more variables onto a real number representing some "cost" associated with the value. This "cost" should be depreciated in order to reach the best results. There are a lot of loss functions that can be used for different tasks. I will focus on loss function in chapters 7.1.5, 7.1.7 and 7.1.8

6.2 Recurrent Networks

The difference between the *FeedForward* model/neuron and the recurrent one is in the backward connection. This connection allows to remember information which is dependent on time or is saved in time sequencies. You can imagine that, for example, with sentences. Humans usually try to predict next words, even if they do it unintentionally. It is natural, when you heard some sequences of words your brain tries to guess next words. I'm goint to show an example of the neuron below however that is just one variant of a neuron. It can be much more complicated. [3]

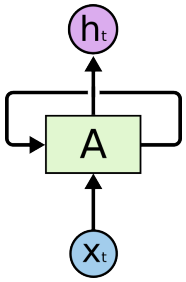

Figure 6.3: Recurrent neuron model [3]

Obstacle that appears with using simple RNNs is called vanishing gradient problem. Firstly I have to explain what is the term backpropagation and then I can explain the problem [14].

6.2.1 Backpropagation

A backpropagation is a type of an algorithm used in order to train a neural network. It uses a gradient descent as an optimization algorithm. More about the gradient descent alogithms see in[15]. The algorithm calculates the gradient of the loss function with respect to all the weights in the network. The gradient are then used to adjust weights up or down according to direction of the error.[4][16][17]

Obstacle that appears with using simple RNNs is called a Vanishing Gradient Problem.

6.2.2 Vanishing Gradient Problem

The backropagation algorithm calculates the gradient from the end to the beginning of the model. The gradient is a rate which costs changes with the respect to weights and the bias. The algorithm uses the chain rule to calculate earlier gradients which is just multiplication of gradients that were calculated before. Usually the *tanh* is used as an activation function therefore the result of the gradient will be between 0 and 1. The gradient is calculated for each neuron in the model. Then all gradients are mutiplied together to get the total difference between the output and the input. If you mutiply several numbers between 0 and 1, you are going to get a number really close

to the zero. Gradient value is then much smaller in the earlier layers. As a result, the earlier neurons are hard to train, or almost impossible to train. It is a big problem. If your first neurons can't detect properly, all other neurons in model will be effected.

Simple solution for solving the vanishing gradient problem is the *LSTM*. In the *LSTM* the activation function is the identity function with a derivative of 1.0. The backpropagated gradient neither vanishes nor explodes when passing through, but remains constant.

6.2.3 LSTM

We can assume that LSTM unit is a neural network too, because it has its own mechanism to control data flow. I can describe the *LSTM* unit with picture easier.

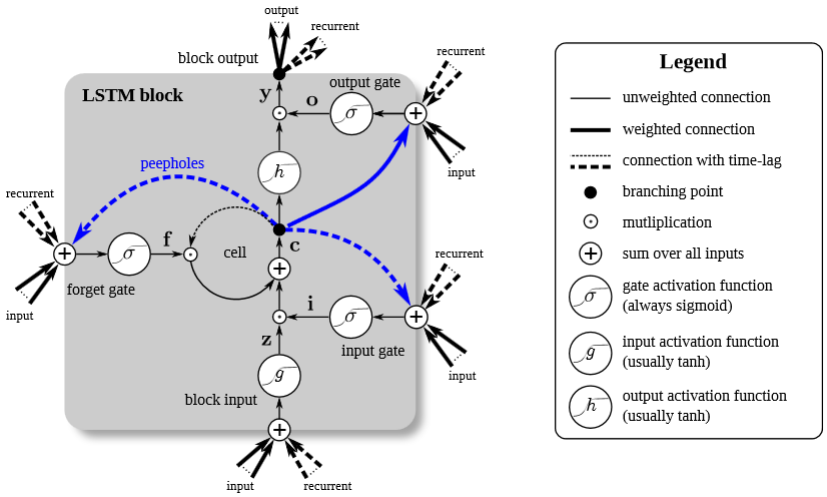

Figure 6.4: LSTM model description [4]

LSTM consists of 4 main parts. **An input gate, a forget gate, a cell** and **an output gate**. All inputs in one *LSTM* unit contain data from previous state to be able to find relations between seqeunces of data. All input data are the same, but they affect different modules as described below.

Input gate

Data that appears in the *input gate* are activated with the *sigmoid* function that decides "how much" data from the *block input* will let in. Simply it just muptiply the input with value in the interval *(0,1)*.

Forget gate

The *forget gate* decides when the cell memory should be changed. According to the result f of the sigmoid activation function the result will be mutiplied with the state of the memory *cell*, that contains information from the previous state. If f is 0, the memory is erased, if f is 1, the memory value remains the same.

Cell

Function of the *cell* is just to keep the value. The input value is provided

by a sum of the input gate result and the *forget gate* result. The output of the *cell* leads to the *tanh* activation function and after is mutiplied by the result from the *output gate*.

Output gate

According to the state of the variable o that is value between *(0,1)* after activation from activation function *sigmoid* in the *output gate*. This value mutiplies an output from the *cell*. In other words the *output gate* decides, how much of the *cell* value will go out [18].

6.3 Conclusion

In this section I have explained basics about neural networks. I have described a basic artificial neuron known as a perceptron, what is an activation function, its examples and basics about a loss function. I explained why I'm using the recurrent networks especially the *LSTM*.

Chapter 7

Network Learning

7.1 Implementation

An implementation of sequential datasets is necessary property of a library that I'm going to use. Sequential datasets allow to divide loaded data into fixed-length sequences of samples that contains only a one surface. It is important having data in the one sequence from raw measured data without shuffling. It means that the order of samples in sequence has to be the same as in raw measured data. Then samples keep information about the surface.

In next section I will explain separately implementation in different libraries. Firstly I will show common settings. At the end, there will be a comparison at testing phase.

7.1.1 Loading

Loading and preparing required data into the dataset is the first step of the implementation. I load data as described in section 5.1.1. Only difference compared to the previous method is that there is one more column with the surface information.

7.1.2 Resampling

Resampling is very important step. Samples that I have obtained from measuring was recorded with a frequency "as fast as possible" as mentioned in 3.1.1. It means that the time interval between the samples is not constant but I require that. This is the reason why I have to do resampling. But how to choose the right resampling time? If intervals between data records are too large, I will lost information about the surface and then learning will be complicated. If an interval is too small, the model could mix up surfaces. The algorithm takes data 5.1 and resample the time stamp with chosen freqeuncy.

I will present an example with 25 ms resampling.

Time Stamp	X	Y	Z	Surface
2016-03-22 17:33:40.700	-4.341486	-2.724069	1.396085	0
2016-03-22 17:33:40.725	-2.400586	4.120156	-3.813697	0
2016-03-22 17:33:40.750	-2.272896	1.949412	2.928374	0

Table 7.1: Example of resampling

Data are now resampled with 25 ms spacing between records. Axes values are now averages of values between time stamps from original data. This prevents from having values NaN (Not a Number).

7.1.3 Dataset

A format of raw data is called the dataset. Creating a dataset is the necessary step. Libraries are using different types of a dataset. There are libraries with their own system of dataset creation, which can be easily prepared, but can't be directly transfered for another library. It is for example the *PyBrain*. More about the *PyBrain* dataset in section 7.1.7.

Second group uses an external library to create the dataset. This dataset can be easier transfered to other libraries. It is for example the *Keras*. More about the *Keras* dataset in 7.1.8.

It is necessary to prepare training and validation data for a model. It is done by splitting whole dataset into two parts. Training and validation data are usually in ratio of 0.8:0.2.

The **training data** are those which the model uses for training in each epoch. The loss function and the accuracy is used to tune model neuron weights and parameters.

The **validation data** are those which the model uses to check quality of the trained model. The validation data loss and the accuracy are usually used to control the end of the learning process.

7.1.4 Model

Now I can create a neural network model containing the *LSTM*.

The first step is to decide how many neurons will the model have. You can design various models with various layers that can affect learning in a good or a bad way.

If you put too much of neurons, your model could be overfitted during training. This means that the model is fitted directly to the training data and if you validate the model with new data on the trained model, results will be bad. This problem could be partially solved by changing the neurons quantity (usually makes the model simplier) or doing a *cross-validation*. I will describe the *cross-validation* in section 7.1.5.

7.1.5 Training methods

Firstly I have to choose an appropriate loss function and an optimizer. This can also significantly influent the learning process.

Then a choice of learning method follows. In summary you have 2 basic methods. A "training until epoch" or a "training until convergence" (train until minimal loss or accuracy is achieved). These two methods can be used in the *cross-validation* method or just alone.

The **training until epoch** just trains the model for given number of epochs without respect to any other parameters. It is a basic method but not much efficient. For example, you can easily overfit your model.

The **training until convergence** is much more suitable method. It monitors chosen value while training and tries to find the minimal/maximal value of the monitored parameter. If you monitor the loss, you require the minimal value. With the accuracy it is vice versa. Using this method you can usually find the best solution for given data.

The **Cross-validation** is method that splits the given dataset into K parts. The algorithm runs for K steps and in each step of training it validates with the *Kth* part of the validation set and trains with the rest. If you calculate the loss for each training, you can get the total loss over chosen dataset and thus eliminate influence of data error. Here is an example of the *cross-validation*

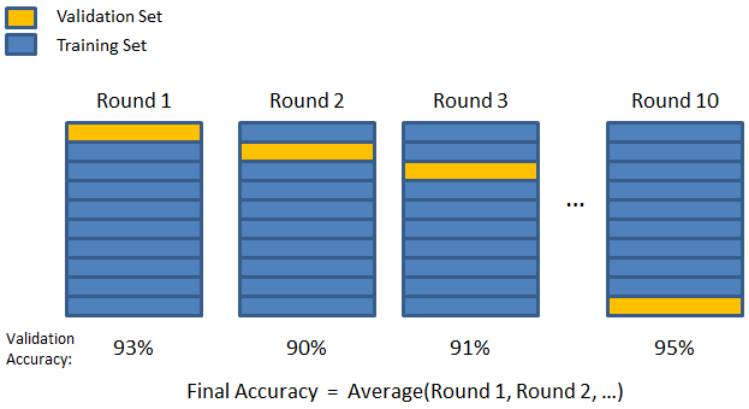

Figure 7.1: *Cross-validation* [5]

7.1.6 Result

Calculating of the training success is the last step of learning process. Trainers usually provide results during the training, but it is important to get the result at the end. A **validation accuracy** or a **validation loss** is probably a reasonable parameter for watching. It shows how well you trained the model. I will use the validation accuracy.

I will also use a **confusion matrix** to display success of training. More about the *confusion-matrix* see in [19]. The *confusion matrix* could be used for calculating derivatives like a **F1 score**. The *F1 score* is weighted average

of the true positive rate and the precision. It provides good overview of the training success rate.

7.1.7 PyBrain

This library allows to implement a neural network. It is easy to start with and more suitable for beginners. However nowadays the library is not actively updated, and not commented well.

The slow speed of calucation while using *LSTMs* is its biggest disadvantage. The library isn't optimized enough for multithreading and working with models using *LSTMs*. Models with *LSTMs* have lot of parameters and variables that can be tuned while training. Time of the final training of one model could be around couple of weeks. It is necessary to experiment with various combinations of loss functions and optimizers which causes that training with the *PyBrain* can be lengthy. Also the *PyBrain* has just limited amount of loss functions and optimizers. You can find more about the *PyBrain* in [20].

Dataset

The *PyBrain* offers 4 options of datasets. I will use the **SequenceClassificationDataSet** which combines sequences of samples and the classification into classes. The dataset is created internally by function *appendLinked()* which accepts the input data and the target output. The *PyBrain* uses internally the *NumPy* library to create the dataset. Sequences are created by calling the *newSequence()* method which ends a sequence and starts a new one. The *PyBrain* allows to have sequences which don't have the same length.

Model

Firstly I have to specify parameters for the *buildNetwork* function. This function will create a model that will be trained.

buildNetwork(indim, numneurons, outdim, hiddenclass, outclass, recurrent)

where:
indim - input dimension into the neural network which is the size (length) of the one sample in the dataset.
numneurons - the number of neurons in the model. This model will contain only one fully connected layer.
outdim - output dimension from the model which has to be the same as the output dimension of the dataset.
hiddenclass - hidden class is type of the neuron that is used in the model. In my case it will be set to the *LSTMLayer*.
outclass - out class is the output representation of the model output. In my case it will be the *SoftMaxLayer*. This will represent the probability of class in relation to the input. The summary of probabilities will be always 1.

recurrent - indicates if network is recurrent or not. If set to *True* a *RecurrentClass* instance will be created. If set to *False* a *FeedForwardNetwork* instance will be created.

Then I have to choose the trainer. The *PyBrain* doesn't have list of trainers to choose from. There are just two: the **Backpropagation** and the **RPropMinusTrainer**. I used the *RPropMinusTrainer*, because it is more suitable for my task. More about the *RPropMinusTrainer* you can find in [21]

Training

The *PyBrain* offers all options of training methods as I mentioned in section 7.1.5. Problem appeared using the *Cross-validation* implementation in the *PyBrain*. At the beggining of training, I succesfully created an instance of the *CrossValidator* class but then an error occured. The error was in the third party library therefore I decided to create my own *Cross-validation* method.

I decided to do the *Cross-validation* with 10 folds. Firstly I prepared the dataset which means to divide data into 10 parts. Then I created two lists. The former was created for the validation data and the latter for the training data. Then I run training 10 times for each item in the list and validated the model. You can imagine that seeing picture 7.1

Result

I use the *Scikit-learn* library to present training results. This library has it's own implementation of the neural network learning, but I have used it just for calculating results.

The *Scikit-learn* library allows to create the *confusion matrix* and the *F1 score* too. More about this library you can find in [22].

7.1.8 Keras

All negatives mentioned in the chapter 7.1.7 are practically removed in the *Keras* library. It is based on the *Theano* [23] (or the *TensorFlow*) library which has been made for evaluation of mathematical expressions involving multi-dimensional arrays efficiently. This means that calculations are much faster then with the *PyBrain*.

The *Keras* has also implemented other optimizers and loss functions. It is also newer and actively updated. More information about the *Keras* you can find in [24].

Dataset

The *Keras* uses directly the *Numpy* library to create a dataset. In comparison with the *PyBrain*, the *Keras* require already created dataset, but the *PyBrain* creates the array by its own methods. If you want to know more information about the *Numpy* visit [25]. Basically I need to create the dataset, which has 3 dimensions.

(nb_sequences, sequence_length, sample_length)

where:
nb_sequences - total number of sequences. This number depends on amount of loaded data and the sequence length.
sequence_length - length of one sequence. This value depends on the set value.
sample_length - length of one sample. It depends on number of used sensors (resources).

The problem is that the length of each sequence of the dataset in the *Keras* has to be the same. This is why I can't use the*PyBrain* dataset directly. It is necessary to cut the shorter sequences from the *PyBrain* dataset. Then it is possible to create the *Numpy* array from the *PyBrain*.

Model

The *Keras* has big variability of models. It is possible to create multi-layer models with different neurons.

I require just the *LSTM* and the *Sigmoid* layer for my task. Here is code example of creating a model in the *Keras*:

```
model = Sequential()
model.add(LSTM(num_neu, input_shape=(seq_len, input_dim
    )))
model.add(Dense(num_out, activation='softmax'))
model.compile(loss="loss", optimizer="optimizer",
    metrics=["accuracy"])
```

At first I have to create an instance of the **Sequential** model. The *Keras* offers just two possibilites of models - the **Sequential** and the **Model**.

I added one *LSTM* layer which has following parameters:
num_neu - amount of neurons in the layer. Actually it means number of outputs, but it means the same thing.
input_shape - shape of the input data. Basically it is 2 dimensions of the dataset in the *Keras*.7.1.8

I added the *Dense* layer which is fully connected layer with the *SoftMax* activation function.

Finally it is needed to compile the model with some chosen loss function and optimizer. The model will evaluate the accuracy during training and testing.

Training

Unfortunately the *Keras* doesn't have its own implementation of the *Cross-validation*. Luckily, it is possible to use the *Scikit-learn* library implementation of the *Cross-validation*.

```
skf = StratifiedKFold(dataset_classes,n_folds=N_FOLDS,
    shuffle=True)
```

```
for i, (train, test) in enumerate(skf):
```

I created an instance of the *StratifiedKFold* class that takes the dataset and splits it into training and validation datasets that I can directly use for training. Then I show how to do the training properly after the data splitting.

It can be done as described because the *Scikit-learn* library also use the *Numpy* library for dataset representation.

Next I have to prepare the *Keras* callbacks for stopping training at the best moment. The *Keras* has several callbacks to be applied at training. I will use the *EarlyStopping* and the *ModelCheckpoint*.

```
stop = EarlyStopping(monitor='val_acc', patience=30,
    verbose=0, mode='auto')
checkpointer = ModelCheckpoint(monitor='val_acc',
    filepath="weights.hdf5", verbose=0, save_best_only=
    True)
```

I created the *EarlyStopping* instance that stops training when monitoring value, in my case *'val_acc'*, reaches its minimum and haven't changed for 30 epochs. This callback just stop the training, but I need to use the best model parameters values.

This is why I set also the *ModelCheckpointer* which saves parameters of the model, which of *'val_acc'* is better than saved before. This provides to remember just the best values from training.

Result

I have used the same library and system as for the *PyBrain* library.7.1.7

7.2 Training results

In following chapters I describe the results of trials where I have used various combinations of the *LSTMs* quantity, the sample sequence length, the resample value, the optimizer and the loss function.

7.2.1 Tests with the *PyBrain*

First series of tests with the *PyBrain*

Firstly I trained just with data originated from the *Asphalt* and the *Bricks* surface. It was the first measurement I had made. It was small dataset containing around 20 minutes of useable data. Here are the results of the first training in the table followed by the *confusion matrix* shown below.

All tests are made with the 25 ms resample and trainer was the *Rprop-*.

Testing	LSTM	Sequence	Method	Validation F1
1	10	100	Min	0,9711
2	10	100	Epoch	0,9681
3	20	100	Epoch	0,9638
4	5	100	Epoch	0,9591
5	10	50	Epoch	0,9591
6	10	200	Epoch	0,9022

Table 7.2: First neural network model training with the *PyBrain*

Result	Asphalt	Bricks
Asphalt	1483	21
Bricks	95	1954

Table 7.3: *Confusion matrix* for testing Nr. 1

Explanation for the **Method** column:
Min - means that the algorithm tries to find the best solution and ends training when the best solution is found.
Epoch - means that the algorithm runs until chosen epoch passes regardless any parameter. If you want to know more about the training methods, read 7.1.5.

Colours in the table indicate how good is the result in comparision with the others in the table. The green indicates the best value, otherwise the red indicates the worst.

As you can see the results are really good. Even it is just a small dataset it shows that it is possible to detect the surface from mobile sensors. The neural network model was able to recognize almost all new sequences. It is also important to notice, that there are small but relevant differences in the *Validation result* between the testing #2, #3 and #4. I trained just with the different number of *LSTMs*. It looks like 5 units is not enough but this finding must be tested to be approved. I also didn't have enough samples.

I found out also one big problem. My computer is not powerful enough to run all necessary experiments. One training could also lasts for weeks. This was the reason why I started to do tests on the MetaCenter, which is a virtual organization for academics providing computing and storage resources. It has helped a lot.

Second series of tests with the *PyBrain*

Secondly, I went on with much more measured data. I also extended the number of surfaces. Now there are the *Asphalt*, the *Bricks*, the *Pavement* and the *Countryside*. I started to do tests just with the *Cross-validation* with 10 folds. I made the *F1 score* at the end of training of one fold of the *Cross-validation*. Then I made an average of *F1 scores*. Since now I will test exclusively on the MetaCenter.

Neurons	Sampling	Seq length	F1 average
5	15	100	0,60443
5	15	200	0,59971
5	15	400	0,58992
5	25	100	0,57683
5	25	200	0,60954
5	25	400	0,61680
10	15	100	0,62239
10	15	200	0,59789
10	15	400	0,60740
10	25	100	0,63437
10	25	200	0,63171
10	25	400	0,63525
20	15	100	0,59702
20	25	100	0,60714
20	25	200	0,60482
20	25	400	0,60479

Table 7.4: Second neural network model training with the *PyBrain*

Colours in the table indicate how good is the result in comparision with the others in the table. The green indicates the best value, otherwise the red indicates the worst.

In the table above you can see that the validation *F1 score* (**F1 average** column) rapidly decreased. First idea was that the increasing number of surfaces made training much harder and results would not be good, but then I found a mistake in my implementation of the *Cross-validation*. Mistake was related to the sequence length which was much longer than the set value.

Third series of tests with the *PyBrain*

In this phase of testing I fixed the problem of the sequence length and I tested a lot of combinations of parameters.

Test	Neurons	Sampling	Sequence	F1 score
1	8	10	400	0,61444
2	8	15	100	0,63349
3	8	15	200	0,63249
4	8	15	400	0,64465
5	8	25	100	0,61753
6	8	25	200	0,62230
7	8	30	100	0,62356
8	8	30	200	0,67125
9	8	40	100	0,70523
10	8	45	200	0,65369
11	8	50	30	0,61949
12	8	50	50	0,61492
13	8	50	100	0,67787
14	8	50	200	0,66884
15	8	50	400	0,68612
16	8	50	500	0,61628
17	10	25	100	0,62662
18	10	25	200	0,63169
19	10	25	400	0,63068
20	10	50	100	0,68229
21	10	50	200	0,66261
22	10	50	400	0,67554

Table 7.5: Third neural network model training with the *PyBrain*

Colours in the table indicate how good is the result in comparision with the others in the table. The green indicates the best value, otherwise the red indicates the worst.

As you can see, the *F1 score* has improved, but not so much. Even the best result are probably the combinations shown in the middle, it doesn't mean they are the best ones. It just shows, that these settings can give good results, but it can be also little bit coincidence. Since the *PyBrain* library doesn't allow more availability of settings and because of other disadvantages mentioned in the section 7.1.7 I decided for neural network learning implementation in the *Keras*.

7.2.2 Tests with the *Keras*

At the end I tested just with the *Keras* and on the MetaCenter. In all tests I used the *Cross-validation*. I tested each combinations at least five times in order to precisely test the measured data for deviations and then make average of the *F1 score*. I think that this method was the best one, because I almost eliminated all errors.

Firstly I tested just with the ***Rmsprop*** optimizer and the ***Categorical crossentropy*** loss function because it is usually used for recurrent networks

and *LSTMs*. Then I found out that the *Keras* offers various loss functions and optimizers so I wanted to test their combinations in order to find the best ones even most of them are not suitable for recurrent networks.

Optimizer Tests

The *Keras* has now 11 loss functions and 6 optimizers. If you want to know more about the *Keras* loss functions and optimizers, read about them in the *Keras* documentation at [26]. To test all of the combinations with the *Cross-validation* and several times it requires lots of computation resources. This is the reason why I tested optimizers just with one loss function that is the *Categorical crossentropy*. Settings of model are 8 *LSTMs*, 200 samples sequence length and 50 ms resample.

test	sgd	rmsprop	adagrad	adadelta	adam	adamax
1	0,6191	0,7202	0,5975	0,7217	0,6104	0,6243
2	0,5999	0,7259	0,5784	0,7560	0,6576	0,6292
3	0,6534	0,7080	0,5544	0,7145	0,6787	0,5679
4	0,6052	0,7048	0,6059	0,7286	0,7076	0,5755
5	0,5917	0,6988	0,5990	0,7270	0,7042	0,5801
6	0,6235	0,7004	0,5781	0,7079	0,6866	0,5729
7	0,6354	0,6633	0,5661	0,7128	0,6963	0,6421
8	0,6041	0,7169	0,6430	0,7646	0,6947	0,5649
9	0,6012	0,7042	0,6240	0,7266	0,6995	0,5990
10	0,6026	0,7100	0,5469	0,7316	0,6720	0,5901
avg	0,6136	0,7053	0,5893	0,7291	0,6808	0,5946

Table 7.6: Comparison of optimizers in the *Keras* for the *Categorical crossentropy*

Colours indicate how good optimizers are in comparition with the other optimizers.

You can see in the table, that the 3 best optimizers are the ***rmsprop*, the *adadelta*** and the ***adam***. I consciously displayed every test values, because I wanted to show that individual averages of the *Cross-validation* tests results are similar. That means that training datasets are properly shuffled.

Loss functions test

For the loss functions test I will use just the three best optimizers chosen in the optimizer test chapter 7.2.2. The table is devided into two parts, because it is too large for the one table. Settings of the model are 8 *LSTMs*, 200 samples sequence length and 50 ms resample.

TEST	mse	mae	mape	msle	cosine
rmsprop	0,6874	0,6245	0,6055	0,6274	0,6383
adadelta	0,7316	0,6494	0,6292	0,6540	0,6701
adam	0,6656	0,6169	0,5995	0,6215	0,6304

Table 7.7: Comparison of loss functions - part 1

TEST	cate-cros	sqr-hinge	hinge	poisson	bin-cross
rmsprop	0,6491	0,6542	0,6419	0,6479	0,6528
adadelta	0,6807	0,6844	0,6688	0,6766	0,6826
adam	0,6379	0,6425	0,6330	0,6392	0,6440

Table 7.8: Comparison of loss functions - part 2

Colours indicate in each row how good the loss function is in comparition with the other loss functions in the single row.

In tables 7.7 and 7.8 you can see that the ***Mean squared error***(*mse*) optimizer is the best one of all otpimizers. Further there are several loss-functions that has the similar average. In the next test I will continue with the ***mse*, the *Categorical crossentropy*, the *Squared hinge*** and the ***Binary crossentropy***.

First series of tests with the *Keras*

So far the *mse* had been the best loss function but I decided to test with the *Categorical crossentropy* as suitable loss function for recurrent networks as well. Simultaneously I compared some results obtained by both loss functions.

LSTM	Seq	Resample	Loss func	Rmsprop	Adadelta	Adam
5	200	50	Categ cross	75,24%	75,05%	73,25%
8	100	50	Categ cross	73,07%	74,20%	72,40%
8	200	10	Categ cross	78,05%	77,84%	77,79%
8	200	25	Categ cross	73,48%	73,14%	69,80%
8	200	50	Categ cross	74,88%	76,95%	74,26%
8	200	50	MSE	75,25%	76,07%	72,41%
8	400	50	Categ cross	69,26%	71,99%	68,57%
15	200	50	Categ cross	78,31%	79,58%	78,65%
20	200	10	Categ cross	76,78%	75,86%	75,01%
20	200	25	Categ cross	76,95%	77,38%	75,33%
20	100	50	Categ cross	80,25%	79,59%	77,63%
20	200	50	Categ cross	80,95%	80,34%	79,01%
20	200	50	MSE	80,17%	80,16%	78,21%

Table 7.9: First neural network model training with the *Keras*

Colours in table indicates, how good the results are. It means that I focused

just on the best result regardless the loss function or the optimizer.

As you can see from the table, the more *LSTMs* model has, the better results are. It looks like the increasing number of *LSTMs* could rapidly improved the results. It looks really promising. The validation *F1 score* in the average 82% is a great success. Because of low results from the *adam* optimizer I decided to remove it from my future tests.

Second series of tests with the *Keras*

I decided to start the training with all chosen loss functions from Loss functions test at 7.2.2 and two optimizers: the *Rmsprop* and the *Adadelta*. In the table below I display the validation *F1 score* results just for the *Categorical crossentropy* loss functions, because the table would be otherwise really big. For the best result with the *Categorical crossentropy* I will display results for other optimizers.

Test	LSTM	Seq	Resample	Loss func	Rmsprop	Adadelta
1	20	400	50	Categ cross	78,66%	78,29%
2	30	200	50	Categ cross	82,25%	81,18%
3	40	100	50	Categ cross	81,57%	80,88%
4	40	200	25	Categ cross	78,94%	77,68%
5	40	200	50	Categ cross	82,23%	80,98%
6	40	300	50	Categ cross	81,84%	80,58%
7	40	400	50	Categ cross	81,29%	80,13%
8	40	500	20	Categ cross	75,16%	72,84%
9	40	500	40	Categ cross	81,50%	80,24%
10	40	500	50	Categ cross	82,77%	81,44%
11	40	500	60	Categ cross	82,08%	80,53%
12	40	500	70	Categ cross	81,46%	79,68%
13	40	600	50	Categ cross	80,69%	79,12%
14	40	700	50	Categ cross	79,91%	78,63%
15	40	1000	10	Categ cross	68,17%	67,23%
16	60	200	50	Categ cross	82,67%	80,27%
17	100	200	50	Categ cross	82,46%	81,47%
18	200	200	50	Categ cross	81,92%	80,54%

Table 7.10: Second series of tests with the *Keras*

Colours in table indicates, how good the results are. It means that I focused just on the best result regardless the loss function or the optimizer.

As you can see I mostly tested with 40 *LSTMs* in the model. It is because I found out that it is probably the best qunatity. I also tried much more *LSTMs* in last tests but the tests lasted much longer because of the number of *LSTMs* parameters in model.

After settling the quantity of *LSTMs* I tried to find the best combination of the sequence length and the resampling time. It is important to change the both parameters, because they are dependent. According to the results I

chose 500 samples per sequence and 50 ms resample time as the best one.

Best result

Below is displayed the table and the *confusion matrix* with the best results I've got for settings with 40 *LSTMs*, 500 sample sequence length and 50 ms resample.

Loss func	Rmsprop	Adadelta
MSE	82,36%	80,88%
Categ cross	82,77%	81,44%
Sqr Hinge	82,79%	81,17%
Bin cross	82,58%	81,27%

Table 7.11: Best result of tests with the *Keras*

Result	Asphalt	Bricks	Pavement	Countryside
Asphalt	7	0	0	0
Bricks	0	2	0	0
Pavement	1	0	2	0
Countryside	0	0	0	9

Table 7.12: Confusion matrix for one of the the *Cross-validation* test of best result

At the end, the best loss function is the *Square Hinge*, but the *Categorical crossentropy* has almost the same results. The *Categorical crossentropy* is more suitable for recurrent networks therefore I will use it. Risky is that the *confusion matrix* had just 20 samples to analyse. With more sequences to analyze the success may could decrease and other models could be better. This is why I also display results of second best settings: 60 *LSTMs*, 200 sample sequence length, 50 ms resample

Loss func	Rmsprop	Adadelta
MSE	82,18 %	80,34 %
Categ cross	82,50%	81,06%
Sqr Hinge	82,64%	81,07%
Bin cross	82,58%	81,11%

Table 7.13: Second best result

Result	Asphalt	Bricks	Pavement	Countryside
Asphalt	18	0	0	2
Bricks	0	5	0	0
Pavement	1	0	9	1
Countryside	2	1	0	21

Table 7.14: Confusion matrix for one of the the *Cross-validation* test of second best result

As shown in the table 7.14 the validation *F1 score* is almost the same but number of the samples in the *confusion matrix* is 60 which is more than doubled in the previous test. It means that this option might be more suitable than the first best option.

Best result test

At the end I decided to test the two best options without the *Cross-validation*. It means testing just once with all available data. I have to split the dataset just into the validation and the training part in ratio 0.2:0.8. This split could also infulents how the data will occur in both sets. This is why I decided to test the option 10 times to eliminates the wrongly shuffled data.
Option 1: 40 *LSTM*, 500 sample sequence length, 50 resample
Option 2: 60 *LSTM*, 200 sample sequence length, 50 resample

Option	1	2
	80,12%	81,7%

Table 7.15: Final result

Result	Asphalt	Bricks	Pavement	Countryside
Asphalt	34	0	3	3
Bricks	0	10	0	0
Pavement	1	0	17	1
Countryside	2	0	4	46

Table 7.16: *Confusion matrix* for one of the tests of option 2

The last test shows that the second option looks better. The *confusion matrix* also contains lot of sequencies so this result is considered as a success.

Gyroscope test

As the end I tested my two best options also with the gyroscope data. I made new measurement for both the accelerometer and the gyroscope. It is not same amount of data as it was before, but the main point is to test these options with the two sensors. I used the same training method as in the previous section.

Option 1: 40 *LSTM*, 500 sequence length, 50 resample
Option 2: 60 *LSTM*, 200 sequence length, 50 resample

Option	1	2
	67,68%	76,71%

Table 7.17: The validation result of training with the gyroscope

As you can see the second option is again better than the first one. The lower validation *F1 score* result is caused with small amount of data, but I think that using these two sensor could leads to the better results.

Special test

I also tested settings with 100 ms resample. Firstly, I thought it is probably the best solution I found, but then I found out in the *confusion matrix* that it is not entirely true.

LSTM	Seq	Resample	Loss func	Rmsprop
40	500	100	Categ cross	85,57%

Table 7.18: Special test

Result	Asphalt	Bricks	Pavement	Countryside
Asphalt	3	0	0	0
Bricks	0	1	0	0
Pavement	0	0	1	0
Countryside	0	0	0	4

Table 7.19: Confusion matrix for one of the the special *Cross-validation* tests

In the *confusion matrix* is shown that the model succesfully analysed all given sequences. But problem is, that the model had to analyse just 9 sequences which is really small number. This is why I'm not using this option. It is important to have big amount of measured data to test this option.

7.3 Conclusion

I spent the most of the time on training because it was the main part of my work. I have to studied the *Python* libraries for neural network learning such as the *PyBrain*, the *Keras* and the *Scikit-learn*. It is necessary to know basic functionality of used neurons but it's still a little bit magic to find the best option. It is almost impossible for me to declare one option as the best one, because the model needs to be trained and validated with much more data. But this uncertainty will be always present and it must be taken into account.

There were various combinations and setting options to try and test. The best result was obtained with the option: 60 *LSTMs*, 200 sample sequence length and 50 ms resample. This model can predict the surface with success rate of 81,7% using the accelerometer data.

The same model with the data from both the gyroscope and the accelerometer can predict the surface with success rate of 76,71%.

Chapter 8

Conclusion

In this work I have developed a functional process from a mobile phone data measuring on a bike over data analyzing to a neural network model predicting surfaces. This process contains the Android application which allows recording and saving data from chosen sensors. I have studied the Android operating system, especially how to implement loading and savig the data for the sensors, the GPS and the audio.

The application has provided valuable data which I have used. I rode about a hundred kilometers which provided approximately 6 hours of records.

The application for filtering and annotating data has followed as a next process step. I have studied the *Python* language, development tools and libraries, for example, the *Pandas* and the *Matplotlib*. It was shown that it is possible to analyse differences between surfaces.

Afterwards I have chosen the *LSTM* as the best option for sequential and time dependent learning. The *LSTM* could be successfully used with different training settings and options. I have studied a neural network implementation libraries such as the *PyBrain* and the *Keras*. I made a few hundred tests and experiments both on my computer and the MetaCenter. The best option I have discovered for neural network model is: 60 *LSTM*, 200 sample sequence length, 50 ms resample. This model can predict the surface with success rate of 81,7%.

The same model with data from both the gyroscope and the accelerometer can predict the surface with success rate of 76,71%.

This work could be extended in different ways.

Firstly, you can focus on other mobile phone sensors. Mobile phones offer more sensors which can be useful to analyze. It could also include further development of the Android application development and phone positioning on a bike. I suggest to consider modifications in the measuring application or in the data preparation so that measured data are as little as possible dependent on a phone position.

Secondly, it is possible to work with the measured data trying to improve neural network learning process. Variability of neural network models is large.

Appendix A

Bibliography

[1] Google Inc. Sensor Coordinate System. Android Developers. [online] 20.4.2016 [cit. 2016-04-20]. http://developer.android.com/guide/topics/sensors/sensors_overview.html#sensors-coords.

[2] Jeanlagi. Formal neuron model. [online] 1.4.208 [cit. 2016-04-10].

[3] Christopher Olah. Colah's blog. Artificial Neural Networks. [online] 1.8.2015 [cit. 2016-05-03].

[4] Skymind. A Beginner Guide to Recurrent Networks and LSTMs. Deep Learning. [online] 28.3.2016 [cit. 2016-03-28]. http://deeplearning4j.org/lstm.html.

[5] Chris McCormick. K-Fold Cross-Validation, With MATLAB Code. [online] 1.8.2013 [cit. 2016-04-10]. https://chrisjmccormick.files.wordpress.com/2013/07/10_fold_cv.png.

[6] Google Inc. Sensor Overview. Android Developers. [online] 16.4.2016 [cit. 2016-04-16]. http://developer.android.com/guide/topics/sensors/sensors_overview.html.

[7] Google Inc. Motion Sensors. Android Developers. [online] 16.4.2016 [cit. 2016-04-16]. http://developer.android.com/guide/topics/sensors/sensors_motion.html#sensors-motion-accel.

[8] Google Inc. Audio Capture. Android Developers. [online] 22.5.2016 [cit. 2016-05-22]. https://developer.android.com/guide/topics/media/audio-capture.html.

[9] Convert CSV to KML. [online] 10.3. 2015 [cit. 2015-03-10]. http://www.convertcsv.com/csv-to-kml.htm.

[10] Dimitrios Siganos and Christos Stergiou. Neural Networks. Imperial College London - Deparment of Computing. [online] 12.5.1997 [cit. 2016-04-24]. https://www.doc.ic.ac.uk/~nd/surprise_96/journal/vol4/cs11/report.html#A%20simple%20neuron.

[11] Paul Boersma and David Weenink. What is a feedforward neural network? Feedforward neural networks [online] 30.10.2015 [cit. 2016-03-28]. http://www.fon.hum.uva.nl/praat/manual/Feedforward_neural_networks_1__What_is_a_feedforward_ne.html.

[12] Christopher M. Bishop. *Pattern recognition and machine learning*. Springer, 2006.

[13] Kiyoshi Kawaguchi. The McCulloch-pitts model of neuron. Artificial Neural Networks. [online] 17.6.2000 [cit. 2016-04-19]. http://wwwold.ece.utep.edu/research/webfuzzy/docs/kk-thesis/kk-thesis-html/node12.html.

[14] Moez Baccouche and et al. Sequential deep learning for human action recognition. *Human Behavior Understanding. Springer Berlin Heidelberg*, pages 29–39, 2011.

[15] Sebastian Ruder. An overview of gradient descent optimization algorithms. [online] 19.1.2016 [cit. 2016-03-29]. http://sebastianruder.com/optimizing-gradient-descent/.

[16] Martin Riedmiller. Rprop - Description and implementation Details. Technical Report. [online] 1.1.1994 [cit. 2016-05-10]. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.21.3428&rep=rep1&type=pdf.

[17] Michael A. Nielsen. How the backpropagation algorithm works. Determination Press, 2015. http://neuralnetworksanddeeplearning.com/chap2.html.

[18] Klaus Greff, et al. LSTM: A Search Space Odyssey. *arXiv preprint arXiv:1503.04069*, 2015.

[19] Kevin Markham. Simple guide to confusion matrix terminology. *Data School*, 26.3.2014. http://www.dataschool.io/simple-guide-to-confusion-matrix-terminology/.

[20] PyBrain library. [online] 28.4.2016 [cit. 2016-04-28]. http://pybrain.org/.

[21] Christian Igel and Michael Husken. Empirical evaluation of the improved Rprop learning algorithms. *Neurocomputing*, 2001.

[22] Scikit-learn library. [online] 28.4.2016 [cit. 2016-04-28]. http://scikit-learn.org/stable/index.html#.

[23] Theano library. [online] 28.4.2016 [cit. 2016-04-28]. http://www.deeplearning.net/software/theano/.

[24] Francois Chollet. Keras. [online] 12.5.2016 [cit. 2016-05-12]. https://github.com/fchollet/keras.

[25] NumPy library. [online] 28.4.2016 [cit. 2016-04-28]. http://www.numpy.org/.

[26] Francois Chollet. Keras Optimizers. [online] 12.5.2016 [cit. 2016-05-12]. http://keras.io/optimizers/.

Appendix B

Content of the CD

`sensorData`	The folder with the Android project of the Android application in the chapter 3
`Files_loading.py`	The *Python* file containing code described in the chapter 5
`Network_PyBrain.py`	The *Python* file containing code described in the chapter 7.1.7
`Network_Keras.py`	The *Python* file containing code described in the chapter 7.1.8
`BW_Bednar_2016.pdf`	The electronic version of bachelor work

Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Control Engineering

BACHELOR PROJECT ASSIGNMENT

Student: **Jan Bednář**

Study programme: Cybernetics and Robotics
Specialisation: Systems and Control

Title of Bachelor Project: **Bicycle Transport Network Parameters Extraction Based on Mobile Phone Sensors**

Guidelines:

1. Develop an Android application collecting time-series data from available mobile phone sensors. An appropriate subset of sensors will be chosen in a cooperation with the supervisor. It might include: GPS, accelerometer, gyro, microphone or even a camera.
2. Use the application to collect a reasonable training dataset by recording while riding a bicycle on preselected tracks. The task might involve a development of a simple time-series annotation tool.
3. Propose and implement machine learning methods for modelling transport network properties like surface type or traffic intensity. Focus on approaches based on Long-Short Term Memory (LSTM).
4. Evaluate implemented methods using real-world data.

Bibliography/Sources:

[1] Baccouche, Moez, et al. "Sequential deep learning for human action recognition." Human Behavior Understanding. Springer Berlin Heidelberg, 2011. 29-39.
[2] Bishop, Christopher M. Pattern recognition and machine learning. springer, 2006.
[3] Greff, Klaus, et al. "LSTM: A Search Space Odyssey." arXiv preprint arXiv:1503.04069 (2015).

Bachelor Project Supervisor: Ing. Jan Drchal, Ph.D.

Valid until the summer semester 2016/2017

L.S.

prof. Ing. Michael Šebek, DrSc.
Head of Department

prof. Ing. Pavel Ripka, CSc.
Dean

Prague, February 1, 2016