University of Southern Denmark

Master Thesis

Classification of terrain based on proprioception and tactile sensing for multi-legged walking robot

Author:
Bc. Martin Bulín

Supervisors:
Dr. Tomas Kulvicius
Dr. Poramate Manoonpong

A thesis submitted in fulfillment of the requirements for the degree of Master of Science

in the

Embodied AI & Neurorobotics Lab
Faculty of Engineering

May 14, 2016

Declaration of Authorship

I, Bc. Martin Bulín, declare that this thesis titled, "Classification of terrain based on proprioception and tactile sensing for multi-legged walking robot" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

"*Favorite quotation.*"

Quotation Author

UNIVERSITY OF SOUTHERN DENMARK

Abstract

Faculty of Engineering

Embodied AI & Neurorobotics Lab

Master of Science

Classification of terrain based on proprioception and tactile sensing for multi-legged walking robot

by Bc. Martin BULÍN

The abstract is a concise and accurate summary of the research described in the document. It states the problem, the methods of investigation, and the general conclusions, and should not contain tables, graphs, complex equations, or illustrations. There is a single abstract for the entire work, and it must not exceed 350 words in length....

Acknowledgements

Students may include a brief statement acknowledging the contribution to their research and studies from various sources, including (but not limited to)

Their research supervisor and committee, Funding agencies, Fellow students, and Family.

The acknowledgments and the people to thank go here, don't forget to include your project advisor…

Contents

List of Figures

List of Tables

List of Algorithms and Code Parts

List of Abbreviations

API	**A**pplication **P**rogramming **I**nterface
GUI	**G**raphical **U**ser **I**nterface

Chapter 1

Introduction

The thesis must clearly state its theme, hypotheses and/or goals (sometimes called “the research question(s)”), and provide sufficient background information to enable a non-specialist researcher to understand them. It must contain a thorough review of relevant literature, perhaps in a separate chapter.

1-2 pages intro

1.1 Problem Formulation

1 page Motivation and Research Questions

1.2 Hypotheses

1/2 page

1.3 Relation to the State of the Art

motivation for using proprioception sensing motivation for using a neural net as a classifier

in this section you can relate your work to the existing state of the art methods and tell why you have chosen those and what is your contribution to the state of the art

1/2 page

1.4 Thesis Outline

1/2 page

Chapter 2

State of the Art

chapter intro

2.1 Machine Learning and Classification

Machine Learning and Classification in general, different classifiers (SVM, k-NN, RandomForest, Bayes...)

2-3 pages

Classification, one of the most widely used areas of machine learning, has a broad array of applications. To fit a classifier to a problem, one needs to define a problem data structure. Data consists of samples and discrete targets, often called classes. The samples are sooner or later converted into so called feature vectors of a fixed length. The length of feature vectors usually determines an input of a chosen classifier and the number of classes sets an output.

2.2 Introduction to Neural Networks

neural networks from the beginning, network types, principles its usage for classification

4-5 pages

2.3 Pruning Algorithms

based on the paper Pruning Algorithms - A Survey: a summary of what has been already done, principles 1-2 pages

2.4 Terrain Classification for Legged Robots

based on the literature : a summary of what has been already done in terrain classification, summary of different methods (visual, laser, haptic, proprioception, ...)

5-8 pages

Chapter 3

Master Thesis Objectives

objectives (goals) 1/2 page

Chapter 4

Neural Net Implementation

Plenty of neural network implementations are available nowadays. Nevertheless, one of the objectives of this thesis is to implement own framework capable of using the idea behind artificial feedforward neural networks. Besides prooving a knowledge of mathematical and algorithmical backgrounds, an integration of own utilities and functions is the main reason for the from-scratch implementation.

To accomplish the reasearch objectives, the new framework must meet following requirements, which might be unusal for some of the provided implementations (mentioned in chapter 2).

- ability to remove any synapse in a network and then to retrain the network of the new structure
- ability to evaluate a network after each learning epoch and basically to provide an open-sourced learning algorithm
- ability to illustrate a network structure and to visualize the learning process in real time (an extra property)

In this thesis, the implemented neural network framework is called **kitt_nn** and has been developed in programming language Python. The following diagram (4.1) shows the structure of the *kitt_nn .py package*.

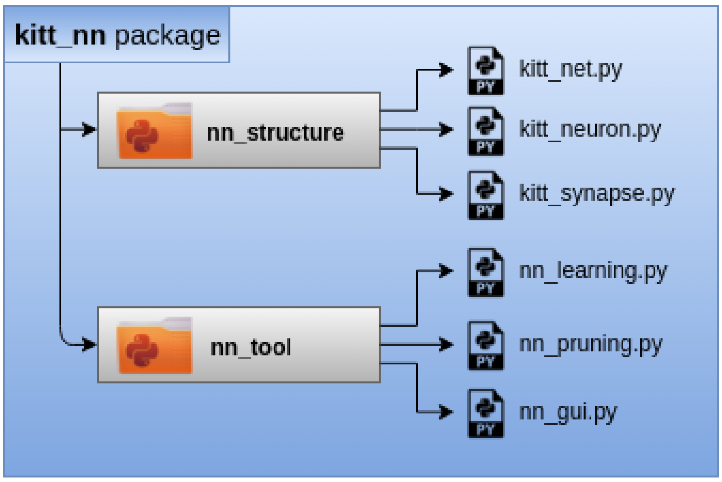

FIGURE 4.1: kitt_nn package : Implemented neural network framework

Moreover, the framework must have some standard functions implemented, meaning it must be capable of:

- initializing a feedforward network of any structure supplied by some randomly set parameters
- fitting a model to a network (function *fit()*), training a network on some data of a conventional structure
- predicting a target of never-seen samples (function *predict()*), evaluating a classification performance

The *kitt_nn* implementation is based on some general knowledge gained at school and/or from [], the idea is pretty straight forward.

4.1 Structural Elements

The overall idea is based on the object-oriented programming. There are three main *.py* files containing the main classes corresponding to strucutural elements - a network, a neuron and a synapse (a connection). A detailed API is attached as an appendix (A3).

kitt_neuron.py

The very basic units of a neural net are called neurons. In case of artificial systems, these units are responsible for transfering all their inputs into one output. The behavior is moreless the same for all of the units, therefore a class called *Neuron* implements some basic common functions.

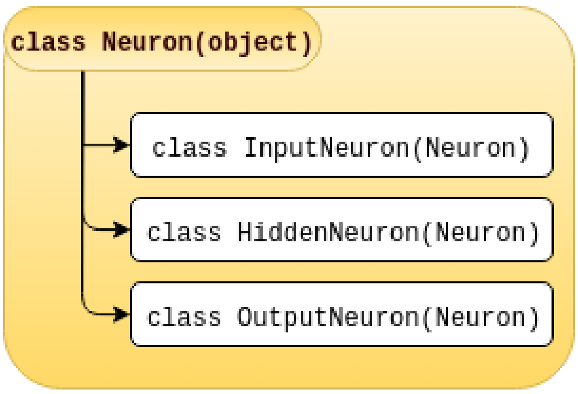

FIGURE 4.2: kitt_neuron.py : Neuron class inheritance

Then, as Fig. 4.2 shows, three classes are inherited from the *Neuron* class. Some special functions, depending on the layer a neuron is part of, can be implemented this way.

kitt_synapse.py

Next, there is a class representing a neural connection - a synapse. An instance of this class takes care of the corresponding weight and remembers the two connected neurons.

Additionally, a function called **remove_self()** is implemented, which sets the weight to zero and removes the synapse from all databases of the corresponding neural net. Then it also checks the two connected neurons, if they have some other connections remaining. If not, they are labeled as *dead*, as they are not a part of the network anymore.

kitt_net.py

The network is initialized by creating an instance of *NeuralNetwork()* class from *kitt_net.py*. The initialization process is illustrated in Fig. 4.3. Basically, the only parameter is the network structure, which is expected as a *.py iterable* type. For instance, a network with 2 input, 5 hidden and 3 output units would be created as *NeuralNetwork(structure=[2, 5, 3])*. Number of hidden layers is not limited.

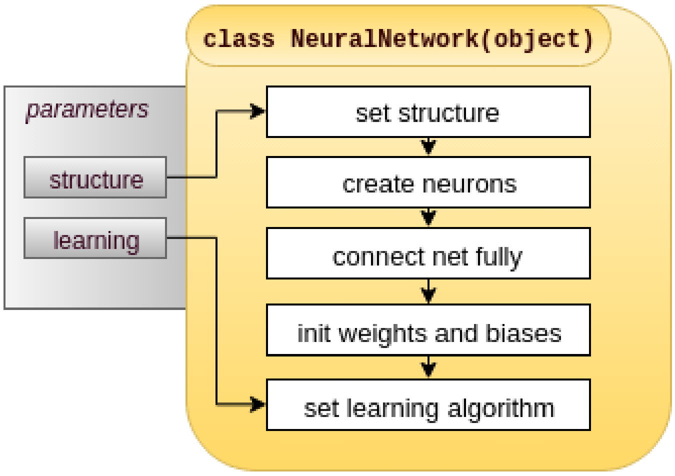

FIGURE 4.3: kitt_net.py : Neural Network Initialization

A learning algorithm is added to the initialized network thereafter (see section 4.2). The network class implements basic functions like *fit()*, *predict()* in order to be used as a classifier. Moreover, it has some additional utilities like *copy_self()* or *print_self()*, which are essential for this research (section 4.3, section 4.4).

4.2 Learning Algorithm

Backpropagation implementation in python

algorithm

1-2 pages

4.3 Graphical User Interface

The graphical interface has been implemented as an extension for *kitt_nn* framework. It is actually not strictly needed for this research, but it provides some interesting functions, which are worth of being introduced. Anyway, any type of visualization usually helps to understand a problem better.

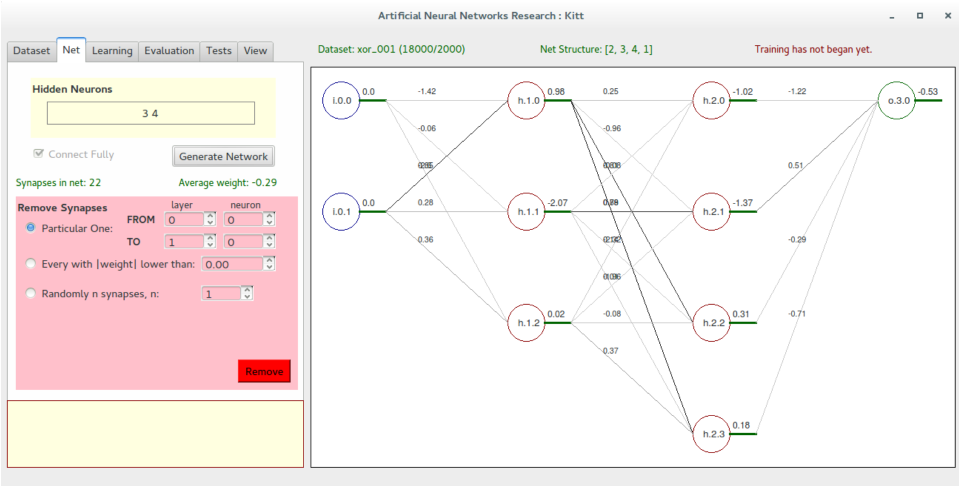

FIGURE 4.4: Screenshot of the graphical user interface

This GUI is capable of:

1. Loading a dataset in a specific form and, if possible, visualizing it (see XOR data in Fig. 4.6 for an example, this image is generated by the GUI).
2. Generating a network of any hidden structure. The input and output layers are defined by the chosen dataset. The network is then visualized (as shown in Fig. 4.4).
3. Removing synapses of the network, while the visualization is interactive with the structure changes.
4. Training the network, while the visualization is interactive, so the weights changes can be seen online.
5. Performing some tests and plotting simple evaluations.
6. Adjusting the visualization view in sense of zooming, resizing or changing colors.

The visualization is not that useful for huge network structures, however, it can be essential at some points of the research. Nevertheless, it is considered as the very fist version for now and aimed to be upgraded in the future.

4.4 Network Pruning Algorithm

4.4.1 Idea

4.4.2 Realization

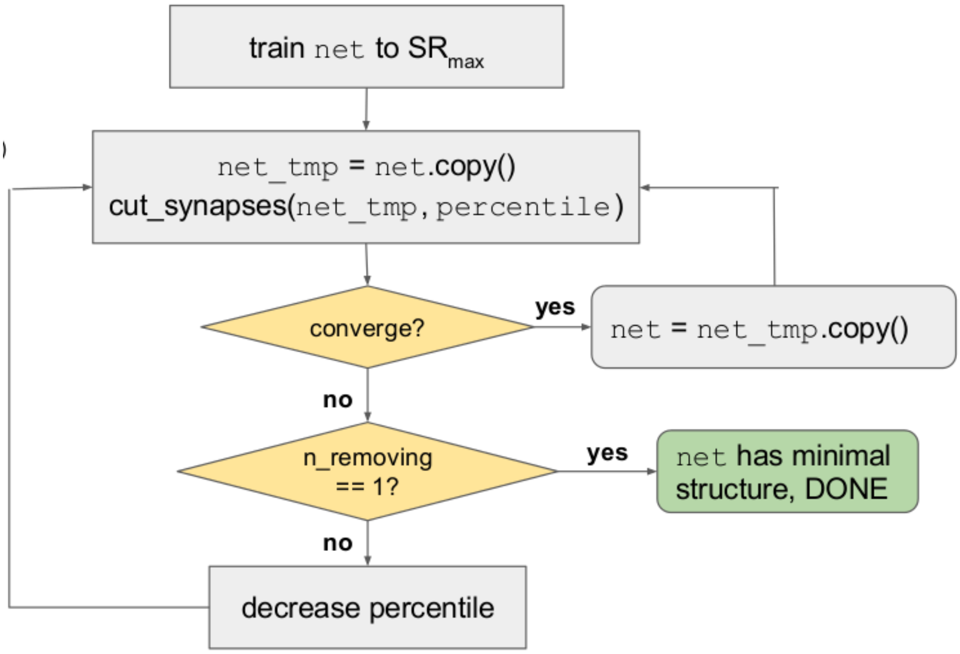

FIGURE 4.5: The Pruning Algorithm

4.4.3 Testing Datasets

XOR Dataset

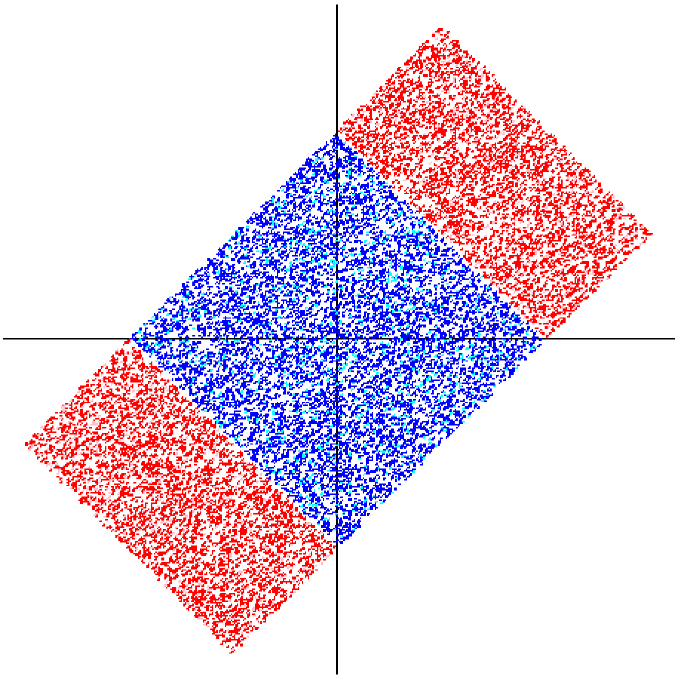

FIGURE 4.6: 2D XOR Data illustration

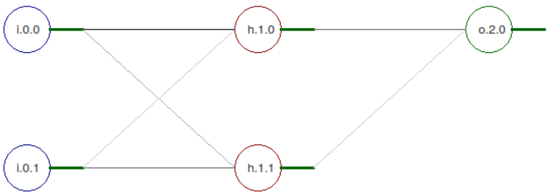

FIGURE 4.7: XOR min structure 1

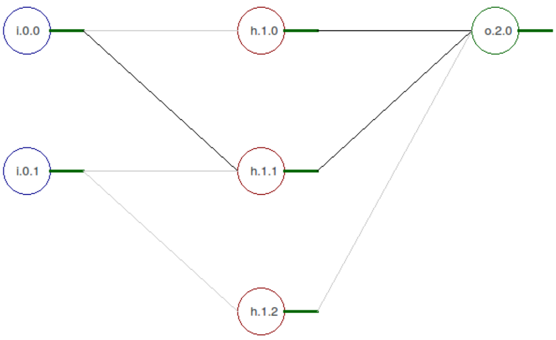

FIGURE 4.8: XOR min structure 2

MNIST Dataset

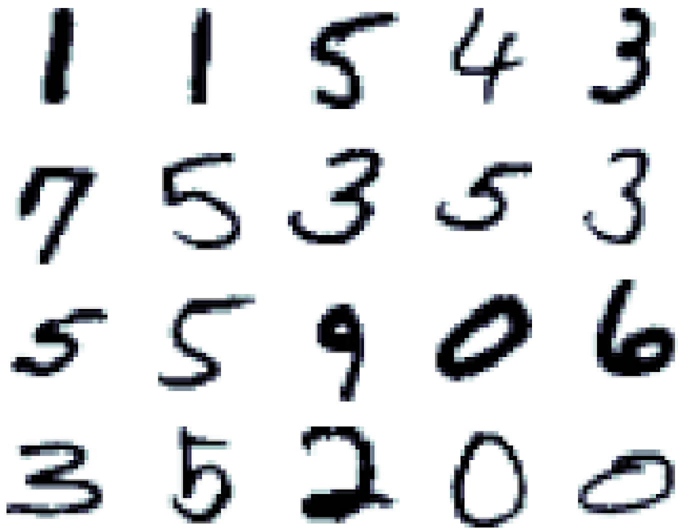

FIGURE 4.9: MNIST Data illustration

4.4.4 Minimal Structures Utilization

further MNIST analysis

figures, tables

4-5 pages

Chapter 5

Terrain Classification for hexapod robot AMOS II

The classification problem in this thesis relates to AMOS II, an open-source multi sensori-motor robotic platform (see Fig. 5.2). The task is to classify various terrain types based on proprioceptive (joint angles) and tactile (ground contact) sensors. The overall process is based on simulation data and as stated in chapter 4, feedforward neural networks are used for classification.

5.1 Overall Process Summary

The overall process consists of several modules. The workflow is illustrated in Fig. 5.1 (a more detailed diagram can be found in Fig. A2.1).

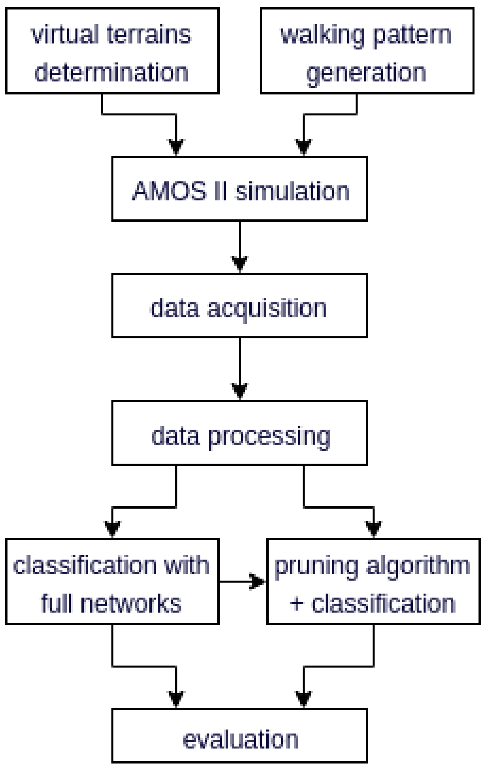

FIGURE 5.1: Terrain Classification: overall process diagram

The very first step is to make the AMOS II simulation run (appendix A2.1). Then a simple tripod gait controller is implemented (section 5.2.3). To generate various terrain types, the number of variable terrain features and their ranges are determined (section 5.3.1). Based on these features (parameters), a number of virtual terrains is defined (section 5.3.2) and an optimality of these parameters is briefly analysed (section 6.2.1).

Next, AMOS II (its simulation alternative) is forced to walk on every defined terrain type several times and for a sufficiently long period of time and the data from all proprioceptors are saved. This data is then verified and failing experiments are removed. The data acquisition step is parameterized by a standard deviation of an additive (Guassian) terrain noise and is run for several values.

Having the clean simulation data from all sensors, a feature vector structure is determined. Then a Gaussian signal noise is added.

Finally, a dataset is created by splitting all the data into training, validation and testing sets. As it is indicated in Fig. A2.1, several datesets and several classifiers are generated during the process.

The dataset packages may differ in these parameters:

- terrain types included (-> number of classes)
- sensors on input
- samples length (number of simulation timesteps)
- terrain noise and signal noise
- number of samples

The trained networks may differ in the following parameters:

- dataset that the network has been tested on
- neural network structure, learning rate and number of epochs

An optimal neural network classifier is found. The optimal network is then pruned by the algorithm developed in section 4.4. The classification performance of developed tools is compared to *Scikitlearn-neuralnetwork* classification library (*sknn: Deep Neural Networks without the Learning Cliff*).

5.2 Experimental Environment Specification

The ultimate goal is to implement an online terrain classifier for selection of optimal walking gait on real hexapod robot AMOS II. Therefore the real robot is presented in the following section 5.2.1.

However, as already stated above, the proposed approach will be evaluated using simulated robot in a virtual environment. In this case, *LPZ Robots simulator* (*Research Network for Self-Organization of Robot Behavior*) was used and a description is given in section 6.2.2.

5.2.1 Hexapod Robot AMOS II

The *AMOS II* abbreviation stands for Advanced Mobility Sensor Driven-Walking Device - version II (*Open-source multi sensori-motor robotic platform AMOS II*). It is a biologically inspired hardware platform of size 30x40x20 cm and weight 5.8 Kg (see Fig. 5.2). It is mainly used to study a neural control, memory and learning for machines with many degrees of freedom. The body and parts of the robot are inspired by a cockroach.

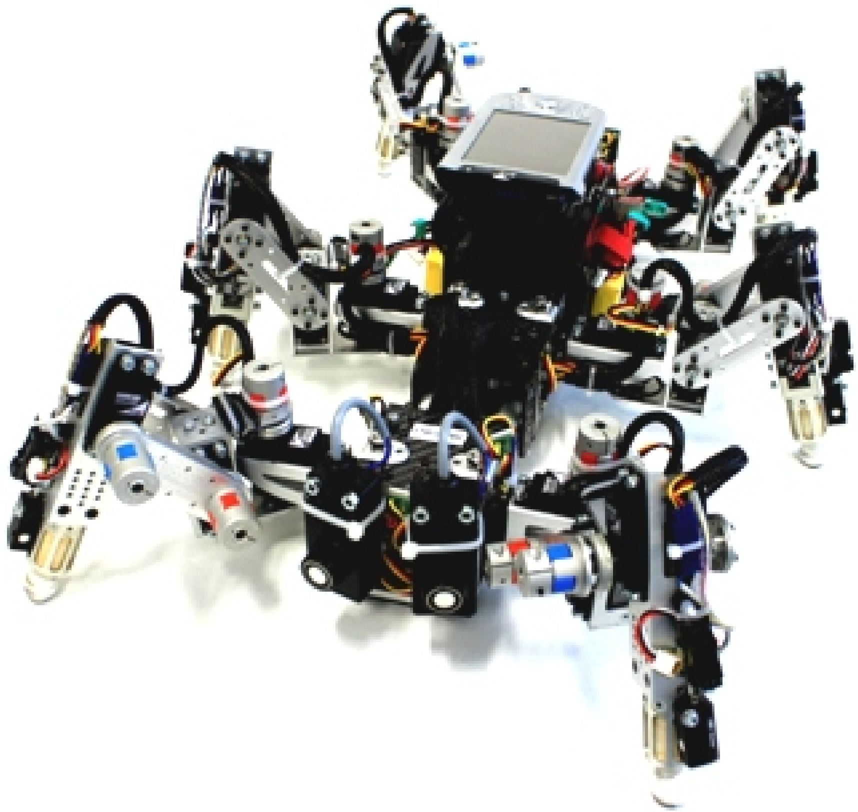

Figure 5.2: AMOS II. (*Open-source multi sensori-motor robotic platform AMOS II*)

A wide range of sensors (for instance infra-red, reflexive optical, light-dependent, laser, camera, inclinometer sensors) allows AMOS II to perform several kinds of autonomous behaviour including foothold searching, elevator reflex (swinging a leg over obstacles), self-protective reflex (standing in an upside-down position), obstacle avoidance, escape responses etc. (ibid.). However, only proprioceptive and tactile sensors are important for this study. Therefore, we focus on joint angle sensors and foot contact sensors. All of them are located on robot's legs. The leg structure is shown in Fig. 5.3.

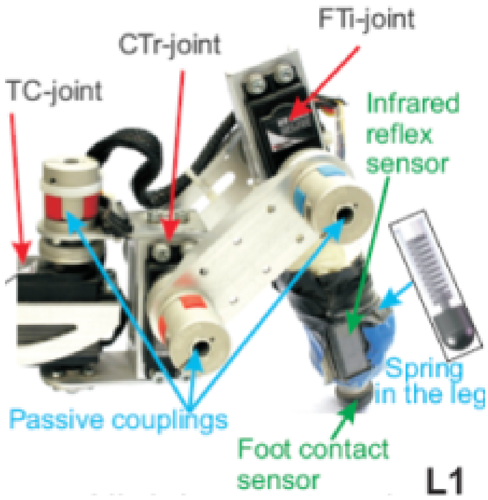

FIGURE 5.3: Structure of the AMOS's leg. (*Open-source multi sensori-motor robotic platform AMOS II*)

As shown in Fig. 5.2 and Fig. 5.3, the robot has *6 foot contact sensors* in total, one on each leg. Each of them returns a value from range $[0.0, 1.0]$ depending on how strong the foot contact is, i.e., it is equal 1.0 if the robot stands on the leg with its full weight and it equals 0.0 when the leg is in the air.

There are three joints on each of the robot's legs. The thoraco-coxal (TC-) joint is responsible for forward/backward movements. The coxa-trochanteral (CTr-) joint enables elevation and depression of the leg and the last one, femur-tibia (FTi-) joint is used for extension and flexion of the tibia.

These joints are physically actuated by standard servo motors. Angles of the joints are measured by the servo motors and are considered as propriceptive sensors. As AMOS II has six legs and there are three joints on each leg, there are *18 angle sensors* in total. There is also one backbone joint angle, however, as this one is not implemented in the simulation (see appendix A2.1), it is omitted in this work.

In Table 5.1 all the propriceptive sensors, their abbreviations and original ranges are listed. The ranges are based on the individual servo motors locations and are manually set to avoid collisions. In section 5.5 a normalization of these ranges is discussed.

Robot actuators (servo motors) can generate movements of variable compliance by utilizing a virtual muscle model (see *Open-source multi sensori-motor robotic platform AMOS II* for details).

TABLE 5.1: Summary of proprioceptive sensors of AMOS II hexapod robot

abbr.	*sensor description*	*original range*
ATRf	Angle sensor, Thoraco joint, Right front leg	[-1.5, 1.5]
ATRm	Angle sensor, Thoraco joint, Right middle leg	[-1.5, 1.5]
ATRh	Angle sensor, Thoraco joint, Right hind leg	[-1.5, 1.5]
ATLf	Angle sensor, Thoraco joint, Left front leg	[-1.5, 1.5]
ATLm	Angle sensor, Thoraco joint, Left middle leg	[-1.5, 1.5]
ATLh	Angle sensor, Thoraco joint, Left hind leg	[-1.5, 1.5]
ACRf	Angle sensor, Coxa joint, Right front leg	[-1.5, 1.5]
ACRm	Angle sensor, Coxa joint, Right middle leg	[-1.5, 1.5]
ACRh	Angle sensor, Coxa joint, Right hind leg	[-1.5, 1.5]
ACLf	Angle sensor, Coxa joint, Left front leg	[-1.5, 1.5]
ACLm	Angle sensor, Coxa joint, Left middle leg	[-1.5, 1.5]
ACLh	Angle sensor, Coxa joint, Left hind leg	[-1.5, 1.5]
AFRf	Angle sensor, Femur joint, Right front leg	[-1.5, 1.5]
AFRm	Angle sensor, Femur joint, Right middle leg	[-1.5, 1.5]
AFRh	Angle sensor, Femur joint, Right hind leg	[-1.5, 1.5]
AFLf	Angle sensor, Femur joint, Left front leg	[-1.5, 1.5]
AFLm	Angle sensor, Femur joint, Left middle leg	[-1.5, 1.5]
AFLh	Angle sensor, Femur joint, Left hind leg	[-1.5, 1.5]
FRf	Foot contact sensor, Right front leg	[0.0, 1.0]
FRm	Foot contact sensor, Right middle leg	[0.0, 1.0]
FRh	Foot contact sensor, Right hind leg	[0.0, 1.0]
FLf	Foot contact sensor, Left front leg	[0.0, 1.0]
FLm	Foot contact sensor, Left middle leg	[0.0, 1.0]
FLh	Foot contact sensor, Left hind leg	[0.0, 1.0]

It is possible to generate various gaits using joint actuators and robot's neural locomotion control. The gait controller used to generate robot locomotion is described in section 5.2.3.

5.2.2 AMOS II Simulation

The *lpzrobots* project, developed by a research group at the University of Leipzig (*Research Network for Self-Organization of Robot Behavior*) under GPL license, is used for AMOS II virtual representation. Some implementation details are discussed in appendix A2.1. The project modules important for this study are shown in Fig. 5.4.

With reference to appendix A3, the *main.cpp* file from `root/simulation/mbulinai22015-gorobots_edu-fork/practices/amosii` directory can be called as the main simulation file for purposes of the thesis. It sets up the environment with initial parameters:

- $controlinterval = 10$
- $simstepsize = 0.01$

This results in setting the simulation sensitivity to *10 steps* per second.

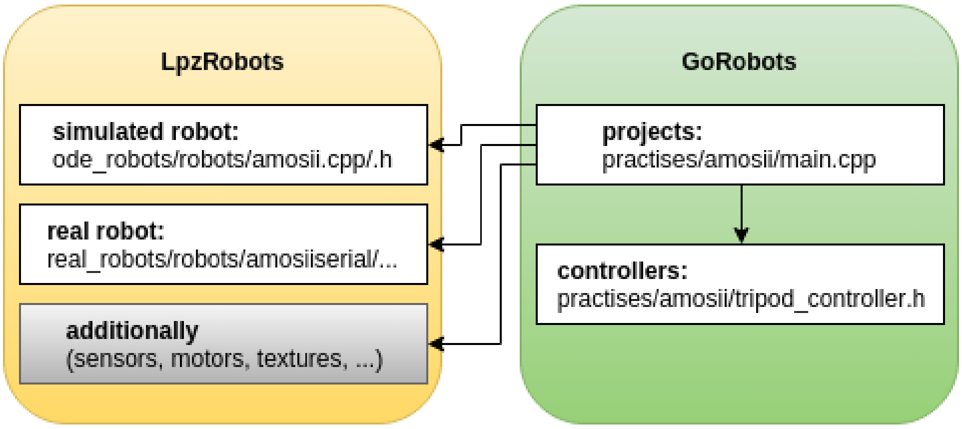

FIGURE 5.4: Structure of the two repositories: LPZRobots and GoRobots. (*Research Network for Self-Organization of Robot Behavior*)

The initial robot position in the map is chosen randomly in order to generate a different route everytime the simulation is launched. The robot fixator, which is originally implemented for AMOS II, is removed, so the robot starts walking right after the simulation is launched.

The *main.cpp* file contains all terrain types parameters introduced in **??**. The required terrain to be simulated is then passed to this file as an argument. Additionally, the standard deviation value of Gaussian terrain noise (details in section 5.3.3) is set as another argument.

Finally, the file is ready to take one more argument, which is a simulation noise represented by a float number. In this study it is fixed to zero though and only the terrain noise combined with a signal noise is used.

The virtual vizualization of AMOS II is illustrated in Fig. 5.5.

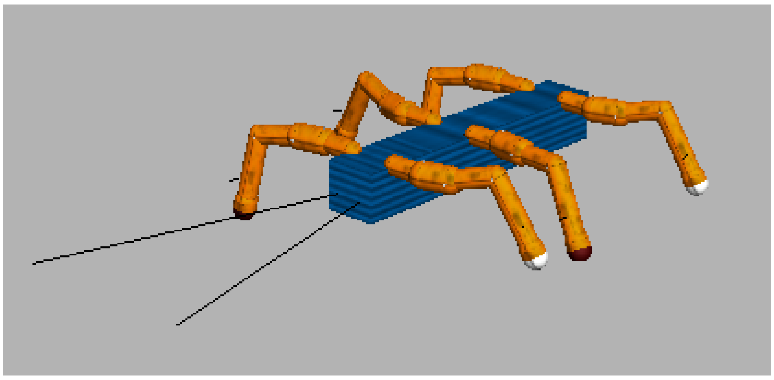

FIGURE 5.5: Virtual alternative for AMOS II.

Besides the backbone joint, all AMOS II actuators, proprioceptive and tactile sensors are modeled in the simulation and *LpzRobots* framework is considered to provide an accurate simulated model of AMOS II.

5.2.3 Tripod Gait Controller

The main motivation for the terrain classification is to adjust the current robot's gait accordingly and this way save some energy. In this work a *tripod* gait (three legsg touching ground when walking) is used for classification. The tripod pattern is the fastest and most common gait for hexapods.

To generate the tripod gait, a central pattern generator (CPG) is used ("Adaptive Embodied Locomotion Control Systems"). It is implemented as a 2-neuron neural network as shown in Fig. 5.6.

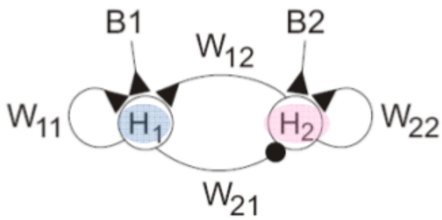

FIGURE 5.6: 2-neuron network oscillator ("Adaptive Embodied Locomotion Control Systems")

The initial conditions and parameters of the implemented controller are shown in Table 5.2.

TABLE 5.2: Initialization of *tripod_controller.h* (see appendix A3)

parameter	*initial value*	*description*
aH_1	0.0	activity of neuron H_1
aH_2	0.0	activity of neuron H_2
oH_1	0.001	output of neuron H_1
oH_2	0.001	output of neuron H_2
bH_1	0.0	bias for neuron H_1
bH_2	0.0	bias for neuron H_2
wH_1H_1	1.4	weight of the synapse from H_1 to H_1
wH_1H_2	0.4	weight of the synapse from H_2 to H_1
wH_2H_1	-0.4	weight of the synapse from H_1 to H_2
wH_2H_2	1.4	weight of the synapse from H_2 to H_2
p_1	0.35	parameter for Thoraco joints
p_2	0.3	parameter for Coxa joints

Then, during the simulation, robot's joints are controlled in every simulation step by performing three actions:

1. **The activation function application**

$$a_i(t+1) = \sum_{j=1}^{n} w_{ij} o_j(t) + b_i, i = 1, ..., n \tag{5.1}$$

2. **The transfer function application**

$$f(a_i) = tanh(a_i) = \frac{2}{1 + e^{-2a_i}} - 1 \tag{5.2}$$

3. **Joint settings**

   With the reference to previous equations and variables names, the actuators are set as shown in Fig. 5.7. The *femur* joints (red ones) stay unchanged (set to zero). This setting generates a tripod gait for AMOS II.

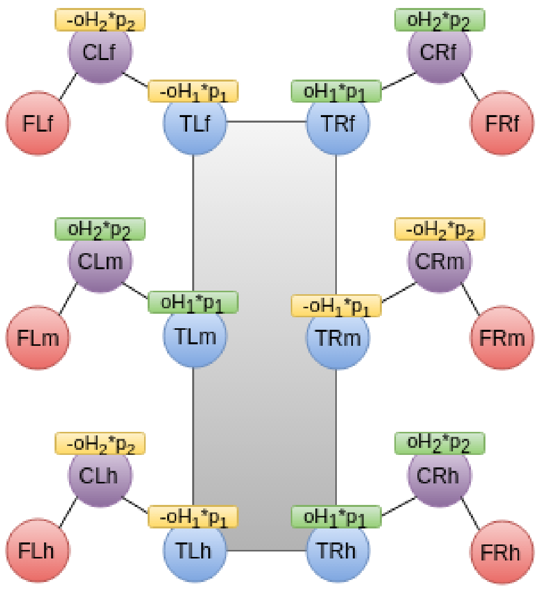

FIGURE 5.7: Schematic diagram of tripod gait controller

5.3 Generation of Virtual Terrains

Since the verification is based on the simulation only, the goal is to design a virtual environment. For this purpose various terrain types need to be virtually simulated.

A terrain is defined by four parameters: *roughness*, *slipperiness*, *hardness* and *elasticity*. These parameters form a substance together (this process is described in appendix A2.1).

Besides these four parameters represented as a substance handle, a terrain constructor takes six more arguments (used in code part A2.1):

terrain_color : simulation ground color

"rough1.ppm" : an image in the .ppm format, a lowest common denominator color image file format (*PPM Format Specification*), a bitmap height file

"" : texture image (not used)

20 : walking area x-size

25 : walking area y-size

terrain_height : maximum terrain height

5.3.1 Terrain Features

Out of the listed ground parameters, some of them are picked up and being called *terrain features*, as they define a specific terrain type.

Therefore, a virtual terrain type is defined by five features. Each of them is a float number from an empirically stated range [1]. (Table 5.3).

Table 5.3: Terrain features and their ranges

	min value	min meaning	max value	max meaning
roughness	0.0	smooth	10.0	rough
slipperiness	0.0	friction	100.0	slippy
hardness	0.0	soft	100.0	hard
elasticity	0.0	rigid	2.0	elastic
height	0.0	low	0.1	high

5.3.2 Features Determination for Various Terrains

To determine a terrain type, one has to come up with the five parameters from Table 5.3.

In this work we use 14 terrain types. Their parameters (shown in Table 5.4) have been set up manually. With respect to the feature ranges from Table 5.3, the values have been normalized between 0 and 1.

Table 5.4: Parameters of virtual terrain types

#	*terrain title*	*roughness*	*slipperiness*	*hardness*	*elasticity*	*height*
1	**carpet**	0.3	0.0	0.4	0.15	0.2
2	**concrete**	1.0	0.0	1.0	0.0	0.0
3	**foam**	0.5	0.0	0.0	1.0	0.7
4	**grass**	0.5	0.0	0.3	0.3	0.5
5	**gravel**	0.7	0.001	1.0	0.0	0.3
6	**ice**	0.0	1.0	1.0	0.0	0.0
7	**mud**	0.05	0.05	0.005	0.25	0.2
8	**plastic**	0.1	0.02	0.6	0.5	0.0
9	**rock**	1.0	0.0	1.0	0.0	1.0
10	**rubber**	0.8	0.0	0.8	1.0	0.0
11	**sand**	0.1	0.001	0.3	0.0	0.2
12	**snow**	0.0	0.8	0.2	0.0	0.2
13	**swamp**	0.0	0.05	0.0	0.0	1.0
14	**wood**	0.6	0.0	0.8	0.1	0.2

A brief analysis of this setting has been performed in section 6.2.1.

[1]The upper range limits have been set up based on significant changes in the robot behaviour for various parameter values.

5.3.3 Terrain Noise

In general simulations are widely used coming with many benefits and being usually the right way to start, however, the real world is always different from the simulated one and these differences may influence the results significantly.

In this work, 14 terrain types have been simulated based on five features (Table 5.3). The parameters in Table 5.4 have been set up manually by an intuition. Therefore, one should take into account that the real terrains might be different from the virtual ones in some ways.

Secondly, if, for instance, there is a terrain defined as grass, this definition cannot be unique, since there are many types of grass and those differ from each other at least in the reffered features.

Consequently, the terrain parameters shown in Table 5.4 are noised. Regarding individual features and their upper limits from Table 5.3, the following Eq. (5.3) shows, how the noise is added.

For noise generation, the normal (Gaussian) distribution is used:

$$feature\_noise \sim N(\mu, \sigma^2)$$

$$feature\_noise = fRand(0, feature_{up\_limit} * std_p) \tag{5.3}$$

std_p : a standard deviation percentage, passed as a simulation argument

fRand() : a function generating a random float number using the normal (Gaussian) distribution with zero mean and a specified standard deviation defined by the feature's range and percentage (std_p)

For instance, assuming *roughness* as a feature, the feature upper limit equals 10.0 (Table 5.3). Then having the std_p equal 0.1 for example, the noise value is generated as a random number between -1 and 1.

Once the noise is generated, it is added to an original feature value (before normalization as shown in Table 5.4) as given in Eq. (5.4).

$$feature\_value += feature\_noise \tag{5.4}$$

Additionally, there is some limits checking as the parameters cannot take negative values. The final form is set as shown in Eq. (5.5).

$$feature\_value = max(feature\_value, 0) \tag{5.5}$$

5.4 Data Acquisition

The data comes from the 18 proprioceptive and 6 tactile sensors and one needs to find a way how to form feature vectors (classification samples) out of it (section 5.5), which is one of the most essential parts of the process.

As it is later described in more detail, several sensors values in time need to be used to obtain the robot's dynamics on various terrains. Therefore, to generate a single data sample candidate, the simulation must be run for a period of time. We use the 'candidate' significaiton as the optimal duration of one sample is not known. Samples are then formed out of sample candidates.

To gather the data sample candidates, the simulator is launched several times in order to generate several candidates for every terrain type. It has been chosen to let the robot walk for 10 seconds each time, which leads to 100 values per sensor for one run (see simulation settings in appendix A2.1).

For illustration, two examples of data candidates gathered from sensor *ATRf* when the robot was walking on a *concrete*, and on a *rock* respectively, for approximately 10 seconds is shown in Fig. 5.8.

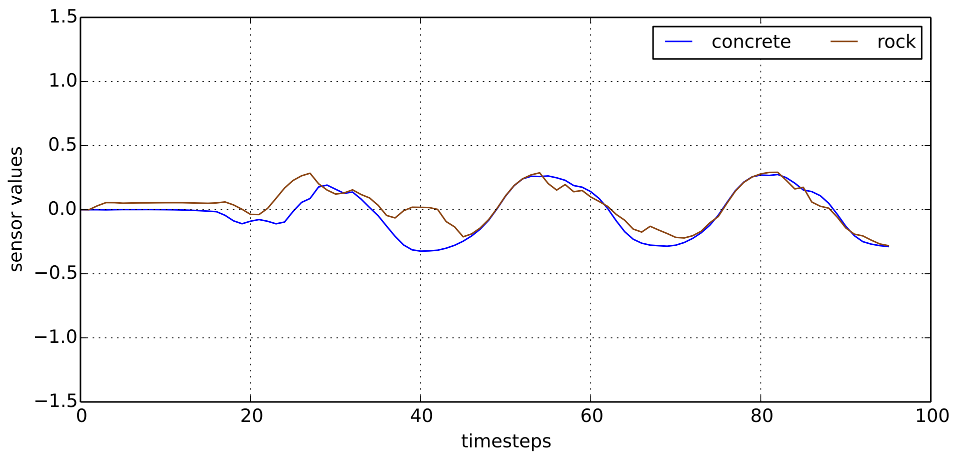

FIGURE 5.8: Data example: ATRf, concrete, approx. 10 seconds

More detailed results for all of the terrain types and several sensors are shown in section 6.2.2.

As an optimal standard deviation value of the additive terrain noise is not known, some data for several values of this parameter have been generated. The simulation has been gradually run for:

- $\sigma_p = 0.0$ (no noise)
- $\sigma_p = 0.01$ (1% relative noise)
- $\sigma_p = 0.03$ (3% relative noise)
- $\sigma_p = 0.05$ (5% relative noise)
- $\sigma_p = 0.1$ (10% relative noise)
- $\sigma_p = 0.2$ (20% relative noise)

The σ_p corresponds to the std_p parameter used in Eq. (5.3). The influence of additive terrain noise is analyzed in section 6.2.4.

The approach of storing the gathered data is described in appendix A2.1. As Fig. A2.4 shows, 500 sample candidates are generated for every *noise/terrain* configuration. This allows creating datasets of 500 samples per class.

5.5 Building a Feature Vector

Classification tasks are generally based on datasets consisting of samples and corresponding targets. The samples need to be represented in a numerical way in order to be processed by a computer and its appropriate algorithms. In machine learning, this numerical representation of an object is called *a feature vector*, an n-dimensional vector of numerical values. This section is devoted to building a feature vector out of the data gathered from proprioceptive and tactile sensors.

As the optimal structure is not known, several possibilities are tested and therefore some new global process parameters appear at this point (mentioned already in section 5.1).

For this particular problem, the task is to form one feature vector out of the content of one stored data file (see appendix A2.1), as each of these files contains data for one sample (see Fig. 5.9).

FIGURE 5.9: Forming a feature vector out of a data file.

It is assumed that a proper terrain classification using proprioceptors at one moment in time is at least difficult, if not impossible. Therefore the idea is to let the robot walk for a while and take down the dynamics of the sensors. Of course, the more timesteps are used for one sample, the more time the classification takes. Because of these arguments the number of timesteps is left as a global process parameter and it is a subject for later discussion.

Sensor selection defines another global process parameter. The anticipation is that the feature vector becomes redundant using all of the 24 sensors, as many of them may contain similar information. However, for now all of them are used to show how the feature vector is built and it is also left for later discussion.

With reference to Fig. 5.9, feature vectors have been constructed by fixing the *timesteps* parameter and concatenating columns of the matrix into one vector. This results into having data from all sensors one by one next to each other and forming one feature vector together.

In Fig. 5.10 an illustration of the vector formation for three terrain types is shown. The number of timesteps is set to 40 and all 24 sensors are used, hence a feature vector of length 960 is obtained. The corresponding sensor abbreviations (see Table 5.1) are added to the x-axis annotation. The 18 angle sensors are followed by the 6 foot contact sensors.

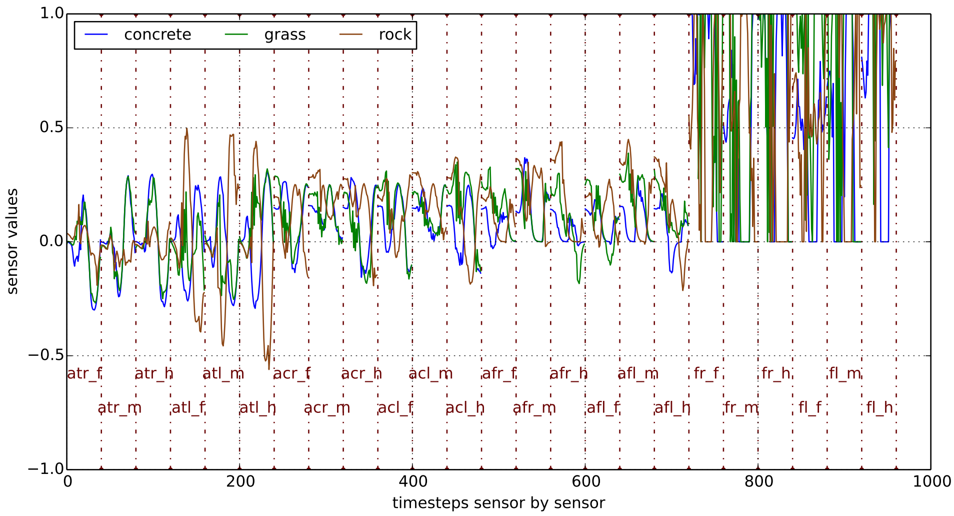

FIGURE 5.10: Example of feature vector (raw data), 40 timesteps, no signal noise

5.5.1 Feature Vector Normalisation

It is a good manner to keep the data normalised - mapped to $[0.0, 1.0]$ interval. The default range of foot contact sensors is already set to $[0.0, 1.0]$, so there is nothing to change. For the joint angle sensors, the following approach, sometimes called *feature scaling*, is used to map the data.

For each element S_i of signal S:

$$S'_i = \frac{S_i - r_{min}}{r_{max} - r_{min}} \tag{5.6}$$

r_{min}, r_{max} : bounds of the corresponding original sensor range (listed in Table 5.1)

S'_i : scaled element of the normalised signal

Also a $[0, 1]$ interval overflow checking is added and values are adjusted if needed (Eq. (5.7)). This is a cover for the case when ranges from Table 5.1 were not accurate.

$$S'_i = min(max(S'_i, 0), 1) \tag{5.7}$$

The following figure (5.11) shows normalised feature vector examples for three terrains. The influence of normalisation on classification results is another subject for the discussion.

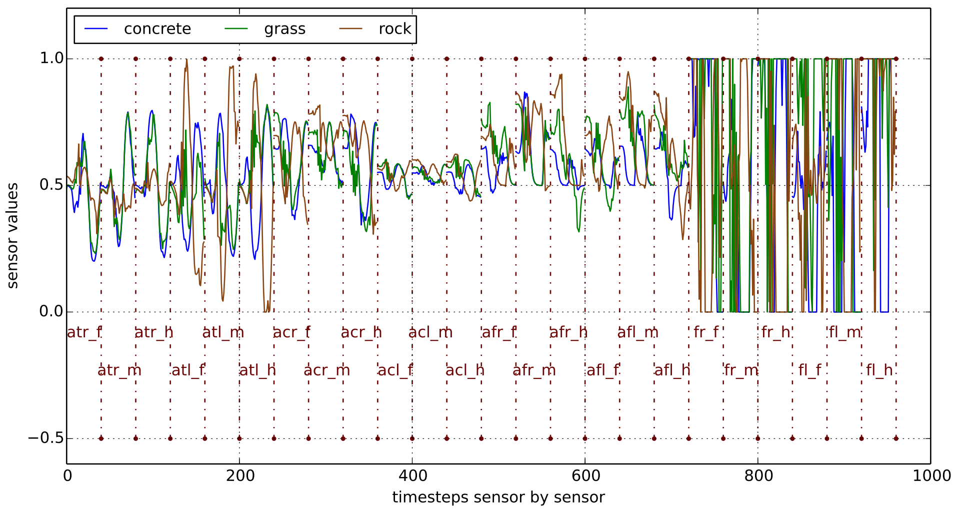

FIGURE 5.11: Normalised feature vector examples

5.5.2 Signal Noise

In section 5.3.3 a few general reasons for adding a noise to simulation data were disscused. In that case an additive Gaussian noise is used to generate variability in the data and to make the terrain types definitions (from Table 5.1) more general.

For similar reasons a signal noise is also added to the sensory data. In reality the data obtained from mechanical sensors are noisy (environmental conditions, failures of electrical devices, etc.), while the data coming from the simulated sensors are always deterministic.

In this case, a white Gaussian noise is added to the normalised feature vectors. Similarly to equations in section 5.3.3, at first a noise is generated using the normal distribution with zero mean and specified standard deviation. This time, a vector of length n needs to be generated as a noise.

$$signal\_noise = [sn_1, sn_2, ..., sn_n] \tag{5.8}$$

$$sn_i \sim N(\mu, \sigma^2), \quad i = 1, 2, ..., n \tag{5.9}$$

Then, the generated vector is added to a normalised feature vector from section 5.5.1 (Eq. (5.10)).

$$noised\_signal_i = raw\_signal_i + sn_i, \quad i = 1, 2, ..., n \tag{5.10}$$

Finally, the noised signal is checked, whether its values do not overflow out of the $[0, 1]$ range.

$$noised\_signal_i = min(max(noised\_signal_i, 0), 1), \quad i = 1, 2, ..., n \tag{5.11}$$

Also in this case, it is difficult to estimate an optimal signal noise power (standard deviation of the normal distribution). Therefore it is left as another global process parameter and its influence is discussed in the results part. It is defined as a percentage of the $[0.0, 1.0]$ interval and as the signals are normed in advance, there is no need for another processing of this parameter.

5.6 Creation of Datasets

In this section, the task is to transform all the data into so called datasets. There are usually three sets of data used for classification tasks - training, validation and testing data. These three sets must be disjunctive, meaning they cannot have a single element in common. All these three sets together form a dataset.

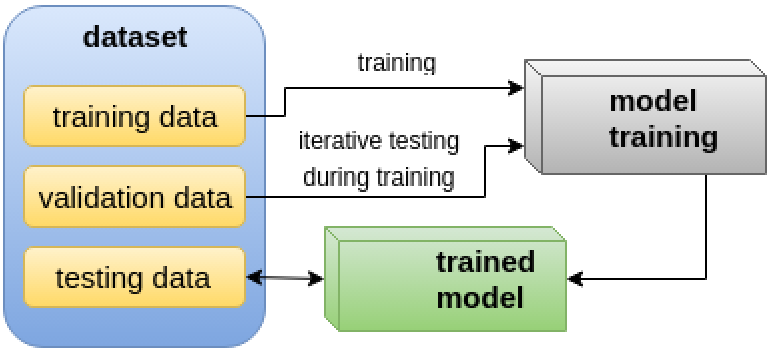

FIGURE 5.12: Three sets of data in a dataset.

Each set of data consists of samples and targets (class labels). The samples are represented by normalised feature vectors (section 5.5) - lists of numerical floating point values from $[0.0, 1.0]$ interval. Every sample must be uniformly assigned to precisely one target. The targets, in this case, match the virtually created terrain types (listed in Table 5.4) in the following manner.

The target vector is of length 14, as there are 14 terrain types. Every terrain type has an unique identificator (numbers listed in Table 5.1) corresponding to positions in the target vector. In any case, the vector contains 13 'zeros' and 1 'one'. The vector is then matched to a terrain type depending on the position where the 'one' is. For instance, a target vector corresponding to *concrete* is illustrated on Fig. 5.13.

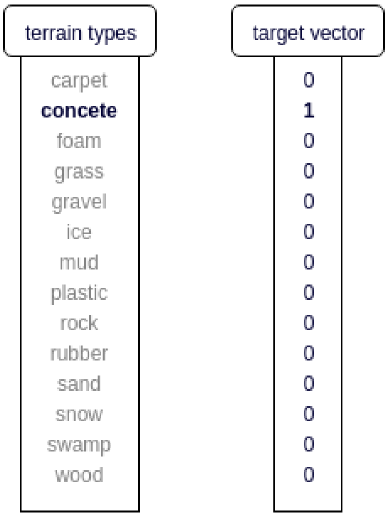

FIGURE 5.13: Target vector for concrete

Once there are two ordered lists - a list of samples and a corresponding list of targets, these lists are split into the three sets shown in Fig. 5.12. There is a parameter called *data_split_ratio* defining the proportions among the sets sizes. By default the ratio is set to generate 80% training, 10% validation and 10% of testing data.

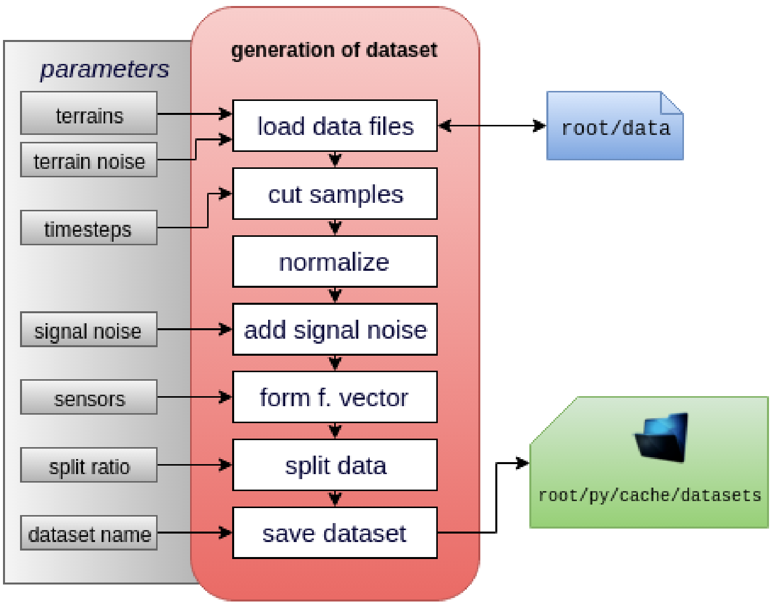

FIGURE 5.14: Workflow of generating a dataset

The workflow of data generation procedure is illustrated in Fig. 5.14.

The list of all generated datasets can be found in Table 6.1. These datasets and influence of individual parameters are evaluated in chapter 6.

5.7 Training and Classification

Having a dataset enables to train a classifier, a machine learning tool that is able to learn some behavior on one part of some data (training and validation) and then perform similarly on another "never seen" part of the data (testing) - as shown in Fig. 5.12.

There are many classification methods differing in mathematical backgrounds and each of them has some adventages and disadvantages on various types of data. However, all of them have some general functionalities that comply with some kind of convention. For instance, there are at least two procedures that every classifier should be capable of:

model fitting : In this procedure, an initialized classifier is usually given training samples and their corresponding targets. Additiionally, it can take some validation data and/or learning parameters. Then a model is trained using some math behind the selected classification method.

unlabeled observation prediction : Once the model is trained, it is capable of predicting classes of unlabeled samples. It takes one or more samples of testing data and returns the predicted target(s).

This convention enables testing different classification approaches on the same data in the same way. Therefore also the implemented network library *kitt_nn* (see chapter 4) provides these functions and is capable of working with datasets of the same structure as the public *.py* classifiers (discussed in sections A2.2 and 5.7.4).

In the overall process diagram (Fig. 5.1) there is a box called *classification with full networks*. The procedure behind this box is illustrated on Fig. 5.15.

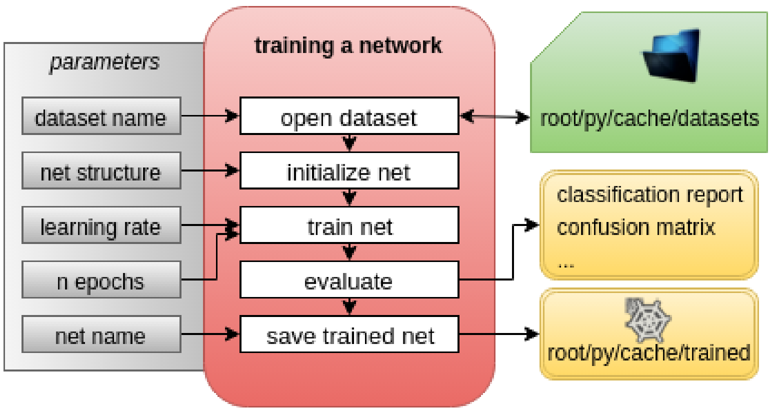

FIGURE 5.15: Procedure of training and testing a network

This workflow is performed by the implemented framework *kitt_nn* (chapter 4, as well as by other provided classifiers (discussed in section 5.7.4) for comparison. It is adventageous, that each of these tools can use the same workflow and so the comparison is fair.

There are several arguments (firstly listed in section 5.1) that differentiate the final trained networks and their performances. The first one is the dataset that the network is trained on. This parameter brings its own configuration (see its input parameters in Fig. 5.14) and so its setting parametrizes the classifier as well.

Next, one needs to define the network initial structure in sense of number of hidden layers and number of neurons in each of these layers. The input and output layers are determined by the dataset. There are many parameters to be defined for learning like *batch size*, *initial random state* etc. In this work, only the learning rate and the number of epochs are used as training parameters. The learning process follows the implemented backpropagation algorithm described in section 4.2.

5.7.1 Evaluation Methods

A trained network is evaluated on testing data. This evaluation provides a set of the most important classification metrics ("Scikit-learn: Machine Learning in Python").

accuracy : the set of labels predicted for a sample must exactly match the corresponding set of true labels

precision : ability of the classifier not to label as positive a sample that is negative

recall : the ability of the classifier to find all the positive samples

F1 score is interpreted as a weighted average of the precision and recall, where an F1 score reaches its best value at 1 and worst score at 0. The relative contribution of precision and recall to the F1 score are equal. Formula:

$$F1 = \frac{2 * precision * recall}{precision + recall} \tag{5.12}$$

confusion matrix : a confusion matrix C is such that $C_{i,j}$ is equal to the number of observations known to be in group i but predicted to be in group j.

5.7.2 Terrain Classification using Network Pruning

As the overall process diagram (**??**) shows, the developed network pruning algorithm (**??**) is tested on the terrain datasets. The approach has been already described.

Evaluation in... **TODO:** describe the process here and reffer the results after they are gathered.

5.7.3 Searching for Optimal Configuration (Grid Search)

TODO : describe here how the best parameters have been found using GridSearch datasets implication -> nets based on nets params (learning rate and number of epochs), fixed batch size, number of stable iterations ....

5.7.4 Other Classifiers

TODO : describe here how other classifiers have been tested and reffer to the results part

SVM, k-NN, RandomForest

Chapter 6

Experimental Evaluation

The account of the research should be presented in a manner suitable for the field. It should be complete, systematic, and sufficiently detailed to enable a reader to understand how the data were gathered and how to apply similar methods in another study. Notation and formatting must be consistent throughout the thesis, including units of measure, abbreviations, and the numbering scheme for tables, figures, footnotes, and citations. One or more chapters may consist of material published (or submitted for publication) elsewhere. See “Including Published Material in a Thesis or Dissertation” for details.

6.1 Verification of Network Implementation

6.2 Terrain Processing Results

6.2.1 Analysis of Terrain Similarity

In the following a brief analysis of terrain similarities is presented. In general, a (dis-) similarity between terrains should correlate with classification results, i.e., the more two terrains differ from each other the better classification results are expected, and vice versa.

In order to quantify and visualise similarity among various terrains, a similarity measure was calculated as given in Eq. (6.1). The five qualities are listed in Table 5.3 and in Table 5.4.

$$SF_{t_1,t_2} = \frac{\sum_{i=1}^{5} |quality(i,t_1) - quality(i,t_2)|}{5} \tag{6.1}$$

The similarity measure equals 0 if two terrains are identical (have the same parameter values) and equals 1.0 if two terrains are totally different.

The following Fig. 6.1 shows the similarity measures among generated terrains.

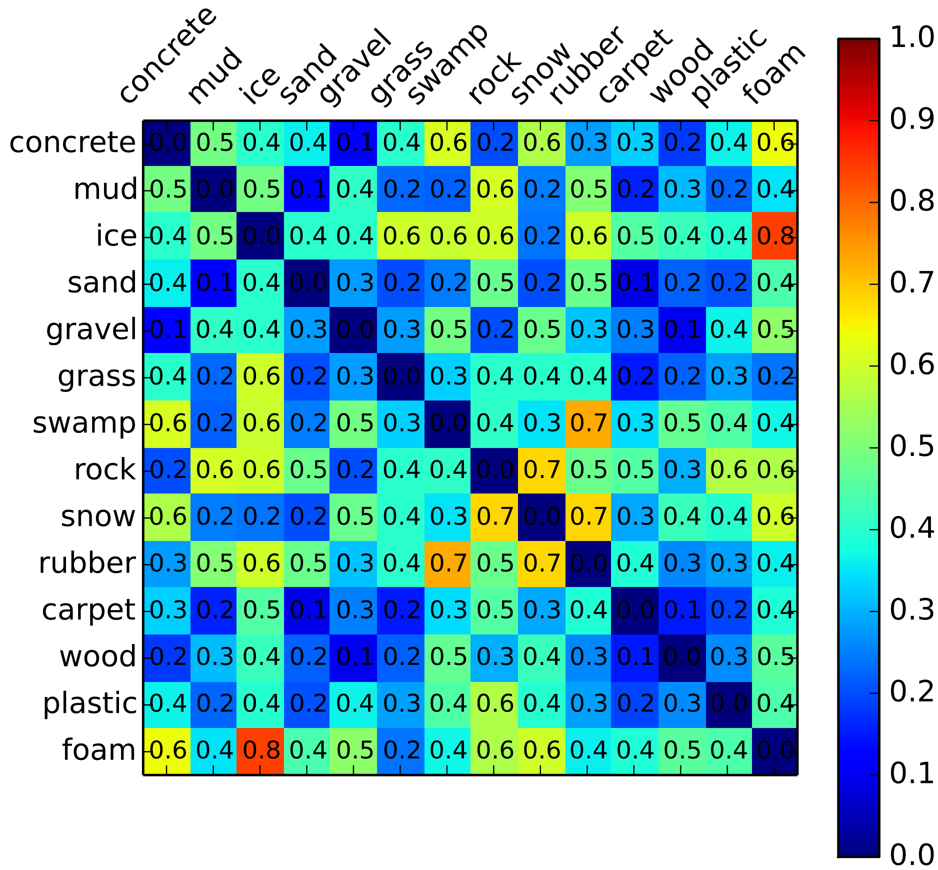

FIGURE 6.1: Similarity measures among various terrain types.

The surfaces have been generated virtually and their similarity to real world terrains has not been verified. However, results demonstrate that foam is very different from ice or, for instance, sand is quite similar to mud. A low similarity measure can be seen among concrete, carpet and rubber as all of them are moreless flat.

Surprisingly, a wooden terrain ended up as very similar to gravel and carpet, but a limited number of simulated terrain features has to be considered. On the other hand, rubber results as very different from swamp and snow, which is a positive outcome. Also the high grass-ice or rock-snow similarity measures make sense.

6.2.2 Gathered Data

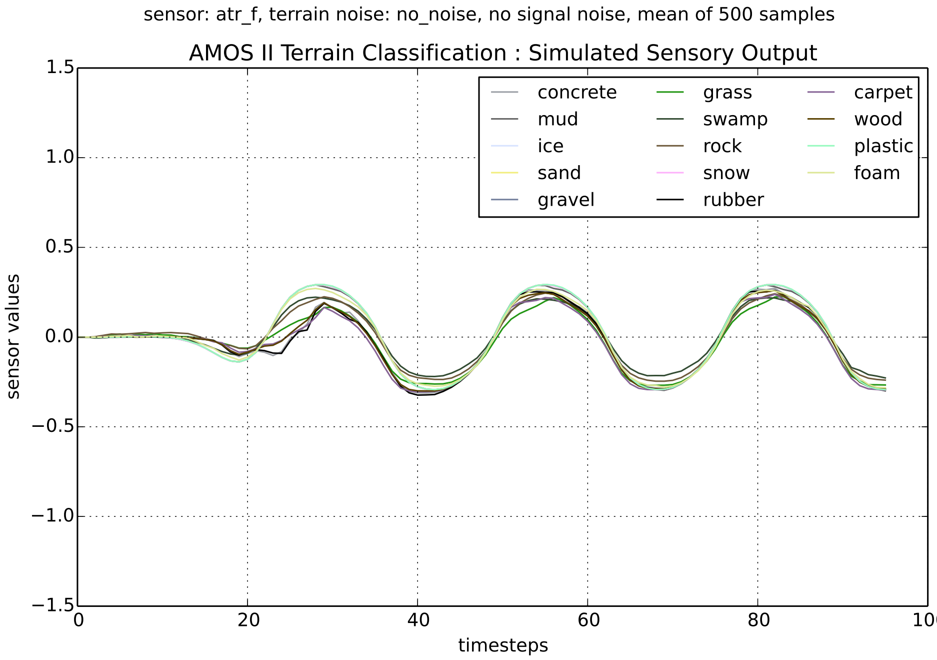

FIGURE 6.2: Sensor ATRf : mean of 500 samples, 14 terrains

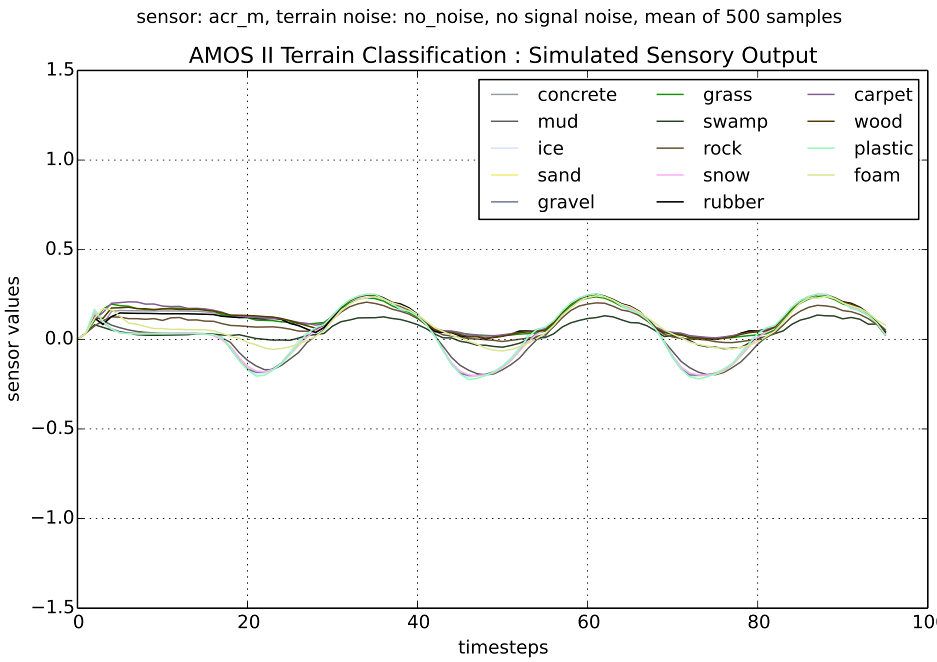

FIGURE 6.3: Sensor ACRm : mean of 500 samples, 14 terrains

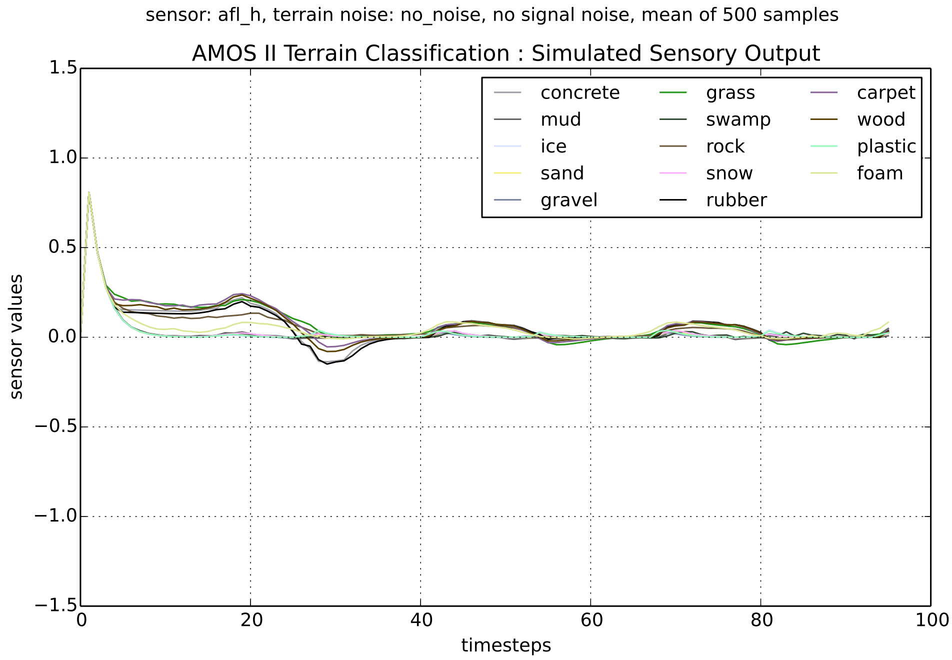

Figure 6.4: Sensor AFLh : mean of 500 samples, 14 terrains

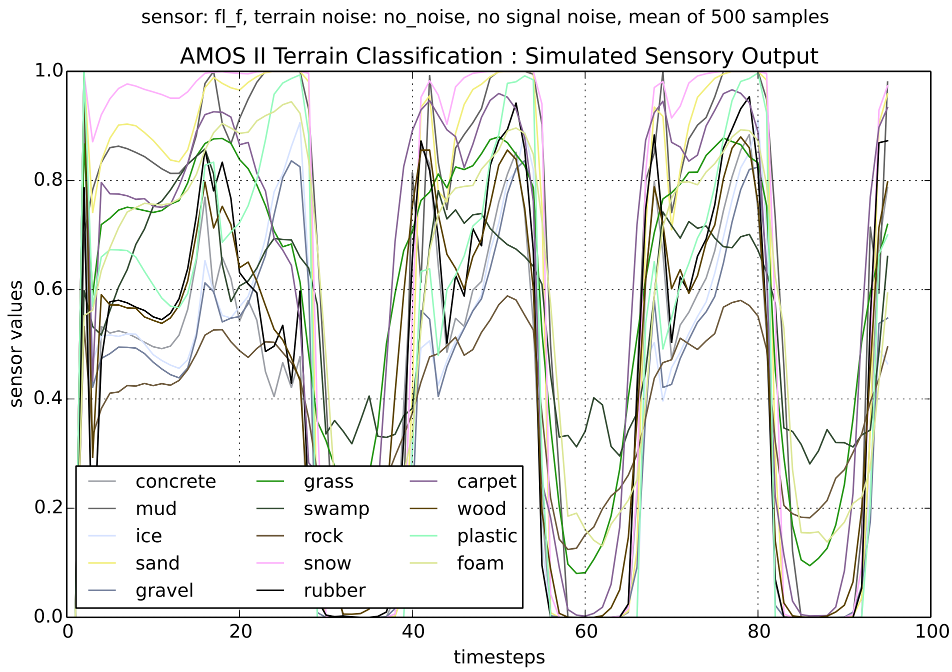

Figure 6.5: Sensor FLf : mean of 500 samples, 14 terrains

6.2.3 Built Feature Vector

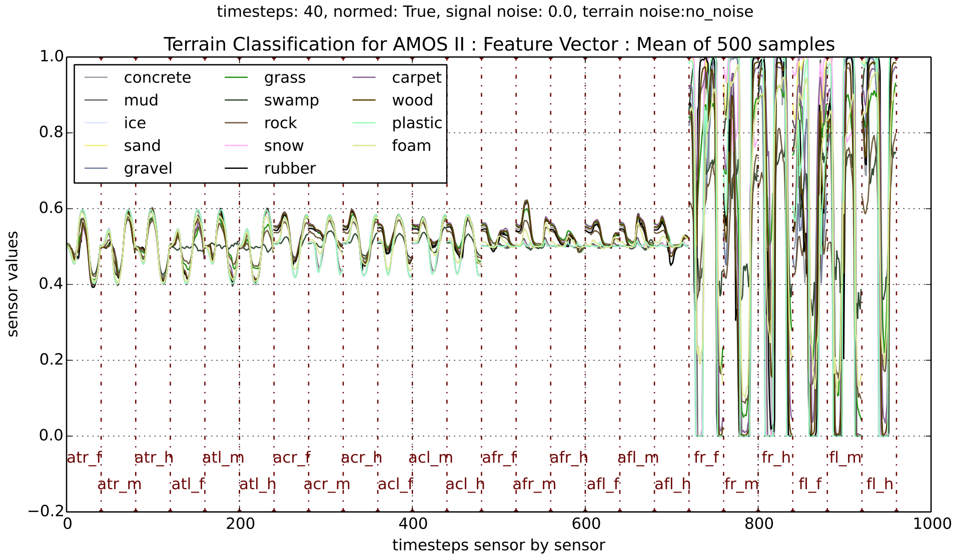

FIGURE 6.6: Feature Vector : mean of 500 samples, 14 terrains, no noise, 40 timesteps

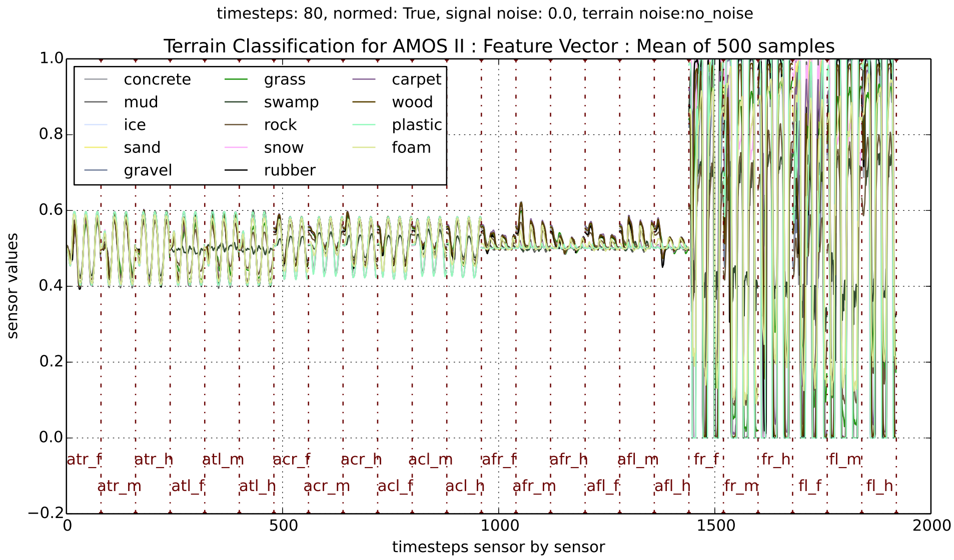

FIGURE 6.7: Feature Vector : mean of 500 samples, 14 terrains, no noise, 80 timesteps

6.2.4 Terrain Noise Influence

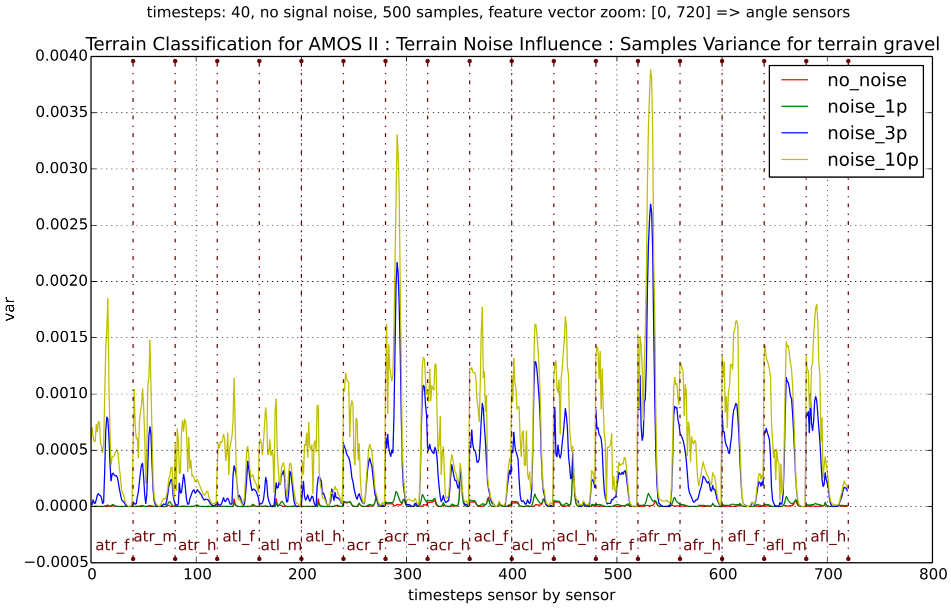

FIGURE 6.8: Terrain Noise Analysis (samples variance): 500 samples, terrain gravel, angle sensors (feature vector [0:720] for 40 timesteps)

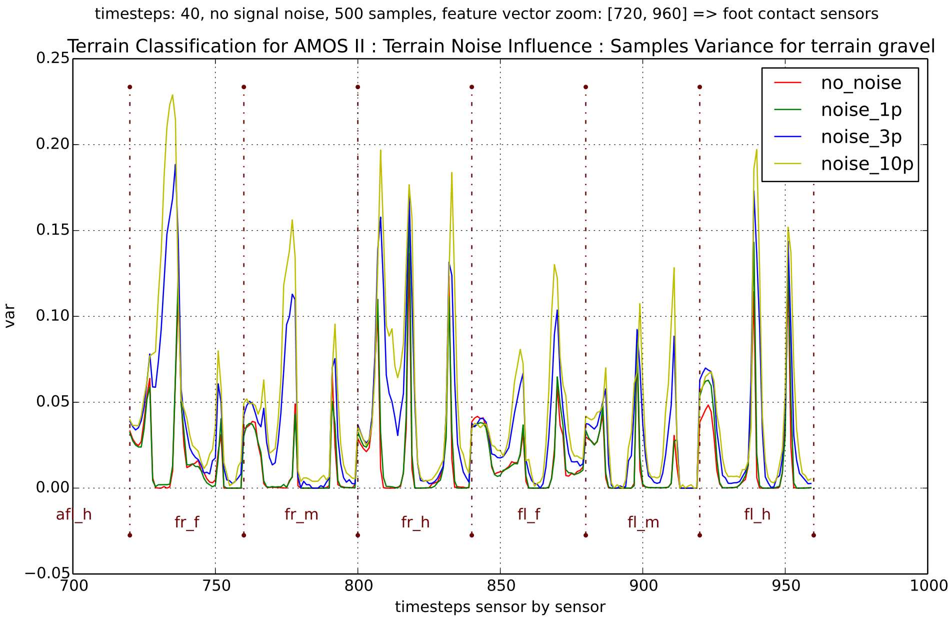

FIGURE 6.9: Terrain Noise Analysis (samples variance): 500 samples, terrain gravel, foot contact sensors (feature vector [720:960] for 40 timesteps)

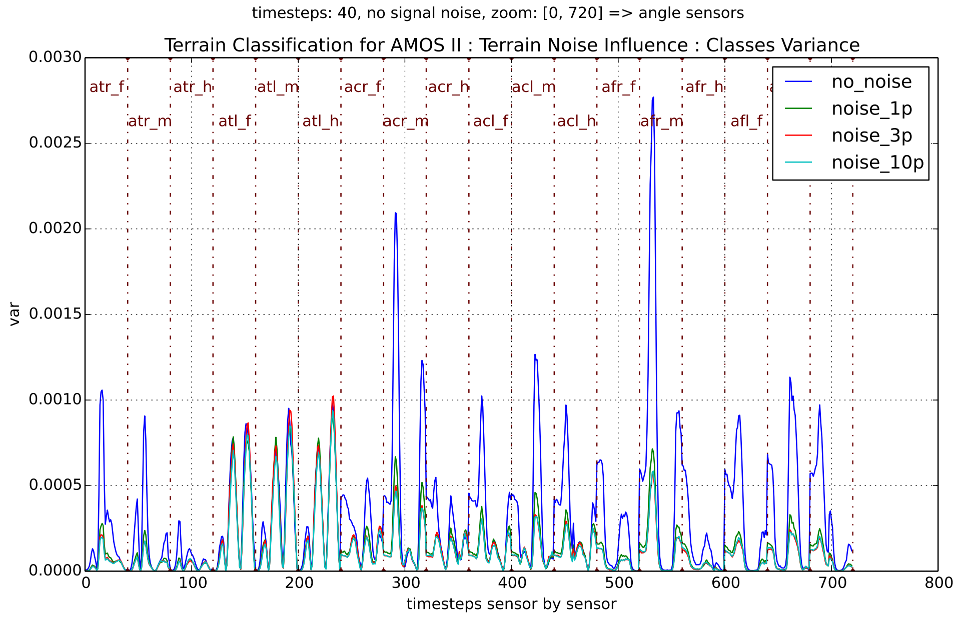

FIGURE 6.10: Terrain Noise Analysis (classes variance): means of 500 samples, 14 terrains, angle sensors

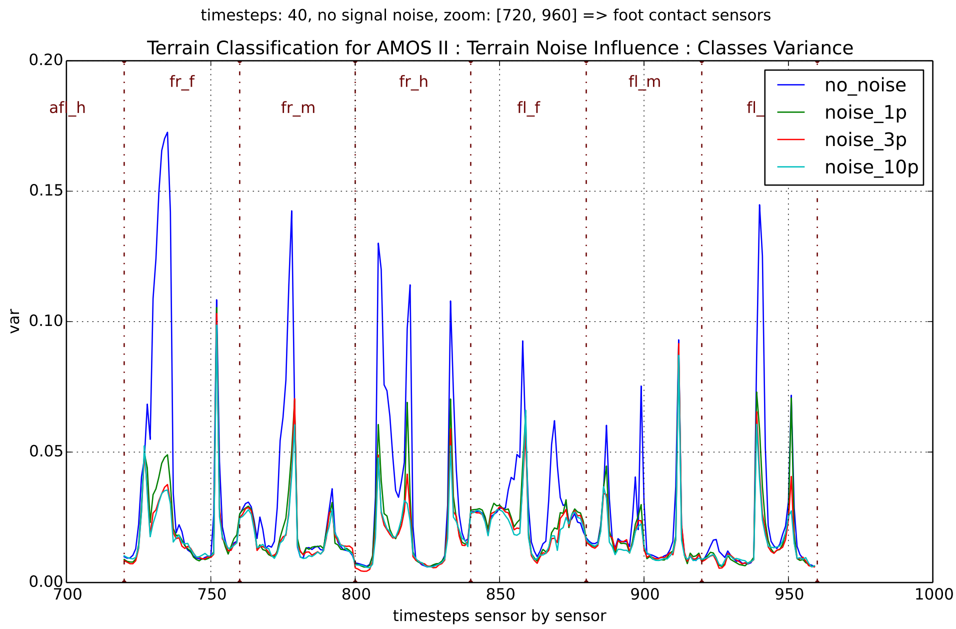

FIGURE 6.11: Terrain Noise Analysis (classes variance): means of 500 samples, 14 terrains, foot contact sensors

6.2.5 Signal Noise Influence

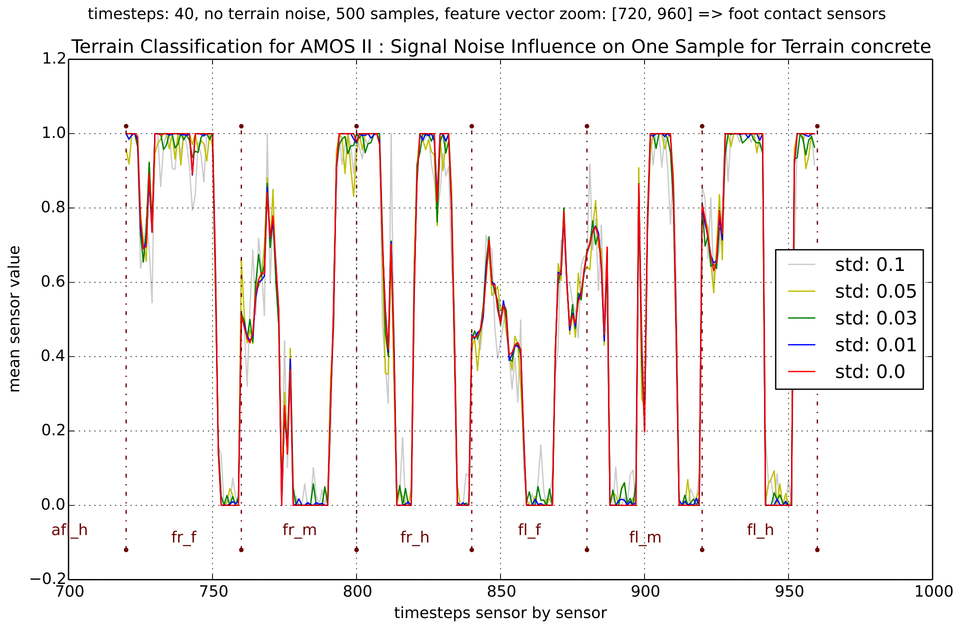

FIGURE 6.12: Signal noise influence on one sample, terrain: concrete, angle sensors

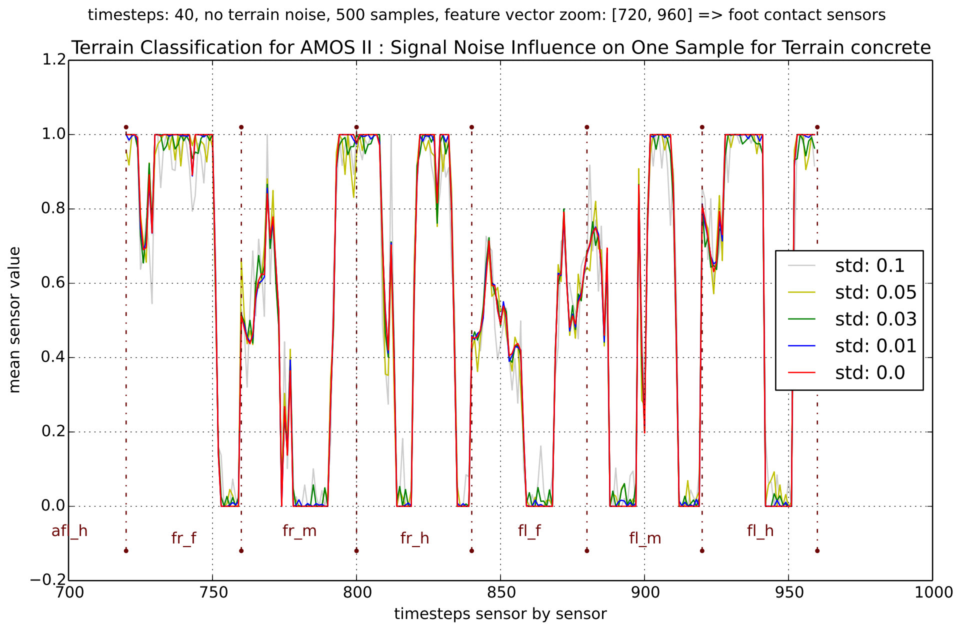

FIGURE 6.13: Signal noise influence on one sample, terrain: concrete, foot contact sensors

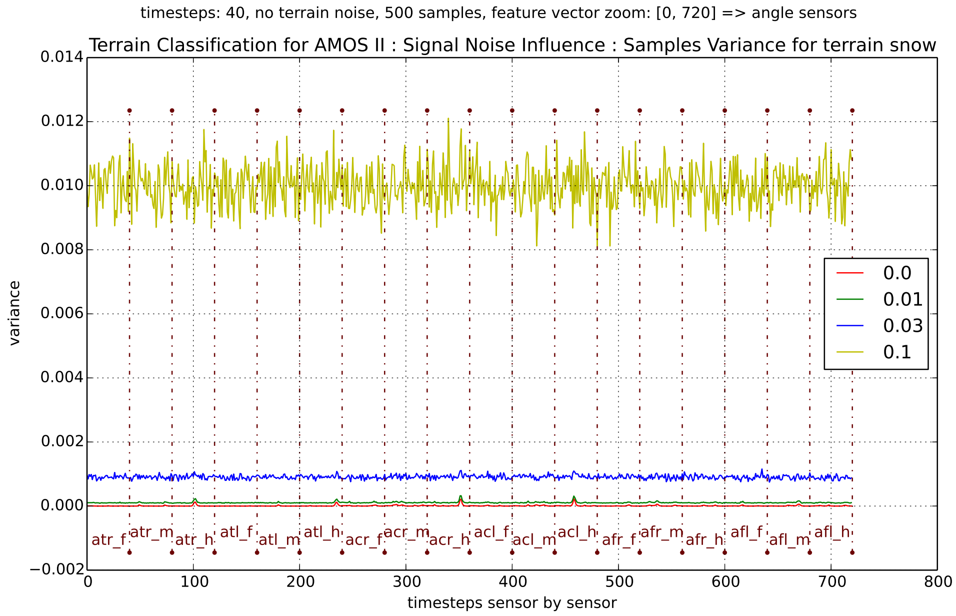

FIGURE 6.14: Signal noise analysis : samples variance, terrain: snow, angle sensors

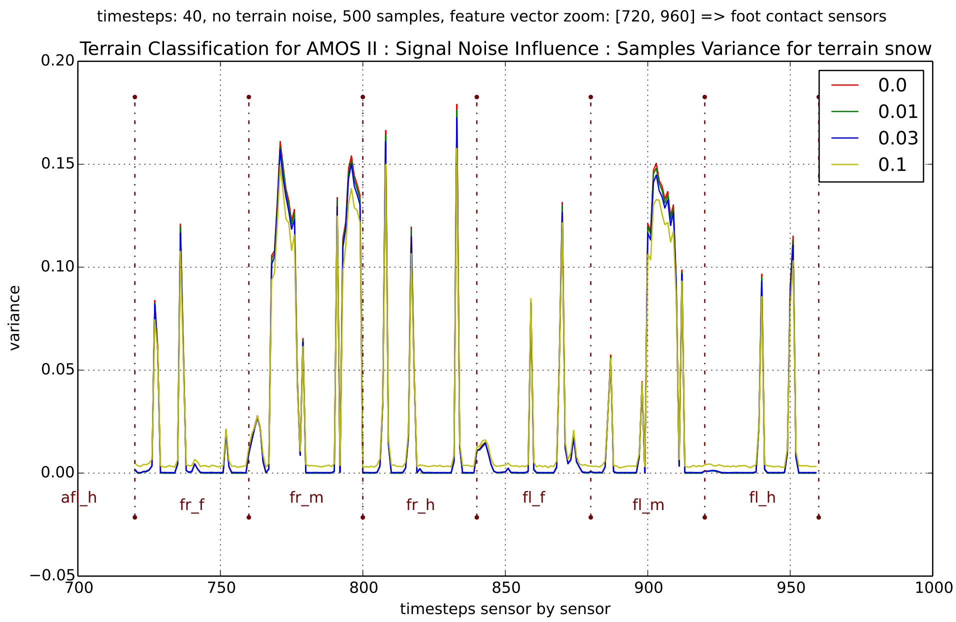

FIGURE 6.15: Signal noise analysis : samples variance, terrain: snow, foot contact sensors

6.2.6 Generated Datasets

Table 6.1: Generated datasets

name	*ter. noise*	*sig. noise*	*timesteps*	*sensors*	*terrains*	*samples*
ds_01	0.0	0.0	40	all	all	500
ds_02	0.0	0.0	40	angle	all	500
ds_03	0.0	0.0	40	foot	all	500
ds_04	0.0	0.0	1	all	all	500
ds_05	0.0	0.0	10	all	all	500
ds_06	0.0	0.0	80	all	all	500
ds_07	0.0	0.01	40	all	all	500
ds_08	0.0	0.03	40	all	all	500
ds_09	0.0	0.05	40	all	all	500
ds_10	0.0	0.1	40	all	all	500
ds_11	0.01	0.0	40	all	all	500
ds_12	0.03	0.0	40	all	all	500
ds_13	0.05	0.0	40	all	all	500
ds_14	0.1	0.0	40	all	all	500
ds_15	0.2	0.0	40	all	all	500

6.2.7 Classification results

Table 6.2: Classification results

dataset	***kitt net***			***sknn net***		
	accuracy	*precision*	*recall*	*accuracy*	*precision*	*recall*
ds_01						
ds_02						
ds_03						
ds_04						
ds_05						
ds_06						
ds_07						
ds_08						
ds_09						
ds_10						
ds_11						
ds_12						
ds_13						
ds_14						
ds_15						

6.2.8 Final Configuration

6.3 Pruning Algorithm Results

evaluation (tables and figures) of classification:

- various terrain noise standard deviation values
- various signal noise standard deviation values
- various sensors on network input (only foot, only angle...)
- various timesteps used as one sample (-> time needed for detection)
- various number of detected terrains as outputs
- various network structures
- various training parameters (epochs, learning rate, batch size...)

evaluation of neural nets as a classifier:

- comparison to other classifiers on the same data, classifiers are ready provided by sknn library

evaluation of proprioception sensing against other methods (visual, haptic, laser...):

- comparison to the results from the literature

evaluation of the pruning algorithm:

- various starting structures, ends up with the same minimal-optimal structure?
- various noise types, same minimal structure?
- speed comparisons of the fully-connected vs. pruned structure
- further analysis:
  - which sensors are redundant/crucial
  - which sensors are important for which terrain
  - comments on the minimal structure and benefits of having it

10-15 pages (many figures, tables)

Chapter 7

Discussion

Chapter 8

Conclussion and Outlook

In this section the student must demonstrate his/her mastery of the field and describe the work's overall contribution to the broader discipline in context. A strong conclusion includes the following:

Conclusions regarding the goals or hypotheses presented in the Introduction, Reflective analysis of the research and its conclusions in light of current knowledge in the field, Comments on the significance and contribution of the research reported, Comments on strengths and limitations of the research, Discussion of any potential applications of the research findings, and A description of possible future research directions, drawing on the work reported. A submission's success in addressing the expectations above is appropriately judged by an expert in the relevant discipline. Students should rely on their research supervisors and committee members for guidance. Doctoral students should also take into account the expectations articulated in the University's "Instructions for Preparing the External Examiner's Report".

2-3 pages

All references:

("Visual terrain classification for selecting energy efficient gaits of a hexapod robot") and ("Obstacle-Gap Detection and Terrain Classification of Walking Robots based on a 2D Laser Range Finder") and ("Neuromechanical control for hexapedal robot walking on challenging surfaces and surface classification") and ("Online learning terrain classification for adaptive velocity control") and ("Fundamentals and Methods of Terrain Classification Using Proprioceptive Sensors") and ("Haptic terrain classification for legged robots") and ("An Intelligent Architecture for Legged Robot Terrain Classification Using Proprioceptive and Exteroceptive Data") and ("Terrain identification for RHex-type robots") and ("Performance analysis and terrain classification for a legged robot over rough terrain") and ("Pruning Algorithms - A Survey") and (*The Bio-Inspired SCORPION Robot: Design, Control and Lessons Learned, Climbing and Walking Robots: towards New Applications*) and ("Gait control of the six-legged robot on a rough terrain using computational intelligence learning and optimization methods")

Bibliography

[Ree93] R. Reed. “Pruning Algorithms - A Survey”. In: *IEEE Transactions on Neural Networks (Volume:4 , Issue: 5)* (Sept. 1993), pp. 740–747. URL: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=248452.

[Wik04] Wikipedia. *Plagiarism — Wikipedia, The Free Encyclopedia.* [Online; accessed 04-May-2016]. 2004. URL: \url{http://en.wikipedia.org/w/index.php?title=Plagiarism&oldid=5139350}.

[SK07] D. Spenneberg and F. Kirchner. *The Bio-Inspired SCORPION Robot: Design, Control and Lessons Learned, Climbing and Walking Robots: towards New Applications.* ISBN 978-3-902613-16-5, 2007.

[Coy10] E. Coyle. “Fundamentals and Methods of Terrain Classification Using Proprioceptive Sensors”. PhD thesis. Florida State University Tallahassee, 2010.

[Hoe+10] M. A. Hoepflinger et al. “Haptic terrain classification for legged robots”. In: *Robotics and Automation (ICRA), IEEE International Conference* 3 (May 2010), pp. 2828–2833. URL: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=5509309.

[MK10] W. Mou and A. Kleiner. “Online learning terrain classification for adaptive velocity control”. In: *Safety Security and Rescue Robotics* 26 (July 2010), pp. 1–7. URL: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=5981563.

[Bel11] Dominik Belter. “Gait control of the six-legged robot on a rough terrain using computational intelligence learning and optimization methods”. PhD thesis. Poznan University of Technology, Nov. 2011.

[Ped+11] F. Pedregosa et al. “Scikit-learn: Machine Learning in Python”. In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[Ber+12] F. L. G. Bermudez et al. “Performance analysis and terrain classification for a legged robot over rough terrain”. In: *IEEE/RSJ International Conference on Intelligent Robots and Systems* 7 (Dec. 2012). URL: http://ieeexplore.ieee.org/xpl/login.jsp?tp=&arnumber=6386243.

[Kes+12] P. Kesper et al. “Obstacle-Gap Detection and Terrain Classification of Walking Robots based on a 2D Laser Range Finder”. In: *Nature-inspired Mobile Robotics* (2012), pp. 419–426. URL: http://manoonpong.com/paper/2013/CLAWAR2013_Kesper.pdf.

[Ord+13] C. Ordonez et al. “Terrain identification for RHex-type robots”. In: *Unmanned Systems Technology XV* 17 (May 2013). URL: `http://proceedings.spiedigitallibrary.org/proceeding.aspx?articleid=1689675`.

[Zen+13] S. Zenker et al. “Visual terrain classification for selecting energy efficient gaits of a hexapod robot”. In: *International Conference on Advanced Intelligent Mechatronics* 12 (July 2013), pp. 577–584. URL: `http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6584154&tag=1`.

[XWM14] X. Xiong, F. Worgotter, and P. Manoonpong. “Neuromechanical control for hexapedal robot walking on challenging surfaces and surface classification”. In: *Robotics and Autonomous Systems* 7 (Aug. 2014), pp. 1777–1790. URL: `www.elsevier.com/locate/robot`.

[Ahm15] Mohammed Nour Abdel Gwad Ahmed. “An Intelligent Architecture for Legged Robot Terrain Classification Using Proprioceptive and Exteroceptive Data”. PhD thesis. University of Bremen, June 2015.

[CS15] Alex J. Champandard and Spyridon Samothrakis. *sknn: Deep Neural Networks without the Learning Cliff.* [Online; accessed 06-May-2016; nucl.ai Conference 2015]. 2015. URL: `\url{http://scikit-neuralnetwork.readthedocs.io/en/latest/}`.

[Man] Poramate Manoonpong. “Adaptive Embodied Locomotion Control Systems”. Lecutre 3 - page 133 - Tripod Gait.

[Misa] *Open-source multi sensori-motor robotic platform AMOS II.* `http://manoonpong.com/AMOSII.html`.

[Misb] *PPM Format Specification.* `http://netpbm.sourceforge.net/doc/ppm.html`. Updated: 02 November 2013.

[Misc] *Research Network for Self-Organization of Robot Behavior.* `http://robot.informatik.uni-leipzig.de/software/`. last modified: 06. July 2015.

Appendix A1

Working Directory Structure

the mt_folder

Appendix A2

Implementation Details

A2.1 Terrain Classification Implementation

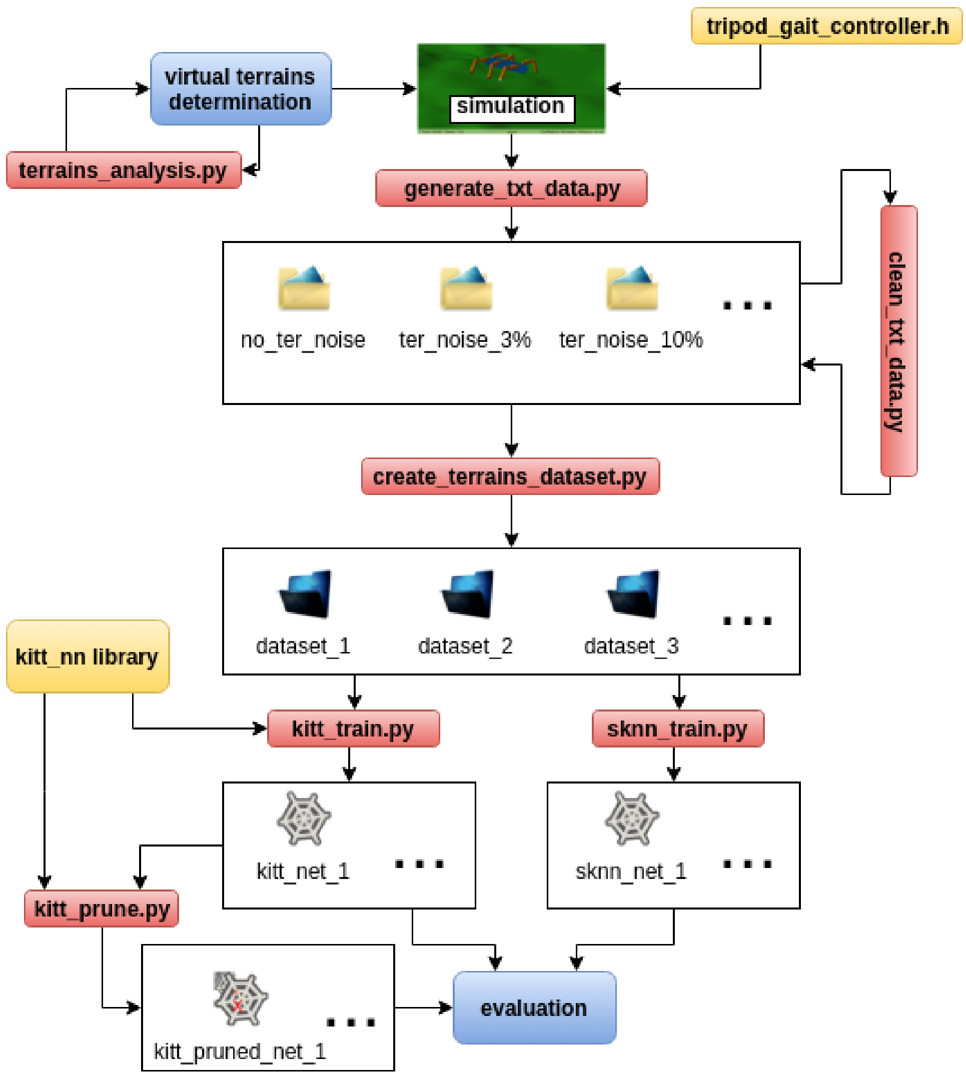

FIGURE A2.1: Terrain classification process - overall diagram.

LPZ Robots Simulation

The *lpzrobots* project contains many subprojects. For this study, the most important ones are:

selforg : homeokinetic controllers implementation framework

ode_robots : a 3D physically correct robot simulator

The project is implemented in *C++* and needs a Unix system to be run. It consists of two main GIT repositories to be forked - lpzrobots and go_robots. The overall software architecture is shown in Fig. A2.2.

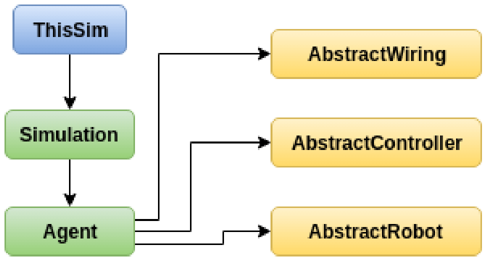

FIGURE A2.2: Software architecture for LPZRobots and GoRobots. (*Research Network for Self-Organization of Robot Behavior*)

To introduce the elements in Fig. A2.2, *ThisSim* is an inherited class of another class called *Simulation* and is initialized everytime the simulation is launched. It integrates all elements together, controls the environment as well as the robot and sets up initial parameters. An instance of the *Agent* class integrates all components of the agent (robot) by using the shown classes.

Terrain Construction in main.cpp

The **LpzRobots** AMOS II simulator supports some terrain setting. In the main simulation file (*main.cpp* - see A3), a *'rough terrain'* substance is being initialized and passed through a handle to a *TerrainGround* constructor.

PART OF CODE A2.1: Setting a terrain ground in main.cpp

```
Substance roughterrainSubstance(terrain_roughness, terrain_slip,
                       terrain_hardness, terrain_elasticity);
oodeHandle.substance = roughterrainSubstance;
TerrainGround* terrainground = new TerrainGround(oodeHandle,
                       osgHandle.changeColor(terrain_color),
                       "rough1.ppm", "", 20, 25, terrain_height);
```

Data Storing

It is always recommended to store rough data before some processing, hence the simulator creates *.txt* files of structure symbolized in code part A2.2 (with the reference to sensors shortcuts in Table 5.1).

PART OF CODE A2.2: Rough sensory data files structure

```
timestep_001;ATRf;ATRm;ATRh;ATLf;...;FRh;FLf;FLm;FLh
timestep_002;ATRf;ATRm;ATRh;ATLf;...;FRh;FLf;FLm;FLh
...
timestep_100;ATRf;ATRm;ATRh;ATLf;...;FRh;FLf;FLm;FLh
```

There is a *.txt* file of this structure for every single simulation run in the *root/data/* directory (see appendix A3).

All the data files are generated by a script called *generate_txt_data.py* (A3). This script takes several arguments, like the number of jobs (simulation runs), terrain types involved or the terrain noise *std* (σ_p). Then a loop based on these parameters starts, where the simulation is launched and stopped after ten seconds each iteration. This is performed by calling a bash command (since the simulation is *.cpp* based) and then killing the called process from python. The corresponding *.txt* file is saved after each iteration by the simulation and then copied by the python script to a corresponding folder in *root/data/*.

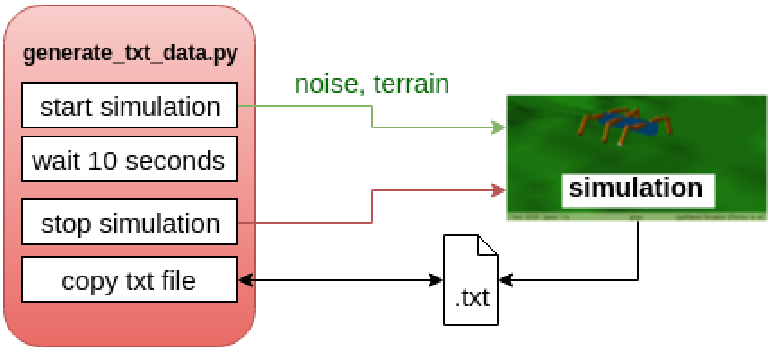

FIGURE A2.3: The process of data acquisition from the simulation.

In this manner, *.txt* files for all terrains and all mentioned σ_p are saved into a structure illustrated on Fig. A2.4. Each *.txt* file contains approximately 100 lines, one for each simulation step (as shown in code part A2.2). Every line then contains values of the 24 proprioceptive sensors.

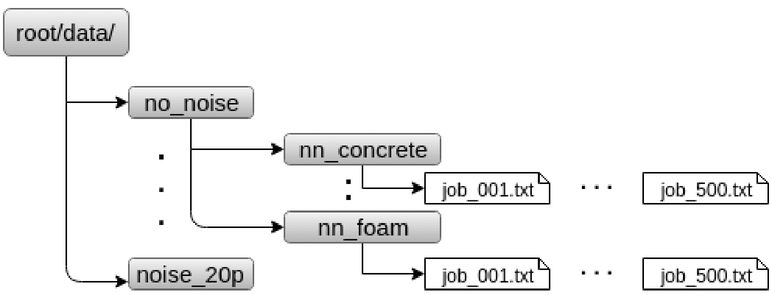

FIGURE A2.4: The structure of rough data directory.

Right after the data generation, a script called *clean_txt_data.py* (A3) is used to check the created *.txt* files. As it takes a long time to generate all the data, sometimes the simulation fails and the files are incomplete. Hence the script checks whether there are enough timesteps (at least more than 95) and also if the steps are not messed. Files that fail during the inspection are removed.

Datasets Storing

During the overall process description in previous sections, some global process parameters have been collected. These configurations are now passed as arguments to the script called *create_terrains_dataset.py* and therefore several datasets of various properties can be generated.

The datasets files are saved in directory `root/py/cache/datasets/amos_terrains_sim/` (see A3). Their structure is based on a powerful serializing and de-serializing Python algorithm implemented under a package called *pickle (cPickle)*. On the same basis a package called *shelve* is used to represent a dataset as a dictionary-link object. The files are saved with the *.ds* suffix.

Trained Network Storing

Finally, the trained network needs a file name, as it is saved the same way as the datasets (see section 5.6) - using the *pickle (cPickle)* package, just with the *.net* suffix.

A2.2 NN Implementation Details

Scikit-learn Neural Network Library

In order to verify the functionality of implemented neural network library (**??**), a provided public library is used. As the official description says (*sknn: Deep Neural Networks without the Learning Cliff*), this library implements multi-layer perceptrons as a wrapper for the powerful *pylearn2* library that is compatible with *scikit-learn* for a more user-friendly and Pythonic interface.

This step has been considered with the aim to test another implementation of the learning algorithm rather than to obtain better classification results. As the only learning parameters are the *net structure*, the *learning rate* and the *number of epochs*, some other default parameters of the tested network are shown in code part A2.3.

Part of Code A2.3: Sknn classifier specification (ibid.)

```
class sknn.mlp.Classifier(layers, warning=None, parameters=None,
random_state=None, learning_rule=u'sgd', learning_rate=0.01,
learning_momentum=0.9, normalize=None, regularize=None,
weight_decay=None, dropout_rate=None, batch_size=1, n_iter=None,
n_stable=10, f_stable=0.001,  valid_set=None, valid_size=0.0,
loss_type=None, callback=None, debug=False,  verbose=None)
```

Appendix A3

Code Documentation

Write your Appendix content here.

Appendices must be limited to supporting material genuinely subsidiary to the main argument of the work. They must only include material that is referred to in the document.

Material suitable for inclusion in appendices includes the following:

Additional details of methodology and/or data Diagrams of specialized equipment developed Copies of questionnaires or surveys used in the research Do not include copies of the Ethics Certificates in the Appendices.

Appendix A4

Detailed Results

all classification reports, confusion matrices and so on...