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 21, 2016

Declaration of Authorship

“*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

AI	**A**rtificial **I**ntelligence
NN	**N**eural **N**etwork
ANN	**A**rtificial **N**eural **N**etwork
SVM	**S**upport **V**ector **M**achines
RF	**R**andom **F**orest
k-NN	**k**-**N**earest **N**eighbours
API	**A**pplication **P**rogramming **I**nterface
GUI	**G**raphical **U**ser **I**nterface
AMOS II	**A**dvanced **Mo**bility **S**ensor Driven-Walking Device - version **II**
CPG	**C**entral **P**attern **G**enerator
GDA	**G**radient **D**escent **A**lgorithm
PA	**P**runing **A**lgorithm
sknn	**s**ci**k**it-**n**eural**n**etwork library

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 Master Thesis Objectives

""" just saved this text here, will be reformulated.... """ 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.

1.5 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

Classification Method

Principles of feedforward neural networks, based on the perceptron idea (see [ref to lit]), are used for developing the new classification framework.

As the first part of the study is devoted to the evolution of a new network pruning algorithm, besides some standard functions, the new framework implementation must be capable of:

1. pruning of the network during training process;
2. retraining the pruned network.

In this thesis, the implemented neural network framework is called *kitt_nn*. Implementation details are described in appendix A2.2.

3.1 Network Structure

In this work, a *feedforward* neural network is used, where none of the neurons are connected to other neurons in the same layer or any neurons in the previous layers and are connected to all neurons in the following layer (Fig. 3.1).

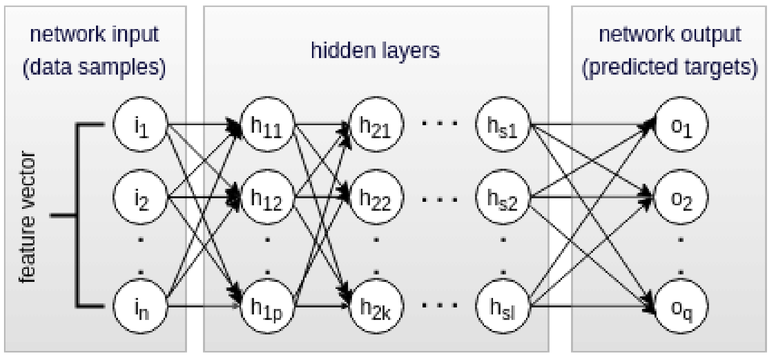


FIGURE 3.1: Structure of a feedforward neural network

As the number of neurons in input and output layers are determined by a chosen dataset, the network structure is defined by:

1. number of hidden layers;
2. number of neurons in each of the hidden layers.

3.2 Neuron Principle

The behaviour of artificial neurons follows our understanding of how biological neurons work. One unit consists of multiple inputs and a single output. A model of neuron is shown in Fig. 3.2. The diagram complies with the following notation:

$a_k^{(i)}$: activity of k^{th} neuron in i^{th} layer

$w_{l,k}^{(i)}$: weight of synapse connecting l^{th} neuron in i^{th} layer with k^{th} neuron in $(i+1)^{th}$ layer

$neuron_k^{(i)}$: k^{th} neuron in i^{th} layer

$b_k^{(i)}$: bias connected to k^{th} neuron in i^{th} layer

$z_k^{(i)}$: activation of k^{th} neuron in i^{th} layer

$f()$: transfer function (Eq. (3.2))

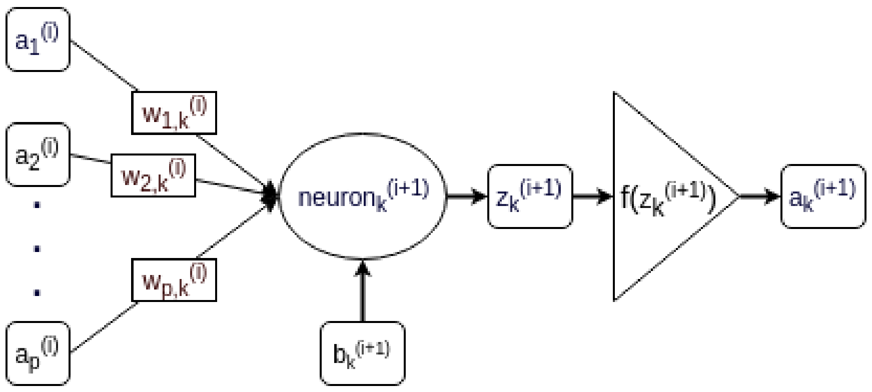


FIGURE 3.2: A model neuron

Assuming p being the number of neurons in i^{th} layer, the activation of $neuron_k^{(i+1)}$ is computed as in 3.1

$$neuron_k^{(i+1)} = \sum_{l=1}^{p} a_l^{(i)} \cdot w_{l,k}^{(i)} + b_k^{(i+1)} \tag{3.1}$$

The sigmoid function given as in Eq. (3.2) is used as the transfer function for computing the activities of individual neurons.

$$f(z) = \frac{1}{1+e^{-z}} \tag{3.2}$$

The sigmoid function maps neuron activations into $[0.0, 1.0]$ interval. This approach is used for most of the work in this study. Additionaly, the *tanh(z)* function is implemented (compared to *sigmoid(z)* in Fig. 3.3), in order to test one of the pruning methods based on weights sensitivity (discussed later). The *tanh(z)* function maps the input z into $[-1.0, 1.0]$ interval.

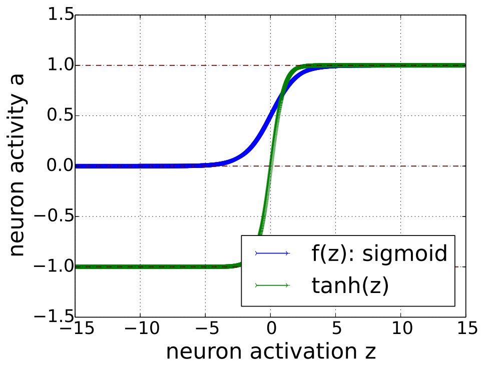


FIGURE 3.3: Used transfer functions: sigmoid and tanh

3.3 Learning Algorithm for Network Training

In general, learning algorithms represent the part of artificial systems that makes them behave intelligently when accomplishing a specific task. In classification, the goal of learning is to fit some training data to a model, which generally means to set some parameters based on the chosen classification approach.

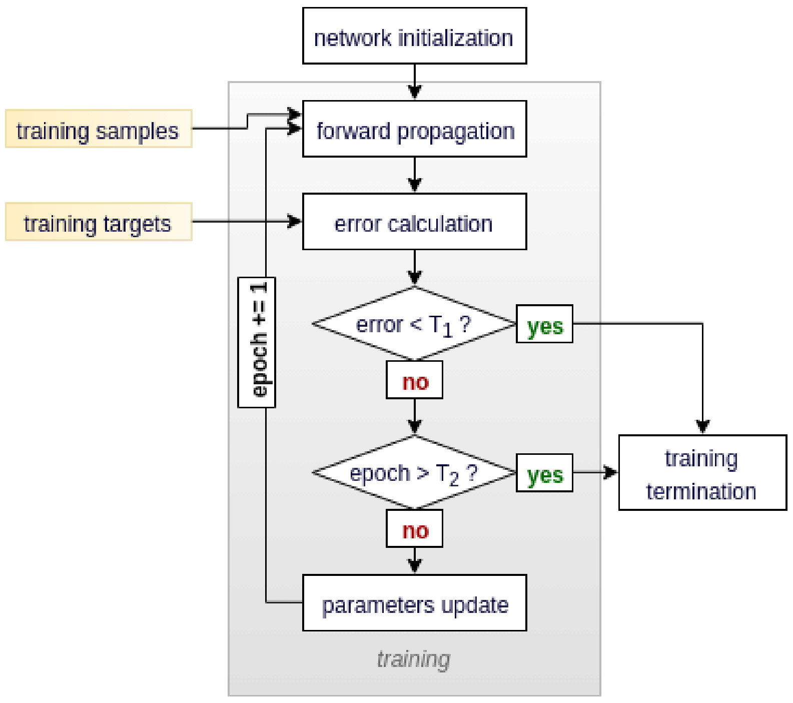


FIGURE 3.4: Training process flowchart. T_1: Threshold for a terminating condition based on a learning error. If the error is reduced to be lower than this threshold, the learning process is stopped. T_2: Threshold for a terminating condition based on number of iterations (epochs). The learning process is stopped after a specified number of epochs, no matter how successful the training has been.

In case of feedforward neural networks, the learning goal is to find optimal values for two groups of parameters - *weights* and *biases* (described in section 2.2).

A popular algorithm called *Backpropagation* is used to deal with the learning task. The backpropagation abbreviation stands for *backward propagation of errors.* The approach is based on an optimization method called *Gradient Descent Algorithm (GDA).*

In this work, the implementation is made to be compatible with the *kitt_nn* network (appendix A2.2) and adjusted to the needs of the developed pruning algorithm (section 3.4). The learning process is summarized by the flowchart in Fig. 3.4. The procedure follows algorithmical steps found in (Labs, 2014).

3.3.1 Using Mini-batches

The training procedure of multilayer perceptrons ([ref to lit]) is based on propagating one sample at a time through a network. However, more samples can be send to the network, while activities and activations of neurons in one layer are computed at the same time for all of those samples.

This group of samples is called *a mini-batch.* Using mini-batches can speed up the process, however it can bring some problems with finding the right solution by the *Gradient Descent* method (evident from Eq. (3.8)). Usually, the mini-batch size is left as a training parameter. In this work, it is set to *10.*

3.3.2 Matrix Notations

Assuming feedforward neural networks and reffering to Fig. 3.2, the following notation is used for multiple samples processing.

X : network input: m-by-n matrix, where m is the number of samples and n is the size of one sample

$W^{(i)}$: p-by-r matrix of weights for synapses from neurons in layer i (layer of p neurons) to neurons in layer $(i+1)$ (layer of r neurons)

$B^{(i)}$: vector of length p including biases for neurons in layer i (layer of p neurons)

$Z^{(i)}$: r-by-m matrix of activations for neurons in layer (i) (layer of r neurons), m is the number of processed samples

$A^{(i)}$: r-by-m matrix of activities of neurons in layer (i) (layer of r neurons), m is the number of processed samples

$\hat{y}$: network predicted output: q-by-m matrix, where q is the number of output neurons and m is the number of processed samples ($\hat{y} = f(Z^{(j)})$, where j is the number of layers)

y : network actual output (known targets): q-by-m matrix, where q is the number of output neurons and m is the number of processed samples

$\delta^{(i)}$: error vector of length p for p neurons of i^{th} layer

3.3.3 Forward Propagation

With reference to previous sections, the following equations are used to propagate a batch of samples X through a network and get a corresponding matrix of outputs $\hat{y}$.

$$Z^{(2)} = X \cdot W^{(1)} + B^{(2)} \tag{3.3}$$

$$Z^{i+1} = A^{i} \cdot W^{i} + B^{(i+1)} \tag{3.4}$$

$$A^{(i)} = f(Z^{(i)}) \tag{3.5}$$

$$\hat{y} = f(Z^{(j)}) \tag{3.6}$$

3.3.4 Error Calculation

To get an idea about how wrong the networks predictions are, the network needs a teacher. For this reason the learning is called *supervised*, as there is a batch of training data with known targets. Comparing the predictions with the correct targets, one can calculate a prediction error.

The prediction error of the propagated batch of samples is expressed as a cost function J. The goal is to minimize J.

$$J = \sum_{k=1}^{m} \frac{1}{2}(y_k - \hat{y}_k)^2 \tag{3.7}$$

3.3.5 Parameter Update

Knowing the prediction error, the goal of the learning algorithm is to update the network weights and biases in order to reduce the error for next prediction. It is known, that some combination of the parameters makes J (Eq. (3.7)) minimal. There is no chance to check all possible combinations for bigger networks, therefore *GDA* is used here.

Partial derivatives $\frac{dJ}{dw}$ for all weights w of chosen weight matrix $W^{(i)}$ belonging to layer i are computed and set equal to zero ($\frac{dJ}{dw} = 0$). Applying this on a mini-batch, we get m derivatives $(\frac{dJ}{dW^{(i)}})_k$ for m input samples.

GDA is then applied on the summation of these derivatives and so all examples are considered as one (Eq. (3.8)). In other words, one can understand it as every sample has a vote on how to find the minimal error and the result is obtained as a compromise.

$$\frac{dJ}{dW^{(i)}} = \sum_{k=1}^{m}(\frac{dJ}{dw})_k \tag{3.8}$$

Having several layers of a network results in several weights (and biases) mattrices, while the goal is to find optimal parameters of overall network. In order to compute optimal parameters in all of the mattrices, the sum rule in differentation (Eq. (3.9)) followed by the chain rule (Eq. (3.10)) are applied.

$$\frac{\delta}{\delta x}(u+v) = \frac{\delta u}{\delta x} + \frac{\delta v}{\delta x} \tag{3.9}$$

$$(f \circ g)' = (f' \circ g) \cdot g' \tag{3.10}$$

Due to these properties, the error obtained at the network output (section 3.3.4) is propagated backwards layer by layer through the network (Eq. (3.12)) and derivatives $\frac{dJ}{dW^{(i)}}$ for all i layers are found (Eq. (3.13)). For this procedure, the derivative of our transfer function is needed (Eq. (3.11)).

$$f'(z) = f(z) \cdot (1 - f(z)) \tag{3.11}$$

$$\delta^{(i)} = \delta^{(i+1)} \cdot (W^{(i)})^T \cdot f'(Z^{(i)}) \tag{3.12}$$

$$\frac{dJ}{dW^{(i)}} = (a^{(2)})^T \cdot \delta^{(i+1)} \qquad \frac{dJ}{dB^{(i)}} = \delta^{(i+1)} \tag{3.13}$$

The parameters are then updated for the next iteration as shown in Eq. (3.14).

$$W^{(i)}_{(t+1)} = W^{(i)}_{(t)} + \frac{dJ}{dW^{(i)}}_{(t)} \qquad B^{(i)}_{(t+1)} = B^{(i)}_{(t)} + \frac{dJ}{dB^{(i)}}_{(t)} \tag{3.14}$$

In this work, the learning algorithm is implemented in *Python*, using mostly the *numpy* library for the expensive matrix operations. The implementation complies with the needs of the pruning algorithm from section 3.4 and is fully compatible with the *kitt_nn* framework for any type of data.

3.4 Network Pruning Algorithm

The network pruning algorithm (PA) is the novelty of this study. The state-of-the-art methods based on feedforward neural networks (see section 2.2) use a fully-connected structure by default. This means that each unit is connected to all units in the next layer. Hence a net structure is defined just by the number of hidden layers and the number of neurons in each of those. The question is if all of by default generated connections are significant and necessary for classification.

3.4.1 Pruning Method

The hypothesis is that some synapses of a fully connected feedforward network can be removed without the net's classification performance being influenced significantly. The problem is graphically illustrated in Fig. 3.5.

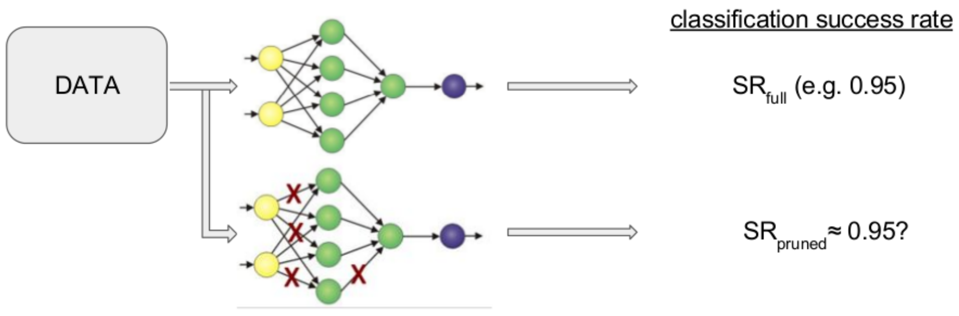


FIGURE 3.5: Pruning Algorithm: hypothesis formulation

The task is to identify the unimportant synapses. The basic idea of the algorithm is as follows:

1. a fully-connected network is initialized and trained on some data in order to reach as high accuracy as possible;
2. some of the synapses are removed and a classification ability of the pruned net is tested;
3. if it has not dropped significantly, the removed synapses were unimportant

The way of removing synpases is called *pruning algorithm (PA)*.

The idea behind the PA implemented in this study is based on weight changes during the network training. It is expected that a synapse, whose weight does not evolve while the network is trained, does not contribute to classification much.

3.4.2 Algorithm Realization

The pruning algorithm itself is an iterative process, however, as shown in Fig. 3.6, two important steps are done in advance.

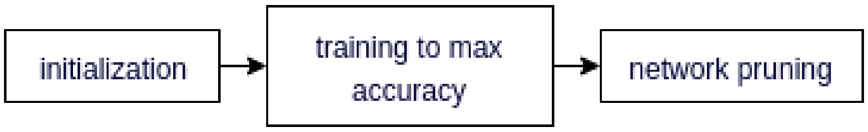


FIGURE 3.6: Overall flowchart of the pruning process

First of all, the following variables are initialized:

net : a fully-connected *kitt_nn Network()* with an oversized structure (many hidden neurons) and randomly set weights and biases

percentile : algorithm variable, set to 75 by default

percentile levels : array of final variables, set to $[75, 50, 20, 5, 1]$ by default

required accuracy : required classification accuracy for a chosen problem (e.g. 0.95)

stability threshold : if the classification does not improve over several learning iterations, this is the number of stable iterations to terminate the training after

Additionally, some standard learning parameters like the *learning rate*, *number of epochs* and *mini-batch size* can be set and, of course, a dataset is chosen.

Once the initialization is done, the network is trained with some optimal learning parameters until it reaches the required classification success rate. As mentioned above, the pruning algorithm is based on weight changes. Therefore, the initial weights for synapses are kept. Then, the trained network is passed to the pruning phase, which is shown in Fig. 3.7.

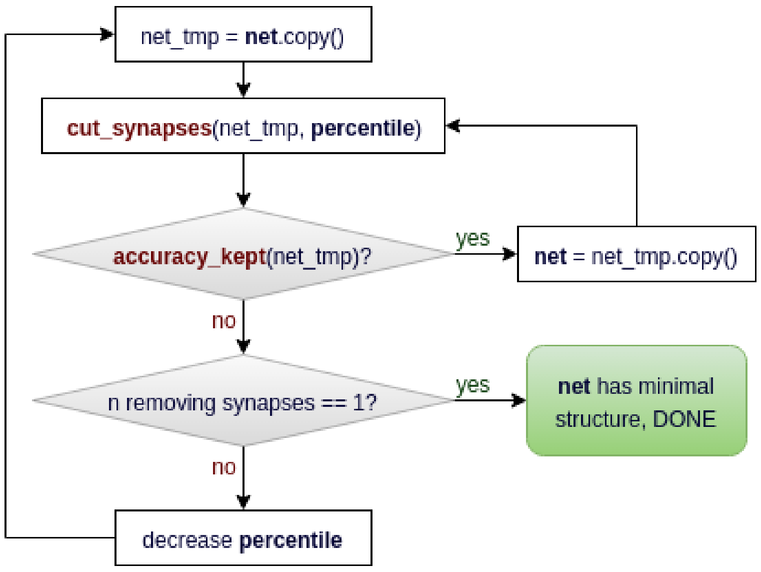


FIGURE 3.7: The Pruning Algorithm: initialized variables are in bold, red marked functions reffer to 3.8 and 3.9 respectively

Initially, a backup of the current network structure is made by creating a network copy. The pruning is then performed using this copy. The pruning method is shown in Fig. 3.8.

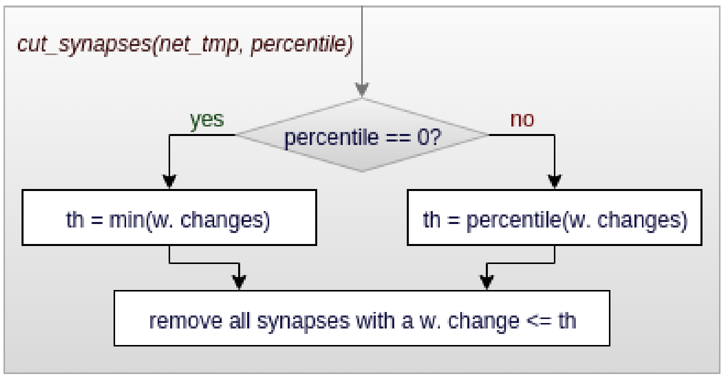

cut_synapses(net_tmp, percentile)
percentile == 0?
yes
no
th = min(w. changes)
th = percentile(w. changes)
remove all synapses with a w. change <= th

FIGURE 3.8: Synaptic pruning based on weight changes and current percentile value

As discussed above, an initial weight value had been saved for each synapse before the training. Hence, for n being the total number of synapses, weight changes Δw_i are known (3.15).

$$\Delta w_i = |w\_init_i - w\_current_i|, \quad i = 1, ..., n \tag{3.15}$$

Based on these changes and on the current *percentile* value, a threshold (*th* in Fig. 3.8) is determined. Then, all synapses with a lower change in weight than this threshold are removed, and, if there is a neuron with no connections left, it is removed as well.

If the *percentile* variable has been decreased such that it equals zero, the threshold is set to the minimum of all weight changes. In this case, only one synapse is then removed at a time.

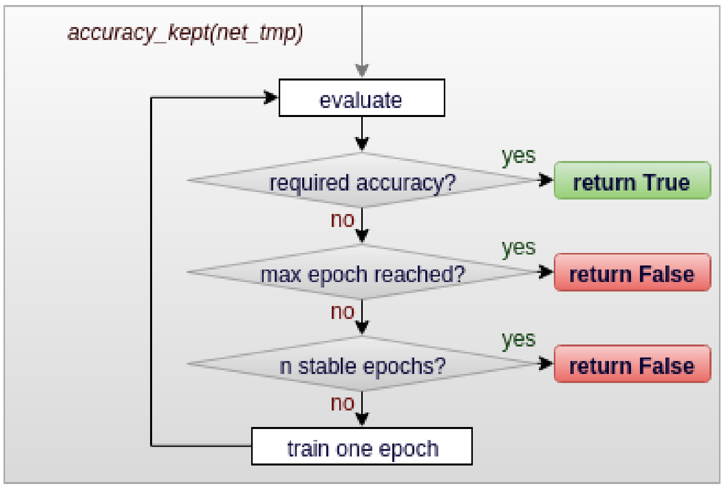


FIGURE 3.9: Evaluation of the classification accuracy after pruning

After cutting some synapses, the network is checked for keeping the required classification accuracy as shown in Fig. 3.9.

The evaluation is performed by testing the network on validation data. If the classification accuracy has dropped, the network is retrained using training data of the dataset.

If the classification accuracy has been kept after the pruning, the current net structure is saved and considered as a reference for the next pruning loop.

If it is not possible to retrain the pruned net and to reach the required accuracy, two possibilities arise:

1. Only one synapse has been removed during the last pruning step and the accuracy has been broken. This means that even this single synapse with the least change in weight is important for classification. Therefore, the pruning is stopped and the current net structure (including this last synapse) is saved as the minimal structure.
2. More synapses have been removed during the last pruning step, and this broke the accuracy. In this case, the percentile level is decreased (based on the initialized array of percentile levels) and so less synapses are removed during the next pruning iteration.

   In this manner, at some point of the pruning process, the algorithm will come to removing only one synapse at once and then, finally, only case 1 will remain.

Therefore, the algorithm is finite. Moreover, it guarantees that the classification success rate does not drop. It is evaluated in detail and compared to some other approaches from (Reed, 1993); (also discussed in section 2.3) in the results part (section 5.2).

3.4.3 Datasets for Evaluation of the Pruning Algorithm

Two datasets were used for verification of the pruning algorithm: XOR and MNIST. Those are introduced in the following sections and the evaluation is presented in section X.

The pruning algorithm was also applied on the terrain classification problem (evaluated in section X).

XOR Dataset

This dataset relates to the well-known XOR problem defined by a thruth table (Table 3.1).

TABLE 3.1: XOR problem definition

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

Formulating the XOR gate as a classification problem, it is represented by two linearly inseparable classes with labels 0 and 1. In this case, each class consists of 1000 samples, which have been generated using the developed GUI (section 3.5). The composition of the two classes in a 2D space is shown in Fig. 3.10.

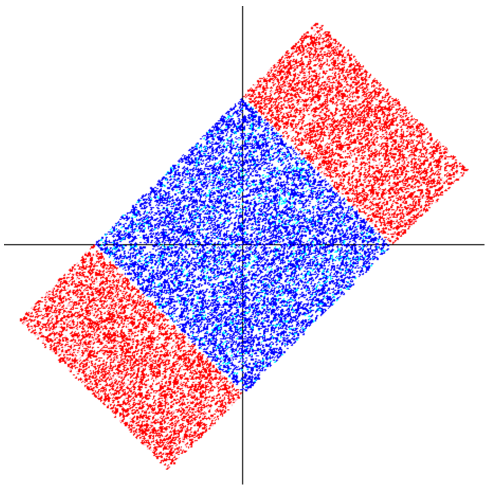
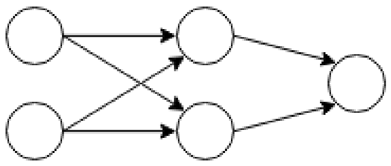


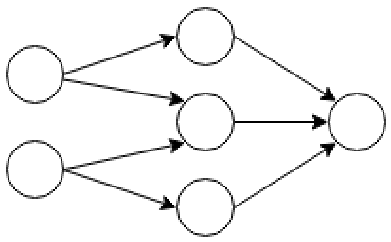
FIGURE 3.10: 2D XOR Data illustration

The points (samples) have been generated pseudo-randomly, while it is guaranteed that the 'red' samples are distributed half-and-half between the two 'red' areas. The minimal distance between a 'blue' and a 'red' sample is 0.001 and it is possible to separate the classes using two lines.

The XOR dataset is essential for evaluation of the implemented PA, as the minimal network structure capable of solving the problem is known. There are two structures (Fig. 3.11), both considered as minimal. Geometrically, the version shown in Fig. 3.11a creates the two lines in 2D space to separate the classes, while the second one (Fig. 3.11b) transfers the problem from 2D space into 3D space and creates a separation plane.



(A) Minimal structure 1



(B) Minimal structure 2

FIGURE 3.11: XOR Dataset: minimal network structures

The goal of the pruning algorithm is to end up with a network of the structure in Fig. 3.11a, when the network is initially oversized. The evaluation is presented in section 5.2.1.

MNIST Dataset

The second testing problem is the well-known classification of handwritten digits. The dataset is provided by (LeCun and Cortes, 1998). Some examples of digit samples are shown in Fig. 3.12.

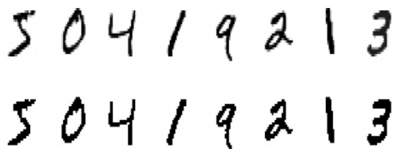


FIGURE 3.12: MNIST Data illustration (LeCun and Cortes, 1998)

The dataset has a training set of $60,000$ examples (later split into $50,000$ training and $10,000$ validation examples), and a test set of $10,000$ examples. The digits have been size-normalized and centered in a fixed-size image.

In this case, the minimal network structure is not known. On the other hand, detailed results of classification accuracy for various classifiers are provided, which is good for comparison.

3.4.4 Using Network Pruning for Feature Selection

In general, only the number of neurons (inputs and outputs) is determined for a chosen dataset. The pruning algorithm brings an additional information about the hidden part of the network. A minimal network structure is obtained as the PA outcome. This means that all the neurons and synapses contained in the network are important for classification.

Knowing that each of these units takes a part and is not useless, one can ask what role does a single neuron/synapse have with respect to the chosen dataset.

This can be investigated by tracking the connections from the input to the output layer. Based on this approach, one can find some correlations between feature selection of input vectors and a selected output class (demonstrated on a MNIST example in Fig. 3.13). Evaluation on MNIST dataset is shown in section 5.2.2.

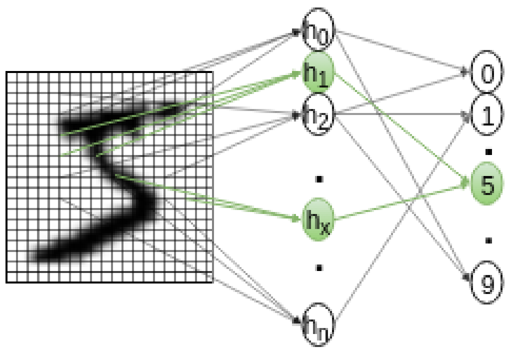
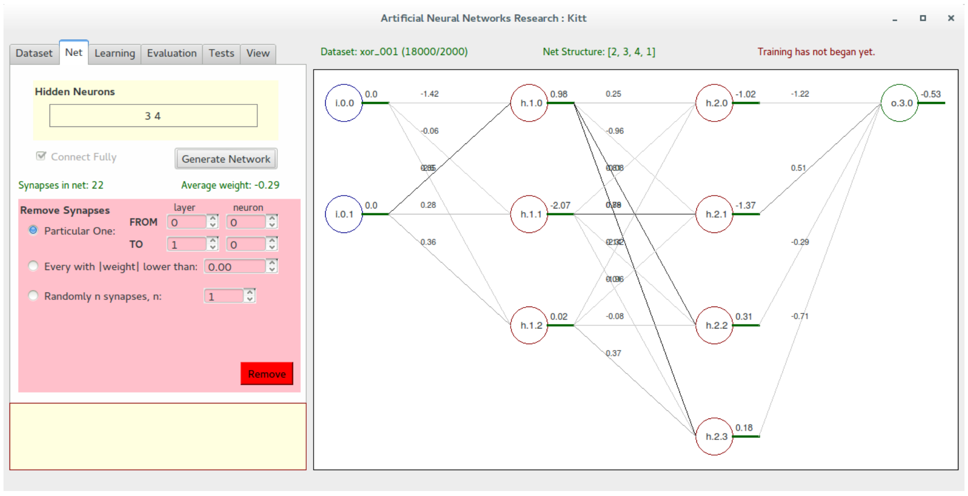


FIGURE 3.13: Analysis of the minimal structure examplified on digit 5 (MNIST dataset)

3.5 Graphical User Interface

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


Artificial Neural Networks Research : Kitt

Dataset: xor_001 (18000/2000)

Net Structure: [2, 3, 4, 1]

Training has not began yet.

Hidden Neurons: 3 4

Connect Fully

Synapses in net: 22

Average weight: -0.29

Remove Synapses

Particular One: FROM layer 0 neuron 0; TO layer 1 neuron 0

Every with |weight| lower than: 0.00

Randomly n synapses, n: 1

FIGURE 3.14: 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. 3.10 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. 3.14).
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 basic 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 workflow. Nevertheless, it is considered as the very fist version for now and aimed to be upgraded in the future.

Chapter 4

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. 4.2). 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 3, feedforward neural networks are used for classification.

4.1 Overall Process Summary

The overall process consists of several modules. The workflow is illustrated in Fig. 4.1 (a more detailed diagram can be found in Fig. A2.1).

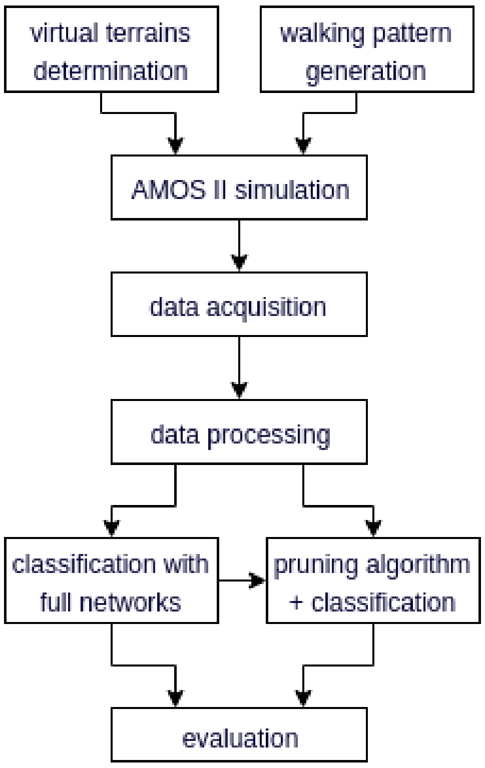

virtual terrains determination
walking pattern generation
AMOS II simulation
data acquisition
data processing
classification with full networks
pruning algorithm + classification
evaluation

FIGURE 4.1: 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 4.2.3). To generate various terrain types, the number of variable terrain features and their ranges are determined (section 4.3.1). Based on these features (parameters), a number of virtual terrains is defined (section 4.3.2) and an optimality of these parameters is briefly analysed (section 5.3.1).

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 parameters: learning rate and number of epochs

Finally, an optimal neural network classifier is found. The optimal network is then pruned by the algorithm developed in section 3.4. The classification performance of developed tools is compared to *Scikitlearn-neuralnetwork* classification library (Champandard and Samothrakis, 2015).

4.2 Experimental Environment Specification

The final objective of this thesis was to implement an online terrain classifier for selection of optimal walking gait on real hexapod robot AMOS II. Therefore the real hexapod robot is presented in the following section 4.2.1.

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.

4.2.1 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. 4.2). 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.

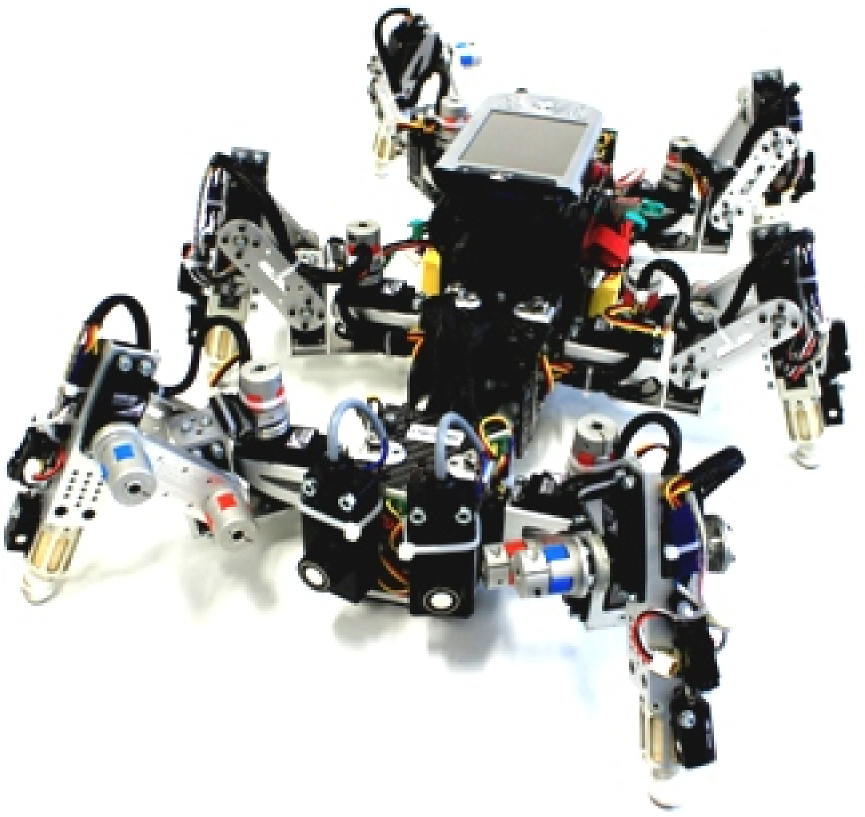


FIGURE 4.2: AMOS II. (*Open-source multi sensori-motor robotic platform AMOS II*)

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. 4.3.

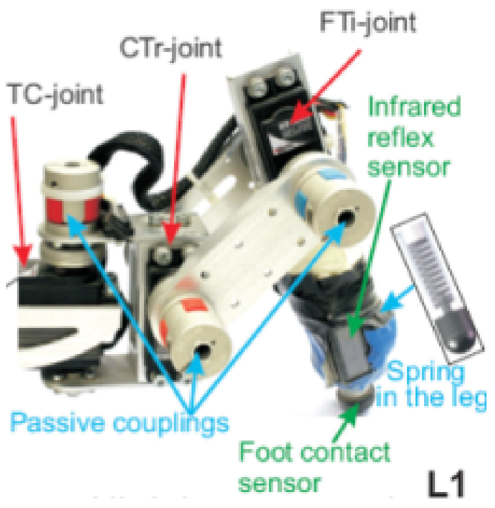


Figure 4.3: Structure of the AMOS's leg. (*Open-source multi sensori-motor robotic platform AMOS II*)

As shown in Fig. 4.2 and Fig. 4.3, 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 4.1 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 4.5 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 4.1: Summary of proprioceptive sensors of AMOS II hexapod robot

abbr.	*sensor description*	*original range*
ATRf	Angle sensor, Thoraco joint, Right front leg	[-0.5, 0.5]
ATRm	Angle sensor, Thoraco joint, Right middle leg	[-0.5, 0.5]
ATRh	Angle sensor, Thoraco joint, Right hind leg	[-0.5, 0.5]
ATLf	Angle sensor, Thoraco joint, Left front leg	[-0.5, 0.5]
ATLm	Angle sensor, Thoraco joint, Left middle leg	[-0.5, 0.5]
ATLh	Angle sensor, Thoraco joint, Left hind leg	[-0.5, 0.5]
ACRf	Angle sensor, Coxa joint, Right front leg	[-0.5, 0.5]
ACRm	Angle sensor, Coxa joint, Right middle leg	[-0.5, 0.5]
ACRh	Angle sensor, Coxa joint, Right hind leg	[-0.5, 0.5]
ACLf	Angle sensor, Coxa joint, Left front leg	[-0.5, 0.5]
ACLm	Angle sensor, Coxa joint, Left middle leg	[-0.5, 0.5]
ACLh	Angle sensor, Coxa joint, Left hind leg	[-0.5, 0.5]
AFRf	Angle sensor, Femur joint, Right front leg	[-0.5, 0.5]
AFRm	Angle sensor, Femur joint, Right middle leg	[-0.5, 0.5]
AFRh	Angle sensor, Femur joint, Right hind leg	[-0.5, 0.5]
AFLf	Angle sensor, Femur joint, Left front leg	[-0.5, 0.5]
AFLm	Angle sensor, Femur joint, Left middle leg	[-0.5, 0.5]
AFLh	Angle sensor, Femur joint, Left hind leg	[-0.5, 0.5]
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 4.2.3.

4.2.2 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. 4.4.

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.

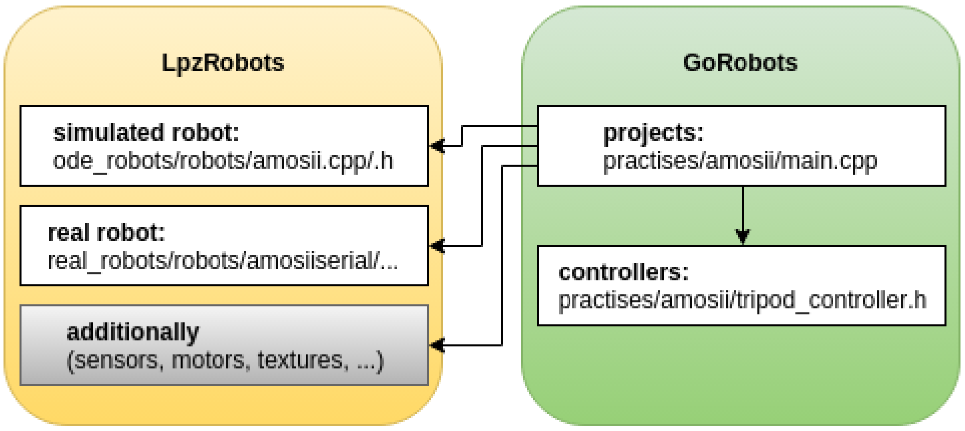

LpzRobots
simulated robot:
ode_robots/robots/amosii.cpp/.h
real robot:
real_robots/robots/amosiiserial/...
additionally
(sensors, motors, textures, ...)
GoRobots
projects:
practises/amosii/main.cpp
controllers:
practises/amosii/tripod_controller.h

FIGURE 4.4: 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 section 4.3.2. 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 4.3.3) 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. 4.5.

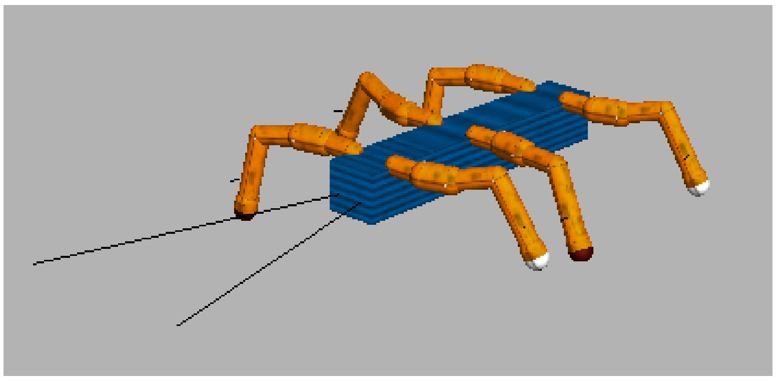


FIGURE 4.5: Virtual alternative for AMOS II.

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.

4.2.3 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. 4.6.

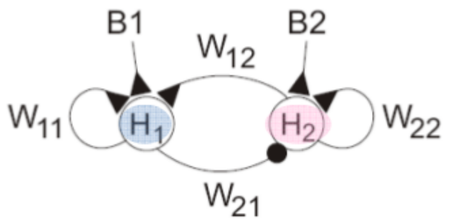


FIGURE 4.6: 2-neuron network oscillator ("Adaptive Embodied Locomotion Control Systems")

The initial conditions and parameters of the implemented controller are shown in Table 4.2.

TABLE 4.2: 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**

$$a_i(t+1) = \sum_{j=1}^{n} w_{ij} o_j(t) + b_i, i = 1, ..., n \tag{4.1}$$

2. **The transfer function application**

$$f(a_i) = tanh(a_i) = \frac{2}{1+e^{-2a_i}} - 1 \qquad (4.2)$$

3. **Joint settings**

   With the reference to previous equations and variables names, the actuators are set as shown in Fig. 4.7. The *femur* joints (red ones) stay unchanged (set to zero). This setting generates a tripod gait for AMOS II.

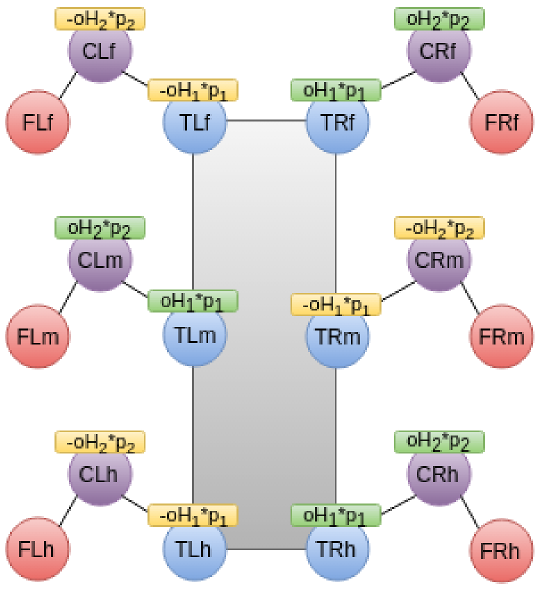


FIGURE 4.7: Schematic diagram of tripod gait controller

4.3 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

4.3.1 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 4.3).

TABLE 4.3: 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

4.3.2 Features Determination for Various Terrains

To determine a terrain type, one has to come up with the five parameters from Table 4.3.

In this work we use 14 terrain types. Their parameters (shown in Table 4.4) have been set up manually. With respect to the feature ranges from Table 4.3, the values have been normalized between 0 and 1.

TABLE 4.4: 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 5.3.1.

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

4.3.3 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 4.3). The parameters in Table 4.4 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 4.4 are noised. Regarding individual features and their upper limits from Table 4.3, the following Eq. (4.3) shows, how the noise is added.

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

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

$$feature\_noise = fRand(0, feature_{up\_limit} * std_p) \tag{4.3}$$

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 4.3). 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 4.4) as given in Eq. (4.4).

$$feature\_value \mathrel{+}= feature\_noise \tag{4.4}$$

Additionally, there is some limits checking as the parameters cannot take negative values. The final form is set as shown in Eq. (4.5).

$$feature\_value = max(feature\_value, 0) \tag{4.5}$$

Influence of Terrain Noise

Based on the explanation of adding the terrain noise, single samples representing various volumes of the additive terrain noise may vary, but not necessarily. For illustration, three noisy terrain examples (of different noise level) for *rock* are shown in Fig. 4.8.

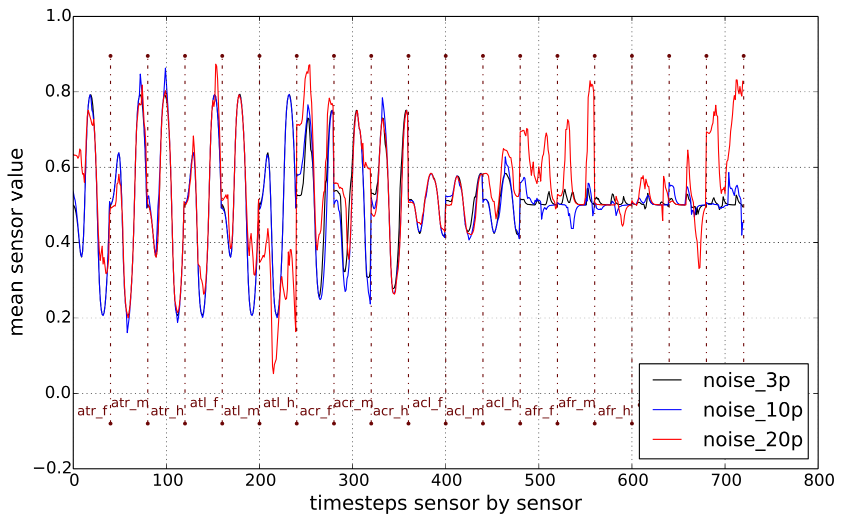


FIGURE 4.8: Examples of noisy terrains: terrain rock, angle sensors

The purpose of adding the terrain noise is to generate a variability among samples of one terrain, which makes the terrain definition more flexible. As shown in Fig. 4.8, the sample representing the $20\% - noise$ class fluctuates more in comparison to the others, expecially for the femur-tibia sensors. This might indicate a generation of an unusual rocky terrain.

4.4 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 4.5), 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).

As showin in Fig. 4.3, the robot has 3 proprioceptive and 1 tactile sensor on each leg. In the following figures (Fig. 4.9 - Fig. 4.12), examples for each of these sensor types are shown for all reffered terrains.

In Fig. 4.9, outputs of the *Thoraco-Coxal* joint angle sensor on the right front leg are shown. *Thoraco* sensors produce similar outputs for all terrain types, however little variances (mostly for grass and carpet) can be seen.

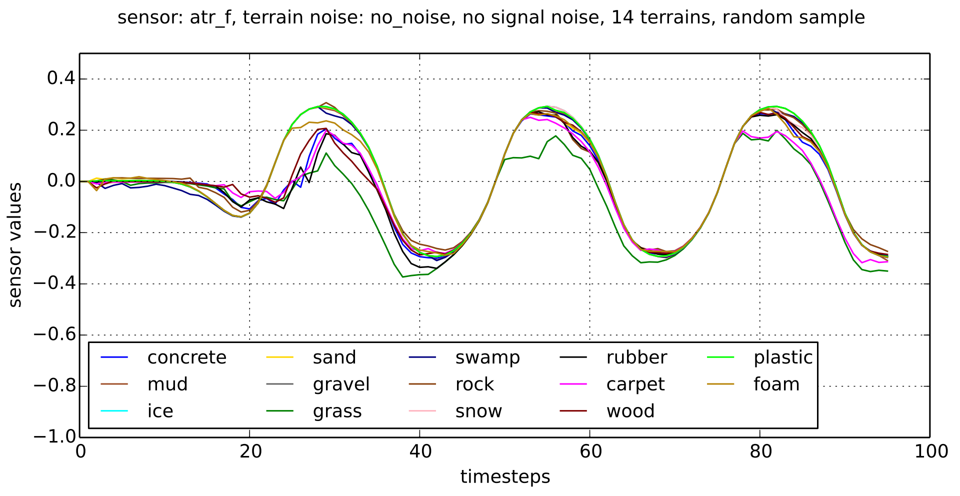


FIGURE 4.9: Thoraco Sensor (ATRf) output examples, 14 terrains

Fig. 4.10 presents outputs of the *Coxa-Trochanteral* joint angle sensor on the right middle leg. These joints are responsible for the elevation and depression of the leg and their signals vary especially at the decreasing parts of the signals. This might indicate that the leg is depressed differently on various terrains.

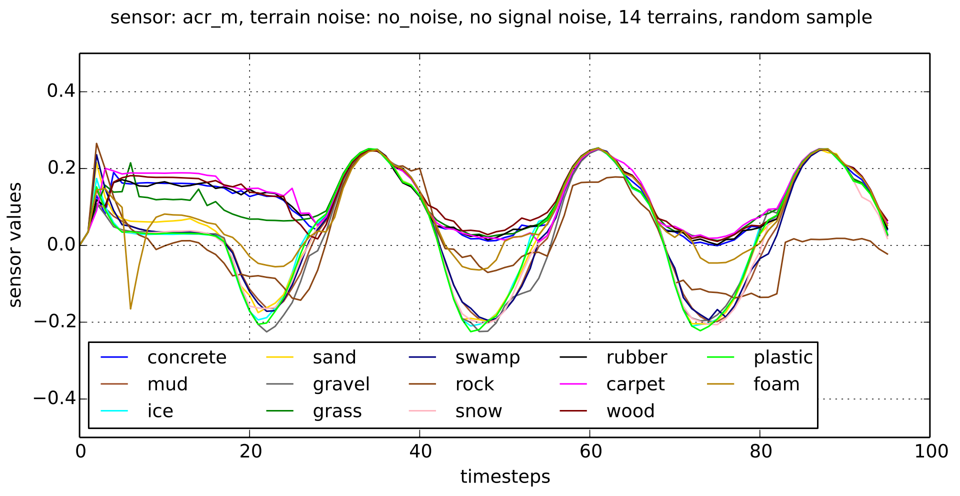


FIGURE 4.10: Coxa Sensor (ACRm) output examples, 14 terrains

The *Femur-Tibia* sensor outputs, for the joint sensor on the right hint leg, are illustrated in 4.11. The signals indicate that this joint is not much used compared to previous ones. This makes sense as there is no active movement of this joint generated by the tripod gait controller (section 4.2.3). The signals fluctuations must be caused by passive movements of the *Femur-Tibia* joint, but still might be essential for classification.

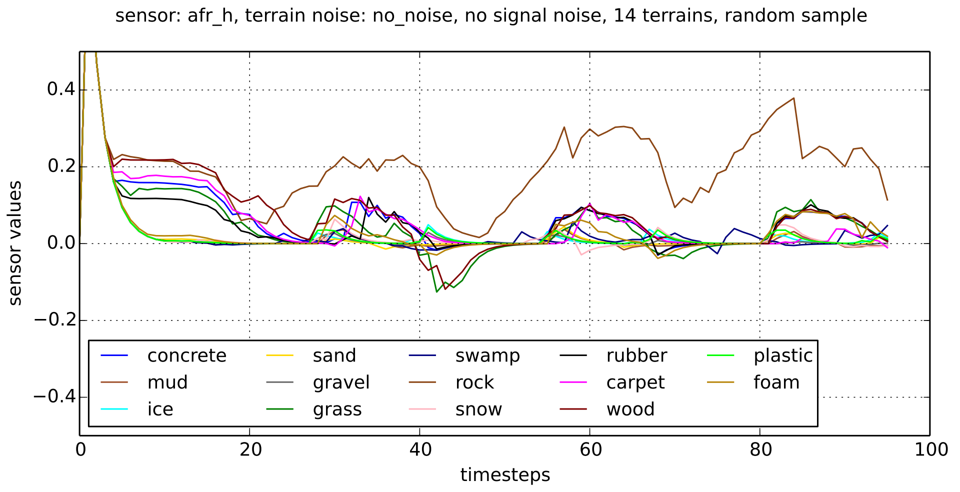


FIGURE 4.11: Femur Sensor (AFRh) output examples, 14 terrains

The biggest variance among signals for various terrains is obtained from the foot contact sensors (example from the sensor on the right front leg in Fig. 4.12). If the value of the foot contact sensor is 1, the robot stays on the foot with its full weight, and vice versa.

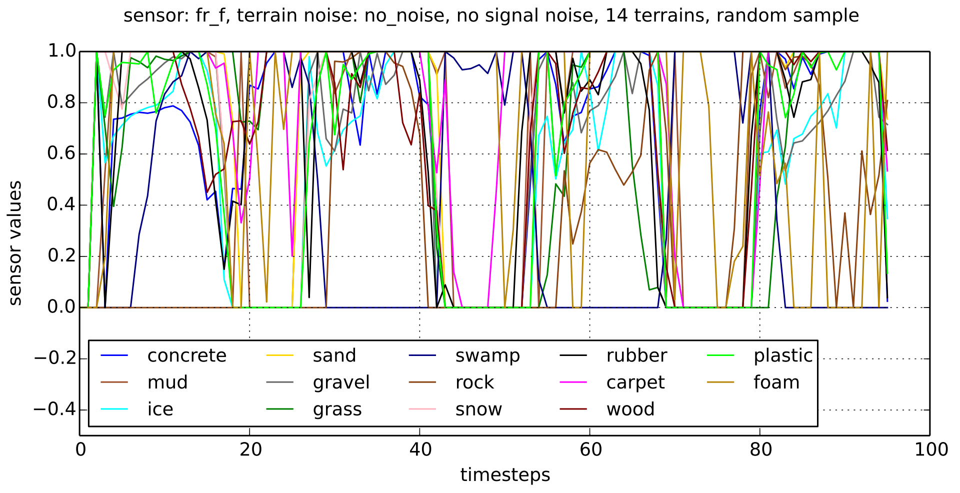


FIGURE 4.12: Foot Contact Sensor (FRf) output examples, 14 terrains

Examples of signals from all 24 sensors can be found in appendix A4.1.

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. (4.3). The influence of additive terrain noise is analyzed in section 4.3.3.

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.

4.5 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 4.1).

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. 4.13).

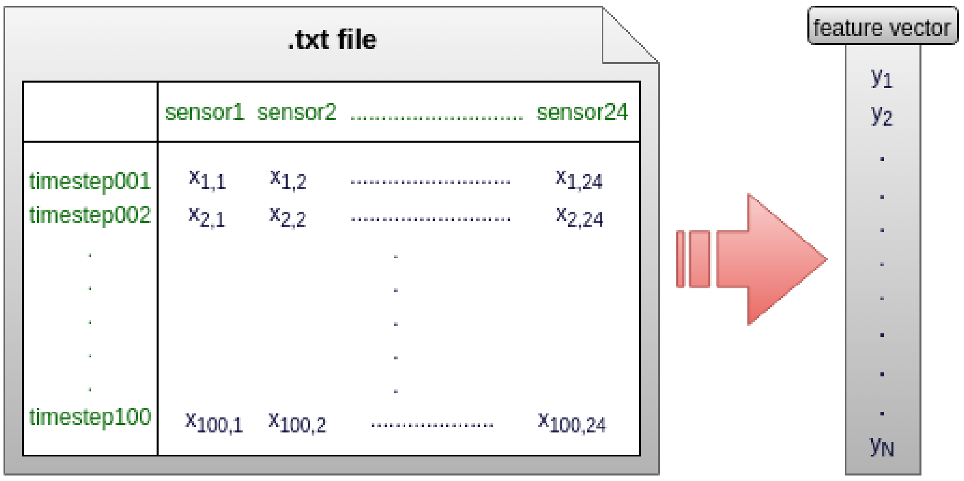

.txt file
sensor1 sensor2 .... sensor24
timestep001 $x_{1,1}$ $x_{1,2}$ .... $x_{1,24}$
timestep002 $x_{2,1}$ $x_{2,2}$ .... $x_{2,24}$
timestep100 $x_{100,1}$ $x_{100,2}$ .... $x_{100,24}$
feature vector
y_1
y_2
y_N

FIGURE 4.13: 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. 4.13, 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. 4.14 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 4.1) are added to the x-axis annotation. The 18 angle sensors are followed by the 6 foot contact sensors.

4.5.1 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:

$$S'_i = \frac{S_i - r_{min}}{r_{max} - r_{min}} \tag{4.6}$$

r_{min}, r_{max} : bounds of the corresponding original sensor range (listed in Table 4.1)

S'_i : scaled element of the normalised signal

Also a $[0, 1]$ interval overflow checking is added and values are adjusted if needed (Eq. (4.7)). This is a cover for the case when ranges from Table 4.1 were not accurate.

$$S'_i = min(max(S'_i, 0), 1) \tag{4.7}$$

The following figure (4.14) shows normalised feature vector examples for three terrains. The influence of normalisation on classification results is another subject for the discussion.

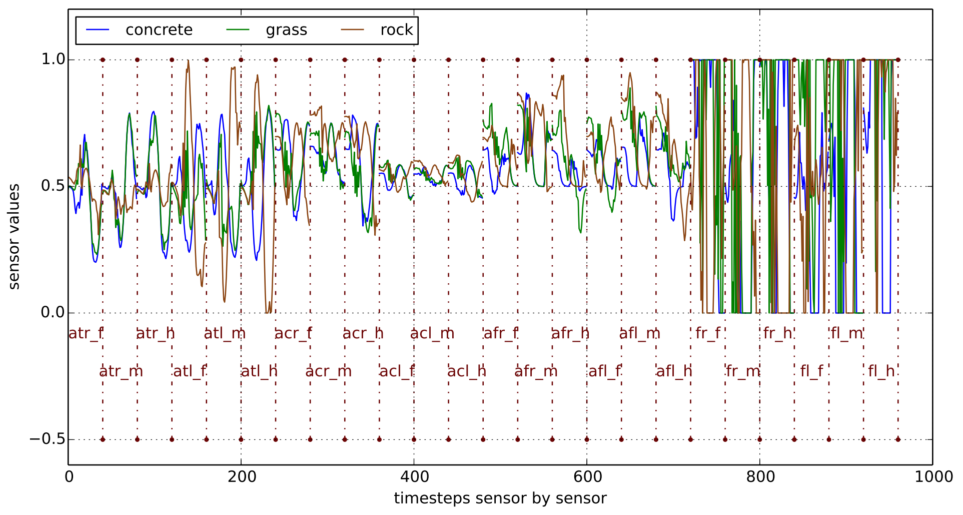


FIGURE 4.14: Normalised feature vector examples

4.5.2 Signal Noise

In section 4.3.3 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 4.1) 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 4.3.3, 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.

$$signal\_noise = [sn_1, sn_2, ..., sn_n] \tag{4.8}$$

$$sn_i \sim N(\mu, \sigma^2), \quad i = 1, 2, ..., n \tag{4.9}$$

Then, the generated vector is added to a normalised feature vector from section 4.5.1 (Eq. (4.10)).

$$noised\_signal_i = raw\_signal_i + sn_i, \quad i = 1, 2, ..., n \tag{4.10}$$

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

$$noised\_signal_i = min(max(noised\_signal_i, 0), 1), \quad i = 1, 2, ..., n \quad (4.11)$$

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.

Influence of Signal Noise

Fig. 4.15 shows the influence of the signal noise on one sample of *concrete* for joint angle sensors. A similar figure for foot contact sensors is put in appendix A4.2 (Fig. A4.16).

Naturally, the higher the standard deviation of additive Gaussian noise is, the more a corresponding signal fluctuates around the *red signal* representing no signal noise.

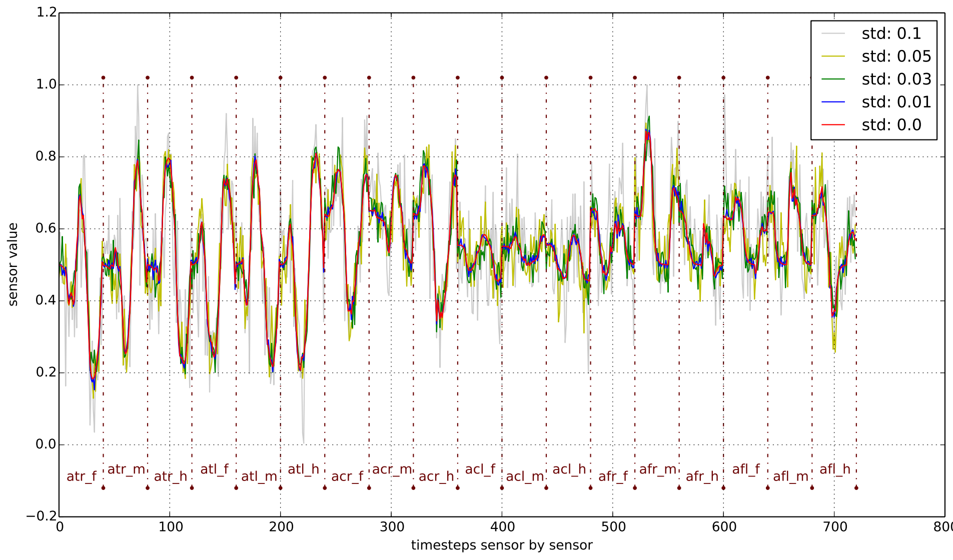


FIGURE 4.15: Examples of noisy signals: concrete, angle sensors

4.6 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.

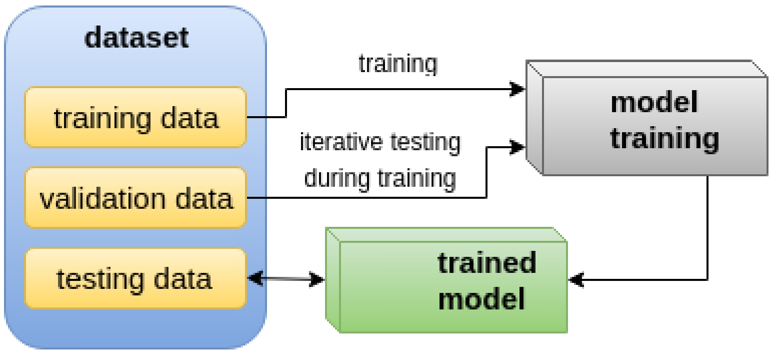


FIGURE 4.16: Three sets of data in a dataset.

Each set of data consists of samples and targets (class labels). The samples are represented by normalised feature vectors (section 4.5) - 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 4.4) 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 4.1) 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. 4.17.

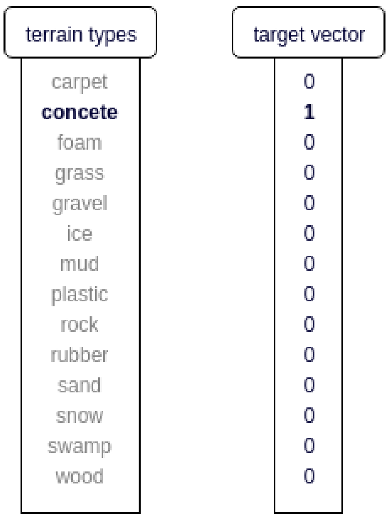


FIGURE 4.17: Target vector for concrete

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. 4.16. 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.

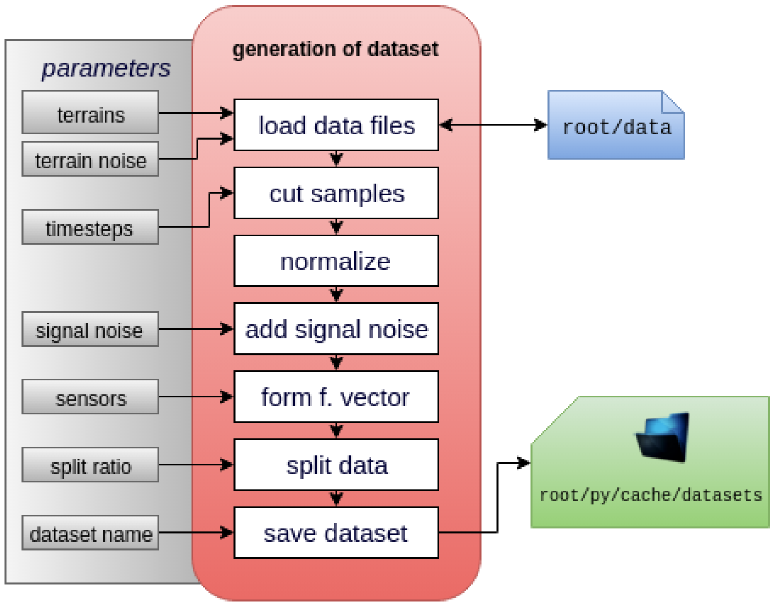


FIGURE 4.18: Workflow of generating a dataset

The workflow of data generation procedure is illustrated in Fig. 4.18.

Generated Datasets

TABLE 4.5: 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

4.7 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. 4.16.

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 3) 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 4.7.4).

In the overall process diagram (Fig. 4.1) there is a box called *classification with full networks*. The procedure behind this box is illustrated on Fig. 4.19.

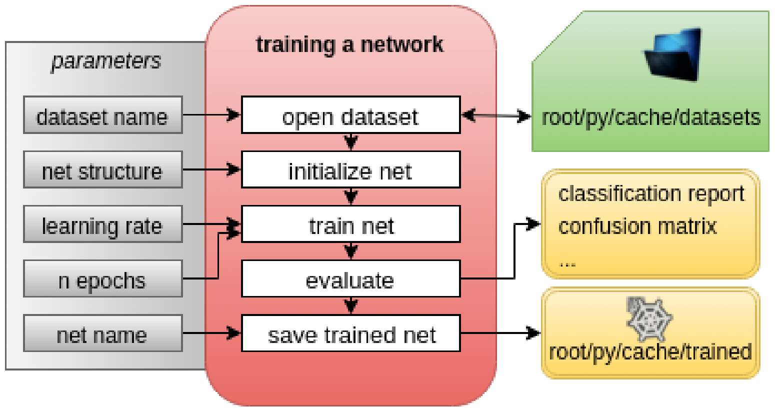


FIGURE 4.19: Procedure of training and testing a network

This workflow is performed by the implemented framework *kitt_nn* (chapter 3, as well as by other provided classifiers (discussed in section 4.7.4) 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 4.1) 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. 4.18) 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 3.3.

4.7.1 Evaluation Measures

A trained network is evaluated on testing data. This evaluation provides a set of the most important classification metrics (Pedregosa et al., 2011).

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:

$$F1 = \frac{2 * precision * recall}{precision + recall} \tag{4.12}$$

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.

classification error : DESCRIBE (formula).

average epoch time : DESCRIBE.

evarage classification time : DESCRIBE.

structure, n synapses ?

4.7.2 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.

4.7.3 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 ....

4.7.4 Other Classifiers

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

SVM, k-NN, RandomForest

Chapter 5

Experimental Evaluation

In this chapter the methods introduced in previous sections are evaluated and results are presented. In section 5.1 the implemented classification framework called *kitt_nn* (see chapter 3 and details in appendix A2.2) is verified by comparing to a publicly provided framework. Results of the developed pruning algorithm are shown in section 5.2.

Then, modules of the overall terrain classification process are gradually evaluated in (section 5.3).

5.1 Verification of the Network Implementation

Classification skills of the implemented method *kitt_nn* are compared to a *Scikit-neuralnetwork* classifier (*sknn*) presented in appendix A2.2. The evaluation is performed on two datasets introduced in section 3.4.3.

The following figure 5.1 shows the progress of classification accuracy within learning epochs. For each dataset/framework combination, 10 observations were performed and mean values with standard deviation ranges are shown.

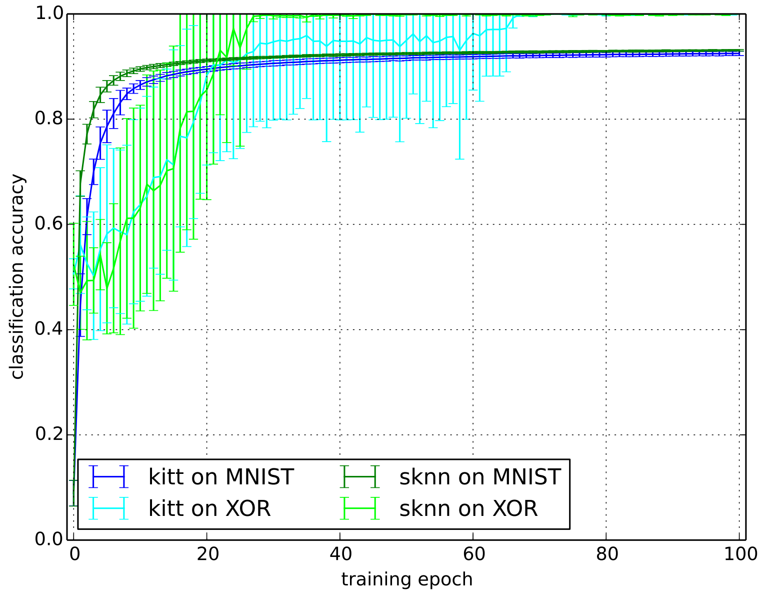


FIGURE 5.1: Comparison of the learning process to another framework (sknn).

Regarding the XOR dataset, both nets start with the accuracy of about 0.5, as it has 2 classes and, naturally, with accuracy of about 0.1 for the 10 classes of the MNIST dataset. Individual observations differ more to each other (see the standard deviation in Fig. 5.1) for XOR, as there are much less training samples compared to MNIST (50 times less). However, both nets are able to reach the accuracy of 1.0 on XOR within 100 epochs.

In Table 5.1, the *f1-score* measure (see Eq. (4.12) in **??**) is shown for individual classes (digits) of the MNIST dataset. This evaluation is done on the testing data.

TABLE 5.1: Comparison of f1-score on MNIST to another framework (sknn)

net	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*avg*
kitt	0.94	0.98	0.91	0.90	0.93	0.89	0.93	0.93	0.90	0.91	**0.92**
sknn	0.96	0.97	0.92	0.91	0.93	0.89	0.94	0.93	0.90	0.91	**0.93**

In Fig. 5.2, a comparison of average epoch processing time is shown. The *sknn* library provides an option to use *gpu* in order to decrease the training time, hence also this training variant is included to the comparison.

This evaluation is done on the MNIST dataset only, as the training is quite fast on XOR for both implementations due to the smaller amount and size of samples.

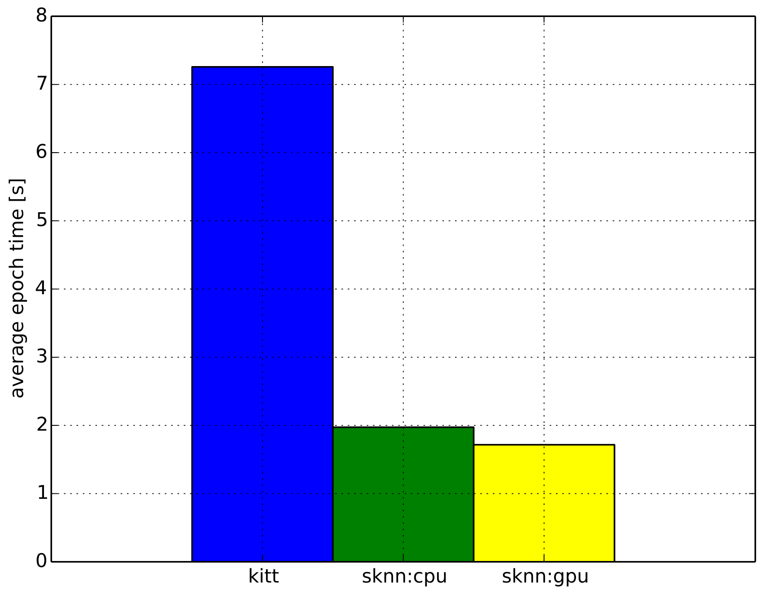


FIGURE 5.2: Comparison of average epoch processing time to another framework (on MNIST).

The implemented *kitt_nn* framework cannot compete in speed with the optimized stochastic *GDA* (see (Champandard and Samothrakis, 2015)) of *sknn* library. However, importantly for this study, the classification abilities have been verified.

5.2 Pruning Algorithm Results

This section presents results of the implemented pruning algorithm (introduced in section 3.4). The input of the algorithm is given by a dataset and an obviously oversized neural network with a fully-connected structure. On the output, a pruned network of a minimal structure, but keeping a required classification accuracy, is expected.

The evalutaion of the algorithm is performed on the two datasets described in section 3.4.3: XOR and MNIST.

5.2.1 Evaluation on XOR Dataset

The XOR dataset is essential for testing the functionality of the algorithm, as we know the minimal network structure for this problem ($[2, 2, 1]$). It is chosen to use a network with one hidden layer of 100 neurons as the algorithm input (Fig. 5.4a). The desired structure is shown in Fig. 5.4b.

During the pruning process, three key network properties are observed:

1. network structure
2. number of synapses in the network
3. classification accuracy on a chosen dataset

Those are shown with respect to the pruning step (see the pruning loop in Fig. 3.7) all together in Fig. 5.3.

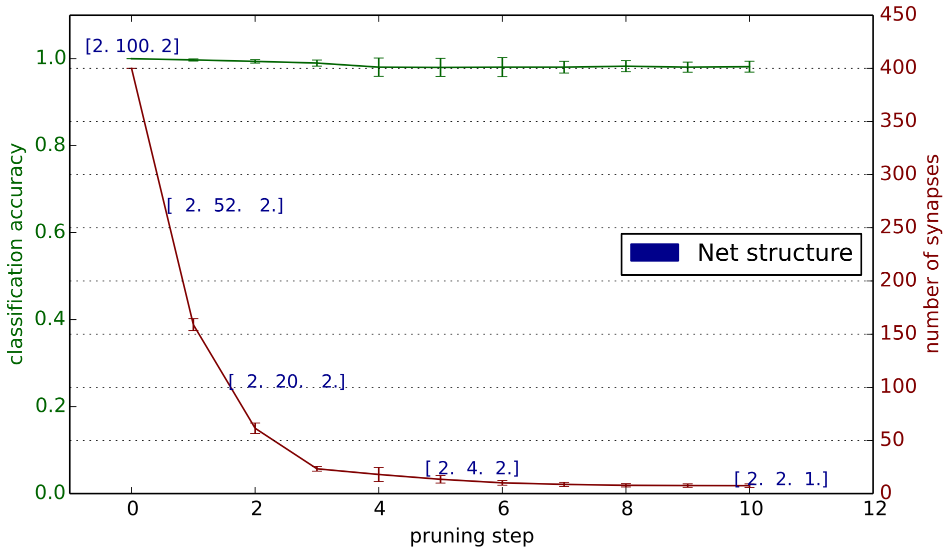


FIGURE 5.3: Pruning Algorithm Results on the XOR Dataset.

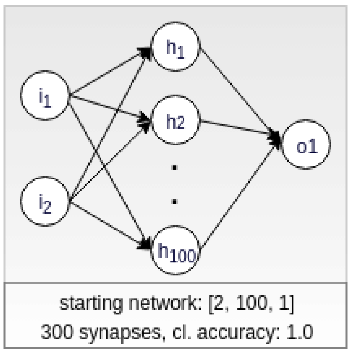
The results are obtained by averaging 10 observations, so they are not dependent on initial conditions. For retraining the network after a pruning step, training data is used. For verification, whether the network is capable of classification, validation data is used. Testing data is used to check the accuracy after pruning (values in Fig. 5.3).

Algorithm parameters:

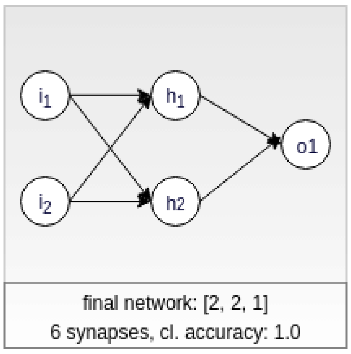
- initial network structure: $[2, 100, 1]$
- required classification accuracy on validation data: 0.9
- learning rate: 0.5
- percentile levels: $[50, 20, 5, 0]$

Algorithm outcomes:

- pruning steps: 10
- final strucuture: $[2, 2, 1]$
- number of removed synapses: 294 (300 -> 6)
- classification accuracy on testing data: 0.99


i1
i2
h1
h2
h100
o1
starting network: [2, 100, 1]
300 synapses, cl. accuracy: 1.0

(A) PA input: oversized and fully-connected network (XOR)



(B) PA output: minimal network structure (XOR)

FIGURE 5.4: Application of the pruning algorithm on XOR

5.2.2 Evaluation on MNIST Dataset

Similarly, the algorithm is evaluated on the MNIST dataset. In this case, the minimal structure is not known. The recommended size of the hidden layer for MNIST classification is 15. In Fig. 5.1 the classification accuracy of *kitt_nn* framework on this dataset is presented. Apparently, it is possible to reach a success rate around 90%. Hence these parameters are used:

- initial network structure: $[784, 15, 10]$ (784, because the samples are images of size $28x28$, and 10, because we have ten digits)
- required classification accuracy on validation data: 0.89
- learning rate: 0.1
- percentile levels: $[50, 35, 20, 10, 5, 0]$

Fig. 5.5 shows a significant reduction of synapses while the classification accuracy is kept on 0.89. The shown results are obtained by averaging 10 observations.

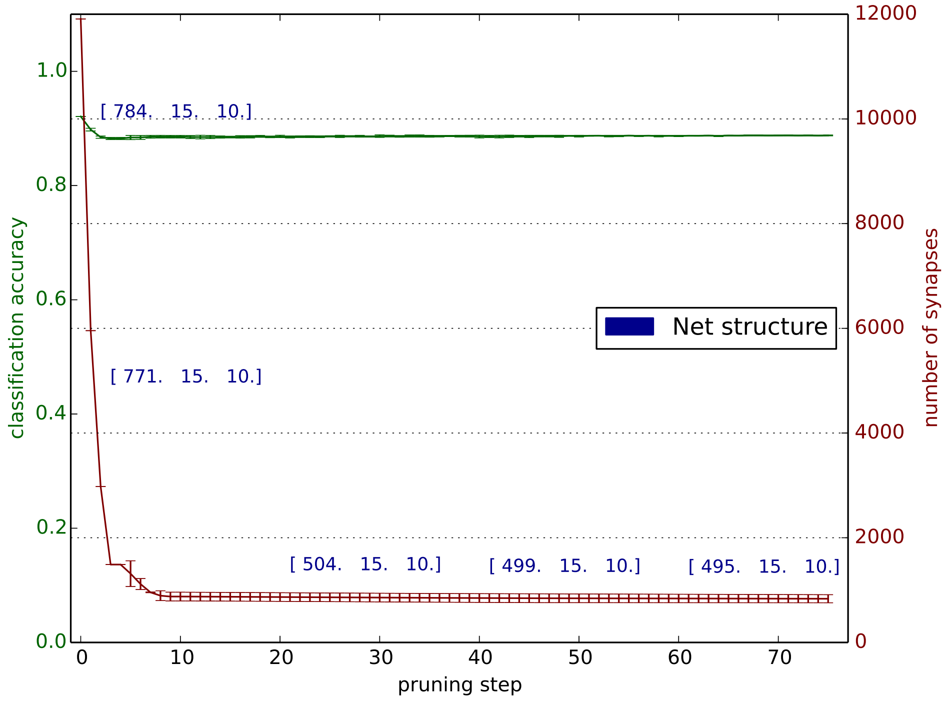


FIGURE 5.5: Pruning Algorithm Results on MNIST Dataset.

Results of the pruning algorithm on MNIST:

- pruning steps: 76
- final strucuture: $[495, 15, 10]$
- number of removed synapses: 11075 (11910 -> 835 : 92.9%)
- classification accuracy on testing data: 0.88776

The figure 5.5 says that more than 90% of the synapses in the initial network structure are redundant. Regarding the number of neurons in the network, mostly the input layer is reduced. These results make a pretty good sense, as shown in section 5.2.2.

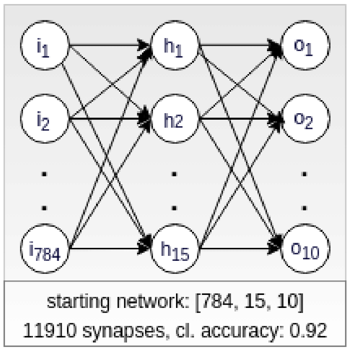
The following Table 5.2 presents statistics of the first seven steps of one of the performed observations. For each step we can see the current percentile level (see Fig. 3.7), corresponding reduction in the network and retraining results.

TABLE 5.2: PA progress example on MNIST

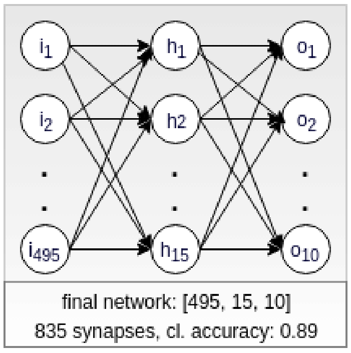
step	*0 (initial)*	*1*	*2*	*3*	*4*	*5*	*6*	*7*
percentile level	50	50	50	50	50	35	20	20
synapses removed	X	5955	2978	1489	744	521	298	238
input neurons left	784	784	769	664	464	539	602	521
synapses left	11910	5955	2977	1488	744	967	1190	952
acc. after pruning	X	0.9	0.887	0.851	0.817	0.858	0.880	0.878
retrained?	X	yes	yes	yes	no	no	yes	yes
epochs to retrain	X	0	4	63	>100	>100	8	12

In this example, the algorithm uses the percentile level of 50 in the first three steps, meaning it cuts out the half of the synapses each time. In the fourth step, the network is not able to retrain after cutting half of the remaining synapses, hence the percentile level is decreased to 35 and less synapses are tried to be removed in the fifth step. However, the retraining is not successful again (the accuracy of 0.89 is not reached in 100 epochs), so the percentile level is decreased to 20. This time the pruned network is able to retrain, so a new structure is saved and the algorithm continues with the percentile level of 20 in the next steps.

The following figures (5.6) illustrate the transformation of the network.


i1
i2
i784
h1
h2
h15
o1
o2
o10
starting network: [784, 15, 10]
11910 synapses, cl. accuracy: 0.92

(A) PA input: oversized and fully-connected network (MNIST)


i1
i2
i495
h1
h2
h15
o1
o2
o10
final network: [495, 15, 10]
835 synapses, cl. accuracy: 0.89

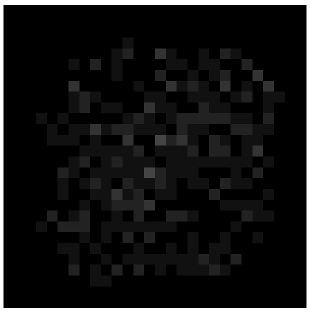
(B) PA output: minimal network structure (MNIST)

FIGURE 5.6: Application of the pruning algorithm on MNIST

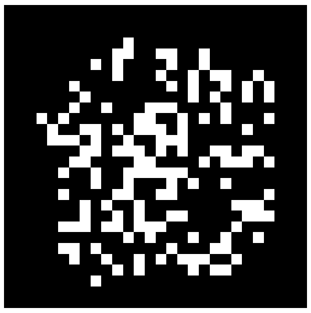
Minimal Structure Analysis for MNIST

As mentioned in section 3.4.4, minimal structures obtained from the pruning algorithm can be further researched.

For instance, considering the MNIST dataset, there is an image (meaning a vector of pixels) as the network input. As the output, there are 10 classes corresponding to digits. Having the minimal structure, one can find out which pixels are related to e.g. digit 7 class or which pixels are totally useless for classification.



(A) Pixels of the MNIST samples that are essential for classification



(B) Pixels of the MNIST samples influencing the class 7

FIGURE 5.7: Demonstration of the minimal structures utilization on MNIST

As illustrated in Fig. 3.13, paths from the input to the output layer can be tracked. This way we can find paths for every output neuron and see which input neurons are connected with it. Fig. 5.7b shows in white all input neurons (pixels of the $28x28$ input images) that are connected to output neuron 7.

In Fig. 5.7a, an analysis of all 784 input neurons is presented. As there are 15 hidden neurons, each input neuron can have maximally 15 connections and, naturally, minimially 0 connections in the final structure.

The number of remaining connections n_con is mapped into $[0, 255]$ range, in order to be represented as a greyscale color, using the following equation:

$$intensity = \frac{255 \cdot n\_con}{15} \tag{5.1}$$

Fig. 5.7a shows the number of connections to the hidden layer n_con as a greyscale value ($intensity$) for every pixel of input samples. The more bright the pixel is, the more connections it has. Black pixels have no connections to the hidden layer and so they are not used for classification at all.

The results on both figures (5.7a and 5.7b) make a good sense, as the pixels close to borders are black. One could expect, that these pixels are not important, as the digits are centered in the images.

5.3 Terrain Processing Results

In this section, results of the overall terrain classification process are shown. The results are sorted in correspondance to the methods explanation in chapter 4.

5.3.1 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. (5.2). The five qualities are listed in Table 4.3 and in Table 4.4.

$$SM_{t_1,t_2} = \frac{\sum_{i=1}^{5} \left| quality(i, t_1) - quality(i, t_2) \right|}{5} \tag{5.2}$$

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. 5.8 shows the similarity measures among generated terrains.

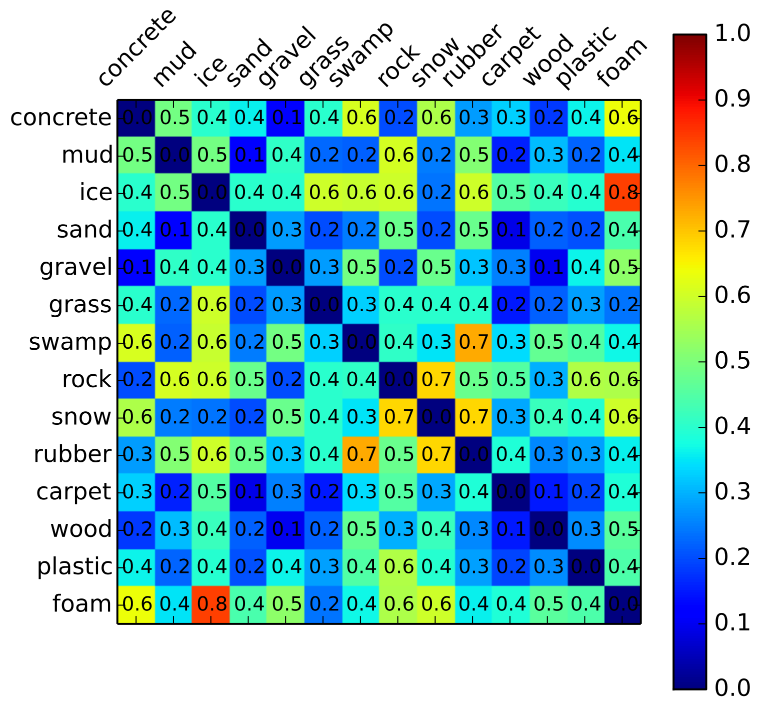


FIGURE 5.8: Similarity measures among various terrain types.

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.

5.3.2 Classification Results

Network Learning Parameters

learning rate

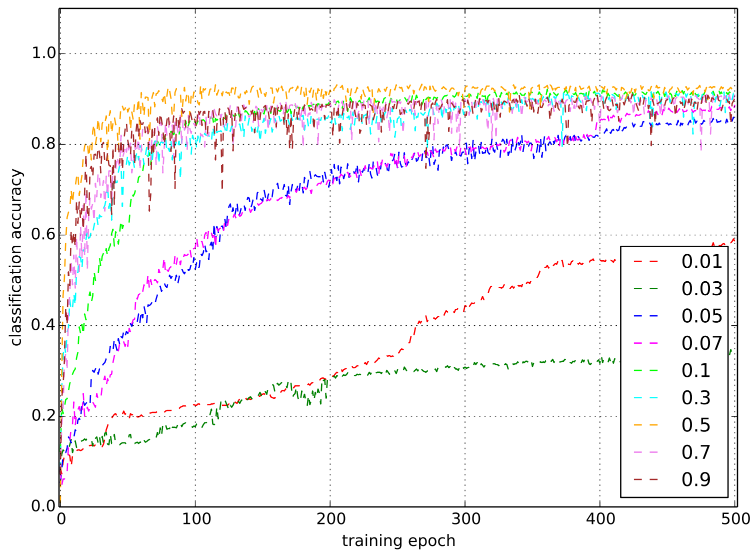


FIGURE 5.9: Learning rate analysis. Net structure: [960, 20, 14].

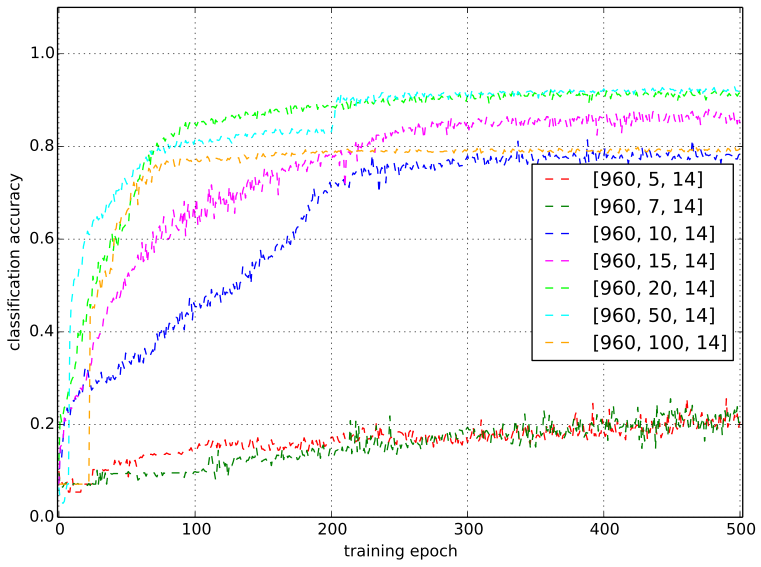


FIGURE 5.10: Network structure analysis. Learning rate: 0.1.

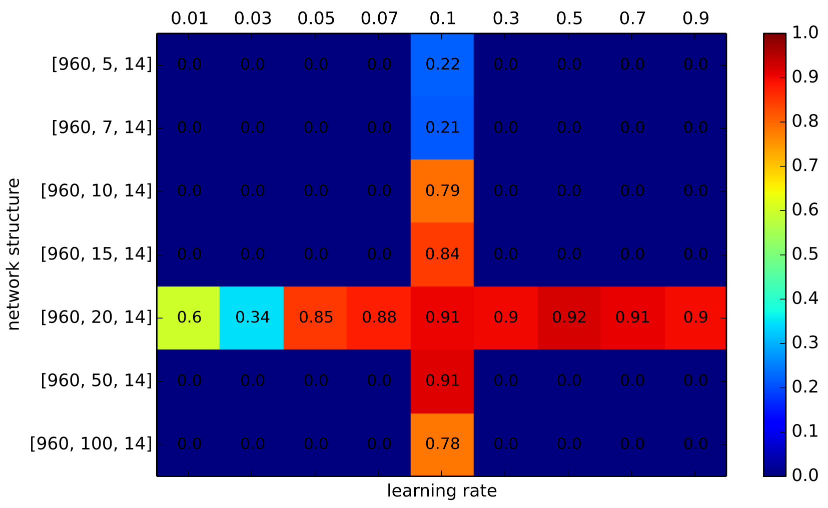


FIGURE 5.11: Learning parameters analysis: Classification accuracy vs. learning rate and network structure

Required Number of Timesteps

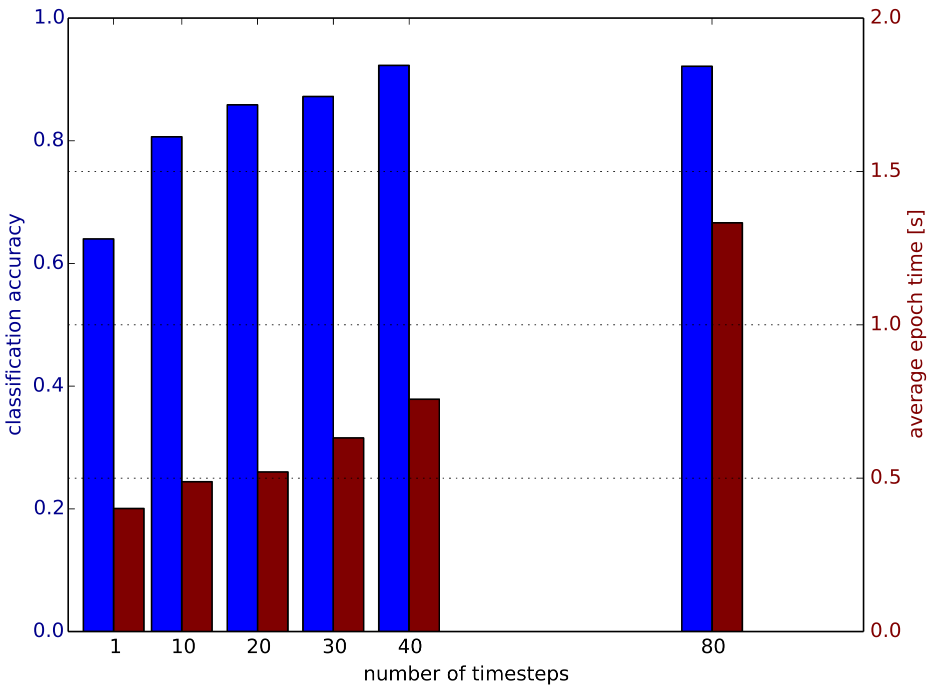


FIGURE 5.12: Number of simulation timesteps: influence on the accuracy and training time

5.3.3 Allowed Data Noise

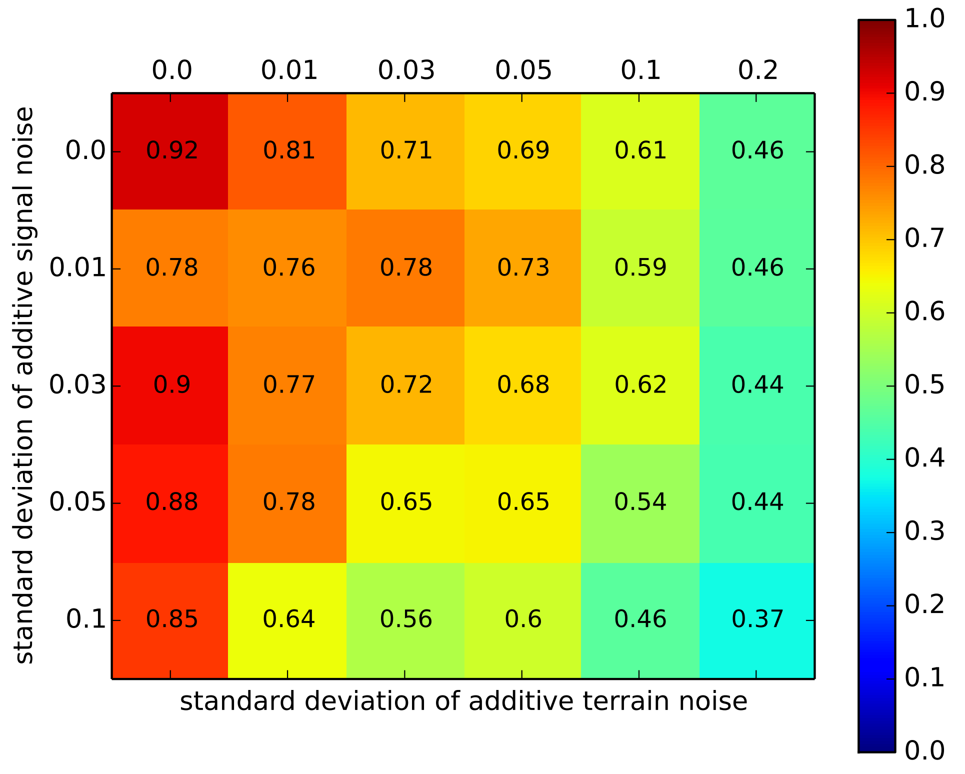


FIGURE 5.13: Additive terrain and signal noise: influence on the accuracy

5.3.4 Needed Sensors

Results Complete Overview

TABLE 5.3: Classification results

dataset	*accuracy*	*precision*	*recall*	*f1-score*	*c. error*	*avg. ep. time [s]*
ds_01	0.923	0.920	0.920	0.920	0.015	0.757
ds_02	0.682	0.660	0.680	0.640	0.068	0.650
ds_03	0.708	0.700	0.710	0.680	0.063	0.439
ds_04	0.640	0.650	0.640	0.620	0.071	0.401
ds_05	0.806	0.810	0.810	0.800	0.039	0.488
ds_06	0.921	0.920	0.920	0.920	0.015	1.332
ds_07	0.776	0.790	0.780	0.760	0.039	0.751
ds_08						
ds_09						
ds_10						
ds_11						
ds_12						
ds_13						
ds_14						
ds_15						

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 6

Discussion

Chapter 7

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:

(Zenker et al., 2013) and (Kesper et al., 2012) and (Xiong, Worgotter, and Manoonpong, 2014) and (Mou and Kleiner, 2010) and (Coyle, 2010) and (Hoepflinger et al., 2010) and (Ahmed, 2015) and (Ordonez et al., 2013) and (Bermudez et al., 2012) and (**article:10:pruningalgs**) and (Spenneberg and Kirchner, 2007) and (Belter, 2011)

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.

[LC98] Yann LeCun and Corinna Cortes. *The MNIST database of handwritten digits*. 1998. URL: http://yann.lecun.com/exdb/mnist/.

[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.

[Lab14] Welch Labs. *Neural Networks Demystified.* Youtube. 2014. URL: https://www.youtube.com/watch?v=bxe2T-V8XRs.

[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

Methods Implementation

A2.1 Implementation of the Terrain Classification

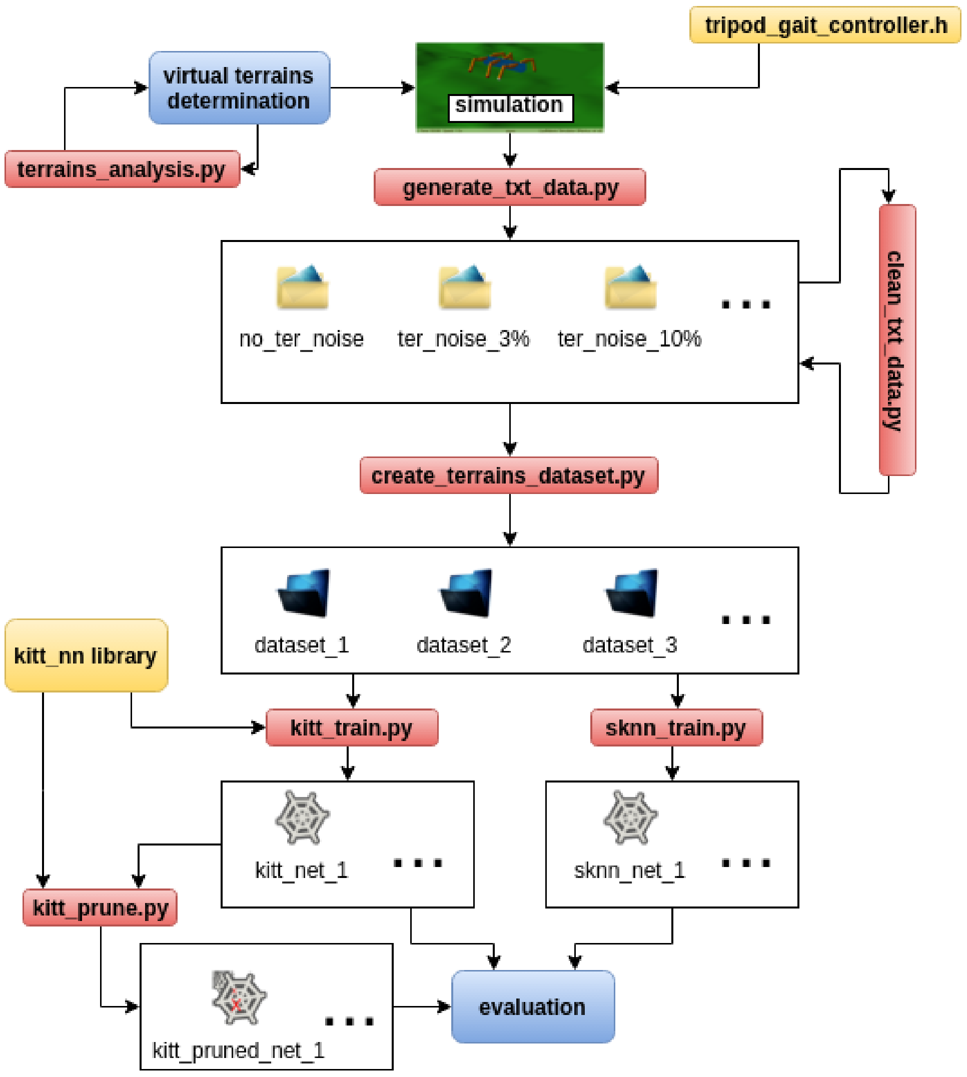


FIGURE A2.1: Terrain classification process - overall diagram.

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.

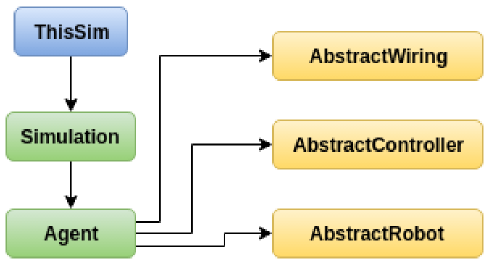


FIGURE A2.2: Software architecture for LPZRobots and GoRobots. (*Research Network for Self-Organization of Robot Behavior*)

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 4.1).

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/*.

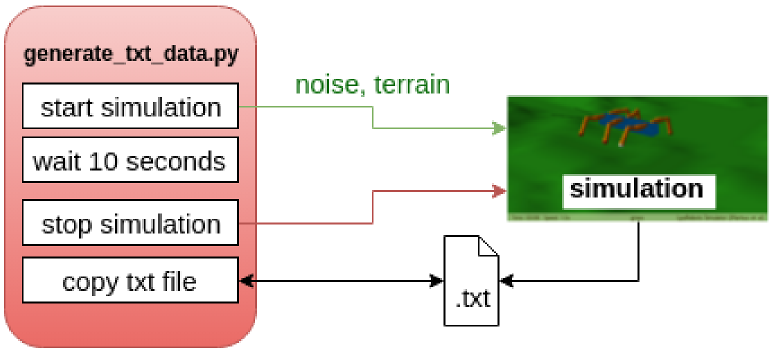


Figure A2.3: The process of data acquisition from the simulation.

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.

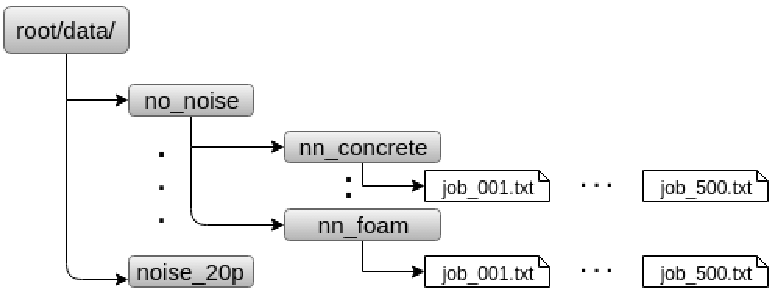


Figure A2.4: The structure of rough data directory.

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 4.6) - using the *pickle (cPickle)* package, just with the *.net* suffix.

A2.2 Implementation of the Neural Network

The following diagram (A2.5) shows the structure of the *kitt_nn .py package.*

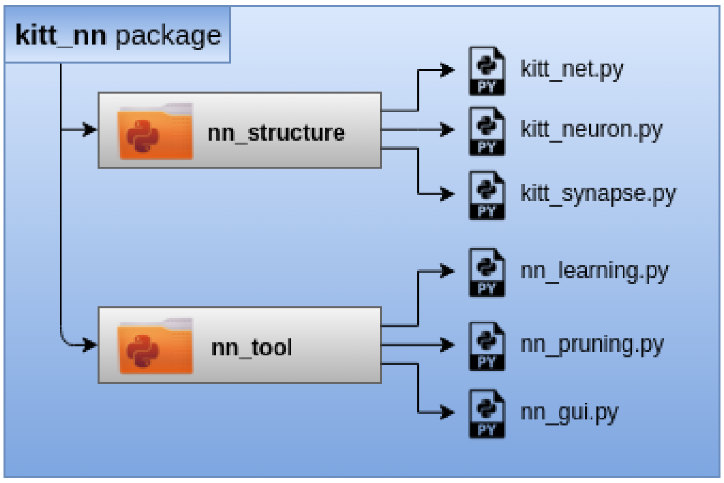


FIGURE A2.5: kitt_nn package : Implemented neural network framework

The *kitt_nn* implementation is based on some general knowledge gained at school and/or from (Labs, 2014), the idea is pretty straight forward.

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

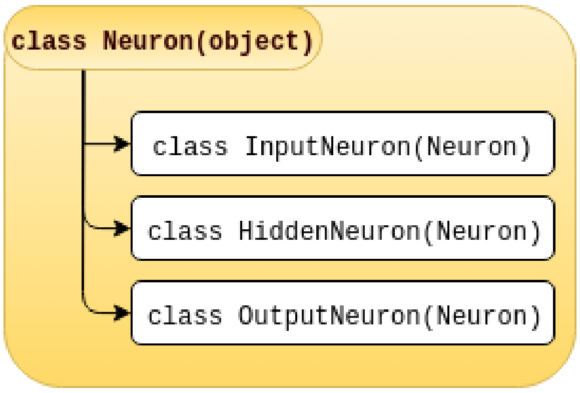

class Neuron(object)
class InputNeuron(Neuron)
class HiddenNeuron(Neuron)
class OutputNeuron(Neuron)

FIGURE A2.6: kitt_neuron.py : Neuron class inheritance

Then, as Fig. A2.6 shows, three classes are inherited from the *Neuron* class. Some special functions, like fitting a sample in case of input layers or producing network outcome by output layers respectively, can be implemented this way, while some common functions are shared in the mother *Neuron* class.

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 a database 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. A2.7. 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.

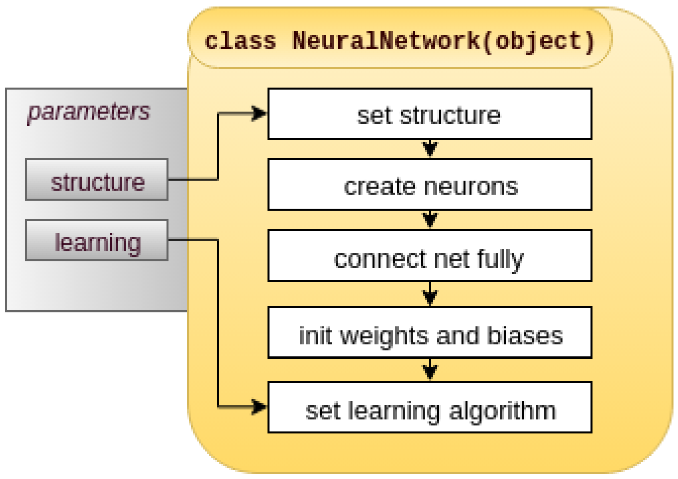


FIGURE A2.7: kitt_net.py : Neural Network Initialization

A learning algorithm is added to the initialized network thereafter (see section 3.3). 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 work (section 3.5, section 3.4).

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 (Champandard and Samothrakis, 2015), 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.

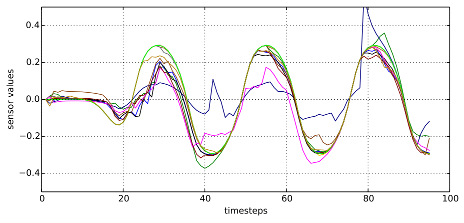
Material suitable for inclusion in appendices includes the following:

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.

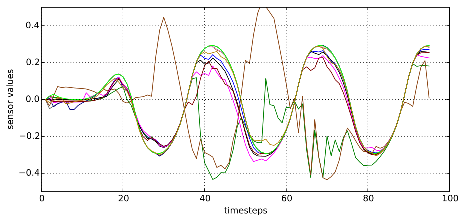
Appendix A4

Detailed Results

A4.1 Complete Sensory Data

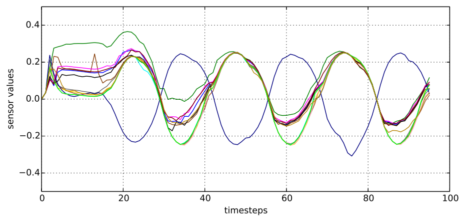


(A) ATRf

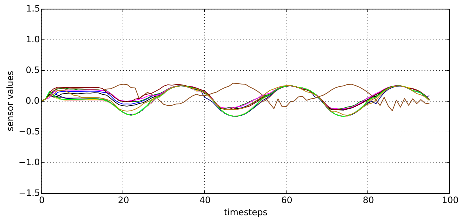


(B) ATLf

Figure A4.1: Thoraco joints on the front legs

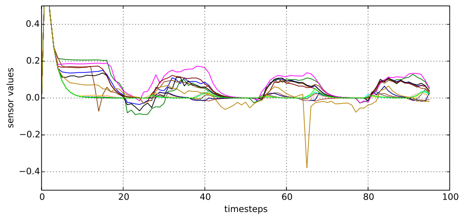


(A) ACRf

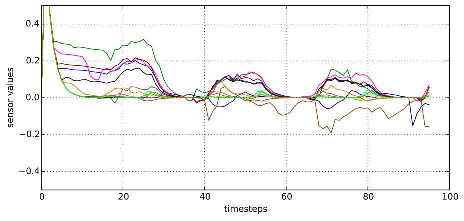


(B) ACLf

Figure A4.2: Coxa joints on the front legs

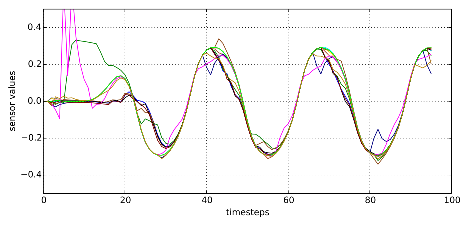


(A) AFRf

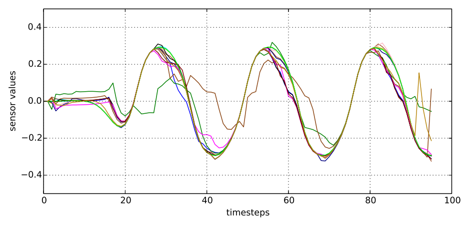


(B) AFLf

Figure A4.3: Femur joints on the front legs

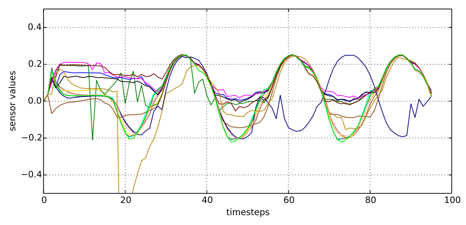


(A) ATRm

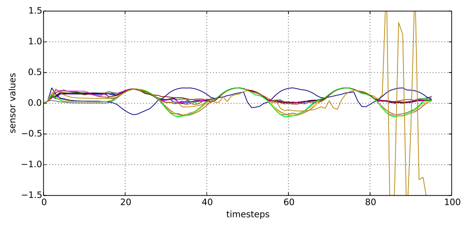


(B) ATLm

FIGURE A4.4: Thoraco joints on the middle legs

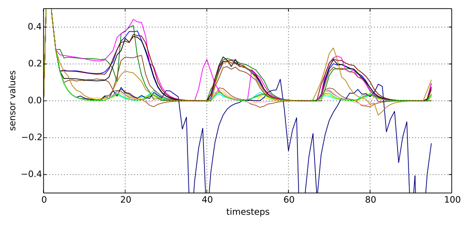


(A) ACRm

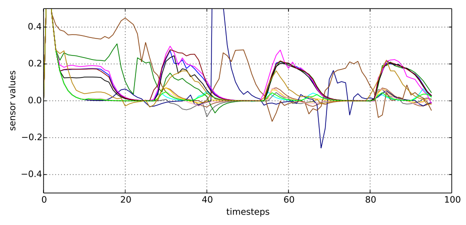


(B) ACLm

FIGURE A4.5: Coxa joints on the middle legs

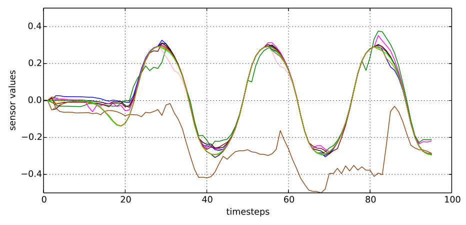


(A) AFRm

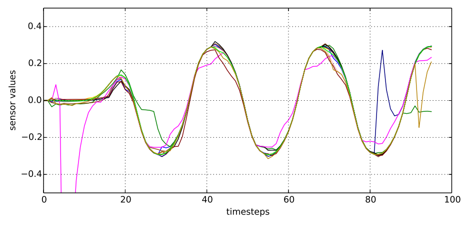


(B) AFLm

FIGURE A4.6: Thoraco joints on the middle legs

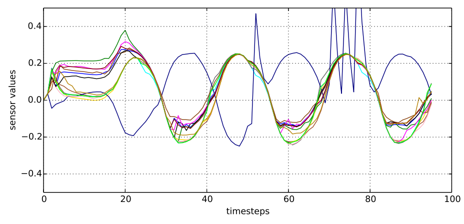


(A) ATRh



(B) ATLh

FIGURE A4.7: Thoraco joints on the hint legs


(A) ACRh

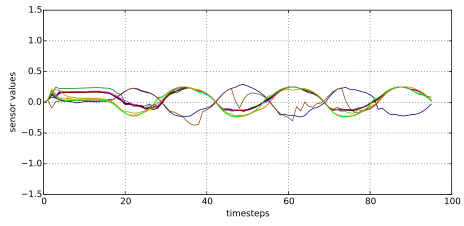
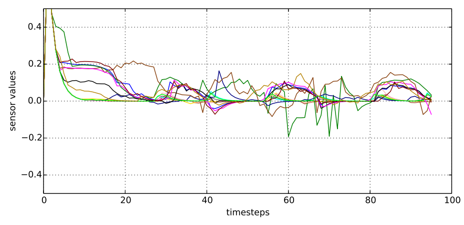

(B) ACLh

Figure A4.8: Coxa joints on the hint legs


(A) AFRh

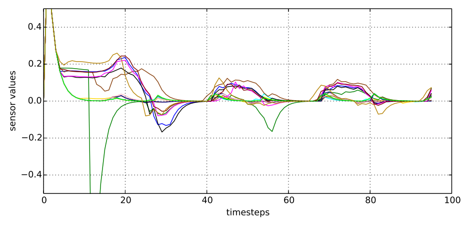
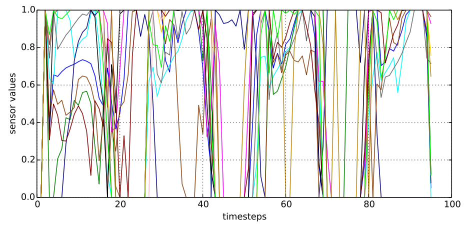

(B) AFLh

Figure A4.9: Thoraco joints on the hint legs


(A) FRf

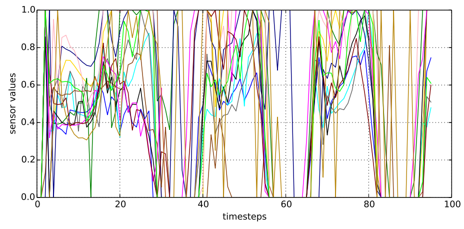
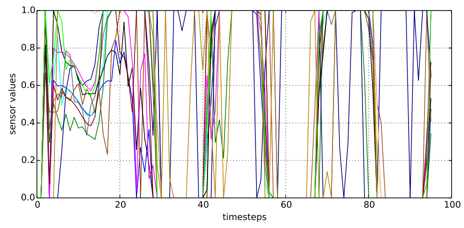

(B) FLf

Figure A4.10: Foot contact sensors on the front legs


(A) FRm

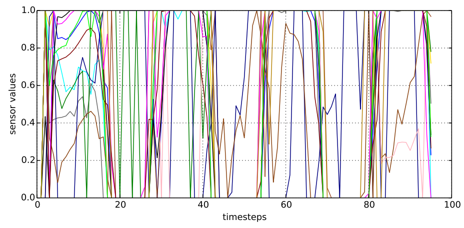
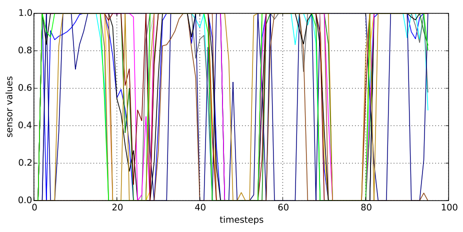
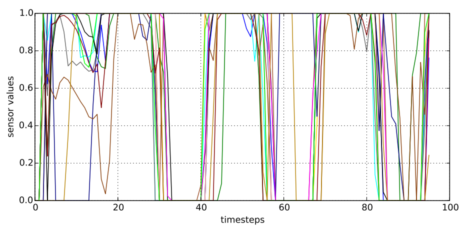

(B) FLm

Figure A4.11: Foot contact sensors on the middle legs



(A) FRh



(B) FLh

Figure A4.12: Foot contact sensors on the hint legs

A4.2 Further Data Analysis

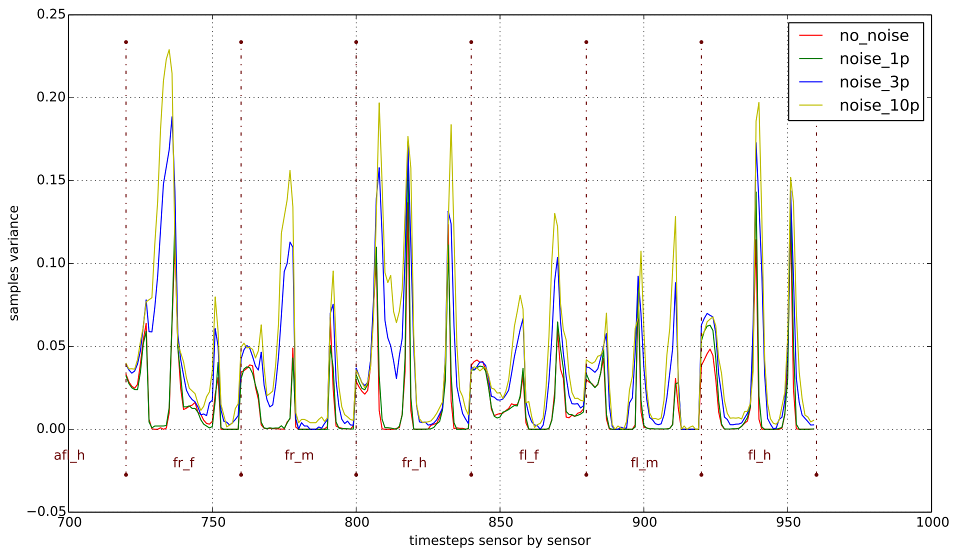


Figure A4.13: Terrain Noise Analysis (samples variance): 500 samples, terrain gravel, foot contact sensors (feature vector [720:960] for 40 timesteps)

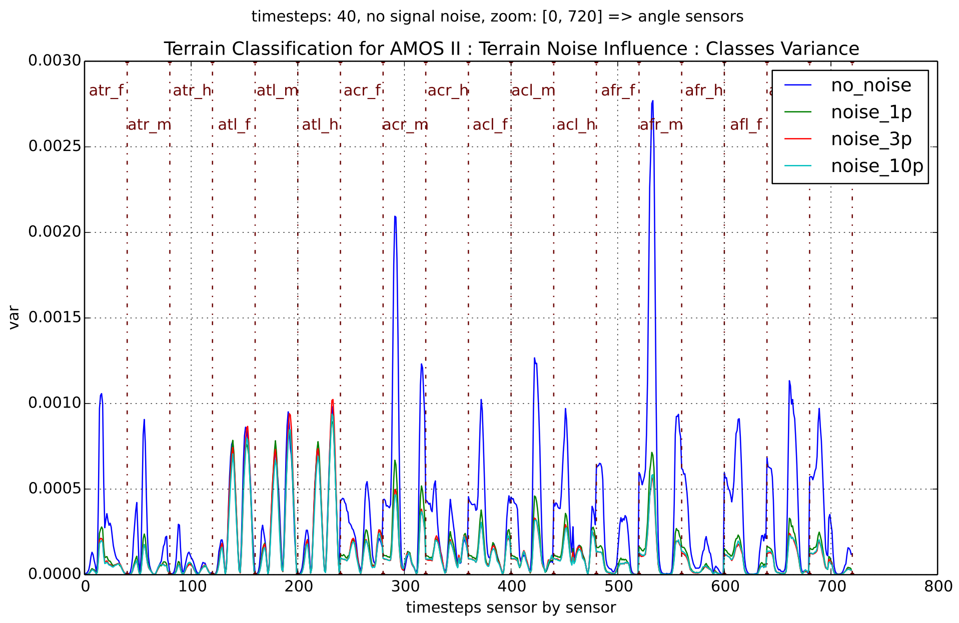


FIGURE A4.14: Terrain Noise Analysis (classes variance): means of 500 samples, 14 terrains, angle sensors

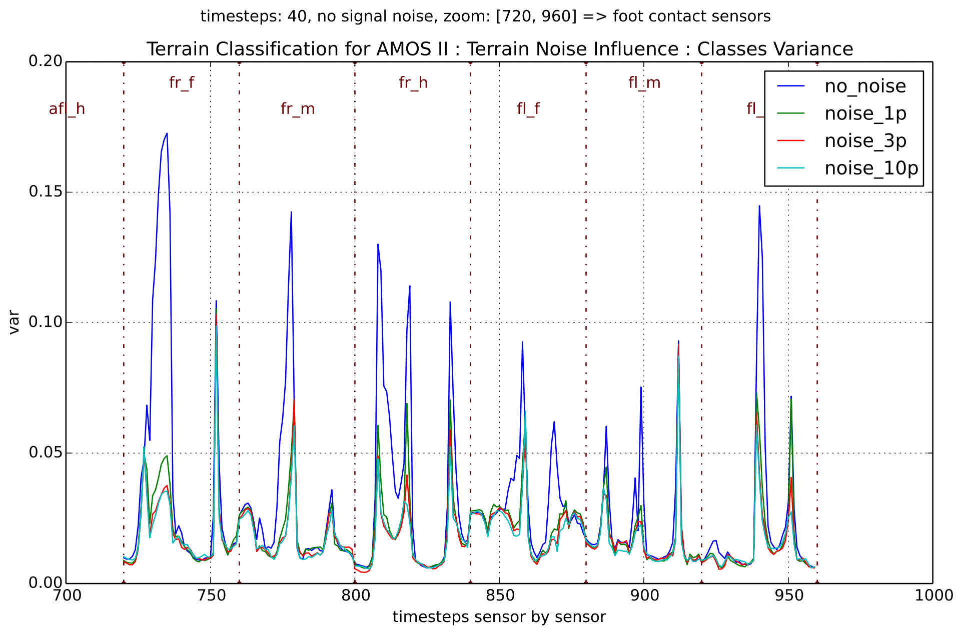


FIGURE A4.15: Terrain Noise Analysis (classes variance): means of 500 samples, 14 terrains, foot contact sensors

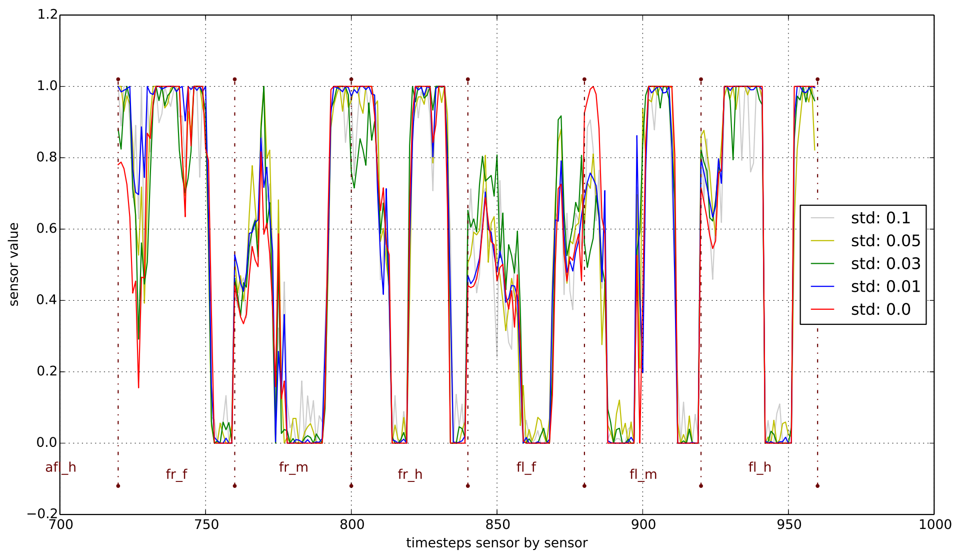


FIGURE A4.16: Signal noise influence on one sample, terrain: concrete, foot contact sensors