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
Maersk Mc-Kinney Moller Institute

May 27, 2016

Declaration of Authorship

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

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

Signed:

Date:

"*It is our choices... that show what we truly are, far more than our abilities.*"

J. K. Rowling

UNIVERSITY OF SOUTHERN DENMARK

Abstract

Faculty of Engineering

Maersk Mc-Kinney Moller Institute

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

Proprioception and tactile sensing in insect-like legged robots is a fast, illumination insensitive and biologically inspired way of ground perception. In this thesis, 14 virtually generated terrains are classified based on the mentioned sensor types for a simulated version of hexapod robot AMOS II. A feedforward neural network framework equipped with a novel network pruning algorithm has been developed for classification. We observe over 92% classification accuracy on deterministic terrain data and 72% on manually noised data. The pruning algorithm removes unimportant synapses (generally more than 90%) from a fully-connected network, while the classification accuracy does not drop significantly. The number of input neurons is reduced by 65%, resulting in the minimal network structure for the classification problem. A theory of using minimal structures for feature selection is proposed. The thesis outcome consists of a minimal neural network capable of terrain classification based on selected features of proprioceptive and tactile sensory signals.

Acknowledgements

I would like to express my sincere gratitude to my advisor Dr. Tomas Kulvicius for the continuous support of my work, for his valuable comments and supervision of my master thesis.

Next, I would like to thank my second supervisor Dr. Poramate Manoonpong for helping me get started with the simulation and for a great introduction to neural networks during his courses at the University of Southern Denmark.

Finally, I would like to thank people from the Research Network for Self-Organization of Robot Behavior in Leipzig for providing the LpzRobots simulator, which was used for the data collection.

Contents

Abstract **iii**

1 Introduction **1**
1.1 State of the Art 2
1.2 Master Thesis Objectives 6
1.3 Relation to the State of the Art 6
1.4 Thesis Outline 7

2 Classification Method **8**
2.1 Network Structure 8
2.2 Neuron Principle 9
2.3 Learning Algorithm for Network Training 10
2.3.1 Using Mini-batches 11
2.3.2 Matrix Notations 11
2.3.3 Forward Propagation 12
2.3.4 Error Calculation 12
2.3.5 Parameter Update 12
2.4 Network Pruning Algorithm 14
2.4.1 Pruning Method 14
2.4.2 Algorithm Realization 14
2.4.3 Datasets for Evaluation of the Pruning Algorithm . . 17
2.4.4 Using Network Pruning for Feature Selection 19
2.5 Graphical User Interface 20

3 Terrain Classification for Hexapod Robot AMOS II **21**
3.1 Overall Process Summary 21
3.2 Experimental Environment Specification 23
3.2.1 Hexapod Robot AMOS II 23
3.2.2 AMOS II Simulation 25
3.2.3 Tripod Gait Controller 27
3.3 Generation of Virtual Terrains 28
3.3.1 Terrain Features 29
3.3.2 Features Determination for Various Terrains 29
3.3.3 Terrain Noise 31
3.4 Data Acquisition 32
3.5 Building a Feature Vector 35
3.5.1 Feature Vector Normalisation 36
3.5.2 Signal Noise 37
3.6 Generation of Datasets 39
3.7 Training and Classification 40
3.7.1 Evaluation Measures 41

4 Experimental Evaluation **42**
4.1 Verification of the Network Implementation 42
4.2 Performance Evaluation of the Pruning Algorithm 44
4.2.1 Evaluation on XOR Dataset 44
4.2.2 Evaluation on MNIST Dataset 45
Analysis of Minimal Structure in MNIST Dataset . . . 47
4.2.3 Comparison to Other Pruning Methods 48
4.3 Results of Terrain Classification 49
4.3.1 Classification Performance 49
Comparison to Other Classification Methods 52
4.3.2 Selection of Learning Parameters 53
4.3.3 Influence of Noise on Classification 54
4.3.4 Time Needed for Classification 55
4.3.5 Analysis of Used Sensor Types 56
4.4 Terrain Classification Using Network Pruning 57
4.4.1 Feature Selection for Terrain Classification 61

5 Discussion **64**
5.1 Methods Recapitulation 64
5.2 Comparison of Results 66

6 Conclusion and Outlook **67**

Bibliography **68**

A1 Supplementary Data **70**
A1.1 Sensory Data Examples 70
A1.2 Generated Datasets 73
A1.3 Supplementary Figures for Feature Analysis 74

A2 Method Implementation **76**
A2.1 Implementation of the Neural Network 76
A2.2 Implementation of the Terrain Classification 79

A3 Structure of the Workspace **83**

A4 Code Documentation **84**
A4.1 Neural Network Framework KITTNN (API) 84
A4.2 Terrain Classification Scripts (API) 85

List of Figures

1.1 Illustration of a classification problem. 2

2.1 Structure of a feedforward neural network 8
2.2 A model neuron 9
2.3 Used transfer functions: sigmoid and tanh 10
2.4 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 the number of iterations (epochs). The learning process is stopped after a specified number of epochs, no matter how successful the training has been. 10
2.5 Pruning Algorithm: hypothesis formulation 14
2.6 Overall flowchart of the pruning process 15
2.7 The Pruning Algorithm: initialized variables are in bold, red marked functions refer to 2.8 and 2.9 respectively 15
2.8 Synaptic pruning based on weight changes and current percentile value 16
2.9 Evaluation of the classification accuracy after pruning . . . 16
2.10 2D XOR Data illustration 18
2.11 XOR Dataset: minimal network structures 18
2.12 MNIST Data illustration (LeCun and Cortes, 1998) 19
2.13 Analysis of the minimal structure examplified on digit 5 (MNIST dataset) 19
2.14 Screenshot of the graphical user interface 20

3.1 Terrain Classification: overall process diagram 21
3.2 AMOS II. (Manoonpong, 2011) 23
3.3 Structure of the AMOS's leg. (Manoonpong, 2011) 24
3.4 Structure of the two repositories: LPZRobots and GoRobots. (Martius et al., 2009) 26
3.5 Virtual alternative for AMOS II. 26
3.6 2-neuron network oscillator ("Adaptive Embodied Locomotion Control Systems") 27
3.7 Schematic diagram of tripod gait controller 28
3.8 Similarity measures among various terrain types. 30
3.9 Examples of noisy terrains: terrain rock, angle sensors . . . 32
3.10 Thoraco Sensor (ATRf) output examples, 14 terrains 33
3.11 Coxa Sensor (ACRm) output examples, 14 terrains 33
3.12 Femur Sensor (AFRh) output examples, 14 terrains 34
3.13 Foot Contact Sensor (FRf) output examples, 14 terrains . . 34
3.14 Forming a feature vector out of a data file. 35

3.15 Normalised feature vector examples 37
3.16 Examples of noisy signals: concrete, angle sensors 38
3.17 Three sets of data in a dataset. 39
3.18 Target vector for concrete 39
3.19 Procedure of training and testing a network 40

4.1 Learning process compared to another framework (Scikit-neuralnetwork (sknn): (Champandard and Samothrakis, 2015)). 42
4.2 Comparison of average epoch processing time (1000 samples) to another framework (Scikit-neuralnetwork (*SKNN*): (Champandard and Samothrakis, 2015)), MNIST dataset . 43
4.3 Results of the pruning algorithm on XOR dataset. 44
4.4 PA process illustration on XOR 45
4.5 Pruning Algorithm Results on MNIST Dataset. 46
4.6 PA process illustration on MNIST 47
4.7 Feature Selection : MNIST Analysis. 48
4.8 Comparison of the developed PA to other pruning methods (10 observations). 48
4.9 Confusion matrix of classification results on a deterministic dataset. 51
4.10 Confusion matrix of classification results on a noisy dataset. 51
4.11 Training process with various learning rates. 53
4.12 Training process with various networks differing in the number of hidden neurons. 53
4.13 Classification accuracy vs. learning rate and network structure (10 observations) 54
4.14 Additive terrain and signal noise: influence on the accuracy (10 observations). 54
4.15 Analysis of time needed for proper classification (10 observations). 55
4.16 Average epoch time (10 observations) depending on the number of simulation timesteps. 55
4.17 Evaluation of different sensor types separately (average of 10 observations, timesteps: 10, 40, 80). 56
4.18 Average epoch time for different sensor types. 57
4.19 Pruning Algorithm Results on AMTER Dataset. No noise. 58
4.20 Synaptic pruning of configurations A (reference), B (80 timesteps) and D (noisy data). 59
4.21 Synaptic pruning of configurations A' (reference), C (100 hidden neurons), E (proprioceptive sensors) and F (tactile sensors). 59
4.22 Active neurons in the network after pruning [%]: configurations A, B, D (10 observations) 60
4.23 Active neurons in the network after pruning [%]: configurations A', C, E, F (10 observations) 60
4.24 Number of paths to the output layer for every feature of the input example. 61
4.25 Explanation of a path from input to output layer in a minimal structure. 61
4.26 Used features of thoraco-coxa proprioceptive sensors for individual classes. 62

4.27 Used features of tactile sensors for individual classes. 62
4.28 Influence power of single features on classes: coxa sensors . 63
4.29 Influence power of single features on classes: tactile sensors 63

A1.1 Thoraco proprioceptive sensors, front legs 70
A1.2 Coxa proprioceptive sensors, front legs 70
A1.3 Femur proprioceptive sensors, front legs 70
A1.4 Thoraco proprioceptive sensors, middle legs 71
A1.5 Coxa proprioceptive sensors, middle legs 71
A1.6 Femur proprioceptive sensors, middle legs 71
A1.7 Thoraco proprioceptive sensors, hint legs 71
A1.8 Coxa proprioceptive sensors, hint legs 71
A1.9 Femur proprioceptive sensors, hint legs 72
A1.10 Tactile sensors, front legs 72
A1.11 Tactile sensors, middle legs 72
A1.12 Tactile sensors, hint legs 72
A1.13 Used features of coxa-trochanteral proprioceptive sensors for individual classes. 74
A1.14 Used features of femur-tibia proprioceptive sensors for individual classes. 74
A1.15 Influence power of single features on classes: thoraco-coxa sensors 75
A1.16 Influence power of single features on classes: femur-tibia sensors 75

A2.1 *KITTNN* package : Implemented neural network framework 76
A2.2 kitt_neuron.py : Neuron class inheritance 77
A2.3 kitt_net.py : Neural Network Initialization 78
A2.4 Terrain classification process - overall diagram. 79
A2.5 Software architecture for LPZRobots and GoRobots. (Martius et al., 2009) 80
A2.6 The process of data acquisition from the simulation. 81
A2.7 The structure of rough data directory. 81
A2.8 Workflow of generating a dataset 82

List of Tables

2.1 XOR problem definition 17

3.1 Summary of proprioceptive sensors of AMOS II hexapod robot 25
3.2 Initialization of *tripod_controller.h* (see appendix A4) 27
3.3 Terrain features and their ranges 29
3.4 Parameters of virtual terrain types 29

4.1 Comparison of f1-score on MNIST to another framework (*SKNN*) 43
4.2 PA progress example on MNIST 46
4.3 Comparison of the developed PA to other pruning methods (10 observations). Required accuracy on validation data: XOR: 0.99, MNIST: 0.85. 49
4.4 Classification results on generated datasets (see dataset parameters in Table A1.1). 50
4.5 Classification report for a deterministic dataset and a noisy dataset. 52
4.6 Comparison to other classification methods implemented by (Pedregosa et al., 2011) 52
4.7 Chosen configurations for PA demonstration. 57
4.8 Results of the PA on terrain datasets. 58

5.1 Studies of terrain classification for legged robots. 66

A1.1 Generated datasets 73

List of Abbreviations

AI	**A**rtificial **I**ntelligence
AMOS II	**A**dvanced **Mo**bility **S**ensor Driven-Walking Device - version **II**
ANN	**A**rtificial **N**eural **N**etwork
API	**A**pplication **P**rogramming **I**nterface
CPG	**C**entral **P**attern **G**enerator
GDA	**G**radient **D**escent **A**lgorithm
GUI	**G**raphical **U**ser **I**nterface
KITTNN	**KITT** **N**eural **N**etwork : the developed NN framework
k-NN	**k**-**N**earest **N**eighbours
NN	**N**eural **N**etwork
PA	**P**runing **A**lgorithm
RF	**R**andom **F**orest
SIFT	**S**cale **I**nvariant **F**eature **T**ransform
SKNN	**S**ci**K**it-**N**eural**N**etwork library
SURF	**S**peed **U**p **R**obust **F**eature
SVMs	**S**upport **V**ector **M**achines

Chapter 1

Introduction

The question of how information about the physical world is sensed motivated Frank Rosenblatt in 1958, when he presented his perceptron, a model capable of learning and pattern recognition.

His invention has been evolved over half a century in the field of machine learning, resulting in different kinds of artifical neural networks and deep learning methods. Out of the broad array of applications, we can mention the face recognition helping to find criminals, weather forecasting, searching engines or self-driving cars. Scientists from Cornell University in New York used deep learning to locate whales in the ocean so that ships could avoid hitting them (Kaggle, 2013).

In this thesis, I focus on classification, a widely used area of machine learning, using feedforward neural networks. The first part of the study is devoted to developing a working classification framework, based on a fully-connected network structure, which is commonly being used nowadays.

The novelty of the work consists of inventing a network pruning algorithm, which is able to find a minimal network structure needed for classification. The hypothesis is that some of the synapses are unimportant in the fully-connected network and hence the classification accuracy will not drop significantly after their removal.

In the second part of the study, the developed neural network framework is used for terrain classification for a hexapod robot AMOS II (Manoonpong, 2011). The classification is based on proprioceptive (joint angles) and tactile (ground contacts) sensing of the hexapod and the classification data is virtually generated using the *LpzRobots* simulator (Martius et al., 2009).

The hypothesis is that using proprioceptive and tactile sensors, sensory data, gathered in a period of time, will be needed for terrain classification, however, then, the prediction will be performed instantly, as no further data processing is required.

The invented pruning algorithm is used to find a minimal network structure for the terrain classification problem. Additionally, based on the minimal structure, the input vector features are analysed in terms of their contribution to classification.

Having the information about the terrain, the hexapod can adapt its gait accordingly to save some energy and/or to be even able to traverse the surface.

1.1 State of the Art

The idea of classification is based on training a model on a set of data by setting some model parameters. A wide range of classification methods, differing one from each other in their mathematical backgrounds, are provided nowadays, however, the classification procedure follows a conventional line:

1. model initialization;
2. learning process (using data A);
3. prediction (using never-seen data B).

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 set an output.

Basically, any kind of problem, ordinarily being solved by a human, can be transformed to a classification task for an artificial agent, if we define classes and find a numerical representation.

Overview of Classification Methods

The data samples usually consist of multiple features and the classes are linearly inseparable in most of the cases. The goal is to separate the classes in a high-dimensional space (illustrated in Fig. 1.1).

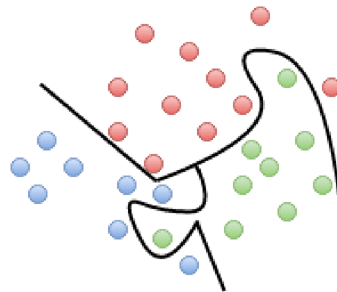


FIGURE 1.1: Illustration of a classification problem.

Support Vector Machines (*SVMs*); (Cortes and Vapnik, 1995) construct a set of hyperplanes in the high dimensional space to separate the classes. The position of the hyperplanes is based on so-called support vectors, which define distances to the nearest training-data points. Using the support vectors, a hyperplane with the largest margin between two classes is created and used for classification. The linear inseparability is solved by a kernel trick, which maps the original finite-dimensional space into a much higher-dimensional space.

The commonly known *k-Nearest Neighbours* (*k-NN*) algorithm uses a more straightforward way to classify samples into classes. Instead of finding the separation planes, this approach computes distances from the classified sample to k nearest samples of the training data. Then the class is determined by

the majority vote of those known samples. This approach skips the training process, however, the prediction is computationally expensive.

The *Random Forest* (*RF*) approach (Ho, 1995) is based on decision trees, which learn simple decision rules inferred from the data features. Decision trees are known for having troubles with over-fitting, therefore the Random Forest uses many decision trees and finds a classification result by averaging their outputs (bagging).

Each of those methods has some advantages and disadvantages on a particular type of data and its distribution. For instance, *SVMs* are a powerful tool for binary problems with an outlier free distribution. For a multiclass data, where many outliers are expected, one might want to choose Random Forest, however, the decision trees usually take a longer time in classification.

To summarize, the best classification method has not been proved yet. However, *neural networks* (*NN*) are being used in more and more fields nowadays.

Feedforward Neural Networks

Classification algorithms are often considered to behave intelligently while successfully accomplishing a particular task. To measure the intelligence of an artificial system, a comparison to the human behaviour is often used. If the goal is to model the human behaviour artificially, neural networks are certainly the most accurate imitation of a human brain out of the proposed methods.

The perceptron (Rosenblatt, 1958) is a binary classifier mapping an input vector $X = [x_1, x_2, ..., x_n]$ to an output $f_p(X)$; (Eq. (1.1)).

$$f_p(X) = \begin{cases} 1, & \text{if } W \cdot X + b > 0 \\ 0, & \text{otherwise} \end{cases} \tag{1.1}$$

where $W = [w_1, w_2, ..., w_n]^T$ is a vector of weights and b is a bias, which shifts the decision boundary away from the origin.

These parameters (W and b) are considered as perceptron's memory. Finding their optimal values, a single perceptron is capable of classifying linearly separable samples of two classes. This searching for the parameter values, based on some labeled data, is considered as learning.

To avoid the requirement of linear separability, multiple perceptrons are connected into a directed graph, which forms a multilayer perceptron. Then, replacing the $f_p()$ by another function, so-called transfer function (e.g. *sigmoid* or *tanh* - see Fig. 2.3), every unit generates a continuous output. The resulting structure is then called a *feedforward neural network* (see Fig. 2.1), which is generally capable of multiclass classification of linearly inseparable classes.

Network Pruning Algorithms

In general, increasing number of neurons in the network helps to deal with outliers and improves the classification. On the other hand, to obtain generalization in systems trained by examples, the smallest system that will fit the data should be used. Moreover, increasing number of synapses increases the dimensionality of weight matrices and so slows down the training process as well as the prediction. The aim of a network designer should be to find a minimal structure, where only connections important for classification remain, while the classification accuracy requirement is met. Two basic approaches of getting such structure are available:

1. train a network that is larger than necessary and then remove the parts that are not needed;
2. start from a small structure, then keep adding neurons and synapses until the network is capable of learning.

In (Reed, 1993), the author makes a good overview of proposed pruning algorithms. Pruning algorithms remove some of the synapses from a fully-connected network, which complies with our option 1. The research question is to distinguish synapses that are important for classification from those that are not used.

A brute force pruning, meaning removing the synapses one by one with an accuracy check after every iteration, results in $O(MW^2)$ time for each pruning pass, where W is the number of weights (synapses), M is the number of training patters and $O(W)$ is the forward propagation time. As this can be slow for larger networks, most of the pruning algorithms take less direct approaches, generally split into two groups:

1.a sensitivity calculation methods;

1.b penalty-term methods.

The first group (1.a) is based on estimating the sensitivity of the error function to removal of an element. In general, a network is trained, sensitivities are estimated and then weights and nodes are removed. The disadvantage of this approach is that correlated elements are not detected. This means that after removal of one synapse, weights of the remaining synapses might not be valid for the smaller network.

The penalty-term methods (1.b) reward the network for choosing an efficient solution by adding terms to the objective function. For instance, weights close to zero are not likely to influence the output much and so can be eliminated. Hence the cost function is modified so that backpropagation drives unnecessary weights to zero and, in effect, removes them during training. In this manner, the training and pruning are effectively done in parallel.

A performance evaluation of the brute force algorithm and an algorithm presuming that zero weights do not influence the output, is presented in section 4.2.3 of this study.

Moreover, having the minimal network structure, features important for classification can be effectively selected and analysed (see section 2.4.4).

Terrain Classification for Legged Robots

Multi-legged autonomous robots have become popular for their ability to deal with various terrain types, which might be impassable for wheeled robots. Terrain classification helps to adapt their gait and so to optimize the walking performance. In general, legged robots are equipped by a broad range of sensors. Several studies have been done on the terrain classification topic, where each of them is based on a specific sensor type:

Starting with the *vision-based classification* methods, in (Zenker et al., 2013), the authors present an online terrain classification system based on a monocular camera. The classification algorithm is based on extracting features from images using either *SIFT* (Lowe, 2004) or *SURF* (Bay, Tuytelaars, and Gool, 2006) and the classification is performed by *SVMs*. The performance is evaluated on 8 terrain types with the accuracy of 90%. This approach is currently used on the hexapod robot AMOS II, which is also the target platform of this study.

In spite of the fact that the Matilda platform in (Mou and Kleiner, 2010) uses belts, not legs, the topic is similar. Vision is used in combination with laser and vibration readings to classify terrain for online adapting robot velocities. The final classification result is provided as a combination of single classifiers. The final classifier is robust towards changing illumination and able to recognize 5 different terrains with an accuracy rate close to 100%.

Regarding a *classification based on laser sensors*, the laser range finder in (Kesper et al., 2012) provides some information about terrain roughness. In this case, it is not a terrain what is actually classified, but just a roughness factor is computed and a proper gait with corresponding behaviour (also on the AMOS II platform) is selected, based on the roughness estimation.

In (Hoepflinger et al., 2010), the author writes about *classification based on tactile (haptic) sensing*. A force sensing device was integrated in a robotic leg to obtain haptic feedback from a prescribed knee joint actuation. The results of a multiclass AdaBoost classifier showed that tactile sensors are capable of recognizing ground shapes, however, the algorithm performed slightly worse when classifying different surface types.

The author of (Coyle, 2010) devoted his thesis to proprioceptive sensors of a vehicle. In (Ordonez et al., 2013) using internal vibration data is also considered as proprioception sensing. In (Bermudez et al., 2012), the author uses vibration data from the on-board inertial measurement unit to classify three types of rough terrain for a legged robot and reaches an accuracy over 90%.

In (Xiong, Worgotter, and Manoonpong, 2014), the authors use observations of the motor signals, generated by the controller, to classify six surfaces with a high accuracy.

Based on the related study, a combination of more sensor types seems to be the best choice regarding the classification accuracy. However, some requirements regarding the target platform, initial purpose of classifying the terrain or another conditions. For instance, vision-based sensors can hardly be used at night. Moreover, processing of data from more sensors

might exceed time limitations for classification. An interesting observation from the literature is that terrain classification results are mostly successful, however, in most of the studies, only a little number (3, 6, maximally 8 in (Zenker et al., 2013)) kinds of terrains are used for classification.

1.2 Master Thesis Objectives

The main objectives of this study are:

1. To implement a general classification framework using feedforward neural networks;
2. To develop a new network pruning algorithm capable of finding the minimal network structure for a given dataset;
3. To generate a dataset of virtual terrain types using the AMOS II simulation;
4. To classify the virtually generated terrain types using proprioceptive and tactile sensing.

The implemented classification method will be capable of learning and classification of commonly known datasets as well as the terrain dataset. It will also be compatible with the needs of the new pruning algorithm.

The pruning algorithm will be evaluated on commonly known datasets to check the functionality. Then, it will be applied on the terrain classification problem.

The terrain classification will be evaluated on the simulation data. As a reference, deterministic data will be classified. Then a Gaussian signal noise will be added to simulate a real world environment and the classification performance on the noisy data will be compared to the reference.

1.3 Relation to the State of the Art

The target platform, hexapod robot AMOS II (see section 3.2.1), is equipped with a wide range of sensor types. However, in this study, we use only proprioceptive and tactile sensing for the terrain classification and in the following lines the reasons for this choice are listed:

1. *Biological inspiration.* Based on (Bräunig and Hustert, 1980), in insects, the cell bodies of the sensory neurons are located in the periphery, close to the site of stimulus reception. It is presumed that these neurons help the insect to sense the position of its legs and so to perceive the shape of the surface. Proprioceptive sensors simulated as joint angles exploit the same idea.

   Tactile sensors are considered as a channel of communication for many insects, as they have poor vision and sound perception (Meyer, 2006).

2. *Insensitivity to light conditions.* Vision-based classification methods are proven to be accurate and very powerful, but those fail completely

in dark and must deal with illumination changes. Proprioceptive and tactile sensing works in any light conditions.

3. *Proprioception sensing is general.* Although the evaluation is performed on a hexapod robot in this study, proprioception sensing results can be applied on any kind of robot with joints. In the future, possibly, for two-legged walking humanoids.

4. *Fast processing.* The sensory data evaluation is presumed to be incomparably faster to the vision method, as no further processing, but a direct classification, is performed on the data.

Secondly, a feedforward neural network was chosen as a classification method. Besides a biological inspiration, neural networks were chosen as the most mysterious approach, which still contains many research questions to be answered.

One of them relates to searching and utilization of network minimal structures. Using a powerful pruning algorithm may contribute to optimization of the network in terms of size and prediction time. As we want to classify the terrain online on a real platform, a minimal model fitting the data will be a great benefit. Using neural networks, the minimal model might be later combined with the central neural controller (Manoonpong, 2011), which drives the target platform, AMOS II, and manages its behaviour.

1.4 Thesis Outline

The thesis consists of 6 chapters in total. The chapter 2 is devoted to the developed classification framework. In section 2.3, the learning algorithm is described. Then, section 2.4 introduces the new network pruning algorithm.

Chapter 3 contains the process of terrain classification. Firstly, in section 3.2, the experimental environment is specified. Next, section 3.3 shows, how the virtual terrains were created. Sections 3.4 and 3.5 are devoted to the data acquisition and forming a feature vector. Generation of terrain datasets is shown in section 3.6. Finally, the training and classification process is described in section 3.7.

Results are presented in chapter 4. First of all, the classification framework is verified on well-known datasets in section 4.1. Then the performance of the pruning algorithm is evaluated in section 4.2. Results of the terrain classification are put in section 4.3. Finally, the pruning algorithm is applied on the networks trained on terrain datasets and the results are shown in section 4.4.

The methods are recapitulated and the results compared in discussion (chapter 5). The study is concluded and an outlook is provided in chapter 6.

Appendix A1 contains some supplementary data and results. In appendix A2, one can find implementation details of the performed methods, while a complete code documentation is in appendix A4. Appendix A3 shows a tree directory structure of the workspace.

Chapter 2

Classification Method

Principles of feedforward neural networks, based on the perceptron idea (see Eq. (1.1)), are used for developing a 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 framework is called *KITT Neural Network* (*KITTNN*). Implementation details are provided in appendix A2.1.

2.1 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. 2.1).

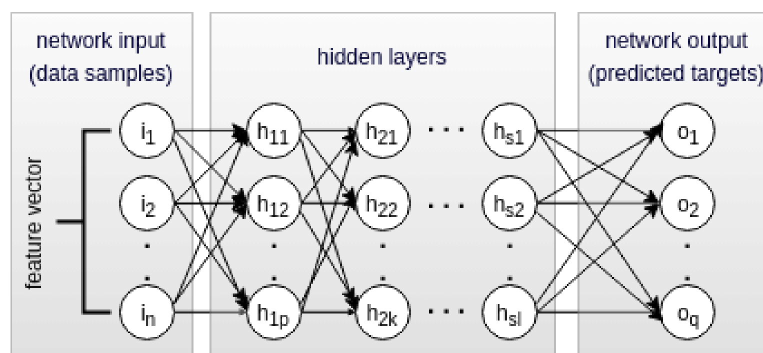


Figure 2.1: Structure of a feedforward neural network

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.

2.2 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. 2.2. The diagram complies with the following notation:

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

$f()$: transfer function (Eq. (2.2))

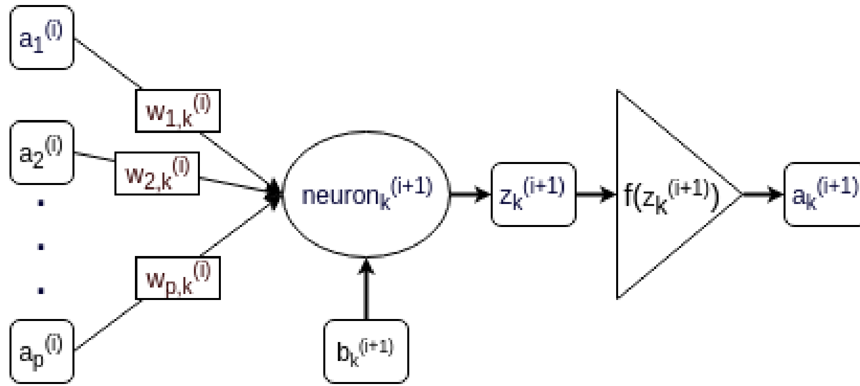


FIGURE 2.2: A model neuron

Assuming p being the number of neurons in i^{th} layer, the activation of $neuron_k^{(i+1)}$ is computed as in 2.1

$$neuron_k^{(i+1)} = \sum_{l=1}^{p} a_l^{(i)} \cdot w_{l,k}^{(i)} + b_k^{(i+1)} \tag{2.1}$$

The sigmoid function given as in Eq. (2.2) is used as the transfer function for computing the activities of individual neurons.

$$f(z) = \frac{1}{1 + e^{-z}} \tag{2.2}$$

The sigmoid function maps neuron activations into $[0.0, 1.0]$ interval. This approach is used by default in this work. Additionally, the *tanh(z)* function is implemented (compared to *sigmoid(z)* in Fig. 2.3), in order to test one of the

pruning methods based on weights sensitivity (evaluated in section 4.2.3). The *tanh(z)* function maps the input z into $[-1.0, 1.0]$ interval.

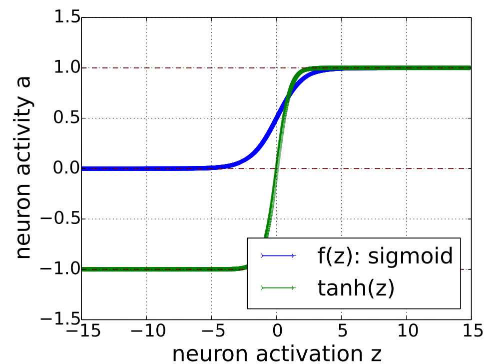


FIGURE 2.3: Used transfer functions: sigmoid and tanh

2.3 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.

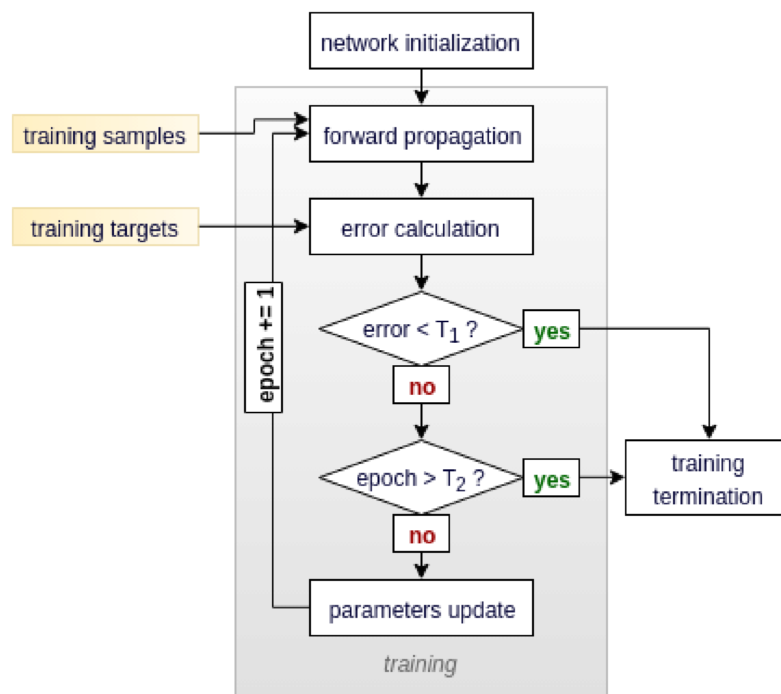


FIGURE 2.4: Training process flowchart. T_1: Threshold for a terminating condition based on a learning error. If the error is reduced to be lower than this threshold, the learning process is stopped. T_2: Threshold for a terminating condition based on the number of iterations (epochs). The learning process 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*.

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 *KITNN* framework (appendix A2.1) and adjusted to the needs of the developed pruning algorithm (section 2.4). The learning process is summarized by the flowchart in Fig. 2.4. The procedure follows algorithmical steps found in (Labs, 2014).

2.3.1 Using Mini-batches

The training procedure of multilayer perceptrons (see Eq. (1.1)) 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. (2.8)). Usually, the mini-batch size is left as a training parameter. In this work, it is set to *10*.

2.3.2 Matrix Notations

Assuming feedforward neural networks and referring to Fig. 2.2, the following notation is used for processing multiple samples.

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 ith layer

2.3.3 Forward Propagation

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

$$Z^{(2)} = X \cdot W^{(1)} + B^{(2)} \tag{2.3}$$

$$Z^{i+1} = A^{i} \cdot W^{i} + B^{(i+1)} \tag{2.4}$$

$$A^{(i)} = f(Z^{(i)}) \tag{2.5}$$

$$\hat{y} = f(Z^{(j)}) \tag{2.6}$$

2.3.4 Error Calculation

To get an idea about how wrong the predictions of the network 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.

$$J = \sum_{k=1}^{m} \frac{1}{2}(y_k - \hat{y}_k)^2 \tag{2.7}$$

2.3.5 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. (2.7)) 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. (2.8)). 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.

$$\frac{dJ}{dW^{(i)}} = \sum_{k=1}^{m} (\frac{dJ}{dw})_k \tag{2.8}$$

Having several layers of a network results in several weight (and bias) matrices, while the goal is to find optimal parameters of overall network. In order to compute optimal parameters in all of the matrices, the sum rule in differentiation (Eq. (2.9)) followed by the chain rule (Eq. (2.10)) are applied.

$$\frac{\delta}{\delta x}(u + v) = \frac{\delta u}{\delta x} + \frac{\delta v}{\delta x} \tag{2.9}$$

$$(f \circ g)' = (f' \circ g) \cdot g' \tag{2.10}$$

Due to these properties, the error obtained at the network output (section 2.3.4) is propagated backwards layer by layer through the network (Eq. (2.12)) and derivatives $\frac{dJ}{dW^{(i)}}$ for all layers i are found (Eq. (2.13)). For this procedure, the derivative of our transfer function is needed (Eq. (2.11)).

$$f'(z) = f(z) \cdot (1 - f(z)) \tag{2.11}$$

$$\delta^{(i)} = \delta^{(i+1)} \cdot (W^{(i)})^T \cdot f'(Z^{(i)}) \tag{2.12}$$

$$\frac{dJ}{dW^{(i)}} = (a^{(2)})^T \cdot \delta^{(i+1)} \qquad \frac{dJ}{dB^{(i)}} = \delta^{(i+1)} \tag{2.13}$$

The parameters are then updated for the next iteration as shown in Eq. (2.14).

$$W^{(i)}_{(t+1)} = W^{(i)}_{(t)} + \frac{dJ}{dW^{(i)}}_{(t)} \qquad B^{(i)}_{(t+1)} = B^{(i)}_{(t)} + \frac{dJ}{dB^{(i)}}_{(t)} \tag{2.14}$$

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 2.4 and is fully compatible with the *KITNN* framework for any type of data.

2.4 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 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 the generated connections are significant and necessary for classification.

2.4.1 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. 2.5.

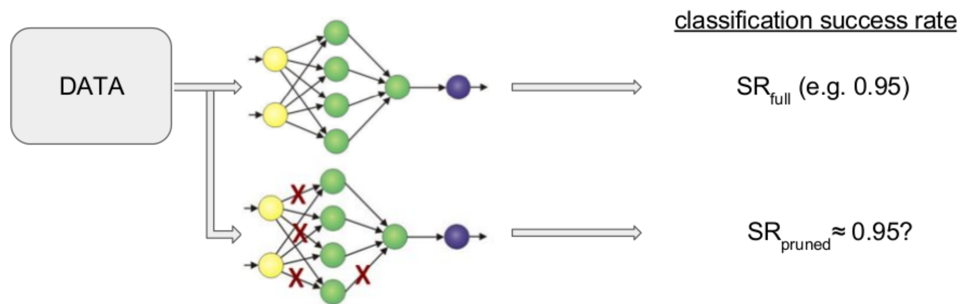


FIGURE 2.5: Pruning Algorithm: hypothesis formulation

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 synapses 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 presumed that a synapse, whose weight does not evolve while the network is trained, does not contribute to classification much.

2.4.2 Algorithm Realization

The pruning algorithm itself is an iterative process, however, as shown in Fig. 2.6, two important steps are done in advance.

FIGURE 2.6: Overall flowchart of the pruning process

First of all, the following variables are initialized:

net : a fully-connected *KITTNN Network()*; (see appendix A4.1) 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. 2.7.

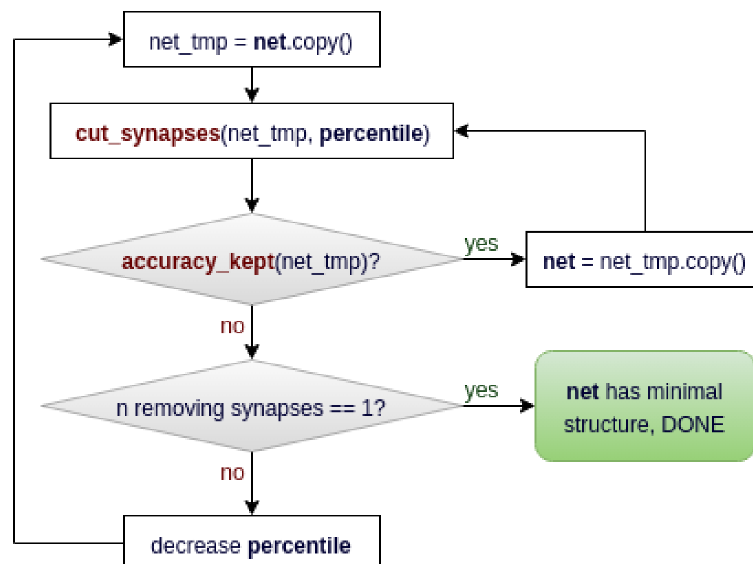


FIGURE 2.7: The Pruning Algorithm: initialized variables are in bold, red marked functions refer to 2.8 and 2.9 respectively

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. 2.8.

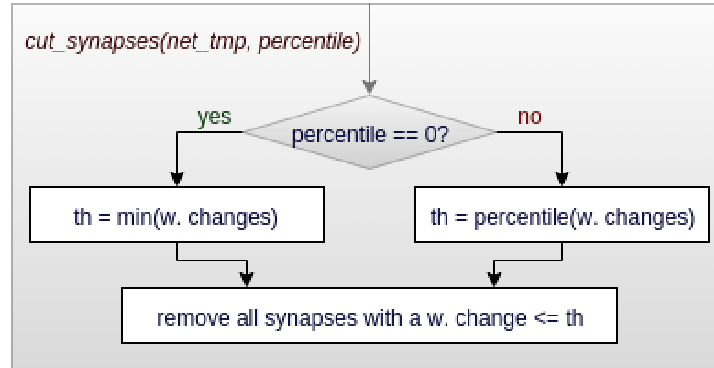


Figure 2.8: Synaptic pruning based on weight changes and current percentile value

As discussed above, an initial weight value was saved for each synapse before the training. Hence, for n being the total number of synapses, weight changes Δw_i are known (2.15).

$$\Delta w_i = |w_init_i - w_current_i|, \quad i = 1, ..., n \tag{2.15}$$

Based on these changes and on the current *percentile* value, a threshold (*th* in Fig. 2.8) 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 so 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.

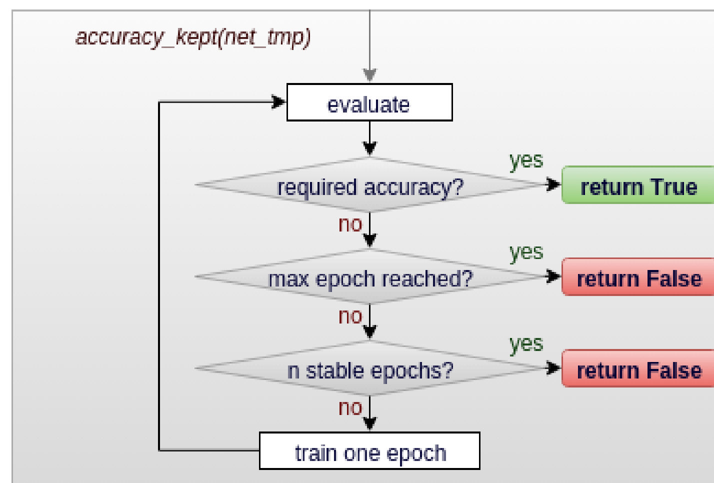


Figure 2.9: Evaluation of the classification accuracy after pruning

After cutting some synapses, the network is checked for keeping the required classification accuracy as shown in Fig. 2.9.

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. The method is evaluated in detail in section 4.2 and compared to two different approaches from (Reed, 1993) in section 4.2.3.

2.4.3 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 sections 4.2.1 and 4.2.2.

The pruning algorithm was also applied on the terrain classification problem (evaluated in section 4.4).

XOR Dataset

This dataset relates to the well-known XOR problem defined by a truth table (Table 2.1).

TABLE 2.1: XOR problem definition

$\mathbf{x_1}$	$\mathbf{x_2}$	$\mathbf{y}$
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 2.5). The composition of the two classes in a 2D space is shown in Fig. 2.10.

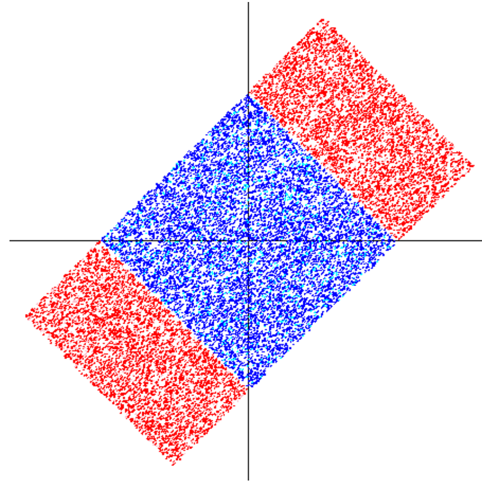
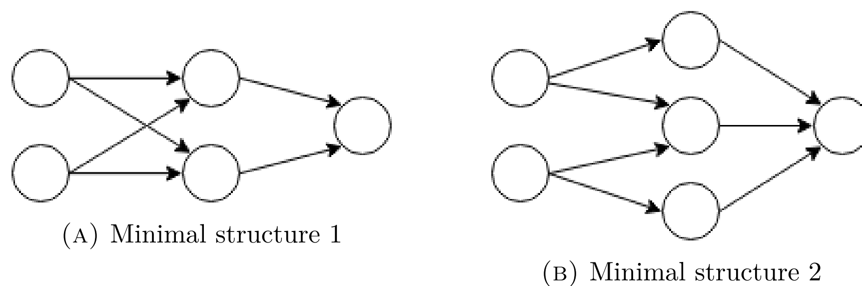


FIGURE 2.10: 2D XOR Data illustration

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. 2.11), both considered as minimal. Geometrically, the version shown in Fig. 2.11a creates the two lines in 2D space to separate the classes, while the second one (Fig. 2.11b) transfers the problem from 2D space into 3D space and creates a separation plane.



(A) Minimal structure 1

(B) Minimal structure 2

FIGURE 2.11: XOR Dataset: minimal network structures

The goal of the pruning algorithm is to end up with a network of the structure in Fig. 2.11a, when the network is initially oversized. The evaluation is presented in section 4.2.1.

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. 2.12.

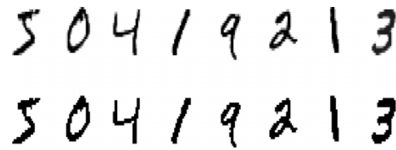


FIGURE 2.12: MNIST Data illustration (LeCun and Cortes, 1998)

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. However, detailed results of classification accuracy for various classifiers are provided, which is useful for comparison.

2.4.4 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 a single neuron/synapse has 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 the feature selection of input vectors and the selected output class (demonstrated on a MNIST example in Fig. 2.13). Evaluation on MNIST dataset is shown in section 4.2.2.

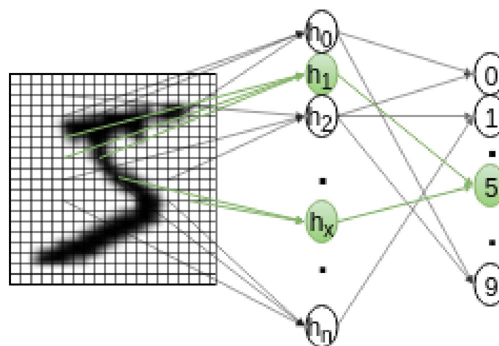


FIGURE 2.13: Analysis of the minimal structure examplified on digit 5 (MNIST dataset)

2.5 Graphical User Interface

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

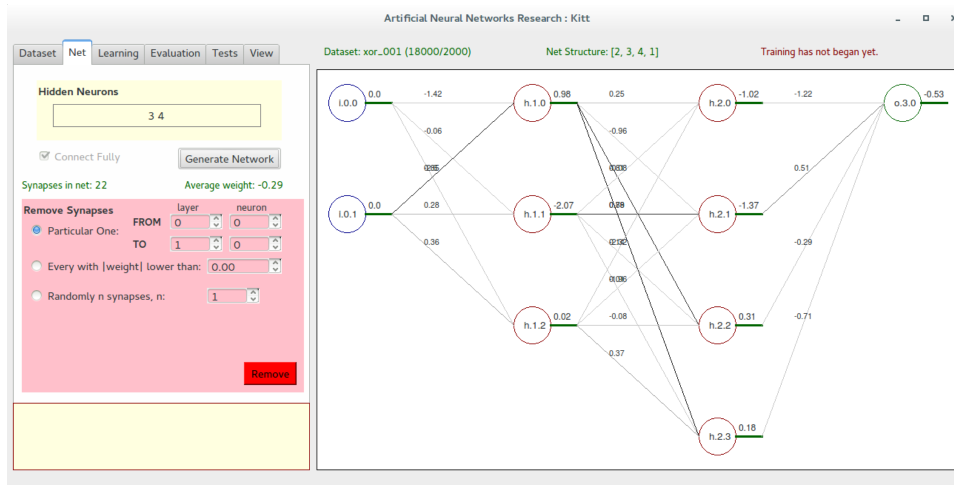


FIGURE 2.14: Screenshot of the graphical user interface

This GUI is capable of:

1. Loading a dataset in a specific form and, if possible, visualizing it (see XOR data in Fig. 2.10 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. 2.14).
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 3

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. 3.2). 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 2, feedforward neural networks are used for classification.

3.1 Overall Process Summary

The overall process consists of several modules. The workflow is illustrated in Fig. 3.1 (a more detailed diagram can be found in Fig. A2.4).

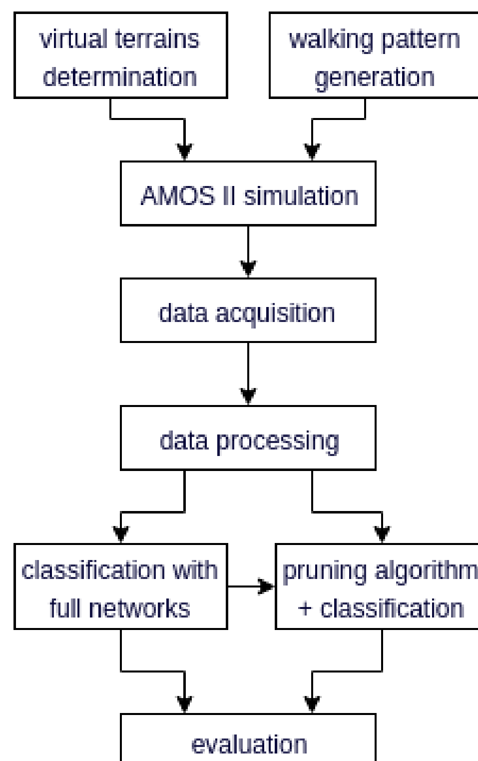


FIGURE 3.1: Terrain Classification: overall process diagram

The very first step is to make the AMOS II simulation run (appendix A2.2). Then a simple tripod gait controller is implemented (section 3.2.3). To generate various terrain types, the number of variable terrain features and their ranges are determined (section 3.3.1). Based on these features (parameters), a number of virtual terrains is defined (section 3.3.2) and an optimality of these parameters is briefly analysed.

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 sensors 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.

Next, a dataset is created by splitting all the data into training, validation and testing sets. As it is indicated in Fig. A2.4, several datasets are generated and several networks trained during the process.

The dataset packages may differ in these parameters:

1. standard deviation of terrain noise;
2. standard deviation of signal noise;
3. number of simulation timesteps;
4. used sensors.

The number of samples is set to 500 and all terrain types are used for classification (resulting in 14 classes, see Table 3.4). An analysis of the dataset parameters is presented in section 4.3.

The trained networks may differ in the following parameters:

1. dataset the network has been trained on;
2. neural network hidden structure;
3. learning rate;
4. number of training epochs.

The parameters of the learning process are evaluated in section 4.3.2.

A dataset with no additive noise is chosen and the classification performance of a neural network trained on this deterministic dataset is considered as a reference. Then, another network is trained on a noisy dataset, to simulate the conditions of real environment, and the classification results are compared to the reference in section 4.3.1.

Finally, a trained network is pruned by the algorithm introduced in section 2.4. Using the network minimal structure, an analysis of feature selection for terrain classification is performed in section 4.4.1.

3.2 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 3.2.1.

However, as already stated above, the proposed approach will be evaluated using simulated robot in a virtual environment. In this case, *LPZ Robots simulator* (Martius et al., 2009) was used and a description is given in section 6.2.2.

3.2.1 Hexapod Robot AMOS II

The *AMOS II* abbreviation stands for Advanced Mobility Sensor Driven-Walking Device - version II (Manoonpong, 2011). It is a biologically inspired hardware platform of size 30x40x20 cm and weight 5.8 Kg (see Fig. 3.2). 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.

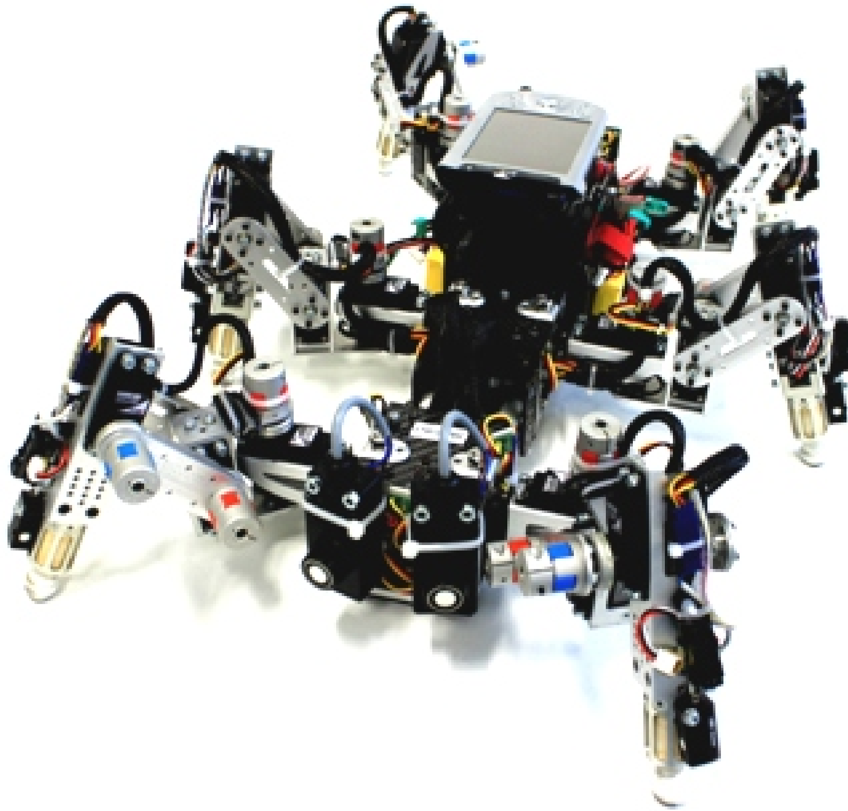


FIGURE 3.2: AMOS II. (Manoonpong, 2011)

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. (Manoonpong, 2011). 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. 3.3.

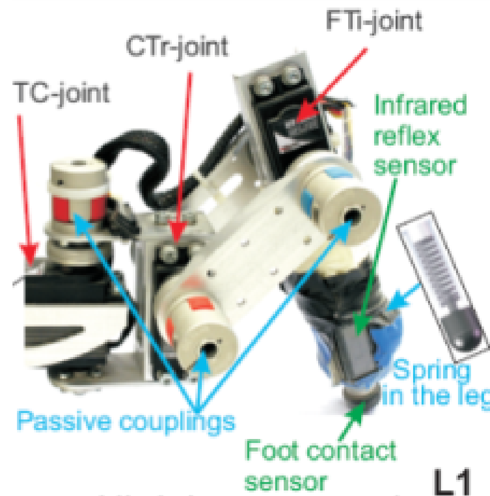


Figure 3.3: Structure of the AMOS's leg. (Manoonpong, 2011)

As shown in Fig. 3.2 and Fig. 3.3, 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 proprioceptive 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 A2.2), it is omitted in this work.

In Table 3.1 all the proprioceptive 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 3.5 a normalization of these ranges is discussed.

Robot actuators (servo motors) can generate movements of variable compliance by utilizing a virtual muscle model (see Manoonpong, 2011 for details).

Table 3.1: 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 3.2.3.

3.2.2 AMOS II Simulation

The *lpzrobots* project, developed by a research group at the University of Leipzig (Martius et al., 2009) under GPL license, is used for AMOS II virtual representation. Some implementation details are discussed in appendix A2.2. The project modules important for this study are shown in Fig. 3.4.

The simulation environment is set up with these initial parameters:

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

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

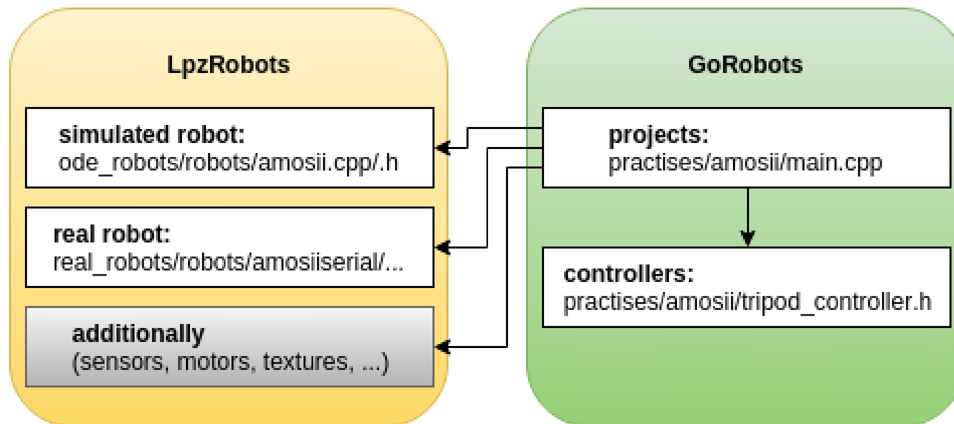


FIGURE 3.4: Structure of the two repositories: LPZRobots and GoRobots. (Martius et al., 2009)

The initial robot position in the map is chosen randomly in order to generate a different route every time the simulation is launched. The required terrain to be simulated is passed to the simulation as an argument. Additionally, the standard deviation value of Gaussian terrain noise (details in section 3.3.3) is set as another argument.

The simulation is made 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 visualization of AMOS II is illustrated in Fig. 3.5.

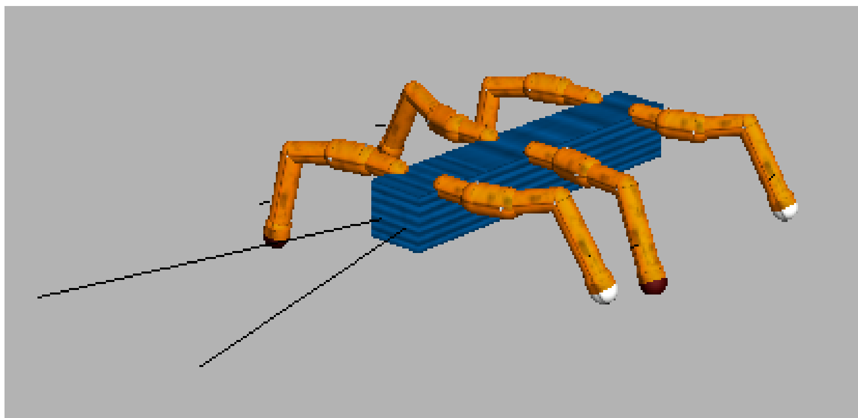


FIGURE 3.5: Virtual alternative for AMOS II.

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

3.2.3 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 legs 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. 3.6.

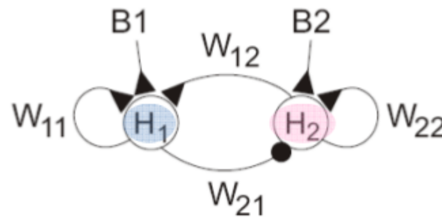


FIGURE 3.6: 2-neuron network oscillator ("Adaptive Embodied Locomotion Control Systems")

The initial conditions and parameters of the implemented controller are shown in Table 3.2.

TABLE 3.2: Initialization of *tripod_controller.h* (see appendix A4)

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**

$$a_i(t+1) = \sum_{j=1}^{n} w_{ij} o_j(t) + b_i, i = 1, ..., n \tag{3.1}$$

2. **The transfer function application**

$$f(a_i) = tanh(a_i) = \frac{2}{1 + e^{-2a_i}} - 1 \tag{3.2}$$

3. **Joint settings**

   With the reference to previous equations and variables names, the actuators are set as shown in Fig. 3.7. The *femur* joints (red ones) stay unchanged (set to zero). This setting generates a tripod gait for AMOS II.

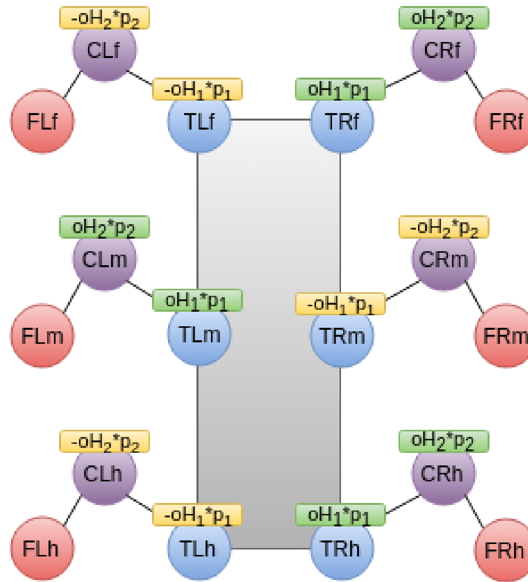


FIGURE 3.7: Schematic diagram of tripod gait controller

3.3 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 A2.2).

Besides these four parameters represented as a substance handle, a terrain constructor takes six more arguments (used in code part A2.2):

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

3.3.1 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 3.3).

TABLE 3.3: 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

3.3.2 Features Determination for Various Terrains

To determine a terrain type, one has to come up with the five parameters from Table 3.3.

In this work we use 14 terrain types. Their parameters (shown in Table 3.4) have been set up manually. With respect to the feature ranges from Table 3.3, the values have been normalized between 0 and 1.

TABLE 3.4: 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 the following section.

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

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. (3.3). The five qualities are listed in Table 3.3 and in Table 3.4.

$$SM_{t_1,t_2} = \frac{\sum_{i=1}^{5} |quality(i,t_1) - quality(i,t_2)|}{5} \tag{3.3}$$

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. 3.8 shows the similarity measures among generated terrains.

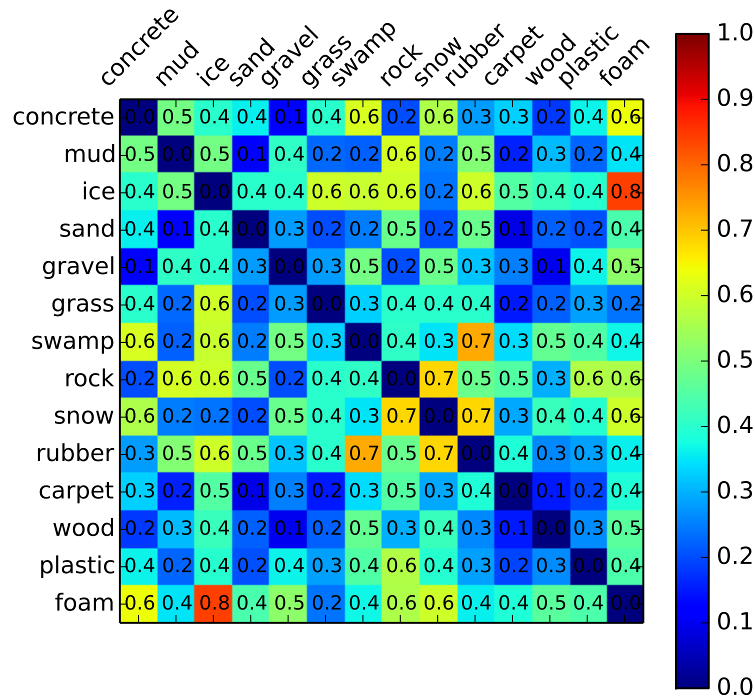


FIGURE 3.8: Similarity measures among various terrain types.

The figure demonstrates 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. Rubber seems to be very different from swamp or snow, which is also a positive outcome. Also the high grass-ice or rock-snow similarity measures make sense.

3.3.3 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 3.3). The parameters in Table 3.4 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 referred features.

Consequently, the terrain parameters shown in Table 3.4 are noised. Regarding individual features and their upper limits from Table 3.3, the following Eq. (3.4) shows, how the noise is added.

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

$$feature_noise \sim N(\mu, \sigma^2)$$

$$feature_noise = fRand(0, feature_{up_limit} * std_p) \tag{3.4}$$

$\boldsymbol{std_p}$: 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 3.3). 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 3.4) as given in Eq. (3.5).

$$feature_value += feature_noise \tag{3.5}$$

Additionally, there is a limit checking as the parameters cannot take negative values. The final form is set as shown in Eq. (3.6).

$$feature_value = max(feature_value, 0) \tag{3.6}$$

Influence of Terrain Noise

Based on the explanation of adding the terrain noise, single samples representing various levels 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. 3.9.

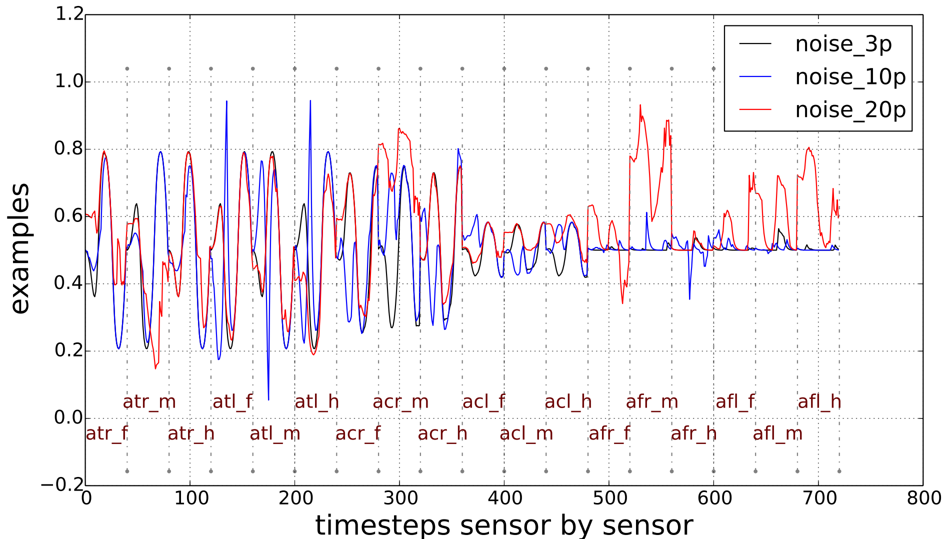


FIGURE 3.9: Examples of noisy terrains: terrain rock, angle sensors

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. 3.9, the sample representing the $20\% - noise$ class fluctuates more in comparison to the others, especially for the femur-tibia sensors. This might indicate a generation of an unusual rocky terrain.

3.4 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 3.5), which is one of the most essential parts of the process.

As it is later described in more detail, several sensor values in time need to be used to obtain the robot's dynamics on various terrains. Therefore, to generate a single sample candidate, the simulation must be run for a period of time. We use the 'sample candidate' term for data obtained from one simulation run. 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 was set 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 A2.2).

As shown in Fig. 3.3, the robot has 3 proprioceptive and 1 tactile sensor on each leg. In the following figures (Fig. 3.10 - Fig. 3.13), examples for each of these sensor types are shown for all referred terrains.

In Fig. 3.10, 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) can be seen.

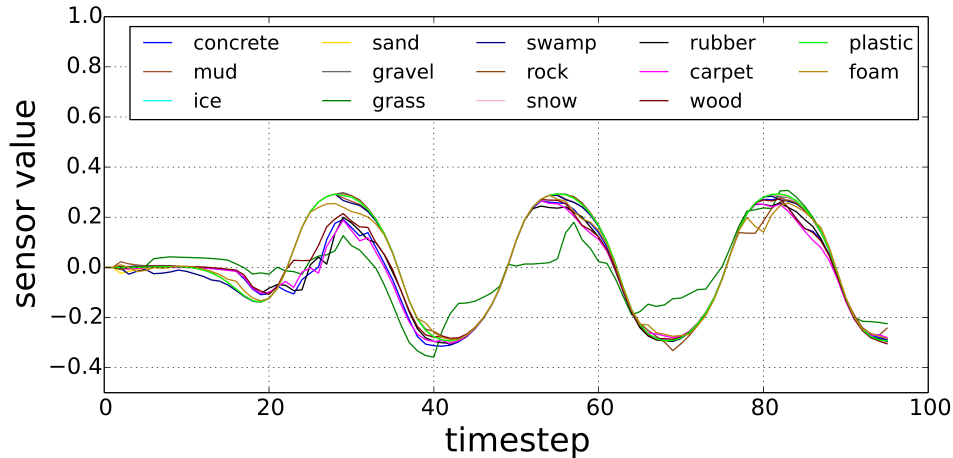


FIGURE 3.10: Thoraco Sensor (ATRf) output examples, 14 terrains

Fig. 3.11 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.

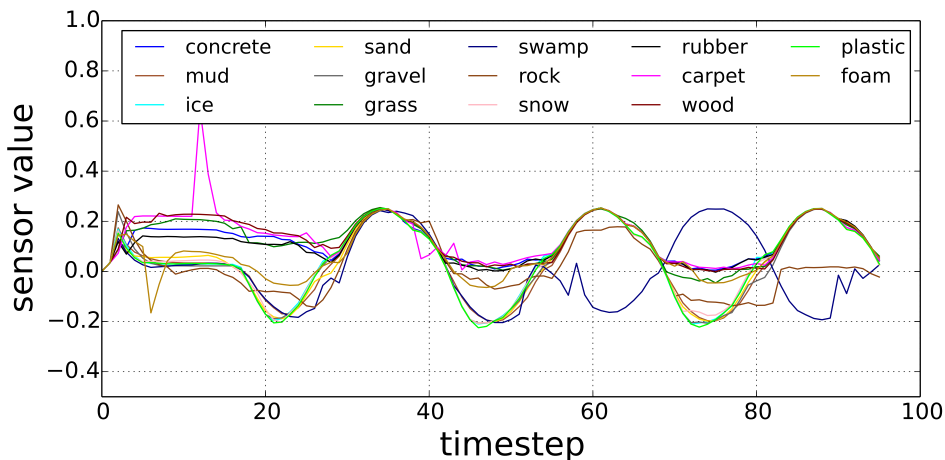


FIGURE 3.11: Coxa Sensor (ACRm) output examples, 14 terrains

The *Femur-Tibia* sensor outputs, for the joint sensor on the right hint leg, are illustrated in 3.12. 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 3.2.3). The signal fluctuations must be caused by passive movements of the *Femur-Tibia* joint, but still might be essential for classification.

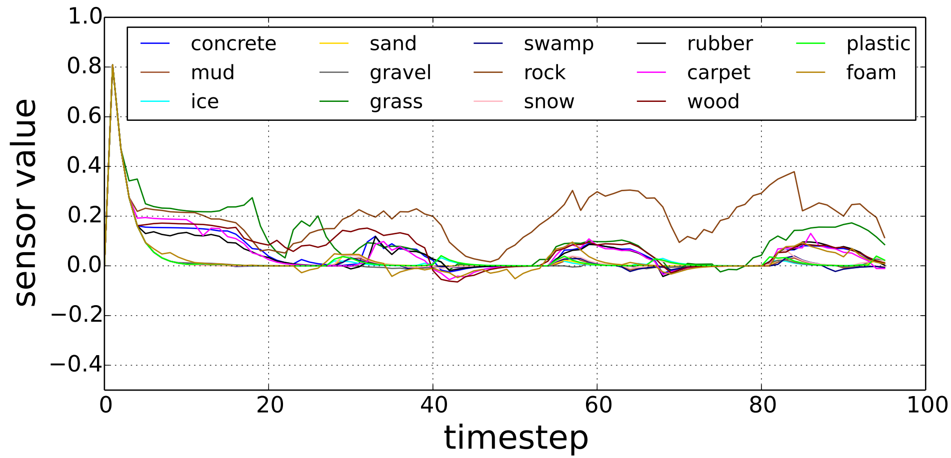


FIGURE 3.12: Femur Sensor (AFRh) output examples, 14 terrains

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. 3.13). If the value of the foot contact sensor is 1, the robot stays on the foot with its full weight, 0 indicates a leg in the air.

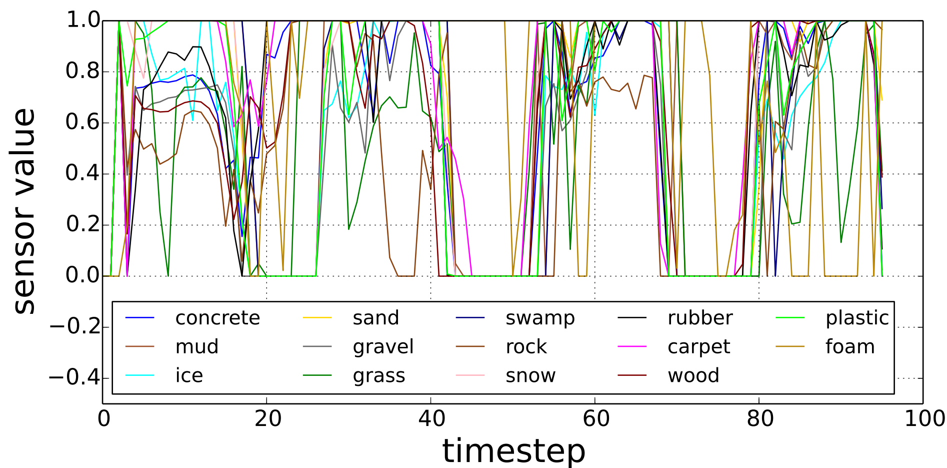


FIGURE 3.13: Foot Contact Sensor (FRf) output examples, 14 terrains

Examples of signals for all 24 sensors can be found in appendix A1.1.

As an optimal standard deviation value of the additive terrain noise is not known, data for several values of this parameter were generated. The simulation was 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. (3.4). The influence of additive terrain noise is analysed in section 3.3.3.

The approach of storing the gathered data is described in appendix A2.2. As Fig. A2.7 shows, 500 sample candidates are generated for every *noise/terrain* configuration. This allows creating datasets of 500 samples per class.

3.5 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 3.1).

For this particular problem, the task is to form one feature vector out of the content of one stored data file (see appendix A2.2), as each of these files contains data for one sample (see Fig. 3.14).

FIGURE 3.14: 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. 3.14, 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. 3.15 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 3.1) are added to the x-axis annotation. The 18 angle sensors are followed by the 6 foot contact sensors.

3.5.1 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:

$$S'_i = \frac{S_i - r_{min}}{r_{max} - r_{min}} \tag{3.7}$$

r_{min}, r_{max} : bounds of the corresponding original sensor range (listed in Table 3.1)

S'_i : scaled element of the normalised signal

Also a $[0, 1]$ interval overflow checking is added and values are adjusted if needed (Eq. (3.8)). This is a cover for the case when ranges from Table 3.1 were not accurate.

$$S'_i = min(max(S'_i, 0), 1) \tag{3.8}$$

The following figure (3.15) shows normalised feature vector examples for three terrains. The influence of normalisation on classification results is another subject for the discussion.

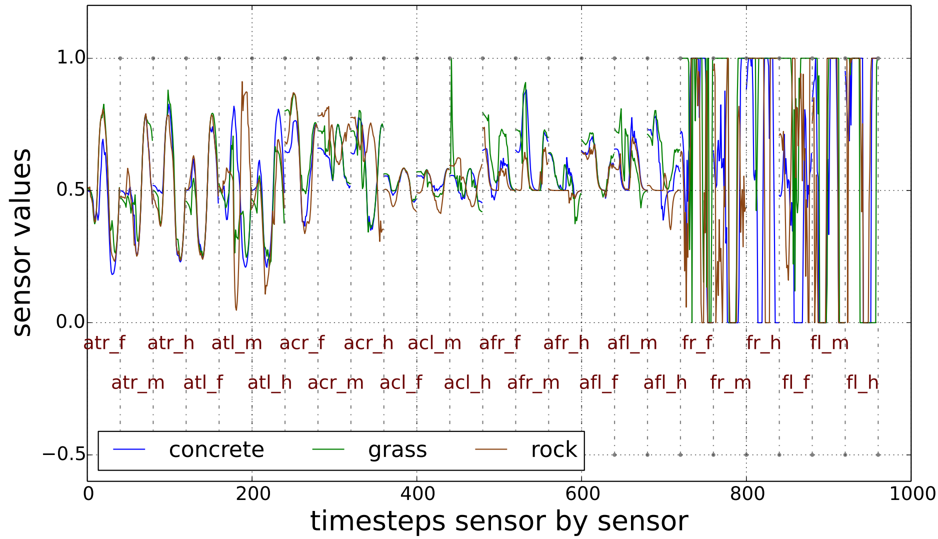


FIGURE 3.15: Normalised feature vector examples

3.5.2 Signal Noise

In section 3.3.3 a few general reasons for adding a noise to simulation data were discussed. In that case an additive Gaussian noise is used to generate variability in the data and to make the terrain types definitions (from Table 3.1) 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 3.3.3, 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.

$$signal_noise = [sn_1, sn_2, ..., sn_n] \tag{3.9}$$

$$sn_i \sim N(\mu, \sigma^2), \quad i = 1, 2, ..., n \tag{3.10}$$

Then, the generated vector is added to a normalised feature vector from section 3.5.1 (Eq. (3.11)).

$$noised_signal_i = raw_signal_i + sn_i, \quad i = 1, 2, ..., n \tag{3.11}$$

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

$$noised_signal_i = min(max(noised_signal_i, 0), 1), \quad i = 1, 2, ..., n \quad (3.12)$$

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. 3.16 shows the influence of the signal noise on one sample of *concrete* for joint angle sensors.

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

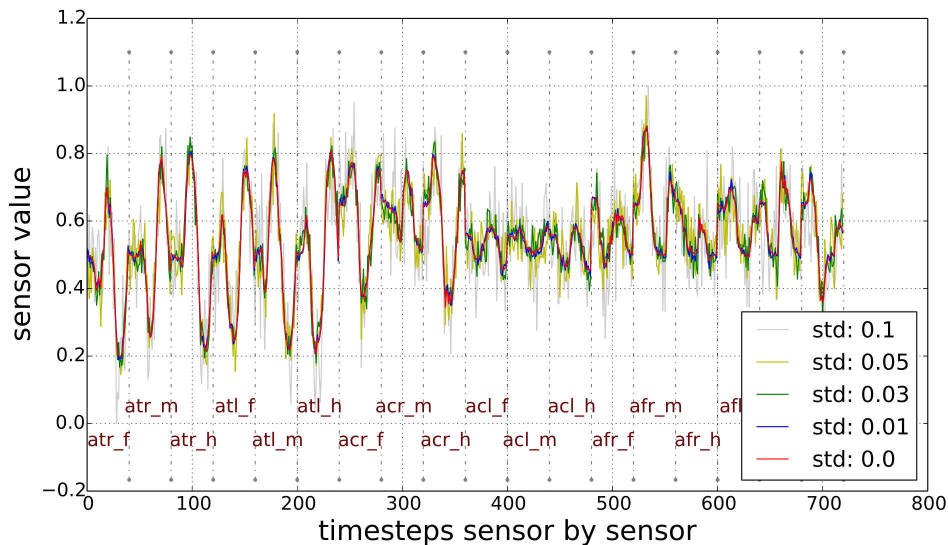


Figure 3.16: Examples of noisy signals: concrete, angle sensors

3.6 Generation 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.

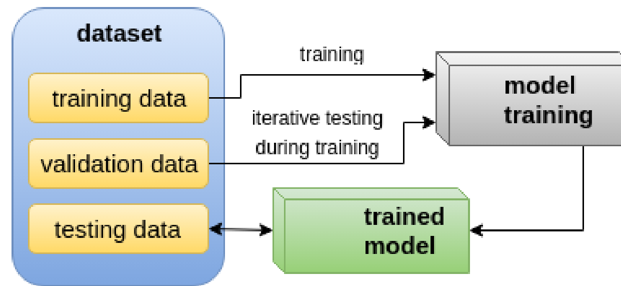


FIGURE 3.17: Three sets of data in a dataset.

Each set of data consists of samples and targets (class labels). The samples are represented by normalised feature vectors (section 3.5) - 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 3.4) 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 3.1) 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. 3.18.

terrain types	target vector
carpet	0
concete	**1**
foam	0
grass	0
gravel	0
ice	0
mud	0
plastic	0
rock	0
rubber	0
sand	0
snow	0
swamp	0
wood	0

FIGURE 3.18: 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. 3.17. 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.

The workflow of data generation procedure is illustrated in Fig. A2.8.

3.7 Training and Classification

Having a dataset enables to train a classifier, a machine learning tool that is able to learn some behaviour 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. 3.17.

There are many classification methods differing in mathematical backgrounds and each of them has some advantages 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. Additionally, 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 *KITTNN* (see chapter 2) provides these functions and is capable of working with datasets of the same structure as the public *.py* classifiers (see API in appendix A4.1).

In the overall process diagram (Fig. 3.1) there is a box called *classification with full networks*. The procedure behind this box is illustrated on Fig. 3.19.

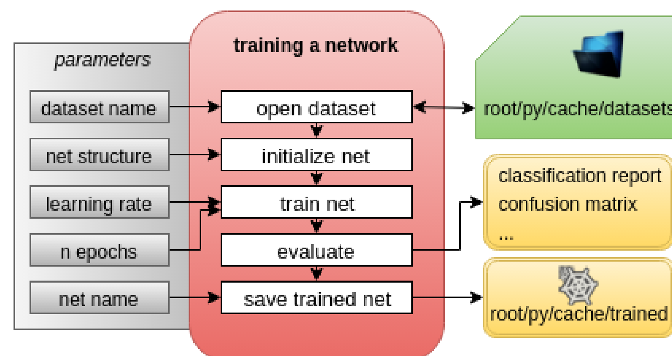


FIGURE 3.19: Procedure of training and testing a network

This workflow is performed by the implemented framework *KITTNN* (chapter 2, as well as by other classification methods (discussed in section 1.1) for comparison. It is advantageous, that each of these tools can use the same workflow and so the comparison is fair.

There are several arguments (firstly listed in section 3.1) 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. A2.8) 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 2.3.

3.7.1 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:

$$F1 = \frac{2 * precision * recall}{precision + recall} \tag{3.13}$$

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 : given as the cost function J in Eq. (2.7).

average epoch time : processing time of one training epoch, computed as an average out of more observations (usually 10)

average classification time : processing time of propagating one sample through the network (also an average out of more observations)

Chapter 4

Experimental Evaluation

In this chapter the methods introduced in the previous sections are evaluated and results are presented. In section 4.1 the implemented classification framework called *KITTNN* (see chapter 2 and API in appendix A4.1) is verified by comparing to a publicly provided framework. The results of the developed pruning algorithm are shown in section 4.2. The overall terrain classification process is gradually evaluated in section 4.3 and the pruning algorithm results on the terrain data are shown in section 4.4.

4.1 Verification of the Network Implementation

Classification performance of the implemented method *KITTNN* is compared to a *Scikit-NeuralNetwork* (*SKNN*) classifier (Champandard and Samothrakis, 2015) presented in appendix A2.1. The evaluation is performed on two datasets introduced in section 2.4.3.

The following Fig. 4.1 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.

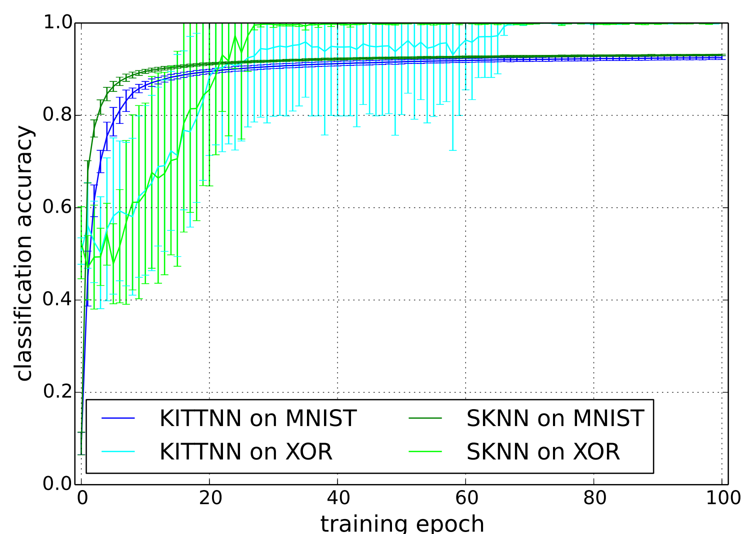


Figure 4.1: Learning process compared to another framework (Scikit-neuralnetwork (sknn): (Champandard and Samothrakis, 2015)).

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. 4.1) for XOR, as there are a lot 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 4.1, the *f1-score* (see Eq. (3.13) in section 3.7.1) is shown for individual classes (digits) of the MNIST dataset. This evaluation is done on the testing data.

TABLE 4.1: Comparison of f1-score on MNIST to another framework (*SKNN*)

net	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*avg*
KITTNN	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. 4.2, a comparison of average epoch processing time is shown.

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. The average is computed out of 1000 samples, as we train 100 epochs in 10 observations.

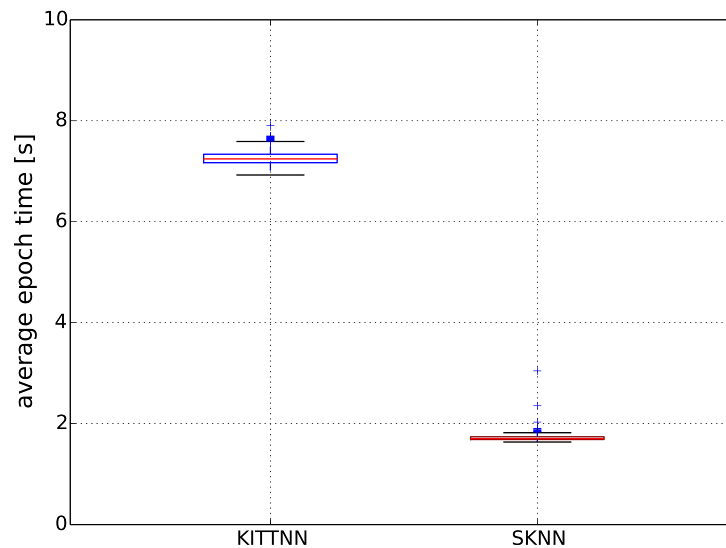


FIGURE 4.2: Comparison of average epoch processing time (1000 samples) to another framework (Scikit-neuralnetwork (*SKNN*): (Champandard and Samothrakis, 2015)), MNIST dataset

The implemented *KITTNN* framework cannot compete in speed with the optimized stochastic *GDA* of *SKNN* library. However, importantly for this study, the classification abilities have been verified.

4.2 Performance Evaluation of the Pruning Algorithm

This section presents results of the implemented pruning algorithm (introduced in section 2.4). 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 evaluation of the algorithm is performed on the two datasets, XOR and MNIST, described in section 2.4.3.

4.2.1 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 ($[2, 2, 1]$). Initially, a network with one hidden layer of 100 neurons is constructed as the algorithm input (Fig. 4.4a). The desired structure is shown in Fig. 4.4b.

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. 2.7) all together in Fig. 4.3.

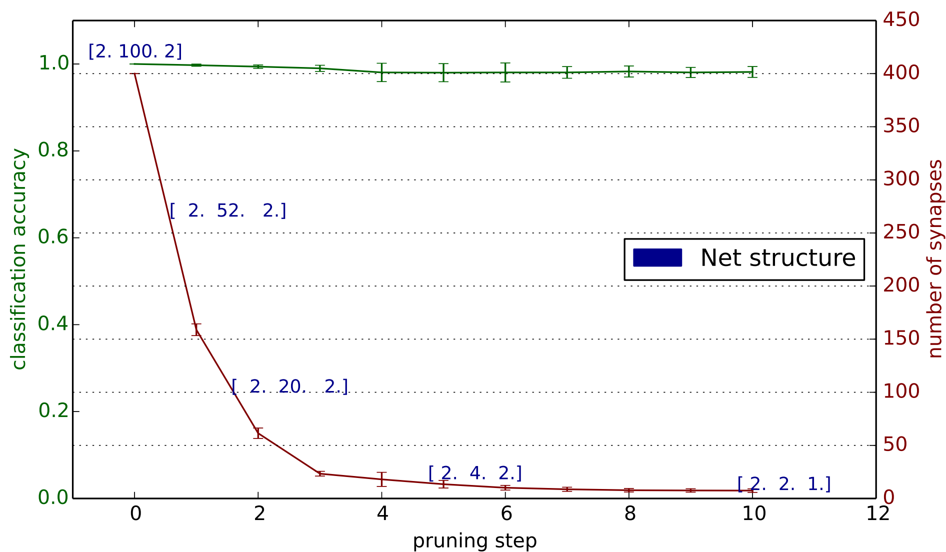


Figure 4.3: Results of the pruning algorithm on XOR dataset.

The results are obtained by averaging 10 observations, so the outcome is independent on initial conditions. For retraining the network after a pruning step, training data is used. For verification, whether the network is capable

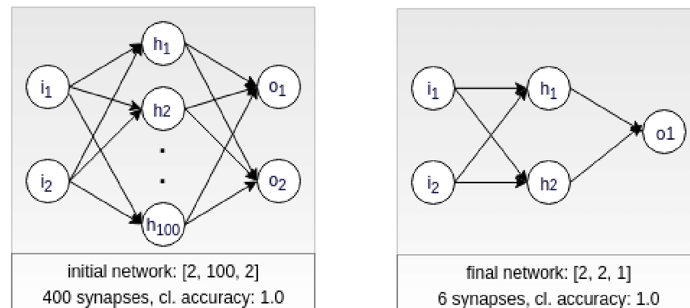
of classification, validation data is used. Testing data is used to check the accuracy after pruning (values in Fig. 4.3).

Algorithm parameters:

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

Algorithm outcomes:

- pruning steps: 10;
- final structure: $[2, 2, 1]$;
- number of removed synapses: 294 (initially: 400, finally: 6);
- classification accuracy on testing data: 0.99.



(A) PA input: oversized and fully-connected network (B) PA output: minimal network structure

FIGURE 4.4: PA process illustration on XOR

4.2.2 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. 4.1 the classification accuracy of *KITTNN* 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. 4.5 shows a significant reduction of synapses while the classification accuracy is kept on 0.89. The shown results are obtained by averaging 10 observations.

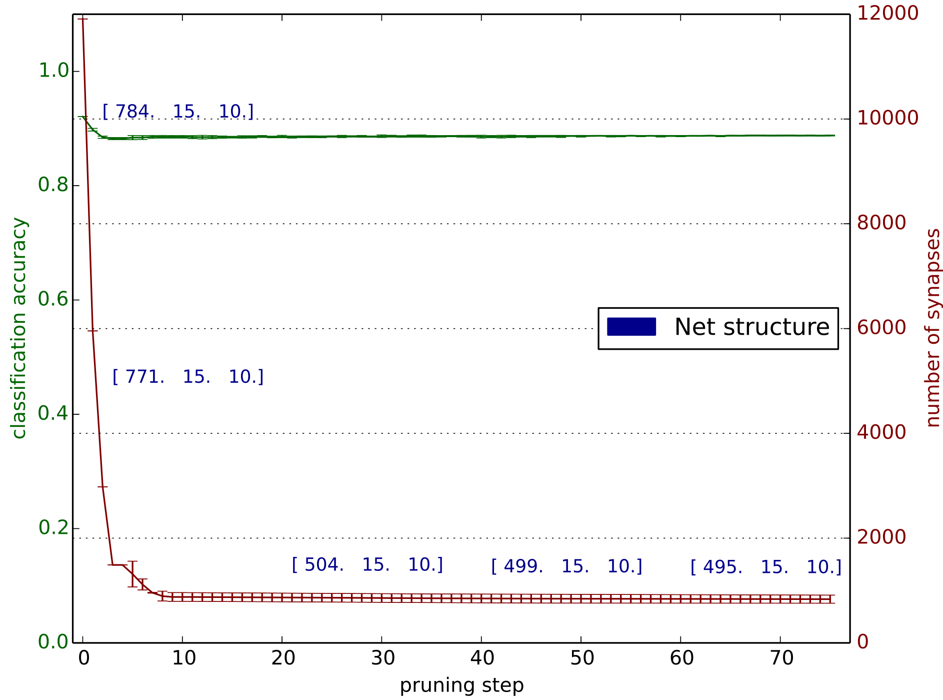


FIGURE 4.5: Pruning Algorithm Results on MNIST Dataset.

Results of the pruning algorithm on MNIST:

- pruning steps: 76;
- final structure: [495, 15, 10];
- number of removed synapses: 11075 (initially: 11910, finally: 835, reduction: 92.9%);
- classification accuracy on testing data: 0.88776.

The figure 4.5 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 4.2.2.

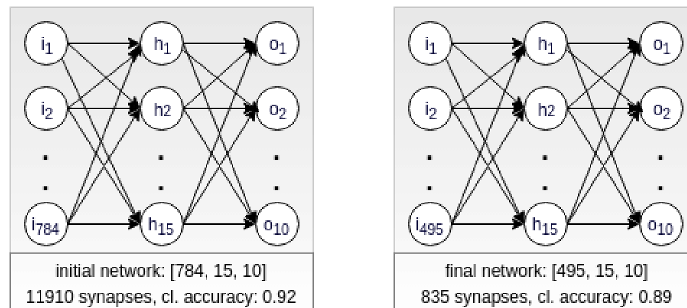
The following Table 4.2 presents statistics of the first seven steps of one of the performed observations. For each step we can see the current percentile level (see Fig. 2.7), corresponding reduction in the network and retraining results.

TABLE 4.2: 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 a half of the synapses each time. In the fourth step, the network is not able to retrain after cutting a 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 (4.6) illustrate the transformation of the network.



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

(B) PA output: minimal network structure

FIGURE 4.6: PA process illustration on MNIST

Analysis of Minimal Structure in MNIST Dataset

As mentioned in section 2.4.4, 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.

As illustrated in Fig. 2.13, 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 to it. Individual subpanels (a-j) in Fig. 4.7 show in red all input neurons (pixels of the 28x28 input images) that are connected to the corresponding output neuron (digit 0-9). Pixels having no path to the corresponding output neuron are in blue.

The subpanel (k) in Fig. 4.7 shows in red all pixels that have at least one path to the output layer. The blue pixels have no connection and hence do not contribute to classification.

As there are 15 hidden neurons, each input neuron can have maximally 15 connections and, naturally, minimally 0 connections in the final structure. The subpanel (l) in Fig. 4.7 presents the number of connections to the hidden layer for each input neuron. The results indicate that one neuron has 3 connections to the hidden layer at most.

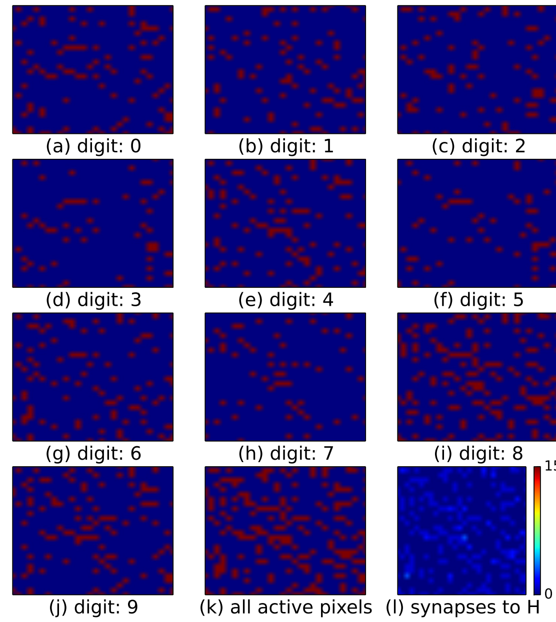


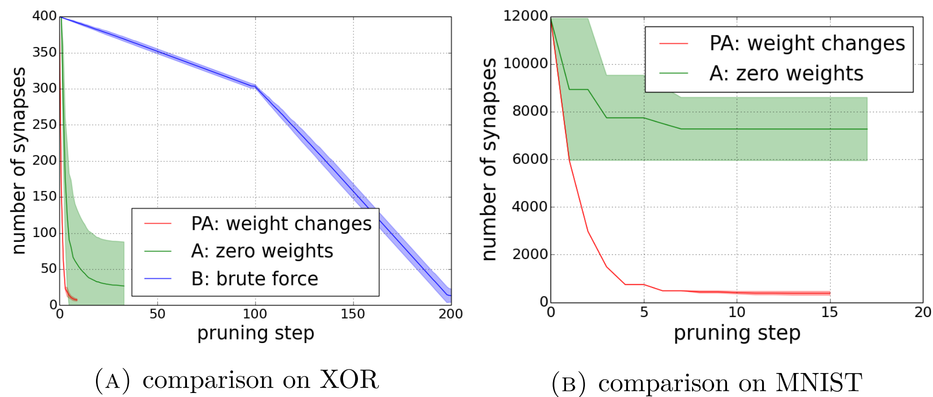
FIGURE 4.7: Feature Selection : MNIST Analysis.

4.2.3 Comparison to Other Pruning Methods

In section 1.1, a few approaches of network pruning are discussed. In this section, the developed pruning algorithm (PA) based on weight changes during network training (see section 2.4) is compared to two different methods:

A *pruning based on actual weight values*: The learning algorithm uses the *tanh* transfer function (see Fig. 2.3). The pruning approach from section 2.4 is used, however, the removed synapses are chosen based on actual weight values. Synapses with the weight closest to zero are removed;

B *brute force pruning algorithm*: In every pruning step, one synapse is removed and the network is retrained. If the retraining was unsuccessful, the synapse is put back to the network;



(A) comparison on XOR

(B) comparison on MNIST

FIGURE 4.8: Comparison of the developed PA to other pruning methods (10 observations).

The comparison was performed on XOR and on MNIST (see section 2.4.3) by averaging results of 10 observations. The brute force approach (B) was only done on XOR as it would take up to 11910 pruning steps and a long evaluation time on MNIST. Regarding its performance on XOR, only one synapse is removed in each of the first 98 steps. Then, removing also the second synapse coming to a hidden neuron, this neuron dies as it has no input, hence also synapses on its output are removed. Therefore more synapses are removed per step after 98th iteration as shown in Fig. 4.8a. The minimal structure is always found by the brute force approach.

Regarding the approach (A), finding the minimal structure is not guaranteed (see Table 4.3). Apparently, the way of selecting removed synapses is not correct, which indicates that even synapses with a low weight may be important.

TABLE 4.3: Comparison of the developed PA to other pruning methods (10 observations). Required accuracy on validation data: XOR: 0.99, MNIST: 0.85.

	XOR				*MNIST*			
	steps	*synapses*	*structure*	*accuracy*	*steps*	synapses	structure	accuracy
PA	10	6	[2, 2, 1]	0.9795	16	377	[252, 15, 10]	0.8396
A	34	27	[2, 8, 1]	0.9875	18	7269	[784, 15, 10]	0.6471
B	218	6	[2, 2, 1]	0.9685	X			

4.3 Results of Terrain Classification

In this section, results of the overall terrain classification process (see chapter 3) are presented and the global process parameters are analysed.

First of all, complete classification results based on the generated datasets are listed. Then, more detailed results of a deterministic configuration are compared to a realistic noisy configuration. Different classification methods are also applied on the noisy dataset for comparison.

In section 4.3.2, the way of finding optimal learning parameters is presented. Then, an analysis of terrain noise and signal noise influence is done in section 4.3.3 and the time needed for classification is determined in section 4.3.4. Classification using separately different sensor types is evaluated in section 4.3.5.

4.3.1 Classification Performance

The generated datasets, differing from each other in the global process parameters, are listed in Table A1.1. For each of them, 10 observations of the classification process were performed and the classification results presented in Table 4.4 were computed by averaging these observations.

Based on the analysis in section 4.3.2, the network hidden structure consisted of 1 layer with 20 neurons. The backpropagation learning rate was 0.5 and the training process lasted for 500 learning epochs.

With reference to Table A1.1, dataset 00_00_40_*a* (meaning no terrain noise, no signal noise, 40 timesteps and all sensors) was selected for more detailed analysis. Classification results on this dataset are considered as a reference, because the data is deterministic and the results best possible.

TABLE 4.4: Classification results on generated datasets (see dataset parameters in Table A1.1).

dataset	*accuracy*	*precision*	*recall*	*f1-score*	*error*	*ep. time [s]*
00_00_80_p	0.624	0.596	0.624	0.576	0.072	1.089
00_00_80_a	0.922	0.920	0.920	0.920	0.020	1.330
00_03_40_a	0.902	0.900	0.900	0.900	0.019	0.765
00_00_30_a	0.847	0.840	0.848	0.842	0.030	0.964
00_00_10_p	0.489	0.468	0.490	0.430	0.089	0.565
00_01_40_a	0.776	0.790	0.780	0.760	0.040	0.751
00_00_01_a	0.639	0.646	0.638	0.624	0.073	0.526
00_00_80_t	0.761	0.764	0.760	0.742	0.051	0.715
00_00_20_a	0.850	0.848	0.850	0.846	0.030	0.894
00_05_40_a	0.884	0.880	0.880	0.880	0.019	0.760
00_00_40_a	0.923	0.920	0.920	0.920	0.019	0.865
00_00_10_a	0.805	0.806	0.804	0.798	0.040	0.526
00_00_40_p	0.696	0.702	0.694	0.666	0.060	0.707
00_00_10_t	0.423	0.432	0.424	0.386	0.102	0.469
00_10_40_a	0.851	0.850	0.850	0.850	0.026	0.792
00_00_40_t	0.701	0.692	0.702	0.670	0.064	0.475
01_03_40_a	0.772	0.780	0.770	0.760	0.044	1.314
01_10_40_a	0.639	0.670	0.640	0.630	0.060	0.944
01_01_40_a	0.759	0.770	0.760	0.750	0.052	1.093
01_00_40_a	0.815	0.830	0.810	0.820	0.041	0.844
01_05_40_a	0.780	0.790	0.780	0.770	0.044	0.900
03_10_40_a	0.565	0.530	0.560	0.540	0.063	0.946
03_05_40_a	0.645	0.640	0.650	0.630	0.054	0.871
03_00_40_a	0.714	0.730	0.710	0.700	0.057	0.831
03_03_40_a	0.719	0.740	0.720	0.720	0.046	0.939
03_01_40_a	0.779	0.780	0.780	0.780	0.042	1.068
05_01_40_a	0.732	0.740	0.730	0.730	0.052	1.018
05_05_40_a	0.651	0.660	0.650	0.640	0.056	0.933
05_10_40_a	0.601	0.610	0.600	0.600	0.062	0.954
05_03_40_a	0.677	0.710	0.680	0.670	0.052	0.949
05_00_40_a	0.685	0.720	0.690	0.680	0.055	0.835
10_10_40_a	0.459	0.460	0.460	0.430	0.087	0.983
10_03_40_a	0.621	0.630	0.620	0.600	0.070	0.824
10_01_40_a	0.594	0.630	0.590	0.580	0.072	1.131
10_00_40_a	0.614	0.610	0.610	0.590	0.067	0.822
10_05_40_a	0.543	0.580	0.540	0.530	0.074	0.875
20_01_40_a	0.459	0.480	0.460	0.440	0.093	1.071
20_00_40_a	0.463	0.470	0.460	0.440	0.087	0.827
20_05_40_a	0.436	0.450	0.440	0.410	0.091	0.947
20_10_40_a	0.371	0.360	0.370	0.360	0.090	0.993
20_03_40_a	0.439	0.470	0.440	0.420	0.091	0.923

The following Fig. 4.9 illustrates a confusion matrix (see section 3.7.1) for classification on the reference deterministic dataset (00_00_40_*a*). The

recall rate for all terrains is shown on the diagonal. Additionally, we can see that 15% of carpet samples was mistakenly labeled as grass and 10% of grass samples as carpet. Similarly, the network had difficulties to distinguish rock from grass and wood or foam from swamp. However, in general, the classification on the deterministic dataset was successful.

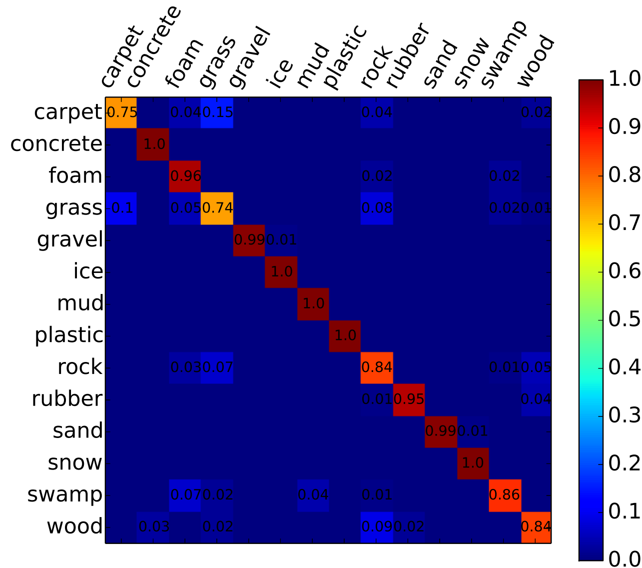


FIGURE 4.9: Confusion matrix of classification results on a deterministic dataset.

Fig. 4.10 shows a confusion matrix of classification results on a noisy dataset 03_03_40_*a* (3% relative terrain noise, 3% relative signal noise, 40 timesteps and all sensors - see Table A1.1).

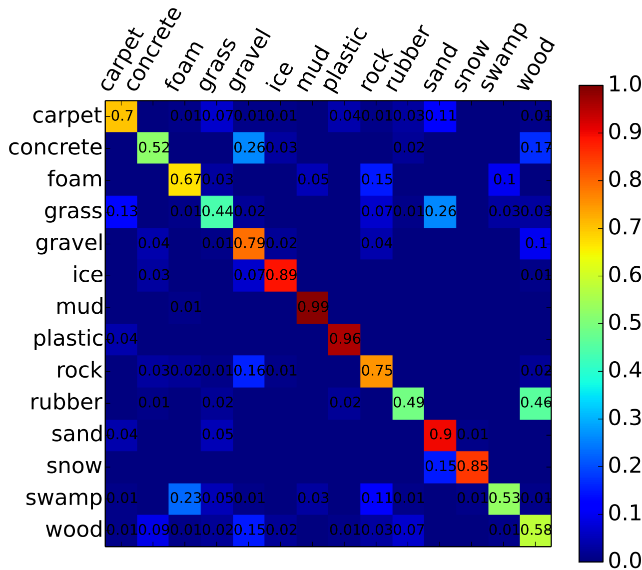


FIGURE 4.10: Confusion matrix of classification results on a noisy dataset.

A complete analysis of the relation between a noise level and classification performance is presented in section 4.3.3. However, assuming that the chosen noise level simulates a real environment, Fig. 4.10 predicts results of the method on the real platform.

In this case, grass (44%) and rubber (49%) result with the worst recall. On the other hand, samples of mud, plastic or sand are classified very well. A complete classification report comparing the deterministic results to the noisy results is shown in Table 4.5.

TABLE 4.5: Classification report for a deterministic dataset and a noisy dataset.

	deterministic data		noisy data		*support*
	precision	*recall*	*precision*	*recall*	
carpet	0.88	0.75	0.75	0.70	100
concrete	0.97	1.00	0.72	0.52	100
foam	0.83	0.96	0.70	0.67	100
grass	0.74	0.74	0.63	0.44	100
gravel	1.00	0.99	0.54	0.79	100
ice	0.99	1.00	0.91	0.89	100
mud	0.96	1.00	0.93	0.99	100
plastic	1.00	1.00	0.93	0.96	100
rock	0.77	0.84	0.65	0.75	100
rubber	0.98	0.95	0.78	0.49	100
sand	1.00	0.99	0.63	0.90	100
snow	0.99	1.00	0.98	0.85	100
swamp	0.95	0.86	0.79	0.53	100
wood	0.88	0.84	0.42	0.58	100
avg / total	0.92	0.92	0.74	0.72	1400

Comparison to Other Classification Methods

Using the noisy dataset 03_03_40_*a*, the classification performance of the developed framework *KITTNN* was compared to other classification approaches presented in section 1.1 (Table 4.6).

TABLE 4.6: Comparison to other classification methods implemented by (Pedregosa et al., 2011)

method	method param.	*accuracy*	*precision*	*recall*	*f1-score*
KITTNN	hidden neurons: 20	0.719	0.740	0.720	0.720
SKNN	hidden neurons: 20	0.812	0.820	0.810	0.810
RF	estimators: 10	0.647	0.650	0.650	0.640
SVMs	kernel: 'rbf'	0.545	0.570	0.550	0.500
k-NN	neighbours: 10	0.733	0.740	0.730	0.720

The comparison to *SKNN* indicates that the *KITTNN* implementation might be optimizable. However, in general, the results proved the suitability of using neural networks.

4.3.2 Selection of Learning Parameters

This section is devoted to an analysis of parameters belonging to the network and the learning process. The goal was to determine:

1. hidden structure of the network;
2. learning rate for the backpropagation;
3. number of learning epochs (iterations).

To perform the analysis, the deterministic dataset (00_00_40_a - see Table A1.1) was used.

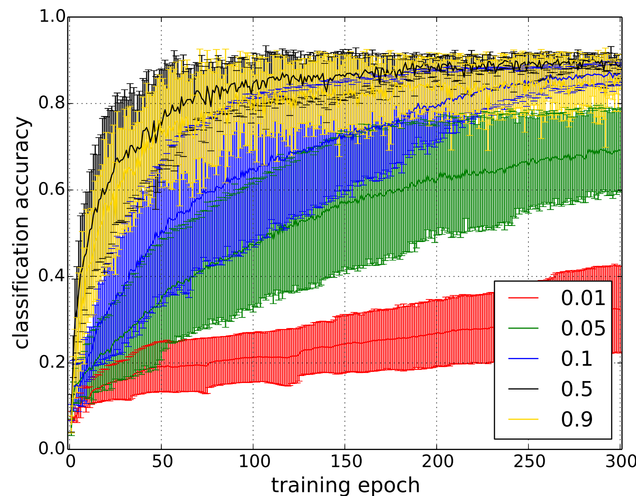


FIGURE 4.11: Training process with various learning rates.

In Fig. 4.11, the training process (progress of accuracy over iterations) is illustrated for five learning rates. The results are obtained by averaging 10 observations and (standard deviation) error bars are shown. The experiment was performed using a network with 20 hidden neurons.

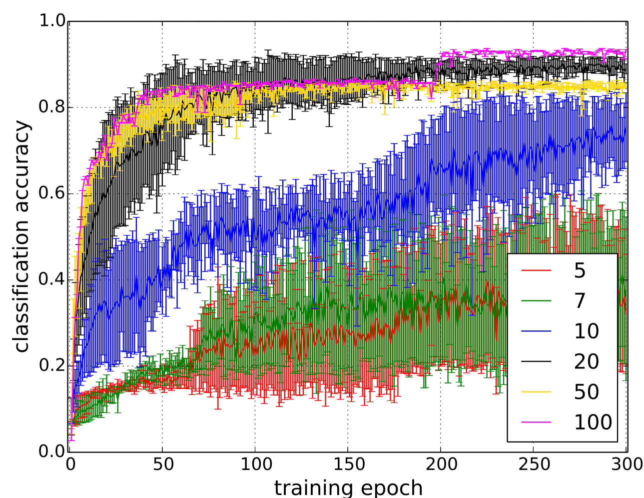


FIGURE 4.12: Training process with various networks differing in the number of hidden neurons.

It was determined to use one hidden layer. The accuracy progress over training epochs for six networks differing in the number of hidden neurons is shown in Fig. 4.12. An overall analysis for these two parameters (learning rate and network structure) is performed in Fig. 4.13.

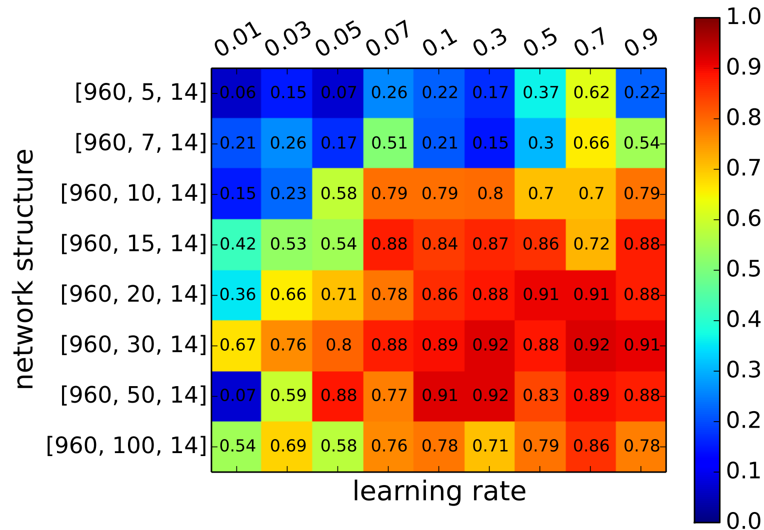


FIGURE 4.13: Classification accuracy vs. learning rate and network structure (10 observations)

Based on the results, 20 neurons in one layer were chosen to form the hidden structure of the network. The learning rate was set to 0.5. The number of training epochs (based on Fig. 4.11 and Fig. 4.12) were set to 500.

4.3.3 Influence of Noise on Classification

Two types of noise were added to the deterministic data during the process: a terrain noise (described in section 3.3.3) and a signal noise (section 3.5.2). Both were parametrized by a standard deviation value. Fig. 4.14 shows the influence of these values on the classification performance.

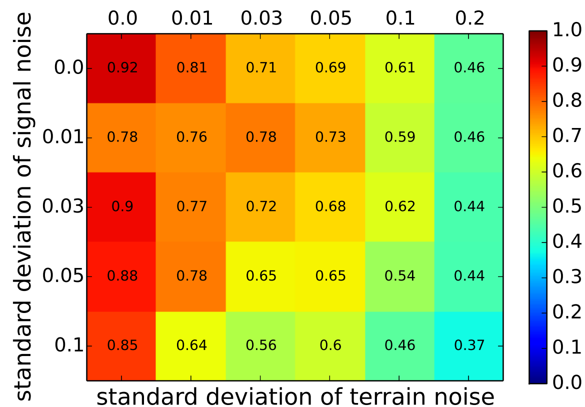


FIGURE 4.14: Additive terrain and signal noise: influence on the accuracy (10 observations).

4.3.4 Time Needed for Classification

As discussed in section 3.4, the assumption is that the robot needs to make several steps and record the sensory data over a period of time, to classify the terrain with proprioceptive and tactile sensors. In this section, the period of time needed for proper classification is analysed. Assuming that one second in reality is equal to 10 simulation steps, Fig. 4.15 shows the classification accuracy for six different numbers of simulation steps. The classification was performed on the deterministic dataset (00_00_40_*a* - see Table A1.1).

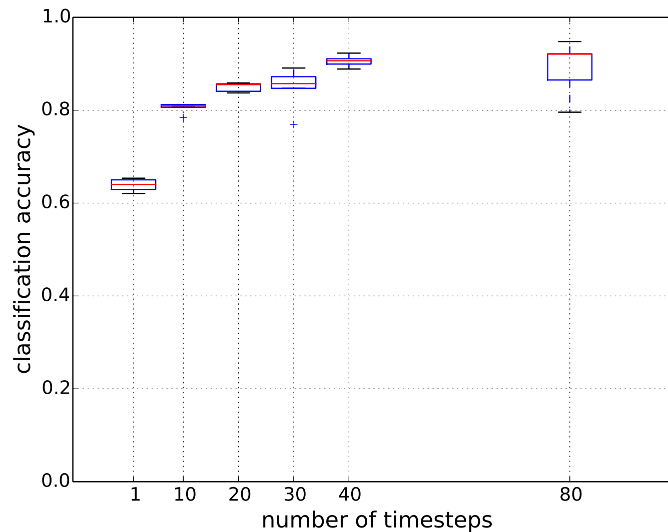


FIGURE 4.15: Analysis of time needed for proper classification (10 observations).

Surprisingly, the classification result based on a single timestep is not bad. In other words, the robot will be able to classify the terrain in real time with more than 0.6 probability of success. In general, the more timesteps are used, the better the result is.

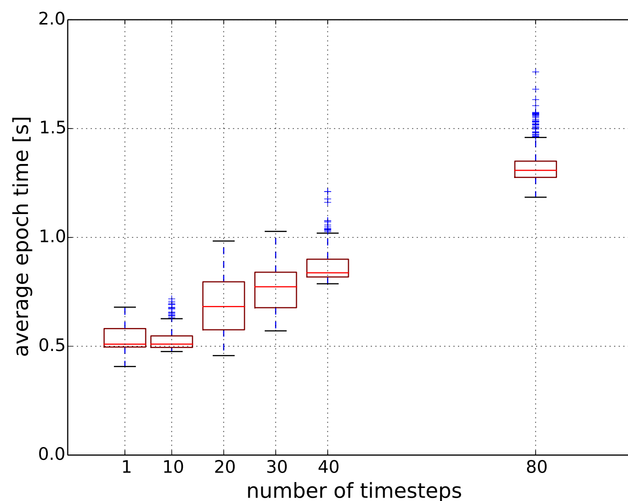


FIGURE 4.16: Average epoch time (10 observations) depending on the number of simulation timesteps.

However, the equally good results for 40 and 80 timesteps indicate, that one period of the tripod gait will be enough (see the periodic sensory signals in appendix A1.1). Moreover, with the increasing number of timesteps, also the feature vector size and network dimensionality increase, and, as shown in Fig. 4.16, the processing time as well. Based on this analysis, 40 was chosen as the default number of timesteps in this study.

4.3.5 Analysis of Used Sensor Types

Two types of sensors are used for terrain classification for the hexapod robot AMOS II:

1. proprioceptive sensors (3 on each leg, 18 in total);
2. tactile sensors (1 on each leg, 6 in total).

The results presented so far are obtained using both of these two types together. In this section, the sensor types are evaluated separately. Fig. 4.17 illustrates the training process using the sensor types one by one and compares their classification performance to a combined configuration. The evaluation is performed for three values of simulation timesteps (see section 4.3.4): 10, 40, 80.

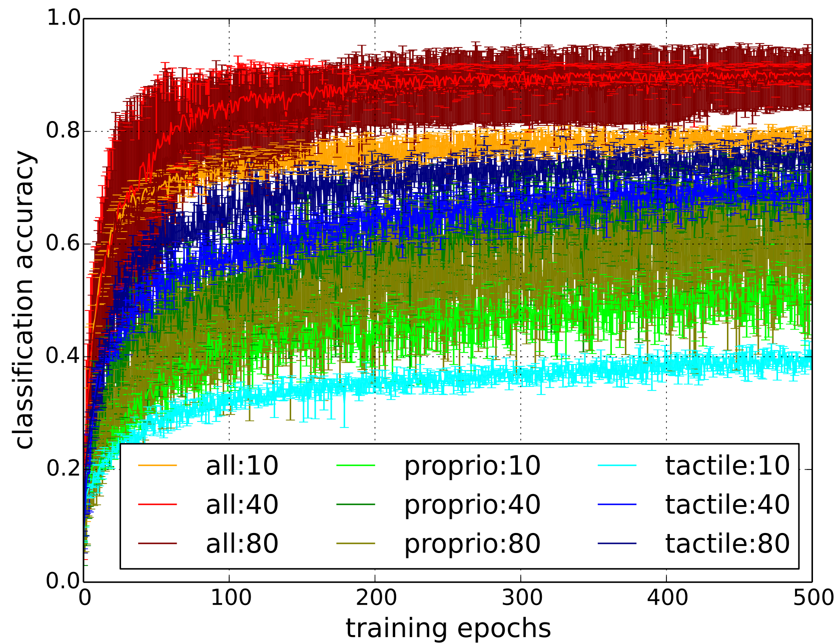


FIGURE 4.17: Evaluation of different sensor types separately (average of 10 observations, timesteps: 10, 40, 80).

Fig. 4.17 shows that the combination of the two types performs with the highest classification accuracy, namely even with 10 simulation steps (1 second of real time) compared to 40 (4 seconds), resp. 80 (8 seconds), for only one of the sensor types. Having more time for classification (4 or 8 seconds), the six tactile sensors are more successful than the proprioceptive ones, however, they fail when the classification needs to be fast (1 second). Complete classification results for all of these configurations can be found in Table 4.4.

The number of used sensors affect the size of the feature vector and hence also the size of the network input layer. The dimensionality of the network matrices influences the processing time. An average epoch time out of 10 observations is shown for the two sensor types in Fig. 4.18.

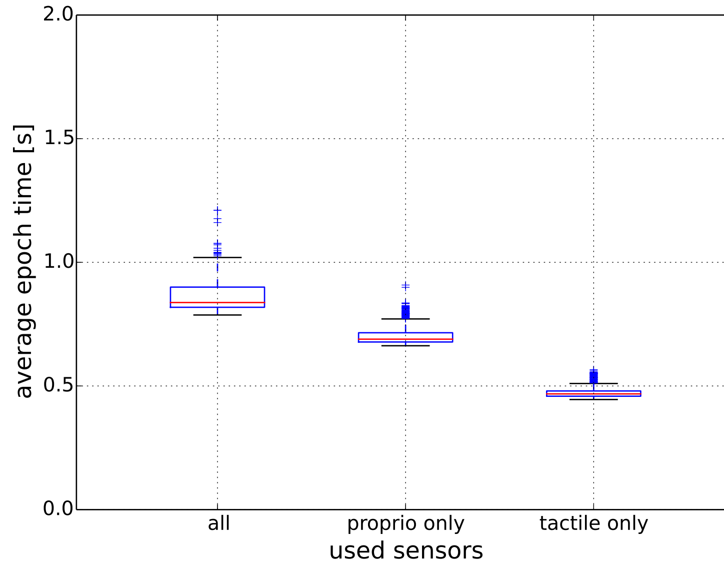


FIGURE 4.18: Average epoch time for different sensor types.

4.4 Terrain Classification Using Network Pruning

In this section, the developed pruning algorithm (section 2.4) is applied on the terrain classification problem (chapter 3). The datasets containing the terrain data are listed in Table A1.1. On each of them, a neural network was trained, evaluated and saved (classification results in Table 4.4). These trained nets were used as the input of the pruning algorithm to obtain the following results.

The configurations listed in Table 4.7 were chosen for demonstration of the PA on terrain classification:

TABLE 4.7: Chosen configurations for PA demonstration.

	dataset	*ter./sig. noise*	*timesteps*	*sensors*	*hidden neurons*
A	00_00_40_a	0/0	40	all	20
B	00_00_80_a	0/0	80	all	20
C	00_00_40_a	0/0	40	all	100
D	03_03_40_a	0.03/0.03	40	all	20
E	00_00_40_p	0/0	40	proprioceptive	20
F	00_00_40_t	0/0	40	tactile	20

The result of the pruning algorithm applied on the default configuration (A) is illustrated in Fig. 4.19. The required classification accuracy, number of

synapses and network structure are shown with respect to the pruning step. The result was obtained by averaging 10 observations.

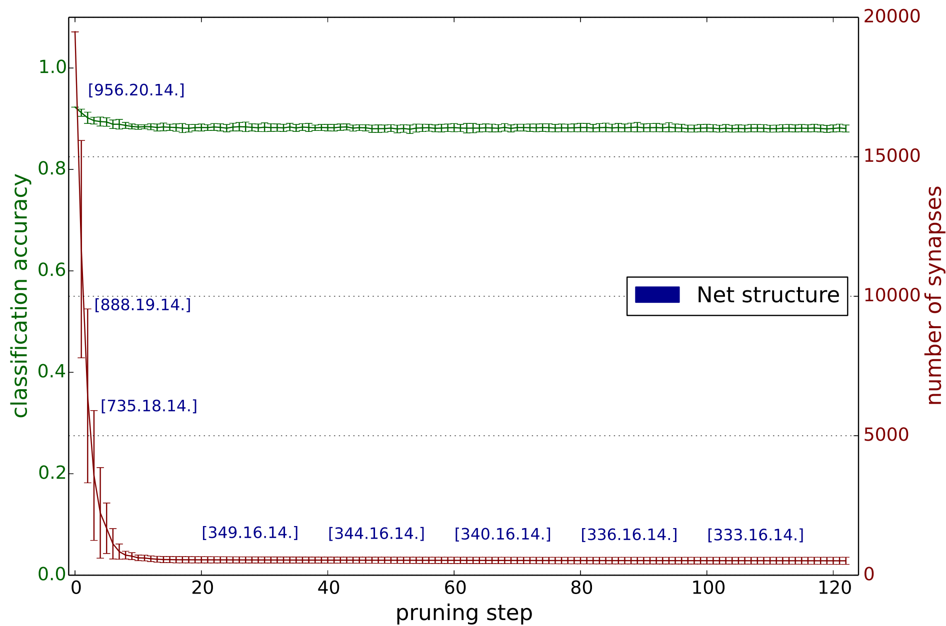


FIGURE 4.19: Pruning Algorithm Results on AMTER Dataset. No noise.

The number of synapses were reduced from 19400 to 516. Complete results for all the listed configurations from Table 4.7 are presented in Table 4.8.

TABLE 4.8: Results of the PA on terrain datasets.

	required accuracy	*structure before*	*synapses before*	*structure after*	*synapses after*	*accuracy on testing set*	*pruning steps*
A	0.9	[960, 20, 14]	19400	[330, 16, 14]	516	0.8807	123
A'	0.75	[960, 20, 14]	19400	[69, 10, 13]	123	0.6437	27
B	0.9	[1920, 20, 14]	38680	[399, 16, 14]	531	0.8859	363
C	0.65	[960, 100, 14]	97400	[136, 33, 14]	204	0.7384	28
D	0.7	[960, 20, 14]	19400	[331, 17, 14]	534	0.6787	76
E	0.65	[720, 20, 14]	14680	[116, 8, 12]	182	0.6601	92
F	0.65	[240, 20, 14]	5080	[44, 16, 13]	144	0.6601	46

As we can see, the network D, trained on the noisy dataset, ends with very similar results as the reference A configuration. The structural and synaptic reduction of configuration B (using 80 timesteps) takes more pruning steps, but also ends up with comparable results. As A, B requires the same accuracy, we can compare their synaptic pruning process in a graph (Fig. 4.20). Additionally, configuration D is included, as its pruning progress is also comparable.

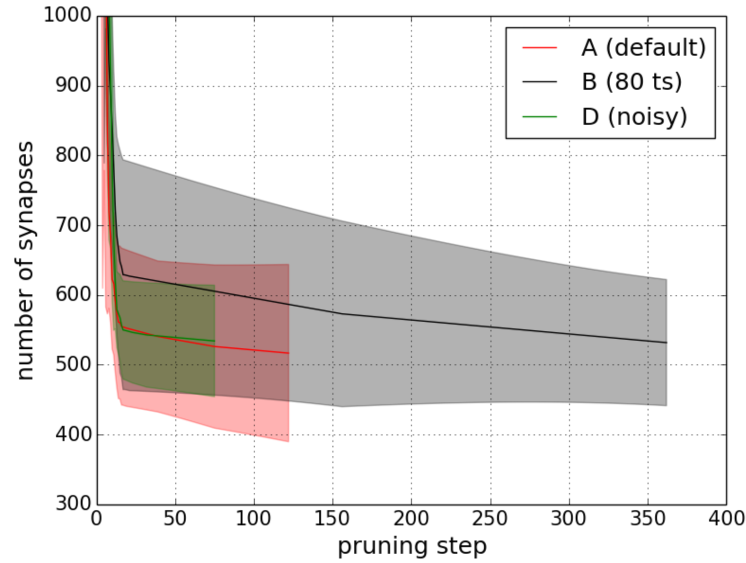


FIGURE 4.20: Synaptic pruning of configurations A (reference), B (80 timesteps) and D (noisy data).

Similarly, we can compare the remaining configurations (*C*, *E*, and *F*), as all of them require the accuracy of 0.65 (Fig. 4.21. Additionally, the *A* (default) configuration is pruned with the same required accuracy as *C* (0.75) and added for comparison (listed as *A'*). Ideally, *A'* would end up with the same result as the huge network (*C*).

Interestingly, when pruning *E* and *F* configurations, the PA decided to omit some of the terrains completely (see Table 4.8 and Fig. 4.23), which is allowed since only 65% of accuracy was required.

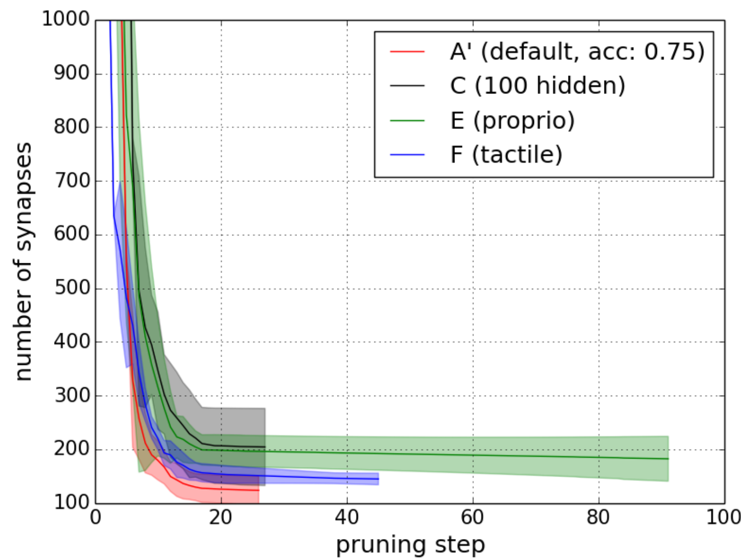


FIGURE 4.21: Synaptic pruning of configurations A' (reference), C (100 hidden neurons), E (proprioceptive sensors) and F (tactile sensors).

The following Fig. 4.22 shows a relative amount of active neurons after pruning for configurations A, B and D. Each configuration is also labeled by the required accuracy and the initial number of neurons per a layer.

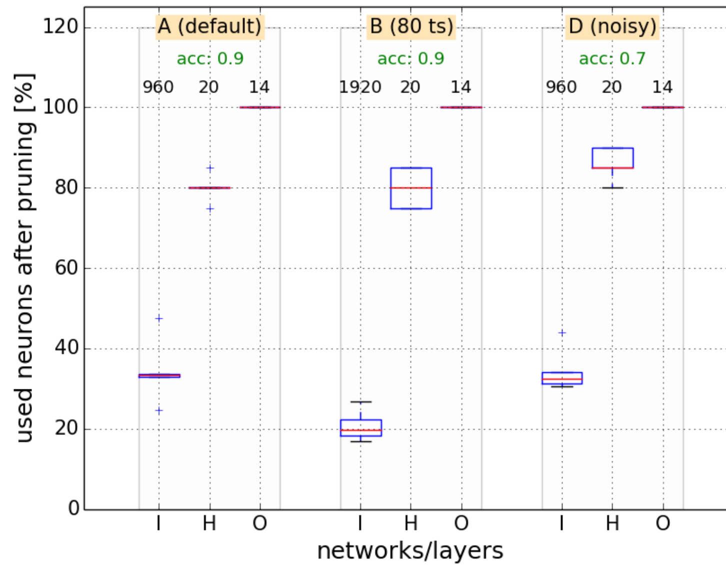


FIGURE 4.22: Active neurons in the network after pruning [%]: configurations A, B, D (10 observations)

In Fig. 4.23, the same statistics for configurations C, E and F, completed by *A'*, is shown. In general, input layers are reduced the most. The results also indicate that proprioceptive sensing does not need so many hidden units compared to tactile sensing.

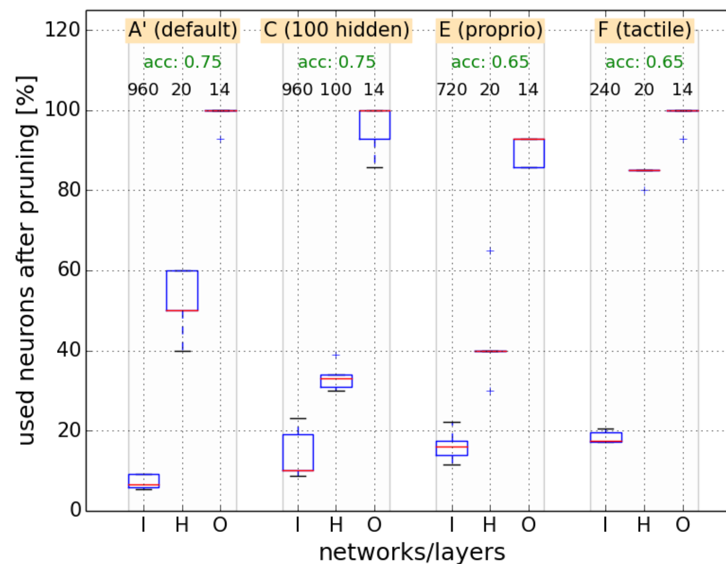


FIGURE 4.23: Active neurons in the network after pruning [%]: configurations A', C, E, F (10 observations)

4.4.1 Feature Selection for Terrain Classification

Section 2.4.4 presents an idea of using minimal structures for a feature selection. The idea is based on tracking paths from input neurons representing individual features to output neurons (classes). This approach was applied on the minimal structure of the A configuration from Table 4.8, hence the analysis is performed on a network of structure $[330, 16, 14]$ with 516 synapses.

Fig. 4.24 shows three sample examples (right y-axis). On the left y-axis, the number of paths (see Fig. 4.25) to the output layer is counted for each feature.

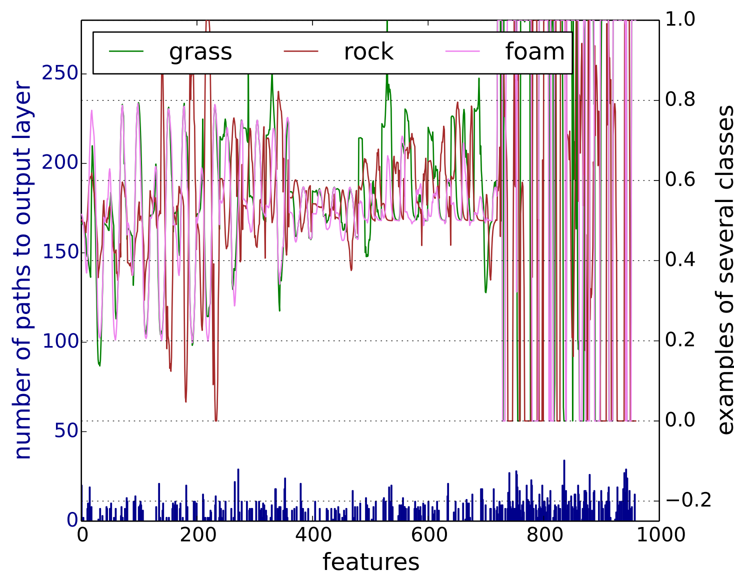


FIGURE 4.24: Number of paths to the output layer for every feature of the input example.

Fig. 4.25 explains the meaning of a path in this context.

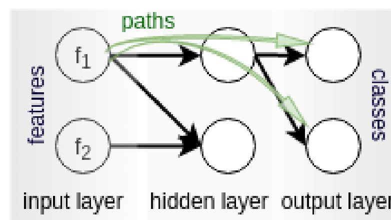


FIGURE 4.25: Explanation of a path from input to output layer in a minimal structure.

Having these paths, we can distinguish features important for individual classes. In Fig. 4.26, the analysis is done for every feature of the Thoraco-Coxa joint angle sensors. Red dots mean that the feature has a path to the corresponding class (terrain). The same analysis for tactile sensors is shown in Fig. 4.27. Similar figures for *coxa* and *femur* joint angle sensors can be found in appendix A1.3.

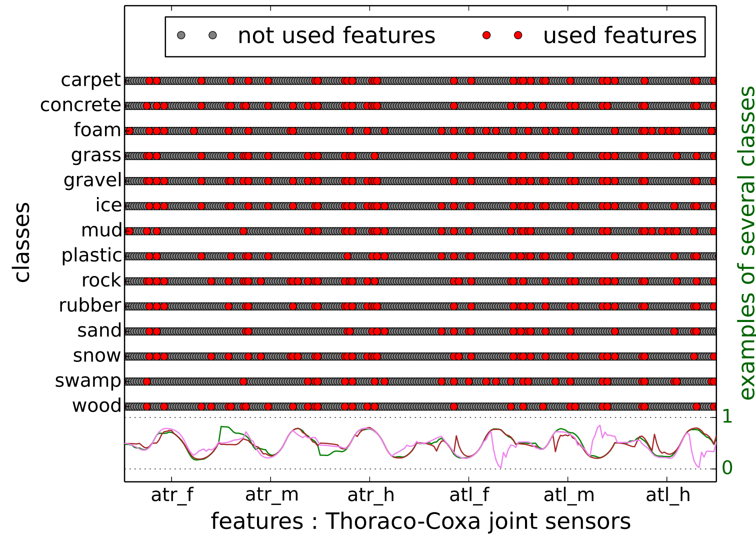


FIGURE 4.26: Used features of thoraco-coxa proprioceptive sensors for individual classes.

Using this analysis, one might find the redundant parts of the feature vector, as some of the features are unimportant for any of the classes. Moreover, we can make statements about single features based on the classes they are connected to. For instance, if a feature, belonging to the foot contact sensor on the right front leg, fires for foam and does not fire for the other classes, we might assume that this sensor is important if we want to classify foam properly.

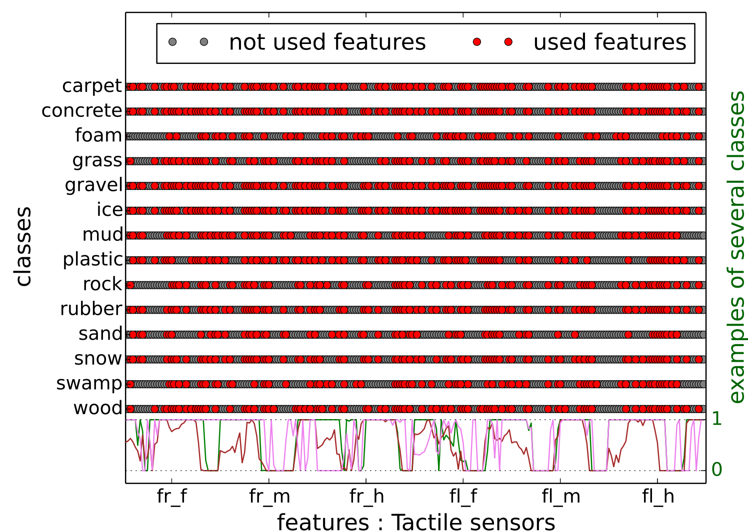


FIGURE 4.27: Used features of tactile sensors for individual classes.

To go even further with this analysis, also weights of the minimal network structure can be used. For every feature and every class, the Fig. 4.29 shows the sum of weights of the path from the feature to the class.

The *power* of feature *f*, corresponding to input neuron i_r, and class c, represented by output neuron o_q, is computed as given in Eq. (4.1); (assuming one hidden layer).

$$power_{r,q} = \sum_{k=1}^{N_{hidden}} w_{r,k} + w_{k,q} \tag{4.1}$$

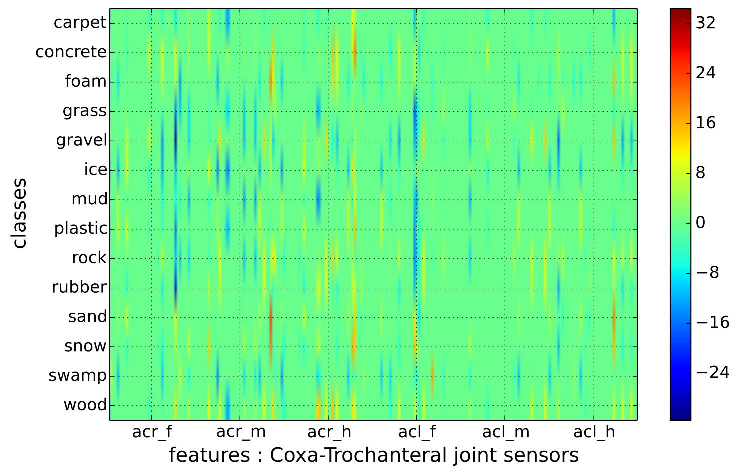


FIGURE 4.28: Influence power of single features on classes: coxa sensors

Based on the results in Fig. 4.29, besides the knowledge if a feature affects a particular class, we can also see the direction (positive/negative) and a power of this influence. For instance, we can find a feature affecting plastic or concrete positively and snow or mud negatively at the same time.

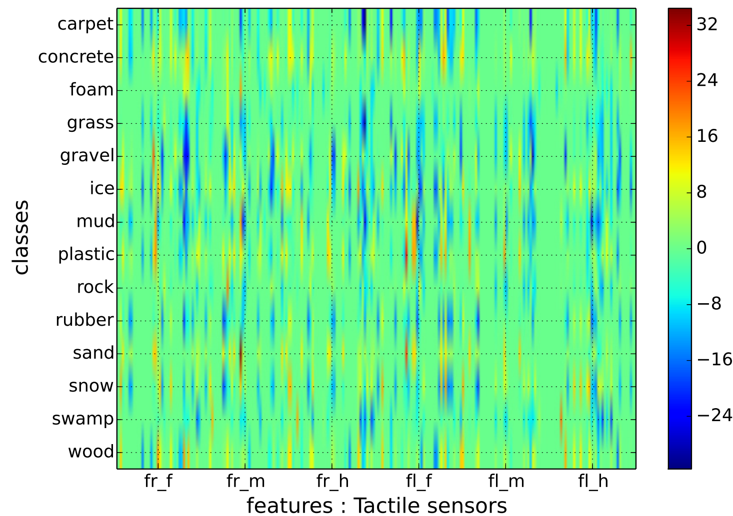


FIGURE 4.29: Influence power of single features on classes: tactile sensors

The results for the thoraco and femur sensors were generated in the same manner and can be found in appendix A1.3.

Chapter 5

Discussion

The objectives of this thesis consisted of four subtasks:

1. implementation of the classification method;
2. development of the new pruning algorithm;
3. generation of datasets for virtual terrains;
4. terrain classification.

5.1 Methods Recapitulation

Firstly, I have implemented a neural network framework capable of classification and I called it *KITTNN* (chapter 2). The network is of the *feedforward* type and the *Backpropagation* learning algorithm is used for network training. As a framework extension, a graphical user interface was created to visualize the training process of smaller networks.

The functionality of the *KITTNN* framework was verified in section 4.1 by comparing to a public implementation (*SKNN*). Two datasets, XOR and MNIST, were used for training and the learning progress was observed over training epochs. The results showed that KITTNN is slower than SKNN in training, however, it is capable of learning and performs with the same classification accuracy once it is trained (section 4.1).

Next, a new network pruning algorithm has been invented. The fundamental idea is to use weight changes during network training for selection of the unimportant synapses. The hypothesis saying that weights of unimportant synapses do not evolve during the training has been experimentally proven. The algorithm was firstly tested on the XOR dataset, where the known minimal network structure $[2, 2, 1]$ was successfully found.

Then the algorithm was used to prune a network for MNIST classification. In this case the number of synapses were reduced from 11910 to 835, which is a reduction of almost 93%, while the classification accuracy of the pruned network was kept on 90%. Pruning the synapses, many of the neurons lost all of their inputs and became inactive. The minimal structure regarding the active neurons for the MNIST dataset is $[495, 15, 10]$ (initially $[784, 15, 10]$).

The obtained structure showed that many of the input neurons (pixels of the MNIST examples) were unimportant for classification, which lead to the

analysis of features for individual classes. In Fig. 4.7, features important for particular digits are shown.

Regarding the third objective, a framework provided by (Martius et al., 2009) was used to simulate the hexapod robot AMOS II walking on different terrains. It was selected to generate 14 terrains in total, where each of them was defined by 5 features: roughness, slipperiness, hardness, elasticity and height (see Table 3.4). These parameters were influenced by a terrain noise to generate more variability for samples of one class.

In total 24 sensors were used (18 proprioceptive and 6 tactile sensors). The signals of these sensors were influenced by an additive signal noise to simulate the real world environment. Finally, feature vectors were built by concatenating sensors one after each other and datasets were generated (a complete list in Table A1.1).

The generated datasets are parametrized by:

1. terrain noise standard deviation;
2. signal noise standard deviation;
3. number of simulation timesteps (length of the sensory signal);
4. used sensors.

Each of the datasets was used for training of a *KITTNN* network, where the trained networks were parametrized by *learning rate*, *network structure*, *number of training epochs.*

The classification results were saved for all of these configurations, which enabled to make a statistical analysis of the parameters.

Here are some observations:

1. Using deterministic proprioceptive and tactile sensory data gathered in a period of 4 seconds, we get 92% of classification accuracy for 14 different terrain types.
2. Using a more realistic configuration with 3% of relative terrain noise and 3% of relative signal noise, the classification accuracy drops to 72% (complete analysis in Fig. 4.14).
3. Compared to 1, when only 2 seconds are used to gather the sensory data, the accuracy drops to 85%. Then, for 1 second the accuracy is 80% and if we want to classify real-time data, we get 64% (analysis in Fig. 4.15).
4. The combination of both sensor types provides the best classification results (hypothesis proved). Using the propriceptors only, the accuracy drops from 92% to 69.6%. Using only tactile sensors ends with the accuracy of 70.1% (see Fig. 4.17).
5. The optimal number of hidden neurons in the fully-connected network is 20, a suitable learning rate can be 0.1 or 0.5 (see section 4.3.2).

Finally, the pruning algorithm was used to find a minimal network structure for the terrain classification task. Regarding the reference configuration A (deterministic dataset, 20 hidden neurons - see 4.7), the number of synapses

in the network were reduced from 19400 to 516 (97.35%) and the structure changed from $[960, 20, 14]$ to $[330, 16, 14]$. The classification accuracy of the pruned network is 88.07% (see Table 4.8 for more detailed results).

Based on the change in the network structure, we know that at least 65% of the features are unimportant and we can even locate them. Furthermore, we can separate features important for individual classes and see the correlations among the classes. Additionally, as shown in Fig. 4.29, we can see the influence power of single features on the classes.

A proper analysis of network minimal structures is definitely a subject for future work.

5.2 Comparison of Results

Based on the literature, this approach is compared to results of 5 terrain classification studies in Table 5.1. The comparison of the classification accuracy must be done with respect to the number of classified terrain types. In this work, we distinguish far more terrain types than the other researchers.

TABLE 5.1: Studies of terrain classification for legged robots.

source	*sensors*	*terrains*	*accuracy*	*platform*	*environment*
(Zenker et al., 2013)	vision	8	0.900	hexapod AMOS II	reality
(Kesper et al., 2012)	laser	3	X	hexapod AMOS II	reality
(Xiong, Worgotter, and Manoonpong, 2014)	tactile	6	0.89	hexapod AMOS II	reality
(Mou and Kleiner, 2010)	vision laser vibration	5	0.96	Matilda Robot	reality
(Hoepflinger et al., 2010)	tactile	4/4	0.94/0.73	tetrapod ALoF	reality
this study	proprioceptive tactile	14	0.923	hexapod AMOS II	simulation deterministic
this study	proprioceptive	14	0.696	hexapod AMOS II	simulation deterministic
this study	tactile	14	0.701	hexapod AMOS II	simulation deterministic
this study	proprioceptive tactile	14	0.719	hexapod AMOS II	simulation noisy
this study	proprioceptive	14	0.426	hexapod AMOS II	simulation noisy
this study	tactile	14	0.362	hexapod AMOS II	simulation noisy

Considering the high number of detected terrains, our results seem to be very positive. However, we must take into account that these results are based on the simulation data. To make a fair comparison, the method should be implemented on the real platform.

Chapter 6

Conclusion and Outlook

A feedforward neural network framework capable of learning and classification, additionally equipped with a new pruning algorithm, has been developed. The hypothesis regarding the network pruning has been proven, as there are many synapses (generally over 90%) in the fully-connected networks, which are unimportant for classification. Furthermore, the resulting minimal structures seem to be useful for feature selection.

The developed framework has been used for terrain classification using 18 proprioception and 6 tactile sensors of a hexapod robot. Using 5 terrain features, 14 different virtual terrain types have been generated.

We achieved over 92% accuracy on deterministic simulation data and 72% on data that was manually noised. Regarding the mentioned sensor types, the sensory data had to be gathered over one period of the tripod walking pattern. Using data of one moment in time only, the accuracy drops to 64% on the deterministic simulation data.

Application of the pruning algorithm for terrain classification showed that the initially used network structure $[960, 20, 14]$ was oversized. The number of input neurons were reduced to 330 and important features were found for individual terrains. The outcome of the process is a minimal, computationally efficient, network. Furthermore, the network knows which features are useful for a successful terrain classification.

To make a final evaluation of the overall study, the main objectives were met and on top of that, some interesting results were obtained.

Regarding the terrain classification process, the implementation on the real platform should be the next step. The proprioception and tactile sensing was proved to work well together on the simulation data.

The field of neural networks provides a wide range of interesting questions to be researched, especially in case of minimal network structures. The feature selection idea has been outlined in this study.

Moreover, the synapses are removed from the network by setting weights to zero, however, the dimension of matrices remains unchanged. Therefore, a network shrinking algorithm might be proposed in the future work.

The application of the developed methods on specific types of data might lead to understanding previously insolvable problems.

Bibliography

[1] Frank Rosenblatt. “The perceptron: A probabilistic model for information storage and organization in the brain”. In: *Psychological Review* 65 (1958), pp. 386–408.

[2] Peter Bräunig and Reinhold Hustert. “Proprioceptors with central cell bodies in insects”. In: *Nature* (1980), pp. 768–770.

[3] 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.

[4] C. Cortes and V. Vapnik. “Support-vector networks”. In: *Machine Learning* 20 (1995), pp. 273–297.

[5] Tin Kam Ho. “Random Decision Forests”. In: *Document Analysis and Recognition, 1995., Proceedings of the Third International Conference on* 1 (1995), pp. 278–282.

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

[7] D. G. Lowe. “Distinctive image features from scale-invariant keypoints”. In: *International Journal Computer Vision* 60 (2004), pp. 91–110.

[8] H. Bay, T. Tuytelaars, and L. V. Gool. “Surf: Speeded up robust features”. In: *ECCV* (2006), pp. 404–417.

[9] John R. Meyer. *Tactile Communication*. [Online; accessed 25-May-2016; General Entomology, ENT 425]. 2006. URL: \url{https://www.cals.ncsu.edu/course/ent425/tutorial/Communication/tactcomm.html}.

[10] Georg Martius et al. *Robot Simulator of the Robotics Group for Self-Organization of Control*. http://robot.informatik.uni-leipzig.de/software/. last modified: 06. July 2015. 2009.

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

[12] 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.

[13] 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.

[14] Poramate Manoonpong. *Open-source multi sensori-motor robotic platform AMOS II*. http://manoonpong.com/AMOSII.html. 2011.

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

[16] 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.

[17] 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.

[18] Kaggle. *The Marinexplore and Cornell University Whale Detection Challenge*. [Online; accessed 26-May-2016]. 2013. URL: \url{http://www.lauradhamilton.com/10-surprising-machine-learning-applications}.

[19] 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.

[20] 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.

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

[22] 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.

[23] 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/}.

[24] Poramate Manoonpong. "Adaptive Embodied Locomotion Control Systems". Lecutre 3 - page 133 - Tripod Gait.

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

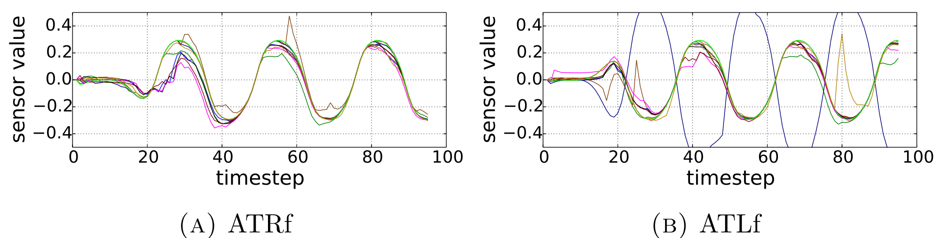
Appendix A1

Supplementary Data

This section contains data and results, which complement the information presented in previous sections.

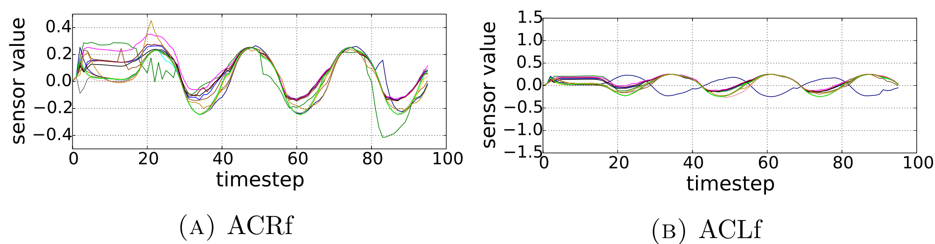
A1.1 Sensory Data Examples

The following figures show signal examples for all 24 sensors (see Table 3.1).



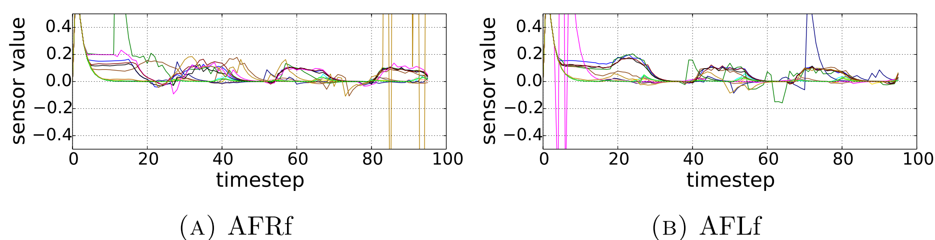
(A) ATRf (B) ATLf

Figure A1.1: Thoraco proprioceptive sensors, front legs



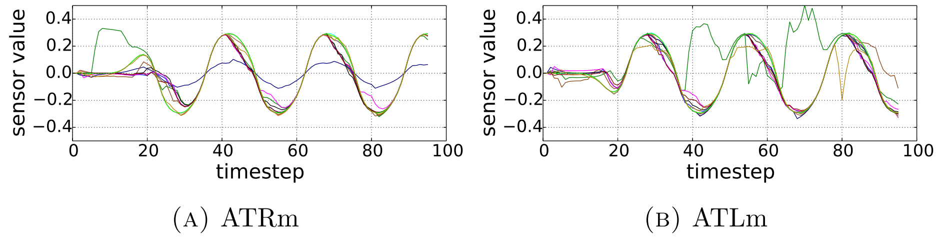
(A) ACRf (B) ACLf

Figure A1.2: Coxa proprioceptive sensors, front legs



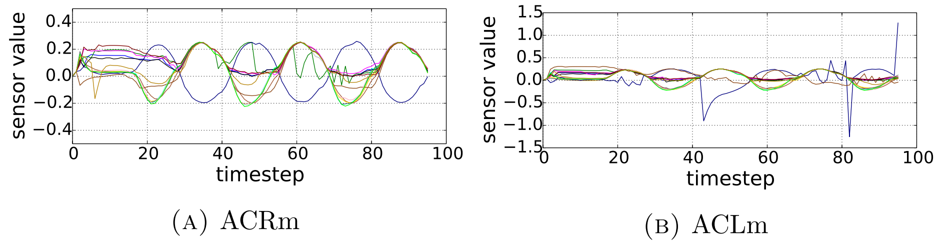
(A) AFRf (B) AFLf

Figure A1.3: Femur proprioceptive sensors, front legs



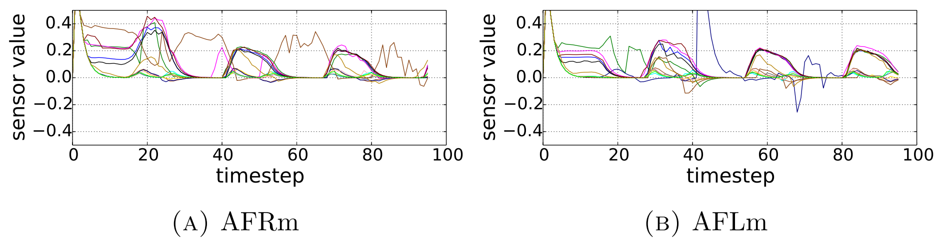
(A) ATRm (B) ATLm

Figure A1.4: Thoraco proprioceptive sensors, middle legs



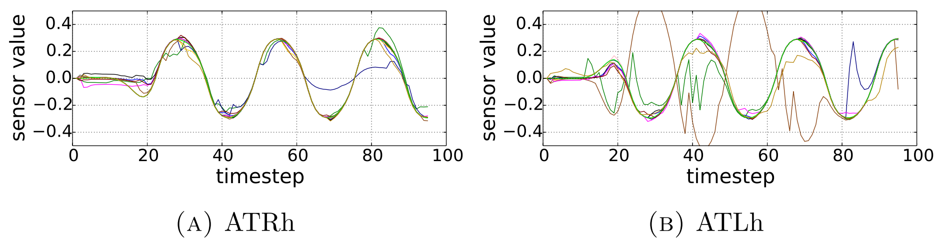
(A) ACRm (B) ACLm

Figure A1.5: Coxa proprioceptive sensors, middle legs



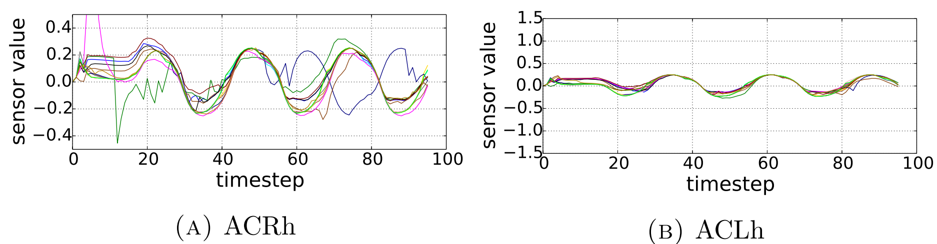
(A) AFRm (B) AFLm

Figure A1.6: Femur proprioceptive sensors, middle legs



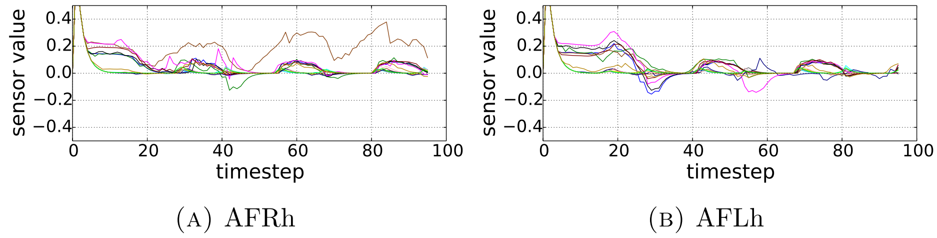
(A) ATRh (B) ATLh

Figure A1.7: Thoraco proprioceptive sensors, hint legs



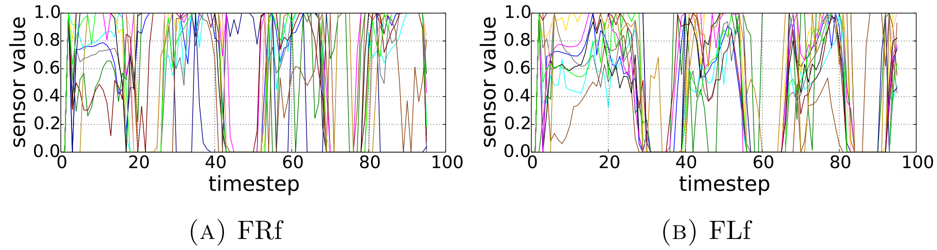
(A) ACRh (B) ACLh

Figure A1.8: Coxa proprioceptive sensors, hint legs



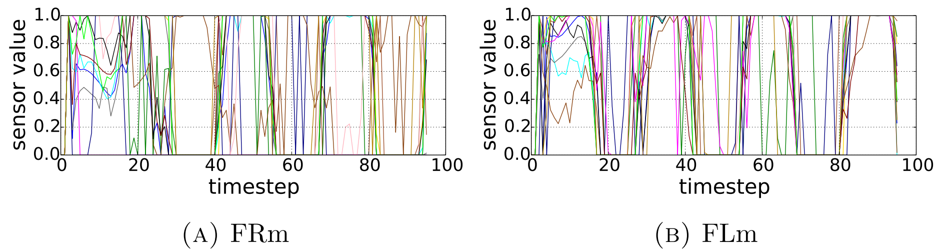
(A) AFRh (B) AFLh

Figure A1.9: Femur proprioceptive sensors, hint legs



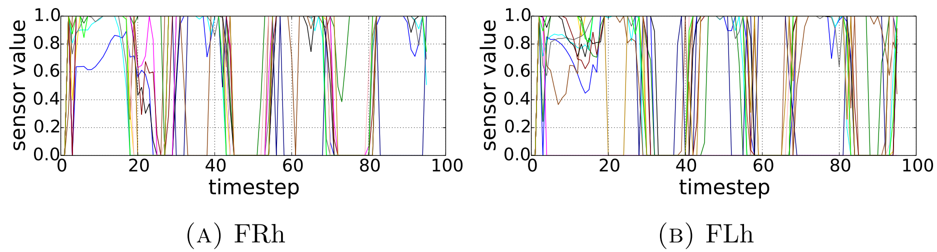
(A) FRf (B) FLf

Figure A1.10: Tactile sensors, front legs



(A) FRm (B) FLm

Figure A1.11: Tactile sensors, middle legs



(A) FRh (B) FLh

Figure A1.12: Tactile sensors, hint legs

A1.2 Generated Datasets

The following Table A1.1 presents all generated datasets (see section 3.6) used for terrain classification analysis (in section 4.3).

TABLE A1.1: Generated datasets

name	*ter. noise*	*sig. noise*	*timesteps*	*sensors*
00_00_80_p	0.00	0.00	80	proprioceptive
00_00_80_a	0.00	0.00	80	all
00_03_40_a	0.00	0.03	40	all
00_00_30_a	0.00	0.00	30	all
00_00_10_p	0.00	0.00	10	proprioceptive
00_01_40_a	0.00	0.01	40	all
00_00_01_a	0.00	0.00	01	all
00_00_80_t	0.00	0.00	80	tactile
00_00_20_a	0.00	0.00	20	all
00_05_40_a	0.00	0.05	40	all
00_00_40_a	0.00	0.00	40	all
00_00_10_a	0.00	0.00	10	all
00_00_40_p	0.00	0.00	40	proprioceptive
00_00_10_t	0.00	0.00	10	tactile
00_10_40_a	0.00	0.10	40	all
00_00_40_t	0.00	0.00	40	tactile
01_03_40_a	0.01	0.03	40	all
01_10_40_a	0.01	0.10	40	all
01_01_40_a	0.01	0.01	40	all
01_00_40_a	0.01	0.00	40	all
01_05_40_a	0.01	0.05	40	all
03_10_40_a	0.03	0.10	40	all
03_05_40_a	0.03	0.05	40	all
03_00_40_a	0.03	0.00	40	all
03_03_40_a	0.03	0.03	40	all
03_01_40_a	0.03	0.01	40	all
05_01_40_a	0.05	0.01	40	all
05_05_40_a	0.05	0.05	40	all
05_10_40_a	0.05	0.10	40	all
05_03_40_a	0.05	0.03	40	all
05_00_40_a	0.05	0.00	40	all
10_10_40_a	0.10	0.10	40	all
10_03_40_a	0.10	0.03	40	all
10_01_40_a	0.10	0.01	40	all
10_00_40_a	0.10	0.00	40	all
10_05_40_a	0.10	0.05	40	all
20_01_40_a	0.20	0.01	40	all
20_00_40_a	0.20	0.00	40	all
20_05_40_a	0.20	0.05	40	all
20_10_40_a	0.20	0.10	40	all
20_03_40_a	0.20	0.03	40	all

A1.3 Supplementary Figures for Feature Analysis

Figures from this section complements the analysis of feature selection from section 4.4.1.

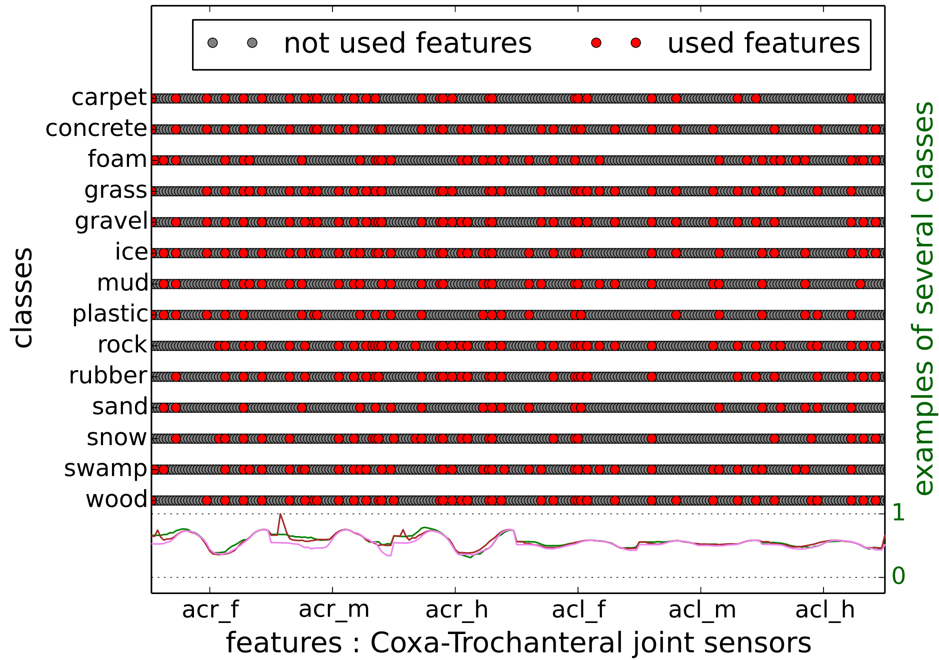


FIGURE A1.13: Used features of coxa-trochanteral proprioceptive sensors for individual classes.

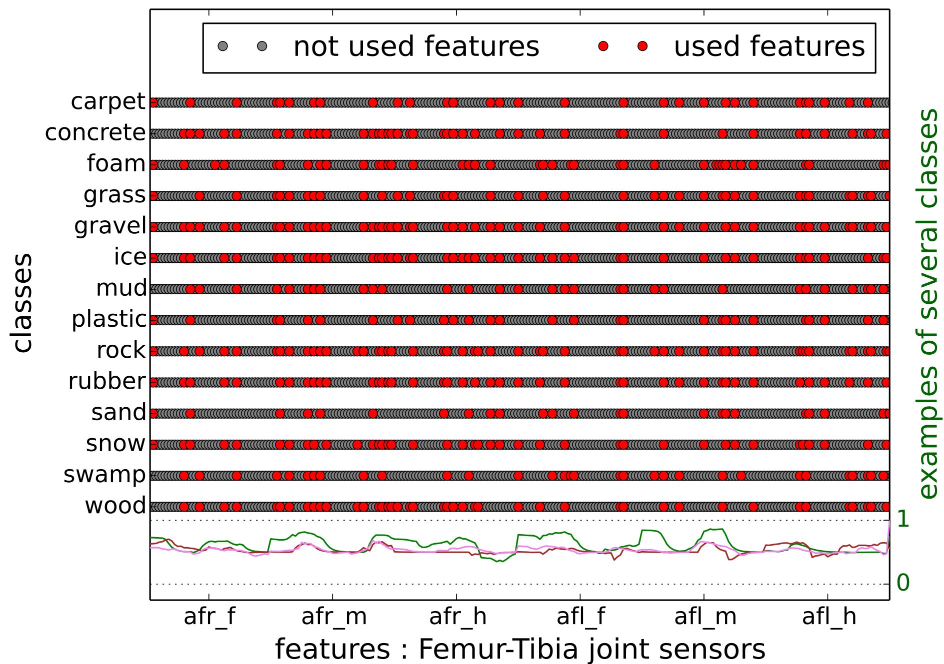


FIGURE A1.14: Used features of femur-tibia proprioceptive sensors for individual classes.

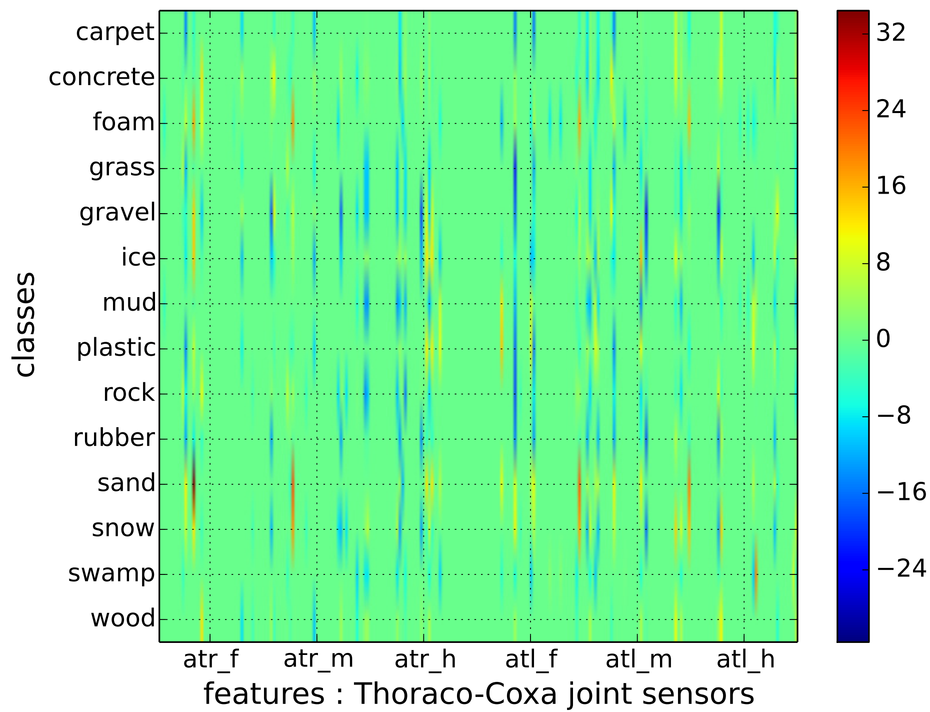


FIGURE A1.15: Influence power of single features on classes: thoraco-coxa sensors

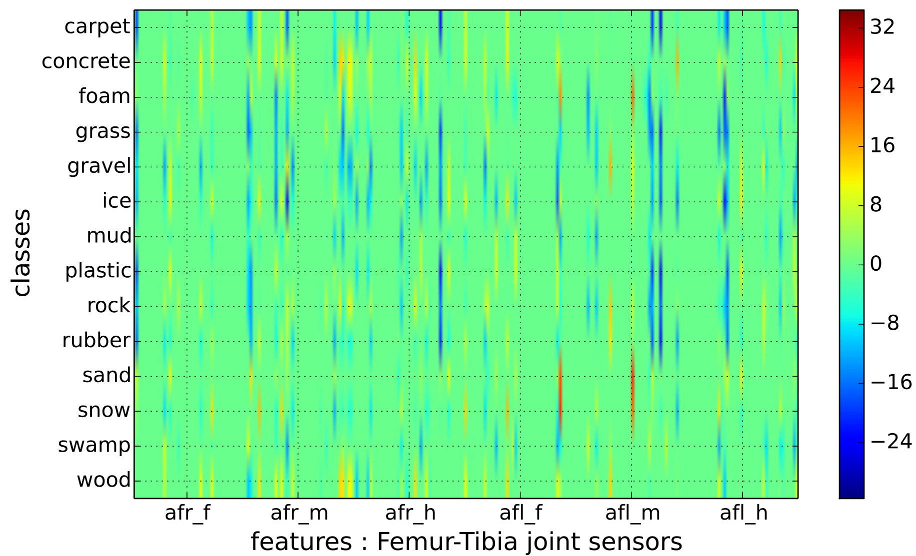


FIGURE A1.16: Influence power of single features on classes: femur-tibia sensors

Appendix A2

Method Implementation

In this section, more detailed implementation information, related to the new framework (appendix A2.1) and to the terrain classification process (appendix A2.2), is provided.

A2.1 Implementation of the Neural Network

This section relates to chapter 2. The framework was implemented in programming language Python and named *KITTNN*. A detailed API for the presented classes is attached in appendix A4.1.

The following diagram (A2.1) shows the structure of the *KITTNN* .py package.

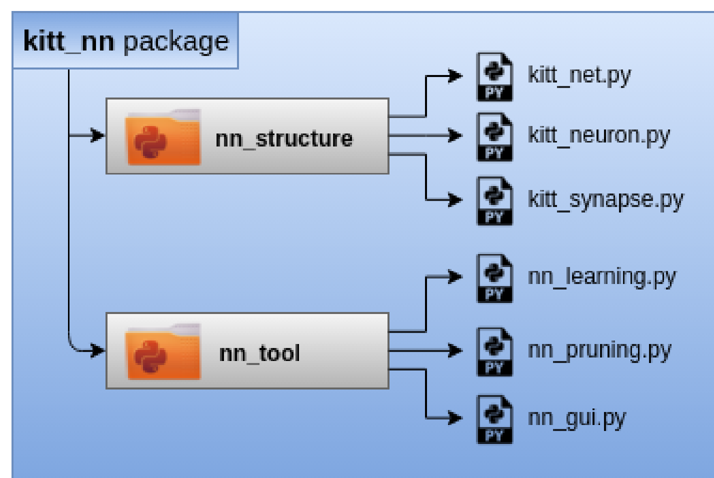


FIGURE A2.1: *KITTNN* package : Implemented neural network framework

The *KITTNN* 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).

kitt_neuron.py

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

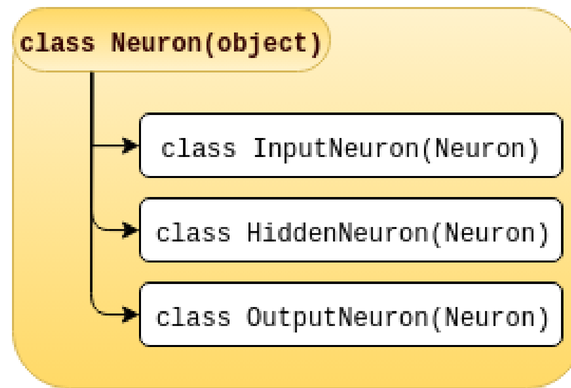


FIGURE A2.2: kitt_neuron.py : Neuron class inheritance

Then, as Fig. A2.2 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 any more.

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. A2.3. Basically, the only parameter is the network structure, which is expected as a *.py iterable* type.

For instance, a network with 2 input, 5 hidden and 3 output units would be created as *NeuralNetwork(structure=[2, 5, 3])*. Number of hidden layers is not limited.

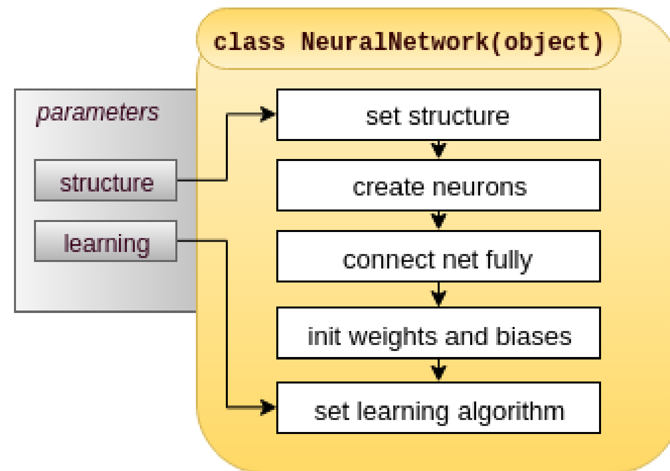


FIGURE A2.3: kitt_net.py : Neural Network Initialization

A learning algorithm is added to the initialized network thereafter (see section 2.3). 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 2.5, section 2.4).

Scikit-learn Neural Network Library

In order to verify the functionality of the implemented neural network framework (*KITTNN*), 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 A2.1.

PART OF CODE A2.1: SKNN classifier specification (Champandard and Samothrakis, 2015)

```
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)
```

A2.2 Implementation of the Terrain Classification

This section relates to chapter 3. The following Fig. A2.4 illustrates the overall terrain classification process and presents an extended version of the diagram in Fig. 3.1.

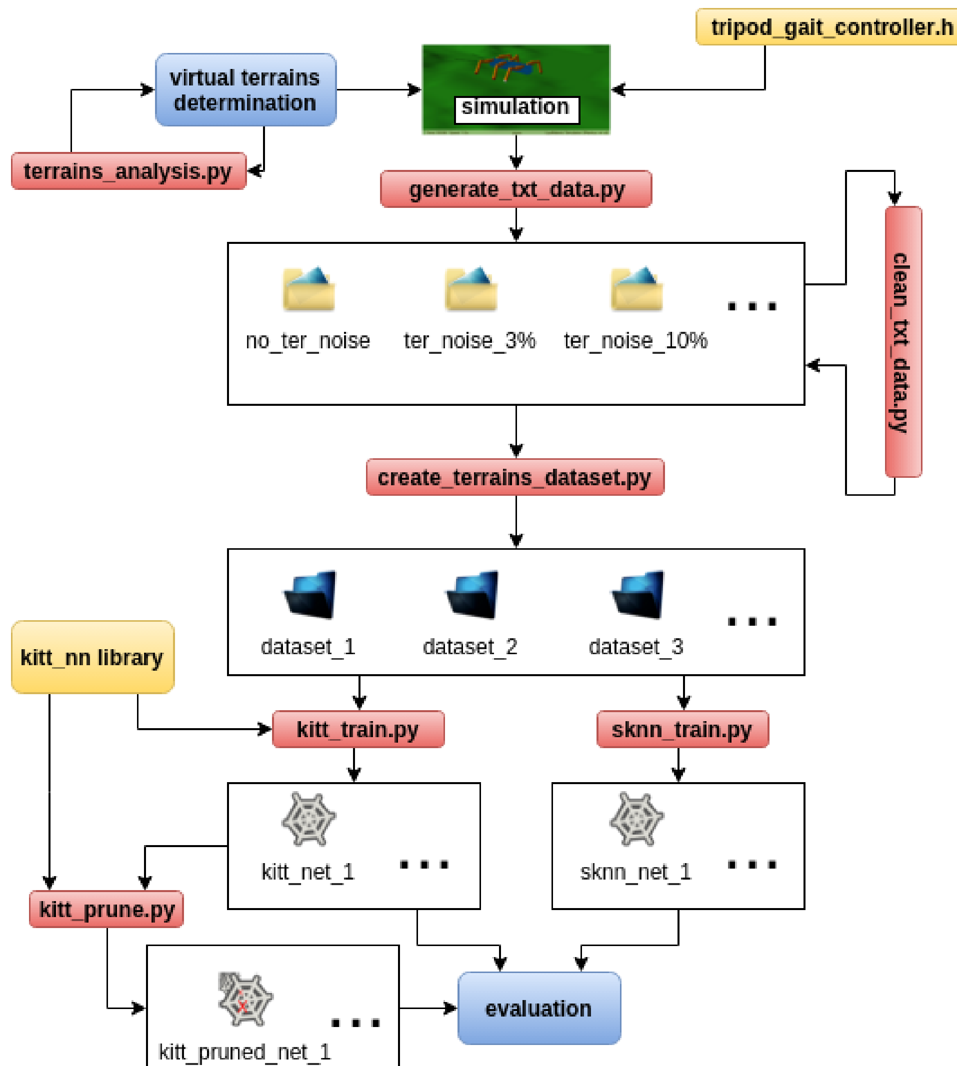


FIGURE A2.4: Terrain classification process - overall diagram.

In the following sections, the individual steps of the terrain classification process are explained in more detail.

LPZ Robots Simulation

This section extends the information from section 3.2.2. It is devoted to the simulation of AMOS II using (Martius et al., 2009)

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. A2.5.

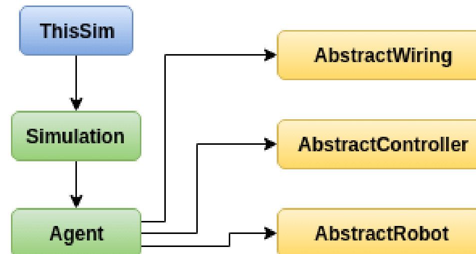


FIGURE A2.5: Software architecture for LPZRobots and GoRobots. (Martius et al., 2009)

To introduce the elements in Fig. A2.5, *ThisSim* is an inherited class of another class called *Simulation* and is initialized every time 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 A4), a *'rough terrain'* substance is being initialized and passed through a handle to a *TerrainGround* constructor.

PART OF CODE A2.2: 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 A2.3 (with the reference to sensors abbreviations in Table 3.1).

PART OF CODE A2.3: 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 A4).

All the data files are generated by a script called *generate_txt_data.py* (A4). 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/*.

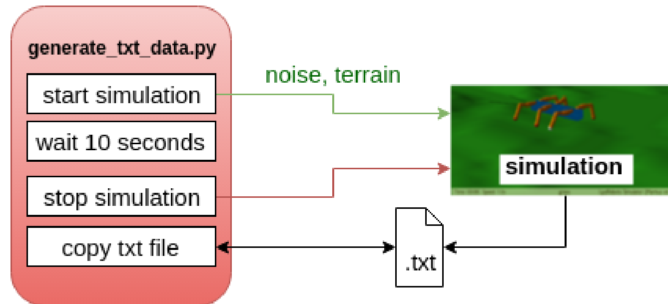


FIGURE A2.6: The process of data acquisition from the simulation.

In this manner, *.txt* files for all terrains and all mentioned σ_p are saved into a structure illustrated on Fig. A2.7. Each *.txt* file contains approximately 100 lines, one for each simulation step (as shown in code part A2.3). Every line then contains values of the 24 proprioceptive sensors.

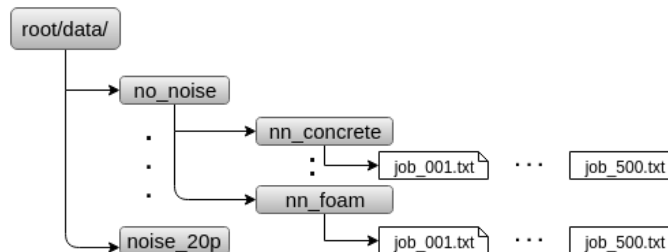


FIGURE A2.7: The structure of rough data directory.

Right after the data generation, a script called *clean_txt_data.py* (A4) 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 section 3.6, 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 workflow of this script is illustrated in Fig. A2.8.

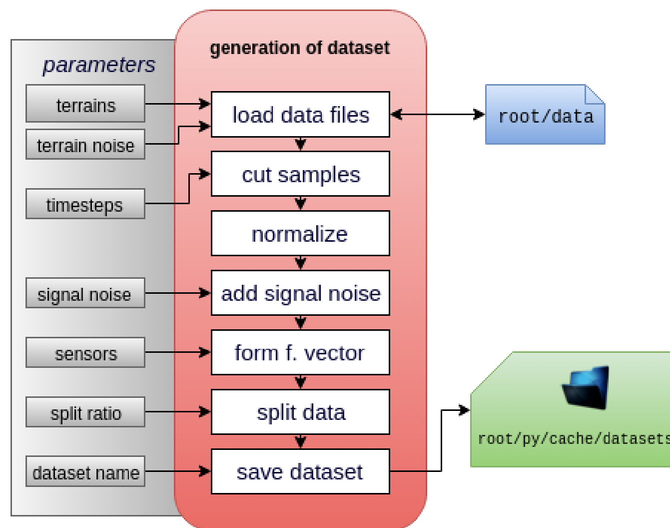


FIGURE A2.8: Workflow of generating a dataset

The datasets files are saved in directory `root/py/cache/datasets/amos_terrains_sim/` (see A4). 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.

Appendix A3

Structure of the Workspace

```
root
├─ simulation
│  ├─ gorobots_edu-fork
│  └─ lpzrobots-fork
├─ data
│  ├─ no_noise
│  ├─ noise_1p
│  ├─ noise_3p
│  ├─ noise_5p
│  ├─ noise_10p
│  └─ noise_20p
├─ py
│  ├─ cache
│  │  ├─ params
│  │  ├─ downloads
│  │  ├─ datasets
│  │  ├─ trained
│  │  └─ pruned
│  ├─ kitt_nn
│  └─ scripts
├─ results
├─ progress_reports
└─ thesis
```

Appendix A4

Code Documentation

A documentation for the *KITTNN* framework implementation is provided in appendix A4.1. The API for the implementation of the terrain classification process is in appendix A4.2.

A4.1 Neural Network Framework KITTNN (API)

class kitt_nn.nn_structure.kitt_net.**NeuralNetwork(structure)**

The main class representing an artificial neural network.

@ structure (array-like) : len: number of layers; items: number of neurons per layer

def init : Creates neurons and makes a fully-connected structure.

def fit : Feeds the input with samples and trains the model.
- @ train_X : array-like, shape (n_samples, n_inputs)
- @ train_y : array-like, shape (n_samples, n_outputs)
- @ val_X : array-like, shape (n_samples, n_inputs)
- @ val_y : array-like, shape (n_samples, n_outputs)

def predict : Predicts the output.
- @ test_X : array-like, shape (n_samples, n_inputs)
- returns y_pred : array, shape (n_samples, n_outputs)

def copy : Creates a copy of self.
- returns net_copy : kitt_net.NeuralNetwork

class kitt_nn.nn_structure.kitt_neuron.**Neuron(net,layer,id)**

The class representing a single neuron unit.

@ net (kitt_nn.NeuralNetwork) : mother network

@ layer (int) : mother's layer id in the network

@ id (int) : position in the layer

def activate : Activates the neuron axon by the transfer function.

def set_bias : Sets the bias value from the bias matrix hold by the network.

def get_bias : Returns current bias value.
- returns b : float

def set_dead : Removes the neuron from the net.

`class kitt_nn.nn_structure.kitt_synapse.`**`Synapse(net,from,to)`**

The class representing a single synapse.

- @ **net** (kitt_nn.NeuralNetwork) : mother network
- @ **from** (kitt_nn.Neuron) : Neuron, where the synapse comes from
- @ **to** (kitt_nn.Neuron) : Neuron, where the synapse goes to

def set_weight : Sets the weight value from the weight matrix hold by the network.

def get_weight : Returns current weight value.
returns w : float

def remove_self : Removes the synapse from the net.

`class kitt_nn.nn_tool.nn_learning.`**`BackPropagation(net)`**

The class representing the backpropagation learning algorithm.

- @ **net** (kitt_nn.NeuralNetwork) : mother network

def train : Trains the mother network.
@ train_data : array-like, shape (n_samples, 2) [X, y]
@ val_data : array-like, shape (n_samples, 2) [X, y]

def try_to_train : Tries to retrain the mother network.
@ train_data : array-like, shape (n_samples, 2) [X, y]
@ val_data : array-like, shape (n_samples, 2) [X, y]
@ req_accuracy : float
returns retrained : bool

A4.2 Terrain Classification Scripts (API)

`~/py/scripts/`**`generate_txt_data.py`**

Runs the Amos II simulation and saves sensory data as .txt files.

@ nj (–n_jobs) : Number of simulation runs.
- type : int
- choices : [1, 2, ... 1000]
- default : 500

@ t (–terrains) : Terrain (id) to be generated.
- type : int
- choices : [1, 2, ... 14]
- default : [1, 2, ... 14]

@ n (–noise) : Terrain noise level to be generated.
- type : string
- choices : ['nn', 'n1p', 'n3p', 'n5p', 'n10p', 'n20p']
- default : 'nn'

@ nt (–n_timesteps) : Number of simulation timesteps.
- type : int
- choices : [1, 2, ... 1000]
- default : 100

`~/py/scripts/`**`clean_txt_data.py`**

Checks generated .txt files and remove bad/incomplete ones.

@ t (–terrains) : Terrain (id) to be checked.
- type : int
- choices : [1, 2, ... 14]
- default : [1, 2, ... 14]

@ n (–noise) : Terrain noise levels to be checked.
- type : string
- choices : ['nn', 'n1p', 'n3p', 'n5p', 'n10p', 'n20p']
- default : ['nn', 'n1p', 'n3p', 'n5p', 'n10p', 'n20p']

@ sl (–sample_len) : Minimum required sample length.
- type : int
- choices : [1, 2, ... 1000]
- default : 95

`~/py/scripts/`**`create_terrain_dataset.py`**

Creates a dataset out of the cleaned .txt data files and saves it using *cPickle*.

@ rt (–rem_terrains) : Terrain (id) to be avoided from the dataset.
- type : int
- choices : [1, 2, ... 14]
- default : []

@ tn (–terrain_noise) : Terrain noise level.
- type : string
- choices : ['nn', 'n1p', 'n3p', 'n5p', 'n10p', 'n20p']
- default : 'nn'

@ sn (–signal_noise) : Signal noise level.
- type : float
- choices : [0.0, 0.01, 0.03, 0.05, 0.1]
- default : 0.0

@ s (–sensors) : Sensors to be included.
- type : string
- choices : [atr_f, atr_m, ..., fl_h]
- default : [atr_f, atr_m, ..., fl_h]

@ ts (–timesteps) : Number of timesteps per sensor.
- type : int
- choices : [1, 2, ... 80]
- default : 40

@ ds (–data_split) : Ratio for train/val/test splitting.
- type : array-like of int ,s.t. sum = 1
- choices : [0.0, 0.01, ..., 0.99]
- default : [0.7, 0.1, 0.2]

@ ns (–n_samples) : Number of samples per terrain.
- type : int
- choices : [0, 1, ..., 500]
- default : 500

~/py/scripts/kitt_train.py

Trains a *kitt_nn* neural network on the given dataset and saves it using *cPickle*.

@ **ds** (–dataset) : Dataset file name.
- type : string
- choices : any string
- default : ”

@ **s** (–structure) : Hidden structure of the neural network.
- type : array-like of int
- choices : [1, 2, ..., 1000]
- default : [20]

@ **lr** (–learning_rate) : Learning rate for network training.
- type : float
- choices : [0.01, 0.02, ..., 1.0]
- default : 0.03

@ **ni** (–n_iter) : Number of training epochs.
- type : int
- choices : [1, 2, ..., 1000]
- default : 500

~/py/scripts/kitt_prune.py

Prunes the given trained network and finds a minimal structure of it with respect to the given dataset.

@ **n** (–net) : Trained network file name.
- type : string
- choices : any string
- default : ”

@ **ra** (–req_accuracy) : Required classification accuracy of the pruned network.
- type : float
- choices : [0.0, 0.01, 1.0]
- default : [0.99]

@ **lr** (–learning_rate) : Learning rate for network re-training.
- type : float
- choices : [0.01, 0.02, ..., 1.0]
- default : 0.03

@ **mi** (–max_iter) : Maximum number of re-training epochs.
- type : int
- choices : [1, 2, ..., 1000]
- default : 100

@ **ns** (–n_stable) : Number of stable epochs for termination.
- type : int
- choices : [1, 2, ..., 1000]
- default : 15

List of Supplementary Scripts

`create_mnist_dataset.py`, `create_xor_dataset.py`, `list_generated_datasets.py`, `plot_cl_results.py`, `plot_feature_selection.py`, `plot_grid_search.py`, `plot_nn_verification.py`, `plot_pa_results.py`, `plot_sample.py`, `plot_sensor.py`, `plot_transfer_functions.py`, `reed_prune.py`, `set_and_store_params.py`, `sknn_train.py`, `terrain_analysis.py`, `zero_prune.py`