DELIVERABLE REPORT

Project acronym: INPUT
Project number: 687795

D6.3, Report on First Machine Learning System Implemented on Microcontroller

Dissemination type:	DEM
Dissemination level:	PP
Planned delivery date:	2018-04-30
Actual delivery date:	2017-06-02
Reporting Period:	2

WP6, Task 3, Implementation on Portable Hardware (lead: SUPSI-IDSIA)

1 INTRODUCTION

This report describes the implementation of the INPUT machine learning system on portable hardware (i.e. a microcontroller), as specified by the project plan (Task 6.3). This deliverable allows to begin the main phase of patient testing (consider deliverable 6.4, i.e. the actual implementation); the remainder of tasks 6.3 and 6.4 deal with further improving the implementation according to the needs identified during the patient tests.

In order to bootstrap the development, OBHP provided to SUPSI-IDSIA a suitable microcontroller together with an initial implementation of a machine learning based prosthesis control algorithm. This initial system applies Linear Discriminant Analysis (LDA) for movement classification, additionally enabling proportional force/velocity control using the EMG amplitude [1]. It does not allow the simultaneous control of multiple degrees of freedom. The implementation of this task therefore consisted in replacing the (hardcoded) LDA classifier with a feedforward neural net implementation (following the results of task 6.2), thus implementing the proportional control of simultaneous movements. The parameters for the trained neural net are uploaded to the microcontroller via a dedicated user interface. Furthermore, the implementation included a set of postprocessing methods, again following the results from task 6.2.

This report is structured as follows: The parameters of the deliverable are summarized in sections 2 and 3. In section 4, we give an overview of the entire microcontroller-based prosthesis control system in its current state. Section 5 describes how the system is used in practice, and displays examples from our own tests and experiments. Section 7 extracts the concrete contribution of SUPSI-IDSIA to the implementation.

2 DESCRIPTION OF THE TASK

> "During this stage, suitable machine learning-based classification and regression methods, as determined in Task 6.2, will be implemented on a microcontroller integrated into a wearable prosthesis, and evaluated in a series of end-user tests (WP9). For local tests without end-users, OBHP will provide SUPSI-IDSIA with test hardware.

For this implementation, the quality of the recognizer and the hardware constraints have to be carefully balanced, furthermore observations obtained during the user tests will be integrated into the development loop. We note that we expect even relatively complex neural network based processing (see task 6.2) to be feasible on a microcontroller, as soon as the training process has been completed.

As with task 6.2, the main work for this task is to be finished at the end of year two. The remainder of this task will consist of an integrated loop of user tests and algorithmic improvements, tightly integrated with task 6.4."

3 Description of Deliverable

"This report describes the first implementation of a machine learning system for prosthesis control on a microcontroller (in conjunction with deliverable 6.4). "

4 Implementation of the Microcontroller Software

4.1 Microcontroller Hardware and Tools

The microcontroller platform was developed by OBHP for a company internal project and was modified slightly for being used by IDSIA. This way, an optimum of efficiency and reutilization was achieved across partners. On this platform, OBHP developed software to allow usage of basic board functionality, such as writing and reading data to and from the memory modules, interfacing the analogue digital converters, communication interfaces, etc. Further, OBHP also implemented the final stage of hardware drivers, which allow controlling the prosthesis. Lastly, an LDA classifier was implemented by OBHP. This package (hardware and software) was handed from OBHP to IDSIA, including documentation stored on Microsoft OneNote documents. Over several video conference sessions, OBHP supported IDSIA in how to install the exact same development environment, how to read and use the pre-existing code and at which particular sections modifications needed to be inserted to implement the IDISA particular software modules. IDSIA then proceeded to include these modules, most notably the neural network and all the peripheral structures needed. These are described in detail in the sections below.

The system uses a Renesas R5F564MLDDLK#21, 32-bit microcontroller of the Renesas RX64 family. A list of hardware parameters can be found in the following table:

Clock speed	120MHz (10MHz external Oscillator with an internal PLL)
Program Flash	4MB
RAM / ROM	512KB / 4096KB
Power supply	5.5V – 12V, internal voltage regulation to 3.3V for the controller.
Power consumption	App. 22mA for the board (without electrodes)
Persistent Memory	Flash (16MB) and EEPROM (32KB)
EMG Input	8 input channels (3-pin, sampled at 9kHz with 12 bit, software dowsampling to 1.8kHz)
Prosthesis control output	UART Interface (2x 3-pin connector, 19200 b/s) and PWM Interface (2-pin connector, 40kHz)
Host Connections	UART (3-pin serial cable connection, 115200 b/s), Bluetooth (Stollmann BlueMod+SR Module)
Programming	C99 language (tools are described below)

Figure 1 shows the microcontroller (without the protective casing) as provided to IDSIA. The microcontroller interfaces directly with conventional electrodes which can be integrated into a prosthesis, as well as with the 8-electrode testing system which was provided to SUPSI-IDSIA. The latter system is likewise shown in Figure 1, it was described in detail in deliverable D6.2.

For programming and debugging, a USB-based host interface (https://www.renesas.com/en-sg/products/software-tools/tools/emulator/e1.html) and the Renesas RX compiler (https://www.renesas.com/en-eu/products/software-tools/tools/compiler-assembler/compiler-package-for-rx-family.html), together with an Eclipse-based IDE (E2 studio) were acquired by IDSIA. The

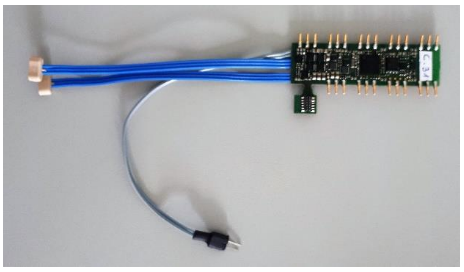
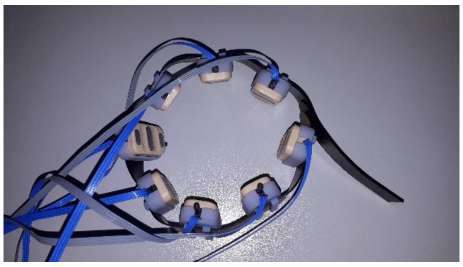



Figure 1: INPUT Microcontroller (without casing) and recording electrodes

combined system allows programming and debugging the system in the C language in a way much like writing a standard PC-based program.

4.2 Functionality Overview of the Prosthesis Controller

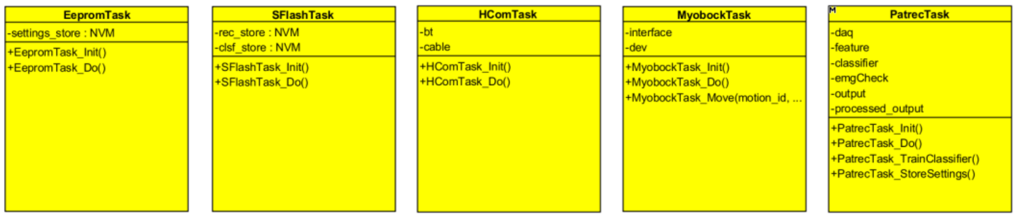
The main responsibilities and functionalities of the prosthesis controller are summarized as follows:

- Capturing EMG data, windowing, and computing EMG features
- Estimating movements and forces from each preprocessed window in real time, using any of a set of ML methods
- Translating movements / forces into control signals which are send to the prosthesis
- *Training* an underlying ML system
- Recording movements for training (above point)
- Saving data and parameters to the inbuilt persistent memory (see section 4.4)
- Communication with a host PC for data visualization, control, and parameter settings.

We remark that the neural network classifier is *not* trained on-chip, instead, the parameters of the trained neural network are uploaded from the host PC. For neural network training, the software *Inputflow* described in deliverable D6.1 remains in use, both for internal tests at IDSIA, and at the clinical partners. The other ML methods which are currently available (see section 0) are, however, trained on the microcontroller.

4.3 SYSTEM OVERVIEW

The prosthesis controller is implemented as a stand-alone program which runs without an operating system. This means that all hardware management and task scheduling is done internally within the controller. Since the C language is used, the system does not *formally* use classes or objects, however, the internal structure does follow a structured approach in which blocks of functionality and related data are grouped into logical units which communicate via dedicated interfaces, much like in object-oriented programming. We therefore use UML class diagrams to describe the structure of the system.



Usually, there is one header file (*.h) and one source code file (*.c) for each “class”; polymorphism (e.g. for the postprocessors) is implemented manually via explicit function pointers. We finally note that almost all data structures are allocated statically in order to avoid the overhead of dynamic memory allocation, following standards practices in microcontroller programming. Thus, normally there is always exactly one “object” of each class (which may, of course, be in active use, or may be disabled).

Fehler! Verweisquelle konnte nicht gefunden werden. shows the “task” objects of the prosthesis controller. These objects form the foundation of the system; they can be thought to represent separate tasks which are required for the working of the entire system. Each task is initialized at startup, calling the *_Init functions, after which the tasks are sequentially processed in an infinite loop, for which the respective *_Do functions are called (see Figure 2).

The tasks have got the following functions:

- `EepromTask`: Manages the EEPROM persistent memory.
- `SFlashTask`: Manages the Flash persistent memory.

Both these tasks interface with an underlying “library” which implements a partition structure and a file system; they export their functionality in form of NVM (non-volatile memory) objects which have functions to save and load arbitrary data, see section 4.4 for details on how persistent memory is used.

- `HComTask`: Controls the communication between the host and the microcontroller, using either Bluetooth or a serial (UART) cable connection. Control commands (e.g. for setting classification parameters) are received from the host, control answers and data dumps are sent to the host. Required variables to access the Bluetooth or UART subsystems are stored in the `bt` and `cable` fields, respectively.
- `MyobockTask`: Controls the attached prosthesis. Hardware parameters and capabilities are contained in the `dev` field.
- `PatrecTask`: Comprises the computation of EMG features, polymorphically calling the classifier and the postprocessors as desired, and translating movements into commands for the `MyobockTask` using a configurable conversion module (`motconv`, not shown). Furthermore, data recording for on-chip training is controlled here as well.

The functionality which is relevant for the INPUT project is mostly part of the `PatrecTask`, whose components are described in detail in section 0.

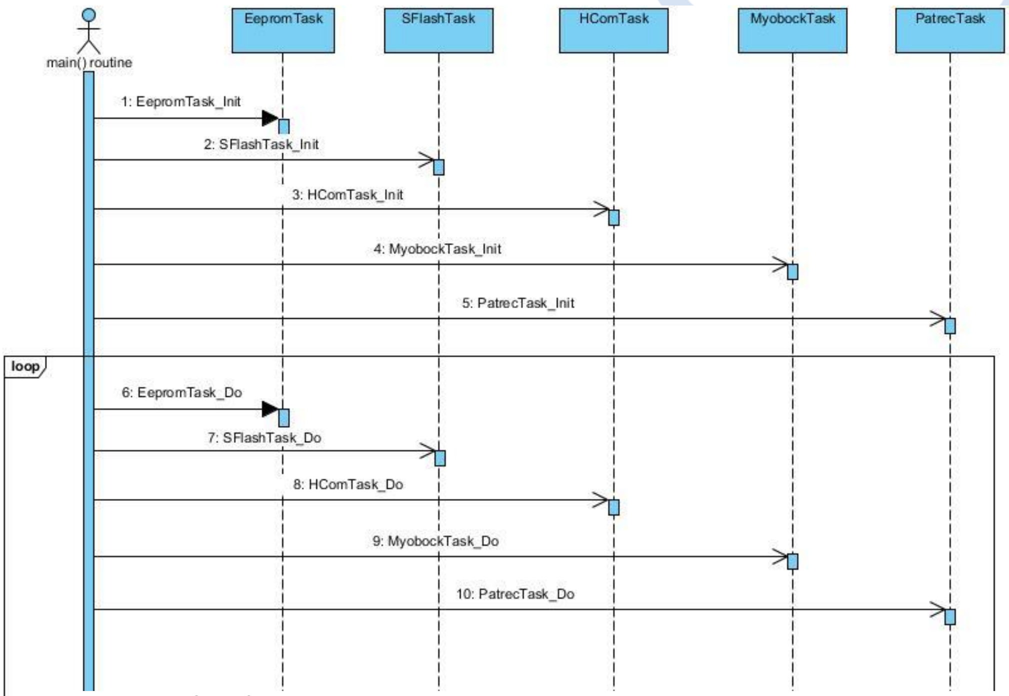


Figure 2: Control flow of the Prosthesis Controller

4.4 Persistent Memory

The persistent memory of the INPUT microcontroller is used for permanently storing the classifier parameters, diverse settings (including the set of possible movements), and also for storing recorded data for classifier training (not in use for the neural network controller). The system contains two memory components (EEPROM and Flash), where the majority of data and parameters are stored on the much larger Flash.

On each of those memory components, a partition table and file systems are created. The current implementation uses three partitions: A *settings* partition which contains general settings of the system (e.g. which classifier should be used), a *classifier* (`clsf`) partition where the parameters of the trained classifiers are stored, and a *records* partition for storing recorded training data. In particular, the settings of the neural network classifier and of the postprocessors are stored in the classifier partition.

Data storage is uniformly accessed with the *NVM* interface, which contains functions for storing, loading, and deleting files indexed by integer IDs. Each file is stored with a CRC checksum to detect storage errors. Since SUPSI-IDSIA did not perform any changes to the persistent storage interface, we refrain from giving a detailed description.

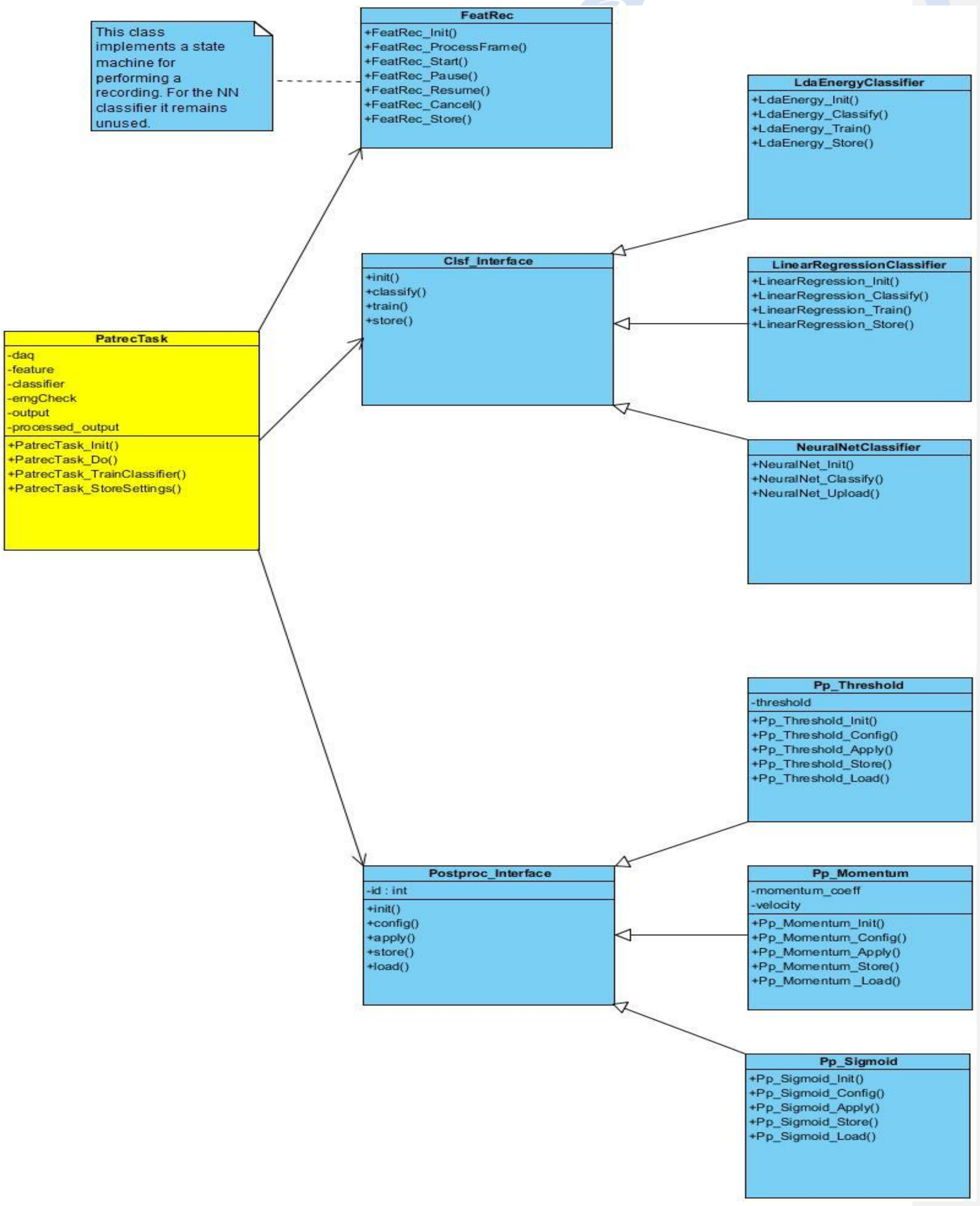


Figure 3: Classifier and Postprocessing Implementation as parts of the PatrecTask (see text for explanation)

4.5 Machine Learning Implementation

Figure 3 gives a closer look on the components which make up the actual machine learning system contained in the prosthesis controller software. As mentioned above, the machine learning functionality is largely controlled and invoked by the `PatrecTask` module.

The most important subfields of the `PatRecTask` object are as follows: `daq` contains information related to the data acquisition subsystem, which is interrupt-based, gets initialized at startup, and fills a data buffer which is subsequently used by the `feature` subsystem to compute a set of Hudgins-style features. Currently four features (Mean Absolute Value, Waveform Length, Slope Sign Change, and Zero Crossing Rate) are supported. Following the results of D6.2, we do not support running the neural network on raw data.

The `emgCheck` module is intended to disable noisy EMG channels. We currently do not use this module since we are working on improved noise robustness methods based on the neural network system, see deliverable D6.2, section 4.5. Such methods will be implemented both on the host system and (as needed) on the microcontroller during the remaining time of the INPUT project.

The classifier[1] is polymorphically called via the `Cslf_Interface` structure, which contains the operations `init`(), `classify`(), `train`(), and `store`(). In the case of the neural network classifier, training on-chip is not possible, instead the trained system must be uploaded to the microcontroller as described in sections 4.6 and 5.

The classifiers work as follows: The `LDAEnergyClassifier` performs an LDA-based movement classification and an amplitude-based force estimation following [1]; it is the only classifier which was originally included in the software delivered to SUPSI-IDSIA. The `LinearRegressionClassifier` computes a linear estimator for the movements and forces using the standard OLS method; It was chosen to be implemented as an intermediate step between LDA and the neural network, for its ease of implementation, to test simultaneous proportional prosthesis control and the post-processing units for simultaneous control methods, in preparation for the neural network implementation. Both LDA and linear regression are trained on data recorded with the microcontroller, which for this purpose must be connected to a controlling host PC.

Finally, the `NeuralNetworkClassifier` performs a regression using a pretrained fully-connected feedforward neural network. The number and shape of its layers, as well as the nonlinearities, are set when the network is uploaded to the microcontroller and are freely configurable (within limits given by the capabilities of the hardware). The classifier output is saved in the variable `output` (essentially a list of estimated forces per movement).

Finally, postprocessors allow processing the output of the regressor as desired. Just as for the PC-based tool *Inputflow* described in deliverable D6.1, they can be freely configured to run in any desired order. Currently, we have implemented the postprocessors `momentum` (adds a momentum term to the estimated forces, thus smoothing the output), `threshold` (suppresses low-amplitude movements), and `sigmoid` (better control of small movements). The output of postprocessing is found in the variable `processed_output`.

[1] The term *classifier* is used for consistency with the previous implementation. The output of each classifier is a list of estimated force parameters per movement, suitable for proportional and simultaneous control.

4.6 Host Communication and Network Upload

Communication with the host PC is comprised in the `HComTask`. At a high level, communication is a bidirectional process which consists of sending and receiving *data frames* which contain up to 200 bytes of arbitrary data, plus a checksum byte to avoid communication errors. Communication can run over a UART serial cable, connected to the host via USB, or a Bluetooth connection. From the host PC, *user commands* are sent to the microcontroller, which can be used to set up classifier and postprocessor, to initiate data recording and on-chip training, to control the attached prosthesis, and to obtain information. Answer frames are sent back to the host, containing either the desired information or an acknowledgement signal (or an error code). Finally, the microcontroller can send *dump* information (a continuous flow of configurable, predefined data) to the host, which allows the user to supervise the system in real time. Section 5 describes how this works from the user's perspective.

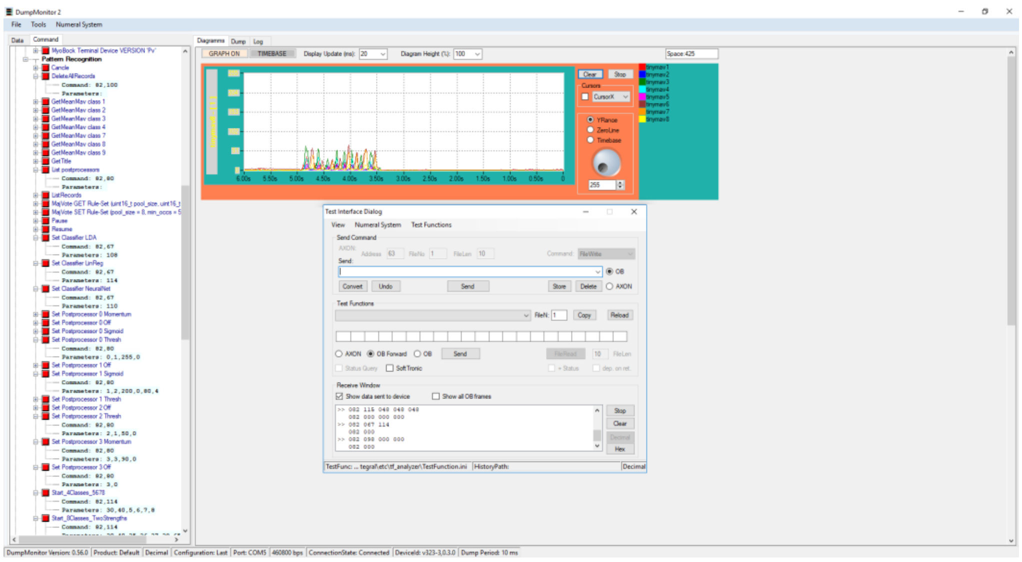
Based on the original software, we have introduced a series of additional user commands to allow flexible control of the classifier and postprocessing, and in particular, to allow uploading of the neural network parameters, which due to the size limit of each frame requires sending a series of frames (see section 7).

5 Usage

The user interacts with the prosthesis controller using a dedicated software, the *Dump Monitor* as provided by OBHP, see Figure 4. The program as a whole has a variety of functions, here we only describe the most important ones.

First, on the right-hand side, one sees a dump of several internal variables of the system. In the image, only one graph is shown, displaying the MAV (mean absolute value) for each of the 8 channels. We typically use the MAV as a debugging method, since when a Bluetooth connection is used, it is not feasible to show raw EMG data. The graphs are freely configurable, a list of possible dump data can be obtained from the *Dump* tab on the left side (not shown).

Finally, control input is given using the control commands displayed on the left side of Figure 4. Several typical commands for initiating the data recording and the training can be seen. Each command consists of several data and parameter bytes, a click on the red square next to the command sends it to the controller. The communication with the controller is supervised with the test dialog, likewise shown in the figure: Each command and response consists of a series of a series of bytes



```
MyoBock Terminal Device VERSION 'Pv'
Pattern Recognition
    Cancle
    DeleteAllRecords
        Command: 82,100
        Parameters:
    GetMeanMav class 1
    GetMeanMav class 2
    GetMeanMav class 3
    GetMeanMav class 4
    GetMeanMav class 7
    GetMeanMav class 8
    GetMeanMav class 9
    GetTitle
    List postprocessors
        Command: 82,80
        Parameters:
    ListRecords
    MajVote GET Rule-Set (uint16_t pool_size, uint16_t
    MajVote SET Rule-Set (pool_size = 8, min_occs = 5
    Pause
    Resume
    Set Classifier LDA
        Command: 82,67
        Parameters: 108
    Set Classifier LinReg
        Command: 82,67
        Parameters: 114
    Set Classifier NeuralNet
        Command: 82,67
        Parameters: 110
    Set Postprocessor 0 Momentum
    Set Postprocessor 0 Off
    Set Postprocessor 0 Sigmoid
    Set Postprocessor 0 Thresh
        Command: 82,80
        Parameters: 0,1,255,0
    Set Postprocessor 1 Off
    Set Postprocessor 1 Sigmoid
        Command: 82,80
        Parameters: 1,2,200,0,80,4
    Set Postprocessor 1 Thresh
    Set Postprocessor 2 Off
    Set Postprocessor 2 Thresh
        Command: 82,80
        Parameters: 2,1,50,0
    Set Postprocessor 3 Momentum
        Command: 82,80
        Parameters: 3,3,90,0
    Set Postprocessor 3 Off
        Command: 82,80
        Parameters: 3,0
    Start_4Classes_5678
        Command: 82,114
        Parameters: 30,40,5,6,7,8
    Start_8Classes_TwoStrengths
        Command: 82,114
```

AXON: Address 63 FileNo 1 FileLen 10 Command: FileWrite

```
>> 082 115 048 048 048
   082 000 000 000
>> 082 067 114
   082 000
>> 082 098 000 000
   082 000
```

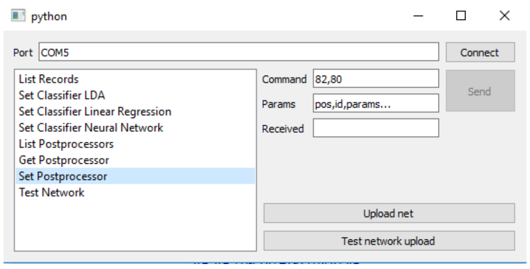
TestFunc: ... tegral\etc\tf_analyzer\TestFunction.ini | HistoryPath: | Decimal

DumpMonitor Version: 0.56.0 | Product: Default | Decimal | Configuration: Last | Port: COM5 | 460800 bps | ConnectionState: Connected | DeviceId: v323-3,0.3.0 | Dump Period: 10 ms

Figure 4: Otto Bock Dump Monitor for host-side control

(command identifier and parameters), up to a total of 200 payload bytes.

The OBHP dump monitor does currently *not* allow to send multi-part commands, which is however required for transferring neural network data (since the size of a neural network is in the range of several KB, drastically exceeding the payload of a single command frame). We therefore implemented a Python-based tool for this purpose, whose interface is shown in Figure 5. When the button "Upload Net" is clicked, a file dialog opens and allows to choose the network file, which is then uploaded in pieces following the protocol described in section 4.6. The tool also supports sending some of the most important standard commands (in particular, for setting the desired classifier), which is a convenience function for experiments. The network file has a simple proprietary format which allows easy integration with the microcontroller data structures, it is created from *Inputflow* using a novel *export* command, which was added to the Inputflow command interface described in deliverable D6.1, section 4.5. Currently it is only possible to export fully connected networks, exporting locally connected structures (convolutional nets) is currently not planned to be supported.


Port: COM5

Command: 82,80

Params: pos,id,params...

Received:

Figure 5: SUPSI-IDSIA tool for neural network data upload

6 First Experiments

In this section we describe our first experiments on validating the implementation. For this purpose, we trained a network on data recorded offline, using the testing system described in section 4.1.We recorded 48 movements (around 4 minutes of data) belonging to the seven classes with which we experimented in the past (Fine Pinch, Hand Open, Key Grip, Wrist Extension/Flexion, Wrist Pronation/Supination), 10% of which was used for early stopping of the neural network training. The neural network was set up with two hidden layers with tanh nonlinearities and 50 resp. 25 neurons, this is (on purpose) larger than the best network determined in deliverable D6.2. The size of the network is 17KB, so that it fits without problems both into the RAM and in the Flash storage of the microcontroller. Further parameters include an Adam optimizer with a learning rate of 0.001, minibatches of 64 single frames, and a (highly recommended) normalization of the input features. We note that feature normalization parameters are exported by inputflow together with the network itself.

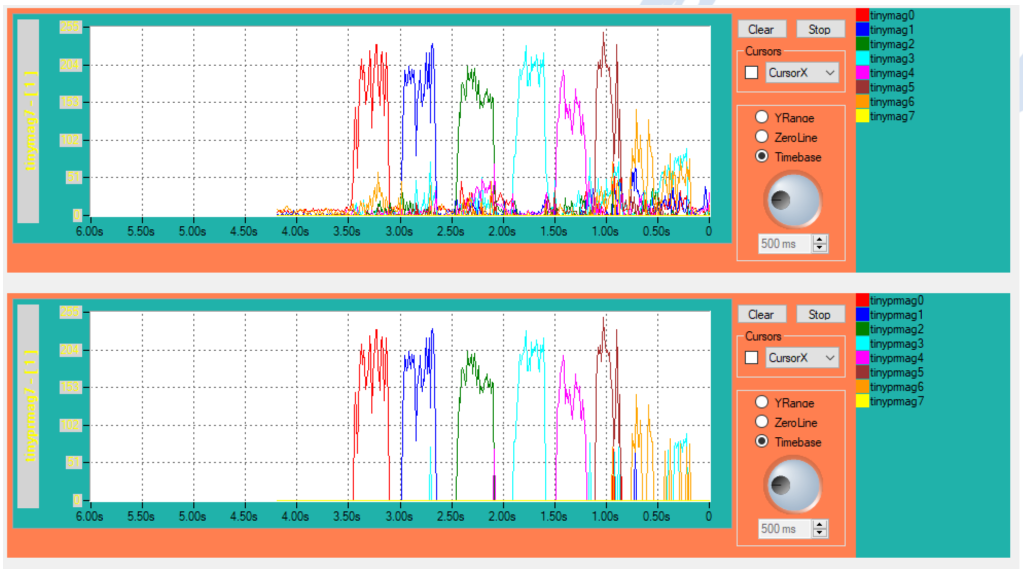


Figure 7: Dump of regression results of the Neural Network, implemented on the microcontroller, for the seven movements *Fine Pinch, Hand Open , Key Grip, Wrist Extension, Wrist Flexion, Wrist Pronation, and Wrist Supination* (in order). The lower graph shows the result of *threshold* postprocessing.

Figure 7 shows a dump of the output of the neural network, run on the microcontroller. The upper graph shows the raw output when the experimenter performed the seven movements *Fine Pinch, Hand Open, Key Grip, Wrist Extension, Wrist Flexion, Wrist Pronation, and Wrist Supination*. We note that sligh spurious movements are detected as well, which is a known problem; as mentioned in deliverable D6.2, an easy way to alleviate this problem is the application of a thresholding postprocessor, which yields the result shown in the lower part of the graph. We remark that the class *Wrist Supination* is not well recognized, which may be due to the relatively small training data set.

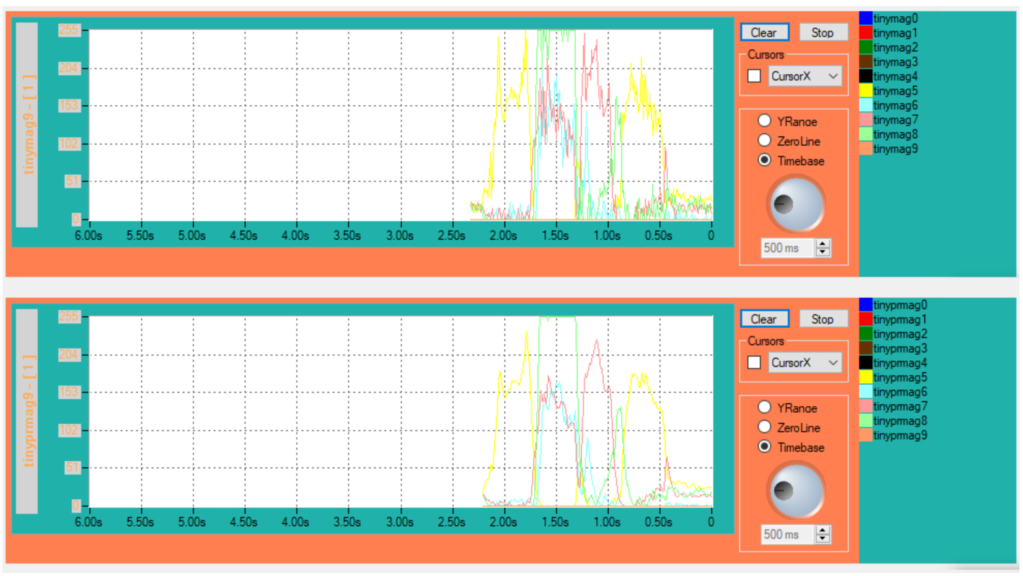


Figure 6: Dump of four movements (Wrist Supination/Pronation, Wrist Flexion/Extension) as classified by a linear regressor. The lower graph shows the result of *momentum* postprocessing.

Formatiert: Englisch (Großbritannien)

Finally, **Figure** 6 shows an example of the linear regression system which we implemented as well. Only four movements were tested here, the lower graph shows the output of the *Momentum* postprocessor, which weights the output over past frames and can likewise be used to reduce the effect of spurious movements.

7 List of SUPSI-IDSIA Contributions

The work done by IDSIA is based on software tools provided by OBHP, as described above. While it should be clear that isolating single pieces of work is of little practical value in this collaborative project, the following list nonetheless aims at summarizing the contributions by SUPSI-IDSIA:

- Introduction of polymorphism for the machine learning (classification/regression) task in the microcontroller software. This is very much in line with the goals of INPUT.
- Implementation of the postprocessing system, following our results described in deliverable D6.2.
- Implementation of a linear regressor trainable on-chip, which has in particular been used to test the implementation of polymorphism, to obtain an intuition about the latency of matrix multiplication, and to test the postprocessors.
- Implementation of the neural network classifier including the communication protocol and the host frontend.
- Reimplementation of the conversion of classifier/regressor results to prosthesis control commands. (This part will be thoroughly tested on actual prosthesis hardware in the remainder of task 6.3.)
- Several changes in *Inputflow* to accommodate the specific properties of the microcontroller EMG recording system, and of the preprocessing implemented in the existing microcontroller software.

1

2

8 Subcontracting

No subcontracting was required for the execution of this task. Hardware and software was provided by OBHP as described above.

9 References

[1] E. Scheme, B. Lock, L. Hargrove, W. Hill, U. Kuruganti and K. Englehart, "Motion Normalized Proportional Control for Improved Pattern Recognition Based Myoelectric Control," *IEEE Trans*

Neural Syst Rehabil Eng, vol. 22, no. 1, pp. 149-157, 2014.