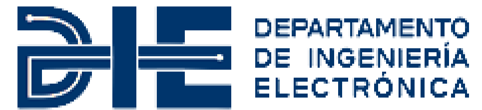


SENSOR FOR FOOD ANALYSIS APPLYING IMPEDANCE SPECTROSCOPY AND ARTIFICIAL NEURAL NETWORKS

Autor: Josef Peter Gessler

Tutor: Rafael Masot Peris

Cotutor: Miguel Alcañiz Fillol

Trabajo Fin de Máster presentado en el Departamento de Ingeniería Electrónica de la Universitat Politècnica de València para la obtención del Título de **Máster Universitario en Ingeniería de Sistemas Electrónicos**

Curso 2020-21

Valencia, Agosto de 2021

Abstract

The objective of the thesis was to design a portable device for the measurement of salt concentration in saline solutions applying electrical impedance spectroscopy (EIS) and artificial neural networks (ANN). In food analysis, electrochemical impedance spectroscopy is an effective method to measure chemical concentrations or to determine certain physical properties. At a basic level, the method consists of the acquisition of electrical impedances at various frequencies and subsequently the evaluation of the resulting series of complex values.

The first step was to investigate the feasibility of measuring with a frequency up to 30 kHz with the microcontroller ESP32. The ESP32 is a family of low-cost dual-core System-on-Chip-Controllers developed by Espressif Systems and designed for the Internet-of-Things.

The following step was to program embedded software in C++ for digital signal processing. Multitasking was realized with FreeRTOS in combination with the Arduino framework. Its main tasks are the stimulation signal generation with the internal DAC, to sample the resulting signals with the ADC in a DMA-accelerated mode and to calculate the complex impedance values.

Several possibilities for an adjustable transimpedance amplifier were examined. To enable precise and accurate measurement a PCB was designed with Proteus Design Suite carrying the ESP32-module and containing the analog amplifier circuit.

A graphical user interface was programmed in Python which communicates with the microcontroller via UART through a virtual serial port, to acquire sensor data, to configure the device and to visualize spectra.

Finally, a feedforward neural network was implemented. Spectroscopy data had to be acquired in order to determine an appropriate architecture and to train the ANN to classify given samples. The applicability of the prototype was proven by determining the salinity of aqueous solutions.

The presented work was carried out at the “Instituto Interuniversitario de Investigación de Reconocimiento Molecular y Desarrollo Tecnológico (IDM)” in Valencia, Spain.

Resumen

El objetivo de esta tesis fue el diseño de un equipo portátil para la medición de concentración de sal en soluciones salinas aplicando la espectroscopía de impedancia eléctrica (EIS) y redes neuronales artificiales (ANN). En el análisis de alimentos, la espectroscopia de impedancia es un método eficaz para medir concentraciones químicas o para determinar diversas propiedades físicas. El método consiste en principio en la adquisición de impedancias eléctricas a varias frecuencias y posteriormente la evaluación de la serie resultante de valores complejos.

El primer paso del trabajo fue investigar la viabilidad de medir con frecuencias de hasta 30 kHz con un microcontrolador ESP32. Los ESP32 son una familia de sistema en chip de doble núcleo y bajo costo desarrollada por Espressif Systems y diseñada para el Internet de las cosas.

El siguiente paso fue la programación de un software embebido en C++ para el procesamiento digital de señales. La multitarea fue realizada usando FreeRTOS en combinación con el framework de Arduino. Sus funciones principales son la generación de señales de estimulación con el DAC interno, medir las señales resultantes con el ADC en un modo acelerado por DMA y el cálculo de los valores complejos de impedancia.

Se examinaron varias posibilidades de un amplificador de transimpedancia ajustable. Para permitir una medición precisa y exacta, se diseñó utilizando Proteus Design Suite una PCB que lleva el módulo ESP32 y que contiene el circuito amplificador analógico.

Una interfaz gráfica se programó con Python que se comunica con el microcontrolador a través de un puerto serie virtual al UART para adquirir datos del sensor, configurar el dispositivo y visualizar espectros.

Finalmente se implementó una red neuronal prealimentada. Se tuvo que adquirir datos espectroscópicos para determinar una arquitectura adecuada y entrenar la ANN con el propósito de clasificar muestras dadas. La aplicabilidad del prototipo se demostró determinando la salinidad de soluciones acuosas.

El trabajo presentado se llevó a cabo en el “Instituto Interuniversitario de Investigación de Reconocimiento Molecular y Desarrollo Tecnológico (IDM)” en Valencia, España.

Index

1 Introduction 5
1.1 Requirements in the food industry 5
1.2 Introduction to EIS 5
1.3 Introduction to artificial neural networks 8
1.3.1 Linear filtering by matrix multiplication 8
1.3.2 Trainable filter 9
1.3.3 Introducing non-linearity 10
1.3.4 Regression and classification with neural networks 12
1.4 Technical objective 13
2 Hardware 14
2.1 Digital hardware: ESP32 and ESP development board 14
2.2 Analog hardware: Amplifier with selectable shunt 15
2.2.1 Impedance measurement 15
2.2.2 Signal conditioning 15
2.2.3 Selectable shunt 18
2.2.4 Further details on the circuit 19
2.3 Printed circuit board 21
2.3.1 Design of a PCB 21
2.3.2 Manufacturing the PCB 23
3 Firmware 26
3.1 Programming the ESP32 26
3.2 Signal processing 26
3.2.1 ADC and DAC configuration 26
3.2.2 Calculation of the impedance 29
3.2.3 Sensor calibration 29
3.2.4 Real-time signal processing on the ESP32 29
3.3 Communication between microcontroller and PC 31
3.3.1 Definition of the communication protocol 31
3.3.2 Implementing the communication module 32
3.3.3 Parameter handling 32
3.4 Implementing a feedforward ANN in C 33
3.4.1 Limitations of the ESP32 33
3.4.2 Solution for a customizable embedded neural network 33
3.4.3 Generating default value code and module testing 35

4 Software GUI 36
4.1 Principal functionalities 36
4.2 Features of Python and how they are applied 37
4.3 Communication between GUI and device 38
5 Programming a neural network 39
5.1 Supervised learning 39
5.2 Training of a neural network 39
5.2.1 Backpropagation 40
5.2.2 Gradient descent 41
5.2.3 Stochastic gradient descent and mini-batch descent 41
5.2.4 Applying momentum 42
5.3 Neural network in Python 43
5.3.1 Object-oriented design of an artificial neural network 43
5.3.2 Experiment 1: Reconstructing a signal 46
5.3.3 Experiment 2: Classifying capacitors 47
5.3.4 Applying neural networks to real-world problems 49
6 System validation with saline solutions 50
6.1 Acquisition of sensor data for training 50
6.2 First attempt to evaluate the data with an ANN 51
6.3 Overfitting and prevention 52
6.3.1 Using validation data and early stopping 52
6.3.2 Quality of the training data 53
6.3.3 Training with noise added to the input 53
6.3.4 Weight decay 54
6.3.5 Deep networks versus wide networks 54
6.4 Results 54
7 Conclusions and future works 59
8 Bibliography 61
9 Appendix 62
9.1 Detailed schematic of the analog hardware 62
9.2 Command keywords 63
9.3 Device parameters 63
9.4 Overview activation functions 64
9.5 Overview loss functions 64

1 Introduction

1.1 Requirements in the food industry

Measurement is an important prerequisite for quality assurance. The food industry requires to maintain certain quality standards and at the same time to keep costs low because there is strong competition. Neither is it affordable to waste food, nor is it acceptable to sell some food with dangerous contaminations of bacteria or chemicals. Next to natural factors, a problem is the possible fraud in the food industry, examples are mixed olive oil falsely labeled as extra virgin oil, frozen fish sold as fresh fish or even wine mixed with dangerous alcohols. Consequently, it is necessary to constantly supervise food products. However, the analysis of food is not straightforward. Since food products are natural products their complex composition varies, a fact which complicates the measurements considerably. Electrical impedance spectroscopy is an advanced technology which already is applied in food analysis, for example to determine freeze damages in citrus fruits [1][2], to measure ethanol content in pineapple waste [3], to monitor bacteria concentration in food [4] or to differentiate between fresh and frozen-thawed fish [5]. The objective is to apply this technology in a portable low-cost device which measures robustly, non-destructive and is customizable for specific applications.

1.2 Introduction to EIS

The EIS technology has a long history and was already used at the end of the nineteenth century but became practical applicable in the 1980s due to the introduction of computer technology. EIS has a wide range of applications, e.g., determining the chemical properties of liquids [6], analyzing the tissue of vegetables and fruits, determining the state of batteries, measuring the muscle and fat ratio in the human body, etc.

The principle consists basically in the stimulation of the sample by applying an alternate voltage and to measure the amplitude and phase of the resulting current at several frequencies.

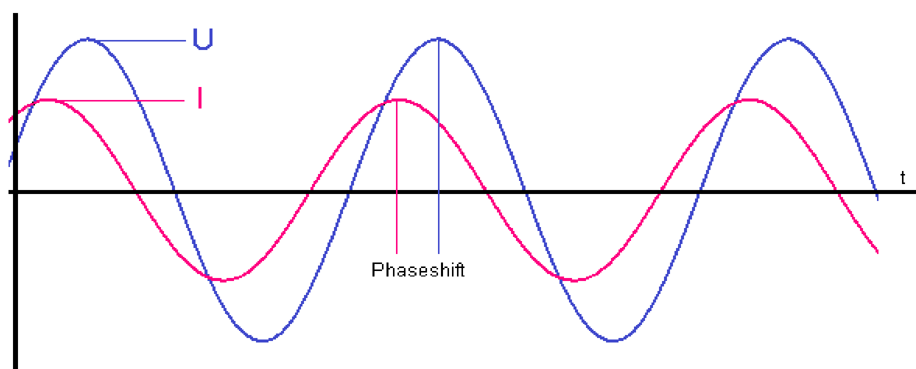


Figure 1: Applying a sine wave as voltage results in a current with sine shape and shifted phase

A very simplified model to understand the electrical characteristics of a sample is to imagine it as an RCR-circuit consisting of two resistors and one capacitor. There are two possible topologies, one with inner parallel resistor and one with inner serial resistor.

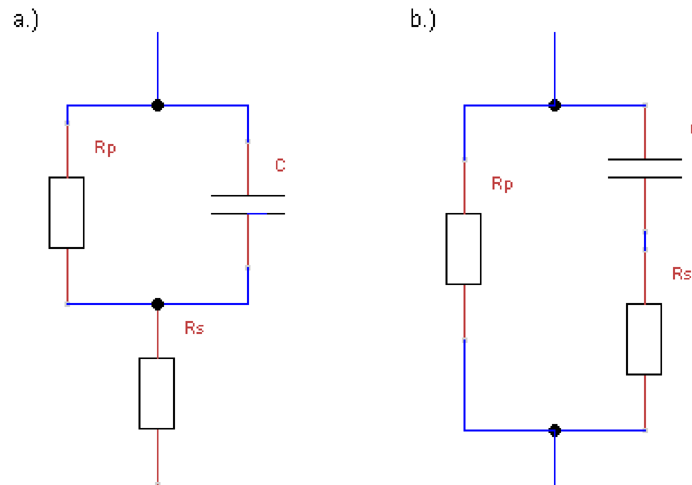


Figure 2: Two possible RCR-circuit topologies

An RCR-circuit with given parameters can be transformed from one topology to the other knowing the rules of the Y-Δ transform, therefore it theoretically doesn't matter how the sample is represented. However, the second topology, with inner serial Rs and outer Rp is a model closer to the natural system of an organic tissue.

The impedance depends non-linearly on the resistances and the capacitance of the circuit, and its complex value can be calculated by following formula:

$$\overline{Z}(\omega) = R_P||(R_S - \frac{j}{\omega C})$$

$$\text{With } \omega = \pi f$$

Accordingly, the impedance of this topology is at very low frequencies:

$$\lim_{\omega \to 0} \bar{Z}(\omega) = R_P$$

And at very high frequencies:

$$\lim_{\omega \to \infty} \bar{Z}(\omega) = R_P||R_S$$

The fact that the capacitance charges and discharges results in the dynamic behavior resulting in phase shift and modulus depending on the frequency. Assuming the network consists only of resistive and capacitive elements, the modulus of the impedance is monotonically decreasing with increasing frequency and the phase shift is negative.

There are several ways to represent an impedance spectrum. In this work the Bode-diagram is used where the X-axis defines the frequency, and the modulus is shown together with the phase. Often it is better for visibility to use a logarithmic scaling for the modulus and the frequency.

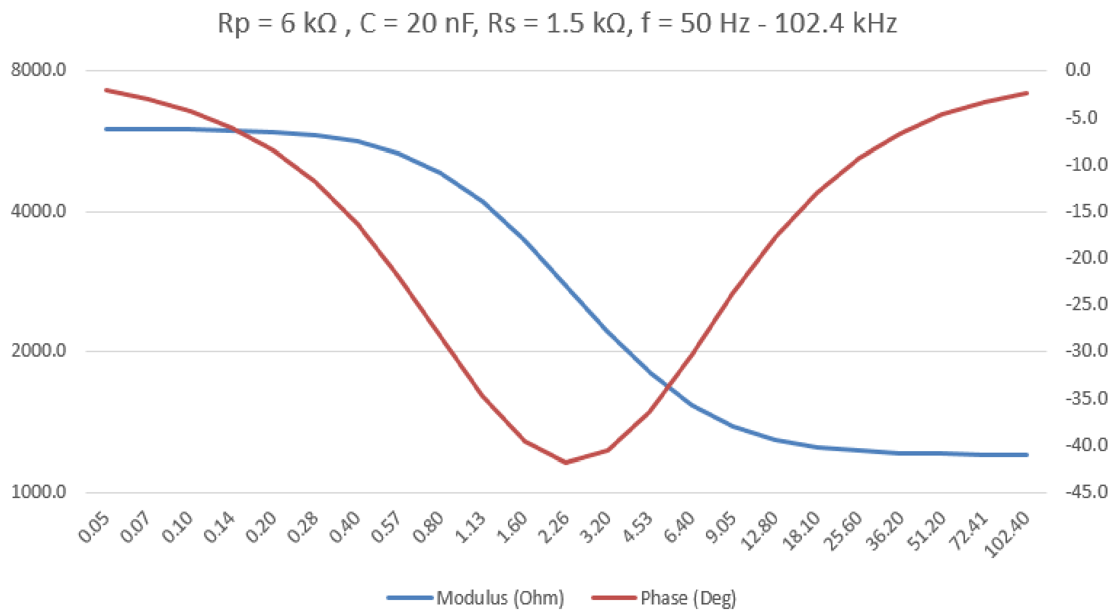


Figure 3: Bode-diagram of typical impedance spectrum of an RCR-circuit

Organic tissues, e.g., of fruits, meat or vegetables are made up of cells which contain a liquid with certain conductivity, that is enclosed in a wall with higher resistance. These cell walls act as capacity C and the inner liquid as resistance Rs. These cells are also surrounded by a conductive medium acting as the outer resistor Rp. If the chemical properties have changed or the cells are damaged, e.g., by freezing, this results in a change of the electrical characteristics visible in the impedance spectrum.

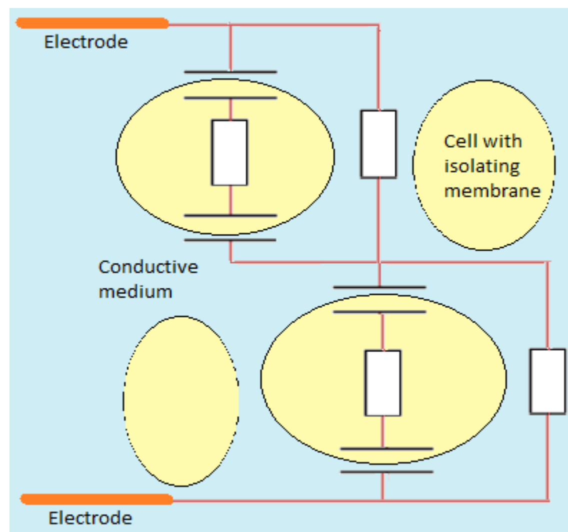


Figure 4: Electrodes measuring a tissue consisting of cells with an isolating barrier

The resulting spectrum of a natural tissue, which is rather a network of countless resistances and capacitances, differs from a simple RCR-circuit and normally can´t be evaluated using a simple formula. Either it is necessary to employ a person with knowledge about the appearance of the signals and who knows to evaluate them, or it is required to apply an algorithm which can learn how the signals appear and what results can be derived. Often the underlying parameters are too complex and not clearly visible to allow a manual evaluation and it is also required to automate the evaluation process. The data evaluation algorithm applied in this work is an artificial neural network, a concept which has proven itself in image recognition.

Multivariate data analysis tools like PCA or PLS have been used in the past [1][2] to build classification models for EIS applications. However, these have the shortcoming of not supporting non-linear functions. EIS in combination with neural networks has successfully been applied for determining physical properties like freeze damages in citrus fruits [1][2] and chemical properties like ethanol content in pineapple waste [3].

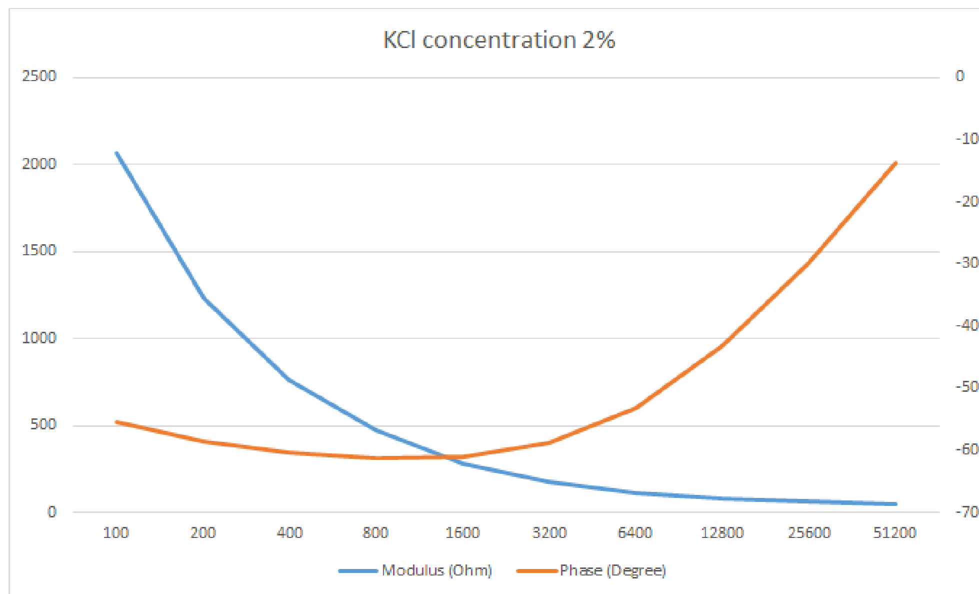


Figure 5: Impedance spectrum with modulus and phase of a 2% KCl solution

1.3 Introduction to artificial neural networks

A fundamental part of this work is the application of ANNs, which requires a basic understanding of this machine learning approach. Neural networks have a very wide range of possible applications and are likely to play a key role in almost every type of science and engineering in the future when more powerful digital hardware will be available. The idea of neural networks doesn´t originate from computer science, electronics or mathematics but from neuroscience and was used to explain the function of the brain. The theory continues to develop and a lot of literature on possible applications has already been published. Here the basic functionality of a neural network is explained from the perspective of signal processing like a non-linear adaptive filter.

1.3.1 Linear filtering by matrix multiplication

Given a vector containing a noisy signal, for example the sequence of spectroscopy, the data can be smoothed by multiplying the signal vector with a matrix containing the impulse response of the required filter. For example, a multiplication by a square matrix **A** containing the values

$$A_{i,j} = e^{-(\frac{i-j}{\alpha})^2}$$

will result in a convolution with the gaussian curve and thus it realizes a filter.

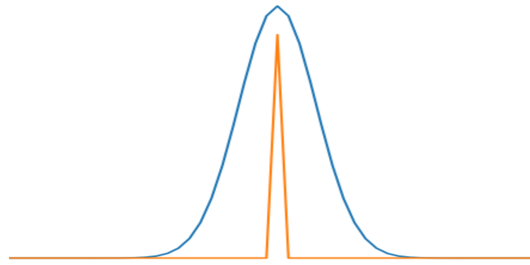


Figure 6: Impulse and impulse response of a gaussian filter

If a vector representing a signal is multiplied by this matrix, the higher frequency components are attenuated.

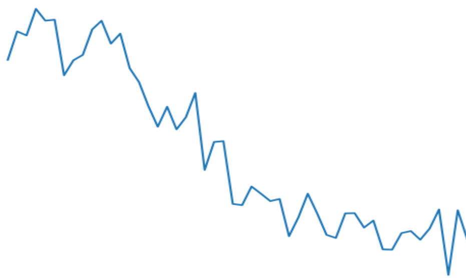


Figure 7: Noisy signal

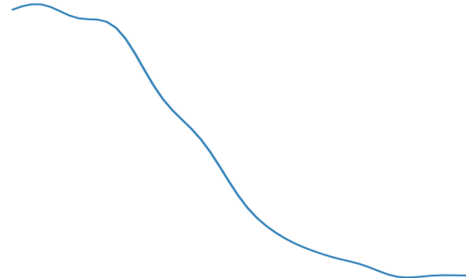


Figure 8: Filtered by matrix multiplication

1.3.2 Trainable filter

Often, it's difficult to find good filter parameters, so it's beneficial to have a filter which can learn to estimate the most likely original signal.

For adapting the filter, it is necessary to have a set of noisy input data and additionally reference data representing the desired output. The filter processes the noisy input data and thus estimates the correct signal. The output must be compared with the reference data and the resulting error values indicates how to update the filter settings **ϑ**.

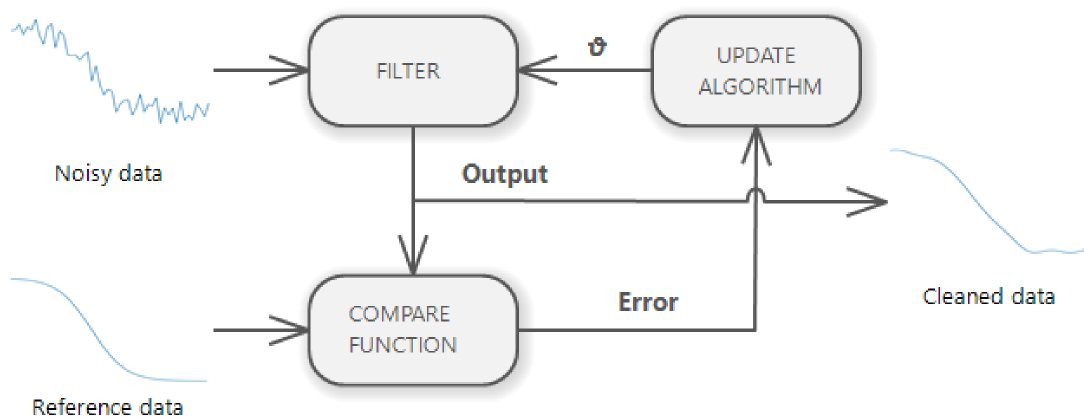


Figure 9: Principle of a trainable filter

To fit a filter usually the sum of the squares of the output errors will be minimized, in that way the filter learns to transmit the frequencies of the correct signal and to reduce the noise.

Minimizing the square of error is the solution to estimate the values with the maximum likelihood if the error is assumed as gaussian distributed [7]. To fit the filter parameters $\boldsymbol{\vartheta}$, an optimization algorithm using the gradient of the error function can be applied [8].

1.3.3 Introducing non-linearity

Linear filters are good in estimating the true value if the disturbance is evenly distributed symmetric noise. For some events, however, disturbances can occur unevenly distributed, for example the signal of a very sensitive photodiode can be disturbed by a single gamma ray. Such an outlier might appear as a needle and could easily be recognized and be marked as faulty data by a manual data evaluation, provided that the evaluating person knows the appearance of the erroneous and correct signals, however, a linear filter would just convolute the signal without distinguishing good from faulty data.

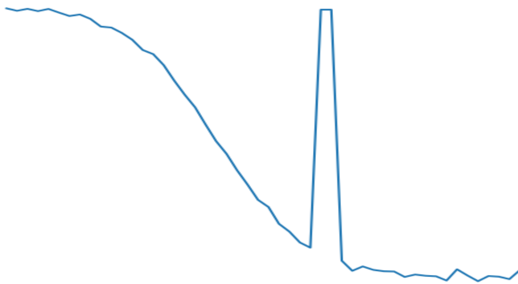


Figure 10: Signal with outlier

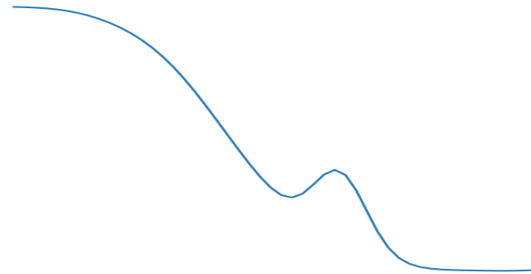


Figure 11: Linear filter convolutes the signal with outlier

For the simple case of an outlier, the signal could be repaired by comparing the values to a threshold and replacing the sample with the mean value of the neighboring samples. But often anomalies in data are very complex to detect and it would be very difficult or practically impossible to define a correcting algorithm and to determine the multitude of parameters.

Therefore, it's preferable to solve the problem using a trainable algorithm that learns to estimate the true data from given examples. The previously explained trainable filter has the shortcoming that it can only perform linear operations, however effective detecting and eliminating anomalies like an outlier requires non-linear operations.

The before mentioned filter using a matrix multiplication can be extended to perform non-linear functions by the usage of a stack of several matrix multiplications with added biases in combination with non-linear activation functions on the output value. This adaptable system can be tuned using an optimization algorithm to fit to the given training data, containing input and desired output values.

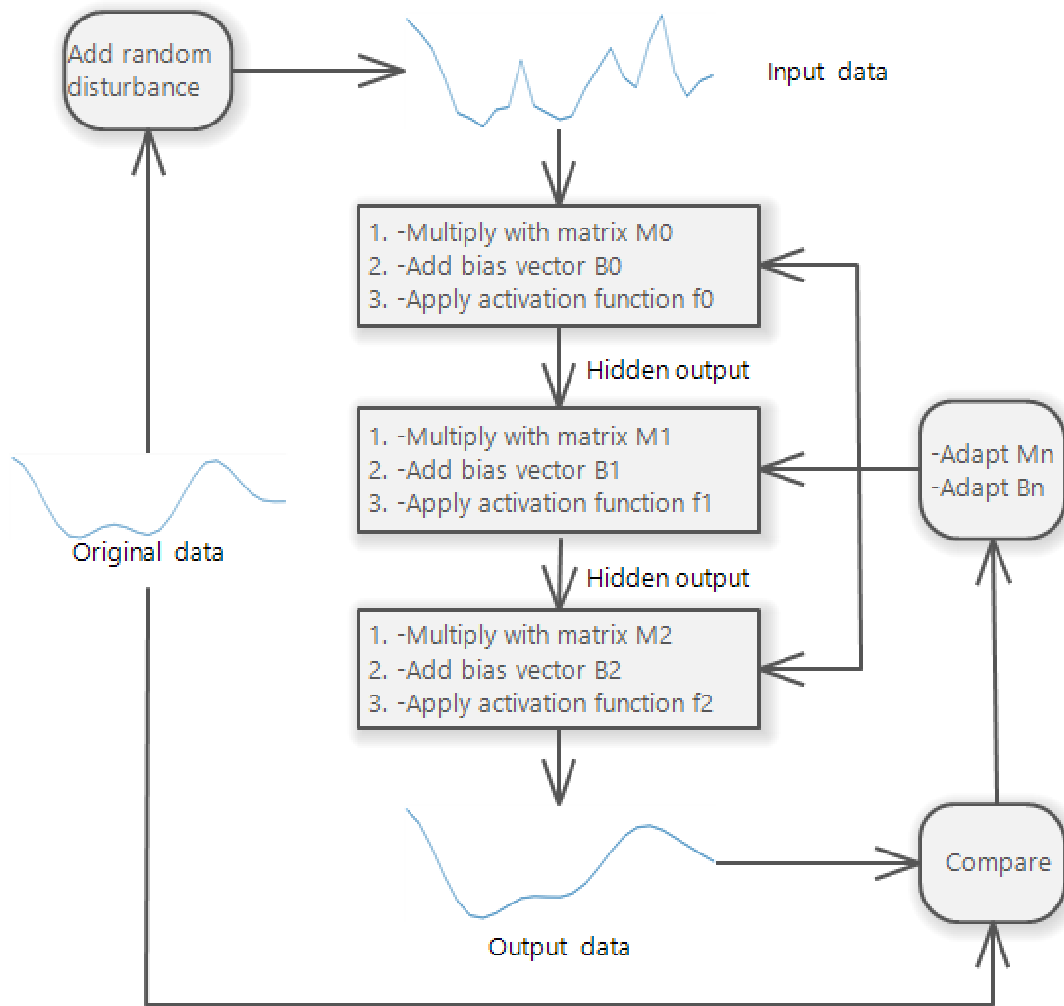


Figure 12: Several adaptable layers with non-linear output

The non-linear function for example could be the sigmoid function:

$$\mathrm{Sigmoid}(x) = \frac{1}{1 + e^{-x}}$$

or the rectified linear unit (ReLU):

$$\mathrm{ReLU}(x) = \begin{cases} 0 \text{ if } x < 0 \\ x \text{ if } x \geq 0 \end{cases}$$

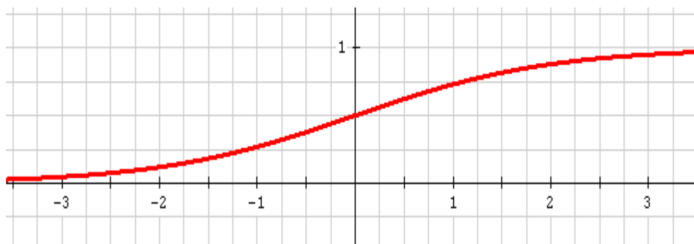


Figure 11: Sigmoid function

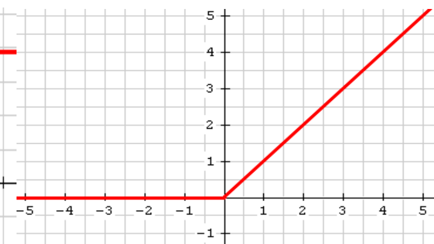


Figure 12: ReLU function

See appendix for often used activation functions.

1.3.4 Regression and classification with neural networks

Like filtering noise from a signal, a matrix can also be used to determine the most likely parameters of a linear combined function from given data:

$$f(x) = \alpha g(x) + \beta h(x) + \ldots$$

The parameters (α,β, etc.) of the model can be estimated by minimizing the sum of the squares of the differences between the data samples and the resulting function. However, this estimation is prone to being falsified by outliers.

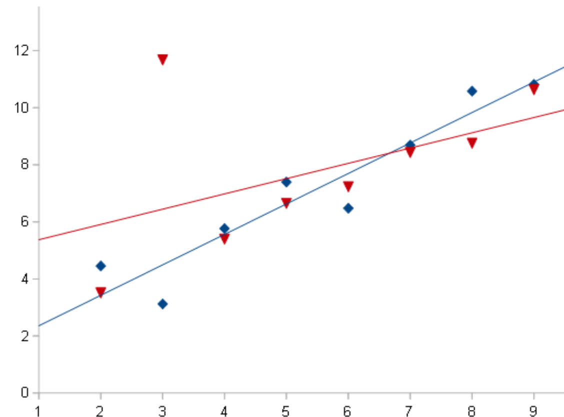


Figure 13: Blue: Data without outlier, Red: Data with outlier and disturbed regression

An ANN, however, can learn to ignore outliers and thus provide a more robust regression. A further advantage is it can learn non-linear dependencies between the estimated parameters and the given data. For example, the spectrum of an RCR-circuit has a non-linear dependency on the values Rp, C and Rs, which can be learned by an ANN, provided that the ANN allows sufficient complexity and enough appropriate data to learn the correct function is available.

In many data analysis applications, it is not required to estimate a parameter value but to perform a classification, that is, to determine the most likely corresponding class from given input data and therefore it is necessary to provide an output indicating the probability of the possible categories. For example, an algorithm can learn to classify images according to the kind of emotion they cause, if there are people indicating the corresponding category of examples for the training process. An industrial application could be to classify citric fruits on the quality of their texture using EIS [1][2].

1.4 Technical objective

Electrical impedance spectroscopy is often carried out by analyzers made for usage in laboratories, not portable and usually quite expensive. However, more powerful and cheaper microprocessors have become available, which allow the application of advanced technologies.

The objective of this work is to prove the feasibility of a portable device applying the microcontroller ESP32. This device should measure the electric impedance from 100 Hz up to at least 30 kHz and evaluate the resulting data using a feedforward ANN for the automated determination of chemical parameters. The impedance shall be acquired using the internal DAC and ADC of the ESP32 in combination with analog amplifiers to generate the stimulation signal and to measure the resulting alternate current. In order to measure in various magnitudes, it is necessary to have various shunts selectable through the microcontroller. The whole device also shall be configurable through a GUI on a personal computer via USB connected to the UART, what requires the implementation of a protocol. Since the required architecture of the neural network can vary depending on the application, it is necessary to implement an ANN algorithm supporting configurable number of layers and layer widths.

The developed hardware and software shall be used to acquire EIS sensor data, which subsequently is used to train the neural network and to determine an appropriate network configuration. The parameters of the trained ANN shall be saved to a file to be used later in the GUI or on the microcontroller. Finally, the resulting device shall be tested by determining at least 10 different saline solutions.

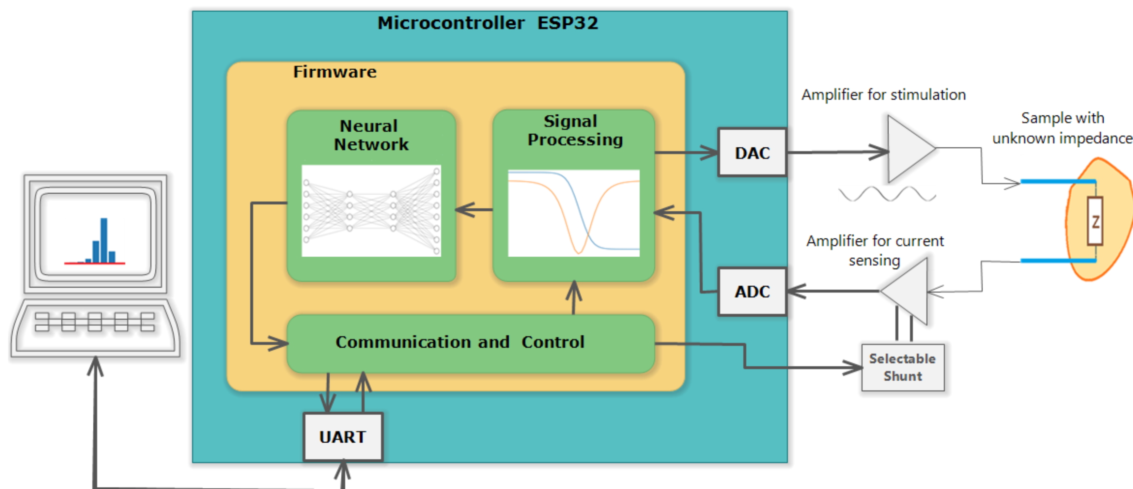


Figure 14: Block diagram of the sensor system

2 Hardware

2.1 Digital hardware: ESP32 and ESP development board

As digital part of the hardware, the development board ESP32-DEVKIT-C is used, which includes the module ESP32-WROOM-32D containing the chip ESP32-D0WD.

The ESP32 is a family of low-cost, low-power System-on-Chip controllers designed for the internet of things. It includes two Tensilica LX6 cores with floating point support, clock frequency up to 240 MHz and 512 kilobytes SRAM. Two integrated 12-Bit ADC can be used in a DMA-accelerated mode with a sampling rate up to 2 mega samples per second and can be fast multiplexed to several channels. Additionally, it includes a random number generator (RNG) for encryption purposes, EEPROM, UART, SPI, DAC, Timers, and many other peripherals.

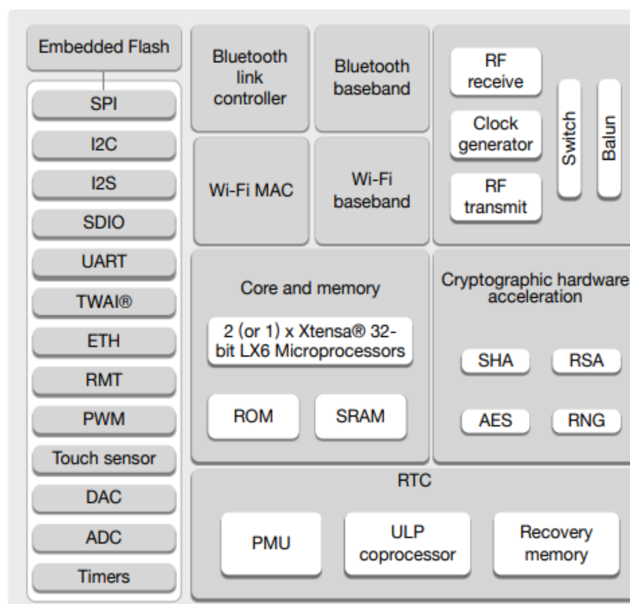


Figure 15: Block diagram of the ESP32 chip

Figure 16: ESP32 development board

The ESP32-DEVKIT-C contains the ESP32-WROOM-32D module enabling the usage of Wi-Fi and Bluetooth, and additionally a USB-interface which allows it to communicate with the UART of the ESP32 and to program the flash via USB. A voltage regulator supplies 3.3V which is lead to the pin header together with the 5V from the USB and thus can be used also for the analog circuits.

2.2 Analog hardware: Amplifier with selectable shunt

2.2.1 Impedance measurement

The electrical impedance is the complex valued ratio of a driving voltage and the flowing current at a given frequency:

$$\overline{Z} = \frac{\overline{U}(f)}{\overline{I}(f)}$$

The impedance of a sample for a specific frequency can be determined by applying a sinusoidal alternating voltage with constant amplitude and measuring the amplitude and phase of the resulting current. If the measured sample is a linear system, it does not make any difference what amplitude for stimulation is used. However, in many applications there are non-linear electrochemical effects causing distortions when high amplitudes are applied, which makes it preferable to use lower amplitudes [9]. On the other hand, a very low stimulation voltage should be avoided because it could result in a noisy measurement. In this work the applied amplitude is 0.5V, which can be configured by a parameter in the firmware.

The ADC of the used microcontroller can measure only positive voltages (0V to 3.3V), thus the voltage should oscillate around the middle of its range (1.65 V) defined as virtual ground.

2.2.2 Signal conditioning

There are several possible circuits to obtain the impedance measurement.

2.2.2.1 Current measurement via shunt voltage

An obvious solution is to measure the applied voltage and to determine the resulting current by measuring the voltage on a shunt like the circuit in the following picture:

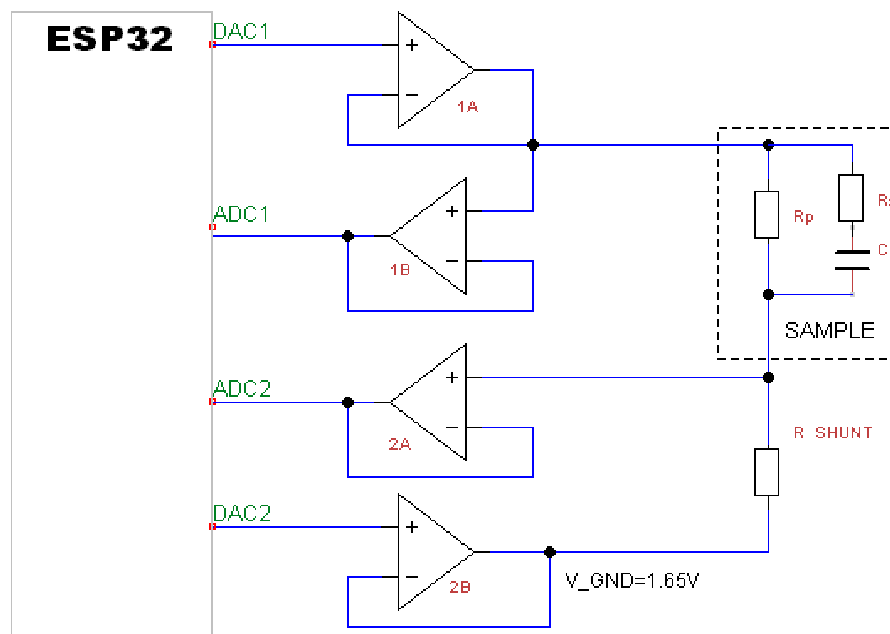


Figure 17: Circuit for impedance measurement which determines the current by measuring the voltage on the shunt

With this configuration the impedance of the sample can be determined, assuming the current in the shunt is equal to the current in the sample.

$$\mathrm{I(t)_{Shunt}} = \mathrm{I(t)_{Sample}} = \frac{\mathrm{U(t)_{ADC2}} - \mathrm{U_{VGND}}}{\mathrm{R_{Shunt}}}$$

$$\mathrm{U(t)_{Sample}} = \mathrm{U(t)_{ADC1}} - \mathrm{U(t)_{ADC2}}$$

After acquiring the signals $U(t)_{Sample}$ and $I(t)_{Sample}$ through ADC1 and ADC2 the complex values $\underline{U}(f)_{Sample}$ and $\underline{I}(f)_{Sample}$ can be determined:

$$\overline{\mathrm{U}}(f)_{Sample} = U(t)_{Sample} * e^{-2\pi f t}$$

$$\overline{\mathrm{I}}(f)_{Sample} = I(t)_{Sample} * e^{-2\pi f t}$$

With ***f*** as the corresponding frequency of the acquired impedance, equal to the frequency of the stimulation generated by DAC1.

The quotient of the complex voltage and current is the complex valued impedance.

$$\overline{\mathrm{Z}}(f)_{Sample} = \frac{\overline{\mathrm{U}}(f)_{Sample}}{\overline{\mathrm{I}}(f)_{Shunt}}$$

More advanced techniques are to use four-terminal sensing and to apply instrumentation amplifiers to measure the voltage on the sample and on the shunt. Instrumentation amplifiers are used to amplify low voltage signals [10], what makes them preferable if the perturbation or shunt voltage is very weak. Four-terminal sensing reduces the error caused by the cables and thus improves the accuracy. Such configurations are appropriate for accurate and precise measurements.

2.2.2.2 Self-balancing bridge

Another possibility to measure the flowing current is to use the shunt on the amplifiers feedback, which requires two operational amplifiers like the circuit in the following picture:

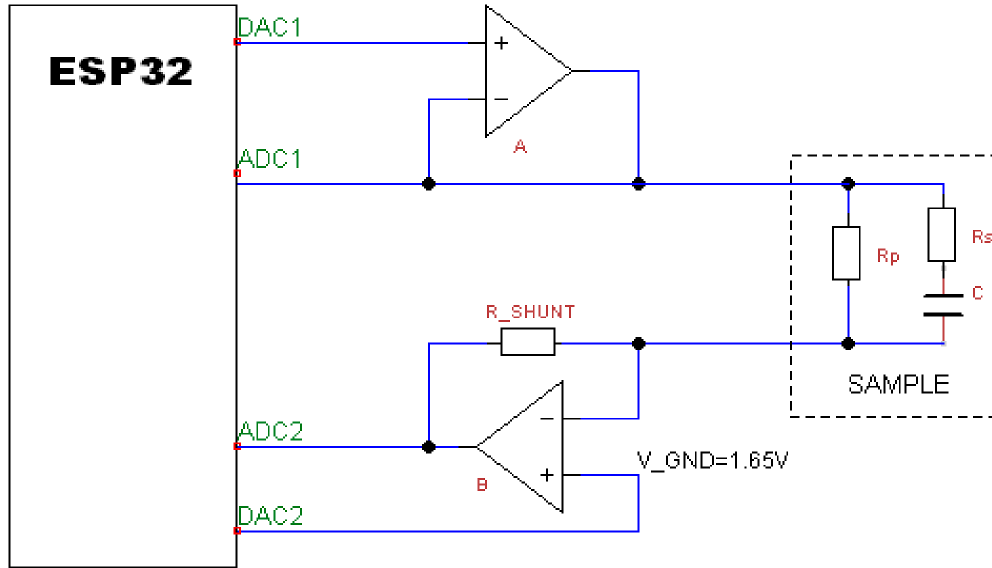


Figure 18: Circuit with self-balancing bridge

For an ideal operational amplifier, it can be assumed that the current through the sample is equal to the current through the shunt. The inverting input of the amplifier will be balanced by the feedback, which is the shunt resistance, to be equal to the virtual ground [13]. Consequently, the voltage on ADC2 depends linearly on the current through the sample.

$$U_{ADC2} = U_{OA_{out}} = U_{VGND} - I_{Sample} * R_{Shunt}$$

$$\frac{U_{VGND} - U_{ADC2}}{R_{Shunt}} = I_{Sample} = \frac{U_{DAC1} - U_{VGND}}{Z_{Sample}}$$

$$U_{ADC2} = U_{VGND} - (U_{DAC1} - U_{VGND}) * \frac{R_{Shunt}}{Z_{Sample}}$$

A disadvantage of this configuration is that the ADC2 could get into saturation if the impedance of the sample is too low. If the amplitude of the stimulation $U_{AmpStim}$ shall be 0.5V and the virtual ground voltage be 1.65V, the highest and lowest output value on the DAC can be determined:

$$U_{DAC1Max} = U_{VGND} + U_{AmpStim} = 1.65V + 0.5V$$
$$U_{DAC1Min} = U_{VGND} - U_{AmpStim} = 1.65V - 0.5V$$

The voltage on the ADC2 does not exceed the input range 0V to 3.3V if:

$$U_{VGND} - U_{AmpStim} * \frac{R_{Shunt}}{Z_{Sample}} > 0V$$

And if:

$$U_{VGND} + U_{AmpStim} * \frac{R_{Shunt}}{Z_{Sample}} < 3.3V$$

Accordingly, the absolute value of the measured impedance must not be much lower than the resistance of the shunt (maximum three times lower since the ADC range is 0V to 3.3V and the applied amplitude 0.5V).

The self-balancing bridge is a preferable circuit topology for low-cost devices and appropriate for measurements with lower frequencies up to 40 MHz [11]. For the objective of this work, where the applied frequency is less than 100 kHz, it is an appropriate solution.

2.2.3 Selectable shunt

The magnitude of the impedance of the measured sample can vary in a wide range (10Ω to 100kΩ), but to measure with a high precision the shunt resistance should be in the same order of magnitude as the impedance to be measured. Therefore, it is necessary to have the possibility to exchange the shunt and for fast automatic processes it is also required to change the resistance through the software.

Digital potentiometers can be configured via SPI and can take a value in a wide range, for example from 100Ω to 100kΩ, with 1024 steps. However, their problem is a high temperature dependency, 5ppm/°C (ppm of their full scale = 100kΩ), thus if the temperature varies in a range of 20°C, the resistance will vary around 10Ω, which is not acceptable for low impedance measurement. Instead of digital potentiometers usually analog multiplexers are used.

The first evaluated multiplexer was the ADG804, which has a low typical on resistance of 0.5 Ω.

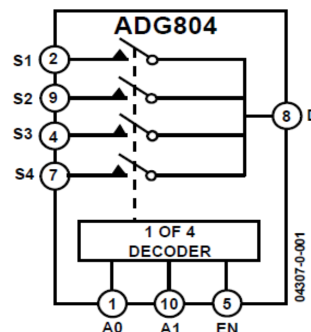


Figure 19: Pinout of the ADG804 (the same as for ADG704)

The multiplexer can be enabled, and the required shunt can be selected by three signals from the microcontroller. To test the measurement with a multiplexer, ohmic resistances were measured, which should result in a spectrum with constant magnitude and without phase shift. The impedance measurement worked well with shunts with low resistance value (100Ω) but using the highest shunt (100K) resulted in an erroneous measurement at higher frequencies. The magnitude at 30 kHz was 50% elevated and the phase shift was around 45°.

This error can be explained by parasitic capacitances in the multiplexer, which can be imagined as capacitors which are connected in parallel with the switches. That is, if a shunt with high resistance is selected, there is like an RC-element with lower resistance in parallel which causes the falsified measurement at higher frequencies.

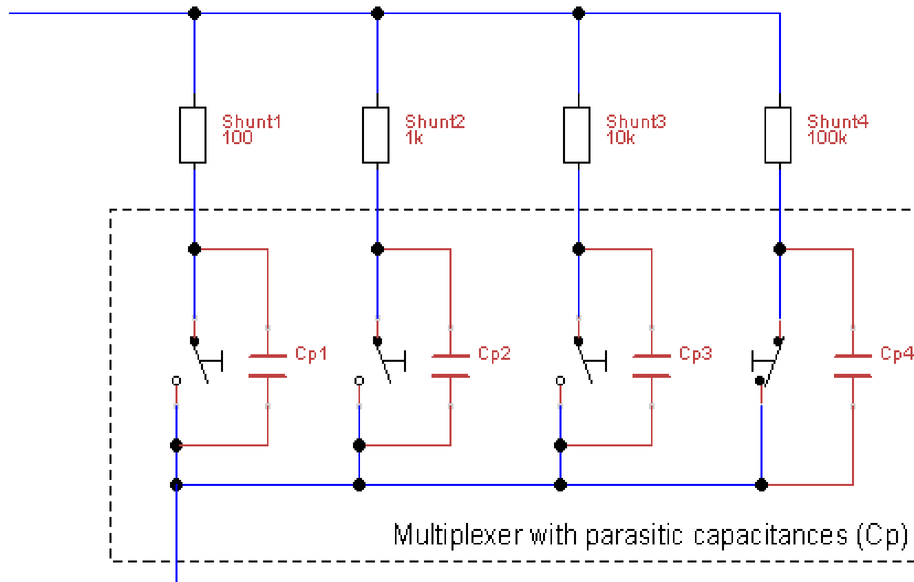


Figure 20: Problem with parasitic capacitances

Theoretically this error can be corrected by calibration. However, the parasitic capacitances could drift due to the temperature, what would result in inaccurate measurements despite an accurate calibration. For that reason, the parasitic capacitances must be as low as possible.

To solve this problem the ADG804 was replaced by the ADG704 (with the same pinout) which has much lower capacitances (Cs_off 9pF, Cd_off 37pF, compared to ADG804: Cs_off 24pF, Cd_off 105pF) and a typical on resistance of 2.5Ω which isn't a problem.

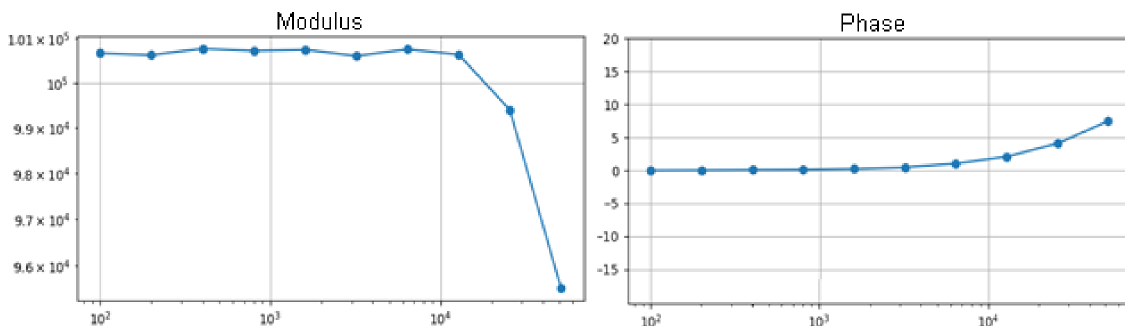


Figure 21:Modulus and phase of an uncalibrated measurement with 100K shunt

The results with the ADG704 measured on a 100K resistor were much better, although a magnitude error (5%) and phase shift (7° at 50 kHz) remained which is acceptable since most applications will have impedances in much lower magnitudes than 100K.

2.2.4 Further details on the circuit

The signal on the DAC for the stimulation does not come out as a perfect sine wave, but appears with a stair pattern, meaning that the signal contains higher frequency components. These components must be filtered, otherwise aliasing errors will occur resulting in corrupted measurements.

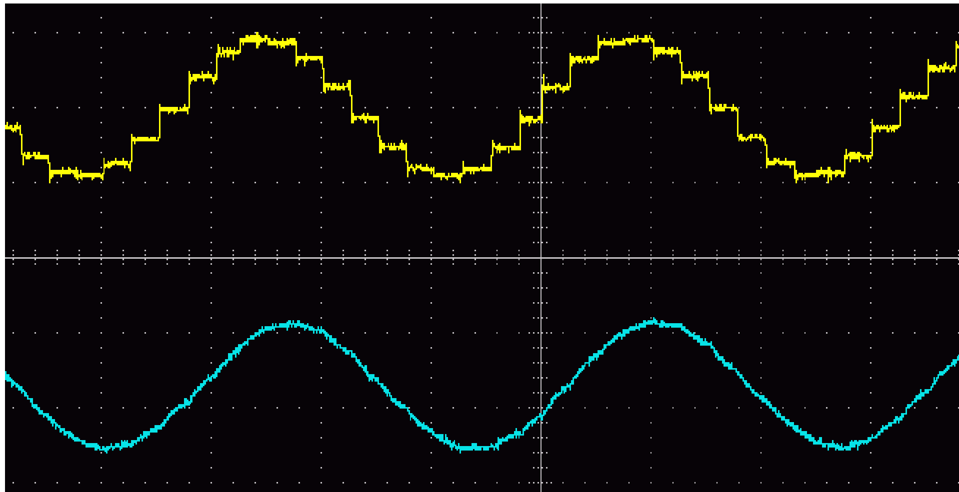


Figure 22:Output of the DAC (yellow) and amplified filtered output (blue) for the stimulation

The filter is realized with only one RC-lowpass on the output of the DAC, instead of two lowpass on the ADC inputs to avoid errors by asymmetric drift in components. The corner frequency of the used filter is around 60 kHz (R=3.9K, C =680p), consequently high frequencies can still be applied. It is possible to measure with frequencies close to the cut off frequency since the impedance is not calculated by the theoretic voltage, but by the measured voltage. Although the applied amplitude changes slightly the resulting impedance won't change.

Since there are problems by using ADC2, both measurements finally are realized by fast multiplexed channels of the ADC1.

The voltage of the required virtual ground of the circuit is set by the DAC2, configurable in the firmware.

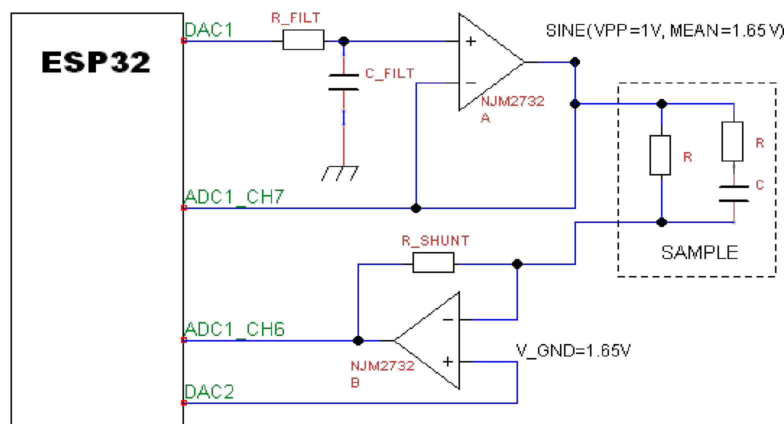


Figure 23: Circuit using RC-lowpass filter on DAC1

In order improve the EMC performance it is advisable to place a decoupling capacitor (not shown in the circuit) of around 0.1 µF on the power supply of the operational amplifier [12].

The applied amplifier is the NJM2732, a dual rail-to-rail operational amplifier. Despite a good performance a problem occurred when the impedances were below 100Ω, faulty signal appeared on the output of both operational amplifiers.

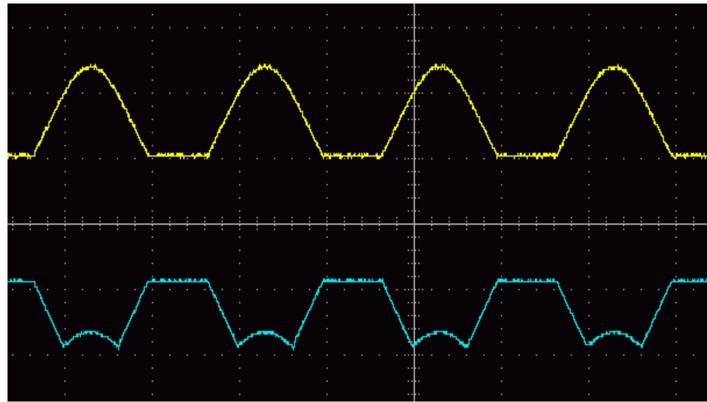


Figure 24: Stimulation signal on OA-a(yellow) and the signal from OA-b (blue) for the current measurement

This problem was caused by a too high load on the amplifier which reduces the maximum output voltage. The problem was solved by placing a 100Ω resistor in series with the output, whose error can be corrected by calibration.

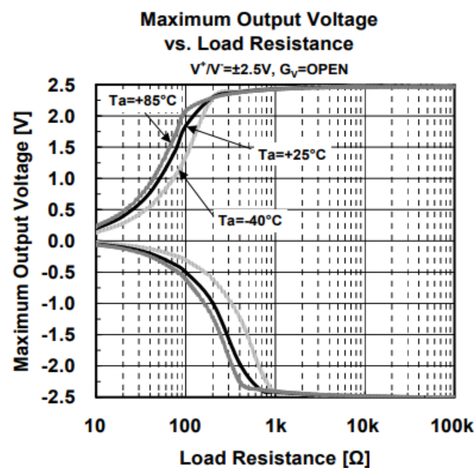


Figure 25: The maximum output voltage of the NJM2732 depending on the load

2.3 Printed circuit board

2.3.1 Design of a PCB

After finding an appropriate configuration on a breadboard, the PCB of a prototype was designed. The PCB was designed with Proteus 8 Professional, where the connections between the parts can be defined in a schematic view and the routes on the PCB can be placed using the auto routing function. Although very useful, the auto routing does not always work as desired, which sometimes requires interventions (putting keep outs, manual routing) by the designer to obtain a better solution for the specific problem.

The designed PCB contains only the analog components (operational amplifier, analog multiplexer, shunt resistor, etc.), while all digital parts are on the ESP32 development board carried by the PCB on two connecting bars. The development board also provides the 5V supply for the operational amplifier and 3.3V supply used by the analog multiplexer.

For the solution two layers are sufficient and the remaining spaces between the routes on both sides are used as ground planes to improve the EMC performance.

The shunt resistors should be soldered with SMD technique, but during the development phase the correct shunt values are not determined. In order to have exchangeable shunt resistors additionally through-hole-pads are placed to use through-hole resistors before the final SMD resistors will be placed.

Since the used manufacturing method does not allow too fine routes the width is set to 40 mils (around 1 mm). However, the analog multiplexer has very fine MSOP10-pins, thus the IC is not placed directly on the PCB, but on a MSOP10-to-DIP10 adaptor board.

For the later possibility to mount the PCB in a case, four holes for M3 screws are placed at the corners.

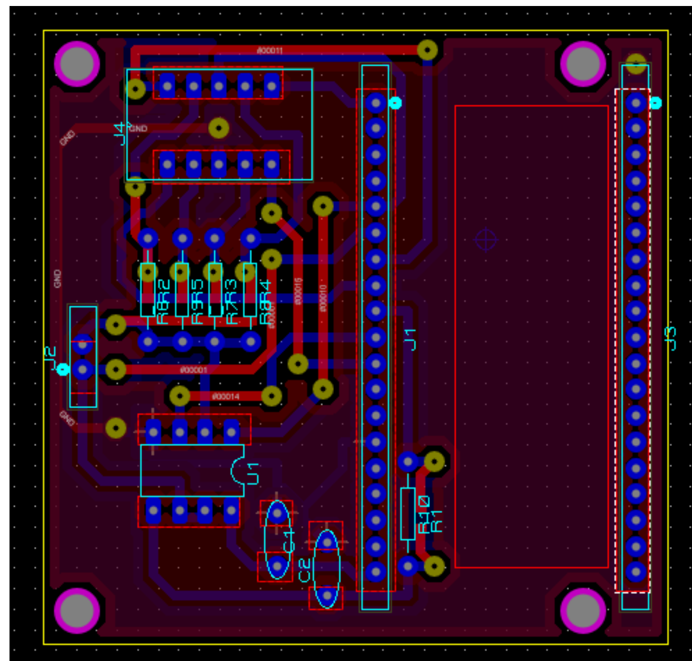


Figure 26: Resulting PCB layout

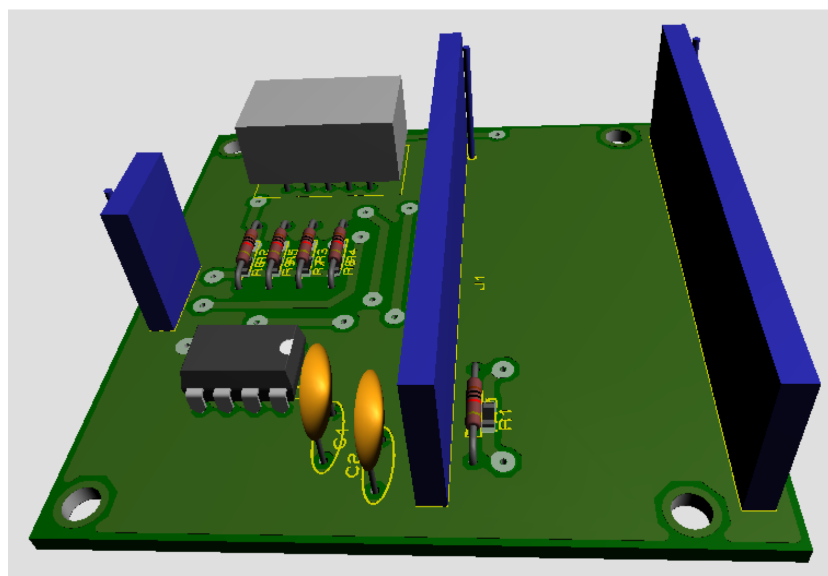


Figure 27: 3D-preview of the designed PCB in Proteus

2.3.2 Manufacturing the PCB

After designing and reviewing the solution, the PCB layout for both sides was printed on a film.

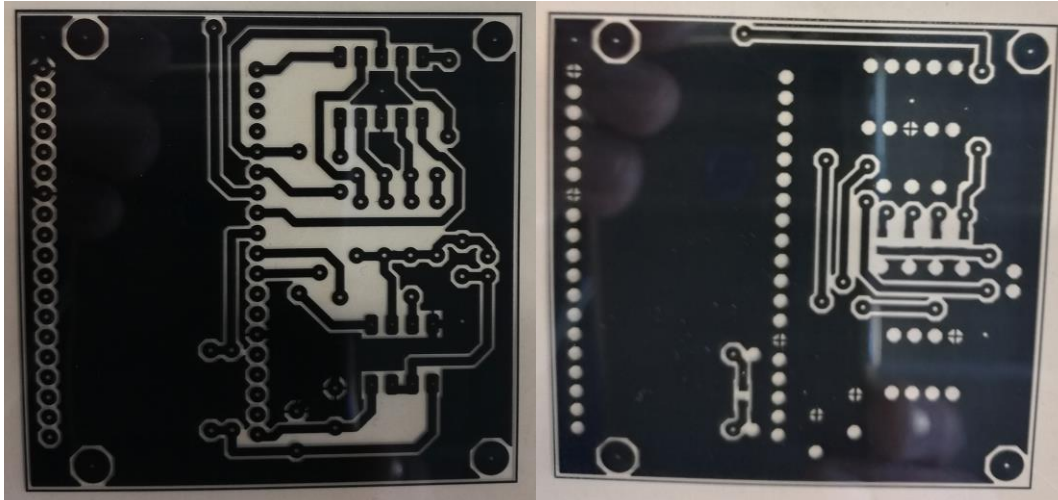


Figure 28: Layout printed on film

These films were attached to the untreated board which is covered by a photo-sensitive layer and subsequently radiated with UV rays.

Figure 29:Device for UV-treatment

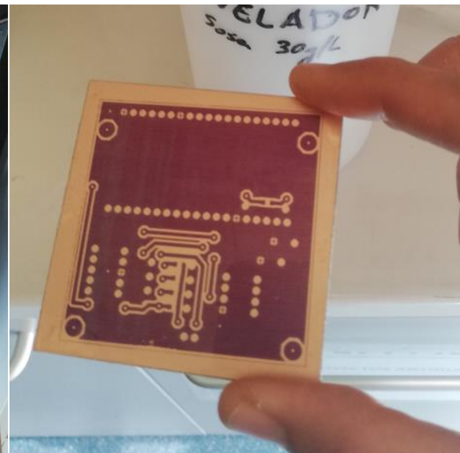


Figure 30: PCB after the UV-treatment

After the treatment with UV rays the board had first a bath in a developer solution, was cleaned and afterwards a bath in an etching mixture of hydrochloric acid and sodium perborate.

Figure 31: Etching chemicals

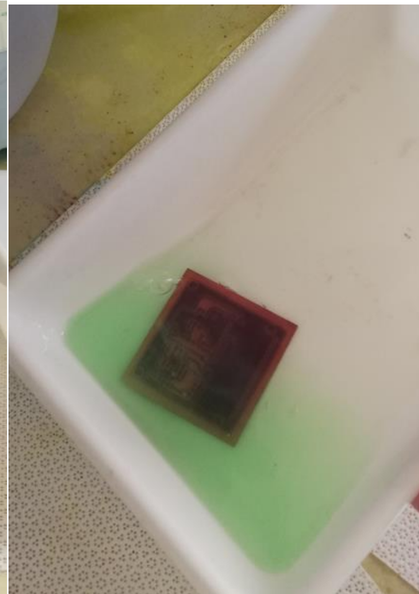


Figure 32: PCB in etching bath

After cleaning and drying the board was automatically drilled. The PCB had to be fastened and the coordinates calibrated, the data for the drilling was provided by the layout file.

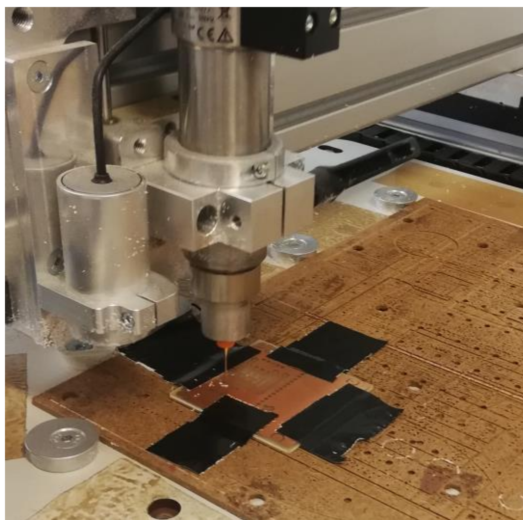


Figure 33: Automatic drilling

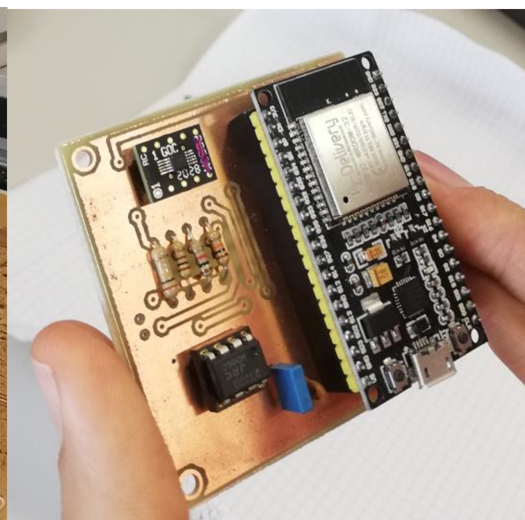


Figure 34: Components placed, ready to solder

Subsequently, the components were placed and soldered on the PCB.

The function of the PCB was verified by measuring RCR-circuits and comparing the resulting spectra to theoretical values.

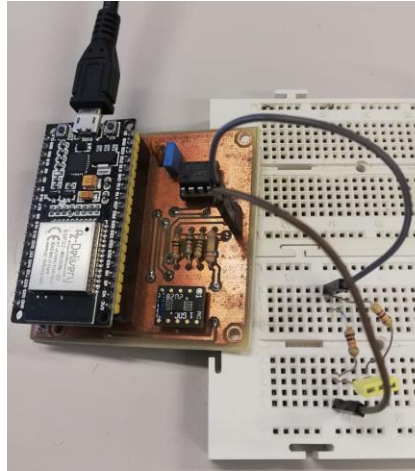


Figure 35: Test of the assembled hardware using an RCR-circuit

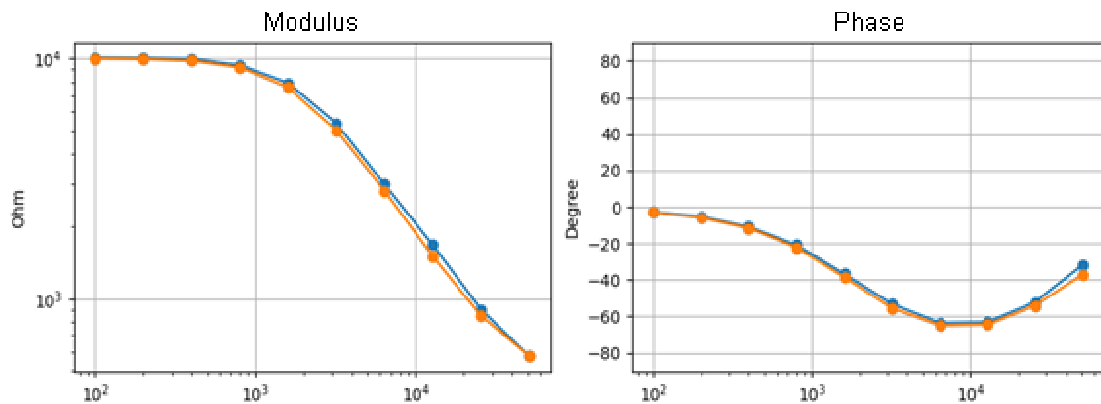


Figure 36: The impedance spectrum of the sensor (blue) and theoretical values (orange) of a circuit with Rp= 10kΩ, C= 8.2nF and Rs =470 Ω

The resulting PCB works well as a first prototype. With more advanced production methods, e.g., SMD, a more compact PCB can be realized which would perform even better.

3 Firmware

3.1 Programming the ESP32

There are several possibilities to program the ESP32. One possibility is to load a MicroPython interpreter into the flash of the ESP32 and to execute MicroPython scripts. The available MicroPython interpreter offers function interfaces allowing access to the ADC and to activate timer interrupts. However, this approach doesn´t allow to work with a sufficient high sampling rate, at least 60 kHz is necessary to fulfill the requirements.

There is the possibility to compile the MicroPython interpreter including external C-modules. This way it can be possible to write fast C-code handling the impedance measurement, which can be accessed via a defined MicroPython function interface [14]. The possibility to include external C-modules was examined, a simple external C-function was included and compiled. This requires working in Linux, to install Python, Git, GCC and many other tools, to modify CMake files and to check out the suitable version of *ESP-IDF*, the Espressif IoT development framework. This technique could be quite useful, e.g., for devices where the impedance measurement data is realized by a C-function encapsulated in the interpreter and subsequently the data are evaluated by an application specific evaluation algorithm implemented in MicroPython. However, since MicroPython is not very mature and it is a quite complex process to compile the MicroPython interpreter with external C-code and the digital impedance measurement with a high sampling rate itself is quite complex, this idea was discarded. Instead, the complete firmware was written in C/C++.

In this work the ESP32 is programmed with *PlatformIO IDE* as an extension of *Visual Studio Code*, which supports working with the JTAG-Debugger *ESP-Prog*. The ESP32 can be programmed with the *Arduino framework* in combination with the real-time operating system *FreeRTOS*. This enables the execution of several tasks and since the ESP32 is a dual core, the tasks can be assigned to two different cores. The RTOS also offers functions and structures such as events, mutexes and queues allowing communication between tasks.

3.2 Signal processing

3.2.1 ADC and DAC configuration

A very fundamental part of the measurement is the usage of the ADC and DAC. The first examined approach was to write on the DAC and to read the ADC in a timer-triggered interrupt routine. It was possible to write on the DAC and to read both ADC, however it worked only with sampling rates up to 58 kHz which is insufficient. Using interrupts at a very high rate also costs a lot of CPU time, another reason why a solution using DMA must be applied.

A DMA-controller is a module which allows transmitting data from a peripheral to the RAM with a high throughput without occupying the CPU [15]. The ESP32 offers the possibility to access the DMA through I2S [16] and accordingly the I2S-driver offers corresponding functions to use the ADC and DAC in a DMA-accelerated mode.

On both DAC can be written through a buffer, where the DAC1 obtains the sine signal with the corresponding frequency and DAC2 a constant value realizing the virtual ground.

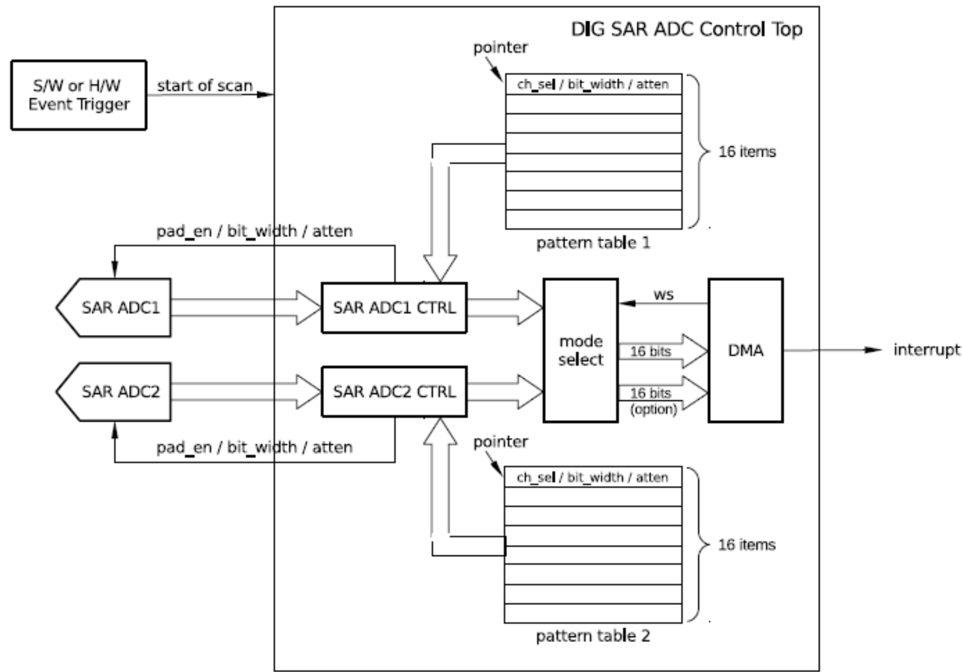


Figure 37: Block diagram of the ADC module with DMA [16]

```
i2s_config_t i2s_config =
{   //I2S with ADC and DAC
    .mode = (i2s_mode_t)   ( I2S_MODE_MASTER
                           | I2S_MODE_RX
                           | I2S_MODE_TX
                           | I2S_MODE_DAC_BUILT_IN
                           | I2S_MODE_ADC_BUILT_IN ),
    .sample_rate             = (2*i_AdcSampRate),
    .bits_per_sample         = I2S_BITS_PER_SAMPLE_16BIT,
    .channel_format          = I2S_CHANNEL_FMT_RIGHT_LEFT,
    .communication_format    = I2S_COMM_FORMAT_I2S_MSB,
    .intr_alloc_flags        = ESP_INTR_FLAG_LEVEL3,
    .dma_buf_count           = NUM_BUFF,
    .dma_buf_len             = SIZE_BUFF,
    .use_apll                = false,
};
```

Figure 38: Settings for the I2S in ADC/DAC mode

The ADC2 cannot be used for two reasons. First the current version of the I2S-driver does not support the usage of the ADC2, and second it should be avoided to use the ADC2 because it is used by the Wi-Fi module, which could be required for applications in the future. However, the ADC-module provides the possibility to scan the ADC1 with automatically multiplexed channels, using registers containing the corresponding switch-pattern. However, this functionality is not supported by the driver in ESP-IDF. To solve this problem the existing driver is used to initialize the ADC1 for both used channels and afterwards the configuration registers are modified to multiplex the channels.

```
#define ADC_CH1   ADC1_CHANNEL_7      // GPIO35
#define ADC_CH2   ADC1_CHANNEL_6      // GPIO34
#define NUM_SAMPS 250
#define NUM_CHANNEL 2
#define ADC_PATT (0x0f0f0f0f | (ADC_CH1<<28)|(ADC_CH1<<12)| (ADC_CH2<<20)|(ADC_CH2<<4) )

//config ADC via I2S-driver
adc1_config_width (ADC_WIDTH_12Bit);
i2s_driver_install (I2S_NUM_0, &i2s_config, 0, NULL);
i2s_set_dac_mode (I2S_DAC_CHANNEL_BOTH_EN );
i2s_set_adc_mode (ADC_UNIT_1, ADC_CH1);
i2s_set_adc_mode (ADC_UNIT_1, ADC_CH2);
i2s_adc_enable (I2S_NUM_0);
i2s_start (I2S_NUM_0);

//set in register numof samples per scan, channel pattern and numof channels
CLEAR_PERI_REG_MASK (APB_CTRL_APB_SARADC_CTRL2_REG, APB_CTRL_SARADC_MAX_MEAS_NUM_M);
SET_PERI_REG_MASK (APB_CTRL_APB_SARADC_CTRL2_REG,((NUM_SAMPS)<<APB_CTRL_SARADC_MAX_MEAS_NUM_S));

CLEAR_PERI_REG_MASK (APB_CTRL_APB_SARADC_SAR1_PATT_TAB1_REG, 0xffffffff);
SET_PERI_REG_MASK (APB_CTRL_APB_SARADC_SAR1_PATT_TAB1_REG, ADC_PATT);

CLEAR_PERI_REG_MASK(APB_CTRL_APB_SARADC_CTRL_REG,APB_CTRL_SARADC_SAR1_PATT_LEN_M);
SET_PERI_REG_MASK(APB_CTRL_APB_SARADC_CTRL_REG,((NUM_CHANNEL-1)<<APB_CTRL_SARADC_SAR1_PATT_LEN_S));
```

Figure 39: Initialization of both channels by the driver functions and modification of register settings afterwards

This is by no means an elegant solution, but due to the project deadline neither it was possible to wait for an extended driver version nor to rewrite the driver.

Generally, the ADC worked well with two channels, with a resolution of 12 bit and a total sampling rate up to 2 MS/s or 1 MS/s for each channel, but a problem occurred during the fast ADC sampling at the end of every scan cycle. After a scan cycle, which has up to 255 samples depending on the setting, the ADC restarts and skips 1 measurement. The minor problem is the loss of one sample, but the trickier problem is the change of the order of the channels. This problem can be solved by evaluating the additional channel information in the 16-Bit word of the samples and the missing sample is replaced by the mean of its neighboring values.

The data from the ADC data was presented in CSV-format and visualized as line diagram to verify the correct function. For the test a sine wave generated on DAC1 was used.

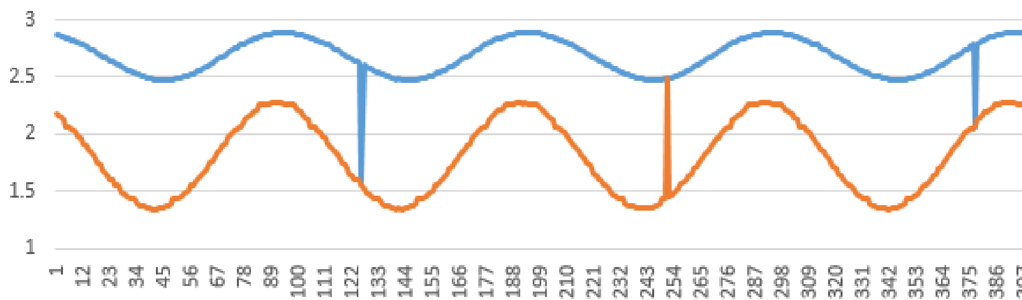


Figure 40: The ADC data was saved as CSV-file to verify the correct function. Here the scan cycle was 250, the missing-out of a sample was not properly corrected causing visible needles in the signal

A further limitation is the maximum number of buffers supported by the I2S driver. If it's required to work with a high sampling rate, a high number of samples needs to be buffered to measure a certain period. However, the driver allows to use only up to ten buffers with a maximum of 1024 samples each. Consequently, a sampling rate of only 200kS/s per ADC channel is used, although a sampling rate of 1 MS/s could be performed by the ADC.

3.2.2 Calculation of the impedance

For the measurement of a period of 20 milliseconds the ADC samples from both channels are stored in arrays with 4000 float values each before the calculation is executed. After sampling the two channels, the signals must be multiplied with a sine and cosine with the corresponding frequency to obtain the complex valued voltage and current.

The calculation of the sine and cosine was accelerated by using a look-up-table, which is initialized at the startup using the sine function from the math library. The calculation with complex numbers was facilitated by defining a class for complex numbers based on two doubles. The class overloaded the usual operations like multiplication, division etc. what allows to use the variables like any float or double. Consequently, the complex value of the voltage can easily be divided by the current, resulting in a complex value representing the impedance.

To test the robustness of the subsequent data evaluation, it can be useful to simulate noise in the measurement. The ESP32 contains a random number generator which can be used to add noise to the voltage or current measurement if it is required to obtain a signal with disturbances. The simulated noise can be activated by setting the noise level value to the corresponding parameters in the firmware.

3.2.3 Sensor calibration

There are two likely error sources in the sensor. First there will be an additional resistance in the cable and electrode and second the value of the shunt will not be exact. These two error sources can be corrected by multiplying the raw impedance with a complex factor $\overline{\mathrm{m}}$ and adding a complex offset $\overline{\mathrm{b}}$.

$$\bar{Z}_{Corrected} = \bar{Z}_{Raw} * \overline{\mathrm{m}} + \overline{\mathrm{b}}$$

The offset $\overline{\mathrm{b}}$ and factor $\overline{\mathrm{m}}$ and depend on the frequency and the selected shunt. These values can be determined by acquiring the raw impedance spectra of two accurate known resistances, a few times higher and lower than the applied shunt. In this work the calibration is realized by logging the raw spectra values in a CSV-file and subsequently executing a Python script which calculates the calibration parameters and saves them in arrays in a C-header file.

3.2.4 Real-time signal processing on the ESP32

After the calibration routine the phase and modulus can be calculated. Subsequently, these values must be normalized with given mean and deviation parameters before they are evaluated by the neural network.

For fast measurement the signal processing task was split in two parts and assigned to different cores. During the impedance measurement for each frequency the ADC samples are acquired, and 16000 multiplications and additions are executed to obtain the complex values representing the voltage and the current at the corresponding frequency. Core 1 of the ESP is exclusively assigned to acquire the complex voltage and current value and forwards the results through a queue to the second task on Core 0, where the impedance value is calculated. The second task handles accordingly the calibration routine, the calculation of the moduli and phases, and the execution of the ANN, which all in all is less time consuming than the calculation on Core1.

In total the solution requires three tasks. First, the signal processing task to acquire the ADC-signal and to calculate the complex voltage and current. Second, the task which calculates complex valued impedances, executes the ANN, and sends the results via UART. And third, a

task which can receive commands and respond through the UART, for example to start or stop the measurement process or to read and set parameters on the device.

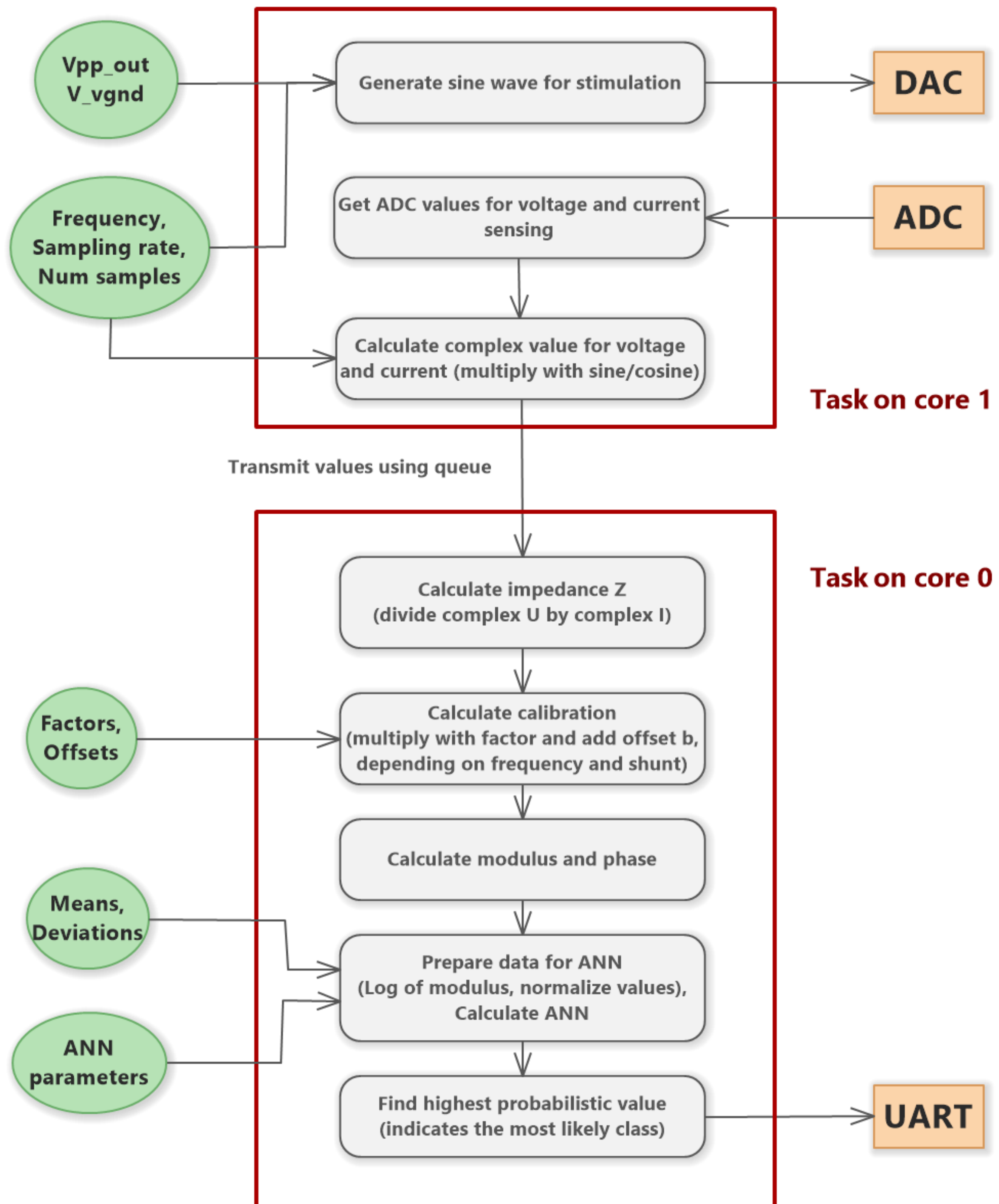


Figure 41: Dataflow of the signal processing algorithm split in two tasks

3.3 Communication between microcontroller and PC

3.3.1 Definition of the communication protocol

The ESP32-DEVKIT-C contains a USB-interface which is connected to the UART-Module, so it can receive and send data through a virtual COM-port to any program on a PC. This can be a common terminal program, or an application developed e.g., in MatLab, C#, LabVIEW, Python etc. In order to configure parameters, to acquire measurement data or to trigger a process, a communication protocol must be defined.

The protocol defined here consists of the following:

-**Start sign** to signal the start of a message

-**Keyword** to identify the required function

-**Optional parameters** if required by the function

-**End sign** to mark the end of the message

These components must be separated by a separator sign, here the whitespace is used.

Example message 1:

< SET_PARAM AdcSampRate 200000 >

Start sign:	**<**
Keyword:	**SET_PARAM**
Additional parameters:	**AdcSampRate** and **200000**
End sign:	**>**

In the previous example a function is called which saves the given value **200000** to the parameter identified by keyword **AdcSampRate**.

Example message 2:

< START_SPECTRO 100 300 1000 3000 10000 30000 >

This message starts the spectroscopy process with the given frequencies.

This protocol can easily be implemented in a program code, as well is easily readable. Thus, it can be used with a simple terminal program without the need for an extra application, what can be useful during the development. A binary data transmission would be faster than an ASCII-style protocol, however the throughput using ASCII-signs should be sufficient for the purposes of the project.

Acknowledge messages

If the function was executed successfully, an acknowledge message will be sent repeating the keyword with additional acknowledge sign **+**.

For example, **< +SET_PARAM >** will be sent if the corresponding command SET_PARAM was executed successfully.

For the cases where the transmission or execution of the command is not successful, a set of negatives acknowledge messages is defined:

Negative acknowledge	Reason/example
INVALID_SEQUENCE	Incorrect message e.g., missing start or end sign
UNKNOWN_COMMAND	Keyword is wrong, not found the list
INVALID_PARAMETERS	E.g., expected numbers but received letters
INCOMPLETE_PARAMETERS	Less parameters received as expected
NOT_IMPLEMENTED	Functionality not yet available
EXECUTION_DENIED	E.g., value exceeds limit
EXECUTION_FAILED	E.g., physical problem with peripheral

3.3.2 Implementing the communication module

To implement a task which receives and executes commands, it is a good practice to build a state machine which receives the messages. When the signs are received, they are written into a buffer and when the message is complete the corresponding function will be executed.

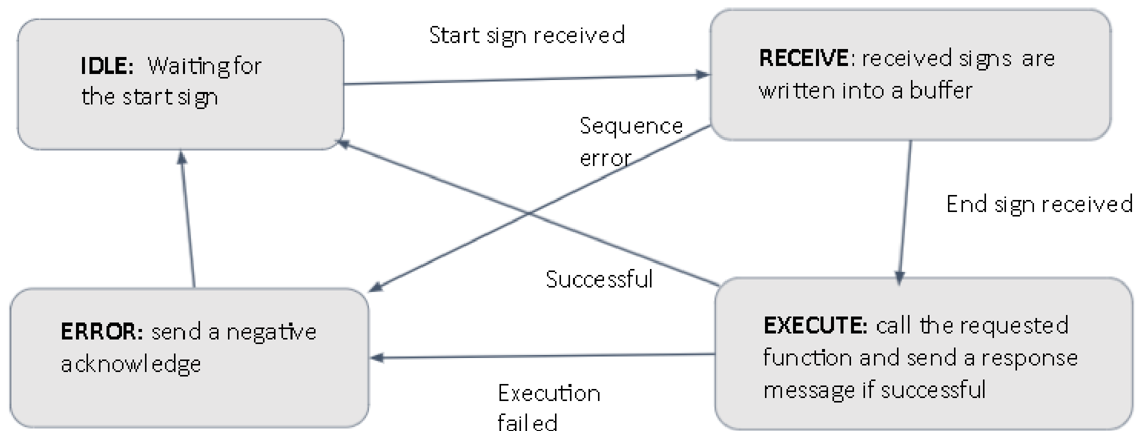


Figure 42: State machine to receive messages and execute the command

The commands and their corresponding function can be clearly listed in the source code using a table which is basically an array of a structure type containing an identifier, a function pointer, and more optional fields. When a command is received, the program searches the keyword in the table and if found, it will execute the assigned function through the function pointer and passes the string containing the optional parameters. For examples if the message **<START_SPEC>** is received, its corresponding function will be called, which sets an event flag to trigger the corresponding task to start the spectroscopy process.

3.3.3 Parameter handling

To manage the multitude of parameters, the source code of the firmware contains a table describing the global accessible parameters, where the size, type, and an identifier keyword for external access etc., is defined. These parameters will be listed by calling a corresponding command (**< PARA_LIST >**). The values are accessible from other firmware modules through read

and write functions, using an identifier listed in an enumeration. These functions encapsulate the usage of a mutex which is important to protect against errors by simultaneous access [17]. Additionally, the write function calls a callback if added in the parameter list, for example if the parameter selecting the shunt is changed, a function will be executed which accordingly changes the control signals of the shunt-multiplexer.

```
typedef struct Param_t
{
    e_ParamId_t e_ID;         //internal ID
    const char* str_ID;       //ID as string for external access
    void* p_Data;             //pointer to the corresponding data
    e_Datatype e_Dt;          //identifier for the datatype
    int i_size;               //number of bytes
    int (*WriteCallback) (void*); //pointer to function called after writing
    unsigned int ui_Options; //optional flags
} Param_t;
```

Figure 43: Data structure to describe device parameters

```
Param_t paralist[] =
{
//ID_enum,  ID_string,  datapointer,                         type_id,    numofbytes,                                afterwrite_callback
{V_VGND,    "V_VGND",   &EepromData.DSP_params.f_VoltVGND,   FLOAT_T,    sizeof(EepromData.DSP_params.f_VoltVGND),   InitDsp },
{V_OUT,     "V_OUT",    &EepromData.DSP_params.f_OutputAmp,  FLOAT_T,    sizeof(EepromData.DSP_params.f_OutputAmp),  InitDsp },
{ADC_SR,    "ADC_SR",   &EepromData.DSP_params.i_AdcSampRate,INTEGER_T,  sizeof(EepromData.DSP_params.i_AdcSampRate),InitDsp },
{SHUNT_SEL, "SHUNT_SEL",&EepromData.DSP_params.ui_ShuntSel,  INTEGER_T,  sizeof(EepromData.DSP_params.ui_ShuntSel),  InitShunt},
```

Figure 44: Table to manage the access to the device parameters

Many settings must be stored non-volatile. Accordingly, there is an instance of a structure which contains all non-volatile parameters. A copy of this structure will be loaded to the RAM from the EEPROM during startup and can be saved from RAM to the EEPROM by the sending the command **<SAVE_NVM>**. The default configuration can be reloaded from the flash into the RAM by the command **<RESET_PARAMS>**.

3.4 Implementing a feedforward ANN in C

3.4.1 Limitations of the ESP32

The used ESP32 is a dual core controller with floating point units, clocked with 200 MHz, thus it will be easy to execute the necessary mathematical operations in a reasonable time.

The main limitation of the ESP32 will be the size of the EEPROM if it is required to store the parameters of the neural network non-volatile and configurable. Since for different applications, different ANN architectures may be required with different number and width of layers, different activation functions etc., it is a clear benefit if these dimensions can be configurable. However, the EEPROM has a size of 4 Kbyte, and thus a maximum of thousand float parameters can be stored (if other device parameters are neglected), why it is not affordable to reserve the maximum number of parameters for all possible layers.

3.4.2 Solution for a customizable embedded neural network

The applied solution is to store all tunable parameters next to each other in one big float array and additionally to place the hyperparameters for all layers in an array of unsigned integer, where each unsigned integer defines the dimensions and activation function of one layer. The hyperparameters of the layers are defined by unsigned integer values from the first value of the array until the value 0 marks the end of the valid layers.

The unsigned integer defines the hyperparameters of a layer as follows:

Bit 31 to 24	Bit 23 to 16	Bit 16 to 8	Bit 7 to 0
Number of input units	Number of output units	Number of parameters used by the activation function	ID of the activation function

The additional parameter of the activation function is in the most cases not tunable, except for some activation units with tunable parameters like **PRelu**. But to keep it simple and the set of possible activation units extendable, it is placed in the float array together with the tunable parameters.

Example 1: If the first unsigned integer value is **0x0A 06 01 02**, it defines a layer with 10 inputs, 6 outputs and the activation with ID 2 (here it's the LeakyReLu) which uses one parameter. Consequently, 60 float values of the tunable parameter array are the weight matrix, followed by 6 float values for the bias vector and finally one float for the activation function. The following values are the parameters of the next layer.

Example 2: If the values in the hyperparameter array are **0x0C 06 01 02**, **0x06 01 00 01** and **0x00 00 00 00**, it defines first a layer with 12 inputs and 6 hidden outputs with LeakyReLU using one parameter, followed by a layer with 6 inputs and one output with activation ID 1 (Sigmoid) which does not use any optional parameter. The parameters in the float array would be arranged beginning with the weight matrix of the first layer, followed by the bias vector, afterwards the parameter for activation function, followed by the weights of the second layer and its biases.

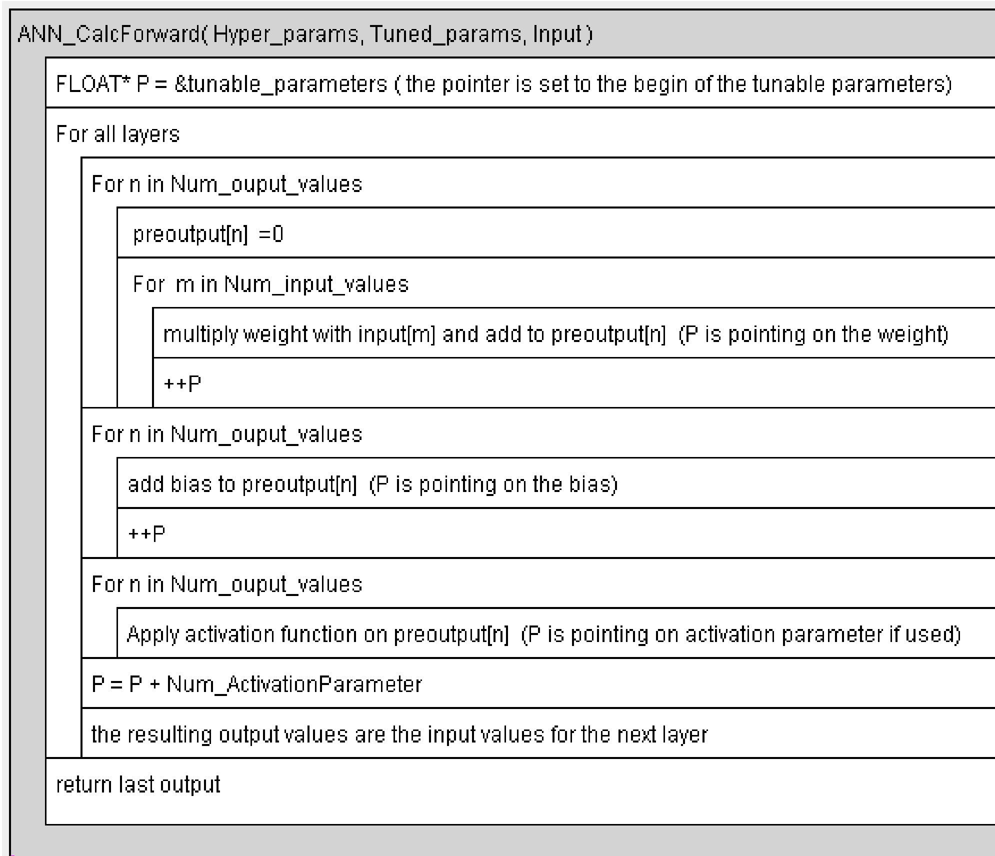


Figure 45: Algorithm of the embedded ANN

The weight matrices will be stored in an order where the first value connects the first input with the first output, followed by the weight that connects the second input with the first output etc. The algorithm uses a pointer iterating over the float array. In the implementation, the loops realizing the multiplication by the matrix, the addition of the bias and the calculation of the activation function are united in the function *CalcLayer*.

3.4.3 Generating default value code and module testing

It is quite difficult to handle the parameters of the solution manually, that's why a Python script is used to generate a C-Header containing the arrays with the necessary coefficients based on a trained ANN saved in a JSON-file.


```
{
"input_mean": [7.71261774930033, 7.231065239854747, 6.761158601184784, 6.285148
"input_dev": [0.31149276465377945, 0.3586320732665049, 0.3970384774103023, 0.45
"ANN": "[{
    "descr\": \"Classification KCl \",
    "dimensions\": [20, 8, 8, 9, 10, 11]},
    "{  \\\
        "WEIGHTS\\\": [[0.07394675275316837, 0.22716017920552417, 0.0693075896
        "BIAS\\\": [0.9178190022718774, 0.7941332720764613, 0.9400755655650046,
        "ACTFUNC\\\": \\\"[\\\\\\\"ActFunc.relu\\\\\\\", [0.01]]\\\
        "}\",
    "{  \\\
        "WEIGHTS\\\": [[0.22012883947828057, 0.2566523593658969, 0.225637921624
        "BIAS\\\": [-0.03338494770993346 4, -0.04615120351758146 6, -0.061027079
        "ACTFUNC\\\": \\\"[\\\\\\\"ActFunc.elu\\\\\\\", [0.3]]\\\
        "}\",
    "{  \\\
        "WEIGHTS\\\": [[0.08687902527107401, 0.0524552969999225, 0.34083322283
        "BIAS\\\": [-1.6313652063048827, -1.471206867433671, -0.43528874556322
        "ACTFUNC\\\": \\\"[\\\\\\\"ActFunc.elu\\\\\\\", [1.0]]\\\
        "}\",
    "{  \\\
        "WEIGHTS\\\": [[0.5095961283655647, 0.4634586173380019, 0.344331677654
        "BIAS\\\": [-1.0941137803905596, -0.7263313234666564, -0.10101073127403
        "ACTFUNC\\\": \\\"[\\\\\\\"ActFunc.relu\\\\\\\", [0.2]]\\\
        "}\",
    "{  \\\
        "WEIGHTS\\\": [[1.1401852422617962, 0.8464651386349568, 0.710158570557
        "BIAS\\\": [-4.055573199681548, -1.7376144997284924, -1.01702872371192
        "ACTFUNC\\\": \\\"[\\\\\\\"ActFunc.softmax\\\\\\\", []]
        "}\"]",
"LABEL": ["<=0.5%", "0.5-1.0%", "1.0-1.5%", "1.5-2.0%", "2.0-2.5%", "2.5-3.0%",
```


Figure 46: Snippet of JSON-file containing the information to run the ANN

Additional to the UINT array for the hyperparameters and the FLOAT array for the tunable parameters, the header contains arrays with the mean and deviation values to normalize the inputs. All these values will be stored as default values in the EEPROM.

Testing and debugging a complex module like the ANN on a microcontroller can get very exhausting and costly. To facilitate the development, it’s a good practice to run the module in a simple PC program executing test cases. The existing reference model defined in Python can be used to generate test vectors, random input vectors with corresponding results calculated by the ANN, which will be compared to the results of the embedded ANN. This approach facilitates testing and thus accelerates modifications of the code or the development of extensions.

For instance, an unexpected error occurred in the activation function Softmax. Although the implementation in C executed the same formula as the ANN in Python, numerical problems arose caused by too high values in the denominator and divisor, which caused invalid results. This error could be reproduced on the PC and the problem was corrected.

4 Software GUI

4.1 Principal functionalities

The developed sensor shall be configurable through a graphical user interface (GUI), which also shall represent the acquired data.

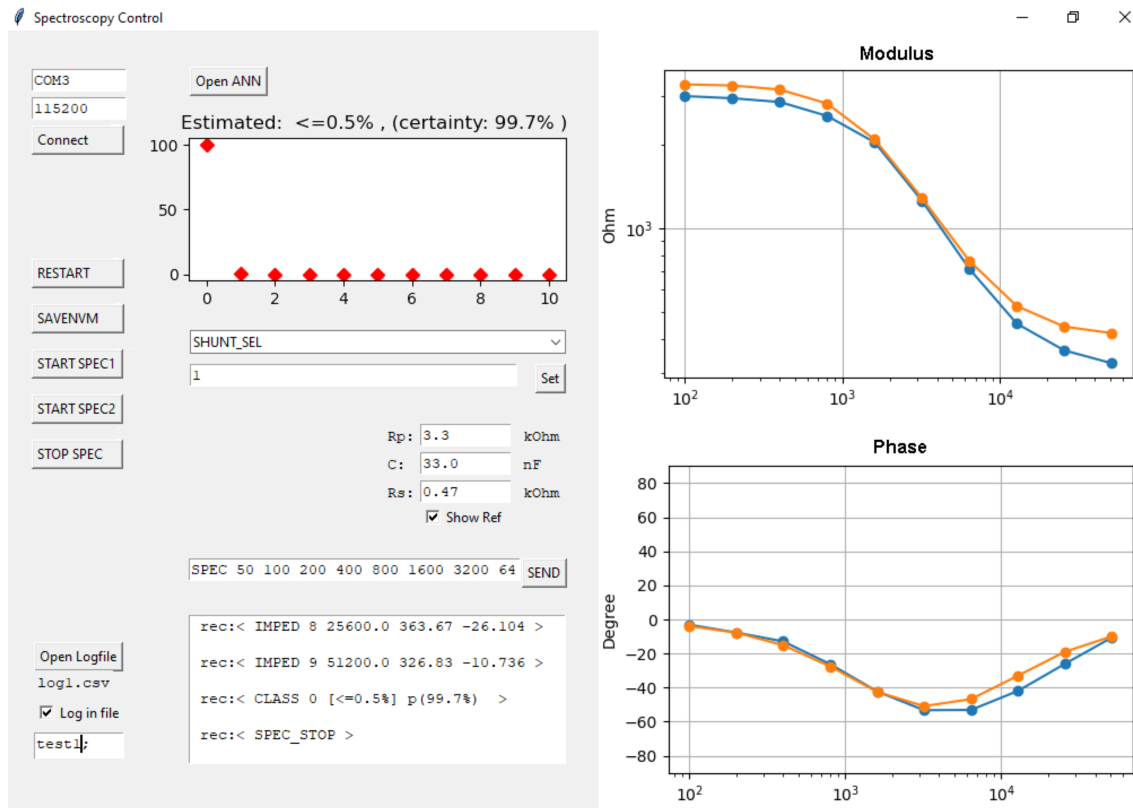


Figure 47: Graphical user interface to control the sensor

The most important functionalities are:

-Establishing a connection with the device through a virtual comport

-Plotting the received impedance spectra

-Plotting the spectrum of a theoretical RCR-circuit to compare visually

-Write received spectra in a log-file in CSV-format

-Read and write the available parameters of the device

-Execute an ANN which can be loaded as JSON-file to evaluate spectra and plot its result

-Offer buttons to send often used commands e.g., **< RESTART >, < SAVE_NVM >**

4.2 Features of Python and how they are applied

Python code can be executed as script using a Python interpreter, but it is also possible to create an executable file from the source code using the program **Pyinstaller**. This exe can also run on a PC without Python installation.

There are many useful libraries available for Python and for the GUI the following are interesting:

Matplotlib is a library which allows to visualize data. In the GUI it is used to plot the phases and moduli of the spectra and to visualize the result of the classification.

Tkinter is a package to create GUIs with buttons, text fields etc.

Serial is the module with the necessary functions to access the virtual comport.

The library **Threading** offers the possibility to run several threads which here is very useful, because it is required to receive simultaneously incoming data from the virtual comport, as well to run simultaneously the GUI which reacts on clicks on buttons etc.

Numpy is a library to handle numerical problems like operations with matrices what is used in the applied neural network.

The module **Json** is used to work with JSON-files. These files can be used to save data in a structured way, like INI-files or XML-files. In this project, the neural network possesses functions to save all its parameters in a JSON-file and to load networks from saved files.

Next to these libraries Python offers further features:

Python **Lists** allows to conveniently store data of different size and type and to access through an indexer like an array. Very useful here is the function which splits strings by a separator into a list of strings. This is used to handle the incoming messages and to separate the keyword from the optional parameters.

Python **Dictionaries** can be used to assign an identifier to any object [18]. Here it is used to assign a string containing the message keyword as identifier to the function which shall be executed. When a message from the device is received the corresponding function will be called and the additional parameters will be passed as a list of strings to the function.

In Python functions can be stored and passed as arguments like an object and **Lambda expressions** offer the possibility to define a function and directly to pass them without defining a function name, with the intention to avoid unnecessary amounts of code by the definition of functions [19]. In the program this feature is used to define buttons which send a specific command to the device with only two lines of code.

```
#Restart button
 self.B_RESTART = Button(self.root, text="RESTART", command= lambda: self.com.SendCommand("RESTART"))
 self.B_RESTART.place(x=25, y=120)

#SAVENVM button
 self.B_SAVE = Button(self.root, text="SAVENVM", command= lambda: self.com.SendCommand("SAVE_NVM"))
 self.B_SAVE.place(x=25, y=160)
```

Figure 48: Usage of lambda expressions

The constructor of the button obtains a function as object which will be executed when it is clicked. Instead of defining for every button several functions containing different code like '*SendCommand("SAVE_NVM")*' or' *SendCommand("RESTART")*' etc. the functions are defined as nameless lambda expressions.

4.3 Communication between GUI and device

The GUI shall be able to read and write parameters on the sensor. After startup, the device sends a message (**<READY >**) to advise to be available. Afterwards the program asks the device to send the ID of all available parameters (**<PARA_LIST >**) and the device responds with messages containing the parameter identifier, type, and size in bytes for every item (**<PARAM *paramID type size* >**). The GUI saves the list with identifiers and offers a combobox to access the items. If an item from the combobox is selected, the GUI sends the command to read the parameter (**<GET_PARAM *paramID* >**) and shows the value in the text field after receiving the response (**<PARAM_VAL *paramID value* >)**. If the button "Set" next to the text field is clicked, the value in the text field will be written to the parameter chosen in the combobox (**<SET_PARAM *paramID value* >**).

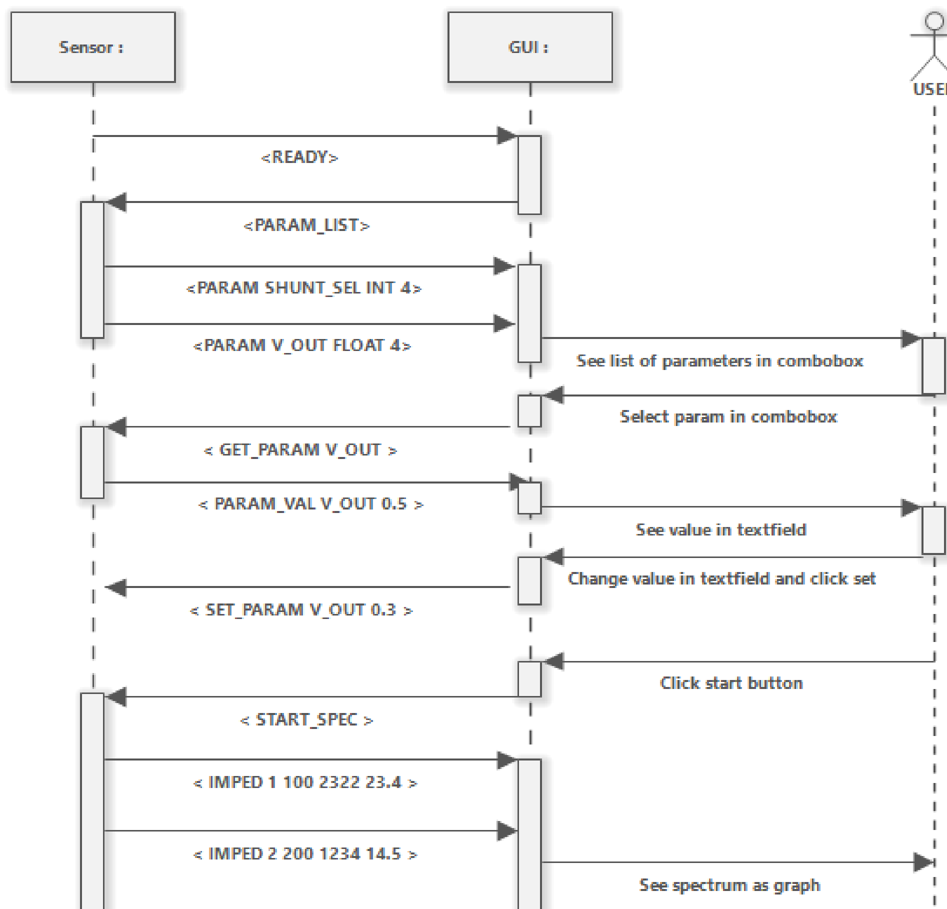


Figure 49: Handshake between sensor and GUI

An advantage is that there is no need for an update of the GUI when a new parameter is added to the firmware since the GUI obtains the necessary information during runtime.

The impedance spectrum is transmitted in separate messages for every frequency.

< IMPED N f Modulus Phase >

With N as index, and f the frequency.

Example: < IMPED 0 100.0 2330.1 -3.4 >

After receiving the message, it is split by the separator sign, here SPACE, and subsequently the dictionary is asked for the keyword "IMPED", the corresponding function in the dictionary is called and get passed the list with the added values, to plot the values in the diagram or to log them in a CSV-file etc.

5 Programming a neural network

5.1 Supervised learning

The machine learning paradigm applied in this project is supervised learning, one of the three machine learning paradigms together with unsupervised and reinforcement learning, which are not described here. This technique basically requires the following:

1. Labeled data
   The idea of machine learning is to learn from data and in the case of supervised learning it's necessary to have labeled data, examples of input data together with their desired output. An example could be certain properties of houses and their price as label. These data could be used to learn to estimate the price of other houses.

2. Trainable model
   The goal of the learning process is to obtain a function which deduces from the input the most likely output. There is the need for a parameterizable function that can be tuned to estimate the output as correctly as possible.

   $$\vec{y} = f(\vec{x}, \Theta)$$

   With $\vec{\boldsymbol{x}}$ as input vector, $\boldsymbol{\Theta}$ the tunable parameters and $\vec{\mathbf{y}}$ the resulting output.

   The trainable model used in this project is the feedforward neural network, where the weights and biases are the tunable parameters. There are more machine learning models for supervised learning, like recurrent neural networks, random forest, or support-vector machines, which are not discussed in this work.

3. Loss function
   To improve the tuning of a model, it is necessary to measure the correctness of the estimated output. There should be a differentiable function which compares the output of the model with the desired output given in the data label, an error function, also called loss function. Often used loss functions are the mean-squared-error, usually used for regressions, the cross-entropy usually used for classification and the binary-cross-entropy used for binary classification. See appendix for the definition of commonly used loss functions.

4. Training algorithm
   Finally, there is the need for an algorithm which adapts the parameters of the model using the training data to minimize the mean value of the loss on the training data. One of the most used optimization algorithms is the gradient descent algorithm, which requires to determine the derivatives of the mean loss with respect to the tunable parameters of the neural network.

5.2 Training of a neural network

One time the training data is available, and the model and loss function are defined, it's necessary to optimize the parameters of the model in a way it minimizes the mean loss value for the whole training data set. The usual optimization strategy is the gradient descent algorithm, and the backpropagation algorithm is used to determine the gradient of the loss.

5.2.1 Backpropagation

The optimization process uses the gradient of the loss, thus the first order derivatives of the loss value with respect to all tunable parameters must be estimated. To obtain the derivatives, the output for one training sample is calculated and subsequently the derivatives of the loss with respect to the network outputs. These values indicate how much the outputs influence the loss and can be used, considering the derivative of the activation function, to calculate the influence of the last layers' biases and weights on the loss function. Additionally, the influence of the input of the last layer can be determined, which is the influence of the output of the second last layer. This can be used to calculate the derivative of the loss with respect to the weights and biases of the second last layer and so on. Using this principle, the algorithm can iterate from the last up to the first layer, so the influence on the output error can be “back propagated” to obtain the derivative with respect to the tunable parameters of all layers. This algorithm is an application of the chain rule for derivation.

For example, a very simple network with three layers transmitting only one value:

$$Y_0 = f_0(B_0 + M_0 * X)$$

$$Y_1 = f_1(B_1 + M_1 * Y_0)$$

$$Y_2 = f_2(B_2 + M_2 * Y_1)$$

With X as the input and Y_2 the final output.

The loss function, for example, is the square of the error:

$$Loss = (Y_2 - Y_{2des})^2$$

And accordingly:

$$Loss = (f_2(B_2 + M_2 * Y_1) - Y_{2des})^2$$

For the square of error, the derivative of the loss with respect to Y_2 is:

$$\frac{dLoss}{dY_2} = 2 (Y_2 - Y_{2des})$$

Applying the chain rule to get the derivatives with respect to B_2, M_2 and to the layer input Y_1:

$$\frac{dLoss}{dB_2} = \frac{dLoss}{dY_2} * f'_2(B_2 + M_2 * Y_1)$$

$$\frac{dLoss}{dM_2} = \frac{dLoss}{dY_2} * f'_2(B_2 + M_2 * Y_1) * Y_1$$

$$\frac{dLoss}{dY_1} = \frac{dLoss}{dY_2} * f'_2(B_2 + M_2 * Y_1) * M_2$$

With f'_2 as the derivative of the activation function of the corresponding layer.

The influence of the layer input $\frac{dLoss}{dY_1}$ is used to calculate the derivatives of the previous layer:

$$\frac{dLoss}{dB_1} = \frac{dLoss}{dY_1} * f'_1(B_1 + M_1 * Y_0)$$

$$\frac{dLoss}{dM_1} = \frac{dLoss}{dY_1} * f'_1(B_1 + M_1 * Y_0) * Y_0$$

$$\frac{dLoss}{dY_o} = \frac{dLoss}{dY_1} * f'_1(B_1 + M_1 * Y_0) * M_1$$

And accordingly, $\frac{dLoss}{dY_o}$ will be used to calculate the derivatives of the first layer.

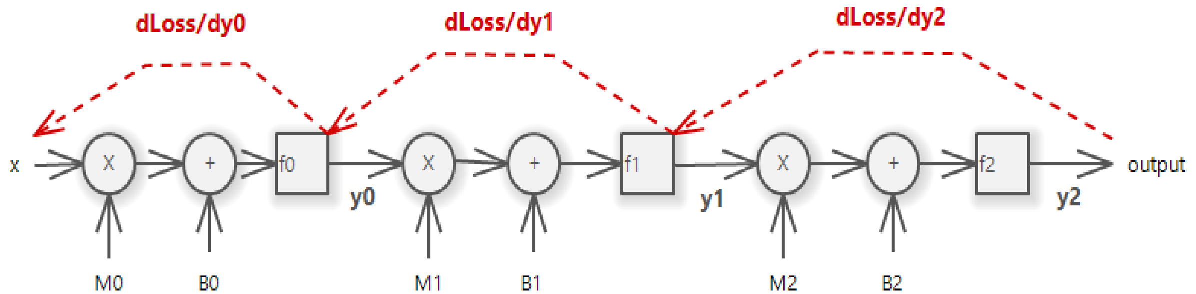


Figure 50: The error is back propagated

Unlike this simple example, the layers have usually more than one output, what complicates the algorithm slightly.

In literature often the whole optimization process is called backpropagation, however exactly it refers only to the algorithm to obtain the gradient values [20].

5.2.2 Gradient descent

After calculating the gradient of the loss, the mean of the derivatives of the loss function for the training data set, it can be used to estimate how to change the parameters to get to a lower loss value. The gradient indicates how much each parameter increases the function and to approximate a lower value, the gradient descent subtracts iteratively a vector proportional to the gradient from the model parameters.

$$\theta_{n+1} = \theta_n - \alpha \frac{dLoss}{d\theta}$$

Here the value α is the step size, an arbitrary training parameter with a value usually around 0.1. A good setting of the step size is important to obtain a good result, if the value is too small, it could get stuck in a local minimum instead of finding a better tuning, but if the step size is too large the process won't settle at a minimum.

5.2.3 Stochastic gradient descent and mini-batch descent

To obtain the gradient of the loss on the whole training data set, it could calculate the backpropagation algorithm for all data samples and calculate the average of all obtained results and iterate one step of the gradient descent and afterwards calculate the gradient over the whole training data again. The process to calculate the gradient on every data sample would be quite time consuming. With the intention to speed it up, the algorithm can randomly choose only one sample, calculate the backpropagation, and execute the gradient descent iteration. Instead of moving slowly towards a minimum, the parameters will move stochastically and probably will be attracted by a minimum. Instead of using a single data sample, a batch of randomly chosen samples from the training data set can be used to calculate the gradient which should result in a sufficiently precise estimation of the gradient on the whole data set. For one epoch of the mini-batch optimization, the data set must be shuffled and be split in batches which are used to iterate to the minimum. When all batches are processed the next epoch begins, and the data must be shuffled again. This process repeats until the loss is minimized sufficiently.

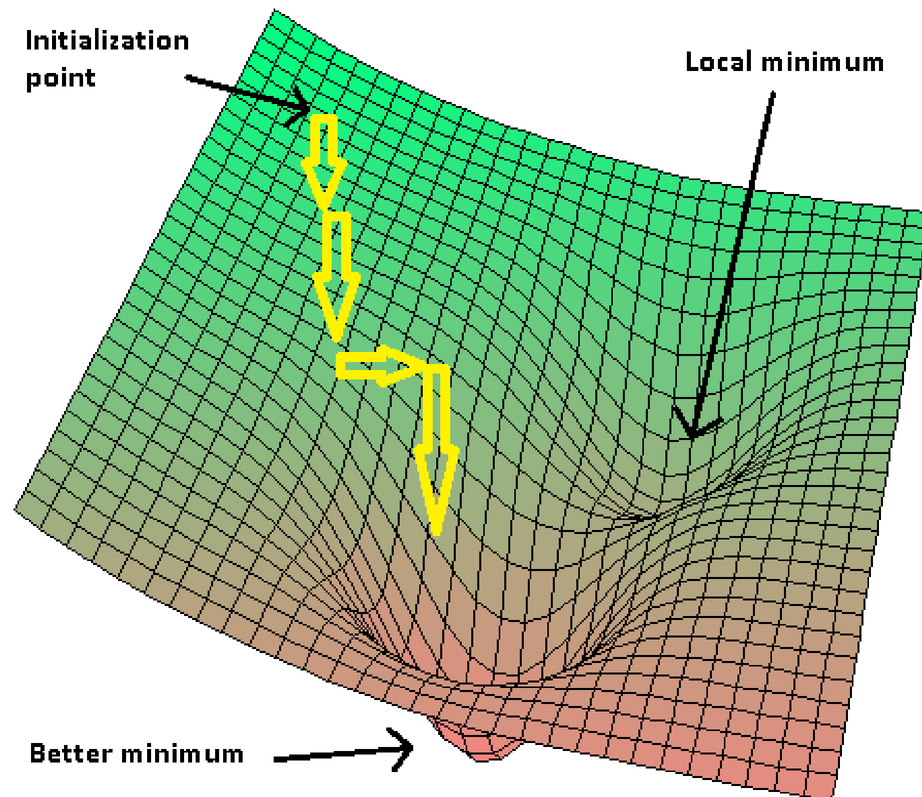


Figure 51: Principle of gradient descent

5.2.4 Applying momentum

The gradient descent algorithm has the problem that it possibly remains in a local minimum instead of moving on towards a lower minimum. To avoid this problem, the gradient doesn't determine the step directly, but is added to a “momentum”. The momentum is an idea from mechanics and can be imagined like a ball rolling down a hill which is accelerated by gravity but keeps its momentum when it rolls into a hole which forces it to jump out. Applying this principle, it appears like the gradient is damped:

$$Mom_{n+1} = \beta \frac{dLoss}{d\theta} + (1-\beta)\, Mom_n$$

$$\theta_{n+1} = \theta_n - \alpha\, Mom_n$$

Here α is the already mentioned step size and β the newly introduced momentum factor. A usual value for β is 0.1, the value must be between 0 and 1, if it is 1 the momentum is not applied at all.

5.3 Neural network in Python

Python is one the most popular programming languages in data science and machine learning. Some of its benefits are dynamical typing and object-oriented programming. Another feature in Python are the list types where data items of different size and different types can be stored, which can be very useful to stack up neural networks. There are also a lot of useful libraries available, very important the library Numpy, enabling to work with matrices and to solve various numerical problems, like in Matlab.

5.3.1 Object-oriented design of an artificial neural network

Object-oriented programming can be very useful to reuse similar structures like a neural layer and to obtain an easily extendable and modifiable solution. Building a neural network can be realized using the following three classes:

1. **ActivationUnit** interface and its subclasses
   Neural networks use different types of non-linear activation units. Additionally, to the activation function it is necessary to get the derivative for the backpropagation algorithm, so it's useful to unite the function and its derivative in one class. There are various types of activation units, but they all have in common to calculate a function and its derivative, so it makes sense to create an interface called "ActivationUnit" which the specific activation subclasses, like LeakyReLU, Sigmoid, Softmax, inherit. Every specific activation unit type has its own constructor to instantiate and some of them also set coefficients, like the leak coefficient in LeakyReLU. Important functions of the interface are:

   -CalcOutput(X)

   Function to calculate the activation output from the sum of weighted inputs and bias.

   -CalcDerivative(X)

   Function to get the derivative of the activation function, necessary for the backpropagation algorithm.

   -GetDescr():

   Function returning a string in JSON-format to identify the type and the optional parameter values, which is later used to save the ANN configuration in a JSON-file.

2. **NeuralLayer** class
   An activation unit can be united with a weight matrix and a bias vector, resulting in a neural layer. This class contains the following functions:

   -NeuralLayer(NumIn, NumOut, ActivationUnit)

   The constructor to instantiate an object of the layer class with a given input size, output size and with an instance of an activation unit. This function also initializes the weight matrix and bias vector with appropriately distributed random numbers.

   -CalcForward(InputVector)

   This function multiplies a given input with the weights, adds the biases and subsequently applies the activation function and returns the resulting output vector. It is used to execute the forward calculation of the ANN.

   -CalcBackpropagation(LossSensitivity)

   This function is used to calculate the derivatives of the loss with respect to the weight and bias values in the layer. To calculate them, the derivative of the loss with respect to

layers output value is given from the next layer and the derivative of the activation unit is considered. The function adds the calculated values to the sum of the derivatives from the samples of the batch, which is later used to estimate the gradient, and returns the derivative with respect to the input of the layer, which will be used by the backpropagation iteration of the previous layer.

-UpdateParameters(TrainingCoeffiecients):

One time the gradient of the loss is estimated from a batch of training data, it will be used to adapt the parameters to get to a lower loss value and obtain better results. This function uses the principle of gradient descent, improved with the idea of the momentum. The function gets as input the training coefficients, for example the step size of the descent and the momentum coefficient and corrects the tunable parameters with the gradient previously estimated by the backpropagation algorithm.

-GetJsonDescription(), LoadJsonParams(string_JSON):

Additionally, there is the need for some utility functions. It is very useful to convert the parameters of the layer into a string (in JSON-format), which allows to save the trained ANN to a file, and correspondingly a function to reload the parameter again.

3. **NeuralNetwork** class
Several instances of the NeuralLayer class can be stacked up to build the neural network. The class containing the layers provides the following functions:

-NeuralNetwork(NumInput)

The constructor of the neural network sets the input size of the first layer. After calling the constructor the obtained ANN doesn't contain any layer.

-AddLayer(NumOutput, ActivationUnit)

This method adds a layer with a given output size and a given activation unit. The input size of the new layer will automatically be set to the output size of the previous layer and in the case, it is the first layer, it is set to the input size of the ANN.

-CalcForward(InputVector)

The principal task of the neural network is to process the input data and to return an output. This is done by iteratively calling the *CalcForward* method of every single layer, every time passing the result from the previous layer as input for the next layer.

-TrainNetwork(BatchLabeledData,…)

The ANN can´t be used before it was trained with the training data set. To train the network, a batch of labeled data is used, a list of data which contains input values and corresponding expected output values. The function calculates the network output and the loss derivative and calls iteratively the backpropagation method from the last layer up to the first layer for all samples in the batch to get the average of the loss derivatives. After the gradient of the loss is estimated, the *UpdateParameters* functions of all layers are called, executing the gradient descent. The function realizes one iteration of the optimization and must be executed several times with changing data batches until the network is fitted to the training data set.

-SaveToFile(Pathname), LoadFromFile(Pathname)

After obtaining a trained network, it should be saved in a file, which later can be applied in a specific application or can be loaded as a pretrained network.

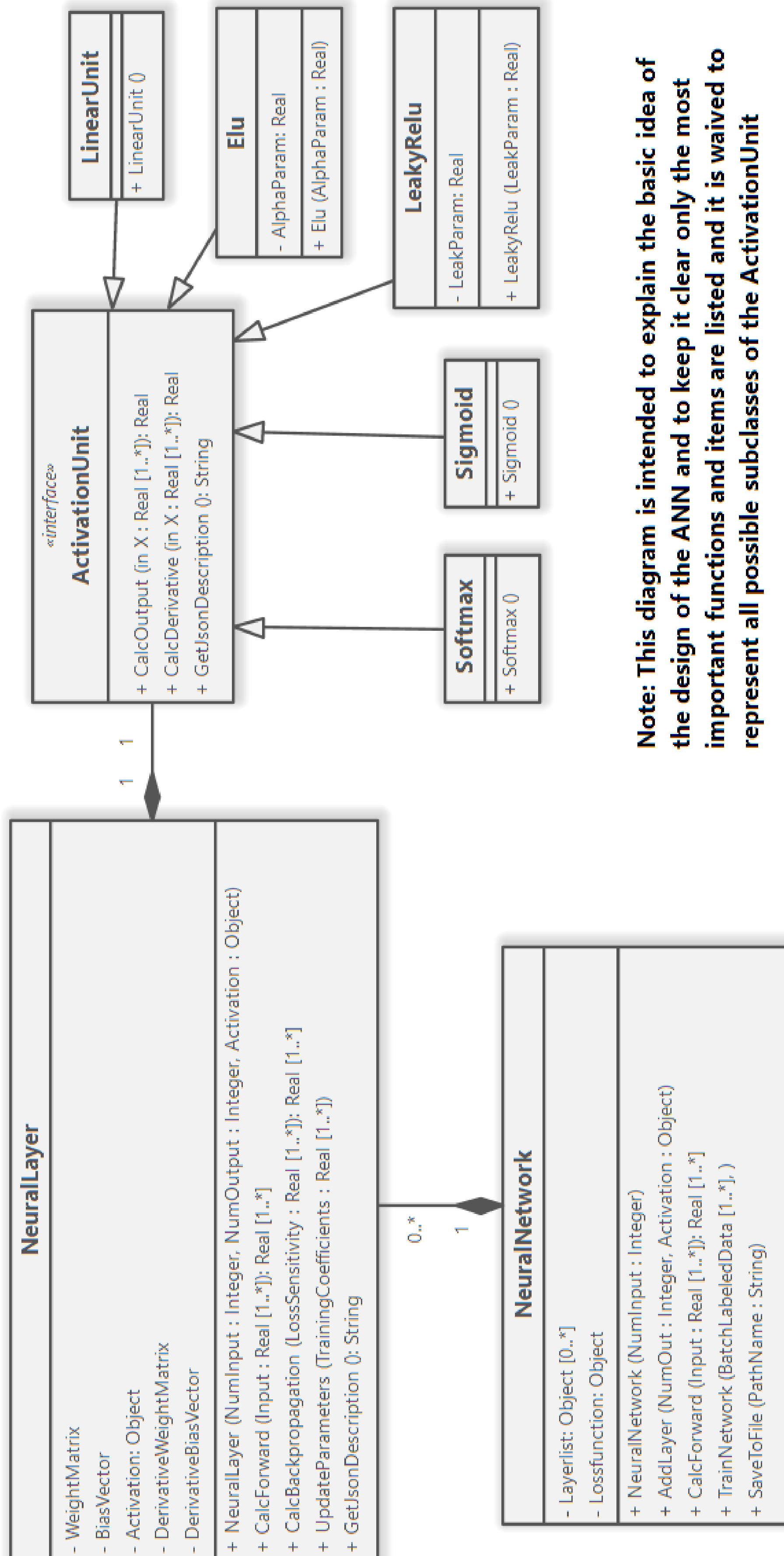


Figure 52: Object-oriented design of a neural network

5.3.2 Experiment 1: Reconstructing a signal

The practicability of the neural network can be proven by cleaning an erroneous signal. In a simulation a set of signals can be generated, for example by superposing several sine waves with random frequencies, random phase shift, randomly varying amplitudes etc.

Additionally, disturbances can be generated, for example normal distributed noise or occasionally occurring outliers or the combination of both and can be added to the original signal to simulate a disturbed signal. The disturbed signal as input in combination with the original signal as desired output is used to train the ANN to estimate the original signal, with the intention to remove errors from signals.

As first step the ANN must be constructed:

```
import NeuralNetwork as nn
import ActivationUnit as au

#constructing the ANN

CleaningANN = nn.NeuralNetwork( 25 )    # ANN with 25 inputs

CleaningANN.AddLayer(100,au.LeakyRelu(0.01))

CleaningANN.AddLayer(100,au.LeakyRelu(0.01))

CleaningANN.AddLayer(25,au.Linear())   # 25 linear outputs
```

Figure 53: Python code to construct the ANN

The constructed neural network instance has two hidden layers using the activation function LeakyReLU with a leak coefficient 0.01 and dimensions 25 to 100 to 100 to 25.

LeakyReLU (Leaky rectified linear unit) is a function defined in sections:

$$LeakyReLU(x, \alpha) \;=\; if\ x < 0 \colon \alpha x\ otherwise \colon x$$

Where α is the leak coefficient typically around 0.01. This leak facilitates the training since the function has always derivative above zero.

For a regression where a real value must be estimated like in this case, the linear unit is appropriate for the output and the least-mean-square-error the used loss function.

```
for m in range(2000):
    data_batch = []
    #generate batch
    for n in range(20):
        original_sig = sg.GenerateSignal(siglen)
        noise = ng.GenerateNoise(siglen)
        #obtain disturbed signal
        noisy_sig = original_sig + noise
        #[ input, expected_output ]
        data_batch.append([noisy_sig,original_sig])
    #fit the network to the given labeled data
    loss = CleaningANN.TrainNetwork(traindata=data_batch,stepsize =0.05,momentum=0.1)
    print( 'Iteration: ' , m , ' Loss: ',loss)
```

Figure 54: For-loop in Python to train the neural network

Here the previously constructed ANN will be fitted during 2000 iterations with data batches of 20 training samples, using a step size of 0.05 and a momentum factor of 0.1.

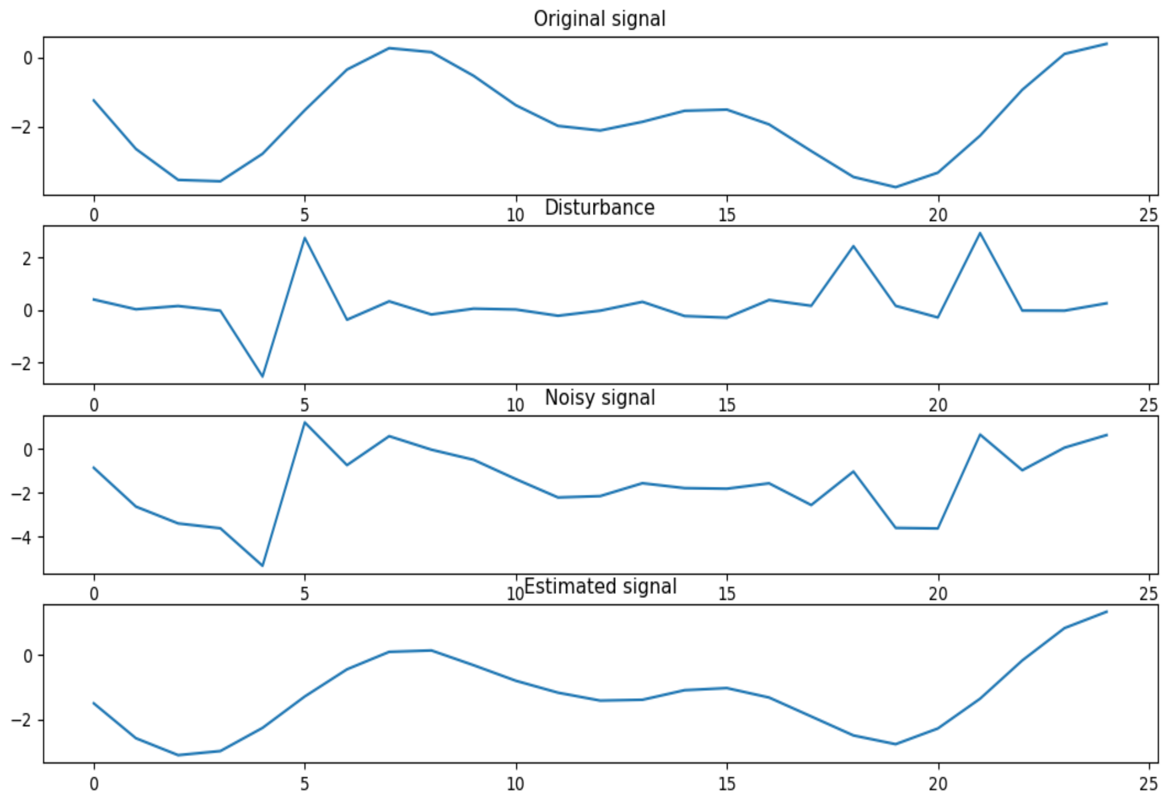


Figure 55: The resulting ANN learned to reconstruct the original signal

The result shows the capability of ANN to remove disturbances from measurements.

5.3.3 Experiment 2: Classifying capacitors

Removing faulty data from signals is by no means the only application of an ANN. It can learn any non-linear function, provided there are sufficient hidden units. For example, they can learn the relation of the capacitor in an RCR-circuit and its corresponding impedance spectrum.

The measurement of the impedance spectrum of an RCR-circuit can easily be simulated by calculating the theoretical spectrum and adding generated noise.

In this example an ANN with three layers and dimension 12 to 12 to 8 to 10 is created, which receives six modulus values and six phase values, in total 12 inputs.

```
# Construct the Neural Network
ANN_C_Estimator = nn.NeuralNetwork( 12 )
ANN_C_Estimator.AddLayer(12,au.LeakyRelu(0.01))
ANN_C_Estimator.AddLayer(8,au.LeakyRelu(0.01))
ANN_C_Estimator.AddLayer(10,au.Softmax())
```

Figure 56: Python code to construct the network

The hidden units use the LeakyReLU function like in the previous example, but the output unit uses the Softmax unit. In multicategory classifications where probability values are assigned to several categories, the Softmax function which provides a probabilistic output is the unit of choice.

$$Softmax_k(x_1, x_2, \dots, x_n) = \frac{e^{x_k}}{\sum_{i=1}^{n} e^{x_i}}$$

The output of the Softmax function is always between 1 and 0 and the sum of all units is 1, thus it can be interpreted as a probability distribution.

Important is also the selection of an appropriate loss function, and in the case of a multi-category classification it is the cross-entropy, also called log-loss, to compare the probabilistic output with the data label:

$$H(p,q) = -\sum_{x \in X} p(x) \log\left(q(x)\right)$$

Here $q(x)$ marks the probabilistic output and $p(x)$ the label value. This loss functions uses the logarithm and thus it results in a very high value when the output is completely wrong, e.g., when it is expected to be one but is close to zero. This accelerates the learning process and results in a better classification.

The data from the spectrum and the classification value must be converted before they can be used by the ANN. The logarithm of the modulus value of the spectrum should be used because the value can move in quite different magnitudes. The input data also must be normalized, that is, the mean values over the training set must be subtracted from the inputs and afterwards the inputs must be divided by the deviation. The resulting normalized inputs should have a mean of 0 and a deviation of 1, which avoids numerical problems and facilitates the training.

The output of the classification is not represented as one value but by a vector of probabilities assigned to the different classes. Consequently, the label value must be a one-hot vector for the corresponding category, a vector where all values are 0, except the value signaling the correct class is 1, e.g., [0,1,0,0] signals the second class is the expected.

To test the applicability of this ANN, it was executed in the GUI, correctly estimating the capacitor in an RCR-circuit using spectroscopy data. The connected Rp and Rs was realized with potentiometers and their value could be changed within the range the ANN was trained, without falsifying the classification.

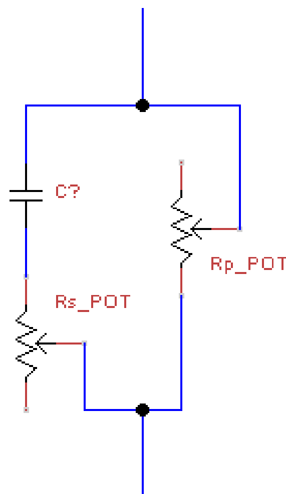


Figure 57: Circuit to test the capacitor classification with cross influences (Rp, Rs)

Obviously, the ANN can handle the cross influences on the spectrum, in this case Rp and Rs, and thus clearly determines the correct capacitor even if both resistances vary.

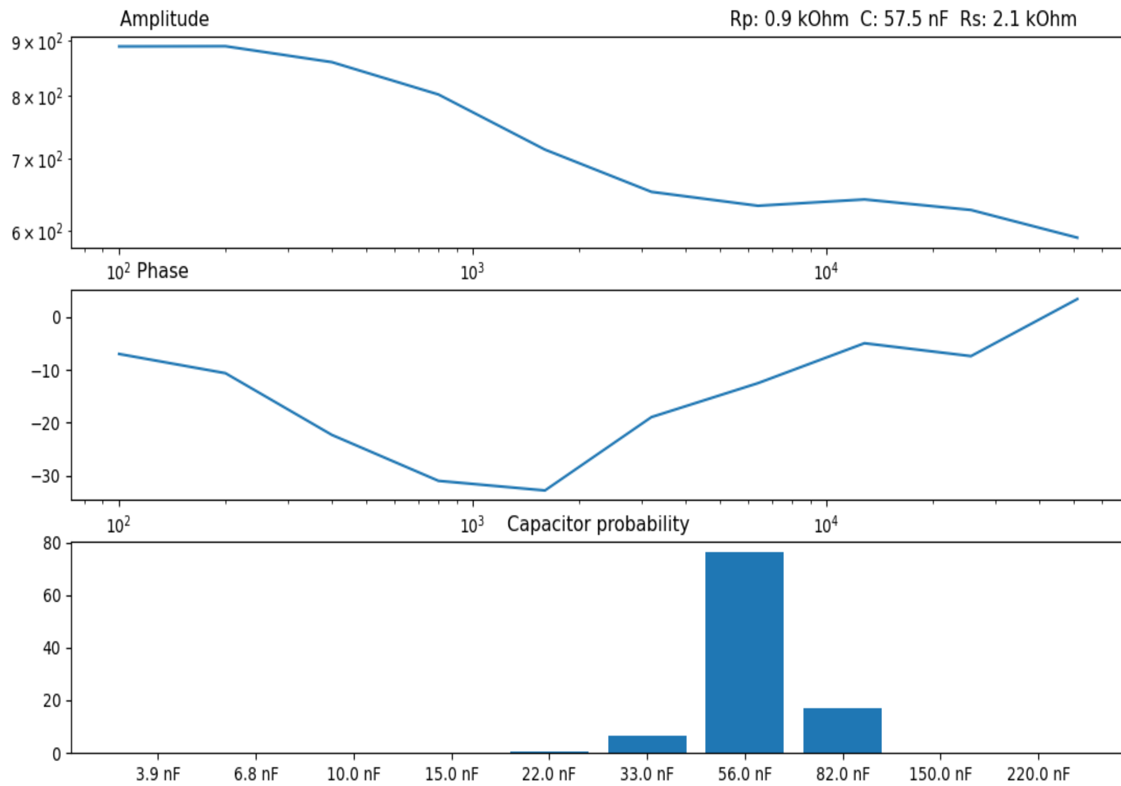


Figure 58: Amplitude and phase of noisy spectrum with C=57.5nF and probabilities of the capacitors

5.3.4 Applying neural networks to real-world problems

The two experiments show the advantages of ANN. One benefit is that they can learn to handle faulty signals, like in the first example what could make the system robust against noise and outliers. The other benefits are that it can learn to estimate a value from given data with non-linear dependence and can handle cross influences. In the example the searched value was the capacitor, the analyzed data was the spectrum of the RCR-circuit which is also influenced by the resistances.

The same idea could be used for example to estimate a chemical concentration by evaluating EIS-signals, which are cross influenced by physical properties like the temperature or by another chemical content.

The previous two examples worked well because in the simulation was an infinite data source. However, in the most cases, for example to process sensor data from the real world, the training data can´t be generated in a simulation by random numbers, but needs to be acquired through measurements, which can be quite costly and time consuming if a high amount of training data is required.

The dependence on data and the fact the amount of data is usually limited is one of the major problems in machine learning.

6 System validation with saline solutions

6.1 Acquisition of sensor data for training

As first step to get the neural network applied to the salt content measurement, sensor data had to be acquired and accordingly samples with 10 different concentrations of potassium chloride (KCl) with distilled water were prepared.

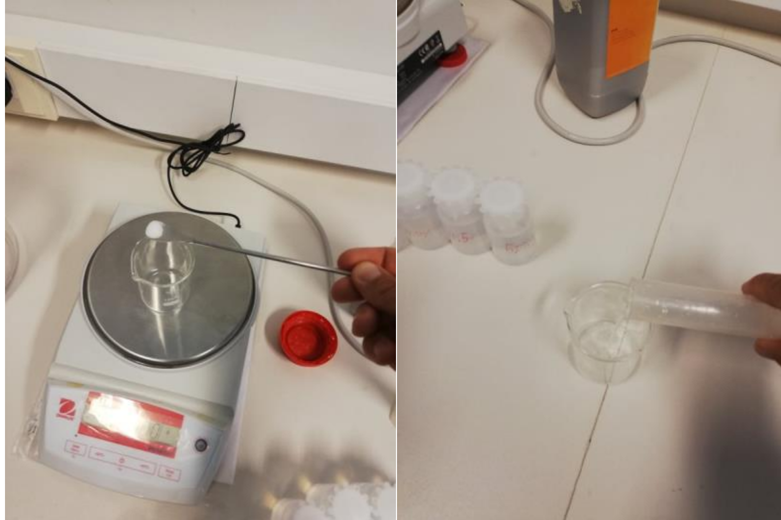


Figure 59: Accurate preparation of samples with different salt content

The sensor measured from each solution sample about 300 spectra which were saved in a CSV-file with additional label indicating the corresponding concentration, (0.5%, 1.0%, 1.5%, 2.0%, 2.5%, 3.0%, 3.5%,4.0%, 4.5%,5.0%, with 1% meaning 1g KCl in 100mL water). The spectra were measured at ten frequencies (100Hz, 200 Hz, 400 Hz, 800 Hz, 1.6 kHz, 3.2 kHz, 6.4 kHz, 12.8 kHz, 25.6 kHz, and 51.2 kHz).

Since the appearance of the spectra can vary due to the room temperature, this process was repeated several times to obtain data with the typical variation. After inserting the probe into the sample, it took several seconds to a minute for the signal to stabilize.

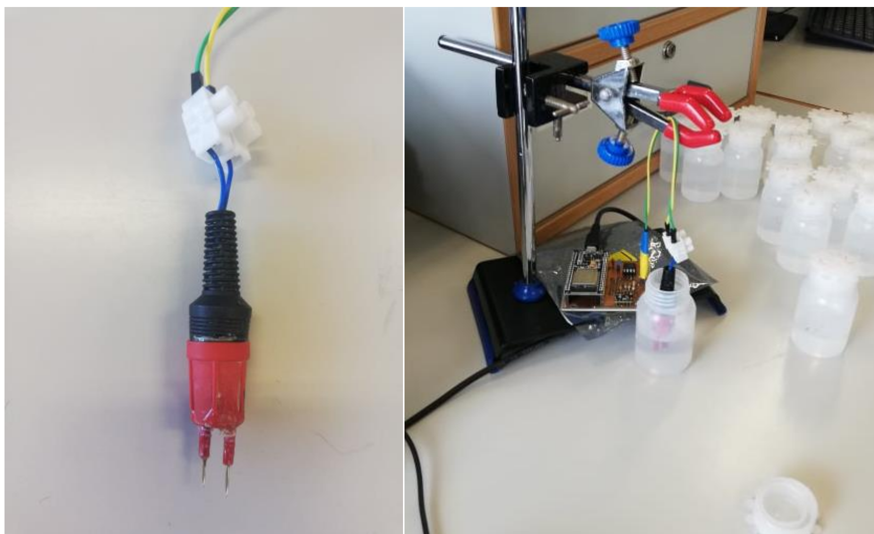


Figure 60: Electrode and set-up used to measure the samples

The spectra have also shown to depend on the temperature as expected, especially in the lower frequencies. The variation of the impedance in a room with varying temperature leads to an overlap of the impedance values of neighboring classes, especially at the higher concentrations (4.0%,4.5%,5.0%). Consequently, it is not possible to classify the concentration clearly if only one value from the spectrum is considered and the temperature is not stabilized. Since the permittivity of the water depends on the temperature as well, the temperature and salt concentration don´t change the spectrum equally. Varying temperature shouldn´t cause an irresolvable ambiguity and the correct concentration could be determined clearly by using an ANN that considers the shape of the entire spectrum, and thus makes the classification robust against the cross influence of the temperature. However, the spectrum of a warmer solution might differ only slightly from the spectrum of a solution with higher concentration, for that reason it would be necessary to measure with a certain precision and to train an appropriate ANN accurately.

6.2 First attempt to evaluate the data with an ANN

The first neural network constructed in Python had the dimension 20:10:10:10

```
import NeuralNet as nn
import ActivationUnit as au

# Construct the Neural Network
ANN_SalinityClass = nn.NeuralNetwork(20)
ANN_SalinityClass.AddLayer(10,au.LeakyRelu(0.01))
ANN_SalinityClass.AddLayer(10,au.LeakyRelu(0.01))
ANN_SalinityClass.AddLayer(10,au.Softmax())
```

Figure 61: Python code to construct the ANN used for classification

The network had 20 inputs to receive ten magnitudes and phase values and ten outputs to signal the probability of the categories. The two hidden layers used the LeakyReLU function with a leakage value of 0.01.

Before the ANN can be fitted to the measured data, the data for the CSV had to be converted. First, the means and deviations of the twenty input values was determined, and subsequently the inputs were normalized to obtain input values with mean equal to zero and deviation equal to one:

$$Input_{norm}[k] = \frac{Input[k] - Mean[k]}{Deviation[k]}$$ with k as the index of the input

The output values were represented as one-hot-vectors signaling the corresponding class.

0.5% as [1,0,0,0,0,0,0,0,0,0], 1.0% as [0,1,0,0,0,0,0,0,0,0], etc.

Afterwards the ANN was trained during several epochs to fit as close as possible to the sensor data. The trained ANN was saved in a JSON-file and loaded into the GUI evaluating online acquired sensor data. The ANN occasionally classified correctly claiming high certainties, about 95%. However, more often the classification was incorrect and even several classes away from the correct class. An interesting observation was the fact that the classification also claimed a very high certainty even when the result was far away from the correct class. This phenomenon could be an effect of a very common problem in machine learning called **overfitting**.

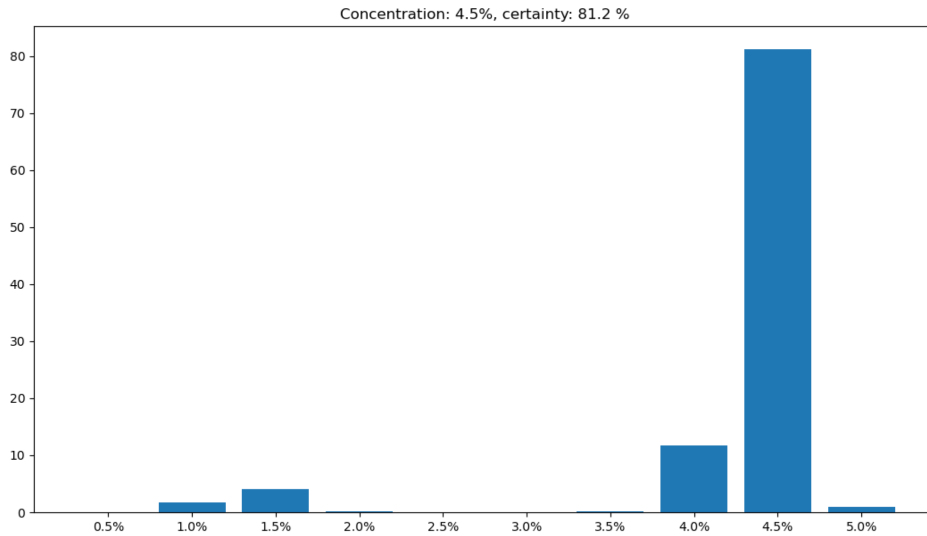


Figure 62: Actual value 2.5% is incorrectly classified as 4.5%, however it claims a high certainty

6.3 Overfitting and prevention

A very popular mistake is the idea that a more complex system will work better. Indeed, an ANN needs a certain complexity to learn complex functions, however a network with a huge number of tunable parameters can only be trained well if there is the necessary data available to fit the complex structure. Otherwise, if the ANN has too much freedom and the data is too sparse, it could fit to the noise in the data and thus it doesn't learn the function correctly.

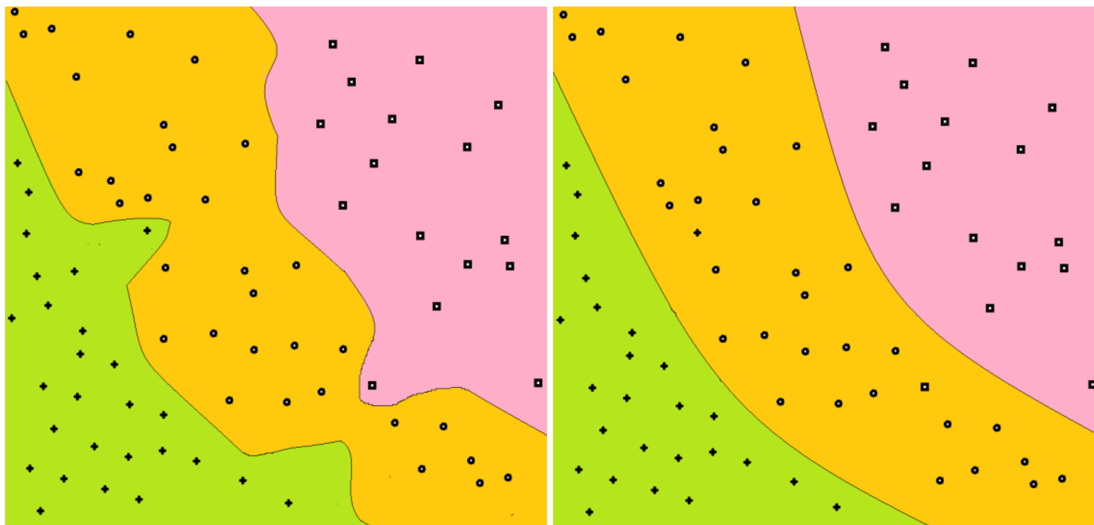


Figure 63: Left: The classification model is very complex and the data too sparse
Right: If the complexity of the model is reduced it can learn with sparse data

Validation data can be evaluated, which could indicate if the system is overfitted. It is necessary to have separate data which was not used for the training.

6.3.1 Using validation data and early stopping

During the training the loss value of the network on the training data, the value that shall be minimized, can be observed and in the case if the model is an ANN with sufficient capacities and the optimizer algorithm is suitable, it will be stochastically minimized until it reaches a low value. Additionally, it makes sense to observe the loss of the model on independent validation data. It

will be observed that the test loss gets lower as the train loss gets lower, but the test loss will stop to decrease and even can rise again. This is the point where the model fits too close to the training data instead of being generalized. At this point the ANN begins to overfit and it doesn't make sense to continue the training. The technique to stop at this point to prevent overfitting is named **Early Stopping** [21].

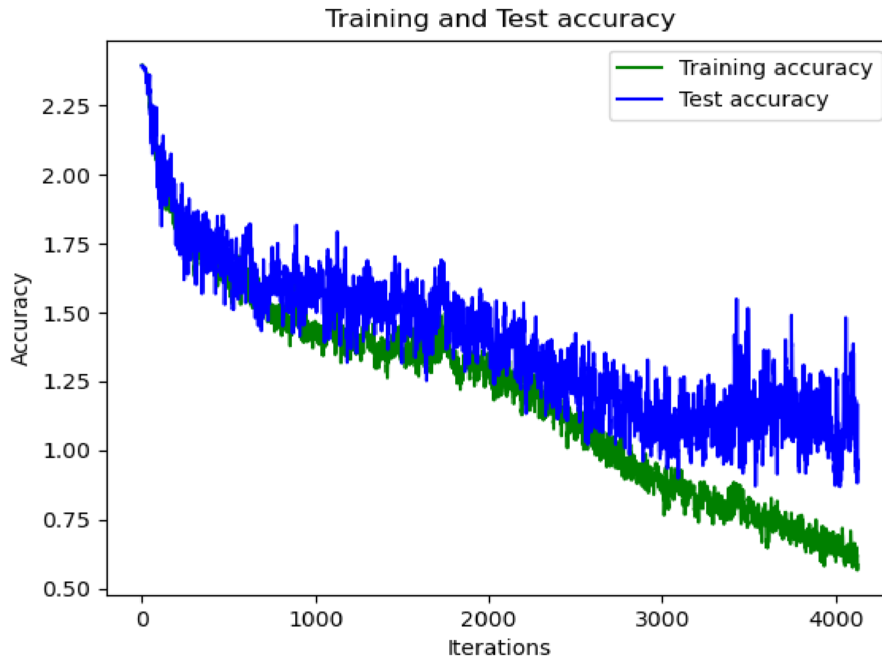


Figure 64: The test loss doesn't get better when the overfitting begins (here at around 3000 iterations)

6.3.2 Quality of the training data

The data used for the training is of crucial importance. If the data is too sparse, noisy data can prevent a correct fitting. The measurements should also be distributed evenly over the entire space of possible concentrations and cross influences, thus there must be approximately the same number of samples of all salinity classes and they all should be measured at different temperatures evenly distributed over the targeted temperature range. For example, if a training data set contains thousands of samples of the concentration 4.5% and only one hundred samples of the concentration 4.0%, the trained ANN might classify a concentration of 4.0% as 4.5%, because the inputs are close, and the ANN has learnt to classify it as 4.5%, because it is considered as more likely.

With the intention to train the ANN with more information, more concentration values were used, **(0.25%, 0.75%, 1.25%, 1.75%, 2.25%, 2.75%, 3.25%, 3.75, 4.25%, 4.75%, 5.5%),** and additionally, 11 classes were defined **(0.0%-0.5%, 0.5-1.0%, 1.0%-1.5%, 1.5-2.0%, 2.0%-2.5%, 2.5-3.0%, 3.0%-3.5%, 3.5-4.0%, 4.0%-4.5%, 4.5-5.0%, >=5.0%),** describing an interval for the salt content. The previously used samples (0.5%, 1.0%, 1.5%, 2.0%, 2.5%, 3.0%, 3.5%,4.0%, 4.5%,5.0%) were used to define the limits of the classes. The training data sample at the limit of the intervals obtained as output label a vector with two 0.5 instead of a single 1, indicating the sample can belong to both classes with 50% to 50% probability, (concentration of 0.5% represented as [**0.5**,**0.5**,0,0,0,0,0,0,0,0,0], 1.0% as [0,**0.5**,**0.5**,0,0,0,0,0,0,0,0], etc.).

6.3.3 Training with noise added to the input

During every training iteration, noise or outliers can randomly be added to the used data, with the intention to optimize the network in a way it gets robust against noise.

Noise on the input displaces the sample point in the input parameter space and that way it can fill the empty places between the data samples, which can also reduce the problems of overfitting. Although by adding noise the input data is changed, it doesn´t augment the data and empty regions in the input space remain empty, which is why extensive and high-quality data acquisition still is indispensable.

A more advanced similar technique is the usage of dropout layers, hidden layers where units randomly are set to zero, which wasn't realized in this work.

6.3.4 Weight decay

One way to prevent overfitting is to keep the model simple by avoiding wide layers and reducing the number of hidden layers, to have less parameters to be tuned. However, it is also possible to train the network to be less complex by having less weights that connect the neurons of each layer. The often-used L2-regularization is to add the sum of the square of the weights multiplied by a factor λ, to the loss with the goal to get a solution with less weights [20][21].

This can also be realized by multiplying the weight with a factor (1-2λ) at every iteration. The weight decay coefficient is usually of a very low magnitude (around 0.001 - 0.00001). A higher value prevents the system from getting too complex, but if the value is very high it could underfit, that is, the system won't be complex enough to perform the required function.

6.3.5 Deep networks versus wide networks

According to the *universal approximation theorem*, a neural network with only one hidden layer could approximate any non-linear function, provided there are enough neurons in the hidden layer [22]. However, ANNs with wide layers can be difficult to get well trained because in a wide neural network, there are a lot of weights that connect the many units, making the system quite complex. That is why it makes sense to use ANN with several layers, so-called deep neural networks. Such networks with many layers can perform complicated functions with a lower number of weights than ANN with wide layers. The term deep learning results from the insight that deep neural networks are more powerful and applicable. However, very deep networks can also be difficult to train cause of the dying-gradient problem, what makes it necessary to apply suitable activation function e.g., LeakyReLU to avoid dying neurons. With more layers, there are also more hyperparameters, for example the leakage value or the number of neurons. These can be challenging to set appropriately because all hyperparameters, those describing the network and those defining the training process will have an influence on the results of the ANN.

6.4 Results

A working network architecture was found by trial-and-error:

```
# Construct the Neural Network
ANN_SalinityClass = nn.NeuralNetwork( 20 )
ANN_SalinityClass.AddLayer(8,au.LeakyRelu(0.01))
ANN_SalinityClass.AddLayer(8,au.Elu(1.0))
ANN_SalinityClass.AddLayer(9,au.Elu(0.5))
ANN_SalinityClass.AddLayer(10,au.LeakyRelu(0.1))
ANN_SalinityClass.AddLayer(11,au.Softmax())
```

Figure 65: Construction of the deep neural network used for the classification

The network was trained on around 10000 spectra as training data for 12 epochs, the batch size was 30, the step size 0.10, the momentum coefficient 0.05, the weight decay coefficient 3e-05, noise with deviation 0.03 was added to the input and the log loss was applied. The network used

five layers with the activation function ELU and LeakyReLU for the hidden units. This network with five layers was tested and produced good results, although it is likely simpler architectures exist that provide even better results.

The resulting ANN, which was trained with techniques to avoid overfitting, doesn't claim high certainties and the probabilities are more distributed over the classes. However, the estimated class indicated by the highest probability is in the most cases correct.

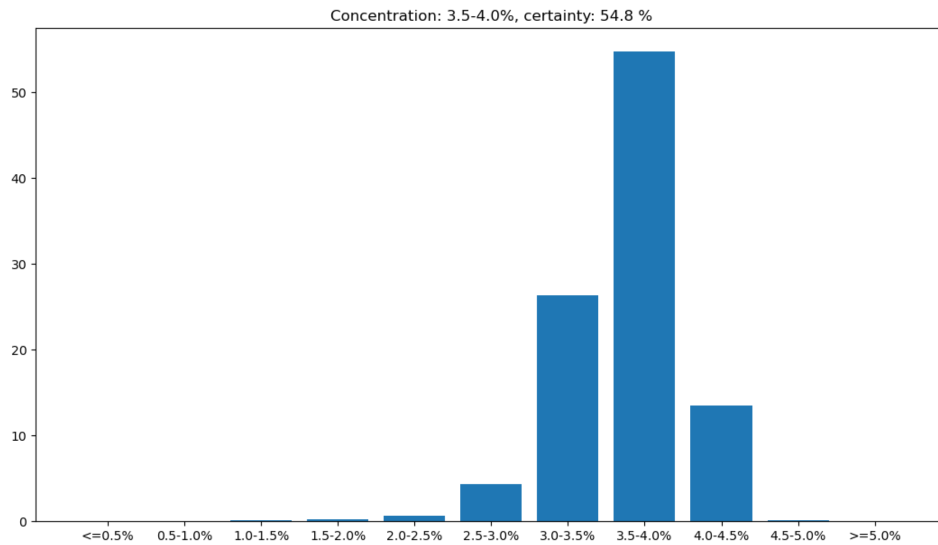


Figure 66: Sample with 3.75% is classified correctly and probabilities are more distributed

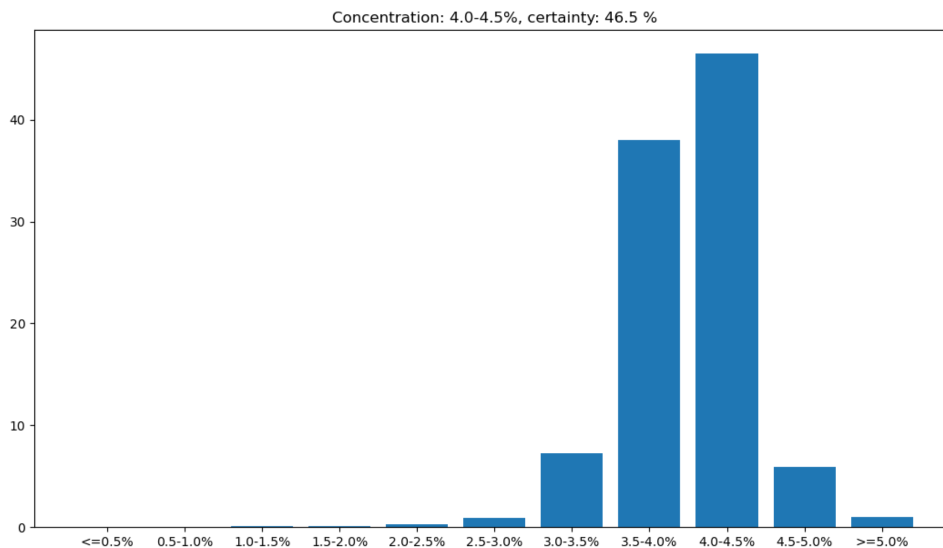


Figure 67: Result of sample with 4.0% on the border between two classes, both classes have similar probabilities

After changing the solution, it takes some seconds until the sensor measures a stabilized signal for the classification. The estimated class of the resulting ANN is in the most cases correct and if it classifies incorrectly, the result is usually a neighboring class. All the concentrations below 3.5% are classified correctly, however higher concentrations are more often classified incorrectly, for example the class "3.5%-4.0%" is sometimes classified as "4.5%-5.0%", probably because the signals of these concentration are too close and, for example, noise or a drift in the measurement caused by the temperature can change the result.

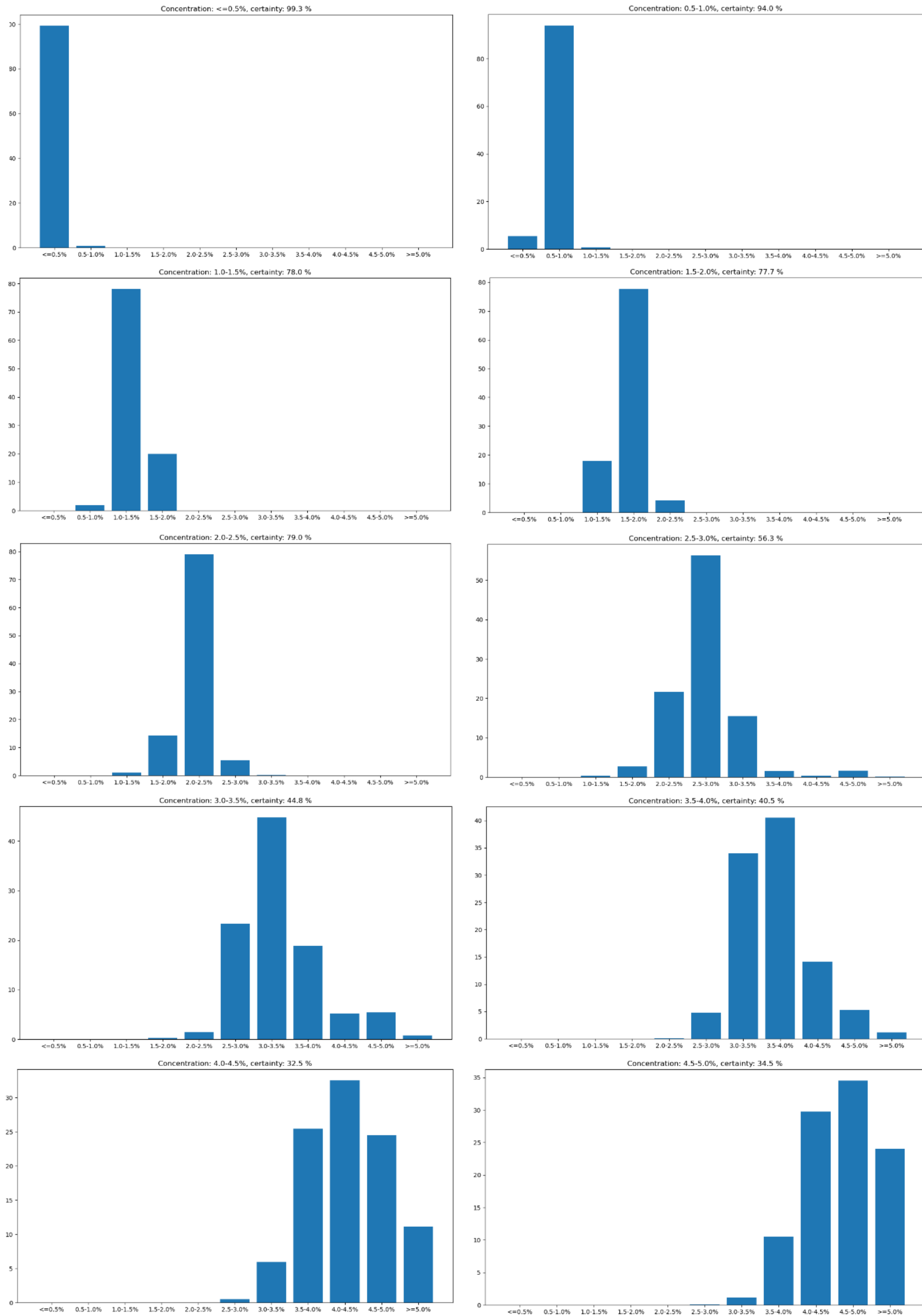


Figure 68: Classification of concentrations from 0.25% to 4.75%, more uncertainty at the higher concentrations

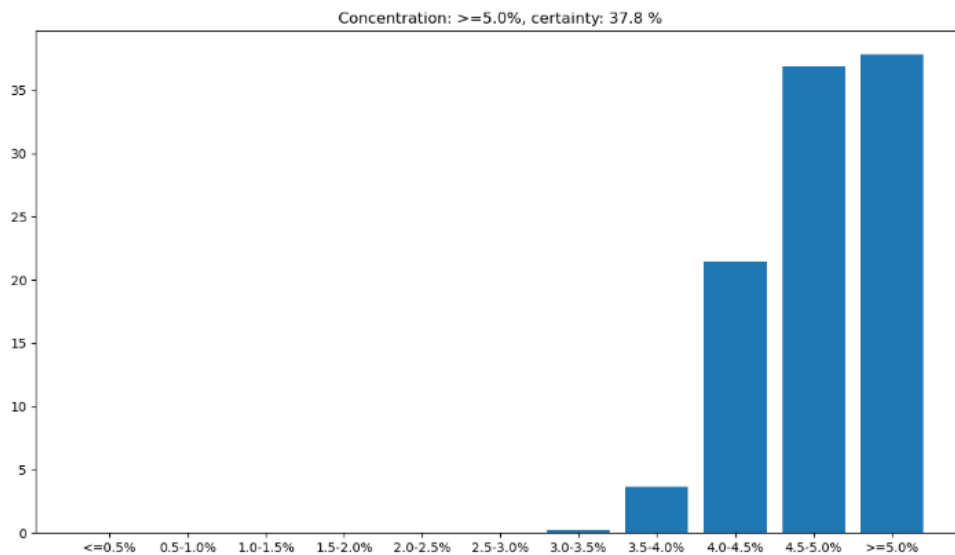


Figure 69: Here the sample of 5.5% is classified correctly, but there is a very thin gap between the output of the correct class and the neighboring class

Especially the higher concentrations, whose signals have a relatively short distance, are often misclassified. It's not clear whether the precision of the transmitter is insufficient, or whether it is a problem of the tuning or the structure of the ANN or whether its problem caused by the quality of the training data.

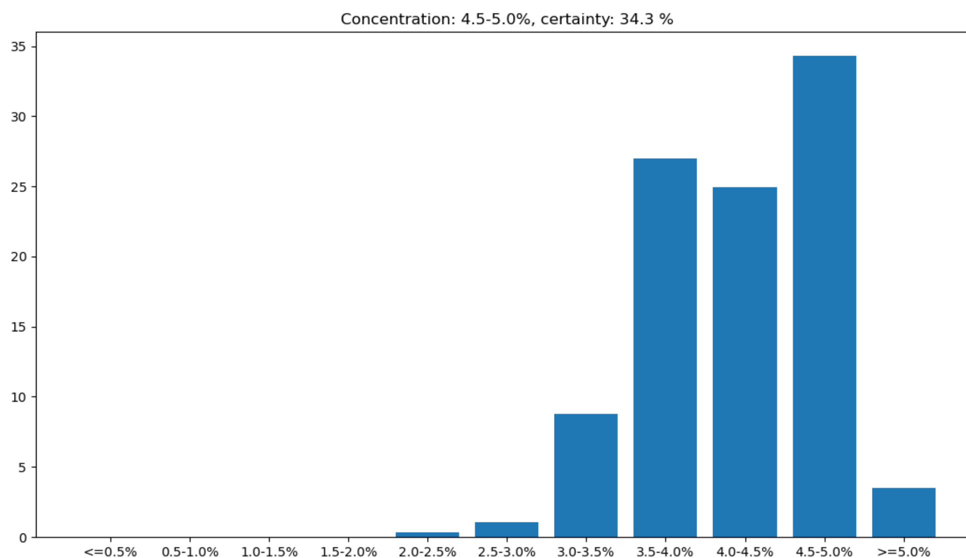


Figure 70: Sample with 3.75% was falsely classified as 4.5-5.0%

There remain temperature dependencies what couldn´t be resolved by the ANN. To learn the classification function correctly, the amount of training data from every class needs to be roughly the same otherwise the classification limits will be distorted. Accordingly, if it shall learn the temperature dependency correctly, the number of samples must be distributed evenly over the required temperature range and be measured very accurately. Tests with noise simulated in the sensor also have shown sensitivity to noise and outliers. Unlike the network used in experiment 1, the applied ANN apparently has not the capability to filter disturbances from the signal, although it was trained with noise and outliers.

After training and verifying a valid ANN, it can also run on the microcontroller following a simple workflow.

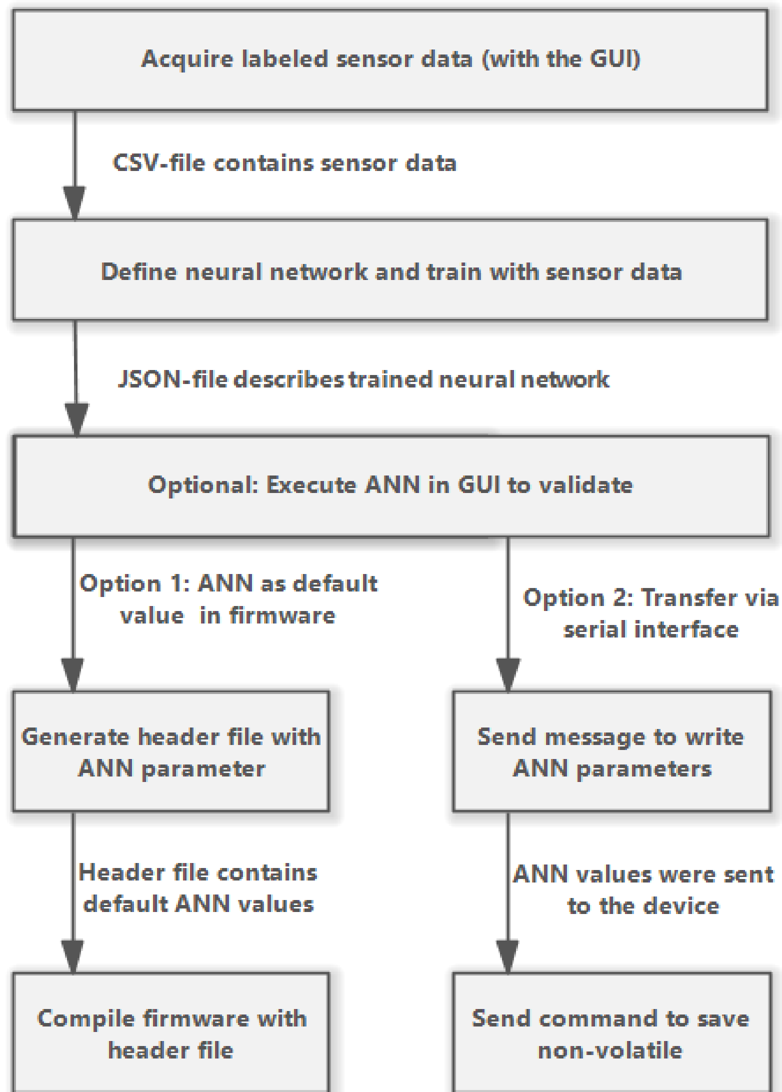


Figure 71: Workflow to customize the neural network on the sensor

The process to obtain a device with a valid neural network would be the same for any other application. First the sensor data with corresponding label must be acquired using the GUI and saved as CSV file, subsequently an appropriate ANN must be defined in a Python script and trained using the before acquired data, and optionally the resulting JSON-file can be loaded in the GUI to be validated with the online acquired data from the sensor. There are two possibilities to get the ANN on the device, by generating a C-header file with the data from the JSON-file, containing the parameters of the ANN which will be compiled with the firmware or by sending the parameters through the serial interface to the device and subsequently saving them to the EEPROM. In this way, the device can be customized for various applications.

The EIS-sensor with embedded ANN was tested by correctly classifying aqueous solutions according to the salt content. The ESP32 performs the acquisition of a spectrum and subsequent evaluation in less than 1.5 seconds, with measurements at 10 different frequencies from 100 Hz to 51.2 kHz, and a sampling rate of 200 ksps for each ADC channel.

7 Conclusions and future works

The performance of the low-cost controller ESP32 is astonishing, both because of its computing power and because of its high ADC sampling rate. Apart from a few minor flaws that can be handled, the ESP32 has absolutely no problem in acquiring the required signal and in executing the neural network. Also, the analog electronic performs well, resulting in accurate measurement, and in the future a more advanced design using SMD could provide even better results. There are also possibilities to improve the performance of the digital signal processing e.g., by applying filters to decrease noise. A very useful feature of the firmware is the possibility to configure the applied frequencies of the spectroscopy and to configure the architecture of the neural network, to adapt the device to different applications.

Although Python is rather simple to learn, it is a very mighty general-purpose language, which is practical to program graphical user interfaces and other utilities. The only thing lacking is the speed, which could be improved in the future. In this work the neural network was programmed from scratch, which at first glance seems unnecessary because nowadays very advanced machine learning libraries are available, but it can have significant advantages. It is a very good example to practice object-oriented programming in Python, it raises awareness of how a neural network indeed works and it is simpler to adapt the code to the specific problem. However, in future works it will make more sense to apply a common machine learning library, for example, TensorFlow, Keras, or PyTorch, since they provide very advanced technologies e.g., modern training algorithms such as Adam, RMSprop etc. The neural network of the chosen dimensions has produced good results, but in the future even better and simpler solutions likely can be found with automated hyperparameter tuning techniques e.g., grid search or evolutionary optimization. Machine learning libraries for embedded solutions in Arduino exist, which are compatible with Python libraries and support learning on the device.

Neural networks can be powerful, though it is necessary to consider many tricky details. Especially the need for appropriate training data can get challenging. The quality of the classification could be improved by a more accurate and precise training data acquisition with evenly distributed cross influences like the temperature or another chemical content. Data from more concentrations should be recorded, possibly by an automated process. Furthermore, the temperature should vary in a controlled way, e.g., with the help of a climate chamber, to learn the spectra at different temperatures to classify correctly regardless of climatic interferences.

The experimental networks which removed disturbances from a signal and classified capacitors have shown the capabilities of neural networks, to learn non-linear functions, to outweigh cross influences and robustness against noise. In the future when even better hardware and improved machine learning libraries will be available, more advanced ANN can be applied on the sensor, which could make the measurement robust against various kinds of interference and noise.

However, to obtain these advantages in practical applications there are some prerequisites. In real world application, unlike measuring a clean salt water solution, many cross influences must be considered. For example, if the salinity of a dough shall be measured, not only the temperature has an influence on the spectra, but also the type of flour, the ratio of flour and water, the density, the content of sugar, baking soda, oil, yeast, presence of carbon dioxide bubbles and more chemical or physical properties. Impedance spectra measured at many frequencies and accurately trained ANN will be necessary to evaluate clearly considering all possible disturbing parameters. The ANN on the device can be adapted to a specific application following a simple workflow, which makes the device easily customizable. As soon as there is a robust sensor, a handy low-cost device can be developed, with a display or communication via Bluetooth to be controlled by the mobile phone to facilitate in-situ measurement.

The development of the EIS-sensor has laid the foundation for numerous possible applications that could be realized in the future to enable more frequent measurements at lower costs. The work was elaborated at the "Instituto Interuniversitario de Investigación de Reconocimiento Molecular y Desarrollo Tecnológico (IDM)" in Valencia, Spain and was presented as oral communication at the "XIV International Workshop on Sensors and Molecular Recognition IWOSMOR".

Figure72: IWOSMOR, Valencia, 8th-9th July 2021

Figure 73: Instituto Interuniversitario de Investigación de Reconocimiento Molecular y Desarrollo Tecnológico

8 Bibliography

[1] Ochandio, Ariel, Masot, Laguarda, Freeze-Damage Detection in Lemons Using Electrochemical Impedance Spectroscopy, *Sensors*, **2019**

[2] Serrano, Muñoz, Pérez, Masot, Laguarda, Early Detection of Freeze Damage in Navelate Oranges with Electrochemical Impedance Spectroscopy, *Sensors*, **2018**

[3] Claudia Conesa, Luis Gil, Lucía Seguía, Pedro Fito, Nicolás Laguarda, Ethanol quantification in pineapple waste by an electrochemical impedance spectroscopy-based system and artificial neural networks, *Chemometrics and Intelligent Laboratory Systems,* **2017**

[4] Jordi Riu, Barbara Giussani, Electrochemical biosensors for the detection of pathogenic bacteria in Food, *Trends in Analytical Chemistry,* **2020**

[5] Fuentes, Masot , Fernández-Segovia , Ruiz-Rico, Alcañiz , Barat, Differentiation between fresh and frozen-thawed sea bream (Sparus aurata) using impedance spectroscopy techniques, *Innovative Food Science and Emerging Technologies,**2013***

[6] Grossi, Parolin, Vitali, Riccò, Electrical Impedance Spectroscopy (EIS) characterization of saline solutions with a low-cost portable measurement system, *Engineering Science and Technology, an International Journal,* **2018**

[7] Haykin, Widrow, Least-Mean-Square Adaptive Filters, *John Wiley & Sons*, **2003**

[8] Kuo, Lee, Tian, Real-Time Digital Signal Processing, *John Wiley & Sons,* **2006**

[9] Andrzej Lasia, Electrochemical Impedance Spectroscopy and its Applications, *Springer Science+Business Media New York* ,**2014**

[10] Capilla, Coll, Tormos, Trénor, Gil, Curso de instrumentación electrónica, *Universidad Politécnica de Valencia Departamento de Ingeniería Electrónica, Editorial UPV,* **2005**

[11] Masot Peris, Desarrollo de un sistema de medida basado en espectroscopía de impedancia para la determinación de parámetros fisicoquímicos en alimentos*, UPV Departamento de Ingeniería Electrónica,* **2010**

[12] Jung, Water G., Op Amp Applications Handbook, *Analog Devices series,***2005**

[13] U. Tietze, Ch. Schenk, E. Gamm, Electronic Circuits Handbook for Design and Application *2nd Ed. (Translation of Tietze, U.; Schenk, Ch.: Halbleiter-Schaltungstechnik. 12. Edition)*, **2002**

[14] **"**MicroPython external C modules**",** *(https://docs.micropython.org/en/latest/develop/cmodules.html) ,* **2020**

[15] Ganssle, Perrin, Kamal, Hyder, Arnold, Katz, Edwards, Eady, Noergaard, Embedded hardware: Know it all, *Newnes*, **2007**

[16] ESP32 Technical Reference Manual, *Espressif Systems,* **2021**

[17] Simon, An embedded software primer, *Addison-Wesley Professional,* **1999**

[18] Lerner, Python workout: 50 ten-minute exercises, *Manning Publications,* **2020**

[19] Martelli, Ravenscroft, Holden, Python in a Nutshell, *O'REILLY*, **2017**

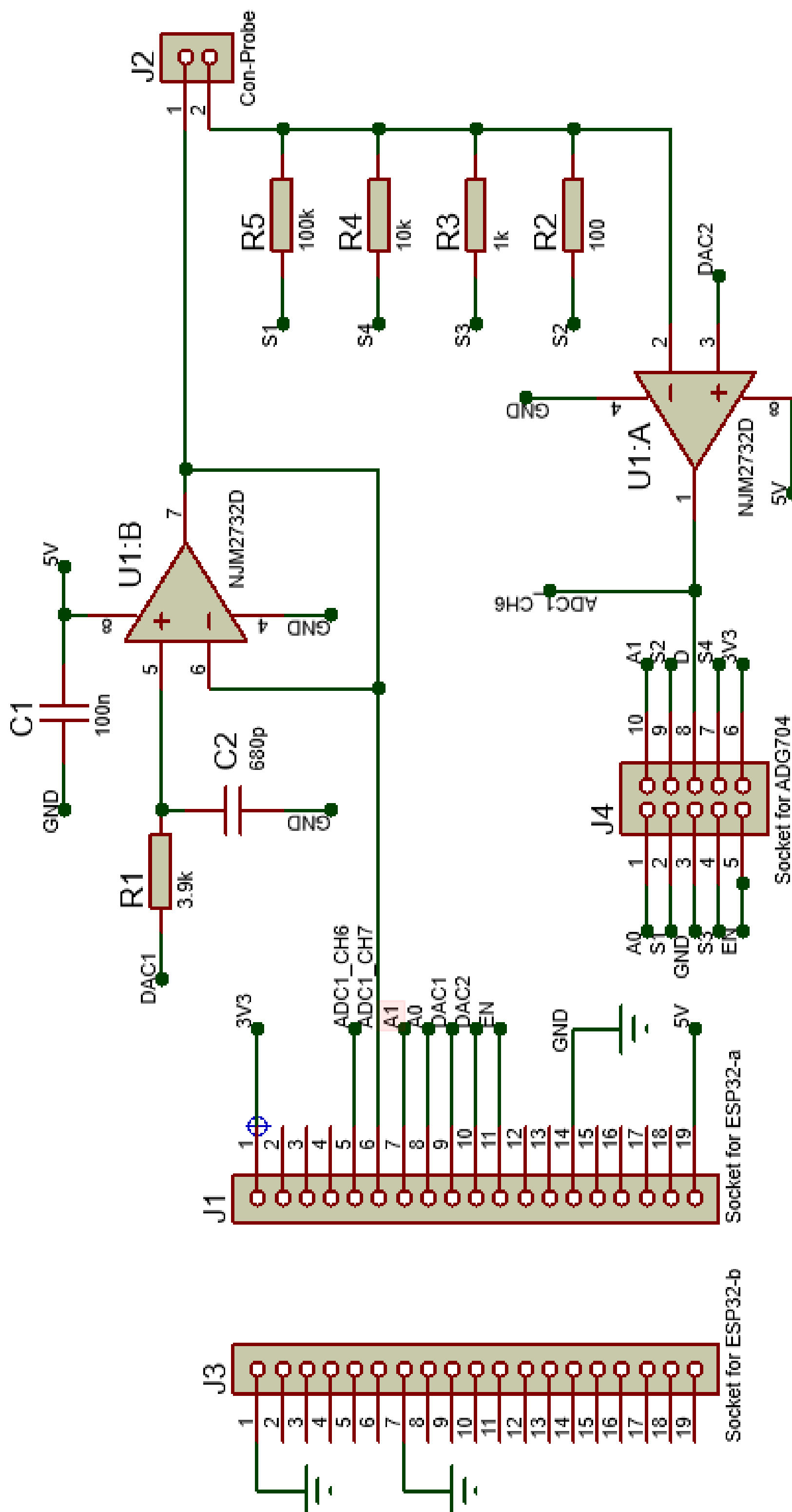
[20] Goodfellow, Bengio,Courville,Deep Learning (Adaptive Computation and Machine Learning series), *The MIT Press* ,**2016**

[21] Chollet, Deep learning with Python, *Manning Publications Co.,**2018***

[22] Haykin, Neural networks and learning machines, *Pearson Education* ,**2009**

9 Appendix

9.1 Detailed schematic of the analog hardware



9.2 Command keywords

Messages to the device:

Keyword	***Optional Parameter***	***Description***
START_SPEC	Frequencies to be applied	Start the impedance spectroscopy, without given frequencies saved values from memory are used
STOP_SPEC	-	Stops the impedance spectroscopy
SEL_SHUNT	Integer (1 to 4)	
SAVE_NVM	-	Copy the current settings from RAM into the EEPROM
RESTART	-	Restart the device
RESET_PARAMS	-	Copy the default values from flash to the RAM
PARAM_LIST	-	Lists all accessible parameters
GET_PARAM	ParamID	Return values of requested parameter
SET_PARAM	ParamID, value	Write value to the corresponding parameter
SET_SUBPARAM	ParamID, SubID, value	Write value to position defined by the SubID
TEST_DACADC	frequency of sine in Hz	Apply a sine wave with given frequency on the DAC and send 4000 samples of the ADC in CSV-format to test the analog circuit

Messages from the device:

Keyword	***Optional Parameter***	***Description***
READY	-	Device started successfully
IMPED	n, f, Mod,Pha,	Measured values of the impedance at frequency f, n is position in spectrum
PARAM	ParamID, Type, Size	Description of a device parameter
PARAM_VAL	ParamID, Value	Get value of requested device parameter
ANN_OUT	P_Classs1, P_Class2…	Output of the ANN

9.3 Device parameters

Keyword	***Type***	***Size***	***Description***
NUM_FREQS	INTEGER	4	Number of frequencies of the spectrum
FREQS_VAL	FLOAT_ARRAY	40	Values of the frequencies of the spectrum
V_VGND	FLOAT	4	Voltage of virtual ground set by DAC2
V_OUT	FLOAT	4	Amplitude of the stimulation signal
ADC_SR	INTEGER	4	Sampling rate of the ADC
LEN_PER	INTEGER	4	Number of samples in ADC value buffer
SAMP_DELAY	INTEGER	4	Delay between applying sine and sensing
SHUNT_SEL	INTEGER	4	Selection of shunt num 1 to 4
SHUNT_VALUES	FLOAT_ARRAY	16	Ohmic value of available shunts
NUM_CALIB	INTEGER	4	Number of calibration points
CALIB_FREQS	FLOAT_ARRAY	40	Frequency of calibration point
CALIB_OFFSET	FLOAT_ARRAY	80	Alternating R and I value of complex valued correction offset

CALIB_FACT_N	FLOAT_ARRAY	80	Alternating R and I value of complex valued correction factor for shunt N
SIMU_NOISE	FLOAT_ARRAY	8	Levels of the simulated noise added to the voltage or current measurement
INP_NORM_MEAN	FLOAT_ARRAY	96	Mean value of training input to normalize
INP_NORM_DEV	FLOAT_ARRAY	96	Deviation of training input to normalize
ANN_HYPERPARAMS	UINT_ARRAY	40	Defining the structure of the ANN layers
ANN_TUNEDPARAMS	FLOAT_ARRAY	2400	Batch of tunable parameters of the ANN
DSP_OPTIONS	UNSIGNED INT	4	Flag 0: Calculate calibration, Flag1: Impedance as Modulus-Phase,

9.4 Overview activation functions

Name	***Definition***	***Usage***
Binary step	$y = \begin{cases} 0 \; if \; x < 0 \\ 1 \; if \; x \geq 0 \end{cases}$	Theoretical, historical concept of ANN, practically not applicable
Linear Unit	$y = x$	Output for regressions
Sigmoid	$y = \frac{1}{1 + e^{-x}}$	Output for binary classifications
Tanh	$y = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$	Used in recurrent networks (LSTM, GRU)
Softmax	$y_k = \frac{e^{x_k}}{\sum_{i=1}^{n} e^{x_i}}$	Output for multi-category classifications
Rectified linear unit (ReLU)	$y = \begin{cases} 0 \; if \; x < 0 \\ x \; if \; x \geq 0 \end{cases}$	Used in hidden layers, most recommended
Softplus	$y = \ln(1 + e^{x})$	Used in hidden layers, less recommended
ELU (exponential linear unit)	$y = \begin{cases} \alpha(e^{x} - 1) \; if \; x < 0 \\ x \; if \; x \geq 0 \end{cases}$	Used in hidden layers
LeakyReLU and PReLU	$y = \begin{cases} \alpha x \; if \; x < 0 \\ x \; if \; x \geq 0 \end{cases}$	Like ReLU but with leakage to avoid dying gradient problem, LeakyReLU has fixed leakage (usually $\alpha = 0.01$), PReLU has leakage as a separate tunable parameter for every unit
Swish	$y = \frac{x}{1 + e^{-x}}$	Used in modern very deep networks

9.5 Overview loss functions

Name	***Definition***	***Usage***
Mean squared error	$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$	Regressions
Cross entropy or Log loss	$H(p,q) = -\sum_{x \in X} p(x) \log(q(x))$ p=probability provided by label, q = prob output	Multi-category classifications
Binary cross entropy	$H(p, q) = -p(x) \log(q(x)) - (1 - p(x)) \log(1 - q(x))$	Binary classification