

PL „Autonomous Intelligent Systems“

RP-US-C Sensor Classifier Implementation and Assessment Project Report

Created by

Sophie Frerk 1368437

Roman Nagel 1359653

1 Statutory Deklaration

We declare that we have authored this thesis independently, that we have not used other than the declared sources / resources, and that we have explicitly marked all material which has been quoted either literally or by content from the used sources.

X 
Frankfurt am Main, 26.04.2021

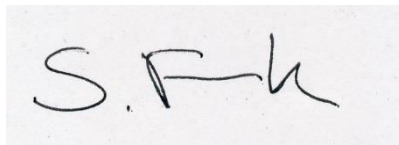
X 
Frankfurt am Main, 26.04.2021

Table of Content

Table of Content.....	3
List of Figures.....	4
List of Tables.....	4
List of Codes	4
Table of Abbreviation	5
1 Introduction.....	6
2 Objectives.....	7
3 Theoretical basics.....	8
3.1 Typical Classifier	8
3.1.1 Box Classifier	8
3.1.2 Support Vector Machine	8
3.1.3 Neural Network/Deep Learning.....	8
3.2 Software used.....	9
3.3 Hardware used.....	9
4 State of the art	10
4.1 Hardware	10
4.2 Software.....	11
5 Realization	12
5.1 Hardware model.....	12
5.2 Software model	12
5.3 Programming	14
5.4 UDP Commands.....	17
5.5 Training.....	19
6 Conclusion	21
References.....	22
Appendix.....	23

List of Figures

<i>Fig. 1: Sensor combi</i>	<i>10</i>
<i>Fig. 2: GUI.....</i>	<i>11</i>
<i>Fig. 3: Test setup</i>	<i>12</i>
<i>Fig. 4: Neural Network Architecture.....</i>	<i>13</i>
<i>Fig. 5: Text-file.....</i>	<i>18</i>
<i>Fig. 6: Wall FFT.....</i>	<i>19</i>
<i>Fig. 7: Human FFT.....</i>	<i>20</i>
<i>Fig. 8: Chair FFT.....</i>	<i>20</i>

List of Tables

<i>Table 1: UDP commands</i>	<i>17</i>
------------------------------------	-----------

List of Codes

<i>Code 1: Struct net_param_t.....</i>	<i>14</i>
<i>Code 2: allocation guideline</i>	<i>15</i>
<i>Code 3: secondary allocation of memory</i>	<i>15</i>
<i>Code 4: Initialization algorithm</i>	<i>16</i>

Table of Abbreviation

ADC	Analog Digital Converter
AI	Artificial Intelligence
DL	Deep Learning
FFT	Fast Fourier Transformation
GUI	Graphic User Interface
IDE	Integrated Development Environment
ML	Machine Learning
NN	Neural Network
SVM	Support Vector Machine
UDP	User Datagram Protocol
US	Ultrasonic Sensor

2 Introduction

In the current evolution in industry 4.0, one of the most important subjects is the connectivity between systems and sensors. The development of smart sensors is part of this process. Neural networks (NN) have recently made their way into the sensor level of the industry. Known examples are smart cameras and onboard image processing software. NNs are known to be resource intense, but this heavily depends on the application. Under the right conditions a small NN can produce decent results. This documentation is part of a sensory project of the Frankfurt UAS with the goal to create a classifier with a high discriminatory strength. The type of classifier was not predefined. To complete this goal, we decided to choose a NN as classifier. The algorithm for the classifier was implemented in the redpitaya. The content of this project is a proof of concept.

3 Objectives

The goal of this project is to develop a sensor for machine decision making, which can distinguish between different classes of objects. The classes are soft object and hard objects. The focus of this project is therefore on the finding and implementation of a machine decision making and machine learning (ML) algorithm with high discriminatory power. Another requirement is that switching between training and prediction modes is to be realized via User Datagram Protocol (UDP). Furthermore, a variable number of features shall be taken from an algorithm developed by group S1 via a standardized interface. Regarding the starting of the measurement of the sensor, there is a requirement that the sensor should start the measurement by itself when it is switched on. All these goals shall be achieved in the following work.

4 Theoretical basics

This chapter deals with the theoretical basics that are relevant for this project work. The methodology and type of ML used are explained in more detail. Furthermore, the required software is presented, and the used sensor and its properties are explained.

4.1 Typical Classifier

A ML classifier is used to automatically assign different data into specific classes. An example is the automatic classification of emails as spam or not spam.

4.1.1 Box Classifier

Each class generates a box around the mean of the training samples. The size of the Box is determined by the standard deviation. If a value is in more than one box, the box with the smallest standard deviation product gets selected [1].

If the values do not fall within any class box, its classified as undefined.

4.1.2 Support Vector Machine

A Support Vector Machine (SVM) is a mathematical method that uses a statistical approach to assign data to different classes. The determined class boundaries are called large margin classifiers. SVM supports both linear and non-linear classifications. For non-linear classifications, the kernel method is used to project nonlinear separable samples onto another higher dimensional space (hyperplane). Typically, SVM is used for image, text, or handwriting recognition [2].

4.1.3 Neural Network/Deep Learning

A NN is a computational learning system. The concept is inspired by the human brain. An input value gets a weight assigned for each connected neuron. The hidden layer contains the neurons. A neuron activates depending on the received value and the used activation function. The output of the neuron gets another weight assigned to it. This value is then sent to the classification layer. The class with the highest calculated value is the detected class. There is no undefined class. Deep learning (DL) uses the same principles but makes use of several hidden layers [3].

4.2 Software used

The code is written in C using visual studios 2019 as an IDE. The algorithm is written in the software version AIS_V0.12c. The current implementation of the algorithm is not compatible with newer versions of the software. The use of the algorithm in a newer software version requires some adjustments. All Artificial Intelligence (AI) elements need to be transferred to a separate file. Furthermore, the input features need to be normalized using the max-value of the input array. The normalization of the input is crucial for the prediction. The used version has some minor software bugs, but they do not interfere with the goal of this project.

4.3 Hardware used

The hardware used in this project is the SRF-02 Ultrasonic Sensor (US) from Robotelectronics. The function of an US is that it cyclically emits short and high-frequency sound pulses. When these sound pulses hit an object, a reflection occurs. Due to this reflection, echo signals return to the US. The US calculates the distance of the object using the resulting time interval between emission and reception [4].

The key data of the sensor used here are as follows [5]:

- Range - 16cm to 6m.
- Power - 5V, 4mA type.
- Frequency - 40KHz.
- Size - 24mm x 20mm x 17mm height.

5 State of the art

5.1 Hardware

The project Hardware consists of a redpitaya and a SRF02 US. The sensor is connected via a custom circuit board Michalik_RPUSv1.1 (see Fig. 1).

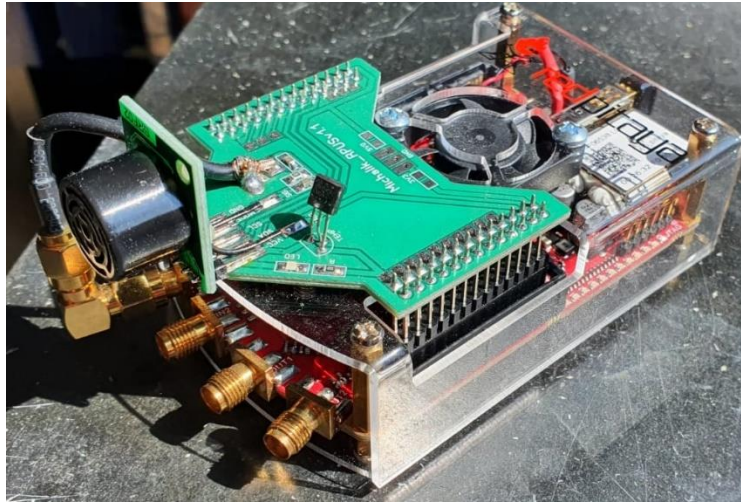


Fig. 1: Sensor combi

5.2 Software

The supplied version at the start of the project was AIS_V0.12c. This version of the software provided the following functionalities as seen in Fig. 2:

- Graphic user interface (GUI)
- Establishment of a Server connection (1)
- UDP – communication between sensor and PC (2)
- Programming redpitaya (3)
- Command prompt (4)
- Distance measurement (console output) (5)
- Fast Fourier Transformation (FFT) plot (6)
- Analog Digital Converter (ADC) plot (7)
- Underlying calculations

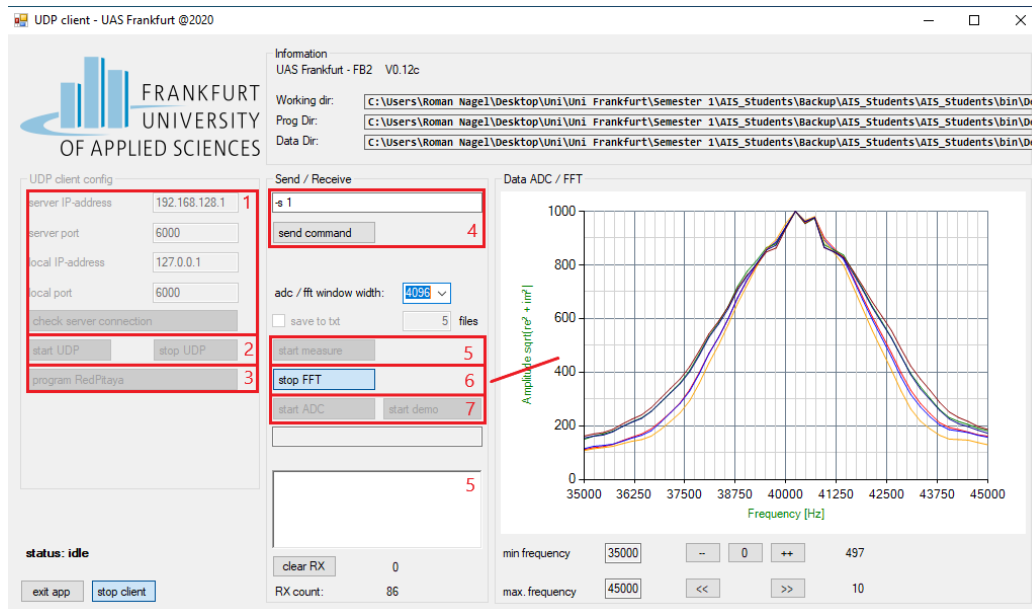


Fig. 2: GUI

6 Realization

6.1 Hardware model

The US is placed at a distance of 250 cm from the wall. The distance to the wall is static to provide a consistent testing environment. During the training of the classifier the human will stand in front of the sensor in varying distances and poses. The test setup displayed in Fig. 3.

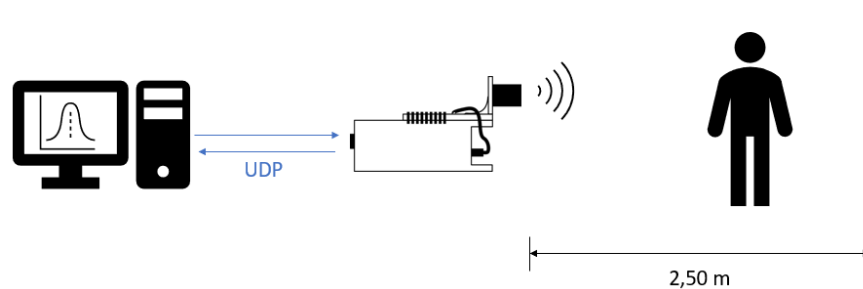


Fig. 3: Test setup

6.2 Software model

The model used for the project is a DL algorithm which is a deviation of a NN. Part of the Project goal is to develop a classifier with variable input size. Further the classifier needs to be able to use any kind of data provided to it.

The architecture of the NN allows for such a variance of application. The initial number of neurons for the input layer can be set to the required size. From this point on the input size is determined and all inputs need to match the same format.

The neural network consists of three layer-types:

- Input layer: it is the linearized version of the provided data
- Hidden layer: feature extraction
- Output layer: prediction of the NN

Each neuron of the input layer is connected to every neuron of the hidden layer. Each neuron of the hidden layer is again connected to every neuron of the next layer. The last layer is the output layer. The difference between a NN and a deep leaning algorithm is the number of hidden layers. A simple NN has only a single hidden layer while the DL algorithm has at least two.

The Architecture of our net consists of three hidden layers (as seen in Fig. 4). They are decreasing in size towards the output layer (120, 60 ,40). Without the decrease in size the net would need 34080 weights. For the trained network, the input layer has 42 neurons and 2 classes which means our net only needs 14720 (43.2% of the NN with even hidden layer size). Additionally, there is a performance increase since there are less weights to be trained.

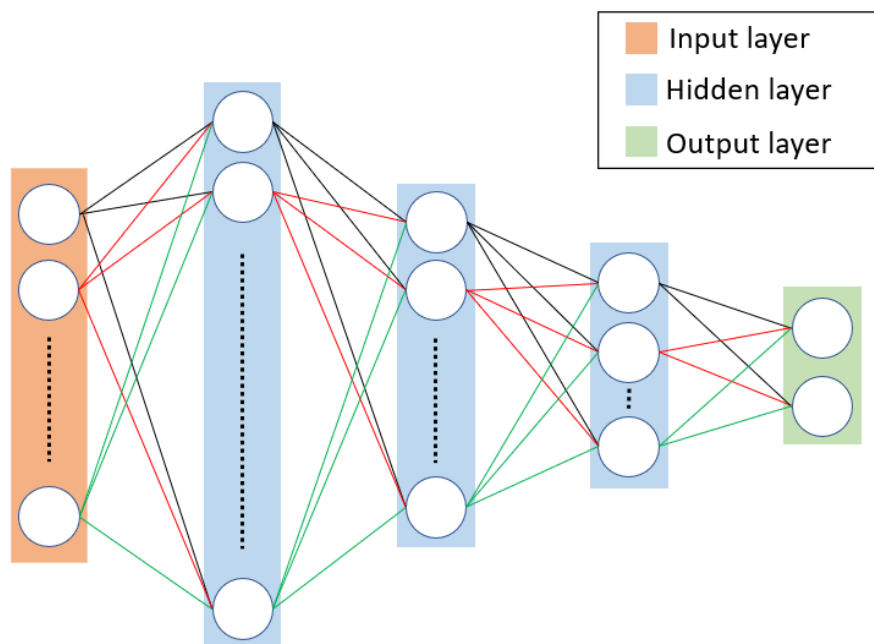


Fig. 4: Neural Network Architecture

The hidden layers are mostly independent from the in- and output layers. Only the first and last hidden layer are influenced. This means that the input is disconnected from the output and allows the net to identify features on its own. The precision of a NN depends on the classes it has to predict, the number of training inputs, the variance in the training data as well as the initial learning rate, the decay, and the used activation function. Furthermore, the number of hidden layers influence the prediction. NNs are generally resource intense algorithms. Even a small net with one or two hidden layers consists of thousands of weights.

Every single connection has a weight assigned to it. The input to a neuron in a layer is the sum of all products of a weight and the value of its neuron in the previous layer.

6.3 Programming

To have an organized program the NN related variables are stored together in a struct defined as *net_param_t*. The struct is a global type and is made available by declaring a variable of this type.

The struct contains architectural and informational variables that are important for the training. Code 1 shows the contained variables in the struct.

```
#define HIDDEN_LAYERS      3
// network struct
typedef struct
{
    uint16_t      class_cnt;
    uint16_t      feature_cnt;
    uint16_t      class_predict;
    char          *class_names[10];
    int           layer_size[2+HIDDEN_LAYERS];
    int           *epoch;
    int           *trained;
    float         **layer;
    float         **error;
    float         **weight;
}net_param_t;
// net related global vars
static net_param_t      nets_param_s;
```

Code 1: Struct *net_param_t*

The Variables *class_cnt*, *feature_cnt*, *class_predict*, *class_names*, *epoch* and *trained* are informational variables. They contain general descriptions about the current state of the training and the network itself. The variables *layer_size*, *layer*, *error*, and *weight* are architectural. They store the information about the actual design of the network and values needed for the training.

The struct itself does not allocate the memory needed for the NN. The code displayed in Code 2 allocates the memory using the malloc function and stores the pointers in the struct. Furthermore, the code checks if the allocation was successful. In case the process failed an error-code is returned to identify the cause. If the allocation was successful an allocation message is sent via UDP as a confirmation for the user. The provided code can be used as a general guideline for allocation of memory in this project.

NOTE: in the current state of the project, we are not capable to free the allocated space. This is due to the fact, that we are working with double pointers in a struct to allocate memory which requires a high skill level of coding. Unfortunately, this exceeds our capabilities.

```

nets_param_s->layer = (float**)malloc((2 + HIDDEN_LAYERS) * sizeof(float*));
if (nets_param_s->layer == NULL)return NET_ERR_INIT_L;
else
{
    send_via_udp("--layer_alloc");
}
nets_param_s->error = (float**)malloc((2 + HIDDEN_LAYERS) * sizeof(float*));
if (nets_param_s->error== NULL)return NET_ERR_INIT_E;
else
{
    send_via_udp("--error_alloc");
}
nets_param_s->weight = (float**)malloc((2 + HIDDEN_LAYERS) * sizeof(float*));
if (nets_param_s->weight == NULL)return NET_ERR_INIT_W;
else
{
    send_via_udp("--weight_alloc");
}

```

Code 2: allocation guideline

The error, layer and weights are two dimensional and therefore need a secondary allocation of memory (as seen in Code 3). The size of each layer is different. The information about the layer size is stored in the variable *layer_size*. The result of the malloc function is then stored in the previously allocated array. Otherwise, it follows the same guidelines, as described in the previous allocation paragraph.

```

//net_param_s = nps
for (int i = 1; i < 2 + HIDDEN_LAYERS; i++)
{
    nps->layer[i] = (float*)malloc((nps->layer_size[i] + 1) * sizeof(float));
    if ((nps->layer[i] == NULL) && (i < HIDDEN_LAYERS + 1))return NET_ERR_HL;
    else if ((nps->layer[i] == NULL) && (i == HIDDEN_LAYERS + 1))return NET_ERR_CL;
    else {
        sprintf(msg, "--layer[%i]", i);
        send_via_udp(msg);
    }
    nps->error[i] = (float*)malloc((nps->layer_size[i]) * sizeof(float));
    if ((nps->error[i] == NULL) && (i < HIDDEN_LAYERS + 1))return NET_ERR_HL_E;
    else if ((nps->error[i] ==NULL) && (i == HIDDEN_LAYERS + 1))return NET_ERR_CL_E;
    else {
        sprintf(msg, "--error[%i]", i);
        send_via_udp(msg);
    }
    nps->weight[i] = (float*)malloc(((nps ->layer_size[i]) *
        (nps ->layer_size[i - 1]) + 1) * sizeof(float));
    if ((nps ->weight[i] == NULL) && (i <= HIDDEN_LAYERS + 1))return NET_ERR_W;
    else {
        sprintf(msg, "--weight[%i]", i);
        send_via_udp(msg);
    }
}
}

```

Code 3: secondary allocation of memory

An activation function determines the output strength of a neuron depending on its value. If the activation function of the NN is a ReLU-function it is usually initialized using the He-Initialization method. With the He-Initialization the weights are initialized with respect to the size of the previous layer. It is a controlled initialization which leads to a faster regression of the gradient [6]. For the initialization of a NN, all weights are randomized. The reason for this lies in the nature of the trainings from NNs. The algorithm used to initialize the net is shown in Code 4.

```
// net_param_s = nps
for (int j = 1; j < 2+ HIDDEN_LAYERS; j++)
{
    //HE INITIALIZATION
    scale = sqrtf((2.0f) / nps->layer_size[j - 1]);

    if (j!=1+HIDDEN_LAYERS)scale = scale * 1.41; // RELU
    for (int i = 0; i < (nps ->layer_size[j]) * (nps ->layer_size[j - 1] + 1); i++)
    {
        nps ->weight[j][i] = scale * ((float)rand() / RAND_MAX - 0.5);
    }

    for (int i = 0; i < (nps ->layer_size[j]); i++) // BIASES
    {
        nps ->weight[j][( nps ->layer_size[j - 1] + 1) * (i + 1) - 1] = 0.0;
    }
}
```

Code 4: Initialization algorithm

The training is a two-step process which requires the two functions *forwardProp* and *backwardProp* (see code in Appendix). The first step is the forward propagation also known as prediction and the second step is the backward propagation. The forward propagation calculates a prediction of a class, to a given input. This prediction is used to calculate an error. For the error-calculation the actual class for the given data must be known. The calculated error will then be used to adjust the weights in each layer. The adjustment starts in the output layer and propagates towards the input layer, hence the name “backward propagation”. The weights can be accessed by using a pointer on the struct. NNs are nonlinear which means that for the optimization of their weights they require the use of a method of gradient decent. A small deviation in the starting point can have a big influence on the outcome. This difference is the reason for the random initialization of the weights.

6.4 UDP Commands

To use the functionalities for a NN on the sensor, the necessary commands are listed in

Table 1. These commands are entered in the command prompt of the GUI. Some of the commands need additional parameter for a successful execution. Due to a lack of time the parameters are not failsafe and need to be executed in the correct order. Otherwise the program might crash.

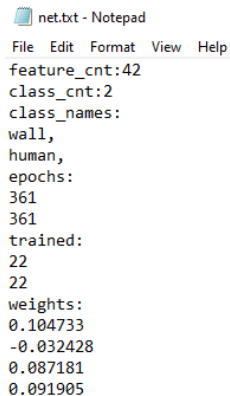
Table 1: UDP commands

UDP commands	Parameter	Functionality
'b'	0	Saving of network
'c'	1: [class number] 2: [class name]	Map class names
'g'	1: [class number]	Getter for class names
'l'	0	Loading of network
'p'	1: [learn] 2: [decay]	Manual adjustment of learn and decay
'n'	1: [class count]	Initialization of a network
't'	1: [class number] 2: [epochs]	Training of network

The commands 'b' and 'l' are the only ones without additional parameters.

The command 'b' saves the currently available network to a text-file on the sensor. The file is saved at: */root/redpitaya/Examples/Communication/C/iic* under the name "net.txt".

The file stores the architecture of the net and the weights (see Fig. 5: Text-fileFig. 5). Its structure contains the number of inputs as *feature_cnt* the number of classes as *class_cnt*. The mapped names are in the lines beneath *class_name*. Due to coding reasons the names are separated by a ','. Each of the classes trained epochs and training iterations are stored as well. At last, the file contains the weights of the network. The weights are saved starting from the input layer towards the output layer.



```
net.txt - Notepad
File Edit Format View Help
feature_cnt:42
class_cnt:2
class_names:
wall,
human,
epochs:
361
361
trained:
22
22
weights:
0.104733
-0.032428
0.087181
0.091905
```

Fig. 5: Text-file

With the command 'l' the network gets loaded from the net.txt file. The loading function first extracts the size of the network and initializes it with random weights. Next the weights get filled with the weights stored in the file and maps the other elements of the network accordingly.

The order of the following commands is important to avoid unwanted crashes. The first step is to use the command 'n' for the initialization of a new network. The parameter class count is separated by a blank space from the command character.

Currently there is no reinitialization possible! If a network needs to be reinitialized or loaded the sensor must be reprogrammed.

When the network is initialized, the next step is to map the names to the classes. To map the classes the command 'c' is used. It has two parameters. The first parameter is the class number that is going to be mapped and the second is the name. The name of the class cannot have more than 10 characters. The class number starts at "0".

Now the command 't' can be used to train the network. The first parameter of the command is the class number that is going to be trained. The second parameter specifies the number of epochs the class is going to be trained. To perform a successful training best practices would be to start with a small epoch number. As a rule of thumb, a value of 10 epochs is a good starting point for the first few iterations of the training. Further the training should be performed in a way that the classes are altered with every training iteration to avoid overfitting.

To have a more diverse training functionality it is possible to adjust the control variables learning and decay. The standard value for *learn* is 0.1 and for *decay* is 0.95. The command to change the value of these variables is 'p'. The first parameter of the command is the value for *learn* and the second is for *decay*.

The last command 'g' has only a class number as a parameter. This command is a getter-function. It returns the name of the class for the given class number.

6.5 Training

The training is performed as described in the previous chapter. Initialization of the network and mapping of the class names were performed over the UDP command prompt.

At an average a training of a network with two classes required approximately 1500 individual training samples. This led to an average prediction certainty of 85%. It has to be noted that it takes a considerable amount of time to perform a training, since the speed of the training depends on the speed of the processor. Most trainings took up to an hour to get a decent prediction reliability.

In the current state the training uses the calculated FFT-Data as an input to the network. The use of the feature extraction from group S1 could not be implemented due to a lack of time created by the use of different software versions.

Fig. 6 shows the FFT of a wall. The general shape is even with little to no spiking. It is a widened bell shape and peaks at a frequency around 40500 Hz. The curve does not change too much over several measurements.

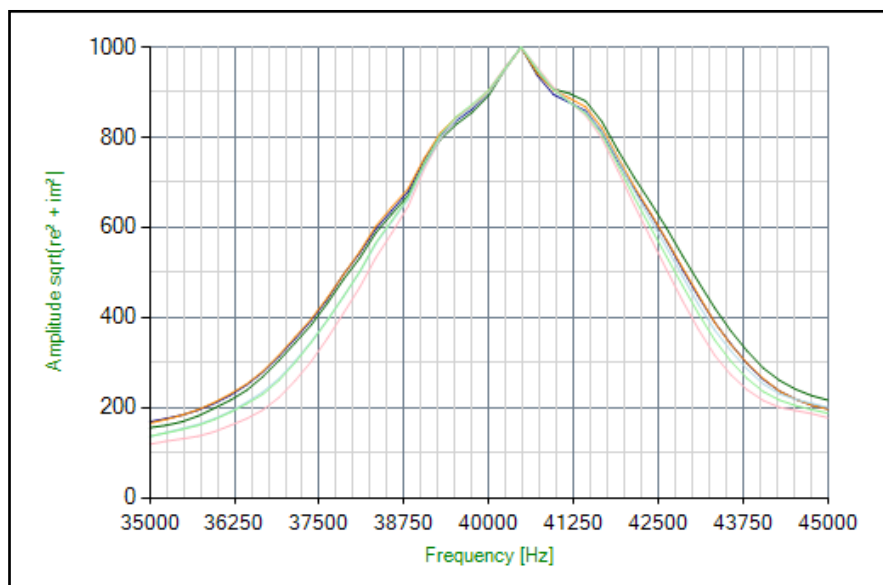


Fig. 6: Wall FFT

Fig. 7 shows the FFT of a human facing the sensor. Its general curve is a narrower curve than the wall. It has a high spikiness and a high deviation between each measurement. The peak is located and around 40,500 Hz within an interval between 39,000 and 41,500 Hz. Its appearance differs a lot from the wall's FFT. This by itself means that the NN can be trained.

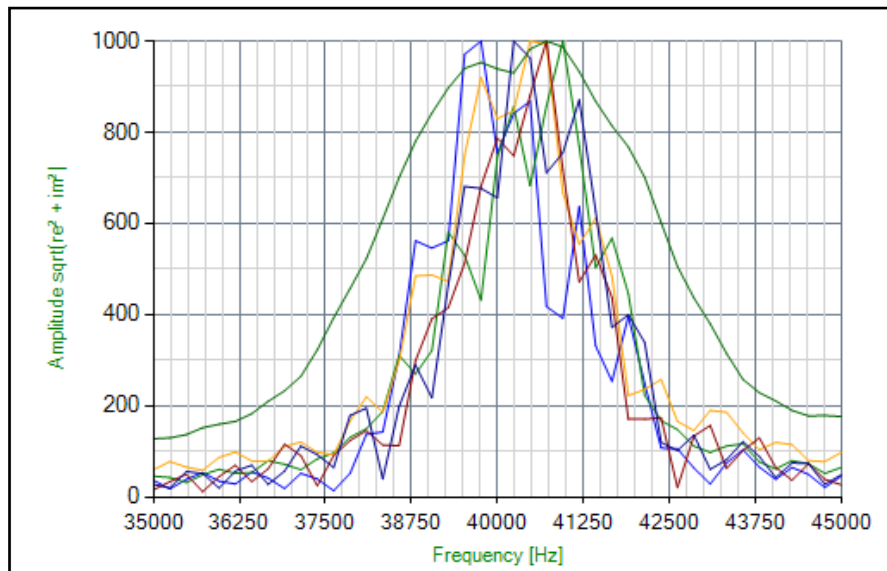


Fig. 7: Human FFT

The FFT in Fig. 8 is generated by a chair with the rest facing to the right. In comparison to the other two FFTs the FFT shares similarities with both. It shares the width with the human but has the smoothness of the wall. In theory it should be possible to train more than just soft and hard objects.

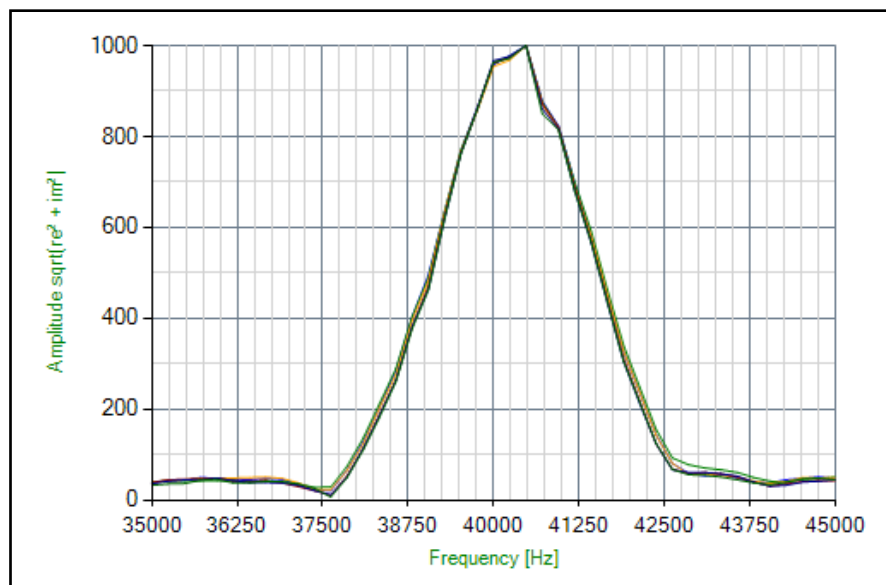


Fig. 8: Chair FFT

7 Conclusion

It is possible to implement a trainable NN on the sensor. The network architecture must factor in the limitations induced by the hardware. As for the set goal, the classification of soft and hard objects is possible. The data used for the training shows a difference which is significant enough. The used objects were wall (hard object) and human (soft object). The only incomplete points of the project are the use of the feature extraction provided by group 1 and the ability to reinitialize a network.

For future improvements it is recommended to implement the reinitialization of a network. To improve the trainability of the network it would be reasonable to add a learning rate adjustment, a dropout rate as well as a pooling and convolutions. Further it should be considered to design a data generator to disconnect the size of the input to the net and allow for a variance in the data.

References

- [1] "spatial-analyst.net," [Online]. Available: http://spatial-analyst.net/ILWIS/htm/ilwisapp/classify_functionality.htm. [Accessed 17 April 2021].
- [2] "bigdata-insider.de," [Online]. Available: <https://www.bigdata-insider.de/was-ist-eine-support-vector-machine-a-880134/>. [Accessed 16 April 2021].
- [3] "neuralnetworksanddeeplearning.com," [Online]. Available: <http://neuralnetworksanddeeplearning.com/chap1.html>. [Accessed 17 April 2021].
- [4] "microsonic.de," [Online]. Available: <https://www.microsonic.de/de/service/ultraschallsensoren/prinzip.htm> . [Accessed 15 April 2021].
- [5] "exp-tech.de," [Online]. Available: <https://www.exp-tech.de/sensoren/entfernungnaeherung/4459/srf02-ultrasonic-ranger>. [Accessed 15 April 2021].
- [6] "holehouse.org," [Online]. Available: http://www.holehouse.org/mlclass/01_02_Introduction_regression_analysis_and_gr.html. [Accessed 16 April 2021].
- [7] "kaggle.com," [Online]. Available: <https://www.kaggle.com/cdeotte/mnist-cnn-coded-in-c-0-995>. [Accessed 25 February 2021].

Appendix

```

int forwardProp(net_param_t* nets_param_s)
{
    char msg[40];
    float sum = 0.0;
    float esum = 0.0;
    float max = 0.0;
    int i, j, k;
    int imax = 0;
    // Input/Feature Layer
    for (i = 0; i < (nets_param_s->layer_size[0]); i++)
    {
        nets_param_s->layer[0][i] = (float)fft_Buff[i] / 1000;
    }
    //Hidden Layer -ReLU activation
    for (k = 1; k < 1 + HIDDEN_LAYERS; k++) //access Hidden
        Layers
        for (i = 0; i < nets_param_s->layer_size[k]; i++) { //access values of
            Hidden Layer
            sum = 0.0;

            for (j = 0; j < nets_param_s->layer_size[k - 1] + 1; j++)
            {
                sum += nets_param_s->layer[k - 1][j] * nets_param_s->weight[k]
                    [i * (nets_param_s->layer_size[k - 1] + 1) + j];
            }
            if (sum >= 0.0)
            {
                nets_param_s->layer[k][i] = sum;
            }
            else
            {
                nets_param_s->layer[k][i] = 0.0;
            }
        }

    }
    // OUTPUT LAYER - SOFTMAX ACTIVATION
    esum = 0.0;
    for (i = 0; i < nets_param_s->layer_size[1 + HIDDEN_LAYERS]; i++) {
        sum = 0.0;
        for (j = 0; j < nets_param_s->layer_size[HIDDEN_LAYERS] + 1; j++)
        {
            sum += nets_param_s->layer[HIDDEN_LAYERS][j] * nets_param_s->weight
                [1 + HIDDEN_LAYERS][i * (nets_param_s->layer_size[HIDDEN_LAYERS]
                    + 1) + j];
        }
        if (sum > (float)GRADIENT_LIMIT) return NET_ERR_GR; //GRADIENT EXPLODED
        nets_param_s->layer[1 + HIDDEN_LAYERS][i] = expf(sum);
        esum += nets_param_s->layer[1 + HIDDEN_LAYERS][i];
    }
    // SOFTMAX FUNCTION
    max = nets_param_s->layer[1 + HIDDEN_LAYERS][0]; imax = 0;
    for (i = 0; i < nets_param_s->layer_size[1 + HIDDEN_LAYERS]; i++) {
        if (nets_param_s->layer[1 + HIDDEN_LAYERS][i] > max) {
            max = nets_param_s->layer[1 + HIDDEN_LAYERS][i];
            imax = i;
        }
        nets_param_s->layer[1 + HIDDEN_LAYERS][i] = nets_param_s->layer[1 +
            HIDDEN_LAYERS][i] / esum;
        sprintf(msg, "Out: %2.3f at: %d", nets_param_s->layer[1 +
            HIDDEN_LAYERS][i], i);
        send_via_udp(msg);
    }
    return imax;
}

```

```

int backwardProp(net_param_t* nets_param_s, int is_class)
{
    char msg[40];
    int i, j, k, p, r = 0;
    uint8_t resc = 0;
    //int total_epochs=0;
    float der = 1.0;
    float pow_scaler = 1.0;

    p = forwardProp(nets_param_s);
    if (p == NET_ERR_GR) {
        //return NET_ERR_GR;
    }
    if (p == is_class) // test if prediction and class are identical
    {
        r = 1;
    }
    // OUTPUT LAYER - CALCULATE ERRORS
    for (i = 0; i < nets_param_s->layer_size[1 + HIDDEN_LAYERS]; i++) {
        if ((nets_param_s->layer[1 + HIDDEN_LAYERS][i] > 0.998) && (!resc))
        {
            resc = 1;
            decay -= 0.005;
        }
        pow_scaler = (nets_param_s->epoch[i]) / (nets_param_s->trained[i]);
        nets_param_s->error[1 + HIDDEN_LAYERS][i] = learn * (0 - nets_param_s->layer[1 + HIDDEN_LAYERS][i]) * powf(decay, 1 + pow_scaler);

        if (i == is_class) {
            nets_param_s->error[1 + HIDDEN_LAYERS][i] = learn * (1 -
                nets_param_s->layer[1 + HIDDEN_LAYERS][i]) * powf(decay, 1 +
                pow_scaler);
            sprintf(msg, "e: %2.3f lr: %2.3f epoch: %d", nets_param_s->error[1
                + HIDDEN_LAYERS][i], learn * powf(decay, 1 + pow_scaler),
                nets_param_s->epoch[i]);
            send_via_udp(msg);
            nets_param_s->epoch[i] += 1;
            if (nets_param_s->layer[1 + HIDDEN_LAYERS][i] == 0) return
                NET_ERR_GR; // GRADIENT VANISHED
        }
    }
    // HIDDEN LAYERS - CALCULATE ERRORS
    for (k = HIDDEN_LAYERS; k > 0; k--) {
        for (i = 0; i < nets_param_s->layer_size[k]; i++) {
            nets_param_s->error[k][i] = 0;
            if (nets_param_s->layer[k][i] > 0) // ReLU DERIVATIVE
            {
                for (j = 0; j < nets_param_s->layer_size[k + 1]; j++) {
                    nets_param_s->error[k][i] += nets_param_s->error[k + 1][j]
                        * nets_param_s->weight[k + 1][j] * (nets_param_s->layer_size
                            [k + 1] + i) * der;
                }
            }
        }
    }
    // UPDATE WEIGHTS - GRADIENT DESCENT
    for (k = 1; k < 2 + HIDDEN_LAYERS; k++)
    {
        for (i = 0; i < nets_param_s->layer_size[k]; i++)
        {
            for (j = 0; j < nets_param_s->layer_size[k - 1] + 1; j++)
            {
                nets_param_s->weight[k][i * (nets_param_s->layer_size[k - 1] +
                    1) + j] += nets_param_s->error[k][i] * nets_param_s->layer[k
                    - 1][j];
            }
        }
    }
    return r;
}

```