Faculty of Electronics and Information Technology
Warsaw University of Technology

Image and Speech Recognition

Visual digits recognition on embedded system basing on neural networks

Jakub Wieczorek

Szczecin, 2020

Contents

1. Introduction

Digits recognition is a well-known and thoroughly explained subject in the image processing field. The outcome of those studies has significant impact for many aspects in people's lives. We used to provide credit curd number by hand some time ago, until the cell phones started to do it for us with the help of the camera. Automatic opening the gate basing on the plates numbers for entitled cars in the parking or motorway gates for those whose owners paid the toll upfront. These examples are just a pick of an iceberg of the potential digits recognition use cases.

Image recognition and deep learning algorithms became globally available not only for scientists, but also for ordinary individuals, especially because of high level libraries like Tensorflow or PyTorch where the exact knowledge of how the particular algorithm works is often irrelevant for the sake of the API. Developer has to invoke proper function with the desired parameters, but the function itself which for example creates and trains neural network is in the form of the black box. Enormous amount of tutorials and resources also make the entry threshold relatively low for the person without prior knowledge in the topic.

MNIST Crucial role in many if not the most image recognition algorithms and deep learning plays a dataset. It is necessary to gather significant amount of training data to properly teach for example the neural network and subsequently test in order to evaluate the quality of recognition. The process of gathering such a huge amount of data is more often the not tedious and takes a lot of time. This way MNIST database emerged for the sake of digits recognition. It contains 70000 handwritten digits, where 60000 and 10000 are consequently images for training and testing purposes [1]. Digits were written by high school students and employees of the United States Census Bureau. Figure 1.1 shows several images from the MNIST dataset.

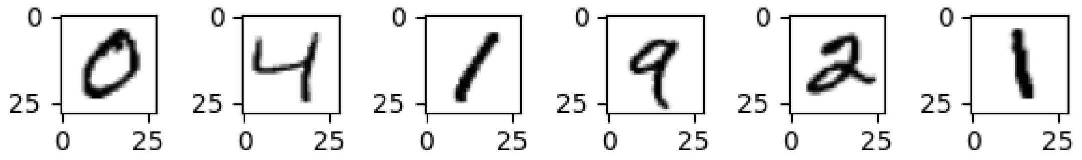


Figure 1.1. MNIST dataset example

To give an impression of the current level of digits recognition algorithms basing on MNIST dataset, several machine learning methods are juxtaposed on the Table 1.1. It is very easy to

Type	Classifier	Error rate (%)
Linear classifier	Pairwise linear classifier	7,6 [2]
K-Nearest Neighbors	K-NN with non-linear deformation (P2DHMDM)	0,52 [3]
Deep neural network (DNN)	2-layer 784-800-10	1,6 [4]
Deep neural network (DNN)	6-layer 784-2500-2000-1500-1000-500-10	0,35 [5]

Table 1.1. Several deep learning algorithms comparison in the digits recognition basing on MNIST database

take an advantage of this database, whilst working with for example Tensorflow, because the dataset can be downloaded directly using the library API. In this paper MNIST dataset will be exploited only for the training purposes.

How it works To goal is to recognise individual handwritten digits (outside the MNIST database) in real time by the camera. The images are captured in Python, resized to the 28x28 size and sent to the embedded system through UART, where deep feed forward neural network is implemented. Detected digit is displayed on the 7 segment display. System provides two ways of preprocessing the images. Adaptive Gaussian thresholding with the help of Open CV's API and Otsu thresholding implemented from the basis both in pure Python and C (in embedded system). Desired preprocessing might be chosen in the main script of communication application (main.py):

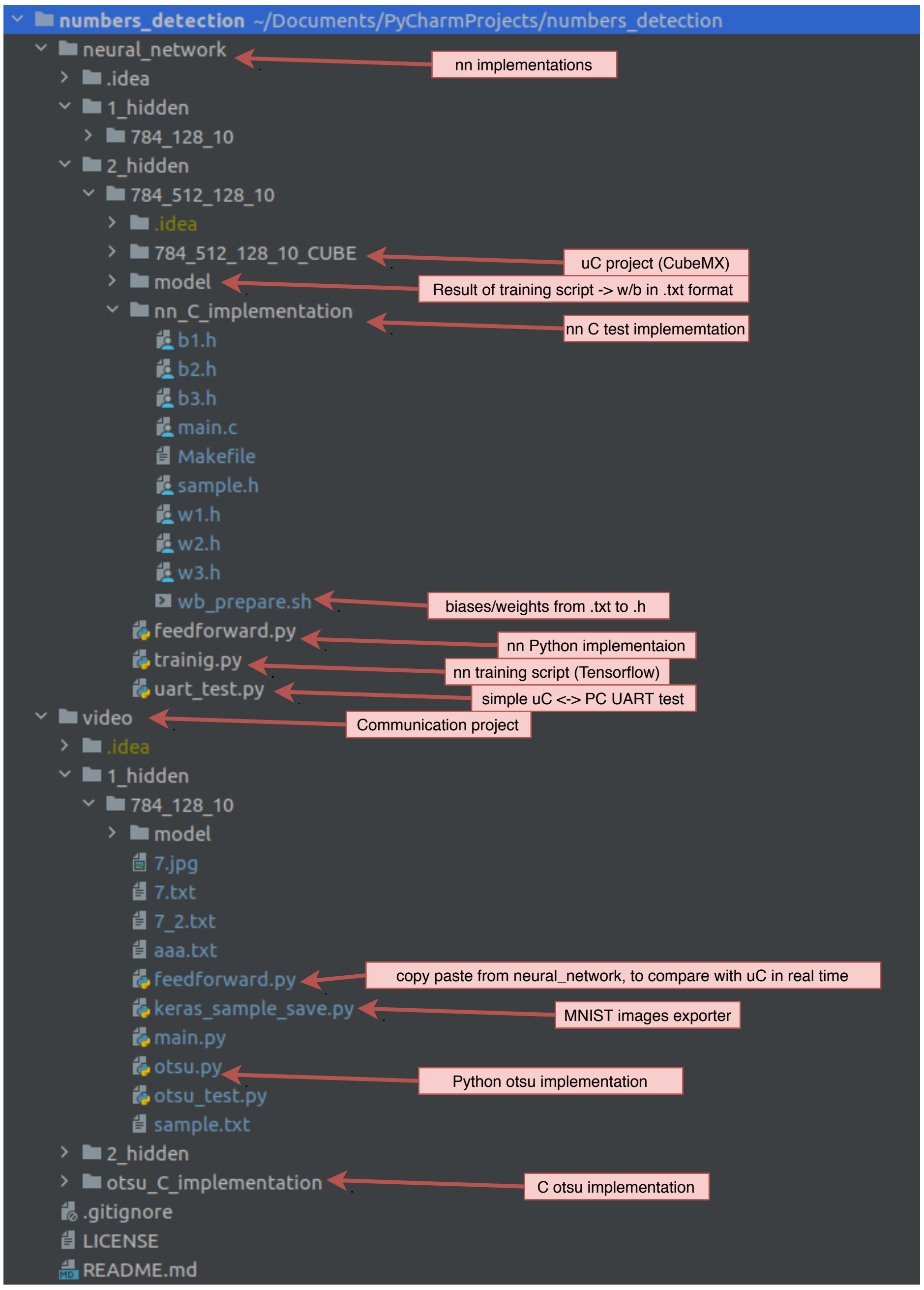
```python
class Preprocessing():
    ADAPTIVE_THRESHOLD = 1
    OTSU = 2
    OTSU_ORIGINAL_IMAGE_NO_SERIAL = 3 # no data transfer to the uC -- just
                                      # for the visualisation. OTSU
                                      # on not resized image, high quality
    OTSU_EMBEDDED = 4

###################################################
# configuration part:
###################################################
SERIAL = False
PREPROCESSING = Preprocessing.OTSU_ORIGINAL_IMAGE_NO_SERIAL
```

Two variables are to be adapted by the user. If the SERIAL flag is true, data are being sent to the microcontroller, otherwise all process of recognition is done on the PC. User chooses the preprocessing algorithm with the PREPROCESSING variable.

Results evaluation and the whole process of recognition is thoroughly discussed in the following parts of the paper.

Project structure Figure 1.2 presents project structure. It is divided into two main sub projects – neural_network with Python and C feed forward implementation in two version: one and two hidden layers, and video, which is an communication project – main.py contains aforementioned SERIAL and PREPROCESSING configuration, so it directly sends an image content to the microcontroller. 1_hidden and 2_hidden contains the same content, but with different amount of neural network hidden layers.



```
numbers_detection ~/Documents/PyCharmProjects/numbers_detection
  neural_network
    .idea
    1_hidden
      784_128_10
    2_hidden
      784_512_128_10
        .idea
        784_512_128_10_CUBE
        model
        nn_C_implementation
          b1.h
          b2.h
          b3.h
          main.c
          Makefile
          sample.h
          w1.h
          w2.h
          w3.h
          wb_prepare.sh
        feedforward.py
        trainig.py
        uart_test.py
  video
    .idea
    1_hidden
      784_128_10
        model
        7.jpg
        7.txt
        7_2.txt
        aaa.txt
        feedforward.py
        keras_sample_save.py
        main.py
        otsu.py
        otsu_test.py
        sample.txt
    2_hidden
    otsu_C_implementation
    .gitignore
    LICENSE
    README.md
```

nn implementations
uC project (CubeMX)
Result of training script -> w/b in .txt format
nn C test implememtation
biases/weights from .txt to .h
nn Python implementaion
nn training script (Tensorflow)
simple uC <-> PC UART test
Communication project
copy paste from neural_network, to compare with uC in real time
MNIST images exporter
Python otsu implementation
C otsu implementation

Figure 1.2. Project structure

2. Preprocessing

Training images are provided in the 28x28 array already preprocessed in the MNIST database, however images provided in the real time even though conceptually the same are significantly different. Two first pictures from the left of the Figure 2.1 present digit 7. First one comes from the MNIST database, second one is a raw picture taken by the camera. The difference is in the background of the digit, which is not purely white, what introduces huge distortion in the process of recognition. Background of the original MNIST image in gray scale is zero, background of the raw image is bigger than zero, maybe not high, but still, it makes the neural network which was trained on the MNIST dataset incapable of detecting the digits correctly. Therefore thresholding is necessary.

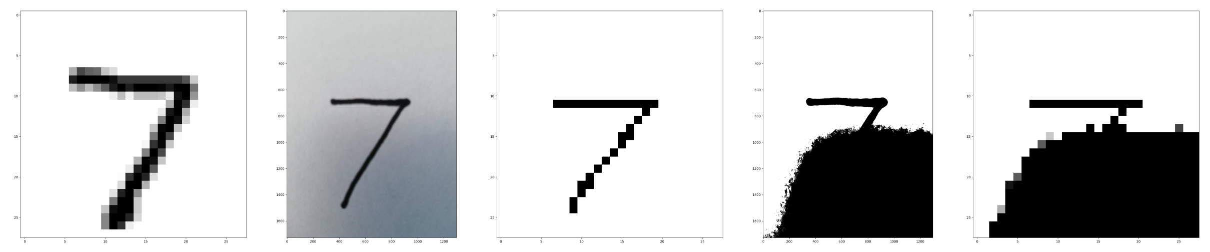


Figure 2.1. Set of preprocessed images. From the left: MNIST example, taken by the camera (raw), adaptive thresholding, Otsu on the original image, Otsu on the previously resized.

Image in the centre shows the outcome of adaptive thresholding. This algorithm helps when the lighting conditions are not always the same, which is exactly the case here. The idea is to mute the background and express the digit. Algorithm calculates the threshold for every pixel based on a small region around it. More narrowly it is a gaussian-weighted sum of the pixel's neighbourhood. Pixel with higher value than the threshold is cast to 255 (black), with lower to 0 (white). Open CV provides an API to take an advantage of this thresholding:

```
arr   = np.asarray(cv2.imread('7.jpg', cv2.IMREAD_GRAYSCALE))
arr2  = cv2.resize(arr, (28, 28))
trunc = cv2.adaptiveThreshold(arr2, 255, cv2.ADAPTIVE_THRESH_GAUSSIAN_C,
    cv2.THRESH_BINARY_INV, 11, 9)
```

2.1. Otsu

Otsu method is named after its author Nobuyuki Otsu. Calculated threshold separates the pixels into two classes foreground and background [6]. The method bases on minimising the between class variance or equally by maximising within class variance, but the first one is significantly faster. Figure 2.2 shows the result of Otsu thresholding for the image captured by laptop build-in camera.

Figure 2.2. Image captured by build-in laptop camera processed by Otsu

2.1.1. Algorithm

The outcome of the method comes down to the division of the whole set of pixels into two classes, foreground and background. Otsu proposed to separate the pixel clusters by maximising the withing class variances, what means maximising the clusters separation. The variance is calculated on the distribution of the pixel intensities, so the histogram of the image, what leads to the probability distribution. This is given by the formula 2.1, where σ_W represents within class variance, σ_b^2 and σ_f^2 variances of the background and foreground classes, W_b and W_f are the probabilities that the classes are separated by the threshold T.

$$\sigma_W^2(T) = W_b(T)\sigma_b^2(T) + W_f(T)\sigma_f^2(T) \tag{2.1}$$

Between class variance is given by the relation 2.2

$$\sigma_B^2(T) = \sigma^2 - \sigma_W^2(T) \tag{2.2}$$

What finally results (after the calculations) in the relation 2.3, where μ_b and μ_f are the weighted means of the particular cluster.

$$\sigma_B^2(T) = W_b(T)W_f(T)(\mu_b(T) - \mu_f(T))^2 \tag{2.3}$$

The following listing shows the algorithm implementation in C:

```c
void otsu(int image[], int size) {
    int histogram[BINS_NUMBER] = {0};
    calculate_histogram(histogram, image, size);

    float sum = 0;
    for(int i = 0; i < BINS_NUMBER; i++) sum += i*histogram[i];

    float sumB = 0, wB = 0, wF = 0, varMax = 0,
    int threshold = 0;
    for(int t = 0; t < size; t++) {
        wB += histogram[t];
        if(wB == 0) continue;
        wF = size - wB;
        if(wF == 0) break;

        sumB += (float)t*histogram[t];
        float mB = sumB / wB;
        float mF = (sum-sumB) / wF;
        float varBetween = wB * wF * (mB-mF)*(mB-mF);

        if(varBetween > varMax) {
            varMax = varBetween;
            threshold = t;
        }
    }
    printf("threshold = %d\n", threshold);
    for(int i = 0; i < size; i++) image[i] = image[i] > threshold ? 0 : 255;
}
```

Where the histogram is calculated in the following way:

```c
void calculate_histogram(int histogram[], int image[], int size) {
        for(int i = 0; i < size; i++) {
                histogram[image[i]]++;
        }
}
```

Two last images from the picture 2.1 present consecutively the result of original image Otsu thresholding and the same image, but previously resized to 28x28 pixels. Adaptive threshold for this image gives better results, however it is not always the case.

3. Neural network

Deep feed forward was chosen as a neural network. Feed forward pass was implemented in pure Python without high level libraries like NumPy or Tensorflow, then reimplemented in C in order to be placed in the microcontroller. However the process of training was outsourced to the Tensorflow in version 2. Weights and biases from the trained multilayer perceptron then are saved in the C header files.

3.1. Feedforward neural network

Feedforward neural network is a multilayer perceptron with several hidden layers extensively used in machine learning along with deep convolutional neural networks and recurrent neural networks. The forward pass, so the process of thinking is relatively simple and performance effective. Figure 3.1 shows deep feedforward neural network with multiple inputs and outputs. In the feed forward pass all inputs are multiplied by weights in the connection with the first hidden node what results in the net input $z_1[0] = i[0]w_1[0][0] + i[1]w_1[1][0] + \cdots + i[k]w_1[k][0] + b_1[0]$. Subsequently the net input z is applied to the activation function. The process is performed for every node in every hidden and output layer, where output from the particular node is an input of the next node. It is worth noting that the number of nodes in the hidden layers not necessarily has to be equal. Formal process of the feed forward pass for particular the node is written in 3.1, where f_k represents activation function for the layer k which is either ReLU for hidden layers or sigmoid for the output one. Rectified linear unit (ReLU) has an advantage over sigmoid function used in the output layer, because it does not have vanishing gradient, which prevents neural network from efficient learning. In fact ReLU has the constant gradient. Sigmoid activation function is given by 3.2, whereas ReLU by 3.3.

$$\begin{aligned} z_k[0] &= n_{k-1}[0]w_k[0][0] + n_{k-1}[1]w_k[1][0] + \cdots + n_{k-1}[m]w_k[m][0] + b_k[0] \\ n_k[0] &= f_k(z_k[0]) \end{aligned} \tag{3.1}$$

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

$$f(z) = max(0, z) \tag{3.3}$$

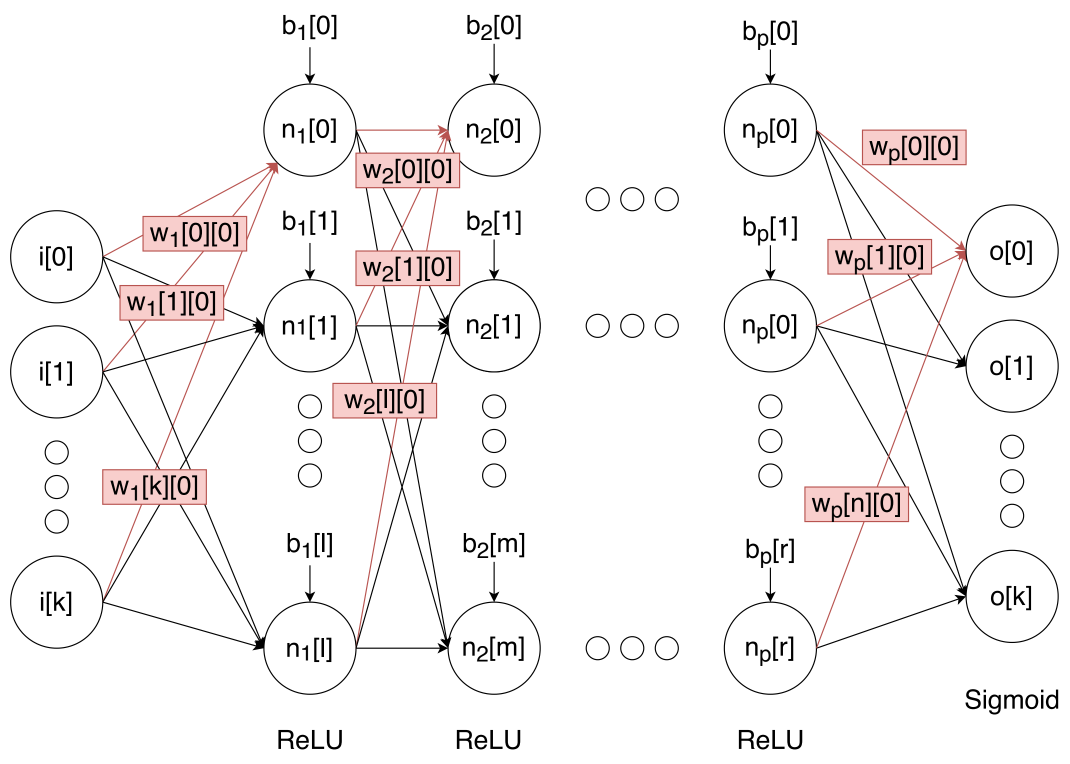


Figure 3.1. Feedforward neural network with multiple inputs and outputs. On the bottom of every layer, activation function was presented.

Python implementation of the feedforward pass is presented on the following listing. This function is invoked for every hidden and output layer. Then the result of particular layer is passed to the consecutive one.

```python
def feedForward(self, inputs, weights_matrix, biases, activationFunction):
    input_length = len(weights_matrix)
    output_length = len(weights_matrix[0])
    x = inputs
    y = []
    z = []  # without activation

    for o in range(0, output_length):  # calculate each output in order
        y.append(0)
        z.append(0)
        for i in range(0, input_length):
            y[o] += weights_matrix[i][o] * x[i]

        z[o] = y[o] + biases[o]

    for o in range(0, output_length):
        y[o] = activationFunction(z[o])

    return y, z
```

And the process of thinking, so the way of feedForward invocation for the consecutive layers. In the example 1 hidden layer.

```python
def think(self, x):
    (self.l1_output, self.l1_z) = self.feedForward(x, self.w1, self.b1,
        self.ReLU)
    (self.output, self.output_z) = self.feedForward(self.l1_output, self.w2,
        self.b2, self.sigmoid)
    return self.output
```

Following two listings show the implementation in C which are directly used by the microcontroller. Moving from Python to C is not trivial in this case, because two dimensional weights matrix is expressed in one dimension. Otherwise it would not be possible to pass different sizes of this matrix. Particular weight is obtained in the line 9. Activation function is passed as a function pointer.

```c
1  void feedforward(const double x[], double z[], const double weights_matrix[],
2      int ROW, int COL, const double biases[], double y[],
3      double (*activationFunction)(double)) {
4          int o, i;
5          double cur_w;
6  
7          for(o = 0; o < COL; o++) {
8                  for(i = 0; i < ROW; i++) {
9                          cur_w = *(weights_matrix + i*COL + o);
10                         y[o] += cur_w * x[i];
11                 }
12 
13                 z[o] = y[o] + biases[o];
14         }
15 
16         for(o = 0; o < COL; o++) {
17                 y[o] = (*activationFunction)(z[o]);
18         }
19 }
```

And the process of thinking:

```c
void think(const double x[], double y[], double z[]) {
        double l1_output[128] = {0}, z1[128] = {0};
        feedforward(x        , z1, w1[0], 784, 128, b1, l1_output, &ReLU);
        feedforward(l1_output, z,  w2[0], 128, 10,  b2, y, &sigmoid);
}
```

3.2. Training process

Training was performed using Tensorflow library in the version 2.2 however implementation in pure Python is also provided (check Appendix A). Time of the training varies between several seconds to roughly a few minutes, depending on the amount of epochs, nodes in the particular layer and the layers itself. Neural network is fully connected. As a lost function sparse categorical cross entropy was used, because the classes are mutually exclusive. It is given by the formula

3.4, where y_a and y_d are the actual outputs of the neural network respectively and L is the loss function.

$$L(y_a, y_d) = -\frac{1}{N}\sum_{i=1}^{N}[y_{a_i}log(y_{d_i}) + (1 - y_{a_i})log(1 - y_{d_i})] \tag{3.4}$$

The training process starts with the model definition, goes through the training itself and finally finishes on the model saving as well as the weights and biases in the proper format. The following listing shows the training implementation for 1 hidden layer with 128 nodes, input with $28 \cdot 28 = 784$ nodes and 10 output neurons.

```
def createModel(self):
    self.model = keras.Sequential([ # layers in sequence
                keras.layers.Flatten(input_shape=(28, 28)),
                keras.layers.Dense(128, activation="relu"),  # fully connected
                keras.layers.Dense(10, activation="sigmoid")
            ])
    self.model.compile(optimizer="adam",
    loss="sparse_categorical_crossentropy", metrics=["accuracy"])

def trainModel(self, epochs_num, save=True):
    self.model.fit(self.train_data, self.train_labels, epochs=epochs_num)
    test_loss, test_acc = self.model.evaluate(self.test_data, self.test_labels)
```

Training takes an advantage of MNIST database as it was already aforementioned. When loaded, images are scaled to the range of 0-1.

```
(self.train_data, self.train_labels), (self.test_data, self.test_labels) =
    keras.datasets.mnist.load_data()
self.train_data = self.train_data / 255.0
self.test_data = self.test_data / 255.0
```

Weights and biases are saved in the proper format, more narrowly they are exported to the .txt file along with the whole model and then transformed to the header files with the help of regexp and specially designed bash script.

4. PC side

In this project camera is directly connected to the PC, but may be external one. The images are captured in real time, resized and depending on the need directly sent to the microcontroller or previously preprocessed. All of that is implemented in Python 3 with the help of OpenCV and serial package. The following listing shows the serial configuration.

```
ser = serial.Serial(
    port='/dev/ttyACM0',
    baudrate=115200,
    parity=serial.PARITY_NONE,
    stopbits=serial.STOPBITS_ONE,
    bytesize=serial.EIGHTBITS
)
```

When the picture is ready to be sent it is flattened, that is changed to one dimensional array. Then every pixel is packed in 3 bytes format and sent. Synchronisation with the microcontroller is sygnalised by 2 special values: ”aaa” stands for the beginning of the transfer and ”bbb” for the end. Right after the beginning characters 3 bytes distinguishing the process of preprocessing are sent – ”OOO” in case of Otsu implemented on embedded system, ”AAA” otherwise (outside microcontroller) for example adaptive thresholding.

```python
def send(ser, data, type):
    ser.write("aaa".encode())
    if type == Preprocessing.OTSU_EMBEDDED:
        ser.write("OOO".encode())
    else:
        ser.write("AAA".encode())

    for i in data.flatten():
        to_send = '{:3d}'.format(i)
        ser.write(to_send.encode())
    ser.write("bbb".encode())
```

It is worth to mention, that the bottleneck of the system is exactly the communication part. Even with the maximum baud rate accessible for UART, sending roughly $28 \cdot 28 \cdot 3 + 6$ bytes takes significant amount of time (around 1,5 second) which stops systems fluency. Time of neural network computation comparing to this is negligible.

5. Microcontroller

Neural network and the whole process of recognition is implemented on the microcontroller STM32H743ZIT6U. Embedded system is placed on the Nucleo board, which is directly connected to the computer through USB. The board was chosen mostly because of high volume of memory. It has 2 Mbytes of Flash memory, where weights and biases resides in the constant arrays and 1 Mbyte SRAM where is the rest of the program, image array for instance. Moreover the core of this device is highly efficient ARM Cortex-M7 at 480MHz, which is desired from the neural network calculations point of view. Figure 5.1 presents the Nucleo board with 7 segment display showing number 5.

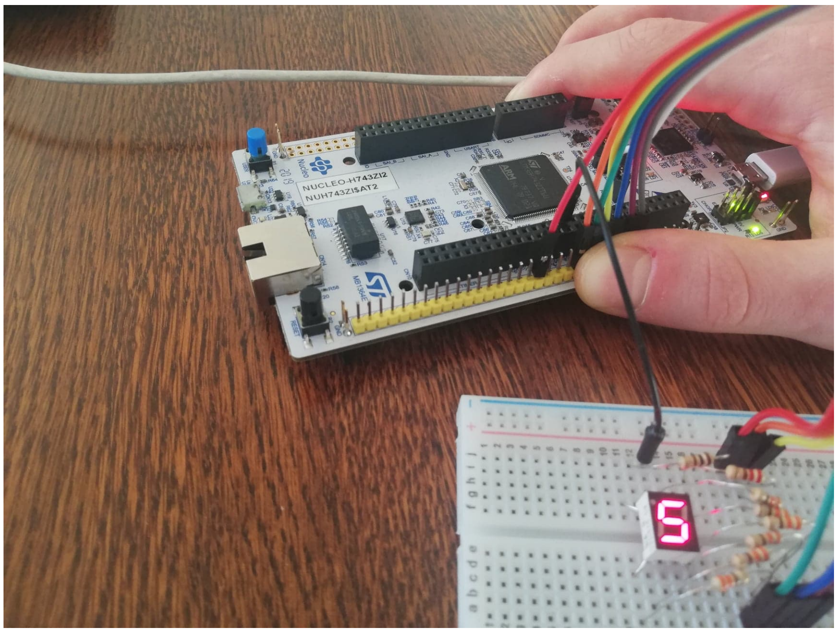


Figure 5.1. STM32H743ZIT6U with 7 segment display, which shows number 5 as a result of the image recognition

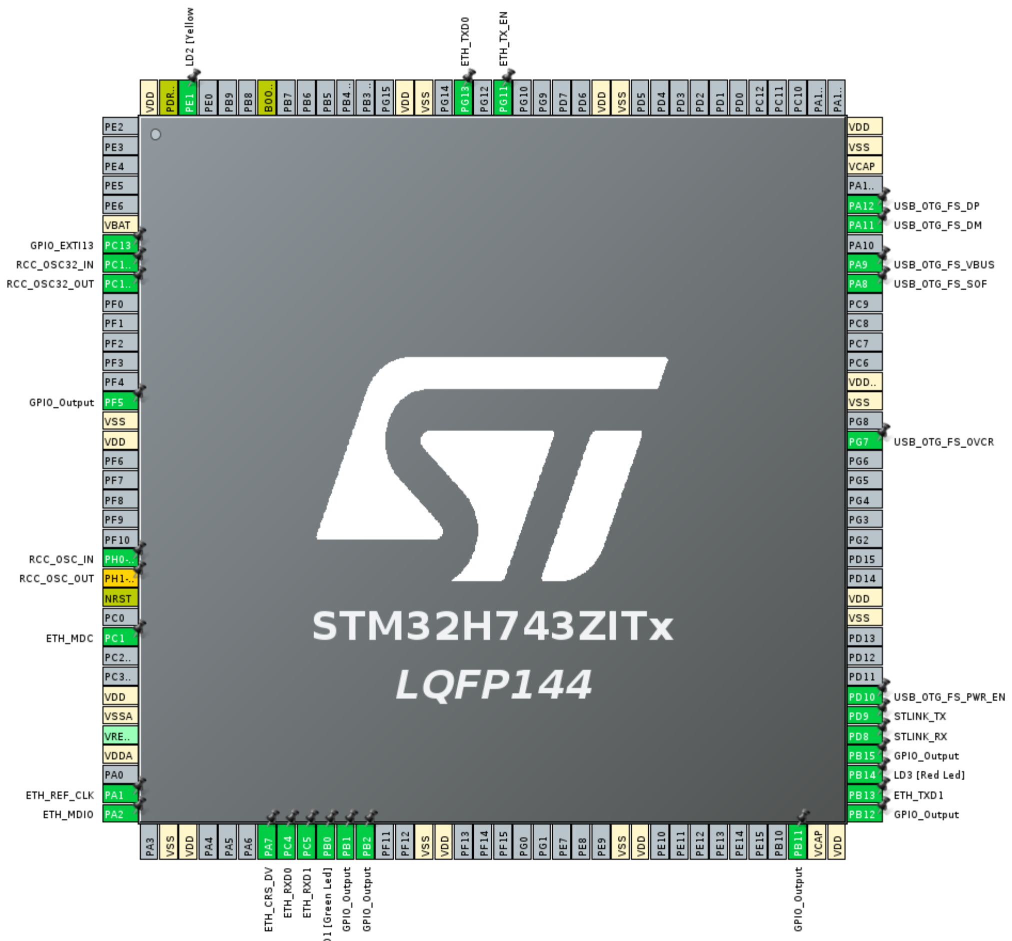


Figure 5.2. Peripherals STM32CubeMX configuration

The explanation of the whole process of peripherals configuration is omitted to focus mainly on the program logic. Nevertheless STM32CubeMX peripherals assignment is presented on the Figure 5.2. Microcontroller's main loop is responsible mainly for the neural network feed forward pass. Nevertheless apart from that there are a few operation worth to explain. First of all "done" flag indicates the end of the image transfer through UART. This flag is set in the interrupt callback, hence volatile. Subsequently Otsu preprocessing is done depending if it is requested via UART message ("OOO" bytes), this flag is also set in the interrupt callback handler. When it is done, image is rescaled and the process of recognition is performed. The result is displayed on the 7 segment display and finally the readiness of transfer is signalised by setting done flag to false.

```c
#define IMAGE_SIZE 784
uint8_t Received[3];
double image[IMAGE_SIZE];
volatile bool done = false;
volatile bool is_otsu = false;
////
while (1) {
    if(done) {
        if(is_otsu) {
            otsu(image, IMAGE_SIZE);
        }

        for(int i = 0; i < IMAGE_SIZE; i++) {
            image[i] = image[i] / 255.0;
        }

        think(image, y, z);
        display_digit(max_idx(y));
        done = false;
    }
}
```

Interrupt handler for the UART receive event is presented on the following listing. The logic is similar to the process of sending the messages in the communication program, discussed previously. Start and finish flag, "a" (first letter of "aaa"), "b" consecutively. Type of the preprocessing, and the image content itself. Passing around 2000 bytes through UART is a bottleneck of the system as it was already stated. Interrupts introduce the notion of overlapping of the two images transmission – during the detection of the first one, second image may be being transmitted. However the question is why not to exploit the DMA? Unfortunately there are several issues with DMA in this type of STM32 [1], so the notion of using it was abandoned.

```c
int counter = 0;

void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart) {
        if(Received[0] == 'a') {
                counter = 0;
        } else
        if(Received[0] == 'b') {
                done = true;
        } else
        if(Received[0] == 'O') {
```

[1] https://community.st.com/s/article/FAQ-DMA-is-not-working-on-STM32H7-devices

```
                is_otsu = true;
        } else
        if(Received[0] == 'A') {
                is_otsu = false;
        } else {
                int value = atoi((const char*)Received);
                image[counter++] = (double)value;
        }

        HAL_UART_Receive_IT(&huart3, Received, 3);
}
```

6. Experiments and results

Feed forward with one hidden and 2 hidden layers was tested. More layers could not fit in the microcontroller's flash memory. First of all accuracy of the training process on the MNIST database is presented on the Table 6.1. Admittedly the training quality rates, namely accuracy

Type	Classifier	Accuracy (%)	Epochs number	Loss
Feed forward	1-layer 784-800-10	0,9721	5	0,0864
Deep feed forward	2-layer 784-512-128-10	0,9793	5	0,0727

Table 6.1. 1 hidden and 2 hidden layers training results

and loss (sparse categorical cross entropy) are similar. However one has to remember, that the training was performed on the MNIST database, but the real testing is in the real time, with hand written, subsequently preprocessed images. From the quality point of view, so how two neural networks behave in such conditions, they are capable of detecting the digits correctly. Deep feed forward neural network from the user experience behaves better than 1 hidden layer neural network, in the same time characterising by the similar performance. It is difficult to evaluate the efficiency quantitatively without huge set of images. Same applies to the preprocessing algorithms. Both of them are able to preprocessed the images satisfactory nonetheless. However as it might have been observed on the Figure 2.1, Otsu is highly vulnerable for lighting differences, hence from time to time some part of the digit may not be exposed correctly (last two images on the Figure 2.1). As an attachment to this report several test recordings may be found in the projects repository[1].

7. Conclusions

In this project deep feed forward neural network was implemented on the embedded system for the sake of visual recognition. Nowadays the biggest obstacle for deep learning algorithms is not the performance of the microcontroller, but the memory volume. Even with 2 Mbytes

[1] `https://gitlab-stud.elka.pw.edu.pl/jwieczo1/numbers_detection`

of the flash memory, it was possible to place there only 2 hidden layers. The bottleneck of the system is the image transfer to the STM32, therefore in the future improvement USB protocol is worth considering. Neural networks were trained basing on MNIST dataset instead of specially prepared images. Nevertheless preprocessing approximated the images captured in the real time, so it was not a problem for the detection. It is worth noting, that a digit should be placed in the centre of the camera to be correctly read.

In addition to the neural network two preprocessing algorithms were compared, where Otsu was implemented from the basis and placed onto microcontroller. Adaptive threshold is more resistant to the light differences, so it behaves slightly better.

Appendices

A. Back propagation

The following appendix shows an implementation of the one hidden layer feed forward back propagation algorithm. Without high level libraries, just pure Python. The difficulties in algorithm implementation grow with the number of layers, because back propagation bases on the gradient calculation, where particular gradient in the layer depends on many factors in the following layers. Thus, for the need of the project back propagation was outsourced to the Tensorflow.

A.1. Output layer

```python
def gradientsOutput(self, inputs, dedo, w, dActivationFunction, acti_arg1):
    """
    @param inputs, The outputs of the last hidden layer
    @param dedo,   Error gradient with respect to the output
    @param w,      An array of weights between output and the
                   hidden layer. Is constant
    @param dActivationFunction in the case of this layer is
                   dSigmoid
    @param acti_arg1 Argument for dActivationFunction. In this case
                   an output of the last layer
    """
    outputs_length = len(w[0])
    inputs_length = len(w)

    dedw = [[0 for o in range(outputs_length)] for i in range(inputs_length)]
    dedb = [0 for o in range(outputs_length)]
    for o in range(0, outputs_length):
        for i in range(0, inputs_length):
            dedw[i][o] = dedo[o] * dActivationFunction(acti_arg1[o])*inputs[i]

        dedb[o] = dedo[o] * dActivationFunction(acti_arg1[o]) * 1
    return dedw, dedb
```

A.2. Hidden layer

```python
def gradients(self, inputs, dedo, w, w2, dActivationFunction, acti_arg1,
    dActivationFunction_2, acti_arg1_2):
    """
    @param inputs, The input of the nn
    @param dedo,   Error gradient with respect to the output
    @param w,      An array of weights between input and the
                   hidden layer. Is constant
    @param w2,     An array of weights between hidden layer
```

```
                    and the output. Gradient of w depends on them.
                    Is constant
    @param dActivationFunction activation for the hidden layer
    @param acti_arg1 Argument for dActivationFunction. If ReLU then
           should be net inputs z.
    @param dActivationFunction2 activation for the output layer
    @param acti_arg2 Argument for dActivationFunction2.
    """
    outputs_length = len(w[0]) # of w!
    inputs_length = len(w)

    dedw = [[0 for o in range(outputs_length)] for i in range(inputs_length)]
    dedb = [0 for o in range(outputs_length)]

    dedohw = [[0 for o in range(outputs_length)] for i in range(inputs_length)]
    dedohb = [0 for o in range(outputs_length)]

    outputs_length2 = len(w2[0])
    inputs_length2 = len(w2)

    for o2 in range(0, outputs_length2):
        dEdActiv = dedo[o2] * dActivationFunction_2(acti_arg1_2[o2])
        for o in range(0, outputs_length):
            dedoh = dEdActiv * w2[o][o2]
            for i in range(0, inputs_length):
                dedohw[i][o] += dedoh

            dedohb[o] += dedoh

    for o in range(0, outputs_length):
        dActiv = dActivationFunction(acti_arg1[o])
        for i in range(0, inputs_length):
            dedw[i][o] = dedohw[i][o] * dActiv * inputs[i]

        dedb[o] = dedohb[o] * dActiv * 1

    return dedw, dedb
```

A.3. Back propagation chain

```
def learn(self, x, y_desired):
    y = self.think(x)
    dedo = self.dSqe(y, y_desired)

    (dedw2, dedb2) = self.gradientsOutput(self.l1_output, dedo, self.w2,
        self.dSigmoid, self.output)
    (dedw1, dedb1) = self.gradients(x, dedo, self.w1, self.w2,
        self.dReLU, self.l1_z, self.dSigmoid, self.output)

    self.update(self.w2, self.b2, dedw2, dedb2)
    self.update(self.w1, self.b1, dedw1, dedb1)
```

A.4. Update function

```
def update(self, w, b, dedw, dedb):
    outputs_length = len(w[0])
    inputs_length = len(w)

    for o in range(0, outputs_length):
        for i in range(0, inputs_length):
            w[i][o] += dedw[i][o] * self.lr

        b[o] += dedb[o] * self.lr
```

Bibliography

[1] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010.

[2] Yann Lecun, Leon Bottou, Y. Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86:2278 – 2324, 12 1998.

[3] Daniel Keysers, Thomas Deselaers, Christian Gollan, and Hermann Ney. Deformation models for image recognition. *IEEE transactions on pattern analysis and machine intelligence*, 29:1422–35, 09 2007.

[4] Patrice Simard, David Steinkraus, and John Platt. Best practices for convolutional neural networks applied to visual document analysis. pages 958–962, 01 2003.

[5] Dan Claudiu Cireşan, Ueli Meier, Luca Maria Gambardella, and Jürgen Schmidhuber. Deep, big, simple neural nets for handwritten digit recognition. *Neural Computation*, 22(12):3207–3220, Dec 2010.

[6] N. Otsu. A threshold selection method from gray-level histograms. *IEEE Transactions on Systems, Man, and Cybernetics*, 9(1):62–66, 1979.

List of Figures

List of Tables