DIGITS 3-D

Practical Assignment

Lappeenranta–Lahti University of Technology LUT

Master degree in Computational Engineering

BM40A0702 – Pattern Recognition and Machine Learning

10th December 2021

GROUP 6

Student number:

000285359 – Thanh Ngoc Dan Tran (Model trainer and tester)

000275466 – Nghia Nguyen (Data processor)

Examiner(s): Prof. Lasse Lensu

Dr. Toni Kuronen

SYMBOLS AND ABBREVIATIONS

Abbreviations

MLP	Multi-layer Perceptron

Table of contents

Figures

Tables

1. Assignment Introduction

The fundamental idea behind this report is to reveal the process of constructing a learning system for air – written digits from 0 to 9 with Matlab. By using a LeapMotion sensor, 3-D location information (x, y, z) of the fingertip during generating trajectories has been recorded at each certain timestep. The training data was given for implementing a pattern recognition and machine learning model in order to perform the recognition of different digits with high accuracy.

In this project, it is decided to utilize the concept of multi-layer perceptron in implementing the recognition system due to the complexity of the task and the nature of the data. The model was constructed, and parameters were saved for later use without the need of training the model all again.

As an ultimate goal of this project, a recognition function **C = digit_classify(testdata)** was created in order to take one data sample at a time and predict its label. The parameters generated during the process of constructing model are utilized in this function in order to feed forward the given data and output network label. Before that can be done, however, input data needs to be pre-processed to ensure the compatibility with the model.

It should be noted that the project used no high-level functions in the implementation. All code files were programmed from scratch based on formulas.

2. Implementation procedure

Prior to successfully implement the function digit_classify, three main steps were proceeded, ten supporting functions or code files were needed, and several data files were configured. Due to not using high-level functions available in matlab, the implementation procedure was considerably longer than it should be. However, flexibility is the top benefit.

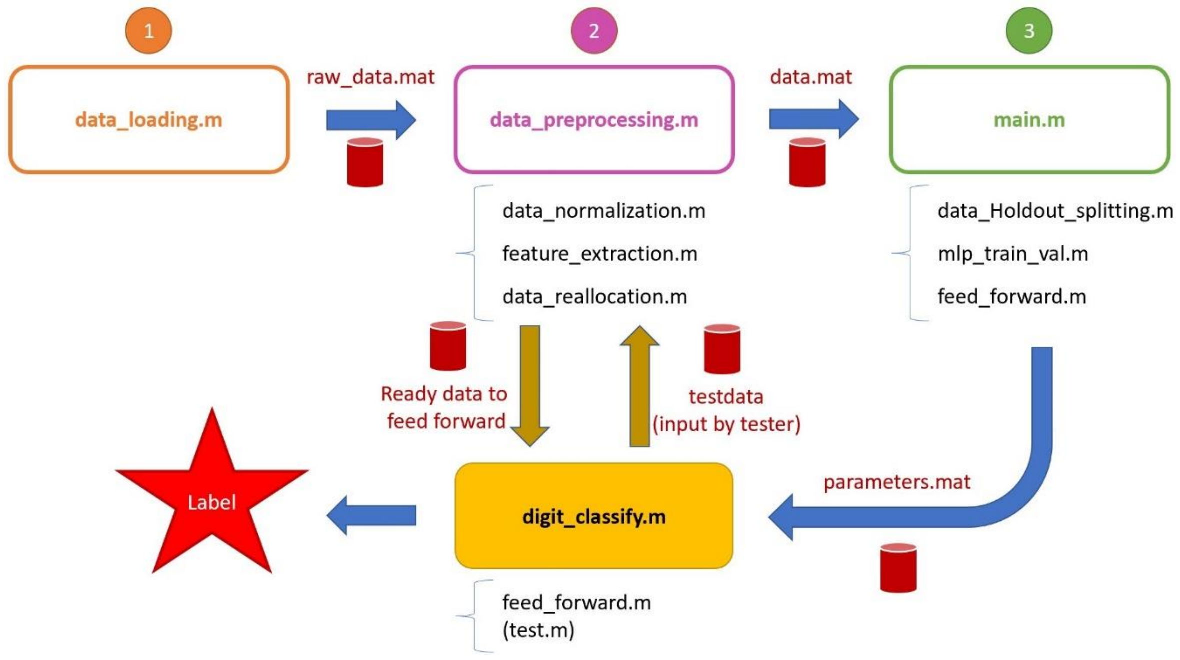


Figure 1. Implementation Procedure

As shown in the figure above, the three main steps were loading data, pre-processing data and main where the model was built and tested.

As each sample data was stored in separate data file, the data was firstly needed to be loaded and stored in a same data structure. All samples' 3-D time-series data was loaded and saved in a cell array. This step output a raw data file with a variable raw_data being a cell array of training data and class being a row vector of training data label.

Secondly, the raw loaded data were pre-processed due to a lack of compatibility with the MLP model. At first, each sample's raw data were on different scale depending on how big or small the stroke was written in front of the censor and that may cause some problem with the learning. By acknowledging that a number can still be correctly identify regardless of how big or small the stroke was, it is decided to normalize each sample's data using min-max scaling to bring all sample to roughly same size while still maintain the proportion of all side of the trajectory. Additionally, since every sample had different number of timestep data which was against the goal of consistency in data to feed MLP, the sample with the minimum number of timesteps (datapoints) was identified and the number of timesteps was chosen to be a standard for feature extraction. All samples were investigated and only standard number of timesteps was smartly extracted from the data. This sub-step was implemented carefully to ensure no loss in critical curve a number. After that, the cell array of extracted normalized data was transformed into a matrix with columns corresponding to samples and rows corresponding to features. This second step output a data matrix (data.mat) which was ready to be utilize for training MLP model.

The final step in preparing parameters for goal function digit_classify was to train an MLP model. The first sub-step was to split the data using holdout method in a way that 90 percent of the data set was used the train model and 10 rest percent of the data set was used to test the trained model parameters. The function data_Holdout_splitting.m aims at dividing the data set with equal class proportion, and thereby ensuring that no imbalanced data existed. After that, the train data and train class were input into function mlp_train_val.m to train the model. At this stage, the input train data was once again split into two sets, one (90%) for train model and one (10%) for validate the model to avoid overfitting. We used 100000 epoch at the maximum to train the model with 2 hidden layers and 16 neurons each. The reason behind this decision is explained in the section about constructing model. We next saved the trained model's parameters into Matlab data file named parameters.mat and utilized it to predict label for the test data set. The function feed_forward was implemented by first loading the parameters.mat data file and then performing feed forward operation and output the most possibly correct labels for the input data. We achieved the accuracy in the range of 92% and 97% with the best estimate at 95%.

As now the parameters were well-achieved, we could implement the recognitionfunction digit_classify which takes input as a matrix with N*3 size (N timesteps of 3-D location data), performs some pre-processing steps to get the test data ready for feeding into the trained model and finally assigns the test data a label with high accuracy. It may sound simple as that, but more detailed implementation can be found in the section for this function.

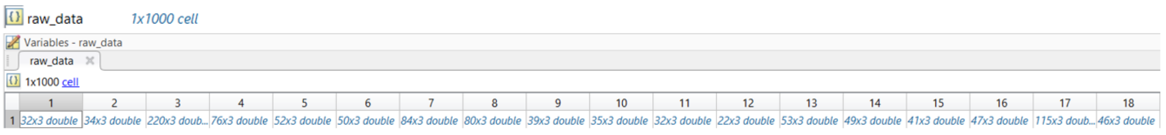
3. Raw data structure and Pre-processing

3.1 Raw data structure

First and foremost, the location information of each stroke of the experiment has been originally stored into separate Matlab files. Each file contains the three-dimensional location information of the trajectory. The class of the trajectory can be identified from the data file name. Due to the fact that we have 10 numbers ranging from 0 to 9 and each number has

100 different samples, the first step in creating the raw data structure was to load and store all these provided Matlab data files into one Matlab data file called *raw_data.mat* which contains one variable raw_data being a cell array of data samples and one variable named class being the class label of each of the data sample. The full code script can be found in appendix 1. The data pre-processing step and other steps afterwards can be easily built with the union of data for efficient data management.

The Matlab raw_data file is in a cell array consisting of 1x1000 cells, each cell represents one sample of a number.



raw_data 1x1000 cell

Variables - raw_data

raw_data

1x1000 cell

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1	32x3 double	34x3 double	220x3 doub...	76x3 double	52x3 double	50x3 double	84x3 double	80x3 double	39x3 double	35x3 double	32x3 double	22x3 double	53x3 double	49x3 double	41x3 double	47x3 double	115x3 doub...	46x3 double

Figure 2. raw_data.mat demo

3.2 Data Pre-processing

The critical idea behind the data pre-processing step is to normalize the raw data, extract features from the normalized data and finally reallocate the extracted data in a way that the output should be in a matrix of feature vectors. To be more precise, each coordinate (x,y,z) of each data point per sample is considered as a feature at this stage of data treatment. The detailed information of why and how the raw dataset needed to be normalized, the principles of methods used for the feature extraction and data reallocation can be found below. The full code script can be also found in appendix 2. There are 3 functions called in this stage.

3.2.1 Data Normalization

Data normalization plays a critical role in the recognition model to ensure that the quality of result is guaranteed and avoid biased data-handling that may happen. The trajectories are in different size because one number can be air-written big or small depending fully on the input author(s). The fact that the size of trajectory has no effect on the recognition of the that trajectory but the shape. Therefore, the target output of this data normalization step is that we bring all trajectories on the same scale and no shapes are changed. The code script for this function can be found in appendix 3.

In order to normalize the raw dataset, the min-max scaling method was chosen because it is very effective for the Multi-layer Perceptron model (MLP), where the backpropagation can be more stable and even consumes less computational resource when input features are min-max scaled compared to the original unscaled data (Serafeim, 2020).

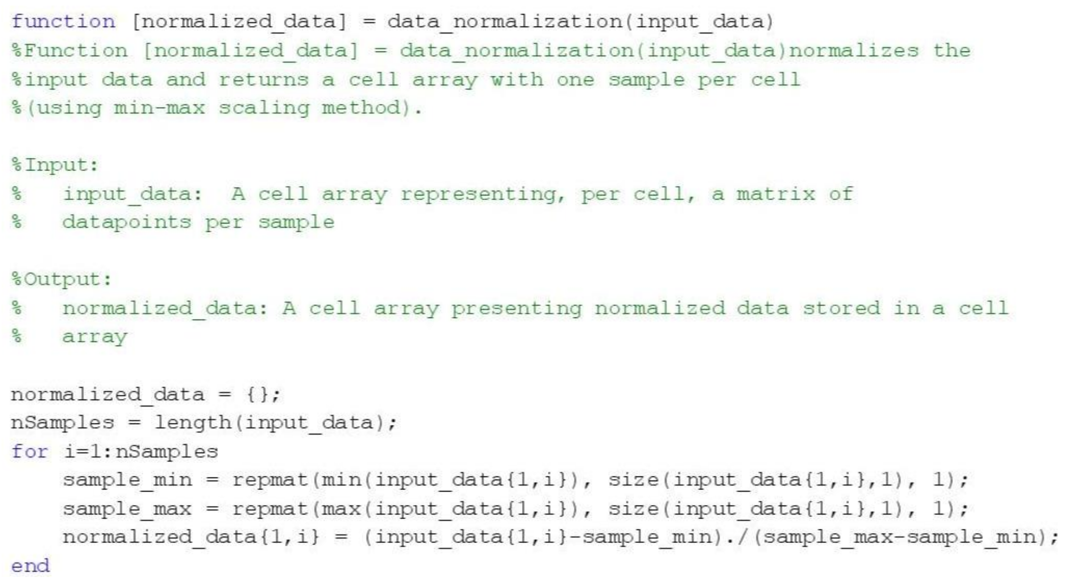
- Minmax-scaling

$$x_k^{\min} = \min_i x_{ik}, \quad x_k^{\max} = \max_i x_{ik}, \quad k = 1, 2, \ldots, l$$

$$\hat{x}_{ik} = \frac{x_{ik} - x_k^{\min}}{x_k^{\max} - x_k^{\min}} \qquad (1)$$

Figure 3. Min-max scaling formula

Our data normalization function was created based on the formula above. Specifically, in order to implement the formula, we need to identify 2 variables including x_k^{min} and x_k^{max}. Let’s take a look at Figure 4 below and break the code in detailed.



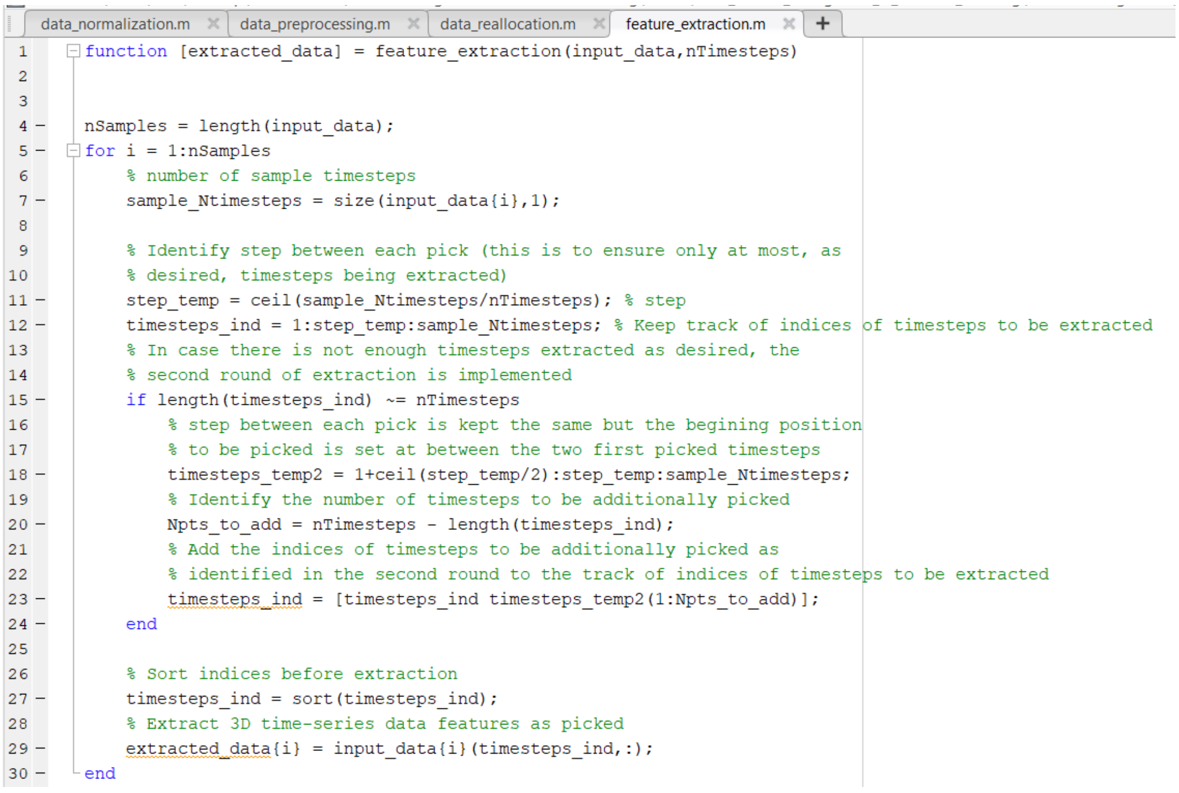
```
function [normalized_data] = data_normalization(input_data)
%Function [normalized_data] = data_normalization(input_data)normalizes the
%input data and returns a cell array with one sample per cell
%(using min-max scaling method).

%Input:
%   input_data:  A cell array representing, per cell, a matrix of
%   datapoints per sample

%Output:
%   normalized_data: A cell array presenting normalized data stored in a cell
%   array

normalized_data = {};
nSamples = length(input_data);
for i=1:nSamples
    sample_min = repmat(min(input_data{1,i}), size(input_data{1,i},1), 1);
    sample_max = repmat(max(input_data{1,i}), size(input_data{1,i},1), 1);
    normalized_data{1,i} = (input_data{1,i}-sample_min)./(sample_max-sample_min);
end
```

Figure 4. data_normalization function

According to the Figure 4, line 15 represents for x_k^{min}, line 16 represents for x_k^{max} and line 17 represents for putting these variables into the formula. The main idea behind using one single *min* or *max* functions in this case is to bring all samples onto the same scale without changing the shape of trajectory. Particularly, in line 15, the code part *"min(input_data{i}"* returns the output of a 1x3 vector containing minimum value of each of three dimensions. The same idea goes with the *max* function also. We utilized the *repmat* function to help with the subtraction later. The last step is putting sample_min and sample_max into the formula. In summary, via implementing formula above, a cell array containing normalized data is the desired output.

3.2.2 Feature Extraction

The fundamental idea behind this method is to extract a desired number of timesteps (also called trajectory points) per sample to ensure that all the output data sample has the same size. We decided to find the sample with the smallest number of three-dimensional datapoints and utilized that value as the standard number of three-dimensional features. Specifically, in this case, per sample, 19 trajectory points were chosen as a standard value. We chose to proceed with the minimum value because it is easier to extract from the available data than create more datapoints. Let's take a look at Figure 5 below to understand how the idea was applied in Matlab. The full code script can be found in appendix 4.



```
function [extracted_data] = feature_extraction(input_data,nTimesteps)


nSamples = length(input_data);
for i = 1:nSamples
    % number of sample timesteps
    sample_Ntimesteps = size(input_data{i},1);

    % Identify step between each pick (this is to ensure only at most, as
    % desired, timesteps being extracted)
    step_temp = ceil(sample_Ntimesteps/nTimesteps); % step
    timesteps_ind = 1:step_temp:sample_Ntimesteps; % Keep track of indices of timesteps to be extracted
    % In case there is not enough timesteps extracted as desired, the
    % second round of extraction is implemented
    if length(timesteps_ind) ~= nTimesteps
        % step between each pick is kept the same but the begining position
        % to be picked is set at between the two first picked timesteps
        timesteps_temp2 = 1+ceil(step_temp/2):step_temp:sample_Ntimesteps;
        % Identify the number of timesteps to be additionally picked
        Npts_to_add = nTimesteps - length(timesteps_ind);
        % Add the indices of timesteps to be additionally picked as
        % identified in the second round to the track of indices of timesteps to be extracted
        timesteps_ind = [timesteps_ind timesteps_temp2(1:Npts_to_add)];
    end

    % Sort indices before extraction
    timesteps_ind = sort(timesteps_ind);
    % Extract 3D time-series data features as picked
    extracted_data{i} = input_data{i}(timesteps_ind,:);
end
```

Figure 5. feature_extraction function

Due to the fact that the output we obtained after taking the length of each sample divided by the desired number of timesteps to be extracted is an integer value, therefore, the *ceil* function was chosen to round it up. Particularly, if we used the *round* function to round each element of the matrix to the nearest integer, all timesteps in one sample would not covered completely and the shape can be omitted because the *round* function only rounds the value of an element to the integer with larger magnitude when that element has a fractional part over 0.5. As a result, the *ceil* function is considered as the most optimal function in this case to calculate the step between every two to-be-extracted trajectory points. The ceil function ensures that there are at most 19 trajectories chosen and all chosen points covers the original shape from the start to the end.

The extraction may be proceeded in more than one round if there is not enough 19 trajectory points chosen. In case there is not enough 19 trajectory points chosen in the first round, the optimal solution we came up with is to select some more points until 19 points are accomplished. To do that, we keep the step size but move the second round's starting point to the median point between the first round's starting point and its next chosen point. To illustrate what I mean, let's take a look at Figure 6 below.

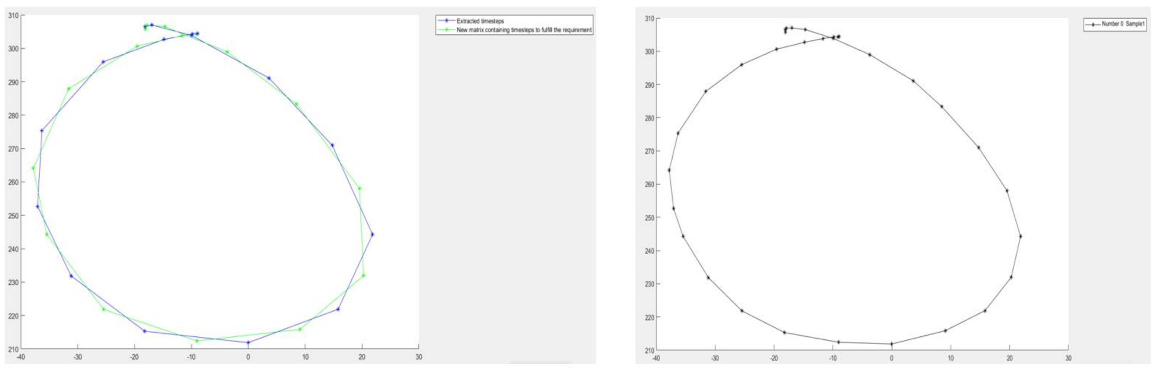
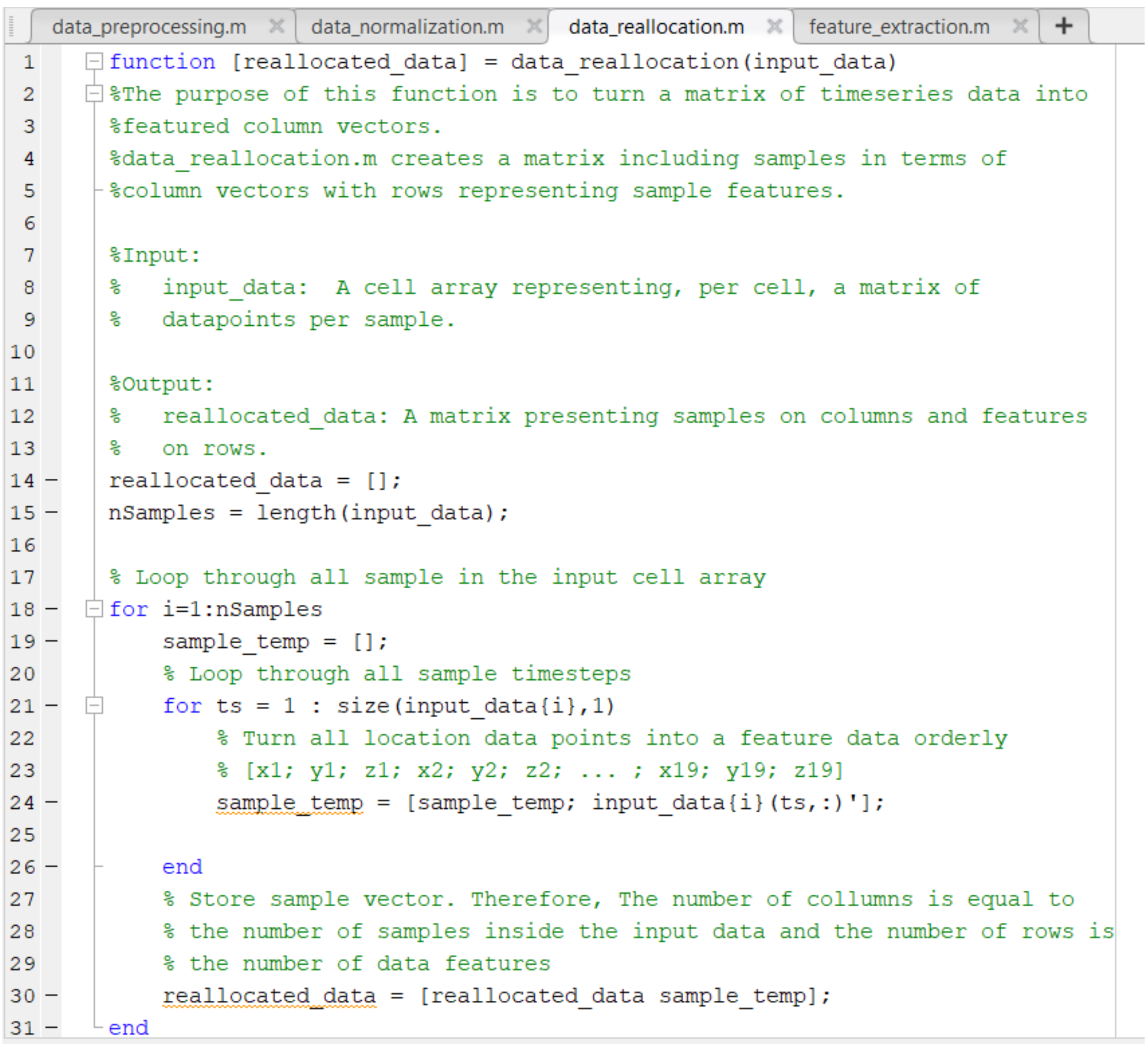


Figure 6. feature_extraction function explanation

In accordance to Figure 6 above, the picture on the right side is the original plot of the first sample of number 0 with the total number of 32 trajectory points. Due to the fact that the calculated step for this sample is 2, as a result, 16 trajectory points can be picked from this calculation. However, this output did not meet our expectation, therefore, we decided to create another new matrix so that we can pick 3 more trajectory points to fulfil our need. In particular, the plot on the left side of Figure 6 is a combination of 2 plots which the blue plot represents for matrix containing 16 extracted trajectory points based on calculated steps and the green plot represents for a new matrix containing timesteps which can be picked to fulfil the requirement of this function. In this example, 3 first points plotted in green can be picked. Lastly, after getting enough 19 timesteps, the *sort* function from line 27 (Figure 5) will guarantee the order of trajectory points is always in the correct position.

3.2.3 Data Reallocation

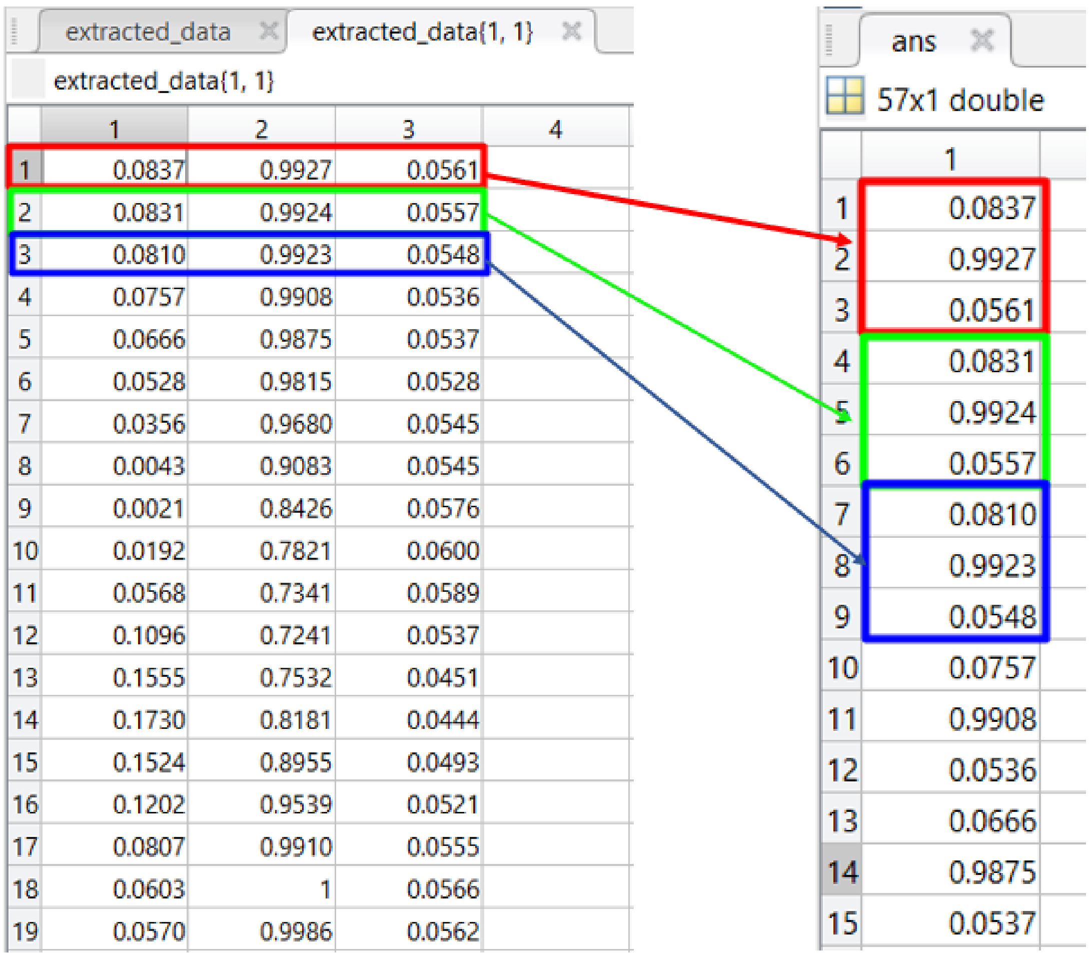
After the first 2 steps including data normalization and feature extract, the final step in data pre-processing is data reallocation. To be more precise, the purpose of this step is to create a function what help user turn a matrix of timeseries data into one single feature column vector. Specifically, an output matrix including samples in terms of column vectors with rows representing features can certainly be achieved after inputting a cell array of matrices of trajectory points per sample in each cell. At this stage, all the input data matrices must have the same size. Let's take a look at Figure 7 below to understand how the idea was applied in Matlab. The code script can also be found in appendix 5.



```
1     function [reallocated_data] = data_reallocation(input_data)
2     %The purpose of this function is to turn a matrix of timeseries data into
3     %featured column vectors.
4     %data_reallocation.m creates a matrix including samples in terms of
5     %column vectors with rows representing sample features.
6
7     %Input:
8     %   input_data:  A cell array representing, per cell, a matrix of
9     %   datapoints per sample.
10
11    %Output:
12    %   reallocated_data: A matrix presenting samples on columns and features
13    %   on rows.
14 -  reallocated_data = [];
15 -  nSamples = length(input_data);
16
17    % Loop through all sample in the input cell array
18 -  for i=1:nSamples
19 -      sample_temp = [];
20        % Loop through all sample timesteps
21 -      for ts = 1 : size(input_data{i},1)
22            % Turn all location data points into a feature data orderly
23            % [x1; y1; z1; x2; y2; z2; ... ; x19; y19; z19]
24 -          sample_temp = [sample_temp; input_data{i}(ts,:)'];
25
26 -      end
27        % Store sample vector. Therefore, The number of collumns is equal to
28        % the number of samples inside the input data and the number of rows is
29        % the number of data features
30 -      reallocated_data = [reallocated_data sample_temp];
31 -  end
```

Figure 7. data_reallocation function

In order to convert all rows of one sample into a single column, we need to use for loop function 2 times which the first loop runs through each sample in the data set and the second loop runs through each timestep of a sample so that total of 1000 samples and their features can be reallocated. Let's assume that the input data in this data reallocation function is the extracted data we got after running the second function (Figure 5). When the first loop starts running with i is equal to 1, the second loop will start running all timesteps inside the first sample in the extracted data cell array with ts ranging from 1 to 19. After that, inside the second loop function, line 24 in Figure 7 will start converting each row into a single column which means, after all, from a matrix 19x3 will be converted to a column vector having 57 elements (because 19*3=57). In simpler terms, let's look at the example (Figure 8) below.



extracted_data{1, 1}

	1	2	3	4
1	0.0837	0.9927	0.0561	
2	0.0831	0.9924	0.0557	
3	0.0810	0.9923	0.0548	
4	0.0757	0.9908	0.0536	
5	0.0666	0.9875	0.0537	
6	0.0528	0.9815	0.0528	
7	0.0356	0.9680	0.0545	
8	0.0043	0.9083	0.0545	
9	0.0021	0.8426	0.0576	
10	0.0192	0.7821	0.0600	
11	0.0568	0.7341	0.0589	
12	0.1096	0.7241	0.0537	
13	0.1555	0.7532	0.0451	
14	0.1730	0.8181	0.0444	
15	0.1524	0.8955	0.0493	
16	0.1202	0.9539	0.0521	
17	0.0807	0.9910	0.0555	
18	0.0603	1	0.0566	
19	0.0570	0.9986	0.0562	

ans — 57x1 double

	1
1	0.0837
2	0.9927
3	0.0561
4	0.0831
5	0.9924
6	0.0557
7	0.0810
8	0.9923
9	0.0548
10	0.0757
11	0.9908
12	0.0536
13	0.0666
14	0.9875
15	0.0537

Figure 8. Example of data reallocation

In an example of the first sample (as shown in Figure 8), for the first timestep (framed in red rectangle) , we have a matrix of 1x3 including 0.0837, 0.9927 and 0.0561 respectively. After running the code from line 24 (Figure 7), the matrix 1x3 will be converted into a column vector 3x1. The process continues running until the final row of the first sample. The same operation is done until the final sample is converted into a column vector containing 57 elements. The outer for loop is responsible for sticking all these column vectors together after every one of them is fully converted. The output of this function is certainly a matrix of data with 1000 columns and 57 rows.

4. MLP Recognition Model (*main.m*)

This stage aims at randomly splitting data for training model and test it. The code file for this stage is *main.m* which can be found in appendix 6.

The target output of this stage is the achievement of the trained MLP model parameters which are mainly responsible for performing prediction of the future inputs. The parameters were tested for accuracy acknowledgement. The test results are discussed in the section *Results*.

There were three functions called in this phase:

- data_Holdout_splitting.m (Appendix 7)
- mlp_train_val.m (Appendix 8)
- feed_forward.m (Appendix 9)

4.1 Data splitting

The first step was to split the available 1000 data samples which was pre-processed and turned into a matrix with the size of 1000 columns and 57 rows into two sets of data. One of them which accounted for 90% of the data set was input into the training process and the other one accounting for 10% of the data set was utilize for testing purpose after the availability of the trained model's parameters. The data was randomly split while still ensure equal class proportion inside each set. The function *data_Holdout_splitting.m* was programmed in respect of that purpose.

Additionally, the train set, after entered the model training phase, it was then split once more time into two sets, one accounting for 90% was utilized for literally training the model and the other one set performed as a validation set. The main responsibility of the validation set was to acknowledge overfitting in case it happened. The splitting of it into two sets for train and validation purposes was done inside the function *mlp_train_val.m*.

After the parameters achieved, the testing phase started by input the test set and the parameters into the function *feed_forward.m*. The result about model accuracy can be determined in the matter of seconds and it can be seen on the screen.

4.2 Training MLP model

This phase was performed by calling the *function mlp_train_val.m.* The input value was the training data set which included 90% of all 1000 data samples which was achieved after the first splitting round and the target outputs were parameters of the trained model.

This function prints some information about the network layout on the screen, does the learning process, visualizes the loss function values over iterations and output the trained model parameters with overfitting avoided.

4.2.1 Network Architecture

The network starts at the input layer currently having 57 neurons (19 three-dimensional location data * 3). The number of neurons is equal to the number of features in the training data.

The number of neurons on the output layer was simply decided based to the number of classes which ranges from 0 to 9. Therefore, we have 10 neurons on the output layer.

In terms of deciding on the architecture of the hidden layer(s), many adjustments of network architecture had been made from very complex to rather simple before the chosen option was considered persuasive and the final decision was made. By considering the complexity of the digit recognition written in the air, we believe that more than three hidden layers and/or more than 20 neurons would be an overkill. By having experimented several options, the performance of each option was manually collected and analyzed as shown in the below table. It should be noted that the program was run on a mobile workstation which has 8 cores and 16 threads providing a computing power thanks to the 32GB of RAM. Other configurations can also affect the results.

Table 1. Network architecture adjustment experimenting results

Experiment	**1**	**2**	**3**	**4**	**5**
nHidden	[8,8]	[16,16]	[16,16,16]	[32]	[32,32]

Max epochs	100,000	100,000	100,000	100,000	100,000
Overfitting rate	20%	70%	80%	50%	80%
Accuracy (Test set)	81% - 89%	92% - 97%	94% - 97%	87% - 93%	92% - 98%
Training time (s)	~ 102	~ 135	~ 273	~ 117	~ 264

Each option was implemented 10 times before the final decision was made. It is obvious that the option number 5 with two hidden layers having 32 neurons each brought the best result in term of classification result. However, the training time was long, the rate of overfitting occurrence was high, and it was considered to be an overkill for such task as this. It is undeniable that the options 1 with two hidden layers having 8 neurons each and 4 with one hidden layer having 32 neurons underperformed. That results suggested that a slightly more complex or deep network would work better. We came to the last two options 2 and 3. Since the option 3 which brought 3 hidden layers and 16 neurons each did not result in dominated performance while its training time was more than double the option 2's and the rate of overfitting was higher than that of option 2, the option 2 was chosen to be our final network architecture.

To summarize, our final network ended up with 1 input layer with as many neurons as the number of features in our data (57 neurons), 2 hidden layers which have 16 neurons each and one output layer with as many neurons as the number of unique classes (10 neurons). The layout of the trained network is demonstrated in the below figure.

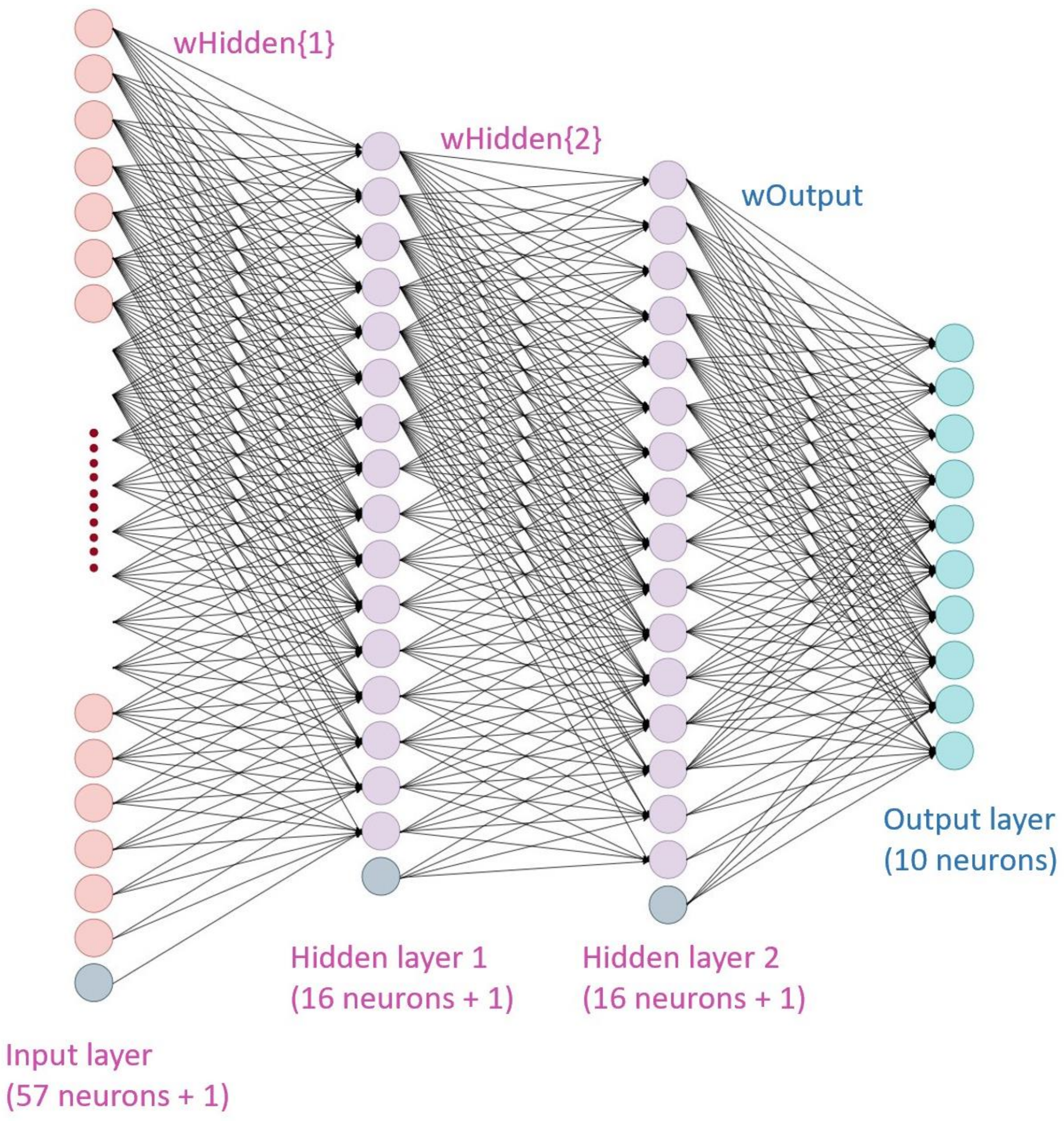


Figure 9. Network Architecture

We considered biases as weights of input value 1. Therefore, the first hidden layer has 928 parameters (58 input feature vectors multiply 16 neurons), the second hidden layer has 272 parameters (17 input vectors output from the previous layer multiply 16 neurons) and the output layer has 170 parameters (17 input vectors output from the last hidden layer multiple 10 neurons). After training, the model is expected to generate **totally 1370 parameters** including all weights and biases as the target outputs.

4.2.2 Model validation, Maximum epochs and Overfitting

As overfitting is one of the common problems with neural network, validation was implemented during the learning process to avoid it. Our model was built to not only train a promising model, but also be able to identify overfitting and track back to parameters.

First approach is that the training data was randomly splitting using holdout method into two sets of data with equal class proportion as the split proportion, one set accounting for 90% of the input data was used for training the model and the other set accounting for 10% of the input data was used for the validation purpose. In fact, the loss function of training data always produces a downward trend as the learning process proceeds. That is just the primary idea of backpropagation. However, as the learning process advances, overfitting can be identified at the time when the loss function error of validation starts to go back upward. Therefore, the parameters at every 100 epochs were recorded for use in case of having overfitting in the model.

Additionally, the maximum number of epochs has been set to be 100,000. At first, we had tried with 150,000 epochs, no better accuracy was achieved but overfitting occurred more often. Then we gradually reduced that number and ended up at 100,000 epochs.

When overfitting was identified, parameters were reselected as the ones updated right before the time of overfitting happens. Therefore, our code performs well in recognizing overfitting and tracking back to parameters. Our trained model whose parameters were stored in data file *parameters.mat* was the tenth run.

4.2.3 Learning process

The learning process followed exactly the idea of neural network concept. It started at randomly initializing weights for each hidden layer and output layer. It then performed feed forward operation. The Tanh activation function was utilized for hidden layers and there was no activation function for the output layer. After that, back propagation was performed to update all the weights based on the base concept of gradient descent and chain rule. The operation of feed forward and back propagation were iterated for 100,000 times. The target output of the process is the parameters of the network model. At the end of 100,000th iteration, there is a condition to check overfitting and reselect parameters which was updated right before overfitting occurs. The code script for the learning process can be found in appendix 8 about the function *mlp_train_val.m*

The function takes inputs as train data, its classes, a vector verifying the network architecture and the maximum number of epochs. Before the learning process starts, the function firstly acknowledges the size of the input train data, performs data splitting into two sets (train set and validation set) as described earlier, prints some prompts to announce about the network architecture, transforms the class vector into a matrix format, initializes some variable to keep track of the iterations, parameters and the loss function value, extends the input data by adding one row vector of N (the number of samples) number of value 1 at the end of the train set and initializes the learning rate. We set 0.0001 as our model's learning rate.

The next step is to initialize weights. In order to do that, we needed to acknowledge how many weights needed for each layer. We decided to store weights of hidden layers in a cell array. Since there was 2 hidden layers in our network, there were 2 cells in the variable wHidden. The weights of the first hidden layer were stored as a matrix of 58 rows and 16 columns. The weights of the second hidden layer were stored as a matrix of 17 rows and 16 columns. The variable wOutput is a matrix of 17 rows and 10 columns storing the weights of the output layer. The function randomly initializes the weights accordingly with *rand* Matlab function. The weights which are output from the rand function are then subtracted the by 0.5 and divided that by 10 to make the values small enough. Now, the learning process can be performed.

The learning process is iterated 100,000 times as defined previous. The ultimate purpose of the learning process is to find the connections of the features and the classes. Firstly, the feed-forward operation is done for layer by layer (except the input layer) with the current parameters. There are two small steps in this operation. The first one is to calculate the net activation for each neuron in a layer based on the equation (2).

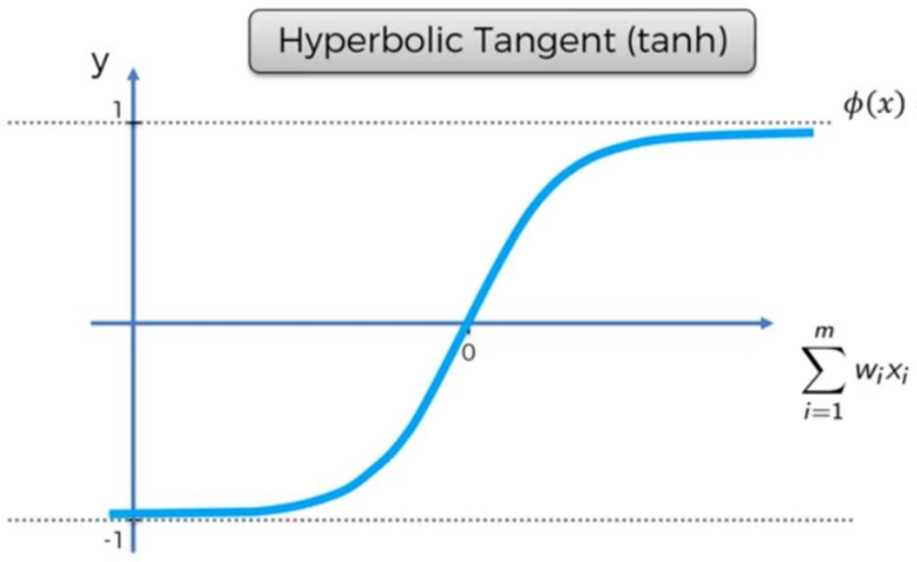
- Using extended input vectors ($x_0 = 1$), net activation of a neuron j is

$$v_j = \sum_{i=1}^{l} w_{ji} x_i + w_{j0} = \sum_{i=0}^{l} w_{ji} x_i = \mathbf{w}_j^{\mathsf{T}} \mathbf{x} \qquad (2)$$

where i is the index of the feature vector elements and w_{ji} is the weight from input i to neuron j.

Figure 10. Feed-forward equation (net activation)

The second one is to determine the output from the first step as roughly “yes” or “no” using activation function. We utilize hyperbolic tangent (known as tanh) non-linear activation in our model for hidden layers as shown in the below figure. It should be noted that we do not use activation for the output layer.



$$\phi(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}} \qquad (3)$$

Figure 11. Hyperbolic Tangent (tanh)

In the function *mlp_train_val.m* (appendix 8), *vHidden* and *vOutput* is the result of the feed-forward equation for hidden layers and output layer respectively. More specifically, the *vHidden* variable is a cell array containing two matrices representing two hidden layers net activations. The first hidden layer’s net activation is the result of dot product of the weights of this layer and the extended input. This simply means that we feed the input into the first hidden layer. After that, we use *tanh* Matlab function to determine the output of the net activation and store the value to the cell array *yHidden*. The output is extended following the same idea of the extended input. The second hidden layer’s net activation was calculated following the same idea. To be specific, the net activation of the second hidden layer is the result of the dot product of the weights of this layer and the output from the previous layer and then, the net activation is computed using the activation function. Similarly, the net activation of the output layer can be calculated by performing the dot product of the weights of the output layer and the output from the last hidden layer. There is no activation function applied in the output layer. The output layer determines the prediction of the class of the input data. The higher the output of a neuron is, the higher the possibility that the class of the input data is the index of that neuron minus by 1 (because the index starts at 1 instead of 0).

After the feed-forward operation is done for all layers, the loss function value is calculated to determine the train data set's error generated by the current parameters and stored into vector *Jtrain*. In this step, since we decided to validate the model simultaneously, we also calculate the error that the model generates on the validation data set and store it into vector *Jval*. The equation for loss function is shown below.

$$J(\boldsymbol{w}) = \frac{1}{2}\sum_{i=1}^{N} \|\boldsymbol{y}(i) - \hat{\boldsymbol{y}}(i)\|^2 \qquad (4)$$

Figure 12. Sum of squared error equation (Loss function)

That is the function for sum of squared error (SSE). According to the equation, error can be calculated by dividing the sum of the squared of the subtraction of the current model output and the desired output (our transform class matrix) by 2. There is a separate subfunction named *cal_cost* programmed with in the *mlp_train_val.m* file (appendix 8) to calculate the loss function value of the validation set by performing the feed-forward operation on the input data set and parameters.

Visualization can be generated for these two variables every 1000 epochs. The visualization illustrates the improvement of the model outputs on training and validation data. The main idea of this learning process is to minimise the loss function. Therefore, the Jtrain is checked in every epoch. We use threshold at 1e-10 to check the error of output on training data. If the loss function value is less than the threshold or the improvement of error in this epoch compared to that in the previous epoch is less than the threshold, the learning process can be finished. Otherwise, the learning process can be stopped once the maximum number of epochs is reached.

The parameters at every 100 epochs are recorded in variable wHidden_track and wOutput_track. The record provides the possibility to track back parameters when overfitting is identified during the learning process. The condition to acknowledge overfitting is discussed later in this section.

The next step is to perform backpropagation. Backpropagation for MLP follows the idea of gradient descent optimization which identifies the direction forwards the minimum point of the loss function. The equations to proceed backpropagation can be found in the below figure.

- Gradient descent:

$$\boldsymbol{w}_j^r(t+1) = \boldsymbol{w}_j^r(t) + \Delta\boldsymbol{w}_j^r, \quad \Delta\boldsymbol{w}_j^r = -\rho\frac{\partial J}{\partial \boldsymbol{w}_j^r} \tag{5}$$

- Training rule for SSE, sensitivities δ and learning rate ρ:

$$\Delta\boldsymbol{w}_j^r = -\,\rho\sum_{i=1}^{N}\delta_j^r(i)\boldsymbol{y}^{r-1}(i) \tag{6}$$

$$\delta_j^L(i) = (\hat{y}_j(i) - y_j(i))f'(v_j^L(i)) \tag{7}$$

$$\delta_j^{r-1}(i) = \left(\sum_{k=1}^{k_r}\delta_k^r(i)w_{kj}^r\right)f'(v_j^{r-1}(i)) \tag{8}$$

Figure 13. Backpropagation equations

As how it is called, in the process of backpropagation, we move backwards from the output layer back to the input layer in order to update the parameters. Firstly, we calculate the *deltaOutput* following to the equation (7) by subtracting the model output by the desired output. Because no activation function was used for the output layer, we do not multiply the result by the derivative of the activation function. Secondly, we calculate the delta weight output (*deltawOutput*) following to the equation (6) by performing the dot product of the calculated delta of the output layer and the output of the last hidden layer and then multiplying it by negative of the learning rate. Lastly, we update the parameters of the output layer by summing the current parameters *wOutput* and the *deltawOutput* up. The similar operations are done with all hidden layers' parameters. Two variables *deltaHidden* and *deltawHidden* are cell arrays storing value of two hidden layers. For the last hidden layer, we calculate the deltaHidden by following the equation (8), so that we operate the dot product of the weights of the next layer (which is the output layer) and the delta of the next layer (which is the output layer) and then, multiply that by the derivative of the activation function (tanh) (equation (9)).

$$\frac{d}{dx}\tanh(x) = \frac{(e^x + e^{-x})(e^x + e^{-x}) - (e^x - e^{-x})(e^x - e^{-x})}{(e^x + e^{-x})^2}$$
$$= 1 - \frac{(e^x - e^{-x})^2}{(e^x + e^{-x})^2} = 1 - \tanh^2(x) \qquad (9)$$

Figure 14. Derivative of tanh function

deltawHidden of this layer can be calculated by following the equation (6), so that it is calculated by multiplying the negative learning rate by the dot product of the output of the previous layer by the delta of this layer. The new weights of this layer can be updated by summing the current weights of this layer and *deltawHidden* of this layer up (equation (5)). To other hidden layers, the same operations can be done.

That is all about the learning process in one epoch. The error decreases after every epoch and the parameters are updated. After 100,000 epochs are completed, it is time to check whether overfitting occurs in the learning process. The overfitting can simply be identified if the loss function value starts to increase. Therefore, we identify it by finding the minimum value of the *Jval* and if it is not result of the last epoch, then it is highly acknowledged as the value calculated right before overfitting occurs. In that scenario, our model parameters are not the one updated lastly but the one updated at that very moment. Therefore, we can track back the parameters by indexing from the two variable *wHidden_track* and *wOutput_track*. Otherwise, if overfitting does not occur, the parameters of our model are the one lastly updated.

4.2.4 Results

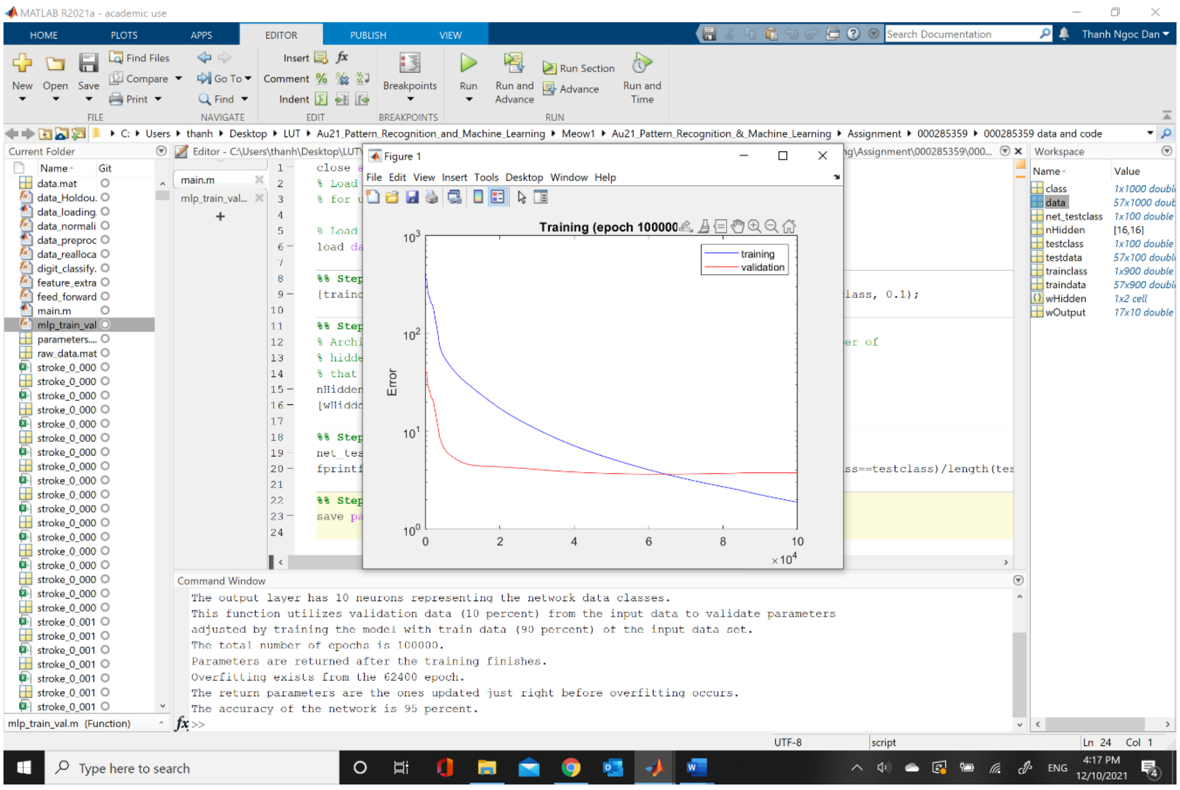
The code file *main.m* was run 10 times to acknowledge how the results distributes and the final parameters stored was the one generated on the 10th run.

As fore-mentioned about how the training and testing were designed to proceed, the test runs were done accordingly and the results from testing the trained model with test set (10% of 1000 training data samples) ranged from 92% to 97%. It should be noted that there was condition setting aiming at acknowledging and avoiding overfitting if occurred. Therefore, parameters updated after overfitting occurred (in case it occurred) were not selected for output. The table below reveals records of 10 trained models about overfitting and accuracy.

Table 2. Training results

Run	1	2	3	4	5	6	7	8	9	10
Overfitting at … epoch	19,800	NO	24,900	92,200	75,500	NO	91,000	85,100	NO	62,400
Accuracy	94%	97%	93%	95%	95%	93%	95%	94%	92%	95%

On the last run, we were able to capture the result as shown in the figure below. The parameters generated in this run were stored in data file parameters.mat as representative of our trained model and they were used for future prediction purpose.



```
The output layer has 10 neurons representing the network data classes.
This function utilizes validation data (10 percent) from the input data to validate parameters
adjusted by training the model with train data (90 percent) of the input data set.
The total number of epochs is 100000.
Parameters are returned after the training finishes.
Overfitting exists from the 62400 epoch.
The return parameters are the ones updated just right before overfitting occurs.
The accuracy of the network is 95 percent.
>>
```

Figure 15. Trained model results

There was a prompt about the overfitting and when it happened during the 100,000 epochs. In the tenth run, overfitting occurred after epoch number 62,400. The recorded parameters were the ones updated right before it happened.

The graph visualized the loss function value of training data and validation data over epochs. Even though it illustrated a very promising improvement rate in the training data, we cannot capture the parameters at the end of the learning process but the ones before overfitting happened. It is not clear that from the visualization where the overfitting occurred, but according to the prompt, it occurred after the epoch 62,400.

The accuracy rate of 95% was achieved in this run. This means that with the test set of 100 data samples which was not included in the training or validation, there were 95 samples labelled correctly by our trained model. Such high accuracy was not rarely achieved during the ten runs. Due to the fact that the data samples were selected and split completely randomly every run, no reconsiderations or confusions should be concerned.

5. Classification function C = digit_classify(testdata)

As required in the training phase, the test data sample passed to this function must be firstly pre-processed as well to ensure the consistency and compatibility with the trained model. To be more specific, the input data must be normalized, its features must be extracted if needed and then reallocated before feeding our trained model. There is one special case which is discussed thoroughly as well.

The target output of this function is one single scalar which is a prediction about the label of the test sample. More details concerning the implementation of this classifier function can be consulted from Appendix 10.

5.1 Data Preprocessing

As the input type of all the data pre-processing functions is a cell array, the matrix (N*3) of the test data sample needs to be turned into a cell array with a cell in advance. Then, it is ready to enter to pre-processing stage.

5.1.1 Data Normalization

The first step is to normalize the input data sample. The aim is to scale the input data sample to the same scale of all data in the training phase. This step can be done with simply one line of code. To have more details about how the normalization goes, moving back to the section *3.2.1 Data Normalization* is recommended. This step ensures no change in the shape of the test data sample, but the size is scaled to the standard size of 0 and 1.

5.1.2 Feature Extraction

This step is the most important step to ensure that the new input data is compatible with the trained or to-be-trained models. For more information about how the features are extracted, section *3.2.2 Feature Extraction* can be consulted.

There are three different scenarios being able to happen. As in the training phase, we proceeded by extracting 19 three-dimensional datapoints in each sample and the model was trained using the extracted data. Therefore, ensuring that the test data sample has same number of features is crucial to proceed further.

The first scenario is that the test sample is an exactly-19-point trajectory. In this case, this step can be completely skipped.

The second scenario is that the test sample has more than 19 three-dimensional datapoints. Then, we can just simply call function feature_extraction to help extract exactly 19 points from the data and it is ready to move on the pre-processing step.

The third scenario is called the **special case** when there is less than 19 datapoints in the test data. This case requires some extensive configurations in training data and our model needs to be retrained as well. The new standard number of datapoints per sample is the number of datapoints that the test data sample has. Therefore, extraction of feature for the test data sample is not necessary. The re-pre-processing for training data and retraining model procedure can be found in the section *5.3 Special case: Model re-training required.*

5.1.3 Data reallocation

After the feature extraction step is done, the last pre-processing step is to reallocate the data which is currently in form of a cell array by turning it to a single column vector. Section *3.2.3 Data Reallocation* reveals how the process can be easily done.

5.2 Normal case: Feed forward operation with trained parameters

In the normal scenario where the number of datapoints of the test trajectory is equal or more than the standard number of datapoints of data used in training model, the trained model’s parameters can be loaded from the data file *parameters.mat* and the pre-processed test data sample can be labelled in the matter millisecond by calling *feed-forward* function (appendix 9).

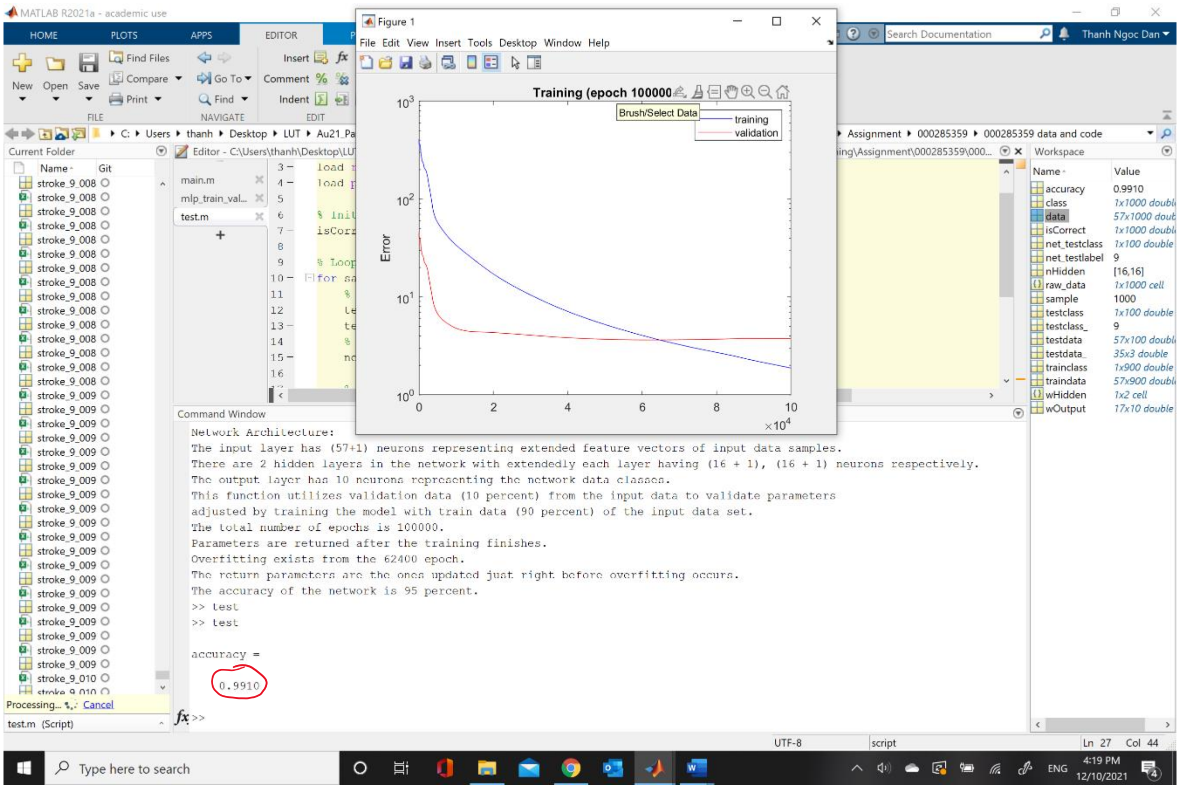
5.3 Special case: Model re-training required

As mentioned, if the test data sample has less than 19 three-dimensional datapoints, it is considered as a special case. In this case, the data file *raw_data.mat* containing all 1000 training data sample in the raw format is loaded and pre-processed using the standard number of datapoints as the number of datapoints of the test sample instead of 19. Since change in the training data leads to change in trained model, the MLP model must be retrained using the newly pre-processed training data. Feed forward operation can, at this stage, be done with the newly trained model to make prediction.

6. Comprehensive testing and results

In order to test the performance of the trained model and the classification function digit_classify, all 1000 available data samples were utilized. The code script for this phase can be found in appendix 11. Specifically, in this phase, we tested each raw data sample (in N*3 matrix format) at a time and got the prediction as a single label ranging from 0 to 9 with function digit_classify. Then the predicted label was compared with the real class of the input sample.

The below figure reveals a near perfect accuracy of the model at 99.1% in the digit recognition task. Since we had been able to capture the parameters which brought 95% accuracy (mention in section 4.2.4), this result was expectable.



Name	Value
accuracy	0.9910
class	1x1000 doubl
data	57x1000 doub
isCorrect	1x1000 doubl
net_testclass	1x100 double
net_testlabel	9
nHidden	[16,16]
raw_data	1x1000 cell
sample	1000
testclass	1x100 double
testclass_	9
testdata	57x100 doubl
testdata_	35x3 double
trainclass	1x900 double
traindata	57x900 doubl
wHidden	1x2 cell
wOutput	17x10 double

```
Network Architecture:
The input layer has (57+1) neurons representing extended feature vectors of input data samples.
There are 2 hidden layers in the network with extendedly each layer having (16 + 1), (16 + 1) neurons respectively.
The output layer has 10 neurons representing the network data classes.
This function utilizes validation data (10 percent) from the input data to validate parameters
adjusted by training the model with train data (90 percent) of the input data set.
The total number of epochs is 100000.
Parameters are returned after the training finishes.
Overfitting exists from the 62400 epoch.
The return parameters are the ones updated just right before overfitting occurs.
The accuracy of the network is 95 percent.
>> test
>> test

accuracy =

    0.9910

>>
```

Figure 16. Testing with each raw data sample and results

We were able to conclude that the stored parameters before overfitting, all supporting functions for data pre-processing and the function digit_classify were fully functional.

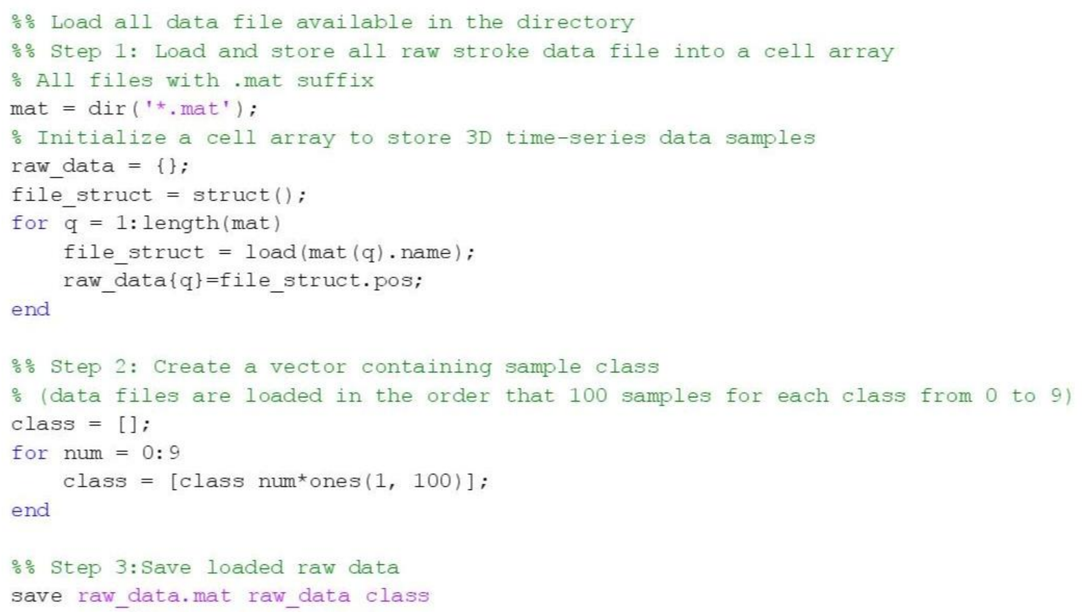
7. Conclusion

There is no exaggeration to state that our trained MLP model performed well with high prediction accuracy with the available 1000 samples. In fact, while only roughly 80% of the data set was utilized for training, a surprisingly near perfect result of 99.1% accuracy was achieved. As clearly fore-mentioned, we were well-aware of the possibility of overfitting and thereby, a solution for checking overfitting was implemented within the training model function. Our trained model identified no signs of overfitting during the learning process. Therefore, we strongly believed that our final model would produce promising results in the test phase for unknown data samples.

References

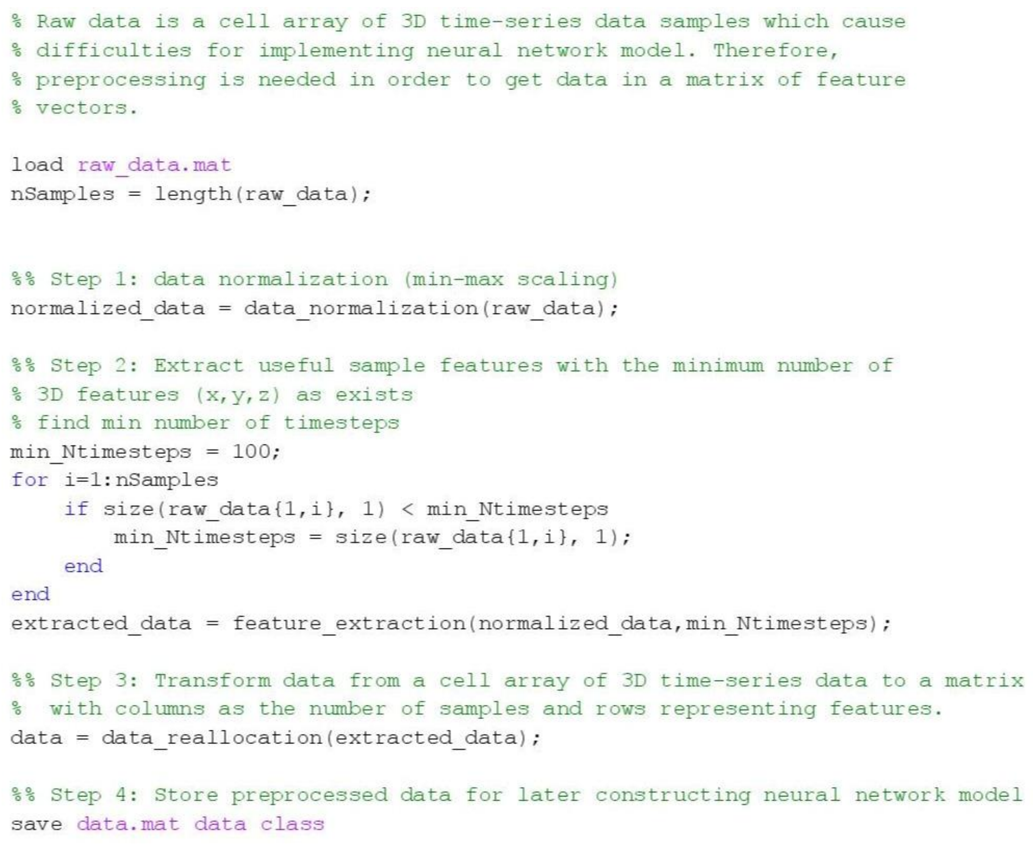
Serafeim, L. 2020. Everything You Need To Know About Min-Max Normalization: A Python tutorial. *Toward Data Science, 28 May 2020*. Accessed on 7 December 2021. Retrieved from https://towardsdatascience.com/everything-you-need-to-know-about-min-max-normalization-in-python-b79592732b79

Appendix 1. Code file data_loading.m



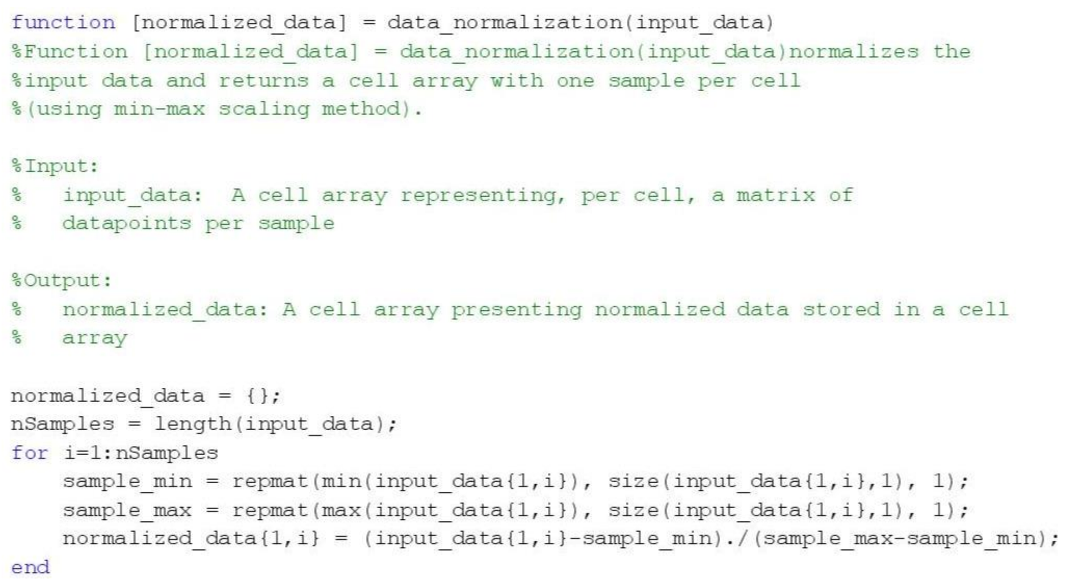
```
%% Load all data file available in the directory
%% Step 1: Load and store all raw stroke data file into a cell array
% All files with .mat suffix
mat = dir('*.mat');
% Initialize a cell array to store 3D time-series data samples
raw_data = {};
file_struct = struct();
for q = 1:length(mat)
    file_struct = load(mat(q).name);
    raw_data{q}=file_struct.pos;
end

%% Step 2: Create a vector containing sample class
% (data files are loaded in the order that 100 samples for each class from 0 to 9)
class = [];
for num = 0:9
    class = [class num*ones(1, 100)];
end

%% Step 3:Save loaded raw data
save raw_data.mat raw_data class
```

Appendix 2. Code file data_preprocessing.m



```
% Raw data is a cell array of 3D time-series data samples which cause
% difficulties for implementing neural network model. Therefore,
% preprocessing is needed in order to get data in a matrix of feature
% vectors.

load raw_data.mat
nSamples = length(raw_data);


%% Step 1: data normalization (min-max scaling)
normalized_data = data_normalization(raw_data);

%% Step 2: Extract useful sample features with the minimum number of
% 3D features (x,y,z) as exists
% find min number of timesteps
min_Ntimesteps = 100;
for i=1:nSamples
    if size(raw_data{1,i}, 1) < min_Ntimesteps
        min_Ntimesteps = size(raw_data{1,i}, 1);
    end
end
extracted_data = feature_extraction(normalized_data,min_Ntimesteps);

%% Step 3: Transform data from a cell array of 3D time-series data to a matrix
%  with columns as the number of samples and rows representing features.
data = data_reallocation(extracted_data);

%% Step 4: Store preprocessed data for later constructing neural network model
save data.mat data class
```

Appendix 3. Function data_normalization.m

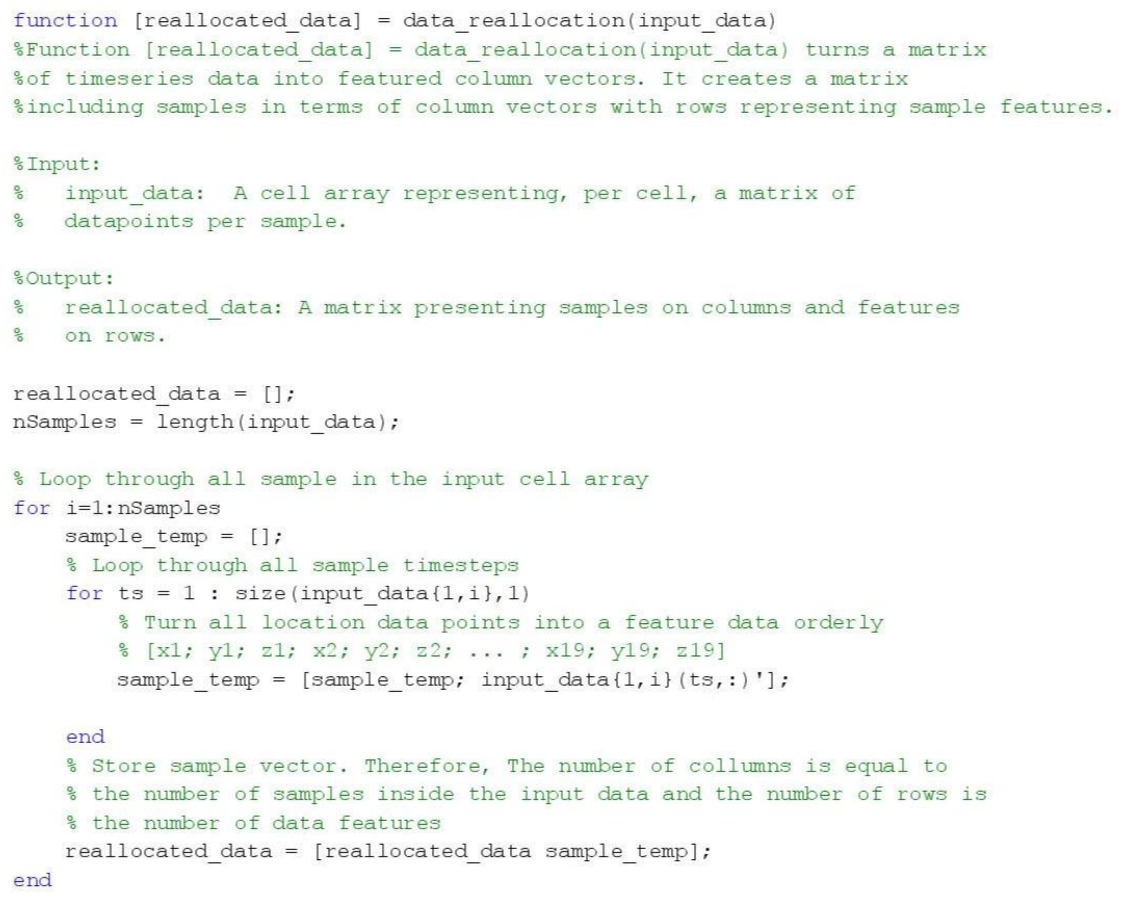


```
function [normalized_data] = data_normalization(input_data)
%Function [normalized_data] = data_normalization(input_data)normalizes the
%input data and returns a cell array with one sample per cell
%(using min-max scaling method).

%Input:
%   input_data:  A cell array representing, per cell, a matrix of
%   datapoints per sample

%Output:
%   normalized_data: A cell array presenting normalized data stored in a cell
%   array

normalized_data = {};
nSamples = length(input_data);
for i=1:nSamples
    sample_min = repmat(min(input_data{1,i}), size(input_data{1,i},1), 1);
    sample_max = repmat(max(input_data{1,i}), size(input_data{1,i},1), 1);
    normalized_data{1,i} = (input_data{1,i}-sample_min)./(sample_max-sample_min);
end
```

Appendix 4. Function feature_extraction.m

```
function [extracted_data] = feature_extraction(input_data,nTimesteps)
%Function [extracted_data] = feature_extraction(input_data,nTimesteps)
%extracts a desired number of data points (timesteps) per sample to ensure
%that the output data sample has same size.
%Note: Feature may be extracted several rounds until enough timesteps were
%picked as desired

%Inputs:
%   input_data:  A cell array representing, per cell, a matrix of
%   datapoints per sample.
%   nTimesteps: The desired number of data points to be extracted

%Output:
%   extracted_data: A cell array having data samples with the same size as
%   required

nSamples = length(input_data);
for i = 1:nSamples
    % number of sample timesteps
    sample_Ntimesteps = size(input_data{1,i},1);

    % Identify step between each pick (this is to ensure only at most, as
    % desired, timesteps being extracted)
    step_temp = ceil(sample_Ntimesteps/nTimesteps);  % step
    timesteps_ind = 1:step_temp:sample_Ntimesteps;  % Keep track of indices of timesteps ↙
to be extracted
    % In case there is not enough timesteps extracted as desired, the
    % second round of extraction is implemented
    if length(timesteps_ind) ~= nTimesteps
        % step between each pick is kept the same but the begining position
        % to be picked is set at between the two first picked timesteps
        timesteps_temp2 = 1+ceil(step_temp/2):step_temp:sample_Ntimesteps;
        % Identify the number of timesteps to be additionally picked
        Npts_to_add = nTimesteps - length(timesteps_ind);
        % Add the indices of timesteps to be additionally picked as
        % identified in the second round to the track of indices of timesteps to be ↙
extracted
        timesteps_ind = [timesteps_ind timesteps_temp2(1:Npts_to_add)];
    end

    % Sort indices before extraction
    timesteps_ind = sort(timesteps_ind);
    % Extract 3D time-series data features as picked
    extracted_data{1,i} = input_data{1,i}(timesteps_ind,:);
end
```

Appendix 5. Function data_reallocation.m



```
function [reallocated_data] = data_reallocation(input_data)
%Function [reallocated_data] = data_reallocation(input_data) turns a matrix
%of timeseries data into featured column vectors. It creates a matrix
%including samples in terms of column vectors with rows representing sample features.

%Input:
%   input_data:  A cell array representing, per cell, a matrix of
%   datapoints per sample.

%Output:
%   reallocated_data: A matrix presenting samples on columns and features
%   on rows.

reallocated_data = [];
nSamples = length(input_data);

% Loop through all sample in the input cell array
for i=1:nSamples
    sample_temp = [];
    % Loop through all sample timesteps
    for ts = 1 : size(input_data{1,i},1)
        % Turn all location data points into a feature data orderly
        % [x1; y1; z1; x2; y2; z2; ... ; x19; y19; z19]
        sample_temp = [sample_temp; input_data{1,i}(ts,:)'];

    end
    % Store sample vector. Therefore, The number of collumns is equal to
    % the number of samples inside the input data and the number of rows is
    % the number of data features
    reallocated_data = [reallocated_data sample_temp];
end
```

Appendix 6. Code file main.m

```
close all, clear all, clc
% Load ready data for training neural network model and store parameters
% for using in function C = digit_classify(testdata)

% Load preprocessed data (variables: data, class)
load data.mat

%% Step 1: Split data into train set and test set (90% train and 10% test)
[traindata, trainclass, testdata, testclass] = data_Holdout_splitting(data, class, ↙
0.1);

%% Step 2: Train and validate model (using 90% of data set)
% Architect of hidden layers (row vector). The number of elements is the number of
% hidden layers and the value of each element is the number of neurons in
% that hidden layer.
nHidden = [16,16];
[wHidden, wOutput, nHidden] = mlp_train_val(traindata, trainclass, nHidden);

%% Step 3: Test model with test set (using 10% of data set)
net_testclass = feed_forward(testdata, wHidden, wOutput);
fprintf("The accuracy of the network is %d percent.\n" , round(sum↙
(net_testclass==testclass)/length(testclass),2)*100);  % check accuracy on traindata↙
test set

%% Step 4: Store parameters
save parameters.mat wHidden wOutput
```

Appendix 7. Function data_Holdout_splitting.m

```
function [traindata, trainclass, testdata, testclass] = data_Holdout_splitting(data, ↙
class, testdataproportion)
%data_Holdout_splitting.m splits data into 2 parts (train sets and test
%sets) ensuring same proportion of samples of each class per set as the
%proportion of each set in the input data set.

%Input:
%   data: dataset(rows representing features and columns representing
%   samples)
%   class: sample label of the given data (1 row and as much column as the
%   number of samples)
%   testdataproportion: a number (less than 1) representing the proportion
%   of test set, the train set is the rest of the data.

%Output:
%   traindata: splitted train data set (rows representing features and
%   columns representing samples)
%   trainclass: sample class of the train data set (1 row and as much
%   column as the number of samples)
%   testdata: splitted test data set (rows representing features and
%   columns representing samples)
%   testclass: sample class of the test data set (1 row and as much
%   column as the number of samples)

if testdataproportion >= 1
    disp("Invalid input: test data proportion must be less than 1" );
    traindata = [];
    trainclass = [];

    testdata = [];
    testclass = [];
else
    trainInd = [];
    testInd = [];

    % Loop through each class and split the indices to the given proportion
    for label = 0:length(class)
        % find indices of class in question
        classInds = find(class==label);
        % number of samples whose class is in question
        nIndclass = length(classInds);
        % random permutation of indices of elements of classInds
        randomInds = randperm(length(classInds));

        % number of samples with such class in test data
        nTestInd = round(testdataproportion*nIndclass);
        % number of samples with such class in train data
        nTrainInd = nIndclass - nTestInd;

        % take test indices from vector of indices of class in question
        testlabelInd = classInds(randomInds(1:nTestInd));
        % take train indices from vector of indices of class in question
        trainlabelInd = classInds(randomInds(nTestInd+1:end));

        % store train indices of this class
        trainInd = [trainInd, trainlabelInd];
        % store test indices of this class
        testInd = [testInd, testlabelInd];
    end

    % Split data and class based on picked indices of each set
    traindata = data(:,trainInd);
    trainclass = class(trainInd);

    testdata = data(:,testInd);
    testclass = class(testInd);
end
```

Appendix 8. Function mlp_train_val.m

```
function [wHidden, wOutput, nHidden] = ...
  mlp_train_val(traindata_, trainclass_, nHidden, maxEpochs)
% mlp_train_val.m implements a shallow multilayer perceptron network using
% 90% of input data for training and the rest 10% for validation purpose.

% Input:
%   traindata_: input data set which is a data matrix with the orientation as that ↙
number of rows
%   is number of features and number of columns is the number of data
%   samples.
%   trainclass_: input data class which is a row vector with the number of
%   elements is the number of data samples.
%   nHidden: a row vector of which number of elements represents the number
%   of hidden layers and the value of each elements is the number of
%   neurons in per layer.

% Output:
%   t: scalar representing the number of epoches
%   wHidden: a cell array of matrices of weights of each hidden layer. The
%   number of cell is equal to the number of hidden layers. Each weight
%   matrix has the number of columns representing the number of neurons on
%   that hidden layer and the number of rows representing the number of
%   neurons on the preview layer.
%   wOutput: a matrix of weights of output layer. The number of columns is
%   equal to neurons on the output layer and is all equal to the number of
%   unique class in the input data set. The number of rows is equal to the
%   number of neurons (+1) on the last hidden layer.

%% Step 1: Split data into 2 sets, one for training model (90% of input data)
% and one for validate the model (10% of input data) and identify when
% overfitting might occurs
[traindata, trainclass, valdata, valclass] = data_Holdout_splitting(traindata_, ↙
trainclass_, 0.1);

%% Step 2: Acknowledge some attributes to recognize the suitable model's
% architecture and parameters
N = size(traindata, 2); % number of samples
d = size(traindata, 1); % number of features
nclass = length(unique(trainclass)); % number of classes
nHiddenLayers = size(nHidden, 2); % number of hidden layers
% Initialize maximum epochs if not provided
if ~exist('maxEpochs', 'var')
  maxEpochs = 100000;
end
%% Step 3: Print information about Network Architecture
fprintf("Network Architecture:\n");
fprintf("The input layer has (%d+1) neurons representing extended feature vectors of ↙
input data samples.\n", d);
fprintf("There are %d hidden layers in the network with extendedly each layer having " ,↙
nHiddenLayers);
```

```
for hidlayer = 1:nHiddenLayers
    fprintf("(%d + 1)", nHidden(hidlayer));
    if hidlayer~=nHiddenLayers
        fprintf(", ");
    end
end
fprintf(" neurons respectively.\n");
fprintf("The output layer has %d neurons representing the network data classes.\n" ,↙
nclass);
fprintf("This function utilizes validation data (10 percent) from the input data to ↙
validate parameters \nadjusted by training the model with train data (90 percent) of ↙
the input data set.\n");
fprintf("The total number of epochs is %d.\n", maxEpochs);
disp("Parameters are returned after the training finishes.");

%% Step 4: Initialize model parameters and attributes of model
% Extend input train data
extendedInput = [traindata; ones(1, N)];

% Tranform trainclass vector into a matrix form
trainOutput = zeros(nclass, N);
for i = 1:N
  trainOutput(trainclass(i)+1, i) = 1;
end

% Initialize weights for each hidden layer
wHidden = {};
for hidlayer = 1:nHiddenLayers
    if hidlayer == 1
        wHidden{hidlayer} = (rand(d+1, nHidden(hidlayer))-0.5) / 10;
    else
        wHidden{hidlayer} = (rand(nHidden(hidlayer-1)+1, nHidden(hidlayer))-0.5) / 10;
    end
end
% Initialize weights for output layers
wOutput = (rand(nHidden(nHiddenLayers)+1, nclass)-0.5) / 10;

% Initialize storage place of wHidden and wOutput per 100 epoch. This track
% is only used to track back the epoch when overfitting occurs if any.
wHidden_track = {};
wOutput_track = {};

% loss function value vector initialisation
Jtrain = zeros(1, maxEpochs); % of train data
Jval = zeros(1, maxEpochs); % of validation data
% Define learning rate
rho = 0.0001;

%fh1 = figure; % uncomment (line 92 and line 131-139) for visualization
% Initialize epoch count
```

```
t = 0; % Epoch count

%% Step 5: Train model
while 1
    t = t+1;
    % Initialize cel arrays to store output per hidden layer
    vHidden = {};
    yHidden = {};
    % Feed-forward operation
    for hidlayer = 1:nHiddenLayers
        % hidden layer net activation
        if hidlayer == 1
            vHidden{hidlayer} = wHidden{hidlayer}'*extendedInput;
        else
            vHidden{hidlayer} = wHidden{hidlayer}'*yHidden{hidlayer-1};
        end
        % hidden layer activation function
        yHidden{hidlayer} = tanh(vHidden{hidlayer});
        yHidden{hidlayer} = [yHidden{hidlayer}; ones(1,N)];  % hidden layer extended↙
output
    end

    % output layer net activation
    vOutput = wOutput'*yHidden{nHiddenLayers};
    % output layer output without activation function
    yOutput = vOutput;

    % loss function evaluation
    Jtrain(t) = 1/2*sum(sum((yOutput-trainOutput).^2));  % of traindata
    Jval(t) = cal_cost(wHidden, wOutput, valdata, valclass);  % of validation data

    % save wHidden and wOutput for every 100 epoches to have a track of
    % parameters when overfitting exists
    if (mod(t, 100) == 0)
        wHidden_track{t/100} = wHidden;
        wOutput_track{t/100} = wOutput;
    end

    % Plot training error at every 100 epoch % uncomment (line 92 and line 131-139) for ↙
visualization
%     if (mod(t, 1000) == 0)
%         semilogy(1:t, Jtrain(1:t), "b-"), hold on;
%         semilogy(1:t, Jval(1:t), "r-");
%         title(sprintf('Training (epoch %d)', t));
%         ylabel('Error');
%         legend("training","validation");
%         drawnow;
%     end

    % Check if the learning is good enough, if yes, stop the training
    if (Jtrain(t) <1e-10)
    break;
    end

    % Check if maximum epochs has been reached, if yes, stop the training
    if t >= maxEpochs
    break;
    end
    % Check if the improvement is too small, if yes, stop the training
    if t > 1 % this is not the first epoch
        if abs(Jtrain(t) - Jtrain(t-1)) < 1e-10
          break;
        end
    end

    % Update the sensitivities and the weights
    % For output layer
    deltaOutput = (yOutput - trainOutput);
    deltawOutput = -rho * yHidden{nHiddenLayers} * deltaOutput';  % (E7)
    wOutput = wOutput + deltawOutput; % update wOutput

    % For hidden layers
    deltaHidden = {};
    for hidlayer = nHiddenLayers:-1:1
        if hidlayer == nHiddenLayers % if this is the last hidden layer, consider ↙
output layer parameters
            deltaHidden{hidlayer} = (wOutput(1:end-1,:)*deltaOutput).*(1-yHidden ↙
{hidlayer}(1:end-1,:).^2);
            deltawHidden{hidlayer} = -rho * yHidden{hidlayer-1} * deltaHidden ↙
{hidlayer}';
        else % otherwise, consider the next hidden layer
            deltaHidden{hidlayer} = (wHidden{hidlayer+1}(1:end-1,:)*deltaHidden ↙
{hidlayer+1}).*(1-yHidden{hidlayer}(1:end-1,:).^2);
            if hidlayer == 1 % if this is the first hidden layer, consider input layer ↙
parameters
                deltawHidden{hidlayer} = -rho * extendedInput * deltaHidden{hidlayer}';
            else % otherwise, consider the previous hidden layer
                deltawHidden{hidlayer} = -rho * yHidden{hidlayer-1} * deltaHidden ↙
{hidlayer}';
            end
        end
        wHidden{hidlayer} = wHidden{hidlayer} + deltawHidden{hidlayer};  % update↙
wHidden
    end
end

%% Step 6: Check if overfitting exists
% Overfitting is likely identified when the validation data has loss
% function value starting to increase
[min_Jval, min_Jval_ind] = min(Jval(100:100:end));
```

```
% Acknowledge overfitting if the last loss function value is not the
% minimum value
if min_Jval ~= Jval(end)
    fprintf("Overfitting exists from the %d epoch.\n", min_Jval_ind*100);
    wHidden = wHidden_track{min_Jval_ind};
    wOutput = wOutput_track{min_Jval_ind};
    disp("The return parameters are the ones updated just right before overfitting ↙
occurs.");
end
end

function Jval = cal_cost(wHidden, wOutput, valdata, valclass)
% This function operates feed-forward for the input data using given
% parameters and calculates loss function value

% Acknowledge attributes
nHiddenLayers = length(wHidden);
nVal = size(valdata, 2); % number of validation data samples
nclass = length(unique(valclass)); % number of classes
extendedVal = [valdata; ones(1, nVal)]; % extended input test data

% Tranform input class into a matrix to ease loss calculation
valOutput = zeros(nclass, nVal);
for i = 1:nVal
  valOutput(valclass(i)+1, i) = 1;
end

% Initialize cel arrays to store output per hidden layer
vHiddenVal = {}; % Input into per hidden layers
yHiddenVal = {}; % Output per hidden layers (apply tanh activation function onto input)

% Feed-forward operation
for hidlayer = 1:nHiddenLayers
    % hidden layer net activation
    if hidlayer == 1
        vHiddenVal{hidlayer} = wHidden{hidlayer}'*extendedVal;
    else
        vHiddenVal{hidlayer} = wHidden{hidlayer}'*yHiddenVal{hidlayer-1};
    end
    % hidden layer activation function
    yHiddenVal{hidlayer} = tanh(vHiddenVal{hidlayer});  % hidden layer activation↙
function
    yHiddenVal{hidlayer} = [yHiddenVal{hidlayer}; ones(1,nVal)];  % hidden layer↙
extended output
end

vOutputVal = wOutput'*yHiddenVal{nHiddenLayers};  % output layer net activation
yOutputVal = vOutputVal;  % output layer output without activation f

% loss function evaluation
Jval = 1/2*sum(sum((yOutputVal-valOutput).^2));
end
```

Appendix 9. Function feed_forward.m

```
function net_class = feed_forward(data, wHidden, wOutput)
%Function net_class = feed_forward(data, wHidden, wOutput)
%feed input data sample into trained model to predict its label

%Input:
%   data: Input data sample to be feed into model (a matrix with columns
%   corresponding samples and rows corresponding features)
%   wHidden: Parameters of hidden layers (a cell array with each cell
%   representing weights of each hidden layer) of the trained model
%   wOutput: Parameters of output layer (a matrix) of the trained model

%Output:
%   net_class: The label of the input data sample predicted by the
%   model

nHiddenLayers = length(wHidden); % number of hidden layers
ntest = size(data, 2); % number of data samples
extendeddata = [data; ones(1, ntest)]; % extended input data

% Initialize cel arrays to store output per hidden layer
vHiddentest = {}; % Input into per hidden layers
yHiddentest = {}; % Output per hidden layers (apply tanh activation function onto ↙
input)
% Feed-forward operation
for hidlayer = 1:nHiddenLayers
    % hidden layer net activation
    if hidlayer == 1
        vHiddentest{hidlayer} = wHidden{hidlayer}'*extendeddata;
    else
        vHiddentest{hidlayer} = wHidden{hidlayer}'*yHiddentest{hidlayer-1};
    end
    % hidden layer activation function
    yHiddentest{hidlayer} = tanh(vHiddentest{hidlayer});
    yHiddentest{hidlayer} = [yHiddentest{hidlayer}; ones(1,ntest)];  % hidden layer↙
extended output
end

% output layer net activation
vOutputtest = wOutput'*yHiddentest{nHiddenLayers};
% output layer output without activation function
yOutputtest = vOutputtest;

% find most possibly correct class label for input data
[~, net_class] = max(yOutputtest, [], 1);
% because the max function outputs as indices from 1-10 and our class is from 0-9
net_class = net_class - 1;
```

Appendix 10. Function digit_classify.m

```
function C = digit_classify(testdata)
%Function C = digit_classify(testdata) takes input as a matrix N*3 data sample
% of an air-written digit collected by LeapMotion sensor and does the
% recognition of the written digit.

%Input:
%   testdata: a matrix N*3 data sample (N number of 3-D location datapoint ↙
trajectories)

%Output:
%   C: The label of the written digit predicted by the trained neural
%   network model

    % Turn sample data from a matrix to a cell array with one cell because
    % of input format of called functions
    test_datacell ={};
    test_datacell{1,1} = testdata;
    % number of timesteps (data points) of the given to-be-classified sample
    nTimesteps = size(test_datacell{1},1);
    % Load data set which was used to build network model
    load data.mat data class
    %% Step 1: Normalize data
    test_datacell = data_normalization(test_datacell);

    %% Step 2: Extract features to ensure consistent size of data sample and
    model_data_size = size(data,1)/3;
    % If the input testdata sample has more point trajectories than the
    % data samples used to train the model, the testdata sample needs to
    % proceed feature extraction
    if nTimesteps > model_data_size
        test_datacell = feature_extraction(test_datacell, model_data_size);
    % SPECIAL CASE: MODEL RETRAINING NEEDED
    elseif nTimesteps < model_data_size
        % extract feature for the given data sample
        test_datacell = feature_extraction(test_datacell, nTimesteps);
        % Repreprocess raw data
        load raw_data.mat raw_data class
        normalized_traindata = data_normalization(raw_data);
        extracted_traindata = feature_extraction(normalized_traindata,nTimesteps);
        train_data = data_reallocation(extracted_traindata);
        % Retrain network model and identify parameters
        [traindata_, trainclass_, testdata_, testclass_] = data_Holdout_splitting ↙
(train_data, class, 0.1);
        nHidden = [16,16];
        [wHidden, wOutput, nHidden] = mlp_train_val(traindata_, trainclass_, nHidden);
        net_testclass = feed_forward(testdata_, wHidden_, wOutput_);
        % check accuracy on traindata test set and print prompt
        fprintf("The accuracy of reimplemented network is %d.\n" , sum↙
(net_testclass==testclass_)/length(testclass_));
    end
    %% Step 3: Reallocate test data sample (turn a cell array with one cell into a ↙
column vector)
    test_datavec = data_reallocation(test_datacell);

    %% Step 4: Predict label for input testdata
    if nTimesteps < model_data_size
        % SPECIAL CASE: USE RETRAINED PARAMETERS
        C = feed_forward(test_datavec, wHidden_, wOutput_);
    else
        % NORMAL CASE: USE TRAINED PARAMETERS
        load parameters.mat
        C = feed_forward(test_datavec, wHidden, wOutput);
    end
```

Appendix 11. Code file test.m

```
%% Test function digit_classify.m and the trained model with all raw data available in ↙
training phase
% Load preprocessed data (variables: data, class)
load raw_data.mat raw_data class
load parameters.mat

% Initialize vector to keep track of predict performance
isCorrect = zeros(1,length(raw_data));

% Loop through the whole available data set
for sample = 1:length(raw_data)
    % testdata (one sample at a time)
    testdata_ = raw_data{1,sample};
    testclass_ = class(sample);
    % Predict class for testdata with function digit_classify
    net_testlabel = digit_classify(testdata_);

    % Printing result
%     fprintf("The test data class is %d and the network label that test data as class ↙
%d.\n", testclass_, net_testlabel);
    if testclass_ == net_testlabel
        isCorrect(sample) = 1;
%         disp("The model has correctly labelled the given test data");
    else
%         disp("The model has incorrectly labelled the given test data");
    end
end

accuracy = sum(isCorrect)/length(isCorrect);
```