INF3490 Mandatory Assignment 2: Multilayer Perceptron

Paul Wieland

Deadline: Tuesday, October 16th, 2018 23:59:00

Contents

1 Introduction

1.1 Task

We will build a Multilayer Perceptron to steer a robotic prosthetic hand. There are 40 inputs of electromyographic signals that we will classify.
There are 8 classification values corresponding to a different hand motion:

Figure 1: Possible motions [1]

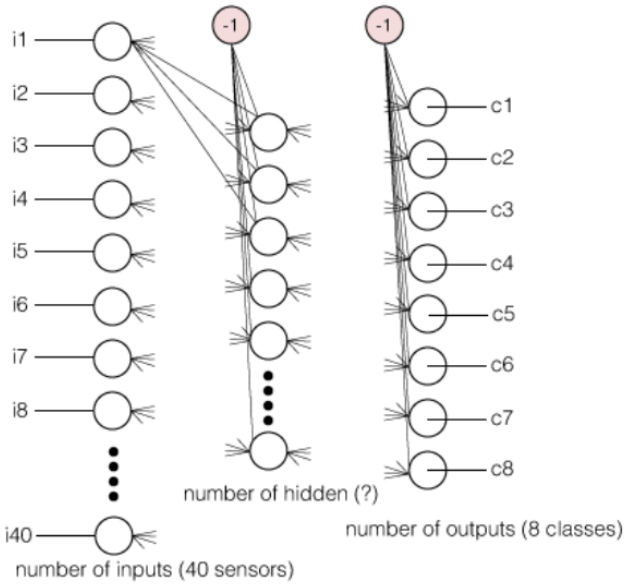

Figure 2: Multilayer Perceptron for our problem [2]

We build a Multilayer Perceptron with 40 entry nodes, that means one node for each input. Then there is one hidden layer with a various number of hidden nodes. For classifying the input, there are 8 output nodes corresponding to the 8 hand motions. We only use one hidden layer to solve this problem.

[1] http://folk.uio.no/kyrrehg/pf/papers/glette-ahs08.pdf
[2] https://www.uio.no/studier/emner/matnat/ifi/INF3490/h18/assignments/assignment-2/assignment_2.pdf

1.2 Training Data

For each input vector:

$$input = [i_1, i_2, i_3, i_4, ..., i_{40}], i_n \in \mathbb{R}, n \in [40] \tag{1}$$

we have a target output vector:

$$output = [c_1, c_2, c_3, c_4, ..., c_8], c_n \in \{0, 1\}, n \in [8], \sum_{n=1}^{8} c_n = 1 \tag{2}$$

That means, forwarding the input should result in the given target vector.

2 Implementation

The file *mlp.py* contains the class mlp. There are 5 functions that i will explain in detail.

2.1 Initialization

The function *__init__(self, inputs, targets, nhidden)* has three important parameter that we need to initialize the Multilayer Perceptron.
As the input data is given as a vector, it is a good choice to create two 2D-Array for the two weight layers. As the parameters *inputs* and *targets* have the type *<class 'numpy.ndarray'>*, it is a good idea to work only with numpy arrays.

2.1.1 Dimension of the weight matrix

- weight_matrix_1:
  The input vector in (1) has of course a size of 40. But we need to add the *bias_value -1* that can be seen in Figure 2. That means:

$$weight\_matrix\_1 \in \mathbb{R}^{41 \times nhidden} \tag{3}$$

$w_{i,j} \in weight\_matrix\_1, w_{i,j}$: weight between input node(i) and hidden node(j)

- weight_matrix_2:
  There are *nhidden* hidden nodes and 8 exit nodes. So we also need to take into account the *bias_value -1*. That means:

$$weight\_matrix\_2 \in \mathbb{R}^{(nhidden+1)\times 8} \tag{4}$$

$w_{i,j} \in weight\_matrix\_1, w_{i,j}$: weight between hidden node(i) and output node(j)

Both, *weigh_matrix_1* and *weigh_matrix_2*, will be initialized randomly with values in [-1,1].

2.2 Forward

The forward function takes one input vector and runs it on the network. At first, the input vector must be expanded. The reason for this is that we have to take into account the bias value:

$$input \in \mathbb{R}^{1\times 41} \tag{5}$$

2.2.1 Forward Phase 1

Subsequently it is possible to compute the *hidden_values*:

$$hidden\_values = [h_1, h_2, h_3, h_4, ..., h_8] \tag{6}$$

$$h_i = \sum_{n=1}^{41} input[n] \times weight\_matrix\_1[n][i] \tag{7}$$

This operation is done by the function *vec_matr_mult()* (that function can be found in the file *operations.py*) that is doing a vector matrix multiplication with two for-loops.

2.2.2 Activation Function

After that is the activation function applied to all hidden nodes:

$$a_\zeta = h_{new,i} = \frac{1}{1 + \ exp(-\beta h_i)} \tag{8}$$

(*apply_sigmoid_activation()* in *operations.py*)

2.2.3 Forward Phase 2

The result *hidden_values* is also expanded by the bias -1:

$$hidden\_activation \in \mathbb{R}^{1\times(nhidden+1)} \tag{9}$$

So the *output* vector can be calculated easily by another vector matrix multiplication:

$$output = hidden\_activation \cdot weight\_matrix\_2 \tag{10}$$

2.2.4 Output Error

For the first implementation a linear output function was used. But the behavior of the algorithm was not really good. So I decided to apply the sigmoid function (8) to the output as well. The error function for the output node is therefor:

$$\delta_o(\kappa) = (y_\kappa - t_\kappa)y_\kappa(1 - y_\kappa) \tag{11}$$

This works much better. For example, we can see that the target value converges to 1 and the other to 0. Here is an example of the training set:

output	1.6e-02	7.0e-02	1.6e-02	2.9e-02	4.7e-04	1.1e-05	6.6e-05	8.4e-01
target	0	0	0	0	0	0	0	1

And another output of the validation set:

output	1.6e-02	1.0e-04	9.8e-01	9.4e-03	6.4e-05	2.6e-04	1.6e-02	2.0e-02
target	0	0	1	0	0	0	0	0

2.3 Calculate Hidden-Error

From the forward phase we have gained the *output-error*(11) and the *activation-values*(7),(8) of the hidden nodes. That means it is possible to calculate the errors of the hidden nodes. The error of the hidden layer is determined by the function *calculate_hidden_error()*. It follows the formula:

$$\delta_h(\zeta) = a_\zeta(1 - a_\zeta)\sum_{k=1}^{N} w_\zeta\delta_o(k) \quad ,\beta = 1 \tag{12}$$

Where:

- $a_\zeta \in$ *hidden_activation*, because $\beta a_\zeta(1-a_\zeta) = \frac{d}{dh}\frac{1}{1+\ exp(-\beta h_i)}$, see (8)
- $w_\zeta \in$ *weight_matrix_2*, see (4)
- $\delta_o(k)$ *inoutput_error*, see (11)

So, w_ζ is the weight that connects the hidden node with activation a_ζ and the output node with output-error $\delta_o(k)$.

2.4 Train the network

To train the network means to backpropagate the errors we that we have calculated in 2.2.4 and 2.3. So the weights of *weight_matrix_1* and *weight_matrix_2* will be adjusted by the following formula:

$$w_{ij} = w_{ij} - \eta\delta_j x_i \tag{13}$$

Where:

- w_{ij} is the weight that connects an input node(i) and an output node(j). Input and output means only the activation direction, not any specific layer.
- x_i is the activation value of an input node. From the view of *weight_matrix_1* is the input vector of the network the meant input. From the view of *weight_matrix_2* is the hidden layer the input one.
- δ_j is the error of the output node.
- η is the learning rate of the network.

2.5 Earlystopping

The function *earlystopping()* observes the learning of the network. It should avoid overfitting. That means that the network is adjusted too much to the training data so unknown data will be classified very bad.

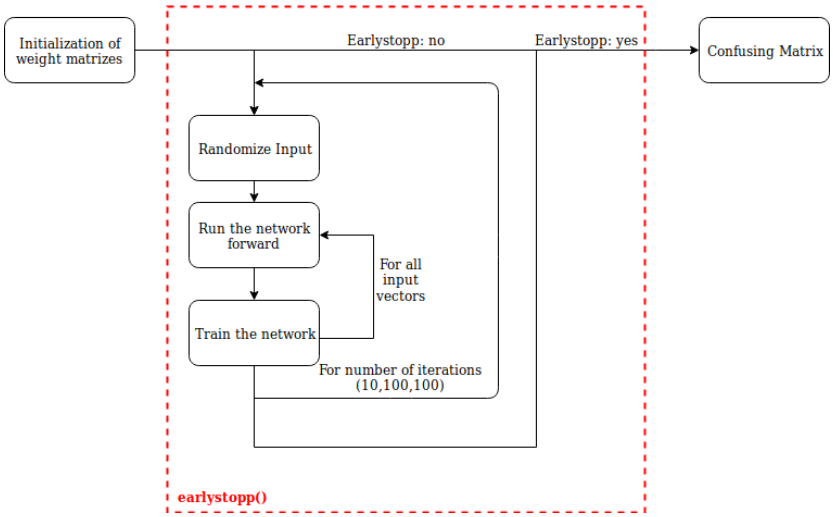

Figure 1: Main flow of the Multilayer-Perceptron

As you can see in the figure above, *Sequential Training* is used. That means the network will be trained after one input vector. One iteration is therefor one iteration through the whole data input. That is also the reason why the input data will be randomized each iteration. The randomization should avoid learning the network with data in the same order. That may lead to bad generalization issues.

2.5.1 Decision earlystopping

As you can see in the graphic above, the algorithm trains the network for a certain number of iterations before the validation set will be applied.The validation set is unknown to the network. The Error-Function:

$$E(w) = \frac{1}{2}\sum_{k}(t_\kappa - y_\kappa)^2 \tag{14}$$

is used to estimate the quality of the network. So after each epoch (one is a certain number of iterations, eg. 10,100,1000) the Error-Function will be applied to the validation set. If the error of one epoch if bigger than the epoch before, the algorithm stops to avoid overfitting.

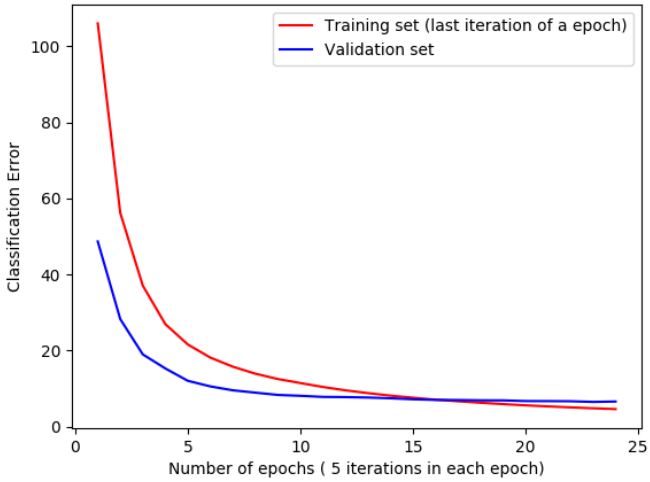

Figure 2: Effect of Earlystopping

In the graphic above you see very good how earlystopping works. Both, the Training-Error and the Validation-Error decrease. The Training-Error would reach zero if there is no stopping criteria. That means also that the Validation-Error would increase again because the network is not able to classify unknown data as good as when the algorithm stops earlier. So the Algorithm stops if the blue line starts to increase.

2.6 Confusion

For the confusion phase, a set of unknown data will be forwarded. The number of wrong classifications is represented as a confusion-matrix. Where on the x-axis are the target classes on on the y-axis the classifications that are made by the network.

3 Results

The network will be tuned and forwarded to evaluate and compare the results. All tests are made with 10 iterations in each epoch. For each number of nodes were made 10 tests because the random initialization of the weight matrices can influence the quality of the result.

3.1 Hidden Nodes: 6

$$confusion\_matrix = \begin{bmatrix} 12 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 15 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 13 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 17 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & 11 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 12 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 10 & 0 \\ 0 & 0 & 0 & 2 & 0 & 0 & 0 & 12 \end{bmatrix}$$

Percentage of correct classes:	37.5 %
Number of inputs:	111
Wrong classifications:	9
Percentage wrong classification:	8.1 %

For more results see the file *mlp_6_hidden_nodes.txt*

3.2 Hidden Nodes: 12

$$confusion\_matrix = \begin{bmatrix} 11 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 17 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 14 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 9 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 16 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 14 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 13 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 13 \end{bmatrix}$$

Percentage of correct classes:	62.5 %
Number of inputs:	111
Wrong classifications:	4
Percentage wrong classification:	3.6 %

For more results see the file *mlp_12_hidden_nodes.txt*

3.3 Hidden Nodes: 18

$$confusion\_matrix = \begin{bmatrix} 13 & 0 & 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 9 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 16 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 12 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 12 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 13 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 13 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 16 \end{bmatrix}$$

Percentage of correct classes:	25.0 %
Number of inputs:	111
Wrong classifications:	7
Percentage wrong classification:	6.3 %

For more results see the file *mlp_18_hidden_nodes.txt*

3.4 Conclusion

As you can in the files the results differ sometimes more, sometimes less. The reason for this behavior is that the weight are initialized randomly. So the network will be trained non-deterministic.
So when we take a look on a common result like in 3.1, 3.2 and 3.3 we can see, that a network with 6 and 18 hidden node gives us very similar results. The number of wrong classified vectors is very similar. But there is also a difference. When I run a network with 18 nodes, it is common to get confusion-matrices with 25 % correct classes. For 6 hidden nodes it is more like 37.5 %. That means when we use 18 hidden nodes, the result is not really better, but the failures are more spread to all classes.
The best result, I think, is produces a network with 12 hidden layers. The confusion matrices are having the highest percentage of correct classifications. The total wrong classifications is in many results also very low.