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:

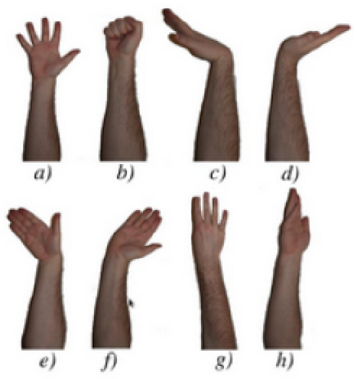

Figure 1: Possible motions [1]

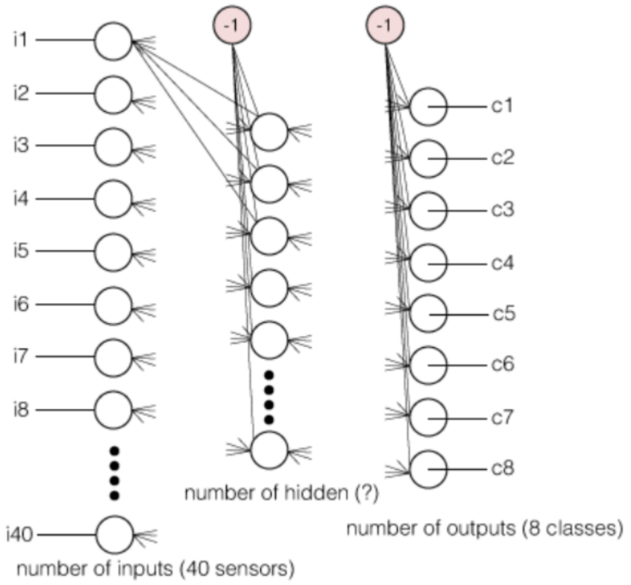

Figure 2: Multilayer Perceptron for our problem [2]

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:

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

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

- 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:

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

(*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

The forward function returns also a vector that is a converted version of the output-vector. This *output_discrete* has a 1 where the output vector has its highest value, the rest of the vector is 0. So the output-error for each node can be calculated very easy:

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

$$y_\kappa \in output\_discrete,$$
$$t_\kappa \in output, \text{ see (2)},$$
$$output\_error = [\delta_o(1), ..., \delta_o(8)]$$

This calculation results from a linear output activation function.

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:

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

Where:

- $a_\zeta \in hidden\_activation$, see (7), (8)
- $w_\zeta \in weight\_matrix\_2$, see (4)
- $\delta_o(k)\ in output\_error$, 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.

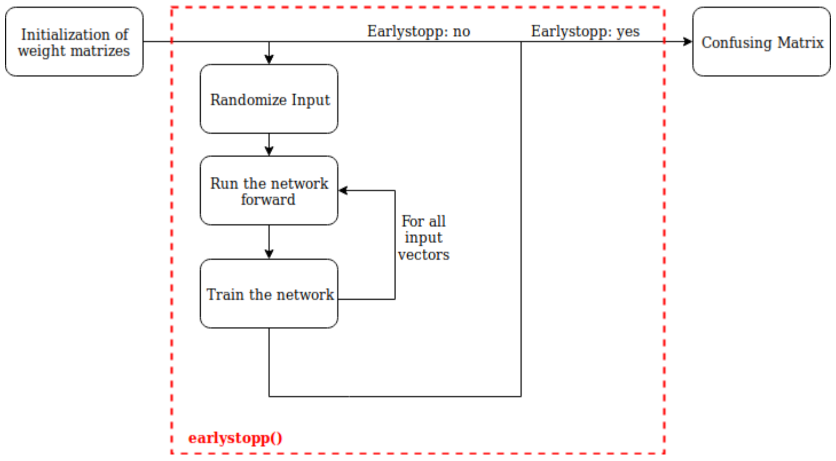

Figure 1: Main flow of the Multilayer-Perceptron

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 bad generalization issues.

2.5.1 Decision earlystopping