ASSIGNEMENT 4

Neural networks and PSO learning

**ASSIGNEMENT**:

Within this assignment you will build system for breast cancer classification. In the data.mat there are values for certain characteristics obtained from breast scans. Based on these values one can classify cancer as benign or malign. You will build classification system using neural networks. For learning of the neural network you will use default matlab learning algorithm (net = train(net,x,t);) and compare it to the PSO algorithm, that you will program. The first column in the data matrix denotes the tumor (2-malign, 4-benign).

Analyze and comment the obtained results (results on learning and on test set). What if all activation functions in the neural network are linear? Do you still get good results? Does the neural network still provide nonlinear border between the classes?

The report should contain basic theoretical background about the used algorithms, analysis of obtained results and your comments.

**HELP:**

# PSO optimization

The PSO method belongs to evolutionary optimization methods. It has some similar implementation steps than GA: random generation of initial population, evaluation of criterion function for each particle and generation of new population based on criterion function value and some random number. Both methods are good for finding the global sub optimum, but do not guarantee optimum solution. With the PSO the particles are moving in the problem space with the help of their speed. Each particle has its own memory of its best position. Population is denoted as swarm in the PSO optimization. Individuals are particles. The algorithm is based on deterministic adaptation of particle positions with added stochastic.

The algorithm usually has two factors that need to be defined *c1* and *c2*. Acceleration or self-confidence factor *c1* is a weight of the *pbest-*its own best position. Acceleration or swarm-confidence factor *c2* is a weight of swarm best position *gbest*. They are usually chosen from interval [0, 4] and *c1* usually equals *c2*.

When doing the optimization, the problem space is usually bounded with *Xmax* and *Xmin* constants. Boundaries ensure, that the particles don’t leave the problem space. If a particle is outside the limits its speed/velocity is set to 0 and the position is reset to Xmin or Xmax. Maximal speed/velocity is also defined. There are several possibilities mentioned in the literature. One of those is to set the maximal speed as one fifth of the problem space range: *Vmax = (Xmax - Xmin) ./ 5*.

The number of particles for the optimization is usually set somewhere between 20 and 50. One of the options found in the literature is also to bind the maximal number of particles to the dimensionality of the problem: *PART\_NUM = round(10+2\*sqrt(Dim))*, where *Dim* represents the problem space dimension.

The optimization algorithm starts by setting the initial position ***xi***, and speed ***v****i*  of each particle randomly inside the allowed bounds. For the initial positions the criterion function is calculated f*\_best*. The current position of each particle is stored also as best particle position ***pbest****i.* Next we find the global best position (***gbest***) and the global best value of criterion function (*fg\_best*). With this the initial steps of the PSO optimization are finished. From now on the iterative procedure starts with the following steps:

* Calculate new particles’ positions
* Check if the positions of all particles are inside the allowed bounds if not put them inside the bounds and set speed to 0
* Check if the speed of the particle is inside the allowed bounds if not set the speed to 0
* Calculate the criterion function for each particle
* Compare the new criterion function value with *f\_best*. If the new value is better, replace the old value with the new and replace the best position ***pbest****i* with the current position
* Find the particle with the best *f\_best value.* Compare that value to the *fg\_best*. If it is better, replace the *fg\_best* with this value and change the old ***gbest*** with the particle ***pbest****i*.
* Check the conditions for ending the optimization. If the criterions are met then our solution is in the ***gbest*** variable. If not repeat the above steps.

**Some Examples of PSO**

1. **SIMPLE PSO**:

Equations for position and speed adaptation are:

where *i* denotes particle and *j* denotes dimension.

1. **Weighted PSO (PSO-W: PSO with inertia weight):**

Equations for position and speed adaptation are:

where *i* denotes particle and *j* denotes dimension. To the basic equation weight *W* is added. Weight represents the particle inertia. With this weight the activity of particle is changed. Usually this weight is set to 0.7. Some versions also change this weight dynamically.

1. **PSO with constriction factor (PSO-C):**

Equations for position and speed adaptation are:

where *i* denotes particle and *j* denotes dimension. Factor was introduced for intention of improving the convergence of the algorithm.

1. **Hybrid PSO with mutation (HPSOM):**

Equations for position and speed adaptation are:

where *i* denotes particle and *j* denotes dimension. The speed and position of particles are adapted with standard equations. After the adaptation the mutation on particles is made. The mutation is defined by probability factor λm. If *rand< λm* then the mutation of particle position is made. The mutation is done as:

The *dxj* is obtained with the generation of random number from the interval [0, ***Xmax***j-***Xmin***j]. With the mutation the behavior of algorithm is improved around the local optimum.

# Neural networks

For solving the classification problem, we can use a feedforward neural network. Neural networks usually consist of three layers: input layer responsible for connecting the inputs to neurons in the hidden layer; hidden layer (we can have multiple hidden layers); and an output layer where the number of neurons is equal to the number of outputs. The number of neurons in the hidden layer depends on a problem and is set by the user. The circles in figure 1 represent the neurons.

The functioning of the neuron is presented in figure 2. The input neurons have no transfer functions and weights – are direct feed-trough.

Figure 1: graphical representation of neural network

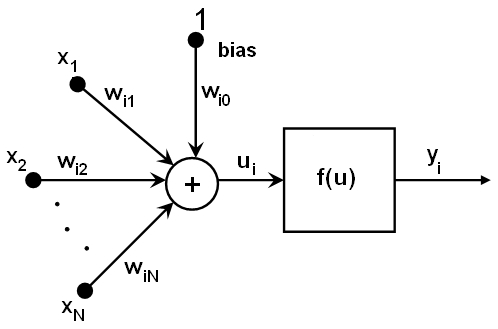


Figure 2: neuron representation

The output of the neuron is the value of function f(u). The value of u is a linear combination of neuron’s inputs (x1\*w1i + x2\*w2i+...+xN\*wNi+wi0). The learning procedure is a procedure that optimizes the weights so that for the given pairs of inputs and outputs the output of the neural network is the same (or almost the same) as the given outputs.

# Help for the assignment

The neural network structure is already implemented in Matlab. The first thing you have to do is split the data into learning and test data set. Try to split the data set so that the distribution of classes is the same in test and in learning data. For example, if in the whole data set you have 20 samples of one and 10 samples of other class the data set would be partitioned in the 2:1 ratio (for example: 14 samples from one class and 7 samples from other class will go into learning set other samples will go into test set). Due to numerical reason the data is usually normalized/standardized to the interval [0 1].

The neural network is generated as:

net = newff(x,t,n,{'tansig','tansig'});

With this the feedforward neural network is created with tansig transfer functions in the hidden and output layer. The parameter n defines the number of neurons in the hidden layer. The x is the matrix of inputs and t is the matrix/vector of target outputs. Note that vector t and matrix x are row based (each variable is represented in a row not in a column).

The NN is initialized with:

net = init(net);

net = configure(net,x,t);

The initialization initializes the weights and sets the proper dimensions of input and output layer. The net can be viewed with a call of the following function: view(net).

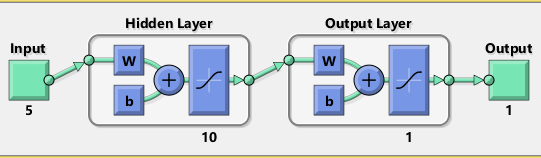


Figure 3: example of neural network with 5 inputs and one output

The structure of the network is displayed in Matlab if we type in the name of the neural network and press enter in the command line. The information is displayed in sections. In section dimensions the dimensions of inputs, number of layers, number of outputs and weights are displayed along with some other information. The Matlab treats the data a bit differently than expected. The number of inputs will in your case be one (numInputs = 1). Note that the numInputs denotes the number of input sequences. In most cases this is one. The number of inputs is found in the variable net.inputs{1}.size. This number must equal to the number of input variables. The number of layers is in our case two (numLayers = 2). We are working with three-layered neural network and Matlab doesn’t consider the input layer as a layer. The number of weights (numWeightElements) must in our case be equal to: (number of weights + 1) \* number of hidden layers’ neurons + number of hidden layers’ neurons +1.

In section connections information about the connections between the layers is displayed. The biasConnect: [1; 1] : defines the connection between bias and layers (b). The vector [1;1] means that the output layer also has bias b; inputConnect: [1; 0] shows the connection of input layer. Vector [1;0] means that the inputs are connected to the hidden layer and no one goes directly to the output layer. The outputConnect: [0 1] defines which layer calculates the end result. Vector [0 1] – means that the output is calculated in the output layer. The layerConnect: [0 0; 1 0] tells us which layer is connected together with the weights. If we want to know if the signal goes from layer one to layer two trough the weight we can type net.layerConnect(2,1) in the command window and press enter. The return of 1 means YES the return of 0 means NO.

ACCESSING THE WEIGHTS OF THE NEURAL NETWORK:

The weights are defined in net.IW (in our case net.IW{1}). These are the weights for connections going from input layer to the hidden layer. The number of rows of this matrix equals to the number of neurons in the hidden layer, the number of columns equals to the number of NN inputs.

The bias values are stored in net.b. The hidden layer biases are in net.b{1} and the output layer biases are in net.b{2}.

The weights for connection between layers are stored in net.LW. In our case for connections between hidden and output layer (net.LW{2,1}). For getting the vector of weights we can call function:

getwb(net)

The weights can be set with function call: setwb(net,weights)

The output of neural network can be calculated with function call: yest = net(x).

The MSE criterion can be calculated as: mse\_calc = sum((yest-targets).^2)/length(yest).

Some other comments for this assignement:

If you use all input data (all given characteristics) the border between the two classes is supposed to be linear. This means that lower number of hidden layer neurons are needed (1-5 neurons should be enough).

If you do not use all input data, you will have to use a bit more neurons in the hidden layer.

The PSO optimization will take a long time so I recommend you to first test the number of neurons using matlab toolbox.

Since the PSO is stochastic method the results will vary for each optimization run. Try to run the optimization several times and analyze the results.

Calculate the results for learning and test data separately. Present them in a form of confusion matrix. This is usually the standard form for representing classification results:

plotconfusion(y,yest)

where y and yest are row vectors. Note that in our case we have two classes and we worked with only one output. Therefore, we have to round the outputs of the neural network before we plot the confusion matrix. We also have to split the output vector into two dimensions. Let’s say that value 0 denotes first class and value 1 the second class. The vectors y and yest should be transformed as:

y = [0.0 1.0 0.0 1.0 1.0 0.0]-> y =[1.0 0.0 1.0 0.0 0.0 1.0; 0.0 1.0 0.0 1.0 1.0 0.0]

yest = [0.1 0.6 0.01 0.02 0.8 0.3] -> yest = [1.0 0.0 1.0 1.0 0.0 1.0; 0.0 1.0 0.0 0.0 1.0 0.0]