Faculty of Mechanical Engineering and Robotics

Department of Robotics and Mechatronics

Basics of AI and Deep Learning

Course for Mechatronic Engineering with English as instruction language

Instruction 4:

<u>Neural networks in Matlab toolbox</u>

You will learn: How to use matlab toolbox for designing and using neural networks and how to evaluate their efficiency based on simulated data. We will learn how to recognize and deal with overfitting, we'll also design some experiments of our own that will help us understand neural-networks a bit better.

Additional materials:

- Course lectures 1 - 4

ChatGPT was used in preparation of this instruction – for commenting codes, unification of architecture of codes to adjust them for selected tasks and bugfixes and for preparation of additional tasks 4.9 – 4.11

Course supervisor:
Ziemowit Dworakowski, zdw@agh.edu.pl

Instruction author:
Ziemowit Dworakowski, zdw@agh.edu.pl

Baseline for comparison

We have been designing polynomial regressors and neural-network-based classifiers from scratch (and we were training them using our own implementations of optimization algorithms). While it hopefully provided a better insight into how the method internals work, it might not be the preferred approach in terms of efficiency in solving practical problems. Lets now compare implementations developed in laboratory 2 and 3 with those available in matlab toolboxes. Lets start with a regression problem. Similarly as before, we'll use **RegressionTrainingData** to configure model parameters and **RegressionTestingData** for final evaluation. We will use multilayered neural network (MLP) as our model for regression.

In order to train the network, we will have to divide our data into training samples and their corresponding target values – which so far were stored just in the third column of the data matrix. Now we'll need to pass them separately to our network:

```
load('RegressionData1.mat');  % Make sure the .mat file is in your path

% Inputs (first two columns) and targets (third column)
X = RegressionTrainingData(:, 1:2)';
T = RegressionTrainingData(:, 3)';
```

Now we need to design our network and then train it, which we can do using the following code:

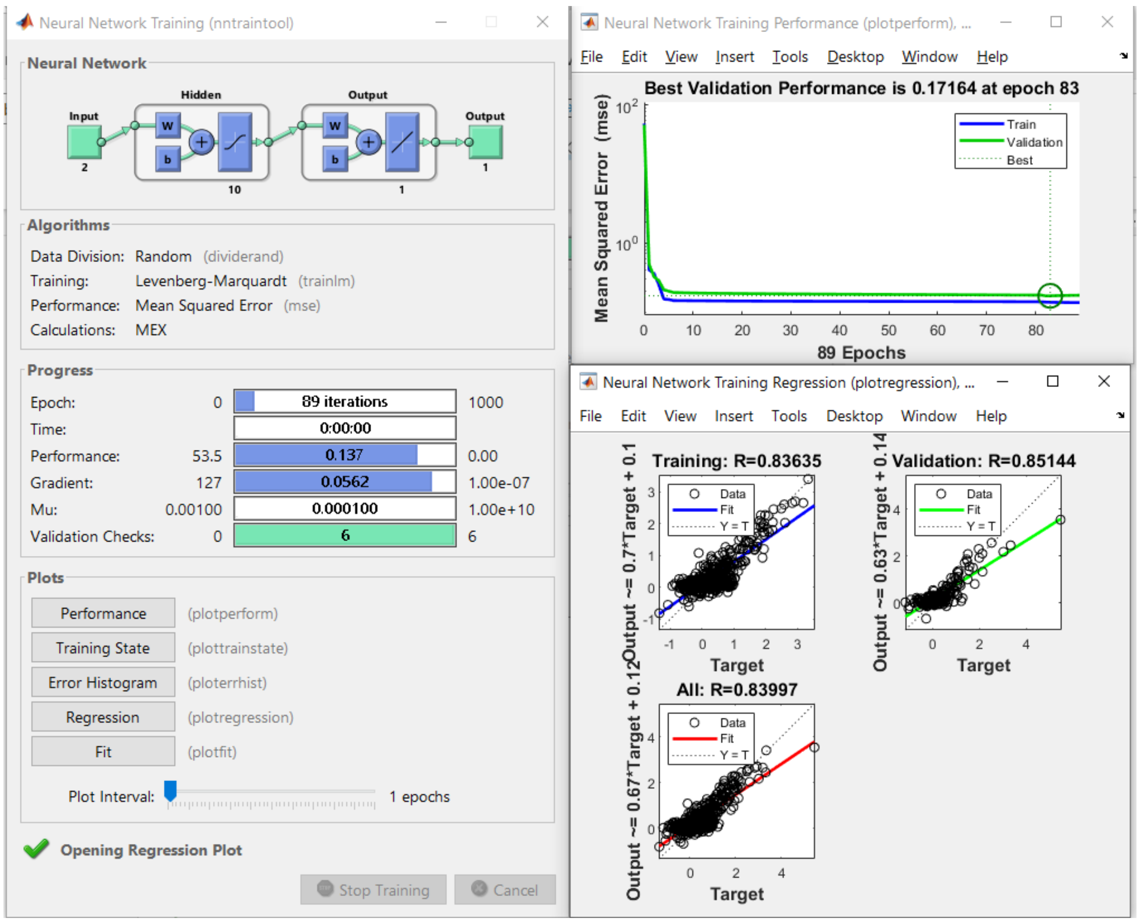
```
%% Create a feedforward neural network
hiddenLayerSize = 10;
net = fitnet(hiddenLayerSize);  % Regression network

% Set training/validation/test split
net.divideParam.trainRatio = 0.7;
net.divideParam.valRatio   = 0.3;
net.divideParam.testRatio  = 0.0;  % We will not use the internal test set

net.trainFcn = 'trainlm';           % Define training function
net.trainParam.showWindow = true;  % Show training window

[net, tr] = train(net, X, T);      % Train the network
```

Note, that within the code we can divide dataset into subsets internally (data we pass to the network will be splitted into subsets while the network is trained). We already have a *Testing* set prepared – so we are using this option only to craft training and validation subsets. Lets investigate the network we created. The result of training should look somewhat similar to one presented in Figure 1 (note that depending on matlab and toolbox version the graphics may be different, but similar information should be provided by them).


Neural Network Training (nntraintool)

Neural Network

Algorithms

Data Division: Random (dividerand)
Training: Levenberg-Marquardt (trainlm)
Performance: Mean Squared Error (mse)
Calculations: MEX

Progress

Epoch:	0	89 iterations	1000
Time:		0:00:00	
Performance:	53.5	0.137	0.00
Gradient:	127	0.0562	1.00e-07
Mu:	0.00100	0.000100	1.00e+10
Validation Checks:	0	6	6

Opening Regression Plot

Best Validation Performance is 0.17164 at epoch 83

Training: R=0.83635

Validation: R=0.85144

All: R=0.83997

Hidden
Output
Input
Output
W
b
W
b
2
10
1
1
Plots
Performance (plotperform)
Training State (plottrainstate)
Error Histogram (ploterrhist)
Regression (plotregression)
Fit (plotfit)
Plot Interval: 1 epochs
Stop Training
Cancel
Neural Network Training Performance (plotperform), ...
File Edit View Insert Tools Desktop Window Help
Mean Squared Error (mse)
89 Epochs
Train
Validation
Best
Neural Network Training Regression (plotregression), ...
File Edit View Insert Tools Desktop Window Help
Output ~= 0.7*Target + 0.1
Output ~= 0.63*Target + 0.14
Output ~= 0.67*Target + 0.12
Target
Data
Fit
Y = T

Figure 1 – Results of training in matlab GUI: nntraintool provides information about neural network shape (for us: 2 inputs, 10 neurons in hidden layer), other metaparameters and training progress, indicating stopping criterion that "fired" in green. You can open plots to look e.g. for training progress ("Performance") or for spread of results from ideal line (is there a skew for particular targets?)

Lets now check the results that we obtained. We can get the prediction from our trained network and compare it with one returned by polynomial model from laboratory 2.

```
PredictedValues = net(X);  % If we want to use our trained net to predict new data, we
do that here
MSE = mse(T,PredictedValues)

fprintf('MSE on all training+validation data: %.4f\n', MSE);

%% Plot predicted vs actual
figure;
plot(T, PredictedValues, '.')
xlabel('Actual Target')
ylabel('Predicted Output')
title('Neural Network Regression: Actual vs Predicted')
```

Note, however, that this is MSE for a training dataset. In order to calculate MSE for our external testing dataset, we need to load and pass through the network the *Testing* part of our data. Of course we don't want to include it in the *train* command – just using *net* command from the above code (highlighted in yellow) on new data should be enough.

Task 4.1: Using your **individual*** dataset for regression train the neural network using the structure given above. Then test your trained network on a *testing* subset of your data – and compare its performance with your polynomial regressor from laboratory 2. Show the results of the comparison using a bar graph.

Now lets do the same process for the classification problem that you were solving in scope of Laboratory 3. Similarly as before, we'll have to prepare our data for use by the neural network. We can do that using the following code:

```
% Load training and validation data
load('TR_DATA.mat');
load('VA_DATA.mat');

% Extract features and labels from training data
X_train = TR_DATA(2:3, :);   % Features (2×N)
T_train = TR_DATA(1, :);     % Class labels (1×N)

% Extract features and labels from validation data
X_val = VA_DATA(2:3, :);
T_val = VA_DATA(1, :);
```

Design and configuration of the network works similarly as before – the only difference is that we are using the *patternnet* instead of the *fitnet* – which tells Matlab that our network is supposed to recognize classes in the final layer, and that we are refraining from automatic division of training and validation datasets. Note also that we are changing the training algorithm in our pattern recognition task:

```
% Create a MLP network:
hiddenLayerSize = 10;
net = patternnet(hiddenLayerSize);

% Training metaparameters:
net.divideFcn = 'dividetrain';  % For manual control of validation subset
net.trainFcn = 'trainscg';      % Scaled conjugate gradient training method
net.trainParam.showWindow = true;

% Train the network
[net, tr] = train(net, X_train, T_train);
```

And finally we are able to calculate network performance on any dataset. Lets start with the training one:

```
% Predict and compute training accuracy
Y_train = net(X_train);             % Network output: probabilities
Y_predtrain = round(Y_train);       % Convert to binary class predictions
trainaccuracy(iteration) = sum(Y_predtrain == T_train) / length(T_train);
fprintf('Training Accuracy: %.2f%%\n', trainaccuracy(iteration) * 100);
```

We of course want to know not only training accuracy but also result of validation (based on VA_DATA subset, which we've loaded above and didn't use yet).

Task 4.2: Using your **individual*** dataset for classification from laboratory 3 train the neural network using the TR_DATA subset. Then check its performance using TR_DATA and VA_DATA subsets. Compare the solution with multiline 1+1-based and multiline gradient-based solutions developed in scope of Laboratory 3. Show the results of these comparisons using a bar graph.

OK – now the question arises: Do we actually have the best possible network for the task? Here we start thinking about metaparameters – training algorithms, network structure, ending conditions for optimization and so on. Lets start simple by asking what will happen if we change number of neurons in our classification problem. Lets do it systematically by organizing our tests in the following loop. Note that I have prepared a vector *hlsConfigurations* with preset configurations that I want to test (marked in cyan). The reason for that is that while interesting stuff may happen if we change neuron number from 2 to 3 (so I don't want to miss on that), I don't really think it is important to check the difference between, say, 43 and 44 neurons:

```
hlsConfigurations = [1,2,3,6,10,15,20,30,40,60,100,200];

for ConfigurationNumber = 1:length(hlsConfigurations)
        hiddenLayerSize = hlsConfigurations(ConfigurationNumber);
        % Create a MLP network:
        net = patternnet(hiddenLayerSize);

% ... - And here the rest of the code for testing the network
```

The picture that I got is presented in Figure 2.

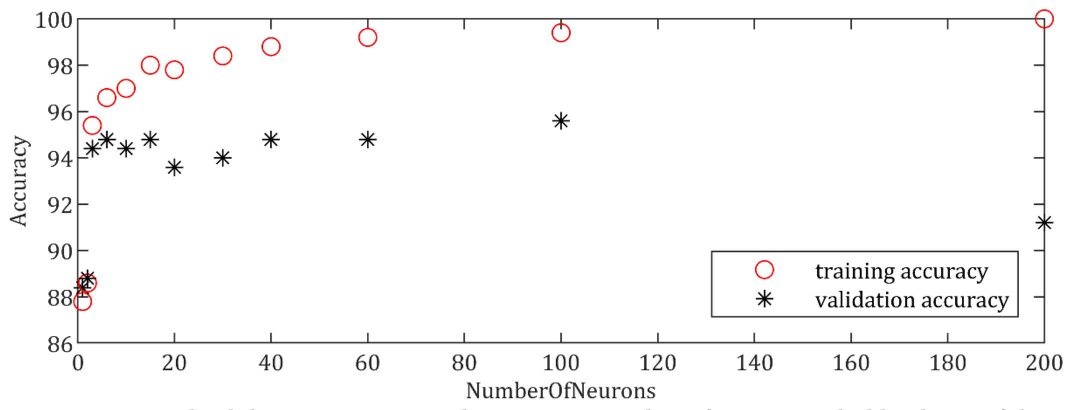


Figure 2 – Training and validation accuracy with respect to number of neurons in hidden layer of the network.

The interesting fact is that, apparently, validation accuracy lowers significantly for bigger networks despite the fact that training accuracy keeps climbing. What we also see is the fact that the relationship appears to be noisy – there is a dip in validation accuracy for 20 and 30-neuron networks which may or may not be caused by actual network properties. Lets repeat this test statistically to make sure. I decided to run the test 10 times for each network size and then store my results in a new *Statistics* structure. The number assigned to *Statistics* vector tracks the number of the tested configuration. Partial results are stored in *TrainResults* and *ValResults* vectors. After that I can show the results using the following code, ending with the result displayed in Figure 3. Note the green fragment of the code. It serves to provide manual ticks on the x axis (corresponding to actual numbers of tested neurons). If you want to generate similar visualizations, you'll need to adjust this code to be compatible with your data structure.

```
figure;
for configurationNumber = 1:length(Statistics)
    plot(configurationNumber, 100 * Statistics(configurationNumber).TrainResults, 'or'); hold on
    plot(configurationNumber, 100 * Statistics(configurationNumber).ValResults, '*k'); hold on
end

xlabel('Number of Neurons')
ylabel('Accuracy (%)')
xticks(1:length(Statistics))
xticklabels(string([Statistics(:).NumberOfNeurons]))
```

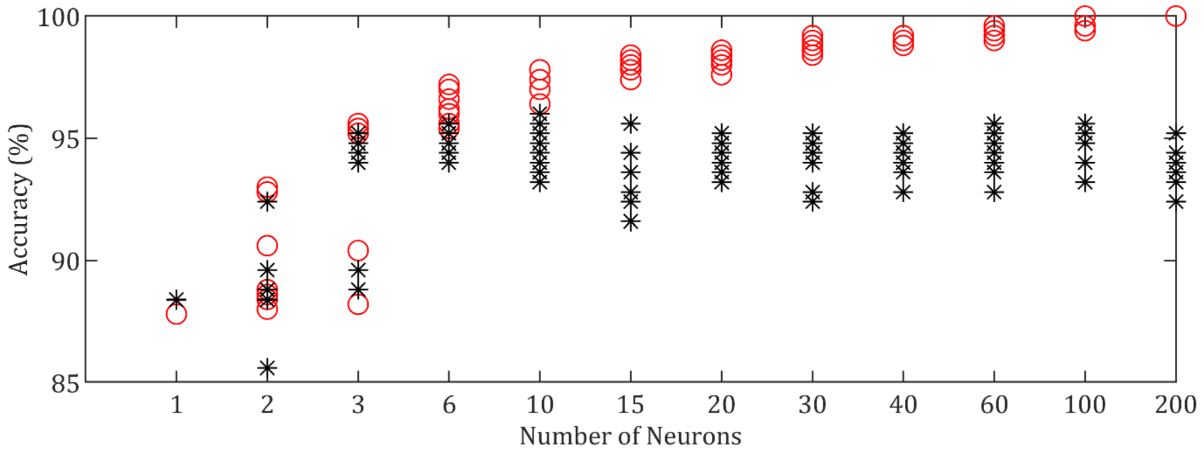


Figure 3 – Training and validation accuracy with respect to number of neurons in hidden layer of the network – results of statistical evaluation.

Task 4.3: Using your **individual*** dataset for classification from laboratory 3 evaluate statistically training and validation performance for increasing number of neurons, starting from 1, up to 200. Show the results in a graphical form. Then, repeat this task for the regression task. Save all the results to your data structure.

If the task 4.3 was done properly, you probably noticed something interesting: Our classification network appears to overfit easily while the regression net feels much more robust and able to generalize better. It is because of the fact that we did not actually trained the classifier properly. By designing training and validation dataset manually we forced the algorithm to not use validation checks as a stopping criterion. We can solve this problem in two ways. One: we can actually prepare another validation dataset by cutting part of training data (which will reduce amount of samples we can use for training but is simple to code):

```
        % Set custom division
        net.divideFcn = 'dividerand';
        net.divideParam.trainRatio = 0.7;
        net.divideParam.valRatio   = 0.3;
        net.divideParam.testRatio  = 0.0;
```

Or better: we can merge training and validation datasets and then force the network to not mix them up randomly but instead use them in order we provided:

```
% Combine data
X_all = [X_train, X_val];
T_all = [T_train, T_val];

% We will not be using the following division function now:
% net.divideFcn = 'dividetrain';

% Define indices (1 = train, 2 = validation, 3 = test)
trainInd = 1:size(X_train, 2);
valInd   = size(X_train, 2)+1 : size(X_train, 2)+size(X_val, 2);
testInd  = [];  % We again have no test data defined here

% Set custom division
net.divideFcn = 'divideind';
net.divideParam.trainInd = trainInd;
net.divideParam.valInd   = valInd;
net.divideParam.testInd  = testInd;
```

Note that in order for the second approach to work, you'll also need to pass the new dataset to the network during training:

```
[net, tr] = train(net, X_all, T_all);
```

Task 4.4: Update the code for classification network evaluation allowing the network to use validation dataset as a stopping criterion. Check if it modifies network's statistical performance (whether the overfitting is still present)

Task 4.5: Evaluate statistically the performance of the classification network for two different training algorithms: *trainscg* and *trainlm* – with respect to network size. Let the size vary in large range, so your testing will include networks from 10 to at least 300 neurons in hidden layer. Check both validation accuracy and training time for both algorithms and then show the results in a graphical form.

Hint: Time of training can be extracted using the **`tr.time(end)`** *value – this is the time of the last training iteration.*

Task 4.6: Evaluate importance of the network depth: Design a research goal and then an experiment that will tell you what are the implications of using network with higher number of hidden layers. Consult your choice of experiment with the teacher before you start experimentation.

Hint: There is just too many things to check at this stage so you will have to pick something. Would you like to check if number of layers' influence is similar across problems (regression and classification)? Does it influence training time? Does it allow for better accuracy or does it make the training harder? Is it similar or different across different training algorithms? The goal is to think about a question that interests you and then design a procedure that will allow you to find out what the answer is.

Additional Tasks

Task 4.7: Look into Matlab documentation in search of other neural network training algorithms possible for application in a classification tasks and then find information about them on the internet (you can use ChatGPT to this end) to compare them with *trainlm* and *trainscg*. Then pick two candidates which feel different to the algorithms we already tested. Design experiments that will either prove or disprove the properties you learned about them and then perform them, drawing conclusions.

Task 4.8: Create a subset of your training data by randomly selecting 10%, 30%, 60% and 100% of the original samples. Then do the same with validation subset. Then train the same network for each combination of training and validation subset. Repeat results so you are able to evaluate them statistically. What is more important: size of the training or size of the validation dataset? What do you observe about learning behavior, overfitting, and stability in low-data regimes?

Task 4.9: Try repeating the same network training setup (same architecture, same data, same algorithm) multiple times. Do the results vary between runs? Why? Then explicitly fix the network's initial weights using *net.initFcn* and/or *net.IW{}* using numbers from different ranges and repeat your experiments. How does random initialization affect the learning and outcome? Show the results graphically and draw conclusions about the importance of initialization in classification tasks.

Task 4.10 Using a network trained for 2-dimensional classification, evaluate the output across a dense grid of points covering the input space. Plot the network's predicted class for each grid point to visualize the decision boundary. Then repeat for networks with different sizes. How does model complexity influence the sharpness and shape of the decision boundary?

Task 4.11 Disable the default input normalization by setting *net.inputs{1}.processFcns = {}* before training. Then train your network and record its performance. Next, manually normalize the data yourself (e.g., standardize to zero mean and unit variance), and compare training speed and validation accuracy. What does this tell you about the importance of input preprocessing?