Lab 1:
Supervised Learning and Multilayer Perceptrons

2021-02-03

Introduction

This is the first of four labs in this course. You are expected to work as a team with the lab, meaning that each step is performed as a group. It is therefore *not* allowed to split up the work. Start by skimming through the instructions to get a feel of what will happen and then work through it from beginning to end. Make observations and take notes and it is usually a good idea to try and formulate the answers to questions right away. However, sometimes it is also useful to go back and revisit questions if you clear out a previous misconception.

As of covid-19 regulations, the course as a whole is given remotely. As such the lab hours are conducted over Zoom. See Studium for detatils about date and time, as well as the Zoom link.

Goals

The goal of this assignment is to learn how to create and train a multilayer perceptron (MLP) for classification and function approximation. For this you will be using MATLAB. You will learn how an MLP translates inputs into outputs, and gain insight into the issues of generalization and hidden layer dimensioning.

Preparations

Read through these instructions carefully in order to understand what is expected of you. Read your lecture notes from lectures 4 and 5. The most relevant parts of the book are in chapters 3 and 7. See the reading instructions for details.

You can play around with the demos included in MATLAB's neural network toolbox. Especially the demos `nnd2n1`, `nnd2n2` and `nnd4db` can be useful to better understand the functioning of single artificial neurons. (To run a demo, simply type in its name in MATLAB's command window.)

Launch MATLAB by navigating to:

Applications → **Education** → **MATLAB**

As MATLAB is run on central servers, the first time you start the application, you will get a terminal window asking you to accept to connect to that server. Type "`yes`". Your password might be requested, which is standard procedure for you to connect to the server. Enter your account password and press `Enter`.

The default editor mode for the Linux version of MATLAB is set to 'Emacs' by default. Basically, this manifests by having different specific shortcuts for, e.g., cutting and pasting text or saving files. If you are used to Emacs, this is no problem, but if you would prefer keyboard shortcuts common in, e.g., Windows, this may be changed by selecting the **HOME** tab and click **Preferences**. In the left pane, go to:

MATLAB → **Keyboard** → **Shortcuts**

... and select `Windows Default Set` instead of `Emacs` in the topmost **Active settings** pane. Click **OK**.

Report

Hand in a report according to the instructions found on Studium. It should contain answers to all questions, and all requested plots. Note that some questions are marked with ★. These questions are extra important as they reflect basic understanding of the topic. They are also indicative of what exam questions may look like.

Files

You will need the files `plot_xor.m` and `housing.mat`, which you can download from the corresponding lab assignment page on Studium (where you found this document). Place them in your working directory.

Automation

It is strongly encouraged to learn how to write simple MATLAB scripts which allows you to repeat experiments easily. During the assignments it is often fruitful (and many times you will be required) to run an experiments many times to make sure the results you are getting are not statistical outliers. Writing MATLAB scripts, allowing for repeated experiments with the stroke of a key, is not complicated, feel free to ask the TAs for help. Also note that there is a file named `conduct_n_experiments.m` available in the lab page on Studium. This

file demonstrates how to conduct several experiments after each other and store the resulting, trained networks in a separate vector for further analysis. Feel free to use and play around!

Task 1: Simple classification: XOR

In this task you will train an MLP to implement the exclusive or (XOR) boolean function. The XOR problem is not a realistic example of a problem that would normally be solved using neural networks. It is, however, interesting as it is a striking example of a problem that is not linearly separable, showing the need for more than one layer of neurons. It is also easy to analyse, and is therefore a suitable choice when we want to study the inner workings of an MLP that is trained to implement it.

1.1 Create the data

Begin by creating the input (`p`) and target (`t`) data that should be used in the training.

```
p = [[0; 0] [0; 1] [1; 0] [1; 1]]
t = [0 1 1 0]
```

In MATLAB the training data is represented as matrices. The input data is a matrix of size $m_p \times n_p$ (containing n_p column vectors, each with m_p input values) and the target data is a matrix of size $m_t \times n_t$ (defined similarly). For this particular problem we have $m_p = 2$ (since there are two inputs), $m_t = 1$ (since there is a single output) and $n_p = n_t = 4$ (since there are four data points).

1.2 Create the network

We now want to create an MLP and train it on this data. From the problem definition it is clear that the MLP should have two input nodes and one output node. We will also create a hidden layer containing two nodes, resulting in a network like the one in Figure 1.

We can create an MLP in MATLAB with the `newff` command. To create such a network, called `net`, run the following.

```
net = newff(p, t, [2], {'tansig' 'logsig'}, 'traingd', ...
```

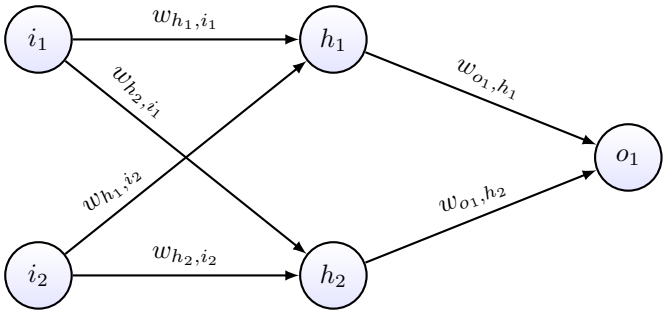

Figure 1: A 2-2-1 MLP.

```
            '', 'mse', {}, {}, '')
```

Note that the three dots (...) are there to tell MATLAB that the command continues on the next line, you should ignore them if you enter the command as a single line. The arguments to `newff` are, in order

1. A matrix of representative input data, which MATLAB uses to automatically set the correct number of input nodes, as well as finding a range for the initial values of weights.
2. A matrix of representative target data, used similarly to the above.
3. A vector containing the sizes of the hidden layers. Since we want only one hidden layer (with size 2), it contains only a 2.
4. A cell array containing the transfer (activation) functions of the different layers (MATLAB does not count the input nodes as a layer). We use the hyperbolic tangent (`tansig`) as the transfer function for the hidden layer, and the logistic function (`logsig`) as the transfer function for the output layer.
5. A training algorithm to use when training the network. We use gradient descent training (`traingd`), which is MATLAB's name for batch backpropagation.
6. A learning algorithm to use if the training algorithm does not specify one (the distinction between a training algorithm and a learning algorithm is MATLAB specific, and we do not have to care about learning algorithms for this lab). The training algorithm `traingd` specifies a learning policy, so we can leave this empty.
7. An error function to minimize. We use mean squared error (`mse`).
8. A cell array of input pre-processing functions. We do not want any, so we leave it empty. (Note that we have to explicitly set it to empty, if we simply omit the argument MATLAB will use several default processing functions.) above.

9. A function for automatically dividing training data into training, validation and test sets. We don't want such a function, so we leave it empty (also this one must explicitly be set to empty).

For more details, run

```
help newff
```

Documentation for the whole neural network toolbox can be found by using the menu **Help** → **Documentation** and then selecting **Neural Network Toolbox**.

1.3 Train the network

A network is trained by using the `train` function, which takes as arguments the network to be trained, the input data and the target data. When the training is complete the `train` function returns the trained network and some training statistics. Before a network is trained it should be initialised with the `init` function. (Note that `init` and `train` return *new* networks, i.e., they are non-destructive functions.) To do both initialisation and training in one go, run

```
net = init(net); [trained_net, stats] = train(net, p, t);
```

This will open up a new training window, where information about the training is displayed as it is being performed. The information includes the number of epochs trained and the performance (i.e., the value of the error function). You can click on the **Performance** button to open up a graph of the error as a function of the epoch number (logarithmic scale). Note that having such a graph open during training will slow MATLAB down, but you can increase the plot interval to mitigate the slowdown if you want the performance plot displayed during training.

Notice that the training continues for 1000 iterations (epochs) and then stops. This is because the default setting when training with batch backpropagation (`traingd`) in MATLAB is to train for at most 1000 epochs. If you rerun the training a few times (with the command above, and remember to reinitialise each time) you will notice that 1000 epochs are usually not enough to make the error function converge. Thus, you will have to change some settings to achieve satisfactory training results. The training parameters for the network's training algorithm are stored in the structure `net.trainParam` (if `net` is the name of your network). Type `net.trainParam` in the command window to see the parameters and their current values. You can change all of the parameters by writing to the variables in this structure. Of most interest right now is `net.trainParam.epochs`

(the maximum number of epochs to train) and `net.trainParam.lr` (the learning rate, which is called η in the text book). Also `net.trainParam.min_grad` could be of interest; it gives the minimum gradient of the error function after which MATLAB will stop the training (if it is reached before the maximum number of epochs have passed). It should not happen often during training with this data set that the default minimum gradient is reached, but you can set it to 0 to make sure that MATLAB never stops until the given number of epochs have passed.

You can visualise the function implemented by the (trained or untrained) network with the `plot_xor` function. It will display the output of the network (colour coded) for all combinations of the two input values, in the domain $[-0.5, 1.5]^2$. It will also display the activations of the two hidden nodes, and for reference it will draw markers at the positions (0,0), (0,1), (1,0) and (1,1).

```
plot_xor(trained_net)
```

You can also produce the output of the network with the `sim` function. Feeding an input vector (or a vector of input vectors) to `sim` will return the network's output values for those inputs.

```
sim(trained_net, [0; 0]) % Returns one output.
sim(trained_net, p)      % Returns a vector of outputs.
```

Now train the network with some different learning rates, ranging between 0.01 and 20. For each learning rate, adapt the number of epochs to some suitable number for the training to converge (i.e., flatten out at some value) most of the times. When a training results in an error value close to zero, use the `plot_xor` function to convince yourself that the network actually implements the XOR function.

You can create a figure containing plots of the error as a function of epoch from several different training runs in the following way. First create a new figure.

```
figure    % Create a new figure.
ax = axes % Get a handle to the figure's axes.
hold on   % Set the figure to not overwrite old plots.
grid on   % Turn on the grid.
```

Then after each training, plot a line for the performance of this training session on the figure (where `stats` is the training statistics structure returned by `train`).

```
plot(ax, stats.perf)
```

Note that this figure will have a linear scale, as opposed to the performance plot that can be seen during training. When you want a new figure, create it in the same way as above.

Plot 1: *Include three figures, each one should contain plotted performance statistics for 10 training sessions (i.e., each graph should have 10 lines). The training should be done with learning rates of 0.1, 2 and 20, respectively (one learning rate per figure). Make sure that it is clear which figure is which in the report, for an example by adding a title to each using the command* `title(ax, 'lr = ?')`.

Question 1: *As you have seen, the error (performance) sometimes converges to a value greater than zero. Explain why this happens.*

Question 2 ★: *How does the learning rate affect the training of the network? What are the likely effects of using a too low value? What are the likely effects of using a too high value?*

It is now time to look closer at how the MLP implements the learnt function. If you have studied the outputs of the nodes in the network with `plot_xor`, you have hopefully seen several different characteristic solutions by now. If not, train the network with suitable parameters until you have seen at least two very different solutions to the problem (different weight settings with which the MLP solves the problem).

Plot 2: *Include the figures drawn by* `plot_xor` *for two different solutions you found to the* XOR *problem. Motivate why they are solutions.*

Question 3: *Why are the activations of the hidden nodes in the range [-1,1], but the output of the network in the range [0,1]? Hint: remember how the network was created.*

Question 4: *Why does the training not always end up with the same solution, even when the same training parameters are used?*

We can interpret the nodes as implementing boolean functions by considering their (real-valued) output as either 0 or 1, depending on which side of the discriminant we end up on. Denote the boolean functions thus formed by the

hidden nodes H_1 and H_2, respectively. Denote the boolean function formed by the output node O. Denote the input values I_1 and I_2.

1.4 Train with resilient backpropagation

You have now trained an MLP to implement the XOR problem with (standard) batch backpropagation. You should now train the network with another training algorithm called resilient backpropagation (Rprop). To do so, create a new network with this training algorithm set.

```
net = newff(p, t, [2], {'tansig' 'logsig'}, 'trainrp', ...
            '', 'mse', {}, {}, '')
```

You may notice that we replaced `traingd` (which is MATLAB's name for batch backpropagation) with `trainrp`, which is its name for Rprop.

Rprop is one of several training algorithms that were designed with the goal of faster training. As opposed to backpropagation (which uses gradient descent), Rprop uses only the signs of the partial derivates, not their actual value. As long as the derivate keeps the same sign, the weight changes grow larger and larger, but as soon as a sign change is noticed, the weight changes are decreased.

Rprop has a different set of training parameters than backpropagation, you can access them in the structure `net.trainParam`. Of most interest are the following parameters.

- `delta0`: the original size of the weight changes (at epoch 0).
- `deltamax`: the maximum allowed weight change.
- `delt_inc`: the factor for increasing the weight change when the sign of the partial derivate has not changed.
- `delt_dec`: the factor for decreasing the weight change when the sign of the partial derivate has changed.

Play around with some different training parameters until you find some settings that you are satisfied with. (Rprop was designed to not be very sensitive to the choice of `delta0` and `deltamax`. It can on the other hand be quite sensitive to the choice of `delt_inc` and `delt_dec`, the original intent was that user should never have to change these from the defaults. You will likely not have to change these to get good results, but it can be enlightening to do so anyway in order to observe the effects.)

Plot 3: *Include a figure of performance statistics for 10 training sessions under the same conditions using Rprop, in the same way as before. What parameters did you use?*

Question 5: *What are the most important differences that you can observe between the results produced by Rprop and backpropagation? Also look at the function implemented by the network using* `plot_xor`*, and see if you notice anything different when the network is trained using Rprop. Which of the two training algorithms do you think is most suitable for this problem, and why?*

Task 2: Simple function approximation

In the previous task you trained an MLP to solve a classification problem (with a discrete set of targets). In this task you will train an MLP to solve a function approximation problem (with a continuous set of targets). The function that you will train the network to approximate is $f(x) = \sin(x) \cdot \sin(5x)$ on the interval $[0, \pi]$, which is plotted in Figure 2.

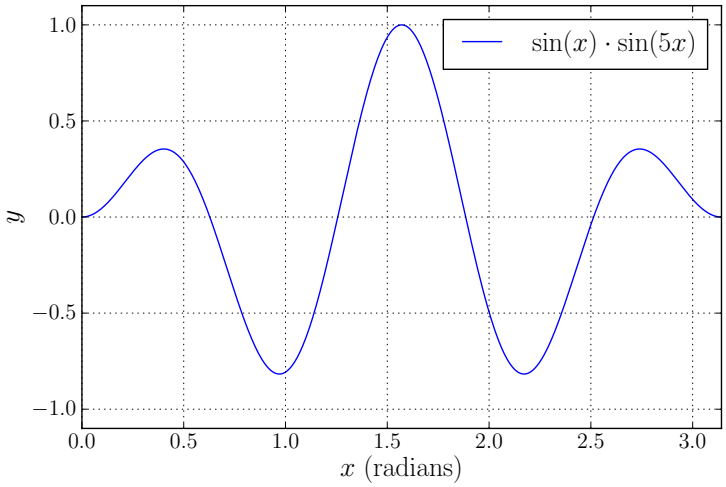

Figure 2: The function to approximate.

We will not assume that the function is known or that we can evaluate it at all points. Instead we will train it using only a few data points from the function. Create 10 (input) data points evenly spread out in the range [0,π].

```
p = linspace(0, pi, 10)
```

Then create the target values for these points

```
t = sin(p) .* sin(5*p)
```

You can visualize the training data by plotting it and comparing to the function plot above.

```
plot(p, t, 'o')
```

Now create the network to train, using the function `newfit`. (The function `newfit` is the same as `newff` used before, except that it adds a button to the training window to graphically display curve fitting. The network and the training work exactly the same as before.)

```
net = newfit(p, t, [?], {'tansig' 'purelin'}, 'traingd', ...
             '', 'mse', {}, {}, '')
```

Notice that we used `purelin` (the identity function) as the transfer function for the output layer, this is a common choice for function approximation problems where we want the network to be able to produce any output. You may also notice that the number of nodes in the hidden layer is unspecified, this is for you to set.

Create some networks with different number of hidden nodes (from 1 to 20), and train them using backpropagation on the data that you created. You will have to figure out suitable parameter settings (number of epochs and learning rate) for the training in each case. Note that the parameter settings may have to be varied with the number of hidden nodes. In the training window you can press the **Fit** button to graphically display the function that the network implements in the range covered by the input data. Note that it is hard to get any reasonable insight from too few experiments – you should therefore run at least 10 experiments per setup in order to answer the following questions.

Plot 4: *Include figures of the function (as given by the* **Fit** *button) implemented by networks with 3, 6, 10 and 20 hidden nodes, after typical training sessions with good parameter values. Make sure it is clear which figure is which, and what training parameters you used. Also include the* MSE *of the trained networks on the training data.*

Question 6: *For what number of hidden nodes do you usually get the lowest training errors (smallest* MSE*)? What seems to be the general relation between the number of hidden nodes and training error?*

Question 7: *What number of hidden nodes usually gives the best approximation to the function* $f(x) = \sin(x) \cdot \sin(5x)$*, in your opinion? Motivate your answer.*

Question 8 ★: *Function approximation suffers both when you have few hidden nodes and when you have too many nodes. Explain, respectively, why that is.*

Question 9: *If you knew beforehand that the function to approximate looks something like the one in Figure 2, could you have used that knowledge to make an educated guess at a suitable number of hidden nodes to use (without trial-and-error)? Explain how you would come up with that number. Is this number close to the number that you found to be best by trial-and-error?*

Now train the network with Rprop instead of backpropagation. Use some suitable number of hidden nodes and training parameters.

Question 10: *What differences can you observe in the results compared to those achieved with backpropagation? Based on your observations, which of the two training algorithms do you think is most suitable for this problem? Motivate your answer.*

Task 3: Classification of wine data

In this task you will again solve a classification problem, but a much larger one than the XOR problem. The data that you will use contain chemical analysis results from 178 different wines that were all produced from grapes grown in the same region in Italy. The grapes used for the wines were of three different varieties. Your task is to train an MLP to classify wines into one of three classes depending on which one of the three grape types it was made from.

The data consist of 178 samples in total, with each sample belonging to one of the three classes. The input data is 13-dimensional, with 13 numerical values of different metrics obtained through the chemical analysis. Load the data using the command

```
load wine_dataset
```

This will create two matrices, named `wineInputs` and `wineTargets`, containing the data. Look at these matrices and convince yourselves that you understand their structure and how inputs map to validation outputs.

Now train an MLP with one layer of hidden nodes on this data using either backpropagation or Rprop, and try to achieve as good results as possible. Try to find some settings for the following that enables you to get reasonably good results from the training (only a few percent of the wines misclassified):

- The training algorithm to use.
- The parameters for the training algorithm, if different from the default.
- The number of hidden nodes.
- The transfer functions for the hidden and output layers.

Do *not* use any of MATLAB's built-in functions for automatic pre or post-processing of the data (i.e., create the network using `newff` in the same way as before).

You can evaluate the results by looking at the error (performance plot), or by plotting a *confusion matrix*. A confusion matrix will display which class that each data point was classified to (the y-axis) and which class that it should have been classified to (x-axis), as well as percentages of correctly classified data points. Plot the confusion matrix using this command:

```
plotconfusion(wineTargets, sim(trained_net, wineInputs))
```

The arguments will provide `plotconfusion` with the targets and the outputs of the network.

Question 11: *What settings did you use to get good results? What was the smallest number of hidden nodes that you needed? Is this number reasonable with respect to the size of the dataset? Approximately how many percent of the wines are placed in the right class after training this network?*

You probably needed quite a few hidden nodes to get good results with this data, which might indicate that the problem is quite complicated and that the classes are somewhat difficult to separate. This is not necessarily the case though. Look again at the data, particularly the `wineInputs`, and try to get a feeling for the distribution of values in it (note that MATLAB, upon printing the matrix, may display the values as multiples of 1000). Indeed, there are input values ranging from less than 0.2 to more than 1000 in it. This big difference in variability can make training harder, as some get relatively larger impact on

training than others, even though they are perhaps not more important for the classification.

A possible solution to this problem is to use *normalization* of the data, which scales it into some form where the differences between values in different dimensions are smaller. You will use a normalization that linearly scales and shifts each input dimension separately, so that the minimum value in the data in each dimension becomes -1, and the maximum value becomes 1. Use the function `mapminmax` for this.

```
normWineInputs = mapminmax(wineInputs)
```

Look at the normalized matrix to see that it looks as expected. Then create an MLP and train it using this data instead.

Question 12: *Did the normalization have any significant impact on the results of the training? How many hidden nodes did you need now in order to get good results?*

Question 13 ★: *In general, will normalization always help in training an* MLP*? What possible dangers can normalization in this way pose to a supervised learning problem?*

Task 4: Approximating house prices

The last task is to train an MLP to implement a function that estimates house prices in Boston based on some statistics about the area that the house is situated in. The data that you will use contain 506 samples, the statistics as the input data, and the house prices as output data. The input data is 13-dimensional, and contains the following fields.

1. Per capita crime rate by town.
2. Proportion of residential land zoned for lots over 25,000 sq.ft.
3. Proportion of non-retail business acres per town.
4. 1 if tract bounds Charles river, 0 otherwise.
5. Nitric oxides concentration (parts per 10 million).
6. Average number of rooms per dwelling.
7. Proportion of owner-occupied units built prior to 1940.

8. Weighted distances to five Boston employment centres.
9. Index of accessibility to radial highways.
10. Full-value property-tax rate per $10,000.
11. Pupil-teacher ratio by town.
12. $1000(Bk - 0.63)^2$, where Bk is the proportion of blacks by town.
13. Percent lower status of the population.

The target data is the median value of owner-occupied homes in $1000's. The data was collected from Boston suburbs in 1978 (and is thus outdated).

Load the data into MATLAB by double-clicking the `housing.mat` file (which should be in your working directory). This will create the matrices `houseInputs` and `houseTargets`. Normalize the input data as in the previous task.

```
normHouseInputs = mapminmax(houseInputs)
```

For this task we will split the data into three sets: a *training set*, a *validation set* and a *test set*. The training set will be used to train the network as before. The validation and test sets are not immediately involved in the training, but are used as a measure of how well the network has been trained. Specifically, they are used to determine how well the network behaves on inputs that it didn't see during the training, i.e., how well it generalizes.

We can tell MATLAB to do this splitting automatically for us, by specifying a splitting function when we create a network. You will use the function `dividerand` which randomly selects data points for each of the three sets (by default it selects 60% of the data points for the training set, and 20% for each of the other two sets). This function is specified as the last argument to `newff`.

```
net = newff(normHouseInputs, houseTargets, [20], ...
            {'tansig' 'purelin'}, 'trainrp', '', ...
            'mse', {}, {}, 'dividerand');
```

Create an MLP using Rprop, with one hidden layer of 20 nodes, and with `tansig` and `purelin` as the transfer functions for the hidden and the output layers, respectively.

By default MATLAB will stop training as soon as the error on the validation set has been increasing for 6 epochs in a row (which can happen very quickly for this dataset). Avoid this early stopping by setting the training parameter `max_fail` to a higher number.

```
net.trainParam.max_fail = 1000
```

Also set `min_grad` to 0 and `epochs` to at least 5000.

Now train this network with Rprop a some number of times – at least 10 – and remember to reinitialize between every training session (`dividerand` will pick new random sets every time you train the network). Study the performance plot for each training session.

Plot 5: *Include the performance plot from one of the training sessions.*

Question 14 ★: *As the training goes along, the three error curves usually behave differently. Describe these trends, and explain why the curves behave in such a way.*

Question 15: *In light of your answer to the question above, what should one have in mind (in terms of training epochs) when training a neural network?*

Task 5: Wrapping up

You have now learned how to train multilayer perceptrons to solve classification and function approximation problems. The problems solved have ranged from very simple to near-real world problems.

Question 16: *Propose a real-world problem that you consider interesting and that you believe could be solved using supervised learning of neural networks. Explain in a few sentences how the network would be used, and what kind of data that it would be trained on.*