Artificial Neural Networks and Deep Architectures,
DD2437

Lab assignment 1

Learning and generalisation in feed-forward networks —
from perceptron learning to backprop

1 Introduction

This exercise is concerned with supervised (error-based) learning approaches for feed-forward neural networks, both single-layer and multi-layer perceptron.

1.1 Aim and objectives

After completion of the lab assignment, you should be able to

- design and apply networks in *classification*, *function approximation* and *generalisation* tasks
- identify key limitations of single-layer networks
- configure and monitor the behaviour of learning algorithms for single- and multi-layer perceptrons networks
- recognise risks associated with backpropagation and minimise them for robust learning of multi-layer perceptrons.

1.2 Scope

In this lab you will implement single- and multi-layer perceptrons with focus on the supervised learning algorithms. You will then study their properties by means of simulations. Since the calculations are naturally formulated in terms of vectors and matrices, the exercise was originally conceived with MATLAB[1] in mind. However, you are free to choose your own programming/scripting language, environment etc. In the first part you will be asked to develop all the code

[1] It is also possible to use Octave, the free version of MATLAB.

from scratch whereas in the second part you can use one of the recommended libraries, i.e. NN toolbox in MATLAB, scikit-learn in Python or TensorFlow), to examine more advanced aspects of training multi-layer perceptrons with backpropagation. If you prefer to exploit other libraries, software for the second part of the assignment, please let us know in advance.

In the first part, the focus will be on two learning algorithms: the *Delta rule* (for a single-layer perceptron) and the *generalised Delta rule* for two-layer perceptron. The generalised Delta rule is also known as the *error backpropagation algorithm* or simply "*backprop*". This is one of the most common and generic supervised learning algorithms for neural networks and it stems from the concept of gradient descent. In this exercise you will have the opportunity to use it for *classification*, *data compression*, and *function approximation*.

In the second part of the lab assignment, you will work with multi-layer perceptrons to solve the problem of chaotic time series prediction. In this task you will design, train, validate (including model selection) and evaluate your neural network with the ambition to deliver a robust solution with good generalisation capabilities. Since you will have to rely on more advanced features of neural network training and evaluation, you will be asked to exploit the existing libraries (as mentioned above, NN toolbox in MATLAB, sckit-learn in Python and TensorFlow with interfaces such as Keras, for example, are recommended; there is a lot of solid documentation available to familiarise yourselves wth these tools).

2 Background

2.1 Data Representation

The data can be effectively represented in matrices (collection of vectors). Since this is a supervised learning approach, our traning data should consist of input patterns (vectors) and the associated output patterns, often called labels (e.g., scalar values for classification and regression). There are two options to perform training - sequential on a sample-by-sample basis and batch. In this lab we first focus on *batch* learning. This means that all patterns in the training set will be used as a whole at the same time instead of stepping through them one by one and updating weights successively for each sample (input pattern with its associated label/output). Batch learning is better suited for a matrix representation and is significantly more effective given built-in functions for quick matrix operations in most programming/scripting languages. In batch learning, each use of the whole set of available training patterns is commonly referred to as an *epoch* and the entire training process involves many iterative epochs. By a suitable choice of representation, an epoch can be performed with just a few matrix operations.

Further, in problems where binary representation (0/1) is inherent, it is convenient sometimes and practical to rely instead on a symmetric ($-1/1$) representation of the patterns. Most importantly, you must make sure that your learning rule is compliant with the representation of your class labels/targets, as discussed in the lecture.

The input patterns (vectors) as well as their corresponding targets/labels (predominantly scalar values) can be represented as columns in two matrixes, X and T, respectively. With this representation, the XOR problem would for instance be described by

$$X = \begin{bmatrix} -1 & 1 & -1 & 1 \\ -1 & -1 & 1 & 1 \end{bmatrix} \qquad T = \begin{bmatrix} -1 & 1 & 1 & -1 \end{bmatrix}$$

If we read the matrices column-wise, we get the pattern $(-1, -1)$ to be associated with the output -1, and the pattern $(1, -1)$ with the output 1 etc.

A single-layer perceptron sums the weighted inputs, adds the *bias* term and produces the thresholded output. If you have more than one output, you have to have one set of weights for each output. These computations become very simple in matrix form. Make sure however that you account for the bias term by adding an extra input signal whose value always is one (and a weight corresponding to the bias value, as shown in the lecture). In the XOR example we thus get an extra column:

$$X_{\texttt{input}} = \begin{bmatrix} -1 & 1 & -1 & 1 \\ -1 & -1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} \qquad T = \begin{bmatrix} -1 & 1 & 1 & -1 \end{bmatrix}$$

The weights are stored in matrix W with as many columns as the dimensionality of the input patterns and with the number of rows matching the number of the outputs (dimensionality of the output). The network outputs corresponding to *all* input patterns can then be calculated by a simple matrix multiplication followed by thresholding at zero (since the bias has been already taken into account in the extra column of the weight matrix, provided that an extra entry with the constant value 1 was also included in the formation of the inputs, as explained earlier and discussed in the lecture). Learning with the Delta rule aims, with the representation selected, to find the weights W that give the best approximation:

$$W \cdot \begin{bmatrix} -1 & 1 & -1 & 1 \\ -1 & -1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} \approx \begin{bmatrix} -1 & 1 & 1 & -1 \end{bmatrix}$$

Unfortunately, the XOR problem is one of the classical problems that a single-layer perceptron cannot solve.

2.2 Implementation of the Delta rule

Store the training data in variables `patterns` and `targets`. As discussed above, add an extra row to the input patterns with ones corresponding to the extra bias terms in the weight matrix.

The Delta rule can be written as:

$$\Delta w_{j,i} = -\eta x_i \left(\sum_k w_{j,k} x_k - t_j \right)$$

where $\bar{x}$ is the input pattern, $\bar{t}$ is the wanted output pattern and $w_{j,i}$ is the connection x_i to t_j. This can be more compactly written in matrix form:

$$\Delta W = -\eta (W\bar{x} - \bar{t})\bar{x}^T$$

The formula above describes how the weights should be changed based on *one* training pattern (and its matching target label). To get the total weight change for the entire epoch, i.e. accounting for all training patterns, the weight update contributions from *all* patterns should be summed. Since we store the patterns as columns in X and T, we get this sum "for free" when the matrixes are multiplied. The total weight change from a whole epoch (batch mode since the learning contributions from each sample are summed and the weight update is performed at once) can therefore be written in this compact way:

$$\Delta W = -\eta (WX - T)X^T$$

Write your code so that the learning according to the formula above can be flexibly repeated `epochs` times (where 20 is a suitable number for a low-dimensional perceptron). Try to avoid loops as much as possible at the cost of powerful matrix operations (especially multiplications). Make sure that your code works for arbitrary sizes of input and output patterns and the number of training patterns. The step length or *learning rate* η should be set to some suitable small value like 0.001. Note: a common mistake when implementing this is to accidentally orient the matrixes wrongly so that columns and rows are interchanged. Make a sketch on paper where you write down the sizes of all components starting by the input and how the dimensionality propagatesto the weights to the output. This will be particularly important in the next part of the lab with a two-layer perceptron.

Before the learning phase can be executed, the weights must be initialised (have initial values assigned). The normal procedure is to start with small random numbers drawn from the normal distribution with zero mean. Construct a function to create an initial weight matrix by using random number generators built into programming/scripting languages. Note that the matrix must have matching dimensions.

2.3 Implementation of a two-layer perceptron

The focus is here on the implementation of the generalised Delta rule, more commonly known as backprop. You are going to use it in several different experiments, so it is worth making this general. Specifically, please make sure that the number of nodes in the hidden layer easily can be varied, for instance by changing the value of a global parameter. Also let the number of iterations and the step length be controlled in this way.

For multi-layer feed-forward networks, non-linear transfer functions should be used, often denoted φ. Commonly in classical multi-layer perceptrons (especially in shallow architectures) one chooses a function with the derivative that is simple to compute, e.g.

$$\varphi(x) = \frac{2}{1+e^{-x}} - 1$$

which has the derivative

$$\varphi'(x) = \frac{[1+\varphi(x)][1-\varphi(x)]}{2}$$

Note that it is advantageous to express the derivative in terms of $\varphi(x)$ itself since this value, used in the backward pass of the backpropagation learning algorithm, has to be computed anyway in the forward pass. This way we can save on the extra computations that would otherwise have to be performed to calculate derivatives (as discussed in the lecture, we get these derivatives in the scenario descibed above almost for free").

In backprop, each epoch consists of three parts. First, the so called forward pass is performed. In this the activities of the nodes are computed layer for layer. Secondly, there is the backward pass when an error signal δ is computed for each node. Since the δ-values depend on the δ-values in the following layers, this computation must start in the output layer and successively work its way (propagate) backwards layer by layer (thereby giving rise to the name - backpropagation).

2.3.1 The forward pass

Let x_i denote the activity level in node i in the output layer and let h_j be the activity in node j in the hidden layer. The output signal h_j now becomes

$$h_j = \varphi(h_j^*)$$

where h_j^* denotes the summed input signal to node node j, i.e.

$$h_j^* = \sum_i w_{j,i} x_i$$

Thereafter the same happens in the next layer, which eventually gives the final output pattern in the form of the vector $\bar{o}$.

$$o_k = \varphi(o_k^*)$$

$$o_k^* = \sum_j v_{k,j} h_j$$

Just as for the one layer perceptron, these computations can efficiently be written in matrix form.This also means that the computations are performed simultaneously over all the training patterns.

$$H = \varphi(WX)$$

$$O = \varphi(VH)$$

The transfer function φ should here be applied to all elements in the matrix, independently of each other.

We have so far omitted a small but important point, the so called *bias* term. For the algorithm to work, we must add an input signal in each layer which has the value one[2]. In our case the matrixes X and H must be extended with a row of ones at the end.

In MATLAB, the forward pass can be expressed like this:

```
hin = w * [patterns ; ones(1,ndata)];
hout = [2 ./ (1+exp(-hin)) - 1 ; ones(1,ndata)];

oin = v * hout;
out = 2 ./ (1+exp(-oin)) - 1;
```

Here we use the variables `hin` for H^*, `hout` for H, `oin` for O^* and `out` for O. Observe the use of the MATLAB operator `./` which denotes element wise division (in contrast to matrix division). The corresponding operator `.*` has been used already to get element wise multiplication.

2.3.2 The backward pass

The backward pass aims at calculating the generalised error signals δ that are used in the weight update. For the output layer nodes, δ is calculated as the error in output multiplied with the derivative of the transfer function (φ'), thus:

$$\delta_k^{(o)} = (o_k - t_k) \cdot \varphi'(o_k^*)$$

To compute δ in the next layer, one uses the previously calculated $\delta^{(o)}$:

$$\delta_j^{(h)} = \left(\sum_k v_{k,j} \delta_k^{(o)} \right) \cdot \varphi'(h_j^*)$$

[2] Some authors choose to let the bias term go outside the sum in the formulas, but this leads to the effect that it must be given special treatment all the way through the algorithm. Both the formulas and the implementation becomes simpler if you make sure that the extra input signal with the value 1 is added for each layer.

It should be expressed in matrix form:

$$\delta^{(o)} = (O - T) \odot \varphi'(O^*)$$

$$\delta^{(h)} = (V^T \delta^{(o)}) \odot \varphi'(H^*)$$

(where $\odot$ denotes element wise multiplication).

As an example, the corresponding MATLAB implementation could be coded in the following way:

```
delta_o = (out - targets) .* ((1 + out) .* (1 - out)) * 0.5;
delta_h = (v' * delta_o) .* ((1 + hout) .* (1 - hout)) * 0.5;
delta_h = delta_h(1:Nhidden, :);
```

The last line only has the purpose of removing the extra row that we previously added to the forward pass to take care of the bias term. We have here assumed that the variable Nhidden contains the number of nodes in the hidden layer.

2.3.3 Weight update

After the backward pass, it is now time to perform the actual weight update. The formula for the update is :

$$\Delta w_{j,i} = -\eta x_i \delta_j^{(h)}$$

$$\Delta v_{k,j} = -\eta h_j \delta_k^{(o)}$$

which we as usual convert to matrix form

$$\Delta W = -\eta \delta^{(h)} X^T$$

$$\Delta V = -\eta \delta^{(o)} H^T$$

As discussed in the lecture, to facilitate the convergence of our backprop learning algorithm a so-called *momentum* term can be added. This implies that the weights are not modified exclusively with with the update values from above, but with a moving average taking into account previous update(s) as well. This approach suppresses fast variations and allows the use of a larger learning rate. All in all, it balances out the contribution of the larger learning rate promoting faster convergence with the momentum slowing-down the process (exploration vs exploitation in the search through the weight space). A scalar factor α controls how much the old weight update vector contributes to the new update. A suitable value of α is often 0.9. The new update rule then becomes (in matrix form):

$$\Theta = \alpha\Theta - (1 - \alpha)\delta^{(h)} X^T$$

$$\Psi = \alpha\Psi - (1 - \alpha)\delta^{(o)} H^T$$

$$\Delta W = \eta \Theta$$
$$\Delta V = \eta \Psi$$

In MATLAB, it could be implemented as follows:

```
dw = (dw .* alpha) - (delta_h * pat’) .* (1-alpha);
dv = (dv .* alpha) - (delta_o * hout’) .* (1-alpha);
W = w + dw .* eta;
V = v + dv .* eta;
```

We have now gone through all the central parts of the algorithm. What remains is to put all parts together. Do not forget that the forward pass, the backward pass and the weight update should be performed for each epoch. For this, a `for`-loop can preferentially be used to successively get better and better weights in the iteration over epochs.

2.4 Monitoring the learning process and evaluation

Monitoring the process of learning for multi-layer perceptrons is not as simple as for a single-layer perceptron, which could be done by drawing the line of separation - decision boundary. For multi-layer networks we commonly rely on the output error as a probe for the advancement of the learning process. It is a common practice therefore to plot learning curves with the error estimated either by the mean square error or, in classification tasks, as the total number or proportion of misclassifications (there are also other error measures). Such *learning curves* illustrate the progress made over consecutive epochs (the error is usually estimated for the entire epoch, i.e. across all the training patterns). Keep in mind however that the training error does not provide a good estimate for a generalisation capability of the network.

2.5 Generalisation, regularisation, validation for robust learning

In the second part of the lab assignment, more advanced concepts will be introduced to make the development of a multi-layer perceptron and, particularly, the learning process more robust. In essence, the objective is to improve generalisation capabilities of your neural network. As discussed in the lecture, there are a number of approaches that practitioners adopt (here in the context of shallow networks). In this more advanced part of the lab assignment, we will focus on the problem of model selection (how the architecture is decided), validation, estimation of the generalisation error and regularisation (with a penalty term weighted by λ). These concepts are covered in detail in the lecture. Here, I would just like to draw your attention to the problem of validation and estimation of the "true" generalisation error, as it often involves sampling your data. In short, unlike in the first part of the assignment where the primary focus is on weight updates, in the second part you will have to split your available data in the development of your *robust* multi-layer perceptron to conduct

- *training*: updating weights in the learning process,
- *validation*: monitoring the process of learning and providing basis for a range of developer's decisions including model selection, and
- *testing*: the final/ultimate evaluation of the accuracy and generalisation power of your network on a separate (unseen) data subset.

3 Assignment - Part I

3.1 Classification with a single-layer perceptron

3.1.1 Generation of linearly-separable data

In the first place, please generate some data that can be used for binary classification (two classes). To simplify visual inspection, let us work with two-dimensional data. To start with, please draw two sets of points/patterns in 2D from multivariate normal distribution. In MATLAB, one could easily do that in the following way:

```
n = 100;
mA = [ 1.0, 0.5];   sigmaA = 0.5;
mB = [-1.0, 0.0];   sigmaB = 0.5;
classA(1,:) = randn(1,n) .* sigmaA + mA(1);
classA(2,:) = randn(1,n) .* sigmaA + mA(2);
classB(1,:) = randn(1,n) .* sigmaB + mB(1);
classB(2,:) = randn(1,n) .* sigmaB + mB(2);
```

Please choose parameters, `mA`, `mB`, `sigmaA`, `sigmaB`, yourselves to make sure that the two sets are linearly separable (so the means of the two distributions, `mA` and `mB`, should be sufficiently different). You can generate `n`=100 points per class and shuffle samples (`randperm` indexes in MATLAB) so that in your dataset you would not have just two concatenated blocks of samples from the same class. Although this reordering (shuffling) does not matter for bacth learning, it has implications for the speed of convergence for sequential (on-line) learning, where updares are made on a sample-by-sample basis. Please plot your patterns with different colours per class.

3.1.2 Classification with a single-layer perceptron and analysis

In this part, you are requested to implement and apply percpetron learning rules and analyse the results obtained on the generated dataset. It is helpful and fun to visualise the learning process by plotting a separating line (decision boundary) after each epoch of training. For that you could generate a sort of animation by iteratively re-plotting the updated decision boundary (see `drawnow` in MATLAB), in that case remember to keep the scame scaling in the plot every time you update it (in MATLAB you can force the scaling using `axis`). You are not required to demonstrate this animation to the teaching assistant, it is sufficient to show the final decision boundary (decision boundary can be derived from the weights and biases in W, i.e. it is a line defined by $Wx = 0$).

These are the tasks to focus on in the exercise:

1. Apply and compare perceptron learning with the Delta learning rule in batch mode on the generated dataset. Adjust the learning rate and study the convergence of the two algorithms.
2. Compare sequential with a batch learning approach for the Delta rule. How quickly (in terms of epochs) do the algorithms converge? Please adjust the learning rate and plot the learning curves for each variant. Bear in mind that for sequential learning you should not use the matrix form of the learning rule discussed in section 2.2 and instead perform updates iteratively for each sample. How sensitive is learning to random initialisation?
3. Remove the bias, train your network with the Delta rule in batch mode and test its behaviour. In what cases would the perceptron without bias converge and classify correctly all data samples? Please verify your hypothesis by adjusting data parameters, `mA` and `mB`.

Comparisons can be made using some evaluation metrics that could be the number or ratio of misclassified examples and/or the mean squared error at each epoch (iteration through the entire dataset).

3.1.3 Classification of samples that are not linearly separable

In this exercise, you should study how perceptron deals with data samples that are not linearly separable. To this end, please generate first a dataset with such property by adjusting `mA`, `mB`, `sigmaA` and `sigmaB`. In particular, you can make the means of the two multivariate normal distributions more similar and/or increase the spreads. As a result, you should see that the two clouds of points (corresponding to the two classes) overlap when you plot the samples. You can control the amount of overlap by the parameters of the distributions. Apply and compare perceptron learning and the Delta rules using either batch or sequential learning mode, similarly as in the previous exercise (p.1, section 3.1.2).

In the second part of this exercise please generate a different version of linearly non-separable dataset in the following way:

```
   ndata = 100;
   mA = [ 1.0, 0.3];    sigmaA = 0.2;
   mB = [ 0.0, -0.1];   sigmaB = 0.3;
   classA(1,:) = [ randn(1,round(0.5*ndata)) .* sigmaA - mA(1), ...
randn(1,round(0.5*ndata)) .* sigmaA + mA(1)];
   classA(2,:) = randn(1,ndata) .* sigmaA + mA(2);
   classB(1,:) = randn(1,ndata) .* sigmaB + mB(1);
   classB(2,:) = randn(1,ndata) .* sigmaB + mB(2);
```

Then apply the Delta learning rule in batch mode to this new dataset as well as to different versions of the subsampled data, i.e. before training please remove 25% of data samples (for two classes with n=100 samples each, remove 50 samples) according to the following scenarios:

- random 25% from each class
- random 50% from classA
- random 50% from classB
- 20% from a subset of classA for which classA(1,:)<0 and 80% from a subset of classA for which classA(1,:)>0

Perform simulations, i.e. subsample your data (from the same dataset generated originally) and perform perceptron learning, a few times. What is the effect of different data subsampling manipulations on the results, i.e. the localisation of the decision boundary as well as the performance? Since the number of samples representing different classes are different for the last three cases, it may be easier to measure the performance using the accuracy rate estimated independently for each class (alternatively, one could quantify *sensitivity* and *specificity*).
In your observations reflect on the implications of subsampling - data points that you remove could be thought of as an unseen test set. How would the resulting sampling bias, both in the context of uneven class representations and non-representative sample distribution, affect the generalisation?

3.2 Classification and regression with a two-layer perceptron

3.2.1 Classification of linearly non-separable data

Now we are ready to return to the problem of linearly non-separable patterns (the second part in section 3.1.3) with a multi-layer perceptron approach. To this end, please test a two-layer perceptron trained with backprop (the generalized Delta rule) in batch mode and examine how well it performs in separating the two classes. In particular, conduct the following experiments:

1. Modify the number of hidden nodes (just a few configurations, e.g. 3, 8, 15, 20, 30 nodes) and demonstrate the effect the size of the hidden layer has on the performance (both the mean squared error and the number/ratio of misclassifications). How many hidden nodes do you need to perfectly separate all the available data (if manageable at all given your data randomisation)?

2. Then, formulate this problem into a more realistic assignment where only a subset of data points is available for training a network (data that you use to calculate weight updates using backprop) and the remaining samples constitute a validation dataset for probing generalisation capabilites of the network. To do that, subsample the data in the same way as in the second part of section 3.1.3 and the removed samples treat as a validation set - compare two cases: 80% for training, 20% for validation and vice versa, 20% for training and the rest for validation. Make sure you do not use this hold-out validation set in the training process and instead you only use it to calculate the error (mean squared error or the ratio of misclassifications) at different stages/epochs of learning to monitor the progress. In this configuration, you can address the following tasks and questions

   - How do the learning/error curves for the training and the validation sets compare? How do they depend on the size of the hidden layer and the number of training samples (80% vs 20%)? Are they similar or in what cases do you observe more dissimilarity?
   - For a selected number of hidden nodes and the case with 80% training samples, compare a batch with sequential learning approach - is there any difference in terms of the validation performance?
   - Make an attempt at approximating/visualising the resulting decision boundary (for two networks with 3 and 30 hidden nodes, 80% training samples), i.e. where the network output is 0 (between the target labels of -1 and 1 for two classes, respectively).

3.2.2 The encoder problem *(Non-mandatory task)*

The encoder problem is a classical problem for how one can force a network to find a compact coding of sparse data. This is done by using a network with a hour-glass shaped topology, i.e. the number of hidden nodes are considerably fewer than the dimensionality of the data. The network is trained with identical input and output patterns, which forces the network to find a compact representation in the hidden layer. For this reason we often refer to such networks as autoencoders (finding a new representation basis or encoding through auto-association, i.e. input=output).

We will study a simple autoencoder with 8–3–8 feed-forward architecture (two-layer perceptron). The data are originally represented using ”one out of n” coding, i.e. only one input variable is active (=1) and the rest of input variables are in an inactive state, here: -1. For example:

$$\begin{bmatrix} -1 & -1 & -1 & 1 & -1 & -1 & -1 & -1 \end{bmatrix}^T.$$

There are eight such patterns in total. By letting the hidden layer have only three nodes, we can force the network to produce a representation where the eight-dimensional input patterns are represented in the three-dimensional space (spanned by the activations of the hidden nodes). Your task is to study what type of representation is created in the hidden layer.

We can use the Matlab-function `eye`, which generates a unit matrix, to create our pattern. Note that the input- and output patterns should be identical. The code to create the patterns becomes:

```
patterns = eye(8) * 2 - 1;
targets = patterns;
```

Please, use your implementation of the generalised Delta rule to train the network until the learning converges (it does not necessarily have to imply that the mean squared error is 0 but that the rounded outputs match the corresponding inputs). Then answer the following questions:

- Does the network always converge and map inputs to themselves?
- How does the internal code look, what does it represent? For that, you can inspect the activations of the hidden layer corresponding to input patterns. You could also examine the weight matrix for the first layer. Can you deduce anything from the sign of the weights?
- What happens when the size of the hidden layer is two instead of three, i.e. the architecture is 8-2-8?
- What purpose could autoencoders serve?

3.2.3 Function approximation

So far we have used the perceptron’s ability to classify data or find low-dimensional representations. Multi-layer perceptrons are known however for their ability to approximate an arbitrary continuous function. We will here study how one can train a two-layer perceptron network to approximate a function based on available input-output data examples. To enable visual inspection, the task is formulated for a function of two variables with a real value as output, f: $\mathbb{R}^2 \to \mathbb{R}$.

Generate function data As the function to approximate we choose the well known bell shaped Gauss function[3].

$$f(x, y) = e^{-(x^2+y^2)/10} - 0.5$$

For example, the following lines of MATLAB code create the input vectors x and y as well as the corresponding outputs z, and make a 3D plot.

```
x=[-5:0.5:5]';
y=[-5:0.5:5]';
z=exp(-x.*x*0.1) * exp(-y.*y*0.1)' - 0.5;
mesh(x, y, z);
```

This form of storage for input and output is perfect for visualizing graphically in MATLAB, but to be able to use the patterns as training data, i.e. to have all pair combinations of x and y dimensions, they must be changed to pattern matrices. In MATLAB the functions `reshape` and `meshgrid` can be used for that purpose. The following commands will put together the two matrices `patterns` and `targets` that are needed for training.

```
targets = reshape (z, 1, ndata);
[xx, yy] = meshgrid (x, y);
patterns = [reshape(xx, 1, ndata); reshape(yy, 1, n)];
```

(`ndata` is here the number of patterns, i.e. the product of the number of element in x and in y.)

Train the network and visualise the approximated function Once you have patterns paired with the corresponding targets, you can train your two-layer perceptron network with backprop in batch mode. Just as you monitored the training performance during the learning process in the previous sections, you could also illustrate the progress by visualising the output of the network across the training patterns (pairs of coordinates, x and y). The network's approximation, `out`, can be shown by performing the following transformations in MATLAB:

```
zz = reshape(out, gridsize, gridsize);
mesh(x,y,zz);
axis([-5 5 -5 5 -0.7 0.7]);
drawnow;
```

Here we assume that `gridsize` is the number of elements in x or y (e.g. `ndata` $=$ `gridsize` $\cdot$ `gridsize`). The variable `out` is the output produced in the forward

[3] Use the interval -0.5 to $+0.5$ to make sure that the output node will produce the values needed

pass when all the training data patterns (samples) are presented as input data. When all works, you should see an animated function that successively becomes more and more similar to the underlying Gaussian. Please, experiment with different number of nodes in the hidden layer to see how this parameter affects the final representation.

Evaluate generalisation performance As mentioned earlier, an important property of neural networks is their ability to generalise. This means producing sensible output data even for input data samples that have not been part of the training. As before, you are requested to train the network with a limited number of available data points. You should still look at the approximation ability at all points, reflected in the approximation error (mean squared error).

To subsample data for training, one can make a random permutation of the vectors `patterns` and `targets` and choose only *nsamp* first patterns (and examine different values of *nsamp*) for training. Following the training on the selected subset of samples, please validate the performance to see how well the network generalises even though it has learnt based on only a few example samples. Please plot also the resulting network's approximations in the 3D space and colour code the network's outputs corresponding to training samples (blue) and to validation samples (red).
To make this study more systematic, conduct the following experiments and report error estimates:

1. Choose 80% samples for training and the remaining 20% for validation. Vary the number of nodes in the hidden layer, e.g. 1, 3, 5, 10, 25, and try to observe any trends. Try to make a model comparison - what error estimates should you use to identify the "best"[4] model? What happens when you have very few (less than 5) or very many (more than 20) hidden nodes? Can you explain your observations?

2. For the selected "best" model with respect to the number of hidden nodes, compare the original configuration with 80% against the one with 20% training samples. How sensitive is the generalization to a training data subset (the actual subsets randomly sampled for training)? Do you think it matters which part of the input space training data happens to come from?

3. *A non-mandatory* study, but interesting and didactical, could be to examine the behaviour of your two-layer perceptron with a varying number of hidden nodes in the presence of Gaussian noise, controlled by the variance, added to z variables (function outputs) in the training data subset.

[4] Let's disregard here the need for a statistical test to make a valid comparison but still estimate the first and second moment statistics for each model examined.

4 Assignment - Part II

In the second part of the assignment you will develop a multi-layer perceptron network for chaotic time-series prediction. In particular, you will run a benchmark test with Mackey-Glass time series, which is a solution to the following differential equation (with real parameters β and γ, and integer n and delay τ)

$$\frac{dx}{dt} = \beta \frac{x(t-\tau)}{1 + x^n(t-\tau)} - \gamma x, \quad \gamma, \beta, n > 0.$$

Let's set β=0.2 and γ=0.1,n=10 and τ>17, e.g. τ=25. The equation can be discretised and solved with Euler's method:

$$x(t+1) = x(t) + \frac{0.2x(t-25)}{1 + x^{10}(t-25)} - 0.1x(t).$$

You can use this iterative formulation with $x(0)$=1.5 ($x(t)$=0 for t<0).

4.1 Data

Please configure the network to predict x(t+5) from four past values of the time series, i.e.: x(t-20), x(t-15), x(t-10), x(t-5) and x(t). This is an embedded time-lagged representation, which is often used with feed-forward networks operating on time series. Let us pick 1200 points from t=301 to 1500, and use them for *training, validation and testing.* In MATLAB, assuming that the time series is stored in a long row vector x, the inputs and outputs could be defined as follows:

```
t = 301:1500;
input = [x(t-20); x(t-15); x(t-10); x(t-5); x(t)];
output = x(t+5);
```

Important: Check how functions in the neural network library of your choice handles data (matrix dimensionality) and how they account for the bias term.

As mentioned, the selected 1200 samples should be divided into three consecutive non-overlapping blocks for training, validation and testing. It is expected that the last 200 samples are used for the final evaluation (test), and how exactly you split the rest of available data into training and validation is your own decision (though the subsets should constitute consecutive blocks since it is a time series). As a measure of performance, the mean squared error is recommended.

4.2 Network configuration

In the library of your choice, please construct a multi-layer perceptron network with five inputs and one output to handle the time series prediction problem.

Make sure that you configure your data in an appropriate manner, consistently with the instructions in the previous section. Set up the training process with a batch backprop algorithm and *early stopping* to control the duration of learning (number of epochs), thereby preventing from overfitting, based on the error estimate on the *hold-out* validation data subset. If you identify that early stopping leads to premature ending of the learning process, try to fix it or in the worst case scenario - remove it and define an alternative criterion for stopping. In addition, you will be asked to make use of some regularisation techniques, e.g. weight decay, to further boost the generalisation performance. Please identify suitable library functions and decide which regularisation approach you are going to adopt, motivate your choice in the end.

When you design your network, please parameterise the number of hidden layers and the number of nodes. You will be asked to evaluate three-layer perceptrons with varying number of nodes in the hidden layers (please note that there are two hidden layers for a three-layer perceptron). It is recommended that you use sigmoidal transfer functions in the hidden layer(s) and linear in the output. Also, parameterise the regularisation method (hyperparameter λ) and control/monitor the speed of convergence - keep in mind that inappropriate choice of the learning rate, η, can lead to premature convergence or to excessively long simulations (even without the effect of convergence). The parameterisation of key hyperparameters will make your implementation generic so that running different simulations can be done without any extra implementation overhead. To report the performance you should also run multiple simulations for estimating the mean statistics in the presence of stichastic factors, e.g. initialisation.

4.3 Simulations and evaluation

Here the focus is on understanding the implications of noise for generalisation. The idea is to use the generated data to design a multi-layer perceptron for Mackey-Glass time series prediction with the aim of maximising generalisation capability under varying noise conditions.
In particular, you are asked to complete the tasks listed below and address the corresponding questions.

1. Generate data without any noise, plot the resulting time series and split data into training, validation and test subsets.
2. Propose a few configurations of a three-layer perceptron with different combinations of the number of nodes in the hidden layer[5] (nh_1 = 2, 4, nh_2 = 2, 5, 15) and choose the most and least suitable architectures based on the validation error. It is recommended that you use early stopping with the validation subset as a hold-out set.
How robust are the most and least suitable architectures, which you have identified, with respect to random weight initialisation?

[5] This is a so-called grid-search over a hyperparameter space, i.e. evaluate a set of combinations of hyperparameter values (Cartesian product) such as the number of nodes in respective hidden layers nh_1 x nh_2, e.g. 2x2, 2x5, 2x15, 4x2, 4x5, 4x15.

3. Report the test performance for these two architectures - the most and least suitable ones. This is the conclusive estimate of the generalisation error on the unseen data subset. Please plot also these test predictions along with the known target values.

4. Add zero-mean Gaussan noise to training data to simulate a real-world scenario. Select the number of nodes in the first hidden layer that corresponds to the most suitable architecture identified in p.2, and then vary only the number of nodes in the second hidden layer, say nh_2 = 2, 15[6], to examine what effect it has on the validation prediction performance in the presence of noise (experiment with the additive Gaussian noise that corrupts the training data subset, $\sigma = 0.05, 0.3$). Use weight decay regularisation parameterised with the regularisation constant, λ (say λ = 1e-1, 1e-2, 1e-5), and you can skip early stopping.
Are there any strong trends or particular observations to report? What is the effect of regularisation on the error as well as the distribution of weights? How does the regularisation parameter, λ, interact with the amount of noise (as the amount of noise increases, are there any incentives to change the regularisation paremeter - does it have any effect on the training and validation performance)?

5. *(Non-mandatory task)* For intermediate level of noise ($\sigma = 0.1$) choose the best three-layer model described by nh_1 x nh_2 and λ, used for training (explain and motivate which data subset you use for this purpose), and compare the selected model with the two networks you picked as the most and least suitable on the clean original data in p.2 (keep in mind that you have to train your network architectures from scratch on the noisy data) in terms of generalisation error estimated on the test set. On this occasion, the validation set will only serve to implement early stopping as no extra hyperparameters must be identified (you have three fixed architectures and you can set to λ=0 for the two networks found in p.2).
How does the generalisation performance on noisy data compare to the results reported in p.3 for the original clean data?

These tasks can be handled in the same set of experiments so you do not have to run separate simulations for each item on the agenda. Also, you can automate a lot of computations, e.g. grid parameter search can be implemented in iterations over different combinations of the number of hidden layer units. Then more iterations can be added for varying λ and/or σ.
Present your key findings in tables and figures.

Importantly, bear in mind that your neural network simulation are stochastic in nature with different sources of variability, including random initialisation of weights and random noise added in Part II. Make sure that you account for these factors in your evaluation and analysis (in many cases you should perform multiple iterations of your simulations and estimate both the first and the second statistical moment of the error/loss).

[6] If you were to optimise these hyperparameters it would be more in the spirit of a greedy search where you first optimise one hyperparameter, fix its value and optimise with respect to another hyperparameter.