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 associated 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 back-propagation. 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 sole 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 rely on the existing libraries (as mentioned above, NN toolbox in Matlab, sckit-learn in Python and TensorFlow 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 $(-1/1)$ represen-

tation of the patterns. This representation however is not intuitive for visualisation. Therefore, for visual inspection, you may choose to write a function that transforms symmetric -1/+1 pattern into a binary 0/1 pattern.

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 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 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. 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).

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. 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 truegeneralisation error, as it often involves sampling your data. In short, unlike in the first part of the assignment with the primary focus on weight updates, in the second part you

will have to split your available data in the development of your 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 Generate 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's work with two-dimensional data. To start with, please draw two sets of points in 2D from multivariate normal distribution. Choose parameters yourselves to make sure that the two sets are linearly separable (so the means of the two distributions should be sufficiently different). You can generate 100 points per class and shuffle samples 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 points with different colours per class.

3.1.2 Perform classification with a single-layer perceptron and analyse the results

Apply and compare both perceptron learning and Delta learning rules on the generated dataset. Please try also to compare sequential with a batch learning approach. Comparisons can be made using some evaluation metrics that could be the number or ratio of misclassified examples at each epoch (iteration through the entire dataset). How quickly do the algorithms converge? Please plot the learnign curves for each variant of learning. You could also 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; you are not required though to demonstrate this animation to the teaching assistant).

3.1.3 Classification of samples that are not linearly separable

Perform the same task as described above, i.e. including data generation, percpetron learning (both perceptron and Delta rules) and evaluation, for data that are not linearly separable. The easiest way to synthesise such data is to make the means of the two multivariate normal distributions mentioned above more similar and/or increase the spread (variance). 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. Next you can train the perceptron to classify the data and monitor its performance. Please make similar analysis as in the case of linearly-separable data. Pay special attention to the manifestations of non-convergent learning with the use of a classicial perceptron learning rule.

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 previous problem of linearly non-separable patterns. Test a two-layer perceptron trained with backprop and verify that it can solve the problem (separate the two classes). Modify the number of hidden 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 the available data (if manageable at all given your data randomisation)? In parallel with the evaluation on the training data (data that you use to calculate weight updates using backprop), please make also evaluation on a new, previously unseen, test dataset. To this end, please generate from your random distributions 50 new samples for each class and use them only to calculate the error (mean squared error or the ratio of misclassifications) at different stages/epochs of learning. Make sure that you do not use them in the learning process. You can then study the following questions:

- How do the training and test learning curves compare?
- How do the training and test classification results depend on the size of the hidden layer?
- How many epoch iterations do you need for convergence?
- Is there any difference between batch and sequential learning approaches?

3.2.2 The encoder problem

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 öne 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.

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). Does the network always succeed in doing this? 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?

3.3 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: $\mathbb{R}^2 \rightarrow \mathbb{R}$.

3.3.1 Generate function data

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

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

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

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);
```

The form of storage we now have 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 matrixes `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, ndata)];
```

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

3.3.2 Train the network and visualise the approximated function

Now put together all parts to get a program that generates function data. This will during the learning after each epoch show how the function approximation looks like. When all works, you should see an animated function that successively becomes more and more similar to a Gaussian. Experiment with different number of nodes in the hidden layer to get a feeling for how this parameter affects the final representation.

The network's approximation can be visualised by performing the transformations corresponding to those in the previous section in the reverse direction. Matlab commands to do this are:

```
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 pass when all the training data patterns (samples) are presented as input data.

(To create an animation of the learning process in Matlab, function `drawnow` can be used.)

3.3.3 Evaluate generalisation performance

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. We will not modify the experiment above but only train the network with a limited number of available data points. We will still look at the approximation ability at all points as before, thi time we will focus on 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 n first patterns (and examine different values of n) for training. The program will then need to do two different forward passes; one for the training points and one for all points. Only the first pass is associated with an update of the weights. The result of the second pass is used to see how well the network generalises. In our case, a good approximation means that the network can recreate the whole function even though it has learnt based on only a few example samples (training data points).

Test with $n = 1$ up to $n = 25$. Vary the number of nodes in the hidden layer and try to observe any trends. What happens when you have very few (less than 5) or very many (more than 20) hidden nodes? Can you explain your observations? 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 added to z variables (function outputs) in the test data subset.