Artificial Neural Networks and other Learning Systems, DD2432

Exercise 1:

Feed forward networks — The delta rule and back propagation

1 Objectives

This exercise is about feed forward networks and how they can be trained using error based learning methods. When you are finished you should understand:

- how networks can be used for *classification*, *function approximation* and *generalization*
- the limitations of single layer networks
- that multilayer networks are general classifiers and function approximators
- how the *delta rule* and the *generalized delta rule* can be used to train a network.

2 Introduction

In this exercise you will implement a single layer and a two layer perceptron and study their properties. Since the calculations are naturally formulated in terms of vectors and matrices, it makes sense to work in Matlab[1].

We will limit ourselves to two learning algorithms: the *delta rule* (for the single layer perceptron) and the *generalized delta rule* (for the two layered perceptron). The generalized delta rule is also known as the *error backpropagation algorithm* or simply "*BackProp*". This is one of the most common supervised learning algorithms for neural networks and can be used for different kinds of applications. In this exercise you will have the opportunity to use it for *classification*, *data compression*, and *function approximation*.

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

2.1 Data Representation

Let us first decide how the patterns that are going to be learn are doing to be represented. Since this is supervised learning, it means that training data consists of pairs of patterns in and patterns out. We will choose to do *batch* learning. This means that all patterns in the training set will be used simultaneously (in parallel) instead of stepping through them one by one and update weights successively for each pattern. Batch learning is better suited for a matrix representation and is significantly more effective in Matlab. In batch learning, each use of the whole pattern set once is commonly denoted an *epoch* (Swedish epok) By a suitable choice of representation, an epoch can be performed with just a few matrix operations.

Further, we must also decide whether we want to use a binary representation $(0/1)$ or a symmetric $(-1/1)$ representation of the patterns. It is practical to use the symmetric representation -1 to 1 since several of the formulas become simpler and all thresholdings can be done at zero. This representation however is not as clear for the eye as the binary. For visual inspection, you may therefore choose to write a function that transforms a symmetric pattern into a binary pattern.

We are going to choose to store the patterns in and the patterns out as columns in two matrixes: X and T respectively. With this representation, the XOR problem would for instance be described by:

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

This is read column wise and means that the pattern $(-1, -1)$ should produce the result -1, the pattern $(1, -1)$ should give 1 etc.

The one layer perceptron works in the following way. First the sum of the weighted inputs, then adds the *bias* term. Finally the thresholding is done. 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. Firstly, we make sure to get rid of the bias term by adding an extra input signal whose value always is one (and a weight corresponding to the bias value). In our 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 described by using a matrix W with as many columns as the length of the input patterns as with as many rows as the length of the output patterns. The output patterns from the network produced by *all* input patterns can then be calculated by a simple matrix multiplication followed by a

thresholding at zero. Learning with the Delta rule aims, with the representation selected, to find the weights W that gives 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 for which a one layer perceptron can not classify all patterns perfectly.

3 Computer environment for the course

If you have hot yet done this, start by adjust your computer environment for the course. This is done by the command "`course join`" and is necessary for the labs to work according to the instructions. Further, if you have not yet registered yourself in the lab data base on these computers (with the comment "`res checkin`") it also time to do that now.

```
> res checkin ann08
> course join ann08
```

You only have to do these command once. But, for the latter command to take effect, you have to log out. Probably your lab partner also have to do this as well, so repeat all up to now for the other person(s) in the lab group.

There is no ready code to copy for this lab, and instead you will write all code yourself (not that much), in a catalog as suggested below:

```
> mkdir -p ~/ann08/lab1
> cd ~/ann08/lab1
```

4 Classification with a one layer perceptron

4.1 Generate training data

Start by generate some test data that can be used for classification. To make it simple to look graphically at the results, we limit ourselves with data points in 3 dimensions. We let input data consist of two classes, each one with a normally distributed spread around is mean point.

It is convenient to make a file (for instance named `sepdata.m`) with all Matlab commands needed to create the test set. In that way one can use the commend `sepdata` to generate new training data.

Numbers that are normally distributed around a mean can be generated with the Matlab function `randn`. This function creates a matrix with normally distributed random numbers with the mean zero and the standard deviation one. Two classes with one hundred data points each can be generated with this:

```
classA(1,:) = randn(1,100) .* 0.5 + 1.0;
classA(2,:) = randn(1,100) .* 0.5 + 0.5;
classB(1,:) = randn(1,100) .* 0.5 - 1.0;
classB(2,:) = randn(1,100) .* 0.5 + 0.0;
```

We have here scaled and translated the points so that class A has its mid point at $(1.0, 0.5)$ and class B with $(-1.0, 0.0)$. Both have standard deviation 0.5. The Matlab operator `.*` designates element wise multiplication (as opposed to matrix/vector multiplication).

Now what remains is to put together the two classes to a single set. You can use the Matlab matrix operations to put together all points to a single matrix, named `patterns`, with 200 columns:

```
patterns = [classA, classB];
```

We also have to create a matrix `targets` with the correct answers. Let class A correspond to the value 1 and class B to the value -1. Write yourself a Matlab command that puts together a `targets`-matrix. Hint: you can use the Matlab function `ones`.

It is perhaps not that realistic that the points in the data set are ordered so that all points from one class comes first and the other next. The points should instead come in a random order [2] That can be done by use the function `randperm`, that creates a random permutation vector. This vector can be used to shift order for the columns in both `patterns` and `targets`.

```
permute = randperm(200);
patterns = patterns(:, permute);
targets = targets(:, permute);
```

If you want to look at where your data points end up, you can use the following commands:

[2] For batch learning this does not really matter since all contributions are summed up before the weight update. But for a sequential update the order may play a role.

```
plot (patterns(1, find(targets>0)), ...
      patterns(2, find(targets>0)), '*', ...
      patterns(1, find(targets<0)), ...
      patterns(2, find(targets<0)), '+');
```

4.2 Implementation of the Delta rule

We are now going to implement the Delta rule. Preferably you write these Matlab commands in a file of its own, `delta.m`, so that you can use the command `delta` to run the algorithm again and again. Start by assuming that the training data are stored in the global variables `patterns` and `targets`. That makes it possible to later test the same algorithm with other training data.

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. To get the total change from an epoch, the 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 the Matlab commands that implements these computations. Note that X not directly corresponds to the content of the variable `patterns` since it doesn´t contain the row with ones needed for the bias terms to enter the computation.

Write your code so that the learning according to the formula above is repeated `epochs` times (where 20 is a suitable number). 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 it propagates to the weights to the output. This will be especially important in the next part of the lab with a two layer perceptron.

Hint: you can easily get the dimensions of the matrixes by the commands:

```
[insize, ndata] = size(patterns);
[outsize, ndata] = size(targets);
```

But, don´t get tempted to use `for`-loops to step through the single elements in the matrixes. Instead, use the capability of Matlab to deal with matrixes and vectors instead.

Before the learning can take place, the weights must have start values. The normal procedure is to start with small random numbers. Construct yourself the commands needed to create an initial weight matrix, for instance by using the `randn`-function. Note that the matrix must have the correct dimensions.

Inspect the separation line during the learning

For the program you just have written to be meaningful, it must be complemented with commands which in some way presents the result of the computations. In principle one could inspect the values of the W-matrix after learning, but that is not especially clear. A better way is to collect some form of statistics of how many of the patterns that are classified correctly.

We will however this time use the fact that we are dealing with two dimensional data. That makes it possible to graphically show how learning succeeds to position the separation line between the two classes A and B. The following commands draws the separation line and the data points in the input space:

```
p = w(1,1:2);
k = -w(1, insize+1) / (p*p');
l = sqrt(p*p');
plot (patterns(1, find(targets>0)), ...
      patterns(2, find(targets>0)), '*', ...
      patterns(1, find(targets<0)), ...
      patterns(2, find(targets<0)), '+', ...
      [p(1), p(1)]*k + [-p(2), p(2)]/l, ...
      [p(2), p(2)]*k + [p(1), -p(1)]/l, '-');
drawnow;
```

These lines can be used after the learning is done, but it is more fun to put them inside the `for` loop for the learning epochs so that you incrementally can see how the line moves following each learning epoch. To avoid that Matlab changes the scale of the axis while the "animation" runs, you can set a fixed scale for the axes by:

```
axis ([-2, 2, -2, 2], 'square');
```

Now put together all parts to a working program and test. Try out different numbers for η and `epochs`.

4.3 Non-separable data

Now we are going to generate data for a classification problem that is not that simple. The following commands generate two classes that are not linearly separable.

```
classA(1,:) = [randn(1,50) .* 0.2 - 1.0, ...
               randn(1,50) .* 0.2 + 1.0];
classA(2,:) = randn(1,100) .* 0.2 + 0.3;
classB(1,:) = randn(1,100) .* 0.3 + 0.0;
classB(2,:) = randn(1,100) .* 0.3 - 0.1;
```

Make a new file, `nsepdata.m` containing these lines and what is needed to get the variables `patterns` and `targets` again. Test what now happens when using the Delta rule.

Hint: When proceeding it is convenient to remove all old variables with the command `clear` before each new run to make debugging easier.

5 Classification with the two layer perceptron

5.1 Implementation of a two layer perceptron

We have now come to the most important part of the lab, implementation of the generalized Delta rule, also denoted Backprop. You are going to use this in several different experiments, so it pays off to make this general. Specifically 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 a multi layered network to be able to add something new it is important to use a non-linear transfer function, often denoted φ. Normally one chooses a function that has a derivative which is simple to compute, like

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

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 $\varphi(x)$ since this value has to be computed anyway.

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 itself backwards layer by layer (thereby giving rise to the backpropagation). Finally the weight update is done. We start by looking at how the forward pass can be implemented.

5.1.1 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 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[3]. 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:

[3]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.

```
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.

5.1.2 The backward pass

The backward pass aims at calculating the generalized 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 $\delta^{(o)}$:

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

Let us express this 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).

Now we can express also this in code:

```
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:hidden, :);
```

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 `hidden` contains the number of nodes in the hidden layer.

5.1.3 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

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

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

Before we implement this as Matlab code, we will make sure to introduce an improvement to the algorithm commonly referred to as *momentum* (Swedish tröghet). This means that one does not directly update the weights with the values from above, but with a sliding average. This will suppress fast variations. This is has the consequence that one can use a larger learning rate, which in term leads to a faster convergence of the algorithm. A scalar factor α controls how much of the old vector of changes one keeps. A suitable value of α is often 0.9. The new update rule then becomes (in matrix form):

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

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

$$\Delta W = \eta\Theta$$

$$\Delta V = \eta\Psi$$

The same thing expressed in code becomes:

```
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. Don't 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.

A special problem is how you can see what the algorithm is doing. This is not as simple as for the one layer perceptron to draw the line of separation even when the algorithm is used for classification. The best is probably to calculate the number of nodes with wrong classification over all patterns and all epochs, and incrementally save these values in a vector that can be plotted at the end of

the learning. Not that the algorithm, as it has been described here, gives graded values the matrix `out`. If you want to see how the network classifies, you can threshold this at zero. This is most easily done with the function `sign`.

The total number of erroneous nodes for all patterns can be calculated by a double sum over all element, like in this example:

```
error(epoch) = sum(sum(abs(sign(out) - targets)./2));
```

We have here assumed that there is a vector `error` of sufficient length where we can store the result.

5.2 Non-linear separable data (once more)

Now we are ready to return to the previous problem of non-linearly separable classes. Test the new algorithm and check that it can solve the problem. Observe that it is not absolutely guaranteed that *all* patterns are going to be correctly classified unless one has very many hidden nodes. The data set is generated by random, and it is likely that some points ends up very off. Study how many nodes are needed in the hidden layer.

5.3 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 much fewer than the number in the input and output layers. The network is trained with identical input and output patterns which forces the network to find a compact representation in the hidden layer.

We will here study the classical 8–3–8 problem. You let input data (and thus also output data) consist of patterns where only one element out of eight is "on". With the representation in the interval $(-1, 1)$ this means patterns of the type

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

There are in total eight such patterns. By letting the hidden layer have only three nodes, we can force the network to produce a representation where the eight patterns are represented by only three values. Your task is to study what type of representation that 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;
```

Use your implementation of the generalized Delta rule from above to train the network until the classification becomes correct. Does the network always succeed doing this? Then, study how the internal code looks like. This is most simply done by looking at the weight matrix for the first layer. Specifically note whether there is any particular pattern in the sing of the weights. To enable this, use the sign function.

6 Function approximation

So far we have used the perceptron's ability to classify data. But,just as important is the ability to approximate an arbitrary continuous function. We will here study how one can train a two layer network (i.e. a network with one hidden layer) to approximate a function which is provided by examples of it. To make it convenient to look at the function and its approximation we choose a function of two variables with a real value as output.

6.1 Generate function data

As the function to approximate we choose the well known bell shaped Gauss function[4].

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

The following code-lines creates the argument vectors x and y as well as the result matrix z.

```
x=[-5:1:5]';
y=x;
z=exp(-x.*x*0.1) * exp(-y.*y*0.1)' - 0.5;
```

We can use the function `mesh` to see how the function looks like:

```
mesh (x, y, z);
```

The form of storage we now have for input and output is perfect for visualizing graphically, but to be able to use the patterns as training data they must be changed to patterns matrixes of the proper shape. We can here use the functions

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

`reshape` and `meshgrid`. The following commands will put together the two matrixes `patterns` and `targets` that are needed for the 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.)

6.2 Plot the result

By performing the corresponding transformation in the reverse direction, one can look at the network's approximation graphically. The 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 have assumed that `gridsize` is the number of elements in x or y (e.g. `ndata` = `gridsize` $\cdot$ `gridsize`). The variable `out` is the result from the forward pass when all the training data is presented as input data.

The function `drawnow` is used to force Matlab to show the diagram even though the calculations are not completed. This means that we can insert the code lines above inside the loop that is used to successively change the weights. This way we get an animation of the learning progress.

6.3 Train the network

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 it affects the final representation.

7 Generalization

An important property of neural networks is their ability to generalize. This means producing sensible output data even for input data which has not been part of the training. We will not modify the experiment of above by only train the network with a limited number of data points. We will still look at the approximation ability at all points as before.

To make this, one can make a random permutation of the vectors `patterns` and `targets` and only train with the n first patterns (and test different values of n). 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 generalizes. In our case a good approximation means recreating the whole function even though it has only been showed examples from a few points.

Test with $n = 10$ and $n = 25$. Vary the number of nodes in the hidden layer and try to see any trend. What happens when you have very few hidden nodes (less than 5)? What happens when you have very many (more than 20)? Can you explain your observations?