Artificial Neural Networks – Lab 3
Simple neuron models and learning algorithms

Purpose
To study some basic neuron models and learning algorithms by using Matlab’s neural network toolbox.

Presentation
The results from the exercise should be presented individually to one of the lab organizers. If a question or sub-task appears with a box [ ] then your respective answer must be presented. Students must be present at the next lab to answer questions on their work (the deadline for completion is the start of the next lab). If two students work together in a pair, then both students must be present and able to demonstrate the software that they have written and explain the answers.

Data and Matlab functions
Data and m-files can be downloaded from the course homepage (http://www.aass.oru.se/~tdt/ann). The files you need for this lab are

adaptalc.m  dsig.m  initalc.m  
lab3.mat  messages.mat  trainp_sh.m

Note: Read each task carefully before starting to solve it.

Preparation
Read chapter 1 of the course textbook. Then read each task carefully before starting to solve it.

Task 1, Demonstrations in Matlab
There are a number of demonstrations of different types of neural networks and learning algorithms in the neural network toolbox of Matlab. To access them, go to the ‘Launch Pad’ window and click on ‘Neural Network Toolbox’, then select ‘Demos’. In this task you have to run the following demonstrations:

• Simple neuron and transfer functions
• Neuron with vector input
• Decision boundaries
• Perceptron learning rule
• Classification with a 2-input perceptron (note - there's an error in the text here: it says there are 5 input vectors, but really there are only 4)
• Linearly non-separable vectors

Read the text and study the pictures in the demonstrations carefully. Try to understand the following things:

1. How the weights and bias values affect the output of a neuron.
2. How the choice of activation function (or transfer function) affects the output of a neuron. Experiment with the following functions: identity (purelin), binary threshold (hardlim, hardlims) and sigmoid (logsig, tansig).

3. How the weights and bias values are able to represent a decision boundary in the feature space.

4. How this decision boundary changes during training with the perceptron learning rule.

5. How the perceptron learning rule works for linearly separable problems.

6. How the perceptron learning rule works for non-linearly separable problems.

Task 2, Pavlov’s dog and Hebbian learning

In this task you will do an extended version of the famous dog experiment by Pavlov. Simulate the dog’s behaviour by using a single neuron with four inputs and the binary threshold function hardlim. Write an m-function, hebbian, with an interface given by

```
function [y, w] = hebbian(x,w,lr)

% %HEBBIAN Hebb’s rule for a single neuron
% x - input vector (column)
% w - weight vector (column)
% lr - learning rate

% Compute the output of the neuron
y = ????

% If there are 3 arguments then do one step of Hebb’s learning rule
if nargin > 2
    Dw = ????
    w = w + Dw;
end
```

There are two ways to call this function. If you want to update the weights (e.g., for training) then use
```
>> [y, w] = hebbian(x,w,lr);
```
If you don’t want to update the weights (e.g., for testing) then use
```
>> [y, w] = hebbian(x,w);
```

The experiment you are going to do is similar to Pavlov’s classical experiment, but with some extensions. The dog’s primary stimulus is food but there are some other stimuli too: light, bell and whistle. If the output of the neuron is 1, the dog is producing saliva (that is, it is drooling). If the output is 0, the dog is not responding to the stimuli. The input vector has the following meaning:

\[
\mathbf{x} = \begin{bmatrix}
\text{Food} \\
\text{Light} \\
\text{Bell} \\
\text{Whistle}
\end{bmatrix}
\]

If an element in \( \mathbf{x} \) is one it means that a particular stimulus is active. For example, \( \mathbf{x} = [1 \ 0 \ 1 \ 0]^T \) means that the dog is shown food at the same time as a bell is ringing. If you like, you can define variables called food, light, etc., as follows:
then you can call the function as follows:

>> [y, w] = hebbian(food+light, w, lr);

OK, now for some exercises...

1. Start with the weight vector $w = [1 \ 1 \ -2 \ -2]^T$ and compute the output of the neuron for each stimulus, with only one stimulus active at a time. Do not update the weights now. Note the result.

2. Call `hebbian` three times with an input vector showing only food (use the $w$ from 1). This time, you must train the neuron using Hebb’s rule with a learning rate of 1. What happens to the weight vector?

3. Initialize the weight vector again (same $w$ as in 1). Call `hebbian` three times with learning rate 1 and an input vector showing both food and light but no other stimulus. How do the weights in $w$ change?

4. Why?
4. Initialize the weight vector again. Call `hebbian` five times with learning rate 1 and an input vector that shows food, light and whistle. What happens to the weights in $w$?

Why?

5. Now, using the resulting weight vector from 4, compute the output of the neuron for each stimulus, with only one stimulus active at a time (don’t update the weights). What’s the result? Compare with 1 and explain.

---

**Task 3, Training the perceptron**

Here you will study the behaviour of the perceptron learning rule in two different classification problems. The file `lab3.mat` contains four matrices, as follows:

<table>
<thead>
<tr>
<th>Matrix</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>Input vectors for the first problem</td>
</tr>
<tr>
<td>t1</td>
<td>Target outputs (classes) for the first problem</td>
</tr>
<tr>
<td>p2</td>
<td>Input vectors for the second problem</td>
</tr>
<tr>
<td>t2</td>
<td>Target outputs (classes) for the second problem</td>
</tr>
</tbody>
</table>

Note that there is only one set of data (inputs + target output) for each problem because we are only going to train the perceptron. Of course, in a real world problem, you would want to use at least two data sets — one for training, and one for testing :-)

If you look at $t1$ and $t2$, you will see that in each problem there are two classes. Plot the training examples using different colours or symbols for each class, e.g., using

```matlab
>> plot (p1(1,1:50),p1(2,1:50),'b+',p1(1,51:100),p1(2,51:100),'ro')
```

The neural network toolbox has some special functions for using the perceptron — see the online help for `newp`. This function requires an argument which specifies the expected range for each of the inputs in the input vector. The function `minmax` can be used for this purpose, as follows:

```matlab
>> mm = minmax(p1);
>> net = newp(mm,1);
```

The above steps will automatically work out the correct range for each input, and then create a new perceptron with one neuron (and therefore one output).

You can train the perceptron using the m-function `trainp.sh`. This function is called as follows:
>> net = trainp_sh(net, p1, t1, 1);

The last argument to \texttt{trainp\_sh}, the number of epochs (training steps) should be set to 1. See also the online help. This function will also update the graphical display, showing the decision line as well as the training examples.

For each of the two classification problems, plot the training examples and create a new perceptron, as described above. Then, do successive calls to \texttt{trainp\_sh} and study the graph (resize the Matlab window so that you can watch the graph at the same time). Once you get the general idea of what’s happening, you can set the number of epochs to a higher number so that things go a little faster. In your report, please write answers to the following questions (for both data sets):

\begin{enumerate}
\item Does the training converge?
\item Is the perceptron able to classify all training vectors correctly after training?
\item If not, why?
\item Can you say something about the perceptron’s ability to classify new unknown patterns? (note the position of the decision line)
\end{enumerate}

\section*{Task 4, Adaptive filtering of speech signals using Adaline}

In this task you will use an Adaline in an application for adaptive echo cancellation. Echos in speech signals may be a problem in telecommunication over long distances. Two signals are given in \texttt{messages.mat}: \texttt{inmessage} and \texttt{outmessage}. The sampling frequency is 22050 Hz. It is possible to listen to the signals using

\begin{verbatim}
>> sound(mymessage, 22050);
\end{verbatim}

You can also plot the signals. Assume that the incoming signal \texttt{inmessage} is contaminated with an echo from the outgoing signal \texttt{outmessage}.

To simulate on-line adaptive training, delayed vectors of \texttt{outmessage} must be created. Use the command \texttt{dsig} with seven units delay. To understand \texttt{dsig} you can try it on some small signal such as \{1, 2, 3, 4, 5\} with three units delay. Initialize one Adaline with 8 inputs + bias, using the function \texttt{initalc}. Adaptive training with the Widrow-Hoff learning rule is done using the function \texttt{adaptalc}.

Now you have to find out which inputs and target outputs to use for the Adaline. You also need to find out which one of the returned signals from \texttt{adaptalc} carries the decontaminated input message. Then you can try to adaptively cancel the echo using \texttt{adaptalc}.
1. Run `adaptalc` with a learning constant of 0.01. Plot and listen to the result. Explain the result. What is really happening?

2. Investigate also the effect of different learning constants. Use 10 evenly spaced constants between 2 and 0.0001. Listen to the result. What happens if the learning constant is too big?

What happens if the learning constant is too small?

Which learning constant gave the best result?