MINISTRY OF EDUCATION AND SCIENCE OF UKRAINE
SUMY STATE UNIVERSITY

I. Knyaz’

Modelling of Neural Networks

Lecture notes

In two parts

Part 1

Sumy
Sumy State University
2017

MINISTRY OF EDUCATION AND SCIENCE OF UKRAINE
SUMY STATE UNIVERSITY

I. Knyaz’

Modeling of Neural Networks

Lecture notes

In two parts

Part 1

APPROVED
at the session of Applied Mathematics and Complex Systems Modelling Department as as lecture notes on discipline “Modelling of Neural Networks”.
Minutes №4 of 04.02.2017

Sumy
Sumy State University
2017

Modelling of Neural Networks: lecture notes in two parts. Part 1 / compiler I. Knyaz’. – Sumy : Sumy State University, 2017. – – 62 p.

Department of Applied Mathematics and Complex Systems Modelling

CONTENTS

P.

Introduction..4
  1.1 The Brain as an Information Processing System......................4
  1.2 Artificial Neuron Models..6
  1.3 Historical background...9
  1.4 Neural networks versus conventional computers 10
Artificial Neuron Networks (ANNs) 12
  2.1 Learning problem...12
  2.2 Perceptrons 16
Multi-Layer Artificial Neural Networks..27
  3.1 Multi-Layer Network Architectures 27
  3.2 The Backpropagation Learning Routine................................31
Competitive Networks – the Kohonen Self-organizing Map 42
  4.1 Self-organizing maps..42
  4.2 Learning in biological systems 43
  4.3 Architecture of the Kohonen Network 44
  4.4 The Kohonen Network in Operation 45
  4.5 Training the Kohonen Network..47
Radial Basis Function Networks 52
  5.1 Architecture 52
  5.2 Training an RBF network...56
  5.3 Advantages of an RBF...58

Lecture 1

Introduction

1.1 The Brain as an Information Processing System

The human brain contains about 10 billion nerve cells, or neurons. On average, each neuron is connected to other neurons through about 10000 **synapses**. The brain's network of neurons forms a massively parallel information processing system. This contrasts with conventional computers, in which a single processor executes a single series of instructions.

Against this, consider the time taken for each elementary operation: neurons typically operate at a maximum rate of about 100 Hz, while a conventional CPU carries out several hundred million machine level operations per second. Despite of being built with very slow hardware, the brain has quite remarkable capabilities:

- its performance tends to degrade gracefully under partial damage. In contrast, most programs and engineered systems are brittle: if you remove some arbitrary parts, very likely the whole will cease to function;
- it can learn (reorganize itself) from experience;
- this means that partial recovery from damage is possible if healthy units can learn to take over the functions previously carried out by the damaged areas;
- it performs massively parallel computations extremely efficiently. For example, complex visual perception occurs within less than 100 ms, that is, 10 processing steps!
- it supports our intelligence and self-awareness.

The basic computational unit in the nervous system is the nerve cell, or **neuron** (see Fig. 1).

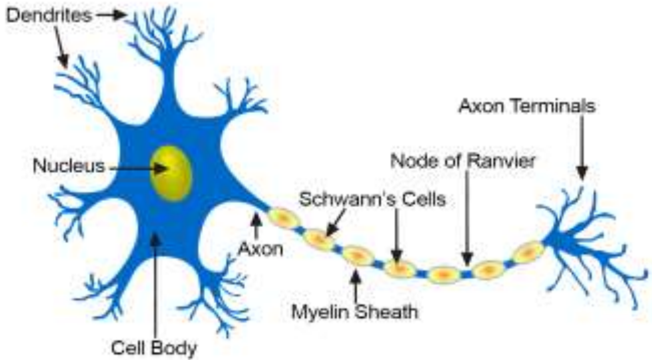


Figure 1 – Structure of a Typical Neuron (from SEER Training Web Site)

A neuron has:

- Dendrites (inputs)
- Cell body
- Axon (output)

A neuron receives input from other neurons (typically many thousands). Inputs sum (approximately). Once input exceeds a critical level, the neuron discharges a **spike** – an electrical pulse that travels from the body, down the axon, to the next neuron(s) (or other receptors). This spiking event is also called **depolarization**, and is followed by a **refractory period**, during which the neuron is unable to fire.

The axon endings (Output Zone) almost touch the dendrites or cell body of the next neuron. Transmission of an electrical signal from one neuron to the next is effected by **neurotransmittors**, chemicals which are released from the first neuron and which bind to receptors in the second. This link is called a **synapse**. The extent to which the signal from one neuron is passed on to the next depends on many factors, e. g. the amount of neurotransmittor available, the

number and arrangement of receptors, amount of neurotransmittor reabsorbed.

The efficacy of a synapse can change as a result of experience, providing both memory and learning through **long-term potentiation (LTP)**. One way this happens is through release of more neurotransmitters. Many other changes may also be involved.

Hebbs Postulate: “When an axon of cell A... excites[s] cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells so that A’s efficiency as one of the cells firing B is increased”.

1.2 Artificial Neuron Models

To model the brain we need to model a neuron. Each neuron performs a simple computation. It receives signals from its input links and it uses these values to compute the activation level (or output) for the neuron. This value is passed to other neurons via its output links.

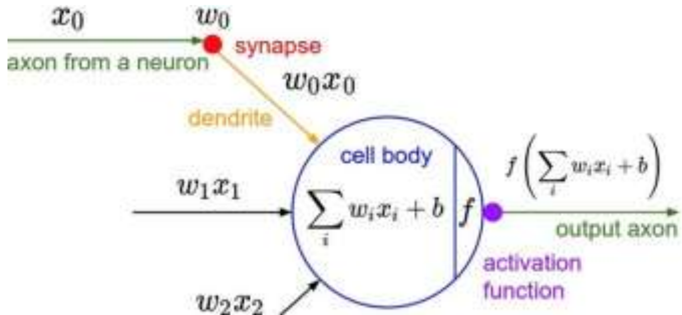


Figure 2 – Basic model of a single neuron

An artificial neuron is a simplistic representation that emulates the signal integration and threshold firing behavior of biological neurons by means of mathematical equations. Like their biological counterpart, artificial neurons are bound together by

connections that determine the flow of information between peer neurons. Stimuli are transmitted from one processing element to another via synapses or interconnections, which can be excitatory or inhibitory. If the input to a neuron is excitatory, it is more likely that this neuron will transmit an excitatory signal to the other neurons connected to it. Whereas an inhibitory input will most likely be propagated as inhibitory.

The input value received of a neuron is calculated by summing the weighted input values from its input links. An activation function takes the neuron input value

$$Sum = \sum_{i=1}^{n} x_i w_i$$

and produces a value y which becomes the output value of the neuron

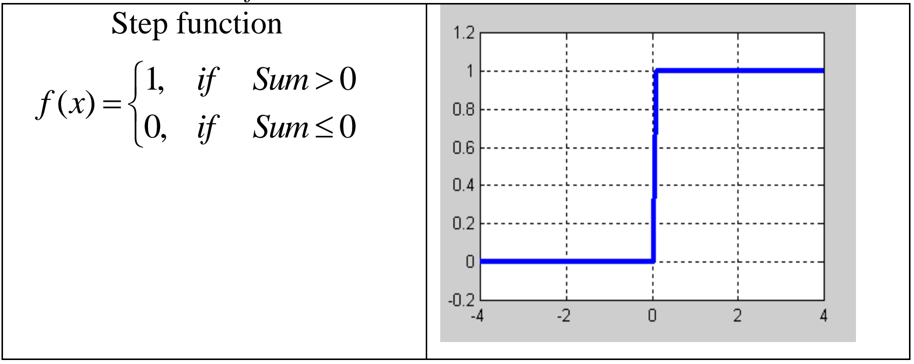
$$y = f\left(\sum_{i=0}^{n} x_i w_i\right).$$

This value is passed to other neurons in the network.

The activation function defines the output of that node given an input or set of inputs. A standard computer chip circuit can be seen as a digital network of activation functions that can be "ON" (1) or "OFF" (0), depending on input. This is similar to the behavior of the linear perceptron in neural networks.

Some common activation functions are shown below.

Table 1 – Activation functions



Step function $f(x) = \begin{cases} 1, & if \quad Sum > 0 \\ 0, & if \quad Sum \le 0 \end{cases}$	

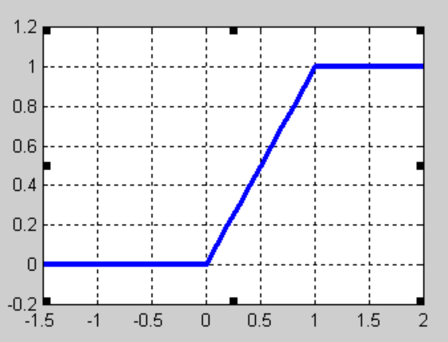
<table>
<tr><td>

Saturation function.

The activation value corresponds to the value of the weighted sum, where k is a constant only if this sum does not exceed a pre-defined MAX value.

$$f(x)=\begin{cases} k\cdot Sum, & if\ Sum < MAX \\ MAX, & otherwise \end{cases}$$

</td><td>



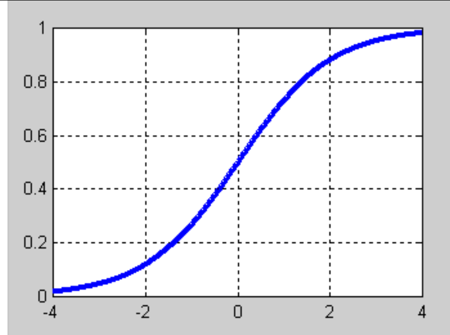
</td></tr>
<tr><td>

Sigmoid function

The saturation levels range from 0 to 1.

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

</td><td>



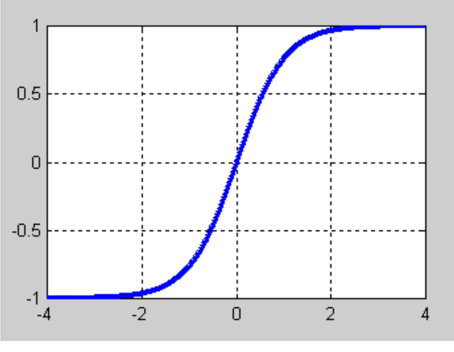
</td></tr>
<tr><td>

Hyperbolic tangent function.

The saturation levels range from –1 to 1.

$$f(x)=\frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$$

</td><td>



</td></tr>
</table>

On occasions an identifying function is also used (i. e. where the input to the neuron becomes the output). This function is

normally used in the input layer where the inputs to the neural network are passed unchanged into the network.

1.3 Historical background

Neural network simulations appear to be a recent development. However, this field was established before the advent of computers, and has survived at least one major setback and several eras.

In 1943, neurophysiologist Warren McCulloch and mathematician Walter Pitts wrote a paper on how neurons might work. In order to describe how neurons in the brain might work, they modelled a simple neural network using electrical circuits.

In 1949, Donald Hebb wrote The Organization of Behaviour, a work which pointed out the fact that neural pathways are strengthened each time they are used, a concept fundamentally essential to the ways in which humans learn. If two nerves fire at the same time, he argued, the connection between them is enhanced.

As computers became more advanced in the 1950’s, it was finally possible to simulate a hypothetical neural network. The first step towards this was made by Nathanial Rochester from the IBM research laboratories. Unfortunately for him, the first attempt to do so failed.

In 1959, Bernard Widrow and Marcian Hoff of Stanford developed models called “ADALINE” and “MADALINE”. In a typical display of Stanford’s love for acronymns, the names come from their use of Multiple ADAptive LINear Elements. ADALINE was developed to recognize binary patterns so that if it was reading streaming bits from a phone line, it could predict the next bit. MADALINE was the first neural network applied to a real world problem, using an adaptive filter that eliminates echoes on phone lines. While the system is as ancient as air traffic control systems, like air traffic control systems, it is still in commercial use.

In 1962, Widrow & Hoff developed a learning procedure that examines the value before the weight adjusts it (i. e. 0 or 1)

according to the rule: Weight Change = (Pre-Weight line value)· (Error / (Number of Inputs)). It is based on the idea that while one active perceptron may have a big error, one can adjust the weight values to distribute it across the network, or at least to adjacent perceptrons. Applying this rule still results in an error if the line before the weight is 0, although this will eventually correct itself. If the error is conserved so that all of it is distributed to all of the weights then the error is eliminated.

Many important advances have been boosted by the use of inexpensive computer emulations. Following an initial period of enthusiasm, the field survived a period of frustration and disrepute. During this period when funding and professional support was minimal, important advances were made by relatively few researchers. These pioneers were able to develop convincing technology which surpassed the limitations identified by Minsky and Papert. Minsky and Papert, published a book (in 1969) in which they summed up a general feeling of frustration (against neural networks) among researchers, and was thus accepted by most of them without further analysis.

In 1972, Kohonen and Anderson developed a similar network independently of one another, which we will discuss more about later. They both used matrix mathematics to describe their ideas but did not realize that what they were doing was creating an array of analog ADALINE circuits. The neurons are supposed to activate a set of outputs instead of just one. The first multilayered network was developed in 1975, an unsupervised network. Currently, the neural network field enjoys a resurgence of interest and a corresponding increase in funding.

1.4 Neural networks versus conventional computers

Neural networks take a different approach to problem solving than that of conventional computers. Conventional computers use an algorithmic approach, i. e. the computer follows a set of instructions

in order to solve a problem. Unless the specific steps that the computer needs to follow are known the computer cannot solve the problem. That restricts the problem solving capability of conventional computers to problems that we already understand and know how to solve. But computers would be so much more useful if they could do things that we don't exactly know how to do.

Neural networks process information in a similar way the human brain does. The network is composed of a large number of highly interconnected processing elements (neurones) working in parallel to solve a specific problem. Neural networks learn by example. They cannot be programmed to perform a specific task. The examples must be selected carefully otherwise useful time is wasted or even worse the network might be functioning incorrectly. The disadvantage is that because the network finds out how to solve the problem by itself, its operation can be unpredictable.

On the other hand, conventional computers use a cognitive approach to problem solving; the way the problem is solved must be known and stated in small unambiguous instructions. These instructions are then converted to a high level language program and then into machine code that the computer can understand. These machines are totally predictable; if anything goes wrong is due to a software or hardware fault.

Neural networks and conventional algorithmic computers are not in competition but complement each other. There are tasks more suitable to an algorithmic approach like arithmetic operations and tasks that are more suitable to neural networks. Even more, a large number of tasks require systems that use a combination of both approaches (normally a conventional computer is used to supervise the neural network) in order to perform at maximum efficiency.

Lecture 2

Artificial Neuron Networks (ANNs)

2.1 Learning problem

Suppose you need to find the function which takes the following inputs and produces their associated outputs:

Input	Output
1	1
2	8
3	27
4	64

Presumably, the function you would learn would be $f(x) = x^3$. Imagine now that you had a set of values, rather than a single instance as input to your function:

Input	Output
[1, 2, 3]	1
[2, 3, 4]	5
[3, 4, 5]	11
[4, 5, 6]	19

Here, it is still possible to learn a function: for example, multiply the first and last element and take the middle one from the product. Note that the functions we are learning are getting more complicated, but they are still mathematical. ANNs just take this further: the functions they learn are generally so complicated that it's difficult to understand them on a global level. But they are still just functions that play around with numbers.

Imagine, now, for example, that the inputs to our function were arrays of pixels, actually taken from photographs of vehicles, and that the output of the function is either 1, 2 or 3, where 1 stands for a motorcycle, 2 stands for a bus and 3 stands for a tank:

Input	Output	Input	Output
	3		1
	2		1

In this case, the function which takes an array of integers representing pixel data and outputs either 1, 2 or 3 will be fairly complicated, but it's just doing the same kind of thing as two simpler functions.

Because the functions learned to, for example, categorise photos of vehicles into a category of motorcycle, bus or tank, are so complicated, we say the ANN approach is a black box approach because, while the function performs well at its job, we cannot look inside it to gain a knowledge of how it works. This is a little unfair, as there are some projects which have addressed the problem of translating learned neural networks into human readable forms. However, in general, ANNs are used in cases where the predictive accuracy is of greater importance than understanding the learned concept.

Artificial Neural Networks consist of a number of units which are mini calculation devices. They take in real-valued input from

other multiple nodes and they produce a single real valued output. By real-valued input and output we mean real numbers which are able to take any decimal value. The architecture of ANNs is as follows:

- A set of input units which take in information about the example to be propagated through the network. By propagation, we mean that the information from the input will be passed through the network and an output produced. The set of input units forms what is known as the input layer.
- A set of hidden units which take input from the input layer. The hidden units collectively form the hidden layer. For simplicity, we assume that each unit in the input layer is connected to each unit of the hidden layer, but this isn't necessarily the case. A weighted sum of the output from the input units forms the input to every hidden unit. Note that the number of hidden units is usually smaller than the number of input units.
- A set of output units which, in learning tasks, dictate the category assigned to an example propagated through the network. The output units form the output layer. Again, for simplicity, we assume that each unit in the hidden layer is connected to each unit in the output layer. A weighted sum of the output from the hidden units forms the input to every output unit.

Hence ANNs look like this in the general case:

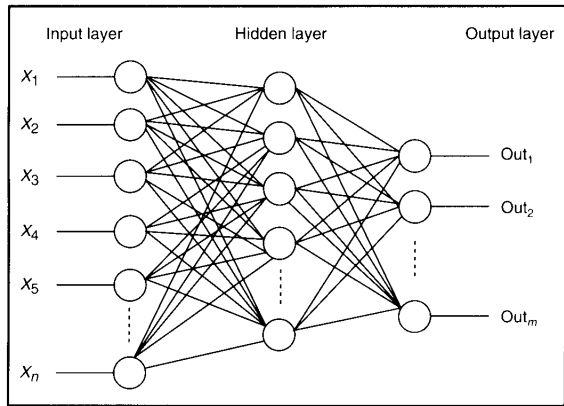


Figure 3 – Structure of an ANN

Note that the x_1, x_2, … x_n and Out_1,…, Out_2 represent real values and that all the edges in this graph have weights associated with them. Note also that more complicated ANNs are certainly possible. In particular, many ANNs have multiple hidden layers, with the output from one hidden layer forming the input to another hidden layer. Also, ANNs with no hidden layer – where the input units are connected directly to the output units – are possible. These tend to be too simple to use for real world learning problems, but they are useful to study for illustrative purposes, and we look at the simplest kind of neural networks, perceptron's, in the next section.

In our vehicle example, it is likely that all the images will be normalised to having the same number of pixels. Then there may be an input unit for each red, green and blue intensity for each pixel. Alternatively, greyscale images may be used, in this case there needs only to be an input node for each pixel, which takes in the brightness of the pixel. The hidden layer is likely to contain far fewer units (probably between 3 and 10) than the number of input units. The output layer will contain three units, one for each of the possible categories (motorcycle, bus, tank). Then, when the pixel data for an image are given as the initial values for the input units, this information will propagate through the network and all three output units will produce a real value. The output unit which produces the highest value is taken as the categorization for the input image.

So, for instance, when this image is used as input:

then, if output unit 1 [motorcycle] produces value 0.5, output unit 2 [bus] produces value 0.05 and output unit 3 [tank] produces value 0.1, then this image has been (correctly) classified as a motorcycle, because the output from the corresponding motorcycle output unit is

higher than for the other two. Exactly as the function embedded within a neural network computes the outputs given by the inputs is best explained using example networks. In the next section, we'll consider the networks simplest of all, perceptrons, which consist of a set of input units connected to a single output unit.

2.2 Perceptrons

The weights in any ANN are always just real numbers and the learning problem boils down to choosing the best value for each weight in the network. This means there are two important decisions to make before we train an artificial neural network: (i) the overall architecture of the system (how input nodes represent given examples, how many hidden units/hidden layers to have and how the output information will give us an answer) and (ii) how the units calculate their real value output from the weighted sum of real valued inputs.

The answer to (i) is usually found by experimentation with respect to the learning problem at hand: different architectures are tried and evaluated on the learning problem until the best one emerges. In perceptrons, given that we have no hidden layer, the architecture problem boils down to just specifying how the input units represent the examples given to the network. The answer to (ii) is discussed in the next subsection.

The input units simply output the value which was input to them from the example to be propagated. Every other unit in a network normally has the same internal calculation function, which takes the weighted sum of inputs to it and calculates an output. There are different possibilities for the unit function and this dictates to some extent how learning over networks of that type is performed. Firstly, there is a simple linear unit which does no calculation, it just outputs the weighted sum which was input to it.

Secondly, there are other unit functions which are called threshold functions, because they are set up to produce low values up until the weighted sum reaches a particular threshold, then they produce high values after this threshold. The simplest type of the threshold function produces 1 if the weighted sum of the inputs is over a threshold value T, and produces a –1 otherwise. We call such functions step functions, due to the fact that, when drawn as a graph, it looks like a step. Another type of the threshold function is called a sigma function, which has similarities with the step function, but is advantageous over it.

Example

As an example, consider an ANN which has been trained to learn the following rule categorizing the brightness of 2 x 2 black and white pixel images: if it contains 3 or 4 black pixels, it is dark; if it contains 2, 3 or 4 white pixels, it is bright. We can model this with a perceptron by saying that there are 4 input units, one for each pixel, and they output +1 if the pixel is white and –1 if the pixel is black. Also, the output unit produces 1 if the input example is to be categorized as bright and –1 if the example is dark. If we choose the weights as in the following diagram, the perceptron will perfectly categorize any image of four pixels into dark or light according to our rule:

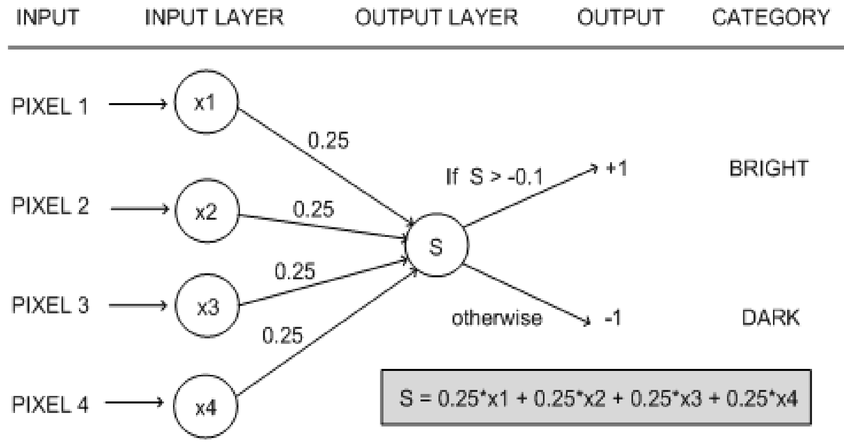


Figure 4 – The rule to categorize an image of four pixels

We see that, in this case, the output unit has a step function, with the threshold set to –0.1. Note that the weights in this network are all the same, which is not true in the general case. Also, it is convenient to make the weights going in to a node add up to 1, so that it is possible to compare them easily. The reason this network perfectly captures our notion of darkness and lightness is because, if three white pixels are input, then three of the input units produce +1 and one input unit produces –1. This goes into the weighted sum, giving a value of

$$S = 0.25 \cdot 1 + 0.25 \cdot 1 + 0.25 \cdot 1 + 0.25 \cdot (-1) = 0.5.$$

As this is greater than the threshold of –0.1, the output node produces +1, which relates to our notion of a bright image. Similarly, four white pixels will produce a weighted sum of 1, which is greater than the threshold, and two white pixels will produce a sum of 0, also greater than the threshold. However, if there are three black pixels, S will be –0.5, which is below the threshold, hence the output node will output –1, and the image will be categorised as dark. Similarly, an image with four black pixels will be categorised as dark. As an exercise: keeping the weights the same, how low would the threshold have to be in order to misclassify an example with three or four black pixels?

Learning Weights in Perceptrons

We will look in detail at the learning method for weights in multi-layer networks in the next lecture. The following description of learning in perceptrons will help clarify what is going on in the multi-layer case. We are in a machine learning setting, so we can expect the task to learn a target function which categorises examples into categories, given (at least) a set of training examples supplied with their correct categorisations. A little thought will be needed in order to choose the correct way of thinking about the examples as input to a set of input units, but, due to the simple nature of a perceptron, there isn't much choice for the rest of the architecture.

In order to produce a perceptron able to perform our categorisation task, we need to use the examples to train the weights between the input units and the output unit, and to train the threshold. To simplify the routine, we think of the threshold as a special weight, which comes from a special input node that always outputs 1. So, we think of our perceptron like this:

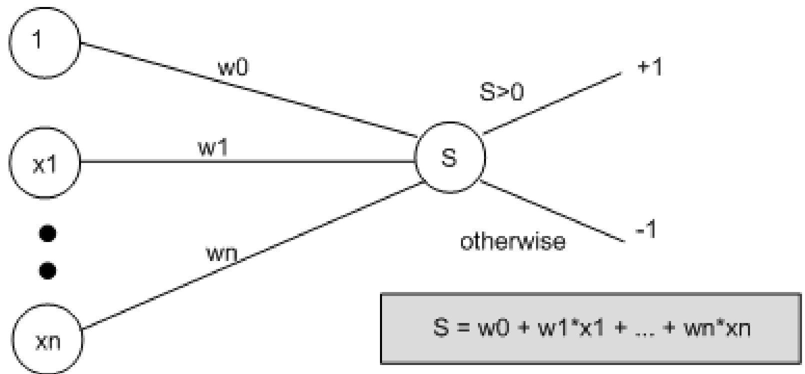


Figure 5 – Structure of perceptron

Then, we say that the output from the perceptron is +1 if the weighted sum from all the input units (including the special one) is greater than zero, and it outputs –1 otherwise. We see that weight w_0 is simply the threshold value. However, thinking of the network like this means we can train w_0 in the same way as we train all the other weights.

The weights are initially assigned randomly and training examples are used one after another to tweak the weights in the network. All the examples in the training set are used and the whole process (using all the examples again) is iterated until all examples are correctly categorised by the network. The tweaking is known as the perceptron training rule, and is as follows: If the training example, *E*, is correctly categorised by the network, then no tweaking is carried out. If *E* is mis-classified, then each weight is tweaked by adding on a small value, Δ. Suppose we are trying to calculate weight w_i, which is between the *i*th input unit, x_i and the

output unit. Then, given that the network should have calculated the target value $t(E)$ for example E, but actually calculated the observed value $o(E)$, then Δ is calculated as:

$$\Delta = \eta\ (t(E) - o(E))x_i\,.$$

Note that η is a fixed positive constant called the learning rate. Ignoring η briefly, we see that the value Δ that we add on to our weight w_i is calculated by multiplying the input value x_i by $t(E) - (E)$. $t(E) - o(E)$ will either be +2 or –2, because perceptrons output only +1 or –1, and $t(E)$ cannot be equal to $o(E)$, otherwise we wouldn't be doing any tweaking. So, we can think of $t(E) - o(E)$ as a movement in a particular numerical direction, i. e., positive or negative. This direction will be such that, if the overall sum, S, was too low to get over the threshold and produce the correct categorisation, then the contribution to S from $w_i x_i$ will be increased. Conversely, if S is too high, the contribution from $w_i x_i$ is reduced. Because $t(E) - o(E)$ is multiplied by x_i, then if x_i is a big value (positive or negative), the change to the weight will be greater. To get a better feel for why this direction correction works, it's a good idea to do some simple calculations by hand.

The learning rate η simply controls how far the correction should go at one time, and is usually set to be a fairly low value, e. g., 0.1. The weight learning problem can be seen as finding the global minimum error, calculated as the proportion of miscategorised training examples, over a space where all the input values can vary. Therefore, it is possible to move too far in a direction and improve one particular weight to the detriment of the overall sum: while the sum may work for the training example being looked at, it may no longer be a good value for categorizing all the examples correctly. For this reason, η restricts the amount of movement possible. If a large movement is actually required for a weight, then this will happen over a series of iterations through the example set. Sometimes, η is set to decay as the number of such iterations through the whole set of training examples increases, so that it can move more slowly towards the global minimum in order not to overshoot

in one direction. This kind of gradient descent is at the heart of the learning algorithm for multi-layered networks, as discussed in the next lecture.

Perceptrons with step functions have limited abilities when it comes to the range of concepts that can be learned. One way to improve matters is to replace the threshold function with a linear unit, so that the network outputs a real value, rather than a 1 or –1. This enables us to use another rule, called the delta rule, which is also based on gradient descent. We don't look at this rule here, because the backpropagation learning method for multi-layer networks is similar.

Worked Example

Suppose we are trying to learn a perceptron to represent the brightness rules above, in such a way that if it outputs 1, the image is categorised as bright, and if it outputs –1, the image is categorised as dark. Remember that we said a 2 x 2 black and white pixel image is categorised as bright if it has two or more white pixels in it. We shall call the pixels *p*1 to *p*4, with the numbers going from left to right, top to bottom in the 2 x 2 image. A black pixel will produce an input of –1 to the network, and a white pixel will give an input of +1.

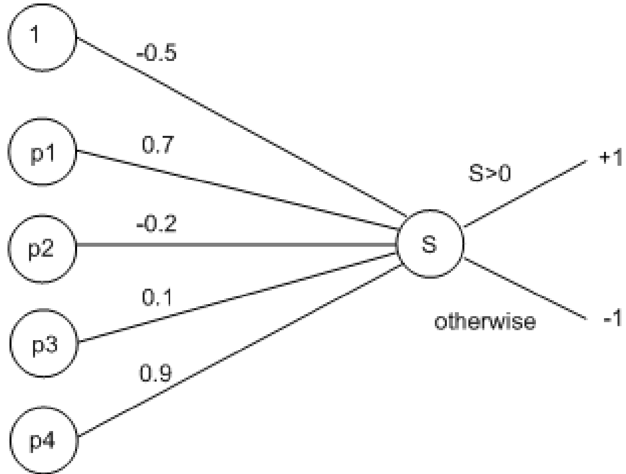
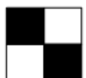


Figure 6 – Initial state of perceptron

Given our new way of thinking about the threshold as a weight from a special input node, our network will have five input nodes and five weights. Suppose also that we have assigned the weights randomly to values between –1 and 1, namely –0.5, 0.7, –0.2, 0.1 and 0.9. Then our perceptron will initially look like in Figure 6.

We will now train the network with the first training example, using a learning rate of $\eta = 0.1$. Suppose the first example image, E, is this:



With two white squares, this is categorized as bright. Hence, the target output for E is: $t(E) = +1$. Also, $p1$ (top left) is black, so the input x_1 is –1. Similarly, x_2 is +1, x_3 is +1 and x_4 is –1. Hence, when we propagate this through the network, we get the value:

$$S = (-0.5 \cdot 1) + (0.7 \cdot (-1)) + (-0.2 \cdot 1) + (0.1 \cdot 1) + (0.9 \cdot (-1)) = \\ = -2.2$$

As this value is less than zero, the network outputs $o(E) = -1$, which is not the correct value. This means that we should now tweak the weights in light of the incorrectly categorized example. Using the perception training rule, we need to calculate the value of Δ to add on to each weight in the network. Plugging values into the formula for each weight gives us:

$$\Delta_0 = \eta\ (t(E) - o(E))x_i = 0.1 \cdot (1 - (-1)) \cdot (1) = 0.1 \cdot 2 = 0.2$$
$$\Delta_1 = 0.1 \cdot (1 - (-1)) \cdot (-1) = 0.1 \cdot (-2) = -0.2$$
$$\Delta_2 = 0.1 \cdot (1 - (-1)) \cdot (1) = 0.1 \cdot 2 = 0.2$$
$$\Delta_3 = 0.1 \cdot (1 - (-1)) \cdot (1) = 0.1 \cdot 2 = 0.2$$
$$\Delta_4 = 0.1 \cdot (1 - (-1)) \cdot (-1) = 0.1 \cdot (-2) = -0.2$$

When we add these values on to our existing weights, we get the new weights for the network as follows:

$$w'_0 = -0.5 + \Delta_0 = -0.5 + 0.2 = -0.3$$
$$w'_1 = 0.7 + \Delta_1 = 0.7 + (-0.2) = 0.5$$
$$w'_2 = -0.2 + \Delta_2 = -0.2 + 0.2 = 0$$
$$w'_3 = 0.1 + \Delta_3 = 0.1 + 0.2 = 0.3$$
$$w'_4 = 0.9 + \Delta_4 = 0.9 - 0.2 = 0.7$$

Our newly trained network will now look like this:

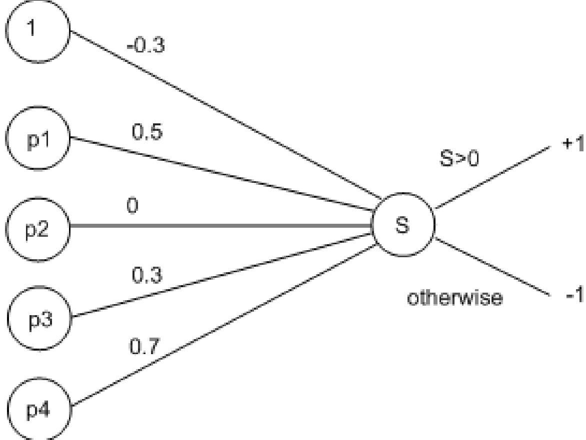


Figure 7 – Updated state of the perceptron

To see how this has improved the situation with respect to the training example, we can propagate it through the network again. This time, we get the weighted sum to be:

$$S = (-0.3 \cdot 1) + (0.5 \cdot (-1)) + (0 \cdot 1) + (0.3 \cdot 1) + (0.7 \cdot (-1)) = -1.2$$

This is still negative, and hence the network categorizes the example as dark, when it should be light. However, it is less negative. We can see that, by repeatedly training using this example,

the training rule would eventually bring the network to a state where it would correctly categorise this example.

The Learning Abilities of Perceptrons

Computational learning theory is the study of what concepts particular learning schemes (representation and method) can and can't learn. We don't look at this in detail, but a famous example, first highlighted in a very influential book by Minsky and Papert involves perceptrons. It has been mathematically proven that the above mentioned method for learning perceptron weights will converge to a perfect classifier for learning tasks where the target concept is linearly separable.

To understand what is and what isn't a linearly separable target function, we look at the simplest functions of all, boolean functions. These take two inputs, which are either 1 or –1 and output either 1 or –1. Note that, in other contexts, the values 0 and 1 are used instead of –1 and 1. As an example function, the AND boolean function outputs a 1 only if both inputs are 1, whereas the OR function only outputs a 1 if either inputs are 1. Obviously, these relate to the connectives we studied in the first logic order. The following two perceptrons can represent the AND and OR boolean functions respectively:

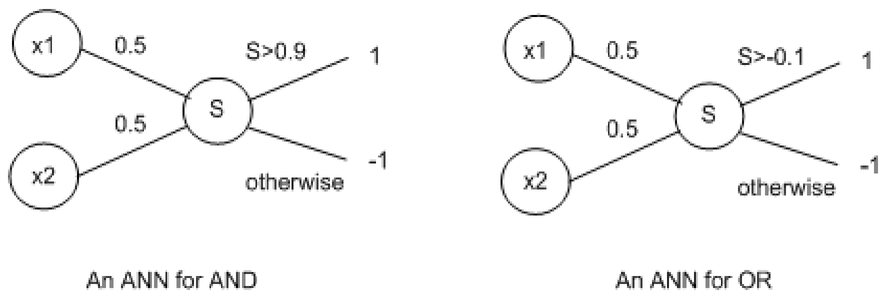


Figure 8 – Representation of AND and OR boolean functions

One of the major impacts of Minsky and Papert's book was to highlight the fact that perceptrons cannot learn a particular boolean function called XOR. This function outputs 1 if the two inputs are not the same. To see why XOR cannot be learned, try and write down a perceptron to do the job.

We've plotted the values taken by the boolean function when the inputs are particular values: (–1, –1); (1, –1); (–1, 1) and (1, 1). For the AND function, there is only one place where a 1 is plotted, namely when both inputs are 1. This meant that we could draw the dotted line to separate the output –1s from the 1s. We were able to draw a similar line in the OR case. Because we can draw these lines, we say that these functions are linearly separable. Note that it is not possible to draw such a line for the XOR plot: wherever you try, you never get a clean split into 1s and –1s.

The following diagram highlights the notion of linear separability in boolean functions, which explains why they can't be learned by perceptrons:

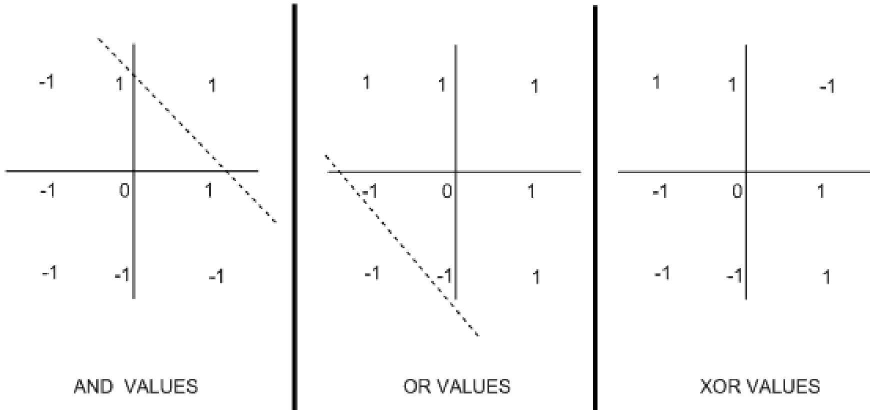


Figure 9 – Linear separability in boolean functions

The dotted lines can be seen as the threshold in perceptrons: if the weighted sum, S, falls below it, then the perceptron outputs one value, and if S falls above it, the alternative output is produced. It doesn't matter how the weights are organised, the threshold will still

be a line on the graph. Therefore, functions which are not linearly separable cannot be represented by perceptrons.

Note that this result extends to functions over any number of variables, which can take in any input, but which produce a boolean output (and hence could, in principle be learned by a perceptron).

Unfortunately, the disclosure in Minsky and Papert's book that perceptrons cannot learn even such a simple function was taken the wrong way: people believed it represented a fundamental flaw in the use of ANNs to perform learning tasks. This led to a winter of ANN research within AI, which lasted over a decade. In reality, perceptrons were being studied in order to gain insights into more complicated architectures with hidden layers, which do not have the limitations that perceptrons have. No one ever suggested that perceptrons would be eventually used to solve real world learning problems. Fortunately, people studying ANNs within other sciences (notably neuro-science) revived interest in the study of ANNs. For more details of computational learning theory, see chapter 7 of Tom Mitchell's machine learning book.

Lecture 3

Multi-Layer Artificial Neural Networks

We can now look at more sophisticated ANNs, which are known as multi-layer artificial neural networks because they have hidden layers. These will naturally be used to undertake more complicated tasks than perceptrons. We first look at the network structure for multi-layer ANNs, and then in detail at the way in which the weights in such structures can be determined to solve machine learning problems. There are many considerations involved with learning such ANNs, and we consider some of them here. First and foremost, the algorithm can get stuck in local minima, and there are some ways to try to get around this. As with any learning technique, we will also consider the problem of overfitting, and discuss which types of problems an ANN approach is suitable for.

3.1 Multi-Layer Network Architectures

We saw in the previous lecture that perceptrons have limited scope in the type of concepts they can learn – they can only learn linearly separable functions. However, we can think of constructing larger networks by building them out of perceptrons. In such larger networks, we call the step function units the perceptron units in multi-layer networks.

As with individual perceptrons, multi-layer networks can be used for learning tasks. However, the learning algorithm that we look at (the backpropagation routine) is derived mathematically, using differential calculus. The derivation relies on having a differentiable threshold function, which effectively rules out using perceptron units if we want to be sure that backpropagation works correctly. The step function in perceptrons is not continuous, hence non-differentiable. An alternative unit was therefore chosen which

had similar properties to the step function in perceptron units, but which was differentiable. There are many possibilities, one of which is sigmoid units, as described below.

Sigmoid units

Remember that the function inside units takes as input the weighted sum, S, of the values coming from the units connected to it. The function inside sigmoid units calculates the following value, given a real-valued input S:

$$\sigma(Sum) = \frac{1}{1+e^{-Sum}}.$$

When we plot the output from sigmoid units given various weighted sums as input, it looks remarkably like a step function:

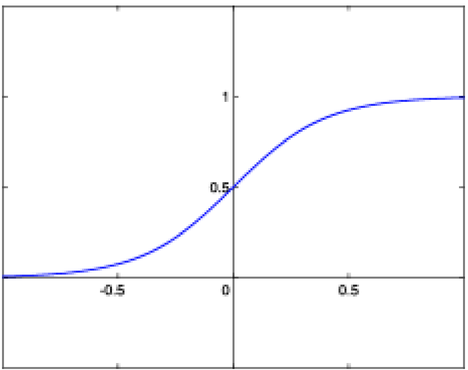


Figure 10 – Sigmoid step function

Of course, getting a differentiable function which looks like the step function was the whole point of the exercise. In fact, not only is this function differentiable, but the derivative is fairly simply expressed in terms of the function itself:

$$\frac{d\sigma(S)}{dS} = \sigma(S)(1-\sigma(S)).$$

Note that the output values for the σ function range between but never make it to 0 and 1. This is because e^{-S} is never negative, and the denominator of the fraction tends to 0 as S gets very big in the negative direction, and tends to 1 as it gets very big in the positive direction. This tendency happens fairly quickly: the middle ground between 0 and 1 is rarely seen because of the sharp (near) step in the function. Because of it looking like a step function, we can think of it firing and not-firing as in a perceptron: if a positive real is input, the output will generally be close to +1 and if a negative real is input the output will generally be close to –1.

Example of Multi-layer ANN with Sigmoid Units

We will concern ourselves here with ANNs containing only one hidden layer, as this makes describing the backpropagation routine easier. Note that networks where you can feed in the input on the left and propagate it forward to get an output are called feed forward networks. Below is such an ANN, with two sigmoid units in the hidden layer. The weights have been set arbitrarily between all the units.

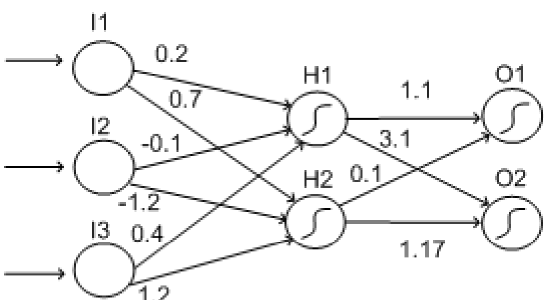


Figure 11 – Multi-layer ANN

Note that the sigma units have been identified with sigma signs in the node on the graph. As we did with perceptrons, we can give this network an input and determine the output. We can also look to see which units "fired", i. e., had a value closer to 1 than to 0.

Suppose we input the values 10, 30, 20 into the three input units, from top to bottom. Then the weighted sum coming into H1 will be:

$$S_{H1} = (0.2 \cdot 10) + (-0.1 \cdot 30) + (0.4 \cdot 20) = 7.$$

Then the σ function is applied to SH1 to give:

$$\sigma(S_{H1}) = 1/(1 + e^{-7}) = 1/(1 + 0.000912) = 0.999.$$

Similarly, the weighted sum coming into H2 will be:

$$S_{H2} = (0.7 \cdot 10) + (-1.2 \cdot 30) + (1.2 \cdot 20) = -5,$$

and σ applied to S_{H2} gives:

$$\sigma(S_{H2}) = 1/(1 + e^{5}) = 1/(1 + 148.4) = 0.0067.$$

From this, we can see that H1 has fired, but H2 has not. We can now calculate that the weighted sum going in to output unit O1 will be:

$$S_{O1} = (1.1 \cdot 0.999) + (0.1 \cdot 0.0067) = 1.0996,$$

and the weighted sum going in to output unit O2 will be:

$$S_{O2} = (3.1 \cdot 0.999) + (1.17 \cdot 0.0067) = 3.1047.$$

The output sigmoid unit in O1 will now calculate the output values from the network for O1:

$$\sigma(S_{O1}) = 1/(1 + e^{-1.0996}) = 1/(1+0.333) = 0.750,$$

and the output from the network for O2:

$$\sigma(S_{O2}) = 1/(1 + e^{-3.1047})= 0.957.$$

Therefore, if this network represented the learned rules for a categorization problem, the input triple (10, 30, 20) would be categorized into the category associated with O2, because this has the larger output.

3.2 The Backpropagation Learning Routine

As with perceptrons, the information in the network is stored in the weights, so the learning problem comes down to the question: how do we train the weights to best categorize the training examples. We then hope that this representation provides a good way to categorize unseen examples.

In outline, the backpropagation method is the same as for perceptrons:

- We choose and fix our architecture for the network, which will contain input, hidden and output units, all of which will contain sigmoid functions.
- We randomly assign the weights between all the nodes. The assignments should be to small numbers, usually between –0.5 and 0.5.
- Each training example is used, one after another, to re-train the weights in the network. The way this is done is given in detail below.
- After each epoch (run through all the training examples), a termination condition is checked (also detailed below). Note that, for this method, we are not guaranteed to find weights which give the network the global minimum error, i. e., perfectly correct categorization of the training examples. Hence the termination condition may have to be in terms of a (possibly small) number of mis-categorizations. We see later that this might not be such a good idea, though.

Weight Training Calculations

Because we have more weights in our network than in perceptrons, we firstly need to introduce the notation: w_{ij} to specify the weight between unit i and unit j. As with perceptrons, we will calculate a value Δ_{ij} to add on to each weight in the network after an example has been tried. To calculate the weight changes for a particular example, E, we first start with the information about how the network should perform for E. That is, we write down the target values $t_i(E)$ that each output unit O_i should produce for E. Note that, for categorization problems, $t_i(E)$ will be zero for all the output units except one, which is the unit associated with the correct categorization for E. For that unit, $t_i(E)$ will be 1.

Next, example E is propagated through the network so that we can record all the observed values $o_i(E)$ for the output nodes O_i. At the same time, we record all the observed values $h_i(E)$ for the hidden nodes. Then, for each output unit O_k, we calculate its error term as follows:

$$\delta_{O_k} = o_k(E)(1 - o_k(E))(t_k(E) - o_k(E)).$$

The error terms from the output units are used to calculate error terms for the hidden units. In fact, this method gets its name because we propagate this information backwards through the network. For each hidden unit H_k, we calculate the error term as follows:

$$\delta_{H_k} = h_k(E)(1 - h_k(E)) \sum_{i \in outputs} \omega_{ki} \delta_{O_i} .$$

This means that we take the error term for every output unit and multiply it by the weight from hidden unit H_k to the output unit. We then add all these together and multiply the sum by factor $h_k(E)(1 - h_k(E))$.

Having calculated all the error values associated with each unit (hidden and output), we can now transfer this information into the weight changes Δ_{ij} between units i and j. The calculation is as follows: for weights w_{ij} between input unit I_i and hidden unit H_j, we add on:

$$\Delta_{ij} = \eta \delta_{H_j} x_i .$$

Remembering that x_i is the input to the ith input node for example E; that η is a small value known as the learning rate and that δ_{Hj} is the error value we calculated for hidden node H_j using the formula above.

For weights w_{ij} between hidden unit H_i and output unit O_j, we add on:

$$\Delta_{ij} = \eta \delta_{O_j} h_i(E) .$$

Remembering that $h_i(E)$ is the output from hidden node H_i when example E is propagated through the network, and that δ_{Oj} is the error value we calculated for output node Oj using the formula above.

Each alteration Δ is added to the weights and this concludes the calculation for example E. The next example is then used to tweak the weights further. As with perceptrons, the learning rate is used to ensure that the weights are only moved a short distance for each example, so that the training for previous examples is not lost. Note that the mathematical derivation for the above mentioned calculations is based on the derivative of σ that we saw above.

Worked Example

We will re-use the example from the previous section, where our network originally looked like this:

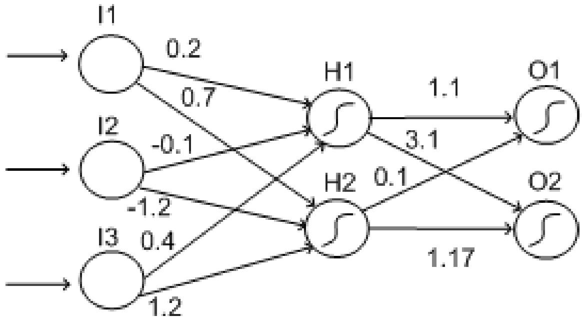


Figure 12 – Initial state of ANN

and we propagated the values (10, 30, 20) through the network. When we did so, we observed the following values:

Table 2 – The state of ANN in the first step

Input units		**Hidden units**			**Output units**		
Unit	**Output**	**Unit**	**Weighted Sum Input**	**Output**	**Unit**	**Weighted Sum Input**	**Output**
I1	10	H1	7	0.999	O1	1.0996	0.750
I2	20	H2	–5	0.0067	O2	3.1047	0.957
I3	30						

Suppose now that the target categorization for the example was the one associated with O1. This means that the network mis-categorizes the example and gives us an opportunity to demonstrate the backpropagation algorithm: we will update the weights in the network according to the weight training calculations provided above, using a learning rate of $\eta = 0.1$.

If the target categorization was associated with O1, this means that the target output for O1 was 1, and the target output for O2 was 0. Hence, using the above notation,

$t1(E) = 1; \quad t2(E) = 0; \quad o1(E) = 0.750; \quad o2(E) = 0.957.$

That means we can calculate the error values for the output units O1 and O2 as follows:

$$\delta O1 = o1(E)(1 - o1(E))(t1(E) - o1(E)) = \\ = 0.750(1 - 0.750)(1 - 0.750) = 0.0469;$$

$$\delta O2 = o2(E)(1 - o2(E))(t2(E) - o2(E)) = \\ = 0.957(1 - 0.957)(0 - 0.957) = -0.0394.$$

We can now propagate this information backwards to calculate the error terms for the hidden nodes H1 and H2. To do this for H1, we multiply the error term for O1 by the weight from H1 to O1, then add this to the multiplication of the error term for O2 and the weight between H1 and O2. This gives us:

$$(1.1 \cdot 0.0469) + (3.1 \cdot (-0.0394)) = -0.0706.$$

To turn this into the error value for H1, we multiply by $h_1(E) \cdot (1 - h_1(E))$, where $h_1(E)$ is the output from H1 for example E, as recorded in the table above. This gives us:

$$\delta_{H1} = -0.0706 \cdot (0.999 \cdot (1 - 0.999)) = -0.0000705.$$

A similar calculation for H2 gives the first part to be: $(0.1 \cdot 0.0469) + (1.17 \cdot (-0.0394)) = -0.0414$, and the overall error value to be:

$$\delta_{H2} = -0.0414 \cdot (0.067 \cdot (1 - 0.067)) = -0.00259.$$

We now have all the information required to calculate the weight changes for the network. We will deal with the 6 weights between the input units and the hidden units first:

Table 3 – New weight of ANN between the input units and the hidden

Input unit	Hidden unit	η	δH	xi	$\Delta = \eta \cdot \delta_H \cdot x_i$	Old weight	New weight
I1	H1	0.1	–0.0000705	10	–0.0000705	0.2	0.1999295
I1	H2	0.1	–0.00259	10	–0.00259	0.7	0.69741
I2	H1	0.1	–0.0000705	30	–0.0002115	–0.1	–0.100211
I2	H2	0.1	–0.00259	30	–0.00777	–1.2	–1.20777
I3	H1	0.1	–0.0000705	20	–0.000141	1.4	0.39999
I3	H2	0.1	–0.00259	20	–0.00518	1.2	1.1948

We now turn to the problem of altering the weights between the hidden layer and the output layer. The calculations are similar, but instead of relying on the input values from *E*, they use the values calculated by the sigmoid functions in the hidden nodes: $h_i(E)$. The following table calculates the relevant values:

Table 4 – New weight of ANN between the hidden and the output layer

Hidden unit	Output unit	η	δ_O	$h_i(E)$	$\Delta = \eta \cdot \delta_O \cdot h_i$	Old weight	New weight
H1	O1	0.1	0.0469	0.999	0.000469	1.1	1.100469
H1	O2	0.1	–0.0394	0.999	–0.00394	3.1	3.0961
H2	O1	0.1	0.0469	0.0067	0.00314	0.1	0.10314
H2	O2	0.1	–0.0394	0.0067	–0.0000264	1.17	1.16998

We note that the weights haven’t altered all that much, so it might be a good idea in this situation to use a bigger learning rate. However, remember that, with sigmoid units, small changes in the weighted sum can produce big changes in the output from the unit.

As an exercise, check whether the re–trained network performs better with respect to the example than the original network.

Avoiding Local Minima

The error rate of multi-layered networks over a training set could be calculated as the number of mis-classified examples. Remembering, however, that there are many output nodes, all of which could potentially misfire (e. g., giving a value close to 1 when it should have output 0, and vice-versa), we can be more sophisticated in our error evaluation. In practice the overall network error is calculated as:

$$\frac{1}{2}\sum_{E\in examples}\left(\sum_{k\in outputs}\left(t_k(E)-o_k(E)\right)^2\right).$$

This is not as complicated as it first appears. The calculation simply involves working out the difference between the observed output for each output unit and the target output and squaring this to make sure it is positive, then adding up all these squared differences for each output unit and for each example.

Backpropagation can be seen as using a searching space of network configurations (weights) in order to find a configuration with the least error, measured in the above fashion. The more complicated network structure means that the error surface which is searched can have local minima, and this is a problem for multi-layer networks, and we look at ways around it below. Having said that, even if a learned network is in a local minima, it may still perform adequately, and multi-layer networks have been used to great effect in real world situations.

One way around the problem of local minima is to use random re-start as described in the lecture on search techniques. Different initial random weightings for the network may mean that it converges to different local minima, and the best of these can be

taken for the learned ANN. Alternatively, a "committee" of networks could be learned, with the (possibly weighted) average of their decisions taken as an overall decision for a given test example. Another alternative is to try and skip over some of the smaller local minima, as described below.

Adding Momentum

Imagine a ball rolling down a hill. As it does so, it gains momentum, so that its speed increases and it becomes more difficult to stop. As it rolls down the hill towards the valley floor (the global minimum), it might occasionally wander into local hollows. However, it may be that the momentum it has obtained keeps it rolling up and out of the hollow and back on track to the valley floor.

The crude analogy describes one heuristic technique for avoiding local minima, called adding momentum, funnily enough. The method is simple: for each weight remember the previous value of Δ which was added on to the weight in the last epoch. Then, when updating that weight for the current epoch, add on a little of the previous Δ. How small to make the additional extra is controlled by a parameter α called the momentum, which is set to a value between 0 and 1.

To see why this might help bypass local minima, note that if the weight change carries on in the direction it was going in the previous epoch, then the movement will be a little more pronounced in the current epoch. This effect will be compounded as the search continues in the same direction. When the trend finally reverses, then the search may be at the global minimum, in which case it is hoped that the momentum won't be enough to take it anywhere else than where it is. Alternatively, the search may be at a fairly narrow local minimum. In this case, even though the backpropagation algorithm dictates that Δ will change direction, it may be that the additional extra from the previous epoch (the momentum) may be enough to counteract this effect for a few steps. These few steps may be all that is needed to bypass the local minimum.

In addition to getting over some local minima, when the gradient is constant in one direction, adding momentum will increase the size of the weight change after each epoch, and the network may converge quicker. Note that it is possible to have cases where (a) the momentum is not enough to carry the search out of a local minima or (b) the momentum carries the search out of the global minima into a local minima. This is why this technique is a heuristic method and should be used somewhat carefully (it is used in practice a great deal).

Overfitting Considerations

Left unchecked, backpropagation in multi-layer networks can be highly susceptible to overfitting itself to the training examples. The following graph plots the error on the training and test set as the number of weight updates increases. It is typical of networks left to train unchecked.

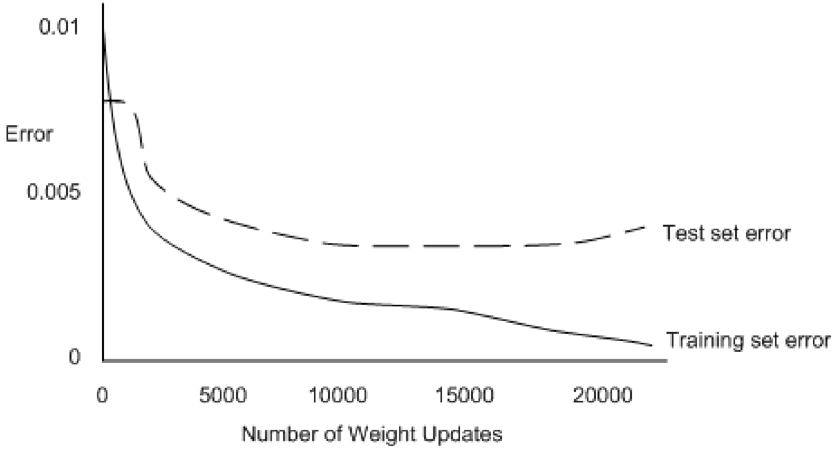


Figure 13 – The error versus number of weight updates

Alarmingly, even though the error on the training set continues to gradually decrease, the error on the test set actually begins to increase towards the end. This is clearly overfitting, and it relates to the network beginning to find and fine-tune to

idiosyncrasies in the data, rather than to general properties. Given this phenomenon, it would be unwise to use some kind of threshold for the error as the termination condition for backpropagation.

In cases where the number of training examples is high, one antidote to overfitting is to split the training examples into a set to train the weight and a set to hold back as an internal validation set. This is a mini-test set, which can be used to keep the network in check: if the error on the validation set reaches a minimum and then begins to increase, then it could be that overfitting is beginning to occur.

Note that (time permitting) it is worth giving the training algorithm the benefit of the doubt as much as possible. That is, the error in the validation set can also go through local minima, and it is not wise to stop training as soon as the validation set error starts to increase, as a better minima may be achieved later on. Of course, if the minimum is never bettered, then the network which is finally presented by the learning algorithm should be re-wound to be the one which produced the minimum on the validation set.

Another way around overfitting is to decrease each weight by a small weight decay factor during each epoch. Learned networks with large (positive or negative) weights tend to have overfitted the data, because larger weights are needed to accommodate outliers in the data. Hence, keeping the weights low with a weight decay factor may help to steer the network from overfitting.

Appropriate Problems for ANN learning

As we did for decision trees, it's important to know when ANNs are the right representation scheme for the job. The following are some characteristics of learning tasks for which artificial neural networks are an appropriate representation:

The concept (target function) to be learned can be characterized in terms of a real-valued function. That is, there is some translation from the training examples to a set of real numbers, and the output from the function is either real-valued or (if a categorization) can be mapped to a set of real values. It's important

to remember that ANNs are just giant mathematical functions, so are the data they play around with are numbers, rather than logical expressions, etc. This may sound restrictive, but many learning problems can be expressed in a way that ANNs can tackle them, especially as real numbers contain booleans (true and false mapped to +1 and -1), integers, and vectors of these data types can also be used.

Long training time is acceptable. Neural networks generally take a longer time to train than, for example, decision trees. Many factors, including the number of training examples, the value chosen for the learning rate and the architecture of the network, have an affect on the time required to train a network. Training time can vary from a few minutes to many hours.

It is not vitally important that humans should be able to understand exactly how the learned network carries out categorizations. As we discussed above, ANNs are black boxes and it is difficult for us to get a handle on what its calculations are doing.

When in use for the actual purpose it was learned for, the evaluation of the target function needs to be quick. While it may take a long time to learn a network to decide, for instance, whether a vehicle is a tank, bus or car, once the ANN has been learned, using it for the categorization task is typically very fast. This may be very important: if the network was to be used in a battle situation, then a quick decision about whether the object moving hurriedly towards it is a tank, bus, car or old lady could be vital.

In addition, neural network learning is quite robust to errors in the training data, because it is not trying to learn exact rules for the task, but rather to minimize an error function.

Lecture 4

Competitive Networks – the Kohonen Self-organizing Map

4.1 Self-organizing maps

The inspiration for many of these networks came from biology. They have been developed either to model some biological function (particularly in cognitive modelling) or in response to the demand for biological plausibility in neural networks. One important organizing principle of sensory pathways in the brain is that the placement of neurons is orderly and often reflects some physical characteristic of the external stimulus being sensed. For example, at each level of the auditory pathway, nerve cells and fibres are arranged anatomically in relation to the frequency which elicits the greatest response from each neuron. This topologic organization in the auditory pathway also extends up to the auditory cortex. Although much of this low-level organization is genetically pre-determined, it is likely that some of the organization at higher levels is created during learning by algorithms which promote self-organization. Kohonen took inspiration from this physical structure of the brain to produce self-organizing feature maps (topology preserving maps). In a self-organizing map, units located physically next to one another will respond to input vectors that are in some way 'next to one another'. Although it is easy to visualize units being next to one another in a two-dimensional array, it is not so easy to determine which classes of vectors are next to each other in a high-dimensional space. Large dimensional input vectors are in a sense 'projected down' onto the two-dimensional map in a way that maintains the natural order of the input vectors. This dimensional reduction can allow us to easily visualize important relationships among the data that otherwise may not have been noticed.

4.2 Learning in biological systems – the self-organizing paradigm

The type of learning utilized in multilayer perceptrons requires the correct response to be provided during training (supervised training).

Biological systems display this type of learning, but they are also capable of learning by themselves – without a supervisor showing the correct response (unsupervised learning).

A neural network with a similar capability is called a self-organizing system because during training, the network changes its weights to learn appropriate associations, without any right answers being provided.

The propagation of biological neural activation via axons can be modelled using a Mexican hat function:

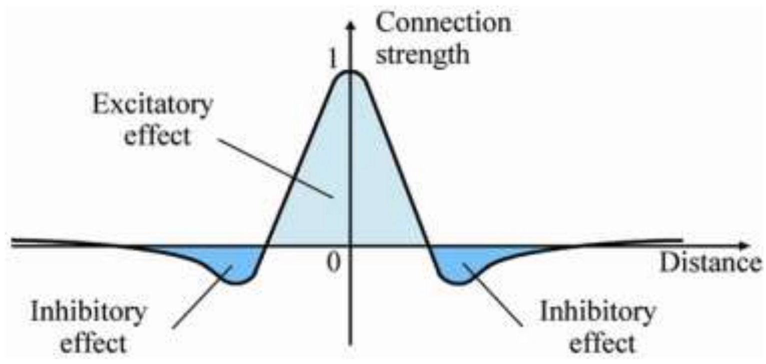


Figure 14 – Mexican hat function

Cells close to the active cell have excitatory links. The strengths of the links drop off with distance and then turn inhibitory. The Kohonen neural network also uses only locally connected neurons and restricts the adjustment of weight values to localized “neighbourhoods”.

4.3 Architecture of the Kohonen Network

Teuvo Kohonen was the originator of this type of self-organizing network. The aim of a Kohonen network is to produce a pattern classifier, which is self-organizing and uses a form of unsupervised learning to adjust the weights. Typically, a Kohonen network consists of a two-dimensional array of neurons with all of the inputs arriving at all of the neurons. Each neuron has its own set of weights which can be regarded as an exemplar pattern. When an input pattern arrives at the network, the neuron with the exemplar pattern that is most similar to the input pattern will give the largest response. One difference from other self-organizing systems, however, is that the exemplar patterns are stored in such a way that similar exemplars are to be found in neurons that are physically close to one another and exemplars that are very different are situated far apart. Self-Organizing Maps (SOMs) aim to produce a network where the weights represent the coordinates of some kind of topological system or map and the individual elements in the network are arranged in an ordered way.

The Kohonen network consists of an input layer, which distributes the inputs to each node in a second layer, the so-called competitive layer (feature map).

Each of the nodes on this layer acts as an output node.

Each neuron in the competitive layer is connected to other neurons in its neighbourhood and feedback is restricted to neighbors through these lateral connections.

Neurons in the competitive layer have excitatory connections to immediate neighbours and inhibitory connections to more distant neurons.

All neurons in the competitive layer receive a mixture of excitatory and inhibitory signals from the input layer neurons and from other competitive layer neurons.

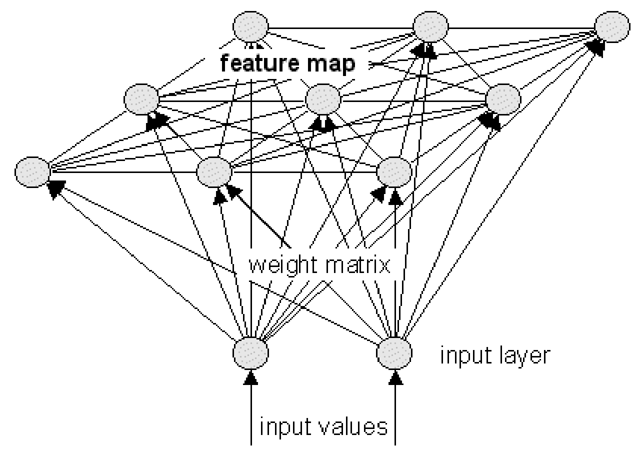


Figure 15 – Structure of Kohonen network

4.4 The Kohonen Network in Operation

As an input pattern is presented, some of the neurons are sufficiently activated to produce outputs which are fed back to other neurons in their neighbourhoods.

The node with the weight vector closest to the input pattern vector (the so-called "winning node") produces the largest output. During training, input weights of the winning neuron and its neighbors are adjusted to make them resemble the input pattern even more closely.

At the completion of training, the winning node ends up with its weight vector aligned with the input pattern and produces the strongest output whenever that particular pattern is presented.

The nodes in the winning node's neighbourhood also have their weights modified to settle down to an average representation of that pattern class. As a result, unseen patterns belonging to that class are also classified correctly (generalization).

The m neighbourhoods, corresponding to the m possible pattern classes are said to form a topological map representing the patterns.

The initial size of the neighbourhood mentioned above and the fixed values of excitatory (positive) and inhibitory (negative) weights to neurons in the neighbourhood are among the design decisions to be made.

Derivation of the learning rule for the Kohonen net

The sum squared error for pattern p for all output layer neurons can be written as

$$E_p = \frac{1}{2}\sum_j (w_{ij} - x_j^p)^2,$$

where x_{jp} is the ith component of pattern p for neuron j. The summation is done over all j neurons.

Any change Δw_{ij} in the weight is expected to cause a reduction in error E_p.

Now E_p is a function of all the weights, so its rate of change with respect to any one weight value w_{ij} has to be measured by calculating its partial derivative with respect to w_{ij}.

$$\Delta_p w_{ij} = -\eta \frac{\delta E_p}{\partial w_{ij}}.$$

where η is a constant of proportionality.

Now we have to calculate the partial derivative of E_p:

$$\frac{\delta E_p}{\partial w_{ij}} = w_{ij} - x_j^p.$$

Combining two last expressions, we get

$$\Delta_p w_{ij} = -\eta \frac{\delta E_p}{\partial w_{ij}} = -\eta\,(w_{ij} - \quad x_j^p) = \eta\ (x_j^p - w_{ij}).$$

4.5 Training the Kohonen Network

The Kohonen Algorithm.

1. Initialize weights. Initialize weights from N inputs to the nodes to small random values. Set the initial radius of the neighbourhood.

2. Present new input $x_0(t)$, $x_1(t)$, $x_2(t)$, ... $x_{n-1}(t)$, where $x_i(t)$ is the input to node i at time t.

3. Compute distances to all nodes

Compute distances d_j between the input and each output node j using

$$d_j = \sum_{i}^{N-1} \left(x_i(t) - w_{ij}(t) \right)^2 ,$$

where $x_i(t)$ is the input to node i at time t and $w_{ij}(t)$ is the weight from input node i to output node j at time t.

4. Select output node with minimum distance. Select output node j^* as the output node with minimum d_j.

5. Update weights to node j^* and neighbors. Weights updated for node j^* and all nodes in the neighbourhood defined by $N_{j^*}(t)$. New weights are

$$w_{ij}(t+1) = w_{ij}(t) + \eta(t)(x_i(t) - w_{ij}(t)), \text{ for } j \text{ in } N_{j^*},\ 0 \le i \le \text{N}-1.$$

The term $\eta(t)$ is a gain term $0 \le \eta \le 1$. Both η and $N_{j^*}(t)$ decrease with time.

6. Repeat by going to the step 2.

Training issues in Kohonen Neural Nets

Vector normalization. To make vector comparison independent of magnitudes and dependent on orientation only, the vectors are normalized by dividing them by their magnitudes. This also helps to reduce training time.

Weight initialization. A random distribution of initial weights may not be optimal, resulting in sparsely populated trainable nodes and poor classification performance.

Possible remedies:

a. Initialization of weights to the same value and lumping of input vectors to similar orientation. This increases likelihood of all nodes being closer to a pattern vector. Inputs slowly return to original orientation with training.

b. Addition of random noise to inputs to distribute vectors over a larger pattern space.

c. Using a large initial neighbourhood, changing slowly.

Reducing neighbourhood size. Should be decreasing linearly with time (iterations). Neighbourhood shape may vary to suit application – e. g., circular or hexagonal instead of rectangular.

The decisions about the size of N_{j*} and the value of η are important. The sideways 'spread' of the Mexican hat function must change over time, hence changing the size of the neighbourhood of the units. Both N_{j*} and the value of η must decrease with time, and there are several ways of doing this. The value of η and the size of N_{j*} could decrease linearly with time, however, it has been pointed out that there are two distinct phases – an initial ordering phase, in which the elements find their correct topological order, and a final convergence phase in which the accuracy of the weights improves. For example, the initial ordering phase might take 1000 iterations where η decreases linearly from 0.9 to 0.01 say, and N_{j*} decreases linearly from half the diameter of the network to one spacing. During the final convergence phase η may decrease from 0.01 to 0 while N_{j*} stays at one spacing. This final stage could take form between 10 to 100 times longer than the initial stage depending on the desired accuracy.

An example is shown below where a two-dimensional array of elements is arranged in a square to map a rectangular two-dimensional coordinate space onto this array (which is the simplest case to imagine). Figure 16 shows the network for the example where units are trained to recognize their relative positions in two-dimensional space. This figure illustrates the dynamics of the learning process. Instead of plotting the position of the processing elements according to their physical location, they are plotted according to their location in weight space. As training proceeds, the map evolves. In the initial map, weight vectors have random values near the centre of the map coordinates (i. e., if these values are plotted on a two-dimensional image, they would be shown as a set of randomly distributed points). We want to indicate that some elements are next to other elements. This is done by drawing a line between adjacent elements so that the image ends up as a set of lines, the elements being situated at the points where the lines intersect. These lines are not physical, in the sense that the elements are not joined together, but show units that are neighbors in physical space.

The system is presented with a set of randomly chosen coordinates. As the map begins to evolve, weights spread out from the centre. Eventually the final structure of the map begins to appear. Finally the relationship between the weight vectors mimics the relationship between the physical coordinates of the processing elements (i. e., as time elapses, the weights order themselves so that they correspond to the positions in the coordinate system). Another way of thinking about this is that the weights distribute themselves in an even manner across the coordinate space so that, in effect, they learn to 'fill the space'.

Although the above example uses input points that are uniformly distributed within the region, they can in fact be distributed according to any distribution function. Once the SOM has been trained, the weight vectors will be organized into an approximation of the distribution function of the input vectors. Kohonen has shown that (more formally) "the point density function of the weight vectors tends to approximate the probability density

function $p(x)$ of the input vectors x, and the weight vectors tend to be ordered according to their mutual similarity".

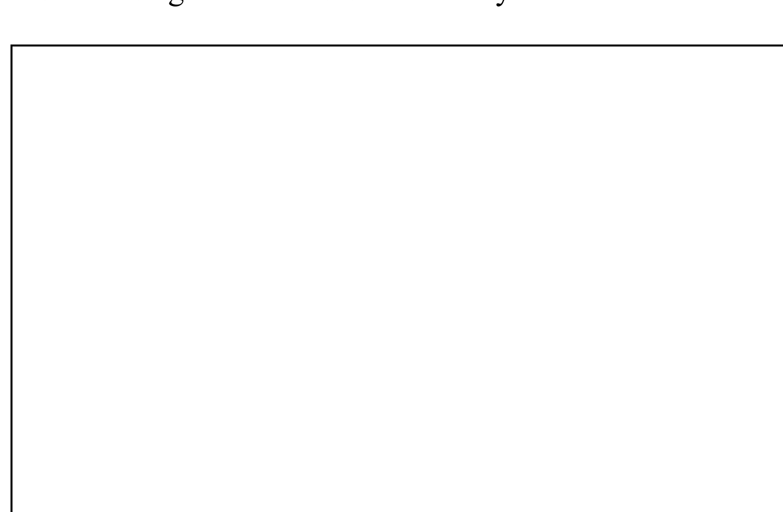


Figure 16 – Weight vectors during the ordering phase.

The network output doesn't need to be two-dimensional, even though the layout of the physical devices might be two-dimensional. If there are n weights, then each weight corresponds to a coordinate. So although a two-dimensional image, the elements of which are active, is produced when patterns are presented to the input, the interpretation of that map might have more dimensions. For example, a system where each element has three weights would organize itself so that the different pattern classes occupy different parts of a three-dimensional space. If the network is observed, only individual elements firing would be seen, so it is misleading to think in terms of the physical layout.

This (and other examples) shows how two-dimensional arrays which map on to a coordinate system can arrange the weights so that the 'nodes' in that system are distributed evenly. One thing that has not been mentioned yet is the output. What is the output of a Kohonen network? Training involves grouping similar patterns in close proximity in this pattern space, so that clusters of similar

patterns cause neurons to fire that are physically located close together in the network. Clearly, the outputs need to be interpreted, but it should be possible to identify which regions belong to which class by showing the network known patterns and seeing which areas are active.

Lecture 5

Radial Basis Function Networks

Statistics can be used in feedforward networks, and one of the most important of uses is in the radial basis function (RBF) network. This is becoming an increasingly popular neural network with diverse applications and is probably the main rival to the multi-layered perceptron. Much of the inspiration for RBF networks has come from traditional statistical pattern classification techniques, which are essentially getting a new lease of life as a form of neural network. However, by including RBFs in the general category of neural networks these techniques, which would only have been known to the few, have become widely used.

5.1 Architecture

The basic architecture for an RBF is a 3-layer network, as shown in Figure 17. The input layer is simply a fan-out layer and does no processing. The second or hidden layer performs a non-linear mapping from the input space into a (usually) higher dimensional space in which the patterns become linearly separable. The final layer therefore performs a simple weighted sum with a linear output. If the RBF network is used for function approximation (matching a real number) then this output is fine. However, if pattern classification is required, then a hard-limiter or sigmoid function could be placed on the output neurons to give 0 or 1 output values.

The unique feature of the RBF network is the process performed in the hidden layer. The idea is that the patterns in the input space form clusters. If the centres of these clusters are known, then the distance from the cluster centre can be measured. Furthermore, this distance measure is made non-linear, so that if a pattern is in an area that is close to a cluster centre it gives a value

close to 1. Beyond this area, the value drops dramatically. The notion is that this area is radially symmetrical around the cluster centre, so that the non-linear function becomes known as the radial-basis function.

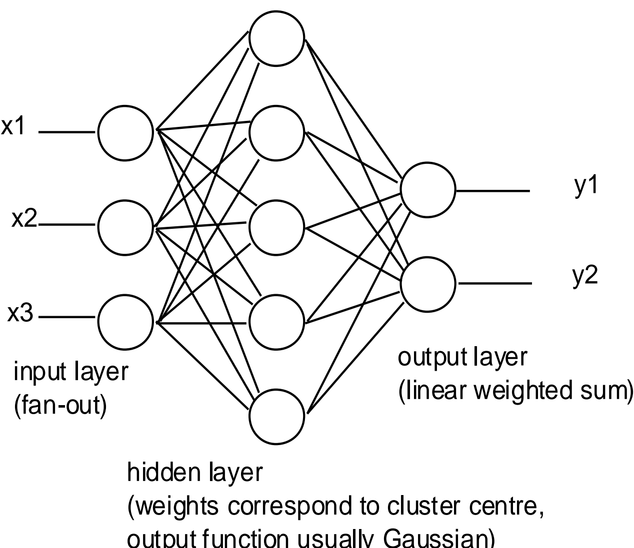


Figure 17 – Radial Basis Function Network

The most commonly used radial-basis function is:

$$\phi(r) = \exp\left(-\frac{r^2}{2\sigma^2}\right).$$

In an RBF network, r is the distance from the cluster centre. The equation represents a Gaussian bell-shaped curve, as shown in Figure 18. The distance measured from the cluster centre is usually the Euclidean distance. For each neuron in the hidden layer, the weights represent the coordinates of the centre of the cluster. Therefore, when that neuron receives an input pattern, X, the distance is found using the following equation:

$$r_j = \sqrt{\sum_{i=1}^{n}(x_i - w_{ij})^2} \; .$$

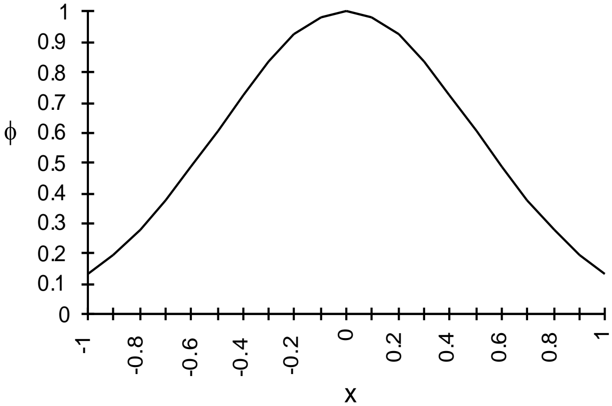


Figure 18 – A Gaussian centred at 0 with $\sigma = 0.5$

The variable σ defines the width or radius of the bell-shape and is something that has to be determined empirically. When the distance from the centre of the Gaussian reaches σ, the output drops from 1 to 0.6.

An often quoted example which shows how the RBF network can handle a non-linearly separable function is the exclusive-OR problem. One solution has 2 inputs, 2 hidden units and 1 output. The centres for the two hidden units are set at $c1 = 0.0$ and $c2 = 1.1$, and the value of radius σ is chosen such that $2\sigma^2 = 1$. When all four examples of input patterns are shown to the network, the outputs of the two hidden units are shown in the following table. The inputs are x, the distances from the centres squared are r, and the outputs from the hidden units are φ.

Table 5 – The outputs of the hidden units

x1	x2	r1	r2	φ1	φ2
0	0	0	2	1	0.1
0	1	1	1	0.4	0.4
1	0	1	1	0.4	0.4
1	1	2	0	0.1	1

$$(hidden_unit)\phi_j = \exp\left(-\frac{\sum_{i=1}^{n}(x_i - w_{ij})^2}{2\sigma^2} \right)$$

Figure 19 shows the position of the four input patterns using the output of the two hidden units as the axes on the graph. It can be seen that the patterns are now linearly separable.

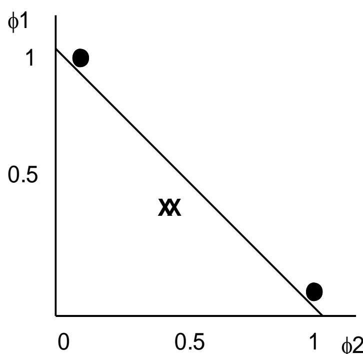


Figure 19 – The input patterns after being transformed by the hidden layer

This demonstrates the power of transforming from one domain to another using an RBF network. However, the centres were chosen carefully to show this result. The methods generally adopted

for learning in an RBF network would find it impossible to arrive at those centre values. The learning methods that are usually adopted will be described in the next section.

5.2 Training an RBF network

Hidden layer

The hidden layer in an RBF network has units which have weights that correspond to the vector representation of the centre of a cluster. These weights are found either using a traditional clustering algorithm such as the *k*-means algorithm, or adaptively using essentially the Kohonen algorithm. In either case, the training is unsupervised but the number of clusters that you expect, *k*, is set in advance. The algorithms then find the best fit to these clusters. The *k*-means algorithm will be briefly outlined. Initially *k* points in the pattern space are randomly set. Then for each item of data in the training set, the distances are found from all of the *k* centres. The closest centre is chosen for each item of data. This is the initial classification, so all items of data will be assigned a class from 1 to *k*. Then, for all data which has been found to be class 1, the average or mean values are found for each of co-ordinates. These become the new values for the centre corresponding to class 1. This is repeated for all data found to be in class 2, then class 3 and so on until class *k* is dealt with. We now have *k* new centres. The process of measuring the distance between the centres and each item of data and re-classifying the data is repeated until there is no further change. The sum of the distances can be monitored and the process halts when the total distance no longer falls.

The alternative is to use an adaptive *k*-means algorithm similar to Kohonen learning. Input patterns are presented to all of the cluster centres one at a time, and the cluster centres adjusted after each one. The cluster centre that is nearest to the input data wins, and is shifted slightly towards the new data. This has the advantage that you don't have to store all of the training data so can be done on-line.

Having found the cluster centres using one or other of these methods, the next step is determining the radius of the Gaussian curves. This is usually done using the *P*-nearest neighbor algorithm. A number *P* is chosen, and for each centre, the *P* nearest centres are found. The root-mean squared distance between the current cluster centre and its *P* nearest neighbors is calculated, and this is the value chosen for σ. So, if the current cluster centre is c_j, the value is:

A typical value for *P* is 2, in which case σ is set to be the average distance from the two nearest neighboring cluster centres:

$$\sigma_j = \sqrt{\frac{1}{P}\sum_{i=1}^{P}(c_k - c_i)^2}.$$

Using this method for training the hidden layer, exclusive-OR function can be implemented using a minimum of 4 hidden units. If more than four units are used, the additional units duplicate the centres and therefore do not contribute any further discrimination to the network. So, assuming four neurons in the hidden layer, each unit is centred on one of the four input patterns, namely 00, 01, 10 and 11. The *P*-nearest neighbor algorithm with *P* set to 2 is used to find the size of the radii. In each of the neurons, the distances to the other three neurons is 1, 1 and 1.414, so the two nearest cluster centres are at a distance of 1. Using the mean squared distance as the radii gives each neuron a radius of 1. Using these values for the centres and radius, if each of the four input patterns is presented to the network, the output of the hidden layer would be:

Table 5 – The output of the hidden layer

input	neuron 1	neuron 2	neuron 3	neuron 4
00	0.6	0.4	1.0	0.6
01	0.4	0.6	0.6	1.0
10	1.0	0.6	0.6	0.4
11	0.6	1.0	0.4	0.6

Output layer

Having trained the hidden layer with some unsupervised learning, the final step is to train the output layer using a standard gradient descent technique such as the Least Mean Squares algorithm. In the example of the exclusive-OR function given above a suitable set of weights would be +1, –1, –1 and +1. With these weights the value of net and the output is:

Table 6 – The output of the output layer

input	neuron 1	neuron 2	neuron 3	neuron 4	net	output
00	0.6	0.4	1.0	0.6	–0.2	0
01	0.4	0.6	0.6	1.0	0.2	1
10	1.0	0.6	0.6	0.4	0.2	1
11	0.6	1.0	0.4	0.6	–0.2	0

5.3 Advantages of an RBF

Many advantages are claimed for RBF networks over multi-layer perceptrons (MLPs). It is said that an RBF trains faster than an MLP and that it produces better decision boundaries. Another advantage that is claimed is that the hidden layer is easier to interpret than the hidden layer in an MLP. Some of the disadvantages that are claimed for an RBF are that an MLP gives better distributed representation. Although the RBF is quick to train, when training is finished and it is being used it is slower than an MLP, so where speed is a factor an MLP may be more appropriate.

Statistical feed-forward networks such as the radial basis function network have become very popular, and are serious rivals to the multi-layered perceptron. Their success is likely to be due to the fact they are essentially well tried statistical techniques being presented as neural networks. The learning mechanisms in statistical neural networks are not biologically plausible, with the result that these networks have not been taken up by those researchers who insist on biological analogies.

REFERENCES

1. Wells Ian. Intelligent Information Systems / Ian Wells, Tony Browne [Електронний ресурс]. – Режим доступу : http://www.computing.surrey.ac.uk/courses/csm10/.
2. Simon Colton. Artificial Intelligence / Colton Simon [Електронний ресурс]. – Режим доступу : http://www.doc.ic.ac.uk/~sgc/teaching/v231/.
3. Beale R. Neural Computing: An Introduction / R. Beale, T. Jackson. – Bristol, Philadelphia and New York : IOP Publishing Ltd, 1990. – 227 p.
4. Raul Rojas. Neural Networks / Rojas Raul. – Springer–Verlag, Berlin, 1996. – 509 p.
5. Haykin Simon. Neural Networks and Learning Machines. Third Edition / Simon Haykin. – the USA, New Jersey : Pearson Education Inc., 1999. – 906 p.

Навчальне видання

Князь Ігор Олександрович

Конспект лекцій
із курсу «Моделювання нейронних мереж»
У двох частинах
Частина 1
(Англійською мовою)

Відповідальний за випуск О. В. Лисенко
Редактор Л. В. Штихно
Комп’ютерне верстання І. О. Князя

Формат 60х84/16. Ум. друк. арк. 3,49. Обл.-вид. арк. 4,25.

Видавець і виготовлювач
Сумський державний університет,
вул. Римського-Корсакова, 2, м. Суми, 40007
Свідоцтво суб’єкта видавничої справи ДК № 3062 від 17.12.2007.