January 17th 2005

Introduction to Neural Networks and Data Mining – lecture 1

BARGIELA Andrzej *(ISM – NTU, Hirota Lab. – TITech)*

1. Motivation

In this cycle of three lectures we will study the properties of idealized "neurons". There are three complementary motivations that support this broad interdisciplinary field:

- **Biology**. One of the unresolved scientific problems is to understand how the brain works. Building some neural network models helps to advance this understanding by allowing the medical data to be tested against expected simulation results. This is a common approach sometimes referred to as *simulation modeling*.

- **Engineering.** Many engineering applications generate data that if taken in its *raw* format can be overwhelming. Thus there is a need for intelligent data reduction (summarization) through the identification of *patterns in data* as well as the identification of *relevant characteristics of the data*. An attempt to formalize the process of information abstractions is being made by the emerging discipline of *Granular Computing*.

- **Complex Systems.** The complex nature of computations accomplished by neural networks is interesting in its own right as a subject of scientific investigation as it gives us an alternative to the classical *fetch/decode-execute/store* Von Neumann model of computations.

Having a short cycle of three lectures I need to emphasise strongly that:

- We will be able to discuss only a limited number of neural network types. There are many interesting neural networks that we will have no time to touch on.
- The models that we discuss are not intended to be faithful models of biological systems. They are all relevant to biology but their relevance is on an abstract level.
- I will describe some neural networks that are used in data mining but I will have no time to give a full description of the state of the art. If you wish to solve real problems with neural networks, please read the relevant papers from the archival-quality journals such as IEEE Transactions on Neural Networks.

2. Memories

Before we start discussing neural networks with their learning algorithms that allow them to function as *memories* it is useful to reflect on the idea of memory in digital computation and in real life.

A memory stored in a computer can be for example a string of 1000 bytes describing a name of the person, date of birth, address and an image of the person's face. This string of bytes is stored at a specific address; so, to retrieve the *memory* about the specific

person we need to provide an exact address where this memory is stored. The address has nothing to do with the *memory* itself. Any *memory* can be stored at the given address and any address can be used for a specific *memory*.

Notice that the address-based method of storing *memories* has the following disadvantages:

- It does not allow retrieval of complete *memories* on the basis of partial memories (even if we know the name of the person we cannot retrieve directly the image of the person's face without the exact knowledge of the memory's address). There are methods of generating such addresses from partial knowledge of memory content but it is clear that these methods just attempt to fix the inherent problem.
- It is not robust and fault-tolerant. If we make an error in the specification of the memory address we are going to retrieve details of a completely different person. There is no way of knowing that an error has occurred except by developing an additional functionality of error checking which is using some information redundancy to fix the problem.
- It is inherently local in nature even if it may be part of a distributed computing resource. Retrieval of memory content in a distributed computation is not different from the retrieval of memory content in a single-processor computation except that it may be replicated several times over.

By contrast *biological memory systems* work in a completely different way.

- Errors in the cues for memory recall can be corrected. For example if you were asked to recall "*An American politician who was highly intelligent and whose father did not like broccoli"*. Many will think of President Bush – even though one of the cues contains an error.
- Errors in hardware can be tolerated. While the cells that make up the brain are in a constant state of renewal the memories that have been encoded by these cells persist despite this dynamic renewal process.
- Biological memories are distributed to some extent as evidenced by the medical studies of people who suffered some brain injuries. Although there is some functional specialization of the various areas of the brain it seems that multiple memories are stored by widely distributed neurons giving rise to recovery of significant proportion of memories in case of a loss of "specialist" cells for these memories.

These properties of biological memory systems motivate the study of "artificial neural networks", which are distributed computing systems consisting of many interacting simple elements. The hope is that by building models of neural networks and comparing their performance to that of real biological systems one may gain better insight into how nature copes with the fundamental problem of information storage and retrieval.

3. Biological brain (*Second most important organ,* Woody Allen)

3.1. Computations in the brain

The human brain contains about 10 billion nerve cells, or **neurons**. On average, each neuron is connected to other neurons through about 10 000 **synapses**. (The actual figures vary greatly, depending on the local neuroanatomy.) 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. (Nobody knows yet how this occurs.)

	Processing elements	Element size	Energy use	Speed	Style of comput.	Fault tollerance	Learning	Intell-igence
brain	10^{14} synapses	10^{-6} m	30 W	100Hz	Parallel Distrib.	yes	yes	usually
computer	10^{8} transistors	10^{-6} m	30 W (CPU)	10^{9}Hz	Serial	no	little	Not yet

As a discipline of Artificial Intelligence, Neural Networks attempt to bring computers a little closer to the brain's capabilities by imitating certain aspects of information processing in the brain, in a highly simplified way.

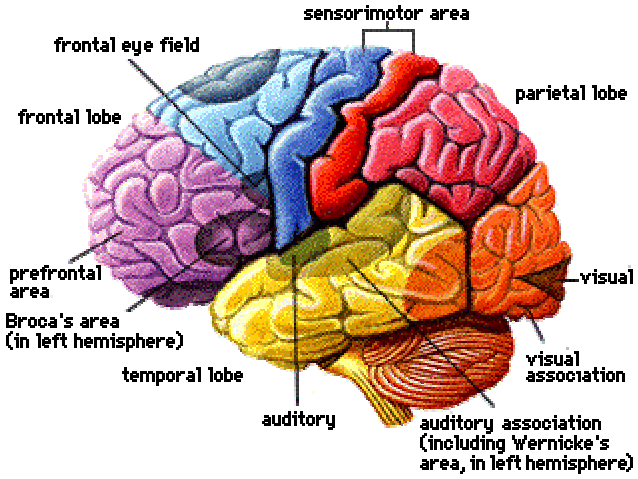
3.2. Anatomy of the brain

The brain is not homogeneous. At the largest anatomical scale, we distinguish **cortex**, **midbrain**, **brainstem**, and **cerebellum**. Each of these can be hierarchically subdivided into many **regions**, and **areas** within each region, either according to the anatomical structure of the neural networks within it, or according to the function performed by them.

The overall pattern of **projections** (bundles of neural connections) between areas is extremely complex, and only partially known. The best mapped (and largest) system in

the human brain is the visual system, where the first 10 or 11 processing stages have been identified. We distinguish **feedforward** projections that go from earlier processing stages (near the sensory input) to later ones (near the motor output), from **feedback** connections that go in the opposite direction.

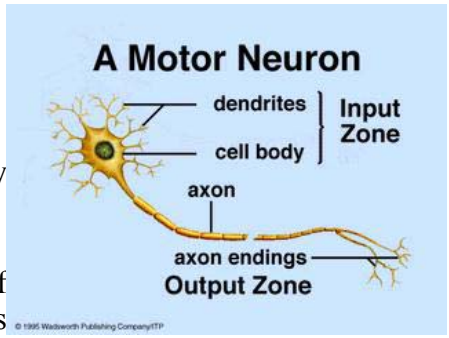
In addition to these long-range connections, neurons also link up with many thousands of their neighbours. In this way they form very dense, complex local networks:



3.3. Neurons and synapses

The basic computational unit in the nervous system is the nerve cell, or **neuron**. A neuron has:

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



A neuron receives input from other neurons (typically many thousands).

Inputs sum up (approximately) and once the sum of inputs 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, etc.

3.4. Synaptic learning

Brains learn. Of course. From what we know of neuronal structures, the way brains learn is by altering the strengths of connections between neurons, and by adding or deleting connections between neurons. Furthermore, they learn "on-line", based on experience, and typically without the benefit of a benevolent teacher.

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

Long-term Potentiation:

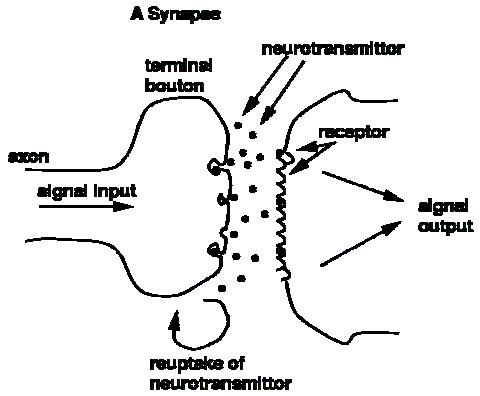
> An enduring (>1 hour) increase in synaptic efficacy that results from high-frequency stimulation of an afferent (input) pathway

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."

Points to note about LTP:

- Synapses become more or less important over time (plasticity)
- LTP is based on experience
- LTP is based only on *local* information (Hebb's postulate)



4. Artificial neuron models

Computational neurobiologists have constructed very elaborate computer models of neurons in order to run detailed simulations of particular circuits in the brain. As Computer Scientists, we are more interested in the general properties of neural networks, independent of how they are actually "implemented" in the brain. This means that we can use much simpler, abstract "neurons", which (hopefully) capture the essence of neural computation even if they leave out much of the details of how biological neurons work.

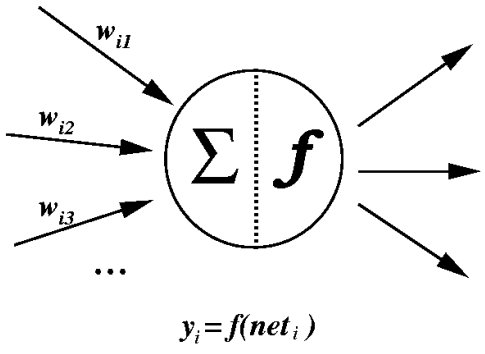
People have implemented model neurons in hardware as electronic circuits, often integrated on VLSI chips. Remember though that computers run much faster than brains - we can therefore run fairly large networks of simple model neurons as software simulations in reasonable time. This has obvious advantages over having to use special "neural" computer hardware.

4.1 A simple model of a neuron

Our basic computational element (model neuron) is often called a **node** or **unit**. It receives input from some other units, or perhaps from an external source. Each input has an associated **weight** *w*, which can be modified so as to model synaptic learning. The unit computes some function *f* of the weighted sum of its inputs:

$$y_i = f\left(\sum_j w_{ij} y_j\right)$$

Its output, in turn, can serve as input to other units. (weights **w** reflect synaptic learning)



- The weighted sum $\sum_j w_{ij} y_j$ is called the **net input** to unit *i*, often written net_i.
- Note that w_{ij} refers to the weight from unit *j* to unit *i* (not the other way around).
- The function *f* is the unit's **activation function**. In the simplest case, *f* is the identity function, and the unit's output is just its net input. This is called a **linear unit**.
- We will address other functions *f* next week

5. Linear regression

Let's consider the problem of fitting model to the following data:

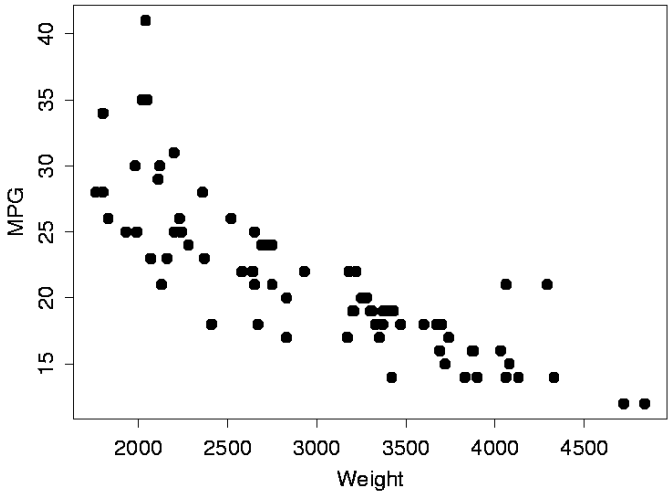


Figure 5.1. Fuel consumption data

Each dot in the figure provides information about the weight (x-axis, units: U.S. pounds) and fuel consumption (y-axis, units: miles per gallon) for one of 74 cars (data from 1979). Clearly weight and fuel consumption are linked, so that, in general, heavier cars use more fuel.

Now suppose we are given the weight of a 75th car, and asked to predict how much fuel it will use, based on the above data. Such questions can be answered by using a **model** - a short mathematical description - of the data. The simplest useful model here is of the form

$$y = w_1 x + w_0 \qquad (5.1)$$

This is a **linear** model: in an xy-plot, equation 1 describes a straight line with **slope** w_1 and **intersection** w_0 with the y-axis, as shown in Fig. 5.2. (Note that we have rescaled the coordinate axes - this does not change the problem in any fundamental way.)

How do we choose the two parameters w_0 and w_1 of our model? Clearly, any straight line drawn somehow through the data could be used as a predictor, but some lines will do a better job than others. The line in Fig. 5.2 is certainly not a good model: for most cars, it will predict too much fuel consumption for a given weight.

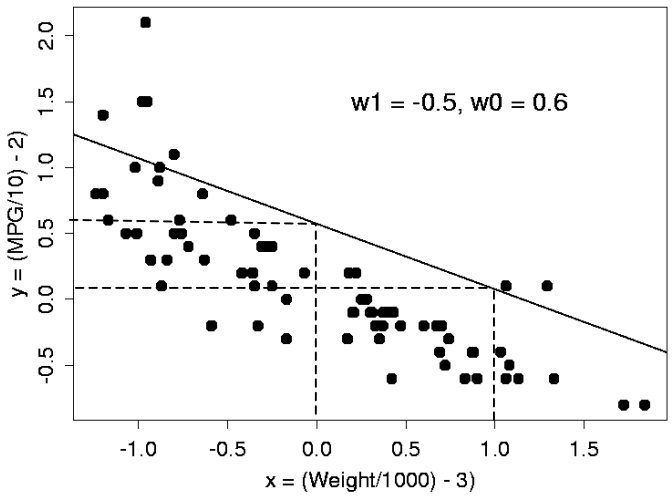


Figure 5.2. Linear model of fuel consumption

5.1. The loss function

In order to make precise what we mean by being a "good predictor", we define a **loss** (also called **objective** or **error**) function E over the model parameters. A popular choice for E is the **sum-squared error**:

$$E = \sum_{p=1}^{n} E^p \qquad E^p = \frac{1}{2}(t^p - y^p)^2 \tag{5.2}$$

In words, it is the sum over all points $p=1,...n$ in our data set of the squared difference between the **target** value t^p (here: actual fuel consumption) and the model's prediction y^p, calculated from the input value x^p (here: weight of the car) by equation 1. For a linear model, the sum-squared error is a quadratic function of the model parameters. Figure 5.3 shows E for a range of values of w_0 and w_1. Figure 5.4 shows the same functions as a contour plot.

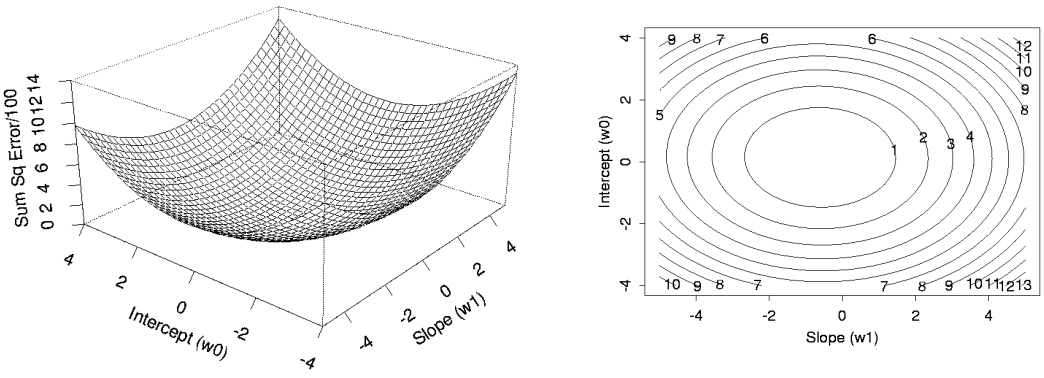
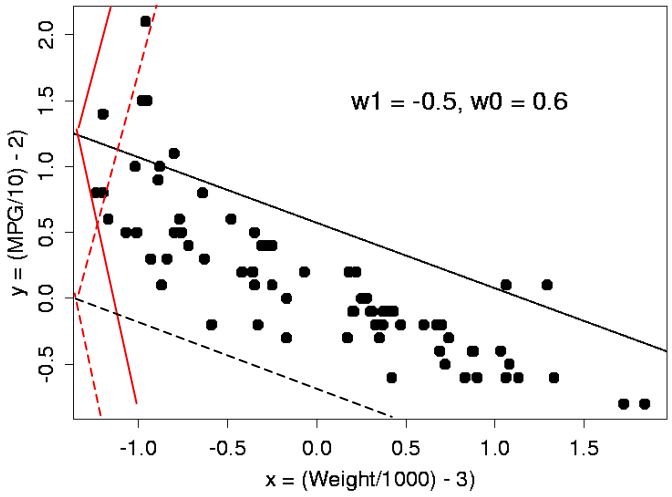


Figure 5.3. 3-D plot of the cost function

Figure 5.4. Contour plot of the cost function

(in the above we have evaluated eq. (5.2) for w0 and w1 from interval [-4, 4] with 0.1 increments on the values of the parameters)



5.2. Minimising the loss

The loss function *E* provides us with an objective measure of predictive error for a specific choice of model parameters. We can thus restate our goal of finding the best (linear) model as finding the values for the model parameters that minimize *E*.

For linear models, **linear regression** provides a direct way to compute these optimal model parameters. (See any statistics textbook for details.) However, this analytical approach does not generalize to **nonlinear** models. Even though the solution cannot be calculated explicitly in that case, the problem can still be solved by an iterative numerical technique called **gradient descent**. It works as follows:

1. Choose some (random) initial values for the model parameters.
2. Calculate the gradient G of the error function with respect to each model parameter.
3. Change the model parameters so that we move a short distance in the direction of the greatest rate of decrease of the error, i.e., in the direction of -G.
4. Repeat steps 2 and 3 until G gets close to zero.

How does this work? The gradient of E gives us the direction in which the loss function at the current setting of the *w* has the steepest **slope**. In order to decrease *E*, we take a small step in the opposite direction, -*G* (Fig. 5.5).

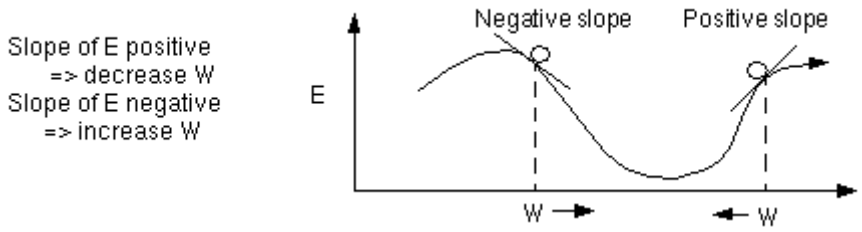


Figure 5.5. Illustration of the gradient descent

By repeating this over and over, we move "downhill" in *E* until we reach a minimum, where $G = 0$, so that no further progress is possible (Fig. 5.6).

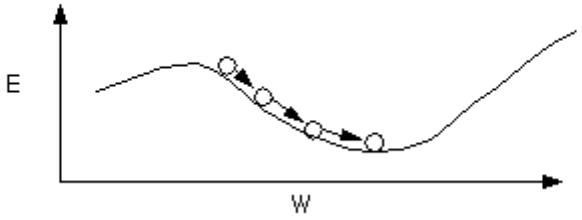


Figure 5.6. Tuning of the weights

Fig. 5.7 shows the best linear model for our car data, found by this procedure.

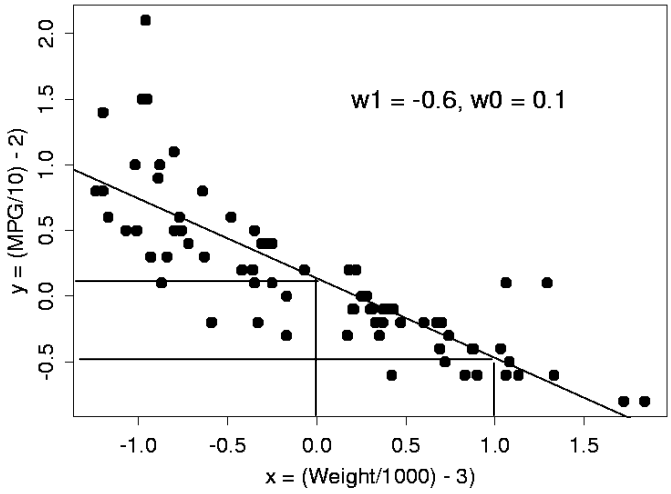


Figure 5.7. The best linear model of the data

5.3 It's a Neural Network !

Our linear model of equation 1 can in fact be implemented by the simple neural network shown in Fig. 8. It consists of a **bias** unit, an **input** unit, and a linear **output** unit. The input unit makes external input x (here: the weight of a car) available to the network, while the bias unit always has a constant output of 1. The output unit computes the sum:

$$y_2 = y_1 w_{21} + 1.0 w_{20} \tag{5.3}$$

It is easy to see that this is equivalent to equation 1, with w_{21} implementing the slope of the straight line, and w_{20} its intercept with the y-axis.

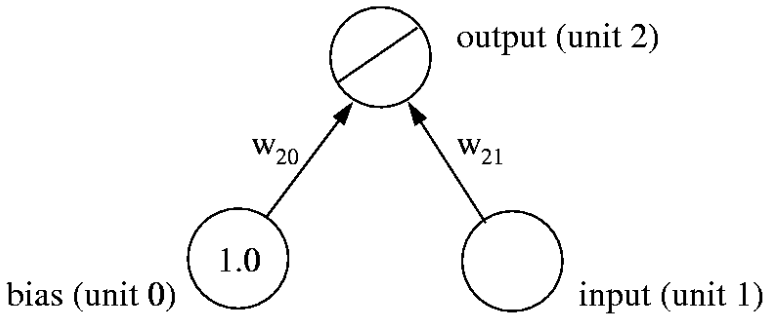


Figure 8. NN model of the regression line

6. Linear Neural Networks

6.1. Multiple regression

Our car example showed how we could discover an optimal linear function for predicting one variable (fuel consumption) from another variable (weight). Suppose now that we are also given one or more additional variables, which could be useful as predictors. Our simple neural network model can easily be extended to this case by adding more input units (Fig. 6.1).

Similarly, we may want to predict more than one variable from the data that we are given. This can easily be accommodated by adding more output units (Fig. 6.2). The loss function for a network with multiple outputs is obtained simply by adding the loss for each output unit together. The network now has a typical layered structure: a layer of input units (and the bias), connected by a layer of weights to a layer of output units.

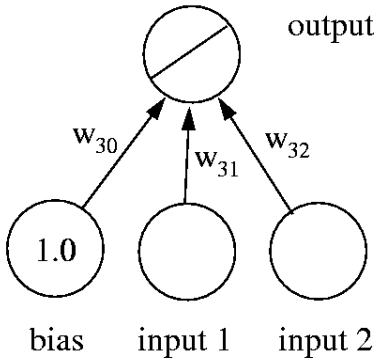


Figure 6.1. Multiple inputs NN

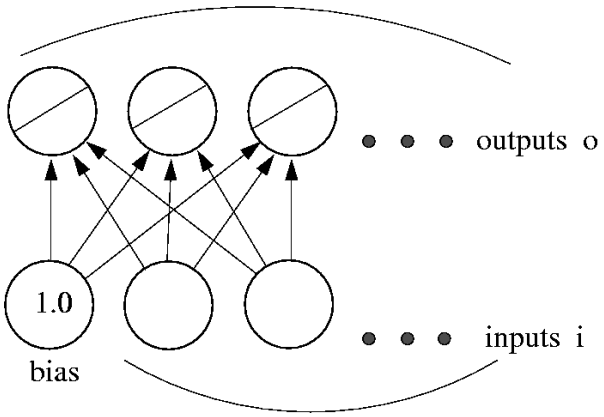


Figure 6.2. Multiple inputs and outputs NN

6.2. Computing the gradient

In order to train neural networks such as the ones shown above by gradient descent, we need to be able to compute the gradient G of the loss function with respect to each weight w_{ij} of the network. It tells us how a small change in that weight will affect the overall error E. We begin by splitting the loss function into separate terms for each point p in the training data:

$$E = \sum_{p=1}^{n} E^p \qquad\qquad E^p = \frac{1}{2} \sum_{o=1}^{m} (t_o^p - y_o^p)^2 \tag{6.1}$$

where o ranges over the m output units of the network. (As before we use the superscript p to denote the training point). Since differentiation and summation are interchangeable, we can likewise split the gradient into separate components for each training point:

$$G = \frac{\partial E}{\partial w_{oi}} = \frac{\partial}{\partial w_{oi}} \sum_{p=1}^{n} E^p = \sum_{p=1}^{n} \frac{\partial E^p}{\partial w_{oi}} \tag{6.2}$$

In what follows, we describe the computation of the gradient for a single data point, omitting the superscript p in order to make the notation easier to follow.

First use the **chain rule** to decompose the gradient into two factors:

$$\frac{\partial E^p}{\partial w_{oi}} = \frac{\partial E^p}{\partial y_o^p} \frac{\partial y_o^p}{\partial w_{oi}} \tag{6.3}$$

The first factor can be obtained by differentiating Eqn. 1 above:

$$\frac{\partial E^p}{\partial y_o^p} = -(t_o^p - y_o^p) \tag{6.4}$$

Using $y_o^p = \sum_{i=0}^{n} w_{oi} x_i^p$, the second factor becomes

$$\frac{\partial y_o^p}{\partial w_{oi}} = \frac{\partial}{\partial w_{oi}} \sum_{i=0}^{n} w_{oi} x_i^p \tag{6.5}$$

Combining equations (3-5), we obtain

$$\frac{\partial E^p}{\partial w_{oi}} = -(t_o^p - y_o^p) x_i^p \tag{6.6}$$

To find the gradient *G* for the entire data set, we sum at each weight the contribution given by equation 6 over all the data points. We can then subtract a small proportion μ (called the **learning rate**) of *G* from the weights to perform gradient descent.

6.3. The Gradient Descent Algorithm

1. Initialize all weights to small random values.
2. REPEAT until done
    1. For each weight w_{oi} set $\triangle w_{oi} := 0$
    2. For each data point $(\mathbf{x}, \mathbf{t})^p$, p=1,…,n
        1. set input units to $\mathbf{x}^p$
        2. compute value of output units $\mathbf{y}^p$
        3. For each weight w_{oi}
           set $\triangle w_{oi} := \triangle w_{oi} + (t_o^p - y_o^p) x_i^p$
    3. For each weight w_{oj} set $w_{oi} := w_{oi} + \mu \triangle w_{oi}$

The algorithm terminates once we are at, or sufficiently near to, the minimum of the error function, where G = 0. We say then that the algorithm has **converged**.

In summary:

	general case	linear network
Training data	**(x,t)**	**(x,t)**
Model parameters	**w**	**w**
Model	**y** = g(**w**,**x**)	$y_o^p = \sum_{i=0}^{n} w_{oi} x_i^p$
Error function	E(**y**,**t**)	$E = \sum_{p=1}^{n} E^p \qquad E^p = \frac{1}{2} \sum_{o=1}^{m} (t_o^p - y_o^p)^2$
Gradient with respect to w_{oi}	$\frac{\partial E^p}{\partial w_{oi}}$	$\frac{\partial E^p}{\partial w_{oi}} = -(t_o^p - y_o^p) x_i^p$
Weight update rule	$\Delta w_{oi} = -\mu \frac{\partial E^p}{\partial w_{oi}}$	$\Delta w_{oi} = -\mu (t_o^p - y_o^p) x_i^p$

6.4 The learning rate

An important consideration is the learning rate μ, which determines by how much we change the weights w at each step. If μ is too small, the algorithm will take a long time to converge (Fig. 6.3).

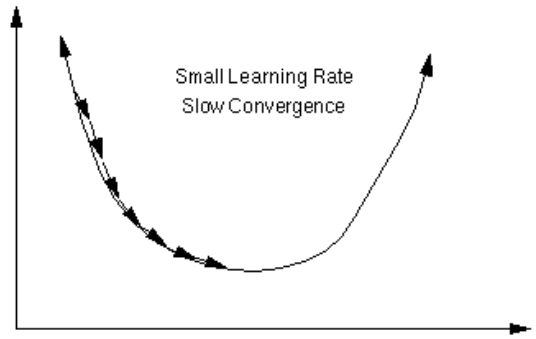


Figure 6.3. Small μ

Conversely, if μ is too large, we may end up bouncing around the error surface out of control - the algorithm **diverges** (Fig. 6.4). This usually ends with an overflow error in the computer's floating-point arithmetic.

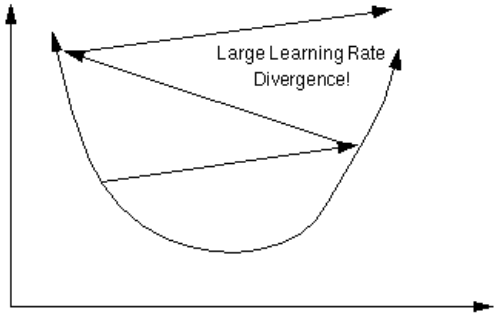


Figure 6.4. Large μ

6.5 Batch vs. online learning

Above we have accumulated the gradient contributions for all data points in the training set before updating the weights. This method is often referred to as **batch** learning. An alternative approach is **online** learning, where the weights are updated immediately after seeing each data point. Since the gradient for a single data point can be considered a noisy approximation to the overall gradient G (Fig. 6.5), this is also called **stochastic** (noisy) gradient descent.

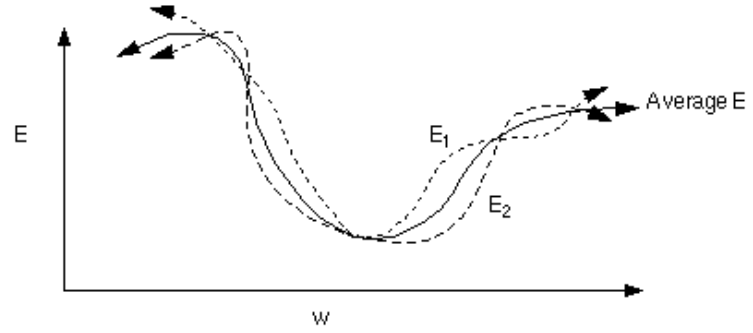


Figure 6.5. Noisy approximations of the gradient G

Online learning has a number of advantages:

- it is often much faster, especially when the training set is **redundant** (contains many similar data points),
- it can be used when there is no fixed training set (new data keeps coming in),
- it is better at tracking **nonstationary** environments (where the best model gradually changes over time),
- the noise in the gradient can help to escape from **local minima** (which are a problem for gradient descent in nonlinear models).

These advantages are, however, bought at a price: many powerful optimization techniques (such as: conjugate and second-order gradient methods, support vector machines, Bayesian methods, etc.) - which we will not talk about in this course! - are batch methods that cannot be used online. (Of course this also means that in order to implement batch learning really well, one has to learn an awful lot about these rather complicated methods!)

A compromise between batch and online learning is the use of "mini-batches": the weights are updated after every *n* data points, where *n* is greater than 1 but smaller than the training set size.

In order to keep things simple, we will focus very much on online learning, where plain gradient descent is among the best available techniques. Online learning is also highly suitable for implementing things such as reactive control strategies in adapative agents, and should thus fit in well with the rest of your course.