

11

Multilayer Perceptrons

The multilayer perceptron is an artificial neural network structure and is a nonparametric estimator that can be used for classification and regression. We discuss the backpropagation algorithm to train a multilayer perceptron for a variety of applications.

11.1 Introduction

ARTIFICIAL NEURAL network models, one of which is the *perceptron* we discuss in this chapter, take their inspiration from the brain. There are cognitive scientists and neuroscientists whose aim is to understand the functioning of the brain (Posner 1989; Thagard 2005), and toward this aim, build models of the natural neural networks in the brain and make simulation studies.

ARTIFICIAL NEURAL
NETWORKS

However, in engineering, our aim is not to understand the brain *per se*, but to build useful machines. We are interested in *artificial neural networks* because we believe that they may help us build better computer systems. The brain is an information processing device that has some incredible abilities and surpasses current engineering products in many domains—for example, vision, speech recognition, and learning, to name three. These applications have evident economic utility if implemented on machines. If we can understand how the brain performs these functions, we can define solutions to these tasks as formal algorithms and implement them on computers.

NEURONS

The human brain is quite different from a computer. Whereas a computer generally has one processor, the brain is composed of a very large (10^{11}) number of processing units, namely, *neurons*, operating in parallel. Though the details are not known, the processing units are believed to be

SYNAPSES

much simpler and slower than a processor in a computer. What also makes the brain different, and is believed to provide its computational power, is the large connectivity. Neurons in the brain have connections, called *synapses*, to around 10^4 other neurons, all operating in parallel. In a computer, the processor is active and the memory is separate and passive, but it is believed that in the brain, both the processing and memory are distributed together over the network; processing is done by the neurons, and the memory is in the synapses between the neurons.

11.1.1 Understanding the Brain

LEVELS OF ANALYSIS

According to Marr (1982), understanding an information processing system has three levels, called the *levels of analysis*:

1. *Computational theory* corresponds to the goal of computation and an abstract definition of the task.
2. *Representation and algorithm* is about how the input and the output are represented and about the specification of the algorithm for the transformation from the input to the output.
3. *Hardware implementation* is the actual physical realization of the system.

One example is sorting: The computational theory is to order a given set of elements. The representation may use integers, and the algorithm may be Quicksort. After compilation, the executable code for a particular processor sorting integers represented in binary is one hardware implementation.

The idea is that for the same computational theory, there may be multiple representations and algorithms manipulating symbols in that representation. Similarly, for any given representation and algorithm, there may be multiple hardware implementations. We can use one of various sorting algorithms, and even the same algorithm can be compiled on computers with different processors and lead to different hardware implementations.

To take another example, '6', 'VI', and '110' are three different representations of the number six. There is a different algorithm for addition depending on the representation used. Digital computers use binary representation and have circuitry to add in this representation, which is one

particular hardware implementation. Numbers are represented differently, and addition corresponds to a different set of instructions on an abacus, which is another hardware implementation. When we add two numbers in our head, we use another representation and an algorithm suitable to that representation, which is implemented by the neurons. But all these different hardware implementations—for example, us, abacus, digital computer—implement the same computational theory, addition.

The classic example is the difference between natural and artificial flying machines. A sparrow flaps its wings; a commercial airplane does not flap its wings but uses jet engines. The sparrow and the airplane are two hardware implementations built for different purposes, satisfying different constraints. But they both implement the same theory, which is aerodynamics.

The brain is one hardware implementation for learning or pattern recognition. If from this particular implementation, we can do reverse engineering and extract the representation and the algorithm used, and if from that in turn, we can get the computational theory, we can then use another representation and algorithm, and in turn a hardware implementation more suited to the means and constraints we have. One hopes our implementation will be cheaper, faster, and more accurate.

Just as the initial attempts to build flying machines looked very much like birds until we discovered aerodynamics, it is also expected that the first attempts to build structures possessing brain's abilities will look like the brain with networks of large numbers of processing units, until we discover the computational theory of intelligence. So it can be said that in understanding the brain, when we are working on artificial neural networks, we are at the representation and algorithm level.

Just as the feathers are irrelevant to flying, in time we may discover that neurons and synapses are irrelevant to intelligence. But until that time there is one other reason why we are interested in understanding the functioning of the brain, and that is related to parallel processing.

11.1.2 Neural Networks as a Paradigm for Parallel Processing

Since the 1980s, computer systems with thousands of processors have been commercially available. The software for such parallel architectures, however, has not advanced as quickly as hardware. The reason for this is that almost all our theory of computation up to that point was based

on serial, one-processor machines. We are not able to use the parallel machines we have efficiently because we cannot program them efficiently.

There are mainly two paradigms for *parallel processing*: In single instruction, multiple data (SIMD) machines, all processors execute the same instruction but on different pieces of data. In multiple instruction, multiple data (MIMD) machines, different processors may execute different instructions on different data. SIMD machines are easier to program because there is only one program to write. However, problems rarely have such a regular structure that they can be parallelized over a SIMD machine. MIMD machines are more general, but it is not an easy task to write separate programs for all the individual processors; additional problems are related to synchronization, data transfer between processors, and so forth. SIMD machines are also easier to build, and machines with more processors can be constructed if they are SIMD. In MIMD machines, processors are more complex, and a more complex communication network should be constructed for the processors to exchange data arbitrarily.

Assume now that we can have machines where processors are a little bit more complex than SIMD processors but not as complex as MIMD processors. Assume we have simple processors with a small amount of local memory where some parameters can be stored. Each processor implements a fixed function and executes the same instructions as SIMD processors; but by loading different values into the local memory, they can be doing different things and the whole operation can be distributed over such processors. We will then have what we can call neural instruction, multiple data (NIMD) machines, where each processor corresponds to a neuron, local parameters correspond to its synaptic weights, and the whole structure is a neural network. If the function implemented in each processor is simple and if the local memory is small, then many such processors can be fit on a single chip.

The problem now is to distribute a task over a network of such processors and to determine the local parameter values. This is where learning comes into play: We do not need to program such machines and determine the parameter values ourselves if such machines can learn from examples.

Thus, artificial neural networks are a way to make use of the parallel hardware we can build with current technology and—thanks to learning—they need not be programmed. Therefore, we also save ourselves the effort of programming them.

In this chapter, we discuss such structures and how they are trained.

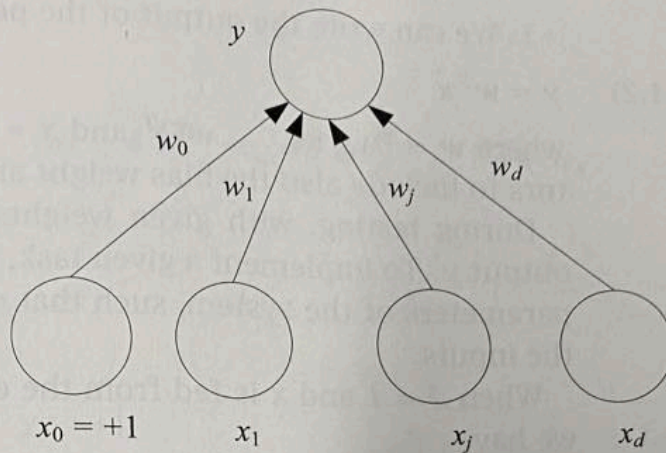


Figure 11.1 Simple perceptron. $x_j, j = 1, \dots, d$ are the input units. x_0 is the bias unit that always has the value 1. y is the output unit. w_j is the weight of the directed connection from input x_j to the output.

Keep in mind that the operation of an artificial neural network is a mathematical function that can be implemented on a serial computer—as it generally is—and training the network is not much different from statistical techniques that we have discussed in the previous chapters. Thinking of this operation as being carried out on a network of simple processing units is meaningful only if we have the parallel hardware, and only if the network is so large that it cannot be simulated fast enough on a serial computer.

11.2 The Perceptron

PERCEPTRON

The *perceptron* is the basic processing element. It has inputs that may come from the environment or may be the outputs of other perceptrons.

CONNECTION WEIGHT

Associated with each input, $x_j \in \mathfrak{R}, j = 1, \dots, d$, is a *connection weight*,

SYNAPTIC WEIGHT

or *synaptic weight* $w_j \in \mathfrak{R}$, and the output, y , in the simplest case is a weighted sum of the inputs (see figure 11.1):

$$(11.1) \quad y = \sum_{j=1}^d w_j x_j + w_0$$

BIAS UNIT

w_0 is the intercept value to make the model more general; it is generally modeled as the weight coming from an extra *bias unit*, x_0 , which is always