2 Basic Concepts

In this chapter, we introduce the basic neural network models used throughout the book, which date back to ideas developed in the 1940s and 1950s, as described in the previous chapter. The basic idea is to view neural networks as parallel distributed processing systems consisting of interconnected networks of simple processing units, or "neurons". The connections have parameters, or "synaptic weights", and learning and memory processes are implemented through algorithms for adjusting these synaptic weights to influence the activity of the units in a suitable way.

The relation of these models to biological neural networks is briefly discussed at the end of the book. However, regardless of how strong the connection is, these models are the simplest and best approach we have found so far to explore the style of computing adopted by biological neural systems. This style is radically different from the style of standard computers with: holographic-based instead of tape-based information storage, and intimate colocation and intertwining of memory and computation, instead of their clear separation in digital systems.

2.1 Synapses

Biological synapses are difficult to study – among other things their size is in a range difficult to analyze with traditional instruments, roughly half way between the size of molecules and cells. However new technologies are continuously being developed and are revealing how complex and dynamical biological synapses are, how they exist in different types, how they are dependent on many cellular phenomena (e.g. gene expression, molecular transport), and how they can be characterized by several attributes (e.g. shape, surface, volume, composition). It also seems to be common for a biological neuron to have more than one synaptic contact onto a neuron to which it is connected, and synapses are organized along dendritic trees with complex geometries. It is not clear how much of this complexity is essential to neural computation, and how much is the result of having to deal with all the constraints posed by carbon-based computing.

In this book, most of the time, we take the simplified view that a neuron i can only receive one synaptic connection from neuron j, and this connection is characterized by a single real-valued parameter w_{ij}, representing the synaptic strength or efficacy. If neuron j has multiple synapses onto neuron i, we consider that they have been lumped together into the value w_{ij} which globally captures how strongly neuron j influences

https://doi.org/10.1017/9781108955652.003 Published online by Cambridge University Press

neuron i. However, by simplifying a synapse to a single number w_{ij}, we are able to focus on the most fundamental problem of learning and memory: how is information about the world stored in w_{ij}?

Before we treat the learning problem, a fundamental question is how many bits of information can be stored in a synapse. Within the models considered in this book, we will see that this fundamental question has a precise answer. Although most of the time we will work with real-valued (or rational-valued) synapses as an approximation, obviously in a physical neural system there may be limits on the dynamic range and precision of w_{ij}. And thus it can be instructive to consider also cases where w_{ij} has a finite range or a limited precision of a few bits, possibly down to a single bit [111]. In some cases, one may want to include a constraint on the sign of w_{ij}, for instance not allowing excitatory synapses to become inhibitory or vice versa. These cases will occasionally be treated and, while real-valued synapses are convenient to use in theoretical calculations and computer simulations, one should never take them to imply that an infinite amount of information can be stored at each synapse.

In order to build more intelligent systems using neural networks, it is likely that the current synaptic model will have to be expanded and new mechanisms included. For example, think about situations where one is reading the end of a long paragraph and must remember its beginning, or walking in a new environment and must remember one's path. It is unlikely that this kind of short-term information, corresponding to periods of seconds to minutes, is stored in neural activity alone. A more likely scenario is that this information is stored, at least in part, in fast synaptic weights, thus requiring the introduction of synapses with different time scales as well as erasing mechanisms. Another example of new mechanism could be the introduction of computational circuits within groups of synapses (e.g. [582]), or synapses that can modulate other synapses, giving rise to neural networks that can modulate the function of other neural networks.

2.2 Units or Neurons

In this book, we consider neural networks, consisting of simple computing elements called "units", or "gates", or "neurons". A unit with n inputs $I_1, \ldots, I_n$ and n corresponding synapses, computes an output $O = f(I_1, \ldots, I_n, w_1, \ldots, w_n)$. Most of the time, we will consider that the operation of the unit can be decomposed into two steps: the computation of a simple activation $S = g(I_1, \ldots, I_n, w_1, \ldots, w_n)$, followed by the application of a transfer (or activation) function f to produce the output $O = f(S)$.

2.3 Activations

In most cases, we will use the linear activation:

$$S = \sum_{i=1}^{n} w_i I_i \quad \text{or} \quad S = \sum_{i=0}^{n} w_i I_i. \tag{2.1}$$

https://doi.org/10.1017/9781108955652.003 Published online by Cambridge University Press

In this notation, the parameter w_i represents the synaptic strength of the connection of the ith input to the unit. Furthermore, we assume that I_0 is always set to 1 and w_0 is called the bias of the unit. In other words, the activation is simply the weighted average of all the inputs weighted by the synaptic weights, or the dot product between the input vector and the vector of synaptic weights. There is evidence that neuronal dendrites may be well suited to computing dot products [575]. Although several different non-linear transfer functions will be considered in the next section, equating the dot product to zero defines a fundamental hyperplane that partitions the neuron's input space into two halves and provides the direction of affine hyperplanes where the dot product remains constant.

In the search for greater biological realism, or more powerful computing units, it is also natural to introduce polynomial activations where S is a polynomial P of degree d in the input variables. For instance, if P is a homogeneous polynomial of degree two, then:

$$S = \sum_{i,j} v_{ij} I_i I_j. \tag{2.2}$$

Note that here v_{ij} is an interaction weight between I_i and I_j. More generally, polynomial activations of degree d require synaptic weights associated with the product of up to d inputs. As we shall see in Chapter 6, quadratic activation functions are one way of implementing gating or attention mechanisms.

2.4 Transfer Functions

Different kinds of computing units are obtained by using different kinds of transfer functions f to compute the output $O = f(S)$. We briefly review the most common ones.

2.4.1 Identity and Linear Transfer Functions

In this case, $f(S) = S$ or $f(S) = aS + b$ for some real numbers a and b. When the bias is included in the activation S through w_0, these two transfer functions are interchangeable by adjusting the weights w_i accordingly. Thus, in many cases, only $f(S) = S$ needs to be considered or implemented. If the activation S is linear ($d = 1$), this yields a linear unit. As a special case, a pooling unit that computes the average of its inputs is of course a linear unit. Networks of linear units are often dismissed as being uninteresting. As we shall see, this is far from being the case. However it is essential to introduce and study non-linear transfer functions.

2.4.2 Polynomial Transfer Functions

In this case, the activation function can be a polynomial Q of degree d. In fact, most other non-linear transfer functions described below can be well approximated, at least over a finite range, by a polynomial transfer function for a suitable degree d, depending on the precision required. Note that in general, polynomial activations and polynomial transfer functions are not equivalent, even when their degree is the same.

https://doi.org/10.1017/9781108955652.003 Published online by Cambridge University Press

2.4.3 Max (Pool) Transfer Functions

Sometimes, for instance in convolutional architectures for vision problems to be described below, it can be useful to use max pooling units of the form:

$$O = \max(I_1, \ldots, I_n). \tag{2.3}$$

Note that although listed here as a different kind of transfer function, within the general framework described above this corresponds rather to a different kind of non-linear activation function, although the distinction is not consequential.

2.4.4 Threshold Transfer Functions

A threshold (or step) function has the form:

$$f(S) = \begin{cases} 1 & \text{if } S > 0 \\ 0 & \text{otherwise,} \end{cases} \tag{2.4}$$

using a $\{0, 1\}$ formalism (also called Heaviside function), or:

$$f(S) = \begin{cases} +1 & \text{if } S > 0 \\ -1 & \text{otherwise,} \end{cases} \tag{2.5}$$

using a $\{-1, +1\}$ formalism. The latter can also be written as $f(S) = \text{sgn}(S)$. [Note that usually $\text{sgn}(0) = 0$ but the decision of what to do when S is exactly equal to 0 is usually not consequential.] Usually the two formalisms are equivalent using the affine transformations $t(x) = 2x - 1$ to go from $\{0, 1\}$ to $\{-1, +1\}$, or $t(x) = (x + 1)/2$ to go from $\{-1, +1\}$ to $\{0, 1\}$. However, it is always good to check the equivalence and look at its effects on the synaptic weights. Unless otherwise stated, we will use the $\{-1, +1\}$ formalism. If S is linear, then this case leads to the notion of a linear threshold function. If in addition the inputs are binary, this leads to the notion of a Boolean threshold gate, a particular kind of Boolean function. Likewise, if S is a polynomial of degree d, then this leads to the notion of a polynomial or algebraic threshold function of degree d. If in addition the inputs are binary, this leads to the notion of Boolean polynomial threshold gate of degree d.

To be more specific, we consider the n-dimensional hypercube $H^n = \{-1, 1\}^n$. A homogeneous Boolean linear threshold gate f of n variables is a Boolean function over H^n of the form:

$$f(I_1, \ldots, I_n) = \text{sgn}\left(\sum_{i=1}^{n} w_i I_i\right). \tag{2.6}$$

A non-homogenous Boolean linear threshold gate has an additional bias w_0 and is given by:

$$f(I_1, \ldots, I_n) = \text{sgn}\left(w_0 + \sum_{i=1}^{n} w_i I_i\right) = \text{sgn}\left(\sum_{i=0}^{n} w_i I_i\right), \tag{2.7}$$

assuming that $I_0 = 1$.

https://doi.org/10.1017/9781108955652.003 Published online by Cambridge University Press

Likewise, a homogeneous Boolean polynomial threshold gate of degree d is a Boolean function over H given by:

$$f(I_1, \ldots, I_n) = \text{sgn}\left(\sum_{J \in \mathcal{J}_d} w_J I^J\right), \tag{2.8}$$

where $\mathcal{J}_d$ denotes all the subsets of size d of $\{1, 2, \ldots, n\}$. If $J = \{j_1, j_2, \ldots, j_d\}$ is such a subset, then $I^J = I_{j_1} I_{j_2} \cdots I_{j_d}$, and $w = (w_J)$ is the vector of weights representing interactions of degree d. Note that if the variables are valued in $\{-1, 1\}$, then for any index i, $I_i^2 = +1$ and therefore integer exponents greater than 1 can be ignored. Alternatively, one can also define homogeneous polynomial threshold functions where the polynomial is homogenous over $\mathbb{R}^n$ rather than H^n. A non-homogenous Boolean polynomial threshold gate of degree d is given by the same expression:

$$f(I_1, \ldots, I_n) = \text{sgn}\left(\sum_{J \in \mathcal{J}_{\leq d}} w_J x^J\right). \tag{2.9}$$

This time $\mathcal{J}_{\leq d}$ represents all possible subsets of $\{1, 2, \ldots, n\}$ of size d or less, including possibly the empty set associated with a bias term. Note that for most practical purposes, including developing more complex models of synaptic integration, one is interested in fixed, relatively small values of d.

As we shall see, it is useful to consider networks of Boolean threshold gates and compare their properties to networks consisting of other Boolean gates, such as unrestricted Boolean gates, or standard Boolean gates (AND, OR, NOT). In particular, it is important to keep in mind the fundamental difference between a linear threshold gate and a standard gate. A standard gate does not have a memory, or at best one could say that it has a frozen memory associated with the list of patterns it maps to one. In contrast, the units we consider in this book have an adaptive memory embodied in the synaptic weights. Instead of being frozen, their input-output function varies as a function of the information stored in the weights. The adaptive storage is the fundamental difference, and not whether a certain class of transfer functions is better than another one.

2.4.5 Rectified and Piecewise Linear Transfer Functions

The rectified linear transfer function, often called ReLU (Rectified Linear Unit), is a piecewise linear transfer function corresponding to:

$$O = f(S) = \max(0, S). \tag{2.10}$$

Thus the function is equal to 0 if $S \leq 0$, and equal to the identity function $f(x) = x$ if $S \geq 0$. The ReLU transfer function is differentiable everywhere except for $S = 0$ and one of its key advantages, which will become important during gradient descent learning, is that its derivative is very simple: it is either 0 or 1. Leaky ReLU transfer functions have a small non-zero slope for $S \leq 0$. More generally, it is possible to consider more general piecewise linear (PL) transfer functions with multiple hinges and different slopes between the hinges (e.g. [13]). Useful sub-classes are the even PLs satisfying $f(-x) = f(x)$, and the odd PLs satisfying $f(-x) = -f(x)$. As we shall see, it is also

https://doi.org/10.1017/9781108955652.003 Published online by Cambridge University Press

possible to learn the parameters of the transfer functions, in this case the location of the hinges or the slope of the linear segments, resulting in adaptive PLs (APLs). If needed, there are also continuously differentiable approximation to PLs using polynomial transfer functions of suitable degree or, other approximations. For instance, the ReLU can be approximated by the integral of the logistic function described below.

2.4.6 Sigmoidal Transfer Functions

Sigmoidal transfer functions can be viewed as continuous differentiable versions of threshold transfer functions. Over the $(0, 1)$ range, it is common to use the logistic function:

$$f(S) = \frac{1}{1 + e^{-S}}. \tag{2.11}$$

As we shall see, in many cases $f(S)$ can be interpreted as a probability. The derivative of the logistic function satisfies: $f'(S) = f(S)(1 - f(S))$. If necessary, the location and slope of the fast growing region can be changed by using the slightly more general form:

$$f(S) = \frac{1}{1 + Ce^{-\lambda S}}, \tag{2.12}$$

where C and λ are additional parameters. In this case, $f'(S) = \lambda f(S)(1 - f(S))$. Over the $(-1, +1)$ range, it is common to use the tanh transfer function:

$$f(S) = \tanh S = \frac{e^{S} - e^{-S}}{e^{S} + e^{-S}}, \tag{2.13}$$

which satisfies $f'(S) = 1 - \tanh^2 S = 1 - (f(S))^2$. Other sigmoidal functions that are more rarely used include: $f(S) = \arctan S$ and $f(S) = S/\sqrt{1 + S^2}$ and their parameterized variations.

2.4.7 Softmax Transfer Functions

Starting from the vector $(I_1, \ldots, I_n)$ it is sometimes desirable to produce an n-dimensional output probability vector $(O_1, \ldots, O_n)$, satisfying for every i $O_i \geq 0$, and $\sum_i O_i = 1$. This is typically the case in classification problems where multiple classes are present and O_i can be interpreted as the probability of membership in class i. The softmax, or normalized exponential, transfer function (with a slight abuse of language) can be written as:

$$O_i = \frac{e^{I_i}}{\sum_j e^{I_j}}, \tag{2.14}$$

or

$$O_i = \frac{e^{\lambda I_i}}{\sum_j e^{\lambda I_j}}, \tag{2.15}$$

with an additional parameter $\lambda > 0$, which plays the role of an inverse temperature in relation to the Boltzmann distribution in statistical mechanics (see below). At high temperature, $\lambda \to 0$ and the distribution approaches the uniform distribution. At low

https://doi.org/10.1017/9781108955652.003 Published online by Cambridge University Press

temperature, $\lambda \to \infty$ and the distribution becomes increasingly concentrated where the maximum of I_j occurs. When $n = 2$, each output can be written as a logistic function of the corresponding difference of the inputs: $O_1 = 1/(1 + e^{-(I_1 - I_2)})$. For every i, the partial derivatives of the softmax transfer function satisfy: $\partial O_i / \partial I_i = O_i(1 - O_i)$ and for $j \neq i$ $\partial O_i / \partial I_j = -O_i O_j$. When there is a temperature parameter, these formula must be multiplied by the factor λ. Note also that there are many other ways of producing probability vectors, for instance: $O_i = I_i^2 / \sum_j I_j^2$.

2.5 Discrete versus Continuous Time

In most engineering applications of neural networks today, neural networks are simulated on digital machines where time is simply ignored. More precisely, time considerations are handled artificially by external computer programs that decide when to update the activity of each neuron, when to update each synaptic weight, and so forth using a discretized version of time. Quantities, such as how long it takes for a signal to propagate from one neuron to the next, or for a synapse or neuron to update itself, are completely ignored.

However, in a physical neural system, time is continuous and its role is essential. Various continuous-time models of neurons exist, and can themselves be simulated on digital machines. However these models are seldom used in current applications and will not be treated extensively in this book. Usually the continuous-time aspect of these systems is modeled using differential equations that may lead to neuronal activities that are either smoothly continuous or produce spikes at precise times. For example, [69], we can consider a network of n interacting neurons, where each neuron i is described by a smooth activation function $S_i(t)$, which can be thought of as a "voltage" and a smooth output function $O_i(t)$, governed by the equations:

$$\frac{dS_i}{dt} = -\frac{S_i}{\tau_i} + \sum_j w_{ij} O_j + I_i \quad \text{and} \quad O_i = f_i(S_i), \tag{2.16}$$

where τ_i is the time constant of neuron i, I_i is an external input to neuron i, and f_i is the transfer function of neuron i, for instance a logistic or tanh function. The logistic output could be further interpreted as a short-term average firing rate, or as the probability of producing a spike if a continuous-time spiking model is required.

A different version of this model can be written as:

$$\frac{dS_i}{dt} = -\frac{S_i}{\tau_i} + \alpha_i f_i\left(\sum_j w_{ij} S_j\right) + I_i \quad \text{and} \quad O_i = f_i(S_i). \tag{2.17}$$

It is easy to show that provided all the time constants are identical ($\tau_i = \tau$) and the matrix (w_{ij}) is invertible, the systems described by Equations 2.16 and 2.17 are equivalent. The discretization of Equation 2.17 with $\tau_i = \alpha_i = \Delta t = 1$ yields back the standard discrete models described above.

These single-compartment models were already well-known in the 1970s and 1980s [22, 664, 355, 204] and they were applied to, for instance, combinatorial optimization

https://doi.org/10.1017/9781108955652.003 Published online by Cambridge University Press

problems [356, 357] and speech recognition [720]. More detailed models of biological neurons, using multiple compartments, were also developed in the early 1980s and continue to be developed today [415, 141, 342, 142, 170]. The issues of whether spikes play a fundamental computational role, or other auxiliary roles, such as energy conservation, is still not clear. Spiking models are also used in neuromorphic chips. While detailed compartmental models of spiking neurons may be too slow to simulate for applicaitons, there are several simplified spiking models that enable the simulations of large spiking networks on digital machines, from the early model of FitzHugh–Nagumo [275, 527] to its modern descendants [373].

2.6 Networks and Architectures

Now that the basic models for units and synapses are in place, we want to think about the circuits that can be built with them. The words circuit, network, and architecture are used more or less interchangeably in the field, although they sometimes have slightly different connotations. The word architecture, for instance, is often associated with a notion of "design"; often it is also used to refer to the topology (i.e. the underlying directed graph) of a network, irrespective of the particular values of the synaptic weights.

To learn complex tasks, it is natural to consider large neural networks with both visible and hidden units. The visible units are the units at the periphery of the architecture, where "external" inputs can be applied, or outputs can be produced. All the other units are called non-visible units, or hidden units.

2.6.1 The Proper Definition of Deep

In loose terms, the depth of a circuit or network refers to the number of processing stages that are required to transform an input into an output. In circuit theory [201], the word deep typically refers to circuits where the depth scales like n^α ($\alpha > 0$), where n denotes the size of the input vectors. This is not the definition used in this book.

In neural networks, the word "deep" is meant to establish a contrast with "shallow" methods, epitomized by simple kernel methods [657]. Some have used loose definitions of depth (e.g. at least half a dozen layers), or arbitrary cutoffs (e.g. at least three layers). Instead, it is preferable to use a simple but precise definition.

As we shall see, the learning problem is relatively easy when there are no hidden units. However, as soon as there are hidden units, the question of how to train their incoming weights becomes tricky. This is sometimes referred to as the credit assignment problem in the literature. For these reasons, in this book, we define as shallow any architecture that contains only visible units, and as deep any architecture that contains at least one hidden unit.

Hence in the next chapter we will first study shallow architectures, with no hidden units. In the meantime, it is worth defining a few broad classes of architectures and establish the corresponding notation.

https://doi.org/10.1017/9781108955652.003 Published online by Cambridge University Press

2.6.2 Feedforward Architectures

A feedforward architecture with n neurons is an architecture that does not contain any directed cycles. Alternatively, it is an architecture where the neurons can be numbered in such a way that a connection from neuron i to neuron j can exist if and only if $i < j$. The source and sink neurons are visible and correspond to inputs and outputs respectively. All other neurons are hidden. In a feedforward architecture, the numbering of the neurons can also provide a natural ordering for updating them. The first part of the book focuses on feedforward architectures.

2.6.3 Recurrent Architectures

A recurrent architecture is any architecture that contains at least one directed cycle. To define how the architecture operates, one must also specify in which order the neurons are updated (e.g. synchronously, stochastically, according to a fixed order). By discretizing time and using synchronous updates, a recurrent architecture with n neurons can always be "unfolded in time" into a feedforward layered architecture (see Chapter 9), where each layer has n neurons, and the depth of the unfolded architecture is equal to the number of time steps.

2.6.4 Layered Architectures

A layered architecture is an architecture where the units have been grouped into layers. Both feedforward and recurrent architectures can be layered, or not. In a layered feedforward architecture, connections typically run from one layer to the next. Neighboring layers are said to be fully connected if every neuron in one layer is connected to every neuron in the next layer. Otherwise the two layers are partially connected. Depending on the degree of connectivity of the layers and their arrangement, the layers can also be described as sparsely connected, or locally connected. In locally connected architectures, a neuron in one layer may receive connections only from a (small) neighborhood in the previous layer. A layered architecture may skip connections, connecting two non-consecutive layers. Intra-layer connections are also possible, and often observed in biology (e.g."inhibitory interneurons") but these are rarely used in current practical applications.

In general, a feedforward architecture with L layers of neurons (including both input and output neurons) will be denoted by:

$$A(n_1, n_2, \ldots, n_L), \tag{2.18}$$

where n_1 is the number of inputs, n_L is the number of outputs, and n_h is the number of units in hidden layer h. For instance, if we write $A(n_0, \ldots, n_L)$ the architecture has $L + 1$ layers and the input layer has size n_0. Likewise, $A(n, m, p)$ is an architecture with n input units, m hidden units, and p output units. Two layers l_1 and l_2 ($l_1 < l_2$) are said to be fully connected if every unit in layer l_1 is connected to every unit in layer l_2. A weight running from unit j in layer k to unit i in layer $h \geq k$ will be denoted by w_{ij}^{hk}.

https://doi.org/10.1017/9781108955652.003 Published online by Cambridge University Press

This notation allows both skip connections and intra-layer connections. In this case, the output of unit i in layer h can be written as:

$$O_i^h = f_i^h(S_i^h) = f_i^h\left(\sum_{k \le h}\sum_{j=1}^{n_k} w_{ij}^{hk} O_j^k\right), \tag{2.19}$$

where f_i^h is the transfer function of the unit. However, in general, there is not much benefit to be derived by allowing skip connections and, if necessary, it is usually easy to adapt a result obtained in the non-skip case to the skip case. Most often, we will also disallow intra-layer connections and assume that all the transfer functions in one layer are identical. Under these assumptions, we can simplify the notation slightly by denoting a weight running from unit j in layer $h-1$ to unit i in layer h by w_{ij}^h, keeping only the target layer as an upper index. Thus, in this most typical case:

$$O_i^h = f(S_i^h) = f\left(\sum_{j=1}^{n_{h-1}} w_{ij}^h O_j^{h-1}\right). \tag{2.20}$$

If each layer is fully connected to the next, then the total number of weights is given by:

$$W = \sum_{h=1}^{L-1}(n_h + 1)n_{h+1} \approx \sum_{h=1}^{L-1} n_h n_{h+1}, \tag{2.21}$$

where the approximation on the right ignores the biases of the units.

2.6.5 Weight Sharing, Convolutional, and Siamese Architectures

It is sometimes useful to constrain two different connections in a neural network to have the same synaptic strength. This may be difficult to realize, or even implausible, in a physical neural system. However it is easy to implement in digital simulations. This so-called weight-sharing technique is often used and one of its benefits is to reduce the number of free parameters that need to be learnt. This technique is most commonly used in the context of convolutional architectures, for computer vision and other applications. Convolutional neural networks, originally inspired by the work of Hubel and Wiesel, contain entire layers of locally connected units that share the exact same pattern of connection weights. Thus these units apply the same transformation (e.g. vertical edge detection) at all possible locations of the input field, as in a convolution operation. Furthermore, within the same level of the architecture, multiple such layers can co-exist to detect different features (e.g. edge detection at all possible orientations). In these architectures, the total number of parameters is greatly reduced by the weight sharing approach and, even more so, by having local connectivity patterns between layers, as opposed to full connectivity.

The weight-sharing technique is also used in other standard architectures. For instance, it is used in Siamese architectures where the outputs of two identical networks, sharing all their connections, are fed into a final "comparator" network. Siamese architectures are used when two similar input objects need to be processed in a similar way and then compared to assess their similarity, or to order them. This is the case, for instance, when

https://doi.org/10.1017/9781108955652.003 Published online by Cambridge University Press

comparing two fingerprint images [76], or when ranking possible source–sink pairs in predicting elementary chemical reactions [396, 397].

Several other kinds of architectures exist, such as autoencoder, ensemble, and adversarial architectures – these will be introduced later in the book. Armed with these basic architectural notions, it is natural to ask at least three fundamental questions:

(1) capacity: what can be done with a given architecture?
(2) design: how can one design architecture be suitable for a given problem?
(3) learning: how can one find suitable weight values for a given architecture?

These questions will occupy much of the rest of this book and will have to be examined multiple times for different kinds of architectures and situations. Here we introduce some of the basic concepts that will be used to deal with the questions of capacity, design, and learning.

2.7 Functional and Cardinal Capacity of Architectures

A basic framework for the study of learning complexity is to consider that there is a function h that one wishes to learn and a class of functions or hypotheses A that is available to the learner. The function h may be known explicitly or through examples and often h is not in A. The class A, for instance, could be all the functions that can be implemented by a given neural network architecture as the synaptic weights are varied. Starting from some initial function in A, often selected at random, learning is often cast as the process of finding functions in A that are as close as possible to h in some sense (Figure 2.1). Obviously how well h can be learnt critically depends on the class A and thus it is natural to seek to define a notion of "capacity" for any class A.

Ideally, one would like to be able to describe the *functional capacity* of a neural network architecture, i.e. completely characterize the class of functions that it can compute as its synaptic weights are varied. In Chapter 4, we will show that networks with a single hidden layer of arbitrary size have universal approximation properties, in the sense that they can approximate any reasonably smooth function. We will also resolve the functional capacity of feedforward linear and unrestricted Boolean architectures. However, in most of the other cases, the problem of resolving the functional capacity remains open.

In part as a proxy, we will use a simpler notion of capacity, the *cardinal capacity*, or just capacity when the context is clear. The cardinal capacity $C(A)$ of a class A of functions (Figure 2.1) is simply the logarithms base two of the size or volume of A, in other words:

$$C(A) = \log_2 |A|. \tag{2.22}$$

Given an architecture $A(n_1, \ldots, n_L)$ we will denote its capacity by $C(n_1, \ldots, n_L)$. To measure capacity in a continuous setting, one must define $|A|$ in some measure-theoretic sense. In the coming chapters, we will simplify the problem by using threshold gate

https://doi.org/10.1017/9781108955652.003 Published online by Cambridge University Press

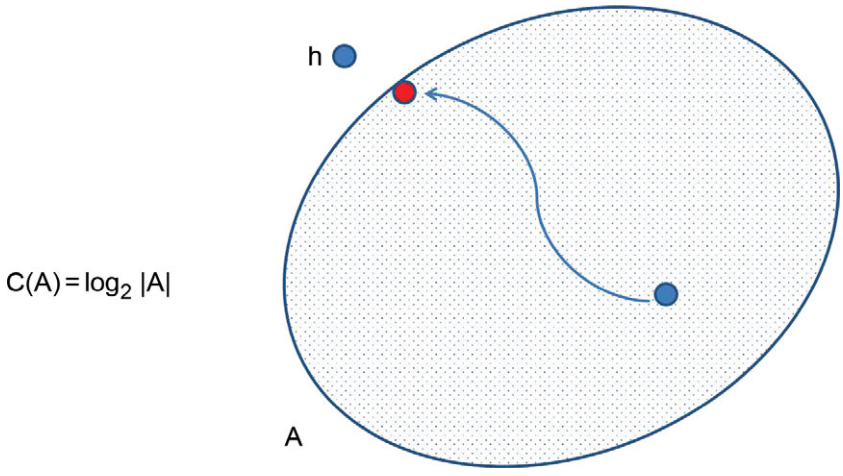

Figure 2.1 Learning framework where h is the function to be learnt and A is the available class of hypotheses or approximating functions.

neurons, so that the class A is finite (provided the inputs are restricted to be binary, or in any other finite set) and therefore we can simply let $|A|$ be the number of functions contained in A. We will show that in this discrete setting the cardinal capacity of many architectures can be estimated.

There are other notions of capacity, besides functional and cardinal capacity, that will be examined in later chapters. The kind of capacity that is being discussed should either be specified or clear from the context. But, in general, the default usage will refer to the cardinal capacity. The fundamental reason why the cardinal capacity is important is that it can be viewed as the number of bits required to specify, or communicate, an element of A. More precisely, it is the minimum average number of bits required to specify an element of A in the worst case of a uniform distribution over A. Another way to look at learning is to view it as a communication process by which information about the "world" is communicated to, and stored in, the synapses of a network. Thus the cardinal capacity provides an estimate of the number of bits that must be communicated to the architecture, and stored in the synaptic weights, so that it can implement the right function.

To address questions of design, as well as other questions, a sound statistical framework is useful. Perhaps the most elegant statistical framework is the Bayesian framework. In particular, in Chapter 3 we will show how the Bayesian framework allows one to solve the design question for shallow networks, as well as for the top layer of any architecture, under a broad set of conditions. Thus next we provide a very brief introduction to the Bayesian statistical framework.

https://doi.org/10.1017/9781108955652.003 Published online by Cambridge University Press

2.8 The Bayesian Statistical Framework

One of the most elegant and coherent frameworks to cope with data is the Bayesian statistical framework. To begin, it is useful to review the axiomatic foundations of the Bayesian statistical framework ([215, 648, 376, 124]). Broadly speaking there are at least two related kinds of axiomatic frameworks. The first kind is based on "monetary" notions, such as utility and betting. As we do not think, perhaps naively, that monetary considerations are essential to science and statistics, we choose the second framework, based on the notion of "degrees of belief".

The starting point for this framework is to consider an observer who has some background knowledge $\mathcal{B}$ and observes some data D. The observer is capable of generating hypotheses or models, from a certain class of hypotheses or models. The process by which these models are generated, or by which the class of models is changed, are outside the immediate scope of the framework.

Bayesian analysis is concerned with the assessment of the quality of the hypotheses given the data and the background model. Given any hypothesis H, the fundamental object of Bayesian statistics is the "observer's degree of belief" $\pi(H|D, \mathcal{B})$ in H, given D and $\mathcal{B}$. If the background knowledge $\mathcal{B}$ changes, the degree of belief may change, which is one of the main reasons why this approach is sometimes called "subjective" (incidentally, a major marketing error of the Bayesian camp and a main argument for the detractors of this approach). Although "subjective", the approach aims to be rational in the sense that the fundamental object $\pi(H|D, \mathcal{B})$ should satisfy a set of reasonable axioms. Usually three axioms are imposed on $\pi(H|D, \mathcal{B})$.

First π should be transitive, in the sense that given three hypotheses H_1, H_2, and H_3, if $\pi(H_1|D, \mathcal{B}) \preceq \pi(H_2|D, \mathcal{B})$ and $\pi(H_2|D, \mathcal{B}) \preceq \pi(H_3|D, \mathcal{B})$, then:

$$\pi(H_1|D, \mathcal{B}) \preceq \pi(H_3|D, \mathcal{B}). \tag{2.23}$$

Here $X \preceq Y$ is meant to represent that hypothesis Y is preferable to hypothesis X. The transitivity hypothesis essentially allows mapping degrees of beliefs to real numbers and replacing $\preceq$ with $\leq$.

The second axiom states that there exists a function $f(x)$ establishing a systematic relationship between the degree of belief in an hypothesis H and the degree of belief in its negation $\neg H$. Intuitively, the greater the belief in H, the smaller the belief in $\neg H$ should be. In other words:

$$\pi(H|D, \mathcal{B}) = f(\pi(\neg H|D, \mathcal{B})). \tag{2.24}$$

Finally, the third axiom states that given two hypotheses H_1 and H_2, there exists a function $F(x, y)$ establishing a systematic relationship such that:

$$\pi(H_1, H_2|D, \mathcal{B}) = F[\pi(H_1|D, \mathcal{B}), \pi(H_2|H_1, D, \mathcal{B})]. \tag{2.25}$$

The fundamental theorem that results from these axioms is that degrees of beliefs can be represented by real numbers and that if these numbers are re-scaled to the [0, 1] interval, then $\pi(H|D, \mathcal{B})$ must obey all the rules of probability. In particular, $f(x) = 1 - x$ and

https://doi.org/10.1017/9781108955652.003 Published online by Cambridge University Press

$F(x, y) = xy$. As a result, in what follows we will use the notation $P(H|D, \mathcal{B})$ and call it the probability of H given D and $\mathcal{B}$.

In particular, when re-scaled to $[0, 1]$, degrees of belief must satisfy Bayes' theorem:

$$P(H|D, \mathcal{B}) = \frac{P(D|H, \mathcal{B})P(H|\mathcal{B})}{P(D|\mathcal{B})}, \tag{2.26}$$

where $P(H|\mathcal{B})$ is called the prior of the hypothesis, $P(D|H, \mathcal{B})$ the likelihood of the data, and $P(D|\mathcal{B})$ the evidence. Bayes' theorem is the fundamental tool of inversion in probability and allows one to express the posterior as a function of the likelihood.

Thus, in essence, in this framework probabilities are viewed very broadly as degrees of beliefs assigned to statements (or hypotheses or models) about the world, rather than the special case of frequencies associated with repeatable events. Note that a model M can be reduced to a binary hypothesis H by asking whether M fits the data within some level of error ϵ. In general, the goal in the Bayesian framework is to estimate the posterior distribution, as well as the expectation of relevant quantities with respect to the posterior. The framework can also be applied iteratively in time, with the posterior at a given iteration becoming the prior at the following iteration, when new data is received, thus providing a natural way to update one's beliefs.

In practical situations, the elegance and flexibility of the Bayesian framework is often faced with two well-known challenges:
(1) the choice of the prior degree of belief $P(H|\mathcal{B})$; and
(2) the actual computation of the posterior $P(H|D, \mathcal{B})$, and any related expectations, which may not always be solvable analytically and may require Monte Carlo approaches ([296, 300]).

In the cases relevant to this book, one is interested in a class of models $M(w)$ parameterized by a vector of parameters w. Working with the full posterior distribution $P(w|D, \mathcal{B})$ can be challenging and thus in practice one must often resort to point-estimation approaches, such as finding the parameters with the highest posterior probability – or maximum *a posteriori* (MAP) estimation corresponding to (dropping $\mathcal{B}$ for simplicity):

$$\max_w P(w|D) \Leftrightarrow \max_w \log P(w|D) \Leftrightarrow \min_w -\log P(w|D). \tag{2.27}$$

Using Bayes' theorem, this yields:

$$\max_w P(w|D) \Leftrightarrow \min_w -\log P(D|w) - \log P(w) + P(D). \tag{2.28}$$

The evidence $P(D)$ does not depend on w and thus can be omitted from this first level of analysis, so that:

$$\max_w P(w|D) \Leftrightarrow \min_w -\log P(D|w) - \log P(w). \tag{2.29}$$

When the prior term can be ignored, as in the case of a uniform prior over a compact set, this problem reduces to maximum likelihood (ML) estimation:

$$\max_w P(D|w) \Leftrightarrow \min_w -\log P(D|w). \tag{2.30}$$

https://doi.org/10.1017/9781108955652.003 Published online by Cambridge University Press

2.8.1 Variational Approaches

This section requires some notions from information theory that are given in Section 2.9. One may want to skip this and return to it after reading that section on information theory. The Bayesian framework can be applied at multiple levels not only in time, but also across levels of modeling. For example, suppose that one is interested in modeling (and approximating) a probability distribution $R(x)$ over some space X, which itself could be for instance the posterior obtained from a previous level of modeling. Thus we view R as the "data", and we model this data using a family of distributions $\mathcal{Q} = (Q_\theta(x))$ parameterized by some parameter vector θ. In general, Q_θ consists of a family of relatively simple distributions, such as factorial distributions and distributions from the exponential family [153]. In terms of point estimation, we thus wish to estimate:

$$\min_\theta \left(-\log P(R|Q_\theta) - \log P(Q_\theta) \right). \tag{2.31}$$

For the prior $P(Q_\theta)$ it is reasonable to choose an entropic prior (see Section 2.9) where $P(Q_\theta)$ is proportional to $e^{-H(Q_\theta)}$ and $H(Q_\theta)$ is the entropy of Q (as an exercise, check that it does not matter whether one uses natural logarithms or logarithms base 2). Thus:

$$-\log P(Q_\theta) = H(Q_\theta) + C = -\sum_x Q_\theta(x) \log(Q_\theta(x)) + C, \tag{2.32}$$

where C is a constant that is not relevant for the optimization. For the log-likelihood term:

$$-\log P(R|Q_\theta) = -\sum_x Q_\theta(x) \log R(x), \tag{2.33}$$

up to constant terms. This is essentially derived from a multinomial likelihood of the form: $\prod_x R(x)^{Q_\theta(x)}$.

This general approach is commonly called the variational approach. It is often described directly, in a less principled way, as trying to find a distribution Q_θ to approximate R by minimizing the relative entropy $\mathrm{KL}(Q_\theta, R)$:

$$\min_Q \mathrm{KL}(Q_\theta, R) = \min_\theta \sum_x Q_\theta \log Q_\theta - \sum_x Q \log R. \tag{2.34}$$

Notice, more broadly, that the same Bayesian ideas can be applied more generally to the approximation of functions in analysis [238, 123]. For instance, from a data modeling perspective, approximation formula like Taylor expansions can be derived from the Bayesian framework.

We now turn to the basic concepts of information theory which play a fundamental role in deep learning and neural networks, not surprisingly, since one of the fundamental aspects of deep learning is the storage of information in synapses. These basic concepts are probabilistic in nature.

https://doi.org/10.1017/9781108955652.003 Published online by Cambridge University Press

2.9 Information Theory

What we call Information Theory was originally developed by Shannon in 1948. Shannon's fundamental insight was to avoid getting bogged down in semantic issues – what does a message means to me – and rather think of information in terms of communication – the problem of "reproducing at one point either exactly or approximately a message selected at another point" [669]. The three most fundamental concepts in information theory are those of entropy, relative entropy, and mutual information. We review them succinctly, more complete treatments can be found, for example, in [129, 213, 491]. Throughout this section, we will consider random variables X and Y with distributions $P(x)$ and $Q(y)$. When needed, we will also consider the joint distribution $R(x, y)$.

Given any random variable X with distribution $P(x)$, the entropy $H(X)$ is defined by:

$$H(X) = -\sum_x P(x) \log_2 P(x). \tag{2.35}$$

Using logarithms in base 2 ensures that the entropy is measured in bits. The entropy is the average number of bits of information gained by observing an outcome of X. It is also the minimum average number of bits needed to transmit the outcome in the absence of noise. Obviously $H(X) \geq 0$ and the entropy is maximal for a uniform distribution, and minimal for a distribution concentrated at a single point. Given a second random variable Y with distribution $Q(y)$, the conditional entropy $H(X|Y)$ is defined by the expectation:

$$E_y H(X|Y = y) = -\sum_y Q(y) \sum_x P(X = x|Y = y) \log_2 P(X = x|Y = y). \tag{2.36}$$

It is easy to check that: $H(X|Y) \leq H(X)$.

Given two distributions $P(x)$ and $Q(x)$ defined over the same space $\mathcal{X}$, the relative entropy or Kullback–Leibler divergence KL (P, Q) between $P(x)$ and $Q(x)$ is defined by:

$$\mathrm{KL}\,(P, Q) = \sum_x P(x) \log_2 \frac{P(x)}{Q(x)}. \tag{2.37}$$

Here and throughout the book, in the continuous case, the summation is to be replaced by an integral. Thus the KL can be interpreted as the expected value with respect to P of the log-likelihood ratio between P and Q. It is also the average additional cost, in terms of bits per symbol, incurred by using the distribution Q instead of the true distribution P. It is not symmetric and thus is not a distance. If necessary, it can easily be made symmetric, although this is rarely needed. Using Jensen's inequality, it is easy to see that KL $(P, Q) \geq 0$, and KL $(P, Q) = 0$ if and only if $P = Q$. Jensen's inequality, which is graphically obvious, states that if f is convex down and X is a random variable then:

$$E(f(X)) \leq f(E(X)), \tag{2.38}$$

where E denotes the expectation. In addition, KL (P, Q) is convex down with respect to P and Q. The entropy $H(P)$ can be derived from the relative entropy by computing the relative entropy with respect to the uniform distribution $U = (1/n, \ldots, 1/n)$, assuming a finite set of n possible outcomes. Using the definition, one obtains:

https://doi.org/10.1017/9781108955652.003 Published online by Cambridge University Press

$$\mathrm{KL}\,(P, U) = \log_2 n - H(P). \tag{2.39}$$

In a Bayesian setting, the relative entropy between the prior and posterior distributions (or vice versa) is called the surprise and it can be used to measure the degree of impact of the data on its observer [369, 82].

To define the mutual information $I(X, Y)$ between X and Y, we use the relative entropy between their joint distribution $R(x, y)$ and the product of the marginal distributions $P(x)$ and $Q(y)$. Both distributions are defined on the same joint space so that:

$$I(X, Y) = \mathrm{KL}\,(R, PQ) = \sum_{x,y} R(x, y) \log_2 \frac{R(x, y)}{P(x)Q(y)}. \tag{2.40}$$

A simple calculation shows that:

$$I(X, Y) = D(R, PQ) = H(X) - H(X|Y) = H(Y) - H(Y|X). \tag{2.41}$$

These quantities can be visualized in a Venn diagram where the total information $H(X, Y)$ satisfies:

$$H(X, Y) = H(X|Y) + I(X, Y) + H(Y|X). \tag{2.42}$$

It is easy to check that $I(X, X) = H(X)$. Furthermore, if X and Y are independent, then $I(X, Y) = 0$. The mutual information has a fundamental property, shared with the relative entropy: it is invariant with respect to reparameterization (e.g. rescaling). More precisely, for any invertible functions f and g:

$$I(X, Y) = I(f(X), g(Y)). \tag{2.43}$$

Finally, the data processing inequality states that for any Markov chain $X \to Y \to Z$:

$$I(X, Z) \leq I(X, Y) \quad \text{and} \quad I(X, Z) \leq I(Y, Z); \tag{2.44}$$

in other words, feedforward (or post-) processing cannot increase information.

An interesting application of information theory related to the data processing inequality is the information bottleneck method [524, 733, 677] (see exercises). This method was introduced as an information-theoretic approach for extracting the relevant information an input random variable X provides about an output random variable Y. Given their joint distribution $P(x, y)$, this information is precisely measured by the mutual information $I(X, Y)$. An optimal (compressed) representation Z of X for this task would capture the relevant features of X for predicting Y and discard the irrelevant ones. In statistical terms, the relevant part of X with respect to Y is a *minimal sufficient statistic*. Thus, in the bottleneck framework, we want to simultaneously minimize $I(X, Z)$ to optimize compression, while maximizing $I(Z, Y)$ to optimize prediction. Using Lagrange multipliers, this leads to the optimization problem:

$$\min_{P(z|x)} I(X, Z) - \lambda I(Z, Y), \tag{2.45}$$

where $\lambda \geq 0$ is a Lagrange multiplier controlling the tradeoff between compression and prediction. Another elegant framework related to information theory, which cannot be reviewed here, is information geometry [24], connecting information theory and statistical inference to geometrical invariants and differential geometry.

https://doi.org/10.1017/9781108955652.003 Published online by Cambridge University Press

We have briefly presented some of the key ideas needed to address some of the questions related to capacity and design. Beyond these questions, we remain concerned with the fundamental problem of learning from data, i.e. of extracting and storing information contained in the data about the "world" into the synaptic weights of an architecture. Thus we now briefly consider different kinds of data and learning settings.

2.10 Data and Learning Settings

Over the years different learning paradigms have emerged (e.g. supervised, unsupervised, reinforcement, on-line) and new ones are periodically being introduced. Some of these paradigms get their inspiration from biology, others are unabashedly oriented towards machines. While it is not clear how this taxonomy will evolve and what will be left standing once the fog clears up, here we briefly describe only the most basic paradigms.

One key observation is that, from an engineering perspective, learning from data can be viewed as a different way of writing computer programs, by automatically learning the programs from the data. This is particularly, but not exclusively, useful in situations where we do not know how to write the corresponding program (e.g. how to recognize a cat in an image).

2.10.1 Supervised Learning

Perhaps the clearest setting corresponds to supervised learning. In supervised learning, one assumes that for each input example there is a known desirable target. Thus the data comes in the form of input-target pairs (I, T). Note that both I and T can be multi-dimensional objects, such as pixel values in an input image and the corresponding target categories. Thus, in this case, the data typically consists of a set of such pairs $D = \{(I(t), T(t))\}$ for $t = 1, 2, \ldots, K$ which can be viewed as samples of some function $f(I) = T$ one wishes to learn. We use t as a convenient index here, but not necessarily to indicate a notion of time. Within supervised learning tasks, it is common to distinguish regression tasks where the targets are continuous, or at least numerical, versus classification tasks where the targets are binary, or categorical. The standard approach to supervised learning is to define an error function $\mathcal{E}(w)$ that depends on the weights w of the network and somehow measures the mismatch between the outputs produced by the network and the targets. The goal of learning is to minimize the error $\mathcal{E}(w)$ and we will see in later chapters that the main algorithm for doing so is stochastic gradient descent. In Chapter 3, we will see how to design the error function. A useful observation is that, at least in the cases where the underlying function f is injective, the same data can be used to try to learn the inverse function f^{-1} simply by permuting inputs and targets.

2.10.2 Unsupervised Learning

In unsupervised learning, for each input example there is no target. Thus in general the data will come in the form $D = \{I(t)\}$ for $t = 1, 2, \ldots, K$ and one would like to discover

https://doi.org/10.1017/9781108955652.003 Published online by Cambridge University Press

useful patterns in this data. A standard example of unsupervised machine learning task is clustering.

2.10.3 Semi-supervised Learning

The term semi-supervised learning is used to describe situations where both supervised and unsupervised data is available. This is quite common in situations where acquiring labels or targets is difficult or expensive. Thus typically one ends up with a small labeled data set, and a large unlabeled data set and one of the main questions is how to leverage both kinds of data in a useful way.

2.10.4 Self-supervised Learning

Self-supervised learning provides a bridge between supervised and unsupervised learning, by solving unsupervised learning problems using supervised learning methods. In self-supervised learning, the data itself provides the targets for supervised learning. This is the basic idea used behind different kinds of autoencoders, or when one trains a system to predict the next item in a sequence, such as the next image frame in a video sequence in computer vision, or the next word in a text sequence in natural language processing. Likewise, one can train a system to reconstruct an item from a partial version of it, such as completing a sequence from which a word has been deleted, or completing an image from which an object or a patch has been deleted or altered.

2.10.5 Transfer Learning

The term transfer learning is used in general in settings where one is trying to learn multiple tasks, based on the same input data, and where there may be learning synergies between the tasks that may lead to improvements over the obvious strategy of learning each task in isolation. Thus in this setting one typically tries to transfer knowledge from one task to another task, either by learning them simultaneously or sequentially. The implicit underlying assumption is that core elements common to multiple related tasks must exist that are easier to extract and learn when multiple tasks are learnt together, as opposed to each one of them in isolation. Elements of transfer learning are already present in the multi-class classification learning task.

2.10.6 On-line versus Batch Learning

Across the various learning paradigms, learning algorithms can be applied after the presentation of each example (on-line) or after the presentation of the entire training set (batch). Of course all regimes between these two extremes are possible and in practice one often uses so-called mini-batches containing several training examples. The size of these mini-batches plays a role and will be discussed in Chapter 3.

https://doi.org/10.1017/9781108955652.003 Published online by Cambridge University Press

2.10.7 Reinforcement Learning

While common, the settings above may seem somewhat unnatural from a biological standpoint or inapplicable to a large number of situations where there are active "agents" interacting with a "world" that is trying to maximize their utility. In its most basic setting, reinforcement learning assumes that there is an agent which can take a number of actions. Given the current state of the world, the action taken changes the state of the world and a certain reward is obtained. The goal of the agent is to learn how to act in order to maximize some form of long-term reward. Reinforcement learning, and its combinations with neural networks, are studied in the appendix.

2.11 Learning Rules

Within the framework adopted here, learning is the process by which information contained in the training data is communicated and stored in the synaptic weights. Given a synaptic weight w_{ij} connecting neuron j to neuron i, learning algorithms are expressed in terms of formulas, or learning rules, for adjusting w_{ij}. Typically these rules have the iterative form:

$$w_{ij}(k+1) = w_{ij}(k) + \eta F(R) \quad \text{or} \quad \Delta w_{ij} = \eta F(R). \tag{2.46}$$

Weight changes can occur after the presentation of each example in on-line learning, or after the presentation of batches of a particular size, in batch or mini-batch learning. The parameter η is the learning rate which controls the size of the steps taken at each iteration. The vector R is used here to represent the relevant variables on which learning depends, for instance the activities of the pre- and post-synaptic neurons. The function F provides the functional form of the rule, for instance a polynomial of degree two. As examples, we have already seen three learning rules in the introductory chapter where $F(R)$ is equal to $O_i O_j$ in the simple Hebb rule, $(T - O_i)O_j$ in the perceptron rule, and $B_i O_j$ in the backpropagation rule. Note that all three rules share a common form consisting of a product between a post-synaptic term associated with neuron i and a pre-synaptic term associated with neuron j, in fact equal to the activity of neuron j in all three cases. It will be useful to write such equations in matrix form when considering neurons in two consecutive layers. In this case, using column vectors, the matrix learning rule is given by the outer product between the post-synaptic vector term and the transpose of the pre-synaptic vector term. For instance, in the example above for backpropagation, one can simply write:

$$\Delta w_{ij} = \eta B_i O_j \quad \text{(scalar form)}; \qquad \Delta w = \eta B O^{\mathrm{T}} \quad \text{(matrix form)}, \tag{2.47}$$

where the column vector $B = (B_i)$ runs over the post-synaptic layer, and the column vector $O = (O_j)$ runs over the pre-synaptic layer, and O^{T} denotes transposition. Within this framework, as one varies the choice of R and F, there is potentially a very large space of learning rules to be explored. In later chapters, we will see how this space can be organized and greatly restricted, in particular using the concept of locality for learning rules.

https://doi.org/10.1017/9781108955652.003 Published online by Cambridge University Press

To study the dynamic behavior of learning rules, one can sometimes resort to geometric or convexity considerations. Alternatively, one can try to derive an ordinary or stochastic differential equation. In batch learning, the learning dynamic is deterministic and one can try to solve the difference equation (Equation 2.46) or, assuming a small learning rate ($\eta \approx \Delta t$) the ordinary differential equation:

$$\frac{dw_{ij}}{dt} = E(F(R)), \tag{2.48}$$

where E denotes expectation taken over one epoch. On-line or mini-batch learning will induce stochastic fluctuations – the smaller the batches the larger the fluctuations – which can be modeled as an additional noise term. We will see examples where this differential equation, or its stochastic version obtained with the addition of a noise term, can be analyzed.

2.12 Computational Complexity Theory

Finally, to complete these preliminaries, we give a very brief introduction to some of the most basic concepts of computational complexity theory. Please refer to standard text books (e.g. [560, 46]) for more complete treatments.

The first important distinction is between what can be computed and what cannot be computed by a Turing machine, or a computer program. For this purpose, basic notions of cardinality are necessary. Two sets, finite or infinite, have the same cardinality if there is a one-to-one correspondence (bijective function) between the two sets. An infinite set S is said to be countable if there exists a one to one function between S and the set $\mathbb{N}$ of all the integers. For instance the sets of all even integers, all signed integers, and all possible fractions are countable. The set of all real numbers $\mathbb{R}$ is not countable. Likewise, it is easy to show that the set of all finite strings over a finite (non-empty) alphabet is countable, while the set of all infinite strings over an alphabet with at least two symbols is not countable. As a result, the set of all functions from $\mathbb{N}$ to $\{0, 1\}$ is not countable, whereas the set of all possible computer programs in Python (or any other computer language) is countable. Therefore, there exists functions from $\mathbb{N}$ to $\{0, 1\}$ that are not computable, i.e. for which it is not possible to write a computer program that computes them. More broadly, there exist decision problems, i.e. problems with a yes/no answer that are not computable. The argument above provides an existential proof of problems that are not computable or decidable. For a constructive proof, a classical example is provided by the halting problem: deciding if a given program will halt or not when applied to a given input. That the halting problem cannot be solved by a computer program can be proven fairly easily by contradiction using a diagonal argument, in essence similar to the argument one uses to prove that the set of all real numbers is not computable (see exercises).

In general, the decision problems considered in this book are computable. The important question is whether they can be computed efficiently. Consider for instance the Traveling Salesman Decision Problem (TSP): given n cities and their pairwise distances,

https://doi.org/10.1017/9781108955652.003 Published online by Cambridge University Press

is there a tour traversing all the cities of length less than k? Notice that the problem of finding the shortest tour can easily be reduced to a short (logarithmic) sequence of decision problems by varying k. This decision problem has an easy solution obtained by enumerating all possible tours and their corresponding lengths. However this solution is not efficient, as the number of all possible tours ($n!$) is exponential in n and rapidly exceeds the number of all particles in the known universe as n increases. Thus one is led to define the class P of decision problems that can be solved in polynomial time, i.e. for which there is an algorithm whose running time is bounded by a polynomial function of the input size. The next useful class is called NP (non-deterministic polynomial), this is the class of problems where a proposed solution can be checked in polynomial time. Obviosuly, TSP is in NP: if one is given a particular tour α, by adding the distances along α it is easy to decide in polynomial time if the length of α is less than k or not. Thus TSP is in NP. But is TSP in P? One may conjecture that it is not, but no-one has been able to prove it. Thus one of the most important open problems in mathematics and computer science today is whether $P = NP$ or not? Finally, within the class NP one can define the sub-class NP-complete, consisting of problems that are equivalent to each other, in the sense that any one of them can be transformed into any other one of them through a transformation which has at most a polynomial cost. TSP is in NP-complete as well as hundreds of other such problems [293]. A polynomial solution for any one of these problems would lead to a polynomial solution for all the other ones, simply by using the polynomial transformations that exist between them. Thus to prove that a problem in NP is NP-complete is typically done by polynomial reduction to another NP-complete problem. A problem is NP-hard if it is at least as hard as the hardest problems in NP (i.e. every problem in NP can be polynomially reduced to it). The TSP optimization problem is NP-hard. It is possible to show that training a three-node neural network is NP-complete [132]. In Chapter 5 we will show that optimizing the unrestricted Boolean autoencoder is NP-hard.

2.13 Exercises

EXERCISE 2.1 Define a notion of equivalence between networks of threshold gates using $\{0, 1\}$ values, and network of threshold gates using $\{-1, 1\}$ values with the same architecture. Which weight transformations convert a $\{0, 1\}$ network into an equivalent $\{-1, 1\}$ network, and vice versa? Given any two distinct real numbers a and b, can this be generalized to threshold gates with $\{a, b\}$ values (i.e. the output is a if the activation is below the threshold, and b otherwise). Same questions if the threshold gates are replaced by sigmoidal gates with logistic and tanh transfer functions, respectively.

EXERCISE 2.2 Prove the equivalence between the models described by Equations 2.16 and 2.17 under the proper assumptions.

EXERCISE 2.3 Consider a logistic unit with inputs in the interval $[0, 1]$. Can it be considered equivalent, and if so in which sense, to a tanh unit with inputs in the interval $[-1, 1]$?

https://doi.org/10.1017/9781108955652.003 Published online by Cambridge University Press

EXERCISE 2.4 Which of the following Boolean functions of n arguments can be implemented by a single linear threshold gate with a bias, and why:
(1) OR;
(2) AND;
(3) NOT ($n = 1$);
(4) XOR ($n = 2$) and its n-dimensional generalization PARITY;
(5) SINGLE_u which is the Boolean function equal to $+1$ for the binary vector u, and -1 (or 0) for all other binary vector inputs; and
(6) SELECT_k which is the Boolean function that is equal to the kth component of its input.

Show that for the vast majority of pairs of distinct points x and y on the n-dimensional hypercube, there is no linear threshold functions with value $+1$ on those two points, and value -1 (or 0) on the remaining $2^n - 2$ points.

EXERCISE 2.5 Consider the hypercube of dimension n. 1) Show that any vertex of the hypercube is linearly separable from all the other vertices. 2) Show that any two adjacent vertices of the hypercube are linearly separable from all the other vertices. 3) Show that any three adjacent vertices of the hypercube are linearly separable from all the other vertices. 4) Show that any sub-cube of the hypercube is linearly separable from all the other vertices of the hypercube. A sub-cube of dimension $k \leq n$ has exactly 2^k vertices which share identical coordinates in $n - k$ positions.

EXERCISE 2.6 How many terms are there in a homogenous polynomial of degree d over H^n? How many terms are there in a homogeneous polynomial of degree d over $\mathbb{R}^n$?

EXERCISE 2.7 Consider a Boolean architecture $A(n, 2, 1)$ with n binary inputs, two Boolean functions f and g in the hidden layer, and one output Boolean function $h(f, g)$. Assume that: h is the AND Boolean function, f and g are linear (or polynomial) threshold functions with biases. In addition, we assume that all the positive examples of f have the same activation value. Show that the same overall Boolean function can be implemented by a single linear (or polynomial) threshold function of n variables with bias. [This result will be needed in Chapter 4 to estimate the cardinal capacity of an $A(n, m, 1)$ architecture. Do the units in the hidden layer need to be fully connected to the input layer for this result to be true?

EXERCISE 2.8 (CARDINAL CAPACITY) Prove that the cardinal capacity of an architecture satisfies the following properties.
Monotonicity: If $n_k \leq m_k$ for all k, then:

$$C(n_1, \ldots, n_L) \leq C(m_1, \ldots, m_L).$$

Sub-additivity: For any $1 < k < L - 1$, we have:

$$C(n_1, \ldots, n_L) \leq C(n_1, \ldots, n_k) + C(n_{k+1}, \ldots, n_L).$$

Contractivity: Capacity may only increase if a layer is duplicated. For example:

$$C(n, m, p) \leq C(n, m, m, m, p).$$

https://doi.org/10.1017/9781108955652.003 Published online by Cambridge University Press

EXERCISE 2.9 (INFORMATION THEORY) In which sense and why can an event that has probability p be communicated using $-\log_2 p$ bits? Prove that:
(1) $H(X|Y) \leq H(X)$;
(2) $KL(P, Q) \geq 0$ with equality if and only if $P = Q$;
(3) $KL(P, Q)$ is convex in P and Q;
(4) $I(X, Y) = H(X) - H(X|Y)$;
(5) $I(X, X) = H(X)$; and
(6) $I(X, Y)$ is invariant under invertible reparameterizations.

EXERCISE 2.10 (DATA PROCESSING INEQUALITY) Given a Markov Chain $X \to Y \to Z$, prove the data processing inequality, i.e. $I(X, Z) \leq I(X, Y)$. Under which necessary and sufficient conditions can there be equality $I(X, Z) = I(X, Y)$?

EXERCISE 2.11 (INFORMATION BOTTLENECK METHOD) As described in Section 2.9, consider the minimization problem:

$$\min_{P(z|x)} I(X, Z) - \lambda I(Z, Y). \tag{2.49}$$

Derive a set of consistency equations and an iterative algorithm for solving this problem in general. Further analyze the case where the variables X and Y are jointly multivariate zero-mean Gaussian vectors with covariances Σ_{XX}, Σ_{YY}, and Σ_{XY}.

EXERCISE 2.12 Consider a layered feedforward neural network and let X represent the vector of activities in one of its layers. Assuming that all of the weights of the networks are fixed, how would you define and estimate the mutual information between the input layer and X? Address this problem first in the case of binary variables and then in the case of continuous variables. This is needed for the information plane visualization method described in a later chapter.

EXERCISE 2.13 Recast a Taylor approximation formula such as: "$\cos x \approx 1 - x^2/2$ for x small" in a proper Bayesian framework. In particular describe the data, the family of approximating functions, the prior distribution, and the likelihood function.

EXERCISE 2.14 A law of physics, such as Newton's second law $F = ma$, can be viewed as a hypothesis on how nature works. Given some experimental data D consisting of various measurements of forces, masses, and corresponding accelerations, how would you compute the posterior probability of the hypothesis within a Bayesian framework? How would you compare this hypothesis to an alternative hypothesis, such as $F = ma^{1.001}$?

EXERCISE 2.15 In what sense can the use of an entropic prior be viewed as "reasonable"?

EXERCISE 2.16 Conduct a census, as complete as possible, of all the learning paradigms (e.g. supervised, unsupervised, semi-supervised) found in the literature.

EXERCISE 2.17 Prove that if the set S is infinite, S is countable if and only if there is an injective function from S to $\mathbb{N}$. Prove that the following sets are countable:

https://doi.org/10.1017/9781108955652.003 Published online by Cambridge University Press

(1) the signed numbers ($\mathbb{Z}$);
(2) the rational numbers ($\mathbb{Q}$);
(3) all finite strings over a finite alphabet;
(4) all functions from $\{0, 1\}$ to $\mathbb{N}$;
(5) all finite strings of length m or less, over an infinite but countable alphabet;
(6) all finite strings over an infinite but countable alphabet; and
(7) the union of all possible computer programs and corresponding inputs taken over all existing computer programming languages.

EXERCISE 2.18 Prove the following sets are not countable:
(1) the real numbers ($\mathbb{R}$) (by contradiction, using a diagonal argument);
(2) all infinite strings over a finite alphabet with at least two letters;
(3) all functions from $\mathbb{N}$ to $\{0, 1\}$.

Prove that there exist decision problems that are not computable, i.e. they cannot be solved by a computer program (existential proof). Prove that the halting decision problem – i.e. the problem of deciding whether, given an input, a computer program with halt or not – is not computable by contradiction, using a diagonal argument (constructive proof).

https://doi.org/10.1017/9781108955652.003 Published online by Cambridge University Press