

3 From Early Perceptrons to Deep Learning

*The Brain—is wider than the Sky—
For—put them side by side—
The one the other will contain
With ease—and You—beside—*
—Emily Dickinson, 1862

The essence of behaviorism is the belief that the study of man will reveal nothing except what is adequately describable in the concepts of mechanics and chemistry.
—Karl Lashley, “The Behavioristic Interpretation of Consciousness,” 1928

Nongrammatical Alternatives

The plot that has unfolded so far goes as follows. We humans communicate linguistically by virtue of several rule types: morphophonology (resulting in plurals like *bus-es* or *sock-s*, but not *bus-s* or *sock-es*), syntax (*Could they have run?* but not **Have they could run?*), semantics (*It is not the case that John didn't sleep last night* means the same as *John slept last night*), and pragmatics, which I did not discuss (*John has five children* is taken to mean that he has *exactly* five children, although this is not explicitly articulated).

In the foregoing, we've focused on grammatical knowledge. Two critical questions were set aside. First, how does this rich knowledge base develop in the individual heads of all English speakers (is it innate, or is it somehow acquired, and if so, how)? Second, how do speakers recruit this knowledge when speaking or listening to others speak?

It is clear that for language development and the practice of language use, we are dependent on our brain. Our main question may therefore be: **Does our brain *really* possess and follow the rules of grammar?** TGG answered

this question affirmatively, but it offered no implementation. The variety of machine learning approaches which I present below, and especially current day LLMs, answer this question with a flat **no**. The claims behind these models thus imply that the theory we have discussed is at best epiphenomenal, and they seem to offer the language sciences a fresh start. As will soon unfold, LLMs' aims, subject matter, and empirical evidence indeed differ in interesting ways from those of the linguistic approach. Linguistic research was presented as a scientific journey into the cognitive architecture of language; this chapter, by contrast, will be about language technologies and their applications, in keeping with this rapidly developing field. Are these perspectives really different? We'll soon see.

The story begins at a basic level, that of individual nerve cells and descriptions of their electrical activity. It then continues to neural networks that are capable of learning, and it travels all the way up, to the remarkable success that LLMs have had in the scientific exploration of the brain, especially the visual system. Can this success extend to language, as LLM proponents say? The answer is complicated, at the very least. But first things first; let's begin with the basic unit of a neural network, the neuron.

Origins of Computational Neuroscience—"The Neuron Doctrine"

Neurons are universally recognized as distinct, naturally occurring biological entities. This conception has been crucial to the development of the neural networks project, and hence it is a good place to begin. But the fact that neurons are distinct cellular entities, massively connected to others in special ways, was officially recognized only in 1906. Two eminent anatomists, Santiago Ramón y Cajal and Camillo Golgi, were awarded the Nobel Prize in Physiology or Medicine for that year, in appreciation of methods and discoveries relating to the brain's neural network.¹ From that time on, the scientific community has supported the notion that the brain is made out of cells, with neurons playing a central role in its functioning. It may be odd to think that as late as the beginning of the twentieth century, the idea of the neuron was a mere "doctrine"—a belief you may hold about the real world, but one lacking empirical justification. Indeed, convincing the scientific community that the brain is built out of cells was difficult, because light microscopy failed to provide a good view of brain tissue. It thus took Golgi's and Ramón y Cajal's imagination, technique and tenacity

to convince the world that like other bodily organs, the brain is made out of cells.

Once the neuron was recognized as a basic unit, scientists began studying its properties, initially by measuring its electrical activity. They understood that the effect of neurons on behavior (from vision to language behavior, abstract and complex as it may be) is what their theory seeks to explain. But how do neurons work, and how do we model them? Contributions have come from a variety of fields: mathematics, neurophysiology, electrical engineering, linguistics, and psychology. These have colluded to build a discipline that has become known as computational neuroscience.

I begin this chapter with the story of the first, twentieth-century attempts to theorize about neurons. I then move on to network models of neuronal aggregates, which have focused on the visual system. Finally, I discuss past and current deep neural networks. This will serve as an introduction to computational approaches to language, including LLMs, which will conclude this short journey. As before, precision and theoretical detail fall victim to the need for clarity, as the challenge is to present LLMs and their precursor systems at a level of detail that is sufficient for an intuitive understanding of their overall structure, using no mathematical notation. Hopefully, a reader will get enough of an idea of the main tenets of LLMs to make subsequent discussion comprehensible and effective. Here, too, slightly more technical issues are boxed, and while readers are encouraged to brave these, flow-seeking ones may skip them, hopefully without loss of continuity.

Early Neural Network Models—The McCulloch–Pitts Neuron and Perceptrons

It took forty years from the time Ramón y Cajal and Golgi's work on the neuron was recognized to the first serious mathematical characterization of this biological entity.² Physician Warren McCulloch and logician Walter Pitts presented a model that drew a connection between logical circuits and networks of neurons. The phenomenon of all-or-none neuronal spiking was already recognized, and on its basis, McCulloch and Pitts designed a network of small computational devices that resembled neurons in being binary and interconnected and in having a fixed spiking threshold (the voltage on a neuron's membrane that's required for firing); they showed that this network could solve logic problems.

Each neuron in the McCulloch–Pitts model has two states (0, 1) and is fed by many others with excitatory and inhibitory inputs. All inputs have equal weights. Excitatory inputs are aggregated in the neuron, which moves to state 1 and fires if its excitatory level exceeds a given threshold θ . If the aggregated excitatory input is below θ or if it contains *any* inhibitory input, the neuron remains silent at state 0.

McCulloch and Pitts claimed that through a choice of different θ values, this simple computational device could calculate any Boolean function (i.e., a function that operates on logical values with binary variables). That is, given n inputs, it could calculate NOT (with $\theta=0$ and inhibitory input), AND, and OR (with excitatory input and $\theta=n$ and $\theta=1$, respectively); it could thus calculate NOT(OR) and especially NOT(AND) = NAND, a key component in basic logic circuits, hence familiar to engineers.

The McCulloch–Pitts model was an important theoretical accomplishment, as it connected neurophysiology with logic and engineering for the first time. It demonstrated that a network of computational nodes that shares traits with the brain’s basic unit can work together and model a logic circuit, which can aggregate input and produce a binary output (0, 1). Still, while being a major advancement, this “neuron” was nonetheless static. The weights on the connections were all equal and fixed, and the spiking threshold θ (known as the “bias” term) was set in advance. As a result, the mode of action of the “neuron” was unchangeable on input. In other words, the McCulloch–Pitts neuron could compute basic logical equivalences, but it lacked the capacity to learn.

Fifteen years later, psychologist Frank Rosenblatt wanted to build a machine that could be trained to identify images. The McCulloch–Pitts neuron served as his point of departure, and he proposed important modifications: he allowed the fixed thresholds and weights of input units to be varied and modifiable, and he canceled the absolute power of inhibitory “synapses.”³

These new features provided flexibility, which opened the door to an algorithm that, given certain external inputs (“a training set” or “teacher”), could modify the values of the weights of incoming inputs. Rosenblatt understood that the modifiability of these weights would bring about a change in the machine’s capabilities, and he put emphasis on this feature. In other words, he proposed a sketch of the first learning rule that was connected to a binary classifier—a device that could produce a yes/no (or true/false) decision in an

image recognition task. He called the machine that materialized his ideas a *perceptron*, and he tried to implement a learning rule within it. As the purpose of the machine was to recognize images, he embedded the algorithm in a vision device. Using multiple detectors, he expected his perceptron to learn to identify objects in its visual field. He reasoned that a working perceptron, which would be built out of neuron-like units, would be a good model of the behaving brain, as it would illustrate how complex human behavior follows directly from the electrical activity of neuronal aggregates.

The ingredients of the first perceptron (without the learning rule) are shown in figure 3.1. The inputs from other units (“neurons”) are weighted, summed, and added to the bias term θ , which can be chosen anew for each problem. This is converted into an output (0, 1).

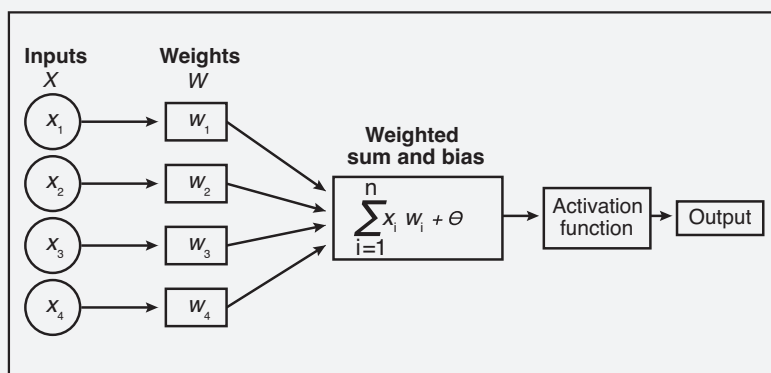


Figure 3.1

This perceptron has input and output layers, and a weight is associated with each input node; the input nodes feed into the unit, where all inputs are summed up and added to the bias term θ . This sum is passed through an activation function—a binary classifier whose job is to map a continuous numerical value onto 0 or 1 (true/false).

This setup has multiple parameters: the input column X has as many numbers as there are “neurons”; then there is a column of weights, which may now be variable, described by the vector W —a shorthand for the column of numbers it stands for, w_1, w_2, w_3, w_4 , which is as long as the number of input units—and θ . The activation function (typically a sigmoid) determines the output, based on the weighted sum plus bias. (Again, no learning rule is represented in the diagram.)

Learning in a Perceptron?

A learning rule is valuable only if it is provably capable of learning its task on a reasonably small input set and within a reasonably short training cycle. But let's reflect on what learning in a perceptron might be. The schema in figure 3.1 shows inputs x_i being assigned different weights w_i as they are passed on to be summed up. Suppose that we set all weights at the same value (let's say that $w_1 = w_2 = w_3 = w_4 = 1$); this means that the contributions of all inputs are equal. If this setup leads to a correct output, then the program works well, and no change in its internal state is necessary. But if the output on its first try is incorrect and if the program has a way of knowing it, then the way for it to try to improve its next output is to change the relative contributions of each input node, by changing their corresponding weights—perhaps x_1 is less important than the others, and thus its weight w_1 needs to be reduced? To illustrate, let's take the English plural example from chapter 1, and let's suppose that the program's task is to pluralize correctly two types of nouns: *drop* → *drop-s*; *clog* → *clog-z*. The program receives four letters for each noun (*drop*: $x_1 = d$, $x_2 = r$, $x_3 = o$, $x_4 = p$; *clog*: $x_1 = c$, $x_2 = l$, $x_3 = o$, $x_4 = g$, etc.), and in the absence of a rule (as the program has no access to abstract constructs such as “voiced consonant” and its effect on the plural -s), it must decide whether an -s or a -z needs to be added to the right of the sequence to produce a well-formed plural noun. Suppose further that we have a way of telling the program whether its current output is correct or incorrect and what to do in order to improve its performance on its next trial. The only modifiable elements are the weights on the connections of the network. If it starts off with the same weight assigned to all input letters, a correct answer in every trial is not more likely than pure guessing. Yet we need it to improve quickly. The English plural rule hangs on the noun's last letter, and it is on this letter that more weight should be put. Thus, weights w_1 , w_2 , w_3 should be minimized, because their inputs x_1 , x_2 , x_3 are not relevant to the phonetic shape of the plural; w_4 needs to be maximized, because it determines the correct choice. Weight modification would mean a change in the internal state of the program (as the weights would no longer be equal), which would improve its performance.

This move helps the program to focus on the most relevant element in the input sequence, but our problem is not solved yet: a relation between the input word's last consonant and the right choice between -s and -z needs to be established. Forcing a program, a machine, or an organism to change its internal state based on input amounts to *training* it; it learns, which is what we sought to accomplish in the first place. Rosenblatt understood this, and he claimed that once the right parameters were chosen, his algorithm converged on the correct weights, which meant that his perceptron could learn a rule on finite input and on a finite training cycle. The implication was that a computing machine built out of “neurons” can learn to perform a task by trial and error, presumably simulating a human thought process.

In keeping with his predecessors, Rosenblatt claimed that his algorithm had the capacity to implement any Boolean function (a function that can only return 0, 1). As it turned out, he was wrong. As a technology, performance turned out to be below expectations, and funding was soon terminated. Other ambitious, high visibility attempts to construct a rule-based machine translation device also resulted in failure, which led to their demise as well.⁴ Not much was happening in machine learning in the 1960s.

Hidden Layers

About a decade later, AI pioneers Marvin Minsky and Seymour Papert of MIT turned to a limitation of the perceptron, the issue of *linear separability*.⁵

A function is linearly separable if, in a geometric representation of its outputs, a straight line can separate those assigned 1 and those assigned 0. The graphs in figure 3.2 below illustrate this with AND, OR, and XOR (exclusive OR— x OR y but not both). The figure displays two-dimensional representations with two parameters, x_1 , x_2 , that get values 0, 1 (an empty or full dot, respectively); x_1 AND x_2 is assigned 1 just in case $x_1=1$, $x_2=1$; x_1 OR x_2 is assigned 1 just in case $x_1=0$, $x_2=1$ or $x_1=1$, $x_2=0$ or $x_1=1$, $x_2=1$; finally, x_1 XOR x_2 is assigned 1 more narrowly, namely just in case $x_1=0$, $x_2=1$ or $x_1=1$, $x_2=0$. A diagonal line is the geometric representation of the linear classifier that separates AND and OR

(continued)

successfully (left and middle panels). However, no straight line can do that with XOR (right panel).

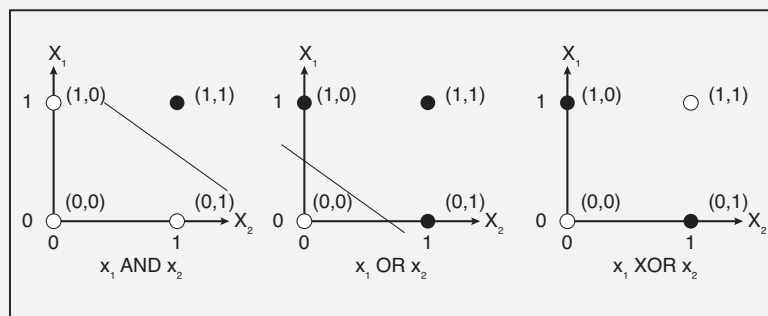


Figure 3.2

Linear separability of the logical functions AND, OR, and the inseparability of XOR: a straight line can separate the true formulas (full dots) from the false formulas (empty dots) for AND and OR but not for XOR.

Returning to the earlier goal of building a machine that calculates logical functions, Minsky and Papert demonstrated that Rosenblatt's perceptron, which had to decide whether an input is true or false (returning 1 or 0, respectively), could indeed implement Boolean functions such as AND and OR (inclusive OR, i.e., x or y or both) but failed with the function XOR (exclusive OR— x or y but not both), which is not linearly separable.

As is often the case, the discovery of an important limitation harbors rather than hinders progress. Minsky and Papert's proposed solution to the XOR problem was to extend the class of perceptrons beyond linearity. They showed that XOR is classifiable if a layer is added to the network; and not just any layer but one consisting of units that are neither input nor output layers. They proposed an intermediate and invisible, hence "hidden" layer. Such a layer has no access to the actual input or output; rather, it is completely locked in the machine, receiving a weighted sum of the input layer's output, which it passes on to the output layer.

This perceptron was indeed capable of a richer range of computations, but at a cost of increased network complexity: even if the number of units was kept unchanged, their interconnectivity grew by necessity, which is exactly what made the system go beyond linearity. Thus, Minsky and Papert's design laid the foundation for many future generations of neural

network models. Still, the added complexity left many (Minsky and Papert included) in doubt about the network's theoretical validity. The addition of hidden units was moreover anathema to many associationist psychologists, who abhorred theoretical moves for which the evidence appeared to be indirect (an objection similar to the one that led them to dismiss Chomsky's TGG). To them, the scientific justification for a hidden layer relied on reasoning that was too complex and nontransparent. The network approach to the explanation of human behavior was left by the way-side, as even its progenitors preferred the symbolic path. The first round of attempts to build intelligent machines that learn was facing a dead end.

But there were those who nonetheless continued to investigate networks and their properties. The freedom to add hidden layers and choose different values for the weights of units opened the door to richer and more powerful designs. These were mostly based on modifications of network architecture—the number and relative size of the layers, their degree of interconnectivity, and so on—and subsequent tests of their performance. An illustration of a simple network with a hidden layer is in figure 3.3A; it features an input layer, a hidden layer, an output unit with an activation function, and full connectivity, as each node is connected to all the nodes of the subsequent layer (unlike Rosenblatt's network in figure 3.1, in which each node in the input layer is connected to a single node in the output layer).

Each unit $x_{i,j}$ has two indices: the first index, i , marks the layer that the unit is in (1=input layer, 2=hidden layer), and the second, j , marks its row within its layer (1 through 4). That is, $x_{i,j}$ is the unit in the i -th column and the j -th row. The values of the units in layer 1 are thus described by the vector $X_1 = (x_{1,1}, x_{1,2}, x_{1,3}, x_{1,4})$, whereas those in the hidden layer are described by $X_2 = (x_{2,1}, x_{2,2}, x_{2,3}, x_{2,4})$.

Now, to the weights on the connections. The first two layers of nodes in this network, the input layer and the hidden layer, are fully connected. Each connection has its weight, marked by $w_{i,j}$, where i =the position of the origin and j =the position of the destination. The arrangement of both panels of figure 3.3 displays such connectivity: in 3.3A, each unit in the input layer is connected to every unit in the hidden layer; in 3.3B—a network with an additional hidden layer, labeled “output layer”—full connectivity between the input layer and the hidden layer is depicted as well as between the latter and the output layer. The final connections (vector U in 3.3A, V in 3.3B) are partial.

(continued)

As weights are now allowed to vary. Listing W and U requires a somewhat cumbersome notation. Below, I illustrate with all the weights in W (this level of details is suppressed in the figure to avoid clutter):

$w_{1, \bullet}$: the weight on the connection between the first unit in the input layer	
$x_{1, 1}$ and each unit in the hidden layer, $x_{2, 1}$ – $x_{2, 4}$	$w_{1, 1} \ w_{1, 2} \ w_{1, 3} \ w_{1, 4}$
$w_{2, \bullet}$: same for $x_{1, 2}$, the second input unit, and	
the hidden $x_{2, 1}$ – $x_{2, 4}$	$w_{2, 1} \ w_{2, 2} \ w_{2, 3} \ w_{2, 4}$
$w_{3, \bullet}$: same for $x_{1, 3}$, the third input unit, and the	
hidden $x_{2, 1}$ – $x_{2, 4}$	$w_{3, 1} \ w_{3, 2} \ w_{3, 3} \ w_{3, 4}$
$w_{4, \bullet}$: same for $x_{1, 4}$, the fourth input unit, and the	
hidden $x_{2, 1}$ – $x_{2, 4}$	$w_{4, 1} \ w_{4, 2} \ w_{4, 3} \ w_{4, 4}$

If you took the pedantic route and followed the details, you noted that W is not a vector but rather a 4×4 matrix of parameters $w_{i, j}$, where $1 \leq i, j \leq 4$. We can think about each column or row as an ordered series of values or, better yet, as a vector, which together with the weights of the other units in the input layer forms a matrix W . Likewise for U in figure 3.3B. However, U in 3.3A and V in 3.3B are vectors, each consisting of four weights. The additional hidden layer in the network in 3.3B enhances its performance.

In massively connected networks, which are currently used in AI and which are also found in neural tissue, the weights on the connections, called “synapses,” are crucial, and their values are often represented in matrices. This is where mathematicians and physicists make a contribution. Matrices are mathematical objects whose properties are well-understood. Neural networks can thus be analyzed, and conclusions and predictions that follow from these analyses can be pitted against behavioral and physiological experimental results, inspiring new experimental explorations. This has indeed happened since the mid-1980s; multilayered networks (as in the relatively simple one in figure 3.4) have been constructed and their classificatory abilities explored.

Note that we have thus far set no constraints on network connectivity. That is, no upper bound has been set on the number of units per layer, the number of layers, their similarity to one another, or the relation between rows and columns. All we know at this point (admittedly without much detail) is that, when input X passes through hidden layers, each with its own weights, the end result at the output layer is (likely) different from the input; and that this input–output difference is a result of the path the input

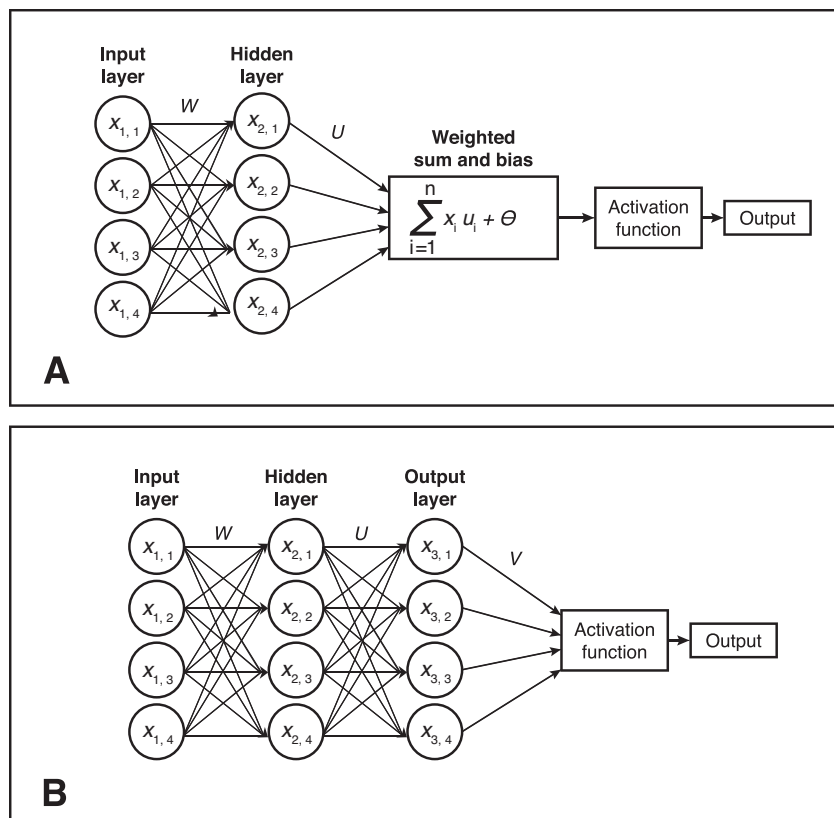


Figure 3.3
Perceptrons with one (A) and two (B) hidden layers.

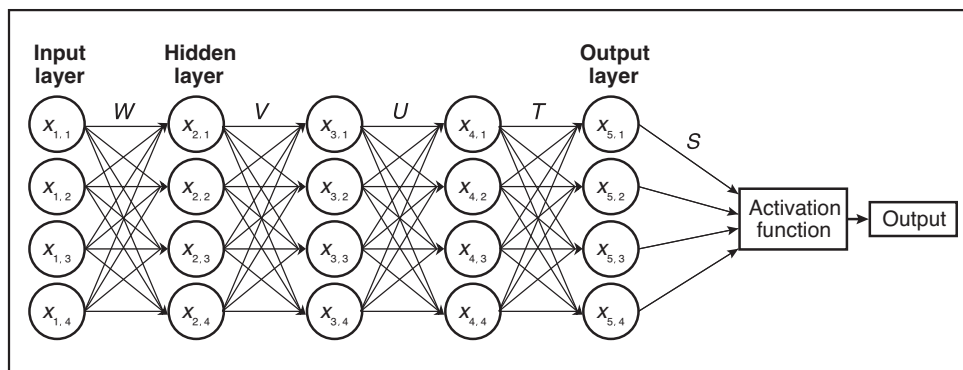


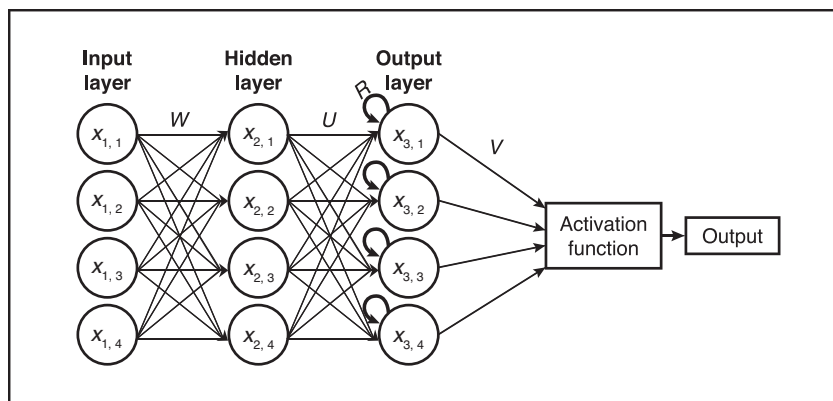
Figure 3.4
A multilayered network with weight matrices W , V , U , and T and vector S .

traverses as it is passed through the network's layers and on to the output terminal. This difference is a result of learning, which is, very generally, a change in a machine's state that is due to input. At this point, we don't know much about this input–output difference—how small or big it may be or how exactly it is brought about; but we have to acknowledge that given an input, any change in the machine's current state relative to its previous state would be indicative of learning.

Feedforward and Recurrent Neural Networks

In the above architectural designs, the unidirectionality of information flow stands out. All panels in figures 3.1–3.4 assume that the input data vector X is passed only forward (input units $\rightarrow$ hidden units $\rightarrow$ output unit). Such architectures are therefore called *feedforward networks*, and they seem to classify objects in static images at a high level of accuracy. Note that the graph in figure 3.4 allows full connectivity. That is, every node in layer 1 is connected to every node in layer 2 and so on. This may not always be the case. As we shall see later, networks that are not fully connected (a.k.a. *convolutional networks*) are closer to the biological reality of the visual system and moreover are constrained in terms of what they can learn, which is a virtue, as their power is not unlimited.

Feedforward networks don't work as well with nonstatic materials that unfold over time—tone by tone, frame by frame, or word by word—that is, with tasks in which the input's temporal structure matters. Such tasks harbor multiple connections between positions in the sequence, for which a local memory is critically needed. Engineering ingenuity augmented feedforward networks with additional components. A loop that connects a node to itself, and is fed by the value of its immediately preceding state, which may make a contribution to the node's current state, is added. Because the process recurs, these networks have been called *recurrent neural networks* (RNNs). An example of this class of networks is given in figure 3.5, in which every node in the Output layer is connected to itself (thick looped connections whose weights constitute the vector $R = (r_1, r_2, r_3, r_4)$). When we compute the value of any node at this layer (e.g., $x_{3,j}$) at time t , we take into account not only all the connections that pass information to this node now ($x_{2,1}, x_{2,2}, x_{2,3}, x_{2,4}$), but also, its own previous state at $t-1$, weighted by the values of R . Any node in a layer that has recurrence receives at time t both the

**Figure 3.5**

An RNN with weight vector R on the recurrent loop at the Output layer.

weighted input from previous layers and the weighted input from its own state $x_{i,j}$ at time $t-1$. This recurrence serves as a memory of things past (at $t-1$), which contributes to its present state t . Note that figure 3.5 features partial recurrence: the nodes that are fed not only by all others but also by themselves are only those in the Output layer.

Recurrence means that the network is no longer a feedforward network, but rather, one in which the flow of information is not unidirectional. RNNs thus help us to create a multiplicity of local memories at any given point during the processing of a sequence: one memory for a node's most recent state, one representing the relative contribution of both recent and less recent states, and so on, limited only by the machine's capacity to represent distant past state, known as the "vanishing gradient problem"—the further away you get from an input, the lesser its impact. A solution of this problem will need to weigh the importance and relevance of the current input against the importance and relevance of the context within which it is couched—the input that is more distant.

Our next step will be to extend the capacity of the memories RNNs avail us, which would optimally give a better expression of the possible of "past context." This will be captured by cleverly engineered mechanisms such as *attention* and the intentionally oxymoronic Long Short Term Memory (LSTM). Such mechanisms enhance the performance of these machines quite significantly, as will be shown below.

It is time to pause and see how far we've gotten. We have seen how the first quantitative model of a single neuron was enriched by the somewhat more complex architecture of the first perceptron, which functioned as a reasonable, but limited, classifier with minimal capacity to learn. Next came perceptrons with yet another architectural augmentation—a hidden layer—which further complicated the network but was more powerful, having gone beyond linearity. This move was historic, as it opened the way to massive connectivity, to enriched varieties of synaptic weights, which resulted in more powerful machines that could solve more problems. The feedforward nature of these networks was later augmented by loops serving as memories, which enhance their performance significantly, especially with input data that unfolds over time. These are RNNs.

This is also the time to wonder about the notion of Deep Neural Network (DNNs). What makes a neural network deep? In principle, any network with at least one intermediate, or hidden, layer (which is neither an input nor an output layer) is a deep network. But this name also implies architectural flexibility that enables the extraction of abstract, perhaps deeper, generalizations from the input. Depth, moreover, is a feature that might be closer to the actual biology; modeling brain processes may become more realistic when networks are allowed to be deeper.

Brain modeling will later be at the very center. For the moment, we stay at the machine level and move the time arrow ahead, which brings us to a critical issue: learning in networks. We have thus far assumed that the network is already trained, that is, that it has reached a steady state, hence the values of its weights are fixed. But little was said about *how* these parameters were set.

The Backpropagation Algorithm

In the animal kingdom, organisms learn through experience, whether supervised by an “instructor” (who informs them whether or not their output is correct) or unsupervised—by mere exposure to sufficient amounts of naturally occurring inputs. In machines, learning is described as a change in the machine's internal state, induced by input. In networks, such a change is brought about gradually through iterated trial and error. Typically, the network's initial output is compared to the expected behavior; if the match between the two is not perfect, an iterative process that adjusts the network's

weight system, or parameters, is initiated. If the method that determines the adjusted weights is effective, then the level of error measured in the next round will be reduced. The process is then reiterated, until the error becomes minimal, and the adjustment of parameters is stopped. If this repeated modification of the network's internal state is successful, it will eventually perform as expected. Interestingly, this construal of learning by machines is in keeping with the idea that in the biological world, learning occurs through gradual adjustment of the strength of synaptic connections.⁶ For learning machines, the readjustment of weights may be a possible solution for the learning problem that Frank Rosenblatt dreamed about. He called this process "backpropagation," but he was unable to formulate it.

How can the weights be modified to produce a network's correct performance? It took a long time before a solution was found. It was based on optimization techniques and clever mathematical tricks, assembled and connected to behavioral tasks. A landmark paper about backpropagation (or backprop, for short), which is most closely associated with this solution, was published in 1986.⁷ It describes an algorithm that compares the actual performance of a layered network to its desired output, calculates the network's output error, and on this basis recomputes the weights for each layer. The paper then demonstrated that this algorithm successfully performs several tasks.

This demonstration was powerful and reignited interest in neural networks. Over 30 years later, one of the authors (Geoffrey Hinton), together with the two other progenitors of this approach and ideas (Yann LeCun and Yoshua Bengio), would win the prestigious Turing Award, for their contributions to machine learning.⁸ Hinton would go on to win the 2024 Nobel Prize in Physics, though for a different innovation, as the Nobel citation notes (see prologue).

Let's examine backpropagation in some detail. Learning, as we said, is a change in the internal state of the network, as manifested by the values of its parameters—the weights on its connections. The goal is to teach, or "train," the network to perform exactly as expected, namely, until its output contains few, if any, errors. Error level is evaluated by comparison to a preexisting "gold" norm (the programmer's expectation of the network's ultimate performance). A clever algorithm determines small changes in the values of the network's weights, proceeding on a layer-by-layer basis, starting from the output layer and moving backwards. Once this readjustment

process is completed, the network receives input again, its performance is once more evaluated for error against the golden norm, and the algorithm makes a slight readjustment of parameter values, bringing about yet another small change in its performance. This training process is reiterated up to a point where error level (the difference between the desired and the actual performance) can no longer be minimized. If error level approaches zero, we can say that the network has reached a steady state, having learned its task.

The introduction of the backprop algorithm moved the world of networks to a new phase, for this algorithm is no longer strictly feedforward. It allows, in fact requires, that at least during the training phase, signals propagate backwards in a manner that allows the weight values to be readjusted. This means that information flow is bidirectional.

Backprop begins with the evaluation of the network's output error (calculated by the distance between an input and the correct or expected output), which it proceeds to minimize. On this basis, it calculates and implements adjustments to the network's weights that are needed for error minimization. The algorithm iterates evaluation, adjustment, and retraining on more input, and it stops only when an optimal solution—output that is as close as possible to the target output—is reached.

How is this done? Prior to training, the network is in its initial state, where all weights are randomly set. The network's output vector is therefore incorrect (which would lead to mistaken classification by the activation function). The extent of the error is calculated by a similarity, or "loss," function that extracts the output vector (values of the units in the network's last layer) and compares it to the target, or "gold," vector that the designer provides. At this stage, the weights are fixed. The comparison gives a measure of similarity (or lack thereof) between the network's current performance and its goals.

The goal here is error minimization. Any chosen loss function has a minimum (at least one). Once loss is calculated, a path to reach the function's minimum needs to be drawn. In its search for the minimum, backprop is typically guided by an analytic method named gradient descent, that determines the fastest route to the minimum of the loss function. The search consists of evaluating the current weights in each layer and deriving adjusted weight values that will get the network closer to the desired minimum. This derivation and subsequent assignment are done backwards—from last layer to first. The readjustment of the weight values in all the hidden, intermediate layers is the hardest part: the algorithm must find new weight values for all hidden

layers, and these depend on one another. The calculation process is complex, and proceeds from the Output layer backwards. A Chain Rule, invented for infinitesimal calculus by Leibnitz, enables the calculation of the rate of change of a function as a composition of change rates of a series of functions, which allows the breakdown of the calculation of weight values into a layer-by-layer process.

Backprop operates in cycles—training, error evaluation, and weight adjustment, followed by repeated training, evaluation and adjustment, and so on, until the loss function is minimized and the iterative process stops. Importantly, the effectiveness of the loss function depends on its having a single minimum. This is so because, with multiple minima, the system may get stuck in a local minimum that is not the desired one. Special measures are required to get around this problem.

The process—the loss function, the computation of gradient descent, and the calculation and subsequent readjustment of weights, followed by another input cycle—is reiterated until the minimum of the loss function is reached, and processing is then terminated. Backprop enabled the system to find a way to produce the correct results, as the network optimized its overall internal state.

Backprop applies to a rich variety of network architectures, and it exhibits an impressively broad range of abilities: from the identification of handwritten language to remarkable visual accomplishments. It was, without a doubt, a foundational step that brought us closer to the current LLMs.

Backprop and Questions About Learning

The central role of backprop in machine learning brings four important issues to the fore, especially in the context of natural language:

- *The nature of learning.* Backprop presupposes access to the target “gold” output. Error cannot be determined without such access, because a comparison between the network’s output and the expected output is required. This comparison is known as “supervised learning,” a process of change of the machine’s internal state when the target is known. Learning may not always take place in this fashion, as has been repeatedly noted. Some learning is “unsupervised,” occurring without correction: it is fairly clear that most, in fact almost all, syntactic development takes place in the absence of correction.⁹

- *Initial state.* As described, the weights in the initial, pretraining “knowledge state” of a network are randomly set. Does this feature correspond to an actual cognitive state? In the context of language, it might translate to a position on the Innateness Hypothesis, that is, on the question of whether humans possess knowledge about language that is not acquired through input.¹⁰
- *The manner of learning.* A commonsense, perhaps school-based, conception of learning (and especially teaching) is generalization. We conceive of learning by internalizing a rule as being more effective than learning by example. Machine learning experts are quick to point to the failure of past attempts to create AI-based technologies with rules, especially in the language domain. This is true, and what remains to be discussed is the significance of these failures for scientific explorations of language and linguistic ability.
- *The neurocognitive basis of backprop.* DNNs pride themselves of having much neurophysiological evidence that supports them (in chapter 8, I will review evidence for feedforward networks in the monkey’s visual system). But as backprop requires bidirectional information flow, are neural systems in the biological world bidirectional? And when it comes to humans, we should ask: in what way, if any, is backprop related to the way we learn and behave? The answer to this question is far from clear.¹¹ An anecdotal but not unimportant hint regarding this question is Geoffrey Hinton’s recent remark, “It’s not clear that the brain can do backpropagation.”¹² Here, too, the significance of these doubts about machines for scientific exploration needs to be discussed. We will do so soon.

2012 and the Explosion of Deep Learning

Superfast and more powerful computers now enable larger numbers of network nodes and layers. Larger storage capacities allow the size of the training sets to grow as well. Creative engineers have been trying out many network designs, with new mathematical and architectural tricks and innovations, which have been predicted to bring about computational explosions (some have actually occurred). The field has been brewing with more and more possibilities. But before we continue, one neglected but important point must be acknowledged: little was said above about *impossibility*—about what networks *cannot* do. This was negligent, but for a reason: in 2012, everything

about network performance changed, so there is no point in presenting the obsolete. The publication of AlexNet, an image classification network, hugely scaled up networks' performances as well as their learning sets, sweeping the field up to an entirely new level of engineering capacity.¹³

Alex Krizhevsky and his colleagues Ilya Sutskever and Geoffrey Hinton at the University of Toronto collected 1.2 million images from the web and had them crowdlabeled manually with one of 1,000 predetermined categories (crowdlabeling is made possible through websites such as Amazon Mechanical Turk; such sites harness remote labor for such needs, often from third world countries, at a cost that may be as low as \$1.77 per task).¹⁴ Training set size was the main novelty in this program, which contained eight layers in all. This program competed in a high visibility contest on September 30, 2012. It won, and its accuracy level shocked the world: image classification accuracy went up to 85%, whereas the runner-up was more than 10 percentage points behind. The leap in capacity was a landmark achievement that pioneered a new era. Predictably, the accuracy level of this program was later surpassed, as have its training set size and its number of layers. More recent machines have trained models with hundreds of layers on hundreds of millions of prelabeled items, and performance has improved. And the count keeps rising, albeit at a much slower pace, allegedly due to the law of "diminishing returns." A new era, that of *deep learning*, has begun. Let's see what effect it has had on automated language processing.

We will soon talk about the "neuron" and "synapse" metaphors and about the fruitful dialog between brain scientists and network theoreticians. But before going there, note that even the relatively small network displayed in figure 3.4 raises serious questions:

- *Size.* To approximate a real neural system, this network must be scaled up, in the number of layers and in the size of each. As nodes are massively interconnected, this increase would immediately affect the number of parameters in this network (the weights which are encoded in matrices W , V , U , T and vector S). To illustrate, figure 3.4 has five layers with four nodes each, and its weights $w_{\bullet}$ form matrices W , V , U , T , which each have $4 \times 5 = 20$ parameters or weights; there is also vector S , which has four parameters. That's a total of 24 parameters. Keeping the same network architecture but increasing the number of nodes and number of layers just by one, we get a total of $25 \times 5 + 5 = 130$ parameters; scaling it up by an order of magnitude, namely 10-fold (40 nodes per

layer; 50 layers), the number of parameters exceeds 80,000. That's a fast growth rate, and you can only imagine how easy it is to go up to the current size, which may be in the billions. Is any growth permissible, and how do such scaleups affect network performance?

- *Architecture.* Is any layer configuration possible, or is this instead constrained by computational considerations? Figure 3.4 showed a simple multilayered but two-dimensional network, which could be easily depicted on a flat screen. But how far can we go architecturally, and what does this tell us about the nature of computation and about its connection to neurobiology? These questions and many others must be discussed before these network models are taken to be scientific descriptions of human cognition. The next chapter touches on this discussion.

Taking Stock

This chapter presented the idea of a neural network and its evolution. We saw:

- An early computational perspective on single neuron activity
- The first perceptron: a network of neurons and a learning rule that does not work well
- That perceptrons with hidden units are capable of solving non-linearly separable problems
- Feedforward networks and RNNs
- The backpropagation algorithm
- The 2012 deep learning revolution

This is a section of [doi:10.7551/mitpress/15932.001.0001](https://doi.org/10.7551/mitpress/15932.001.0001)

How Deeply Human Is Language?

Chomsky, the Brain, and the AI Fantasy

By: Yosef Grodzinsky

Citation:

How Deeply Human Is Language?: Chomsky, the Brain, and the AI Fantasy

By: Yosef Grodzinsky

DOI: 10.7551/mitpress/15932.001.0001

ISBN (electronic): 9780262052023

Publisher: The MIT Press

Published: 2026



The MIT Press

The MIT Press
Massachusetts Institute of Technology
77 Massachusetts Avenue
Cambridge, MA 02139
mitpress.mit.edu

© 2026 Massachusetts Institute of Technology

All rights reserved. No part of this book may be used to train artificial intelligence systems or reproduced in any form by any electronic or mechanical means (including photocopying, recording, or information storage and retrieval) without permission in writing from the publisher.

The MIT Press would like to thank the anonymous peer reviewers who provided comments on drafts of this book. The generous work of academic experts is essential for establishing the authority and quality of our publications. We acknowledge with gratitude the contributions of these otherwise uncredited readers.

This book was set in Stone Serif and Stone Sans by Westchester Publishing Services.

Library of Congress Cataloging-in-Publication Data is available.

ISBN: 978-0-262-05200-9

EU Authorised Representative: Easy Access System Europe, Mustamäe tee 50, 10621 Tallinn, Estonia | Email: gpsr.requests@easproject.com