7 Neurobiologically Plausible Computational Models

Supplementary content at http://bit.ly/2HpAqRm

We have been traveling through the wonderful territory of the visual cortex, examining the properties of different brain areas and neural circuits, learning about how animals and their neurons respond to visual stimuli and what happens when different parts of the visual cortex are lesioned or artificially stimulated. It is now time to put all this biological knowledge into a theory of visual recognition and to instantiate this theory through a computational model that can see and interpret the world. En route toward this goal, here we start by discussing how scientists describe neural circuits using computational models and define the basic properties of neural networks.

7.1 Why Bother with Computational Models?

I have to start by admitting that I am quite biased here. Building quantitative models is *necessary* for understanding. In fact, I would go even further and claim that understanding *means* building quantitative, predictive, and falsifiable models. For computer scientists, physicists, or mathematicians, this statement may be preaching to the converted because computational models are routinely taught in courses, and building such models is a daily endeavor. However, too often, biologists or psychologists look upon computational models with suspicion and wonder why we need models at all. The curricula in biology or psychology tend to lack examples of quantitative models; instead, concepts are often conveyed through language-based frameworks and graphics that aim to describe ideas about how the visual system works.

The progression from verbal ideas to formal quantitative descriptions is a sign of maturity in a field. The language of science is mathematics, not English or Esperanto. Descriptions that are not rigorously substantiated by mathematical thinking are often imprecise, ambiguous, and prone to failure. Another problem with verbal models is that they are usually not falsifiable because word definitions are not sufficiently well articulated, and the meaning of the words may be malleable enough to account for a wide variety of findings. An even more emphatic version of this claim was elegantly articulated by Max Tegmark, a famous MIT astrophysicist, in his argument for a mathematical universe.

In the course of formulating hypotheses, designing experiments, and interpreting the results, scientists implicitly make several assumptions, consider certain intuitions to represent established facts, and jump through presumably logical connections. Quantitative models force us to think about and formalize these hypotheses and assumptions. This process of explicitly stating assumptions can help us design better experiments, discover logical flaws in our thinking, and further understand the results.

It is often the case that the same questions, or closely related questions, are analyzed from different angles, using different experimental systems, or using the same systems in different laboratories. Scientists often use qualitative descriptions of the observations, and the same words can be interpreted in substantially distinct ways, giving rise to useless discussions. Consider statements such as "we recorded high-quality multiunit activity," "the neuron was highly selective," "the neuron responded more strongly to faces than other stimuli," or "the representation was strikingly sparse." These statements are full of ambiguity.

It is not trivial to compare results across different reports. Quantitative models can integrate observations across experiments, measurements, techniques, and laboratories. Seemingly unrelated observations can be linked together using a common theoretical framework. A model can point to critical missing data, critical information, and decisive experiments. A good model can lead to non-intuitive experimental predictions. It is often the case that experimentalists rightly or wrongly believe that they can come up with predictions for the next set of experiments based on their intuitions; however, intuition often fails (unfortunately). Striking examples of how intuition can fail (at the time) include the idea that the Sun rotates around the Earth rather than the other way around, the duality of waves and particles, and the tunnel effect in quantum mechanics. We discussed multiple examples of erroneous intuitions in previous chapters, including the idea that vision is instantaneous, that vision reflects precisely what is out there in the world, and that the entire world around us has the same high resolution. The power of abstraction is critical to be able to extrapolate and push the frontiers of knowledge beyond the limitations imposed by our biases and intuitions.

In addition, a quantitative model implemented through simulations can be useful from an engineering viewpoint (we will come back to this in Section 9.5). For example, consider the problem of building algorithms that will take inputs from a digital camera and recognize objects. As we will soon see, a theoretical model that describes how the primate visual cortex recognizes objects can lead to computational algorithms with broad applicability in the real world.

Sometimes experimentalists are afraid of formulating quantitative models and feel that building such models should be the domain of computer scientists or physicists exclusively. I have often encountered brilliant scientists who seem to be reluctant to venture into the wonderful land of computational models and theoretical neuroscience. One of the reasons may be the perennial fear of mathematics. In other cases, scientists may believe that they have to be "professional theoreticians" to build quantitative models. I would strongly argue against this notion.

Some of the most provocative computational models have come from scientists who probably do not consider themselves to be theoreticians, and who spend most of their lives perfecting insightful experiments. One could provide a long list of neat computational insights put forward by experimentalists. An excellent example of a model suggested by experimentalists is the proposal for how orientation tuning arises in primary visual cortex (V1). Hubel and Wiesel, the Nobel laureates introduced in Section 5.4, discovered that V1 neurons are tuned to the orientation of a bar within their receptive fields. In addition to describing the empirical findings, they went on to propose an elegant model of how orientation tuning could arise. They considered a feedforward model that pooled the activity of multiple units in the LGN with circular center-surround receptive fields (Figure 5.7). Hubel and Wiesel proposed that orientation tuning in simple cells in V1 arises by combining the activity of LGN units with receptive fields that are aligned along the preferred orientation of the V1 neuron. Since then, there has been a large body of computational work to describe the activity of V1 neurons. The insights of Hubel and Wiesel have played a key role in inspiring generations of experimentalists and theoreticians alike: modern computational theories of vision can trace their roots to those models proposed by Hubel and Wiesel.

7.2 Models of Single Neurons

At the heart of computational models of brain function is the fundamental "atom" of computation: the neuron. I reserve the word *neuron* to refer to real biological cells and the word *unit* to refer to a computational abstraction of what a neuron does (but some people in the field use these two terms interchangeably). Many models have been proposed to describe the activity of individual neurons. These models range from the use of filter operations to describe firing rates to simulations that include dendritic spines and even individual ionic channels. We can distinguish several categories of single-neuron models, in increasing order of complexity: filter models, integrate-and-fire models, Hodgkin-Huxley models, multi-compartmental models, models including dendritic subcompartments like spines, and models that incorporate realistic geometries.

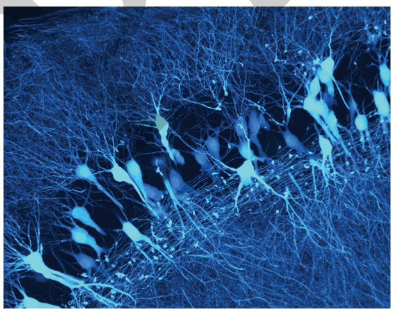
As we move from filter operations toward realistic geometries, there is a significant increase in the biological accuracy of the model. Analytical solutions become more challenging, and often nonexistent, as we increase the complexity of the model (an equation is said to have an *analytical solution* if we can explicitly write down a closed-form expression that represents the solution). There is also a concomitant increase in the computational cost of the simulations as we move toward more complex models.

More biologically accurate models are not necessarily better, if the additional realism comes at the cost of too much complexity that is not directly relevant for the task at hand. As the famous fiction writer, Jorge Luis Borges, once said: "To think is to forget a difference, to generalize, to abstract." Borges illustrated this point in a delightful short story about abstraction and maps. A map constitutes a simple everyday example of how abstract models can be extremely useful. By definition, a map abstracts away many details to reveal fundamental properties, such as how to navigate from point A to point B. A city map with a 1:1 scale (where each foot in the city is represented by a foot in the map) would be much more realistic and contain every possible detail. Such a 1:1 scale map would occupy as much space as the city itself and would not be very useful for navigation. Biological systems may seem to be resilient to abstraction; evolution cares about fitness and does not optimize for human interpretability. The random variations that accompany evolutionary time scales lead to biological systems breaking “rules” all the time and the development of complexity that “just works.”

There are several questions that we need to address to model the activity of a neuron. The answers to these questions depend on which specific aspects of the neuronal responses we are interested in capturing. Let us consider a simple analogy from fundamental physics. Imagine that we want to understand how an object of mass m – say, a cow – will accelerate as we apply a force F. We can consider a simple model that assumes that the object is a point mass – that is, that the entire mass is concentrated on a point where the force is applied – and write a one-parameter model $F = m \bullet a$. We are well aware that cows are not point masses; this assumption ignores the entire geometry of the cow. Although a trivial point, it should be noted that this one-parameter model does not do a perfect job of describing the movement of the cow in the presence of friction. Nevertheless, this simple model can capture essential ingredients of the problem, and it can even help us understand that the same principles behind the cow’s movement also explain the movement of the planets.

In a similar vein, theoreticians often ignore the geometrical shape of a neuron with its dendrites and axons (Figure 7.1A). A simple idealization considers the unit as a single compartment, where inputs are received and integrated and the output is decided. For example, in the Hubel-Wiesel model mentioned earlier (Section 7.1, Figure 5.7), one can model the activity of individual V1 neurons as a filter operating on the visual input and describe aspects of the V1 responses without getting into the details of dendritic computation, biophysics of action potential generation, or other interesting neuronal properties.

A



B

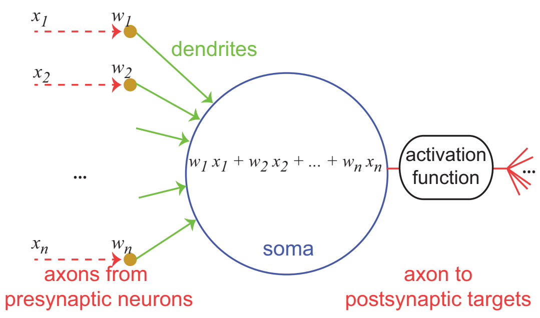

x_1
w_1
dendrites
x_2
w_2
...
$w_1 x_1 + w_2 x_2 + ... + w_n x_n$
activation function
...
x_n
w_n
soma
axons from presynaptic neurons
axon to postsynaptic targets

Figure 7.1 From real neurons to computational units. (**A**) Network of hippocampal neurons labeled with soluble tdTomato. Straub and Sabatini 2016. (**B**) A typical computational unit (blue circle) receives inputs from n presynaptic units $x_1, x_2, \ldots, x_n$. Each one of those inputs is multiplied by a synaptic weight w which controls the magnitude of the postsynaptic potential triggered by that specific synapse (orange circles). The dendrites (green) convey the information to the soma (blue), which computes a weighted sum of the inputs. A nonlinear activation function dictates the output for a given summed input level. This output is, in turn, communicated via the axons to other units.

Depending on the question, other times, it may be critical to consider multiple compartments – such as soma, axon, and dendrites. Different computations may take place depending on the location of inputs within a dendrite, and one may need to pay attention to the exact three-dimensional shape of every single axonal branch and the spatial distribution of spines and synapses on each branch. Einstein famously stated: "Make things as simple as possible, but not simpler."

A useful conceptualization of a neuron that is extensively used in neural network models is illustrated in Figure 7.1B. We can subdivide the neuron into three main compartments: dendrites, soma, and axon. Each dendrite receives inputs from another unit in the network. The presynaptic activity is denoted by x_i, with $i = 1, \dots, n$, where n represents the total number of inputs. The activity of each input unit is a scalar value, which can be coarsely thought of as the firing rate of presynaptic input i. The impact of a given presynaptic input i on the unit of interest depends on a weight factor w_i, which can be coarsely thought of as the synaptic strength between the two units. In the simplest version, each of these inputs is considered to be independent, and their contributions are linearly added into the somatic voltage z:

$$z = w_1x_1 + w_2x_2 + \dots + w_nx_n. \tag{7.1}$$

The summed activity is then passed through a nonlinear activation function to produce the output. This nonlinearity captures the notion that firing rates cannot be less than zero. It may also impose a maximum firing rate, and it may simulate other effects such as neuronal adaptation (Equation (7.2) implements only the first of these constraints). A particularly simple and commonly used activation function is the rectifying linear unit (ReLU), schematically illustrated in Figure 7.2:

$$y(z) = max\,(0, z). \tag{7.2}$$

The resulting activity y is then propagated to all the postsynaptic units. A nonlinearity such as the one in Equation (7.2) plays a critical role. First, there are whole families of functions that cannot be approximated without the introduction of nonlinearities. Second, as we will discuss soon (Section 7.4), we want to

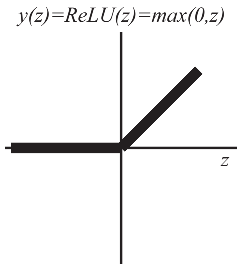


Figure 7.2 The rectifying linear unit (ReLU). A simple nonlinearity that is very popular in neural network models. The unit's activation is represented by a scalar value, loosely thought of as the "firing rate" of a real neuron. The unit receives a total input z, loosely thought of as the total summed voltage in the soma. The unit's output is rectified such that negative inputs lead to no activation, and the output is linearly proportional to z.

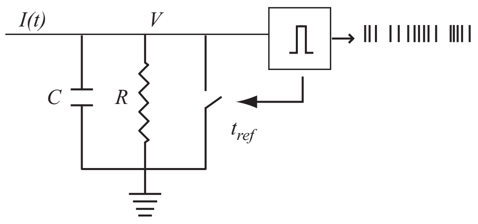


Figure 7.3 The leaky integrate-and-fire unit. The leaky integrate-and-fire model represents a neuron as an RC circuit with a capacitor *C* that integrates the incoming currents *I*(*t*) and a leaky resistor *R*. When the voltage reaches a certain threshold, a spike is emitted, and the voltage is reset. *A* refractory period t_{ref} may be imposed before emitting another spike.

combine many units to build neural networks; the output $y(z)$ will constitute the input to another unit, and so on. If all we have at our disposal are linear functions, then instead of having multiple layers of units, each one linearly summing previous inputs, we might as well combine all the steps into a single linear operation (mathematically, if $y = Ax$ and $z = By$, then we might as well write $z = Cx$). Equation (7.2) is undoubtedly an oversimplification, but it is often a useful oversimplification.

The operations illustrated in Figure 7.1 and Equations (7.1) and (7.2) do not have any internal dynamics. A step up in complexity is the *leaky integrate-and-fire model*, which dates back to 1907 and is arguably one of the most often used conceptualizations for single units in computational neuroscience. The simplest instantiation of a leaky integrate-and-fire model is a resistor–capacitor circuit (Figure 7.3). A current $I(t)$ is integrated through a capacitance C and is leaked through a resistance R. The dynamics of the intracellular voltage $V(t)$ can be described by

$$C\frac{dV(t)}{dt} = -\frac{V(t)}{R} + I(t). \tag{7.3}$$

Whenever the voltage crosses a threshold, a spike is emitted, the voltage is reset, and an absolute refractory period is imposed. This oversimplified version of a real neuron captures some of our most basic intuitions about neuronal integration. Synaptic inputs are conveyed from dendrites onto the soma where information is integrated, and an output action potential is generated when the somatic voltage exceeds a threshold. This model does not capture several biophysical phenomena including spike rate adaptation, different computations in multiple compartments, spike generation outside the soma, the sub-millisecond events during an action potential, the neuronal geometry, or other vital nuances of neurons. However, the integrate-and-fire model simulates basic properties of how inputs are integrated to give rise to outputs quite well.

It is quite straightforward to write code to simulate the dynamic behavior of integrate-and-fire units. For example, here is a simple (and not entirely correct for the aficionados) implementation of the integrate-and-fire unit in a programming language called MATLAB.

```
V(1)=V_res;                  % Initial resting voltage
for t=2:n                    % For each time in the simulation from 2 to n
      V(t)=V(t-1)+(dt/tau_m) * (E_L - V(t-1) + R_m * I_e(t));
                             % Change in voltage at time t
      if (V(t)>V_th)         % If V(t) is above threshold V_th
           spk(t)=1;         % Emit a spike
           V(t)=V_res;       % And reset the voltage to a value V_res
      end
end
```

In just a few lines, one can simulate this simple first-order differential equation and create spikes (`spk`) in response to arbitrary input currents (given by `I_e(t)`). As an example, we can set `E_L=−65 mV`, `V_res=E_L`, `V_th=−50 mV`, `tau_m=10 ms`, `R_m=10 Mohm`, `n=1000 time steps`, and `dt=0.1 ms`. We can play with different input patterns (e.g., a random input signal like `I_e=2+3* randn(n,1)`). The integrate-and-fire model can describe some of the basic instantaneous firing properties of cortical neurons. For example, when current is injected into a pyramidal neuron in cat primary visual cortex, the initial firing rate computed from the first two spikes can be well approximated by an integrate-and-fire model. Real neurons are fancier devices. Among other properties, neurons show adaptation, and the firing beyond the first two spikes is not well described by the simple integrate-and-fire model (but adjustments can be made to describe adaptation).

The integrate-and-fire unit does not capture the biophysical processes in the sub-millisecond dynamics describing the shape of action potentials. In another remarkable example of powerful intuition by experimentalists, Alan Hodgkin (1914–1998) and Andrew Huxley (1917–2012) provided fundamental insights into the generation of action potentials. They received the Nobel Prize for this work, which preceded the biological characterization of different ionic channels. The *Hodgkin-Huxley model* characterizes the shape of the action potential by incorporating the key sodium and potassium currents that are responsible for membrane depolarization and repolarization:

$$I(t) = C\frac{dV}{dt} + \bar{g}_L(V - E_L) + \bar{g}_K n^4(V - E_K) + \bar{g}_{Na} m^3 h(V - E_{Na}). \tag{7.4}$$

E_L, E_K, and E_{Na} represent the leak, potassium, and sodium reversal potentials, respectively; g_L is the leak conductance; $\bar{g}_K n^4$ describes the time and voltage-dependent potassium conductance; and $\bar{g}_{Na} m^3 h$ describes the time and voltage-dependent sodium conductance.

Again, it is straightforward to write the necessary code to simulate the dynamics in a Hodgkin-Huxley model unit. The Hodgkin-Huxley model provides a significantly richer view of intracellular voltage dynamics compared to the simpler integrate-and-fire models, and is also widely used when exploring the properties of neural networks.

7.3 Network Models

Now that we have briefly described a family of increasingly more sophisticated models of single neurons, we are going to simplify each unit in a substantial way, going back to the representation in Figure 7.1. We are going to shift the focus from individual units to the properties of networks of interconnected units. Even though each individual neuron can perform interesting computations, visual selectivity, invariance, and the ability to solve different visual tasks emerges as a consequence of the interactions that take place at the network level. We will consider networks consisting of millions of units (a recent estimate calculated that there are about 416 million neurons in macaque area V1). Because of the computational cost of studying networks with large numbers of interconnected units when each of those units themselves can perform fancy computations, the vast majority of neural network models deal with elementary units.

Even highly oversimplified units can perform interesting computations when connected in sophisticated ways. Collective computation refers to the emergent functional properties of a group of interconnected neurons. Ultimately, to understand the output of a complex system like the brain, we need to think about circuits of units and their interactions. Intuition often breaks down quickly when considering the activity of the circuit as a whole, and neural network models can help understand those emergent circuit properties. To study fluid mechanics, one can abstract from the details of the collisions and trajectories of individual molecules and instead characterize properties of the fluid such as temperature and viscosity. Similarly, most neural network models idealize and simplify the component units. Networks can be built from simple electronic devices (operational amplifiers replace neurons; cables, resistors, and capacitors replace axons, dendrites, and synapses). The dynamics of neural networks systems can also be readily simulated in computers.

A typical neural network architecture involves arranging units in layers that process information sequentially. The initial layer represents the input, and we often think of the final layer as representing the output (although one might as well read out information from any of the layers). A three-layer network is schematically illustrated in Figure 7.4. Focusing on the middle layer only (gray rectangle), and assuming that the bottom of the diagram represents the input, the connections that go from the bottom layer to the middle layer are referred to as bottom-up or *feedforward* (shown in red). Without any other connections, this type of network is referred to as a bottom-up or purely feedforward network. The simplest version of a feedforward network is the *perceptron*, with a single input layer and an output. Connections between units in the same layer are referred to as *horizontal* (shown in blue, sometimes also referred to as *lateral* connections). Connections from the top layer back to the preceding middle layer are known as top-down or *feedback* (shown in green). Some investigators use the term *recurrent* connections to jointly refer to horizontal and top-down connections, but it is preferable to describe these connections separately since they can be involved in different computations. The connection strengths are characterized by strengths or weights – here denoted as W_{ij} for the

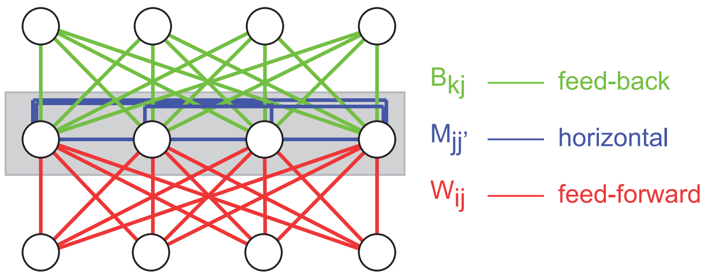


Figure 7.4 Feedforward, horizontal, and feedback connections in neural networks. Neural network models consist of multiple interconnected neuron-like units (circles here), each one of which follows the types of computations illustrated in Figure 7.1. A typical neural network architecture is to arrange units in layers. This diagram shows three layers. Assuming that the input is at the bottom of the diagram and the final output is at the top, we can distinguish feedforward connections (red), horizontal connections (blue), and feedback connections (green).

bottom-up connections, $M_{jj'}$ for the horizontal connections, and B_{kj} for the top-down connections. In the diagram in Figure 7.4, units are connected in an all-to-all fashion; that is, every unit in the bottom layer projects to every unit in the middle layer, and the same holds for all other connection types. Connectivity does not need not be all-to-all; some of the connection strengths can be set to 0 to indicate missing connections. Also, in the schematic in Figure 7.4, there are four units in each layer, and therefore all the indices *i, j,* and *k* go from 1 to 4, but this need not be the case; there could be different numbers of units in each layer. The diagram focuses on the connectivity to the middle layer, but in general, there would also be further bottom-up connections from the middle layer to the top layer and top-down connections from the middle layer to the bottom layer. In general, there would not be any horizontal connections in the bottom layer; we often think of the bottom layer as the input image. Similarly, in general, there would not be any horizontal connections in the top layer; we often think of the top layer as the output, indicating perhaps the presence of different classes of objects in the image.

Model units in neural networks may be either excitatory (positive weights) or inhibitory (negative weights). The same model unit could excite some postsynaptic targets and inhibit others. Except for a few counterexamples, this is not the case in biology, where a single neuron either provides excitatory outputs *or* inhibitory outputs, but not both.

Figure 7.4 does not constitute an exhaustive description of all the possible ways in which units can be connected in a neural network. In the most typical scenarios, units are connected within a layer (horizontal connections) or between adjacent layers (in a bottom-up and top-down fashion). However, it is also possible to build in “bypass” connections that skip a particular layer – for example, from the bottom layer to the top layer in Figure 7.4. Figure 7.5 schematically shows a variety of important neural network architectures that have been studied in the literature. This figure highlights some of the most important neural network architectures that have been used to model

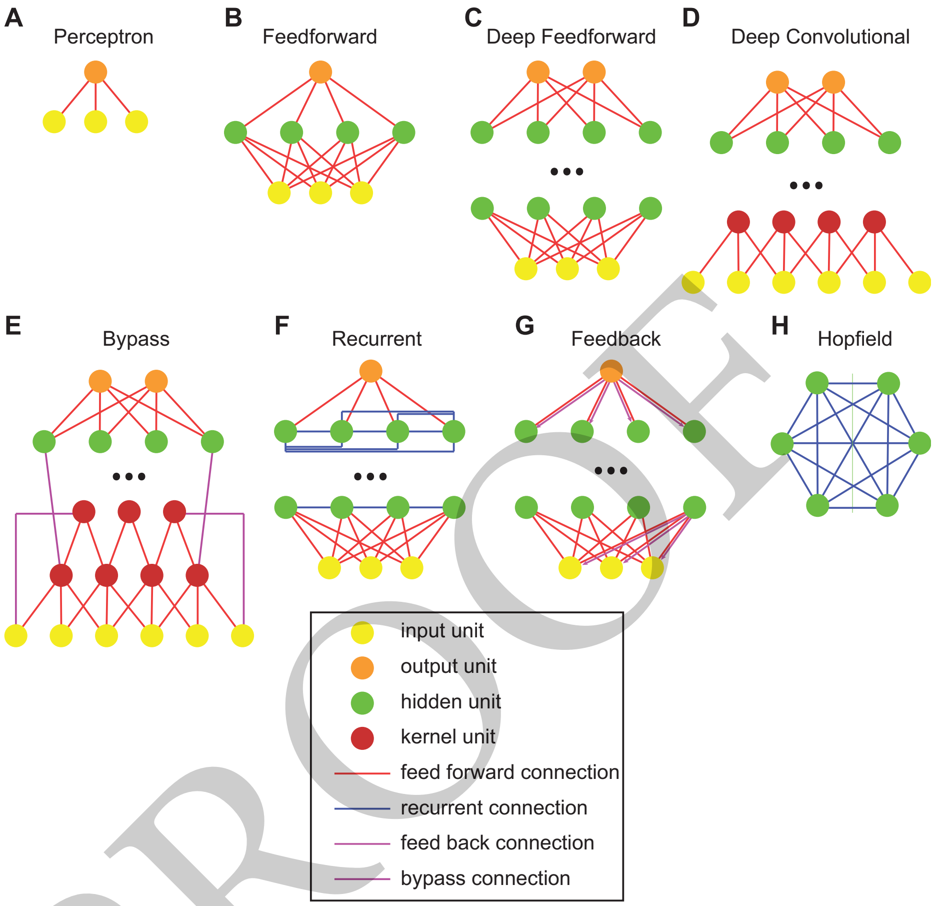


Figure 7.5 A family of neural network models. This figure illustrates a variety of important neural network models. In each diagram, the bottom layer provides the input (yellow), and the top layer provides the output (orange). Only a handful of units or layers are shown for illustration purposes. The "…" indicates that there could be many layers in between. In most cases, information flows from bottom to top via the feedforward connections (red). In **F**, there are additional recurrent connections (blue), and in **G**, there are feedback connections that provide information from an upper layer to a lower layer. The network in **H** is a different type of architecture where the units are all in the same layer, and they are reciprocally connected in an all-to-all fashion.

visual computations, but it is not exhaustive either. In all of those diagrams, only a handful of units are shown for illustration purposes, but there can be many more units.

In Figure 7.5A–E, units are organized in a cascade of layers conveying information from the bottom to the top, similar to the example in Figure 7.4, considering only the red lines. The hierarchical organization in these networks loosely resembles the hierarchical arrangement of computations in visual cortex (Figure 1.5), though even a glance of Figure 1.5 shows that current architectures only capture a small fraction of the complexity of the visual system.

7.4 Firing-Rate Network Models

Firing-rate network models constitute a simple yet instructive class of circuits. In the simplest instantiation, consider a feedforward circuit with N units projecting to a given output unit. The vector $\boldsymbol{x}$ represents the input activity. We can think of the components of $\boldsymbol{x}$ as the firing rate of each input unit. A scalar value y denotes the output firing rate. A synaptic kernel K_s describes how the input firing rate is (linearly) converted into an input current for the output unit. Theoreticians often represent the strength of a given synapse i (i =1, . . . ,N) by a scalar value w_i. This value could represent a combination of the probability of synaptic release from the presynaptic neuron and the amplitude of the postsynaptic potential (positive or negative) evoked by the incoming neurotransmitters. The total input to the output unit I_s is given by

$$I_s = \sum_{i=1}^{N} w_i \int_{-\infty}^{t} d\tau K_s(t-\tau)x_i(\tau), \tag{7.5}$$

where w_i represents the weight or strength of each synapse. Using an exponential kernel, the dynamics of this circuit can be described by

$$\tau_s \frac{dI_s}{dt} = -I_s + \sum_{i=1}^{N} w_i x_i. \tag{7.6}$$

The firing rate of the output unit is usually a nonlinear function of the total input current: $y = F(I_s)$. F could be a sigmoid function or a rectifying threshold function.

7.5 The Convolution Operation

One of the key computational ingredients of visual processing is that the same operation is typically repeated throughout the visual field. For example, we find neurons in primary visual cortex that show orientation tuning with receptive fields tiling the entire visual field. Thus, a computational operation that filters the image to extract orientation information needs to be repeated over and over throughout the image. This type of operation is readily implemented through the *convolution operation*.

Given two functions $f(t)$ and $g(t)$, the operation of convolution (in Latin, *convolvere* means "to roll together"), denoted by the symbol * in the following equation, is defined as the integral of one signal being reflected, shifted, and multiplied by the other:

$$f(t) * g(t) = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau. \tag{7.7}$$

In image processing, the process of convolution refers to shifting a given filter throughout the entire image (or the entire previous layer) and returning the output at each location. An example of this process is illustrated in Figure 7.6. For simplicity, here the input grayscale image is a handwritten version of the number 3, reduced to

Comp. by: s.Jani Stage: Revises1 Chapter No.: 7 Title Name: Kreiman
Date:22/10/20 Time:20:51:51 Page Number: 144

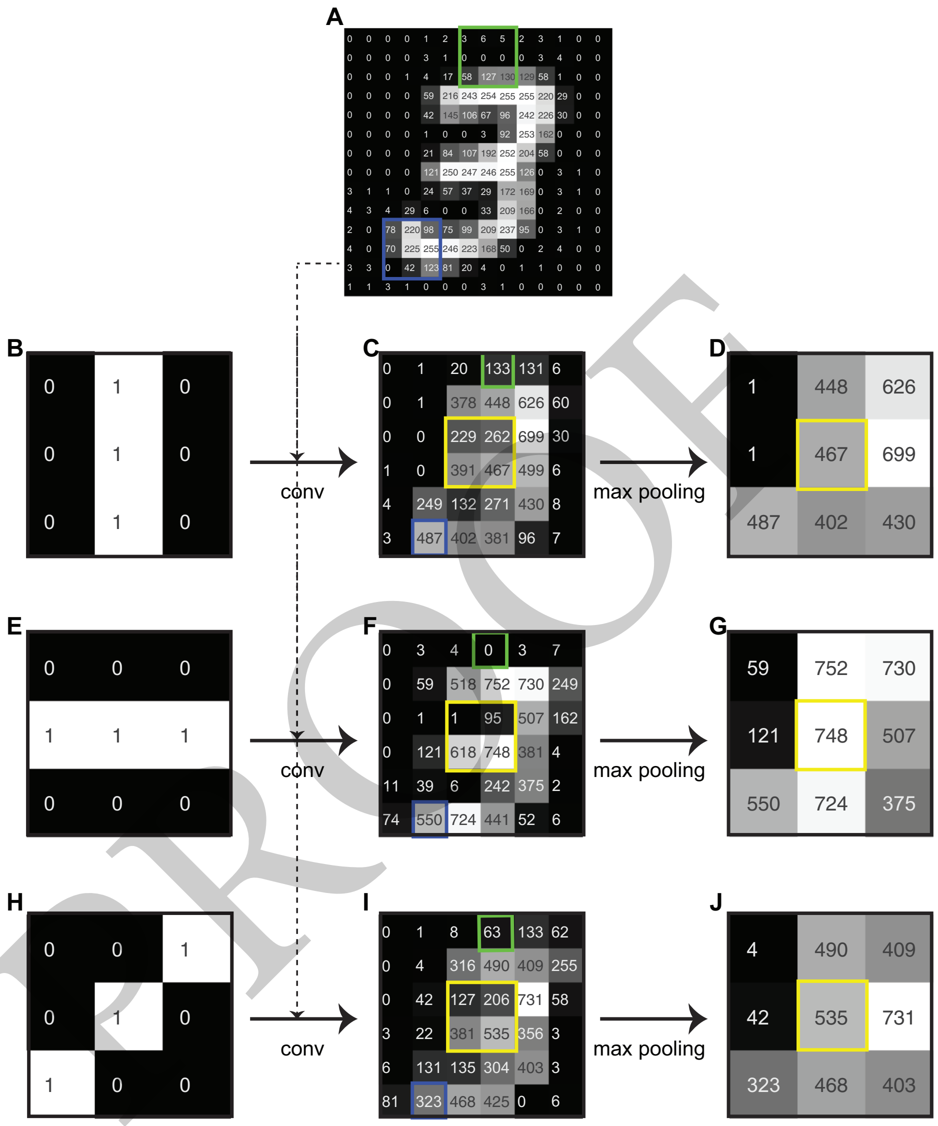


Figure 7.6 Basic operations in neural networks. A grayscale image (14 x 14 pixel image representing number 3) is convolved with three different filters (**B**, **E**, **H**). In this case, each of the filters is 3 × 3 pixels and, for simplicity, the values are only 0s and 1s. The convolution operation here has a "stride" of 2, meaning that the filter skips through one pixel as it slides through the image. The green (blue) location in the image (**A**) yields the output highlighted in green (blue) after convolution in (**C**, **F**, **I**). A pooling operation takes the output of the convolution and extracts the maximum in blocks of size 2 × 2, also with a stride of 2. The yellow location after convolution corresponds to the yellow location in the final output in **D**, **G**, **J**.

14 × 14 pixels. Each pixel has an intensity between 0 (black) and 255 (white). In general, the activation value of each unit does not need to be an integer, and the input image could have three colors, not just one. We consider three possible feature filters, shown in Figure 7.6B (vertical filter), **E** (horizontal filter), and **H** (diagonal filter). To simplify the numbers, here, the 3 × 3 pixel filter weights consist of zeros and ones, but again, in general, these filters would contain real values. The filter is placed at each location, and the filter values are multiplied by the corresponding values in the image. For example, consider the vertical filter and the green square at the top of the image containing the values 3, 6, 5 in the first row, 0, 0, 0 in the second row, and 58, 127, 130 in the third row (Figure 7.6A). We get $\mathbf{0} \times 3 + \mathbf{1} \times 6 + \mathbf{0} \times 5 = 6$ in the first row (bolded numbers come from the filter), $\mathbf{0} \times 0 + \mathbf{1} \times 0 + \mathbf{0} \times 0 = 0$ in the second row, and $\mathbf{0} \times 58 + \mathbf{1} \times 127 + \mathbf{0} \times 130 = 127$ in the third row. Adding these three numbers yields the value of 133 in the corresponding green square in Figure 7.6C. The same process is repeated throughout the entire image to yield the matrix in Figure 7.6C. Because the filter resembles a vertical line, after adequate normalization, the operation highlights regions of the input image that contain pixels that look like short vertical lines. Similarly, the filter in Figure 7.6E highlights horizontal edges, and the one in Figure 7.6H highlights diagonal edges.

We can think of these filters as a coarse approximation to simple neurons in area V1, responding to oriented lines (Section 5.4). The next step in area V1 is to pool signals from multiple simple neurons to create a complex neuron with similar tuning but responding more or less independently of the position of the preferred feature within the receptive field (Section 5.5). Inspired by the idea of simple and complex neurons, after convolution, we implement a *pooling* operation that combines multiple values within a window. This pooling operation increases the receptive field size. A typical pooling operation is to take a maximum of all the input values. For example, consider the yellow square at the center of Figure 7.6C, consisting of a 2 × 2 matrix with values 229, 262 in the first row, and 391, 467 in the second row. These four numbers are combined through the max operation to yield 467 in the corresponding yellow square in Figure 7.6D. The max-pooling operation provides position invariance by allowing high activity in any of the four locations.

The convolution and pooling operations provide a way to develop a system of hierarchical feature extraction steps. In the example shown in Figure 7.6, all the operations are fixed. In general, we are going to be interested in designing adequate filters to solve a particular problem or, even better, to learn those filters automatically. After learning to solve visual tasks, successive convolution and pooling layers in a network learn to extract progressively more complex features from the image, from edges to complex shapes and objects. We will come back to the question of how to train neural networks to learn the weights in Section 8.6.

7.6 Hopfield Networks

The dynamics of feedforward networks are quite simple, with information proceeding from one layer to the next. More elaborate dynamics can be generated in networks with recurrent

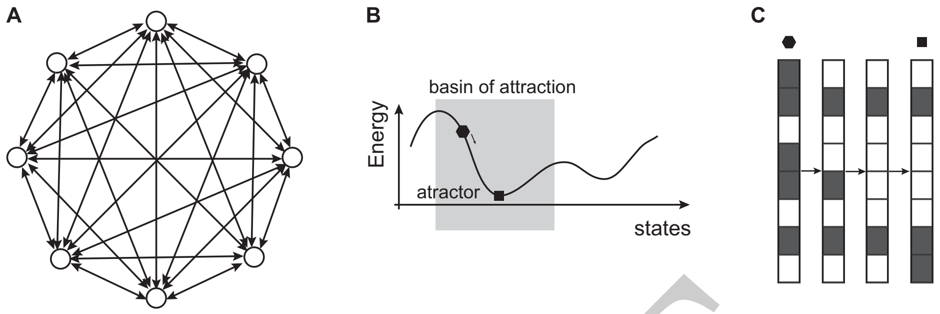


Figure 7.7 Attractor-based recurrent neural networks. (**A**) Schematic of an eight-unit Hopfield network with all-to-all connectivity and symmetric connectivity matrix ($w_{ij} = w_{ji}$). (**B**) The state of the network is characterized by an energy function with attractor states defined by the weight matrix. Starting in a state within the basin of attraction (gray rectangle) like the point represented by the hexagon will lead the network down the energy landscape to the attractor state represented by the square. (**C**) Example evolution of the network state from an initial state (hexagon) toward an attractor (square). Here each square represents the activity of a unit (gray = on, white = off). Three update states are shown here (arrows). The network can perform pattern completion because when it is initiated in a state that is close to but not identical to a memory (attractor), the dynamics will move the state toward the attractor.

connectivity. A simple yet rich example is the case of *Hopfield recurrent networks* (Figure 7.7). What is particularly attractive about these networks is that there are emergent properties of the circuit that are not easy to identify or describe upon considering only individual units without paying attention to the interactions. A Hopfield network can solve rather challenging computational problems and has interesting properties such as robustness to perturbations and the possibility of performing pattern completion.

The most basic version of the Hopfield network is defined by a single layer with binary units that are connected in an all-to-all fashion with symmetric weights. Figure 7.7A shows an example Hopfield network with eight units. Let the state of unit i at time t be represented by $s_i(t)$; this state can take the values 0 or 1 for a binary network. The network state is then represented by the vector $\mathbf{s}(t) = [s_1(t), \ldots, s_N(t)]$, where N is the total number of units (Figure 7.7C). There are no self-connections ($w_{ii} = 0$), and units are connected all-to-all in a symmetric fashion ($w_{ij} = w_{ji}$). Following Equations (7.1) and (7.2), the state of each unit is updated according to the thresholded and weighted sum of inputs from all the other units:

$$s_i(t+1) = sign\left(\sum_{j \neq i}^{N} w_{ij} s_j(t) - \theta\right), \tag{7.8}$$

where θ is a threshold. What is interesting about this type of recurrent architecture is that it is possible to define an energy function (Figure 7.7B) given by

$$E(t) = -\frac{1}{2}\sum_{i,j} w_{ij} s_i(t) s_j(t) + \sum_i s_i(t)\theta_i. \tag{7.9}$$

This energy function can be shown to be bounded below and to decrease monotonically according to the dynamics defined by Equation (7.8). In other words, the network has attractor states that it will converge to upon starting it at arbitrary states. If the network starts at a state represented by the hexagon in Figure 7.7B (state on the left in Figure 7.7C), it will dynamically evolve, always decreasing the energy of the network, until it reaches an attractor state represented by the square in Figure 7.7B (state on the right in Figure 7.7C).

Now suppose that we want to store a series of patterns $\mu = 1, \ldots, m$, defined by the state of each of the units: $\epsilon_1^\mu, \ldots, \epsilon_N^\mu$. We can use a Hebbian learning rule to calculate the weights of the units in the Hopfield network:

$$w_{ij} = \frac{1}{m}\sum_{\mu=1}^{m}\epsilon_i^\mu \epsilon_j^\mu. \tag{7.10}$$

These patterns define attractor states for the network. If we initialize the network at some arbitrary state, as long as that state is within the *basin of attraction* of a given attractor, the network state will evolve toward the corresponding attractor (Figure 7.7B and C).

From an implementation standpoint, a recurrent network with discrete time steps can be "unrolled" to convert it into a feedforward network with shared weights. For example, three time steps of a recurrent network with eight units can be implemented as a four-layer feedforward network with eight units in each layer, with all-to-all connections, and where the weights from one layer to the next are all the same across layers. For the Hopfield recurrent network, the lack of self-connections implies setting the weights from unit *i* in a given layer to unit *i* in the next layer to 0, and the symmetric connectivity matrix implies setting the weights from unit *i* in a given layer to unit *j* in the next layer equal to the weight from unit *j* to unit *i* in the next layer.

Despite this equivalence between recurrent and feedforward networks, the recurrent connectivity offers several advantages. First, the recurrent network requires fewer units (if *T* is the number of recurrent steps, the number of units in the feedforward equivalent network is *T*+1 times the number of units in the recurrent network). In biology, the size of the brain matters a great deal because of weight constraints and especially because of energetic constraints. The brain is particularly expensive from an energetic standpoint. Size and energy consumption considerations may also be relevant for certain computational applications such as implementing computer vision algorithms in a smartphone. Second, the recurrent network also requires fewer weights (again by a factor of *T*+1). The number of weights is also important in terms of size constraints in biology.

Furthermore, a critical advantage of recurrent networks is their computational flexibility. In a recurrent network, the architecture does not need to specify the number of steps, *T*, ahead of time. Some problems may be harder and require rumination during more steps, whereas other problems may be easier and require fewer steps. In contrast, the feedforward equivalent network offers a rigid structure where the computations always must traverse all the *T*+1 layers. To add flexibility and circumvent this problem, some feedforward networks include bypass connections where information processing can skip certain layers (Figure 7.5E). Achieving the full flexibility of the Hopfield network via bypass connections would require connecting every layer to every other layer, leading to an enormous increase in the number of weights. Most deep neural network models only include a small subset of all possible bypass connections.

One criticism of Hopfield networks is that there is no evidence of all-to-all connectivity in biological circuits. However, there is extensive evidence of partial horizontal connections between neurons within a given layer in cortex, and these connections can bring the multiple benefits outlined here: efficiency in space and energy requirements, flexible computations, and pattern completion. Another consideration is that reciprocal connections where unit *i* connects to unit *j* and unit *j* connects to unit *i* are the exception rather than the rule in biology, especially if the strength has to be symmetrical.

7.7 Neural Networks Can Solve Vision Problems

How can neural networks solve any type of vision problem? Let us consider a simple visual recognition task. Imagine that we have a set of images consisting of handwritten versions of the number 3 and handwritten versions of the number 7 (Figure 7.8A). Humans can look at each picture and rapidly tell that the one on the left is a 3 and the one on the right is a 7. Now consider a neural network. The exact architecture of the neural network is not relevant for the moment; we can think of any of the network architectures in Figure 7.5 for now, and we will have more to say about different architectures in Sections 8.3–5. The input to the network is the intensity of every pixel in the image. The size of the examples in Figure 7.8A are 16×16 pixels, so there would be 256 input units (a vector of 256 numbers concatenating all the rows in the image matrix). The activation of each input is an intensity value from 0 (black) to 255 (white). Each image would have a different combination of those 256 values. As we discussed in Section 2.11, we can think of these numbers as a coarse rendering of the firing rates of retinal ganglion cells in response to the image.

We can try to classify the images directly based on those 256 values. Alternatively, we can build a neural network, such as the one in Figure 7.4, with 256 input units (instead of the four inputs shown in that figure). Armed with many examples of 3s and 7s, the neural network can be *trained* to adjust the connection strengths to learn suitable features that may make it easier to separate the two groups of images. In Section 8.6, we will discuss how those connection strengths can be adjusted. For the moment, let us assume that we have already trained the network. After training, the neural network extracts a set of features from each image. We can represent those features and plot all the images in a multidimensional graph like the one in Figure 7.8B, where each point corresponds to a different image. The number of dimensions corresponds to the number of features – that is, the number of units in the neural network, before the classification layer. The output classification layer will have as many units as the number of classes to separate – in this case, two output units: one indicating the presence of a 3 and another one indicating the presence of a 7.

We can think of the cascade of computations that take place from one layer to the next as the set of computations that happen from the retina to visual cortex (Chapters 5 and 6). Using this mapping, we can think of the activation of each of the output units as a coarse rendering of the firing rates of neurons in the visual cortex. The exact area in

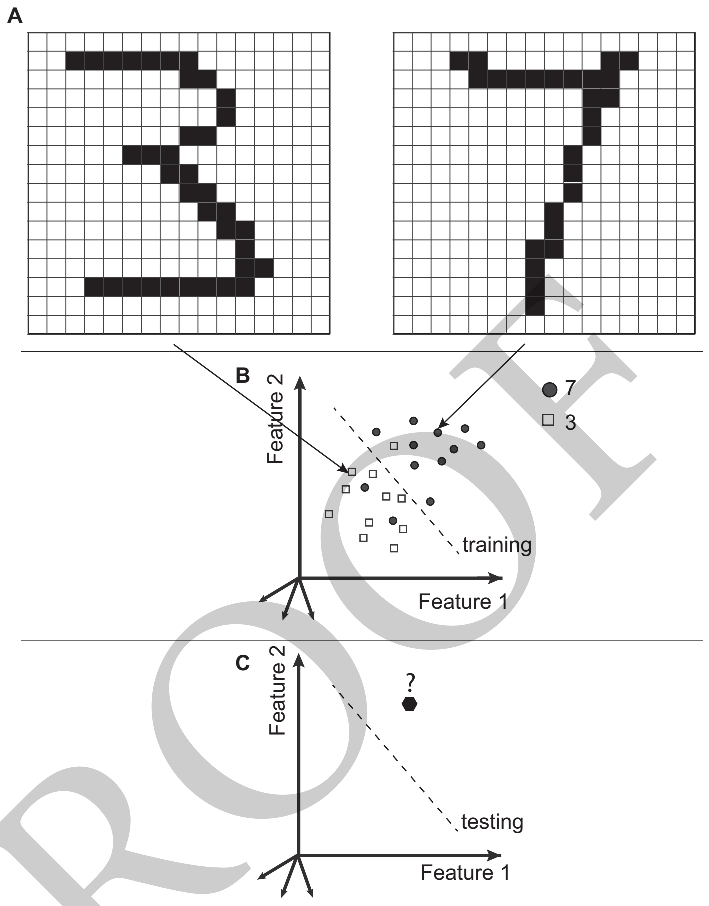


Figure 7.8 Schematic example of a vision problem solved by a neural network. **(A)** Consider a set of many images representing handwritten digits 3 and 7, only two of which are shown here. (**B**) The pixel intensities can be fed onto a neural network that will extract a set of features. Each image can then be represented by a point in a multidimensional space consisting of multiple features. Here all the 3s are represented by white squares and all the 7s by gray circles. The dataset is used to train a classifier (schematically represented here by the dashed line) to separate the two types of images. (**C**) Given a new image that was not used during training, the classifier will label it as 3 or 7, depending on which side of the line it falls.

the visual cortex is not relevant for the current discussion here; we will come back to comparisons between neural networks and the responses of neurons in different parts of the visual cortex in Section 8.14.

In Section 6.7, we illustrated how a classifier can learn to discriminate between different types of pictures based on the firing rates of a population of neurons (Figure 6.4). We can now use the same procedure to classify the images based on the features extracted by the neural network. The dashed line in Figure 7.8B represents the classifier: an image is classified as a 3 if the point falls “below” the dashed line in this graph and as a 7 otherwise. Of course, the dashed line corresponds to a hyperplane in a high-dimensional space because, generally, there are more than just two features. Additionally, the procedure can be readily extended to a classification problem with multiple classes, not just two (for example, classifying all ten handwritten digits, Section 8.10).

If we are now presented with a new image – that is, an image that has *not* been used to train the neural network – we can again compute the activation values and plot the new test image on the same graph (Figure 7.8C). The classifier can thus assign a label of 3 or 7 to the new image. Extending the mapping between units in a network and neurons in the brain, we may think that the activity of a population of neurons in visual cortex is read out by neurons in another brain area that is ultimately responsible for our ability to say "this image is a 3" and "that image is a 7." In Chapter 8, we will dig deeper into the architectures of neural networks (Section 8.5), how well they map onto the cascade of computations throughout ventral visual cortex (Section 8.14), and how well they can explain visual behaviors (Sections 8.12 and 8.13).

7.8 Extreme Biological Realism: The "Blue Brain" Project

Before ending this chapter, we come back to the notion of computational models and abstraction. Many biologists strongly feel that oversimplified networks like the ones described here fail to capture the complexity and richness of neurobiological circuitry. This observation is, of course, completely accurate.

At the other end of the spectrum in network models, one encounters efforts like the "Blue Brain" project. This project aims to introduce a significant amount of biological realism, using sophisticated and intensive network simulations. The ambitious goal is to create an in silico replica of a rodent brain, maybe even a human brain one day. In contrast to the abstractions used in neural networks, the project intends to create biophysically more realistic simulations of individual neurons, and incorporate direct data about neuronal shapes and interconnections between neurons.

Current neural networks, even in the simplified and abstracted format of Figure 7.4, have an enormous number of tunable parameters (Section 8.9). Building biologically detailed models of neural circuitry adds many orders of magnitude of complexity in terms of the numbers of free parameters. For example, should models consider the detailed geometry of every dendrite, the distance between neurons, the amount of myelin surrounding each axon, the distinct biophysical properties of the myriad different types of interneurons? The list of biological properties goes on and on. For many of these additional parameters, we still do not have sufficient data to constrain realistic models. Even if we did have enough experimental data to constrain the enormous parameter space, it is not immediately apparent that we would want to include all the minutia of the biological machinery. The previous brief discussion regarding the appropriate level of abstraction and realism in modeling single neurons is equally applicable here in the context of network models.

7.9 Summary

- To understand vision, it is essential to build quantitative computational models.
- We use models with varying degrees of abstraction, where biological properties are simplified to extract basic computational principles.
- The integrate-and-fire neuron consists of a leaky integrator and captures essential properties of how inputs to a neuron are converted into output activity.
- The convolution operation allows extracting the same visual features throughout the entire visual field.
- Basic elementary computations include filtering, normalization, pooling, and nonlinearities.
- Combining multiple units leads to neural network models with emergent computational properties that ultimately boil down to the combination of simple elementary steps.
- Neural networks typically include feedforward connections, horizontal connections, and top-down connections.
- Mixing these different types of connections, it is possible to construct a wide variety of different neural network architectures.
- Attractor-based recurrent neural networks like the Hopfield network can show interesting dynamic properties that save energy, provide flexible computational power, and show robustness to perturbations.
- Neural networks can solve vision problems.

Further Reading

See more references at http://bit.ly/2HpAqRm

- Dayan, P., and Abbott, L. (2001). *Theoretical neuroscience*. Cambridge: MIT Press.
- Gabbiani, F., and Cox, S. (2010). *Mathematics for neuroscientists*. London: Academic Press.
- Hopfield, J. J. (1982). Neural networks and physical systems with emergent collective computational abilities. *PNAS* 79: 2554–2558.
- Koch, C. (1999). *Biophysics of computation*. New York: Oxford University Press.
- Markram, H. (2006). The blue brain project. *Nat Rev Neurosci* 7: 153–160.