CHAPTER 2

The methodology of simulations

Why build models?

The idea of building formal models to test a theory of behavior is not a new one, nor original with connectionists. Models in general play an important role in the behavioral sciences; one has only to look through a text in mathematical psychology, in economics, or in sociology to see this. In fact, a model is simply a detailed theory. Actual computer simulations of models are perhaps not so common, but they play a special role in connectionism.

There are several reasons why computer simulations are so useful. First, there is the matter of explicitness. Constructing a model of a theory or an account, and then implementing it as a computer program, requires a level of detail which is not required by mere verbal description of the behavior. These details may turn out to be crucial for testing the model. Even the process of converting the model to a program can be useful, because it encourages us to clarify our thinking and consider aspects of the problem we might have skipped over if we were simply describing the theory in words.

Second, there is the fact that it is often difficult to predict what the consequences of a model are, especially if it is at all complicated. There may be interactions between different parts of our model which we cannot work out in our heads. Connectionist models have the additional problem that, although in some ways they are very simple, they involve processing elements which are nonlinear. While linear systems can sometimes be analyzed in advance, nonlinear systems frequently need to be simulated and then studied empirically. This is common practice among physicists who study nonlinear dynamics,

for example; much of their work is empirical. Frequently the only way to see how a model will behave is to run it and see.

Third, we find that the unexpected behaviors exhibited by the simulations can suggest new experiments we can run with humans. We often construct our models to fit some behavioral data we are interested in understanding. But once we've got the simulation up and running, we can see how it will respond in novel situations. In many cases, these responses are unanticipated and let us make predictions about how humans would respond in these untested situations. We can run these experiments, and if our models are good—and we are lucky!—the new data may bear out the predictions. If they don't, that's useful information too, because it lets us know there is something wrong in the model.

Fourth, there may be practical reasons why it is difficult to test a theory in the real world. For example, if we have hypotheses about the effects of brain damage on human subjects, our "experiments" are limited to the tragedies of nature. The paucity of available data may make it difficult to see global patterns. With a model, we can systematically vary parameters of interest through their full range of possible values. This often reveals patterns which we might have otherwise missed. We can then go search for corresponding instances in the real world, armed with our model's predictions. Models can thus help stimulate new empirical work, as well as describe existing data.

Finally, and perhaps most important, simulations play an important role in helping us understand *why* a behavior might occur. Sometimes it is enough to just build a simulation and have it do what we want. But in most cases, what we're after is an explanation. When we work with human subjects, we build hypotheses, based on observed behaviors, about why they act as they do. We try to be creative and think of clever ways to test our hypotheses. We almost never can get inside the subject to verify whether our theories are accurate. Our computer simulations, on the other hand, are open for direct inspection. We need not be content with simply making theories about their behavior. We can actually open them up and peer at the innards. Doing this is not always easy, but we think it is one of the most important reasons to do simulations. For this reason, we will spend a lot of time in this book talking about network analysis and about understanding why networks work the way they do. We will also focus on trying to understand the basic principles which underlie the

simulations so that we can go beyond the specifics of any given simulation.

We should add a word of caution here. Computer models play a similar role in behavioral analysis as animal models play in medicine and the biological sciences. Both sorts of models are metaphors. As such, they are only as good as their resemblance to the real system whose behavior they claim to model. Some animal models are very useful for understanding mechanisms of human physiology, because the animals are known to have close anatomical or physiological similarities to humans. Other animals make poor models (we wouldn't want to study human vision using the fruitfly, for example). The same holds true for computer models.

This is one reason why the question of a model's plausibility is so important. We need to worry not only about the neural plausibility of the model, but also it's psychological plausibility. We want our models to be learning behaviors which we think resemble those of humans. And we want the models to be given the same kind of information which we think is plausibly available to humans as well. This is not always an easy thing to know.

There is never any guarantee that the model is an accurate mirror of the human. The most we can say is that there is such a close correspondence between the behaviors, and between the kinds of constraints which are built into the model and which operate in humans, that we believe the model captures the essence of what is going on in the human.

The exciting thing is that the study of these artificial systems can also liberate us from biases and preconceptions which we might not even be aware of. We approach our study of behavior with a rich legacy of theories. Sometimes this legacy blinds us to ways of seeing phenomena which might lead to new analyses. Connectionist models often exhibit behaviors which are eerily like those of humans. We are surprised when we look inside the model to see what it is really doing, and discover that the model's solution to the problem may be very different from what we assumed was the obvious and only reasonable solution. Sometimes the model's solution is much simpler than our own theories; sometimes it is more complicated. But often, the solution is different, and can generate new hypotheses and insights.

Simulations as experiments

It's easy to do simulations, but hard to do them well. The ready availability of off-the-shelf neural network simulators makes it comparatively easy to train networks to do lots of things. Much of the initial flurry of excitement and interest in neural network learning was brought about by the discovery that networks could be successfully trained on a wide range of tasks. During the early period of connectionist research, exploring the many things that networks could do was intrinsically interesting.

But in the long haul, novelty wears off. We begin to find, too, that it is actually not all that easy to train networks on many tasks. And even when a network is trained successfully, we begin to ask ourselves, "So what? What have we learned or demonstrated that we did not know before?"

In fact, in our view, running a good simulation is very much like running any good experiment. We begin with a *problem* or goal that is clearly articulated. We have a well-defined *hypothesis*, a *design for testing* the hypothesis, and a plan for how to *evaluate the result*. And all of this is done *prior* to ever getting close to the computer.

The hypothesis typically arises from issues which are current in the literature. Simulations are not generated out of the blue. The nature of the hypothesis may vary. In some cases we may wish to test a prediction that has been made. Or we might wish to see whether an observed behavior might be generated by another mechanism (for example, as a way of testing the claim that only a certain mechanism is able to produce the behavior). Or we might have a theory of how some behavior is accomplished, and the simulation lets us test our model. Only rarely do we indulge in "fishing expeditions" (in which we start with no clearly defined goal); when we do, we must be prepared to come home empty-handed.

The hypothesis and goals of the simulation must be formulated very explicitly before we begin to think about the design. This is crucial, because when the simulation is done, we will need to know how to evaluate the results. What tests we run to evaluate the simulation will depend on what our initial questions were. Furthermore, our ability to run these tests will be limited by the design of the simulation. So the design itself is highly constrained by the hypothesis.

Such considerations are of course true of any experiment. We are simply emphasizing that these methodological constraints are equally true of simulations. However, there are some ways in which the correspondences between running simulations and running experiments with human subjects might not be as obviously similar. These largely have to do with design and evaluation issues.

The design of a simulation involves several things which may be unfamiliar to some. These include the notion of a *task*, *stimulus representations*, and *network architecture*. Ultimately, how to find the right task, representation, and architecture are best illustrated by example. That is one of the purposes of the simulation exercises which follow in the remaining chapters. But we think it is useful to begin with some explicit discussion of these issues to help focus your attention later as you work through the exercises.

The task

When we train a network, our intent is to get it to produce some behavior. The *task* is simply the behavior we are training the network to do. For example, we might teach a network to associate the present tense form of a verb with its past tense form; to decide whether or not a beam with two weights placed at opposite ends will balance; to maintain a running count of whether a digit sequence sums to an odd or even number; or to produce the phonological output corresponding to written text.

We need to be a bit more specific, however. What does it mean to teach a network to read text, for example?

In the networks we will be dealing with in this book, a task is defined very precisely as learning to produce the correct output for a given input. (There are other ways of defining tasks, but this is the definition that works best for the class of networks we will use.) This definition presupposes that there is then a set of input stimuli; and paired with each input is the correct output. This set of input/output pairs is often called the "training environment."

There are several important implications of this which we need to make explicit. First, we will have to be able to conceptualize behaviors in terms of inputs and outputs. Sometimes it will be easy to do this, but other times we may have to adopt a more abstract notion of what constitutes an input and what constitutes an output. We'll give a

simple example. Suppose we wish to teach a network to associate two forms of a verb. Neither form properly constitutes an input, in the sense that it is a stimulus which elicits a behavior. However, we might nonetheless train the network to take one form as input and produce the paired associate as output. This gives us the input/output relationship that is required for training, but conceptually we might interpret the task in more abstract terms as one of *associating* two forms.

Secondly, note that we are teaching the network a task by example, not by explicit rule. If we are successful in our training, the network will learn the underlying relationship between the input and output by induction. This is an appealing model for those situations where we believe that children (and others) learn by example. However, it is extremely important not to assume that the network has in fact learned the generalization which *we* assume underlies the behavior. The network may have found some other generalization which captures the data it has been exposed to equally well, but is an easier generalization to extract, given the limited data to which the network has been exposed. (What, precisely, is meant by "easier" is an important and fascinating question which is not fully understood.) This is where our testing and evaluation become important, because we need ways to probe and understand the content of the network's knowledge.

Finally, a concomitant of this style of inductive learning is that the nature of the training data becomes extremely important for learning. In general, networks are data grubs: They cannot get too much information! The more data we have, the better. And if we have too little information, we run the risk that the network will extract a spurious generalization. But success in learning is not merely a question of quantity of information. Quality counts too, and this is something which we will explore in several chapters. The structure of our training environment will have a great deal to do with outcome. And in some cases, it is even better to start with less, rather than more data.

A word about strategy is also appropriate here. Some tasks are better (more convincing, more effective, more informative) than others for demonstrating a point. It is easy to fall into the pitfall of "giving away the secret." This has to do with the role which is played by the output we are training the network to give us.

When we train the network to produce a behavior, we have a target output which the network learns to generate in response to a given input. This target output is called the "teacher," because it contains the information which the network is trying to learn. An important

question we should ask ourselves, when we think about teaching networks a task, is whether the information represented in the teacher is plausibly available to human learners. For example, it might be interesting to teach a network to judge sentences as grammatical or ungrammatical. But we must be cautious in interpreting the results, because it is questionable that the kind of information we have provided our network (the information about when to produce an output of "ungrammatical" and when to say "grammatical") is available to children. If we believe that this information is not available, then the lessons from the network must be interpreted with caution.

Or consider the case where we are interested in the process by which children learn to segment sounds into words. Let us say we are interested in modeling this in a network in order to gain some insight into the factors which make it possible, and the way in which the task might be solved. There are several ways to model this process. One task might be to expose a network to sequences of sounds (presenting them one at a time, in order, with no breaks between words), and training the network to produce a "yes" whenever a sequence makes a word. At other times, the network should say "not yet." Alternatively, we might train a network on a different task. The network could be given the same sequences of sounds as input, but its task would be simply to predict the next sound. (This example in fact comes from one of the later exercises in the book.)

It turns out that networks can learn both sorts of tasks without too much difficulty. In the second task, however, there is an interesting by-product. If we look at the network's mistakes as it tries to predict the next sound, we find that at the beginning of a word, the network makes many mistakes. As it hears more and more of the word, the prediction error declines rapidly. But at the end of the word, when the network tries to predict the beginning of the next sound, the prediction error increases abruptly. This is sensible, and tells us that the network has learned the fact that certain sequences (namely, sequences which make up words) cooccur reliably, and once it hears enough of a sequence it can predict the rest. The prediction error also ends up being a kind of graph of word boundaries.

In both tasks the network learns about words. In the first case, it does so explicitly, from information about where words start. In the second task, the network learns about words also, but implicitly. Segmentation emerges as an indirect consequence of the task. The disadvantage of the first task is that it really does give away the secret. That

is, if our goal is to understand *how* it is that segmentation of discrete words might be learned, assuming a continuous input in which segment boundaries are not marked, then we learn very little by teaching the task directly. The question is of interest precisely because the information is presumably *not* available to the child in a way which permits direct access to boundary information. Giving the information to the network thus defeats the purpose of the simulation.

Representing stimuli

We have said that a network is trained on inputs and outputs, but we have not been very specific about what those inputs and outputs look like. Network representations take the form of numbers; these may be binary (0's or 1's), integer valued (-4, 2, 34, etc.), or continuously valued (0.108, 0.489, etc.). Very frequently, information is represented by a collection of numbers considered together; these are called *vectors* and might be shown as [1 0 1 1 0]. In this case, we must have 5 input nodes; this vector tells us that the first, third, and fourth nodes are "on" (i.e., are set to 1) and the second and fifth are "off" (set to 0).

So the question is how we represent information of the sort we are interested in—words, actions, images, etc.—using numbers. This is in fact a very similar sort of problem which the nervous system solves for us. It converts sensory inputs into internal codes (neuronal firing rate, for example) which can be thought of as numerical. We might think of our problem as that of building artificial sensors and effectors for our networks.

The problem for us is that there are usually many different ways of representing inputs, and the way we choose to represent information interacts closely with the goals of the simulation. The basic issue is how much information we want to make explicitly available to the network.

At one extreme, we might wish to represent information quasi-veridically. We might present an image as a two-dimensional array of dots, in which black dots are represented as 1s and white dots are represented as 0s (or vice versa; interestingly, it doesn't matter as long as we are consistent in our mapping). This is like the half-tone format used in newspapers. If our inputs are speech, we might present the digitized voltages from an analog-digital converter, in which each

input is a number corresponding to the intensity of the sound pressure wave.

Or we might choose more abstract representations. We might encode a spoken word in terms of phonetic features such as *voiced*, or *consonantal*. Each sound would be represented as a bundle of such features (a bundle being a vector), and the word would be a sequence of feature vectors.

There are two bottom lines which guide us. First, the network can only learn if it is given sufficient information. If our encoding includes information which is irrelevant, the network may learn to ignore it and solve the task using the useful information; but if the input representation does not include information which is crucial for the task, the network cannot learn at all. Second, just as was true in the design of the task, we must be concerned not to give the answer away. One goal of a simulation might be to see if a network can learn certain internal representations (say, at the level of hidden unit activation patterns). In that case, it makes little sense to go ahead and give the network the target representations as part of its input.

Choosing the right architecture

The *architecture* of a network is defined primarily by the number and arrangement of nodes in a network: How many nodes there are, and how they are interconnected. Although in theory, any task can be solved by some neural network, not any neural network can solve any task. To a large degree, the form of the network determines the class of problems which can be solved.

In this handbook we utilize two major classes of architectures: feedforward networks and simple recurrent networks. In feedforward networks, the information flows from inputs to outputs; in recurrent networks, some nodes may also receive feedback from nodes further downstream. In both cases of networks, some nodes are designated as inputs and others are designated as outputs. How many of each depends on the task and the way in which inputs and outputs are represented. (For example, if a network will be taking images as inputs and these are represented as 100x100 dot arrays, then there must be 10,000 input nodes.) Additional hidden units may also be designated.

Determining the best architecture for a task is not easy and there are no automatic procedures for choosing architectures. To a large

degree, choice of architecture reflects the modeler's theory about what information processing is required for that task. As you work through the exercises, try to be aware of the differences in architecture. Ask yourself what the consequences might be if different architectures were chosen; you can even try simulations to test your predictions.

Analysis

Simulations generally involve two phases. In the first, we train a network on a task. In the second, we evaluate its performance and try to understand the basis for its performance. It is important that we anticipate the kinds of tests we plan to perform *before* the training phase. Otherwise we may find, once training is over, that we have not structured the experiment in a way which lets us ask important questions. Let us consider some of the ways in which network performance can be evaluated.

Global error. During training, the simulator calculates the discrepancy between the actual network output activations, and the target activations it is being taught to produce. Most simulators will report this error on-line, frequently summing or averaging it over a number of patterns. As learning progresses, the error will decline. If learning is perfect on all the training patterns, the error should go to zero.

But there are many cases where error cannot go to zero, even in the presence of learning. If a network is being trained on a task in which the same input may produce different outputs (that is, a task which is probabilistic or non-deterministic), then the best the network can do is learn the correct probabilities. However, there will still be some error.

Individual pattern error: Global error may also be misleading because if there are a large number of patterns (i.e., input/output pairs) to be learned, the global error (averaged over all of them) may be quite low even though some patterns are not learned correctly. If these are in fact the interesting patterns, then we want to know this. So it is important to look at performance in detail and be precise about whether the entire training set has been learned.

It may also be valuable to construct special test stimuli which have not been presented to the network during training. These stimuli are designed to ask specific questions. Does the network's perform-

ance generalize to novel cases? What in fact has the network learned? Since there are often multiple generalizations which can be extracted from a finite data set, probing with carefully constructed test stimuli can be a good way to find out what regularities have been extracted.

Analyzing weights and internal representations: Ultimately, the only way to really know what a network is doing, as well as understand how it is doing it, is to crack it open and look at its innards. This is an important advantage which network models have over human subjects; but like humans, the insides are not always easy to understand. Finding new methods to analyze networks is an area which is of great interest at the moment.

One technique which has been used profitably is hierarchical clustering of hidden unit activations. This technique can be useful in understanding the network's internal representation of patterns. After training, test patterns are presented to the network. These patterns produce activations on the hidden units, which are then recorded and tagged. These hidden unit patterns are vectors in a multi-dimensional space (each hidden unit is a dimension), and what clustering does is to reveal the similarity structure of that space. Inputs which are treated as similar by the network will usually produce internal representations which are similar—i.e., closer in Euclidean distance—to each other. Clustering measures the inter-pattern distance and represents it in tree format, with closer patterns joined lower on the tree. The partitioning of its internal state space is often used by networks to encode categories. The hierarchical nature of the category structure is captured by the subspace organization.

The limitation of hierarchical clustering is that it does not let us look at the space directly. What we might like is to be able to actually visualize the hidden unit activation patterns as vectors in hidden unit space in order to see the relationships. The problem is that it's not easy to visualize high-dimensional spaces. Methods like principal component analysis and projection pursuit can be used to identify interesting lower-dimensional "slices" in which interesting things happen. We can then move our viewing perspective around in this space, looking at activity in various different planes.

A third useful technique involves looking at activation in conjunction with the actual weights. When we look at activation patterns in a network, we are only looking at part of what a network "knows." The network manipulates and transforms information by means of the connections between nodes. Looking at the weights on these connections

is what tells us how the transformations are being carried out. One of the most popular methods for representing this information is by means of what are called "Hinton diagrams" (because they were first introduced by Geoff Hinton; Hinton, 1986), in which weights are shown as colored squares with the color and size of the square indicating the magnitude and sign of the connection. Techniques involving numerical analysis have also been proposed, including skeletonization (Mozer & Smolensky 1989) and contribution analysis (Sanger, 1989).

What do we learn from a simulation?

Much of the above may still seem abstract and unclear. That's ok. Our goal here is simply to raise some questions now so that when you do the simulation exercises in the remaining chapters, you will be conscious of the design issues we have discussed here. Ask yourselves (and be critical) whether or not the simulations are framed in a way which clearly addresses some issue, whether or not the task and the stimuli are appropriate for the points that are being made, and whether you feel at the end that you have learned something from the simulation.

That's the bottom line: What have we learned from a simulation? In general there are two kinds of lessons we might learn from a simulation. One is when we have a hypothesis about how some task is carried out, and our simulation shows us that our hypothesis can in fact provide a reasonable account of the data. The second thing we may learn is new ways in which behaviors may be generated. Having trained an artificial system to reproduce some behavior of interest, our analysis of the model's solution may generate new ideas about how and why behaviors occur in humans. Of course, the model's solution need not be the human's; verifying that it is requires additional empirical work. But if nothing else, the opportunity to discover new solutions to old problems is both valuable and exhilarating!