Computational Models of Neural Systems

February 3, 2009

1 Introduction

The basic mechanisms of neural activity lend themselves to computational modeling at a great many levels of detail. These models can help neuroscientists understand the brain by providing virtual test environments for their theories, sometimes saving millions of dollars of wet-lab research. Still, computer programs are only simplifying abstractions of reality, and in *untamed frontiers* like brain science, it is not always clear which details can safely be ignored.

Many standard forms of artificial neural networks (ANNs) ignore significantly more details than a neuroscientist would recommend. In fact, some ANN designers completely disregard biology in their pursuit of efficient engineering solutions. They seek to exploit the basic neural paradigm of numerous simple processors tightly intertwined in a complex network, but beyond that, the details may derive more inspiration from statistical mechanics, for example, than neuroscience. Indeed, the universe of neural-inspired models is extremely vast.

In this book, we begin our modeling pursuits at a relatively high (but still biologically consistent) level. We gradually add more biological detail, most at the topological rather than the individual-neuron level, in order to achieve progressively more complex behaviors that reflect neuropsychological phenomena such as pattern completion, memory, attention, and action selection, as well as common learning forms such as the supervised, unsupervised and reinforced varieties.

Along the way, we investigate some of the classic engineering-based ANNs and show their similarities and differences to what is known about the neurobiological realities.

2 The Abstraction's of ANN

We begin by cutting away 99.9% of the fascinating details of neuroscience, as best detailed by Kandel [6], but retaining those which seem essential for achieving the advantages of neural computation, namely, neurons that integrate inputs from several sources before determining their own output, and modifiable synapses.

The details of ion flow into and out of neural processes is amenable to standard electrophysical models, and researchers interested in the intricacies of AP transmission along axons and dendrites employ them frequently, as best summarized by Dayan and Abbott [1]. However, most ANN designers can safely ignore this level of detail and simply assume that signals sent by an upstream neuron's soma will reach all downstream

neighboring soma, albeit in different states of attenuation, or possibly amplification. When the upstream neuron is an inhibitor, a *negative signal* is transmitted.

The only preserved component along the path from one soma to the next is the synapse, which we, like most ANN designers, abstract into a simple real number, known as the *weight* of the connection between the two soma. We also drop the distinctions between soma, axons and dendrites and simply view the neural network as a matrix of nodes, where any two nodes that are connected have a single weight that modifies the signal sent from the upstream to the downstream node. It is common to retain the unidirectionality of these links/arcs/connections, so in those cases where two nodes are bidirectionally connected, the weights in each direction need not be equivalent.

Figure 1 summarizes the standard abstraction of natural neural networks to ANNs, which only preserve the concepts of nodes and connections with heterogeneous and modifiable weights. The bottom of the figure shows the three key processes that are standard components of ANNs:

1. an integration function, which normally sums the weighted outputs from all upstream neighbors, with weights coming from the connections.
2. an activation function, which transforms the integrated sum into an activation level for the neuron, which then uses it as an output value, to be weighted and summed by downstream neighbors.
3. a learning function, which uses the activation values of connected nodes (plus, possibly, global signals akin to neuromodulators) to compute weight changes to the arc(s) that connect them.

2.1 Integration

Integrating the inputs from all upstream neighbors is normally performed via the following simple equation:

$$net_i = \sum_{j=1}^{n} o_j w_{i,j} \tag{1}$$

Here, o_j is the output (or, equivalently, the activation level) of neuron j, while $w_{i,j}$ is the weight on the arc from neuron j to neuron i. Be aware that some authors use the opposite notation: $w_{i,j}$ is the weight on the arc from neuron i to neuron j.

2.2 Activation Functions

A wide variety of activation functions are used for ANNs, with the logistic equation (below) being one of the more popular:

$$o_i = \frac{1}{1 + e^{-net_i}} \tag{2}$$

Figure 2 plots the logistic function (a sigmoidal curve) along with several others. All except the identity function exhibit a thresholding effect, which nicely mirrors the behavior of real neurons. The logistic function

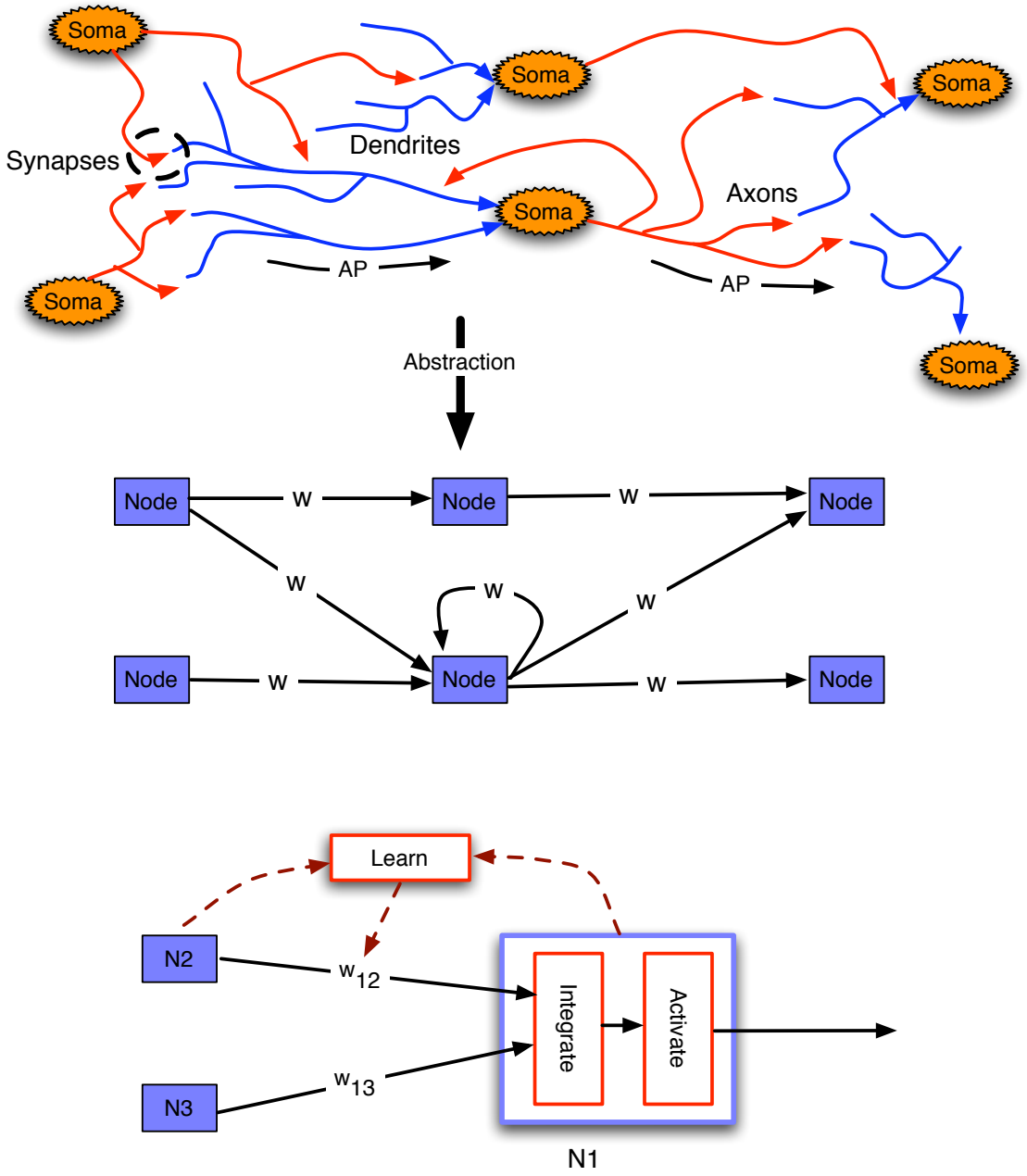

Figure 1: (Top) Summary of the abstraction from a detailed neural network to the standard model used in the majority of ANN research. Here, w denotes a weight on the link. (Bottom) The three key biological processes preserved in most ANNs: a) integration of the incoming signals to a node, b) conversion of the integrated signal into an output *activation* level, and c) use of activation levels in connected nodes to modify the weight(s) on the arc(s) between them, i.e. learning.

has the advantage of being continuous and thus differentiable at all points in its domain. Differentiability is a critical property for ANNs used for backpropagation learning, which is discussed later.

The hyperbolic tangent gives a similar continuous sigmoidal curve, but it ranges from -1 to 1, which is a useful property for networks in which some neurons need to have powerful influences whose sign can quickly toggle, depending upon net_i: if it's low, the node becomes an inhibitor of those downstream neighbors linked to with positive weights, but if it goes high, the effect toggles to stimulatory. With the logistic function, low values of net_i give a near-zero activation level and curtailed downstream influence (of either type).

The equation for the hyperbolic tangent is:

$$\frac{e^{2net_i} - 1}{e^{2net_i} + 1} \tag{3}$$

2.3 Learning in Neural Networks

Learning is the core aspect of neural networks. It is also the most complicated. A wide range of diverse learning algorithms exists for ANNs, some staying true to neuroscience, and some straying far afield in order to adequately solve real-world problems. This section can only scratch the surface of the contemporary ANN learning approaches.

In the machine learning [8] , neural network [4], and neuroscience [2, 1] literature, authors often distinguish three types of learning: unsupervised, reinforced, and supervised.

In unsupervised learning, the agent learns recurring patterns without any tutoring input. Essentially, the neural system detects correlations between neuronal firing patterns and between those patterns and the structure of inputs to the network. These correlations are strengthened by changes to ANN weights such that, in the future, portions of a pattern suffice to predict/retrieve much of the remainder.

In supervised learning, the agent receives very frequent feedback that not only signals good/bad, but also indicates the action that **should** have been taken in cases where the agent makes a mistake. This is the classic form of learning handled by neural networks in many practical applications, with gradient-descent methods, such as the classic backpropagation algorithm, used to modify weights so as to reduce error.

In reinforcement learning, the agent receives an occasional reward or punishment that essentially indicates that the net result of many activities was good or bad. The trick, known formally as the *credit-assignment problem*, is to figure out, from this general global signal of *good* or *bad*, how to modify the individual weights so as to improve performance in the future.

The three learning paradigms are summarized in Figure 3 with an example of a maze-following agent and a sketch of typical ANN controllers.

Clearly, the supervised approach, when feasible, can yield much faster learning of useful situation-action pairs than the other two methods. Unfortunately, omniscent tutors are not available for the brunt of biological learning, and their assistance, when available, normally comes closer to reinforced than supervised. In non-human animals, the level of detailed supervision is even less. Yet, all animals are capable of learning useful behavioral information in a relatively short period. So if there is any element of supervision in this process, where is the feedback coming from? Prediction provides an answer.

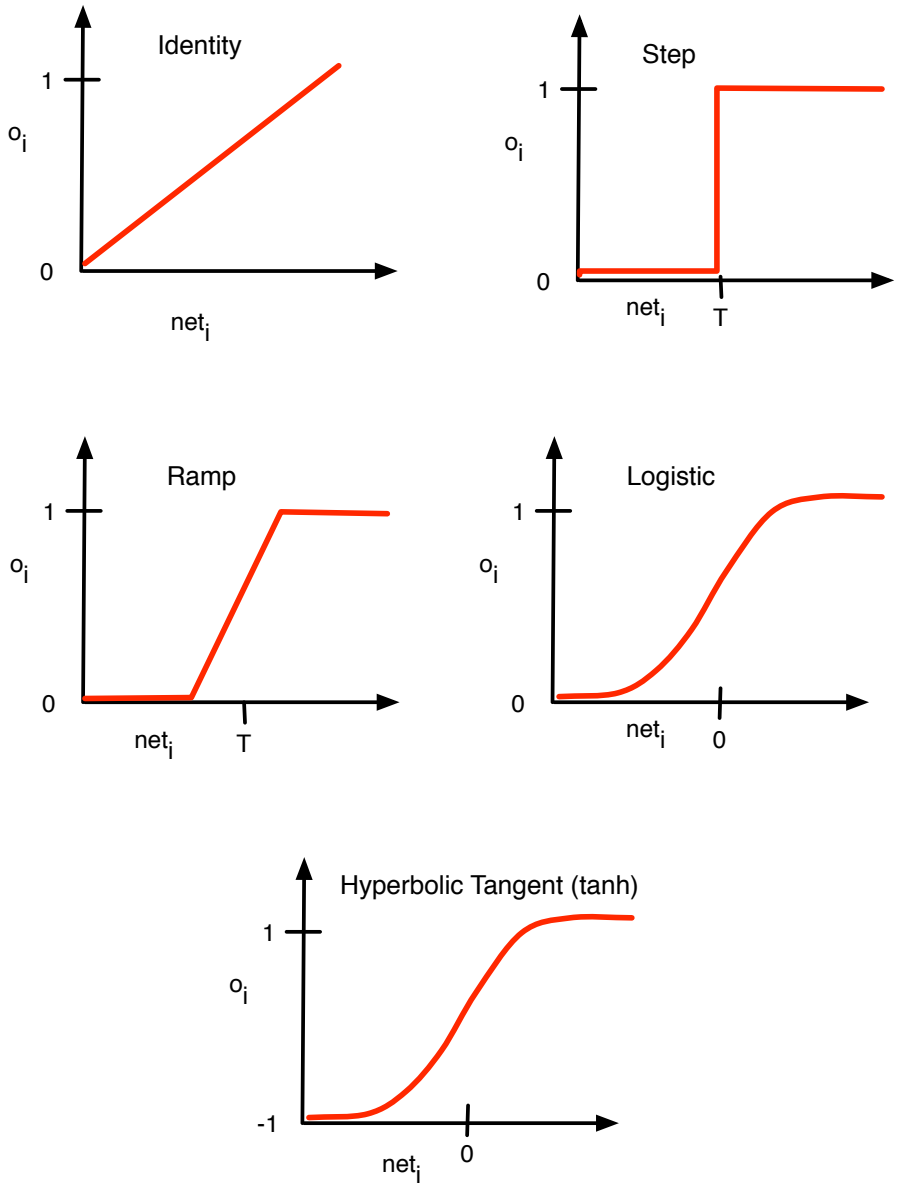

Figure 2: A variety of activation functions commonly used for artificial neural networks. Only the logistic and hyperbolic-tangent equations have the desirable combination of thresholding and continuity.

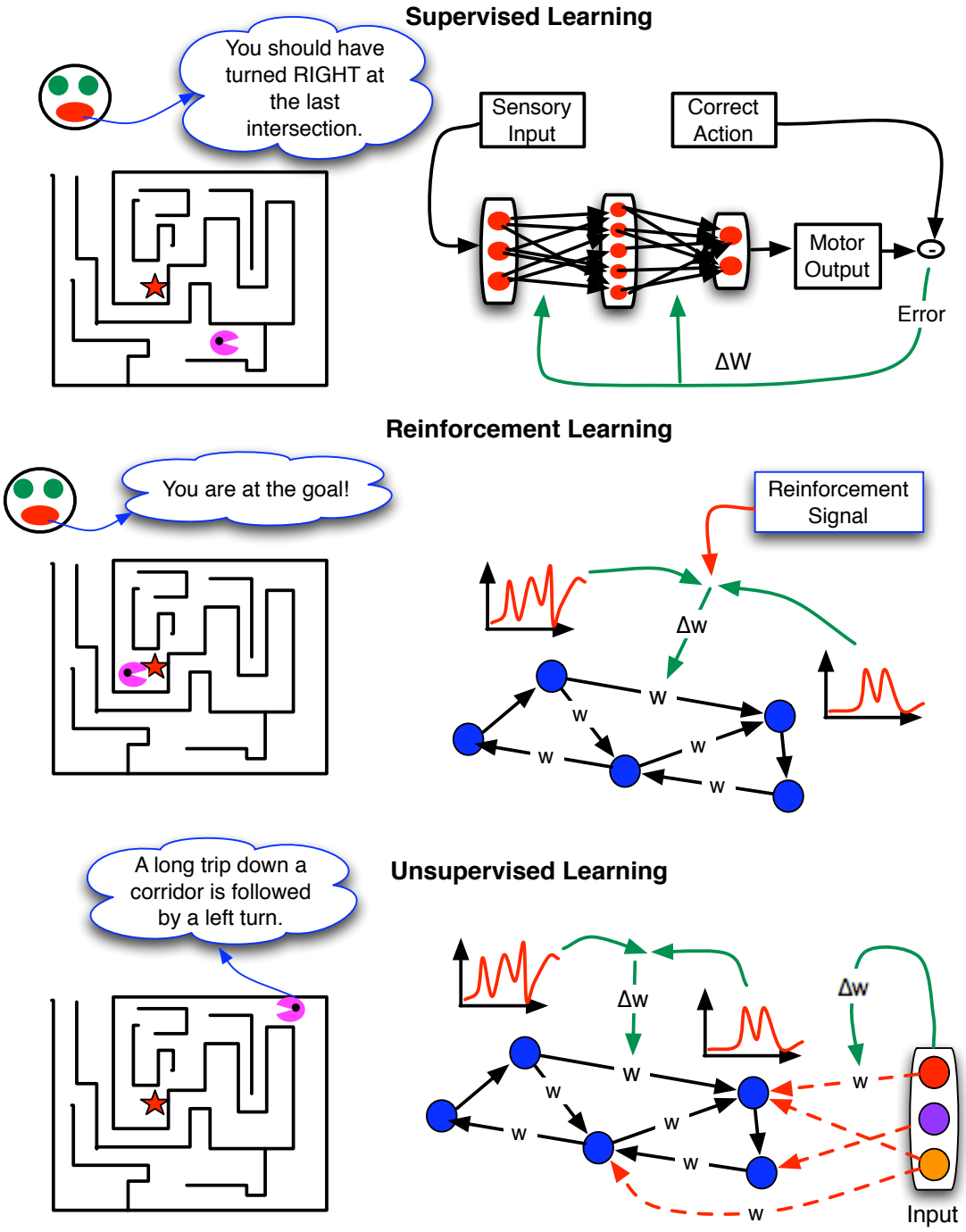

Figure 3: Sketch of the 3 basic neural-network learning paradigms: supervised, reinforcement and unsupervised. On the left is a maze-following task and a rough idea of the feedback that the network would receive from a tutor (small face). In unsupervised learning, there is no feedback, so the network learns associations based solely on correlations among neural firing patterns and between those patterns and sensory inputs and/or motor outputs.

Essentially, an agent can be its own teacher when doing predictive tasks. At time t, it predicts the future state of its body and immediate surroundings at t+1. Then, at time t+1, it learns the correct answer and can use that knowledge to adjust its own mapping from current to future states. This can be done almost continuously as the agent moves about the world, as researchers in evolutionary robotics have long recognized and exploited [9].

In natural neural systems, Hebb's rule (i.e. neurons that fire together, wire together) holds true in many cases: neurons that are active within the same time window (of about 100 ms) often initiate chemical mechanisms that strengthen the synapse between them. However, this rule has many exceptions, as discussed below.

Figure 4 depicts the general learning scenario, wherein a post-synaptic neuron (v) receives input from many pre-synaptic neurons ($u_i \forall i = 1..n$). The temporal firing pattern of v is compared to that of each u_i to determine the change in weight w_i.

Learning in artificial neural networks (ANNs) often abides by similar Hebbian principles in that the weights between co-active neighbor neurons are often modified. The quantitative variables used for this comparison are simply the output values of pre-synaptic and post-synaptic neurons. In the following, we will denote these values as u_i and v, respectively.

Depending upon the temporal granularity (and other perspectives) with which one views natural neural systems, u_i and v can be seen as representing many physical variables, including:

1. the instantaneous membrane potential of a neuron,
2. the firing rate of a neuron, typically measured in AP's per second,
3. the average firing rate of a neuron over a particular time window, or
4. the difference between a neuron's current firing rate and its average firing rate.

The first and fourth alternatives allow for negative values, while the second and third are strictly non-negative.

In the discussion that follows, u_i and v will typically represent either the last activation level or the difference between a) the firing count of the neuron during the past m time steps, and b) the average firing count (per m time steps) computed over M such m-step periods. The term *neural output* will be used as a general reference to u_i or v, without any assumption about the exact nature of the physical variable.

Also, the term long-term potentiation (LTP) refers to a strengthening of a weight, while long-term depression (LTD) denotes a weakening. The *long-term* aspect of each change stems from the biological uses of LTP and LTD, where this type of synaptic change persists for hours, days or longer. In ANNs, there is no such guarantee of the duration of the change.

The classic Hebbian learning rule is simply:

$$\triangle w_i = \lambda u_i v \tag{4}$$

where λ is a positive real number (often less than 1) representing the learning rate. Equation 4 captures the basic proportionality between weight change and the correlation between the two neural outputs.

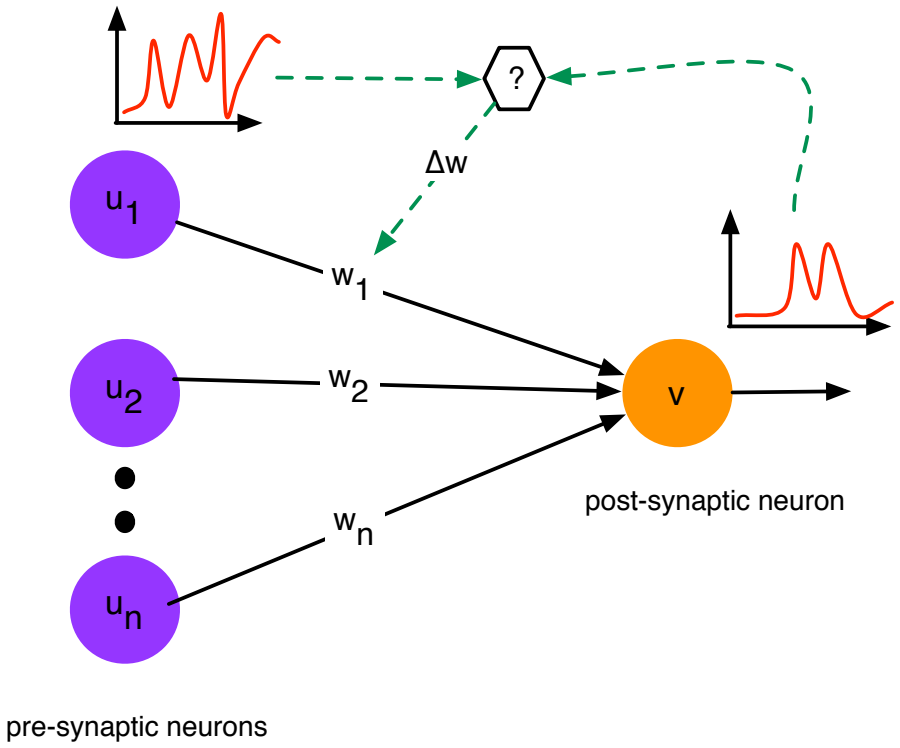

Figure 4: The essence of learning in neural networks: the comparison of pre-synaptic and post-synaptic firing histories determines changes to the connection weight between the two neurons. Here, weights have a single subscript, denoting the pre-synaptic neuron. Graphs are of the neuron's membrane potential as a function of time.

When u_i and v represent recent or time-averaged firing-rates, as they often do, they are never negative. Thus, the right-hand side of equation 4 is always positive, and weights increase without bound. In fact, a positive feedback occurs, since positive firing rates produce weight increases, which insure that the influence of u_i upon v gets stronger, which tends to raise v's firing rate, which then elevates the weight, etc.

One can simply put an upper bound on weight values, but then all weights eventually reach this limit and the ANN has no diversity and hence no interesting information content.

Another attempt to relieve the problem is to view u_i and v as either a) membrane potentials, or b) the differences between current and average firing rates. Both of these interpretations allow u_i and v to be positive or negative. Thus, $u_i v$ will often be negative, and weights should decrease as well as increase.

The latter interpretation has spawned several useful learning rules in which presynaptic, postsynaptic or both firing rates are relativized to a *normal* level. For example, the weight-updating expression, known as a *homosynaptic* rule normalizes only the postsynaptic output to a threshold, θ_v:

$$\triangle w_i = \lambda(v - \theta_v)u_i \tag{5}$$

Notice that this rule can exhibit long-term potentiation LTP or LTD depending upon whether v is above or below θ_v, which typically represents an average firing rate (computed and updated over the course of a simulation. The term *homosynaptic* refers to the fact that if v is above (below) threshold, then **all** presynaptic neurons that have a non-zero firing rate will experience LTP (LTD) along their connection to v. That is, all presynaptic neurons that fire will see the same type of change.

Conversely, the following rule is *heterosynaptic*, since, when $v > 0$, only those presynaptic neurons that fire above threshold will experience LTP, while the others - even those that do not fire at all - will experience LTD:

$$\triangle w_i = \lambda v(u_i - \theta_i) \tag{6}$$

A popular learning scheme that only normalizes the postsynaptic neuron but which requires both pre- and postsynaptic firing is the BCM (Bienenstock, Cooper and Munro) rule:

$$\triangle w_i = \lambda u_i v(v - \theta_v) \tag{7}$$

Rules that normalize both pre- and postsynaptic neurons include:

$$\triangle w_i = \lambda(v - \theta_v)(u_i - \theta_i) \tag{8}$$

and

$$\triangle w_i = \lambda v(v - \theta_v)u_i(u_i - \theta_i) \tag{9}$$

2.3.1 Instability of Hebbian Learning

Returning to the pure Hebbian learning rule of equation 4, as described above, when u_i and v are non-negative numbers, the weights will increase without bound. One seemingly obvious fix is to use activation functions whose outputs can be negative, for example, the hyperbolic tangent function shown at the bottom of Figure 2. Unfortunately, this is not enough to avoid positive feedback and the ensuing unstable increase in weights. The proof, as described in [1] (pp. 286-287) is straightforward and summarized below.

To assess the change of an entire weight vector, in this case, the vector of all input weights to node v, W_v, we can look at the geometric length of that vector, which is simply:

$$\mid W_v \mid = \sqrt{\sum_{i=1}^{n} w_i^2} \tag{10}$$

We are interested in whether that length continually increases, decreases or fluctuates. In other words, we are interested in $\frac{d|W_v|}{dt}$. To simply our analysis, we can look at $\frac{d|W_v|^2}{dt}$, which will have the same sign as $\frac{d|W_v|}{dt}$. Then:

$$\frac{d \mid W_v \mid^2}{dt} = \frac{d(\sum_{i=1}^{n} w_i^2)}{dt} = \frac{dw_1^2}{dt} + \cdots + \frac{dw_n^2}{dt} = 2w_1 \frac{dw_1}{dt} + \cdots + 2w_n \frac{dw_n}{dt} = 2W_v \bullet \frac{dW_v}{dt} \tag{11}$$

So the key relationship is:

$$\frac{d \mid W_v \mid^2}{dt} = 2W_v \bullet \frac{dW_v}{dt} \tag{12}$$

Now, if we rewrite equation 4 to account for all input arcs to v, we get:

$$\frac{dW_v}{dt} = \lambda U v \tag{13}$$

where U is the vector composed of $\{u_i : i = 1..n\}$.

Combining equations 12 and 13, we get:

$$\frac{d \mid W_v \mid^2}{dt} = 2W_v \bullet \lambda U v \tag{14}$$

If we assume:

$$v = W_v \bullet U \tag{15}$$

then:

$$\frac{d \mid W_v \mid^2}{dt} = 2\lambda v W_v \bullet U = 2\lambda v^2 \tag{16}$$

Thus, except for the trivial case when v = 0:

$$\frac{d \mid W_v \mid^2}{dt} > 0 \tag{17}$$

Thus, the length of the weight vector is always increasing, which generally means that the positive weights keep getting larger and the negative weights keep getting smaller (i.e. becoming larger negative numbers). This instability of the weight vector typically leads to a saturation of the network, with many nodes always firing hard and others never firing at all.

It is important to remember that the final step of equation 16 hinges on the assumption in equation 15, which entails that v is the sum of its weighted inputs, net_v. This is equivalent to assuming the linear transfer function of Figure 2, which, of course, is not always the case.

However, $\frac{d|W_v|^2}{dt}$ remains non-negative as long as v and $W_v \bullet U = net_v$ never have the opposite sign. This is the case for transfer functions with thresholds at $net_v = 0$ and which produce positive outputs above that threshold and non-positive outputs below it. Sigmoidal transfer functions used in ANNs often have this property, although it may be violated in a small neighborhood below the threshold, where small (i.e. near zero) negative net_v values yield small positive outputs.

At any rate, the general argument should be clear: the use of both negative and positive values for u_i and v does not, on its own, prevent the positive feedbacks that lead to instability (i.e., infinite growth in the positive or negative direction) of the ANNs weights.

Furthermore, if $W_v \bullet U$ and v have the same sign for long periods of time, $\triangle W_v$ can easily become dominated by the positive-feedback dynamics. As net_v moves further toward the extremes of the activation function (i.e., *saturates*), this relationship is more likely. Hence, it is wise to design networks that can avoid saturation, but, instead, can allow neurons to work near the thresholds of their activation functions, where the signs of net_v and v often vary.

Similar problems arise with learning rules such as BCM, which is unstable when θ_v is held constant [1] (pg. 288) but becomes stable when θ_v is permitted to grow faster than does the average value of v, as described in [1] (pp. 288-9).

2.4 Weight normalization

To avoid problems of weight-vector instability that occur with so many of the popular learning rules, more direct methods of weight restriction are available.

One of the most direct mechanisms is to simply normalize the weights associated with a particular neuron. Here, the options include:

1. **which** weights to normalize, for instance weights on all incoming (or outgoing) links to (from) each neuron.
2. **how** to normalize, whether by:
   (a) dividing each weight by the sum of the weights or the sum of the absolute values of the weights (in cases where weights can be positive or negative), or

(b) subtracting the same quantity from each weight.

Another common approach is to randomly initialize the weights and then, for each node, store one of two sums: those of its incoming or outgoing weights to(from) that node. Assume that you choose the input weights, whose initial sum for node i is σ_i. Then, after each round of learning-based weight change, use equation 18 to renormalize each weight such that the input sums for each node equal the originals.

$$w_{ij} \leftarrow \frac{\sigma_i w_{ij}}{\sum_{j=1}^{n} \mid w_{ij} \mid} \tag{18}$$

Although computationally expensive, these methods do have the desired effect of preventing runaway weights.

A simpler and computationally cheaper method is the Oja rule, which includes a *forgetting* term that involves the weight itself:

$$\triangle w_i = \lambda v(u_i - v \mid w_i \mid) = u_i v - v^2 \mid w_i \mid \tag{19}$$

Notice that the forgetting (rightmost) term increases as the postsynaptic firing rate increases. This combats a standard positive-feedback problem with the earlier rules: when neurons fire at high rates, synapses tend to strengthen, which helps neurons to fire even harder in the future.

The Oja rule has nice theoretical properties, including the ability to perform principle component analysis (PCA), but it lacks biological plausibility. However, it's stability is easily proven [1] (pg. 289), and thus it is a popular learning mechanism for ANNs that are not designed to accurately mimic the brain, but rather, to solve complex engineering problems.

3 ANNs for Unsupervised Learning

Unsupervised Learning appears to be very common in the brain, particularly the neocortex, as discussed later. However, basic Hebbian processes occur in many parts of the brain, and in most cases, one sees a rich set of neuronal interactions, some of which have a **cooperative** flavor, with some neurons promoting others, while others have a distinctly *competitive* nature, with neurons actively inhibiting other neurons when they themselves fire. Both processes are clearly important for the neural basis of intelligence. This section summarizes a few of the standard ANN architectures for unsupervised learning that embody cooperation and/or competition.

3.1 Hopfield Networks

The basic Hebbian notion of firing together and wiring together has a very cooperative connotation: neurons working in concert will tend to promote on another's activity. Thus, networks whose main learning algorithm is the purely Hebbian formula of equation 4 have a strong cooperative feel.

Hopfield networks are perhaps the simplest ANN type that incorporates all three of the basic components (integration, activation and learning). Despite their simplicity, Hopfield network capture two of the brain's

key functions: storage and retrieval of distributed patterns. They do this in a completely **unsupervised** manner by recognizing correlations among neuron firing histories and modifying weights to record those relationships.

In the brain, many forms of information appear to be distributed across large populations of neurons. This *population coding* has several advantages:

1. Storage efficiency - in theory, k neurons with m differentiable states can store m^k patterns.
2. Robustness - if a few neurons die, each pattern may be slightly corrupted, but none is lost completely.
3. Pattern completion - given part of a pattern, the network can often *fill in* the rest. This allows memory to be *content addressable*, that is, patterns are retrieved not by specifying their location in memory but by specifying a portion of the pattern.

Hopfield nets take advantage of population coding to store many patterns across a shared collection of nodes, with each node representing the same aspect of all stored patterns and each connection weight denoting the average correlation between two aspects across all stored patterns.

Figure 5 displays an auto-associative Hopfield network, where *auto* implies that the correlation is between aspects/components of the **same** pattern. In this case, the components are simply small regions of the image, with components a and b representing small regions centered at distinct locations of the image plane. An auto-associative network encodes the average correlations (across all stored patterns) between all pairs of components.

In Figure 5, the correlation between components a and b is computed for each of the two images. It is positive in the left image, since a and b both contain some black color, but it is negative in the right image, since a contains black but b does not. Thus, the left image contributes a +1 to the average correlation between a and b, while the right image contributes a -1. If there are P patterns to store, then P such correlations will be averaged for every pair of components. This correlation analysis of all P patterns constitutes the *training phase* of the Hopfield algorithm. It is where the learning occurs.

Note that training reflects Hebbian learning at a coarse level: when two components are highly correlated in a pattern, then the link between their nodes in the ANN will be strengthened. Alternatively, a negative correlation (i.e. one component is on/black while the other is off/white) leads to a weight reduction.

The Hopfield network (a small portion of which appears at the bottom of Figure 5) has a *clique* topology, meaning that every node is connected to every other node. Each node represents a component, and each arc weight denotes the average correlation between the components of the arc. In Hopfield networks, the arcs are bidirectional, since a correlation is a symmetric property.

Once trained, the auto-associative Hopfield net can be employed to retrieve one of the P original patterns when given only a portion (or corrupted version) of it, as shown at the bottom of Figure 5.

The retrieval process begins by *loading* the partial pattern into the network: components that are on/black in the partial pattern will have their corresponding ANN nodes set to a high activation level, while those that are off will engender low activation levels. The network is then run, with nodes summing their inputs and using their activation functions to compute new output levels. This continues until either the network reaches a quiescent state (i.e., no nodes are changing activation levels) or a fixed number of update steps have been performed. The final activation levels of the nodes are then mapped back to the image plane to create the output pattern. For example, if node f has a high activation level, then the f component of the image will be painted black.

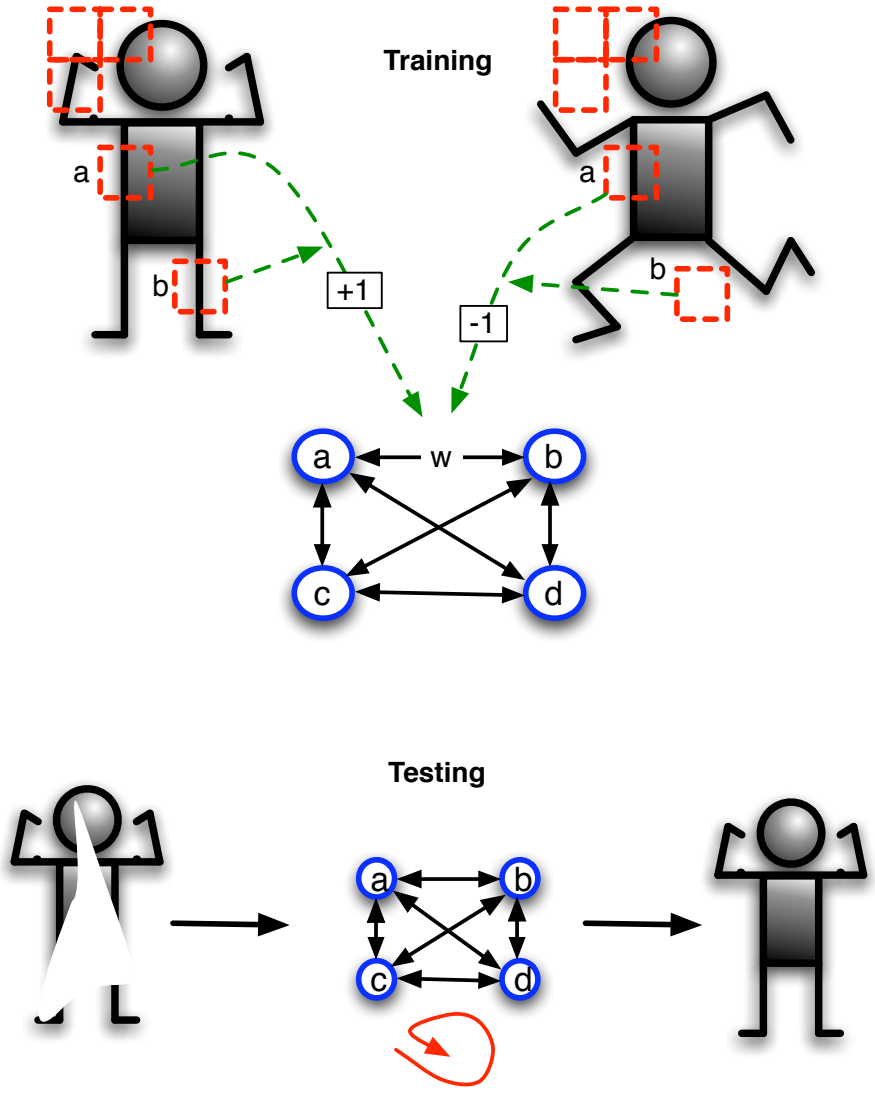

Figure 5: Overview of the training (learning) and testing (pattern retrieval) procedures for auto-associative Hopfield networks.

Similar to a Hopfield net, a hetero-associative net records average correlations between **pairs** of patterns. These are useful for associating one pattern with its typical successor pattern in a sequence. Hence, when given the predecessor pattern, the network can **predict** the successor.

Figure 6 shows the essence of a hetero-associative network, where correlations between component a in the two figures are computed, as are those between the b regions. In the complete algorithm, all component pairs between the two patterns are examined: (a,a), (a, b), (a, c), etc. Note that the graph has a bipartite topology, meaning that it is split into two halves, with each node connected to all nodes on the other half but none on its own side.

Once trained, the hetero-associative Hopfield network can return a successor pattern when given a partial or corrupted version of its predecessor pattern, as shown at the bottom of Figure 6.

Here, the testing phase begins by loading the partial predecessor pattern onto the input (left) half of the network. Next, the activation levels on the output (right) side are computed based on the integrated inputs from the left side. Then, the left-side activation levels are recomputed, based on both a) the reloaded components of the original input pattern, and b) inputs from the right side of the net.

The process of recomputing activations for the left and right sides continues until quiescence (or a fixed number of steps). The values of the output nodes are then translated into the output image.

3.2 Basic Computations for Hopfield Networks

In Hopfield and many other associative networks, the learning phase is a one-shot, batch process in which all patterns are analyzed and all correlations averaged. The weights of the network are then set to those averages and never modified.

A typical learning (i.e. weight-assignment) scheme for Hopfield networks is:

$$w_{jk} \leftarrow \frac{1}{P} \sum_{p=1}^{P} c_{pk} c_{pj} \tag{20}$$

where P is the number of patterns, c_{pk} is the value of component k in pattern p, and $c_{pk}c_{pj}$ is the local correlation in pattern p between components k and j.

For a hetero-associative network, the corresponding scheme is:

$$w_{jk} \leftarrow \frac{1}{P} \sum_{p=1}^{P} i_{pk} o_{pj} \tag{21}$$

where P is now the number of pattern **pairs**, i_{pk} is the kth component of the predecessor (input) pattern of pair p, and o_{pj} is the jth component of the successor (output) pattern of pair p. The product of the two components is the local (k,j) correlation for pair p.

Once all weights have been computed, the net can be run by loading input patterns and updating activation levels. For discrete Hopfield networks, where firing levels are either +1 or -1, the following activation function is common:

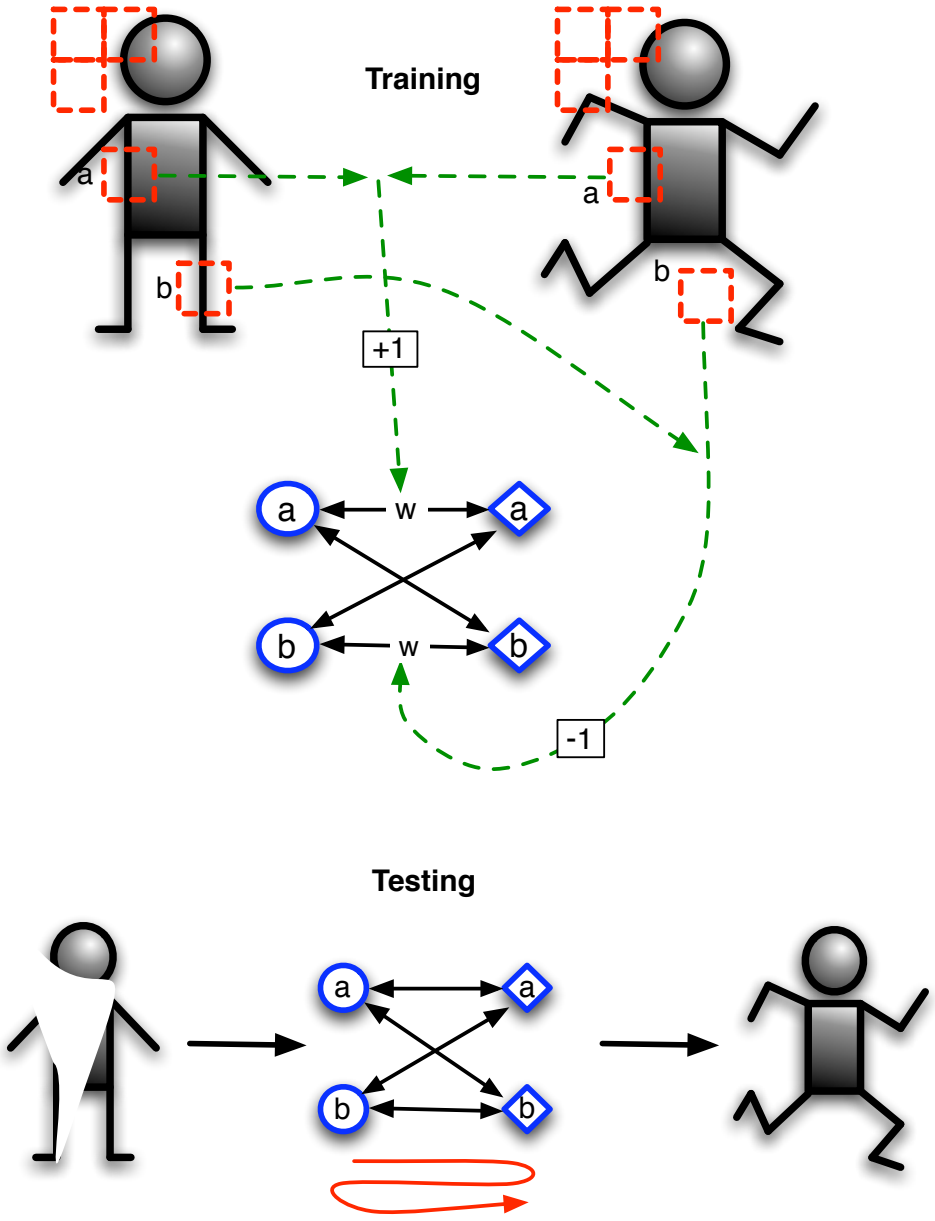

Figure 6: Overview of the training (learning) and testing (pattern retrieval) procedures for hetero-associative networks.

$$c_k(t+1) \leftarrow sign(\sum_{j=1}^{C} w_{kj} c_j(t) + I_k) \tag{22}$$

where C is the number of components (for example, 64 in an 8-by-8 image plane), $c_k(t+1)$ is the activation level of the kth component's neuron at time t+1, w_{kj} is the weight on the arc from node j to node k, and I_k is the original input value for component k. The I_k term insures that the original bias imposed by the input pattern has an effect throughout the entire run of the network.

Note that the summation in equation 22 is the standard integration function for computing net_k, as given in equation 1.

Figure 7 illustrates the basic training and test procedure for Hopfield networks.

3.3 Search in Hopfield Networks

The process by which a running associative network transitions to quiescence has a search-like quality: the network seeks a stable state. In this case, a state is a vector consisting of the activation levels of each neuron in the ANN.

Unfortunately, stable states do not necessarily correspond to any of the original P patterns. They may be *spurious*, i.e., they map to patterns that were not part of the training set.

Figure 8 illustrates this problem. The original input pattern (top) is loaded into the auto-associative network, but nothing guarantees that the quiescent state will map to the correct pattern (bottom middle) or one of several spurious ones (bottom left and bottom right).

Hopfield [5] quantified the search for quiescence as a search for minima on an energy landscape by defining a function for mapping ANN states to energy levels. The general Hopfield learning procedure (equation **??**) and activation function (equation 22) then insure that low-energy states are those corresponding to patterns on which the net was trained. However, nothing prohibits some spurious patterns from achieving locally-minimal energy as well. Here, a local minimum is defined as a state, M, such that any state M* that is created by updating all activation levels of M **once**, has higher energy than M.

Hopfield's energy function is:

$$E = -a \sum_{k=1}^{C} \sum_{j=1}^{C} w_{jk} c_j c_k - b \sum_{k=1}^{C} I_k c_k \tag{23}$$

where c_k is the activation level of component k's neuron, I_k is the original input value loaded into the kth component's neuron, and a and b are positive constants.

Albeit complex in appearance, equation 23 is actually quite intuitive. Consider the term:

$$w_{jk} c_j c_k \tag{24}$$

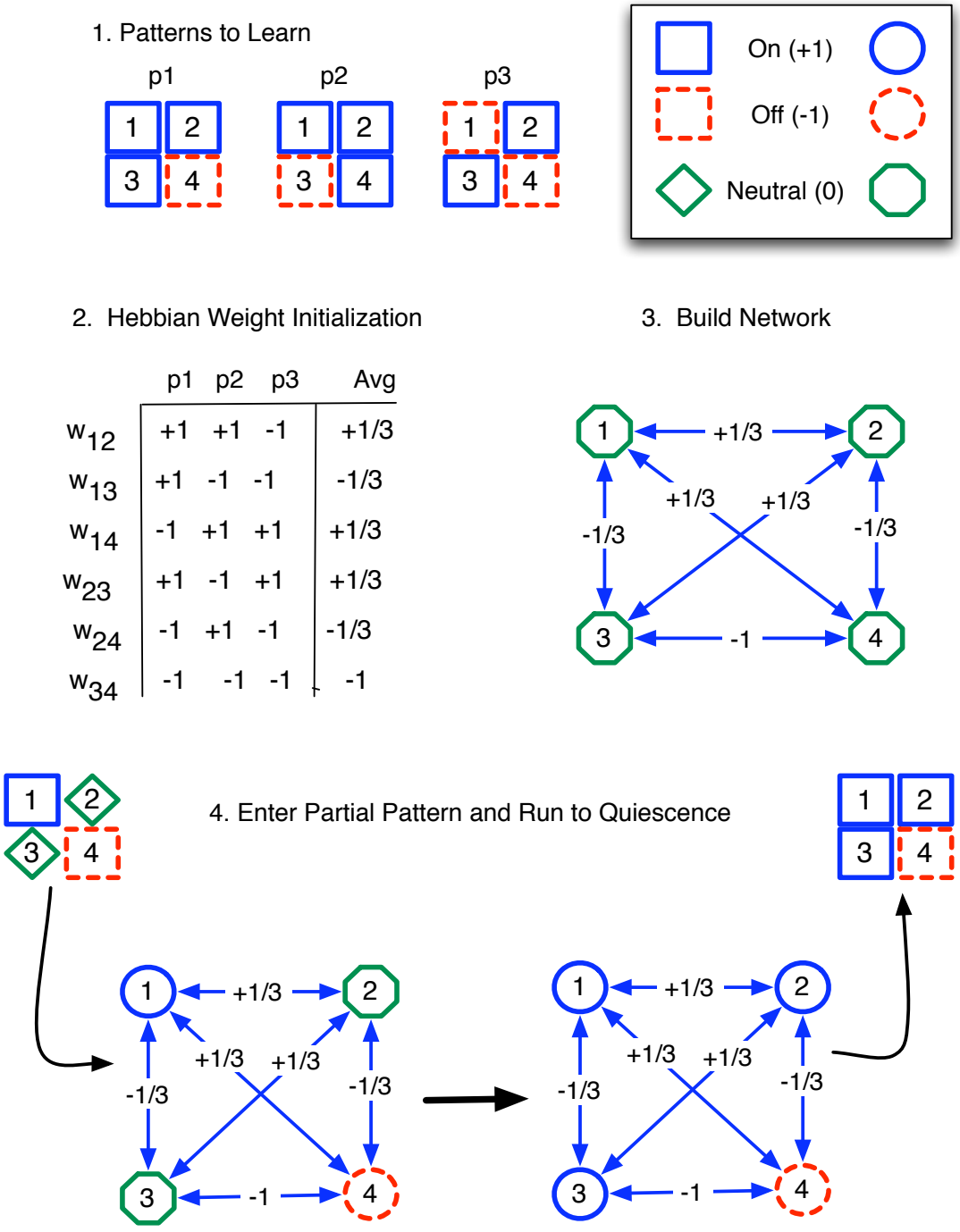

Figure 7: The basic procedure for encoding patterns in a Hopfield network and then retrieving them via partial patterns. Steps 1-3 involve computing the average correlations between all pairs of components (i.e. pixels) in the input patterns and then using them as weights in the network. Step 4 shows the encoding of a simple partial pattern. All nodes then update their activation levels **simultaneously**, i.e. synchronously, to attain the next state, which turns out to be stable: further updates will not change any activation levels. The final state corresponds to pattern p1.

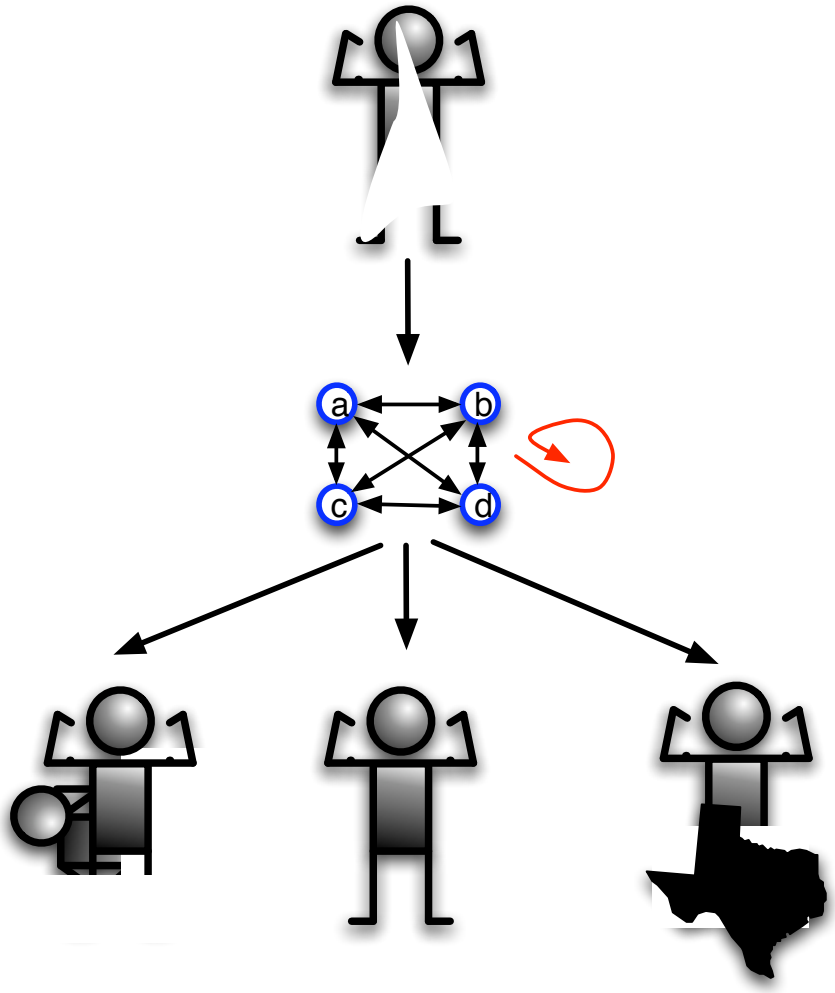

Figure 8: Example of correct (middle) and spurious (left and right) outputs of a Hopfield network when given a corrupted input pattern (top).

Now, assume that $w_{jk} > 0$, meaning that the jth and kth components had, on average, a positive correlation in the training patterns. Consider two cases:

1. $\text{sign}(c_j) = \text{sign}(c_k)$, which means that the jth and kth components are both on, or both off. Hence, they are positively correlated in the current state of the ANN. This **agrees** with w_{jk}, which also indicates a positive correlation between components j and k, since $w_{jk} > 0$. Hence, there is no conflict, only agreement, between c_j, c_k, and w_{jk}. This is reflected in the fact that $w_{jk}c_jc_k > 0$. Since the summations in equation 23 are preceded by negative factors, each pair of activation values that agrees with the corresponding weight will contribute negatively to the total energy, where **lower** total energy means **more** local agreement.
2. $\text{sign}(c_j) \neq \text{sign}(c_k)$, which means that components j and k are negatively correlated in the current state. This **disagrees** with the positive weight between them and is reflected in the fact that $w_{jk}c_jc_k < 0$. Hence, this pair of components will contribute positively to the total energy.

There are a similar set of cases when $w_{jk} < 0$; in those, agreement is signaled when $\text{sign}(c_j) \neq \text{sign}(c_k)$.

Given Hopfield's energy function, we can now sketch an *energy landscape* for an associative network, where network activation states map to energy levels. Figure 9 illustrates a possible landscape for the hypothetical situation of Figure 8. Note that the two spurious patterns occupy local minima and are thus *deceptive* quiescent states for Hopfield search.

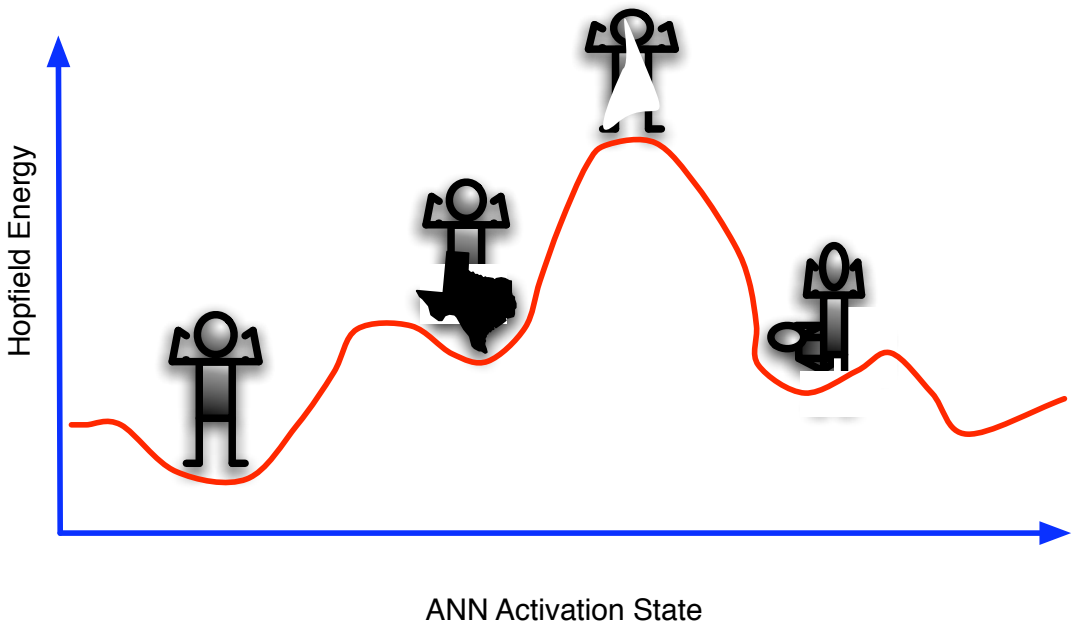

Figure 9: A Hopfield energy landscape in which the initial corrupted pattern from Figure 8 occupies a high-energy point, and the correct pattern resides at the global minimum. Spurious activation states correspond to local minima in the landscape, minima to which the Hopfield network could easily quiesce.

3.4 Hopfield Search in the Brain

The behavior of real brains is believed to exhibit many of the same properties as Hopfield networks in that:

1. The strength of synaptic connections between two neurons (or populations of neurons) often reflects the degree to which the receptive fields of those neurons are correlated, where the *receptive field* of a neuron is that part of the sensory space (for example, a small region in the upper left quadrant of the visual field) for which the neuron appears to be a detector.

2. Certain activity patterns appear to be *stable attractors*, i.e. quiescent states, to which real neural networks eventually transition. Many believe that these attractors represent salient concepts in the brain.

Consider the duck-rabbit flip-flop picture of Figure 10. Most people cannot view this as both a rabbit and duck simultaneously, but both interpretations seem to alternate, particularly if you stare at the picture for a long period. This is evidence that both interpretations represent stable attractors in memory but that there exists some overlap between the neural states corresponding to each.

The components of each attractor stimulate one another, as shown by the thick links in Figure 10, in much the same way that highly correlated nodes in a Hopfield network excite one another due to their high connection weights. However, due to overlap, when one attractor is active, some of its components (such as the *eyes* and *neck* nodes) also stimulate portions of competing attractors.

In addition, neurons are known to *habituate* to stimuli, meaning that they tend to reduce their AP production in the presence of a continuous stimuli. To see this, try staring at something for a few minutes and feel how hard it is to keep your mind only on that particular item; the mind tends to wander.

Hence, a stable attractor such as the duck pattern will habituate while simultaneously lending some stimulation to the *mouth* and *ears* nodes. Eventually, the balance of firing power shifts and *rabbit* becomes the main interpretation, until it habituates and the duck returns.

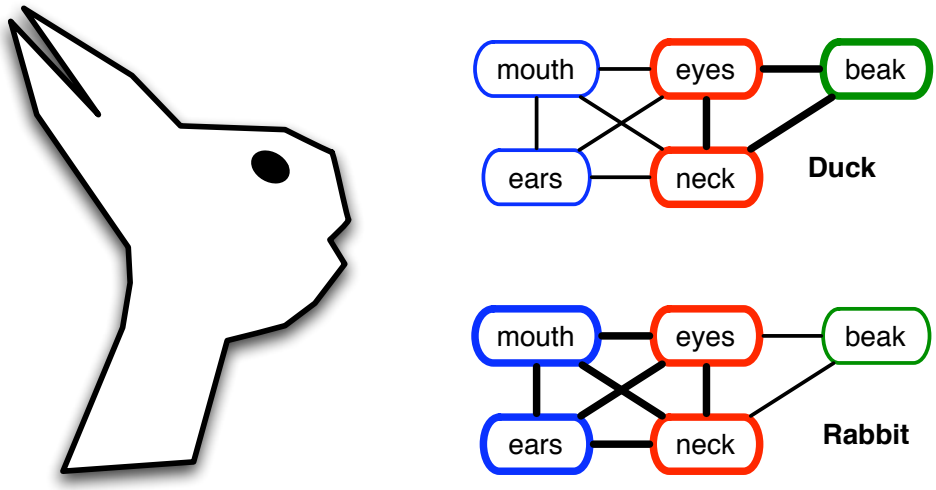

Figure 10: (Left) A flip-flop figure that resembles both a duck and a rabbit, depending upon perspective. (Right) Two copies of the same neural network, with nodes coding for properties of the upper body. When

4 Competitive Networks

In certain types of ANNs, the nodes compete for activity such that the most active nodes can both a) inhibit other nodes from firing, and b) localize learning to only their connections, typically the incoming links. In the brain, topologies in which neurons inhibit many of their neighbors (whether immediate or slightly more distant) are commonplace. This often serves the important function of filtering noise, such that the final stable pattern consists of only the neurons that detect meaningful signals. Other competitions lead to useful structural isomorphisms between aspects of the physical world and regions of the brain specialized to handle those properties. These *topological maps* are beautiful examples how the brain encodes much of the inherent structure of the physical world.

In practical applications of competitive ANNs, the focus is on the weights of the input arcs to each output node, n_i. The vector of input weights to n_i, $\langle w_i \rangle = \langle w_{i1}, w_{i1}, \ldots w_{im} \rangle$, typically represents a *prototype* of the patterns that n_i is (or has learned to become) specialized to detect. In this sense, each output node represents a class or category, and input patterns can be **clustered** according to the output node that they maximally stimulate.

Figure 11 illustrates a standard topology for competitive networks, with one input and one output layer. The output nodes represent categories/classes that essentially compete to capture the different input vectors, with each such vector falling into the category whose prototype it most closely matches.

The generic competitive ANN algorithm is quite simple. Patterns are repeatedly presented to the input layer and activations recorded on the output layer. The output node with the highest activation on pattern P *wins* and has its input weights adjusted so that its prototype more closely resembles P. The update formula for the weights into winning node n_i on input pattern P is then:

$$w_{ij} \leftarrow w_{ij} + \eta(P_j - w_{ij}) \tag{25}$$

where P_j is the jth value of pattern P, i.e., the value loaded onto input neuron j. After many epochs (i.e. presentations of the entire training set), the cases often become clearly segregated into classes, with the prototype vector of each class located close to the middle of subspace delineated by its cases. In this way, competitive networks function as clustering mechanisms for complex data sets.

The curious reader surely wonders how adjusting a prototype weight vector, V, to more closely match an input case, C, could possibly insure that the node attached to V, N_v fires harder on the next presentation of C.

In practice, many competitive networks are not really neural networks at all. They are simply lists of prototype vectors that are matched to case vectors. The vector with the closest match, using a standard Euclidian distance metric for n-dimensional space, is updated in the direction of the case.

However, one can capture these same dynamics in an actual neural network that involves a few key components:

1. A multitude of inhibitory links between all output units.
2. Excitatory links between output units and themselves.
3. Normalized case and prototype vectors

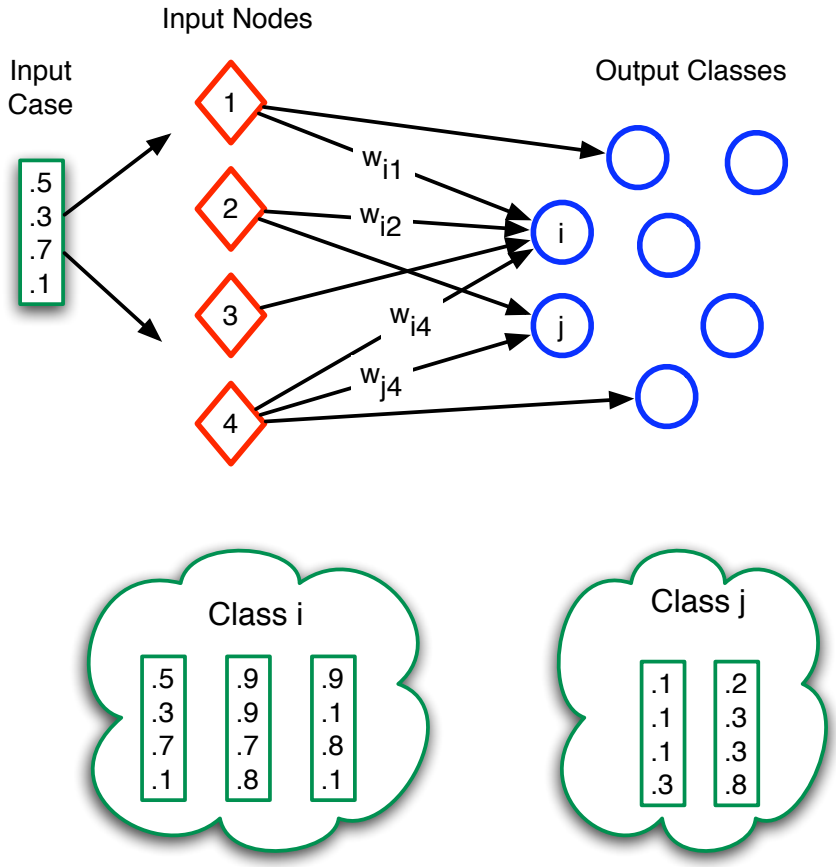

Figure 11: The basic topology of a competitive network, with an input layer and an output layer, with each node in the output layer representing a class whose prototype is given by the input weights to that node. Through learning, input cases become associated with different output nodes and thereby become clustered into instance sets (clouds) of the classes represented by those nodes.

The first two features constitute a **Maxnet**, which insures that when all output nodes are activated, they will compete to shut each other off, while stimulating themselves. After a transitory period, the network settles into a state where the only neuron with a non-zero activity level is the one that originally had the highest activation level.

Maxnets are easy to implement, but care must be taken to properly set the (fixed) weights on the inhibitory and self-stimulating arcs, ϵ and θ, respectively. For example, if the network consists of m output neurons, then the following settings work well:

$$\theta = 1 \quad \epsilon \leq \tfrac{1}{m} \tag{26}$$

Normalization is a slightly more complex issue. First consider the basic Euclidean distance between an input pattern vector, P, and the weight vector for output neuron i, $\langle w_i \rangle$:

$$\sqrt{\sum_{j=1}^{n}(P_j - w_{ij})^2} = \sqrt{\sum_{j=1}^{n} P_j^2 - 2P_j w_{ij} + w_{ij}^2} \tag{27}$$

Now, if input vectors and prototype weight vectors are in normalized form, then we know that their unit length is 1. Hence:

$$\sum_{j=1}^{n} P_j^2 = 1 = \sum_{j=1}^{n} w_{ij}^2 \tag{28}$$

Combining equations 27 and 28, we find that:

$$\sqrt{\sum_{j=1}^{n}(P_j - w_{ij})^2} = \sqrt{\sum_{j=1}^{n} P_j^2 - 2P_j w_{ij} + w_{ij}^2} = \sqrt{2 - 2\sum_{j=1}^{n} P_j w_{ij}} \tag{29}$$

Thus, to minimize the distance between P and $\langle w_i \rangle$, we should maximize $\sum_{j=1}^{n} P_j w_{ij}$, which is just the sum of weighted inputs to neuron i. Hence, the prototype vector with the best match to the input case (i.e., that which is closest to it in Euclidean space) will have the highest sum of weighted inputs and thus will have the highest activation level.[1]

Another way of looking at this is that each normalized vector represents a unit vector in n-dimensional space. The term $\sum_{j=1}^{n} P_j w_{ij}$ is the dot product of these vectors, and with unit vectors, the dot product is equal to the cosine of the angle, ϕ, between the vectors. Then, $cos\phi$ is large, i.e., approaches 1, if and only if ϕ approaches 0, i.e. the vectors are similar. Hence, a good match between the normalized input case and prototype is indicated by a large dot product, $\sum_{j=1}^{n} P_j w_{ij}$.

In summary, if a) the application allows a normalization of input patterns and weight vectors, and b) computational resources permit output neurons to participate in Maxnet competitions to determine the largest activation level, then competitive networks can be implemented as true ANNs.

[1] Competitive learning networks often use a linear activation function, so a higher weighted sum of inputs yields a higher activation.

4.1 Self-Organizing Maps

In many competitive maps, the spatial relationships between the output neurons have no significance, and thus it makes no sense to discuss the *neighbors* of a neuron. However, these relationships have meaning inside the brain, since the firing of a neuron in a region of the brain will often have consequences for nearby neurons.

When neurons are viewed as detectors of various phenomena, whether visual, olfactory, auditory or tactile, it is very often the case that nearby neurons in the brain serve as detectors for similar stimuli. Classic examples include the visual and auditory cortices, the latter of which is shown in Figure **??**. These are referred to as *topological maps*, and they are abundant in the brain.

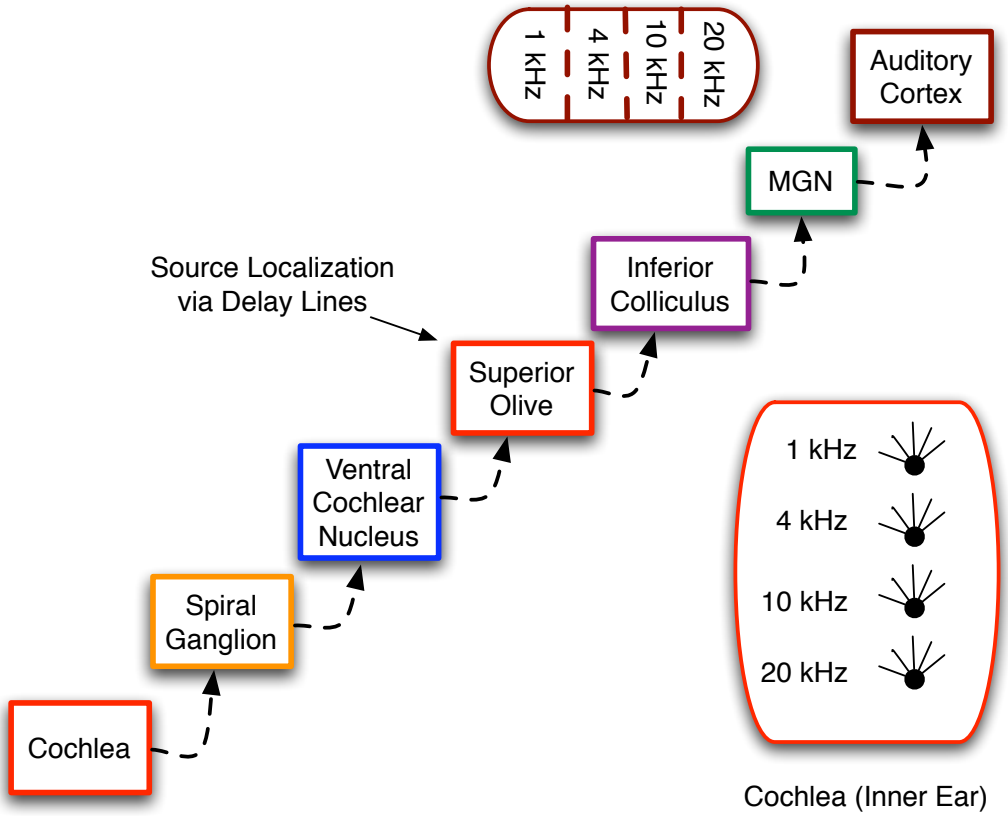

Figure 12: Tonotopic Maps: Topographic mapping between the space of sound frequencies and 7 successive layers of the auditory processing system in the mammalian brain. The correlation between frequencies and neuron locations is preserved through all 7 layers.

The essence of a topological map is the isomorphism between two spaces, at least one of which is a population of neurons. The two spaces are isomorphic when components that are close (distant) in one space map to components that are close (distant) in the other space. This basic idea is shown in Figure 13, where the top mapping is not isomorphic, but the bottom one is.

Invented by Kohonen in the early 1980's, Self-Organizing Maps (SOMs), also known as Kohonen Maps, employ a dynamic mixture of competition and cooperation to enable the emergent formation of an isomorphism between a feature space and an array of neurons [7].

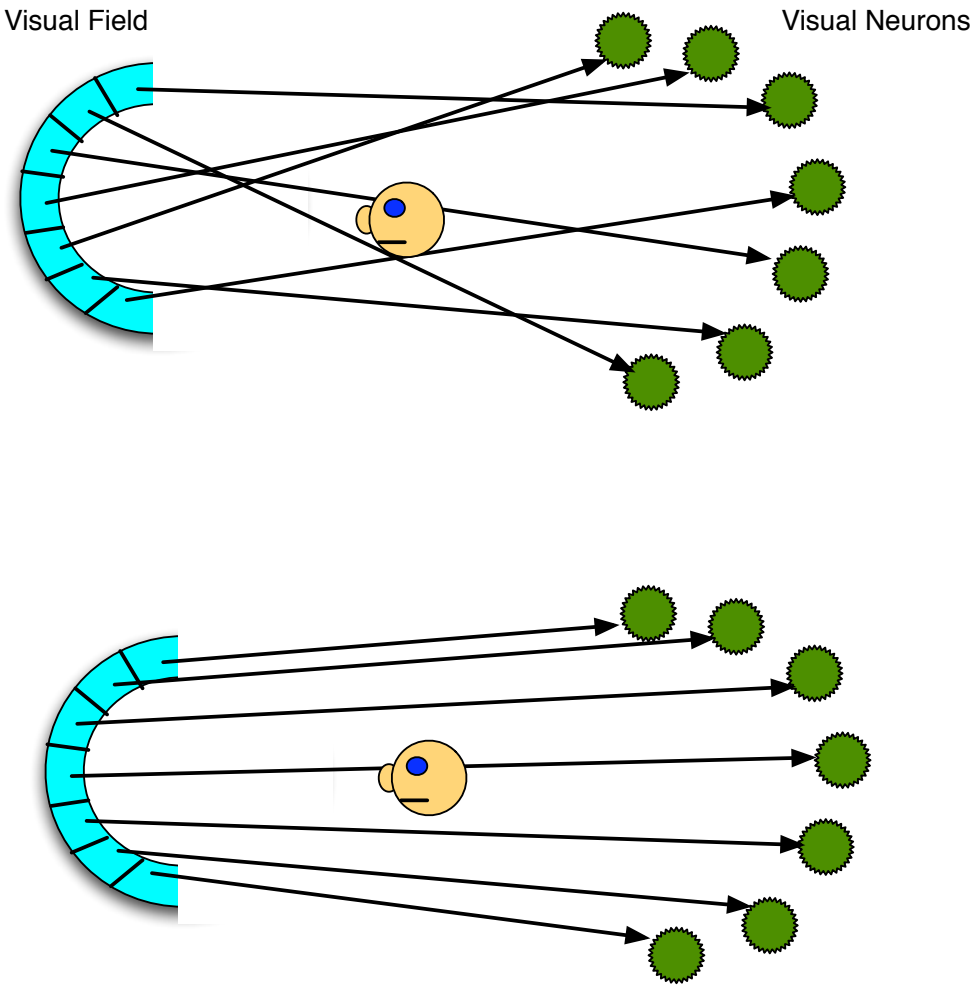

Figure 13: (Above) The typical initial situation for the application of a self-organizing map (SOM), with poor correlation between the two spaces: the visual field and the collection of neurons. (Below) A well-correlated (i.e., isomorphic) mapping between the two spaces.

SOMs operate similarly to standard competitive networks in that a case vector is presented, and the best-matching prototype is modified in the direction of the case. However, Kohonen added an interesting twist: neurons have a meaningful spatial relationship to other neurons, and when an output neuron *wins* a case, it *shares the prize* with its neighbors in that both winner and neighbors adjust their prototypes toward the case.

As shown in Figure 14, the neighborhood for sharing changes during the training phase. It typically begins with a large radius that decreases as training progresses and prototypes specialize toward particular subsets of the case patterns. Over time, this sharing results in an isomorphic mapping in which the prototypes of nearby (in neuron space) neurons are more similar to one another than to those of distant neurons, as shown at the bottom of Figure 14.

In addition, SOMs typically have a dynamic learning rate (η in equation 25) that begins large and decreases during the run.

The net result is a convergence of the Euclidean and topological neighborhoods of the neurons in the SOM, as depicted in Figure 15. The significance of this convergence is multi-faceted, including physical factors such as reduced total *wiring* between brain layers, measured in terms of the lengths of axons and dendrites.

In terms of brain function, a key advantage of topological maps is that if situation A_1 maps to neuron (or more likely neuron population) N_1, presumably via a learning process, then situations similar to A_1, such as A_2 and A_3 would map to neighbors of N_1, which would elicit similar behavior. Thus, the topological map allows the brain to reuse and generalize its behaviors to situations similar to those that it has explicitly learned. The neighbors of N_1 may not share all of its functionality, which would seem appropriate, since A_2 and A_3 may require similar, but not exactly the same, actions as N_1 supports.

4.1.1 Self-Organizing Maps and the Traveling Salesman Problem

SOM's have a wide range of applications, both in biology and in engineering. They provide insights for neuroscientists as to the emergence of the brain's many topological maps, while serving as a powerful clustering algorithm for many practical situations where adjacency and neighborhoods in neuron space have significance.

One of the more intriguing applications that exploits the emergent isomorphism between two spaces is the use of an *elastic* ring of neurons to solve the Traveling Salesman Problem (TSP) [3]. As shown in Figure 16, the neuron space consists of a ring, with neighborhoods defined simply by adjacency on the ring. Each neuron has a two-element weight vector which represents a location on the cartesian plane.

As seen at the bottom of Figure 16, the neurons begin with randomly-assigned weight vectors and thus have random locations in cartesian space. The locations of each TSP city are then used as the training patterns for the neuron population, whose members compete to classify the locations, with the winner and its neighbors updating their weights to move closer to the pattern.

After many rounds of training, the neurons tend to move closer to the city locations. A simulation of the learning process, with visualization of the changing neuron locations, gives the population the appearance of an elastic ring that stretches to fit the cities. If the SOM begins with a large enough neighborhood, with a radius of approximately one tenth the neuron-population size, the ring often forms a good scaffold for a TSP solution.

To use the scaffold, simply go through the city list and associate each with its nearest neuron, i.e., the neuron

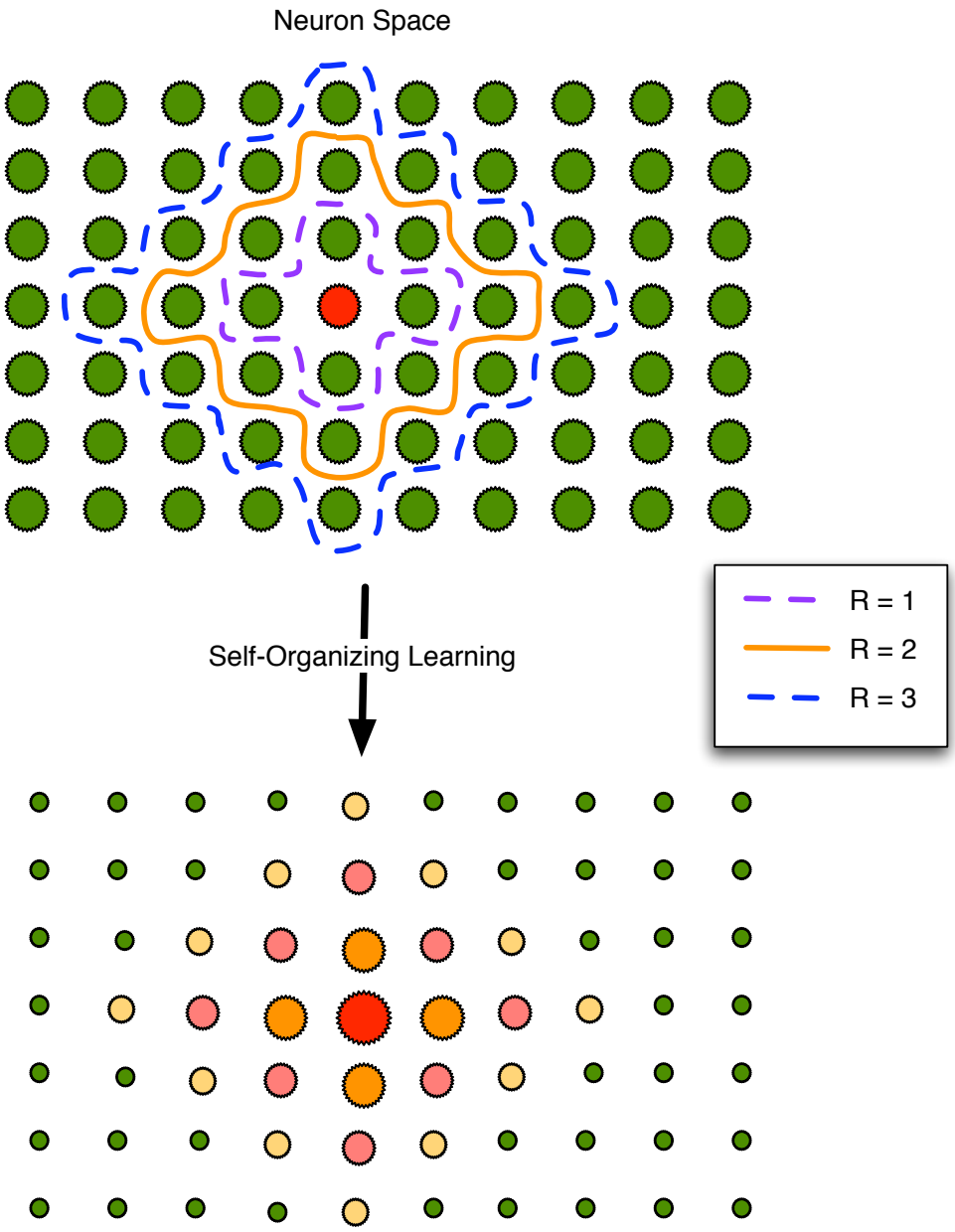

Figure 14: (Top) Illustration of shrinking neighborhoods (about the central red neuron) used to create self-organizing maps (SOMs). R is the radius of the neighborhood, with the Manhattan distance as the metric. (Bottom) After training, the neurons closest to the central neuron, C, have weight vectors that closely ressemble that of C. At greatest distances from C, the resemblance decreases, as denoted by differences in color and size between C and other neurons.

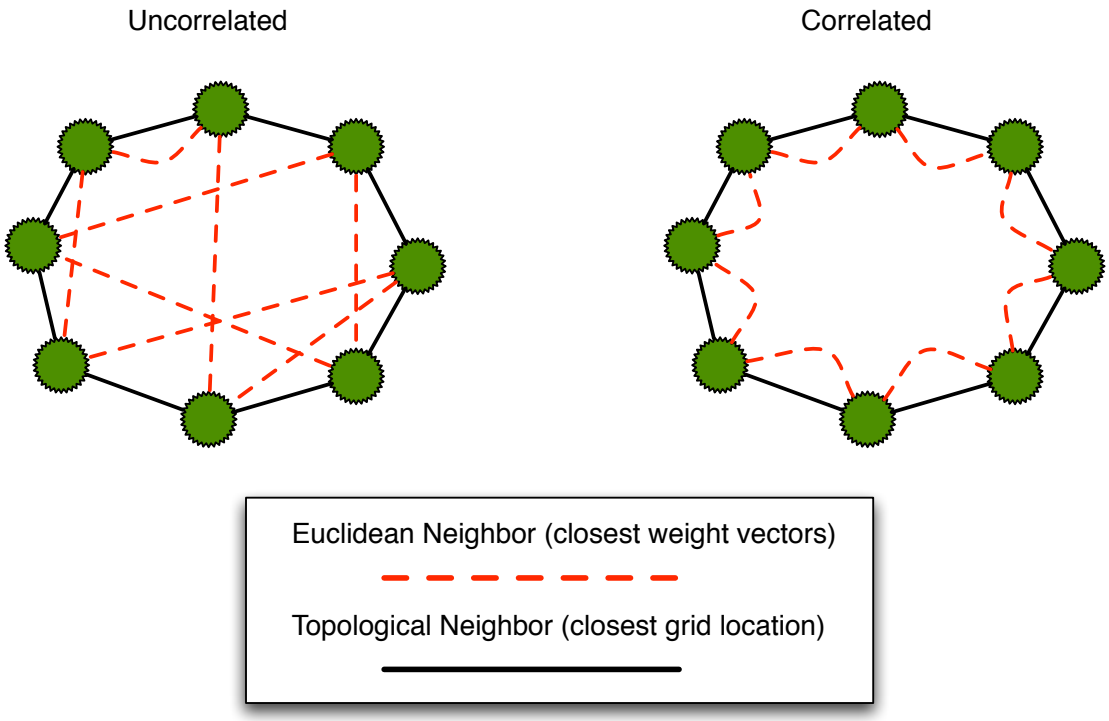

Figure 15: Neurons in a self-organizing map (SOM) are trained to achieve a high correlation between the Euclidean space (which is the space of prototype vectors, which each vector element representing a feature of, for example, the visual field of Figure 13) and the topological space (which is the space in which the neurons reside). Thus, neighboring elements in neuron space have weight vectors (i.e., prototypes) that are neighbors in Euclidian space.

that wins on that city. Once all cities are *attached* to the ring, simply *walk* around the ring and read off the city indices in the order that they appear. The resulting sequence constitutes a TSP solution. If 2 or more cities attach to the same neuron, then the ordering between the cities probably has little consequence and can safely be ignored.[2]

To simplify the implementation, it suffices to use scaled coordinates for the city locations, where all x and y values are real numbers between 0 and 1. The neuron weight vector should also be constrained to lie within the unit (i.e., 1 x 1) plane. Given a TSP city permutation based on the neuron ring in the unit plane (as described above), the actual inter-city distances are easily found by computation (using the original geographic locations), and any efficient solution in the unit plane will translate into a good solution on the original (geographic) map.

References

[1] P. Dayan and L. Abbott, *Theoretical Neuroscience: Computational and Mathematical Modeling of Neural Systems*, The MIT Press, Cambridge, MA, 2001.

[2] K. Doya, *What are the computations of the cerebellum, the basal ganglia, and the cerebral cortex?*, Neural Networks, 12 (1999), pp. 961–974.

[3] R. Durbin and D. Willshaw, *An analogue approach to the traveling salesman problem using an elastic net method*, Nature (London), 326 (1987), pp. 689–691.

[4] S. Haykin, *Neural Networks: A Comprehensive Foundation*, Prentice Hall, Inc., Upper Saddle River, N.J., 1999.

[5] J. Hopfield, *Neural networks and physical systems with emergent collective computational abilities*, Proceedings of the National Academy of Sciences, 79 (1982), pp. 2554–2558.

[6] E. Kandel, J. Schwartz, and T. Jessell, *Principles of Neural Science*, McGraw-Hill, New York, NY, 2000.

[7] T. Kohonen, *Self-Organizing Maps*, Springer, Berlin, 2001.

[8] T. Mitchell, *Machine Learning*, WCB/McGraw-Hill, Boston, MA, 1997.

[9] S. Nolfi and D. Floreano, *Evolutionary Robotics: The Biology, Intelligence, and Technology of Self-Organizing Machines*, The MIT Press, Cambridge, MA, 2000.

[2] If one seeks the optimal TSP solution, then all orderings must be considered. Alternatively, by adding more and more neurons to the population, the probability of two cities mapping to the same neuron decreases rapidly.

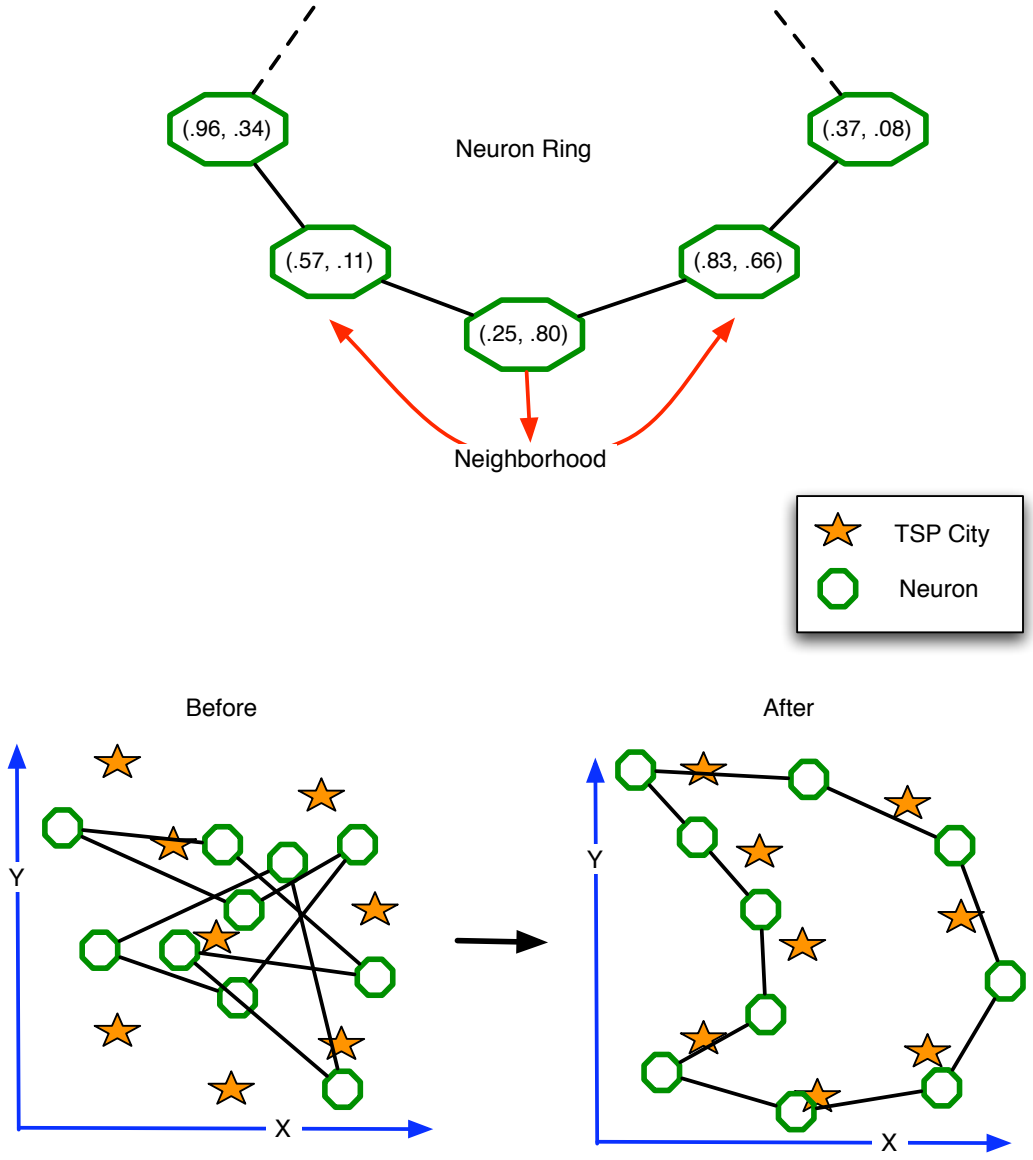

Figure 16: Illustration of the use of a self-organizing map to solve the Traveling Salesman Problem (TSP). (Above) The neuron topology is a ring, with each neuron's weight vector denoting a location in 2-d space. (Bottom) City locations (stars) and neurons (octagons) shown on the cartesian plane, with coordinates corresponding to a) the locations of the cities on a scaled map, and b) the weight vectors of each neuron.