A Tutorial on Helmholtz Machines

Kevin G. Kirby
Department of Computer Science, Northern Kentucky University
June 2006

Preface

"When one understands the causes, all vanished images can easily
be found again in the brain through the impression of the cause.
This is the true art of memory…"

Rene Descartes, *Cogitationes privatae*.
Quoted in Frances Yates, *The Art of Memory* (1966)

Helmholtz machines are artificial neural networks that, through many cycles of sensing and dreaming, gradually learn to make their dreams converge to reality, and, in the process, create a succinct internal model of a fluctuating world.

This tutorial is meant to provide a comfortable introduction for readers with backgrounds in computer science or mathematics, at a level roughly the same as the artificial intelligence textbook of Russell and Norvig [1]. The description here is expanded from the work of Dayan and Hinton [2,3,4,5], with some change of notation and a path of exposition that I hope readers will find helpful. My attempt to provide "comfortable" reading means I include a lot of necessary background information (to keep one from having to pull down an old textbook), a lot of calculational detail (to keep one from having to grab a pencil), and explicit pseudocode.

To minimize prerequisites, I have trimmed out very important discussions of the relationship of Helmholtz machines to the EM algorithm, to autoencoders, to variational learning in Bayesian networks, to information geometry, and to models of the cerebral cortex. This material is meant to lead a reader up to, but not into, that literature.

I wrote this tutorial for a workshop I gave at the Kunsthochschule für Medien at Cologne in June 2006. The thesis of this workshop was that Helmholtz machines can play aesthetic and pedagogical roles that transcend their use in "applications." To date they are, I believe, under-appreciated and under-explained. The slides from my lecture partly address the former aspect; this tutorial is meant to partly address the latter. Comments to kirby@nku.edu are welcome.

1. Notation and Review of Basic Probability for Machine Learning

Bit vectors appear in bold lowercase: e.g., $\mathbf{d} \in \{0,1\}^N$. This subject matter is all about assigning probabilities to bit vectors, that is, about functions p: $\{0,1\}^N \rightarrow [0,1]$, with $p(\mathbf{d}) \geq 0$ and $\sum_{\mathbf{d}} p(\mathbf{d}) = 1$, where the sum runs over all 2^N bit vectors **d**. Such a probability assignment gives the distribution of a discrete random variable **D**: $p(\mathbf{d}) =$ Prob[**D** = **d**], the probability that the random bit vector **D** takes on the specific value **d**.

When using a probability assignment to a pair of bit vectors, p: $\{0,1\}^L \times \{0,1\}^M \rightarrow [0,1]$, we will write $p(\mathbf{xy})$ for $p(\mathbf{x},\mathbf{y})$. This describes the *joint* probability distribution of two random bit vectors **X** and **Y**, with $p(\mathbf{xy})$ = Prob[**X**= **x** and **Y**=**y**].

Several derived quantities come from this $p(\mathbf{xy})$. First, we can define

$$p_1(\mathbf{x}) \equiv \sum_{\mathbf{y}} p(\mathbf{xy}) \quad \text{and} \quad p_2(\mathbf{y}) \equiv \sum_{\mathbf{x}} p(\mathbf{xy}).$$

This process of summing over one argument is called *marginalization*. By a common abuse of notation, we just call these functions $p(\mathbf{x})$ and $p(\mathbf{y})$, respectively, distinguishing them informally by the names of their arguments. As it turns out, $p(\mathbf{x})$ = Prob[**X**= **x**] and $p(\mathbf{y})$ = Prob[**Y**= **y**]. The random variables **X** and **Y** are *independent* if $p(\mathbf{xy}) = p(\mathbf{x})p(\mathbf{y})$.

A second useful quantity is $p(\mathbf{x}|\mathbf{y}) \equiv p(\mathbf{xy}) / p(\mathbf{y})$. Then $p(\mathbf{x}|\mathbf{y})$ = Prob[**X**= **x,** given **Y**=**y**], a *conditional* probability. Note that if **X** and **Y** are independent, then $p(\mathbf{x}|\mathbf{y}) = p(\mathbf{x})$.

These often-used formulas follow from the definitions above:

$$p(\mathbf{xy}) = p(\mathbf{x}|\mathbf{y})\, p(\mathbf{y})$$

$$p(\mathbf{x}) = \sum_{\mathbf{y}} p(\mathbf{x}|\mathbf{y})\, p(\mathbf{y})$$

$$p(\mathbf{xy}|\mathbf{d}) = p(\mathbf{x}|\mathbf{yd})\, p(\mathbf{y}|\mathbf{d})$$

$$p(\mathbf{y}\,|\,\mathbf{x}) = \frac{p(\mathbf{y})}{p(\mathbf{x})}\, p(\mathbf{x}\,|\,\mathbf{y})$$

The last of these is *Bayes Theorem*; it assumes, of course, that $p(\mathbf{x}) > 0$.

A random bit vector **D** with distribution $p(\mathbf{d})$ can itself be described as a joint distribution over all N bits in **d**: $p(\mathbf{d}) = p(d_1 d_2 \ldots d_N)$. It is often the case here that each bit is an independent and identically distributed ("IID") random variable. *Independent*, as above, means that $p(d_1 d_2 \ldots d_N) = p(d_1)p(d_2)\ldots p(d_N)$. *Identically distributed*, here, means that there is some constant probability value p_0 such that for all i=1,2…N:

$$\begin{aligned} p(d_i) &= p_0 \quad && \text{when } d_i = 1 \\ p(d_i) &= 1 - p_0 \quad && \text{when } d_i = 0 \end{aligned}$$

This IID property means that one can write, for any specific bit vector **d**, the compact expression:

$$p(\mathbf{d}) = \prod_i p_0^{\,d_i} \, (1 - p_0)^{1-d_i} \tag{1.1}$$

It is often convenient to recast a probability value p (running from 0 to 1) into a quantity we can call *surprise*: $s \equiv -\log p$. An event occurring with probability 1 has zero surprise, and an event with probability 0 has infinite surprise. Surprise is never negative.

A function of bit vectors $f(\mathbf{d})$ turns **D** into another random variable $f(\mathbf{D})$. The expected value of this function as you apply it to bit vectors drawn with probability $p(\mathbf{d})$ is given by the weighted average

$$\langle f(\mathbf{D}) \rangle = \sum_{\mathbf{d}} p(\mathbf{d}) \, f(\mathbf{d}). \tag{1.2}$$

The expected value of the surprise is called the *entropy*:

$$H(\mathbf{D}) \equiv \langle -\log p(\mathbf{D}) \rangle = -\sum_{\mathbf{d}} p(\mathbf{d}) \log p(\mathbf{d}) \tag{1.3}$$

To fully define the expression on the right, one takes $0 \log 0 \equiv 0$.

When a random variable has low entropy, the outcomes you tend to observe offer little surprise. This occurs, for example, when one particular bit vector occurs with probability near 1, and all the other bit vectors occur with probability near 0. By contrast, when a random variable has high entropy, the observed outcomes tend to have a lot of surprise. This occurs, for example, when all the bit vectors occur with equal probability.

When one has a conditional probability assignment $p(\mathbf{x}|\mathbf{y})$, one speaks of a *conditional entropy*, but the definition is the same:

$$H(\mathbf{X}|\mathbf{y}) = -\sum_{\mathbf{x}} p(\mathbf{x}|\mathbf{y}) \log p(\mathbf{x}|\mathbf{y}). \tag{1.4}$$

Given two probability assignments $p_A(\mathbf{d})$ and $p_B(\mathbf{d})$, one way to quantify how different they are is to use the Kullback-Leibler divergence, also known as the relative entropy, from A to B:

$$\mathrm{KL}[\, p_A(\mathbf{D}) \,,\, p_B(\mathbf{D})] = \sum_{\mathbf{d}} p_A(\mathbf{d}) \, \log \frac{p_A(\mathbf{d})}{p_B(\mathbf{d})} \tag{1.5}$$

This is equal to zero when $p_A = p_B$, and is never negative. Using the properties of the logarithm and the definition of expectation above, we see that

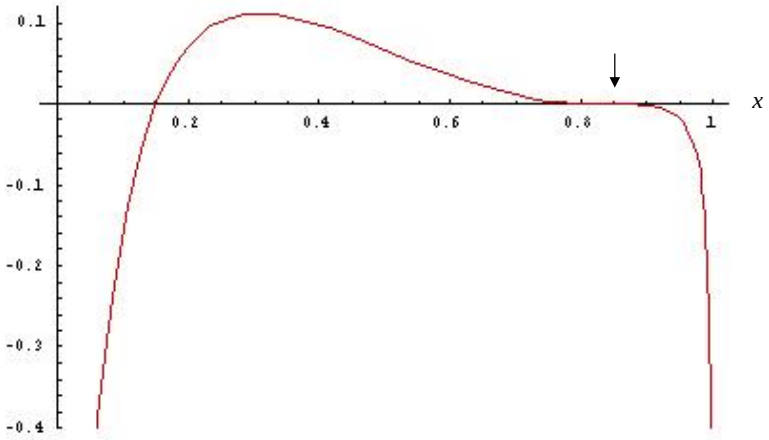
$$\mathrm{KL}[\, p_A(\mathbf{D}) \,,\, p_B(\mathbf{D}) \,] = \langle -\log p_B(D) \rangle_A - \langle -\log p_A(D) \rangle_A$$

In other words, the KL divergence from *A* to *B* is simply the difference in surprise, as averaged by *A*. This is not a symmetric function: KL[$p_A(\mathbf{D})$, $p_B(\mathbf{D})$] ≠ KL[$p_B(\mathbf{D})$, $p_A(\mathbf{D})$], so the terms "difference" or "divergence" are used, rather than the term "distance."

One way to see the asymmetry in KL divergence is to look at the function $A(\mathbf{p},\mathbf{q}) \equiv kl(\mathbf{p},\mathbf{q}) - kl(\mathbf{q},\mathbf{p})$, where **p** and **q** are probability vectors (vectors with positive components summing to 1), and

$$kl(\mathbf{p},\mathbf{q}) \equiv \sum_i p_i \log \frac{p_i}{q_i}$$

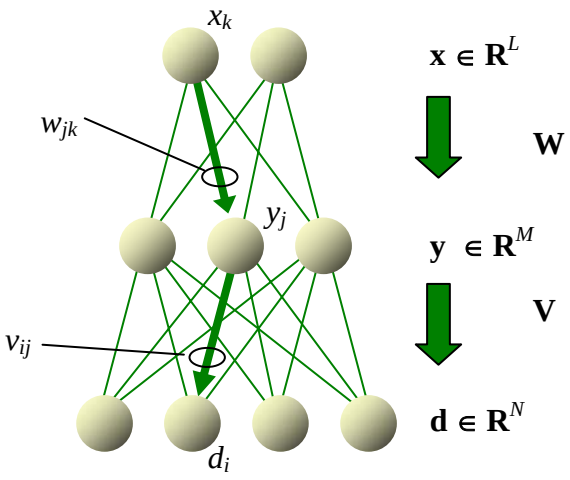
The plot below shows $A([x,1-x]^T, [0.85, 0.15]^T)$. If the KL divergence were symmetric, this curve would be a horizontal line. Note that the degree of asymmetry is very small when **p** and **q** are nearly equal (at $x = 0.85$).



One remark about notation, for those fussy enough, like me, to care. It will be easiest to say "the random bit vector **d**" even though "the random bit vector **D**" is the technically correct phrasing, as bold capital letters denote random variables. (Precisely, a random bit vector **D** is a function from a sample space into $\{0,1\}^n$; whereas **d** is merely an element of $\{0,1\}^n$.) One surviving use for the upper/lower case distinction is exemplified in equations 1.2 – 1.5, where capital letters are placeholders for bound variables. For example, the notation $H(\mathbf{X}|\mathbf{y})$ correctly suggests that we view $H(\mathbf{X}|\mathbf{y})$ as a one-argument function of the bit vector **y**. Such placeholders are necessary because function names (e.g. "*H*", "*p*") are overloaded in the common notations of probability theory, so functions must be disambiguated by the names of their arguments.

2. Notation and Review of Layered Neural Networks

Consider a neural network with 3 layers. We will draw it vertically.



In a linear neural network, the patterns **x**, **y**, **d** are vectors of real numbers. The input pattern **x** passes through the connection weight matrix **W** to produce the pattern **y**. In turn, **y** passes through the connection weight matrix **V** to produce the pattern **d**. Each neuron computes a weighted sum of its inputs, plus an additional bias weight. This yields the following:

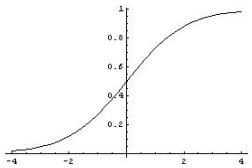
$$y_j = \sum_{k=1}^{L} w_{jk} x_k + w_{j,L+1}, \qquad \text{or simply:} \quad \mathbf{y} = \mathbf{W}[\mathbf{x}|1]$$

$$d_k = \sum_{j=1}^{M} v_{ij} y_j + v_{i,M+1}, \qquad \text{or simply:} \quad \mathbf{d} = \mathbf{V}[\mathbf{y}|1]$$

where the notation [**a**|1] denotes a column vector **a** with a 1 appended on the end. This means the last column of the weight matrices **W** and **V** hold the bias weights. Accordingly, the output pattern of each layer is an affine transformation of the its input pattern. (An affine transformation is a linear transformation followed by a translation.)

The simplest nonlinearity to introduce into this network is to take the linearly transformed input pattern and push it through a nonlinear squashing function such as the logistic or "sigmoid" function:

$$\sigma(u) = 1 / 1 + e^{-u}.$$



Let the vector function $\boldsymbol{\sigma}$ be defined by applying σ to each component. Then the neural network processes patterns as follows:

$$\mathbf{y} = \boldsymbol{\sigma}\,(\,\mathbf{W}[\mathbf{x}|1]\,)$$
$$\mathbf{d} = \boldsymbol{\sigma}\,(\,\mathbf{V}[\mathbf{y}|1]\,)$$

Since σ yields a number between 0 and 1, we can move from interpreting this value as the output of a real-valued neuron, to the *probability* that a binary-valued neuron produces the output 1 (that is, the probability that is "fires"). Then we can write:

$$\mathbf{y} = \textbf{SAMPLE}[\,\mathbf{p}_\mathbf{y}\,] \quad \text{where} \quad \mathbf{p}_\mathbf{y} = \sigma\,(\,\mathbf{W}[\mathbf{x}|1]\,) \tag{2.1a}$$
$$\mathbf{d} = \textbf{SAMPLE}[\,\mathbf{p}_\mathbf{d}\,] \quad \text{where} \quad \mathbf{p}_\mathbf{d} = \sigma\,(\,\mathbf{V}[\mathbf{y}|1]\,) \tag{2.1b}$$

where SAMPLE[p] is a stochastic function that yields 1 with probability p, and 0 with probability $1-p$. **SAMPLE**[**p**] applies SAMPLE to each component of a vector. Because the range of the function σ never reaches 0 or 1, a neuron will never fire (or not fire) with complete certainty. This kind of layered stochastic network will be the starting point for the Helmholtz machine.

3. Top-Down Pattern Generation

The world, as the Helmholtz machine sees it, is made of up patterns of flickering bits, with each bit pattern **d** appearing with some probability $p(\mathbf{d})$. Since **d** can take on an exponential number of values, specifying $p(\mathbf{d})$ requires a *lot* of information. Or it *would* require a lot of information if the world were completely random. In fact, there may be much less information here that it seems, and the Helmholtz machine attempts to exploit this fact.

Start with a data distribution $p(\mathbf{d})$. We want to explain it, or, equivalently, be able to communicate it in a compressed form. In neural representational learning we do the following. We assume a general form for generative distributions that probabilistically produce **d** from the casual chain $1 \rightarrow \mathbf{x} \rightarrow \mathbf{y} \rightarrow \mathbf{d}$. This chain will be implemented as a layered neural network, as illustrated in the previous section. The constant 1 at the start of the chain indicates that the pattern **x** is not regarded as an input to the network, but is stochastically generated from a constant bias input value of 1. Only the layer holding **d** is connected to the world; the layers holding **x** and **y** are considered hidden layers.

A generative distribution G requires specification of three distributions:

$p_G(\mathbf{x})$	- via a bias vector producing internal causes
$p_G(\mathbf{y}\|\mathbf{x})$	- via connection weights from the top to middle layer
$p_G(\mathbf{d}\|\mathbf{y})$	- via connection weights from the middle to bottom layer

Writing $\mathbf{x} \rightarrow \mathbf{y} \rightarrow \mathbf{d}$ is a way of saying we have *conditional* independence of **x** and **d** given **y**, which, by definition, is

$$p_G(\mathbf{xd}|\mathbf{y}) = p_G(\mathbf{x}|\mathbf{y})\, p_G(\mathbf{d}|\mathbf{y}). \tag{3.1}$$

A relation that captures the causal chain more directly by saying that **x** influences **d** only through **y** is

$$p_G(\mathbf{d}|\mathbf{xy}) = p_G(\mathbf{d}|\mathbf{y}).$$

This follows from (3.1) and the relations derived in section 1:

$$p_G(\mathbf{d}\,|\,\mathbf{y}) = \frac{p_G(\mathbf{xd}\,|\,\mathbf{y})}{p_G(\mathbf{x}\,|\,\mathbf{y})} = \frac{p_G(\mathbf{xyd})}{p_G(\mathbf{xy})} = p_G(\mathbf{d}\,|\,\mathbf{xy})$$

This conditional independence means can get the joint distribution, which is the complete description of behavior, merely by multiplying together the 3 "givens" of the generative distribution for $1 \rightarrow \mathbf{x} \rightarrow \mathbf{y} \rightarrow \mathbf{d}$ **:**

$$\begin{aligned} p_G(\mathbf{xyd}) &= p_G(\mathbf{yd}|\mathbf{x})\, p_G(\mathbf{x}) \\ &= [\, p_G(\mathbf{d}|\mathbf{yx})\, p_G(\mathbf{y}|\mathbf{x})\,]\, p_G(\mathbf{x}) \\ &= p_G(\mathbf{d}|\mathbf{y})\, p_G(\mathbf{y}|\mathbf{x})\, p_G(\mathbf{x}) \end{aligned}$$

Conversely, we can get the 3 "givens" from the joint as well:

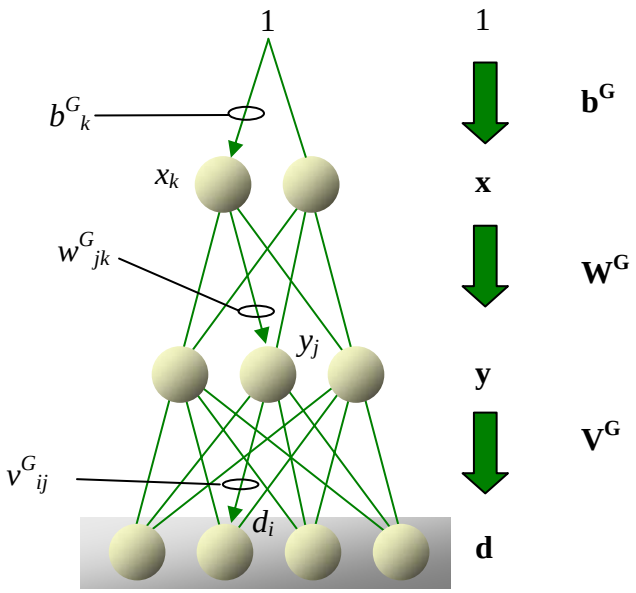
$$\begin{aligned} p_G(\mathbf{x}) &= \textstyle\sum_{\mathbf{yd}} p_G(\mathbf{xyd}) \\ p_G(\mathbf{y}|\mathbf{x}) &= p_G(\mathbf{xy}) \,/\, p_G(\mathbf{x}) = \textstyle\sum_{\mathbf{d}} p_G(\mathbf{xyd}) \,/\, \sum_{\mathbf{yd}} p_G(\mathbf{xyd}) \\ p_G(\mathbf{d}|\mathbf{y}) &= p_G(\mathbf{yd}) \,/\, p_G(\mathbf{y}) = \textstyle\sum_{\mathbf{x}} p_G(\mathbf{xyd}) \,/\, \sum_{\mathbf{xd}} p_G(\mathbf{xyd}) \end{aligned}$$

So we can just speak of $p_G(\mathbf{xyd})$ as *the* generative distribution, packaged nicely in a single function (though unpacking it into the three "directional" factors is more useful).

The ultimate quantity of interest here is not any of these four quantities in themselves, but $p_G(\mathbf{d}) = \sum_{\mathbf{xy}} p_G(\mathbf{xyd})$. The goal is to make the network's $p_G(\mathbf{d})$ as close as possible to the real data distribution $p(\mathbf{d})$.

In a Helmholtz machine, the bottom layer holding **d** is called the *data* layer. The layers above are called *hidden* layers. Generation of a data pattern **d** begins with a pattern **x** at the top layer. The network generates it by sampling $p_G(\mathbf{x})$. This is implemented by having a single bias input (equal to 1) coming into each neuron k in the top layer, with bias weight b^G_k. We collect these in a vector $\mathbf{b}^G$. Similarly, $p_G(\mathbf{y}|\mathbf{x})$ is determined by weight matrix $\mathbf{W}^G$ and $p_G(\mathbf{d}|\mathbf{y})$ is determined by weight matrix $\mathbf{V}^G$.

This is shown in the figure on the next page. The data layer is highlighted to indicate that it is the one layer that interfaces with the outside world.



The set of network connection weight matrices ($\mathbf{b}^G$, $\mathbf{W}^G$, $\mathbf{V}^G$) is nothing but a way to specify ($p_G(\mathbf{x})$, $p_G(\mathbf{y}|\mathbf{x})$, $p_G(\mathbf{d}|\mathbf{y})$), or, equivalently, $p_G(\mathbf{xyd})$. Of course, this is a highly constrained form for a probability distribution. The $p_G(\mathbf{d})$ they produce may not be capable of conforming exactly to the observed $p(\mathbf{d})$, but the machine will try to get it as close as possible.

We get the directional probabilities from the neural network weights as follows, based on equations 1.1 and 2.1:

$$p_G(\mathbf{x}) = \prod_k p_G(x_k)^{x_k} [1 - p_G(x_k)]^{1-x_k} \quad \text{where} \quad p_G(x_k) = \sigma\left(b^G_k\right)$$

$$p_G(\mathbf{y}\,|\,\mathbf{x}) = \prod_j p_G(y_j\,|\,\mathbf{x})^{y_j} [1 - p_G(y_j\,|\,\mathbf{x})]^{1-y_j} \quad \text{where} \quad p_G(y_j\,|\,\mathbf{x}) = \sigma\left(\sum_{k=1}^{L} w^G{}_{jk} x_k + w^G{}_{j,L+1}\right)$$

$$p_G(\mathbf{d}\,|\,\mathbf{y}) = \prod_i p_G(d_i\,|\,\mathbf{y})^{d_i} [1 - p_G(d_i\,|\,\mathbf{y})]^{1-d_i} \quad \text{where} \quad p_G(d_i\,|\,\mathbf{y}) = \sigma\left(\sum_{j=1}^{M} v^G{}_{ij} y_j + v^G{}_{i,M+1}\right)$$

where σ is the sigmoid function $\sigma(u) = (1 + \exp -u)^{-1}$.

Call **x** by itself the "cause." Call **xy** (the state of all the hidden units) the "explanation" of **d**. Of course, the actual number of neurons in each layer is unrestricted. And we can imagine variants of Helmholtz machines with more or fewer layers, different activation functions, lateral connections, and so on [6]. But for the purposes of this tutorial, we concentrate on the architecture depicted above.

4. Energy

Now let us look at the interesting ways the probabilities involved are interrelated in the generative distribution. Given the role of explanations, one might ask about the probability of an explanation **xy** given some fixed piece of generated data **d**:

$$p_G(\mathbf{xy} \mid \mathbf{d}) = \frac{p_G(\mathbf{xyd})}{p_G(\mathbf{d})} = \frac{p_G(\mathbf{xyd})}{\sum_{\mathbf{xy}} p_G(\mathbf{xyd})} \tag{4.1}$$

With the innocuous abbreviation

$$E_G(\mathbf{xy};\mathbf{d}) \equiv -\log p_G(\mathbf{xyd})$$

equation (4.1) becomes

$$p_G(\mathbf{xy} \mid \mathbf{d}) = \frac{\exp\,[-E_G(\mathbf{xy};\mathbf{d})]}{\sum_{\mathbf{xy}} \exp\,[-E_G(\mathbf{xy};\mathbf{d})]}$$

This rewrite is more than just cosmetic. If we use the suggestive name *generative energy* for $E_G(\mathbf{xy};\mathbf{d})$, an analogy to statistical physics emerges.

Suppose the state of a physical system fluctuates among a set of states $\{ q_1, q_2, \ldots \}$. One sign that the system is in *thermal equilibrium* is that the probability of finding the system in a state q_i is related to its energy $E(q_i)$ according to a rule called the *Boltzmann distribution*:

$$p(q_i) = \frac{\exp\,[-E(q_i)/T]}{\sum_i \exp\,[-E(q_i)/T]}$$

where T is the temperature. The denominator is known as the *partition function Z.*

In formal, non-physical systems one can go backwards and devise an "energy" function so that the probability distribution over the states can be said to obey a Boltzmann distribution. This then enables one to plunder statistical physics for useful theorems and relationships.

So we declare that $E_G(\mathbf{xy};\mathbf{d}) \equiv -\log p_G(\mathbf{xyd})$ will be known as *the energy of the explanation* **xy** *of the data pattern* **d**. We are taking the temperature to be T=1. The energy is *the surprise associated with the occurrence of a particular complete state.* The semicolon in the argument list of $E_G(\mathbf{xy};\mathbf{d})$ is meant to suggest the way we will use it below: as a function of **x** and **y**, with **d** held fixed. Since the energy function is just the negative log of the full joint distribution, we can recover all the pieces of the generative distribution from the energy function, so the energy function is a third way to "package" the generative distribution *G*.

Incidentally, the partition function *Z* (the denominator) is just $p_G(\mathbf{d})$, the ultimate quantity of interest in this whole enterprise.

5. Free Energy

Now the goal of Helmholtz machine learning is to find a G (that is, find $\mathbf{b}^G$, $\mathbf{W}^G$, and $\mathbf{V}^G$) that makes $p_G(\mathbf{d})$ as close to $p(\mathbf{d})$ as possible. When this happens, the machine will have made a generative model of the world. So we want to find a G that minimizes the function

$$\Phi(G) \equiv \mathrm{KL}[\, p(\mathbf{D}), p_G(\mathbf{D}) \,] \tag{5.1}$$

It would be nice if we could find a G such that $\Phi(G) = 0$, but this is unlikely to happen since the feedforward neural network implementation of G constrains the distribution. Minimizing (5.1) is somewhat simpler than it looks, since if we use the definition of KL divergence we get

$$\Phi(G) = \sum_{\mathbf{d}} p(\mathbf{d}) \log \frac{p(\mathbf{d})}{p_G(\mathbf{d})} = \sum_{\mathbf{d}} p(\mathbf{d}) \log p(\mathbf{d}) - \sum_{\mathbf{d}} p(\mathbf{d}) \log p_G(\mathbf{d})$$

where the first term does not depend on G and thus can be ignored in our optimization of G. What's left is nothing but the expected surprise of the data generated by the network, weighted by the real-world probability of the data:

$$\Phi_0(\mathrm{G}) \equiv \langle -\log p_G(\mathbf{D}) \rangle = - \sum_{\mathbf{d}} p(\mathbf{d}) \log p_G(\mathbf{d}) . \tag{5.2}$$

As we sample patterns in the real world we keep track of how surprising it would be if the machine generated that pattern. This accumulated surprise as we sample will get smaller as the machine learns to model the world, as $p_G(\mathbf{d})$ moves closer to $p(\mathbf{d})$.

So now our optimization problem is one of finding a G that minimizes this expected surprise $\Phi_0(\mathrm{G})$. As is common in neural network learning, we will use a gradient descent algorithm. That means we will need to compute various gradients ∇ of $\Phi_0(\mathrm{G})$. Since any gradient operator ∇ is linear, we will have

$$\nabla\Phi_0 = \sum_{\mathbf{d}} p(\mathbf{d}) \nabla [-\log p_G(\mathbf{d})] . \tag{5.3}$$

In gradient descent optimization we will make a lot of little changes proportional to $\nabla\Phi_0$. Since the gradient consists of a $p(\mathbf{d})$-weighted sum of the gradients of $-\log p_G(\mathbf{d})$, our descent algorithm can just sample a $\mathbf{d}$ from the real world $p(\mathbf{d})$ distribution, then add in an increment proportional to $\nabla[-\log p_G(\mathbf{d})]$. As we keep adding these up, sampling according to $p(\mathbf{d})$, we get the same weighted sum as in equation (5.3). So finding the various gradients of $-\log p_G(\mathbf{d})$ is all we need to do to prepare a gradient descent algorithm to make p approach p_G.

Accordingly, we turn our attention to minimizing $-\log p_G(\mathbf{d})$, the generative surprise, and rearrange it to get an interesting new form:

$$\begin{aligned} -\log p_G(\mathbf{d}) &= -\log p_G(\mathbf{d})\, 1 \\ &= -\log p_G(\mathbf{d}) \,[\, \textstyle\sum_{\mathbf{xy}} p_G(\mathbf{xy}|\mathbf{d}) \,] \\ &= -\textstyle\sum_{\mathbf{xy}} p_G(\mathbf{xy}|\mathbf{d}) \log p_G(\mathbf{d}) \end{aligned}$$

$$
\begin{aligned}
&= -\sum_{\mathbf{xy}} p_G(\mathbf{xy}|\mathbf{d}) \log [\, p_G(\mathbf{xyd}) / p_G(\mathbf{xy}|\mathbf{d}) \,] \\
&= -\sum_{\mathbf{xy}} p_G(\mathbf{xy}|\mathbf{d}) \log p_G(\mathbf{xyd}) \;+\; \sum_{\mathbf{xy}} p_G(\mathbf{xy}|\mathbf{d}) \log p_G(\mathbf{xy}|\mathbf{d}) \\
&= \sum_{\mathbf{xy}} p_G(\mathbf{xy}|\mathbf{d})\, E_G(\mathbf{xy};\mathbf{d}) \;-\; H_G(\mathbf{XY}|\mathbf{d}) \\
&= \langle\, E_G(\mathbf{XY};\mathbf{d}) \,\rangle_G \;-\; H_G(\mathbf{XY}|\mathbf{d})
\end{aligned}
$$

The first term is the average generative energy of explanations we observe with a pattern **d** when we let the generative distribution operate. The second term is the entropy of explanations we observe with a pattern **d** when we let the generative distribution operate.

In statistical physics, the *Helmholtz free energy* of a system with average energy $\langle E \rangle$, temperature T, and entropy H, is given by:

$$F = \langle E \rangle - TH$$

Since the expression for $-\log p_G(\mathbf{d})$ above displays it as "average energy minus entropy," it is analogous to a Helmholtz free energy with temperature 1. We can call this quantity *the free energy of* **d** *in* G:

$$F_G(\mathbf{d}) \equiv -\log p_G(\mathbf{d}) = \langle\, E_G(\mathbf{XY};\mathbf{d}) \,\rangle_G - H_G(\mathbf{XY}|\mathbf{d}). \tag{5.4}$$

Thus there is a three-way identification here dealing with machine-generated patterns:

free energy of pattern $\leftrightarrow$ *surprise of pattern* $\leftrightarrow$ *average energy minus entropy*

So we see that to minimize the KL divergence from the real world data distribution to the Helmholtz machine data distribution means, according to (5.3) above, that we need to minimize $\langle F_G(\mathbf{D}) \rangle$, the generative free energy averaged according to the real world distribution.

At this point it would seem that gradient-descent learning in a Helmholtz machine all boils down to calculating some derivatives of $F_G(\mathbf{d})$. Specifically, we need to calculate three sets of derivatives:

$$\frac{\partial}{\partial b_k^G} F_G(\mathbf{d}) \qquad \frac{\partial}{\partial w_{jk}^G} F_G(\mathbf{d}) \qquad \frac{\partial}{\partial v_{ij}^G} F_G(\mathbf{d})$$

To calculate these derivatives we need to express $F_G(\mathbf{d})$ in terms of the weights via the formulas for $p_G(\mathbf{x})$, $p_G(\mathbf{y}|\mathbf{x})$, $p_G(\mathbf{d}|\mathbf{y})$ at the end of section 3. Try out the math. Things don't seem to work out very well; the equations do not "clean up nice" and give us simple expressions for the derivatives. We must think a little more deeply.

6. Variational Free Energy

We pause now and consider *another* distribution $p_R(\mathbf{xy}|\mathbf{d})$, which can be any arbitrary distribution for the moment. We have no need for a full joint $p_R(\mathbf{xyd})$ in what follows. In particular, there is no role for $p_R(\mathbf{d})$.

For fixed data pattern **d**, compute the Kullback-Leibler divergence of explanations from p_R to the generative distribution p_G:

$$\mathrm{KL}[p_R(\mathbf{XY}|\mathbf{d}), p_G(\mathbf{XY}|\mathbf{d})] \equiv \sum_{\mathbf{xy}} p_R(\mathbf{xy}|\mathbf{d}) \log \frac{p_R(\mathbf{xy} \mid \mathbf{d})}{p_G(\mathbf{xy} \mid \mathbf{d})}$$

$$= \sum_{\mathbf{xy}} p_R(\mathbf{xy}|\mathbf{d}) \log p_R(\mathbf{xy}|\mathbf{d}) \;-\; \sum_{\mathbf{xy}} p_R(\mathbf{xy}|\mathbf{d}) \log p_G(\mathbf{xy}|\mathbf{d})$$

$$= \sum_{\mathbf{xy}} p_R(\mathbf{xy}|\mathbf{d}) \log p_R(\mathbf{xy}|\mathbf{d}) \;-\; \sum_{\mathbf{xy}} p_R(\mathbf{xy}|\mathbf{d}) \log \frac{p_G(\mathbf{xyd})}{p_G(\mathbf{d})}$$

$$= -H_R(\mathbf{XY}|\mathbf{d}) \;-\; \sum_{\mathbf{xy}} p_R(\mathbf{xy}|\mathbf{d}) \log p_G(\mathbf{xyd}) \;+\; \sum_{\mathbf{xy}} p_R(\mathbf{xy}|\mathbf{d}) \log p_G(\mathbf{d})$$

$$= -H_R(\mathbf{XY}|\mathbf{d}) \;+\; \sum_{\mathbf{xy}} p_R(\mathbf{xy}|\mathbf{d})\, E_G(\mathbf{xy;d}) \;+\; \sum_{\mathbf{xy}} p_R(\mathbf{xy}|\mathbf{d}) \log p_G(\mathbf{d})$$

$$= -H_R(\mathbf{XY}|\mathbf{d}) \;+\; \langle\, E_G(\mathbf{XY};\mathbf{d}) \,\rangle_R \;+\; \log p_G(\mathbf{d}) \; \cancel{\sum_{\mathbf{xy}} p_R(\mathbf{xy}|\mathbf{d})}$$

$$= -H_R(\mathbf{XY}|\mathbf{d}) \;+\; \langle\, E_G(\mathbf{XY};\mathbf{d}) \,\rangle_R \;-\; F_G(\mathbf{d}).$$

Rewrite this to get the free energy $F_G(\mathbf{d})$, and juxtapose this with the previous section's expression for the same quantity:

$$F_G(\mathbf{d}) = \langle\, E_G(\mathbf{XY};\mathbf{d}) \,\rangle_G \;-\; H_G(\mathbf{XY}|\mathbf{d})$$
$$F_G(\mathbf{d}) = \langle\, E_G(\mathbf{XY};\mathbf{d}) \rangle_R \;-\; H_R(\mathbf{XY}|\mathbf{d}) \;-\; \mathrm{KL}[p_R(\mathbf{XY}|\mathbf{d}), p_G(\mathbf{XY}|\mathbf{d})]$$

So what's going on? We have a simple expression for the generative Helmholtz free energy, and then a more complicated expression for the same thing involving a whole new quantity, the distribution p_R. As it turns out, this involves us in something called a *variational method*.

Since KL divergences are never negative, we have $\langle\, E_G(\mathbf{XY};\mathbf{d}) \,\rangle_R \;-\; H_R(\mathbf{XY}|\mathbf{d}) \;- F_G(\mathbf{d}) \;\geq 0$, i.e.

$$F_G(\mathbf{d}) \;\leq\; \langle\, E_G(\mathbf{XY};\mathbf{d}) \,\rangle_R \;-\; H_R(\mathbf{XY}|\mathbf{d})$$

Define the *variational* free energy (from R to G) as

$$\boxed{\begin{aligned} F^R{}_G(\mathbf{d}) &\equiv \langle\, E_G(\mathbf{XY};\mathbf{d}) \,\rangle_R \;-\; H_R(\mathbf{XY}|\mathbf{d}) && \text{(6.1a)} \\ &= F_G(\mathbf{d}) \;+\; \mathrm{KL}[p_R(\mathbf{XY}|\mathbf{d}), p_G(\mathbf{XY}|\mathbf{d})] && \text{(6.1b)} \end{aligned}}$$

Compare the first of these to equation 4.5:

$$F^{G}{}_{G}(\mathbf{d}) = F_G(\mathbf{d})$$

This is so important, let's paraphrase all this:

variational free energy from R to G	=	average energy of G in R − entropy of G
	=	surprise in G + divergence from R to G
	≥	free energy of G
variational free energy from G to G	=	free energy of G

Were the variational free energy $F^{R}{}_{G}(\mathbf{d})$ minimized on both R and G, we would find that $p_R \rightarrow p_G$ and $F_G(\mathbf{d})$ would be minimized.

So far nothing in this section has assumed anything about where $p_R(\mathbf{xy}|\mathbf{d})$ comes from. Now is the time to pin it down.

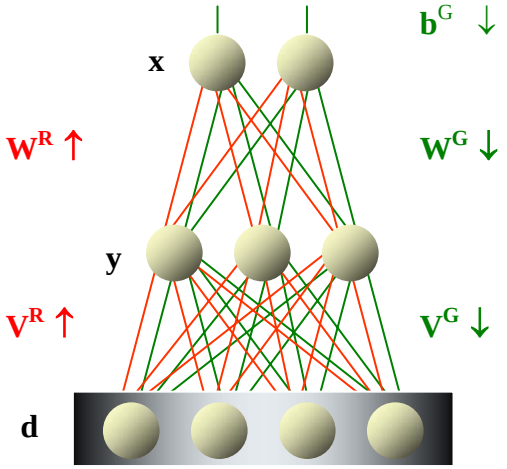
7. Bottom-Up Pattern Recognition

Call $p_R(\mathbf{xy}|\mathbf{d})$ the *recognition* distribution, and assume it follows a causal chain $\mathbf{d} \rightarrow \mathbf{x} \rightarrow \mathbf{y}$. That means we can factor $p_R(\mathbf{xy}|\mathbf{d}) = p_R(\mathbf{x}|\mathbf{y})\, p_R(\mathbf{y}|\mathbf{d})$. Let these two factors be determined by upward connections in the Helmholtz machines, called the recognition weights, $\mathbf{V}^{\mathrm{R}}$ and $\mathbf{W}^{\mathrm{R}}$. Analogously to the equations for generation (at the end of section 3), we have the equations of recognition:

$$p_R(\mathbf{x}\,|\,\mathbf{y}) = \prod_k p_R(x_k\,|\,\mathbf{y})^{x_k}\,[\,1 - p_R(x_k\,|\,\mathbf{y})]^{1-x_k} \qquad \text{where} \quad p_R(x_k\,|\,\mathbf{y}) = \sigma\left(\sum_{j=1}^{M} w^{R}_{kj} y_j + w^{R}{}_{k,M+1}\right)$$

$$p_R(\mathbf{y}\,|\,\mathbf{d}) = \prod_j p_R(y_j\,|\,\mathbf{d})^{y_j}\,[\,1 - p_R(y_j\,|\,\mathbf{d})]^{1-y_j} \qquad \text{where} \quad p_R(y_j\,|\,\mathbf{d}) = \sigma\left(\sum_{i=1}^{N} v^{R}_{ji} d_i + v^{R}{}_{j,N+1}\right)$$

Since the pattern **d** is an input, there are no bias weights coming into the data layer from below. Thus unlike the generative case there are only two equations here. The resulting neural network, with both generative and recognition connections, is shown below.



8. Learning

We saw in section 5 that a Helmholtz machine can learn the structure of the world by minimizing its own generative free energy $F_G(\mathbf{d})$. The idea is to use a gradient descent method for this. However, calculation of gradients is complicated if one uses the direct form of $F_G(\mathbf{d})$. That led us to turn to the variational free energy $F^R{}_G(\mathbf{d})$, involving a separate recognition distribution. Minimizing $F^R{}_G(\mathbf{d})$ simultaneously minimizes the free energy $F_G(\mathbf{d})$ and the KL divergence from $p_R(\mathbf{XY}|\mathbf{d})$ to $p_G(\mathbf{XY}|\mathbf{d})$, which will make recognition and generation approximate inverses of each other.

Each iteration in this revised gradient descent algorithm will involve two phases, one involving the generative weights and one involving the recognition weights. That is, we alternately make small changes in G and in R.

In the first phase, to be called the "wake" phase, we make a small change in the generative weights to decrease the variational free energy from equation (6.1b):

$$F^R{}_G(\mathbf{d}) \;=\; F_G(\mathbf{d}) \;+\; \mathrm{KL}[p_R(\mathbf{XY}|\mathbf{d}),\, p_G(\mathbf{XY}|\mathbf{d})].$$

There is one free variable here: **d**. Where does it come from? We sample it from the real world $p(\mathbf{d})$ (*not* from the generative distribution), as our original goal was to minimize $\langle F_G(\mathbf{d})\rangle$ where the average is taken according to the real world distribution. So we sample a **d** from the world and tweak the $\mathbf{b}^G$, $\mathbf{W}^G$, $\mathbf{V}^G$ so as to decrease $F^R{}_G(\mathbf{d})$.

In the second phase, to be called the "sleep" phase, we make a small change in the recognition weights to decrease not $F^R{}_G(\mathbf{d})$ but a slightly different quantity that approximates it:

$$\tilde{F}^R{}_G(\mathbf{d}) \;= F_G(\mathbf{d}) \;+\; \mathrm{KL}[p_G(\mathbf{XY}|\mathbf{d}),\, p_R(\mathbf{XY}|\mathbf{d})].$$

Note the order of the KL arguments is switched! Since the KL divergence is not symmetric, this is a nontrivial kludge. It gives us only one term involving *R*, rather than two– a very efficient form which simplifies the learning algorithm.

These adjustments will occur by gradient descent. So we now turn to evaluation of the derivatives of $F^R{}_G(\mathbf{d})$ and $\widetilde{F}^R{}_G(\mathbf{d})$ in order to derive the algorithm.

9. Wake

For the wake phase we will take the derivatives of the variational free energy with respect to generative weights. Let's write $F^R{}_G(\mathbf{d})$ in the form from equation (6.1a):

$$\begin{aligned} F^R{}_G(\mathbf{d}) &= \langle E_G(\mathbf{XY};\mathbf{d}) \rangle_R - H_R(\mathbf{XY}|\mathbf{d}) \\ &= \sum_{\mathbf{xy}} p_R(\mathbf{xy}|\mathbf{d})\, E_G(\mathbf{xy};\mathbf{d}) - \text{const}(R) \end{aligned}$$

noting that the entropy depends only on *R*, so can be considered a constant as we now focus on dependence on *G*.

Let the symbol ∇_G stand for any derivative with respect to the generative biases and weights (i.e. for $\partial/\partial b^G{}_k$, $\partial/\partial w^G{}_{jk}$, or $\partial/\partial v^G{}_{ij}$). Then:

$$\begin{aligned} \nabla_G F^R{}_G(\mathbf{d}) &= \nabla_G \sum_{\mathbf{xy}} p_R(\mathbf{xy}|\mathbf{d})\, E_G(\mathbf{xy};\mathbf{d}) - \nabla_G \text{const}(R) \\ &= \sum_{\mathbf{xy}} p_R(\mathbf{xy}|\mathbf{d})\, \nabla_G E_G(\mathbf{xy};\mathbf{d}) \\ &= \langle \nabla_G E_G(\mathbf{XY};\mathbf{d}) \rangle_R \end{aligned} \qquad (9.1)$$

Now our gradient descent learning algorithm will sample a pattern **d** from the world (according to $p(\mathbf{d})$) then update the weights by the small step downhill $-\varepsilon \nabla_G F^R{}_G(\mathbf{d})$, where $\varepsilon > 0$ is small. How do we evaluate this? The right hand side just above says we just sample $\nabla_G E_G(\mathbf{xy};\mathbf{d})$ where **x** and **y** are drawn from the *recognition* distribution.

We turn to evaluating this gradient of the energy:

$$\begin{aligned} \nabla_G E_G(\mathbf{xy};\mathbf{d}) &= -\nabla_G \log p_G(\mathbf{xyd}) \\ &= -\nabla_G \log p_G(\mathbf{x})\, p_G(\mathbf{y}|\mathbf{x})\, p_G(\mathbf{d}|\mathbf{y}) \\ &= -\nabla_G \log p_G(\mathbf{x}) - \nabla_G \log p_G(\mathbf{y}|\mathbf{x}) - \nabla_G \log p_G(\mathbf{d}|\mathbf{y}) \end{aligned} \qquad (9.2)$$

The needed expressions for $\log p_G(\mathbf{x})$, $\log p_G(\mathbf{y}|\mathbf{x})$, and $\log p_G(\mathbf{d}|\mathbf{y})$ are available from the last equations in section 3 for the three layers of weights from the top down:

$$\begin{aligned}
\log\ p_G(\mathbf{x}) &= \log \prod_k \xi_k^{\,x_k}\ (1-\xi_k)^{1-x_k} \qquad \text{where}\ \ \xi_k \equiv \sigma\left(b_k^G\right) \\
&= \sum_k x_k \log\ \xi_k \quad + \quad \sum_k (1-x_k)\log\,(1-\xi_k)
\end{aligned}$$

$$\begin{aligned}
\log\ p_G(\mathbf{y}|\mathbf{x}) &= \log \prod_j \psi_j^{\,y_j}\ (1-\psi_j)^{1-y_j} \qquad \text{where}\ \ \psi_j \equiv \sigma\left(\sum_{k=1}^{L} w^G{}_{jk} x_k + w^G{}_{j,L+1}\right) \\
&= \sum_j y_j \log\ \psi_j \quad + \quad \sum_j (1-y_j)\log\,(1-\psi_j)
\end{aligned}$$

$$\begin{aligned}
\log\ p_G(\mathbf{d}|\mathbf{y}) &= \log \prod_i \delta_i^{\,d_i}\ (1-\delta_i)^{1-d_i} \qquad \text{where}\ \ \delta_i \equiv \sigma\left(\sum_{j=1}^{M} v^G{}_{ij} y_j + v^G{}_{i,M+1}\right) \\
&= \sum_i d_i \log\ \delta_i \quad + \quad \sum_i (1-d_i)\log\,(1-\delta_i)
\end{aligned}$$

The various connection weight derivatives that make up $\nabla_G E_G(\mathbf{xy};\mathbf{d})$ are then straightforward to calculate. These calculations use the fact that if $a = \sigma(b)$, then $da/db = a\,(1–a)$.

$$\begin{aligned}
\frac{\partial}{\partial b_k^G}\log p_G(\mathbf{x}) &= \frac{\partial}{\partial b_k^G}\sum_K x_K \log\ \xi_K \quad + \quad \frac{\partial}{\partial b_k^G}\sum_K (1-x_K)\log\,(1-\xi_K) \\
&= \frac{\partial}{\partial b_k^G}[x_k \log\ \xi_k\ ] \ + \ \frac{\partial}{\partial b_k^G}[(1-x_k)\log\,(1-\xi_k)] \\
&= x_k \frac{\partial}{\partial b_k^G}\log\ \xi_k \quad + \ (1-x_k)\frac{\partial}{\partial b_k^G}\log\,(1-\xi_k) \\
&= \frac{x_k}{\xi_k}\frac{\partial}{\partial b_k^G}\xi_k \quad + \quad \frac{(1-x_k)}{(1-\xi_k)}\frac{\partial}{\partial b_k^G}(1-\xi_k) \\
&= \frac{x_k}{\xi_k}\frac{\partial}{\partial b_k^G}\xi_k \quad - \quad \frac{(1-x_k)}{(1-\xi_k)}\frac{\partial}{\partial b_k^G}\xi_k \\
&= \frac{x_k}{\xi_k}\xi_k(1-\xi_k) \quad - \quad \frac{(1-x_k)}{(1-\xi_k)}\xi_k(1-\xi_k) \\
&= x_k(1-\xi_k) \quad - \quad \xi_k(1-x_k) \\
&= x_k \ - \xi_{k.}
\end{aligned}$$

$$\frac{\partial}{\partial b_k^G}\log p_G(\mathbf{y}\,|\,\mathbf{x}) = 0$$

$$\frac{\partial}{\partial b_k^G}\log p_G(\mathbf{d}\,|\,\mathbf{y}) = 0$$

Similar calculations show that for the top-to-middle weights:

$$\frac{\partial}{\partial w^G_{jk}} \log p_G(\mathbf{x}) = 0$$

$$\frac{\partial}{\partial w^G_{jk}} \log p_G(\mathbf{y} \mid \mathbf{x}) = \begin{cases} (y_j - \psi_j) x_k, & k = 1,2,...,L \\ y_j - \psi_j, & k = L+1 \end{cases}$$

$$\frac{\partial}{\partial w^G_{jk}} \log p_G(\mathbf{d} \mid \mathbf{y}) = 0$$

And for the middle-to-bottom weights:

$$\frac{\partial}{\partial v^G_{ij}} \log p_G(\mathbf{x}) = 0$$

$$\frac{\partial}{\partial v^G_{ij}} \log p_G(\mathbf{y} \mid \mathbf{x}) = 0$$

$$\frac{\partial}{\partial v^G_{ij}} \log p_G(\mathbf{d} \mid \mathbf{y}) = \begin{cases} (d_i - \delta_i) y_j, & j = 1,2,...,M \\ d_i - \delta_i, & j = M+1 \end{cases}$$

Combining all this with equation (8.2) and writing them in vector form, we have

$$\nabla_{\mathbf{b}^G} E_G(\mathbf{xy};\mathbf{d}) = -(\mathbf{x} - \boldsymbol{\xi})$$
$$\nabla_{\mathbf{W}^G} E_G(\mathbf{xy};\mathbf{d}) = -(\mathbf{y} - \boldsymbol{\psi}) [\, \mathbf{x} \mid 1 \,]^T$$
$$\nabla_{\mathbf{V}^G} E_G(\mathbf{xy};\mathbf{d}) = -(\mathbf{d} - \boldsymbol{\delta}) [\, \mathbf{y} \mid 1 \,]^T$$

Keep note of where the quantities on the right hand side came from: **d** came from the world; **x** and **y** came from the recognition distribution given **d**; and **ξ, ψ** and **δ** came from the generative distribution given **x** and **y**.

This means gradient descent takes the form of very simple weight increments at each layer:

$$\mathbf{b}^G \mathrel{+}= \varepsilon (\mathbf{x} - \boldsymbol{\xi})$$
$$\mathbf{W}^G \mathrel{+}= \varepsilon (\mathbf{y} - \boldsymbol{\psi}) [\, \mathbf{x} \mid 1 \,]^T$$
$$\mathbf{V}^G \mathrel{+}= \varepsilon (\mathbf{d} - \boldsymbol{\delta}) [\, \mathbf{y} \mid 1 \,]^T.$$

where ε is the learning rate. It is not unusual in "neural smithing" to allow learning rates to vary from layer to layer and from iteration to iteration.

This rule has fortuitously turned out to be nothing but a variant of the classic *delta rule*, in which the change of a connection weight is proportional to an outer product. That is, weight matrices change according a form like $\mathbf{M} \mathrel{+}= \mathbf{a}\mathbf{b}^T$. In components, this is

$$m_{ij} \mathrel{+}= a_i b_j$$

so the change in the connection between *i* and *j* only involves what happens at the neurons at each end *i* and *j*. Indeed, algorithms for changing neural network weight matrices can only fairly be called "neural network algorithms" if the weight changes are local in this way.

We can summarize the wake phase of an iteration in the following pseudocode.

WAKE PHASE

// Experience reality!
$\mathbf{d}$ = getSampleFromWorld( )

// Pass sense datum up through recognition network
$\mathbf{y}$ = SAMPLE(σ ($\mathbf{V}^R$ [$\mathbf{d}$ | 1] ^{T}))
$\mathbf{x}$ = SAMPLE(σ ($\mathbf{W}^R$[$\mathbf{y}$ | 1] ^{T}))

// Pass back down through generation network, saving computed probabilities
$\xi = \sigma$ ($\mathbf{b}^G$)
$\psi = \sigma$ ($\mathbf{W}^G$ [$\mathbf{x}$ | 1] ^{T})
$\delta = \sigma$ ($\mathbf{V}^G$ [$\mathbf{y}$ | 1] ^{T})

// Adjust generative weights by delta rule
$\mathbf{b}^G$ += ε ($\mathbf{x} - \xi$)
$\mathbf{W}^G$ += ε ($\mathbf{y} - \psi$) [$\mathbf{x}$ | 1]^T
$\mathbf{V}^G$ += ε ($\mathbf{d} - \delta$) [$\mathbf{y}$ | 1]^T

10. Sleep

For the wake phase we will take the derivatives with respect to recognition weights of an approximation $\widetilde{F}^R{}_G(\mathbf{d})$ to the variational free energy $F^R{}_G(\mathbf{d})$:

$$\widetilde{F}^R{}_G(\mathbf{d}) = F_G(\mathbf{d}) + \mathrm{KL}[p_G(\mathbf{XY}|\mathbf{d}), p_R(\mathbf{XY}|\mathbf{d})].$$

noting that the first term depends only on *G*, so can be considered a constant as we now focus on dependence on *R*.

Let the symbol ∇_R stand for any derivative with respect to the recognition weights (i.e. for $\partial/\partial w^R{}_{kj}$, or $\partial/\partial v^R{}_{ji}$). Then:

$$\begin{aligned}
\nabla_R \widetilde{F}^R{}_G(\mathbf{d}) &= \nabla_R \ \mathrm{KL}[p_G(\mathbf{XY}|\mathbf{d}), p_R(\mathbf{XY}|\mathbf{d})] \\
&= \nabla_R \left[\textstyle\sum_{\mathbf{xy}} p_G(\mathbf{xy}|\mathbf{d}) \log p_G(\mathbf{xy}|\mathbf{d}) - \sum_{\mathbf{xy}} p_G(\mathbf{xy}|\mathbf{d}) \log p_R(\mathbf{xy}|\mathbf{d}) \right] \\
&= - \nabla_R \textstyle\sum_{\mathbf{xy}} p_G(\mathbf{xy}|\mathbf{d}) \log p_R(\mathbf{xy}|\mathbf{d}) \\
&= - \textstyle\sum_{\mathbf{xy}} p_G(\mathbf{xy}|\mathbf{d}) \nabla_R \log p_R(\mathbf{xy}|\mathbf{d}) \\
&= - \langle \nabla_R \log p_R(\mathbf{XY}|\mathbf{d}) \rangle_G \qquad (10.1)
\end{aligned}$$

Now our gradient descent learning algorithm will, given a pattern **d**, update the weights by the small step downhill $-\varepsilon\nabla_R \tilde{F}^R{}_G(\mathbf{d})$, where $\varepsilon > 0$ is small. How do we evaluate this? The right hand side just above says we just use a sample $\nabla_R \log p_R(\mathbf{xy}|\mathbf{d})$ where **x** and **y** are drawn from the *generative* distribution. But exactly what **d** do we use here? We defer this question until we have developed the update rule.

So now we turn to evaluating the gradient of $\log p_R(\mathbf{xy}|\mathbf{d})$, using the layered nature of R we introduced in section 7:

$$\begin{aligned} \nabla_R \log p_R(\mathbf{xy}|\mathbf{d}) &= \nabla_R \log p_R(\mathbf{y}|\mathbf{d})\, p_R(\mathbf{x}|\mathbf{y}) \\ &= \nabla_R \log p_R(\mathbf{y}|\mathbf{d}) + \nabla_R \log p_R(\mathbf{x}|\mathbf{y}) \end{aligned} \qquad (10.2)$$

We get the needed expressions for $\log p_R(\mathbf{y}|\mathbf{d})$ and $\log p_R(\mathbf{x}|\mathbf{y})$ from the last equations in section 7 for the three layers of weights from the top down:

$$\begin{aligned} \log p_R(\mathbf{x}\,|\,\mathbf{y}) &= \log \prod_k \xi_k^{\,x_k}\,(1-\xi_k)^{1-x_k} \qquad \text{where} \quad \xi_k = \sigma\left(\sum_{j=1}^{M} w^R_{kj} y_j + w^R{}_{k,M+1}\right) \\ &= \sum_k x_k \log \xi_k + \sum_k (1-x_k)\log(1-\xi_k) \end{aligned}$$

$$\begin{aligned} \log p_R(\mathbf{y}\,|\,\mathbf{d}) &= \log \prod_j \psi_j^{\,y_j}\,(1-\psi_j)^{1-y_j} \qquad \text{where} \quad \psi_j = \sigma\left(\sum_{i=1}^{N} v^R_{ji} d_i + v^R{}_{j,N+1}\right) \\ &= \sum_j y_j \log \psi_j + \sum_j (1-y_j)\log(1-\psi_j) \end{aligned}$$

The various connection weight derivatives that make up $\nabla_R \log p_R(\mathbf{xy}|\mathbf{d})$ are then straightforward to calculate. The algebra and elementary calculus follows the same pattern as in the previous section. For the middle-to-top recognition weights:

$$\frac{\partial}{\partial w^R_{kj}} \log p_R(\mathbf{x}\,|\,\mathbf{y}) = \begin{cases} (x_k - \xi_k)x_j, & j = 1,2,\ldots,M \\ x_k - \xi_k, & j = M+1 \end{cases}$$

$$\frac{\partial}{\partial w^R_{kj}} \log p_R(\mathbf{y}\,|\,\mathbf{d}) = 0$$

For the bottom-to-middle recognition weights:

$$\frac{\partial}{\partial v^R_{ji}} \log p_R(\mathbf{x}\,|\,\mathbf{y}) = 0$$

$$\frac{\partial}{\partial v^R_{ji}} \log p_R(\mathbf{y}\,|\,\mathbf{d}) = \begin{cases} (y_j - \psi_j)d_i, & i = 1,2,\ldots,N \\ y_j - \psi_j, & i = N+1 \end{cases}$$

Combining all this and collecting components in vector form, we have

$$\nabla_{\mathbf{W}^R} \log p_R(\mathbf{xy}|\mathbf{d}) = (\mathbf{x} - \boldsymbol{\xi})\,[\,\mathbf{y} \mid 1\,]^{\mathrm{T}}$$
$$\nabla_{\mathbf{V}^R} \log p_R(\mathbf{xy}|\mathbf{d}) = (\mathbf{y} - \boldsymbol{\psi})\,[\,\mathbf{d} \mid 1\,]^{\mathrm{T}}$$

This means gradient descent takes the form of very simple weight increments at each layer:

$$\mathbf{W}^R \mathrel{+}= \varepsilon\,(\mathbf{x} - \boldsymbol{\xi})\,[\,\mathbf{y} \mid 1\,]^{\mathrm{T}}$$
$$\mathbf{V}^R \mathrel{+}= \varepsilon\,(\mathbf{y} - \boldsymbol{\psi})\,[\,\mathbf{d} \mid 1\,]^{\mathrm{T}}.$$

As before, keep note of where the quantities on the right hand side came from: **x** and **y** come from the generative distribution; and $\boldsymbol{\xi}$ and $\boldsymbol{\psi}$ come from the recognition distribution given **x** and **y**. What about **d**? In the wake phase, we started at the beginning of the recognition causal chain $\mathbf{x} \leftarrow \mathbf{y} \leftarrow \mathbf{d}$, generating the explanation **xy** by sampling **d** from the world. By symmetry, in the sleep phase we start at the beginning of the generative causal chain $1 \rightarrow \mathbf{x} \rightarrow \mathbf{y} \rightarrow \mathbf{d}$ by sampling **x** from the generative bias input and feeding it forward to produce the pattern **d**.

So again we have a delta rule, nicely symmetric with the sleep phase. In summary:

SLEEP PHASE

// Initiate a dream !

$\mathbf{x}$ = SAMPLE($\boldsymbol{\sigma}$ ($\mathbf{b}^G$))

// Pass dream signal down through generation network

$\mathbf{y}$ = SAMPLE($\boldsymbol{\sigma}$ ($\mathbf{W}^G[\mathbf{x} \mid 1]^{\mathrm{T}}$))

$\mathbf{d}$ = SAMPLE($\boldsymbol{\sigma}$ ($\mathbf{V}^G[\mathbf{y} \mid 1]^{\mathrm{T}}$))

// Pass back up through recognition network, saving computed probabilities

$\boldsymbol{\psi} = \boldsymbol{\sigma}$ ($\mathbf{V}^R$ $[\mathbf{d} \mid 1\,]^{\mathrm{T}}$)

$\boldsymbol{\xi}$ = $\boldsymbol{\sigma}$ ($\mathbf{W}^R$ $[\mathbf{y} \mid 1\,]^{\mathrm{T}}$)

// Adjust recognition weights by delta rule

$\mathbf{V}^R \mathrel{+}= \varepsilon\,(\mathbf{y} - \boldsymbol{\psi})\,[\mathbf{d} \mid 1\,]^{\mathrm{T}}$

$\mathbf{W}^R \mathrel{+}= \varepsilon\,(\mathbf{x} - \boldsymbol{\xi})\,[\mathbf{y} \mid 1\,]^{\mathrm{T}}$

11. The Algorithm

The sections above have led us to a learning algorithm for Helmholtz machines in the following form. All we need to add is that it is customary to start all the weights at 0, so every neuron initially has a 50-50 chance of firing regardless of its input (since $\sigma(0) = \frac{1}{2}$).

$\mathbf{V}^G$, $\mathbf{W}^G$, $\mathbf{b}^G = \mathbf{0}$
$\mathbf{V}^R$, $\mathbf{W}^R = \mathbf{0}$
repeat
    wake phase to change $\mathbf{V}^G$,$\mathbf{W}^G$, $\mathbf{b}^G$
    sleep phase to change $\mathbf{V}^R$, $\mathbf{W}^R$
until
    KL[$p(\mathbf{D})$, $p_G(\mathbf{D})$] is sufficiently close to 0.

If we kick away some of the scaffolding we used to get to this point, the optimization problem begins to look quite simple itself. Equation (9.1) showed that the wake phase's minimization of the variational free energy just amounts to minimizing the expected energy, which means maximizing $\langle \log p_G(\mathbf{XYd}) \rangle_R$. Equation (10.1) showed that the sleep phase's minimization of an approximation to the variational free energy just amounts to maximizing $\langle \log p_R(\mathbf{XY}|\mathbf{d}) \rangle_G$.

In summary, we have the following bare-bones description:

Given a generative distribution $p_G(\mathbf{xyd})$ and a recognition distribution $p_R(\mathbf{xy}|\mathbf{d})$ with the following causal structure

$$1 \xrightarrow{G} \mathbf{x} \underset{R}{\overset{G}{\rightleftarrows}} \mathbf{y} \underset{R}{\overset{G}{\rightleftarrows}} \mathbf{d}$$

alternately maximize their expected "log likelihoods", where the expectation is taken against the *other* distribution:

$$\langle \log p_G(\mathbf{XYd}) \rangle_R \text{ and } \langle \log p_R(\mathbf{XY}|\mathbf{d}) \rangle_G$$

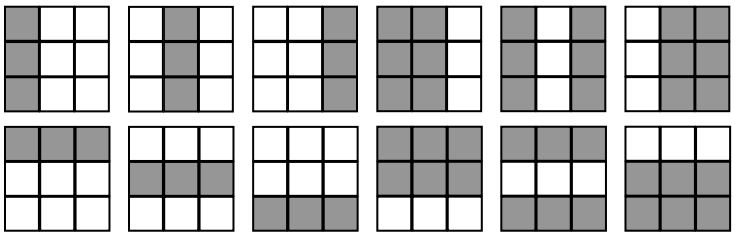
We have found that one easy way to implement such a stochastic causal chain is through a neural network, with connections running in opposite directions. And when the neurons use sigmoid activation functions the derivatives of log probabilities of outputs with respect to inputs always take on a simple delta rule form; this allows the wake-sleep algorithm to work via simple local weight updates.

12. A Demonstration

We conclude with a quick confirmation that the code above actually works. This is a variant of one example used in the Hinton-Dayan paper in *Science* [2]. The world consists of 3×3 images of vertical and horizontal bars. Vertical bars occur with twice the probability of horizontal bars, but within each category (horizontal or vertical) each of the possibilities occurs with the same probability.

By contrast, Hinton and Dayan used 4×4 images, but all nonzero probabilities were identical, so that the problem amounted to learning a binary classification. Here we set things up more generally, to show the Helmholtz machine trying to match a nontrivial probability distribution.

We take $\mathbf{d} \in \{0,1\}^9$, so there are 512 possible patterns. To get the distribution above, we assign probabilities as follows:



high frequency: $p(\mathbf{d}) = 2/18 = 0.111$ for each vertical bar pattern $\mathbf{d}$

low frequency: $p(\mathbf{d}) = 1/18 = 0.056$ for each horizontal bar pattern $\mathbf{d}$

with $p(\mathbf{d}) = 0$ for the other 500 patterns..

A Helmholtz machine in its initial state zero-weight state will generate all 512 patterns uniformly, so that $p_G(\mathbf{d}) = 1/512 \approx 0.002$ for every pattern $\mathbf{d} \in \{0,1\}^9$. As the learning algorithm proceeds, we expect $p_G(\mathbf{d})$ to approach the given $p(\mathbf{d})$.

We simulate a 1-6-9 Helmholtz machine (so $\mathbf{x} \in \{0,1\}^1$ and $\mathbf{y} \in \{0,1\}^6$) using the algorithm *exactly* as shown in the code in the previous sections. In the Hinton-Dayan 4×4 example the code was tweaked with some neural smithing techniques (e.g. initializing certain weights to specific nonzero values and adjusting some of the weight updates to keep certain weights positive). Our implementation is pure vanilla.

With learning rates set to $\varepsilon = 0.01$ for the top ($\mathbf{W}$) layer, and $\varepsilon = 0.15$ for the bottom ($\mathbf{V}$) layer, we run the algorithm for 60,000 iterations. (With a straightforward C++ implementation of the algorithm, this takes about ten seconds on a typical 2006 model desktop computer.) We then stop to evaluate the generative distribution, taking 100,000 samples to estimate probabilities.

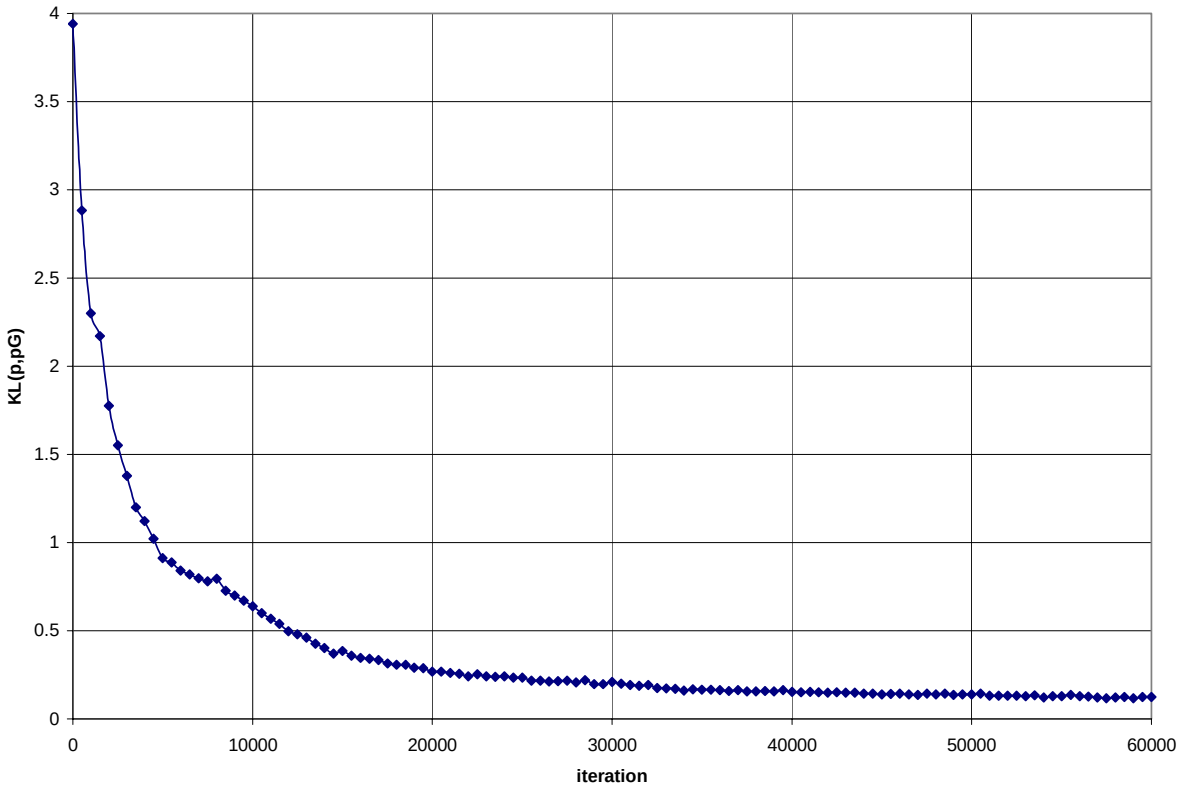
The following table lists all the patterns the machine generates with estimated probability $p_G(\mathbf{d}) \geq 0.0010$:

		Pattern **d**	World $p(\mathbf{d})$	Helmholtz Machine $p_G(\mathbf{d})$	
WORLD	**vertical bars**	[0 0 1 0 0 1 0 0 1]	0.1111	0.1075	88%
		[0 1 1 0 1 1 0 1 1]	0.1111	0.1070	
		[1 0 0 1 0 0 1 0 0]	0.1111	0.0999	
		[1 1 0 1 1 0 1 1 0]	0.1111	0.0981	
		[0 1 0 0 1 0 0 1 0]	0.1111	0.0955	
		[1 0 1 1 0 1 1 0 1]	0.1111	0.0911	
	horizontal bars	[0 0 0 1 1 1 1 1 1]	0.0556	0.0615	
		[1 1 1 0 0 0 1 1 1]	0.0556	0.0529	
		[0 0 0 1 1 1 0 0 0]	0.0556	0.0455	
		[1 1 1 1 1 1 0 0 0]	0.0556	0.0419	
		[0 0 0 0 0 0 1 1 1]	0.0556	0.0415	
		[1 1 1 0 0 0 0 0 0]	0.0556	0.0391	
other generated patterns		[0 0 0 0 1 0 1 1 1]	0	0.0412	
		[1 1 1 1 0 1 0 0 0]	0	0.0287	
		[1 1 1 1 0 0 0 0 0]	0	0.0105	
		[0 0 0 0 1 1 1 1 1]	0	0.0079	
		[0 0 1 0 1 1 0 0 1]	0	0.0012	
		[0 0 1 1 0 1 0 0 1]	0	0.0011	
		[0 0 0 1 1 1 1 0 1]	0	0.0011	
		[1 1 0 0 1 0 1 1 0]	0	0.0010	
		others	0	< 0.0010	

The machine has clearly captured much of the world's structure here: vertical bars appear with higher probability than horizontal bars, all patterns in the world are generated by the machine, and patterns not in the world occur with low probability and are mostly just a bit away from real patterns. Yet, as experience with these machines has shown (e.g. [6]), the machine hasn't quite captured the world in what we might judge to be the most natural way, as shown by the unreal pattern 000010111 with probability 0.0412. *El sueño de la razon produce monstruos.*

Recall from section 5 that we began our work by trying to minimize KL[$p(\mathbf{D})$, $p_G(\mathbf{D})$]. We can track the change in this quantity as the learning process proceeds[1]; this is shown in the plot below. The simulation stops to sample the generative distribution 10,000 times every 500 iterations and plots the KL divergence.

[1] Since our sampling will, early in the learning process, occasionally estimate $p_G(\mathbf{d}) = 0$ for some $\mathbf{d}$ with $p(\mathbf{d})>0$, we arbitrarily substitute a small value $p_G(\mathbf{d}) = 10^{-6}$ << 1/512 in place of zero so the KL divergence will not blow up. This is completely separate from the algorithm; it merely affects the display of results.



Since sleep-wake learning is based on steepest descent, it is amenable to many neural smithing techniques from the savvy optimizer's toolbox to improve performance further.

References

1. Russell, S. and P. Norvig. 2003. *Artificial Intelligence: A Modern Approach,* 2nd Ed. Prentice-Hall.

2. Hinton, G.E., P. Dayan, B.J. Frey and R.M. Neal. 1995. The wake-sleep algorithm for unsupervised neural networks. *Science 268*: 1158-1161.

3. Dayan, P., G.E. Hinton, R.M. Neal, and R.S. Zemel. 1995. The Helmholtz machine. *Neural Computation 7*:5,889–904.

4. Dayan, P. and L.F. Abbott. 2001. *Theoretical Neuroscience: Computational and Mathematical Modeling of Neural Systems.* MIT Press.

5. Dayan, P. 2003. Helmholtz machines and sleep-wake learning. In M.A. Arbib, Ed., *Handbook of Brain Theory and Neural Networks*, Second Edition. MIT Press.

6. Dayan, P. and G.E. Hinton. 1996. Varieties of Helmholtz machine. *Neural Networks* 9:8, 1385-1403.

7. Ikeda, S., S.-I. Amari and H. Nakahara. 1999. Convergence of the wake-sleep algorithm. In M.S. Kearns et al, Eds., *Advances in Neural Information Processing Systems* 11.