Chapter – 11

**Neural Networks for Machine Learning**

**Geoffrey Hinton**

with **Nitish Srivastava** & **Kevin Swersky**

**Hopfield Nets**

Lectures: Geoffrey Hinton

Hopfield Nets

Dealing with spurious minima

Hopfield nets with hidden units

Using stochastic units to improve search

How a Boltzmann machine models data

**11.1 Hopfield Nets**

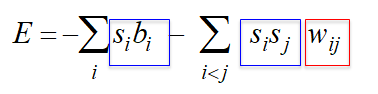
In this section, we'll discuss *Hopfield Nets*. Together with *Back Propagation*, these were one of the main reasons for the *resurgence of interest in neural networks* in the 1980s. *Hopfield networks* are beautifully simple devices that can be used for *storing memories* as *distributed patterns of activity*.

* **Hopfield Nets**

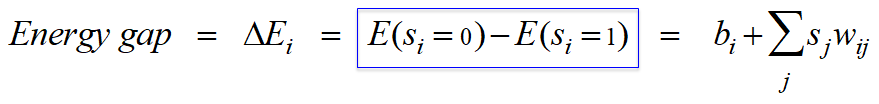
We are now going to learn about a different kind of model from a *Feed Forward Neural Net*. These are sometimes called *Energy Based* *Models* because their properties derive from a *Global Energy Function*.

* ***Hopfield net:*** A Hopfield net is one of the *simplest* kinds of *energy-based model*. It's composed of *binary threshold units* with *recurrent* connections between them.
* In general, if you have networks of *Non-Linear Units* with *recurrent connections*, they're very *hard* to *analyze*. They can behave in many different ways.
* They can *settle* to a *stable state*.
* They can *oscillate*.
* They follow *chaotic trajectories* that *cannot be predicted far* into the future. Which means that— unless you know their *starting state* with *infinite precision*, you *can't predict the state* they'll be in very far into the *future*.
* Fortunately, ***John Hopfield*** and various other groups, like ***Stephen Grossberg's*** *group*, realized that— if the *connections* are *symmetric*, there is a *Global Energy Function*.

* Each *binary configuration* of the whole network *has* an *energy*. Here a *binary configuration* is— an *assignment of binary values* to *each* *neuron* in the network. So, every *neuron* has a particular *binary value* in that *configuration*.
* The thing that Hopfield realized is that— if you set up the *right energy function* for *binary threshold decision rule*, is actually causing the *network*, to go *downhill* in *energy*, and if you *keep applying* that *rule*, it'll end up in a *energy minima*.
* The binary threshold decision rule causes the network to settle to a minimum of this energy function.
* **The energy function**
* Everything's controlled by the *energy function*. The *global energy* of a *configuration* is the *sum* of a number of *local contributions*, and the *main contributions* have the form of the *product* of *one connection weight*, with the *binary states of two neurons*.
* The global energy is the sum of many contributions.
* Each contribution depends on one connection weight and the binary states of two neurons.
* Following is the energy function. The minus signs means that— Energy is bad, so *low energy is good*.
* The notation i<j in the summation, means that the sum runs only over unique pairs of units, where i is always less than j. This *avoids* *double-counting* the same pair. For example if we just have just 3 units then the summation expands as:
* ***Weight & Activities:*** If you look at the main term here, it has a *weight* which is the *symmetric connection strength* between *two neurons*. And it has the *activities* of the two connected neurons.
* Here, is a *binary variable* that has values of 1 *or* 0 (or in another kind of Hopfield net, it has values of 1 or -1).
* ***Quadratic Term:*** The *quadratic term* (of ) involves the *states of two units*.
* ***The bias term:*** The *bias term* (of ) that only *involves* the *state* of *individual units*.



* The simple *Quadratic Energy Function* makes it possible for *each unit* to *compute locally* *how changing its state* will *change/affects* the *global energy*.
* ***Energy Gap:*** We first need to define the energy gap.
* The *energy gap* for a unit i is the *difference* in the *global energy* of the *whole configuration* depending on *whether* or *not* i is on. So, the *energy gap* can be actually defined as the difference between the *energy* when i is off and the *energy* when i is on i.e. .
* And that difference is what is just what is being computed by the *Binary Threshold Decision Rule*.
* So, if you look at the *equation for the energy* and you *differentiate* it w.r to the *state of the i-th unit*, (it's a funny thing to do because it's a binary variable). But, if you *differentiate* it, you'll see you get the *binary threshold decision rule*, but *without* the *minus sign* because that's for going *down hill* in energy.



|  |  |
| --- | --- |
| * The intuition is: * Energy: Lower energy = more "stable" state. * : Firing () *reduces* the *system’s energy* i.e. *neuron activates*. Because this means that is, the energy is lower when the neuron fires (). Since lower energy indicates a more stable state, activating the neuron is preferred. |  |

* : Staying *inactive* () is *better* i.e. *neuron remains off*. This implies the energy is not reduced (or is increased) by firing the neuron. Thus, keeping the neuron off () is more stable.

Binary Threshold Rule

* The *Binary Threshold Rule* is a *decision rule* used in neural networks (e.g., Hopfield networks and perceptrons) to update the state of a binary neuron () based on its total input.

is the *bias (or threshold)* for neuron i .

represents the *synaptic weight* between neuron i and neuron j. It is the weight of the connection from neuron j to i.

is the *state* (0 or 1) of connected *neighboring neuron j.*

* The neuron activates () if the weighted sum of inputs *exceeds a threshold* (biased by ).
* The neuron stays inactive () if the total input is too low.

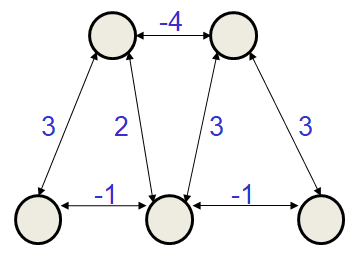
|  |  |  |
| --- | --- | --- |
|  | i.e. |  |

This rule allows *neurons* to make simple, *deterministic decisions* based on their *inputs* and *biases*, enabling networks to solve *classification tasks* or stabilize in *energy-minimizing configurations* (e.g., in associative memory models). It’s a foundational concept for understanding neural computation.

* **Settling to an energy minimum**

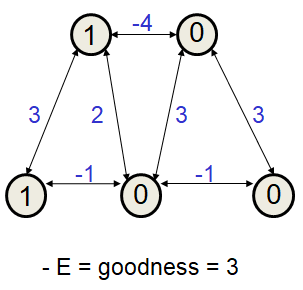
By following the *Binary Threshold Decision Rule*, a *Hopfield net* will go *downhill* in its *global energy*.

* One way to find an *energy minimum* in a *Hopfield net*— is to *start* from a *random state* and then *update the units* *one at a time* in *random* *order*.
* So, we're doing a *sequential update*.
* For *each unit* that you pick, you compute *whichever of its two states* gives the *lowest global energy* and you *put it* in *that state* *independent* of what *state* it was *previously* in.
* So we update each unit to whichever of its two states gives the lowest global energy.
* That's equivalent to saying you just used the *Binary Threshold Decision Rule* (i.e. use binary threshold units).
* Now, let's look at a little example for the following net:



Consider this net

* We'll start with a *random global state*. This was a *carefully selected* *random state*, and that has an *energy* of -3, or a goodness of 3 i.e. -E.
* It's easier to think about *negative energies* which are called *goodnesses*.



Start with "random global sate"

* There aren't any *biases* here. So to compute the *goodness*, you just look at *all pairs of units* that are *on* and *add* in the *weight* between them.

So to calculate goodness/global energy the equation—

Changes to below since there's no bias present:

The notation i<j in the summation, means that the sum *runs* only over *unique pairs* of units, where i is always less than j. This *avoids* *double-counting* the same pair.

* In this configuration, there's only *one pair of units* that's *active*. And that has a *weight of 3*, so we get a goodness of 3.
* Since the pairs containing 0 (state 0) results zero (e.g. on the left of the net notice ). So we just compute the only for the non-zero pairs of states. Then

Since there's only one pair (1,1) and it has weight 3.

* Now, let's *start probing* the *units*. Let's pick a unit at *random*—like that one in the *top right*—and ask: *What state should it be in, given the current states of all the other units?*
* So, if we look at *total input* to that, it gets an *input* of , so it gets a total input of . That's *below zero*, so we *turn it off*, i.e. it stays in the off state. Also we have only one pair (1,1), E=-(1×1×3) i.e.-E=3.

|  |  |
| --- | --- |
|  |  |
| Pick a unit at random | Since its below 0, we turn it off |

* Now let's probe another unit (in the bottom left). If we look at this unit, again, it gets a total input of , so it gets a *total input* of 3, so the *binary threshold decision rule* will make it turn on. And E=-(1×1×3) i.e.-E=3 since other pairs contains 0.

|  |  |
| --- | --- |
|  |  |
| Lets took another unit | Since its above 0 (3) we turn it on |

* Let's probe one more unit (bottom middle). This unit's more *interesting*. It's getting an input of that's a *total input* of 1. So, it will now turn on. *Previously* it was off. And so, when it *turns on*, the *global energy* *changes*.

|  |  |
| --- | --- |
|  |  |
| Let's take one more unit | This unit more interesting  (global energy changes after turn on) |

* Now notice, we have 3 pairs (1,1) of non-zero units with weights 3, 2, -1. Therefore E=-(1×1×3 + 1×1×2 + 1×1×(-1)) i.e.-E=4. So, we now have a global energy of -4, or a goodness of 4. And that's a *local energy minimum*.
* If we now *try* probing *any* of the *units*, we'll see that they *don't want to change* their *current state*.
* The net is now *settled* to a *minimum*.

|  |  |
| --- | --- |
| * **A deeper energy minimum**   However, the *minimum* it *settled* to is *not* the *deepest energy minimum* (considering previous example). It's just *one* of *two minima* that this net has.   * The *deepest* energy *minimum* is shown on the right, and it's when the *other triangle of units* that support each other is on (previously, all units on the left side were active; now, all units on the right side are active). * Calculating : Since, we now have 3 pairs (1,1) of non-zero units with weights 3, 3, -1we get E=-(1×1×3 + 1×1×3 + 1×1×(-1)) i.e.-E=5 that has a goodness of 5, that's a slightly better energy minima. |  |

* If you look at that net, you can see the net is composed of *two triangles* in which the *units mostly support each other*, although there's a bit of disagreement at the bottom.
* And *each* of those *triangles* mostly *hates* the *other triangle* *via* that *connection* at the *top*.
* The *triangle* on the *left* *differs* from the one on the *right* by having a *weight* of 2, where the other one has a *weight* of 3. So, the *triangle* on the *right* will give you the *deepest minimum*.
* So *turning on* the units in the *triangle on the right* gives the deepest minimum.
* **Why do the decisions need to be sequential?**

Now, if you ask— why did the ***decisions*** need to be ***sequential*** in the Hopfield net? The problem is that—

* If units make *simultaneous decisions*, they could each think they’re *reducing the energy* but actually the *energy could go up*.
* With *simultaneous parallel updating*, we can get *oscillations* which always have a *period of 2*.

|  |  |
| --- | --- |
| * ***Example:*** Consider the following little network where the units have biases of +5, and a weight between them of -100. * When *both units* are *off*, the *next parallel step*, if we update them both at the same time, *will turn both units on* (because they each think they can improve things via the *bias* term). * But, as soon as you do that, you get this -100, i.e. so you've actually made things *much worse*. So then, in the next parallel step, both units will *turn off* again. | At the next parallel step, both units will turn on.  But this has very high energy, so then they will both turn off again. |

* If we do the *updates in parallel* but with *random timing*, in other words— we *don't wait* for one update to *communicate* the *state to* *everybody* before we consider another update.
* But we do *wait* for *random lengths of time* between doing *updates* of a *given unit*. Then, those *random timings* will often *destroy* these *bi-phasic oscillations*.
* i.e. if the updates occur in parallel but with random timing, the oscillations are usually destroyed.

That means that the idea that— the *updates have to be sequential* isn't quite as bad as it seems from a biological perspective.

* **Bi-phasic oscillations** are *repeating patterns* with *two phases* in each cycle. The term "biphasic" itself means "having two phases". These phases typically involve alternating states, such as:
* ***High and Low (Positive and Negative Phases):*** The signal switches between *excitatory* (high) and *inhibitory* (low) states.
* ***On and Off:*** The system *alternates* between being turned *on* (active) and *off* ( inactive).
* **A neat way to make use of this type of computation**

The 1982 paper titled *"Neural Networks and Physical Systems with Emergent Collective Computational Abilities"* by John J. Hopfield is the original publication that introduced what is now known as the *Hopfield network*. In this seminal work, Hopfield demonstrated how *neural networks with symmetric connections* could *store* and *retrieve* ***memories*** by associating them with *energy minima in the system*.

* What Hopfield suggested was that— we could make *use* of this kind of *energy based model* that *settles to a minimum of its energy* for *storing memories*.
* ***Hopfield (1982) proposed that memories could be energy minima of a neural net:***
* So, we had a very influential paper in 1982 that proposed that— *memories* could be *energy minima* of a *neural net with symmetric weights*.
* Then the *binary threshold decision rule* can take *partial memories*, and *clean them up* into *full memories*. So, the memory could be *corrupted* by *part of it being wrong*, or part of it could just be *undecided*, and we can use the net, to *fill out the memory*.
* i.e. the *Binary Threshold Decision Rule* can then be used to “clean up” *incomplete* or *corrupted* memories.
* ***Crystal analogy:*** The idea of *memories* as *energy minima* goes back a long way. The first example we know of is in a book called *“Principles of Literary Criticism” by I. A. Richards in 1924*, where he proposes that *memories* are like a *large crystal* that can *sit on different faces*.
* Using *energy minima* to *represent memories* gives a *content-addressable memory*, as Hopfield realized.
* So we can *access an item* just by knowing *part of its content*.
* So if we know a *few properties* of *something* that'll *set the states* of some of the neurons in the net. And if we've put the *other neurons* in *random states* and now go around applying the *binary threshold rule*— with a bit of luck, we'll fill out that memory *to be some stored item* that we know about.
* When Hopfield nets were proposed in *1982*, that was a very interesting property (*16 BG*, i.e. 16 year before Google). 1982 was 16 years before Google, now that we have Google, we regard this as perfectly obvious.
* ***Robust against hardware damage:*** Another property of Hopfield nets is biologically interesting is— they are *robust against hardware damage*. You could *remove* a *few* of the *units in the net* and unlike the *central processor of your computer*, *everything* will still *work fine*.
* ***Reconstructing a Dinosaur from a few bones:*** Psychologists have a nice analogy for this kind of memory— It's like "*Reconstructing* a *Dinosaur* from just a *few* of its *bones*" because you *know* something about “*how the bones are meant to fit together”*.
* So, the *weights* in the network *give you information* about how *states* of neurons *fit together*. And now, *given* the *state of a few neurons*, we can *fill out the whole state* to *recover a whole memory*.

Clarification on Crystal metaphor

That metaphor — *memories being like a crystal that can rest on different faces* — comes from I.A. Richards, and it's not meant to be literally true, but rather a conceptual analogy.

Let's connect the *crystal metaphor* from I.A. Richards to the *energy landscape in Hopfield networks*, which offer a modern scientific way to model memory.

* Imagine a *crystal* with *many flat faces*.
* You can place it on *any* one of these *faces*, and it will stay stable — that’s what crystals do.
* *Each face* represents a *different stable configuration*.
* ***In memory terms:*** each stable face = a particular memory the brain can settle into.
* Hopfield Networks: Energy Landscape Hopfield networks are a type of *recurrent neural network* where neurons are connected with symmetric weights.
* The network defines an energy function like this:

(If biases are zero, it simplifies to the *sum over pairwise weights*.)

* The *network updates neuron states* to *reduce energy* — like a ball rolling downhill.
* Eventually, the network reaches a *minimum energy state*, which represents a *stored memory*.
* Connecting the Two:

|  |  |
| --- | --- |
| **Crystal Metaphor** | **Hopfield Network** |
| Crystal has **stable faces** | Network has **stable energy minima** |
| Crystal **settles** on a **face** | Network **settles** into a **memory** |
| Each face = one configuration | Each energy well = one memory |
| Jumping between faces is hard | Network resists changing memories |

* ***Memory Completion:*** Hopfield networks can also *complete partial memories*, just like a *crystal* might *“tilt” slightly* and then *snap flat* onto the *closest stable face*.
* If you give the network a noisy/incomplete version of a memory:
  + It will settle into the closest energy minimum.
  + That’s like the crystal snapping to a stable face.

Hopfield nets are recurrent?

Yes. *Hopfield networks are recurrent* because their *neurons are connected in a loop* — *outputs* can *feed back* as *inputs*. This *feedback* lets the network *evolve over time* and *settle* into *stable memory states*.

* ***Recurrent connections:*** Neurons influence each other in cycles (*unlike* one-way *feedforward nets*). Neurons are interconnected in a *feedback loop* (outputs can circle back as inputs).
* *Unlike feedforward networks*, Hopfield networks have *cyclic connections*, allowing them to *model* *memory and state* *evolution* over time.
* ***Symmetric weights:*** The connection from neuron i to j is equal to j to i i.e. . This *symmetry* ensures the network *always settles* into a *stable state*.
* This *symmetry guarantees* that the network’s *dynamics converge* to a stable equilibrium (unlike asymmetric recurrent networks, which may oscillate chaotically).
* *Symmetry* is critical for *stability* but implies *limitations* (e.g., it *can’t model directed sequences* like LSTMs).
* ***Energy function:*** The network uses an *energy function* that always *decreases*, leading it to *a local minimum* — which represents a *memory*.
* The *symmetric weights* enable the network to have an *energy function* that decreases over time.
* The network *settles* into a *local minimum* of this energy, which *corresponds* to a stored *memory pattern*.
* Think of it like a *ball rolling downhill into a valley* (memory state). Once it’s there, it stays.

The original Hopfield network (1982) uses binary neurons (states 0 or 1, or sometimes -1/1). Later variants (e.g., continuous Hopfield networks) allow analog-valued neurons. Hopfield networks are indeed *recurrent networks* with *symmetric weights*, making them a foundational model for *associative memory* and *energy-based learning*.

* **Storing memories in a Hopfield net**

The storage rule for memories in the Hopfield net is very simple. The idea is, if we use activities of 1 and -1, that we can *store* a *binary state vector* by just *incrementing* the *weights* between any two units by the *product of their activities*.

* So, it's a very simple rule that is not error-driven (which is both its strength and its weakness):

|  |  |
| --- | --- |
| * One nice thing about this rule is that— you just *go through* the data *once* and you're done. So, it really is the *genuine online rule*. * That's because it's *not error driven*. You're *not comparing* what you would have *predicted* with what the *right answer* is, and then making small adjustments. |  |

* The fact that it's *not* an *error correction rule* is both it's *strength* and it's *weakness*.
* It means— it can be *online*, but as we'll see later, it also means it's *not* a very *efficient* way to *store* things.
* We can also have biases, and as usual, we *treat the biases* as *weights* from a *permanently on* unit.

|  |  |
| --- | --- |
| * If you want to use states of 0 and 1 for units, (which is what we'll use later), the update rule is only *slightly more complicated*. |  |

**11.2 Dealing with Spurious Minima**

In this section, we're going to talk about the *Storage Capacity* of *Hopfield Nets*. Their *ability to store a lot of memories* is *limited* by what are called *spurious memories*.

These occur when two *nearby energy minima combine* to make a *new minimum* in the *wrong place*.

Attempts to *remove these spurious minima* eventually led to a very *interesting way of doing learning* in things considerably more *complicated than a basic Hopfield net*.

At the end of the video, we'll also talk about a curious *historical rediscovery* where the physicist trying to *increase the capacity* of Hopfield Nets, rediscovered the *Perceptron Convergence Procedure*.

* **The storage capacity of a Hopfield net**

After Hopfield introduced *Hopfield nets* as *memory storage devices*— the field (researchers and scientists) became obsessed with their *storage capacity*.

* Using the *Hopfield's Storage Rule* for a fully connected network with N units, the capacity is approximately 0.15N memories.
* i.e, if you have N binary threshold units, the number of memories you can store is about 0.15N *before* memories start *getting confused* with one another. (Canadians read 0 as "naught" or "nought").
* So that's the number you can store and still hope to retrieve them sensibly.
* Each *memory* is a *random configuration* of the N units, so it has *N bits of information* in it. And so, the *total information* being stored, in a Hopfield net is about bits.
* This doesn't make *efficient use of the bits* that are required to store the weights. In other words, if you look at how many bits the computer is using to store the weights, it's using well over bits to store the weights.
* And therefore, this kind of *distributed memory* and *local energy minima* is not making efficient use of the bits in the computer.
* We can analyze how many bits we should be able to store— if we were making *efficient use of the bits* in the computer.
* There are weights and biases in the net.
* And after storing memories, each *connection weight* has an integer value in the range . That's because we *increase* it by 1 or *decrease* it by 1 each time we store a memory, assuming that we used states of -1 and 1.
* Now, of course, not all values will be *equiprobable*, so we could compress that information.
* But ignoring that, the *number bits* it would take us to store a *connection rate* in a naive way is: , because that's the number of *alternative connection rates* and that's a log to the base 2.
* *Bits required to store the weights and biases:* And so, the *total* number of *bits of computer memory* that we use is of the order of—

|  |  |
| --- | --- |
| * Notice that, that scales logarithmically with . Whereas, if you store things in the way that Hopfield suggests, you get this constants 0.15 instead of something this *scales logarithmically*. * So, we're not so worried about the fact that the *constant* is a lot *less than 2*, what we're worried about is this ***logarithmic scaling***. That shows we ought to be able to do something better. |  |

* **Spurious minima limit capacity**

If we ask— *What limits the capacity of a Hopfield net? What is it that causes it to break down?* Then, its merging of *energy minima*.

|  |  |
| --- | --- |
| * Each time we *memorize* a *binary configuration*, we hope that we'll create a *new energy minima*. * So, we might have a *state space* for *all the states of the net* being *depicted horizontally* here, and the *energy* being *depicted vertically*. * And, we might have one energy minimum for the *blue pattern* and another for the *green pattern*. |  |

|  |  |
| --- | --- |
| * But, if those two patterns are nearby (two nearby minima), what will happen is— we won't get two *separate minima*. They'll *merge* to create *one minimum* at an intermediate location. * That means, we *can't distinguish* those two *separate memories*, and indeed we'll recall something— that's a *blend of memories* rather than the *individual memories*. * That's what *limits* the *capacity of a Hopfield net*, that kind of *merging* of *nearby minima*. |  |

* NOTE: The picture we created above is a major *misrepresentation*. The *states* of a Hopfield net are actually the *corners of a hypercube*, and it's not very good to show, the corners of a hyper cube, as if they were *continuous one-dimensional horizontal space*.
* i.e. the *state space* is the *corners of a hypercube*. Showing it as a *1-D continuous space* is a misrepresentation.
* *"Depict"* means to *show or represent something*, especially in a picture, diagram, words, or another form of illustration.
* **Avoiding "spurious minima" by unlearning**

One very interesting idea that came out of thinking about how to improve the capacity of the Hopfield net is— *the idea of unlearning*. This was first suggested by *Hopfield, Feinstein* and *Palmer*, who suggested the following strategies:

* Let the net *settle* from a *random initial state*, and then do *unlearning*, i.e. whatever *binary state* it *settles* to, you apply *opposite* of the *storage rule*.
* We can see that with the previous example, that *red merge minimum*—

If we let the net settle there and did some *unlearning* on that *merge minimum*, we'd *get back* the two *separate minima*. Because we'd *pull* *up* that *red point*.

* So, by getting rid of *deep spurious minima*, we can actually *increase* the *memory capacity*.
* *Hopfield, Feinstein* and *Palmer* showed that this actually *worked*, but they *didn't have a good analysis* of what was really *going* *on*.
* Francis Crick, one of the discovers of the structure of DNA, and Graeme Mitchison, proposed— "*that unlearning*" might be what's going on during **REM sleep**, i.e. *Rapid Eye Movement sleep*.
* Means *unlearning* as a model of what *dreams are* for.
* ***Dream REM and unlearning:*** The idea was that *during the day, you store lots of things*, and you'll get *spurious minima*. Then at *night*, you *put the network in a random state*, you *settle to a minimum*, and you *unlearn* what you settled to.
* And that actually *explains* a big *puzzle*. This is a puzzle that doesn't seem to puzzle most *people that study sleep* but it ought to. Each night, you go to *sleep* and you *dream for several hours*. When you wake up in the *morning*, those dreams are *all gone*, you don’t remember them (unless you wake up during the dream).
* Well, they're not quite all gone. The *dream* you had *just before* you *woke up*, you can get into *short term memory* and you'll *remember* it *for a while*.
* And if you *think* about it more, you might *remember* it for a *long time*.
* But, we know perfectly well that— if we'd *woken* you up at other times in the *night*, you'd have been *having other dreams*, and in the morning their just not there.
* So, it looks like you're simply *not storing* what you're *dreaming about*, and the question is, why? In fact, why do you bother to dream at all?
* Dreaming is paradoxical and that the *state of your brain when you're dreaming* looks extremely like the *state of your brain when you're awake*, except that it's *not* being *driven* by *real input*.
* It's being driven by a *relay station* just *after the real input* called the *thalamus*.
* So the *Crick and Mitchison's theory* at least explains, functionally, what the *point of dreams* is— is to *get rid* of the *spurious minima*.
* ***How much unlearning should we do:*** But, there's another *problem with unlearning*, which is more *mathematical problem*, which is— *How much unlearning should we do?*
* A real solution to that problem will be to show that— *unlearning is part of the process of fitting a model to data*. And, if you do *maximum likelihood fitting* of that model, then *unlearning* will automatically come out and *fit into the model*.
* And also, you'll know exactly how much unlearning to do.
* So, what we're going to try and do is— *derive unlearning as the right way* to *minimize a cost function*. Where the *cost function* is, how well your neural net *models the data* that you *saw during the day*.
* Can we derive unlearning as the right way to minimize some cost function?

More explanations

Above discussion associated with *Crick and Mitchison’s "reverse learning" hypothesis (1983)* and further explored by researchers like Geoffrey Hinton. It describes a computational theory of sleep and dreaming, inspired by earlier neural network models — particularly Hopfield networks (1982) and Boltzmann machines (1980), which use energy minima to store patterns.

* In *Hopfield networks* (introduced by John Hopfield in 1982), memories are stored as *stable energy minima*. However, the original Hopfield model had issues with *spurious minima* — false memory patterns that aren't actual stored memories.
* Crick and Mitchison (1983) proposed that during sleep, the brain might "unlearn" these spurious memories, similar to cleaning out unwanted minima in neural networks.
* Geoffrey Hinton later expanded these ideas through his work on *Boltzmann machines* (mid-1980s), which also use stochastic units and energy-based learning. Although his later *wake-sleep algorithm* (early 1990s) focused more on learning useful representations rather than directly modeling dreaming, it continued the broader idea that brain-like systems could benefit from alternating between learning and "self-generated" phases.
* **Daytime Learning ("Storing"):** During the day, the brain (or a neural network) *learns* and *stores memories*. However, this process creates *spurious minima*—*false* or *noisy memory patterns* that don’t represent real experiences.
* **Nighttime "Unlearning" ("Dreaming"):** At night, the brain enters a "random" state (akin to dreaming— randomizing the state). It settles into a memory pattern (a "minimum" in energy terms) and unlearns that pattern. This removes *spurious* or *unimportant memories*, preventing overload (it unlearns those spurious patterns).
* This process is likened to dreaming, where most dreams (spurious patterns) are forgotten, except the last one before waking, which might enter short-term memory.
* "Unlearning" **resembles** *contrastive divergence* (used in Boltzmann machines), where the network "forgets" spurious states by adjusting weights away from them.
* ***Unlearning***, as proposed by ***Crick and Mitchison***, is not quite the same as *contrastive divergence* but shares the idea of moving away from unwanted configurations.
* ***Boltzmann machines***, however, do use energy-based learning and have a more formal mechanism (like *contrastive divergence*) for reconciling desired and sampled states—a deeper model that aligns with these ideas.
* Most *dreams* are *forgotten* because they reflect the brain’s process of *pruning spurious patterns*. Only the dream just before waking might enter *short-term memory*, as it’s the *last "unlearning" step* interrupted by waking up.

Dreaming may help the brain ***discard*** unhelpful or ***spurious memory traces***, acting as a *mental housekeeping process* that keeps the memory system clean and efficient. Rather than being meaningful narratives, dreams may simply reflect the *"mental noise"* generated during this *neural optimization*.

Geoffrey Hinton worked on **Boltzmann machines**, which use stochastic units and a sleep phase for learning. The **"wake-sleep" algorithm** involves phases where the network learns from data (wake) and then generates its own data to adjust weights (sleep).

* **Boltzmann Machines**

Hinton co-developed Boltzmann Machines with Terry Sejnowski in the 1980s.

* These models introduced *stochastic (random) binary units* and used concepts from *statistical mechanics* (like energy functions).
* Later, Hinton proposed the **Restricted Boltzmann Machine (RBM)** — a simplified version that made training practical using **Contrastive Divergence**.
* **Wake-Sleep Algorithm**

In 1995, Hinton, along with Peter Dayan and Zoubin Ghahramani, introduced the **Wake-Sleep algorithm** for training **Helmholtz Machines**.

* This algorithm is considered a precursor to modern variational learning techniques (like **Variational Autoencoders**, or **VAEs**).
* It laid the groundwork for *generative models* that learn both to *encode* and *decode*.
* **Increasing the capacity of a Hopfield net**

Before we go further, let's talk a little bit about the ways physicists discovered to *increase* the *capacity* of the Hopfield net. As we said, this was a major obsession within the field.

* Physicists really love the idea that— *math* they already know might *explain* how the *brain works*. (That means, post doctoral fellows in physics who can't get a job in physics might be able to get a job in neuroscience.)
* So, there are a very large number of papers were published in physics journals about Hopfield nets and their storage capacity.
* Eventually, Elizabeth Gardiner figured out that there was a much better storage rule that uses the *full capacity of the weights*.
* Instead of trying to *store* vectors in *one shot*, what we're going to do is— we're going to *cycle through* the *training set* many times. So, we *lose* our nice *online property* that you only have to *go through the data once*. But in return, we're going to *gain*, more *efficient storage*.
* What we going to do is— we going to use the *Perceptron Convergence Procedure* to *train each unit* to have the *correct state* given the *states of all the other units* in that *global vector* that we want to store.
* ***The procedure:***
* Take your network and *put* it into the *memory state* you want to *store*.
* And then you take each unit separately.
* And ask: "Would this unit *adopt* the *desired state* (state I want for it), *given* the *states* of all the *other units*?"
  + If *yes*, leave its *incoming weights unchanged*.
  + If *no*, *adjust* its *incoming weights* according to the *Perceptron Convergence Procedure*.

Notice, these would be integer changes to the weights.

* You may need to repeat this process several times.
* And of course, if you give it *too many memories*, this *won't converge*.
* You only get convergence with a *Perceptron Convergence Procedure* *if there exists a set of weights* that can solve the problem.
* Assuming a solution exists, this is a much more efficient way to store memories in a Hopfield net compared to the basic method.
* ***Pseudo-likelihood:*** This technique is also being developed in another field— *Statistics*. And statisticians call the technique "pseudo-likelihood". The idea is— to *get one thing right* given *all the other things*.
* So, with *high dimensional data*, if you want to build a model of it, the idea is— you build a model that *tries to get* *the value on one dimension right* given *the values on all the other dimensions*.
* The *main difference* between the *Perceptron Convergence Procedure* as is normally used and *Pseudo-Likelihood* is that— in the Hopfield net, the *weights are symmetric*. So, we have to get *two sets of gradients* for *each weight* and *average them*.
* But, apart from that, the way to use the full capacity of a Hopfield net is to use the *Perceptron Convergence Procedure* and to go through the data several times.

**11.3 Hopfield nets with hidden units**

In this section, we're going to explain a very *different way of using* *Hopfield's energy function*. We add some *hidden units* to the *network*. What we are trying to do is— make the *states* of those hidden units *represent* an *interpretation of the perception input* that's shown on the *visible units*.

* So, the idea is that the *weights* on between units represent *constraints* on *good interpretations*.
* And by *finding* a *low energy state*, we find a good *interpretation* of the *input data*.
* **A different computational role for Hopfield nets**

Hopfield nets combine two ideas—

* The idea of that you can find a *local energy minimum* by using a network of *symmetrically connected binary threshold units*, and
* The idea that these *local energy minima* might *correspond to memories*.

|  |  |
| --- | --- |
| * There's a *different way* of using the ability to find *local minima*. *Instead* of using the net to *store memories*, we can use it to *construct* *interpretations* of the *sensory input*. So, the idea is that— * We have the *input represented* by some *visible units*. * And we *construct* an *interpretation of that input* in the *set of hidden units*. So, the *interpretation or explanation of the input* is going to be a *binary configuration* over the *hidden units*. * i.e. the *interpretation* is represented by the *states* of the *hidden units*. |  |

* *Badness of the interpretation is represented by the energy:* The *energy* of the *whole system* will *represent* the *badness* of that *interpretation*.
* So, to get *good interpretations* according to our current model of the world (which is in the energy function)— we need to find *low energy states* of the *hidden units*, *given*, the *input represented* by the *visible units*.
* **What can we infer about 3-D edges from 2-D lines in an image?**

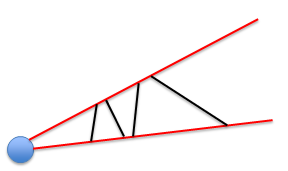
Let's see an example of this to make the above idea clearer, in order to give the example, we need to go into a little bit of detail, about—

* *What we can infer*, when we see a *2-D line* in an image.
* What does that tell you about the *3-dimensional world*?

|  |  |
| --- | --- |
| * A *2-D line* in an *image* could have been *caused* by many *different 3D edges* in the world. Consider the following images— * *Lines of sight:* If this *blue dot* is your *eyeball* and the *red lines* are two lines of sight coming from the center of your eyeball. Then *the black line* is a possible *3-D edge* that would lead to a *2D line* on your *retina*. * Here's another 3-D edge that would *lead to exactly the same thing* on your *retina*. Also another 2 lines. |  |

|  |  |
| --- | --- |
|  |  |

* If we assume it’s a *straight 3D edge*, the *information* that has been *lost in the image* is— the *3D depth of each end* of the 2D line.
* In the above figure, all of these *different 3-D edges* have exactly the *same appearance* in the image. That's because we've *lost* the *information* about *how far away the ends* of the line are along that line of sight.
* We know the *end* is somewhere along the *line of sight* but we *don't know* the *depth*.
* So, if we assume that— a *straight 3-D edge in the world* is the cause of a *straight 2-D line in the image*, then we've *lost two degrees of freedom* of that *3-D edge*— its *depth* at each end.
* So there is a whole *family of 3-D edges* that all correspond to the *same 2-D line*.



* You can only see *one* of these 3-D edges *at a time* because they all *get in the way of each other (occlude* *they one another)*.
* So now, we're in a *position* to see a *little example* of what you might be able to do, if you can use the fact— that you can find *low energy states* of a *network of binary units*, to help you find *interpretations of sensory input*.

Clarification

* ***2D Line in an Image:*** You see a straight line on a flat image (like on a screen or retina). That line looks like the edge of something 3D in the real world.
* ***Multiple Possible 3D Sources:*** Many *different 3D edges* (lines in space) could create the *same 2D image line*. Imagine a blue dot as your eyeball. A red line is your line of sight.
* The black line is a possible edge in the world that aligns with your sightline.

**Depth Ambiguity:**

You don't know how far away the ends of the line are — only their direction.

**That means the same 2D line could come from:**

A short edge close to your eye,

Or a long edge far away, as long as it lines up the same way.

**Degrees of Freedom Lost:**

In 3D, a straight edge has 4 degrees of freedom (e.g., positions of both endpoints).

From a 2D image, you lose 2 of those — the depth (distance) of each end.

**Family of Possible Edges:**

Because of that depth uncertainty, there's a whole set of 3D edges that all look like the same 2D line.

But you can only see one at a time in reality because they'd physically block each other.

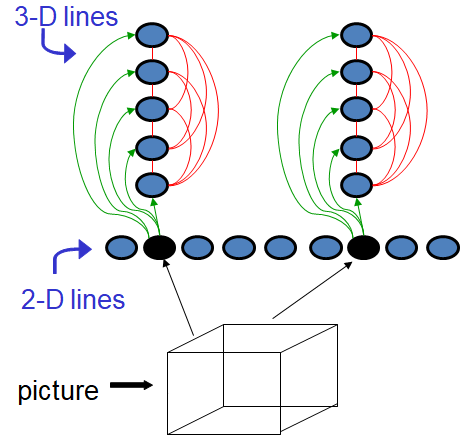
**So Why Is This Important?**

Hinton is about to show how neural networks (specifically ones with binary units and energy minimization) could help infer the most likely 3D structure from ambiguous 2D inputs — even when multiple interpretations are possible.

|  |  |
| --- | --- |
| * **An example: Interpreting a line drawing**   So, here's the example. We *imagine* we see a *line drawing*, and we want to *interpret* it as a *three-dimensional thing*.   * The data we have— is a *bunch of 2-D lines* like the lines shown in the following *picture (the cube)*, for each possible *line*, we will have *set* aside a *neuron* (don't worry for now about the fact that— it will require too many neurons). * So, for *every possible 2-D line*, we have *neuron*. |  |

* In any one picture, only a *few* of the *possible lines* will be *present*. And so, we'll *activate* just a *few* of those *neurons*. Notice the *two edges* in that picture, *activating* two of the *neurons*. And those are neurons that *represent 2-D lines*. They're the *data*.
* Use one *"2-D line"* unit for *each possible line* in the picture.
* Any particular picture will *only activate* a very *small subset* of the line units.
* Now, what we're going to do is— *have* a whole bunch of *3-D line units*, one for each *possible 3-D line* or *3-D edge* (use one "3-D line" unit for each possible 3-D line in the scene). So, *each* of the *2-D line units* could be the *projection* of many *different possible 3-D lines*.
* *Make these 3-D lines compete:* Therefore we need to make the *2-D line unit* *excite* all those *3-D lines*, but we also need to make them all *compete with one another*, because you can only see *one* of them *at a time*.

|  |  |
| --- | --- |
| * Following is an example where we have a *stack of 3-D line units*. The *green* connections are *excitatory connections* coming from the *2-D line unit*, all of them with equal weights, *indicating*: *if this 2-D line unit is active, it will try to activate all those 3-D line units*.   (***Excitatory connections*** are links between neurons where activity in one helps turn on the other. In simple terms: If Neuron A is active and it's connected to Neuron B with an excitatory link, Neuron B is more likely to turn on too.)   * But in addition, we need *competition* between those so that *only one* of them will *turn on*, and that's what the *red lines* represent. And we do that for *each 2-D line unit*. Notice, we do this for the *2-D line units* that happen to be *active at present*.   Don't worry about the fact that this would need far too many units. |  |

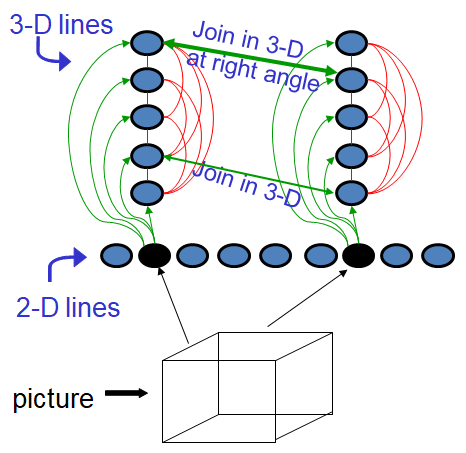


|  |  |
| --- | --- |
| * Now, the story is not quite complete. We've now *wired into* the neural network the *information about projection* that we discussed previously (projection of 3D edge on 2D line). * i.e. The *neural network* in those *green* and *red connections* understands that *each 2-D line* can correspond to *many different 3-D edges*, but *only one* of them should be *present* at a time.   *(wired into = explicitly designed the network's connections).*   * *Make 3D lines support each other if they join in 3D:* We already know a lot about *how 3D edges connect*. For example, when we see *two 2D lines* meet in an image, we think it's almost certain that they correspond to *3D edges* that share the *same depth* at the *point* where the lines *connect*. |  |

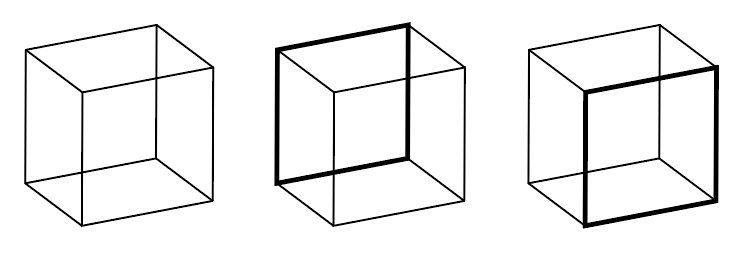
* In the above figure, let's suppose that the *two 3D edges* we've *joined* *correspond* to having the *same depth* at the point where the *two 2D lines* meet.
* That means they should *support each other*.
* However, it doesn't have to be that way. You could be *viewing* from an *unusual angle* where one line ends at a *different depth than the other*, yet they happen to coincide on your retina. But that's very unlikely.

So, we're going to need to use the fact that we expect *2D lines* that *coincide* in the *image* to *correspond* to *3D edges* that *agree on the depth* of that *point*. And, we'll put in a lot of connections like that.

* But there's an even stronger fact we can use, which is that— in our manufactured world, we expect that quite often, *3D edges will join in a right angle*.
* So, for two particular *3D edges* that happen to *agree in depth* and join at a *right angle*, we'll put in a particularly *strong connection* (indicated that by a thicker green line in the figure).



* So, by putting in lots of connections like above, we can indicate how we expect *3D edges* to go *together* to form a *coherent 3D object*.
* And now, we have a *network* that *contains information* about how *edges go together* in the *world* and about how *edges project* to cause lines *in the image*.
* So, if we give that network an image, it should be able to come up with an *interpretation of the image*. And for the image in our case, there's two quite *different interpretations*.
* Necker cube: It's called a *Necker cube*, and if you look at it long enough, it will *flip in depth* on you.
* And this network would have two pretty much *equally deep energy minima* that correspond to those *two interpretations* of the *Necker cube*. Following is an illustration of *Necker cube*.



Remember, this is all just a analogy so you understand the idea of using *low energy states* as *interpretations of perceptual data*. To actually build a proper model of what happens when the "Necker cube flips" will be a lot more complicated than this.

|  |  |
| --- | --- |
| * **Two difficult computational issues**   So, if we decide we're going to use low energy states to represent good perceptual interpretations, then we have two issues—   * *The first is to do with search (we'll discuss it in the next):* The search question is— *How do we avoid the hidden units getting trapped in poor local minima of the energy function?* * The poor minima represent interpretations that are sub-optimal, given our current model and the weights of the network. * Can we do anything better than simply going downhill in energy from some random starting state? |  |

* *The second issue, which seems even more difficult, is:* *How do we learn the weights on the connections between the hidden units, and between the visible and hidden units?* (we'll discuss in following chapter)
* Is there some *simple learning algorithm* for *adjusting* all those *weights* so that we get *sensible perceptual interpretations*?
* And notice here— we haven't got a supervisor anywhere.
* We're *just showing it input* and we would like it to *construct patterns of activity* in the *hidden units* that represent *sensible* *interpretations*. This seems like a rather tall order.

More explanation

Here’s a much simpler, step-by-step explanation of *Hinton’s Necker‐cube example*—showing how a *network* of *2D* and *3D “line” units* can settle into *coherent 3D interpretations*:

**Necker cube :**

The Necker cube is a famous optical illusion — a **simple line drawing of a cube** that can be perceived in two different 3D orientations, and your perception tends to flip back and forth between them over time.

* It's a wireframe cube — just 12 lines forming the edges of a cube — with **no shading** or **depth cues**.
* That ambiguity allows your brain to interpret it in more than one way. It shows how our visual system adds 3D structure to 2D input.
* In Hinton's neural networks lecture, it’s often used as an example to illustrate how **different interpretations** can be modeled as **different low-energy states** in a network.
* Hinton uses the Necker cube to explain how a network might **settle** into one of two **low-energy states**, each representing a **different 3D interpretation** of the cube. It helps illustrate how networks can represent multiple plausible interpretations of the same sensory data.
* **How it works:**
* **Data units (2D-line units):** One neuron for every possible straight line you might see in the image.
* **Activation:** When you feed in the drawing, only the neurons whose corresponding lines actually appear will turn on.
* **Hypothesis units (3D-line units):** One neuron for each possible 3D edge (line in space) that could project to each 2D line.
* **Excitatory links:** Every active 2D-line unit sends equal “go-on” signals to all the 3D-line units that could have produced it.
* **Inhibitory links:** Those same candidate 3D-line units mutually inhibit each other—so only one “best” 3D hypothesis per 2D line survives.
* **Enforcing Spatial Consistency**
* **Co‐depth support:** If two 2D lines meet at a corner, their chosen 3D edges should meet at the same depth. We add supportive connections between their 3D-line units.
* **Right‐angle boost:** In man-made scenes, lots of edges meet at right angles. If two hypothesized 3D edges both agree on depth and form (or nearly form) a right angle, we strengthen their mutual support even more.
* **Energy‐Based Inference**

This whole network—2D-line inputs, 3D-line hypotheses, plus the support/inhibition links—defines an energy landscape.

* **Dynamics:** Starting from all the active 2D lines and random guesses for 3D lines, you iteratively update each 3D-line neuron to reduce the network’s energy.
* **Outcome:** The network “falls into” one of its low‐energy minima, each minimum representing a coherent 3D interpretation of the drawing.

Necker Cube Example

* An example: Interpreting a line drawing

We imagine we see a line drawing, and we want to interpret it as a three-dimensional thing.

|  |  |
| --- | --- |
| * **The Necker Cube Example**   The famous Necker cube drawing has **two** equally plausible **3D forms** (front face “pops” either way).   * Our network will have **two deep**, roughly **equal minima**, so if you let it run, it can spontaneously flip between those interpretations—just like you see when you stare at a Necker cube. |  |

* The data we have— is a bunch of 2-D lines like the lines shown in the picture (the cube). For **each possible line**, we will have set aside a **neuron**. [That is, one “2-D line unit” per straight line orientation/position in the image.]
* So, for every possible 2-D line, we have a neuron.
* **Use one “3-D line” unit for each possible 3-D line in the scene.**

[Now we introduce “hypothesis” neurons, each representing a candidate edge in 3D space.]

* Each 2-D line unit could be the projection of many possible 3-D lines. Make these 3-D lines compete.

**[Excitatory links (green)]:** When a 2-D line neuron is active, it “turns on” all 3-D hypotheses that could project to it.

**[Inhibitory links (red)]:** Those candidate 3-D hypotheses fight among themselves so that only the single best one survives activation.

* **Co-depth support:** Make 3-D lines support each other if they join in 3-D. If two 2-D lines meet at a corner, we expect their 3-D edges to meet at the same distance from the viewer. So their units mutually boost each other.
* **Right-angle boost:** Make them strongly support each other if they join at right angles. In man-made objects (like cubes), edges often meet at . When two 3-D edges agree on depth and angle, we give them an extra-strong connection.
* In any one picture, only a few of the possible lines will be present.
* And so, we’ll **activate** just a **few** of those **2-D line neurons**. [Only the lines you actually see in the drawing light up their units.]
* Now, what we’re going to do is have a whole bunch of **3-D line units**, one for each **possible 3-D edge**.
* Each 2-D line could come from many 3-D lines, so we connect that 2-D neuron to all its candidates (excitatory), then let those candidates inhibit one another so only one “wins.”

[This is repeated for every **active 2-D line** in the image.]

* We’ve now wired in the projection information (green/red connections) that says:

1. A 2-D line -> many 3-D candidates, but choose one.
2. Chosen 3-D edges should agree on depth where they meet.
3. If they meet at right angles, they get extra support.

* With all these connections, the network encodes both how 3-D edges project to 2-D and how 3-D edges join in the world.
* If we give that network an image, it will settle into a low-energy state (an attractor) that best fits those rules.
* For the **Necker cube**, there are **two** equally valid **3-D interpretations** (front face in/out). The **network** ends up with **two** deep **energy minima**, one for each interpretation—just like your perception flips back and forth.
* **Remember:** This is an analogy to illustrate how *energy-minimizing networks* can perform *perceptual interpretation*. Real vision systems add many more cues (shading, texture, motion), but this captures the core idea:
* “Use **low-energy states** of a network to represent the most consistent interpretation of ambiguous sensory input.”
* **"Wired into" means:** Built into the structure of the network — either physically (in hardware) or by design (in software or mathematical formulation).
* **In our context:** We have explicitly designed the network's connections (like excitatory and inhibitory links) to represent the relationship between 2D lines and possible 3D edges.

**11.4 Using stochastic units to improve search**

In this section, we're going to explain how *adding noise* can help systems *escape* from *local minima*. And, we're going to see what we have to do to the *units* in *Hopfield net* to add *noise* in the *appropriate way*.

* **Noisy networks find better energy minima**

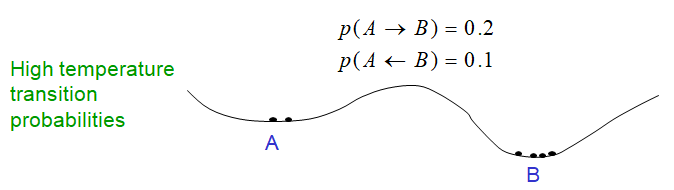
First, let's discuss the idea that we can find *better minima* by using *noise*.

|  |  |
| --- | --- |
| * Hopfield net always makes decisions that *reduce the energy*, or if it *doesn't change the state of the unit*, the *energy stays the same*. * This makes it *impossible* to *escape* (climb out) from *local minima*. |  |

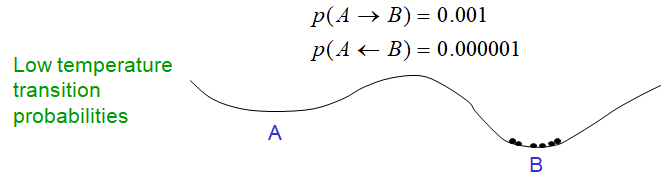
* Consider the above landscape. If we get into the *local minimum* A, there's no way we're going to *get over* the *energy barrier* to get to the *better minimum* B because we *can't go uphill* in *energy*.
* ***Adding noise:*** If we add *random noise*, we can *escape* from *poor minima*, especially minima that is shallow, that is, ones that *don't have big energy barriers* around them.
* ***Start with a lot of noise so its easy to cross energy barriers:*** It turns out, rather than using a *fixed noise level*, the *most effective strategy* is— to *start* with a *lot of noise* which allows you to explore the space on a *coarse scale* and find the generally good regions of the space, and then to *decrease* the *noise level*.
* With a *lot* of *noise*, you can *cross big barriers*.
* As you *decrease* the *noise level*, you start *concentrating* on the *best nearby minima*.
* ***Simulated Annealing:*** If we *slowly reduce* the *noise*, so the system ends up in a deep minimum, that's called "*simulated annealing*". And this ideal was, proposed by *Kirkpatrick* (et.al. 1981) at around the same time as *Hopfield* nets were proposed.
* **How temperature affects transition probabilities**

The reason for *simulated annealing* is because the *temperature*, (in a *physical system*, or in a *simulated system* with an *energy function*), affects the *transition probabilities*.

* ***High temperature transition probabilities:*** In a high temperature system, the *probability of going uphill from B to A* is *lower* than the *probability of going downhill from A to B*.
* But it's *not much lower*. In effect, the *temperature flattens* the *energy landscape*.



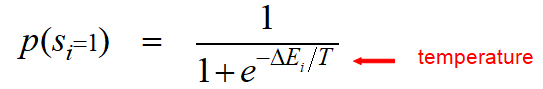
* Notice in this figure­— the little *black dots* are meant to be *particles*. And what we are imagining is— particles moving about according to the *transition probabilities* that we get with an *energy function* and a *temperature*.
* This might be a *typical distribution* if we're on the *system of high temperature* where it's *easier to cross barriers*, but it's also *hard to stay in a deep minimum* once we've got there.
* ***Low temperature transition probabilities:*** If we are in the system of much *lower temperature*. Then our *probability of crossing barriers* gets much *smaller* but our *ratio* gets much *better*.
* So, the *ratio* of the *probability of going from A to B* versus the *probability of going from B to A* is much *better* in the *low temperature system*. If we run it long enough, we would expect *all of the particles* to *end up in B*.

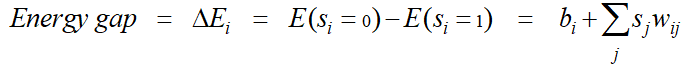


* ***Reduce the temperature gradually:*** But if we just run it for a *long time* at *low temperature*, it will take a very *long time* for particles *to escape* from A. And it turns out a good compromise is to *start at a high temperature* and *gradually reduce* the temperature.
* **Stochastic binary units**

The way we get *noise* into Hopfield net is to replace the *binary threshold units* by *binary stochastic units* and make *biased random decisions*.

* The amount of noise is controlled by something called *temperature*, (in the following equation).
* *Raising the noise level* is equivalent to *decreasing* all the *energy gaps* between configurations.
* So, this is our *normal logistic equation* but with the *energy gap* scaled by a *temperature* .
* If the *temperature* is *very high*, that exponent will be roughly zero, so the right hand side will be . And so, the *probability* of the *unit turning on* will be about a *half*. i.e. it'll be in it's on and off states.





* As we *lower the temperature*, depending on the sign of , the unit will become either more and more *firmly on* and more and more *firmly off*.
* At zero temperature, which is what we're be using in a Hopfield net, then the sign of determines whether the *right hand side* goes to *zero* or goes to *one*.
* But, with , it will either be 0 or 1 on the right hand side. And so, the unit will *behave deterministically* and that's a *binary threshold unit*. It will always *adopt* whatever of the two states is the *lowest energy*.
* So, the energy gap we saw on ***section 11.1***, and it's just the difference in the energy of the whole system depending on whether *unit i is off*, or the *unit i is on*.
* **Simulated annealing is a distraction**

Although *Simulated Annealing* is a very powerful method for *improving searches* that get *stuck* in *local optima*, and although it was one of the influential ideas in leading *Terry Sejnowski* and *I (G. Hinton)* to the ideas behind Boltzmann machines.

* But it's actually a big distraction from understanding Boltzmann machines.
* So it will not be covered in this course.
* And from now on, we will use *binary stochastic units* that have a *temperature of 1* i.e., it's the *standard logistic function* in the *energy gap*.
* **Terry Sejnowski** and **Geoffrey Hinton** made significant contributions to Boltzmann Machines (BMs) in the early 1980s, particularly in developing the learning algorithm for these **stochastic neural networks**.
* **What is a Boltzmann Machine?**

A Boltzmann Machine is a type of stochastic (randomized) recurrent neural network that can learn a probability distribution over its inputs.

* Hinton and Sejnowski pioneered a learning method for Boltzmann Machines that could train networks with hidden units using statistical physics ideas.
* Their work laid the foundations for probabilistic learning in neural networks and inspired many later deep learning architectures.
* **What did Hinton and Sejnowski do?**
* Developed the learning rule (1983–1985): Hinton and Sejnowski derived a stochastic learning algorithm for Boltzmann Machines, based on maximum likelihood. This was one of the first learning rules that could adjust the weights in multi-layer networks with hidden units.
* Introduced the idea of using gradient descent in energy-based models: The algorithm minimizes the difference between the data distribution and the model's distribution by adjusting weights. It uses a procedure inspired by **simulated annealing** and **thermodynamics**.
* Inspired later developments: The Boltzmann Machine was the forerunner to Restricted Boltzmann Machines (RBMs), which removed some connections for computational efficiency. RBMs later became a key component of Deep Belief Networks (DBNs), which Hinton helped popularize in the 2000s.
* **Key Paper:** *Hinton, G. E., & Sejnowski, T. J. (1985). Learning and Relearning in Boltzmann Machines, in Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1*.
* **Thermal equilibrium at a temperature of 1**

One concept that you need to understand in order to understand the *learning procedure* for *Boltzmann machines*, is the concept of *thermal equilibrium*. And since we're setting the temperature to 1, it's the concept of— *thermal equilibrium at a fix temperature*.

* It's a difficult concept. Most people think that it means— the *system is settled down* and *isn't changing anymore*. That's normally what equilibrium means. But reaching *thermal equilibrium* does not mean that the system has settled down into the *lowest energy configuration*.
* It's not the *states* of the *individual units* that are *settled down*. The *individual units* are still *rattling around* at *thermal equilibrium*, and *less temperature zero* (*rattling around* means— *to live or spend time in a place that is very big*).
* The thing that *settles down* is the *probability distribution over configurations*. That's a difficult concept the first time you meet it, and so consider the next example.
* The *probability distribution* *settles* to a particular distribution called the *Stationary Distribution*. The *stationary distribution* is determined by the *energy function* of the system.
* And, in fact, in the *stationary distribution*, the *probability of any configuration* is proportional to .
* A nice intuitive way to think about *thermal equilibrium* is to imagine a *huge ensemble of identical systems* that all have exactly the *same energy function*. So, imagine a very large number of *stochastic Hopfield nets* all with the *same weights*.
* Now, in that *huge ensemble*, we can define the *probability of configuration* as the *fraction of the systems* that are in that *configuration*.
* **Approaching thermal equilibrium**

So, now we can understand what's happening as we *approach* *thermal equilibrium*.

* We can *start with any distribution* we like over all these *identical systems*.
* We could make them all, be in the *same configuration*. So, that's the *distribution* with a probability of 1 *on one configuration*, and 0 on everything else.
* Or we could *start* them with an *equal number of systems* in each possible configuration. So that's a *uniform distribution*.
* And then, we're going to keep applying our *stochastic update rule* to pick the next configuration for each individual system. Which, in the case of a *stochastic Hopfield net* would mean:
* You pick a unit, and you look at its *energy gap*.
* And you make a *random decision* based on that *energy gap* about whether to turn it *on* or turn it *off*.
* Then, you go and pick another unit, and so on.
* We keep *applying* that *stochastic rule*. And after we've run systems stochastically in this way (the right way), we may eventually reach a situation where the *fraction of the systems* in *each configuration* remains *constant*.
* In fact, that's what will happen if we have *symmetric connections*.
* That's the *stationary distribution* that physicists call *thermal equilibrium*.
* Any given system *keeps changing* its *configuration*— we apply the update rule. And the *states* of its *units* will keep *flipping* between zero and one.
* But, the *fraction of systems* in any particular configuration (i.e. each configuration) *doesn't change*. And that's because we have many, *many more systems* than we have *configurations*.
* **An analogy**

Here's an analogy to help with the concept. Imagine a very large *casino* in *Las Vegas* with *lots of card dealers*. And, in fact, we have many *more than 52 factorial* (i.e. 52!) card dealers.

* We start with all the *card packs* in the *standard order* that they come from the manufacturer. Let's suppose that has the *ace of spades*, and the *king of spades*, and the *queen of spades*.
* Then, the dealers start *shuffling*. And they do *random shuffles* (not fancy shuffles that bring them back to the same order again).
* After a few *shuffles* (a few time steps), there's still a *good chance* that the *king of spades* will be *next to the queen of spades* in *any given pack*.
* So, the packs have *not* yet *forgotten* where they *started*.
* Their *initial order* is still *influencing* their *current order*.
* If we *keep shuffling*, eventually the *initial order* will be *irrelevant*. After prolonged shuffling the packs will have *forgotten* where they *started*.
* And, in fact, in this example, there will be an equal number of packs in each of the 52! *possible orders*.
* Once this has happened, if we carry on shuffling— there'll still be an equal number of packs in each of the 52! orders. That's why it's called equilibrium. It's because the *fraction* in *any one configuration* *doesn't change*, even though the *individual systems* are still *changing*.
* Once equilibrium has been reached, the number of packs that *"leave a configuration at each time step"* will be equal to the *"number that enter the configuration"*.
* The thing that's wrong with this analogy is that— *once we've reach equilibrium here, all configurations have equal energy*. And so, they all have the *same probability*.
* In general, *we're interested* in reaching *equilibrium* for systems where *some configurations* have *lower energy* than others.

Step-by-step Breakdown: Shuffling Card Packs in a Huge Casino Analogy

Imagine you're in a huge casino with tons of card dealers, even more than 52! (which is a huge number—the number of ways you can arrange a deck of 52 cards).

1. **Starting Point: Ordered Decks**

All dealers start with the same, ordered deck (like ace of spades, king of spades, queen of spades, etc.). This represents a system where all copies start in a known, low-entropy state (ordered and predictable).

* A **deck** (or pack) of **cards** refers to the full set of **52** standard playing **cards**

1. **Shuffling Begins**

Each dealer starts **randomly shuffling** their deck.

At first, the cards are still influenced by the original order—for example, the king and queen of spades might still be near each other. This is like the system still “remembering” where it came from—the initial state still affects the current state.

1. **More Shuffling: Losing Memory**

After many shuffles, the order becomes more and more random. Eventually, the decks “forget” their starting order completely.

1. **Equilibrium: Full Randomness**

At this point, every possible arrangement of cards is equally likely.

* We say the system has reached equilibrium:
* It’s not static—cards are still being shuffled.
* But the overall distribution doesn't change: the number of decks (sets of cards) in each possible arrangement stays roughly the same over time.
* For every deck that leaves a configuration, another enters it.
* **Important Note (What’s Wrong with the Analogy):**

In this analogy, every **card arrangement** has the **same probability**—because we assume no configuration is better than another.

* But in neural networks like Boltzmann Machines, we care about some configurations having lower energy than others, meaning:
* Some states are more likely, because they are more meaningful or “better” in the model.
* So unlike the shuffled decks, in real models, probabilities are not equal—they depend on energy.
* **Core Insight:**
* Hinton uses this analogy to explain how systems like Boltzmann Machines start with some structure, move toward randomness, and eventually settle into a stable pattern (equilibrium).
* But unlike the deck shuffling where all outcomes are equal, Boltzmann Machines are designed to prefer low-energy (more meaningful) states.

**11.5 How a Boltzmann machine models data**

In this section we'll explain, how a Boltzmann machine *models* a *set of binary data vectors*.

* We're going to start by explaining, "*why we might want to model a set of binary data vectors*", and *what we could do with such a model* if we had it.
* And then we're going to see— how the *probabilities assigned to binary data vectors* are *determined* by the *weights* in a Boltzmann machine.
* **Modeling binary data**

*Stochastic Hopfield nets with hidden units*, which we also call *Boltzmann Machines* are good at *modeling binary data*. Given a *training set* of *binary vectors*, they can use the *hidden units* to *fit a model* that will *assign a probability* to every possible binary vector. There are several reasons, why we might like to be able to do that:

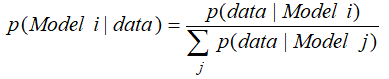
* For example, suppose you have *several* different *distributions of binary vectors*. You may want to look at a *new binary vector* and decide *which distribution* it most likely came from.
* Imagine *representing* a *document* using a *set of binary features*, where *each feature* indicates whether a *specific word appears* at least once in that document.
* So, you might have different kinds of documents (like *news articles* and *scientific papers*), and you might represent a document by, a number of binary features.
* *Different* types of *documents* (e.g., news articles vs. scientific papers) would have *different patterns of word usage*. You would expect *different correlations between words* depending on the *document type*.

Using a *set of hidden units*, the *Boltzmann Machine* can model the binary vector distribution for each document class.

* So in this case we could use a *set of hidden units* to *model the distribution* for *each document* and then we could pick the most likely document, by observing the probability.
* For a test document, you can determine which class it most likely belongs to by checking which *class-specific model* assigns the *highest* *probability* to its binary vector.

Therefore, this *Boltzmann Machine* is useful for deciding if *other binary vectors* come from the *same distribution* (e.g. documents represented by binary features that represents the occurrence of a particular word).

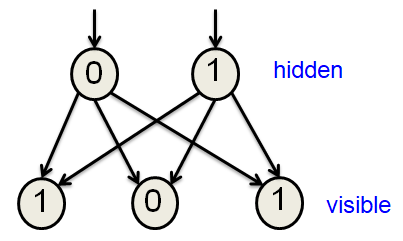
* *It can be used for monitoring complex systems to detect unusual behavior:* We could also use *Boltzmann Machines* for *monitoring complex systems* to detect *unusual behavior*.
* Suppose for example that we have a *nuclear power station*, and all of the *dials* were *binary*. So we get a *whole bunch of binary numbers* that tell us something about the *state* of the *power station*.
  + What we'd like to do is— notice that it's in an *unusual state*. A state that's not like states we've seen before.
  + And we *don't want* to use *supervised learning* for that. Because really we don't want to have any examples of *states* that cause it to *blowup*.
  + We'd rather be able to *detect* that it's going into *such a state* *without ever having seen* such a state before.
* We could do that by building a *model* of a *normal state* look like and *noticing* that this state is *different* from the *normal states*.



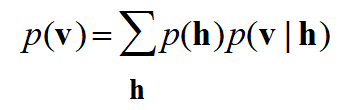
* If we have *models* of several *different distributions*. We can use a *Boltzmann Machine* to *compute* the *posterior probability* that a *particular distribution* produced the *observed data* (by using Bayes' Theorem).
* So *given* the *observed data*, the *probability* it came from *Model i*, under the assumption that it came from *one of our models*, is the *probability* that *Model i* would have produced that data, divided by the *same quantity for all models*.
* **How a Causal Model generates data**

|  |  |
| --- | --- |
| Now let's talk about *two ways* of *producing models of data* in particular *binary vectors*. The most natural way to think about generating a binary vector is to *first generate* the *states* of some *latent variables*, and then *use* the *latent variables* to *generate* the *binary vector*. |  |

* In a causal model we *generate data* in *two sequential steps*:
* ***First pick the hidden states from their prior distribution:*** Notice 0 and 1 in above layer, these are the *latent variables*, or *hidden units*, and we first pick the *states of the latent variables* from their *prior distributions*.
* Often in the causal model, these will be *independent* in the *prior*.
* So their *probability of turning on*, if they were *binary latent variables*, would just *depend* on some *bias* that *each one of them has*.
* Then *pick the visible states* from their *conditional distribution* given the hidden states: Once we picked a *state* for those *latent variables*, we would *use those* to *generate the states* of the *visible units* by using *weighted connections* in this model.



* So this is a kind of neural network— ***Causal Generative Model***. It's using *logistic units*, and it uses *biases* for the *hidden units* and *weights* on the *connections between* *hidden* and *visible* units to *assign* a *probability* to *every possible visible vector*.
* The *probability* *.*  of *generating* a particular *vector* , is just the *sum over all the possible hidden states* () of the *probability of generating those hidden state*  times the *probability of generating , given that you've already generated that hidden state* .



* i.e. *probability of generating a visible vector, v i.e.* , is computed by *summing over all possible hidden states* . *Each hidden state* is an *“explanation” of v*. Notice and are the vectors.
* In the above discussion of *causal generative models* and *Boltzmann Machines*— the symbols and represent *vectors* of binary units:

= *hidden (latent) variables vector*

= *visible variables vector*

These vectors typically consist of *binary values* (0 or 1) and are used to *describe* the *states of neurons* or *units* in two layers: hidden and visible.

* ***Is it a generated model?***

Above is a *causal model*. Factor analysis for example is a *causal model using continuous variables*. And, it's probably the most natural way to think about *generating data*. In fact, some people when they say *"generated model"* mean— the *causal model* like this.

* But that's a *completely different kind of model*.
* **How a Boltzmann Machine generates data**
* ***BM is not a causal generative model:*** A Boltzmann Machine is an *energy based model*, and, in this kind of model, you *don't generate data causally*.
* It's *not a causal generative model*.
* Instead everything is defined in terms of the *energies* of *joint configurations* of the *visible* and *hidden* units.
* *The energies of joint configurations are related to their probabilities in two ways:* There's two ways of *relating* the *energy of a joint configuration* to its *probability*—
* We can *simply define the probability* to be— the *probability* of a *joint configuration* of the *visible* and *hidden* variables is *proportional* to to the *negative energy* of that *joint configuration* i.e. .
* Or we can *define it procedurally* by saying— we are going to *define the probability* as the *probability finding the network in that state* after we've *updating all the stochastic binary units* for enough time so that we reached *thermal equilibrium*.

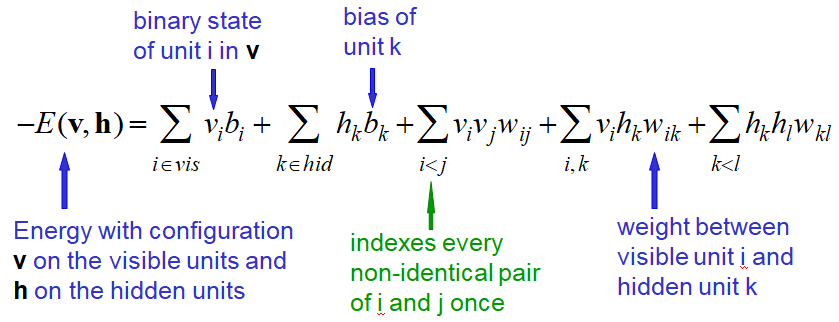


i.e. we can define the probability to be— the *probability of finding the network in that joint configuration* after we have *updated* *all of the stochastic binary units* many times.

* The good news is that those two definitions agree.
* **The Energy of a joint configuration**

The *energy of a joint configuration* of the *visible* and *hidden units* has *five terms* in it. So we've put the *negative energy* to save having to *put* lots of *minus signs*. So its' the *negative energy* of the *joint configuration* with vector on the visible units, and on the hidden units.

* Here is the *Energy configuration* on the *visible units* and on the *hidden units*.
* *Bias Terms:* It has the bias terms and where is the *binary state* of the i-th unit in vector , and is the *bias* of the k-th unit, (in this case, a *hidden unit* ). So that's the first two terms.
* is the *binary state* (i.e. either 0 or 1) of visible unit i, is the *bias* of the i-th visible unit.
* is the *binary state* (i.e. ) of hidden unit k, is the *bias* of the k-th hidden unit.
* *Visible-Visible interactions:* Then there's the visible-visible interactions .
* *To avoid counting* each of those interactions *twice*, we're going to count indices i and j and make sure that i is always less than j i.e. i<j.
* That'll avoid *counting the interactions something with itself*, and also avoid *counting pairs twice*, and so we don't have to put a in front.
* and are the *binary states* of the i-th and j-th visible units.
* is the *weight* between visible units i and j.



* *Visible-Hidden interactions:* Notice there's the *visible-hidden interactions* . Where is a weight on a *visible-hidden interaction*.
* is the *binary state* (i.e. either 0 or 1) of the i-th visible unit.
* is the *binary state* (i.e. ) of the k-th hidden unit.
* is a weight between i-th visible unit and k-th hidden unit.
* *Hidden-Hidden interactions:* Lastly, we have the *hidden to hidden interactions* .
* and are the *binary states* of the k-th and l-th hidden units.
* is the *weight* between hidden units k and l.
* **Using energies to define probabilities**

The *probability of a joint configuration* over both *visible* and *hidden* units *depends on* the energy of *that joint configuration* compared with the *energy of all other joint configurations*.

Where:

: visible vector

: hidden vector

: energy of configuration

: partition function

* ***Joint Probability of a Configuration:*** The way we use the *energies* to define *probabilities* is that— the *probability* of a *joint configuration* over and is *proportional* to .
* To make that an *equality* we need to normalize the right hand side by all *possible configurations* over the *visible* and *hidden* units and that's what the divisor there is, i.e .
* ***Partition function:*** The divisor of the above configuration is often called the *partition function* (that's what physicists call it). And notice it has *exponentially many terms*.

This sum is over all possible *visible vectors* and *hidden vectors* .

The *partition function* ensures the *total probability* sums to 1.

|  |  |
| --- | --- |
|  |  |

* ***Marginal Probability of a Visible Vector:*** To get the *probability* of a *configuration of the visible units* alone, we have to *sum over* all *possible configurations* of the *hidden units*.
* is the *sum over* all possible 's, and is the energy you get with that , normalized by the *partition function* .
* i.e. the *probability* of a *configuration of the visible units* is the *sum of the probabilities* of all the *joint configurations* that *contain* it.

Here, probability of visible vector, which is obtained by summing out (marginalizing) the hidden units (hidden configurations).

* **An example of how weights define a distribution**

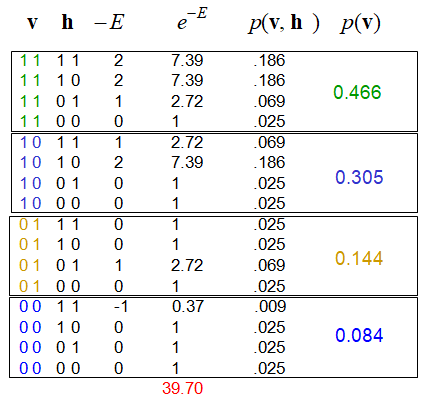
|  |  |
| --- | --- |
| Here is an example of how we *compute the probabilities* of the *different visible vectors*, it'll give us a good feel for what's involved. It's all very well to see the equations, but we understand it much better when we worked through the computation.   * Let's take a network with *two hidden* units and *two visible* units and we'll *ignore biases*, so we just got *three weights* here (-1, +2, +1). * To keep things *simple*, we're *not gonna connect visible units* to each other. |  |

* The first thing we do is— write down *all possible states* 1 1, 1 0, 0 1, 0 0 of the *visible units* (column ). We need to put them in *different colors*, and we're going to write *each state* *four times*,
* Because for *each state* of *visible units*, there are *four possible states* of the *hidden units* that could go with it (eg: for *visible* 0 1 state there's *hidden* states 1 1, 1 0, 0 1, 0 0).
* So that gives us *16* possible *joint configurations*.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |
| 1 1 | 1 1 | 2 | 7.39 | 0.186 |  |
| 1 1 | 1 0 | 2 | 7.39 | 0.186 | **0.466** |
| 1 1 | 0 1 | 1 | 2.72 | 0.069 |  |
| 1 1 | 0 0 | 0 | 1 | 0.025 |  |
|  | | | | | |
| 1 0 | 1 1 | 1 | 2.72 | 0.069 |  |
| 1 0 | 1 0 | 2 | 7.39 | 0.186 | **0.305** |
| 1 0 | 0 1 | 0 | 1 | 0.025 |  |
| 1 0 | 0 0 | 0 | 1 | 0.025 |  |
|  | | | | | |
| 0 1 | 1 1 | 0 | 1 | 0.025 |  |
| 0 1 | 1 0 | 0 | 1 | 0.025 | **0.144** |
| 0 1 | 0 1 | 1 | 2.72 | 0.069 |  |
| 0 1 | 0 0 | 0 | 1 | 0.025 |  |
|  | | | | | |
| 0 0 | 1 1 | -1 | 0.37 | 0.009 |  |
| 0 0 | 1 0 | 0 | 1 | 0.025 | **0.084** |
| 0 0 | 0 1 | 0 | 1 | 0.025 |  |
| 0 0 | 0 0 | 0 | 1 | 0.025 |  |
|  | | | **39.70** |  | |

* Now, for *each* of those *joint configurations*, we're going to compute it's *negative energy* .
* So if we look at the *first line*, when *all* of the *units* are *on*,(i.e. visible 1 1 hidden 1 1) the negative energy will be +2, -1, +1 which is +2 because (considering the weights in the image).
* We do this for all *sixteen* possible *joint configurations*.
* ***Probabilities of joint configurations:*** We then take the *negative energies* and we *exponentiate* them. And that will give us *un-normalized probabilities* () of the configurations. Their probabilities are proportional to these, i.e. before normalization (dividing by the partition function ), the values are not actual probabilities, but they are *proportional* to the *final probabilities*..
* If we add all those up to 39.7 (which is ) and then we divide those *un-normalized probabilities* by 39.7, we get the *probabilities of joint configurations*.
* Now, if we want the *probability* of a *particular visible configuration*, we have to *sum* over all the *hidden configurations* that could go with it. And so we add up the numbers in *each block* (i.e. for first block, 0.186 + 0.186 + 0.069 + 0.025 = 0.466).

And that’s how we've computed the probability of each possible *visible vector* in a *Boltzmann Machine* that has these *three weights* in it.



* **Getting a sample from the model**

Now let's ask— *How we get a sample from the model when the network's bigger than that?* Obviously, in the network we just computed, we can figure out the probability of everything because it's small. But when the network's big, we can't do these exponentially large computations.

* If there are more than a few hidden units, we *cannot actually compute* the *normalizing term* (the *partition function* ) because it has exponentially many (i.e. too many) terms in it.
* So we use *Markov Chain Monte Carlo (MCMC)* to get *samples* from the *model* *starting* from a *random global configuration*:
* And then *picking* units *at random* and *updating* them (their *states*) *stochastically* based on their *energy gaps*.
* Those *energy gaps* being *determined* by the *states* of all the *other units* in the network.

|  |  |
| --- | --- |
| * If we keep doing that until the *Markov chain* reaches its *stationary distribution* (thermal equilibrium at a temperature of 1), then we have a *sample* from the *model*. * And the *probability* of that *sample* (i.e. probability of a global configuration) is related to its *energy* by the *Boltzmann distribution*, that is, the probability of the sample is proportional to . |  |

* **Getting a SAMPLE from the posterior distribution over hidden configurations for a given data vector**

What about getting a *sample* from the *posterior distribution over hidden configurations*, when *given a data vector*? It turns out we're going to need that for learning.

* The number of *possible hidden configurations* is again *exponential*. So again, we use *Markov Chain Monte Carlo (MCMC)* to sample from the *posterior*.
* It is just the same as *getting a sample* from the *model*, except that we keep the *visible units clamped to the data vector* we're interested in.
* So we only update the hidden units (i.e. allowed to change states).
* The *reason* we need to get *samples* from the *posterior distribution, given a data vector*, is— we might want to know a *good explanation* for the *observed data*. And, we might want to base our actions on that good explanation. But, we also need to know that for learning.
* Samples from the posterior are *required* for *learning the weights*.
* Each *hidden configuration* is an “*explanation*” of an *observed visible configuration*.
* *Better explanations* have *lower energy*.

MCMC review

Markov Chain Monte Carlo (MCMC) is a method used to **get random samples** from a **complicated probability distribution** — especially when we can't draw samples directly.

Sometimes, you **know the shape of a distribution** (like from a Bayesian model), but you **can't sample from it directly** (especially in Bayesian statistics where the posterior distribution is often complex and known only up to a proportionality constant). MCMC helps you generate samples that match that distribution.

**Markov Chain:** A process that moves step-by-step, where each step depends only on the last one. This defines **memorylessness**. The next sample position depends only on the current position.

**Monte Carlo:** Using randomness to solve problems (like random sampling). Defines the stochastic (random) nature of the method.

**How it works:** MCMC explores a distribution by taking smart random steps, eventually giving good samples.

* Walks randomly through possible values, where **each next step** depends only on the **current position** (Markov property).
* Adjusts steps to favor more probable values (Monte Carlo randomness).
* Over time, the samples converge to the true distribution.

**What MCMC does:** It creates a **sequence (chain) of samples** that slowly "walks around" the space where your **target distribution** lives. Over time, this walk visits regions according to how likely they are.

* This is the key visual/intuition. The chain is a correlated random walk. Crucially, it spends more time (visits more often) in high-probability regions of the target distribution and less time in low-probability regions.