March 7, 2003
5th Annual Lecture

Geoffrey Hinton
Professor, Department of Computer Science
University of Toronto

Learning Representations by Unlearning Beliefs

The issue I am going to talk about is how the brain learns internal representations. There are two broad theories of how it might do this: one is it might develop representations by finding combinations of features that are good for predicting the correct output and the other is that it might develop internal representations by building a model of the world that explains why you have got those sensory inputs and not other ones. It is much easier to learn things if you are given the correct answer. But unfortunately we are not always given the correct answer, although sometimes we get lucky.

There are good old-fashioned neural networks that were supervised learning devices. This [SLIDE 3] is a typical backpropagation network where you have multiple layers of units and you learn the weights on the connections. You learn these weights by taking an output, comparing it with the correct answer (so you have to have been given the correct answer while you are training) and sending signals backwards through the net to compute derivatives for all these weights, and then you learn them.

There are some problems with backpropagation: almost all data you can easily get is unlabeled. You can point a TV camera at the world and get a lot of data, but nobody is going to tell you where all the edges are in that data. Also, the brain has about $10^{14}$ parameters in it, and you only have about $10^9$ seconds to fit them. So you have to fit about $10^5$ per second and you are not going to get that much information from the labels. If you are lucky, someone will say "That's a cow" and "This is a sheep". That is not going to give you $10^5$ parameters a second. Now, it might be that you do not use all these parameters effectively; but there is only one place where you can get $10^5$ parameters a second from, and that is the sensory input. There, there is plenty of bandwidth. So, backpropagation is bad because it needs these labels. But it is appealing because it is an efficient way to figure out derivatives for the parameters in the network, and it would be
a shame to give up on that efficiency, in particular if you have built your career on it. So, maybe we can recycle backpropagation.

If we go to the idea of learning to model the structure of the sensory input data - unsupervised learning - there is a standard approach to this now, which is to say we are going to have a stochastic causal generative model. We are going to imagine there are internal causes that gave rise to the data; and we are going to try to fit that model to the data by adjusting the parameters in this generative process, so that the generative model is more likely to generate the data we observe. That is a good thing to do, because later on you would like to associate your responses with these internal causes, not with the raw data; it is helpful for understanding the data; and it would be interesting neuroscience if that was what neurons really did. And the standard way to go about doing this is to have something often called a Bayes net, which has directed connections. [refers to picture on SLIDE 6] The generative process starts by choosing random activities here. Given these activities, they provide inputs to this layer. That allows you to choose activities here, which will be correlated by this thing up here. And finally you get the things you are actually going to observe, like pixels in an image or bits of a speech wave, and they can have all sorts of complicated correlations in them, caused by these underlying causes, which are stochastic. The good thing about these models, or one good thing, is that it is very easy to generate samples of what the model believes. You start up here [SLIDE 6], and you work downwards. It is all probabilistic inside, and the model will produce the kind of things it believes in. Each time you run it, you are going to get a different sample. So you can see what it is that your model has learned, or what the model tends to believe. It is typically hard, (particularly in networks with nonlinear interactions, and with dense connectivity) even if you knew the weights on the connections, when you see some data that the net is meant to have produced, to figure out how the net produced it. This is the kind of model where, even if we know all the parameters of the model, it is hard to figure out how the model did something, because there are so many possible ways it might do it. The best we can typically hope to do in densely connected nets is to get the sample from the distribution of the ways it might have done it. The good news is: if we get a sample, if we get some typical way the network might have produced this data, then we can learn the parameters in this network. Getting samples is sufficient, and, having got these samples, it is very easy to do the learning. The learning all decouples, and you can learn. If you have got a sample value for this and a sample value for this when the network was producing this image down here [SLIDE 6], then it is very easy to learn how to adjust that weight [SLIDE 6]. There is a lot of research now on how to approximate the posterior, and that is one of the main lines in AI. It is a big link between the statistics community, the neural nets community, the AI community. It is sort of mainline now, which makes me want to do something else. So, here [SLIDE 7, but no picture] is a different kind of hidden structure. We can characterize data in terms of a causal process that gave rise to it, but there are other ways of characterizing data that are different. We could, for example, say that a lot of data satisfies constraints, and it does not always satisfy these constraints, but it usually does. When it violates the constraints, it might violate them by a lot. But a big set of constraints like this, most of which are usually satisfied, is a good way to characterize all sorts of data sets. I will call these Frequently Approximatively Satisfied Constraints. The reason I will have to call them that is because
there is really no theory behind this, but if you use an acronym that is close to something where there is a lot of theory, then, you know, people think you are doing OK here. So, we are going to assume about these constraints that when they are satisfied, they are satisfied quite accurately, and when they are violated, they might be violated by a whole lot. For those of you who are used to Independent Component Analysis, this is equivalent to assuming that the violations of these constraints do not form some Gaussian distribution. They form a very heavy tail distribution, where you get occasionally very big ones, but most of the violations are very small. The causal generative stochastic model is a way of defining probability of data via the probability of these hidden causes. So what you do to define the probability of data vector $D$ in these causal models is you say “I have my deepest hidden causes (let us call those $B$) and there is some probability of those hidden causes arising. Given that these deepest causes arose, there is some probability of some more superficial causes arising, caused by the deepest causes, and that is a conditional probability. And then, given those deepest causes and the more superficial ones, there is some probability of them giving rise to the data.” So, the overall probability of observing some particular image or data vector is the product of all these probabilities: first the deepest ones, then the more superficial ones, and finally the data itself. That is the order in which you generate things from this model. As I said, in these models, it is easy to generate from the model. It is typically hard to figure out how the model actually did generate some data if I show you the data, but if you can do that, or get samples of that, learning is easy. And on the other side, we have a different kind of model, where we do not represent the probability of the data, or we do not model the probability of the data as the result of some causal process. We say “There are all sorts of constraints in this data, and if I look at the data and then take all my different constraints that might apply and see which ones apply and which ones do not. Each time I violate a constraint, that will give rise to some energy (think of energy as badness = violation of constraints), and all I am going to do for the data is just to add up all these constraint violations that a given data vector has. Obviously, what I would like is that the kind of data I am trying to model does not violate the constraints much, but random data violates the constraints by a whole lot. Then I can explain why the kind of data I am trying to model is more likely to occur than random data. The way we do that is we simply say the probability of the data, $p(d)$, is $e^{-E(d)}$ [E(d) = energy of the data $d$] (so that when the data has very high energy, this is a very low probability. When the data has low energy, this can be quite big.), normalized by that same quantity for all possible data $c$:

$$p(d) = \frac{e^{-E(d)}}{\sum_c e^{-E(c)}}$$

If you try to model an image, this is slightly problematic, because you have to look at all possible images. Think of an image with 256 pixels. Even if they are binary, that is $2^{256}$ images going to this sum, in order just to compute the probability of the data. That seems to be a bit of a problem, and it is this (normalization) term here that seems to make learning hard. It also makes it very hard just to generate a sample of what the model believes. So, on the face of it, these do not look too good. But there is a subclass of these models that have a wonderful property, which is that the inference is very easy (so you can do perception in this model very fast). Given the data, if you have already done
learning (which is going to be a slow business), then if you ask “How does my internal representation get derived from the data so that I can represent this data?”, that is a very quick process, and you are going to represent the data in terms of the constraints that it violates, which is a good way to represent things. You represent data by what is peculiar about it. Within this framework, you also have features, so you can represent what is good about the data too. So, the question is “How are we going to learn these models?”, given that we have got this huge exponentially large sum to deal with here. Before I get into that, let me talk about a particular kind of model like this, and this is where we are going to recycle backpropagation. We are going to assign energy to data by taking a data vector and having a feed-forward neural network composed of nonlinear units. We just take the data and do a forward pass through the net, very quick. It takes one synapse time delay to activate each of these guys, so you can go through six layers in a few tens of milliseconds if you wanted to, and then we are going to say that the energy of the data is just the sum of energies contributed by each of these units. Typically, a unit like this one that might be representing some constraint would say “I would like to give an output of zero, and if I don’t give an output of zero, that is bad.” And you add up all these badnesses to see just how bad this data was. The unit could also give an output that says “If I give an output, that is good”, but I will talk less about this in this talk.

So we can use a feedforward neural network just as a way of assigning energy to data. The big advantage of that is that we can now backpropagate through this net. If everything is smooth, there is a very efficient way of computing how the energy of the data would change if we were to change the data a little bit, and also there is an efficient way of how to compute how the energy would change if I changed the parameters in the network, the weights on the connections.

Let us think about taking some data and changing the parameters in that net (the weights in the neural net) so as to make that data more probable. There are two things we have got to do: there is the easy bit and the tough bit. The easy bit is: to make this data probable, we need to make it low energy. We need to adjust our constraints so that they fit in with this data. If it is violating some constraints, correct the constraints so that it is not. That simply says “Make the energy assigned to this data low”, so make this little neural net not produce much output when it is shown this data. Do not complain with this data. But that is not enough. Just making this energy low is not sufficient to make the probability high. It might be that there are other energies for other data vectors that are even lower, in which case this $e^{E(c)}$ will be a big term here (because low energy makes this big), and this data will only get small probability because of these big terms in here $\sum e^{E(c)}$. So what we need to do also is to go and find other things that the network things are really good, maybe even better than the data, and we need to zap them. We need to, among these huge numbers of possibilities, go and find the other ones the network really believes in, and we need to make them worse. What you must not do is go and find the things you believe in and make them better, that is, change your model so that what you believe in gets to be more plausible. This is the George Bush algorithm. It is better to run the algorithm where you actually make the data increase your probabilities on things, and then your beliefs decrease them. So how are we going to find all these things that the model believes in that we need to zap in order to make the data more
likely? There is an obvious method, which is to start with the data, and then perturb it a little bit and see if the model prefers that. To see if when we perturb the data we get something of lower energy, and if we do, we will accept that and perturb it again to see if we get lower energy again. If we just keep doing that, and if we perturb it in the right kind of noisy way, if we do that long enough, we will eventually start sampling the things the model really strongly believes in. Those are things we really want to zap. We really want to unlearn those. The problem with this is that it is a very slow business. There might be some region in the data space where the model's beliefs are more or less equal whatever you do, and then suddenly you fix things up so that it is something the model likes, and it starts believing it a lot. When you move very slowly over those regions, there might be barriers in this space where the data is in a slow local minimum of things the model thinks are not too bad, but if you were to change it a lot you would find things the model thinks are much better, that you want to unlearn. There is a better method which we can use in continuous spaces where we can get gradients, called Hybrid Monte Carlo, where we actually use a gradient and we use momentum. The nice thing is that backpropagation in this neural net that assigns energies to data can give us this gradient. In Hybrid Monte Carlo, you take your data, we do a forward pass, which will allow us to compute the energy of that data, and then we are going to do a backward pass. As we come backwards through the network, we will compute all the things we need, so that when we get all the way back to the input of the network, we can figure out how the energy of this data would change if I were to change one of the inputs a little bit - the derivative of the energy with respect to the components in this input vector. Then we can fix up the input, we can start changing the input so that the network is happier with it. That is a way of finding the things the network really believes in. You adjust the facts so that they are more plausible to you, but you must remember that you are then meant to do unlearning. So, that is another reason why this network is nice. We can go off and adjust these things to find things that are more plausible to the model.

Let me show you how that works. I am going to take two-dimensional data, because the best way to describe this is with a picture. I am going to take two-dimensional data so that I can plot out what all the data look like. In this case I had data that lay along a line here, a line here and another line there - three lines. And this is the model towards the end of learning. The height here is the energy that the model assigns to a data point. The horizontal coordinate is the data point, the height is the energy assigned to that data point by the network's current model. Here is a data point. What the network is going to do is take this data point, make up a little bit of random momentum, so start moving randomly in some direction, and then it is going to follow the dynamics that a particle would follow if it moved over this energy surface. And it does act like going backwards and forwards in this neural net, computing derivatives. The derivatives give it the gradient of this energy. It can move along the gradient, but it is using momentum. If you look at this path, for example, it goes up and over this little ridge here and down again. These are just three different trajectories that involve different initial random momentum. You can see, starting here, it tends to end up in places that the model likes, low energy places.
The learning algorithm is now just: at the data, change the weights in the network to make this image be lower energy, and at the things that the network really believes in, change the weights in the network to make this be higher energy. I think you can see that if we keep doing that, eventually the data will end up as a low-energy region and everything else will end up as a high-energy region, which is what we want.

Here is a simple 2D dataset [SLIDE 14]. I generated some 2D data from within these squares with a uniform distribution. This blue dots is not the data, this is what the model generates after it has learned the data. You can see, it is a pretty good fit. The model knows that the data has to lie around here, and you must not have any data anywhere around here or here. Now I have made a little movie [SLIDES 16-27] of what happens to the energy surface as the model learns this data. You initially just show the model a bunch of samples drawn from this data set, and the model, with an initially very flat energy surface, will say "well, how can I change that?", and then it will wander off randomly to places around here. Now what you will do is you are learning on the data, so that will push the energy surface down in the middle here and unlearning on the places the model wanders off to, so that will raise the energy surface out here. The first thing it will do is to make a bowl. It will make a big bowl like this with the data in the middle, and it will have learned that you do not get data out here. Gradually it will refine that bowl.

The model that we are using for this data set has two inputs for the two-dimensional data [SLIDE 15]. We use twenty logistic units, typical backpropagation units, and three logistic units in the second layer. Each of the units is going to have an activity that is some non-linear function of the input it gets. It is going to contribute energy just according to its activity. It can create a little energy rise like this, with a plateau at the top. And it is going to be able to scale this axis, so it can learn to make this very high or to make it quite low, so it can create big energies if it wants to.

[referring to SLIDES 16-27] After about ten iterations of learning, it has learned that the data is pushing the energy surface down here. The confabulations produced by the model, that is the things that it got when it started with data and then wandered off, raise the energy surface around here. After a while it will stop producing confabulations out here and start concentrating here. The energy surface gets deeper in the middle and higher at the edges. Now it is beginning to learn the shape of the central region of the data. It is beginning to learn that the data lies in a square. It has got the square quite well now. So the data all lies in there, and will not go outside there when it is confabulating. After a while it starts learning that the data is in two ridges, two columns like that, and you do not tend to get data in the middle. Again, it is learning that because the data is pushing it down here and here. It wanders off from the data to the middle, and it raises the energy of those confabulations, so it is pulling this piece of the surface up. We keep going, and it gets a nice surface which has two columns. Then it realizes there is something else going on -- and that is the final thing it produces. If you keep going, it will make much deeper ridges. Now it has got four energy minima here, which correspond to the four squares where the data is. This is now getting to be quite a big scale, of about 20 here, so it thinks
that this thing here is \( e^{20} \) times as likely as things up here. It has produced a little model of that data and the data that I showed you at the beginning is data from this model.

Let us apply this to something that is slightly less toy. Before I do that, let me give you the whole learning procedure. It works like this: you start with a data vector \( d \). You do a forward and backward pass through the net, and on the first backward pass you compute the derivatives \( \partial E(d)/\partial \theta \) of the energy of the data \( E(d) \) with respect to the weights \( \theta \) in the network. Then you do many steps up and down through the net, figuring out how to corrupt the data so that the network is happier with it. So you are finding things it really wants to believe in. After you have finished doing that, you now do an up-down pass to compute the derivatives on its beliefs. Then you actually change the parameters by some small learning rate

\[
\Delta \theta = \varepsilon (-\partial E(d)/\partial \theta + \partial E(c)/\partial \theta )
\]

(On the data we lower the energy, so we have a minus sign; on the things the network believes, we raise the energy, so it is stamping out what it would like to believe. Of course, if it believes in the data, those two terms cancel out, so it will stay where it is).

This algorithm as I showed it to you would be a correct maximum-likelihood learning algorithm if we could only sample from what the model really believes. But, even on that little model that I showed you, it would take many, many minutes just to produce one sample of what the model really believes. On bigger networks it takes hours and hours to produce a sample of what the model believes, so it is hopeless. People do learning like this, and they try to solve this hopeless problem. After a while they say “Well, let's not run it that long”, which is what I started doing, and it still seems to work, so you say “Let's run it for a little less time, and it still works.” It took me a very long time to say “Let's just run it for a few steps”, because you are obviously nowhere near what the model really believes. You have just changed the data a bit in the direction of what the model believes - and the algorithm works even better then! We get this shortcut. Instead of having to find out what the model really believes in, which could take an awful long time, all we need to do is take the data and just slightly corrupt it in the direction that the model would like to corrupt it in. That is a much quicker thing to do, and I think you can see that if whatever data you give the model it always corrupts it in the direction of Saddam Hussein being even worse than that, then you can see something the model really believes in.

This is a much, much more efficient algorithm. It has much less noise in it, because the data vector and the corruption of the data vector look now quite similar to each other. And it has one other nice property: if we did have a perfect model, and infinite amounts of data, then this would actually be the correct maximum likelihood learning algorithm, because with a perfect model it would not want to go anywhere, so it would diffuse around at random. This would be the right thing to do. We would not need to run it for very long. Towards the end of learning, this is going to work better. And that is when you need accurate learning anyway. The intuitive motivation is: it is silly to run this process of finding what the model believes in all the way to equilibrium, when the first few steps
tell you a lot about what is wrong with the model. You can see how it is trying to distort the data, and that is enough to learn. It has a drawback: the model could have very strong modes very far from any data, and you would not see those, and it is an empirical question as to whether these kinds of models really suffer much from that. They do not seem to suffer from that too badly, because if they did we would not be able to learn much this way.

Let me justify something about the kinds of energy functions I am going to use. I will use these neural nets, and the output of each neuron, or even the activity of the neurons, will be representing these approximately satisfied constraints. My neurons will be trying to give an output of zero. If the output is near zero, they will get a big gradient, this green curve, saying “Change the weights to make the output exactly zero”, but if the output is far away from zero, they will be saying “Well, this is hopeless. Don’t change anything.” This is like when you go to a math lecture and it does not make any sense. It is not a good idea to fix up your model there. Only fix up your model when you are beginning to understand what is going on. These green curves refer to graph on SLIDE 31 are very good for modeling constraints like, in an image with continuously changing intensities, if I put down a filter like this, it will almost always produce an output of zero, because the intensities in the middle, which are multiplied by +1 will cancel out the intensities at the sides, which are multiplied by -1, and I get an output of 0. Of course, that will not happen if there is an edge in the image. But edges are rare, and so a filter like this is really good for giving zero almost all the time, and then occasionally giving some huge value here in the tail. That is the kind of things it should learn. Later on, I will show you it learning filters like that.

First of all, let us do another toy problem that is slightly more interesting than the two-dimensional one. We are going to take a little robot arm, a synthetic one. It has got links of a constant length. It is in 3D, so this one left picture on SLIDE 32 got foreshortened. I am going to take something with five joints and four links. If I give you the three-dimensional coordinates of each of these joints, that is 15-dimensional data, three coordinates for each joint. This 15-dimensional data is not random. It has got a lot of randomness in it, because I put this arm in random configurations, but it has four constraints in it. These are non-linear constraints; they are not trivial to get from the data. The non-linear constraints are of the form “If I take two neighboring joints, and if I were to take the difference in the x-coordinate and square it, and take the difference in the y-coordinate and square it, and take the difference in the z-coordinate and square it, that should come to the length of the link, so if I subtract off the squared length of the link, I should get zero.” Now we are going to design a little net that could learn that right picture on SLIDE 32. We will take the 15 coordinates in here. We will have hidden units that take linear combinations of coordinates. So, by taking a plus and a minus, I can take the difference of the two x-coordinates. Then we will square the output of these guys. So I have cheated in a sense; I have told the model “Squaring is a very good non-linearity to use”. And then we will have a top layer. The top layer could learn to put in this $-l^2$, and then it will get out a zero. So let us see what happens when we turn learning loose on a network like this. SLIDE 33 These are the weights in the first hidden layer that it learns. And these are the weights in the second hidden layer. In the first hidden
layer, each unit corresponds to a column here. So these are all the weights of one of the hidden units in the first layer, and you can see if you look at this unit that it is ignoring all these coordinates, and it is looking at these three and these three, and what is more, these weights are the exact opposite of these weights, so it is taking the difference in some direction in three-dimensional space between the coordinates of this joint and of this joint. And what is more, there is some other guy who is taking that same difference. This guy here is taking the difference between this joint and this joint, and this guy next to him is taking the same difference between the same two joints. These are three orthogonal vectors. So it has set up a little rectangular coordinate system to take three differences, so that it can square them and sum the output up here. If you look here you see that for each pair of joints it has set up a little orthogonal coordinate system. There are three for this joint here, there are three for that one, three for this one and so on. It then sends the outputs up here. Here I have shown a hidden unit as a row, so you can see which of these goes to which of those. This black weight here will be the connection between this top level unit and this first level unit. If you look, for example, at this guy, and this guy and this guy, they are all measuring the difference between two neighboring joints, and they all have the same large negative weight to this top level unit. The idea is: they will send their squared differences here. These guys will add up the squared differences, and then you will subtract off a threshold, and then you will get zero, except that these guys in the top layer doing linear combinations of this, because it does not know that I wanted to interpret it. If you look at the biases that these top level units learn, they look like this. If you look at the average input they get from below, the mean of the input they get from below looks like this. So you can see that to within one percent of error, it is just the opposite of the bias, so these guys can produce outputs of zero. Now it is very happy with the data. If you show this arm in any configuration that satisfies the link constraints, it says it is very happy. If you violate the link constraints by making the coordinates wrong, it says it is very unhappy. What is more impressive is that it can learn this even if you take ten percent of the coordinates and make them to be random. To do that, what you have to do is: take these random coordinates, put in random values for them. Then, as it is learning the constraints, it is slowly going to fix up those random values. So it simultaneously going to learn the constraints that apply to the data and how to fix up the data so that these constraints apply.

Now let me talk a little bit about the application of this to natural images, which is a more real problem. A student called Yee-Whye Teh tried applying this kind of algorithm to 16 x 16 patches of natural images that were carefully traced on the web, using just one layer of 768 hidden units, so far more units than there are pixels. He had a net where he would take the data, produce confabulations, that is get the net to try and improve the data according to its current beliefs, by running 30 little steps over this energy surface. Then he would do unlearning on the confabulations and learning on the data. After every 100 examples he would update the weights. He would also introduce something to keep the weights small, so they are easy to interpret. He got weights like this: [SLIDE 36] (this is not all of them, this is just a subset). Some of them are kind of weird, and that is because it is pixelated data, and he did not filter it properly to begin with. But most of them are like this. You see what we have got here: low frequency edge detectors, higher frequency edge detectors and very high frequency edge detectors (he laid these out by hand; they
did not just come out like that), which is pretty much like what you find in the brain. It has got this just by looking at images and saying “Can I find constraints that are nearly always satisfied by an image, but occasionally violated by a lot”, and you can think of these things as constraints. That is, almost always if I multiply an image by positive numbers here and negative numbers there (so white things are positive numbers and black things are negative numbers), if I multiply an image that is this size by those numbers, then I will get a zero, almost always. Just occasionally I won’t, because just occasionally there will be an edge that lines up with this filter. In that case I get a huge non-zero, but that does not cost me too much, because of the kind of cost function I am using. Actually, people interpret all these filters as good for finding edges, but another interpretation of them is that they are a very good model of the constraints that images satisfy. Images satisfy the constraint that this guy should not get active, almost always, but then occasionally they violate that constraint by a lot. So you can also think of these filters as a way of expressing images in terms of constraints.

We would of course like it to lay out these filters like they are laid out in the brain of a cat or a monkey. There is a guy called Aapo Hyvärinen who discovered a neat trick that he used in conjunction with a different learning algorithm that will make these things learn topographic maps. The trick is this: we take the pixels of an image, we have a whole bunch of linear filters that just take linear combinations of pixel values. These linear filters then do something, like square their outputs, and send them to the next level. This is global connectivity, so these filters can decide where they are going to look in the image. But between the first and the second hidden layer we have local connectivity. So this hidden unit will only see things coming from here, and this one will only see things coming from this red region over here. This local connectivity is going to cause us to learn topography. Because of the way the local connectivity interacts with this heavy-tailed constraint. It is quite a complicated story, and I will try to explain it, but I may well fail.

In order to be good at rejecting nonsense, that is to get high energy for things that are not at all like the data, this network would like these linear filters to sometimes turn on. It cannot just make all their weight zero, otherwise it would not get high energy for nonsense, which is what it wants. So it is going to make these linear filters turn on sometimes. Now it has got a choice: when it makes them turn on, should it make ones near one another turn on at the same time, or should it make one over here and one over here and one over here turn on? Well, if it makes ones near one another turn on at the same time, then that is going to be good on real images. On real images what happens is that there is typically a lot going on in one part of an image. So these filters will get violated at the same time. And then they will both go to the same unit here. If we put filters that look at the same part of the image next to each other, they will go to the same top level unit. When the first filter gets turned on, it gets activity like this. The top level unit makes a spare cost like this, because if we send this much input to the top level unit, we pay this cost. When the second one gets turned on in the same green receptive field, it causes more activity, and we pay this additional cost. And you can see we are getting diminishing returns. So what this top guy would like is, it might prefer it for none of these guys to turn on, but if any of them turn on, he does not mind if a whole lot turn on.
We would much rather have a whole lot turned on in one receptive field than have them scattered all over the place. What we would like to do is arrange the filters so that on natural images when they come on, all the ones next to one another come on, whereas on random junk, like snow on the TV, then they will come on all over the place. So we can see what that does now. To get it to learn, we used a slightly different algorithm for producing the confabulations. It is the same basic idea. We used a different algorithm, but I am not going to talk about that. The slides are on the web, and I will give you the pointer later. This [SLIDE 39] is what it is learned. We are not laying these out by hand anymore. It is deciding where to put all these filters. It learns a patch of low-frequency filters in various orientations and positions, and then as you move away from this patch, the filters get to be higher frequency, finer in detail. The filters have nice, smooth, changes in orientation as you move through the space - you definitely see that in the brain. Also, the filters have nice, smooth, changes in their fine position. People do not really know whether the brain does that, because the techniques that you use for looking at these receptive fields, like optical dyes, cannot tell you about these fine positions. But they probably do. If you look here, for example, all of these filters that were near one another, they are interested in the top of the image. And as you move along, they become interested in the top right-hand side, and then the top right-hand corner. If you keep going, they move down. This is a wrap-around here, so they keep moving down. So, it produces a nice topographic map if you give it natural images. What we are getting here is just from a little constraint on the connectivity of the net, and what I think of as a sensible algorithm for extracting constraints from data we can learn these topographic maps that you can find all over the place in biology.

Now, I managed actually to go slightly faster than I thought I would, so I have time to actually now do a little bit of math, which is what I really wanted to do, but what I cut out. If we do the proper maximum-likelihood learning, where we take the data, and then we run this Markov chain for an awful long time to try and find the things the model really strongly believes in (in fact, we will have to run the chain for so long that the model has forgotten where it started) and it will now sample from the things the model really believes in. In that model I showed you for learning edges, for example, you might have to run the chain for ten years or so to get a sample of what the model believes in; that is just one sample to get one data point for doing learning. What that is doing when you run that chain, and you do the learning based on samples of what the model really believes in, is minimizing a difference between two probability distributions. \( P \) is the distribution of the data. That is what we are trying to model, the thing we are trying to give low energy to. And \( Q^\infty \) is the distribution of samples you would get if you ran this Markov chain forever, you kept sliding around on this energy surface, you kept adding new random momentum, you kept following the trajectory of a particle, and you did that for an awful long time. We then get a distribution of things the model really believes in, and we want these two distributions to be the same, and we would be following the gradient of the difference \( KL \), the Kullback-Leibler divergence, which is just a way of measuring the difference between two probability distributions.

If we do the algorithm that I suggested, of just running the chain for a very short time, we are taking the difference between these two distributions as one term, but we are also
If we actually figure out what these derivatives look like, then (sorry, I have called
if that is the case, you will have a good model.

The idea of the learning is: you want it to be the case that when the model corrupts the
data with its beliefs, it does not get any closer to its beliefs. If that is the case—if it
cannot make any progress by corrupting the data—that means that its beliefs are already
in accord with the data. That is what this is saying. This is saying I want it to be the case
that you cannot make progress by corrupting the data according to your model. Because
if that is the case, you will have a good model.

If we actually figure out what these derivatives look like, then (sorry, I have called \( P Q^0 \)
here, because that is where we start the chain), that is the data, \( Q^0 \), the distribution of the
data; and I put a minus sign, because we want to minimize those divergences:

\[
\frac{\partial KL(Q^0 \mid Q^\infty)}{\partial \theta} = -\left\{ \frac{\partial E}{\partial \theta} \right\}_U' + \left\{ \frac{\partial E}{\partial \theta} \right\}_V'
\]

The derivative of this divergence here with respect to the parameters of our model, if you
average the gradient over sample drawn from the data -- you take the energy derivative --,
and then you have to average over samples that the model believes in (these angled
brackets just mean: take an average over what the model believes in), and you get this
other term, which says "raise the energy of what the model believes in, and lower the
energy of the data". But of course, you have to get what the model believes in.

If you take this second term here, you can see that its derivatives have a term here which
is nice and easy to get:

\[
-\frac{\partial KL(Q^0 \mid Q^\infty)}{\partial \theta} = -\left\{ \frac{\partial E}{\partial \theta} \right\}_U' + \left\{ \frac{\partial E}{\partial \theta} \right\}_V' - \left\{ \frac{\partial Q}{\partial \theta} \right\}_U' \frac{\partial KL(Q^0 \mid Q^\infty)}{\partial Q^0}
\]

This says “take the derivatives of the energy after one time-step of running this chain, and
subtract off (unlearn) this same term you get here.” If we take the difference between
those two, the nasty things cancel out, and the whole point of taking the difference is to
make this term go away. Now, there is a nasty little extra term here that makes the story
not so nice theoretically, but in practice this extra term seems to be small on the toy
examples where you can compute how big it is. And the fact that learning works at all
suggests that this term is not messing us up much. That is the more mathematical story about what is going on.

Now I just wanted to show you where you can get more information about this. There are now quite a few papers on my website about this kind of learning. You can show that a method called Independent Component Analysis in the standard case can be viewed as both a causal generative model and as one of these energy-based models. I have got a paper showing lots of other kinds of learning with this method of contrastive divergence, where you do not use a backpropagation network, you use something else. There is also a technical version of this particular talk with more equations in it that you can get on the web.