Chapter – 1

**Neural Networks for Machine Learning**

**Geoffrey Hinton**

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

**Introduction to**

**Neural Networks**

Lectures: Geoffrey Hinton

Why do we need Machine Learning

What are Neural Networks

Some simple Models Of Neurons

A simple Example of learning

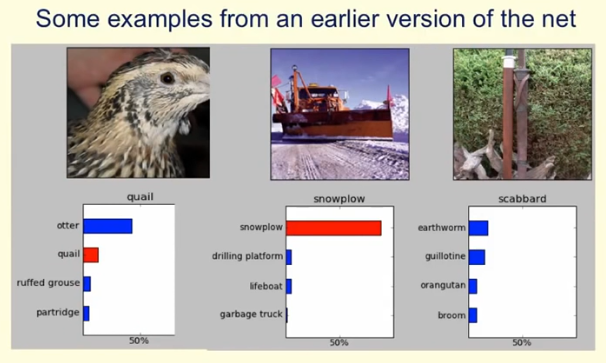
Three Types of learning

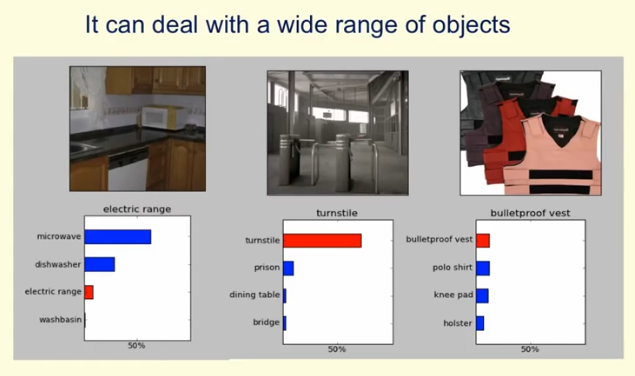
**1.1 Why do we need Machine Learning**

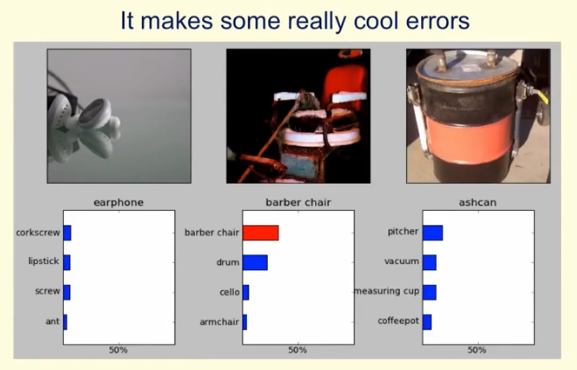
* What is Machine Learning?
* It is very hard to write programs that solve problems like recognizing a three-dimensional object from a novel viewpoint in new lighting conditions in a cluttered scene.
* We don’t know what program to write because we don't know how its done in our brain.
* Even if we had a good idea about how to do it, the program might be horrendously complicated.
* It is hard to write a program to compute the probability that a credit card transaction is fraudulent.
* There may not be any rules that are both simple and reliable. We need to combine a very large number of weak rules.
* Fraud is a moving target. The program needs to keep changing.
* The Machine Learning Approach
* Instead of writing a program by hand for each specific task, we collect lots of examples that specify the correct output for a given input.
* A machine learning algorithm then takes these examples and produces a program that does the job.
* The program produced by the learning algorithm may look very different from a typical hand-written program. It may contain millions of numbers.
* If we do it right, the program works for new cases as well as the ones we trained it on.
* If the data changes the program can change too by training on the new data.
* Massive amounts of computation are now cheaper than paying someone to write a task-specific program.
* Some examples of tasks best solved by ML
* Recognizing patterns:
* Objects in real scenes
* Facial identities or facial expressions
* Spoken words
* Recognizing anomalies:
* Unusual sequences of credit card transactions
* Unusual patterns of sensor readings in a nuclear power plant
* Prediction:
* Future stock prices or currency exchange rates
* Movie Recommender systems: Which movies will a person like?
* A standard example of machine learning
* A lot of genetics is done on fruit flies.
* They are convenient because they breed fast.
* We already know a lot about them.

|  |  |
| --- | --- |
| * The MNIST database of hand-written digits is the machine learning equivalent of fruit flies. * They are publicly available and we can learn them quite fast in a moderate-sized neural net. * We know a huge amount about how well various machine learning methods do on MNIST. * We will use MNIST as our standard task. * Beyond MNIST: The ImageNet task * 1000 different object classes in 1.3 million high-resolution training images from the web. * Best system in 2010 competition got **47% error** for its first choice and **25% error** for its **top 5 choices**. * Jitendra Malik (an eminent neural net sceptic) said that this competition is a good test of whether deep neural networks work well for object recognition. | * Example of some of the digits in the MNIST data-set |

* A very deep neural net (Krizhevsky et. al. 2012) gets less that 40% error for its first choice and less than 20% for its top 5 choices (see lecture 5).







|  |  |
| --- | --- |
| * The Speech Recognition Task * A speech recognition system has several stages: * Pre-processing: Convert the sound wave into a vector of acoustic coefficients. Extract a new vector about every 10 mille seconds. * The acoustic model: Use a few adjacent vectors of acoustic coefficients to place bets on which part of which phoneme is being spoken. * Decoding: Find the sequence of bets that does the best job of fitting the acoustic data and also fitting a model of the kinds of things people say. * Deep neural networks pioneered by George Dahl and Abdel-rahman Mohamed are now replacing the previous machine learning method for the acoustic model. |  |

**1.2 What are Neural Networks**

* Why should we need to understand how real Neurons of human bran works:
* To understand how the brain actually works.
* It's very big and very complicated and made of stuff that dies when you poke it around. So we need to use computer simulations.
* To understand a style of parallel computation inspired by neurons and their adaptive connections.
* Very different style from sequential computation.
* should be good for things that brains are good at (e.g. vision)
* Should be bad for things that brains are bad at (e.g. multiplication 23 x 71)
* To solve practical problems by using Novel Learning Algorithms inspired by the brain (this course)
* Learning algorithms can be very useful even if they are not how the brain actually works.

|  |  |
| --- | --- |
| * A typical cortical neuron: * Gross physical structure: * There is one axon that branches * There is a dendritic tree that *collects input* from other neurons. * Axons typically *contact* dendritic trees at *synapses* * A spike of activity in the axon causes charge to be injected into the *post-synaptic neuron*. |  |

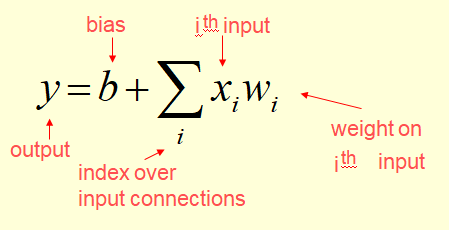
* Spike generation:
* There is an axon hillock that generates outgoing spikes whenever enough charge has flowed in at synapses to depolarize the cell membrane.
* Synapses
* When a spike of activity travels along an axon and arrives at a synapse it causes vesicles of transmitter chemical to be released.
* There are several kinds of transmitter.
* Ones that implement +ve weights
* Ones that implement -ve weights
* The transmitter molecules diffuse across the synaptic cleft and bind to receptor molecules in the membrane of the post-synaptic neuron thus by binding, they are changing their shape.
* That creates holes in the membrane
* These holes allow specific ions to flow in or out from the *post synaptic neuron*, and that changes the state of depolarization
* Synapses participate learning process by changing their effectiveness.
* How synapses adapt
* The effectiveness of the synapse can be changed:
* vary the number of vesicles of transmitter.
* vary the number of receptor molecules.
* Synapses are slow, but they have advantages over RAM
* They are very small and use very low-power.
* They adapt using locally available signals
* But what rules do they use to decide how to change?
* How the brain works
* Each neuron receives inputs from other neurons
* A few neurons also connect to receptors.
* Cortical neurons use spikes to communicate.

* The effect of each input line on the neuron is controlled by a synaptic weight
* The weights can be positive or negative.
* The synaptic weights **adapt** so that the whole network learns to perform useful computations
* Recognizing objects, understanding language, making plans, controlling the body.
* You have about **1011** neurons each with about **104** weights.
* A huge number of weights (**1014**) can affect the computation in a very short time. Much better bandwidth than a workstation.
* Modularity and the brain
* The cortex is modular. Different part of the cortex are learned to do different things but not genetically.
* Different bits of the cortex do different things.
* Local damage to the brain has specific effects.
* If an adult brains some part is damaged e.g. vision part, then we have visual problems.
* Or loose ability to recognize objects.
* Specific tasks increase the blood flow to specific regions. Because to do specific task some part of the brain needs more energy by blood flow. That's how we can specify specific parts of the brain by doing specific tasks.
* But cortex looks pretty much the same all over.
* Because it's not determined genetically which part of the brain to perform which function.
* So in this case if vision part of the brain is damaged, other parts can do the job.
* Early brain damage makes functions relocate. i.e. if specific part of a new-born brain is somehow damaged, the other undamaged parts can do the same work.
* Parallel computing: Cortex is made of general purpose stuff that has the ability to turn into special purpose hardware in response to experience.
* This gives rapid parallel computation plus flexibility.
* So it can learn new functions
* It's more like FPGA (Field Programmable Gate Arrays, it can inherit parallelism) that after it been built a parallel hardware you can put *information* in it to do *specific tasks*
* Conventional computers get flexibility by having stored sequential programs, but this requires very fast central processors to perform long sequential computations.

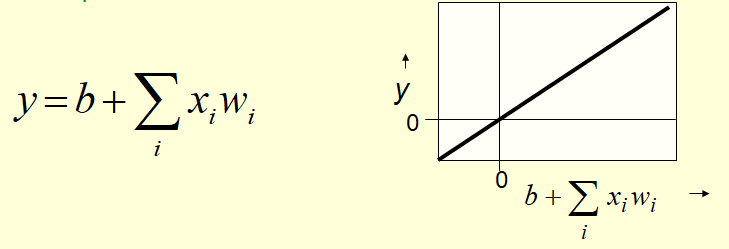
**1.3 Some simple Models Of Neurons**

Now we discuss about some simpler neurons, like: Simple Linear Threshold Neurons. However these are much simpler than the real neurons. But these simple neurons are complicated enough to build a NN that can do some interesting machine learning.

* Idealized neurons: To model things we have to idealize them (e.g. atoms, we may consider it as like solar system). We need to make simplification to understand how the thing works.
* Idealization removes complicated details that are not essential for understanding the main principles.
* It allows us to apply mathematics and to make analogies to other, familiar systems.
* Once we understand the basic principles, its easy to add complexity to make the model more faithful.
* But be careful during idealization, that we do not remove the main principals.
* It is often worth understanding models that are known to be wrong (but we must not forget that they are wrong!)
* E.g. In Deep-Learning neurons communicate with the real values rather than discrete spikes of activity. But it worth to know how the real-neuron inside brain works.
* Linear neurons: These are simple neurons but computationally limited.
* If we can make them learn we may get insight into more complicated neurons.

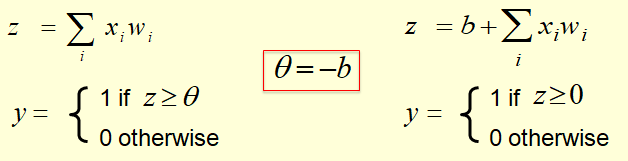


* Output vs weighted activity of the input lines.

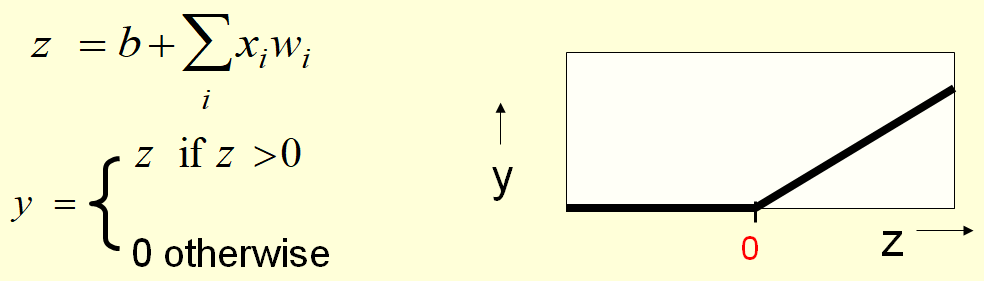


|  |  |
| --- | --- |
| * Binary threshold neurons: It's much more different from the "Linear-Neurons". These kind of neurons are first represented by McCulloch-Pitts (1943): influenced Von Neumann. * First compute a weighted sum of the inputs. * Then send out a fixed size spike of activity if the weighted sum exceeds a threshold. * McCulloch and Pitts thought that each spike is like the truth value of a propositions * So each neuron is combining the truth values against other neurons to produce the truth value of its own * That's like combining some proposition to compute the truth values of another proposition! |  |

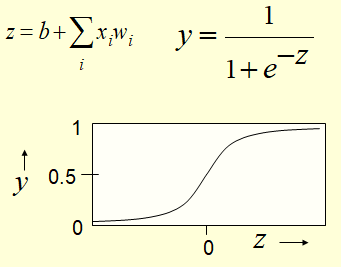
* There are two equivalent ways to write the equations for a binary threshold neuron:



* We set as the threshold in the first case (it makes both cases equivalent). **z** is the totaling point.
* In second case, the **total input** **z** includes the **bias** term **b**.
* Rectified Linear Neurons (sometimes called linear threshold neurons): These kinds of neurons combine the properties of both "Linear-neurons" and "Binary-Threshold-neurons".
* They compute a linear weighted sum of their inputs.
* The output is a non-linear function of the total input.



* Sigmoid neurons: These are the common kind of neurons used in ANN.
* These give a real-valued output that is a *smooth* and *bounded function* of their total input.
* Typically they use the logistic function
* They have nice derivatives which make learning easy (see lecture 3). Smooth derivatives changes continuously.
* If **z** is big and **+ve** we'll get **y=1**.
* For a large **–ve** **z**, **y= 0**
* For total input **0** we'll have **y= 0.5**



|  |  |
| --- | --- |
| * Stochastic binary neurons: These use the same equations as logistic units. They calculate the inputs same way. * But they treat the output of the logistic as the probability of producing a spike in a short time window. Then instead output that probability as a real number, they make a **probabilistic decision** to output either **1** or **0**, they are intrinsically random. * They are treating **p** as probability of producing **1**. * Of course if the **input** is very **big** and **positive** they always **produce 1** and if input is **big** and **negative**, it always **produce 0**. * We can do a similar trick for rectified linear units (RELU): * The **output** is treated as the **Poisson rate** for **spikes**. * Output of **RELU** if the input is above **0** the rate of producing spike is deterministic. * Once we've found that rate of producing spike, the actual time the spike is being produced is a random process, it’s a POISSON process. |  |

So RELU determines the rate but ***intrinsic randomness*** in the ***unit*** determines when the spikes are actually produced.

**1.4 A simple Example of learning**

We gonna look at the very simple learning algorithm for training a very simple network to recognize Handwritten Shapes.

We'll see how the weights evolve, as we run a very *simple learning algorithm*.

* A very simple way to recognize handwritten shapes.
* It's a very simple kind of NN that will learn to recognize digits.
* We'll see how the weights evolve as we run a very simple learning algorithm.

|  |  |
| --- | --- |
| * Consider a neural network with two layers of Neurons. * Input neurons: whose activities represent the intensities of pixels. * Output neurons: whose activities represent the classes. * Neurons in the top layer represent known shapes. * Neurons in the bottom layer represent pixel intensities. * When we show it a particular shape, the output neuron of that shape gets active. |  |

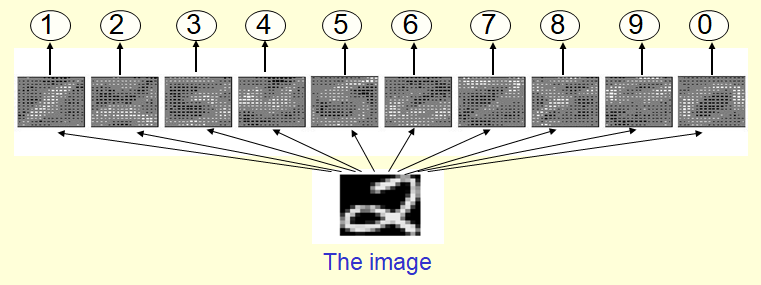
* If a pixel is active what it does:
* It votes for particular shape, assuming that shape contains that pixel
* A pixel gets to vote if it has ink on it.
* Each inked pixel can vote for several different shapes. And votes can have different intensities.
* The shape that gets the most votes wins. So we are assuming there is a competition between the output units.
* How to display the weights:

It seems natural to write the weights between the input-unit and output unit, but we'll never be able to see what's going on visually.

* We need to see the changing values of thousand of weights.
* We draw a little map for each output units.
* In those map we show the strength of the connection coming from input pixel in the location of our input pixels .
* We show the *strength of the connections* by black & white blobs.
* The area representing the magnitude of the weight
* the color representing the sign
* The initial weights are just the random initialization of small weights
* Now what we do is :
* Show that network some data
* And learn to get the weights that are *better than* the *random weights*.

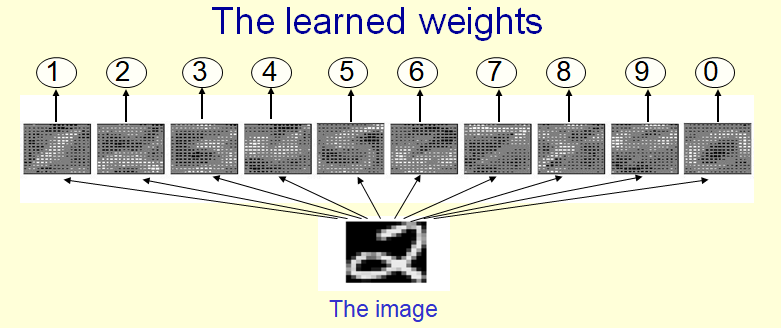
|  |  |
| --- | --- |
| * For each output unit its own “map” of the input image and display the weight coming from each pixel in the location of that pixel in the map. * Use a black & white blob with the area representing the magnitude of the weight and the color representing the sign. |  |

* We need a display in which we can see the values of thousands of weights.
* So the idea is for each output unit, we make a little map. In that map we show the Strength of Connection coming from each input pixel in the location of that input pixel.
* We show the Strength of Connection by using black and white blobs, whose area represents the magnitude.
* The initial weights that you see there are just small random weights.
* How to learn the weights:
* Show the network an image and increment the weights from active pixels to the correct class.
* Then decrement the weights from *active pixels* to whatever class the *network guesses*.
* Now what we are gonna do is: show that network some data and get it to learn weights that are better than the random weights.
* When we show it an image, we are going to increment the weights from the active pixels in the image to the correct class.
* Issue: If we just did that, the weights could get only bigger and eventually every class will get huge input whenever we show it to the image.
* So we need some way of keeping the weights under the control.
* The solution is: we will also gonna decrement the weights from the active pixels to whatever class the network guesses.
* So we really training it to the right thing, rather than it currently has a tendency to do.
* If of course it does the right thing, then the increments we make in the first step, the learning rule will exactly cancel the decrements so nothing will change, which is what we want.
* So, following are the weights that learned from several hundreds of examples shown to them. They started to form regular patterns.
* And after showing them a *few more hundred*, the weights are pretty much at their final values.

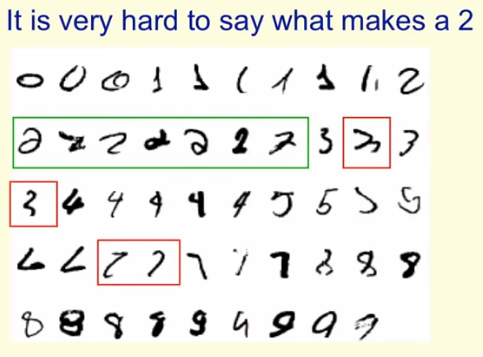


* The Learned weights: We can see that the weights now look like the little templates from the shapes. If you look at the weights going into the **1 unit** for example, they look like a little template for identifying **1's**.
* Actually they are not quite templates e.g. look at the **9 unit**, it don’t have any positive weight below the half way line.
* That's because for telling the difference between **9s** and **7s** the weights below the half way line aren't much used.
* You have to tell the difference by deciding whether there is a **loop** **(9)** at the top or **horizontal bar** at the top (7).

And so, those output units are focused on that discrimination.



* Why the simple learning algorithm is insufficient?
* Since the network is so simple, it's unable to learn a very good way of discriminating shapes.
* What it learns is equivalent to having a little template for each shape.
* And then deciding the winner based on which shape has the template that overlapped most with the ink.
* A two layer network with a single winner in the top layer is equivalent to having a rigid template for each shape.
* The winner is the template that has the biggest overlap with the ink.
* The ways in which hand-written digits vary are much too complicated to be captured by simple template matches of whole shapes.
* To capture all the allowable variations of a digit we need to learn the features that it is composed of.
* The problem is that the weights in which handwritten digits vary are much too complicated to be captured by simple template matches of whole shapes.
* You have to model allowable variation for digits.
* By first extracting features and then looking arrangement of those features.
* Here is examples we have seen already.
* If you look at those **2's** in green box, you can see there is no template that will fit all those well
* It will also will fail to fit that **3** in the red box there.



So task simply can't be solved by a simple network like that. The network did the best it could but it can't solve this problem.

**1.5 Three types of learning**

Here we'll discuss about three different types of machine learning: supervised learning, reinforcement learning and unsupervised learning.

* SUPERVISED LEARNING: In supervised learning, you're trying to predict an output when given an input vector.
* Supervised learning itself comes in two different types. Each training case consists of an input vector x and a target output t.

1. Regression, the target output is a real number or a whole vector of real numbers.

* E.G. the price of a stock in six months time, or the temperature at noon tomorrow.
* And the aim is to (predict) get as close as you can to the correct real number.

1. In classification, the target output is a class label. The simplest case is a choice between one and zero. Or, Between positive and negative cases.

* But obviously, we can have multiple alternative labels as when we're classifying handwritten digits.
* How Supervised learning works: Supervised learning works by initially selecting a model class , that is, a whole set of models that we're prepared to consider as candidates.
* You can think of a model class as a function that takes an input vector and some parameters and gives you an output **y**.
* Learning usually means adjusting the parameters to reduce the discrepancy between the target output, t, on each training case and the actual output, y, produced by the model.
* A model class is simply a way of mapping, an input to an output using some numerical parameters W.
* A model-class, **f**, is a way of using some numerical parameters, **W**, to map each input vector, **x**, into a predicted output **y**. Then we adjust these numerical parameters to make the mapping fit the supervised training data.
* What we mean by fit is *minimizing the discrepancy* between the Target Output on each training case and the Actual Output produced by a *machine learning system*.
* For Regression an obvious measure of that discrepancy, if we're using real values as outputs, is the square difference between the predicted output from our system **y**, and the correct output **t**, We put in that 1/2, so it cancels the 2 when we differentiate.
* For Classification you could use this measure, but there's other more sensible measures which we'll come to later, and these more sensible measures typically work better as well.
* REINFORCEMENT LEARNING: In reinforcement learning, the output is an actual sequence of actions, and you have to decide on those actions based on occasional rewards.
* In reinforcement learning you're trying to select actions or sequences of actions to maximize the rewards you get, and the rewards may only occur occasionally.
* The goal in selecting each action is to maximize the expected sum of the future rewards.
* We usually use a discount factor for delayed rewards so that we don’t have to look too far into the future.
* We say that rewards far in the future don't count for as much as rewards that you get fairly quickly.
* The goal in selecting each action is to maximize the expected sum of the future rewards.
* Reinforcement learning is difficult because the rewards are typically delayed, so it's hard to know exactly which action was the wrong one in a long sequence of actions.
* It's also difficult because a scalar reward, especially one that only occurs occasionally, *does not supply much information*, on which to base the changes in parameters.
* So typically, you can't learn millions of parameters using Reinforcement Learning.
* Whereas supervised learning and unsupervised learning, you can.
* Typically, in reinforcement learning, you're trying to learn dozens of parameters or maybe 1,000 parameters, but not millions.
* UNSUPERVISED LEARNING: In unsupervised learning you're trying to discover a good internal representation of the input.
* For about 40 years, unsupervised learning was largely ignored by the machine learning community. Except for one very limited form called clustering.
* Some widely used definitions of machine learning actually excluded it. e.g in some textbooks, machine learning is defined as mapping from inputs to outputs.
* Many researchers thought that clustering was the only form of unsupervised learning.
* It is hard to say what the aim of unsupervised learning is.
* One major aim of unsupervised learning is to create an internal representation of the input that is useful for subsequent supervised or reinforcement learning.
* You can compute the distance to a surface by using the disparity between two images. But you don’t want to learn to compute disparities by stubbing your toe thousands of times.
* Other goals for unsupervised learning
* It provides a compact, low-dimensional representation of the input.
* High-dimensional inputs typically live on or near a low-dimensional manifold (or several such manifolds in the case of the handwritten digits).
* If you have a million pixels, there aren't really a million degrees of freedom in what can happen.
* There may only be a few hundred degrees of freedom in what can happen.
* So we move from a million pixels to a representation of those few hundred degrees of freedom.
* Principal Component Analysis is a widely used linear method for finding a low-dimensional representation.
* It assumes that there's one manifold, and the manifold is a plane in the high dimensional space.
* It provides an economical high-dimensional representation of the input in terms of learned features.
* Binary features are economical.
* We can represent the input in terms of binary features, that's typically economical because it'll take only one bit to say the state of a binary feature.
* So are real-valued features that are nearly all zero.
* Alternatively we could use a large number of real valued features but insist that for each input almost all of those features are exactly zero.
* In that case for each input we only need to represent a few real numbers and that's economical.
* It finds sensible clusters in the input.
* This is an example of a very sparse code in which only one of the features is non-zero.
* Clustering could be viewed as a very sparse code, i.e. we have one feature per cluster and we insist that all the features except that one are zero and that one feature has a value of **1**.
* So clustering is really just an extreme case of finding sparse features.