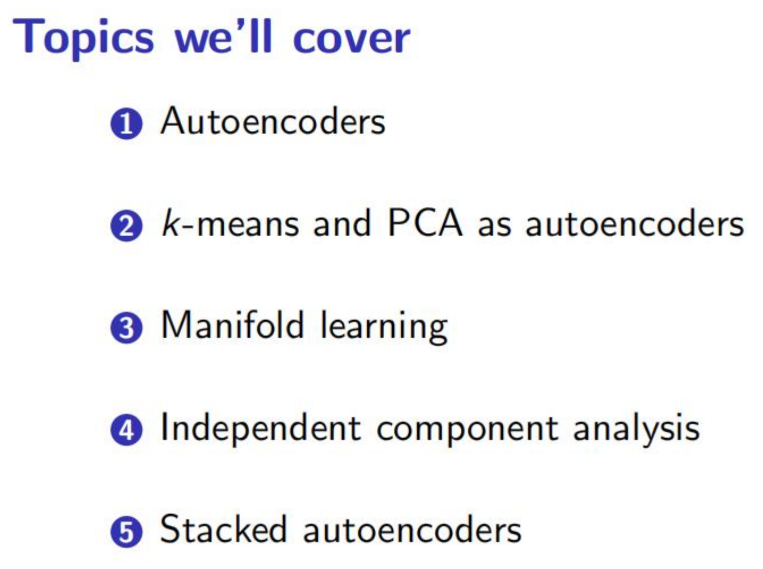
Module 10 - Deep Learning

- 10.1 Autoencoders
- 10.2 Distributed Representations
- 10.3 Feedforward Neural Networks
- 10.4 Training Neural Networks
- 10.5 What We Skipped

总结：

10.1 Autoencoders

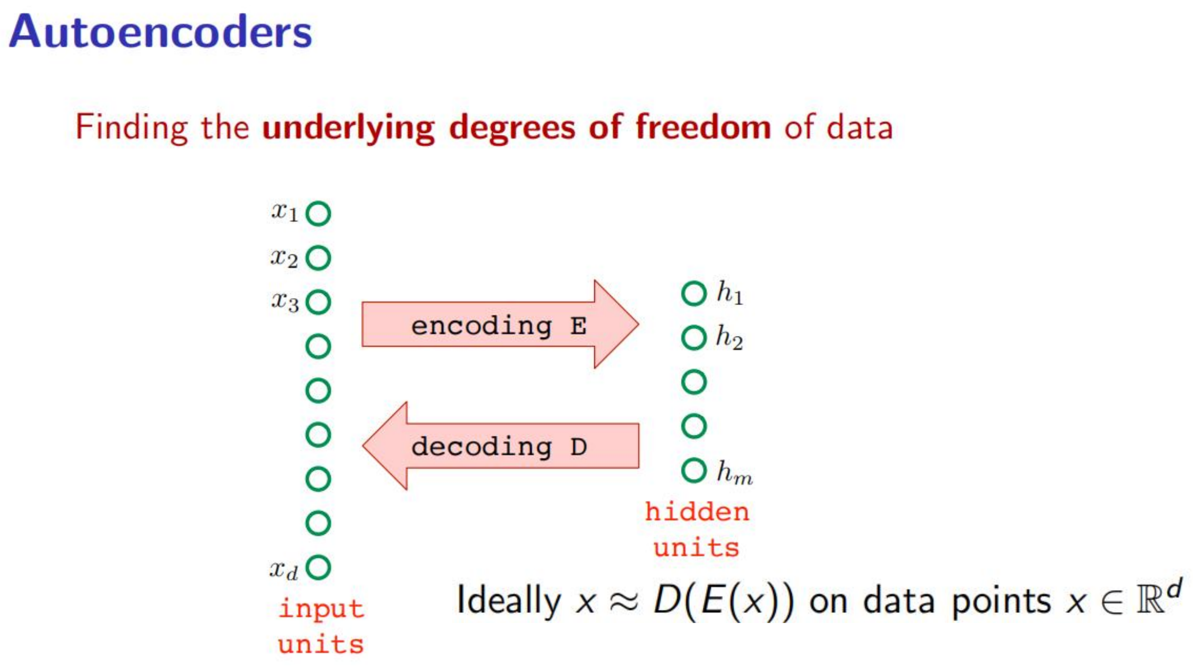


Topics we'll cover

1. Autoencoders
2. *k*-means and PCA as autoencoders
3. Manifold learning
4. Independent component analysis
5. Stacked autoencoders

This week we will be studying deep learning. The models we'll be looking at are neural networks. This is a very expressive class of functions that can capture arbitrary non-linear decision boundaries. Now we've already covered most of the technology that we'll be needing for deep learning. Things like using stochastic gradient descent to minimize a loss function. But there are two additional themes or concepts that would be good to go over before we delve into the nuts and bolts of neural networks. These are autoencoders and distributed representations.

And today we'll talk about autoencoding. We'll see what an autoencoder is and we'll look at how clustering and projection can be expressed in this framework. We'll then turn to some other kinds of autoencoders that are also commonly used, manifold learning, and independent component analysis. Finally we'll end by looking at the prospects for stacking autoencoders in order to arrive at a multi-scale hierarchical representation of data.



Autoencoders

Finding the **underlying degrees of freedom** of data

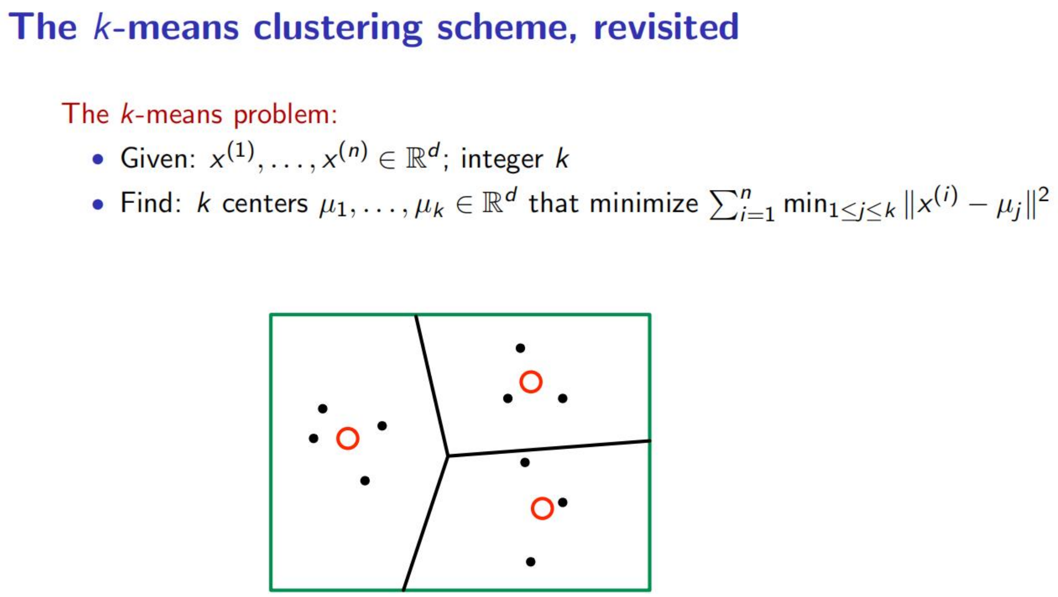
Ideally $x \approx D(E(x))$ on data points $x \in \mathbb{R}^d$

So an **autoencoder** in a nutshell is something that brings out some of the underlying degrees of freedom of data. It produces a representation of data that brings out some of these degrees of freedom and a generic autoencoder is shown over here pictorially. Okay, so on the left we have a data point which is dimensional so it has these features X1 through Xd and on the right we have the representation produced by the autoencoder. This contains m numbers, h1 through hm and these numbers contain some of the signal in the data. Some aspect of the data that is useful to us. Now this representation is something that needs to be discovered, something that needs to be learned. In a search it's often called a **latent representation** or a **hidden representation**.

We'll sometimes refer to the vector on the right as being a latent or hidden vector composed of hidden units, in this case, m hidden units. Now the mapping from the original data point to this latent representation, h1 through hm, can be thought of as an encoding given by some function E. And going in the opposite direction we often have some vector in the latent space and want to send this back into the original space. That can be thought of as a decoding operation given by some function D. Now ideally this autoencoder, this encoding process would not lose too much of the information and the data. (后续)

总结:

So if we take a data point X and then we encode itto get its hidden or latent representation and then we decode that, ideally we'd get back something that's close to the original x. And so this will be an optimality criterian that repeatedly surfaces when we are discussing autoencoders. Okay, all of this has been rather abstract so let's get a little bit more concrete and look at some specific clustering and projection problems that can be expressed in this way.

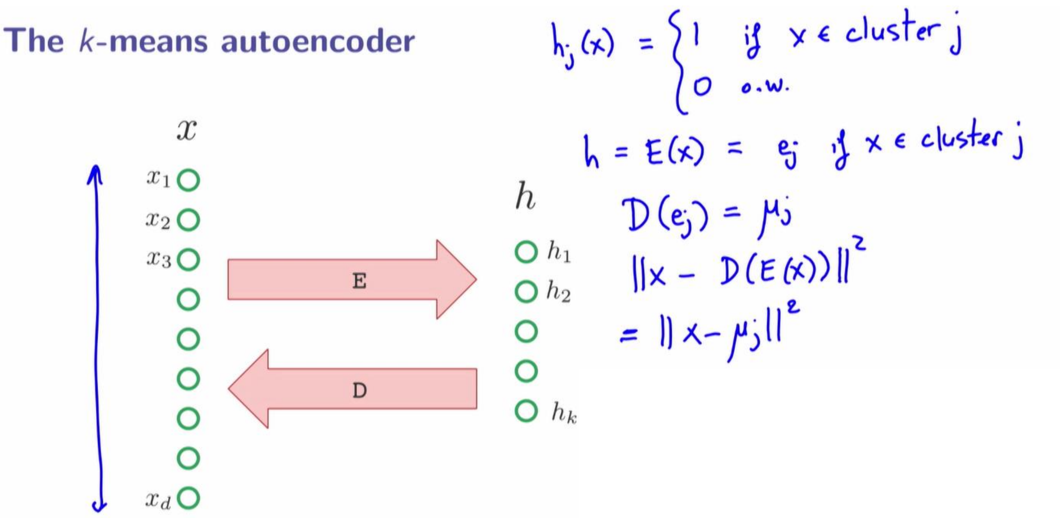


The k-means clustering scheme, revisited

The k-means problem:

- Given: $x^{(1)}, \dots, x^{(n)} \in \mathbb{R}^d$; integer k
- Find: k centers $\mu_1, \dots, \mu_k \in \mathbb{R}^d$ that minimize $\sum_{i=1}^{n} \min_{1 \le j \le k} \|x^{(i)} - \mu_j\|^2$

So we've talked about the k-means clustering problem. What happens here is that we have n data points and we want to group them into k clusters and also to find a representative or center for each of these clusters. So in this picture for example, there are 10 data points and we want to group them into three clusters as well as to find a center for each cluster. So this center is mu one, this red center here is mu two and this center here is mu three. Okay, and we can approximate each of the data points by replacing it with its nearest center. What is the approximation error induced in this way? Well for this point for example, the approximation error is this distance and as is often the case, it will be more convenient for us to work with squared distances. Okay, so in general for the i data point the approximation error is simply the square distance to its nearest center and the goal of k-means clustering is to find the k clusters and the k centers that produced the smallest approximation error possible.



The k-means autoencoder

$$h_j(x) = \begin{cases} 1 & \text{if } x \in \text{cluster } j \\ 0 & \text{o.w.} \end{cases}$$

$$h = E(x) = e_j \text{ if } x \in \text{cluster } j$$

$$D(e_j) = \mu_j$$

$$\|x - D(E(x))\|^2$$

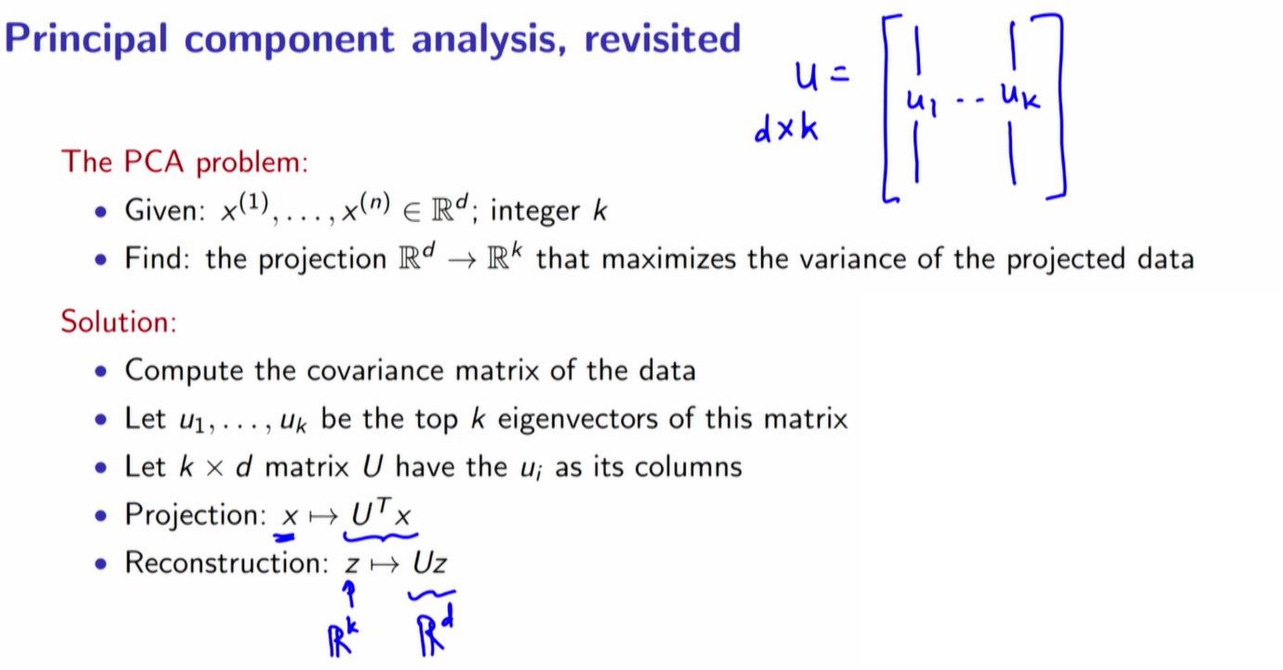
$$= \|x - \mu_j\|^2$$

Now in what sense is this an autoencoder? In this case the left hand side is the original data points. Okay we have d dimensional data points over here. And the latent representation in this case is simply the cluster label. Okay we send each point to one of these clusters, one through k. So one way of doing this is to have these units on the right, h1 through hk represent individual clusters and we could say for example that hj of x, that unit gets turned on if x belongs to cluster j. So hj of x is one if x is in cluster j. And it's zero otherwise. So if we do things this way then the hidden representation is a binary vector. A vector of zeroes and ones and in fact all of it is zeroes except for a one at one position, the position that identifies the cluster that x belongs to.

- So if we wanted to write this in vector form, what we would say is that the vector h is a binary vector. It is the encoding of x, and it is e sub j which means the zero one vector where the one's at position j if x is in cluster j. That's the encoding function.
- Now what about decoding? So all we know is the cluster label. How do we decode? How do we send back to the original space? Well we would just send back to the mean of that cluster, to mu sub j for that cluster. Okay, so the decoding of a specific vector, e sub j is simply mu sub j.
- And what is the error induced by this autoencoder? Well if we take a vector x and we look at the error induced by first encoding it and then decoding it, if we look at that square distance it's exactly the distance from x to its nearest center, mu sub j squared.

Okay, and so what we see is that the k-means solution is actually an optimal autoencoder of this particular type.

总结：



Principal component analysis, revisited

$$\underset{d\times k}{U} = \begin{bmatrix} | & & | \\ u_1 & \cdots & u_k \\ | & & | \end{bmatrix}$$

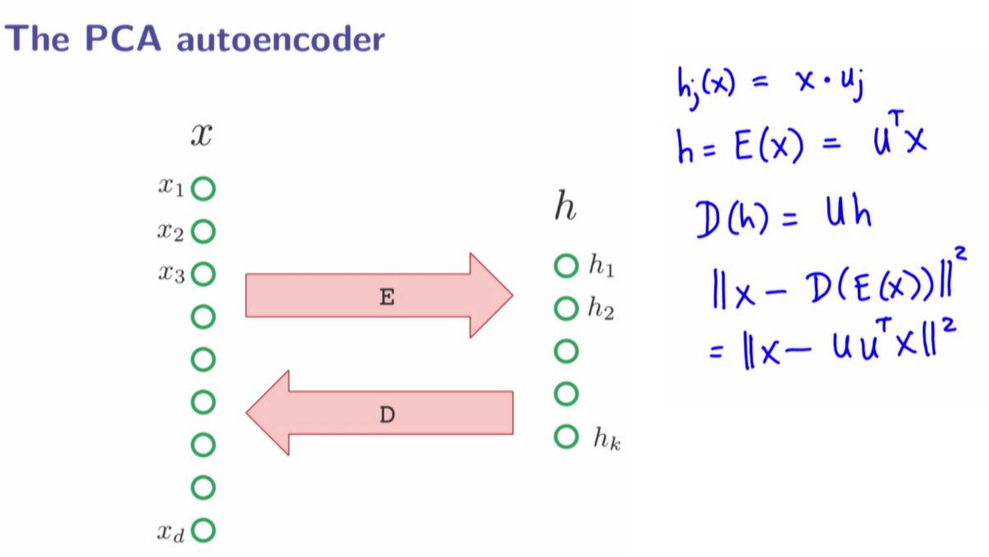
The PCA problem:

- Given: $x^{(1)}, \ldots, x^{(n)} \in \mathbb{R}^d$; integer k
- Find: the projection $\mathbb{R}^d \to \mathbb{R}^k$ that maximizes the variance of the projected data

Solution:

- Compute the covariance matrix of the data
- Let $u_1, \ldots, u_k$ be the top k eigenvectors of this matrix
- Let $k \times d$ matrix U have the u_i as its columns
- Projection: $\underline{x} \mapsto \underbrace{U^T x}$
- Reconstruction: $\underset{\uparrow \atop \mathbb{R}^k}{z} \mapsto \underbrace{Uz}_{\mathbb{R}^d}$

Now let's see how something similar holds to principal component analysis. So in PCA if you'll recall, we have a bunch of data points and what we want to do is to find a projection from the original data space, say Rd to some lower dimensional space, say Rk and we want to find the projection that maximizes the variance of the projective data. And we saw that this can be done quite easily. Okay, there's just a few simple steps to follow. The first thing we need to do is to compute the covariance matrix of the data. A d by d covariance matrix. Then we find the top k eigenvectors of this matrix and those are the directions we project onto. Okay, so if we want to express this in matrix form what we can say is that let's define a matrix u whose columns are these top k eigenvectors, u1 through uk. Okay so this is a matrix that's d by k. And now what we do, the projection sends a vector x in the original space to u transpose x. Now if we have a vector in the projected space and we want to map it back to the original space, that's easy as well. That reconstruction is done by sending a vector z in R sub k to Uz which lies in R sub d. Okay, so that's PCA in a nutshell. Now what kind of autoencoder is this?



The PCA autoencoder

$h_j(x) = x \cdot u_j$

$h = E(x) = u^T x$

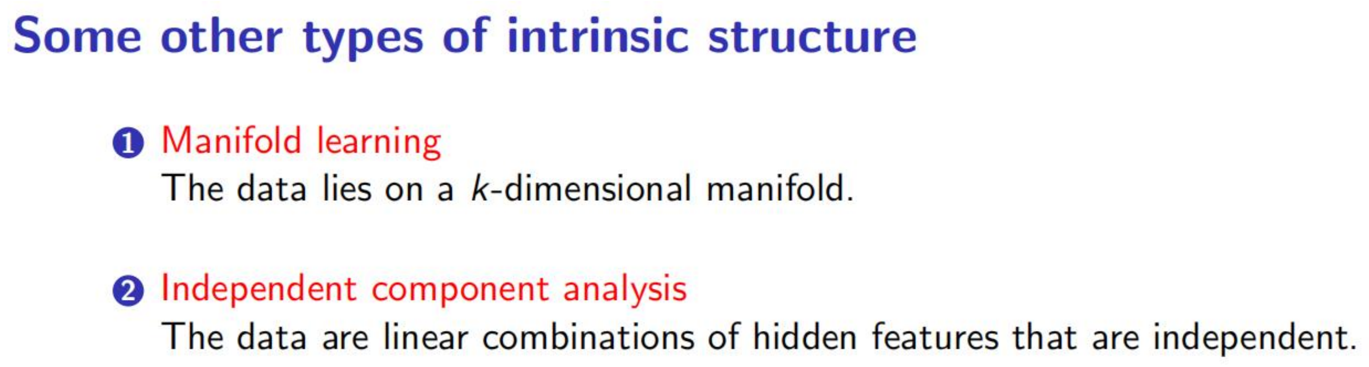
$D(h) = uh$

$\|x - D(E(x))\|^2$

$= \|x - uu^T x\|^2$

So on the left hand side of the autoencoder as usual we have the original data representation. We have x a d dimensional vector. On the right hand side we have k hidden units and in this case these are simply the projections onto different directions. So hj of x is simply the projection of x onto the j eigenvector. It's x.u sub j and if we want to write this in vector form then h which is the embedding of x is simply u transpose x. Now, if we have a k dimensional vector in this latent space and we want to send it back up to the original space, we also saw how to do that. That decoding operation sends h to the matrix u times h. This is clearly an autoencoder and what is the error induced by this encoding and decoding process? Okay so what happens? We take a point x and what we're effectively approximating it by is the result of first encoding it and then decoding it. Okay so this is x minus u, u transposed x. That's squared distance. And it turns out that what PCA is doing is finding the projection that minimizes this reconstruction error. So once again, PCA is an optimal autoencoder of this particular form.

总结：

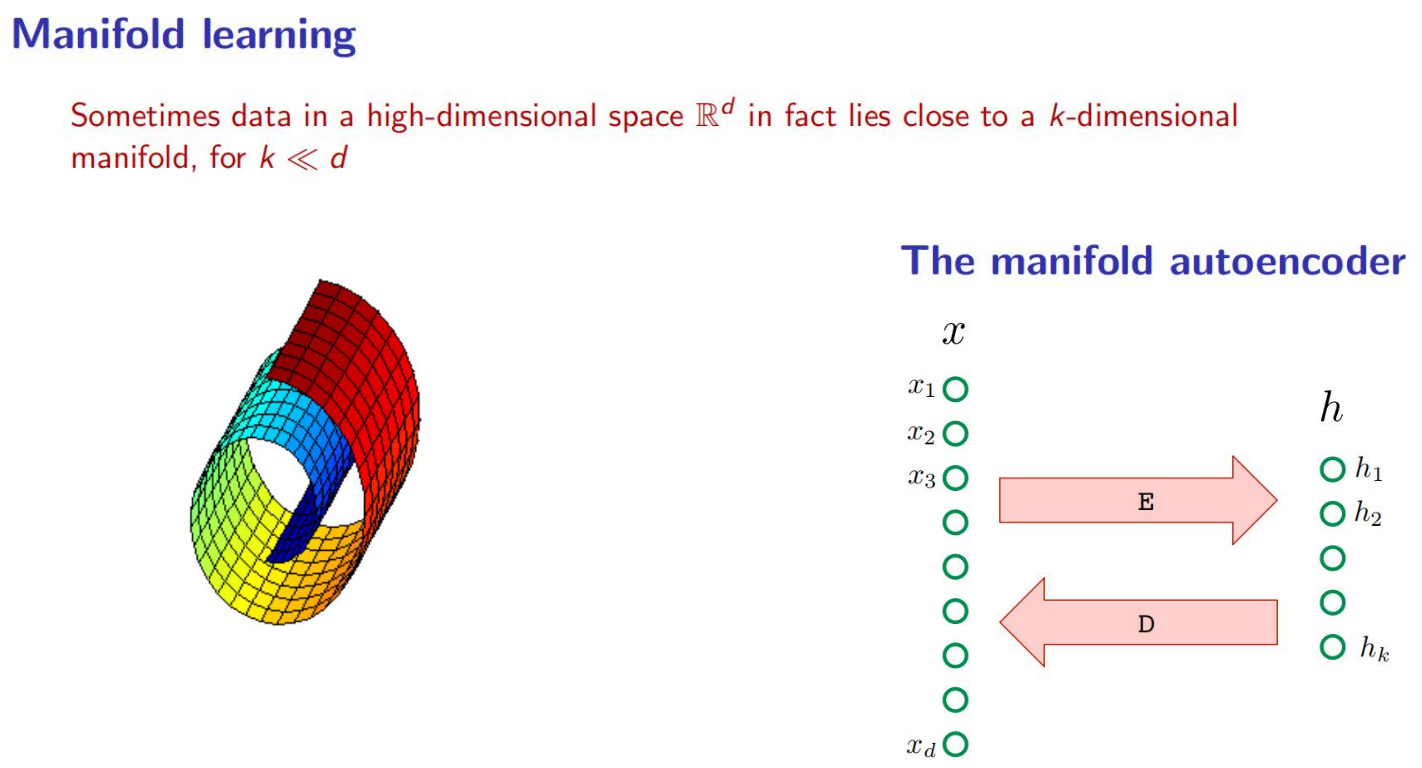


Some other types of intrinsic structure

1. Manifold learning
   The data lies on a k-dimensional manifold.

2. Independent component analysis
   The data are linear combinations of hidden features that are independent.

We've already seen that k-means clustering and PCA fall readily within the autoencoder framework and in fact there are all sorts of other types of unsupervised learning that can be expressed easily in this way. Let's just look at a couple of examples, manifold learning and independent component analysis.



Manifold learning

Sometimes data in a high-dimensional space $\mathbb{R}^d$ in fact lies close to a k-dimensional manifold, for $k \ll d$

The manifold autoencoder

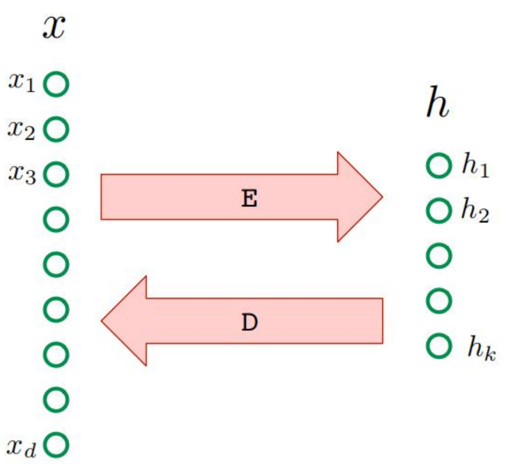
So the idea in manifold learning is that it's often the case that you have data that's high dimensional, there are lots of features, d features but the data points lie on or close to a much lower dimensional surface. They lie on or close to a manifold whose dimension is some number k that's much smaller than d. Okay, we saw one example of this with speech signals. Okay, so as we discussed when you have a speech signal and you want to represent it on the computer, you can make the representation as high dimensional as you like by using lots and lots of different filters, but at the end of the day the speech was produced by somebody's vocal tract and that's a physical system with just a few degrees of freedom. So it's the sort of case where you would have a high dimensional data set that lies close to a low dimensional manifold.
Here is a toy example. Here we have data that's in three dimension so maybe higher, but the data all lies on this two dimensional surface, this thing that is a Swiss roll. So the question is if we are giving data of this kind can we automatically determine the surface and can we map the higher dimensional representation into the two d coordinates shown over here by grid lines? Can we in a sense unroll this surface? Now there are several algorithms that attempt to do exactly this and unfortunately we won't have time to really get into them, but what I wanted to do today is to see how this falls within the autoencoder framework just by ways of another example.

So here is the autoencoder picture again. What we have on the left if the original d dimensional data. D's get represented or d's get sent to the k numbers on the right which are simply the coordinates on the manifold. So the encoding process sends data in the original space to its manifold coordinates and the decoding process goes from the coordinate system on the manifold back to the original space in which the manifold happens to lie. Okay, so it's clearly again an autoencoder.

总结:

Independent component analysis

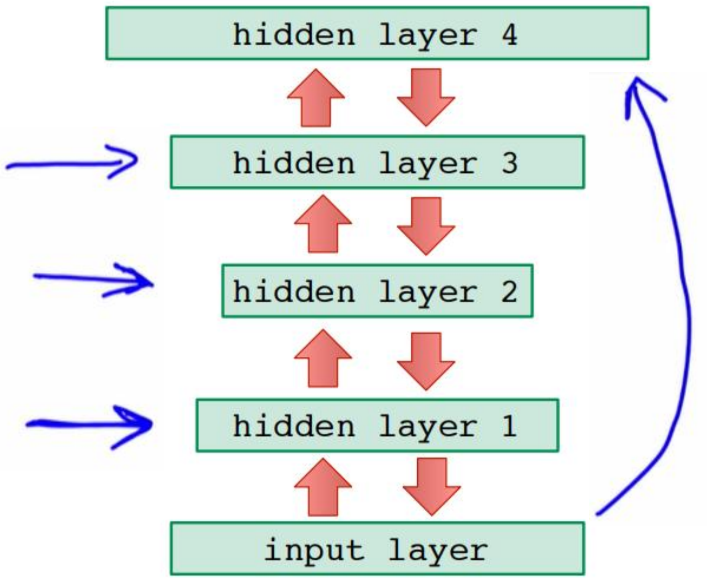
The cocktail party problem



Let's do one more example. This is independent component analysis which is a form of unsupervised learning that has been very popular in signal processing and in neural science. Okay, it's a way of solving among other things the **cocktail party problem**. So when we are at a party and there are lots of people who are talking and mingling, at any given time there are a lot of different conversations that are reaching our ear. What we're hearing is a mixture of all these different threads of conversation. How are we able to sort these out and somehow focus on just one conversation? Maybe the person we're speaking to or maybe some other conversation somewhere else in the room that we're interested in. How do we un-mix all of these conversations? Okay, now if we were trying to get a computer to do this what we might do is to set up d microphones around the room at different locations in the room and each microphone receives a sound signal which is a mixture of a bunch of different conversations.

So that's what's shown on the left over here. We have d input units, one per microphone and each of these units is receiving a mixture of different conversations. It's a time series of speech data. Let's say that there are actually only k distinct conversations taking place. What we would like to do is to take these d mixed up sound signals and recover the k pure conversations, one from each source. That's the right hand side of the autoencoder. And the process of doing this is called **independent component analysis** or in this case sometimes **blind source separation**. And clearly, it's another form of autoencoding.

Stacked autoencoders



- Fit one layer at a time to the previous layer's activations
- Then fine-tune whole structure to minimize reconstruction error

Now, the current revolution in deep learning was in some sense spurred when researchers found ways of stacking autoencoders. Of somehow using multiple autoencoders hierarchically in order to get a multi-scale representation. Okay so the idea here is that at the bottom you have the original data, that's the input layer and then you apply an autoencoder in it to find some structure in the data. To pull out some interesting structure in the data. Then you apply an autoencoder to that to find structure in the structure. Okay so in other words, a sort of higher level structure and then even higher leveled structure and so on and so forth until you get a hierarchy of features that are at an increasingly higher level of generality.

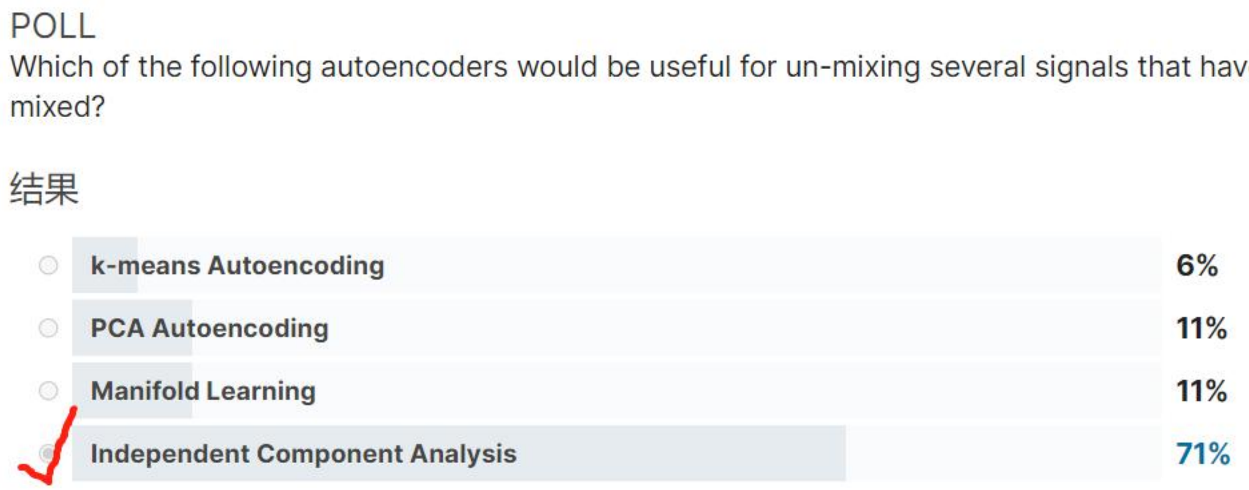
For example imagine that the input layer is an image. Then the first hidden layer over here might consist of some very low leveled structure, tiny fragments of edges or little blobs. The next layer might contain structure at just a slightly higher level, bonafide

总结：

edges for example, and then as you go higher up the chain you get things like little boxes or other standard shapes of which more complicated objects are constituted. Indeed the human visual cortex has hierarchical processing of this kind and that's been an inspiration in designing models of this sort.

So how could we train a model like this? How could we automatically learn this kind of stacked autoencoder?
- Well one very natural approach is to do it little by little, bottom up. Okay, so we start with the input level and then we learn an autoencoder that gets us the first hidden layer. Then we think of the values in that hidden layer as being the new inputs. And we fit another autoencoder that gives us the second hidden layer and we get the third hidden layer and so on. And in this way we get the entire hierarchy at which point we can think of the mapping from the input to the final layer as also being an autoencoder and we can then find him there.
- Now we've been talking about doing this in a purely unsupervised manner(上面这种方式), but it turns out that given the current state of technology, things work out better if we actually do this in a supervised way with a lot of label data and that's what we'll be talking about later when we get to feedforward neural networks.

Okay, so that's it for our treatment of **autoencoders**. We've seen how autoencoding is **a very general framework** that lets us unify many types of unsupervised learning. Things like clustering and projection. It's a very useful framework.
- First of all, it lets us identify similarities and differences between these different kinds of representational learning.
- Second, it also suggests ways in which we can marginally combine different types of representation learning as we saw with stacked autoencoders.



POLL

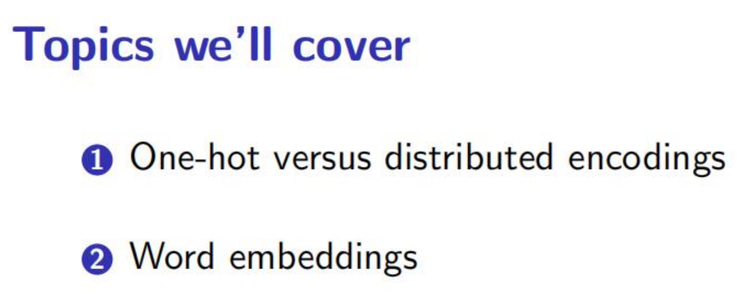
Which of the following autoencoders would be useful for un-mixing several signals that have been linearly mixed?

结果

○	k-means Autoencoding	6%
○	PCA Autoencoding	11%
○	Manifold Learning	11%
✓ ●	Independent Component Analysis	71%

总结：

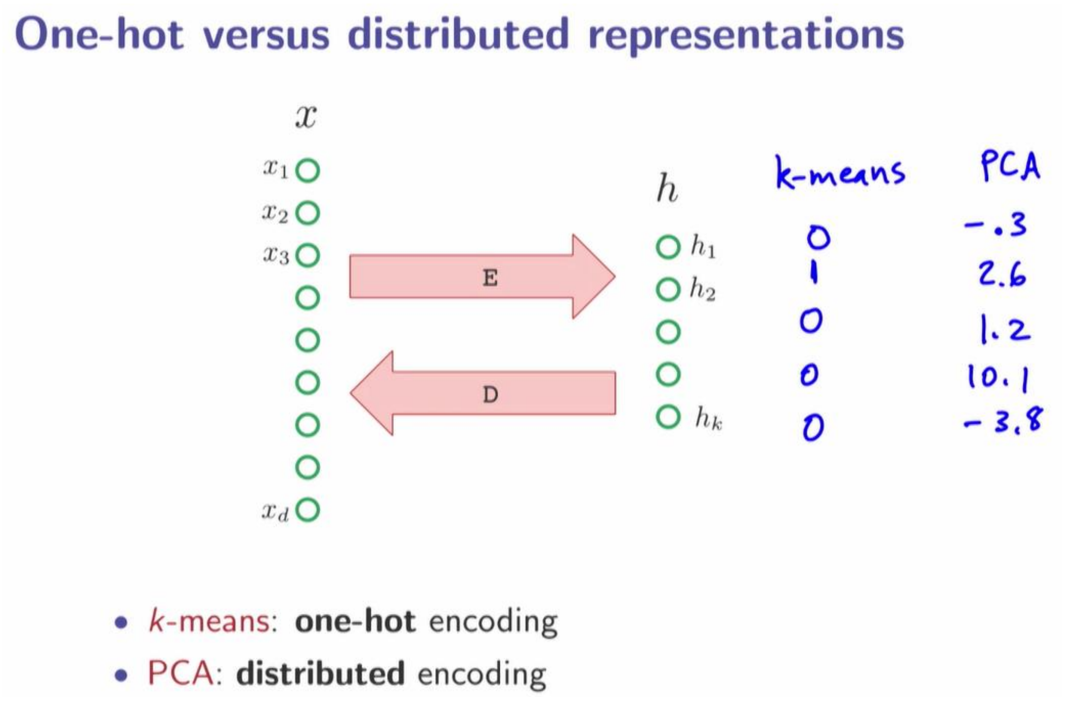
10.2 Distributed Representations



Topics we'll cover

1. One-hot versus distributed encodings
2. Word embeddings

In studies of the brain, and in studies of artificial neural networks, a central theme is **distributed representation**. Today we'll look at what this means. So we'll start by introducing distributed encodings by contrasting them with one-hot encodings. Then we'll look specifically at how a distributed encoding can be constructed for words.



One-hot versus distributed representations

- *k*-means: **one-hot** encoding
- PCA: **distributed** encoding

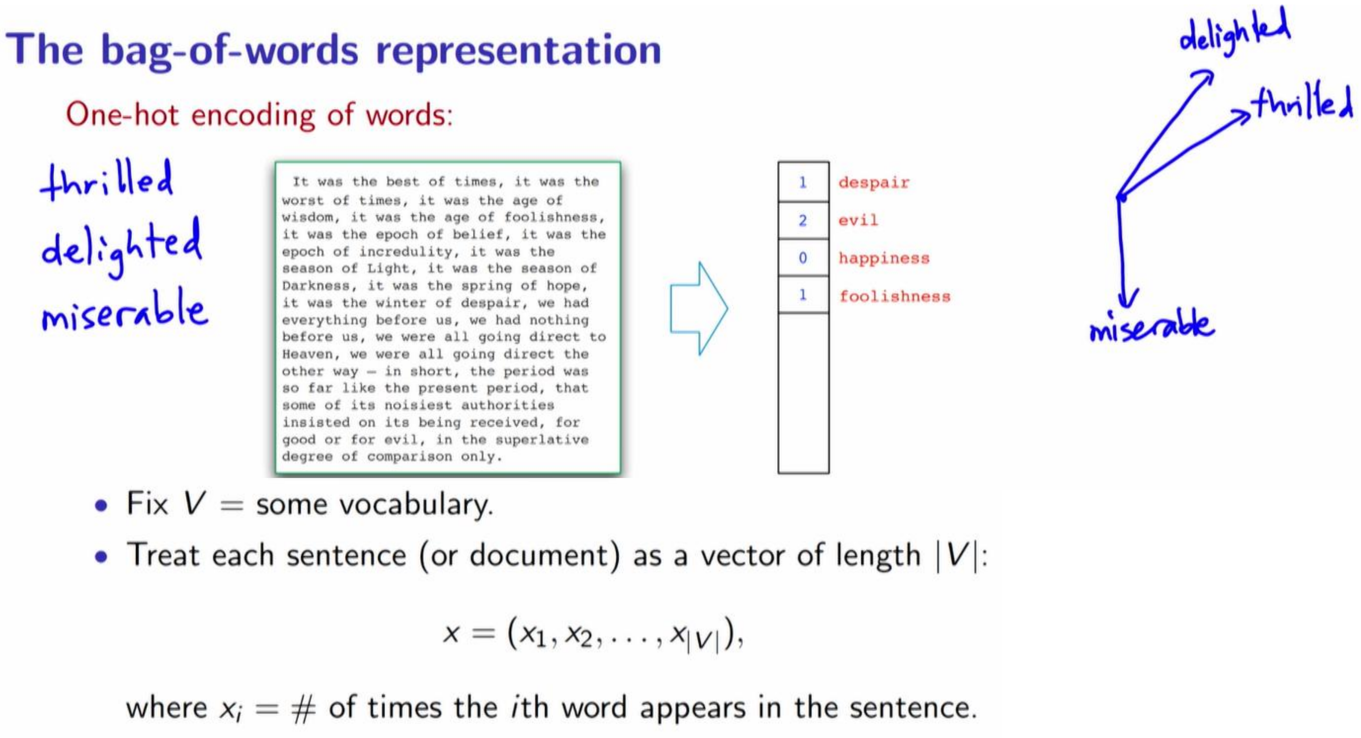
So last time we saw how both k-means clustering and principle component analysis can be treated within the framework of other encoders. It was interesting how two rather different forms of unsupervised learning could be unified in this way. It turned out, however, that the encodings they produce are really fairly different. Let's see how specifically.

Let's start by looking at k-means. On the left we have the data point and the hidden representation, the latent encoding, that k-means produces is the cluster label for that point. Now, if a point x lies in cluster j, what we get is a binary factor that is all zero except for one position j. For example, let's say point x lies in cluster number two. Then in k-means, the latent representation on the right will be zero, one at position two, and then a bunch of zeros at all the other positions. This is the k-means encoding. This is a **one-hot encoding**. It's all zeros except at one position.

The encoding produced by PCA, by contrast, is a dense encoding. It takes a point x and maps it to its projections onto k different directions. So it also produces k numbers but now these numbers are dense. This is a **distributed encoding**. The information in x is spread out, or distributed, amongst these k numbers, typical encoding might be something like, - .3, 2.6. Something in this form.

So this is the distinction between a one-hot encoding and a distributed encoding. It's rather a curious distinction.

总结：



The bag-of-words representation

One-hot encoding of words:

thrilled
delighted
miserable

It was the best of times, it was the worst of times, it was the age of wisdom, it was the age of foolishness, it was the epoch of belief, it was the epoch of incredulity, it was the season of Light, it was the season of Darkness, it was the spring of hope, it was the winter of despair, we had everything before us, we had nothing before us, we were all going direct to Heaven, we were all going direct the other way – in short, the period was so far like the present period, that some of its noisiest authorities insisted on its being received, for good or for evil, in the superlative degree of comparison only.

1	despair
2	evil
0	happiness
1	foolishness

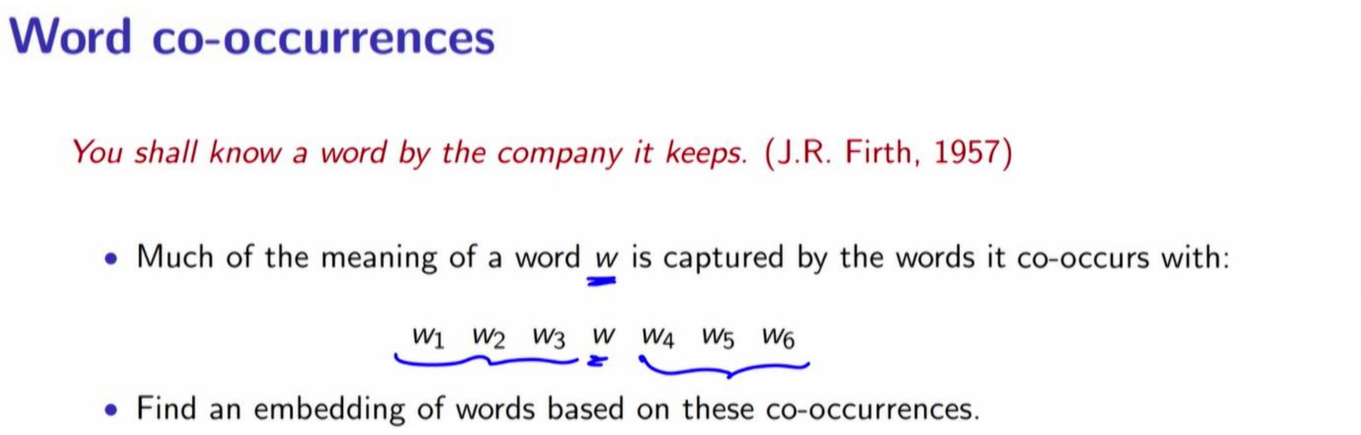
- Fix V = some vocabulary.
- Treat each sentence (or document) as a vector of length $|V|$:

$$x = (x_1, x_2, \ldots, x_{|V|}),$$

where x_i = # of times the ith word appears in the sentence.

Let's see how this plays out in the case of words. So we've talked a little bit about the bag-of-words representation of documents. The way we construct this is that we first start by fixing a vocabulary. Say the 10,000 most common words. We then represent a document by a 10,000 dimensional vector which has a position for each of the chosen words, and the value of that position is the number of times that word occurs in the document. Now how would a single word be represented under this scheme? Well, it would be represented by a 10,000 dimensional vector which was zero everywhere, except for a one at the coordinate corresponding to that specific word. In other words, a one-hot encoding. Now this encoding has got some disadvantages. For example, consider these three words: thrilled, delighted, and miserable. Two of these words are very similar in meaning, they're almost synonyms, while the third word is almost an antonym. But this is not reflected anywhere in the encoding. To put it differently, the encoding seems insensitive to the semantics of words, and this cannot be a good thing.

Is there a different encoding? Is there a way to assign words to vectors, to have a vector for each word, such that words with similar meanings have vectors that are close together? For example, for these three words, we'd have a vector for each of them. Let's say the vector for "thrilled" looks like this. We'd want the vector for "delighted" to be close to that vector. Maybe something like this. And we'd want the vector for "miserable" to be far away from those. Maybe something like this. Is there a way to construct these sorts of embeddings of words? It turns out there are several different ways to do this and some of them are quite sophisticated. What we'll do today is look at one very simple recipe for producing such word embeddings.



Word co-occurrences

You shall know a word by the company it keeps. (J.R. Firth, 1957)

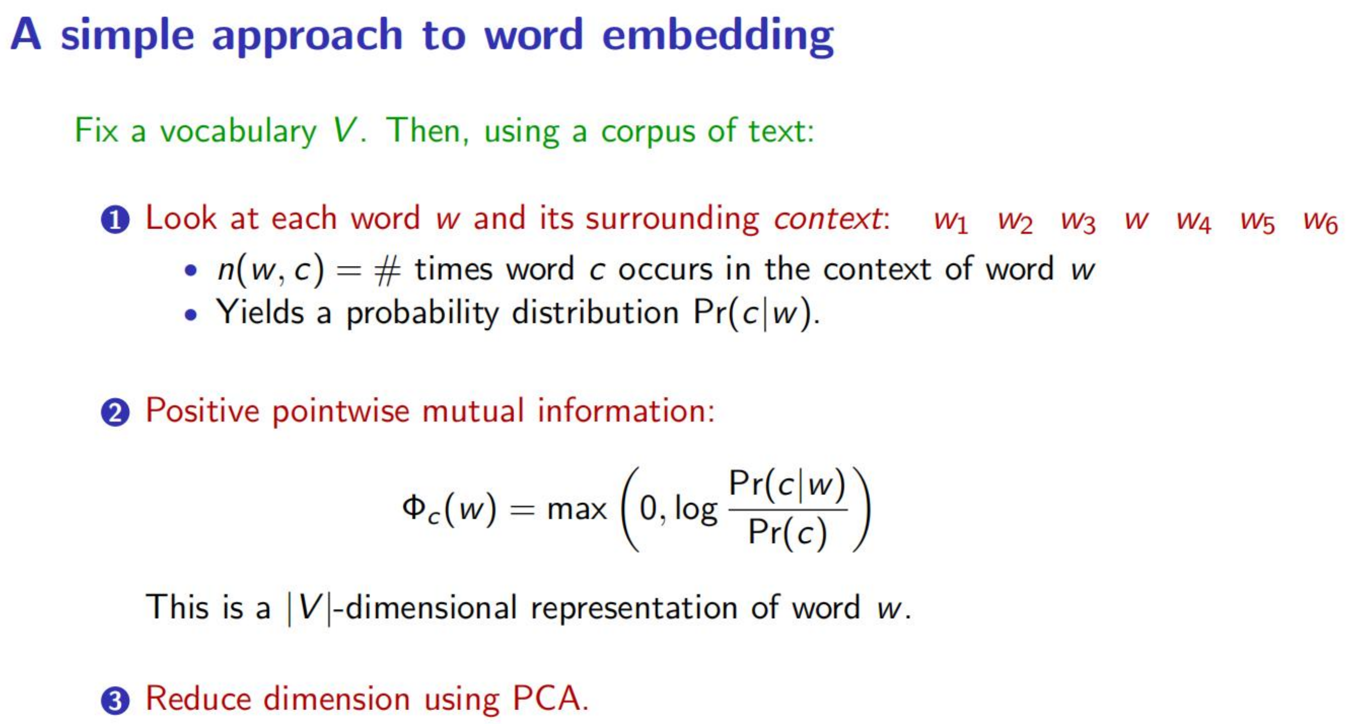
- Much of the meaning of a word $\underline{w}$ is captured by the words it co-occurs with:

$$\underbrace{w_1 \quad w_2 \quad w_3}\quad \underline{w} \quad \underbrace{w_4 \quad w_5 \quad w_6}$$

- Find an embedding of words based on these co-occurrences.

So the general idea in word embeddings is captured by this famous quotation from the British linguist, J.R. Firth: "You shall know a word by the company it keeps." What this means is that the meaning of a word can, to a large extent, be captured by the words that it co-occurs with, by the other words that occur in its **vicinity**. So let's pick a particular word and call it "w". Now let's look at all occurrences of this word in some corpus of text, say, "The Collected Works of Shakespeare". So we look at everywhere this word "w" occurs. The company it keeps refers to words in its immediate vicinity, words that lie in some window around "w". We can define this window however we like. For example, we can choose a very narrow window consisting of the three words before and the three words after. So here is "w" and these are the context words that occur in the vicinity of "w". What we would like then, is to assess the distribution of these context words for each "w" and to find a vector for words, so a vector for each word "w", that reflects this context information. Two words that have similar context distributions should also have vectors that are close together.

总结:



A simple approach to word embedding

Fix a vocabulary V. Then, using a corpus of text:

1. Look at each word w and its surrounding *context*: $w_1 \quad w_2 \quad w_3 \quad w \quad w_4 \quad w_5 \quad w_6$
   - $n(w, c) = \#$ times word c occurs in the context of word w
   - Yields a probability distribution $\Pr(c|w)$.
2. Positive pointwise mutual information:

$$\Phi_c(w) = \max\left(0, \log\frac{\Pr(c|w)}{\Pr(c)}\right)$$

   This is a $|V|$-dimensional representation of word w.

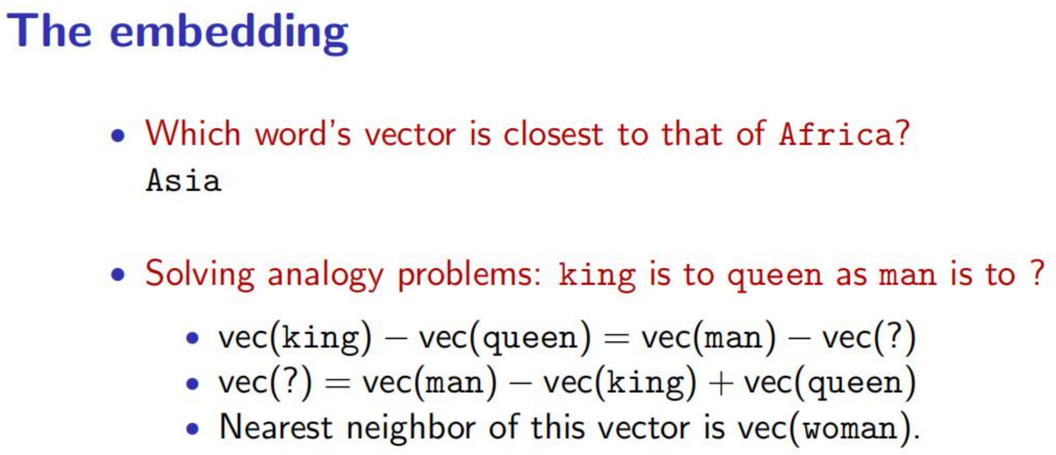
3. Reduce dimension using PCA.

So let's look at a very simple and specific way in which this might be done. So the first step is to fix a vocabulary, call it "v". Maybe a vocabulary of 10,000 words. Maybe something much larger. We also need to fix a corpus of text. We could, for example, use the Brown University corpus of American English, which is a corpus of moderate size, or something that is orders of magnitude larger, like the Wikipedia corpus. Okay, so we have a vocabulary and a corpus. We are gonna construct a vector, an embedding, for every word in the vocabulary, and we're gonna ignore the other words. So look at each word "w" in the vocabulary, go through the corpus, and find all occurrences of "w". For each occurrence, look at the surrounding context. In our case, maybe the three words before and the three words after. By keeping counts of these context words, we can come up with the probability distribution of the nearby words. So the probability that word "c" occurs in the context of "w". This is a probability distribution over all words "c". So it's a vector of dimension "v". It's a way of formalizing the company that "w" keeps. In fact, we could use this entire vector, this v-dimensional vector, as an embedding of "w".

Now it turns out that there are two further tranformations that improve this embedding somewhat. The first is to look at the pointwise mutual information. So what is this? Instead of looking at the probability, the frequency, with which "c" occurs in the context of "w", look at the frequency with which "c" occurs in the context of "w" relative to c's baseline frequency. So this is a form of normalization that's helpful. We can also apply a logarithm to this, which tends to spread out the numbers a little bit. So the result is a vector that's also v-dimensional, if we have a vocabulary of a size 10,000. This is a 10,000 dimensional vector. So for each "w" we get a 10,000 dimensional vector. We could use this as an embedding of "w".

Now the main downside is that this is a very high dimensional embedding. We're representing the word "w" by a 10,000 dimensional vector. Can we reduce the dimension? Yes, for example, we could use principle component analysis. That's a final step that we could try. It turns out that embeddings of just 100, or 200, or 300 dimensions are really very effective at capturing quite a bit about the meanings of words.

Now this is a particular scheme for word embedding that you can try out at home if you want to. But there are also standard word embeddings, word vectors, sets of word vectors, that you can just download, things like Word2vec or Glove.



The embedding

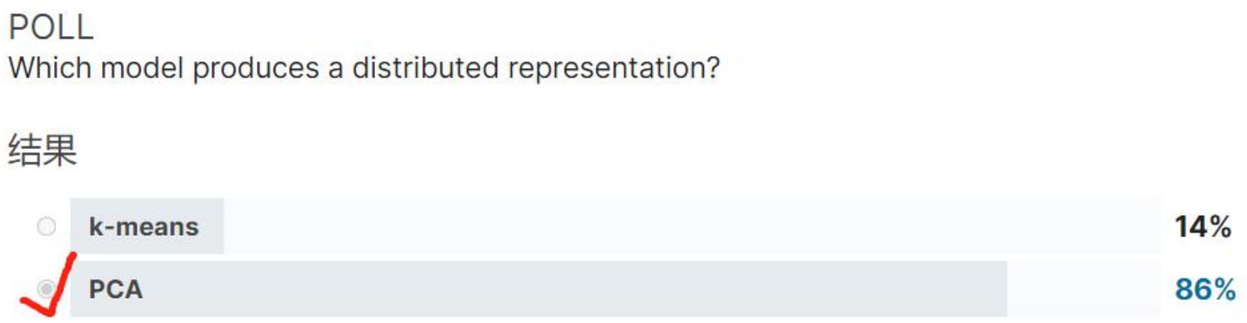
- Which word's vector is closest to that of `Africa`?
  `Asia`
- Solving analogy problems: `king` is to `queen` as `man` is to ?
  - $\text{vec}(\texttt{king}) - \text{vec}(\texttt{queen}) = \text{vec}(\texttt{man}) - \text{vec}(?)$
  - $\text{vec}(?) = \text{vec}(\texttt{man}) - \text{vec}(\texttt{king}) + \text{vec}(\texttt{queen})$
  - Nearest neighbor of this vector is $\text{vec}(\texttt{woman})$.

总结：

So let's finish by looking at a couple of examples that just illustrate the kind of information that's captured in these word vectors. So we construct a word embedding. Now let's look at a word like Africa. Let's look at its vector, some 200 dimensional vector. What is its nearest neighbor? What is the word whose vector is closest to that of Africa? How do we figure this out? Well, using nearest neighbor search on our list of word vectors. If you do that, then the results depend upon the specific embedding that you're using. On the one I have, the result turns out to be Asia. That's perfectly sensible. Both of these words, Africa and Asia, have a high overlap in their context words. So it seems quite sensible. You can try this with a whole slew of other words: communism, war, hysteria, and so on, and the results typically turn out to be quite reasonable and understandable. This is a very reassuring aspect of these word embeddings. The nearest neighbors tend to be something that is really quite sensible.

Now one cute way in which these embeddings have been used is in order to solve analogy problems. So, king is to queen as man is to what? Well, woman obviously. But how can we do this on a computer using word embeddings? Well the way we do this is by writing the analogy as a linear algebraic equation. So what we are saying is that there is some mystery word that we're looking for, and the vector for king minus the vector for queen equals the vector for man minus the vector for the mystery word. What is this mystery word? Well, we have an equation. We can rearrange it. And then we have a precise expression for the vector for the mystery word. We have the vector for man, we have it for king and for queen, so we can compute this expression on the right. Now we look for the word whose vector is exactly that. Well, chances are, there's no word whose vector is exactly that, but once we construct that vector, we can look for its nearest neighbor in the list of word vectors. When you do that, it turns out to be woman.

Okay, so that's it for our discussion of distributed representations. We've looked specifically at the case of embeddings of words and we've seen how this can be done in a way that both captures semantics of words, as well as keeping the dimensionality fairly low. See you next time.



POLL

Which model produces a distributed representation?

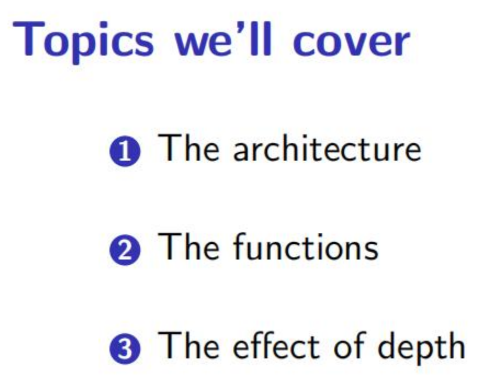
结果

○ k-means 14%

[✓ PCA] 86%

总结：

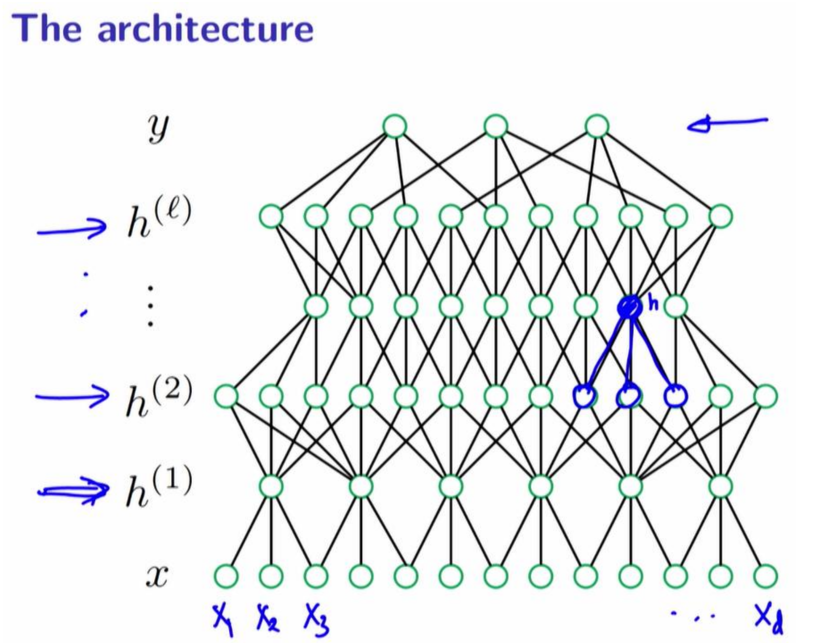
10.3 Feedforward Neural Networks



Topics we'll cover

1. The architecture
2. The functions
3. The effect of depth

Today we will talk about feedforward neural nets. We'll see what these nets look like, how they're parametrized, and what kinds of functions they're able to express.

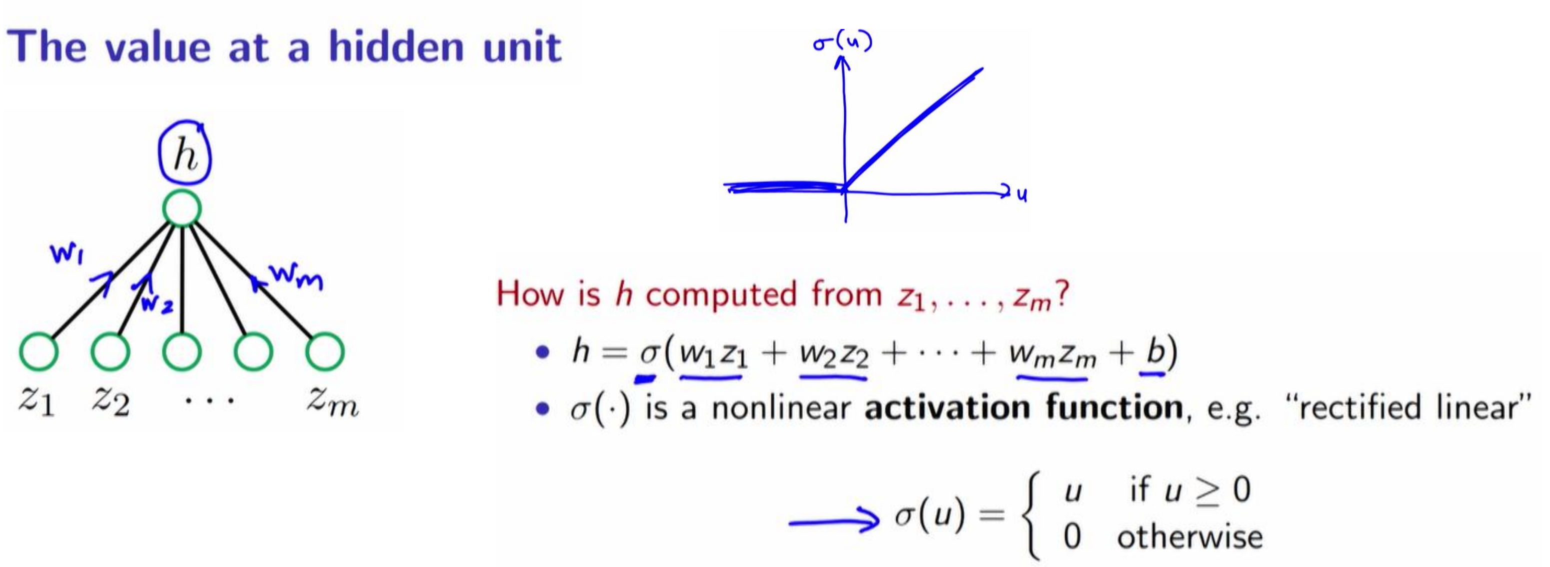


The architecture

So, a feedforward net is inspired loosely by the neural circuitry of the brain. So, generically it looks something like this, a large graph. Each node on the graph is often called a unit, and these units are organized into layers. At the very bottom is the input layer. Okay, so if we are dealing with data that is, say, d dimensional, then this input layer contains d units. One for each of the features of the input x. So, there's a unit for x1, for x2, for x3, all the way to xd. So, the input is presented at the bottom and then various computations are done in a bottom to top pass and the output comes out all the way at the very top. So, what's going on in the middle? Well, the other nodes, the ones that aren't inputs or outputs are called hidden units, and they're organized into layers. Okay, so there's hidden layer number one, then hidden layer number two, and in this picture all the way up to hidden layer number l.

Given the inputs, the first thing we do is to compute the values of the units in hidden layer number one, then we use those to compute the values in hidden layer number two, and so on and so forth until we get to the very top.

So how are these values computed, however? Well, let's look at any internal node in this graph. Let's look at this node for example. It has connections to various nodes on the previous level, in this case three of them, and these are the three connections. We'll call those nodes its parents, and the value of that node is a function of the values of its parents. In particular, once we know the input layer we can simply apply this function to compute the values in the first hidden layer. So, what are these functions exactly? Well, let's pick one node particularly, call it h. How do we get the value of h given the values of its, in this case, three parents?



The value at a hidden unit

How is h computed from $z_1, \ldots, z_m$?

- $h = \underline{\sigma}(\underline{w_1 z_1} + \underline{w_2 z_2} + \cdots + \underline{w_m z_m} + \underline{b})$
- $\sigma(\cdot)$ is a nonlinear **activation function**, e.g. "rectified linear"

$$\longrightarrow \sigma(u) = \begin{cases} u & \text{if } u \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

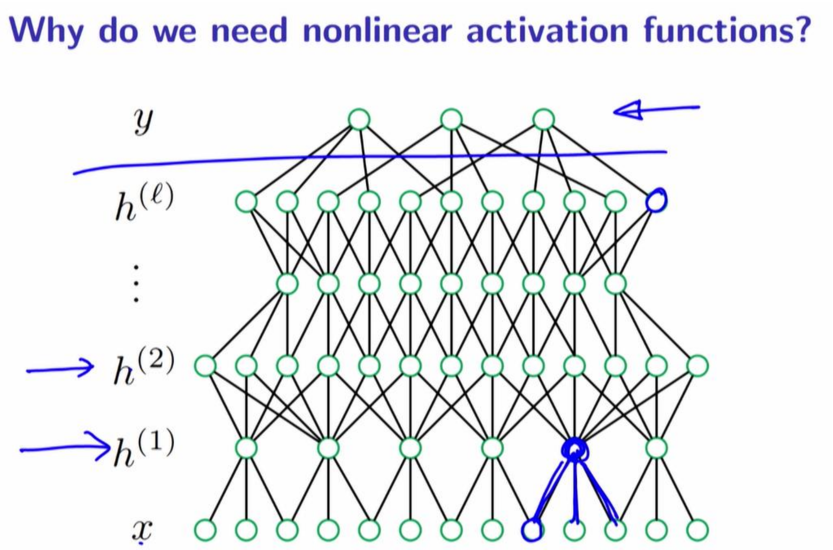
总结：

So let's look at a hidden unit and let's say that it's got n parents. Given the values of those parents, the way we get age is very simple. For the most part it's just a linear function. What we do is we start by computing a linear function of z1 through zm, so a function of the form w1z1, plus w2z2, plus wmzm, plus an offset, b. We compute that linear function, which gives us a real number, and again, at the end we apply a non linear activation function to the answer, and that's what I'm denoting over here by sigma.

What is this activation function, sigma? Well, for instance it could just be the squashing function of logistic recreation, but in the study of neural nets the activation function that's used most commonly is what's called a **rectified linear function**, and it's specified by this simple rule over here. It takes a real number, and if the number's positive it keeps the number. If the number is negative it returns zero. If we were to draw a graph of this it would look something like this. We have a number, u, and we want to compute the rectified linear function of u. If u is negative we return zero, And if u is positive then we just return u. So we get something like this.

Let's summarize... The function that gives us h from its parents, z1 through zm, is parametrized by w1 through wm and by an offset, b. We can in fact imagine that w1 through wm are weights on these edges. W1, w2, all the way to wm, so what we do is we compute z1 times w1, z2 times w2, zm times wm, we add them up, we add b, that's our linear function.

And again, we look at the result, which is some real number. If it's positive we keep it, if it's negative we return zero, and that is h.

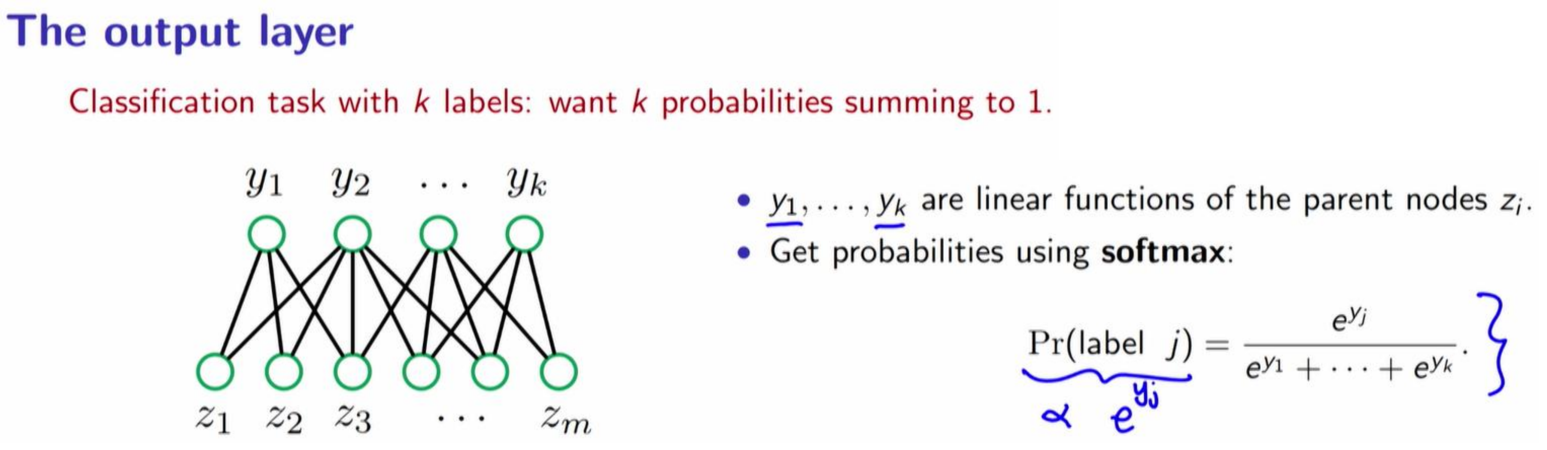


Why do we need nonlinear activation functions?

So this is our graph again, and now we know how to get from any node, from any layer to the next layer. We just compute the value of each node as a function of the values of its parents.

Now, what was the need for having non linear activations? Why not just have linear functions the whole way, why do we need those sigmas? Well, suppose we just had linear functions. In that case, the entire layer h1 would just be a linear function of x and the entire layer of h2 would just be a linear function of h1, and h3 would be a linear function of h2... Now, a linear function of a linear function is, again, a linear function. So in this case, h2 would be a linear function of x and in fact, any node in the entire graph would just be given by a linear function of x. The entire computation of the network could be summarized by a linear function and there would be no need for any hidden units at all. So, in order to make use of a hierarchy like this with multiple hidden units, it's essential to have non linearities of some form in there.

So we've seen how to obtain values at all of the hidden layers, but what happens at the output layer? How are those values obtained? For concreteness let's say that what we're trying to solve is a classification problem in which there are k possible labels.



The output layer

Classification task with k labels: want k probabilities summing to 1.

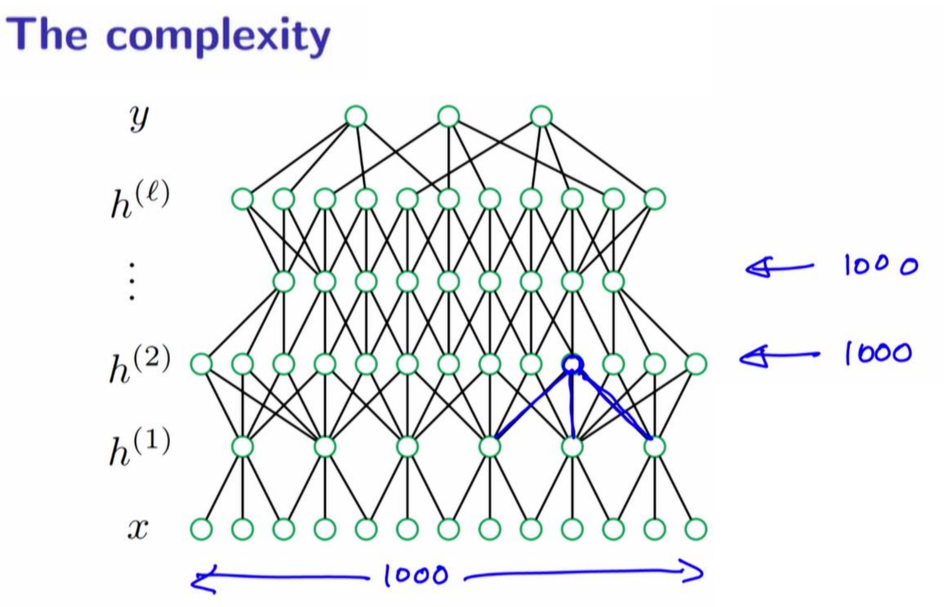
- $\underline{y_1}, \ldots, \underline{y_k}$ are linear functions of the parent nodes z_i.
- Get probabilities using **softmax**:

$$\underbrace{\Pr(\text{label } j)}_{\propto\, e^{y_j}} = \frac{e^{y_j}}{e^{y_1} + \cdots + e^{y_k}}.$$

In that case, what we would do is that we would most likely have k units in the output layer, y1 through yk. One unit corresponding to each possible label and what we want is the probability of each label, so given an input x we would want to know what is the probability of label one giving x, the probability of label 2 giving x, all the way to the probability of label k. So, how do we do this? Well, the first thing we do is that we just let each y be a linear function of its parents. So, we end up with k real numbers, y1 through yk, but these could be numbers like minus two point three or six point seven. How do we convert them into probabilities? Well, we do this exactly the same way that we did for multi class logistic regression. We just use a softmax function.
So, after we've computed these values, y1 through yk, in order to get probabilities what we say is that the probability of a particular

总结：

label, like label j, is proportional to e to the yj. Okay, so the most likely labels are the ones for which the corresponding y value is the largest. And the specific probability is just the normalized version of e to the yj, and e to the yj divided by e to the y1 plus all the way to e to the yk. So this is very familiar, just like logistic regression. In a sense, that last layer of the network is simply implementing multi class logistic regression.

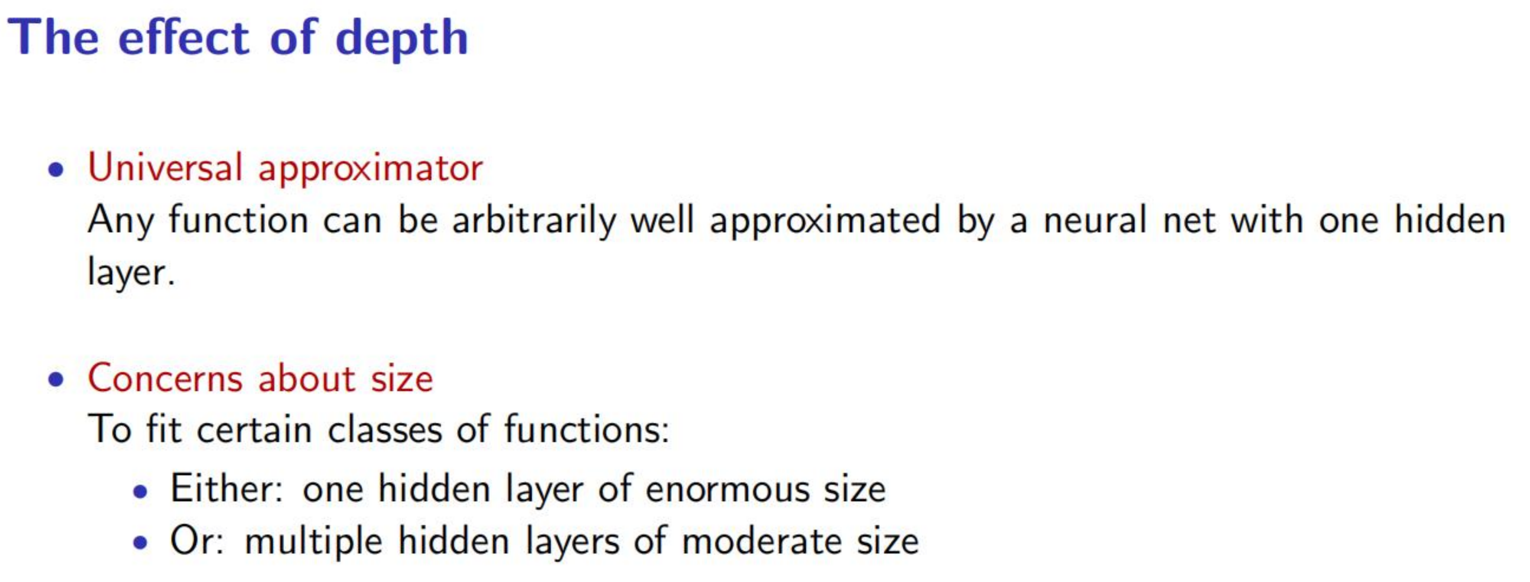


The complexity

Okay, now let's think a little bit about how many parameters there are overall in a neural net like this. So, let's see... We saw that each node is a linear function of its parents plus a non linearity, okay, so if a node for example has got three parents, like this one, then it has its own linear function with four parameters, one from each parent and then an offset, too. So, even if we ignore the offset term there is at least one parameter per edge of the graph. Okay, so the number of parameters in the neural net is at least the number of edges. So, how many edges are there? Well, why don't we do something a little bit concrete. Okay, let's say the input is of moderate size, say 1,000...

Now, the hidden layers could be of any size. There could be hidden layers of size 20 or of size 10,000, okay, so let's just say for concrete, let's say we have two layers that are both of size 1,000. So, a layer with 1,000 nodes and another layer with 1,000 nodes and let's say that they're fully connected. How many edges do we have between them? Well, if they each have 1,000 nodes then the number of edges is 1,000 times 1,000, in other words a million, and that's just one layer of the network. So as you can see, the number of parameters in a net like this can easily be enormous. And this immediately means two things.

- First of all, these networks are extremely expressive. They can express a wide range of different functions.
- Second, it's clear that we're going to need quite a lot of data in order to learn all of these parameters.



The effect of depth

- Universal approximator
  Any function can be arbitrarily well approximated by a neural net with one hidden layer.
- Concerns about size
  To fit certain classes of functions:
  - Either: one hidden layer of enormous size
  - Or: multiple hidden layers of moderate size

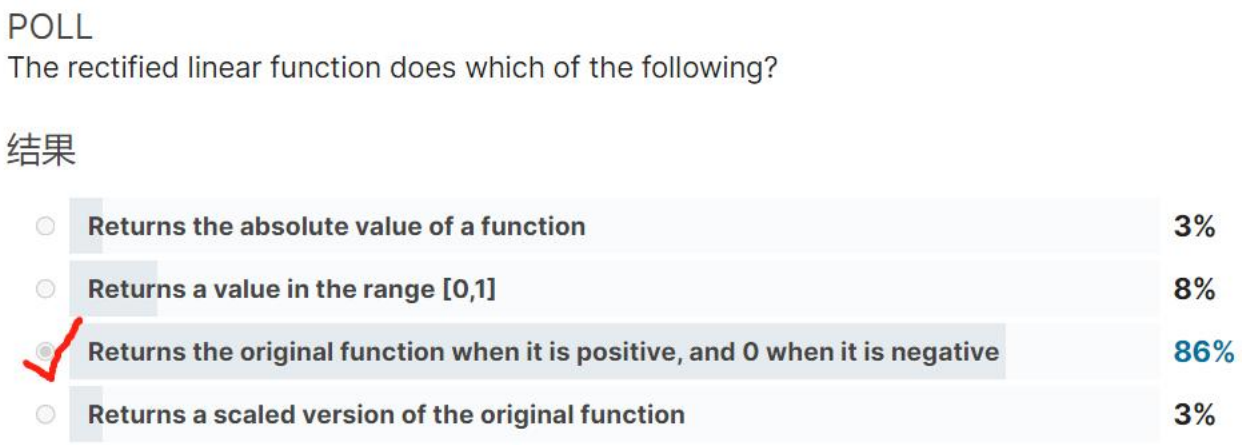
总结：

Let's now talk a little bit about the depth of the network. How does the depth influence the expressiveness of the net, the range of functions that it can compute? So, there's a very well known result, which says that a neural net with just one hidden layer can compute any function to arbitrary accuracy. Okay, so it can arbitrarily well approximate any function at all.

That sounds very powerful, and it also raises the question of why we would ever want to use more than one hidden layer. If we can get any function with just one layer, why would we ever need more layers? So, that's a very good question and it's not something to which there is a completely definitive answer.

- But here's one thing that we can say, so one thing that people have pointed out is that there are certain functions that you can write down and if you... And of course, you can approximate them by a one layer network, a network with one hidden layer, but if you do that, the hidden layer has to be enormous. The number of nodes, the number of hidden nodes in that one layer has to be, for instance, exponential in other problem parameters such as the input dimension.
- But if instead you use a deeper network, then each of the individual layers can actually be of moderate size. So, that's one way in which depth can be quite helpful.

That's good for now. We have seen what a feedforward neural net is and next time we'll talk about how to learn the parameters of a net like this, see you there.



POLL

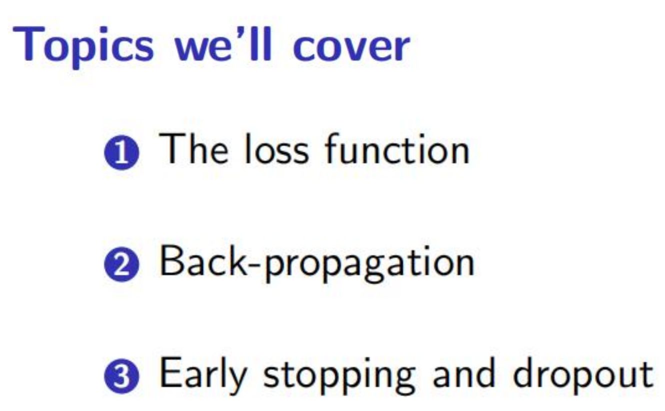
The rectified linear function does which of the following?

结果

○	Returns the absolute value of a function	3%
○	Returns a value in the range [0,1]	8%
[✓ ●]	Returns the original function when it is positive, and 0 when it is negative	86%
○	Returns a scaled version of the original function	3%

总结：

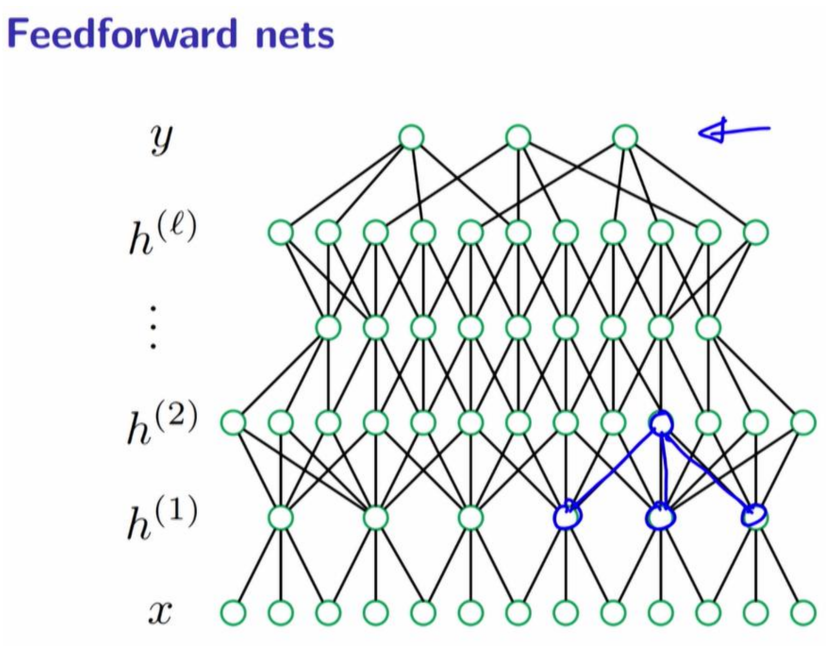
10.4 Training Neural Networks



Topics we'll cover

1. The loss function
2. Back-propagation
3. Early stopping and dropout

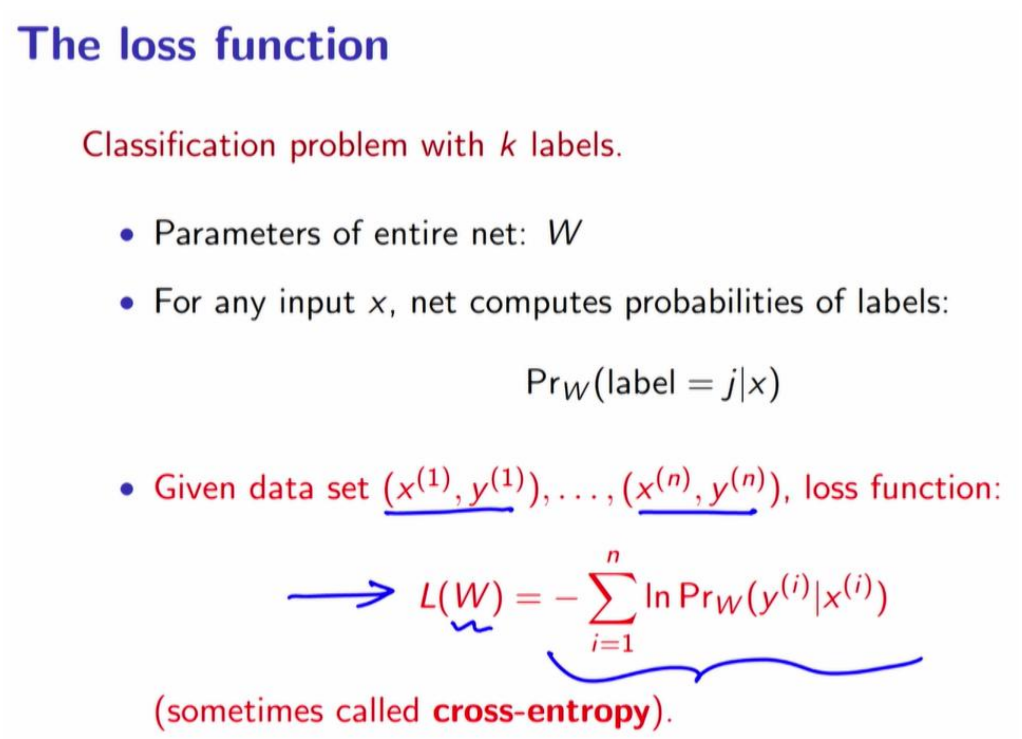
Last time, we introduced the feedforward neural network, a function class of great expressivity. Today, we'll see how to learn these models. We'll start by talking about a suitable loss function for learning neural networks. And we'll see how to set up a stochastic gradient descent algorithm for minimizing this loss function by using a clever scheme called back-propagation. We'll also talk about some tricks that are very helpful in getting a model that generalizes well.



Feedforward nets

Here again is the architecture of a feedforward net. It contains a large number of units that are organized in layers. At the bottom, there is the input layer, where the input vector x is presented. It goes through a computation that starts at the bottom and proceeds all the way to the top, ending in the output layer y. Each node is a simple function of its parents. So we have a node here. These are its parents in the previous layer. Each node is computed as a linear function of its parents, followed by a nonlinear activation, typically a rectified linear function. And in this way, the computations proceed upwards and reach the output.

Now, as we saw last time, networks like this can easily have vast numbers of parameters. How do we learn these parameters using a training set? The approach, as usual, will be to formulate learning as an optimization problem in which the goal is to find the parameters that minimize a suitable loss function.



The loss function

Classification problem with k labels.

- Parameters of entire net: W
- For any input x, net computes probabilities of labels:

$$\Pr_W(\text{label} = j|x)$$

- Given data set $\underline{(x^{(1)}, y^{(1)})}, \ldots, \underline{(x^{(n)}, y^{(n)})}$, loss function:

$$\longrightarrow \quad L(W) = \underbrace{-\sum_{i=1}^{n} \ln \Pr_W(y^{(i)}|x^{(i)})}$$

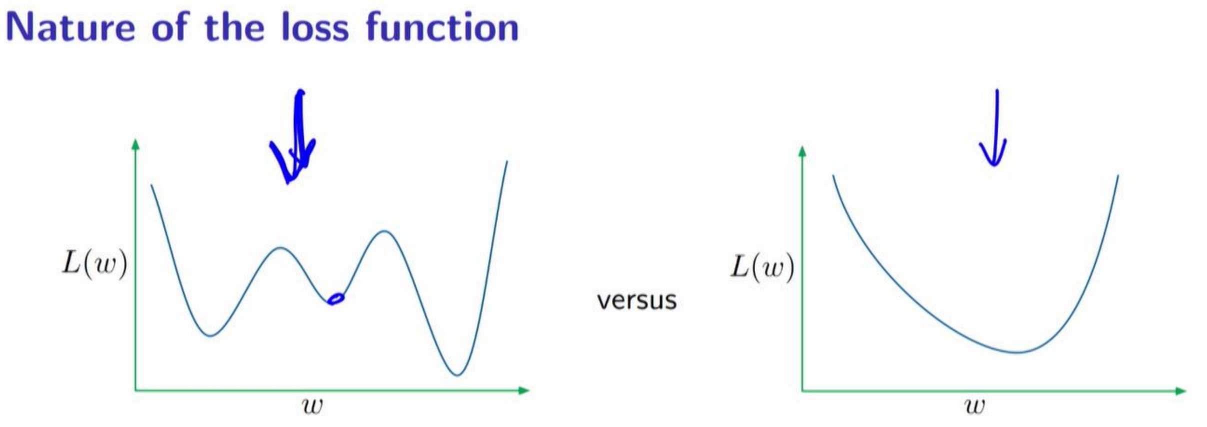
(sometimes called **cross-entropy**).

总结：

So for concreteness, let's say that we are using the neural net to solve a classification problem in which there are k labels. So in this case the final output layer will contain k units, one per label. And we saw last time how we can generate probabilities for each label, essentially by having something like multi-class logistic regression at that very last layer by using a soft max k.

So the parameters we want to learn are the parameters of the entire net, the parameters that specify all the different linear functions, the linear function at each and every different node. Now once we've fixed these parameters, we have a fully-specified net. And what that gives us is for any input x, the probabilities of the various possible labels. This is a lot like logistic regression. And indeed, we can follow some of the same methodology. So for each data point x, we get the probabilities over the k possible labels. What we can do is to look for the parameters W, this vast number of parameters, that maximize the probability of the training set, that is to say, the probability of point one's label times the probability of point two's label, times the probability of point three's label, and so on. We get a cost function. We get this thing we wanna maximize, which is a product of the probabilities of each of the individual data points.

Now as usual, we can take the logarithm of this to make it a sum and put a minus sign in front of it to get something to minimize. And if we do that, this is the loss function we end up with. Given a data set of N training points, we wanna find the set of parameters W, the parameters for the entire net, that minimize this loss function. And if you notice, this is exactly the loss function that we use in the case of logistic regression. It's sometimes called the cross-entropy. Now, in the case of logistic regression, it turned out that this was a convex optimization problem, a very nice problem. Is that still the case when we're dealing with neural nets? Sadly, no. In fact, it is very, very far from convex.



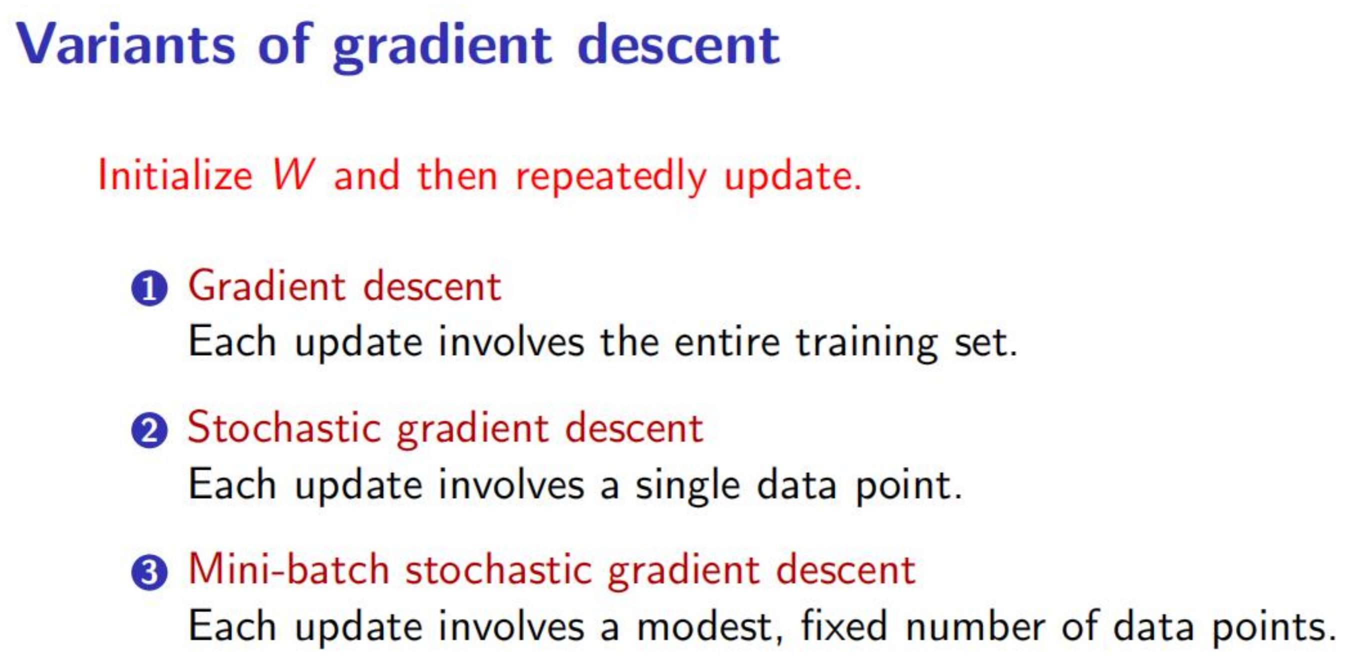
Nature of the loss function

versus

In talking about loss functions, there are two cases that we have always distinguished, the case over here, where the loss function is badly behaved. It has lots of little local optima that we can get trapped in. These local optima are of widely varying quality. So if we end up in the wrong one, then it could be a pretty bad solution. The kind of loss functions we like are the ones that look like this on the right, a convex loss function.

- In fact, almost every problem we've studied in the course so far has had a convex loss function. **Least squares**, **Lasso**, **support vector machines** and so on.
- In fact, the only problem we've studied with this kind of cost function (non-convex，左图) has been **k means**. When we were talking about k means, we noted that it's an n p hard optimization problem. We cannot hope to efficiently find the optimal k mean solution. It turns out, however, that there are algorithms that are guaranteed to efficiently find a solution that is close to the optimal for k means. Unfortunately, **neural net optimization** is a different beast altogether. It is highly non-convex, and there really isn't much hope of even finding something that's necessarily close to optimal. Nonetheless, it turns out that, perhaps because of the extreme expressiveness of these models, we nevertheless very often end up with functions with networks that perform very well.

总结：



Variants of gradient descent

Initialize W and then repeatedly update.

1. Gradient descent
   Each update involves the entire training set.
2. Stochastic gradient descent
   Each update involves a single data point.
3. Mini-batch stochastic gradient descent
   Each update involves a modest, fixed number of data points.

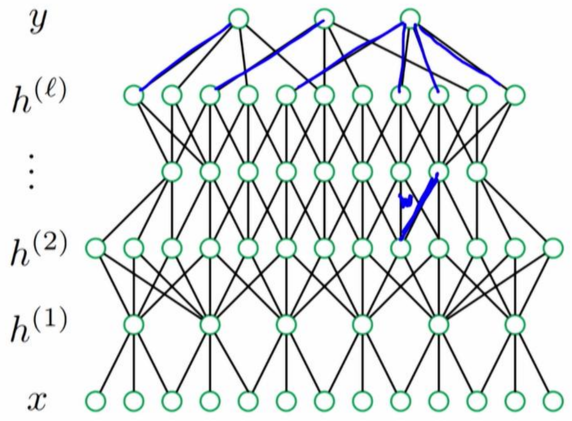
How are we gonna do this minimization? As usual, we will use some sort gradient descent. Now, we've looked at three variants of gradient descent, vanilla gradient descent, stochastic gradient descent, and mini-batch stochastic gradient descent. All three of them start by initializing the parameter W in some way and then doing a bunch of updates, where W keeps getting adjusted and hopefully the loss function l of W is going down steadily the whole time. Now, all three variants are based on using the derivative of the loss with respect to W.

In gradient descent, each individual update involves using the entire dataset. This can be impractical if the dataset is enormous, as it frequently is in the case of neural networks. The other extreme is stochastic gradient descent, where each update involves just one data point. That's much better. But in that case, the updates can be quite noisy. A nice compromise is mini-batch stochastic gradient descent, where the updates don't involve the full dataset nor single point, but they involve a modest number of data points, say, 1,000 data points at a time. We can use any of these. The most common option is some kind of mini-batch stochastic gradient descent.

For any of these, what we need to compute is the derivative of the loss with respect to each of the parameters in the network. And as we know, there are a lot of parameters.

Derivative of the loss function $\frac{dL}{dw}$

Update for a specific parameter: derivative of loss function wrt that parameter.

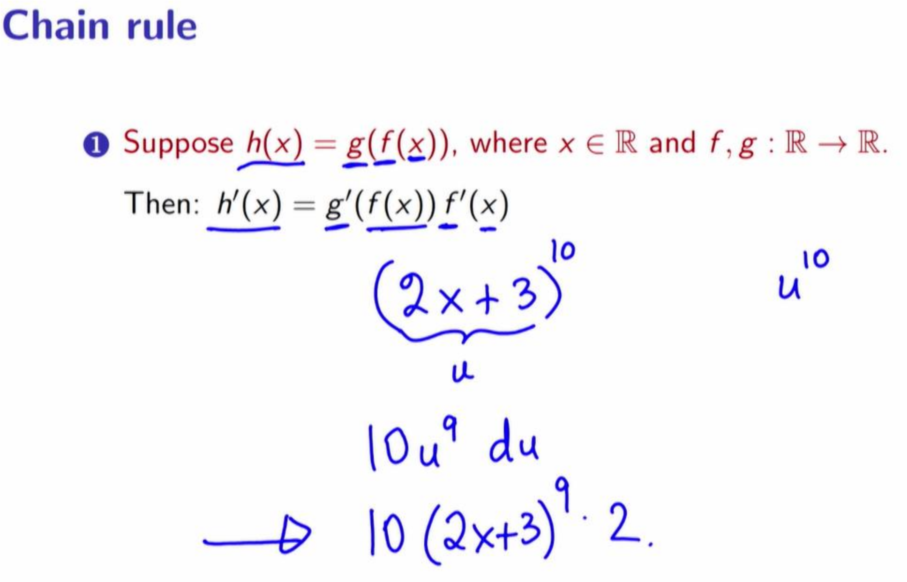


Here's our network. We present the input at the bottom, the x. We get a y at the top. And once we get y, we can assess the loss, the probability of the correct label. So we want the derivative of that loss with respect to every parameter in the network. Now, remember that there is a parameter for every edge in here. So every edge, for example this edge, has got an associated parameter w. We need the derivative of the loss with respect to every such parameter w. Why? Because the way we're gonna update w is using that derivative. We're gonna move w in the opposite direction of that derivative. So how do we compute all these derivatives?

Well, the ones that are easy to compute are for the parameters in the very last layer, the w's along these edges. As we saw, the last layer is really just multi-class logistic regression. And we already know how to do those derivatives. But what about parameters further down the network?、

What about w's like this? How do we compute the derivative of L with respect to that w? It turns out that in order to do so, we rely heavily upon the chain rule of calculus.

总结：



Chain rule

1. Suppose $h(x) = g(f(x))$, where $x \in \mathbb{R}$ and $f, g : \mathbb{R} \to \mathbb{R}$.

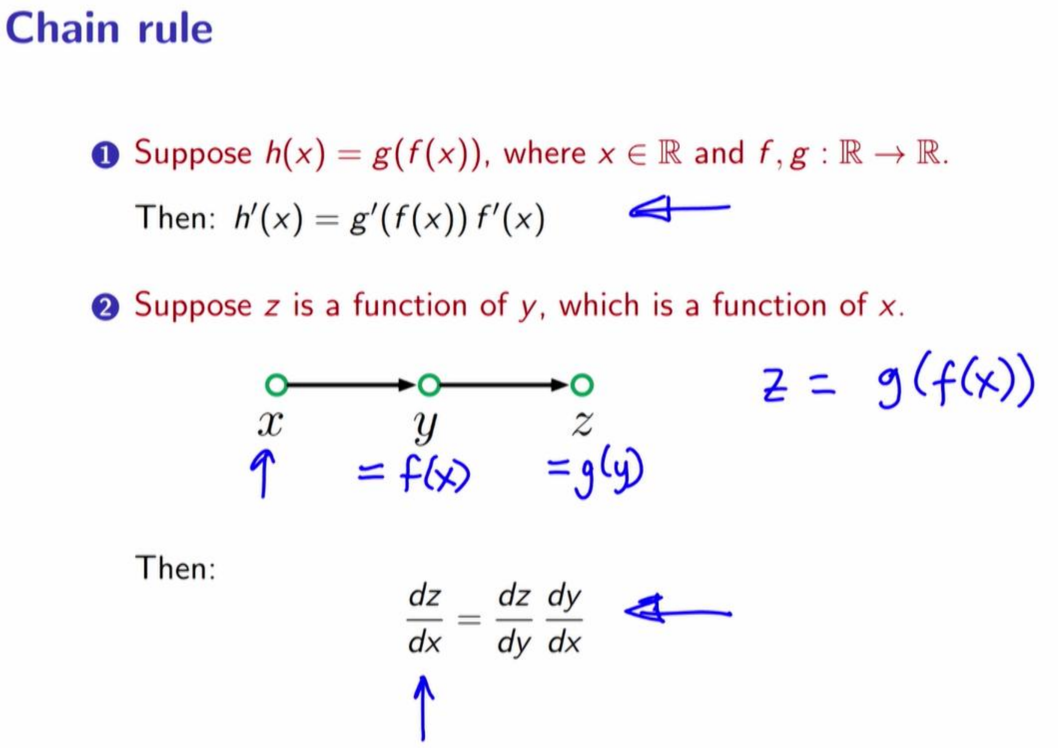
   Then: $h'(x) = g'(f(x))\,f'(x)$

$$\underbrace{(2x+3)}_{u}^{10} \qquad u^{10}$$

$$10u^9\,du$$

$$\longrightarrow 10(2x+3)^9 \cdot 2.$$

Here is chain rule of calculus, written out in full mathematical notation. But actually, this is a rule that we've been using all along in order to compute derivatives. For example, suppose I ask you to compute the derivative of two x plus three, raised to the power of 10. What is that? Well, the way you do it is, you say, "Hm. "Let's take that expression, two x plus three, "and let's call it u." So we are computing the derivative of u to the 10th. What is that? That is 10 u to the ninth, d u. Now, what's d u? Well, u is two x plus three, so d u is two, so it's 10 times two x plus three to the ninth times two. That's the derivative. More generally, let's say you have a function h, which is a composition of two functions. H of x is the result of taking x and first applying function f to it and then applying function g to the result, g of f of x. Then the derivative of h of x is just the derivative of g applied at f of x times the derivative of f at x. That's exactly the rule we used over here, the chain rule.



Chain rule

1. Suppose $h(x) = g(f(x))$, where $x \in \mathbb{R}$ and $f, g : \mathbb{R} \to \mathbb{R}$.

   Then: $h'(x) = g'(f(x))\,f'(x)$ ←

2. Suppose z is a function of y, which is a function of x.

$$\begin{tikzcd} x \arrow[r] & y \arrow[r] & z \end{tikzcd} \qquad z = g(f(x))$$

$$\uparrow \qquad y = f(x) \qquad z = g(y)$$

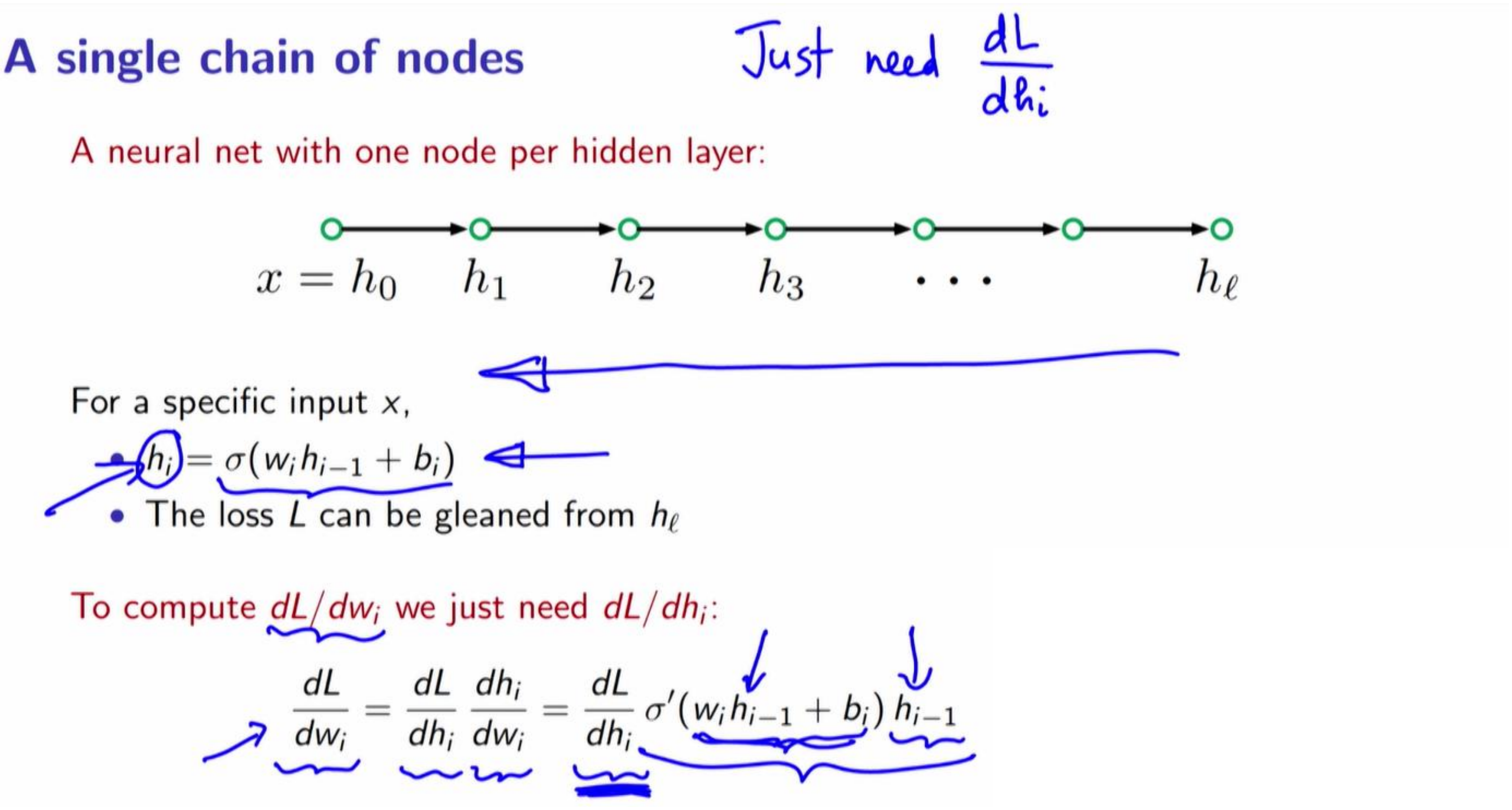
Then:

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx} \leftarrow$$

$$\uparrow$$

Let's look at another way of writing the chain rule. Suppose we have three variables, x, y and z, where y is a function of x and z is a function of y. Then the derivative of z with respect to x is just the derivative of z with respect to y times the derivative of y with respect to x. To see how this is equivalent to the first formulation, how do we do that? Well, we're saying that y is a function of x. So let's write y as f of x. And we're saying z is a function of y, so let's write z as g of y. So z is also g of f of x. Then you'll see that this formulation over here is exactly what we got up here. So this is the chain rule. How do we apply this in neural nets?

总结:



A single chain of nodes

Just need $\frac{dL}{dh_i}$

A neural net with one node per hidden layer:

$$x = h_0 \quad h_1 \quad h_2 \quad h_3 \quad \cdots \quad h_\ell$$

For a specific input x,

- $h_i = \sigma(w_i h_{i-1} + b_i)$
- The loss L can be gleaned from h_ℓ

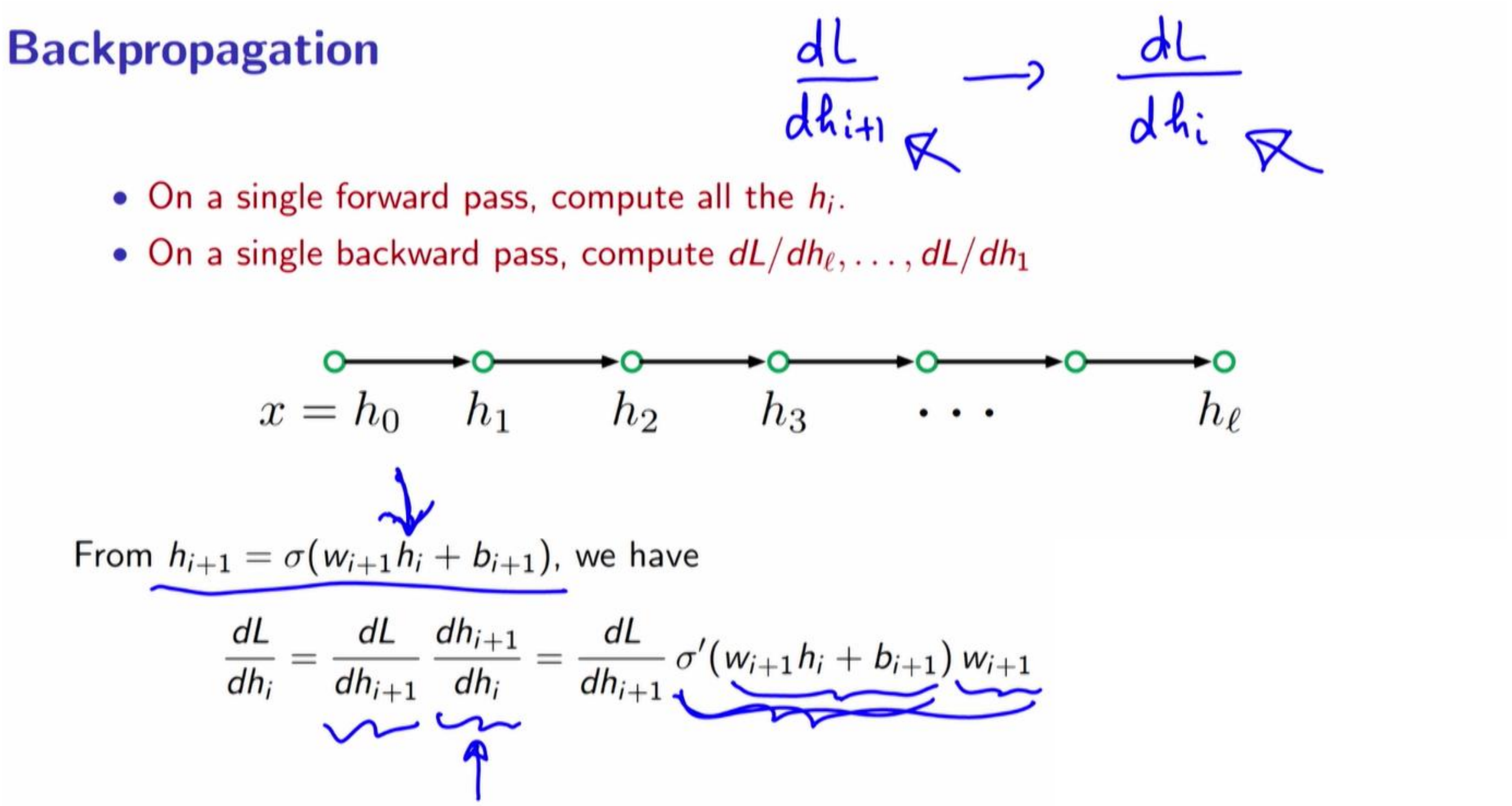
To compute dL/dw_i we just need dL/dh_i:

$$\frac{dL}{dw_i} = \frac{dL}{dh_i}\frac{dh_i}{dw_i} = \frac{dL}{dh_i}\underbrace{\sigma'(w_i h_{i-1} + b_i)\, h_{i-1}}$$

Well, for simplicity, these nets can be huge, so to prevent things from getting messy, let's just look at a case where each layer of the net contains just one node. So the input layer, x, is just one number, x. Layer one just contains one node, h one, layer two just contains one node, h two, layer three contains one node, h three, and at the very end, we have the last layer, h sub l. So the neural net now just looks like a single chain. In fact, to make the notation completely consistent, let's just call the input h zero. That leads to **complete uniformity**.

Now, when we get an input, when we get x, or equivalently, h zero, we compute h one, h two, h three, all the way to h l, in a single feed-forward pass. Each value a, each value like h sub three, is a linear function of the previous node. Here's the formula for the linear function. So each h sub i is some linear function of h sub i minus one, after which we apply the nonlinear activation, for example the rectified linear activation function. So we start at x and we compute h one, h two, h three, and so on. Once we get to the end, once we get to h sub l, then from that we get the probability, the probability of y, and we can assess the loss at that point. What we need is the derivative of the loss with respect to each of these parameters, in particular with respect to each w sub i. How do we obtain that? There are two steps to this.
The first thing is to note that the loss is a function of a whole bunch of things over here, the w's, the b's, the h's. In order to compute the derivative of the loss with w sub i, it turns out that it's enough to just have the derivative of the loss with respect to each of the hidden units h sub i. This is because of the chain rule. So let's see how this works. We wanna compute the derivative of the loss with respect to w i, because that's gonna tell us how to adjust w sub i. To do that, let's look at this equation and apply the chain rule. The derivative of the loss with respect to w i is ***the derivative of the loss with respect to h i*** times ***the derivative of h i with respect to w sub i.*** To compute the derivative of h i with respect to w sub i, well, that's fairly simple. It's just ***the derivative of sigma at that specific point, w i h i, plus b i***, times ***the derivative of this inside stuff with respect to w i, which is h i minus one***. We already know all these h values. We computed them in our feed-forward pass. So we know everything over here and all we need is this **derivative of the loss with respect to h i.** And if we have that, then we can compute the derivative with respect to w i. The first high level message is that we just need the derivative of the loss with respect to each of these hidden nodes, h one, h two and so on. How do we obtain that? It turns out that we can obtain them quite easily, in a single backward pass through the chain. Let's see how that happens.

总结：



Backpropagation

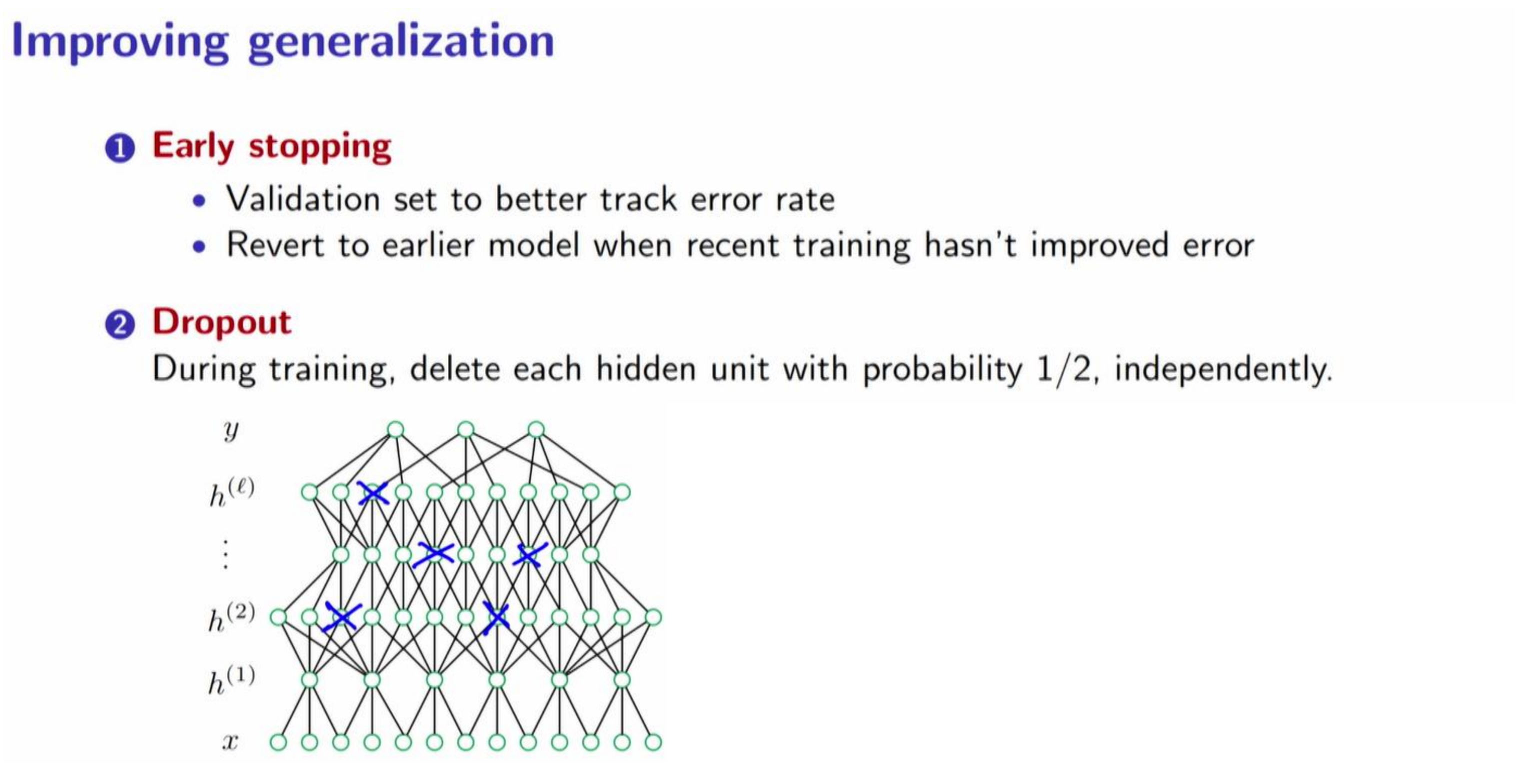
$$\frac{dL}{dh_{i+1}} \longrightarrow \frac{dL}{dh_i}$$

- On a single forward pass, compute all the h_i.
- On a single backward pass, compute $dL/dh_\ell, \ldots, dL/dh_1$

From $h_{i+1} = \sigma(w_{i+1} h_i + b_{i+1})$, we have

$$\frac{dL}{dh_i} = \frac{dL}{dh_{i+1}} \frac{dh_{i+1}}{dh_i} = \frac{dL}{dh_{i+1}} \sigma'(w_{i+1} h_i + b_{i+1})\, w_{i+1}$$

This is the idea of backpropagation. To get the derivative of the loss at the very last unit, that's easy since the loss is a direct function of that. But let's say that we've gone a certain ways down and now we want the derivative of the loss with respect to the next hidden unit. Let's say we already have the derivative of the loss with respect to unit h sub i plus one, and now we want the derivative of the loss with respect to unit h sub i. How do we do that? Well, we write down the formula for h sub i plus one, that's a linear function of h sub i, and then we apply the chain rule. So the derivative with respect to h sub i is the derivative with respect to h sub i plus one times the derivative of h sub i plus one with respect to h sub i. This second term is something we compute directly from this equation. It is the derivative of sigma at that specific point, w i plus one h i plus b i plus one, times the derivative of this inside portion with respect to h i, which is w i plus one. Again, all these terms on the right over here are things we already have. The only thing we really need to compute is the hi's（上式中σ的导数中的参数 hi）, and we obtain them during the feed-forward pass. So given the derivative with respect to h i plus one, we easily get the derivative with respect to h i. And as we saw, this gives us the derivatives with respect to all of the parameters of the model. This is called backpropagation. It's a very ingenious use of dynamic programming to efficiently compute all the derivatives we need. Now, wait, this was only for neural nets that look like a chain, for nets that have a single node per hidden layer. Well, it turns out that this case contains all the complexity of the full situation. The only thing that really changes in moving to wide hidden layers is that things get more messy. The basic idea remains exactly the same.



Improving generalization

1. **Early stopping**
   - Validation set to better track error rate
   - Revert to earlier model when recent training hasn't improved error
2. **Dropout**

   During training, delete each hidden unit with probability 1/2, independently.

总结：

We are dealing with models that have a vast number of parameters and where the function, the loss function that we're optimizing, is highly non-convex. So given these circumstances, we have to be a little bit careful in order to get a model that generalizes well. Now, we saw this situation to a much less extreme degree when we were talking about decision trees and random forests. And indeed, some of the ideas from those settings have also made it over to neural nets. And there are two particular ideas that we'll talk about now, ideas for getting a model that generalizes well. The first is called early stopping. So we have this training process in which we have our training dataset and we are running iterations of mini-batch stochastic gradient descent. As we're running these iterations, we can drive the training error down. We have so many parameters that this is something we can reliably expect to do. This doesn't mean, however, that the true error or test error, the thing we really care about, is going down as well. In fact, the test error could be behaving in some strange way. It might start going up at some point and then come back down again. Who knows? In order to better track it, it helps to have a special separate validation set. So how can we use this?

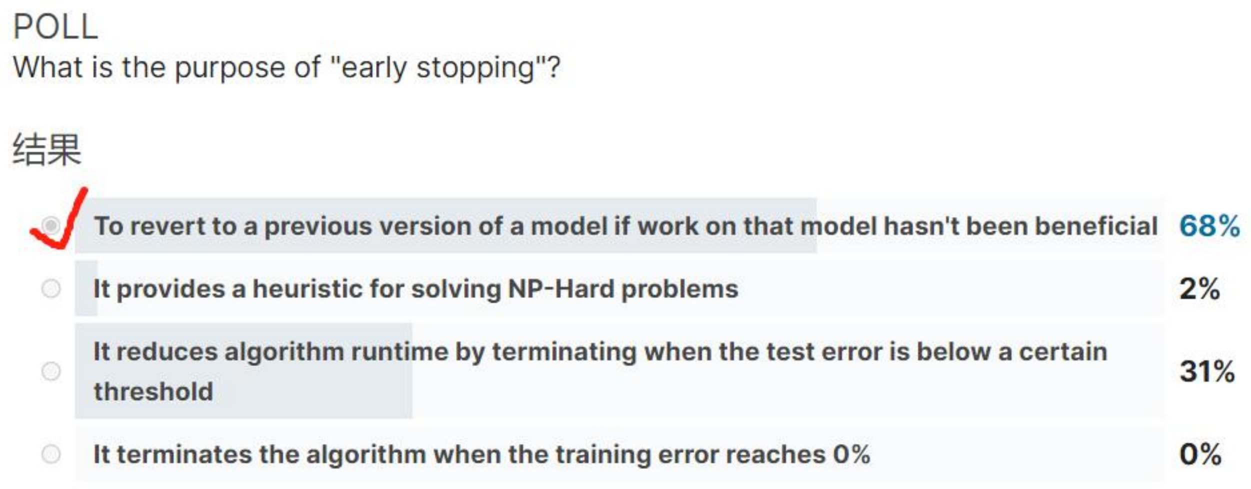
- Here's one thing we can do. We run our iterations of training. And after, if we say 100 iterations, we move over to the validation set and see how the current model is doing. So we get the validation error, which is a good proxy to the actual error, and we see, "Hm, have we improved "from 100 iterations ago? "Do we have a better model now?" If we've improved, great. If we haven't, what we can do is to simply revert to that previous model, the model from 100 iterations ago, and restart the optimization at that point. This is a very handy trick.

- Another trick that's proved to be quite useful is something called dropout. This is somewhat reminiscent of random forests. All we do here is that during training, after presenting an input x, we randomly kill half of the hidden units. Or to be more precise, for each hidden unit, we delete it independently with some fixed probability, like .5. So we end up deleting a whole bunch of these hidden units, just removing them effectively from the graph. Now we still run the feed-forward computation and get a label for y, get a label for x. And we want this label to be correct. In this way, we are forcing the net to arrive at the correct label for x through a variety of different parts. It induces a kind of robustness.

So we've talked a lot about feed-forward nets, about what they are and how to train them. A lot of this is fairly straightforward. The use of the loss function is something very intuitive. The idea of using stochastic gradient descent is very natural. When it gets to the details, computing all the derivatives, using backpropagation, doing things like early stopping and dropout, these are things that in principle seem doable but they really seem like in practice, they might get quite messy and quite hairy. Luckily, you probably won't have to implement these all on your own. There are now packages out there, like TensorFlow, that take care of a lot of the messy details. All you have to do is to specify the architecture as well as the functional form of each of the individual nodes. Once you do that, the package takes care of the rest. It figures out where all the derivatives are, it takes care of the training, and in this way, it makes neural nets much more accessible.

Indeed, if you want to go further with neural nets, I'd say the best thing to do at this point is to just start playing with data using one of these packages. You have the background knowledge you need. So it's time to go out there and start experimenting.

See you next time.



POLL

What is the purpose of "early stopping"?

结果

[✓ ● To revert to a previous version of a model if work on that model hasn't been beneficial] 68%

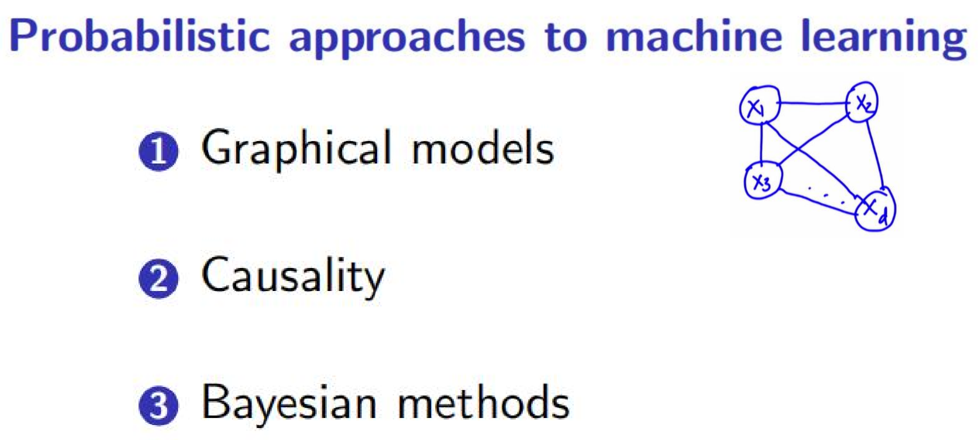
○ It provides a heuristic for solving NP-Hard problems 2%

○ It reduces algorithm runtime by terminating when the test error is below a certain threshold 31%

○ It terminates the algorithm when the training error reaches 0% 0%

总结：

10.5 What We Skipped



Probabilistic approaches to machine learning

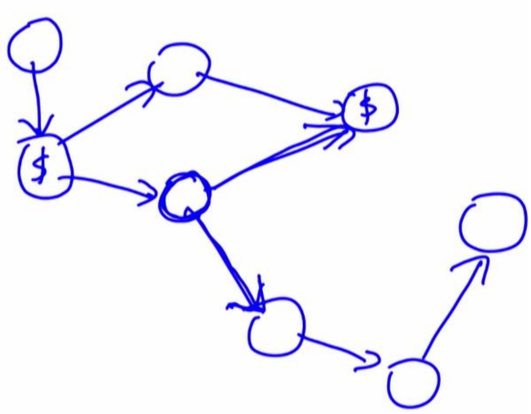
1. Graphical models
2. Causality
3. Bayesian methods

This class is sadly growing to a close. We have certainly covered a lot of ground. Lots of different approaches to making predictions and to representation learning, and quite a bit of the underlying mathematical foundations. But there's also quite a lot that we ended up skipping for want of time. And so what I'd like to do today, to finish up, is to mention just a few of these areas that we overlooked. The first of these is probabilistic approaches to machine learning. Now we saw a little bit of this when we were talking about the generative approach to classification. And we saw how easy it is to build a classifier by just fitting a Gaussian to each class. The nice thing about Gaussians is that they are able to capture correlations between features. And it turned out that these models were frequently pretty effective. This is just the tip of the iceberg.

- However. At the end of the day, Gaussians are a fairly restricted form of probability distribution. For instance, they all have these ellipsoidal shapes. **The framework of graphical models** allows us to capture any probability distribution over data in any dimension. A graphical model is specified by a graph in which there is a node for each feature, a node for X1, for X2, X3, all the way to Xd. And we put an edge between two nodes if we want to capture the dependence between those two features. Now the size of the model and number of parameters we need depends on the number of edges we choose. The more edges, the more parameters. If we put in no edges at all then we end up with a distribution in which the different features are all independent of each other and we need very few parameters to do this. If we put in all possible edges, if we get a complete graph, then we need a lot of parameters, but we are able to model to capture any distribution whatsoever. So in this way we can trade off the accuracy of the model with the number of parameters we need. This is a beautiful framework, and in fact this area is full of beautiful algorithms, ingenious schemes, and rich underlying theory. One of the things that these models help us do is to capture various types of data that we really haven't had an opportunity to discuss. For instance, sequence data, DNA sequences, sentences, speech, or data that is spatial. For example, the distribution of pollution levels across a certain region. So it's a very powerful methodology.
- Another aspect of probabilistic approaches is that they can be used as the launching pad for understanding **causality**. So causality is very important in the way we construct the world, in our explanations, for example. Our reasoning is all about causes and effects. So is causal structure something that can be inferred automatically from data? And if so, what kinds of data do we need in order to do this? These are fascinating questions and these are very important if we want machines to be able to reason about the world.
- The third avenue of probabilistic modeling is **Bayesian methods**. Now this is a rich area, a whole field of statistics unto itself. In this class we have focused on a kind of learning algorithm that takes a dataset and returns a single model. For example, a single linear regression function. Now this function is not gonna be perfect since we only have a finite amount of data, and so we are in a bit of a strange situation where we have a model that's not quite right, and yet we don't have a clear sense of which parts of the model are super reliable and which parts are a little bit more dicey. In Bayesian methods the learning algorithm doesn't just return a single model, but rather a probability distribution over all possible models, okay? So it says this linear regressor has probability 0.2, and this linear regressor has probability 0.1, and this linear regressor has probability so and so. We get back this entire distribution. Now one thing we can with the distribution is to pick out the single most likely model and use that, but what this gives us in addition is it let's us identify what parts of that model are somewhat more set in stone than the other parts, and it lets us quantify the uncertainty in predictions that we make. So it's a very powerful methodology.

总结：

Reinforcement learning



A secondary of machine learning that we completely overlook is reinforcement learning. Now in this course we have really been focusing upon training a machine to answer a very specific type of question over and over again. What is the name of the animal in this picture? What is the name of the animal in this picture? What is the name of the animal in this picture? And so on, you get the idea. But imagine what happens when we move to a slightly broader scenario. For example, when we are training a robot to do something like pick up a cup of coffee, say. So at any given time the robot's in a particular configuration, that's the state of the world, the environment, and it chooses an action that's suitable for that environment. Maybe it moves a little bit. This action changes the state of the world and the action it chooses subsequently has to adapt to this changed environment. So we are in a situation where a machine is moving around, it's choosing actions that are suitable for the current environment, but at the same time these actions are changing the environment. A beautiful framework for modeling such situations is called **reinforcement learning** and it was inspired by intuitions from behaviorist psychology.

So what happens in reinforcement learning is essentially you think of all the possible states, all the possible environments, as being some enormous graph. Okay, so here is one possible state. Here's one possible state of the world. Here's another possible state, and we have lots and lots of these. And they're connected by edges. Now at any given time we're in one of these states, the current state of things. So let's say we are in this state. We then choose an action and the effect of that action is to move us to one of the neighboring state, in this case either state or this state, and in this way we move around. Now the only feedback we ever get is that there is the occasional reward scattered around. Some of these states have a little reward associated with them. And so the goal of the learning machine is to behave in a way that maximizes the overall longterm reward. So it's a beautiful framework that captures a lot of different situations.

The human side of machine learning

1. Trust
2. Transparency
3. Explanations

The third aspect in machine learning that we really should have talked about is the human side of all this.
So machine learning is playing a growing role in our lives. For instance, machine learning systems are constantly monitoring us through our emails, and social media, and purchases, and we also rely on these systems to make a whole lot of everyday decisions. What does it take for a machine learning system to earn our trust? What kind of systems do we think of as trustworthy? Is it enough that they just have high predictive accuracy, or does it help for them to have other important features as well? For example, the ability to say, "I don't know." Now some of these systems are being used to make decisions that are really important. For example, which loan applications get accepted or rejected, and for systems like these it's really important for the reasoning of the machine learning system to be transparent, for it to be able to explain the decision in a way that makes sense to all the parties involved and that can also possibly stand up to scrutiny in court. Now some of the methods we've studied are fairly transparent. Decision trees for example, they are quite understandable. But some of the other methods like neural nets are really pretty inscrutable. What would it take to make these more transparent? Is there some small fix that we can use? Or do we have to start from scratch and just redesign our models and methodologies from the ground up? These are really interesting questions.

So I'm afraid that we're gonna have to end here. Machine learning is a very rich field with powerful technology and beautiful underlying mathematics. There's also no shortage of open problems and challenges for the future. I hope that you found this course interesting and that it has peaked your curiosity to perhaps delve into this lovely area a tiny bit more.

总结：