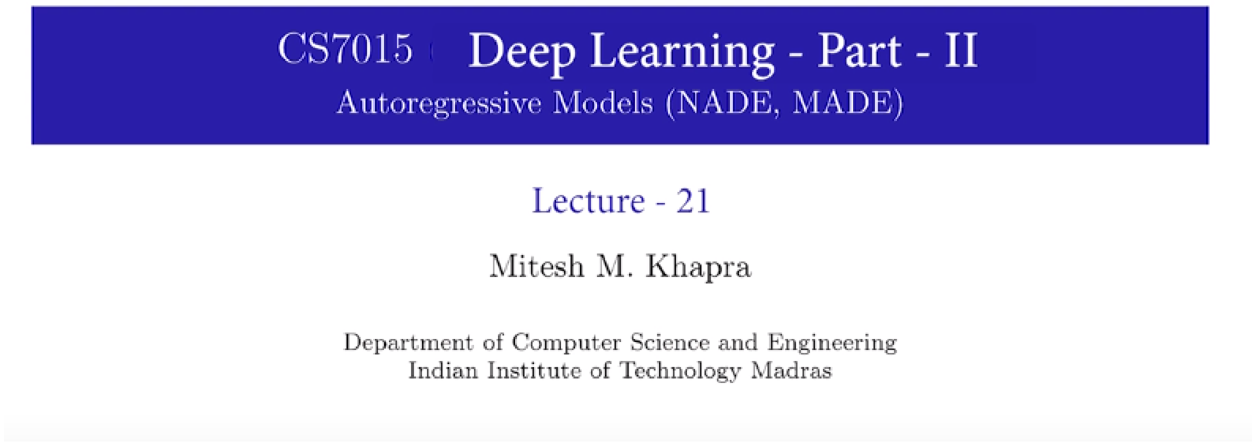
Lecture - 21
Neural Autoregressive Density Estimator (NADE)

Refer Slide Time: (00:10)



CS7015 Deep Learning - Part - II

Autoregressive Models (NADE, MADE)

Lecture - 21

Mitesh M. Khapra

Department of Computer Science and Engineering
Indian Institute of Technology Madras

Today, we will talk about auto regressive models and this is continuing on our journey

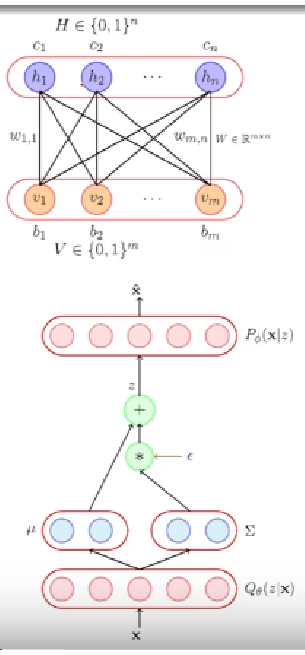
Refer Slide Time: (00:19)

towards deep generative models, we've already seen a few RBMs and not few too actually RBMs and

Module 21.1 Neural Autoregressive Density Estimator (NADE)

various new autoencoders. Today, we look at one more family of deep generative models known as auto regressive models and in particular, we look at two architectures NADE and NADE, I'll tell you what they mean, they're not just made up terms and so let's start with the first one which is neural auto regressive density estimator. Okay? So, let's look at each of these terms. I hope the first term is obvious to everyone by now, I guess density estimator, what does that mean, probably density good, not physics, Okay? So, this just means that we want a neural model for estimating probability densities. Okay? And what does autoregressive mean, of course you don't know Right?? If you knew all these and why am I here. So, I will we look at that and see what auto regressive means, Right?

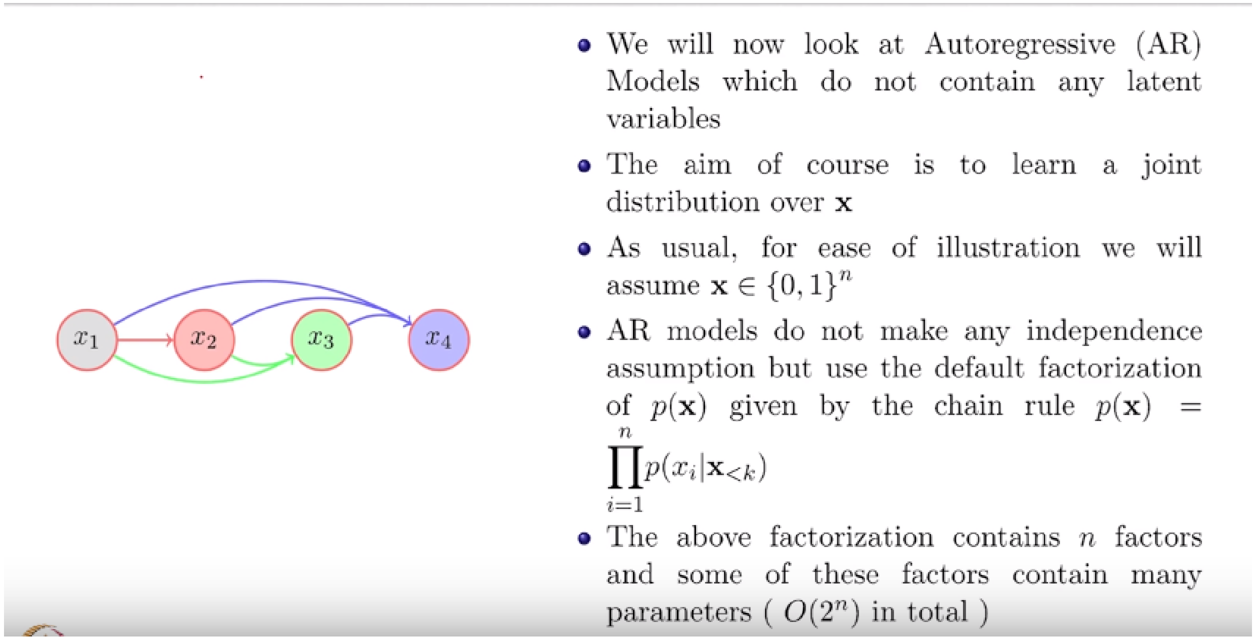
Refer Slide Time: (01:17)



- So far we have seen a few latent variable generation models such as RBMs and VAEs
- Latent variable models make certain independence assumptions which reduces the number of factors and in turn the number of parameters in the model
- For example, in RBMs we assumed that the visible variables were independent given the hidden variables which allowed us to do Block Gibbs Sampling
- Similarly in VAEs we assumed $P(\mathbf{x}|z) = \mathcal{N}(0, I)$ which effectively means that given the latent variables, the $\mathbf{x}$'s are independent of each other (Since $\Sigma = \mathrm{I}$)

So, so far, we have seen a few latent variable models fix RBMs and variational autoencoders and these models make certain independence assumptions by relying on the latent variables. So, for example both in the case of RBMs and the VAEs we made some assumptions, in the case of RBMs, we said that given the latent variables, the visible variables are independent of each other. So, once you bring in this latent variables and the idea is that the number of latent variables is actually much smaller than your total visible variables, and you get rid of the dependencies between the visible variables by just assuming that given the latent variables, all of these are independent. So, that largely simplify is your factorization and reduces the number of parameters in your model and in fact even though it was not so obvious in the case of variation autoencoders, we made a similar assumption there also, we very smartly said that P of X given Z, actually comes from a normal distribution whose covariance matrix is identity, that's the same as saying that given the Z, the X's are independent of each other, Right? hence the Sigma is equal to I. So, we brought in these latent variable models and they had in our eventual goal which was towards having a tractable model, of course even then there were issues, it's not that just bringing in the latent variable model solved everything, despite having latent variables, we still had to do this expensive MCMC sampling, in the case of RBMs and even in the case of VAEs, we had to get rid of this integral or summation or expectation by resolving to this variational inference rate which tries to maximize the lower, lower bound instead of actually working with the original objective function, Right? So, just having latent variables is not enough on top of that also, we had to do certain things,

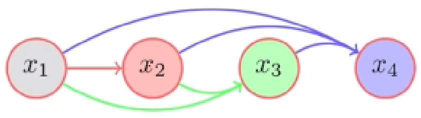
Refer Slide Time: (02:59)



- We will now look at Autoregressive (AR) Models which do not contain any latent variables
- The aim of course is to learn a joint distribution over $\mathbf{x}$
- As usual, for ease of illustration we will assume $\mathbf{x} \in \{0,1\}^n$
- AR models do not make any independence assumption but use the default factorization of $p(\mathbf{x})$ given by the chain rule $p(\mathbf{x}) = \prod_{i=1}^{n} p(x_i|\mathbf{x}_{<k})$
- The above factorization contains n factors and some of these factors contain many parameters ($O(2^n)$ in total)

now we look at Autoregressive models which do not contain any latent variables. Okay? The name, the name of course remains the same that we are still given a bunch of variables X, Right? So, this is x_1 to x_n, and in our example this has been explained to x_{1024}, because you have 32 cross 32 images and you want to learn a Joint Distribution of that and as usual for ease of illustration, we'll just assume that this X belongs all these X's are binary, Right? So, your X comes from 0 comma 1 raise to n, instead of R raise to n. Okay? That's easier for us to illustrate it, now AR models actually do something very bold they assume that there are no independence assumptions, that's a really convoluted way of seeing it. So, they assume no independencies. So, they what they assume instead is the natural factorization and that's not an assumption, they just break down this probability distribution using the chain rule, and you saw the original chain rule had this X_1 priority of x_1 into x_2 given x_1 into x_3 given x_1, x_2 and so on. So, in general one of the factors in this joint, in this factorization is probability of X_i given X_1 to X minus 1, Right? and how would you represent that as a graphical model, what are the nodes in your graphical model, x_1 to x_i, what are the edges, for X_i, what are the edges, it's connected to everything from x_1 to $x_{i \text{ minus } 1}$ and it connects to everything that comes after it, Right? a parent of everything that comes after it, and a child of everything that comes before it, Right? That’s what this figure is trying to illustrate. Okay? So, every node depends on all its previous nodes and all these subsequent nodes are dependent on this node, that's very simple a that's very straight forward,, just convert this factorization into a graphical model. Okay? but this is like really frustrating, Right? we had this entire saga of three weeks where we said that we cannot work with this joint factorization, because it has an exponential number of parameters and after doing all that epic saga and back to the basic factorization and seeing that auto regressive model work with that Right? So, this is really expensive, in fact the total number of parameters that you have in this factorization is order 2 raise to n, there were some of the factors have a large number of parameters, Right? So, everyone gets that. So, this have the first factor has one parameter, the second has two, four and soon, Right? So, it's a exponential number of factors that you get. So, what does I mean, I started with saying that this is bad, this is bad this is bad, we did a lot of stuff along the way to explicitly avoid this kind of factorization and now we are back to this.

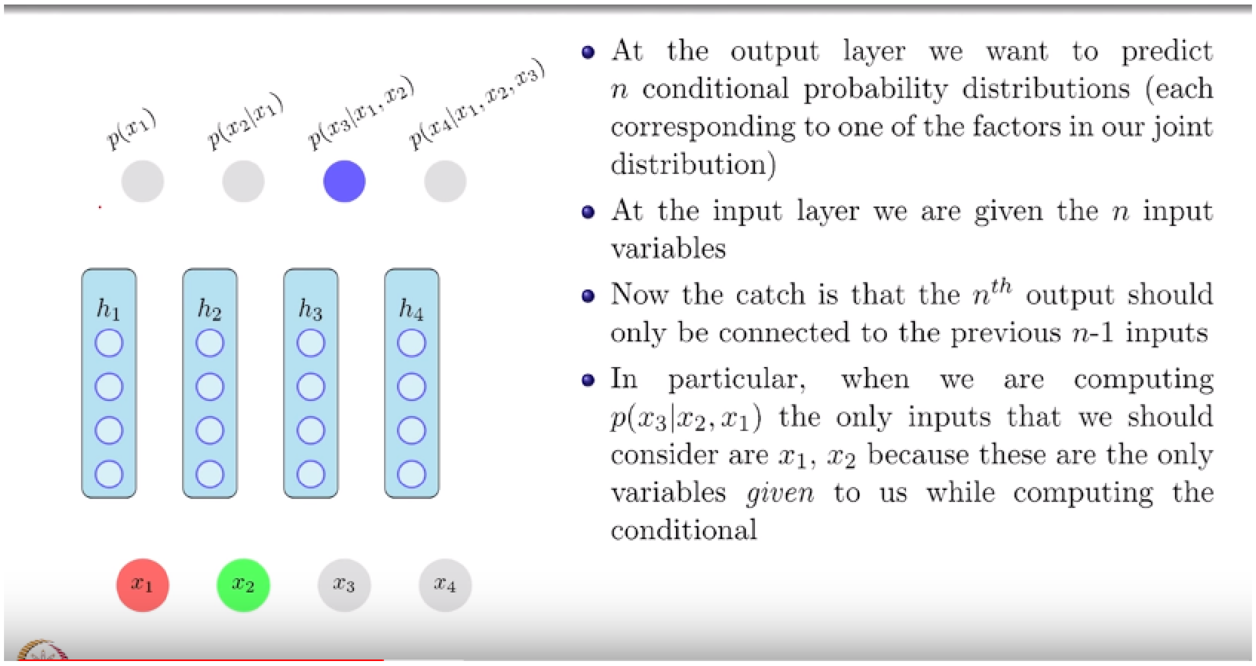
Refer Slide Time: (05:39)



- Obviously, it is infeasible to learn such an exponential number of parameters
- AR models work around this by using a neural network to parameterize these factors and then learn the parameters of this neural network
- What does this mean? Let us see!

So, how does this make sense? So, this is infeasible, but the way auto regressive models work around this, is that they use a neural network to parameterize these factors and then learn the parameters of this neutral network. Again, nothing new, what does this sentence actually say? It is giving you the dash for learning a joint distribution, the standard recipe, Right? this was the recipe that we discussed that if you want to learn a joint distribution, you can say that there are certain parameters associated with the distribution, the parameters in turn can be functions of some other parameters and then, you learn the those parameters and that's exactly what this sentence is saying in short, Right? So, again that does not really explain, how we are going to get rid of this exponential number of parameters. So, let us see what this exactly means and how it allows us to bypass this problem. Okay?

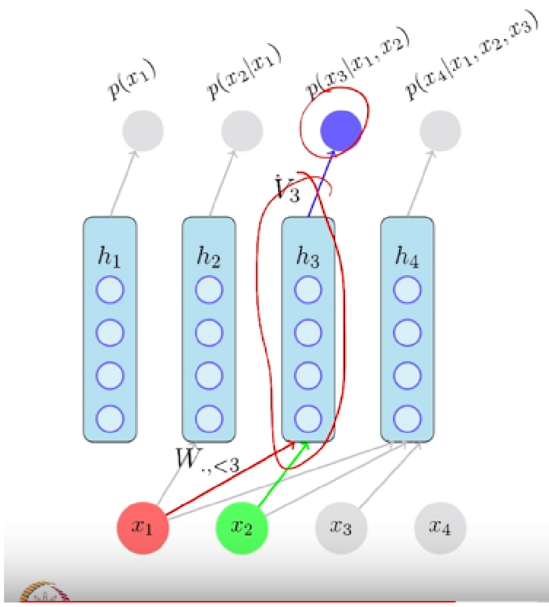
Refer Slide Time: (06:33)



- At the output layer we want to predict n conditional probability distributions (each corresponding to one of the factors in our joint distribution)
- At the input layer we are given the n input variables
- Now the catch is that the n^{th} output should only be connected to the previous n-1 inputs
- In particular, when we are computing $p(x_3|x_2, x_1)$ the only inputs that we should consider are x_1, x_2 because these are the only variables *given* to us while computing the conditional

So, in terms of neural network, Right? So, for the time being just ignore this part, actually we can remove this from this slide. So, what do I want, I want a neural network where I have n outputs, Okay? Each of these outputs predicts one of the conditional probability distributions that I am interested, Okay? There are n conditional probability distributions in my factorization, I will have a neural network which has n outputs each of these outputs will predict p of x_i, given x_1 to $x_{i\ \mathrm{minus}1}$, Okay? Without telling you anything else, can you tell me what is the function that you will use for each of these output nodes, Right? So, remember we used to do this, Right? what's the function for input, it was sigmoid tan it and so on, then what's the output function then, what's the loss function Right? So, I'm just bringing that back without me telling you anything else, I'm just telling you that the job of the neural network is to predict these conditional probability distributions and further, we are assuming that all these variables are binary, Okay? So, now can you tell me what is the output function that you use for the output layer, what do you expect the first node to predict a value between 0 and 1, the second node. So, soft max, we need a probability distribution, Right? So, we should have a soft match, why not? These nodes are all independent of each other, these are all separate factors there some need not be one. So, we don't need a soft max here, what do we need, we need n sigmoid functions, each of these sigmoid functions will predict a value between 0 to 1 which will tell us what's the probability of x_i taking on the value 1, given x_1 to $x_{i \text{ minus } 1}$. So, is the output layer clear to everyone, is that fine! please raise your hands if it is clear, Okay. So, you should understand why we don't need a soft max function and why we will have n independent sigmoid functions. Okay? Fine, now at the input again you are given these x_1 to x_n. Right? So, this is the pixels that are given to you and what you want to predict is probability of x_i given $x_{i \text{ minus } 1}$. Okay? But the catch is that the n^{th} output should see only inputs from 1 to n minus 1, why is it so? I am saying that the n^{th} output or the i^{th} output should only see inputs from 1 to i minus 1, why is it so? Because that's the given part, Right? the i^{th} output is predicting p of x_i, given x_1 to $x_{i \text{ minus } 1}$. So, given means what was the input given to me. So, the input was only x_1 to $x_{i \text{ minus } 1}$. So, when I am predicting this probability, I should only be looking at the inputs x_1 to $x_{i \text{ minus } 1}$, does that make sense, in particular see if I give you x_i, also then what's the point of predicting what is p of x_i, that I have already seen that what's the, what am I trying to predict, does that make sense? Right? So, the given part tells you what is the input given for this computation? So, you should only be looking at x_1 to $x_{i \text{ minus } 1}$. Okay? Now, if I were to use a fully connected neural network that means I start with the input layer, I have some hidden layers and then I go to the output layer, will that be. Okay? I'm saying if I had a fully connected layer, I start with the input layer, think of the multi- layer perception that we did, Right? So, we have the input layer, we have a bunch of hidden layers and then the output layer and everything is fully connected, will that be. Okay? For the solution, yes or no? Everyone. Right? Yes. So, by definition it's fully connected. So, every guy in the hidden layer sees all the inputs which in turn means every guy in the next hidden layer has seen all the inputs and you can argue this up to the output layer, every guy in the output layer has seen all the inputs and that's exactly the problem that I want to avoid, I don't want every guy in the output to see all the inputs, I just wanted to see the relevant inputs. Okay?

Refer Slide Time: (10:48)



- The Neural Autoregressive Density Estimator (NADE) proposes a simple solution for this
- First for every output unit, we compute a hidden representation using only the relevant input units
- For example, for the k^{th} output unit, the hidden representation will be computed using:

$$h_k = \sigma(W_{.,<k}\mathbf{x}_{<k} + b)$$

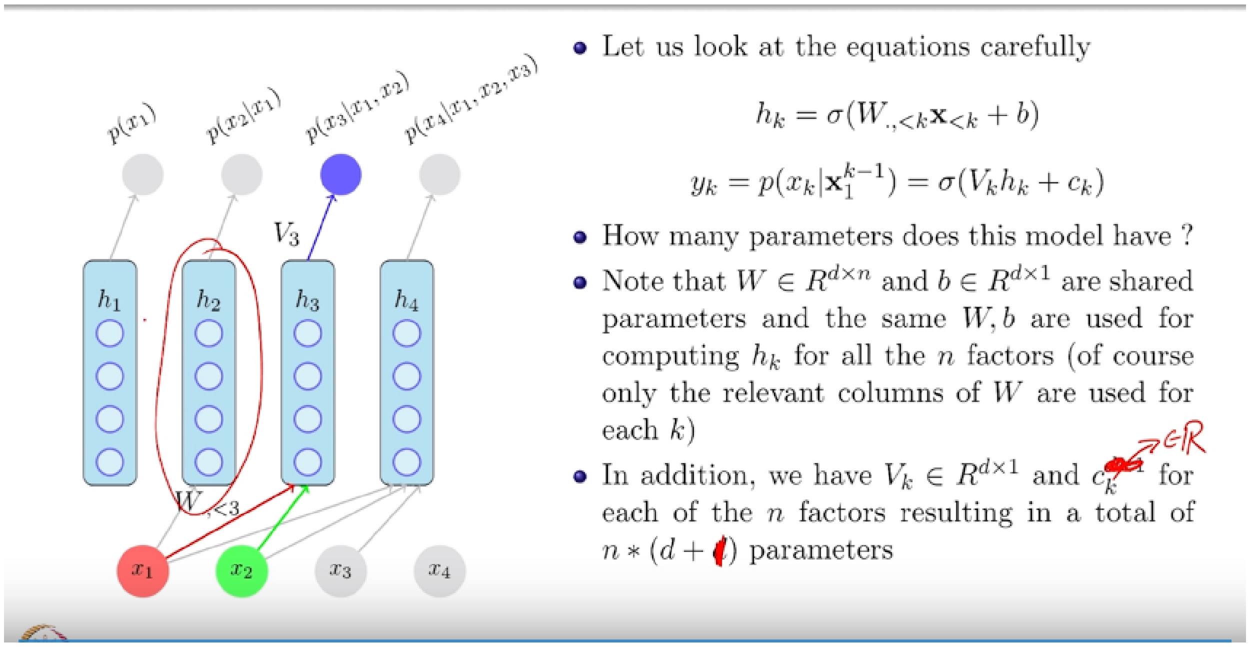
where $h_k \in R^d, W \in R^{d\times n}, W_{.,<k}$ are the first k columns of W

- We now compute the output $p(x_k|\mathbf{x}_1^{k-1})$ as:

$$\boxed{y_k} = \underline{p(x_k|\mathbf{x}_1^{k-1})} = \sigma(V_k' \boxed{h_k} + c_k') \rightarrow x_1 \ldots x_{k-1}$$

So, that's why a fully connected network does not work. So, this neural auto regressive density estimator has a simple solution, for this for a three output unit, we compute a hidden representation using only the relevant inputs. Okay? So, let's look at x_3 this is the output unit that. So, I want to first compute a hidden representation, Okay? I'll have a matrix W. So, think of this W as the fully connected matrix. Okay? But out of those entire W that entire W, I'm going to only look at the first less than k columns. Okay? And from the input I'm also going to look at only the first less than k columns. Does that make sense? Okay? So, that means, what does that mean, my h_k only depends on x_1 to $x_{k \text{ minus } 1}$.So, I have ensured that the hidden representation which I am computing has only seen a selected number of inputs and in particular only those inputs which I was allowed to see, because this is a k^{th} output, I should have seen only inputs up to k minus 1, does this computation make sense to you? Right? it does, I'm just looking at the first k entries of the input, Okay? Fine. And now we can compute the output. So, I'm interested in the k^{th} output which actually gives me the priority of x_k given x_1 to k minus 1 and I'm going to compute it as a function of h_k, which in turn is only a function of x_1 to $x_{k \text{ minus } 1}$. So, everything closure up to this point, I have not seen anything that I was not supposed to see and v_k and c_k are just parameters. So, that doesn't matter, Right? So, I if I do this kind of a computation. So, what I'm doing in effect is, for every output, I am first computing a hidden representation which looks only at the relevant inputs, Right? And once I make sure of that whatever I do with the hidden representation is going to be all fine, because the hidden representation has seen only the relevant inputs, hence the output computed using the certain representation will only see the relevant inputs, is that fine with everyone, how many few clear with this? Please raise your hands, top and high, Okay? Good.

Refer Slide Time: (13:04)


$p(x_1)$
$p(x_2|x_1)$
$p(x_3|x_1,x_2)$
$p(x_4|x_1,x_2,x_3)$
V_3
h_1
h_2
h_3
h_4
$W_{.,<3}$
x_1
x_2
x_3
x_4

- Let us look at the equations carefully

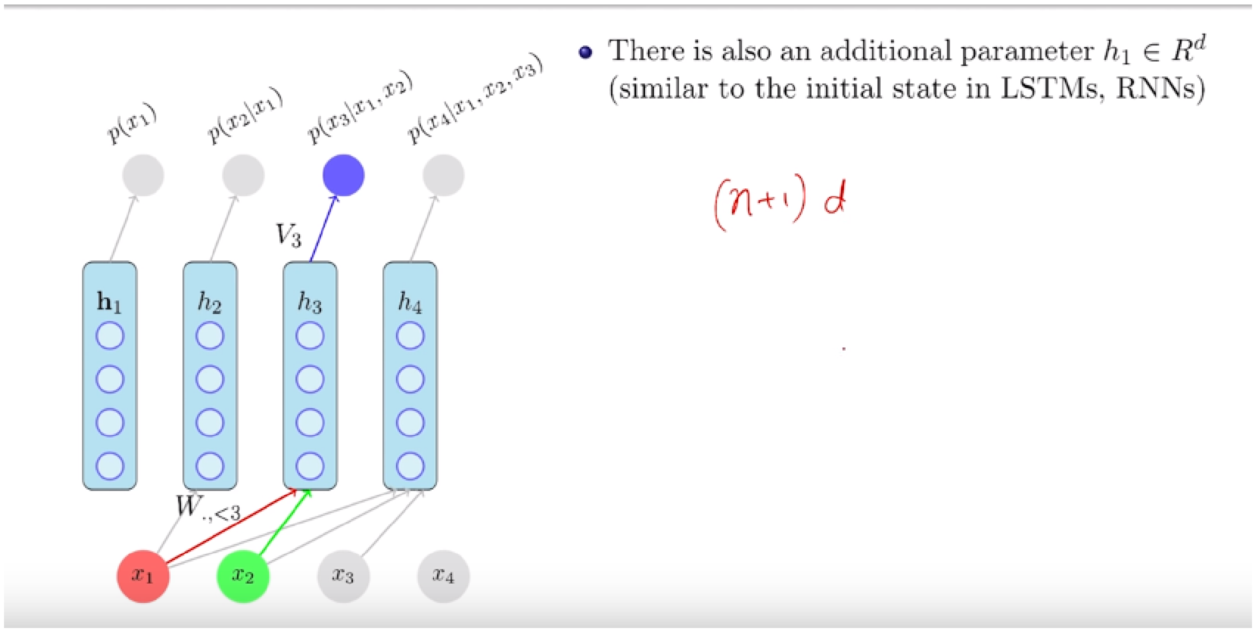
$$h_k = \sigma(W_{.,<k}\mathbf{x}_{<k} + b)$$

$$y_k = p(x_k|\mathbf{x}_1^{k-1}) = \sigma(V_k h_k + c_k)$$

- How many parameters does this model have ?
- Note that $W \in R^{d\times n}$ and $b \in R^{d\times 1}$ are shared parameters and the same W, b are used for computing h_k for all the n factors (of course only the relevant columns of W are used for each k)
- In addition, we have $V_k \in R^{d\times 1}$ and c_k $\in R$ for each of the n factors resulting in a total of $n * (d + 1)$ parameters

So, now let us look at this equation carefully. So, you have h_k is equal to first k columns of the W matrix multiplied by the first k entries of the x vector and then you have this prediction, Okay? How many parameters does this model have, notice something, there is no suffix associated with w_1b which means, they are shared across all the outputs but there is a subscript associated with v_k and c_k, Okay? and I've asked you the question what's the number of parameters, I also need to tell you something about the input and the output. So, we will assume that x belongs to 0, 1 raise to n, that means X is n dimensional Right? it does not matter whether 0, 1 or what? But it's n dimensional and I will assume that h belongs to R^d. So, it's a d-dimensional vector. Okay? Now, you need to tell me, how many parameters does this model have to start counting and why am I asking you this question, what will I have to prove to you, that this does not have exponential number of parameters, I've somehow got rid of exponential number of parameters. Right? So, how many parameters does this model have, what is the size of W, d cross n, what's the size of b, d, everyone please, this a second last lecture of the course, I expect you to do this, how many parameters is w are, everyone should answer, everyone n cos d, how many parameters does be have, d, Okay? So, you have d cross n and d cross 1, Okay? And the same w_1b are used for all the outputs. Okay? how many parameters does the output layer have, what's the size of v_k, what's the size of v_k, everyone please, how many of you get this, that it's 1 cos d, please raise your hands up and high. Okay? how many such v's do you have, n. What's the size of c_k, oh god I gived up, Okay. It's 1 sorry, yeah! So, what's the total number of parameters here, n into and into d plus. Okay? So, that's number of parameters that you have in the output layer, is there any other parameter obvious from the figure, how do you compute h_2, how do you compute h_2? You take the first column of W, the reason I'm asking you this question is because next I'm going to ask you how do you compute h_1, how do you compute h_2, is clear to you, how do you compute h_1, does this start h_1 remind you of anything else that you have seen before, where have you seen a such a similar thing, LSTMs and RNNs you had this edge zero state, what did we do with the edge zero state, we made it our parameter, Right? So, similarly h_1 is going to be a parameter here, what's the size of h_1, d.

Refer Slide Time: (16:49)

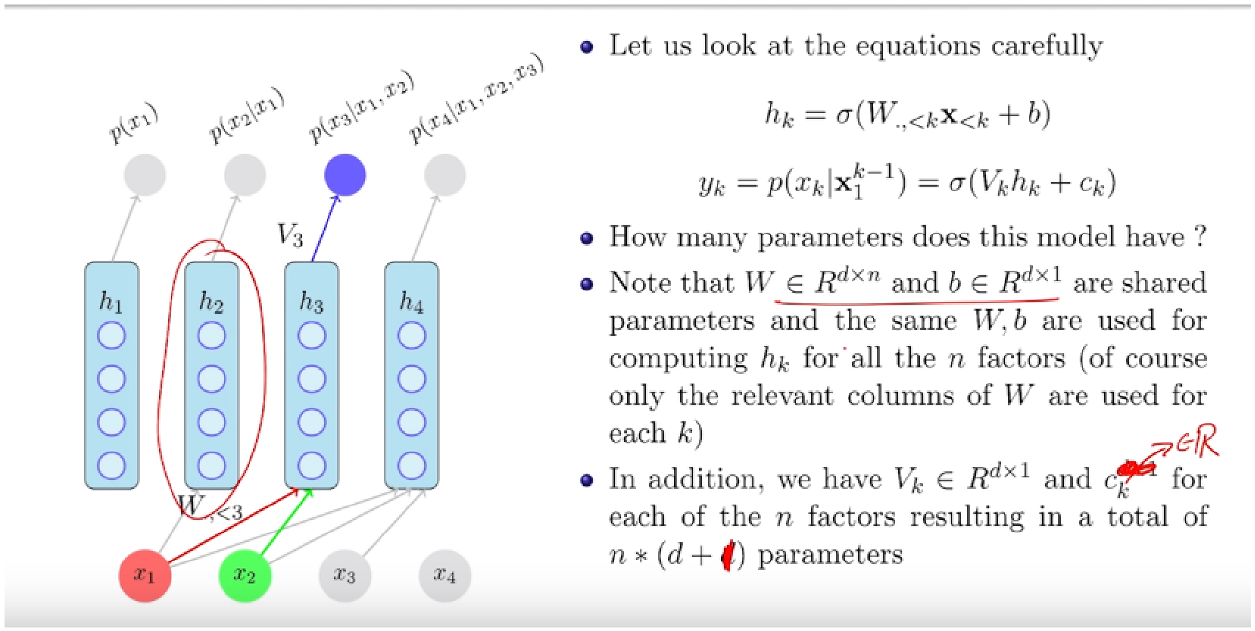


- There is also an additional parameter $h_1 \in R^d$ (similar to the initial state in LSTMs, RNNs)

$(n+1)\ d$

So, what's the total number of parameters that you have, I think I've goofed up here, yeah! So, you can. So, Okay? Let's just do it. Right? So, we have n plus 1 into d here. Okay? So, n plus 1 into d in the input layer,

Refer Slide Time: (17:10)



- Let us look at the equations carefully

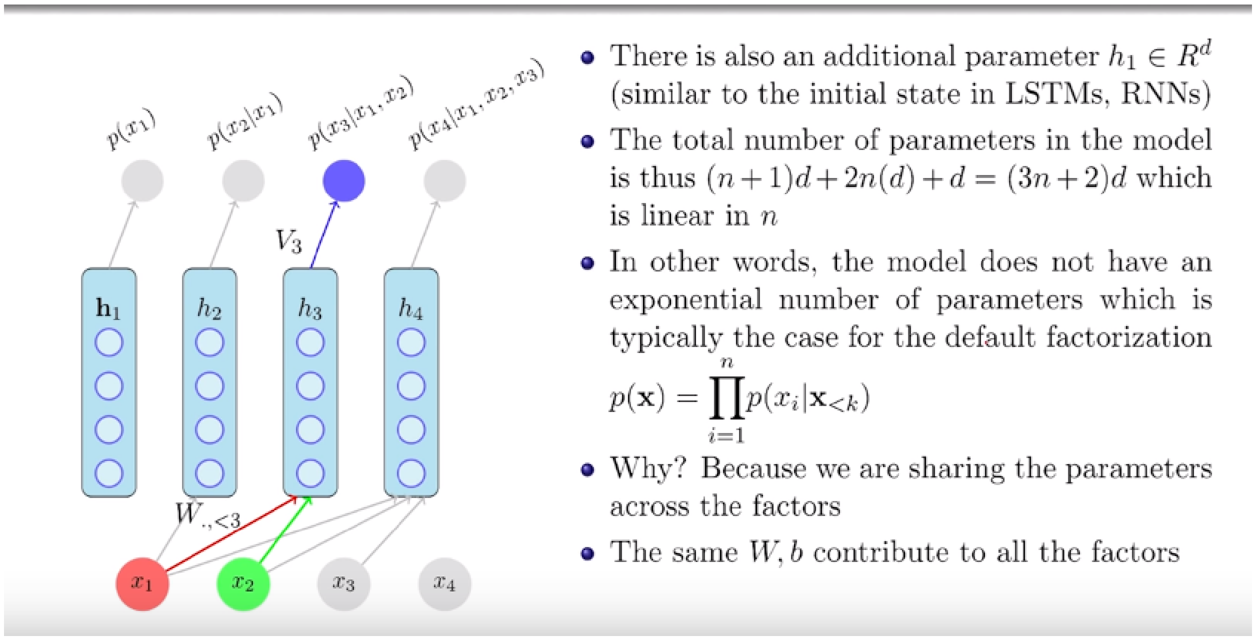
$$h_k = \sigma(W_{.,<k}\mathbf{x}_{<k} + b)$$

$$y_k = p(x_k|\mathbf{x}_1^{k-1}) = \sigma(V_k h_k + c_k)$$

- How many parameters does this model have ?
- Note that $\underline{W \in R^{d\times n} \text{ and } b \in R^{d\times 1}}$ are shared parameters and the same W, b are used for computing h_k for all the n factors (of course only the relevant columns of W are used for each k)
- In addition, we have $V_k \in R^{d\times 1}$ and c_k $\in R$ for each of the n factors resulting in a total of $n * (d + 1)$ parameters

then what was the next thing.

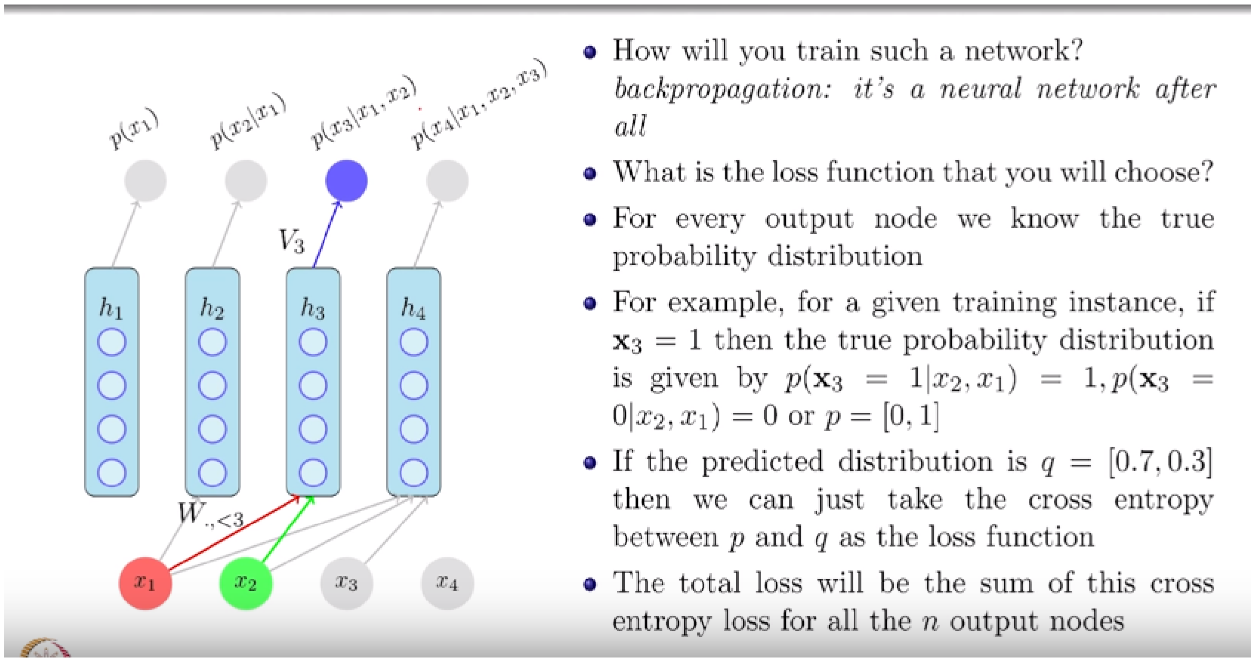
Refer Slide Time: (17:13)



- There is also an additional parameter $h_1 \in R^d$ (similar to the initial state in LSTMs, RNNs)
- The total number of parameters in the model is thus $(n+1)d + 2n(d) + d = (3n+2)d$ which is linear in n
- In other words, the model does not have an exponential number of parameters which is typically the case for the default factorization $p(\mathbf{x}) = \prod_{i=1}^{n} p(x_i|\mathbf{x}_{<k})$
- Why? Because we are sharing the parameters across the factors
- The same W, b contribute to all the factors

Okay? So, let's just call it nd plus d and the next one was nd plus n and then another d, Right? So, this is what you had in the input layer, this is what you had in the output layer and this is your h_1, Okay? So, then in effect, how many parameters do you have, 2 nd plus 2 d plus n Right? So, it's order n. So, do you have an exponential number of parameters, No. So, how did this happen, we started with a factorization which is an explicit factorization and we have been arguing, since the beginning of this section that such a factorization will have an exponential number of parameters. So, why don't you have an exponential number of parameters here, we shared the parameters, Right? That's one reason I don't have exponential number of parameters, Okay? So, now let's just do one more thing. Right? Before we move on. So, I just wanted me make sure that all of you are clear with this computation. So, I had this W less than k into x less than k plus b, Okay? So, let's actually see, what that means? So, what's the size of this output going to be this belongs to what R^d. Okay? Fine. So, now when you are doing, say let's just this is x_1, x_2, x3. Okay? And then you have this W_{11}, W_{12}, W_{13}. Okay? Now, I said that you just take the first k columns, Right? So, this would be if k is equal to 2, then you will take the first column and you'll also take the first row from here, what will this multiplication give you, a scalar, a vector, a tensor? A vector. Right? So, that's exactly what is happening here but the. Right? Way to implement this is actually the following that you take your input and you mask it. Okay? Then whatever vector you get which is essentially going to be x_1, 0, 0. So, then you can directly do this matrix vector multiplication and that's the same as this operation, but that's the more neater way of doing it take your input mask it and then multiply by your weight matrix, how many of you get this. Okay? So, that's what W less than k into x less than k means. So, just get familiar with this concept of masking we'll be using it soon again. Okay? Fine.

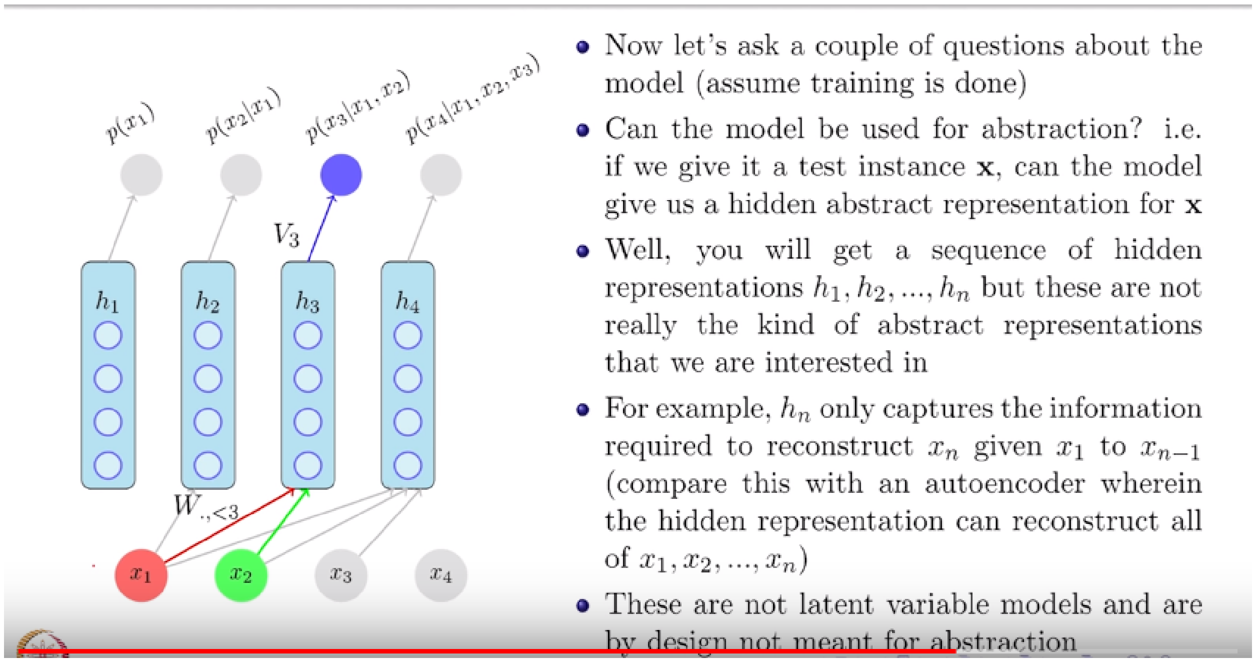
Refer Slide Time: (20:02)



- How will you train such a network? *backpropagation: it's a neural network after all*
- What is the loss function that you will choose?
- For every output node we know the true probability distribution
- For example, for a given training instance, if $\mathbf{x}_3 = 1$ then the true probability distribution is given by $p(\mathbf{x}_3 = 1|x_2, x_1) = 1, p(\mathbf{x}_3 = 0|x_2, x_1) = 0$ or $p = [0, 1]$
- If the predicted distribution is $q = [0.7, 0.3]$ then we can just take the cross entropy between p and q as the loss function
- The total loss will be the sum of this cross entropy loss for all the n output nodes

How will you train this network, how will you train this network, second last lecture of the course, you just, you don't pretend, you just know one algorithm that you can use for training and we don't think as if you have 100 algorithms on which again choose, Right? there's only one algorithm that you know, what is that? Back propagation, after all this is a neural network, Right? what else are you going to use to train a neural network. So, if I say backpropagation what do I need to define, loss function, what's the loss function, what are you trying to predict, probability distribution. So, what's the loss function going to be, cross entropy, sum of cross entropy, third option is cross entropy of sums, what is the loss function going to be? I don't have the problem with the answer, I have a problem with the confidence associated with the answer, what's the loss function going to be, still I have a problem with the confidence associated with the answer, only one contouring there, it's going to be some of the cross entropy, why hesitate? So, let's look at one of these guys, Okay? Now, this you need to understand, how everything falls in place. So, what is your training instance, it's x_1 to x_n given to you, Okay? That means, you know the true probability distribution that you need to predict at x_3, suppose x_3 is equal to 1 at the input, Okay? What's the you said cross entropy. So, for cross entropy, you need the true distribution and the predicted distribution all I'm asking you is what's the true distribution, if x_3 is equal to 1, then what's your true distribution, what kind of a random variable is x_3, a binary random variable, if I ask you the distribution of a random value, what do you need to give me, how many values do you need to give me, 2, probability of it, taking on the value 1, it taking on the value 0, I am telling you that it was one. So, what's the true probability distribution, zero, one the same as the two labels, Right? we have seen this a million times, is that fine? And what's the predicted probability distribution; it's what every one of you get here, Right? So, here you're going to predict some probability and one minus that probability. Right? So, let's say it's 0.7 and 0.3. Now, what's the loss function going to be cross entropy between P and Q, how many such loss functions will you have? This is the one associated with this prediction, how many such loss functions do you have? n of those. So, a total loss function is going to be sum of these n loss functions, is that clear, was it so hard, say at this point I expect you to come up with these answers, I mean it's like very straightforward and it's, it's not satisfying, if at this point you need to, you need to answer these questions Okay? So, that's exactly what's written on this slide. So, the loss function is well defined, we know how to deal with the cross entropy loss function, we know back propagation, the only catch here is that during back propagation the gradient should flow only to the, only to the, only to those connections which were active, this is similar to something that you have seen in, in, louder, someone is saying elasticum's, something similar to what you have seen in dropouts, Right? So, in dropouts also you remove some connections and then you just make sure that the gradients flow through those connections and in practice all while implementing the way, you will achieve this is, is always define this mask matrix. So, during forward propagation also the mass would be active and during backward propagation also the mask would be active, is that clear? Okay.

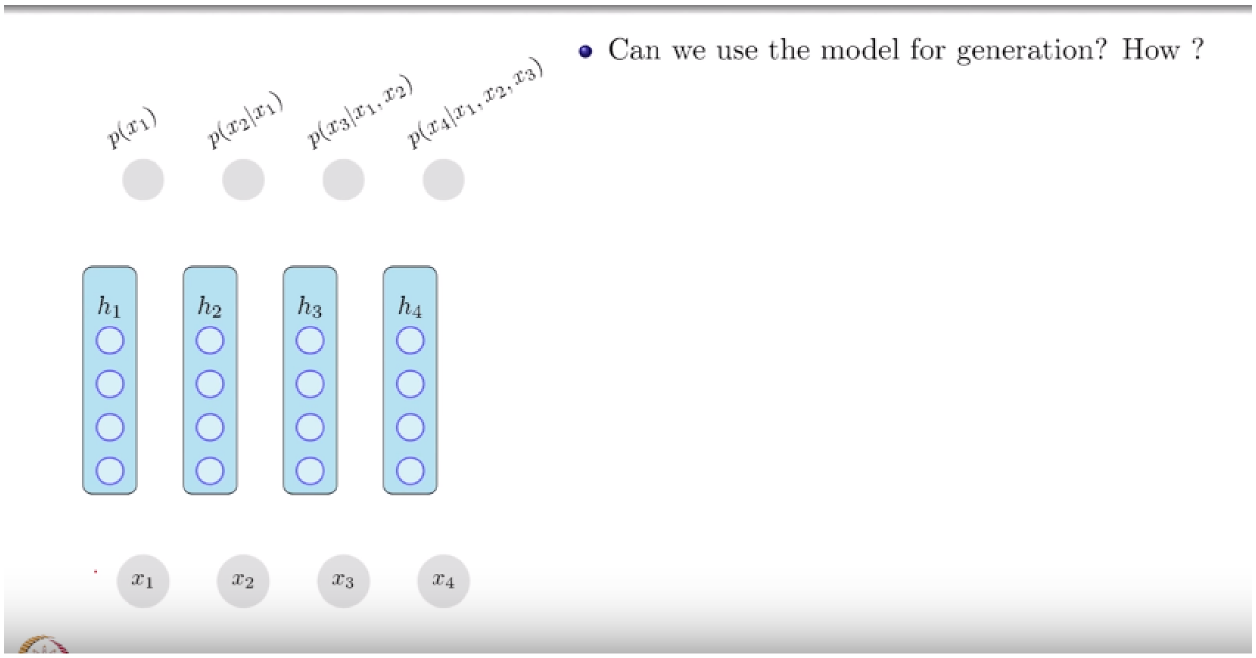
Refer Slide Time: (24:21)



- Now let's ask a couple of questions about the model (assume training is done)
- Can the model be used for abstraction? i.e. if we give it a test instance $\mathbf{x}$, can the model give us a hidden abstract representation for $\mathbf{x}$
- Well, you will get a sequence of hidden representations $h_1, h_2, ..., h_n$ but these are not really the kind of abstract representations that we are interested in
- For example, h_n only captures the information required to reconstruct x_n given x_1 to x_{n-1} (compare this with an autoencoder wherein the hidden representation can reconstruct all of $x_1, x_2, ..., x_n$)
- These are not latent variable models and are by design not meant for abstraction

So, just need to make sure that the gradients only flow along these two paths and not to any of these other guys, Okay? Is that fine? Okay. Now, let's ask a few questions about this model. So, whenever you are talking about generative models, what are the two things that we have been interested in, everyone, abstraction and generation. So let's ask the first question, can this model do abstraction, that is once the model is strained I have learned W b, all the v case and all the c case and I know I can do that because this is a neural network, I have defined the loss function, I can do back propagation, once I do that if I give it a new x, can I ask you to compute a hidden representation, what's the hidden representation of x, in each forward pass, how many hidden representations does this network compute, in each forward pass, how many hidden representations does it compute, n. Right? h_0 to $h_{n \text{ minus } 1}$ or h_1 to h_n, whichever you want to look at it. So, given an x, you are computing n hidden representations. Now, I'm asking you give me a hidden representation, which one will give me this is where you have to start bringing everything together, Right? So, the idea of hidden representations the first time we looked at it was in the context of auto encoders, Okay? The idea always behind and hidden representation is that it should capture the semantics of the input, Okay? Now, which of these n hidden representations that you have computed captures the semantics of the input, semantics of the input the way we have semantics is of course very vague, we have never defined it, but at least in terms of auto encoder, you know what we meant or it given that hidden representation, we could reconstruct the output. So, which of these n hidden representations is the abstract representation of your input, the last one, the first one, all of them, none of them? Last one, what is the last representation actually capturing, what is the job of the last representation, given h_1 to $h_{n \text{ minus } 1}$, reconstruct, Sorry! given x_1 to $x_{n \text{ minus } 1}$,reconstruct x_n, is that good enough for you, what did the hidden representation in auto encoder do, reconstruct x_1 to x_n, each of these hidden representations that you see here is only capable of constructing one of the inputs is that good enough. So, these models are not latent variable models by design they are not meant for abstraction, Right? You can of course come up with arbitrary things that you could take a concatenation of all these and these are, but that's going to be a very large vector n in 2d or you could take a sum of these are an average of these and soon but the fact is that these are not latent variable models they do compute some hidden representation, but by design they are not meant for abstraction. Okay? You could come up with ways of constructing hidden representations from this h_1 to h_n, but that's not very natural, in the sense of how we did it in the case of auto encoders, how we did it in the case of RBMs and how we did it in the case of VAEs, does that distinction make sense? Everyone raise your hands, if you make sense Okay? So, that's why these are not latent variable models and they are not designed for abstraction,

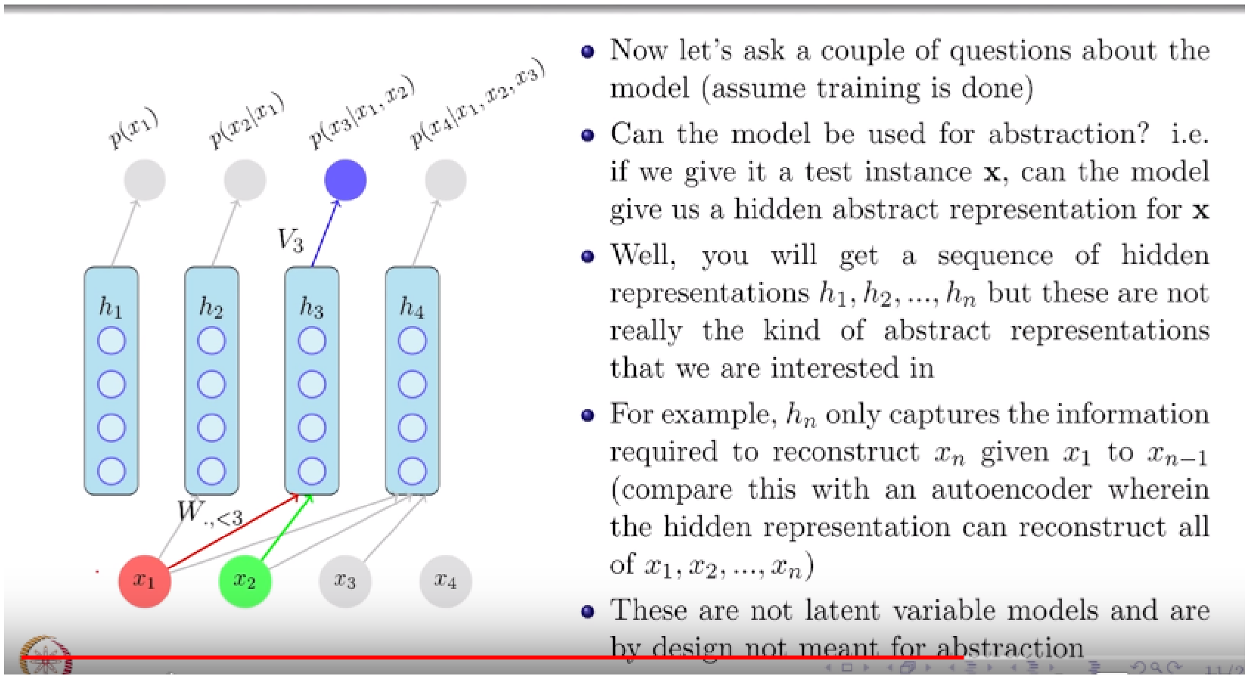
Refer Slide Time: (27:57)



- Can we use the model for generation? How ?

what's the second question I'm going to ask

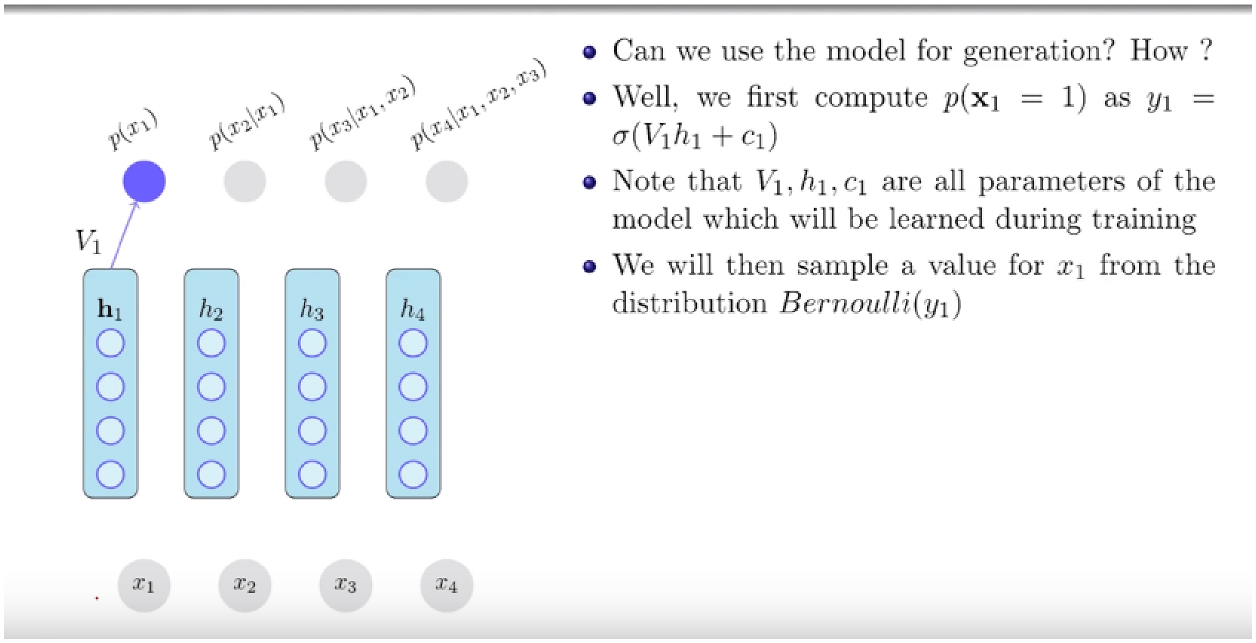
Refer Slide Time: (27:58)



- Now let's ask a couple of questions about the model (assume training is done)
- Can the model be used for abstraction? i.e. if we give it a test instance $\mathbf{x}$, can the model give us a hidden abstract representation for $\mathbf{x}$
- Well, you will get a sequence of hidden representations $h_1, h_2, ..., h_n$ but these are not really the kind of abstract representations that we are interested in
- For example, h_n only captures the information required to reconstruct x_n given x_1 to x_{n-1} (compare this with an autoencoder wherein the hidden representation can reconstruct all of $x_1, x_2, ..., x_n$)
- These are not latent variable models and are by design not meant for abstraction

can we do generation,

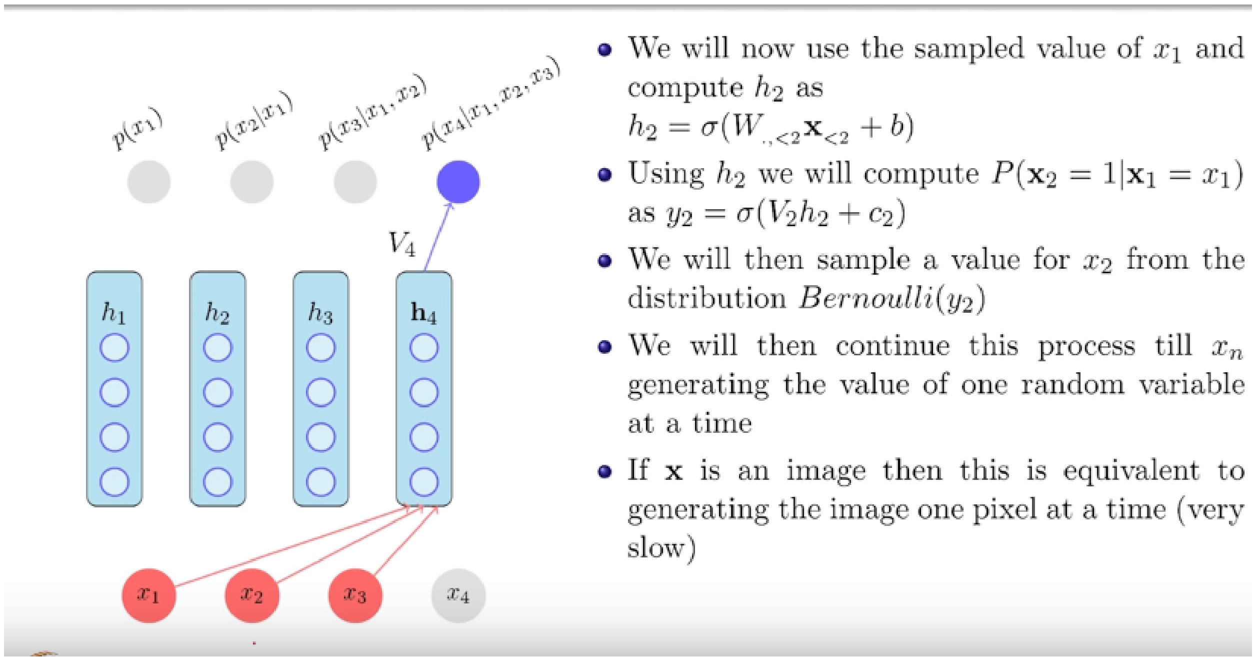
Refer Slide Time: (28:02)



- Can we use the model for generation? How ?
- Well, we first compute $p(\mathbf{x}_1 = 1)$ as $y_1 = \sigma(V_1 h_1 + c_1)$
- Note that V_1, h_1, c_1 are all parameters of the model which will be learned during training
- We will then sample a value for x_1 from the distribution $Bernoulli(y_1)$

what does it mean to do generation? No. I just want to generate samples, how will you do generation? So, you're saying you have an independent binary random variables, you could a sample independently from them. So, why do you need training and all that compute the probabilities, give me a procedure to sample from this distribution, I have actually given you an explicit distribution. Right? Given, Okay? Let before I talk about sampling if I give you an x. Right? So, I had asked you to train this on m-miss digits and attest time, I give you a new image and I ask you to compute p of x, can you compute it, can you compute p of x, where x is a vector actually or a matrix, Right? It's an image, can you compute the probability of x using this model, yes, no, everyone, how? You will compute all the, will compute all the all the n outputs, these are factors in your joint distribution, you just take the product of them and that gives you the total probability or the probability of the entire thing, is that fine? How many of you get that. Okay? Okay. That was just a separate question, now my other question is that how do you sample from this distribution, how do you generate a new image, first question, can you generate the entire image at one go, yes, no? No. What's the generative process going to be think in terms of graphical models, first find x_1, given x_1, find given $x_1$and x_2, Okay? Now, tell me how will you do it, not randomly initialize h_1 was a parameter of the model, its trained sample from p, x_1 but how do you compute p, x_1, h_1 is remember x_1 only takes h_1 as the input and what was $h_{1,}$ a dash of the model, a parameter of the model that means we have done training. Right? I'm assuming that the training is done. So, what's the first thing I'm going to do, I'm going to compute the priority of x_1 equal to 1, using this formula, is everything here known? v_1 is a parameter, h_1 is a parameter, c_1 is a parameter. Okay? I have computed this probability. Now, what do I do, sample from what does that mean, how will you sample, what kind of a distribution is this? Poisson distribution, richly distribution, beta distribution, normal distribution, everyone? Bernoulli distribution. How do you sample from a Bernoulli distribution, this will give you a value, Right? say let's value is 0.4 allows this a million times that ask is this another million times, if it's required, how will you sample from this, you will sample from a uniform distribution, you'll get a value between 0 to 1, if the value is less than 0.4 you will set it to 1 else to 0. So, now you have x_1, generated, you do you have the first pixel generated, now what will you do, you will set x_1 to that value and what will you do next,

Refer Slide Time: (32:02)

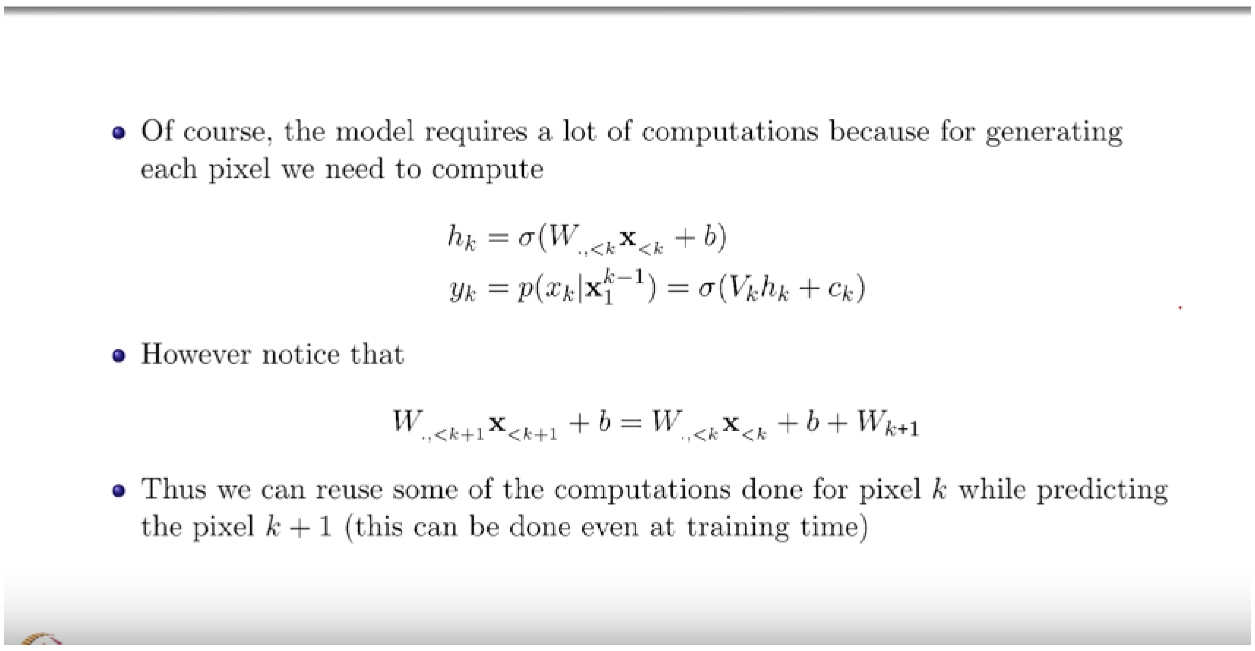


$p(x_1)$
$p(x_2|x_1)$
$p(x_3|x_1,x_2)$
$p(x_4|x_1,x_2,x_3)$
V_4
h_1
h_2
h_3
$\mathbf{h}_4$
x_1
x_2
x_3
x_4

- We will now use the sampled value of x_1 and compute h_2 as $h_2 = \sigma(W_{.,<2}\mathbf{x}_{<2} + b)$
- Using h_2 we will compute $P(\mathbf{x}_2 = 1|\mathbf{x}_1 = x_1)$ as $y_2 = \sigma(V_2 h_2 + c_2)$
- We will then sample a value for x_2 from the distribution $Bernoulli(y_2)$
- We will then continue this process till x_n generating the value of one random variable at a time
- If $\mathbf{x}$ is an image then this is equivalent to generating the image one pixel at a time (very slow)

now you can compute h_2 using x_1, you have a value of x_1 when you have sample it once you have x_1, you can compute h_2 because x_2 only depends on x_1 and the parameters of the model which you have already learned. So, again you will get some value say 0.6 again you will sample from this Bernoulli distribution and what will you get, you'll get a value of 0 or 1 and that you will put it into x_2, you will continue this process one random variable at a time or one pixel at a time and generate the whole image or set the configuration for all the random variables, does that make sense, is everyone clear about how you will sample from this distribution? Right? sample one value at a time, obviously this is very efficient, inefficient? Inefficient and is going to be very slow again, nothing wrong with it all these models come with their own pros and cons. So, this is one disadvantage of this model, the sampling is going to be very very slow, you'll have to do a lot of computations for generating one sample from this distribution, Okay? what's the advantage? You are working with an explicit or exact factorization you're not making any independence assumptions, Okay? Fine.

Refer Slide Time: (33:15)



- Of course, the model requires a lot of computations because for generating each pixel we need to compute

$$h_k = \sigma(W_{.,<k}\mathbf{x}_{<k} + b)$$
$$y_k = p(x_k|\mathbf{x}_1^{k-1}) = \sigma(V_k h_k + c_k)$$

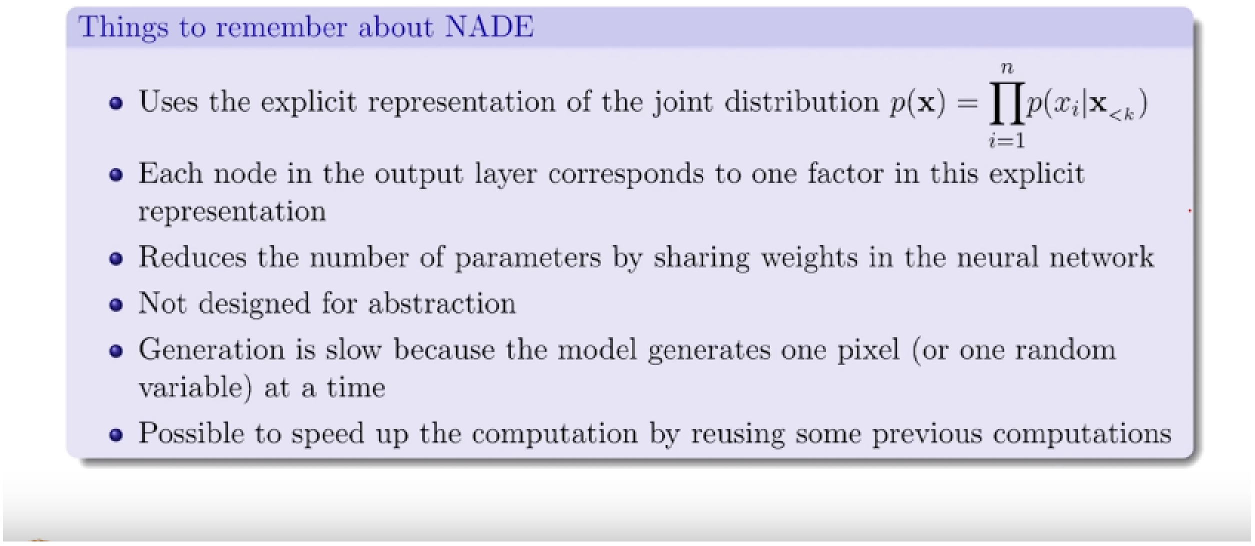
- However notice that

$$W_{.,<k+1}\mathbf{x}_{<k+1} + b = W_{.,<k}\mathbf{x}_{<k} + b + W_{k+1}$$

- Thus we can reuse some of the computations done for pixel k while predicting the pixel $k+1$ (this can be done even at training time)

Now, notice that this model requires many, many computations, Right? Because every time you have to first compute a hidden representation and then compute this $y_{k.}$ Right? Okay? But the good thing is that if I want to compute this for the k plus 1^{th} unit, I can just use whatever I'd done for the k^{th} unit and just add one simple thing to it. So, it should be $x_{k \text{ plus } 1}$. Right? does that make sense? Beside these three columns here and these three rows here. So, the first guy only depends on this product, the second guy is actually a sum of this plus, this second product. Right? So, once I have computed the first guy, I can just reuse that computation. Right? This is, this follows from basic linear algebra or matrix multiplication, everyone gets that. Right? How many of you don't get this, how many of you get this? So, you can just go back and look at it. So, you can do some of these computations efficiently it does not take care of the entire problem, but at least something can be done efficiently. Okay? Okay.

Refer Slide Time: (34:36)



Things to remember about NADE

- Uses the explicit representation of the joint distribution $p(\mathbf{x}) = \prod_{i=1}^{n} p(x_i|\mathbf{x}_{<k})$
- Each node in the output layer corresponds to one factor in this explicit representation
- Reduces the number of parameters by sharing weights in the neural network
- Not designed for abstraction
- Generation is slow because the model generates one pixel (or one random variable) at a time
- Possible to speed up the computation by reusing some previous computations

So, the things that you need to remember about NADE, are that it uses an explicit factorization for the joint probability distribution, each node in the output layer corresponds to one factor, in this explicit representation, it reduces the number of parameters which would otherwise have been exponential by sharing the parameters, w_nb. Right? you have shared the parameters in the neural network, there's it is not designed for abstraction it does compute some hidden representations, but it's not clear whether they are the hidden representations that you really want the Guru's, you could do something on top of that and try to get a hidden representation, but by design it's not meant for abstraction, and generation using the model is going to be very slow, because it's going to generate one pixel at a time and it's possible to speed up this computation by using the previously done computation, that's what I showed on the last slide. So, that's in a nutshell noodle autoregressive density estimator, what's auto regressive noodle and density estimator is fine, Autoregressive it's using the previous prediction to predict the next prediction, Right? Regression is fine, auto comes from the previous query is that fine. Okay?