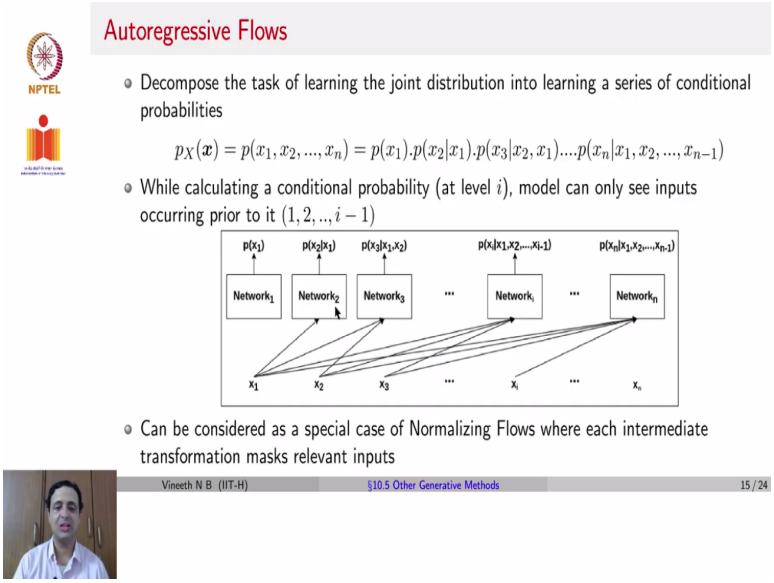
Deep Learning for Computer Vision
Professor Vineeth N Balasubramanian
Department of Computer Science and Engineering
Indian Institute of Technology, Hyderabad
Lecture 67
Beyond VAEs and GANs:
Other Methods for Deep Generative Methods - 02

(Refer Slide Time: 00:12)



The second kind of Deep Generative model that we will discuss in this lecture is Autoregressive flows. Autoregressive flows come, as the name suggests, from Autoregression. Autoregressive models are used in time series data in standard machine learning to look at the past n steps of a time series data and predict the next value. For example, if you look at the stock price for the last ten days, can you predict the stock price on the 11th day? It can be solved by using autoregressive models such as ARMA, ARIMA NARIMA, so on and so forth.

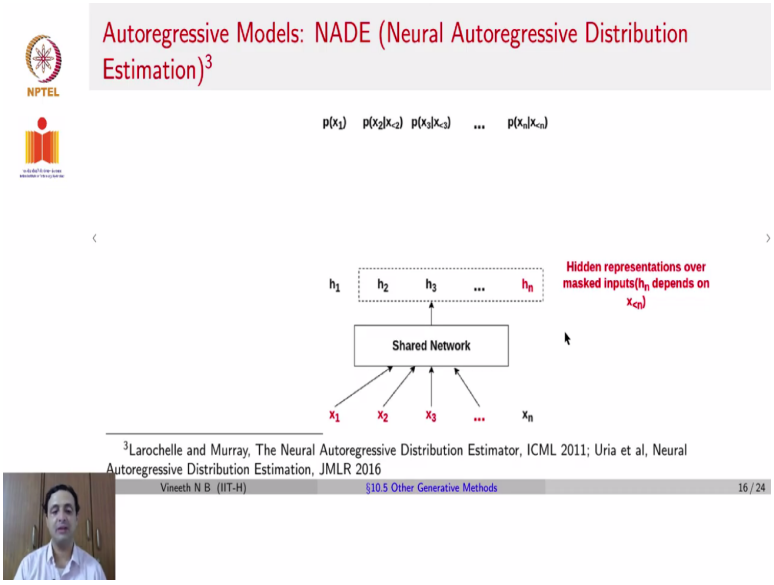
These are popular time series models for this scenario. What we are talking about here is an extension of that thought process. However, now, we want to generate data and not just predict an outcome. Once again, this is a transition from supervised learning to unsupervised learning. So let us see how to use Autoregression for a generation. So we decompose the task of finding the PDF of the real-world data as $p(x_{!}, x_{2,} \ldots . x_{n})$, which is the probability density function of the

real data given to us. It is written as $p(x_!)$ into conditional $p(x_2 \mid x_1)$, and so on, till $p(x_n \mid x_1,\ldots., x_{n-1})$.

So, while calculating a conditional probability at a certain level, the model can only see inputs occurring prior to it. Let us see how this would work. So, if you had x_1, to x_n, as different dimensions have an input, x_1 is used as input to a second network x_2, predicting the probability of x_2 given x_1. Remember that probability, depending on the values x_2 can take, can be learned using a mean squared error or cross-entropy or any other loss that we have seen so far. x_1 and x_2 contribute to the third network to learn the probability of x_3 given x_1, x_2, so on and so forth. For a later x_i given x_1 to x_{i-1}, and finally, x_n given x_1 to x_{n-1}.

So, you could consider that autoregressive flows are a special case of normalizing flows, where each intermediate transformation masks certain inputs. You are not going to look at all inputs in every step. But in each, if you assume that network 1 through network n were different functions, similar to normalizing flows. Each of these networks looks at only a certain part of the input and not the complete input. You could look at it that way, although that is not exactly the way it is implemented.

(Refer Slide Time: 03:30)

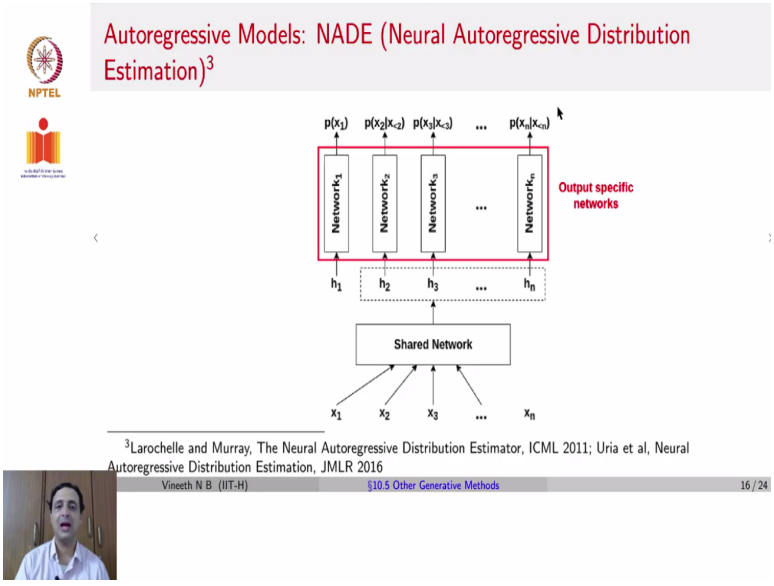


So, one of the popular methods for autoregressive models is called NADE. NADE stands for Neural Autoregressive Distribution Estimation. Here is how NADE works. You have inputs x_1 to x_n in dimensions of every input vector. Similarly, you have $p(x_1)$, $p(x_2 \mid x_{<2})$ - the probability of $x_2$given all the random variables less than 2, $p(x_3 \mid x_{<3})$, so on and so forth till $p(x_n \mid x_{<n})$.

We are trying to understand the probability density function of the real-world data, which is a product of all of these. So, we provide all of these inputs to a shared neural network with as many layers as you choose. The output of that neural network is then passed on to a layer of hidden representations obtained over masked inputs. That is h_n, which is one of the nodes here, depends on all the x's, which is less than n.

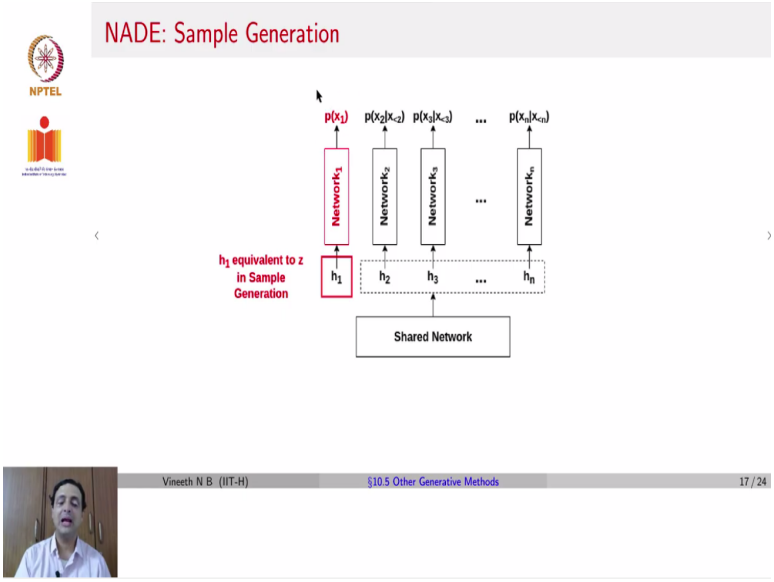
So, here is how it is done. So, x_1 will only be considered to get h_2, x_1 and x_2 will be considered to get $h_{3,}$ and x_1, x_2 x_3 and so on, all the way till x_{n-1} will be considered to get h_n. So one would have to mask the inputs to ensure that the corresponding inputs reach the corresponding nodes in the hidden layer.

(Refer Slide Time: 05:17)



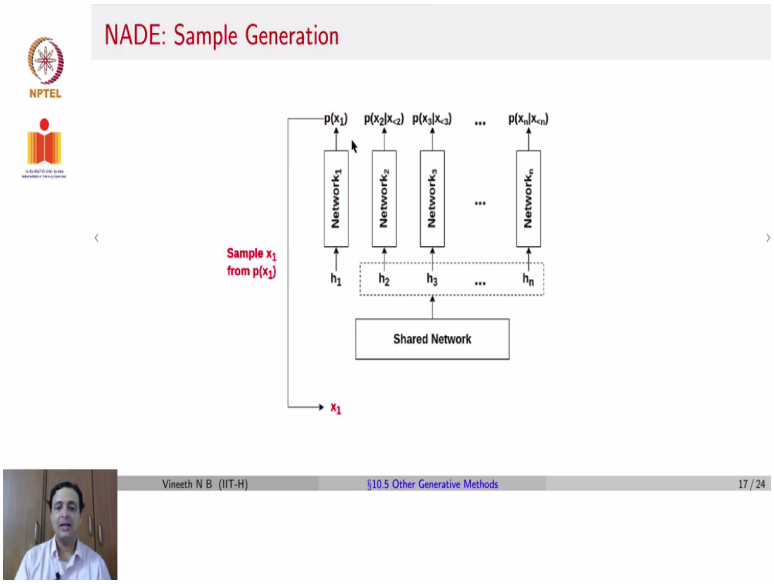
What happens after this? Each of these h_1's to h_n's are passed through different networks to get $p(x_1)$, $p(x_2 \mid x_{<2})$, etc.

(Refer Slide Time: 05:31)



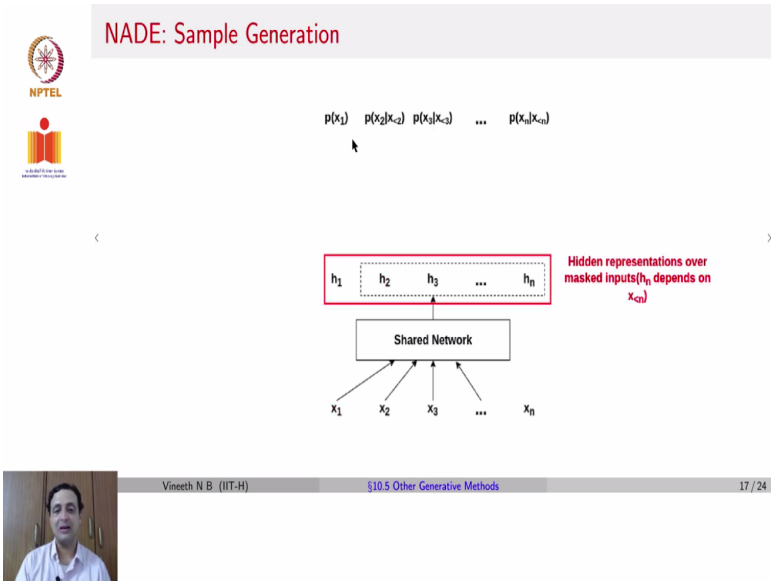
What happens at inference time? How do you generate? So once you have trained such a network, remember in GANs, we knew how you could generate an image after training a GAN, you would just sample a vector from a Gaussian, send it through a generator and get an image. How do you do this with NADE? It is similar. We are going to assume that h_1 is equivalent to z in sample generation. You could now assume that h_1 comes from a Gaussian; you sample a value from there, send it through the first network, you would get a specific x_1.

(Refer Slide Time: 06:09)



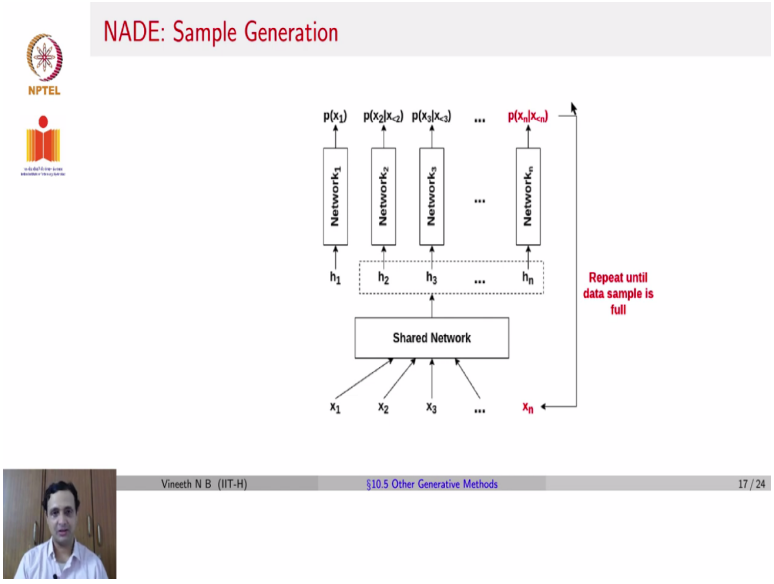
That x_1 is now.

(Refer Slide Time: 06:11)



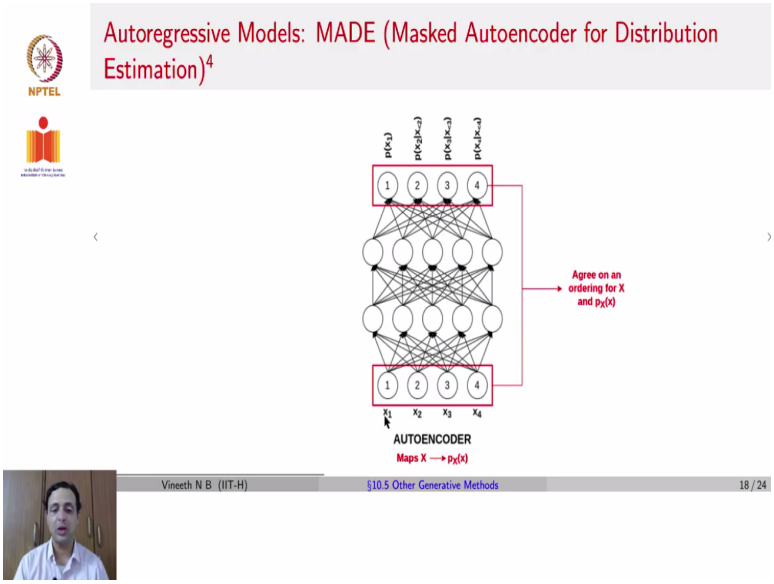
fed to get h_2

(Refer Slide Time: 06:16)



That is used to get $p(x_2)$, then x_1 and x_2 together is fed to get h_3, which is used to get $p(x_3)$, then fed in to get. You continue this process until the last network predicts $p(x_n \mid x_{<n})$. You can keep repeating this until the complete data sample is generated.

(Refer Slide Time: 06:43)

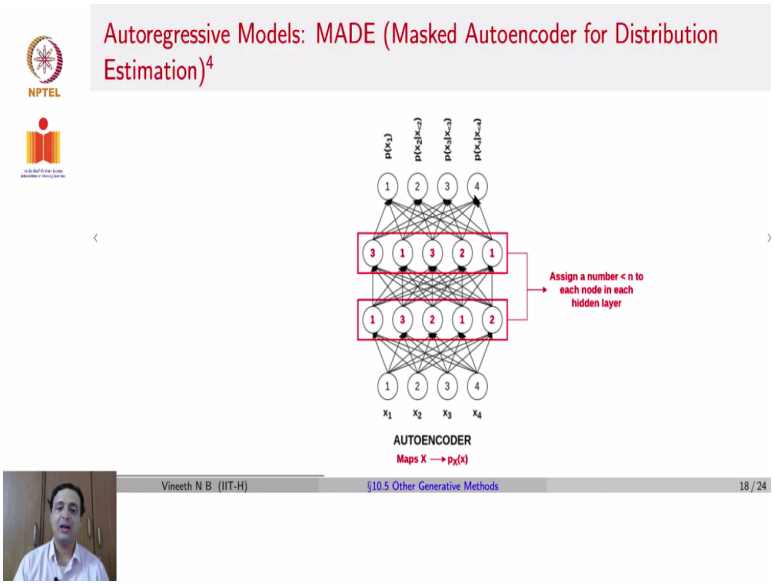


A variant of NADE, which is an improvement over NADE, is called MADE. MADE stands for Masked Auto Encoder for Distribution Estimation, which improves upon the idea of NADE in a different way. In this case, we use an autoencoder to achieve the same effect as NADE. So you

have x_1, x_2, x_3 x_4, you have $p(x_1)$, $p(x_2 \mid x_{<2})$, $p(x_3 \mid x_{<3})$, so on and so forth till the last random variable.

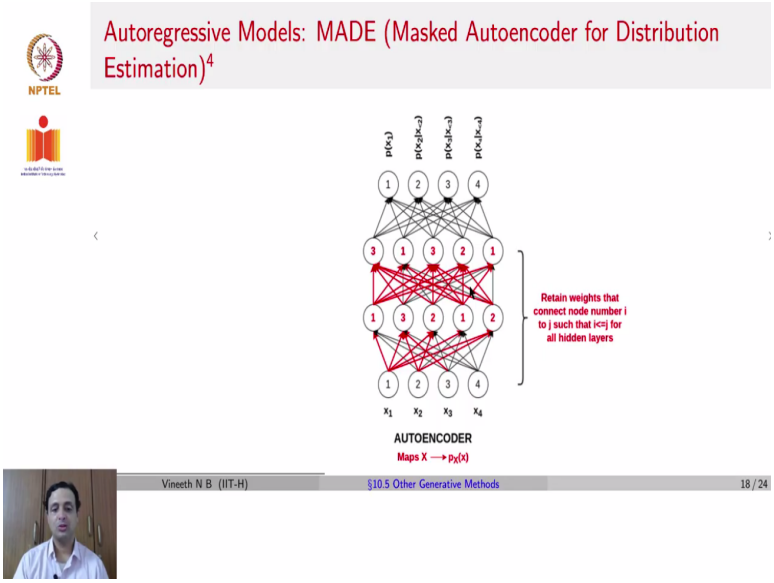
So how do we adapt NADE to this kind of architecture? There is an interesting methodology for this. Firstly, you give an order to each of your input nodes and your output nodes. In this case, we will keep it simple and say 1234 and 1234.

(Refer Slide Time: 07:48)



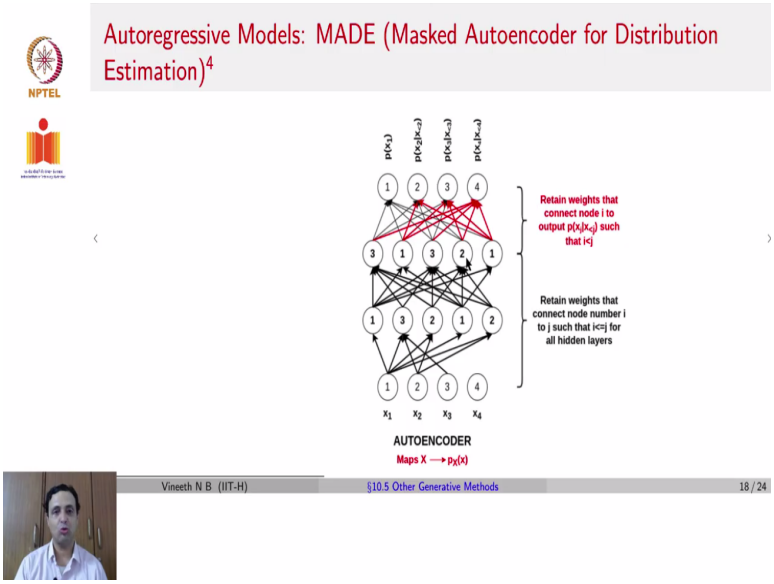
Once you do this, you also give an ordering for each hidden layers nodes. However, each hidden layers nodes should get an ordering less than ‘n’. So, for example, in this hidden layer, n is 4. So this means each node should get a number 1, 2 or 3. It has to be less than 4. Similarly, in the next hidden layer, also each node gets a random number between 1, 2, and 3. It has to be less than 4. Why do we do this? What do we do with this? Let us see that now.

(Refer Slide Time: 08:22)



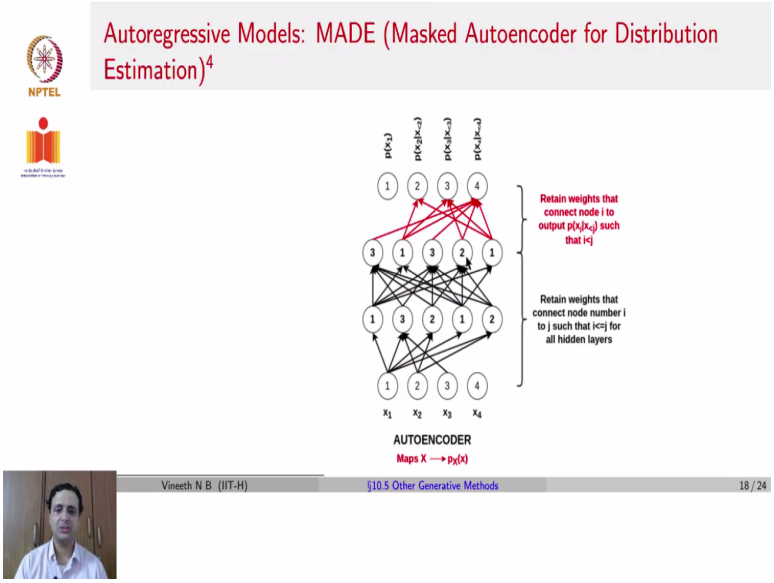
So, now, we do retain weights that connect node number 'i' to 'j', such that $i \leq j$ for all hidden layers. What do we mean here? If you have a hidden node 1, it should be connected to an input node less than or equal to 1, which means only 1 will connect to 1. So, the second node is 3. It will be connected by 1, 2 and 3. Similarly, the hidden node labelled with 2 will be connected with 1 and 1 and 2, so on and so forth.

(Refer Slide Time: 09:07)



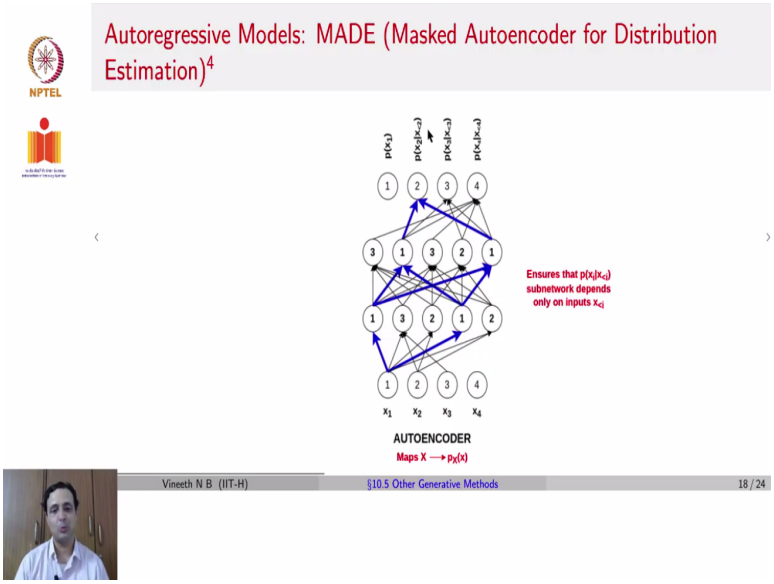
So we retain only those weights and discard all other weights of those layers. We also do this for the last layer, where we only retain weights that connect node 'i' in output $p(x_i \mid x_{<j})$ such that 'i' is less than 'j'. So, if you have 2, you only connect that with 1. Similarly, 3 is connected with all 1s and 2s, so on and so forth.

(Refer Slide Time: 09:38)



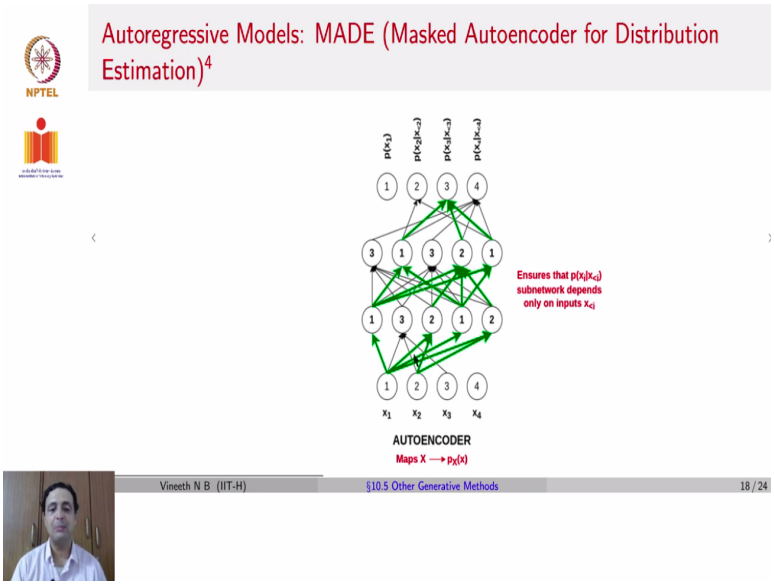
And once again, you eliminate all other weights. What did we achieve through this process?

(Refer Slide Time: 09:46)



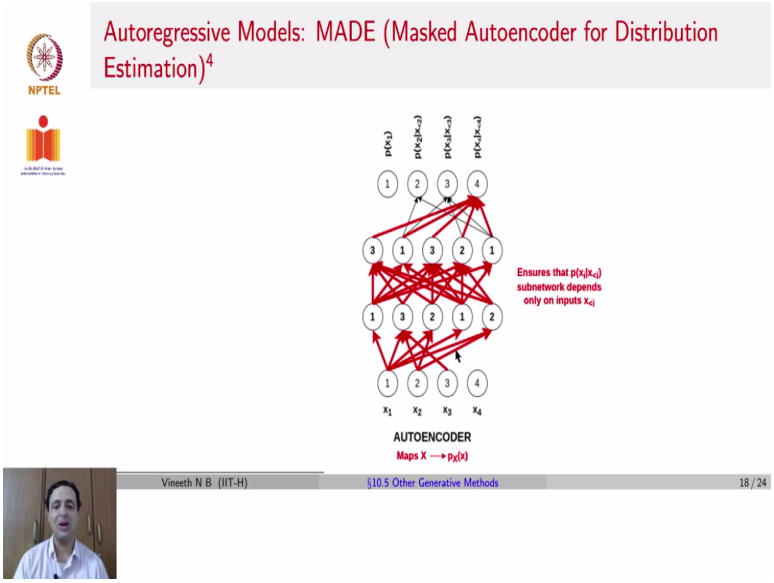
We ensured that if you had $p(x_i \mid x_{<i})$, all the other random variables less than that value 'i' depends only on inputs less than x_i. So this procedure ensured that if you had $p(x_2 \mid x_{<2})$. You notice the blue arrows that it is connected to only nodes labelled 1. It is not connected to even a node with label 2, which was our goal in the first place, ensuring that only input x_1 influences $p(x_2)$, given all the other random variables less than 2.

(Refer Slide Time: 10:32)



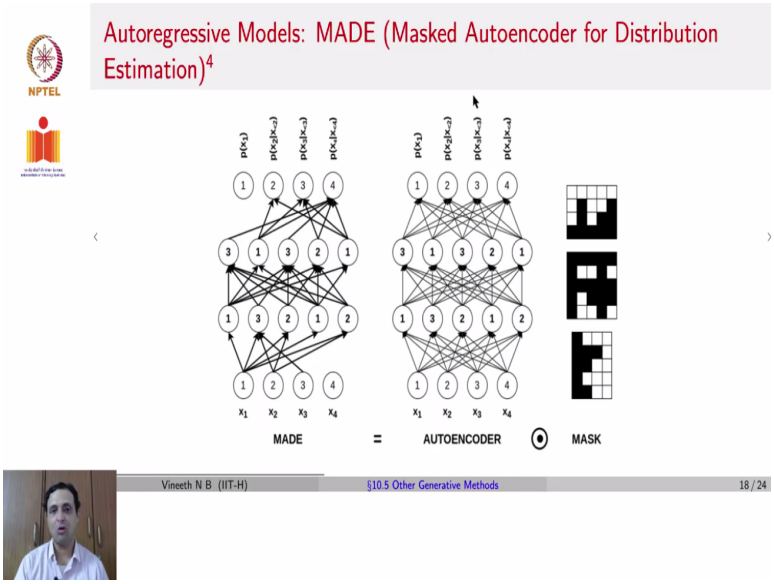
Similarly, you can say it for $p(x_3 \mid x_{<3})$, which depends only on all nodes labelled 1 and 2.

(Refer Slide Time: 10:38)



And for $p(x_4 \mid x_{<4})$, which depends on nodes labelled 1, 2, and 3.

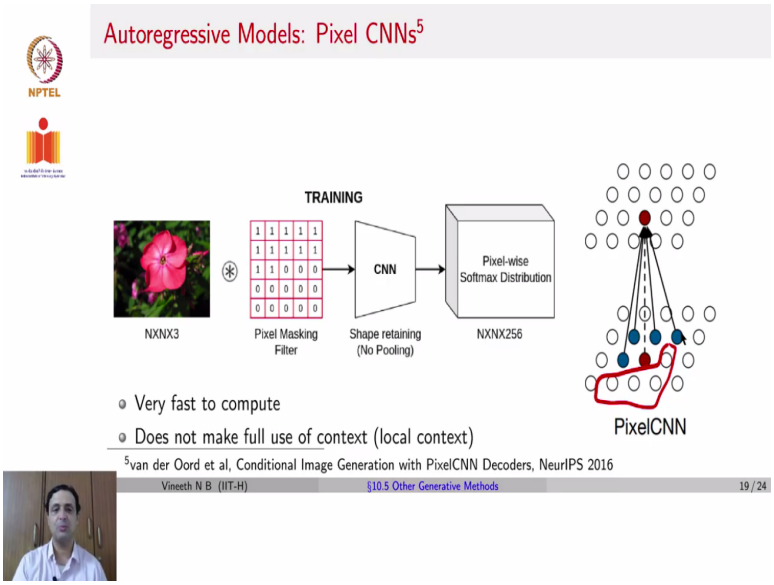
(Refer Slide Time: 10:45)



In the end, the autoencoder has only a few weights saved, and it appears when you implement that you have a full autoencoder. Still, you use a mask only to retain a certain set of weights and discard the other set of weights while forward propagating or doing backpropagation. That gives you MADE, and through this process. At the same time, you train this network. You are

automatically learning each of these functions, your $p(x_1 \mid x_{<1})$, $p(x_2 \mid x_{<2})$, and so on, which together gives us the real density function of the given data.

(Refer Slide Time: 11:29)

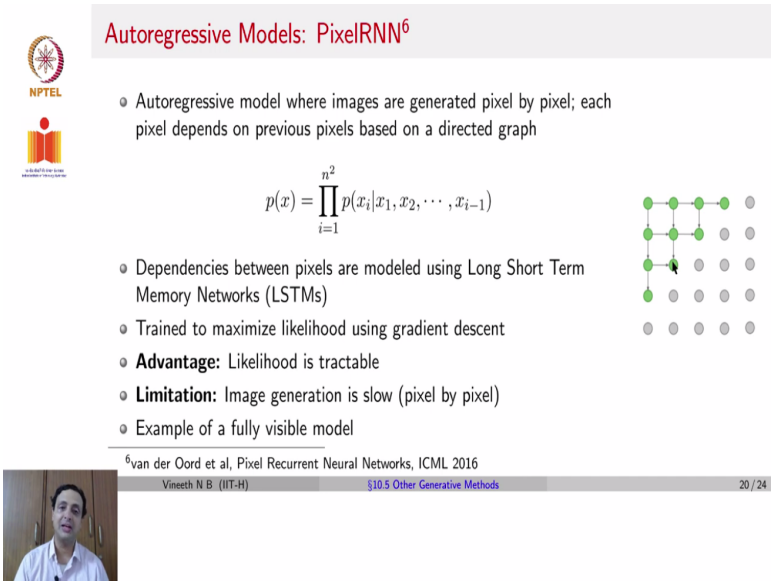


Another class of models for Autoregression are known as PixelCNNs. These were developed in NeurIPS 2016. Given an image, our overall idea would be to send it through a CNN without disturbing the shape and get a pixel-wise softmax distribution to generate the same data. But the way we are going to do it is to introduce a pixel masking filter to say which pixels should be used to predict the value at a specific pixel in the output.

For example, to predict a specific pixel here, you may not need all the pixels to predict the pixel value at one specific location. You may only need a certain set of pixels in that neighbourhood around the pixel you are predicting. That is the idea for pixel CNNs. It is very fast to compute. As you can see, it is not a very complex procedure. However, pixel CNNs may not make use of the full context.

For example, pixel masking is done to predict a particular pixel. It may look at a set of pixel values that occurred before its presence. So, in this case, if you are trying to predict this red pixel here, you may be using all these blue pixels, which is a sense of a local context. But it is also missing the other pixels in this region, which also form a local context. So it could lose some information, while generation, although it is extremely fast to compute.

(Refer Slide Time: 13:18)



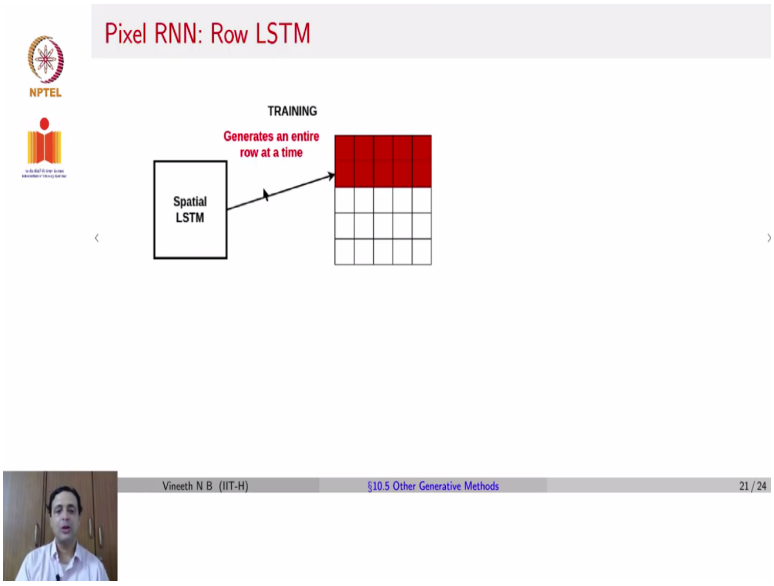
A variant of this by the same authors, which came in ICML 2016, is known as pixelRNNs and pixelRNNs have a very similar idea. However, the generation is done using LSTMs instead of CNNs. It is an autoregressive model, where images are generated pixel by pixel. Each pixel depends on previous pixels based on a directed graph. So you have the overall joint distribution $p(x)$ given by, i is equal to 1 to n^2, assuming an $n * n$ image, probability of x_i given all the other pixels until that particular pixel.

So you can see here that for any specific pixel, you look at all the other pixels that came before it to influence the generation of the value at that pixel for any specific pixel. So these dependencies between pixels are modelled using LSTMs. That is why the name pixelRNN. The learning is very similar to other models that we have seen in this lecture, maximising the likelihood of using gradient descent.

The likelihood is tractable because we are using an LSTM. We know how it works. We just have to structure it accordingly. An LSTM is trained the same way that we saw LSTMs earlier. The image generation in pixelRNNs can be slow because you have to use an LSTM to generate each pixels value. Unlike GANs, where the entire image is generated in one shot.

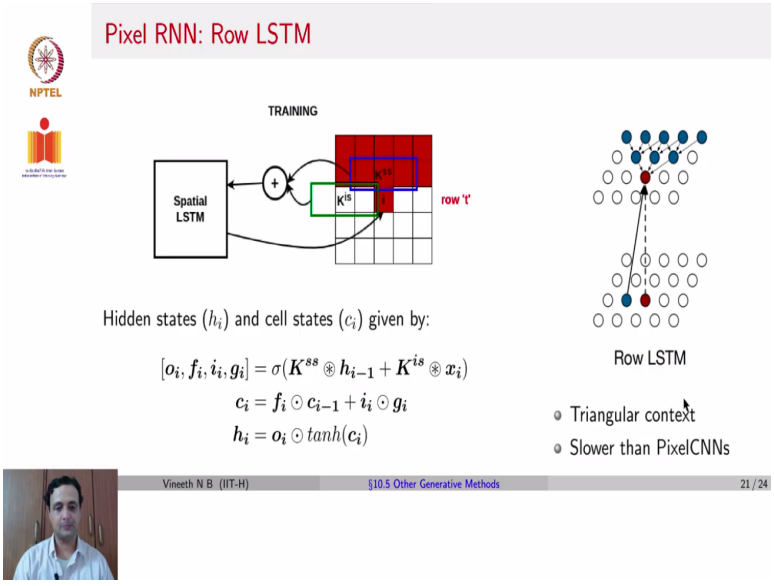
PixelRNNs can also be considered as an example of a fully visible model. There are no latent variables. pixelCNN is also an example of that. For that matter, most Autoregressive models turn out to be fully visible models, where you do not model any latents per se as part of the method.

(Refer Slide Time: 15:31)



Let us look at how the method works. The PixelRNN has two variants; one is known as a Row LSTM, where each LSTM generates an entire row of pixels at a time.

(Refer Slide Time: 15:52)



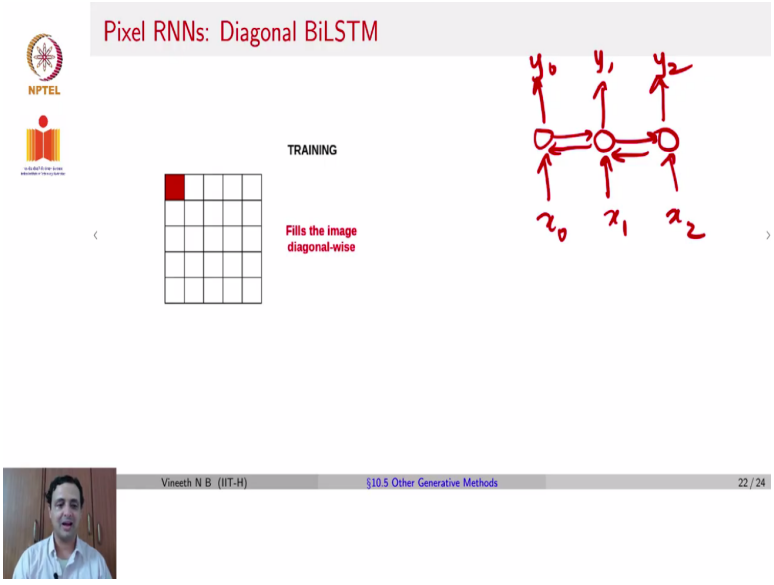
So when you have to generate a specific row 't', let us see how to generate one specific value in that row 't', which would be the output of a one-time step of the LSTM to generate that particular row. Where should the LSTM look? How should the LSTM operate to generate a particular pixel 'i'? We consider the set of pixels before that pixel in that row to be K^{is} and the pixels just above that row in the immediate neighbourhood of 'i' to be K^{ss}.

They both become hidden state contexts, very similar to LSTM to generate the current pixel. So you would have the hidden states and cell states to be given by some weight $K^{ss} * h_{i-1} + K^{is} * x_i$, a sigmoid that will give you all your different gates. The rest of it is very similar to how an LSTM operates. So your inputs are based on you K^{is} and K^{ss}. Those are the values you get to generate 'i'.

One could look at this as if you are generating a particular pixel, say the red pixel. You get the context from the previous pixel in the same row and the top 3 pixels in the previous row. However, those top 3 pixels received inputs from the 5 pixels in the previous row. Hence, you could look at this red pixel that you generate to have a triangular context in its generation.

When you look at this model, it will be far slower than pixel CNNs because of its approach to generation.

(Refer Slide Time: 17:52)

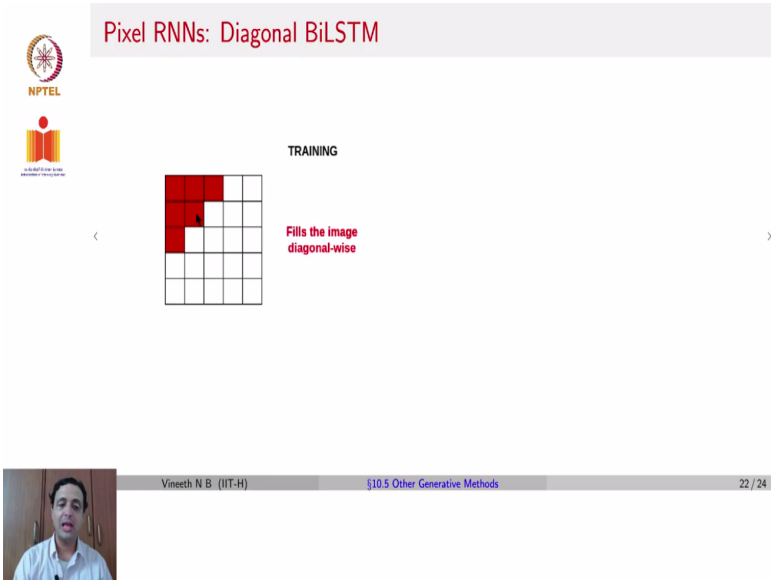


Another variant that was proposed in PixelRNNs is known as the Diagonal BiLSTM. In Diagonal BiLSTM, the approach is similar to what is known as a Bidirectional RNN. A Bidirectional RNN is an RNN where, if you had your traditional RNN to be this way, let us assume three timesteps. A bidirectional RNN is where your inputs are x_0, x_1 x_2 , and the outputs are y_0, y_1, y_2. We know from a standard Vanilla RNN that x_0 influences y_0, x_0 and x_1 influence y_1, x_0, x_1, and x_2 influence y_2.

That is your standard RNN. In a bidirectional RNN, you also go the reverse way. All these arrows can also operate reversely, which means you can reverse the entire RNN. You can now say y_2 is generated based only on x_2, $y_1$is generated based on $x_1$and x_2 , and y_0 is based on x_0, x_1, and x_2 altogether.

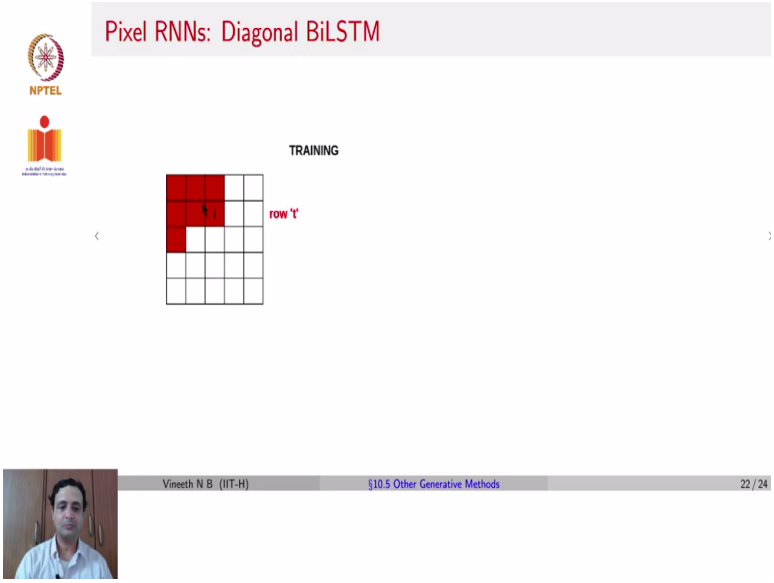
How do you learn such an RNN? You will learn the weights in both directions. You would get two different sets of outputs y_0, y_1, and y_2. You can average those outputs, which may give you a better sense of what your output should be, especially when the direction in the sequence may not matter when there could be context from both directions. So that is the idea used in the diagonal BiLSTM variant of Pixel RNNs.

(Refer Slide Time: 19:46)

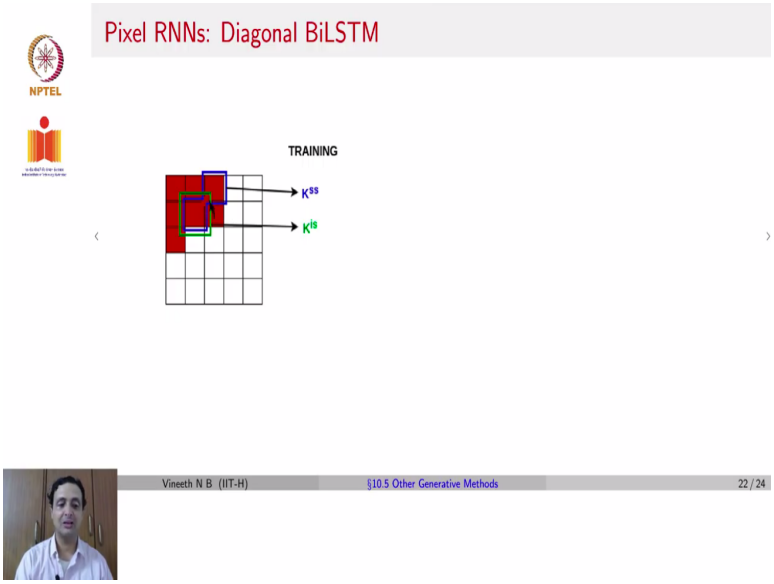


In this case, the image is filled diagonal-wise. So you can see here that each diagonal is filled at a time and when you fill one pixel at that particular location.

(Refer Slide Time: 19:57)

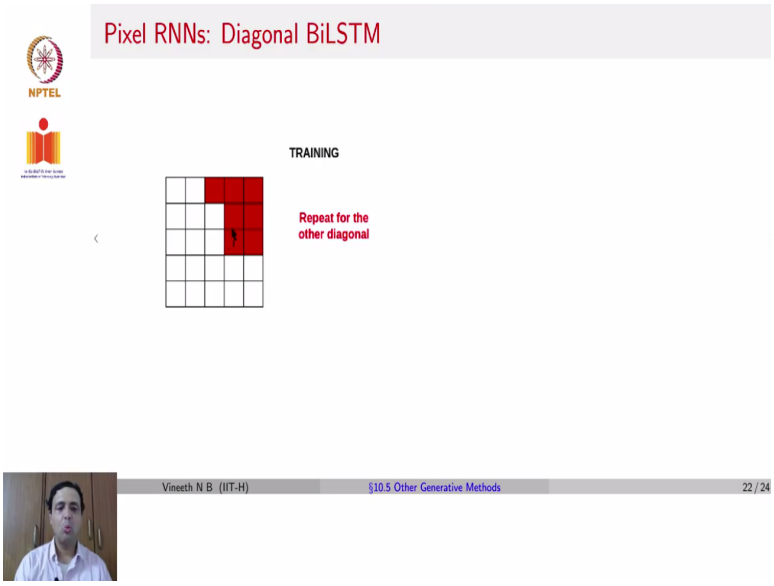


(Refer Slide Time: 20:00)



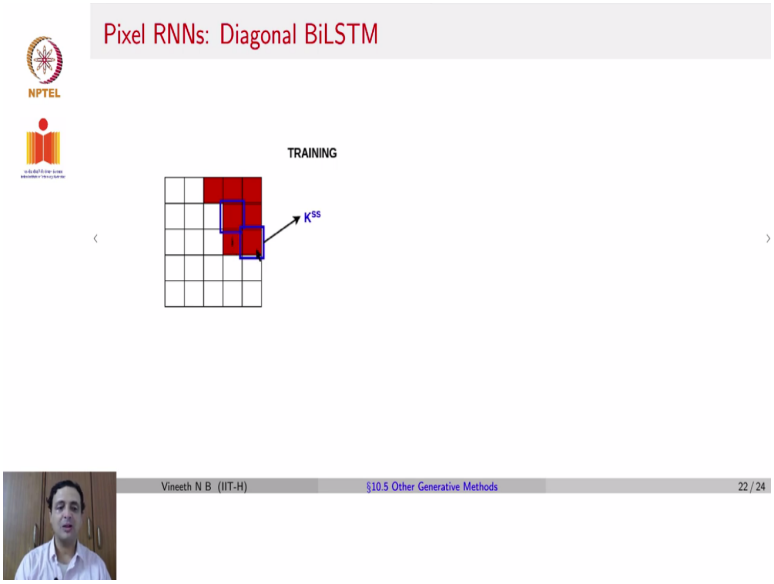
Those are the values that are given us context to fill that ‘i’. In turn, these 2 pixels are denoted in blue squares dependent on the previous pixels, so all of them would eventually influence the generation of the value at this pixel ‘i’.

(Refer Slide Time: 20:20)



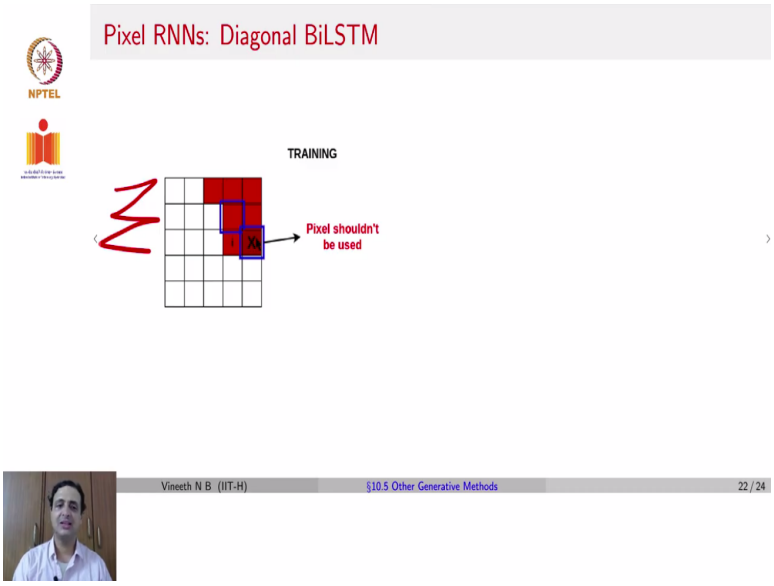
We now repeat this process for the diagonal from the other side, very similar to a Bidirectional RNN. So now, let us see if you had to generate this particular pixel 'i' here. What are the pixels We would consider?

(Refer Slide Time: 20:35)



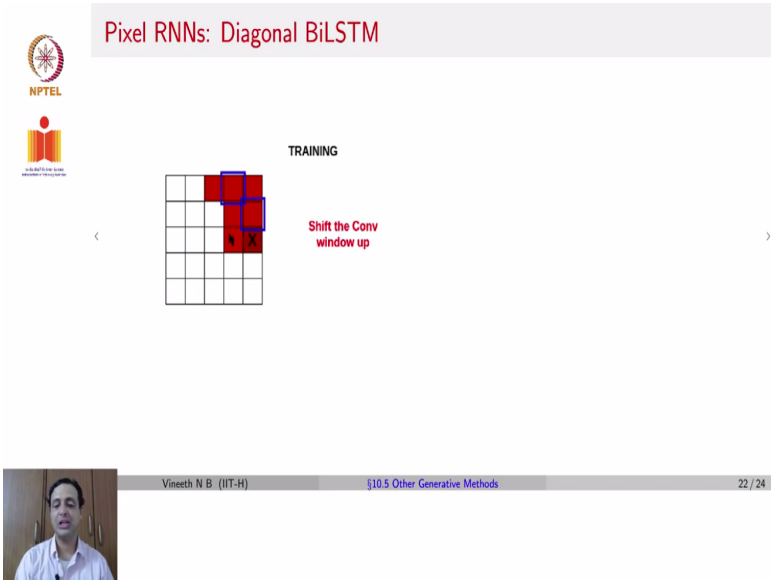
You would say the immediate neighbours, which means the pixel on top and the pixel to the right.

(Refer Slide Time: 20:42)



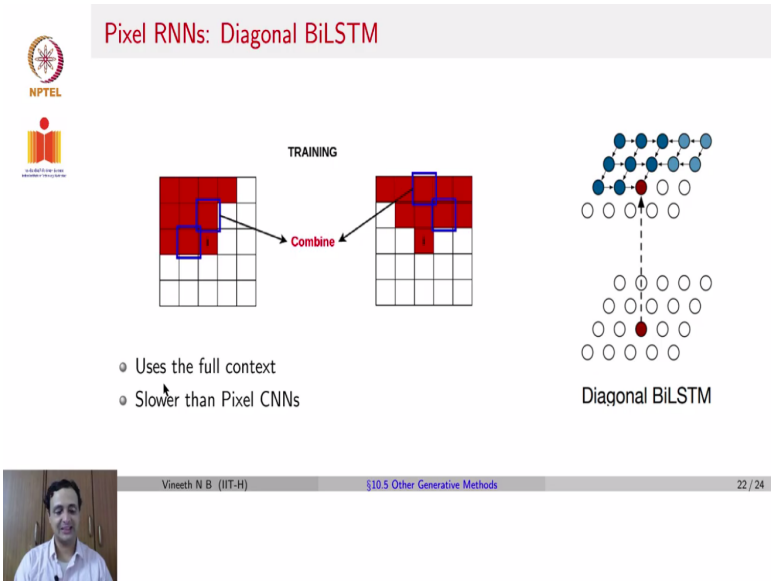
Unfortunately, you cannot use the pixel to the right. It has not been generated yet, according to rasterization of the image. When we say rasterization, we mean this exact pattern involved in generating an image, first row, then the second row, third row, and so on. So you cannot use this pixel. So how do we overcome this?

(Refer Slide Time: 21:06)



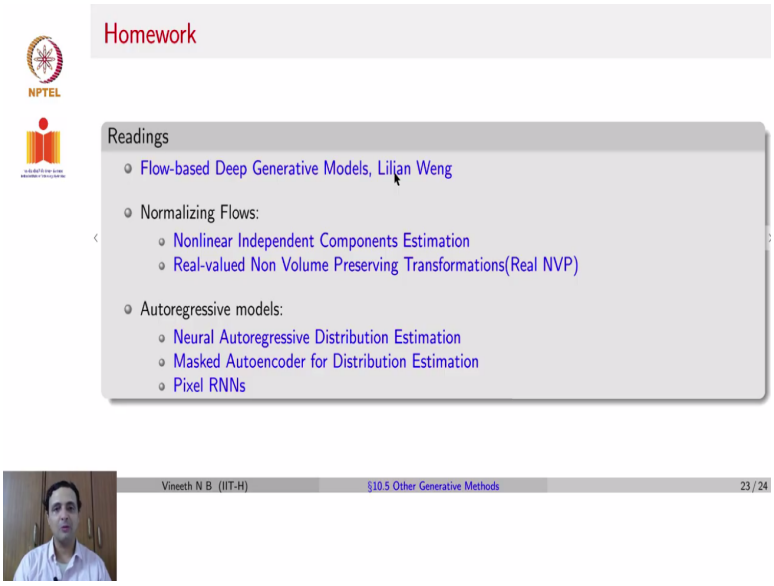
So in a Diagonal BiLSTM approach, when you come from the other diagonal, you use these top 2 pixels as the context to generate this particular pixel ‘i’. That is the only change that occurs.

(Refer Slide Time: 21:20)



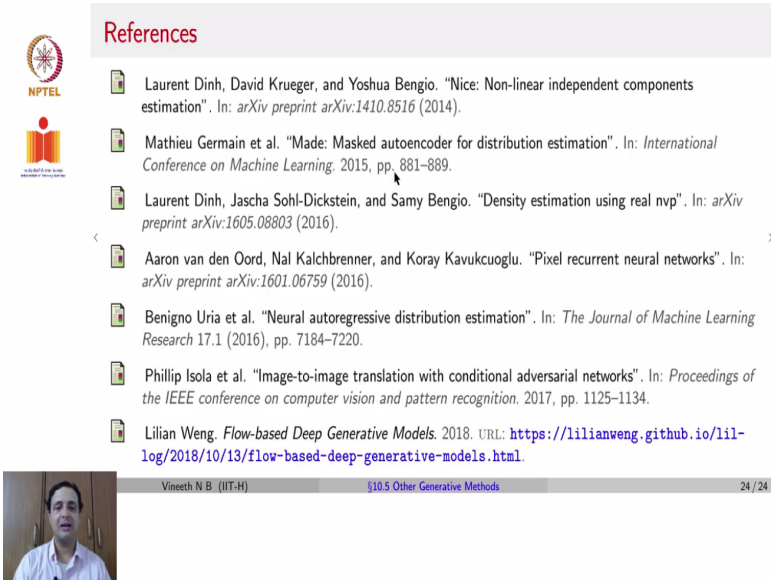
You now combine the context you get from both of these and then get your generation at the value 'i', which we talked about for a Bidirectional RNN. So now you can look at this pixel, the red pixel in the middle, using the complete context from both directions. So to generate the red pixel, you consider all the dark blue pixels that come from 1st diagonal and these light blue pixels that you populate from the other direction, which give us the complete context. Once again, here, you can see that these would be slower than pixel CNNs.

(Refer Slide Time: 22:07)



So your homework for this lecture is a very nice blog on "*Flow-Based Deep Generative models*" by Lilian Weng. Please do read it. If you are interested in knowing each of these methods, please read the respective papers linked here. There are also newer methods, such as Glow for Normalizing flows, which are also covered in this blog if you would like to know more.

(Refer Slide Time: 22:32)



And here are the references.