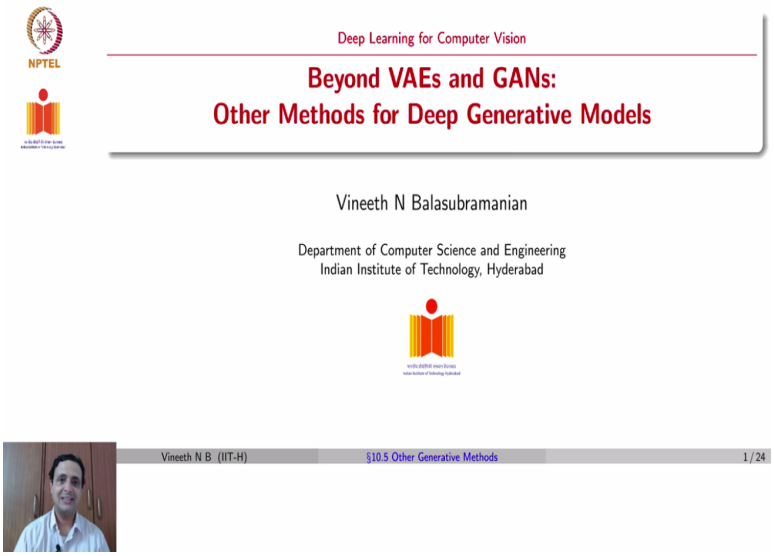
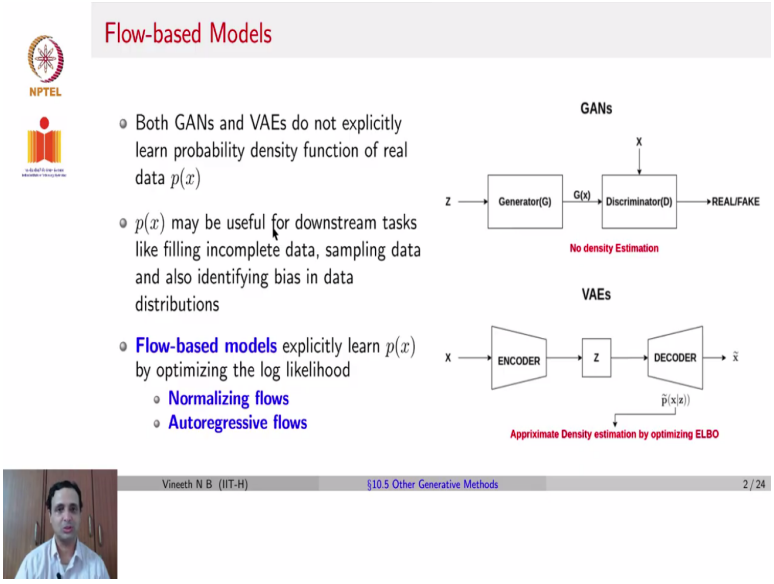
Deep Learning for Computer Vision
Professor Vineeth N Balasubramanian
Department of Computer Science and Engineering
Indian Institute of Technology, Hyderabad
Lecture 66
Beyond VAEs and GANs:
Other Methods for Deep Generative Methods - 01

(Refer Slide Time: 00:15)



Although VAEs and GANs are the most popular kinds of Deep Generative models, other methods have also been successful over the last few years. Let us see a few of them in the last lecture for this week.

(Refer Slide Time: 00:34)

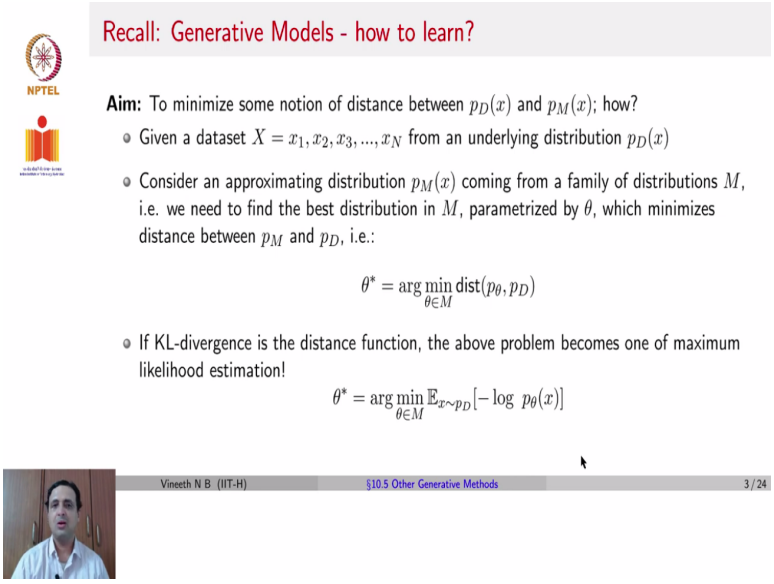


These models are generally known broadly as Flow-based models. How do they differ from GANs and VAEs? There is a significant difference between them and GANs and VAEs. Both GANs and VAEs do not explicitly learn the probability density function of the real data. In the case of GANs, we already saw that the density estimation is implicit. You do not explicitly assign a probability density function and try to estimate it in a GAN.

In the case of VAEs, we get an approximate density estimation by optimizing your evidence lower bound using variational inference. In both cases, you do not get the exact density function. However, the exact PDF $p(x)$ of your real data may be useful for many tasks, such as missing values, sampling data, or even identifying bias in data distributions. Knowing the density function could be handy for these kinds of tasks.

So, the methods that we will discuss in this lecture are methods that estimate the real density of the provided training data. These can be categorized into two different kinds: Normalizing Flows and Autoregressive Methods or Autoregressive Flows.

(Refer Slide Time: 02:13)

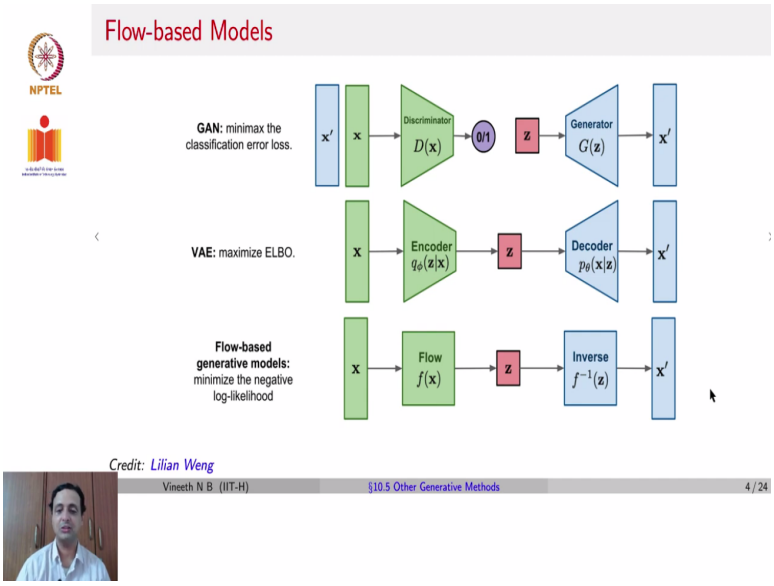


Let us start with normalizing flows. Recall that in the first lecture for this week, we tried to ask this question, that if we had a dataset of N data points coming from an underlying distribution p_D (x), we wanted to find a θ belonging to a family of distributions M in such a way that the distance between p_θ and p_D is minimized by that choice of parameterization in M. We also noted at that time, that if the distance here was replaced by KL-divergence, this turns out to become a maximum likelihood estimation problem, or by minimizing the negative log-likelihood. Why is this important?

All methods that we have covered so far while training neural networks can be looked at to minimise negative log-likelihood. Whether training a neural network using mean squared error or training a neural network or an LSTM using cross-entropy loss for a classification problem, one can show that both of these finally amounts to minimizing the negative log-likelihood.

So we are now saying that we could use a similar approach even in an unsupervised learning setting. Where do we go from here?

(Refer Slide Time: 03:54)

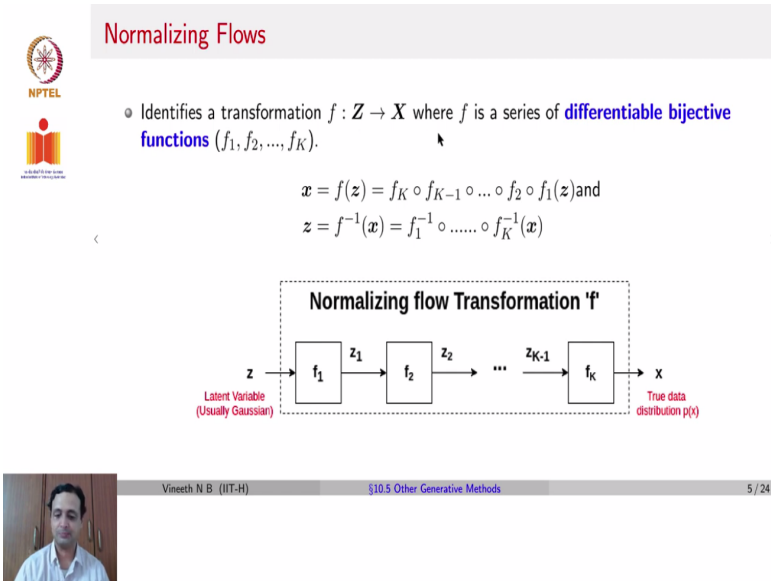


So what Flow-Based models trying to do here is if you could look at GAN as a discriminator and the generator, where the discriminator is the one that distinguishes between real and fake data. The generator takes you from a latent to a generated image x^P, which is then provided as input to the discriminator along with the real data x. Similarly, VAE is an encoder-decoder model, where the encoder captures the approximate posterior q parameterized by parameters ϕ, which are the weights of the encoder.

Similarly, the decoder captures $p_\theta(x \mid z)$, where θ are the parameters of the decoder. So, in this case, it is an approximate density estimation, as we just mentioned. In flow-based generative models, we do what should have been obvious but not very simple. Given x, find the function f to get us a latent representation, which if we invert, we get back x^P or the reconstruction.

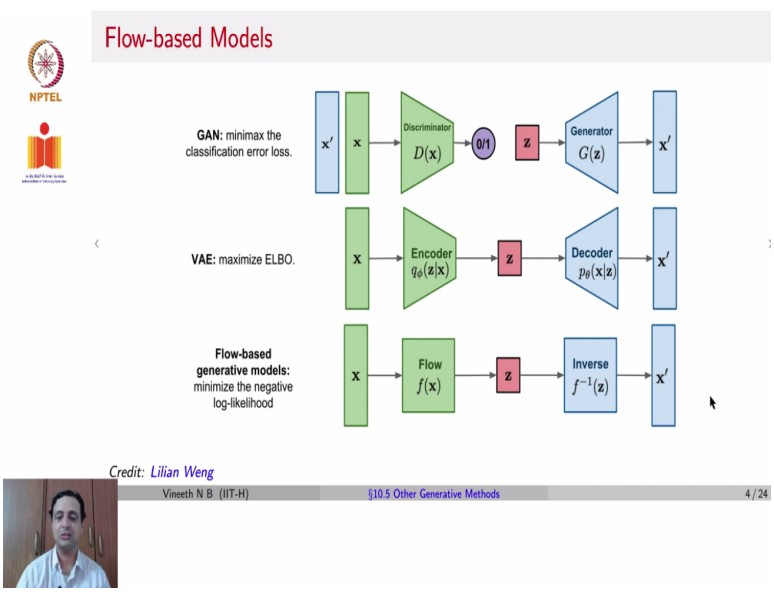
So, the challenge here is how do you find these functions f, which are exactly invertible to get back your original data? That is the challenge.

(Refer Slide Time: 05:27)



So, the first class of Normalizing Flows methods is to identify a transformation f, which takes you from Z to X.

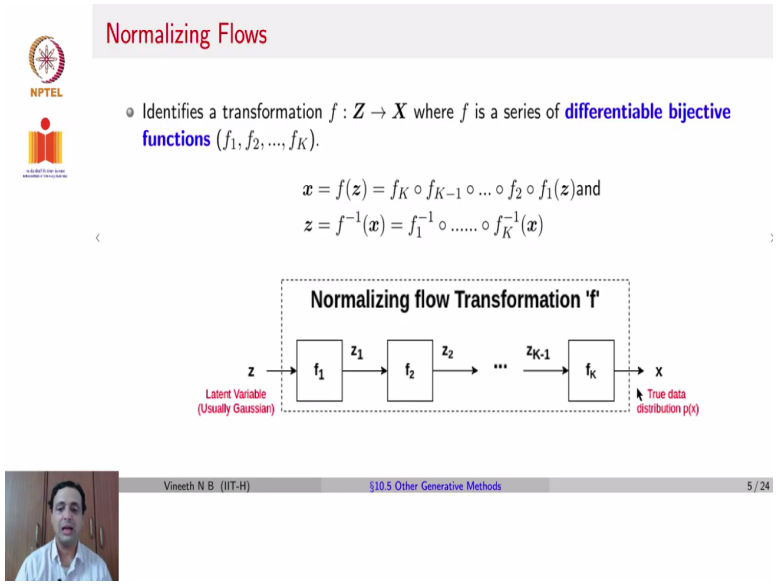
(Refer Slide Time: 05:43)



If you recall, Z is once again the latent very similar to VAE. The main difference between VAE and Flow-based models is in VAE, the encoder captures an approximate posterior with the different parameterization, and the decoder captures p_θ itself. So, we know that q_ϕ is an

approximation of the true posterior $p_\theta(z \mid x)$, whereas, in flow-based models, the functions are an exact inverse.

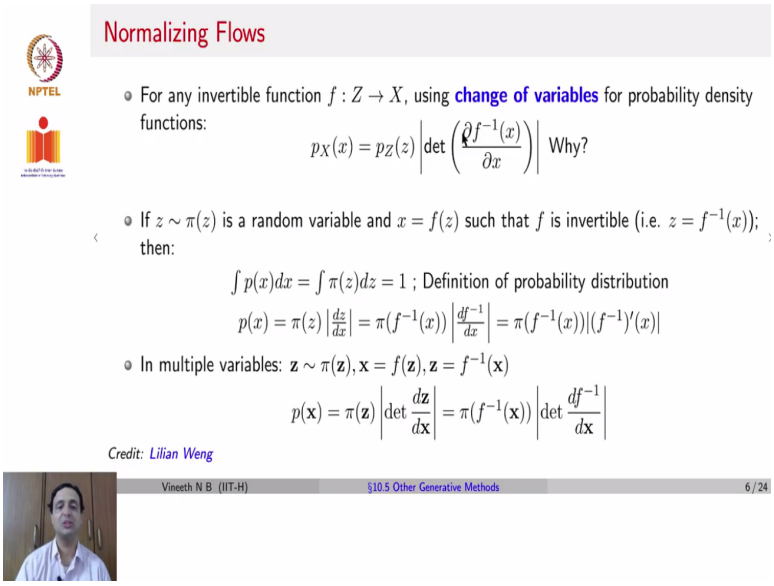
(Refer Slide Time: 06:16)



So here, we identify a transformation f that goes from Z to X. The transformation f is a series of differentiable bijective functions $(f_1, f_2, ... f_K)$ in such a way that x can be written as $f(z)$, which in turn can be written as f_K , composition, f_{K-1} , composition, so on and so forth until $f_1(z)$. Conversely, z can be written as $f^{-1}(x)$, which in turn can be written as f_1^{-1} composition, f_2^{-1}, so on and so forth, till the final composition, $f_K^{-1}(x)$.

Diagrammatically given a latent variable z, which could be a Gaussian, very similar to GANs, you pass this vector input vector sampled from z through a neural network. At this point, we will just call it a function, which outputs z_1, which goes through f_2, and so on, till f_K, which we finally expect to output the original data distribution that x comes from.

(Refer Slide Time: 07:43)

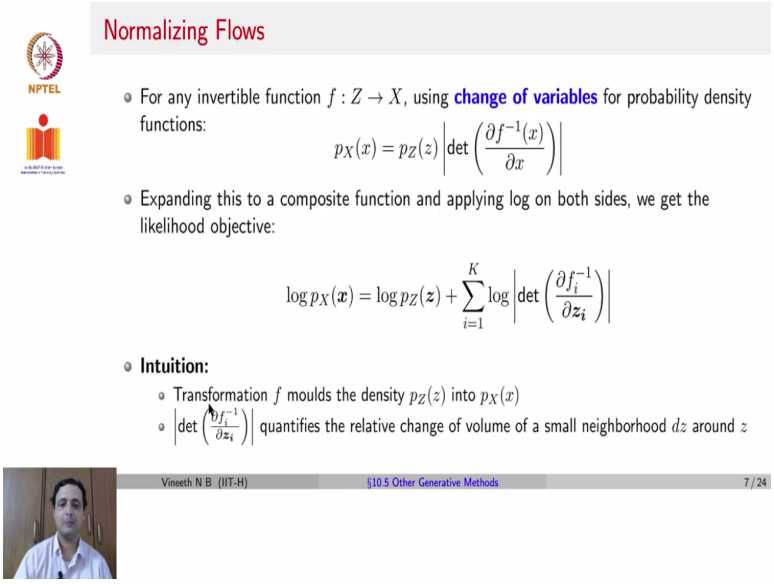


Let us see this in more detail before going to the implementation. For any invertible function f from Z to X, using change of variables of probability density functions, we can write $p_X(x)$ as $p_Z(z)$ into the determinant of $\partial f^{-1}(x)/\partial x$. Why? It is not difficult to show it; if z belongs to distribution, $\pi(z)$is a random variable, and x is equal to $f(z)$ such that f is invertible.

So, z would be given as $f^{-1}(x)$. From the definition of the probability distribution, we have integral $p(x)dx$is equal to 1. It would also be integral $\pi(z)dz$ is equal to 1. So from this equality, we can say that $p(x)$ can be given as $\pi(z)$ into the gradient of $dz\ /\ dx$, the absolute value. But we already know that z is $f^{-1}(x)$, so we will put that here. We would have $\pi\ (f^{-1}(x))$ into the gradient of $f^{-1}(x)/dx$, which we are writing as to $\pi(f^{-1}(x))$ into $(f^{-1})'(x\ )$ where prime here stands for the gradient.

In vector form, this would be $p(x)$ is equal to $\pi(z)$ into a determinant of $dz\ /\ dx$, or $\pi(f^{-1}(x))$ into determinant, $df^{-1}(x)/dx$, which is what we wrote here in the first place.

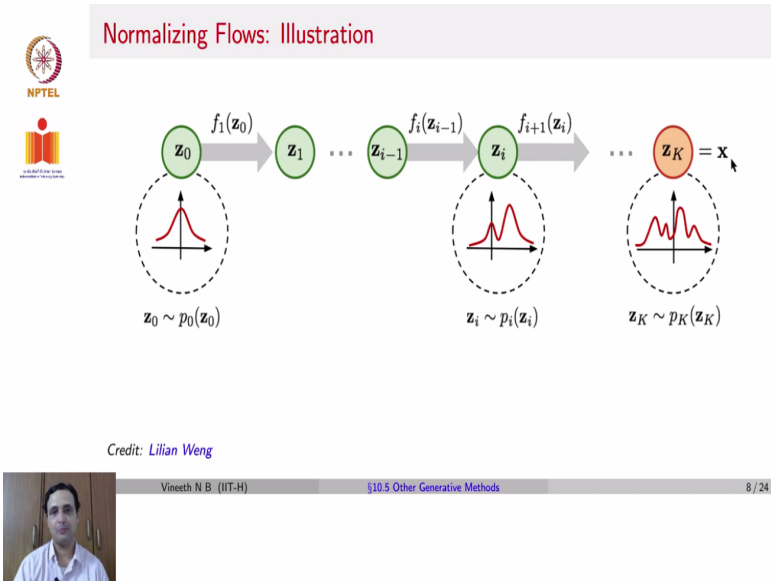
(Refer Slide Time: 09:42)



If we took the same expression here, and if we expanded it and applied log on both sides, we would then get $log\, p_X(x)$, which is the log-likelihood of the density that we are looking for $p_X(x)$ would now become the log-likelihood of z plus summation going from i equal to 1 to K, log of the determinant of $\partial f_i^{-1} / \partial z_i$. How did this come? This came by substituting f with a composition of functions, f_1 through f_K, the composition while taking log becomes a summation. That is how we got this expression here.

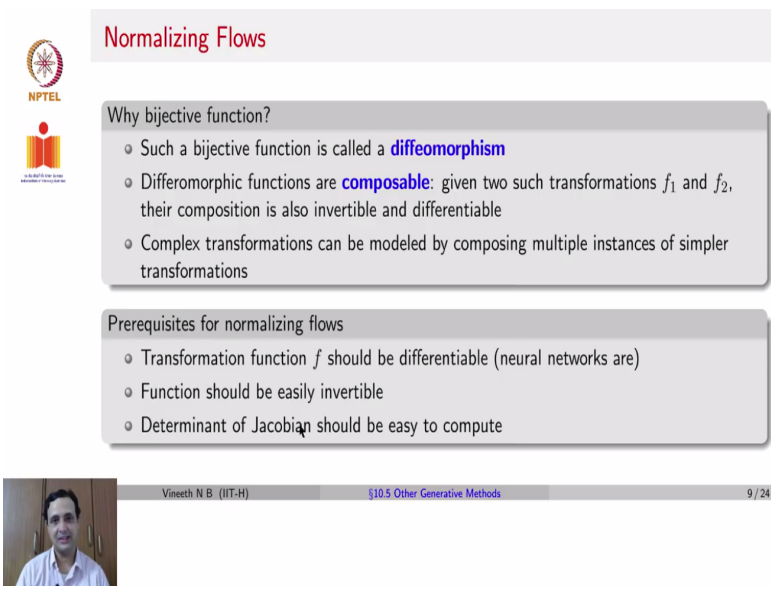
So the intuition of these two terms in estimating the log-likelihood of the probability density function is the first term here that can be looked at as the transformation moulds the density $p_Z(z)$ into $p_X(x)$, that is what it would like to do. The second term here quantifies the relative change of volume of a small neighbourhood dz around z. Remember that is what gradient measures. That is what determinant would also measure. You can look at it as capturing the volume of the space that you are trying to measure.

(Refer Slide Time: 11:18)



Here is an illustration to understand how normalizing flows work. Assuming that you sample from a Gaussian initially, which is z_o you apply a function f_1 get z_1, then you would apply another function, so on and so forth, till you get z_{i-1}, z_i. Finally, you keep applying many functions until you get z_K when the probability density function transforms this way, which is the density function of x that we are looking at.

(Refer Slide Time: 11:52)

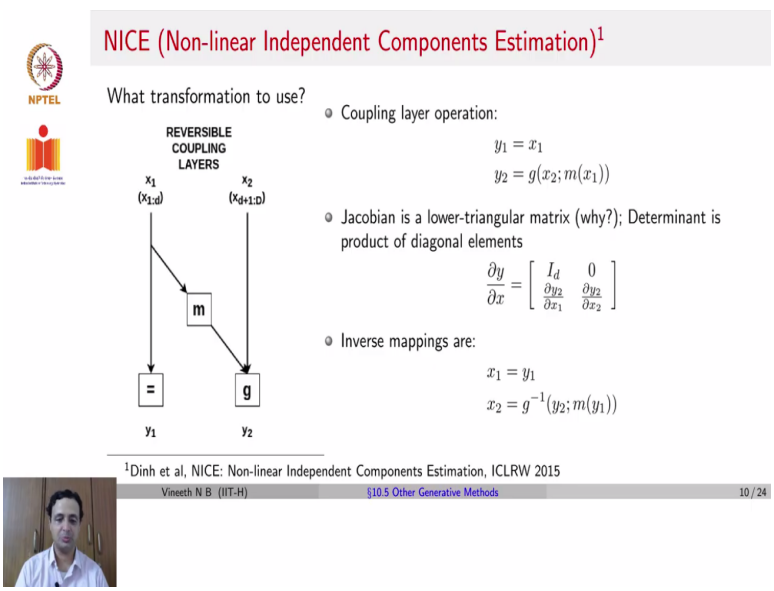


Why did we say we want a differentiable bijective function? Why should each of the f's be a differentiable bijective function? It should be fairly straightforward. We already saw how the inverse was being used. But let us define it more formally. Such a bijective function, the way we are using it, is called a Diffeomorphism. Diffeomorphic functions are composable, which means given two transformations f_1 and f_2, the composition is also invertible and differentiable, which is very important when working with neural networks. So any complex transformation from a Gaussian to a complex probability density function of the real-world data can be modelled by composing multiple instances of simple transformations.

What do we need for normalizing flows? We want the transformation function to be differentiable, so that should give us the answer. If each of those functions that take you from z to x is a layer of a neural network, or an LSTM, or a set of layers of a neural network, they are differentiable, and we have met the first prerequisite. The second is that the function must be easily invertible.

How do we do this? We will see in a moment, and the last is the determinant of the Jacobian should be easy to compute, why because that is 1 of the terms in your loss function, and you want that to be easy to compute so that you can train the entire network using gradient descent.

(Refer Slide Time: 13:46)



Let us see one of the earliest efforts in implementing Normalizing flows. This is known as NICE Non-Linear Independent Components Estimation. This work was published in 2015 by Lauren Dinh et al. The idea here is to introduce known as Reversible Coupling layers. Let us see what those transformations are. So the coupling layer operation used was y_1 was the same as x_1. So if x_1, x_2 are the different dimensions of the data x. y_1, y_2 so on and so forth, are the different dimensions that you're trying to generate, which you would like to match x in principle.

So y_1 is equal to x_1, y_2 is given by some function $g(x_2; m(x_1))$. So you can see here pictorially y_2 is given by function g, which is applied on x_2 and $m(x_1)$. Note that in this particular formulation, y_1 does not depend on x_2 , but y_2 depends on x_1. In this case, the Jacobean will be a lower triangular matrix. Why do you say so? Because y_1 does not depend on x_2. So, all these upper triangular elements of the Jacobean matrix would become 0's.

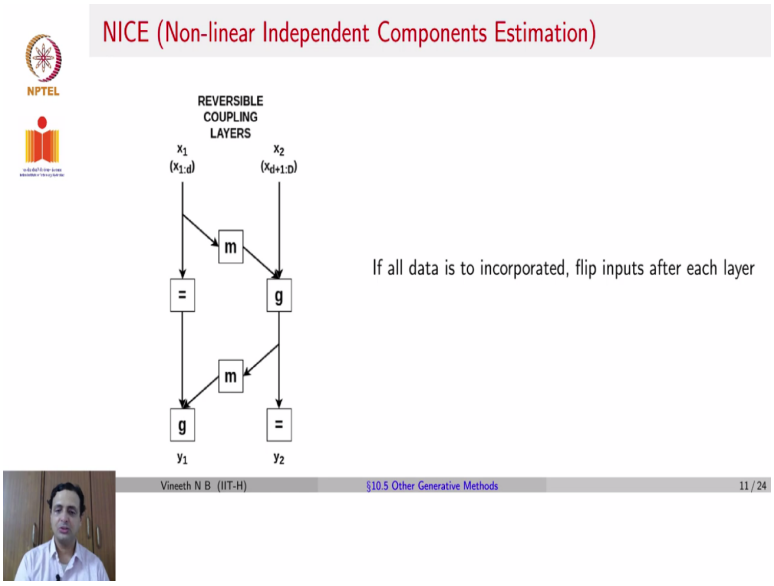
Let us take a moment to recall a Jacobean matrix. A Jacobean matrix is the matrix of all partial derivatives. So, if you had an output vector y, which contains y_1 to y_K, and if you had an input vector x, which was x_1, to x_d, all the pairwise gradients, $\partial y_1 / \partial x_1$, $\partial y_2 / \partial x_1$, $\partial y_3 / \partial x_1$, so on and so forth, till $\partial y_K / \partial x_1$.

And similarly, $\partial y_1 / \partial x_2$, $\partial y_1 / \partial x_3$ so on and so forth will form the rows and the columns of the Jacobean matrix. So, it is a matrix of first partial derivatives between a vector and a vector. So, in this case, you can see that all the upper triangular elements, because of the construction of the operation, y_2 would depend on x_1, but y_1 would not depend on x_2 x_3 or anything till x_d. Similarly, y_2 would depend on x_1 and x_2, but not on x_3, to x_d, which means all those upper triangle elements here would become 0.

You will be left with a lower triangular matrix, and the determinant of a lower triangular matrix is simply the product of the diagonal elements. So, the determinant becomes easy to compute in this construction. What would the inverse mappings be? The inverse mappings would be x_1

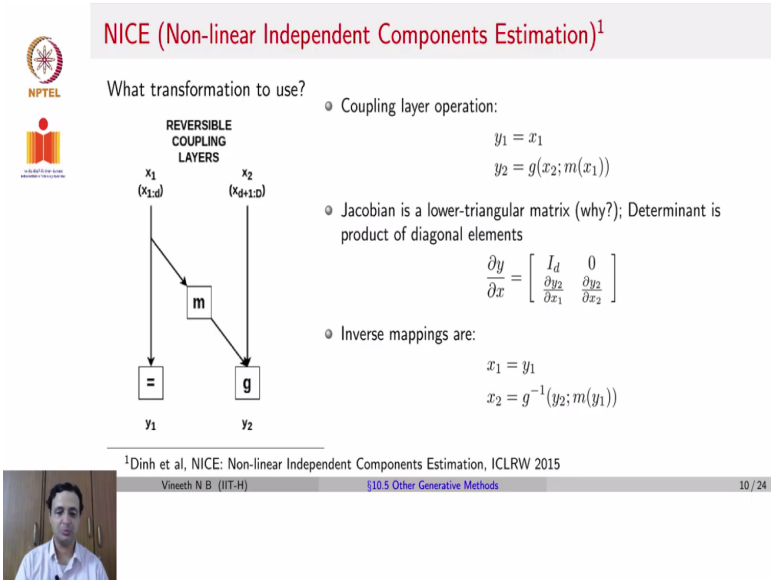
would be. Of course, x_2 would be $g^{-1}(y_2)$ and which would be the inverse operation in this particular case.

(Refer Slide Time: 17:27)



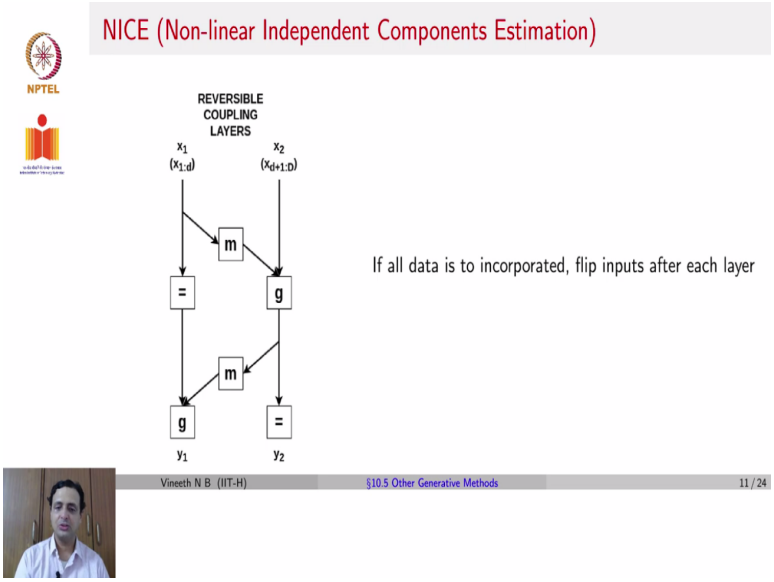
In case you would like all data to be considered, so, in the previous construction,

(Refer Slide Time: 17:33)



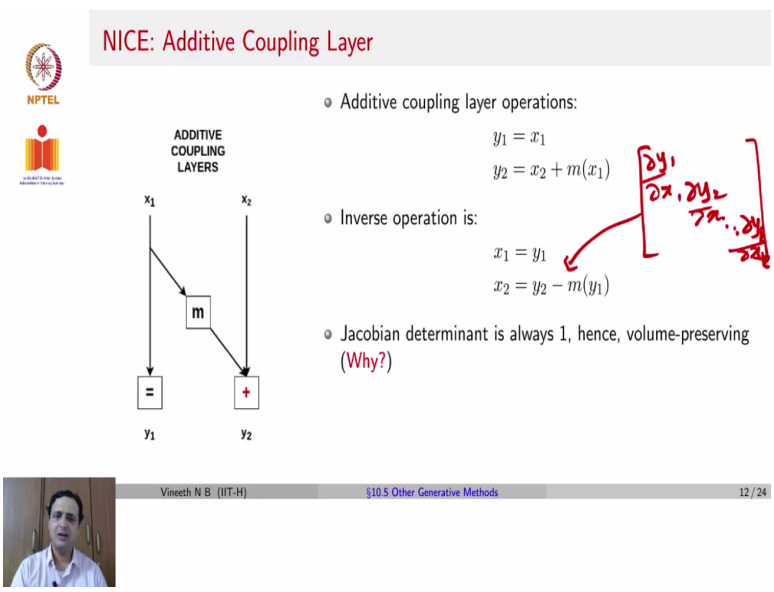
We considered that y_2 could depend on x_1, but y_1 cannot depend on x_2.

(Refer Slide Time: 17:42)



If we want all the output elements to depend on all the input elements, it can be done, but you may have to flip the inputs after each layer. So that way, in the next iteration, next function, remember it is a series of layers or composition of functions, you can have y_1 dependent on x_2, and you can continue this process to ensure that all the output variables depend on all the input variables.

(Refer Slide Time: 18:13)



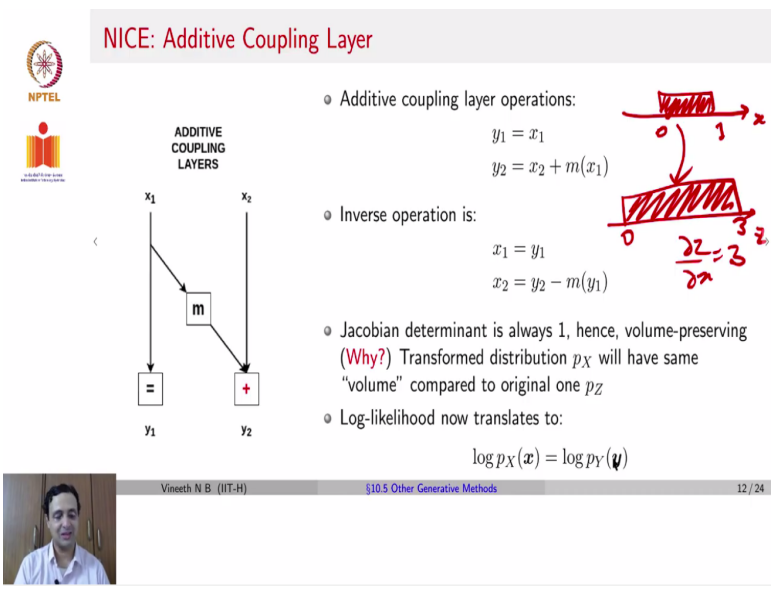
So, you can now write this as Additive Coupling Operations also. In additive coupling operations, you can say that y_1 is equal to x_1 and y_2 is equal to x_2 plus $m(x_1)$. So, in this

particular case, it is not a function. Still, you just have y_2 is equal to x_2 plus $m(x_1)$. The inverse operation here would be x_1 is equal to y_1 itself, and x_2 is equal to y_2 minus $m(y_1)$. In such construction in an additive coupling layer in the previous case, you had a g function here. Still, in the additive coupling layer, the Jacobian determinant will always be 1.

Why do you say so? You can look at the two equations and give the answer; because it is additive coupling, you would have $\partial y_1/\partial x_1$ will be 1, $\partial y_2/\partial x_2$ will also be one because the second term here is additive and does not depend on x_2. So, all those diagonal elements of Jacobian, what are the diagonal elements in your Jacobian matrix? You would have $\partial y_1/\partial x_1$, $\partial y_2/\partial x_2$, so on and so forth till $\partial y_K/\partial x_K$. For the moment, let us assume both are the same dimensions.

If you look at these equations, all these values will be 1 product of the diagonal elements will be 1. The Jacobian determinant will be one; this is called a volume-preserving operation.

(Refer Slide Time: 20:05)

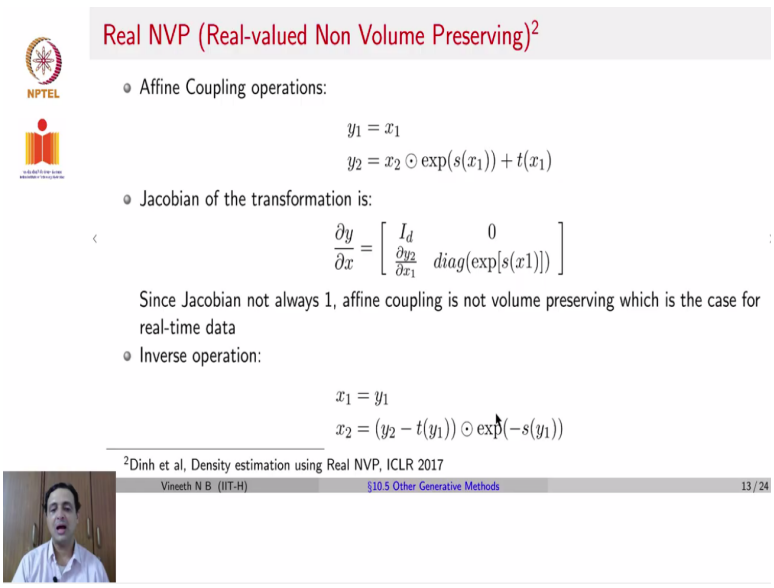


Remember, we said, the Jacobian determinant is an estimate of how much the volume changes. For example, if you have a random variable x, which has a certain density, say between 0 and 1, in this case, a uniform density.

If you add another random variable, say z, which goes from say 0 to 3. If its volume was like this, you can see that the gradient would give you an answer to be 3. So for every 1 unit that you move in x, you will move three units in z. So this tells you that the volume between x and the PDF of z triples. The determinant of the gradient, $\partial z / \partial x$, also gives you the answer 3, which intuitively tells you how much is the volume changing in the new PDF.

So, when the Jacobian determinant is always 1, you have an additive coupling layer. It is a volume-preserving operation. So, in this case, the log-likelihood becomes very simple. You simply have the determinant of the Jacobian term is always 1. So that term would disappear, and you would have the log-likelihood of p_X be the log-likelihood of p_Y itself.

(Refer Slide Time: 21:31)



An improvement over NICE was another normalizing flow method, a popular one known as Real NVP, Real-Valued Non-Volume Preserving Normalizing Flow. As the name states, it is non-volume preserving. So we have to do something beyond additive coupling. What do we do? We have an affine coupling operation. What does the affine coupling operation do? We say y_1 equals x_1, and y_2 equals x_2 into this Hadamard product, which says it is an element-wise product, so x_2 is a vector.

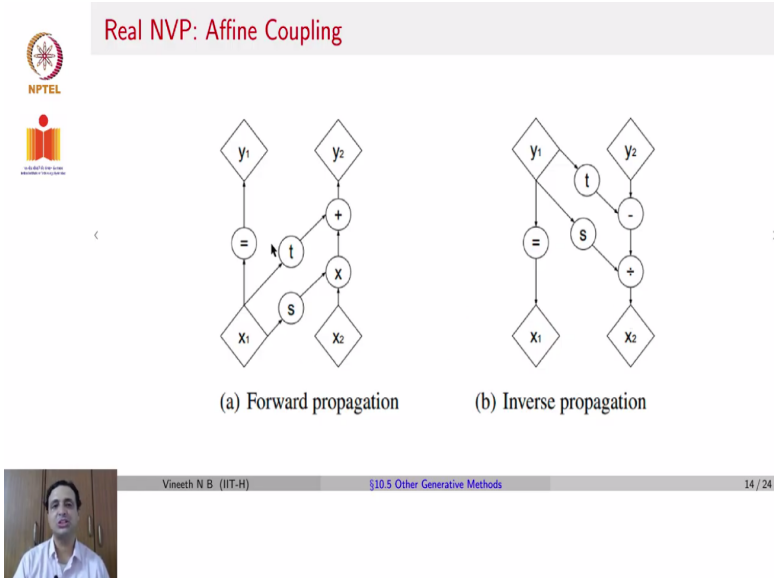
And the second term here is also a vector, and it is an element-wise product into an exponent of $s(x_1)$ plus $t(x_1)$. So there is a translation component, and there is a scale component, which together becomes an affine transformation. The Jacobian of such a transformation would be, it would still be a lower triangular matrix because y_1 does not depend on x_2, x_3, so on and so forth. Similarly, y_2 does not depend on x_3, x_4, and so forth.

So the Jacobian will have the upper triangular entries to be 0. You would then have in the lower triangle entries, $\partial y_2/\partial x_1$. The diagonal entry, in this case, would be $diag(exp[s(x_1)])$ because that is what the differentiation of $\partial y_2/\partial x_2$ will be. In this case, the Jacobian need not be one, and hence, the transformation may not be volume-preserving, which perhaps is more likely in real-world data.

For example, if we gave z, the random variable that we considered for normalizing flow to be a unit Gaussian, expecting that any real-world data distribution will also have the same volume as that unit Gaussian may not be a correct assumption. So in that sense, real NVP is more realistic. The inverse operation, what would it be here, x_1 would be y_1, equality, x_2 would be y_2 minus $t(y_1)$ into an exponent of minus $s(y_1)$.

Remember this exponential function. When you go to the other side, it becomes an inverse exponential function, which this term denotes.

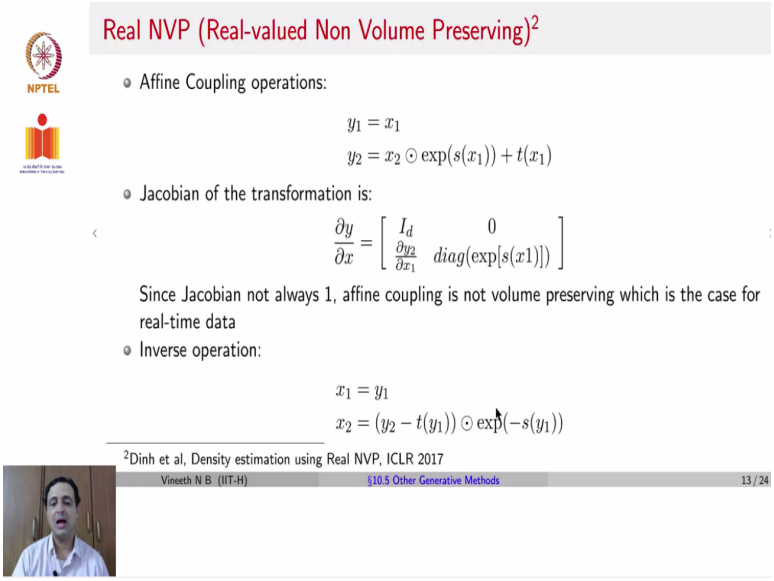
(Refer Slide Time: 24:11)



Figuratively speaking, this is how a real NVP affine coupling would look like. So you have your y_2 depends on x_1, x_2, and y_1 depends only on x_1. How does y_2 depend on x_1 and x_2? A translation and a scale component. The inverse looks something like in subfigure b. x_2 would depend on a scale and a translation component from $y_1$and the contribution from y_2. So how are all of these networks train?

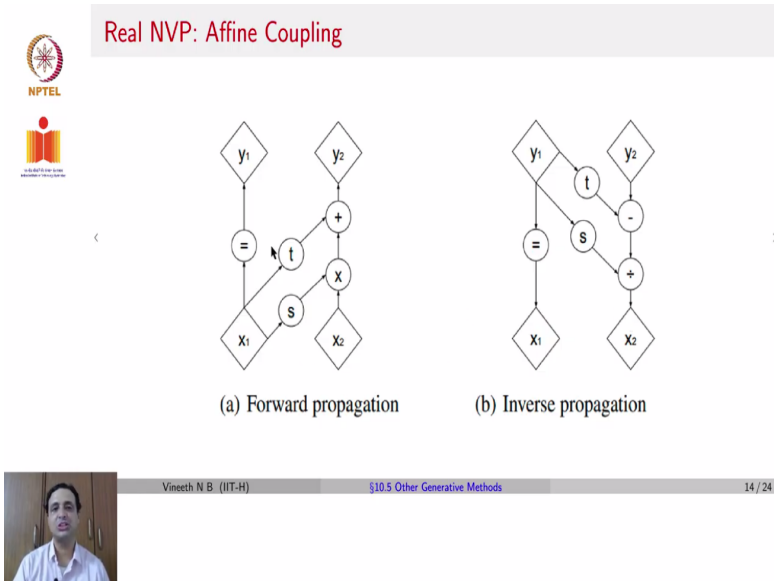
So the rest of it should work like any other neural network, so it is about maximizing the likelihood. You can use standard error functions like mean squared error, cross-entropy to learn the networks depending on what you are trying to reconstruct the output of each of these networks. And each layer corresponds to one of these functions that you are trying to model.

(Refer Slide Time: 25:15)



For example, y_1 is equal to x_1 , would be one of the layers of the neural network. The second layer would be this, where the scale and the translation are learned by the neural network, so on and so forth. That is how the network learns.

(Refer Slide Time: 25:29)



(Refer Slide Time: 25:29)