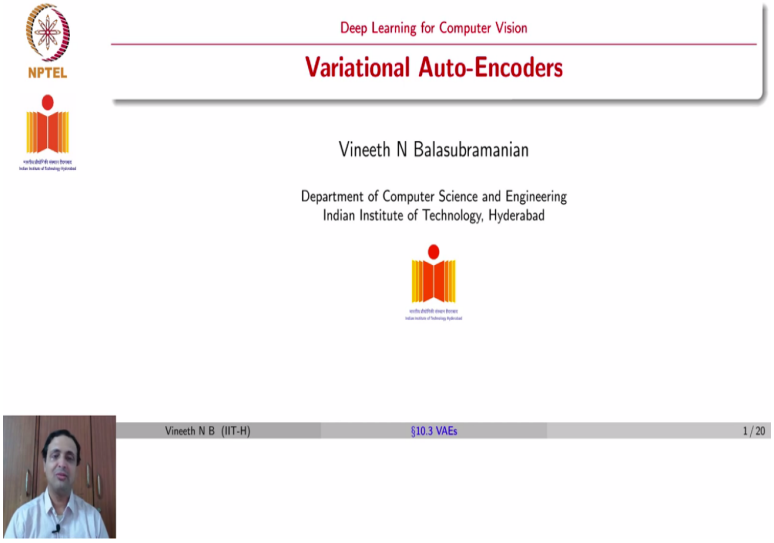
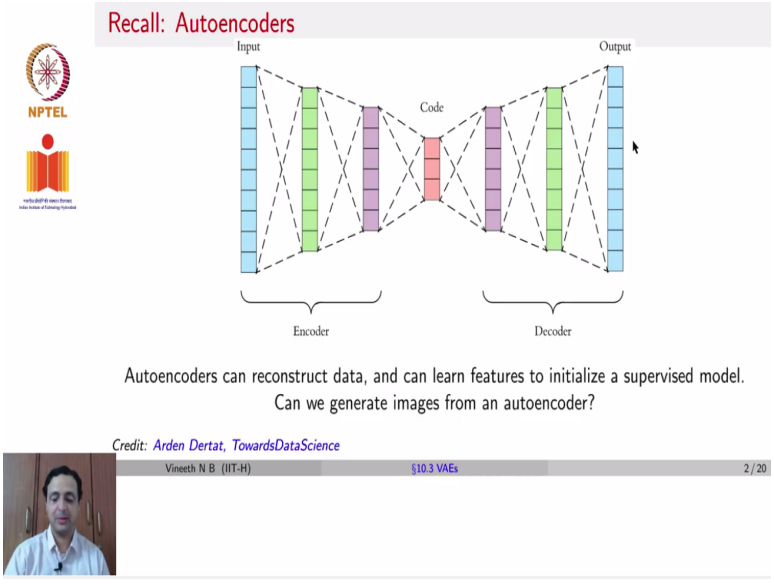
Deep Learning for Computer Vision
Professor Vineeth N Balasubramanian
Department of Computer Science and Engineering
Indian Institute of Technology, Hyderabad
Variational AutoEncoders

(Refer Slide Time: 00:14)



Let us move on from GANs to another kind of Deep Generative Models called Variational Auto Encoders, popularly known as VAEs, before we talk about Variational Auto Encoders. Let us briefly recall autoencoders, which we covered last week.

(Refer Slide Time: 00:34)

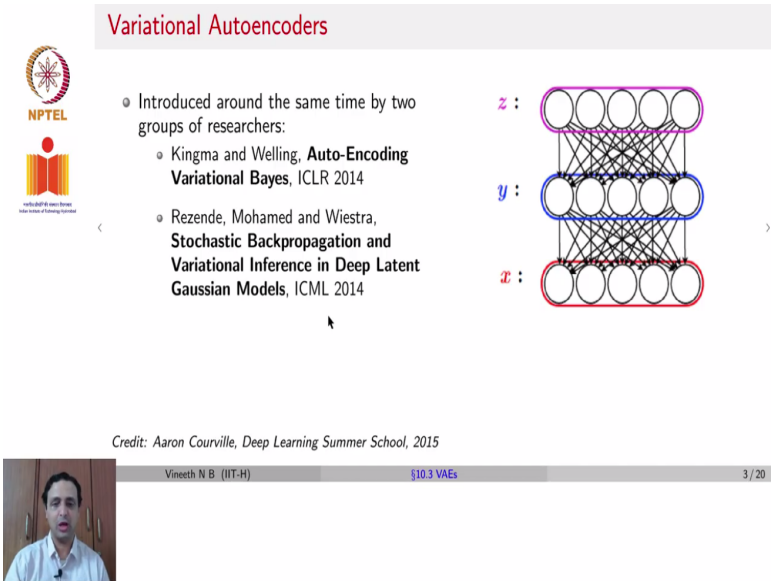


Autoencoders are neural networks used for unsupervised learning, where the network attempts to reconstruct the input itself as the output. Autoencoders have an architecture such as shown, where the input is compressed into a bottleneck layer, where a certain representation of the input is available, which is also called a Latent code, which is then used through a decoder to reconstruct the output, which we would ideally like to assemble the input itself.

The question we would like to ask in the context of this week's topic is, how can we use such an approach to generate images automatically? Remember, when we spoke about autoencoders, we generally discard the decoder once you train an autoencoder. Given an input, we use the encoder alone to get a low dimensional representation of your data. Let us try to see now if we can reverse this idea.

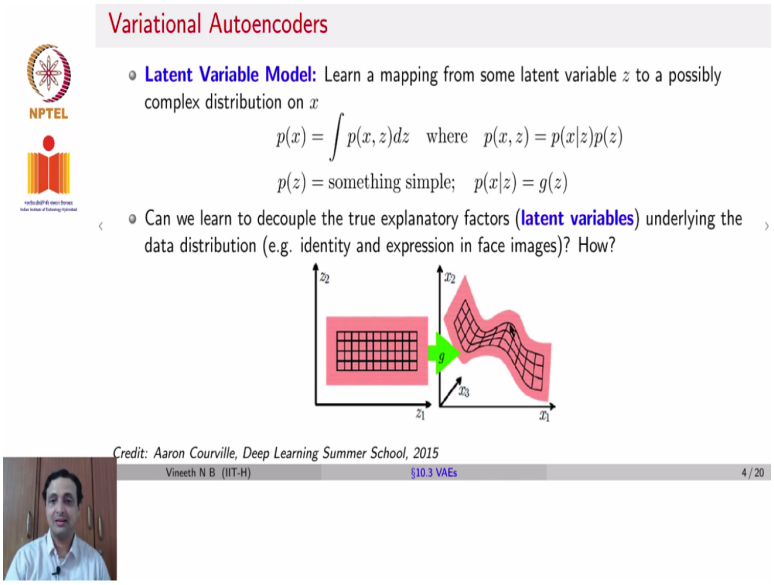
At the end of the training, Can we discard the encoder instead of the decoder? Using some vector as code, we can generate images close to the training data distribution. As you can see, the goal is close to what GANs were trying to achieve but achieved with a different perspective.

(Refer Slide Time: 02:27)



The idea of variational autoencoders was introduced simultaneously by two groups of researchers, Kingma and Welling, in a paper "*Auto Encoding Variational Bayes*" published in ICLR 2014. Also by Rezende, Mohamed, and Wiestra called "*Stochastic Backpropagation and Variational Inference in Deep Latent Gaussian models*", which was published in ICML 2014.

(Refer Slide Time: 02:58)



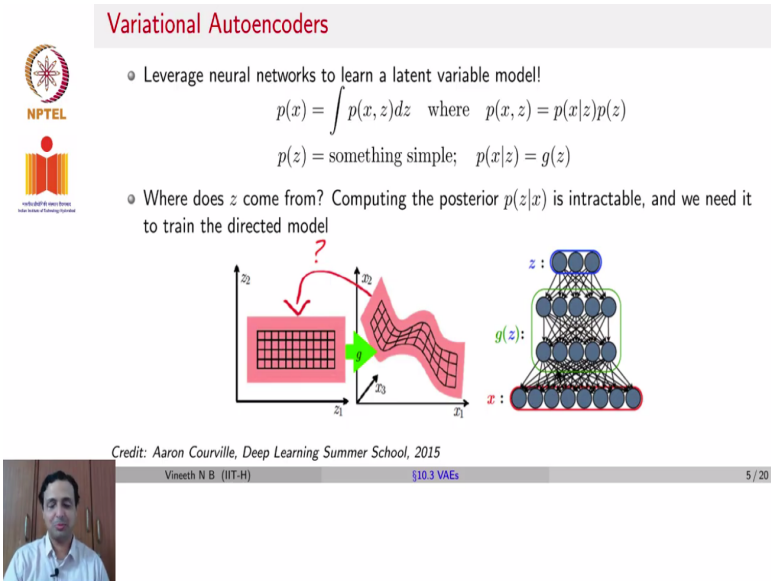
Let us try to understand variational autoencoders now. At the crux of VAEs is about learning what is known as a latent variable model. In a latent variable model, our objective is to learn a mapping from some latent random variable z to a possibly complex distribution on x, which we assume is the real-world training distribution that we would like to generate data similar to. Mathematically speaking, you would then write $p(x)$ is equal to the integral of the joint distribution $p(x, z)$ where $p(x, z)$ is equal to $p(x \mid z)$ into $p(z)$ We assume that the prior or the marginal distribution on z, the latent variable, is something simple.

How would you get $p(x \mid z)$? We assume there is some function of an input vector z. So, that would be the simple mathematical construction here. Given training data, assuming here it is unsupervised learning, so only x. The question that we are asking is can we learn to decouple the true explanatory factors or the latent variables underlying the data distribution, which is perhaps generating the data?

For example, in face images, you may want to assume a latent random variable called identity, a latent random variable called expression. So, if we say person A is happy, you should generate a facial image of person A in a happy mood. So, in this case, identity and expression would be latent variables. And those latent variables may assume different values, identity may assume values of different people like ABC, and expression may assume different values like happy, sad, angry, so on and so forth.

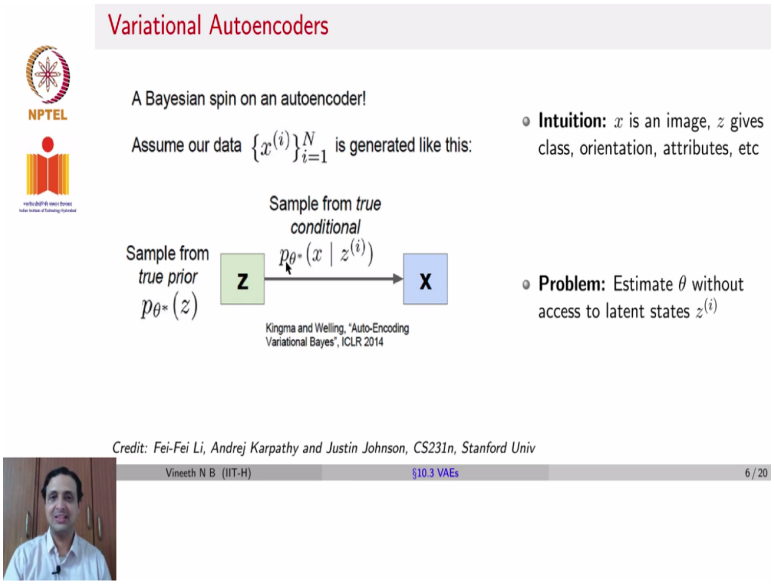
And the combination results in the generation of specific kinds of images. So, we are talking about a latent set of variables, in this case, represented as z_1, z_2, and a function g that takes us from this latent space to the original data space. How do we learn such a function g that takes us from z to x?

(Refer Slide Time: 05:45)



We will assume that we are going to use a neural network to learn such a function that takes us from z to x. So, this conditional distribution here. So, given a vector z, we ideally should use a neural network to give us a vector x. The question now remains, where does that come from? Computing the posterior $p(z \mid x)$ is intractable because we do not know what z looks like that is not available to us. Unfortunately, we need that to be able to train the G of z model. Without values of z, we cannot train the $g(z)$ model, also. So, how do we get around this problem?

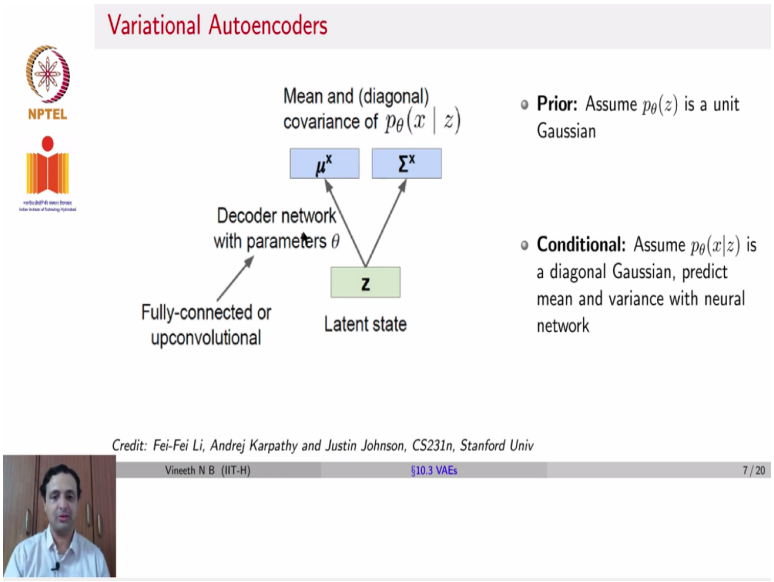
(Refer Slide Time: 06:34)



So, as you can see so far variational autoencoders offer a Bayesian spin on an autoencoder. So, if we assume that our data is generated using this generator process, where z is a set of latent random variables, which we assume belong to a certain prior distribution, we sample from a true conditional distribution, which results in x, which is the training data distribution that we see. The intuition here is, x could be an image, which is generated from some latent attributes, such as a class label, and orientation, attributes, so on and so forth.

So, we could say that the class label is a person's face, the orientation is in which direction they are looking, attributes could be the color of the face, the color of the eyes, the color of the hair, etc. And the problem now for us is, without knowing what these z's are, we need to estimate the parameters of this conditional distribution, $p(x \mid z)$. The θ parameters is something that we would like to estimate without having access to z. How do we do this is the core idea of VAEs.

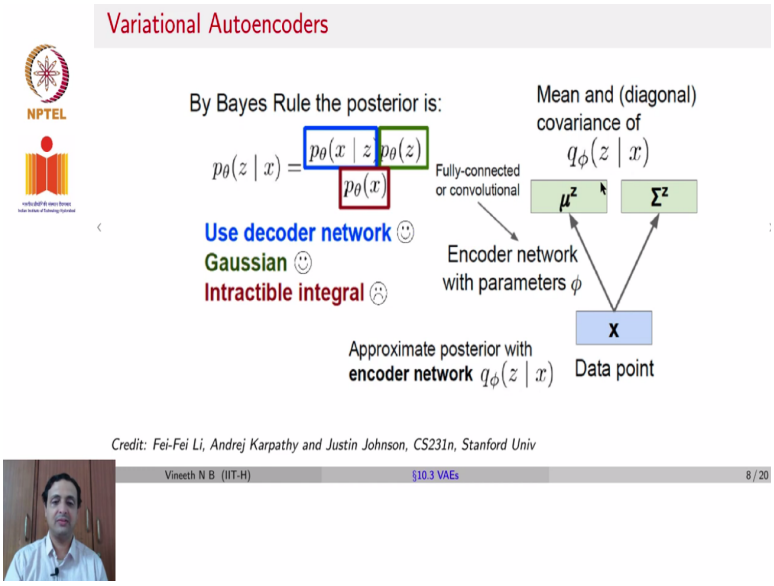
(Refer Slide Time: 08:04)



To start this process, let us assume now that there is a prior distribution on the random variable z, which for convenience, we are going to assume is a unit Gaussian. And the moment you have z, you can now construct a neural network, which we call the decoder network with certain parameters theta, which are the weights of those layers in the decoder network.

And the decoder outputs a set of means and variances, which can then be used to sample and get data similar to x. So, the conditional distribution, $p_\theta(x \mid z)$ is assumed to be a diagonal Gaussian for simplicity. One could assume a multivariate Gaussian with a certain covariance matrix, but the procedure gets a bit complex. We will keep it simple as a diagonal Gaussian, whose means and variances are predicted by the decoder network. Each dimension of that multivariate Gaussian, the mean and the variance is predicted.

(Refer Slide Time: 09:19)

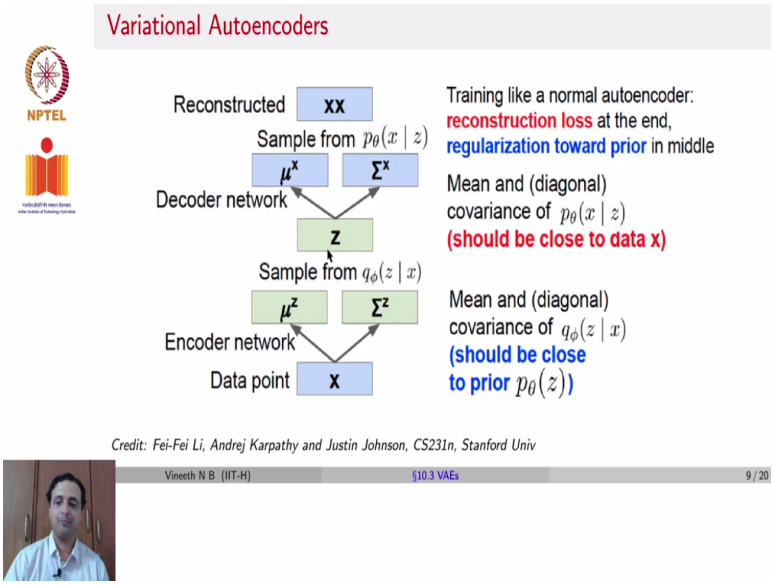


From Bayes rule, we know that $p_\theta(z \mid x)$, which is our posterior, is given by $p_\theta(x \mid z)$ into $p_\theta(z)$ by $p_\theta(x)$. Now, if we observe the terms on the right-hand side, $p_\theta(x \mid z)$, we just talked about how you can obtain it. Assume that your z's are a unit Gaussian and learn a decoder network to give you the output x. So, that gives you $p_\theta(x \mid z)$. What about $p_\theta(z)$?

We just said we are going to assume that to be a Gaussian. So, that is an assumption. What about the denominator $p_\theta(x)$? Unfortunately, that is an intractable integral to find out what distribution and parameterization of that distribution that x comes from. It is something we do not know at this time. So, what do we do to get around this? We will now assume that there is some other distribution $q_\phi(z \mid x)$.

We will now call as an Approximate posterior. So, this is obtained using an encoder network. What does the encoder network do? It takes in your training data distribution, so that is where your neural network takes in all the inputs from your training data. It has a set of layers parameterized by some weights, which we denote as to ϕ. It outputs a set of means and variances of a multivariate diagonal Gaussian distribution. That is given by the approximate posterior $q_\phi(z \mid x)$.

(Refer Slide Time: 11:21)



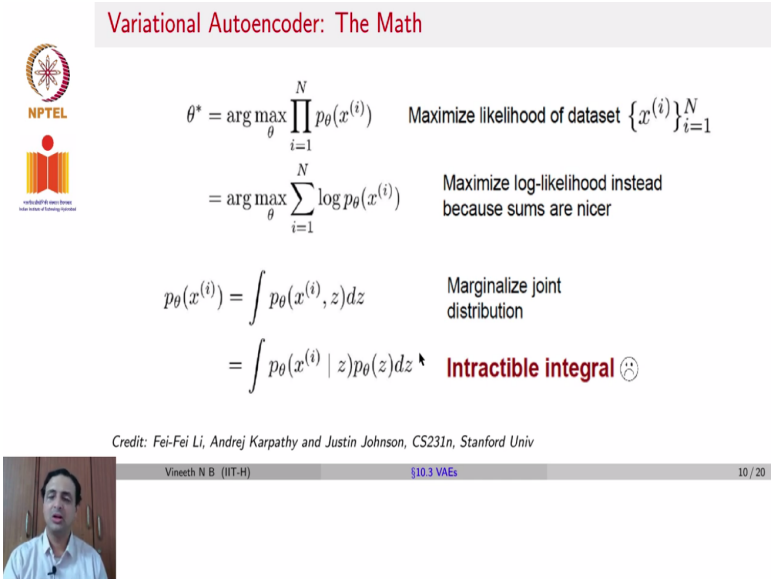
Putting all the pieces together, we have an encoder at the bottom and the decoder on top. The encoder takes x and gives out a set of means and variances for this approximate posterior distribution $q_\phi(z \mid x)$. Once you have the distribution, you can sample a z from that distribution and that sample z is given as input to your decoder network. The decoder network outputs a set of means and variances corresponding to $p_\theta(x \mid z)$.

Given these means and variances, you can sample from $p_\theta(x \mid z)$ to get your output images or reconstructions. We assume that both q_ϕ and p_θ are multivariate Gaussians, with a certain mean and a diagonal covariance. Each dimension is independent. Now, how do we train such a network? Let us try to understand that. We would ideally like to train the decoder network, just like the normal autoencoder.

So, because it will give us an output x, we would ideally like to use something like a reconstruction loss, something like a Mean Squared Loss to ensure that this reconstruction here is as close as possible to x itself. That should be one term of our loss function. What else do we need to do? We also want to ensure that this approximate posterior $q_\phi(z \mid x)$ should be close to the prior of z that we assumed.

We assumed for the decoder network that z comes from a unit Gaussian prior. So, we would ideally like $q_\phi(z \mid x)$ to be close to the distribution, $p_\theta(z)$, which, in our case, we assume to be a Gaussian with mean zero and identity matrix as it is a covariance matrix.

(Refer Slide Time: 13:46)

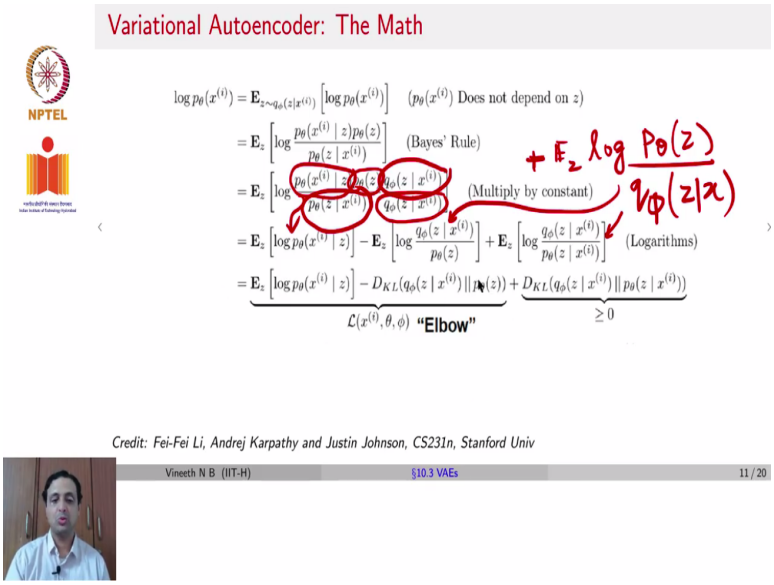


Let us try to put these together and get the formal loss function to train such a variational autoencoder. Remember, the goal is to do a maximum likelihood estimation. We have a training dataset given by x_i's. We ideally like to learn the parameters θ, which are the parameters of the decoder, in such a way that you maximize the likelihood of that distribution, generating your training data points.

That is your maximum likelihood estimation for your dataset. We will convert that to a maximum log-likelihood maximum estimation where the product gets converted to sum. Then you have $log\, p_\theta$, which makes things a bit simpler mathematically. Now, $p_\theta(x)$, which is this internal term here, can be given by integral $p_\theta(x,\, z)\, dz$ by definition, and this can be expanded as $p_\theta(x \mid z)\ *\ p_\theta(z)\, dz$.

Unfortunately, we are once again left with an intractable integral in this particular case. So, let us try to see how to solve this maximum likelihood estimation problem. We are going to solve it with a twist.

(Refer Slide Time: 15:18)



So, let us rewrite the maximum log-likelihood estimation problem. We have the log-likelihood $log\ p_\theta$, which can be given by, expectation over z vectors sampled from the approximate posterior of $log\ p_\theta(x)$. So, if we sample z vectors from your approximate posterior and give them as input to the decoder, the likelihood of generating samples similar to the training distribution must be high. That is what the first sentence means.

Now, $log\ p_\theta(x)$ can be written using Bayes theorem, as $p_\theta(x \mid z)$ * $p_\theta(z)$ by $p_\theta(z \mid x)$. That is just the Bayes theorem written differently. To this particular expression, we are going to multiply and divide the approximate posterior. It is just a simple multiplication and division by constant.

Once we have this expression equivalent to the maximum likelihood estimation setting for this problem, we can group terms to write this differently. The first term, $p_\theta(x \mid z)$, stays the same way, and we write that as the first term here, expectation over z, $log\ p_\theta(x \mid z)$. You have another term here, $q_\phi(z \mid x)$ and $p_\theta(z)$. Ideally, you would be able to write that as plus expectation over z, $log\ (\ p_\theta(z)\ /\ q_\phi(z \mid x)$.

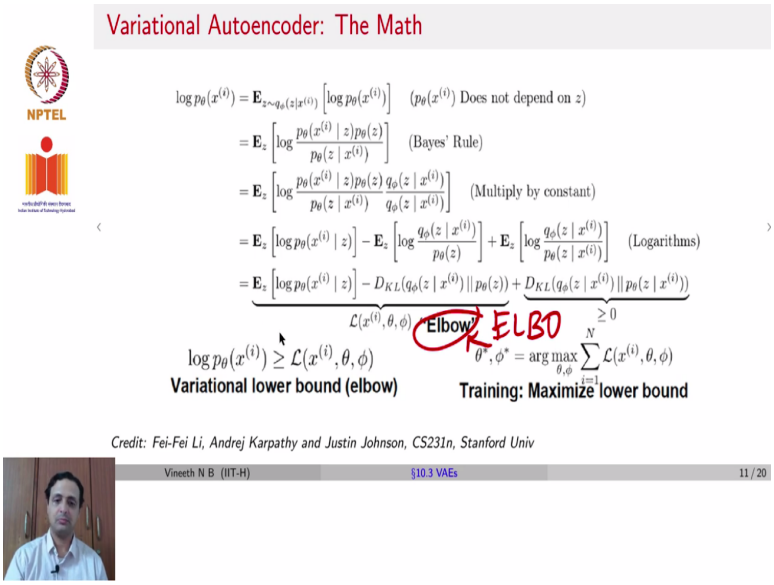
Now, instead of adding it, we are going to subtract it and write it this way. So, we have a minus expectation over z, $log\ (\ q_\phi(z \mid x)\ /\ p_\theta(z)$. It is just the reciprocal, with the sign change in the

beginning. The remaining terms are these two terms here, $p_\theta(z \mid x)$ in the denominator and q_ϕ in the numerator, which is what the third term gives us. So, it is just grouping terms and writing them differently.

Now, if you observe the second and third terms here, you will notice that the second term is the KL divergence between $q_\phi(z \mid x)$, the approximate posterior and $p_\theta(z)$, the prior on z. The third term is the KL divergence between the true posterior, $p_\theta(z \mid x)$ and the approximate posterior, $q_\phi(z \mid x)$. We will write the second term as a certain KL divergence and the third term as a certain KL divergence.

Now, if you observe here, KL divergence is a non-negative quantity. So, the third quantity will be greater than or equal to 0; because this is a log-likelihood estimation problem, we would ideally like to maximize this quantity. So, to maximize this quantity comprised of these three terms, where the third term is greater than or equal to 0. It can be achieved by maximizing the first two terms by itself.

(Refer Slide Time: 19:18)

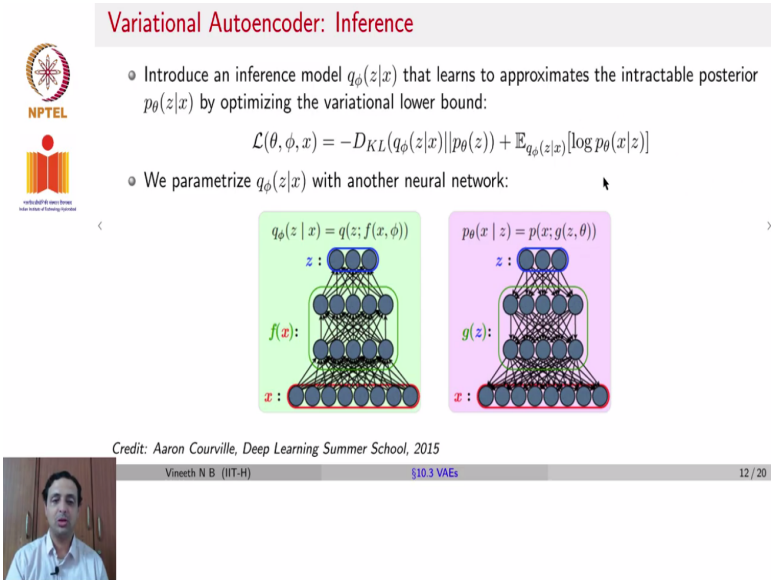


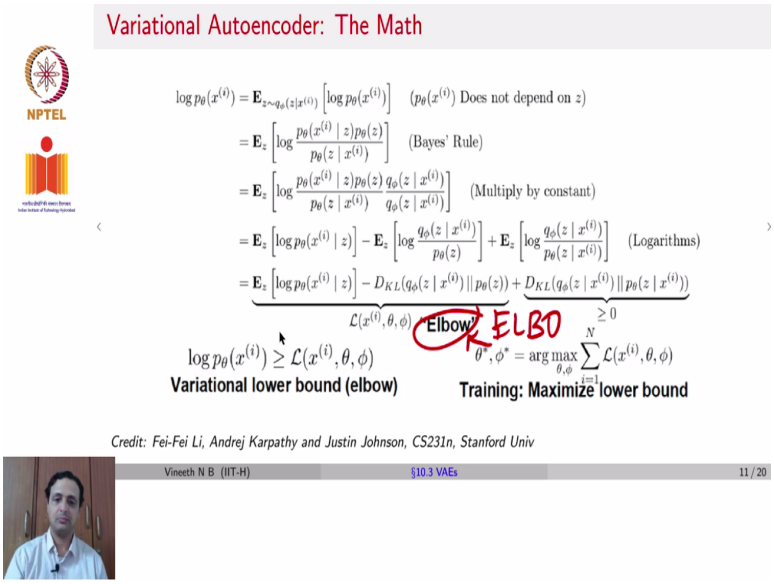
So, we are saying now that we have a lower bound, which is given by the first two terms. Remember, the third term is non-negative and is added to that. So, this entire log-likelihood will be lower bounded by these first two terms. So, if we would like to maximize the log-likelihood,

we can maximize this lower bound. This lower bound is also known as Evidence Lower bound or Elbow. It is read out as Elbow, and that is why it is written like this, but it is ELBO. Evidence Lower BOund is also known as the Variational lower bound.

And this entire procedure of optimization is known as a method called Variational Inference, where we introduce a variational distribution. In this case, the variational distribution for us is the approximate posterior q_ϕ and using that, we try to learn the parametrizations of the distribution. So, our goal is to find θ^* and ϕ^*, which maximize this evidence lower bound. That is what we would like to do.

(Refer Slide Time: 20:40)



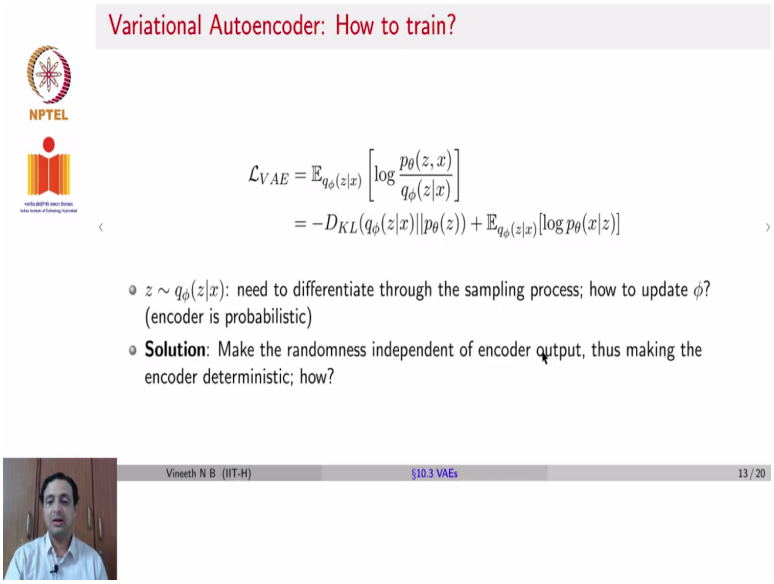


So, let us put this together. We now are saying that we have introduced an inference model $q_\phi(z \mid x)$, the approximate posterior that tries to approximate the original posterior by optimizing the variational lower bound that we saw.

Remember, these two terms here are exactly what we had as the two terms here: expectation over z, $log\, p_\theta(x \mid z)$ minus the KL divergence between approximate posterior and prior on z. We parameterize $q_\phi(z \mid x)$ using our encoder network. So, we now have an encoder and a decoder network.

Let us see them individually before we put them together in this context. Given training data x, provided as input to an encoder, the encoder outputs means and variances of an approximate posterior distribution. We would like to have it as close to the prior on z. Then we sample from z and use the decoder to generate x, which we would like to have as close to the training data distribution.

(Refer Slide Time: 22:06)

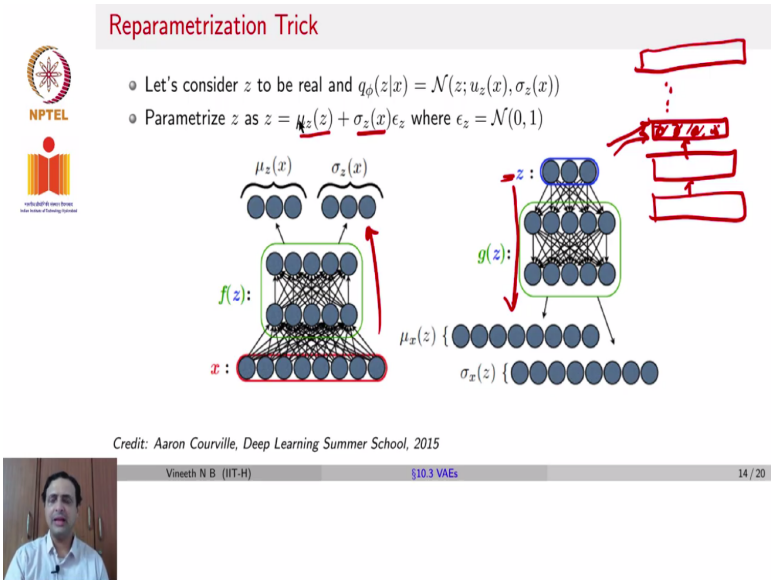


So, how are we going to train this? If you look at the loss by itself, the overall loss. In this case, we will maximize, so not perhaps a loss, just the objective function. But there is one problem with this training procedure. If you noticed, what we said was, we give the input distribution to the encoder, the encoder gives out a parameterization of the approximate posterior. We sample from a prior, which is then given to the decoder, and then you get your final output.

This means we now have an encoder-decoder model, an autoencoder. We will sample a vector rather than forward propagate a vector in the middle step or the bottleneck layer step. Why is this a problem? That sample may not be deterministic, and we would not know how to pass the gradients through that bottleneck layer to the encoder. That is a challenge. What do we do?

We ideally want to ensure that the sample that we get becomes independent of the encoder output. Suppose it becomes independent of the encoder output. In that case, you can simply backpropagate your gradients through your decoder and encoder and assume that the randomness is coming from some external quantity. How do we separate the randomness from the encoder output?

(Refer Slide Time: 23:49)



This is done using a trick known as the Re-parameterization trick. This was one of the main contributions of the Variational Auto Encoder method, a simple trick that solves the problem of back-propagating through this network. So, remember that we will sample z from, say, some approximate posterior distribution, which was given by a Gaussian distribution with mean at a certain $\mu_z(x)$ with variances, so standard deviations are given by $\sigma_z(x)$.

Now, a sample that is drawn from such a distribution can be rewritten slightly differently. We can parameterize that sample z. You randomly sample a vector from a unit standard normal distribution multiplied by the standard deviation learned by the encoder network and add the mean learned by the encoder network, and now this becomes a sample. How did this get around the real problem?

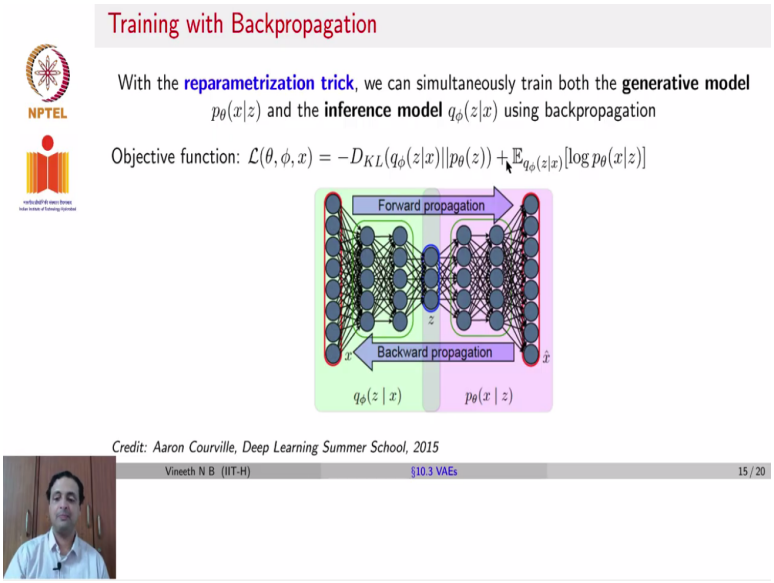
Remember, while in the earlier steps so far, you could now consider this something like this. So, you have a set of layers in a neural network. In the earlier version of the variational autoencoder, one of the layers became stochastic, so you had to sample a vector. It was not forward propagated through a neural network, as we have seen so far. So, because that was a sample, we would not have known how to backpropagate through that layer to the earlier layers, in our case, the encoder.

So, the parameterization trick suggests that let us fix that layer as the previous layer's output. Instead, it brings in the stochasticity from a different variable from the outside and not that layers

output itself. If you see here, the new sample is given by $\mu(z)$ and $\sigma(z)$, which are outputs of the encoder into ε, which you could almost consider now like a bias vector, which is coming from noise.

It does not affect backpropagation into earlier layers. It is an external input given in that particular layer. And now, you can backpropagate through the entire network using gradient descent, the way we have seen it before. So, you now have given x in the encoder. You generate mu's and sigma's, then you sample z using the Re-parameterization trick, go through the decoder and generate mu's and sigma's at the output from which you can sample and get any generations.

(Refer Slide Time: 26:58)

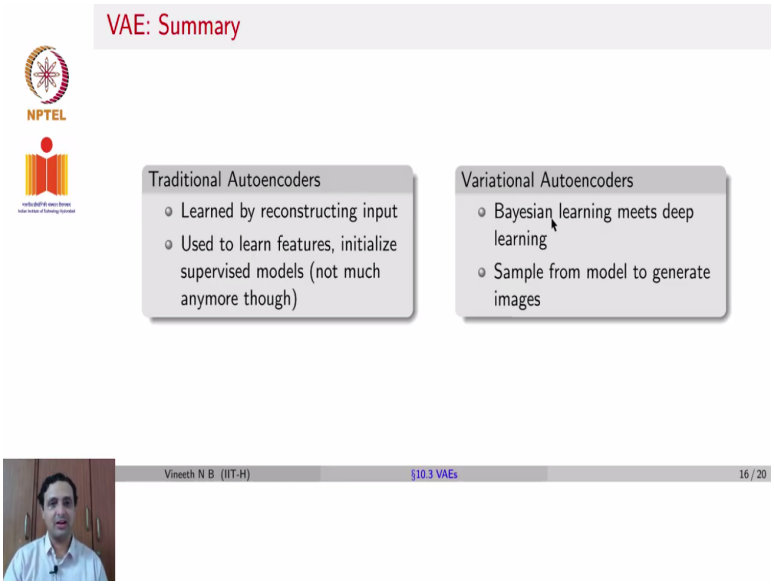


So, with the Re-parameterization trick, the entire variational autoencoder becomes back propagable. You can use gradient descent and backpropagation to update the weights in the VAE and train the network. So, as before, the objective function here is given by minus KL divergence of q_ϕ and p_θ plus the expectation of $log\, p_\theta(x \mid z)$. Now, $log\, p_\theta(x \mid z)$ turns out to be a simple reconstruction loss. We would like this to be close to your training distribution.

So, you can use even a Mean Square Error to maximize this likelihood in the second term. What about the first term? How can we make that differentiable? How can we get the gradient for it to you for backpropagation? The key here is that we have assumed q_ϕ to be a Gaussian with certain means and variances, which the encoder provides; we also assumed p_θ to be a unit Gaussian.

And the KL divergence between Gaussians is a well-defined formula that is differentiable. So, that helps us ensure that we can compute the gradients and backpropagate and train this network end to end.

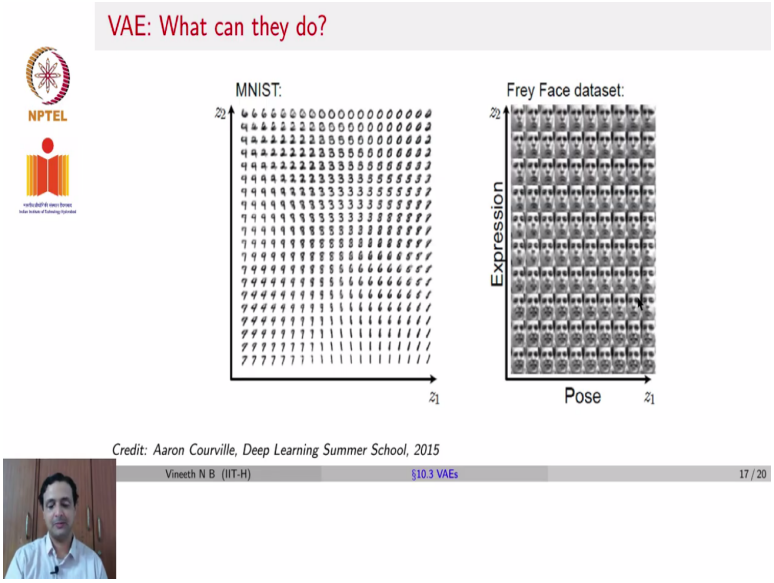
(Refer Slide Time: 28:28)



So, that completes our discussion of how VAEs are learned. To summarize and differentiate from autoencoders. Traditional autoencoders are learned by reconstructing input, and they are often used to learn features, the low dimensional representations, which could then be used to train another neural network or an SVM or to train any other supervised models. On the other hand, variational autoencoders conceptually are a marriage between Bayesian learning and Deep learning.

And after training a variational autoencoder, you could discard the encoder, just sample from your prior $p_\theta(z)$, a unit Gaussian. And now your decoder has learned to generate images, taking a unit Gaussian as input to generate images that look close to your input data distribution. That is what allows VAE to become a generative model.

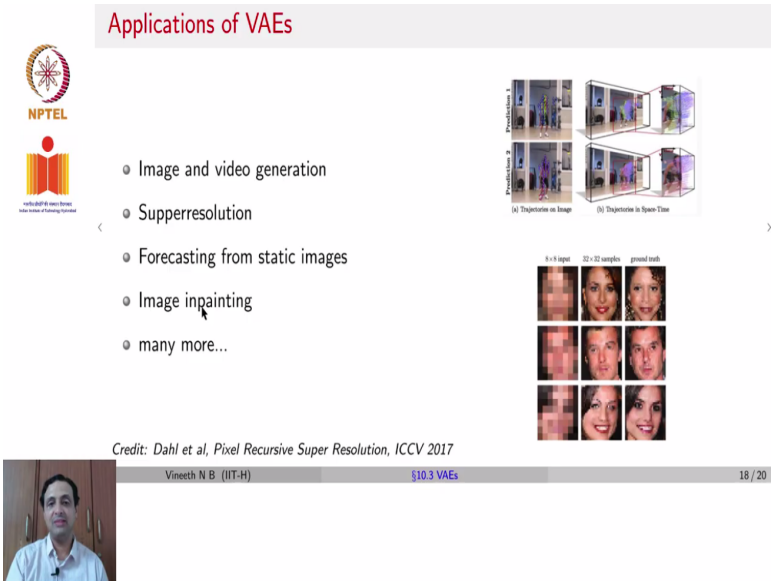
(Refer Slide Time: 29:39)



What can VAE's do? Here are a couple of examples where a VAE was trained on the MNIST data set, and the Frey Face dataset. Here, you can see the MNIST dataset when small values modify the latent variables that are the z's in your VAE. Remember, we will be sampling some vectors. You can see that the generated images, that is, the output of the decoder, gradually varies as the z value is varied.

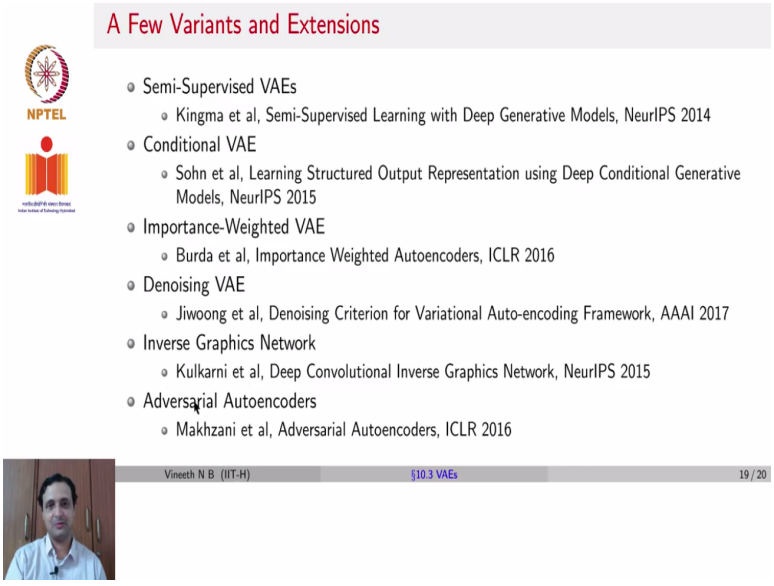
That shows us that the variational autoencoder has indeed learned latent variable values are capable of generating data smoothly on the manifold on which it lies. Similarly, just like how we saw for GANs on the Frey Face dataset, you can learn latent variables such as expression and poses. As you change the values of those corresponding latent random variables, you end up getting generations of face images with different poses and different expressions.

(Refer Slide Time: 30:54)



What else can be VAEs be used for? They have plenty of applications, and we will see some of these in the next week in detail. But here are a few examples, image and video generation, super-resolution, forecasting from static images, image inpainting many more, and we will see some of them next week.

(Refer Slide Time: 31:19)



VAEs have also been extended in numerous ways over the last few years. While we may not have the scope to cover all of these in this course, here are a few pointers if you would like to

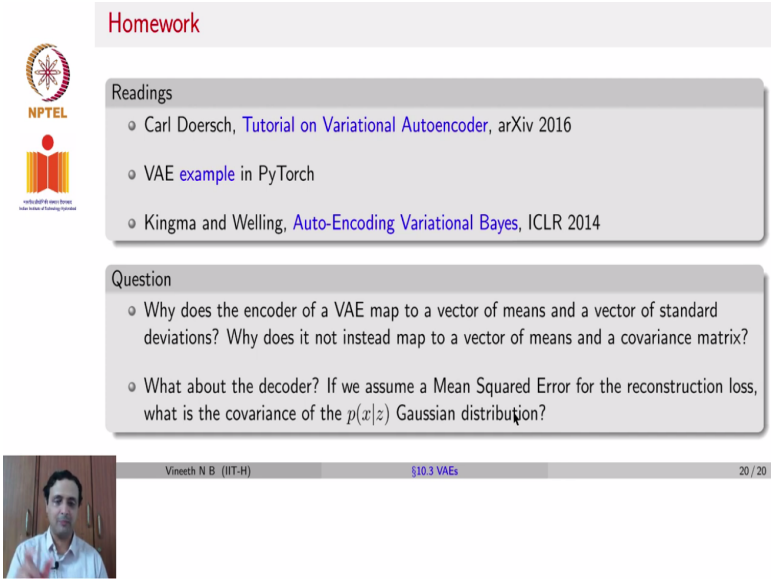
know more about the original author's extended VAEs to get a semi-supervised variant to use labelled and unlabeled data to learn the VAE.

Another popular extension, rather a powerful extension was conditional VAE, where you could generate given a particular class label for instance. So, in the example that we have seen so far, in the construction that we have seen so far, we considered it to be an unsupervised learning problem. And just assumed only data was given and we were generating more data, you could modify that to assume that label data was given to us.

And at inference, if you give a certain class label, the VAE would generate data points belonging to that class label. Those issues are discussed in conditional VAE. Another variant known as Importance-Weighted VAE weights the latent random variables differently to get more powerful generations. Then there is a Denoising VAE, which you could consider as an extension of the Denoising Auto Encoder to a VAE setting.

Then VAEs have also been used for graphics. A popular network is called the inverse graphics network, which uses a VAE like strategy to learn latent variables that can give you control in generating various kinds of objects, 3D objects, etc. You could control the light angle, the rotation, the kind of object, so on and so forth using an inverse graphics network, which effectively uses a VAE strategy. And finally, another kind of network called Adversarial Auto Encoders, which brings together GANs and VAEs, will be a topic that we discuss immediately next lecture.

(Refer Slide Time: 33:38)



Your homework for this time is to go through "*Tutorial on Variational Auto Encoder*", as well as go through this VAE example in PyTorch. It may also be worthwhile going through the original paper on Variational Auto Encoders if you are interested. There are a couple of questions for you to think about. Why does the encoder of a VAE map to a vector of means and standard deviations?

Why does the encoder not map to a vector of means and a covariance matrix? This encoder gets you the approximate posterior. A similar question What about the decoder? If we assumed a Mean Square Error for the reconstruction loss? The second term in your objective for VAE, remember can use the Mean Square to get a reconstruction loss. What is the covariance of the $p(x \mid z)$ Gaussian distribution? What do you think it will look like? Think about it and we will discuss it soon.