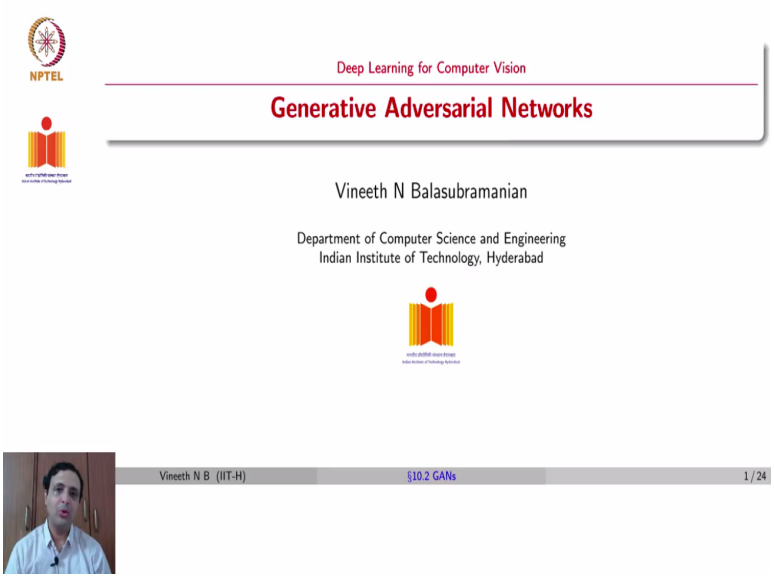
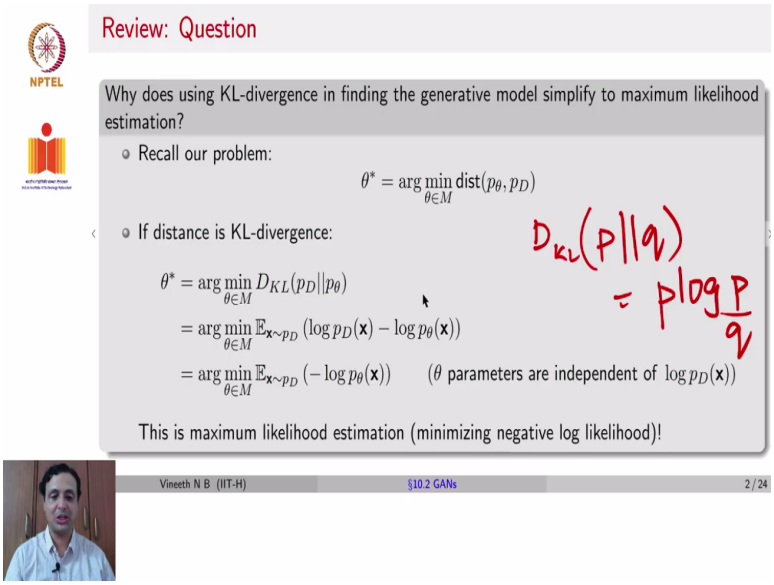
Deep Learning for Computer Vision
Professor Vineeth N Balasubramanian
Department of Computer Science and Engineering
Indian Institute of Technology, Hyderabad
Generative Adversarial Networks
Part 01

(Refer Slide Time: 00:14)



We now talk about the first kind of Deep Generative Models, which are perhaps arguably the most popular Deep Generative Model too, Generative Adversarial Networks, or GANs.

(Refer Slide Time: 00:31)

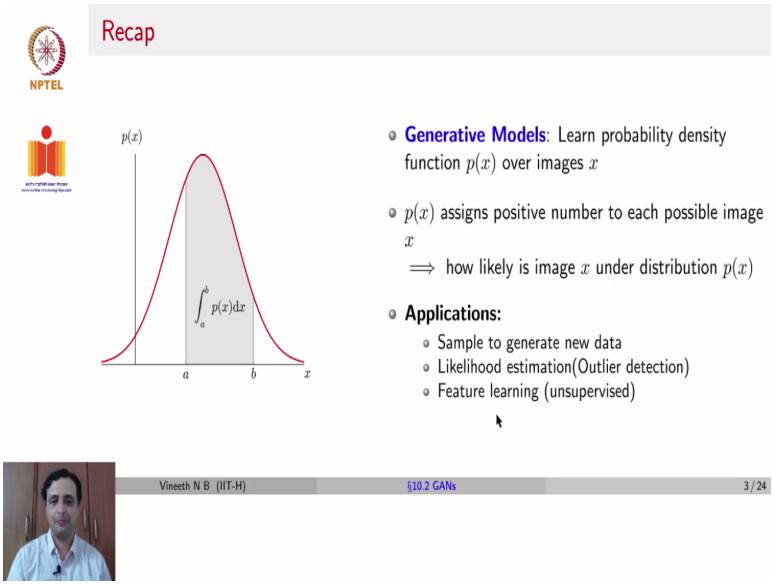


We left behind the question from the last lecture: *Why does using KL divergence in finding the generator model simplify to maximum likelihood estimation*? If you recall the problem, the problem was θ^* is argmin over all θ's from a family of distributions M, some distance function of p_θ from p_D. This distance is KL divergence; remember that the definition of KL-divergence is if you had two probability distributions, p and q, for simplicity. This is given by $plog(p \mid q)$. That is the formal definition of KL divergence.

So, in this case, you would have the KL divergence of p_D and p_θ. So, the first term p_D does not depend on θ, so it does not matter for our optimization. So, you are left with $log(p_D \mid p_\theta)$, which can be written out as $log\, p_D$ minus $log\, p_\theta$. Once again, here, $log\, p_D$ is the distribution of data and does not depend on θ.

So, you are left with argmin over θ expectation over all the data samples from your distribution p_D, $-\ log\, p_\theta(x)$. This is maximum likelihood estimation or minimizing negative log-likelihood. So, fairly trivial thing to show as we mentioned the last time.

(Refer Slide Time: 02:20)

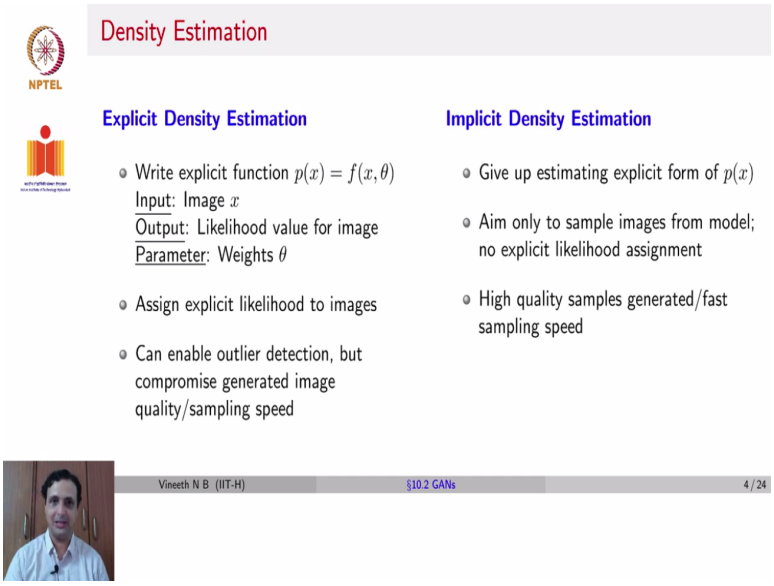


Let us recap some of the concepts that we spoke about in the last lecture. We said that generative models, try to learn a probability density function $p(x)$ over a given set of images, training data,

x. If you knew the distribution, if that was parametrized as a Gaussian, $p(x)$ assigns a positive number to each possible image, depending on its probability of being generated by that distribution, it tells us how likely that image x is under that distribution $p(x)$.

And we said that applications of generative models include sampling and generating new data, likelihood estimation, which could help you in outlier detection, because you have a certain distribution, and you have a certain likelihood of a data point belonging to a distribution. If that likelihood is very low, it is likely that the point was an outlier. Similarly, you can also do feature learning, which we will talk about more in this lecture.

(Refer Slide Time: 03:36)



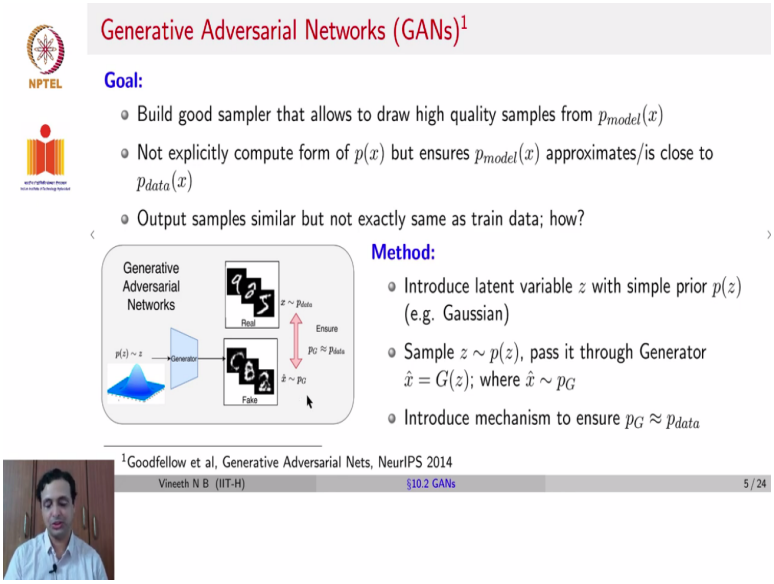
We also talked about the last lecture on density estimation of the probability distribution from an implicit perspective and an explicit perspective. These also define the different approaches for Deep generative models. In explicit density estimation methods, you try to write an explicit function for the probability distribution. For example, you may say $p(x)$ is equal to $f(x, \theta)$, where f could be a Gaussian function in the case in which we, you assumed a Gaussian distribution for the density.

In this case, the input would be image x, the output $p(x)$ would be the likelihood value for image x, and the parameters would be the weights θ, which defines that function that you assign for the density. So, in this case, an explicit likelihood is assigned for each image. With explicit density

estimation approaches, you will realise as you see different methods, they are very good at outlier detection for the same reason that we just said.

So, if a point's likelihood of belonging to distribution is very low, it is perhaps an outlier. But explicit density estimation approaches do struggle to generate very high-quality images, sometimes can also be slow in generating images. On the other hand, implicit density estimation approaches do not assign a functional form to $p(x)$. Instead, they only aim to ensure that given a model, you can sample images from that model without worrying about an explicit likelihood assignment for each such sample. So, that is not part of the model. It happens that such methods end up providing avenues for better sample generation and faster sampling speed.

(Refer Slide Time: 05:58)



With that, recall, let us move on to the focus of this lecture, which is Generative Adversarial Networks. These networks were developed in 2014 by Goodfellow et al. The goal of GANs is to build a good sampler that allows drawing high-quality samples from the $p_{model}(x)$. The $p_{model}(x)$ defines samples drawn from a model learned through this algorithm.

As we just mentioned, there is no explicit computation of a certain functional form for $p(x)$. The only objective we have is to ensure that whatever images we sample from the model, the distribution of those samples, which we call $p_{model}(x)$, remember, we will not assign any

likelihood value for $p_{model}(x)$. We want to ensure that if you collected a set of samples from that model, that distribution of samples should look similar to your original data distribution.

There is no likelihood assignment for each sample. So, we ideally want the output samples to be similar but not the same as your training data. Why do we say that? Because if we have the same samples as training data, the neural network is perhaps memorizing and not learning much. Then you do not need such a model. You only need a model, which can then generate diverse samples beyond what you already have.

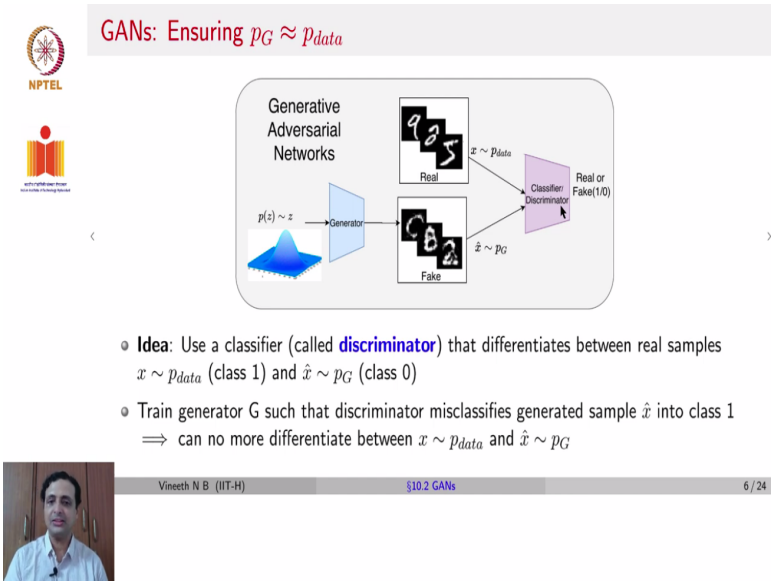
So, How do you achieve this goal? The key idea in Generative Adversarial Networks or GANs is to introduce a latent variable z. So, this is the latent variable with a simple prior, for example, a Gaussian prior. Once you sample from that Gaussian, you pass it through a module called a Generator. In our case, a Generator could be a neural network, and you give input to that generative neural network, and the network outputs an image.

So, this module is known to us from whatever we have seen so far. We have seen semantic segmentation methods, where the output layer is the size of an image itself. Similarly, here also, the output of the generator would be an image itself. And that is what we define as the distribution coming out of the generator. Let us call that $\hat{x}$, the samples from p_G, where p_G denotes the distribution of images generated by the generator G.

Now, what do we want? We want to ensure that if we had a distribution of training data given by p_{data}, we want to ensure that the distribution p_G must be close to p_{data}. Observe that we are not trying to assign a likelihood to every data point $\hat{x}$ or even x, for that matter. We want that distribution p_G to be close to the distribution p_{data}. The challenge in implementing this is not imposing any particular parameterization on p_G or p_{data}.

So, we have to find the equivalence or ensure that p_G becomes close to p_{data} without knowing its distribution. Remember here; the Gaussian is only an input vector. We are not assuming any Gaussian or any parameterization on your p_G or p_{data} distributions.

(Refer Slide Time: 10:23)

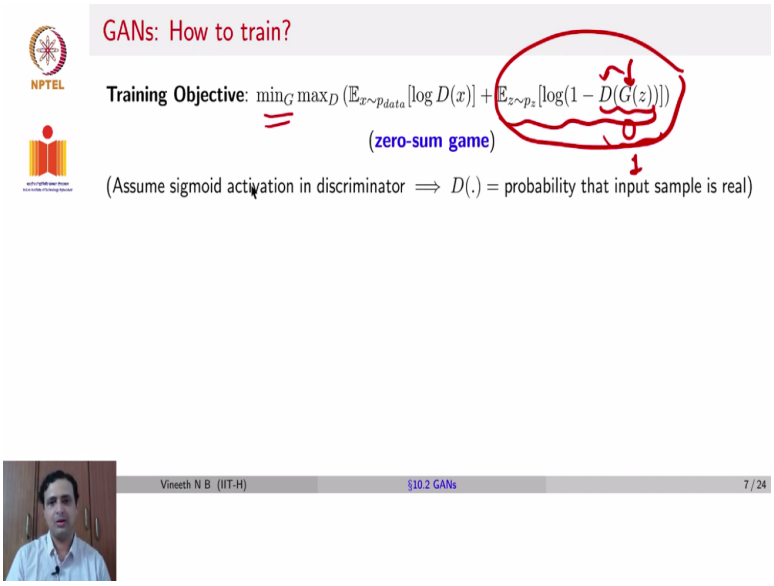


So, how do we ensure that p_G is more or less close to p_{data}? To do this, the originators of this particular method, Goodfellow et al., had an interesting idea. They introduced a classifier called a discriminator, which we will denote as D in this lecture, to differentiate between real samples and generated samples. So, if data came from p_{data}, it would be given class 1.

And if data came from the generated distribution that is $\hat{x}$ coming from p_G, then it would give a class 0 for that image. How does this benefit? Our goal is to train the generator to ensure that the discriminator misclassifies the generated sample $\hat{x}$ into class 1. It can no more differentiate between the original distribution p_{data} and the new distribution, p_G. So, the way we will equate p_G and p_{data} is through this discriminator, which the generator will seek to confuse. The discriminator's job is to separate the fake samples from the real samples. By fake samples, we mean the generated samples.

The job of the generator is to fool the discriminator. So, you can also consider this, like a cop and thief game, where discriminator is like a cop that can separate real and fake and the job of the generator is to fool the discriminator. That is the overall idea.

(Refer Slide Time: 12:24)



How do you train such a generator or a discriminator? So, the training objective for GANs is given by a min-max optimization problem, you minimize over G, which are the parameters of your generator network, you maximize over D, which are the parameters of your discriminator, or the classifier network, the expectation of data coming from the real distribution, $log\ D(x)$, why is this correct?

We are saying that we would like the discriminator to maximize the log-likelihood of those data points that come from your original training distribution. The second part states that if you now take a sample from the Gaussian and give that Gaussian to your generator, G and the generator's output is given to the discriminator. We want to ensure that the output is 0. At least the discriminator must aim to make this value 0.

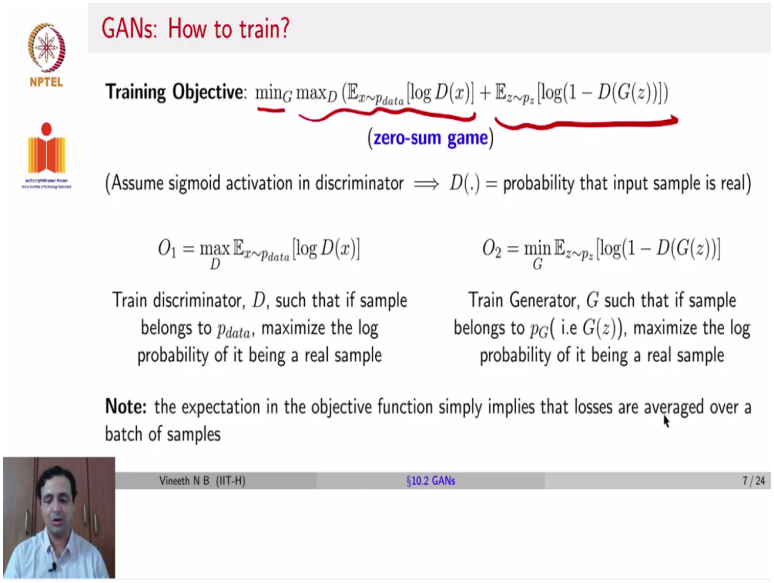
And that would happen when this entire quantity is maximized. On the other hand, the generator's job is to ensure that this last quantity goes to 1, minimzing this entire quantity. That is why you minimize over G and maximize over D. This also is intuitive because generator and discriminator are playing a cop and thief game. Each would like to output the other.

So, while the discriminator wants to maximize terms in the objective, the generator wants to minimize corresponding terms in the objective. So, such a min-max problem is also known as a zero-sum game. This has origins in game theory, which will not get into now. We assume that the

discriminator has a sigmoid activation at its output layer. Remember, the discriminator in this particular example is a binary classifier.

It only has to say real or fake. So, the best activation function that you can have in the output layer of a binary classifier is the sigmoid activation. We now assume that a discriminator has a sigmoid activation function in its output layer.

(Refer Slide Time: 15:18)

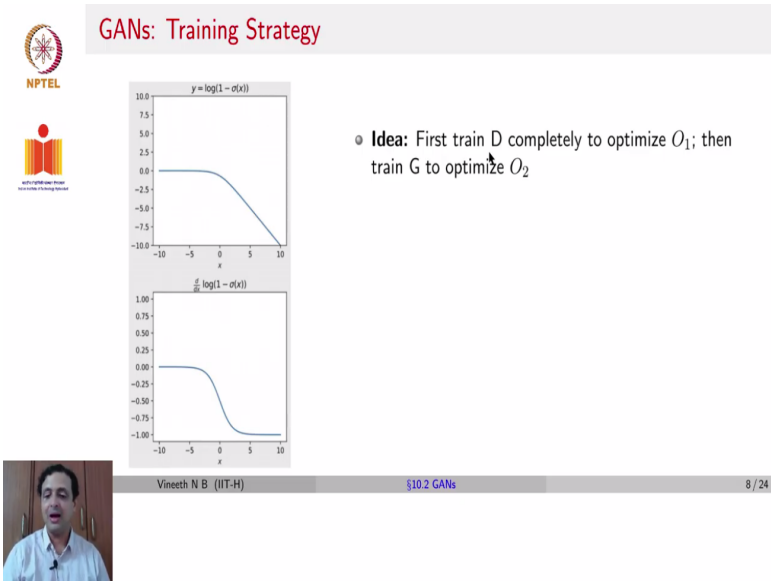


So, let us try to parse this objective function a bit differently. If you look at the first term, if you look at this part of the term, let us write that and call it objective 1. So, objective 1 states that we would like to train the discriminator such that if the sample belongs to p_{data}, which is the true training data distribution. We maximize the log probability of it being a real sample. The second part is objective 2, which talks about minimizing the generator with respect to the second term.

Remember that the first term does not have anything to do with G, and hence, in the minimization of G, the first term does not matter and can be excluded. So, while minimizing G, what are we looking for? We are looking to train the generator G, such that if the sample belongs to p_G, that is, its output of the generator G, we would like to maximize the log probability of it being a real sample. That is what the generator would like to do.

Although, the discriminator would also like to maximize this quantity. What do we mean by expectation in these two terms? In practical implementation, the expectation simply means that the losses are averaged over a batch of samples. What does a batch of samples mean here? We will see that in a moment when we see the algorithm.

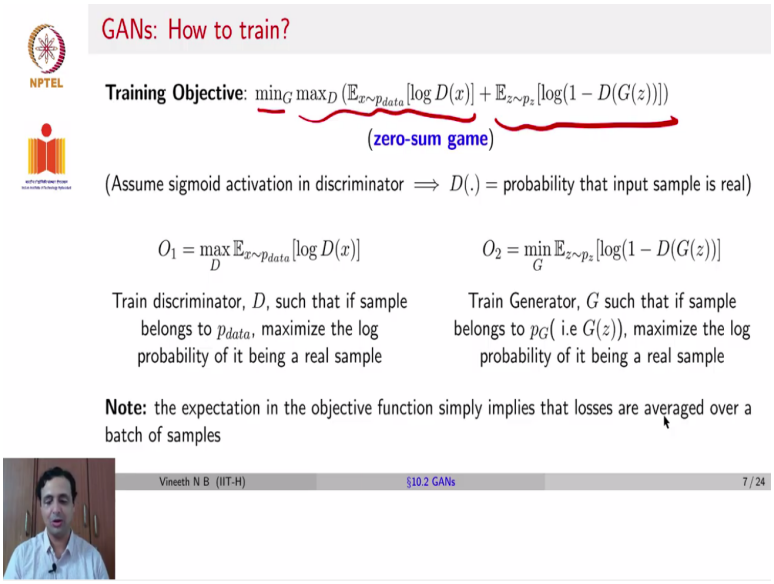
(Refer Slide Time: 16:57)



Now, coming to the training strategy. So, the first thing that you can think of is you have two networks to train D and G. So, far, we have seen several other approaches where we had two networks to train, we had detection with two heads, we talked about a Siamese network with two branches. We talked about a two-stream CNN with two branches. But in all of those examples, both branches were trained with the same loss or the same objective.

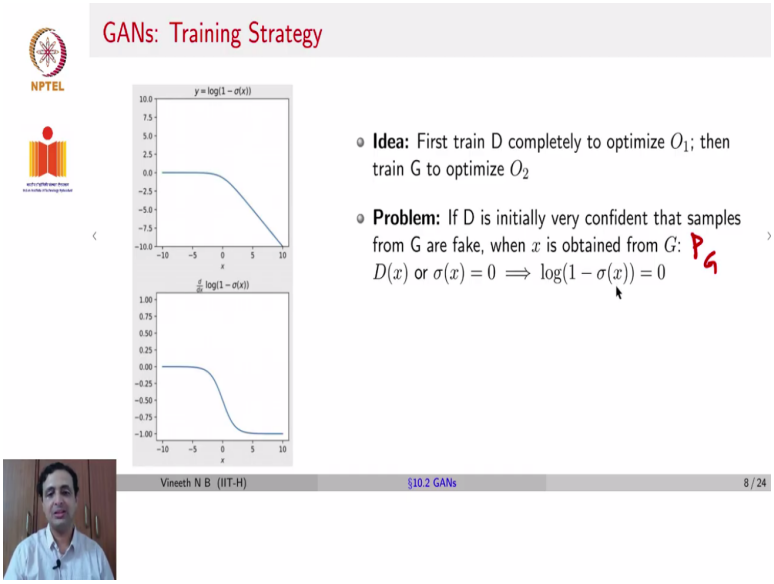
We had examples like triplet loss were things slightly changed. But otherwise, it was the same loss. You minimize the same quantity across the complete network to a large extent. So, what can we do here? One option is we can train D completely first. So, we have a good discriminator.

(Refer Slide Time: 17:56)



And that can be done by optimizing only the first part of the objective O1.

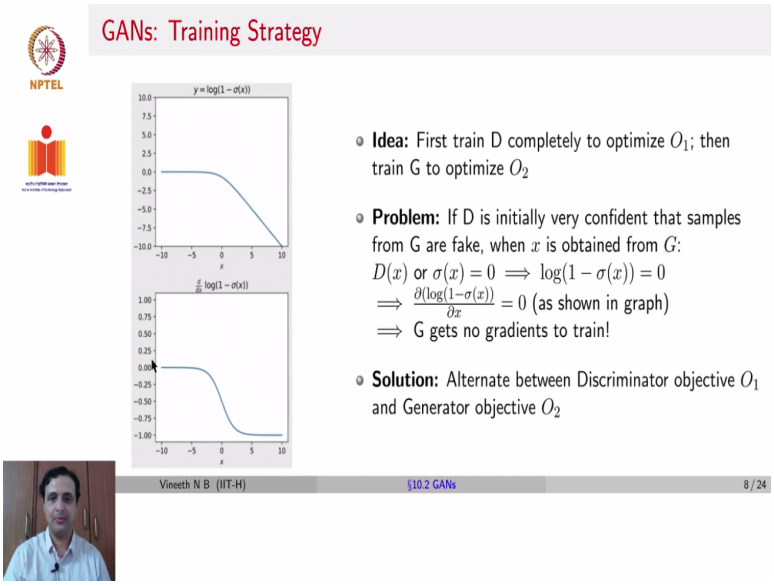
(Refer Slide Time: 18:01)



And once we have that, we can train G to optimize O2. Does that work? Does that have any problems? Let us try to think this through. If D is initially very confident, which means you have trained an excellent discriminator or a classifier, then it would be able to say that any sample that comes from G is a fake. So, which means, if you get an x that is obtained from G, or corresponding to the distribution, p_G, then $D(x)$, or $\sigma(x)$, which is sigmoid activation function in

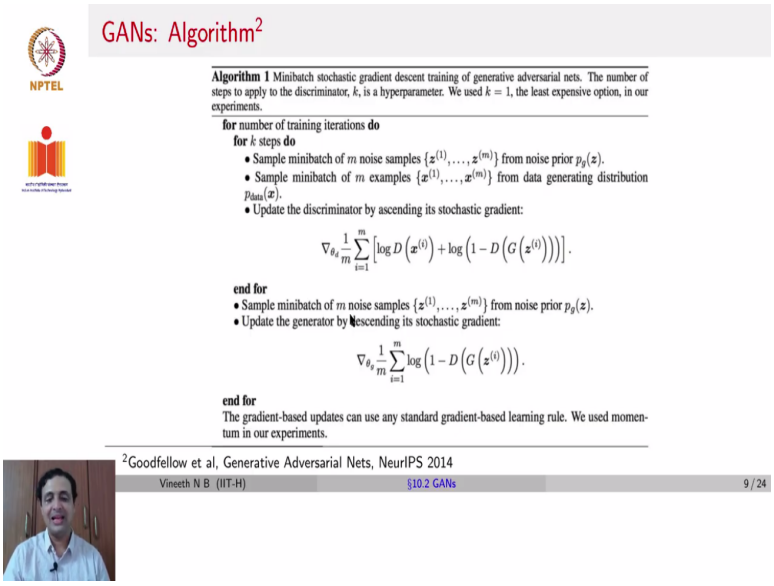
D would be equal to 0. which means $log\,(1 - D(x))$, in this case, would be $log\,(1 - \sigma(x))$ would be equal to 0.

(Refer Slide Time: 19:01)



And if you see the graph of $log\,(1 - \sigma(x))$ and the gradient of $log\,(1 - \sigma(x))$, when $log\,(1 - \sigma(x))$ is 0, obviously the gradient is also 0. So, you will get a zero gradient, which means G will not get any gradients to train and will never learn. So, training discriminator completely well in the beginning will not help us get a good generator, which is the main objective of a GAN. So, what do we do? We want to alternate between training the discriminator and training the generator using O1 and O2, respectively.

(Refer Slide Time: 19:54)



Let us try to see how this is done in the algorithm for GANs. So, the original paper by Goodfellow recommends that we first sample. So, for the K steps, you first sample a mini-batch of M noise samples from the noise prior $p_g(z)$. So, remember, we had a Gaussian, from which we get a few vectors, which we call noise samples. Similarly, we also sample a mini-batch of M examples from the original data distribution p_{data}, which is your training data.

Now, remember, your overall objective is $log(D(x)) + log(1 - D(G(z)))$. And remember, the discriminator wants to maximize both of these. That is what we have said so far. And a maximization problem is solved by gradient ascent, just like how a minimization problem is solved by gradient descent. A maximization problem is solved by a gradient ascent, where in each iteration, you go in the direction of the positive gradient, not the negative gradient the way you did with gradient descent.

So, we update the discriminator by ascending its stochastic gradient, which is obtained by taking this loss function and differentiating with respect to each weight in the discriminator network. And you do this for K steps. So, for the K steps, you update the discriminator. It is not yet completely trained. Once you have done it for the K steps, now switch over and train the generator. So, you sample a mini-batch of M noise samples from the noise prior.

Update the generator now by descending its stochastic gradient. Because for the generator, we wanted to minimize the second term in the objective, the first term in the objective, anyway, did not depend on G, only the second term dependent. So, we would now like to minimize $log(1 - D(G(z)))$ with respect to the parameters in the generator network G. Now, this is repeated over training iterations.

And this is used to come up with a model for the generator finally. At test time, what do you do? Once you have trained the entire model, you can discard the discriminator. You take a sample from your Gaussian, send it through the generator network, and the image you get is what you would assume belongs to your original data distribution. One point here is what does it mean for such a network to converge?

So far, we spoke about whenever we had a loss function, we always wanted to minimize the loss function and wanted to ensure that the loss goes to 0. To avoid overfitting, we perhaps may not let it go to 0. But we at least wanted to see the loss reducing over iterations. However, here, we have two components, where one is perhaps trying to increase the objective function value, the other is trying to reduce the objective function value. So, what does convergence mean for such a network? Let us see that in more detail.