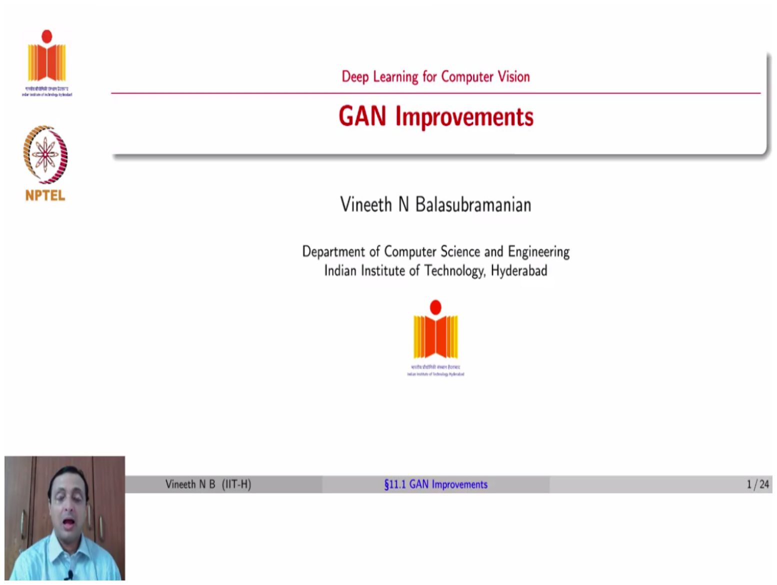
Deep Learning for Computer Vision
Professor Vineeth N Balasubramanian
Department of Computer Science and Engineering
Indian Institute of Technology, Hyderabad
GAN Improvements

(Refer Slide Time: 00:14)



Deep Learning for Computer Vision

GAN Improvements

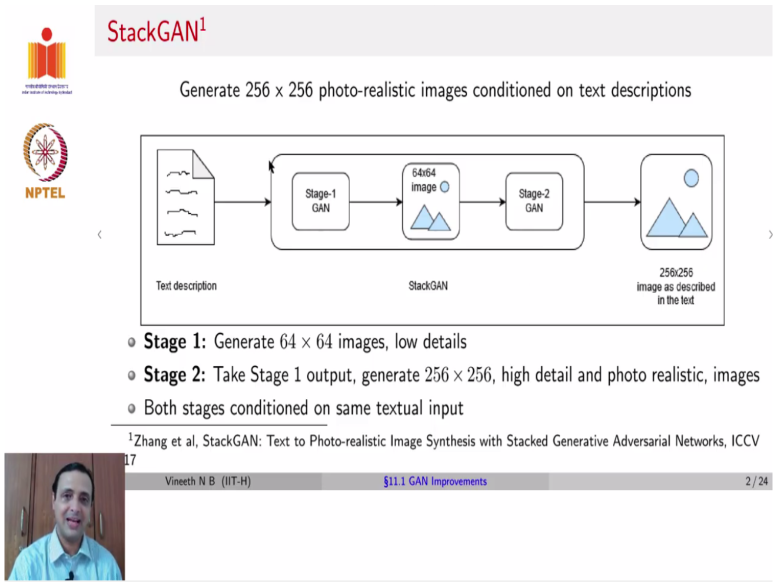
Vineeth N Balasubramanian

Department of Computer Science and Engineering
Indian Institute of Technology, Hyderabad

Vineeth N B (IIT-H) §11.1 GAN Improvements 1 / 24

Having seen various kinds of Deep Generative Models like GANs, VAEs, Normalizing flows, Autoregressive flows last week. We will now move on to the various ways in which GANs have been improved over the last few years. We will talk a bit of disentangling of VAEs, and then move on to applications of GANs to Images and Videos. Let us start with a few improvements over Vanilla GANs.

(Refer Slide Time: 00:50)



StackGAN[1]

Generate 256 x 256 photo-realistic images conditioned on text descriptions

- **Stage 1:** Generate 64×64 images, low details
- **Stage 2:** Take Stage 1 output, generate 256×256, high detail and photo realistic, images
- Both stages conditioned on same textual input

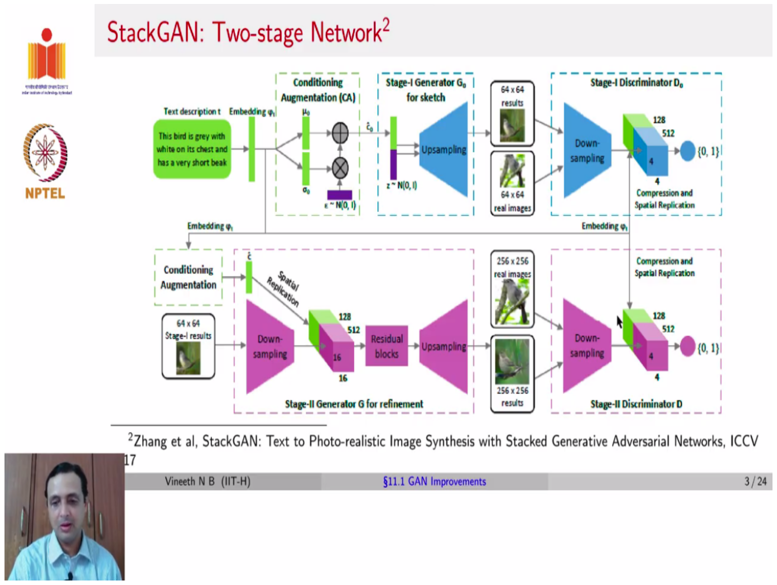
[1]Zhang et al, StackGAN: Text to Photo-realistic Image Synthesis with Stacked Generative Adversarial Networks, ICCV 17

Vineeth N B (IIT-H) §11.1 GAN Improvements 2 / 24

The first method that we will talk about is known as Stack GAN. This work was published in ICCV 2017. The goal of this work was to generate reasonably good resolution 256 * 256 photorealistic images conditioned on Text Descriptions. So here is a high-level flow. The entire GAN model is conditioned on a Text Description which could be a caption, for instance. Standard NLP methods such as Word2Vec, GloVe, BERT, etc., are used to get an embedding of this Text Description, which is provided as an input to the GAN.

With this input in the first stage, the GAN generates 64 * 64 images, but perhaps a few high-level details or low-frequency details. These images are then provided as an input to the next stage of the Stack GAN, which generates 256 * 256 reasonably high detail photorealistic images and both these stages are conditioned on the same text input.

(Refer Slide Time: 02:20)



StackGAN: Two-stage Network[2]

[2]Zhang et al, StackGAN: Text to Photo-realistic Image Synthesis with Stacked Generative Adversarial Networks, ICCV 17

Vineeth N B (IIT-H) §11.1 GAN Improvements 3 / 24

Let us see the entire architecture now. So you have the Text description, which is given as input to the entire Stack GAN. In this example, the text description reads, "*This bird is grey with white on its chest and has a very short beak*". So you obtain an embedding of this text, as I just mentioned, using, say, Word2Vec or GloVe or BERT or any other embedding of your choice and this embedding is added to the standard Gaussian noise is given to a GAN.

So, you can see that you get a mean and standard deviation from this embedding and this mean and standard deviation is used to obtain a sample. The obtained sample is concatenated with a sample from the standard normal, which becomes the entire input to the first stage generator. So the first stage generator then goes through an Upsampling Generator Module, which generates a $64 * 64$image that you see here.

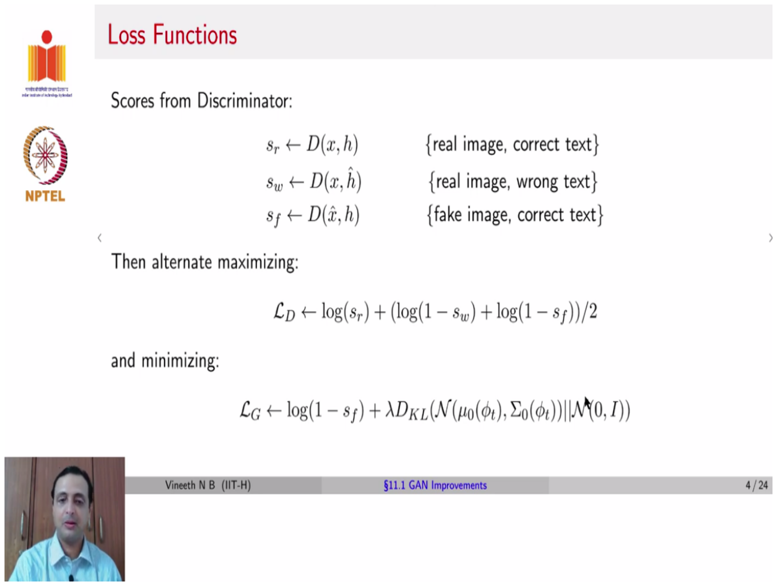
The generated image and a set of real images form a minibatch. This minibatch is provided to the discriminator, which downsamples these images. And before the final step of classifying this input as real or fake, you also include the embedding phi t, again here. So you see here that an arrow comes all the way and gets combined to this output of the Downsampling Module.

And the job of this Discriminator now is to classify this tuple of a generated image whose representation we are considering, which is this blue block here, and the text embedding together to be real or fake. This forms the first stage. In the second stage, the results of the first stage are given as input along with the embedding of the text again; both of these then go through the second generator stage.

So, you can see here that the image goes through a Downsampling, then the Text embedding is then concatenated to the Downsample representation. This then goes through a set of residual blocks, which is then Upsampled to get the final output 256 * 256 image, which is what you see here.

Now, that goes to the second discriminator, where you give the generated 256 * 256 images and one of the real 256 * 256 images. Once again, the output of the Downsample representation is concatenated with the text embedding, and the Discriminator has to classify each such tuple as real or fake.

(Refer Slide Time: 05:33)



Loss Functions

Scores from Discriminator:

$$s_r \leftarrow D(x, h) \qquad \{\text{real image, correct text}\}$$
$$s_w \leftarrow D(x, \hat{h}) \qquad \{\text{real image, wrong text}\}$$
$$s_f \leftarrow D(\hat{x}, h) \qquad \{\text{fake image, correct text}\}$$

Then alternate maximizing:

$$\mathcal{L}_D \leftarrow \log(s_r) + (\log(1 - s_w) + \log(1 - s_f))/2$$

and minimizing:

$$\mathcal{L}_G \leftarrow \log(1 - s_f) + \lambda D_{KL}(\mathcal{N}(\mu_0(\phi_t), \Sigma_0(\phi_t)) || \mathcal{N}(0, I))$$

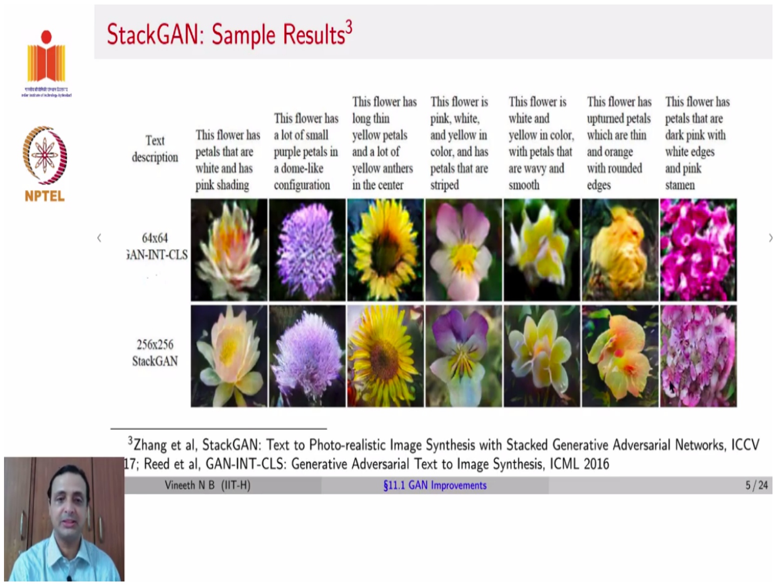
Vineeth N B (IIT-H) §11.1 GAN Improvements 4 / 24

Now, let us understand how to, how the Discriminator can look at the tuple and classify Real or Fake. So, there are three kinds of Scores the Discriminator has to obtain/provide in this case. So, the first one is when the real image is provided with the correct text. Obviously, the discriminator would want the score of such an input to be one, because both are correct. There is a real image and the corresponding correct text. There is also another setting where you have a real image and an incorrect text.

In this case, the discriminator should ideally give a low score. Similarly, you have a fake image with the correct text, which the discriminator should give a low score, but the generator should try to increase this particular score. These are denoted by s_r, s_w and s_f. Now, let us try to understand how optimization actually works. So, the Stack GAN alternately maximizes the discriminator loss and minimizes the generator loss.

Similar to what we saw with a GAN. In this case, the discriminator, as we just saw, would try to maximize the log likelihood of the first score and minimize the log likelihood of the second and third scores, which is what is given by the second and third terms here. We just saw that when we understood the Scores. Similarly, the generator tries to maximize $log(1 - s_f)$ because that is the job of the generator. It also tries to minimize the KL divergence between the output of the mu's and sigma's that you see in the initial layer at the end of the text embedding with the standard normal distribution.

(Refer Slide Time: 07:47)



StackGAN: Sample Results[3]

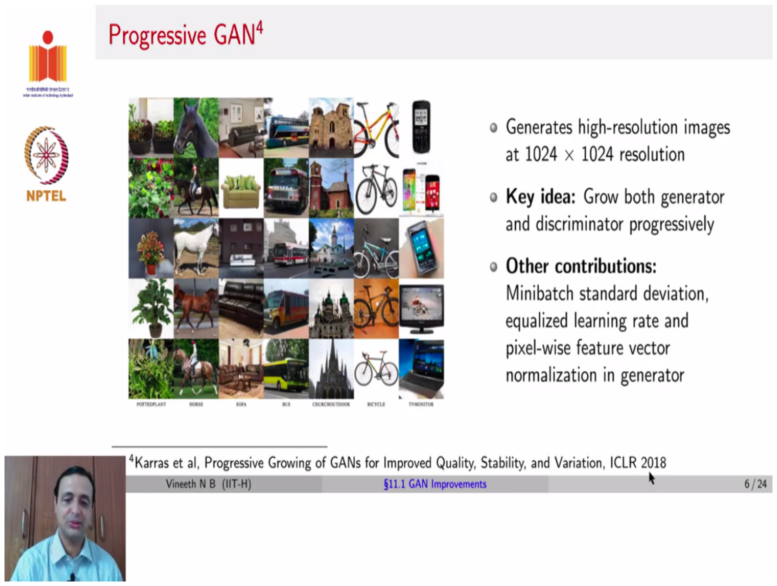
Text description	This flower has petals that are white and has pink shading	This flower has a lot of small purple petals in a dome-like configuration	This flower has long thin yellow petals and a lot of yellow anthers in the center	This flower is pink, white, and yellow in color, and has petals that are striped	This flower is white and yellow in color, with petals that are wavy and smooth	This flower has upturned petals which are thin and orange with rounded edges	This flower has petals that are dark pink with white edges and pink stamen
64x64 GAN-INT-CLS							
256x256 StackGAN							

[3]Zhang et al, StackGAN: Text to Photo-realistic Image Synthesis with Stacked Generative Adversarial Networks, ICCV 17; Reed et al, GAN-INT-CLS: Generative Adversarial Text to Image Synthesis, ICML 2016

Vineeth N B (IIT-H) §11.1 GAN Improvements 5 / 24

Here are some results that are obtained from Stack GAN. What you see on the top row is a baseline method called GAN-INT-CLS that was published in 2016. And the bottom row shows the results of the Stack GAN. So you can see here in the first column, the caption says, this flower has petals that are white and has pink shading. You can see here that the quality obtained by the Stack GAN is far more photorealistic than of earlier methods. And this holds for all these images that you see at the bottom row.

(Refer Slide Time: 08:29)


Progressive GAN[4]

POTTEDPLANT
HORSE
SOFA
BUS
CHURCHOUTDOOR
BICYCLE
TVMONITOR

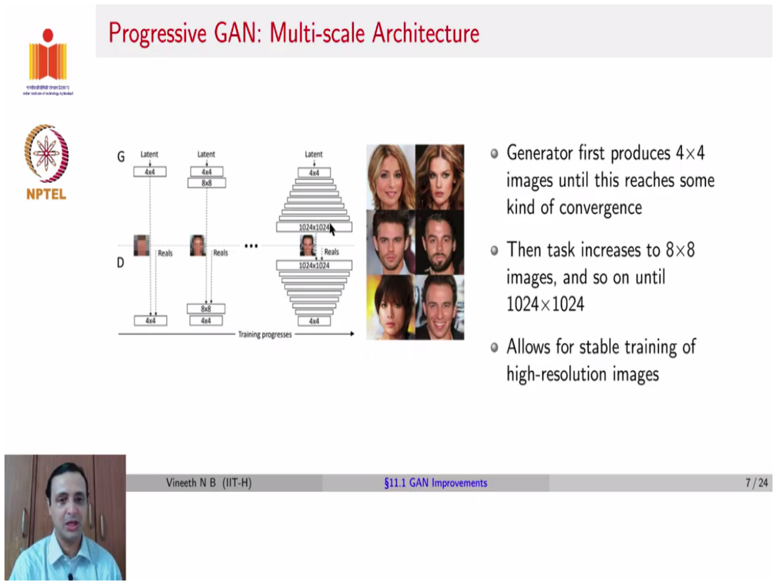
- Generates high-resolution images at 1024 × 1024 resolution
- **Key idea:** Grow both generator and discriminator progressively
- **Other contributions:** Minibatch standard deviation, equalized learning rate and pixel-wise feature vector normalization in generator

[4]Karras et al, Progressive Growing of GANs for Improved Quality, Stability, and Variation, ICLR 2018

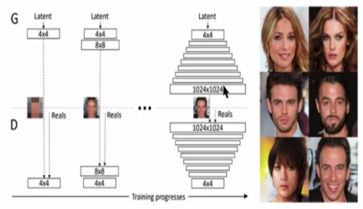
Vineeth N B (IIT-H) §11.1 GAN Improvements 6 / 24

Another popular method that came in 2018 is known as the Progressive GAN. The Progressive GAN is designed for generating high-resolution images up to 1024 * 1024. And the key idea is in the name of the method itself is the method progressively grows, the generator and the discriminator. We will soon try to decipher what this means. This work was published in ICLR 2018. It also had a few other design decisions, such as using a standard deviation in a given Minibatch, a concept of an equalized learning rate, and a Pixel wise feature vector normalization in the generator, which we will also see soon.

(Refer Slide Time: 09:28)



Progressive GAN: Multi-scale Architecture



- Generator first produces 4×4 images until this reaches some kind of convergence
- Then task increases to 8×8 images, and so on until 1024×1024
- Allows for stable training of high-resolution images

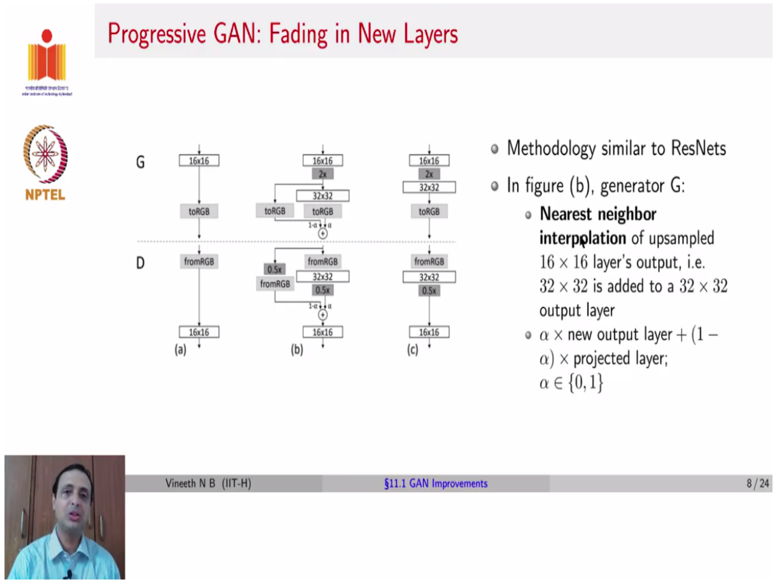
Vineeth N B (IIT-H) §11.1 GAN Improvements 7 / 24

The idea of Progressive GAN is shown in the image on the left. The generator first produces a 4 * 4 image shown in the left-most part of this image. So, you have the latent vector that comes from a standard normal, a very simple network, which generates a 4 * 4 image. This is provided to a Discriminator D, along with a real sample, to judge whether it is real or fake.

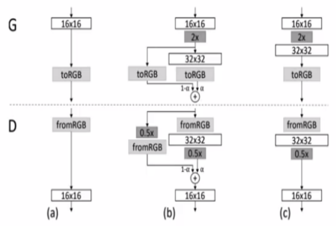
This then increases to 8 * 8 in the next iterations of training. You can see here that compared to the image generated in the 4 * 4 setting, the 8 * 8 setting is still blurry but has some more details when compared to the 4 * 4 setting. You can also notice here that there are some layers added in the generator and the discriminator when the next higher resolution is generated.

And this is repeated over and over again until we get the final generation of 1024 * 1024 image. The key idea that this allows is for stable training of GANs for generating high-resolution images.

(Refer Slide Time: 11:02)



Progressive GAN: Fading in New Layers



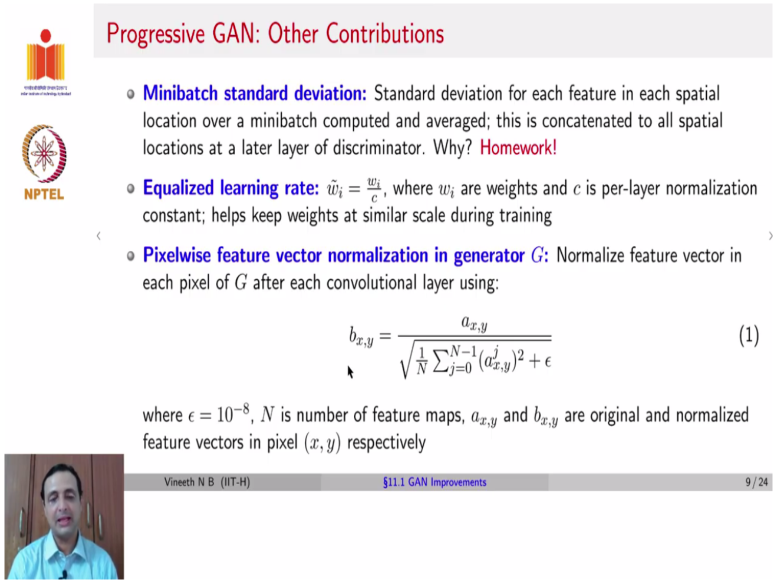
- Methodology similar to ResNets
- In figure (b), generator G:
  - **Nearest neighbor interpolation** of upsampled 16×16 layer's output, i.e. 32×32 is added to a 32×32 output layer
  - $\alpha \times$ new output layer $+ (1 - \alpha) \times$ projected layer; $\alpha \in \{0, 1\}$

Vineeth N B (IIT-H) §11.1 GAN Improvements 8 / 24

Whenever the Progressive GAN goes from generating a certain resolution to the subsequent higher resolution, as I just mentioned, there are new layers introduced. How are these layers introduced? This procedure is very similar to ResNets. So, you can see that in subfigure b in the image. You see now that while you had an initial resolution 16 * 16, when you step up to 32 * 32.

The nearest neighbour interpolated version of the 16 * 16 layer output; when you interpolate, you get a 32 * 32 output added to the 32 * 32 layer through a skip connection. How is that added? You can see an α and 1 − α. So, the new output layer is the final output is given by α into the new output layer plus 1 − α into the projected layer. By projected layer, we mean the output of the nearest neighbour interpolation. This allows Progressive GAN to fade in new layers organically to generate better images.

(Refer Slide Time: 12:31)



Progressive GAN: Other Contributions

- **Minibatch standard deviation:** Standard deviation for each feature in each spatial location over a minibatch computed and averaged; this is concatenated to all spatial locations at a later layer of discriminator. Why? Homework!
- **Equalized learning rate:** $\tilde{w}_i = \frac{w_i}{c}$, where w_i are weights and c is per-layer normalization constant; helps keep weights at similar scale during training
- **Pixelwise feature vector normalization in generator G:** Normalize feature vector in each pixel of G after each convolutional layer using:

$$b_{x,y} = \frac{a_{x,y}}{\sqrt{\frac{1}{N}\sum_{j=0}^{N-1}(a^j_{x,y})^2 + \epsilon}} \quad (1)$$

where $\epsilon = 10^{-8}$, N is number of feature maps, $a_{x,y}$ and $b_{x,y}$ are original and normalized feature vectors in pixel (x, y) respectively

Vineeth N B (IIT-H) §11.1 GAN Improvements 9 / 24

As I mentioned, Progressive GAN also introduces a few other design decisions. One such contribution is known as Minibatch standard deviation. Here, the standard deviation is computed and averaged at each spatial location over a Minibatch, So, you can imagine that when the generator generates a particular feature map, you take every spatial location, which is at every pixel, you compute its average, across the entire Minibatch, you will get a certain standard deviation for that value across the Minibatch.

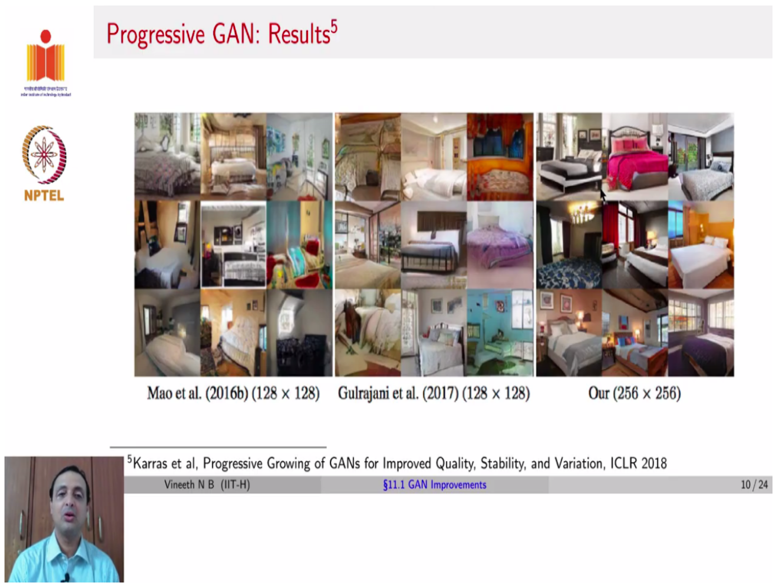
That is concatenated to all spatial locations at a later layer of a discriminator. Why do you think this is done? Think about it. It will be your homework for this lecture. Another contribution that Progressive GAN brings to the table is known as the Equalized learning rate. In this case, the weights in each layer of the generator network are normalized by a constant c, which is specified per-layer.

So, this constant c is a per-layer normalization constant. Why is this done? This allows us to vary c in every layer and thus help keep the weights at a similar scale during training. So, why is this called Equalized learning rate? Because the value of the weight effectively affects the gradient and hence the learning rate. While methods such as Adam, AdaGrad, and so forth adapt the learning rate, they may have low values with the weights themselves very low or very high.

Normalizing by a constant allows these weights to be of a similar scale during training across all layers. This allows better learning. Another contribution that Progressive GAN makes is a pixel-wise feature vector normalization in the generator. For each convolutional layer of the generator, the normalization is defined by $a_{x,y}$. Here, (x,y) is the pixel in a^{th} feature map. It is divided by N, where N is the number of feature maps. The summation j goes from 0 to $N - 1$ that is across the N feature maps, $(a^{j}_{x,y})^2 + \epsilon$.

The denominator here considers the pixel value at the same location across all of the feature maps and normalizes the value at the a^{th} feature map with this denominator. The ϵ here is for numerical stability to avoid a divide by 0 error. And the output is denoted as $b_{x,y}$, which is the normalized value.

(Refer Slide Time: 15:59)



Progressive GAN: Results[5]

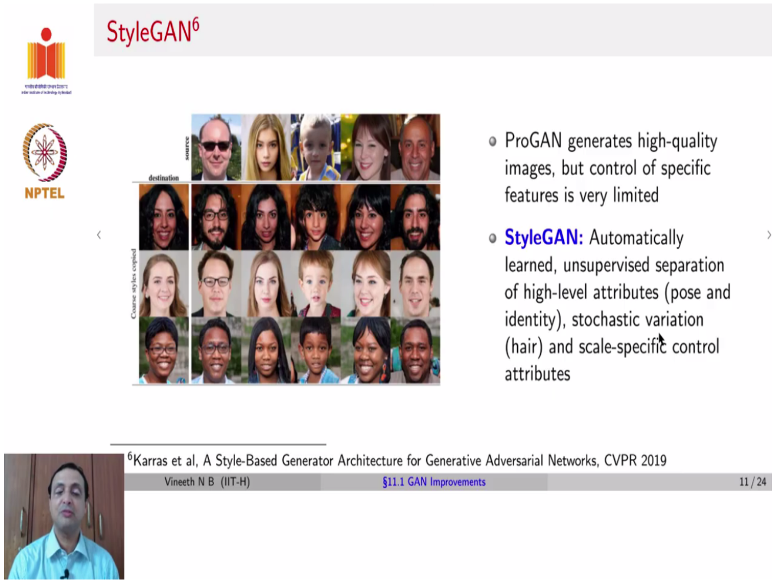
Mao et al. (2016b) (128 × 128) Gulrajani et al. (2017) (128 × 128) Our (256 × 256)

[5]Karras et al, Progressive Growing of GANs for Improved Quality, Stability, and Variation, ICLR 2018

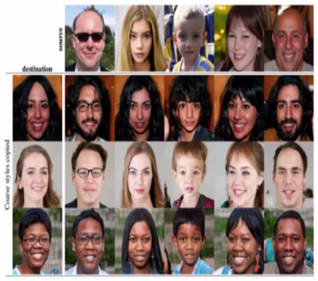
Vineeth N B (IIT-H) §11.1 GAN Improvements 10 / 24

With these contributions, Progressive GAN reports impressive results. The results are compared with earlier works such as Mao et al. and Gulrajani et al., and one can see that with the Progressive GAN approach, the result is fairly photorealistic and high resolution with more details when compared to earlier methods.

(Refer Slide Time: 16:25)



StyleGAN[6]



- ProGAN generates high-quality images, but control of specific features is very limited
- **StyleGAN:** Automatically learned, unsupervised separation of high-level attributes (pose and identity), stochastic variation (hair) and scale-specific control attributes

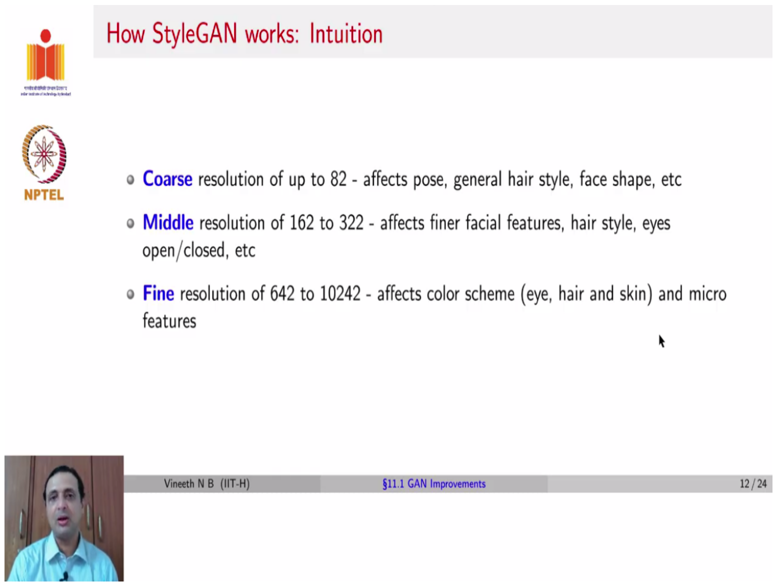
[6]Karras et al, A Style-Based Generator Architecture for Generative Adversarial Networks, CVPR 2019

Vineeth N B (IIT-H) §11.1 GAN Improvements 11 / 24

A third more recent improvement of GAN is known as StyleGAN, published in CVPR 2019. Progressive GAN, also known as ProGAN, generates high-quality images but does not give the capability to control specific features in the generation. For example, it is difficult to use a Progressive GAN and say that you would like to take an image and add a specific colour or change a particular attribute in the image.

StyleGANs objective is to be able to control the generation of an image using a particular predefined style. How does this do it? It automatically learns unsupervised separation of high-level attributes, could be like, could be examples could be pose and identity for face images. Stochastic variation such as hair could have a lot of randomnesses and scale specific control of attributes.

(Refer Slide Time: 17:38)



How StyleGAN works: Intuition

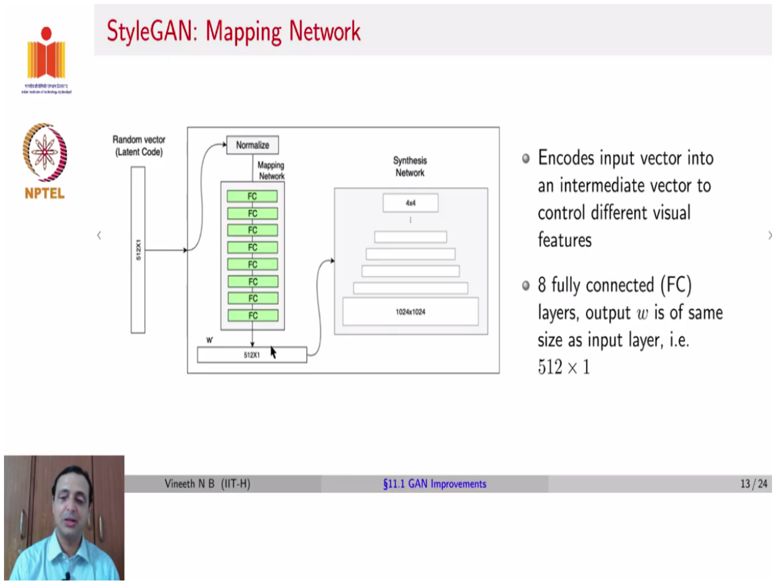
- **Coarse** resolution of up to 82 - affects pose, general hair style, face shape, etc
- **Middle** resolution of 162 to 322 - affects finer facial features, hair style, eyes open/closed, etc
- **Fine** resolution of 642 to 10242 - affects color scheme (eye, hair and skin) and micro features

Vineeth N B (IIT-H) §11.1 GAN Improvements 12 / 24

The intuition for StyleGAN is that a Coarse-resolution in an image, if we consider face images, could affect attributes such as pose, a general hairstyle, face shape, etc. Suppose you go to the next level of resolution, anywhere between 162 and 322. In that case, this resolution throws in final attributes such as facial features, more specific hairstyle, eyes being open, closed, so on and so forth.

At the highest resolution, a Fine resolution anywhere between 642 and 10242 affects the colour scheme, both for eye hair and skin, and introduces micro features on the face. StyleGAN tries to address and introduce some style-specific components at each resolution to attain its desired effect.

(Refer Slide Time: 18:43)

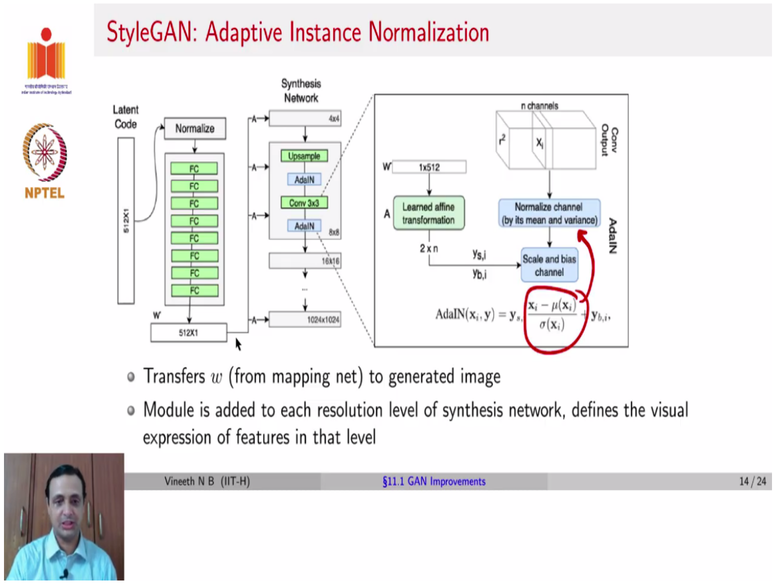

StyleGAN: Mapping Network

- Encodes input vector into an intermediate vector to control different visual features
- 8 fully connected (FC) layers, output w is of same size as input layer, i.e. 512×1

Vineeth N B (IIT-H) §11.1 GAN Improvements 13 / 24

How does it do this? Given a Random input vector, the same as in the Vanilla GAN, StyleGAN first normalizes this vector and then sends this through eight fully connected layers without changing the input dimension and obtaining a 512 * 1 vector. Remember, the input latent was also 512 * 1. So, these eight fully connected layers do not change the dimension of the input. And this transformed vector which we call w, is given as input to the generator or the Synthesis Network. Now, let us see how this vector w affects different resolutions in the generation.

(Refer Slide Time: 19:35)


StyleGAN: Adaptive Instance Normalization

$$\text{AdaIN}(x_i, y) = y_{s,i}\frac{x_i - \mu(x_i)}{\sigma(x_i)} + y_{b,i}$$

- Transfers w (from mapping net) to generated image
- Module is added to each resolution level of synthesis network, defines the visual expression of features in that level

Vineeth N B (IIT-H) §11.1 GAN Improvements 14 / 24

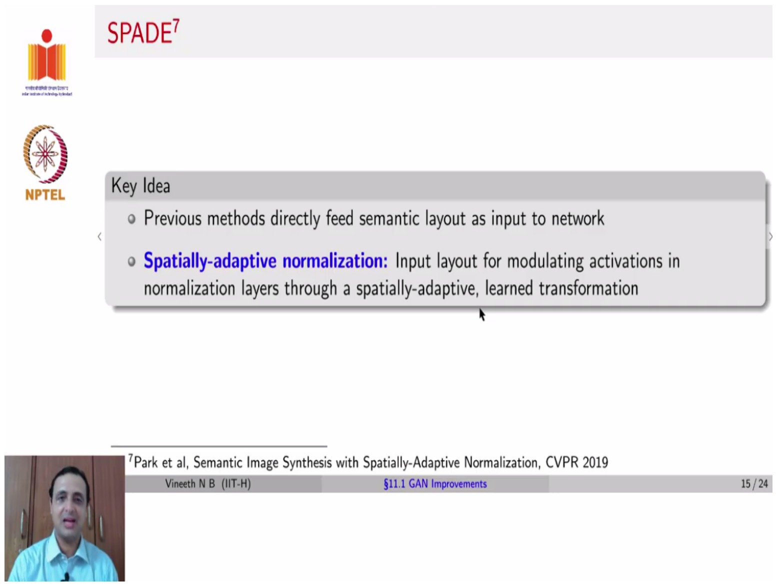
This is achieved by introducing a matrix A, which learns an affine transformation at different resolutions. So, you can see here that the Synthesis Network or the generator in StyleGAN has an Upsampling module and then has something called Adaptive Instance Normalization, which we will see in a minute. A convolution layer follows it, and then another Adaptive Instance Normalization Layer.

And this is then repeated over multiple blocks. Let us take a look at one of these combinations of convolutional and Adaptive Instance Normalization layers. So, if you have a convolutional layer, as seen on the top. In that case, a specific channel of the feature maps in that convolutional layer is normalized by its mean and variance to get an Adaptive Instance Normalization.

In conclusion, the weight vector w is given as an input to each of these blocks. It is then transformed by an affine map, which gives us a set of values $y_{s,i}$ and $y_{b,i}$, which is used to change the scale and bias of the output of the convolutional layer. This is where Adaptive Normalization comes into play. So, you can see here that this now becomes $x_i - \mu(x_i)$, which is normalization with respect to mean and variance.

So, this quantity here on the inside corresponds to subtraction by mean and division by variance. It multiplies them by $y_{s,i}$, which is a scaling value obtained as an output of the affine transformation A and then biased by $y_{b,i}$, which is also the output of the affine transformation. Note that these values, $y_{s,i}$ and $y_{b,i}$ could be different in different blocks of the Synthesis Network. And each such transformation thus defines the visual expression of features in that level. And that allows the input latent to have a different influence at different resolutions of generation.

(Refer Slide Time: 22:19)



SPADE[7]

Key Idea

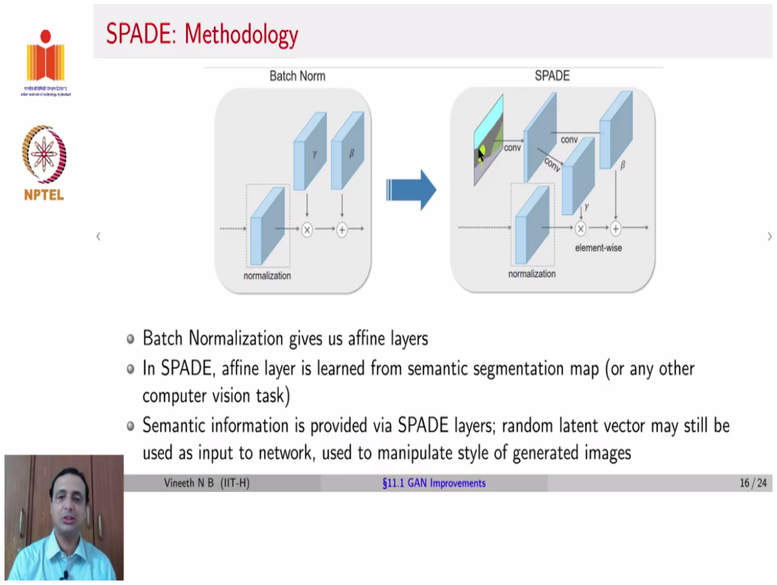
- Previous methods directly feed semantic layout as input to network
- **Spatially-adaptive normalization:** Input layout for modulating activations in normalization layers through a spatially-adaptive, learned transformation

[7]Park et al, Semantic Image Synthesis with Spatially-Adaptive Normalization, CVPR 2019

Vineeth N B (IIT-H) §11.1 GAN Improvements 15 / 24

Another method, the recent one, published in CVPR 2019 again, is known as SPADE. We will see its expansion soon. And the key idea of SPADE is that previous methods directly feed the semantic layout as input to the network. You have a certain image and the entire semantic layout with the pixel configuration of the image. And in SPADE, which is given by Spatially Adaptive Normalization. The input layout for modulating activations in normalization layers happens through this spatially Adaptive Learned Transformation.

(Refer Slide Time: 23:08)



SPADE: Methodology

- Batch Normalization gives us affine layers
- In SPADE, affine layer is learned from semantic segmentation map (or any other computer vision task)
- Semantic information is provided via SPADE layers; random latent vector may still be used as input to network, used to manipulate style of generated images

Vineeth N B (IIT-H) §11.1 GAN Improvements 16 / 24

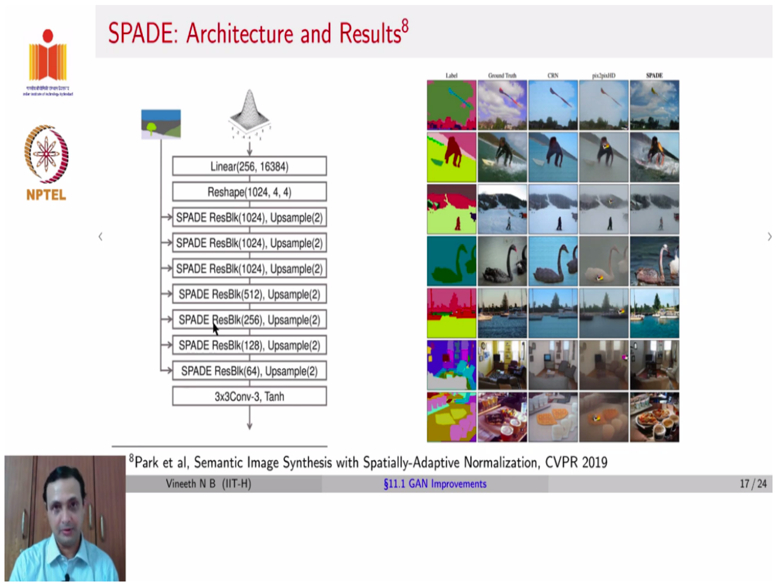
Let us see how this happens. You could look at the output as going through an affine transformation in the standard Batch Normalization layer. You scale up the value, and you add a constant, which is an affine transformation. This is the standard batch norm operation, where you have a value, you multiply it by gamma and add a beta which together defines an affine transformation. However, in SPADE, a semantic segmentation map is given as input.

This semantic segmentation map goes through a convolutional layer, and the convolutional layer outputs a gamma and a beta used to normalize the previous layer of the generator. So, as you can see here, unlike blindly normalizing the output feature map in a specific layer, in this case, the normalization is done in a spatially adaptive manner, where the gamma and beta come from the spatial relationships in this Semantic Segmentation Map.

What is the Semantic Segmentation Map? Recall that a Semantic Segmentation Map has a pixel-wise association of a class label. So, you can see here in the sample, you have a tree, you have a sky, you have a mountain, you have grass, and you have a road, perhaps. That is used to define the gamma and beta for normalization. A random latent vector can also be used to manipulate the style of generated images.

But the semantic segmentation map gives a way of normalizing using spatial content at each pixel. So, you can notice here that the normalization now is defined at each pixel. These gammas and betas are defined at the pixel-wise level, denoted by the cross and plus, which are done element-wise, which can allow each pixel to be normalised differently, based on the Semantic Segmentation mask.

(Refer Slide Time: 25:38)



SPADE: Architecture and Results[8]

Linear(256, 16384)
Reshape(1024, 4, 4)
SPADE ResBlk(1024), Upsample(2)
SPADE ResBlk(1024), Upsample(2)
SPADE ResBlk(1024), Upsample(2)
SPADE ResBlk(512), Upsample(2)
SPADE ResBlk(256), Upsample(2)
SPADE ResBlk(128), Upsample(2)
SPADE ResBlk(64), Upsample(2)
3x3Conv-3, Tanh
Label
Ground Truth
CRN
pix2pixHD
SPADE

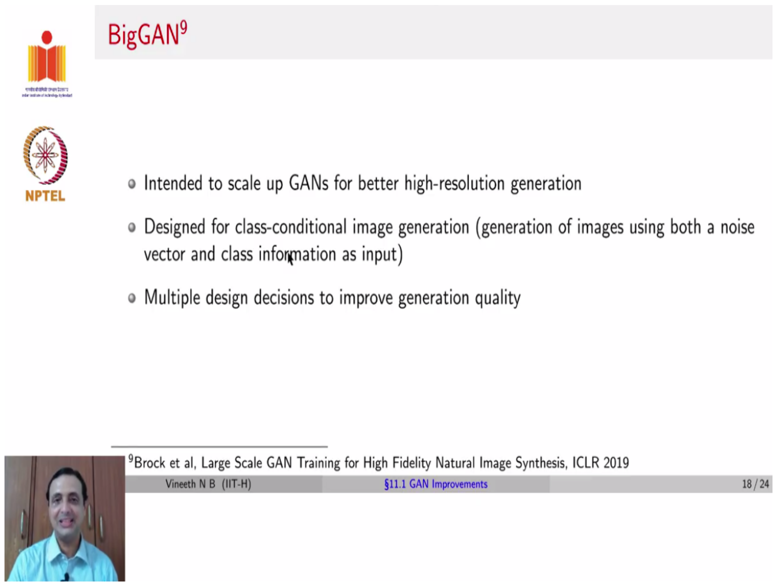
[8]Park et al, Semantic Image Synthesis with Spatially-Adaptive Normalization, CVPR 2019

Vineeth N B (IIT-H) §11.1 GAN Improvements 17 / 24

Here are some interesting results. So, you can see here that the Semantic Segmentation output is given as the input to SPADE. So, these are masks of Semantic Segmentation, Pixel-wise Class Labeling, and this is the actual ground truth image corresponding to these Semantic Segmentation masks. You can see here, while the third and fourth columns correspond to other methods, SPADE gives a fairly photorealistic output close to the ground truth.

On the left side is the architecture of the generator in SPADE. It is similar to many other GAN architectures, but the input of the semantic segmentation mask is coming in at each layer.

(Refer Slide Time: 26:35)



BigGAN[9]

- Intended to scale up GANs for better high-resolution generation
- Designed for class-conditional image generation (generation of images using both a noise vector and class information as input)
- Multiple design decisions to improve generation quality

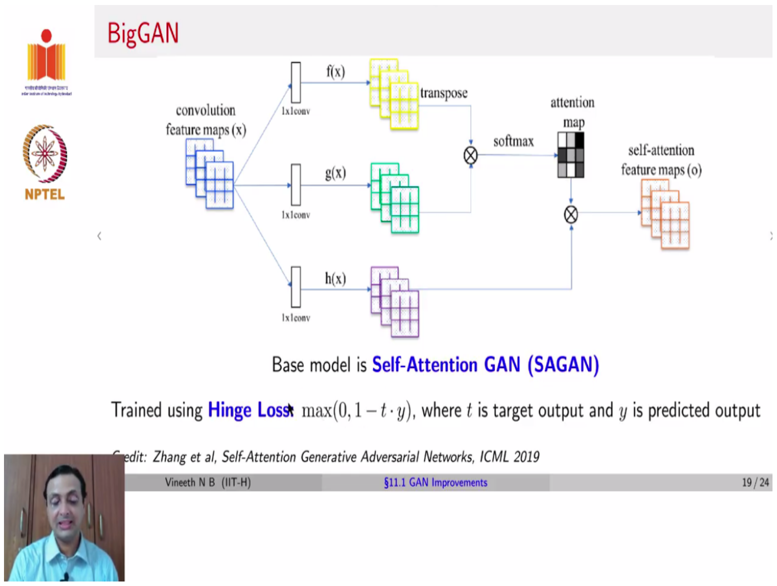
[9]Brock et al, Large Scale GAN Training for High Fidelity Natural Image Synthesis, ICLR 2019

Vineeth N B (IIT-H) §11.1 GAN Improvements 18 / 24

Finally, the last method we will discuss in this lecture is a very popularly used one these days, known as BigGAN, published in ICLR 2019. The focus of BigGAN was to scale up GANs for a better high-resolution generation. It was designed for class conditional image generation, which means the input is both a noise vector, similar to a Vanilla GAN and some class information.

For instance, it could be one hot vector, which together is given to the GAN to generate an image corresponding to that class. BigGAN also introduces a few different design decisions, as we will see in the following few slides.

(Refer Slide Time: 27:28)



BigGAN

Base model is **Self-Attention GAN (SAGAN)**

Trained using **Hinge Loss**: $\max(0, 1 - t \cdot y)$, where t is target output and y is predicted output

Credit: Zhang et al, Self-Attention Generative Adversarial Networks, ICML 2019

Vineeth N B (IIT-H) §11.1 GAN Improvements 19 / 24

An important design decision in BigGAN is to use the idea of a Self-Attention GAN or a SAGAN, an earlier work introduced in late 2018 and published in ICML 2019, which introduces Self-Attention. We already saw Self-Attention in Transformers a week ago. It is the same idea here, where you have a set of convolutional maps as the output of a particular layer in the generator.

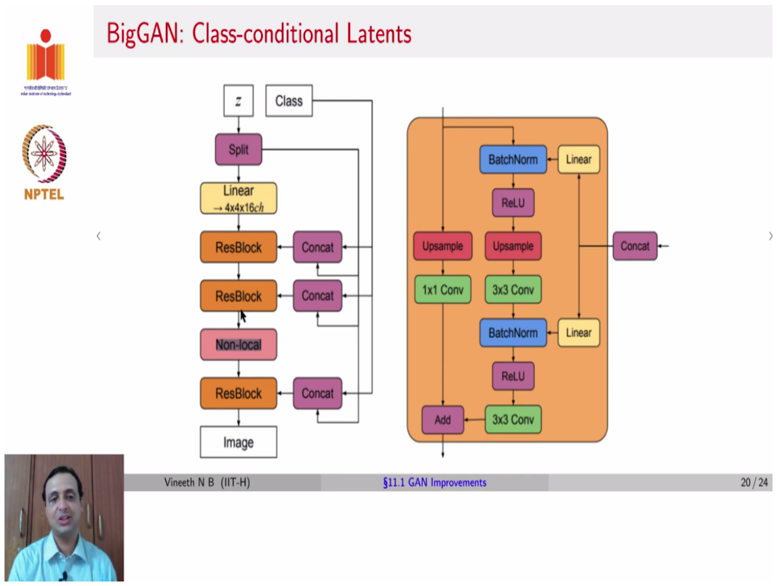
This output goes through multiple 1 * 1convolutions. Some of which transform and transpose to obtain an Attention map. These three branches are very similar to the query, key and value of the transformer architecture. You could consider them to be similar. So, two of those branches generate an Attention map, which is used to focus on a specific part of the convolutional map to generate that component of the image in the next layer. Why is this required?

Self-Attention GANs were introduced because often GANs, while they generated crisp images, would miss out on finer details. For example, in a Dog's image, they may generate the fur on the dog but miss out on the legs of a dog. So, the idea of using Self-Attention is to focus on specific parts of the image and generate every local detail more appropriately. In BigGAN, along with Self-Attention GAN, Hinge Loss is used.

This Hinge Loss is similar to the Hinge Loss used in support vector machines, given by $max\ (0,\ 1 -\ t.y)$, where t is the target output, and y is the predicted output. This is because

BigGAN is class conditional. So, you also want the output image to belong to a particular class, which is then used at the discriminator stage. Because the discriminator not only says real or fake but also gives a class label whose loss, in this case, is given by Hinge Loss.

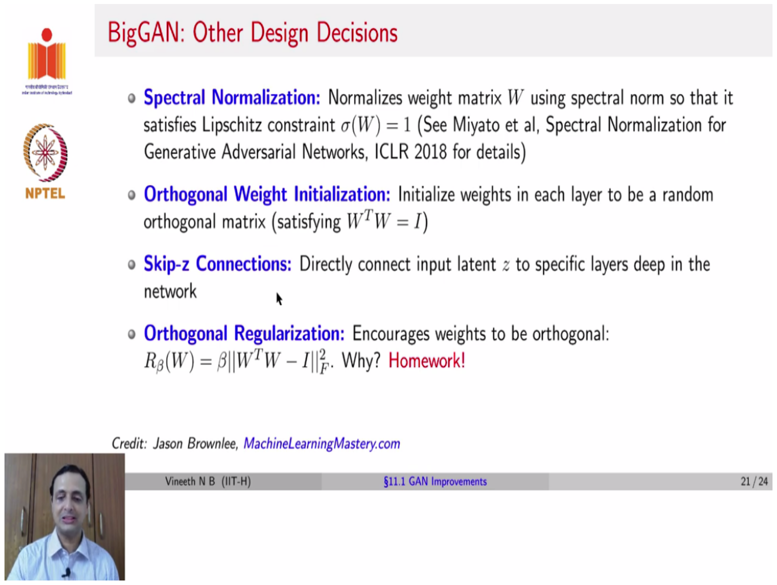
(Refer Slide Time: 30:04)


BigGAN: Class-conditional Latents

In addition, BigGAN also introduces Class-conditional Latents in a slightly different way. Instead of a Class-conditional Latent, which could be a one-hot vector. Instead of giving that directly to the generator's input, the Class-conditional Latent is given as a separate input at multiple stages of generation. It is concatenated with a certain input and then given to each of the Residual blocks. What is done inside each of these Residual blocks?

The concatenated class label vector passes through two different linear transformations given to the two batch norm layers inside each residual block.

(Refer Slide Time: 31:02)



BigGAN: Other Design Decisions

- **Spectral Normalization:** Normalizes weight matrix W using spectral norm so that it satisfies Lipschitz constraint $\sigma(W) = 1$ (See Miyato et al, Spectral Normalization for Generative Adversarial Networks, ICLR 2018 for details)
- **Orthogonal Weight Initialization:** Initialize weights in each layer to be a random orthogonal matrix (satisfying $W^TW = I$)
- **Skip-z Connections:** Directly connect input latent z to specific layers deep in the network
- **Orthogonal Regularization:** Encourages weights to be orthogonal: $R_\beta(W) = \beta||W^TW - I||_F^2$. Why? Homework!

Credit: Jason Brownlee, MachineLearningMastery.com

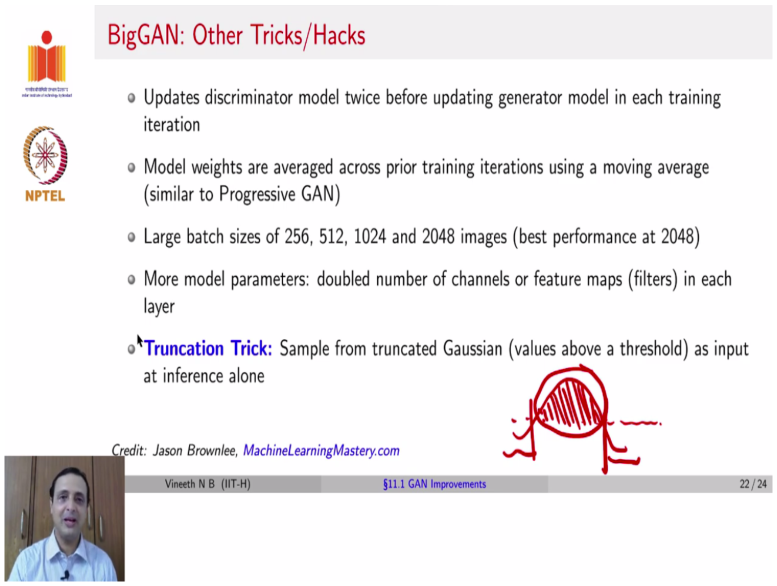
Vineeth N B (IIT-H) §11.1 GAN Improvements 21 / 24

BigGAN also had Other Design Decisions, which helped its performance. One such was Spectral Normalization, where the weight matrix in each layer was normalised to ensure that its spectral norm was maintained. And it also satisfied the Lipschitz constraint, $\sigma(W) = 1$. You can see this paper called Spectral Normalization for GANs ICLR 2018 for more details. The broad idea of such a step is similar to Batch normalisation to constrain the weights with specific properties that ensure that learning can be better and faster.

In this case, we try to constrain the highest singular value of the weight matrix. A second idea is Orthogonal Weight Initialization, a well-known weight initialisation method. Each layer's weights are initialised to ensure $W^TW = I$ or the weight matrix is orthogonal. It also introduced Skip-z Connections, where the latent input z is connected to specific layers deep in the network.

It also introduces another method known as Orthogonal Regularization, which encourages the weights to be orthogonal in each training iteration. This is done using a regulariser$R_\beta$where beta is a coefficient. The regulariser tries to ensure that the Frobenius norm between W^TW and identity $I$is minimised. So, this would be a way to ensure the W^TW is close to identity I, and hence the matrix is Orthogonal. Why would we want Orthogonal Regularization? Think about it; this is another homework for you for this lecture.

(Refer Slide Time: 33:16)



BigGAN: Other Tricks/Hacks

- Updates discriminator model twice before updating generator model in each training iteration
- Model weights are averaged across prior training iterations using a moving average (similar to Progressive GAN)
- Large batch sizes of 256, 512, 1024 and 2048 images (best performance at 2048)
- More model parameters: doubled number of channels or feature maps (filters) in each layer
- **Truncation Trick:** Sample from truncated Gaussian (values above a threshold) as input at inference alone

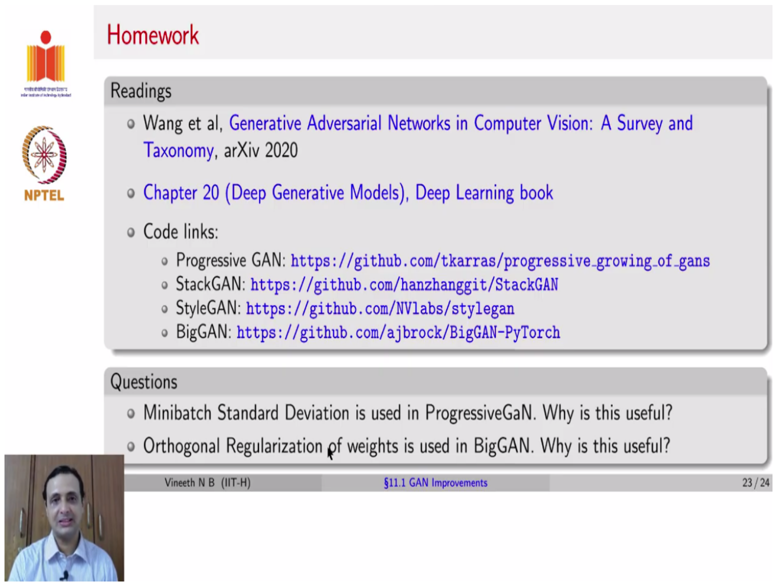
Credit: Jason Brownlee, MachineLearningMastery.com

Vineeth N B (IIT-H) §11.1 GAN Improvements 22 / 24

Other Hacks employed in BigGAN were: 1. They updated the Discriminator model twice before updating the Generator model in each iteration. 2. The model weights were finally averaged across a few training iterations using a moving average approach. Progressive GAN also did this. 3. BigGAN also observed that using very large batch sizes, such as 256, 512, 1024, and 2048. By batch sizes, we mean minibatch sizes while performing Minibatch SGD. They observed the best performance with the largest minibatch size of 2048. 4. They also doubled the model parameters or the number of channels in each layer.

5. They employed a trick known as the Truncation Trick, where the generator initially receives a latent vector from a Gaussian during training time. At test time, you sample a latent from a Gaussian. If that value is less than a threshold, you discard it and sample another value. So, this is called a Truncated Gaussian. You could now imagine that this Truncated Gaussian is something like this, where you are ensuring that something sampled below a threshold is not considered an input to the GAN. So, that way, all inputs to the GAN come from the shaded part of the PDF. The main idea is not to get anomalous generations using these latent vectors, but you get generations that belong to the core part of the PDF of that Gaussian.

(Refer Slide Time: 35:22)



Homework

Readings

- Wang et al, Generative Adversarial Networks in Computer Vision: A Survey and Taxonomy, arXiv 2020
- Chapter 20 (Deep Generative Models), Deep Learning book
- Code links:
  - Progressive GAN: https://github.com/tkarras/progressive_growing_of_gans
  - StackGAN: https://github.com/hanzhanggit/StackGAN
  - StyleGAN: https://github.com/NVlabs/stylegan
  - BigGAN: https://github.com/ajbrock/BigGAN-PyTorch

Questions

- Minibatch Standard Deviation is used in ProgressiveGaN. Why is this useful?
- Orthogonal Regularization of weights is used in BigGAN. Why is this useful?

Vineeth N B (IIT-H) §11.1 GAN Improvements 23 / 24

Your homework for this lecture is to go through this excellent survey of GANs recently released, Chapter 20 of the Deep Learning book. And here are the code links for most of the GANs discussed in this lecture. Suppose you would like to see them and try them out. We left behind the questions: Why is Minibatch standard deviation used in Progressive GAN, and Why is Orthogonal Regularization used in BigGAN? Think about it and we will discuss it next time.