

Deep Generative Models

Prepared by Dr. Sherif Mostafa

AI314

Master of Science in Artificial Intelligence

Lecture 7



Progressive GANs

Introduction

ProGan

Progressive training

Minibatch standard deviation

Pixelwise normalization

Introduction

Generative Adversarial Networks (GANs) are a powerful class of machine learning models that learn to generate new data. They are known for their ability to create realistic and diverse data, making them useful for a wide range of applications depending on Generator and Discriminator.

Challenges in GAN training:**Instability**

Traditional GAN training can be unstable leading to poor convergence and low-quality generated images

Mode Collapse

The generator may produce a limited variety of outputs, effectively "collapsing" to a few modes of the data distribution. This leads to a lack of diversity in generated samples.

Challenges in GAN training:**Slow Convergence**

GAN training is often unstable, and the loss functions may oscillate without converging to a stable solution. This makes it difficult to evaluate performance and can lead to poor-quality outputs.

Vanishing Gradients

If the discriminator becomes too powerful compared to the generator, the generator receives minimal feedback. This hampers the generator's ability to learn effectively and can stall the training process.

Challenges in GAN training:**Hyperparameter Sensitivity**

GANs are sensitive to hyperparameters such as learning rates, batch sizes, and architectures. Finding the right set of hyperparameters can be time-consuming and often requires extensive experimentation.

Long Training Times

Training GANs can be computationally expensive and time-consuming. This can limit experimentation and the ability to iterate quickly on model designs.

Progressive GANs

Progressive Growing of GANs (ProGAN) is a technique that addresses some of the challenges associated with training traditional GANs, particularly in generating high-resolution images. ProGANs built upon the principles of Progressive Growing of GANs while introducing more generalized structures and training methodologies. Here's a detailed look at the architecture and training process:

Progressive GANs Architecture

The architecture consists of two main components:

Generator & Discriminator

Both components grow in complexity and size as training progresses.

- **Input Layer:** Starts with a low-dimensional latent space (e.g., 512-dimensional noise vector).
- **Progressive Layers:**
 - **Initial Stage:** Begins with a simple architecture that generates low-resolution images (e.g., 4x4).
 - **Layer Expansion:** As training progresses, new layers are added to increase the resolution (e.g., 8x8, 16x16, up to 1024x1024).
 - **Blending Mechanism:** Each new layer is blended with the existing layers using a mixing coefficient, allowing for smooth transitions between resolutions.
- **Activation Functions:** Uses Leaky ReLU for hidden layers and Tanh for the output layer to produce images in the range of $[-1, 1]$.

- **Symmetrical Layers:** Mirrors the generator architecture, starting from a low-resolution input and progressively increasing complexity.
- **Progressive Addition:** Similar to the generator, new layers are added at each stage, allowing the discriminator to learn to distinguish between real and generated images at increasing resolutions.
- **Output Layer:** Produces a single scalar value indicating the probability that the input image is real.

Progressive training

The training process for ProGANs follows a structured approach that emphasizes gradual growth and stability.

Instead of training the GAN to generate high-resolution images from the start, ProGAN begins with low-resolution images and progressively increases the resolution as training advances. Benefit: This gradual approach allows the model to learn coarse features first and incrementally add details, leading to improved stability and quality.

Step-by-Step Training

1. Initialization:

- Layers are added to the generator and discriminator networks at each stage with the lowest resolution (e.g., 4x4).
- Initialize both the generator and discriminator with their respective layers for the current resolution.
- The generator is initially trained to transform the latent input noise vector into an image of shape 4x4 while the discriminator is trained to transform the input image of size 4x4 into a single scalar prediction

Progressive training

2. Training Loop:

- For each iteration:
 - **Update Generator:**
 - Generate fake images.
 - Compute the generator's loss based on the discriminator's feedback (the goal is to fool the discriminator).
 - Update the generator parameters.

Progressive training

2. Training Loop:

- For each iteration:
 - **Update Discriminator:**
 - Sample real images from the dataset.
 - Generate fake images using the generator.
 - Compute the discriminator's loss using a binary cross-entropy loss function.
 - Update the discriminator parameters based on this loss.

3. Resolution Increase:

- After a set number of epochs or once the model stabilizes at the current resolution:
 - Gradually add new layers to both the generator and discriminator.
 - Introduce higher resolution images (e.g., moving from 4x4 to 8x8).
 - Blend the output of the new layers with the existing ones to ensure continuity..

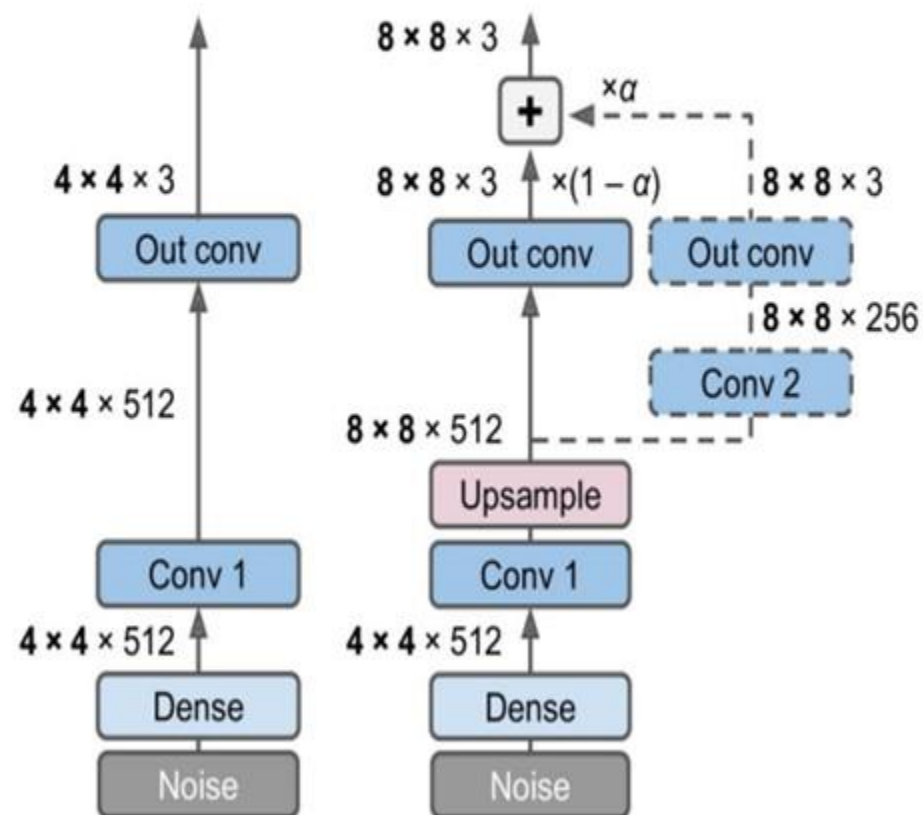
Progressive training

4. Blending Coefficient:

- Use a small value as a blending coefficient α that gradually increase from 0 to 1 to control the influence of new layers. For example, at the transition to a higher resolution, the output might be a weighted combination of the old and new outputs.
- When transitioning to a higher resolution, the previous layers are blended with new layers. This ensures smooth integration and helps maintain learned features. So, the new layers are introduced gradually, with a blending coefficient controlling the mix of new and old layers.

Progressive training

- generating small images at the beginning of training, then gradually adding convolutional layers to both the generator and the discriminator to produce larger and larger images (4×4 , 8×8 , 16×16 , ..., 512×512 , $1,024 \times 1,024$).
- This approach resembles greedy layer-wise training of stacked autoencoders.
- The extra layers get added at the end of the generator and at the beginning of the discriminator, and previously trained layers remain trainable.

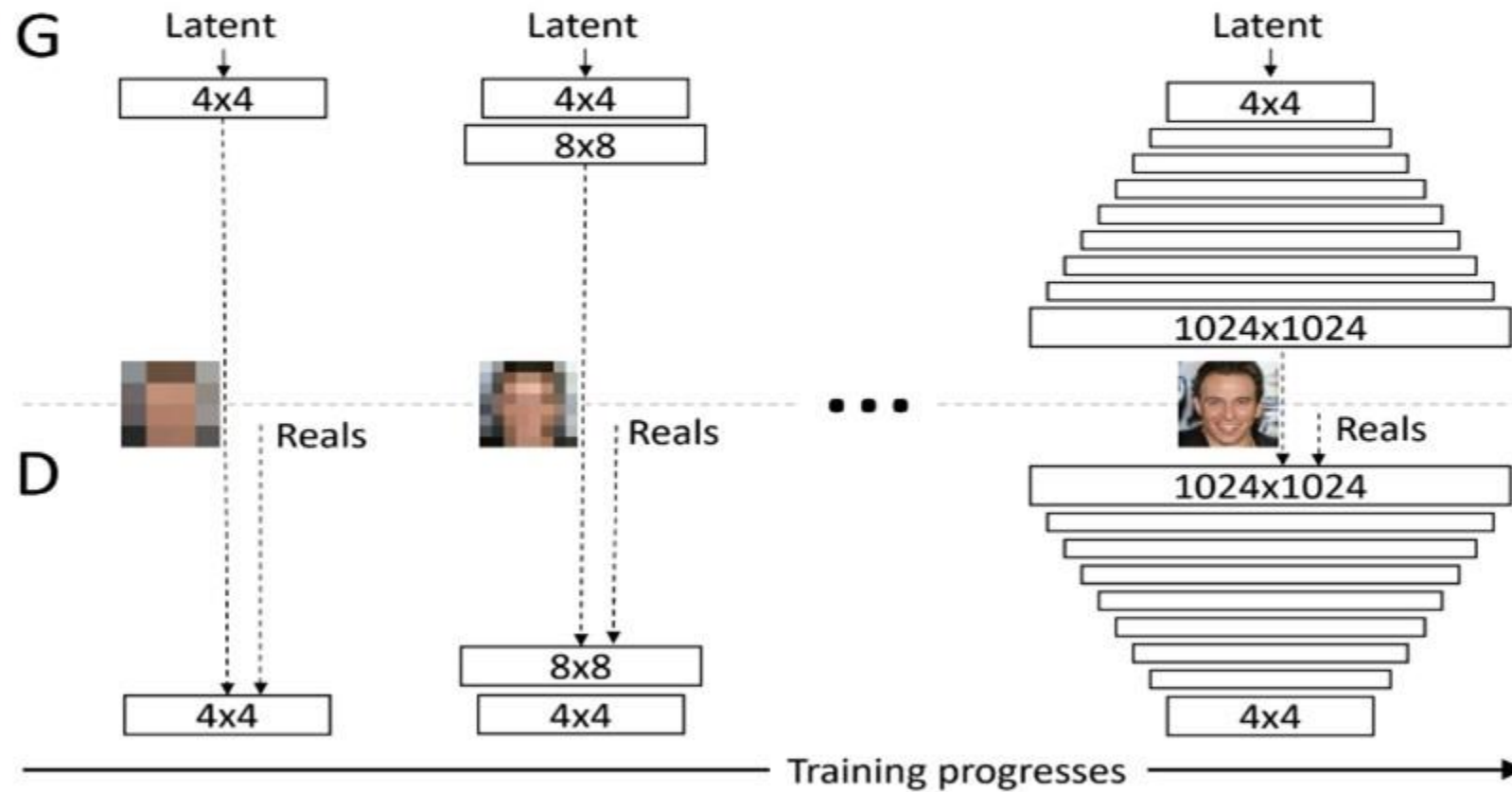


Progressive training

5. Repeat:

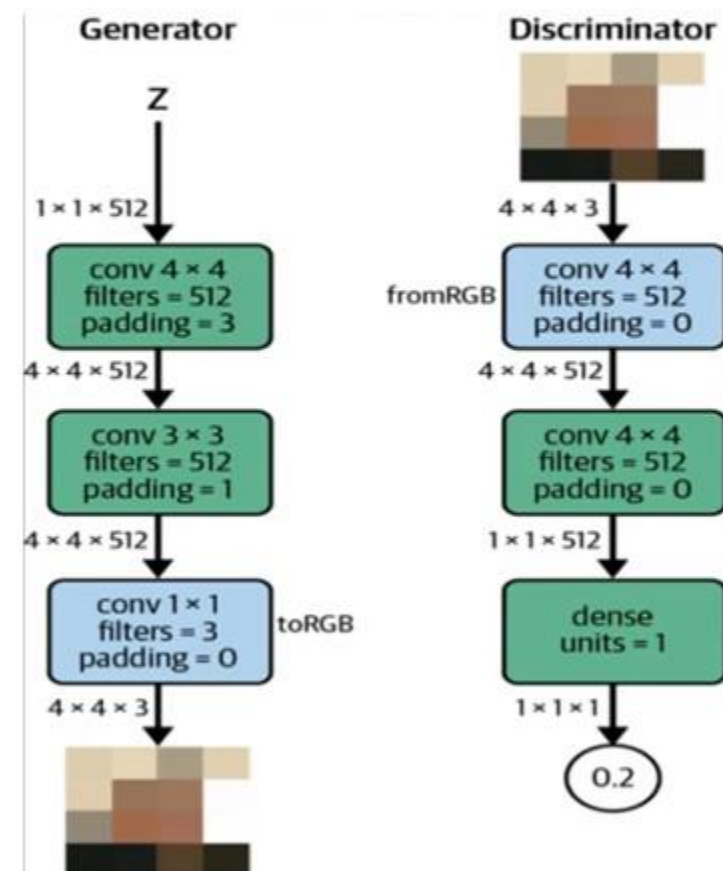
- Continue the training process, progressively increasing the resolution until reaching the target resolution (e.g., 1024x1024).
- So, The network first learns and understands the working of a 4 x 4 image. Once that is completed, it proceed to teach it 8 x 8 images, 16 x 16 resolutions. During each stage, the model trains until convergence before moving to the next resolution level

Progressive GANs Challenges



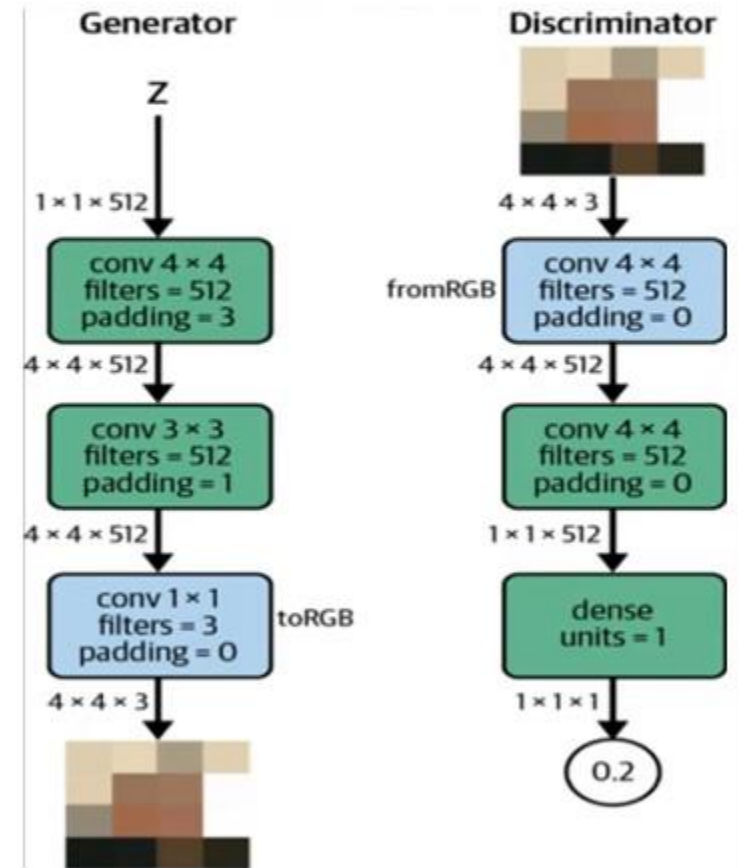
Gradual Transition:

- The new layers in both the generator and discriminator are initially masked using a parameter α that is gradually increased from 0 to 1 throughout the transition phase. This allow for smooth transition as the new layer take over.



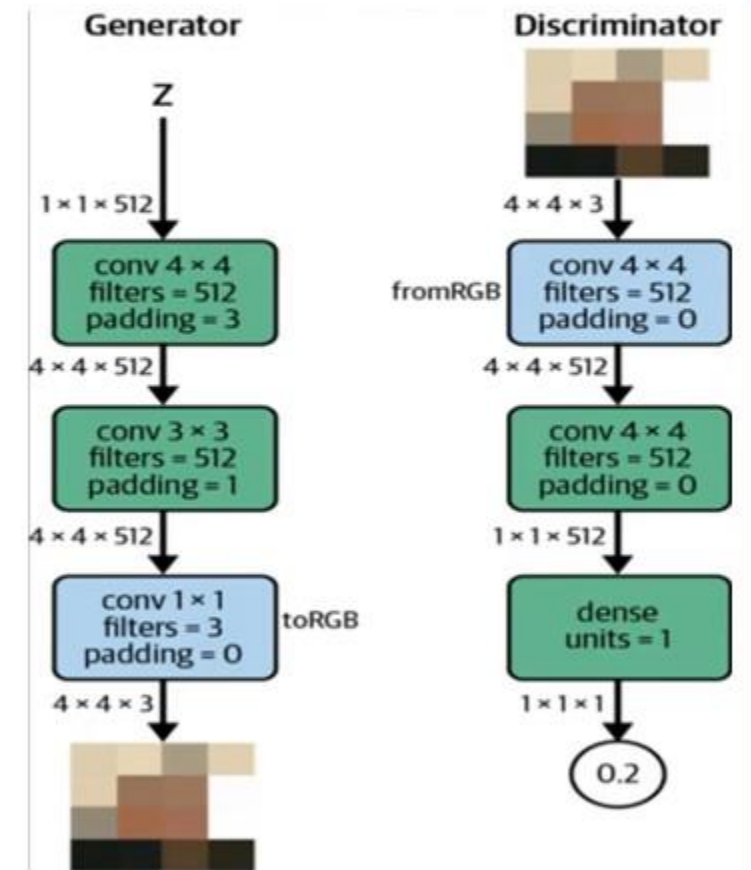
Generator:

- Generator architecture starts with a convolutional layer that converts the set of feature maps into an RGB image (toRGB). As the resolution increases, new upsampling and convolutional layers are appended to the existing network with a residual connection to maintain the output from the existing trained toRGB layer

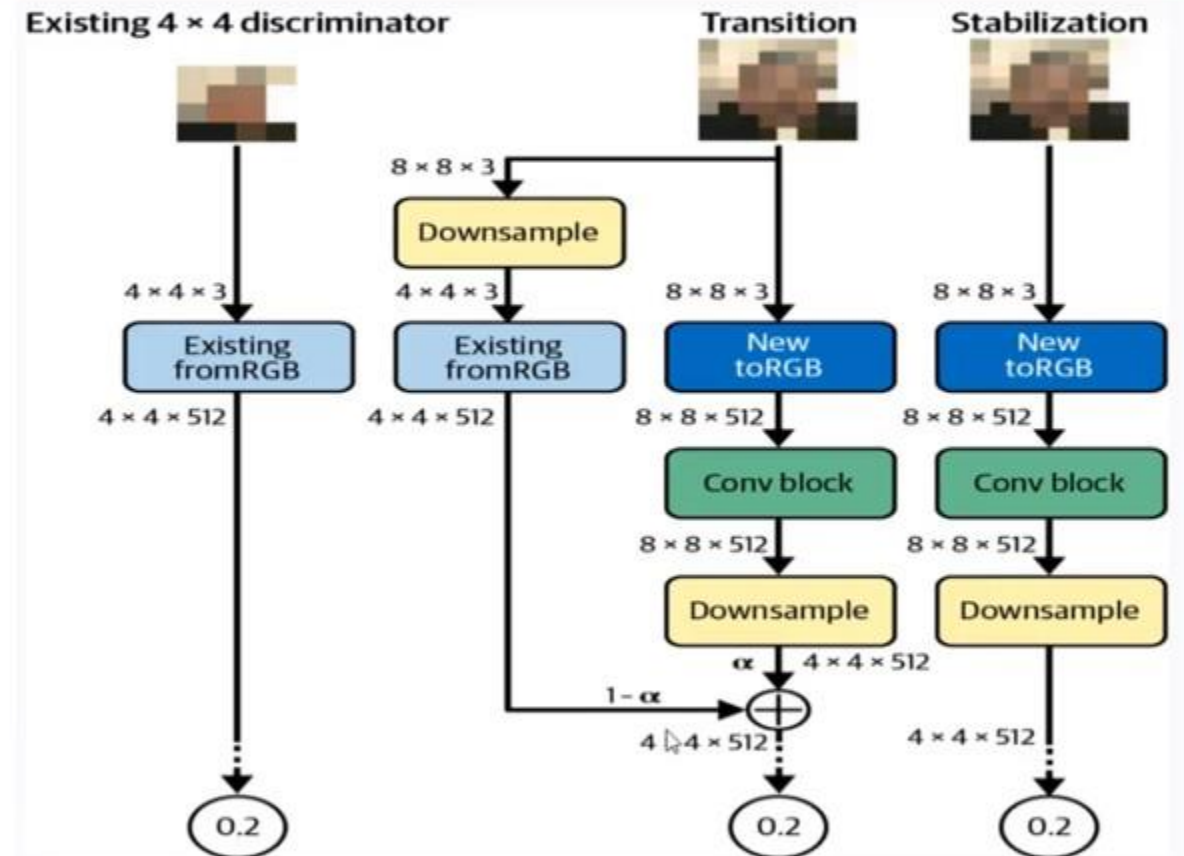


Discriminator:

- Discriminator architecture starts with a convolutional layer that converts the RGB image into a set of feature maps (fromRGB). As the resolution increases, new convolutional layers are added to the existing network the higher resolution inputs.



The training process for the ProGAN Discriminator training process, expanding the network from 4x4 images to 8x8



Progressive GANs Challenges

Training Stability Techniques

- **Adaptive Learning Rates:** Utilize techniques like Adam optimizer with adaptive learning rates for better convergence.
- **Regularization:** Apply techniques such as spectral normalization to stabilize the discriminator and prevent overfitting.
- **Label Smoothing:** Use label smoothing in the discriminator to make it less confident in its predictions, aiding in stability.

Progressive GANs Advantages

- **Improved Stability:**

By starting with simpler tasks, the training process is more stable, reducing issues like mode collapse and non-convergence.

- **High-Quality Outputs:**

ProGANs are capable of generating high-resolution images with fine details, outperforming traditional GANs in many scenarios.

- **Efficient Learning:**

The model learns to generate large images more effectively by focusing on low-level features first before tackling high-level details.

Progressive GANs Applications

- **Image & Video Generation:** ProGANs are widely used in generating realistic images and videos in domains like portrait creation, scenery generation, and even artwork.
- **Transfer Learning:** The progressive training approach can be beneficial for transfer learning tasks, where pre-trained models can be adapted to new datasets.
- **Super Resolution:** The progressive training technique can be applied to image super resolution, upscaling low resolution images to high quality resolution output.

Progressive GANs Challenges

- **Complexity of Implementation:** The progressive training strategy adds complexity to the implementation and requires careful tuning of hyperparameters.
- **Training Time:** While the gradual approach improves stability, it can also increase overall training time compared to training a GAN from scratch.
- **Increased Resource Requirements:**
As the model increases in size and complexity with higher resolutions, it demands more computational resources and memory, which can be a constraint, especially on limited hardware.

Progressive GANs Challenges

- **Coordination Between Generator and Discriminator**

Achieving a balance between the generator and discriminator can be challenging, as one may overpower the other, leading to suboptimal training dynamics.

- **Data Limitations**

The effectiveness of Progressive GANs heavily relies on the quality and diversity of the training dataset. Limited or biased datasets can lead to poor generalization.

Comparison to traditional GAN training

Traditional Gan:

Trains the generator and discriminator on full resolution images from the start leading to instability and slow convergence.

ProGAN:

Stars with low resolution images and gradually increase the model complexity and resolution resulting in faster training and higher quality outputs.

ProGANs progressive approach address the core challenges of traditional GAN training enabling more robust and efficient learning.

Comparison to traditional GAN training

Traditional GANs	ProGANs
Slow Convergence	Faster Convergence
Unstable Convergence	stable Convergence
Limited resolution	High resolution
High memory usage	Low memory usage

Minibatch Standard Deviation Layer

- Minibatch standard deviation layer added near the end of the discriminator. For each position in the inputs, it computes the standard deviation across all channels and all instances in the batch.
- These standard deviations are then averaged across all points to get a single value.
- Finally, an extra feature map is added to each instance in the batch and filled with the computed value.

Minibatch Standard Deviation Layer

How does this help? If the generator generates images with limited variability, the discriminator will have a low standard deviation across feature maps. Due to Mini-Batch layer, the discriminator will have easy access to this statistic, making it less likely to be fooled by a generator that creates insufficient diversity. This encourages the generator to produce more diversified outputs, reducing the likelihood of mode collapse.

How Minibatch Standard Deviation Layer Works!!

Need of Mini Batch Standard Deviation

Mode collapse happens when the generator can only produce a single type of output or a small set of outputs. This may happen due to problems in training, such as the generator finds a type of data that is easily able to fool the discriminator and thus keeps generating that one type. In ProGAN added 1 feature map at the end of discriminator to append the standard deviation of the feature map values averaged across all pixels to address the issue of mode collapse. So, 1 feature map will be added to every image of the mini batch.

How It Works

1.Input: The layer receives a batch of feature maps from the previous layer. This could be the output of a convolutional layer in a GAN's generator.

2.Compute Mean and Standard Deviation:

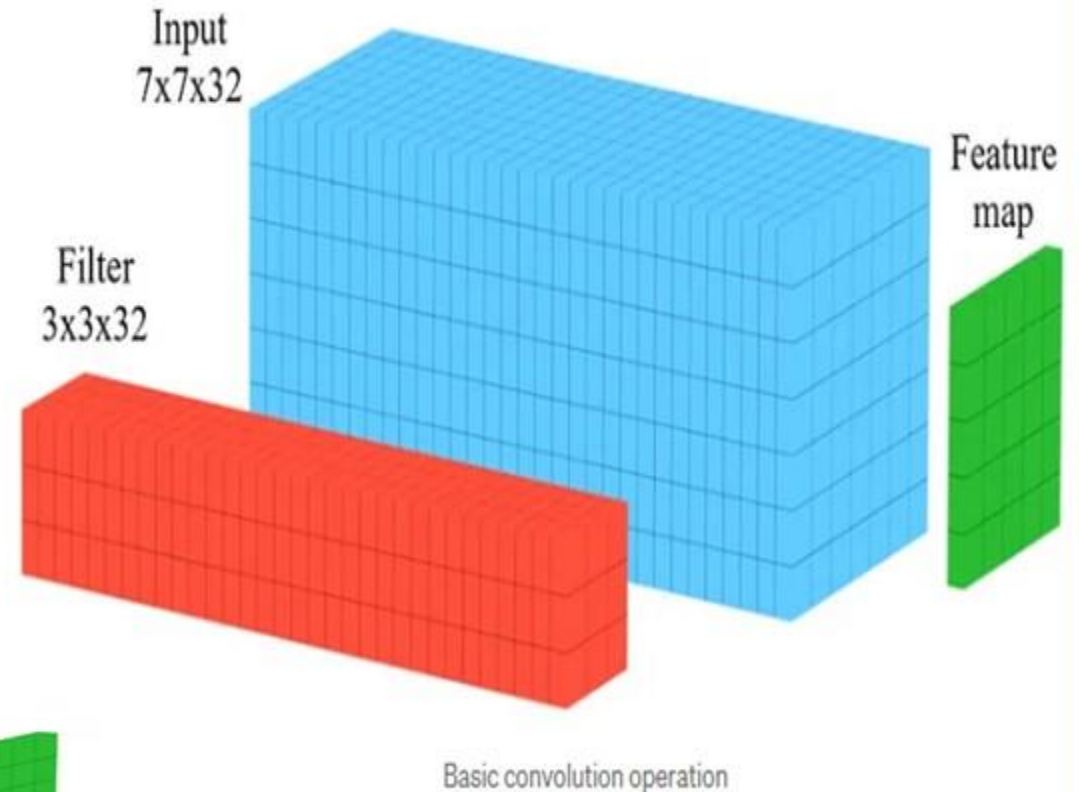
- 1. Mean:** Calculate the mean of the feature maps across the batch.
- 2. Variance:** Compute the variance by finding the squared differences from the mean, then average these differences.
- 3. Standard Deviation:** Take the square root of the variance to get the standard deviation.

3.Concatenate: The computed standard deviation is then concatenated to the original feature maps. This gives the network additional information about the variability within the batch.

4.Output: The output of the Minibatch Standard Deviation Layer is a modified version of the input feature maps that includes the standard deviation information.

How Minibatch Standard Deviation Layer Works!!

Let's consider, we have input of $7 \times 7 \times 32$ and we are applying $3 \times 3 \times 32$ filter to input, which will produce 1 feature map as shown in above Figure 1. But when we apply convolution operation to input, we are not applying only 1 filter. Suppose we have to chosen to apply c filter. So, resulting feature map will have shape of $4 \times 4 \times c$. And in the mini-batch, we do not have only 1 input. Suppose we have m inputs. So, resulting feature map will be of shape $m \times 4 \times 4 \times c$. The below image is only for 1 filter.



How Minibatch Standard Deviation Layer Works!!

Now, standard deviation for each spatial location is calculated using below formula for every filter over whole mini-batch. First, we will calculate the mean for each spatial location. So, we will get total 16 mean (each corresponds to 1 spatial location, and we have total $4 \times 4 = 16$ spatial locations). The same way using mean, we will calculate standard deviation for every 16 spatial location, and it will create standard deviation map

$$mean_{rc} = \frac{1}{m} \sum_{l=1}^m map[l, r, c]$$

$$std_{rc} = \sqrt{\frac{1}{m} \sum_{l=1}^m (map[l, r, c] - mean_{rc})^2}$$

Where, Here $0 \leq r < \text{height}$ and $0 \leq c < \text{width}$

How Minibatch Standard Deviation Layer Works!!

Previous process we have done it 1st filter. We have to do same for every c filters.



Std. Dev map of
filter 1 maps



Std. Dev map of
filter 2 maps



Std. Dev map of
filter 3 maps

...

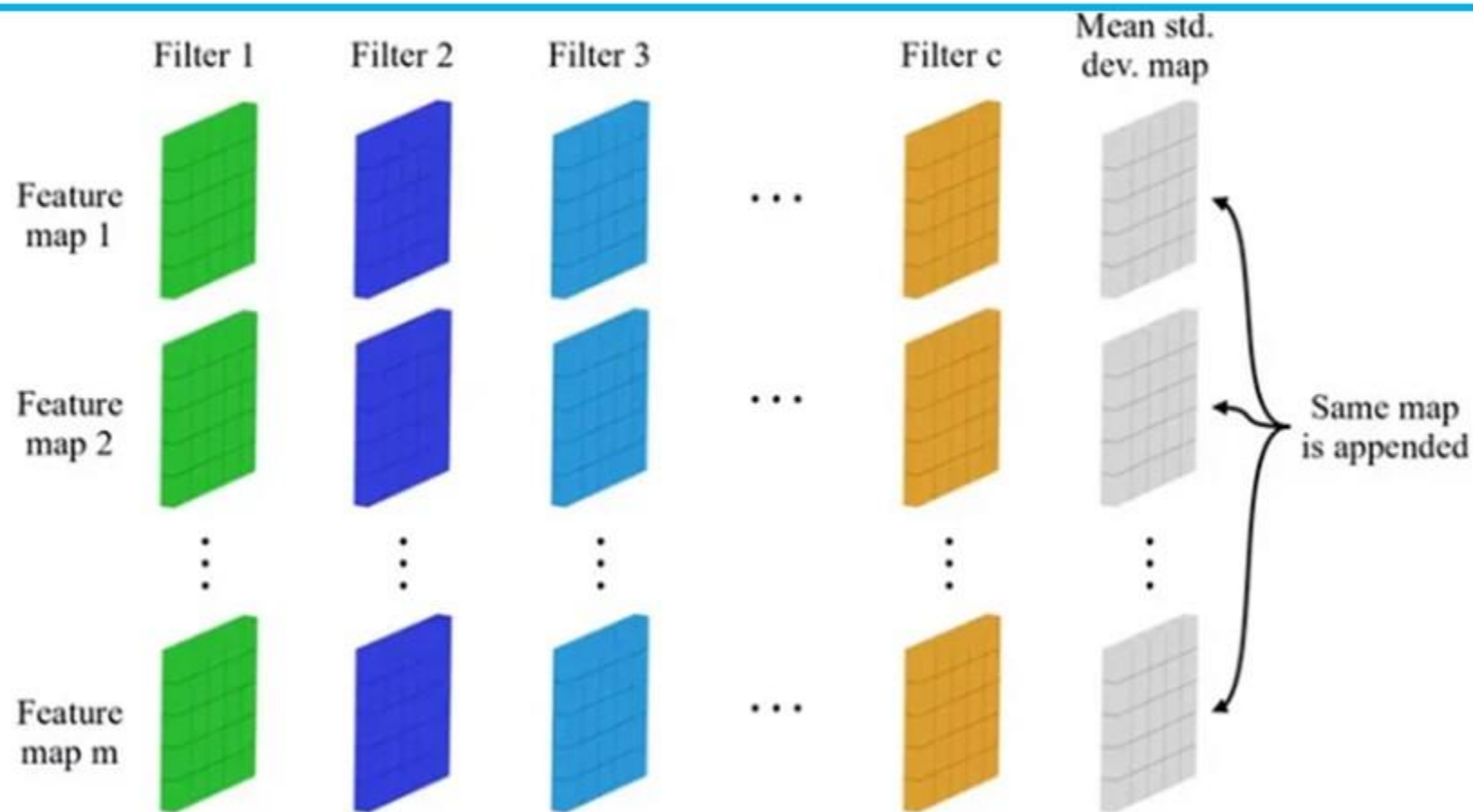


Std. Dev map of
filter c maps

Now, we will calculate scalar mean standard deviation by taking average of above maps over all spatial locations, which will give only 1 scalar value. From this value we will create mean standard deviation map in which all entries will have the same value, which is mean standard deviation of mini-batch.

Now, this mean standard deviation feature map will be appended to each output of previous layer.

How Minibatch Standard Deviation Layer Works!!



Mean std. dev. map appended at end of each output.

How this helps in mode collapse??

The discriminator can then use this additional feature to distinguish between real and fake batches:

Real batch will typically have a higher standard deviation indicating greater variety. While the **fake patches** if they lack variety will have a lower standard deviation.

So, If generator is producing almost similar images in mini-batch because of mode collapse, there will be less variation in pixel value of these images, which eventually leads up to the low value of Mean std. dev. map. Because of this, it will force generator to create images with large variance in its pixel values compare to each other, which will solve the issue of mode collapse.

- **Feature values:**

The minibatch standard deviation layer calculate the standard deviation of the feature values.

- **Averaging:**

These standard deviations are then averaged across all points to get a single value.

- **Additional feature:**

Finally, the average standard deviation is then appended as an extra feature map is added to each instance in the batch and filled with the computed value.

Benefits of Minibatch standard deviation

Increased Realism	Generate more realistic and diverse outputs
Improved Performance	Helps the discriminator better distinguish between real and fake batches
Enhancing training	Incentivizes the generator to learn from the real data

ProGAN employs several innovative techniques to achieve remarkable image generation quality. One of these techniques is the use of Equalized Learning Rates which address the issue of imbalanced training speeds between different layers.

Another key feature is pixelwise normalization, which helps to stabilize the training process and prevent signal explosion.

- Initializes all weights using a simple Gaussian distribution with mean 0 and standard deviation 1 rather than using He initialization (“He” initialization is where the standard deviation is inversely proportional to the square root of the number of inputs to help stabilize the training by setting the initial weights neither too small or too large to prevent vanishing or exploding gradients)
- However, the weights are scaled down at runtime (when every time the layer is executed) by the same factor as in He initialization: they are divided by $\sqrt{\frac{2}{n_{inputs}}}$ where n_{inputs} is the number of inputs to the layer.

- Modern optimizers like RMSProp, Adam, or other adaptive gradient optimizers normalize gradient updates, making them independent of weight scale. However, this can lead to slower adjustments for layers with fewer inputs.
- ProGANs address this imbalance by using equalized learning rates. Weights are initialized with simple Gaussian distribution and normalization is applied dynamically during layer calls. This ensure that the optimizer treats all weights as having similar dynamic ranges resulting in more balanced training.

- **Feature Vector Normalization:**

Pixelwise Normalization in ProGAN's generator normalize the feature vector at each pixel to a unit length. This helps prevent the signal from becoming excessively large as it propagates through the network.

- **Preventing Signal Explosion**

By normalizing the feature vector, pixelwise normalization helps to stabilize the training process and prevent the signal from spiraling out of control. This is crucial for achieving high quality image generation

- **No Trainable Weights:**

The pixelwise normalization layer does not have any trainable weights. This means that it does not introduce additional parameters that need to be learned during training, Keeping the model efficient and focused on learning the underlying data distribution.

- **Signal control:**

Pixelwise Normalization prevent the signal from being excessively large as it propagate through the network. Ensuring that the model remains stable.

- **Improved Gradient flow**

By normalizing the feature vector, pixelwise normalization helps to improve the flow of gradients through the network making it easier for the model to learn underlying data distribution

- **Signal control:**

Pixelwise Normalization improve image quality through stabilizing the training process and prevent signal explosion in the progressive training.



Thank You