شبکه‌های مولد تخاصمی
Generative Adversarial Networks (GAN)

Alireza Akhavanpour

Akhavanpour.ir
CLASS.VISION

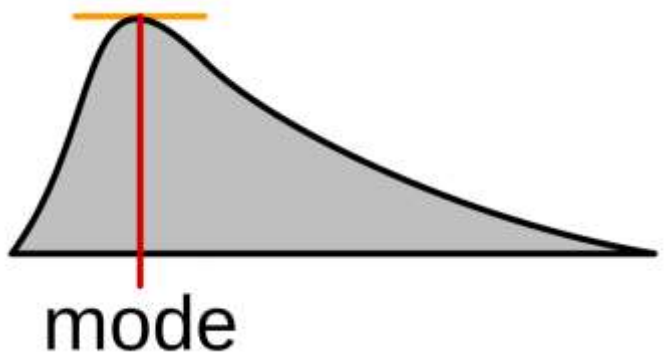
Mode Collapse

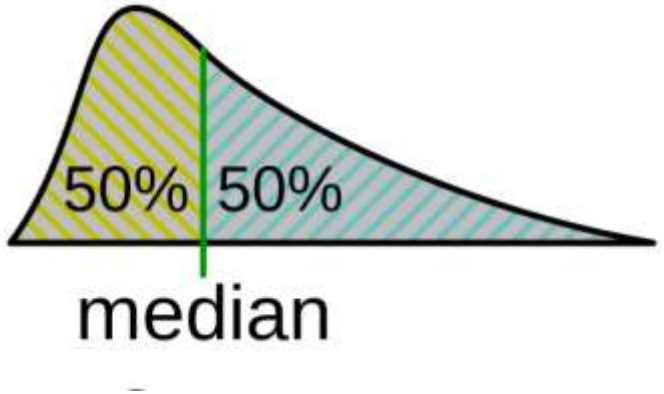
Outline

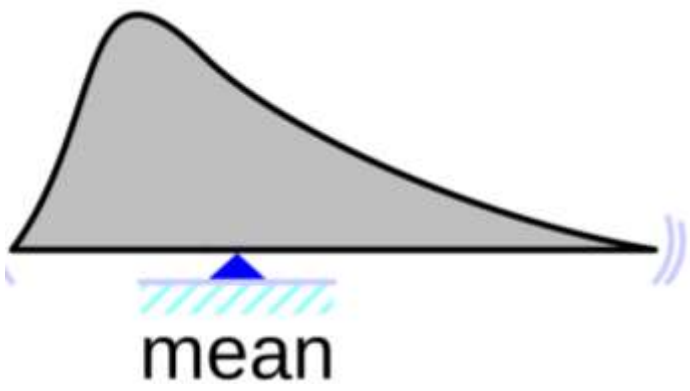
- Modes in distributions
- Mode collapse in GANS
- Intuition behind it during training

Mode

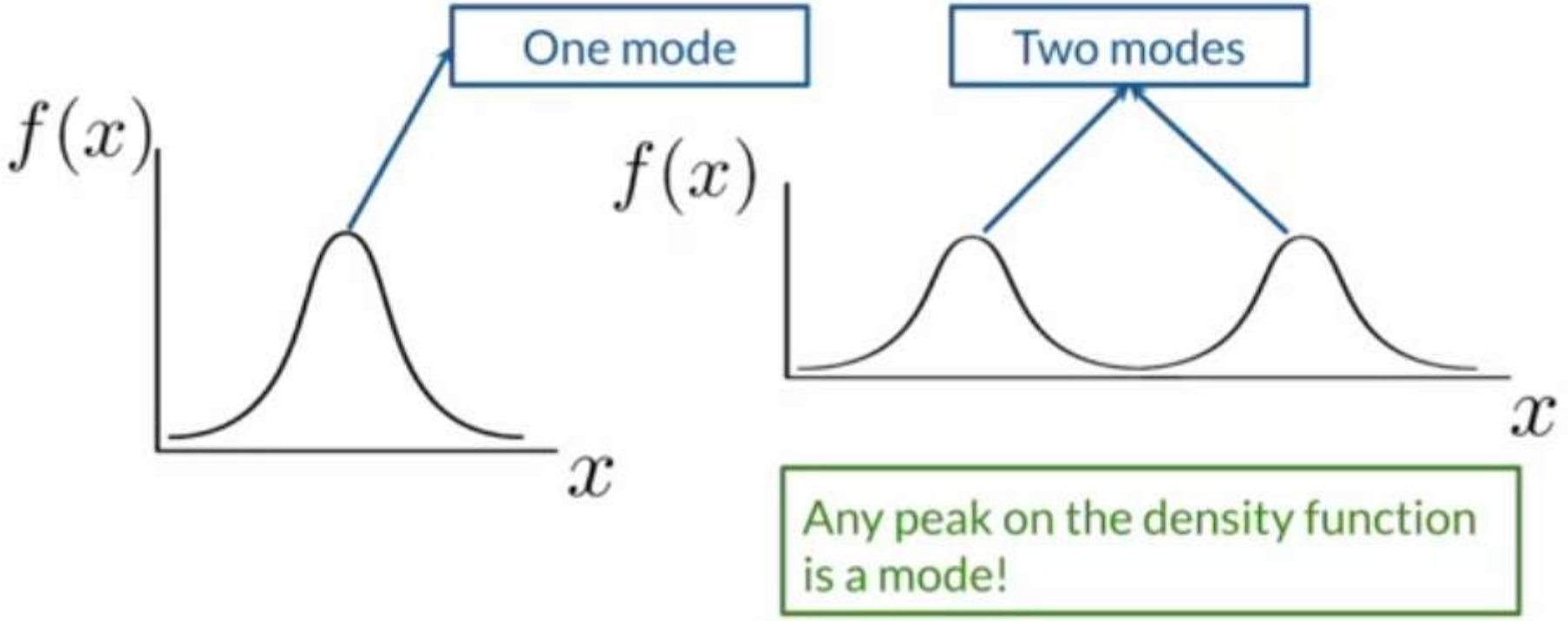
The **mode** is the value that appears most often in a set of data values. (Wikipedia)





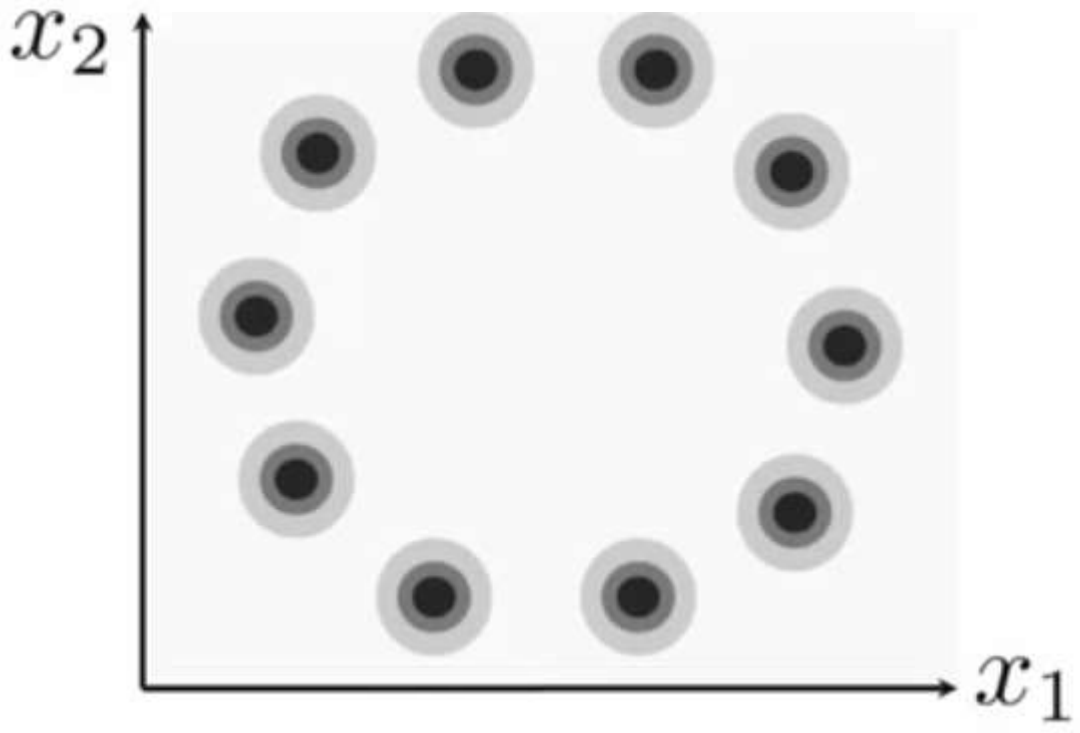


Mode collapse

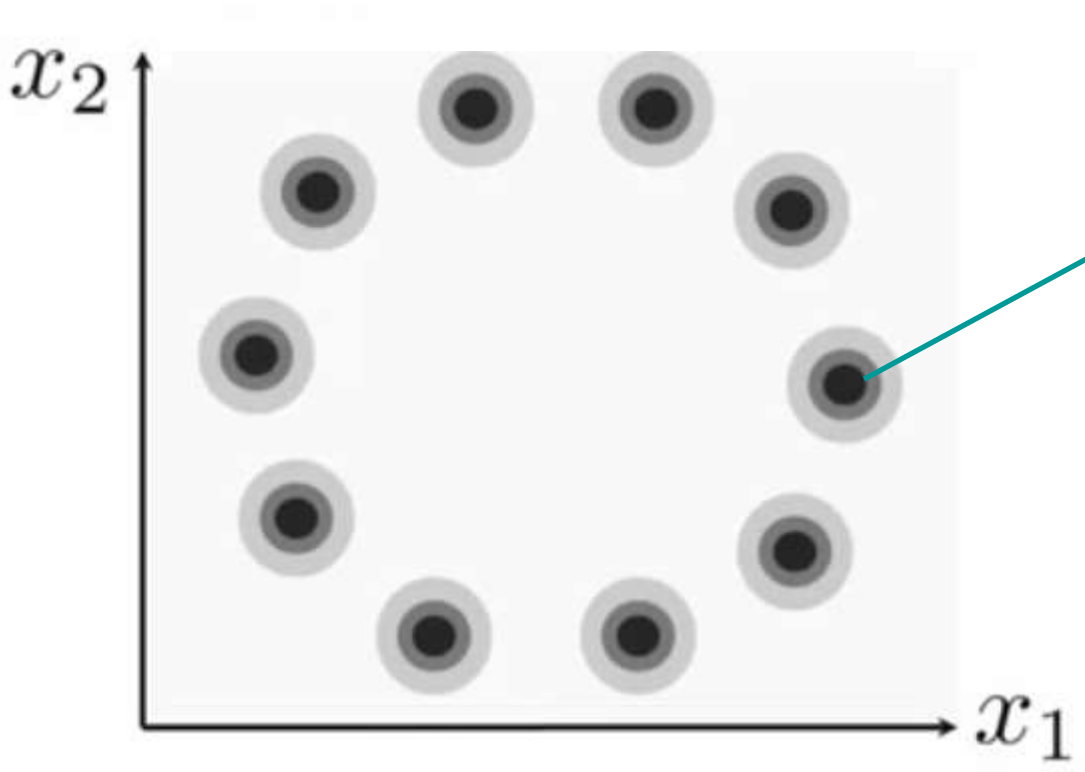


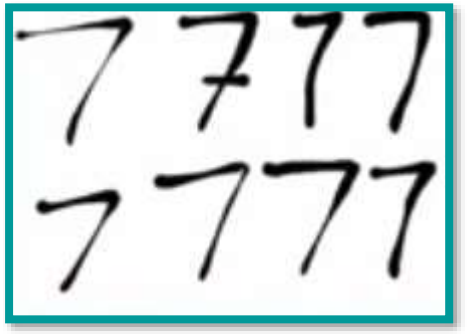
Mode collapse

Mode collapse

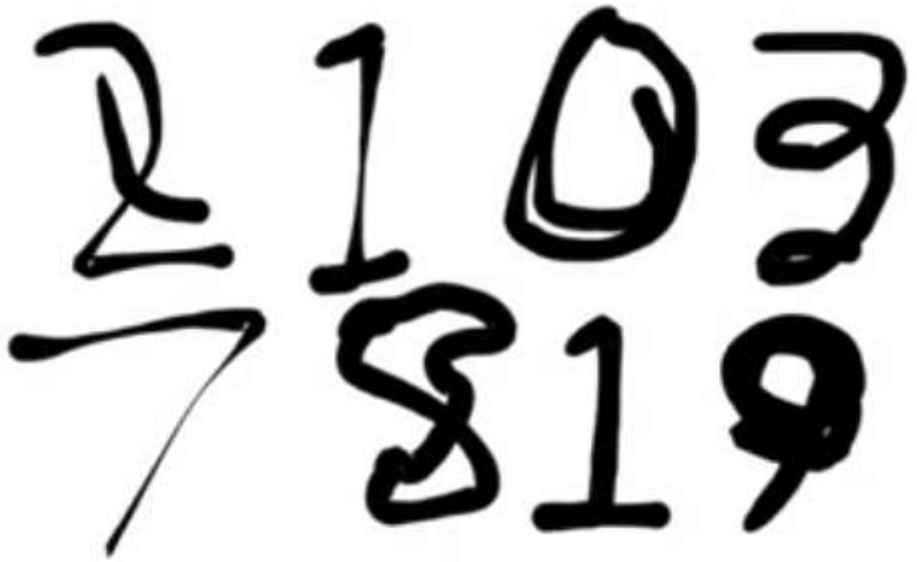


Mode collapse





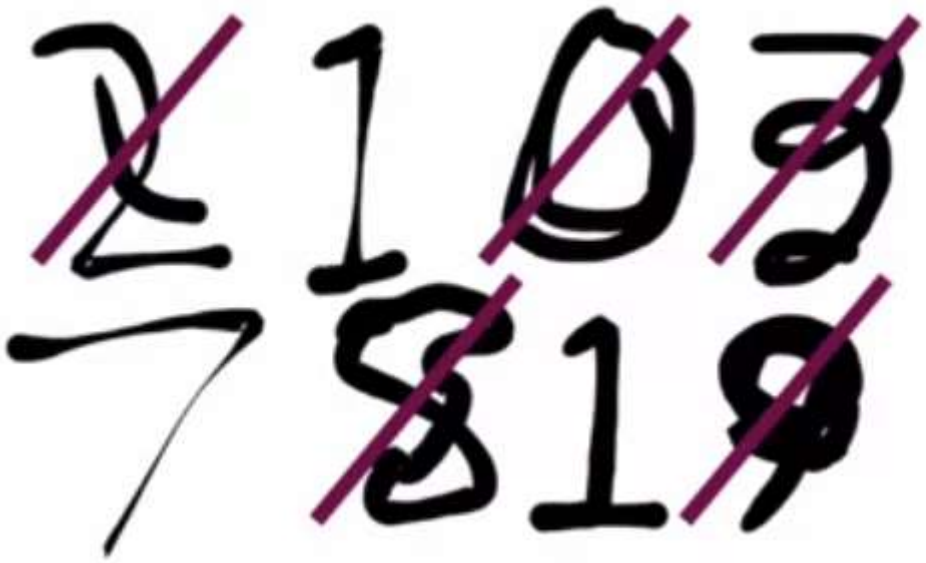
Mode collapse



Mode collapse

Mode collapse

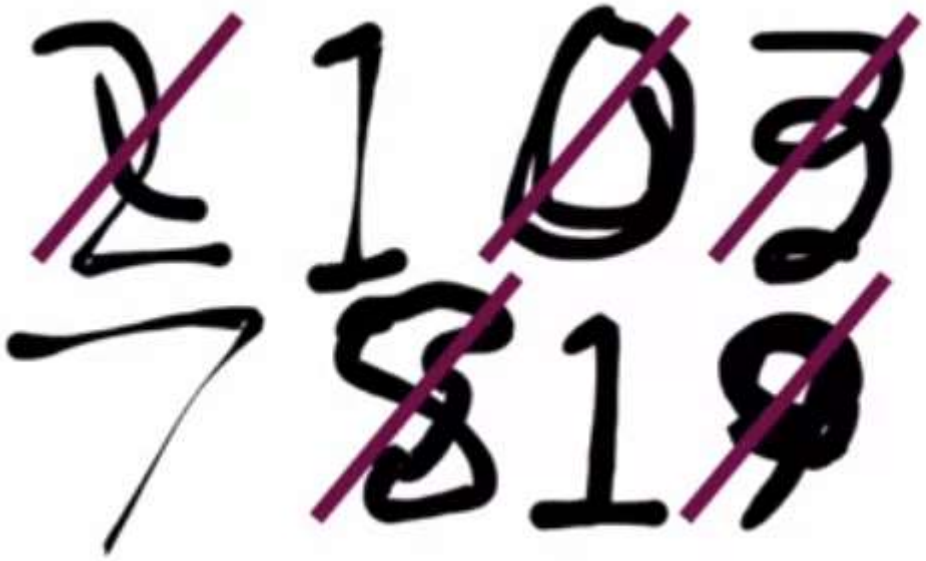
Discriminator



Fakes

Mode collapse

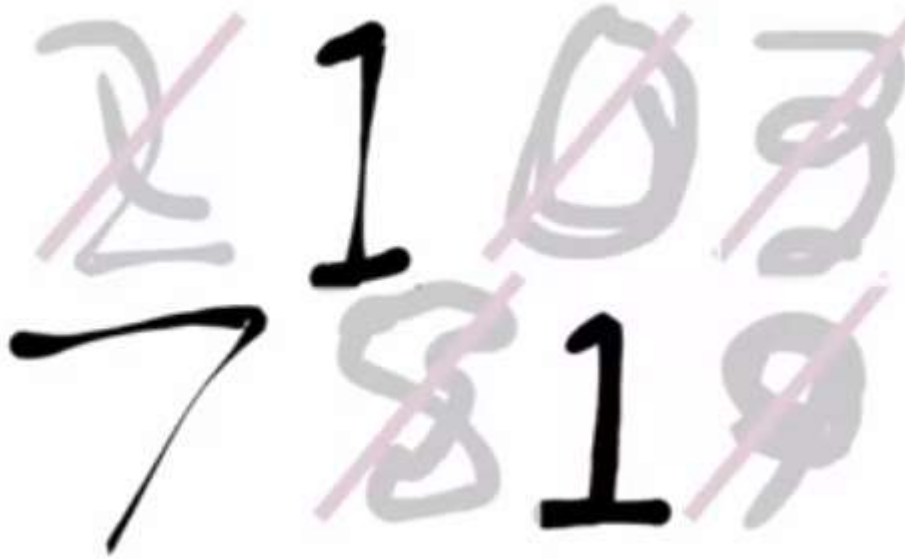
Generator



Fakes

Mode collapse

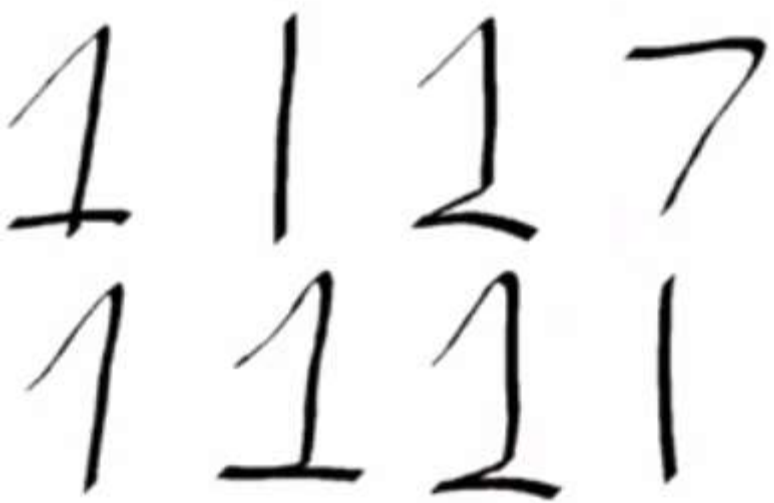
Generator



Fakes that fooled the discriminator

Mode collapse

Generator



Mode collapse

Fakes

Mode collapse

Generator

Fakes

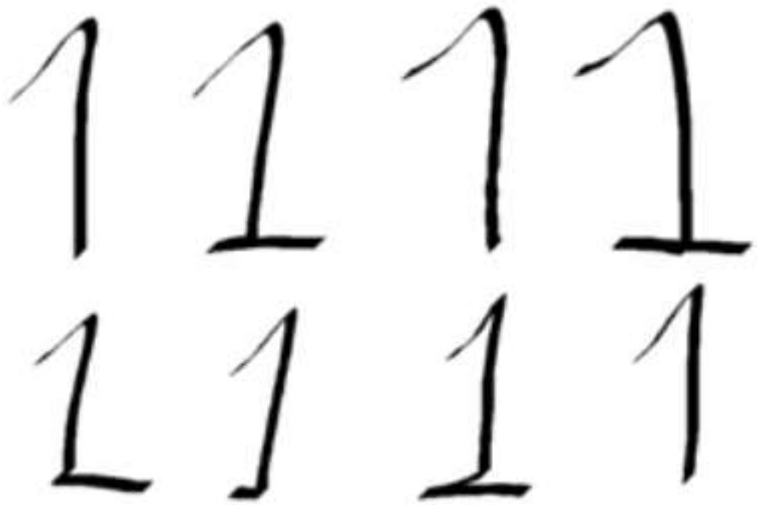
Mode collapse

Generator

Fakes that fooled the discriminator

Mode collapse

Generator



Summary

- Modes are peaks in the distribution of features
- Typical with real-world datasets
- Mode collapse happens when the generator gets stuck in one mode

Problem with BCE Loss

Outline

- BCE Loss and the end objective in GANs
- Problem with BCE Loss

BCE Loss in GANs

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log h\left(x^{(i)},\theta\right) + \left(1-y^{(i)}\right)\log\left(1-\mathrm{h}\left(x^{(i)},\theta\right)\right)\right]$$

Prediction

Label

Features

Parameters

BCE Loss in GANs

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log h\left(x^{(i)},\theta\right) + \left(1-y^{(i)}\right)\log\left(1-\mathrm{h}\left(x^{(i)},\theta\right)\right)\right]$$

Prediction

Label

Features

Parameters

Maximize cost

BCE Loss in GANs

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log h\left(x^{(i)},\theta\right) + \left(1-y^{(i)}\right)\log\left(1-\mathrm{h}\left(x^{(i)},\theta\right)\right)\right]$$

Prediction

Label

Features

Parameters

Generator

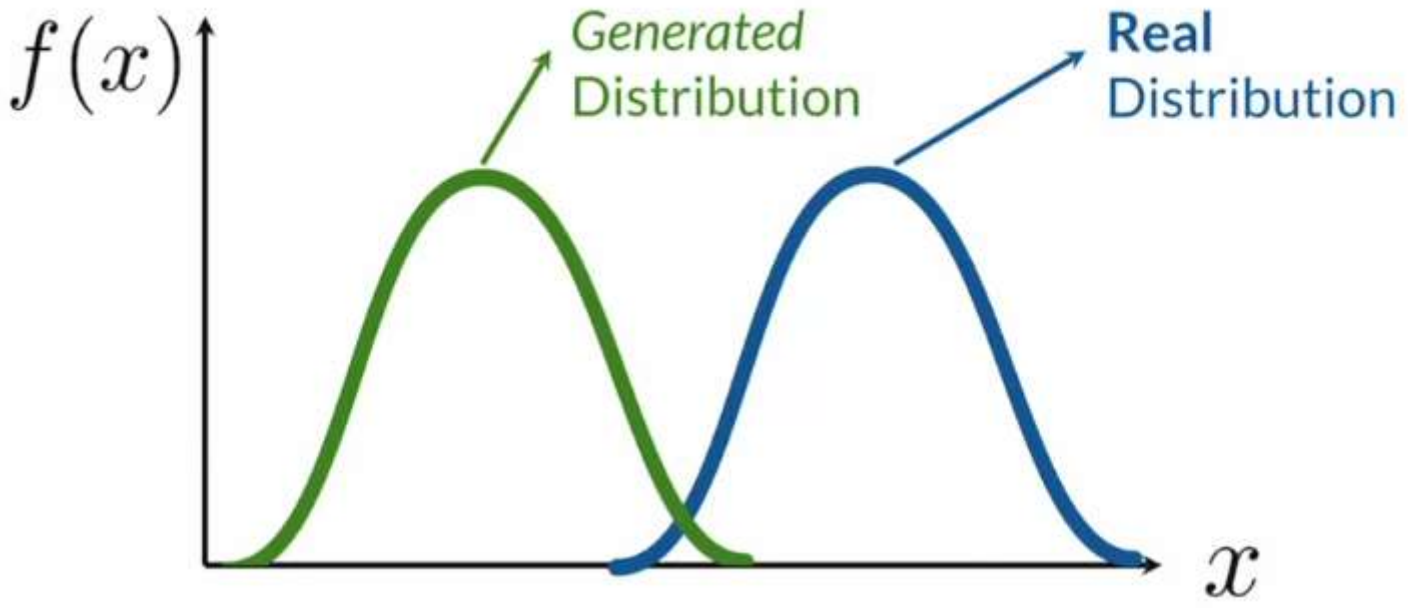
Maximize cost

Discriminator

Minimize cost

Objective in GANs

Make the generated and real distributions look similar



Problem with BCE Loss

Often, the discriminator gets better than the generator

Discriminator

Single output

Easier to train than the generator

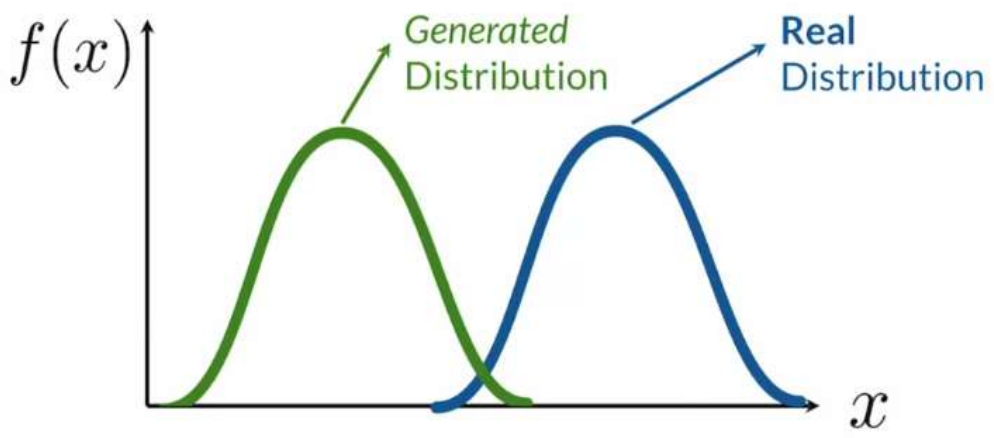
Generator

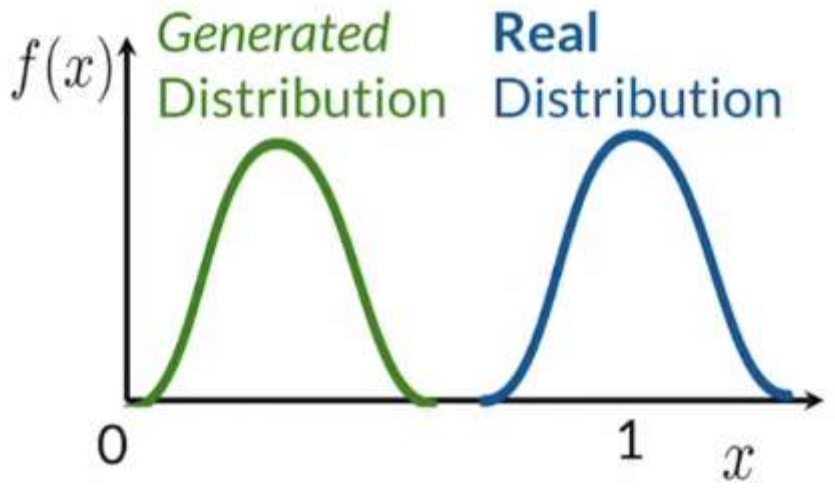
Complex output

Difficult to train

Problem with BCE Loss

- Difference between distributions
- Gradient vanishing!!!





Summary

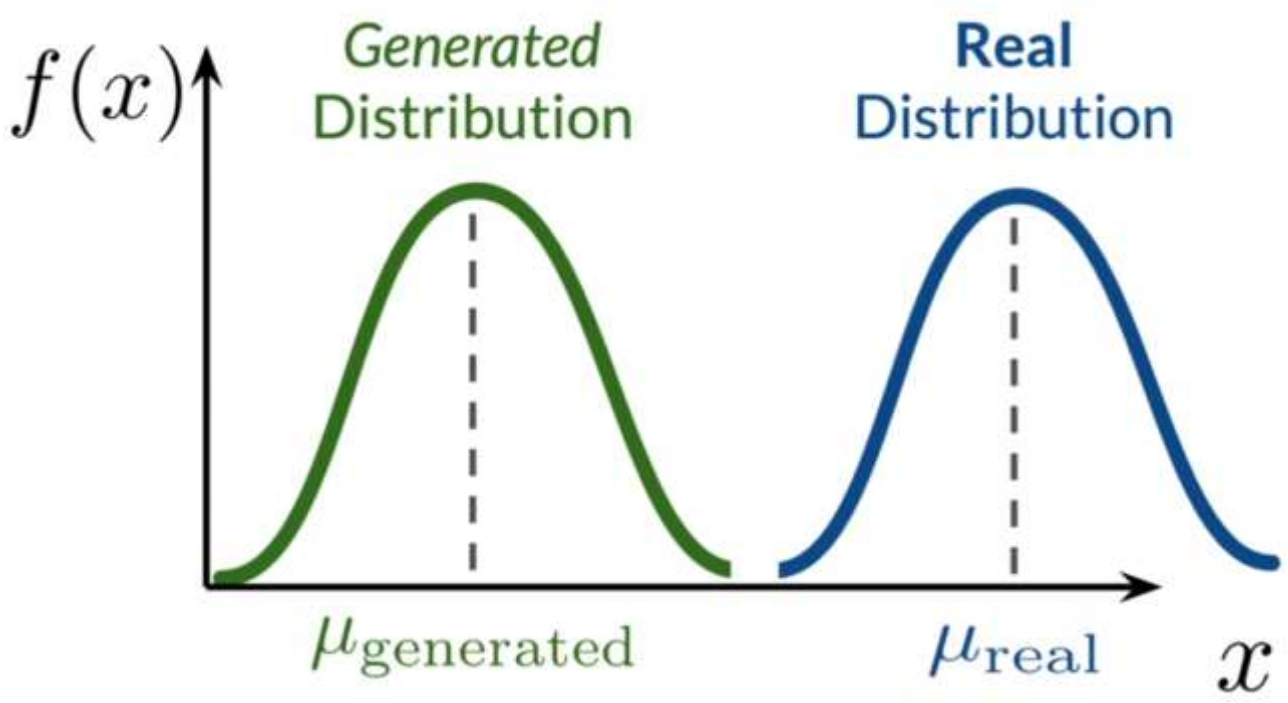
- GANs try to make the real and generated distributions look similar
- When the discriminator improves too much, the function approximated by BCE Loss will contain flat regions
- Flat regions on the cost function **= vanishing gradients**

Earth Mover’s Distance

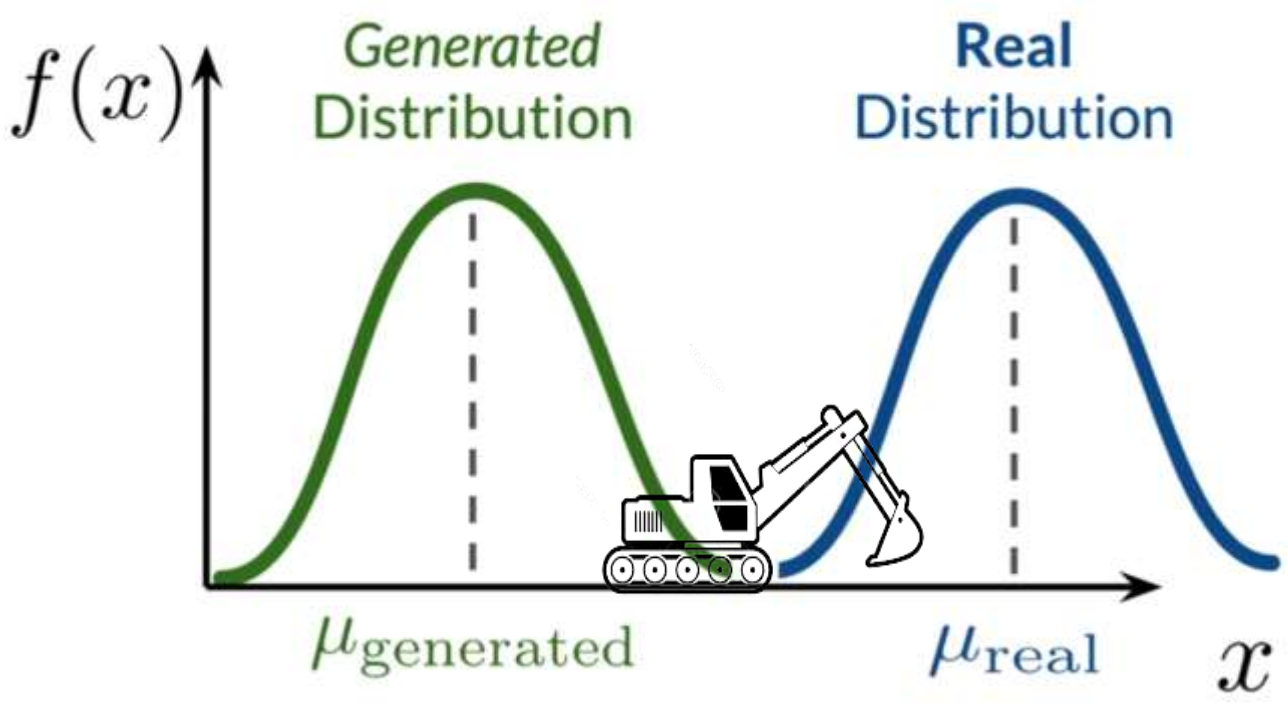
Outline

- Earth Mover's Distance (EMD)
- Why it solves the vanishing gradient problem of BCE Loss

Earth Mover's Distance

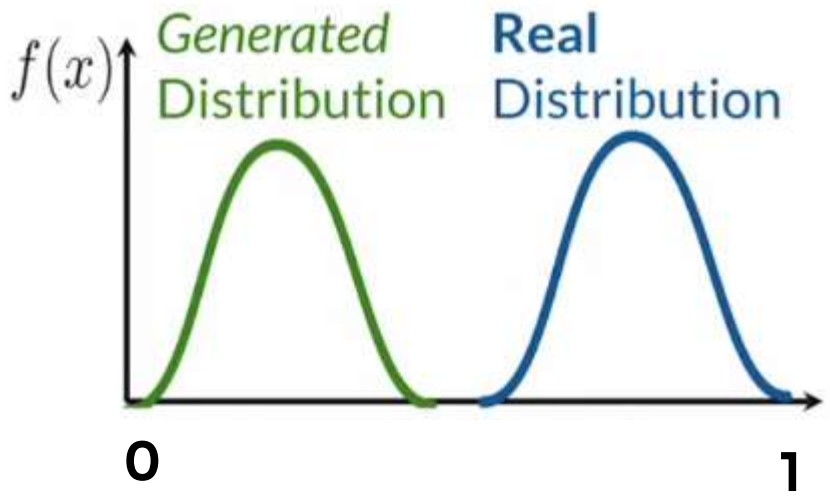


Earth Mover's Distance



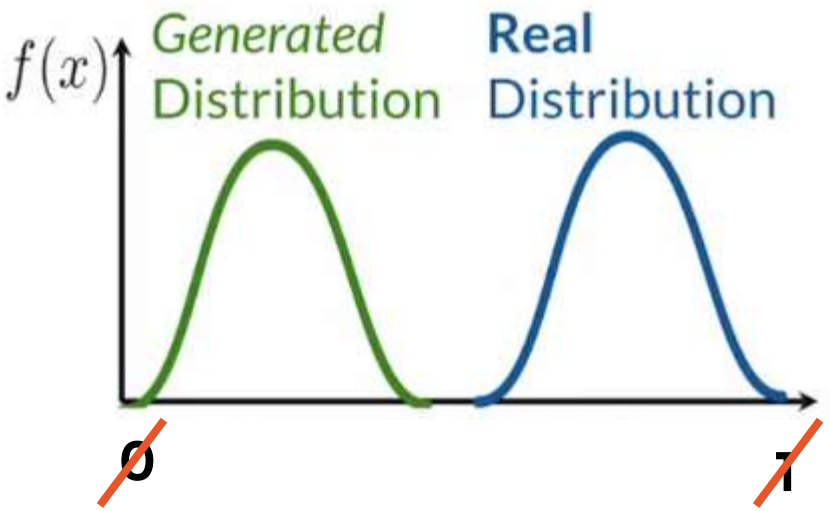
Effort to make the generated distribution equal to the real distribution

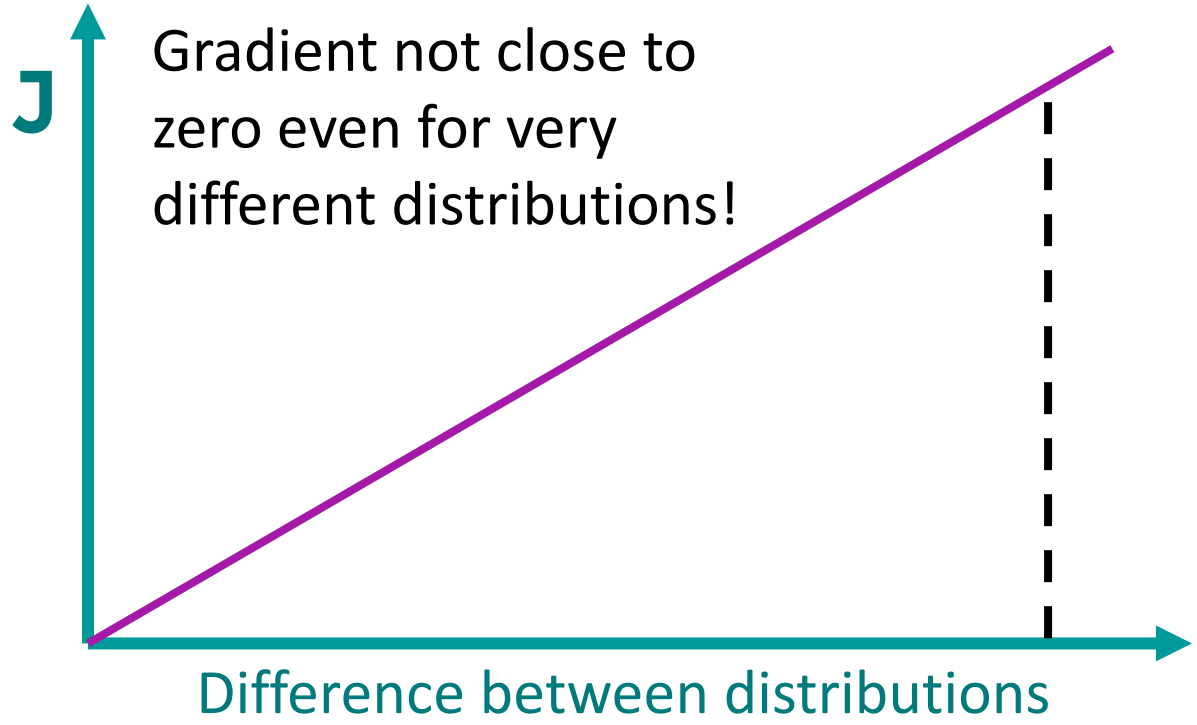
Earth Mover's Distance



Cross Entropy

Earth Mover's Distance



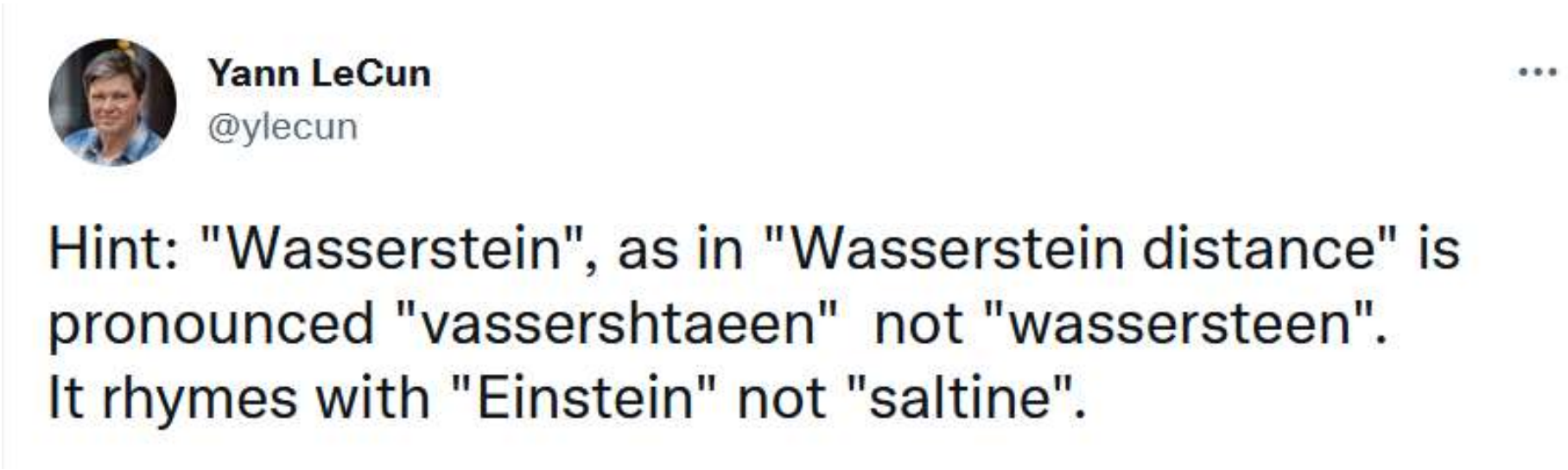


Summary

- Earth mover's distance (EMD) is a function of amount and distance
- Doesn't have flat regions when the distributions are very different
- Approximating EMD solves the problems associated with BCE

Wasserstein Loss

How To Pronounce ☺



https://twitter.com/ylecun/status/991375003626233858?lang=en

Outline

- BCE Loss Simplified
- W-Loss and its comparison with BCE Loss

BCE Loss Simplified

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log\left(h\left(x^{(i)},\theta\right)\right) + \left(1-y^{(i)}\right)\log\left(1-h\left(x^{(i)},\theta\right)\right)\right]$$

BCE Loss Simplified

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log\left(h\left(x^{(i)},\theta\right)\right) + \left(1-y^{(i)}\right)\log\left(1-h\left(x^{(i)},\theta\right)\right)\right]$$

$$\max_{g}$$

Minimize cost

شبکه علیرضا

Maximize cost

BCE Loss Simplified

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log\left(h\left(x^{(i)},\theta\right)\right) + \left(1-y^{(i)}\right)\log\left(1-h\left(x^{(i)},\theta\right)\right)\right]$$

$$\min_{d}\max_{g}$$

Discriminator

Minimize cost

شبکه علیرضا

Generator

Maximize cost

BCE Loss Simplified

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log\left(h\left(x^{(i)},\theta\right)\right) + \left(1-y^{(i)}\right)\log\left(1-h\left(x^{(i)},\theta\right)\right)\right]$$

$$\min_{d}\max_{g} -\left[\mathbb{E}(\quad) + \mathbb{E}(\quad)\right]$$

Discriminator

Minimize cost

شبکه

علیرضا

Generator

Maximize cost

BCE Loss Simplified

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log\left(h\left(x^{(i)},\theta\right)\right) + \left(1-y^{(i)}\right)\log\left(1-h\left(x^{(i)},\theta\right)\right)\right]$$

$$\min_{d}\max_{g} -\left[\mathbb{E}\left(\ log(d(x))\ \right) + \mathbb{E}\left(\ 1 - log\left(d(g(z))\right)\ \right)\right]$$

Discriminator

Minimize cost

شبکه علیرضا

Generator

Maximize cost

W-Loss

- W-Loss approximates the **Earth Mover's Distance**

W-Loss

- W-Loss approximates the **Earth Mover's Distance**

$$\min_{g} \max_{c} \left| \mathbb{E}\left(c(x)\right) - \mathbb{E}\left(c\left(g(z)\right)\right) \right|$$

W-Loss

- W-Loss approximates the **Earth Mover's Distance**

$$\min_{g} \max_{c} \left| \mathbb{E}\left(c(x)\right) - \mathbb{E}\left(c\left(g(z)\right)\right) \right|$$

Maximize the distance

W-Loss

- W-Loss approximates the **Earth Mover's Distance**

$$\min_{g} \max_{c} \left| \mathbb{E}(c(x)) - \mathbb{E}\left(c(g(z))\right) \right|$$

Generator

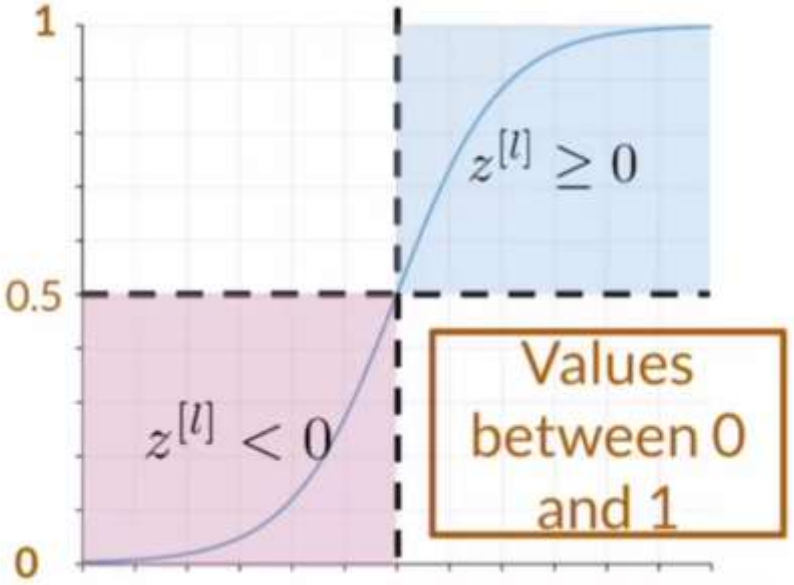
Minimize the distance

Critic

Maximize the distance

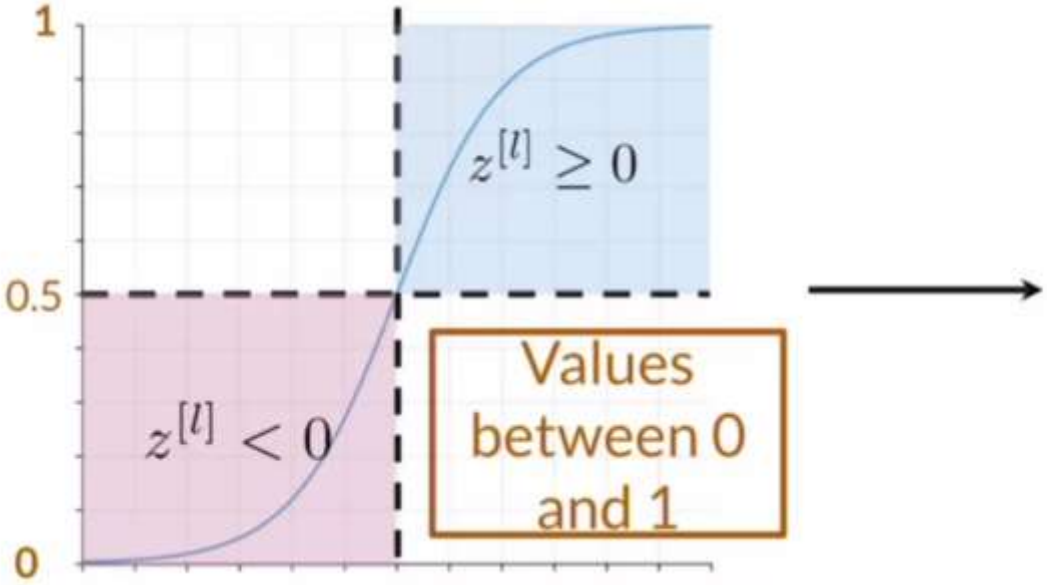
Discriminator Output

Discriminator output



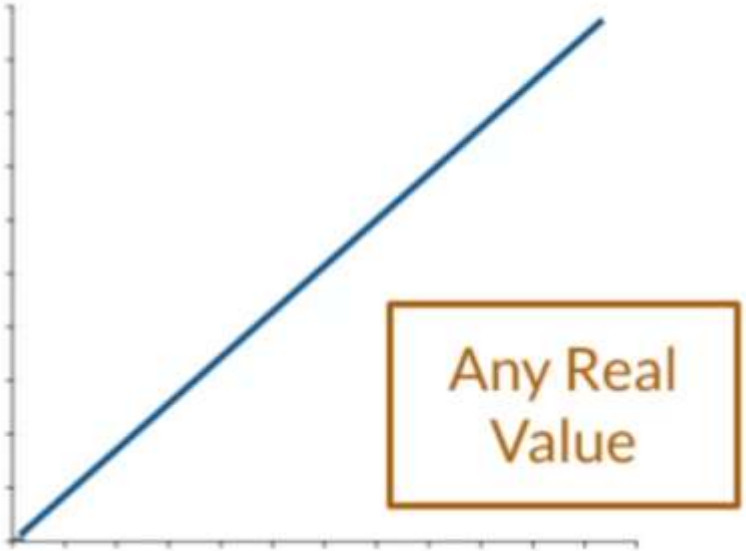
Discriminator Output

Discriminator output



Critic

~~Discriminator~~ output



W-Loss vs BCE Loss

BCE Loss	W-Loss
Discriminator outputs between 0 and 1	Critic outputs any number

W-Loss vs BCE Loss

BCE Loss	W-Loss
Discriminator outputs between 0 and 1	Critic outputs any number
$-\left[\mathbb{E}\left(log\left(d(x)\right)\right) + \mathbb{E}(1 - log(d(g(z)))\right)\right]$	$\left[\mathbb{E}\left(c(x)\right) - \mathbb{E}\left(c\left(g(z)\right)\right)\right]$

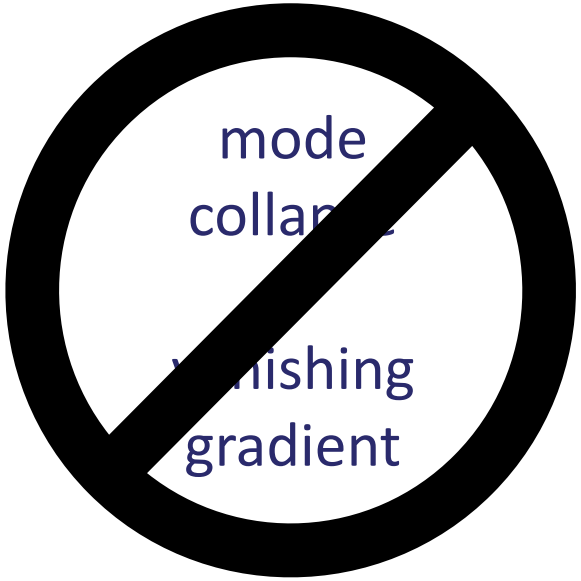
W-Loss vs BCE Loss

BCE Loss	W-Loss
Discriminator outputs between 0 and 1	Critic outputs any number
$-\left[\mathbb{E}\left(log\left(d(x)\right)\right) + \mathbb{E}(1 - log(d(g(z)))\right)\right]$	$\left[\mathbb{E}\left(c(x)\right) - \mathbb{E}\left(c\left(g(z)\right)\right)\right]$

W-Loss helps with mode collapse and vanishing gradient problems

Summary

- W-Loss looks very similar to BCE Loss
- W-Loss prevents mode collapse and vanishing gradient problems



Condition on Wasserstein Critic

Outline

- Continuity condition on the critic's neural network
- Why this condition matters

Condition on W-loss

$$\min_{g} \max_{c} \left| \mathbb{E}\left(c(x)\right) - \mathbb{E}\left(c(g(z))\right) \right|$$

Condition on W-loss

$$\min_{g} \max_{c} \left| \mathbb{E}\left(c(x)\right) - \mathbb{E}\left(c\left(g(z)\right)\right) \right|$$

Condition on W-loss

$$\min_{g} \max_{c} \left| \mathbb{E}\left(c(x)\right) - \mathbb{E}\left(c\left(g(z)\right)\right) \right|$$

CLASS. VISION

Condition on W-loss

$$\min_{g} \max_{c} \left| \mathbb{E}\left(c(x)\right) - \mathbb{E}\left(c(g(z))\right) \right|$$

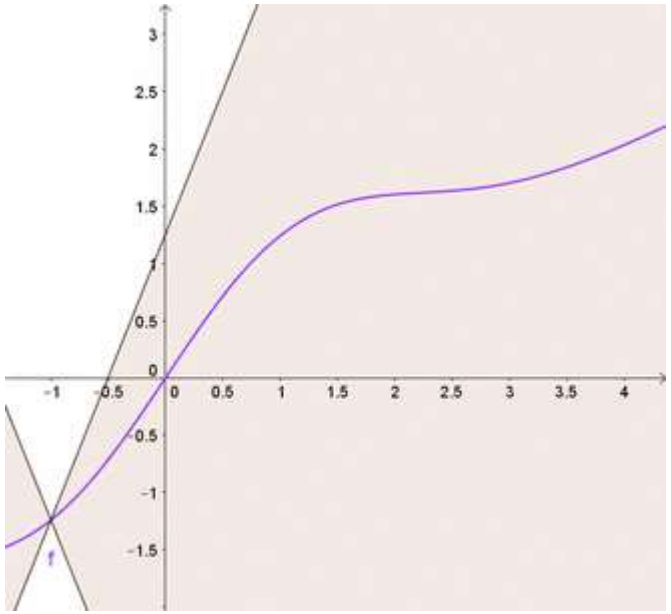
Needs to be 1-Lipschitz Continues

Condition on W-loss

- Critic needs to be 1-Lipschitz continues
- The norm of the gradient should be at most 1 for every point

What is **Lipschitz continuity**

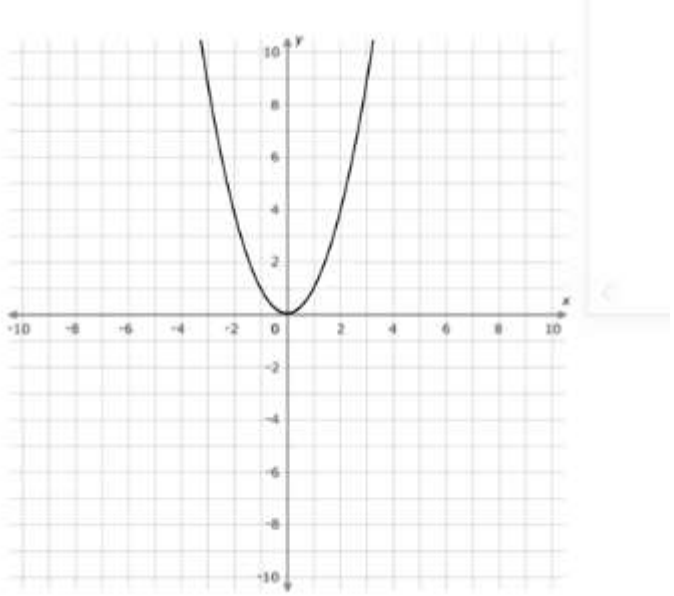
- Lipschitz continuous function is limited in how fast it can change
- For a Lipschitz continuous function, there exists a double cone (white) whose origin can be moved along the graph so that the whole graph always stays outside the double cone



https://en.wikipedia.org/wiki/Lipschitz_continuity

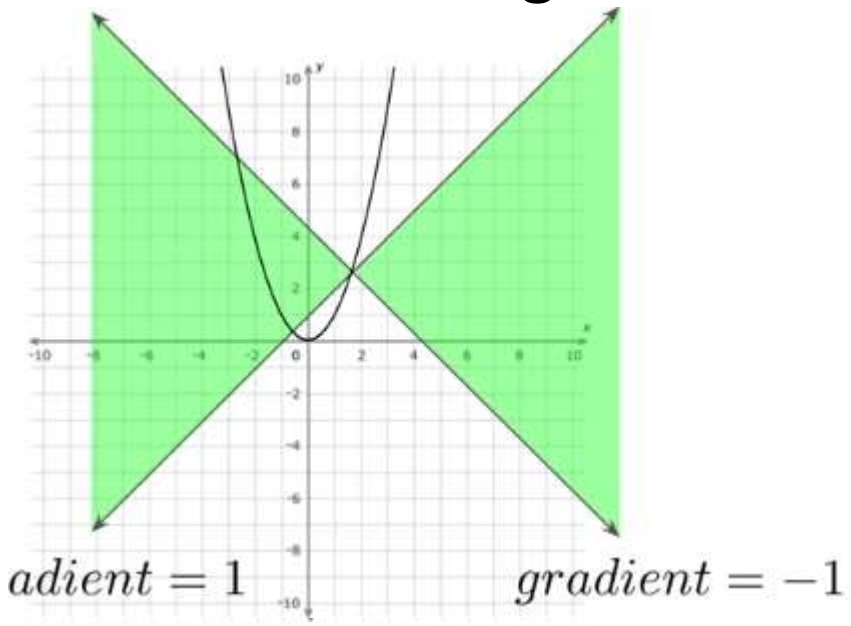
Condition on W-loss

- Critic needs to be 1-Lipschitz continues
- The norm of the gradient should be at most 1 for every point



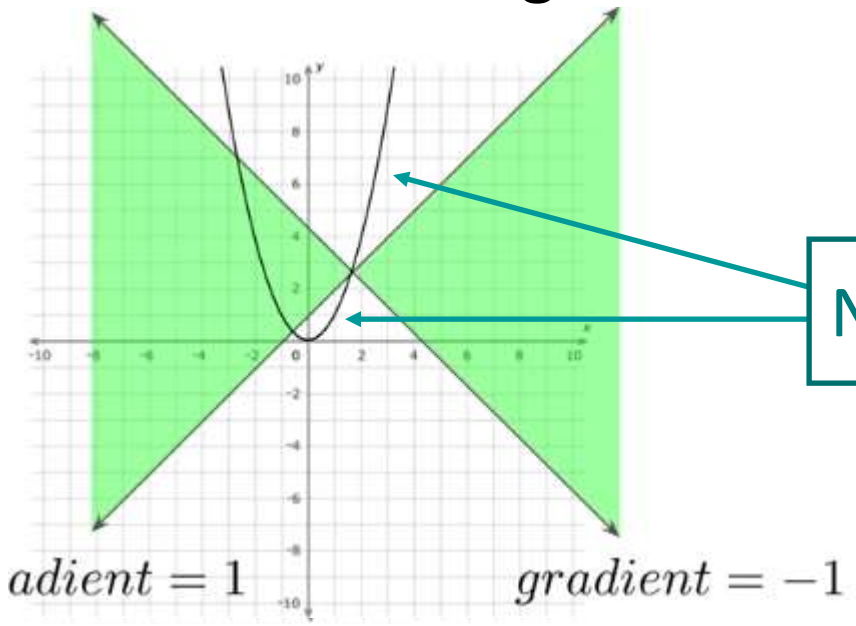
Condition on W-loss

- Critic needs to be 1-Lipschitz continues
- The norm of the gradient should be at most 1 for every point



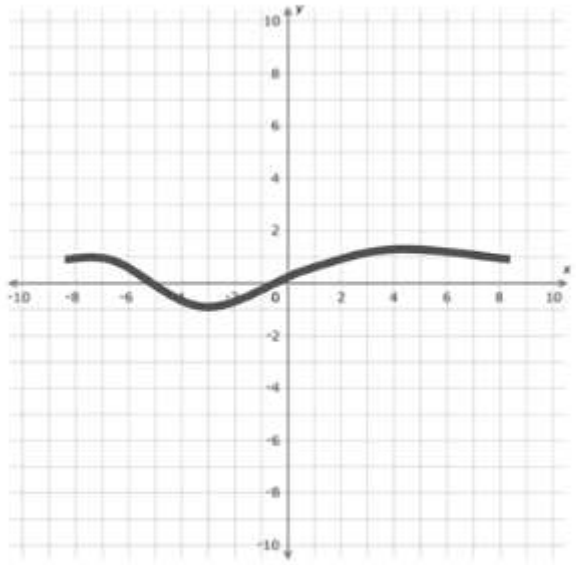
Condition on W-loss

- Critic needs to be 1-Lipschitz continues
- The norm of the gradient should be at most 1 for every point



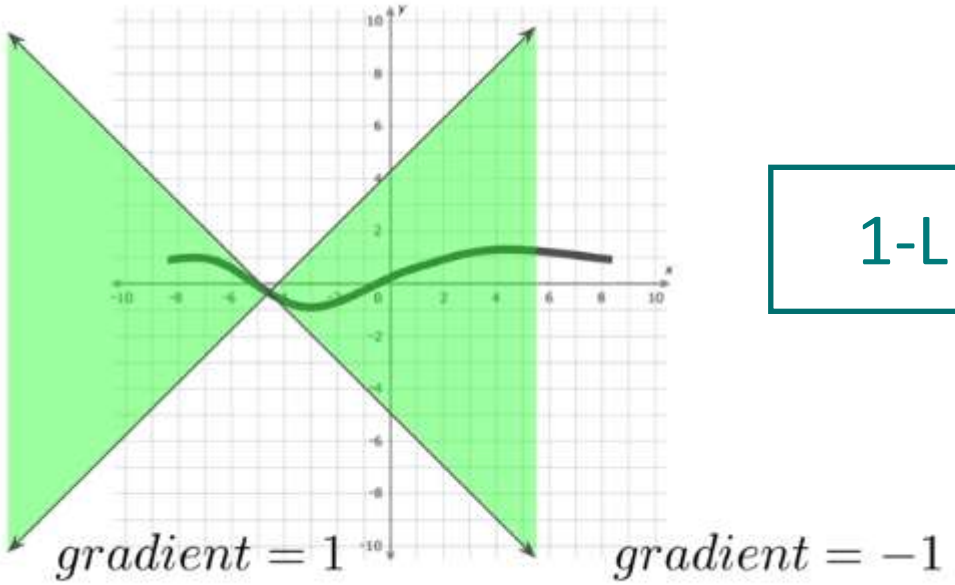
Condition on W-loss

- Critic needs to be 1-Lipschitz continues
- The norm of the gradient should be at most 1 for every point



Condition on W-loss

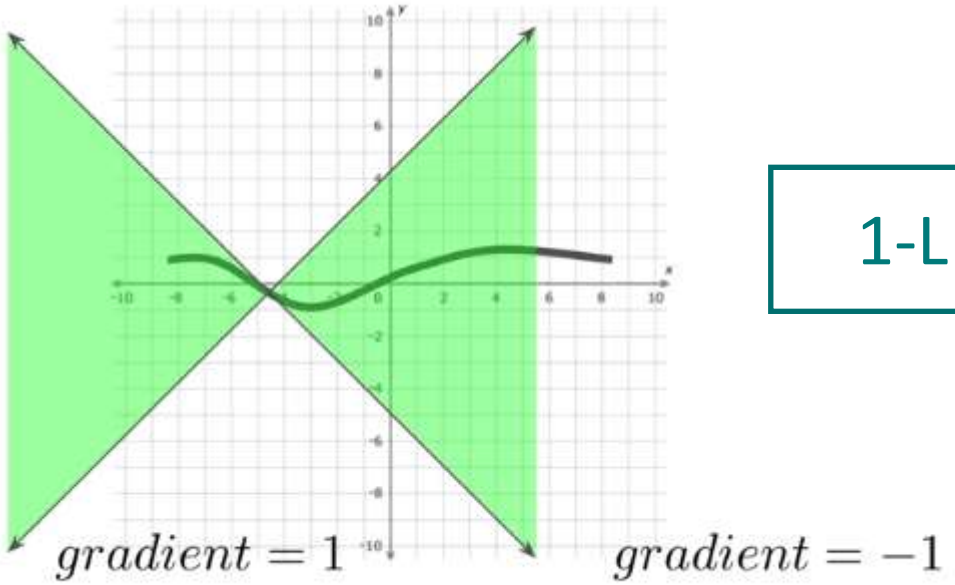
- Critic needs to be 1-Lipschitz continues
- The norm of the gradient should be at most 1 for every point



1-L continues

Condition on W-loss

- Critic needs to be 1-Lipschitz continues
- The norm of the gradient should be at most 1 for every point

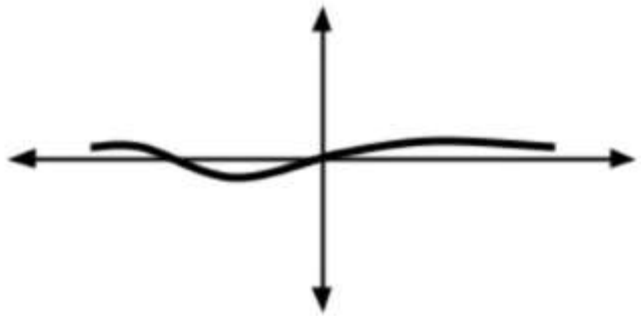


1-L continues

So W-loss is Valid.
Needed for training stable neural networks with W-loss

Summary

- Critic's neural network needs to be 1-L continuous when using w-loss
- This condition ensures that W-Loss is validly approximating Earth Mover's Distance



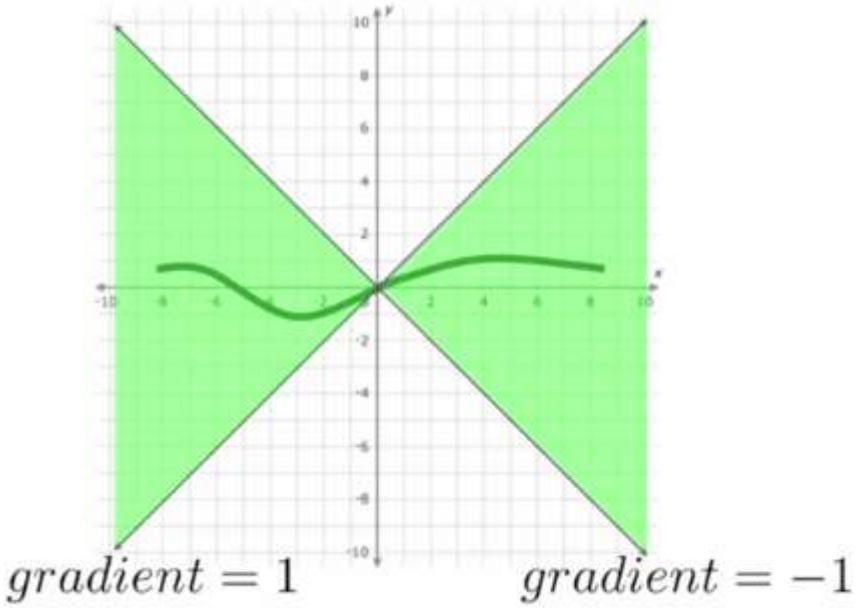
1-Lipschitz Continuity Enforcement

Outline

- Weight clipping and gradient penalty
- Advantages of gradient penalty

Enforcement

- Critic needs to be 1-L continuous



Norm of the gradient at most 1

$$||\nabla f(x)||_2 \leq 1$$

Enforcement: Weight clipping

Weight clipping forces the weights of the critic to a fixed interval

Enforcement: Weight clipping (Tensorflow implementation A)

```
class WeightsClip(tf.keras.constraints.Constraint):
    def __init__(self, min_value=-0.01, max_value=0.01):
        self.min_value = min_value
        self.max_value = max_value
    def __call__(self, w):
        return tf.clip_by_value(w, self.min,
                                self.max_value)
```

Enforcement: Weight clipping (Tensorflow implementation A)

```
model = tf.keras.Sequential(name='critics')
model.add(Conv2D(16, 3, strides=2, padding='same',
                 kernel_constraint=WeightsClip(),
                 bias_constraint=WeightsClip()))
model.add(BatchNormalization(
                 beta_constraint=WeightsClip(),
                 gamma_constraint=WeightsClip()))
```

Enforcement: Weight clipping (Tensorflow implementation B)

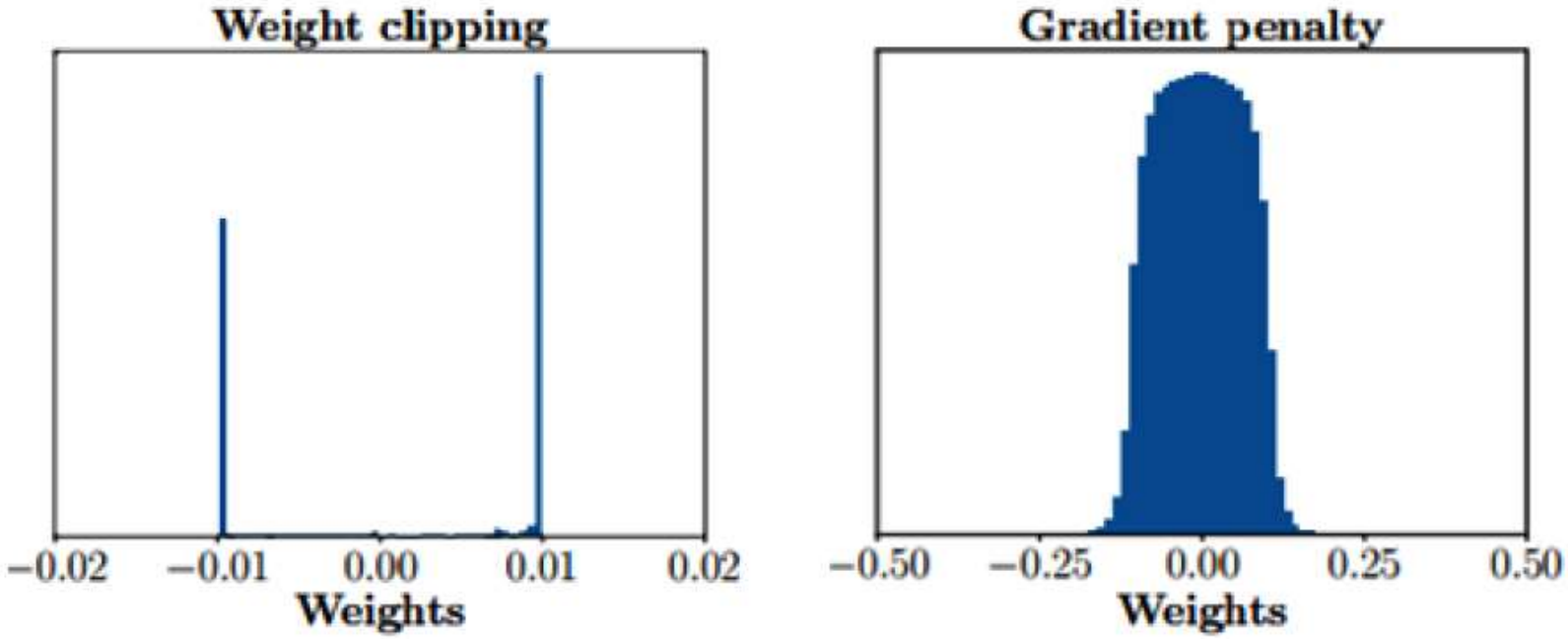
```
for layer in critic.layers:
    weights = layer.get_weights()
    weights = [tf.clip_by_value(w, -0.01, 0.01) for
                  w in weights]
    layer.set_weights(weights)
```

Enforcement: Weight clipping

Weight clipping forces the weights of the critic to a fixed interval

Limits the learning ability of the critic

Enforcement: Weight clipping VS Gradient penalty



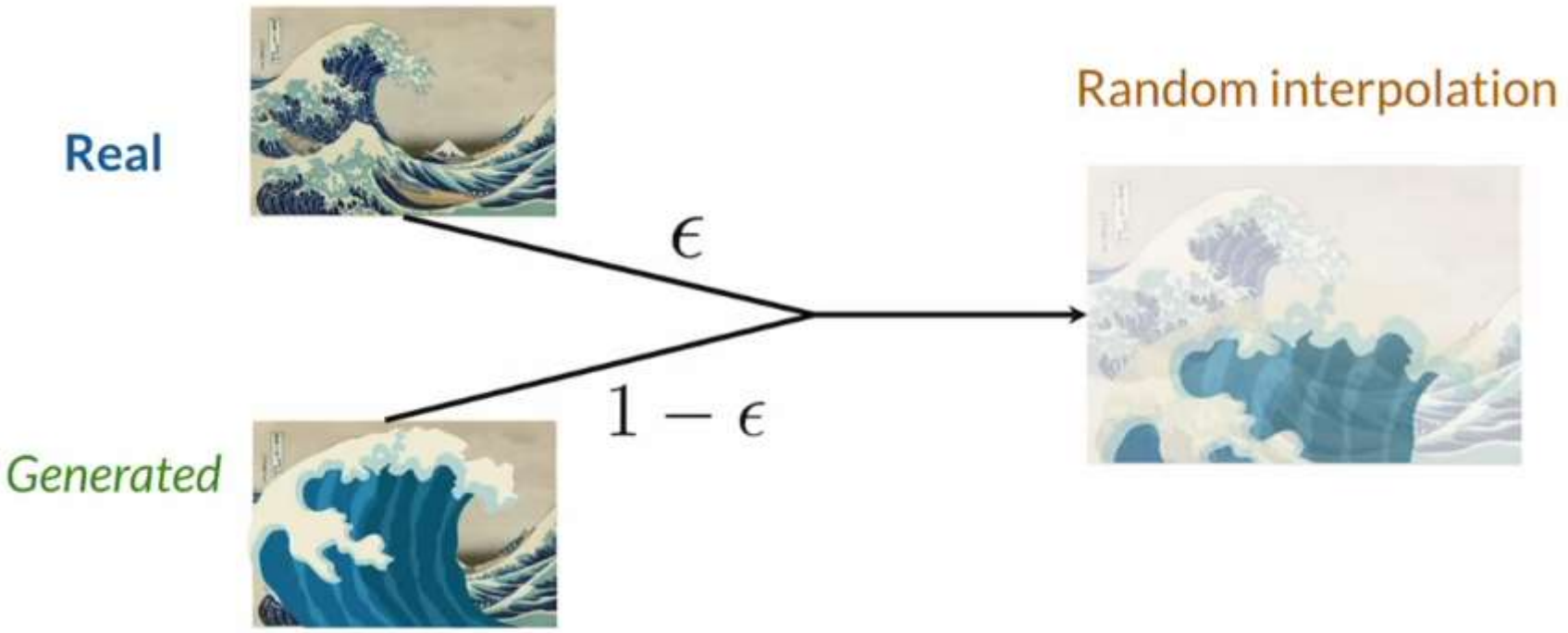
Left: Weight clipping pushes weights toward two values. Right: Gradients produced by gradient penalty. Source: I. Gulrajani et al, 2017, Improved Training of Wasserstein GANs

Enforcement: Gradient penalty

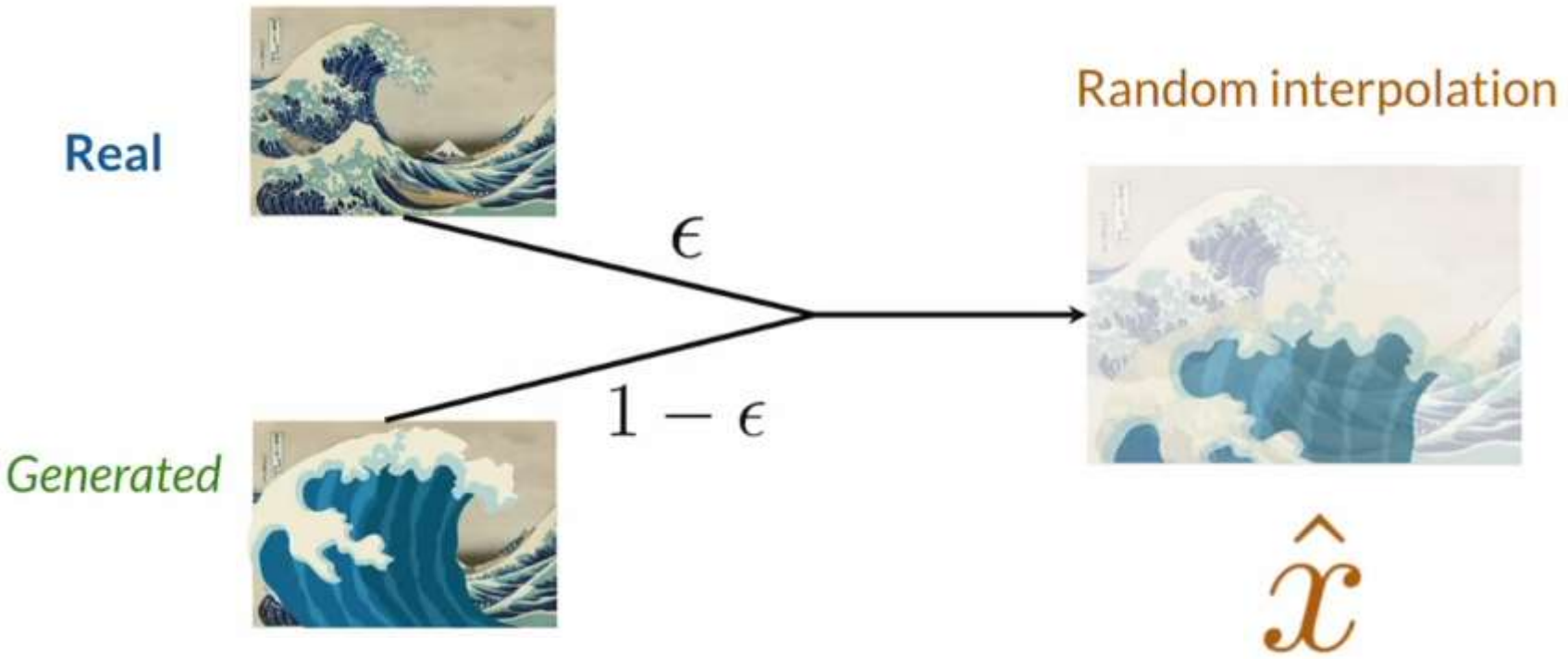
$$\min_{g} \max_{c} \left| \mathbb{E}\big(c(x)\big) - \mathbb{E}\Big(c\big(g(z)\big)\Big) \right| + \lambda\, reg$$

Regularization of critic's gradient

Enforcement: Gradient penalty



Enforcement: Gradient penalty



Enforcement: Gradient penalty

$$(\|\nabla c(\hat{x})\|_2 - 1)^2$$

Regularization term

Enforcement: Gradient penalty

$$(\|\nabla c(\hat{x})\|_2 - 1)^2$$

Regularization term

$$\hat{x} = \epsilon x + (1 - \epsilon)\, g(z)$$

Interpolation

Enforcement: Gradient penalty

$$(\|\nabla c(\hat{x})\|_2 - 1)^2$$

Regularization term

$$\epsilon x + (1 - \epsilon)\, g(z)$$

Interpolation

Real

Generated

4 Gradient penalty

We now propose an alternative way to enforce the Lipschitz constraint. A differentiable function is 1-Lipschtiz if and only if it has gradients with norm at most 1 everywhere, so we consider directly constraining the gradient norm of the critic's output with respect to its input. To circumvent tractability issues, we enforce a soft version of the constraint with a penalty on the gradient norm for random samples $\hat{\boldsymbol{x}} \sim \mathbb{P}_{\hat{\boldsymbol{x}}}$. Our new objective is

$$L = \underbrace{\mathop{\mathbb{E}}_{\tilde{\boldsymbol{x}} \sim \mathbb{P}_g}[D(\tilde{\boldsymbol{x}})] - \mathop{\mathbb{E}}_{\boldsymbol{x} \sim \mathbb{P}_r}[D(\boldsymbol{x})]}_{\text{Original critic loss}} + \underbrace{\lambda \mathop{\mathbb{E}}_{\hat{\boldsymbol{x}} \sim \mathbb{P}_{\hat{\boldsymbol{x}}}}\left[(\|\nabla_{\hat{\boldsymbol{x}}} D(\hat{\boldsymbol{x}})\|_2 - 1)^2\right]}_{\text{Our gradient penalty}}. \tag{3}$$

Sampling distribution We implicitly define $\mathbb{P}_{\hat{\boldsymbol{x}}}$ sampling uniformly along straight lines between pairs of points sampled from the data distribution $\mathbb{P}_r$ and the generator distribution $\mathbb{P}_g$. This is motivated by the fact that the optimal critic contains straight lines with gradient norm 1 connecting coupled points from $\mathbb{P}_r$ and $\mathbb{P}_g$ (see Proposition 1). Given that enforcing the unit gradient norm constraint everywhere is intractable, enforcing it only along these straight lines seems sufficient and experimentally results in good performance.

Penalty coefficient All experiments in this paper use $\lambda = 10$, which we found to work well across a variety of architectures and datasets ranging from toy tasks to large ImageNet CNNs.

WGAN-GP paper

Improved Training of Wasserstein GANs Improved Training of Wasserstein GANs

https://arxiv.org/pdf/1704.00028.pdf

Putting It All Together

$$\min_{g} \max_{c} \; \mathbb{E}\big(c(x)\big) - \mathbb{E}\Big(c\big(g(z)\big)\Big) + \lambda \mathbb{E}(\|\nabla c(\hat{x})\|_2 - 1)^2$$

Putting It All Together

$$\min_{g} \max_{c} \boxed{\mathbb{E}\big(c(x)\big) - \mathbb{E}\Big(c\big(g(z)\big)\Big)} + \lambda \mathbb{E}(\|\nabla c(\hat{x})\|_2 - 1)^2$$

Makes the GAN less prone to **mode collapse** and **vanishing gradient**

Putting It All Together

$$\min_{g} \max_{c} \boxed{\mathbb{E}\big(c(x)\big) - \mathbb{E}\Big(c\big(g(z)\big)\Big)} + \boxed{\lambda \mathbb{E}(\|\nabla c(\hat{x})\|_2 - 1)^2}$$

Makes the GAN less prone to **mode collapse** and **vanishing gradient**

Tries to make the critic be 1-L Continuous, for the loss function to be **continuous and differentiable**

Summary

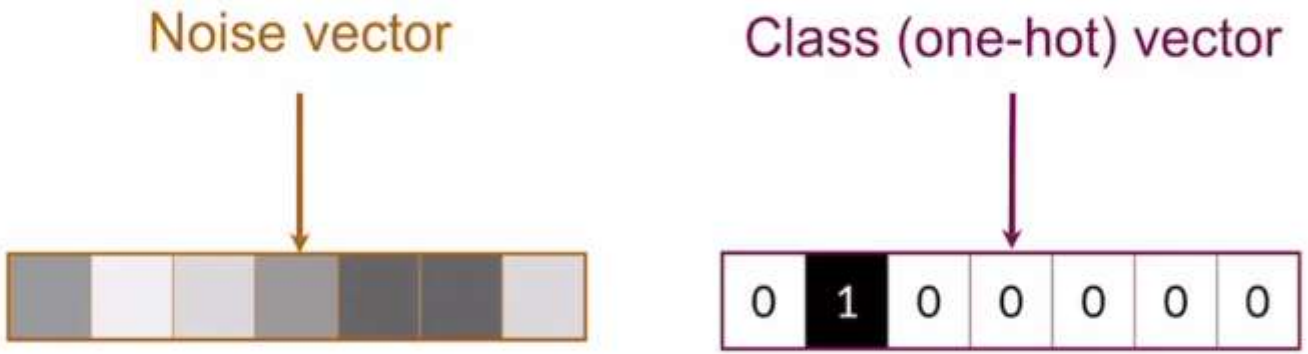
- Weight clipping and gradient penalty are ways to enforce 1-L continuity
- Gradient penalty tends to work better

Implementation

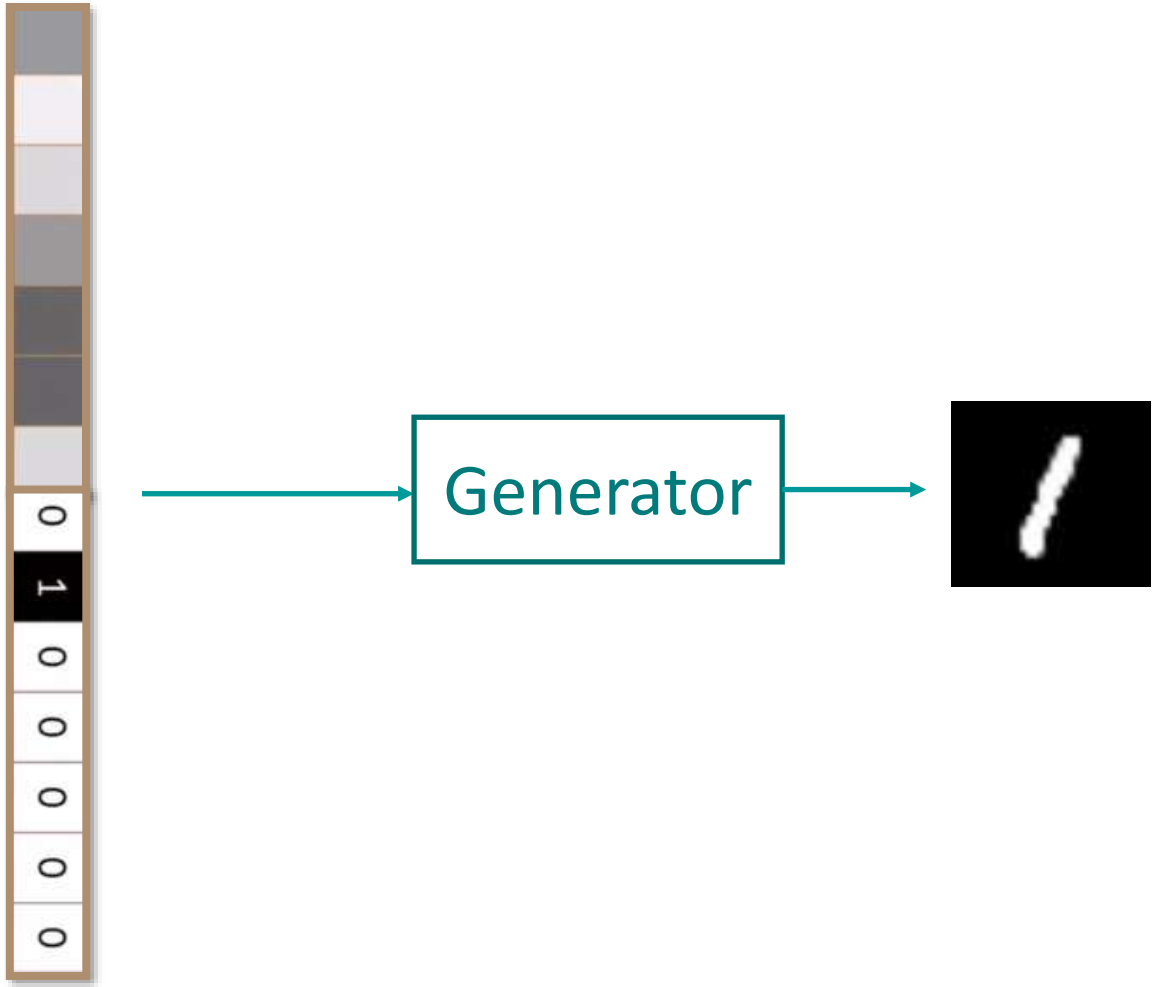
- https://keras.io/examples/generative/wgan_gp/

Conditional Generation

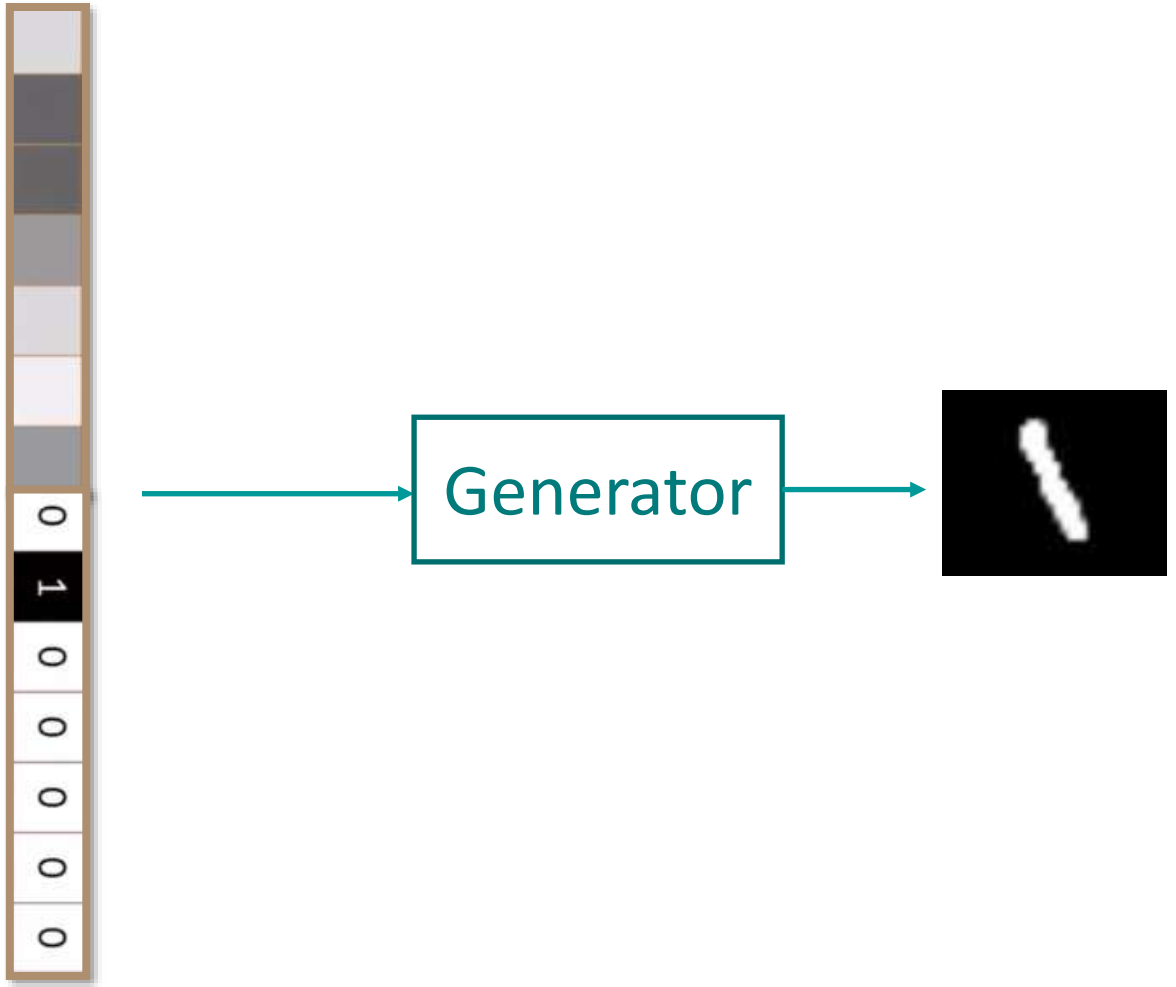
Conditional Generation: Inputs



Conditional Generation: Inputs



Conditional Generation: Inputs



Discriminator Input

Discriminator

Output
3% real

5

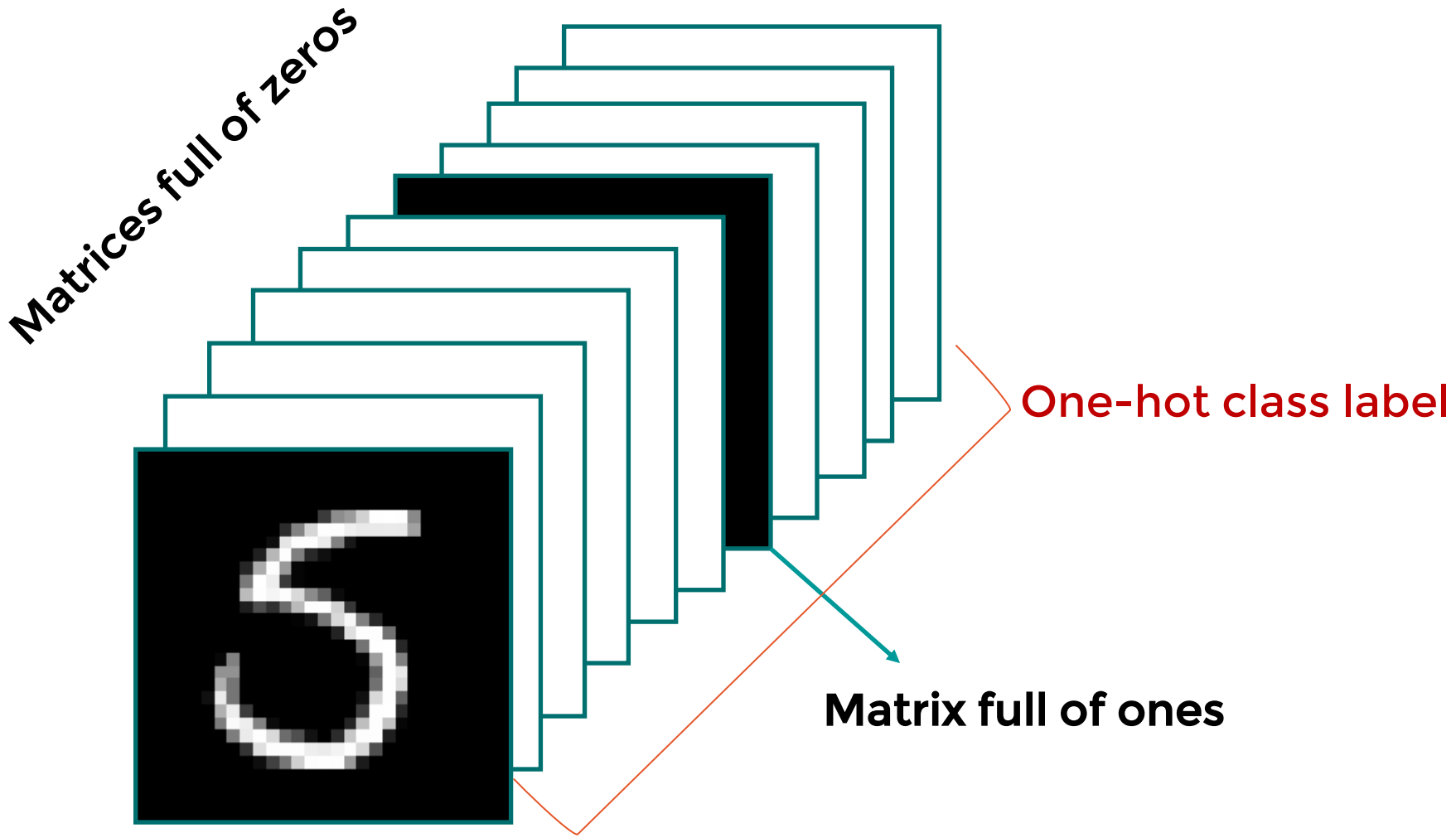
Discriminator Input

Discriminator

Output
97% real

5

Discriminator Input



Summary

- The class is passed to the generator as one-hot vectors
- The class is passed to the discriminator as one-hot matrices
- The size of the vector and the number of matrices represent the number of classes

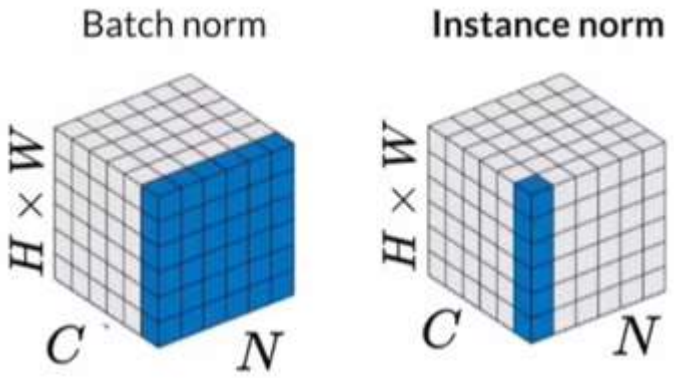
Implementation…

- https://keras.io/examples/generative/conditional_gan/

AdaIN

Instance Norm

$$\frac{x_i - \mu(x_i)}{\delta(x_i)}$$



AdaIN

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

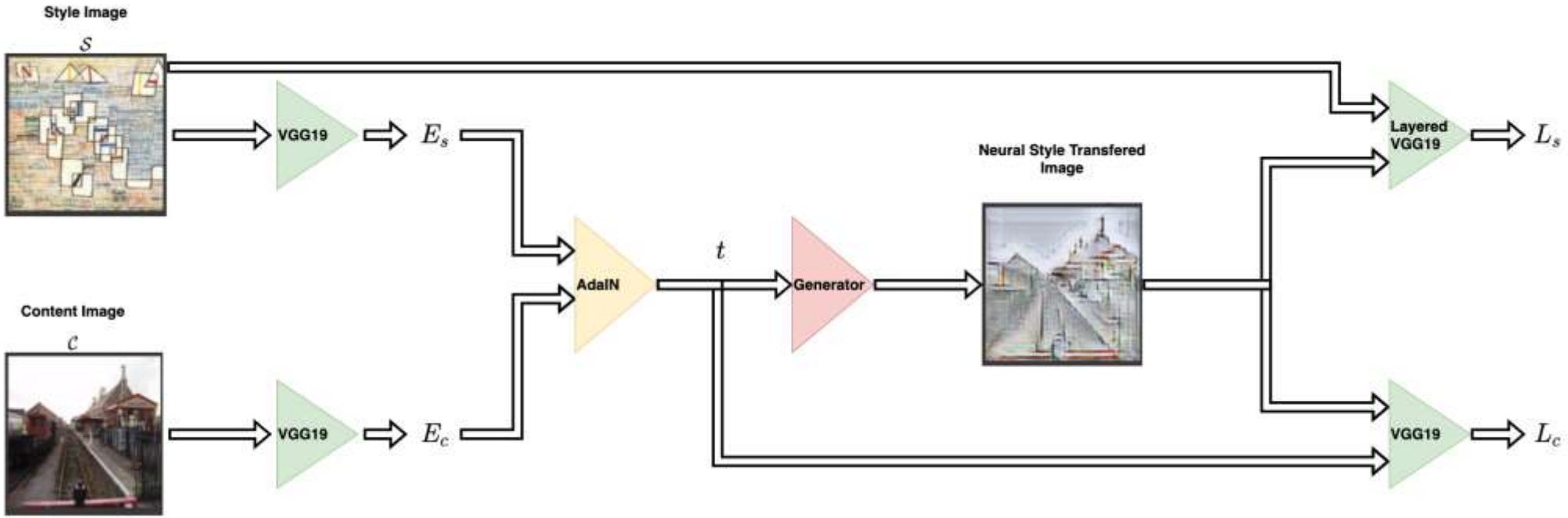
Step 1: Instance Normalization

AdaIN

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

Step 2: Adaptive styles

Architecture



Loss

$$\mathcal{L}_t = \mathcal{L}_c + \lambda \mathcal{L}_s$$

Loss

$$\mathcal{L}_t = \mathcal{L}_c + \lambda \mathcal{L}_s$$

$$\mathcal{L}_c = ||f(g(t)) - t||_2$$

Loss

$$\mathcal{L}_t = \mathcal{L}_c + \lambda \mathcal{L}_s$$

$$\mathcal{L}_s = \sum_{i=1}^{L} ||\mu(\phi_i(g(t))) - \mu(\phi_i(s))||_2 + \sum_{i=1}^{L} ||\sigma(\phi_i(g(t))) - \sigma(\phi_i(s))||_2$$

Implementation

- https://keras.io/examples/generative/adain/

GAN Improvements

CLASS. VISION

4.5 years of GAN progress on face generation.

https://twitter.com/goodfellow_ian/status/1084973596236144640

Main improvements:
(1) Stability

High std

Low std

Main improvements:
(1) Stability

High std

Low std

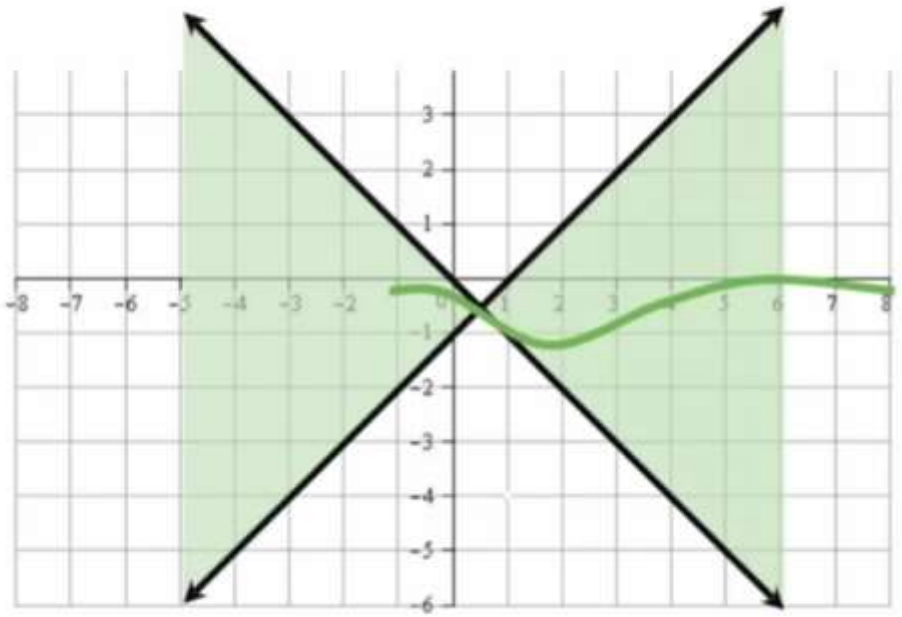
Use standard deviation in batch to encourage diversity.

Main improvements:
(1) Stability

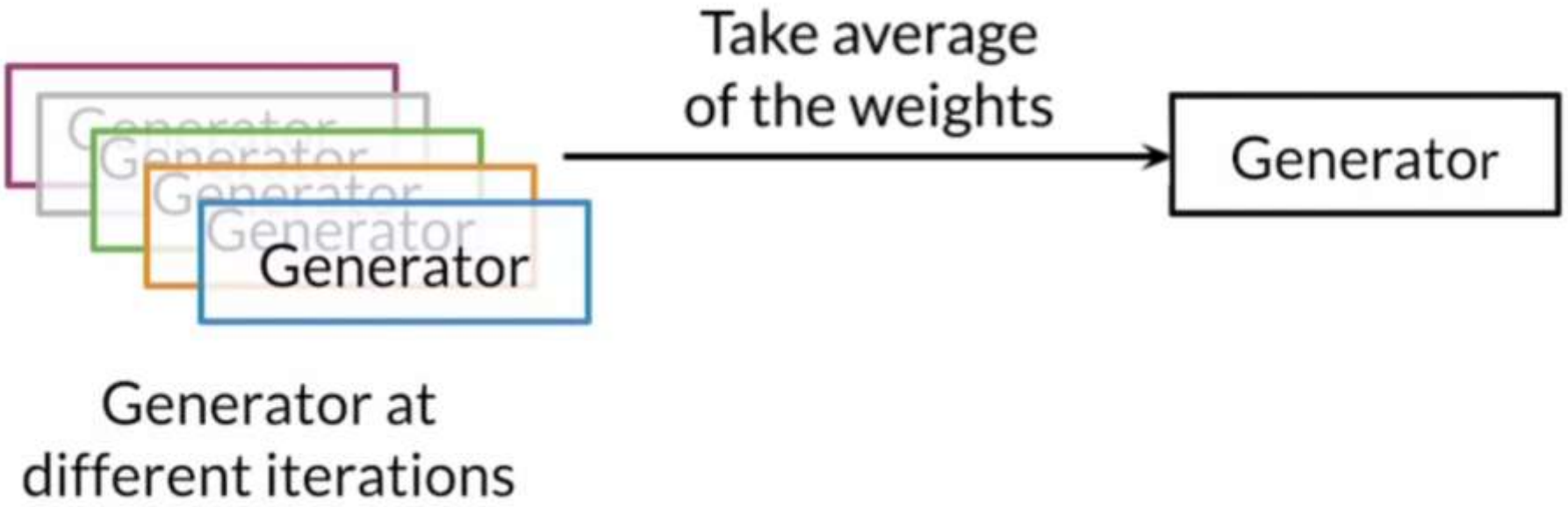
Improve stability by enforcing 1-Lipschitz continuity

E.g. WGAN-GP and Spectral Normalization

∇ : gradient



Main improvements:
(1) Stability



Main improvements:
(1) Stability

Use moving average for smoother results

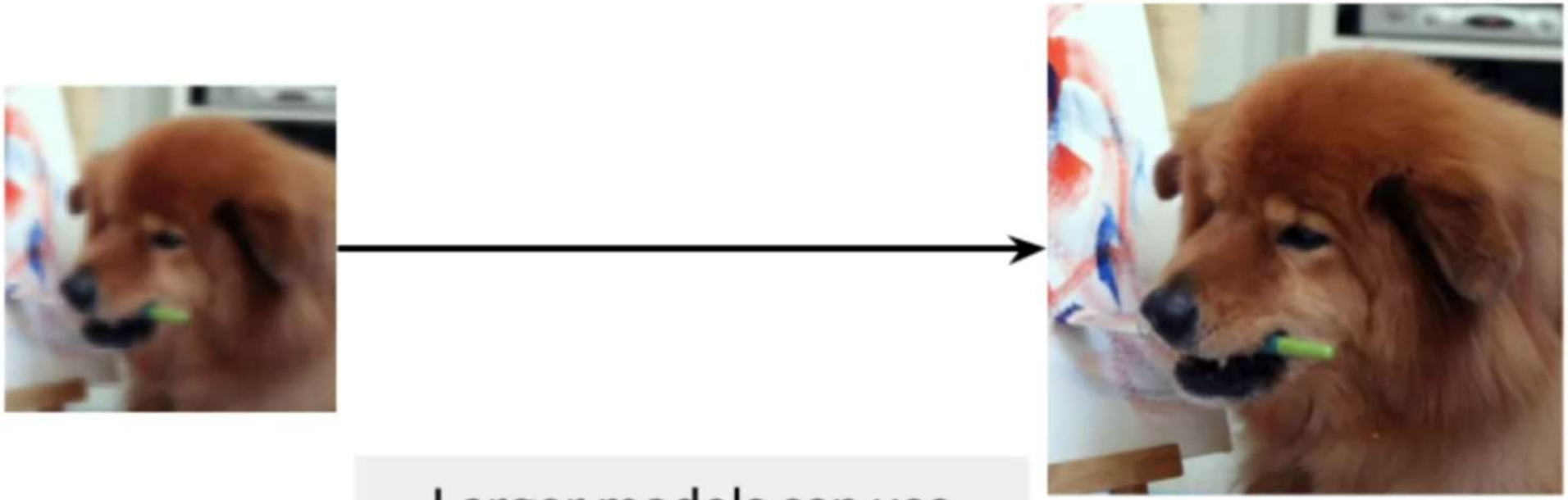
The Unusual Effectiveness of Averaging in GAN Training:
https://arxiv.org/pdf/1806.04498v2.pdf

Main improvements:
(1) Stability

Progressive growing gradually trains GAN at increasing resolutions

Progressive Growing of GANs for Improved Quality, Stability, and Variation:
https://arxiv.org/pdf/1710.10196.pdf

Main improvements:
(2) Capacity



Larger models can use higher resolution images

Main improvements:
(2) Diversity

A Style-Based Generator Architecture for Generative Adversarial Networks
https://arxiv.org/pdf/1812.04948.pdf

Summary

- GANs have improved because of:
  - **Stability** - longer training and better images
  - **Capacity** - larger models and higher resolution images
  - **Diversity** - increasing variety in generated images

Progressive Growing

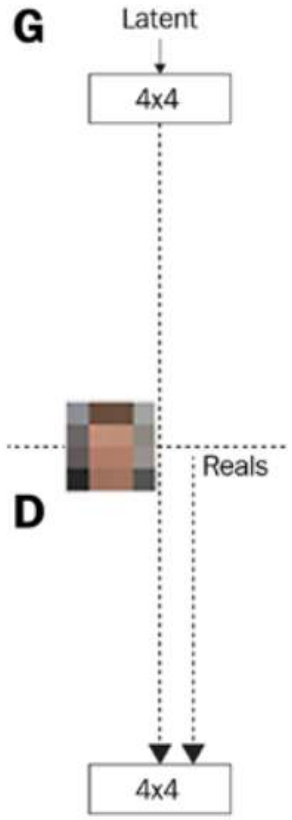
ProGAN

https://arxiv.org/pdf/1710.10196.pdf

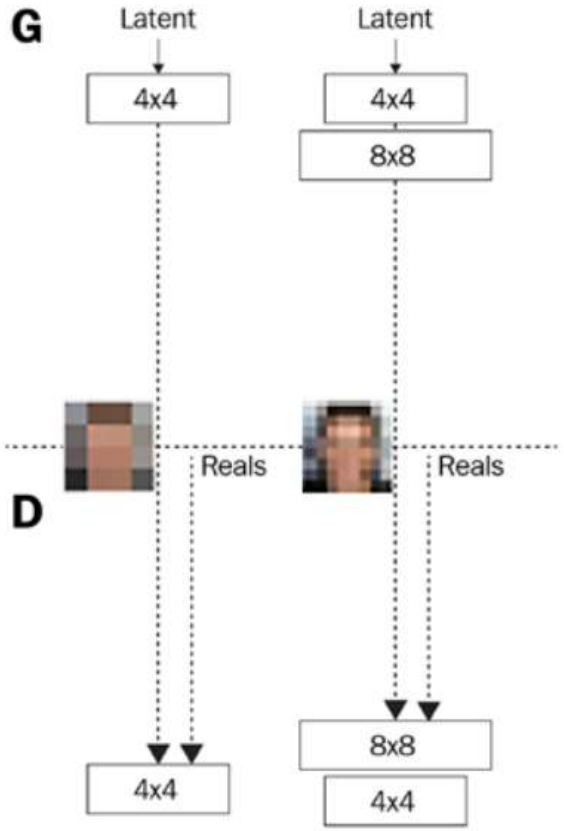
ProGAN

- Progressive growing
- Pixel normalization
- Minibatch statistics
- Equalized learning rate

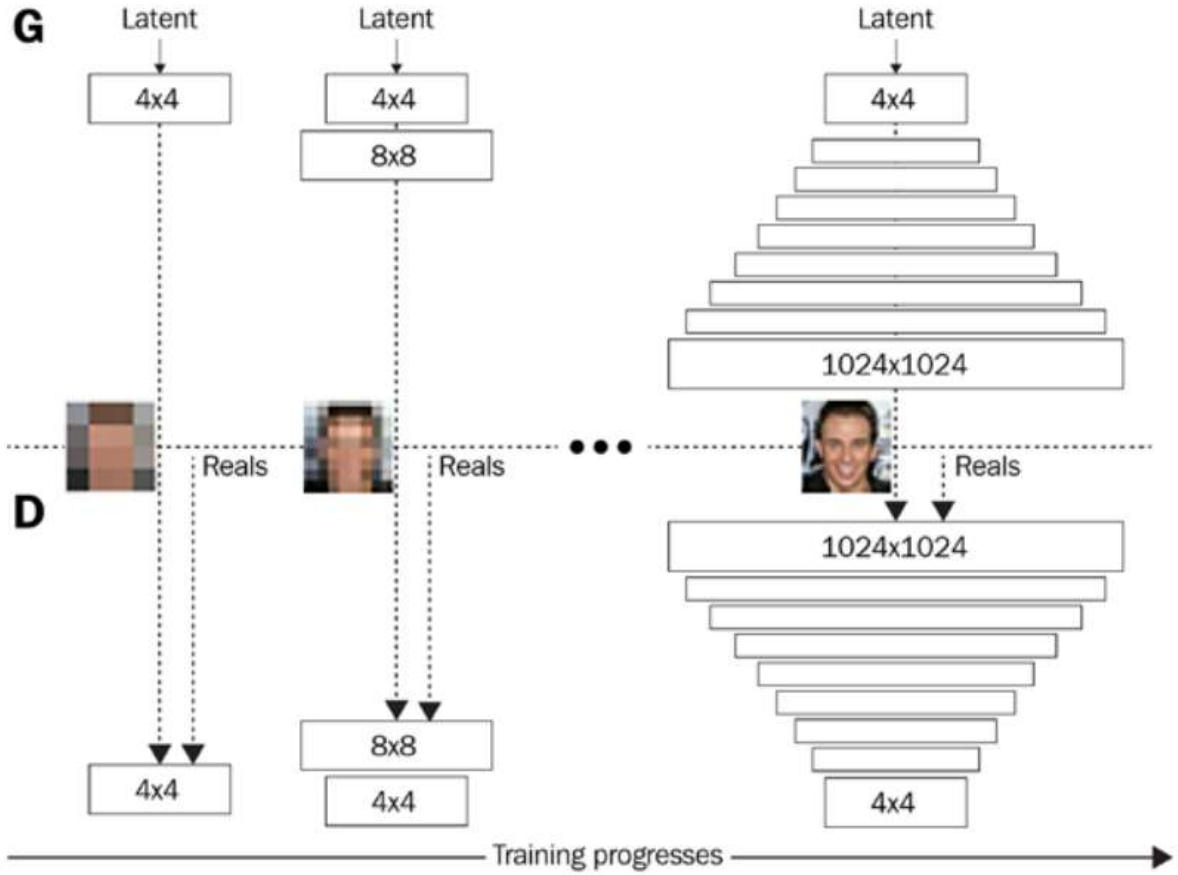
Progressive growing



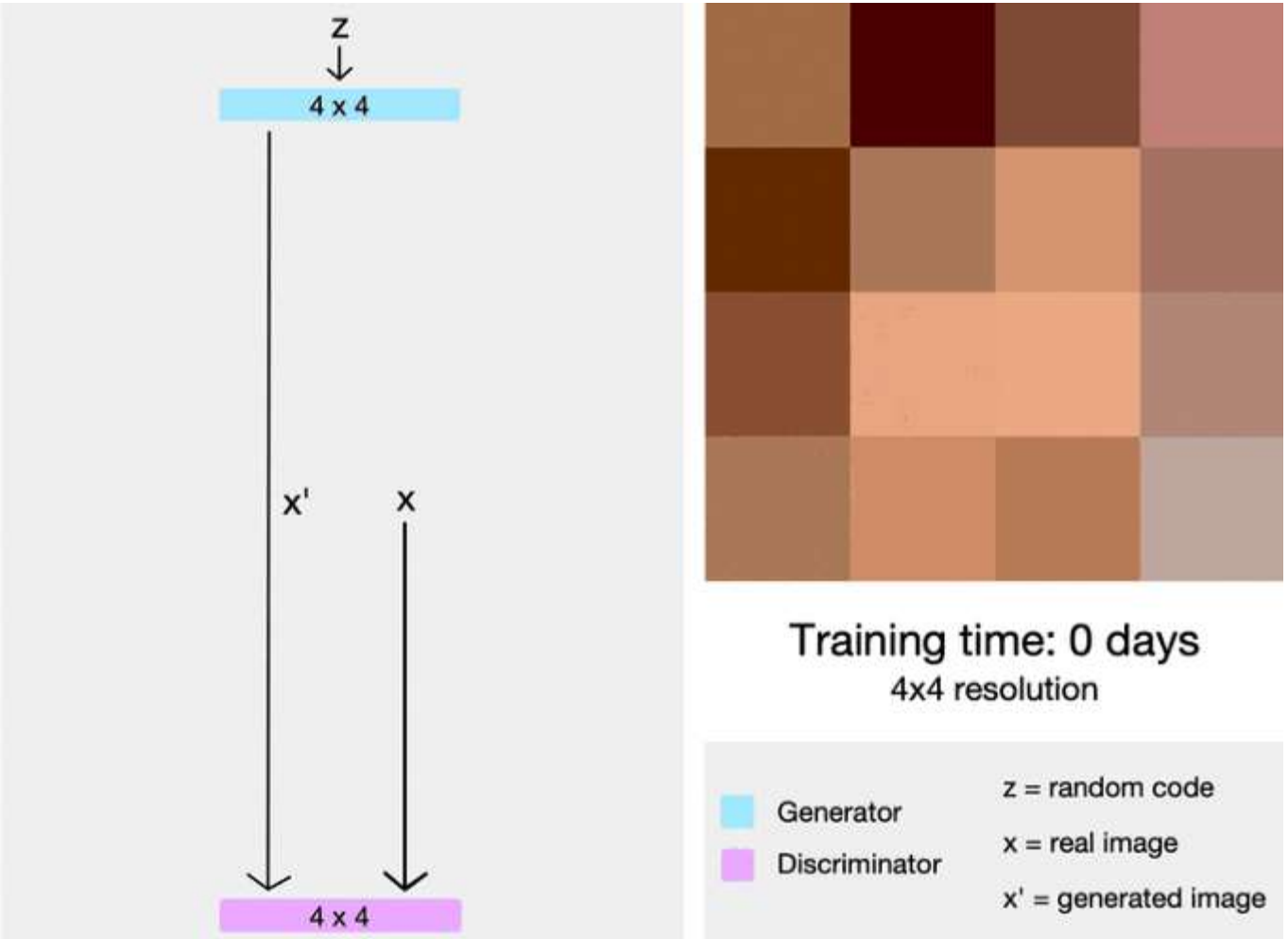
Progressive growing



Progressive growing



Progressive Growing in Action



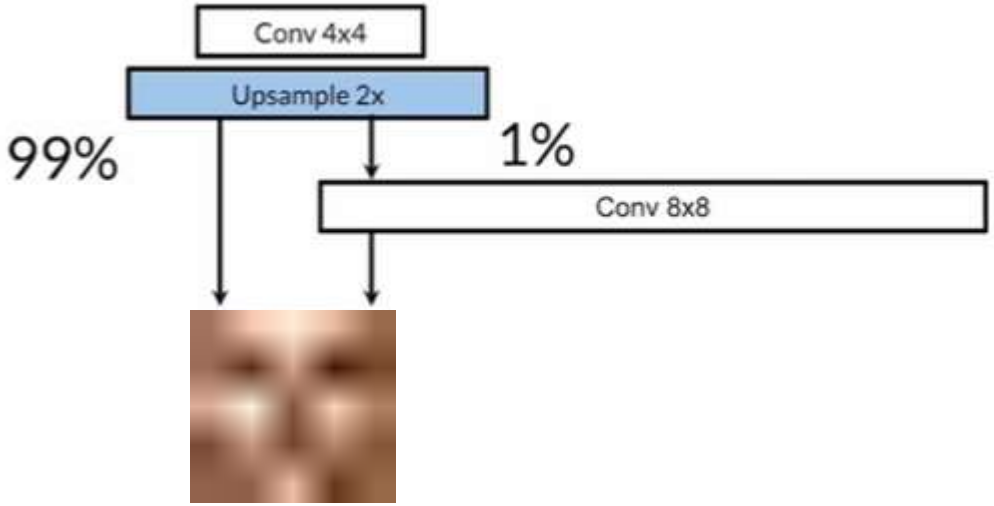
The **ProGAN** starts out generating very low resolution images. When training stabilizes, a new layer is added and the resolution is doubled. This continues until the output reaches the desired resolution. By progressively growing the networks in this fashion, high-level structure is learned first, and training is stabilized.

Source: https://towardsdatascience.com/progan-how-nvidia-generated-images-of-unprecedented-quality-51c98ec2cbd2

Progressive Growing in Action

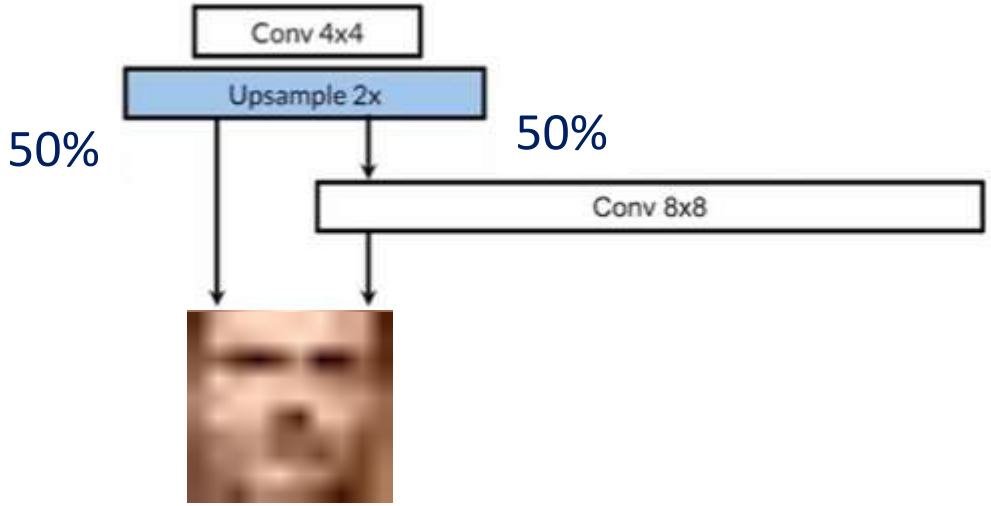
https://www.gwern.net/images/gan/stylegan/2019-03-16-stylegan-facestraining.mp4

Progressive Growing: Generator



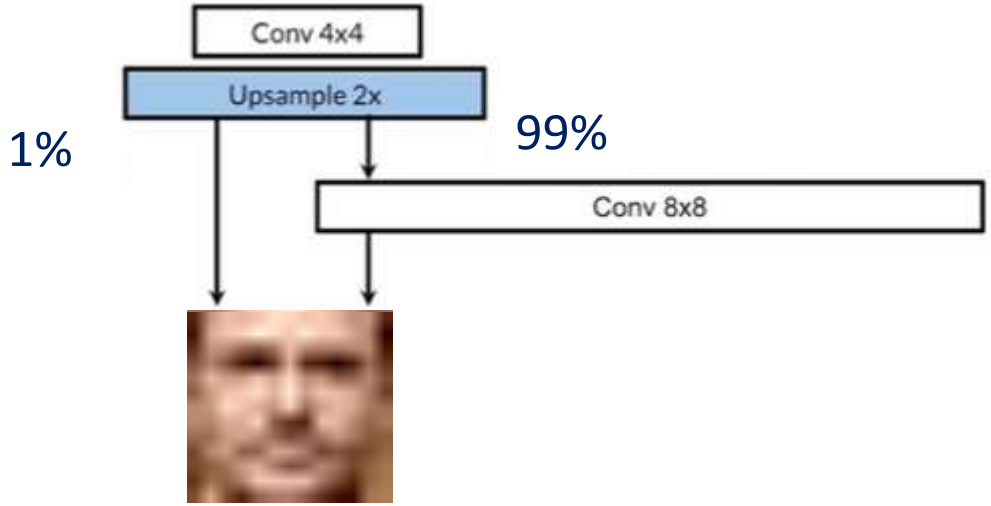
Source: https://bit.ly/2TggMR8

Progressive Growing: Generator



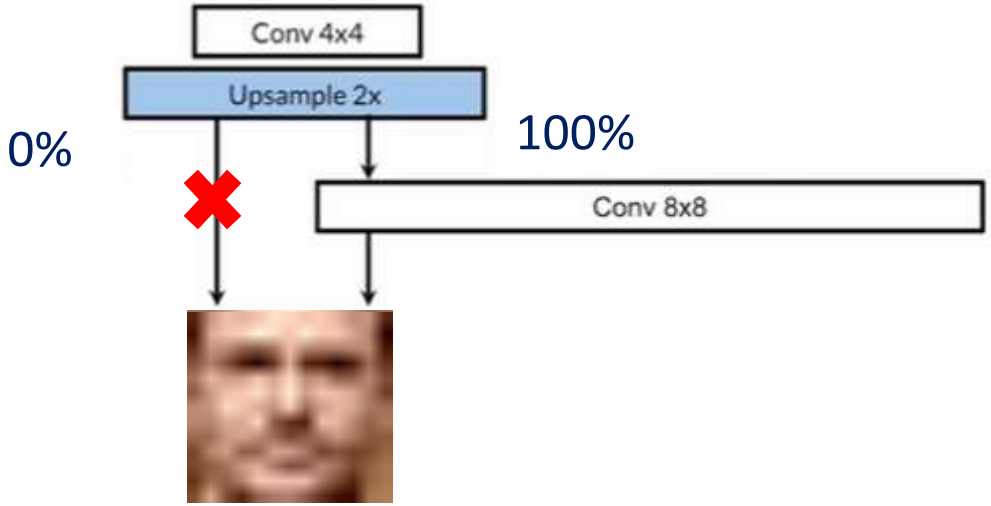
Source: https://bit.ly/2TggMR8

Progressive Growing: Generator



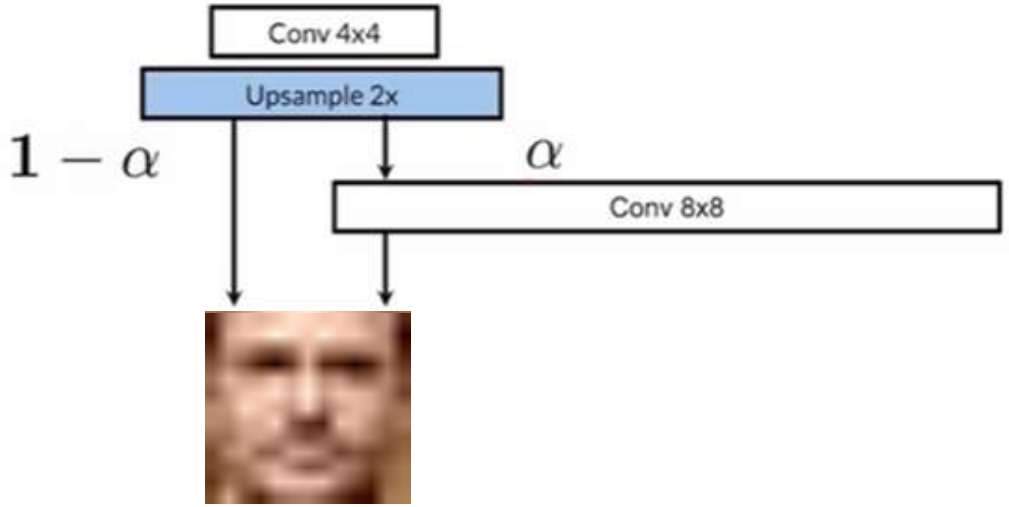
Source: https://bit.ly/2TggMR8

Progressive Growing: Generator



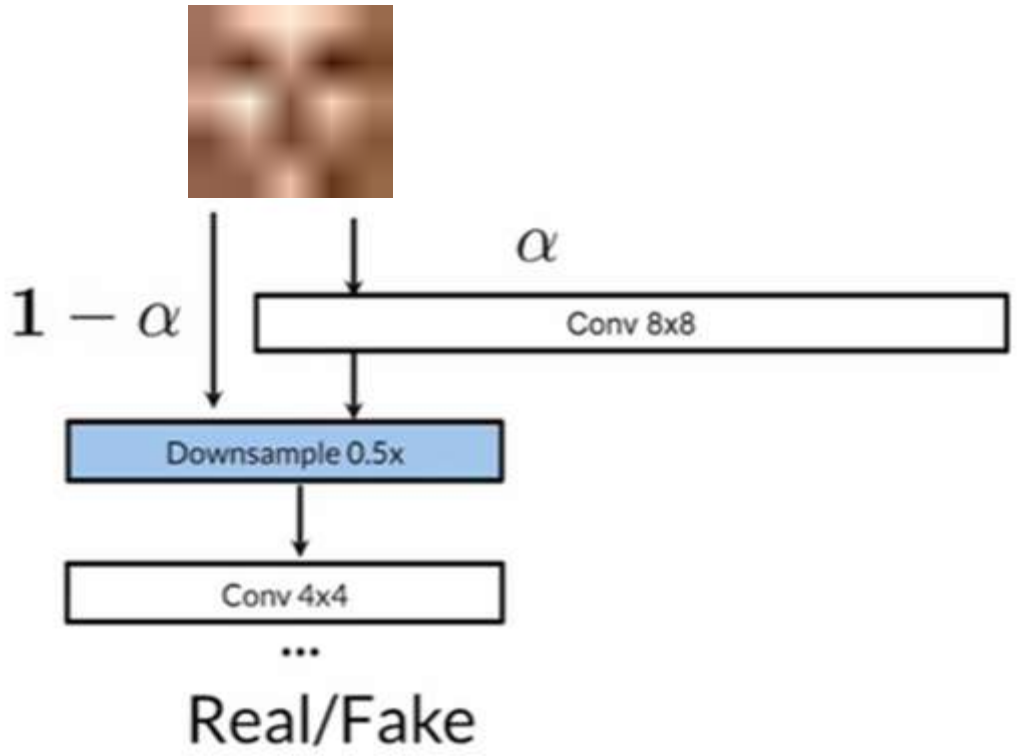
Source: https://bit.ly/2TggMR8

Progressive Growing: Generator



Source: https://bit.ly/2TggMR8

Progressive Growing: Discriminator



Source: https://bit.ly/2TggMR8

Progressive Growing: FadeIn

```
class FadeIn(Layer):
    @tf.function
    def call(self, input_alpha, a, b):
        alpha = tf.reduce_mean(input_alpha)
        y = alpha * a + (1. - alpha) * b
        return y
```

Progressive Growing: FadeIn

```
class FadeIn(Layer):
    @tf.function
    def call(self, input_alpha, a, b):
        alpha = tf.reduce_mean(input_alpha)
        y = alpha * a + (1. - alpha) * b
        return y
```

Why reduce_mean?!

Progressive Growing: FadeIn

```
class FadeIn(Layer):
    @tf.function
    def call(self, input_alpha, a, b):
        alpha = tf.reduce_mean(input_alpha)
        y = alpha * a + (1. - alpha) * b
        return y
```

Why reduce_mean?!

alpha is an input!!

```
x = FadeIn()(alpha, x, y)
```


CLASS. VISION

Progressive Growing: FadeIn

```
class FadeIn(Layer):
    @tf.function
    def call(self, input_alpha, a, b):
        alpha = tf.reduce_mean(input_alpha)
        y = alpha * a + (1. - alpha) * b
        return y
```

Why reduce_mean?!

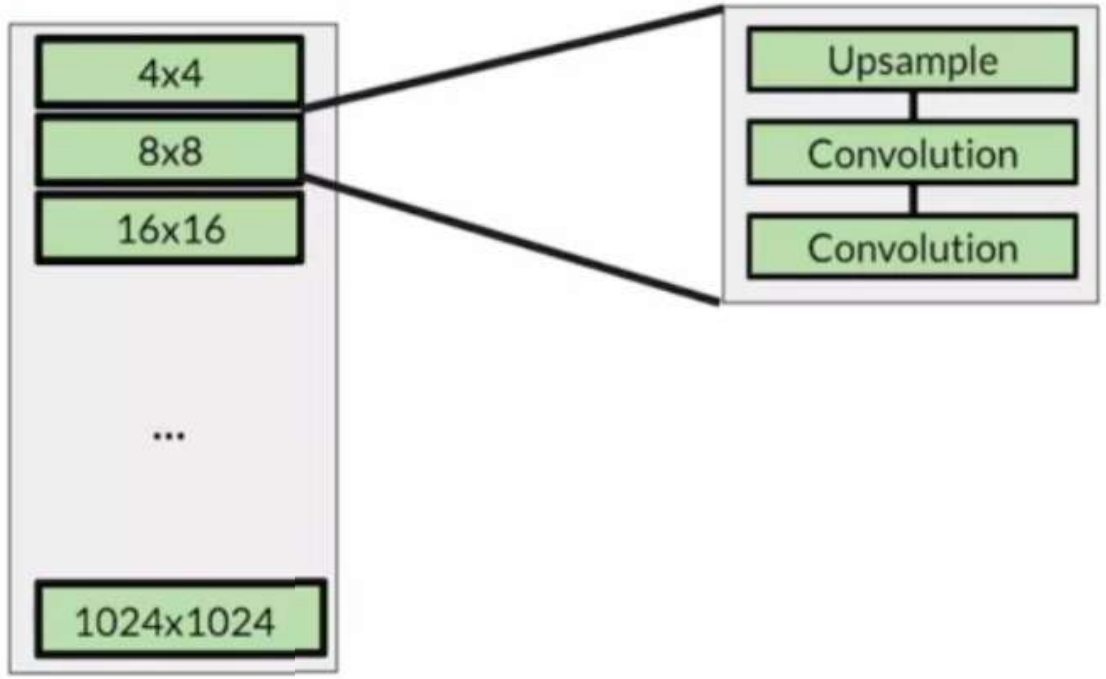
alpha is an input!!

```
x = FadeIn()(alpha, x, y)
```

In this implementation, alpha is an input to our model.
As the model input is assumed to be a minibatch, **tf.reduce_mean()** in **FadeIN()** is meant to convert the batched value to a scalar value.

Progressive Growing in Context

Progressive Growing in Context



Progressive Growing

The implementation uses two functions that build the blocks to convert into/from RGB images.

- Both blocks use a 1x1 convolutional layer
- **to_rgb** block uses a filter size of 3 to match the RGB channels
- **from_rgb** blocks use a filter size that matches the input activation of the discriminator block at that scale.

Progressive Growing

```
def build_to_rgb(self, res, filter_n):
    return Sequential([Input(shape=(res, res, filter_n)),
                       Conv2D(3, 1, gain=1,
                        activation='tanh')])
def build_from_rgb(self, res, filter_n):
    return Sequential([Input(shape=(res, res, 3)),
                       Conv2D(filter_n, 1),
                       LeakyReLU(0.2)])
```

Progressive Growing: Generator

Now, we can look at the following steps to grow the generator to 32x32 (in this example!)

1. Add a 4x4 generator
   the input is a latent vector
2. In a loop, add subsequent generators of gradually increasing resolutions until the one before the target resolution
   in our example, 16x16
3. Add `to_rgb` from 16x16 and upsample it to 32x32.
4. Add the 32x32 generator block.
5. Fade in the two images to create one final RGB image.

```
def grow_generator(self, log2_res):
    res = 2**log2_res
    alpha = Input(shape=(1))

    x = self.generator_blocks[2].input

    for i in range(2, log2_res):
        x = self.generator_blocks[i](x)

    old_rgb = self.to_rgb[log2_res-1](x)
    old_rgb = UpSampling2D((2,2))(old_rgb)

    x = self.generator_blocks[log2_res](x)
    new_rgb = self.to_rgb[log2_res](x)
    rgb = FadeIn()(alpha, new_rgb, old_rgb)
    self.generator = Model([self.generator_blocks[2].input, alpha],
                           rgb,
                           name=f'generator_{res}_x_{res}')
```

```
def grow_generator(self, log2_res):
    res = 2**log2_res
    alpha = Input(shape=(1))

    x = self.generator_blocks[2].input

    for i in range(2, log2_res):
        x = self.generator_blocks[i](x)

    old_rgb = self.to_rgb[log2_res-1](x)
    old_rgb = UpSampling2D((2,2))(old_rgb)

    x = self.generator_blocks[log2_res](x)
    new_rgb = self.to_rgb[log2_res](x)
    rgb = FadeIn()(alpha, new_rgb, old_rgb)
    self.generator = Model([self.generator_blocks[2].input, alpha],
                           rgb,
                           name=f'generator_{res}_x_{res}')
```

```
def grow_generator(self, log2_res):
    res = 2**log2_res
    alpha = Input(shape=(1))

    x = self.generator_blocks[2].input

    for i in range(2, log2_res):
        x = self.generator_blocks[i](x)

    old_rgb = self.to_rgb[log2_res-1](x)
    old_rgb = UpSampling2D((2,2))(old_rgb)

    x = self.generator_blocks[log2_res](x)
    new_rgb = self.to_rgb[log2_res](x)
    rgb = FadeIn()(alpha, new_rgb, old_rgb)
    self.generator = Model([self.generator_blocks[2].input, alpha],
                           rgb,
                           name=f'generator_{res}_x_{res}')
```

```python
def build_all_generators(self):
    # build all the generator block
    self.to_rgb = {}
    self.generator_blocks = {}
    self.generator_blocks[2] = self.build_generator_base(self.z_dim)
    self.to_rgb[2] = self.build_to_rgb(4, self.log2_res_to_filter_size[2])

    for log2_res in range(3, self.log2_resolution+1):
        res = 2**log2_res
        filter_n = self.log2_res_to_filter_size[log2_res]
        self.to_rgb[log2_res] = self.build_to_rgb(res, filter_n)

        input_shape = self.generator_blocks[log2_res-1].output[0].shape
        gen_block = self.build_generator_block(log2_res, input_shape)
        self.generator_blocks[log2_res] = gen_block
```

```python
def build_all_generators(self):
    # build all the generator block
    self.to_rgb = {}
    self.generator_blocks = {}
    self.generator_blocks[2] = self.build_generator_base(self.z_dim)
    self.to_rgb[2] = self.build_to_rgb(4, self.log2_res_to_filter_size[2])

    for log2_res in range(3, self.log2_resolution+1):
        res = 2**log2_res
        filter_n = self.log2_res_to_filter_size[log2_res]
        self.to_rgb[log2_res] = self.build_to_rgb(res, filter_n)

        input_shape = self.generator_blocks[log2_res-1].output[0].shape
        gen_block = self.build_generator_block(log2_res, input_shape)
        self.generator_blocks[log2_res] = gen_block
```

```
def build_generator_base(self, input_shape):

    input_tensor = Input(shape=input_shape)
    x = PixelNorm()(input_tensor)
    x = Dense(8192, gain=1./8)(x)
    x = Reshape((4, 4, 512))(x)
    x = LeakyReLU(0.2)(x)
    x = PixelNorm()(x)

    x = Conv2D(512, 3,
               name='gen_4x4_conv1')(x)
    x = LeakyReLU(0.2)(x)
    x = PixelNorm()(x)

    return Model(input_tensor, x,
                 name='generator base')
```

The input is a latent vector

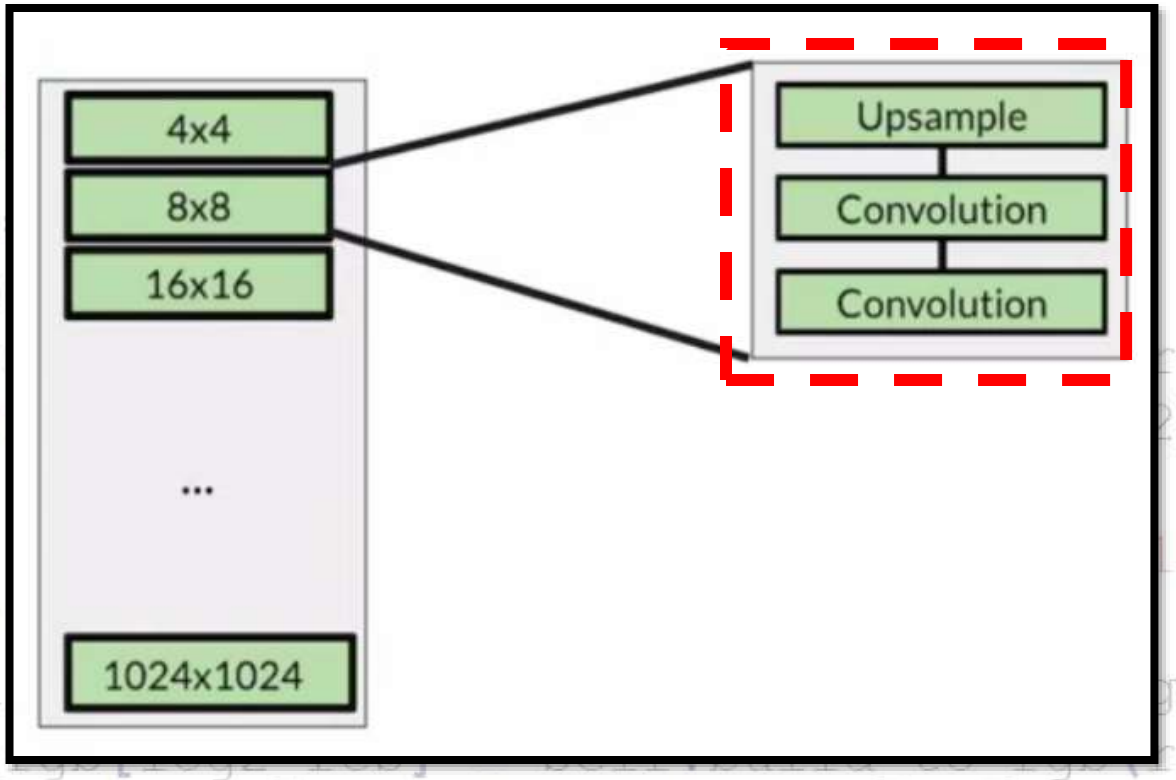
```python
def build_all_generators(self):
    # build all the generator block
    self.to_rgb = {}
    self.generator_blocks = {}
    self.generator_blocks[2] = self.build_generator_base(self.z_dim)
    self.to_rgb[2] = self.build_to_rgb(4, self.log2_res_to_filter_size[2])

    for log2_res in range(3, self.log2_resolution+1):
        res = 2**log2_res
        filter_n = self.log2_res_to_filter_size[log2_res]
        self.to_rgb[log2_res] = self.build_to_rgb(res, filter_n)

        input_shape = self.generator_blocks[log2_res-1].output[0].shape
        gen_block = self.build_generator_block(log2_res, input_shape)
        self.generator_blocks[log2_res] = gen_block
```

```
self.log2_res_to_filter_size = {
    0: 512,
    1: 512,
    2: 512, # 4x4
    3: 512, # 8x8
    4: 512, # 16x16
    5: 512, # 32x32
    6: 256, # 64x64
    7: 128, # 128x128
    8: 64,  # 256x256
    9: 32,  # 512x512
    10: 16} # 1024x1024
```

```python
def build_all_generators(self):
    # build all the generator block
    self.to_rgb = {}
    self.generator_blocks = {}
    self.generator_blocks[2] = self.build_generator_base(self.z_dim)
    self.to_rgb[2] = self.build_to_rgb(4, self.log2_res_to_filter_size[2])

    for log2_res in range(3, self.log2_resolution+1):
        res = 2**log2_res
        filter_n = self.log2_res_to_filter_size[log2_res]
        self.to_rgb[log2_res] = self.build_to_rgb(res, filter_n)

        input_shape = self.generator_blocks[log2_res-1].output[0].shape
        gen_block = self.build_generator_block(log2_res, input_shape)
        self.generator_blocks[log2_res] = gen_block
```

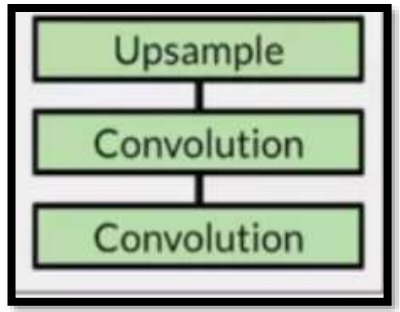
```
def build_all_generators(self):
    # build all the generator block
    self.to_rgb = {}
    self.generator_blocks = {}
    self.generator_blocks[2] = self.build_generator_base(self.z_dim)
    self.to_                                                    ilter_size[2])

    self.log2_resolution = int(np.log2(resolution))

    for log2_res in range(3, self.log2_resolution+1):
        res = 2**log2_res
        filter_n = self.log2_res_to_filter_size[log2_res]
        self.to_rgb[log2_res] = self.build_to_rgb(res, filter_n)

        input_shape = self.generator_blocks[log2_res-1].output[0].shape
        gen_block = self.build_generator_block(log2_res, input_shape)
        self.generator_blocks[log2_res] = gen_block
```

```python
def build_all_generators(self):
    # build all the generator block
    self.to_rgb = {}
    self.generator_blocks = {}
    self.generator_blocks[2] = self.build_generator_base(self.z_dim)
    self.to_rgb[2] = self.build_to_rgb(4, self.log2_res_to_filter_size[2])

    for log2_res in range(3, self.log2_resolution+1):
        res = 2**log2_res
        filter_n = self.log2_res_to_filter_size[log2_res]
        self.to_rgb[log2_res] = self.build_to_rgb(res, filter_n)

        input_shape = self.generator_blocks[log2_res-1].output[0].shape
        gen_block = self.build_generator_block(log2_res, input_shape)
        self.generator_blocks[log2_res] = gen_block
```




```
def build_all_ge
    # build all
    self.to_rgb
    self.generat
    self.generat                                    r_base(self.z_dim)
    self.to_rgb[                                    2_res_to_filter_size[2])

    for log2_res                                    ):
        res = 2*
        filter_n                                    g2_res]
        self.to_                                    res, filter_n)

        input_shape = self.generator_blocks[log2_res-1].output[0].shape
        gen_block = self.build_generator_block(log2_res, input_shape)
        self.generator_blocks[log2_res] = gen_block
```

```python
def build_generator_block(self, log2_res, input_shape):
    res = 2**log2_res
    res_name = f'{res}x{res}'
    filter_n = self.log2_res_to_filter_size[log2_res]

    input_tensor = Input(shape=input_shape)
    x = UpSampling2D((2,2))(input_tensor)

    x = Conv2D(filter_n, 3, name=f'gen_{res_name}_conv1')(x)
    x = LeakyReLU(0.2)(x)
    x = PixelNorm()(x)

    x = Conv2D(filter_n, 3, name=f'gen_{res_name}_conv2')(x)
    x = LeakyReLU(0.2)(x)
    x = PixelNorm()(x)

    return Model(input_tensor, x,
                 name=f'genblock_{res}_x_{res}')
```



Progressive Growing: Discriminator

The growing discriminator is done similarly but in the reverse direction

1. At the resolution of the input image, say, 32x32, add `from_rgb` to the discriminator block of 32x32. The output is the activation with a 16x16 feature map.
2. Parallelly, downsample the input image to 16x16, and add `from_rgb` to the 16x16 discriminator block.
3. Fade in the two preceding features and feed that into the next discriminator block of 8x8.
4. Continue adding the discriminator blocks to the base of 4x4, where the output is a single prediction value.

```
def grow_discriminator(self, log2_res):
    res = 2**log2_res

    input_image = Input(shape=(res, res, 3))
    alpha = Input(shape=(1))

    x = self.from_rgb[log2_res](input_image)
    x = self.discriminator_blocks[log2_res](x)

    downsized_image = AveragePooling2D((2,2))(input_image)
    y = self.from_rgb[log2_res-1](downsized_image)

    x = FadeIn()(alpha, x, y)
    for i in range (log2_res-1, 1, -1):
        x = self.discriminator_blocks[i](x)

    self.discriminator =  Model([input_image, alpha], x,
                          name=f'discriminator_{res}_x_{res}')

    self.optimizer_discriminator = Adam(**self.opt_init)
```

```
def grow_discriminator(self, log2_res):
    res = 2**log2_res

    input_image = Input(shape=(res, res, 3))
    alpha = Input(shape=(1))

    x = self.from_rgb[log2_res](input_image)
    x = self.discriminator_blocks[log2_res](x)

    downsized_image = AveragePooling2D((2,2))(input_image)
    y = self.from_rgb[log2_res-1](downsized_image)

    x = FadeIn()(alpha, x, y)
    for i in range (log2_res-1, 1, -1):
        x = self.discriminator_blocks[i](x)

    self.discriminator =  Model([input_image, alpha], x,
                          name=f'discriminator_{res}_x_{res}')

    self.optimizer_discriminator = Adam(**self.opt_init)
```

```
def build_all_discriminators(self):

    self.from_rgb = {}
    self.discriminator_blocks = {}

    # all but the final block
    for log2_res in range(self.log2_resolution, 1, -1):
        res = 2**log2_res
        filter_n = self.log2_res_to_filter_size[log2_res]
        self.from_rgb[log2_res] = self.build_from_rgb(res, filter_n)

        input_shape = (res, res, filter_n)
        self.discriminator_blocks[log2_res] = self.build_discriminator_block(
        log2_res, input_shape)

    # last block at 4x4 resolution
    log2_res = 2
    filter_n = self.log2_res_to_filter_size[log2_res]
    self.from_rgb[log2_res] = self.build_from_rgb(4, filter_n)
    res = 2**log2_res
    input_shape = (res, res, filter_n)
    self.discriminator_blocks[log2_res] = self.build_discriminator_base(input_shape)
```

```
def build_all_discriminators(self):

    self.from_rgb = {}
    self.discriminator_blocks = {}

    # all but the final block
    for log2_res in range(self.log2_resolution, 1, -1):
        res = 2**log2_res
        filter_n = self.log2_res_to_filter_size[log2_res]
        self.from_rgb[log2_res] = self.build_from_rgb(res, filter_n)

        input_shape = (res, res, filter_n)
        self.discriminator_blocks[log2_res] = self.build_discriminator_block(
        log2_res, input_shape)

    # last block at 4x4 resolution
    log2_res = 2
    filter_n = self.log2_res_to_filter_size[log2_res]
    self.from_rgb[log2_res] = self.build_from_rgb(4, filter_n)
    res = 2**log2_res
    input_shape = (res, res, filter_n)
    self.discriminator_blocks[log2_res] = self.build_discriminator_base(input_shape)
```

```python
def build_discriminator_block(self, log2_res, input_shape):

    filter_n = self.log2_res_to_filter_size[log2_res]
    input_tensor = Input(shape=input_shape)

    # First conv
    x = Conv2D(filter_n, 3)(input_tensor)
    x = LeakyReLU(0.2)(x)

    # Second conv + downsample
    filter_n = self.log2_res_to_filter_size[log2_res-1]
    x = Conv2D(filter_n, 3,)(x)
    x = LeakyReLU(0.2)(x)
    x = AveragePooling2D((2,2))(x)

    res = 2**log2_res
    return Model(input_tensor, x,
                 name=f'disc_block_{res}_x_{res}')
```

Summary

- Progressive growing gradually doubles image resolution
- Helps with faster, more stable training for higher resolutions

Pixel normalization

How to define a norm layer!

The purpose of normalization in ProGAN is to limit the weight values to prevent them from growing exponentially.

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

```
class PixelNorm(Layer):
    def __init__(self, epsilon=1e-8):
        super(PixelNorm, self).__init__()
        self.epsilon = epsilon

    def call(self, input_tensor):
        return input_tensor / tf.math.sqrt(
                              tf.reduce_mean(input_tensor**2,
                              axis=-1, keepdims=True) +
                              self.epsilon)
```

Minibatch statistics

Problem!

- **Mode collapse** happens when a GAN generates similar-looking images as it captures only a subset of the variation found in the training data.

Encourage image variation with mini-batch statistics

High std

Low std

Use standard deviation in batch to encourage diversity.

Implementation

The steps to calculate minibatch statistics are as follows:

1. Calculate the standard deviation for each feature in each spatial location over the minibatch.

```
_, group_var = tf.nn.moments(x, axes=(0), keepdims=False)
   group_std = tf.sqrt(group_var + self.epsilon)
```

2. Calculate the average of these standard deviations across the (*H, W, C*) dimensions to arrive at a single scale value.

```
avg_std = tf.reduce_mean(group_std, axis=[1,2,3], keepdims=True)
```

3. Replicate this value across the feature map of (*H, W*) and append it to the activation. As a result, the output activation has a shape of (*N, H, W, C+1*).

```
x = tf.tile(avg_std, [self.group_size, h, w, 1])
tf.concat([input_tensor, x], axis=-1)
```

Implementation

Before calculating the standard deviation, the activation is first split into groups of 4 or the batch size, whichever is lower. To simplify the code, we assume that the batch size is at least 4 during training. The minibatch layer can be inserted anywhere in the discriminator, but it was found to be more effective toward the end, which is the 4x4 layer.

Code…

```
class MinibatchStd(Layer):
    def __init__(self, group_size=4, epsilon=1e-8):

        super(MinibatchStd, self).__init__()
        self.epsilon = epsilon
        self.group_size = group_size

    def call(self, input_tensor):

        n, h, w, c = input_tensor.shape
        x = tf.reshape(input_tensor, [self.group_size, -1, h, w, c])
        group_mean, group_var = tf.nn.moments(x, axes=(0), keepdims=False)
        group_std = tf.sqrt(group_var + self.epsilon)
        avg_std = tf.reduce_mean(group_std, axis=[1,2,3], keepdims=True)
        x = tf.tile(avg_std, [self.group_size, h, w, 1])

        return tf.concat([input_tensor, x], axis=-1)
```

Equalized learning rate

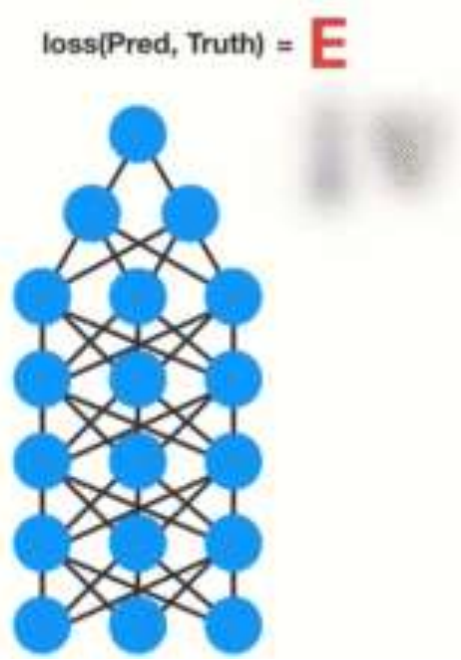
Problem: Vanishing gradient

- vanishing gradient
  - The layers closer to the generator input will receive less gradient

$$w = w - \alpha \frac{\partial L}{\partial w}$$

$$=$$

$$w = w - \alpha \nabla$$



Solution

- What if we want a layer to receive more gradient?

$$\boldsymbol{w} = \boldsymbol{w} - \boldsymbol{m}\,\boldsymbol{\alpha}\,\boldsymbol{\nabla}$$

Solution

- What if we want a layer to receive more gradient?

$$\boldsymbol{w = w - m \alpha \nabla}$$

- Different multiplier constants for different layers to effectively have different learning rates.

CLASS. VISION

He or Kaiming initialization

- In ProGAN, these multiplier constants are calculated from He's initializer

$$std = \sqrt{\frac{2}{fan\ in}}$$

- **Fan in** is the multiplication of the weights dimension except for the output channel.

He or Kaiming initialization

- In ProGAN, these multiplier constants are calculated from He's initializer

$$std = \sqrt{\frac{2}{fan\ in}}$$

- For a convolution weight with the shape (`kernel, kernel, channel_in, channel_out`), the *fan in* is the multiplication of `kernel x kernel x channel_in`.

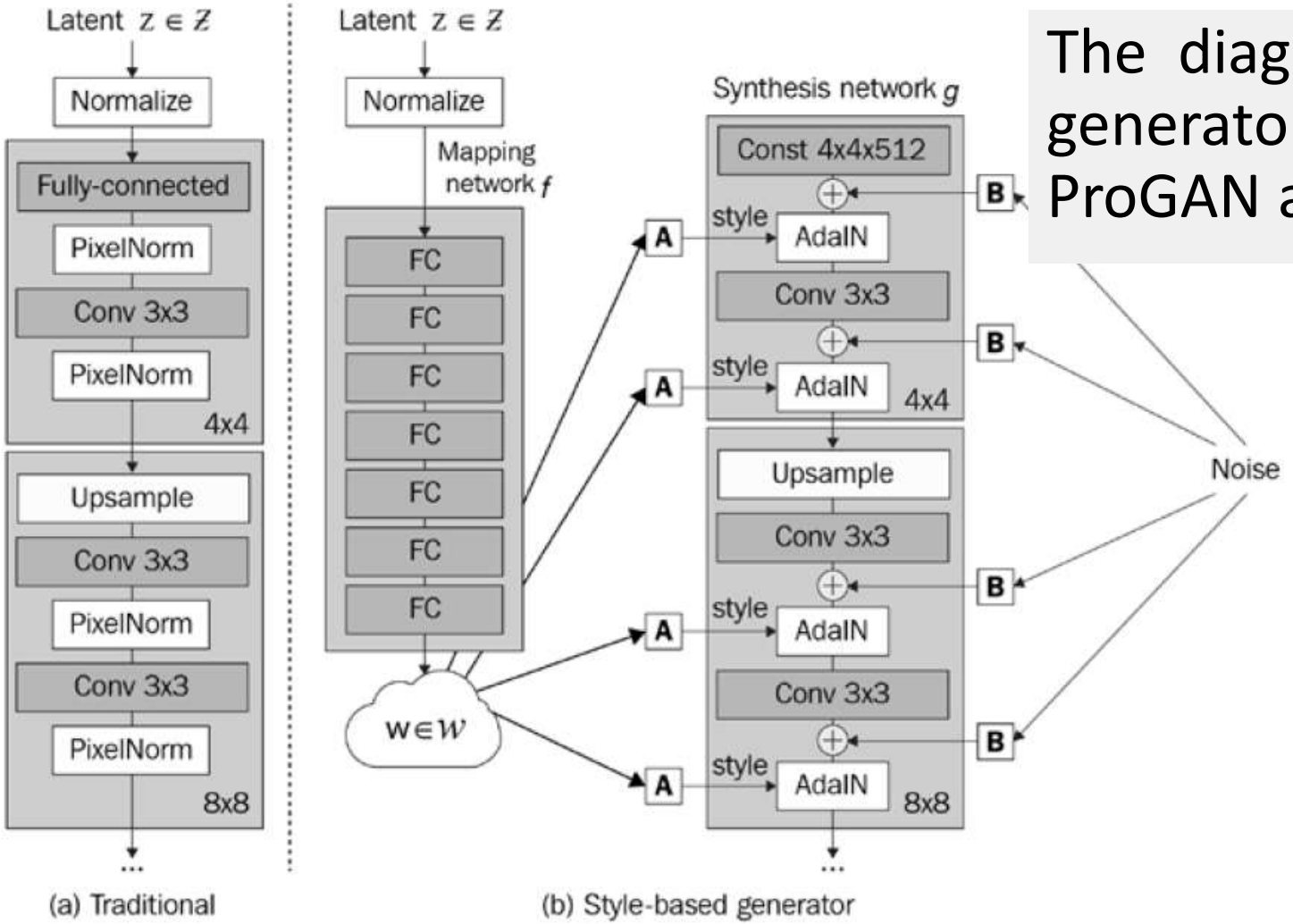
```
fan_in = self.kernel*self.kernel*self.in_channels
self.scale = tf.sqrt(self.gain/fan_in)
```

implementation

- https://colab.research.google.com/github/Alireza-Akhavan/GAN_tutorial/blob/main/progressive_gan.ipynb

StyleGAN

Style-based generator



The diagram compares the generator architectures of ProGAN and StyleGAN

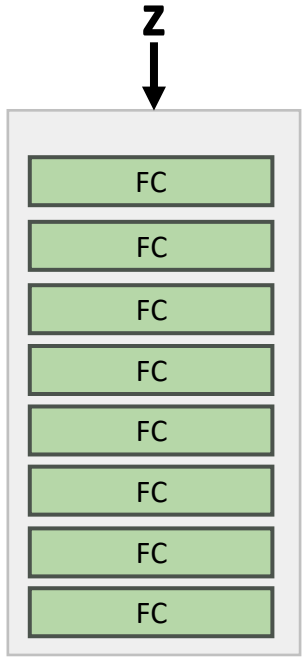
https://arxiv.org/abs/1812.04948

Noise Mapping Network

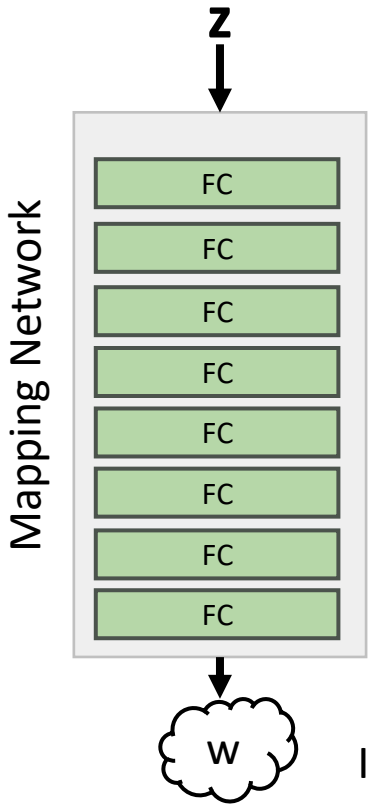
Outline

- Noise mapping network structure
- Motivation behind the noise mapping network
- Where its output W goes

Noise Mapping Network

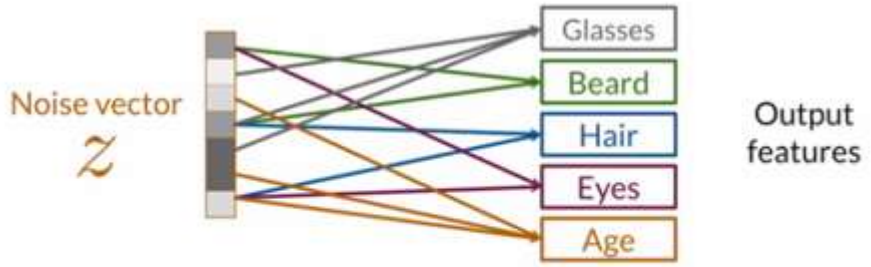


Noise Mapping Network



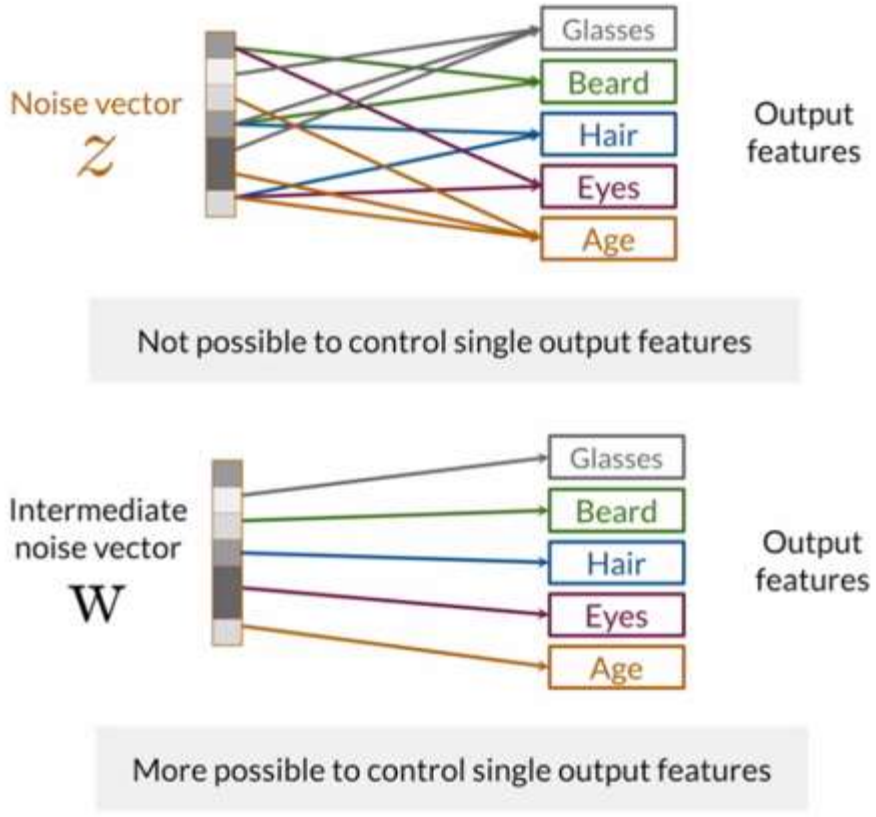
Intermediate noise vector

Z-space Entanglement

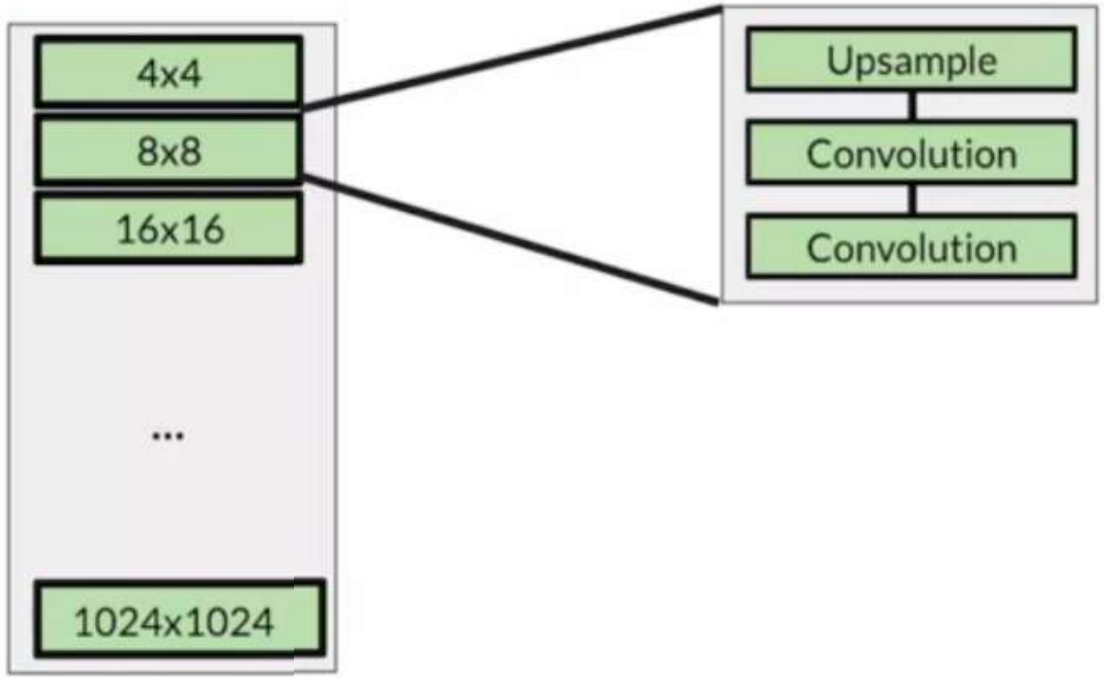


Not possible to control single output features

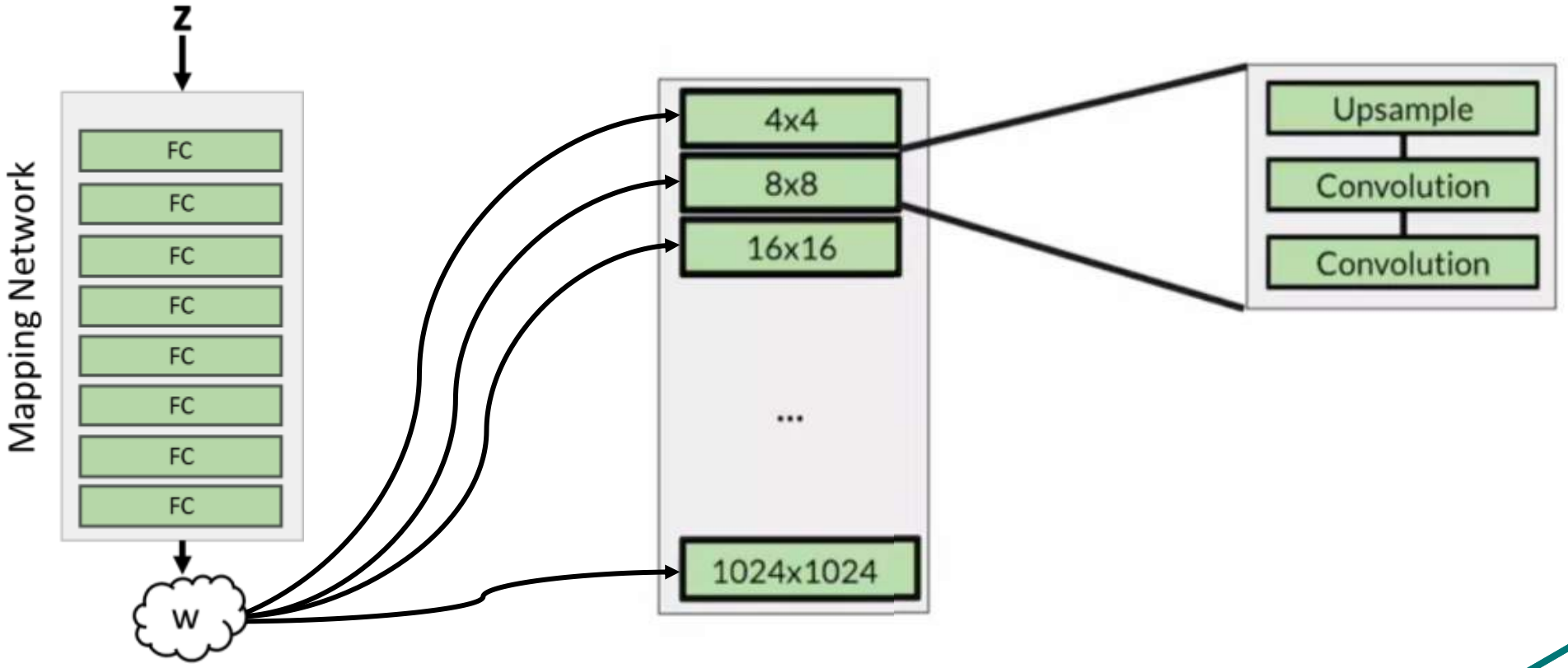
W-space: Less Entangled



Mapping Network in Context



Mapping Network in Context



Summary

- Noise mapping allows for a more disentangled noise space
- The intermediate nose vector is used as input to the generator

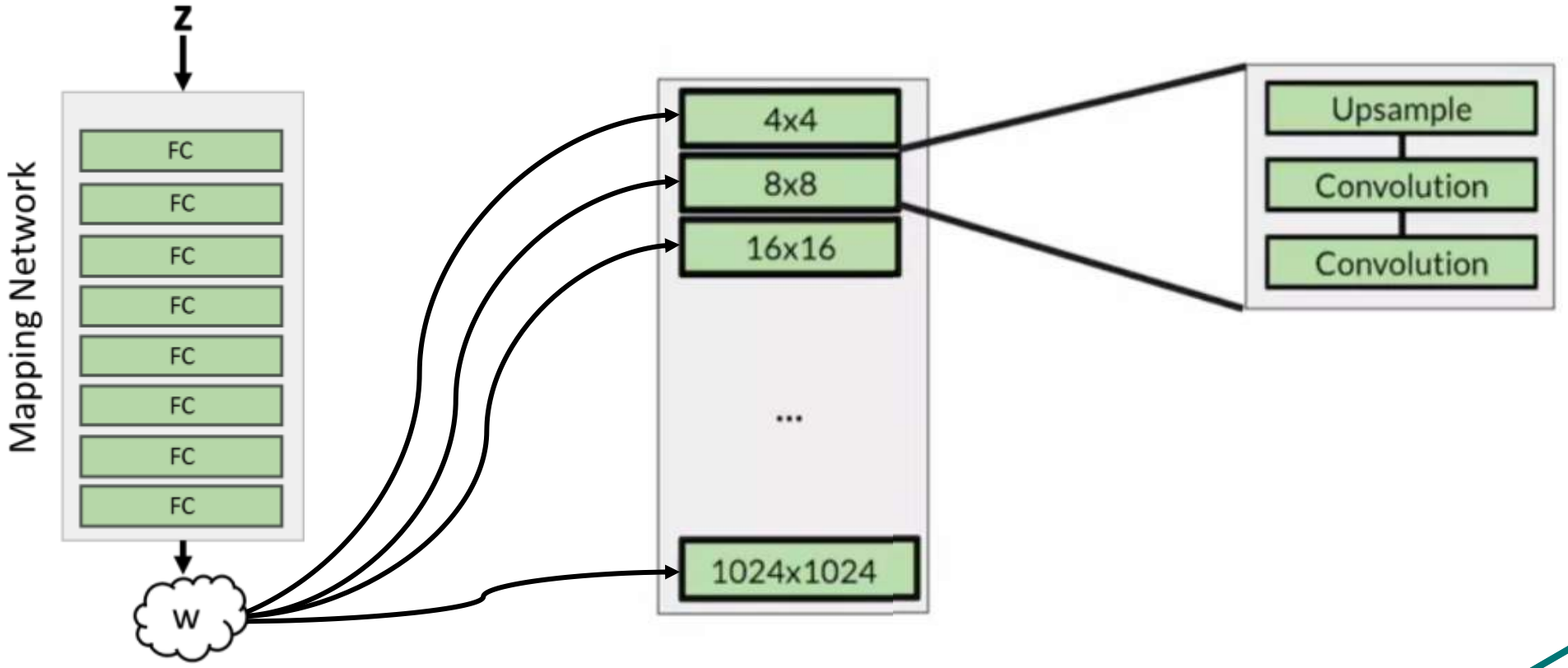
Adaptive Instance Normalization (AdaIN)

CLASS. VISION

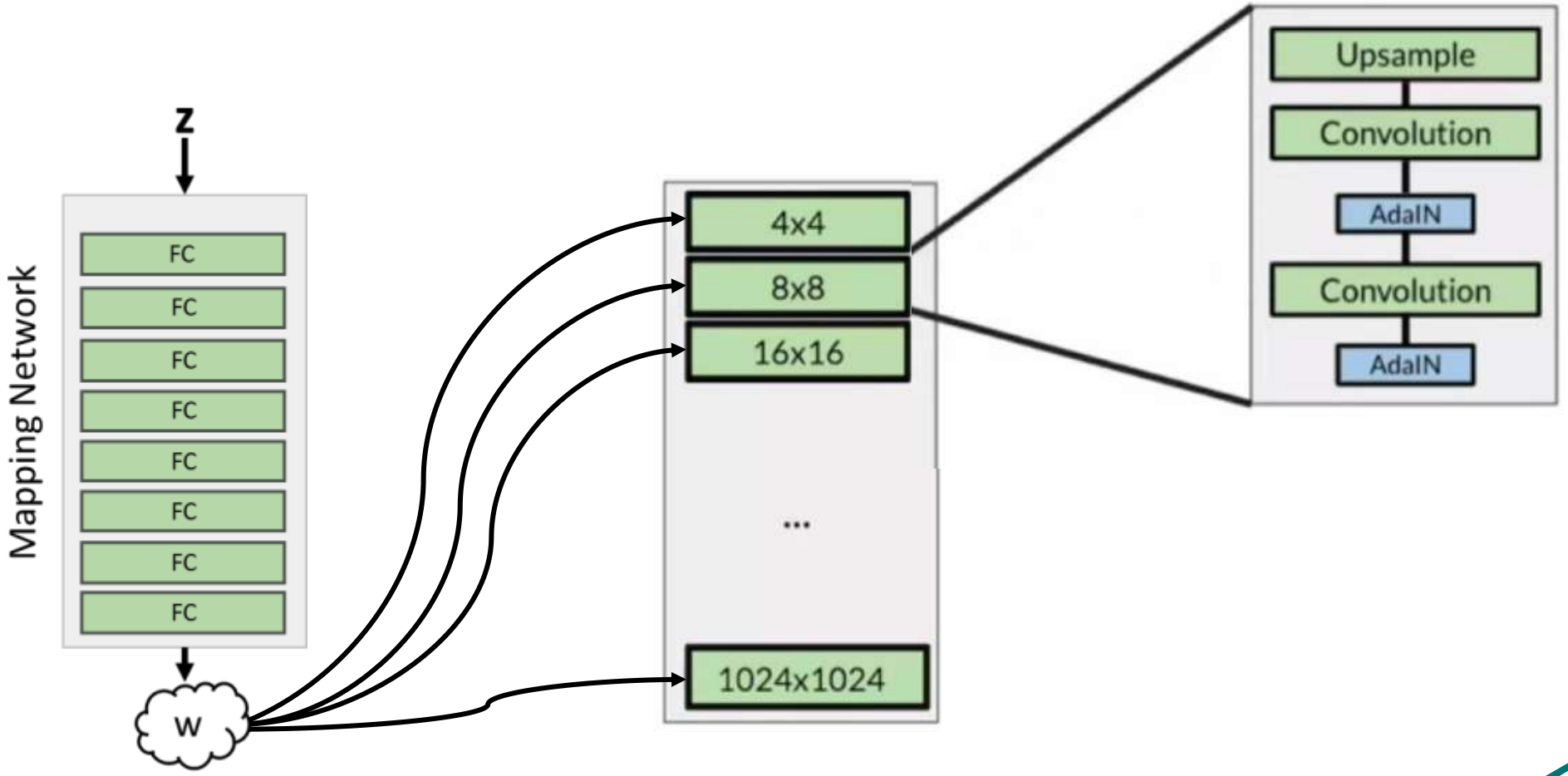
Outline

- Instance Normalization
- Adaptive Instance Normalization
- Where and why AdaIN is used

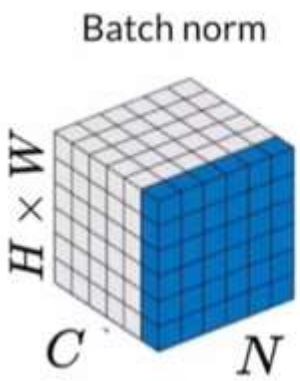
AdaIN in Context

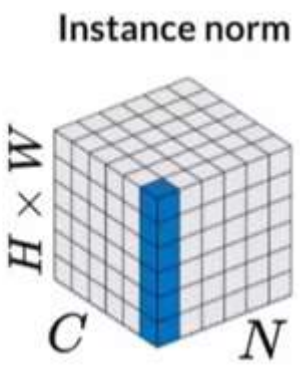


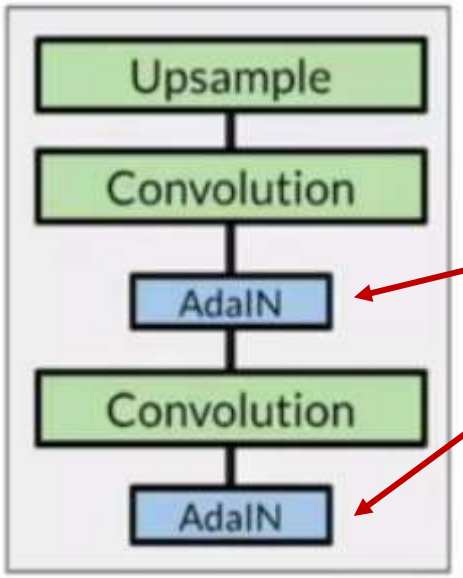
AdaIN in Context



AdaIN



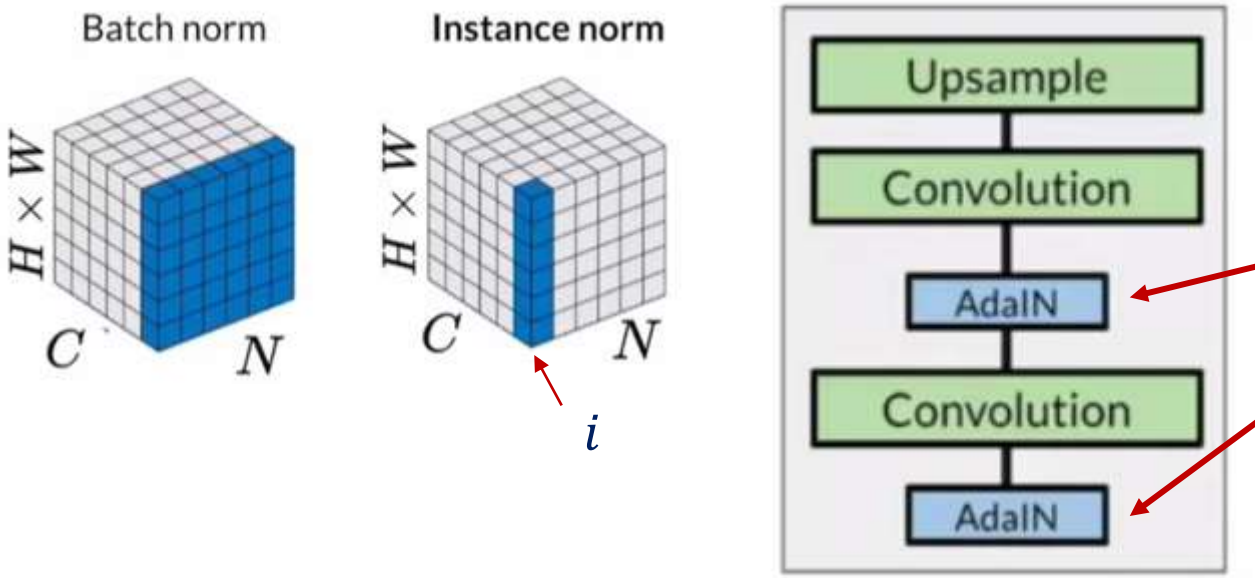




$$\frac{x - \mu(x)}{\delta(x)}$$

Step1: Normalize convolution outputs using **Instance Normalization**

AdaIN

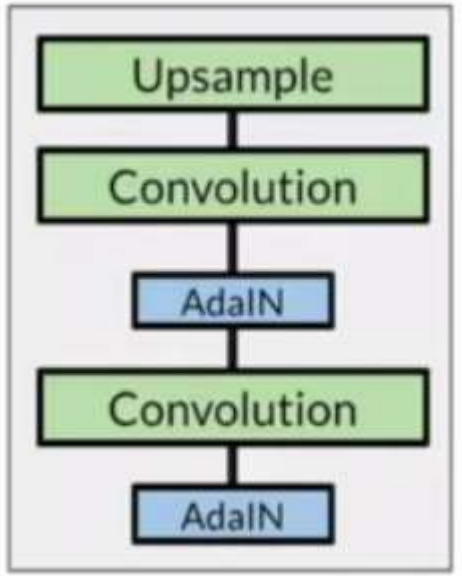


$$\frac{x_i - \mu(x_i)}{\delta(x_i)}$$

Step1: Normalize convolution outputs using **Instance Normalization**

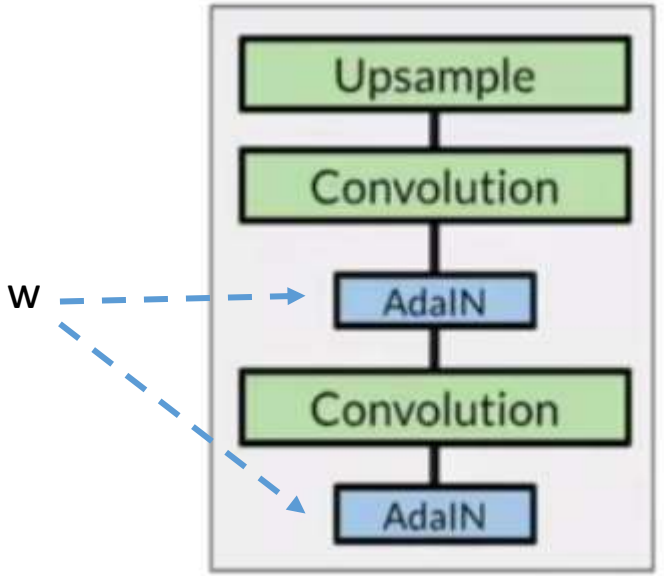
AdaIN

z → Mapping Network → w



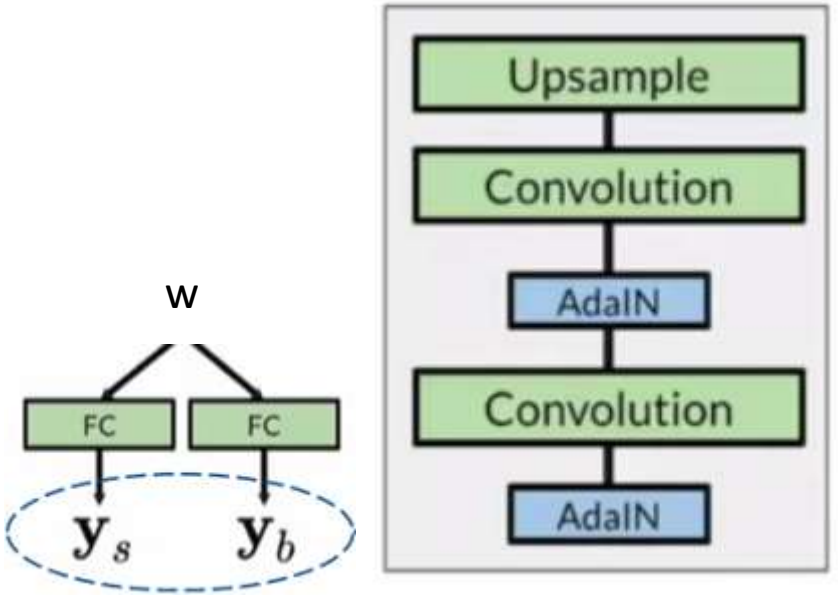
Step 2: Adaptive styles **using the intermediate noise vector**

AdaIN



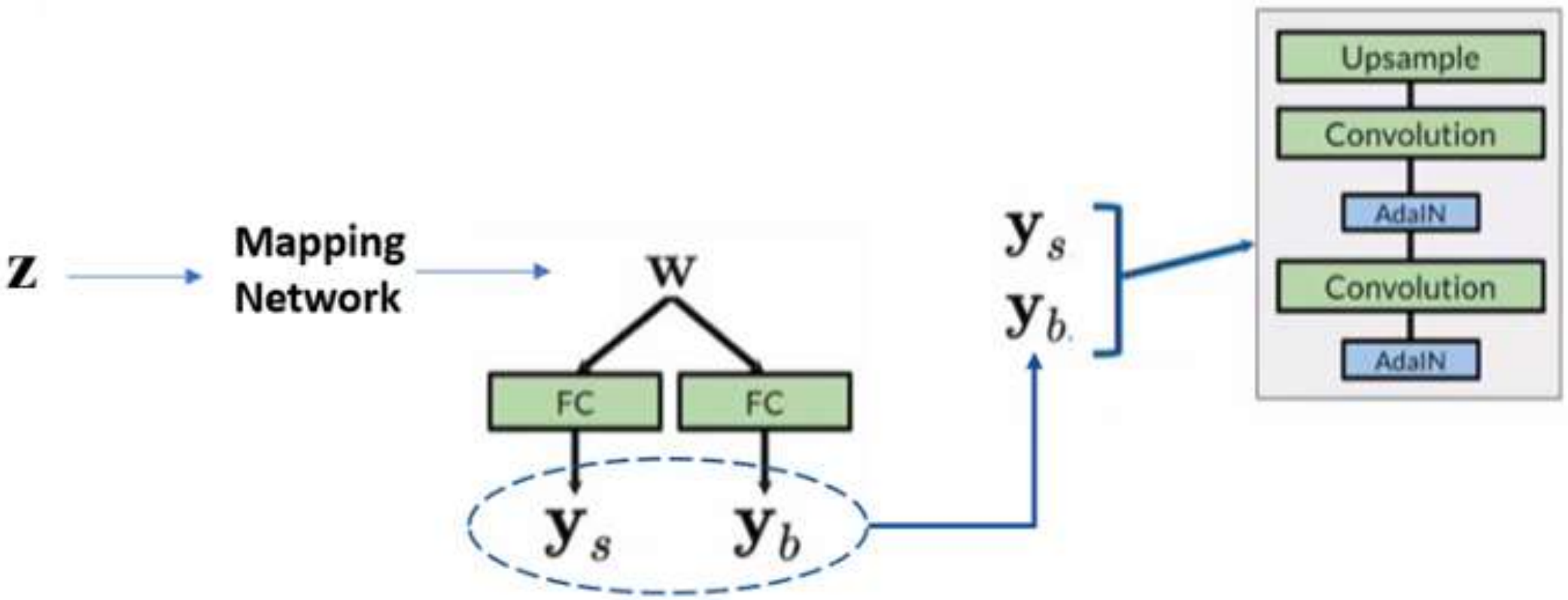
Step 2: Adaptive styles **using the intermediate noise vector**

AdaIN



Step 2: Adaptive styles **using the intermediate noise vector**

AdaIN



Step 2: Adaptive styles **using the intermediate noise vector**

AdaIN

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

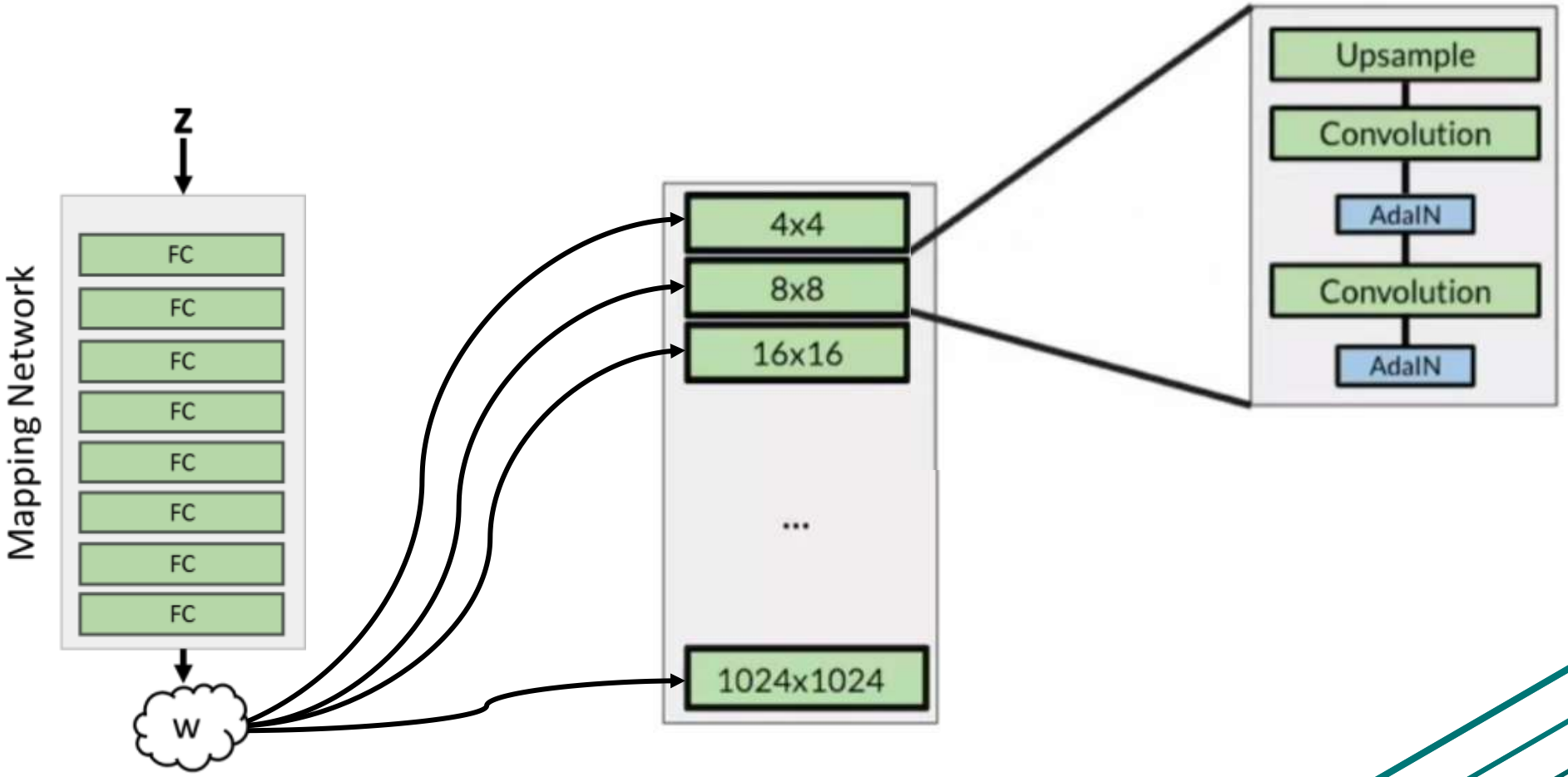
Step 1: Instance Normalization

AdaIN

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

Step 2: Adaptive styles

AdaIN in Context



Summary

- AdaIN transfers style information onto the generated image from the intermediate noise vector w
- Instance Normalization is used to normalize individual examples before apply style statistics from w

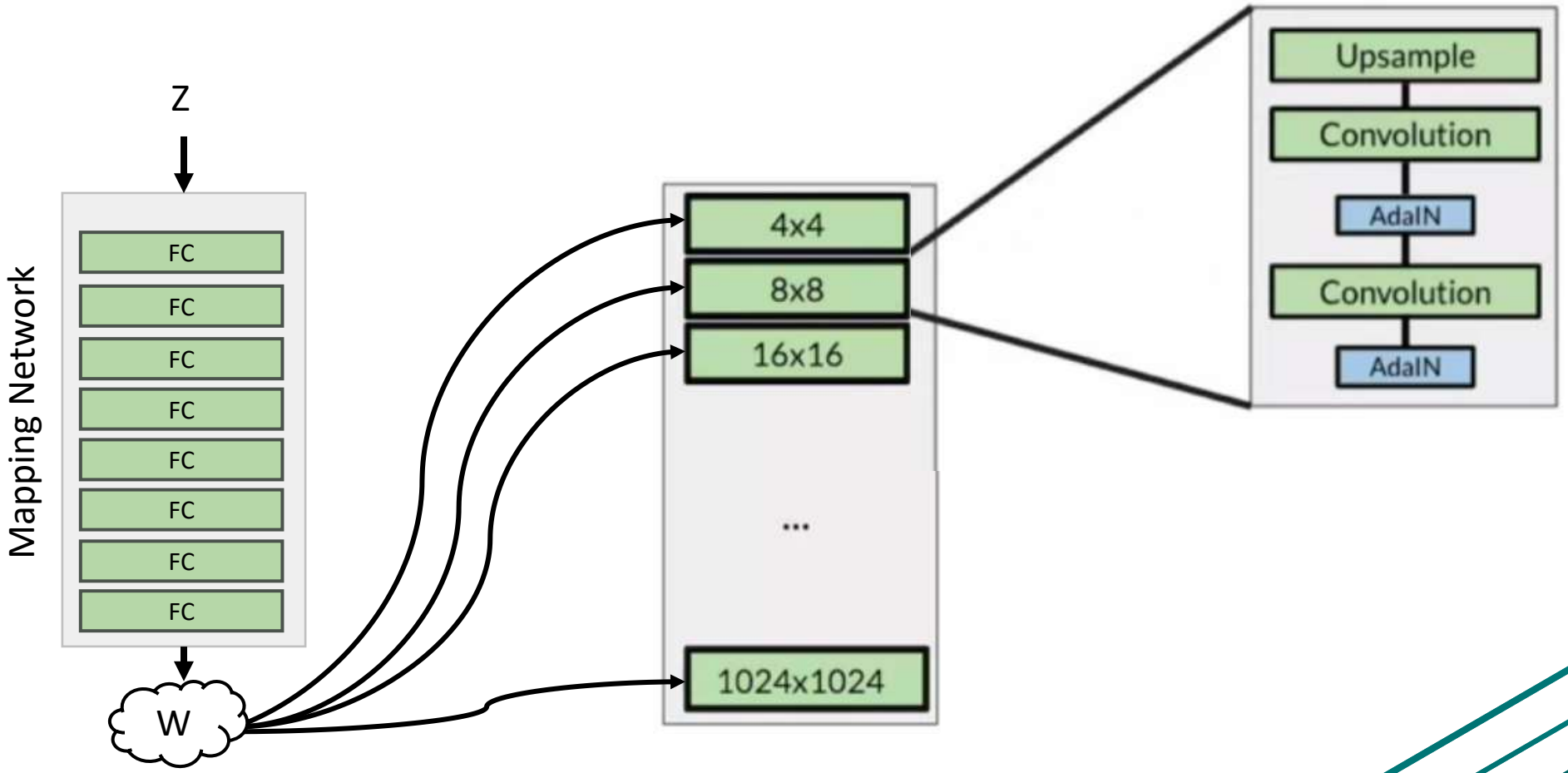
Style and Stochastic Variation

Outline

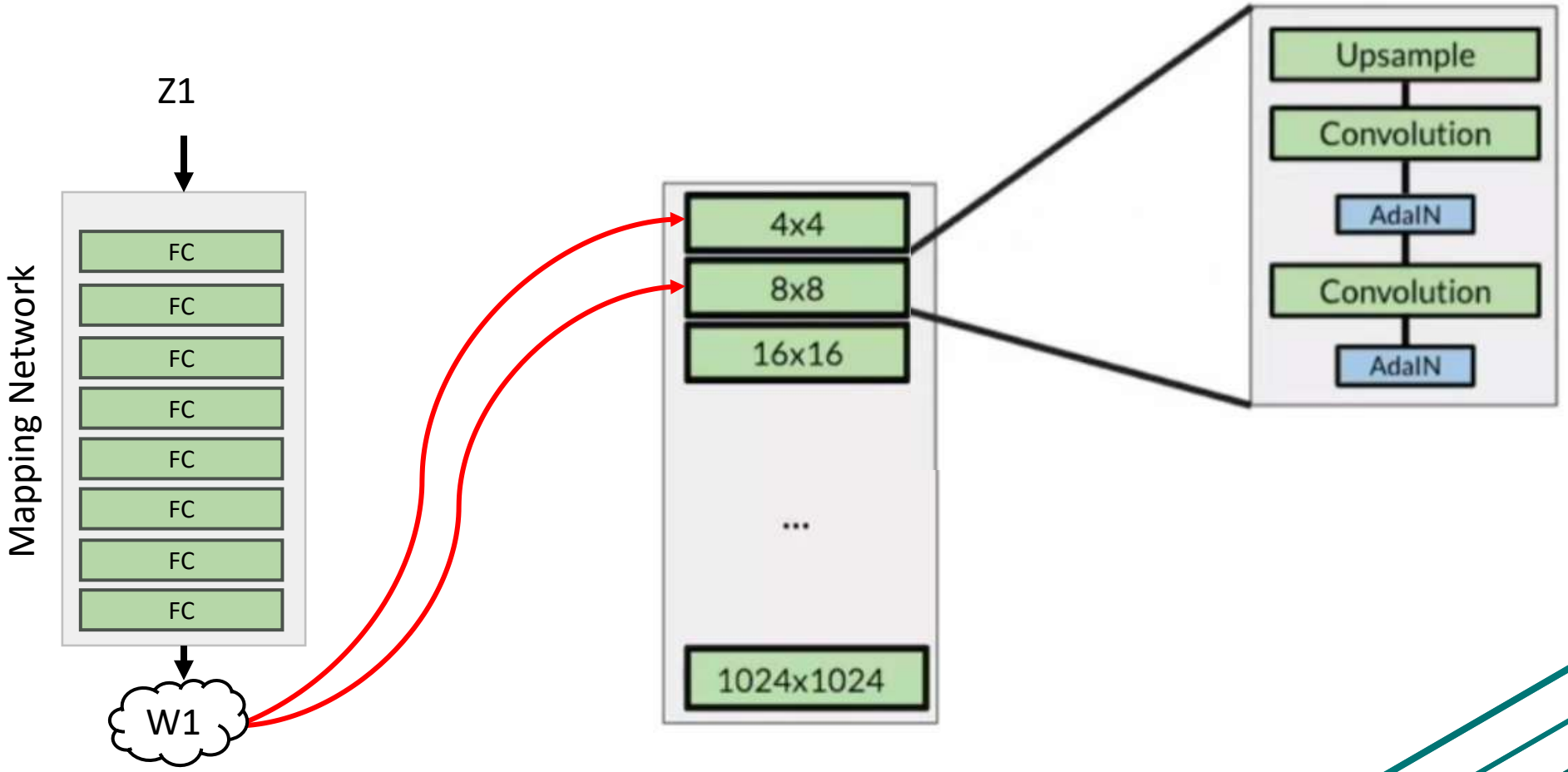
- Controlling coarse and fine styles with StyleGAN
- Style mixing for increased diversity during training/inference
- Stochastic noise for additional variation

Style mixing

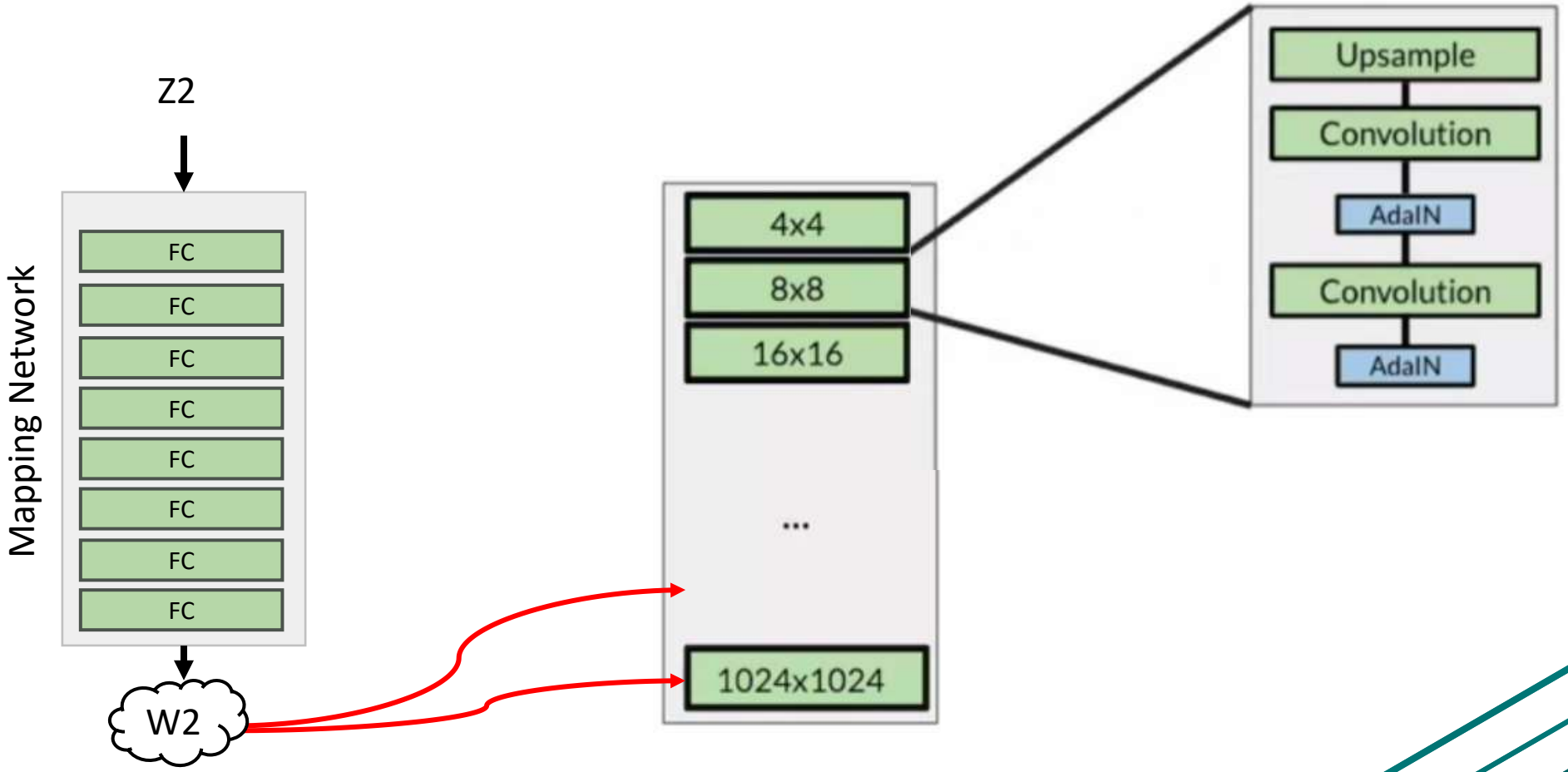
Style mixing in context



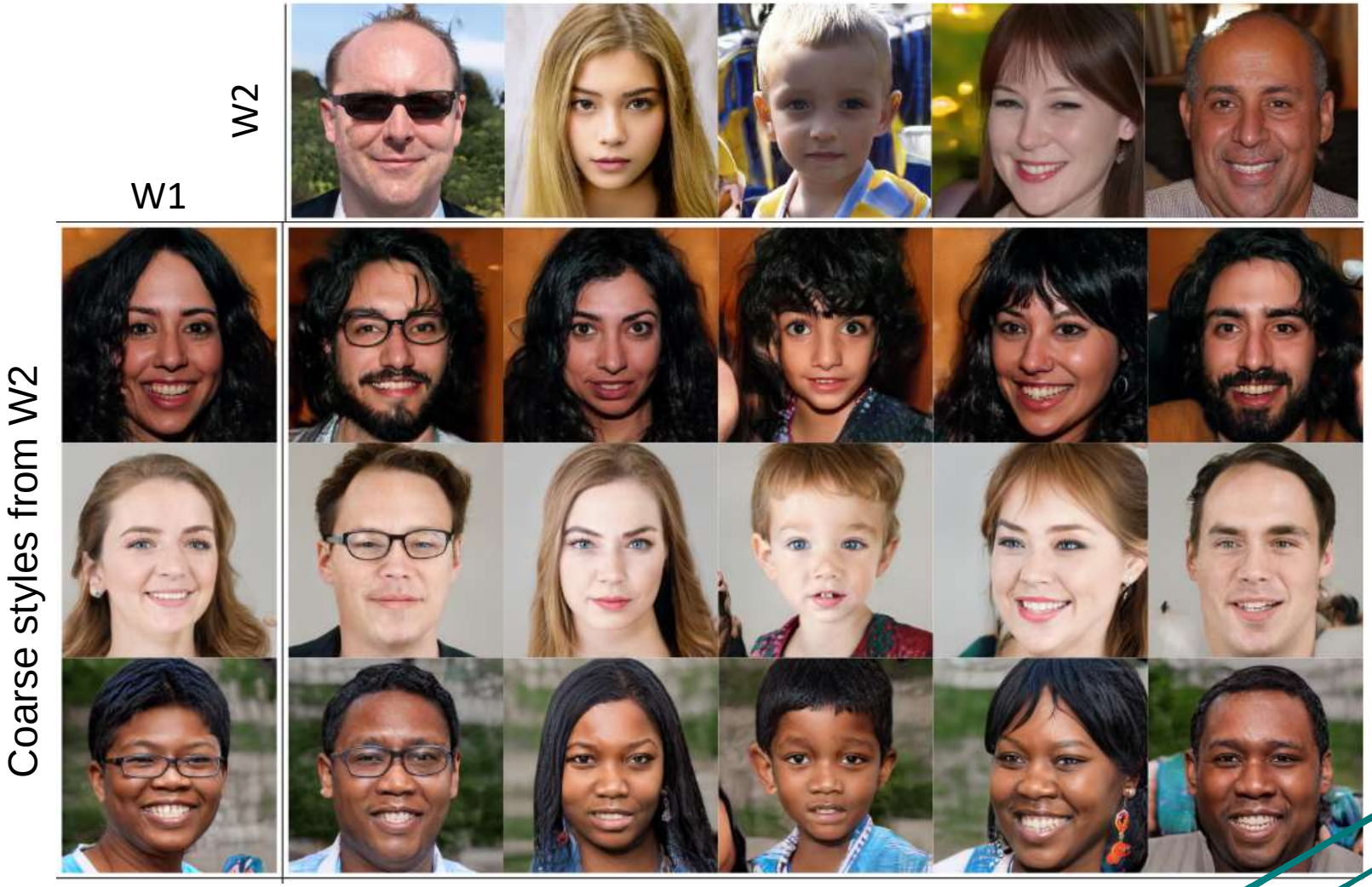
Style mixing in context



Style mixing in context



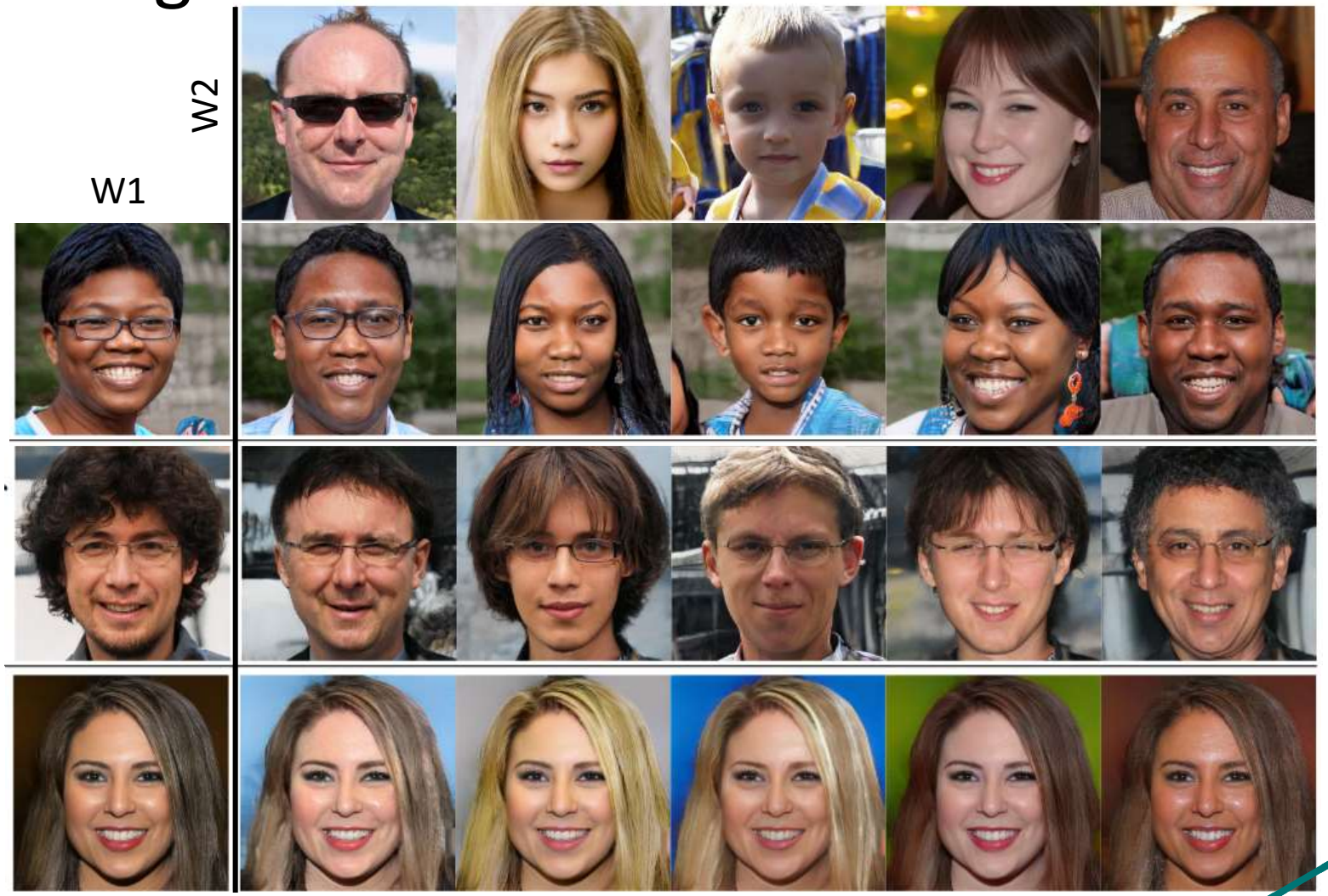
Style mixing: Course style



Style mixing: Middle styles

Style mixing: Fine styles

Style mixing

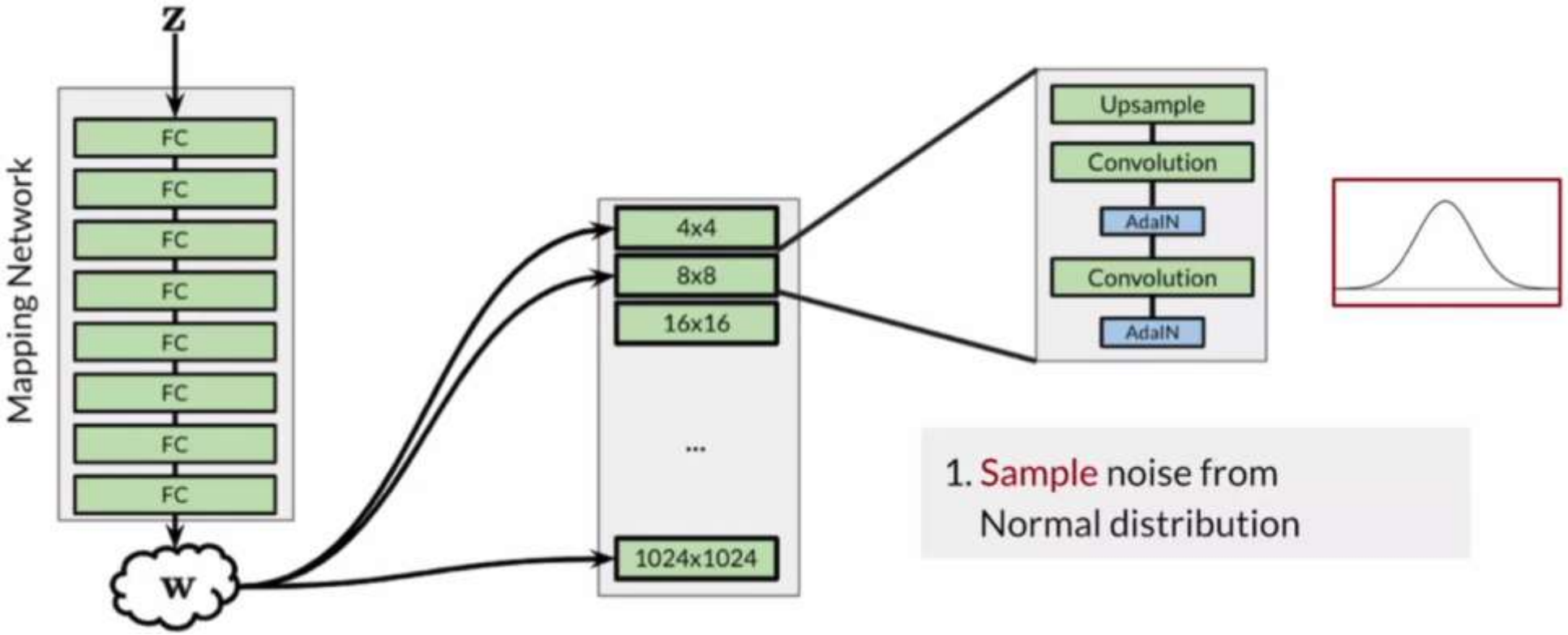


Stochastic variation

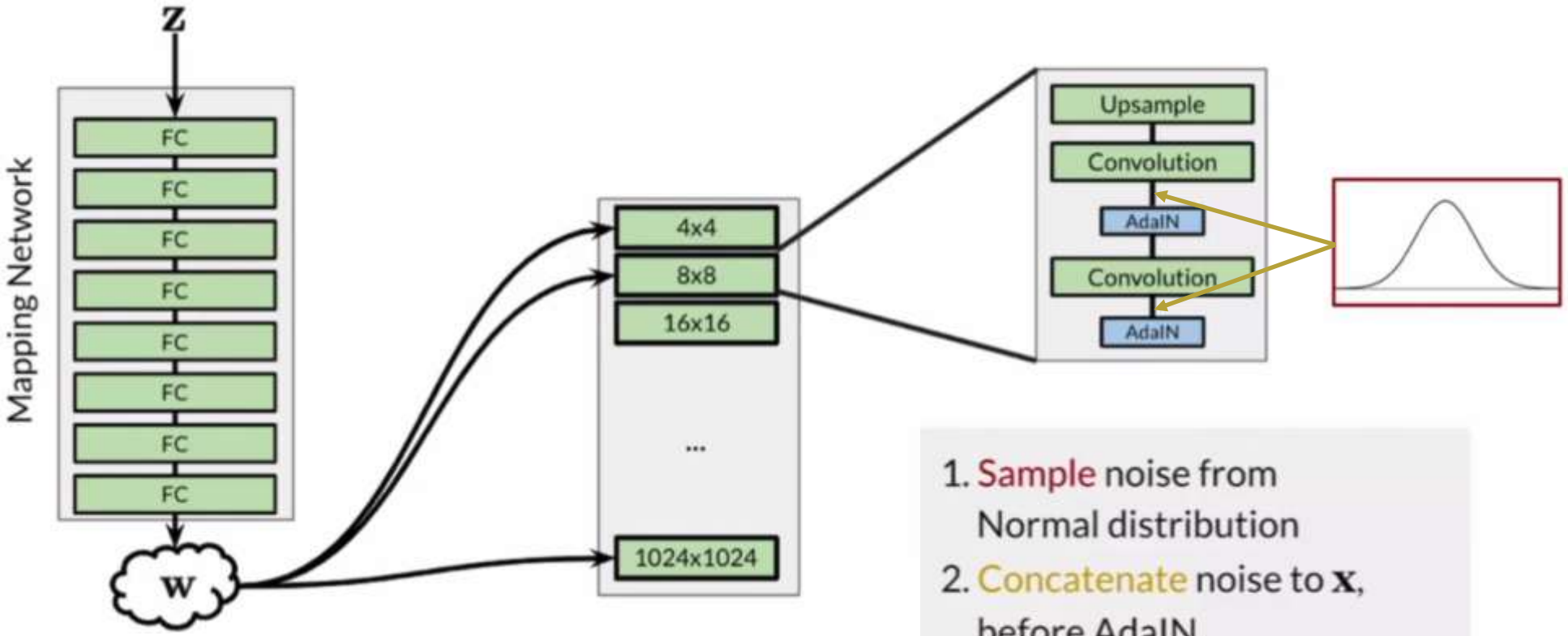
Fine layers

Coarse layers

Stochastic noise in context



Stochastic noise in context

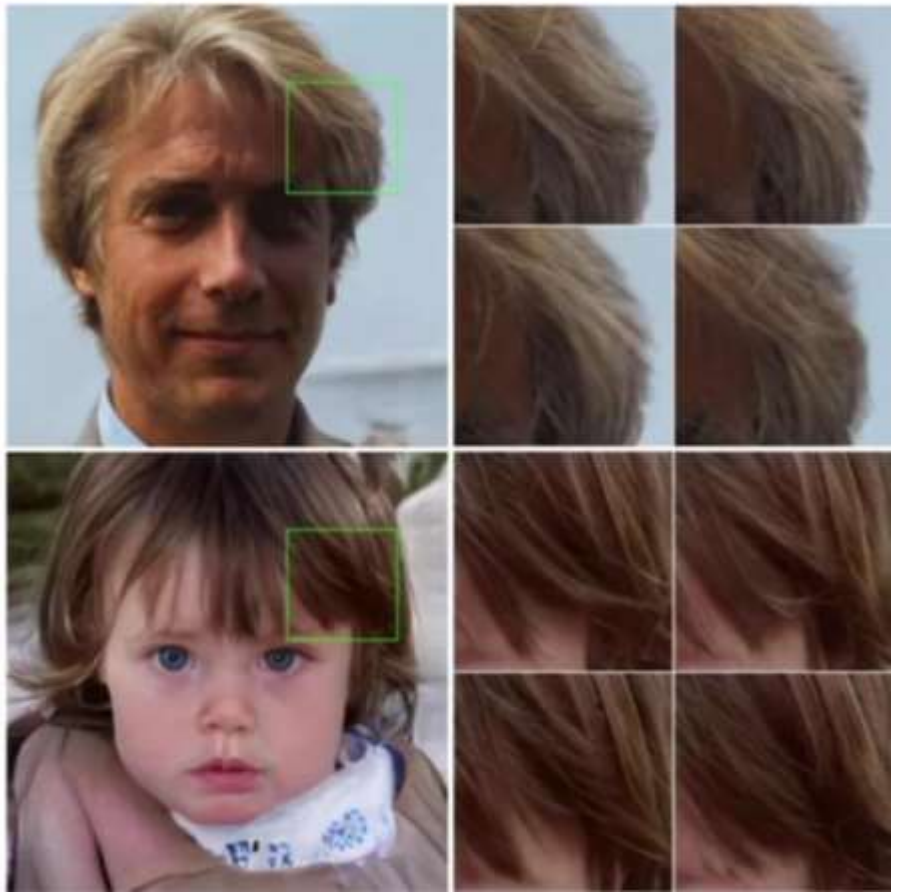


1. Sample noise from Normal distribution
2. Concatenate noise to $\mathbf{x}$, before AdaIN

Stochastic Variation

Small details: hair strands, wrinkles, etc.

Different extra noise values
Create stochastic variation



Summary

- Style mixing increases diversity that the model sees during training
- Stochastic noise causes small variations to output
- Coarse or fineness depends where in the network style or noise is added
  - Earlier for coarser variation
  - Later for finer variation

Implementation

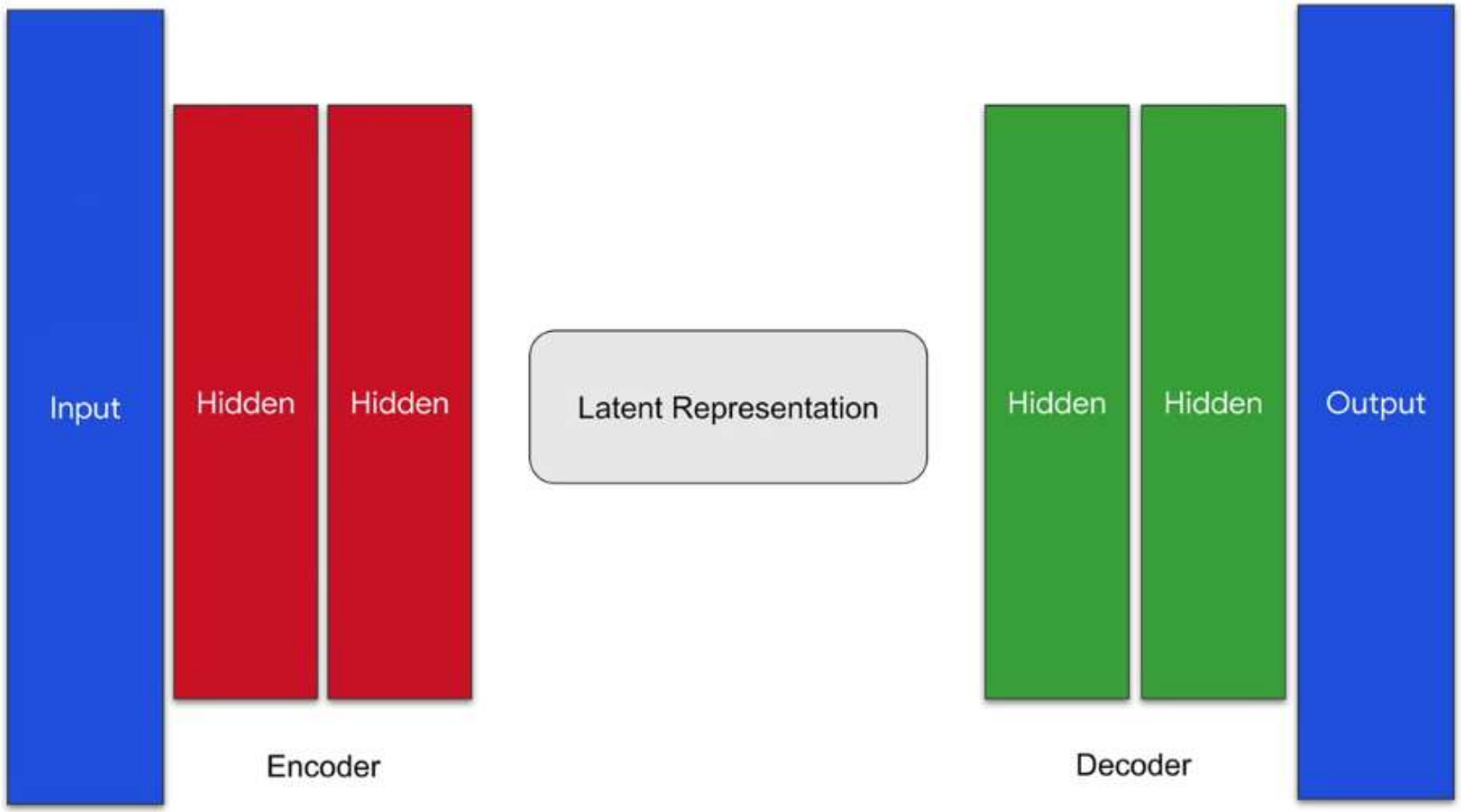
- https://keras.io/examples/generative/stylegan/
- https://colab.research.google.com/github/keras-team/keras-io/blob/master/examples/generative/ipynb/stylegan.ipynb

VAE

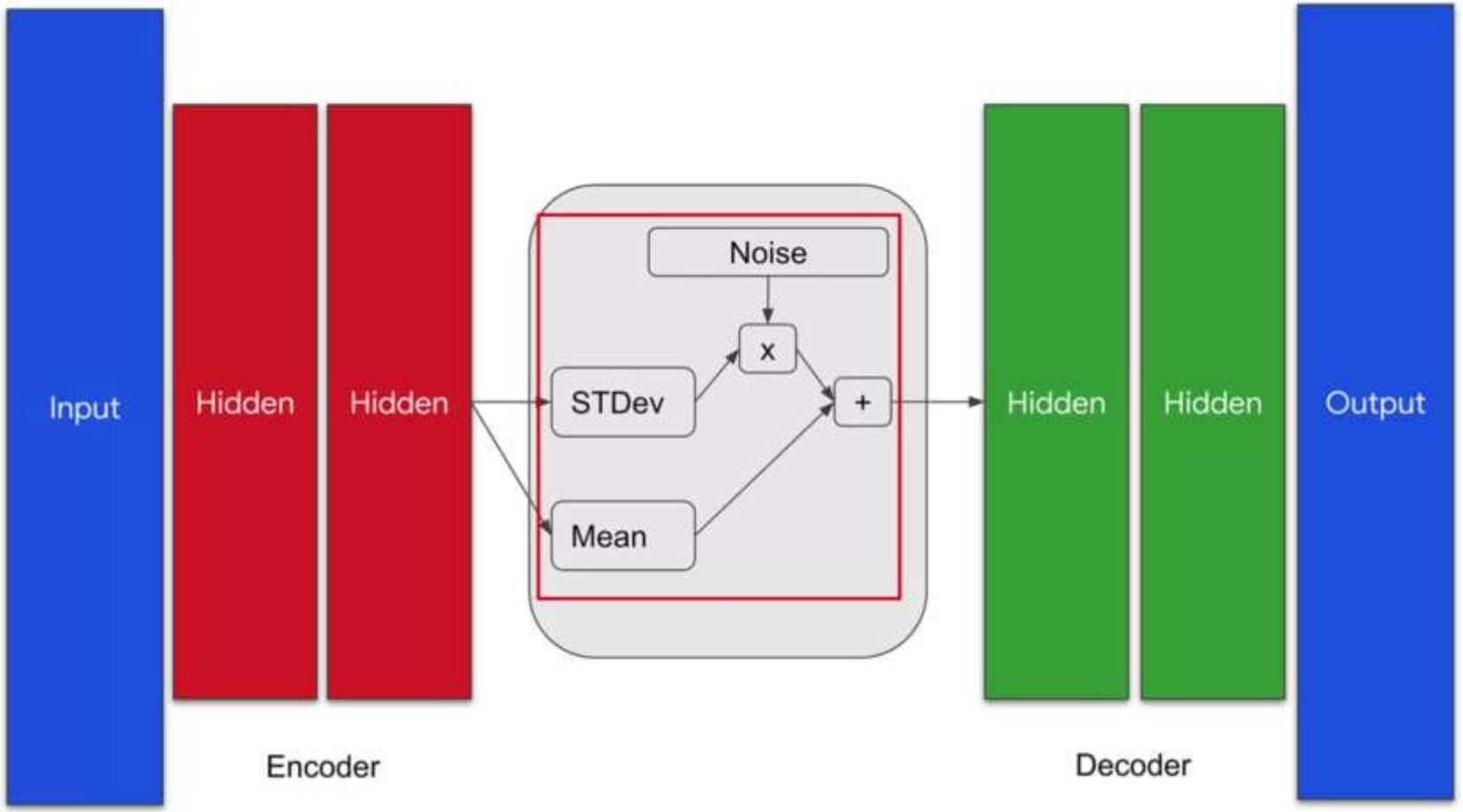
VAE

- Unlike a traditional autoencoder, which maps the input onto a latent vector, a **VAE** maps the input data into the parameters of a probability distribution, such as the mean and variance of a Gaussian.

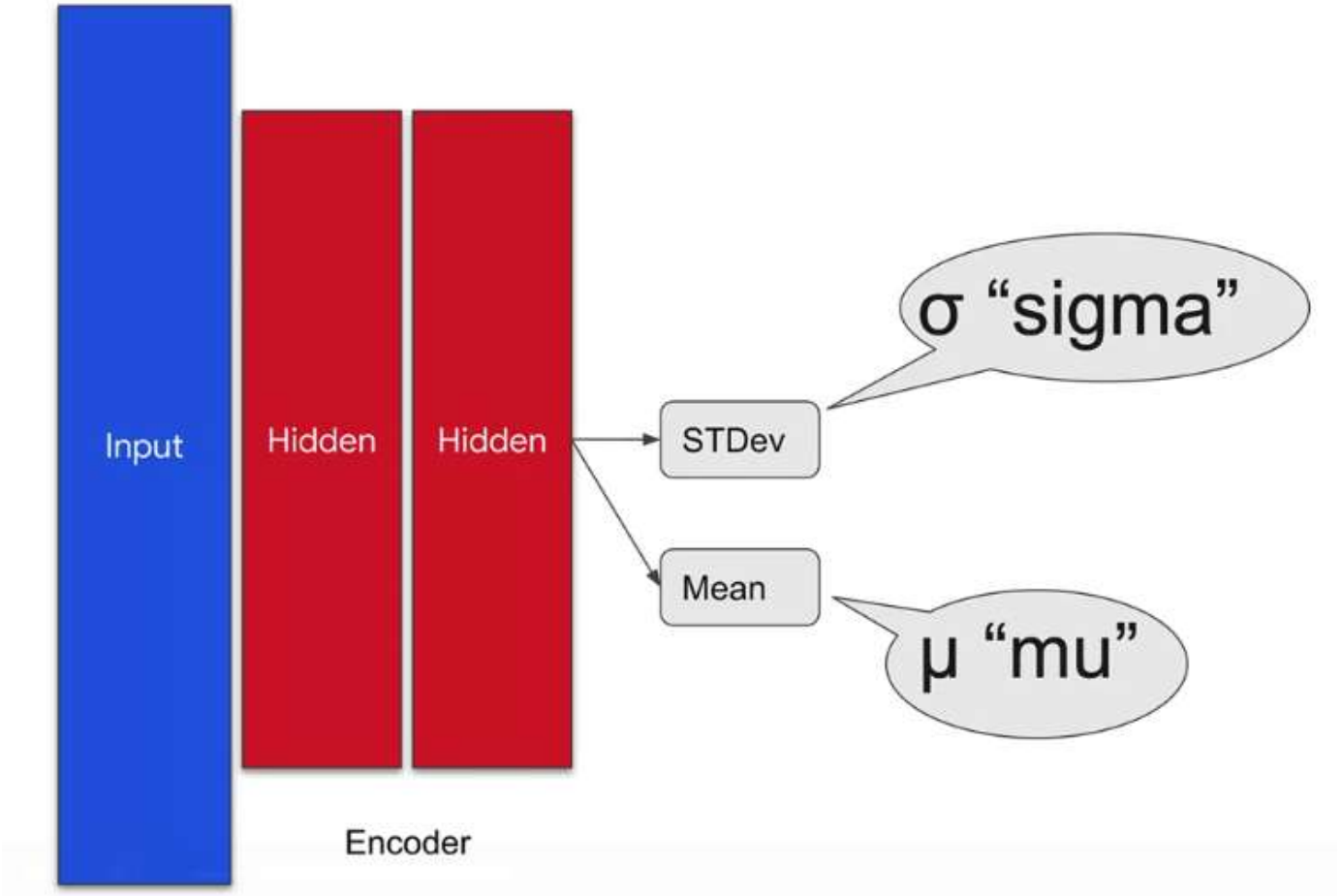
AE


Input
Hidden
Hidden
Encoder
Latent Representation
Hidden
Hidden
Output
Decoder

VAE


Input
Hidden
Hidden
Noise
x
STDev
+
Mean
Hidden
Hidden
Output
Encoder
Decoder

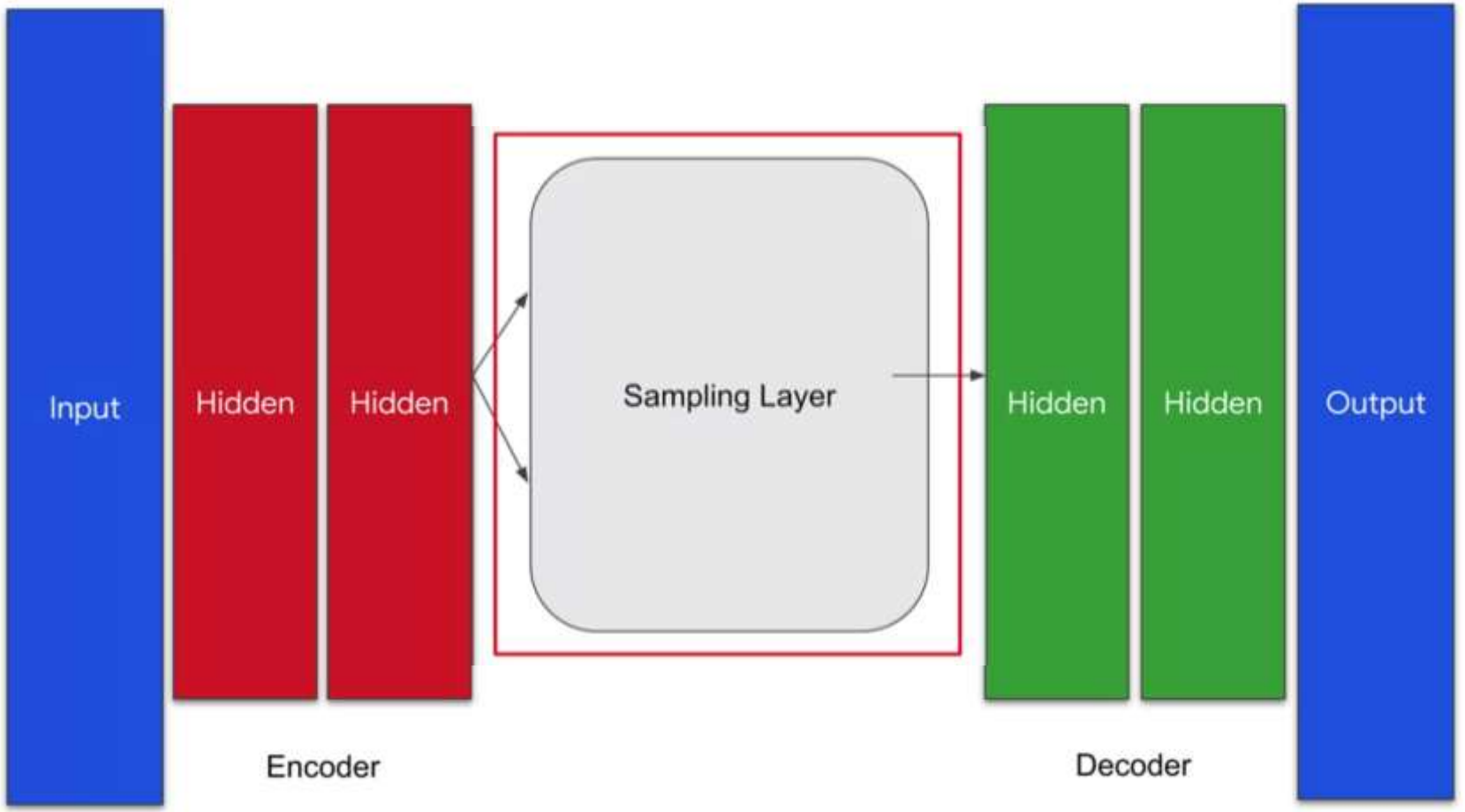
VAE



Build the encoder

```
latent_dim = 2

encoder_inputs = keras.Input(shape=(28, 28, 1))
x = layers.Conv2D(32, 3, activation="relu", strides=2, padding="same")(encoder_inputs)
x = layers.Conv2D(64, 3, activation="relu", strides=2, padding="same")(x)
x = layers.Flatten()(x)
x = layers.Dense(16, activation="relu")(x)
z_mean = layers.Dense(latent_dim, name="z_mean")(x)
z_log_var = layers.Dense(latent_dim, name="z_log_var")(x)
z = Sampling()([z_mean, z_log_var])
encoder = keras.Model(encoder_inputs, [z_mean, z_log_var, z], name="encoder")
encoder.summary()
```

Sampling Layer and Encoder



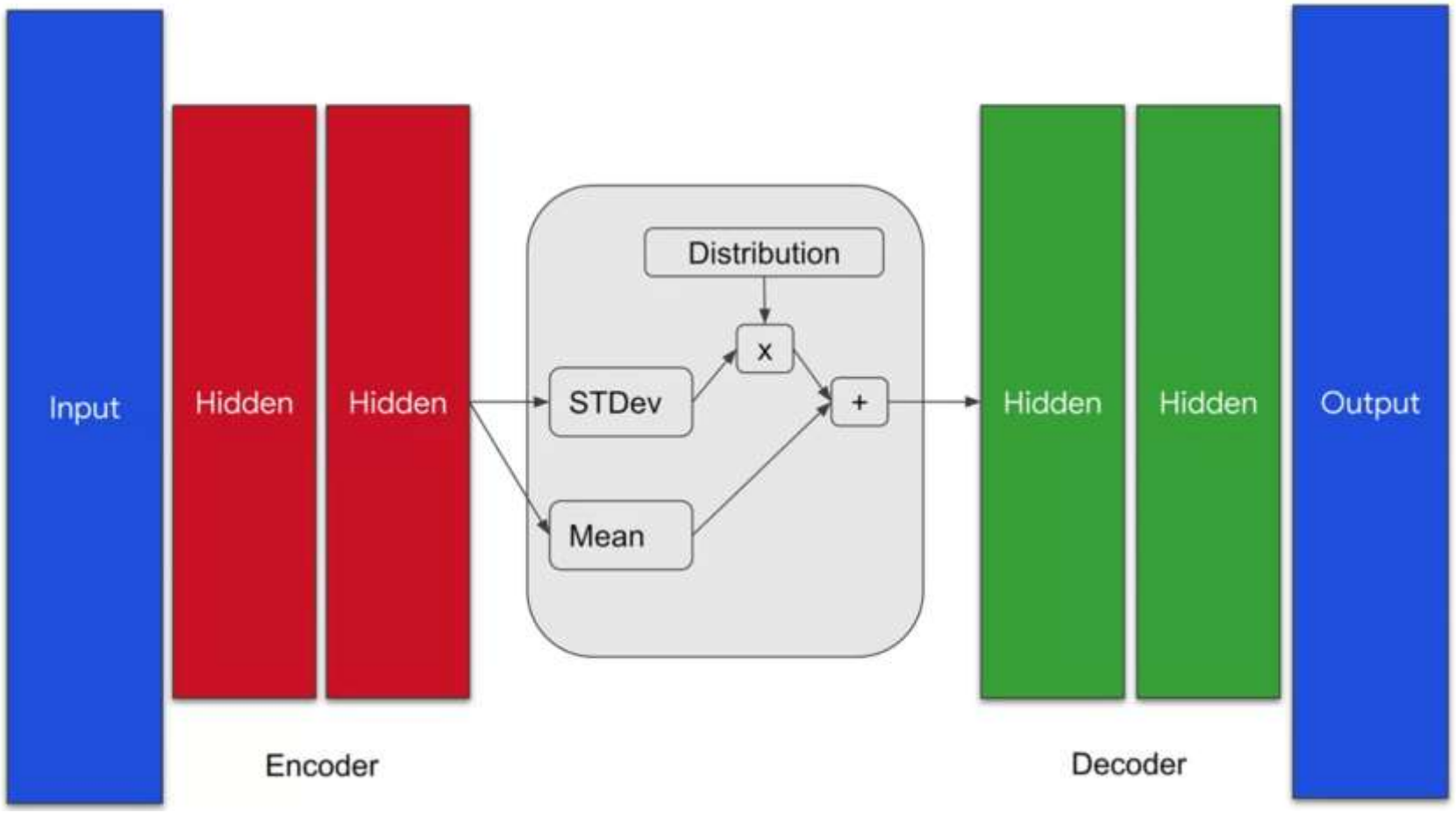
Create a sampling layer

```python
class Sampling(layers.Layer):
    """Uses (z_mean, z_log_var) to sample z, the vector encoding a digit."""

    def call(self, inputs):
        z_mean, z_log_var = inputs
        batch = tf.shape(z_mean)[0]
        dim = tf.shape(z_mean)[1]
        epsilon = tf.keras.backend.random_normal(shape=(batch, dim))
        return z_mean + tf.exp(0.5 * z_log_var) * epsilon
```

Build the decoder

```
latent_inputs = keras.Input(shape=(latent_dim,))
x = layers.Dense(7 * 7 * 64, activation="relu")(latent_inputs)
x = layers.Reshape((7, 7, 64))(x)
x = layers.Conv2DTranspose(64, 3, activation="relu", strides=2, padding="same")(x)
x = layers.Conv2DTranspose(32, 3, activation="relu", strides=2, padding="same")(x)
decoder_outputs = layers.Conv2DTranspose(1, 3, activation="sigmoid", padding="same")(x)
decoder = keras.Model(latent_inputs, decoder_outputs, name="decoder")
decoder.summary()
```

Loss Function and Model Definition



Kullback–Leibler divergence

```
kl_loss = -0.5 * (1 + z_log_var - tf.square(z_mean) - tf.exp(z_log_var))
kl_loss = tf.reduce_mean(tf.reduce_sum(kl_loss, axis=1))
```

Kullback–Leibler divergence

```
kl_loss = -0.5 * (1 + z_log_var - tf.square(z_mean) - tf.exp(z_log_var))
kl_loss = tf.reduce_mean(tf.reduce_sum(kl_loss, axis=1))
```

https://en.wikipedia.org/wiki/Kullback%E2%80%93Leibler_divergence

منابع

- https://www.coursera.org/specializations/generative-adversarial-networks-gans
- Progressive Growing of GANs for Improved Quality, Stability, and Variation
- A Style-Based Generator Architecture for Generative Adversarial Networks
- Hands-On Image Generation with TensorFlow By Soon Yau Cheong
- https://keras.io/examples/generative/