50.039 Theory and Practice of Deep Learning

W10-S3 Generative Models in Deep Learning

Matthieu De Mari

About this week (Week 10)

1. What are typical **generative models** in deep learning?
2. What is an **autoencoder** and what are its uses?
3. What is a **fractionally strided convolution layer**?
4. What are **variational autoencoders** and why do **noise representations of latent features** work better than the ones in standard autoencoders?
5. What are **Generative Adversarial Networks** (**GANs**) and their uses?
6. What are the **advanced techniques** in GANs and deepfakes?

About this week (Week 10)

1. What are typical **generative models** in deep learning?
2. What is an **autoencoder** and what are its uses?
3. What is a **fractionally strided convolution layer**?
4. What are **variational autoencoders** and why do **noise representations of latent features** work better than the ones in standard autoencoders?
5. What are **Generative Adversarial Networks** (**GANs**) and their uses?
6. What are the **advanced techniques** in GANs and deepfakes?

Outline

In this lecture

- More advanced concepts on GANs
- Wasserstein distance in the GAN loss function
- (Progressive GANs)
- (StyleGANs)
- (Conditional GANs and CycleGANs)
- Deepfakes and ethics

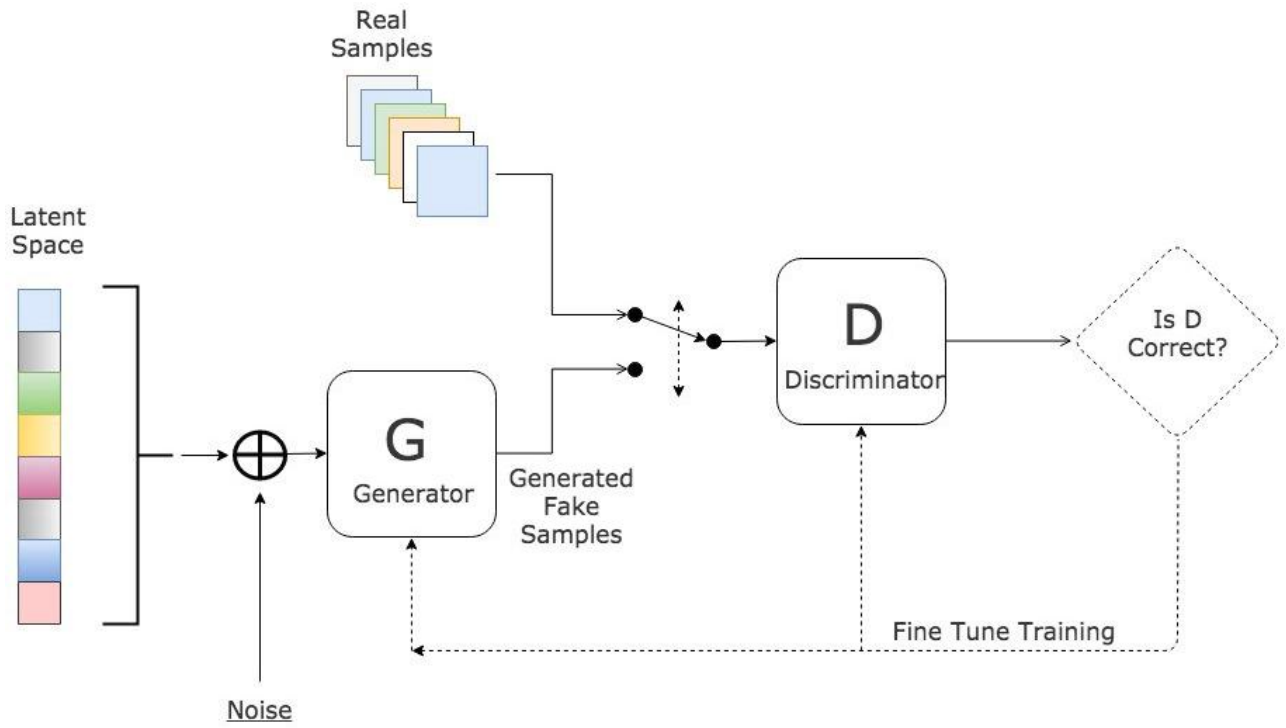
Outline

- GAN - Generative Adversarial Networks
- 3D-GAN - Learning a Probabilistic Latent Space of Object Shapes via 3D Generative-Adversarial Modeling
- acGAN - Face Aging With Conditional Generative Adversarial Networks
- AC-GAN - Conditional Image Synthesis With Auxiliary Classifier GANs
- AdaGAN - AdaGAN: Boosting Generative Models
- AEGAN - Learning Inverse Mapping by Autoencoder based Generative Adversarial Nets
- AffGAN - Amortised MAP Inference for Image Super-resolution
- AL-CGAN - Learning to Generate Images of Outdoor Scenes from Attributes and Semantic Layouts
- ALI - Adversarially Learned Inference
- AM-GAN - Generative Adversarial Nets with Labeled Data by Activation Maximization
- AnoGAN - Unsupervised Anomaly Detection with Generative Adversarial Networks to Guide Marker Discovery
- ArtGAN - ArtGAN: Artwork Synthesis with Conditional Categorial GANs
- b-GAN - b-GAN: Unified Framework of Generative Adversarial Networks
- Bayesian GAN - Deep and Hierarchical Implicit Models
- BEGAN - BEGAN: Boundary Equilibrium Generative Adversarial Networks
- BiGAN - Adversarial Feature Learning
- BS-GAN - Boundary-Seeking Generative Adversarial Networks
- CGAN - Conditional Generative Adversarial Nets
- CaloGAN - CaloGAN: Simulating 3D High Energy Particle Showers in Multi-Layer Electromagnetic Calorimeters with Generative Adversarial Networks
- CCGAN - Semi-Supervised Learning with Context-Conditional Generative Adversarial Networks
- CatGAN - Unsupervised and Semi-supervised Learning with Categorical Generative Adversarial Networks
- CoGAN - Coupled Generative Adversarial Networks
- Context-RNN-GAN - Contextual RNN-GANs for Abstract Reasoning Diagram Generation
- C-RNN-GAN - C-RNN-GAN: Continuous recurrent neural networks with adversarial training
- CS-GAN - Improving Neural Machine Translation with Conditional Sequence Generative Adversarial Nets
- CVAE-GAN - CVAE-GAN: Fine-Grained Image Generation through Asymmetric Training
- CycleGAN - Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks
- DTN - Unsupervised Cross-Domain Image Generation
- DCGAN - Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks
- DiscoGAN - Learning to Discover Cross-Domain Relations with Generative Adversarial Networks
- DR-GAN - Disentangled Representation Learning GAN for Pose-Invariant Face Recognition
- DualGAN - DualGAN: Unsupervised Dual Learning for Image-to-Image Translation
- EBGAN - Energy-based Generative Adversarial Network
- f-GAN - f-GAN: Training Generative Neural Samplers using Variational Divergence Minimization
- FF-GAN - Towards Large-Pose Face Frontalization in the Wild
- GAWWN - Learning What and Where to Draw
- GeneGAN - GeneGAN: Learning Object Transfiguration and Attribute Subspace from Unpaired Data
- Geometric GAN - Geometric GAN
- GoGAN - Gang of GANs: Generative Adversarial Networks with Maximum Margin Ranking
- GP-GAN - GP-GAN: Towards Realistic High-Resolution Image Blending
- IAN - Neural Photo Editing with Introspective Adversarial Networks
- iGAN - Generative Visual Manipulation on the Natural Image Manifold
- IcGAN - Invertible Conditional GANs for image editing
- ID-CGAN - Image De-raining Using a Conditional Generative Adversarial Network
- Improved GAN - Improved Techniques for Training GANs
- InfoGAN - InfoGAN: Interpretable Representation Learning by Information Maximizing Generative Adversarial Nets
- LAGAN - Learning Particle Physics by Example: Location-Aware Generative Adversarial Networks for Physics Synthesis
- LAPGAN - Deep Generative Image Models using a Laplacian Pyramid of Adversarial Networks

Vanilla GAN architecture

To train a GAN, we need five components.

1. **Noise sample generator Z**
2. **Generator G**
3. **Discriminator D**
4. **Loss L_D on discriminator D**
5. **Loss L_G on generator G**



Vanilla GAN architecture

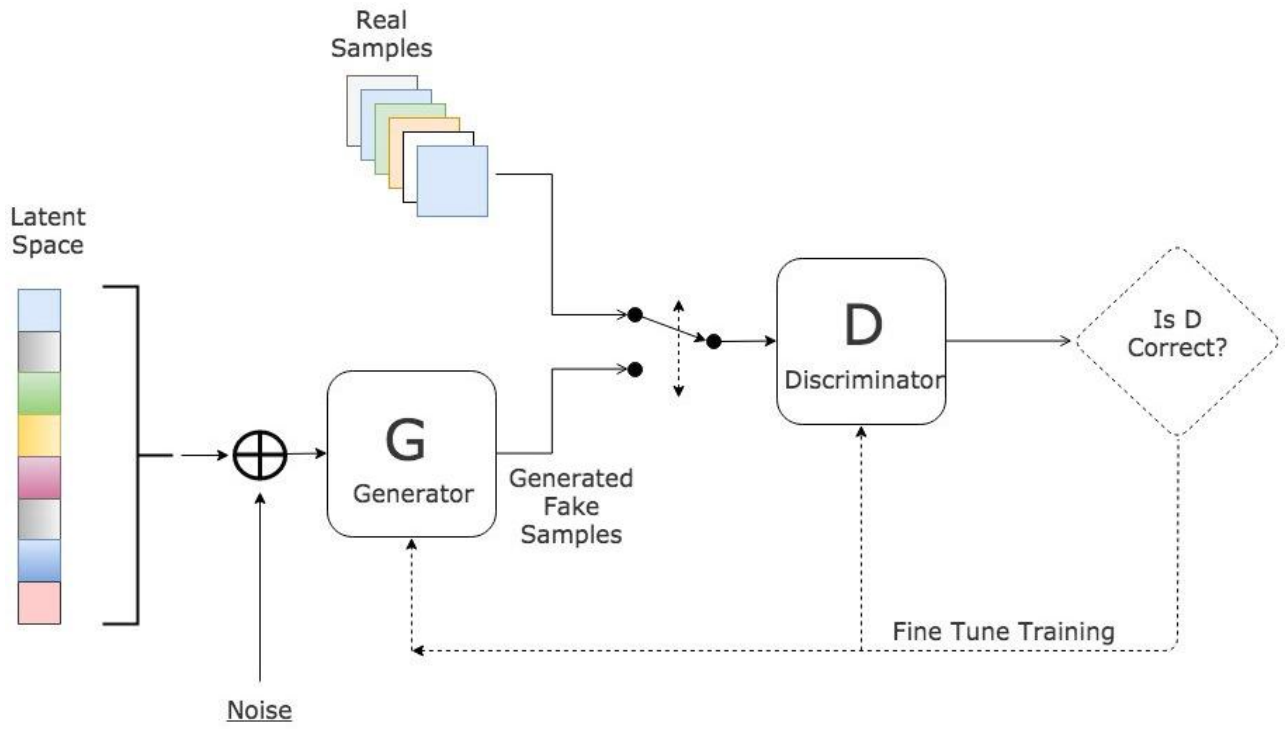
To train a GAN, we need five components.

1. **Noise sample generator Z**

Simply draw some random vector $z \in \mathbb{R}^K$, by using random noise.

$$z \to N(0_K, I_K)$$

It will be fed to our generator G.



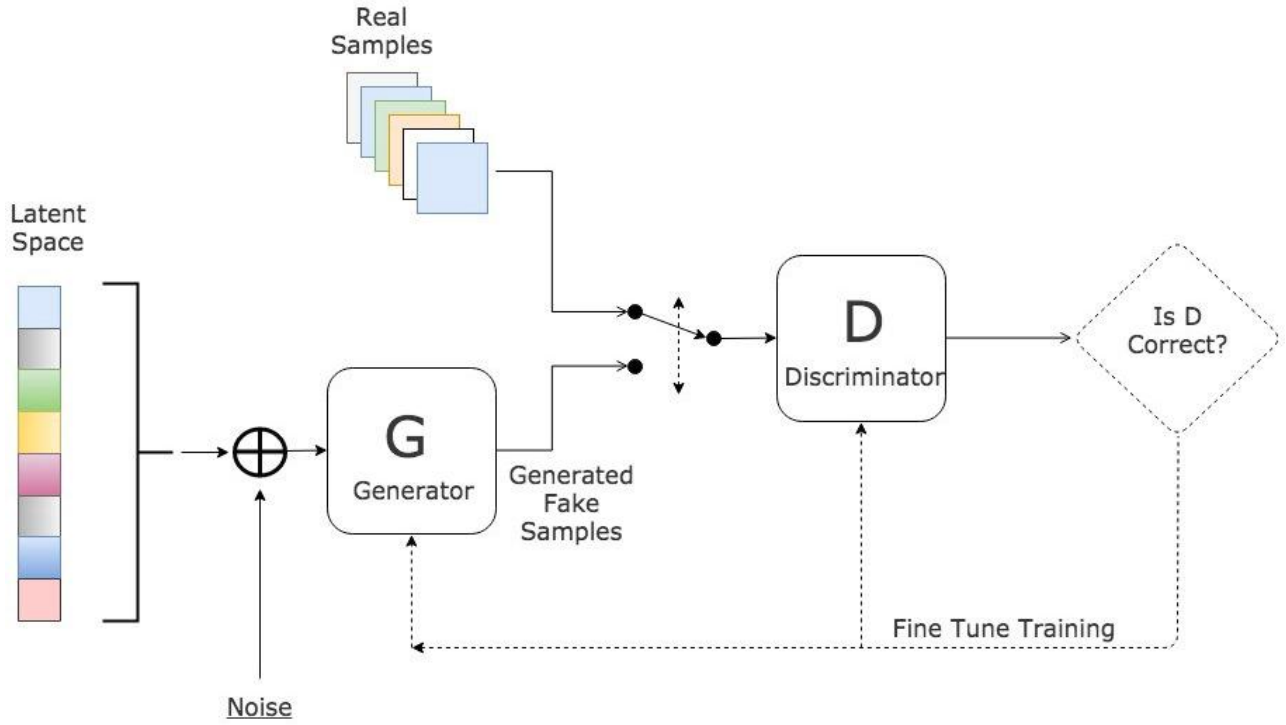
Vanilla GAN architecture

To train a GAN, we need five components.

2. **Generator G**

Receives the noise vector z as input and produces a (fake) image $\hat{x}$ as output.

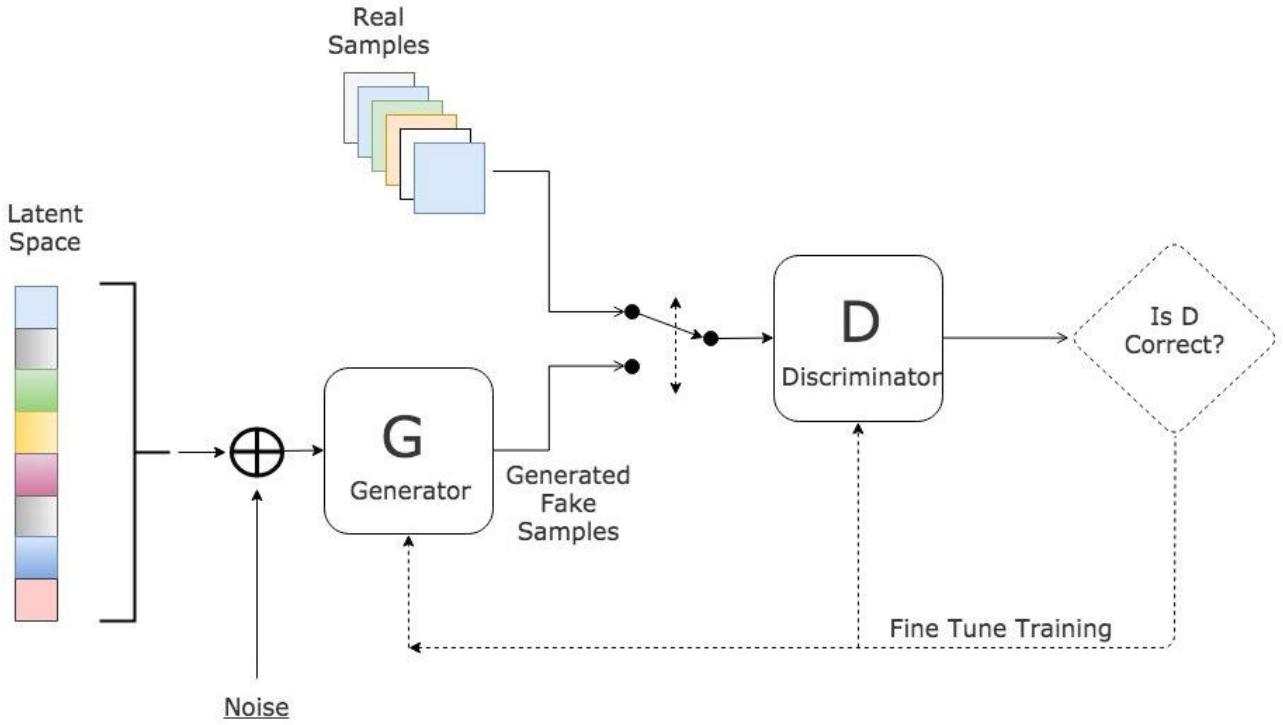
In terms of architecture, same logic as the decoder part of our AE/VAE, i.e. **upsampling FC** or **TransposeConv** layers!



Vanilla GAN architecture

To train a GAN, we need five components.

3. Discriminator D



Will receive

- an image x from the dataset X half of the time,
- an image $\hat{x} = G(z)$ from the generator the other half.

Binary classification: needs to classify the image as fake (0) or real (1).

Vanilla GAN architecture

To train a GAN, we need five components.

4. Loss L_D on discriminator D

The purpose of D is to correctly guess whether the image is real or was generated by G (BCE loss!)

- The first component checks if the discriminator is correctly classifying the real samples.
- The second component checks if the discriminator correctly classifies the fake samples.

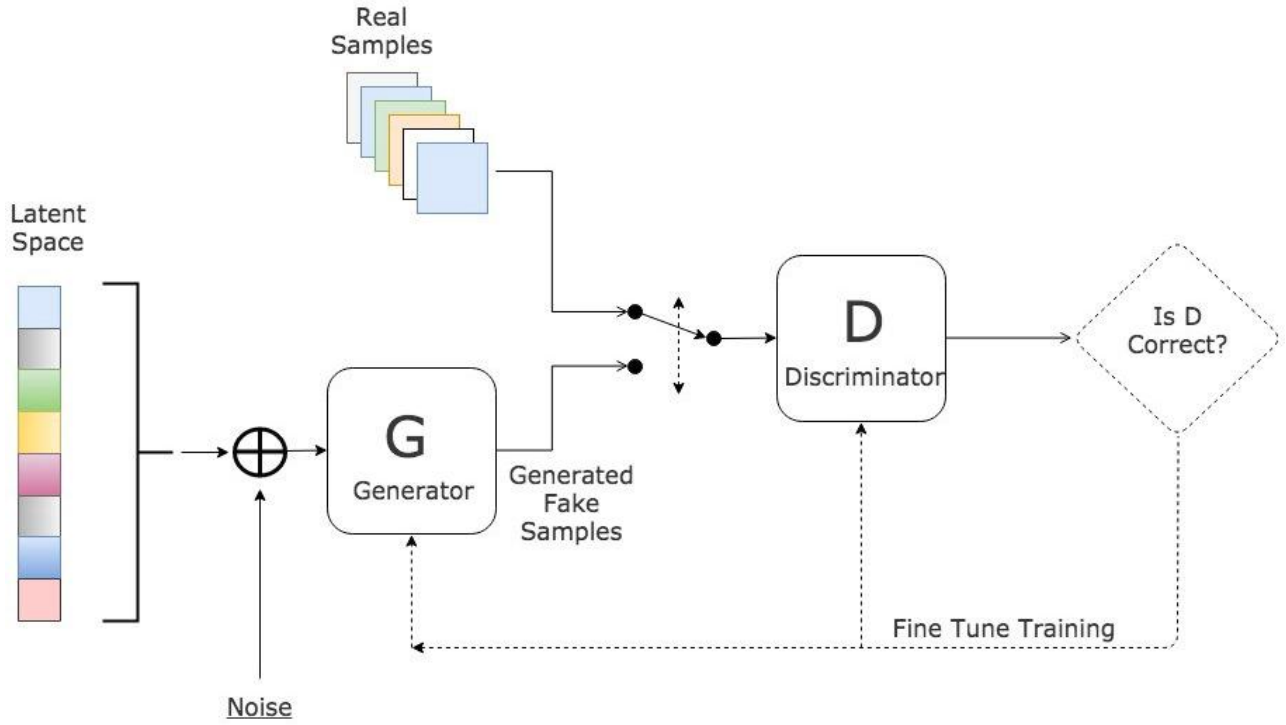
$$L_D = \frac{1}{N} \sum_{x_i \in minibatch(X_N)} -\log\big(D(x_i)\big) + \frac{1}{N} \sum_{z_i \in minibatch(N)} -\log\Big(1 - D\big(G(z_i)\big)\Big)$$

Vanilla GAN architecture

To train a GAN, we need five components.

5. **Loss L_G on generator G**

The purpose of G is to create images that are good enough to fool the discriminator D.



$$L_G = \frac{1}{N} \sum_{z_i \in minibatch(N)} -1.\log\Big(D\big(G(z_i)\big)\Big)$$

Vanilla GAN architecture

To train a GAN, we need five components.

5. Loss L_G on generator G

The purpose of G is to create images that are good enough to fool the discriminator D.

Intuitively,

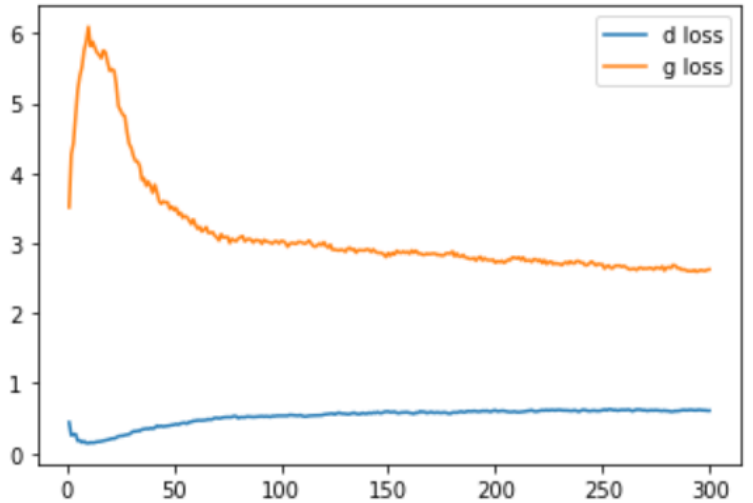
- This loss checks that the discriminator classifies the **generated samples $G(z_i)$**
- as real samples (1), instead of fake ones (0).

$$L_G = \frac{1}{N} \sum_{z_i \in minibatch(N)} -1 . \log\left(D\left(G(z_i)\right)\right)$$

GAN training visuals

- After training, we can visualize the losses.
- Convergence seems to be happening on the losses.
- (A few more iterations would have probably been good).

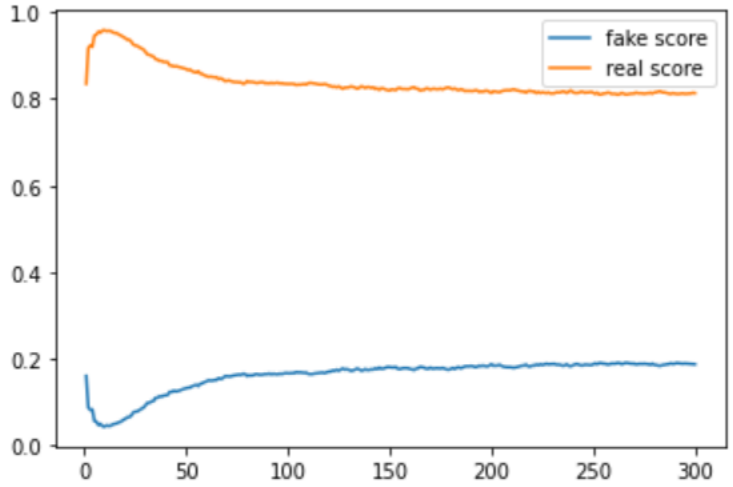
```
# Display losses for both the generator and discriminator
plt.figure()
plt.plot(range(1, num_epochs + 1), d_losses, label = 'd loss')
plt.plot(range(1, num_epochs + 1), g_losses, label = 'g loss')
plt.legend()
plt.show()
```



GAN training visuals

- After training, we can visualize the accuracy scores of D on both the real and fake samples.
- Convergence seems to be happening on the accuracies as well.

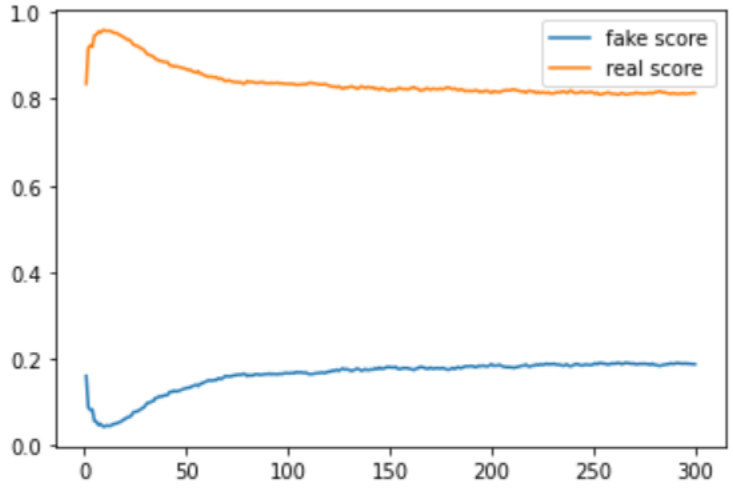
```
# Display accuracy scores for both the generator and discriminator
plt.figure()
plt.plot(range(1, num_epochs + 1), fake_scores, label='fake score')
plt.plot(range(1, num_epochs + 1), real_scores, label='real score')
plt.legend()
plt.show()
```



GAN training visuals

- After training, we can visualize the accuracy scores of D on both the real and fake samples.
- Convergence seems to be happening on the accuracies as well.
- It does not seem that G was able to completely fool D (50% score accuracy as optimal target?)
- **But do we care?**

```
:
1  # Display accuracy scores for both the generator and discriminator
2  plt.figure()
3  plt.plot(range(1, num_epochs + 1), fake_scores, label='fake score')
4  plt.plot(range(1, num_epochs + 1), real_scores, label='real score')
5  plt.legend()
6  plt.show()
```



GAN training visuals

- It does not seem that G was able to completely fool D (50% score accuracy as optimal target?)

But do we care?

- No, because, what matters is that we get a generator G that is trained well enough to generate **plausible (!) images**!

```
# Generate a few fake samples (5 of them) for visualization
n_samples = 5
z = torch.randn(n_samples, latent_size).cuda()
z = Variable(z)
fake_images = G(z)
fake_images = fake_images.cpu().detach().numpy().reshape(n_samples, 28, 28)
print(fake_images.shape)
```

```
(5, 28, 28)
```

```
# Display
plt.figure()
plt.imshow(fake_images[0])
plt.show()
plt.figure()
plt.imshow(fake_images[1])
plt.show()
plt.figure()
plt.imshow(fake_images[2])
plt.show()
plt.figure()
plt.imshow(fake_images[3])
plt.show()
plt.figure()
plt.imshow(fake_images[4])
plt.show()
```

GAN training visuals

```
# Generate a few fake samples (5 of them) for visualization
n_samples = 5
z = torch.randn(n_samples, latent_size).cuda()
z = Variable(z)
fake_images = G(z)
fake_images = fake_images.cpu().detach().numpy().reshape(n_samples, 28, 28)
print(fake_images.shape)
```

```
(5, 28, 28)
```

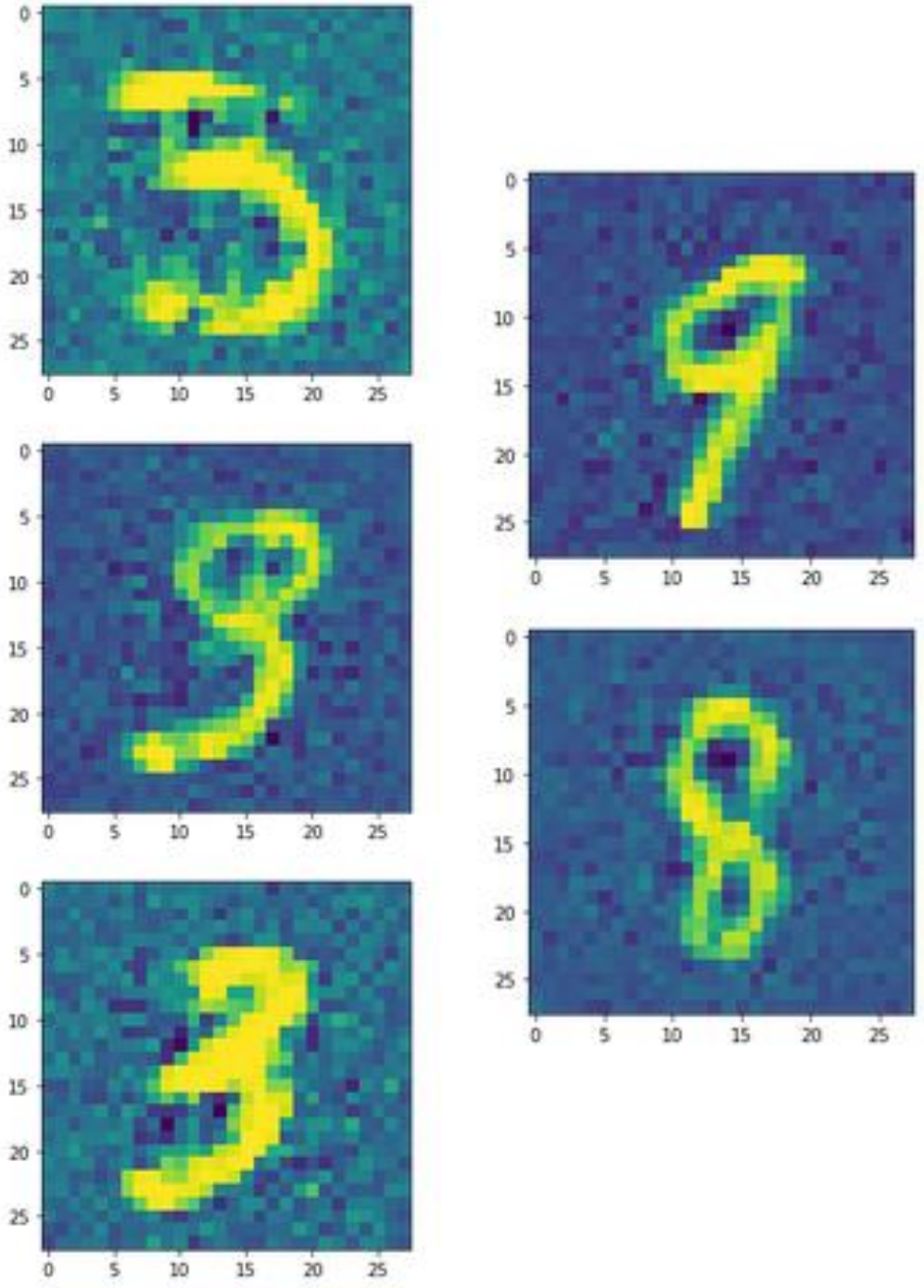
```
# Display
plt.figure()
plt.imshow(fake_images[0])
plt.show()
plt.figure()
plt.imshow(fake_images[1])
plt.show()
plt.figure()
plt.imshow(fake_images[2])
plt.show()
plt.figure()
plt.imshow(fake_images[3])
plt.show()
plt.figure()
plt.imshow(fake_images[4])
plt.show()
```

GAN training visuals

- It does not seem that G was able to completely fool D (50% score accuracy as optimal target?)

But do we care?

- No, because, what matters is that we get a generator G that is trained well enough to generate **plausible (!) images**!
- Overall, plausible images!
- Could remove noise, with TransposeConv2d instead of FC?



Typical issues with vanilla GANs

Vanilla GANs are difficult to train for several reasons.

- **Reason #1 – Convergence of interleaved training is not guaranteed:** we have to alternate between training the discriminator and the generator, but the convergence of said process is not guaranteed.
- **Reason #2 – Convergence is not necessarily the best possible outcome in terms of generator performance.**

Interleaved training

- Optimize discriminator strategy (train), with generator strategy fixed.
- Optimize generator strategy (train), with discriminator strategy fixed.
- Repeat until convergence on both players strategies (= training convergence)

Typical issues with vanilla GANs

Definition (Nash Equilibrium):

In game theory, a **Nash Equilibrium** is the most common way to define the endgame solution of a non-cooperative game involving **two or more players**.

The Nash equilibrium describes a situation where no player is interested to change its own strategy anymore.

Interleaved training

- Optimize **discriminator** strategy (train), with generator strategy fixed.
- Optimize **generator** strategy (train), with discriminator strategy fixed.
- **Repeat until convergence on both players strategies (= training convergence)**

Typical issues with vanilla GANs

Problem: Well-known for people who have done a bit of game theory…!

Nash Equilibrium
(State where both players have achieved stable strategies)

$\neq$

Optimal outcome state
(State where players have the best strategies for themselves)

Typical issues with vanilla GANs

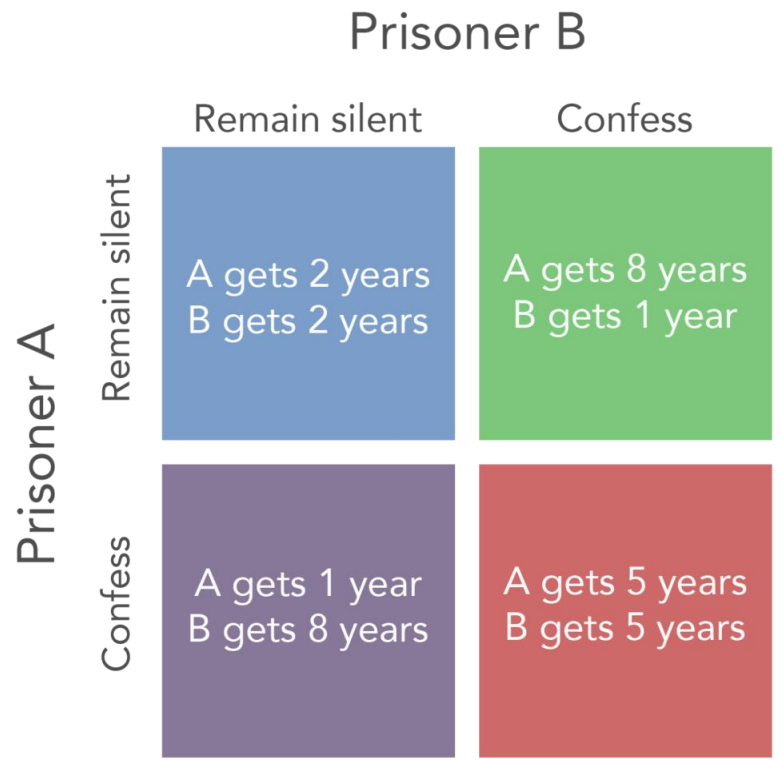
Problem: Well-known for people who have done a bit of game theory…!

Nash Equilibrium
(State where both players have achieved stable strategies)

$\neq$

Optimal outcome state
(State where players have the best strategies for themselves)

- **Counterexample:** the infamous prisoner's dilemma (two prisoners as players).



Typical issues with vanilla GANs

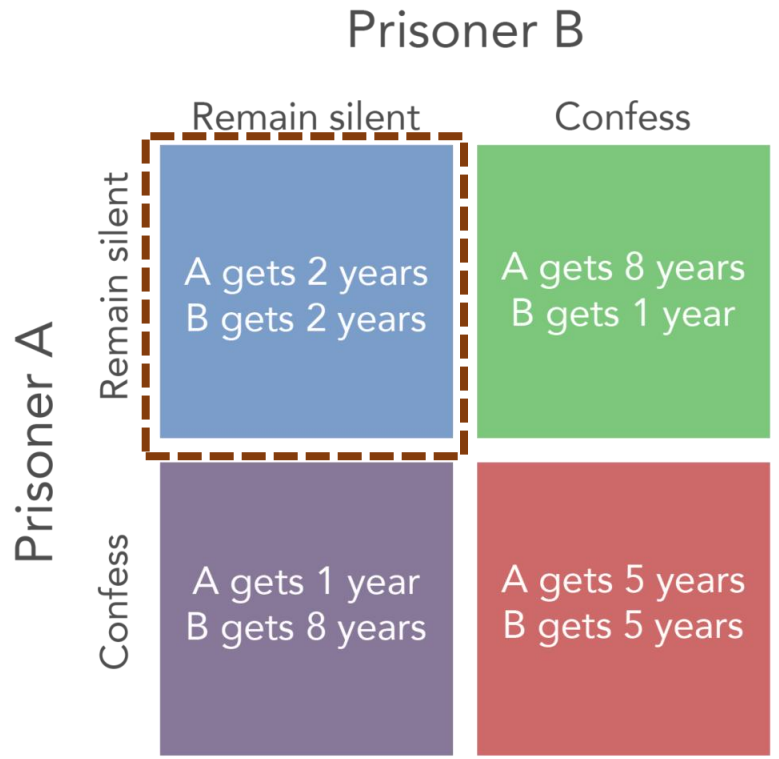
Problem: Well-known for people who have done a bit of game theory…!

Nash Equilibrium
(State where both players have achieved stable strategies)

$\neq$

Optimal outcome state
(State where players have the best strategies for themselves)

- **Counterexample:** the infamous prisoner's dilemma (two prisoners as players).



Typical issues with vanilla GANs

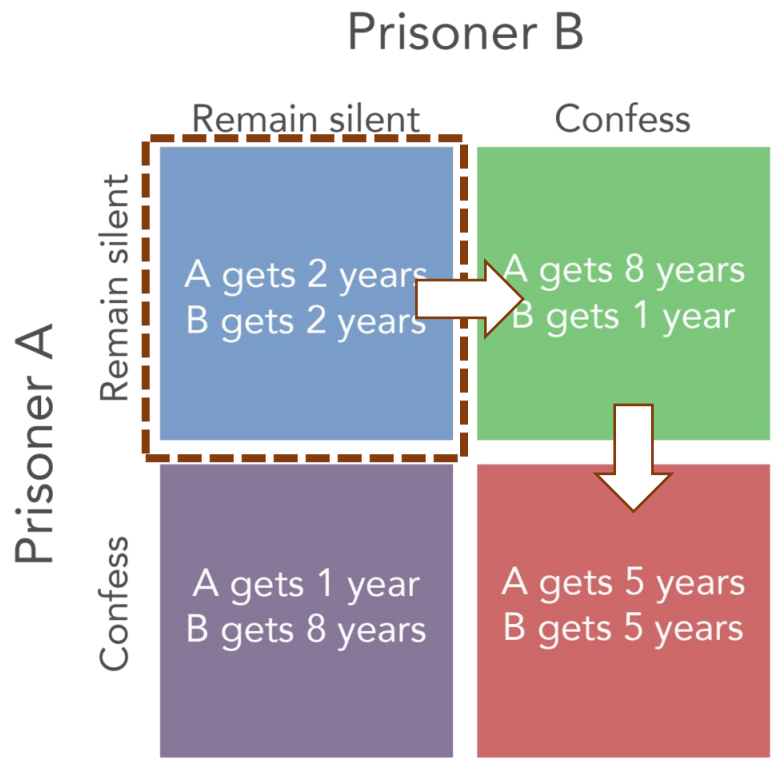
Problem: Well-known for people who have done a bit of game theory…!

Nash Equilibrium
(State where both players have achieved stable strategies)

$\neq$

Optimal outcome state
(State where players have the best strategies for themselves)

- **Counterexample:** the infamous prisoner's dilemma (two prisoners as players).



Typical issues with vanilla GANs

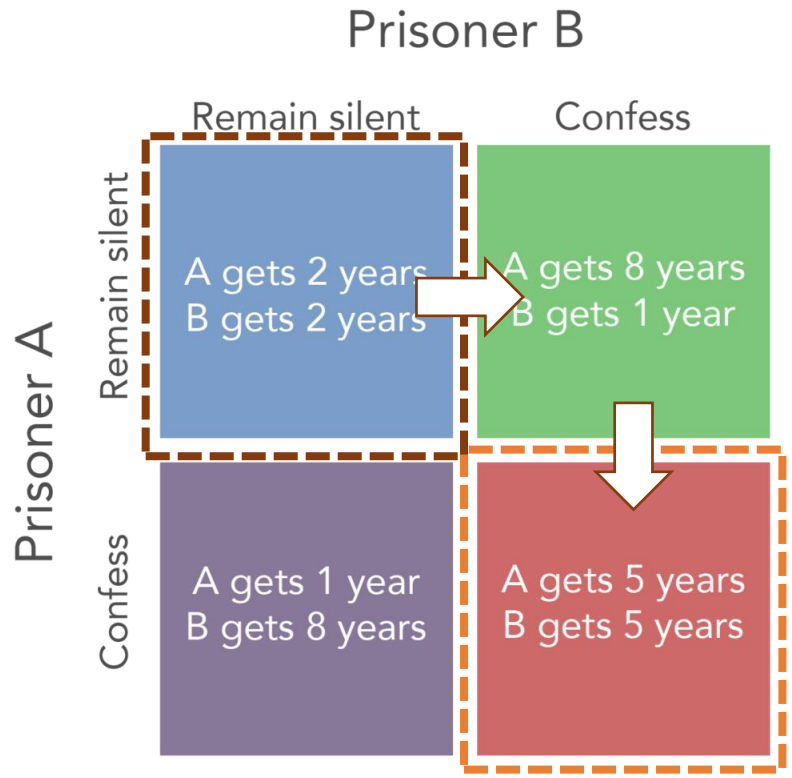
Problem: Well-known for people who have done a bit of game theory…!

Nash Equilibrium
(State where both players have achieved stable strategies)

$\neq$

Optimal outcome state
(State where players have the best strategies for themselves)

- **Counterexample:** the infamous prisoner's dilemma (two prisoners as players).



Typical issues with vanilla GANs

Vanilla GANs are difficult to train for several reasons.

- **Reason #1 – Convergence of interleaved training is not guaranteed:** we have to alternate between training the discriminator and the generator, but the convergence of said process is not guaranteed.
- **Reason #2 – Convergence is not necessarily the best possible outcome in terms of generator performance.**

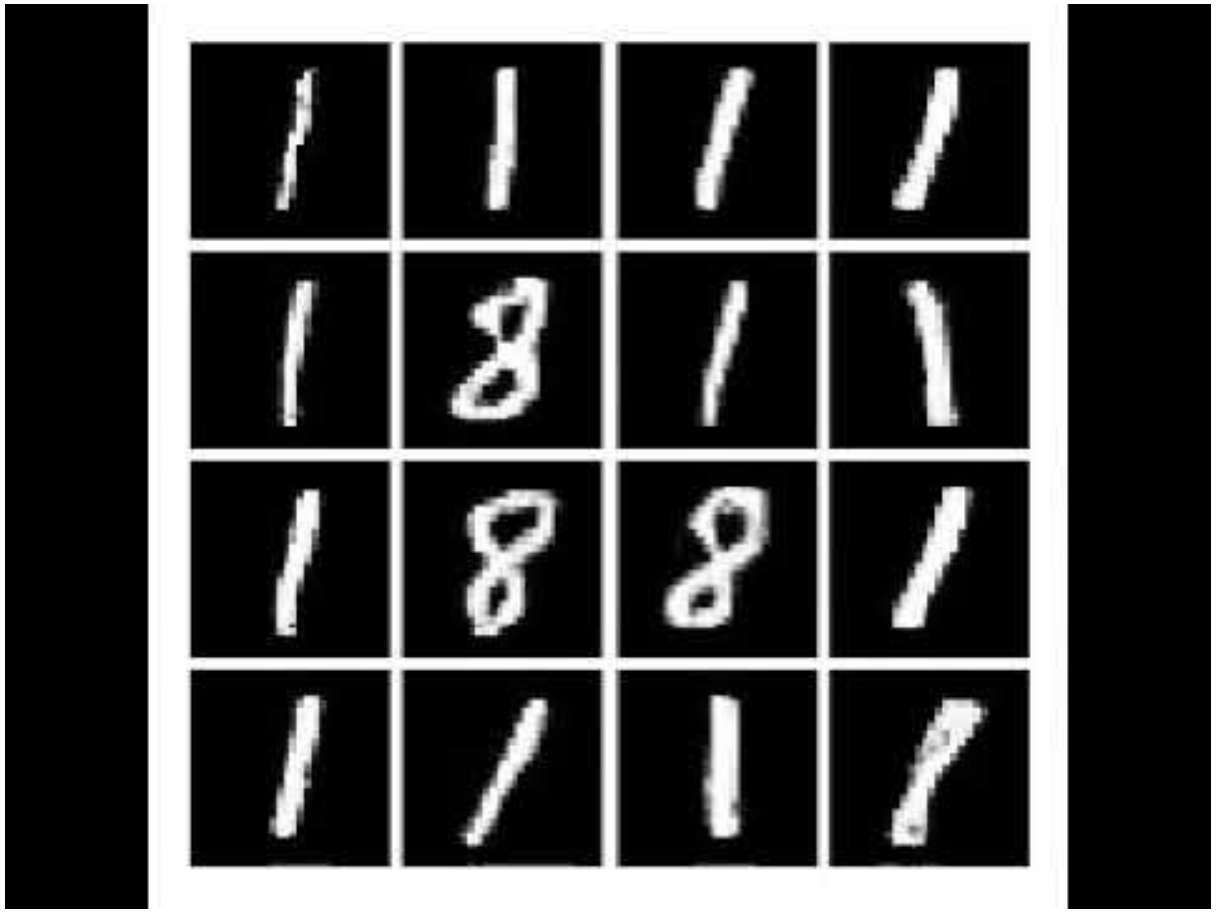
Interleaved training (discriminator and generator are the two players)

- Optimize **discriminator** strategy (train), with generator strategy fixed.
- Optimize **generator** strategy (train), with discriminator strategy fixed.
- **Repeat until convergence on both players strategies (= training convergence)**

Typical issues with vanilla GANs

Vanilla GANs are difficult to train for several reasons.

- **Reason #3 – Mode Collapse:** Mode Collapse refers to situations where the generator has identified it can very easily fool the discriminator by producing the same image (with small variations) over and over again.
- This is considered a **loss of generality** for the generator.



Typical issues with vanilla GANs

Vanilla GANs are difficult to train for several reasons.

- **Reason #4 – Oversensitivity to (unlucky) hyperparameters settings:** Something that has been empirically observed.
- For more info https://jonathan-hui.medium.com/gan-why-it-is-so-hard-to-train-generative-advisory-networks-819a86b3750b

Typical issues with vanilla GANs

Vanilla GANs are difficult to train for several reasons.

- **Reason #4 – Oversensitivity to (unlucky) hyperparameters settings:** Something that has been empirically observed.
- For more info https://jonathan-hui.medium.com/gan-why-it-is-so-hard-to-train-generative-advisory-networks-819a86b3750b
- **Reason #5 – Imbalance in discriminator/generator performance:** If the generator (or discriminator) is too good at its task, it might be difficult for the other model to catch up and learn properly.
- **One of the two models might require more training than the other to catch up!**

Typical issues with vanilla GANs

Vanilla GANs are difficult to train for several reasons.

- **Reason #4 – Oversensitivity to (unlucky) hyperparameters settings:** Something that has been empirically observed.
- For more info https://jonathan-hui.medium.com/gan-why-it-is-so-hard-to-train-generative-advisory-networks-819a86b3750b

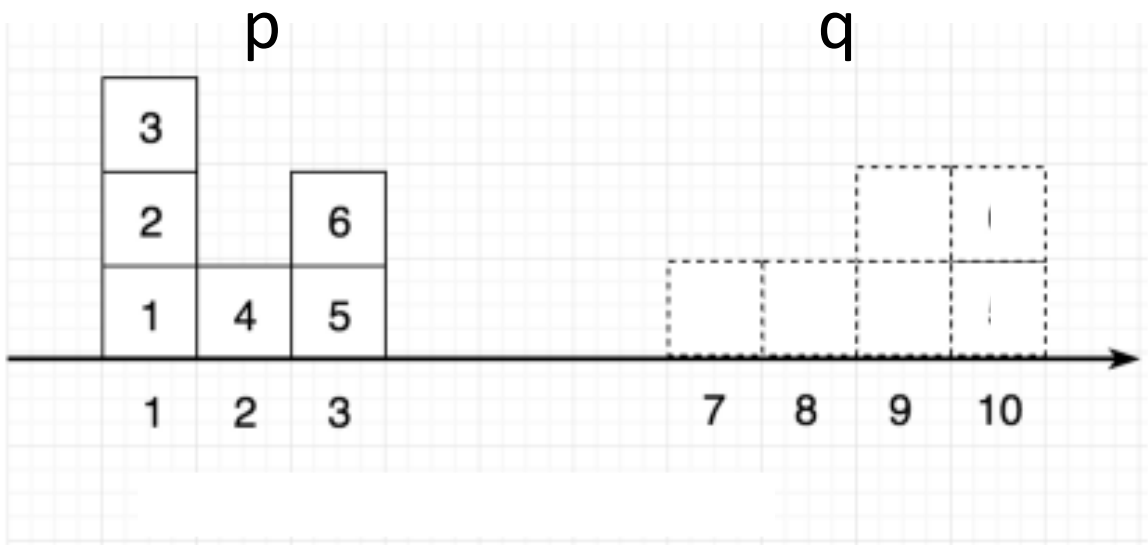
- **Reason #5 – Imbalance in discriminator/generator performance:** If the generator (or discriminator) is too good at its task, it might be difficult for the other model to catch up and learn properly.
- **One of the two models might require more training than the other to catch up!**
- **Especially true when generator is bad at making images!**

Definition: Wasserstein distance

Definition (Wasserstein distance):

Just like the KL and JS divergence metrics, the **Wasserstein distance** (a.k.a. **Earth-movers distance**) is a metric, which can be used to measure **the distance between two distributions p and q**.

When p and q are discrete distributions, it is defined as the minimal transport needed to convert p into q.

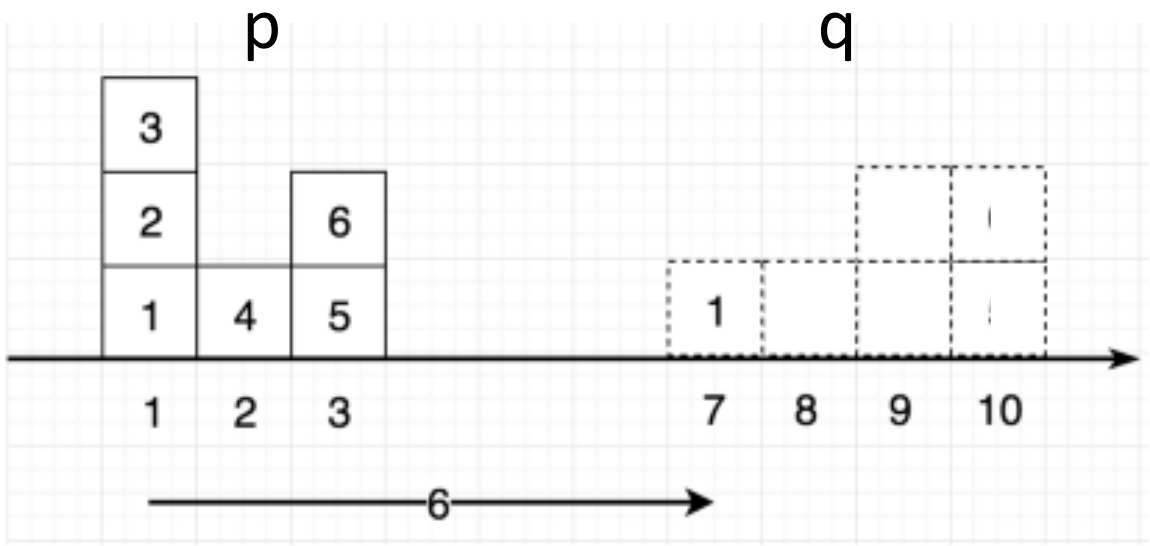


Definition: Wasserstein distance

Definition (Wasserstein distance):

Just like the KL and JS divergence metrics, the **Wasserstein distance** (a.k.a. **Earth-movers distance**) is a metric, which can be used to measure **the distance between two distributions p and q**.

When p and q are discrete distributions, it is defined as the minimal transport needed to convert p into q.

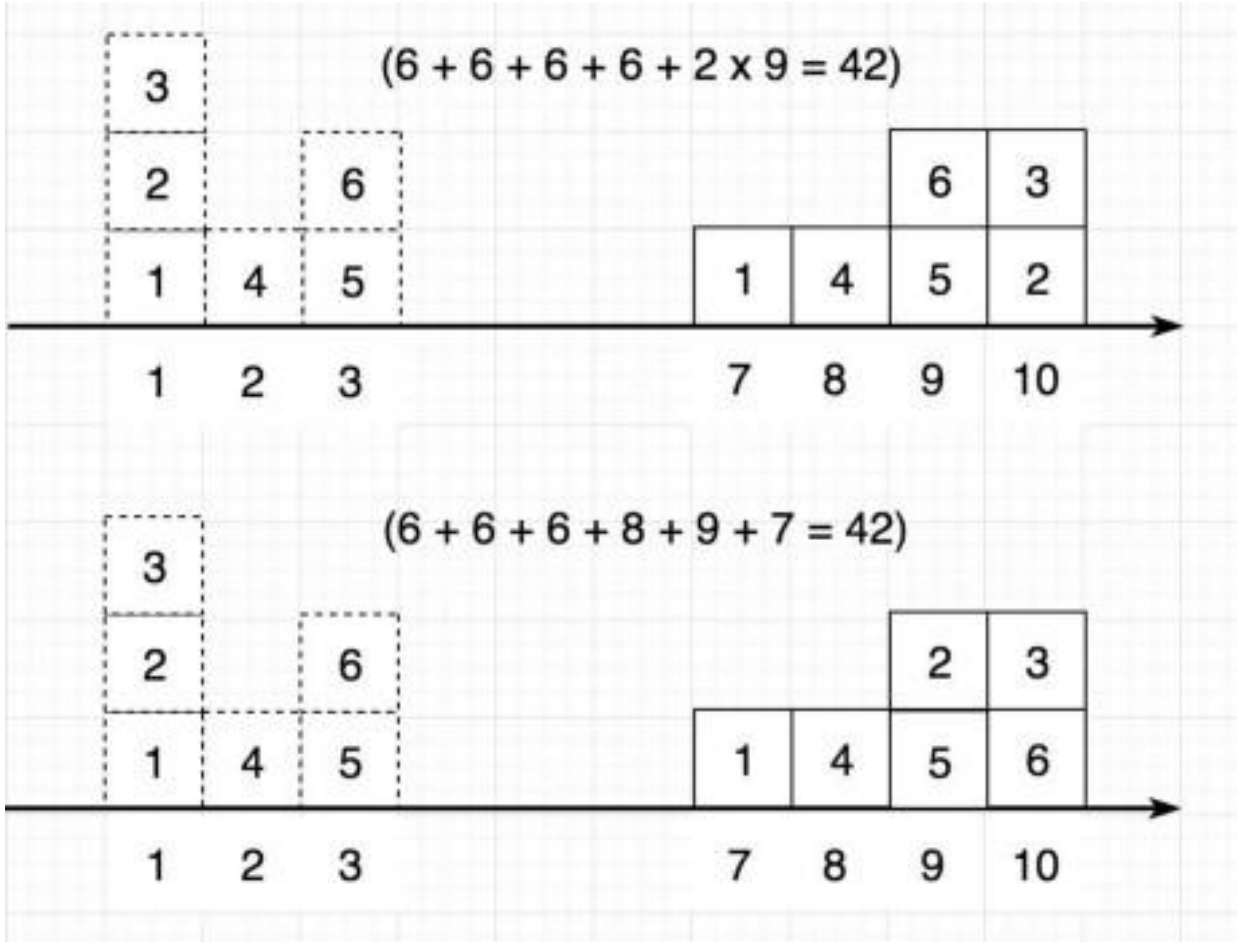


Definition: Wasserstein distance

Definition (Wasserstein distance):

Just like the KL and JS divergence metrics, the **Wasserstein distance** (a.k.a. **Earth-movers distance**) is a metric, which can be used to measure **the distance between two distributions p and q**.

When p and q are discrete distributions, it is defined as the minimal transport needed to convert p into q.

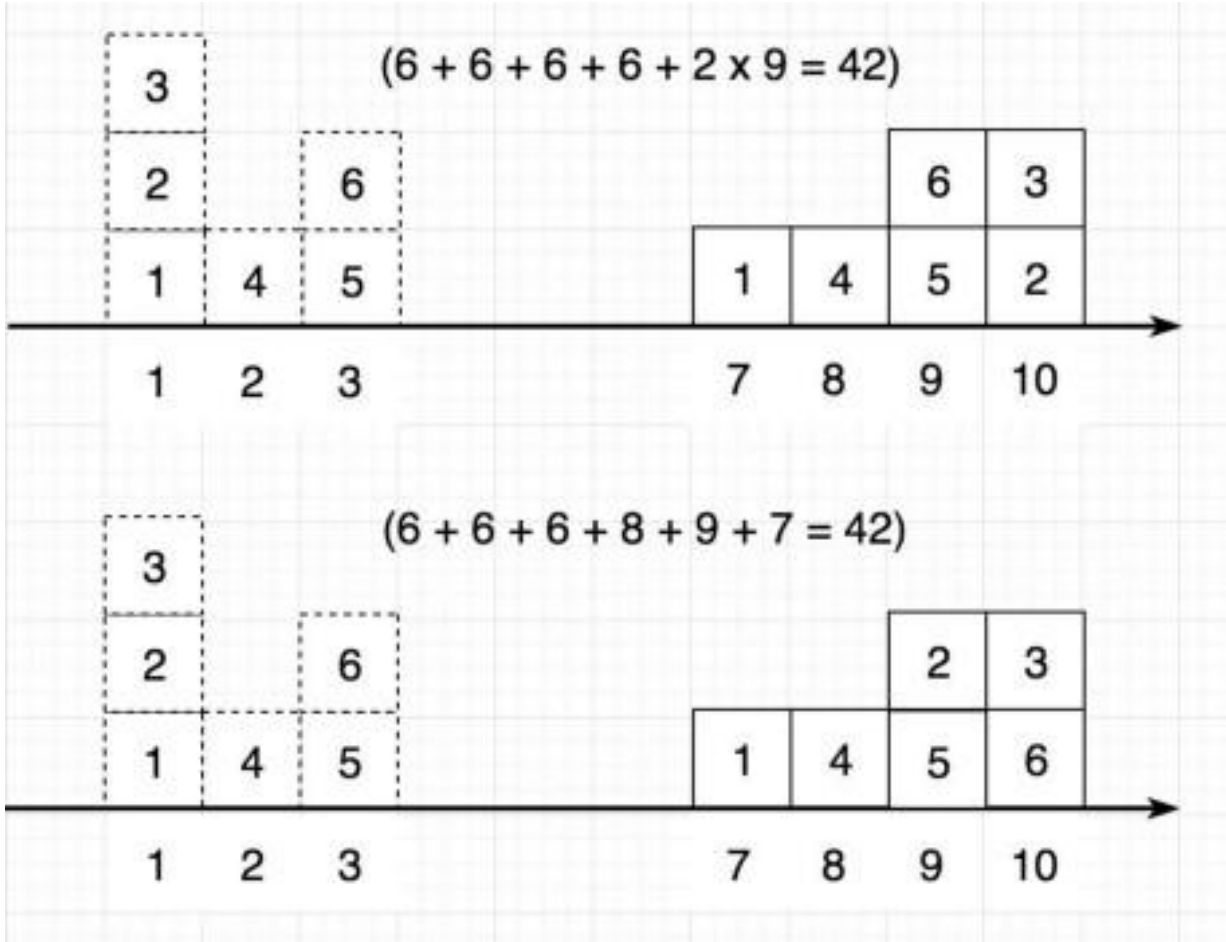


Definition: Wasserstein distance

Definition (Wasserstein distance):

Just like the KL and JS divergence metrics, the **Wasserstein distance** (a.k.a. **Earth-movers distance**) is a metric, which can be used to measure **the distance between two distributions p and q**.

When p and q are discrete distributions, it is defined as the minimal transport needed to convert p into q.

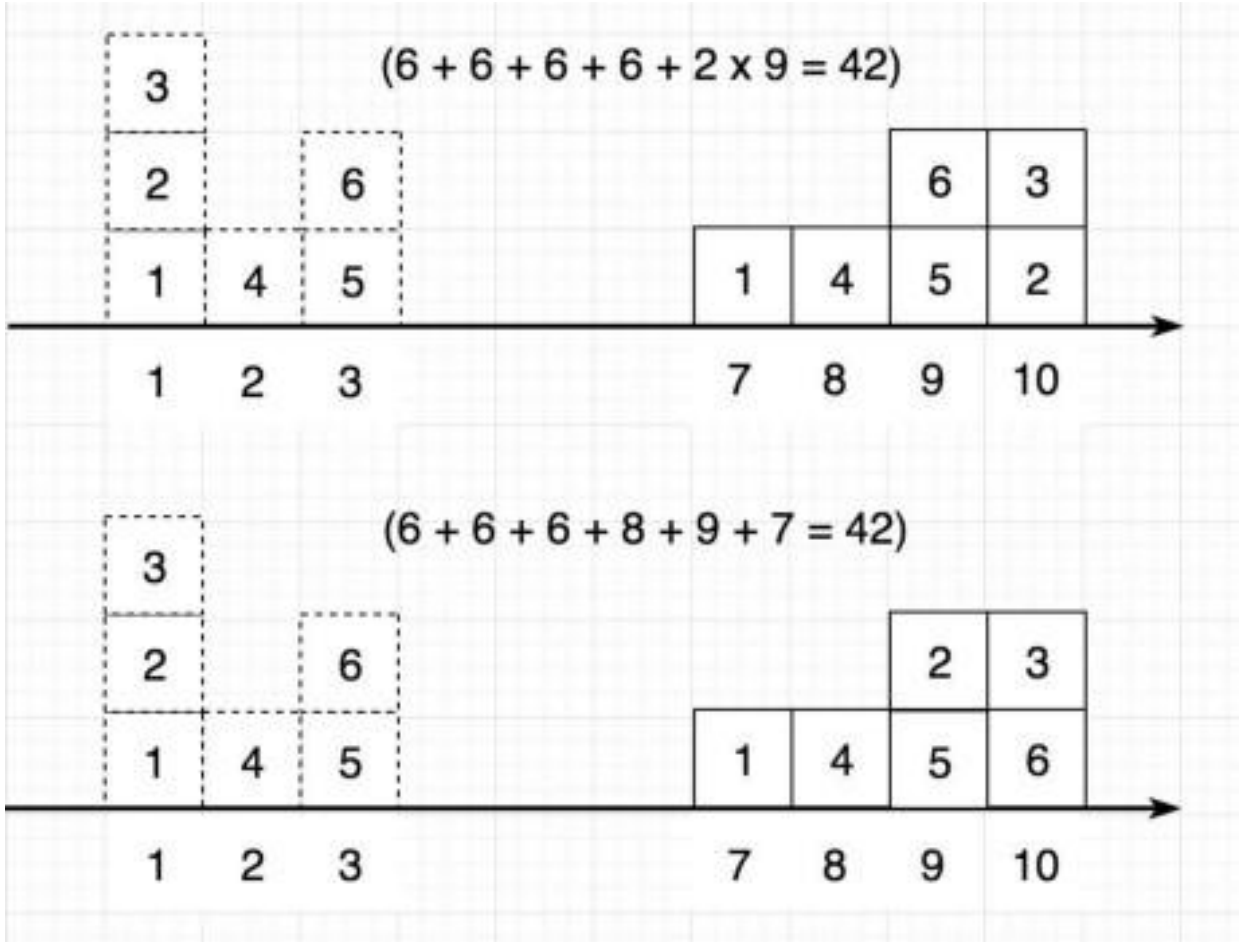


Definition: Wasserstein distance

Definition (Wasserstein distance):

Just like the KL and JS divergence metrics, the **Wasserstein distance** (a.k.a. **Earth-movers distance**) is a metric, which can be used to measure **the distance between two distributions p and q**.

When p and q are discrete distributions, it is defined as the minimal transport needed to convert p into q.

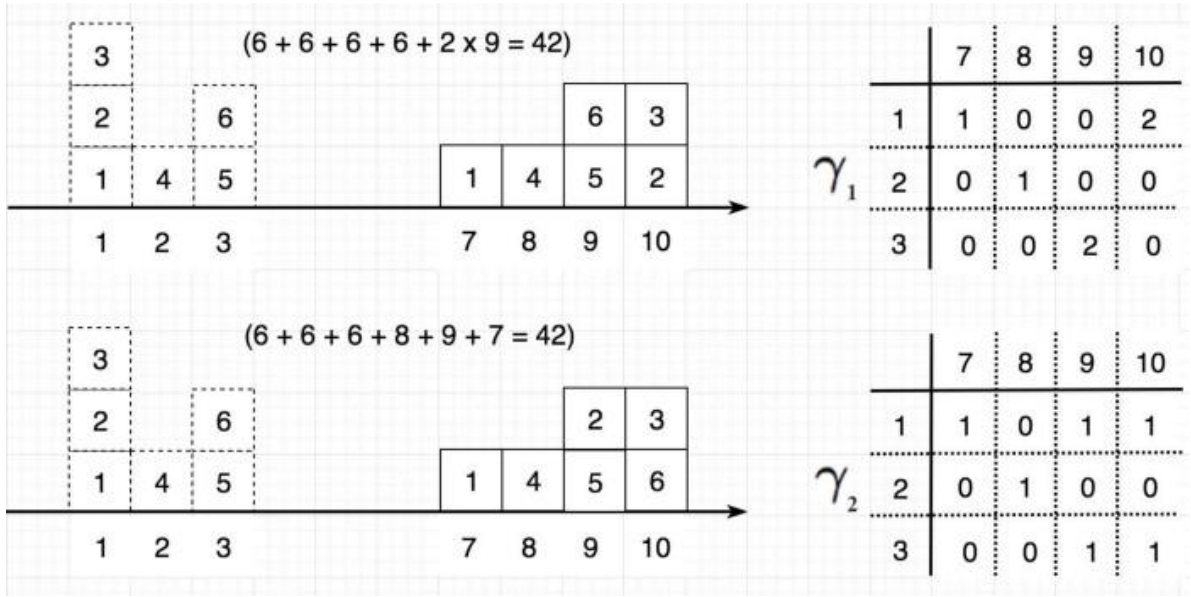


How to solve this optimization problem?

Definition: Wasserstein distance

- In practice, we like to define a **transport matrix γ**, which describes the transport strategy.

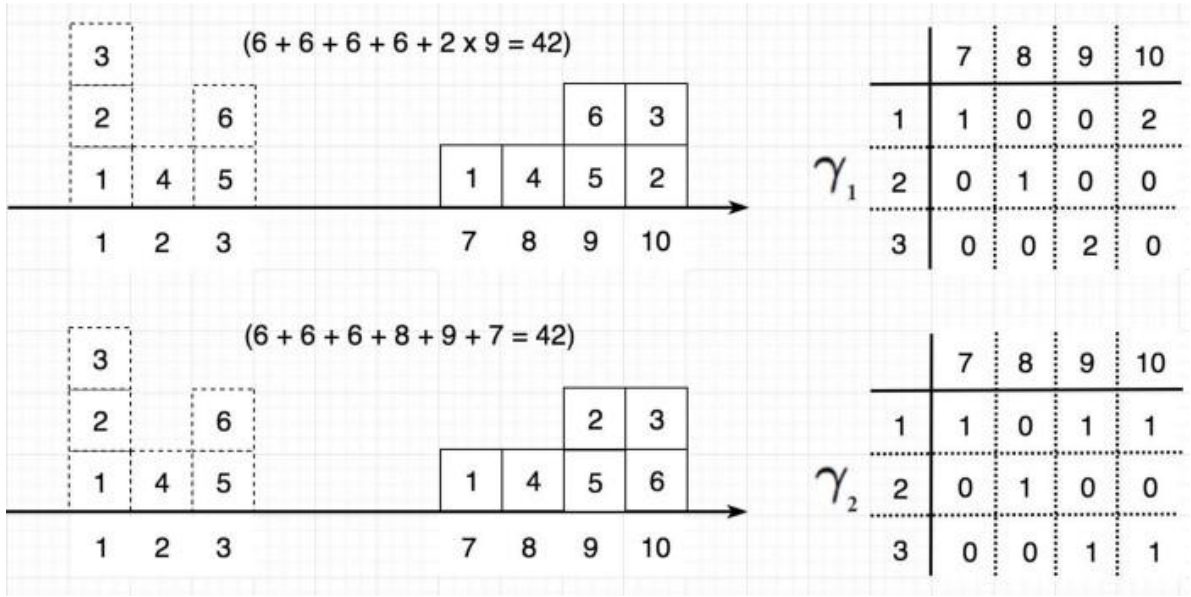
$$\gamma_{i,j} = number\ of\ samples\ to\ move\ from\ position\ i\ to\ j$$



Definition: Wasserstein distance

- In practice, we like to define a **transport matrix $\boldsymbol{\gamma}$**, which describes the transport strategy.

$$\gamma_{i,j} = number\ of\ samples\ to\ move\ from\ position\ i\ to\ j$$



- The **distance $\boldsymbol{d_{\gamma(p,q)}}$** for a strategy γ is then simply defined as

$$d_{\gamma(p,q)} = \sum_{i,j} \gamma_{i,j} \|i - j\|_2$$

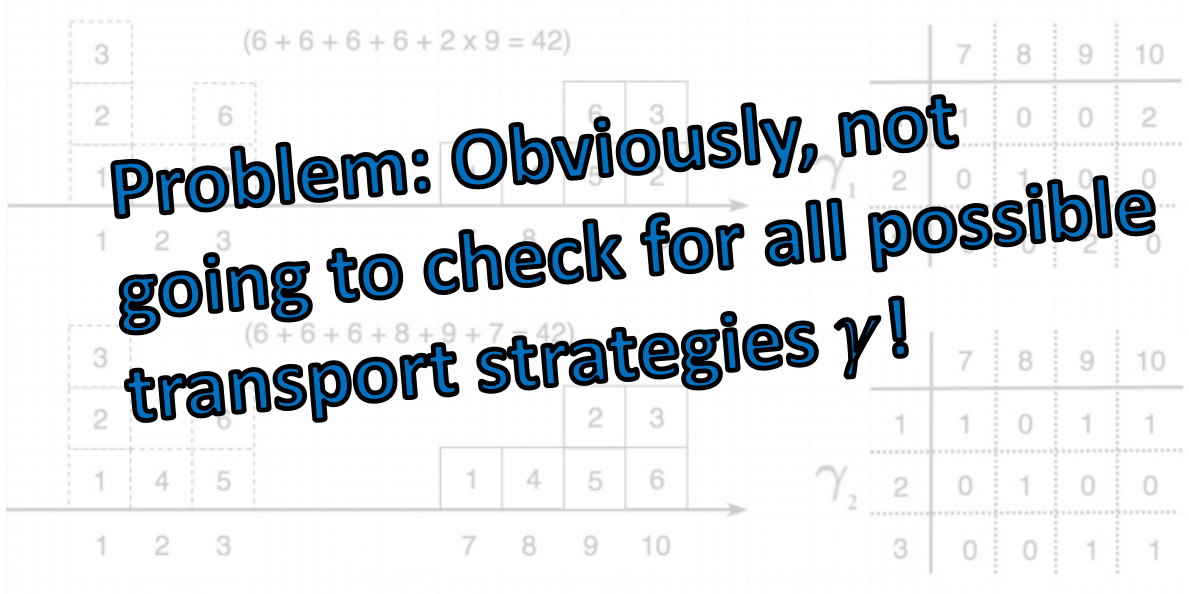
- And the **Wasserstein distance $\boldsymbol{d_W(p,q)}$** is then defined as

$$d_W(p,q) = \min_{\gamma} d_{\gamma(p,q)}$$

Definition: Wasserstein distance

- In practice, we like to define a **transport matrix γ**, which describes the transport strategy.

$$\gamma_{i,j} = number\ of\ samples\ to\ move\ from\ position\ i\ to\ j$$



- The **distance $d_{\gamma(p,q)}$** for a strategy γ is then simply defined as

$$d_{\gamma(p,q)} = \sum_{i,j} \gamma_{i,j} \|i - j\|_2$$

- And the **Wasserstein distance $d_W(p, q)$** is then defined as

$$d_W(p,q) = \min_{\gamma} d_{\gamma(p,q)}$$

Definition: Wasserstein distance

- **Kantorovich-Rubinstein duality theorem:** We have

$$d_W(p,q) = \max_{f \in \backslash \mathrm{Lip}_1} \left[E_{x \to p}[f(x)] - E_{y \to q}[f(y)] \right]$$

Proof: Chapter 5 of Villani, "Optimal Transport: Old and New" [Villani2008].

Definition: Wasserstein distance

- **Kantorovich-Rubinstein duality theorem:** We have

$$d_W(p,q) = \max_{f \in \backslash \mathrm{Lip}_1} \left[E_{x \to p}[f(x)] - E_{y \to q}[f(y)] \right]$$

Proof: Chapter 5 of Villani, "Optimal Transport: Old and New" [Villani2008].

- **Definition (K-Lipschitz function):**

A function f is **K-Lipschitz** (of Lip_K) if its derivative is bounded by K, with $K > 0$, which is equivalent to

$$\forall x, |f'(x)| < K \qquad or \qquad \forall x_1, x_2, |f(x_1) - f(x_2)| < K\,|x_1 - x_2|$$

Wasserstein GAN idea

Let us call

- X the dataset distribution,
- and $\widehat{X_G}$ the distribution produced by the generator G.

Objective for generator G:
generate samples, which look similar, in terms of distribution, to the ones in the dataset.

Wasserstein GAN idea

Let us call

- X the dataset distribution,
- and $\widehat{X_G}$ the distribution produced by the generator G.

Objective for generator G:
generate samples, which look similar, in terms of distribution, to the ones in the dataset.

The discriminator D will not be able to tell apart the fake samples from the real ones, if both distributions are similar, i.e.

$$X \approx \widehat{X_G}$$

or equivalently if

$$d_W\left(X, \widehat{X_G}\right) \approx 0$$

Wasserstein GAN idea

Let us call

- X the dataset distribution,
- and $\widehat{X_G}$ the distribution produced by the generator G.

Kantorovich-Rubinstein duality theorem:

$$d_W\left(X, \widehat{X_G}\right) = \max_{f \in \backslash \mathrm{Lip}_1} \left[E_{x \to X}[f(x)] - E_{y \to \widehat{X_G}}[f(y)]\right]$$

$$d_W\left(X, \widehat{X_G}\right) = \max_{f \in \backslash \mathrm{Lip}_1} \left[E_{x \to X}[f(x)] - E_{z \to N(0_K, I_K)}[f\left(G(z)\right)]\right]$$

Wasserstein GAN idea

$$d_W\left(X, \widehat{X_G}\right) = \max_{f \in \backslash \mathrm{Lip}_1} \left[E_{x \to X}[f(x)] - E_{z \to N(0_K, I_K)}[f\left(G(z)\right)]\right]$$

This can be approximated with mini-batches as

$$d_W\left(X, \widehat{X_G}\right) = \max_{f \in \backslash \mathrm{Lip}_1} \left[\frac{1}{N} \sum_{x_i \in minibatch(X,N)} f(x_i) - \frac{1}{\mathrm{N}} \sum_{z_i \to N(0_K, I_K)} f\left(G(z_i)\right)\right]$$

$$d_W\left(X, \widehat{X_G}\right) = \min_{f \in \backslash \mathrm{Lip}_1} \left[\frac{1}{\mathrm{N}} \sum_{z_i \to N(0_K, I_K)} f\left(G(z_i)\right) - \frac{1}{N} \sum_{x_i \in minibatch(X,N)} f(x_i)\right]$$

Wasserstein GAN idea

$$d_W(X, \widehat{X_G}) = \max_{f \in \backslash \mathrm{Lip}_1} \left[E_{x \to X}[f(x)] - E_{z \to N(0_K, I_K)}[f(G(z))] \right]$$

This can be approximated with mini-batches as

$$d_W(X, \widehat{X_G}) = \max_{f \in \backslash \mathrm{Lip}_1} \left[\frac{1}{N} \sum_{x_i \in minibatch(X,N)} f(x_i) - \frac{1}{N} \sum_{z_i \to N(0_K, I_K)} f(G(z_i)) \right]$$

Problem: Obviously, not going to check for all possible Lip_1 functions in the world!

$$d_W(X, \widehat{X_G}) = \min_{f \in \backslash \mathrm{Lip}_1} \left[\frac{1}{N} \sum_{z_i \to N(0_K, I_K)} f(G(z_i)) - \frac{1}{N} \sum_{x_i \in minibatch(X,N)} f(x_i) \right]$$

Wasserstein GAN idea

$$d_W\left(X, \widehat{X_G}\right) = \min_{\mathrm{f}\in\backslash \mathrm{Lip}_1} \left[\frac{1}{\mathrm{N}} \sum_{z_i \to N(0_K, I_K)} f\left(G(z_i)\right) - \frac{1}{N} \sum_{x_i \in minibatch(X,N)} f(x_i)\right]$$

If we replace f with a Neural Network of some sort, then we could compute $d_W\left(X, \widehat{X_G}\right)$ by loss minimization of said network!

$$L_f = \left[\frac{1}{\mathrm{N}} \sum_{z_i \to N(0_K, I_K)} f\left(G(z_i)\right) - \frac{1}{N} \sum_{x_i \in minibatch(X,N)} f(x_i)\right]$$

Wasserstein GAN idea

$$d_W\left(X, \widehat{X_G}\right) = \min_{f \in \backslash \mathrm{Lip}_1} \left[\frac{1}{\mathrm{N}} \sum_{z_i \to N(0_K, I_K)} f\left(G(z_i)\right) - \frac{1}{N} \sum_{x_i \in minibatch(X,N)} f(x_i) \right]$$

If we replace f with a Neural Network of some sort, then we could compute $d_W\left(X, \widehat{X_G}\right)$ by loss minimization of said network!

$$L_f = \left[\frac{1}{\mathrm{N}} \sum_{z_i \to N(0_K, I_K)} f\left(G(z_i)\right) - \frac{1}{N} \sum_{x_i \in minibatch(X,N)} f(x_i) \right] + \lambda R(f)$$

$$R(f) = a\ regularization\ term\ penalizing\ f$$
$$if\ its\ gradients\ are\ more\ than\ 1\ (eq.to\ f\ not\ being\ Lip_1)$$

Wasserstein GAN idea

$$d_W\left(X, \widehat{X_G}\right) = \min_{f \in \backslash \mathrm{Lip}_1} \left[\frac{1}{\mathrm{N}} \sum_{z_i \to N(0_K, I_K)} f\left(G(z_i)\right) - \frac{1}{N} \sum_{x_i \in minibatch(X,N)} f(x_i) \right]$$

If we replace f with a Neural Network of some sort, then we could compute $d_W\left(X, \widehat{X_G}\right)$ by loss minimization of said network!

$$L_f = \left[\frac{1}{\mathrm{N}} \sum_{z_i \to N(0_K, I_K)} f\left(G(z_i)\right) - \frac{1}{N} \sum_{x_i \in minibatch(X,N)} f(x_i) \right]$$

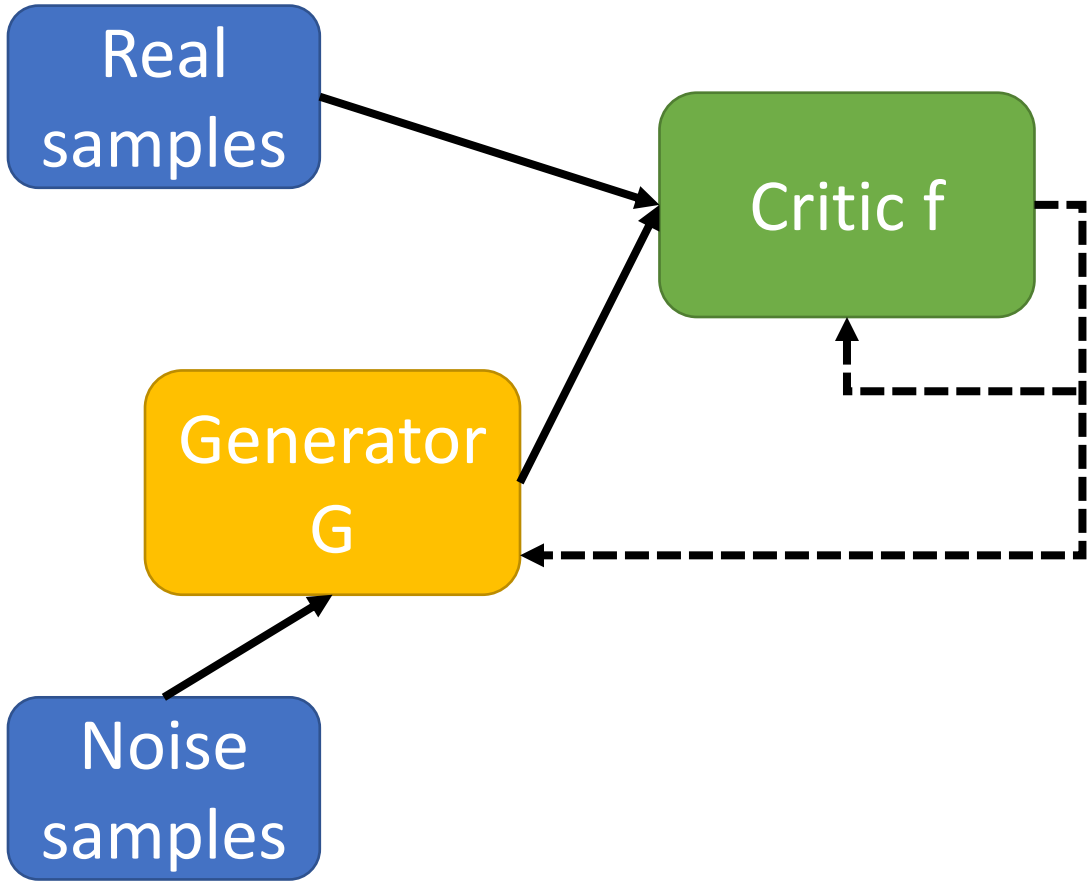
Or just use clipping with a small value,
say 0.01, on weights! (Could be dangerous!)

Wasserstein GAN idea

Definition (Critic): In Wasserstein GANs, we do not use a **discriminator** neural network D.

Instead, we use a **critic** neural network.

And by sheer coincidence (is it?), it just happens to be the Neural Network modelling the Lip_1 function f!



Wasserstein GAN idea

The objective of the critic, is to compute the Wasserstein distance.
Its loss function L_f is then:

$$L_f = \left[\frac{1}{\mathrm{N}} \sum_{z_i \to N(0_K, I_K)} f\big(G(z_i)\big) - \frac{1}{N} \sum_{x_i \in minibatch(X,N)} f(x_i)\right] + \lambda R(f)$$

$$R(f) = \frac{1}{N} \sum_{z_i \to N(0_K, I_K)} (|\nabla f(G(z_i))| - 1)^n$$

$$+ \frac{1}{N} \sum_{x_i \in minibatch(X,N)} (|\nabla f(x_i)| - 1)^n, \qquad with\ n \geq 2, n\ even$$

Wasserstein GAN idea

The objective of the generator is to bring the Wasserstein distance to 0.

$$\min_{G} \max_{f \in Lip_1} \left[\frac{1}{N} \sum_{x_i \in minibatch(X,N)} f(x_i) - \frac{1}{\mathrm{N}} \sum_{z_i \to N(0_K, I_K)} f\big(G(z_i)\big) \right]$$

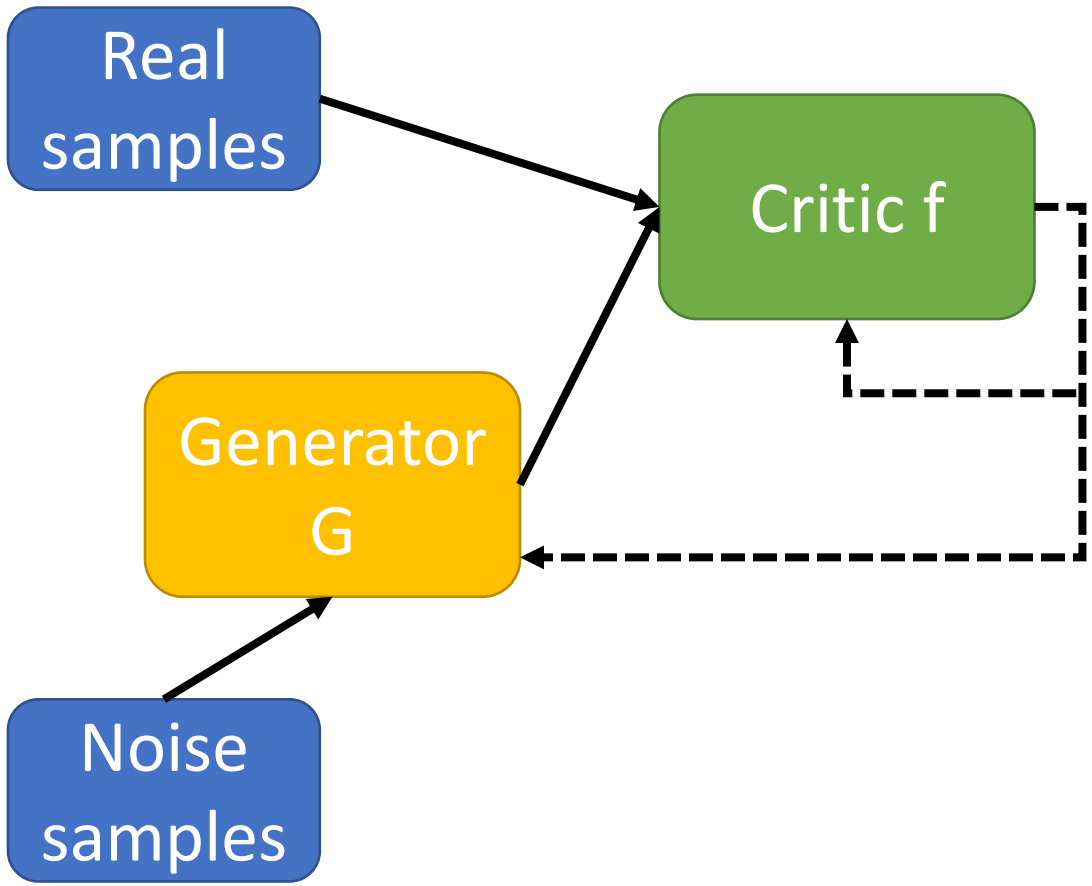
Because f is fixed and has no influence in the minimization for G, this minimization problem simply becomes (it is then the loss function for G):

$$\min_{G} \left[-\frac{1}{\mathrm{N}} \sum_{z_i \to N(0_K, I_K)} f\big(G(z_i)\big) \right]$$

Wasserstein GAN idea

Ok, what the hell is going on?!

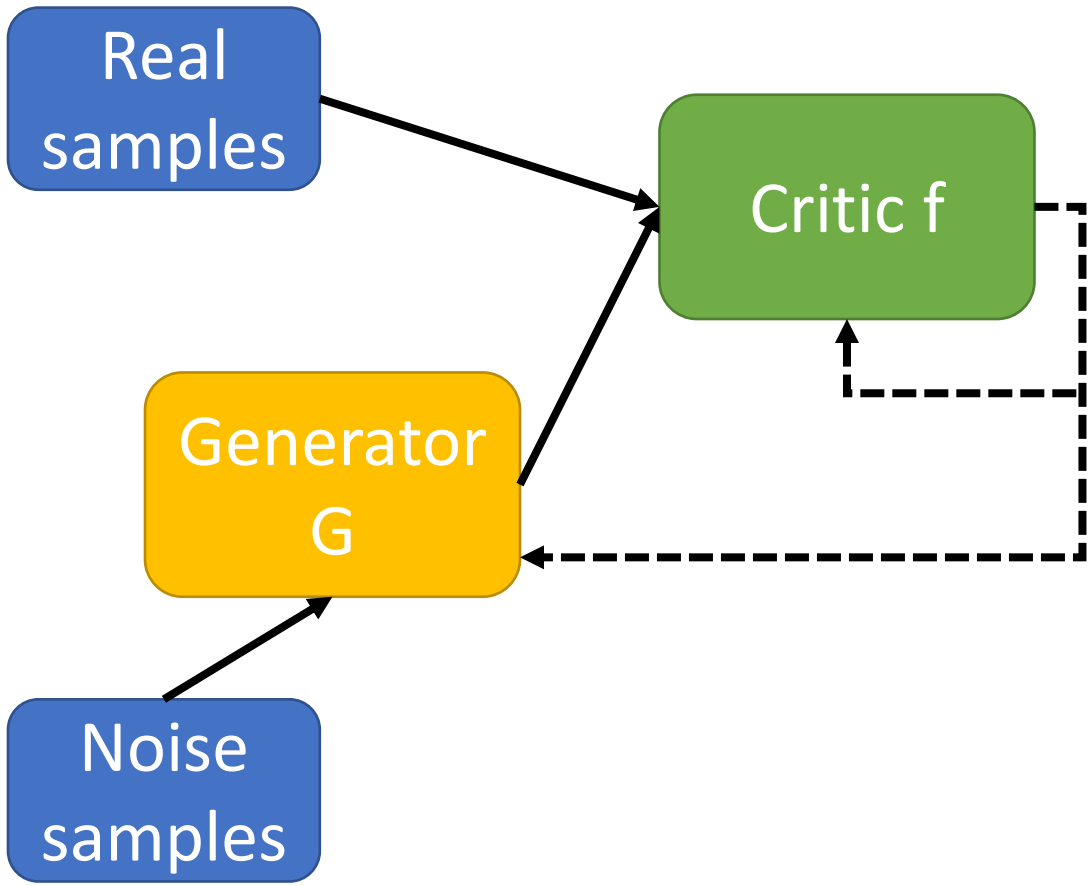
- Keep both Neural Networks structures as in vanilla GANs.
- Rename the **discriminator** as a **critic**, it should only output a score $f(x)$ for any input x (no longer a binary classification)
- Instead, the score $f(x)$ carries an information about how the input x was judged by the critic (i.e. good/bad fake sample).



Wasserstein GAN idea

This information is more useful in training the generator, as it provides a soft feedback, as opposed to a 0/1 score before.

- This helps to train a good generator!
- **Bonus:** all the loss functions are easy to compute, as they are simply mean values of the critic decisions over real/fake samples!



Wasserstein GAN implementation

```python
# Discriminator
class Dicriminator(nn.Module):

    def __init__(self, hidden_size, image_size):
        # Init from nn.Module
        super().__init__()

        # FC layers
        self.D = nn.Sequential(nn.Linear(image_size, hidden_size),
                               nn.LeakyReLU(0.2),
                               nn.Linear(hidden_size, 1),
                               nn.Sigmoid())

    def forward(self, x):
        return self.D(x)
```

```python
# Critic
class Critic(nn.Module):

    def __init__(self, hidden_size, image_size):
        # Init from nn.Module
        super().__init__()

        # FC layers
        self.D = nn.Sequential(nn.Linear(image_size, hidden_size),
                               nn.LeakyReLU(0.2),
                               nn.Linear(hidden_size, 1))

    def forward(self, x):
        return self.D(x)
```

Datasets and dataloaders do not change. Networks do not change; we simply rename the discriminator by convention and remove its final sigmoid layer.

```python
# Generator
class Generator(nn.Module):

    def __init__(self, latent_size, hidden_size, image_size):
        # Init from nn.Module
        super().__init__()

        # FC layers
        self.G = nn.Sequential(nn.Linear(latent_size, hidden_size),
                               nn.ReLU(),
                               nn.Linear(hidden_size, image_size),
                               nn.Tanh())

    def forward(self, x):
        return self.G(x)
```

Wasserstein GAN implementation

Start by processing the samples

- Generate N (mini-batch size) random noise samples, which will later be fed to generator.
- Draw N (mini-batch size) samples from the dataset X, which will be fed to critic later on.
- Flatten images (FC layers!)

```
total_step = len(data_loader)
for epoch in range(num_epochs):
    for i, (images, _) in enumerate(data_loader):

        # 1. Flatten image
        images = images.view(batch_size, -1).to(device)
        images = Variable(images, requires_grad = True)

```

Wasserstein GAN implementation

Train the **critic** (almost like before)

1. Pass real samples to f, get critic scores.
2. Compute the first half of the loss for f on these real samples.
3. Pass noise samples to generator G, and its outputs to critic f, get critic scores.
4. Compute the second half of the loss for f on these fake samples.
5. Calculate gradient penalty and add to loss
6. Backpropagate f on the computed loss.

Wasserstein GAN implementation

```
        # 2. Compute mean of critic decisions using real images
        outputs_real = f(images)

        # 2.bis. Compute mean of critic decisions using fake images
        z = torch.randn(batch_size, latent_size).to(device)
        z = Variable(z)
        fake_images = G(z)
        outputs_fake = f(fake_images)

        # 3. Compute gradient regularization
        real_grad_out = Variable(Tensor(images.size(0), 1).fill_(1.0), requires_grad = False).to(device)
        real_grad = autograd.grad(outputs_real, images, real_grad_out, create_graph = True, \
                                  retain_graph = True, only_inputs = True)[0]
        real_grad_norm = real_grad.view(real_grad.size(0), -1).pow(2).sum(1)
        fake_grad_out = Variable(Tensor(fake_images.size(0), 1).fill_(1.0), requires_grad = False).to(device)
        fake_grad = autograd.grad(outputs_fake, fake_images, fake_grad_out, create_graph = True, \
                                  retain_graph = True, only_inputs = True)[0]
        fake_grad_norm = fake_grad.view(fake_grad.size(0), -1).pow(2).sum(1)
        reg_term = torch.mean(real_grad_norm + fake_grad_norm)

        # 4. Backprop and optimize for f
        # Loss is simply the difference between means, plus regularization term
        # Remember to reset gradients for both optimizers!
        d_loss = -torch.mean(outputs_real) + torch.mean(outputs_fake) + reg_term
        d_optimizer.zero_grad()
        g_optimizer.zero_grad()
        d_loss.backward()
        d_optimizer.step()

        # 4.bis. Optional, weight clipping on critic
        # (Mentioned in WGAN paper)
        for p in f.parameters():
            p.data.clamp_(-0.01, 0.01)
```

Wasserstein GAN implementation

Train the **generator** (almost like before)

1. Produce a fresh batch of noise samples to be fed to the generator.
2. Produce fake images by feeding these noise samples to generator G.
3. Pass fake images to critic f, get critic scores for these samples.
4. Backpropagate G on the computed loss.

Wasserstein GAN implementation

```
53          # 5. Generate fresh noise samples and produce fake images
54          z = torch.randn(batch_size, latent_size).cuda()
55          z = Variable(z)
56          fake_images = G(z)
57          outputs = f(fake_images)
58
59          # 6. Loss for G
60          g_loss = - torch.mean(outputs)
61
62          # 7. Backprop and optimize G
63          # Remember to reset gradients for both optimizers!
64          d_optimizer.zero_grad()
65          g_optimizer.zero_grad()
66          g_loss.backward()
67          g_optimizer.step()
```

Wasserstein GAN implementation

Update loss history after each mini-batch.

Display on periodic epoch values for convenience.

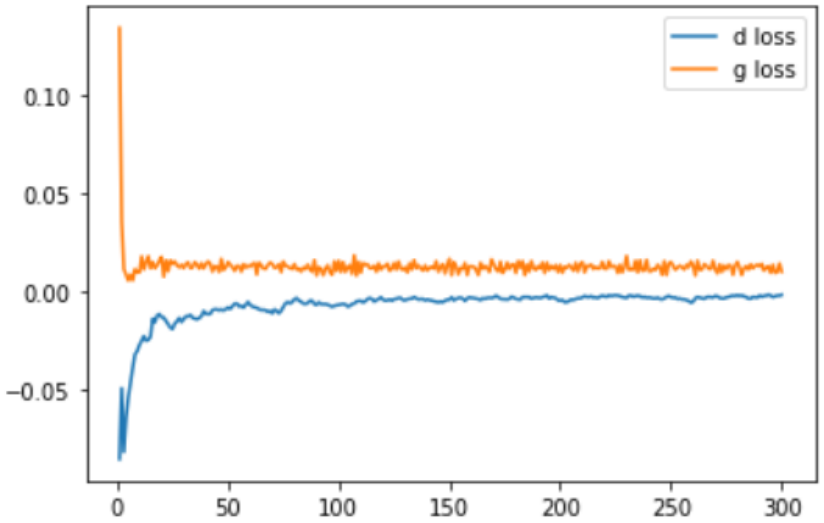
```
74          # 8. Update the losses and scores for mini-batches
75          d_losses[epoch] = d_losses[epoch]*(i/(i+1.)) \
76              + d_loss.item()*(1./(i+1.))
77          g_losses[epoch] = g_losses[epoch]*(i/(i+1.)) \
78              + g_loss.item()*(1./(i+1.))
79
80          # 9. Display
81          if (i+1) % 200 == 0:
82              print('Epoch [{}/{}], Step [{}/{}], d_loss: {:.4f}, g_loss: {:.4f}'
83                    .format(epoch, num_epochs, i+1, total_step, d_loss.item(), g_loss.item()))
```

```
Epoch [0/300], Step [200/1875], d_loss: 0.1082, g_loss: 0.0030
Epoch [0/300], Step [400/1875], d_loss: -0.2722, g_loss: 0.6567
Epoch [0/300], Step [600/1875], d_loss: -0.0265, g_loss: 0.2156
Epoch [0/300], Step [800/1875], d_loss: -0.3539, g_loss: 0.8638
Epoch [0/300], Step [1000/1875], d_loss: 0.0776, g_loss: -0.2423
Epoch [0/300], Step [1200/1875], d_loss: -0.1362, g_loss: 0.1327
Epoch [0/300], Step [1400/1875], d_loss: 0.1418, g_loss: 0.2532
Epoch [0/300], Step [1600/1875], d_loss: -0.0912, g_loss: 0.5868
Epoch [0/300], Step [1800/1875], d_loss: -0.2362, g_loss: 0.5961
Epoch [1/300], Step [200/1875], d_loss: -0.0664, g_loss: 0.1484
Epoch [1/300], Step [400/1875], d_loss: 0.0502, g_loss: 0.3070
```

WGAN training visuals

- After training, we can visualize the losses.
- Convergence seems to be happening on the losses.
- (A few more iterations would have probably been good?).

```
# Display losses for both the generator and discriminator
plt.figure()
plt.plot(range(1, num_epochs + 1), d_losses, label = 'd loss')
plt.plot(range(1, num_epochs + 1), g_losses, label = 'g loss')
plt.legend()
plt.show()
```



WGAN training visuals

```
# Generate a few fake samples (5 of them) for visualization
n_samples = 5
z = torch.randn(n_samples, latent_size).cuda()
z = Variable(z)
fake_images = G(z)
fake_images = fake_images.cpu().detach().numpy().reshape(n_samples, 28, 28)
print(fake_images.shape)
```

```
# Display
plt.figure()
plt.imshow(fake_images[0])
plt.show()
plt.figure()
plt.imshow(fake_images[1])
plt.show()
plt.figure()
plt.imshow(fake_images[2])
plt.show()
plt.figure()
plt.imshow(fake_images[3])
plt.show()
plt.figure()
plt.imshow(fake_images[4])
plt.show()
```

WGAN training visuals

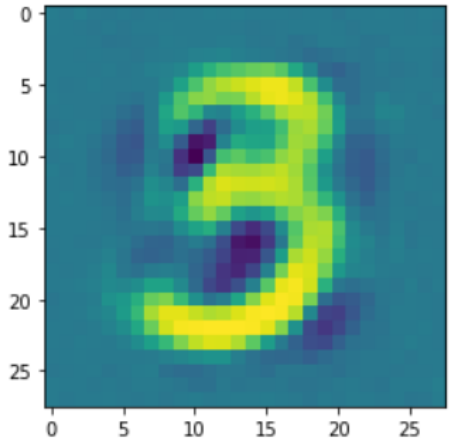
- Let us check if our generator G has been trained well enough to generate **plausible (!) images**!

- Overall, plausible and better looking images!

- As before, could remove noise and improve the quality of images, with TransposeConv2d instead of FC!

```
# Generate a fake sample for visualization
n_samples = 1
z = torch.randn(n_samples, latent_size).cuda()
z = Variable(z)
fake_images = G(z)
fake_images = fake_images.cpu().detach().numpy().reshape(n_samples, 28, 28)
print(fake_images.shape)
# Display
plt.figure()
plt.imshow(fake_images[0])
plt.show()
```

```
(1, 28, 28)
```



Limits of vanilla/Wasserstein GANs

Issue #1: GANs are effective at generating images, but

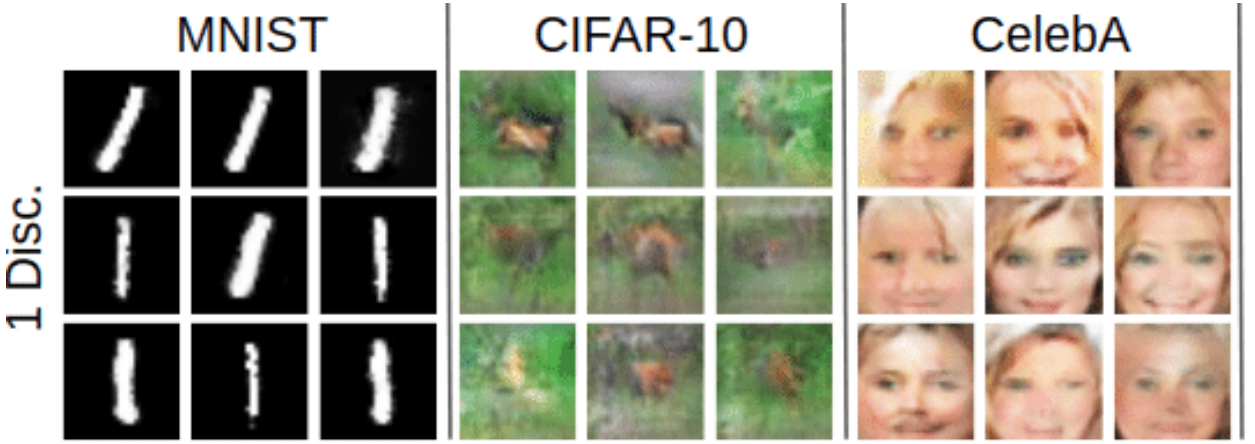
- they are limited to small images sizes,
- because of model stability,
- and training required.

- **Generating high-resolution images is challenging for GANs, because the generator must learn how to produce both large structure and fine details at the same time.**

Limits of vanilla/Wasserstein GANs

Issue #2: Lack of control on the generators outputs

- Limited understanding of the impact of noise vectors used as input in generators (amount of randomness? structure of the latent space?)
- How to disentangle features/properties in images and add controls for these properties to the generator model?



Conclusion

In this lecture

- More advanced concepts on GANs
- Wasserstein distance in the GAN loss function
- Progressive GANs
- Style GANs
- Conditional GANs and CycleGANs
- Deepfakes and ethics

If time allows, some bonus discussions on advanced GANs and identified issues

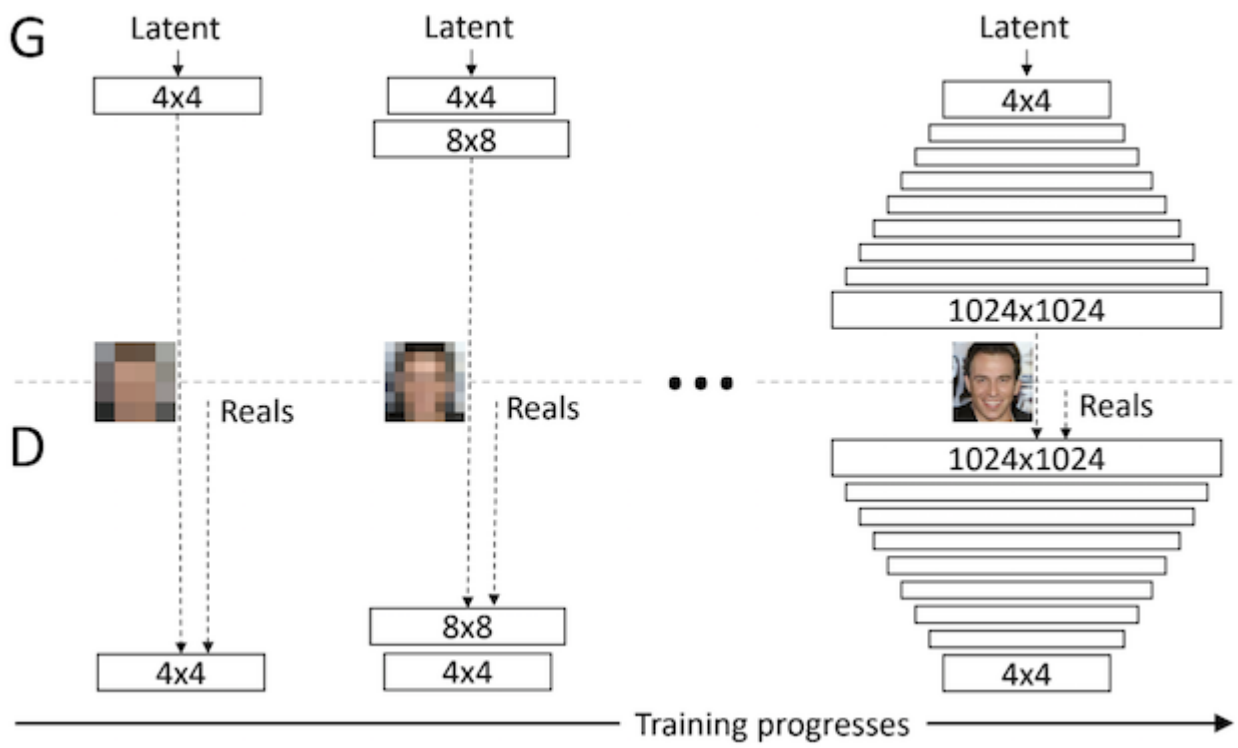
In this bonus lecture

- Limits of Vanilla/Wasserstein GANs and how to deal with them

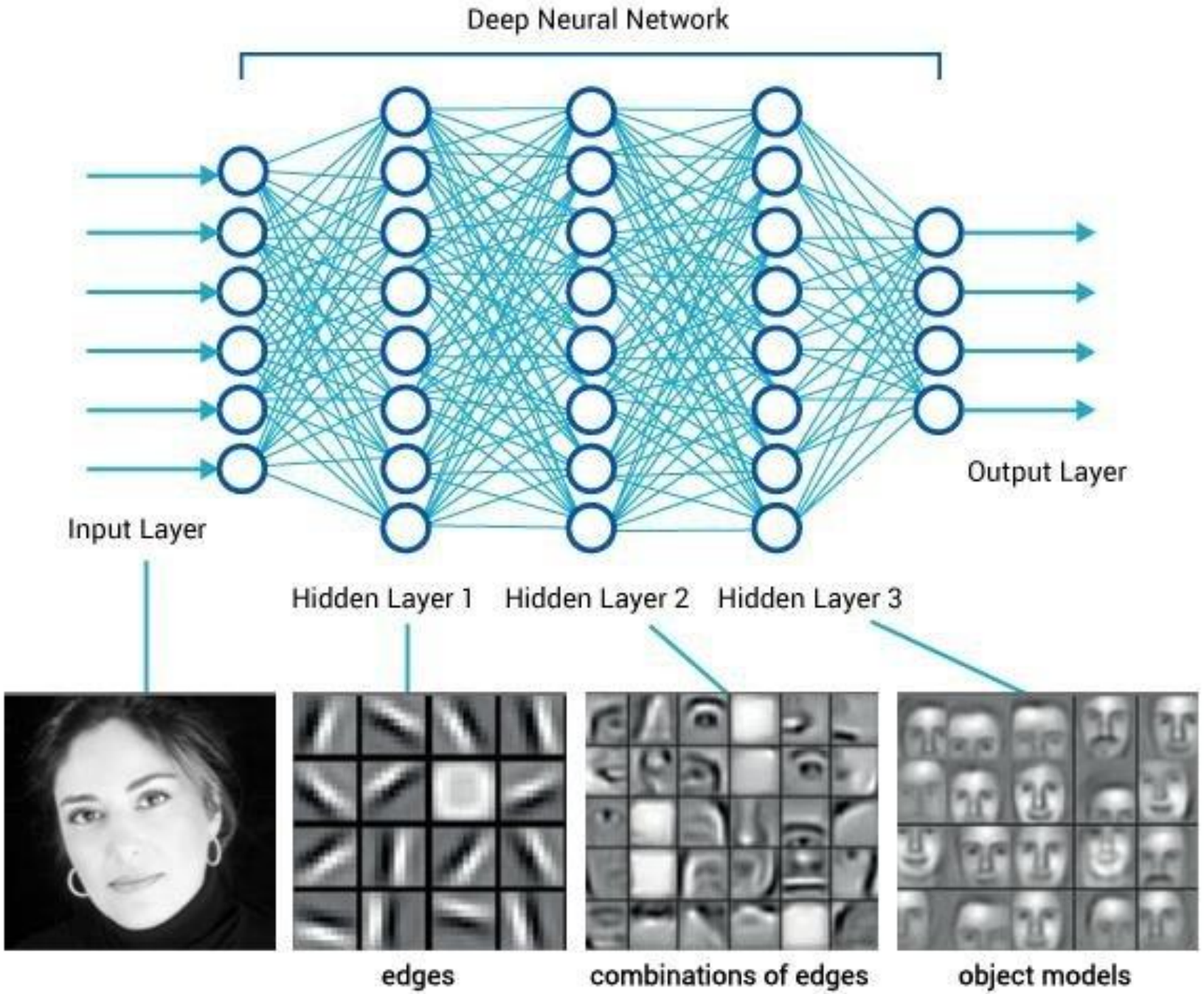
- **Issue#1 → Progressive GANs**
  - key ideas, procedure, practical implementation
- **Issue #2 → Conditional and Style GANs**
  - key ideas, procedure, practical implementation

Progressive GANs: core idea

- **Issue #1:** Generating high-resolution images is challenging for GANs, because the generator must learn how to produce both large structure and fine details at the same time.
- **Core idea:** Novel GAN approach to generate large high-quality images, by incrementally increasing the sizes of the models during training.

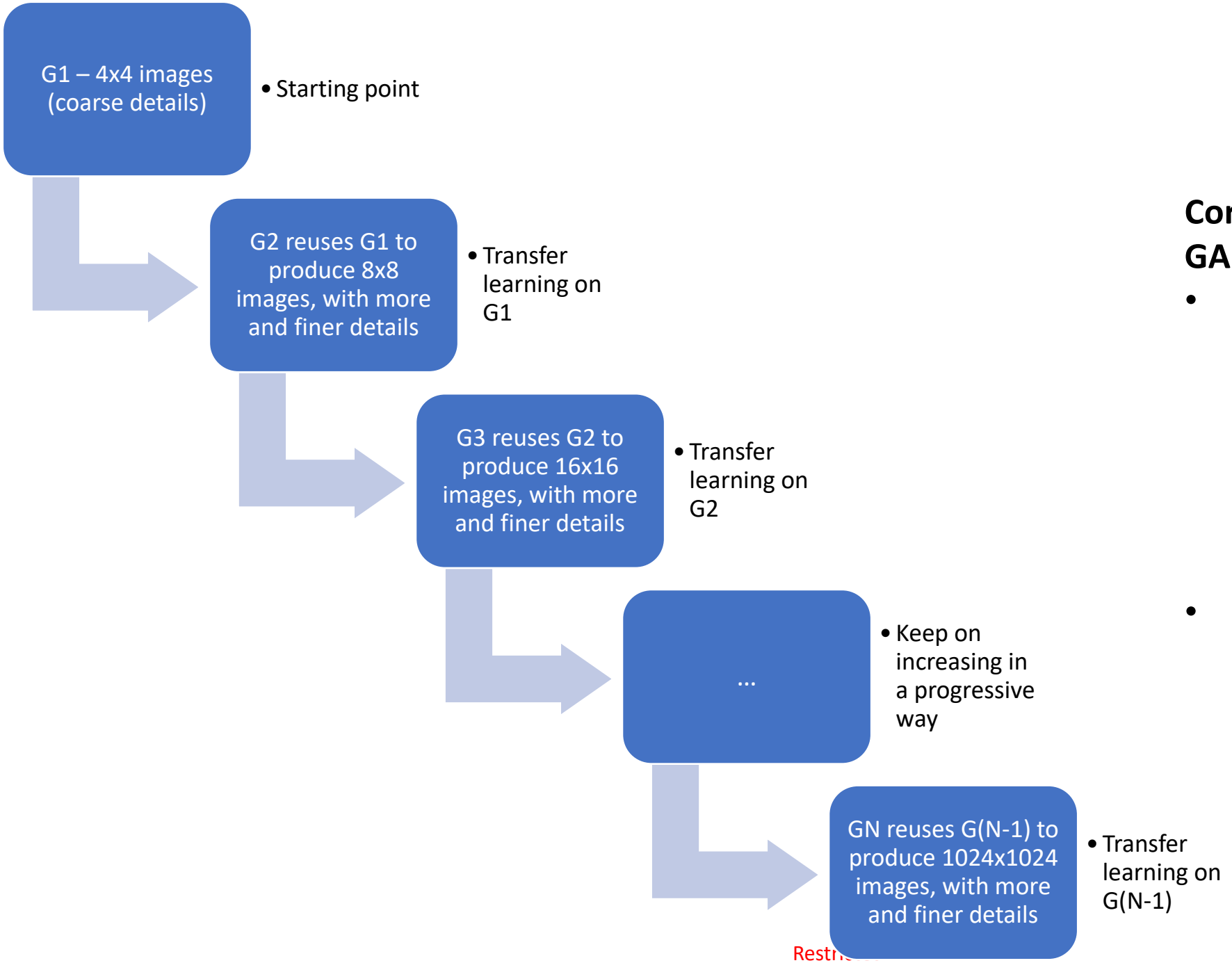


[PGGAN] Karras, T., Aila, T., Laine, S., & Lehtinen, J. (2017). Progressive growing of gans for improved quality, stability, and variation.



Note: CNN layers were progressively learning to recognize finer and finer details.

- Starting from early layers detecting basic patterns like edges, corners, colors…
- To deeper layers recognizing more advanced patterns (eyes, noses, mouths, etc.)…
- And the training progress was gradual: early layers would train on the first iterations, and progressively the training would “propagate” to deeper layers



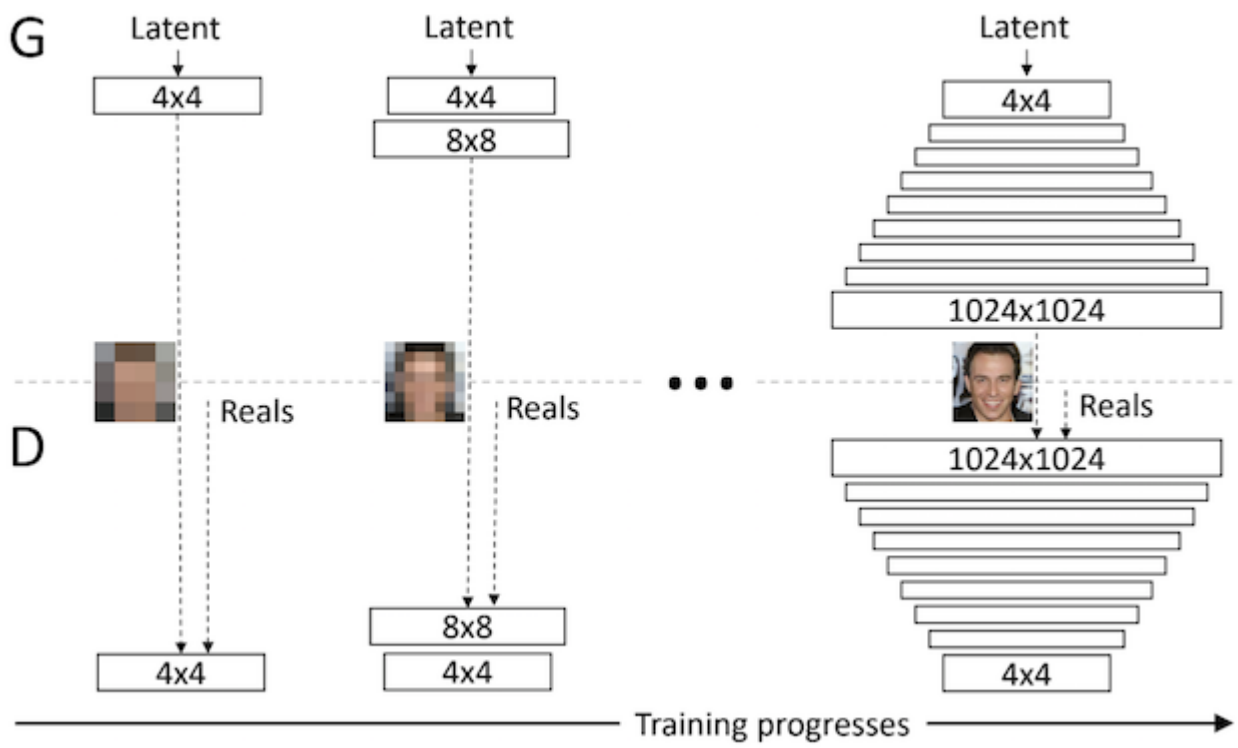
Core idea being the progressive GANs approach:

- **We will basically attempt to reproduce the same thing, by training successive "mutations"/"versions" of the generator, with increasing output sizes on images.**
- **Reusing the previous "mutations"/"versions" of the generator for the next one.**

Progressive GANs: core idea

Progressive (growing) GANs [PGGAN]:

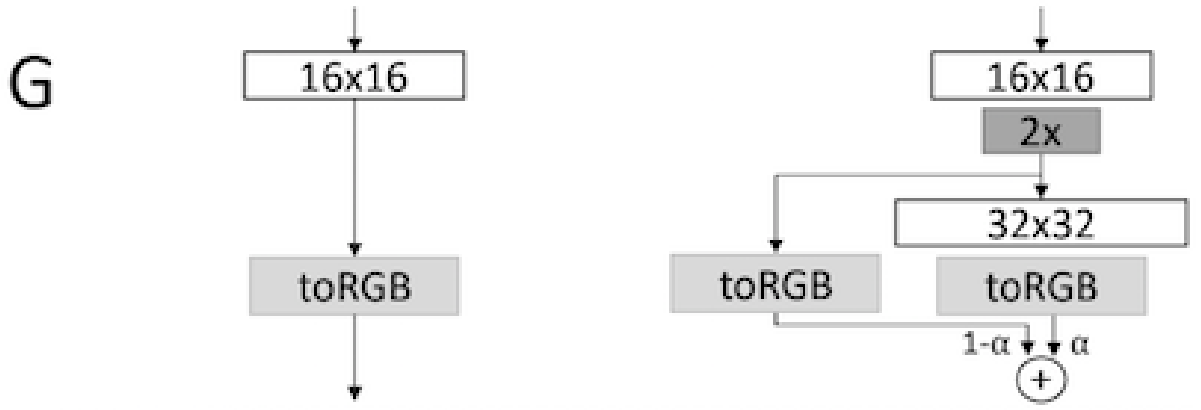
- Progressive Growing GAN involves using a generator and discriminator model with the same general structure and starting with very small images.
- During training, new blocks of convolutional layers are systematically added to both the generator model and the discriminator models.



[PGGAN] Karras, T., Aila, T., Laine, S., & Lehtinen, J. (2017). Progressive growing of gans for improved quality, stability, and variation.

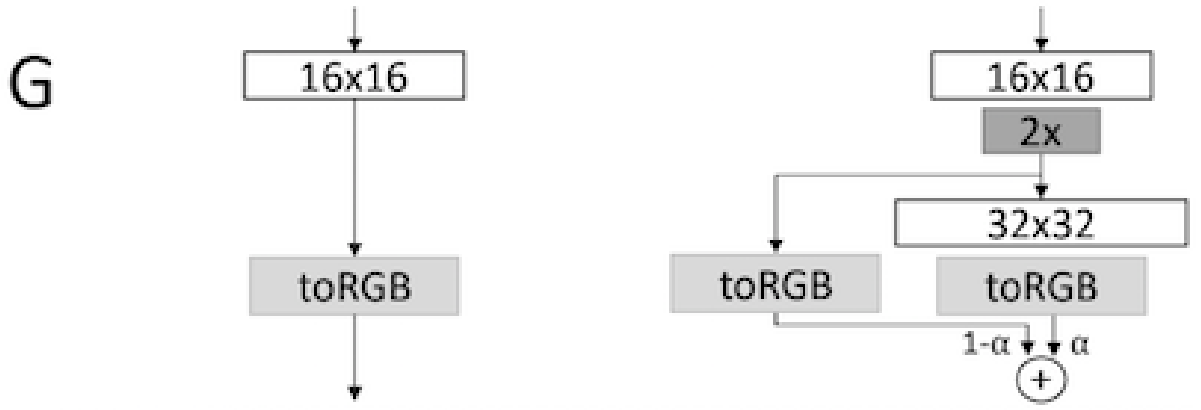
Progressive GANs: implementation details (G)

- **Idea:** add blocks of layers and perform phasing in the addition of the blocks.



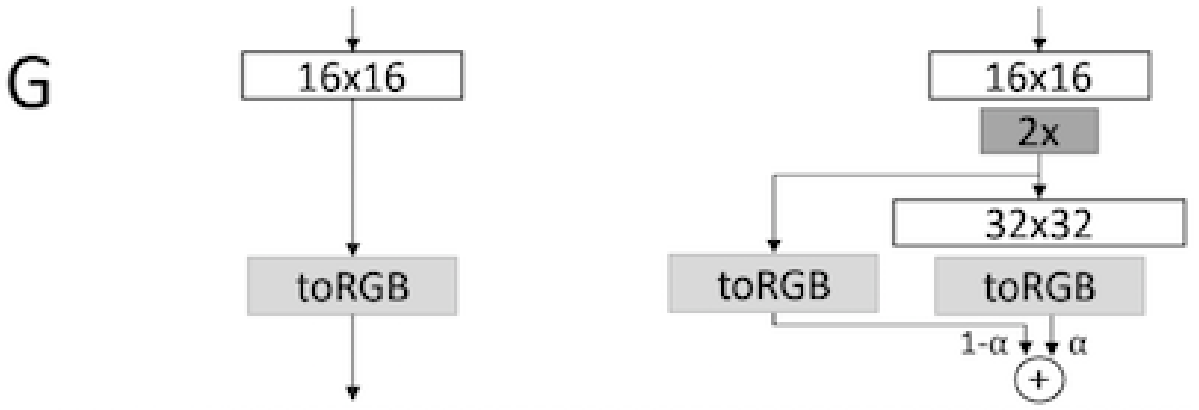
Progressive GANs: implementation details (G)

- **Idea:** add blocks of layers and perform phasing in the addition of the blocks.
- **Phasing:** use a skip connection to connect the new block to the output of the generator and add it to the existing output layer with a weighting.



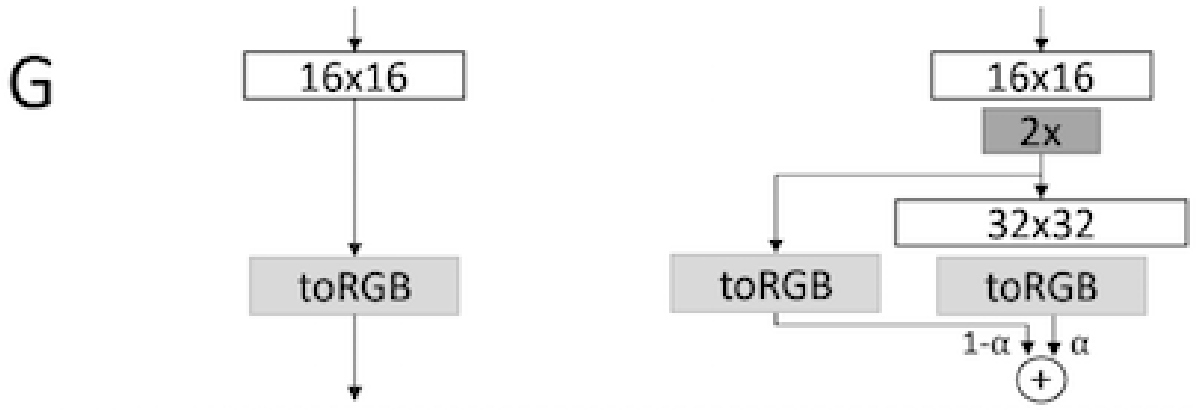
Progressive GANs: implementation details (G)

- **Idea:** add blocks of layers and perform phasing in the addition of the blocks.
- **Phasing:** use a skip connection to connect the new block to the output of the generator and add it to the existing output layer with a weighting.
- **Weighting:** contribution of the upsampled 16×16 layer is weighted by $(1 - \alpha)$, and the contribution of the new 32×32 layer is weighted by α.



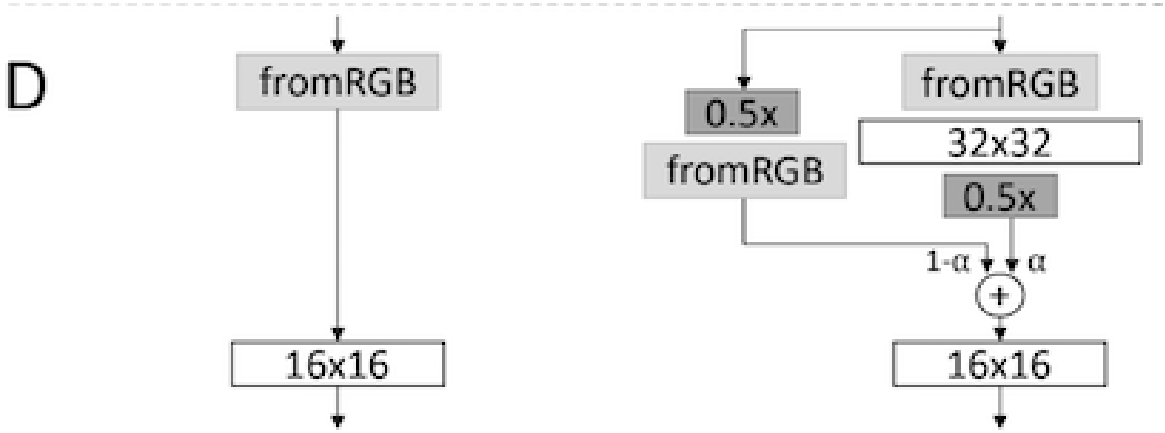
Progressive GANs: implementation details (G)

- **Idea:** add blocks of layers and perform phasing in the addition of the blocks.
- **Phasing:** use a skip connection to connect the new block to the output of the generator and add it to the existing output layer with a weighting.
- **Weighting:** contribution of the upsampled 16×16 layer is weighted by $(1 - \alpha)$, and the contribution of the new 32×32 layer is weighted by α.
- **Resolution increment:** double resolution at each step.
- **Values of α:** progressively increase from 0 to 1.
- All layers remain trainable.



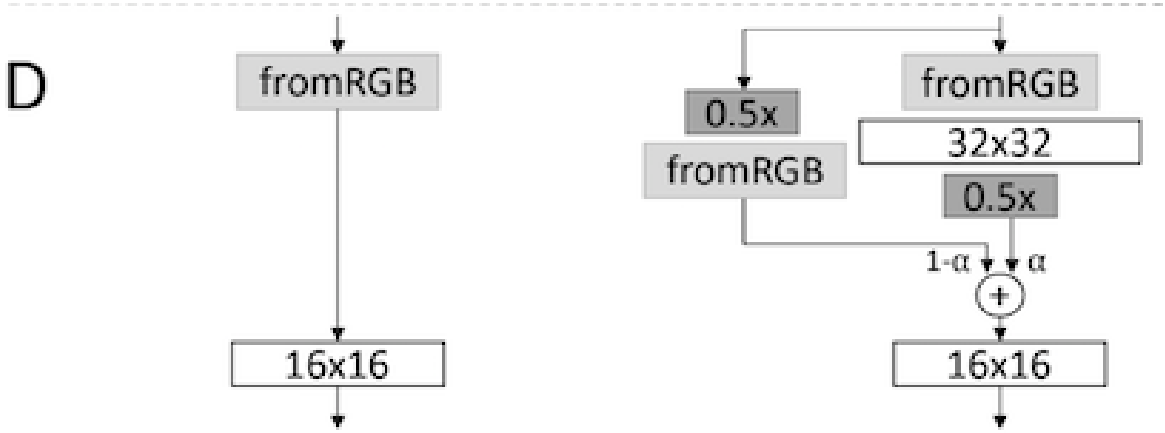
Progressive GANs: implementation details (D)

- **Idea:** same as G, but the other way around. Adding a new block of convolutional layers for the input of D to support larger images.



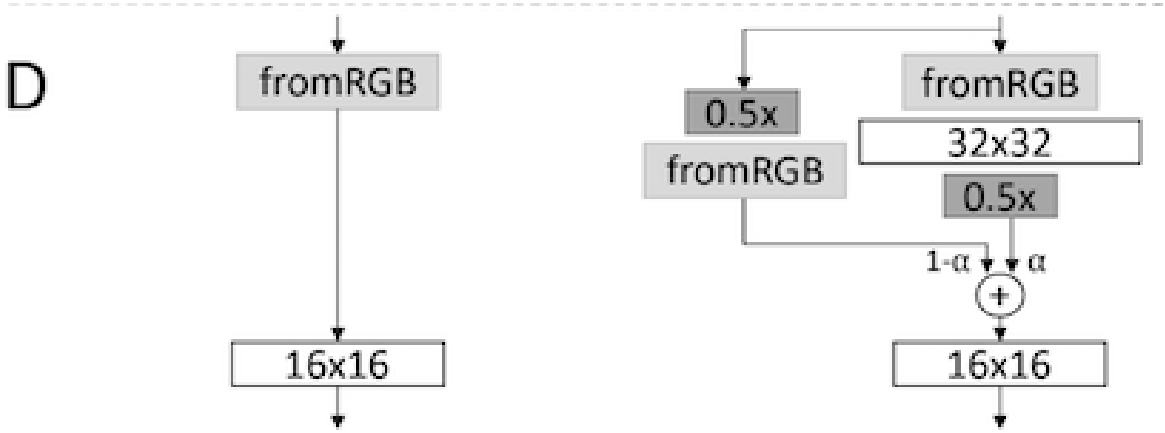
Progressive GANs: implementation details (D)

- **Idea:** same as G, but the other way around. Adding a new block of convolutional layers for the input of D to support larger images.
- **Phasing:** use a skip connection to connect the new block to the input of the discriminator and add it to the existing input layer with a weighting.



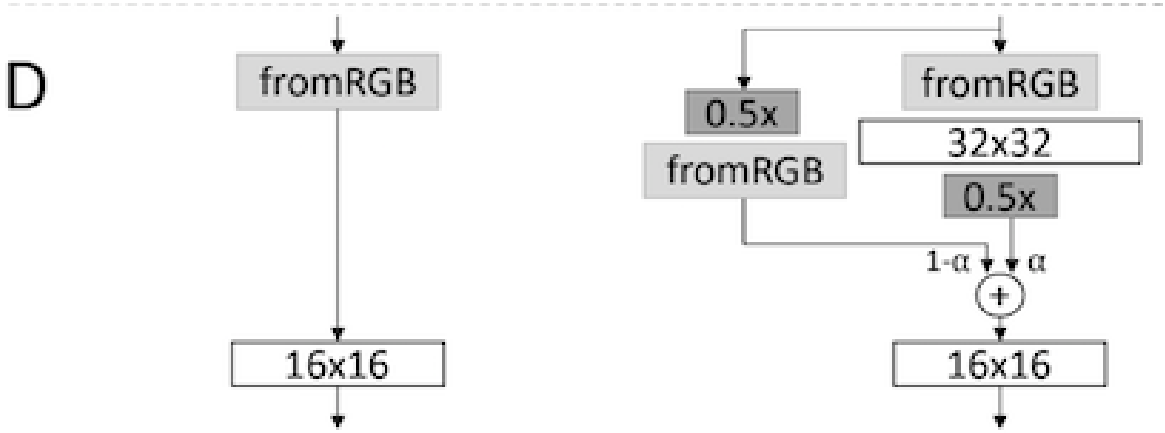
Progressive GANs: implementation details (D)

- **Idea:** same as G, but the other way around. Adding a new block of convolutional layers for the input of D to support larger images.
- **Phasing:** use a skip connection to connect the new block to the input of the discriminator and add it to the existing input layer with a weighting.
- The input image is downsampled to 16×16 using average pooling so that it can pass through the existing 16×16 convolutional layers.



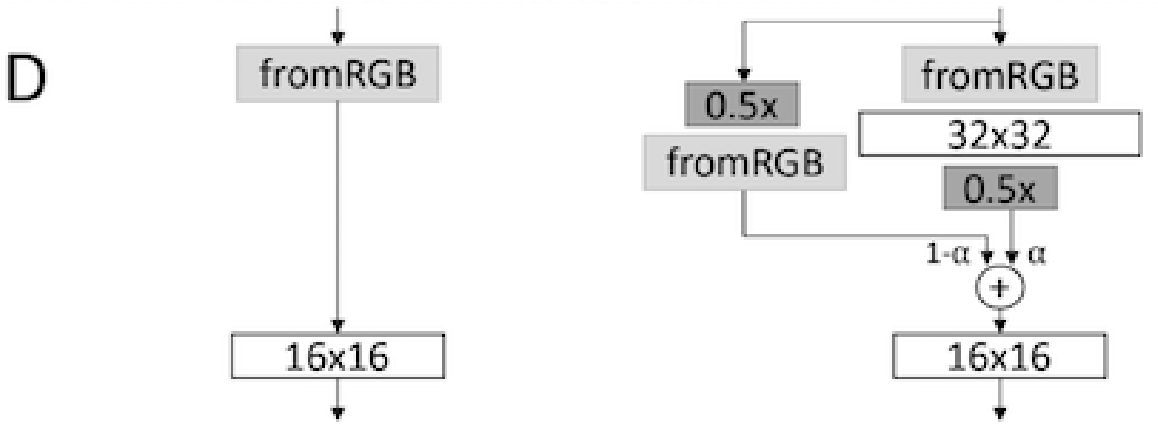
Progressive GANs: implementation details (D)

- **Idea:** same as G, but the other way around. Adding a new block of convolutional layers for the input of D to support larger images.
- **Phasing:** use a skip connection to connect the new block to the input of the discriminator and add it to the existing input layer with a weighting.
- The input image is downsampled to 16×16 using average pooling so that it can pass through the existing 16×16 convolutional layers.
- The output of the new 32×32 block of layers is also downsampled using average pooling so that it can be used as input to the existing 16×16 block.



Progressive GANs: implementation details (D)

- **Idea:** same as G, but the other way around. Adding a new block of convolutional layers for the input of D to support larger images.
- **Phasing:** use a skip connection to connect the new block to the input of the discriminator and add it to the existing input layer with a weighting.
- The input image is downsampled to 16×16 using average pooling so that it can pass through the existing 16×16 convolutional layers.
- The output of the new 32×32 block of layers is also downsampled using average pooling so that it can be used as input to the existing 16×16 block.
- Double resolution size at each step, α progressively increases from 0 to 1. All layers remain trainable.



Progressive GANs: result comparison

Progressive (growing) GANs [PGGAN]:

- **Result:** capable of generating photorealistic synthetic faces and objects at high resolution that are remarkably realistic.

Figure: DCGAN 32x32 generated pictures

[PGGAN] Karras, T., Aila, T., Laine, S., & Lehtinen, J. (2017). Progressive growing of gans for improved quality, stability, and variation.

Progressive GANs: result comparison

Progressive (growing) GANs [PGGAN]:

- **Result:** capable of generating photorealistic synthetic faces and objects at high resolution that are remarkably realistic.
- Original celebrity DCGAN model: outputs 32x32 images, flaws in details.

Figure: DCGAN 32x32 generated pictures

[PGGAN] Karras, T., Aila, T., Laine, S., & Lehtinen, J. (2017). Progressive growing of gans for improved quality, stability, and variation.

Progressive GANs: result comparison

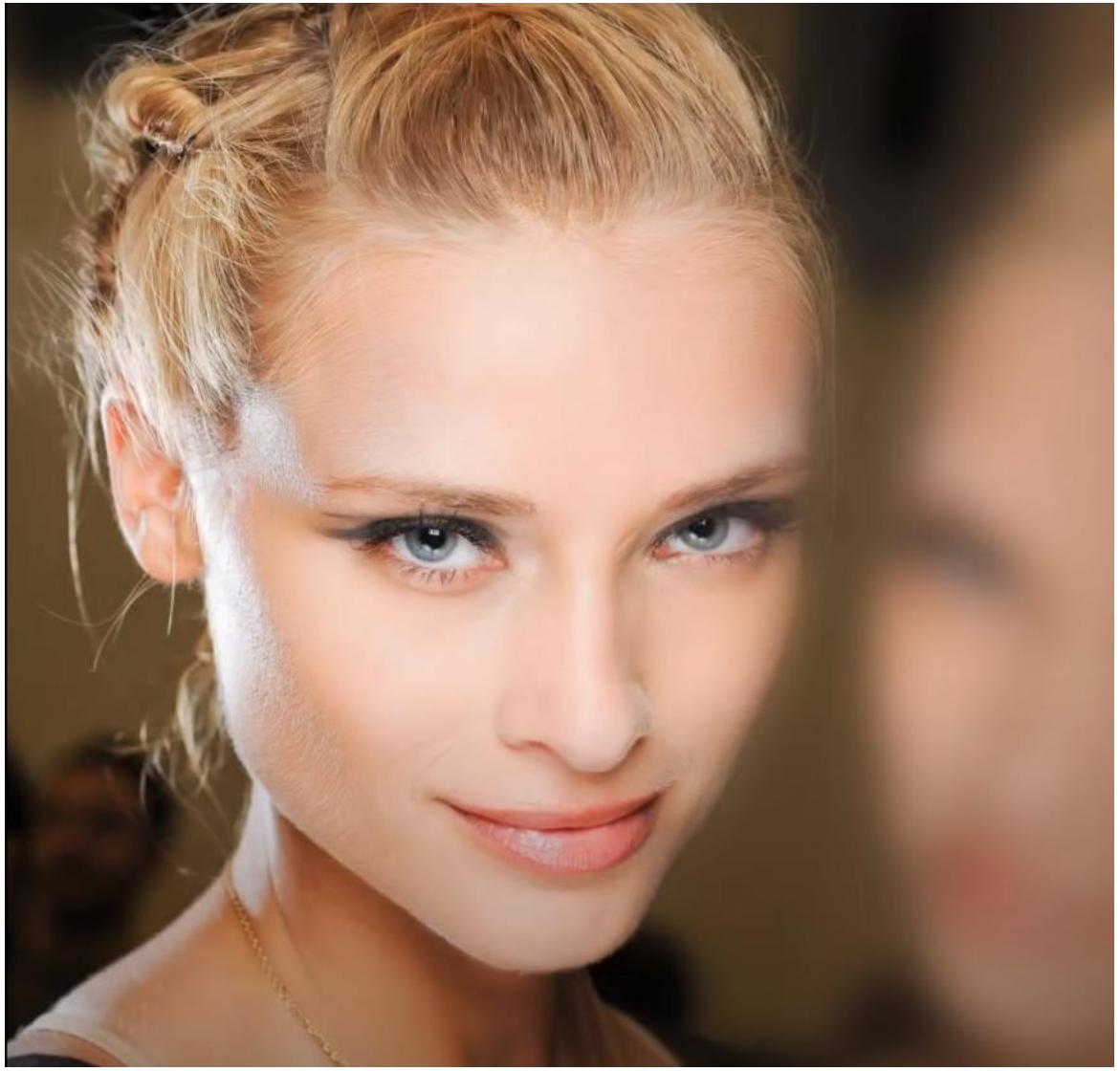
Progressive (growing) GANs [PGGAN]:

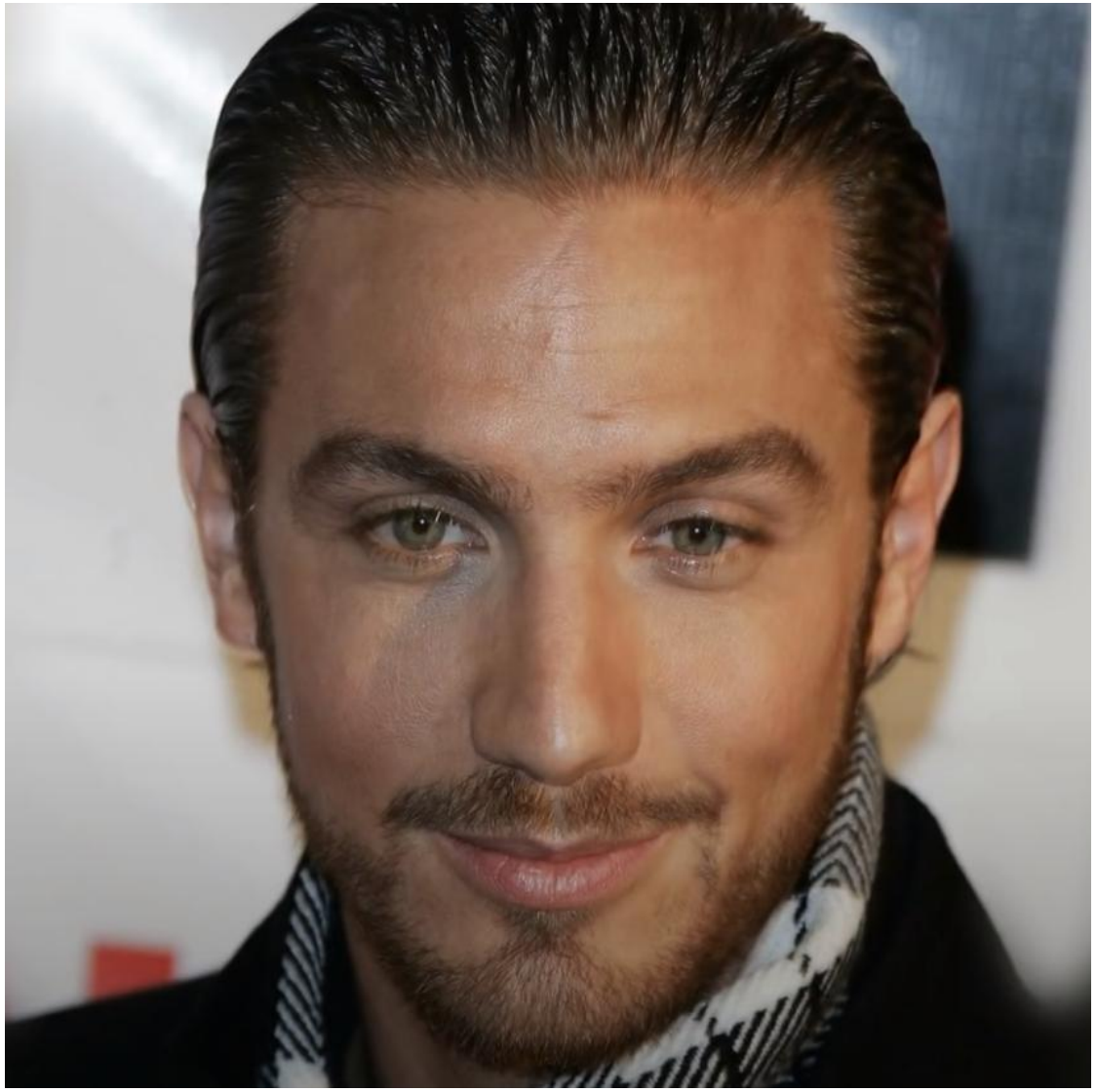
- **Result:** capable of generating photorealistic synthetic faces and objects at high resolution that are remarkably realistic.
- Original celebrity DCGAN model: outputs 32x32 images, flaws in details.
- Progressive celebrity HQ GANs: outputs 1024x1024 images…

Figure: DCGAN 32x32 generated pictures

[PGGAN] Karras, T., Aila, T., Laine, S., & Lehtinen, J. (2017). Progressive growing of gans for improved quality, stability, and variation.

And they are quite realistic (!)





Progressive GANs: useful links

- ***[PGGAN]*** Karras, T., Aila, T., Laine, S., & Lehtinen, J. (2017). Progressive growing of gans for improved quality, stability, and variation.
- Paper ***[PGGAN]***: https://arxiv.org/pdf/1710.10196.pdf
- Demo code and pre-trained models:
  https://pytorch.org/hub/facebookresearch_pytorch-gan-zoo_pgan/
  https://github.com/facebookresearch/pytorch_GAN_zoo
- Learn more (Nvidia/Youtube video with PGGANs paper results):
  https://www.youtube.com/watch?time_continue=4&v=G06dEcZ-QTg&feature=emb_logo

Progressive GANs: a note on training time

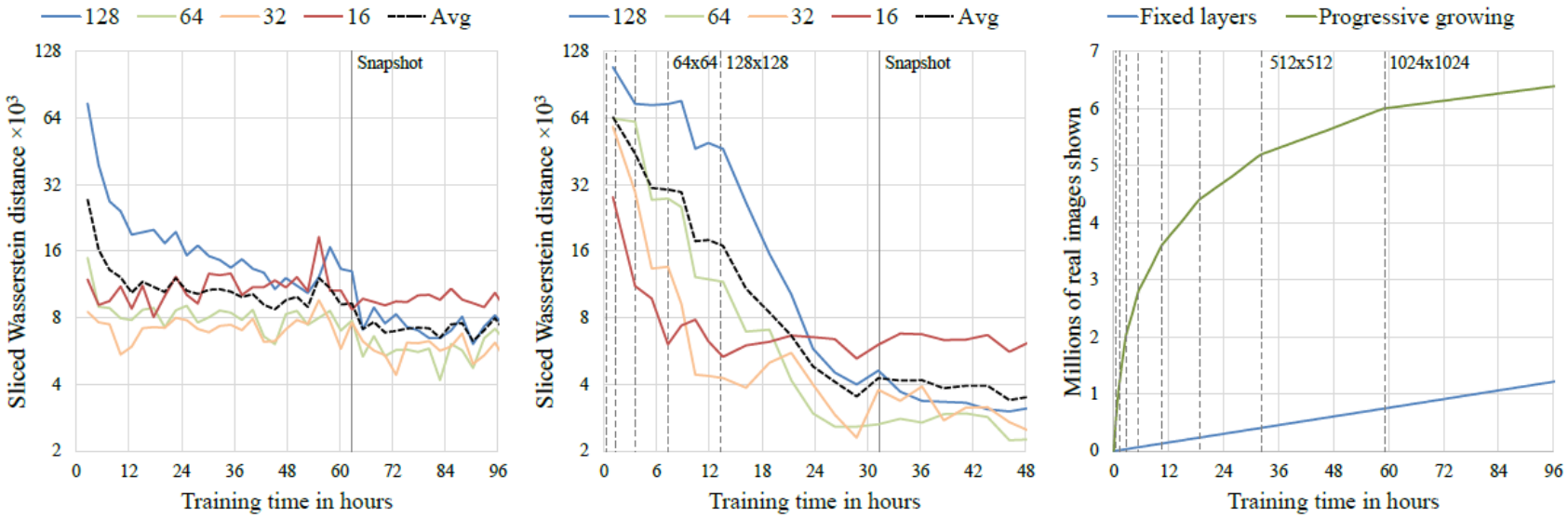


Figure 4: Effect of progressive growing on training speed and convergence. The timings were measured on a single-GPU setup using NVIDIA Tesla P100. (a) Statistical similarity with respect to wall clock time for Gulrajani et al. (2017) using CELEBA at 128 × 128 resolution. Each graph represents sliced Wasserstein distance on one level of the Laplacian pyramid, and the vertical line indicates the point where we stop the training in Table 1. (b) Same graph with progressive growing enabled. The dashed vertical lines indicate points where we double the resolution of G and D. (c) Effect of progressive growing on the raw training speed in 1024 × 1024 resolution.

Source: [PGGAN] Karras, T., Aila, T., Laine, S., & Lehtinen, J. (2017). Progressive growing of gans for improved quality, stability, and variation.

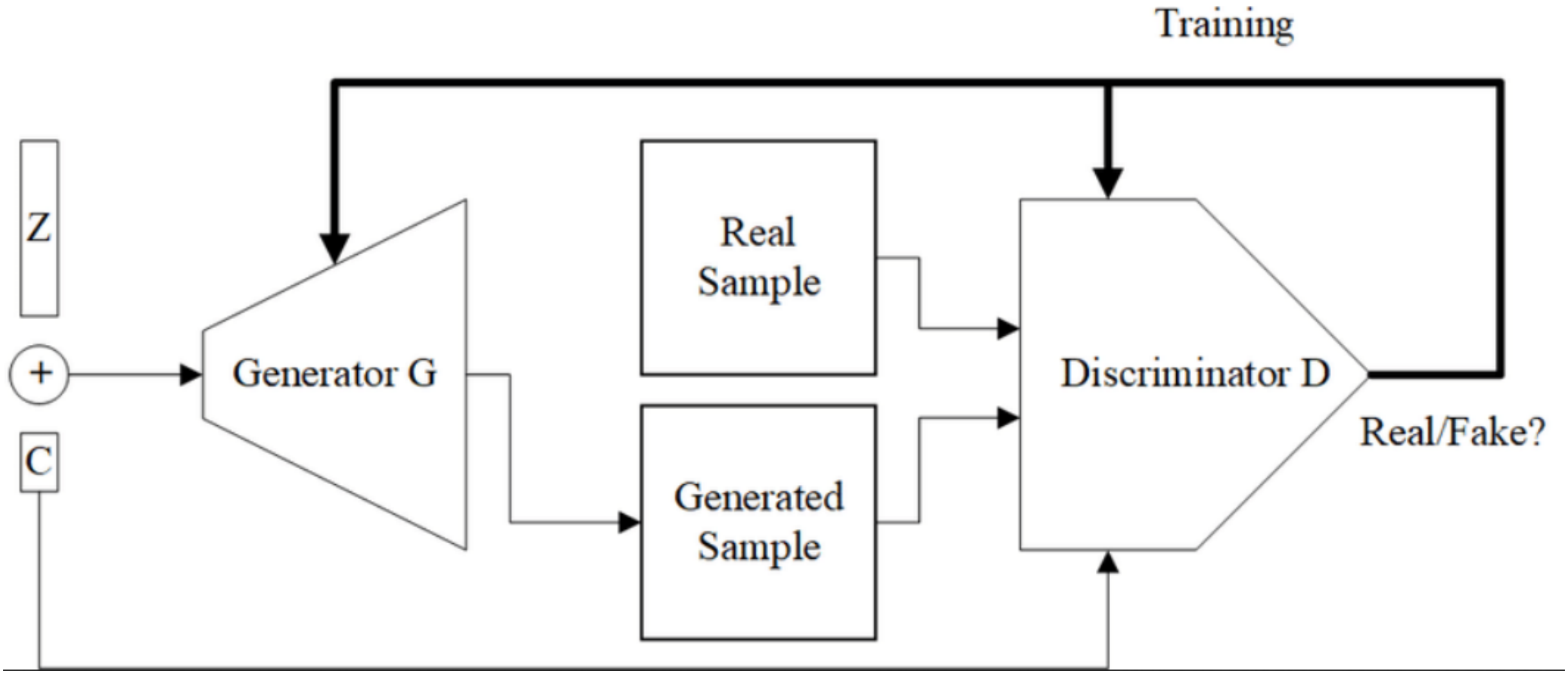
Conditional GANs

- **Issue #2:** Lack of control on the generators outputs.

Core idea of Conditional GANs [CondGAN]**:** Add some deterministic values to the noise samples used in the generator.

- This deterministic value corresponds to the class you expect the sample to have (e.g. generate a handwritten 7).
- Generator task is to fool the discriminator/critic by generating a sample, which will be classified as a “real 7” and not anything else (“fake 7”, “real 6” are both considered failure cases).

Conditional GANs



CycleGAN

- **Core idea of CycleGANs** [CycleGAN]: do not use noise as input for generator, use another image that you wish to transform!
- Train an autoencoder style of architecture, with two generators.
- One generator G_1 that transforms images of zebras into horses, and one generator G_2 that transforms images of horses into zebras.
- Train both generators to accomplish $G_1(G_2(\text{x})) = \text{x}$.
- By doing so, you get a generator, which transforms images of a certain type into images of another type!

GANs and ethics

https://www.youtube.com/watch?v=bE1KWpoX9Hk

GANs and ethics

https://www.youtube.com/watch?v=bE1KWpoX9Hk

App that can remove women's clothes from images shut down

28 June 2019

https://www.bbc.com/news/technology-48799045

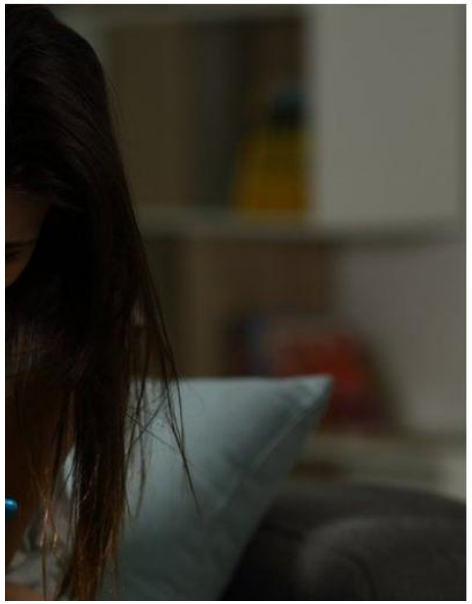
GANs a

Jordan Peele uses AI, President Obama in

A Zelensky Deepfake Was Quickly Defeated. The Next One Might Not Be

PHOTOGRAPH: EMIN SANSAR/GETTY IMAGES

women's
ıut down



The AI Database →

APPLICATION: DEEPFAKES, SAFETY SECTOR: PUBLIC SAFETY, SOCIAL MEDIA

SOURCE DATA: VIDEO

ON MARCH 2, the Ukraine government's Center for Strategic Communication warned that its enemies might be preparing a "deepfake" video that appeared to show president Volodymyr Zelensky announcing his surrender to Russia's invasion. On Wednesday, that warning appeared prescient.

GANs and ethics

Artificial intelligence / Machine learning

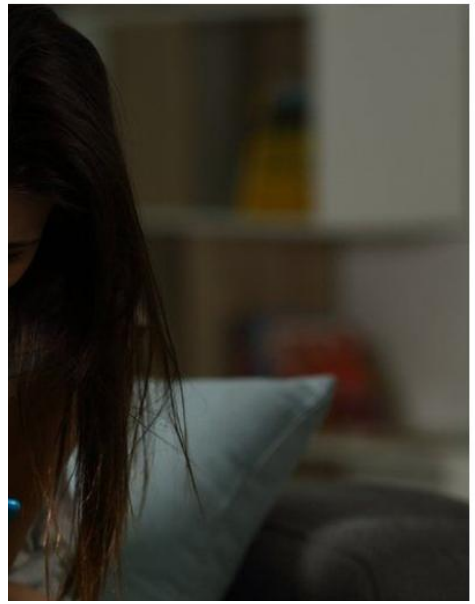
The GANfather: The man who's given machines the gift of imagination

By pitting neural networks against one another, Ian Goodfellow has created a powerful AI tool. Now he, and the rest of us, must face the consequences.

by **Martin Giles** February 21, 2018

https://www.technologyreview.com/2018/02/21/145289/the-ganfather-the-man-whos-given-machines-the-gift-of-imagination/

App that can remove women's
ıut down

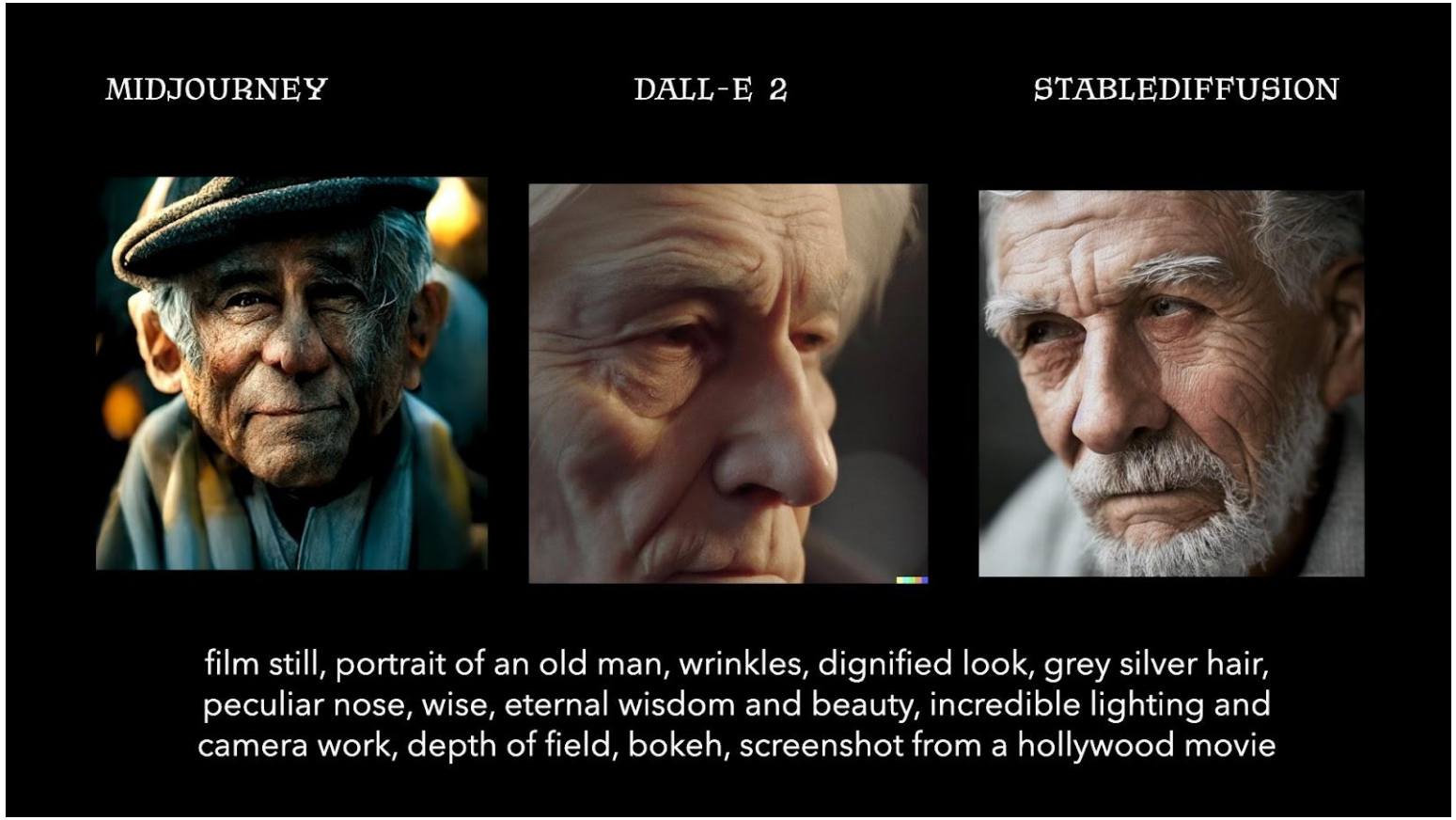


What about diffusion models?!

You might have heard **diffusion models**…

- E.g. Dall-E2, StableDiffusion, MidJourney, etc.

Turns out: special type of generative models, to be discussed on a special lecture during Week 12 or 13!



Learn more about these topics

Out of class, for those of you who are curious

- [AutoEnc] **Hinton** and **Salakhutdinov**, “Reducing the Dimensionality of Data with Neural Networks”, 2006. https://paperswithcode.com/paper/reducing-the-dimensionality-of-data-with
- [VarAutoEnc] **Kingma** and **Welling**, “Auto-Encoding Variational Bayes”, 2014. https://arxiv.org/abs/1312.6114
- [GAN] **Goodfellow** et al., “Generative Adversarial Networks”, 2014. https://arxiv.org/abs/1406.2661

Learn more about these topics

Out of class, for those of you who are curious

- [WGAN] Arjowski et al., ”Towards Principled Methods for Training Generative Adversarial Networks”, 2017. https://arxiv.org/abs/1701.04862
- [PGGAN] **Karras**, et al., “Progressive growing of gans for improved quality, stability, and variation”, 2017. https://arxiv.org/pdf/1710.10196.pdf
- [StyleGAN] **Karras**, T., Laine, S., & Aila, T. (2019). A style-based generator architecture for generative adversarial networks. https://arxiv.org/pdf/1812.04948.pdf

Learn more about these topics

Out of class, for those of you who are curious

- [Villani2008] Villani, “Optimal transport, old and new”, 2008. https://cedricvillani.org/sites/dev/files/old_images/2012/08/preprint-1.pdf
- [CondGAN] **Mirza** et al. “Conditional Generative Adversarial Nets”, 2014. https://arxiv.org/abs/1411.1784
- [CycleGAN] Zhu et al., “Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks”, 2017. https://arxiv.org/abs/1703.10593

Learn more about these topics

Tracking important names (Track their works and follow them on Scholar, Twitter, or whatever works for you!)

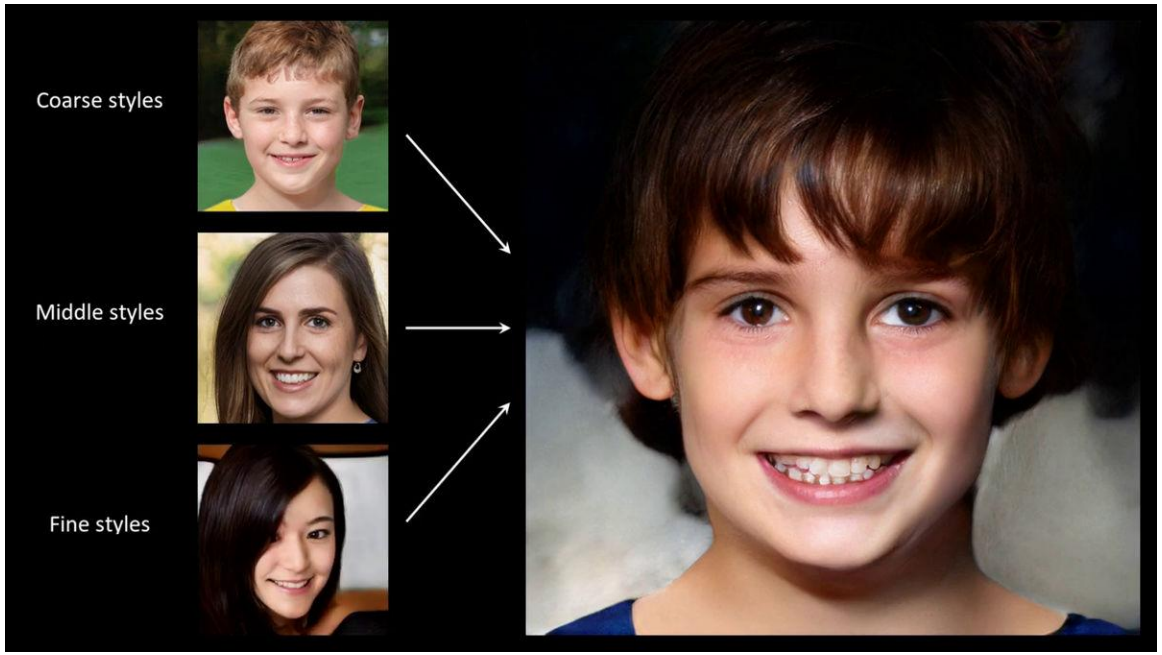
- **Tero Karras**: **Researcher** at **NVIDIA**.
  https://scholar.google.fi/citations?user=-50qJW8AAAAJ&hl=en
- **Mehdi Mirza: Researcher** at **DeepMind.**
  Co-inventor of **GANs** with **Goodfellow.**
  https://scholar.google.com/citations?user=c646VbAAAAAJ&hl=fr

Learn more about these topics

Out of class, for those of you who are curious

- Jordan Peele impersonates Obama using GANs https://www.youtube.com/watch?v=bE1KWpoX9Hk
- Shutting down an AI that could remove clothes from people https://www.bbc.com/news/technology-48799045
- The man who's given machines the gift of imagination https://www.technologyreview.com/2018/02/21/145289/the-ganfather-the-man-whos-given-machines-the-gift-of-imagination/
- Create fake landscape pictures, post these on Instagram, pretend these were taken in Singapore and let people wonder where it is. http://nvidia-research-mingyuliu.com/gaugan/

Style GANs: core idea



- **Issue #2:** Lack of control on the generators outputs.

Style GANs *[StyleGAN]*: PGGAN++

- Core idea: Reuse PGGAN, but change the architecture of generator to introduce control over the style of generated images at different levels of detail, from high-level features such as background and foreground, to fine-grained details of objects or subjects.

[StyleGAN] Karras, T., Laine, S., & Aila, T. (2019). A style-based generator architecture for generative adversarial networks.

Style GAN: implementation details

- **Reuse PGGAN:** Incremental growing of the models, but also change the architecture of the generator significantly.
- **Input change:** no longer take a noise vector as input.
  - instead, there are two new sources of randomness used to generate a synthetic image,
  - a standalone mapping network
  - and noise introduced at each layer of the generator.

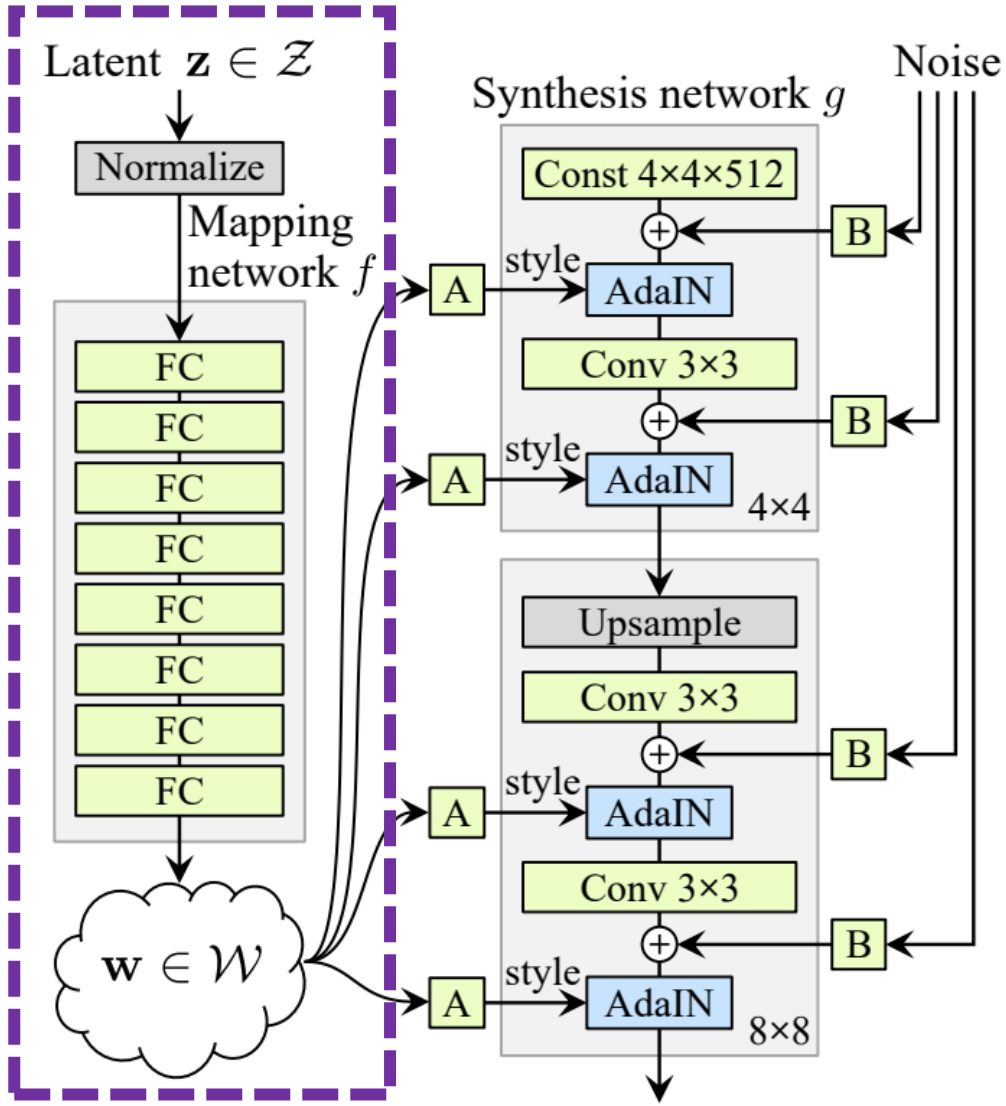
Style GAN: implementation details

- **Reuse PGGAN:** Incremental growing of the models, but also change the architecture of the generator significantly.
- **Input change:** no longer take a noise vector as input.
  - instead, there are two new sources of randomness used to generate a synthetic image,
  - a standalone mapping network
  - and noise introduced at each layer of the generator.
- **Mapping network:** output from the mapping network is a vector defining the styles, later integrated at each point in the generator model. This style vector gives control over the style of the generated image.
- **Noise introduction:** per-block incorporation of style vector and noise allows each block to localize both the interpretation of style and induce random variation to a given level of detail.

Style GAN: implementation details

1. Mapping network: a standalone mapping network.

- consisting of simple Dense layers NN,
- takes a noise vector z as input
- and generates a style vector w.

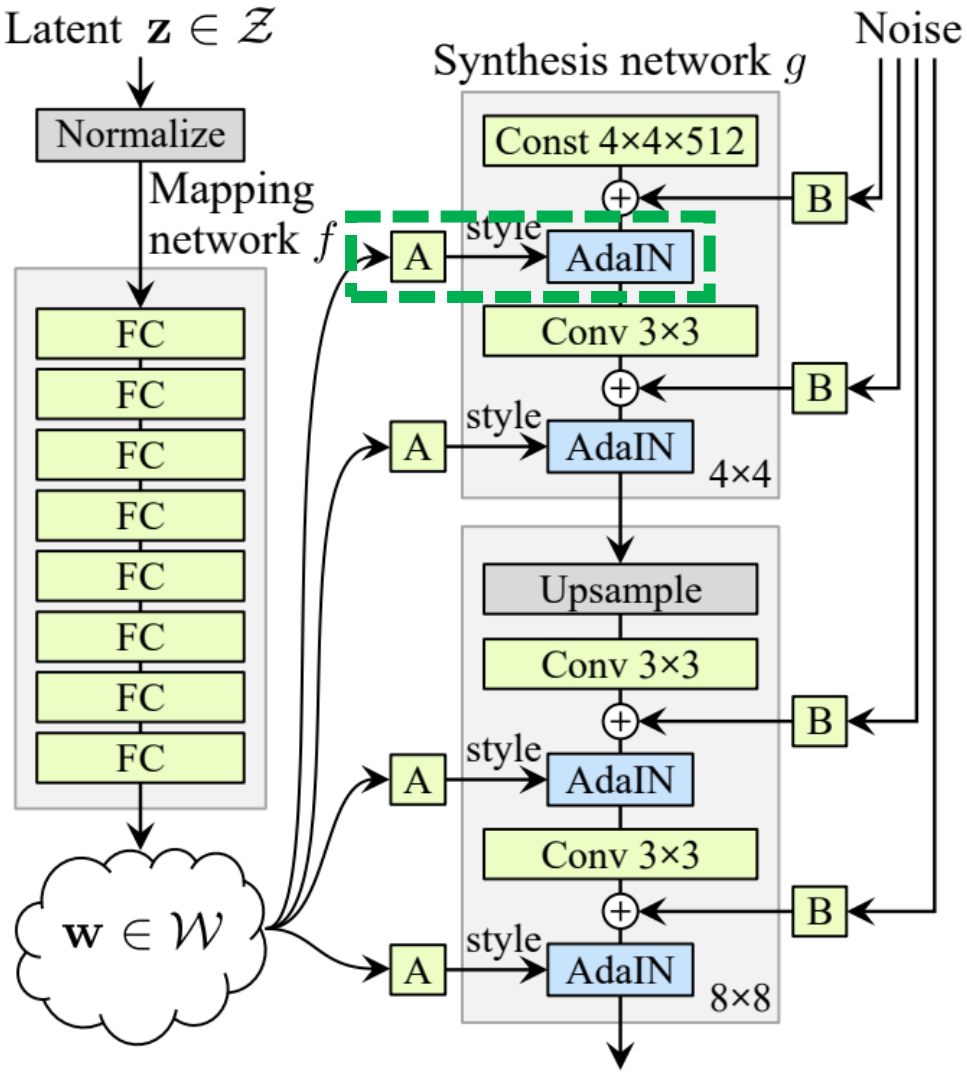


Style GAN: implementation details

2. AdaIN: style vector is incorporated into each block of the generator model after the conv. layers, via adaptive instance normalization (AdaIN).

- standardize the output of feature map to a standard Gaussian,
- in a very similar way as batchnorm.

$$\mathrm{AdaIN}(x, y) = \sigma(y)\left(\frac{x - \mu(x)}{\sigma(x)}\right) + \mu(y)$$

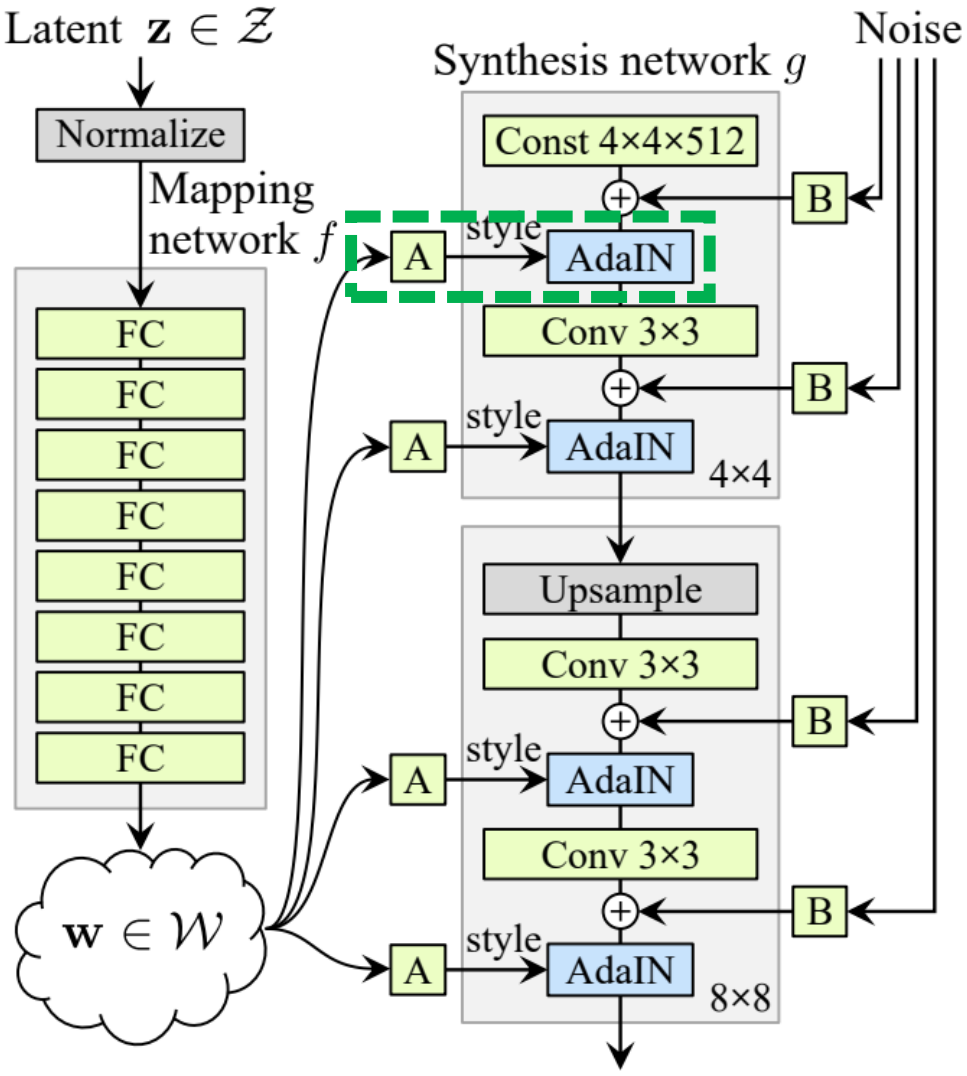


Style GAN: implementation details

2. AdaIN: breaking down the layer

- Given a content input x, with $i \in [1, N]$ channels, and a style input y,
- AdaIN aligns the channel-wise mean and variance of x, to match those of y.
- AdaIN has no learnable affine parameters, it adaptively computes the affine parameters from the style input.

$$\mathrm{AdaIN}(x, y) = \sigma(y)\left(\frac{x - \mu(x)}{\sigma(x)}\right) + \mu(y)$$

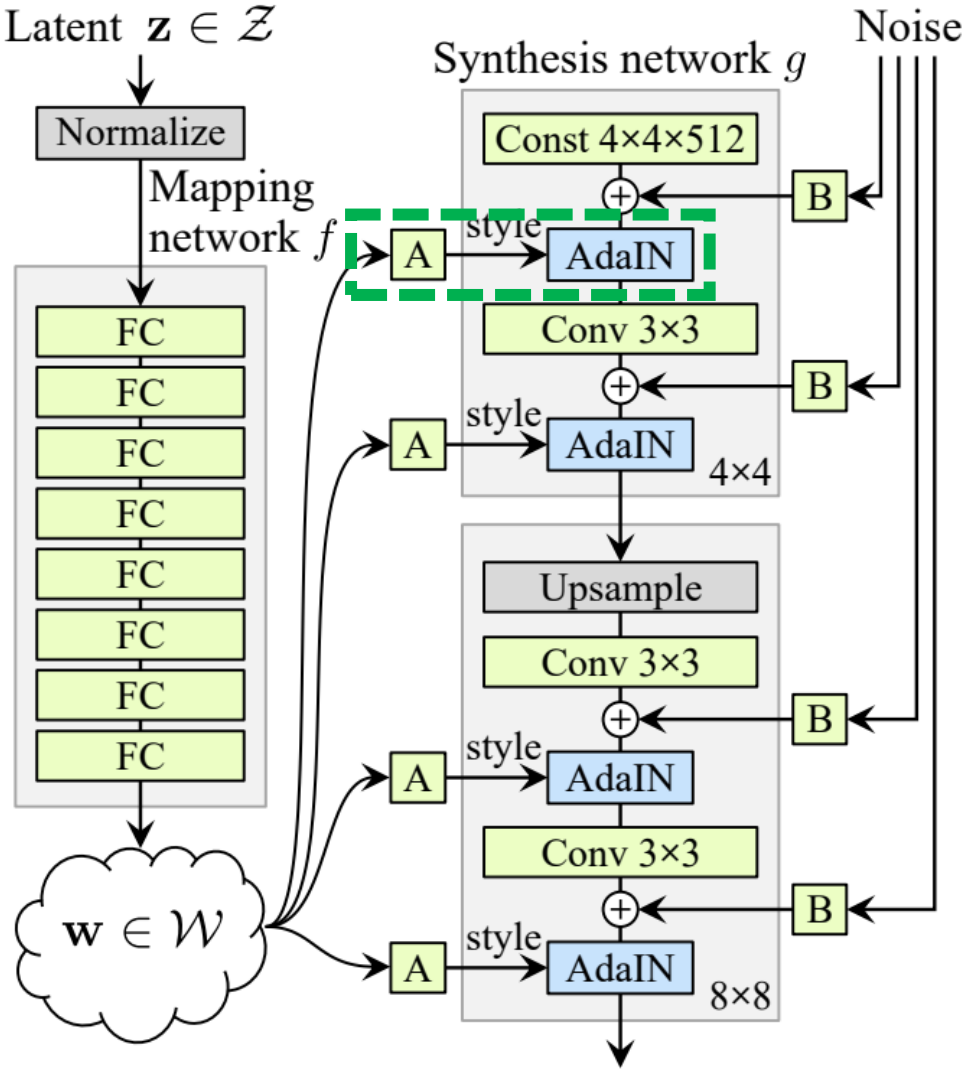


Style GAN: implementation details

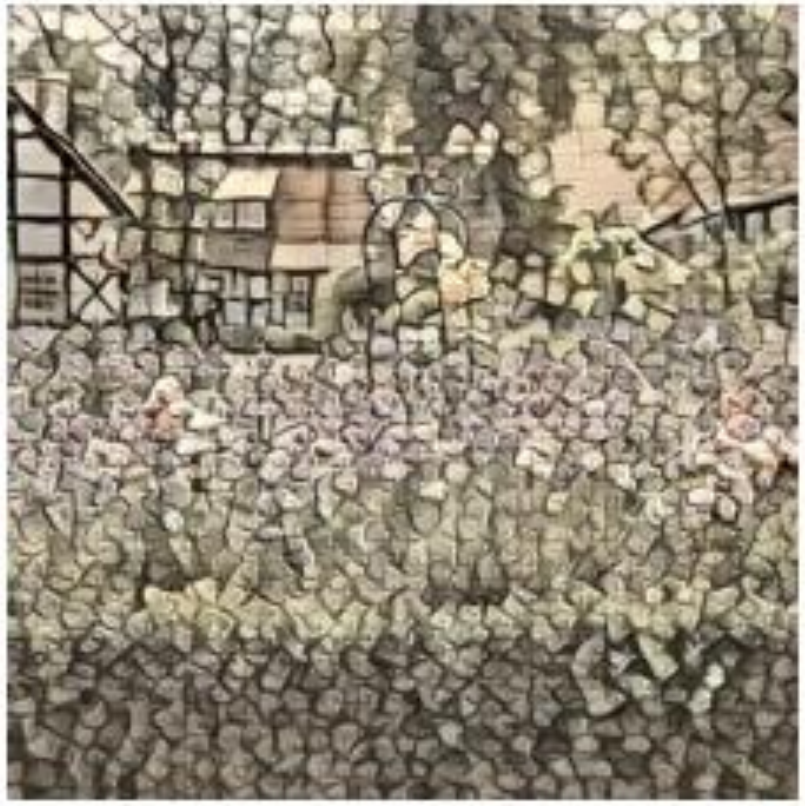
2. AdaIN: breaking down the layer

- As in BatchNorm, we simply scale the normalized content input x with $\sigma(y)$, and shift it with $\mu(y)$.
- These statistics are computed across all spatial locations.
- AdaIN transfers feature statistics, specifically the channel-wise mean and variance (= style transfer).

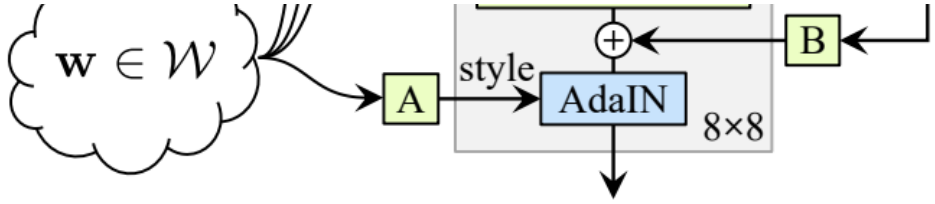
$$\mathrm{AdaIN}(x, y) = \sigma(y)\left(\frac{x - \mu(x)}{\sigma(x)}\right) + \mu(y)$$



Style GAN: implementation details



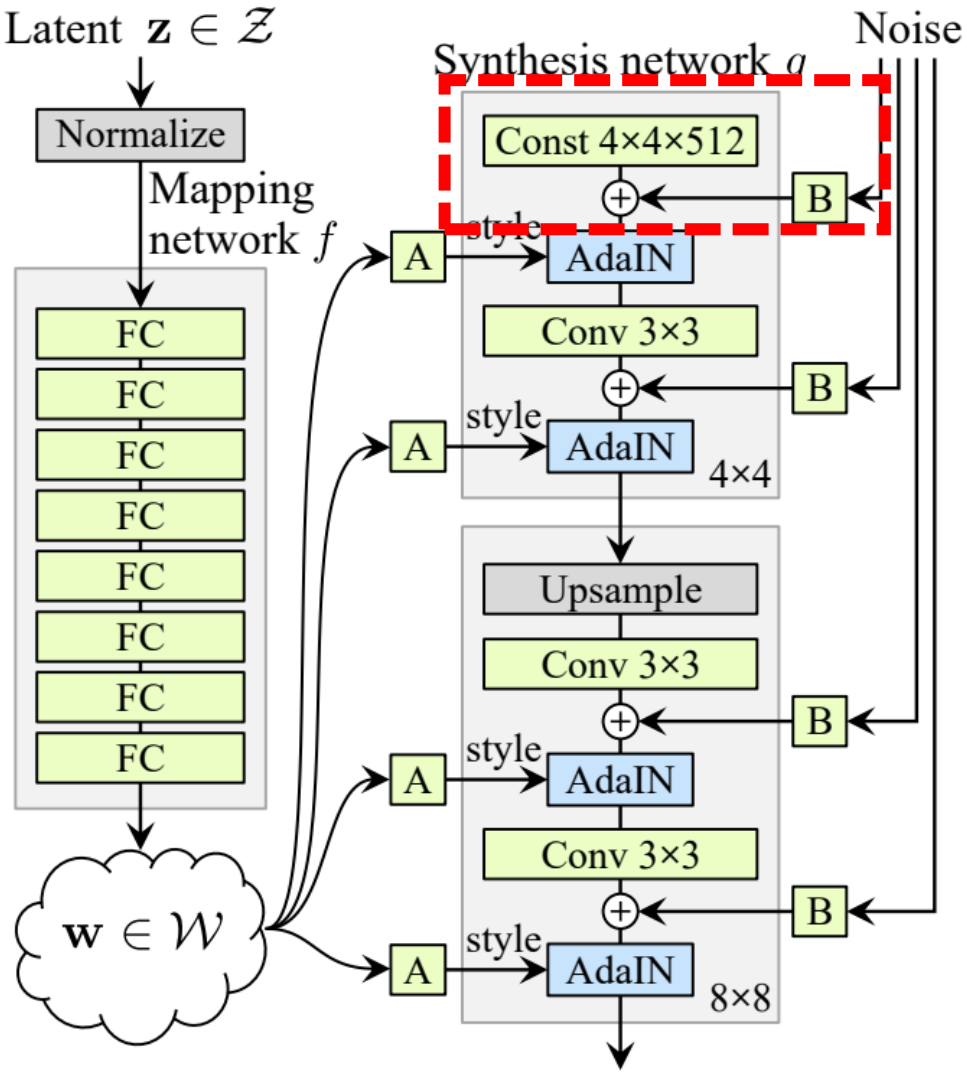
$$\text{AdaIN}(x, y) = \sigma(y)\left(\frac{x - \mu(x)}{\sigma(x)}\right) + \mu(y)$$



Style GAN: implementation details

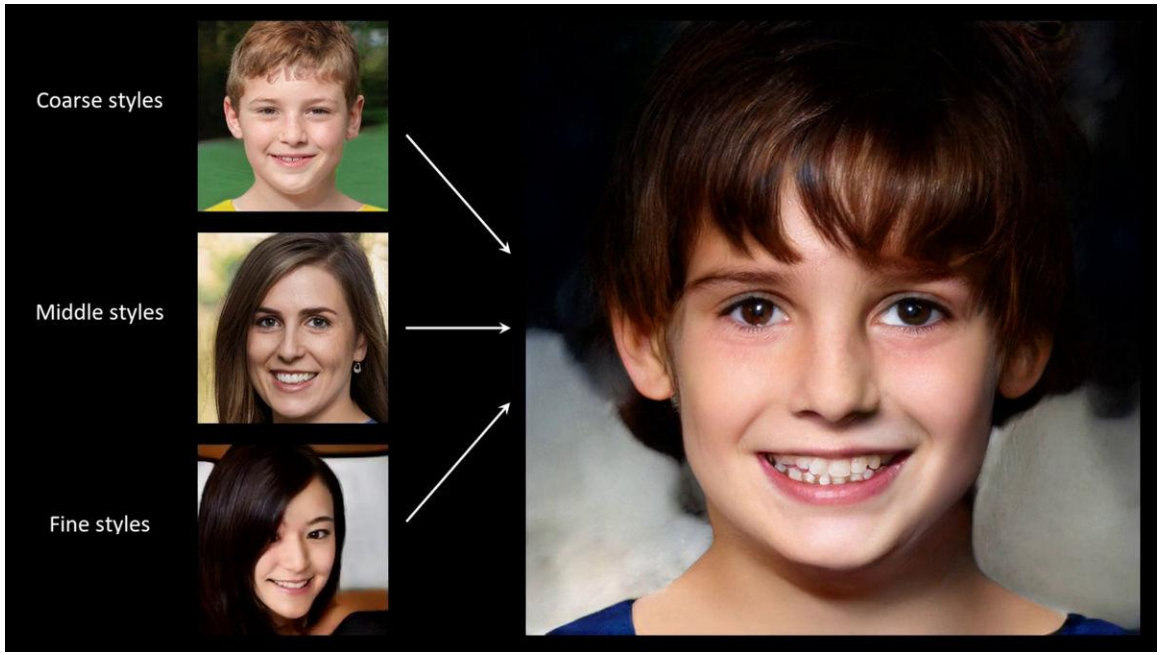
3. Local noise addition: Gaussian noise is added to each of the conv. prior to the AdaIN operations.

- More specifically, single-channel images B consisting of uncorrelated Gaussian noise.
- Feed a dedicated noise image to each layer, broadcasted to all feature maps using learned per-feature scaling factors and added to the output of each convolution
- This noise introduces style-level variation at a given level of detail.



Style GAN: results

- The StyleGAN is both effective at generating large high-quality images and at controlling the style of the generated images.
- Can later extract the style vectors of an image, and synthesize an image combining both style features.



Style GANs: useful links

- ***[StyleGAN]*** Karras, T., Laine, S., & Aila, T. (2019). A style-based generator architecture for generative adversarial networks.
- Paper ***[StyleGAN]***: https://arxiv.org/pdf/1812.04948.pdf
- Demo code and pre-trained models (same repo as PGGAN): https://github.com/facebookresearch/pytorch_GAN_zoo
- Learn more (Nvidia/Youtube video with PGGANs paper results): https://www.youtube.com/watch?v=kSLJriaOumA&feature=emb_title