

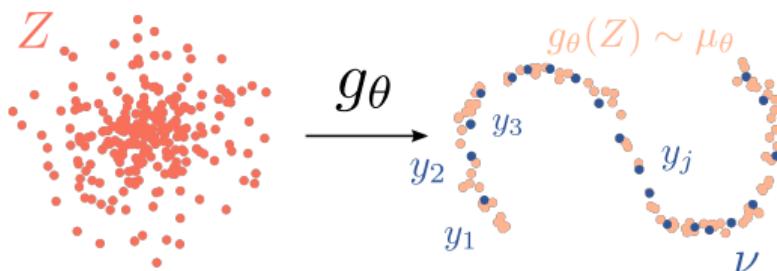
Generative Adversarial Networks

Arthur Leclaire



4AI07
June, 4th, 2025

Learning a Generative Network



GOAL: Estimate a generative model that fits a database $(y_j)_{1 \leq j \leq J}$ of images

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
4 4 8 4 4 4 4 4 4 4 4 4 4 4 4 4
5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6
7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7
8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8
9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9



Outline

In this session, we will study two approaches for learning generative models:

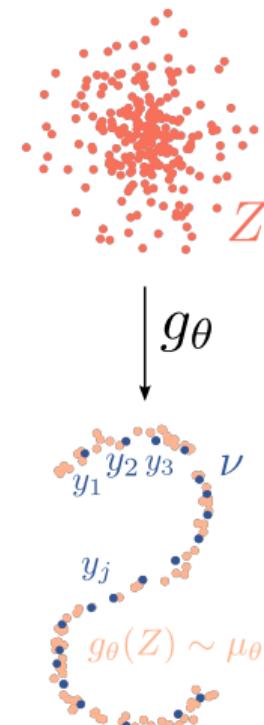
- **Generative Adversarial Networks (GANs)**
[Goodfellow et al., 2014]
- **Wasserstein Generative Adversarial Networks (WGANs)**
[Arjovsky et al., 2017]

Adversarial training: training simultaneously **a generator and a discriminator**.

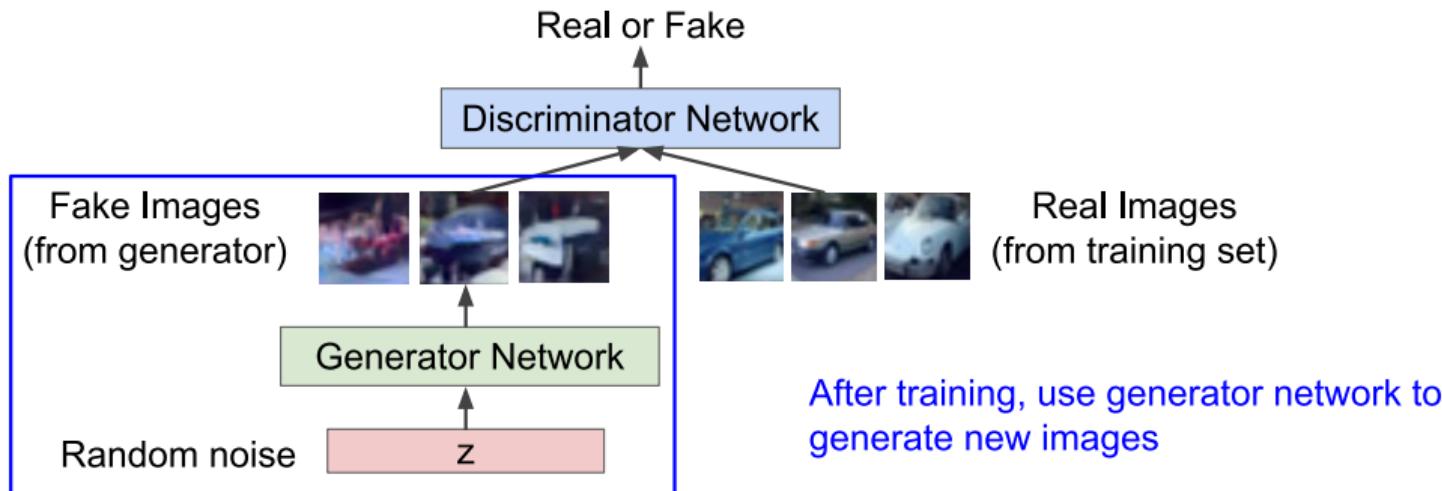
The discriminator D is trained to **discriminate real and fake samples**.

The generator g_θ is trained to fool the discriminator.

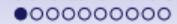
Adversarial training can be implemented with an alternate algorithm.



Generator v.s. Discriminator



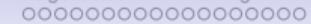
Generative Adversarial Networks (GAN)



Wasserstein GAN (WGAN)



Large-Scale GAN Training



Plan

Generative Adversarial Networks (GAN)

Wasserstein GAN (WGAN)

Large-Scale GAN Training

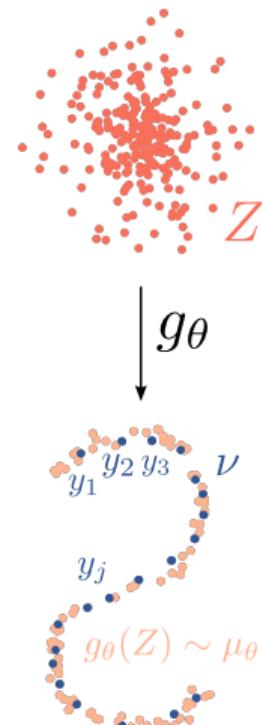
Loss function for Generative Modeling

Today, we will learn Generative Networks by solving

$$\inf_{\theta \in \Theta} \mathcal{L}(\mu_\theta, \nu)$$

where

- \mathcal{L} is a loss function between probability distributions μ, ν on $\mathcal{X}, \mathcal{Y} \subset \mathbf{R}^d$.
 - μ_θ is a probability distribution on a compact $\mathcal{X} \subset \mathbf{R}^d$:
 $g_\theta(Z) \sim \mu_\theta$ with g_θ neural network and $Z \sim \zeta$ input noise.
 - The generator is parameterized by a θ in a open set $\Theta \subset \mathbf{R}^q$.
 - ν is a probability on a compact $\mathcal{Y} \subset \mathbf{R}^d$:
 ν is the empirical distribution of the data.
 $Y \sim \nu$ will be a randomly chosen data point.



The Gist of Adversarial Training

- Train simultaneously a generator g_θ and a discriminator D with alternating updates:
 - Push the discriminator $D : \mathbf{R}^d \rightarrow [0, 1]$ to discriminate between real and fake samples:
 - $D(g_\theta(z))$ should be close to 0 for any z
 - $D(y_j)$ should be close to 1 for any data point y_j
 - Push the generator g_θ to fool the discriminator
 - $D(g_\theta(z))$ should be close to 1 for any z

The Gist of Adversarial Training

- Train simultaneously a generator g_θ and a discriminator D with alternating updates:
 - Push the discriminator $D : \mathbf{R}^d \rightarrow [0, 1]$ to discriminate between real and fake samples:
 - $D(g_\theta(z))$ should be close to 0 for any z
 - $D(y_j)$ should be close to 1 for any data point y_j
 - Push the generator g_θ to fool the discriminator
 - $D(g_\theta(z))$ should be close to 1 for any z
 - The discriminator solves a binary classification problem between real and fake images:

$$\max_{D \in \mathcal{D}} \mathbb{E}[\log D(Y)] + \mathbb{E}[\log(1 - D(g_\theta(Z)))]$$

where \mathcal{D} is a (parametric) set of measurable functions $D : \mathbf{R}^d \rightarrow [0, 1]$. ($\log 0 = -\infty$.)

The Gist of Adversarial Training

- Train simultaneously a generator g_θ and a discriminator D with alternating updates:
 - Push the discriminator $D : \mathbf{R}^d \rightarrow [0, 1]$ to discriminate between real and fake samples:
 - $D(g_\theta(z))$ should be close to 0 for any z
 - $D(y_j)$ should be close to 1 for any data point y_j
 - Push the generator g_θ to fool the discriminator
 - $D(g_\theta(z))$ should be close to 1 for any z
 - The discriminator solves a binary classification problem between real and fake images:

$$\max_{D \in \mathcal{D}} \mathbb{E}[\log D(Y)] + \mathbb{E}[\log(1 - D(g_\theta(Z)))]$$

where \mathcal{D} is a (parametric) set of measurable functions $D : \mathbf{R}^d \rightarrow [0, 1]$. ($\log 0 = -\infty$.)

- Finally, GAN training is a **min-max** two-player game:

$$\min_{\theta \in \Theta} \max_{D \in \mathcal{D}} \mathcal{L}_{\text{GAN}}(\theta, D) := \mathbb{E}[\log D(Y)] + \mathbb{E}[\log(1 - D(g_\theta(Z)))]$$

Classification of fake points vs data points

In practice, the expectations are approximated using finite sums on samples.

With a finite sample $(x^{(i)})$ of real and fake points, it comes down to a logistic regression problem.

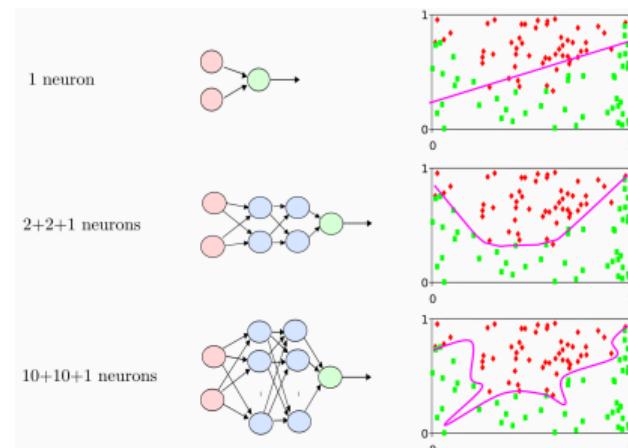
Let

- $\ell^{(i)} = 1$ if $x^{(i)}$ is one of the data points (y_j),
 - $\ell^{(i)} = 0$ if $x^{(i)}$ is a generated point $g_\theta(Z)$.

We can then use the binary cross-entropy loss:

$$\frac{1}{N} \sum_{i=1}^N \left[\ell^{(i)} \log D(x^{(i)}) + (1 - \ell^{(i)}) \log (1 - D(x^{(i)})) \right]$$

\rightarrow torch.nn.BCELoss



Classification of fake points vs data points

In practice, the expectations are approximated using finite sums on samples.

With a finite sample $(x^{(i)})$ of real and fake points, it comes down to a logistic regression problem.

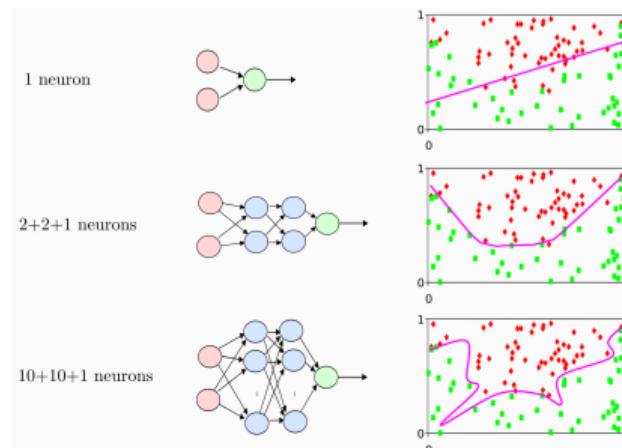
Let

- $\ell^{(i)} = 1$ if $x^{(i)}$ is one of the data points (y_j),
 - $\ell^{(i)} = 0$ if $x^{(i)}$ is a generated point $g_\theta(Z)$.

We can then use the binary cross-entropy loss:

$$\frac{1}{N} \sum_{i=1}^N \left[\ell^{(i)} \log D(x^{(i)}) + (1 - \ell^{(i)}) \log (1 - D(x^{(i)})) \right]$$

→ torch.nn.BCELoss



Warning: If real and fake batches don't have same size, the sums should be averaged accordingly (in order to avoid class unbalance).

Training Algorithm

- In practice, g_θ and D are parameterized by neural networks.

Here, D must have values in $[0, 1]$: take last layer as sigmoid activation $\sigma(x) = \frac{1}{1+e^{-x}}$.
 (Alternately, use `torch.nn.BCEWithLogitsLoss`)

Training Algorithm

- In practice, g_θ and D are parameterized by neural networks.

Here, D must have values in $[0, 1]$: take last layer as sigmoid activation $\sigma(x) = \frac{1}{1+e^{-x}}$.
 (Alternately, use `torch.nn.BCEWithLogitsLoss`)

- The GAN training algorithm alternates between

- Ascent step(s) on $D \mapsto \mathbb{E}[\log D(Y)] + \mathbb{E}[\log(1 - D(g_\theta(Z)))]$
 - Descent step(s) on $\theta \mapsto \mathbb{E}[\log(1 - D(g_\theta(Z)))]$

Training Algorithm

- In practice, g_θ and D are parameterized by neural networks.
Here, D must have values in $[0, 1]$: take last layer as sigmoid activation $\sigma(x) = \frac{1}{1+e^{-x}}$.
(Alternately, use `torch.nn.BCEWithLogitsLoss`)
 - The GAN training algorithm alternates between
 - Ascent step(s) on $D \mapsto \mathbb{E}[\log D(Y)] + \mathbb{E}[\log(1 - D(g_\theta(Z)))]$
 - Descent step(s) on $\theta \mapsto \mathbb{E}[\log(1 - D(g_\theta(Z)))]$
 - For each step, use stochastic gradient-based updates (with optimizers SGD, ADAM, ...).
Each step requires to take samples of $g_\theta(Z)$ and Y

Training Algorithm

- In practice, g_θ and D are parameterized by neural networks.
Here, D must have values in $[0, 1]$: take last layer as sigmoid activation $\sigma(x) = \frac{1}{1+e^{-x}}$.
(Alternately, use `torch.nn.BCEWithLogitsLoss`)
 - The GAN training algorithm alternates between
 - Ascent step(s) on $D \mapsto \mathbb{E}[\log D(Y)] + \mathbb{E}[\log(1 - D(g_\theta(Z)))]$
 - Descent step(s) on $\theta \mapsto \mathbb{E}[\log(1 - D(g_\theta(Z)))]$
 - For each step, use stochastic gradient-based updates (with optimizers SGD, ADAM, ...).
Each step requires to take samples of $g_\theta(Z)$ and Y
 - The update of θ may be based on the *non-saturating loss* $\theta \mapsto \mathbb{E}[\log(D(g_\theta(Z)))]$

When $D(g_\theta(Z))$ goes to 1, then $\log(1 - D(g_\theta(Z)))$ saturates close to 0.

Illustration with a 2D example

Question: can you imagine a good discriminator for the following configuration?

- Dark blue: data points $(y_j)_{1 \leq j \leq J}$
 - Light blue: 100 samples $(g_\theta(z_k))_{1 \leq k \leq 100}$ of μ_θ

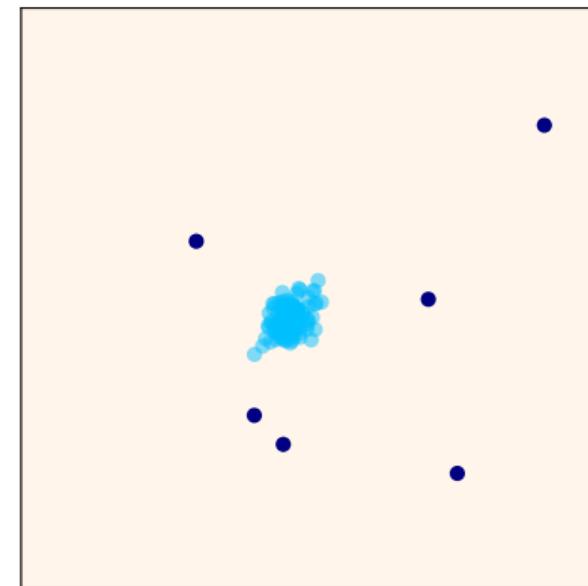


Illustration with a 2D example

Question: can you imagine a good discriminator for the following configuration?

- Dark blue: data points $(y_j)_{1 \leq j \leq J}$
 - Light blue: 100 samples $(g_\theta(z_k))_{1 \leq k \leq 100}$ of μ_θ

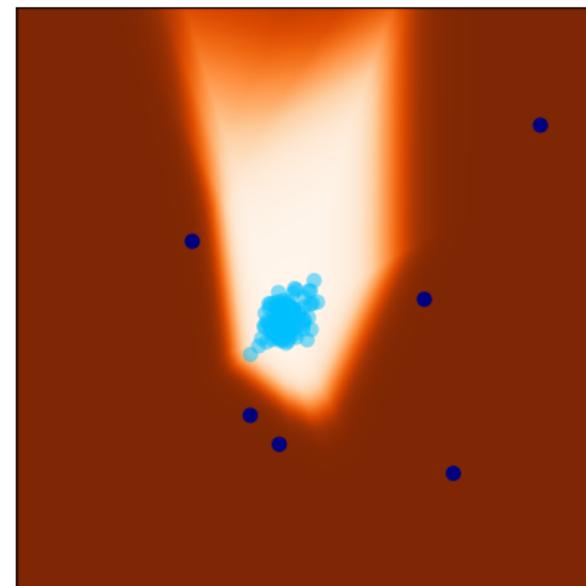
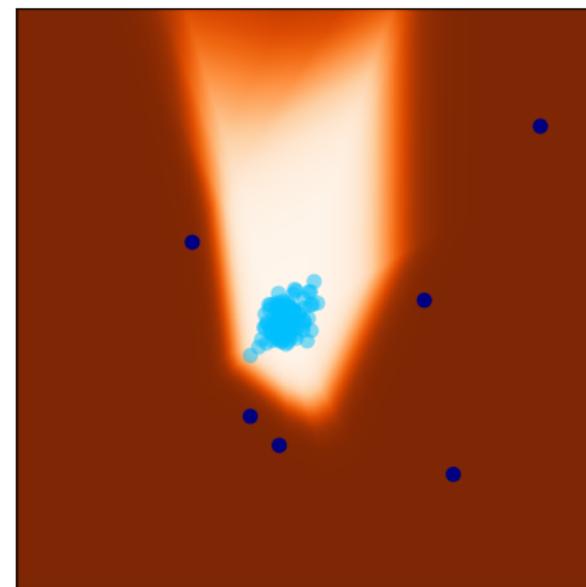


Illustration with a 2D example

Question: can you imagine a good discriminator for the following configuration?

- Dark blue: data points $(y_j)_{1 \leq j \leq J}$
 - Light blue: 100 samples $(g_\theta(z_k))_{1 \leq k \leq 100}$ of μ_θ



Problem: D is close to 0 on $\text{Supp}(\mu_\theta)$ \rightarrow “vanishing gradients” issue (on ∇_θ)

Illustration with a 2D example

And now a tougher example...

- Dark blue: data points $(y_j)_{1 \leq j \leq J}$
- Light blue: 100 samples $(g_\theta(z_k))_{1 \leq k \leq 100}$ of μ_θ

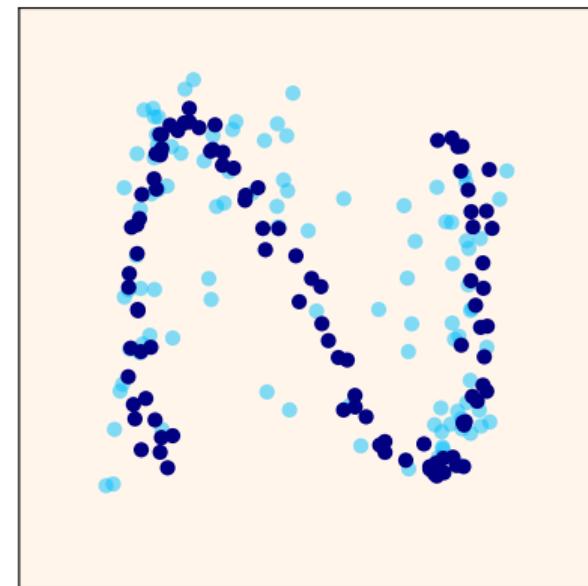
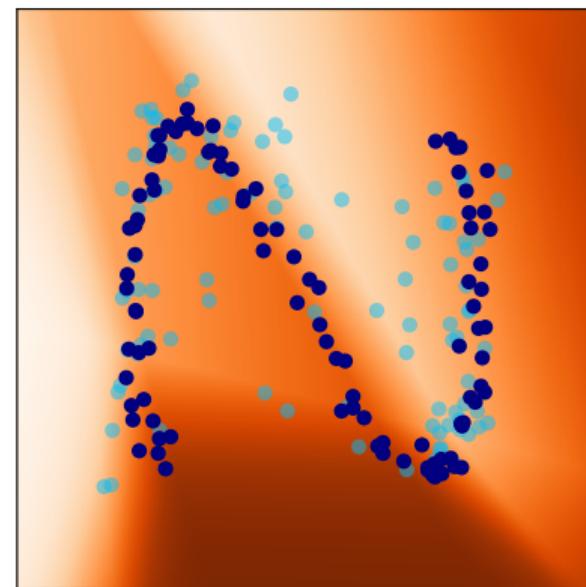


Illustration with a 2D example

And now a tougher example...

- Dark blue: data points $(y_j)_{1 \leq j \leq J}$
 - Light blue: 100 samples $(g_\theta(z_k))_{1 \leq k \leq 100}$ of μ_θ



Optimal Discriminator

Let \mathcal{D}_∞ the set of measurable functions from \mathbf{R}^d to $[0, 1]$, and let

$$L(\theta, D) = \mathcal{L}_{\text{GAN}}(\theta, D) = \int \log(D)d\nu + \int \log(1 - D)d\mu_\theta.$$

Remark that

$$0 \geqslant \sup_{D \in \mathcal{D}_\infty} L(\theta, D) \geqslant L(\theta, \frac{1}{2}) = -\log 4.$$

Proposition ([Biau et al., 2018])

For fixed θ , assume that μ_θ and ν have densities p_θ, q with respect to a reference M . Then

$$\sup_{D \in \mathcal{D}_\infty} L(\theta, D) = L(\theta, D_\theta^*) \quad \text{with} \quad D_\theta^* = \frac{q}{q + p_\theta}.$$

Relation with Jensen-Shannon divergence

Let us define the Kullback-Leibler divergence between probability measures μ, ν by

$$\text{KL}(\mu|\nu) = \begin{cases} \int \log\left(\frac{d\mu}{d\nu}\right) d\mu & \text{if } \frac{d\mu}{d\nu} \text{ exists} \\ +\infty & \text{otherwise} \end{cases}$$

and the Jensen-Shannon divergence by

$$\text{JS}(\mu, \nu) = \frac{1}{2} \text{KL}\left(\mu, \frac{\mu+\nu}{2}\right) + \frac{1}{2} \text{KL}\left(\nu, \frac{\mu+\nu}{2}\right).$$

Relation with Jensen-Shannon divergence

Let us define the Kullback-Leibler divergence between probability measures μ, ν by

$$\text{KL}(\mu|\nu) = \begin{cases} \int \log\left(\frac{d\mu}{d\nu}\right)d\mu & \text{if } \frac{d\mu}{d\nu} \text{ exists} \\ +\infty & \text{otherwise} \end{cases}$$

and the Jensen-Shannon divergence by $\text{JS}(\mu, \nu) = \frac{1}{2} \text{KL}(\mu, \frac{\mu+\nu}{2}) + \frac{1}{2} \text{KL}(\nu, \frac{\mu+\nu}{2}).$

Proposition ([Biau et al., 2018])

We have

$$\sup_{D \in \mathcal{D}_\infty} L(\theta, D) = L(\theta, D_\theta^*) = 2 \text{JS}(\mu_\theta, \nu) - \log 4.$$

Relation with Jensen-Shannon divergence

Let us define the Kullback-Leibler divergence between probability measures μ, ν by

$$\text{KL}(\mu|\nu) = \begin{cases} \int \log\left(\frac{d\mu}{d\nu}\right) d\mu & \text{if } \frac{d\mu}{d\nu} \text{ exists} \\ +\infty & \text{otherwise} \end{cases}$$

and the Jensen-Shannon divergence by $\text{JS}(\mu, \nu) = \frac{1}{2} \text{KL}(\mu, \frac{\mu+\nu}{2}) + \frac{1}{2} \text{KL}(\nu, \frac{\mu+\nu}{2}).$

Proposition ([Biau et al., 2018])

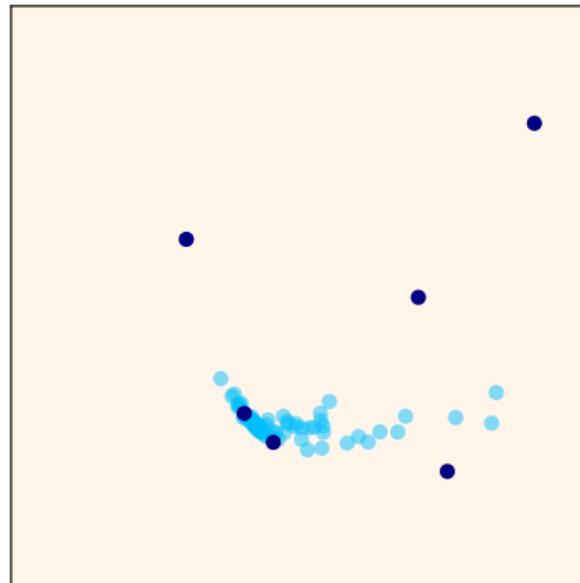
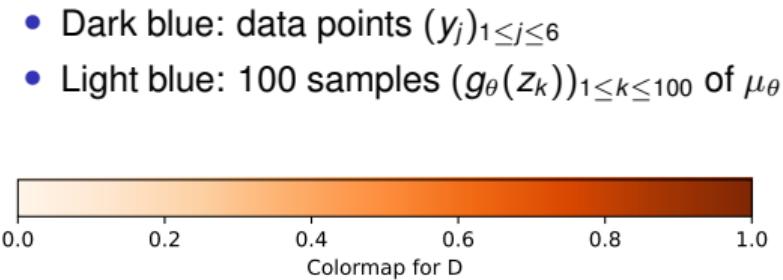
We have

$$\sup_{D \in \mathcal{D}_\infty} L(\theta, D) = L(\theta, D_\theta^*) = 2 \text{JS}(\mu_\theta, \nu) - \log 4.$$

- If μ_θ and ν have disjoint supports, then $\text{JS}(\mu_\theta, \nu) = \log 2$. → vanishing gradient issue!
- Why does it work then?
→ Because the parameterized discriminator is in practice smoother than D_θ^* .

What did you expect?

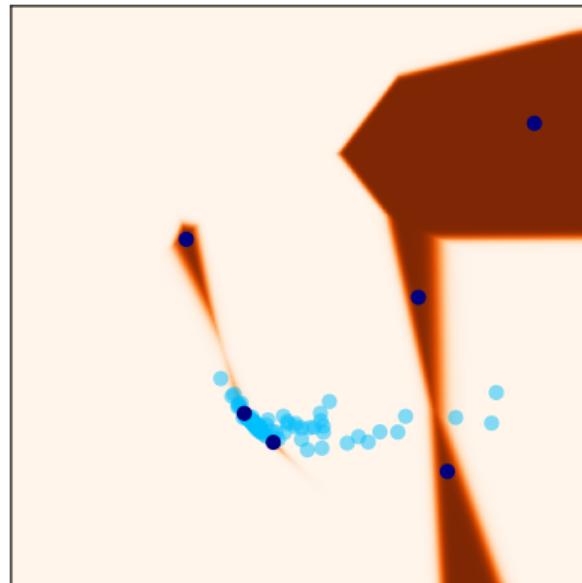
Final configuration. What is the final discriminator?



What did you expect?

Final configuration. What is the final discriminator?

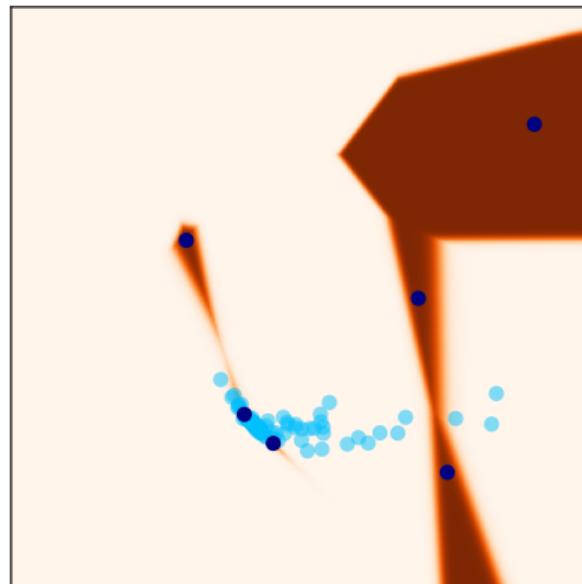
- Dark blue: data points $(y_j)_{1 \leq j \leq 6}$
- Light blue: 100 samples $(g_\theta(z_k))_{1 \leq k \leq 100}$ of μ_θ



What did you expect?

What happens if we update only the generator?

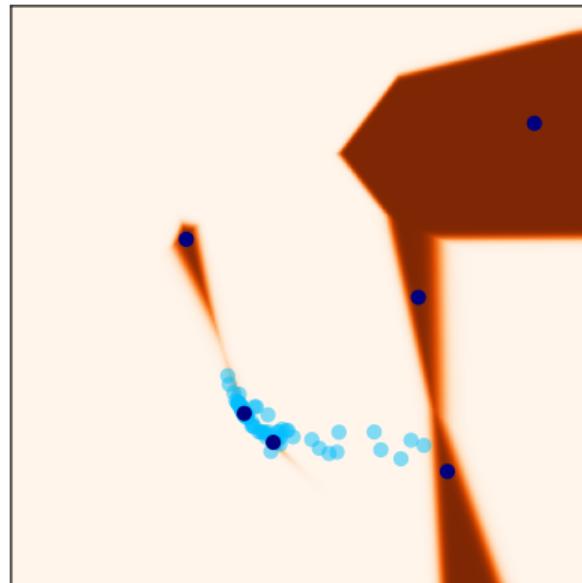
- Dark blue: data points $(y_j)_{1 \leq j \leq 6}$
- Light blue: 100 samples $(g_\theta(z_k))_{1 \leq k \leq 100}$ of μ_θ



What did you expect?

What happens if we update only the generator?

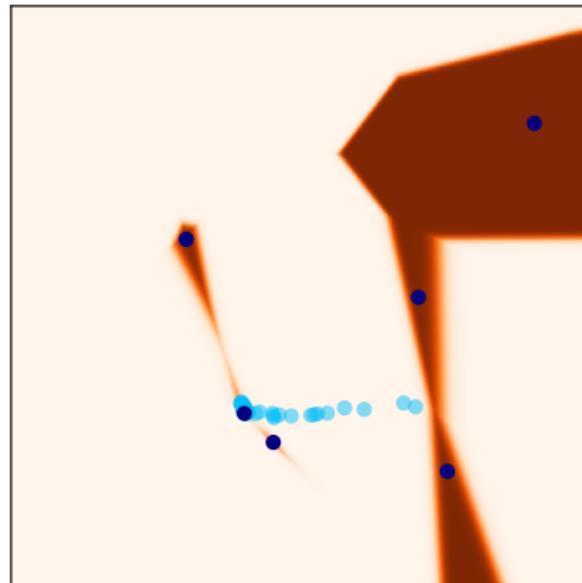
- Dark blue: data points $(y_j)_{1 \leq j \leq 6}$
- Light blue: 100 samples $(g_\theta(z_k))_{1 \leq k \leq 100}$ of μ_θ



What did you expect?

What happens if we update only the generator?

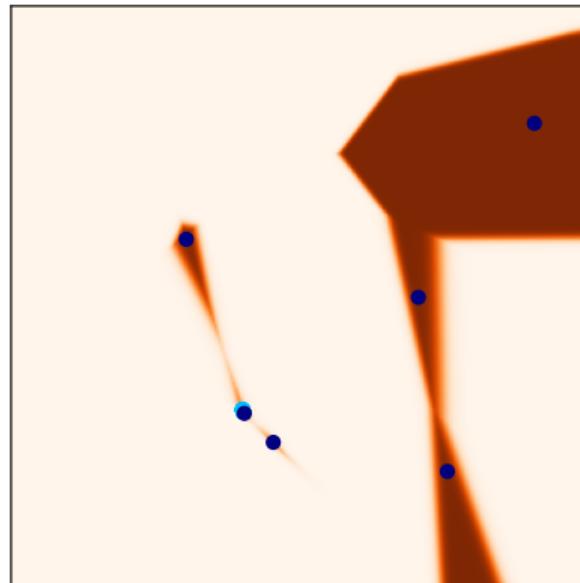
- Dark blue: data points $(y_j)_{1 \leq j \leq 6}$
- Light blue: 100 samples $(g_\theta(z_k))_{1 \leq k \leq 100}$ of μ_θ



What did you expect?

What happens if we update only the generator?

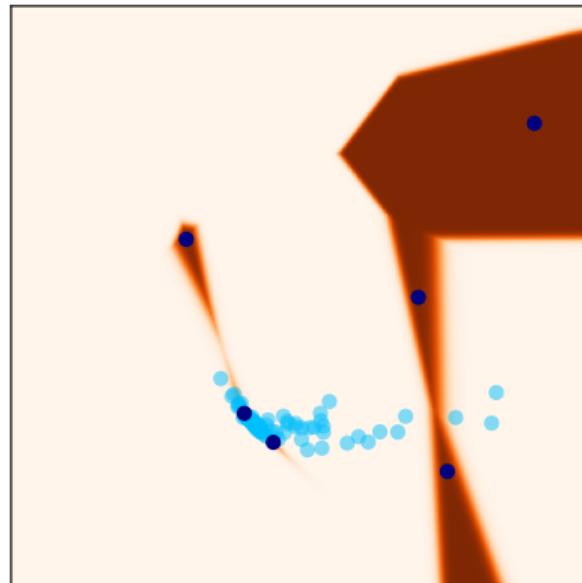
- Dark blue: data points $(y_j)_{1 \leq j \leq 6}$
- Light blue: 100 samples $(g_\theta(z_k))_{1 \leq k \leq 100}$ of μ_θ



What did you expect?

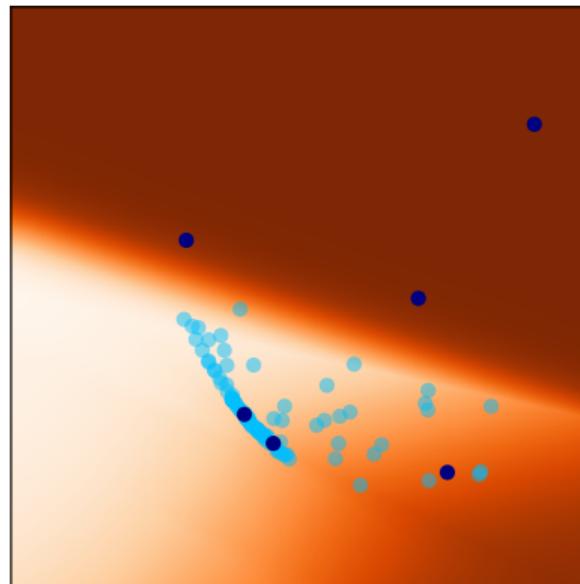
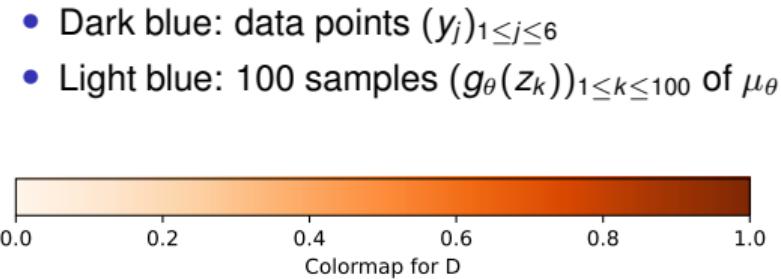
And if we retrain the discriminator?

- Dark blue: data points $(y_j)_{1 \leq j \leq 6}$
- Light blue: 100 samples $(g_\theta(z_k))_{1 \leq k \leq 100}$ of μ_θ



What did you expect?

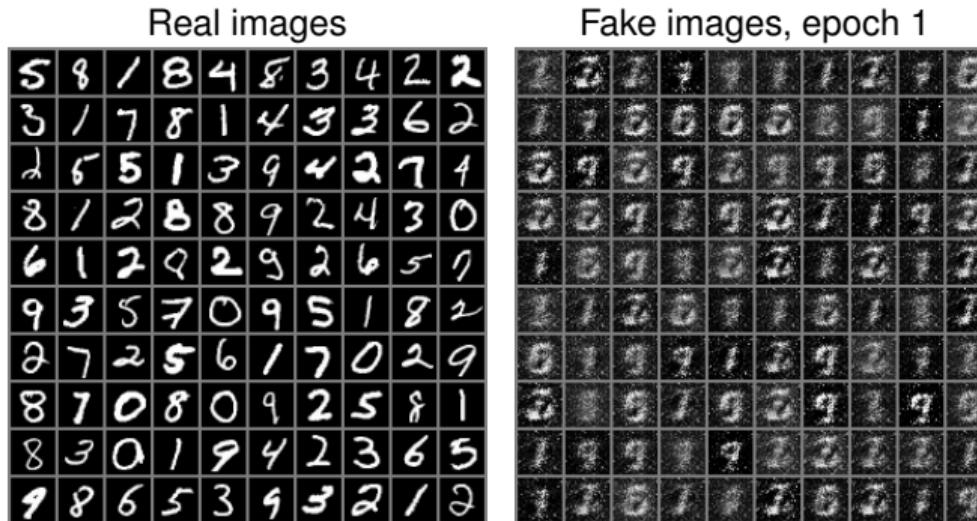
And if we retrain the discriminator?



GAN Training for MNIST digits

Training with MNIST (60 000 images)

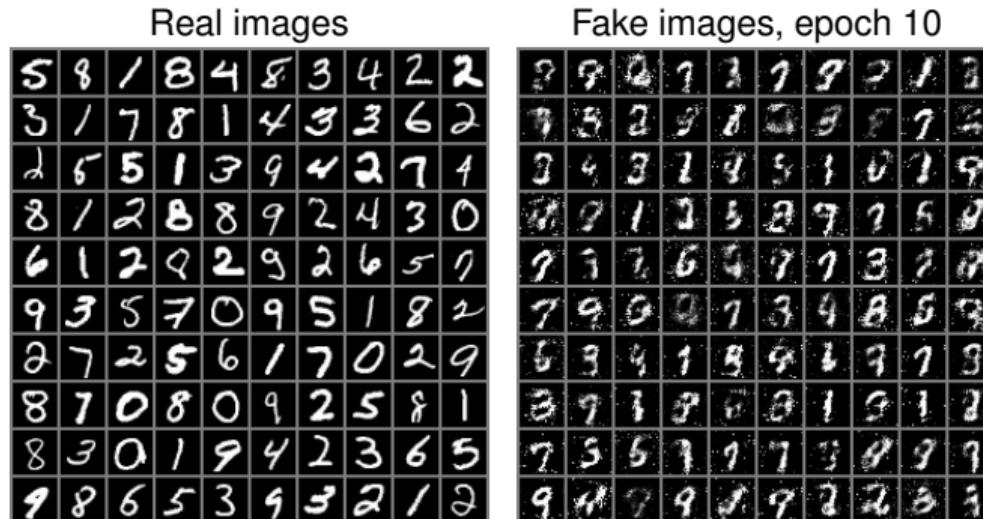
- Adam optimizer
- Learning rate 0.0002 for both the discriminator and the generator



GAN Training for MNIST digits

Training with MNIST (60 000 images)

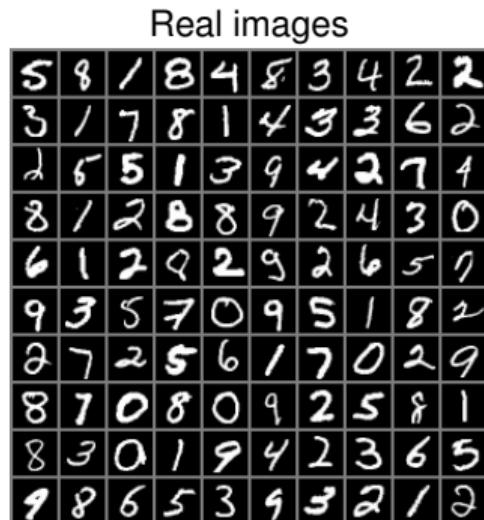
- Adam optimizer
- Learning rate 0.0002 for both the discriminator and the generator



GAN Training for MNIST digits

Training with MNIST (60 000 images)

- Adam optimizer
- Learning rate 0.0002 for both the discriminator and the generator



GAN Training for MNIST digits

Training GANs is quite unstable!

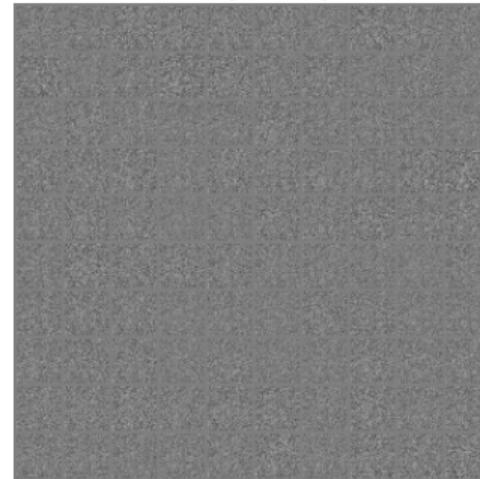
The generator can suffer from *mode collapse*:

i.e. it always produces the same image (one mode only).

Example: same as before **but with SGD instead of Adam.**



Fake images, epoch 1



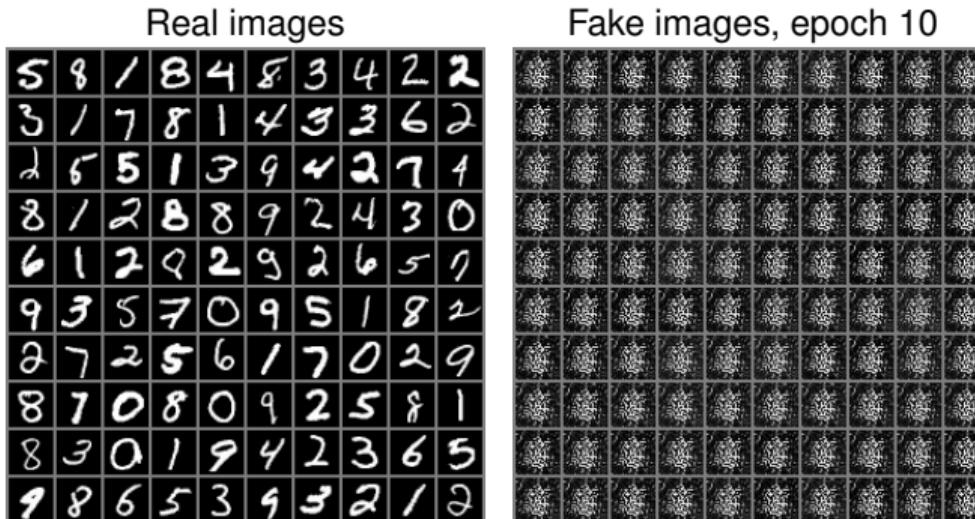
GAN Training for MNIST digits

Training GANs is quite unstable!

The generator can suffer from *mode collapse*:

i.e. it always produces the same image (one mode only).

Example: same as before **but with SGD instead of Adam.**



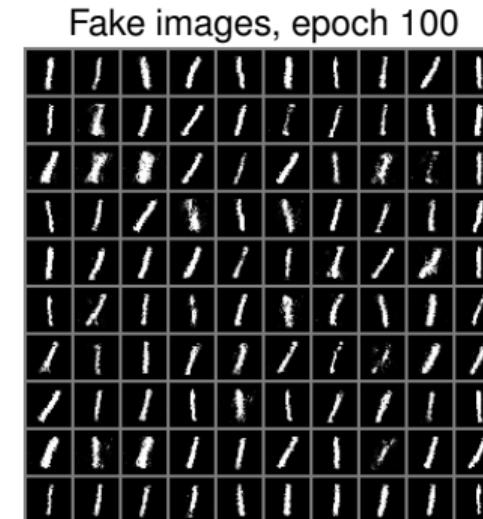
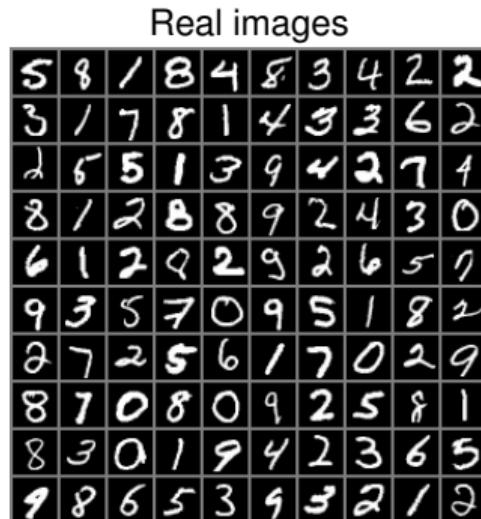
GAN Training for MNIST digits

Training GANs is quite unstable!

The generator can suffer from *mode collapse*:

i.e. it always produces the same image (one mode only).

Example: same as before **but with SGD instead of Adam.**



Generative Adversarial Networks (GAN)
oooooooooooo

Wasserstein GAN (WGAN)
●oooooooooooo

Large-Scale GAN Training
oooooooooooooooooooo

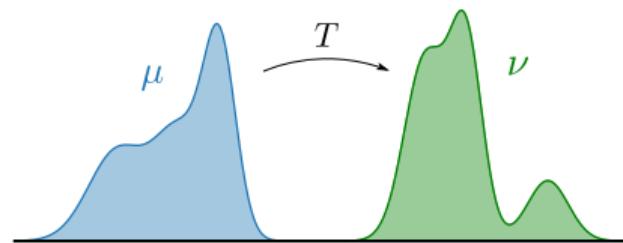
Plan

Generative Adversarial Networks (GAN)

Wasserstein GAN (WGAN)

Large-Scale GAN Training

Wasserstein GANs and Optimal Transport



WGAN are based on **Optimal Transport**.

The Optimal Transport Cost measures the cost of moving some mass μ onto some mass ν .

It is sensitive to a “groundcost”

$$c : \mathbf{R}^d \times \mathbf{R}^d \longrightarrow \mathbf{R}.$$

For example: $c(x, y) = \|x - y\|$.

OT has many applications in image processing.



COLOR TRANSFER

Two formulations of the OT cost

Let μ, ν two probability distributions supported in $\mathcal{X}, \mathcal{Y} \subset \mathbf{R}^d$. Let $c : \mathcal{X} \times \mathcal{Y} \rightarrow \mathbf{R}$ be continuous.

The optimal transport cost is defined by

$$W(\mu, \nu) = \inf_{\pi \in \Pi(\mu, \nu)} \int_{\mathcal{X} \times \mathcal{Y}} c(x, y) \, d\pi(x, y)$$

where $\Pi(\mu, \nu)$ is the set of distributions π on $\mathcal{X} \times \mathcal{Y}$ with marginals μ, ν .

For $c(x, y) = \|x - y\|^p$, $p \in [1, \infty]$, this defines the p -Wasserstein cost W_p .

Two formulations of the OT cost

Let μ, ν two probability distributions supported in $\mathcal{X}, \mathcal{Y} \subset \mathbf{R}^d$. Let $c : \mathcal{X} \times \mathcal{Y} \rightarrow \mathbf{R}$ be continuous.

The optimal transport cost is defined by

$$W(\mu, \nu) = \inf_{\pi \in \Pi(\mu, \nu)} \int_{\mathcal{X} \times \mathcal{Y}} c(x, y) \, d\pi(x, y)$$

where $\Pi(\mu, \nu)$ is the set of distributions π on $\mathcal{X} \times \mathcal{Y}$ with marginals μ, ν .

For $c(x, y) = \|x - y\|^p$, $p \in [1, \infty]$, this defines the p -Wasserstein cost W_p .

The “dual formulation” of the 1-Wasserstein cost is given by

$$W_1(\mu, \nu) = \sup_{\psi \in \text{Lip}_1} - \int \psi(x) d\mu(x) + \int \psi(y) d\nu(y)$$

where Lip_1 is the set of 1 Lipschitz functions $\psi : \mathbf{R}^d \rightarrow \mathbf{R}$.

Wasserstein Generative Networks (WGAN)

[Arjovsky et al., 2017]

Learning a Wasserstein WGAN consists in solving

$$\underset{\theta \in \Theta}{\operatorname{Argmin}} W_1(\mu_\theta, \nu),$$

For $c(x, y) = \|x - y\|$, we get the usual WGAN formulation:

$$\mathcal{L}_{\text{WGAN}}(\theta, D) = \mathbb{E}[D(Y)] - \mathbb{E}[D(g_\theta(Z))],$$

where the discriminator D should be 1-Lipschitz.

- ✓ Benefit of the Wasserstein cost over the Jensen-Shannon divergence:
It is sensitive to the distance between the supports of μ_θ and ν .

Compare Loss functions

- Loss function for **GAN**:

$$\sup_{D \in \mathcal{D}_\infty} \mathbb{E}[\log D(Y)] + \mathbb{E}[\log(1 - D(g_\theta(Z)))]$$

- Loss function for **WGAN**:

$$\sup_{D \in \text{Lip}_1} \mathbb{E}[D(Y)] - \mathbb{E}[D(g_\theta(Z))].$$

Compare Loss functions

- Loss function for **GAN**:

$$\sup_{D \in \mathcal{D}_\infty} \mathbb{E}[\log D(Y)] + \mathbb{E}[\log(1 - D(g_\theta(Z)))]$$

- Loss function for **WGAN**:

$$\sup_{D \in \text{Lip}_1} \mathbb{E}[D(Y)] - \mathbb{E}[D(g_\theta(Z))].$$

We just got rid of the log and $D(x)$ is not in $[0, 1] \dots$ but we now have a constraint " $D \in \text{Lip}_1$ ".

Compare Loss functions

- Loss function for GAN:

$$\sup_{D \in \mathcal{D}_{\infty}} \mathbb{E}[\log D(Y)] + \mathbb{E}[\log(1 - D(g_{\theta}(Z)))]$$

- Loss function for **WGAN**:

$$\sup_{D \in \text{Lip}_1} \mathbb{E}[D(Y)] - \mathbb{E}[D(g_\theta(Z))].$$

We just got rid of the log and $D(x)$ is not in $[0, 1]$... but we now have a constraint " $D \in \text{Lip}_1$ ".

- The WGAN training algorithm alternates between
 - Ascent step(s) on $D \mapsto \mathbb{E}[D(Y)] - \mathbb{E}[D(g_\theta(Z))]$
 - Descent step(s) on $\theta \mapsto \mathbb{E}[-D(g_\theta(Z))]$

Compare Loss functions

- Loss function for **GAN**:

$$\sup_{D \in \mathcal{D}_\infty} \mathbb{E}[\log D(Y)] + \mathbb{E}[\log(1 - D(g_\theta(Z)))]$$

- Loss function for **WGAN**:

$$\sup_{D \in \text{Lip}_1} \mathbb{E}[D(Y)] - \mathbb{E}[D(g_\theta(Z))].$$

We just got rid of the log and $D(x)$ is not in $[0, 1] \dots$ but we now have a constraint " $D \in \text{Lip}_1$ ".

- The WGAN training algorithm alternates between
 - Ascent step(s) on $D \mapsto \mathbb{E}[D(Y)] - \mathbb{E}[D(g_\theta(Z))]$
 - Descent step(s) on $\theta \mapsto \mathbb{E}[-D(g_\theta(Z))]$
- But, we have to constrain $D \in \text{Lip}_1$ along the way...

Learning Lipschitz discriminators

- The original WGAN paper [Arjovsky et al., 2017] uses weight clipping to decrease the Lipschitz constant:

```
for p in D.parameters():
    p.data.clamp_(-c, c)
```

- Alternately, [Gulrajani et al., 2017] proposed to control more precisely the Lipschitz constant of D .
- We can do that by constraining the norms of ∇D at many points a_i .

Estimation of the mean variation

We can estimate a sort of mean variation of the discriminator D with the following:

- Take two batches $(x_i), (y_i)$ of same size.
- Sample the segments $[x_i, y_i]$:
$$a_i = (1 - u_i x_i) + u_i y_i \quad \text{with} \quad u_i \sim \mathcal{U}(0, 1).$$
- Compute $\nabla D(a_i)$ by automatic differentiation.
- Average over all a_i 's.

```
def meanvariation(D,x,y):
    # Calculate interpolation between points in batches x and y (of same size b)
    b = x.shape[0]
    if y.shape[0] != b:
        print('wrong size')
    alpha = torch.rand((b,1,1,1),device=device)
    interp = alpha * y + (1 - alpha) * x
    interp.requires_grad_()

    # Calculate probability of interpolated examples
    Di = D(interp).view(-1)

    # Calculate gradients of probabilities with respect to examples
    gradout = torch.ones(Di.size()).to(device)
    gradients = torch.autograd.grad(outputs=Di, inputs=interp, grad_outputs=gradout)[0]

    # Derivatives of the gradient close to 0 can cause problems because of
    # the square root, so manually calculate norm and add epsilon
    gradients_norm = torch.sqrt(torch.sum(gradients ** 2, dim=[1,2,3]))
```

The Gradient Penalty

[Gulrajani et al., 2017]

- Actually, Gulrajani et al. propose to use a finer property of W_1 :
the optimal dual potential φ satisfies $\|\nabla\varphi\| = 1$ on segments joining samples from μ_θ and ν .
(see e.g. [Santambrogio, 2015], and also a remark later in these slides)
- Therefore, they proposed to include a “gradient penalty” in the loss:

$$\text{GP}(D) = \mathbb{E}[(\|\nabla D(A)\| - 1)^2] \quad \text{where } A \sim \mathcal{U}([g_\theta(Z), Y]).$$

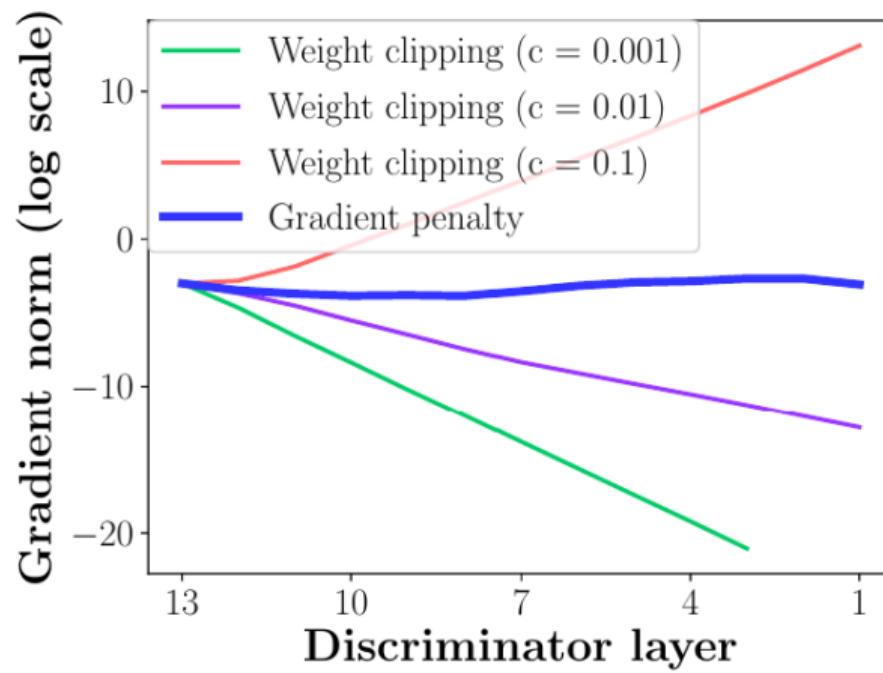
Warning: the gradient is with respect to the variable x and not the parameters θ .

- This leads to the **WGAN-GP** discriminator loss (with penalty weight $\lambda > 0$):

$$\sup_D \mathbb{E}[D(Y)] - \mathbb{E}[D(g_\theta(Z))] - \lambda \text{GP}(D).$$

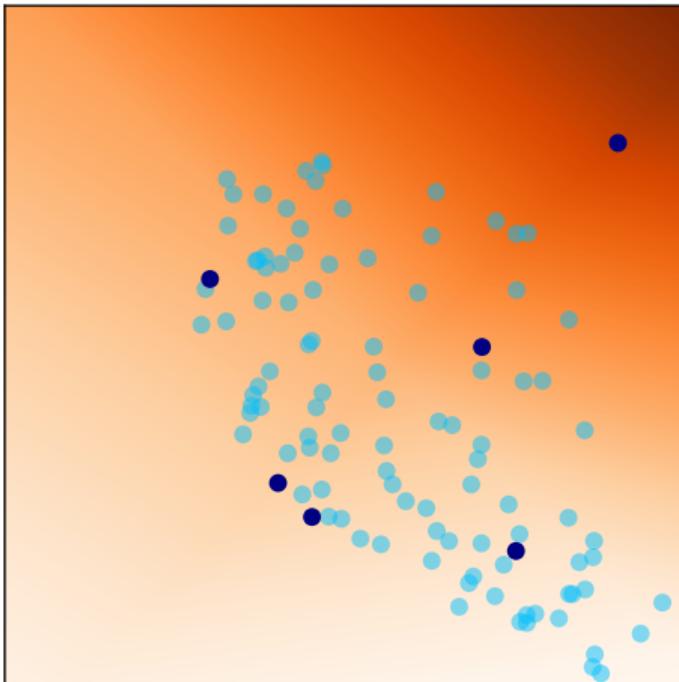
- We could also do a unilateral penalty $\mathbb{E}[(\|\nabla D(X)\| - 1)_+^2]$.

WGAN: Gradient Penalty v.s. Weight clipping

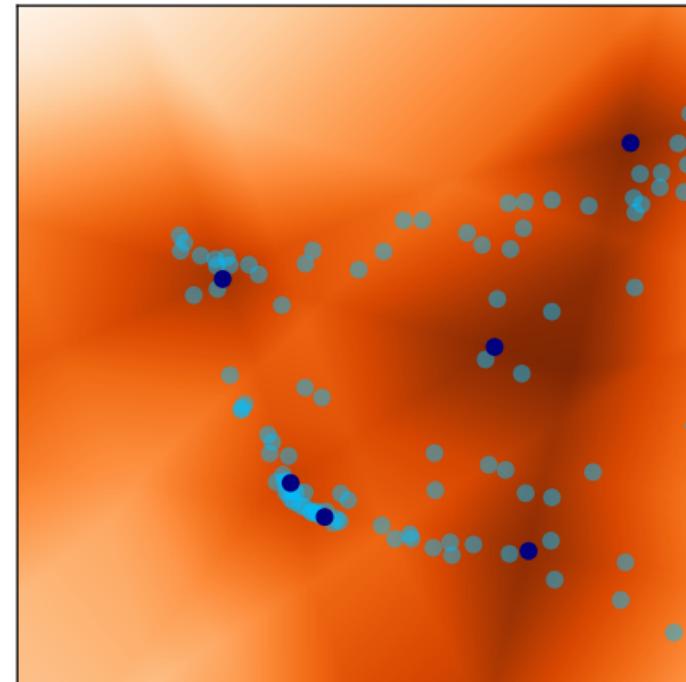


(source: [Gulrajani et al., 2017])

Example of WGAN training



WGAN-WC



WGAN-GP

WGAN Training for MNIST digits

Training with MNIST (60 000 images)

- Adam optimizer
- Learning rate 0.0002 for both the discriminator and the generator

Real images



Fake images, epoch 1



WGAN Training for MNIST digits

Training with MNIST (60 000 images)

- Adam optimizer
- Learning rate 0.0002 for both the discriminator and the generator

Real images



Fake images, epoch 10



WGAN Training for MNIST digits

Training with MNIST (60 000 images)

- Adam optimizer
- Learning rate 0.0002 for both the discriminator and the generator

Real images

5	8	1	8	4	8	3	4	2	2
3	1	7	8	1	4	3	3	6	2
2	6	5	1	3	9	4	2	7	4
8	1	2	8	8	9	2	4	3	0
6	1	2	9	2	9	2	6	5	1
9	3	5	7	0	9	5	1	8	2
2	7	2	5	6	1	7	0	2	9
8	7	0	8	0	9	2	5	8	1
8	3	0	1	9	4	2	3	6	5
9	8	6	5	3	9	3	2	1	2

Fake images, epoch 20

8	3	6	6	8	1	1	3	0	2	6	4	9	4	5
0	6	0	0	6	1	6	2	4	6	5	0	9	0	0
7	5	9	0	0	2	7	0	0	3	1	0	1	9	0
0	4	1	9	0	1	8	3	3	7	0	9	7	2	1
5	6	5	5	0	4	1	3	4	3	8	4	9	8	0
4	1	1	0	8	3	0	6	5	0	9	4	0	8	7
9	4	2	1	5	6	8	6	8	1	0	5	5	2	8
8	6	7	0	1	0	9	5	7	6	8	3	3	0	0

WGAN Training for MNIST digits

Training with MNIST (60 000 images)

- Adam optimizer
- Learning rate 0.0002 for both the discriminator and the generator

Real images



Fake images, epoch 30



WGAN Training for MNIST digits

Training with MNIST (60 000 images)

- Adam optimizer
- Learning rate 0.0002 for both the discriminator and the generator

Real images



Fake images, epoch 40



WGAN Training for MNIST digits

Training with MNIST (60 000 images)

- Adam optimizer
- Learning rate 0.0002 for both the discriminator and the generator

Real images

5	8	1	8	4	8	3	4	2	2
3	1	7	8	1	4	3	3	6	2
2	6	5	1	3	9	4	2	7	4
8	1	2	8	8	9	2	4	3	0
6	1	2	9	2	9	2	6	5	1
9	3	5	7	0	9	5	1	8	2
2	7	2	5	6	1	7	0	2	9
8	7	0	8	0	9	2	5	8	1
8	3	0	1	9	4	2	3	6	5
9	8	6	5	3	9	3	2	1	2

Fake images, epoch 50

8	3	6	6	6	6	1	1	1	3	0	2	1	9	9	4	6
7	6	2	0	6	1	6	3	9	6	9	5	0	0	0	4	
1	5	9	6	0	2	7	8	0	3	2	0	3	8	0	5	
0	1	1	5	8	1	8	0	3	2	0	0	7	2	4	8	
3	6	5	5	5	5	1	3	8	2	8	4	5	2	0	8	
4	1	1	0	9	3	0	6	5	0	9	4	0	8	1	4	
4	5	2	1	8	6	7	6	5	1	0	5	5	2	8	8	
8	6	7	0	1	0	4	5	3	8	8	0	3	0	0	4	

WGAN Training for MNIST digits

Training with MNIST (60 000 images)

- Adam optimizer
- Learning rate 0.0002 for both the discriminator and the generator

Real images



Fake images, epoch 100



WGAN Stability

WGAN-GP is a more stable way to train deep convolutional generators/discriminators.
But the results still depend highly on the optimization strategy and on the networks architectures.

DCGAN	LSGAN	WGAN (clipping)	WGAN-GP (ours)
Baseline (G : DCGAN, D : DCGAN)			
			
G : No BN and a constant number of filters, D : DCGAN			
			
G : 4-layer 512-dim ReLU MLP, D : DCGAN			
			
No normalization in either G or D			
			
Gated multiplicative nonlinearities everywhere in G and D			
			
$tanh$ nonlinearities everywhere in G and D			
			
101-layer ResNet G and D			
			

Figure 2: Different GAN architectures trained with different methods. We only succeeded in training every architecture with a shared set of hyperparameters using WGAN-GP.

(source: [Gulrajani et al., 2017])

Generative Adversarial Networks (GAN)
oooooooooooo

Wasserstein GAN (WGAN)
oooooooooooo

Large-Scale GAN Training
●oooooooooooooooooooo

Plan

Generative Adversarial Networks (GAN)

Wasserstein GAN (WGAN)

Large-Scale GAN Training

Popular Image Databases

- MNIST (digits): $60k$ images with 28^2 px (10 classes)
- Fashion-MNIST (clothes): $70k$ images with 28^2 px (10 classes)
- CIFAR-10: $60k$ images with 32^2 px (10 classes)
- ImageNet: $\approx 1430k$ images of various size (1000 classes)
- CelebA: $\approx 200k$ images with 178×278 px
- CelebA-HQ: $\approx 30k$ images with 1024^2 px
- LSUN (Bedroom/Cat/Churches/...): $\approx 100k$ or $1M$ images with 256^2 px
- FFHQ (or FFHQ-U): $70k$ images with 1024^2 px
- LAION-5B: $5.85B$ images of various size

Neural Network architecture

Generator and discriminator networks can have various layers:

- Fully connected (FC) layers
- Upsampling (interpolation) or Subsampling (max/average pooling) layers
- Convolution/Transposed convolution (with stride)
- Activation functions: RELU, leakyRELU, sigmoid, tanh, etc
- BatchNorm
- ...

Input noise Z has often uniform distribution $\mathcal{U}([0, 1]^p)$ or Gaussian distribution $\mathcal{N}(0, \text{Id})$.

Convolution

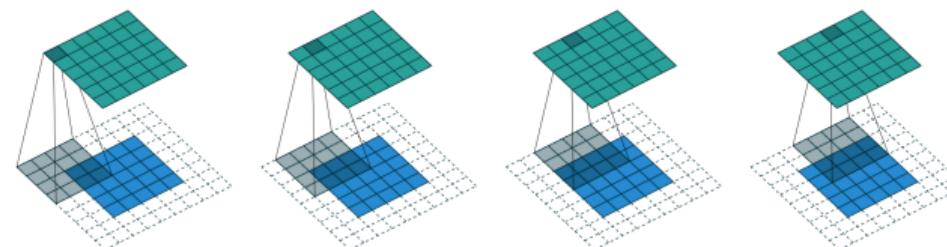
Let $u : \Omega \rightarrow \mathbf{R}^C$ be defined on $\Omega = [0 : M - 1] \times [0 : N - 1]$.

Let $w : \omega \rightarrow \mathbf{R}^{C' \times C}$ be defined on a small $\omega \subset \mathbb{Z}^2$. (Often, $\omega = [-k, k]^2$)

Definition

The convolution $w * u$ of the image u with kernel w is defined by

$$w * u(x) = \sum_{y \in \omega} w(y)u(x-y) = \sum_{z \in -\omega} \tilde{w}(z)u(x+z) \quad \text{where} \quad \tilde{w}(z) = w(-z).$$



NB: There are several possible border conditions (restriction, constant padding, periodic, ...)

Convolution and Transposed convolution

Notice that

- The transpose of a convolution with a $k \times k$ kernel is a convolution with a $k \times k$ kernel
- The transpose of a border crop is zero-padding the borders.
- The transpose of a crude subsampling is zero-inserting.

Strided convolutions:

- A “convolution with stride” is a convolution followed by subsampling.
- Called `Conv2d` in PyTorch

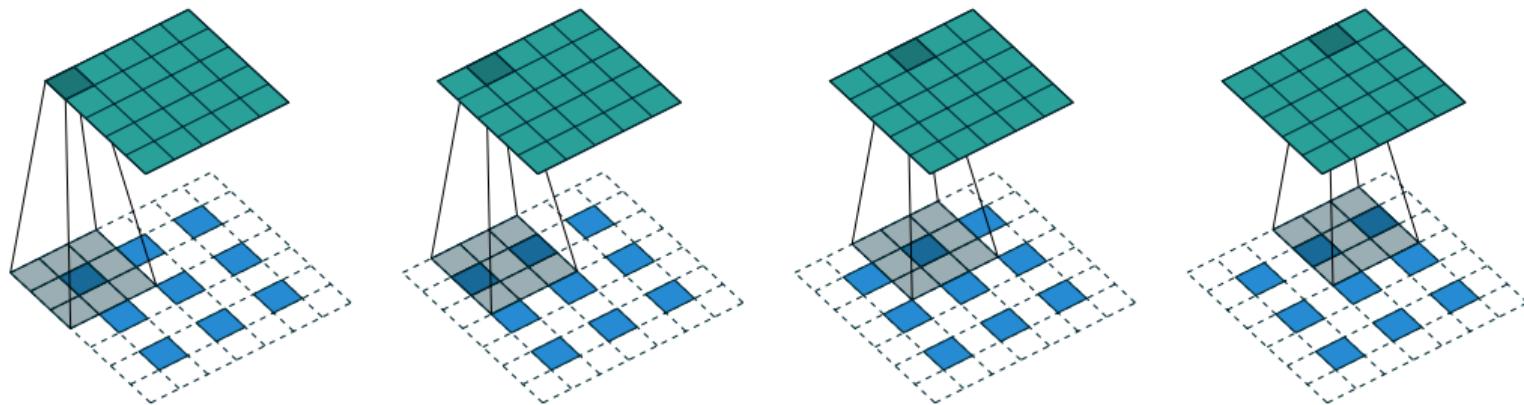
Fractionally strided convolutions:

- This is the transpose operator of convolution with stride.
- Called `ConvTranspose2d` in PyTorch

Useful webpage to compute the size of output after a Conv2D layer:

<https://madebyollin.github.io/convnet-calculator/>

One Example from [Dumoulin and Visin, 2016]



“The transpose of convolving a 3×3 kernel over a 5×5 input padded with a 1×1 border of zeros using 2×2 strides (i.e., $i = 5$, $k = 3$, $s = 2$ and $p = 1$). It is equivalent to convolving a 3×3 kernel over a 3×3 input (with 1 zero inserted between inputs) padded with a 1×1 border of zeros using unit strides (i.e., $i' = 3$, $\tilde{i}' = 5$, $k' = k$, $s' = 1$ and $p' = 1$).”

BatchNorm layer

Principle of BatchNormalization:

- Consider a batch $(x_n)_{1 \leq n \leq N}$ of N responses to a neural layer with C features.
- For each n , $x_{n,i} \in \mathbf{R}^{W \times H}$ is the i -th feature map of the n -th image.
- Batch normalization consists in computing for any n, i

$$y_{n,i} = \gamma_i z_{n,i} + \beta_i \quad \text{with} \quad z_{n,i} = \frac{x_{n,i} - m_i}{\sqrt{\sigma_i^2 + \varepsilon}}$$

where m_i, σ_i are the mean and std of the gathered feature maps $(x_{n,i})_{1 \leq n \leq N}$.
(In other words, m_i, σ_i contains averages over N and spatial dimensions H, W .)

- γ_i, β_i are trainable parameters.
- Implemented in `BatchNorm2d` in PyTorch.

At inference: normalization is done with $m_i, \sigma_i, \gamma_i, \beta_i$ learned during training.
Switch to inference mode with `model.eval()`.

Different Kinds of Normalization

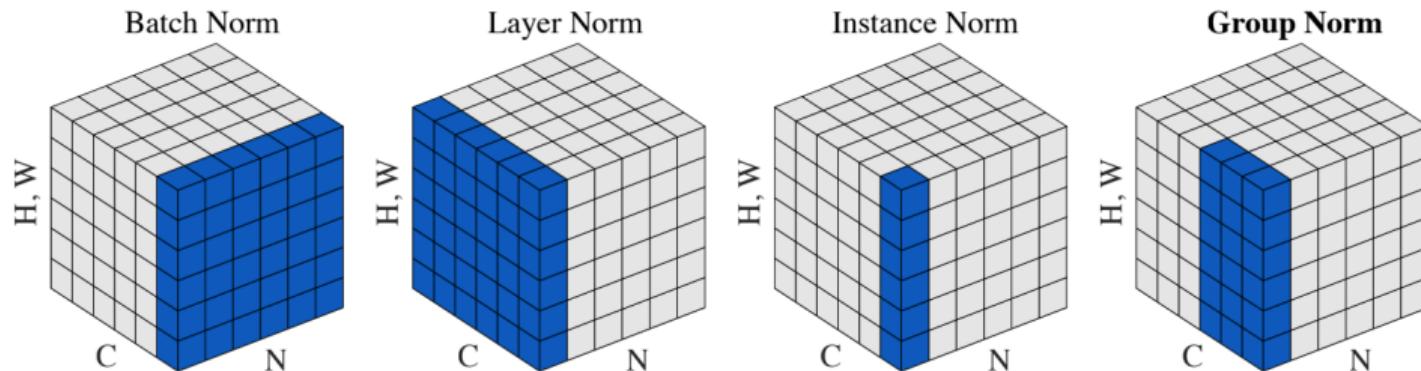


Diagram from [Wu and He, 2018]

- H, W : spatial dimensions
- C : channel dimension
- N : batch dimension

(See the formula for InstanceNorm in [Ulyanov et al., 2017])

Convolutional GAN

[Radford et al., 2016]

Important principles of the construction:

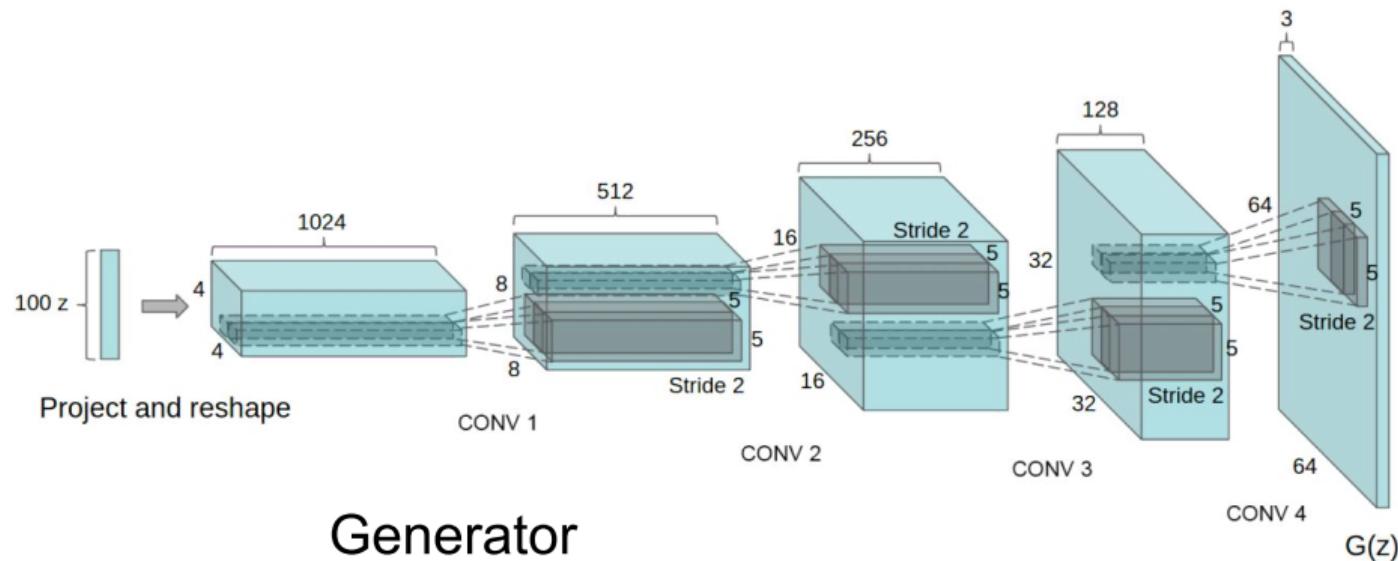
- “All convolutional”: remove max pooling layers, and learn downsampling instead
- Eliminate Fully-Connected Layers
- Batch Normalization to stabilize learning (except on generator output, and discriminator input)
- ReLU activations for the generator
- LeakyReLU activations for the discriminator

Generator: upsampling network with **fractionally strided convolutions**

Discriminator: convolutional network with **strided convolutions**

DCGAN Architecture

[Radford et al., 2016]



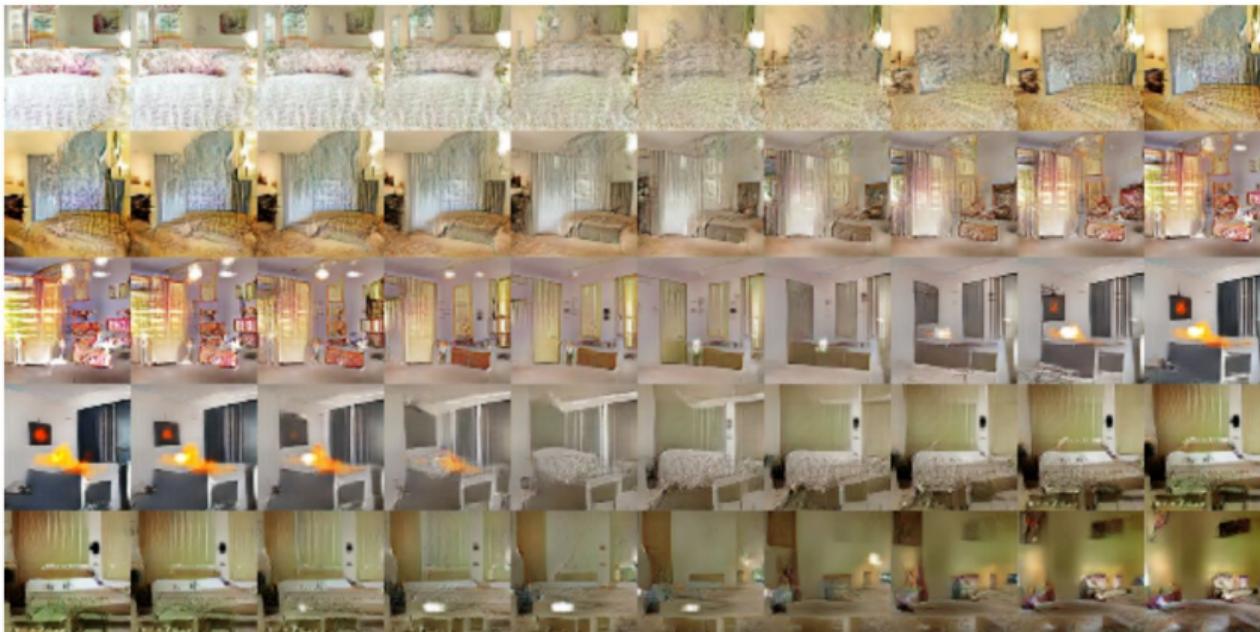
Generator

Image Generation with DCGAN [Radford et al., 2016]



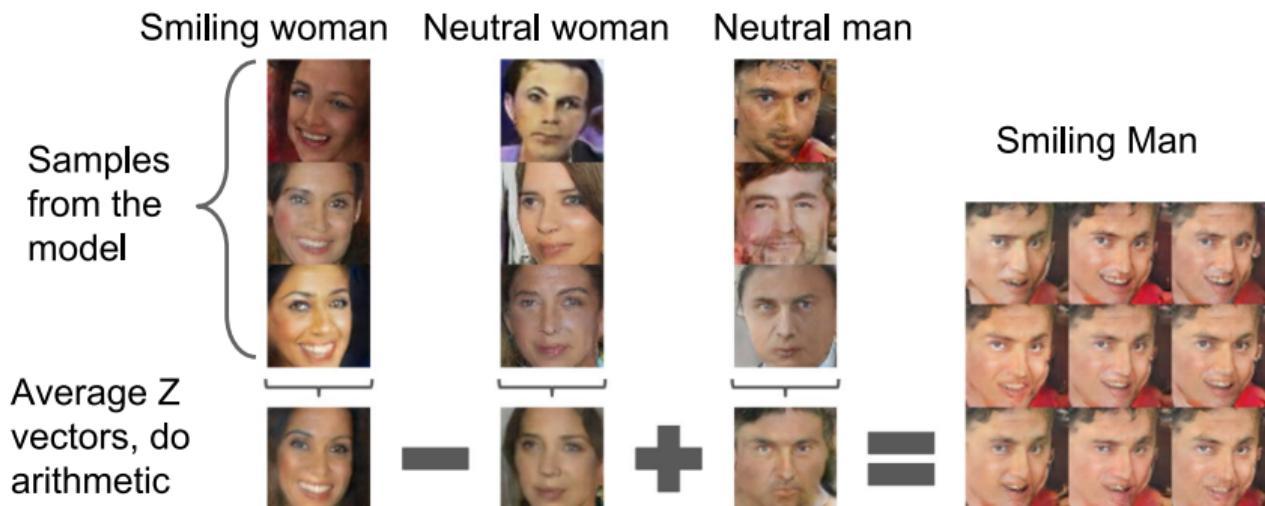
Generations of realistic bedrooms pictures, from randomly generated latent variables.

Image Interpolation with DCGAN [Radford et al., 2016]



Interpolation in between points in latent space.

Arithmetic with DCGAN [Radford et al., 2016]



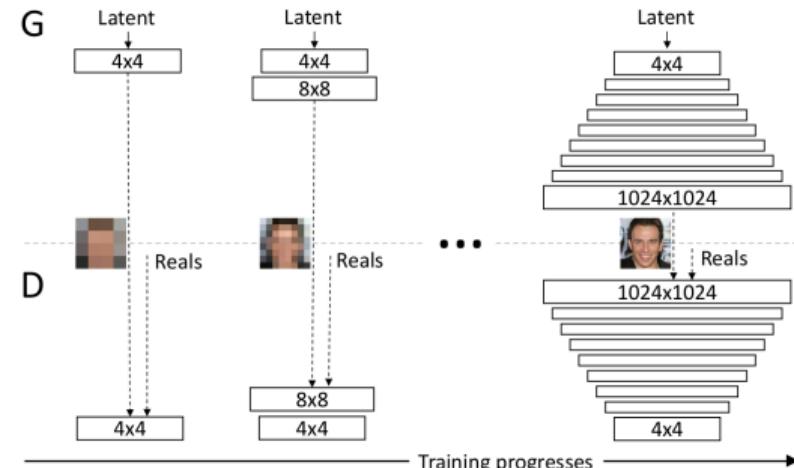
- Average latent vector of several samples
- After arithmetic, add a small random perturbation to get similar samples

Arithmetic with DCGAN [Radford et al., 2016]



Progressive Growing of GANs [Karras et al., 2018]

- Progressive Multiresolution Training
- Mirror architectures for G and D
- Simple upsampling/downsampling
nearest neighbor upsampling;
average pooling downsampling
- Minibatch statistics layer at the end of D
- Pixelwise feature normalization
- Training with WGAN-GP



StyleGAN [Karras et al., 2019]

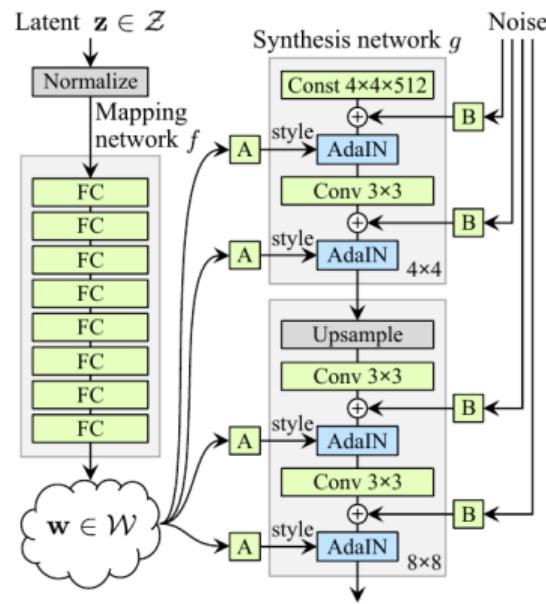
- “Separation of high-level features (pose, identity) from stochastic variation (freckles, hair)”
- Embed latent code z into an intermediate latent space w with a multilayer perceptron (8 FC layers)
- Spatially invariant style vector $y = (y_s, y_b)$ for each feature map, obtained from w
- AdaIN: Adaptive Instance Normalization

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

where the feature map x_i is normalized separately.
(No learned parameters γ, β here.)

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

- Style mixing (playing with two latent codes w_1, w_2)



(b) Style-based generator

StyleGAN [Karras et al., 2019]

- “Separation of high-level features (pose, identity) from stochastic variation (freckles, hair)”
- Embed latent code z into an intermediate latent space w with a multilayer perceptron (8 FC layers)
- Spatially invariant style vector $y = (y_s, y_b)$ for each feature map, obtained from w
- AdaIN: Adaptive Instance Normalization

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

where the feature map x_i is normalized separately.
(No learned parameters γ, β here.)

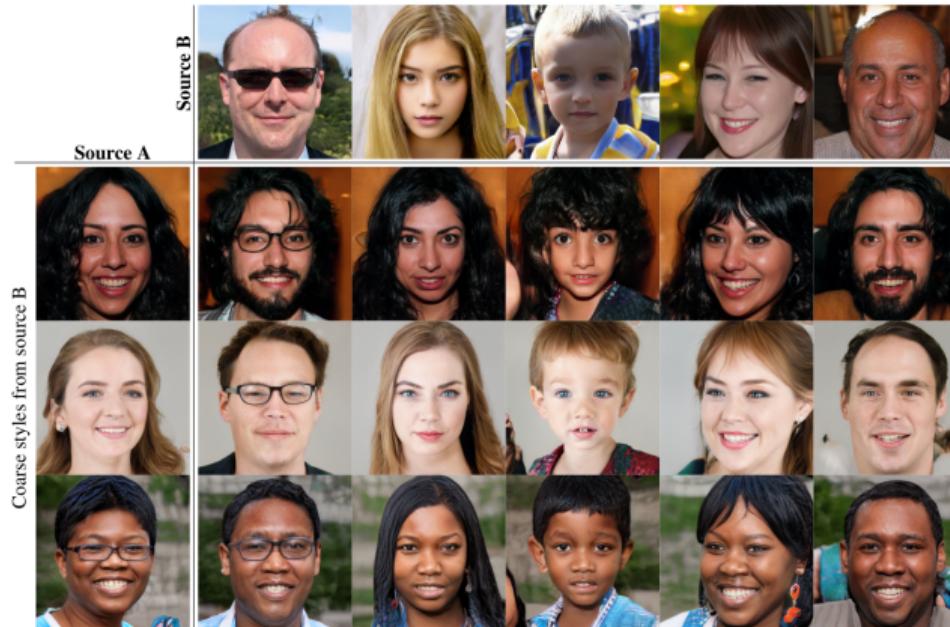
$$\text{AdaIN}(x_{n,i}, y) = y_{s,i} \frac{x_{n,i} - \mu(x_{n,i})}{\sigma(x_{n,i})} + y_{b,i}$$

- Style mixing (playing with two latent codes w_1, w_2)



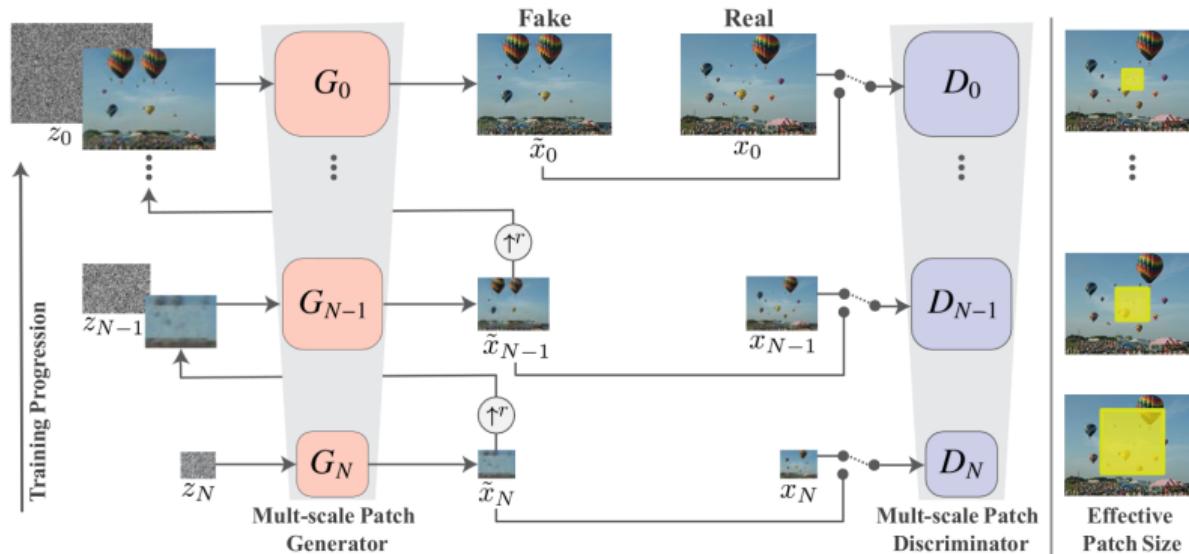
StyleGAN [Karras et al., 2019]

StyleGAN allows for style mixing at different scales (by using the corresponding subpart of w).



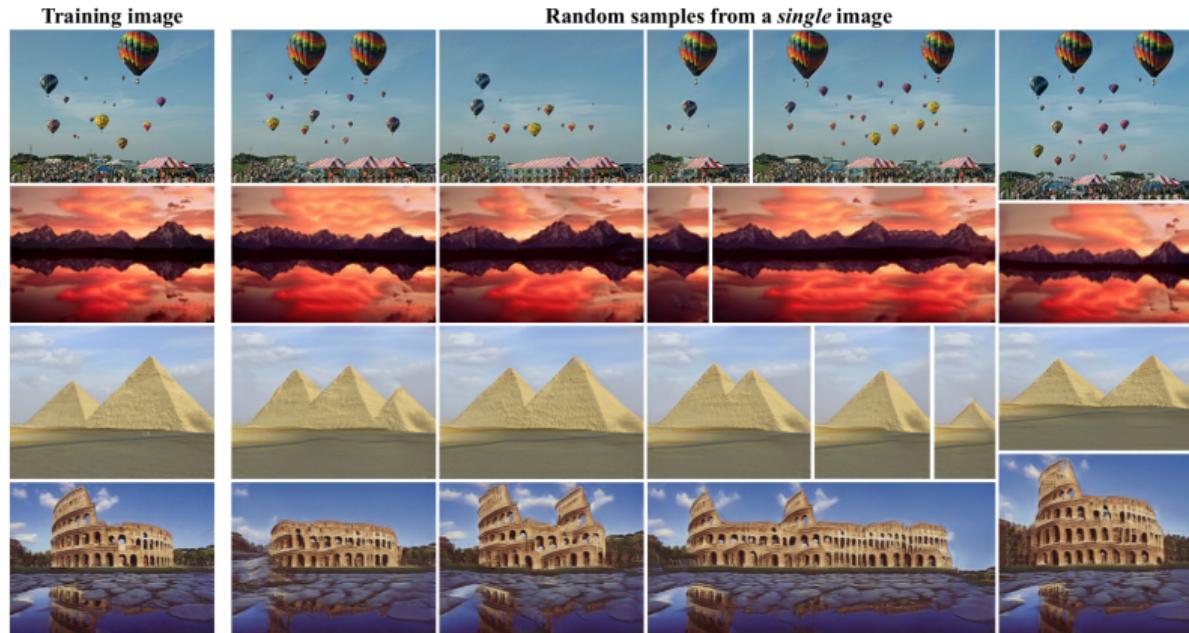
SinGAN: Learning from a Single Image [Shaham et al., 2019]

- Capture the multi-scale patch distributions of an image (possibly non-texture)
- Coarse-to-fine generator
- Patch-based discriminator learned with WGAN-GP loss, at each scale



SinGAN: Learning from a Single Image [Shaham et al., 2019]

- Capture the multi-scale patch distributions of an image (possibly non-texture)
- Coarse-to-fine generator
- Patch-based dicriminator learned with WGAN-GP loss, at each scale



Take-home Messages

SUMMARY AND COMMENTS:

- We introduced GANs and Wasserstein GANs.
- Some constraints (Lipschitz) help to make training more stable.
- Results still depend on the generator/discriminator architectures and the optimization strategy.
- GANs can be used for image synthesis, with convolutional architectures.
- Assessing the quality of a trained generative network requires well-designed quantitative measures (see Appendix).
- Very large scale image synthesis requires tailored network architectures.

THANK YOU FOR YOUR ATTENTION!

References I

-  Arjovsky, M., Chintala, S., and Bottou, L. (2017).
Wasserstein generative adversarial networks.
In *International Conference on Machine Learning*, pages 214–223.
-  Biau, G., Cadre, B., Sangnier, M., and Tanielian, U. (2018).
Some theoretical properties of gans.
arXiv preprint arXiv:1803.07819.
-  Dowson, D. C. and Landau, B. V. (1982).
The fréchet distance between multivariate normal distributions.
Journal of Multivariate Analysis, 12(3):450–455.
-  Dumoulin, V. and Visin, F. (2016).
A guide to convolution arithmetic for deep learning.
ArXiv e-prints.
-  Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. (2014).
Generative adversarial nets.
In *Advances in neural information processing systems*, pages 2672–2680.

References II

-  Gulrajani, I., Ahmed, F., Arjovsky, M., Dumoulin, V., and Courville, A. (2017). Improved training of Wasserstein gans.
In *Advances in neural information processing systems*, pages 5767–5777.
-  Heusel, M., Ramsauer, H., Unterthiner, T., Nessler, B., and Hochreiter, S. (2017). Gans trained by a two time-scale update rule converge to a local Nash equilibrium.
In Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R., editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc.
-  Karras, T., Aila, T., Laine, S., and Lehtinen, J. (2018). Progressive Growing of GANs for Improved Quality, Stability, and Variation.
In *Proceedings of International Conference on Learning Representations*.
-  Karras, T., Laine, S., and Aila, T. (2019). A style-based generator architecture for generative adversarial networks.
In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4401–4410.

References III

-  Radford, A., Metz, L., and Chintala, S. (2016).
Unsupervised representation learning with deep convolutional generative adversarial networks.
In Bengio, Y. and LeCun, Y., editors, *Proceedings of ICLR*.
-  Salimans, T., Goodfellow, I., Zaremba, W., Cheung, V., Radford, A., and Chen, X. (2016).
Improved techniques for training gans.
Advances in neural information processing systems, 29.
-  Santambrogio, F. (2015).
Optimal transport for applied mathematicians.
Birkhäuser, NY.
-  Shaham, T. R., Dekel, T., and Michaeli, T. (2019).
SinGAN: Learning a Generative Model from a Single Natural Image.
In *Proceedings of the IEEE International Conference on Computer Vision*, pages 4570–4580.
-  Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., and Wojna, Z. (2016).
Rethinking the inception architecture for computer vision.
In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

References IV

-  [Ulyanov, D., Vedaldi, A., and Lempitsky, V. \(2017\).](#)
Improved texture networks: Maximizing quality and diversity in feed-forward stylization and texture synthesis.
In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 6924–6932.
-  [Wu, Y. and He, K. \(2018\).](#)
Group normalization.
In *Proceedings of the European conference on computer vision (ECCV)*, pages 3–19.

BatchNorm layer

Principle of BatchNormalization: for any batch $(x_i)_{i \in B}$ of a K -dimensional feature, transform

$$\forall k = 1, \dots, K, \quad y_i^{(k)} = \gamma^{(k)} \hat{x}_i^{(k)} + \beta^{(k)} \quad \text{with} \quad \hat{x}_i^{(k)} = \frac{x_i^{(k)} - m_i^{(k)}}{\sqrt{(\sigma_i^{(k)})^2 + \varepsilon}}$$

where $m_i^{(k)}, \sigma_i^{(k)}$ are mean and std of the k -feature over this batch,
and where $\gamma^{(k)}, \beta^{(k)}$ are trainable parameters.

At inference: normalization is done with $m^{(k)}, \sigma^{(k)}, \gamma^{(k)}, \beta^{(k)}$ learned during training.
Switch to inference mode with `model.eval()`.

Inception Score ↑ [Salimans et al., 2016]

- The inception score measures if μ generates a diverse collection of meaningful pictures
- For an image x , Inception-v3 gives a label distribution $p(y|x)$ (discrete on $N = 1000$ labels)
- Images containing meaningful objects have $p(y|x)$ with low entropy
- In order to generate various images, $p(y) = \int p(y|x)\mu(dx)$ should have high entropy

The Inception Score then writes as

$$\text{IS}(\mu) = \exp \left(\int \text{KL} \left(p(y|x) \| p(y) \right) \mu(dx) \right) \in [1, N]$$

It is 1 iff for a.e. x , $p(\cdot|x) = p(\cdot)$ (label distribution does not depend on x)

It is N iff for a.e. x , $p(\cdot|x)$ is concentrated on one label, and $\forall y$, $\int p(y|x)\mu(dx) = \frac{1}{N}$

How to compute it in practice:

- Compute an estimate $\hat{p}(y)$ of $p(y) = \int p(y|x)\mu(dx)$ by drawing samples of μ
- Estimate $\int \text{KL}(p(y|x)|\hat{p}(y))\mu(dx)$ by drawing samples of μ

Fréchet Inception Distance (FID) ↓ [Heusel et al., 2017]

The FID measures how close are two image distributions μ, ν in terms of features distributions. It is based on the response of Inception-v3 [Szegedy et al., 2016] before last pooling layer:

$$f : \mathbf{R}^d \rightarrow \mathbf{R}^m$$

that extracts $m = 2048$ features (as a generic image summary)

NB: Images may have to be resized/normalized to fit into this network.

Algorithm to compute the FID score:

1. Draw samples (x_i) and (y_j) of $X \sim \mu$ and $Y \sim \nu$ and compute the features $(f(x_i)), (f(y_j))$
2. Fit Gaussian distributions $\mathcal{N}(m_X, \Sigma_X)$ and $\mathcal{N}(m_Y, \Sigma_Y)$ to $(f(x_i)), (f(y_j))$ (in \mathbf{R}^{2048})
3. Return the 2-Wasserstein distance between the Gaussian distributions,
i.e. the Fréchet distance: [Dowson and Landau, 1982]

$$W_2^2\left(\mathcal{N}(m_X, \Sigma_X), \mathcal{N}(m_Y, \Sigma_Y)\right) = \|m_X - m_Y\|_2^2 + \text{Tr}\left(\Sigma_X + \Sigma_Y - 2(\Sigma_X \Sigma_Y)^{\frac{1}{2}}\right)$$

NB: FID can be adapted to the “single-image” case: **SiFID** [Shaham et al., 2019]

SiFID compares distributions of features obtained after a convolution layer (spatially averaged)