Generative Adversarial Networks (GAN)

Alexander Binder and Ghadi Al Hajj

University of Oslo (UiO)

April 26, 2022



UiO : Department of Informatics
University of Oslo

Learning goals

- Explain the key components of a classic GAN: generator and discriminator
- Derive and analyze their objectives
- Understand what mode collapse is
- Explain the Wasserstein loss
- Understand evaluation heuristics of GAN outputs

Motivation and Intuition

Game:

- Person A wants to learn how to replicate existing artwork (paintings/drawings),
- Person B wants to learn how to tell apart fake from original artwork.

Rules:

- Person B has access to real and fake artwork
- Person A has access to how the second person makes her decision about the originality of a painting
- At the beginning, neither of the two is good at her task

Game:

- Person A wants to learn how to replicate existing artwork (paintings/drawings),
- Person B wants to learn how to tell apart fake from original artwork.

Rules:

- Person B has access to real and fake artwork
- Person A has access to how the second person makes her decision about the originality of a painting
- At the beginning, neither of the two is good at her task

Game:

- Person A wants to learn how to replicate existing artwork (paintings/drawings),
- Person B wants to learn how to tell apart fake from original artwork.

Rules:

- Person B has access to real and fake artwork
- Person A has access to how the second person makes her decision about the originality of a painting
- At the beginning, neither of the two is good at her task

Game:

- Person A wants to learn how to replicate existing artwork (paintings/drawings),
- Person B wants to learn how to tell apart fake from original artwork.

Rules:

- Person B has access to real and fake artwork
- Person A has access to how the second person makes her decision about the originality of a painting
- At the beginning, neither of the two is good at her task

Game:

- Person A wants to learn how to replicate existing artwork (paintings/drawings),
- Person B wants to learn how to tell apart fake from original artwork.

Rules:

- Person B has access to real and fake artwork
- Person A has access to how the second person makes her decision about the originality of a painting
- At the beginning, neither of the two is good at her task

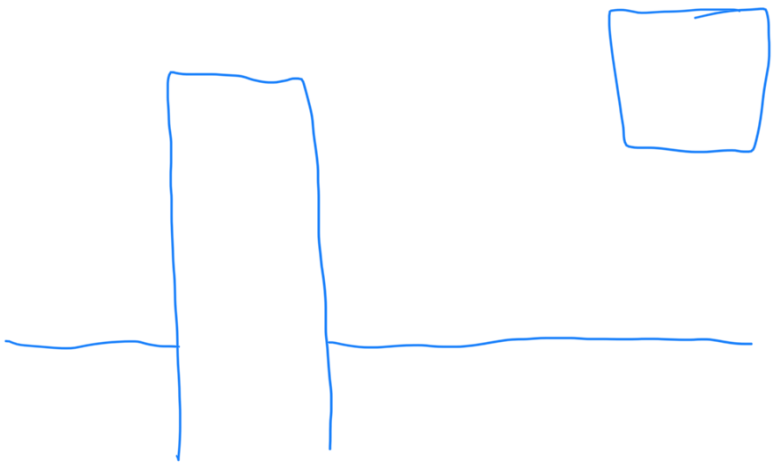


Figure: 1st attempt



Figure: Original drawing

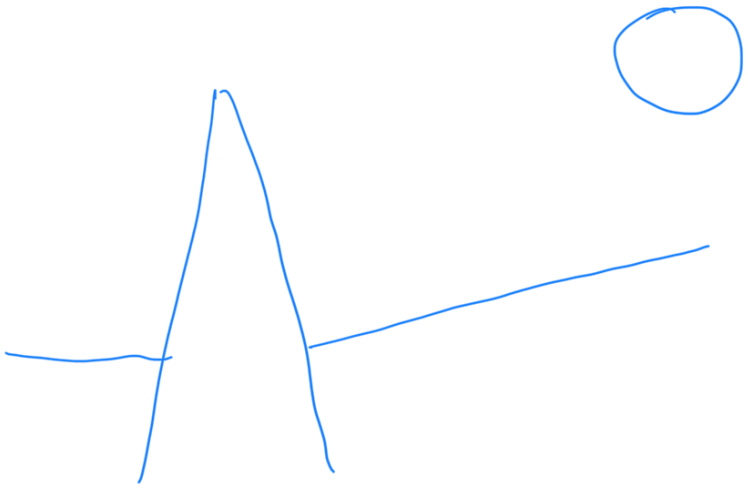


Figure: 2nd attempt



Figure: Original drawing

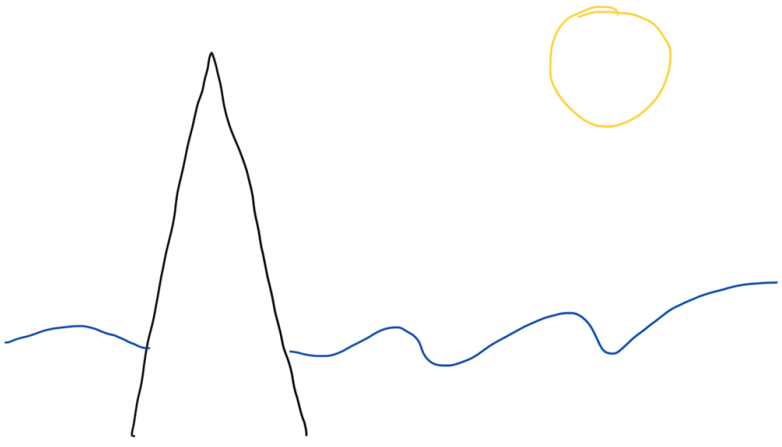


Figure: 3rd attempt



Figure: Original drawing

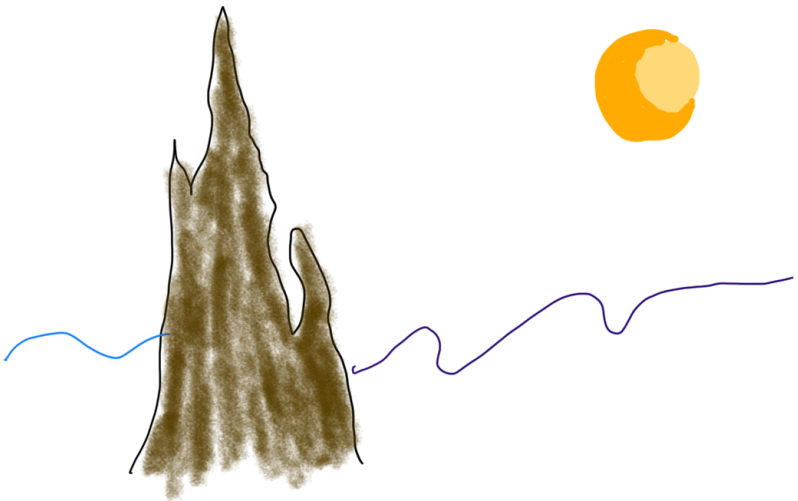


Figure: 4th attempt



Figure: Original drawing

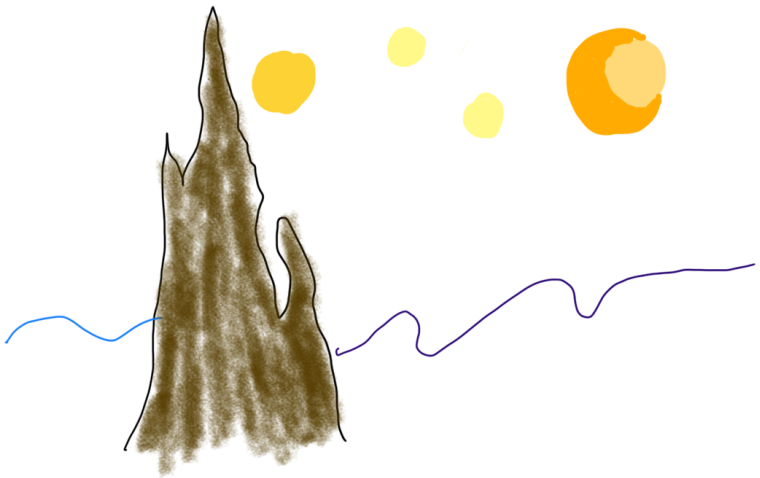


Figure: 5th attempt



Figure: Original drawing



Figure: 6th attempt



Figure: Original drawing



Figure: nth attempt



Figure: Original drawing

Question:

- How do we represent the counterfeiter and the detective in mathematical terms?
- In other words, what mathematical functions can we utilize to represent each role?

Questions:

- How do we represent the counterfeiter and the detective in mathematical terms?
- In other words, what mathematical functions can we utilize to replace each role?

As common in Deep Learning, whenever you don't know the shape, i.e. the functional form, of the function you are want to approximate $\rightarrow$ use an appropriate Neural Network.

Questions:

- How do we represent the counterfeiter and the detective in mathematical terms?
- In other words, what mathematical functions can we utilize to replace each role?

As common in Deep Learning, whenever you don't know the shape, i.e. the functional form, of the function you are want to approximate $\rightarrow$ use an appropriate Neural Network.

- For the "counterfeiter" use a CNN that generates images from noise vectors, i.e. increases the dimensions of an input $\rightarrow$ *Generator*
- For the detective use a CNN that classifies input samples (images in our case) into real vs fake $\rightarrow$ *Discriminator*

Questions:

- How do we represent the counterfeiter and the detective in mathematical terms?
- In other words, what mathematical functions can we utilize to replace each role?

As common in Deep Learning, whenever you don't know the shape, i.e. the functional form, of the function you are want to approximate → use an appropriate Neural Network.

- For the "counterfeiter" use a CNN that generates images from noise vectors, i.e. increases the dimensions of an input → *Generator*
- For the detective use a CNN that classifies input samples (images in our case) into real vs fake → *Discriminator*

Formally: Given a dataset $D_n = \{x_1, \ldots, x_n\}$, for instance images x_i, generate *similar* images to those in the dataset.

The objective for a GAN is twofold:

- Learn a model G that can generate new samples (images in our example), using a randomly generated noise vector as the model input

  $z \sim N(0, I_d)$ with d the number of input dimensions.

- Training of a GAN G should lead to some kind of *similarity* between the samples in D_n and the samples x generated by the GAN, namely

  $$x = G(z)$$

The important keywords in the previous description were *"similar"* and *"similarity"*.

But what do we mean by *"similar"* images?

- Let P_{data} be the distribution from which the samples D_n are sampled
- Let P_{GAN} be the distribution of the images generated by the generator after being trained, i.e. $x \sim P_{GAN}$, where $x = G(z)$ (x is a random variable now)

We need a similarity metric, which measures how similar the samples in both sets P_{data} and P_{GAN} are, where the aim is to make $P_{GAN} \approx P_{data}$.

(Note that we don't have direct access to P_{GAN}, and also not to P_{data}. We can only sample from them)

One solution: we use a discriminator that classifies real vs fake samples

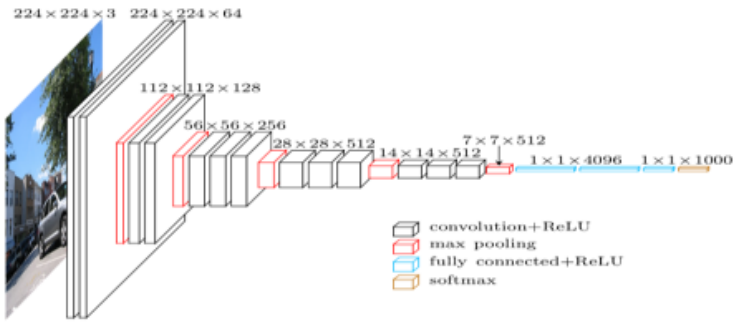
Classic GAN: Generator

Model definition - Generator of a GAN:

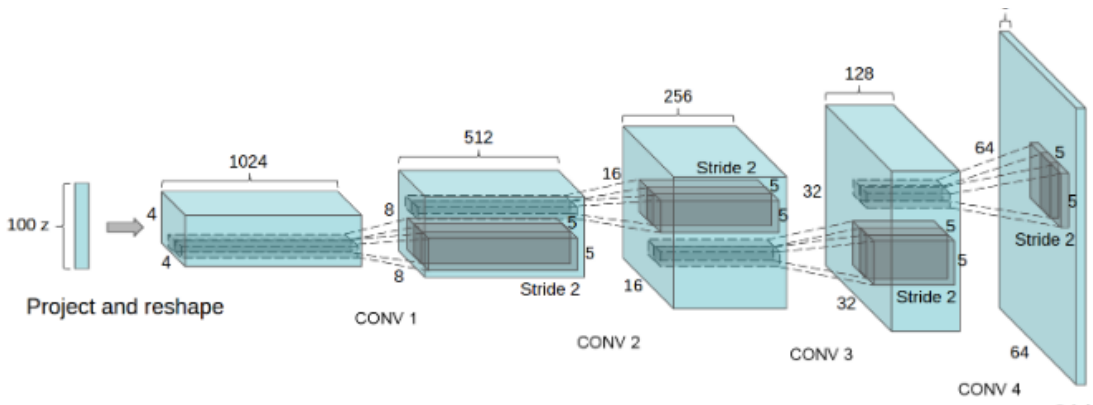
- Input space: vector space $\mathbb{R}^d$, i.e. a vector with d elements,
- The input vectors z will be randomly sampled from a high dimensional random distribution, usually a normal distribution $\mathbb{R}^d \ni z \sim N(0, \sigma^2 I_d)$,
- The GAN outputs, $x = G(z)$, consist of images of some size, e.g. 256×256, but in practice outputs could be any types of objects.

Next: how to map a vector of e.g. 1000 dims onto an image e.g. 256×256?

- In a conventional CNN models (e.g. VGG by Simonyan et al.):
  Every n layers, the model reduces, e.g. halves, the image resolution and increases, e.g. doubles, the number of channels in feature maps.



- GANs generator models (e.g. DCGAN by Radford et al.):
  Every n layers, the model increases the resolution progressively and, if needed, decreases the number of channels in the feature maps.

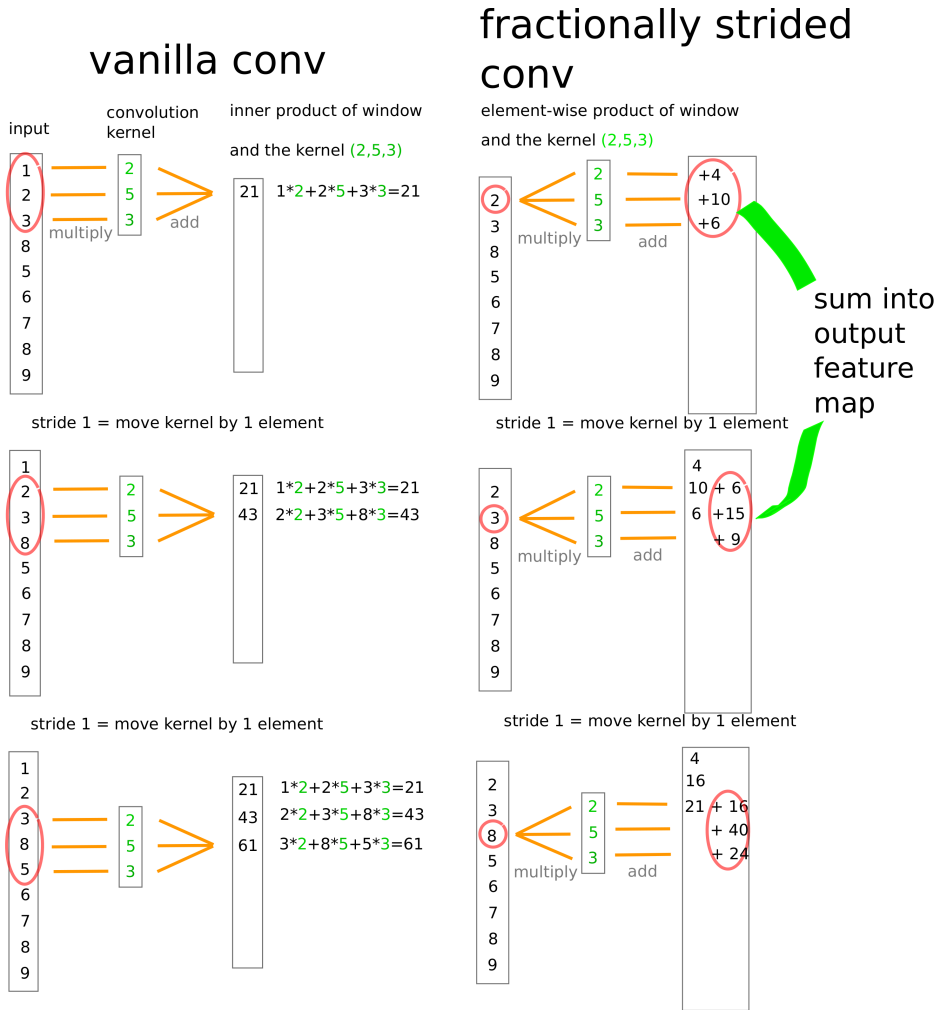


But how to do that?

Option 1: Up-sampling (i.e. interpolation) followed by convolutions.

Option 2: *Fractionally strided convolution*: similar to the convolution operation, but in reverse

For single input channels, the working principle of a fractionally strided convolution is described below.



Classic GAN: Discriminator (generator quality measure)

Definition GAN discriminator for a classic GAN:

A GAN discriminator $D(x)$ is a neural network, which takes sets of images as inputs.

Each set contains either real images or images sampled from the GAN generator. The discriminator is a simple binary classifier, with one output, encoding whether images are real and coming from the training dataset, or fake.

Let denote $D(x) \in [0, 1]$ the probability that a given input sample x is real. $D(x) = 1$ means that the discriminator is sure with probability 1 that x is real.

$D(x)$ is trained to label the real images as class 1 (real), and the ones generated by the GAN generator as fake (0).

Next: how to define a loss using this.

Assume that for now we have a pretrained and fixed discriminator.

The objective for the GAN generator would then be to create images x, which will be (incorrectly) classified as real by the discriminator. Discriminator outputs probabilities $D \in [0, 1]$. The objective is to train G, such that $D(G(z)) \to 1$.

Original Formulation:

$$\min_G \ln[\mathbf{1} - \boldsymbol{D}(\boldsymbol{G}(\boldsymbol{z}))]$$

In other words, find G that minimizes **the probability of an image being fake** .

$$\hat{L}_G = \min_G \frac{1}{k} \sum_{z_i \in minibatch(N(0,1))^k} \ln[1 - D(G(z))]$$

$$L_G = \min_G E_{z \sim N(0,1)} \ln[1 - D(G(z))]$$

- In the first equation, we define the approximation of the expectation, using one minibatch. In the second, we have the general definition of the objective function we want to optimize.
- While this loss function is used to train G, a fixed discriminator is not used because it leads to one central problem of GAN training. What do you think it is?

$$\hat{L}_G = \min_G \frac{1}{k} \sum_{z_i \in minibatch(N(0,1))^k} \ln[1 - D(G(z))]$$

$$L_G = \min_G E_{z \sim N(0,1)} \ln[1 - D(G(z))]$$

- In the first equation, we define the approximation of the expectation, using one minibatch. In the second, we have the general definition of the objective function we want to optimize.
- While this loss function is used to train G, a fixed discriminator is not used because it leads to one central problem of GAN training. What do you think it is?

Training the generator with a fixed discriminator, will usually result in outputting small variations of one single image x', which fools the discriminator.



from Arjovsky et al. `https://arxiv.org/pdf/1701.07875.pdf`

The generator will learn to output very minor variations of a few images $x' = G(z)$ for any input z,because these images x' allows to minimize L_G well.

mode collapse:

- typical observation: generates only a few modes as compared to training dataset, low variance of generated samples within a mode $\rightarrow$ learned to approximate P only over a subset



from Arjovsky et al. `https://arxiv.org/pdf/1701.07875.pdf`

- measurement? see evaluation measures later on

Mode collapse formally

The generator P_{GAN} covers only a small part of the space represented by the the test distribution P_{test}

$$\exists \epsilon > 0,\ A \subset \mathbb{R}^d :\ P_{GAN}(A) \ll P_{test}(A) \text{ and } P_{test}(A) \geq \epsilon$$

See Fig 3 in
`http://proceedings.mlr.press/v80/bang18a/bang18a.pdf`
for a simple example with multiple modes.

Generator Quality vs Generator Diversity tradeoff

There is a performance tradeoff for the GANs generators: sample quality versus sample diversity. They can come in conflict with each other, for example when one would use a fixed discriminator and no additional tricks on the generator.

One (partial) solution

- Train a discriminator and generator jointly – in interleaved steps, from random initializations. coarse idea:
  - fix discriminator, update generator for example using $\hat{L}_G$
  - fix generator, update discriminator using a loss to be defined $\hat{L}_D$

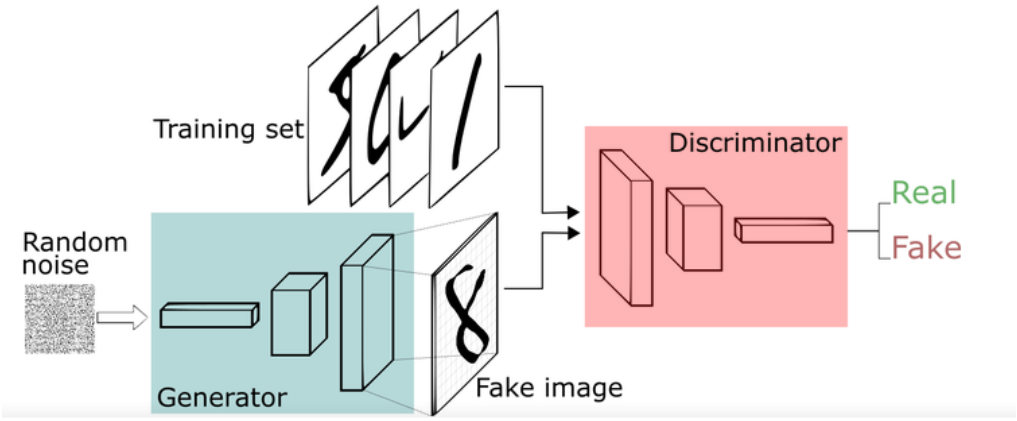


image from Thalles Silva `https://medium.freecodecamp.org/an-intuitive-introduction-to-generative-adversarial-networks-gans-`

- Goal: $x_i \sim D_n$ should be classified by the discriminator $D(\cdot)$ as coming from D_n, thus $D(x_i) \approx 1$, and $G(z)$ should be classified by the discriminator $D(\cdot)$ as **not** coming from D_n, thus $D(G(z)) \approx 0$.

$$\begin{aligned}
&\Rightarrow D(x_i) && \approx 1,\ x_i \in D_n && \text{and } 1 - D(G(z)) && \approx 1 \\
&\Rightarrow -\ln D(x_i) && \approx 0,\ x_i \in D_n && \text{and } -\ln(1 - D(G(z))) && \approx 0
\end{aligned}$$

$$\Rightarrow L_D = E_{x \sim P_{data}}[-\ln(D(x))] + E_{z \sim P_z}[-\ln(1 - D(G(z)))]$$

$$\approx \hat{L}_D = \frac{1}{b} \sum_{x_i \sim D_n}^{b} \underbrace{-\ln D(x_i)}_{x_i \text{ is real}, D(x_i) \approx 1} + \frac{1}{b} \sum_{z_i \sim P_z}^{b} \underbrace{-\ln(1 - D(G(z_i)))}_{G(z_i) \text{ is synth}, 1 - D(G(z_i)) \approx 1}$$

In this original formulation, we used

$$L_G = \min_G E_{z \sim N(0,1)} \ln[1 - D(G(z))]$$

However, when applied, this loss may not provide sufficient gradient for G to learn well. Why?
Early in learning, when G is poor, D can reject samples with high confidence, i.e. classify the image as fake.
In this case,

$$D(G(z)) \approx 0 \implies ln(1-D(G(z))) = ln(\sim 1) \approx 0 \implies \nabla L_G \approx 0$$

What can we do?

Rather than minimizing $ln(1 - D(G(z)))$ we maximize $lnD(G(z))$, i.e. maximizes the probability of the image being real. This way,

$$D(G(z)) \approx 0 \implies -ln(D(G(z))) \approx -(-ve) \implies \nabla L_G \neq 0$$

In summary, we use:

$$\hat{L}_D = \frac{1}{b} \sum_{x_i \sim D_n}^{b} -\ln D(x_i) + \frac{1}{b} \sum_{z_i \sim P_z}^{b} -\ln(1 - D(G(z_i)))$$

$$\hat{L}_G = \frac{1}{b} \sum_{z_i \sim P_z}^{b} -\ln D(G(z_i))$$

Non-saturating GAN

A GAN using $\hat{L}_G$ as above and this $\hat{L}_D$ is called non-saturating GAN

Overview over other loss functions (dont need to memorize them): `https://arxiv.org/pdf/1711.10337.pdf`

Vanilla GAN training (Interleaved training)

- Step 1: train the discriminator, with a fixed generator.
  - Fix generator weights, generate fake samples, label them as class 0 (fake class)
  - Train discriminator weights by minibatches, alternating between fake (generated by the generator, labeled as 0) and real samples (drawn from your dataset D_n, labeled as 1). The loss for the discriminator simply consists of a classification accuracy loss (e.g. cross-entropy).
- Step 2: train the generator, with a fixed discriminator.
  - Fix discriminator weights, and train generator weights so as to fool the discriminator, as detailed in the previous section.
- Repeat!

This consists of a non-cooperative two player game, with two players taking turns:

- The first player, the generator, tries to create images close to real relative to D_n, to fool the discriminator,
- The second player, the discriminator tries to learn to separate the generator fake images from the real images.

GAN at testing time

The generator is a function to sample from a learned probability distribution P_{GAN}:

- draw your input code z from a fixed random distribution such as $z \sim N(0, 1)$
- generate your sample by $s = G(z)$

$$z \sim N(0, 1), \; s = G(z)$$

The distribution P_{GAN} that was learned by the generator is not explicitly given. You can only sample from it.

A classic GAN skeleton code in PyTorch

- we alternatively update the generator, and the discriminator weights, thus we will need two optimizers `optimizerD` and `optimizerG`!
- prepare your one class training data `dataloader= ...`, send to GPU
- define your generator model, send it to GPU, initialize its weights

```
netG = Generatormodel().to(device)
netG.initweights()
```

- define your discriminator model, send it to GPU, init its weights

```
netD = Discriminatormodel().to(device)
netD.initweights()
```

- define optimizers, e.g. using Adam

```
optimizerD=nn.optim.Adam(netD.parameters(), lr=...  ,
otherparameters= ...)
optimizerG=nn.optim.Adam(netG.parameters(), lr=...  ,
otherparameters= ...)
```

A good tutorial:
`https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html`

```
for batch in dataloader:
  # train discriminator on one minibatch of real data, labeled as 1
  netD.zero_grad() #zero out any accumulated gradients
  output=netD(batch)
  label1 = #ones

  loss_D_data = criterion(output,label1)
  # criterion=-log D(x) can be realized as torch.BCELoss with the right label
  loss_D_data.backward()

  # train discriminator on one minibatch of generator data, labeled as 0
  randcodes= torch.randn(batchsize, ...)
  fakebatch= netG(randcodes)

  output=netD(fakebatch.detach())
  #.detach() here prohibits the gradient from flowing
  # past the fakebatch tensor. Why do we want to stop there?
  # if not detached, then gradient computations flow into fakebatch= netG(randcodes)
  # and a later optimizerG.step() would also update the generator model weights.
  # we only want to optimize netD now, while netG should be frozen.
  # netG.zero_grad() would clear them out, but its waste of recorded computations

  label0 = #zeros
  loss_D_fake = criterion(output,label0)

  # -log (1-D(x)) can be realized as torch.BCELoss with the right label
  loss_D_fake.backward()

  #code continues on next page
```

```
for batch in dataloader:
  #[...]
  # code continues from last page from here loss_D_fake.backward()

  #update weights
  optimizerD.step()

  # train generator
  netG.zero_grad()

  randcodes= torch.randn(batchsize, ...)
  fakebatch= netG(randcodes)
  output=netD(fakebatch) # now the gradient should flow into G

  loss_G=  criterion(output,label1)  # yes label1 !,
  #bcs criterion is to minimize -log D (G(z))

  loss_G.backward() # any here computed gradients for netD
  #are flushed in the next netD.zero_grad() !!! :)
  optimizerG.step()
```

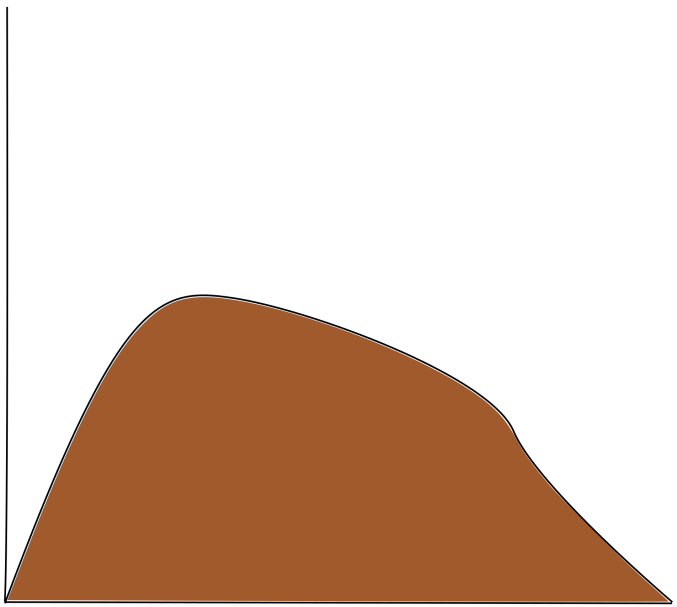
Wasserstein distance – a different loss

- In principle, training GANs is all about trying to make the distribution of generated samples (e.g. images), $P_{G(z)}$ similar to the distribution of the real data, P_{data}
- Therefore, another way to approach GAN training is to minimize the distance between these two distributions.
- While there are, in principle, many candidate distance measures (see Section 2), the Wasserstein distance (also known as the Earth mover's distance) has desirable properties (see Example 1). But what is this Wasserstein distance? and how can we use it for training GANs?

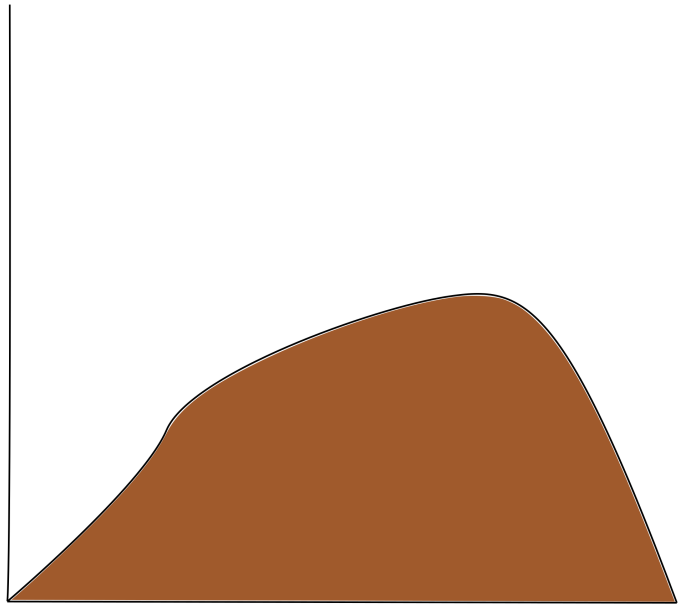
- In principle, training GANs is all about trying to make the distribution of generated samples (e.g. images), $P_{G(z)}$ similar to the distribution of the real data, P_{data}
- Therefore, another way to approach GAN training is to minimize the distance between these two distributions.
- While there are, in principle, many candidate distance measures (see Section 2), the Wasserstein distance (also known as the Earth mover's distance) has desirable properties (see Example 1). But what is this Wasserstein distance? and how can we use it for training GANs?

- In principle, training GANs is all about trying to make the distribution of generated samples (e.g. images), $P_{G(z)}$ similar to the distribution of the real data, P_{data}
- Therefore, another way to approach GAN training is to minimize the distance between these two distributions.
- While there are, in principle, many candidate distance measures (see Section 2), the Wasserstein distance (also known as the Earth mover's distance) has desirable properties (see Example 1). But what is this Wasserstein distance? and how can we use it for training GANs?

Suppose we have two piles of dirt/sand, μ and τ, and we want to make μ look like τ. Intuitively, the cost of moving one bucket of dirt from one position to another is equal to the distance, d, between these positions. The question is what is the set of moving operations that accomplishes this task with the least total cost?

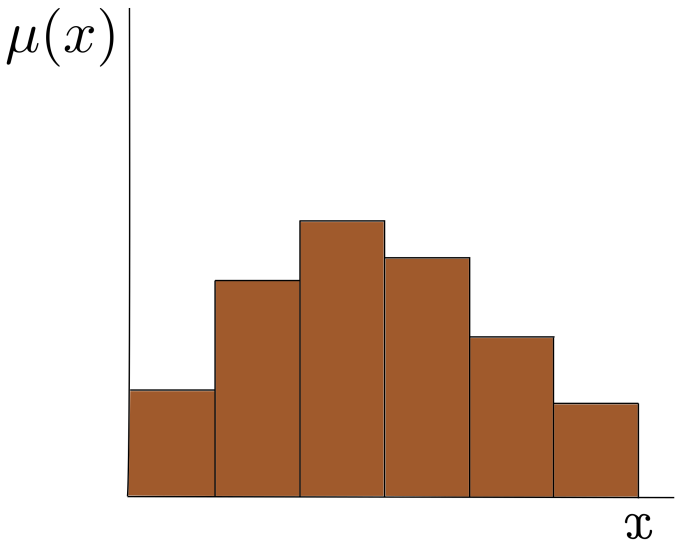


Pile μ

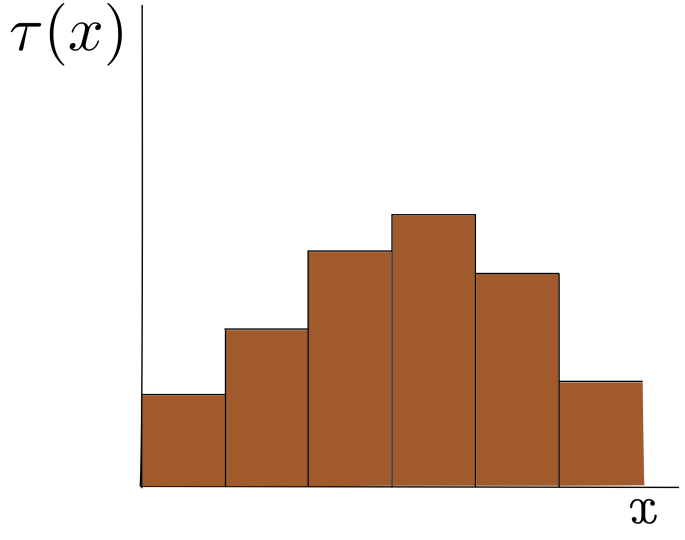


Pile τ

To answer this question, let's simplify matters and discritize our piles. We can also think about these piles as two functions $\mu(x)$ and $\tau(x)$:



Pile 1



Pile 2

Here, $x \in \mathbb{R}^n$ where n is the number of slots, represents the different locations, and $\mu(x_i)$ represents the amount of dirt at position x_i

To formalize the question, let γ be a distribution function such that: $\gamma(x_i, x_k)$ is the **amount redistributed** $x_i \to x_k$. This function will have the following constraints:

- (C1): $\sum_{\text{targets } k} \gamma(x_i, x_k) = \mu(x_i)$, i.e. no spilling of sand
- (C2): $\sum_{\text{sources } i} \gamma(x_i, x_k) =: \tau(x_k)$, i.e. the final amount after distribution will match that of τ.

What is the cost of applying γ?

$$e(\gamma) = \sum_{i=1}^{n} \sum_{k=1}^{n} \gamma(x_i, x_k) d(x_i, x_k)$$

Towards Wasserstein distance: what is the least-cost redistribution $\gamma(x_i, x_k)$ from $\mu(x_i)$ to $\tau(x_k)$? We assume $\sum_i \mu(x_i) = \sum_k \tau(x_k) = 1$
Let be $\Gamma(\mu, \tau)$ the space of all redistributions satisfying (C1) and (C2), then the least cost redistribution is given by

$$\inf_{\gamma \in \Gamma(\mu,\tau)} e(\gamma) = \inf_{\gamma \in \Gamma(\mu,\tau)} \sum_{i=1}^{n} \sum_{k=1}^{n} \gamma(x_i, x_k) d(x_i, x_k)$$

It turns out to be intractable to exhaust all the possible joint distributions in $\Gamma(\mu, \tau)$ to compute the inf.
Here, we can use the Kantorovich-Rubinstein duality theorem (*Optimal Transport: Old and New, Ch 6*)) which states that:

$$\begin{aligned} W_1(\mu, \tau) &= \inf_{\gamma \in \Gamma(\mu,\tau)} \int_{\mathcal{X}\times\mathcal{Y}} d(x,y)\gamma(x,y)dxdy \\ &= \sup_{f \in Lip_1} E_{x\sim\mu}[f(x)] - E_{y\sim\tau}[f(y)] \end{aligned}$$

$$f \in Lip_1 \Leftrightarrow |f(x) - f(y)| \leq 1 \cdot |x - y| \quad \forall x, y$$

But how do we use this loss to train a GAN?

Use $\mu = P_{data} \approx Unif(D_n)$, $\tau = P_{GAN}$

$$W_1(\mu, \tau) = \sup_{f \in Lip_1} E_{x \sim P_{data}}[f(x)] - E_{y \sim P_{GAN}}[f(y)]$$

$$\approx \sup_{f \in Lip_1} \frac{1}{b} \sum_{x_i \sim D_n}^{b} f(x_i) - \frac{1}{b} \sum_{z_i \sim P_z}^{b} f(G(z_i))$$

Here, $f(\cdot) = D(\cdot)$ for the above function f (Remember that D is a function approximator)

Generator loss? We want to minimize the above, thus we optimize for weights $w(G)$ of G:

$$\inf_{w(G)} \frac{1}{b} \sum_{x_i \sim D_n}^{b} f(x_i) - \frac{1}{b} \sum_{v_i \sim P_z}^{b} f(G(z_i))$$

$w(G)$ do not depend on D_n or the first term as a whole, thus (and replacing inf by a finite set min):

The generator loss for a Wasserstein GAN

$$\min_{w(G)} L(G), \; L(G) = -\frac{1}{b} \sum_{z_i \sim \text{input codes}}^{b} f(G(z_i))$$

Discriminator objective?

We want to find the supremum of the term inside:

$$\sup_{D \in Lip_1} \frac{1}{b} \sum_{x_i \sim D_n}^{b} D(x_i) - \frac{1}{b} \sum_{z_i \sim P_z}^{b} D(G(z_i))$$

In other words, we want to maximize it for a given G by changing $D(\cdot)$
If you want a minimum problem, multiply above with -1

The discriminator loss for a Wasserstein GAN

Optimize for weights $w(D)$ for D – note $D(\cdot) = f(\dot{)}$:

$$\min_{w(D):D \in Lip_1} -\frac{1}{b} \sum_{x_i \sim data}^{b} D(x_i) + \frac{1}{b} \sum_{z_i \sim P_z}^{b} D(G(z_i))$$

Steps to obtain a Wasserstein GAN

(A) Write the Wasserstein distance as a difference of two expectations of a function f, maximized over the function f

$$W_1(P_{data}, P_{GAN}) = \max_{f \in Lip_1} E_{x \sim P_{data}}[f(x)] - E_{y \sim P_{GAN}}[f(y)]$$

(B) Replace the expectations $E[\cdot]$ by mini-batch averages, one time over the real data (for $x \sim P_{data}$), one time over the data from the generator (for $y \sim P_{GAN}$)

(C) Represent the function f, the so-called critic, by a neural net.

(D) Train the critic neural net using above max-objective turned into a min (typically with some gradient norm penalty)

(E) Interleave the train-critic-step with the step which optimizes the generator

Problem: need to ensure $f \in Lip_1$. Two options:

(1) Spectral normalization of every trainable convolution weight W `https://arxiv.org/abs/1802.05957` (Miyato et al., ICLR 2018) – (details for this are out of class)

(2) add to $\hat{L}_D$ a gradient penalty term $R(f)$ like WGAN-GP Gulrajani et al. `https://arxiv.org/abs/1704.00028` or WGAN-LP Petzka et al. `https://openreview.net/pdf?id=B1hYRMbCW`

$$R(f) = \frac{1}{n} \sum_{z_i} [(\|\nabla f(z_i)\| - 1)^2] \text{ (WGAN-LP)}$$

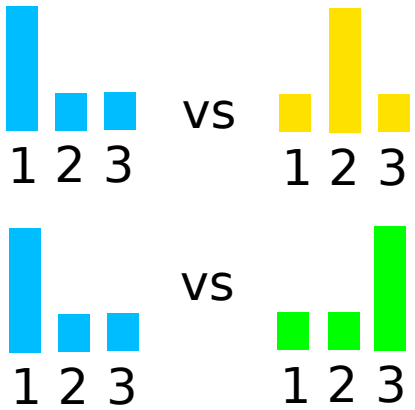
$$R(f) = \frac{1}{n} \sum_{z_i} [max(0, \|\nabla f(z_i)\| - 1)^2] \text{ (WGAN-LP)}$$

$$z_i = tx_i + (1-t)y_i, t \sim U([0,1]), x_i \sim D_n, y_i \sim G(\cdot)$$

- From GAN to WGAN
- WGAN code example
- Improved Training of Wasserstein GANs, WGAN-GP
- On the regularization of Wasserstein GANs, WGAN-LP
- Blog post on Wasserstein distance
- How does Lipschitz Regularization Influence GAN Training?

GAN training can take days for one hyperparameter setting!!!

- Mathematical difference:
  KL-Div $D_{KL}(p||q) = \sum_x p(x) \ln p(x) - p(x) \ln q(x)$ is not symmetric, does not satisfy triangle inequality. Wasserstein distance is a distance metric (symmetric, triangle inequality).
- practical advantage of Wasserstein-distance: takes metric $d(\cdot)$ in sample space for the samples x_i into account



both comparisons will have same KL-Divergence

Evaluation of GAN outputs

Takeaway points for this section

- be able to use the KL-Divergence
- that the inception score computes a distance function between two probability distributions $p(y|x)$ and $p(y)$ where y are the predicted labels
- be able to explain how Frechet inception distance works
- having heard of self-similarity
- be able to explain how perceptual path length works

GAN quality evaluation is a largely unsolved problem.

From Ali Borji `https://arxiv.org/abs/1802.03446`
`https://arxiv.org/pdf/2103.09396`

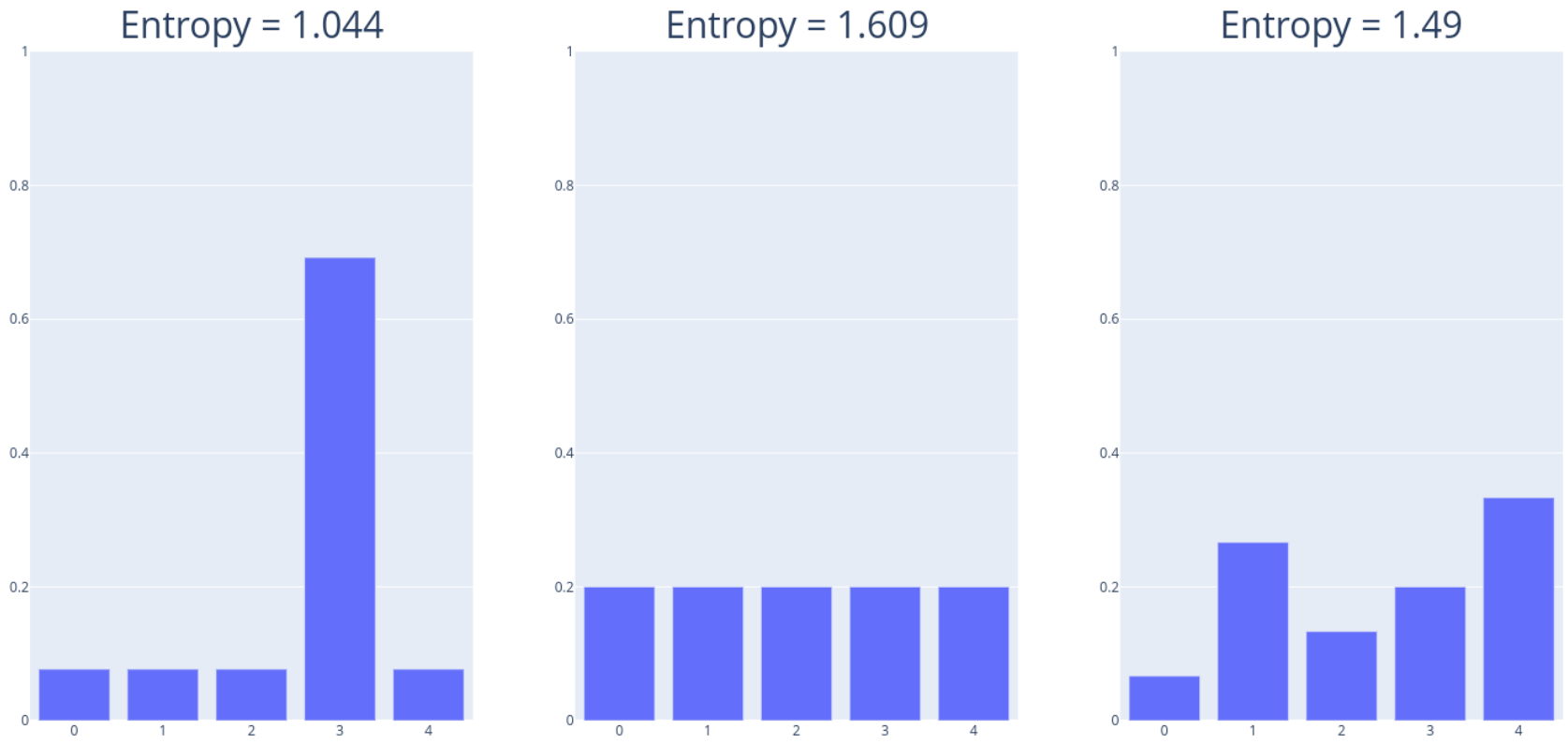
An example of desirable evaluation measure properties

- Prefer models with samples close to the training data distribution
- Prefer models with high diversity, sensitive to mode drop, mode collapse
- Agree with human rankings and human perceptual assessments
- Allow to be sensitive or invariant to certain image transformations, depending on what is needed for the use case (e.g. be mirror-invariant or sensitive to mirroring, same for jitter, color distortions etc)
- Work well with smaller sample sizes
- Optionally: prefer models where latent spaces are disentangled, that means that generation properties can be steered by sampling from certain well-defined regions

GAN quality evaluation is a not fully solved problem.

(One definition of) Entropy is: the average level of "information" or "uncertainty" inherent to the variable's possible outcomes.

$$\mathrm{H}(\boldsymbol{x}) = -\sum_{i=1}^{n} \mathrm{P}(x_i)\ln(\mathrm{P}(x_i))$$



The KL-Divergence $D_{KL}(p||q)$, also called relative entropy, between two distributions is a measure of how one probability distribution q is different from a second, reference probability distribution p

$$D_{KL}(p||q) = +\sum_x p(x) \ln \frac{p(x)}{q(x)} \text{ discrete}$$

$$D_{KL}(p||q) = +\int_x p(x) \ln \frac{p(x)}{q(x)} \text{ continuous with density}$$

$$D_{KL}(p||q) = E_{x\sim P}[\ln \frac{p(x)}{q(x)}] = -E_{x\sim P}[\ln \frac{q(x)}{p(x)}]$$

Asymmetrically defined (!!) expectation of the log ratio of two distributions.

Intuition: A good generator should generate:

- images that clearly belong to a class that humans can relate to, i.e. images that contain meaningful objects
- a variety of images, i.e. a variety of classes

How to represent this mathematically?

- $p(y|x)$ is for an image x the softmax probability for all the classes of an `inceptionV3` network.
  For a good generator, this distribution $p(y|x)$ should be peaked i.e. concentrated on one class, and thus have low entropy. Why? Idea: peaked $p(y|x) \leftrightarrow$ it generates images with clearly distinguishable objects.
- $p(y)$ is the marginal distribution over all classes. A generator should create a diverse range of classes. So $p(y)$ should be flat therefore.
- Thus one should be peaked, one should be flat.
- Summarize both points by calculating the exponential of the average KL-divergence between $p(y|x)$ and $p(y)$ (this should be high for a good generator):

$$\mathrm{IS}(\mathrm{G}) = \exp(\mathbb{E}_{x \sim G} D_{KL}(p(y|x)||p(y)))$$

---

show code example

- $p(y|x)$ is for an image x the softmax probability for all the classes of an `inceptionV3` network.
  For a good generator, this distribution $p(y|x)$ should be peaked i.e. concentrated on one class, and thus have low entropy. Why? Idea: peaked $p(y|x) \leftrightarrow$ it generates images with clearly distinguishable objects.
- $p(y)$ is the marginal distribution over all classes. A generator should create a diverse range of classes. So $p(y)$ should be flat therefore.
- Thus one should be peaked, one should be flat.
- Summarize both points by calculating the exponential of the average KL-divergence between $p(y|x)$ and $p(y)$ (this should be high for a good generator):

$$\mathrm{IS}(G) = \exp(\mathbb{E}_{x \sim G} D_{KL}(p(y|x)||p(y)))$$

---

show code example

- $p(y|x)$ is for an image x the softmax probability for all the classes of an `inceptionV3` network.
  For a good generator, this distribution $p(y|x)$ should be peaked i.e. concentrated on one class, and thus have low entropy. Why? Idea: peaked $p(y|x) \leftrightarrow$ it generates images with clearly distinguishable objects.
- $p(y)$ is the marginal distribution over all classes. A generator should create a diverse range of classes. So $p(y)$ should be flat therefore.
- Thus one should be peaked, one should be flat.
- Summarize both points by calculating the exponential of the average KL-divergence between $p(y|x)$ and $p(y)$ (this should be high for a good generator):

$$\mathrm{IS}(G) = \exp(\mathbb{E}_{x \sim G} D_{KL}(p(y|x)||p(y)))$$

---

show code example

- $p(y|x)$ is for an image x the softmax probability for all the classes of an `inceptionV3` network.
  For a good generator, this distribution $p(y|x)$ should be peaked i.e. concentrated on one class, and thus have low entropy. Why? Idea: peaked $p(y|x) \leftrightarrow$ it generates images with clearly distinguishable objects.
- $p(y)$ is the marginal distribution over all classes. A generator should create a diverse range of classes. So $p(y)$ should be flat therefore.
- Thus one should be peaked, one should be flat.
- Summarize both points by calculating the exponential of the average KL-divergence between $p(y|x)$ and $p(y)$ (this should be high for a good generator):

$$\text{IS(G)} = \exp(\mathbb{E}_{x \sim G} D_{KL}(p(y|x)||p(y)))$$

---

show code example

Criticism of the inception score:

See `https://arxiv.org/pdf/1801.01973.pdf` for a criticism of the usage of inception scores. In short, the paper shows that for datasets other than ImageNet, it is not a good measure.

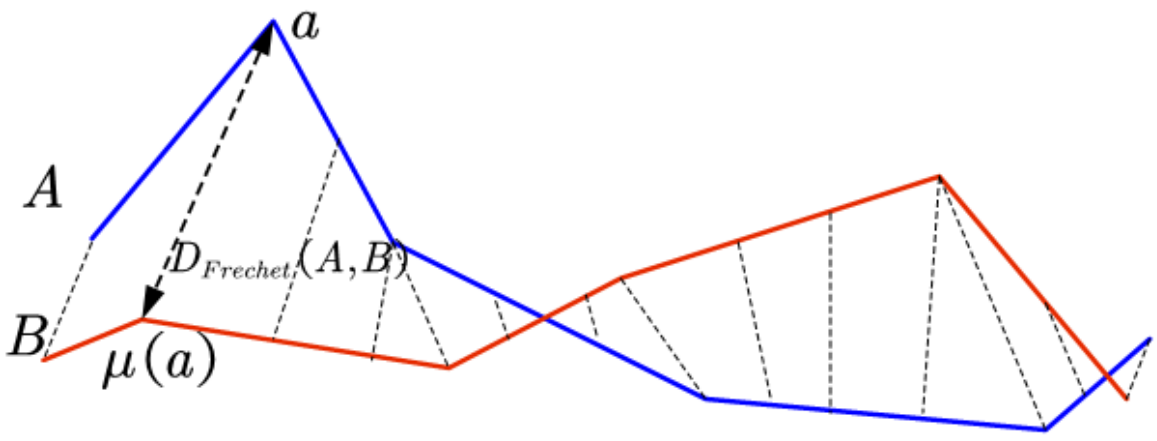
- Obviously, this score depends on your implementation of the InceptionV3 and may vary whether you use tensorflow, CNTK, pytorch, chainer, paddlepaddle or ... ! This is the first unrealiability.
- The other reason is that the estimate of $p(y)$ becomes not a good measure of variability, when the test images of your dataset are not from the set of all imagenet classes.
  In this case your test image classes may cluster into a subset of imagenet (e.g. cats will mostly fall into the cat race classes of imagenet), and the spread of $p(y)$ will not measure the diversity of your generated classes, but how well they cluster relative to the boundaries of imagenet classes.

(–) implementation-dependent

(–) value unclear for problems outside of imagenet

(–) weak at detecting mode collapse

Fréchet distance is a measure of similarity between curves:



https://www.mdpi.com/2220-9964/6/11/326

In the case where these two curves are Gaussian distributions, it takes the form:

$$d((\mu_g, \Sigma_g), (\mu_d, \Sigma_d)) = \sqrt{\|\mu_g - \mu_d\|_2^2 + Tr(\Sigma_d + \Sigma_g - 2(\Sigma_d \Sigma_g)^{1/2})}$$

Frechet Inception Distance (lower is better)

- Compute for your data the feature maps of a layer (pool_3) of an InceptionV3 network.
- Then compute mean and covariance μ_g, Σ_g for the above feature maps for the samples from the generator, and μ_d, Σ_d for the above feature maps for the samples from the data
- Then compute the Frechet Distance as if it were two Gaussians:

$$d((\mu_g, \Sigma_g), (\mu_d, \Sigma_d)) = \sqrt{\|\mu_g - \mu_d\|_2^2 + Tr(\Sigma_d + \Sigma_g - 2(\Sigma_d \Sigma_g)^{1/2})}$$

(FID calculation example)

$$d((\mu_g, \Sigma_g), (\mu_d, \Sigma_d)) = \sqrt{\|\mu_g - \mu_d\|_2^2 + Tr(\Sigma_d + \Sigma_g - 2(\Sigma_d\Sigma_g)^{1/2})}$$

- $\|\mu_g - \mu_d\|_2^2$ is a squared distance between the means of two gaussian distributions (one mean computed from images sampled from the training dataset D_n, one mean computed from images sampled from the generator $G(\cdot)$)
- $Tr(\Sigma_d + \Sigma_g - 2(\Sigma_d\Sigma_g)^{1/2})$ is a squared distance between the covariances of two gaussian distributions
- the Frechet Distance is a distance measure defined in the feature map space of a neural network, not in pixel space. advantage: can capture some high level semantic similarities (e.g. a face must have a nose close to eyes)
- depends on your implementation again (inceptionV3)

(+) can detect mode collapse

(+) OK correlation to human rating evaluation

(−) implementation-dependent

(−) Gaussian assumption is unclear, needs many samples (order of 10k-100k)

- Interpolation of latent-space vectors may yield surprisingly non-linear changes in the image.
- For example, features that are absent in either endpoint may appear in the middle of a linear interpolation path.
- This is a sign that the latent space is entangled and the factors of variation are not properly separated.
- To quantify this effect, we can measure how drastic changes the image undergoes as we perform interpolation in the latent space.
- Intuitively, a less curved latent space should result in perceptually smoother transition than a highly curved latent space

- Interpolation of latent-space vectors may yield surprisingly non-linear changes in the image.
- For example, features that are absent in either endpoint may appear in the middle of a linear interpolation path.
- This is a sign that the latent space is entangled and the factors of variation are not properly separated.
- To quantify this effect, we can measure how drastic changes the image undergoes as we perform interpolation in the latent space.
- Intuitively, a less curved latent space should result in perceptually smoother transition than a highly curved latent space

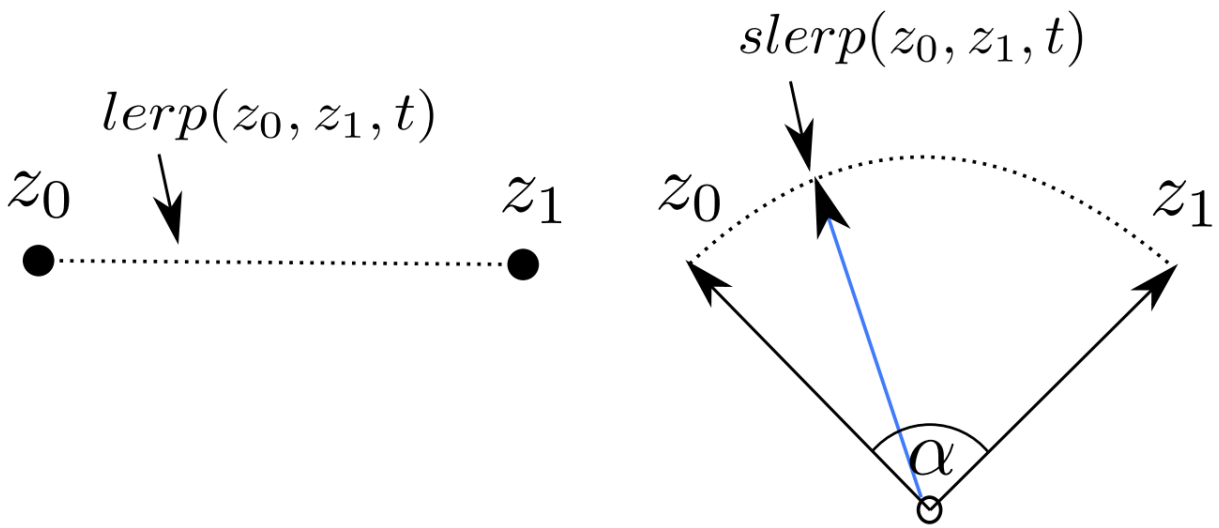
- Interpolation of latent-space vectors may yield surprisingly non-linear changes in the image.
- For example, features that are absent in either endpoint may appear in the middle of a linear interpolation path.
- This is a sign that the latent space is entangled and the factors of variation are not properly separated.
- To quantify this effect, we can measure how drastic changes the image undergoes as we perform interpolation in the latent space.
- Intuitively, a less curved latent space should result in perceptually smoother transition than a highly curved latent space

- Interpolation of latent-space vectors may yield surprisingly non-linear changes in the image.
- For example, features that are absent in either endpoint may appear in the middle of a linear interpolation path.
- This is a sign that the latent space is entangled and the factors of variation are not properly separated.
- To quantify this effect, we can measure how drastic changes the image undergoes as we perform interpolation in the latent space.
- Intuitively, a less curved latent space should result in perceptually smoother transition than a highly curved latent space

- Interpolation of latent-space vectors may yield surprisingly non-linear changes in the image.
- For example, features that are absent in either endpoint may appear in the middle of a linear interpolation path.
- This is a sign that the latent space is entangled and the factors of variation are not properly separated.
- To quantify this effect, we can measure how drastic changes the image undergoes as we perform interpolation in the latent space.
- Intuitively, a less curved latent space should result in perceptually smoother transition than a highly curved latent space

- Draw z_1, z_2 two random input codes. Could generate images $G(z_1)$, $G(z_2)$. Instead:
- Interpolate in code space along a circle between z_1 and z_2 using the $slerp(\cdot)$ function

$$slerp(z_1, z_2, t) = \frac{\sin((1-t)\alpha)}{\sin(\alpha)} z_1 + \frac{\sin(t\alpha)}{\sin(\alpha)} z_2, \; cos(\alpha) = \frac{z_1 \cdot z_2}{\|z_1\| \|z_2\|}$$



- Take some feature map from a CNN (originally a VGG) $\phi(\cdot)$ and look at the finite difference between the features $\phi(\cdot)$ of two very close generated images $G(s)$, $G(s_{+\epsilon})$ along the slerp-path. In other words, look at the squared norm of the directional derivative of $\phi(G(s))$ at $slerp(z_1, z_2, t)$ along the slerp-path

$$g_\epsilon(s) = \left\| \frac{\phi(G(s)) - \phi(G(s_{+\epsilon}))}{\epsilon} \right\|^2$$
$$s = slerp(z_1, z_2, t), \;\; s_{+\epsilon} = slerp(z_1, z_2, t + \epsilon)$$

- Compute the expectation $g_\epsilon(s)$ over random draws of z_1, z_2, t

$$PPL = \mathbb{E}_{z_1 \sim P(z), z_2 \sim P(z), t \sim U(0,1)}[g_\epsilon(\, slerp(z_1, z_2, t))]$$

- Meaning: expected/averaged directional derivative length along a slerp-path between two images, as measured in some CNN feature space. Lower is better

(+) can detect mode collapse

(+) ok correlation to human rating evaluation

(−) exploits a compression property of current GANs, might not hold for all future architectures, thus correlation to human rating evaluation may need to be checked from time to time

Alternative concept, out of class: Multiscale statistical similarity, higher is better.

Originally from a paper from 2004: `https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=1292216`.

Core idea: take images, extract patches at different scales, compute similarity measures between patches from two images.
`https://arxiv.org/abs/2004.01864`, average similarities over all those patches

(+) It can find mode collapse.

All three scores are not trained to approximate how realistic something looks like to a human.
For now, the only way to go is to rely on **evaluation by human raters**, which is costly, as you need to design a human study.

More resources

Generative Adversarial Networks

NIPS 2016 Tutorial: Generative Adversarial Networks