50.039 – Theory and Practice of Deep Learning

Alex

Week 09: Generative Adversarial Networks I

[The following notes are compiled from various sources such as textbooks, lecture materials, Web resources and are shared for academic purposes only, intended for use by students registered for a specific course. In the interest of brevity, every source is not cited. The compiler of these notes gratefully acknowledges all such sources.]

Problem:
given a dataset $D_n = \{x_1, \ldots, x_n\}$, for example of images, be able to generate *similar* images as in the dataset.

Critical thinking: What is similar ?
Formally:

- given D_n, learn a model G (by its parameters) so that one can randomly sample new samples using this model by

$$x = f(z), z \sim N(0, I_d)$$

- the learning process should enforce some kind of similarity between the D_n and $x = G(z), z \sim N(0, I_d)$

What kind of similarity ?

D_n are samples drawn from some distribution P_{data}. $x = G(z), z \sim N(0, I_d)$ allows to sample from the trained distribution P_{GAN}. Optimally one wanted to measure a loss between P_{data} and P_{test}, and learn a model so that $P_{GAN} = P_{data}$. Why this is not directly doable ?
Practically, one has to use measures between the finite sample set D_n and a set of samples drawn from f.

Ugly question: Why learning the mean and the standard deviation of every pixel in images from D_n does not allow to sample things like bedrooms? If you have no clue, compute the (rgb-sub-)pixel-wise mean and std, and sample from it!

We need to go deeper ...

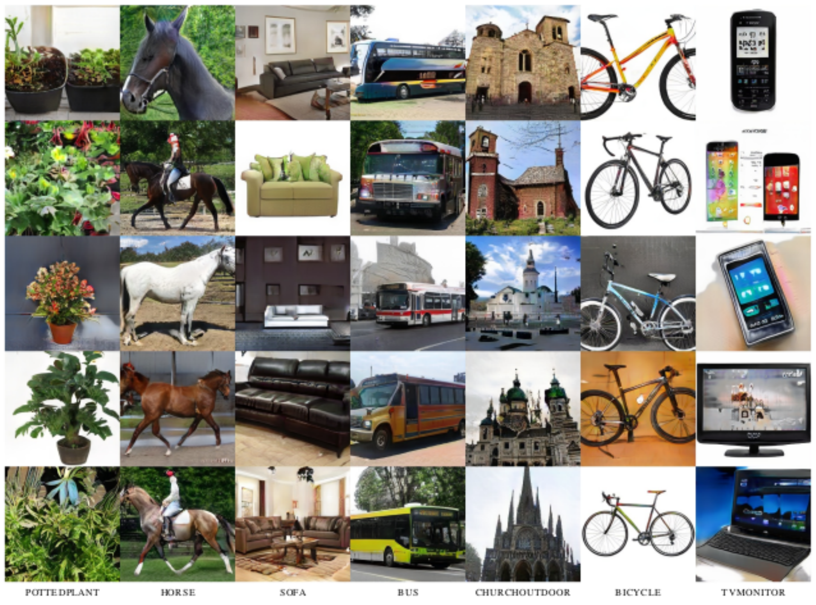


Figure 7: Selection of 256×256 images generated from different LSUN categories.

fake objects created using a Wasserstein GAN with progressive growing `https://arxiv.org/pdf/1710.10196.pdf`.
Find the mistakes in details ... a GAN is not a high level reasoning algorithm

Contents over 2 courses / Key takeaways:

- vanilla GAN: components and its objective
- image generation models for G
  - fractionally strided convolutions
- be able to explain the two main problems of GANs
- Wasserstein GAN
- combining GANs and GAN-type mappings: cycleGAN, Domain Transfer
- students are expected to explain how the above entities work

1 Vanilla GAN

1.1 generating images – the generator as a special case for images

Requirements:

- input space is a vector space $\mathbb{R}^d$
- input to a model is $\mathbb{R}^d \ni z \sim N(0, I_d)$, a vector randomly sampled from a high dim random normal
- output: $x = G(z)$ for example an image of with width height 256

What shape would the output have for batchsize 1?

Idea: do it similar to a convolution, but with layers in which the output feature map after the convolution has a larger spatial resolution than the input feature map
An example model: Radford et al. `https://arxiv.org/pdf/1511.06434.pdf`

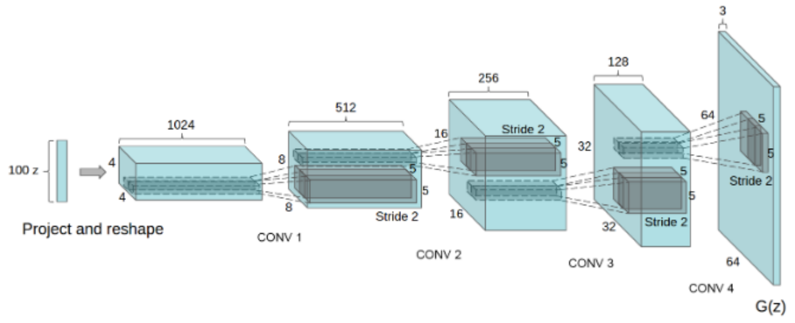
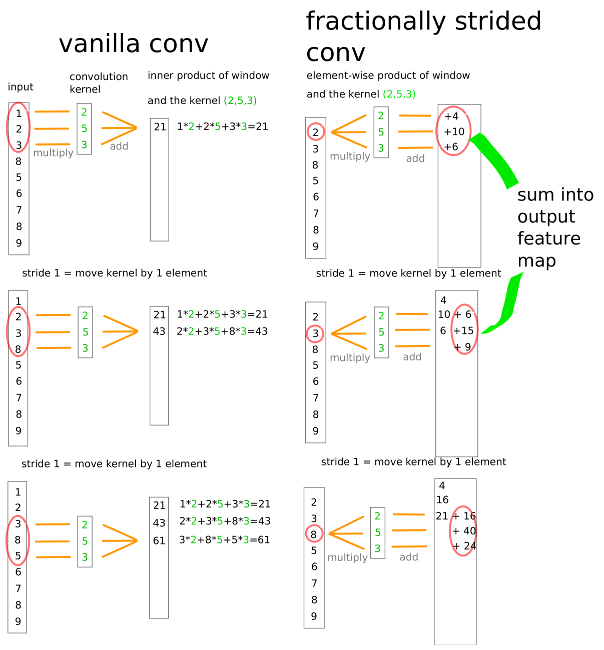


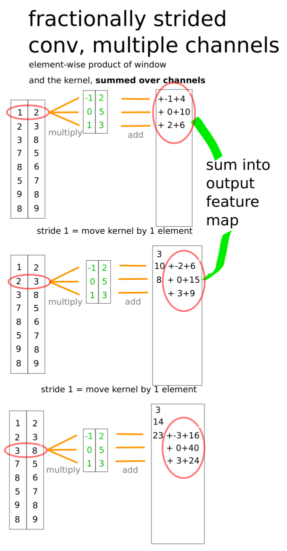
Figure 1: DCGAN generator used for LSUN scene modeling. A 100 dimensional uniform distribution Z is projected to a small spatial extent convolutional representation with many feature maps. A series of four fractionally-strided convolutions (in some recent papers, these are wrongly called deconvolutions) then convert this high level representation into a 64×64 pixel image. Notably, no fully connected or pooling layers are used.

- conventional CNN: every n layers half resolution and increase (double?) number of channels
- GAN generator: every n layers double resolution and decrease (half?) number of channels

How to do a convolution which doubles a resolution? **fractionally strided convolution!**

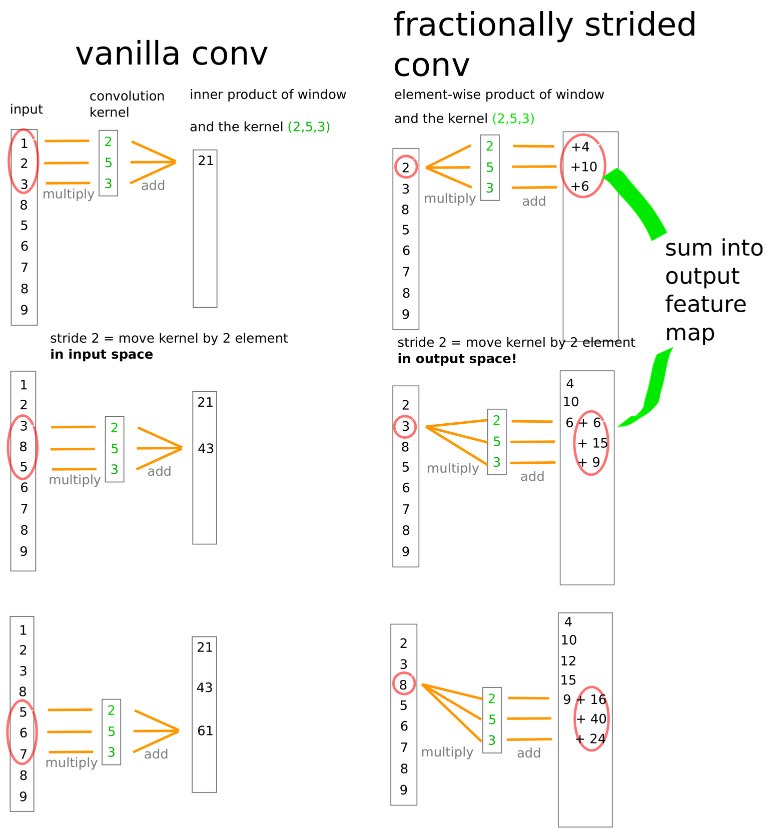


For multiple input channels the working principle is analogous:



fractionally strided conv, multiple channels
element-wise product of window
and the kernel, **summed over channels**
multiply
add
+-1+4
+ 0+10
+ 2+6
sum into output feature map
stride 1 = move kernel by 1 element
10 +-2+6
8 + 0+15
+ 3+9
23 +-3+16
+ 0+40
+ 3+24

A second big difference is how stride is applied:


vanilla conv
fractionally strided conv
input
convolution kernel
inner product of window and the kernel (2,5,3)
element-wise product of window and the kernel (2,5,3)
multiply
add
sum into output feature map
stride 2 = move kernel by 2 element in input space
stride 2 = move kernel by 2 element in output space!
21
43
61
+4
+10
+6
4
10
6 + 6
+ 15
+ 9
12
15
9 + 16
+ 40
+ 24

	vanilla convolution	fractionally strided convolution
kernel size, e.g. 3	multiply 3 input elements with 3 kernel elements as an inner product, copy in 1 output location	multiply one input element with 3 kernel elements, copy in 3 output locations
stride, e.g. 2	stride applied in input feature map	stride applied in output feature map

the same old story

for the generator model apply all the tricks of convolutional NNs (batchnorm, residual, better activation functions, neural architecture search, etc etc)

have an idea for the Generator model $G(z)$ now

1.2 for training: a second model required – the discriminator

How do we define a kind of loss which allows us to train the generator?

The **discriminator** $D(x)$ is a neural network which takes pairs of images as inputs. Each pair contains a real and an image which is the output of a generator. The discriminator has two classes as output: real and fake. It is trained to label the real images as class 1 ("real"), and the generated ones as fake. Let denote $D(x) \in [0, 1]$ the probability for input sample x being real, $D(x) = 1$ means that the discriminator is sure with probability 1 that x is real

Assume now: we would have a pretrained and fixed discriminator – then the goal would be: the generator should create similar to real images. Here: similarity $\leftrightarrow$ deep inside classifier regions classified as real images.

Generator G tries to generate samples such that the discriminator D thinks they are real. Lets assume that the discriminator outputs probabilities $D \in [0, 1]$. Then the aim is $D(G(z)) \to 1$. Maximizing a non-negative function, is equal to minimizing its negative logarithm. As loss L_G of the generator one can write:

$$\begin{aligned} L_G &= \min_G & E_{z \sim N(0,1)} & [-1 \cdot \log D(G(z))] \\ \hat{L}_G &= \min_G \frac{1}{N} & \sum_{z_i \in minibatch(N(0,1))} & -1 \cdot \log D(G(z_i)) \end{aligned}$$

The difference between two equation is: the first is the objective we want to optimize, the second is the approximation of the expectation by one minibatch. P_z is the sampling distribution for the random input codes z for the generator G.

Note: for a fixed and pretrained discriminator, we have one single loss function in which it is a part:
The loss function in this case is not a trivial loss function such as $\|f(x) - y\|^2$ as it involves a function $D(x)$ which is implicitly encoded by the discriminator. Still it is a loss minimization problem for parameters of the generator G.

This formulation is not used because it leads to one central problem of GAN training, namely **mode collapse**: optimizing the generator will result in outputting one single image which fools the discriminator most, that is it will learn to output for any sampled z almost the same $x = G(z)$ - namely that image x such that $-1 \cdot \log D(x)$ is minimal.

generator quality vs diversity

There are two objectives in sample generation: sample quality and 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.

1.3 interleaved joint training: discriminator and generator

The idea of the gan approach is, however, to not assume to have a pretrained discriminator, and instead to train discriminator and generator in interleaved steps from random initializations (also to **avoid mode collapse by making the discriminator a moving target** for the generator!!). Since we know which images comes from the generator, and which comes from the original dataset, we can automatically label them! Once we have labels, we are ready to train the discriminator, too.

The setup is - see page 17 / 19 in Goodfellow's tutorial `https://arxiv.org/pdf/1701.00160.pdf`:

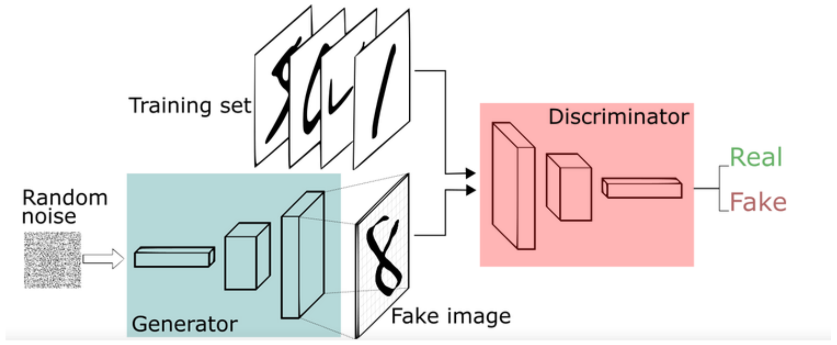


image from Thalles Silva `https://medium.freecodecamp.org/an-intuitive-introduction-to-generative-adversarial-networks-gans-7a2264a81394`

vanilla GAN training

- interleaved training – step 1: train the discriminator
  - fix generator weights, generate fake samples, label them as class 0 (fake class)
  - train discriminator weights by minibatches having one time fake (labeled as 0) and another time real dataset samples (the real ones you draw from your dataset) (labeled as 1) – loss is sum of both losses for a classification problem
- interleaved training – step 2: train the generator
  - fix discriminator weights, train generator weights so as to make the prediction of generated samples to have high scores for class 1 (real)

Two player game:

- generator tries to create images close to real relative to D_n
- discriminator tries to learn to separate the generator images as fakes.

what loss to use for training the discriminator ?
The discriminator tries to classify real samples as real that is $D(x) \to 1, x \in data$ and classify the fake samples as fake, that is $D(G(z)) \to 0$. How to arrive at a minimization objective from this ?

One component will be minimizing the negative logarithm of $D(x)$ for x being drawn from the real data, so

$$\begin{aligned} &\min_D && E_{x\sim P_{data}} && [-1 \cdot \log D(x)] \\ &\min_D && \sum_{x_i \sim minibatch(data)} && -1 \cdot \log D(x_i) \end{aligned}$$

As for the goal $D(G(z)) \to 0$, we can write it as: $1 - D(G(z)) \to 1$, and then arrive at the loss of the discriminator L_D

$$\begin{aligned} L_D &= \min_D && E_{x\sim P_{data}}[-1 \cdot \log D(x)] + E_{z\sim N(0,1)}[-1 \cdot \log(1 - D(G(z)))] \\ \hat{L}_D &= \min_D && \sum_{x_i \sim minibatch(data)} -1 \cdot \log D(x_i) + \sum_{z_i \in minibatch(N(0,1))} -1 \cdot \log(1 - D(G(z))) \end{aligned}$$

losses used in training

- during training one uses as losses $\hat{L}_D$ and $\hat{L}_G$. $\hat{L}_D$ can be realized as a sum of two components – a minibatch over real data, and a minibatch over generator-sampled data

GAN at testing time

The generator is a function to sample from a learned probability distribution ν:

- 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 ν that was learned by the generator is not explicitly given. You can only sample from it.

1.4 Tons of tricks

Also useful `https://github.com/soumith/ganhacks`

1.5 A vanilla GAN skeleton code in pytorch

what components do you need?

- we update once the generator, and once the discriminator weights, thus we will need two optimizers `optimizerD` and `optimizerG` – one of these will update discriminator params, the other generator params!
- prepare your one class training data`dataloader= ...`, send to GPU
- define your generator model, send it to GPU, init 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= ...)`

```
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 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()

#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()
```

`https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html`
We will see in the next class that Wasserstein training of GANs is similar, as it just results in replacing the discriminator and its loss.

- first three windows: packages, parameters, dataloader for D_n
- then weight init
- then generator model (`nn.ConvTranspose2d` is the fractionally strided convolution) and its init
- then discriminator model (`nn.ConvTranspose2d` is the fractionally strided convolution) and its init
- then loss criterion and both optimizers
- actual training similar to above
- code contains example how to deal with dataloader, how to define generator and discriminator models

1.6 Problems with GAN training

- mode collapse: This describes a problem where the generator for one class creates always only the same image (almost, with very minor difference every time). This one image is very good at fooling the discriminator. Obviously there is no randomness in sampling anymore.
- non-convergence: Convergence of this two player game is not guaranteed. You do not know where training will lead you. What can happen: in every iteration one player reverses the progress of the other player, oscillate between 2 or 3 states and no further progress.
- sensitivity to unlucky hyperparameter settings `https://papers.nips.cc/paper/7350-are-gans-created-equal-a-large-scale-study.pdf`

For an out of class view: `https://medium.com/@jonathan_hui/gan-why-it-is-so-hard-to-train-generati`
Open questions:

- modelling quality of a trained GAN: two potentially opposing objectives
  - closedness to training data
  - versus diversity, one cannot be good at both at the same time
  - existing measures are very adhoc
- does a GAN drop some frequent parts of the training data (a subset of the training data which has high probability in the training data but nothing similar is hardly sampled)
- more than just memorizing some training examples?

1.7 A fun application: face morphing

fake VIPs created using a Wasserstein GAN with progressive growing `https://arxiv.org/pdf/1710.10196.pdf`.
it would be very interesting to compute in a feature space of both actor and critic the five or ten closest real celebrities from the training D_n.

What is the idea? Given two real faces, handbags, whateverthings x_1, x_2, find input codes which are close to generating them:

$$z_1 = \text{argmin}_z \|x_1 - G(z)\|$$
$$z_2 = \text{argmin}_z \|x_2 - G(z)\|$$

Now you can interpolate between z_1 and z_2 in generator input space (e.g. linear interpolation, but a spherical interpolation can be better sometimes). For simplicity lets to it linearly:
$z_{(k)} = az_1 + (1-a)z_2,\ a \in [0,1]$, e.g. $a = \{1/5, 2/5, 3/5, 4/5\}$

Now map the interpolates $z_{(k)}$ back to an interpolated face $G(z_{(k)})$. Face-morphing.

not guaranteed to yield semantically meaningful results `https://en.wikipedia.org/wiki/Janus`.
applications of face morphing?
useful for forensics? why or why not ?

practically more useful is maybe bedroom morphing if you plan to move together ...

LSUN bedrooms class is really a GAN benchmark dataset!! So why not test it with something challenging.

1.8 Some Training Hacks for GANs (out of class, exams)

None of those is guaranteed to work better ... must validate each of those

- auxiliary classifier GAN (AC-GAN): if your training data has labels, then:
  - add to GAN inputs a class parameter
  - use as loss your gan loss plus a standard classification loss. The added classification term is added to the discriminator and the generator, so that both learn to classify
- if you do batchnormalization, then use minibatches that consist of only generator and only discriminator samples. Do not mix generator and discriminator samples withhin one minibatch
- alternative to batchnormalization is: instance normalization layers, in which every channel of every sample is separately normalized (equivalent to batch normalization with batch size of 1)
- use of randomized soft labels instead of 1 when training the discriminator. Draw labels from around 1.
- avoid maxpool and reLU – because they have in large ranges zero gradients, use operators that have non-zero gradients mostly such as leaky ReLU and average pooling
- many people report good results with ADAM
- add noise for better learning: Rarely (1% ? 5% of all samples?) flip labels when training the discriminator to add label noise, occasionally use older generator G or older Discriminator D.
- occasionally train discriminator with a history of older generator samples, or train it with a history of samples from the last $N = 50$ iterations.

- inspect your generated samples during training if the discriminator loss goes very fast down to zero, you may need to restart. Initially large gradients in the generator are okay. If they get too small,
- use Wasserstein GANs with gradient penalty terms, Boundary Equilibrium GANs instead of classical formulation.
- see also Gulrajani et al. `https://arxiv.org/abs/1704.00028` and Petzka et al. `https://openreview.net/pdf?id=B1hYRMbCW` on penalization of Wasserstein GANs
- newer results suggest: use together: WGAN with some gradient penalty (the point directly above) + ADAM + neural networks with Lipschitz-smooth gradients, thus one need to use Lipschitz-smooth activation functions such as ELU instead of RELUs + Two-timescale-update rule. `https://arxiv.org/pdf/1706.08500.pdf`:

  Use as learning rate $a(n)$ for the generator, and $b(n)$ for the discriminator:

$$\sum_n a(n) = \infty, \sum_n a^2(n) < \infty$$

$$\sum_n b(n) = \infty, \sum_n b^2(n) < \infty$$

$$a(n) = o(b(n)) \Rightarrow \forall k\ \exists N : \forall n \geq N : a(n) < kb(n)$$

  The last means, that as you do more and more training, the learning rate for the generator decreases faster than for the discriminator.
- `https://arxiv.org/pdf/1711.10337.pdf` – all GANs are very sensitive to hyperparameter settings.
- all these tips might be totally outdated in two years, therefore ... think

2 Evaluation of your GAN outputs

From this chapter I expect you only to understand how Frechet inception distance works, and that inception score uses an inception network to compute conditional and marginal probability and that the inception score computes a distance function between two probability distributions but not its deeper idea. This is a large unsolved problem!

The rest is out of class.

A popular – but questioned – metric is the **inception score** – higher is better.
from Salimans et al: `https://arxiv.org/pdf/1606.03498.pdf`, see also `https://nealjean.com/ml/frechet-inception-distance/`
Coarse idea, the exponential of an averaged (over the generated images) KL-Divergence of two terms $p(y|x)$ and $p(y)$

$$\begin{aligned} IS(G) &= exp(S) \\ S &= E_{x\sim G} D_{KL}(p(y|x) \| p(y)) \\ D_{KL}(r\|s) &= \int r(x)\log\left(\frac{r(x)}{s(x)}\right)dx \end{aligned}$$

What is the idea here ? One needs to know basics about Kullback-Leibler-Divergence and Entropy

- $D_{KL}(r\|s)$ is the Kullback-Leibler-Divergence, a kind of dissimilarity measure between probability distributions given by densities r and s.
- Distributions and entropies: a peaked distribution has low entropy. The uniform distribution has maximal entropy among all distributions.
- $p(y|x)$ 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, 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, and the KL-divergence of these two is high then

One can show (out of class) that KL-divergence is related to a difference of two entropies:

$$D_{KL}(p(y|x)\|p(y)) = H(y) - H(y|x)$$

See `https://arxiv.org/pdf/1801.01973.pdf` for a criticism of the usage of inception scores. In principle they state, 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 or CNTK or pytorch or chainer! 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.
The example for a cat-GAN: A flat $p(y)$ will be achieved when the generated images are misclassified often as non-cats. This is maybe not a good measure of variability, since we are interested in having diverse cats, but still cats. Diversity should be the diversity within the cat classes, not how often the images look like boats.

An alternative is the **Frechet Inception Distance (FID):** – 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 samples from the generator, and μ_d, Σ_d – for the samples from the data
- Then compute the Frechet Distance between the two gaussians: $d(x,g) = \|\mu_g - \mu_d\|_2^2 + Tr(\Sigma_d + \Sigma_g - 2(\Sigma_d\Sigma_g)^{1/2})$

  out of class: To understand the weird term $Tr(\Sigma_d + \Sigma_g - 2(\Sigma_d\Sigma_g)^{1/2})$, consider that:

  $$\Sigma_d + \Sigma_g - 2(\Sigma_d)^{1/2}(\Sigma_g)^{1/2}$$

  could remind you with $x = (\Sigma_d)^{1/2}$, $y = (\Sigma_g)^{1/2}$ of

  $$x \cdot x + y \cdot y - 2x \cdot y = \|x - y\|^2$$

  That looks almost like the norm of a difference of two vectors derived from inner products. The only thing that is wrong is that $(\Sigma_d\Sigma_g)^{1/2}$ is a matrix, but taking $Tr(A) = \sum_i A_{ii}$ turns any matrix A into a scalar.
  What is left to understand:

  $$A \cdot B := Tr(AB^\top)$$

  defines an inner product on the set of all matrices (e.g. `https://math.stackexchange.com/questions/476802/how-do-you-prove-that-trbt-a-is-a-inner-product`), and thus defines a valid metric on every subset of the set of all matrices. Since Σ is positive definite, sigma can be decomposed as (not uniquely)

  $$\Sigma_d = A^\top A, A := (\Sigma_d)^{1/2}$$
  $$\Sigma_g = B^\top B, B := (\Sigma_g)^{1/2}$$

and that A, B is used for defining a metric on symmetric positive definite matrices.

see Heusel et al. “GANs trained by a two time-scale update rule converge to a Nash equilibrium” `https://arxiv.org/pdf/1706.08500.pdf`

Problem:

- not sure if the distribution of your feature maps is close to a Gaussian, if it is not, then the difference of the gaussian approximations says little.

Another alternative is **Multiscale statistical similarity** – lower is better: for example used in: Tero Karras et al. Progressive Growing of Gans for improved Quality, Stability, and Variation `https://openreview.net/pdf?id=Hk99zCeAb`, originally from a paper from 2004: `https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=1292216`. It can find mode collapse, that is the problem when a generator can produce for a class only one image, but that image fools the discriminator very strongly.
The coarse idea: take images, extract patches at different scales, compute similarity measures between patches from two images.

Both scores do not really capture how realistic something looks like to a human. Alternative: **evaluation by human raters**! Its costly, you need to design a human study.

3 outlook

`https://arxiv.org/abs/1812.04948` Style-Gan: Gan with ideas from style transfer
`https://github.com/lernapparat/lernapparat/blob/master/style_gan/pytorch_style_gan.ipynb` (Piotr Bialecki and Thomas Viehmann) with pretrained weights – this code runs out of the box :)

4 in class: face morphing

The dropbox link contains both:
You can use either the github code from above (high res 1000×1000) `stylegan.py, karras2019*.pt` from `https://github.com/lernapparat/lernapparat/blob/master/style_gan/pytorch_style_gan.ipynb` or models and an example code to sample 64 small faces `ganforplay.py, celebA*.pt, improved-wgan-pytorch.zip` from `https://github.com/jalola/improved-wgan-pytorch`.
Your task:

- take 2 faces (a cat face??), **crop them** to have a similar face to back-ground ratio as in the Gan-generated, scale them (to 64×64 , 1000×1000)

- find z_1, z_2 by

$$z_1 = \text{argmin}_z \|x_1 - G(z)\|$$
$$z_2 = \text{argmin}_z \|x_2 - G(z)\|$$

  I suggest multiple starting points. Which one to choose ? Ask the discriminator, or choose manually

- lazy approach: interpolate K points between z_1 and z_2 by some function $interp(z_1, z_2, \lambda)$. If you do it lazy, then you just do linear interpolation:

$$interp(z_1, z_2, \lambda) = \lambda z_1 + (1 - \lambda) z_2$$

  if you want to get a better interpolation, see the `slerp` function in `https://github.com/soumith/dcgan.torch/issues/14`. Then:

$$z(\lambda) = interp(z_1, z_2, \lambda)$$
$$G(z(\lambda)) \to \text{interpolated face}$$

- visualize your interpolated question faces :)