Generative Adversarial Networks (GAN), part1

Alexander Binder

University of Oslo (UiO)

April 21, 2021

UiO : Department of Informatics
University of Oslo

Learning goals

- classic GANs: key components and their objectives: generator, discriminator
- be able to explain what mode collapse is
- evaluation of GAN outputs
- a better model I progressive growing for GANs
- applications I: image morphing
- applications II: cross-domain mappings

...

Core idea: 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:

- a model G (and its parameters) that can randomly generate new samples (images in our example), using a randomly generated noise vector as the model input. In other words, our model G satisfies:

  $$x = G(z), z \sim N(0, I_d), \text{ with } d \text{ the number of input parameters.}$$

- 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 $x = G(z), z \sim N(0, I_d)$.

The important keywords in the previous description were *"similar"* and *"similarity"*. What do we mean by *"similar"* images?

The GAN G, once trained, will generate samples with a distribution P_{GAN} by:

$$x = G(z), z \sim N(0, I_d),\ x \text{ is a random var now}$$
$$x \sim P_{GAN}$$

D_n are samples drawn from some distribution P_{data}.

We need a similarity metric, which measures how similar the samples in both sets P_{data} and P_{GAN} are, and learn a model G, so that $P_{GAN} \approx P_{data}$. In practice, we dont have direct access to P_{GAN}, and also not to P_{data}.

Practically, one has to use measures between a finite samples, namely D_n and samples drawn from G.

Why pixel-wise modelling does not work?

Simple idea:

- estimate for every pixel mean and standard deviation
- draw for every pixel from a gaussian with these parameters, depending on the pixel

Why does learning the mean and the standard deviation of every pixel in images from D_n not allow to sample things like bedrooms?

If you have no idea, compute the (rgb-sub-)pixel-wise mean and std, and sample from it!

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.

Pay attention to the mistakes in the image details: GANs are not a high level reasoning algorithms. They model local pixel correlations ...

Outline

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 1000 dims onto an image 256×256?

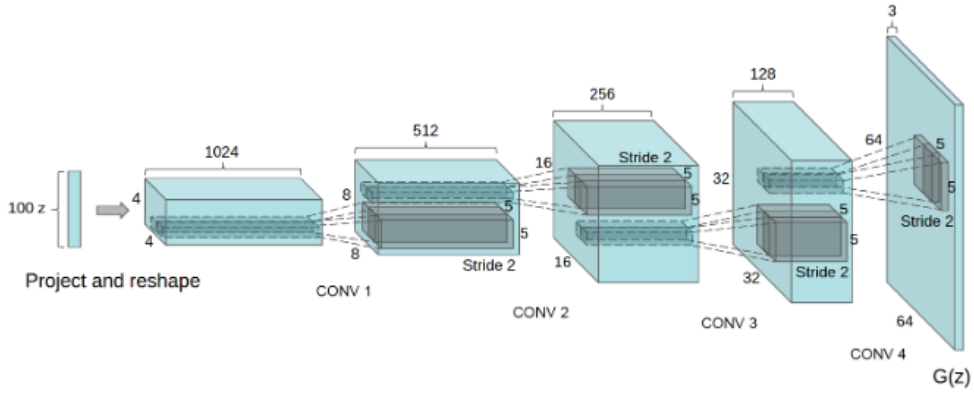


Figure 1: DCGAN generator used for LSUN scene modeling. A 100 dimensional uniform distribution Z is projected to a small spatial extent convolutional representation with many feature maps. A series of four fractionally-strided convolutions (in some recent papers, these are wrongly called deconvolutions)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

By analogy,

- In a conventional CNN models:
  every *n* layers, the model halves the image resolution and doubles / increases the number of channels in feature maps.
- GANs generator models:
  every *n* layers, the model increases the resolution progressively and, if needed, decreases (example: halves) the number of channels in the feature maps.

The layers should have progressively increasing spatial resolutions.

Core idea 1: Can use upsampling and convolutions.

Refer to the figure below, taken from Radford et al. https://arxiv.org/pdf/1511.06434.pdf

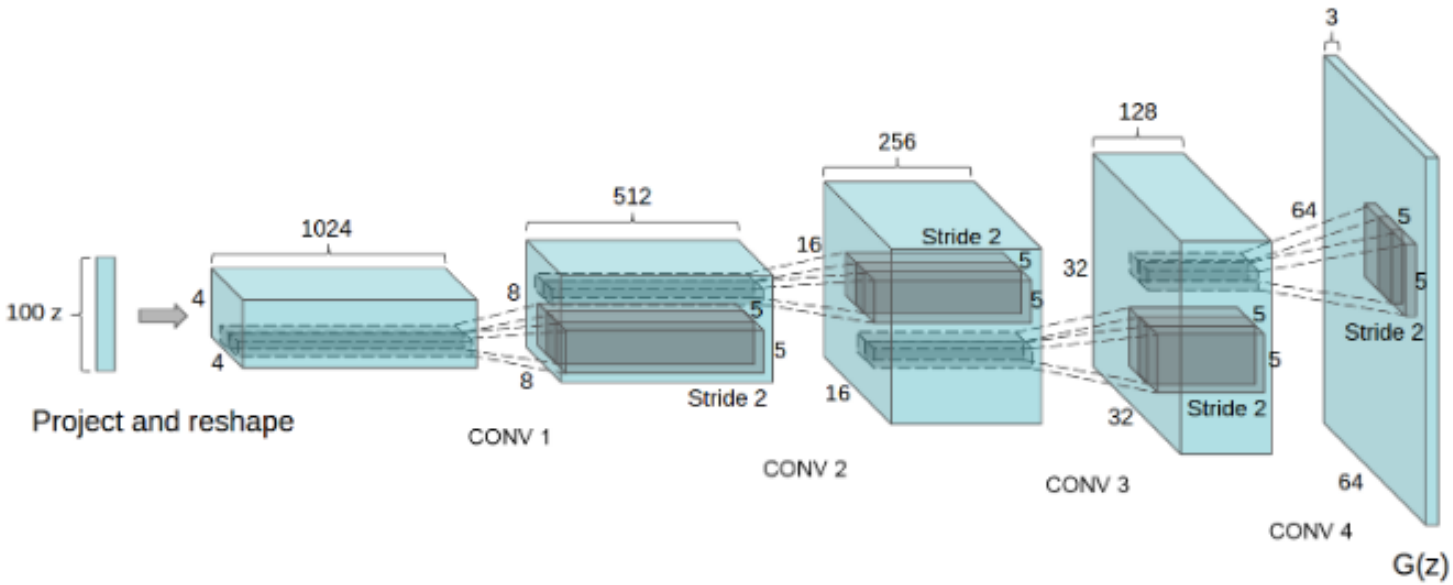


Figure 1: DCGAN generator used for LSUN scene modeling. A 100 dimensional uniform distribution Z is projected to a small spatial extent convolutional representation with many feature maps. A series of four fractionally-strided convolutions (in some recent papers, these are wrongly called deconvolutions)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

The layers should have progressively increasing spatial resolutions.

Core idea 2: Similar to the convolution operation, but in reverse – **fractionally strided convolution**. Refer to the figure below, taken from Radford et al. https://arxiv.org/pdf/1511.06434.pdf

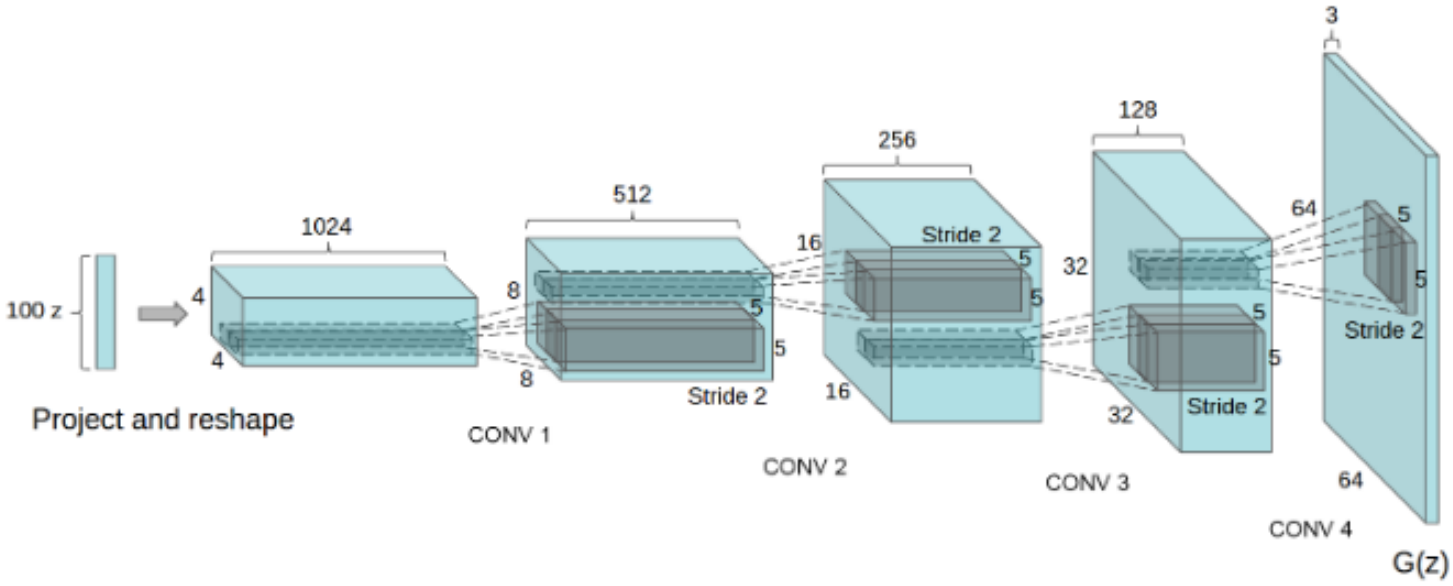
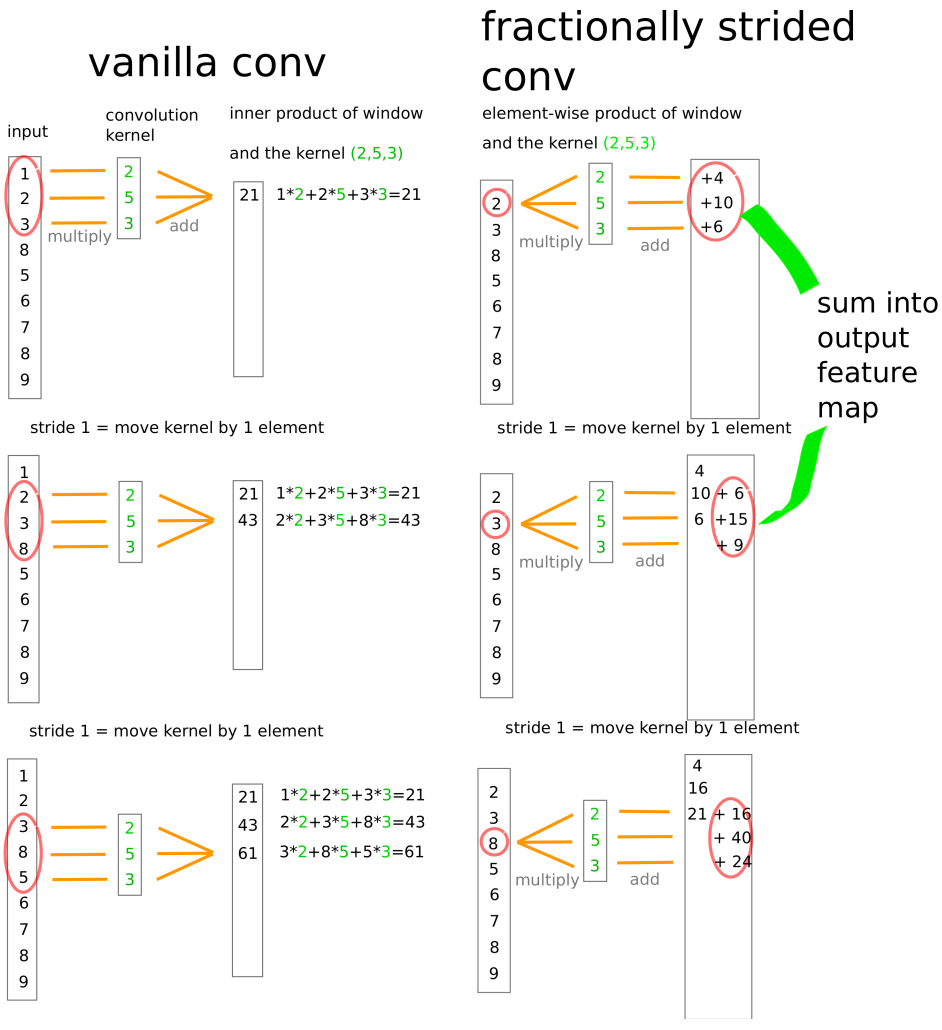
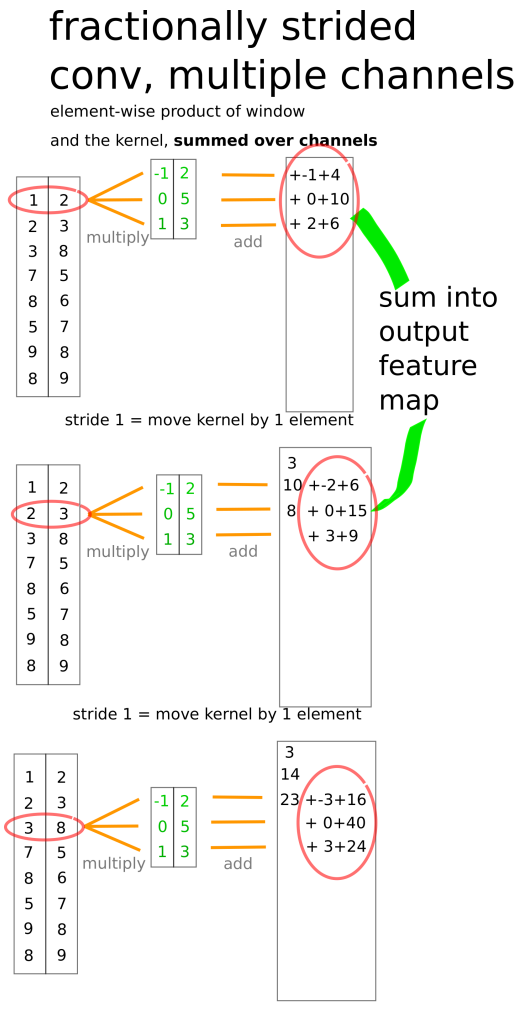


Figure 1: DCGAN generator used for LSUN scene modeling. A 100 dimensional uniform distribution Z is projected to a small spatial extent convolutional representation with many feature maps. A series of four fractionally-strided convolutions (in some recent papers, these are wrongly called deconvolutions)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

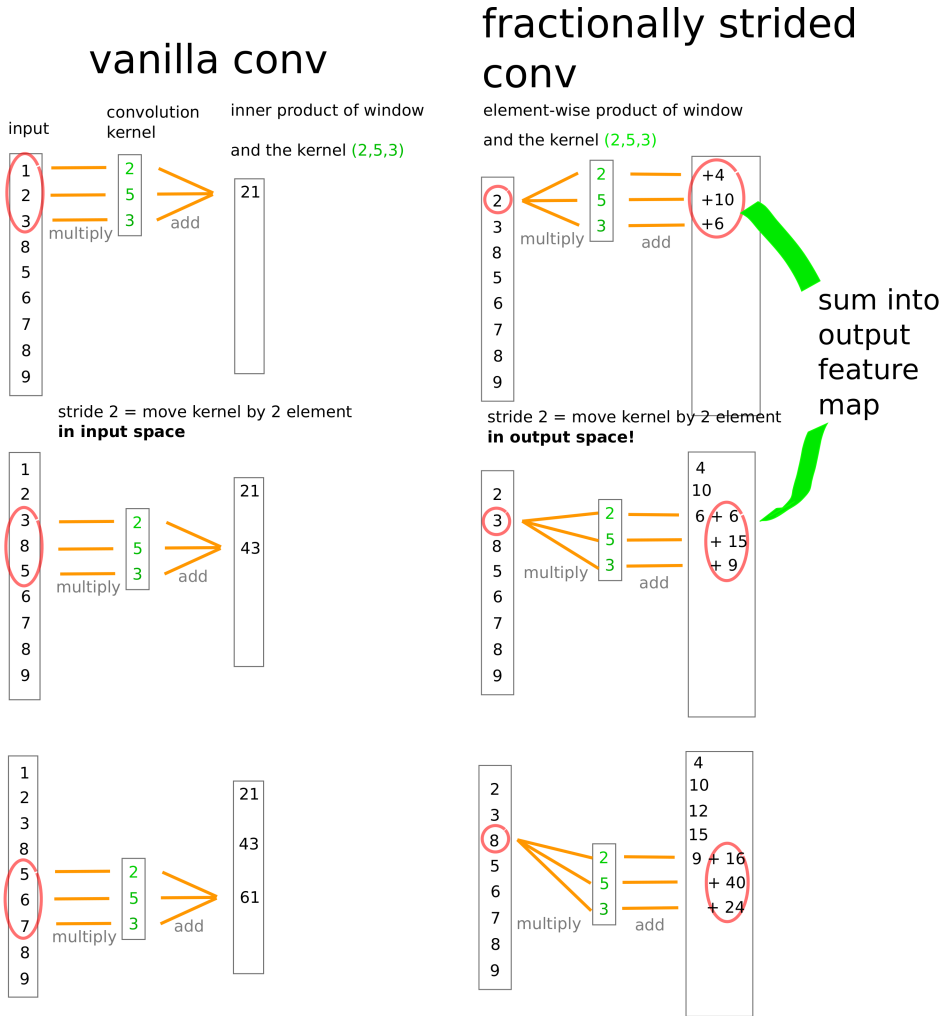
For single input channels, the working principle of a fractionally strided convolution is described below.



For multiple input channels the working principle is analogous.



Pay attention below, how a stride is applied in fractionally strided convolutions.



To summarize:

	Conventional 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

have two ways how to increase the spatial resolution

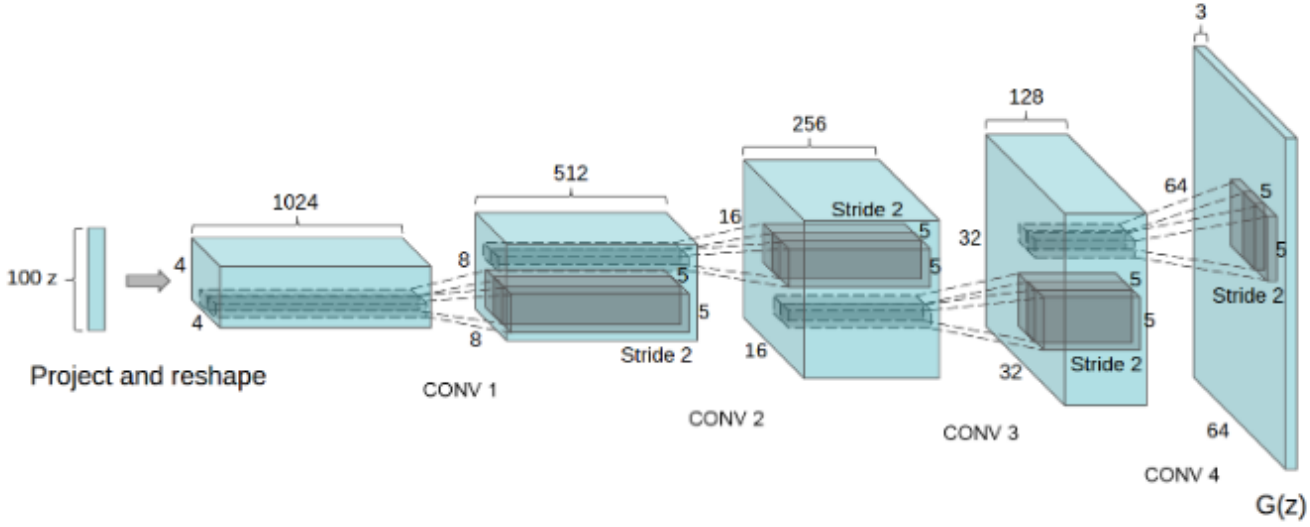


Figure 1: DCGAN generator used for LSUN scene modeling. A 100 dimensional uniform distribution Z is projected to a small spatial extent convolutional representation with many feature maps. A series of four fractionally-strided convolutions (in some recent papers, these are wrongly called deconvolutions)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

- upsampling and convolution
- frac strided convolution and usual convolution

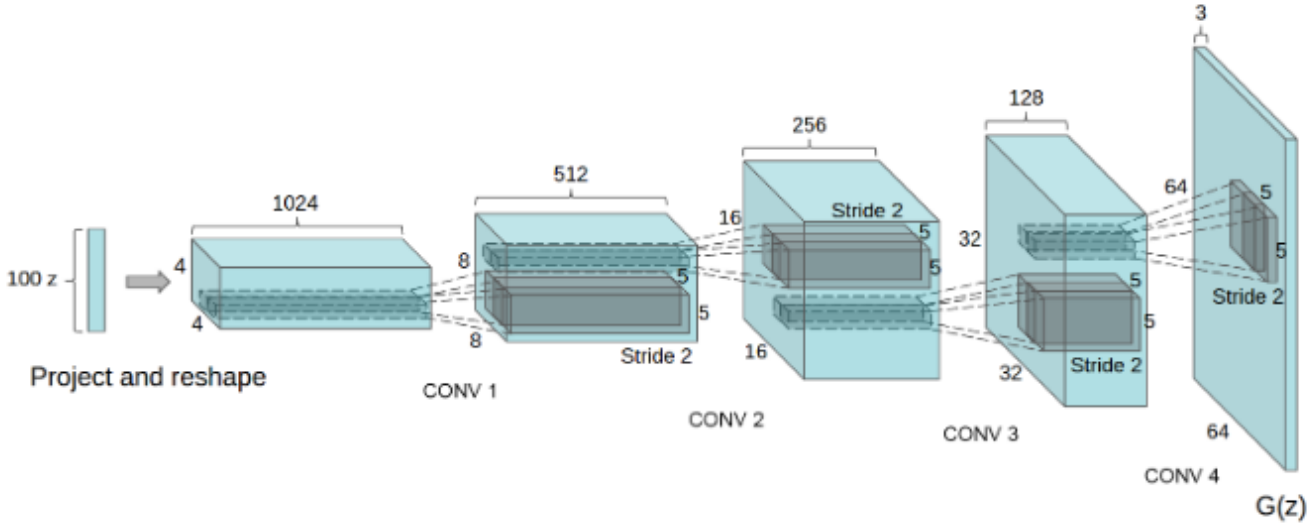


Figure 1: DCGAN generator used for LSUN scene modeling. A 100 dimensional uniform distribution Z is projected to a small spatial extent convolutional representation with many feature maps. A series of four fractionally-strided convolutions (in some recent papers, these are wrongly called deconvolutions)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

Have: a model for the generator

Next: We need a loss to train the generator

Outline

Classic GAN: loss for the generator defined via a discriminator

As mentioned, objective: train a generator that is able to generate images *"similar"* to the ones in a given dataset D_n...

To do so, we need to define a performance metric, a similarity measure, which allows us to train the generator.

Idea:

- define a classifier which classifies real versus fake images. Takes the neg log probability of its outputs as loss for the generator.
- loss function for the generator will involve a mapping defined by a neural net

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 binary classifier, with one output probability, 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: we have a pretrained and for now a 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. Similarity $\leftrightarrow$ be mapped into classifier regions used to represent images from D_n.

Discriminator outputs probabilities $D \in [0, 1]$. The objective is to train G, such that $D(G(z)) \rightarrow 1$.

$$\max_G D(G(z)) \Leftrightarrow \max_G \log D(G(z)) \Leftrightarrow \min_G -\log D(G(z))$$

Average over some generated samples:

$$\hat{L}_G = \min_G \frac{1}{N} \sum_{z_i \in minibatch(N(0, I_d))} -1 \cdot \log D(G(z_i))$$

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

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

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

- 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.
- For a fixed and pretrained discriminator, the loss function L_G is not a trivial loss function, such as $\|f(x) - y\|^2$, as it involves a a pretrained neural net $D(x)$. Still, it is a valid loss function, with its own minimization problem, that can be used to train the generator G (and define its parameters).
- 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.

Training the generator with a fixed discriminator, will usually result in outputting small variations of one single image x' (or few such candidates), 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.

Why this tradeoff exists?

Did we specify diversity in the generator loss criterion?

What is the point of talking about mode collapse ?

One partial solution: use a trainable discriminator, such that its way to predict changes in every time step. The discriminator becomes a moving target for the generator!

one (partial) solution

- also train and update the discriminator. Note: this results in an adaptive, non-static generator loss.
- 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 $\hat{L}_D$

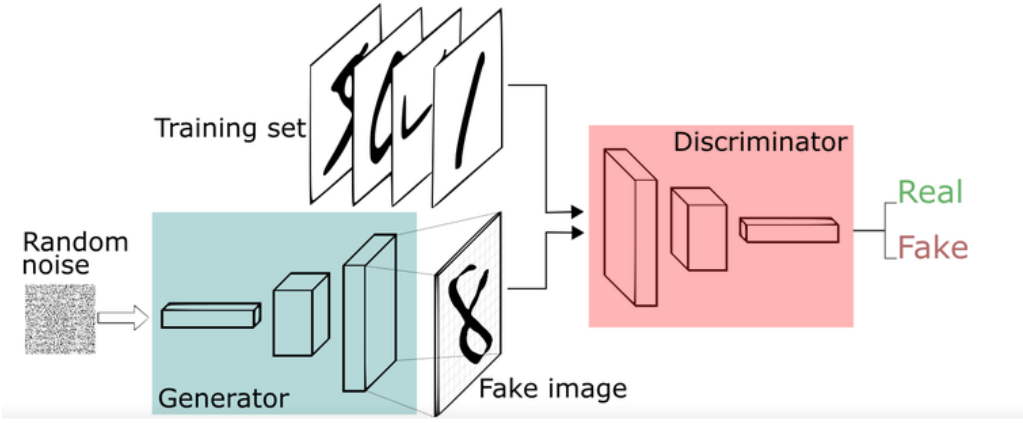


image from Thalles Silva https://medium.freecodecamp.org/an-intuitive-introduction-to-generative-adversarial-networks-gans-7a2264a81394

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

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

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

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

non-saturating GAN

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

- labels for the discriminator ? A discriminator $\leftrightarrow$ binary classification task. We know which images comes from the generator and which comes from the original dataset: we can automatically label them:
  - generated samples: label 0
  - samples drawn from D_n: label 1

thus we are ready to train the discriminator, too.

For every minibatch of data iterate two steps

- fix $w(D)$, optimize $\min_{w(G)} \hat{L}_G(w(G))$
- fix $w(G)$, optimize $\min_{w(D)} \hat{L}_D(w(D))$

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.

Training the discriminator results in a generator loss $\hat{L}_G$ which contains a neural network to represent a function D inside the loss

$$L(G) = \frac{1}{b} \sum_{v_i \sim P_v}^{b} - \ln D(G(v))$$

and which changes over iterations.

another view on the generator loss L_G:

- depends on the discriminator D network.
- example of a loss function which is trainable ($w(D)$), time-varying (updates to $w(D)$) and defined via a neural net. It is an implicitly defined loss without an explicit formula like $\frac{1}{n}\sum_{i=1}^{n}(f(x_i) - y_i)^2$
  L_D is used to train L_G.

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.

Outline

- 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= ...)
```

```
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()
```

A good tutorial, if needed:
https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html

- 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

Where to go next?

- evaluation: how to measure the quality?
- a better model I: Progressive growing of GANs
- applications I: cross-domain mappings for data samples.
- a better loss: Wasserstein-GANs
- a better model II: StyleGAN and StyleGAN2
- applications II: GAN inpainting
- a better training: small-scale training of GANs

Outline

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 loss 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
- have lower, upper bounds to scores
- 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.

The inception score $IS(G)$ uses the KL-Divergence $D_{KL}(p||q)$ between two distributions

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

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

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

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

Properties:

$$D_{KL}(p||q) \geq 0$$

$$D_{KL}(p||q) = 0 \Leftrightarrow p(x) = q(x) \text{ except on a set where } p(S) = 0$$

$$\begin{aligned} D_{KL}(p||q) &= -\sum p(x) \log q(x) - (-\sum_x p(x) \log p(x)) \\ &= H(P, Q) - H(P) \end{aligned}$$

the last says, it is the difference between the crossentropy $H(P, Q)$ and the entropy of P

Why this is not a distance measure?

The inception score $IS(G)$ (higher is better) uses the KL-Divergence $D_{KL}(p||q)$ between two distributions

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

- $p(y|x)$ 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 should be then high

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.

Evaluation of GAN outputs: inception score

(–) implementation-dependent

(–) value unclear for problems outside of imagenet

(–) weak at detecting mode collapse

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:

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

- $\|\mu_g - \mu_d\|_2^2$ 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

- Out of class: To understand the weird term $Tr(\Sigma_d + \Sigma_g - 2(\Sigma_d\Sigma_g)^{1/2})$, note:
  Denoting $x = (\Sigma_d)^{1/2},\ y = (\Sigma_g)^{1/2}$, we get:

$$Tr(\Sigma_d + \Sigma_g - 2(\Sigma_d)^{1/2}(\Sigma_g)^{1/2}) = Tr(x \star x) + Tr(y \star y) - Tr(2x \star y)$$

  if x, y were vectors, we would have:

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

  Unfortunately, $(\Sigma_d\Sigma_g)^{1/2}$ is a matrix, but taking $Tr(Z) = \sum_i Z_{ii}$ turns any matrix A into a scalar.

- What is left to understand:

$$A \cdot_S B := Tr(A \star B^\top)$$

  defines an inner product on the set of all symmetric matrices (e.g. https://math.stackexchange.com/questions/476802/how-do-you-prove-that-trbt-a-is-a-inner-product)

- Since Σ is positive definite, Σcan be decomposed as (not uniquely)

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

- then we obtain

$$\begin{aligned}
&Tr(\Sigma_d + \Sigma_g - 2(\Sigma_d)^{1/2}(\Sigma_g)^{1/2}) \\
&= Tr((\Sigma_d)^{1/2} \star (\Sigma_d)^{1/2}) + Tr(\Sigma_g^{1/2} \star \Sigma_g^{1/2}) - 2\,Tr(\Sigma_d^{1/2} \star \Sigma_g^{1/2}) \\
&= (\Sigma_d)^{1/2} \cdot_S (\Sigma_d)^{1/2} + (\Sigma_g)^{1/2} \cdot_S (\Sigma_g)^{1/2} - 2\Sigma_d^{1/2} \cdot_S \Sigma_g^{1/2} \\
&= \|\Sigma_d^{1/2} - \Sigma_g^{1/2}\|_{\cdot_S}^2
\end{aligned}$$

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

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 between the two Gaussians:

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

- 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)

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.

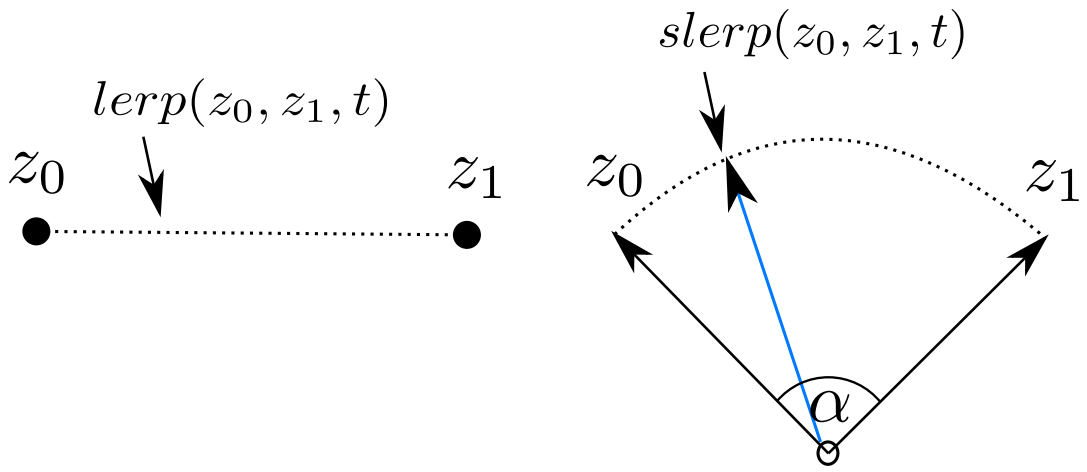
Evaluation of GAN outputs: Multiscale statistical similarity

(+) can detect mode collapse

Evaluation of GAN outputs: Perceptual Path Length (PPL)

- 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

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



Evaluation of GAN outputs: Perceptual Path Length (PPL)

- 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

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

- then take some feature map from a CNN $\phi(\cdot)$ 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

$$g_\epsilon(s) = \left\| \frac{\phi(G(s)) - \phi(G(s_{+\epsilon}))}{\epsilon} \right\|^2 \quad \text{norm}^2 \text{ of directional deriv at } \phi(G(s))$$

$$s = slerp(z_1, z_2, t), \ s_{+\epsilon} = slerp(z_1, z_2, t + \epsilon)$$

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

Evaluation of GAN outputs: Perceptual Path Length (PPL)

- 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
- then take some feature map from a CNN $\phi(\cdot)$ and look at the 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 of $g_\epsilon(s)$ over random draws of z_1, z_2, t

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

Evaluation of GAN outputs: Perceptual Path Length (PPL)

- draw z_1, z_2 two random input codes. Interpolate in code space along a circle between z_1 and z_2 using the $slerp(\cdot)$ function
- then take some feature map from a CNN $\phi(\cdot)$ and look at the norm of the directional derivative of $\phi(G(s))$ 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 of $g_\epsilon(s)$ 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

Evaluation of GAN outputs: Perceptual Path Length (PPL)

- draw z_1, z_2 two random input codes. Interpolate in code space along a circle between z_1 and z_2 using the $slerp(\cdot)$ function
- then take some feature map from a CNN $\phi(\cdot)$ and look at the norm of the directional derivative along the slerp path:

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

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

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

- lower is better. Why that makes sense? Generator seems to learn to compress spaces with bad quality outputs – compressed spaces have large derivatives in them

Evaluation of GAN outputs: Perceptual Path Length

(+) 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

Evaluation of GAN outputs: other measures

not exam stuff

- perception and recall as probability distribution overlap
- local intrinsic dimensionality

- Karras et al. ICLR 2018: https://arxiv.org/abs/1710.10196
- from same authors further development: StyleGAN https://arxiv.org/abs/1812.04948 (next lecture) and StyleGAN2
- Basic idea: do not train complex problem from scratch. Better: train a sequence of increasingly harder problems - at increasing resolutions
- reminiscent: teacher student learning, curriculum learning, layer-wise pretraining.

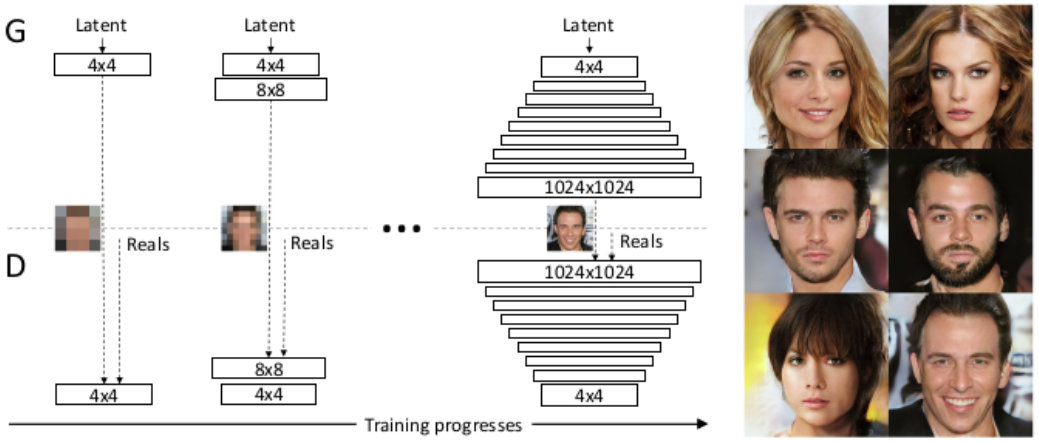
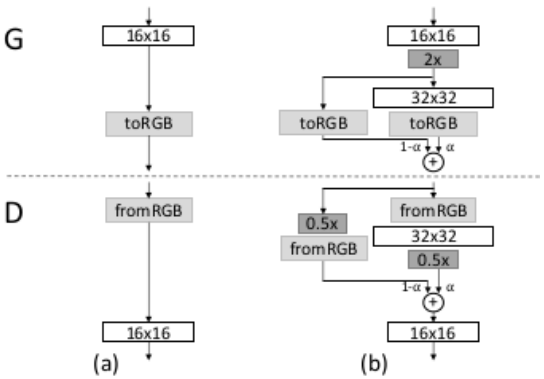


Figure 1: Our training starts with both the generator (G) and discriminator (D) having a low spatial resolution of 4×4 pixels. As the training advances, we incrementally add layers to G and D, thus increasing the spatial resolution of the generated images. All existing layers remain trainable throughout the process. Here $\boxed{N \times N}$ refers to convolutional layers operating on $N \times N$ spatial resolution. This allows stable synthesis in high resolutions and also speeds up training considerably. One the right we show six example images generated using progressive growing at 1024×1024.

from Karras et al. https://arxiv.org/abs/1710.10196

when done with a given resolution (e.g. 16×16), then add module to increase resolution to e.g. 32×32:



from Karras et al. https://arxiv.org/abs/1710.10196

- 2x nearest neighbor upscaling – increases the resolution x2

$$z_l = upscaleNN(y_{l-1})$$

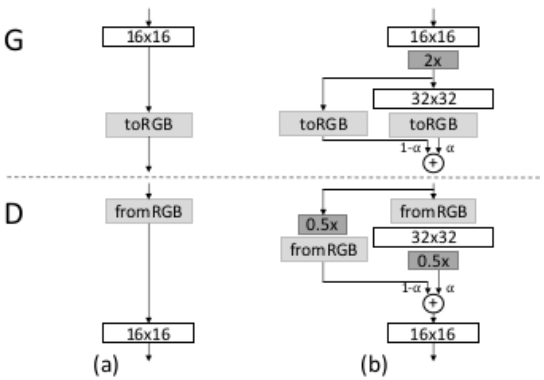
drawback: does not add any finer details!

- add $3x3$ convolution in top

$$y_l^{(*)} = conv3x3(z_l)$$

What drawback does it have when initialized?

when done with a given resolution (e.g. 16×16), then add module to increase resolution to e.g. 32×32:



from Karras et al. https://arxiv.org/abs/1710.10196

- 2x nearest neighbor upscaling – increases the resolution x2

$$z_l = upscaleNN(y_{l-1})$$

drawback: does not add any finer details!

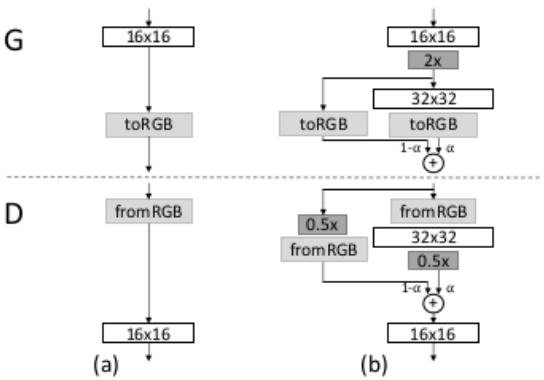
- add $3x3$ convolution in top

$$y_l^{(*)} = conv3x3(z_l)$$

What drawback does it have when initialized?

-

when done with a given resolution (e.g. 16×16), then add module to increase resolution to e.g. 32×32:



from Karras et al. https://arxiv.org/abs/1710.10196

- 2x nearest neighbor upscaling – increases the resolution x2

$$z_l = upscaleNN(y_{l-1})$$

drawback: does not add any finer details!

- add $3x3$ convolution in top

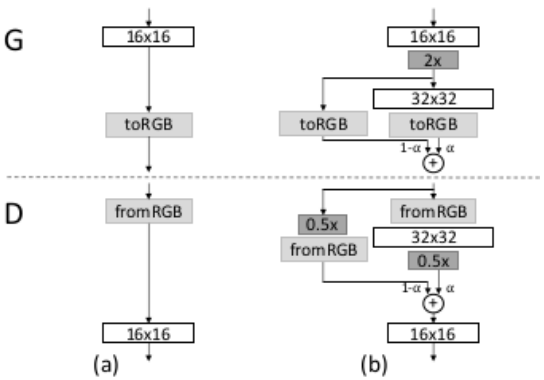
$$y_l^{(*)} = conv3x3(z_l)$$

What drawback does it have when initialized?

- initialize with weighted residual:

$$y_l = 0.01 * conv3x3(z_l) + 0.99 * z_l$$

when done with a given resolution (e.g. 16×16), then add module to increase resolution to e.g. 32×32:



from Karras et al. https://arxiv.org/abs/1710.10196

- 2x nearest neighbor upscaling – increases the resolution x2

$$z_l = upscaleNN(y_{l-1})$$

drawback: does not add any finer details!

- add $3x3$ convolution in top

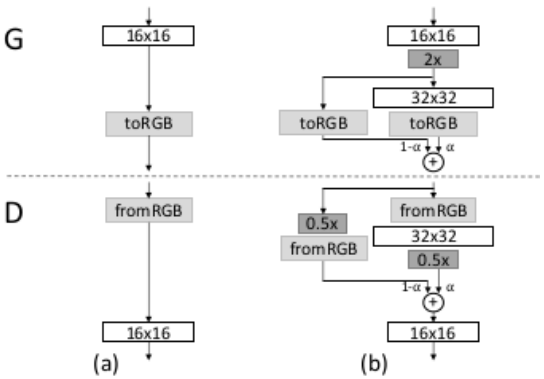
$$y_l^{(*)} = conv3x3(z_l)$$

What drawback does it have when initialized?

- increase weight of conv, decrease for residual:

$$y_l = 0.05 * conv3x3(z_l) + 0.95 * z_l$$

when done with a given resolution (e.g. 16×16), then add module to increase resolution to e.g. 32×32:



from Karras et al. https://arxiv.org/abs/1710.10196

- 2x nearest neighbor upscaling – increases the resolution x2

$$z_l = upscaleNN(y_{l-1})$$

drawback: does not add any finer details!

- add $3x3$ convolution in top

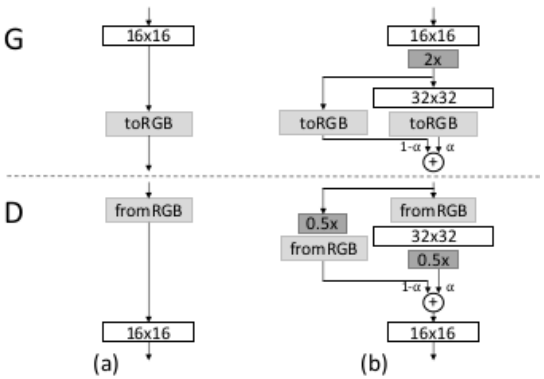
$$y_l^{(*)} = conv3x3(z_l)$$

What drawback does it have when initialized?

- increase weight of conv, decrease for residual:

$$y_l = 0.1 * conv3x3(z_l) + 0.9 * z_l$$

when done with a given resolution (e.g. 16×16), then add module to increase resolution to e.g. 32×32:



from Karras et al. https://arxiv.org/abs/1710.10196

- 2x nearest neighbor upscaling – increases the resolution x2

$$z_l = upscaleNN(y_{l-1})$$

drawback: does not add any finer details!

- add $3x3$ convolution in top

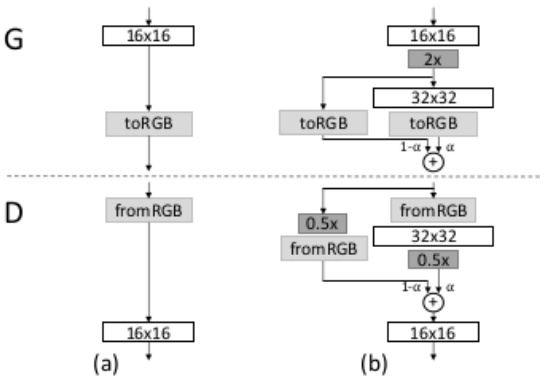
$$y_l^{(*)} = conv3x3(z_l)$$

What drawback does it have when initialized?

- increase weight of conv, decrease for residual:

$$y_l = 0.2 * conv3x3(z_l) + 0.8 * z_l$$

when done with a given resolution (e.g. 16×16), then add module to increase resolution to e.g. 32×32:



from Karras et al. https://arxiv.org/abs/1710.10196

- 2x nearest neighbor upscaling – increases the resolution x2

$$z_l = upscaleNN(y_{l-1})$$

drawback: does not add any finer details!

- add $3x3$ convolution in top

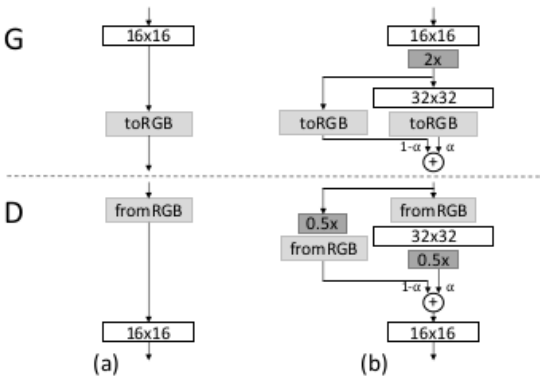
$$y_l^{(*)} = conv3x3(z_l)$$

What drawback does it have when initialized?

- increase weight of conv, decrease for residual:

$$y_l = 0.5 * conv3x3(z_l) + 0.5 * z_l$$

when done with a given resolution (e.g. 16×16), then add module to increase resolution to e.g. 32×32:



from Karras et al. https://arxiv.org/abs/1710.10196

- 2x nearest neighbor upscaling – increases the resolution x2

$$z_l = upscaleNN(y_{l-1})$$

drawback: does not add any finer details!

- add $3x3$ convolution in top

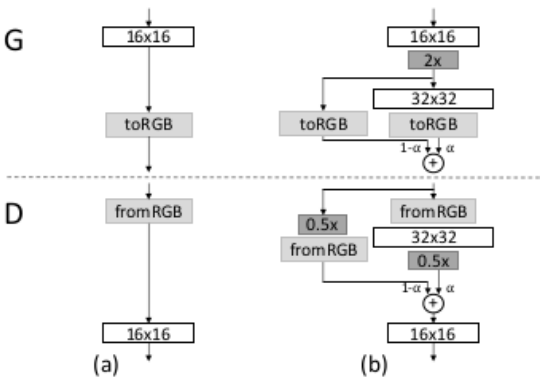
$$y_l^{(*)} = conv3x3(z_l)$$

What drawback does it have when initialized?

- increase weight of conv, decrease for residual:

$$y_l = 0.7 * conv3x3(z_l) + 0.3 * z_l$$

when done with a given resolution (e.g. 16×16), then add module to increase resolution to e.g. 32×32:



from Karras et al. https://arxiv.org/abs/1710.10196

- 2x nearest neighbor upscaling – increases the resolution x2

$$z_l = upscaleNN(y_{l-1})$$

drawback: does not add any finer details!

- add $3x3$ convolution in top

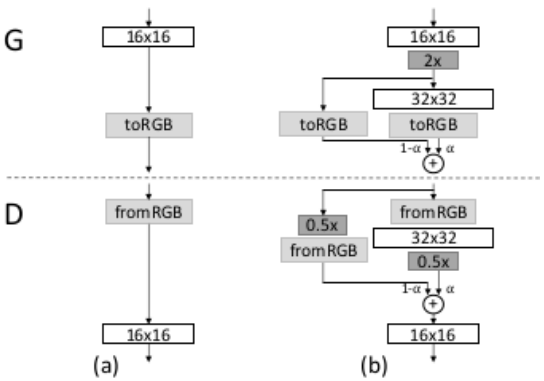
$$y_l^{(*)} = conv3x3(z_l)$$

What drawback does it have when initialized?

- increase weight of conv, decrease for residual:

$$y_l = 0.99 * conv3x3(z_l) + 0.01 * z_l$$

when done with a given resolution (e.g. 16×16), then add module on top of the generator:



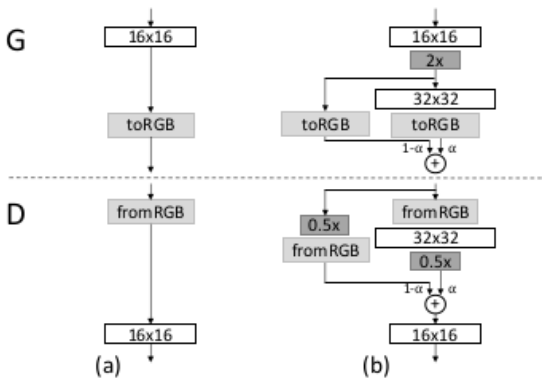
from Karras et al. https://arxiv.org/abs/1710.10196

- 2x nearest neighbor upscaling – increases the resolution x2
- on top of that: a 3x3-convolution weighted with a factor α. Added to it: the feature map from the 2x upscaling, weighted with a factor $1 - \alpha$

$$z_l = upscaleNN(y_{l-1})$$
$$y_l = \alpha conv3x3(z_l) + (1 - \alpha) z_l$$

- phasing in of conv3x3 during training: start with $\alpha = 0$ – have only the 2x upscaling, increase until $\alpha = 1$, then the resulting mapping is only concat of the upscaling and the conv3x3: $y_l = conv3x3(upscaleNN(y_{l-1}))$

when done with a given resolution add in on top of the discrimator:



from Karras et al. https://arxiv.org/abs/1710.10196

- start: stride2 avg pooling – decreases the resolution x0.5
- want: a 3x3-convolution, then avgpool to scale down
- weighted both with a factors α and $1-\alpha$

  $z_l = avgpool2(y_{l-1})$

  $y_l = \alpha\ avgpool2(conv3x3(y_{l-1})) + (1-\alpha)z_l$

- phasing in of conv3x3 during training:
  start with $\alpha = 0$ – have only the stride2 avgpool,
  increase until $\alpha = 1$, after that the resulting mapping is only:
  $y_l = avgpool2(conv3x3(y_{l-1}))$

Phasing in new network layers using weight α in $D(\cdot)$ and $G(\cdot)$

several additional tricks (2 shown here)

- feature maps: normalize in every feature map the vector consisting of all channels for one spatial position aka pixel (h, w) by

$$f[c,h,w] = \frac{f[c,h,w]}{\sqrt{\frac{1}{C}\sum_{c=1}^{C} f[c,h,w]^2 + \epsilon}}, \ \epsilon = 10^{-8}$$

- weights: in every forward pass use rescaled weights $w_{sc} = w/k$ where $k = \sqrt{CHW/2}$ is the constant of the He-initializer https://arxiv.org/abs/1502.01852 (satellite paper to resnets).

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

Training configuration	CELEBA						LSUN BEDROOM					
	Sliced Wasserstein distance $\times 10^3$					MS-SSIM	Sliced Wasserstein distance $\times 10^3$					MS-SSIM
	128	64	32	16	Avg		128	64	32	16	Avg	
(a) Gulrajani et al. (2017)	12.99	7.79	7.62	8.73	9.28	0.2854	11.97	10.51	8.03	14.48	11.25	**0.0587**
(b) + Progressive growing	4.62	**2.64**	3.78	6.06	4.28	**0.2838**	7.09	6.27	7.40	9.64	7.60	0.0615
(c) + Small minibatch	75.42	41.33	41.62	26.57	46.23	0.4065	72.73	40.16	42.75	42.46	49.52	0.1061
(d) + Revised training parameters	9.20	6.53	4.71	11.84	8.07	0.3027	7.39	5.51	3.65	9.63	6.54	0.0662
(e*) + Minibatch discrimination	10.76	6.28	6.04	16.29	9.84	0.3057	10.29	6.22	5.32	11.88	8.43	0.0648
(e) Minibatch stddev	13.94	5.67	2.82	5.71	7.04	0.2950	7.77	5.23	3.27	9.64	6.48	0.0671
(f) + Equalized learning rate	4.42	3.28	2.32	7.52	4.39	0.2902	**3.61**	3.32	**2.71**	6.44	4.02	0.0668
(g) + Pixelwise normalization	**4.06**	3.04	**2.02**	**5.13**	**3.56**	0.2845	3.89	**3.05**	3.24	**5.87**	**4.01**	0.0640
(h) Converged	2.42	2.17	2.24	4.99	2.96	0.2828	3.47	2.60	2.30	4.87	3.31	0.0636

Table 1: Sliced Wasserstein distance (SWD) between the generated and training images (Section 5) and multi-scale structural similarity (MS-SSIM) among the generated images for several training setups at 128×128. For SWD, each column represents one level of the Laplacian pyramid, and the last one gives an average of the four distances.

from Karras et al. https://arxiv.org/abs/1710.10196

from Karras et al. https://arxiv.org/abs/1710.10196

from Karras et al. https://arxiv.org/abs/1710.10196

the variability in the data impacts the quality. Idea: Want to train a GAN on ...
?

from Karras et al. https://arxiv.org/abs/1710.10196 ... Did I just spot ...?

ones sees two similarly looking people

Real Life: you must be relatives somehow (Wow, genes can recombine and produce such a little wonder)
GANs: you must be fakes and the algorithm sucks

ones sees two similarly looking people

Real Life: you must be relatives somehow (Wow, genes can recombine and produce such a little wonder)

GANs: you must be fakes and the algorithm sucks

ones sees two similarly looking people

Real Life: you must be relatives somehow (Wow, genes can recombine and produce such a little wonder)
GANs: you must be fakes and the algorithm sucks

Will people treat contacts with less respect if they know that the contacts are non-real avatars coupled with an conversational algorithm?

- You worthless conversational bot! (I have no one else to look down upon.)
- Would verbal aversion against bots affect self-learning bots?
- Do you know where the word “sabotage” originates from?

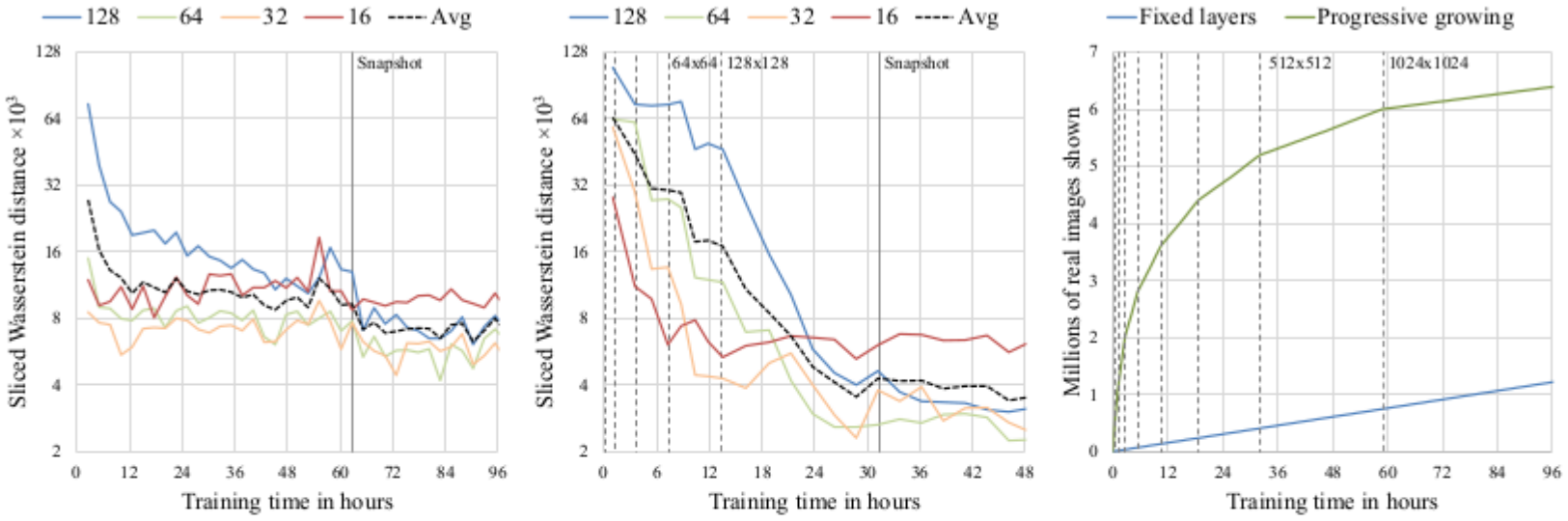


Figure 4: Effect of progressive growing on training speed and convergence. The timings were measured on a single-GPU setup using NVIDIA Tesla P100. (a) Statistical similarity with respect to wall clock time for Gulrajani et al. (2017) using CELEBA at 128×128 resolution. Each graph represents sliced Wasserstein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

from Karras et al. https://arxiv.org/abs/1710.10196

- a trained generator $G(\cdot)$, some feature space encoder $f(\cdot)$, Take 2 images, preprocess+scale them (to 64 $\times$64 , 1000 $\times$ 1000)
- find z_1, z_2 by

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

  I suggest multiple starting points $z^{(0)}$ for starting the optimization.

- 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:

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

- Visualize your interpolated results

Unsupervised Cross Domain Generation

https://arxiv.org/abs/1611.02200 - maps faces to emoji without ground truth labels for what to map on what. Special case here: target domain is included in the source domain.

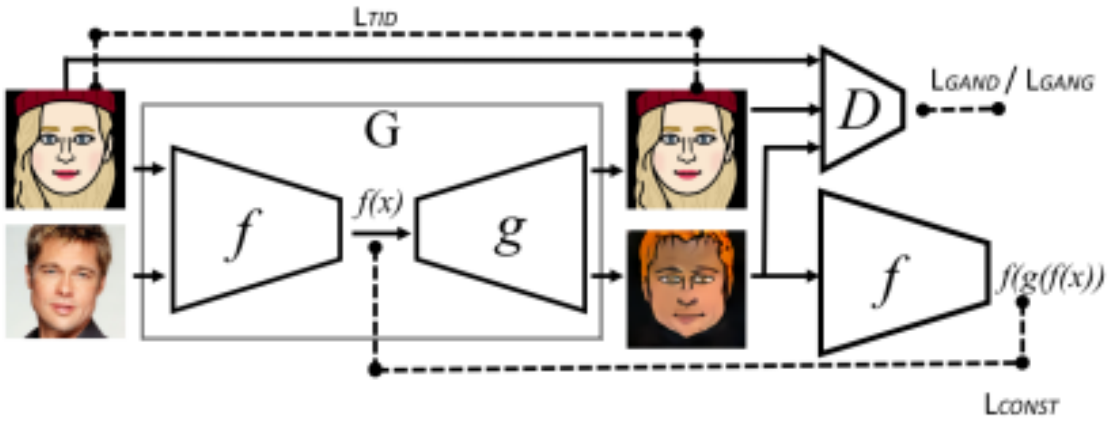


Figure 1: The Domain Transfer Network. Losses are drawn with dashed lines, input/output with solid lines. After training, the forward model G is used for the sample transfer.

Tricks:

- mapping $g \circ f$ is decoder-encoder with decoder g and encoder f. Here the encoder is **pretrained**, not trained as feature layer from a well-trained discriminative neural network. Success comes from this very well given encoder.

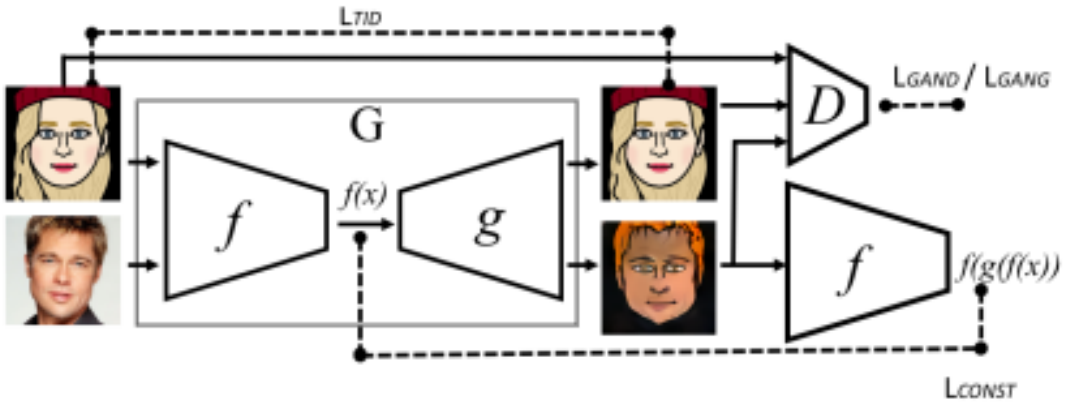


Figure 1: The Domain Transfer Network. Losses are drawn with dashed lines, input/output with solid lines. After training, the forward model G is used for the sample transfer.

Tricks:

- source domain includes the target domain!
- Plus additional constraints: eq (6) an emoji is mapped to almost itself,
- eq(5): a similar autoencoder constraint (compare: one side of a cycle gan) - measured in encoder feature space
- 3-class GAN loss (eq 3) (original data, fake face, fake emoji)

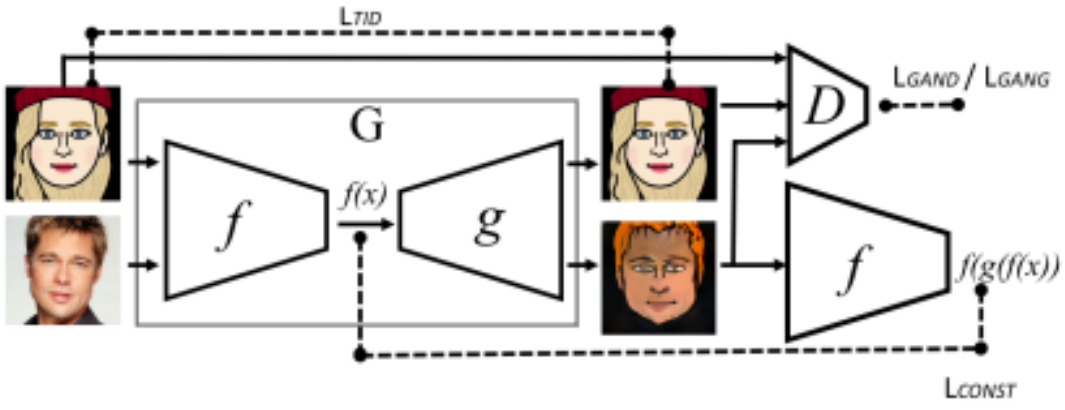


Figure 1: The Domain Transfer Network. Losses are drawn with dashed lines, input/output with solid lines. After training, the forward model G is used for the sample transfer.

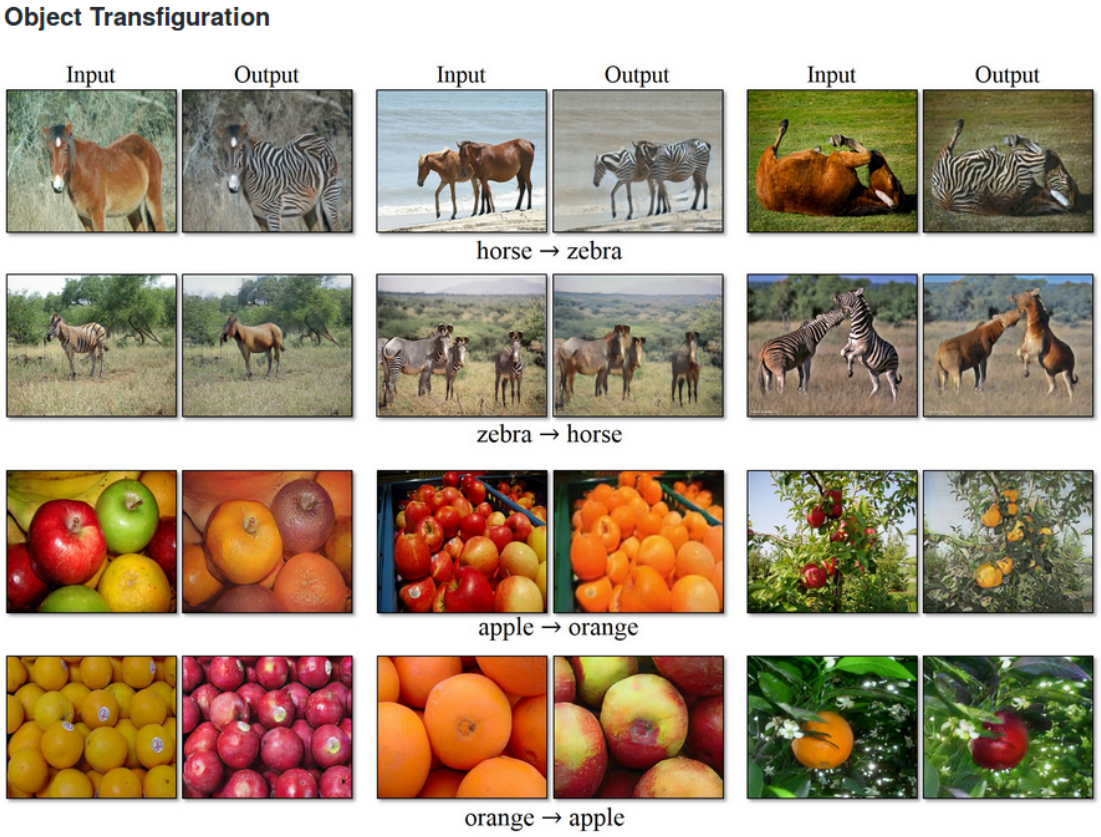
Tricks:

- Implicit tricks: Target domain is more smooth than source, variability of results is more likely acceptable (the other way round is much harder to achieve).

Sketches to Shoes, Handbags to Shoes, Male to Female and the other way round: https://arxiv.org/abs/1706.00826

Outline

Core idea: do not input some abstract code $z \in [-1, +1]^D$ into the generator for getting a zebra. Input an image $z = \mathit{Image}$ into the generator to get a zebra! – see: https://arxiv.org/abs/1703.10593



from https://github.com/junyanz/pytorch-CycleGAN-and-pix2pix, run the code to get a feeling for the quality of results!

See Fig 3 in https://arxiv.org/abs/1703.10593: you now have a generator G mapping from horses to zebras, and one generator F mapping back.

$$G : Horses \to Zebras$$
$$F : Zebras \to Horses$$
$$F \circ G : Zebras \to Zebras$$
$$G \circ F : Horses \to Horses$$

Now you can map Horses onto Zebras and back using the two generators (by training the 2 GANs)!

In practice Domain to Domain Transfer requires additional constraints to make it work in practice![1]

[1] This is what I expect you to understand here!

How to train?

clear idea: GAN losses for generators for both domains + cycle reconstruction losses for both directions.

Cycle reconstruction losses for both directions: eq(2) below. Additional constraint: take an image x, map it to another domain by F, and map it back by G, should result in something similar:

$$F \circ G(x) \approx x$$

$$\|F \circ G(x) - x\|_1 = \text{additional loss term to added to the loss}$$

What kind of training uses as loss: $F \circ G$ such that $\|F \circ G(x) - x\|_1$?

Full objective is in eq(3):

$$\mathcal{L}_{\text{cyc}}(G, F) = \mathbb{E}_{x \sim p_{\text{data}}(x)}[\|F(G(x)) - x\|_1] + \mathbb{E}_{y \sim p_{\text{data}}(y)}[\|G(F(y)) - y\|_1]. \quad (2)$$

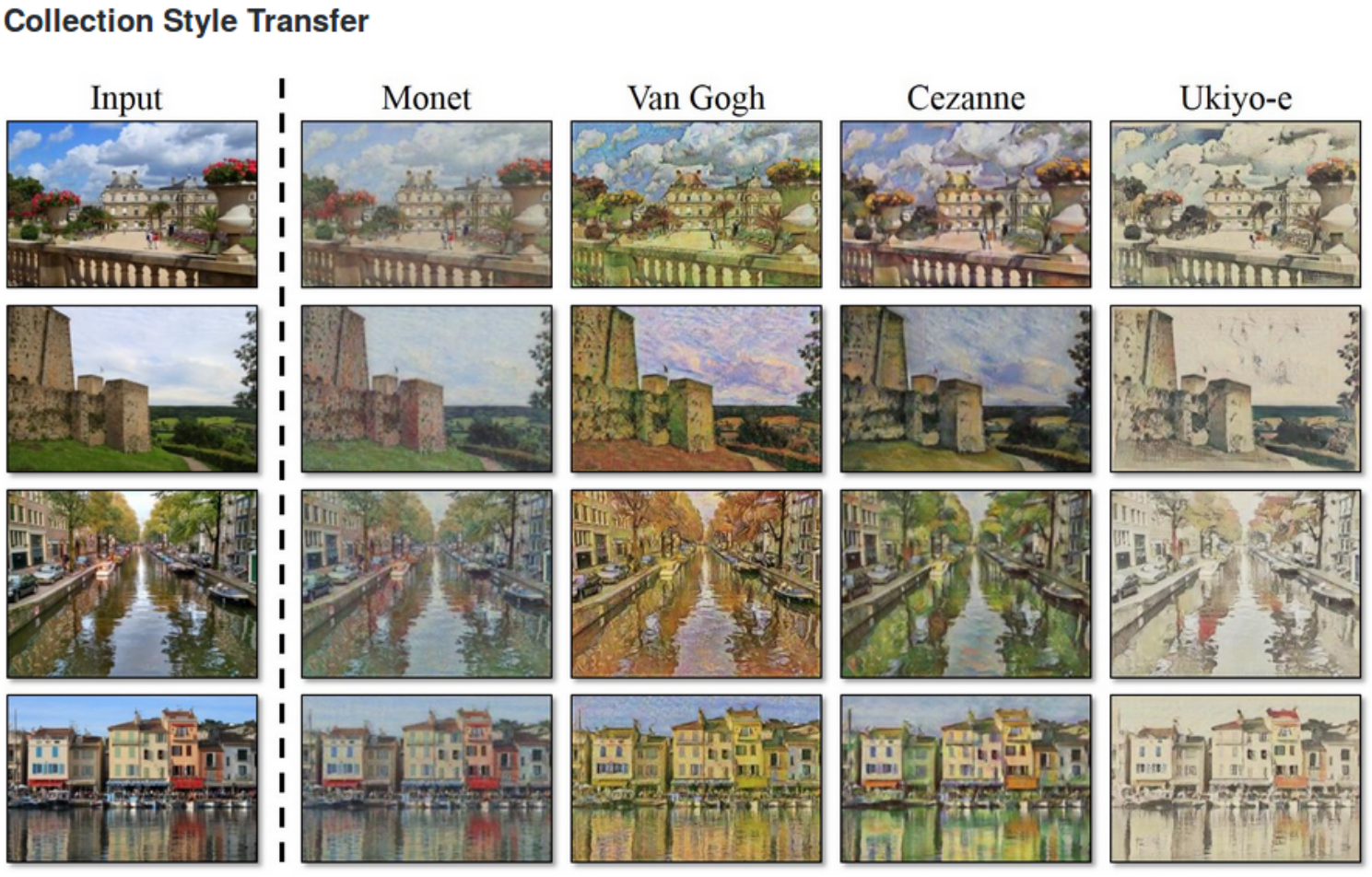
$$\mathcal{L}(G, F, D_X, D_Y) = \mathcal{L}_{\text{GAN}}(G, D_Y, X, Y) + \mathcal{L}_{\text{GAN}}(F, D_X, Y, X) + \lambda \mathcal{L}_{\text{cyc}}(G, F), \quad (3)$$

Season Transfer

winter Yosemite → summer Yosemite

summer Yosemite → winter Yosemite

Pictures from https://github.com/junyanz/CycleGAN



Pictures from https://github.com/junyanz/CycleGAN

Question: For what applications is CycleGAN a bad idea?

Questions?!