Generative Adversarial Networks (GAN)

Alexander Binder

University of Oslo (UiO)

3 May, 2022



UiO : Department of Informatics
University of Oslo

Outline

1. a better model - Progressive Growing
2. Application II – Cross Domain mappings
3. StyleGan
4. Training GANs with smaller dataset sizes
5. Inpainting to make GANs useful

- Karras et al. ICLR 2018: https://arxiv.org/abs/1710.10196
- from same authors further development: StyleGAN https://arxiv.org/abs/1812.04948 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.

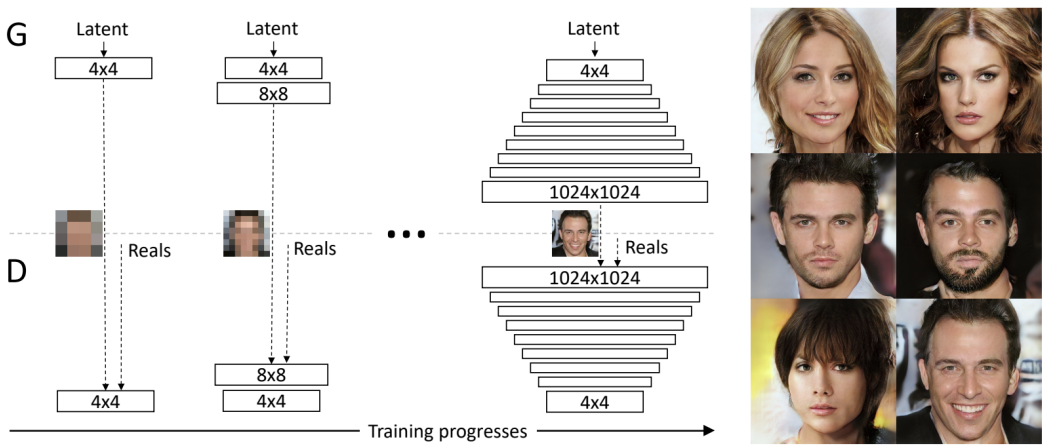
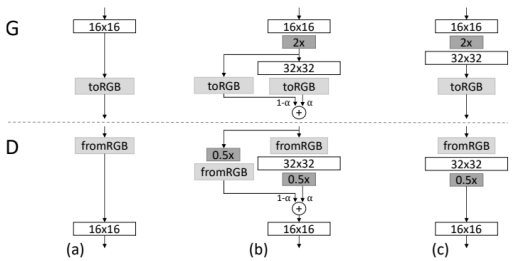


Figure 1: Our training starts with both the generator (G) and discriminator (D) having a low spatial resolution of 4×4 pixels. As the training advances, we incrementally add layers to G and D, thus increasing the spatial resolution of the generated images. All existing layers remain trainable throughout 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

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

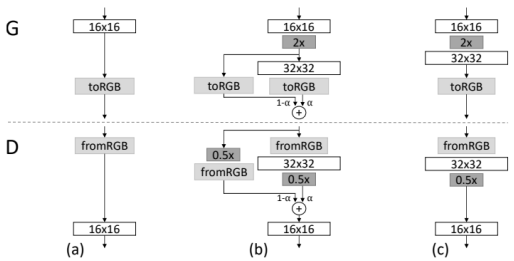
drawback: does not add any finer details!

- 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!

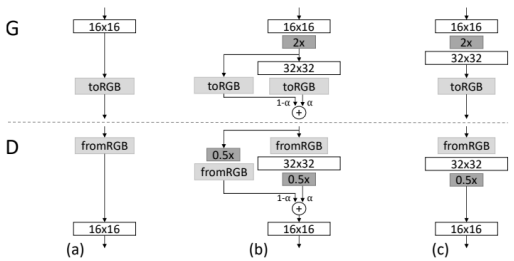
- add $3x3$ convolution in top

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

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

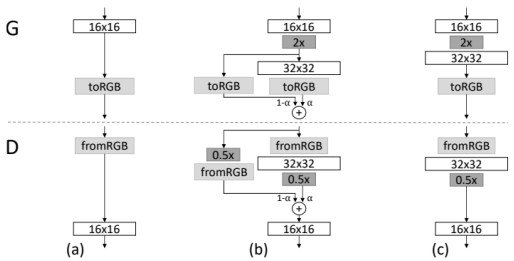
$$y_l^{(*)} = conv3x3(z_l)$$

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

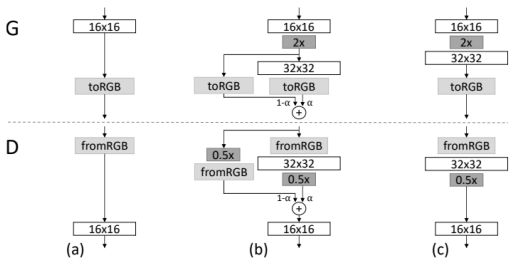
$$y_l^{(*)} = conv3x3(z_l)$$

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

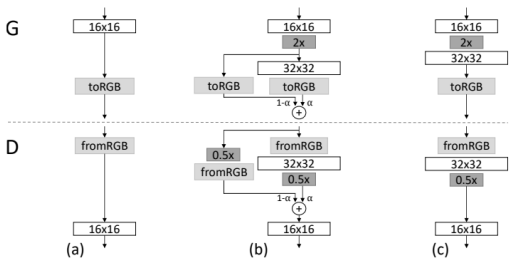
$$y_l^{(*)} = conv3x3(z_l)$$

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

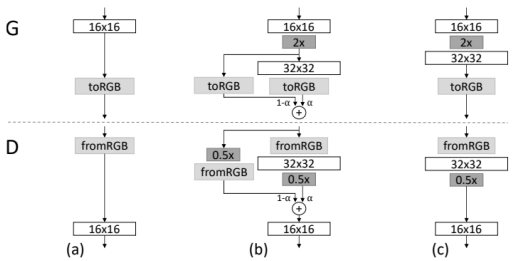
$$y_l^{(*)} = conv3x3(z_l)$$

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

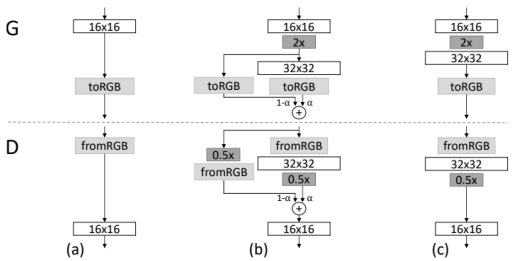
$$y_l^{(*)} = conv3x3(z_l)$$

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

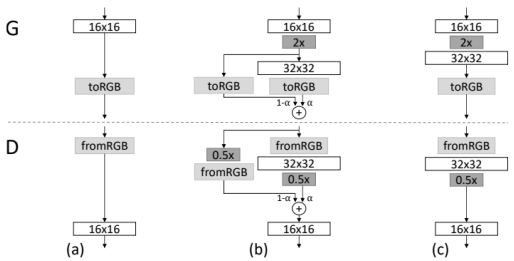
$$y_l^{(*)} = conv3x3(z_l)$$

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

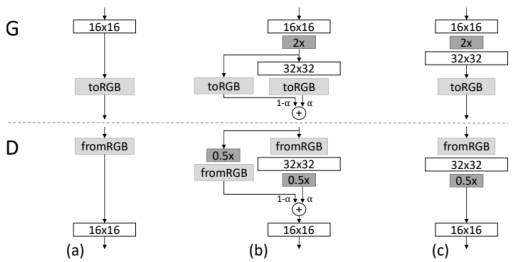
$$y_l^{(*)} = conv3x3(z_l)$$

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:



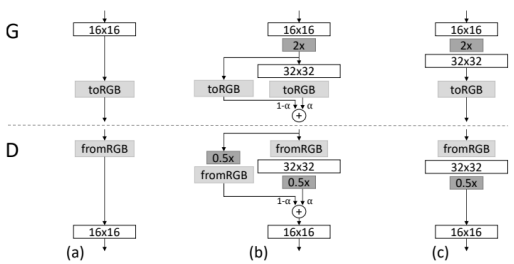
from Karras et al. https://arxiv.org/abs/1710.10196

- 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 $1 - \alpha$

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

- 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: $y_l = \mathit{conv3x3}(\mathit{upscaleNN}(y_{l-1}))$

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

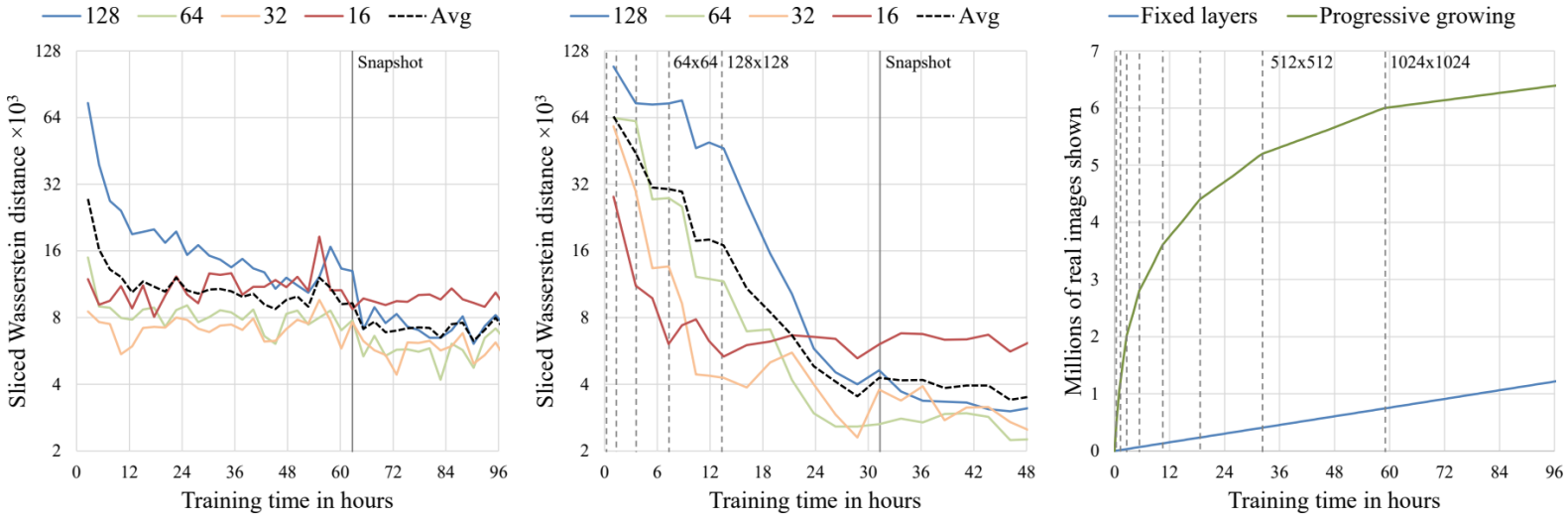


Figure 4: Effect of progressive growing on training speed and convergence. The timings were measured on a single-GPU setup using NVIDIA Tesla P100. (a) Statistical similarity with respect to wall clock time for Gulrajani et al. (2017) using CELEBA at 128 × 128 resolution. Each graph represents sliced 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.

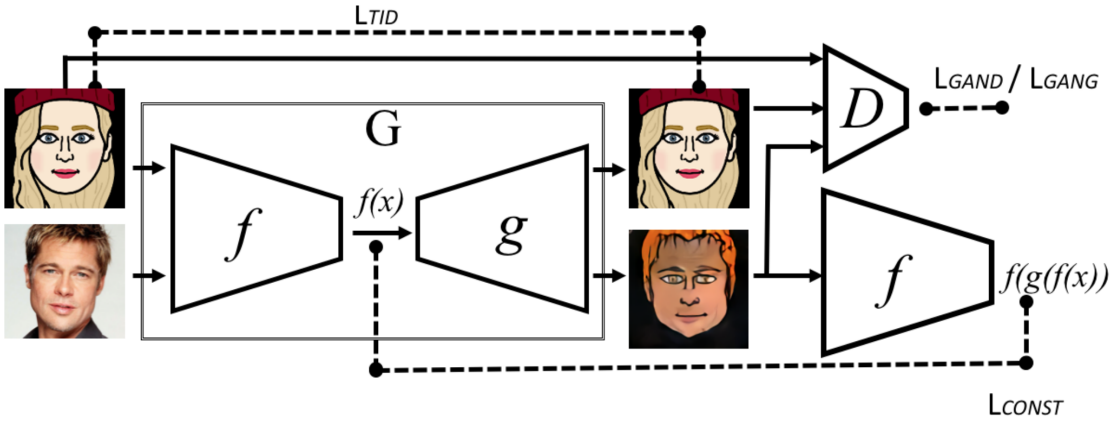


Figure 1: The Domain Transfer Network. Losses are drawn with dashed lines, input/output with solid lines. After training, the forward model G is used for the sample transfer.

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.

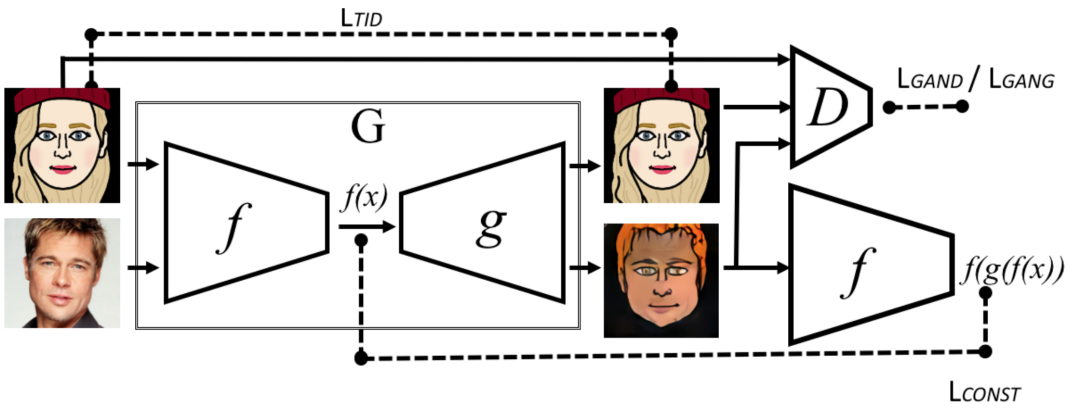


Figure 1: The Domain Transfer Network. Losses are drawn with dashed lines, input/output with solid lines. After training, the forward model G is used for the sample transfer.

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)

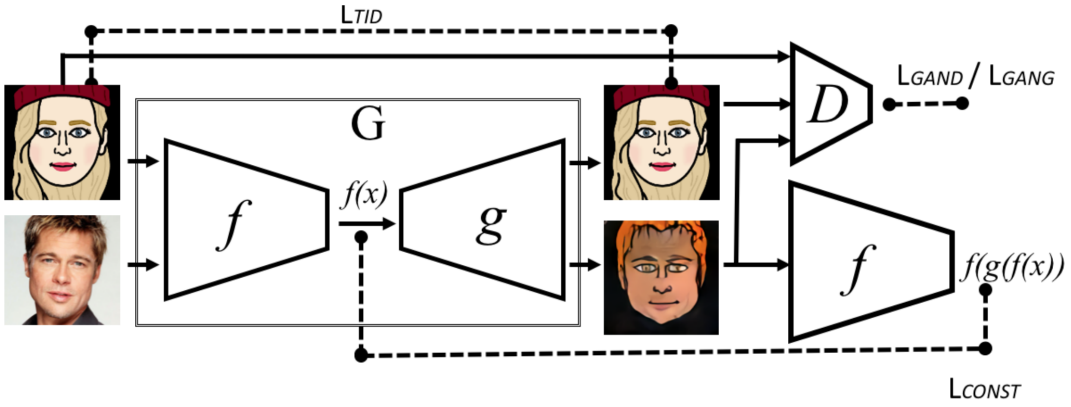


Figure 1: The Domain Transfer Network. Losses are drawn with dashed lines, input/output with solid lines. After training, the forward model G is used for the sample transfer.

Tricks:

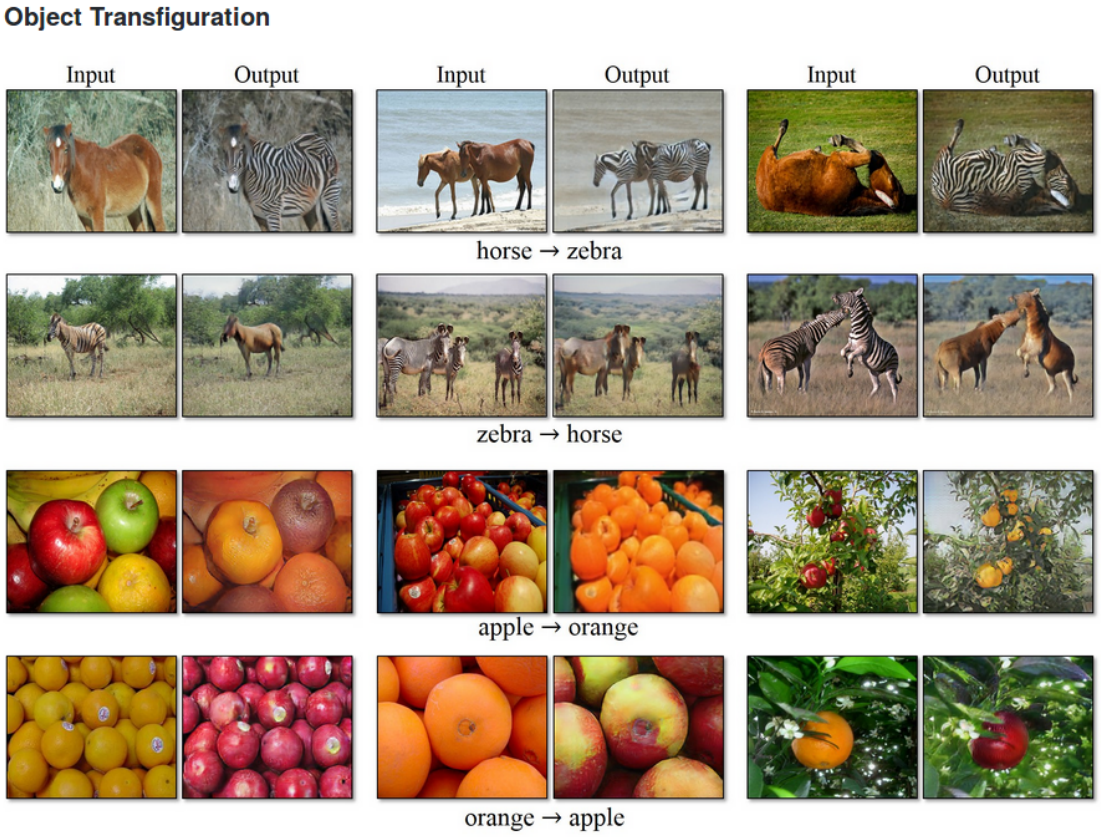
- Implicit tricks: Target domain is more smooth than source, variability of results is more likely acceptable (the other way round is much harder to achieve).

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

Outline

1. a better model - Progressive Growing
2. Application II – Cross Domain mappings
3. StyleGan
4. Training GANs with smaller dataset sizes
5. Inpainting to make GANs useful

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):

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

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

Application - Cross Domain mappings via CycleGAN

Season Transfer



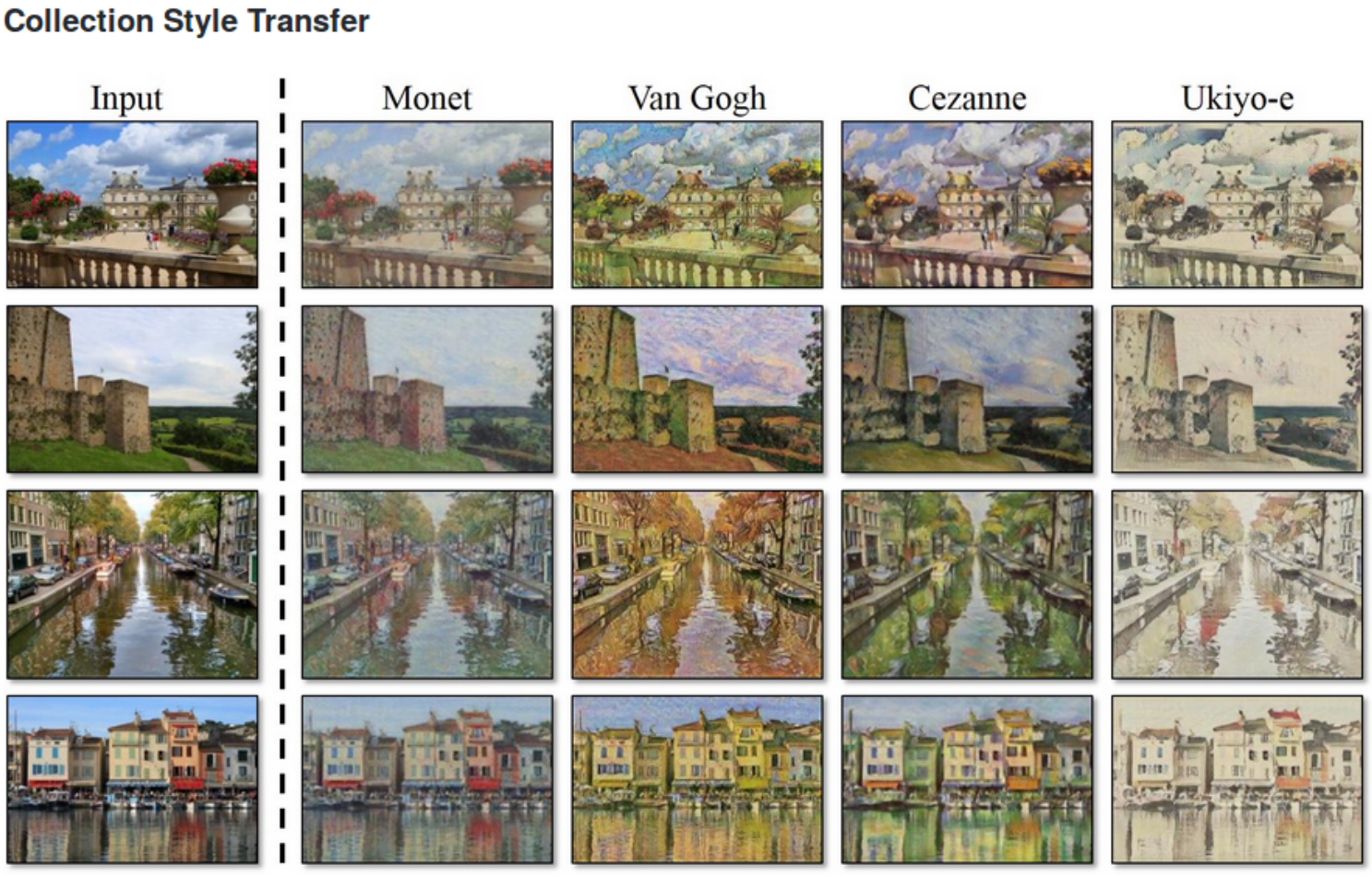
winter Yosemite → summer Yosemite



summer Yosemite → winter Yosemite

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

Application - Cross Domain mappings via CycleGAN



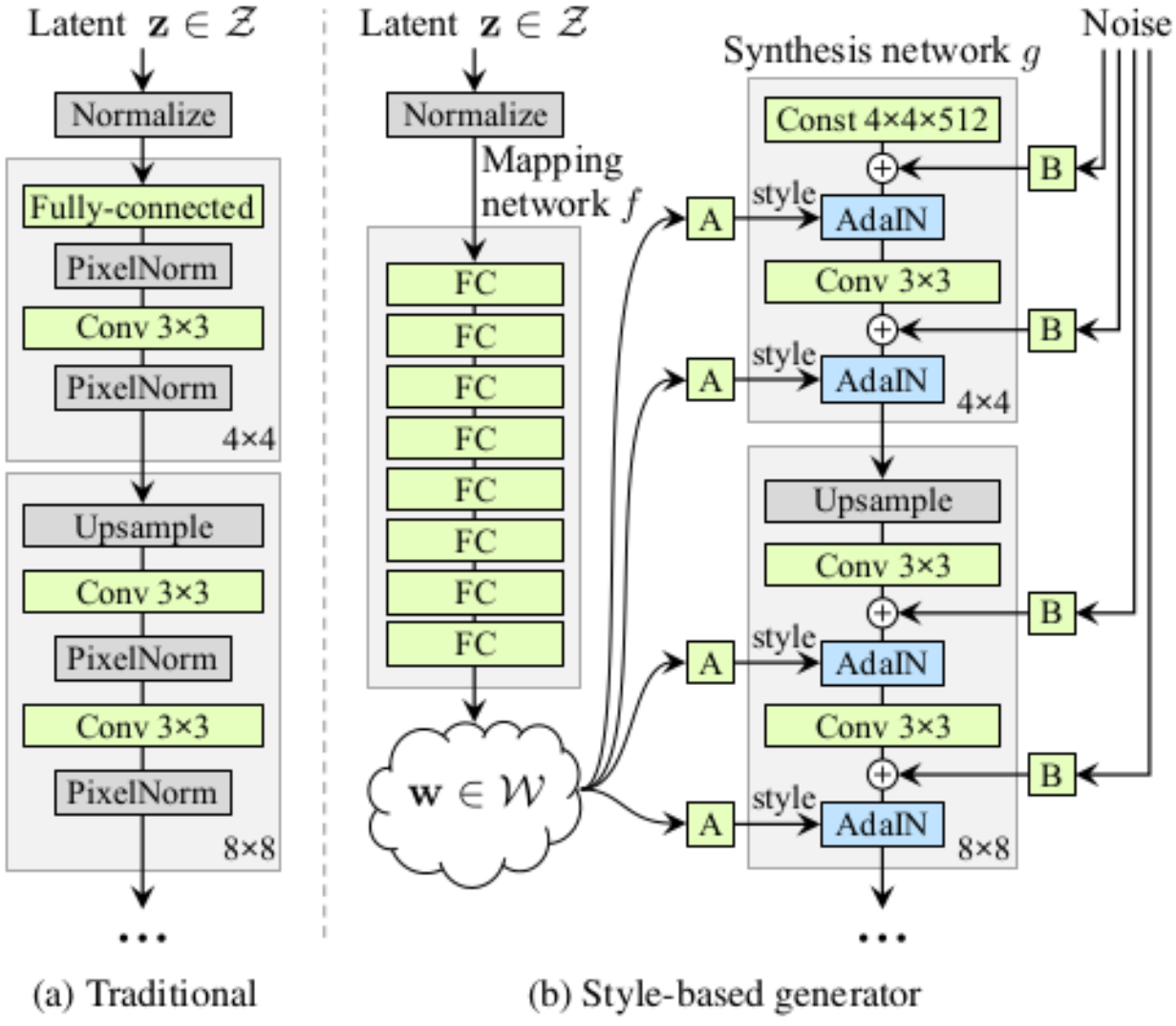
Pictures from https://github.com/junyanz/CycleGAN

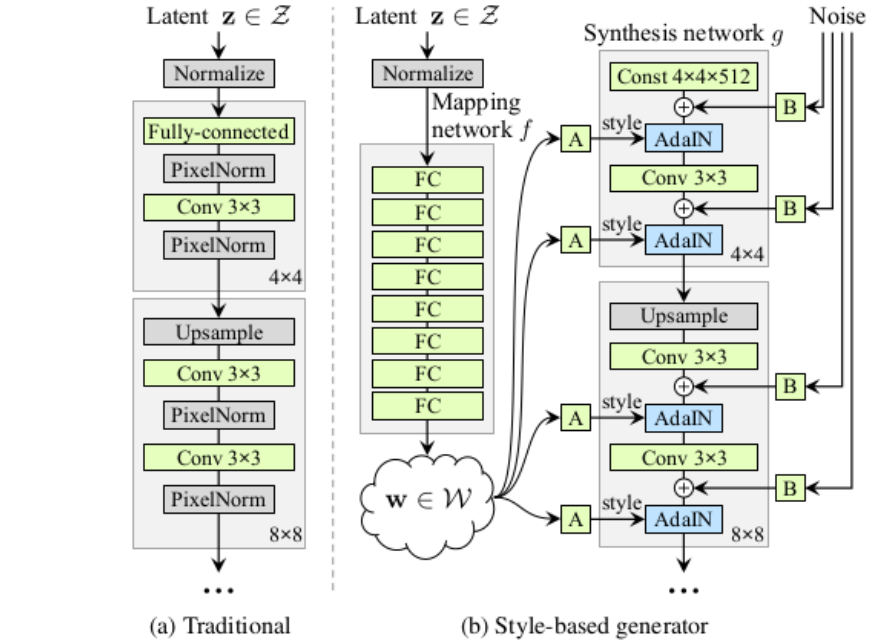
Question: For what applications is CycleGAN a bad idea?

Outline

1. a better model - Progressive Growing
2. Application II – Cross Domain mappings
3. StyleGan
4. Training GANs with smaller dataset sizes
5. Inpainting to make GANs useful

Karras et al. https://arxiv.org/abs/1812.04948





(a) Traditional (b) Style-based generator

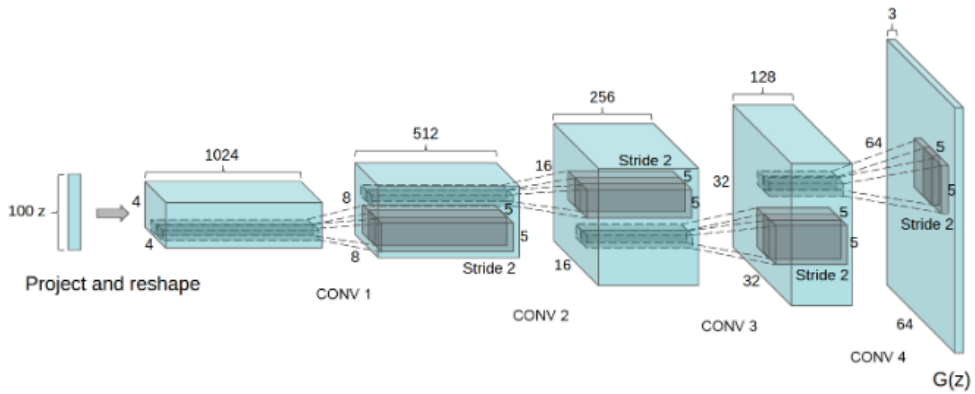
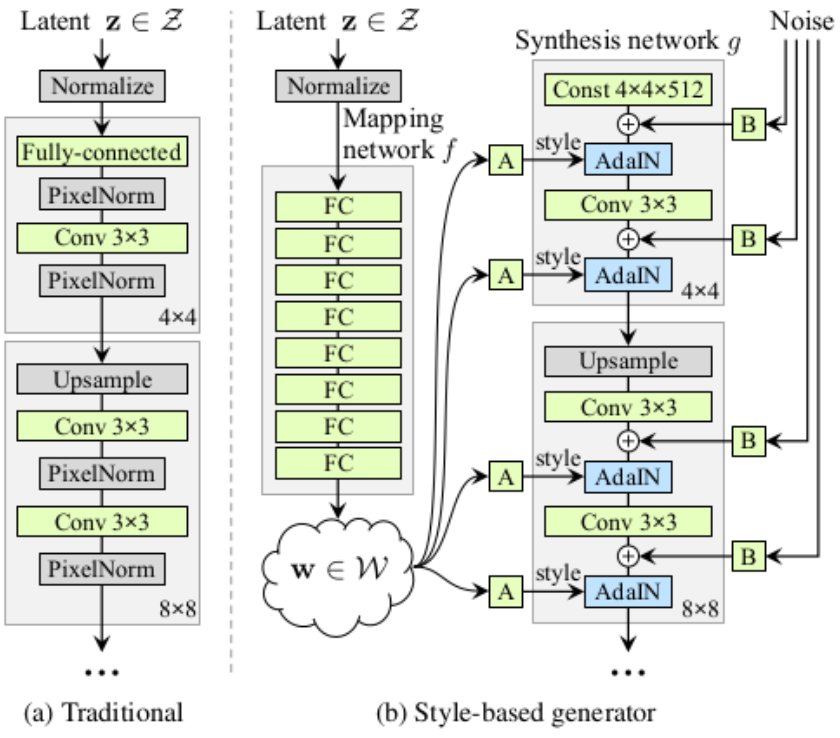
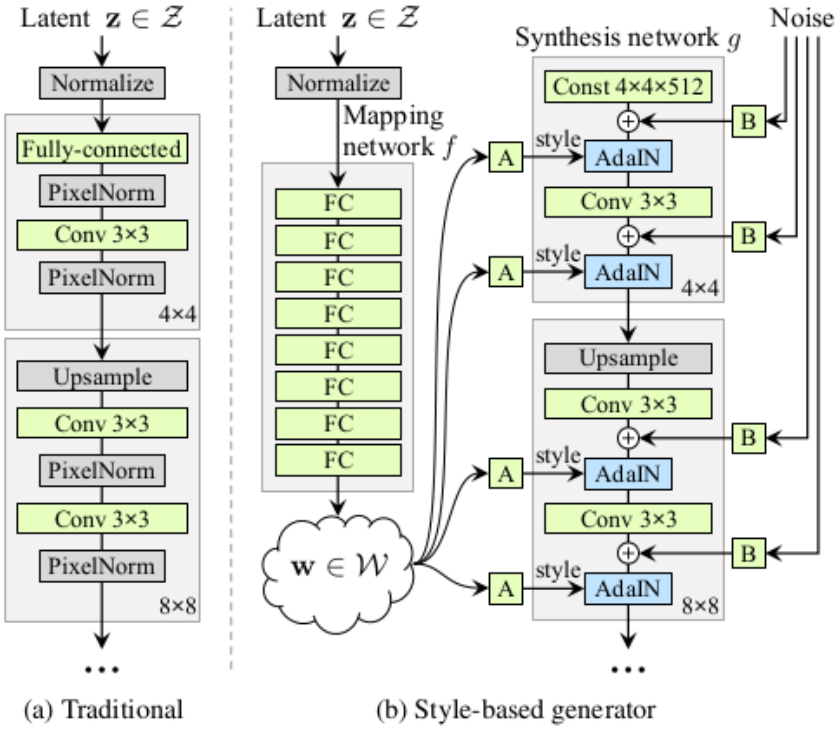


Figure 1: DCGAN generator used for LSUN scene modeling. A 100 dimensional uniform distribution Z is projected to a small spatial extent convolutional representation with many feature maps. A series of four fractionally-strided convolutions (in some recent papers, these are wrongly called deconvolutions)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



Karras et al. https://arxiv.org/abs/1812.04948

- first idea: do not input a latent vector to get a random output.
- Instead add random noise (B) to every feature map.
- Reason: randomness from input vector can get smoothed out as it passes through many layers



Karras et al. https://arxiv.org/abs/1812.04948

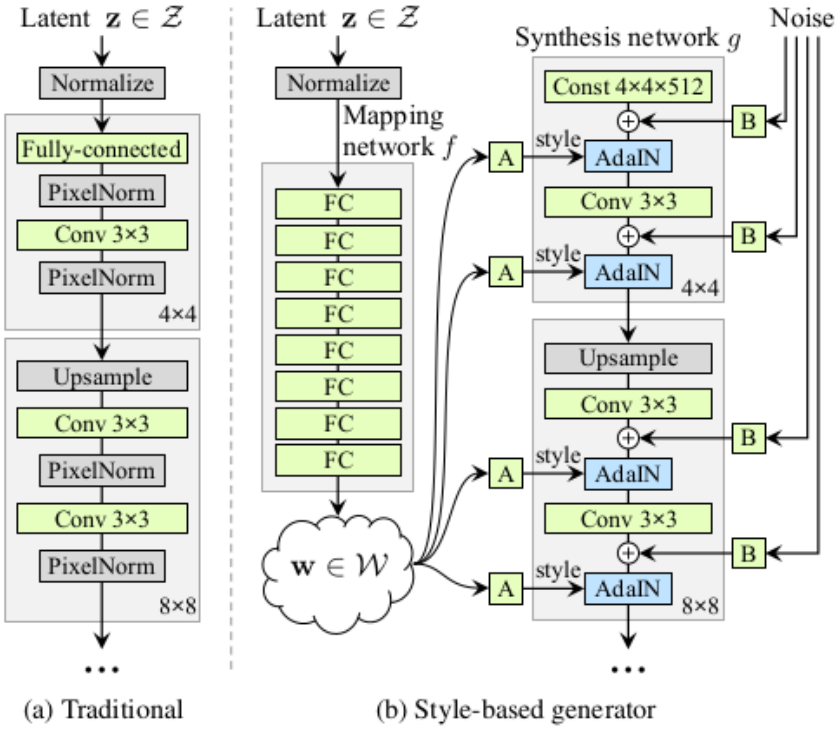
- second big idea: input styles using adaptive instance normalization

$$AI(x, y) = y_s \frac{x - \mu(x)}{\sigma(x)} + y_b$$

$$AI(x, y) = y_s \frac{x - \mu(x)}{\sigma(x)} + y_b$$

Difference to batchnormalization?

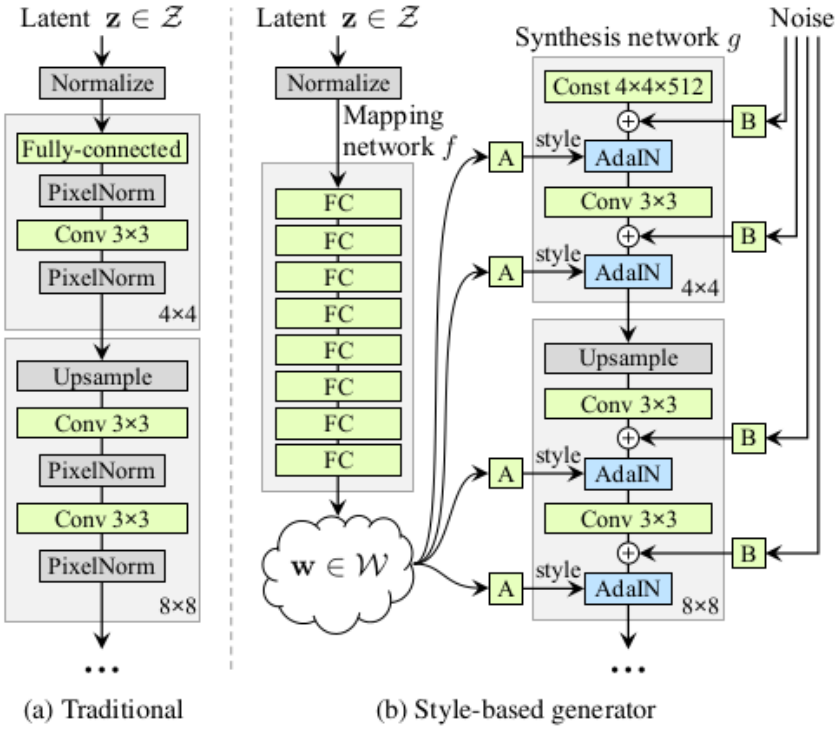
- batchsize$= 1$, same as layer normalization
- y_s, y_b are not directly trainable parameters as in BN but come as inputs from a trainable network



(a) Traditional (b) Style-based generator

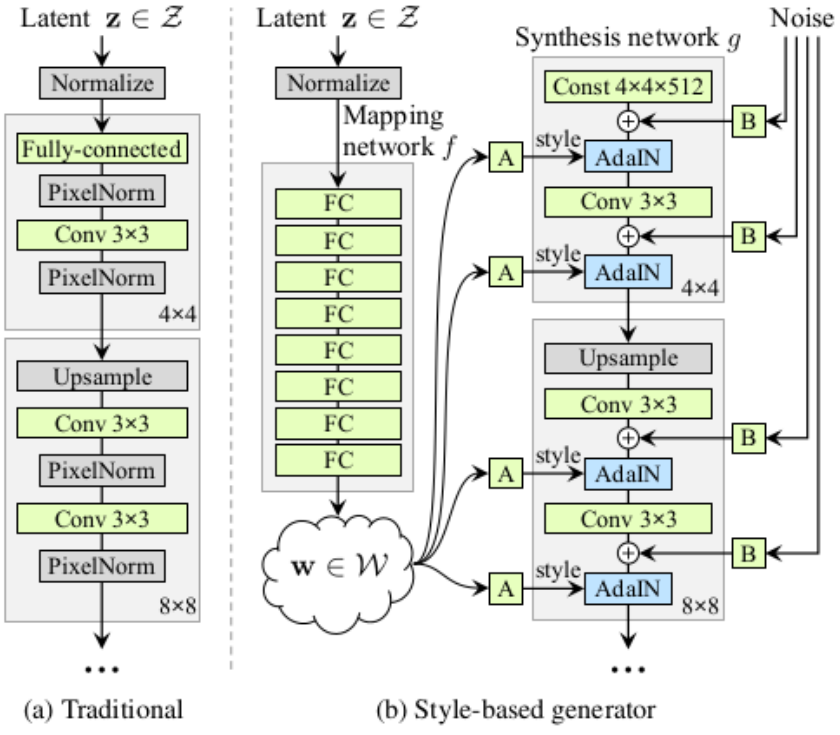
Karras et al. https://arxiv.org/abs/1812.04948

- style vectors y_s, y_b are learnt as an output of an 8-layer FC network.



- third big idea: uses progressive growing

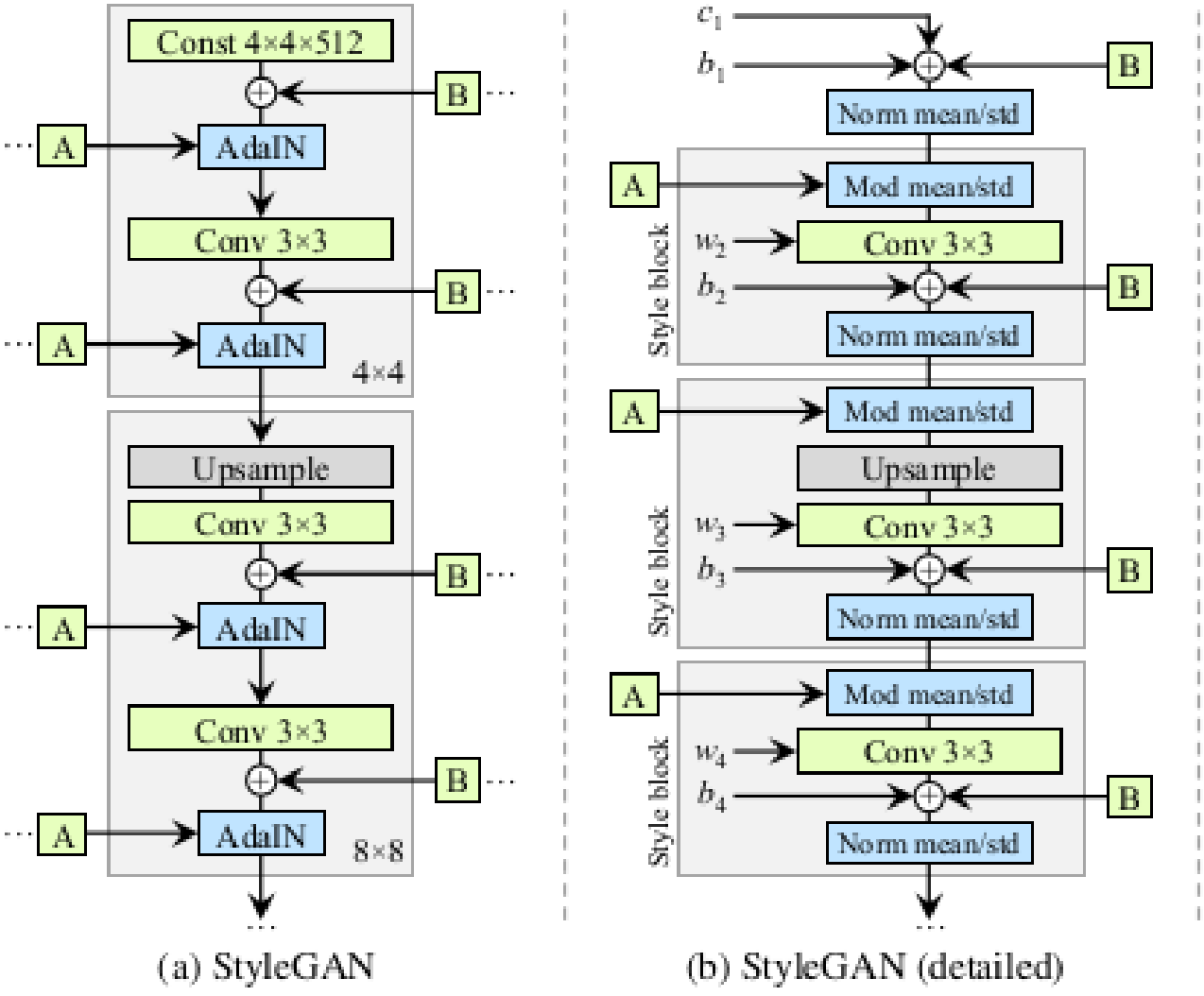
Karras et al. https://arxiv.org/abs/1812.04948



(a) Traditional (b) Style-based generator

Karras et al. https://arxiv.org/abs/1812.04948

- smaller tricks (out of class): 1.using bilinear up- and downsampling. 2. cross-apply style vectors from different runs. 3. Truncate large variations in the style space $\mathcal{W}$.


Const 4×4×512
B
A
AdaIN
Conv 3×3
B
A
AdaIN
4×4
Upsample
Conv 3×3
B
A
AdaIN
Conv 3×3
B
A
AdaIN
8×8
(a) StyleGAN
c_1
b_1
B
Norm mean/std
A
Style block
Mod mean/std
w_2
Conv 3×3
b_2
B
Norm mean/std
A
Style block
Mod mean/std
Upsample
w_3
Conv 3×3
b_3
B
Norm mean/std
A
Style block
Mod mean/std
w_4
Conv 3×3
b_4
B
Norm mean/std
(b) StyleGAN (detailed)

Style-vectors can be recombined from two different runs at different levels

another set of large changes: StyleGan2 (out of class)
https://arxiv.org/abs/1912.04958

including giving up progressive growing

Outline

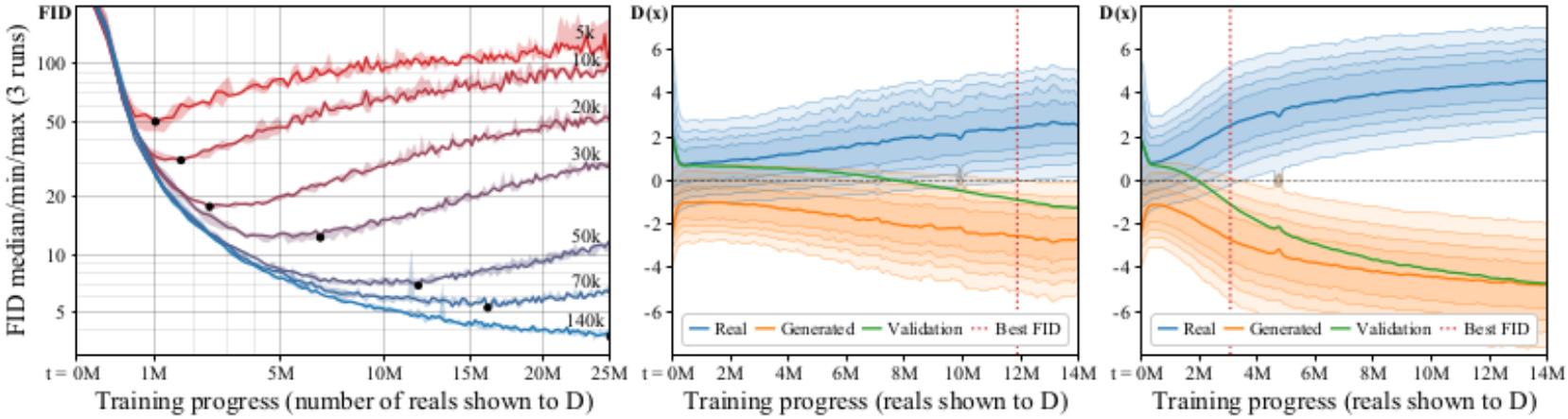
1. a better model - Progressive Growing
2. Application II – Cross Domain mappings
3. StyleGan
4. Training GANs with smaller dataset sizes
5. Inpainting to make GANs useful

Karras et al. Training Generative Adversarial Networks with Limited Data, NeurIPS 2020, https://papers.nips.cc/paper/2020/hash/8d30aa96e72440759f74bd2306c1fa3d-Abstract.html

Takeaway for in-class

- how to create measures to quantify overfitting
- how to use data augmentation in GANs so that the GANs do not start to reproduce the augmentations.
- how to tune augmentation probabilities based on overfitting measures
- how one can apply transfer learning in GANs: to discriminator and freezing lowest layers

Training GANs with smaller dataset sizes: Overfitting

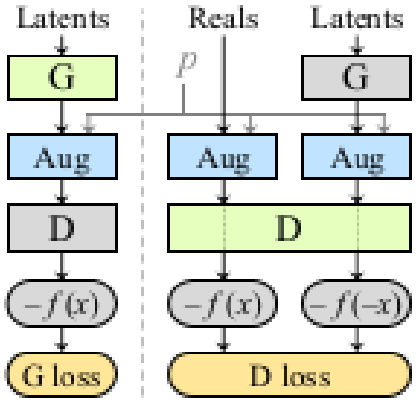


(a) Convergence of FFHQ (256×256) (b) Discriminator outputs, 50k (c) Discriminator outputs, 20k

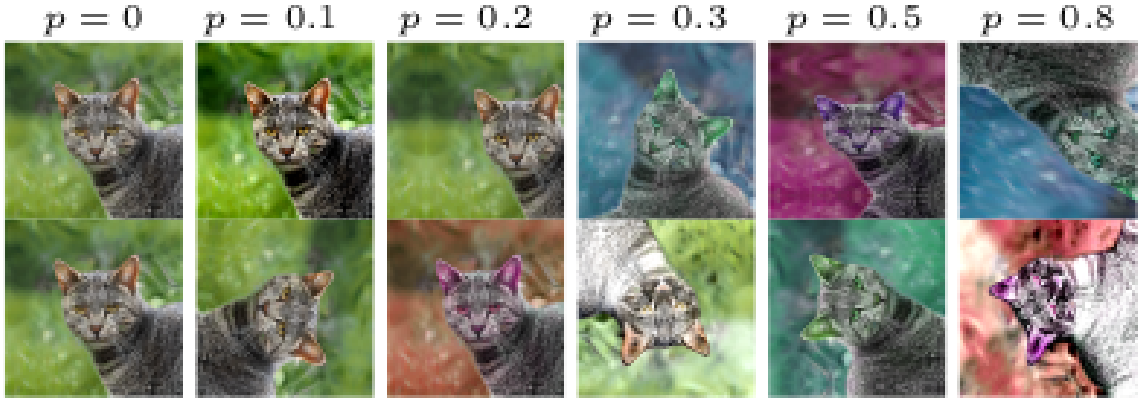
Figure 1: (a) Convergence with different training set sizes. "140k" means that we amplified the 70k dataset by $2\times$ through x-flips; we do not use data amplification in any other case. (b,c) Evolution of discriminator outputs during training. Each vertical slice shows a histogram of $D(x)$, i.e., raw logits.

- GANs overfit
- overfitting can be measured by comparing discriminator scores on: training data, generated data, withheld real images

Training GANs with smaller dataset sizes: Overfitting

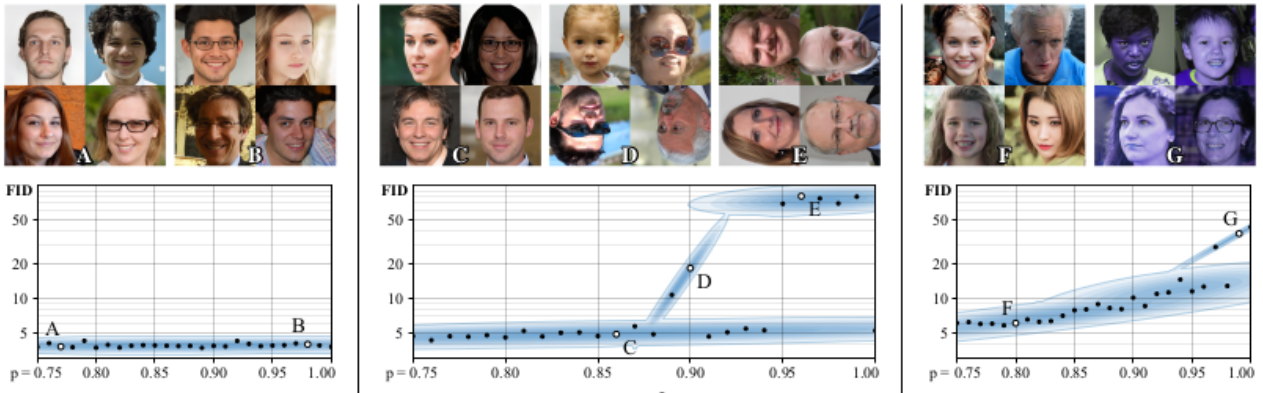


(b) Ours



(c) Effect of augmentation probability p

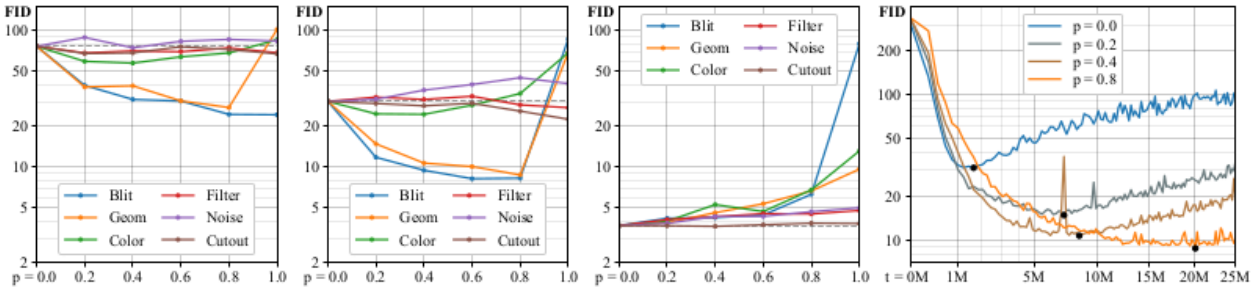
- solution is to use data augmentation for generator and discriminator
- examples: horizontal flip, rotations, color augmentation
- risk/problem is: the generator may learn to produce augmented images.



(a) Isotropic image scaling (b) Random 90° rotations (c) Color transformations

Figure 3: Leaking behavior of three example augmentations, shown as FID w.r.t. the probability of executing the augmentation. Each dot represents a complete training run, and the blue Gaussian mixture is a visualization aid. The top row shows generated example images from selected training runs, indicated by uppercase letters in the plots.

- solution is to use data augmentation for generator and discriminator
- risk/problem is: the generator may learn to produce augmented images.
- solution: allow to not execute each data augmentation with a probability $1 - p \in [0, 1]$
- if p is too large, then it starts to leak the augmentation into the generator

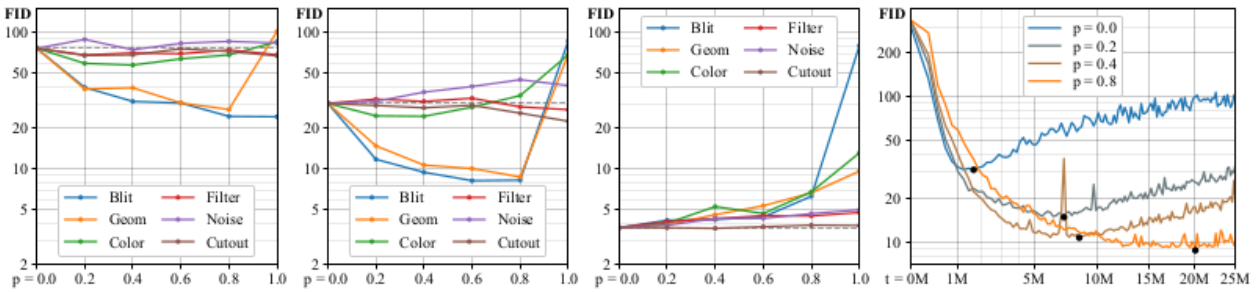


(a) FFHQ-2k (b) FFHQ-10k (c) FFHQ-140k (d) Convergence, 10k, Geom

Figure 4: (a-c) Impact of p for different augmentation categories and dataset sizes. The dashed gray line indicates baseline FID without augmentations. (d) Convergence curves for selected values of p using geometric augmentations with 10k training images.

- solution is to use data augmentation for generator and discriminator
- risk/problem is: the generator may learn to produce augmented images.
- solution: allow to not execute each data augmentation with a probability $1 - p \in [0, 1]$
- observation: different augmentations are differently effective, depends on: dataset size and p.

Training GANs with smaller dataset sizes: Overfitting



(a) FFHQ-2k (b) FFHQ-10k (c) FFHQ-140k (d) Convergence, 10k, Geom

Figure 4: (a-c) Impact of p for different augmentation categories and dataset sizes. The dashed gray line indicates baseline FID without augmentations. (d) Convergence curves for selected values of p using geometric augmentations with 10k training images.

- solution is to use data augmentation for generator and discriminator
- risk/problem is: the generator may learn to produce augmented images.
- solution: allow to not execute each data augmentation with a probability $1 - p \in [0, 1]$
- next problem: how to select the best p?

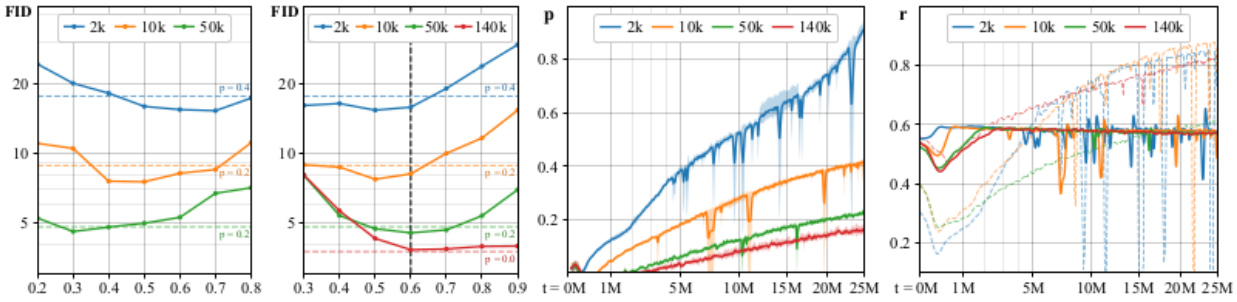
- solution is to use data augmentation for generator and discriminator
- risk/problem is: the generator may learn to produce augmented images.
- solution: allow to not execute each data augmentation with a probability $1 - p \in [0, 1]$
- next problem: how to select the best p?
- define two measures to measure the overfitting, $D(x) \in (-\infty, +\infty)$ as logits:

$$r_v = \frac{E[D_{train}] - E[D_{val}]}{E[D_{train}] - E[D_{generated}]}$$

$$r_t = E_{x \sim \mathrm{TrainData}}[\,\mathrm{sign}(D_{train}(x))\,]$$

- 0 - no overfitting, 1 - bad overfitting

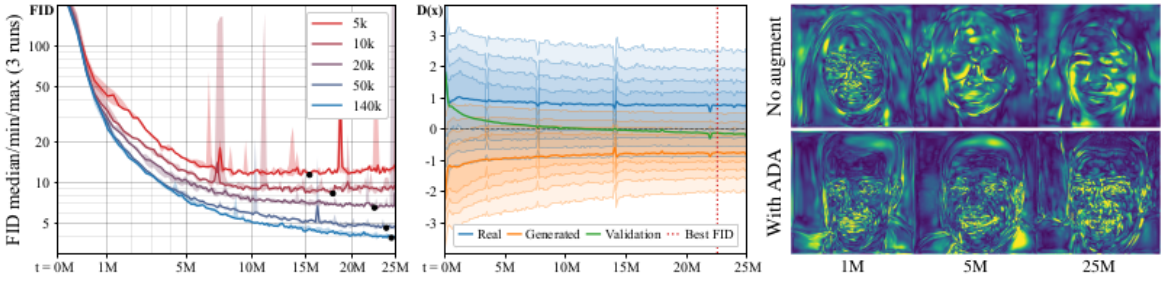
Training GANs with smaller dataset sizes: ADA



(a) r_v target sweep (b) r_t target sweep (c) Evolution of p over training (d) Evolution of r_t

Figure 5: Behavior of our adaptive augmentation strength heuristics in FFHQ. (a,b) FID for different training set sizes as a function of the target value for r_v and r_t. The dashed horizontal lines indicate the best fixed augmentation probability p found using grid search, and the dashed vertical line marks the target value we will use in subsequent tests. (c) Evolution of p over the course of training using heuristic r_t. (d) Evolution of r_t values over training. Dashes correspond to the fixed p values in (b).

- if one keeps r_v or r_t approximately constant by
  - initializing $p = 0$
  - adjusting p every 4 minibatches ...



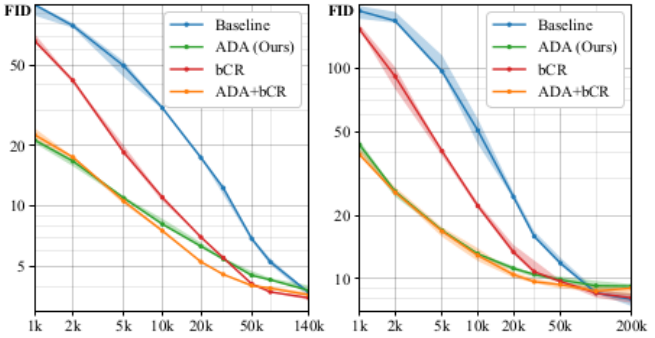
(a) With adaptive augmentation (b) Discriminator outputs, 20k (c) Discriminator gradients, 10k

Figure 6: (a) Training curves for FFHQ with different training set sizes using adaptive augmentation. (b) The supports of real and generated images continue to overlap. (c) Example magnitudes of the gradients the generator receives from the discriminator as the training progresses.

- solution is to use data augmentation for generator and discriminator
- allow to not execute each data augmentation with a probability $1 - p \in [0, 1]$
- Define two measures in $[0, 1]$ to measure the overfitting:

$$r_t = E_{x \sim \text{TrainData}}[\text{sign}(D_{train}(x))]$$

- run an algorithm: initializing $p = 0$, adjusting p every 4 minibatches to keep $r_t \approx 0.6$



(a) FFHQ (256 × 256) (b) LSUN CAT (256 × 256)

Dataset		Baseline	ADA	+ bCR
FFHQ	1k	100.16	**21.29**	22.61
	5k	49.68	10.96	**10.58**
	10k	30.74	8.13	**7.53**
	30k	12.31	5.46	**4.57**
	70k	5.28	4.30	**3.91**
	140k	3.71	3.81	**3.62**
LSUN CAT	1k	186.91	43.25	**38.82**
	5k	96.44	16.95	**16.80**
	10k	50.66	13.13	**12.90**
	30k	15.90	10.50	**9.68**
	100k	**8.56**	9.26	8.73
	200k	**7.98**	9.22	9.03

(c) Median FID

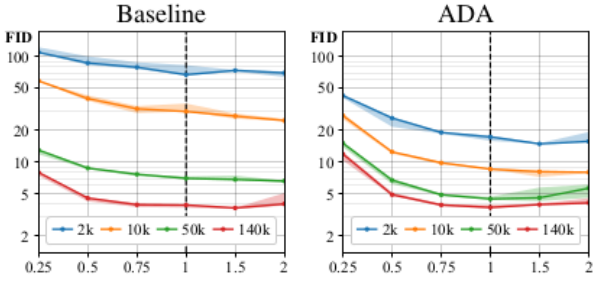


(d) Mean image

Figure 7: (a-c) FID as a function of training set size, reported as median/min/max over 3 training runs. (d) Average of 10k random images generated using the networks trained with 5k subset of FFHQ. ADA matches the average of real data, whereas the xy-translation augmentation in bCR [43] has leaked to the generated images, significantly blurring the average image.

FFHQ (256 × 256)		2k	10k	140k
Baseline		78.80±2.31	30.73±0.48	3.66±0.10
PA-GAN	[39]	56.49±7.28	27.71±2.77	3.78±0.06
WGAN-GP	[13]	79.19±6.30	35.68±1.27	6.54±0.37
zCR	[43]	71.61±9.64	23.02±2.09	**3.45**±0.19
Auxiliary rotation	[6]	66.64±3.64	25.37±1.45	4.16±0.05
Spectral norm	[24]	88.71±3.18	38.58±3.14	4.60±0.19
Shallow mapping		71.35±7.20	27.71±1.96	3.59±0.22
Adaptive dropout		67.23±4.76	23.33±0.98	4.16±0.05
ADA (Ours)		**16.49**±0.65	**8.29**±0.31	3.88±0.13

(a) Comparison methods



(b) Discriminator capacity sweeps

Figure 8: (a) We report the mean and standard deviation for each comparison method, calculated over 3 training runs. (b) FID as a function of discriminator capacity, reported as median/min/max over 3 training runs. We scale the number of feature maps uniformly across all layers by a given factor (x-axis). The baseline configuration (no scaling) is indicated by the dashed vertical line.

- solution is to use data augmentation for generator and discriminator
- run an algorithm: initializing $p = 0$, adjusting p every 4 minibatches to keep $r_t \approx 0.6$

Training GANs with smaller dataset sizes: Transfer Learning

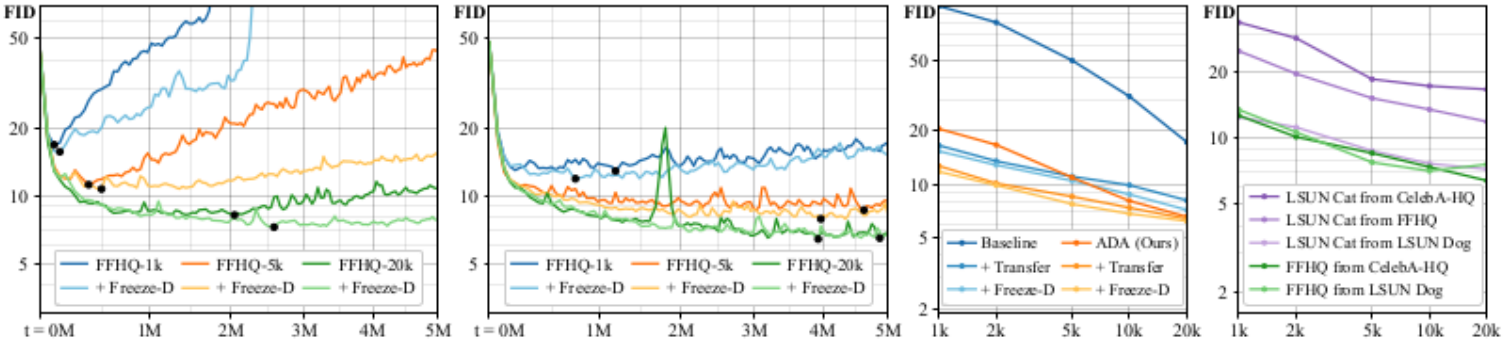


Figure 9: Transfer learning FFHQ starting from a pre-trained CELEBA-HQ model, both 256×256. (a) Training convergence for our baseline method and Freeze-D [26]. (b) The same configurations with ADA. (c) FIDs as a function of dataset size. (d) Effect of source and target datasets.

- the effect of transfer learning for the discriminator
- freeze only the layers for the 3-4 highest resolutions, its a tunable parameter

Training GANs with smaller dataset sizes: Transfer Learning

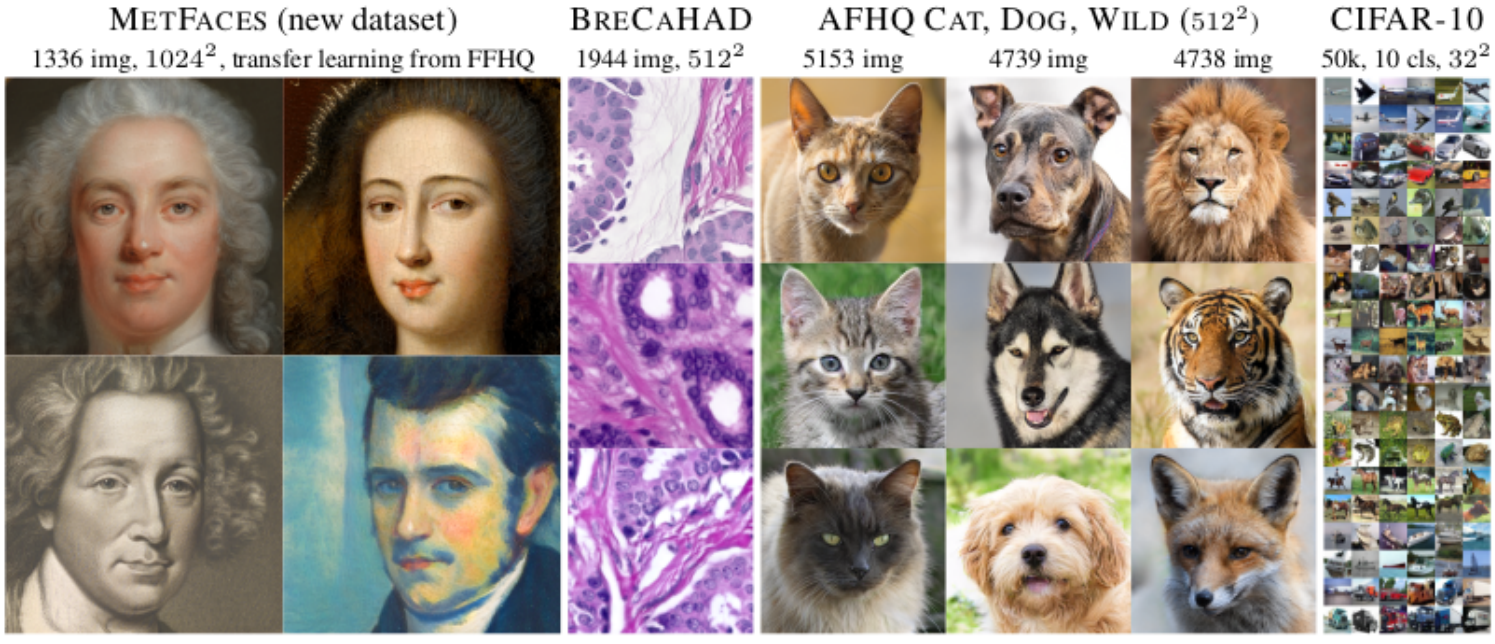


Figure 10: Example generated images for several datasets with limited amount of training data, trained using ADA. We use transfer learning with METFACES and train other datasets from scratch. See Appendix A for uncurated results and real images, and Appendix D for our training configurations.

- the effect of transfer learning for the discriminator
- freeze only the layers for the 3-4 highest resolutions, its a tunable parameter

Outline

1. a better model - Progressive Growing
2. Application II – Cross Domain mappings
3. StyleGan
4. Training GANs with smaller dataset sizes
5. Inpainting to make GANs useful

Why inpainting?

What is required to know for GAN inpainting?

- what problem it solves
- challenges for this setup
- not the model details

Why inpainting?

- create content subject to constraints
- can iteratively postprocess bad regions using different+specialized generators (e.g. for vegetation, building facades, ears of humans, teeth, water waves)

Challenges in inpainting I

- local consistency of the inpainted patch



- solution: use a discriminator on the patch

- global consistency of the image as a whole



- solution: use a discriminator on the whole image

- informing the generators about local statistics of the image



- e.g. what textures / structures should appear in the white patches?
- how to build a model that can provide such an information to the generative part of it?

https://arxiv.org/abs/1801.07892 Yu et al. CVPR 2018

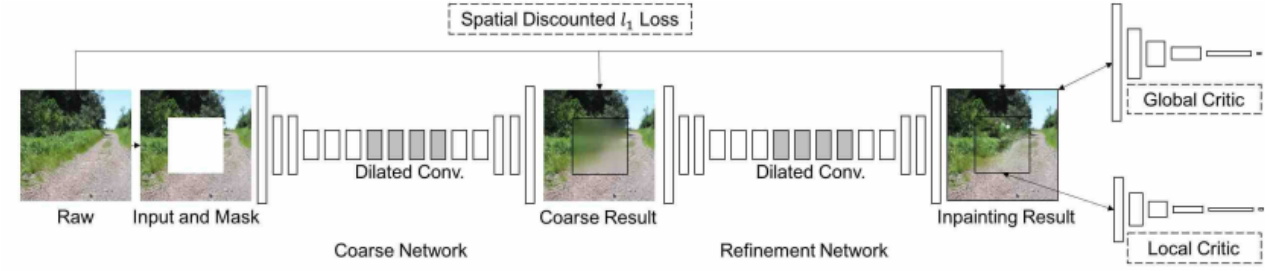
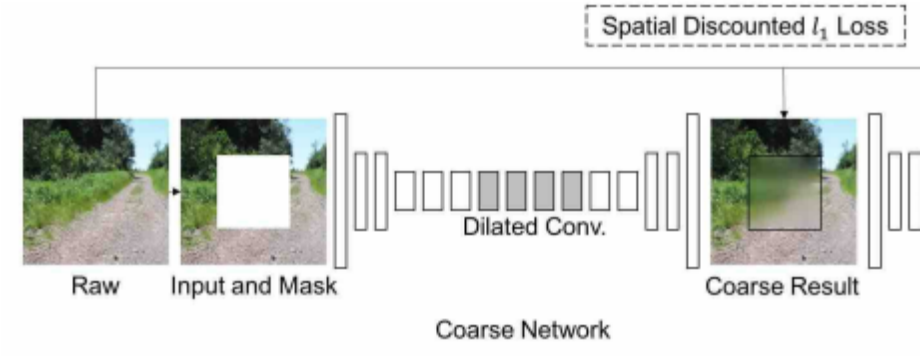


Figure 2: Overview of our improved generative inpainting framework. The coarse network is trained with reconstruction loss explicitly, while the refinement network is trained with reconstruction loss, global and local WGAN-GP adversarial loss.

Two stage architecture

- stage 1: direct, coarse reconstruction
- stage 2: apply reconstruction loss and two discriminators

One example paper – first, coarse reconstruction stage

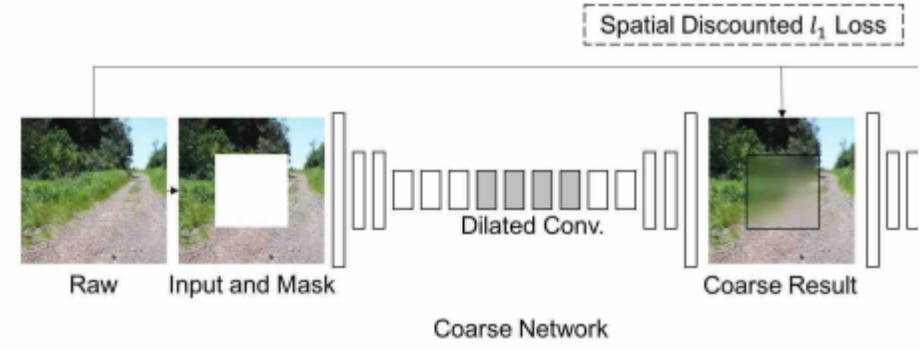


- stage 1 takes an image x as input, adds white patches $x \mapsto h(w)$ in random positions as input and reconstructs the image x based on $h(x)$ as $f(h(x))$ using the model $f(\cdot)$
- Training loss: weighted pixel-wise reconstruction loss

$$L(x, f(x)) = \|(f(h(x)) - x) \odot \gamma(h(x))\|_1$$

The weight $\gamma(h(x))$ is 1 on pixels without white patches, and on white patches it decreases to zero as a function of the distance to the nearest non-masked pixel:

$$\gamma(h(x)) = \gamma^l, \; l = \text{pixel distance to nearest non-masked pixel}$$



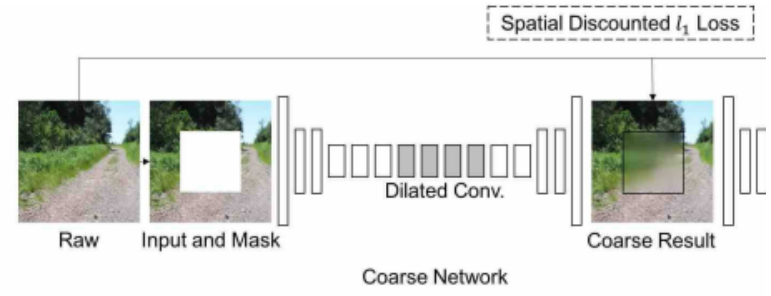
- stage 1 takes an image x as input, adds white patches $x \mapsto h(w)$ in random positions as input and reconstructs the image x based on $h(x)$ as $f(h(x))$ using the model $f(\cdot)$
- Training loss: weighted pixel-wise reconstruction loss

$$L(x, f(x)) = \|(f(h(x)) - x) \odot \gamma(h(x))\|_1$$

$$\gamma(h(x)) = \gamma^l, \; l = \text{pixel distance to nearest non-masked pixel}$$

- Idea: less weight on reconstruction quality for the coarse first stage for pixels far away from a non-masked pixel. Enforce a more truthful reconstruction on unmasked pixels and close to them.

One example paper – first, coarse reconstruction stage



Inpainting network Inpainting network has two encoder-decoder architecture stacked together, with each encoder-decoder of network architecture:

K5S1C32 - K3S2C64 - K3S1C64 - K3S2C128 - K3S1C128 - K3S1C128 - K3D2S1C128 - K3D4S1C128 - K3D8S1C128 - K3D16S1C128 - K3S1C128 - K3S1C128 - resize (2×) - K3S1C64 - K3S1C64 - resize (2×) - K3S1C32 - K3S1C16 - K3S1C3 - clip.

Prediction model:

- takes an image with random white patches, outputs an image
- start: 5 standard convolution layers, 2 with stride 2. Result: feature map compressed to 1/4.

  Creates a bottleneck: lower resolution, higher no. feature channels.
- then 4 dilated convolution layers.
- then (conv-conv-upsample 2x) – repeated two times. to obtain the original image size back.
- then 3x convolutions, last with output with 3 channels

Dilated convolutions?
https://github.com/vdumoulin/conv_arithmetic

- dilation factor 1: take every pixel from input feature map (the usual!)
- dilation factor 2: take every second pixel from input feature map
- dilation factor n: take every $n-th$ pixel from input feature map
- allows to process larger fields of view – good for reconstruction tasks to obtain a multiscale global view of the information in the image

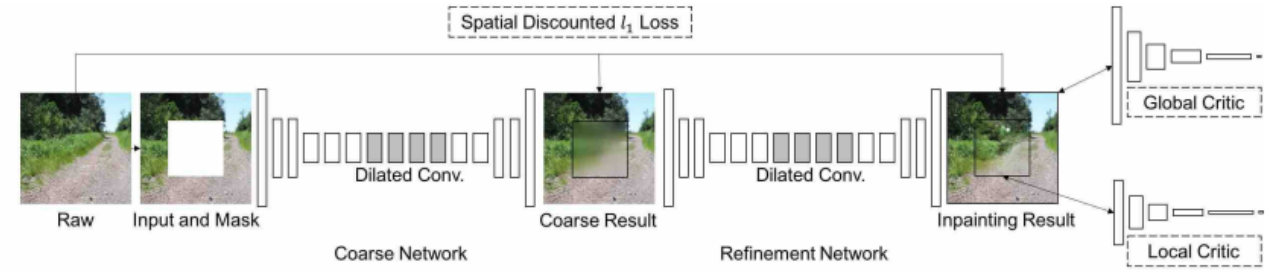


Figure 2: Overview of our improved generative inpainting framework. The coarse network is trained with reconstruction loss explicitly, while the refinement network is trained with reconstruction loss, global and local WGAN-GP adversarial loss.

Training loss: three losses summed

- loss 1: weighted pixel-wise reconstruction loss $\|(f(h(x)) - x) \odot \gamma(h(x))\|_1$
- loss 2: discriminator on patch. WGAN-GP :)
- loss 3: discriminator on the whole image. WGAN-GP

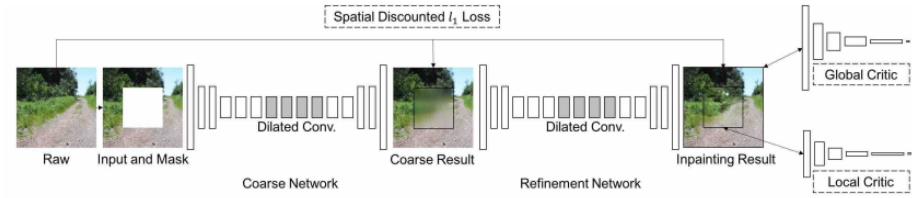
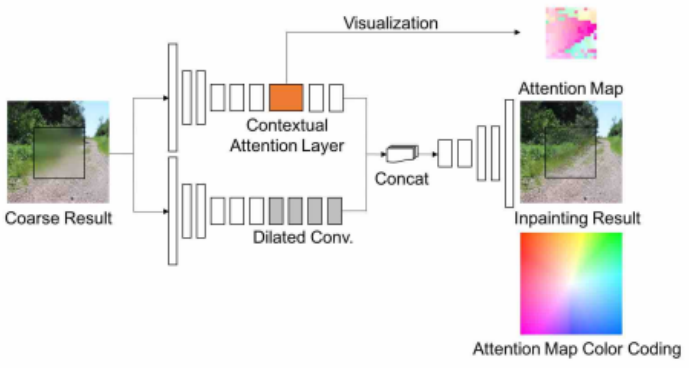
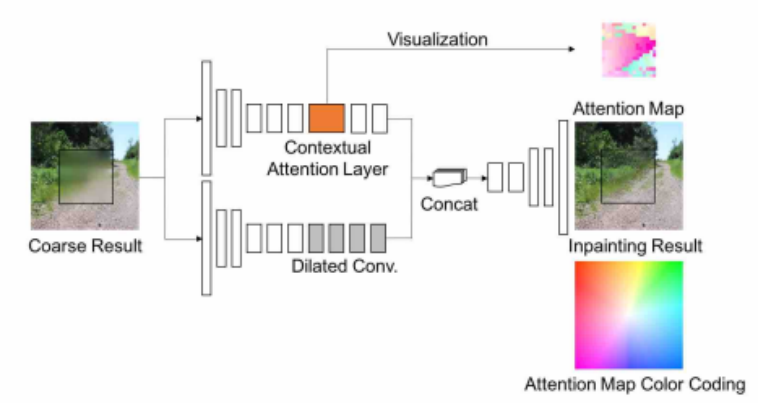


Figure 2: Overview of our improved generative inpainting framework. The coarse network is trained with reconstruction loss explicitly, while the refinement network is trained with reconstruction loss, global and local WGAN-GP adversarial loss.

Finer stage prediction model:

- two streams, concatenated before final 3 convolutions down to three channels, which are same final convolutions as in stage 1 model.
- stream 1: 5 convolutions, dilations, (conv-conv-upsample) twice repeated as in stage 1 model. Without stream 2 it would be the same model again.
- stream 2: informs about patch statistics from non-masked regions, explained next!





Stream 2: contextual attention. idea: compute how good a patch from coarsely reconstructed image matches any of the background patches from the non-masked regions of this image.

- if we had a background patch of spatial size 3×3 in some feature space f_b from the non-masked regions of this image, and a patch f_v from the coarsely reconstructed image, then we could compute the cosine angle as similarity measure

$$s(f_b, f_v) = \frac{f_b \cdot f_v}{\|f_b\| \|f_v\|}$$

- if we had N background patches, we could compute a softmax of similarities over all those N patches $f_b^{(i)}$ from non-masked regions:

$$p(f_b^{(1)}, f_b^{(2)}, \ldots, f_b^{(N)} | f_v) = softmax\left(s(f_b^{(1)}, f_v), \ldots, s(f_b^{(N)}, f_v)\right)$$

Questions?!