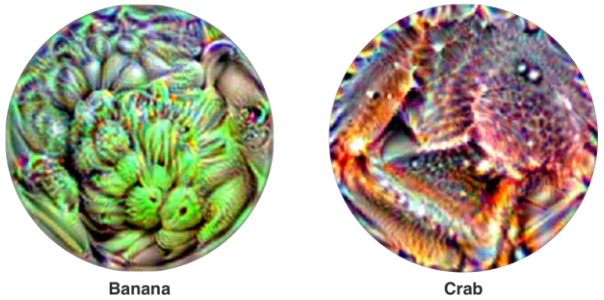
بنام خداوند جان و خرد



Adversarial Attacks, GANs, and VAEs

Mohammad Taher Pilehvar

Deep Learning 99

https://teias-courses.github.io/dl99

Some of the slides were borrowed from Stanford's CS230

Today

- Adversarial attacks
  - FGSM: The Fast Gradient Signed method
  - One pixel attack
  - Adversarial patch
- Adversarial defence
- Generative Adversarial Networks (GANs)

Adversarial Attacks

https://arxiv.org/abs/1904.08653

Adversarial examples

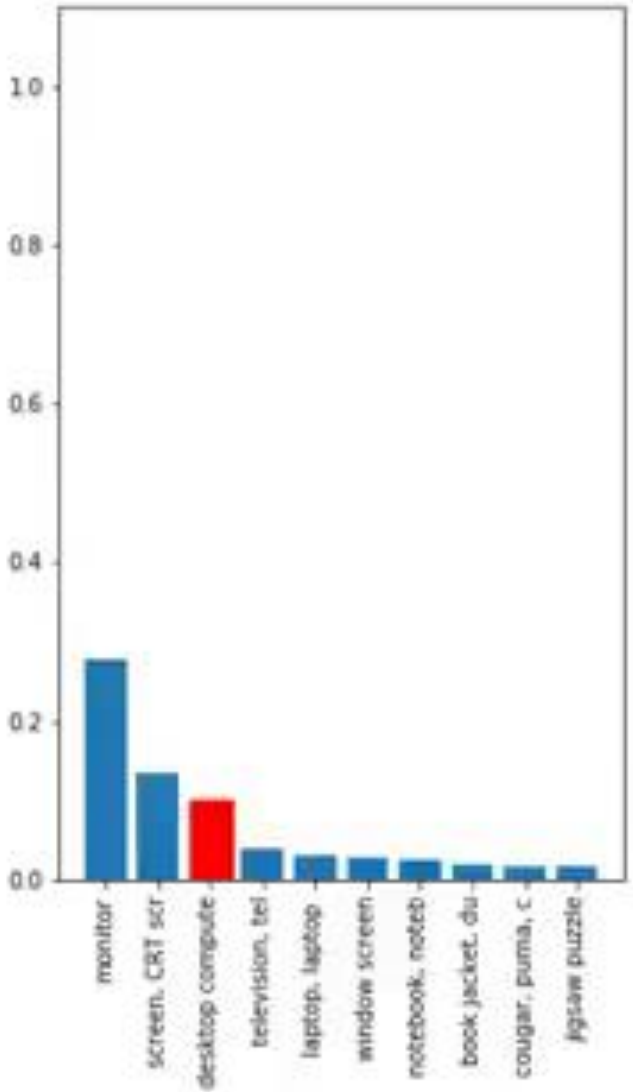
- Specialized inputs created with the purpose of confusing a neural network

Adversarial Attacks

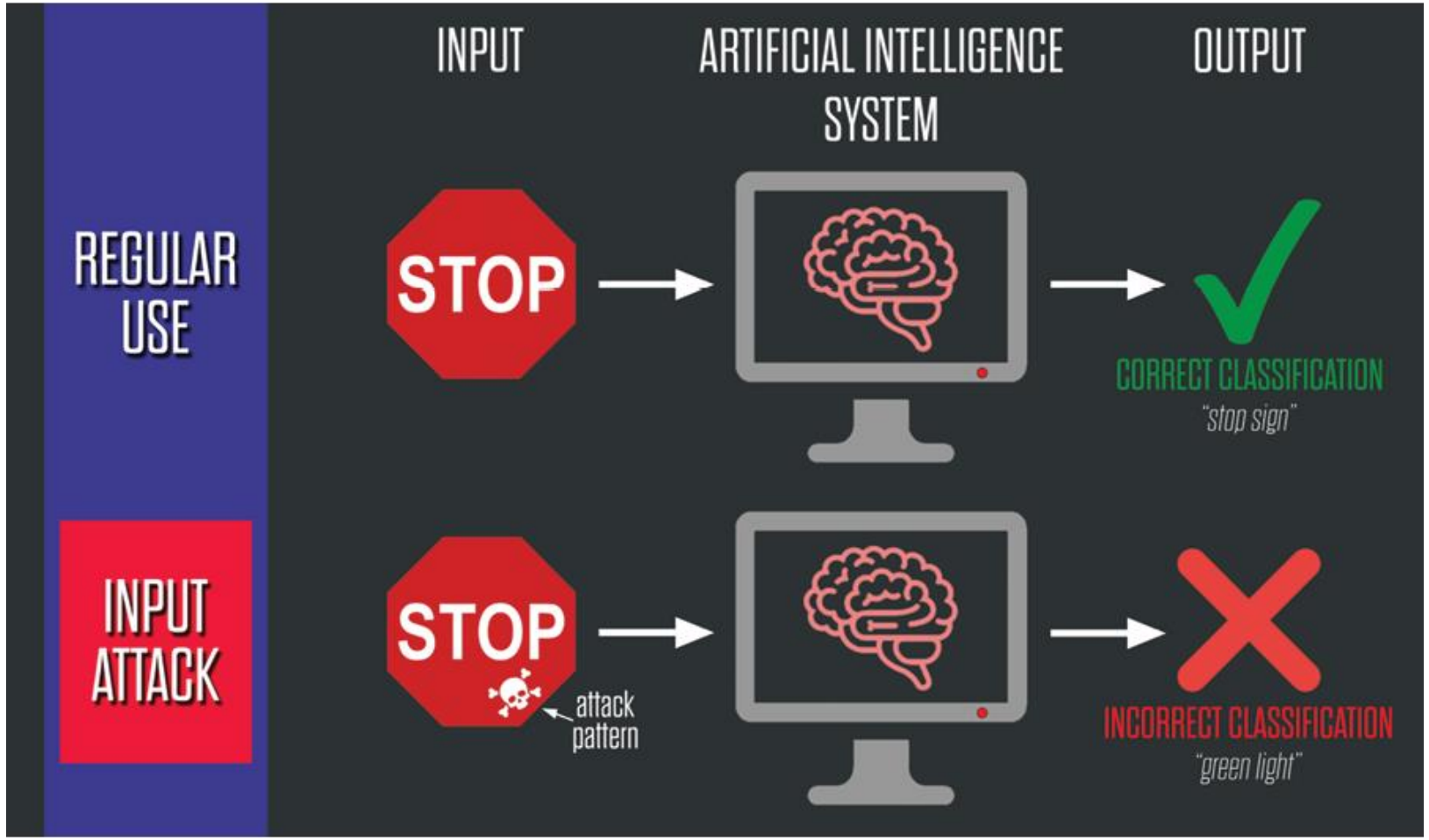
OpenAI

Adversarial examples

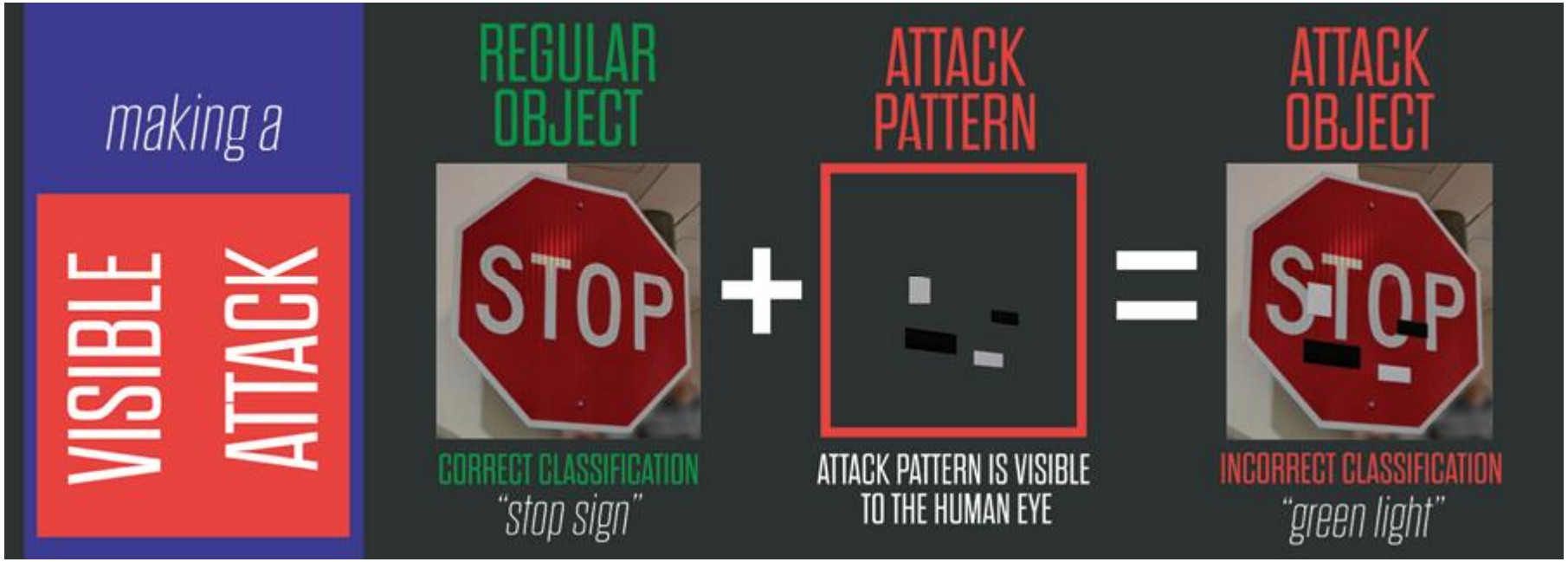
- Specialized inputs created with the purpose of confusing a neural network



Adversarial Attacks



Adversarial Attacks



Read more:
https://www.belfercenter.org/publication/AttackingAI

Adversarial Attacks

Adversarial examples

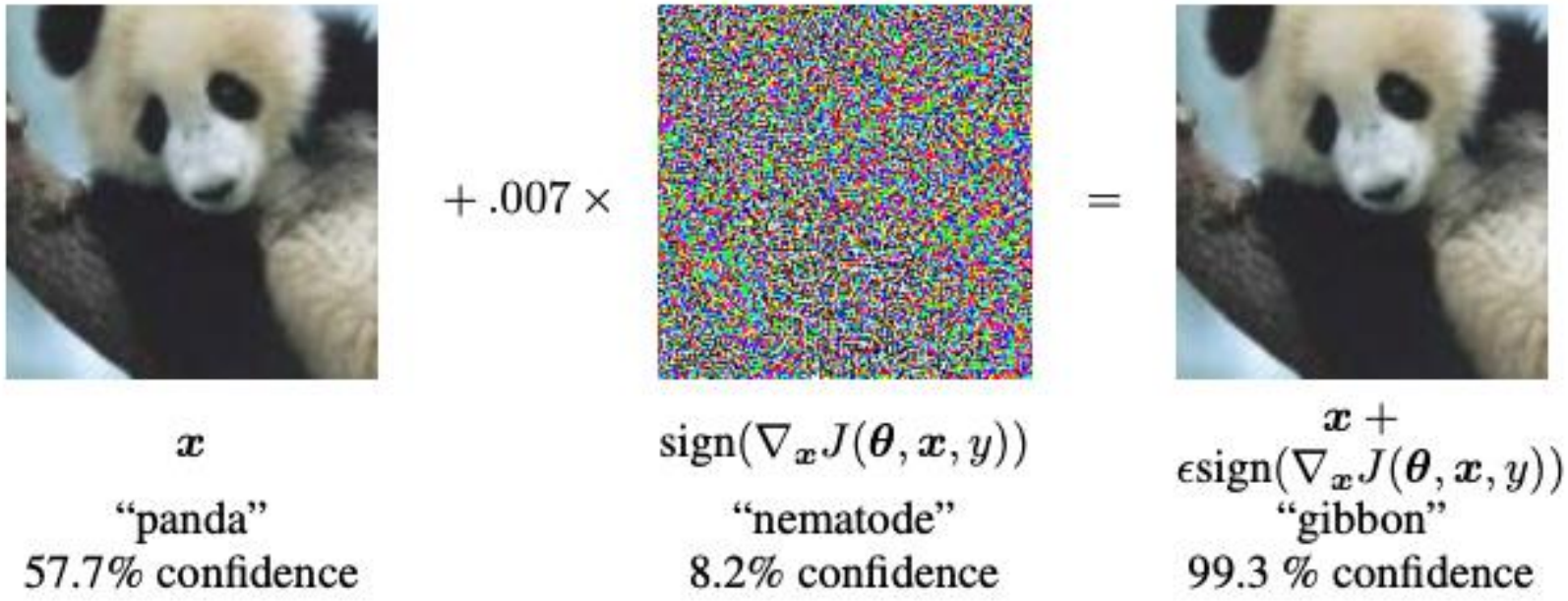
- Specialized inputs created with the purpose of confusing a neural network

- White box attack
  - Complete access to the model being attacked
- Black box attack
  - No knowledge of model internal architectures or training parameters

Contents of this section are based on Tensroflow's FGSM tutorial

Adversarial Attacks

- White box attack
  - FGSM: Fast Gradient Signed Method (Goodfellow et al, 2014)



Adversarial Attacks - FGSM

Create an image that maximizes the loss

$$adv\_x = x + \epsilon * \text{sign}(\nabla_x J(\theta, x, y))$$

- adv_x : Adversarial image.
- x : Original input image.
- y : Original input label.
- ϵ : Multiplier to ensure the perturbations are small.
- θ : Model parameters.
- J : Loss.

Adversarial Attacks - FGSM

```
loss_object = tf.keras.losses.CategoricalCrossentropy()

def create_adversarial_pattern(input_image, input_label):
  with tf.GradientTape() as tape:
    tape.watch(input_image)
    prediction = pretrained_model(input_image)
    loss = loss_object(input_label, prediction)

  # Get the gradients of the loss w.r.t to the input image.
  gradient = tape.gradient(loss, input_image)
  # Get the sign of the gradients to create the perturbation
  signed_grad = tf.sign(gradient)
  return signed_grad
```

FGSM – other variants

- A) **Target Class Method:** This variant of FGSM [43] approach maximizes the probability of some specific target class y_{target}, which is unlikely the true class for a given example. The adversarial example is crafted using the following equation:

$$X_* = X - \epsilon * sign(\nabla_x J(X, y_{target}))$$

- B) **Basic Iterative Method:** This is a straightforward extension of the basic FGSM method [43]. This method generates adversarial samples iteratively using small step size.

$$X_*^0 = X; \quad X_*^{n+1} = Clip_{X,e}\{X_*^n + \alpha * sign(\nabla_x J(X_*^n, y_{true}))\}$$

Here, α is the step size and $Clip_{X,e}\{A\}$ denotes the element-wise clipping of X. The range of $A_{i,j}$ after clipping belongs in the interval $[X_{i,j} - \epsilon, X_{i,j} + \epsilon]$. This method does not typically rely on any approximation of the model and produces additional harmful adversarial examples when run for more iterations.

From Chakraborty et al [2018]

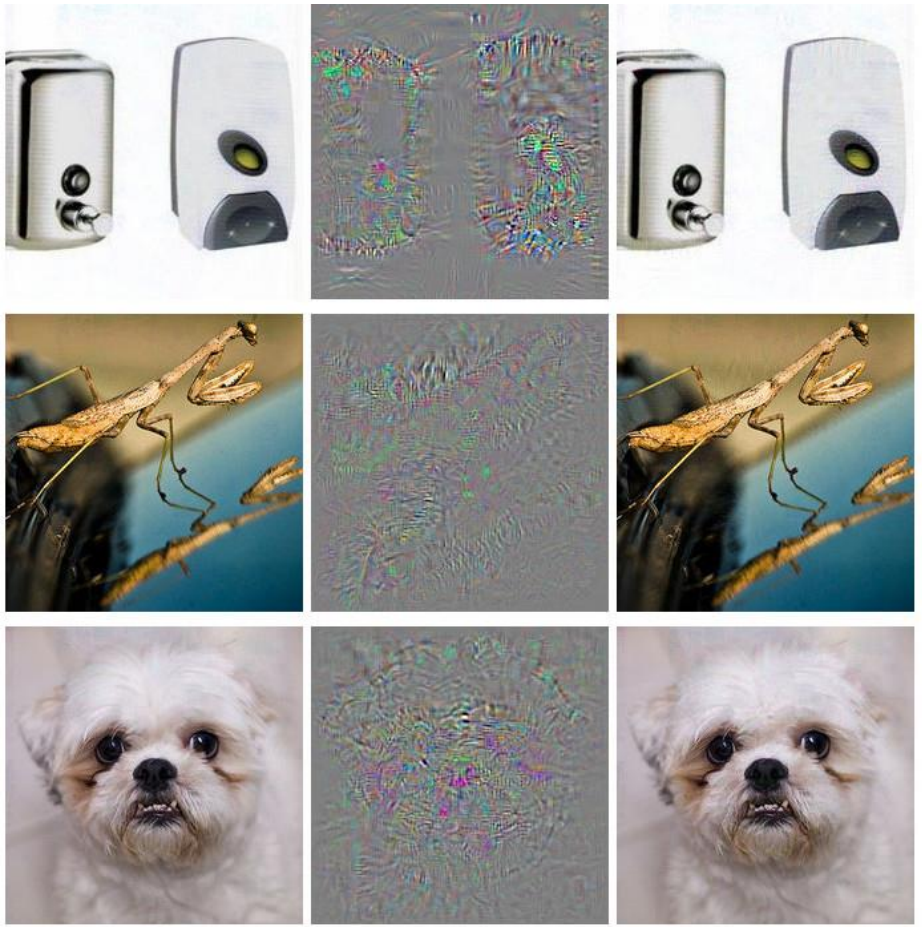
Adversarial Attacks

- Szegedy et. al (2013)

Minimize the following function w.r.t *r*

$$loss(\hat{f}(x+r), l) + c \cdot |r|$$

- *x* is the input
- *r* is the change to the input
- *l* is the desired outcome class
- *c adjusts the distance between images and the distance between predictions*

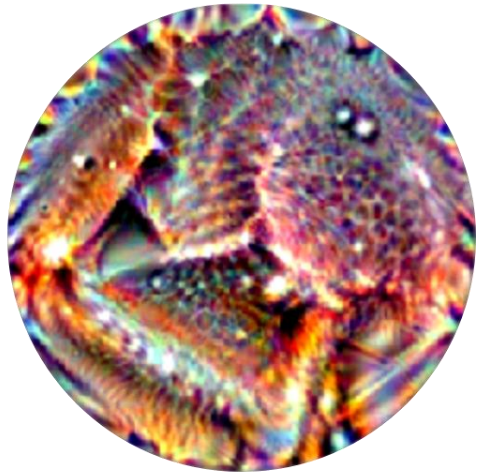


All images in the right column are predicted to be an *ostrich*

Adversarial Patch

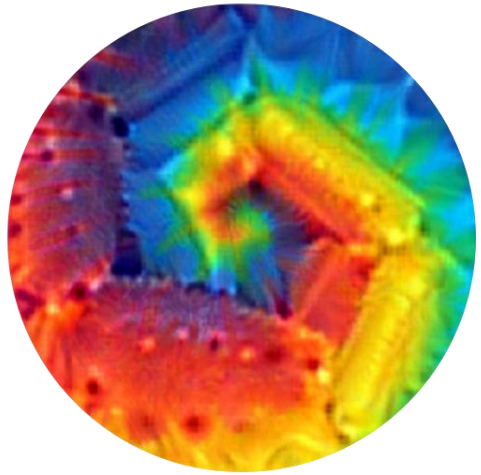
Brown et al (2017)
https://arxiv.org/abs/1712.09665

Banana



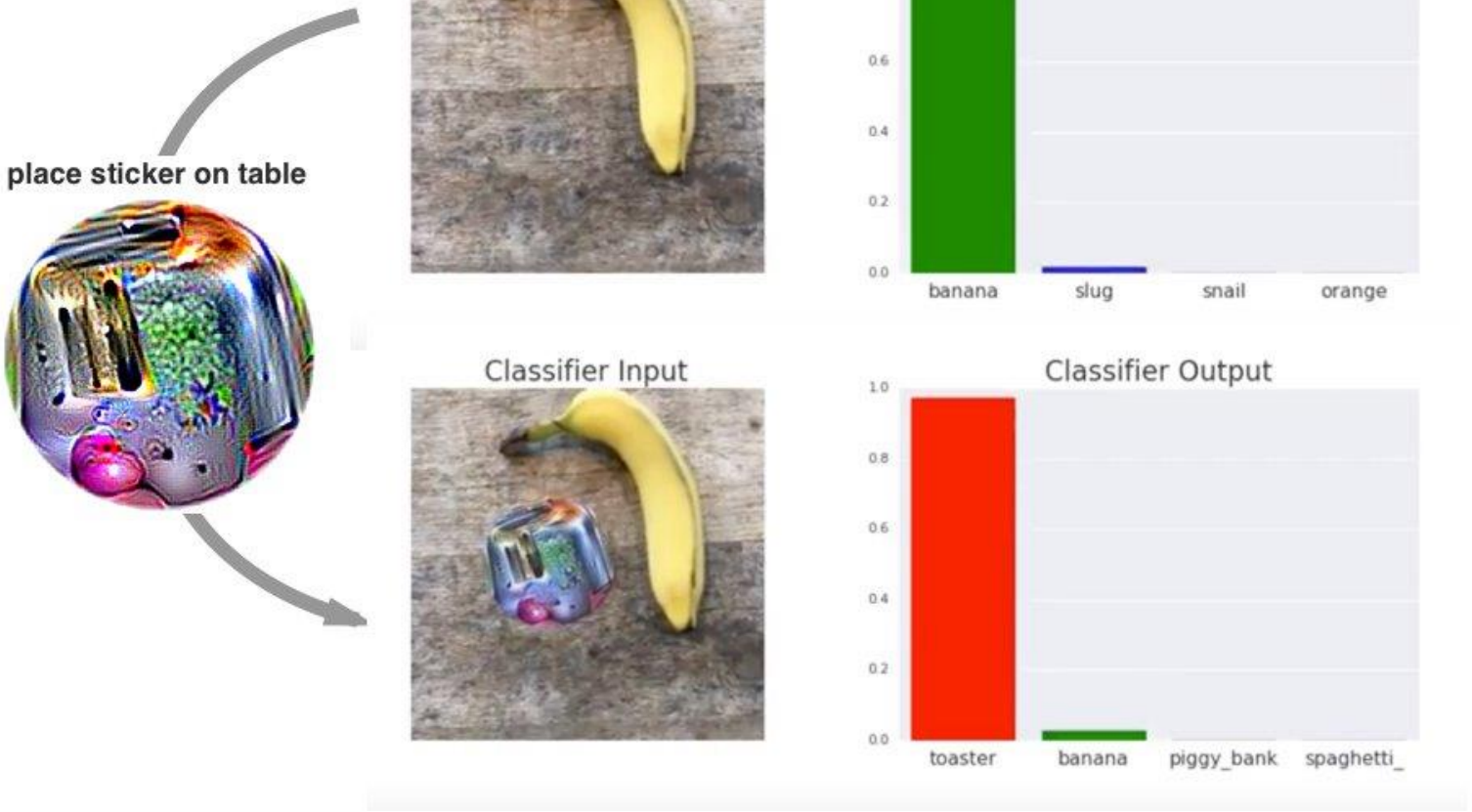
Crab

Toaster

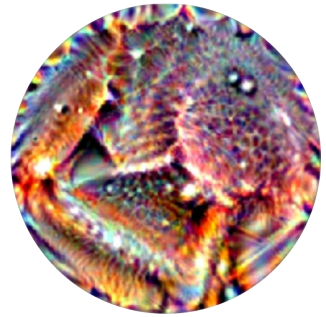


Crab (disguised)

Adversarial Patch

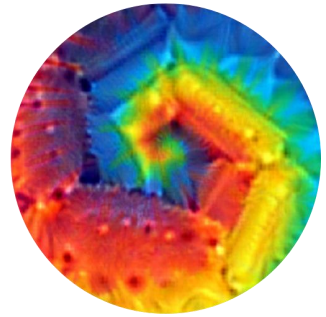


Banana



Crab

Toaster



Crab (disguised)

One Pixel Attack

Su et al (2019)

- https://arxiv.org/abs/1710.08864

Adversarial Defense

- Adversarial training
  - Brute force
  - Train the model on many (automatically generated) adversarial examples not to be fooled by them
  - CleverHans: open-source implementation

- Defensive distillation
  - Papernot et al (2016): https://arxiv.org/abs/1511.04508
  - Attack methods usually rely on large gradients
  - Distilling a network reduces gradients

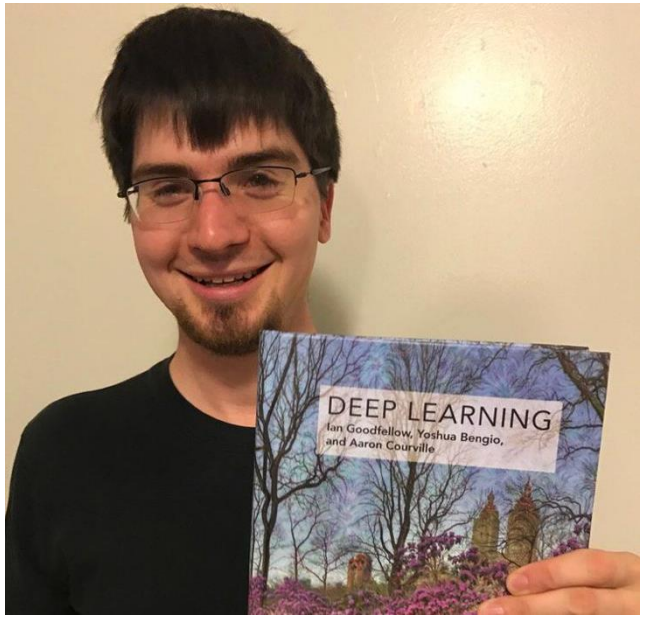
Read more

Adversarial Attacks and Defenses in Deep Learning

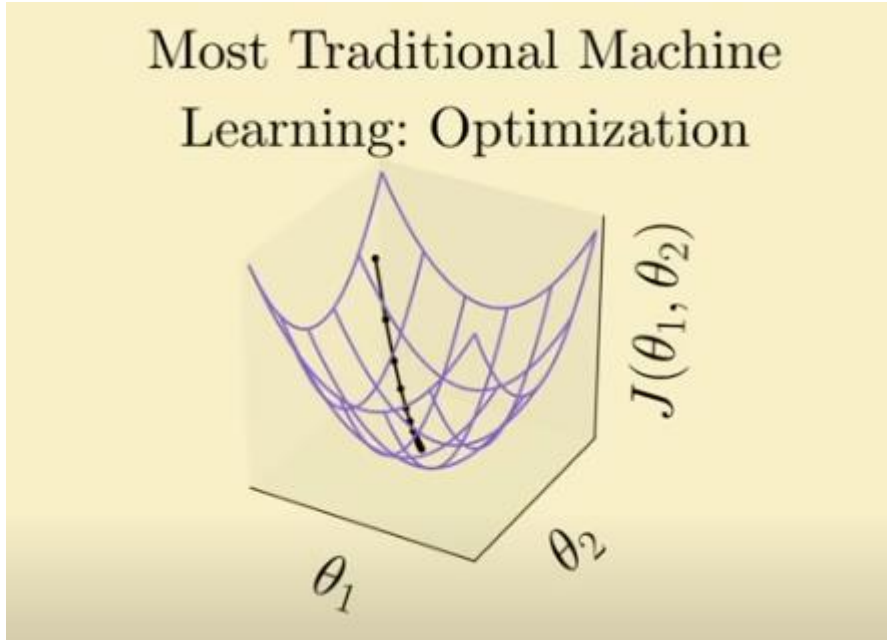
- Ren et al (2020)

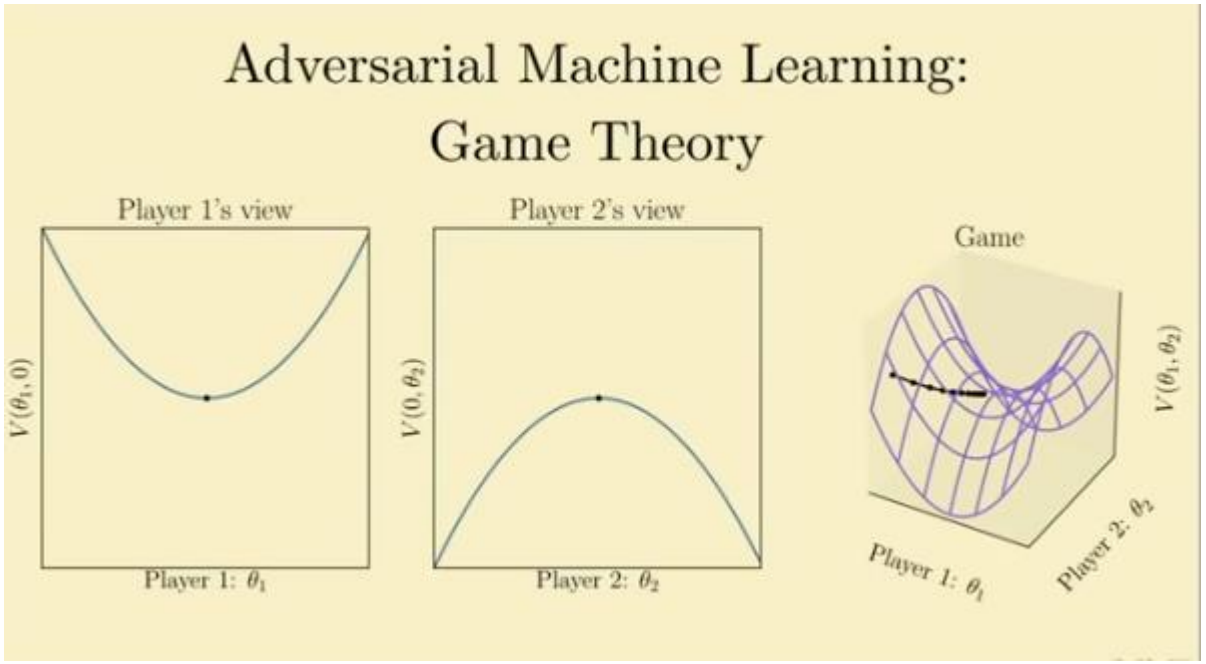
Generative Adversarial Networks (GANs)

Introduced in 2014 by Ian Goodfellow et al

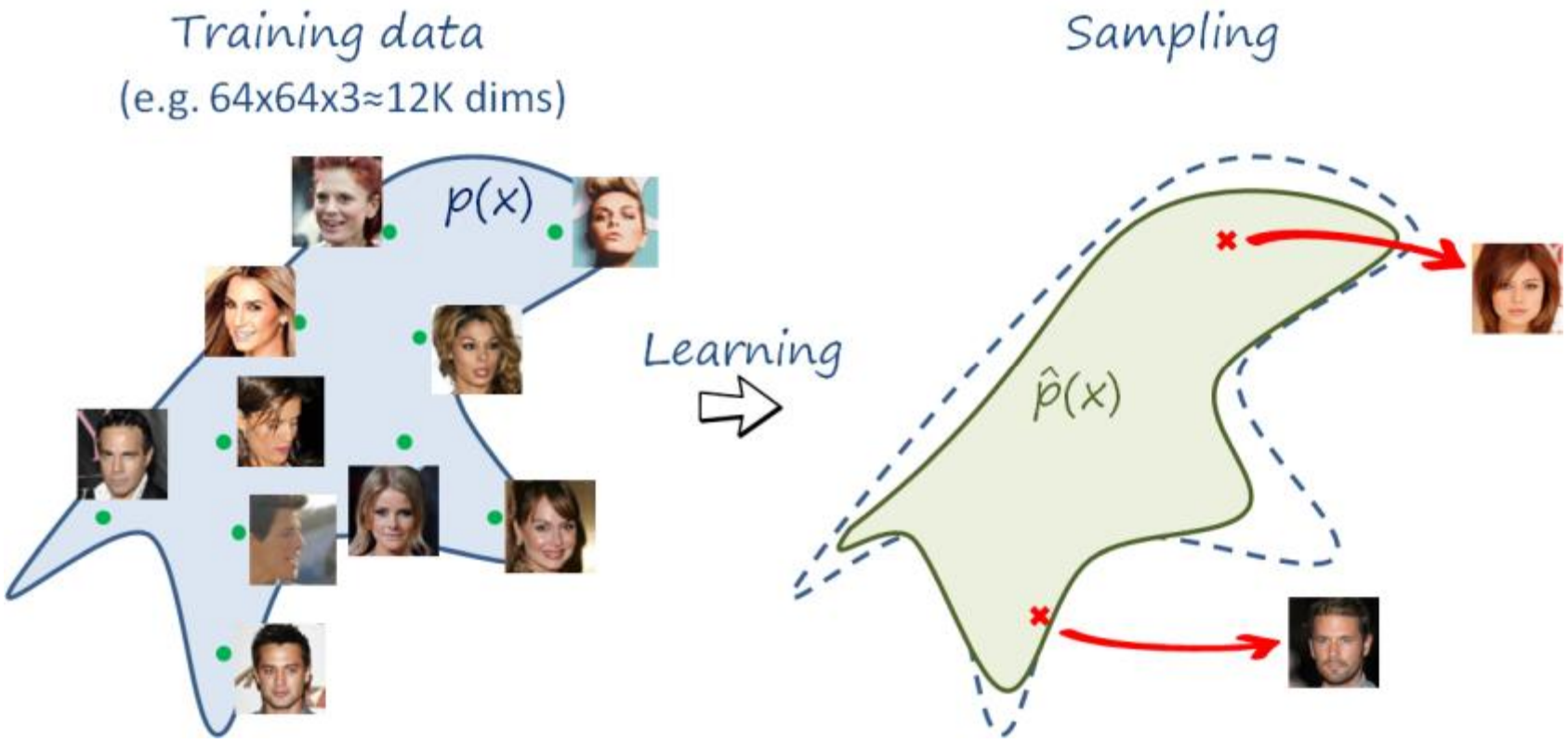


Adversarial ML



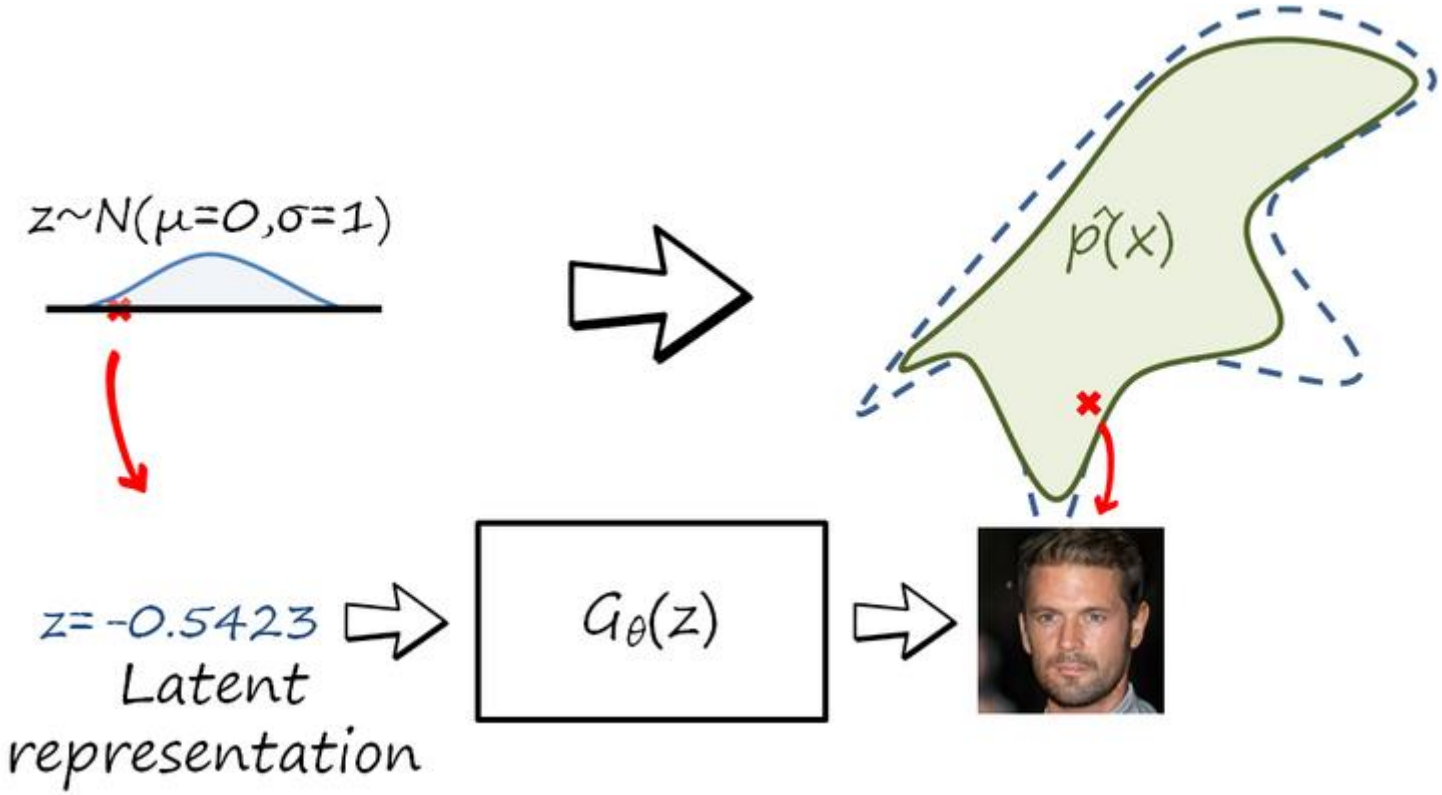


Generative modeling



Courtesy of Luis Herranz

Generative Adversarial Networks



Courtesy of Luis Herranz

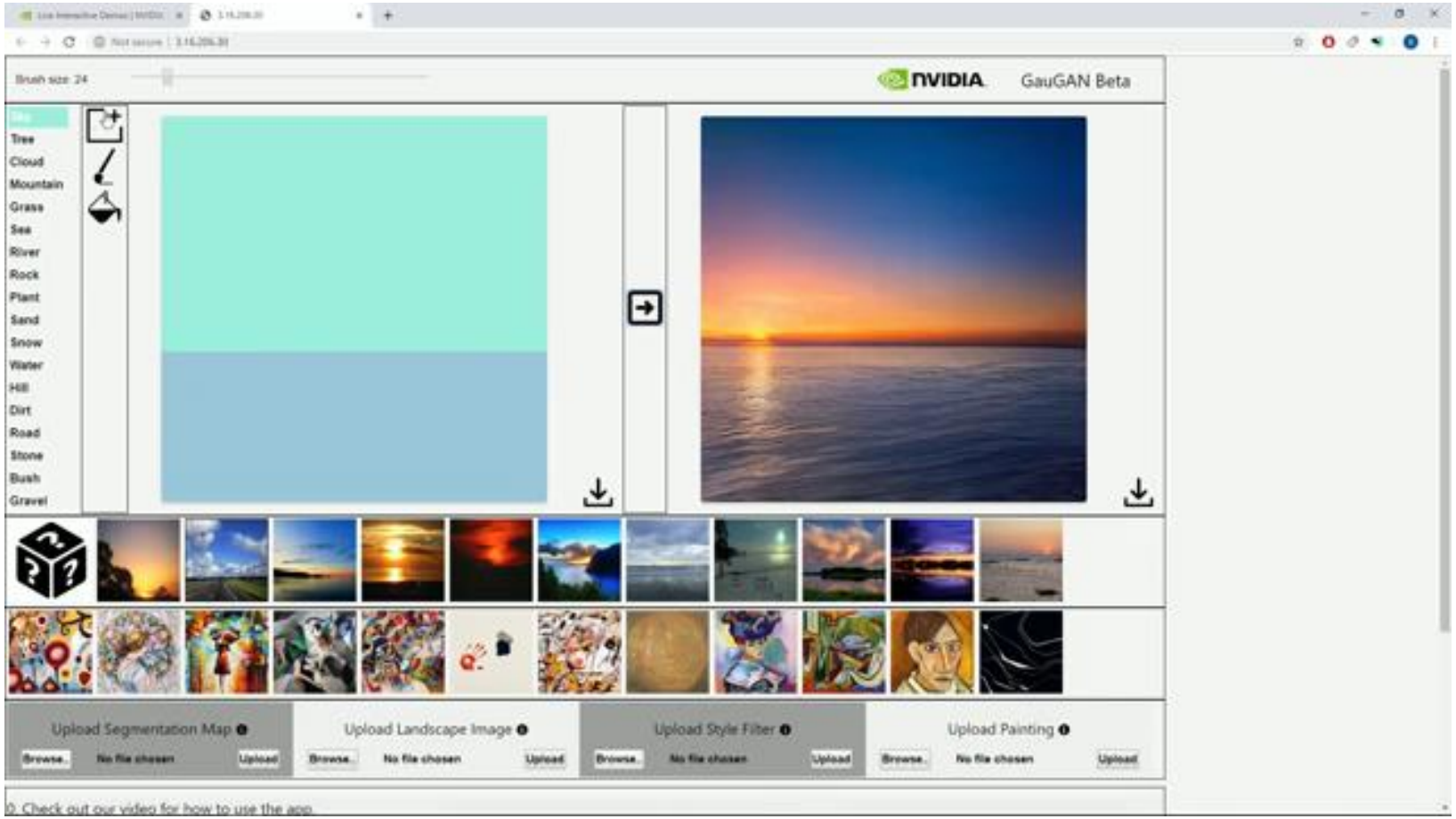
Unsupervised Image to Image Translation

Liu et al (2017)

CycleGAN

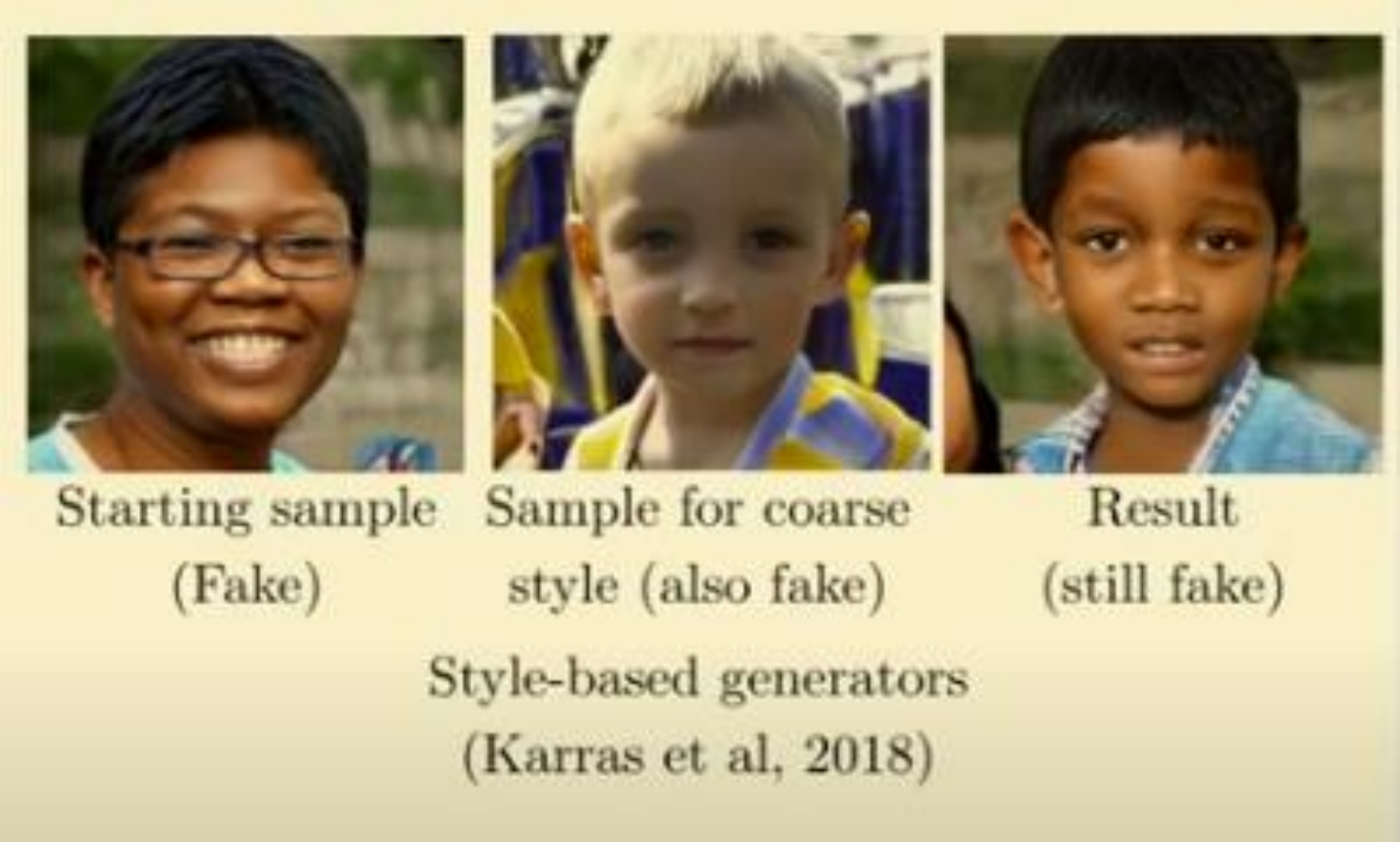
Zhu et al (2017)

GauGan



NVidia

StyleGAN


Starting sample
(Fake)
Sample for coarse
style (also fake)
Result
(still fake)
Style-based generators
(Karras et al, 2018)

GAN

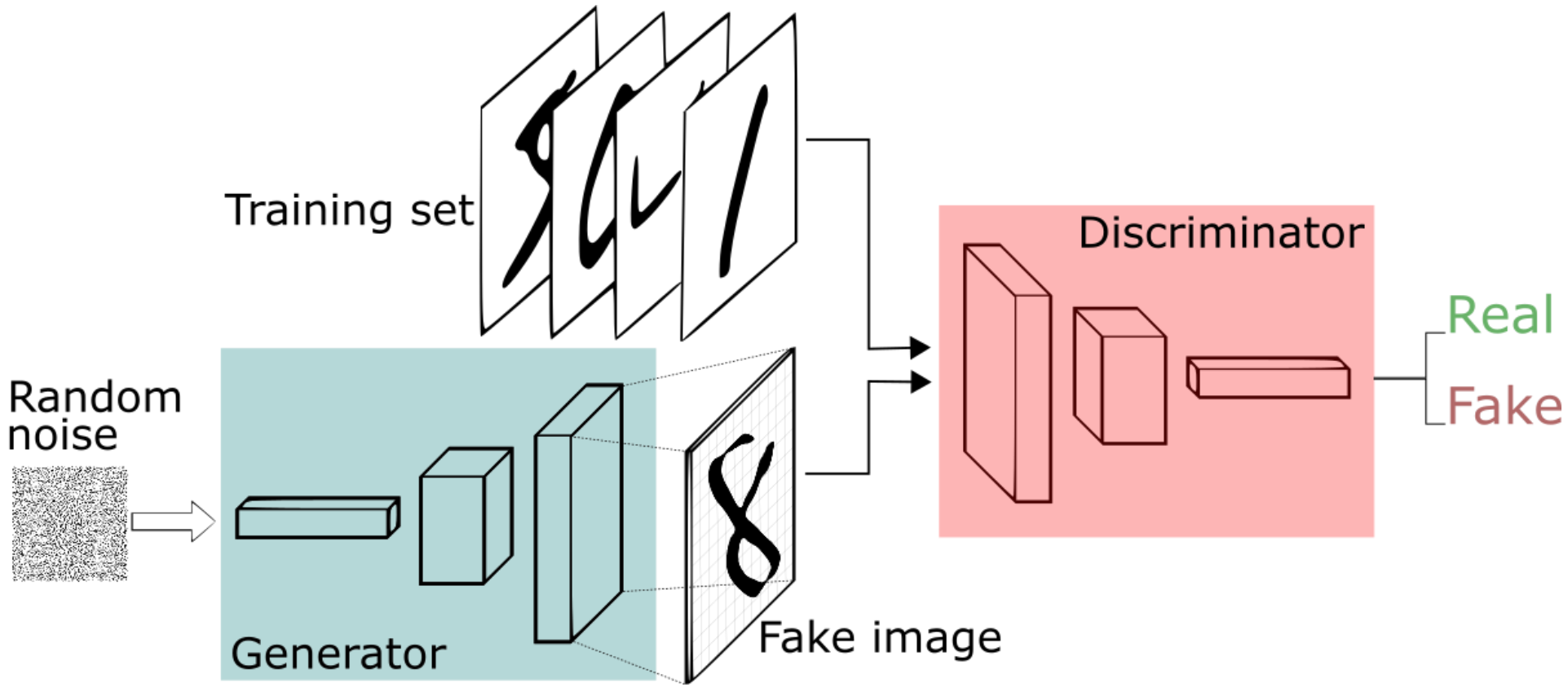


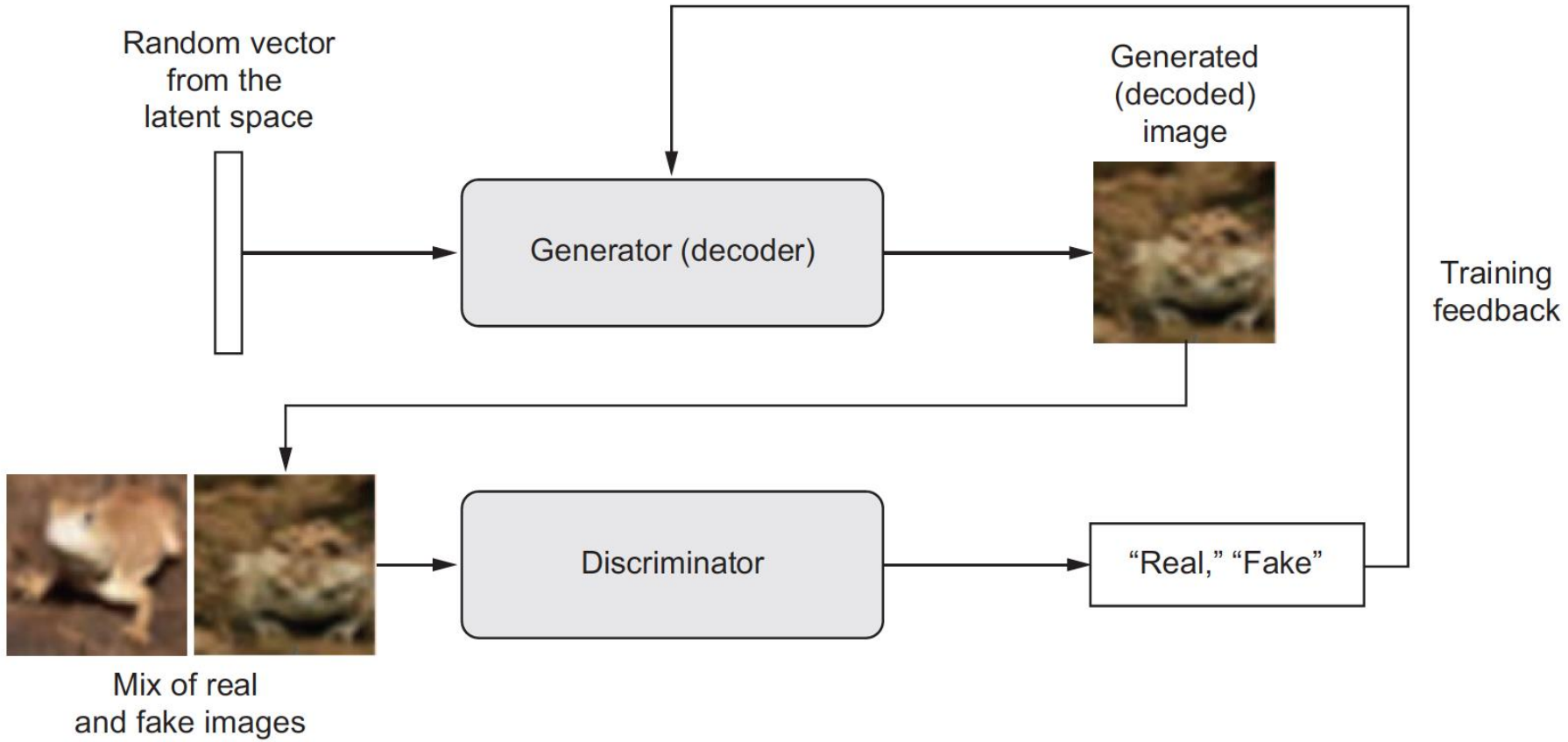
Image credit: Thalles Silva

GAN

- Generator network—Takes as input a random vector (a random point in the latent space), and decodes it into a synthetic image

- Discriminator network (or adversary)—Takes as input an image (real or synthetic), and predicts whether the image came from the training set or was created by the generator network.

GAN

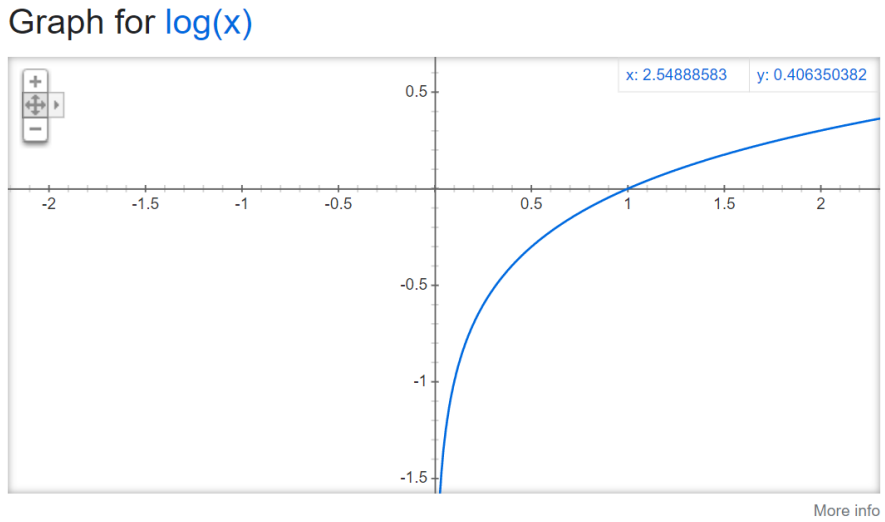

Random vector
from the
latent space
Generator (decoder)
Generated
(decoded)
image
Training
feedback
Discriminator
“Real,” “Fake”
Mix of real
and fake images

GAN: training procedure

Labels: y_real = 1
y_gen = 0

Cost of discriminator

$$J^{(D)} = -\frac{1}{m_{real}} \sum_{i=1}^{m_{real}} y_{real}^{(i)} . \log(D(x^{(i)})) - \frac{1}{m_{gen}} \sum_{i=1}^{m_{gen}} (1 - y_{gen}^{(i)}) . \log(1 - D(G(z^{(i)})))$$



GAN: training procedure

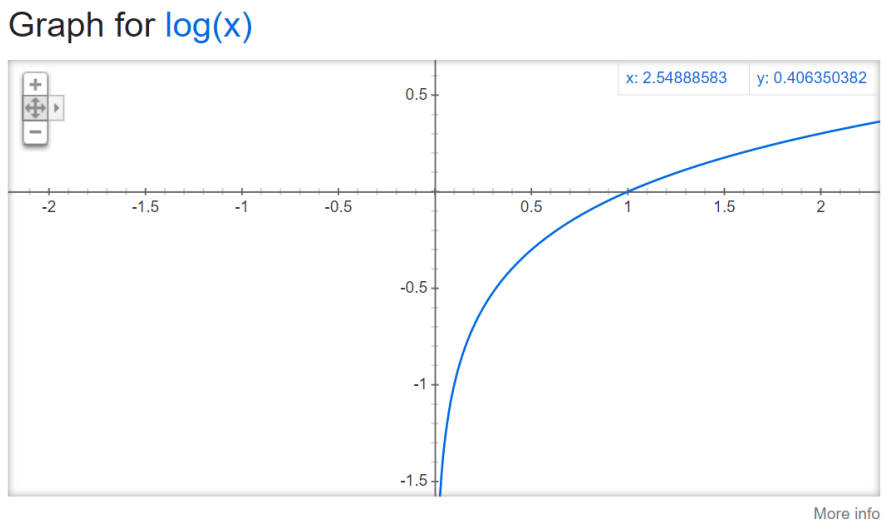
Labels: y_real = 1
y_gen = 0

Cost of discriminator

$$J^{(D)} = -\frac{1}{m_{real}} \sum_{i=1}^{m_{real}} y_{real}^{(i)} . \log(D(x^{(i)})) - \frac{1}{m_{gen}} \sum_{i=1}^{m_{gen}} (1 - y_{gen}^{(i)}) . \log(1 - D(G(z^{(i)})))$$

Cost of generator

$$J^{(G)} = -J^{(D)} = \frac{1}{m_{gen}} \sum_{i=1}^{m_{gen}} \log(1 - D(G(z^{(i)})))$$



GAN: training procedure

Labels: y_real = 1
y_gen = 0

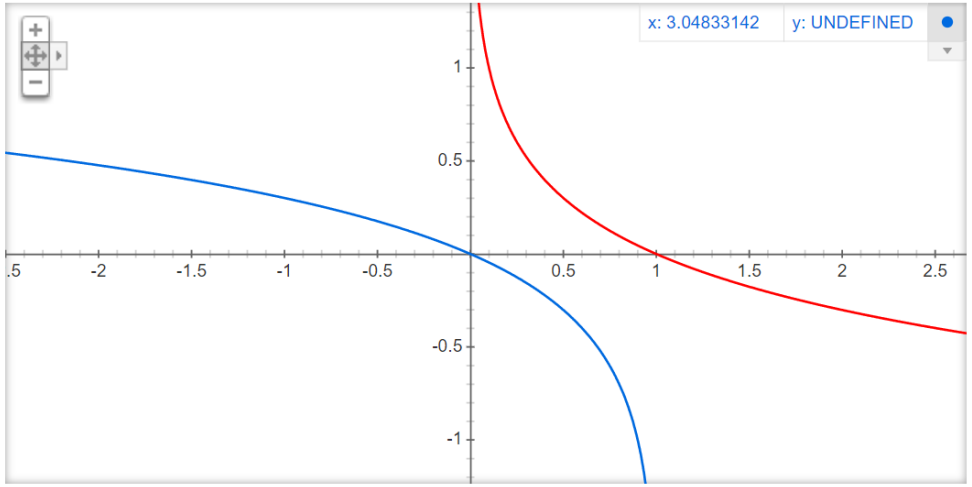
Cost of discriminator

$$J^{(D)} = -\frac{1}{m_{real}} \sum_{i=1}^{m_{real}} y_{real}^{(i)} . \log(D(x^{(i)})) - \frac{1}{m_{gen}} \sum_{i=1}^{m_{gen}} (1 - y_{gen}^{(i)}) . \log(1 - D(G(z^{(i)})))$$

Cost of generator

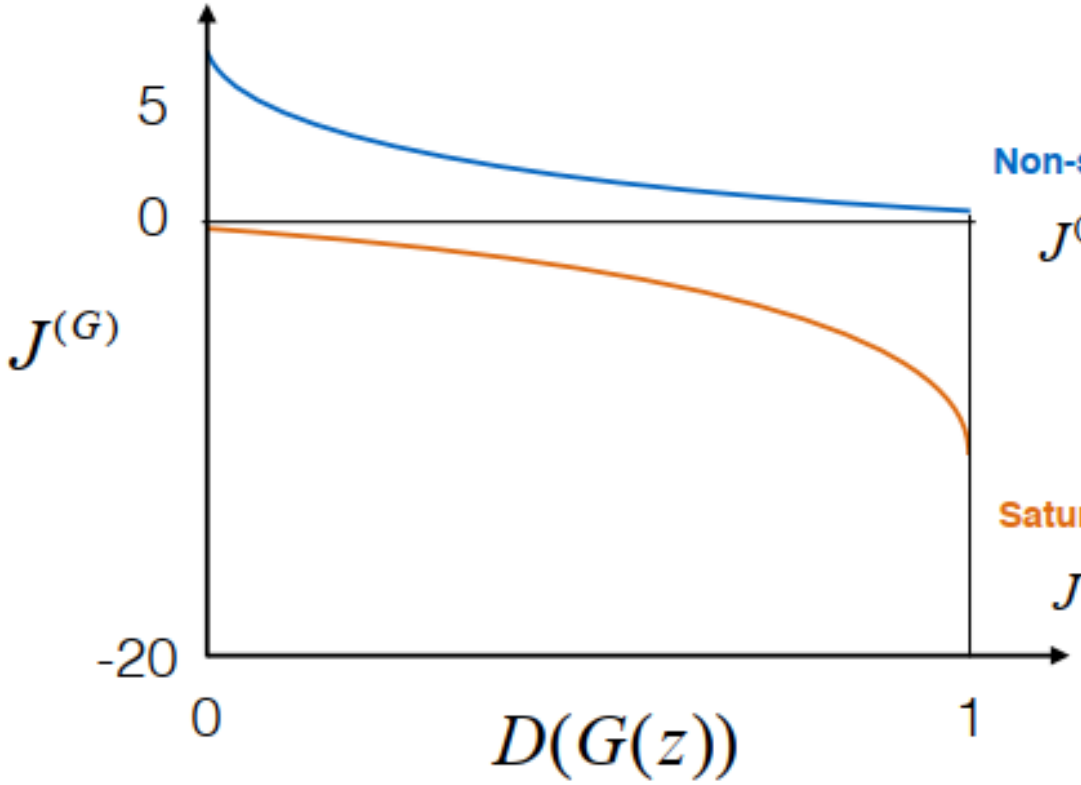
$$J^{(G)} = -\frac{1}{m_{gen}} \sum_{i=1}^{m_{gen}} \log(D(G(z^{(i)})))$$

Graph for log(1-x), -log(x)



GAN: training procedure

Labels: y_real = 1
y_gen = 0



Non-saturating cost

$$J^{(G)} = -\frac{1}{m_{gen}} \sum_{i=1}^{m_{gen}} \log(D(G(z^{(i)})))$$

Saturating cost

$$J^{(G)} = \frac{1}{m_{gen}} \sum_{i=1}^{m_{gen}} \log(1 - D(G(z^{(i)})))$$

Table 1: Generator and discriminator loss functions. The main difference whether the discriminator outputs a probability (MM GAN, NS GAN, DRAGAN) or its output is unbounded (WGAN, WGAN GP, LS GAN, BEGAN), whether the gradient penalty is present (WGAN GP, DRAGAN) and where is it evaluated. We chose those models based on their popularity.

GAN	DISCRIMINATOR LOSS	GENERATOR LOSS
MM GAN	$\mathcal{L}_{\mathrm{D}}^{\mathrm{GAN}} = -\mathbb{E}_{x\sim p_d}[\log(D(x))] - \mathbb{E}_{\hat{x}\sim p_g}[\log(1 - D(\hat{x}))]$	$\mathcal{L}_{\mathrm{G}}^{\mathrm{GAN}} = \mathbb{E}_{\hat{x}\sim p_g}[\log(1 - D(\hat{x}))]$
NS GAN	$\mathcal{L}_{\mathrm{D}}^{\mathrm{NSGAN}} = -\mathbb{E}_{x\sim p_d}[\log(D(x))] - \mathbb{E}_{\hat{x}\sim p_g}[\log(1 - D(\hat{x}))]$	$\mathcal{L}_{\mathrm{G}}^{\mathrm{NSGAN}} = -\mathbb{E}_{\hat{x}\sim p_g}[\log(D(\hat{x}))]$
WGAN	$\mathcal{L}_{\mathrm{D}}^{\mathrm{WGAN}} = -\mathbb{E}_{x\sim p_d}[D(x)] + \mathbb{E}_{\hat{x}\sim p_g}[D(\hat{x})]$	$\mathcal{L}_{\mathrm{G}}^{\mathrm{WGAN}} = -\mathbb{E}_{\hat{x}\sim p_g}[D(\hat{x})]$
WGAN GP	$\mathcal{L}_{\mathrm{D}}^{\mathrm{WGANGP}} = \mathcal{L}_{\mathrm{D}}^{\mathrm{WGAN}} + \lambda\mathbb{E}_{\hat{x}\sim p_g}[(\|\nabla D(\alpha x + (1 - \alpha\hat{x})\|_2 - 1)^2]$	$\mathcal{L}_{\mathrm{G}}^{\mathrm{WGANGP}} = -\mathbb{E}_{\hat{x}\sim p_g}[D(\hat{x})]$
LS GAN	$\mathcal{L}_{\mathrm{D}}^{\mathrm{LSGAN}} = -\mathbb{E}_{x\sim p_d}[(D(x) - 1)^2] + \mathbb{E}_{\hat{x}\sim p_g}[D(\hat{x})^2]$	$\mathcal{L}_{\mathrm{G}}^{\mathrm{LSGAN}} = -\mathbb{E}_{\hat{x}\sim p_g}[(D(\hat{x} - 1)^2]$
DRAGAN	$\mathcal{L}_{\mathrm{D}}^{\mathrm{DRAGAN}} = \mathcal{L}_{\mathrm{D}}^{\mathrm{GAN}} + \lambda\mathbb{E}_{\hat{x}\sim p_d+\mathcal{N}(0,c)}[(\|\nabla D(\hat{x})\|_2 - 1)^2]$	$\mathcal{L}_{\mathrm{G}}^{\mathrm{DRAGAN}} = \mathbb{E}_{\hat{x}\sim p_g}[\log(1 - D(\hat{x}))]$
BEGAN	$\mathcal{L}_{\mathrm{D}}^{\mathrm{BEGAN}} = \mathbb{E}_{x\sim p_d}[\|x - \mathrm{AE}(x)\|_1] - k_t\mathbb{E}_{\hat{x}\sim p_g}[\|\hat{x} - \mathrm{AE}(\hat{x})\|_1]$	$\mathcal{L}_{\mathrm{G}}^{\mathrm{BEGAN}} = \mathbb{E}_{\hat{x}\sim p_g}[\|\hat{x} - \mathrm{AE}(\hat{x})\|_1]$

Kurach et al (2018)

GAN

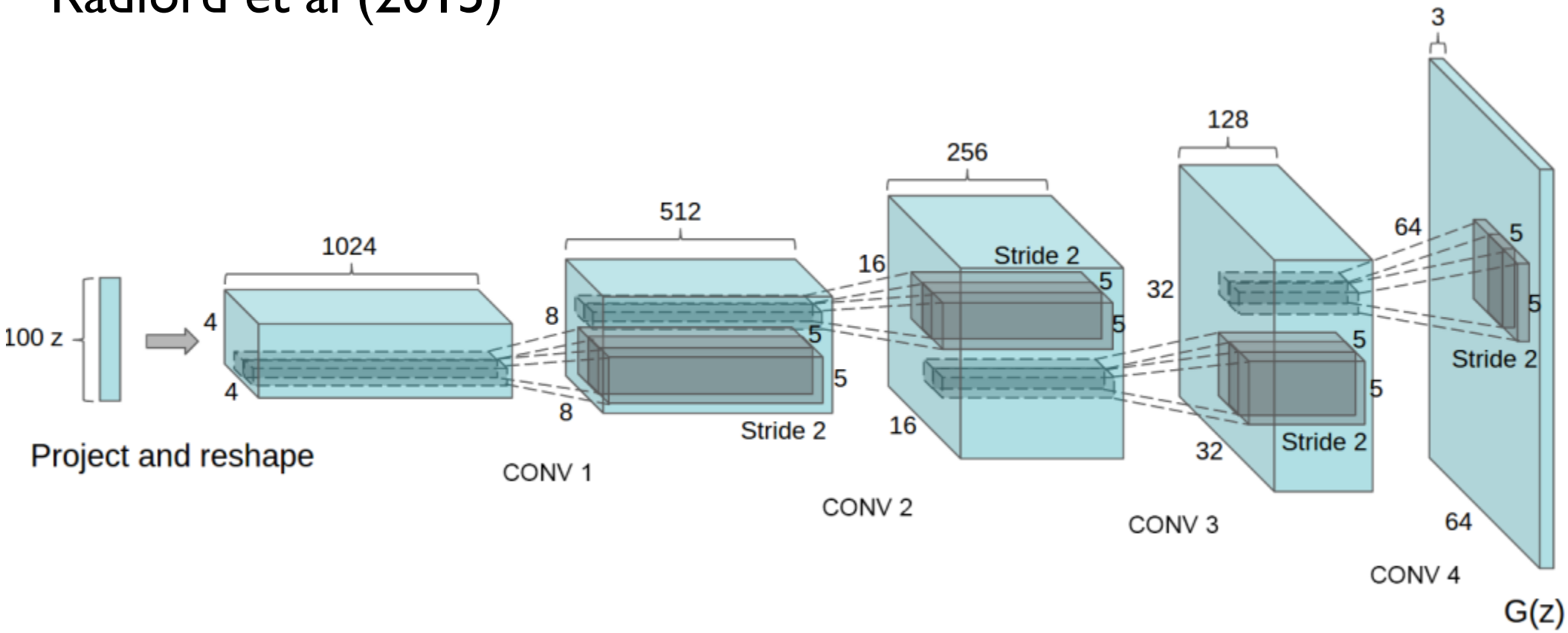
- The optimization minimum isn't fixed, unlike what we had before.
- Normally, gradient descent consists of rolling down hills in a static loss landscape.
- But with a GAN, every step taken down the hill changes the entire landscape a little. It's a dynamic system where the optimization process is seeking not a minimum, but an equilibrium between two forces.

GAN progress

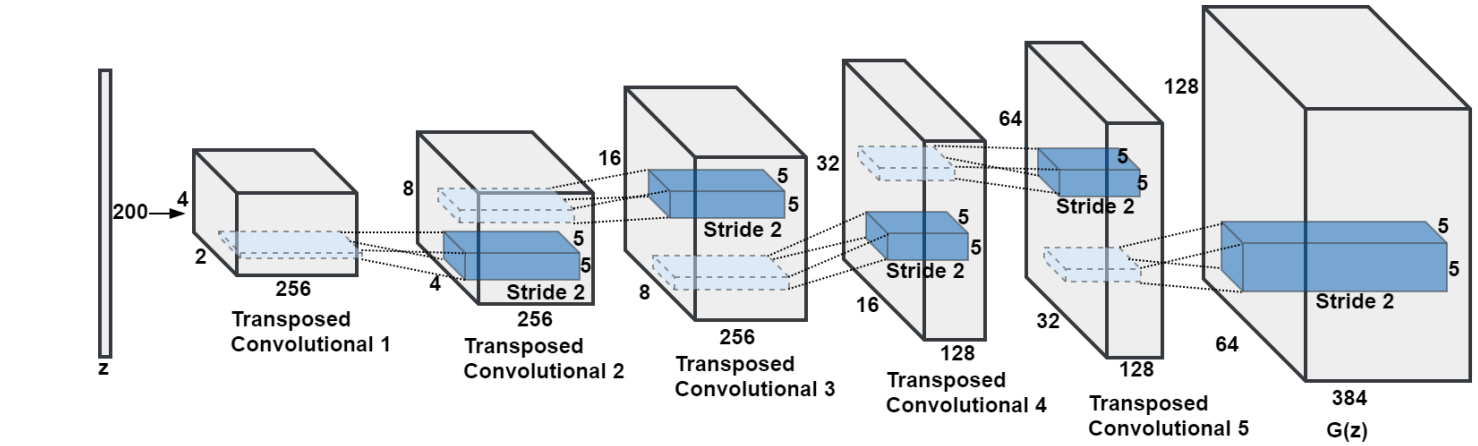
From Ian Goodfellow

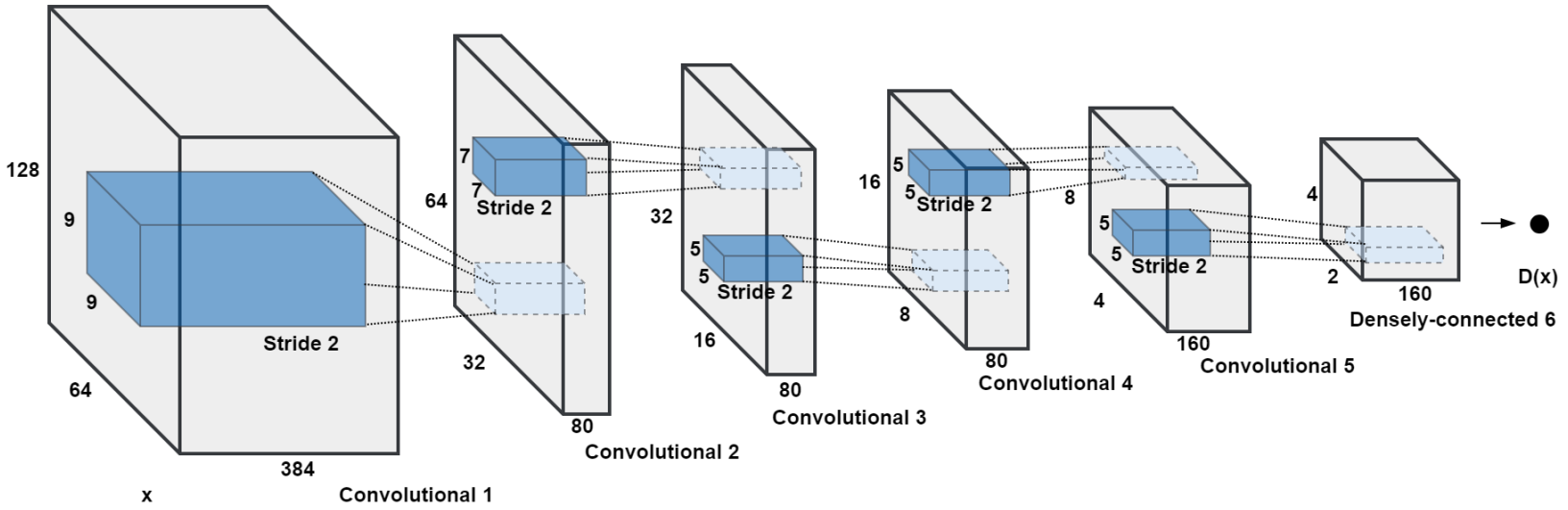
DCGAN

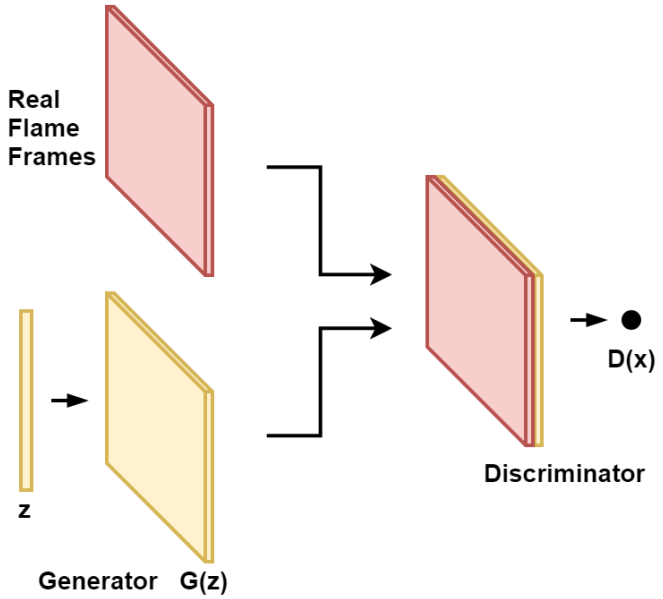
- Radford et al (2015)

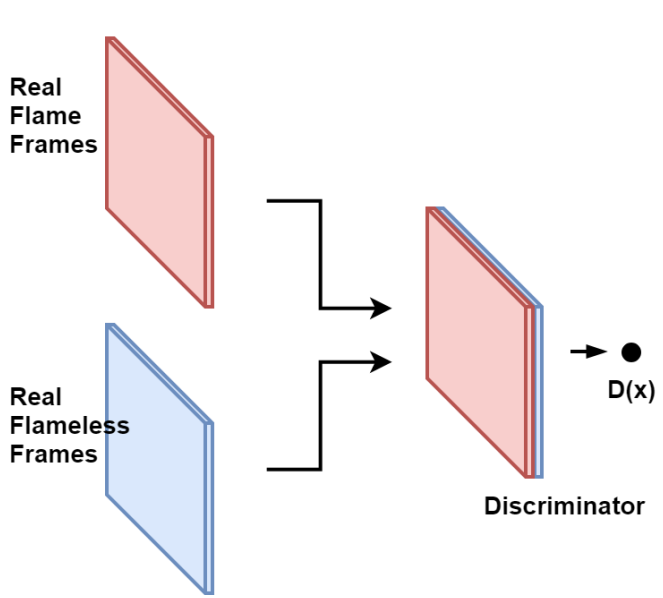


DCGAN


(a)
200
z
4
2
256
Transposed
Convolutional 1
8
4
Stride 2
5
5
256
Transposed
Convolutional 2
16
8
Stride 2
256
Transposed
Convolutional 3
32
16
Stride 2
128
Transposed
Convolutional 4
64
32
Stride 2
128
Transposed
Convolutional 5
128
64
Stride 2
384
G(z)


(b)
128
9
9
Stride 2
64
x
384
Convolutional 1
64
7
7
Stride 2
32
80
Convolutional 2
32
5
5
Stride 2
16
80
Convolutional 3
16
5
5
Stride 2
8
80
Convolutional 4
8
5
5
Stride 2
4
160
Convolutional 5
4
2
160
Densely-connected 6
D(x)


(c)
Real
Flame
Frames
z
Generator G(z)
D(x)
Discriminator


(d)
Real
Flame
Frames
Real
Flameless
Frames
D(x)
Discriminator

DCGAN - deep convolutional GAN

1. A *generator* network maps vectors of shape `(latent_dim,)` to images of shape `(32, 32, 3)`.

2. A *discriminator* network maps images of shape `(32, 32, 3)` to a binary score estimating the probability that the image is real.

3. A gan network chains the generator and the discriminator together: `gan(x) = discriminator(generator(x))`.

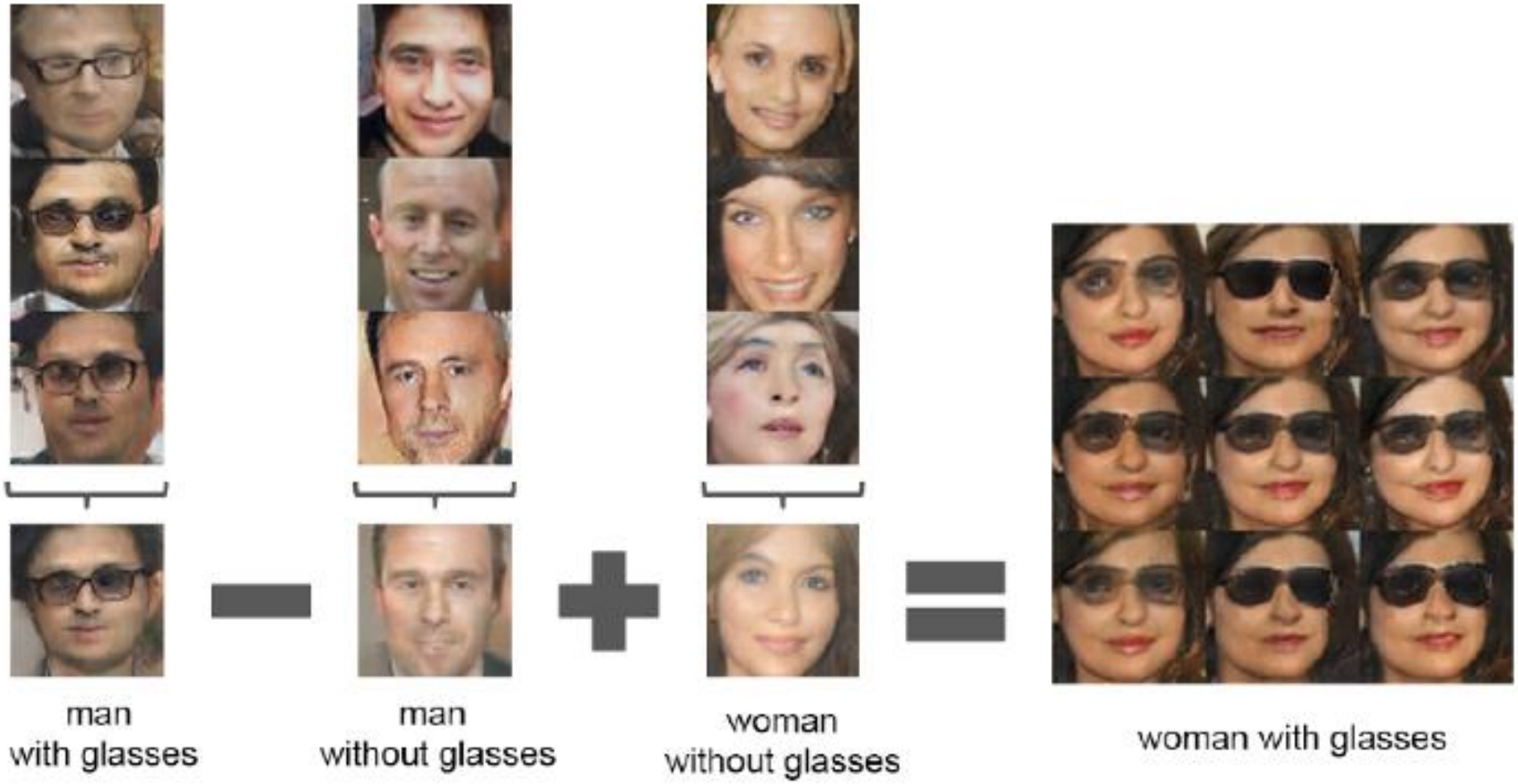
   Thus this gan network maps latent space vectors to the discriminator's assessment of the realism of these latent vectors as decoded by the generator.

DCGAN - deep convolutional GAN

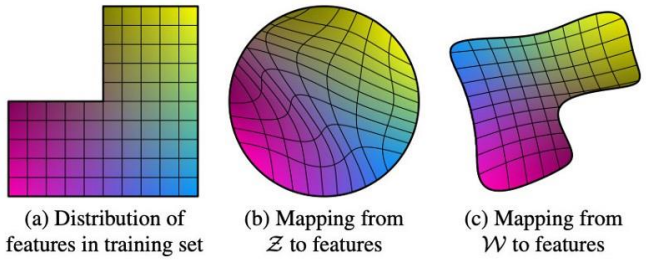
4. You train the *discriminator* using examples of real and fake images along with "real"/"fake" labels, just as you train any regular image-classification model.

5. To train the *generator*, you use the gradients of the *generator's* weights with regard to the loss of the gan model. This means, at every step, you move the weights of the *generator* in a direction that makes the *discriminator* more likely to classify as "real" the images decoded by the generator. In other words, you train the *generator* to fool the *discriminator*.

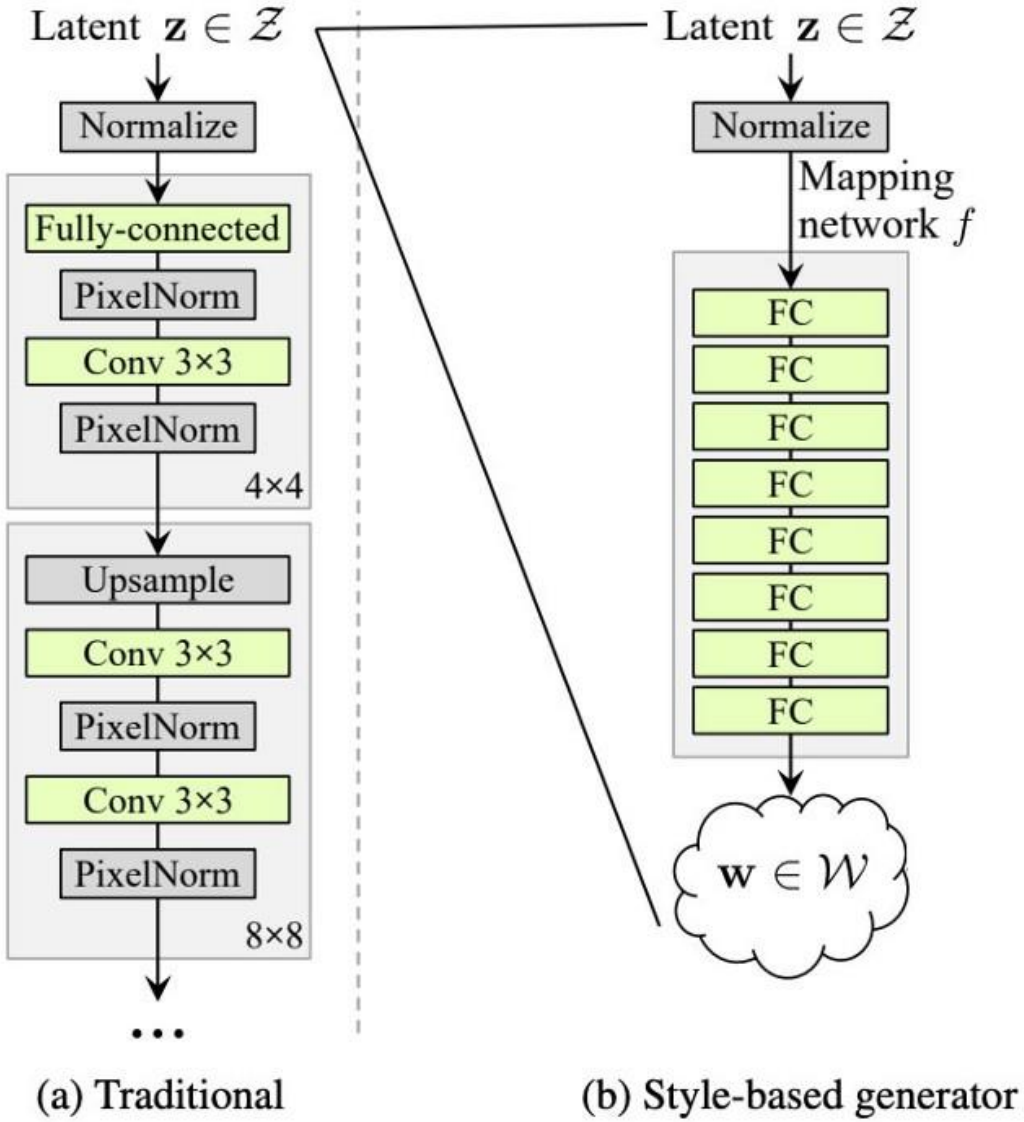
DCGAN – Vector Space Arithmetic



StyleGAN

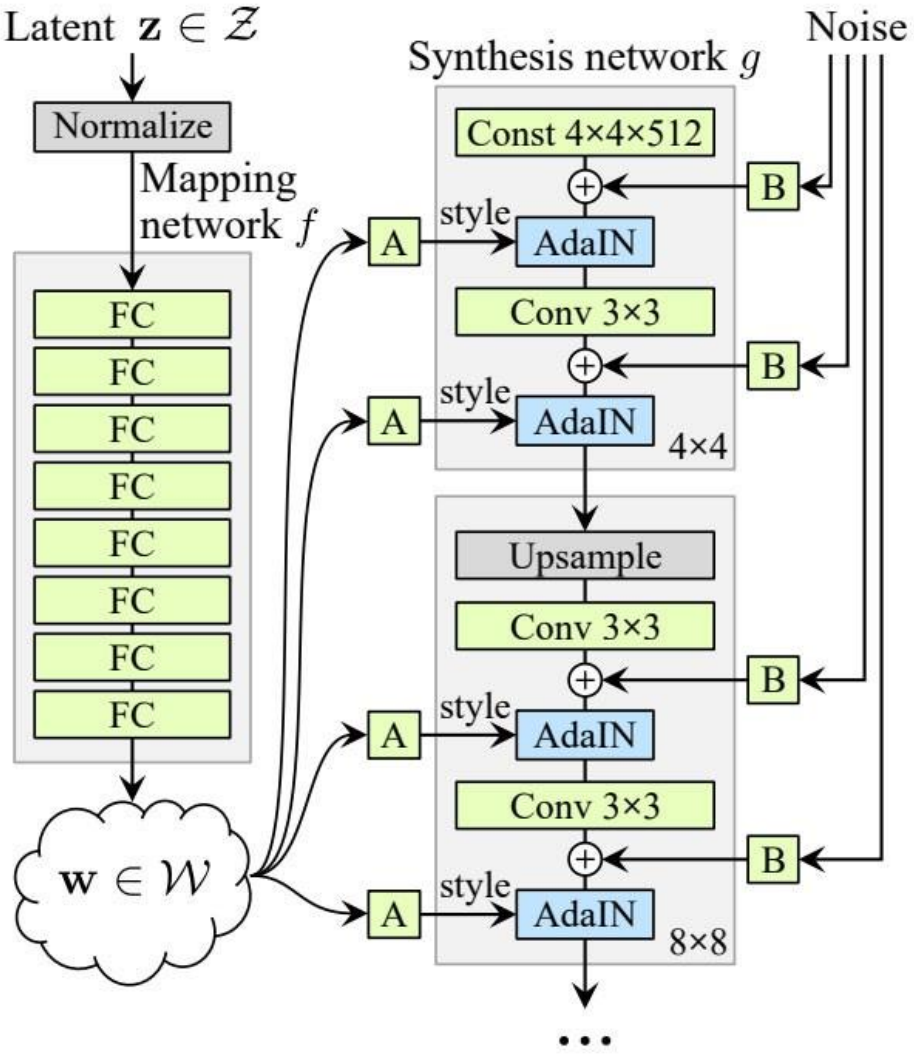


Source



source

StyleGAN



(b) Style-based generator

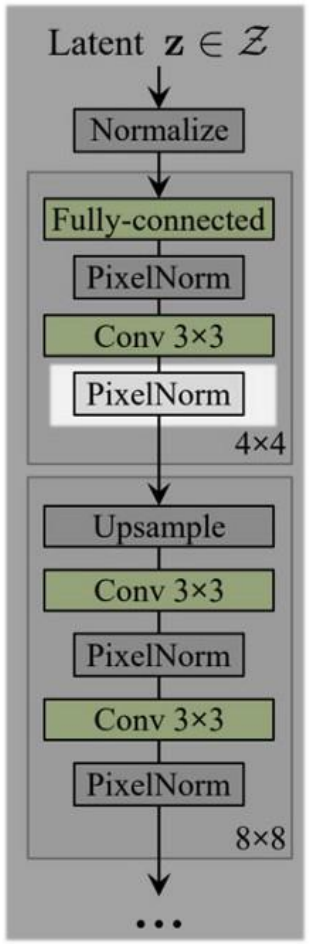
source

StyleGAN

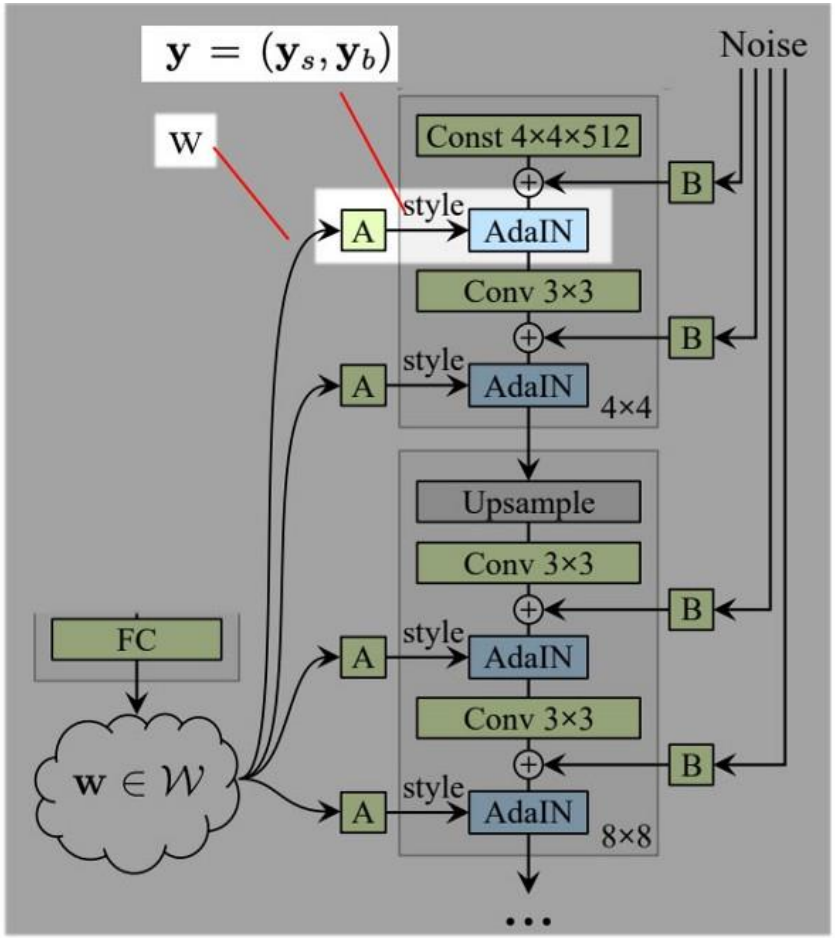
feature map i — the mean & standard deviation of t

$$\mathrm{AdaIN}(\mathbf{x}_i, \mathbf{y}) = \mathbf{y}_{s,i} \frac{\mathbf{x}_i - \mu(\mathbf{x}_i)}{\sigma(\mathbf{x}_i)} + \mathbf{y}_{b,i}$$

normalize the feature map
(instance normalizati



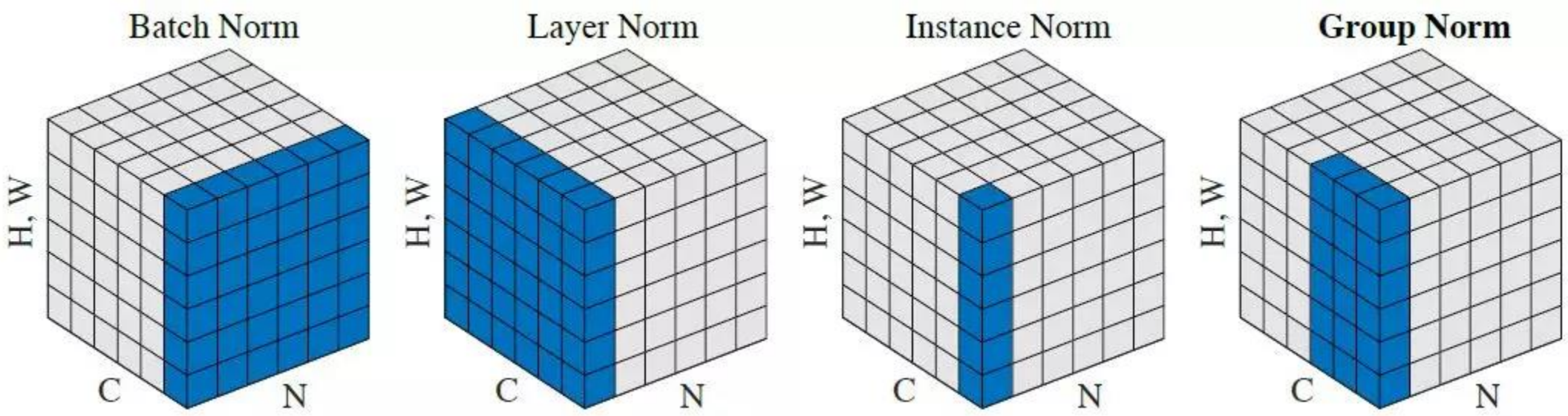
(a) Traditional



(b) Style-based generator

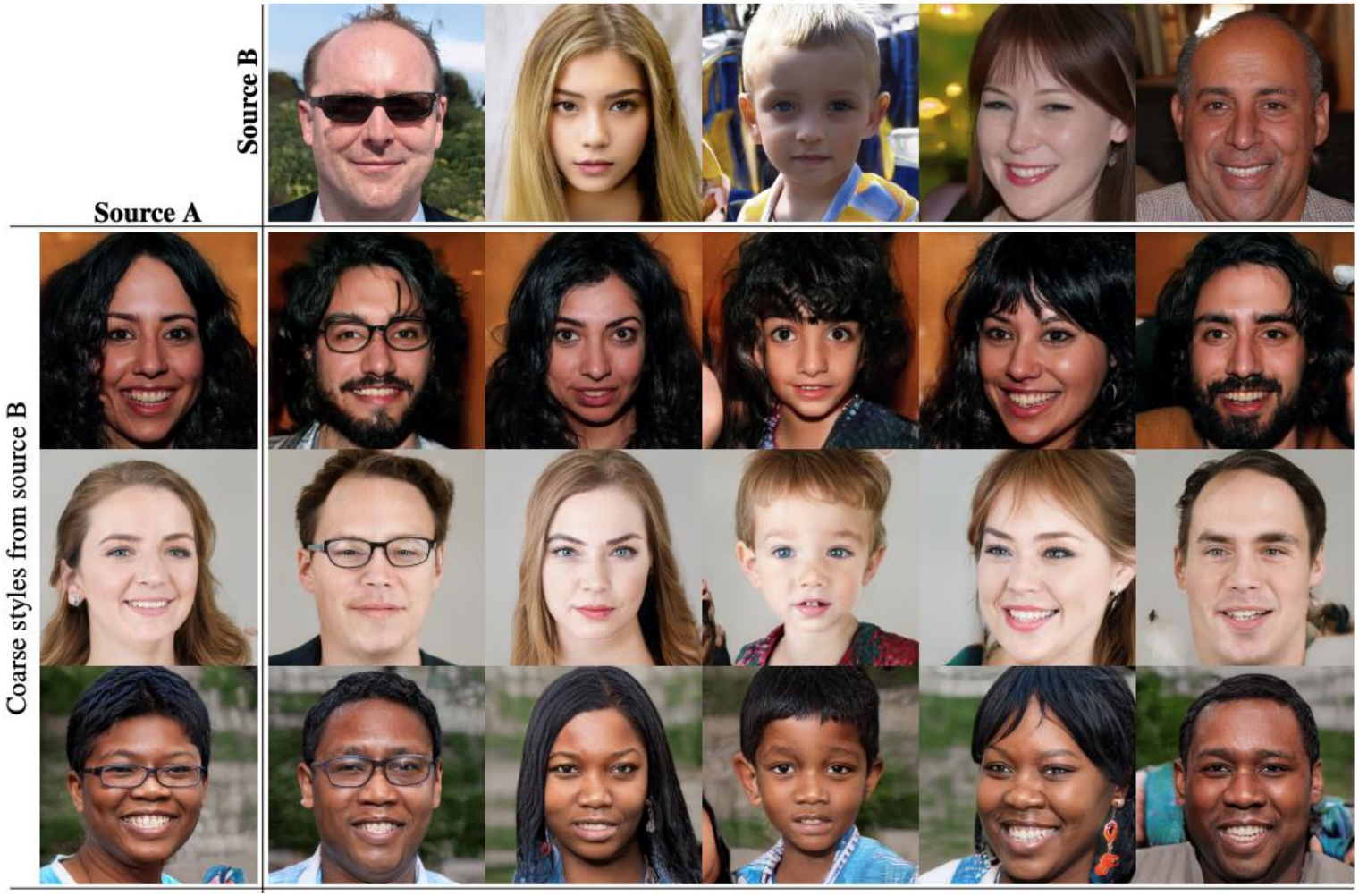
[source]

(Normalization)

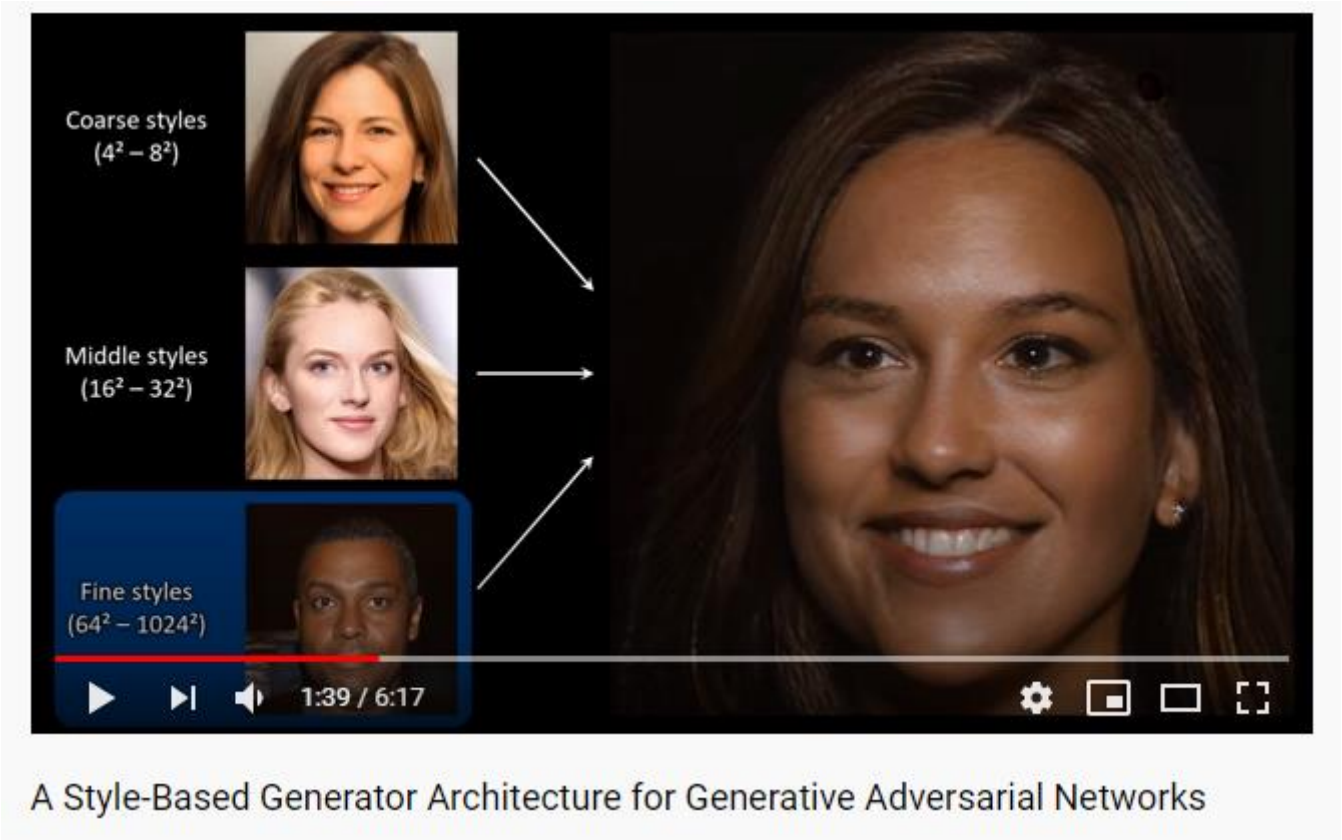


StyleGAN

[Read more]

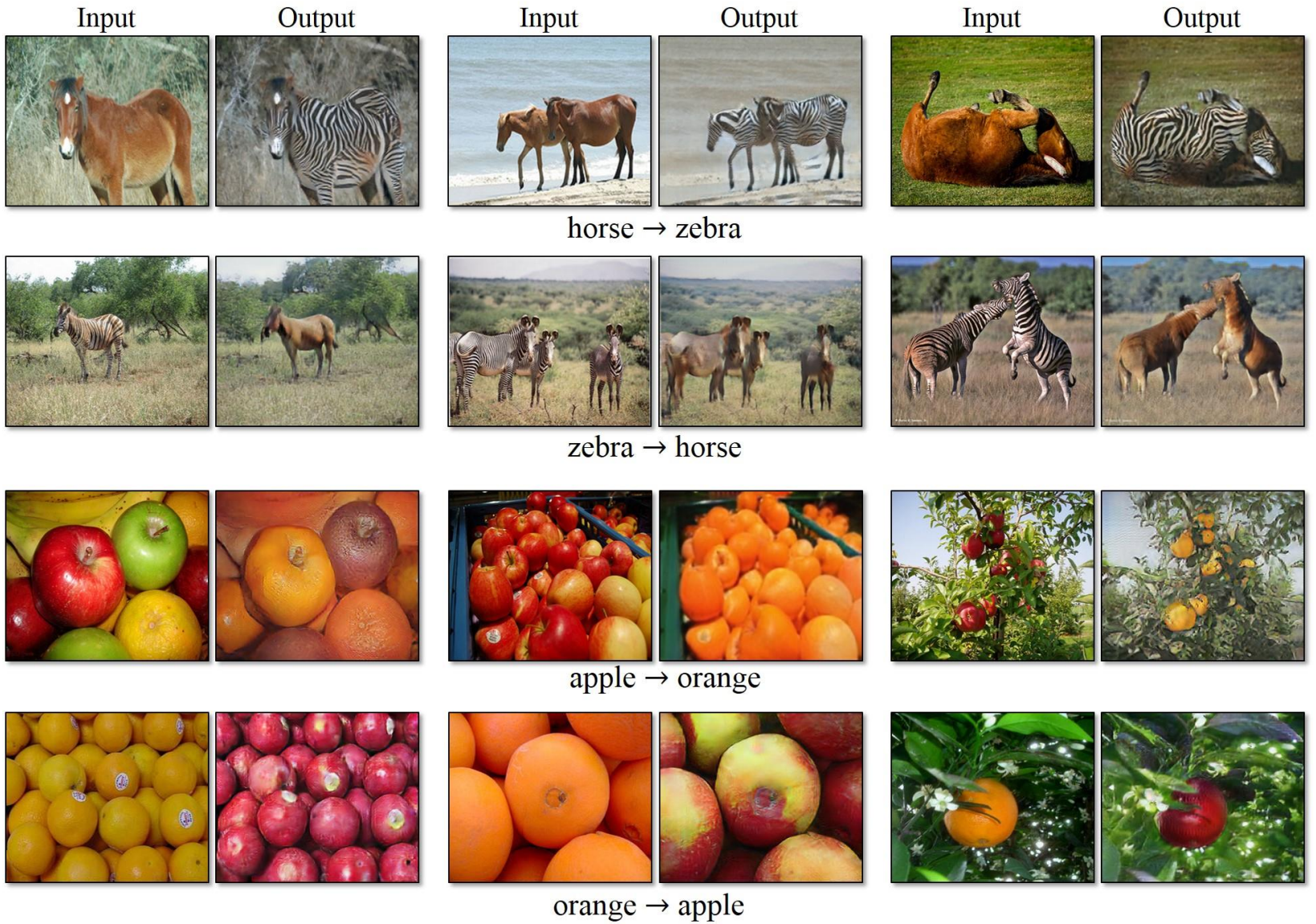


See more

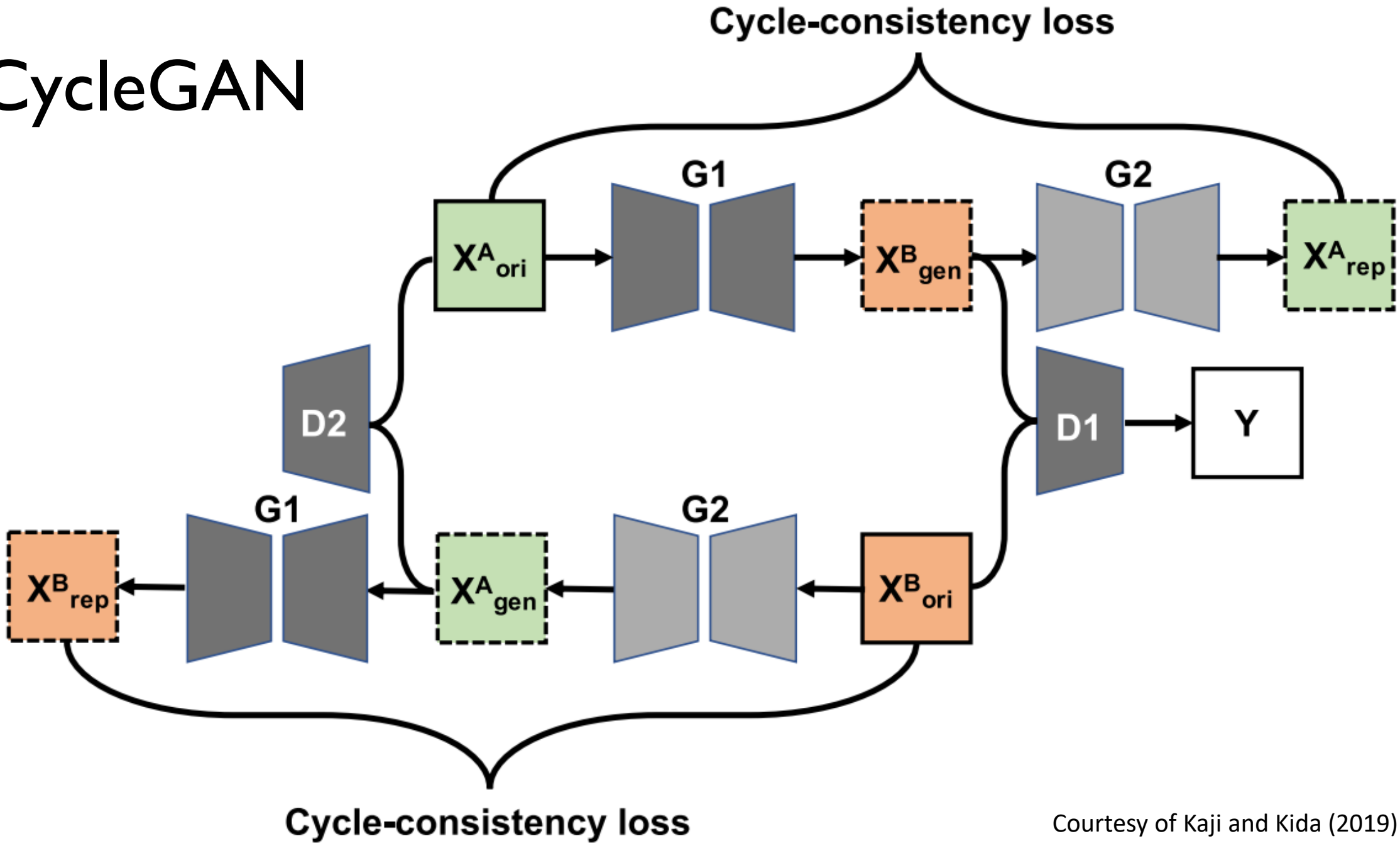


https://www.youtube.com/watch?v=kSLJriaOumA&feature=youtu.be

Image to Image Translation

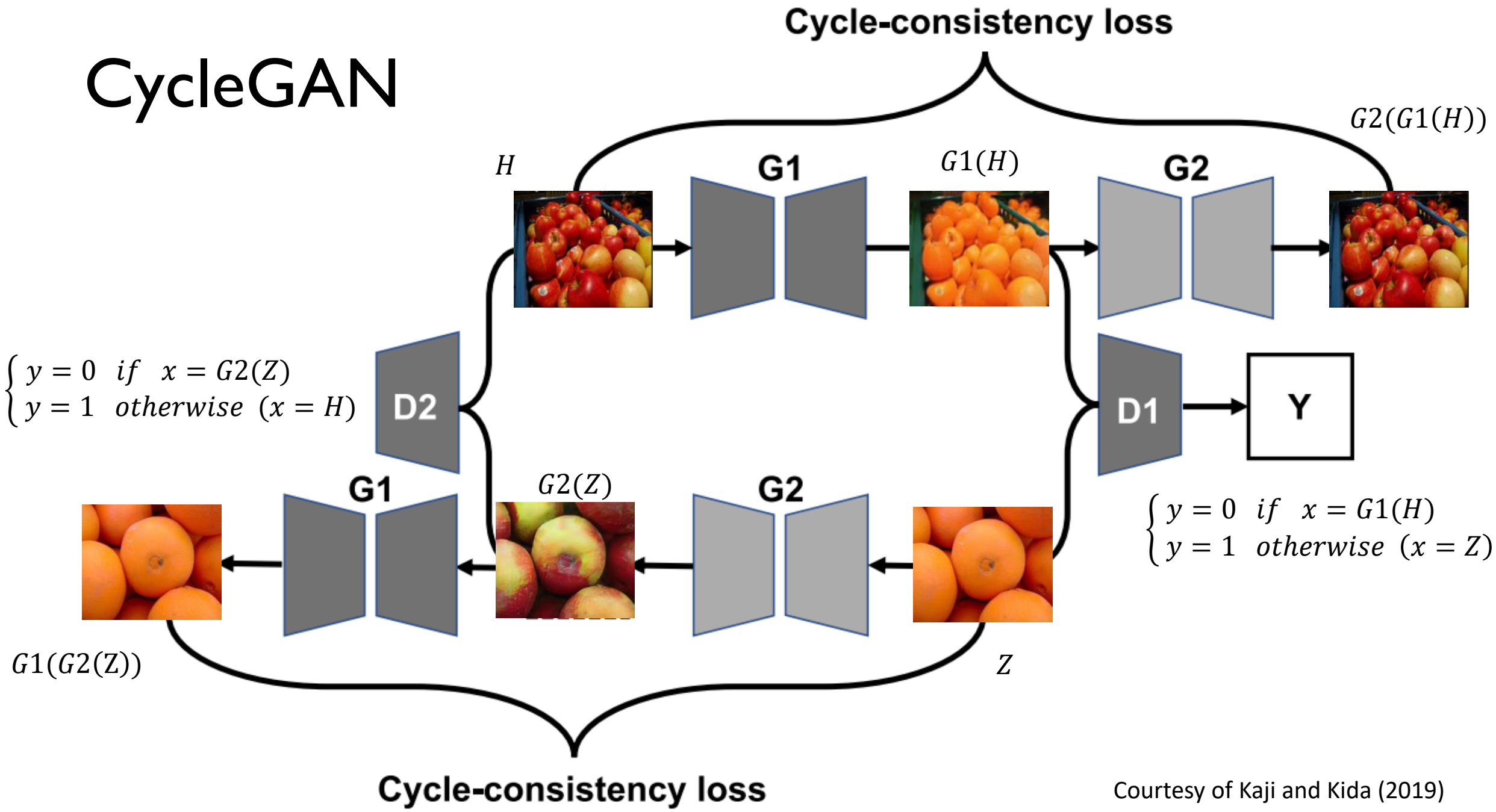


CycleGAN



Courtesy of Kaji and Kida (2019)

CycleGAN



Courtesy of Kaji and Kida (2019)

CycleGAN

$$J^{(D1)} = -\frac{1}{m_{real}}\sum_{i=1}^{m_{real}}\log(D1(z^{(i)})) - \frac{1}{m_{gen}}\sum_{i=1}^{m_{gen}}\log(1 - D1(G1(H^{(i)})))$$

$$J^{(G1)} = -\frac{1}{m_{gen}}\sum_{i=1}^{m_{gen}}\log(D1(G1(H^{(i)})))$$

$$J^{(D2)} = -\frac{1}{m_{real}}\sum_{i=1}^{m_{real}}\log(D2(h^{(i)})) - \frac{1}{m_{gen}}\sum_{i=1}^{m_{gen}}\log(1 - D2(G2(Z^{(i)})))$$

$$J^{(G2)} = -\frac{1}{m_{gen}}\sum_{i=1}^{m_{gen}}\log(D2(G2(Z^{(i)})))$$

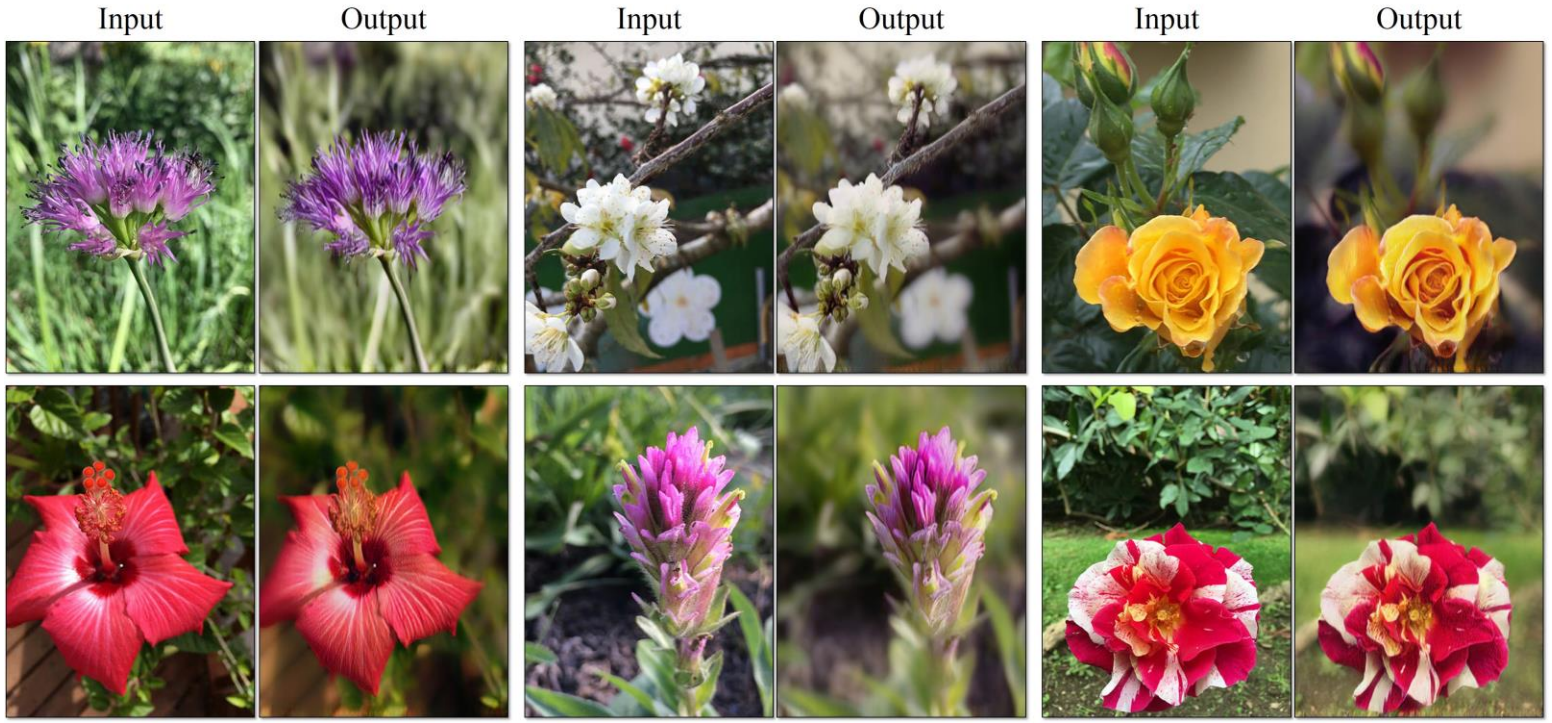
$$J = J^{(D1)} + J^{(G1)} + J^{(D2)} + J^{(G2)} + \lambda J^{cycle}$$

$$J^{cycle} = \frac{1}{m_{gen}}\sum_{i=1}^{m_{gen}} \| G2(G1(H^{(i)}) - H^{(i)} \|_1 + \frac{1}{m_{gen}}\sum_{i=1}^{m_{gen}} \| G1(G2(Z^{(i)}) - Z^{(i)} \|_1$$

Read more

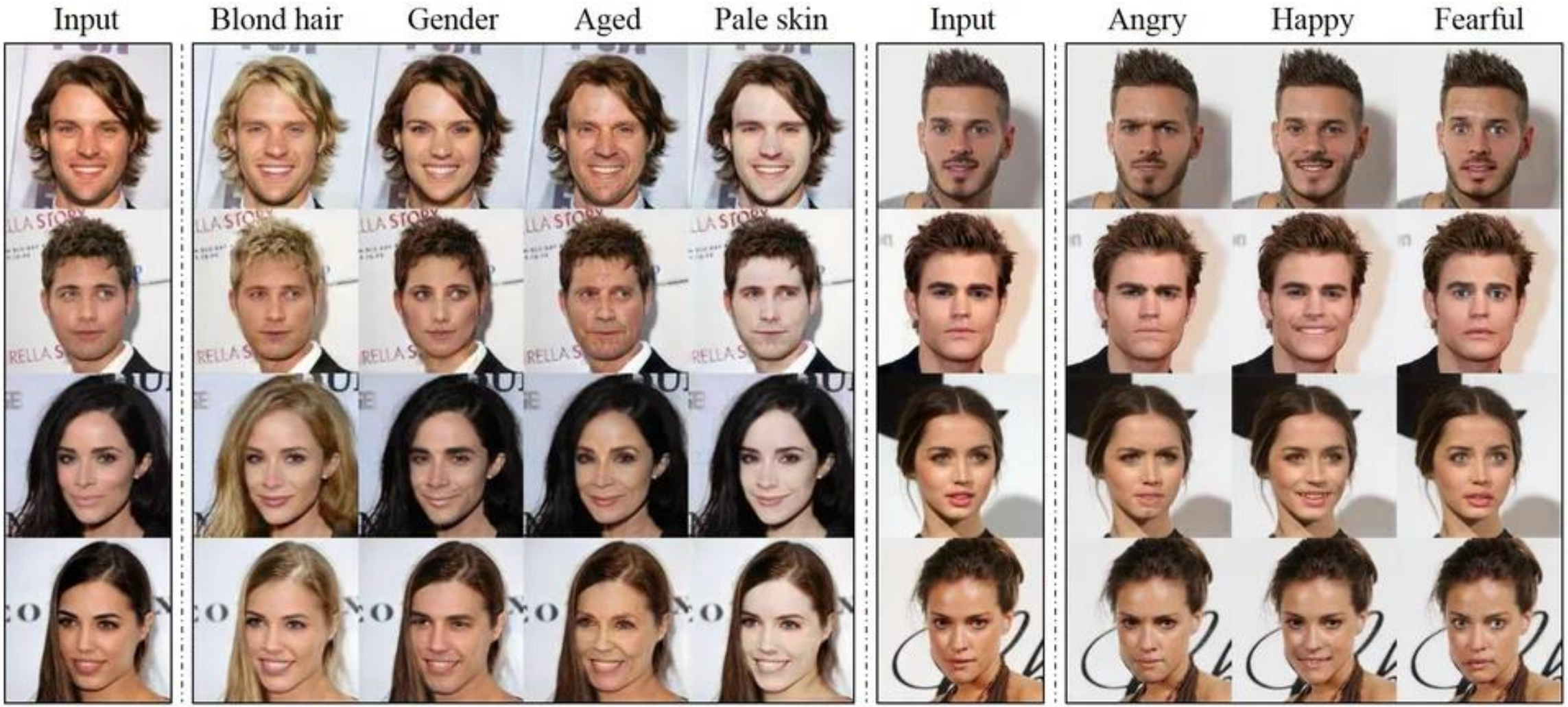
More examples for CycleGAN

- https://junyanz.github.io/CycleGAN/



GTA → Cityscapes

StarGAN


Input
Blond hair
Gender
Aged
Pale skin
Input
Angry
Happy
Fearful

Choi et al (2018)

StarGAN

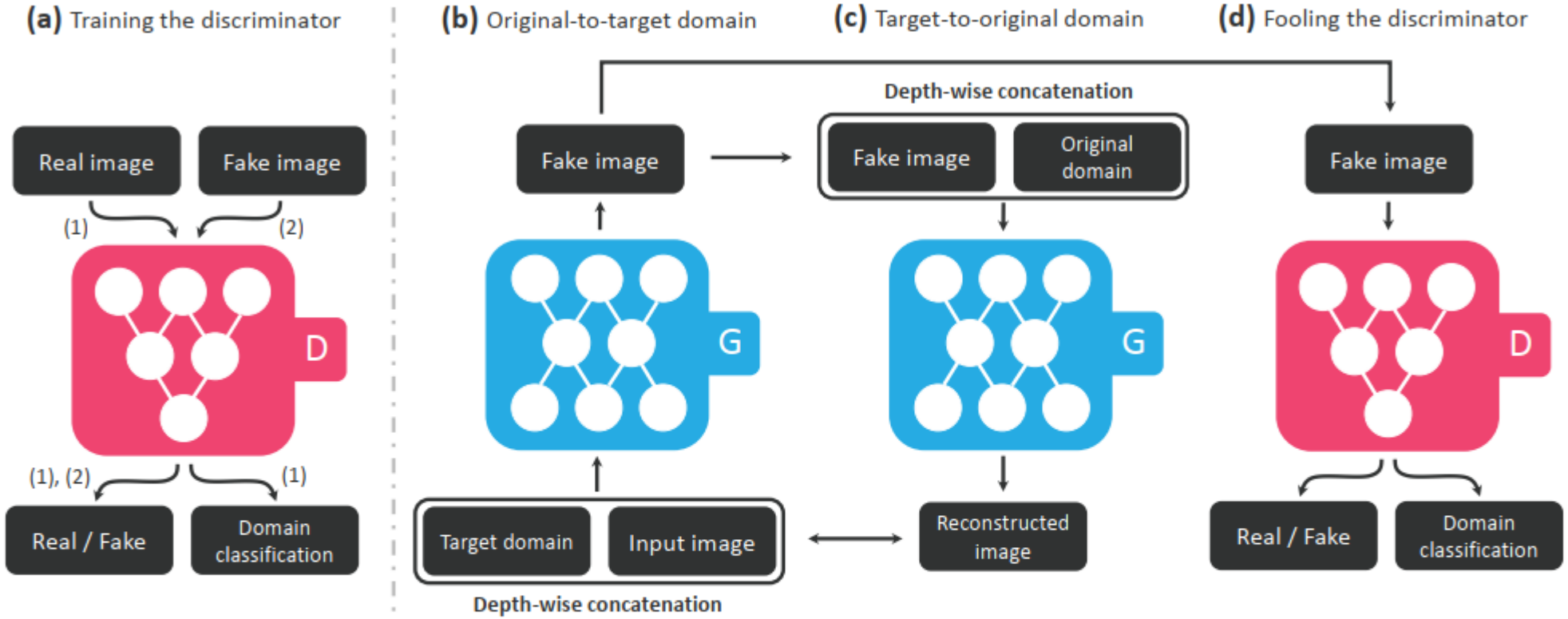
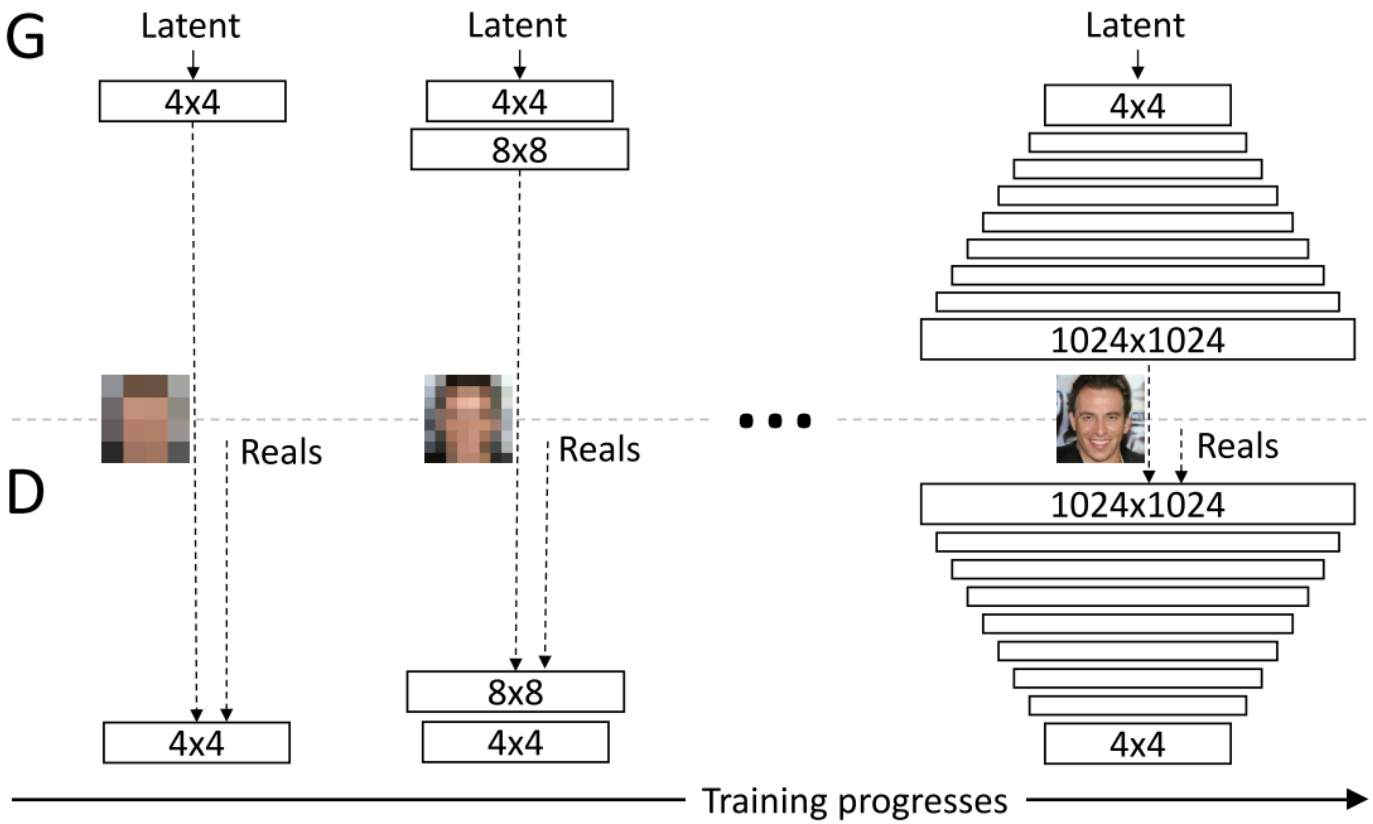


Figure 3. Overview of StarGAN, consisting of two modules, a discriminator D and a generator G. **(a)** D learns to distinguish between real and fake images and classify the real images to its corresponding domain. **(b)** G takes in as input both the image and target domain label and generates an fake image. The target domain label is spatially replicated and concatenated with the input image. **(c)** G tries to reconstruct the original image from the fake image given the original domain label. **(d)** G tries to generate images indistinguishable from real images and classifiable as target domain by D.

Progressive GANs

Karras et al (2018)

- https://arxiv.org/abs/1710.10196



Mode collapse

It consists in the **generator "collapsing"** and always generating a single image for every possible latent vector fed as input.

Read more about GANs, their variants (super-resolution, face inpainting, etc), and their common problems.

https://developers.google.com/machine-learning/gan

GANs: sample code

DCGAN – How to train?

1. Draw random points in the latent space (random noise).
2. Generate images with *generator* using this random noise.
3. Mix the generated images with real ones.
4. Train discriminator using these mixed images, with corresponding targets: either "real" (for the real images) or "fake" (for the generated images).
5. Draw new random points in the latent space.
6. Train gan using these random vectors, with targets that all say "these are real images." This updates the weights of the *generator* (only, because the *discriminator* is frozen inside gan) to move them toward getting the *discriminator* to predict "these are real images" for generated images: this trains the *generator* to fool the *discriminator*.

Generative models

Prisma

Generative models

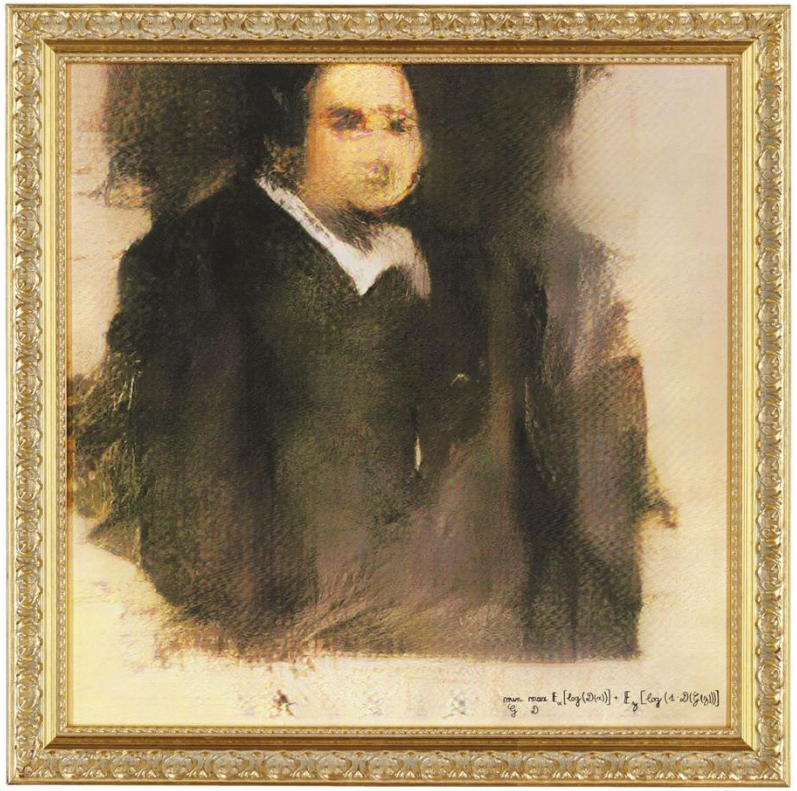
DeepDream

Generative models

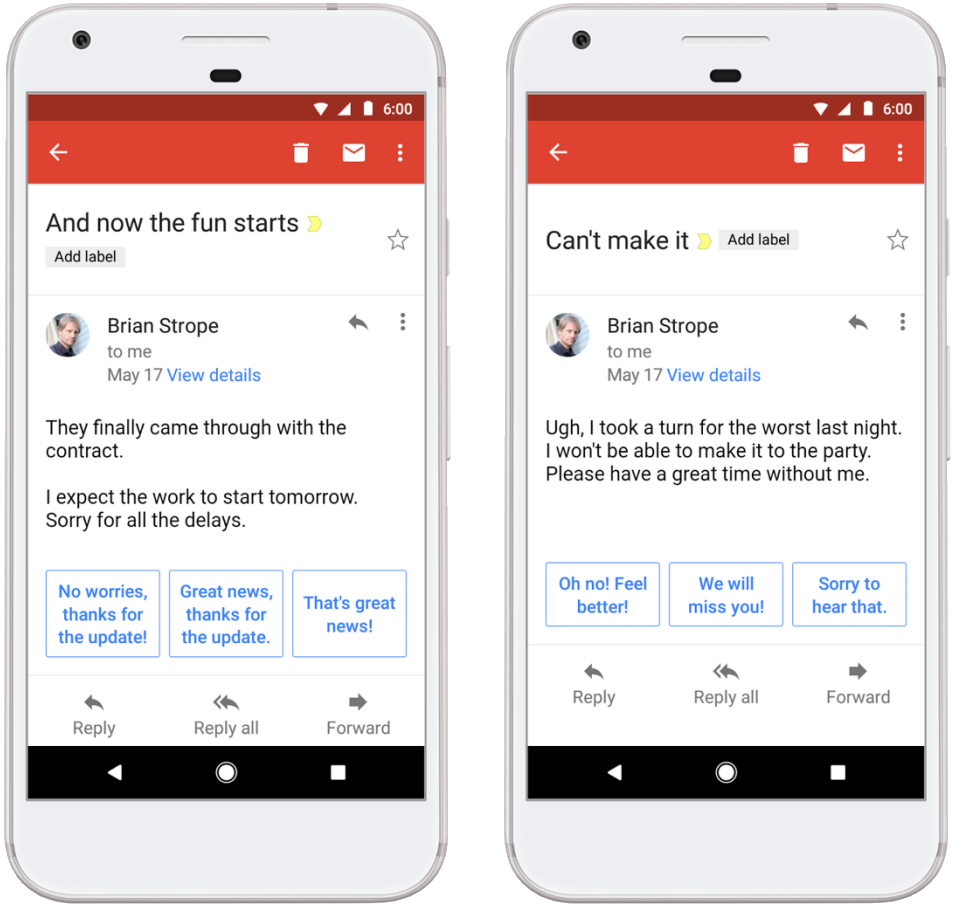
Sunspring

Generative models

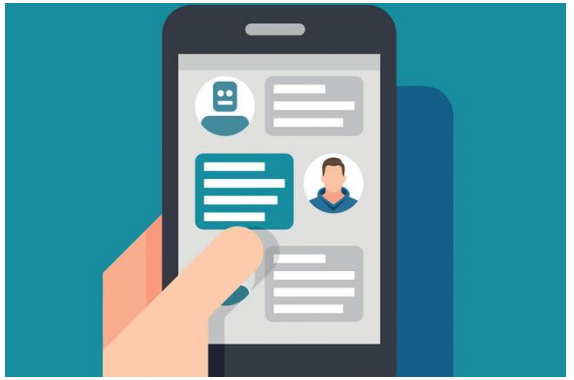
This AI-generated portrait just sold at auction for $432K



Text generation with LSTM



سواران گردنکشان دسته دید
خردمند را او بدان خسته دید
سکندر نگه کرد پس پهلوان
به بدخواه شد شاد و روشن روان
سپاه اندر آمد به پیش سوار
خردمند و شایسته‌ی کارزار
بفرمود تا بنده آگاه دید
چنین تا بر شاه ایران کشید
نهادند چیزی که پوشیده بود
جهان را درم داد و دینار بود
سران افسر از گوهر شاهوار
نخست آفرین کرد بر کردگار
چو بهرام بشنید گریان شدند



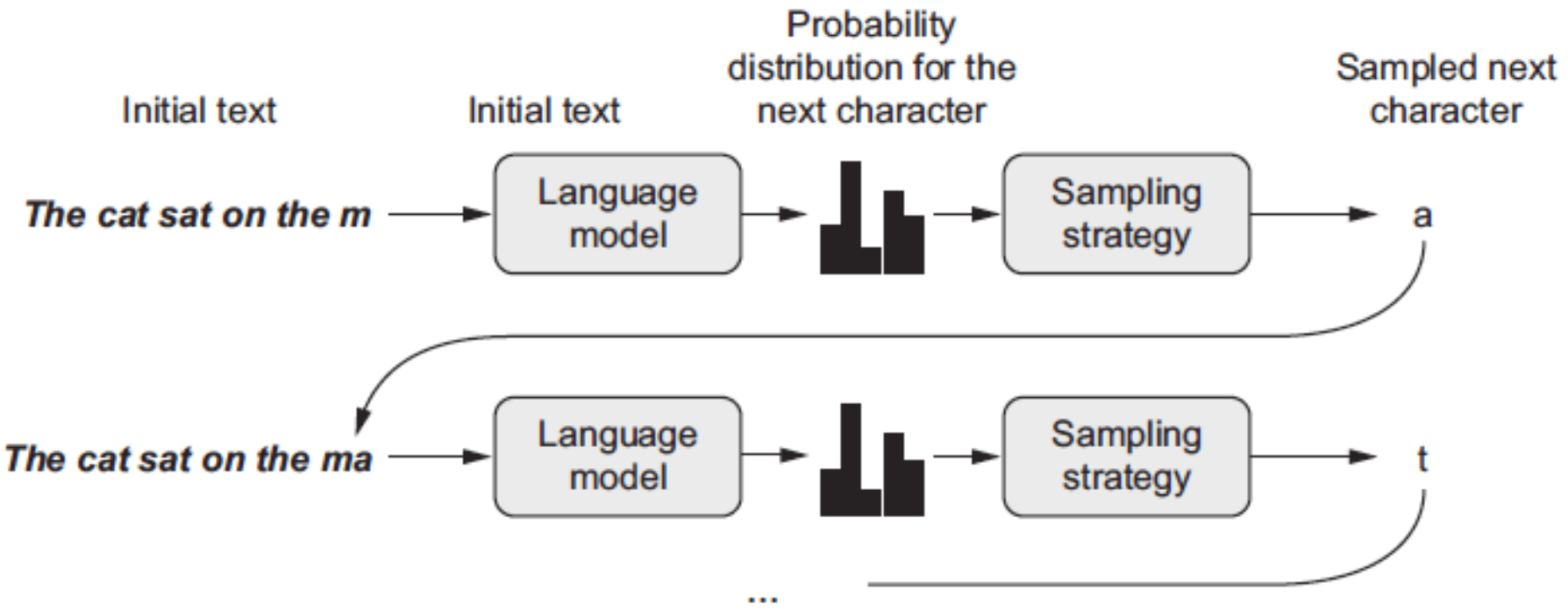
Text generation with LSTM

The universal way to generate sequence data in deep learning:

- Train a network to predict the next token or next few tokens in a sequence, using the previous tokens as input.
  - Called language model

- You can sample from it (generate new sequences)
  - Feed it an initial string of text (called *conditioning data*), ask it to generate the next character or the next word, add the generated output back to the input data, and repeat the process many times.

Text generation with LSTM



Text generation with LSTM

Sampling strategy:

- Greedy sampling
  - Always choosing the most likely next character.
  - Results in repetitive, predictable strings that don't look like coherent language.
- Stochastic sampling
  - Choose a word according to its softmax probability

Text generation with LSTM

How to control the amount of randomness? *Softmax temperature*

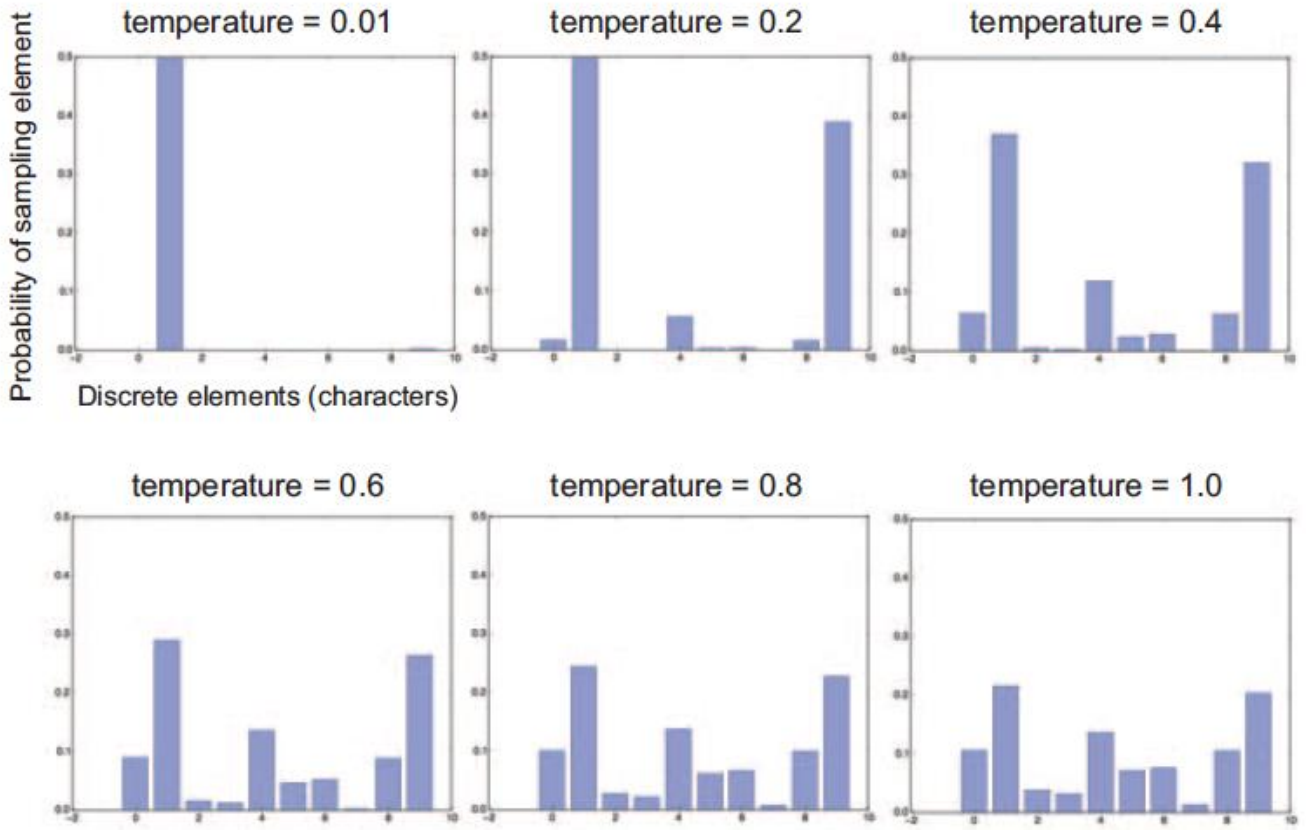
```
import numpy as np

def reweight_distribution(original_distribution, temperature=0.5):
    distribution = np.log(original_distribution) / temperature
    distribution = np.exp(distribution)
    return distribution / np.sum(distribution)
```

original_distribution is a 1D Numpy array of probability values that must sum to 1. temperature is a factor quantifying the entropy of the output distribution.

Returns a reweighted version of the original distribution. The sum of the distribution may no longer be 1, so you divide it by its sum to obtain the new distribution.

Text generation with LSTM

How to control the amount of randomness? *Softmax temperature*



Text generation with LSTM

Text generation with LSTM

Random seed text “new faculty, and the jubilation reached its climax when kant.”

Temp = 0.2, epoch 20

```
new faculty, and the jubilation reached its climax when kant and such a
man
in the same time the spirit of the surely and the such the such
as a man is the sunligh and subject the present to the superiority of the
special pain the most man and strange the subjection of the
special conscience the special and nature and such men the subjection of
the
special men, the most surely the subjection of the special
intellect of the subjection of the same things and
```

Text generation with LSTM

Temp = 0.5

```
new faculty, and the jubilation reached its climax when kant in the
eterned
and such man as it's also become himself the condition of the
experience of off the basis the superiory and the special morty of the
strength, in the langus, as which the same time life and "even who
discless the mankind, with a subject and fact all you have to be the stand
and lave no comes a troveration of the man and surely the
conscience the superiority, and when one must be w
```

Text generation with LSTM

Temp = 1.0

```
new faculty, and the jubilation reached its climax when kant, as a
periliting of manner to all definites and transpects it it so
hicable and ont him artiar resull
too such as if ever the proping to makes as cnecience. to been juden,
all every could coldiciousnike hother aw passife, the plies like
which might thiod was account, indifferent germin, that everythery
certain destrution, intellect into the deteriorablen origin of moralian,
and a lessority o
```

Text generation with LSTM

Temp = 0.2, epoch 60

```
cheerfulness, friendliness and kindness of a heart are the sense of the
spirit is a man with the sense of the sense of the world of the
self-end and self-concerning the subjection of the strengthorixes--the
subjection of the subjection of the subjection of the
self-concerning the feelings in the superiority in the subjection of the
subjection of the spirit isn't to be a man of the sense of the
subjection and said to the strength of the sense of the
```

Text generation with LSTM

Temp = 0.5

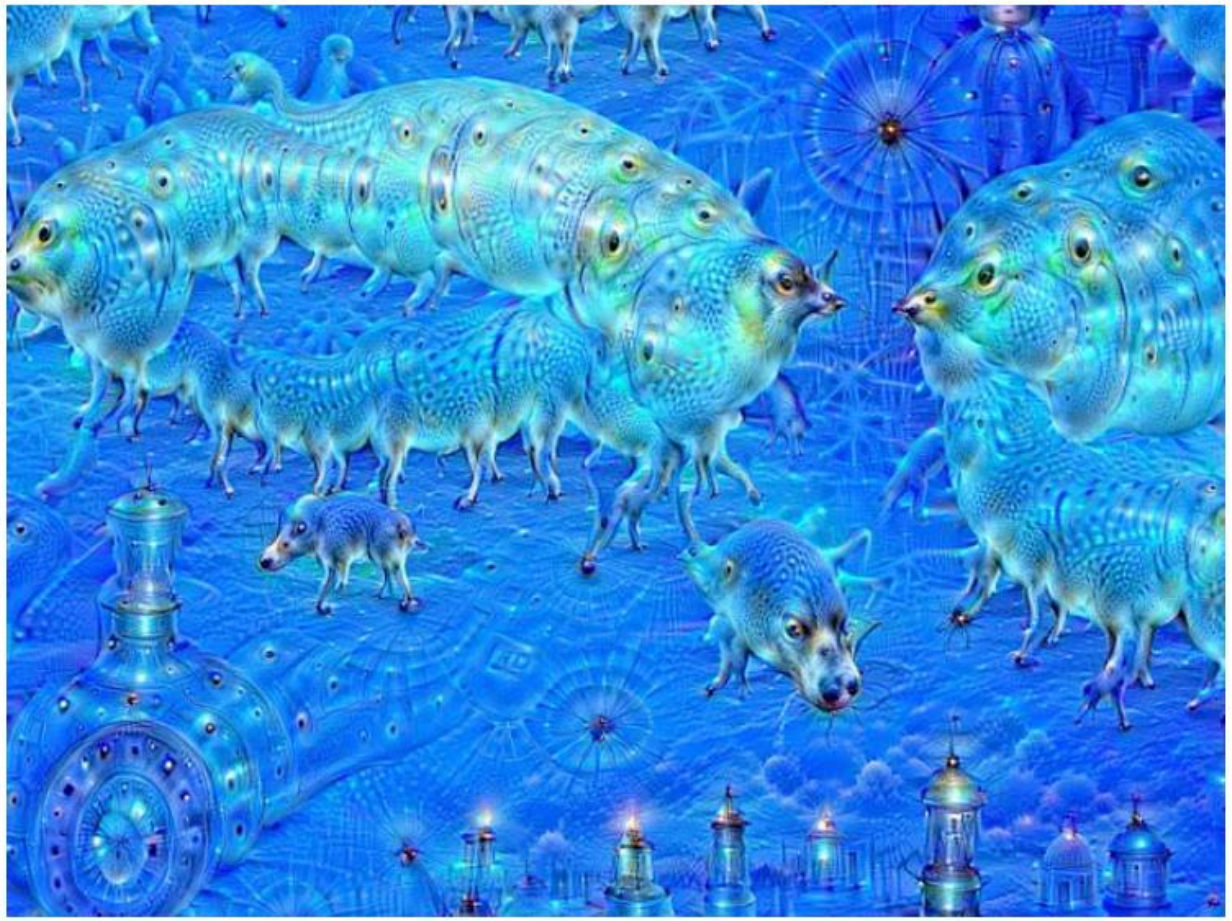
```
cheerfulness, friendliness and kindness of a heart are the part of the
soul
who have been the art of the philosophers, and which the one
won't say, which is it the higher the and with religion of the frences.
the life of the spirit among the most continuess of the
strengther of the sense the conscience of men of precisely before enough
presumption, and can mankind, and something the conceptions, the
subjection of the sense and suffering and the
```

Text generation with LSTM

Temp = 1.0

```
cheerfulness, friendliness and kindness of a heart are spiritual by the
ciuture for the
entalled is, he astraged, or errors to our you idstood--and it needs,
to think by spars to whole the amvives of the newoatly, prefectly
raals! it was
name, for example but voludd atu-especity"--or rank onee, or even all
"solett increessic of the world and
implussional tragedy experience, transf, or insiderar,--must hast
if desires of the strubction is be stronges
```

DeepDream



DeepDream

Very similar to ConvNet filter visualization, with a few simple differences:

- With DeepDream, you try to maximize the activation of entire layers rather than that of a specific filter, thus mixing together visualizations of large numbers of features at once.

- You start not from blank, slightly noisy input, but rather from an existing image.

- The input images are processed at different scales (called octaves), which improves the quality of the visualizations.

DeepDream

DeepDream

Neural style transfer

Neural style transfer

Neural style transfer

Style of a *reference* image is applied to a *target* image, while preserving the content of the target

Neural style transfer

Style of a *reference* image is applied to a *target* image, while preserving the content of the target

Achieved by defining a specific loss function:

- content loss + style loss

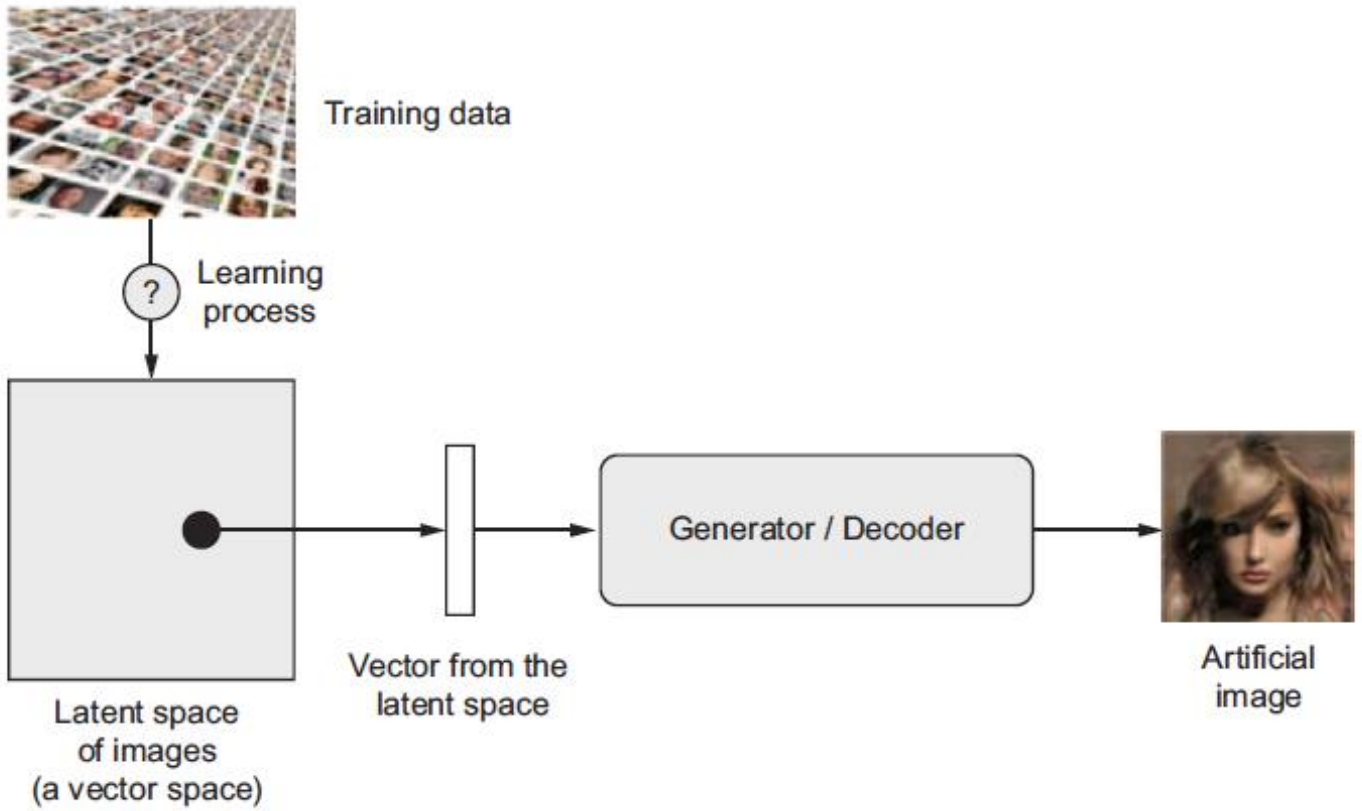
Neural style transfer

Upper layers in a convnet contain global information

- *Content* loss: L2 norm between the activations of an upper layer in a pretrained convnet, computed over the target image, and the activations of the same layer computed over the generated image.

- Style loss: Preserves style by maintaining similar correlations within activations for both low-level layers and high-level layers.
  - Feature correlations capture textures: the generated image and the style-reference image should share the same textures at different spatial scales.

Variational autoencoders



Autoencoders

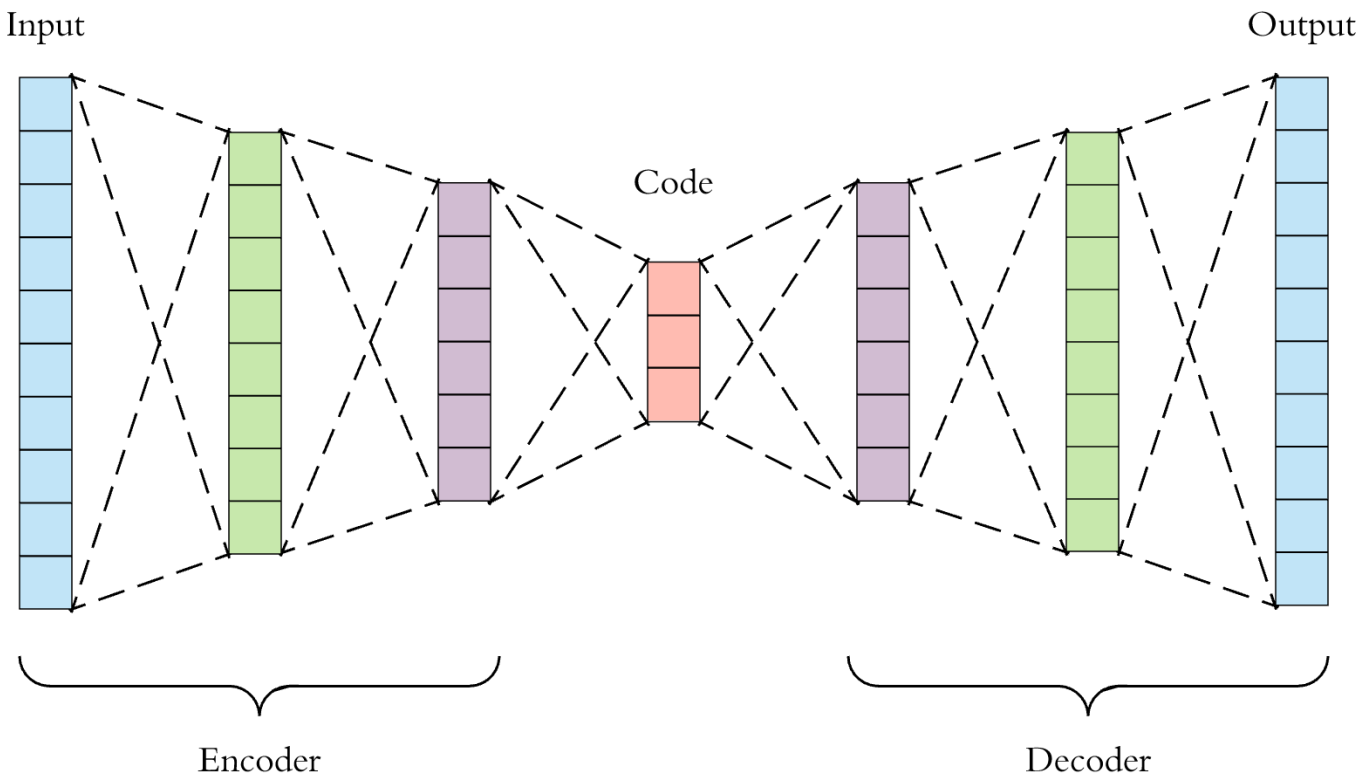
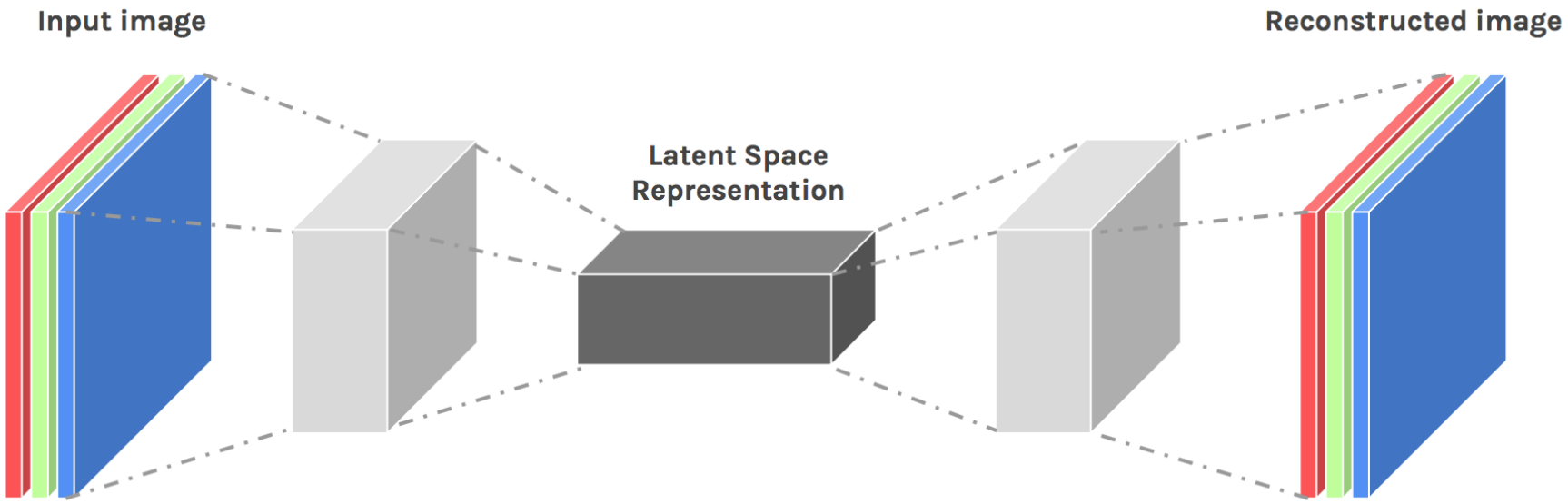
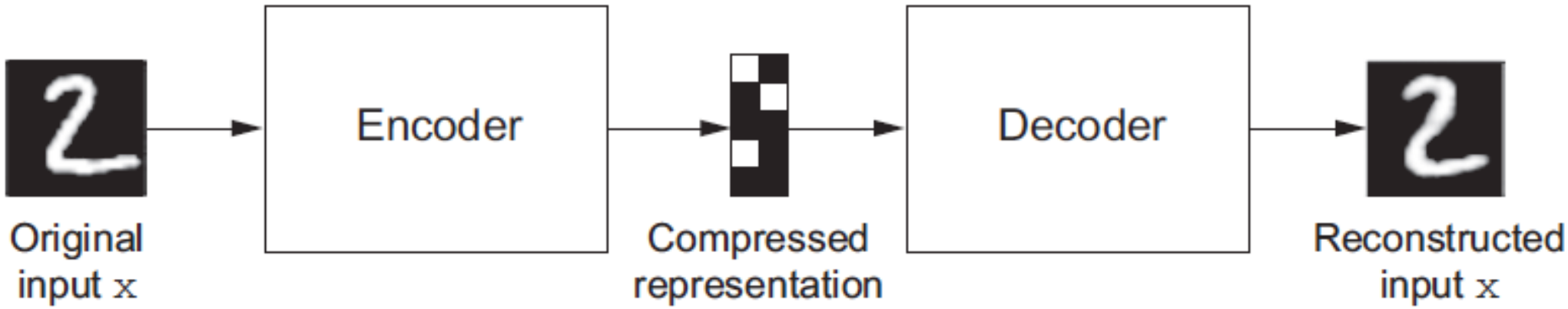


Image by Arden Dertat

Autoencoders



Autoencoders



Autoencoders

Training an autoencoder on the
MNIST dataset, and visualizing
the encodings from a 2D latent space

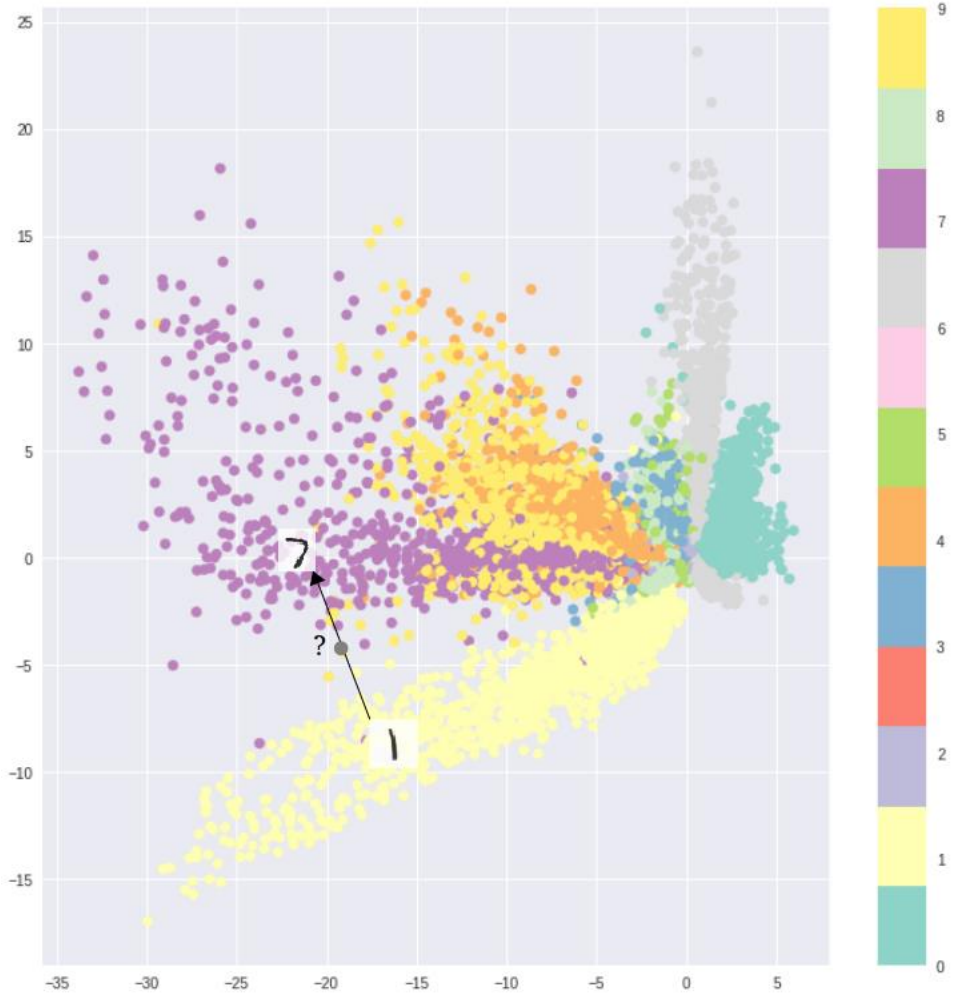
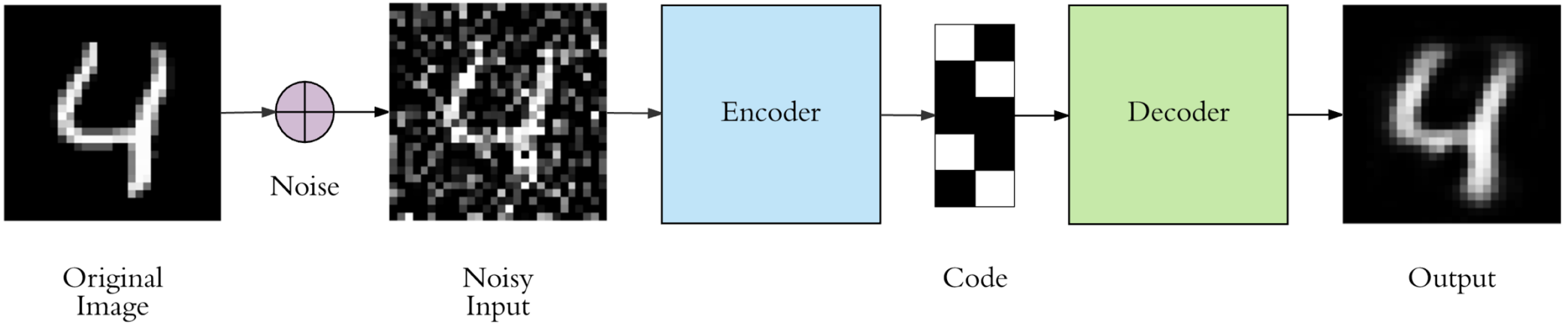


Figure by Irhum Shafkat

Variational autoencoders

Denoising autoencoder



Variational autoencoders

Advantage over autoencoders (that makes them suitable for generative models)

- Latent space is continuous

This is achieved by making the encoder not output an encoding vector of size *n*, rather, outputting two vectors of size n:

- a vector of means **μ**
- a vector of standard deviations **σ**

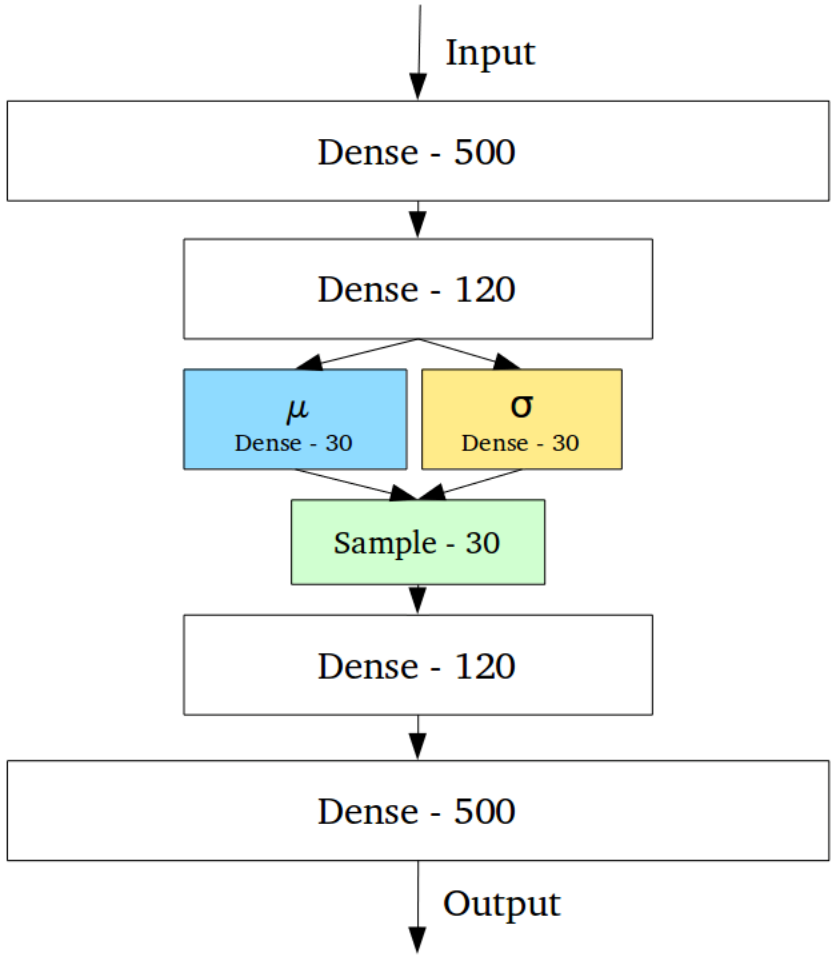


Figure by Irhum Shafkat

Concept space

Semantic space of embeddings is a specific case

Given a latent space of representations, certain directions may encode interesting axes of variation in the original data.

- If **z** is the representation for a face, there might be a smile vector **s**, then **z + s** is the same face with smile
- Same for turning a male face to female
- Or adding sunglasses to a face

Concept space

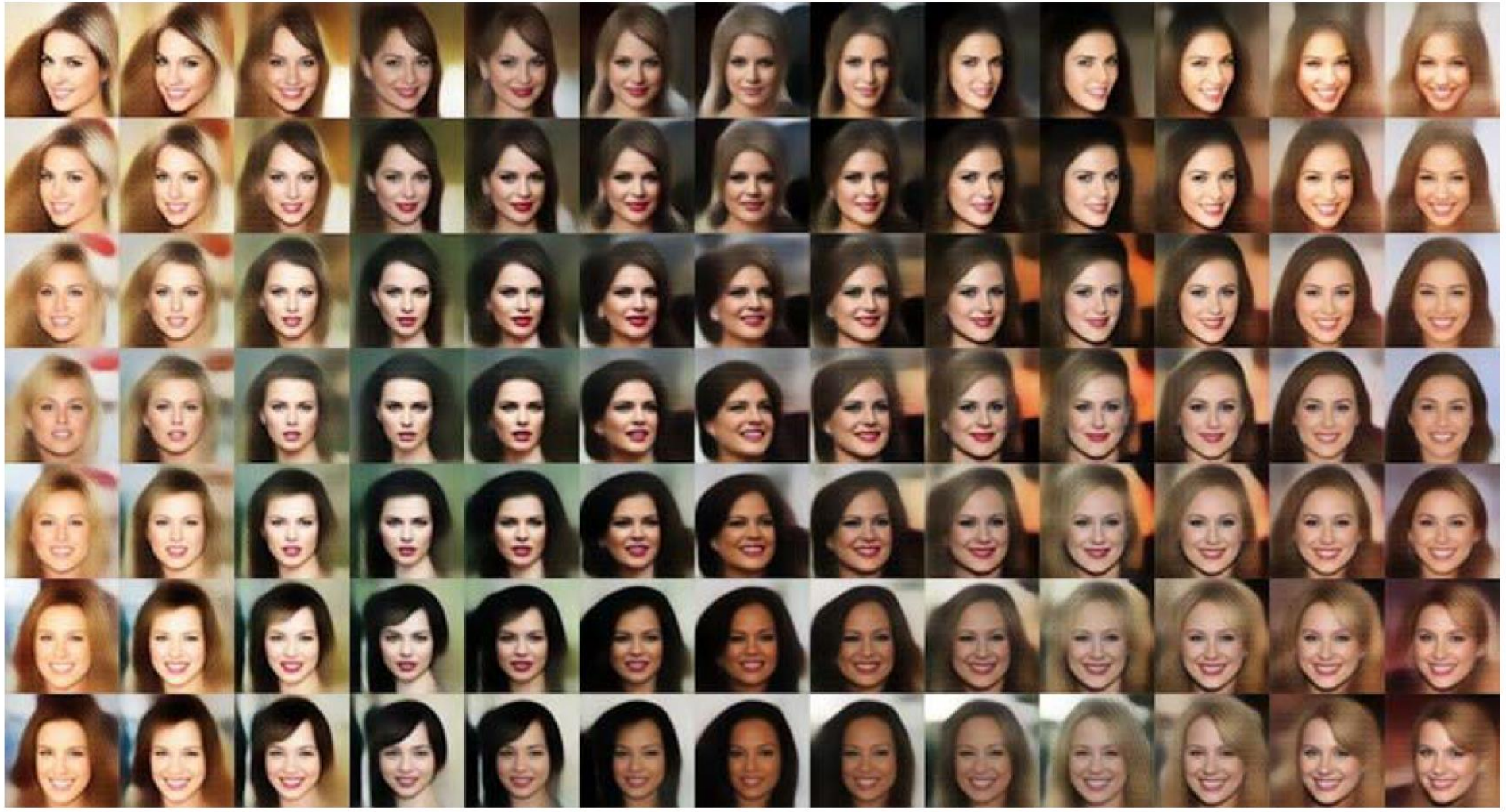


Figure 8.10 A continuous space of faces generated by Tom White using VAEs

Concept space

Smile vector

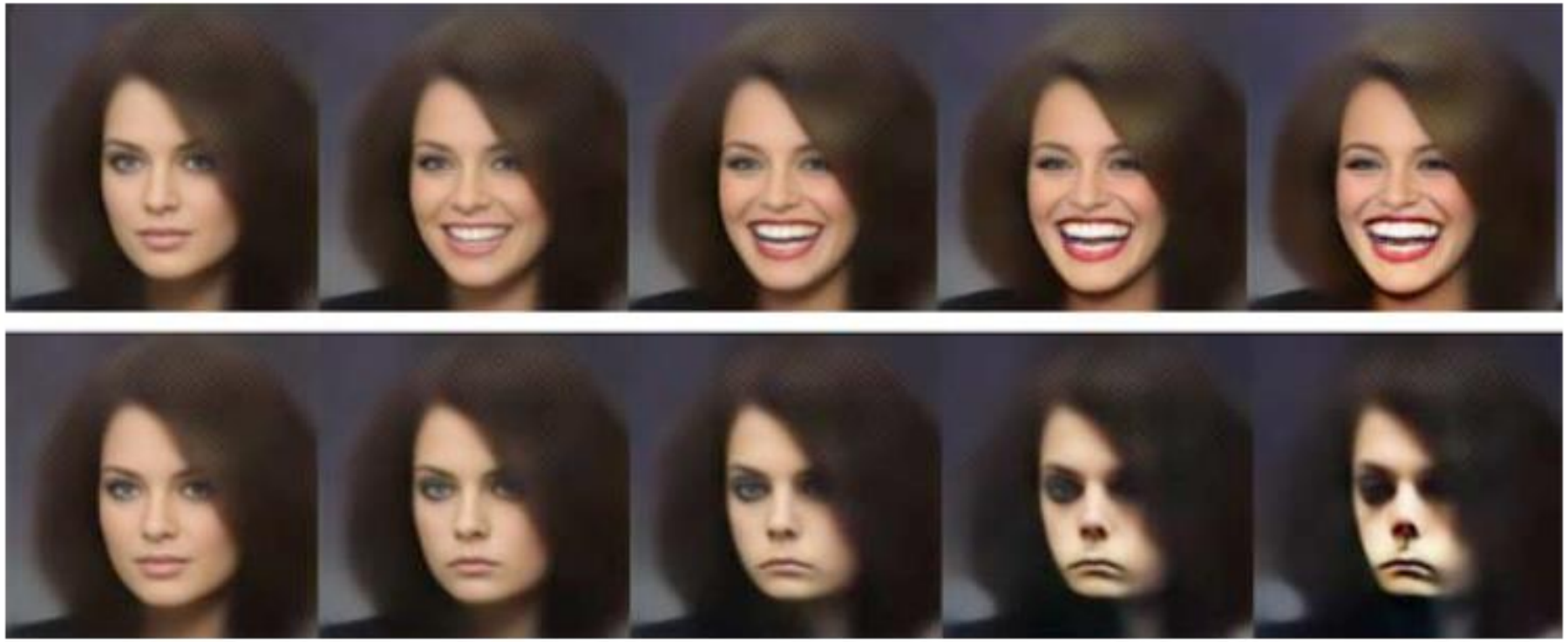
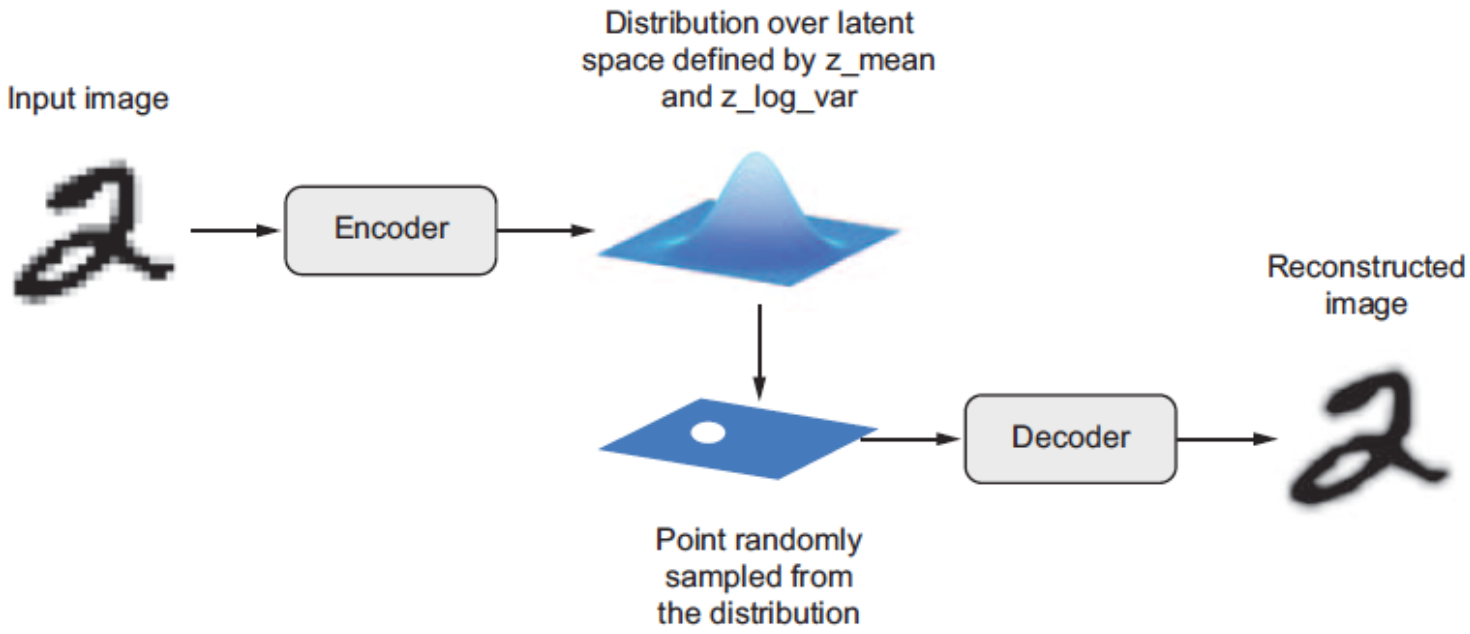


Image from Tom White (Victoria University School of Design in New Zealand)

Variational autoencoders



Variational autoencoders

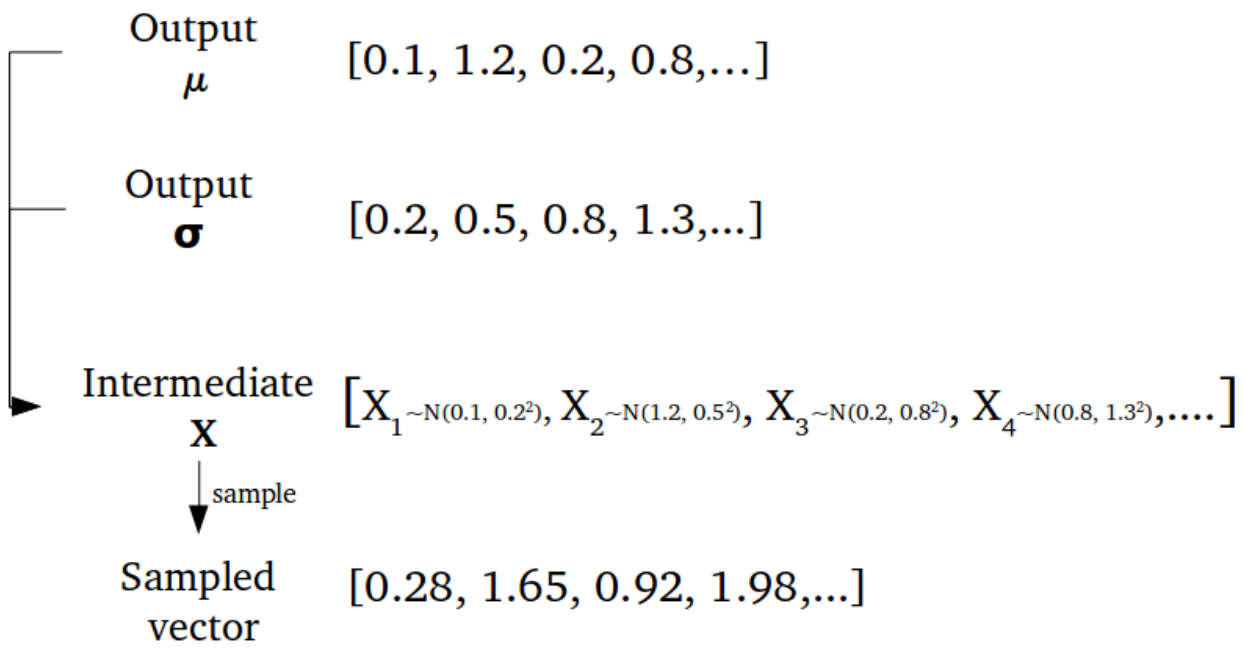


Illustration by Irhum Shafkat

Variational autoencoders

- An encoder module turns the input samples **`input_img`** into two parameters in a latent space of representations, **`z_mean`** and **`z_log_variance`**.

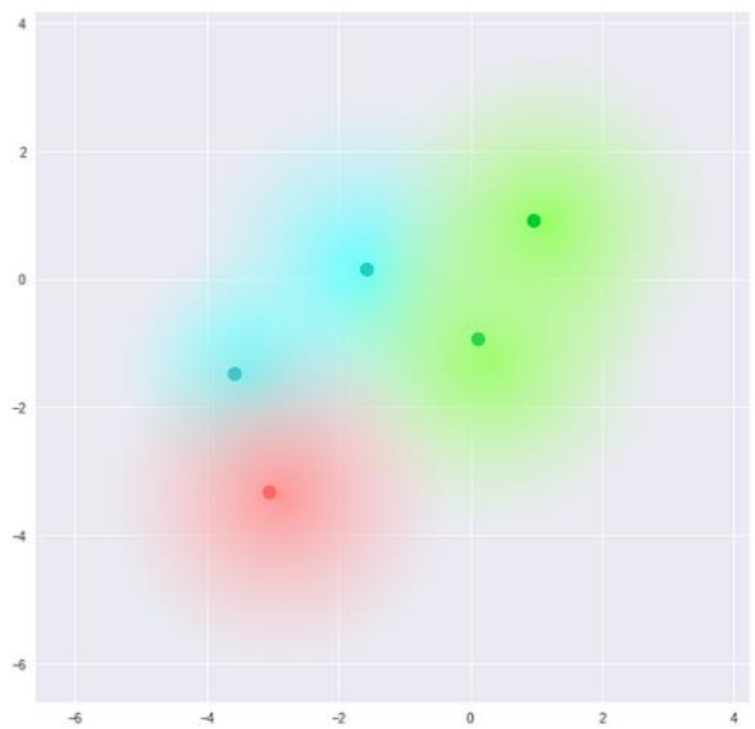
- You randomly sample a point **`z`** from the latent normal distribution that's assumed to generate the input image, via

  `z = z_mean + exp(z_log_variance) * epsilon`

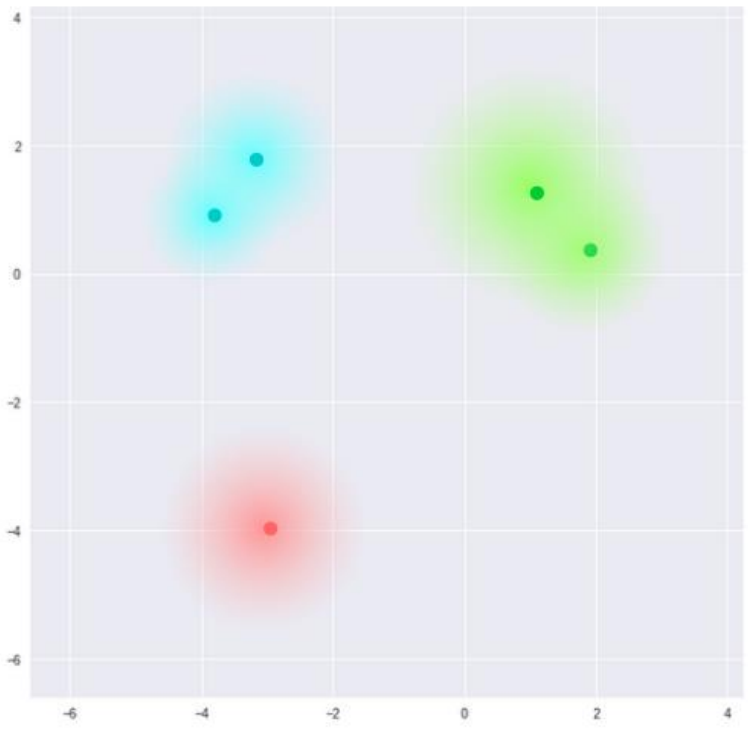
  where epsilon is a random tensor of small values.

- A decoder module maps this point in the latent space back to the original input image.

Variational autoencoders



What we require



What we may inadvertently end up with

Illustration by Irhum Shafkat

$$l_i(\theta, \phi) = -\mathbb{E}_{z \sim q_\theta(z|x_i)}[\log p_\phi(x_i \mid z)] + \mathbb{KL}(q_\theta(z \mid x_i) \,||\, p(z))$$

Variational autoencoders

MNIST visualisation

with KL-divergence loss only

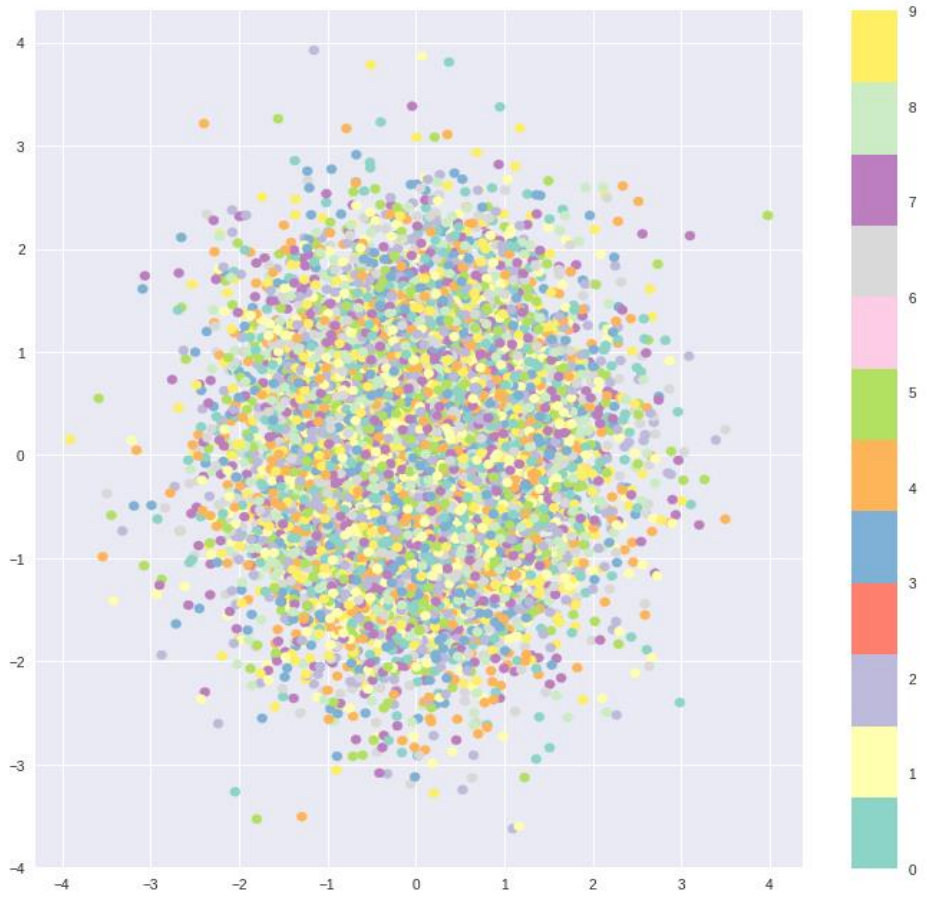


Illustration by Irhum Shafkat

Variational autoencoders

MNIST visualisation

with KL-divergence and classification loss

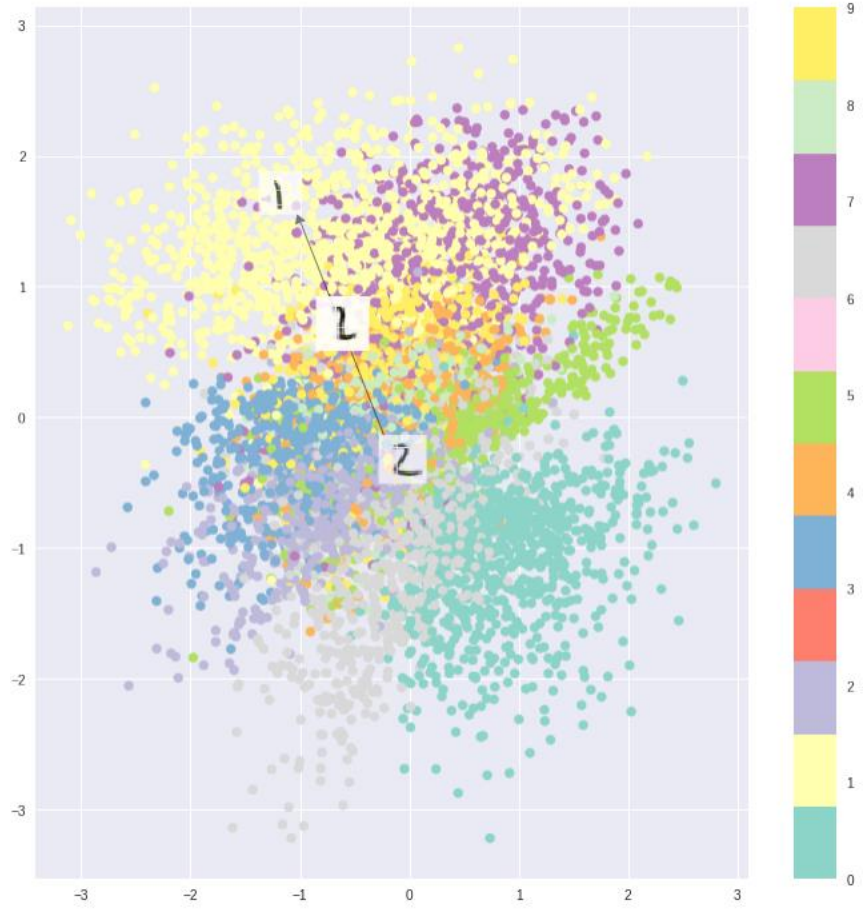


Illustration by Irhum Shafkat

VAE in Keras (encoder)

```
img_shape = (28, 28, 1)
batch_size = 16
latent_dim = 2

input_img = keras.Input(shape=img_shape)

x = layers.Conv2D(32, 3,
                  padding='same', activation='relu')(input_img)
x = layers.Conv2D(64, 3,
                  padding='same', activation='relu',
                  strides=(2, 2))(x)
x = layers.Conv2D(64, 3,
                  padding='same', activation='relu')(x)
x = layers.Conv2D(64, 3,
                  padding='same', activation='relu')(x)
shape_before_flattening = K.int_shape(x)

x = layers.Flatten()(x)
x = layers.Dense(32, activation='relu')(x)

z_mean = layers.Dense(latent_dim)(x)
z_log_var = layers.Dense(latent_dim)(x)
```

Dimensionality of the latent space: a 2D plane

The input image ends up being encoded into these two parameters.

VAE in Keras (sampling)

```
def sampling(args):
    z_mean, z_log_var = args
    epsilon = K.random_normal(shape=(K.shape(z_mean)[0], latent_dim),
                              mean=0., stddev=1.)
    return z_mean + K.exp(z_log_var) * epsilon

z = layers.Lambda(sampling)([z_mean, z_log_var])
```

VAE in Keras (decoder)

```
decoder_input = layers.Input(K.int_shape(z)[1:])

x = layers.Dense(np.prod(shape_before_flattening[1:]),
                 activation='relu')(decoder_input)

x = layers.Reshape(shape_before_flattening[1:])(x)

x = layers.Conv2DTranspose(32, 3,
                           padding='same',
                           activation='relu',
                           strides=(2, 2))(x)
x = layers.Conv2D(1, 3,
                  padding='same',
                  activation='sigmoid')(x)

decoder = Model(decoder_input, x)

z_decoded = decoder(z)
```

Input where you'll feed z

Upsamples the input

Uses a Conv2DTranspose layer and Conv2D layer to decode z into a feature map the same size as the original image input

Instantiates the decoder model, which turns "decoder_input" into the decoded image

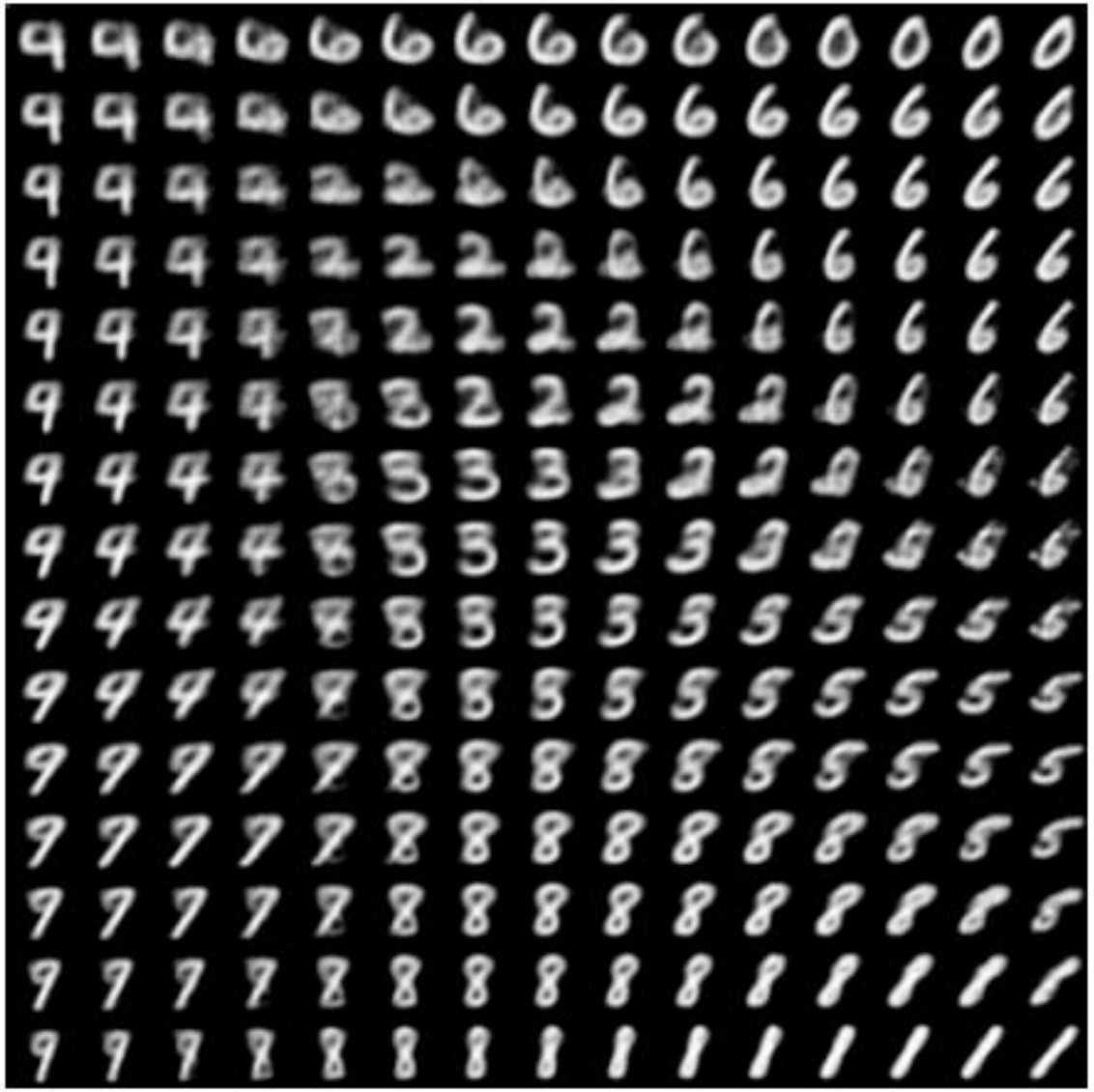
Applies it to z to recover the decoded z

VAE space for MNIST

VAE space for MNIST

The grid of sampled digits

- Specific directions in this space have a meaning: for example, there's a direction for "four-ness," "one-ness," and so on.



Questions?