BIO-INSPIRED COMPUTING: APPLICATIONS AND INTERFACES

GENERATIVE MODELS

Slides created by Jo Plested

LECTURE OUTLINE

- Generative modelling
  - Variational Autoencoders
  - Generative Adversarial Networks

GENERATIVE MODELLING

- In generative modelling we get training examples X distributed according to some unknown distribution $P_{gt}(X)$, and our goal is to learn a model P which we can sample from, such that P is as similar as possible to P_{gt}
- For example: we might have a set of images X and we want to generate more images that look similar but not the same as those in X

GENERATIVE MODELLING

Unique faces created using generative modelling (top line) and their nearest neighbour in the training dataset (bottom line)

http://research.nvidia.com/publication/2017-10_Progressive-Growing-of

GENERATIVE MODELLING

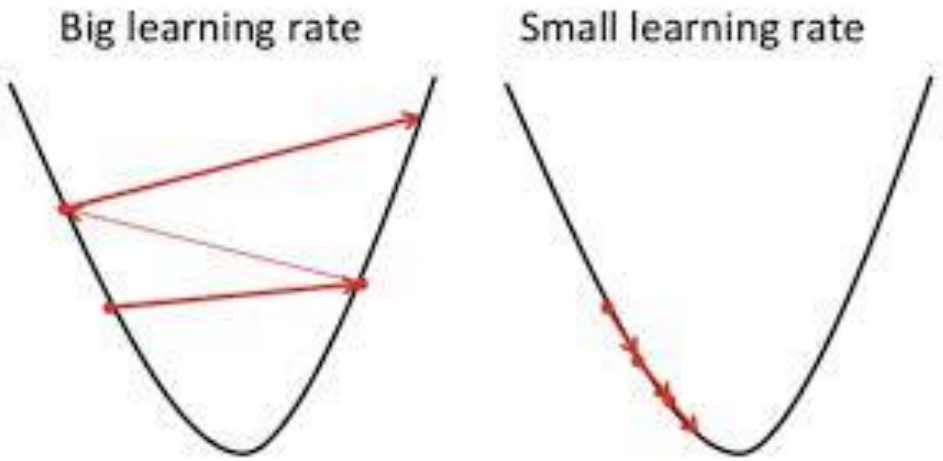
Current state of the art for face generation

https://arxiv.org/abs/1812.04948

DIFFERENTIABLE GENERATIVE MODELLING

- Training generative models has been a long standing problem in the machine learning community
- Traditional techniques have made strong assumptions or approximations leading to suboptimal models or relied on computationally expensive inference procedures like Markov Chain Monte Carlo
- With the rise of neural networks focus has turned to techniques that are able to be trained via backpropagation

DIFFERENTIABLE GENERATIVE MODELLING



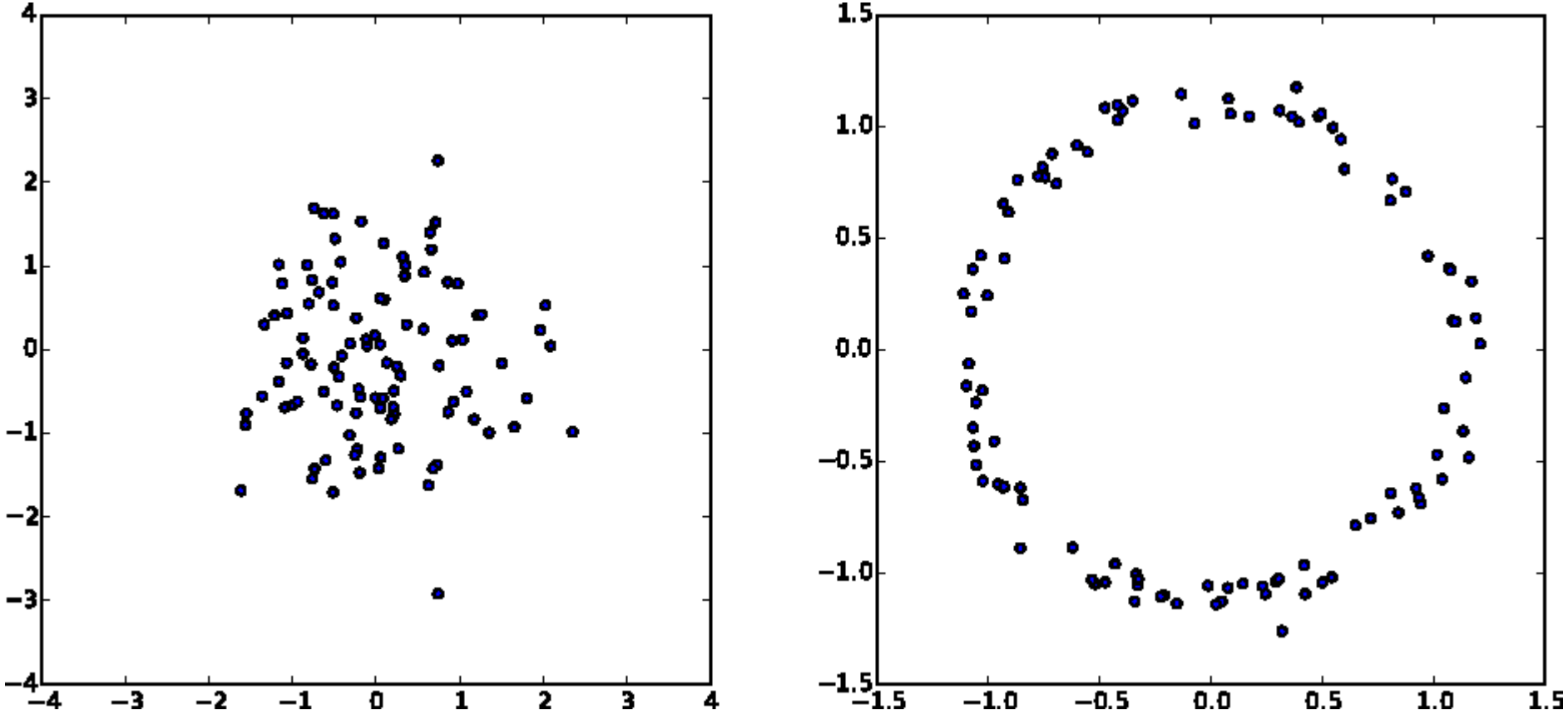
Recap

- To train models with stochastic gradient descent they must be differentiable
- That way we can calculate the gradients of all the weights we are trying to train and work out which direction to move them to reduce the loss function ‘down hill’

DIFFERENTIABLE GENERATIVE MODELLING

- This means that stochastic gradient descent can deal with stochastic inputs, but not stochastic units within the neural network itself
- The solution - move the sampling to the input layer
- For example: sampling from $\mathcal{N}(\mu, \sigma^2)$ can be done via a simple generator model with one layer that takes as input a sample x from $\mathcal{N}(0,1)$ and transforms it using $\mu + \sigma x$

DIFFERENTIABLE GENERATIVE MODELLING



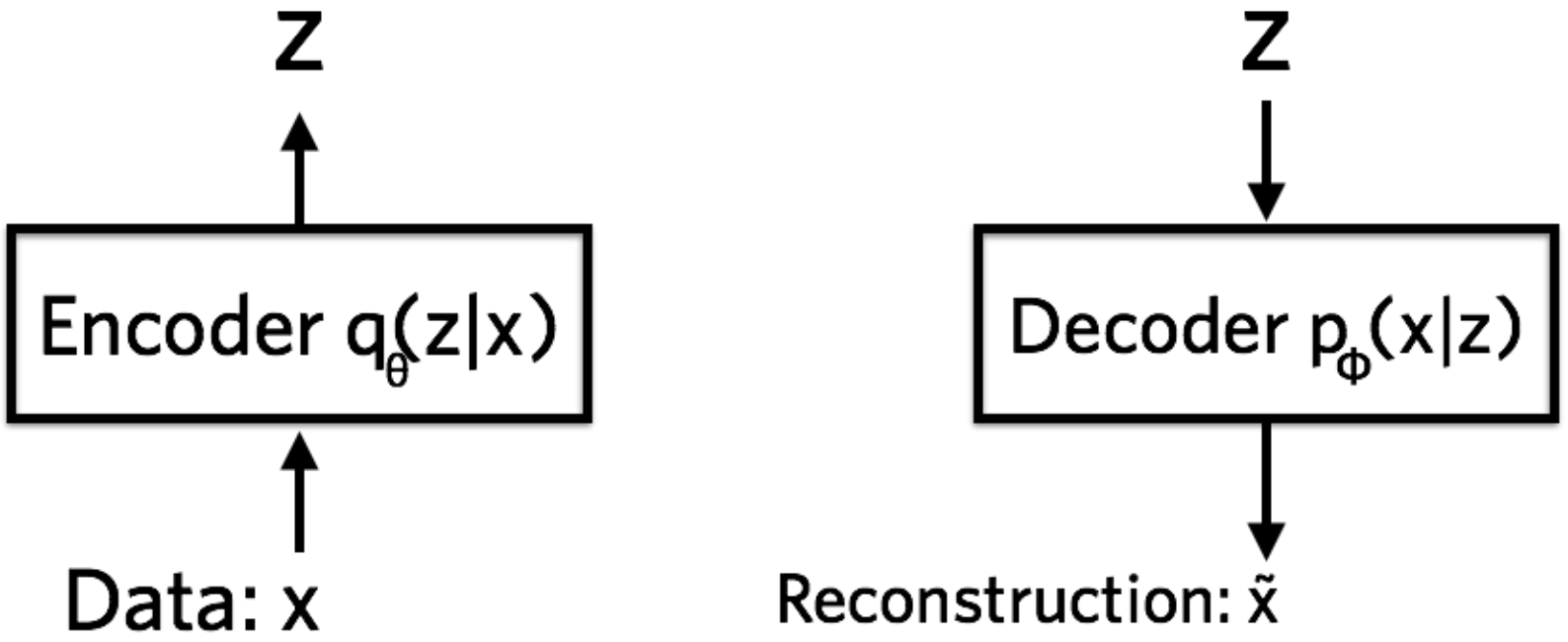
Left: samples from a Normal distribution. Right: those same samples mapped through the function g(z) = z/10 + z/||z|| to form a ring

VARIATIONAL AUTOENCODERS

- When training a generative model the more complicated the dependencies between the dimensions the more difficult the model is to train
- Take for example trying to create unique handwritten digits. Decisions must be made on which digit is being created, which way it's sloped, how thick it is, etc
  - before the actual pixels can be generated
- These are called latent variables generally denoted by z

VARIATIONAL AUTOENCODERS

- A Variational Autoencoder (VAE) can be thought of as an encoder that encodes the data into latent variables z and a decoder that reconstructs the data given z



https://jaan.io/what-is-variational-autoencoder-vae-tutorial/

VARIATIONAL AUTOENCODERS

- The encoder is an NN that takes values of x as inputs and outputs parameters to $q_\theta(z|x)$ which is generally a Normal probability density. This distribution can then be sampled to get values for the latent variable z
- The decoder is an NN that takes values of z and outputs the parameters of the probability distribution of the data $P_\Phi(x|z)$. For the example of generating a 28x28 black and white handwritten digit the decoder would generate 784 Bernoulli parameters (probability that x is 1 - white). One for each pixel in the image

VARIATIONAL AUTOENCODERS

- The loss function of the VAE is:

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

The first term is the reconstruction loss (binary cross-entropy for the case of handwritten digits) and encourages the decoder to learn to reconstruct the data well

VARIATIONAL AUTOENCODERS

- The loss function of the VAE is:

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

The second term is the Kullback-Leibler (KL) divergence (a measure of distance) between $q_\theta(z/x_i)$ *and* $p(z)$ which acts as a regularizer that keeps $q_\theta(z/x_i)$ close to $p(z)$ where $p(z)$ is generally $\mathcal{N}(0, I)$. This regularizer term helps keep the representations of z produced by the encoder meaningful and sufficiently diverse.

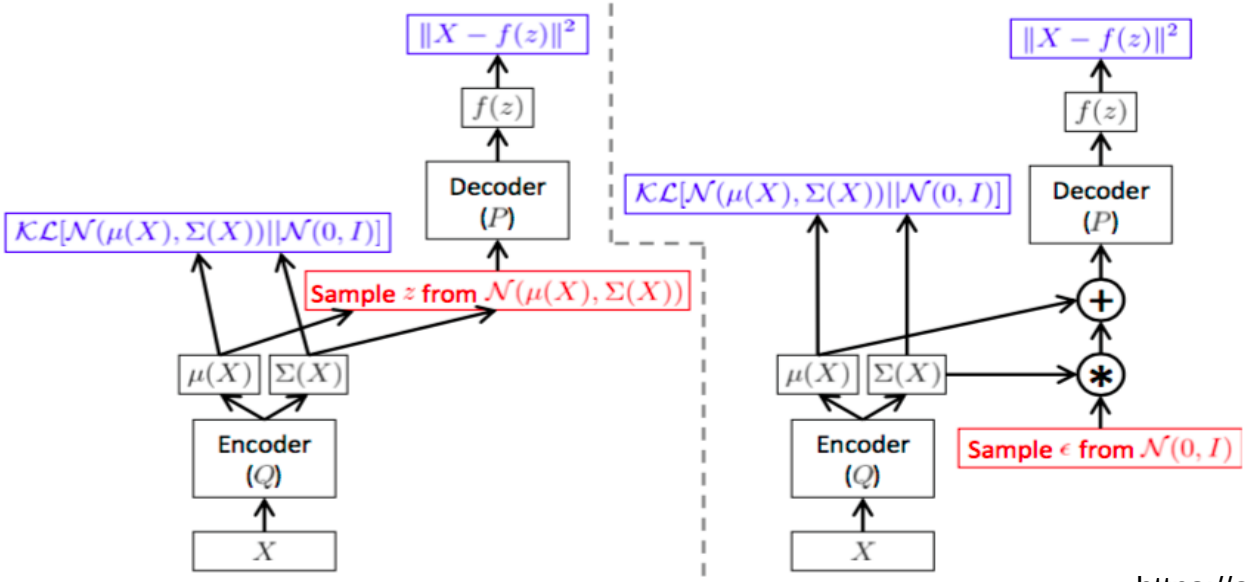
The KL divergence when $q_\theta(z/x_i)$ is $\mathcal{N}(\mu(x_i), \sigma(x_i))$ and $p(z)$ is $\mathcal{N}(0,1)$:

$\{(\mu_1 - \mu_2)^2 + \sigma_1^2 - \sigma_2^2\}/(2.\sigma_2^2) + \ln(\sigma_2/\sigma_1)$ which reduces to:

$\{\mu(x_i)^2 + \sigma(x_i)^2 - 1\}/2 + \ln(1/\sigma(x_i))$

VARIATIONAL AUTOENCODERS ARCHITECTURE

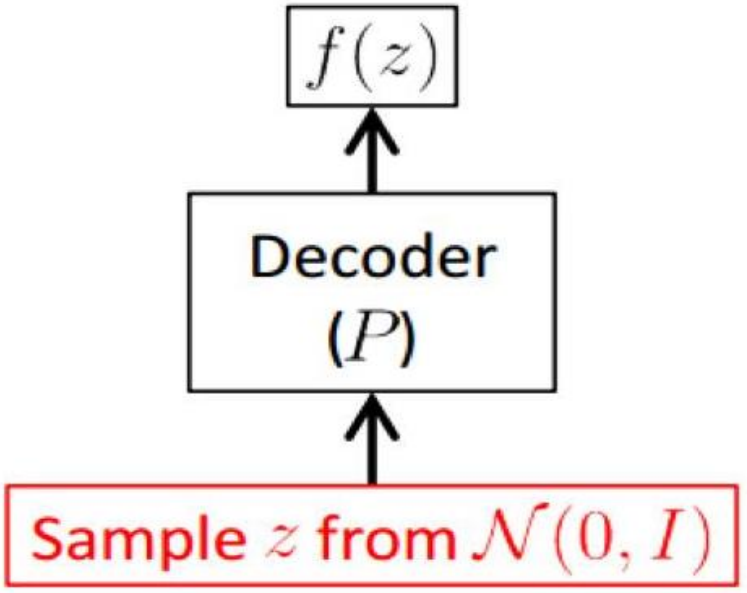
The left hand side shows a VAE model that can't be differentiated therefore can't be trained with stochastic gradient descent. The right hand side shows the same model with the sampling moved to the input layer. This can now be trained via SGD



https://arxiv.org/pdf/1606.05908v1.pdf

VARIATIONAL AUTOENCODERS

To generate a new sample from the model at test time the latent variable z is simply sampled from $\mathcal{N}(0, I)$ and run through the decoder NN to produce parameters of the probability distribution of the data from which samples can be generated



https://arxiv.org/pdf/1606.05908v1.pdf

VARIATIONAL AUTOENCODERS

VAE example. Producing MNIST digits in Pytorch: https://github.com/pytorch/examples/blob/master/vae/main.py

- In this example the encoder is a standard fully connected NN layer with a separate NN layer to produce each of the outputs - mean (mu) and standard deviation (actually logvar - log of the variance in the code) of a Guassian distribution

```
def encode(self, x):
    h1 = F.relu(self.fc1(x))
    return self.fc21(h1), self.fc22(h1)

def forward(self, x):
    mu, logvar = self.encode(x.view(-1, 784))
```

VARIATIONAL AUTOENCODERS

VAE example. Producing MNIST digits in Pytorch:
https://github.com/pytorch/examples/blob/master/vae/main.py

- The latent variable z is created by transforming samples from $\mathcal{N}(0,1)$ to create samples from $\mathcal{N}(\text{mu}, \exp(0.5 * \text{logvar}))$

```
def reparameterize(self, mu, logvar):
    std = torch.exp(0.5*logvar)
    eps = torch.randn_like(std)
    return mu + eps*std

def forward(self, x):
    mu, logvar = self.encode(x.view(-1, 784))
    z = self.reparameterize(mu, logvar)
```

VARIATIONAL AUTOENCODERS

VAE example. Producing MNIST digits in Pytorch:
https://github.com/pytorch/examples/blob/master/vae/main.py

- The output of the decoder is 784 (28x28) sigmoid values (probability of the pixel being coloured white in each location)

```
def decode(self, x):
    h3 = F.relu(self.fc3(x))
    return torch.sigmoid(self.fc4(h3))

def forward(self, x):
    mu, logvar = self.encode(x.view(-1, 784))
    z = self.reparameterize(mu, logvar)
    return self.decode(z), mu, logvar
```

VARIATIONAL AUTOENCODERS

VAE example. Producing MNIST digits in Pytorch:
https://github.com/pytorch/examples/blob/master/vae/main.py

The VAE loss is two losses summed together

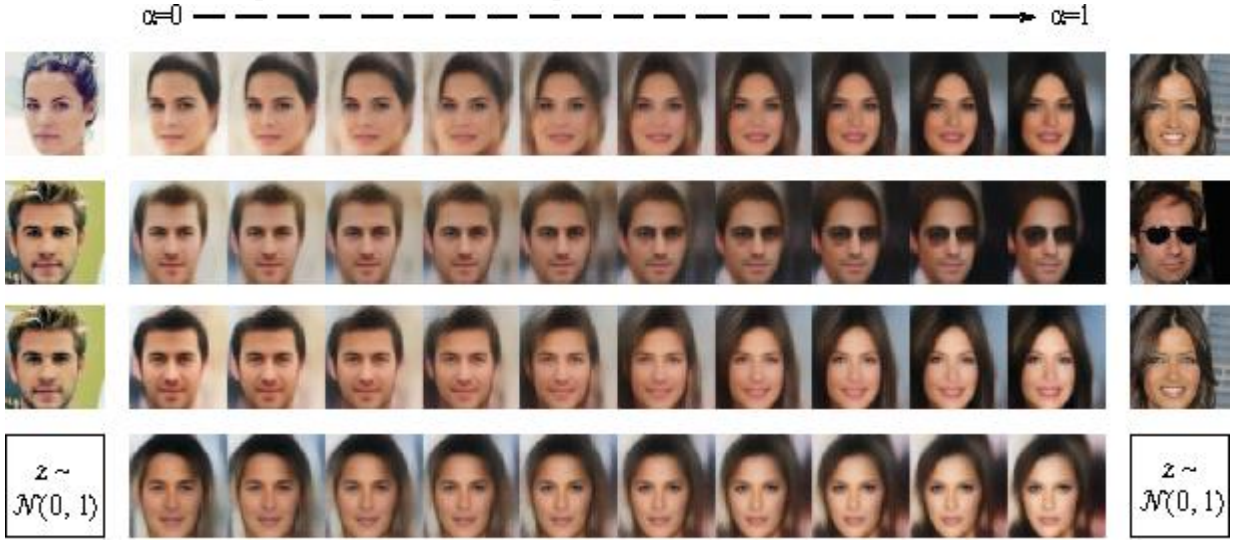
- The first loss is a binary cross-entropy loss. This is an individual cross-entropy loss for each pixel value (how far off the correct value black or white was each pixel)
- The second is the Kullback-Leibler divergence

$$\{\mu(x_i)^2 + \sigma(x_i)^2 - 1\}/2 + \ln(1/\sigma(x_i))$$

```
# Reconstruction + KL divergence losses summed over all elements and batch
def loss_function(recon_x, x, mu, logvar):
    BCE = F.binary_cross_entropy(recon_x, x.view(-1, 784), reduction='sum')
    KLD = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp())
    return BCE + KLD
```

VARIATIONAL AUTOENCODERS EXAMPLES

Latent space experiments with VAEs



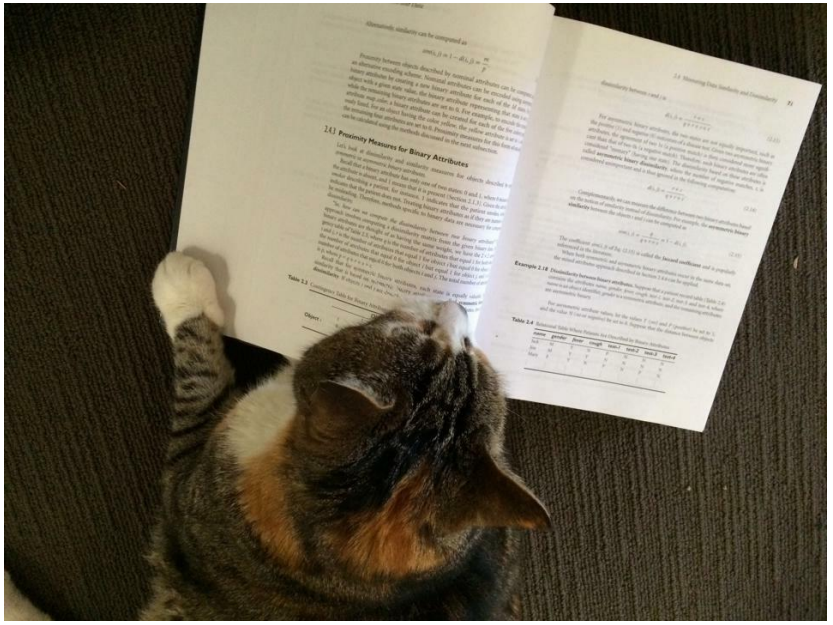
Linear interpolation for latent vector. Each row is the interpolation from left latent vector zleft to right latent vector zright. e.g. (1 − α)zleft + αzright. The first row is the transition from a non-smiling woman to a smiling woman, the second row is the transition from a man without eyeglass to a man with eyeglass, the third row is the transition from a man to a woman, and the last row is the transition between two fake faces decoded from z ~ N (0, 1)

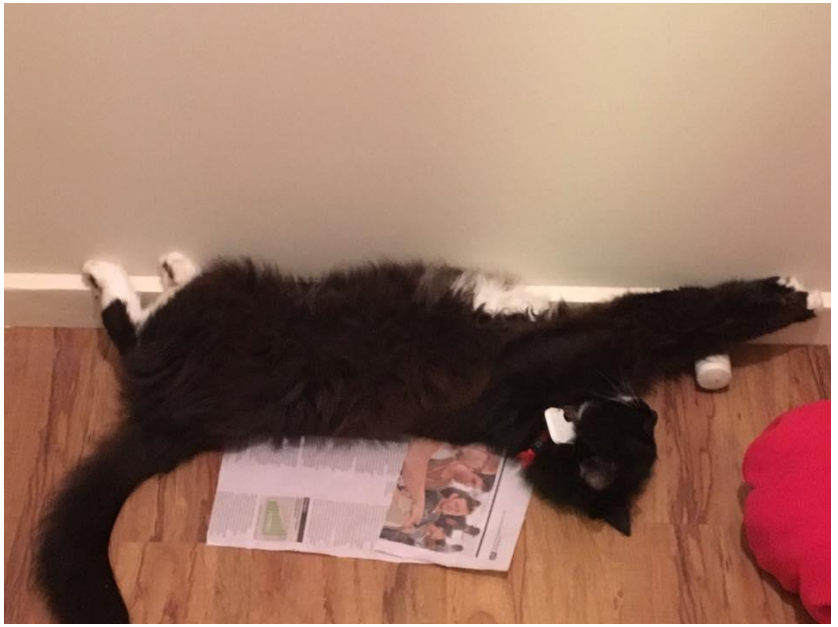
https://arxiv.org/pdf/1610.00291.pdf

GENERATIVE ADVERSARIAL NETWORKS

GENERATIVE ADVERSARIAL NETWORKS

- How do you design a cost function to teach a neural network to generate a semantic concept?

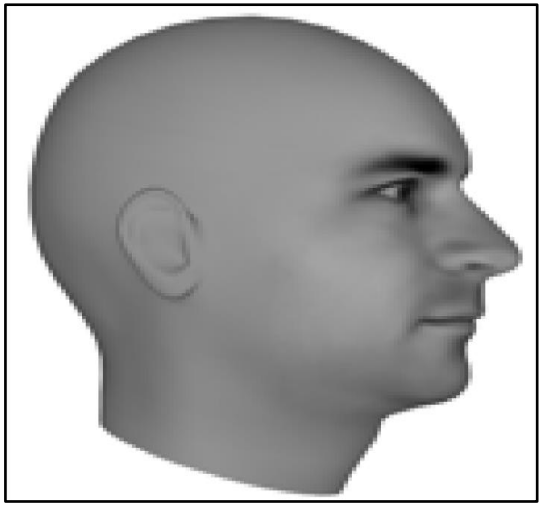




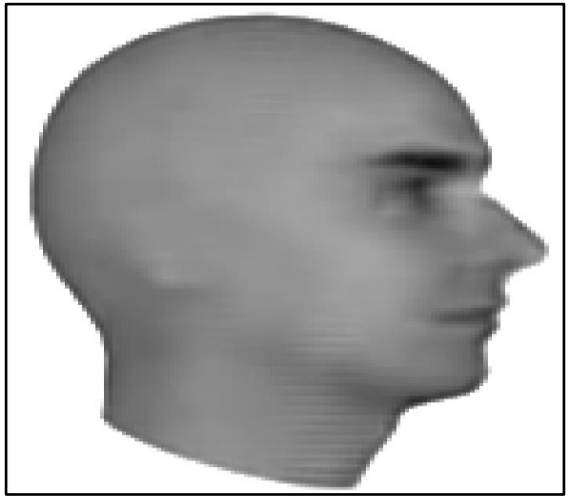
GENERATIVE ADVERSARIAL NETWORKS EXAMPLES

MSE between pixel values = blurs with uncertainty

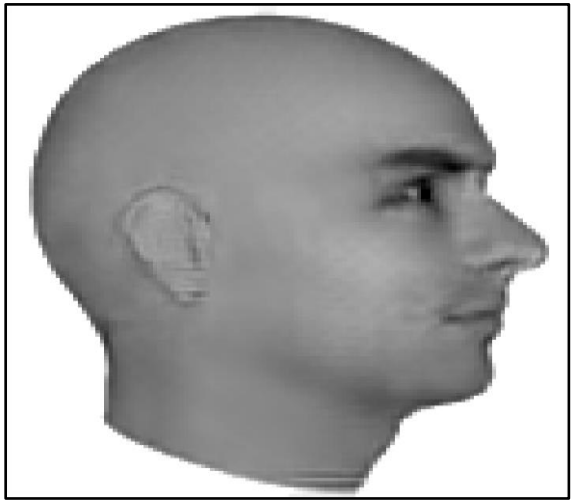
Ground Truth



MSE



GAN

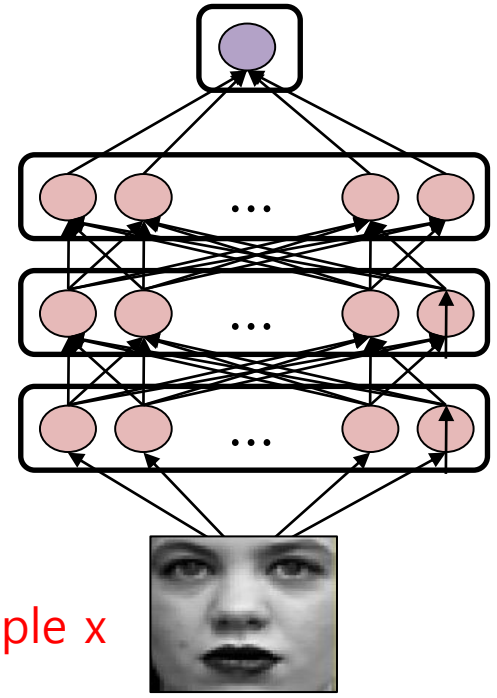


Next Video Frame Prediction

https://arxiv.org/pdf/1605.08104

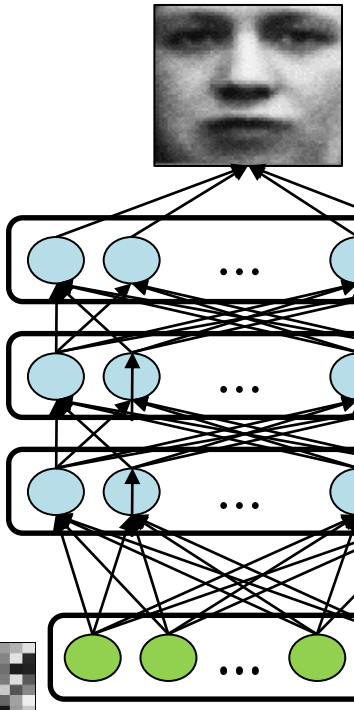
GENERATIVE ADVERSARIAL NETWORKS

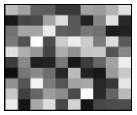
Discriminative Model ***D***



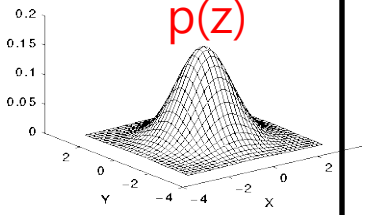
Sample x

Generative Model ***G***



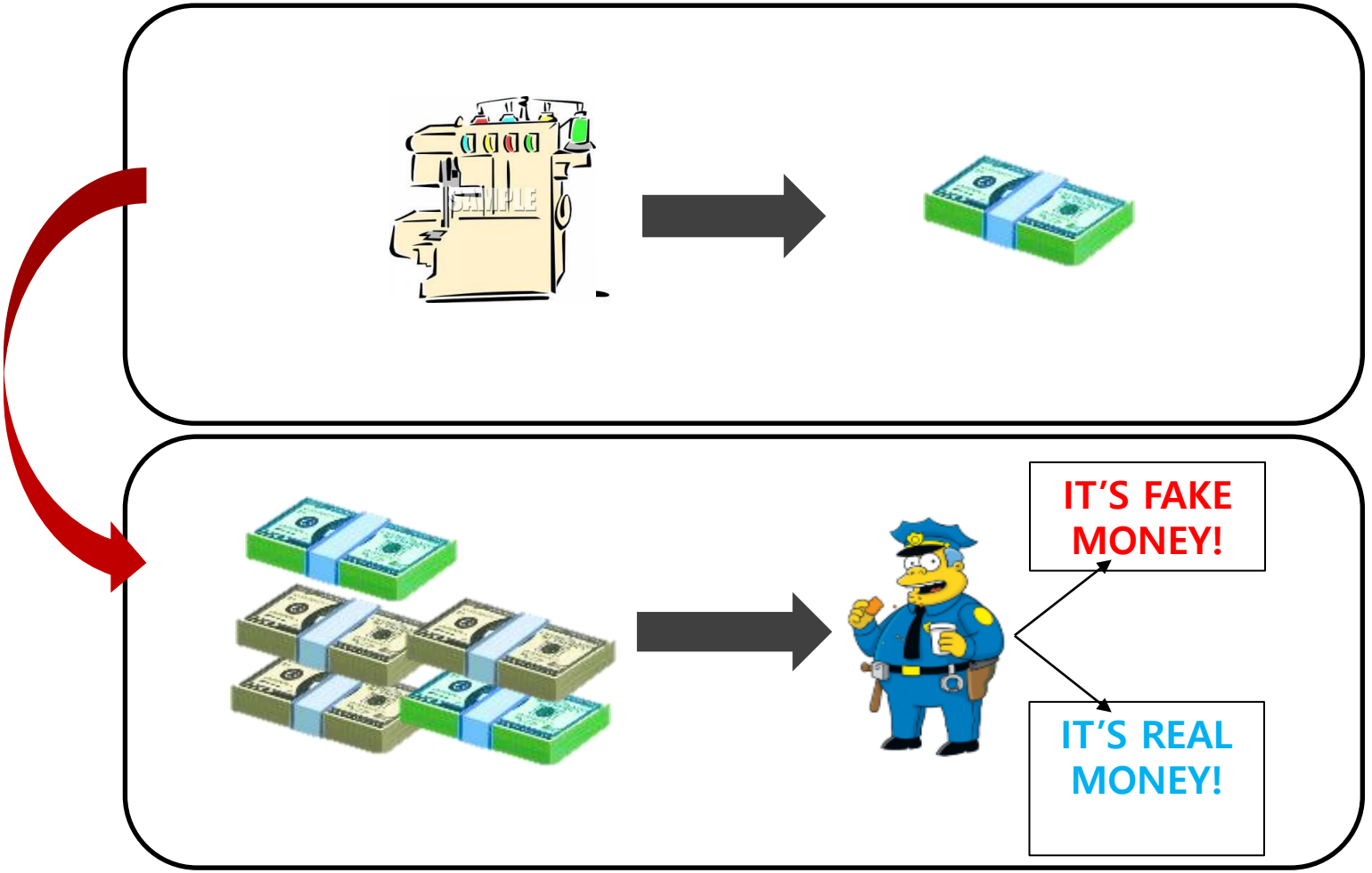


Random noise z



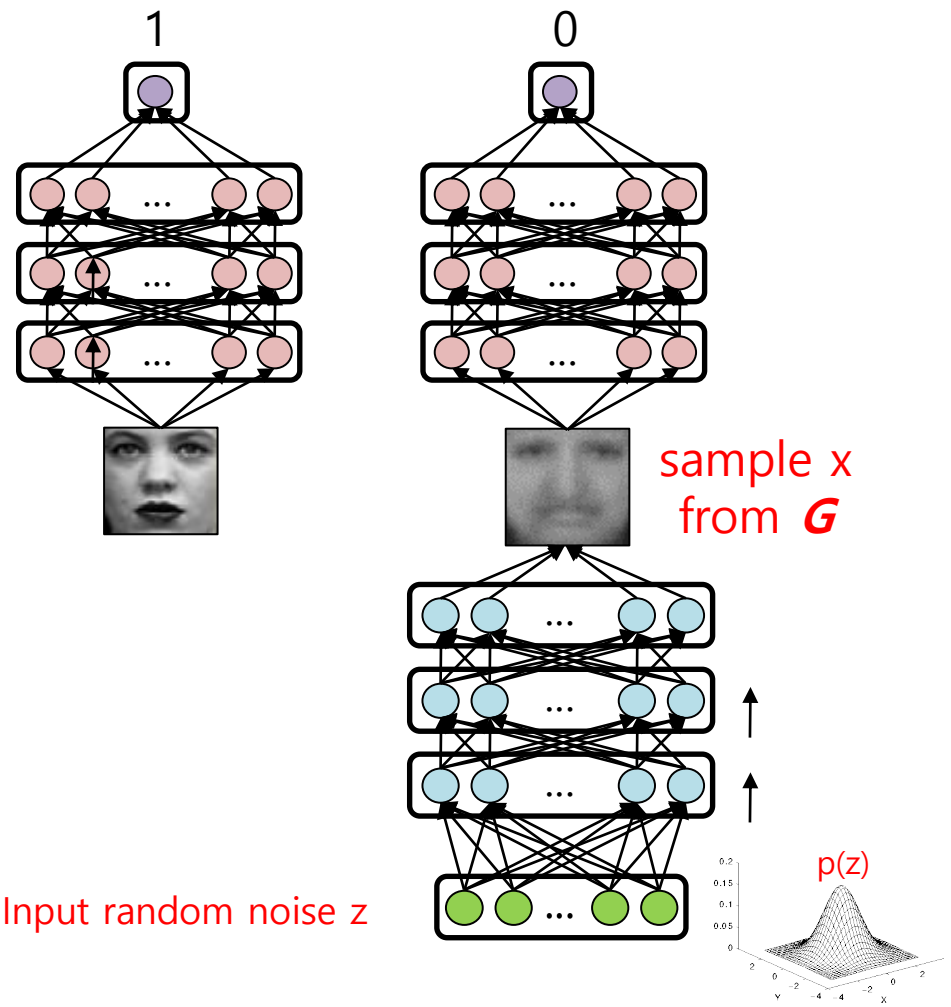
GENERATIVE ADVERSARIAL NETWORKS

Counterfeiters vs Police Game



(Sangdoo Yun @PIL)

GENERATIVE ADVERSARIAL NETWORKS



Discriminative Model *D*

Learns to classify the sample

- CNN
- D(x)=1 when x from Data
- D(G(z))=0 when the sample is from G (generator)

Generative Model *G*

Learns to generate sample

- Deconvolutional/Conv Transpose NN
- As similar as possible to the real data
- Target D(G(z))=1. To fool the Discriminator

GENERATIVE ADVERSARIAL NETWORKS

Output of a GAN on its first two epochs trained on the CelebA celebrity faces image dataset

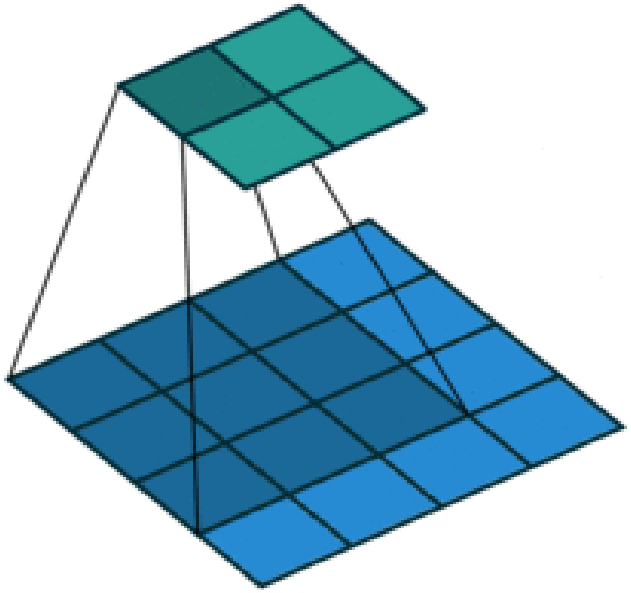
https://youtu.be/fN3egtFdA7s

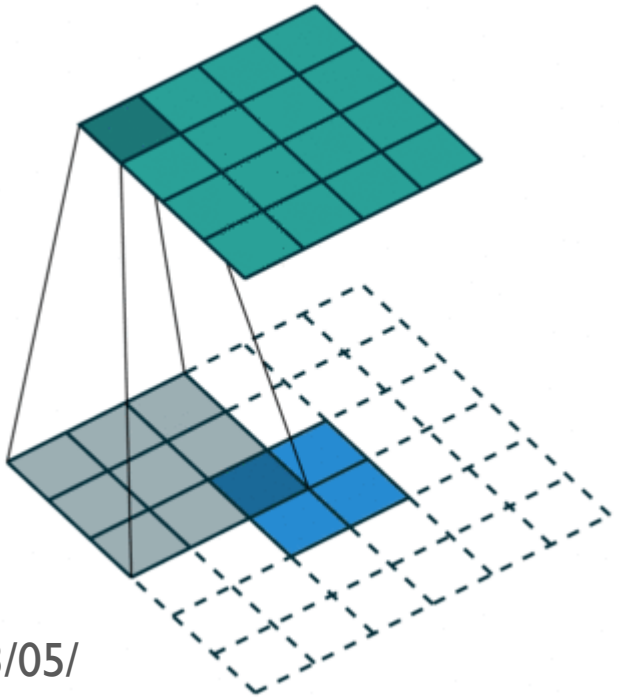
GENERATIVE ADVERSARIAL NETWORKS

- The Discriminator is usually a standard CNN however the Generator is usually a Transpose Convolutional or Deconvolutional Neural Network.
- A Convolutional Transpose layer essentially learns to map back from a feature/activation map that was output by a Convolutional layer to the input of that layer.
- Convolutional Transpose in Pytorch: http://www.pytorch.org/docs/0.3.1/nn.html#convtranspose2d

GENERATIVE ADVERSARIAL NETWORKS STABILITY TRICKS

Convolutional left vs Transpose Convolutional right



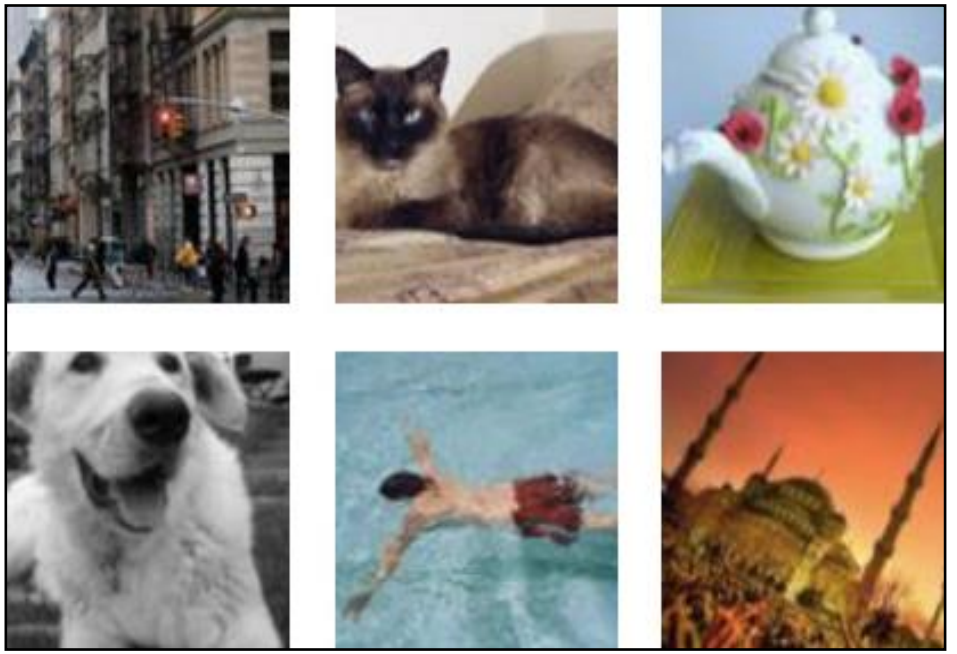


https://jhui.github.io/2017/03/05/
Generative-adversarial-models/

GENERATIVE ADVERSARIAL NETWORKS EXAMPLES

Unique sample generation

Training examples



Model Samples

https://arxiv.org/pdf/1701.00160

GENERATIVE ADVERSARIAL NETWORKS STABILITY TRICKS

GANs tend to be very unstable and are notoriously difficult to train. The GAN community has built a list of tricks that make GANs work. https://github.com/soumith/ganhacks

1. Track failures early

- D loss (or G loss) goes to 0: failure mode
- D or G loss max out and stay there (it's good to have an idea what your max loss is)
- Check norms of gradients: if they are over 100 it's gone wrong
- When things are working, D loss has low variance and goes down over time vs having huge variance and spiking
- If loss of generator steadily decreases, then it's fooling D with garbage

GENERATIVE ADVERSARIAL NETWORKS STABILITY TRICKS

2. A modified loss function

- In GAN papers, the loss function to optimize G is min log(1-D), but in practice max log D is used because the first formulation has vanishing gradients early on.

  Goodfellow et. al (2014)

GENERATIVE ADVERSARIAL NETWORKS STABILITY TRICKS

3. Normalize the inputs

- Normalize the images between -1 and 1
- Tanh as the last layer of the generator output

4. Use a spherical Z

- Don't sample from a Uniform distribution
- Sample from a Normal distribution

GENERATIVE ADVERSARIAL NETWORKS STABILITY TRICKS

5. BatchNorm

- Construct different mini-batches for real and fake, i.e. each mini-batch needs to contain only all real images or all generated images
- When batchnorm is not an option use instance normalization (for each sample, subtract mean and divide by standard deviation)

GENERATIVE ADVERSARIAL NETWORKS STABILITY TRICKS

6. Avoid Sparse Gradients: ReLU, MaxPool

- The stability of the GAN game suffers if you have sparse gradients
- LeakyReLU = good (in both G and D)
- For Downsampling, use: Average Pooling, Conv2d + stride
- For Upsampling, use: ConvTranspose2d + stride

GENERATIVE ADVERSARIAL NETWORKS STABILITY TRICKS

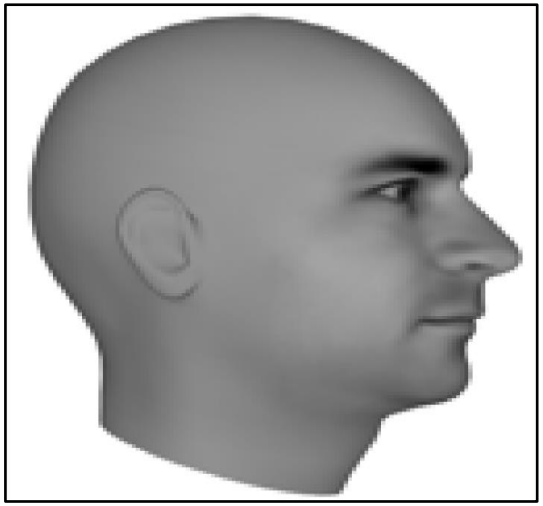
7. Use Soft and Noisy Labels

- Label Smoothing, i.e. if you have two target labels: Real=1 and Fake=0, then for each incoming sample, if it is real, then replace the label with a random number between 0.7 and 1.2, and if it is a fake sample, replace it with 0.0 and 0.3 (for example). Salimans et. al. 2016
- Make the labels noisy for the discriminator: occasionally flip the labels when training the discriminator

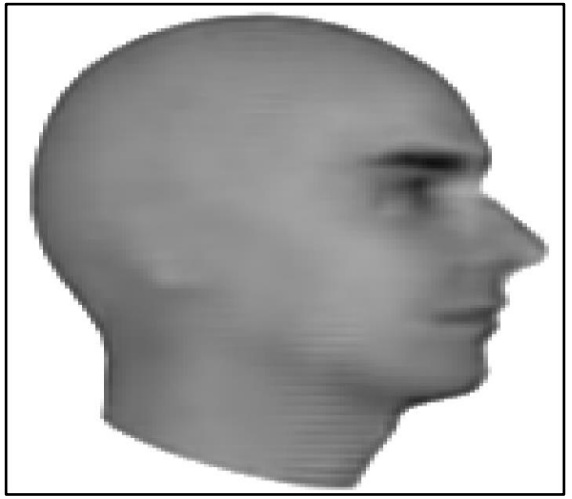
GENERATIVE ADVERSARIAL NETWORKS EXAMPLES

Next Video Frame Prediction

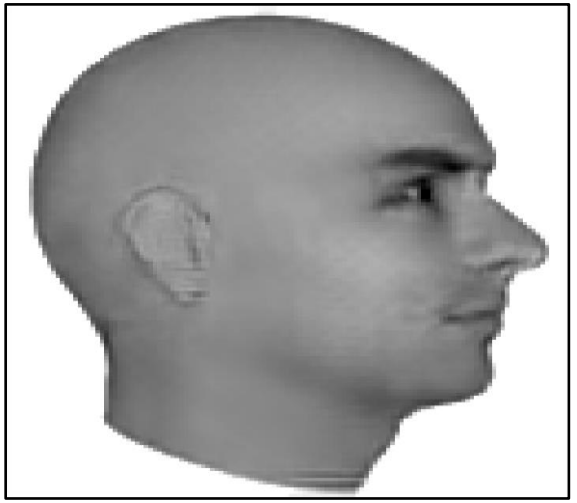
Ground Truth



MSE



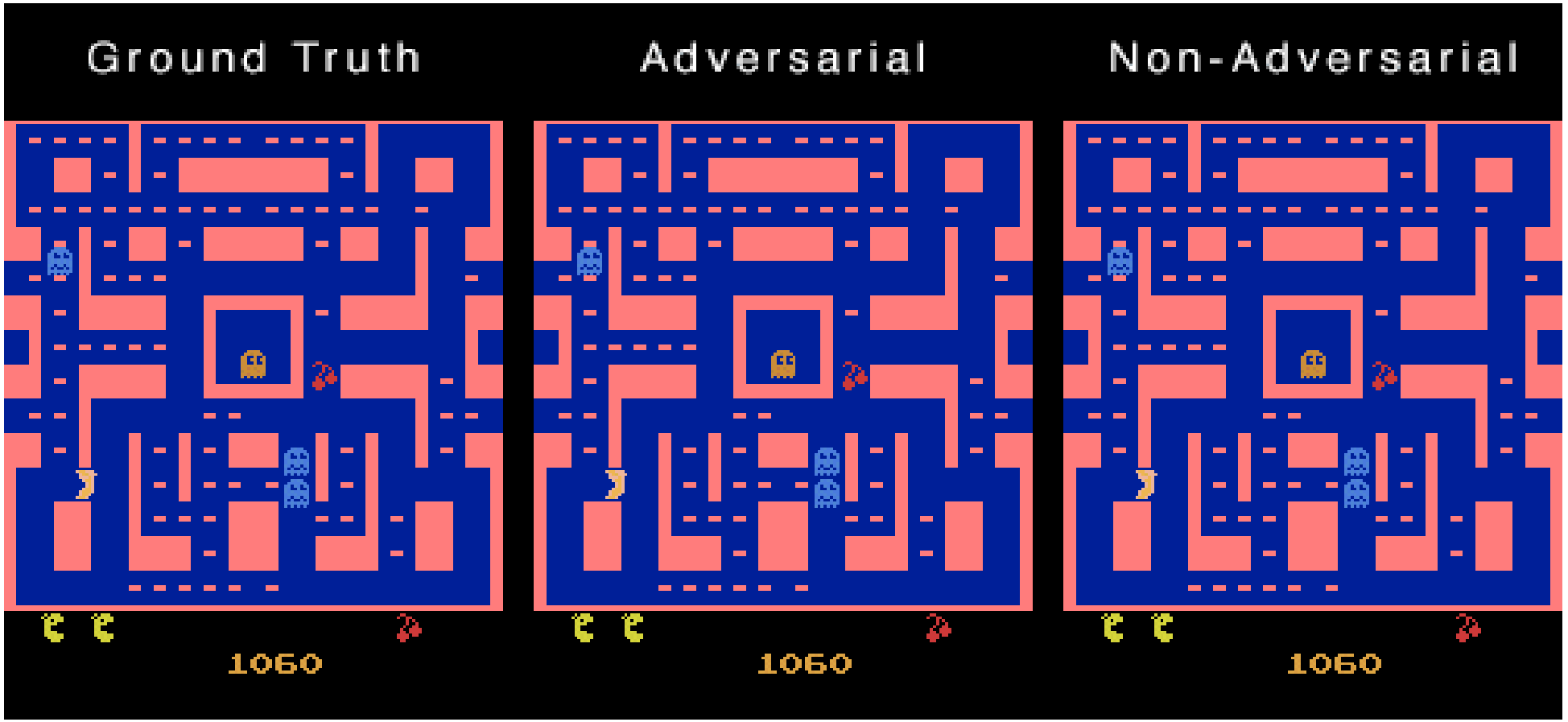
GAN



https://arxiv.org/pdf/1605.08104

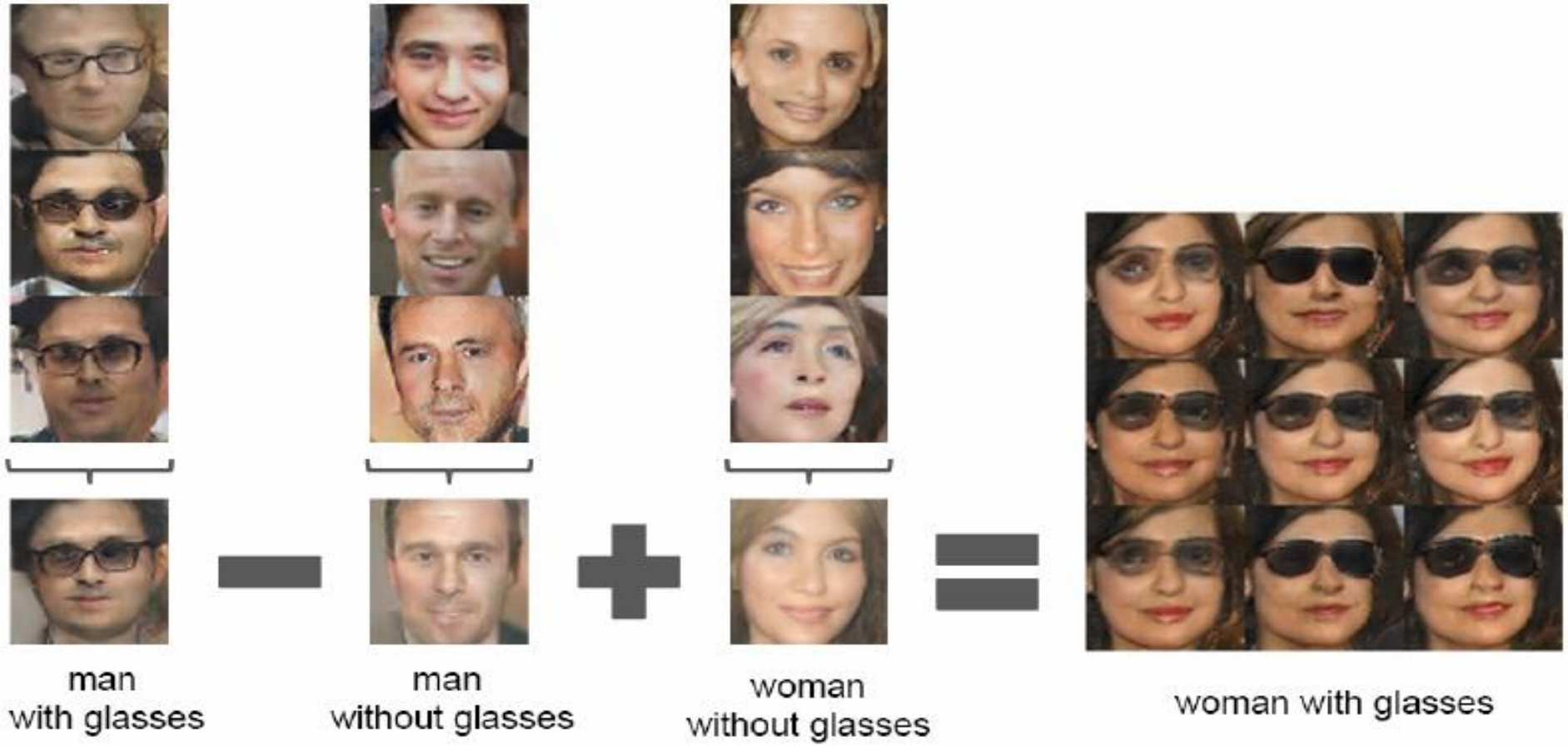
GENERATIVE ADVERSARIAL NETWORKS EXAMPLES

https://arxiv.org/abs/1511.05440



GENERATIVE ADVERSARIAL NETWORKS EXAMPLES

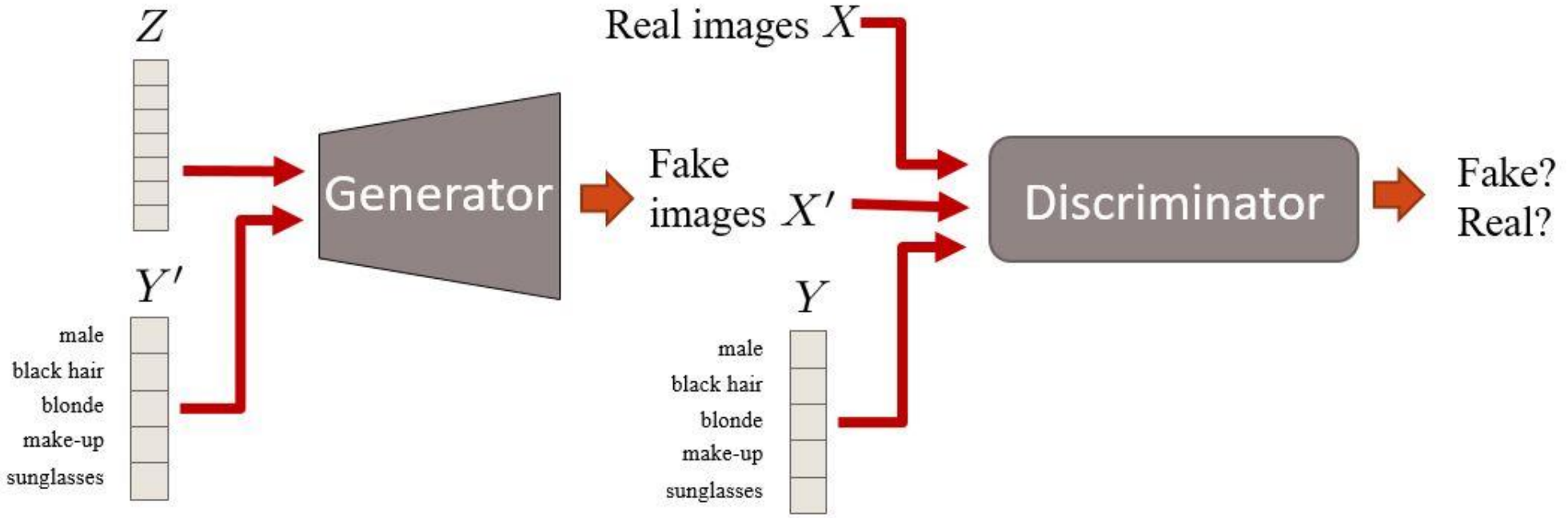
Semantic vector space arithmetic



https://arxiv.org/pdf/1511.06434.pdf%C3%AF%C2%BC%E2%80%B0

GENERATIVE ADVERSARIAL NETWORKS CONDITIONAL GANS

Conditional GANs. If you have labels use them. Your model will produce much better samples



GENERATIVE ADVERSARIAL NETWORKS EXAMPLES

Text to image StackGAN

This bird is completely red with black wings and pointy beak

A small sized bird that has a cream belly and a short pointed bill

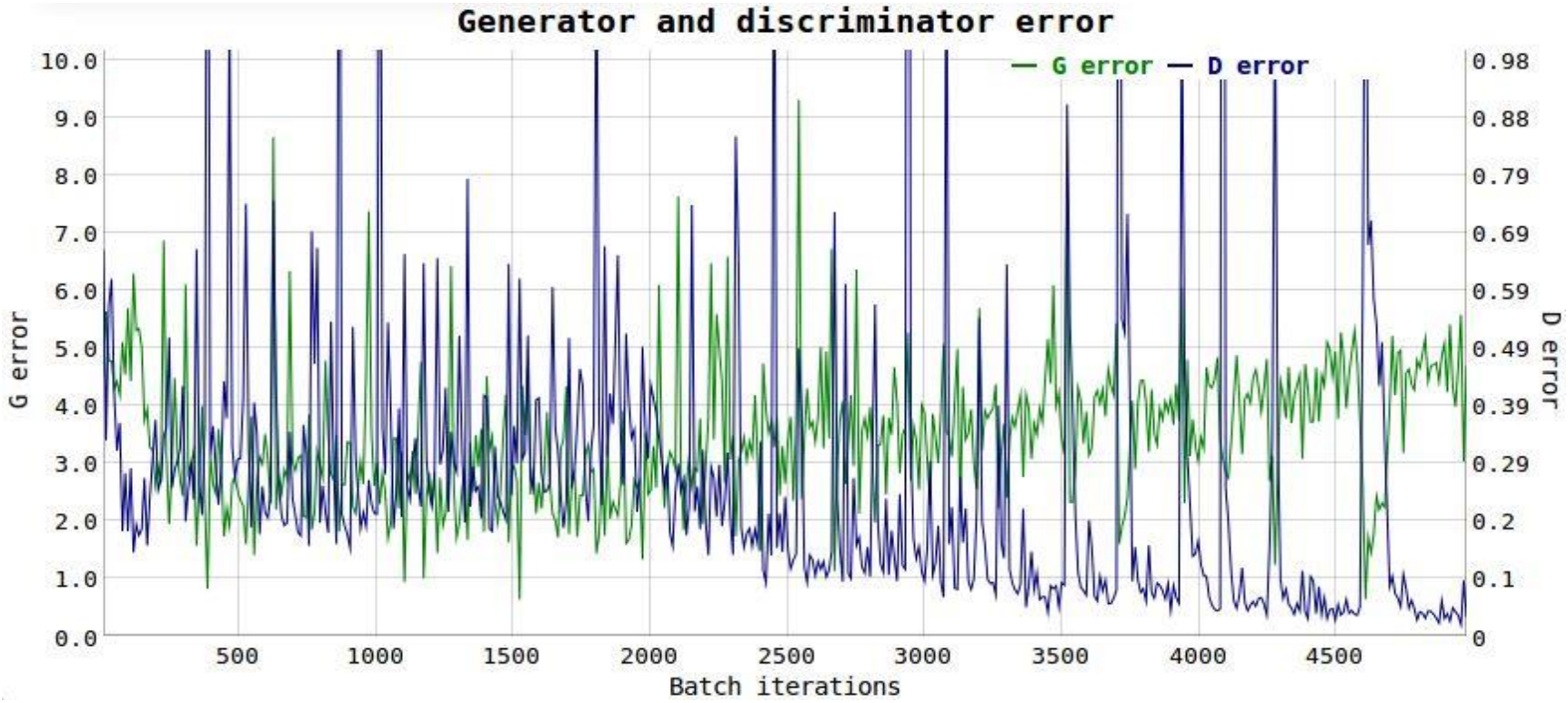
openaccess.thecvf.com/content_ICCV_2017/papers/Zhang_StackGAN_Text_to_ICCV_2017_paper.pdf

GENERATIVE ADVERSARIAL NETWORKS WGAN

- GANs have always had problems with convergence.
- The loss function doesn't correlate with image quality.
- As a consequence, you don't really know when to stop training them.
- So you need to be constantly looking at the samples to tell whether your model is training correctly or not and
- you don't have a numerical value that tells you how well are you tuning the parameters.

GENERATIVE ADVERSARIAL NETWORKS WGAN

Example: This is the plot of the loss functions of a DCGAN able to generate near perfect MNIST samples. Do you know when to stop training just by looking at this figure? Me neither.



http://guimperarnau.com/files/blog/Fantastic-GANs-and-where-to-find-them/crazy_loss_function.jpg

GENERATIVE ADVERSARIAL NETWORKS WGAN

- Traditional GANs minimise the f-divergence between the real data P(x) and the generated data $P_G(x)$.
- f-divergence is a function of the density ratio $\frac{P(x)}{P_G(x)}$.
- But if the two distributions don't overlap significantly the density ratio will be zero or infinite everywhere.

GENERATIVE ADVERSARIAL NETWORKS WGAN

Wasserstein GAN:

- Changes the loss function to include the Wasserstein distance
- The Wasserstein distance describes how far apart two distributions are even when the supports don't overlap significantly. It looks at how far you would need to move one distribution to create the other
- As a result, WGANs have loss functions that correlate with image quality. Also, training stability improves and is not as dependent on the architecture

GENERATIVE ADVERSARIAL NETWORKS WGAN

Paper: https://arxiv.org/abs/1701.07875
Code in Pytorch:
https://github.com/martinarjovsky/WassersteinGAN

GENERATIVE ADVERSARIAL NETWORKS WGAN

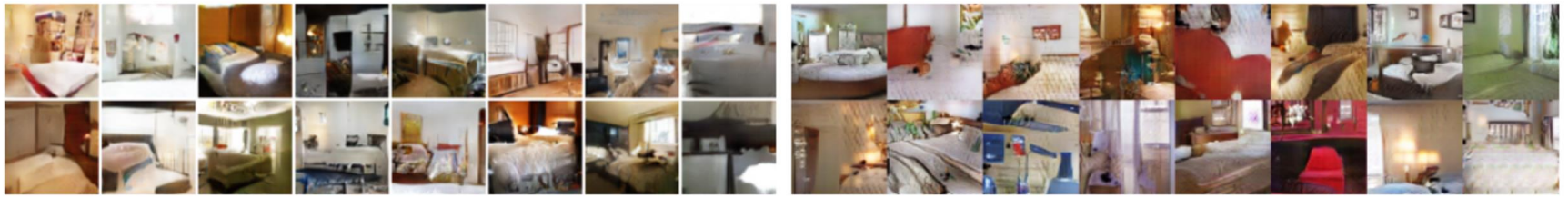


Figure 5: Algorithms trained with a DCGAN generator. Left: WGAN algorithm. Right: standard GAN formulation. Both algorithms produce high quality samples.

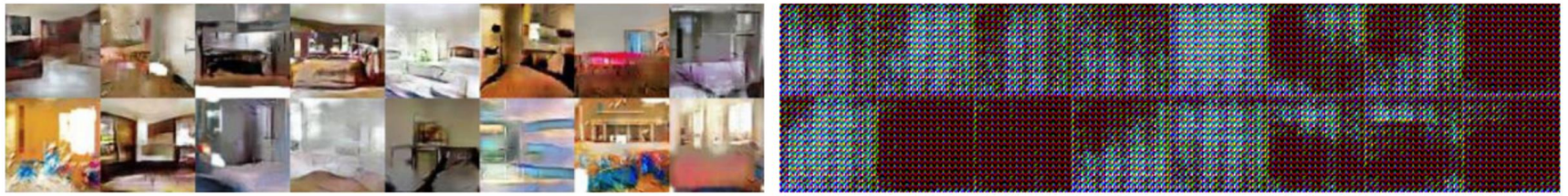
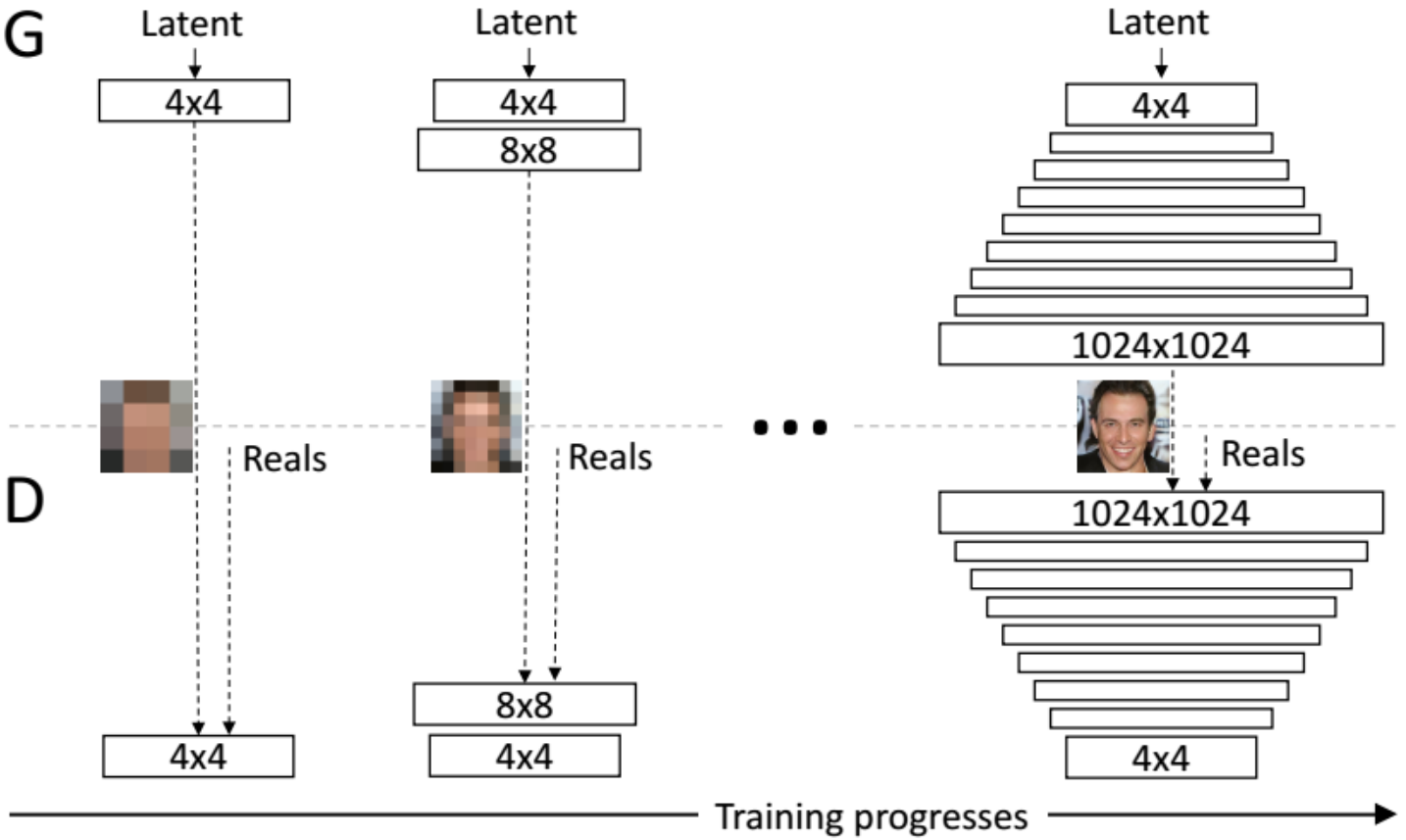


Figure 6: Algorithms trained with a generator without batch normalization and constant number of filters at every layer (as opposed to duplicating them every time as in [17]). Aside from taking out batch normalization, the number of parameters is therefore reduced by a bit more than an order of magnitude. Left: WGAN algorithm. Right: standard GAN formulation. As we can see the standard GAN failed to learn while the WGAN still was able to produce samples.

GENERATIVE ADVERSARIAL NETWORKS EXAMPLES

Progressive growing of GANS to create 1024x1024 images



http://research.nvidia.com/publication/2017-10_Progressive-Growing-of

GENERATIVE ADVERSARIAL NETWORKS EXAMPLES

Final 1024x1024 images unique images - top, with their nearest neighbour from the original dataset - bottom

http://research.nvidia.com/publication/2017-10_Progressive-Growing-of

GENERATIVE ADVERSARIAL NETWORKS EXAMPLES

State of the art face generation. Released late 2018.

GENERATIVE ADVERSARIAL NETWORKS EXAMPLES

State of the art face generation. Released late 2018. Have a go at spotting the difference at:

http://www.whichfaceisreal.com/

If you do badly try reading these tips then see if you do any better:

https://medium.com/@kcimc/how-to-recognize-fake-ai-generated-images-4d1f6f9a2842

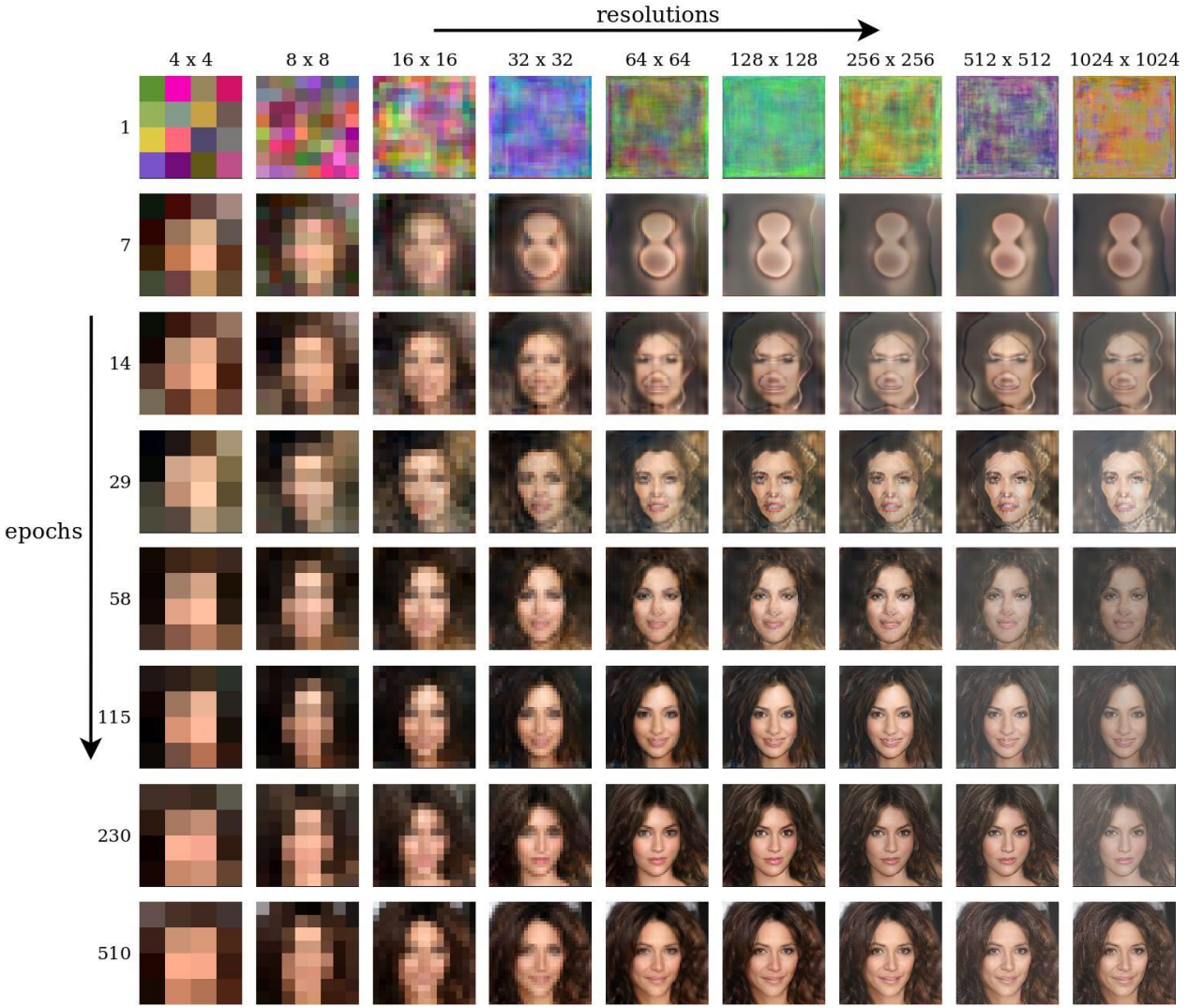
GENERATIVE ADVERSARIAL NETWORKS EXAMPLES

Useful GAN paper and code released last year:

- https://arxiv.org/pdf/1903.06048.pdf
- From the authors: “MSG-GAN architecture requires far less (even none) hyperparameter tuning. Through extensive experimentation, we show that the same hyperparameter settings translate to various model architectures for different datasets.” ..
- The model is not state of the art but produces high quality results across many different datasets with the same training settings.
- Code in Pytorch:

  https://github.com/akanimax/BMSG-GAN?fbclid=IwAR2-lGg3iXVcQS4I4bfzOgtyedMsmBSP5zFOylelpf-3PFLTi0xDMFcx_z8

GENERATIVE ADVERSARIAL NETWORKS EXAMPLES



GENERATIVE ADVERSARIAL NETWORKS EXAMPLES

GENERATIVE ADVERSARIAL NETWORKS EXAMPLES

Semi-supervised learning using GANS

CIFAR-10

Model	Test error rate for a given number of labeled samples			
	1000	2000	4000	8000
Ladder network [25]			20.40±0.47	
CatGAN [14]			19.58±0.46	
Our model	21.83±2.01	19.61±2.09	18.63±2.32	17.72±1.82
Ensemble of 10 of our models	19.22±0.54	17.25±0.66	15.59±0.47	14.87±0.89

SVHN

Model	Number of incorrectly predicted test examples for a given number of labeled samples			
	20	50	100	200
DGN [22]			333 ± 14	
Virtual Adversarial [23]			212	
CatGAN [14]			191 ± 10	
Skip Deep Generative Model [24]			132 ± 7	
Ladder network [25]			106 ± 37	
Auxiliary Deep Generative Model [24]			96 ± 2	
Our model	1677 ± 452	221 ± 136	93 ± 6.5	90 ± 4.2
Ensemble of 10 of our models	1134 ± 445	142 ± 96	86 ± 5.6	81 ± 4.3

papers.nips.cc/paper/6125-improved-techniques-for-training-gans.pdf