

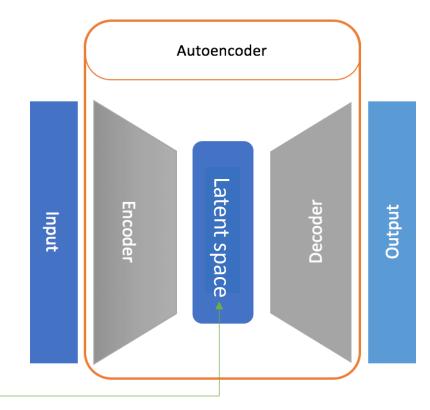
• An autoencoder is a neural network that is trained to produce as output (r)

Code

a duplicate of its input (x).

• The network may be viewed as consisting of two parts:

- An **encoder** function z = f(x)
- z is called **Code** (of the input)
- A **decoder** that produces a reconstruction r = g(z)
- The **objective** is  $x \approx r$
- The **loss function** is expressed as L(x, g(f(x)))



 A trained autoencoder has learnt and summarized the main characteristics of the input data

 These characteristics are able to represent the original data in a shorter data structure

 Statistical properties of the original data are summarized too into a latent space of network weights



- Forcing x = r (strong risk of overfitting)
  - Dimensionality reduction
  - Feature learning
- The real strength of an autoencoder is the capability of abstract the data instead of learning to perfectly copy them.
  - Usually they are restricted in finding approximations, producing an output that likely can belong to the input data domain
    - Data generation
    - Data completion
    - Data reconstruction
    - Anomaly detection



- The Code of an autoencoder can have useful information about the input domain
- An autoencoder, whose code size is lower than the input one, is called under-complete
  - The code is able to compress and summarize the properties of the input data
  - When the decoder is linear and L is the mean squared error, autoencoder = PCA.
- An autoencoder, whose code size is bigger than the input one, is called over-complete
  - The code is able to map the input data in a higher dimensional domain acting as Kernel function (do you remember SVMs?)

### Regularized Autoencoders

- Autoencoder issues:
  - Size of the code
  - Excessive or reduced fitting capability of the encoder and the decoder
- Regularized autoencoders ideally enables to choose the code size and the capacity of the encoder and decoder according to the complexity of the input data distribution
- How to achieve this result?
  - Change the loss function!

### Regularized Autoencoders

- The new loss function encourages the model to have other properties besides the ability to copy its input, e.g.:
  - sparsity of the representation
  - smallness of the derivative of the representation
  - robustness to noise or to missing inputs
- Short list of regularized autoencoders:
  - Sparse Autoencoders
  - Denoising Autoencoders
  - Penalizing Derivative Autoencoders

### Sparse Autoencoders

 Sparse Autoencoders add a sparsity penalty on the code layer to the loss function:

$$L(x,g(f(x))) + \Omega(z)$$

- The component  $\Omega(z)$  penalizes the activations of too many nodes in the code layer
  - This constraint is called the sparsity constraint. It is sparse because each unit only activates to a certain type of inputs, not all of them

• E.g.:

$$\Omega(z) = \lambda \sum_{i \in |h|} |z_i|$$

where  $\lambda$  is a constant

# ▶ Denoising Autoencoders

 Denoising Autoencoders learn underlying patterns by changing the reconstruction error term of the cost function:

$$L\left(x,g(f(\tilde{x}))\right)$$

where  $\tilde{x}$  is the original input affected by some noise distribution

Their main goal is to deleted corruption in a set of data

### Penalizing Derivative

#### Autoencoders

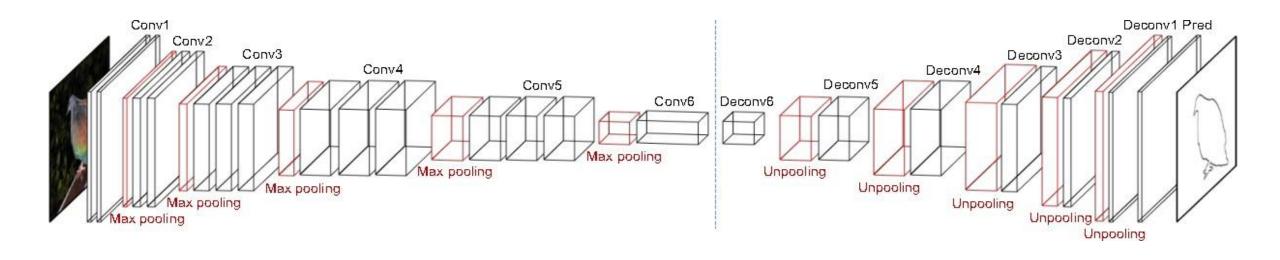
Similarly to sparse autoencoders, we can define another penalty function

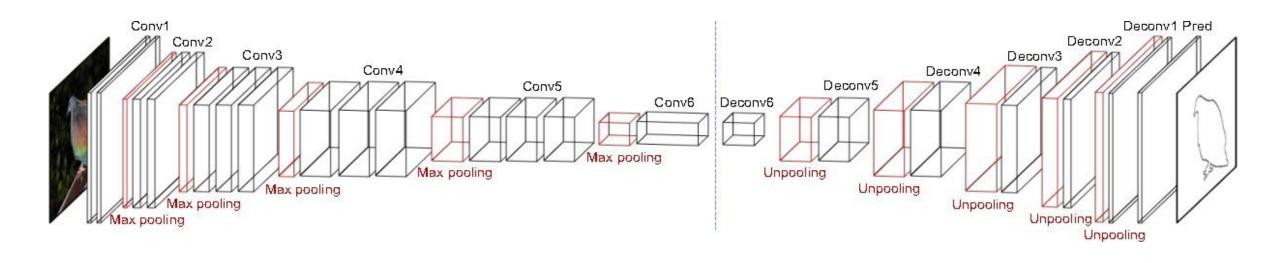
$$L(x,g(f(x))) + \Omega(x,z)$$

where:

$$\Omega(x,z) = \lambda \sum_{i \in |h|} ||\nabla_x z_i||^2$$

- with  $\lambda$  constant
- The model is forced to learn a function that does not change much when x slightly changes



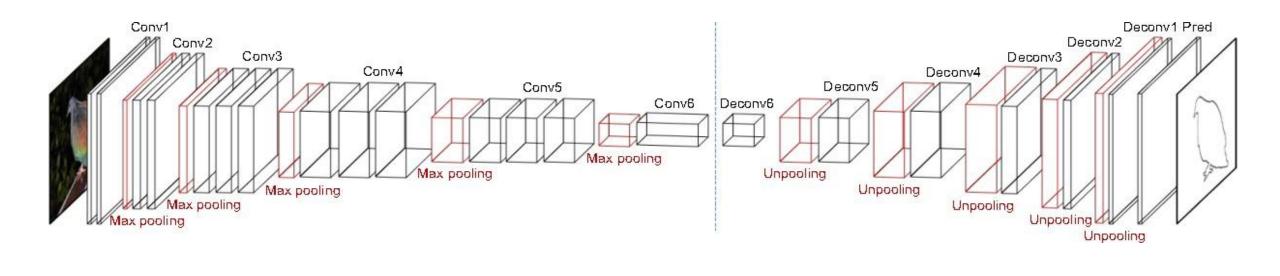


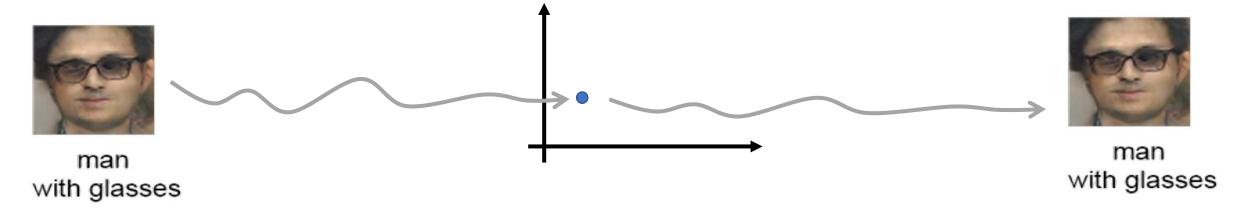


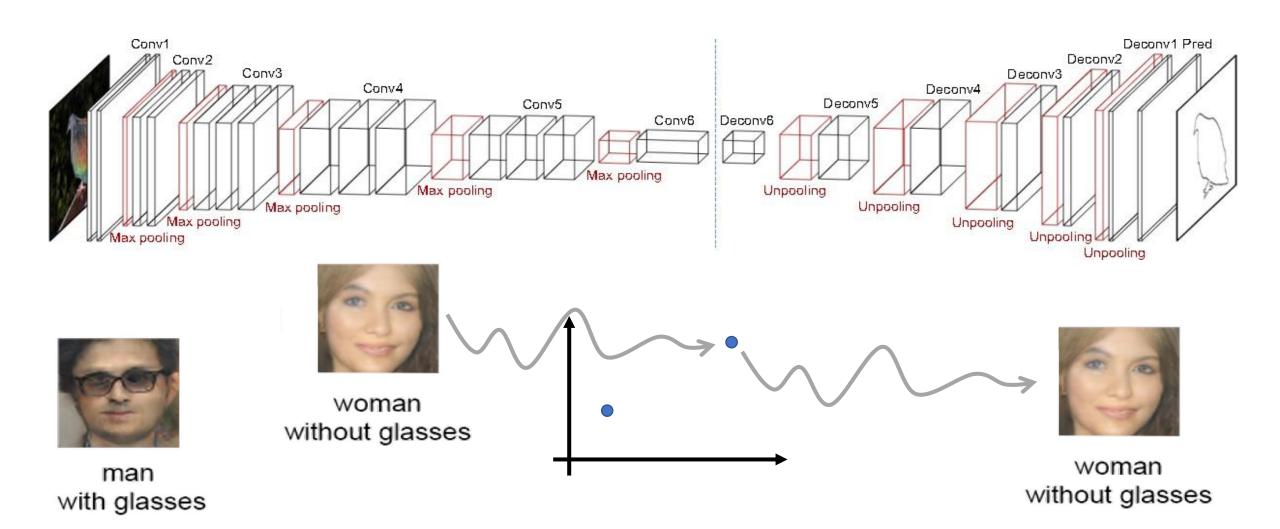
man with glasses

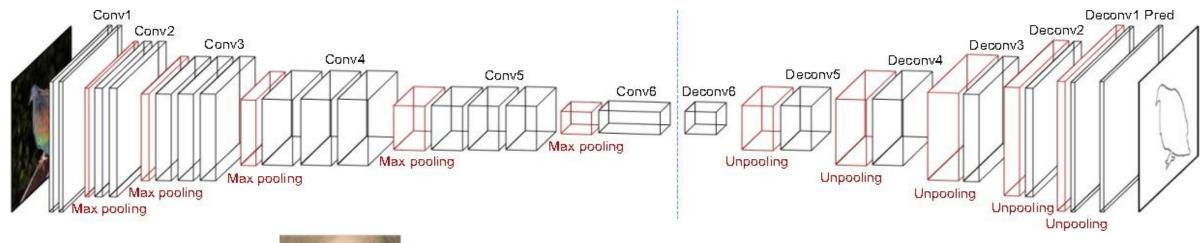


man with glasses







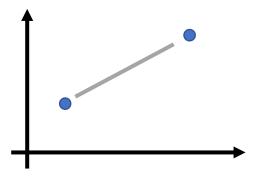


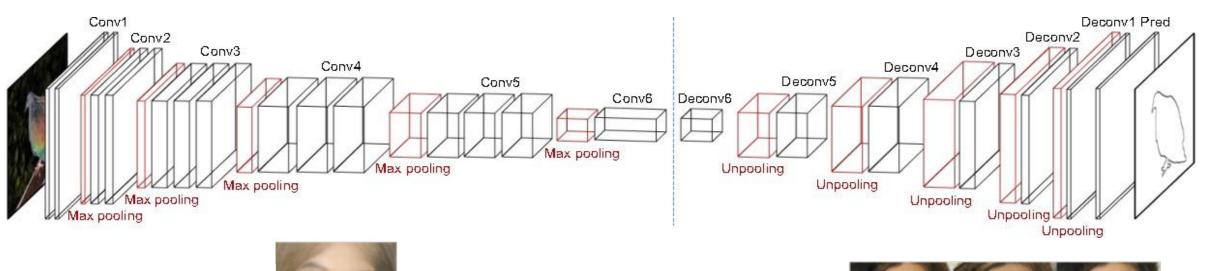


man with glasses



woman without glasses



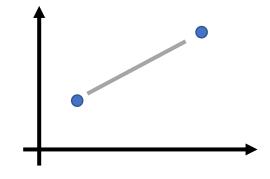




man with glasses



woman without glasses





#### DeepFakes!

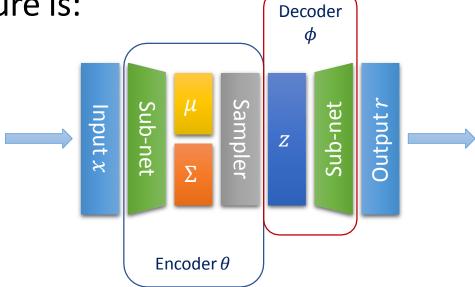
- Immagini non reali, generate con tecniche di intelligenza artificiale
- Altissima fedeltà, indistinguibili da reali fotografie



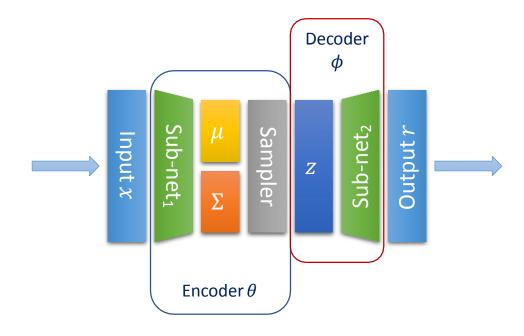
### Variational Autoencoders

 To avoid overfitting and to increase generality we can add a random sampling

• The new architecture is:

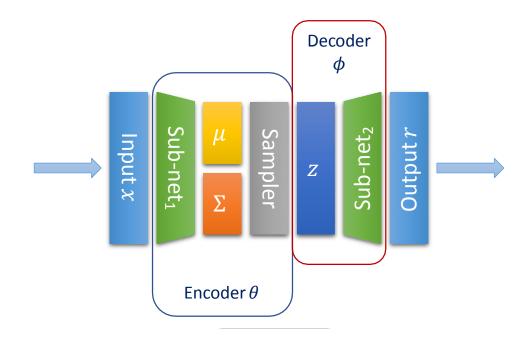


#### Variational Autoencoders



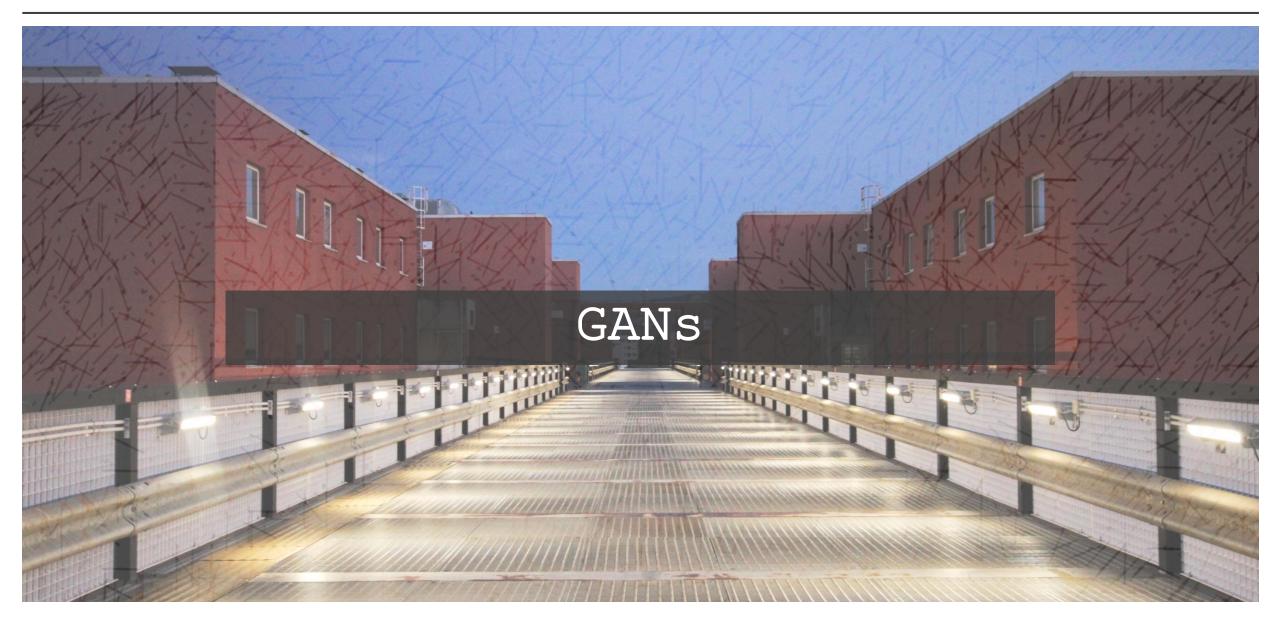
- The sub-net<sub>1</sub> computes two parameters of a Gaussian distribution
- The sampler exploits the parameters to normally sample the code
- The sub-net<sub>2</sub> generates two parameters of a Gaussian distribution

#### Variational Autoencoders

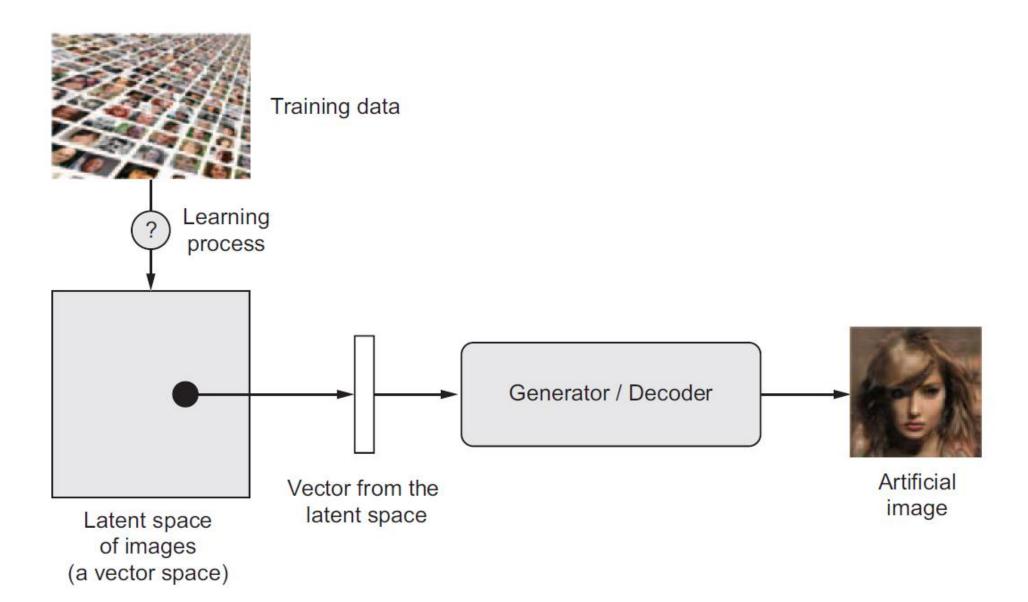


- The encoder, with weights and biases  $\theta$ , samples  $z \sim q_{\theta}(z|x)$
- The decoder, with weights and biases  $\phi$ , generates  $r \approx x$ 
  - It reconstructs  $x \sim p_{\phi}(x|z)$





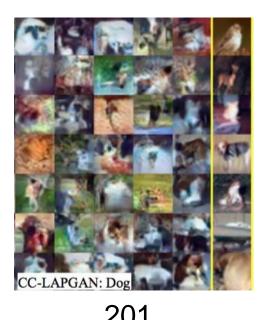




#### Overview

- In generative modeling, we'd like to train a network that models a distribution, such as a distribution over images.
- Done way to judge the quality of the model is to sample from it. This field has seen rapid progress:



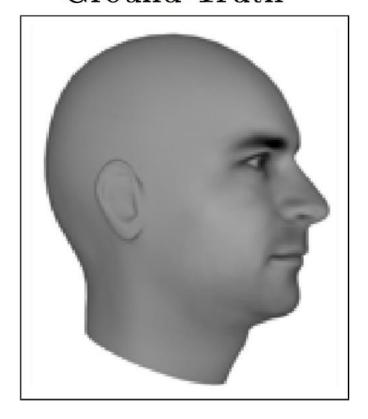




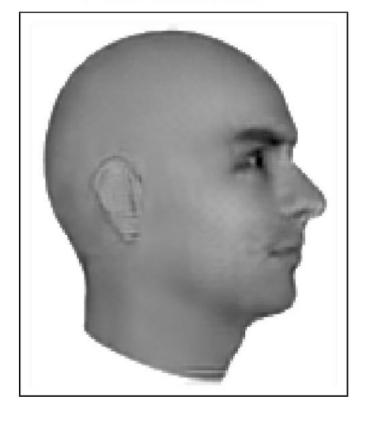
201



Ground Truth



Adversarial

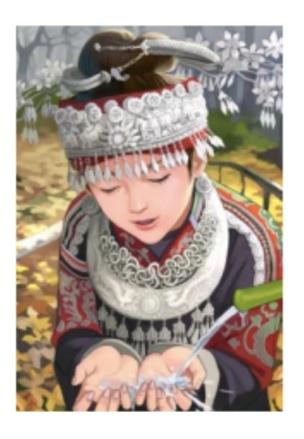


Lotter, William, Gabriel Kreiman, and David Cox. "Unsupervised learning of visual structure using predictive generative networks." arXiv preprint arXiv:1511.06380 (2015).

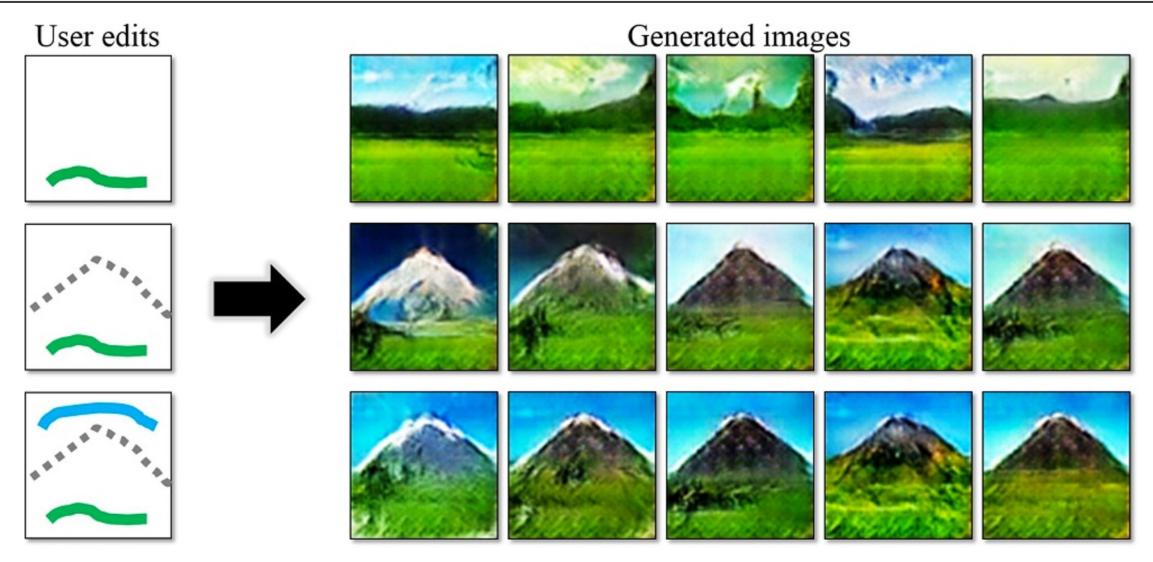


#### Which one is Computergenerated?









#### Implicit Generative Models

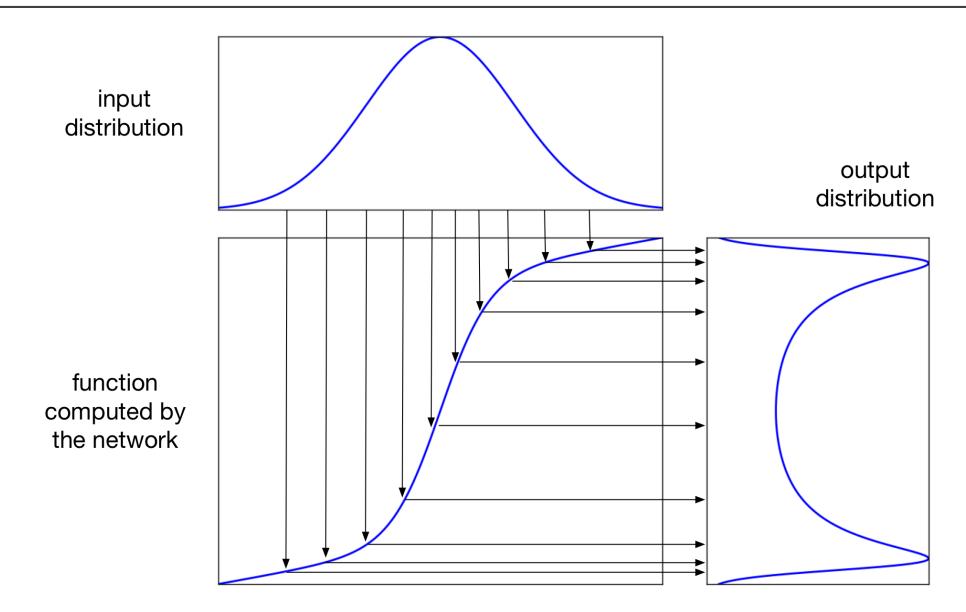
- Implicit generative models implicitly define a probability distribution
- ▶ Start by sampling the code vector **z** from a fixed, simple distribution
- The generator network computes a differentiable function *G* mapping **z** to an **x** in data space

sample

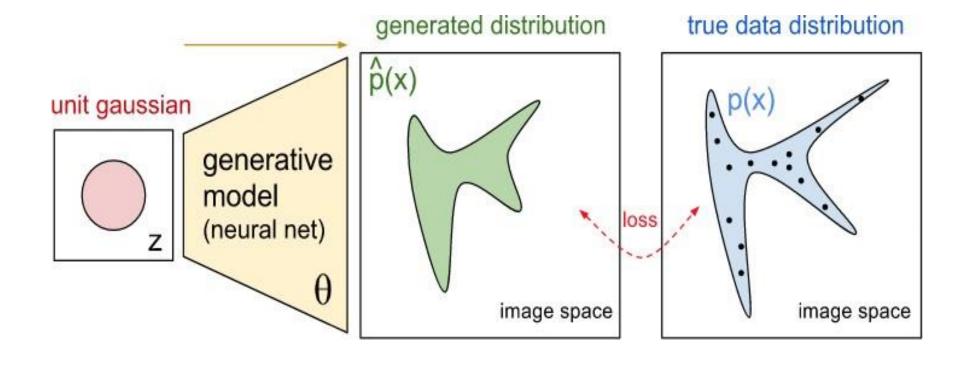
 ${f Z}$ 

code vector

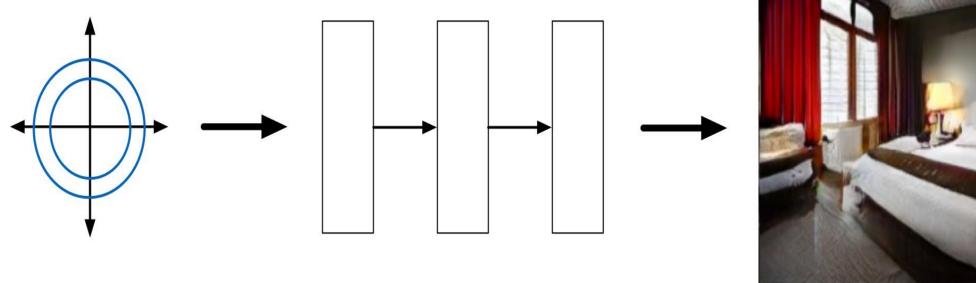
# ■ 1-Dimensional example



# ■2-Dimensional example



#### More general approaches



Each dimension of the code vector is sampled independently from a simple distribution, e.g. Gaussian or uniform.

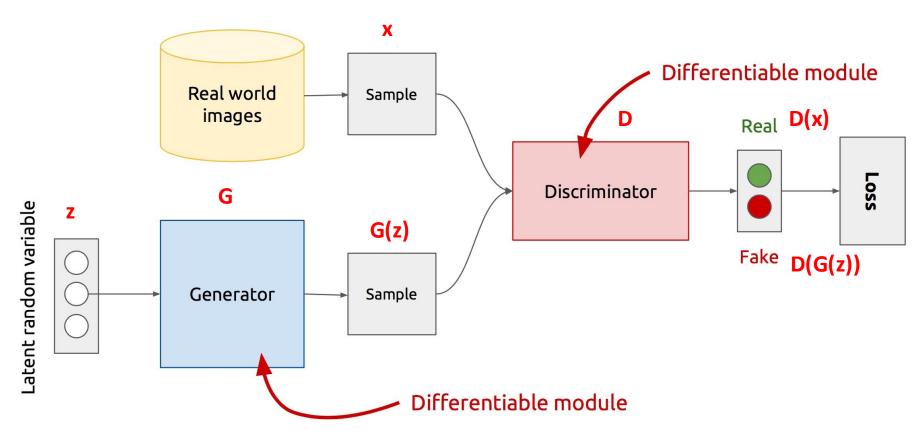
This is fed to a (deterministic) generator network.

The network outputs an image.



- ▶ The advantage of implicit generative models:
  - ▶ if you have some criterion for evaluating the quality of samples, then you can compute its gradient with respect to the network parameters, and update the network's parameters to make the sample a little better
- The idea behind Generative Adversarial Networks (GANs): train two different networks
  - ▶ The generator network tries to produce realistic-looking samples
  - ▶ The discriminator network tries to figure out whether an image came from the training set or the generator network
  - ▶ The generator network tries to fool the discriminator network

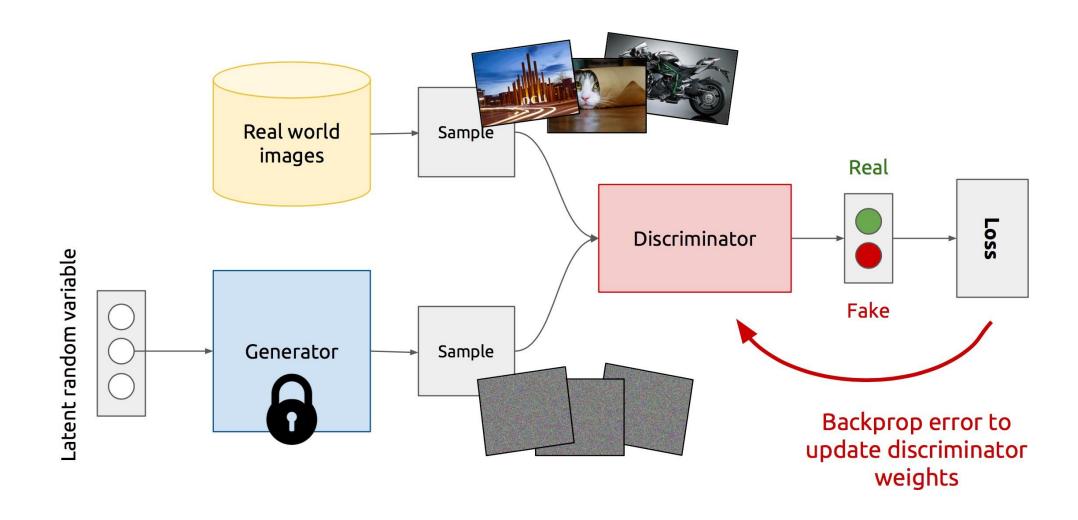
#### GAN's Architecture



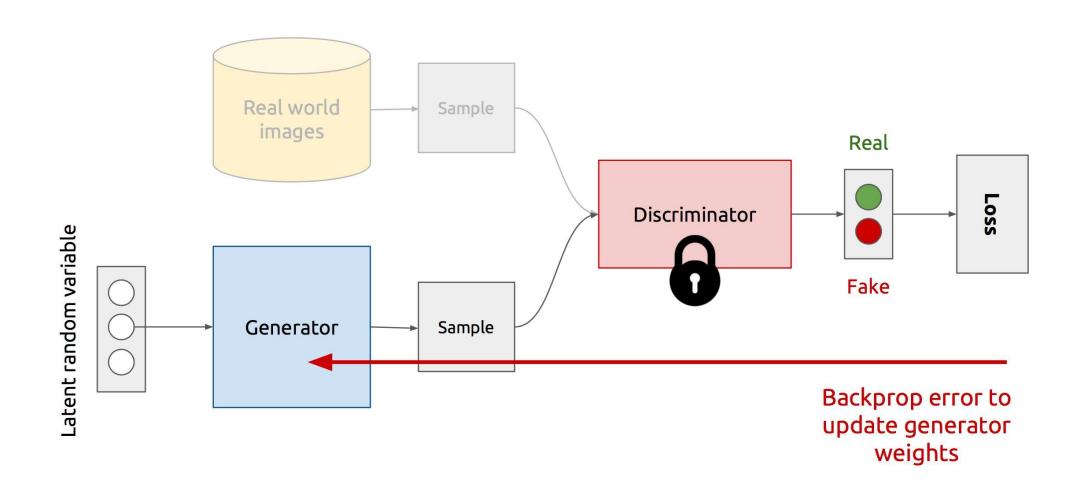
- **Z** is some random noise (Gaussian/Uniform).
- Z can be thought as the latent representation of the image.

2023/2024 MACHINE LEARNING

### Training the Discriminator



# Training the Generator





#### Putting it all together

#### for number of training iterations do

#### **Discriminator** updates

#### for k steps do

- Sample minibatch of m noise samples  $\{z^{(1)}, \ldots, z^{(m)}\}$  from noise prior  $p_q(z)$ .
- Sample minibatch of m examples  $\{x^{(1)}, \ldots, x^{(m)}\}$  from data generating distribution  $p_{\text{data}}(\boldsymbol{x})$ .
- Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^m \left[ \log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right].$$

#### end for

- Sample minibatch of m noise samples  $\{z^{(1)}, \ldots, z^{(m)}\}$  from noise prior  $p_q(z)$ .
- Update the generator by descending its stochastic gradient:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log \left( 1 - D \left( G \left( \boldsymbol{z}^{(i)} \right) \right) \right).$$

#### Generator updates

end for

## GAN Examples

Celebrities and bedrooms

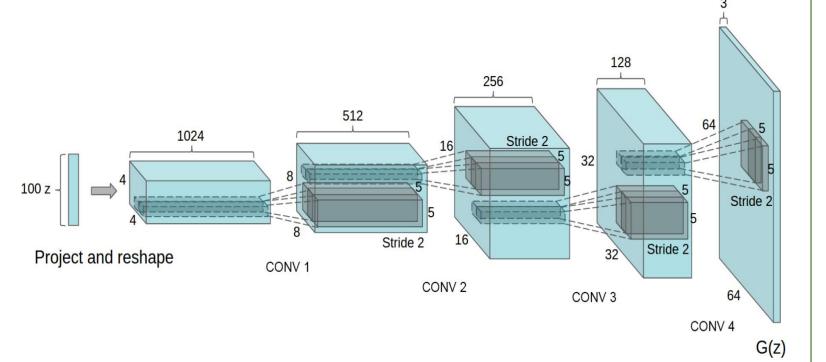




Karras et al., 2017. Progressive growing of GANs for improved quality, stability, and

## Deep Convolutional GANs

#### **Generator Architecture**



#### **Key ideas:**

- Replace FC hidden layers with Convolutions
  - Generator: Fractional-Strided convolutions
- Use Batch Normalization after each layer
- Inside Generator
  - Use ReLU for hidden layers
  - Use Tanh for the output layer



