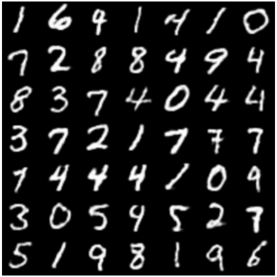
CSC413/2516 Lecture 9: GANs and Reversible Models

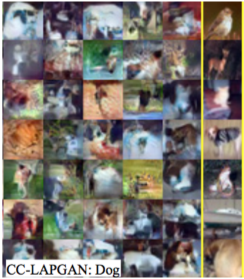
Jimmy Ba and Bo Wang

Overview

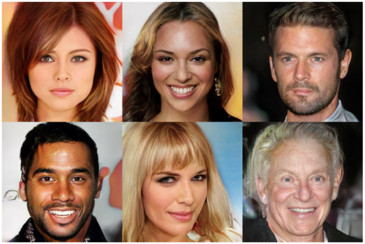
- In generative modeling, we'd like to train a network that models a distribution, such as a distribution over images.
- One way to judge the quality of the model is to sample from it.
- This field has seen rapid progress:



2009



2015



2018

Overview

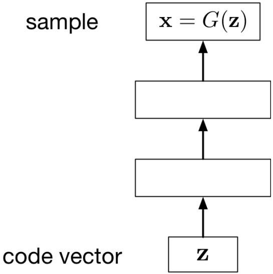
Four modern approaches to generative modeling:

- Autoregressive models (Lectures 3, 7, and 8)
- Generative adversarial networks (this lecture)
- Reversible architectures (this lecture)
- Variational autoencoders (next lecture)

All four approaches have different pros and cons.

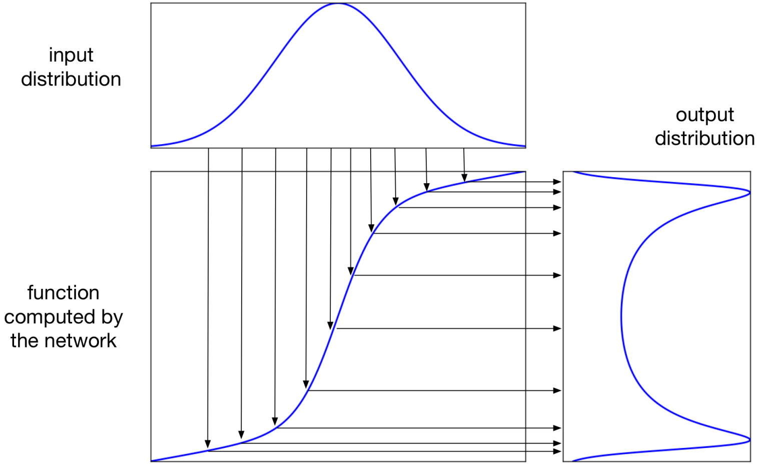
Generator Networks

- Autoregressive models explicitly predict a distribution at each step.
- Another approach to generative modeling is to train a neural net to produce approximate samples from the distribution.
- Start by sampling the code vector z from a fixed, simple distribution (e.g. spherical Gaussian)
- The generator network computes a differentiable function G mapping z to an x in data space

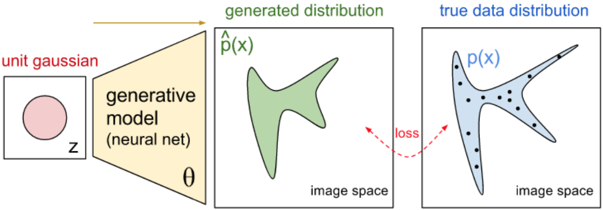


Generator Networks

A 1-dimensional example:

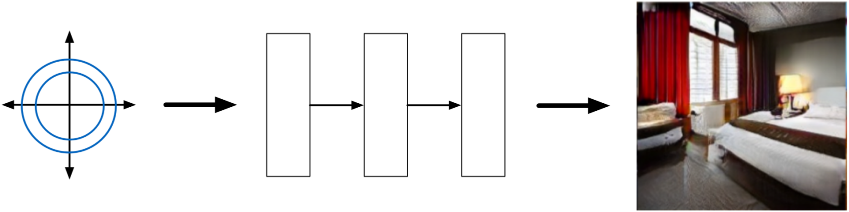


Generator Networks



https://blog.openai.com/generative-models/

Generator Networks



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.

This sort of architecture sounded preposterous to many of us, but amazingly, it works.

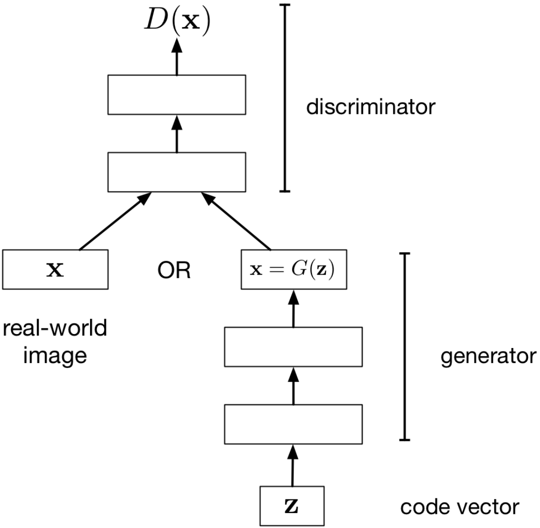
Generative Adversarial Networks

- Implicit generative models learn a mapping from random noise vectors to things that look like, e.g., images
- 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

Generative Adversarial Networks

- Implicit generative models learn a mapping from random noise vectors to things that look like, e.g., images
- 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

Generative Adversarial Networks



Generative Adversarial Networks

- Let D denote the discriminator's predicted probability of being data
- Discriminator's cost function: cross-entropy loss for task of classifying real vs. fake images

$$\mathcal{J}_D = \mathbb{E}_{\mathbf{x}\sim\mathcal{D}}[-\log D(\mathbf{x})] + \mathbb{E}_{\mathbf{z}}[-\log(1 - D(G(\mathbf{z})))]$$

- One possible cost function for the generator: the opposite of the discriminator's

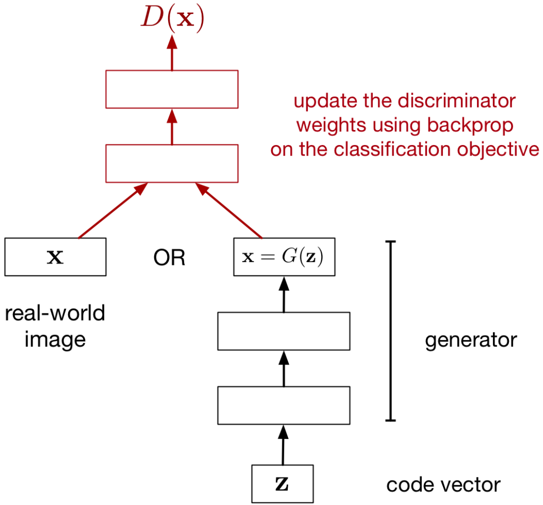
$$\begin{aligned} \mathcal{J}_G &= -\mathcal{J}_D \\ &= \text{const} + \mathbb{E}_{\mathbf{z}}[\log(1 - D(G(\mathbf{z})))] \end{aligned}$$

- This is called the minimax formulation, since the generator and discriminator are playing a zero-sum game against each other:

$$\max_G \min_D \mathcal{J}_D$$

Generative Adversarial Networks

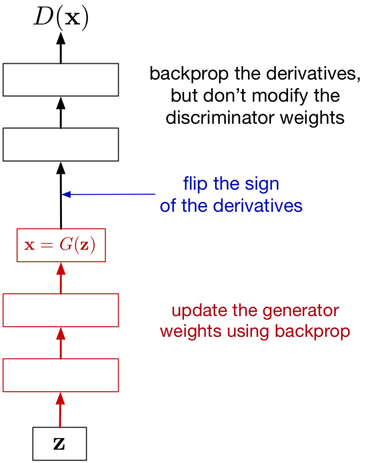
Updating the discriminator:



$D(\mathbf{x})$
update the discriminator weights using backprop on the classification objective
$\mathbf{x}$
OR
$\mathbf{x} = G(\mathbf{z})$
real-world image
generator
$\mathbf{z}$
code vector

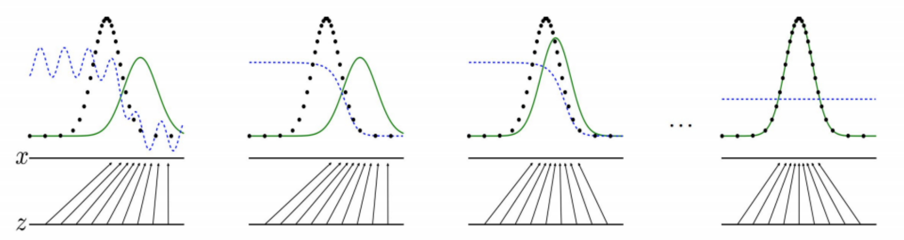
Generative Adversarial Networks

Updating the generator:


$D(\mathbf{x})$
backprop the derivatives, but don't modify the discriminator weights
flip the sign of the derivatives
$\mathbf{x} = G(\mathbf{z})$
update the generator weights using backprop
$\mathbf{z}$

Generative Adversarial Networks

Alternating training of the generator and discriminator:



A Better Cost Function

- We introduced the minimax cost function for the generator:

$$\mathcal{J}_G = \mathbb{E}_{\mathbf{z}}[\log(1 - D(G(\mathbf{z})))]$$

- One problem with this is saturation.
- Recall from our lecture on classification: when the prediction is really wrong,
  - "Logistic + squared error" gets a weak gradient signal
  - "Logistic + cross-entropy" gets a strong gradient signal
- Here, if the generated sample is really bad, the discriminator's prediction is close to 0, and the generator's cost is flat.

A Better Cost Function

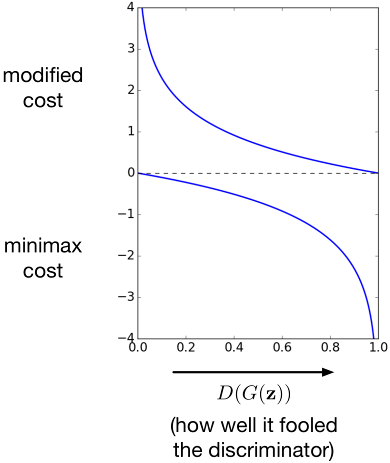
- Original minimax cost:

$$\mathcal{J}_G = \mathbb{E}_{\mathrm{z}}[\log(1 - D(G(\mathrm{z})))]$$

- Modified generator cost:

$$\mathcal{J}_G = \mathbb{E}_{\mathrm{z}}[-\log D(G(\mathrm{z}))]$$

- This fixes the saturation problem.

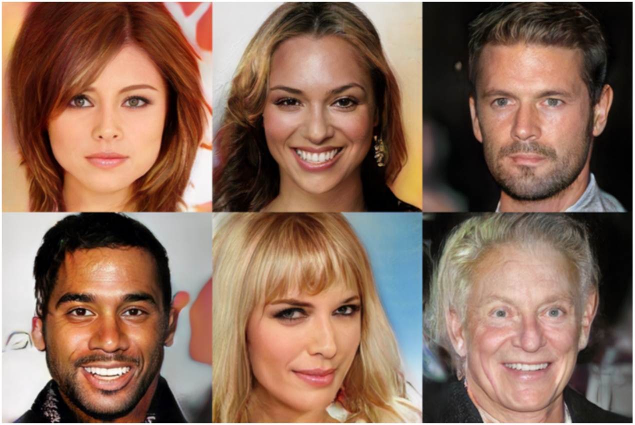


Generative Adversarial Networks

- Since GANs were introduced in 2014, there have been hundreds of papers introducing various architectures and training methods.
- Most modern architectures are based on the Deep Convolutional GAN (DC-GAN), where the generator and discriminator are both conv nets.
- GAN Zoo: `https://github.com/hindupuravinash/the-gan-zoo`
  - Good source of horrible puns (VEEGAN, Checkhov GAN, etc.)

GAN Samples

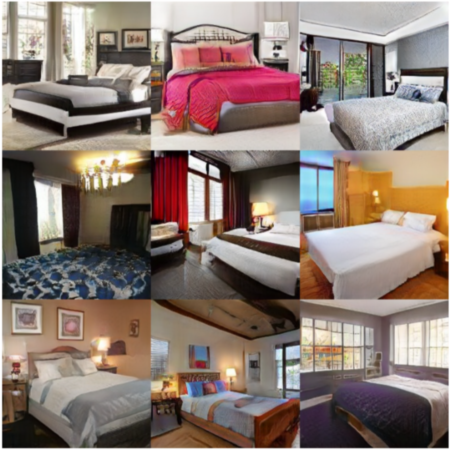
Celebrities:



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

GAN Samples

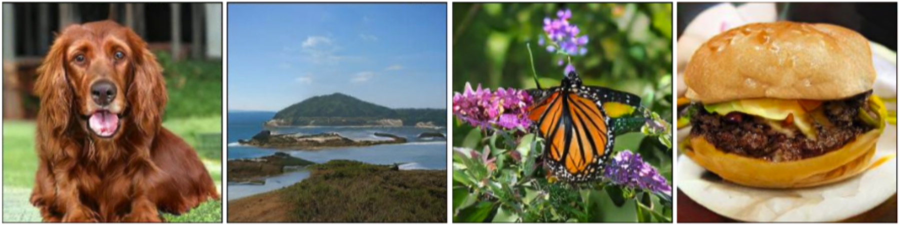
Bedrooms:



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

GAN Samples

ImageNet object categories (by BigGAN, a much larger model with a bunch more engineering tricks):



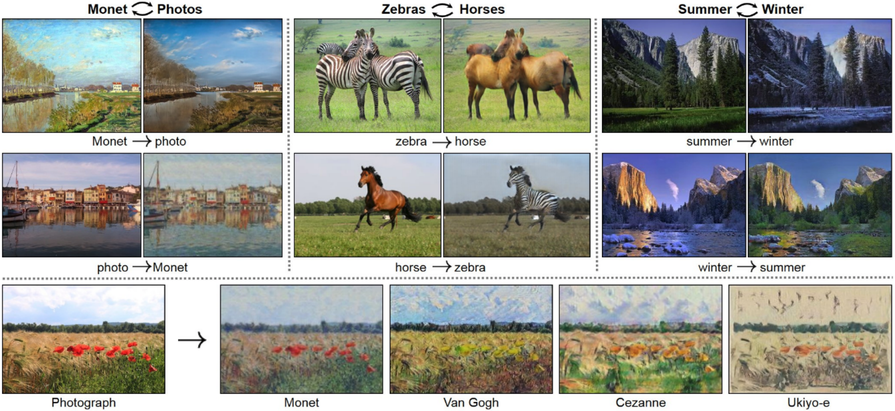
Brock et al., 2019. Large scale GAN training for high fidelity natural image synthesis.

GAN Samples

- GANs revolutionized generative modeling by producing crisp, high-resolution images.
- The catch: we don't know how well they're modeling the distribution.
    - Can't measure the log-likelihood they assign to held-out data.
    - Could they be memorizing training examples? (E.g., maybe they sometimes produce photos of real celebrities?)
    - We have no way to tell if they are dropping important modes from the distribution.
    - See Wu et al., "On the quantitative analysis of decoder-based generative models" for partial answers to these questions.

CycleGAN

Style transfer problem: change the style of an image while preserving the content.

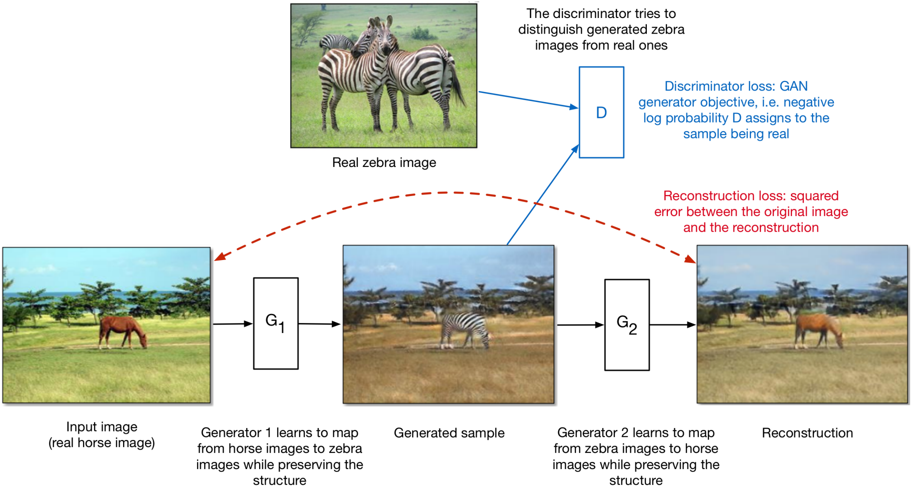


Data: Two unrelated collections of images, one for each style

CycleGAN

- If we had paired data (same content in both styles), this would be a supervised learning problem. But this is hard to find.
- The CycleGAN architecture learns to do it from unpaired data.
  - Train two different generator nets to go from style 1 to style 2, and vice versa.
  - Make sure the generated samples of style 2 are indistinguishable from real images by a discriminator net.
  - Make sure the generators are cycle-consistent: mapping from style 1 to style 2 and back again should give you almost the original image.

CycleGAN



Real zebra image

The discriminator tries to distinguish generated zebra images from real ones

D

Discriminator loss: GAN generator objective, i.e. negative log probability D assigns to the sample being real

Reconstruction loss: squared error between the original image and the reconstruction

G_1

G_2

Input image
(real horse image)

Generator 1 learns to map from horse images to zebra images while preserving the structure

Generated sample

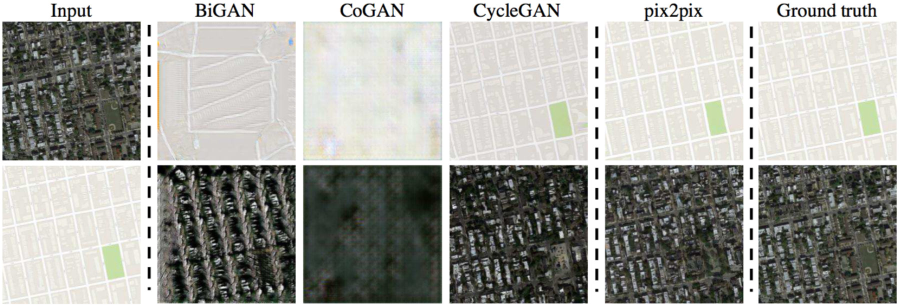
Generator 2 learns to map from zebra images to horse images while preserving the structure

Reconstruction

Total loss = discriminator loss + reconstruction loss

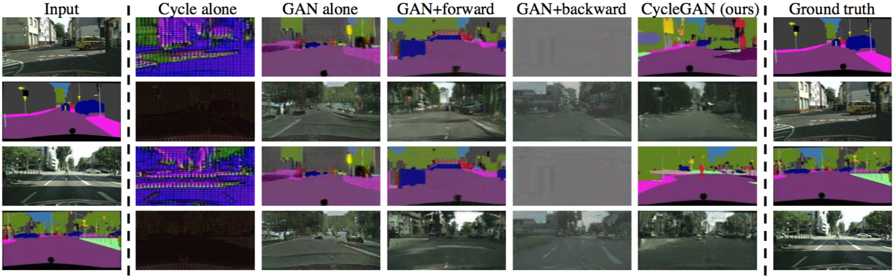
CycleGAN

Style transfer between aerial photos and maps:



CycleGAN

Style transfer between road scenes and semantic segmentations (labels of every pixel in an image by object category):



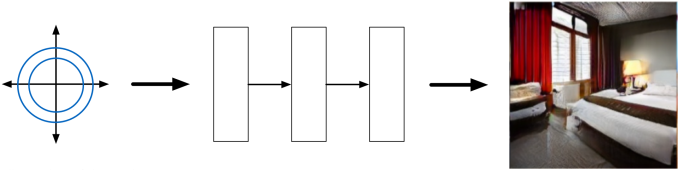
Overview

Four modern approaches to generative modeling:

- Autoregressive models (Lectures 3, 7, and 8)
- Generative adversarial networks (this lecture)
- Reversible architectures (this lecture)
- Variational autoencoders (next lecture)

All four approaches have different pros and cons.

Generator Networks



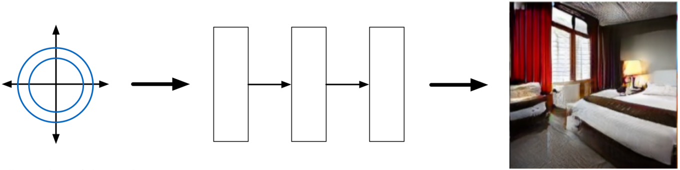
Each dimension of the code vector is sampled independently from a simple distribution, e.g. Gaussian or uniform.

This is fed to a (deterministic) generator network.

The network outputs an image.

- We have seen how to learn generator networks by training a discriminator in GANs.
- Problem:

Generator Networks



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.

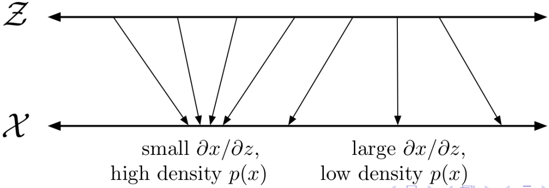
- We have seen how to learn generator networks by training a discriminator in GANs.
- Problem:
    - Learning can be very unstable. Need to tune many hyperparameters.
    - No direct evaluation metric to assess the trained generator networks.
- Idea: learn the generator directly via change of variables. (Calculus!)

Change of Variables Formula

- Let f denote a differentiable, bijective mapping from space $\mathcal{Z}$ to space $\mathcal{X}$. (I.e., it must be 1-to-1 and cover all of $\mathcal{X}$.)
- Since f defines a one-to-one correspondence between values $\mathrm{z} \in \mathcal{Z}$ and $\mathrm{x} \in \mathcal{X}$, we can think of it as a change-of-variables transformation.
- Change-of-Variables Formula from probability theory: if $\mathrm{x} = f(\mathrm{z})$, then

$$p_X(\mathrm{x}) = p_Z(z) \left| \det \left(\frac{\partial \mathrm{x}}{\partial \mathrm{z}} \right) \right|^{-1}$$

- Intuition for the Jacobian term:



Change of Variables Formula

- Suppose we have a generator network which computes the function f. It's tempting to apply the change-of-variables formula in order to compute the density $p_X(\mathrm{x})$.
- I.e., compute $\mathrm{z} = f^{-1}(\mathrm{x})$

$$p_X(\mathrm{x}) = p_Z(z) \left| \det \left(\frac{\partial \mathrm{x}}{\partial z} \right) \right|^{-1}$$

- Problems?

Change of Variables Formula

- Suppose we have a generator network which computes the function f. It's tempting to apply the change-of-variables formula in order to compute the density $p_X(\mathrm{x})$.
- I.e., compute $\mathrm{z} = f^{-1}(\mathrm{x})$

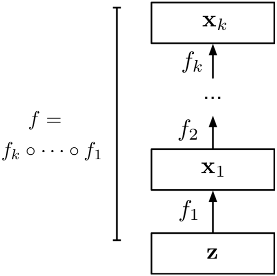
$$p_X(\mathrm{x}) = p_Z(z) \left| \det \left(\frac{\partial \mathrm{x}}{\partial z} \right) \right|^{-1}$$

- Problems?
  - It needs to be differentiable, so that the Jaobian $\partial \mathrm{x}/\partial z$ is defined.
  - The mapping f needs to be invertible, with an easy-to-compute inverse.
  - We need to be able to compute the (log) determinant.
- Differentiability is easy (just use a differentiable activation function), but the other requirements are trickier.

Reversible Blocks

- Now let's define a reversible block which is invertible and has a tractable determinant.

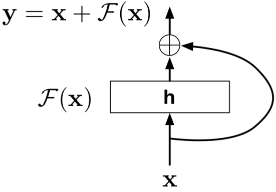
- Such blocks can be composed.
  - Inversion: $f^{-1} = f_1^{-1} \circ \cdots \circ f_k^{-1}$
  - Determinants: $\left|\frac{\partial \mathrm{x}_k}{\partial \mathrm{z}}\right| = \left|\frac{\partial \mathrm{x}_k}{\partial \mathrm{x}_{k-1}}\right| \cdots \left|\frac{\partial \mathrm{x}_2}{\partial \mathrm{x}_1}\right| \left|\frac{\partial \mathrm{x}_1}{\partial \mathrm{z}}\right|$



Reversible Blocks

- Recall the residual blocks:

$$y = x + \mathcal{F}(x)$$



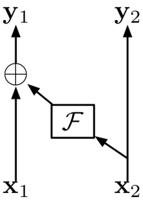
$\mathbf{y} = \mathbf{x} + \mathcal{F}(\mathbf{x})$
$\mathcal{F}(\mathbf{x})$
h
$\mathbf{x}$

- Reversible blocks are a variant of residual blocks. Divide the units into two groups, x_1 and x_2.

$$y_1 = x_1 + \mathcal{F}(x_2)$$
$$y_2 = x_2$$

- Inverting a reversible block:

$$x_2 = y_2$$
$$x_1 = y_1 - \mathcal{F}(x_2)$$



Reversible Blocks

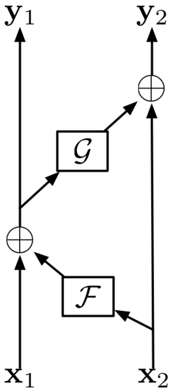
Composition of two reversible blocks, but with x_1 and x_2 swapped:

- Forward:

$$\begin{aligned} y_1 &= x_1 + \mathcal{F}(x_2) \\ y_2 &= x_2 + \mathcal{G}(y_1) \end{aligned}$$

- Backward:

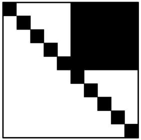
$$\begin{aligned} x_2 &= y_2 - \mathcal{G}(y_1) \\ x_1 &= y_1 - \mathcal{F}(x_2) \end{aligned}$$



Volume Preservation

- It remains to compute the log determinant of the Jacobian.
- The Jacobian of the reversible block:

$$\begin{aligned} y_1 &= x_1 + \mathcal{F}(x_2) \\ y_2 &= x_2 \end{aligned} \qquad \frac{\partial y}{\partial x} = \begin{pmatrix} I & \frac{\partial \mathcal{F}}{\partial x_2} \\ 0 & I \end{pmatrix}$$



- This is an upper triangular matrix. The determinant of an upper triangular matrix is the product of the diagonal entries, or in this case, 1.
- Since the determinant is 1, the mapping is said to be volume preserving.

Nonlinear Independent Components Estimation

- We've just defined the reversible block.
  - Easy to invert by subtracting rather than adding the residual function.
  - The determinant of the Jacobian is 1.
- Nonlinear Independent Components Estimation (NICE) trains a generator network which is a composition of lots of reversible blocks.
- We can compute the likelihood function using the change-of-variables formula:

$$p_X(\mathrm{x}) = p_Z(\mathrm{z}) \left| \det \left(\frac{\partial \mathrm{x}}{\partial \mathrm{z}} \right) \right|^{-1} = p_Z(\mathrm{z})$$

- We can train this model using maximum likelihood. I.e., given a dataset $\{\mathrm{x}^{(1)}, \ldots, \mathrm{x}^{(N)}\}$, we maximize the likelihood

$$\prod_{i=1}^{N} p_X(\mathrm{x}^{(i)}) = \prod_{i=1}^{N} p_Z(f^{-1}(\mathrm{x}^{(i)}))$$

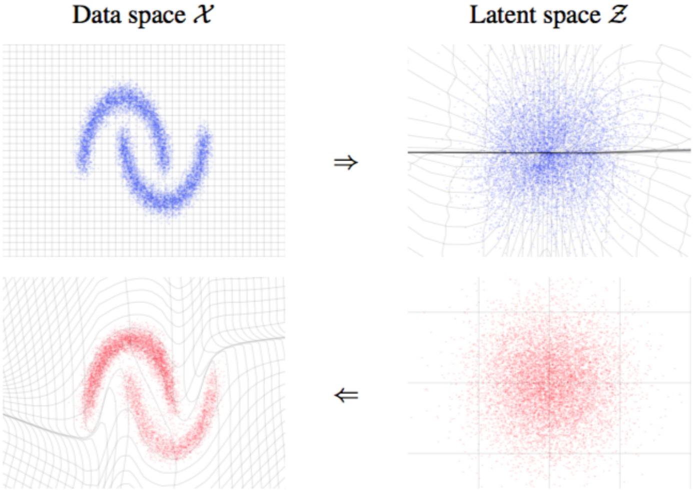
Nonlinear Independent Components Estimation

- Likelihood:

$$p_X(\mathrm{x}) = p_Z(\mathrm{z}) = p_Z(f^{-1}(\mathrm{x}))$$

- Remember, p_Z is a simple, fixed distribution (e.g. independent Gaussians)
- Intuition: train the network such that f^{-1} maps each data point to a high-density region of the code vector space $\mathcal{Z}$.
    - Without constraints on f, it could map everything to 0, and this likelihood objective would make no sense.
    - But it can't do this because it's volume preserving.

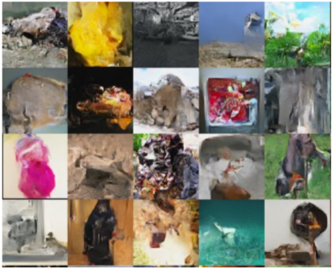
Nonlinear Independent Components Estimation



Dinh et al., 2016. Density estimation using RealNVP.

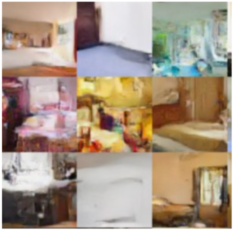
Nonlinear Independent Components Estimation

Samples produced by RealNVP, a model based on NICE.



ImageNet

celebrities



bedrooms

Dinh et al., 2016. Density estimation using RealNVP.

Overview

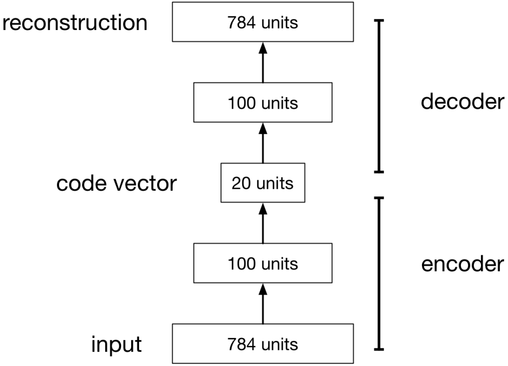
Four modern approaches to generative modeling:

- Autoregressive models (Lectures 3, 7, and 8)
- Generative adversarial networks (this lecture)
- Reversible architectures (this lecture)
- Variational autoencoders (next lecture)

All four approaches have different pros and cons.

Autoencoders

- An autoencoder is a feed-forward neural net whose job it is to take an input x and predict x.
- To make this non-trivial, we need to add a bottleneck layer whose dimension is much smaller than the input.



Autoencoders

Why autoencoders?

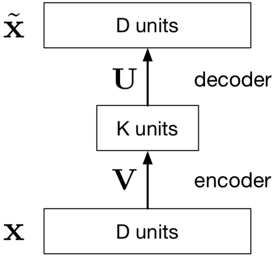
- Map high-dimensional data to two dimensions for visualization
- Compression (i.e. reducing the file size)
  - Note: this requires a VAE, not just an ordinary autoencoder.
- Learn abstract features in an unsupervised way so you can apply them to a supervised task
  - Unlabled data can be much more plentiful than labeled data
- Learn a semantically meaningful representation where you can, e.g., interpolate between different images.

Principal Component Analysis (optional)

- The simplest kind of autoencoder has one hidden layer, linear activations, and squared error loss.

$$\mathcal{L}(\mathrm{x}, \tilde{\mathrm{x}}) = \|\mathrm{x} - \tilde{\mathrm{x}}\|^2$$

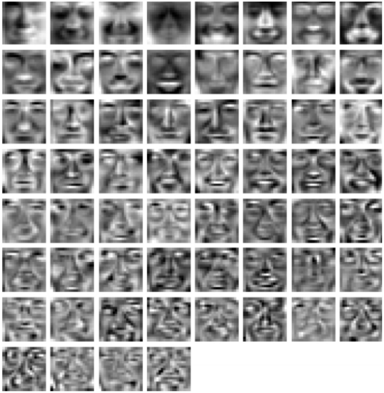
- This network computes $=$ x, which is a linear function.
- If $K \geq D$, we can choose and such that is the identity. This isn't very interesting.



- But suppose $K < D$:
  - maps x to a K-dimensional space, so it's doing dimensionality reduction.
  - The output must lie in a K-dimensional subspace, namely the column space of .

Principal Component Analysis (optional)

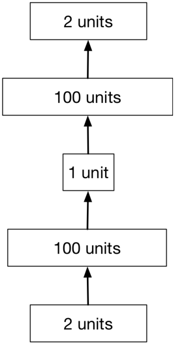
- Review from CSC311: linear autoencoders with squared error loss are equivalent to Principal Component Analysis (PCA).
- Two equivalent formulations:
  - Find the subspace that minimizes the reconstruction error.
  - Find the subspace that maximizes the projected variance.
- The optimal subspace is spanned by the dominant eigenvectors of the empirical covariance matrix.



“Eigenfaces”

Deep Autoencoders

- Deep nonlinear autoencoders learn to project the data, not onto a subspace, but onto a nonlinear manifold
- This manifold is the image of the decoder.
- This is a kind of nonlinear dimensionality reduction.



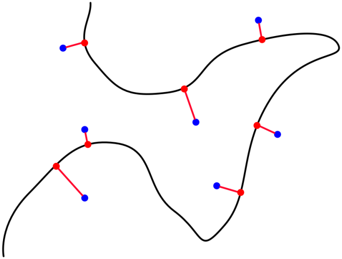
2 units

100 units

1 unit

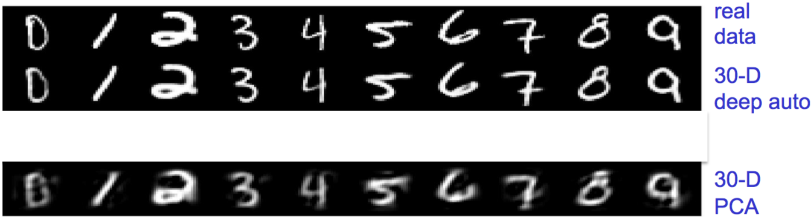
100 units

2 units



Deep Autoencoders

- Nonlinear autoencoders can learn more powerful codes for a given dimensionality, compared with linear autoencoders (PCA)



Deep Autoencoders

- Some limitations of autoencoders
  - They're not generative models, so they don't define a distribution
  - How to choose the latent dimension?

Trade-offs of Generative Approaches

- So far, we have seen four different approaches:
  - Autoregressive models (Lectures 3, 7, and 8)
  - Generative adversarial networks (this lecture)
  - Reversible architectures (this lecture)
  - Variational autoencoders (next lecture)
- They all have their own pro and con. We often pick a method based on our application needs.
- Some considerations for computer vision applications:
  - Do we need to evaluate log likelihood of new data?
  - Do we prefer good samples over evaluation metric?
  - How imporant is representation learning, i.e. meaningful code vectors?
  - How much computational resource can we spent?

Trade-offs of Generative Approaches

- In summary:

	Log-likelihood	Sample	Representation	Computation
Autoregressive				
GANs				
Reversible				
VAEs				

- To be continued...

Trade-offs of Generative Approaches

- In summary:

	Log-likelihood	Sample	Representation	Computation
Autoregressive	**Tractable**	**Good**	Poor	O(#pixels)
GANs	Intractable	**Good**	**Good**	O(#layers)
Reversible	**Tractable**	Poor	Poor	O(#layers)
VAEs				

- To be continued...