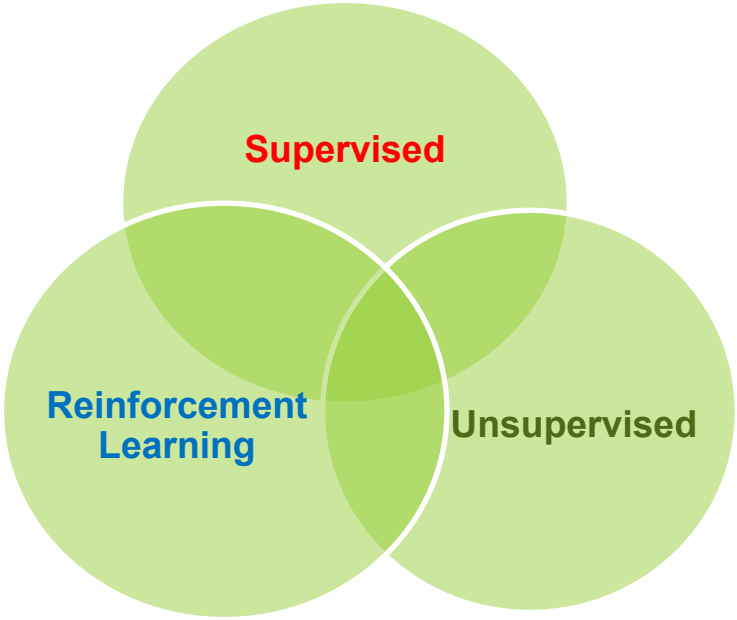
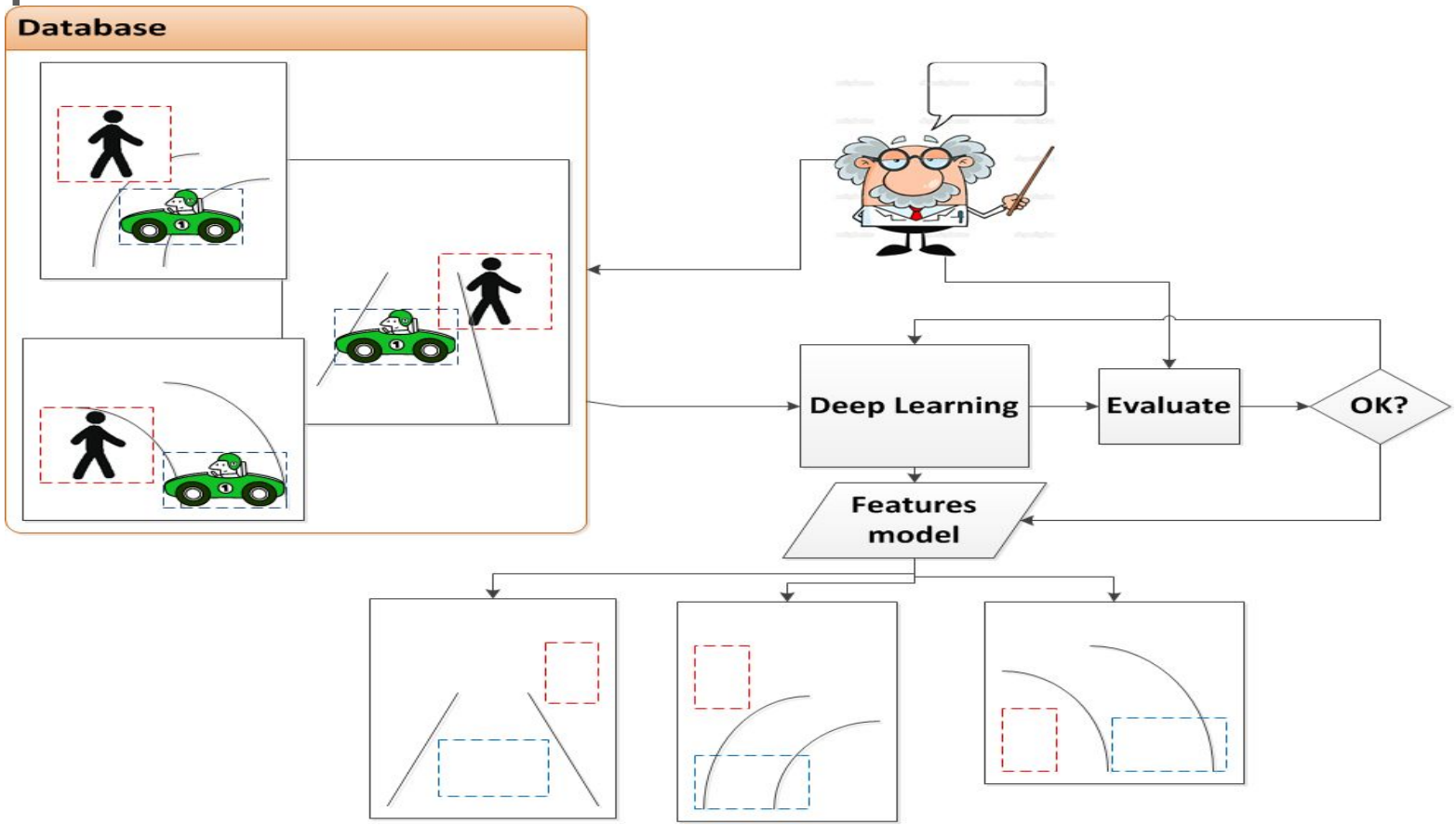
Introduction to Generative Models

Dr. Ahmad El Sallab
Senior Expert

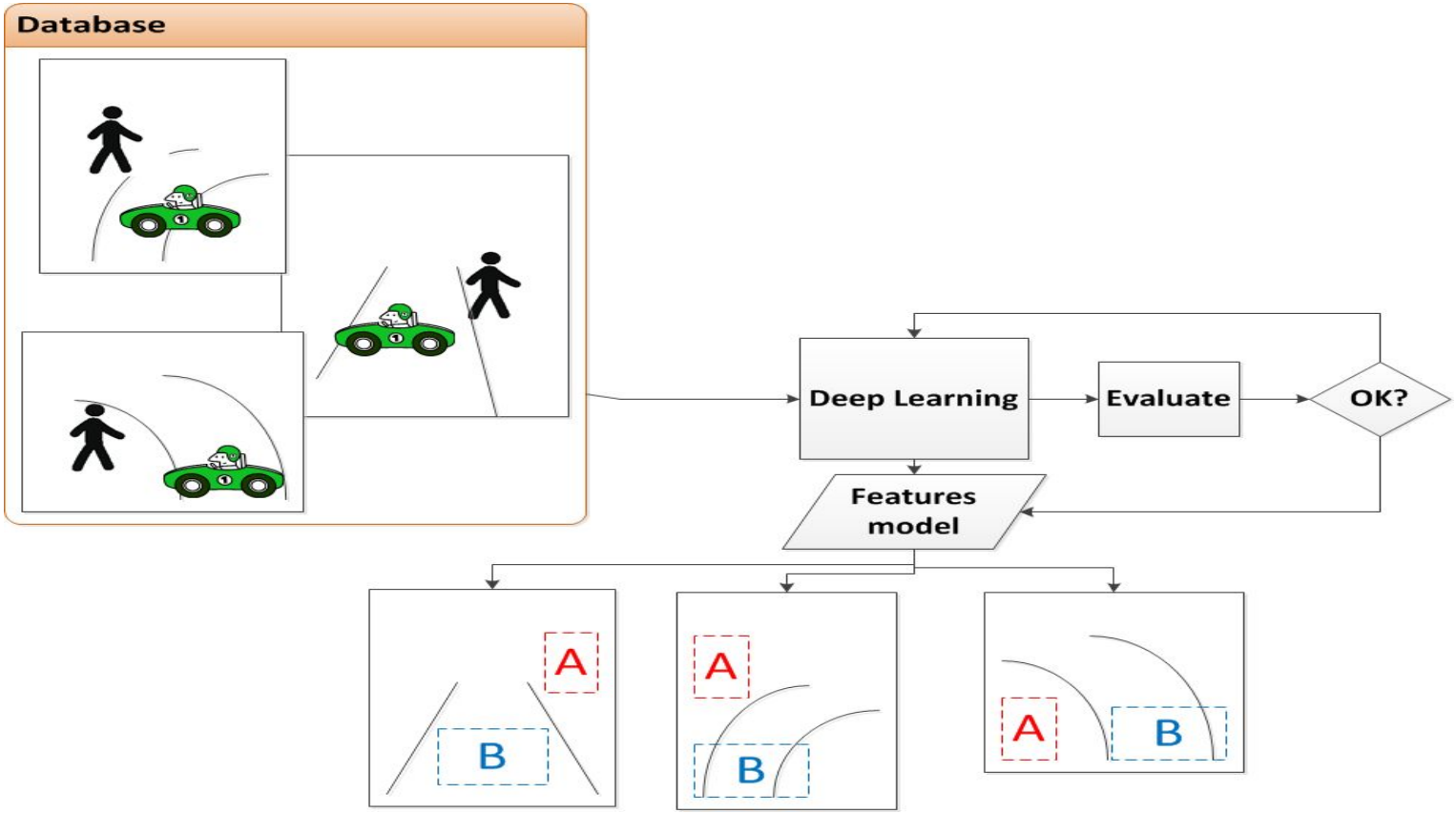
Learning approaches



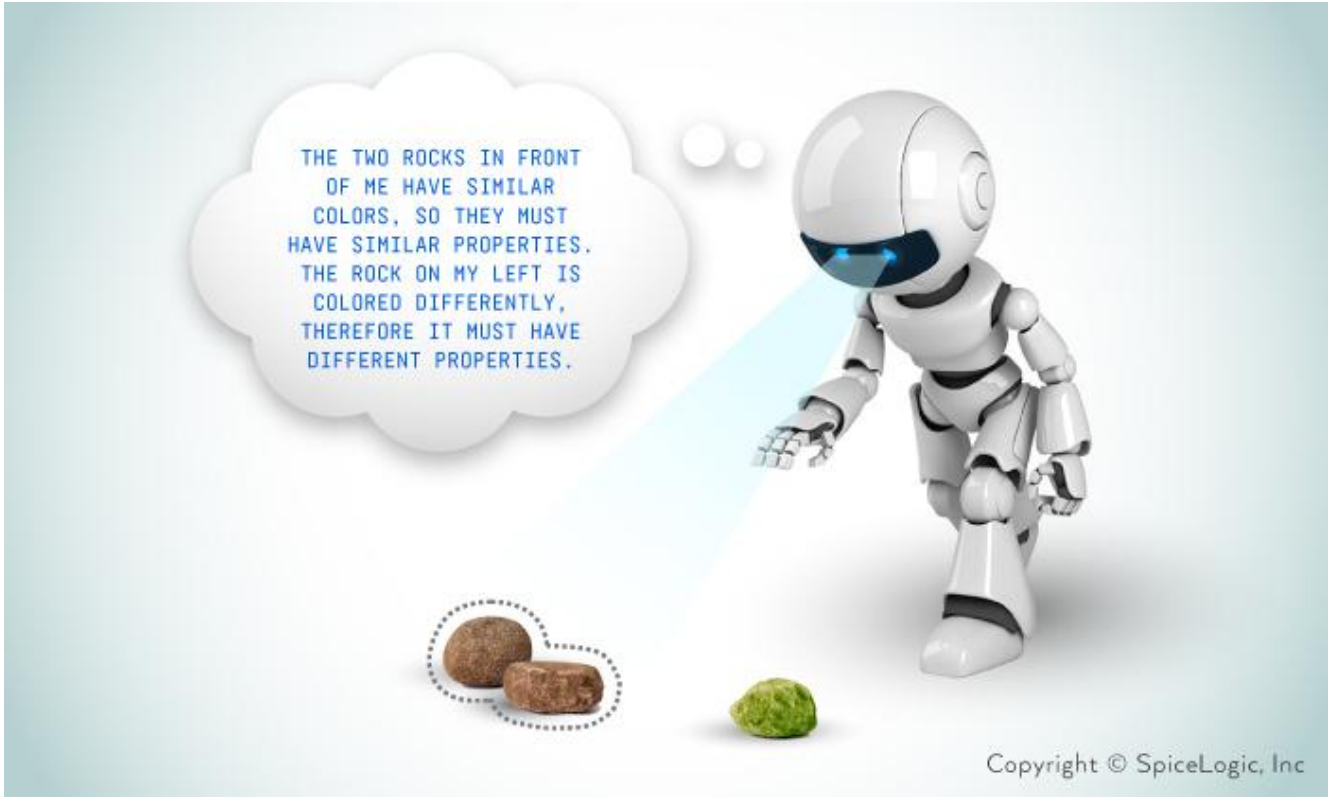
Supervised



Unsupervised from big data

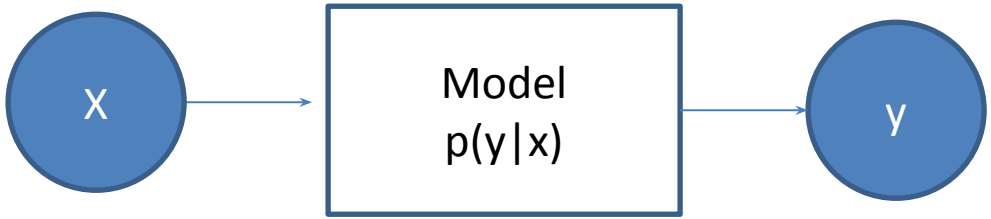


Unsupervised from big data



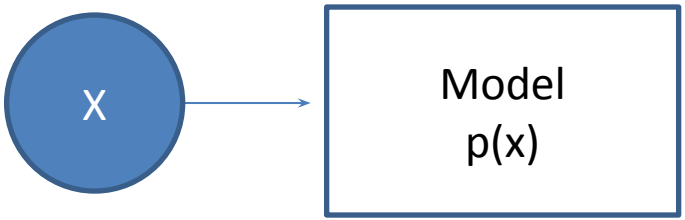
http://www.spicelogic.com/Journal/Perceptron-Artificial-Neural-Networks-10

Supervised vs. Unsupervised models



SL learns about the target

Model the likelihood of the class, given the data

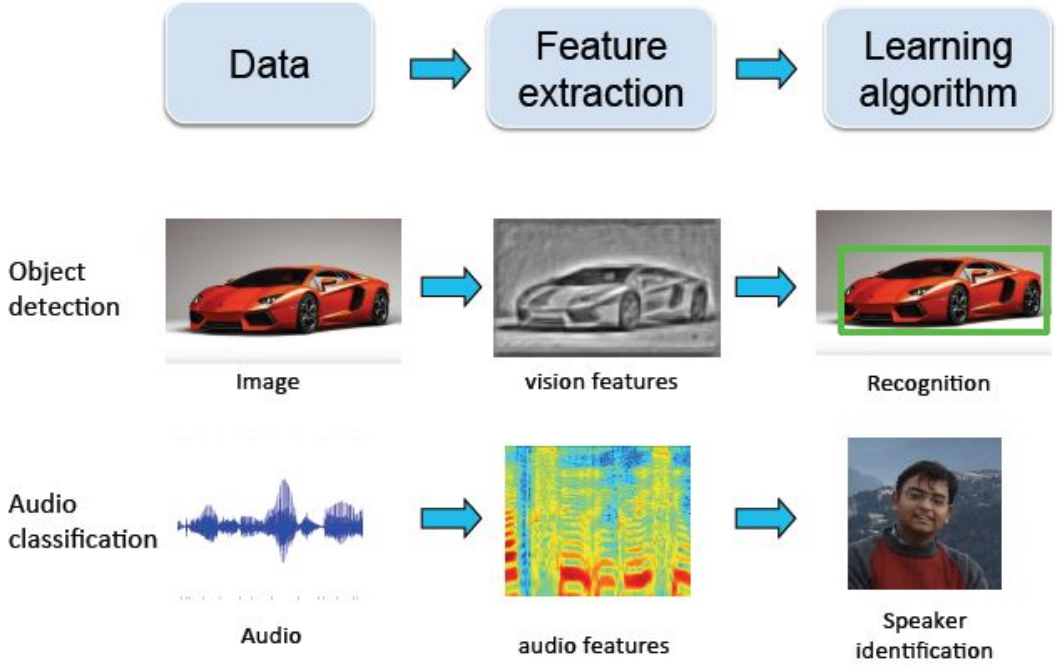


UNSL learns the structure of the data

Model the likelihood of the data

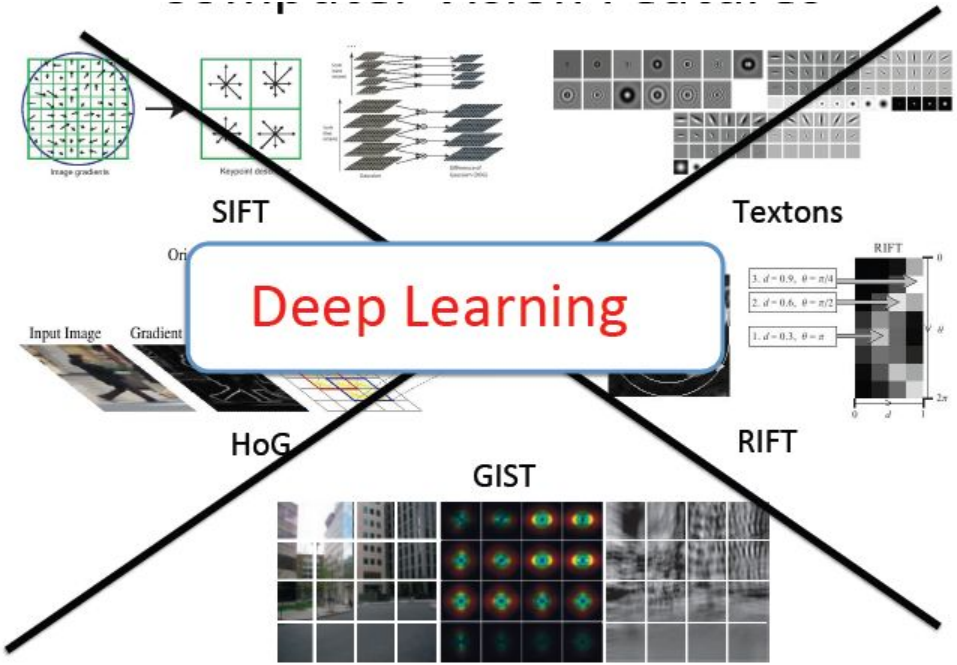
Traditional ML vs. DL

Deep learning summer school, Montreal, 2015: Deep learning 1, Salakhutdinov

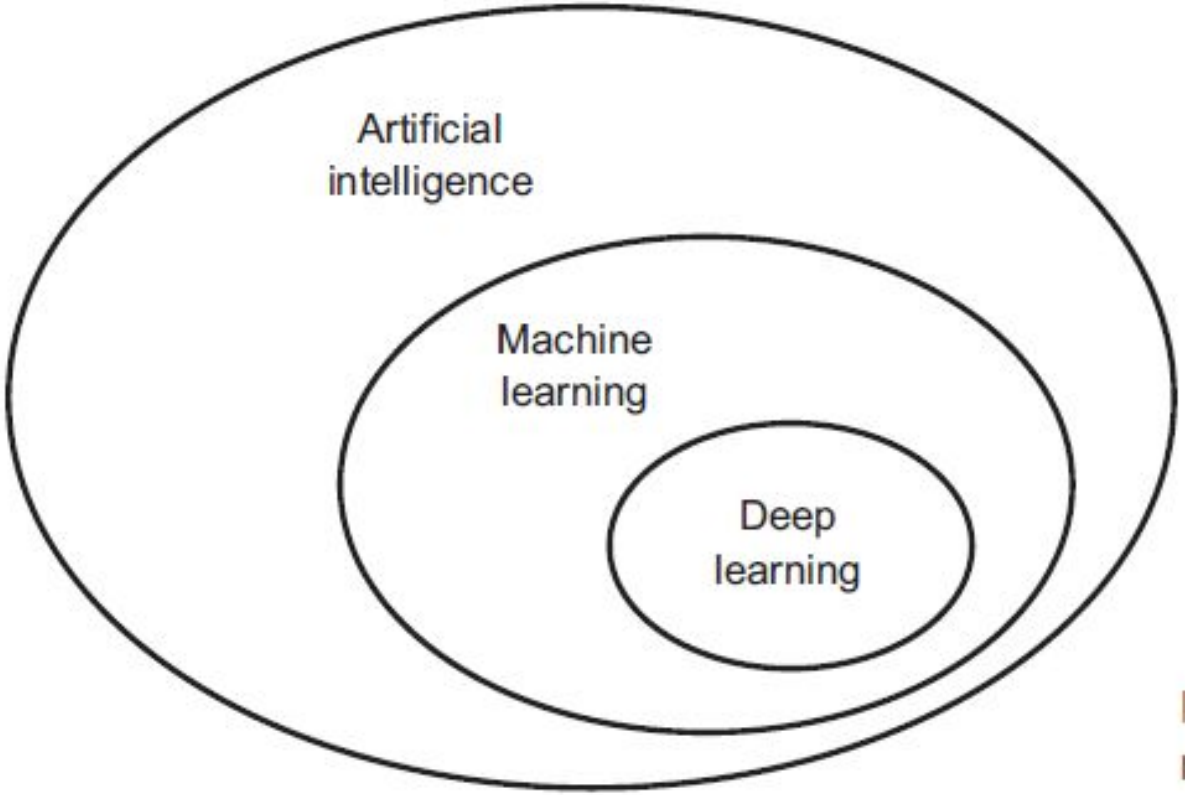


Traditional ML vs. DL

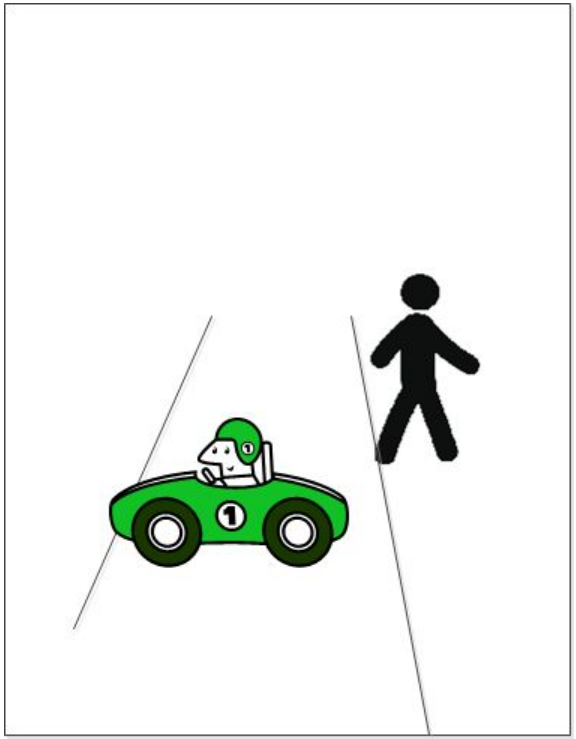
Deep learning summer school, Montreal, 2015: Deep learning: Theoretical motivation



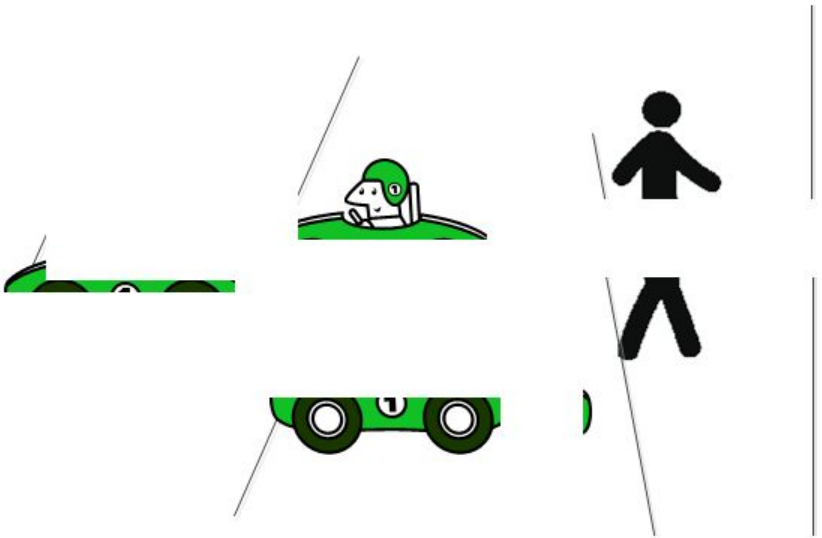
Automatic features extraction



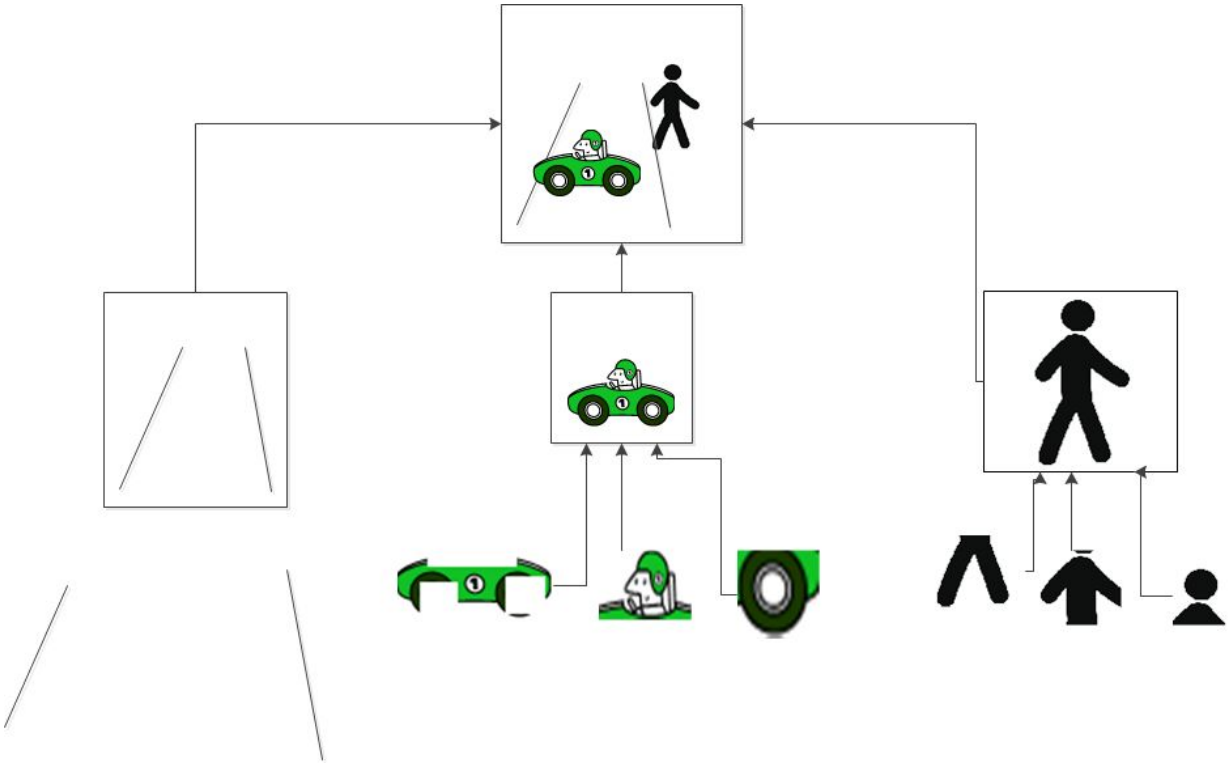
Puzzle Reconstruction



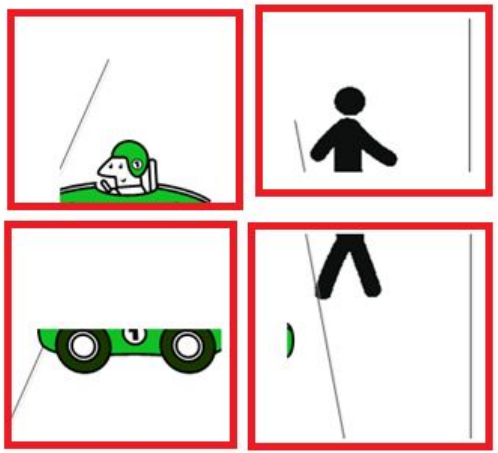
Puzzle Reconstruction



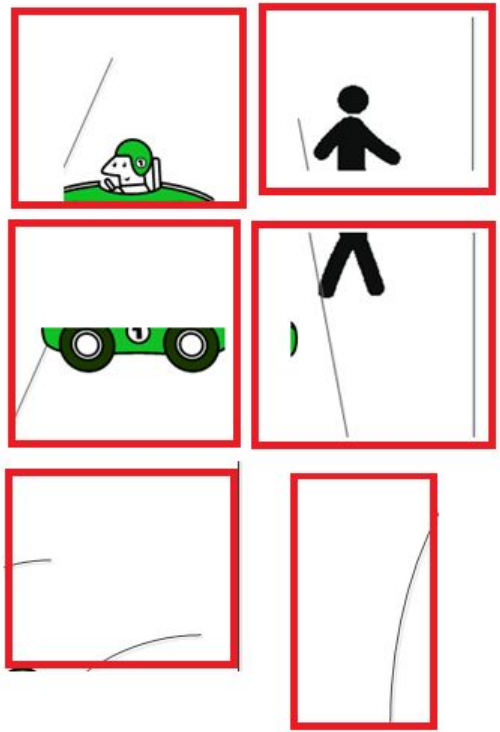
Deconstruction - Reconstruction

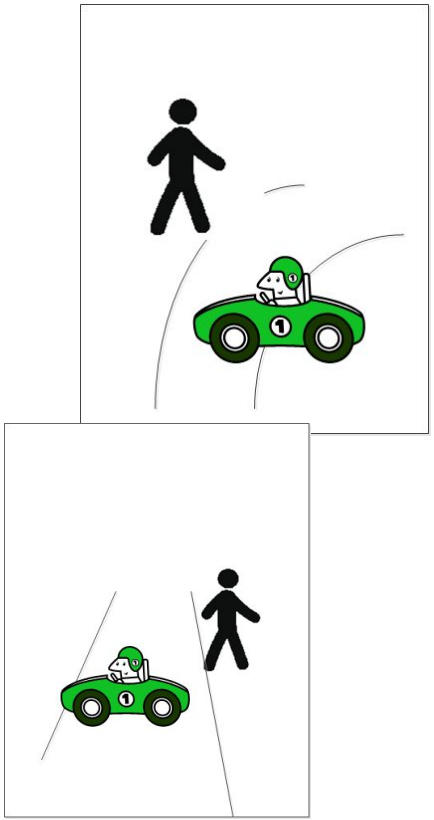


Generic puzzle?

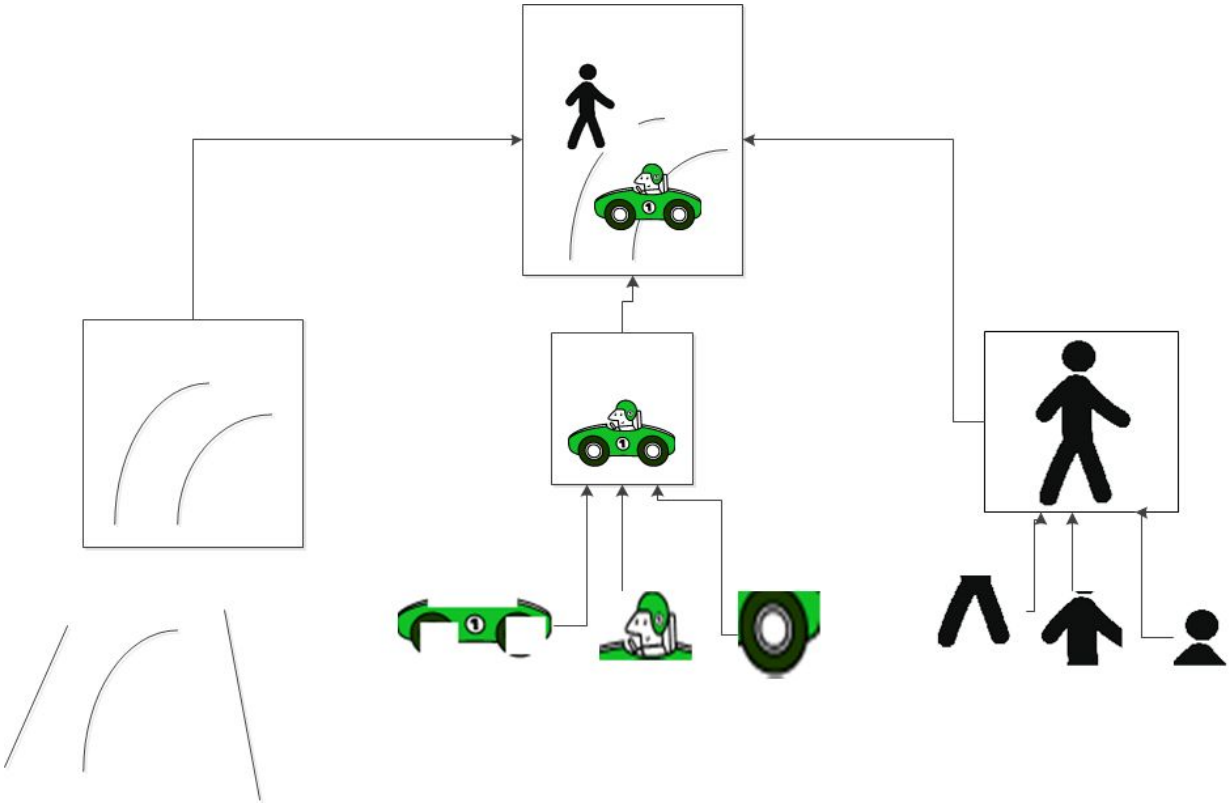


Generic puzzle?

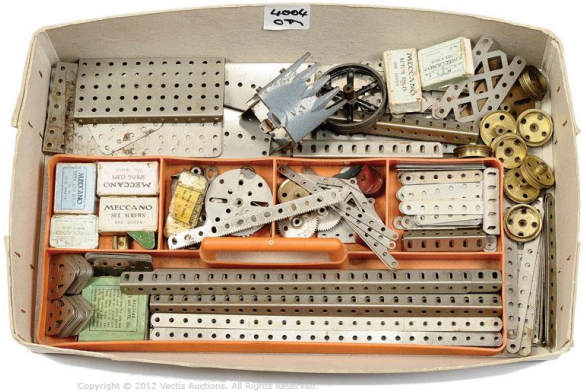


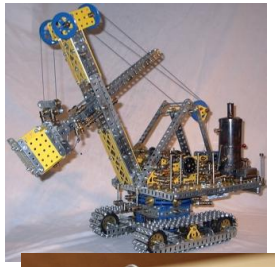


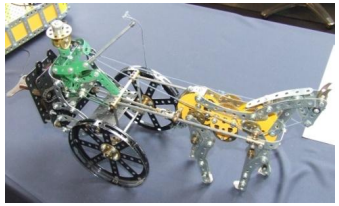
Deconstruction - Reconstruction

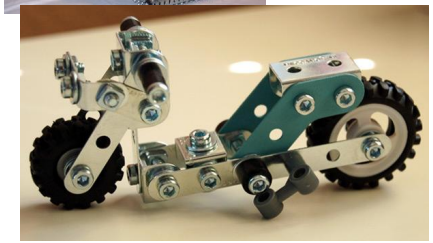


Generic re-usable units



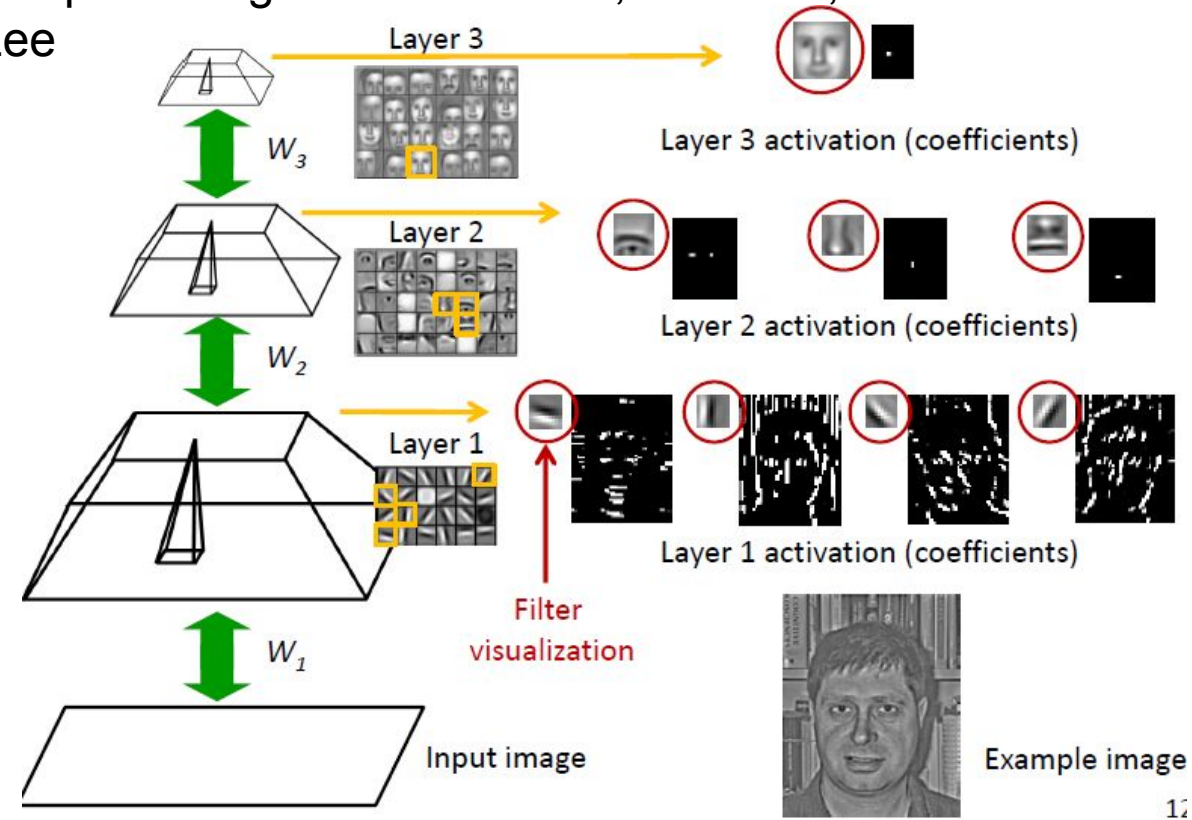






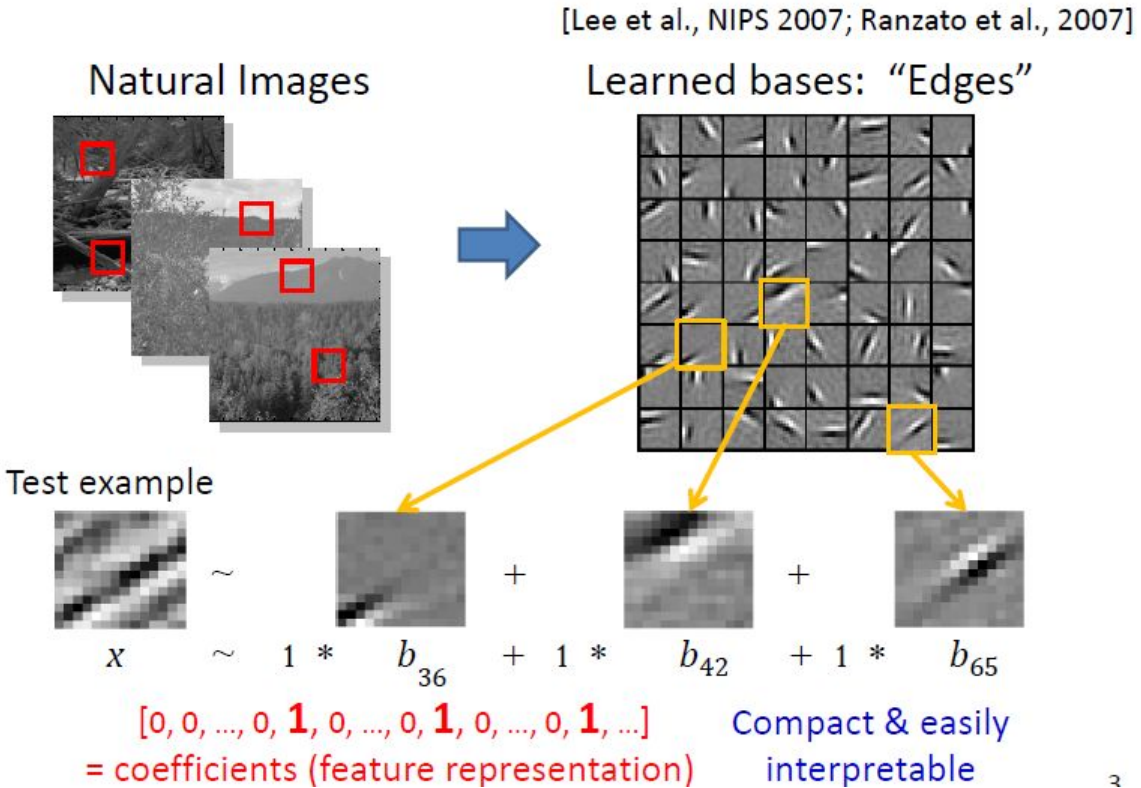
What is meant by “representation”? Eigen Faces

Deep learning summer school, Montreal, 2015: Convolutional NN, Lee



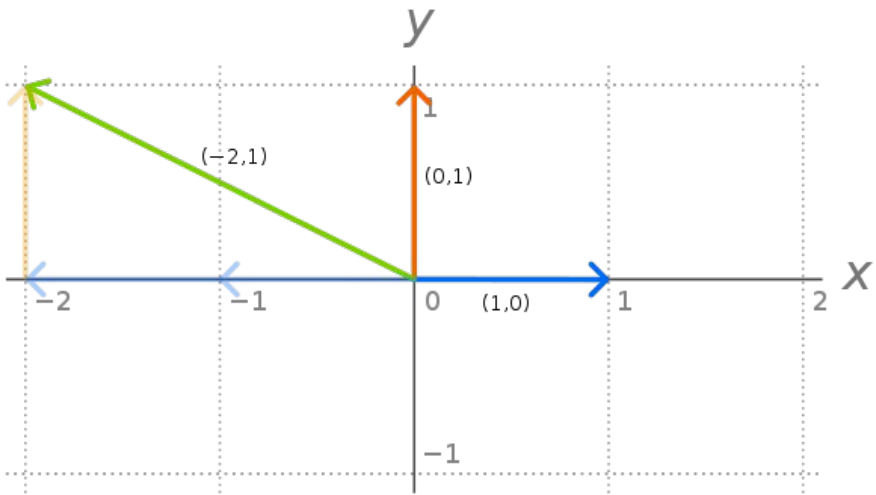
What is meant by “representation”?

Deep learning summer school, Montreal, 2015: Convolutional NN, Lee

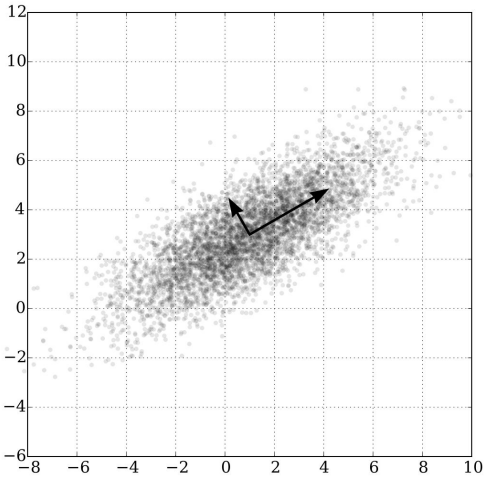


Basis and Eigen vectors

"In mathematics, a set B of elements (vectors) in a vector space V is called a **basis**, if every element of V may be written in a unique way as a (finite) linear combination of elements of B. The coefficients of this linear combination are referred to as **components** or **coordinates** on B of the vector. The elements of a basis are called **basis vectors**." Wikipedia



Eigen vectors = vector components of change
Principal Components = Directions of max change



Features = Weights = Basis vectors

Supervised → Learn W's from x,y → Expensive annotations

Unsupervised → Learn W's from x → No cost of annotation

“To learn computer vision first learn computer graphics”, Geof. Hinton

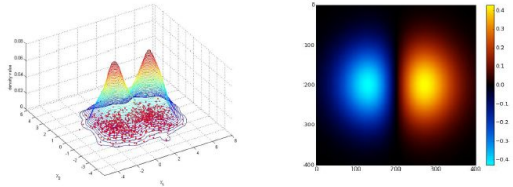
- If you can generate data, you know it’s construction, so better than random initialization

Pre-training = Unsupervised → Supervised

Examples

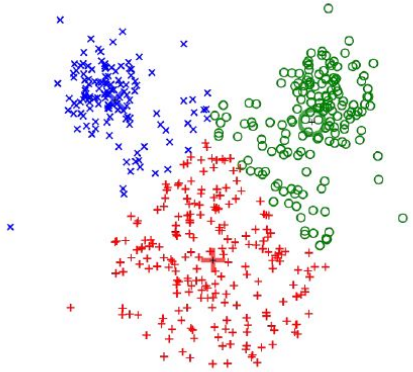
Figure copyright Ian Goodfellow, 2016. Reproduced with permission.

1-d density estimation

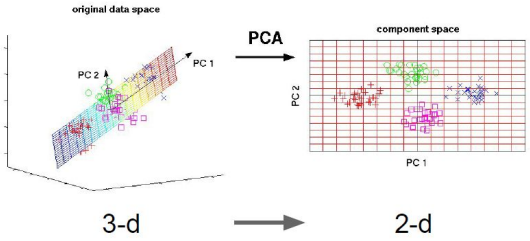


2-d density estimation

2-d density images left and right are CC0 public domain



K-means clustering

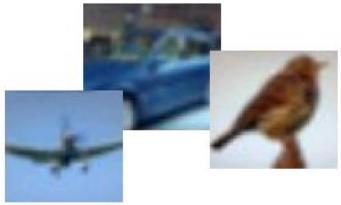


Principal Component Analysis
(Dimensionality reduction)

This image from Matthias Scholz is CC0 public domain

Generative Models

Given training data, generate new samples from same distribution

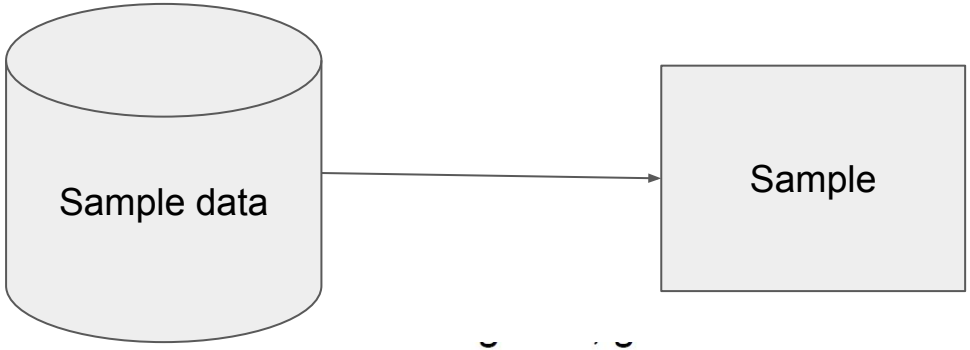


Training data ~ $p_{data}(x)$

Generated samples ~ $p_{model}(x)$

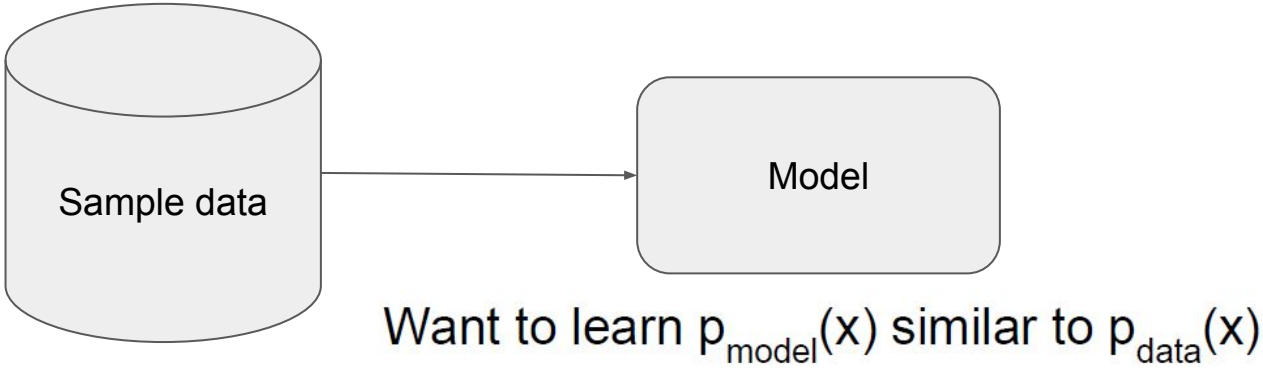
Want to learn $p_{model}(x)$ similar to $p_{data}(x)$

Generative models



Training data ~ $p_{data}(x)$

Generative models



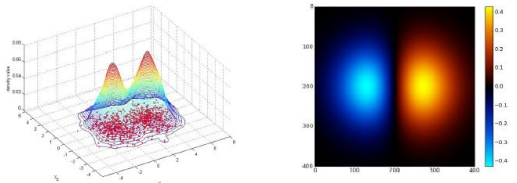
Want to learn $p_{model}(x)$ similar to $p_{data}(x)$

Generated samples ~ $p_{model}(x)$

How to learn $p_{model}(x)$?

Explicit → Model family hypothesis (say Gaussian) → Parametric model of pmodel, and learn the parameters from data→ Density estimation

1-d density estimation



2-d density estimation

How to learn $p_{model}(x)$?

Explicit → Model family hypothesis

Implicit → learn model that can sample from pmodel(x) w/o explicitly defining/optimizing for it

Explicit vs. Implicit

Explicit:

- Loss is directly maximizing likelihood of generative the data → pdata(x)
- Example: Auto-Encoders, VAE, Pixel-RNN, Pixel-CNN

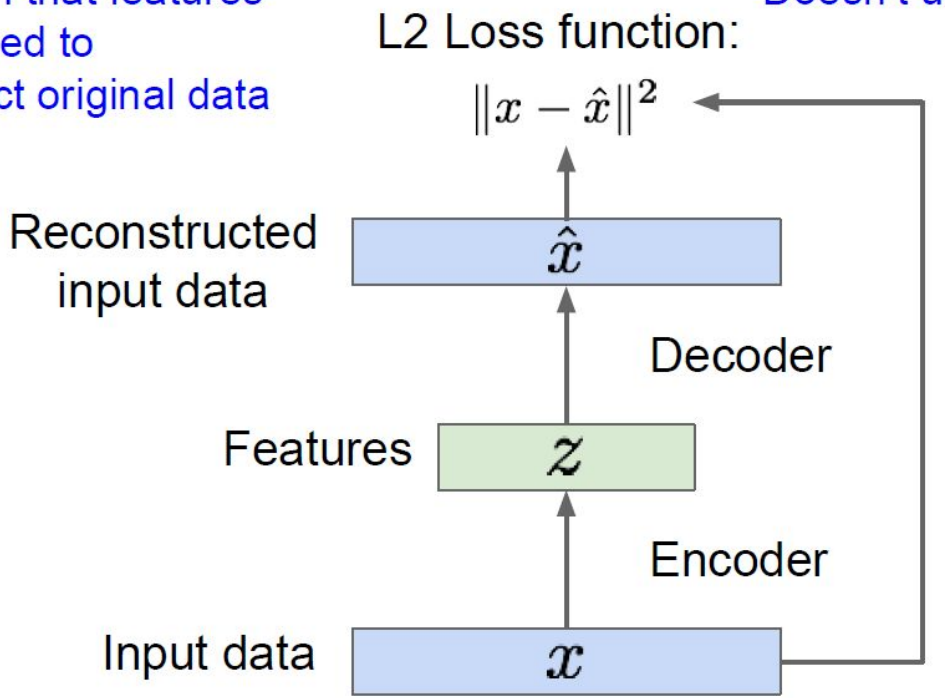
Implicit

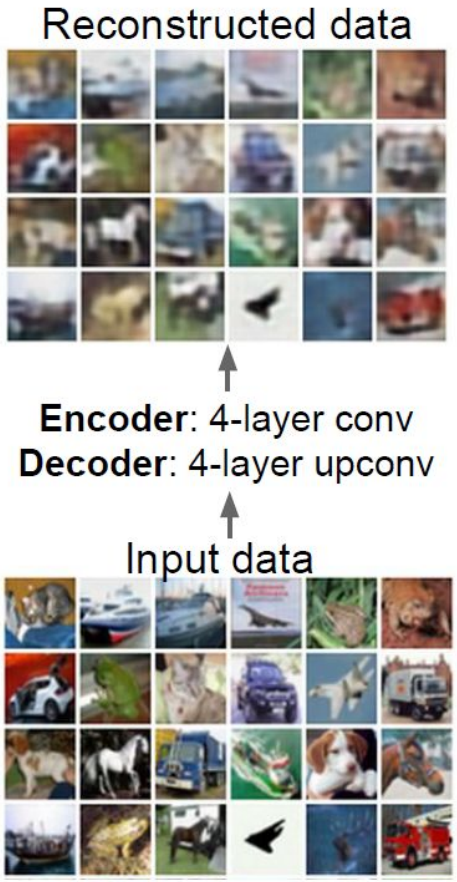
- Loss is not maximizing pdata(x), but something else!
- Example: **GANs**

Some background first: Autoencoders

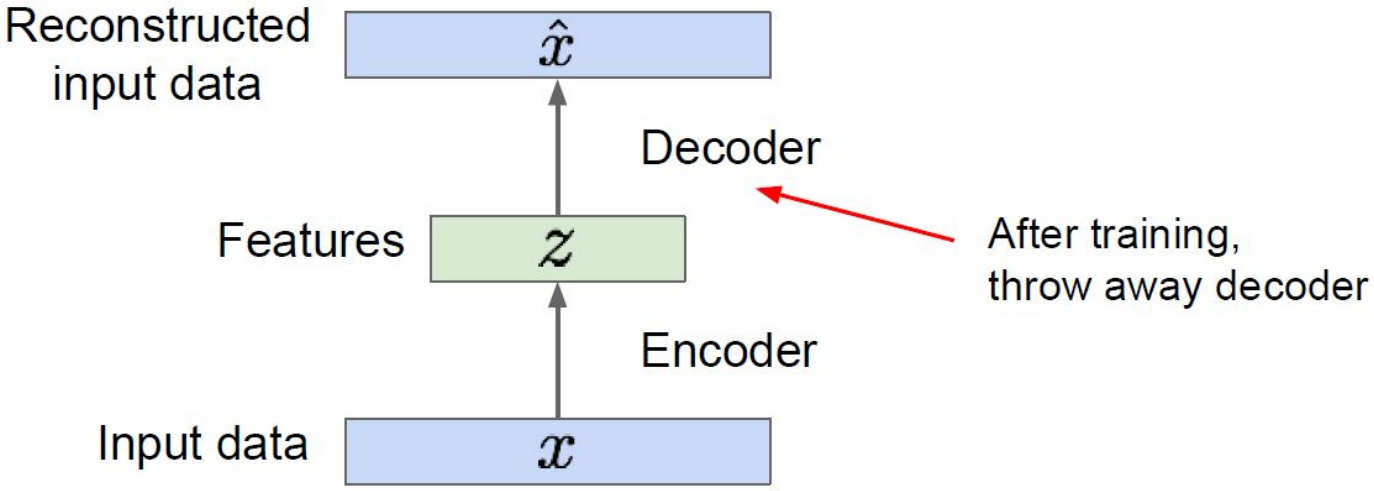
Train such that features can be used to reconstruct original data

Doesn't use labels!



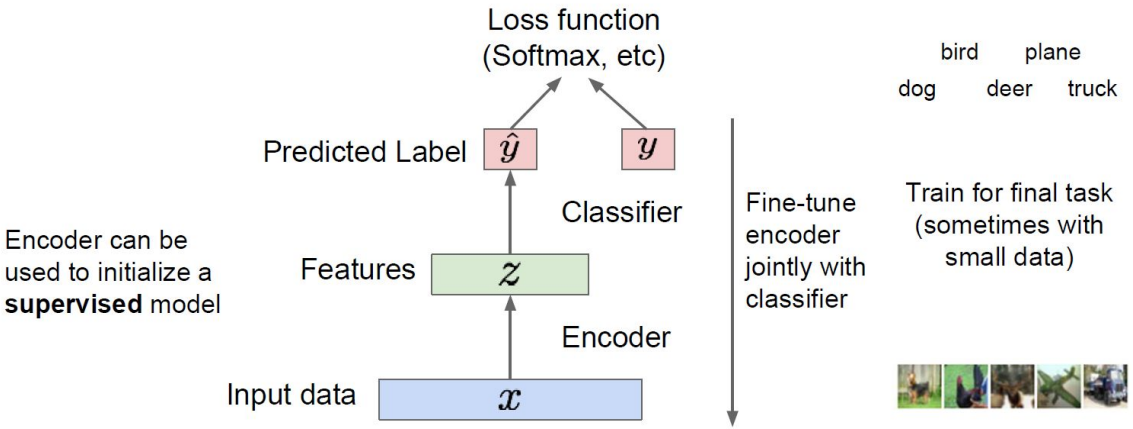


Some background first: Autoencoders



"To learn computer vision first learn computer graphics", Geof. Hinton

- If you can generate data, you know it's construction, so better than random initialization



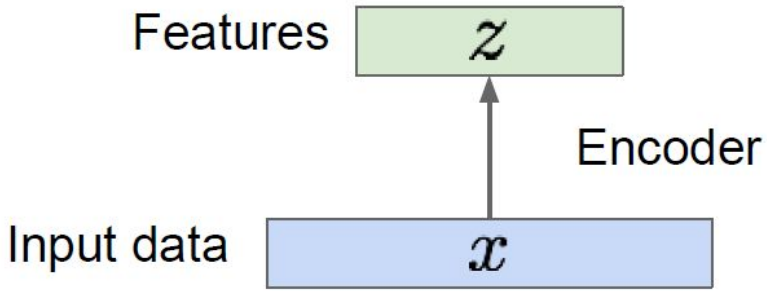
Pre-training = Unsupervised → Supervised

But which category of data we generate?

Z is called the **Latent factor** or **Embedding**

→ Usually much lowe dimension, so called code

Z cannot belong to one uniform range of numbers!



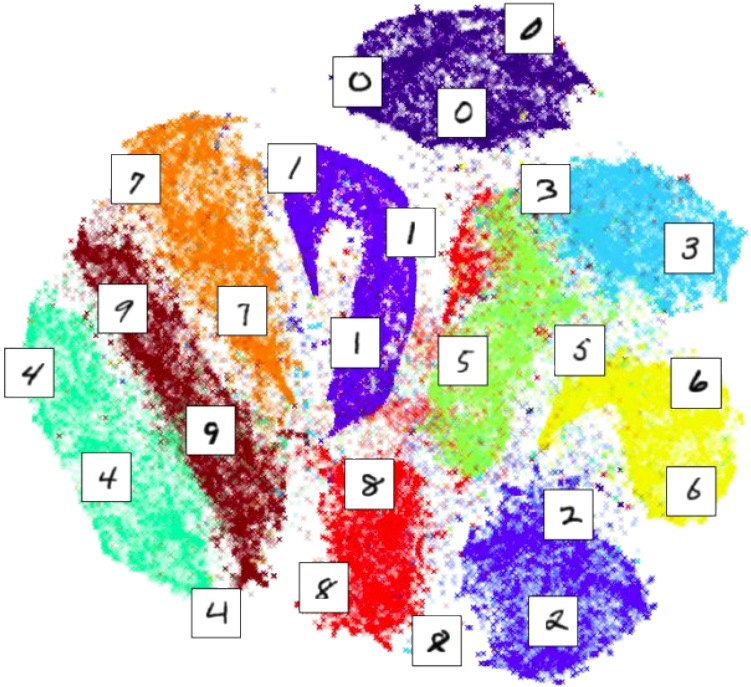
Embeddings and Latent codes

MNIST + tSNE

Points represent values of the Embedding vector Z, annotated with its class

It is clear, the ranges of z are clustered by the categories

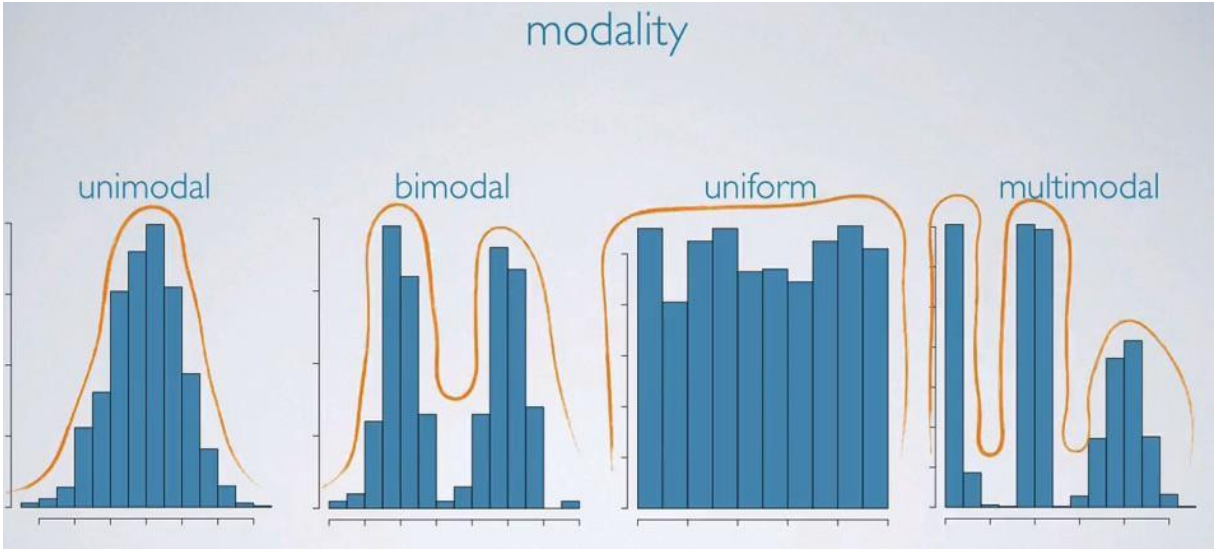
It is still unsupervised



Multi-modal distribution

The model learns p(x) → Cannot be uni-modal

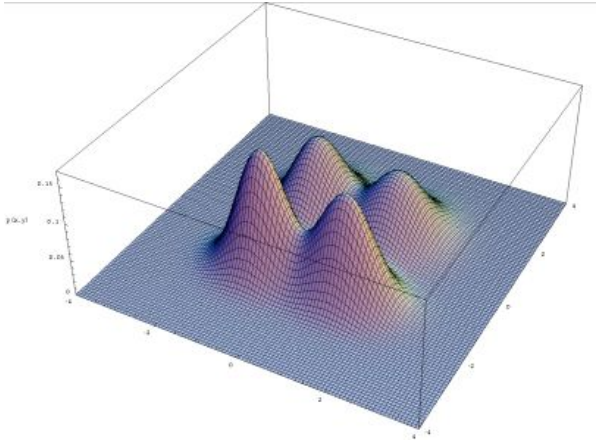
Modes → data categories

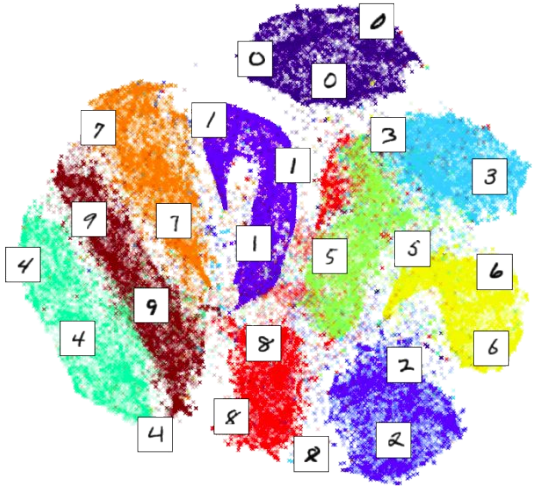


Multi-modal distribution

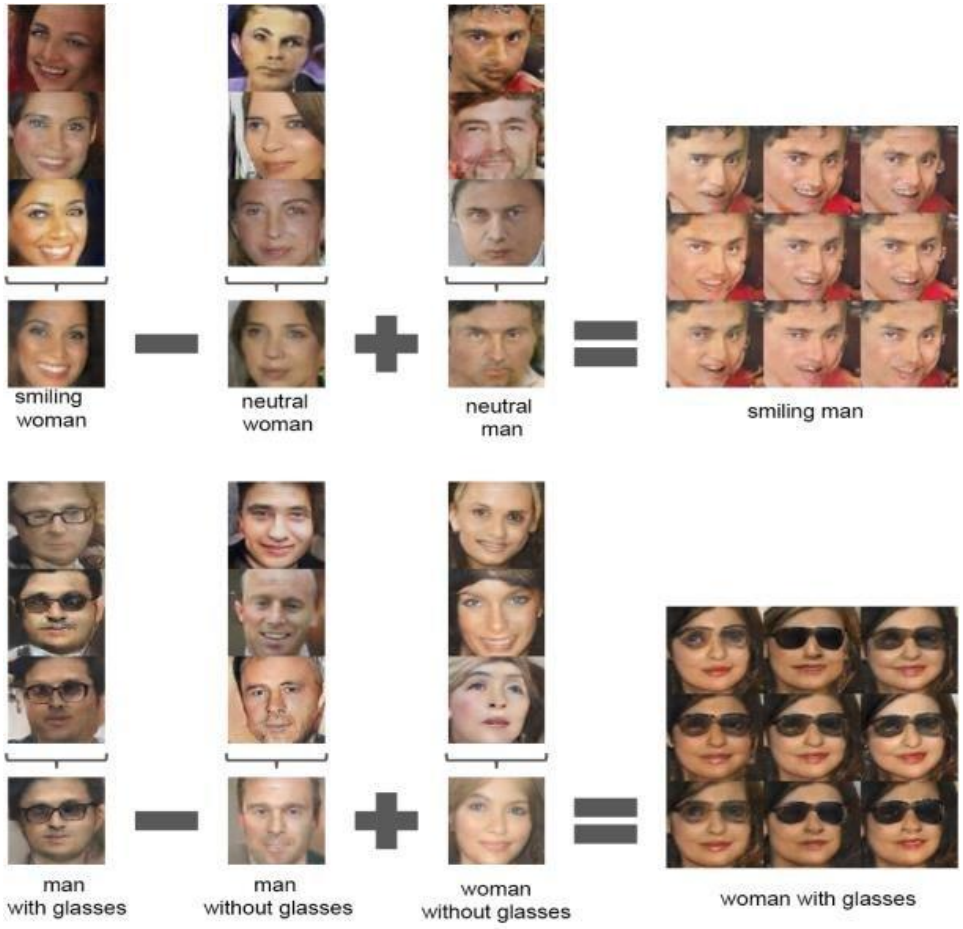
The model learns p(x) → Cannot be uni-modal

Modes → data categories or attributes (orientation, gender, skin color,...etc)





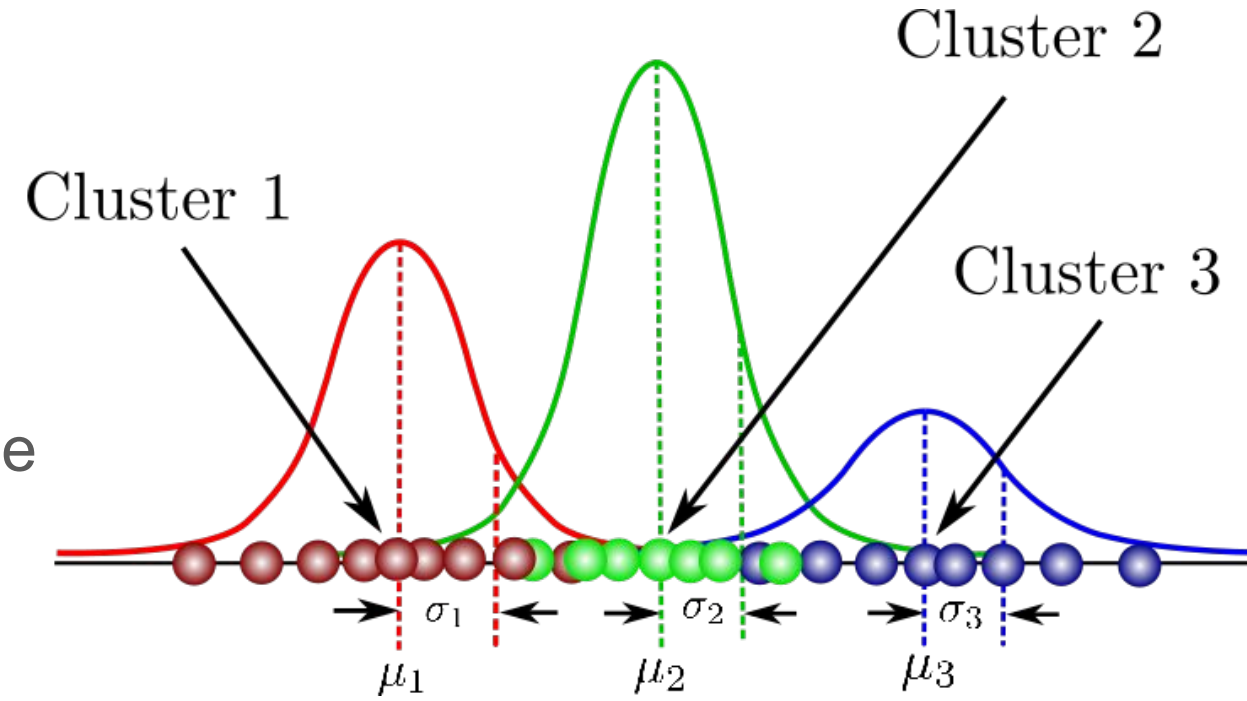
More on modalities and latent factors



How to force multi-modality?

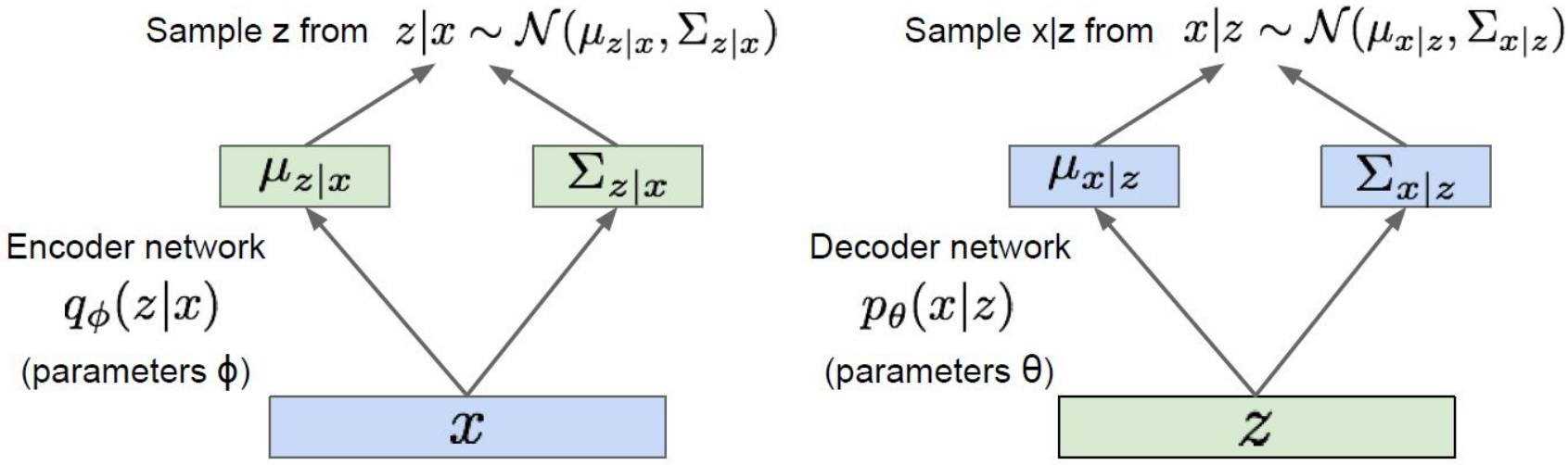
Let's assume the data is coming from Gaussian distribution. And also the latent code.

Not only that, we let the Gaussian parameters be dependent on the data itself→ Enable Mixture of Gaussians



Variational Autoencoders

Since we're modeling probabilistic generation of data, encoder and decoder networks are probabilistic



Encoder and decoder networks also called "recognition"/"inference" and "generation" networks

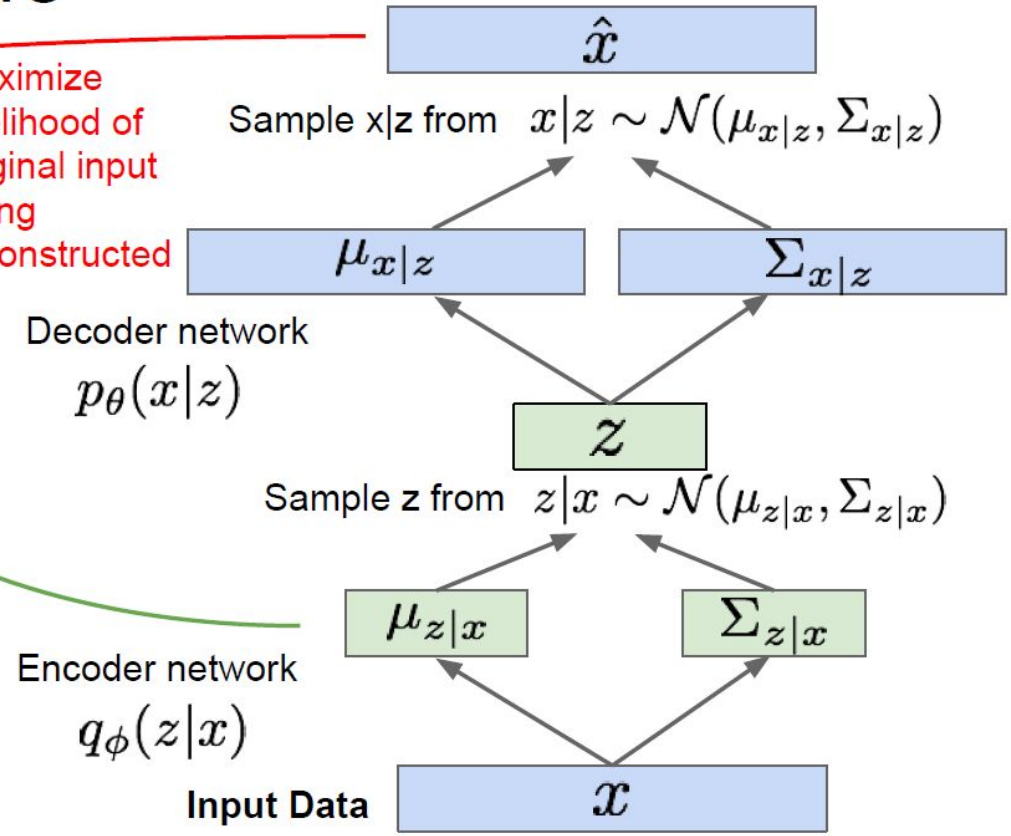
Kingma and Welling, "Auto-Encoding Variational Bayes", ICLR 2014

Variational Autoencoders

Putting it all together: maximizing the likelihood lower bound

$$\underbrace{\mathbf{E}_z\left[\log p_\theta(x^{(i)} \mid z)\right] - D_{KL}(q_\phi(z \mid x^{(i)}) \,||\, p_\theta(z))}_{\mathcal{L}(x^{(i)}, \theta, \phi)}$$

Maximize likelihood of original input being reconstructed

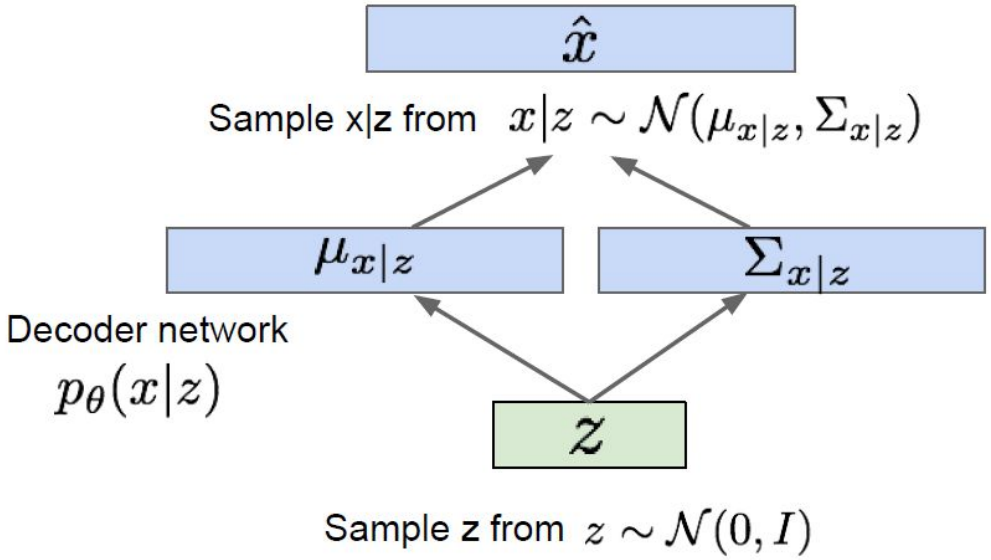


Make approximate posterior distribution close to prior

For every minibatch of input data: compute this forward pass, and then backprop!

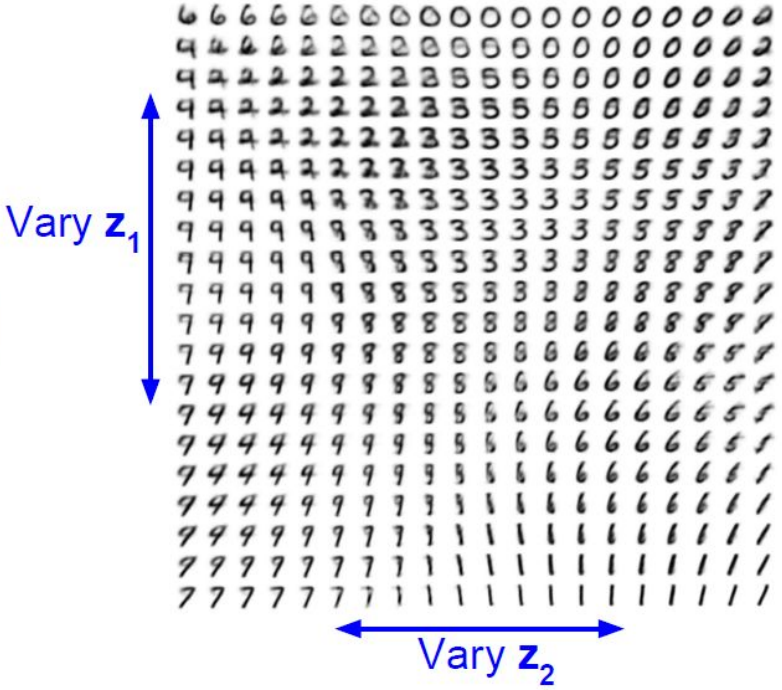
Variational Autoencoders: Generating Data!

Use decoder network. Now sample z from prior!



Kingma and Welling, "Auto-Encoding Variational Bayes", ICLR 2014

Data manifold for 2-d **z**



Variational Autoencoders: Generating Data!

Diagonal prior on **z**
=> independent
latent variables

Different
dimensions of **z**
encode
interpretable factors
of variation

Also good feature representation that can be computed using $q_\phi(z|x)$!

Kingma and Welling, "Auto-Encoding Variational Bayes", ICLR 2014

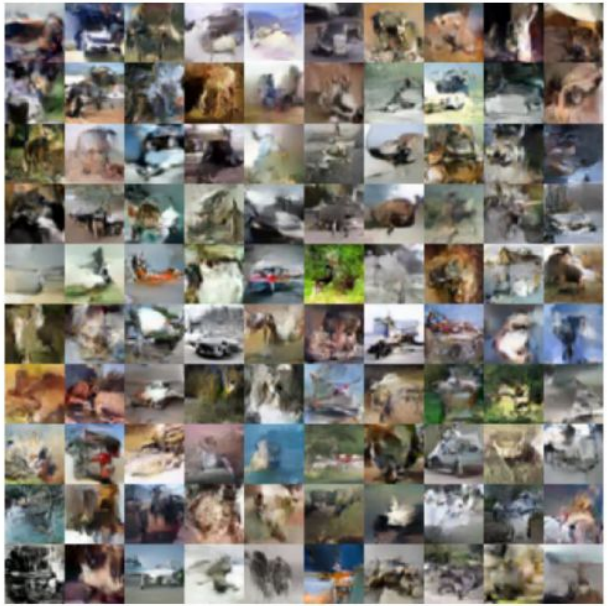
Interpreting latent factors

Z1, z2,... → Modes → Latent factors → attributes

We don't explicitly force them → They emerge (we just set how many of them) → We try to "guess" what they represent

Latent factors are "good features" → automatically extracted → For unsupervised learning

Variational Autoencoders: Generating Data!



32x32 CIFAR-10

Labeled Faces in the Wild

Pros and Cons of VAE

- Explicit density estimation
- Attributes/modes enables as part of model capacity
- Well formulated!

- Low quality of generated images!

Let’s code!

AE and VAE

Generative Adversarial Networks (GANs)

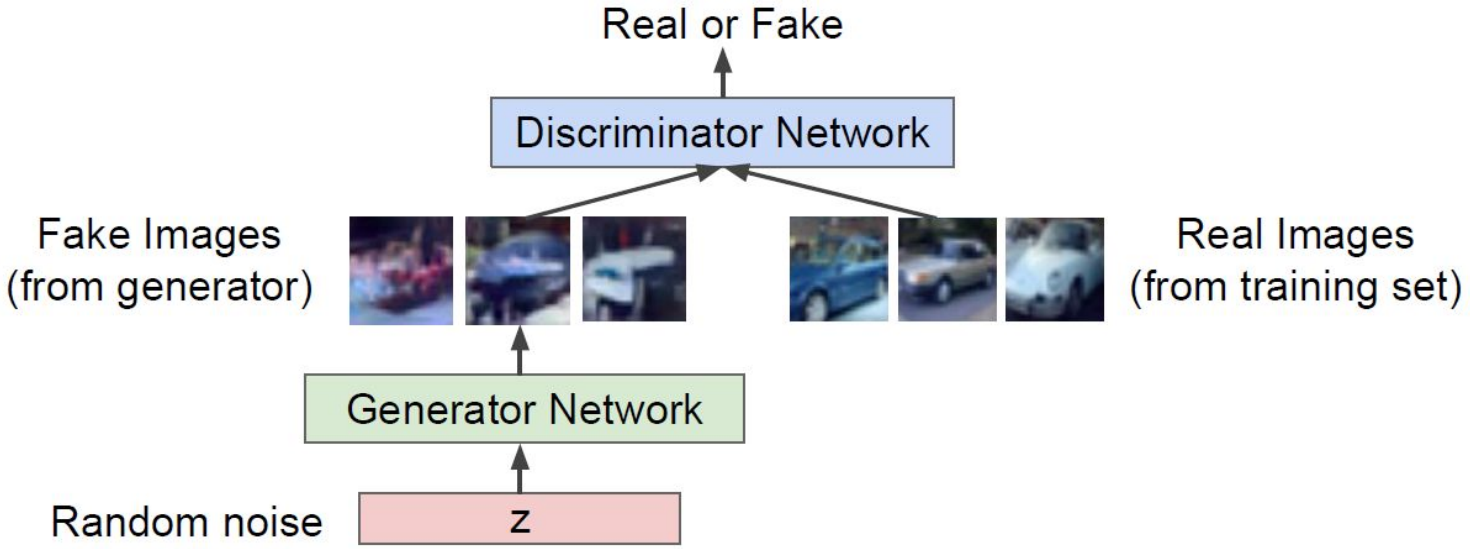
What if we give up on explicitly modeling density, and just want ability to sample?

GANs: don't work with any explicit density function!
Instead, take game-theoretic approach: learn to generate from training distribution through 2-player game

Training GANs: Two-player game

Ian Goodfellow et al., "Generative Adversarial Nets", NIPS 2014

Generator network: try to fool the discriminator by generating real-looking images
Discriminator network: try to distinguish between real and fake images



Training GANs: Two-player game

Ian Goodfellow et al., "Generative Adversarial Nets", NIPS 2014

Generator network: try to fool the discriminator by generating real-looking images
Discriminator network: try to distinguish between real and fake images

Train jointly in **minimax game**

Discriminator outputs likelihood in (0,1) of real image

Minimax objective function:

$$\min_{\theta_g} \max_{\theta_d} \left[\mathbb{E}_{x \sim p_{data}} \log \underbrace{D_{\theta_d}(x)}_{\text{Discriminator output for real data x}} + \mathbb{E}_{z \sim p(z)} \log(1 - \underbrace{D_{\theta_d}(G_{\theta_g}(z))}_{\text{Discriminator output for generated fake data G(z)}}) \right]$$

- Discriminator (θ_d) wants to **maximize objective** such that D(x) is close to 1 (real) and D(G(z)) is close to 0 (fake)
- Generator (θ_g) wants to **minimize objective** such that D(G(z)) is close to 1 (discriminator is fooled into thinking generated G(z) is real)

Implicit!

The loss does not try in anyway to maximize the data likelihood pdata(x)!

Training GANs: Two-player game

Ian Goodfellow et al., "Generative Adversarial Nets", NIPS 2014

Minimax objective function:

$$\min_{\theta_g} \max_{\theta_d} \left[\mathbb{E}_{x \sim p_{data}} \log D_{\theta_d}(x) + \mathbb{E}_{z \sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z))) \right]$$

Alternate between:

1. **Gradient ascent** on discriminator

$$\max_{\theta_d} \left[\mathbb{E}_{x \sim p_{data}} \log D_{\theta_d}(x) + \mathbb{E}_{z \sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z))) \right]$$

Warm up discriminator first on random points (or weak generator)

2. **Gradient descent** on generator

$$\min_{\theta_g} \mathbb{E}_{z \sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z)))$$

Now improve the generator

Training GANs: Two-player game

Ian Goodfellow et al., "Generative Adversarial Nets", NIPS 2014

Minimax objective function:

$$\min_{\theta_g} \max_{\theta_d} \left[\mathbb{E}_{x \sim p_{data}} \log D_{\theta_d}(x) + \mathbb{E}_{z \sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z))) \right]$$

Alternate between:

1. **Gradient ascent** on discriminator

$$\max_{\theta_d} \left[\mathbb{E}_{x \sim p_{data}} \log D_{\theta_d}(x) + \mathbb{E}_{z \sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z))) \right]$$

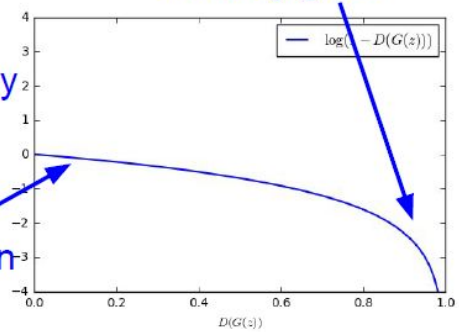
2. **Gradient descent** on generator

$$\min_{\theta_g} \mathbb{E}_{z \sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z)))$$

In practice, optimizing this generator objective does not work well!

Gradient signal dominated by region where sample is already good

When sample is likely fake, want to learn from it to improve generator. But gradient in this region is relatively flat!



Training GANs: Two-player game

Ian Goodfellow et al., "Generative Adversarial Nets", NIPS 2014

Minimax objective function:

$$\min_{\theta_g} \max_{\theta_d} \left[\mathbb{E}_{x \sim p_{data}} \log D_{\theta_d}(x) + \mathbb{E}_{z \sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z))) \right]$$

Alternate between:

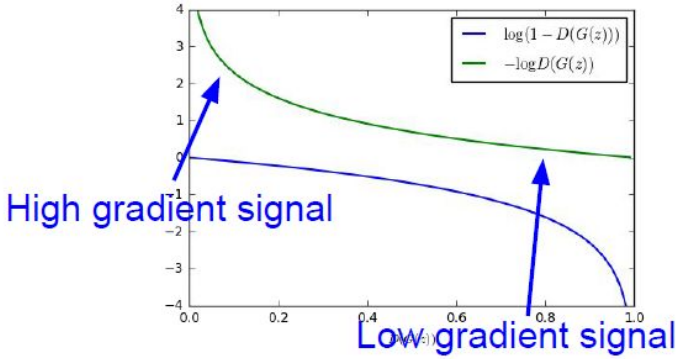
1. **Gradient ascent** on discriminator

$$\max_{\theta_d} \left[\mathbb{E}_{x \sim p_{data}} \log D_{\theta_d}(x) + \mathbb{E}_{z \sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z))) \right]$$

2. **Instead: Gradient ascent** on generator, different objective

$$\max_{\theta_g} \mathbb{E}_{z \sim p(z)} \log(D_{\theta_d}(G_{\theta_g}(z)))$$

Instead of minimizing likelihood of discriminator being correct, now maximize likelihood of discriminator being wrong.
Same objective of fooling discriminator, but now higher gradient signal for bad samples => works much better! Standard in practice.



Training GANs: Two-player game

Ian Goodfellow et al., "Generative Adversarial Nets", NIPS 2014

Putting it together: GAN training algorithm

Some find k=1 more stable, others use k > 1, no best rule.

Recent work (e.g. Wasserstein GAN) alleviates this problem, better stability!

for number of training iterations **do**

  for k steps **do**

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

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D_{\theta_d}(x^{(i)}) + \log(1 - D_{\theta_d}(G_{\theta_g}(z^{(i)}))) \right]$$

  end for

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

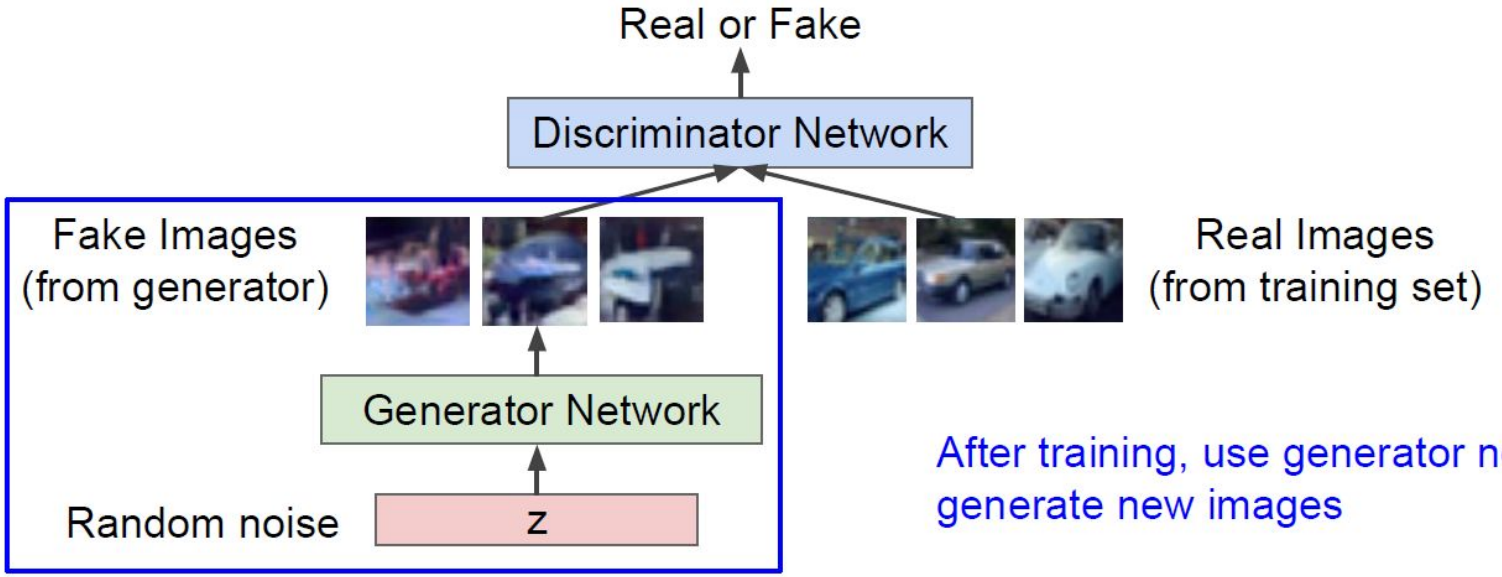
$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log(D_{\theta_d}(G_{\theta_g}(z^{(i)})))$$

end for

Training GANs: Two-player game

Ian Goodfellow et al., "Generative Adversarial Nets", NIPS 2014

Generator network: try to fool the discriminator by generating real-looking images
Discriminator network: try to distinguish between real and fake images



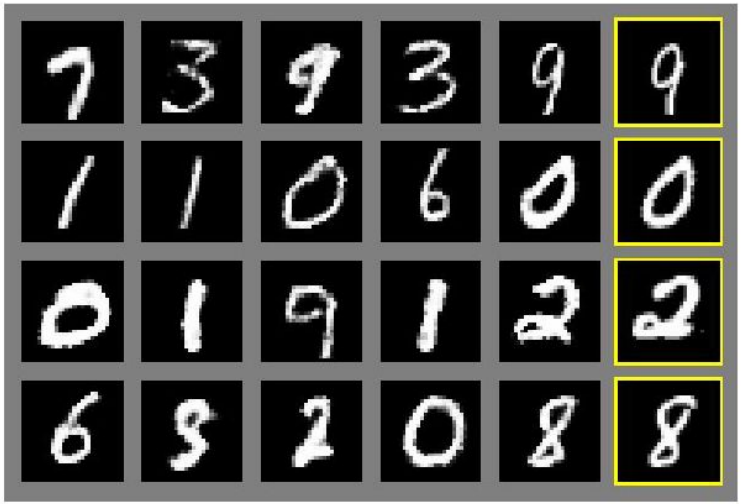
After training, use generator network to generate new images

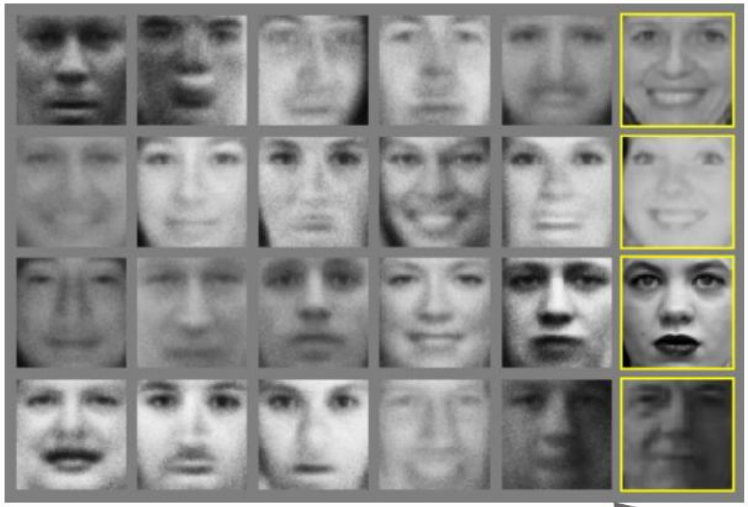
Fake and real images copyright Emily Denton et al. 2015. Reproduced with permission.

Ian Goodfellow et al., "Generative Adversarial Nets", NIPS 2014

Generative Adversarial Nets

Generated samples





Nearest neighbor from training set

Generative Adversarial Nets: Convolutional Architectures

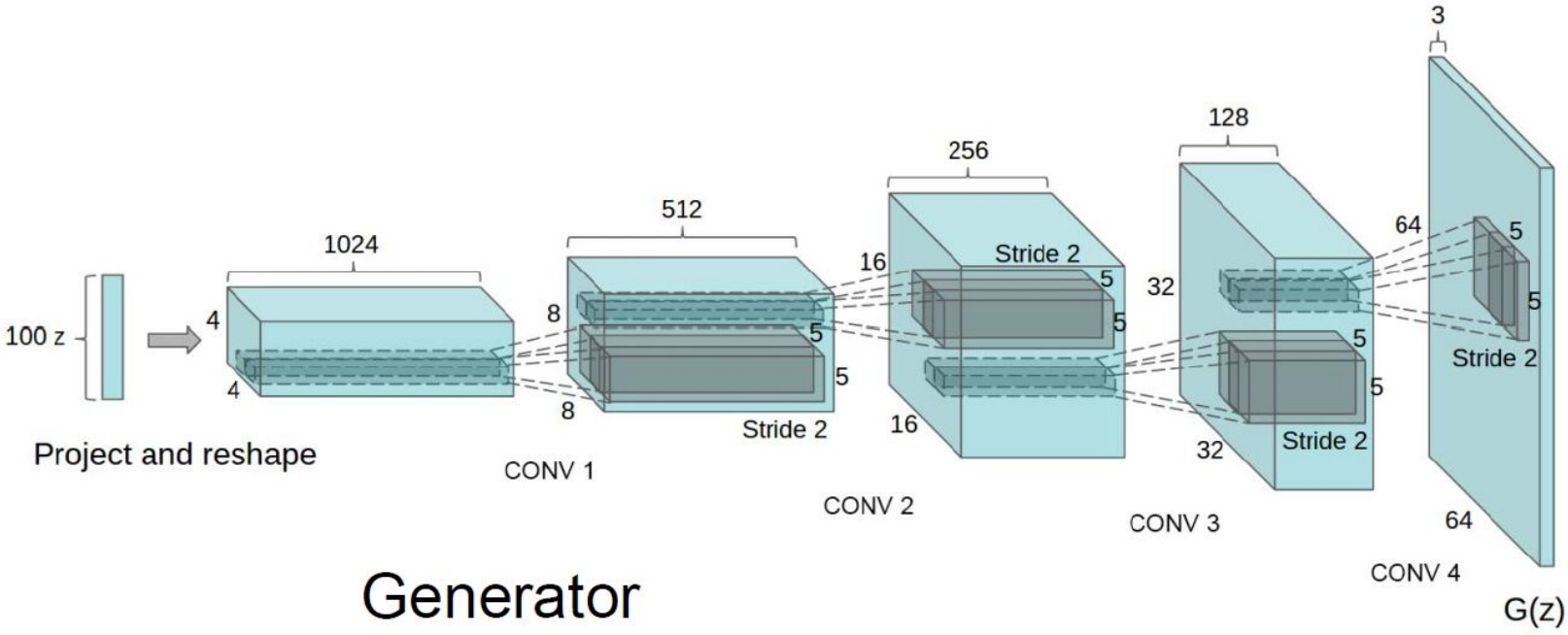
Generator is an upsampling network with fractionally-strided convolutions
Discriminator is a convolutional network

Architecture guidelines for stable Deep Convolutional GANs

- Replace any pooling layers with strided convolutions (discriminator) and fractional-strided convolutions (generator).
- Use batchnorm in both the generator and the discriminator.
- Remove fully connected hidden layers for deeper architectures.
- Use ReLU activation in generator for all layers except for the output, which uses Tanh.
- Use LeakyReLU activation in the discriminator for all layers.

Radford et al, "Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks", ICLR 2016

Generative Adversarial Nets: Convolutional Architectures



Generator

Radford et al, "Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks", ICLR 2016

Generative Adversarial Nets: Convolutional Architectures

Samples from the model look amazing!

Radford et al, ICLR 2016

Let’s code!

Deep Conv GANS: DCGAN-CIFAR10

Training GANs: Two-player game

Ian Goodfellow et al., "Generative Adversarial Nets", NIPS 2014

Putting it together: GAN training algorithm

Some find k=1 more stable, others use k > 1, no best rule.

Recent work (e.g. Wasserstein GAN) alleviates this problem, better stability!

for number of training iterations **do**

  for k steps **do**

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

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D_{\theta_d}(x^{(i)}) + \log(1 - D_{\theta_d}(G_{\theta_g}(z^{(i)}))) \right]$$

  end for

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

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log(D_{\theta_d}(G_{\theta_g}(z^{(i)})))$$

end for

1. Warmup D

Here D changes

```python
# train the discriminator model
def train_discriminator(model, dataset, n_iter=20, n_batch=128):
  half_batch = int(n_batch / 2)
  # manually enumerate epochs
  for i in range(n_iter):
    # get randomly selected 'real' samples
    X_real, y_real = generate_real_samples(dataset, half_batch)
    # update discriminator on real samples
    _, real_acc = model.train_on_batch(X_real, y_real)
    # generate 'fake' examples
    X_fake, y_fake = generate_fake_samples(half_batch)
    # update discriminator on fake samples
    _, fake_acc = model.train_on_batch(X_fake, y_fake)
    # summarize performance
    print('>%d real=%.0f%% fake=%.0f%%' % (i+1, real_acc*100, fake_acc*100))

# define the discriminator model
model = define_discriminator()
# load image data
dataset = load_real_samples()
# fit the model
train_discriminator(model, dataset)
```

Some find k=1 more stable, others use k > 1, no best rule.

for number of training iterations **do**
  for k steps **do**
  - Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
  - Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
  - Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D_{\theta_d}(x^{(i)}) + \log(1 - D_{\theta_d}(G_{\theta_g}(z^{(i)}))) \right]$$

2. Define G

Don’t compile!

G is always optimized based on the discrimination objective as part of the composite model (D(G(Z))

```
# define the size of the latent space
latent_dim = 100
# define the generator model
model = define_generator(latent_dim)
# summarize the model
```

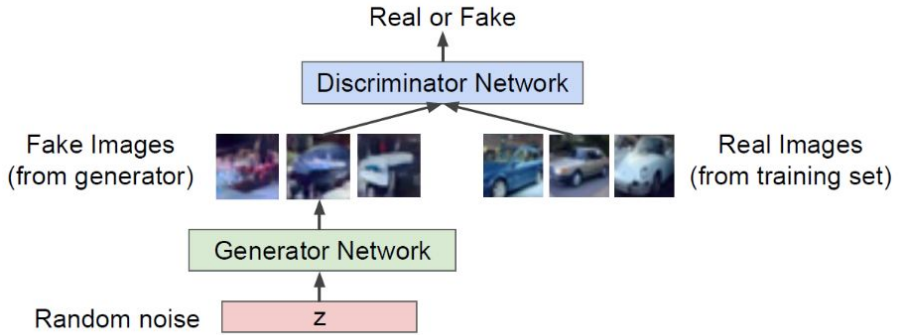
$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log(D_{\theta_d}(G_{\theta_g}(z^{(i)})))$$

3. Composite model

D is not trainable here

Only G is to change:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log(D_{\theta_d}(G_{\theta_g}(z^{(i)})))$$




```python
# define the combined generator and discriminator model, for updating the generator
def define_gan(g_model, d_model):
  # make weights in the discriminator not trainable
  d_model.trainable = False
  # connect them
  model = Sequential()
  # add generator
  model.add(g_model)
  # add the discriminator
  model.add(d_model)
  # compile model
  opt = Adam(lr=0.0002, beta_1=0.5)
  model.compile(loss='binary_crossentropy', optimizer=opt)
  return model
```

4. Train composite

Training GANs: Two-player game

Ian Goodfellow et al., "Generative Adversarial Nets", NIPS 2014

Putting it together: GAN training algorithm

Some find k=1 more stable, others use k > 1, no best rule.

Recent work (e.g. Wasserstein GAN) alleviates this problem, better stability!

for number of training iterations **do**

  for k steps **do**

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

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D_{\theta_d}(x^{(i)}) + \log(1 - D_{\theta_d}(G_{\theta_g}(z^{(i)}))) \right]$$

  end for

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

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log(D_{\theta_d}(G_{\theta_g}(z^{(i)})))$$

end for

4. Train composite

The objective here is to train G→ find theta_G

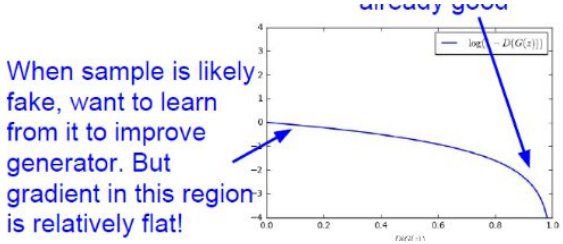
Such that, D is fooled → minimized

So we get samples from G. Two options:

1- Feed as fake (0), and if D says real (1), then it's fooled

$$\min_{\theta_g} \mathbb{E}_{z \sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z)))$$

Gradient Descent



4. Train composite

The objective here is to train G→ find theta_G

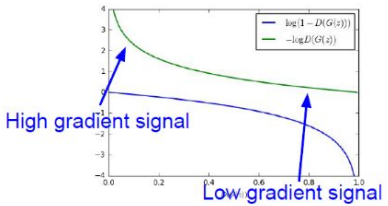
Such that, D is fooled → minimized

So we get samples from G. Two options:

2- Feed as real (1), and if D says real (1), then it's fooled

$$\max_{\theta_g} \mathbb{E}_{z \sim p(z)} \log(D_{\theta_d}(G_{\theta_g}(z)))$$

Gradient Ascent



4. Train composite

for number of training iterations **do**
  for k steps **do**
  - Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
  - Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
  - Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D_{\theta_d}(x^{(i)}) + \log(1 - D_{\theta_d}(G_{\theta_g}(z^{(i)}))) \right]$$

  end for
  - Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
  - Update the generator by ascending its stochastic gradient (improved objective):

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log(D_{\theta_d}(G_{\theta_g}(z^{(i)})))$$

end for

```python
# train the generator and discriminator
def train(g_model, d_model, gan_model, dataset, latent_dim, n_epochs=200, n_batch=128):
  bat_per_epo = int(dataset.shape[0] / n_batch)
  half_batch = int(n_batch / 2)
  # manually enumerate epochs
  for i in range(n_epochs):
    # enumerate batches over the training set
    for j in range(bat_per_epo):
      # get randomly selected 'real' samples
      X_real, y_real = generate_real_samples(dataset, half_batch)
      # update discriminator model weights
      d_loss1, _ = d_model.train_on_batch(X_real, y_real)
      # generate 'fake' examples
      X_fake, y_fake = generate_fake_samples(g_model, latent_dim, half_batch)
      # update discriminator model weights
      d_loss2, _ = d_model.train_on_batch(X_fake, y_fake)
      # prepare points in latent space as input for the generator
      X_gan = generate_latent_points(latent_dim, n_batch)
      # create inverted labels for the fake samples
      y_gan = ones((n_batch, 1))
      # update the generator via the discriminator's error
      g_loss = gan_model.train_on_batch(X_gan, y_gan)
      # summarize loss on this batch
      print('>%d, %d/%d, d1=%.3f, d2=%.3f g=%.3f' %
        (i+1, j+1, bat_per_epo, d_loss1, d_loss2, g_loss))
```

4. Train composite

for number of training iterations **do**

- Sample minibatch of m noise samples $\{z^{(1)}, \ldots, z^{(m)}\}$ from noise prior $p_g(z)$.
- Sample minibatch of m examples $\{x^{(1)}, \ldots, x^{(m)}\}$ from data generating distribution

- Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D_{\theta_d}(x^{(i)}) + \log(1 - D_{\theta_d}(G_{\theta_g}(z^{(i)}))) \right]$$

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

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log(D_{\theta_d}(G_{\theta_g}(z^{(i)})))$$

```python
# train the generator and discriminator
def train(g_model, d_model, gan_model, dataset, latent_dim, n_epochs=200, n_batch=128):
  bat_per_epo = int(dataset.shape[0] / n_batch)
  half_batch = int(n_batch / 2)
  # manually enumerate epochs
  for i in range(n_epochs):
    # enumerate batches over the training set
    for j in range(bat_per_epo):
      # get randomly selected 'real' samples
      X_real, y_real = generate_real_samples(dataset, half_batch)
      # update discriminator model weights
      d_loss1, _ = d_model.train_on_batch(X_real, y_real)
      # generate 'fake' examples
      X_fake, y_fake = generate_fake_samples(g_model, latent_dim, half_batch)
      # update discriminator model weights
      d_loss2, _ = d_model.train_on_batch(X_fake, y_fake)
      # prepare points in latent space as input for the generator
      X_gan = generate_latent_points(latent_dim, n_batch)
      # create inverted labels for the fake samples
      y_gan = ones((n_batch, 1))
      # update the generator via the discriminator's error
      g_loss = gan_model.train_on_batch(X_gan, y_gan)
      # summarize loss on this batch
      print('>%d, %d/%d, d1=%.3f, d2=%.3f g=%.3f' %
        (i+1, j+1, bat_per_epo, d_loss1, d_loss2, g_loss))
```

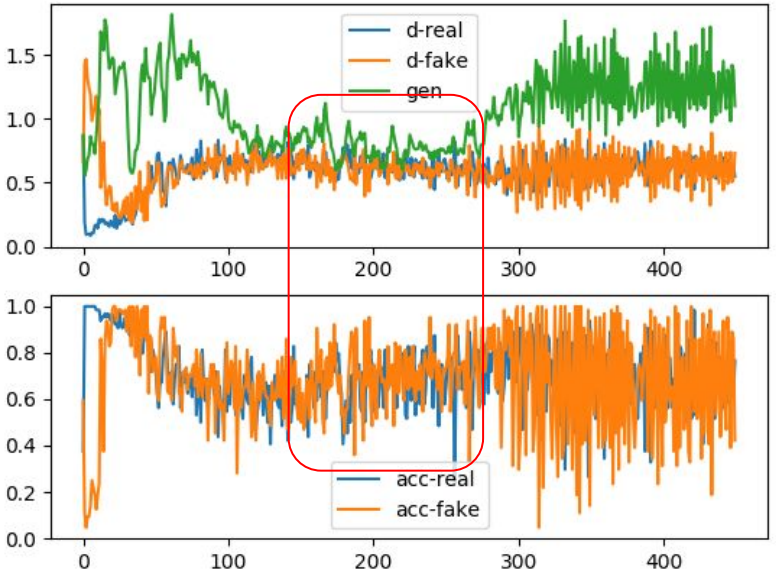
GANs are tricky!

How to diagnose GANs failure modes in practice!

Stable learning curves

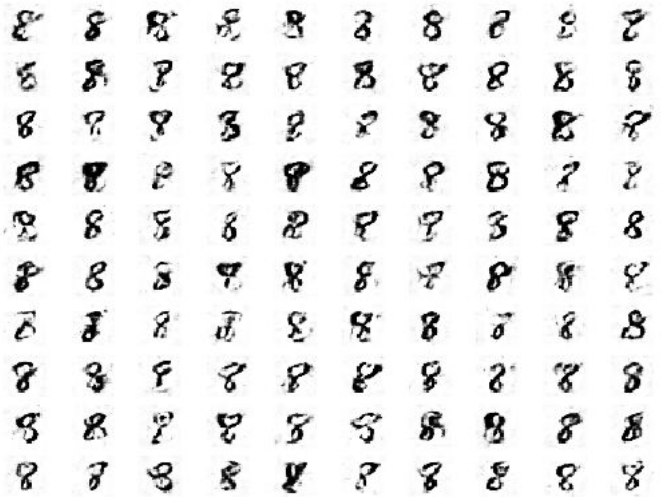
d_real = d_fake = 0.5
→ confused D

acc-real=acc-fake (not loss)



Mode collapse

Method 1: visualize images→
Not one category is produced all the time



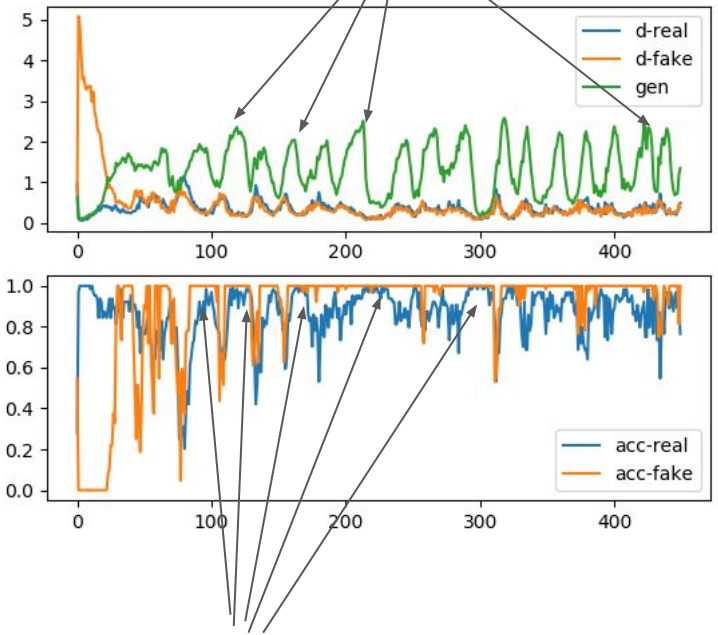
Mode collapse

Method 2: A mode collapse can also be identified by reviewing the line plot of model loss.

The line plot will show oscillations in the loss over time.

Most notably in the generator model, as the generator model is updated and jumps from generating one mode to another model that has different loss.

High gen (disc) loss at the different modes

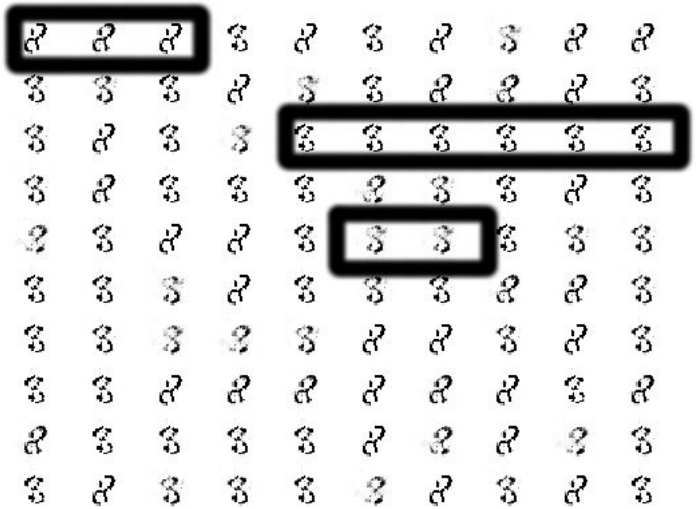


Only low gen(disc) loss, high acc (fake/real), at one model → This will end up to be the favored mode in generation.

Mode collapse

How to cause it?:

We can impair our stable GAN to suffer mode collapse a number of ways. Perhaps the most reliable is to restrict the size of the latent dimension directly, forcing the model to only generate a small subset of plausible outputs.



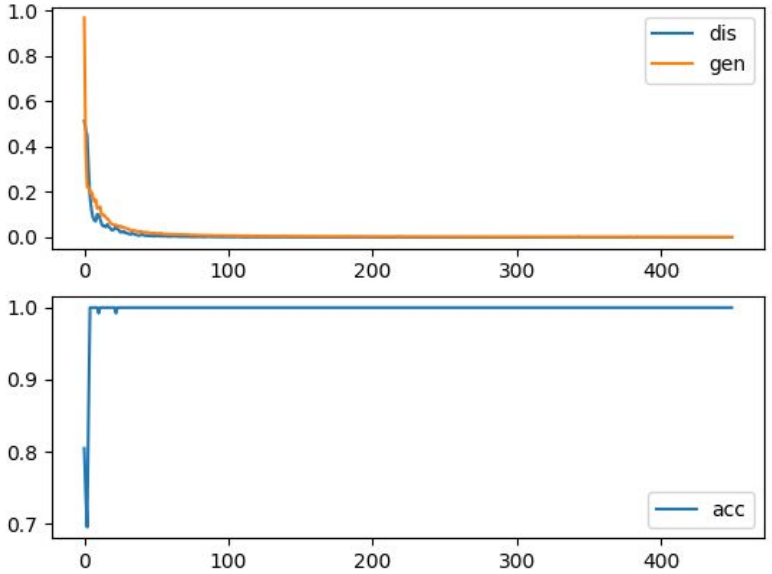
So the cure is to give higher degree of freedom in the latent_dim

Convergence Failure → D not fooled!

The likely way that you will identify this type of failure is that the loss for the discriminator has gone to zero or close to zero.

This type of loss is most commonly caused by the generator outputting garbage images that the discriminator can easily identify.

For some unstable GANs, it is possible for the GAN to fall into this failure mode for a number of batch updates, or even a number of epochs, and then recover.



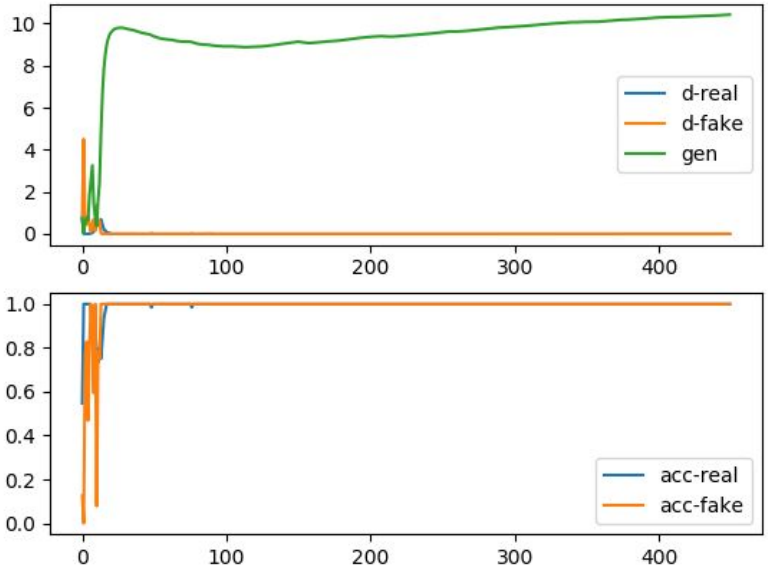
Convergence Failure → D not fooled!

The likely way that you will identify this type of failure is that the loss for the discriminator has gone to zero or close to zero.

In some cases, the loss of the generator may also rise and continue to rise over the same period.

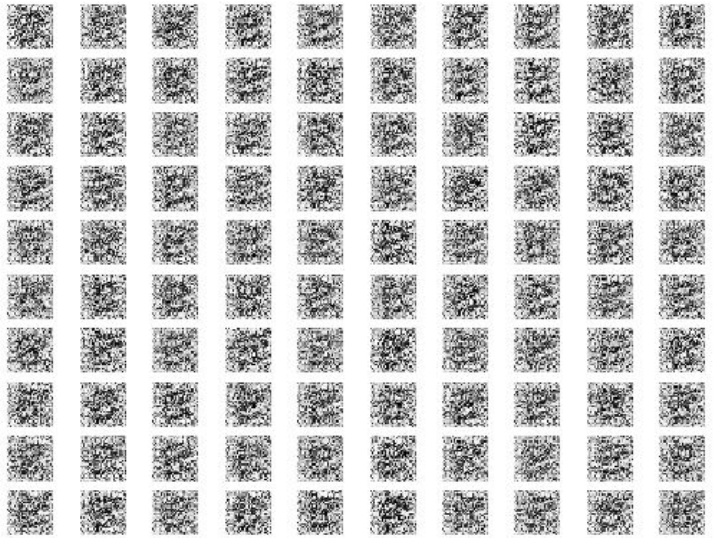
This type of loss is most commonly caused by the generator outputting garbage images that the discriminator can easily identify.

For some unstable GANs, it is possible for the GAN to fall into this failure mode for a number of batch updates, or even a number of epochs, and then recover.



Convergence Failure → D not fooled!

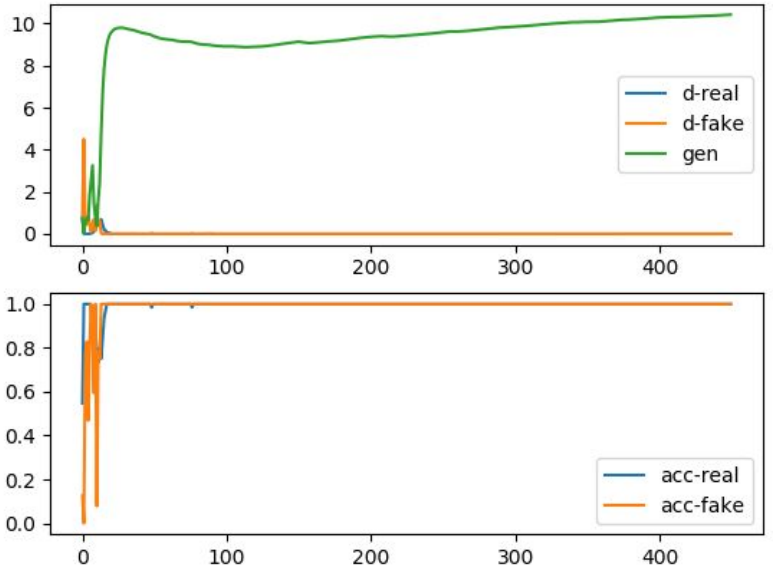
Visual symptoms: low quality (static noise) images



Convergence Failure

How to cause it?

Underfitting one of the models (generator mainly): changing one or both models to have insufficient capacity, changing the Adam optimization algorithm to be too aggressive, and using very large or very small kernel sizes in the models.

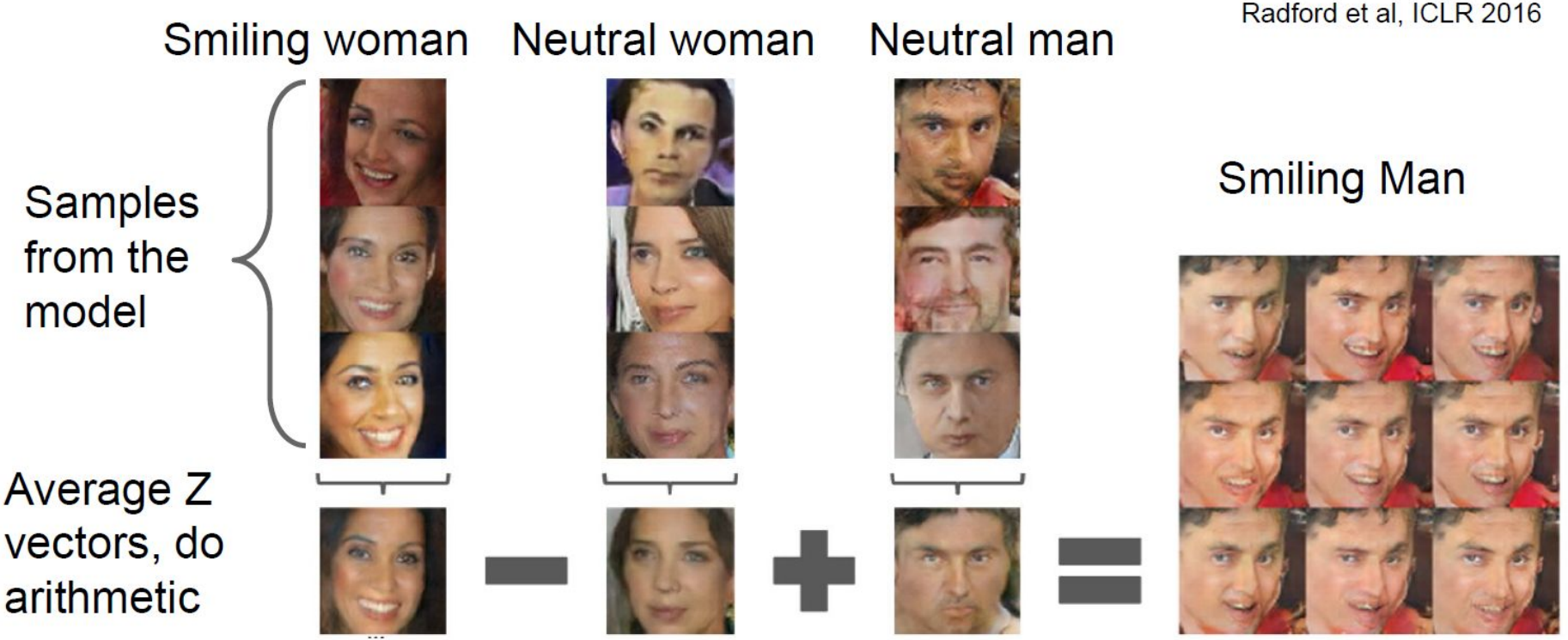


Cure:

Sufficient model (generator) capacity

Generative Adversarial Nets: Interpretable Vector Math

Radford et al, ICLR 2016



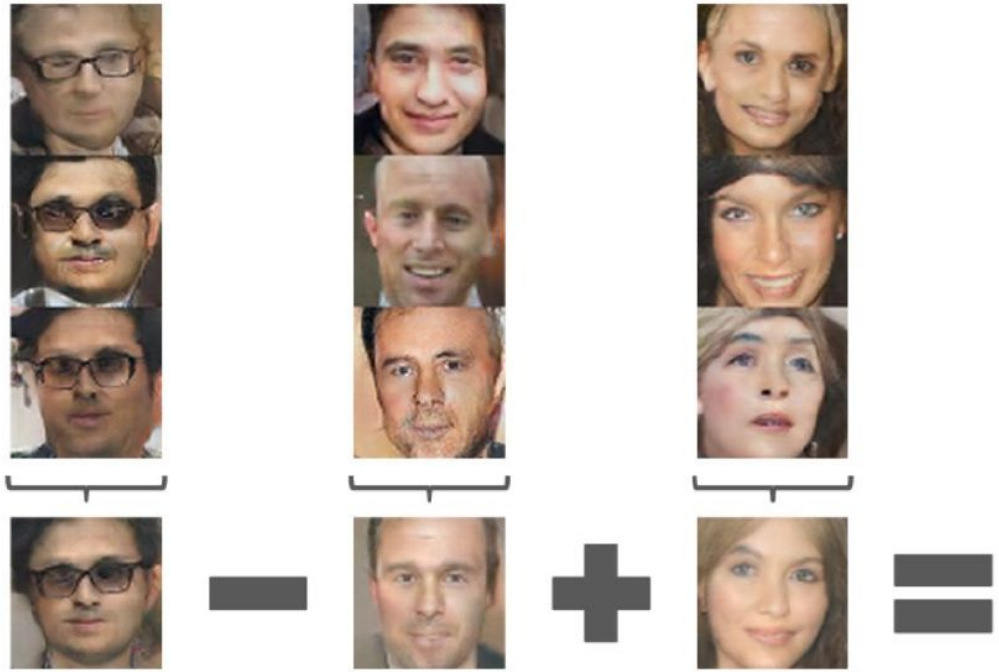
Generative Adversarial Nets: Interpretable Vector Math

Glasses man

No glasses man

No glasses woman

Radford et al, ICLR 2016



Woman with glasses

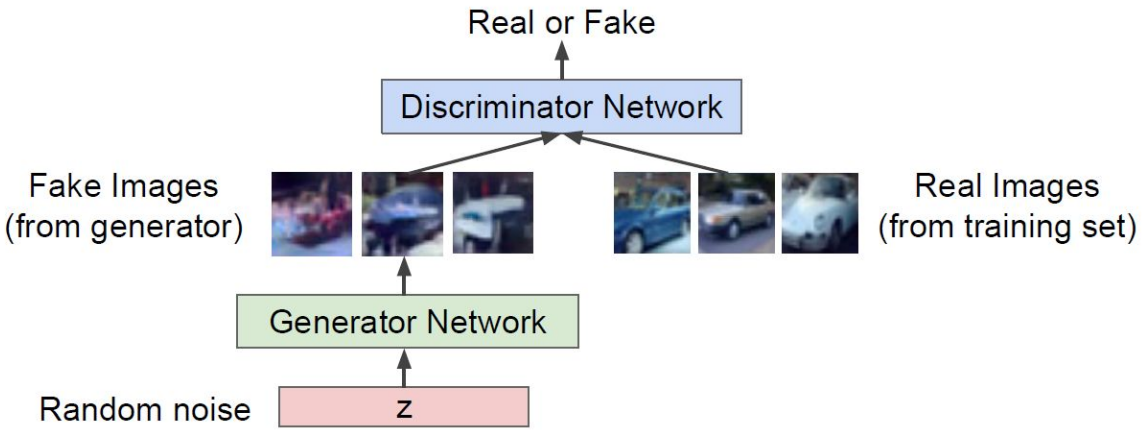
Let’s code!

Explore GAN Latent Space

Conditional GANs

GAN: Z~N(0,1)

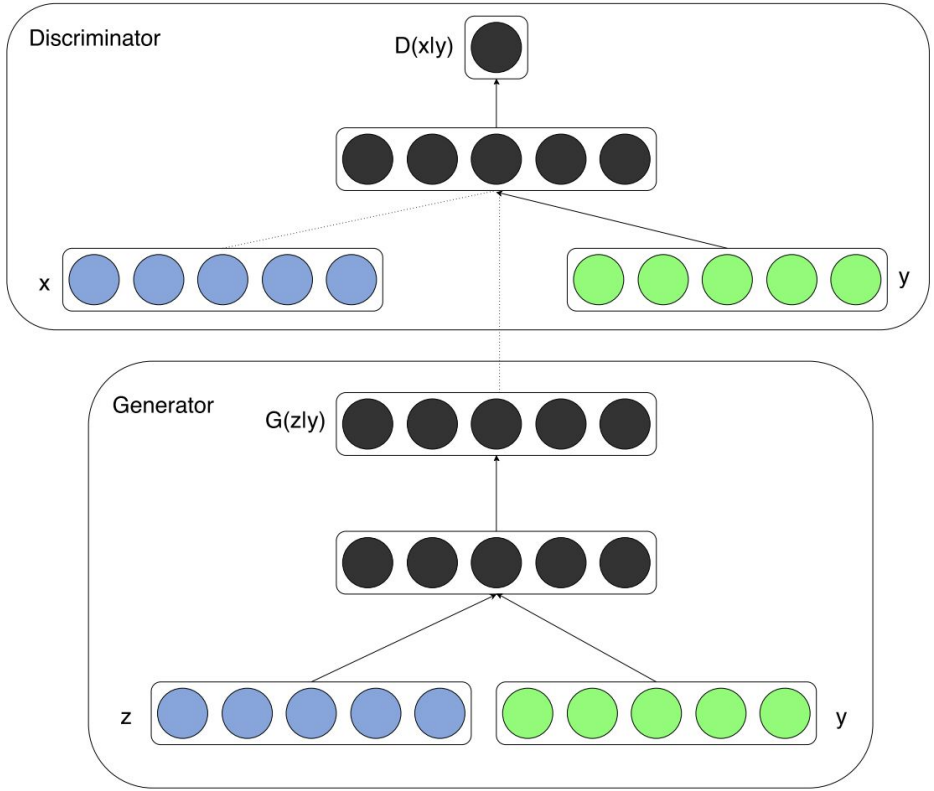
Although we could interpret Z, but we cannot force it!



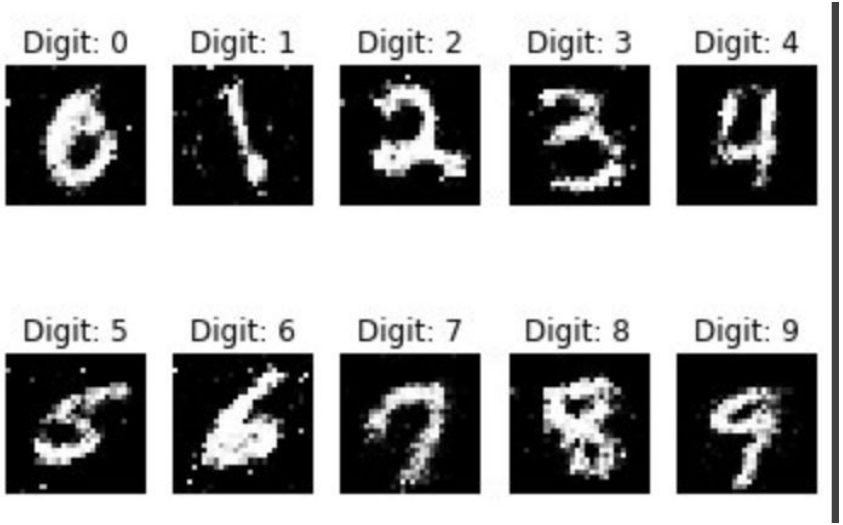
Conditional GANs

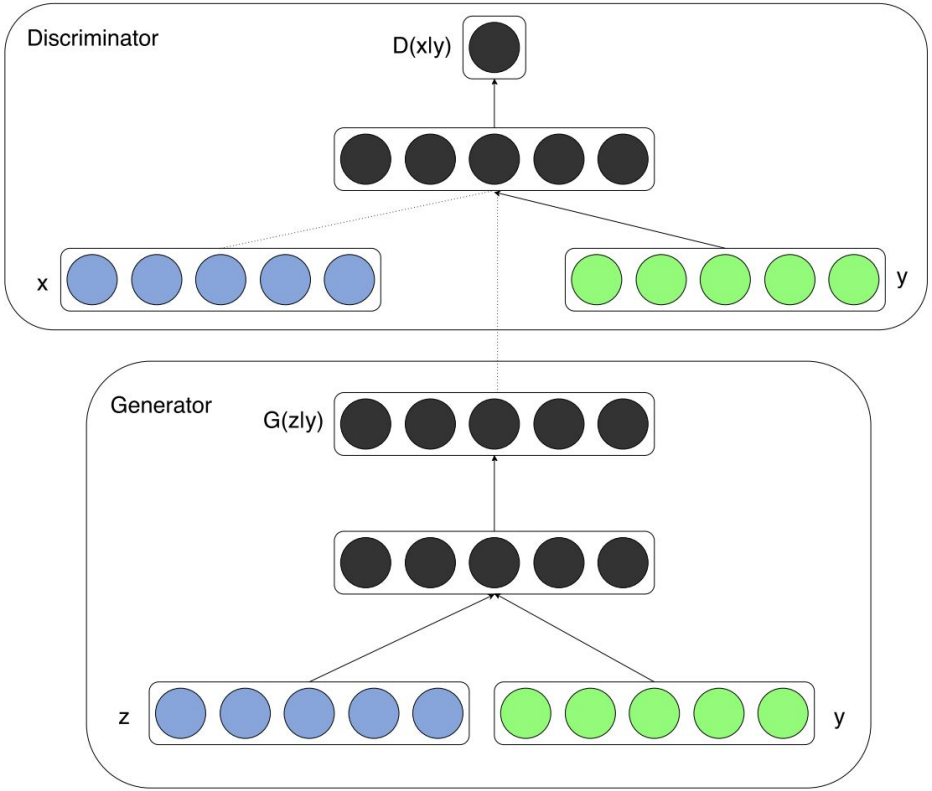
GAN: Z~N(0,1)

cGAN: Z~1-K (MNIST)



Conditional GANs





Let’s code!

Conditional GANs

AttributeGANs

https://arxiv.org/pdf/1711.10678.pdf

Fig. 1. Facial attribute editing results from our AttGAN. Zoom in for better resolution.

AttributeGANs

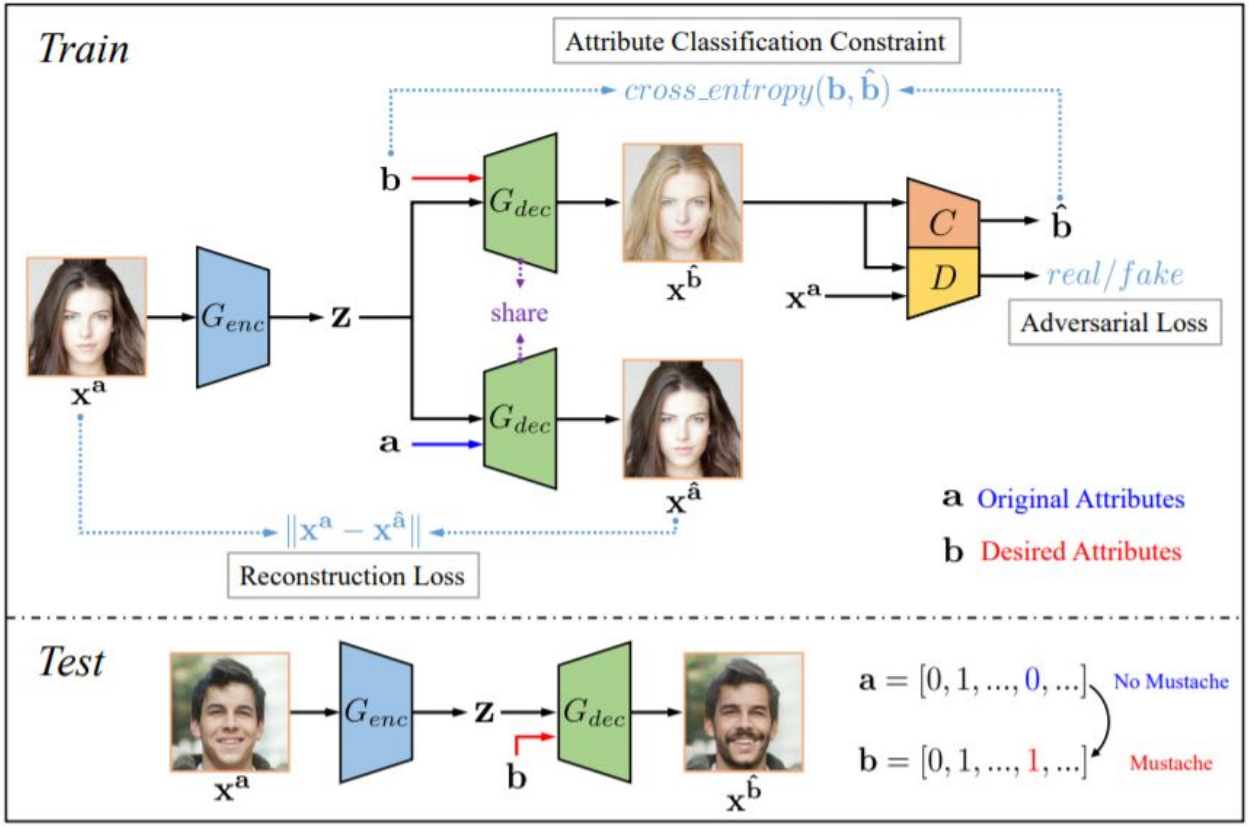
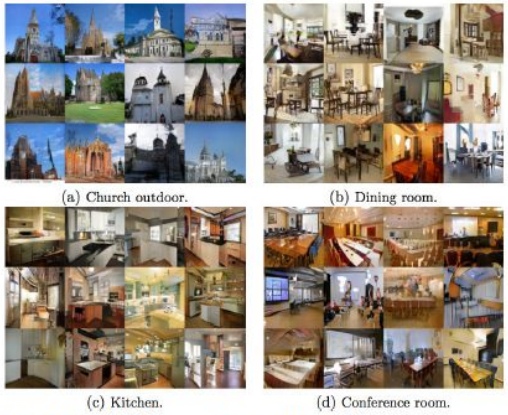


Fig. 2. Overview of our AttGAN, which contains three main components at training: the attribute classification constraint, the reconstruction learning and the adversarial learning. The attribute classification constraint guarantees the correct attribute manipulation on the generated image. The reconstruction learning aims at preserving the attribute-excluding details. The adversarial learning is employed for visually realistic generation.

2017: Year of the GAN

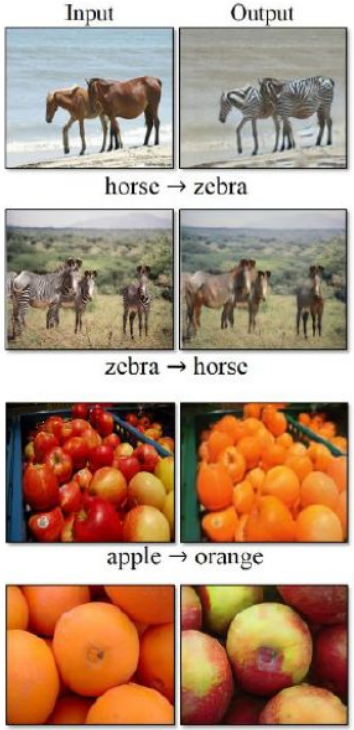
Better training and generation



LSGAN. Mao et al. 2017.

BEGAN. Bertholet et al. 2017.

Source->Target domain transfer



CycleGAN. Zhu et al. 2017.

Text -> Image Synthesis

Reed et al. 2017.

Many GAN applications

Pix2pix. Isola 2017. Many examples at https://phillipi.github.io/pix2pix/

How good are GANs today?

Progressive GANs

StyleGAN

All fake!

Let’s code!

Progressive GANs

Image-to-image translation

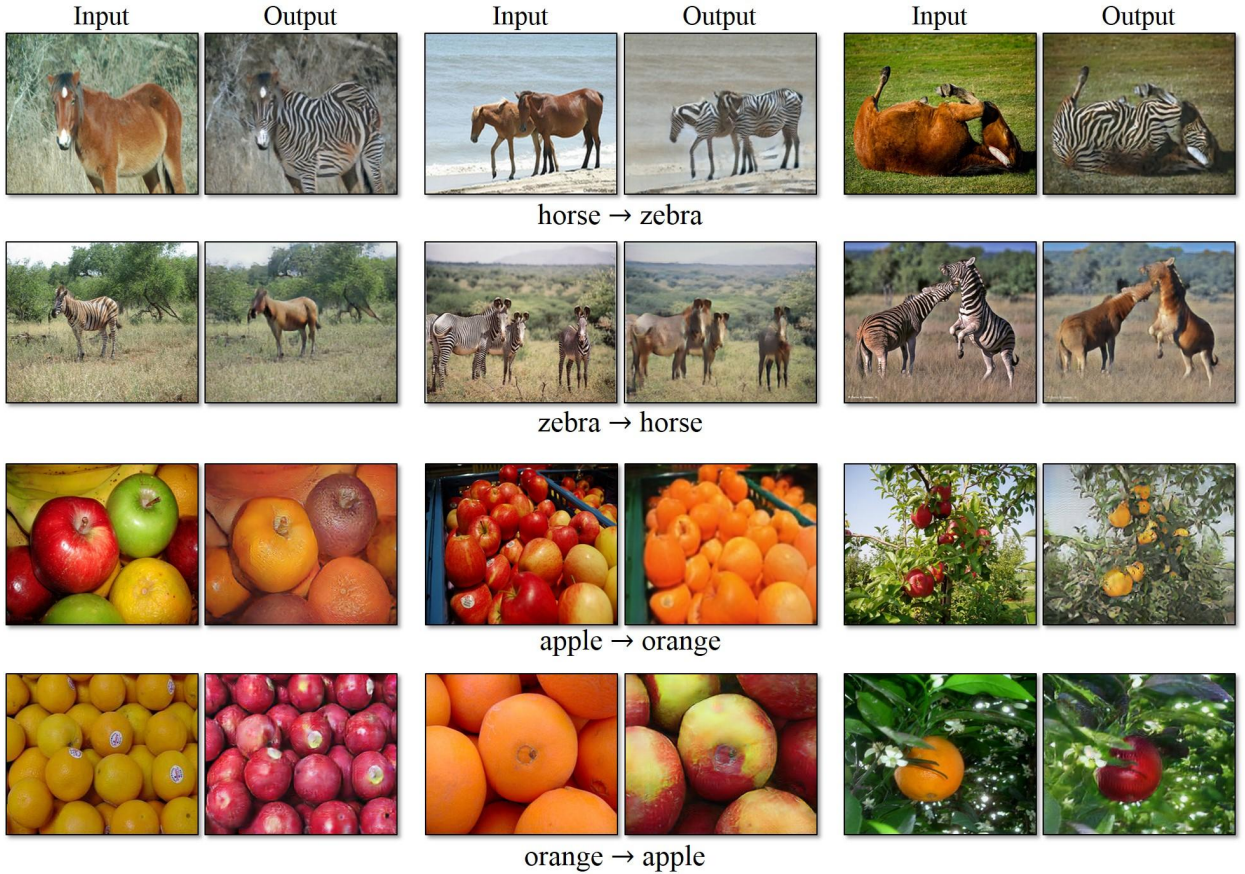
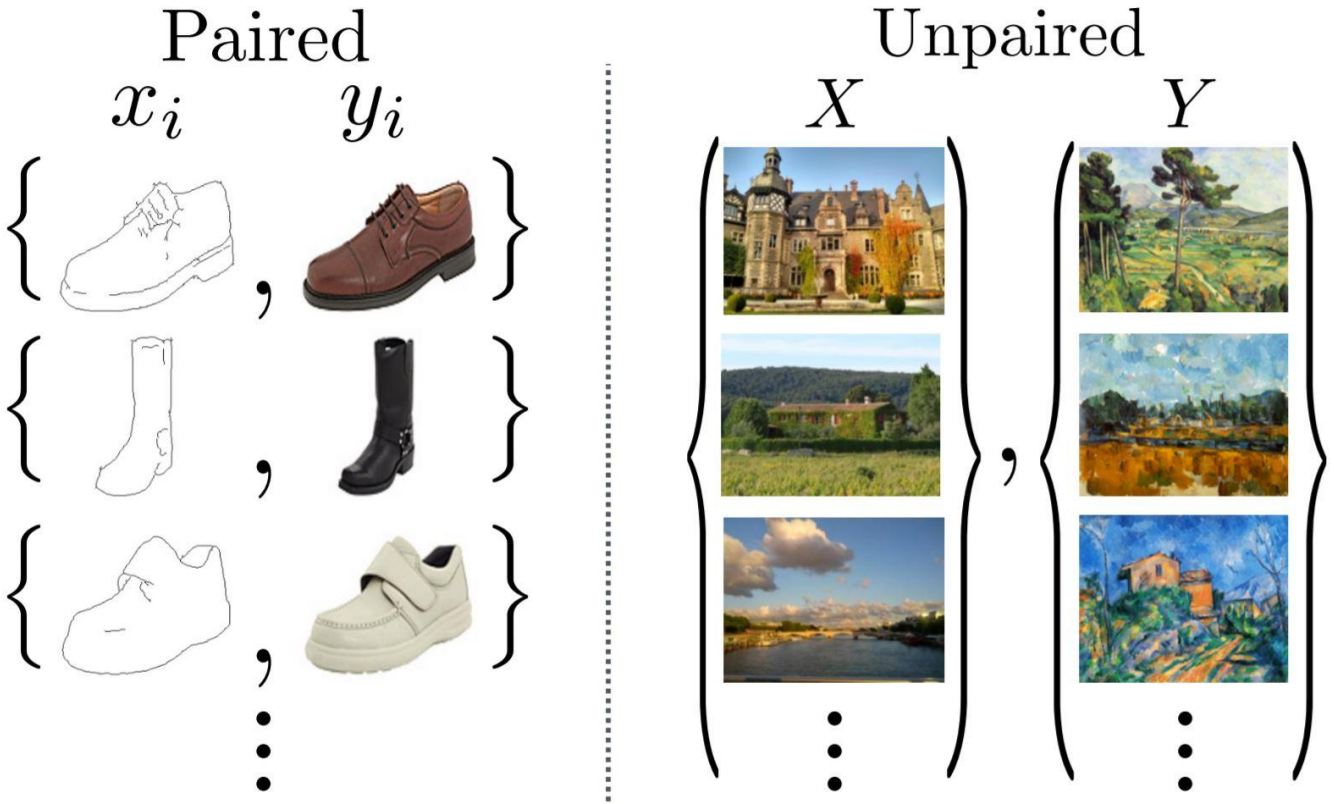
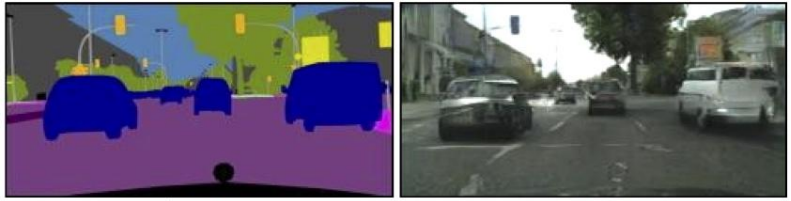


Image-to-image translation



Pix2pix

Labels to Street Scene

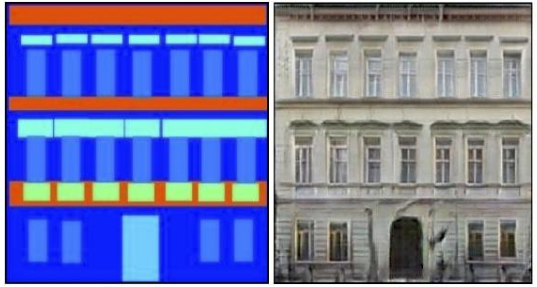


input output

Aerial to Map

input output

Labels to Facade



input output

BW to Color

input output

Day to Night

input output

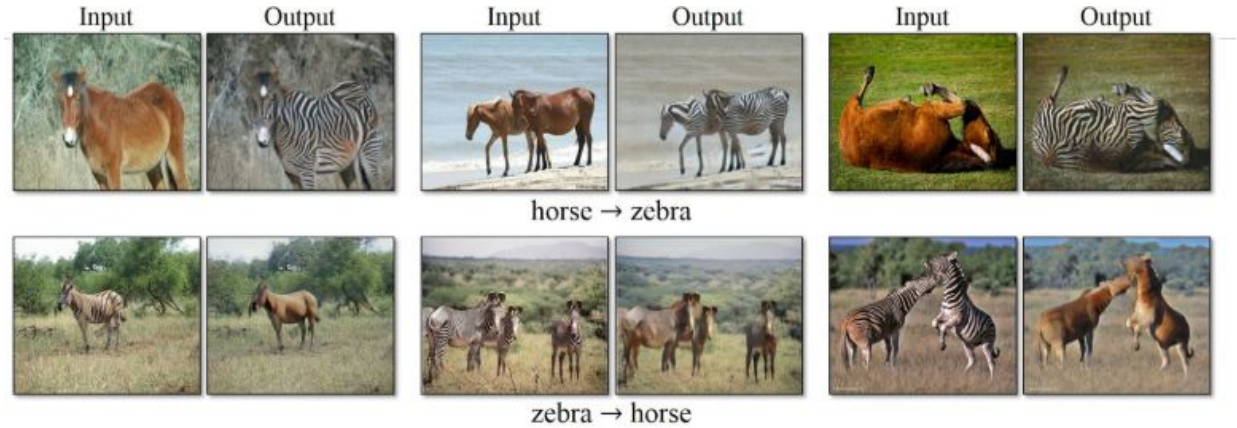
Edges to Photo

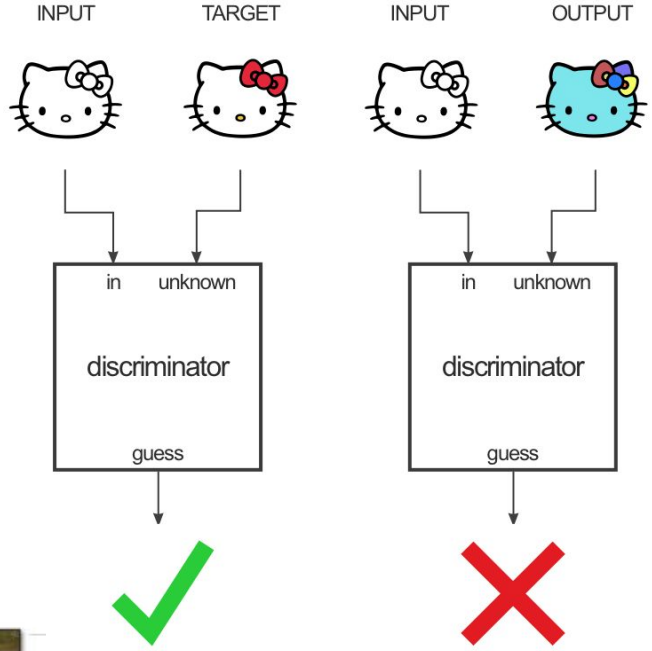
input output

Pix2pix

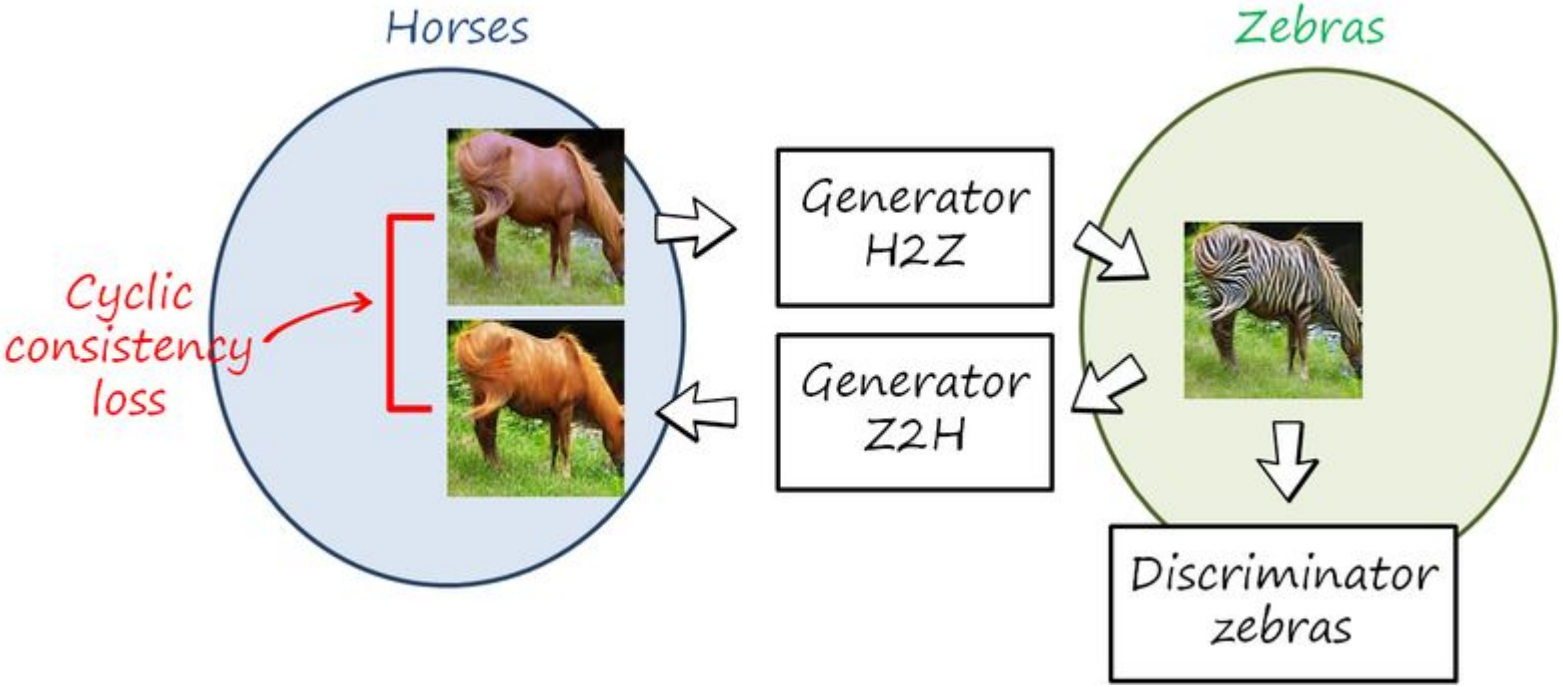
But this is “hard” supervised → Easy to learn!

→ How to get the corresponding output for:

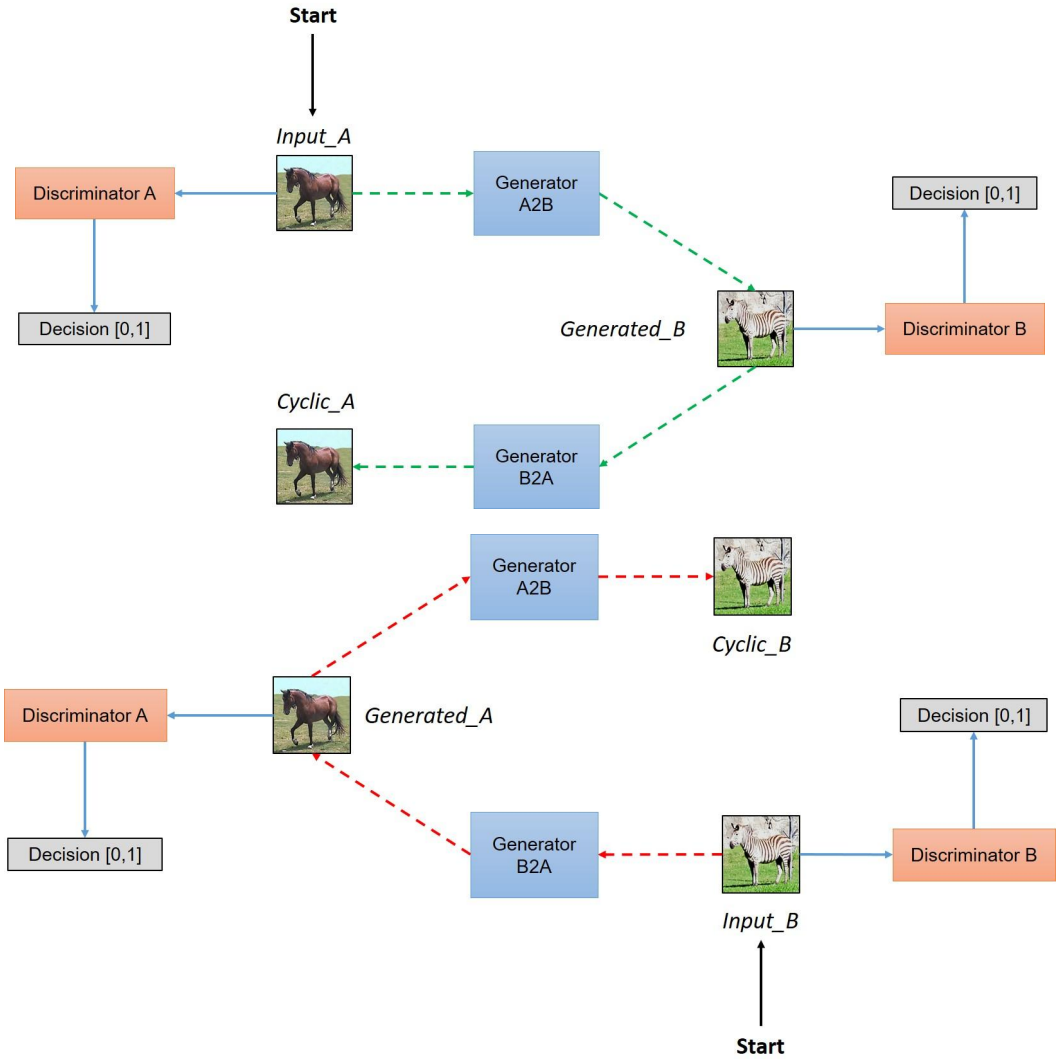




CycleGAN


Horses
Zebras
Cyclic consistency loss
Generator H2Z
Generator Z2H
Discriminator zebras

CycleGAN


Start
Input_A
Discriminator A
Generator A2B
Decision [0,1]
Decision [0,1]
Generated_B
Discriminator B
Cyclic_A
Generator B2A
Generator A2B
Cyclic_B
Discriminator A
Generated_A
Decision [0,1]
Decision [0,1]
Generator B2A
Discriminator B
Input_B
Start

CycleGAN - 3 losses

- (2x) Adversarial loss (GAN loss) → Disc → cross-domain mapping → L2 → MSE
  - X→ G→ Y → DY(Y, G(X)) → Real/Fake Y
  - Y→ F→ X → DX(X, F(Y) → Real/Fake X
- (2x) Identity loss → Disc → same domain mapping → L1 → MAE
  - Y <> G(Y)
  - X <> F(X)

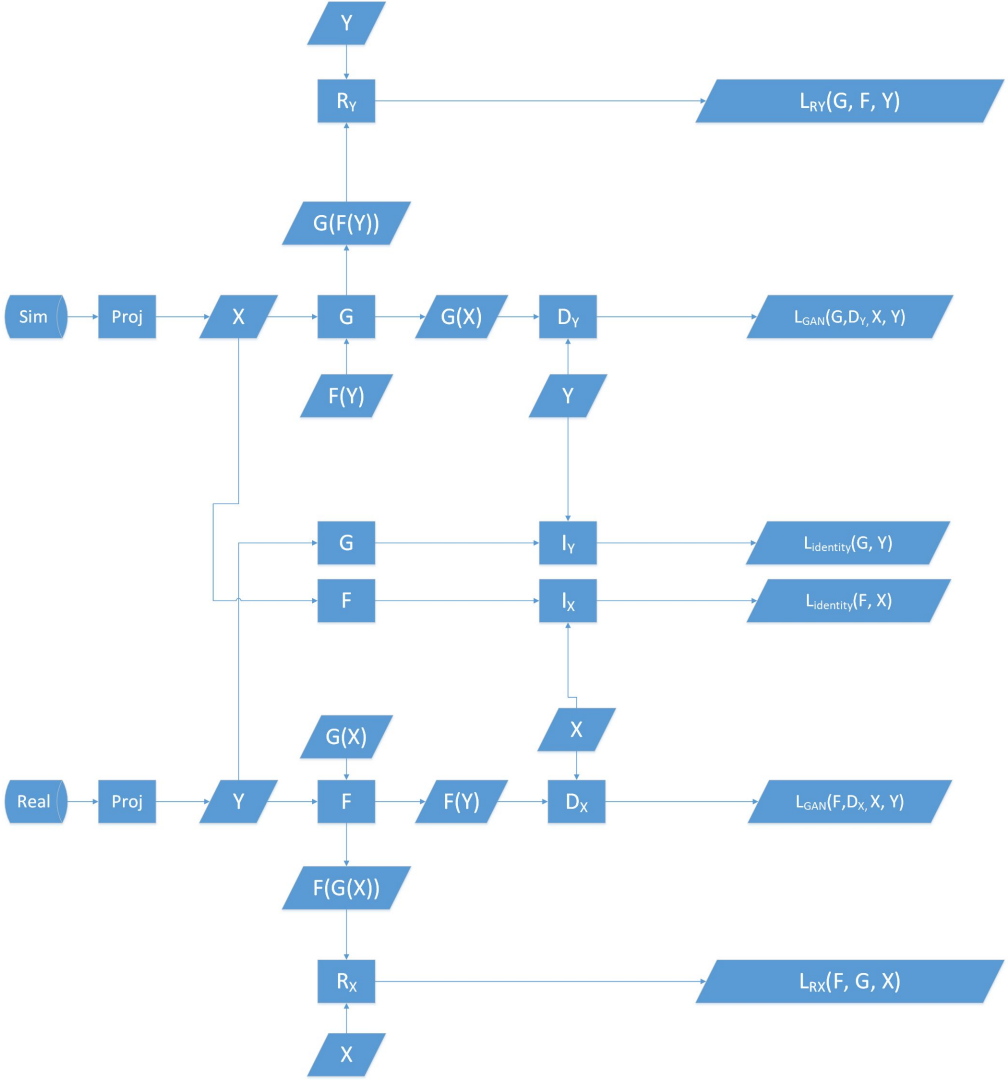
→ Good for learning the mapping

→ Who guarantees B is a translated version of A and vice versa?

- (2x) Cycle consistency / Reconstruction→ paired translation → L1 → MAE
  - X <> F(G(X))
  - Y <> G(F(X))

Discriminators use PatchGAN → generate heatmap for each 70x70 patch of the input = real/fake → Good for variable length images (fully conv)

-


Y
R_Y
$L_{RY}(G, F, Y)$
G(F(Y))
Sim
Proj
X
G
G(X)
D_Y
$L_{GAN}(G,D_Y, X, Y)$
F(Y)
Y
G
I_Y
$L_{identity}(G, Y)$
F
I_X
$L_{identity}(F, X)$
X
G(X)
Real
Proj
Y
F
F(Y)
D_X
$L_{GAN}(F,D_X, X, Y)$
F(G(X))
R_X
$L_{RX}(F, G, X)$
X

$$\mathcal{L}_{GAN}(G, D_Y, X, Y) = E_{y\sim p_{data}(y)}[logD_Y(x)] + E_{x\sim p_{data}(x)}[log(1 - D_Y(G(x)))]$$

$$\mathcal{L}_{GAN}(F, D_X, X, Y) = E_{x\sim p_{data}(x)}[logD_X(x)] + E_{y\sim p_{data}(y)}[log(1 - D_X(F(y)))]$$

$$\mathcal{L}_{R_Y}(G, F, Y) = E_{y\sim p_{data}(y)}[||G(F(x)) - y||_1] = E_{y\sim p_{data}(y)}[R_Y]$$

where: $R_Y = ||G(F(x)) - y||_1$

$$\mathcal{L}_{R_X}(F, G, X) = E_{x\sim p_{data}(x)}[||F(G(y)) - x||_1] = E_{x\sim p_{data}(x)}[R_X]$$

where: $R_X = ||F(G(y)) - x||_1$

$$\mathcal{L}_{I_Y}(G, Y) = E_{y\sim p_{data}(y)}[||G(y) - y||_1] = E_{y\sim p_{data}(y)}[I_Y]$$

where: $I_Y = ||G(y) - y||_1$

$$\mathcal{L}_{I_X}(F, X) = E_{x\sim p_{data}(x)}[||F(x) - x||_1] = E_{x\sim p_{data}(x)}[I_X]$$

where: $I_X = ||F(x) - x||_1$

$$\mathcal{L}(G, F, D_X, D_Y) = \mathcal{L}_{GAN}(G, D_Y, X, Y) + \mathcal{L}_{GAN}(F, D_X, X, Y) + \lambda_{cyc}[\mathcal{L}_{R_Y}(G, F, Y) + \mathcal{L}_{R_X}(F, G, X)] + \lambda_{identity}[\mathcal{L}_{I_Y}(G, Y) + \mathcal{L}_{I_X}(F, X)]$$

The G and F mapping functions are obtained by optimizing the overall loss as follows:

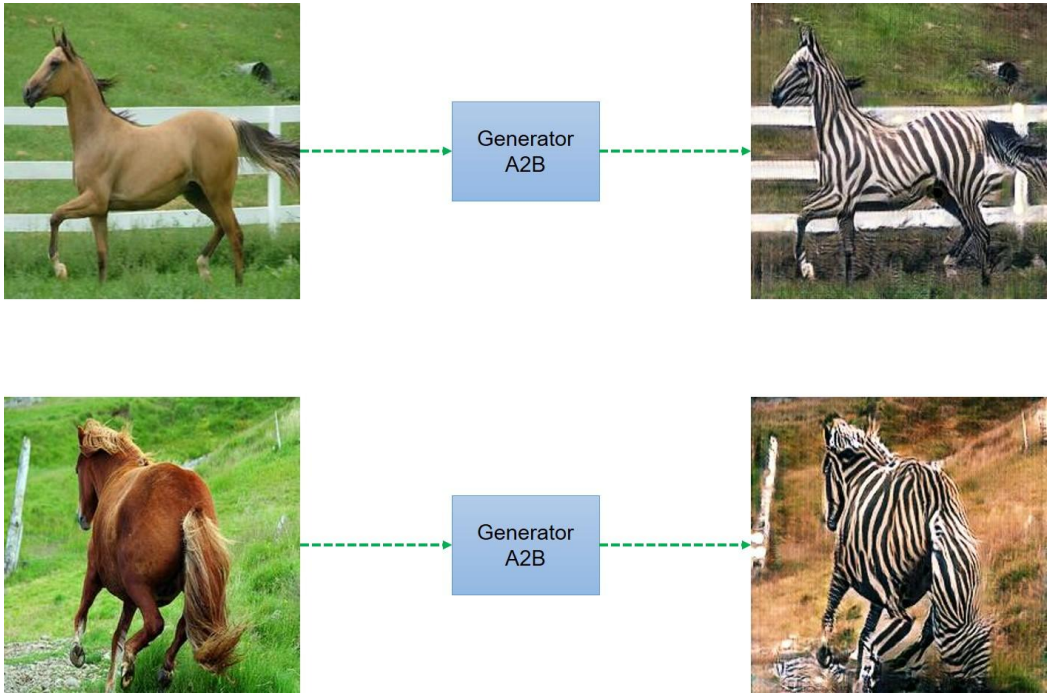
$$G^*, F^* = argmin_{G,F} max_{D_X, D_Y} \mathcal{L}(G, F, D_X, D_Y)$$

Note that, the following can be viewed as autoencoder reconstruction:

$$G(F(Y)) = G.F(Y)$$
$$F(G(X)) = F.G(X)$$

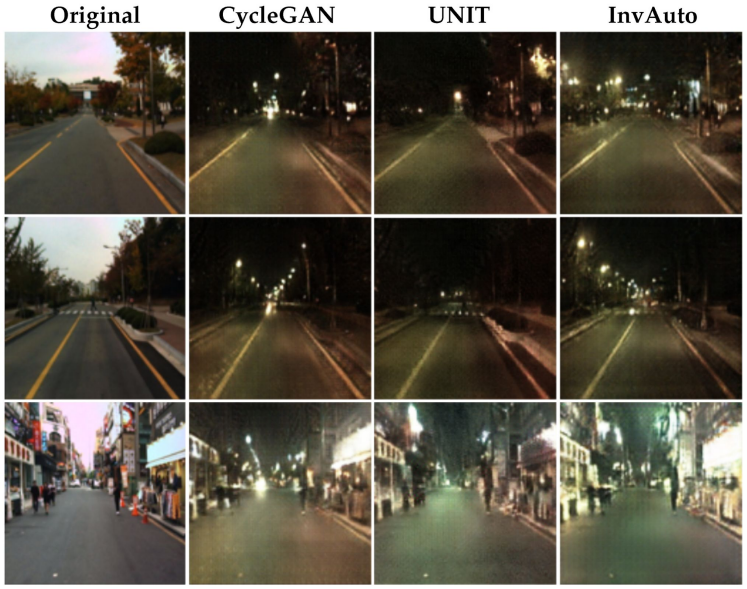
CycleGAN

Generation in both directions



Endless possibilities!

Data augmentation



Endless possibilities!

Sim2Real

Endless possibilities!

Not all good!

DeepFake

Let’s code

- https://machinelearningmastery.com/cyclegan-tutorial-with-keras/

References

- http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture13.pdf
- https://machinelearningmastery.com/how-to-develop-a-generative-adversarial-network-for-a-cifar-10-small-object-photographs-from-scratch/
- https://machinelearningmastery.com/how-to-develop-a-conditional-generative-adversarial-network-from-scratch/
- https://blog.keras.io/building-autoencoders-in-keras.html
- https://machinelearningmastery.com/practical-guide-to-gan-failure-modes/
- https://hardikbansal.github.io/CycleGANBlog/
- https://machinelearningmastery.com/cyclegan-tutorial-with-keras/