DEEP LEARNING FOR VISUAL RECOGNITION

Lecture 9 – Generative Models

Henrik Pedersen, PhD
Part-time lecturer
Department of Computer Science
Aarhus University
hpe@cs.au.dk

Today's agenda

- You will learn about CNN architectures that can generate new images.
- Topics of today
  - Unsupervised learning
  - Generative models
  - Autoencoder (AE) and some of its variants (sparse AE, denoising AE, and contractive AE)
  - Convolutional autoencoders
  - Variational autoencoders
  - Generative Adversarial Networks (GANs)
- Later
  - Diffusion models
  - Self-supervised learning

Last time

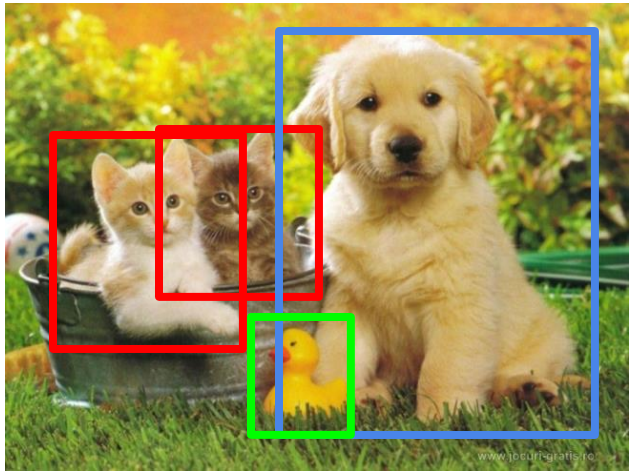
Classification

CAT

Classification
+ Localization

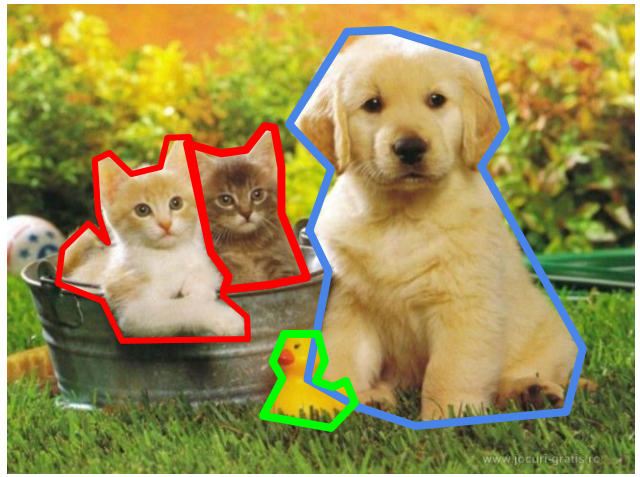
CAT

Object Detection



CAT, DOG, DUCK

Image
Segmentation



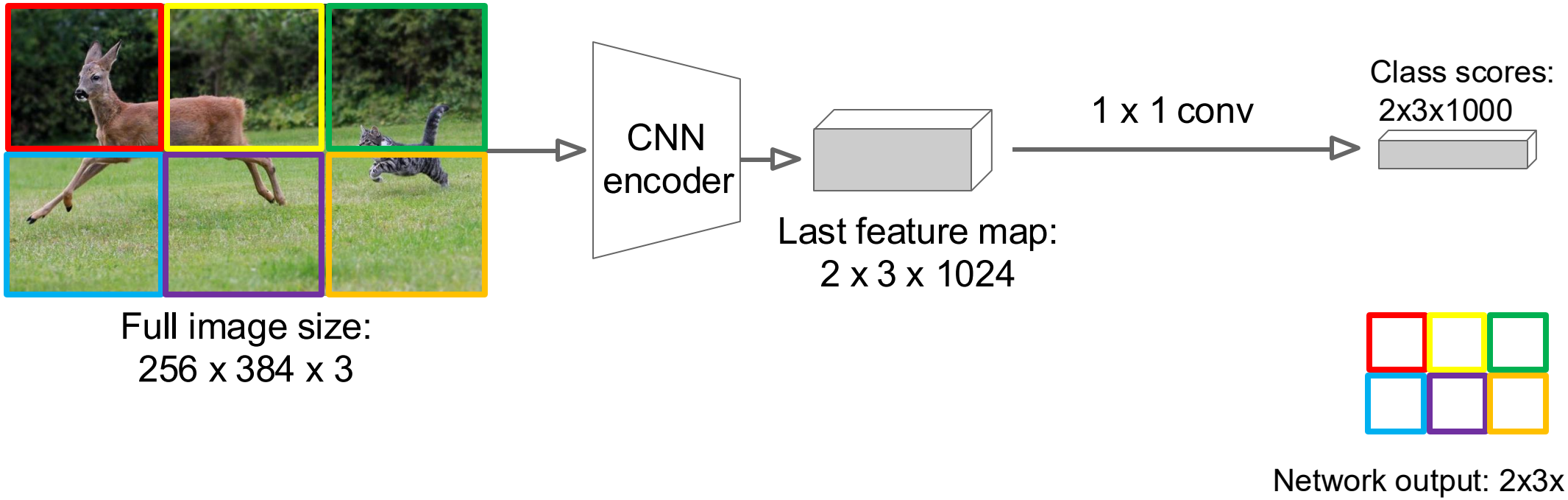
CAT, DOG, DUCK

Single object

Multiple objects

Efficient sliding window using 1x1 conv.

Replacing FC layers with 1x1 convolutions -> a way to do efficient sliding window (in a single forward pass).

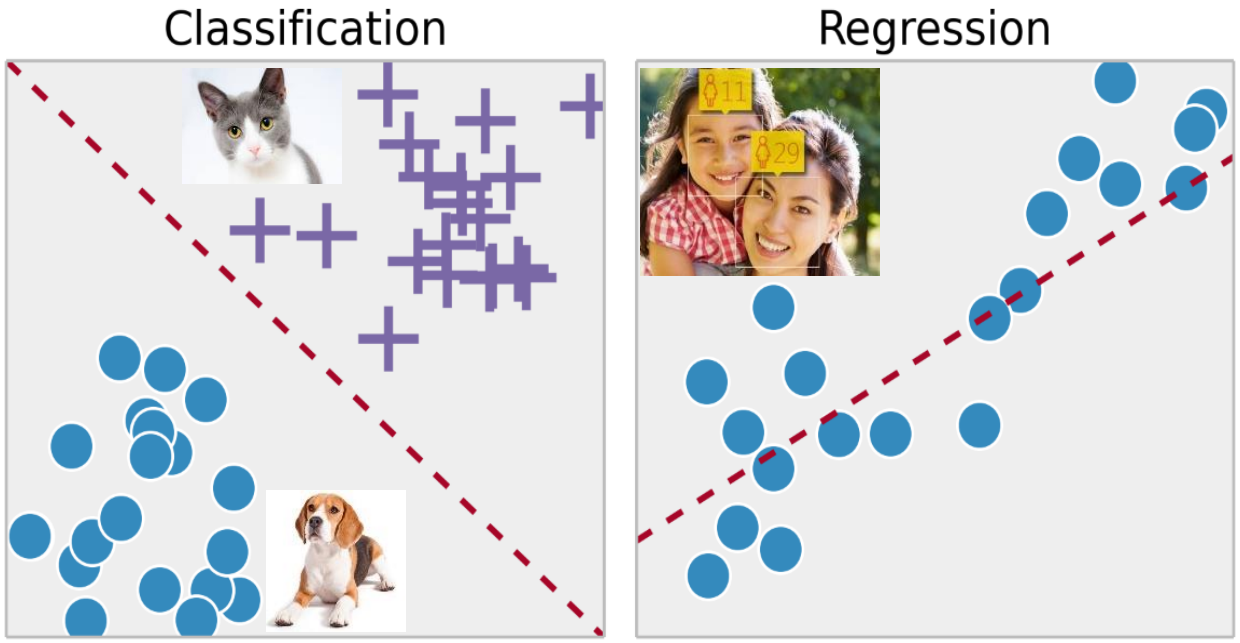


Unsupervised learning

Recall regression vs. classification

Classifier
Uses features to distinguish between two or more classes.

Output:
Discrete labels
(”Dog” or ”Cat”)



Regressor
Uses features to predict some functional relationship.

Output:
Real numbers
(”Age” of person in image)

The problems illustrated above are examples of **supervised learning**

Supervised learning

- **Data**: Comes in pairs (x, y), where x is data (an image in our case), and y is a label/target.
- **Goal**: Learn a *function* to map x -> y
- **Examples**: Classification, localization (regression), object detection, image segmentation, etc.

Classification

Classification + Localization

Object Detection

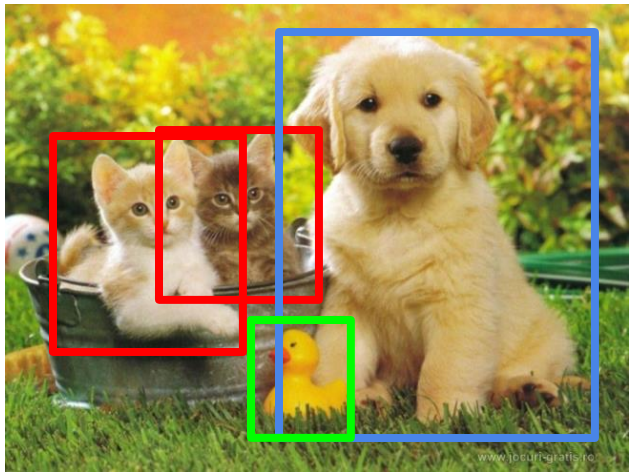
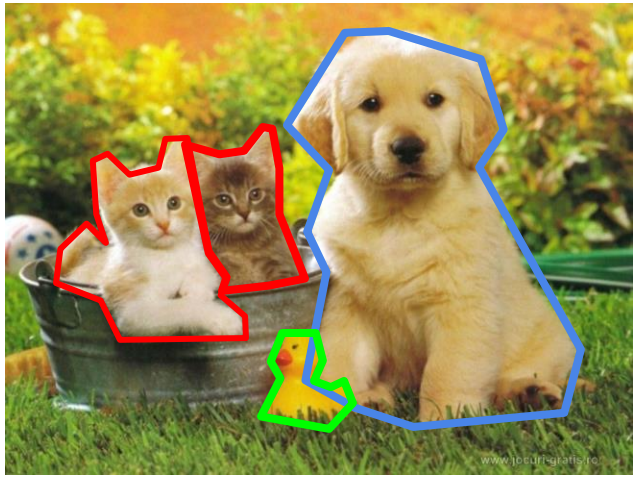
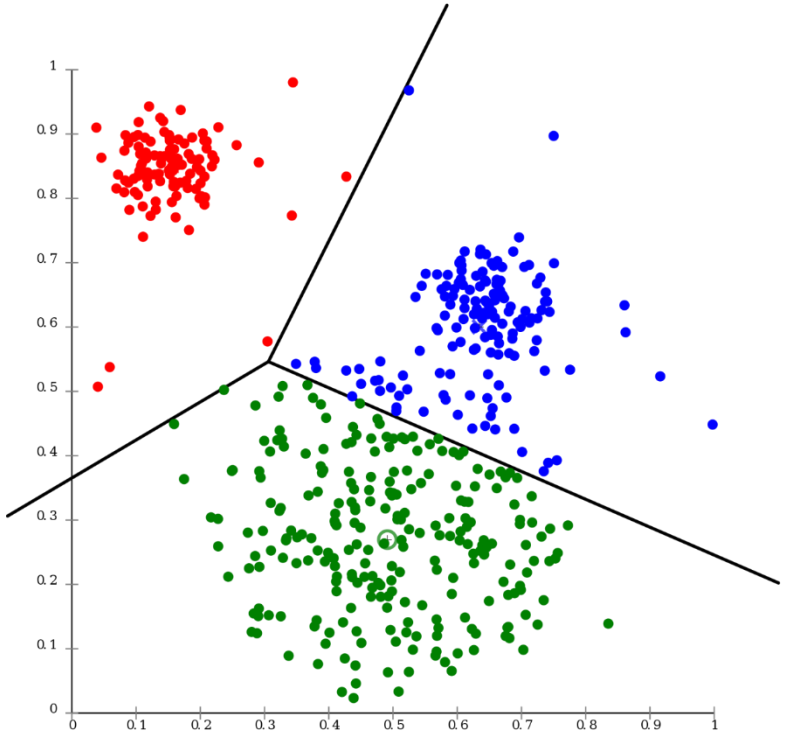


Image Segmentation



Unsupervised learning

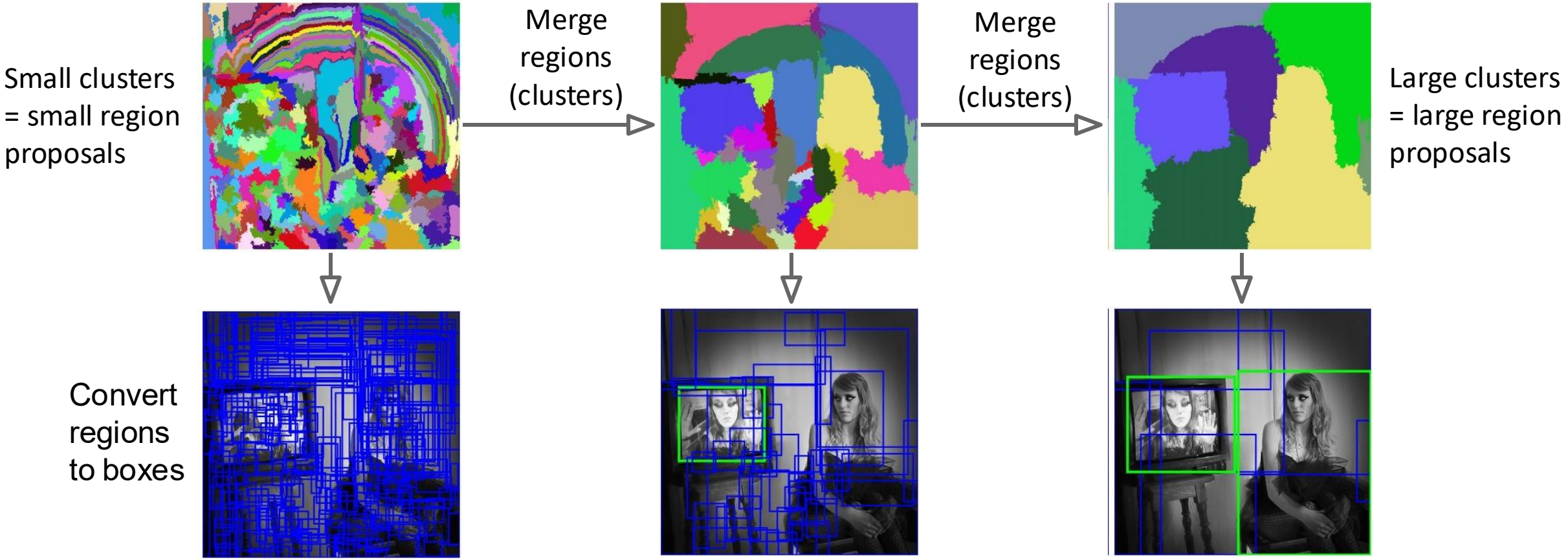
- **Data**: x (just data, no labels!)
- **Goal**: Learn some underlying hidden *structure* of the data
- **Examples**: Clustering, dimensionality reduction, feature learning, density estimation, etc.



Example:
K-means clustering

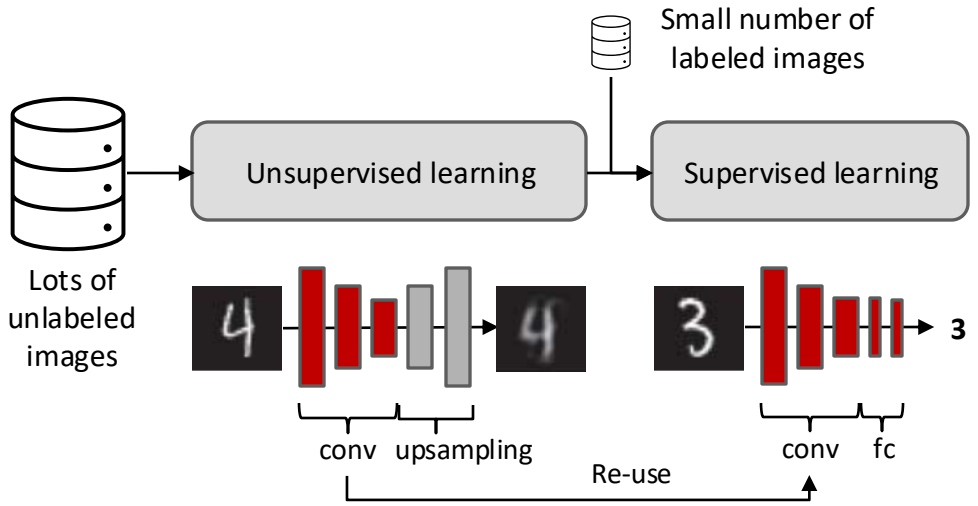
Example: Selective search

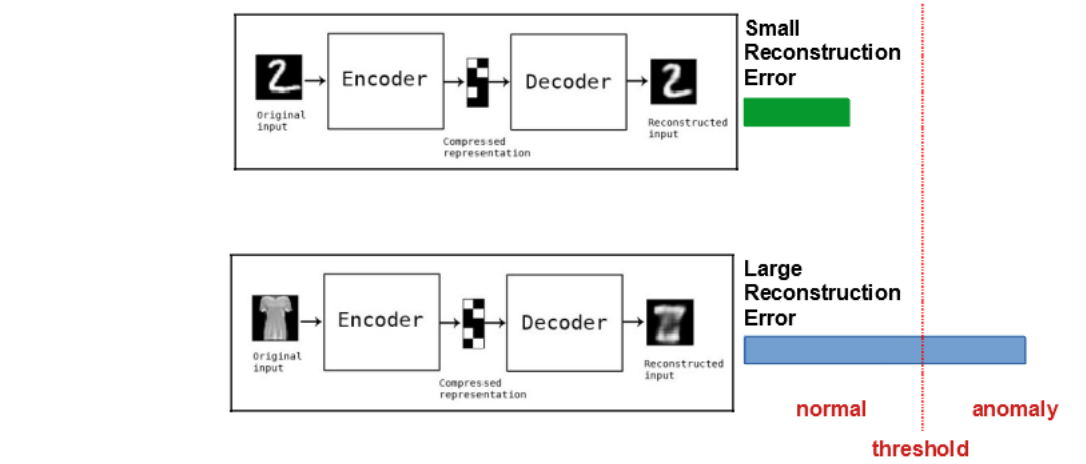
- Bottom-up clustering of similar pixels (unsupervised learning) for finding region proposals.



Why unsupervised learning?

- Common use-case: **Unsupervised pretraining**
  - You have lots of samples in your dataset, but only a few of them are labelled.
  - Pre-train an unsupervised CNN encoder on all samples.
  - Then fine-tune a supervised model on the labelled samples (this is just transfer learning).
- Another common use-case: **Anomaly detection**
  - Training: learn what the data distribution looks like
  - At test time, look for outliers (= anomalies)
- Today: **Generative models**
  - Given training data, generate new samples from same distribution.
  - The generative models you will learn about today are trained using unsupervised learning.





Generative models

Generative models

- A generative model is a class of statistical model that can generate new data instances.
- This contrasts with discriminative models, which discriminate between different kinds of data instances (e.g., a classifier).
- A generative model could generate new photos of birds that look like real birds, while a discriminative model could tell a bird from a cat.

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

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

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

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

Generative models

- More formally, given a set of data instances X and a set of labels Y:
  - **Generative** models capture the joint probability $p(X, Y)$, or just $p(X)$ if there are no labels.
  - **Discriminative** models capture the conditional probability $p(Y \mid X)$.
- A generative model includes the distribution of the data itself and **tells you how likely a given example is**.
- For example, large language models that **predict the next word in a sentence** are generative models, because they can assign a probability to a sequence of words.
- A discriminative model ignores the question of whether a given instance is likely and just tells you how likely a label is to apply to the instance.
- Note that this is a very general definition. **There are many kinds of generative models.**

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

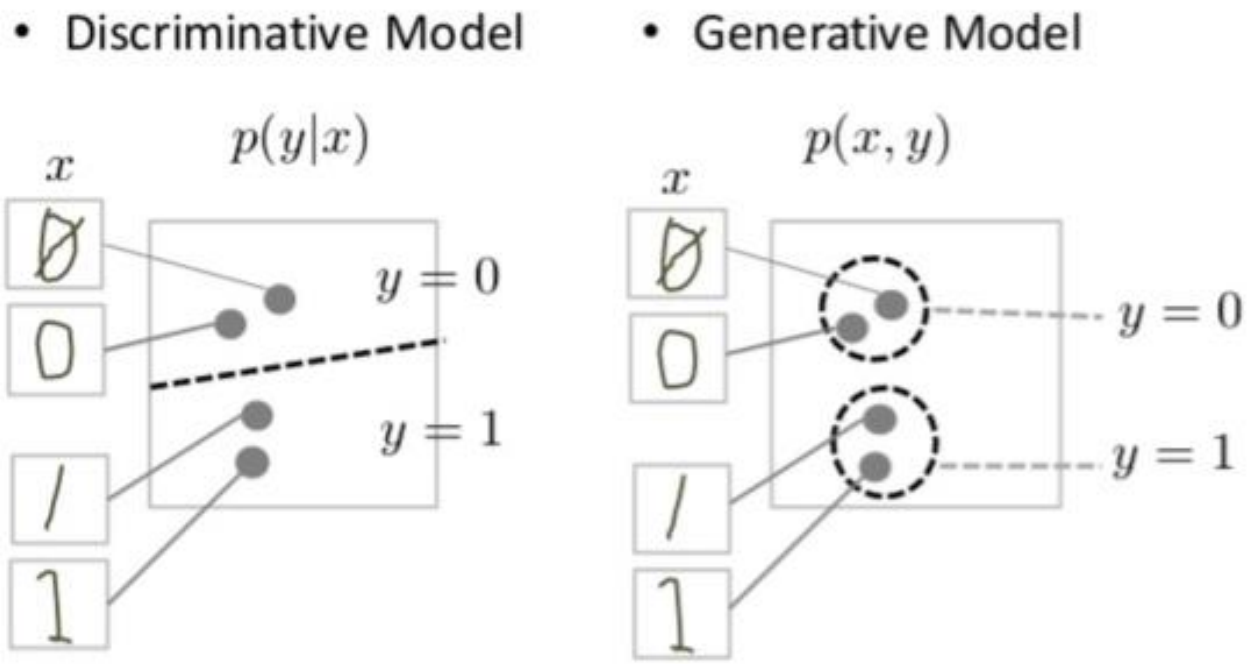
Generative modelling is hard

- **Generative models have to model more** – discriminative models try to draw boundaries in the data space, while generative models try to model how data is placed throughout the space.
- A **generative model** for images might capture correlations like *"things that look like boats are probably going to appear near things that look like water"* and *"eyes are unlikely to appear on foreheads."* **These are very complicated distributions.**
- In contrast, a **discriminative model** might learn the difference between "sailboat" and "not sailboat" just by looking for a few very specific features, like “sail”. It could ignore many of the correlations that the generative model must get right.

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

Generative modelling is hard

- The **discriminative model tries to tell the difference** between handwritten 0's and 1's by drawing a line in the data space. If it gets the line right, it can distinguish 0's from 1's **without ever having to model exactly where the instances are placed** in the data space on either side of the line.
- In contrast, the **generative model tries to produce convincing 1's and 0's** by generating digits that fall close to their real counterparts in the data space. It also has to model the distribution throughout the data space.



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

Autoencoder

Traditional autoencoder (AE)

- An autoencoder is a neural network that is trained to attempt to copy its input to its output.
- Given data x (no labels) we would like to learn functions f (encoder) and g (decoder) such that:

$$h = f(x)$$

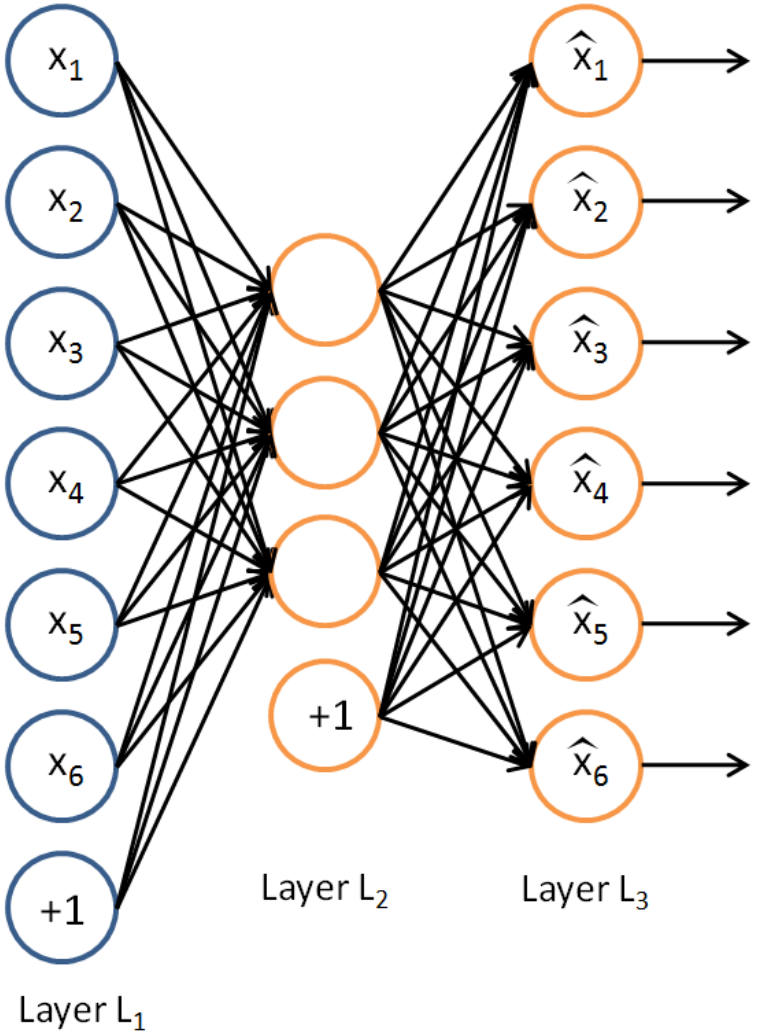
and

$$\hat{x} = g(h)$$

s.t.

$$\hat{x} = g\big(f(x)\big) = x$$

where $g\big(f(x)\big)$ is an **approximation** of the identity function



For more details. See chapter 14 in Ian Goodfellow's Deep Learning book

Traditional autoencoder (AE)

- An autoencoder is a neural network that is trained to attempt to copy its input to its output.
- Given data x (no labels) we would like to learn functions f (encoder) and g (decoder) such that:

$$h = f(x)$$

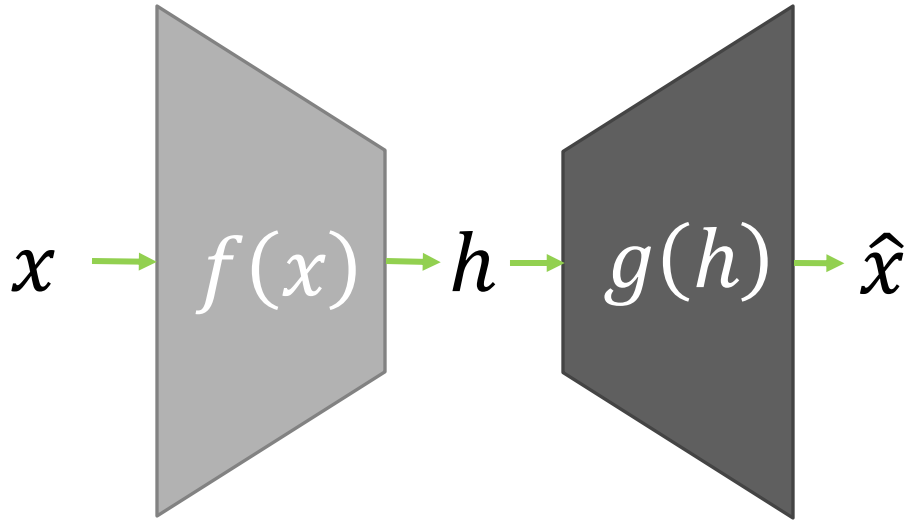
and

$$\hat{x} = g(h)$$

s.t.

$$\hat{x} = g\big(f(x)\big) = x$$

where $g\big(f(x)\big)$ is an **approximation** of the identity function



$h = f(x) = s(Wx + b)$ is the **latent or hidden representation** or **code** and s is a non-linearity such as the sigmoid.

$\hat{x} = g(h) = s(W^T h + b')$ is x's **reconstruction**.

Motivation

- Learning the identity function $g\big(f(x)\big) = x$ everywhere is not especially useful.
- Instead autoencoders are restricted in ways that allow them to **copy only approximately**.
- By adding constraints on the network (such as limiting the number of hidden neurons or regularization), **we can learn information about the structure of the data**.
- Traditionally an autoencoder is used for **dimensionality reduction** (i.e., finding correlations in the data) and **feature learning**.
- As we will see later, autoencoders can be used for generative modeling through latent (hidden) space modeling (basically by throwing away the encoder and using the decoder only).

Training an autoencoder

- The training set is just (no labels): $\left\{x_1^{(i)}, x_2^{(i)}, \ldots, x_m^{(i)}\right\}_{i=1}^{n}$

- Using SGD we can train the model as any other neural network.
- Traditionally with mean squared error loss function (L2 norm)

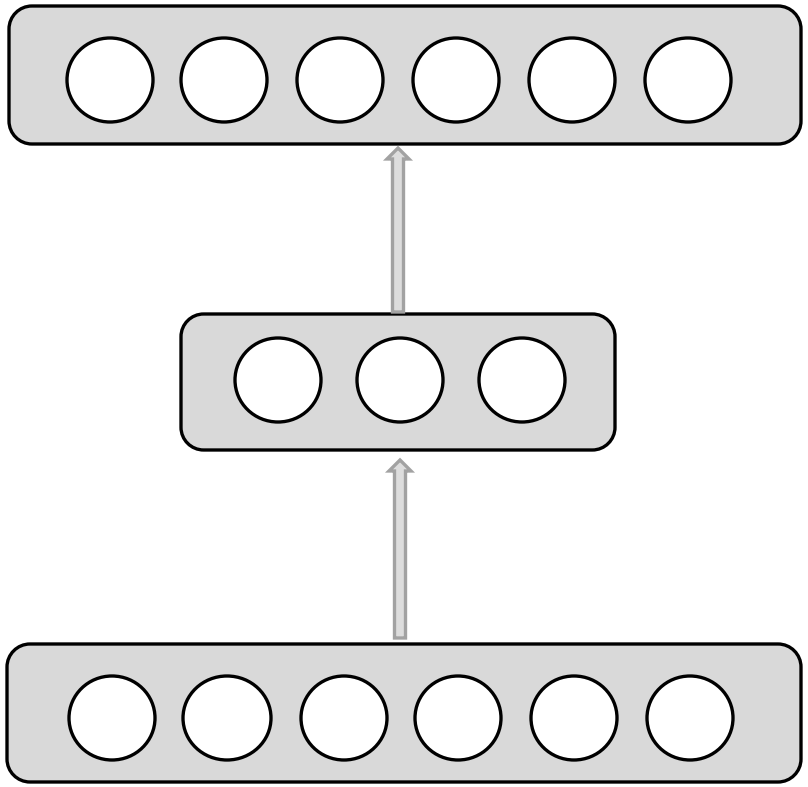
$$J(W) = \frac{1}{2}\sum_{i=1}^{n}\left(g\left(f\left(x^{(i)}\right)\right) - x^{(i)}\right)^2 = \frac{1}{2}\sum_{i=1}^{n}\left\|\hat{x}^{(i)} - x^{(i)}\right\|^2$$

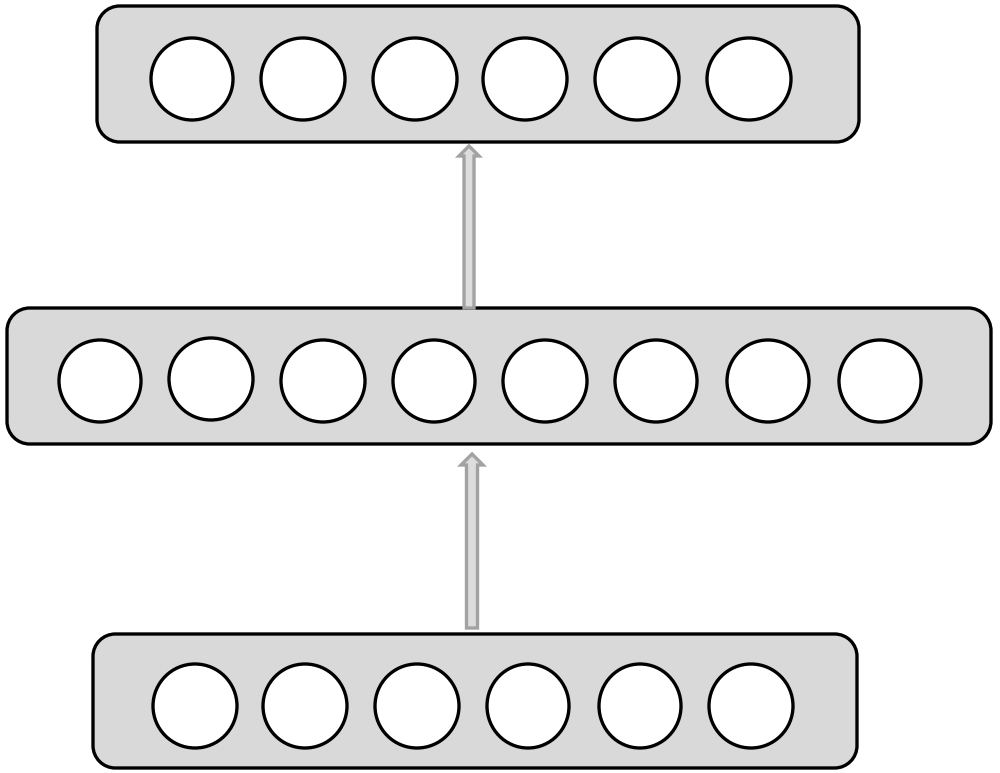
- If our input is interpreted as bit vectors or vectors of bit probabilities the cross entropy can be used

$$J(W) = -\sum_{i=1}^{n} x \log\left(g\left(f\left(x^{(i)}\right)\right)\right)$$

Undercomplete vs overcomplete AE

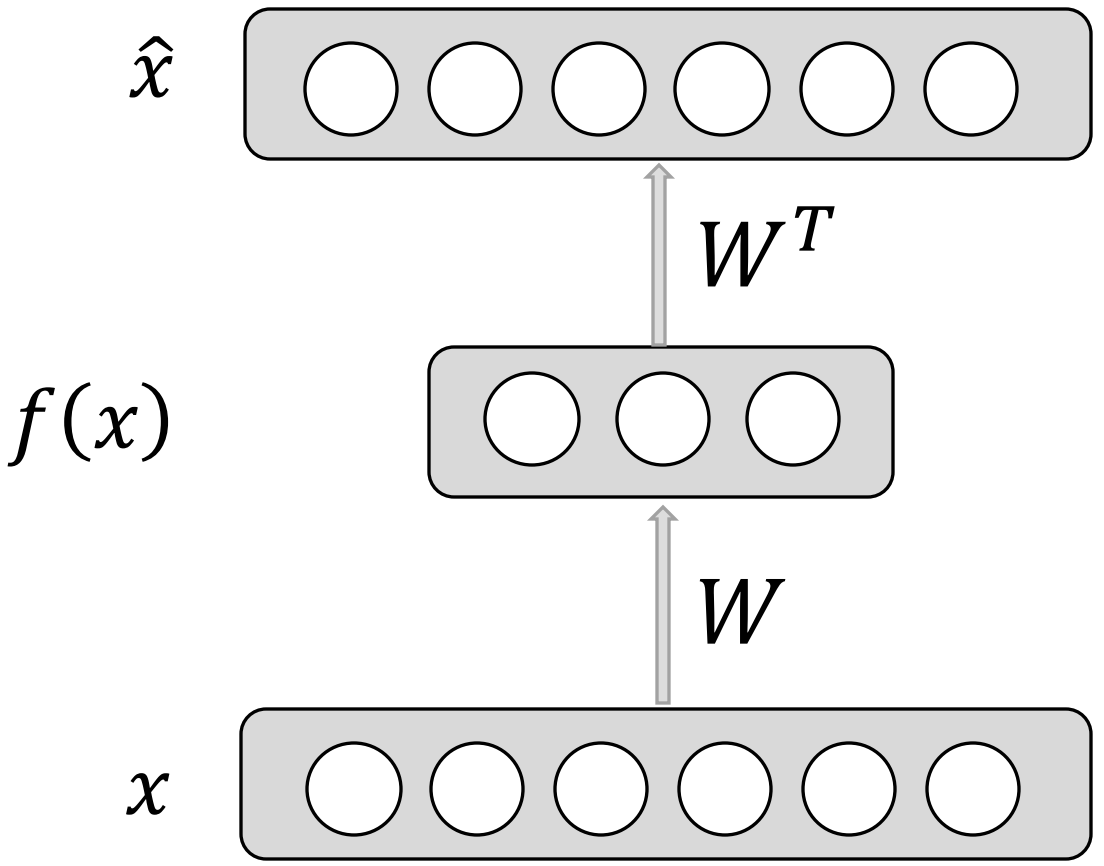
- We distinguish between two types of AE structures:





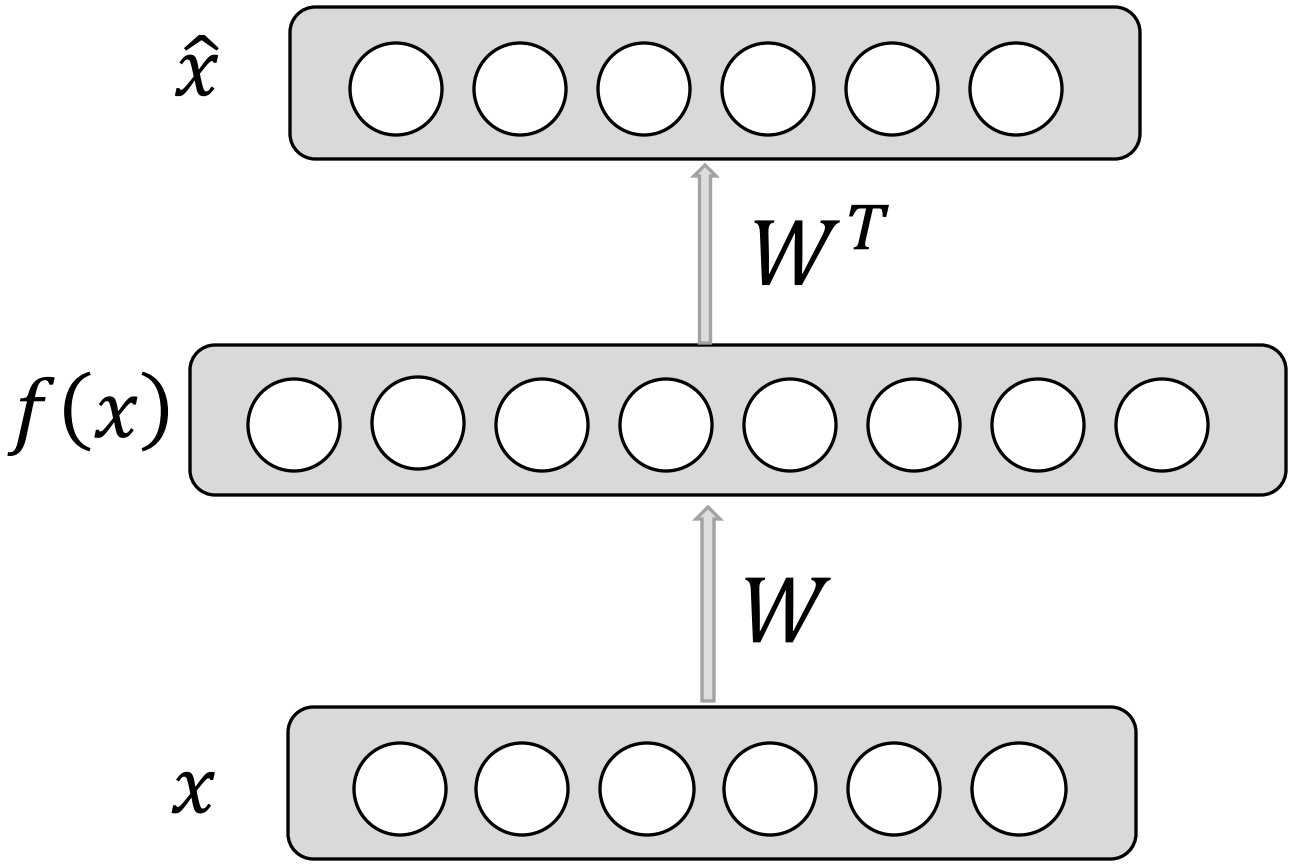
Undercomplete AE

- An AE is **undercomplete** if the hidden representation has smaller dimensionality than the input
  - Compresses the input (forced to find correlations in the data).
  - Compresses well only for the training distribution.
- Hidden nodes will be
  - Good features for the training distribution.
  - Bad for other types of input.
- Side-note:
  - Undercomplete AEs are commonly used for anomaly detection.
  - If $\hat{x}$ deviates too much from x, it is fair to assume that $\hat{x}$ lies outside the training distribution (meaning that it is probably an "anomaly").

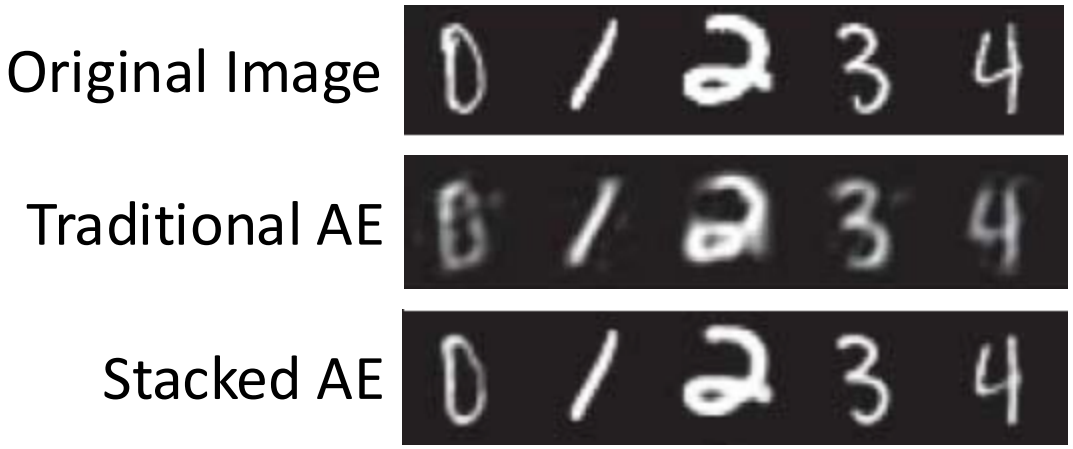


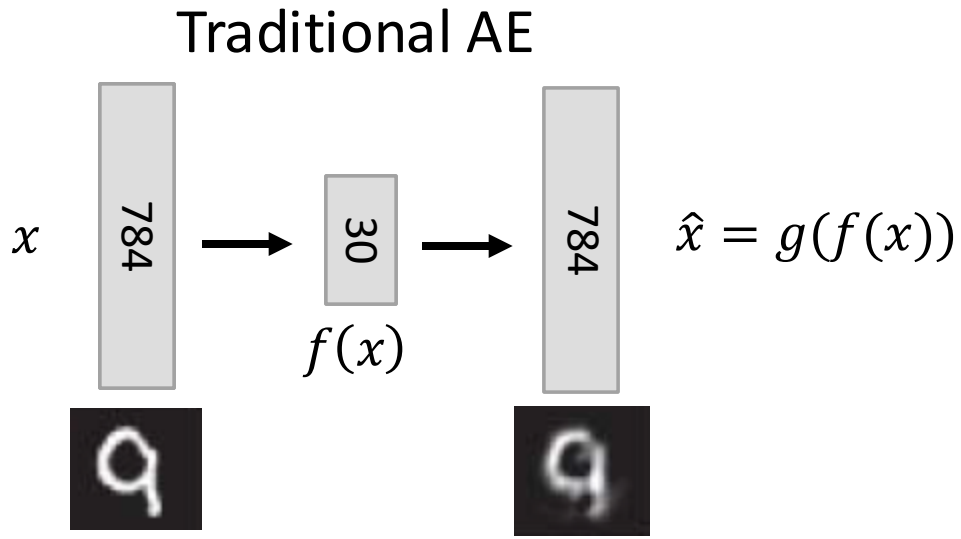
Overcomplete AE

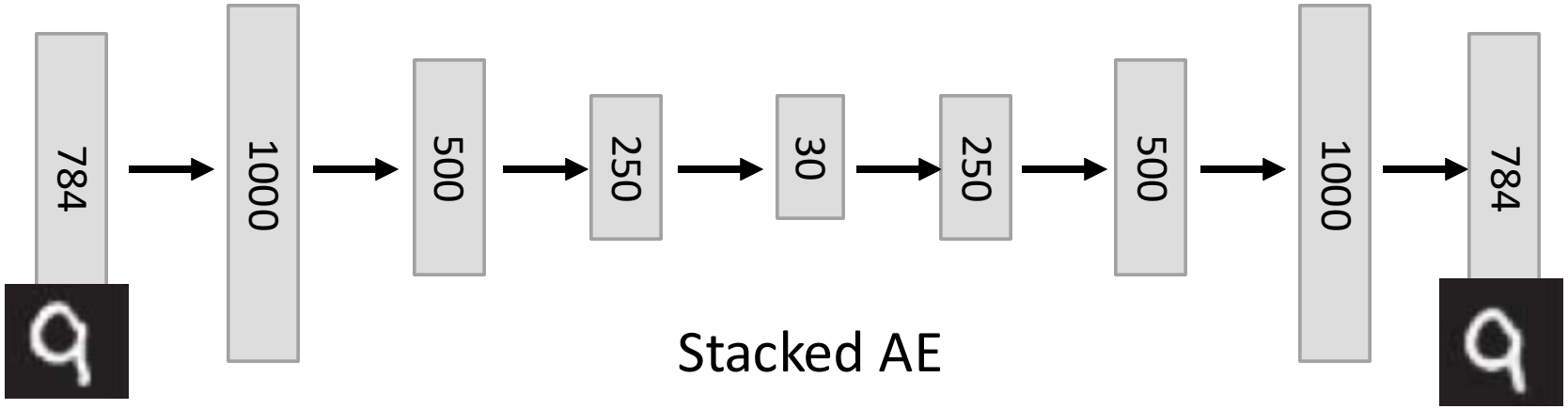
- An AE is **overcomplete** if the hidden representation has greater dimensionality than input
  - No compression in hidden layer.
  - Each hidden unit could copy a different input component.
- No guarantee that the hidden units will extract meaningful structure.
- A higher dimension code helps model a more complex distribution – good for training a linear classifier.



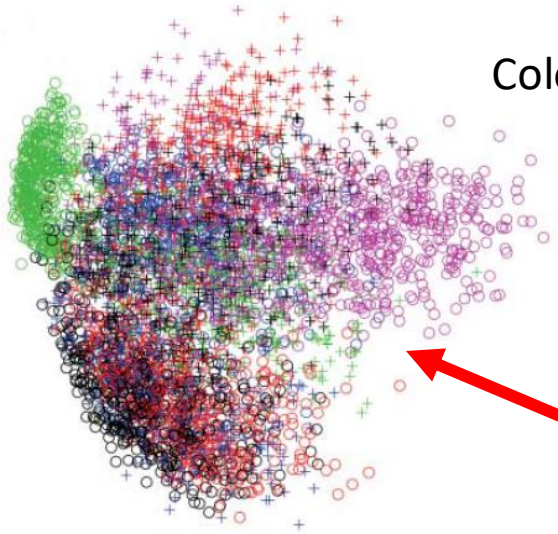
Stacked AE – going deep



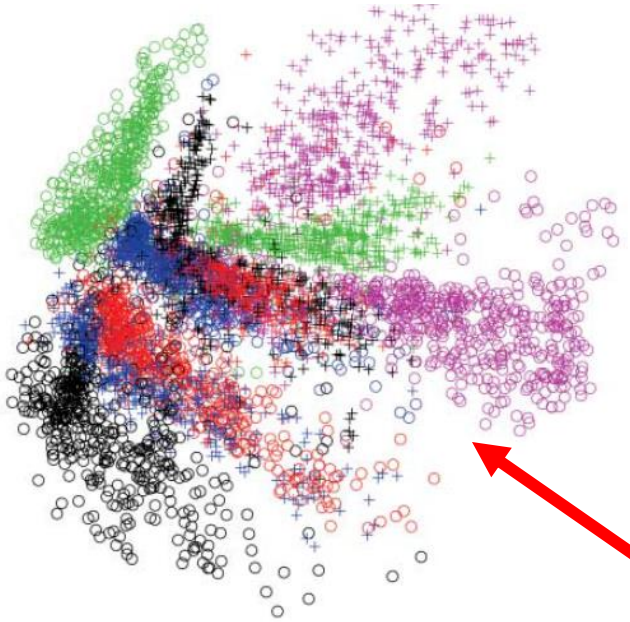




Deep is better

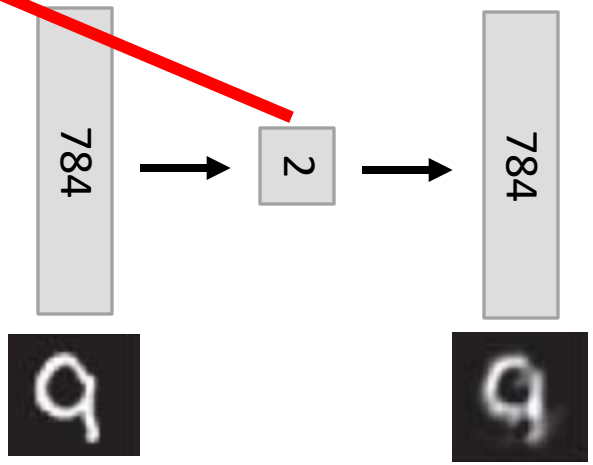


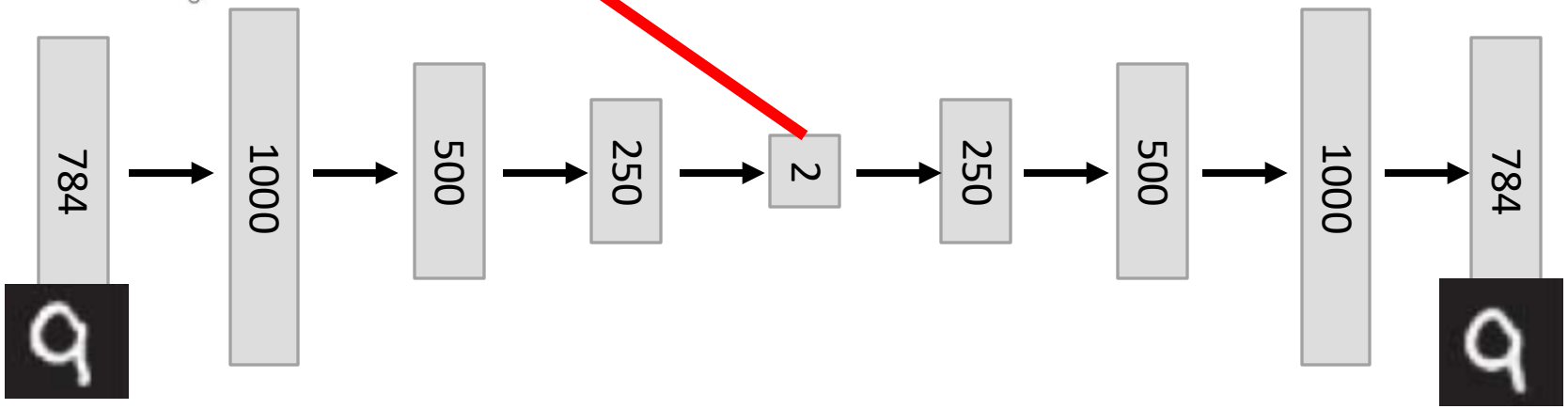
Colors represent MNIST classes



Deep is almost always better:

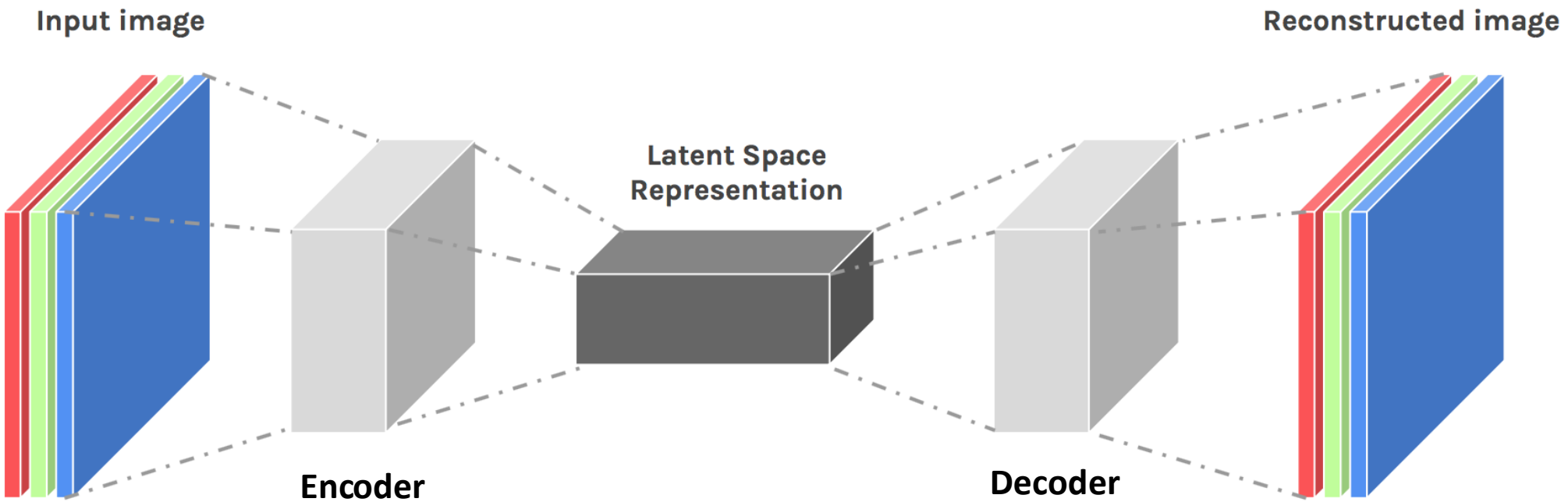
- More representational power
- Easier for similar datapoints to group together
- Better separation of classes





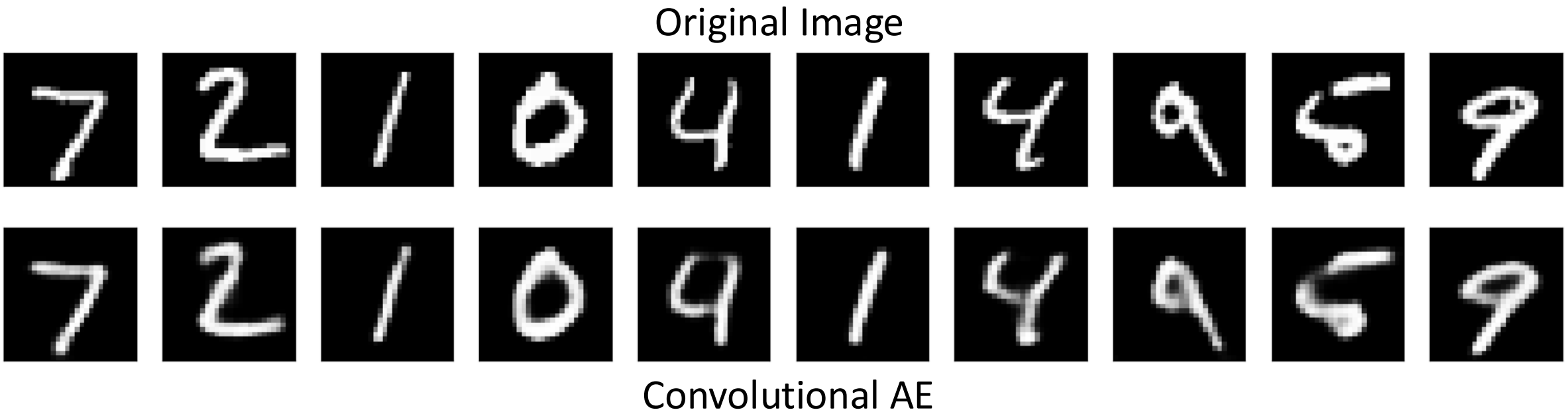
Going convolutional

- We can replace the fully connected layers with convolutional layers (better suited for images).



Example results on MNIST

- Almost perfect reconstruction (after 50 epochs). See lab 3.



Convolutional autoencoder in Keras

- The architecture consists of an **encoder** with convolutional layers + downsampling (e.g., max pooling), and a **decoder** with upsamling layers (sometimes mistakenly called deconvolution layers).
- At the **bottleneck** the image is sometimes converted into a vector.
- Different options for upsampling, but most often **transposed convolution**.
- You can read more about transposed convolution here (section 6): https://towardsdatascience.com/a-comprehensive-introduction-to-different-types-of-convolutions-in-deep-learning-669281e58215

```
Model: "model_4"
_________________________________________________________________
Layer (type)                 Output Shape              
=================================================================
input_4 (InputLayer)         (None, 28, 28, 1)         
_________________________________________________________________
conv2d_13 (Conv2D)           (None, 26, 26, 32)        
_________________________________________________________________
max_pooling2d_7 (MaxPooling2 (None, 13, 13, 32)        
_________________________________________________________________
conv2d_14 (Conv2D)           (None, 11, 11, 64)        
_________________________________________________________________
max_pooling2d_8 (MaxPooling2 (None, 5, 5, 64)          
_________________________________________________________________
conv2d_15 (Conv2D)           (None, 3, 3, 64)          
_________________________________________________________________
flatten_4 (Flatten)          (None, 576)               
_________________________________________________________________
dense_4 (Dense)              (None, 49)                
_________________________________________________________________
reshape_4 (Reshape)          (None, 7, 7, 1)           
_________________________________________________________________
conv2d_transpose_8 (Conv2DTr (None, 14, 14, 64)        
_________________________________________________________________
batch_normalization_8 (Batch (None, 14, 14, 64)        
_________________________________________________________________
conv2d_transpose_9 (Conv2DTr (None, 28, 28, 64)        
_________________________________________________________________
batch_normalization_9 (Batch (None, 28, 28, 64)        
_________________________________________________________________
conv2d_transpose_10 (Conv2DT (None, 28, 28, 32)        
_________________________________________________________________
conv2d_16 (Conv2D)           (None, 28, 28, 1)         
=================================================================
Total params: 140,850 Trainable params: 140,594 N
```

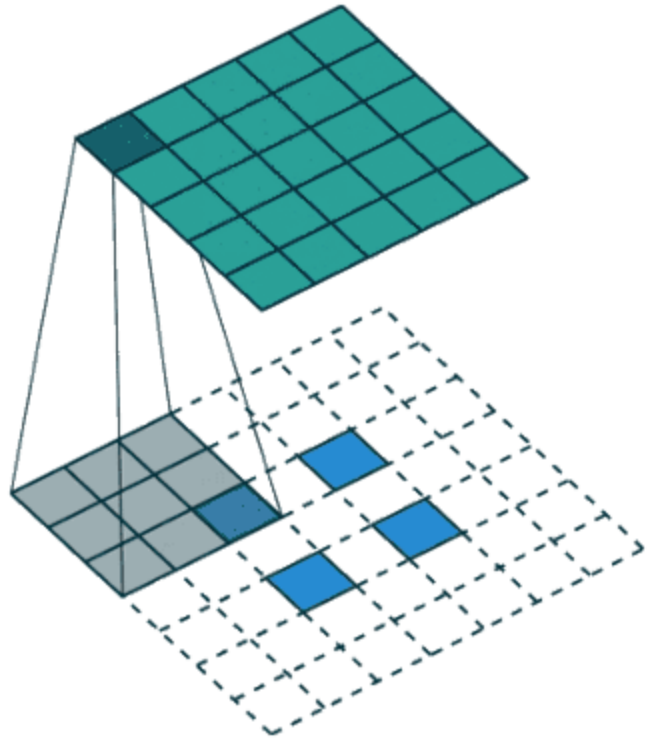
Encoder (downsampling)

Bottleneck

Decoder (upsampling)

Convolutional autoencoder in Keras

- The architecture consists of an **encoder** with convolutional layers + downsampling (e.g., max pooling), and a **decoder** with upsamling layers (sometimes mistakenly called deconvolution layers).
- At the **bottleneck** the image is sometimes converted into a vector.
- Different options for upsampling, but most often **transposed convolution**.
- You can read more about transposed convolution here (section 6): https://towardsdatascience.com/a-comprehensive-introduction-to-different-types-of-convolutions-in-deep-learning-669281e58215



Up-sampling a 2 x 2 input to a 5 x 5 output.

Convolutional autoencoder in Keras

```
#ENCODER
inp = Input((28, 28,1))
e = Conv2D(32, (3, 3), activation='relu')(inp)
e = MaxPooling2D((2, 2))(e)
e = Conv2D(64, (3, 3), activation='relu')(e)
e = MaxPooling2D((2, 2))(e)
e = Conv2D(64, (3, 3), activation='relu')(e)
l = Flatten()(e)
l = Dense(49, activation='relu')(l)
```

```
#DECODER
d = Reshape((7,7,1))(l)
d = Conv2DTranspose(64,(3, 3), strides=2, activation='relu', padding='same')(d)
d = BatchNormalization()(d)
d = Conv2DTranspose(64,(3, 3), strides=2, activation='relu', padding='same')(d)
d = BatchNormalization()(d)
d = Conv2DTranspose(32,(3, 3), activation='relu', padding='same')(d)
decoded = Conv2D(1, (3, 3), padding='same')(d)
ae = Model(inp, decoded)
```

https://medium.com/analytics-vidhya/building-a-convolutional-autoencoder-using-keras-using-conv2dtranspose-ca403c8d144e

Convolutional autoen

```
#ENCODER
inp = Input((28, 28,1))
e = Conv2D(32, (3, 3), activation='relu')(i
e = MaxPooling2D((2, 2))(e)
e = Conv2D(64, (3, 3), activation='relu')(e
e = MaxPooling2D((2, 2))(e)
e = Conv2D(64, (3, 3), activation='relu')(e
l = Flatten()(e)
l = Dense(49, activation='relu')(l)
```

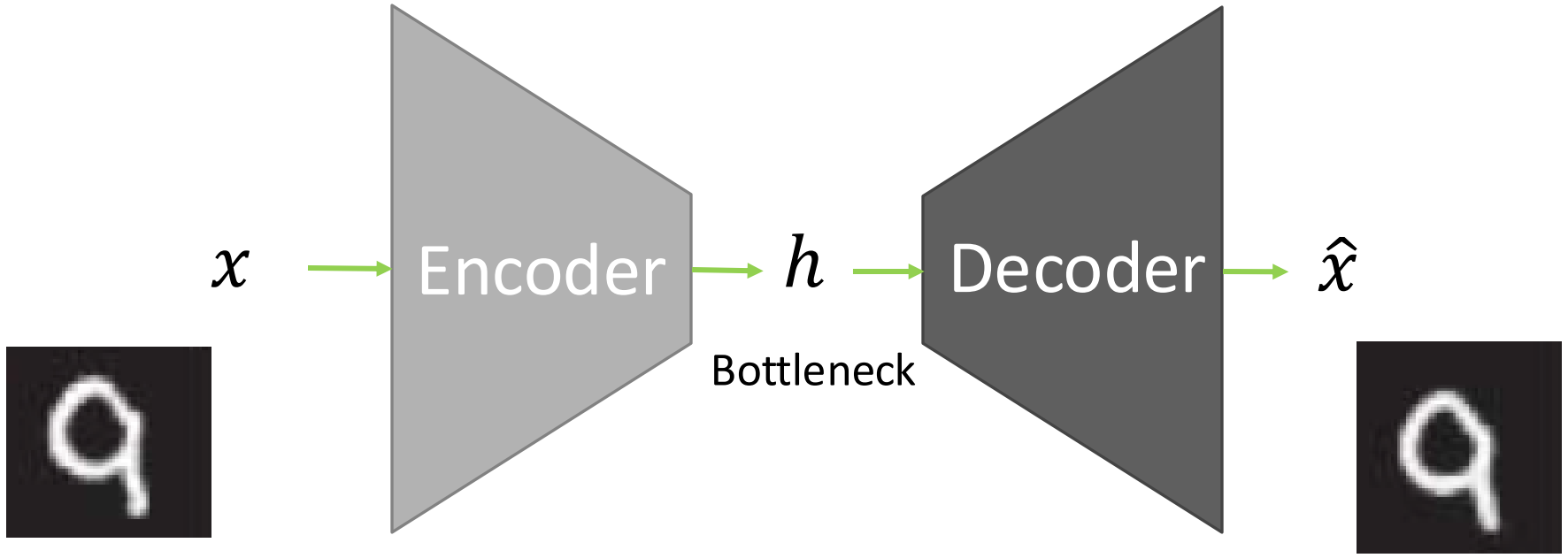
```
#DECODER
d = Reshape((7,7,1))(l)
d = Conv2DTranspose(64,(3, 3), strides=2, a
d = BatchNormalization()(d)
d = Conv2DTranspose(64,(3, 3), strides=2, a
d = BatchNormalization()(d)
d = Conv2DTranspose(32,(3, 3), activation='
decoded = Conv2D(1, (3, 3), padding='same')
ae = Model(inp, decoded)
```

```
Model: "model_4"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_4 (InputLayer)         (None, 28, 28, 1)         0
_________________________________________________________________
conv2d_13 (Conv2D)           (None, 26, 26, 32)        320
_________________________________________________________________
max_pooling2d_7 (MaxPooling2 (None, 13, 13, 32)        0
_________________________________________________________________
conv2d_14 (Conv2D)           (None, 11, 11, 64)        18496
_________________________________________________________________
max_pooling2d_8 (MaxPooling2 (None, 5, 5, 64)          0
_________________________________________________________________
conv2d_15 (Conv2D)           (None, 3, 3, 64)          36928
_________________________________________________________________
flatten_4 (Flatten)          (None, 576)               0
_________________________________________________________________
dense_4 (Dense)              (None, 49)                28273
_________________________________________________________________
reshape_4 (Reshape)          (None, 7, 7, 1)           0
_________________________________________________________________
conv2d_transpose_8 (Conv2DTr (None, 14, 14, 64)        640
_________________________________________________________________
batch_normalization_8 (Batch (None, 14, 14, 64)        256
_________________________________________________________________
conv2d_transpose_9 (Conv2DTr (None, 28, 28, 64)        36928
_________________________________________________________________
batch_normalization_9 (Batch (None, 28, 28, 64)        256
_________________________________________________________________
conv2d_transpose_10 (Conv2DT (None, 28, 28, 32)        18464
_________________________________________________________________
conv2d_16 (Conv2D)           (None, 28, 28, 1)         289
=================================================================
Total params: 140,850 Trainable params: 140,594 Non-trainable params:
```

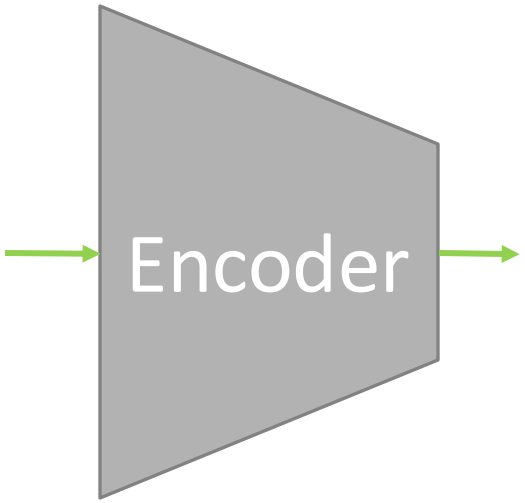
https://medium.com/analytics-vidhya/building-a-convolutional-autoencoder-using-keras-using-conv2dtranspose-ca403c8d144e

Latent space

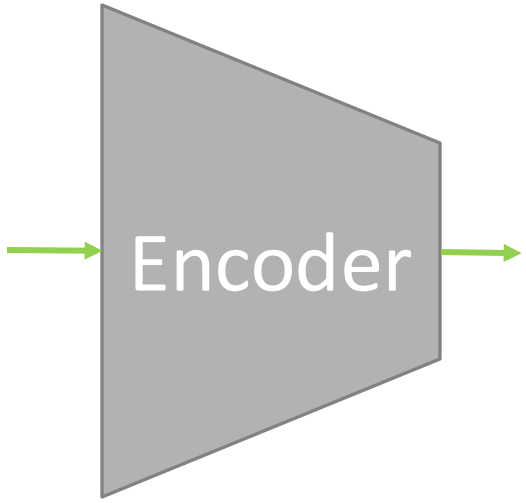
- The space in which the encoding exists is called the **latent** space or the **hidden** space.
- **Question:** How can we use the AE to generate new samples from the training distribution?
- **Idea:** Interpolate between training samples – but do it in the low-dimensional latent space, where we have captured all the important correlations in the data!



Latent space interpolation

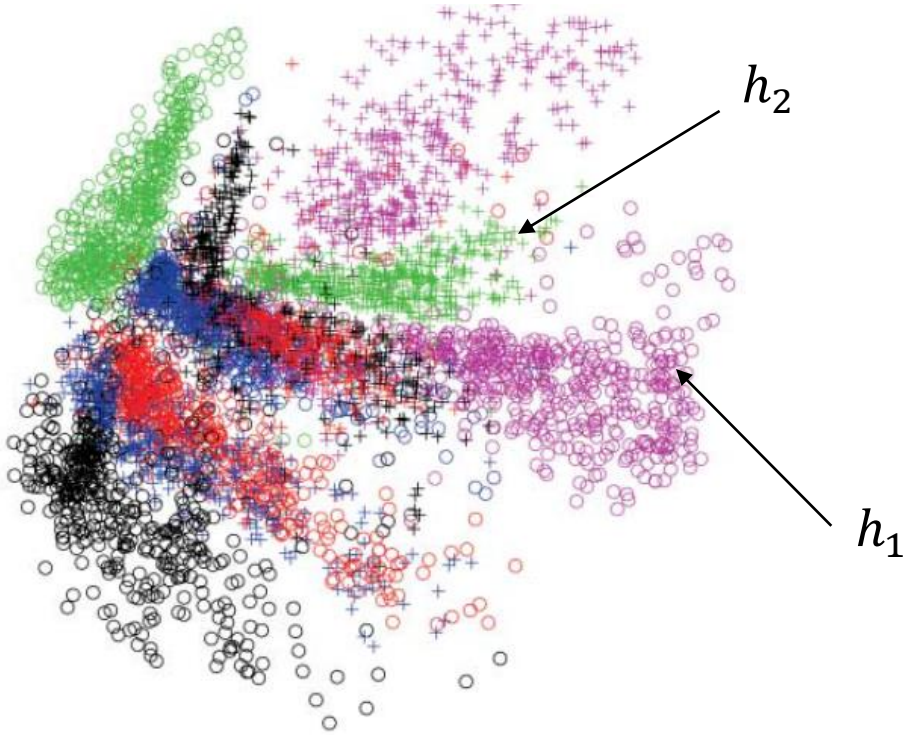


h_1



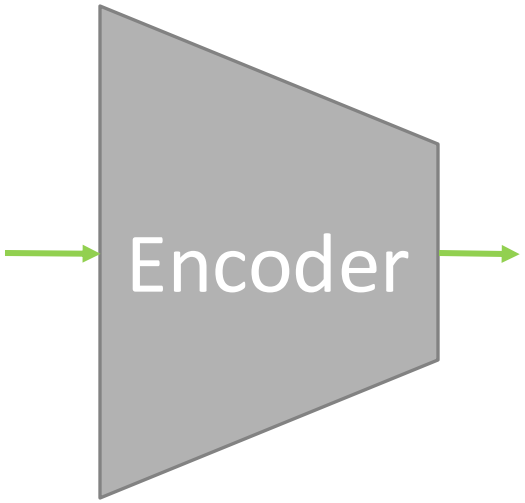
h_2

What happens if we try to interpolate between two latent vectors?

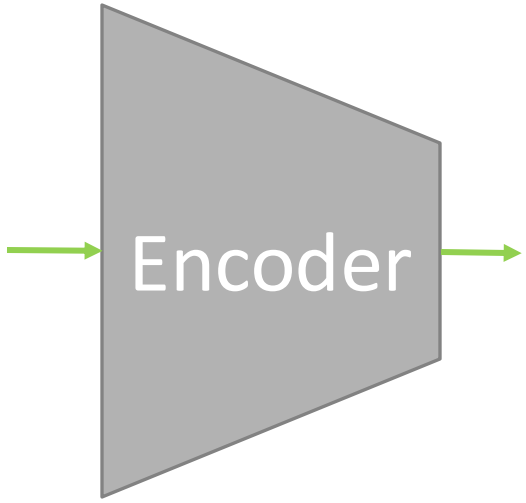


Latent space

Latent space interpolation

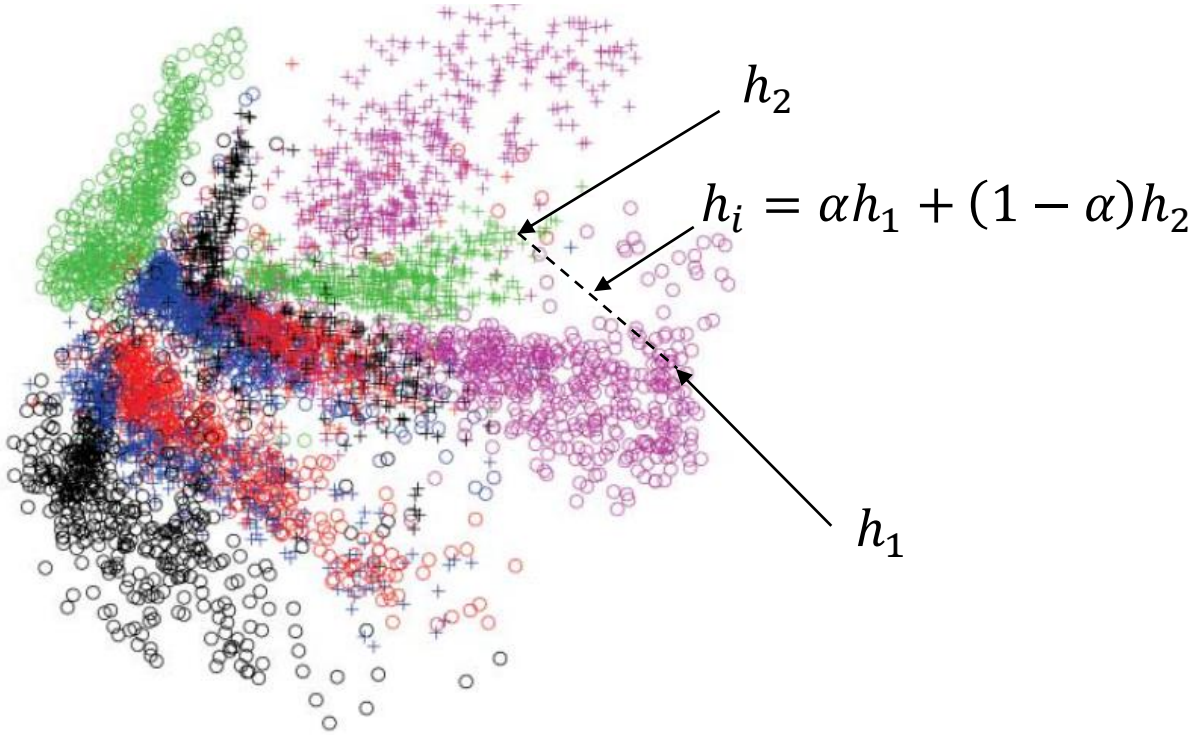


h_1



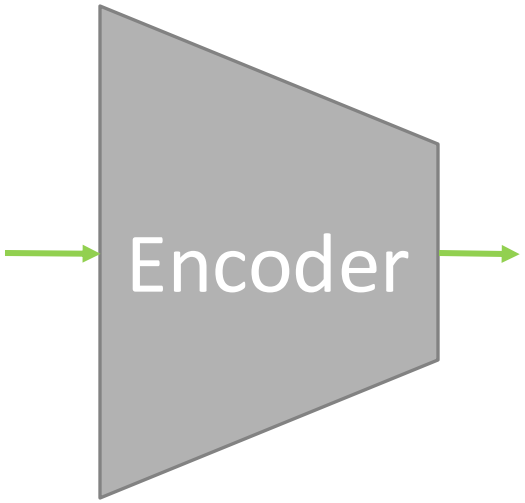
h_2

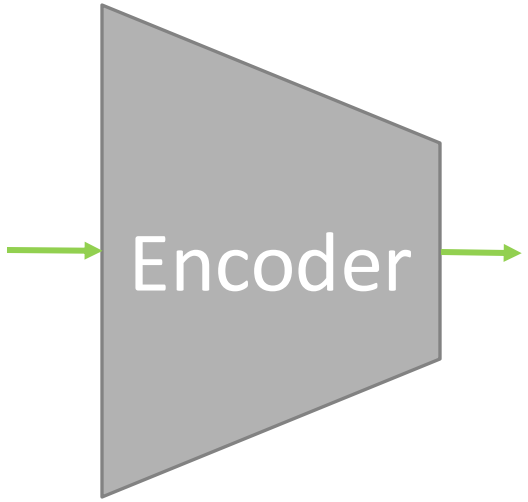
What happens if we try to interpolate between two latent vectors?



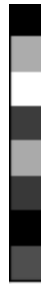
Latent space

Latent space interpolation





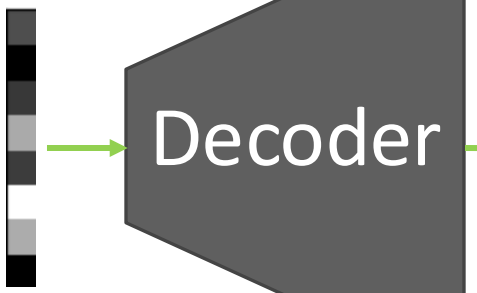
h_1



h_2

What happens if we try to interpolate between two latent vectors?

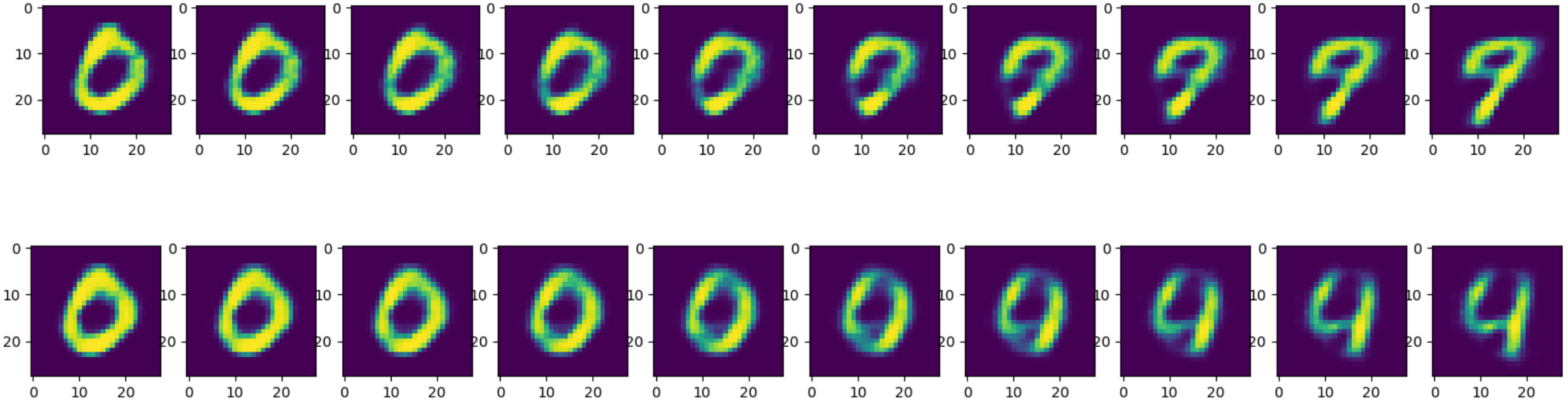
h_i



$$h_i = \alpha h_1 + (1 - \alpha) h_2$$

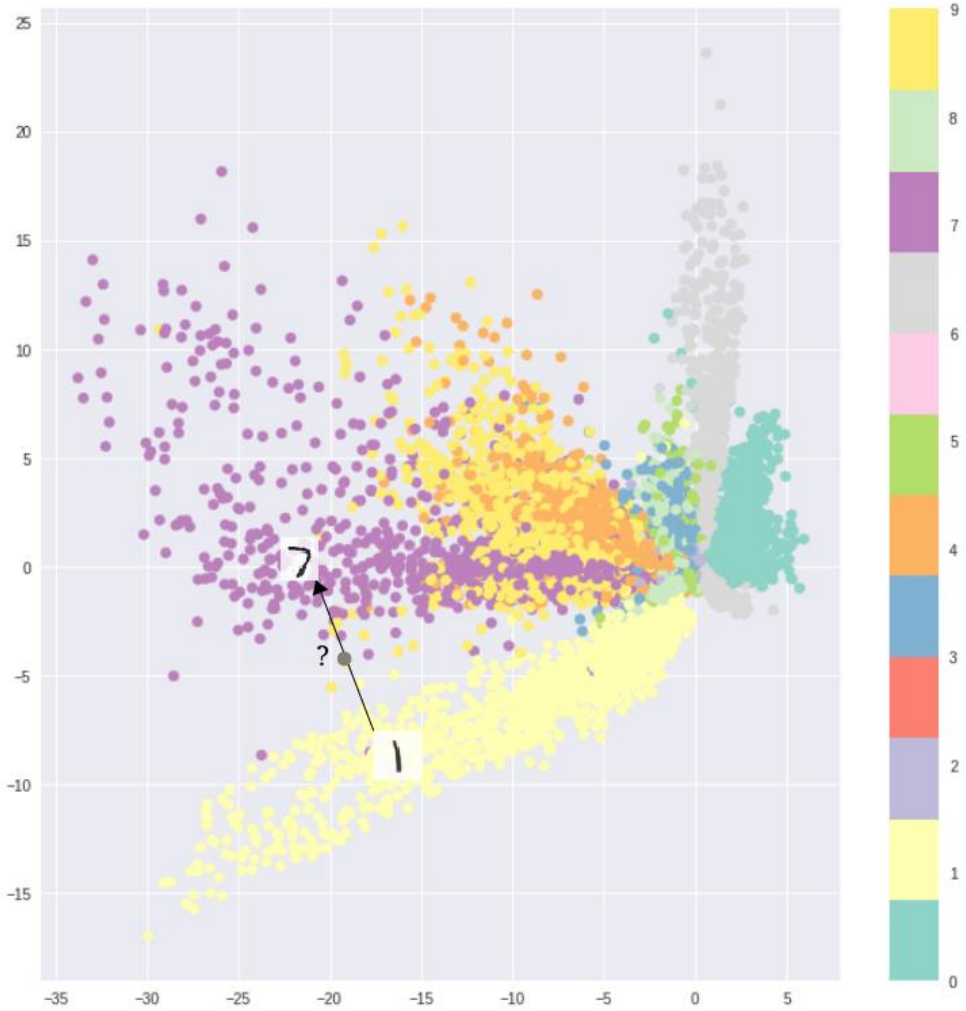
Latent space interpolation

$\alpha = 0$ $\alpha = 1$



Problem with simple autoencoders

- The fundamental problem with autoencoders, for generation, is that the latent space they convert their inputs to and where their encoded vectors lie, **may not be continuous**, or allow easy interpolation.
- For example, training an autoencoder on the MNIST dataset, and visualizing the encodings from a 2D latent space reveals the **formation of distinct clusters.**
- This makes sense, as distinct encodings for each image type makes it far easier for the decoder to decode them. **This is fine if you're just *replicating* the same images.**
- But when you're building a *generative* model, you **don't** want to prepare to *replicate* the same image you put in. You want to randomly sample from the latent space, or generate variations on an input image, from a continuous latent space.

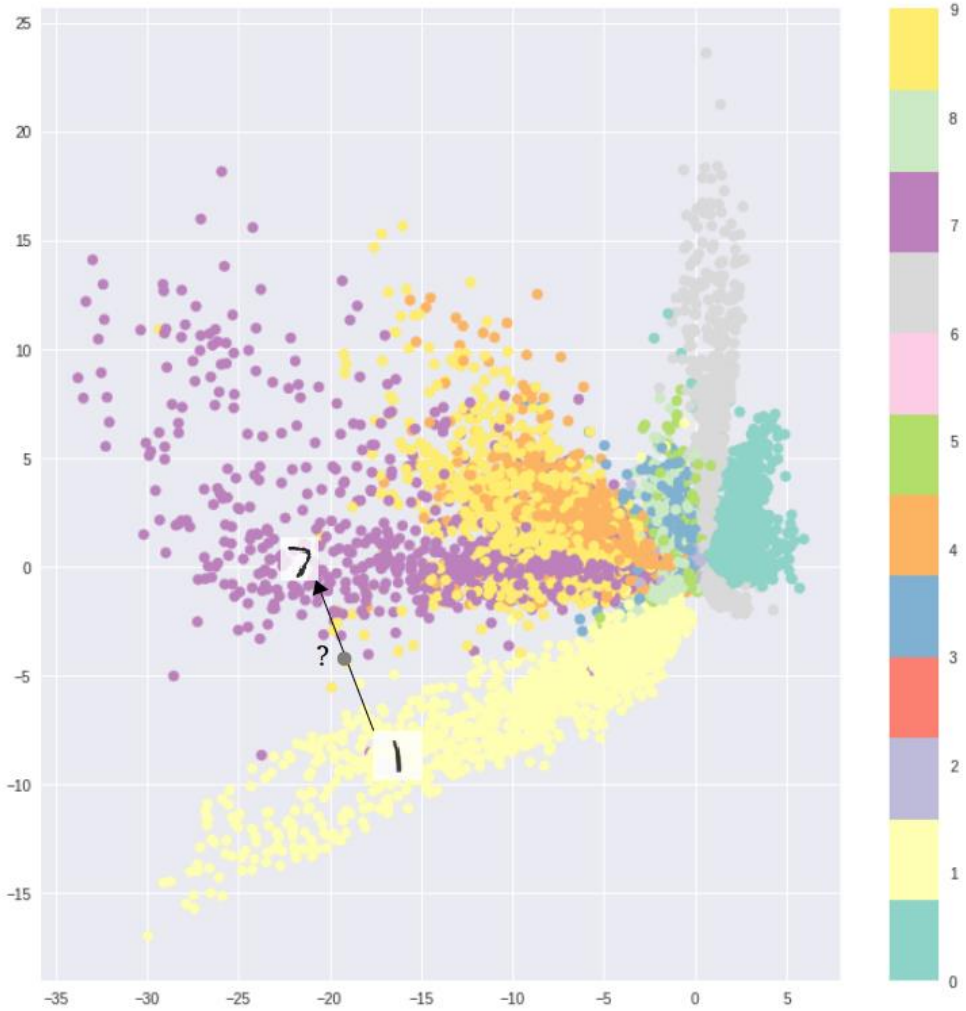


https://towardsdatascience.com/intuitively-understanding-variational-autoencoders-1bfe67eb5daf

Problem with simple autoencoders

- The fundamental problem with autoencoders, for generation, is that the latent space they convert their inputs to and where their encoded vectors lie, **may not be continuous**, or allow easy interpolation.
- For
ST
dat
late
ters.
- This
ima
dec
the
san
- But when you're building a *generative* model, you **don't** want to prepare to *replicate* the same image you put in. You want to randomly sample from the latent space, or generate variations on an input image, from a continuous latent space.

Solution:
Variational autoencoders (later)



https://towardsdatascience.com/intuitively-understanding-variational-autoencoders-1bfe67eb5daf

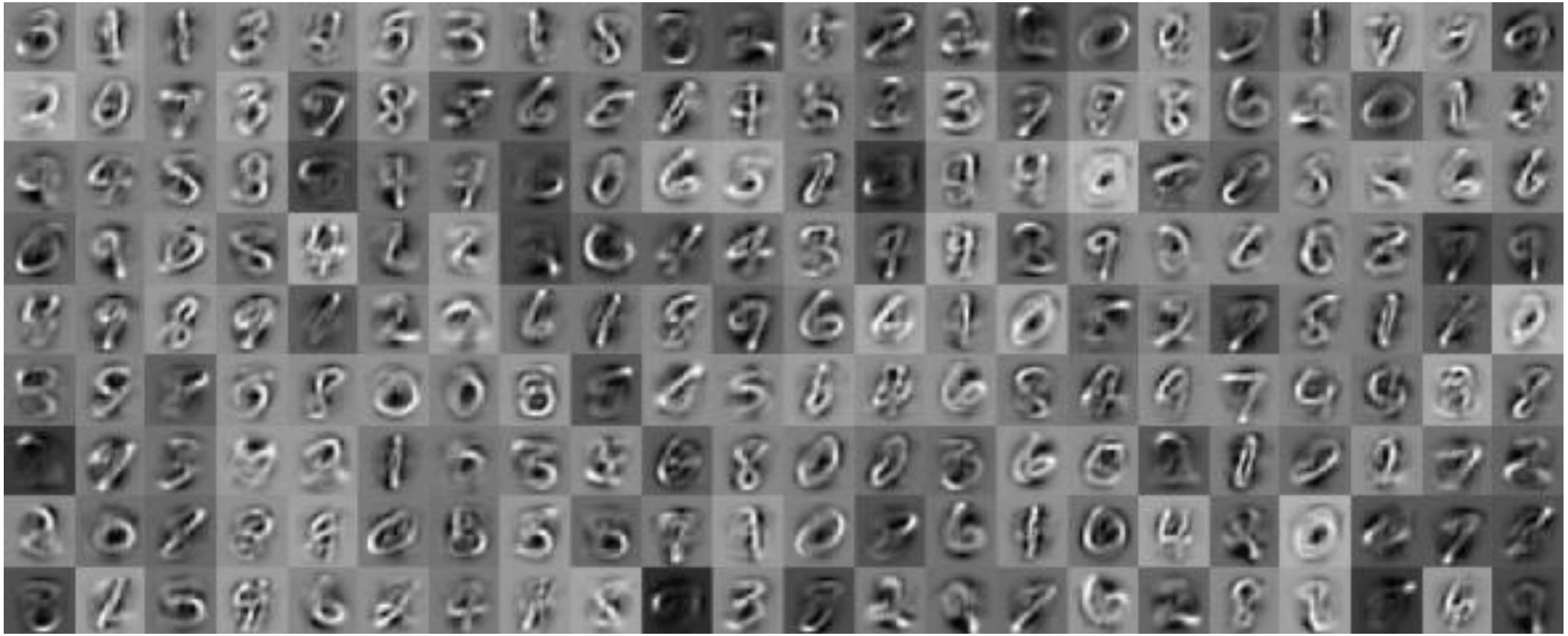
Regularization

Regularization

- **Motivation**
  - Like all deep neural networks, autoencoders are prone to overfitting.
  - We can reduce the risk of overfitting by reducing the dimensionality of the latent space, but that limits the capacity of the model (not able to model complex data distributions).
  - How can we learn good features without reducing the dimensionality of the latent space?
- **Solution**
  - Impose other constraints on the network.

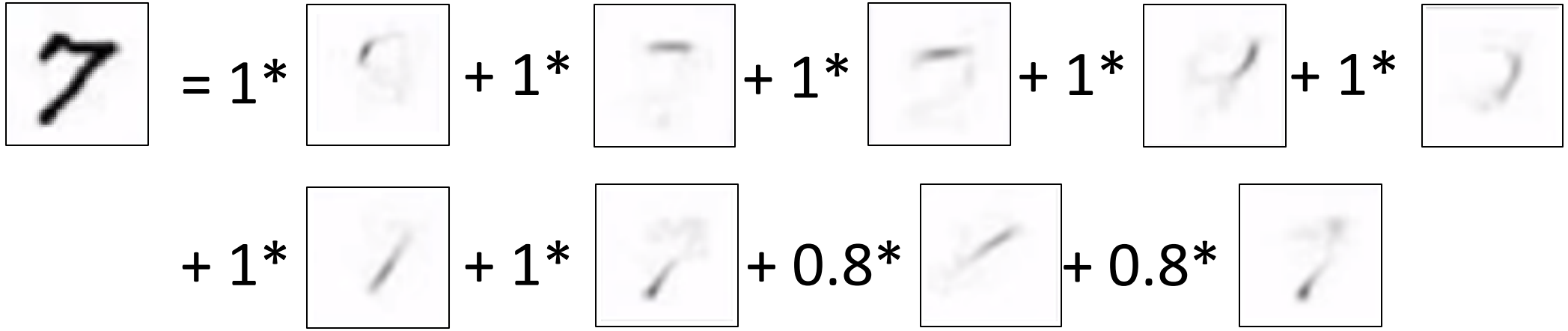
Sparse AE

- Suppose the autoencoder has learned this hidden representation (features). Clearly, it has just “memorized” the data, so no learning. This does not generalize well to unseen data.



Sparse AE

- This representation is much better – the 7 is decomposed into a combination of sparse features.
- If our learned features are sparse, we can generalize better.
- **Sparse means:** most activations are 0, except a few that are close to 1 (the zero-activations are not shown in the figure below).



Sparse AE

- Let $h_j^{(l_{Bn})}$ denote the activation of the j'th hidden unit (bottleneck) of the autoencoder.
- Let $h_j^{(l_{Bn})}(x)$ be the activation of this specific node on a given input x.
- Further let,

$$\hat{\rho}_j = \frac{1}{n}\sum_{i=1}^{n}\left[h_j^{(l_{Bn})}\left(x^{(i)}\right)\right]$$

- be the **average activation** of hidden unit j (over the training set).
- We would like to enforce the constraint $\hat{\rho}_j = \rho$, where ρ is a "sparsity parameter", typically small.
- In other words, we want the average activation of each neuron j to be close to ρ.

https://web.stanford.edu/class/cs294a/sparseAutoencoder.pdf

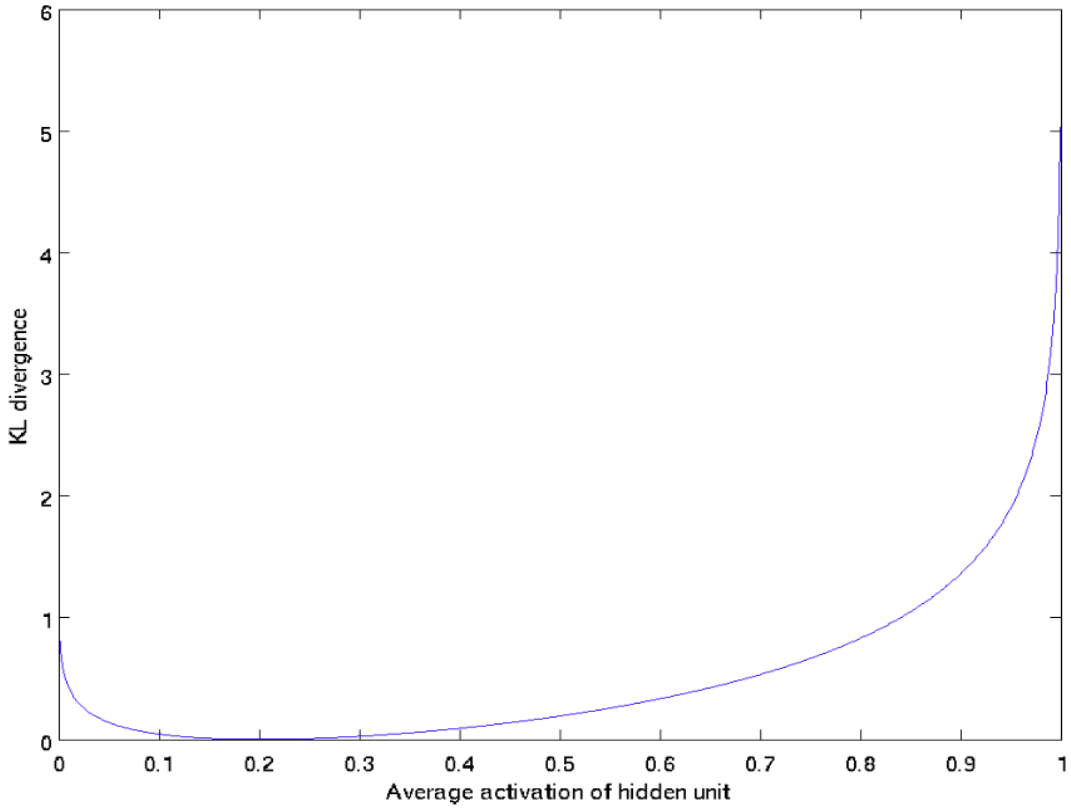
Sparse AE

- We need to penalize $\hat{\rho}_j$ for deviating from ρ.
- Many choices of the penalty term, for example:

$$\sum_{j=1}^{s_{Bn}} KL(\rho|\hat{\rho}_j)$$

- where $KL(\rho|\hat{\rho}_j)$ is a Kullback-Leibler divergence function and s_{Bn} is the number of units in the hidden layer.
- As a reminder: KL is a standard function for measuring the “distance “ between two probability distributions, with the property:

$$KL(\rho|\hat{\rho}_j) = 0 \text{ if } \hat{\rho}_j = \rho$$



$\rho = 0.2$

Sparse AE

- Our overall loss function is then:

Data loss term from earlier: $J(W) = \frac{1}{2}\sum_{i=1}^{n}\left\|\hat{x}^{(i)} - x^{(i)}\right\|^2$

$$J_S(W) = J(W) + \beta \sum_{j=1}^{s_{Bn}} KL\left(p|\hat{\rho}_j\right)$$

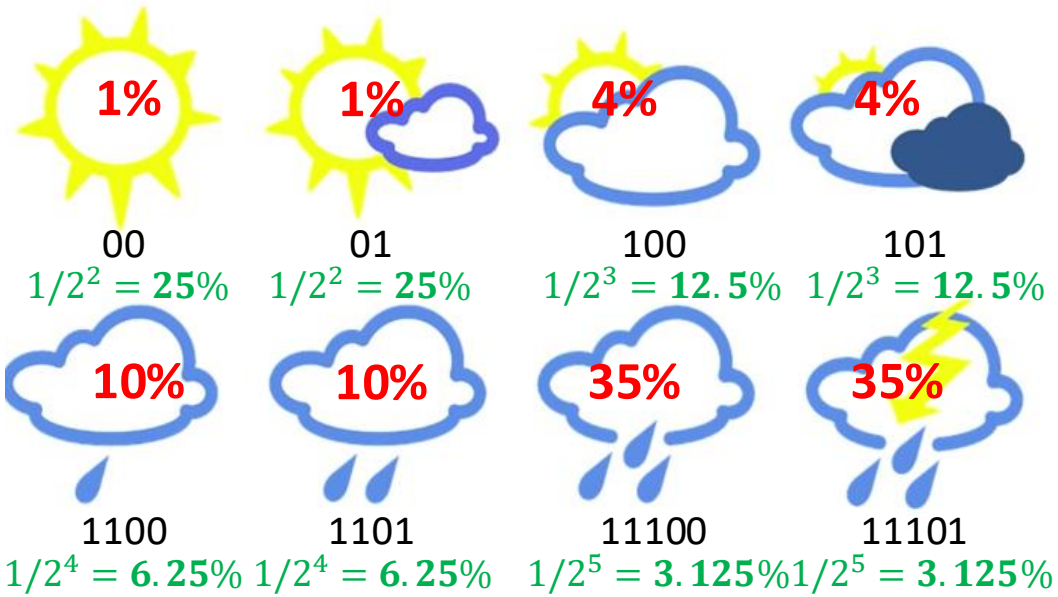
- Note: We need to know $\hat{\rho}_j$ before hand, so we have to compute a forward pass on the whole training set.

Recap

- p = true underlying distribution
- q = predicted distribution

Entropy:

$$H(\mathbf{p}) = -\sum_{i=1}^{n} p_i \log_2(p_i)$$
$$= -0.01 \times \log_2(0.01)$$
$$- 0.01 \times \log_2(0.01)$$
$$- \cdots$$
$$-0.35 \times \log_2(0.35)$$
$$= \mathbf{2.23\ bits}$$

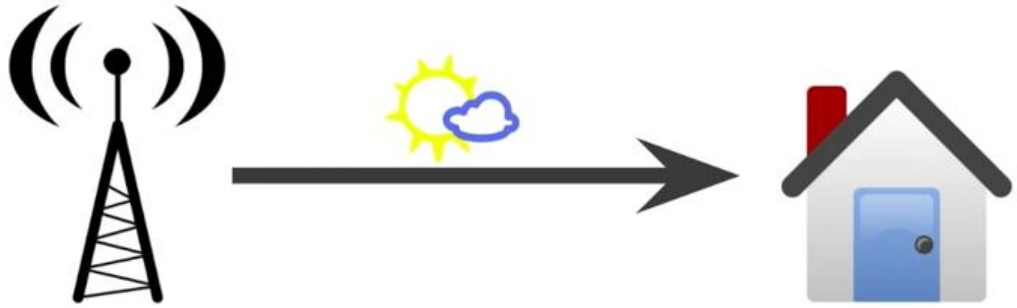


Cross-entropy:

$$H(\mathbf{p}, \mathbf{q})$$
$$= -\sum_{i=1}^{n} p_i \log_2(q_i)$$
$$= -0.01 \times \log_2(0.25)$$
$$- 0.01 \times \log_2(0.25)$$
$$- \cdots$$
$$-0.35 \times \log_2(0.0325)$$
$$= \mathbf{4.58\ bits}$$

KL Divergence:

$$D_{KL}(\mathbf{p} \parallel \mathbf{q}) = H(\mathbf{p}, \mathbf{q}) - H(\mathbf{p})$$
$$= 4.58 - 2.23 = \mathbf{2.35\ bits}$$



https://www.youtube.com/watch?v=ErfnhcEV1O8

Recap

- Note that if the predicted probabilities equal the true probabilities, then the cross-entropy is just the entropy.
- But if the two distributions differ, the cross-entropy will be larger than the entropy by some number of bits.
- The amount by which the cross-entropy exceeds the entropy, is called the relative entropy, or more commonly the Kullback-Leibler Divergence (KL Divergence)

Cross-entropy:

$$H(\mathbf{p}, \mathbf{q}) = -\sum_{i=1}^{n} p_i \log_2(q_i)$$

- p = true underlying distribution
- q = predicted distribution

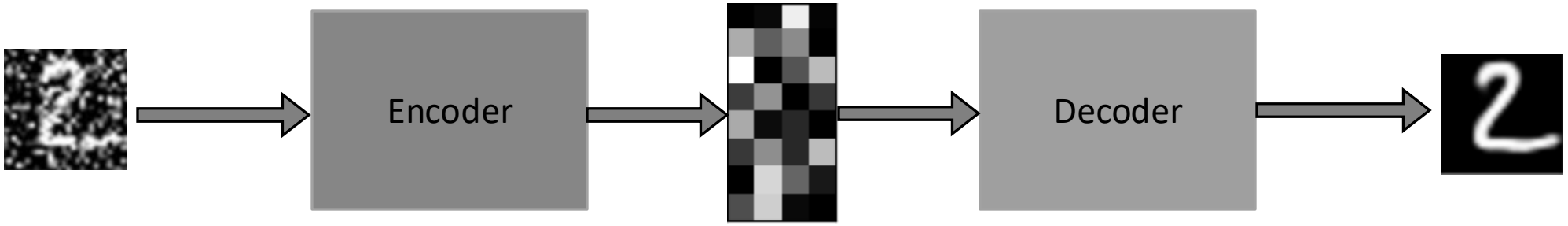
KL Divergence

$$D_{KL}(\mathbf{p} \parallel \mathbf{q}) = H(\mathbf{p}, \mathbf{q}) - H(\mathbf{p}) = -\sum_{i=1}^{n} p_i \log_2(q_i) - \left(-\sum_{i=1}^{n} p_i \log_2(p_i)\right) = -\sum_{i=1}^{n} p_i \log_2\left(\frac{q_i}{p_i}\right)$$

Denoising AE (DAE)

- We still aim to encode the input to **learn and describe latent attributes of the data**.
- In addition, we try to undo the effect of a corruption process stochastically applied to the input.
- With this approach, **our model isn't able to simply develop a mapping which memorizes the training data** because our input and target output are no longer the same.

A more robust model



Noisy Input

Latent space representation

Denoised Input

https://www.jeremyjordan.me/autoencoders/

Denoising AE (DAE)

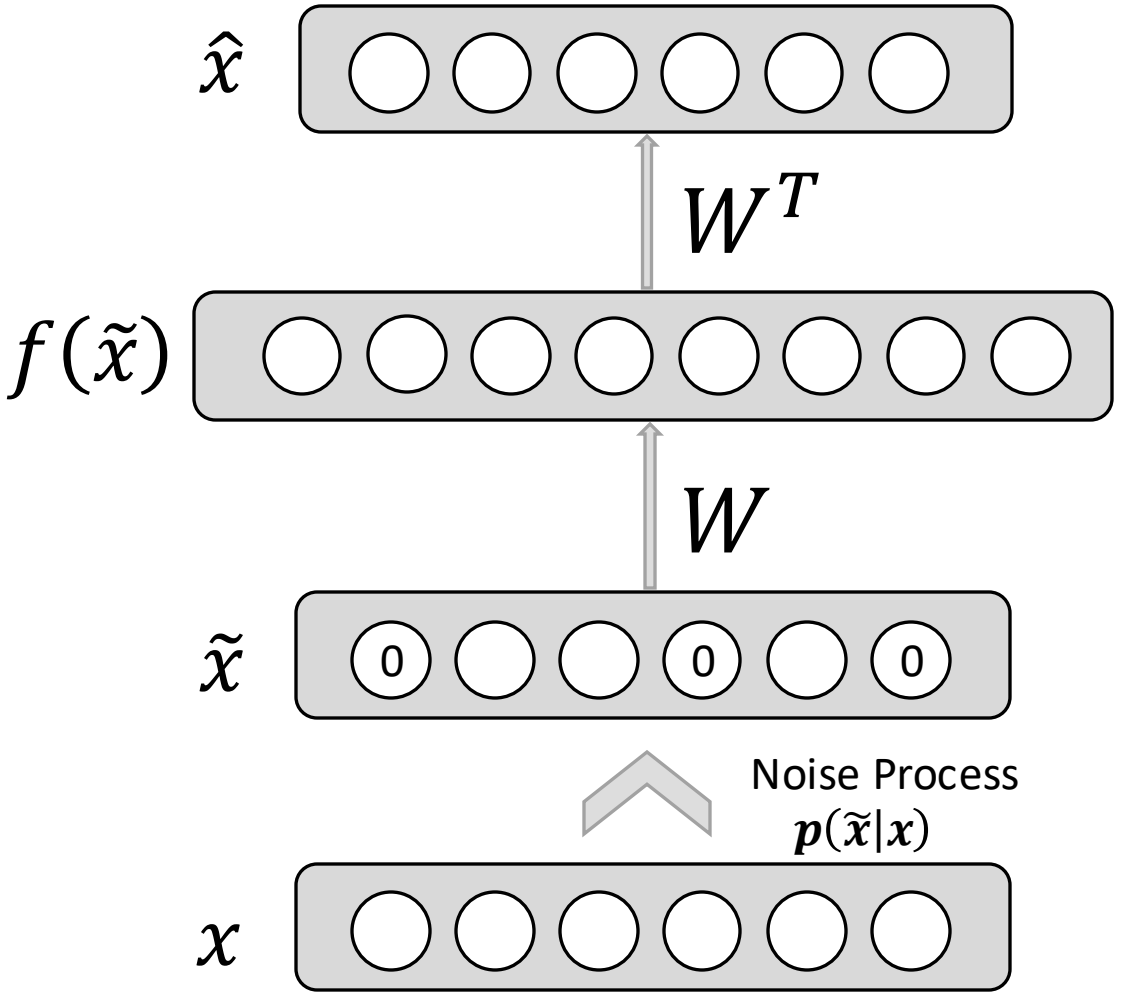
- Instead of simply trying to learn the identity function

$$\hat{x} = g\big(f(x)\big) = x$$

- We learn to "undo" noise corruption

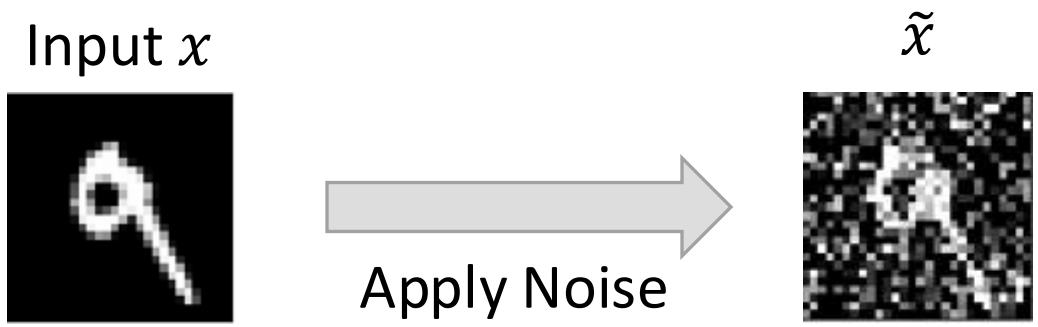
$$\hat{x} = g\big(f(\tilde{x})\big) = x$$

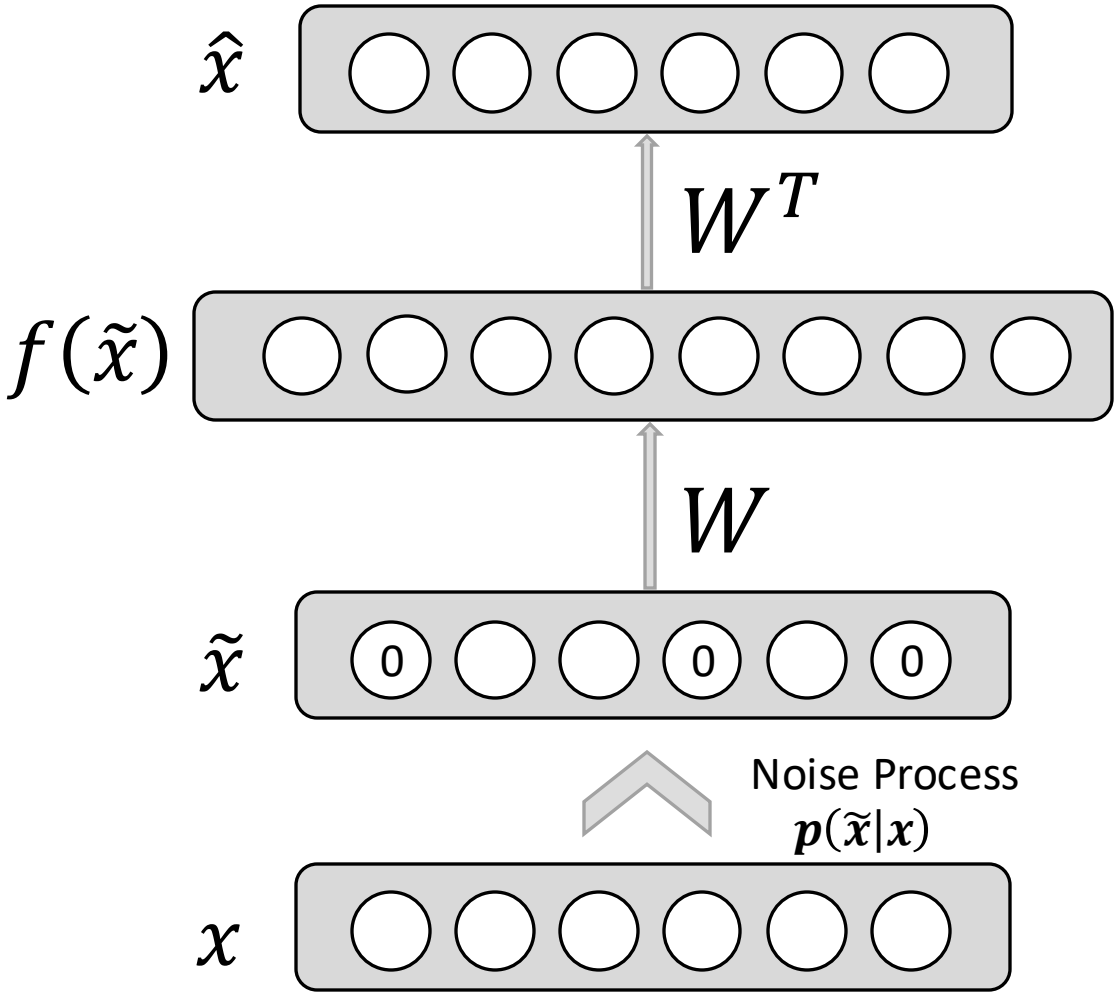
where $\tilde{x}$ is a copy of x that has been corrupted by some noise process.



Denoising AE (DAE)

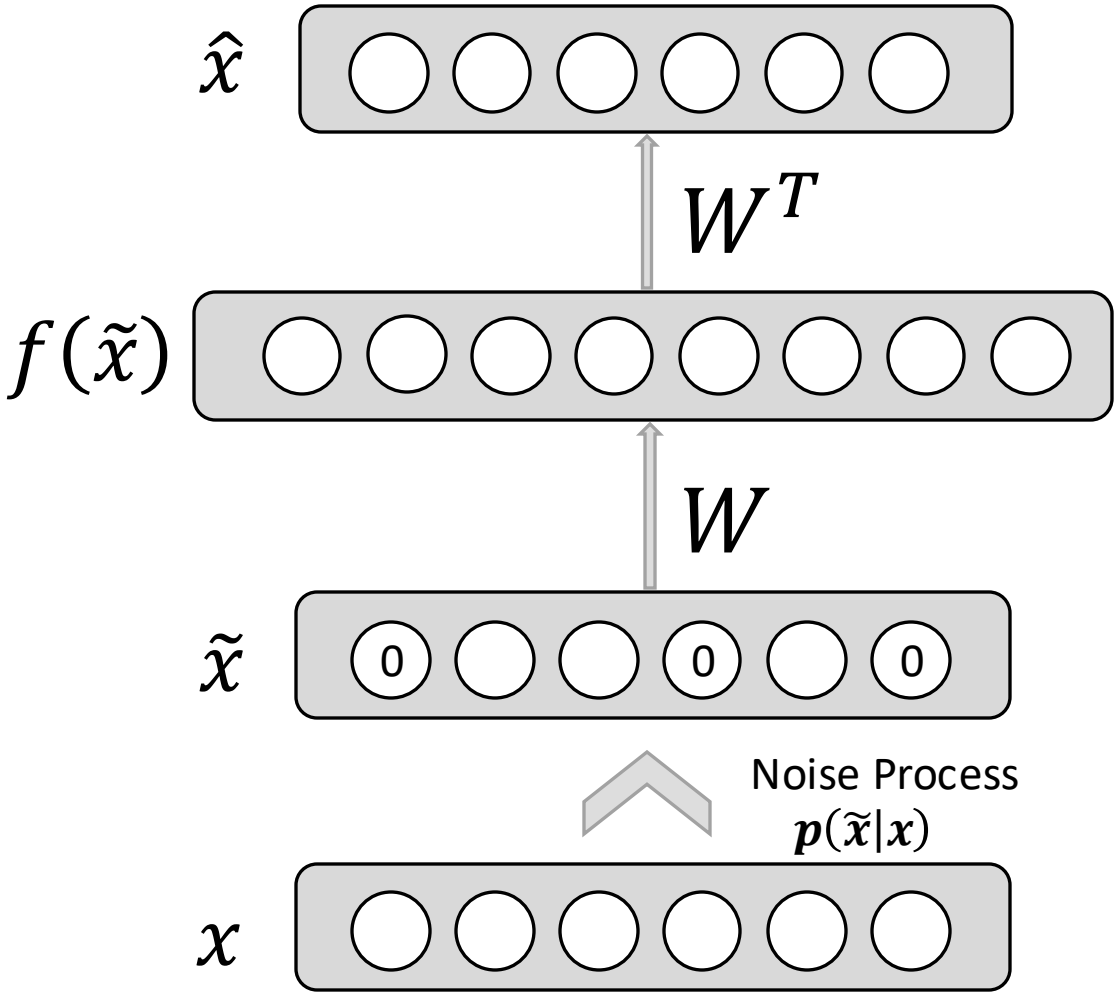
- Examples of noise
  - Random assignment of subset of inputs to 0, with probability v.
  - Gaussian additive noise.





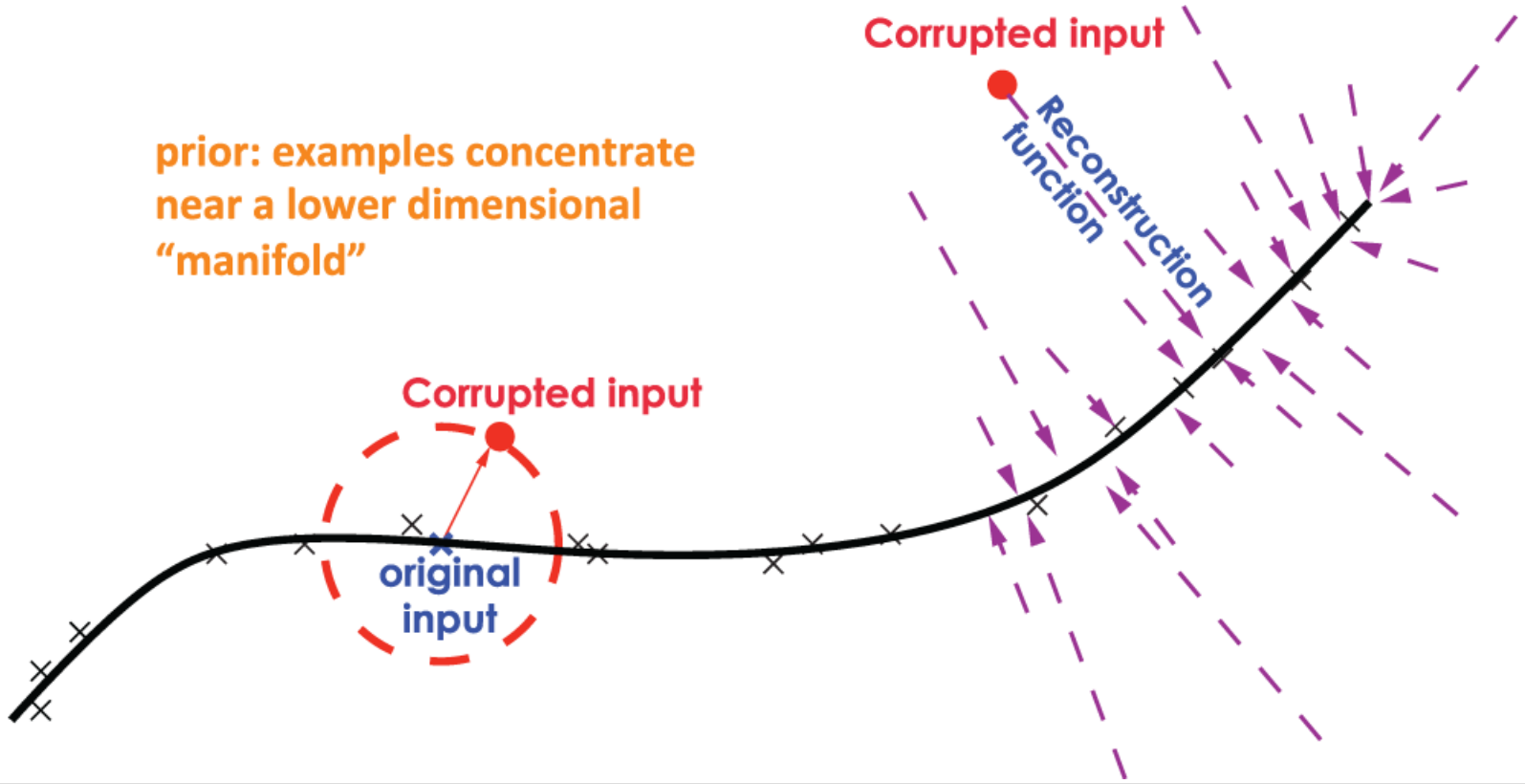
Denoising AE (DAE)

- **Training:**
  - Reconstruction $\hat{x}$ computed from the corrupted input $\tilde{x}$.
  - Loss function compares $\hat{x}$ reconstruction with the noiseless x.
- What does it learn?
  - The autoencoder cannot fully trust each feature of x independently, so it must **learn the correlations of x's features**.
  - Based on those correlations we can learn a more 'not prone to changes' model.
  - **We are forcing the hidden layer to learn a generalized structure of the data.**

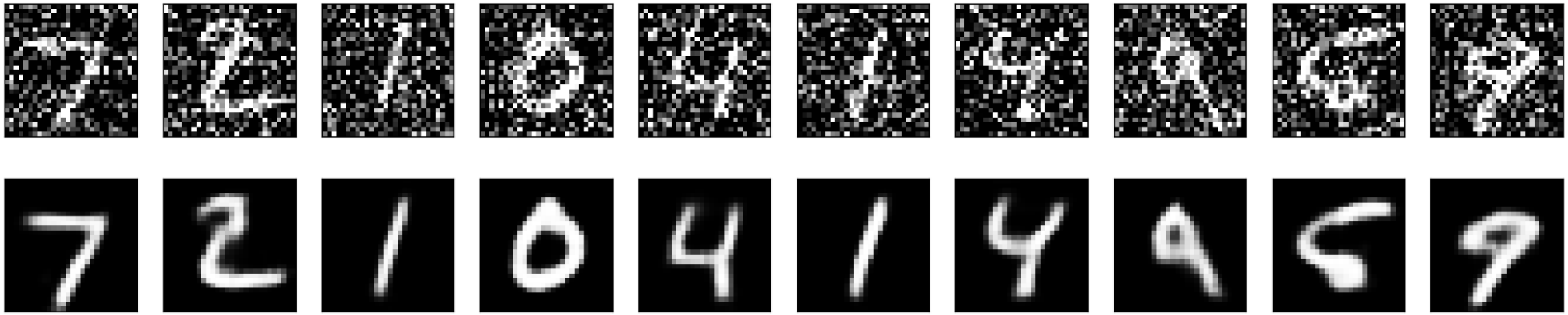


Denoising AE (DAE)

- The DAE is trained to map a corrupted data point back to the manifold.
- It does so by mapping low probability points to higher probability points.



Example results on MNIST

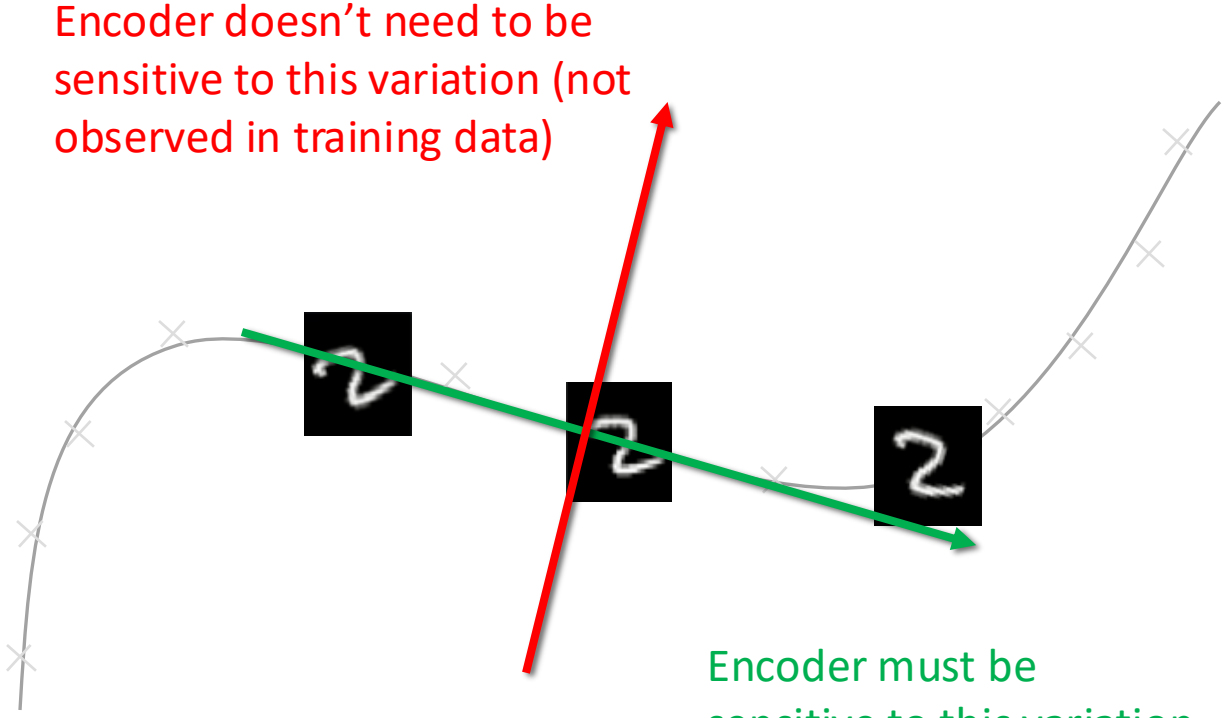


Contractive autoencoder (CAE)

- **Idea:** We wish to extract features that only reflect variations observed in the training set. We would like to be invariant to the other variations.
- Points close to each other in the input space maintain that property in the latent space.
- **Solution:** add a regularization term $\Omega(x)$:

$$J^*(W, x) = J(\mathrm{W}) + \lambda\Omega(x)$$

$$\Omega(x) = \sum_{i,j}\left(\frac{\partial f(x)_j}{\partial x_i}\right)^2$$



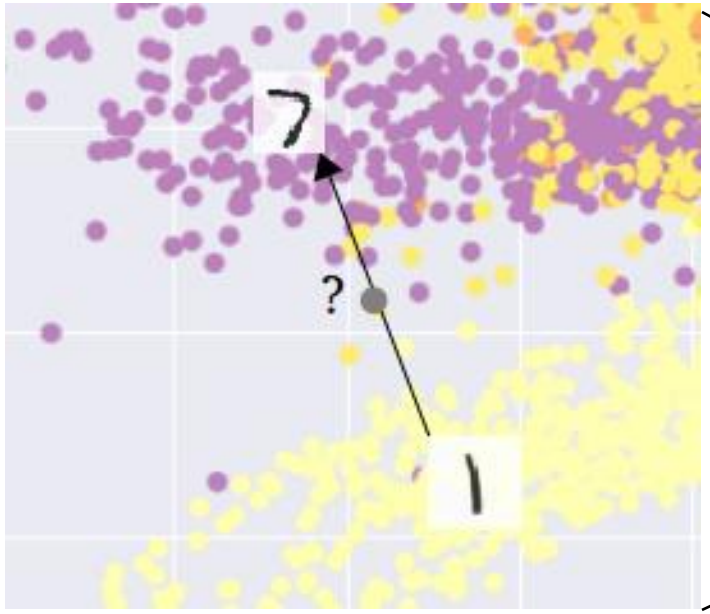
For more details: See chapter 14.7 of Ian Goodfellow's Deep Learning book

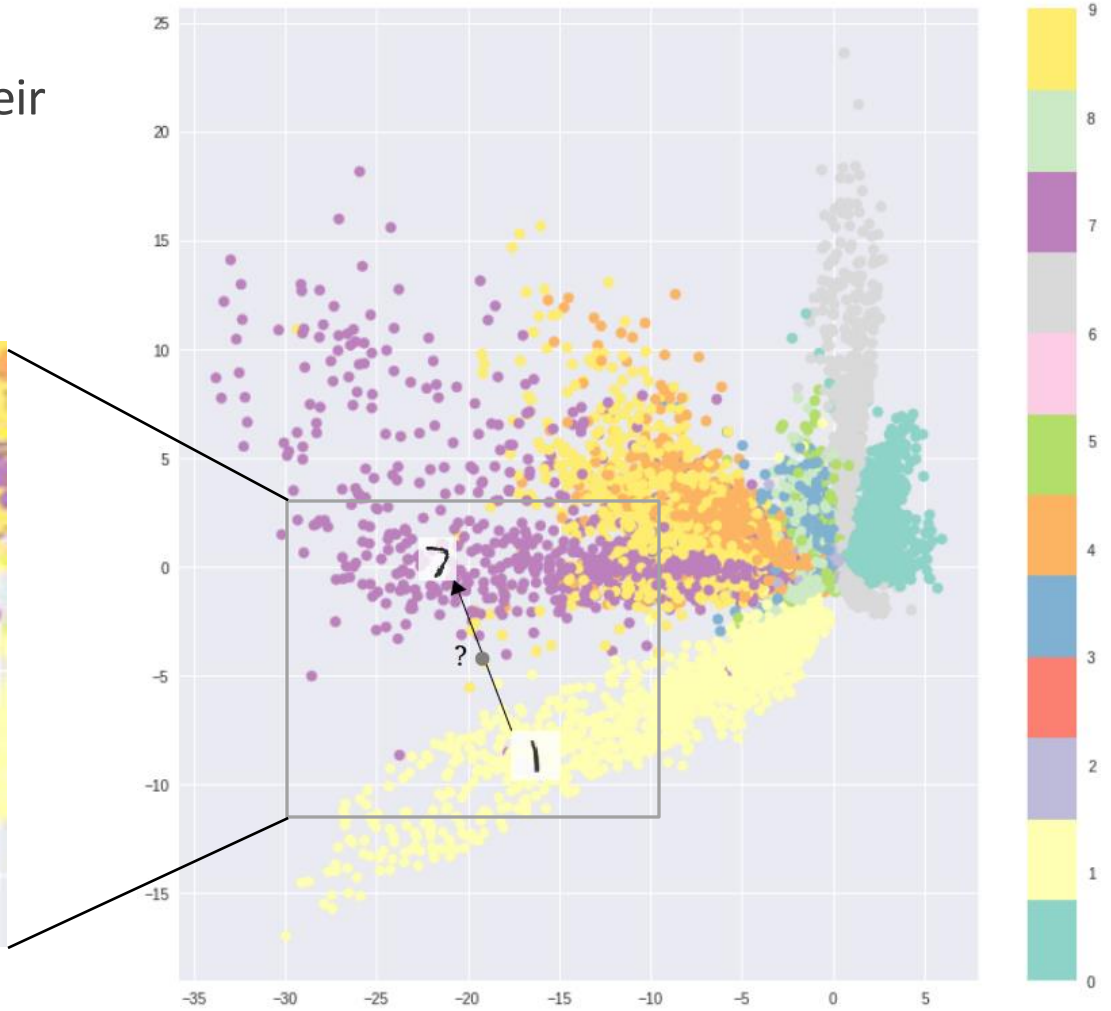
Variational autoencoders

Problem with simple autoencoders

- The fundamental problem with autoencoders, for generation, is that the latent space they convert their inputs to and where their encoded vectors lie, **may not be continuous**, or allow easy interpolation.

If the space has discontinuities (e.g., gaps between clusters) and you sample/generate a variation from there, the decoder will simply generate an **unrealistic output**, because the decoder has *no idea* how to deal with that region of the latent space. During training, it *never saw* encoded vectors coming from that region of latent space.

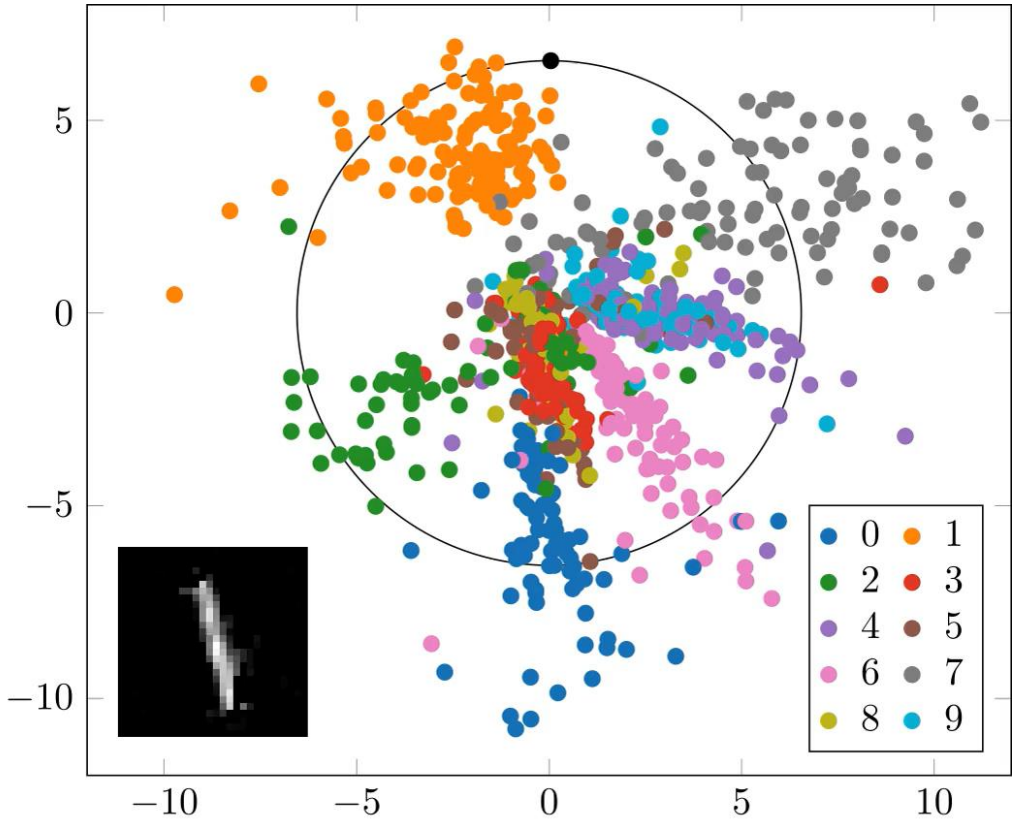




https://towardsdatascience.com/intuitively-understanding-variational-autoencoders-1bfe67eb5daf

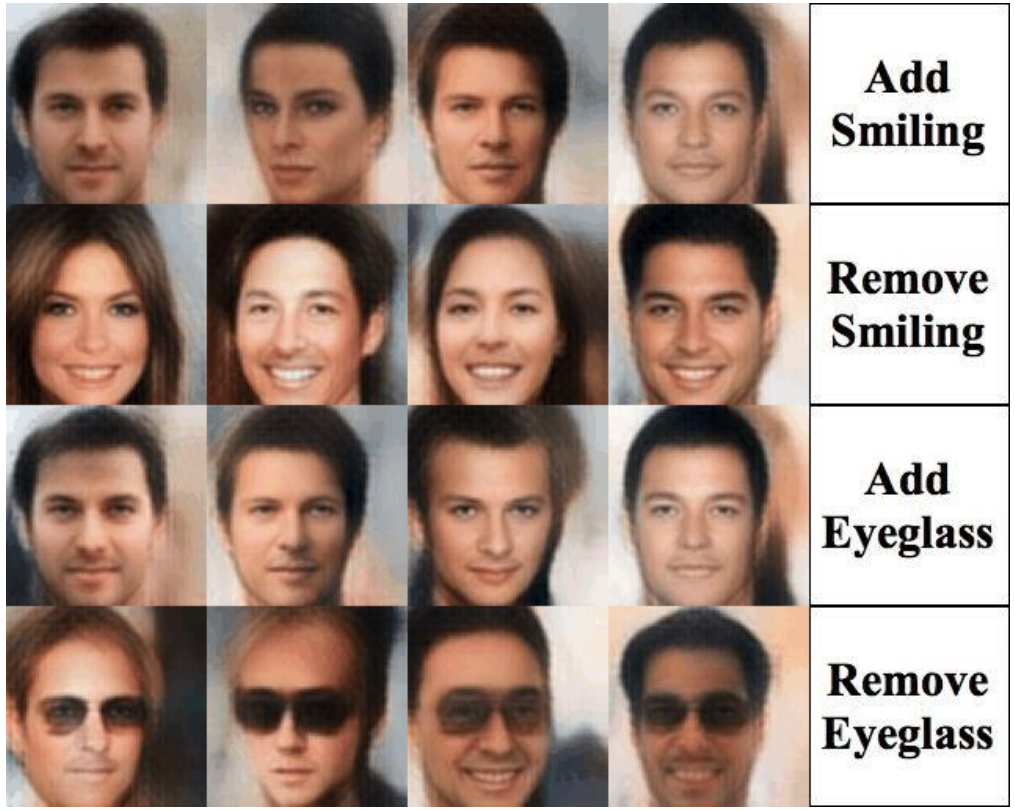
Latent space interpolation

https://gertjanvandenburg.com/blog/autoencoder/



Regular Autoencoder on MNIST dataset

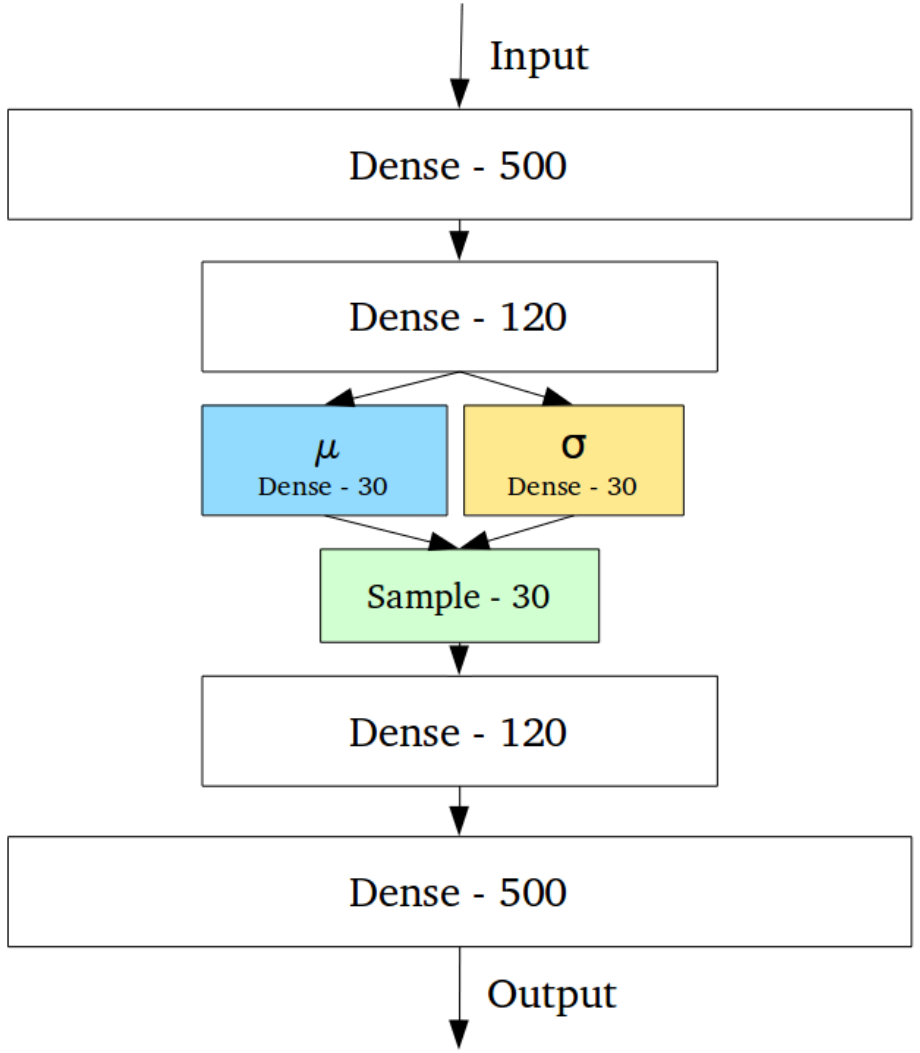
https://github.com/houxianxu/DFC-VAE



Variational Autoencoder on CelebA dataset

Variational autoencoder (VAE)

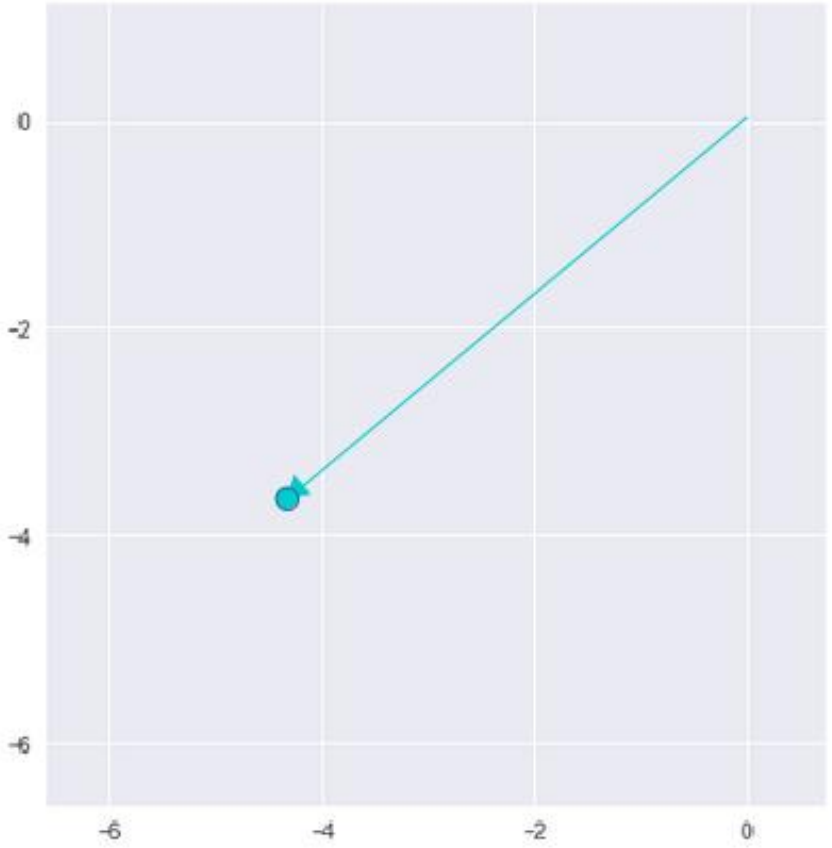
- Very similar to the regular autoencoder
- Fundamentally unique property: their **latent spaces are, *by design,* continuous**, allowing easy random sampling and interpolation.
- How?
- By making its encoder not output an encoding vector of size *n,* but rather **outputting two vectors** of size n:
- a vector of means**, μ**, and another vector of standard deviations, **σ**.
- Probabilistic nature using a **sampling layer**.



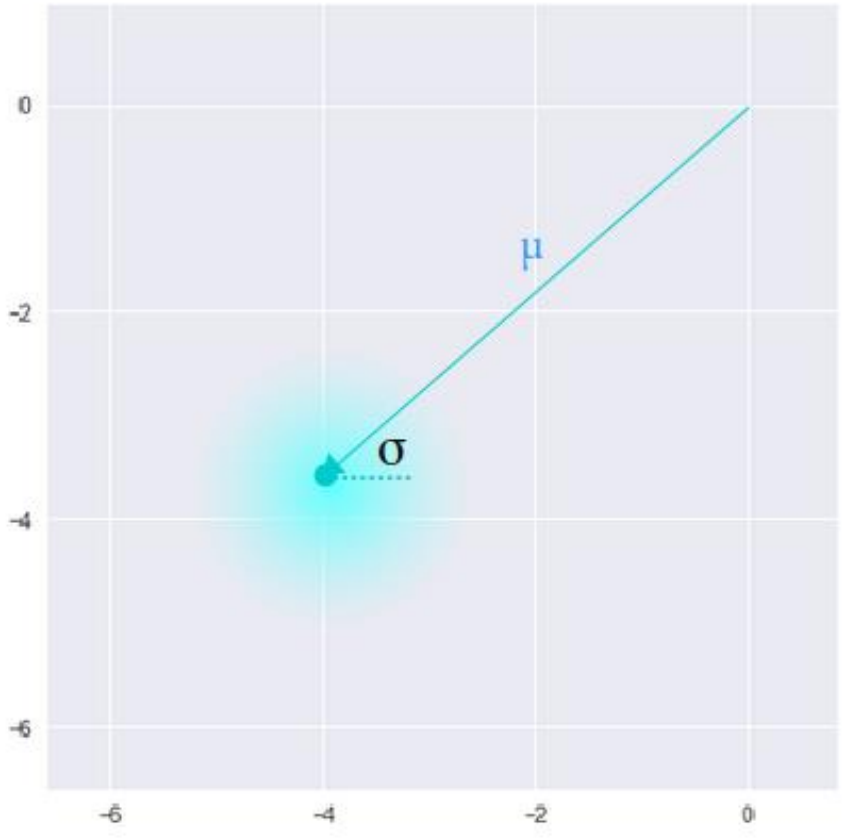
https://arxiv.org/abs/1312.6114

Random sampling layer

The stochastic generation means, that even for the same input, while the mean and standard deviations remain the same, the actual encoding will somewhat vary on every single pass simply due to sampling.



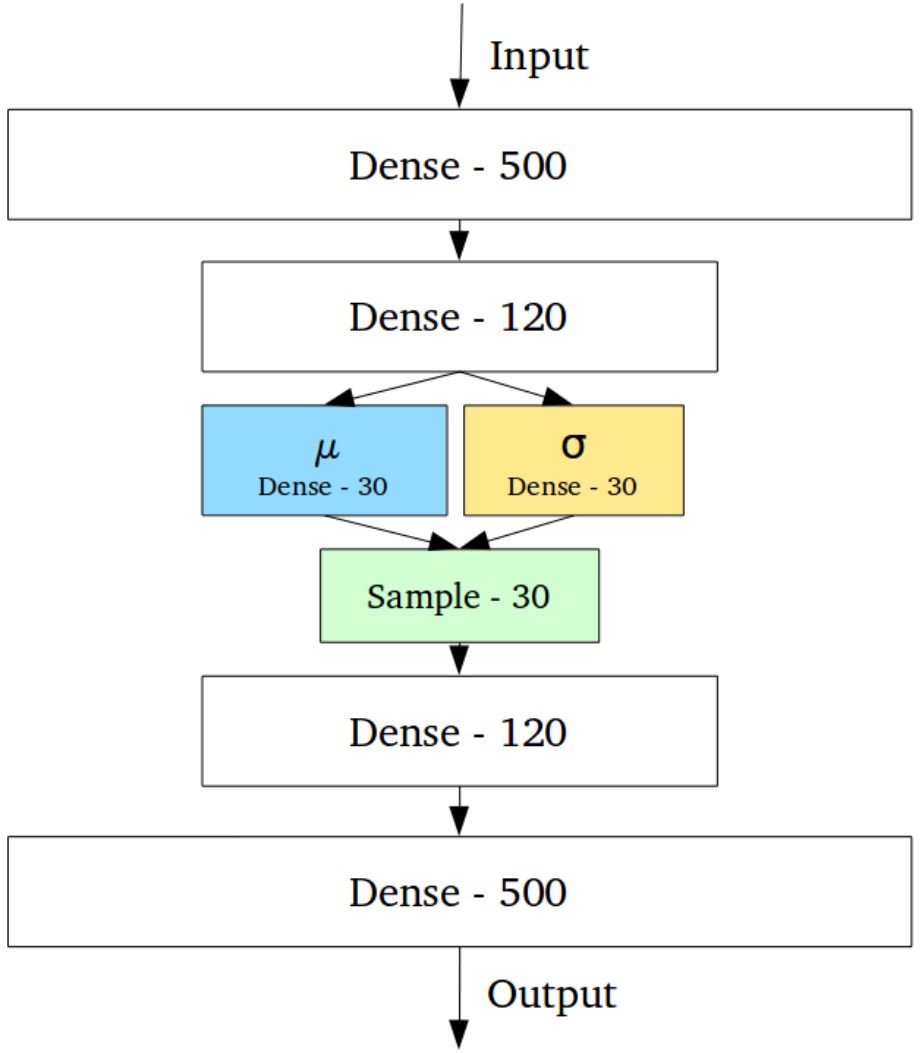
Standard Autoencoder
(direct encoding coordinates)



Variational Autoencoder
(μ and σ initialize a probability distribution)

Random sampling layer

- The hidden layer outputs the parameters of a vector of random variables of length n.
- The *i*'th element of **μ** and **σ** represents the mean and standard deviation of the *i*'th random variable, $\mathbf{X}_i$, from which we sample, to obtain the sampled encoding, which we pass onward to the decoder.



Random sampling layer

- Example:

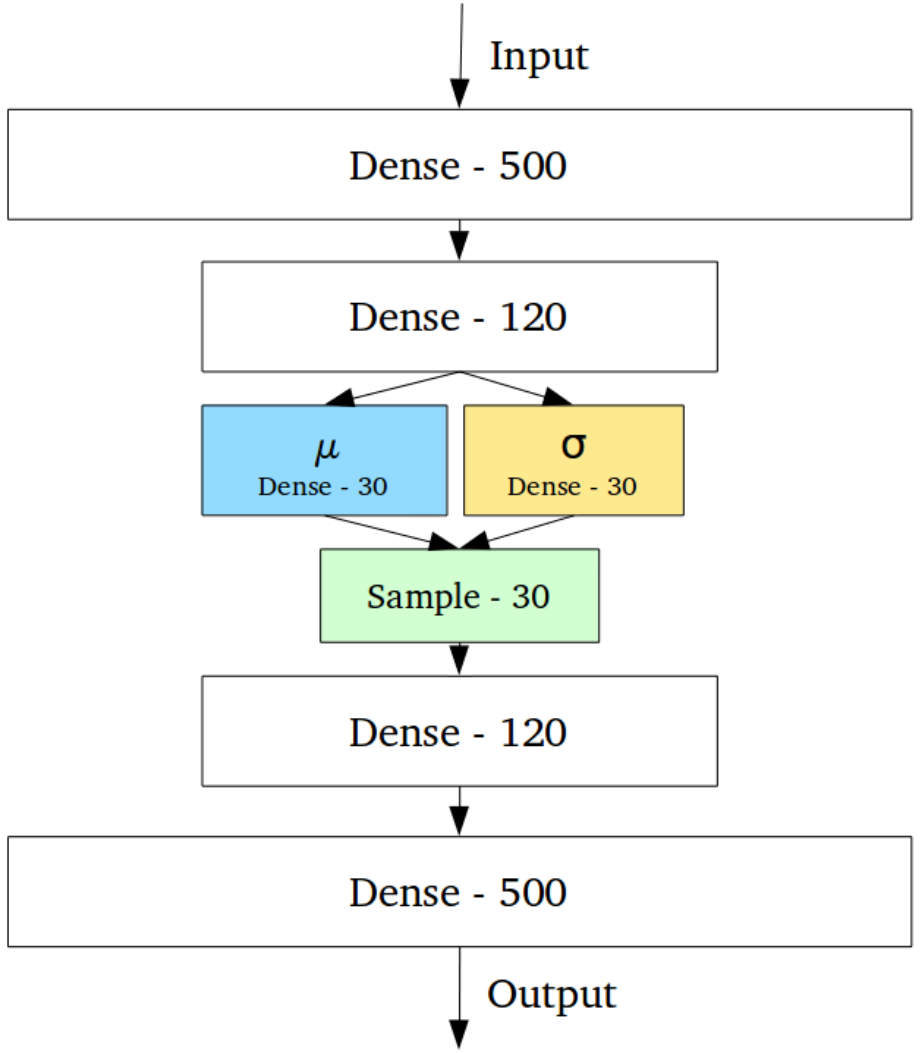
Output μ [0.1, 1.2, 0.2, 0.8,…]

Output σ [0.2, 0.5, 0.8, 1.3,...]

Intermediate X $[X_{1\sim N(0.1,\ 0.2^2)},\ X_{2\sim N(1.2,\ 0.5^2)},\ X_{3\sim N(0.2,\ 0.8^2)},\ X_{4\sim N(0.8,\ 1.3^2)},....]$

sample

Sampled vector [0.28, 1.65, 0.92, 1.98,...]



Random sampling layer

- Example:

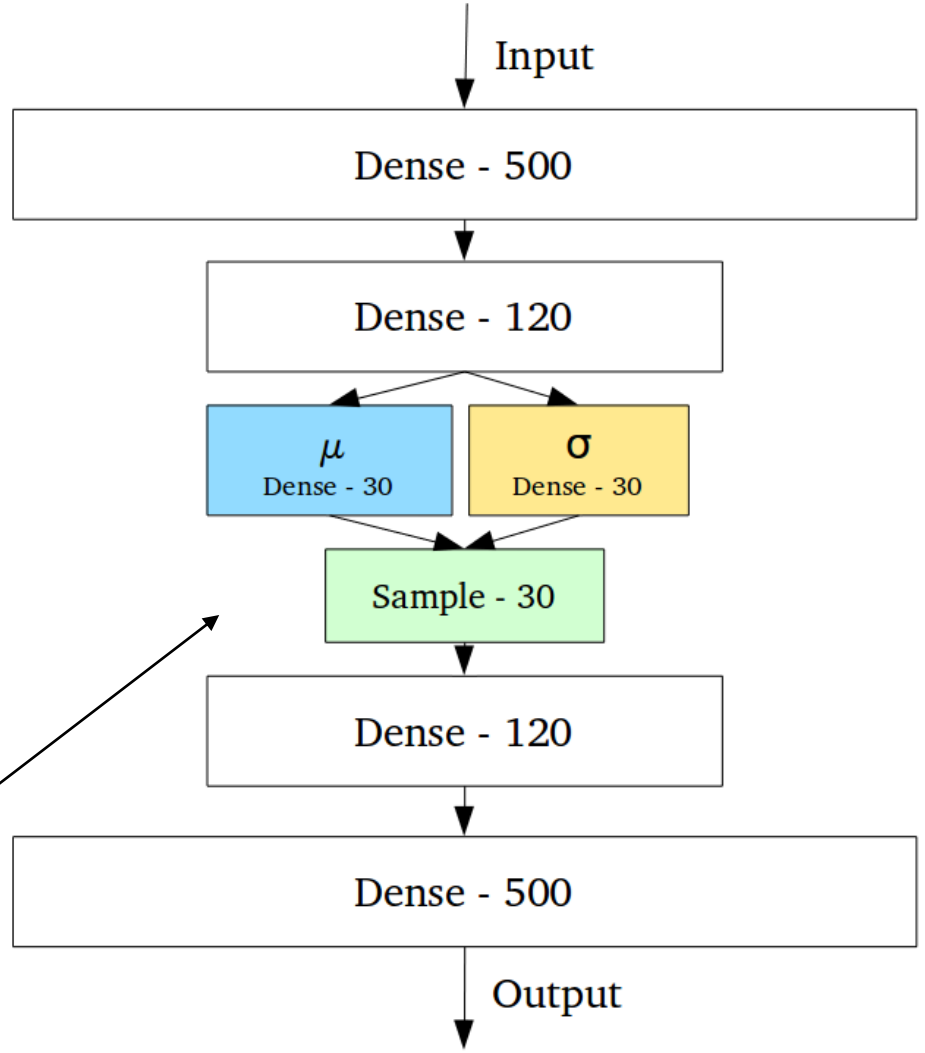
Output μ [0.1, 1.2, 0.2, 0.8,…]

Output σ [0.2, 0.5, 0.8, 1.3,...]

Intermediate **X** $[X_{1\sim N(0.1,\ 0.2^2)},\ X_{2\sim N(1.2,\ 0.5^2)},\ X_{3\sim N(0.2,\ 0.8^2)},\ X_{4\sim N(0.8,\ 1.3^2)},....]$

sample

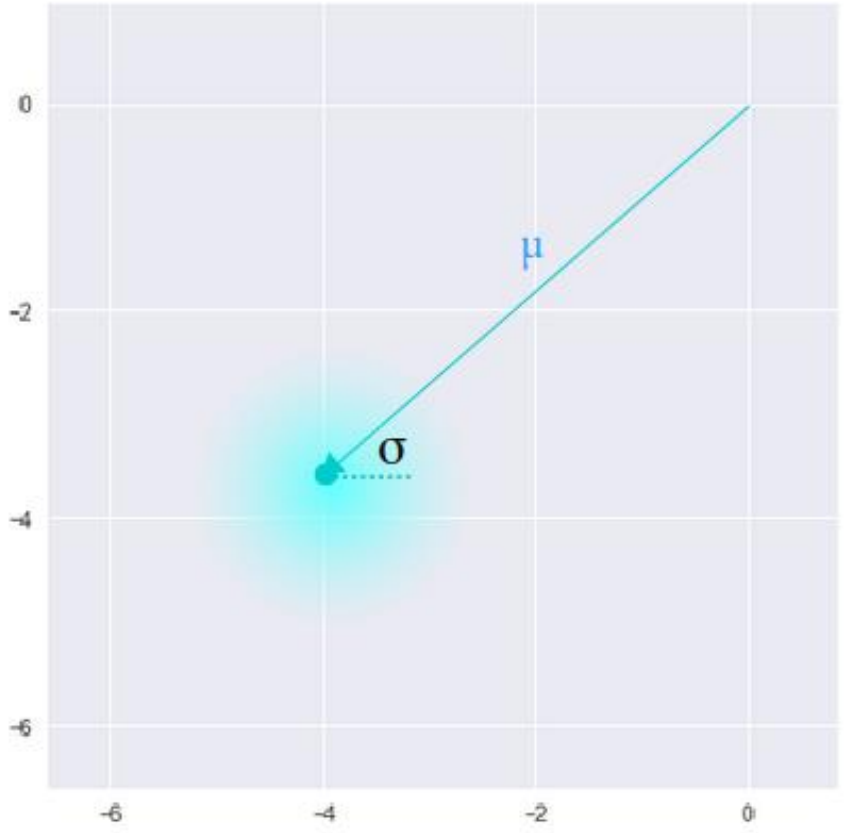
Sampled vector [0.28, 1.65, 0.92, 1.98,...]



Note: Here X denotes the latent representation (normally we would use *h* or *z*).

Probabilistic nature of VAC

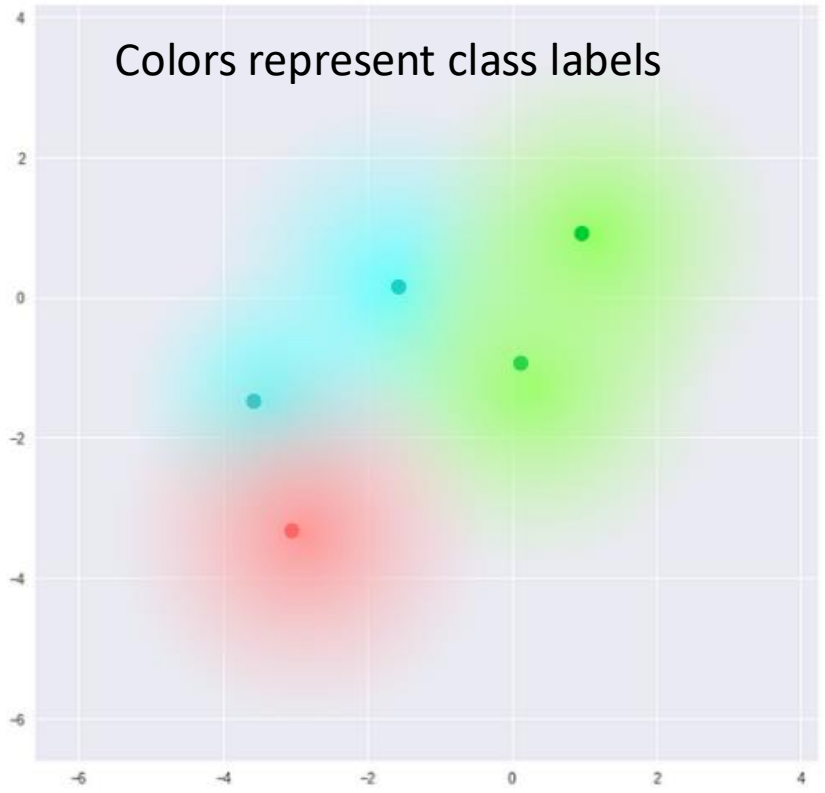
- As encodings are generated at random from anywhere inside the “circle” (the distribution), the decoder learns that **not only is a single point in latent space referring to a sample of that class, but all nearby points refer to the same as well**.
- This allows the decoder to not just decode single, specific encodings in the latent space (leaving the decodable latent space discontinuous), but ones that slightly vary too, as **the decoder is exposed to a range of variations** of the encoding of the same input during training.
- This **results in a smooth latent space** on a local scale, that is, for similar samples.



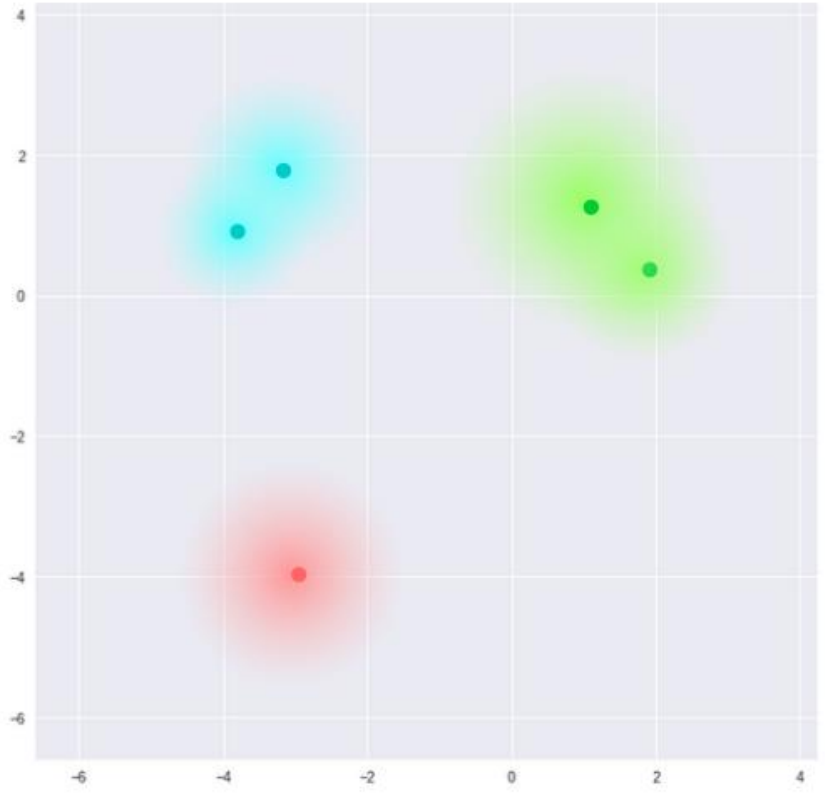
Variational Autoencoder
(μ and σ initialize a probability distribution)

Local smoothness is not enough

We want encodings, *all* of which are as close as possible to each other while still being distinct, allowing smooth interpolation, and enabling the construction of *new* samples.



What we require



What we may inadvertently end up with

If the encoder learns to cluster samples apart, the decoder will be able to reconstruct the *training* data better.

Solution

- We introduce the KL divergence into the loss function.
- Minimizing the KL divergence here means optimizing the probability distribution parameters **(μ** and **σ)** to closely resemble that of some target distribution (standard normal).
- For VAEs, the KL loss is equivalent to the *sum* of all the KL divergences between the latent *component* $Z_i \sim N(\mu_i, \sigma_i^2)$ in **Z**, and the standard normal, where $\mu_i = 0, \sigma_i^2 = 1$.
- The KL loss term has this expression (which is minimal exactly when $\mu_i = 0$ and $\sigma_i = 1$):

$$\sum_{i=1}^{n} \sigma_i^2 + \mu_i^2 - \log(\sigma_i) - 1$$

The generator *g* maps a random vector *z* to *x*.

- We still minimize similarity as well:

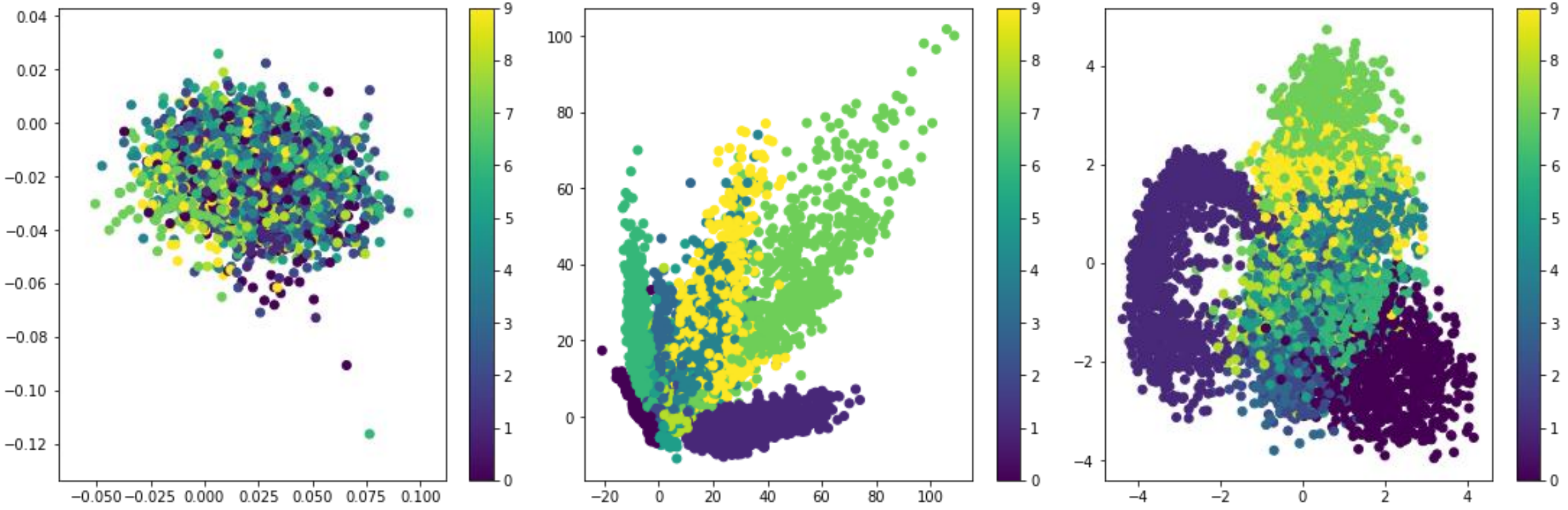
$$\frac{1}{2}\sum_{i=1}^{n} \left\| \widehat{x^{(i)}} - x^{(i)} \right\|^2 = \frac{1}{2}\sum_{i=1}^{n} \left\| g(z^{(i)}) - x^{(i)} \right\|^2$$

Effect of KL loss term

KL loss term only

Similarity loss term only

Similarity + KL

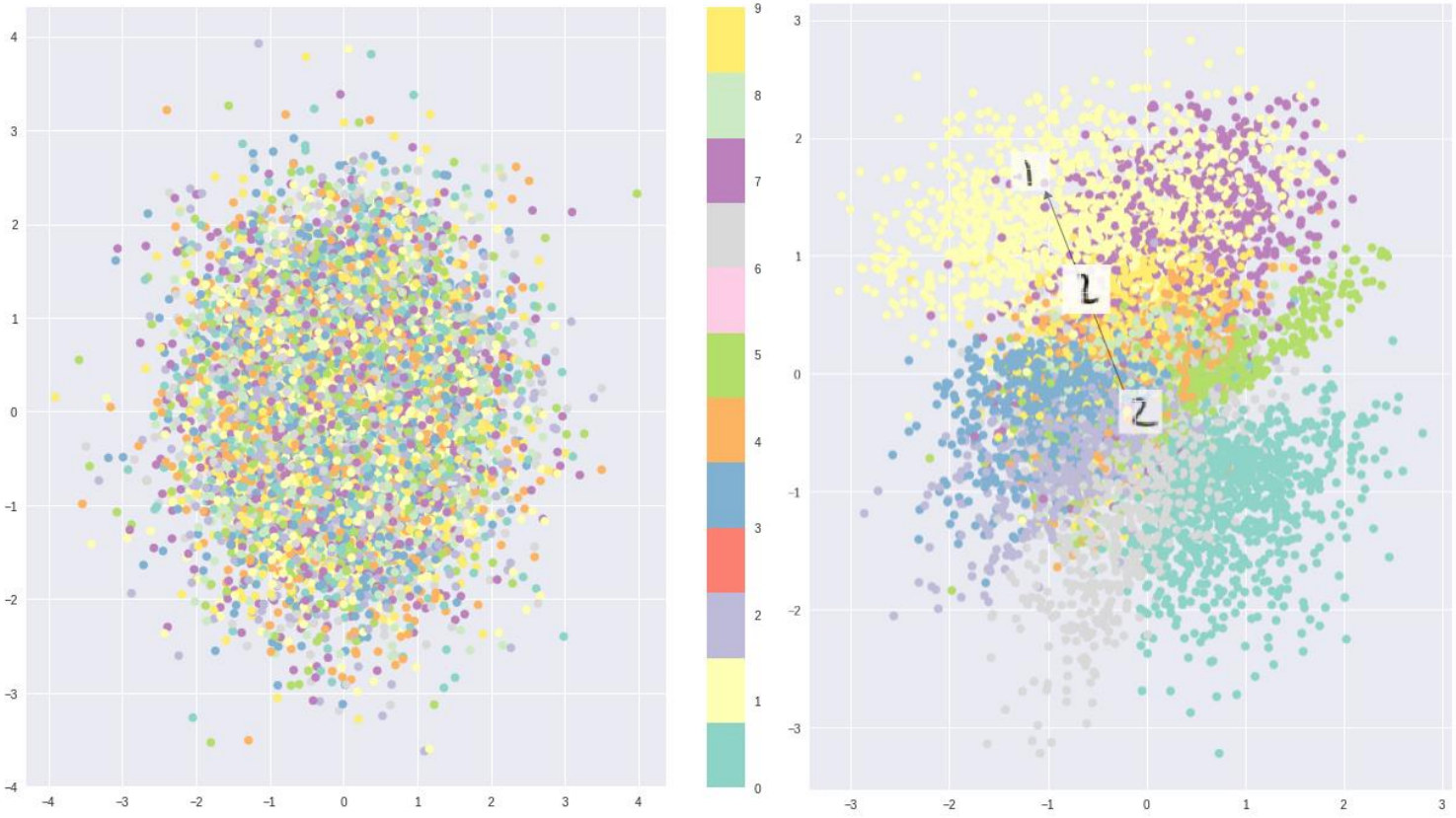


Latent space visualizations

Effect of KL loss term

- Intuitively, this loss encourages the encoder to **distribute all encodings** (for all types of inputs, e.g., all MNIST numbers), **evenly around the center of the latent space**. If it tries to "cheat" by clustering them apart into specific regions, away from the origin, it will be penalized.

Optimizing **purely KL loss** results in latent space encodings densely placed randomly, near the center of the latent space, with little regard for similarity among nearby encodings. **This is impossible to decode.**



Optimizing similarity and KL loss, results in the generation of a latent space which maintains the similarity of nearby encodings on the *local scale* via clustering,
yet *globally,* is very densely packed near the latent space origin.

Statistical motivation (not mandatory)

- Suppose that there exists some hidden variable z which generates an observation x.
- We can only observe x, but we would like to infer the characteristics of z. In other words, we'd like to compute $p(z|x)$.
- According to Bayes' theorem $p(z|x) = p(x|z)p(z)/p(x)$.
- $p(z)$: use simple Gaussian prior, $z \sim N(0,1)$
- $p(x|z)$: **decoder network**
- $p(x) = \int p(x|z)p(z)\,dz$: Unfortunately, computing this for every z is intractable.
- Instead, we approximate $p(z|x)$ by another distribution $q(z|x)$, i.e. **encoder network**, which we'll define such that it has a tractable distribution.
- How? Minimize KL divergence: $\min KL\big(q(z|x)||p(z|x)\big)$
- **Solution:** Maximize $E_{z\sim q(z|x)} \log\big(p(x|z)\big) - KL\big(q(z|x)||p(z)\big)$

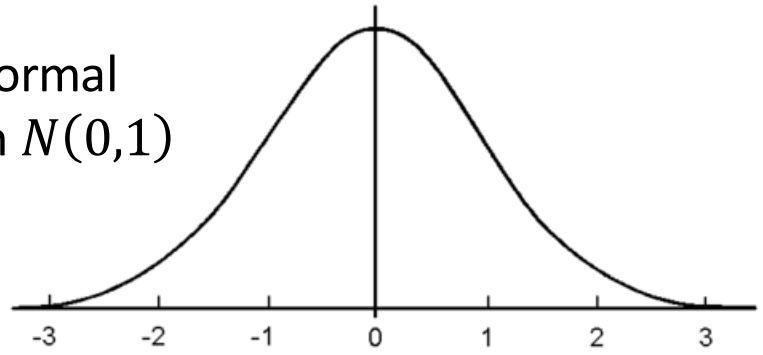
Full derivation here: https://www.youtube.com/watch?v=uaaqyVS9-rM&feature=youtu.be&t=19m42s

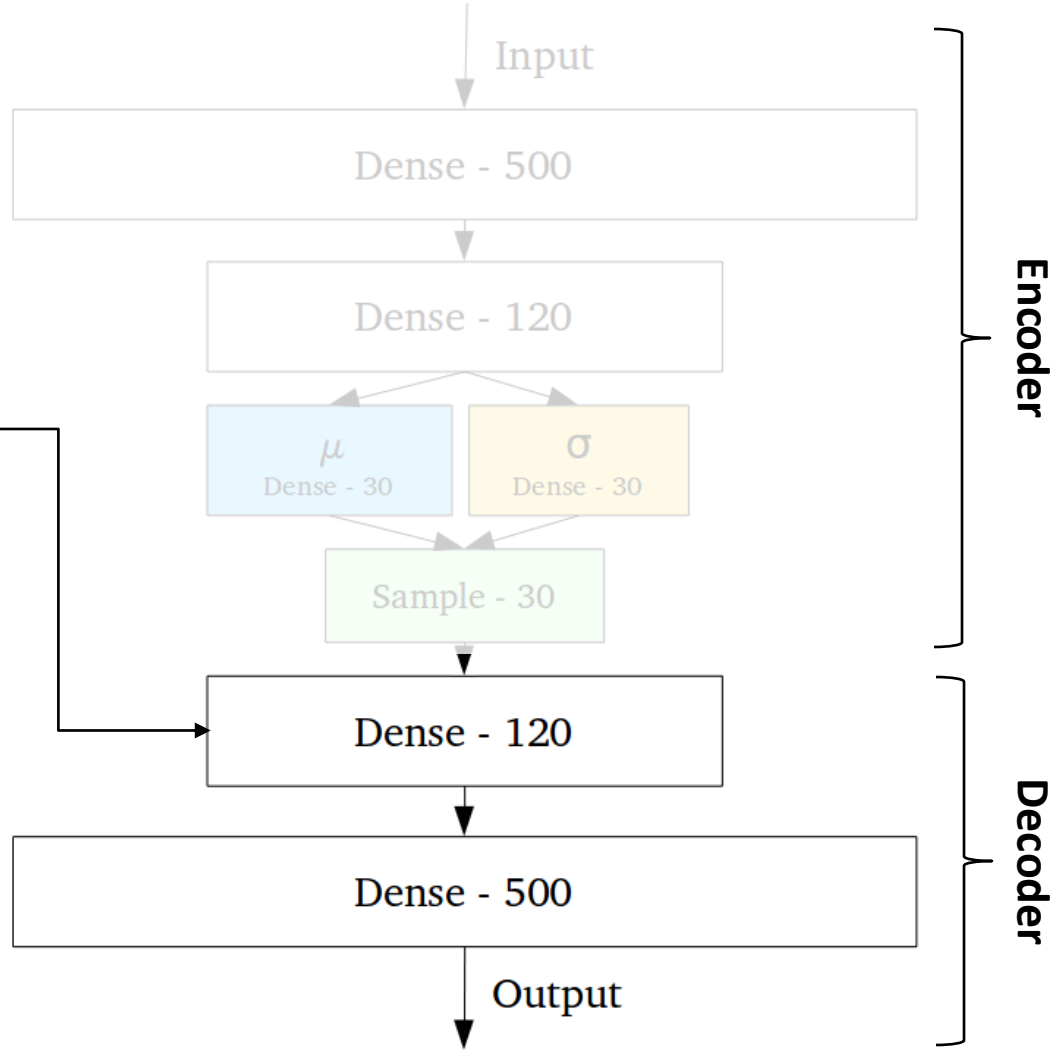
Generating data

- By training the network to output latent vectors that are standard normally distributed, generating new images becomes easy!

1. Sample latent representation from standard normal: $z_i \sim N(0,1), i = 1,2,\dots,120$
2. Feed $[z_1, z_2, \dots, z_{120}]$ directly into decoder (which has been trained to work on standard normally distributed inputs!)

Standard normal distribution $N(0,1)$





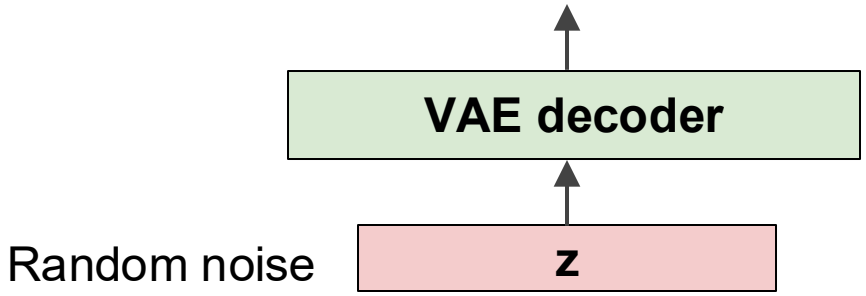
What we just did is actually very clever!

Solution: Sample from a simple distribution, e.g. Gaussian noise. Learn transformation to training distribution (= VAE decoder)

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

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

Problem: Want to sample from complex, high-dimensional training distribution. No direct way to do this!

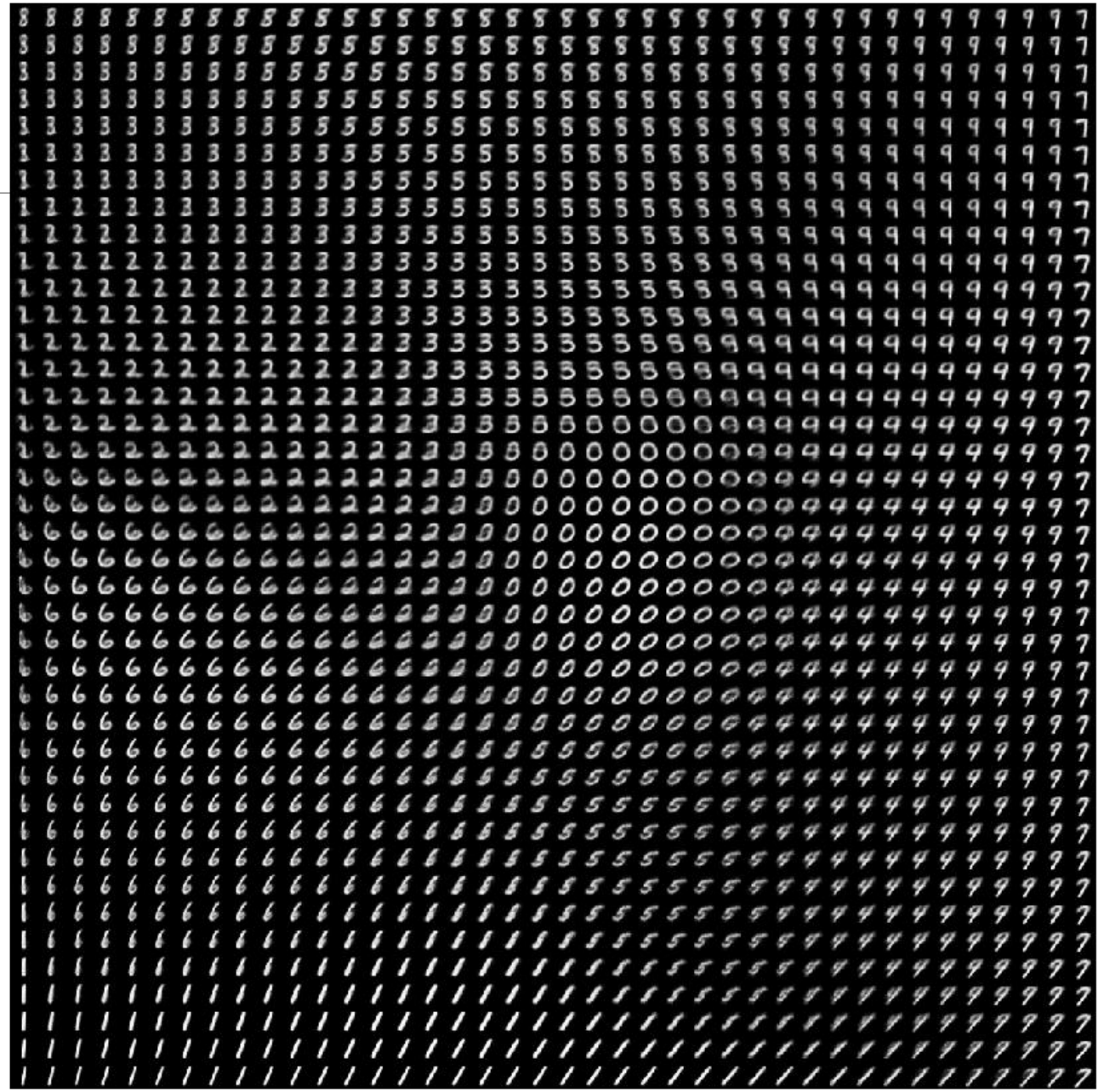


Side-note: Note that we don't explicitly model the distribution $p_{model}(x)$. We just learn to draw samples.

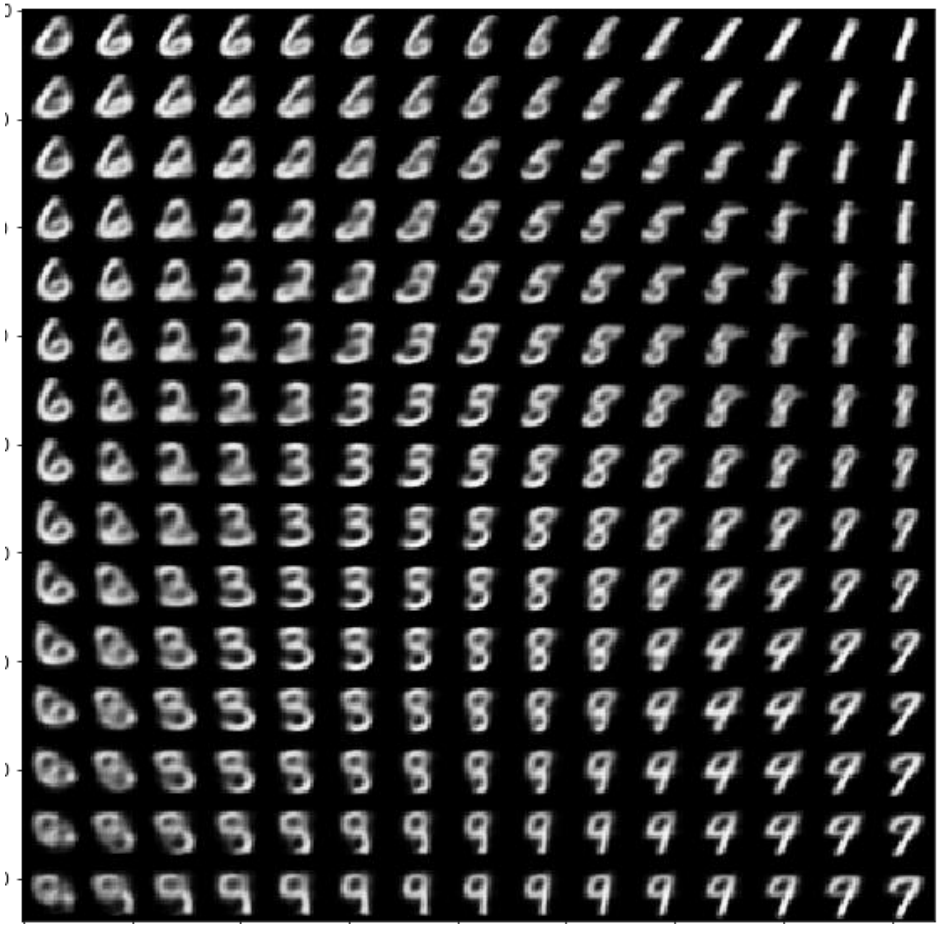
VAE example

- The figure shows images generated using a VAE with a 2-dimensional latent space*.
- Each grid cell corresponds to a latent vector $[z_1, z_2]$ where $-2 \leq z_i \leq 2$.
- As you can see, the distinct digits each exist in different regions of the latent space and smoothly transform from one digit to another. This smooth transformation can be quite useful when you'd like to interpolate between two observations.

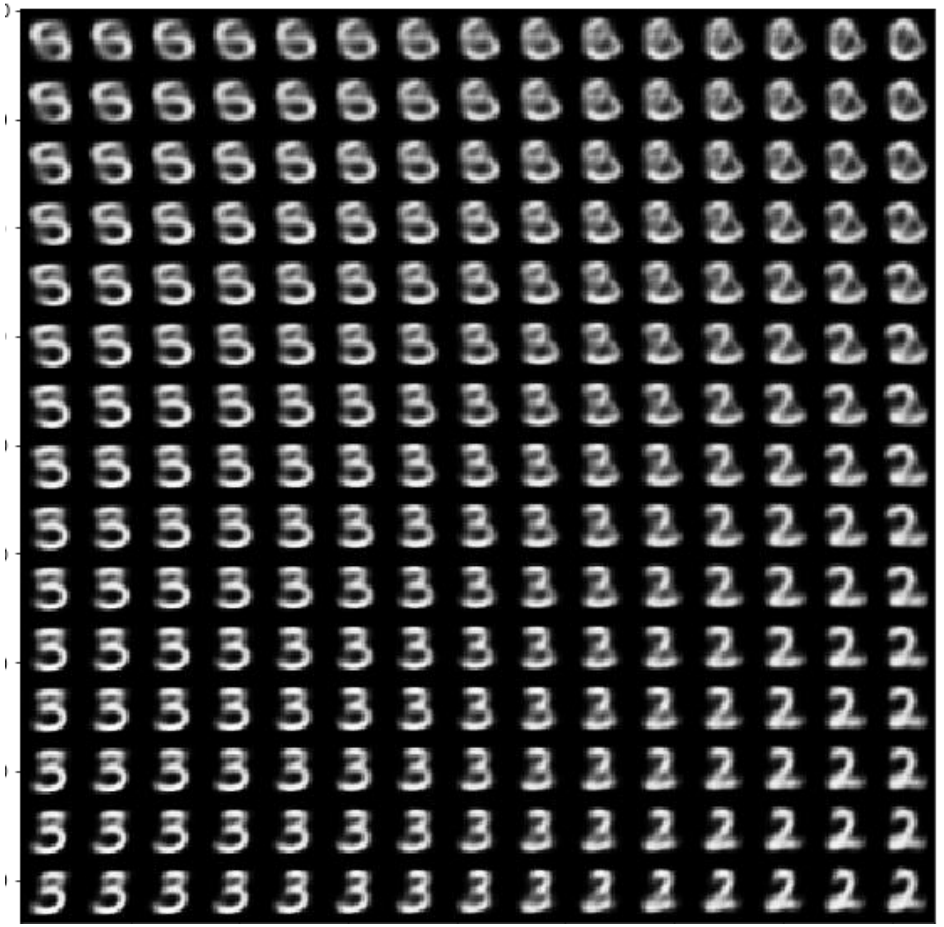
*2-d latent space means low model capacity, so we should expect low image quality. Increasing latent space dimensionality fixes this.



VAE vs traditional AE



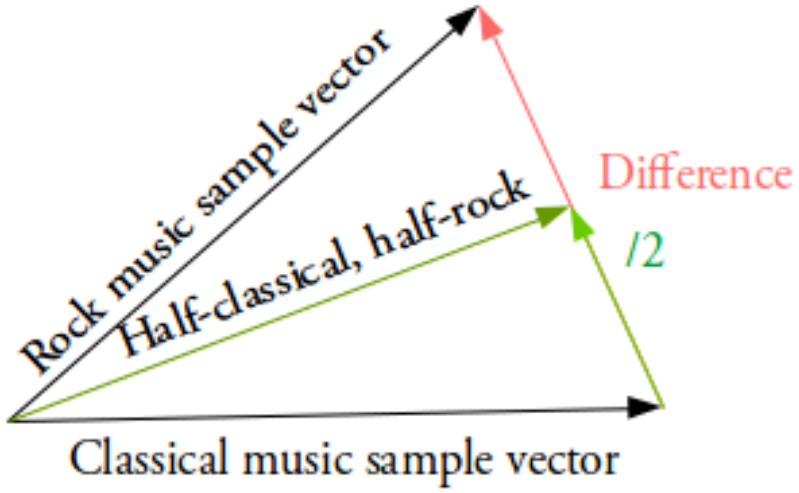
Generating Images from VAE

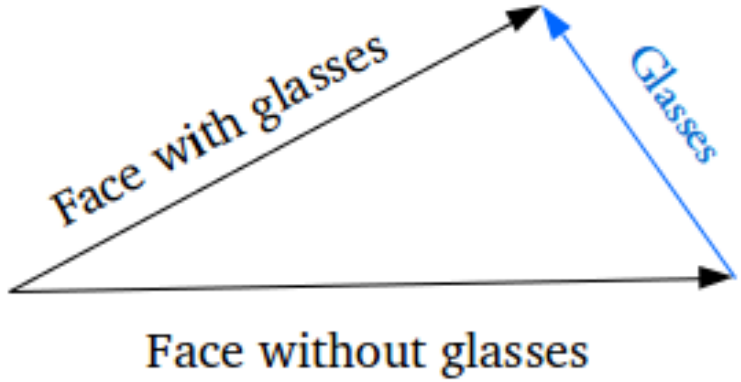


"Generating" Images from Traditional AE

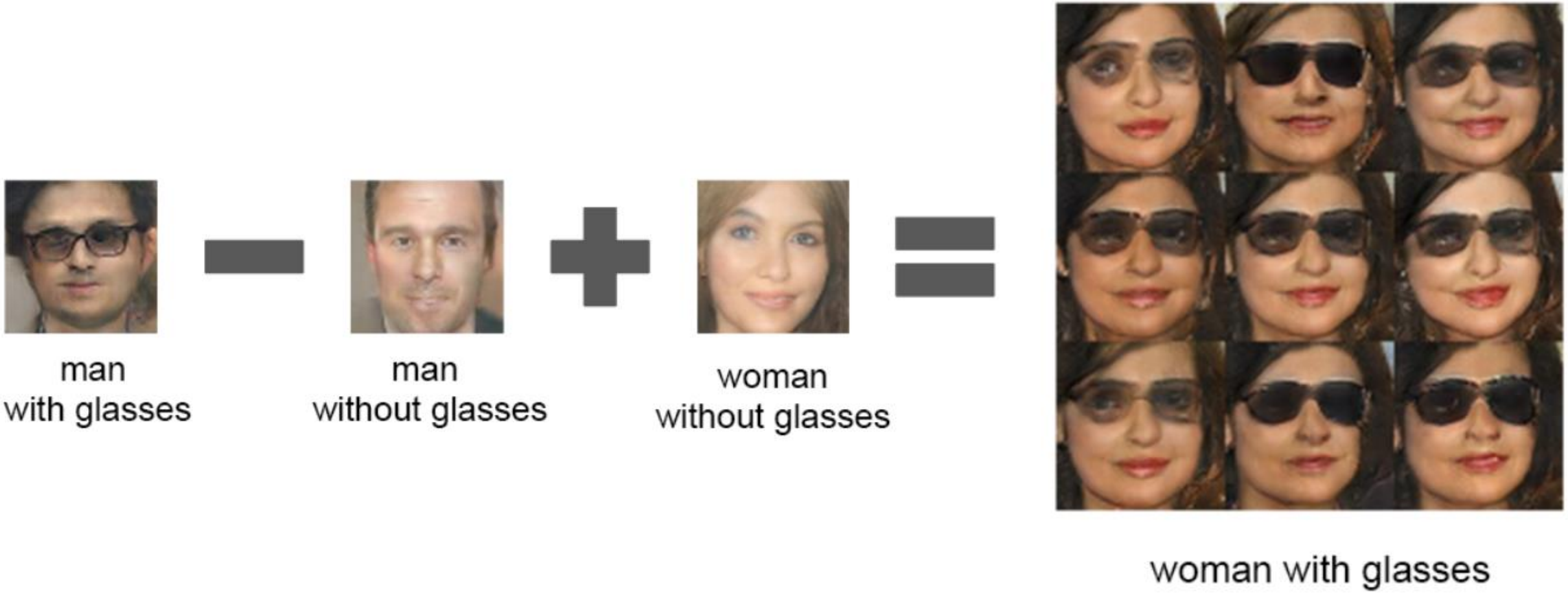
Note: 2-d latent space means low model capacity, so we should expect low image quality. Increasing latent space dimensionality fixes this.

Latent space arithmetic

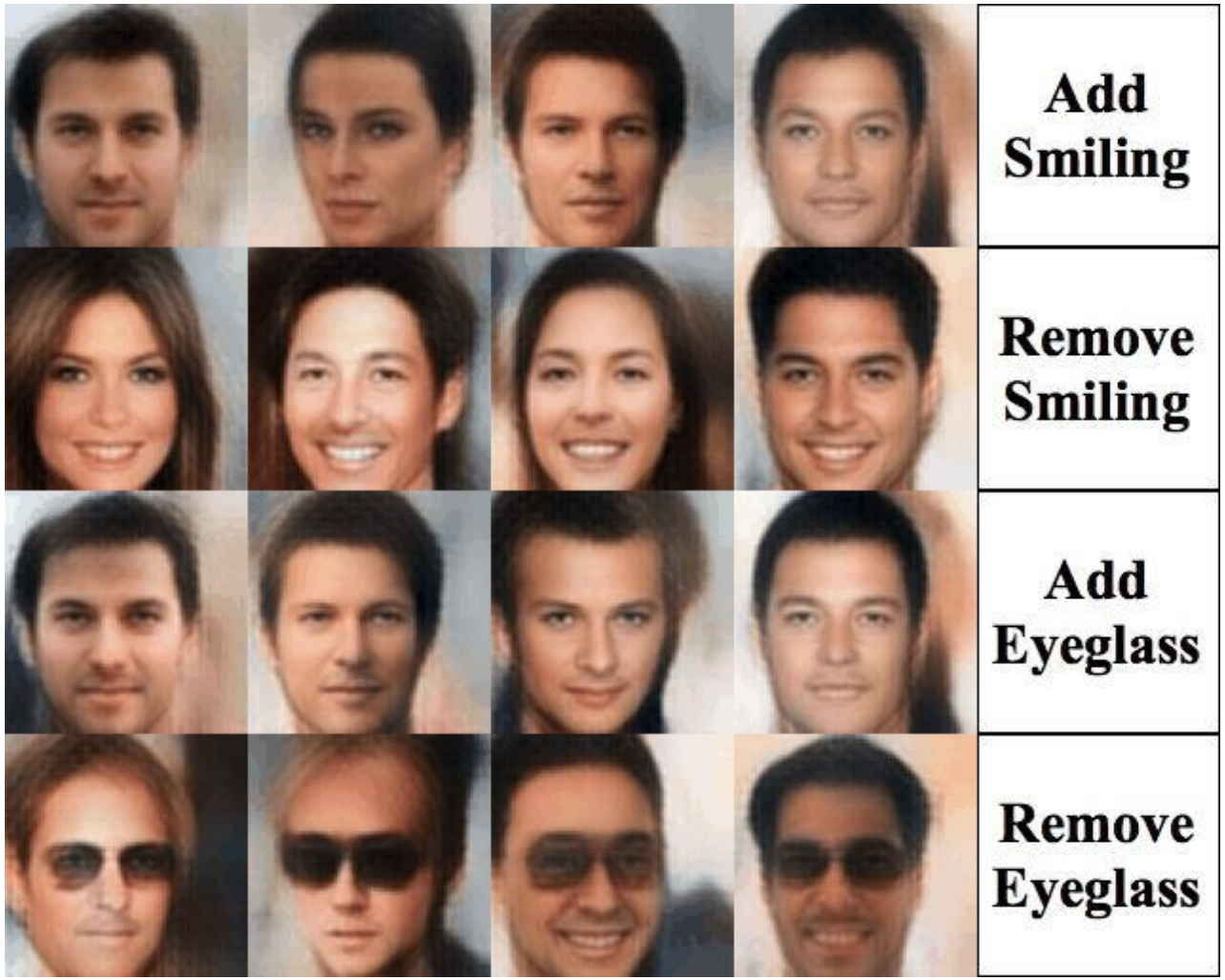




Latent space arithmetic



Latent space arithmetic



https://github.com/houxianxu/DFC-VAE

Latent space arithmetic

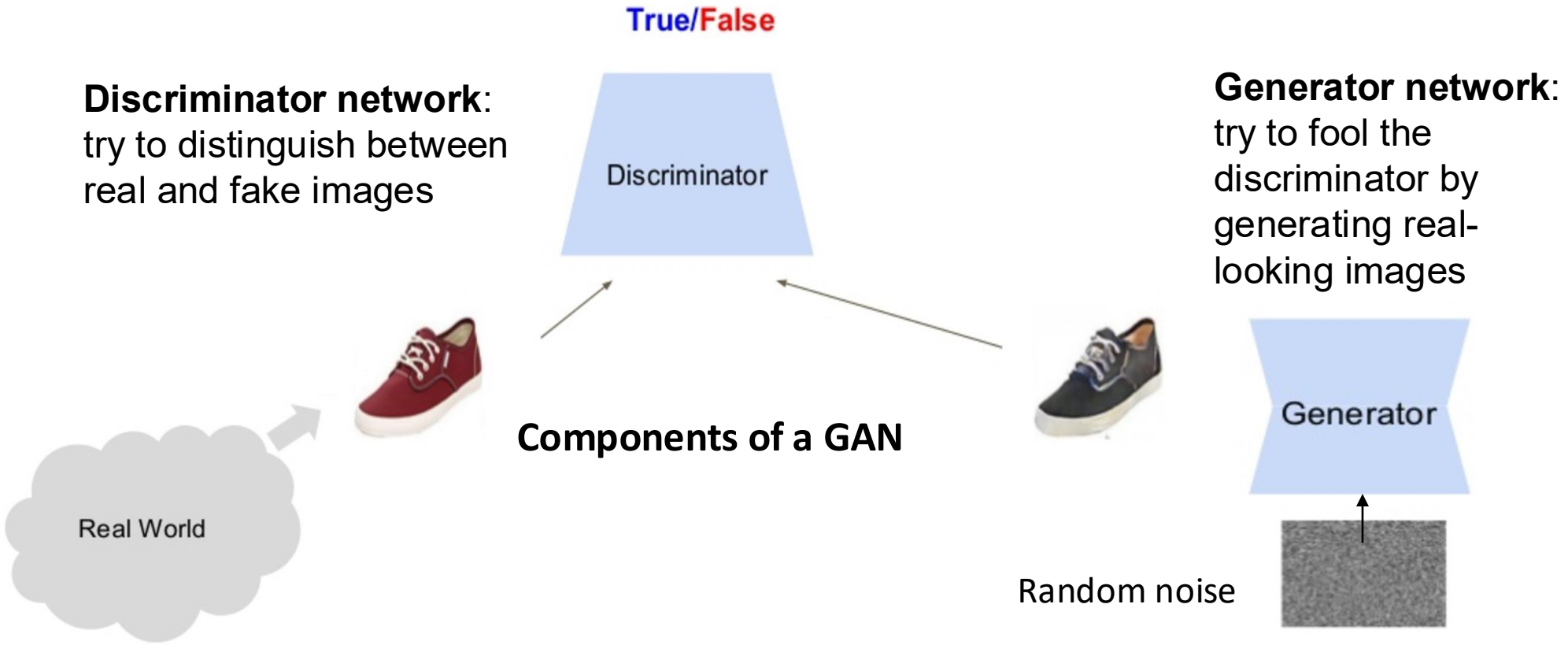
Generative Adversarial Networks (GANs)

Motivation: Problem with VAEs

- VAEs produce coherent and structured images due to the smoothness of the latent space. This makes them ideal for downstream tasks (e.g., unsupervised pretraining followed by supervised fine-tuning).
- **Observation:**
  - The reconstruction loss (e.g., mean squared error) encourages the output to be close to the input on average.
  - As a result, images generated by VAEs **tend to be blurry**, because averaging pixel-level errors smooths out fine details (especially textures and edges).
- **Idea behind GANs:**
  - Add a second network (discriminator) that is trained to judge if a generated image looks real or not.
  - This adds an extra training pressure on making images that “look real”, which results in very sharp, photorealistic images.

Generative Adversarial Networks

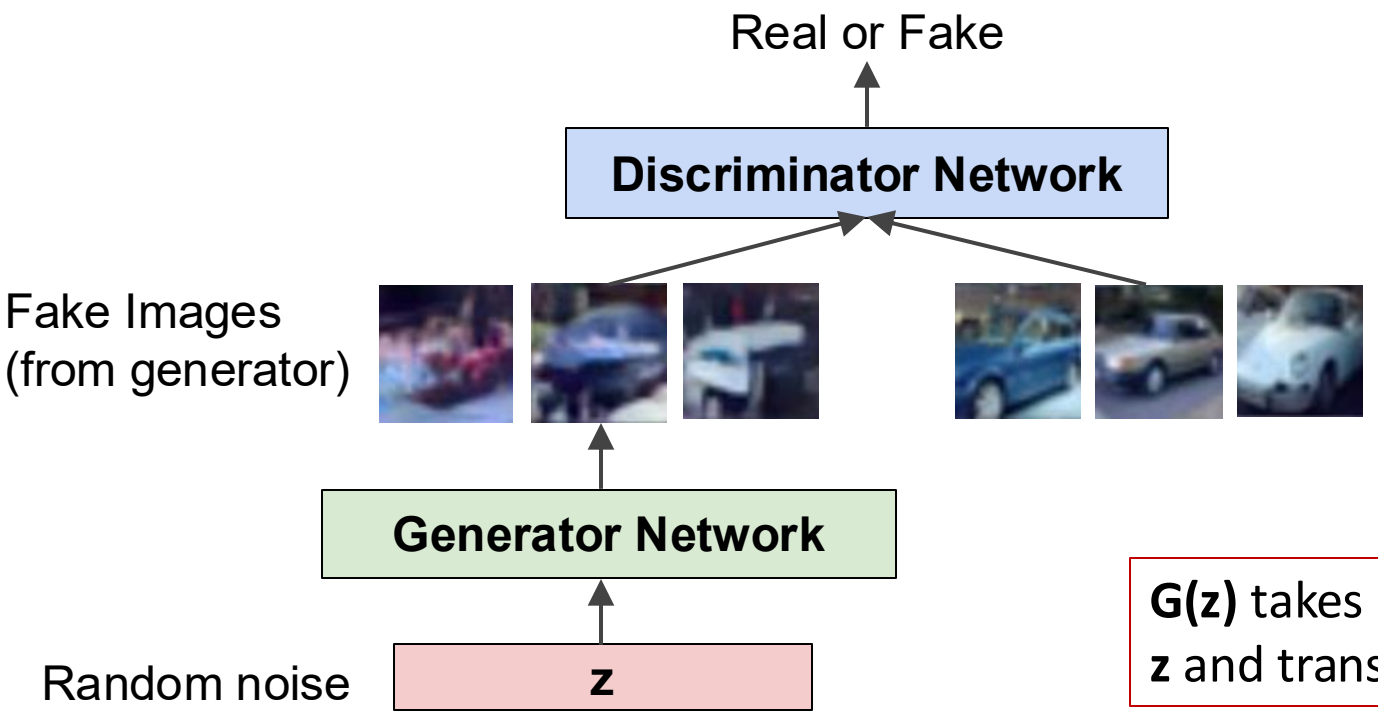
- System of two neural networks competing against each other in a zero-sum game framework.
- Can learn to draw samples from a model that is similar to training data.



First introduced by Ian Goodfellow *et al.* in 2014.

Training GANs

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



D(x) represents the probability that input **x** came from the real data rather than the Generator.

G(z) takes random input noise vector **z** and transforms it into an image

Recall: Logistic regression

- For a set of training examples with binary labels

$$\{y^{(i)}, x^{(i)}\}_{i=1}^{n}, y^{(i)} \in \{0,1\}$$

- the following cost function measures how well a given $h_w(x)$ classifies the training set:

$$J(w) = -\sum_{i=1}^{n} \left(y^{(i)} \log\left(h_w\left(x^{(i)}\right)\right) + (1 - y^{(i)}) \log\left(1 - h_w\left(x^{(i)}\right)\right)\right)$$

- Note that only one of the two terms in the summation is non-zero for each training example (depending on whether the label $y^{(i)}$ is 0 or 1).
- When $y^{(i)} = 1$ minimizing the cost function means we need to make $h_w\left(x^{(i)}\right)$ large, and when $y^{(i)} = 0$ we want to make $1 - h_w\left(x^{(i)}\right)$ large or equivalently $h_w\left(x^{(i)}\right)$ small.

Training GANs

- Minibatch consists of

  $\{x^{(i)}\}_{i=1}^{m}$ m examples from the real distribution

  $\{z^{(i)}\}_{i=1}^{m}$ m samples from a prior distribution (like a Gaussian)

- Then, from the **discriminator's perspective**, we would like to have

  $\mathrm{D}(x^{(i)}) = 1$ (real)

  $\mathrm{D}\left(G(z^{(i)})\right) = 0$ (fake)

  $$\longrightarrow \quad \max_{W_D} \sum_{i=1}^{m} \log\left(D(x^{(i)})\right) + \log\left(1 - \mathrm{D}\left(G(z^{(i)})\right)\right)$$

- But from the **generator's perspective** we would like to have

  $\mathrm{D}\left(G(z^{(i)})\right) = 1$ (fool the discriminator)

  $$\longrightarrow \quad \min_{W_G} \sum_{i=1}^{m} \log\left(1 - \mathrm{D}\left(G(z^{(i)})\right)\right)$$

Algorithm 1 Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator, k, is a hyperparameter. We used $k = 1$, the least expensive option, in our experiments.

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:

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

end for

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

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

end for

The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.

Loss function

- Formally, the training procedure is called a **minimax game**
- The minimax objective is

$$\min_{W_G} \max_{W_D} \mathbb{E}_{x \sim p_{data}} \log\big(D(x)\big) + \mathbb{E}_{z \sim p_z} \log\Big(1 - D\big(G(z)\big)\Big)$$

$$= \min_{W_G} \max_{W_D} \int p_{data}(x) \log\big(D(x)\big)\, dx + \int p_z(z) \log\Big(1 - D\big(G(z)\big)\Big)\, dz$$

- For a fixed generator, it can be shown that this minimax game has a global optimum for $p_{data}(x) = p_{generator}(x)$.
- In that case, the discriminator cannot distinguish the real from the fake, because

$$D^*(x) = Pr(\text{real}|x) = \frac{p_{data}(x)}{p_{data}(x) + p_{generator}(x)} = \frac{1}{2}$$

https://jonathan-hui.medium.com/proof-gan-optimal-point-658116a236fb

It’s a miracle that it works!!!

- G has a reinforcement learning task
  - it knows when it does good (i.e., fools D) but it is not given a supervised signal
  - reinforcement learning is hard
  - back prop through D provides G with a supervised signal; the better D is, the better this signal will be
- Can’t describe optimum via a single loss
  - Will there be an equilibrium?
- D is seldom fooled
  - but G still learns because it gets a gradient telling it how to change in order to do better the next round.

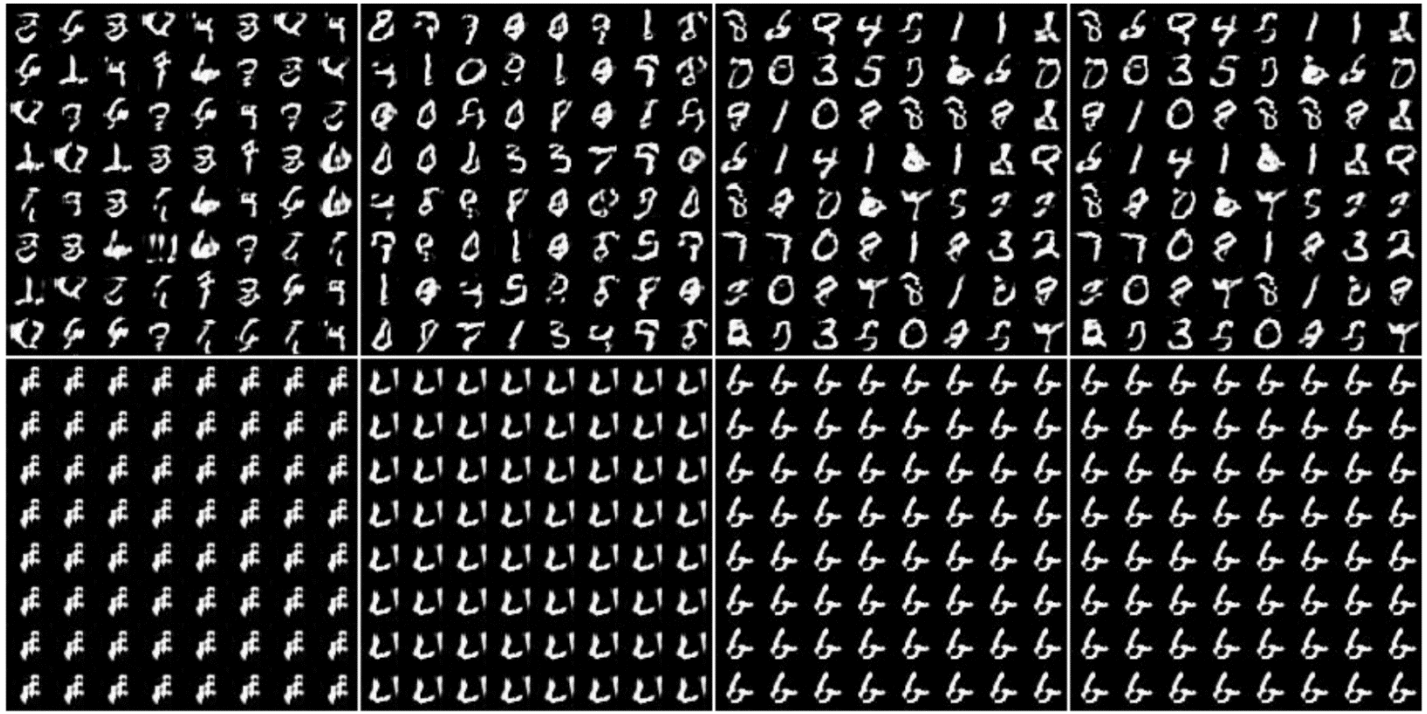
Common problems in training GANs

- GANs are notoriously hard to train! Common problems include the following:
- **Non-convergence**: the model parameters oscillate, destabilize and never converge
- **Mode collapse**: the generator collapses which produces limited varieties of samples
- **Diminished gradient**: the discriminator gets too successful, so the generator gradient vanishes (learns nothing)
- Unbalance between the generator and discriminator causing overfitting
- Highly sensitive to the hyperparameter selections.

https://jonathan-hui.medium.com/gan-why-it-is-so-hard-to-train-generative-advisory-networks-819a86b3750b

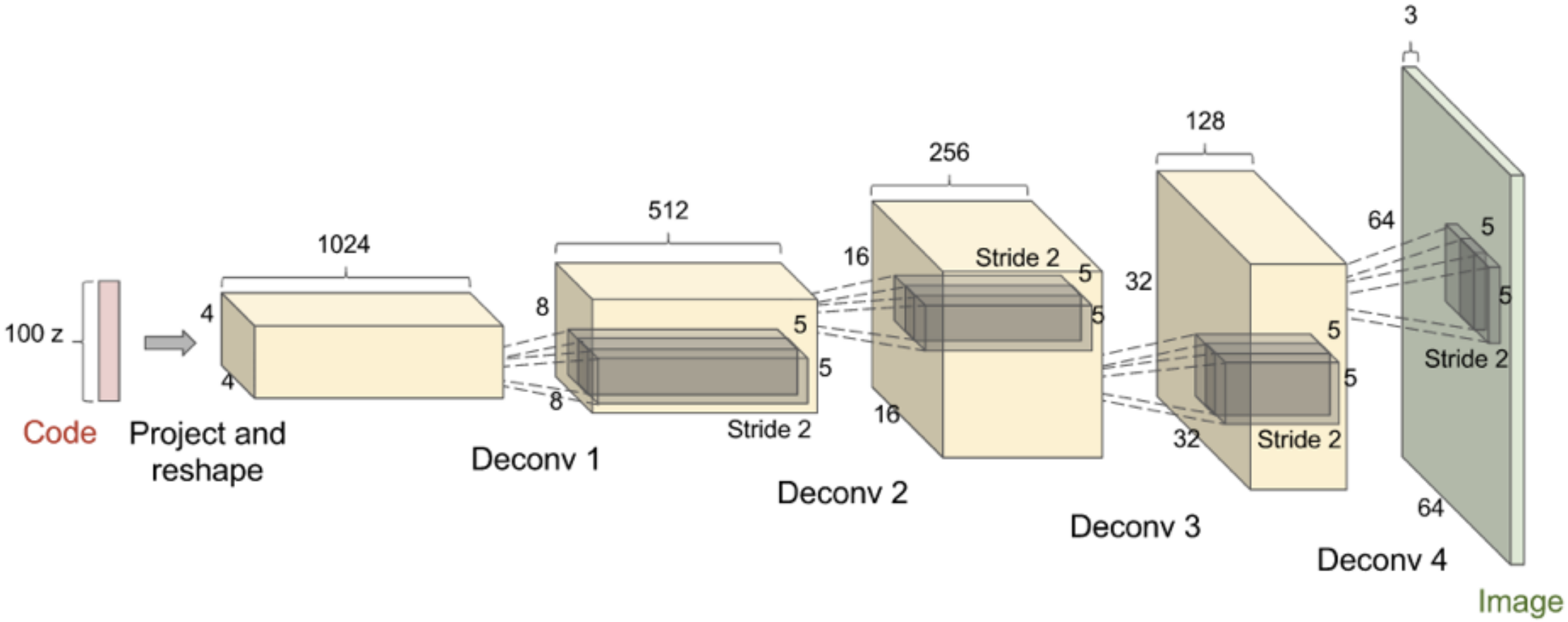
Example of mode collapse

- Real-life data distributions are multimodal. For example, in MNIST, there are 10 major **modes** from 0 to 9. The samples below are generated by two different GANs. The top row produces all 10 modes while the second row creates a single mode only (the digit “6”). This problem is called **mode collapse** when only a few modes of data are generated.



Deep Convolutional GAN (DCGAN)

- Convolutions + GANs = Good for generating images



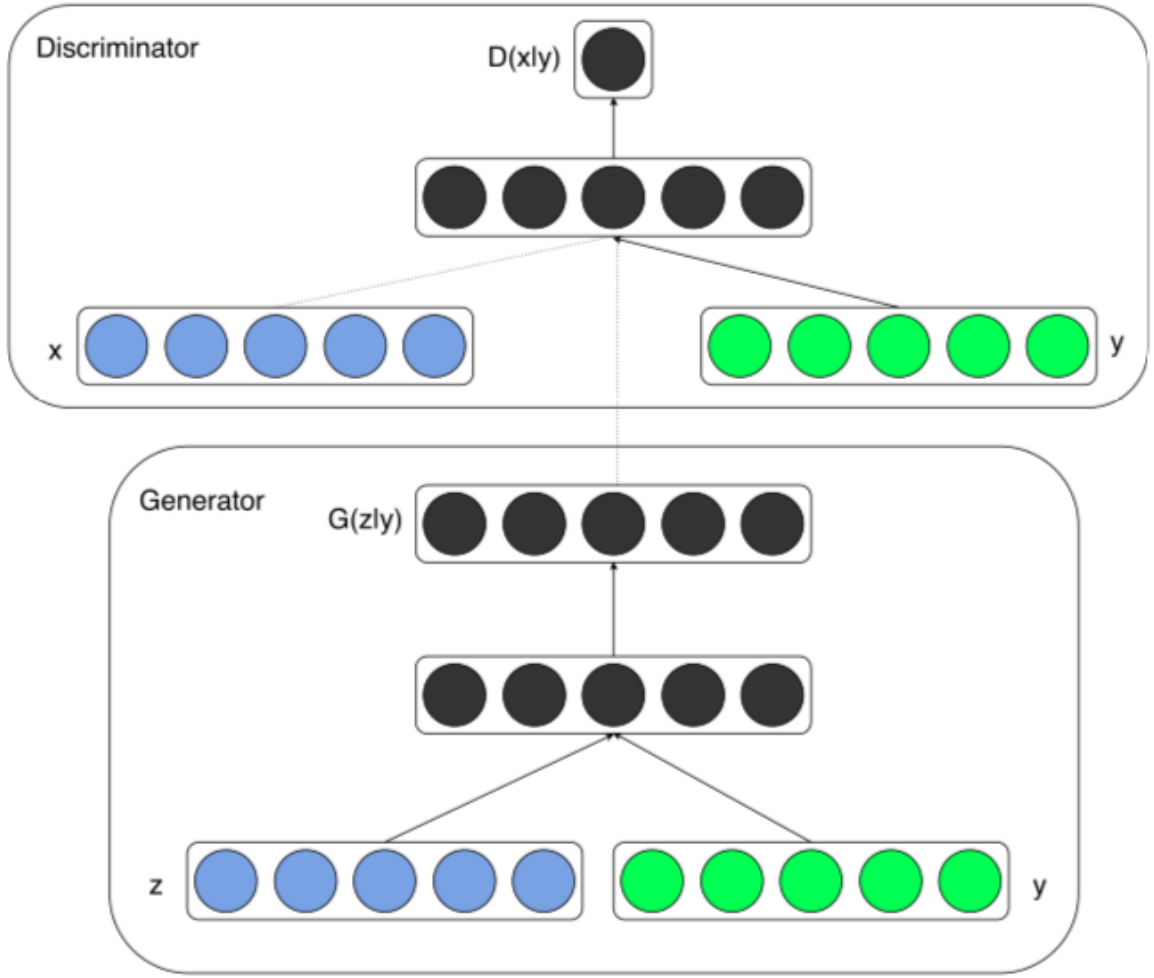
https://arxiv.org/abs/1511.06434

Deep Convolutional GAN (DCGAN)

Generated bedrooms. Source: "Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks" https://arxiv.org/abs/1511.06434v2

Conditional GAN (CGAN)

- The original GAN generates data from random noise but has no knowledge about class labels.
- CGAN aims to solve this issue by telling the generator to generate images of only one particular class, like a cat or dog.
- Specifically, CGAN concatenates a one-hot vector y to the random noise vector z to result in an architecture that looks like this:



https://arxiv.org/abs/1411.1784

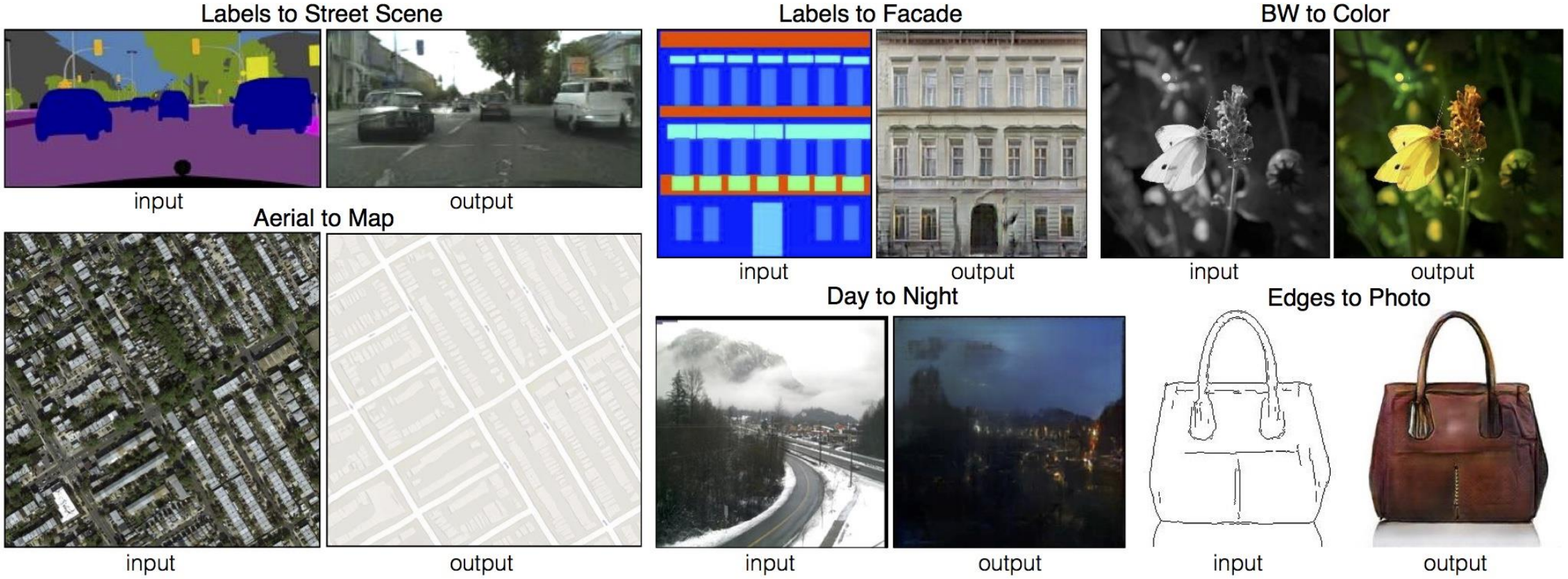
Conditional GAN (CGAN)

- From the paper

https://arxiv.org/abs/1411.1784

Pix2Pix

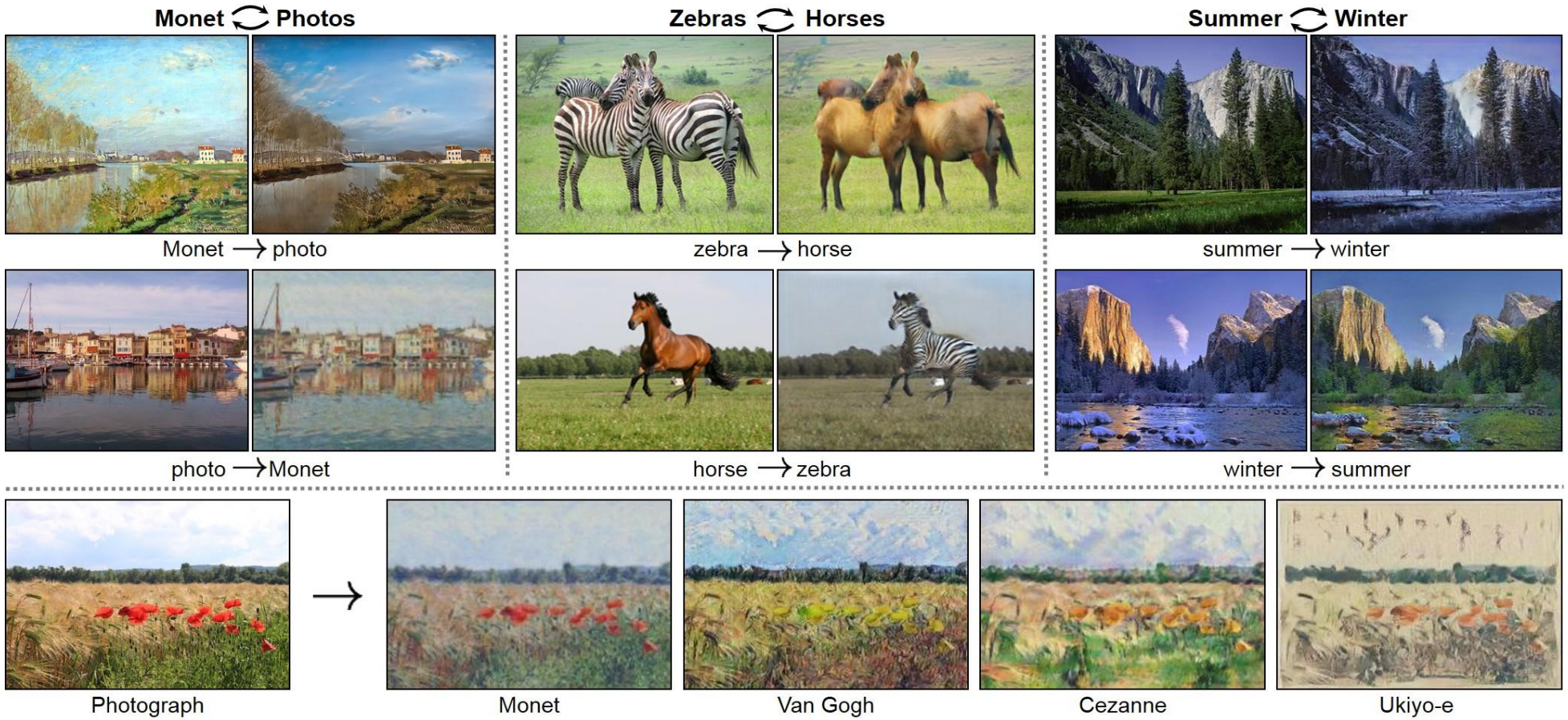
- “Image-to-Image Translation with Conditional Adversarial Networks” (homepage)
- Extension of CGAN



https://arxiv.org/abs/1611.07004

CycleGAN

- “Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks” (homepage)



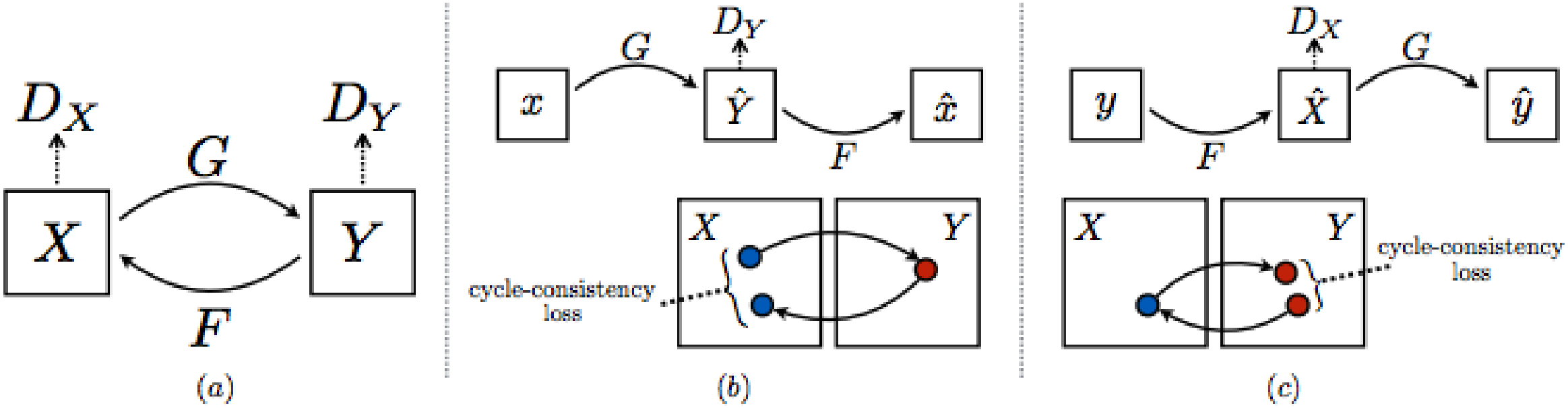
https://arxiv.org/abs/1703.10593v6

CycleGAN

- Cycle consistency loss

F(G(x)) ≈ x , x ∈ X (horse)

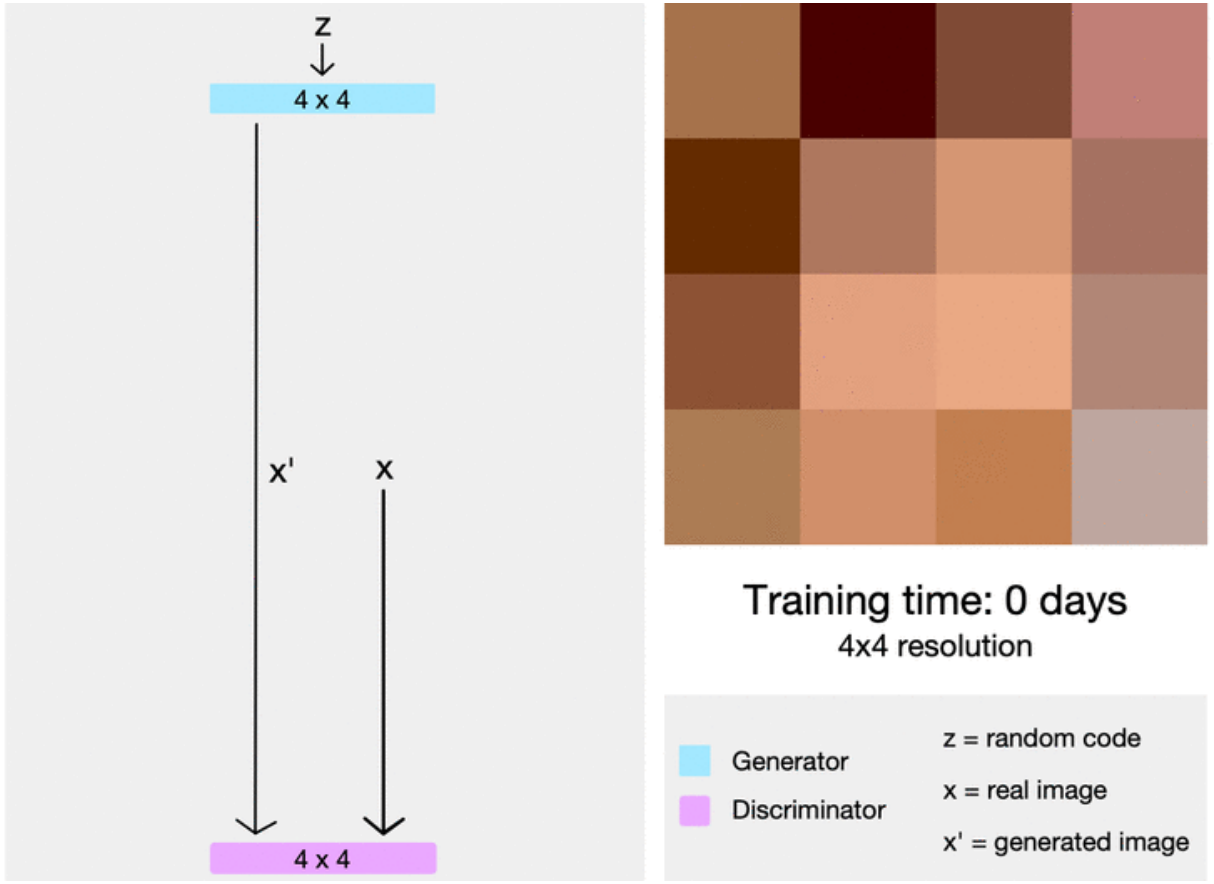
G(F(y)) ≈ y , y ∈ Y (zebra)



https://arxiv.org/abs/1703.10593v6

ProGAN

- Progressive growing of GAN – purpose is to generate high resolution images



https://cdn-images-1.medium.com/max/1600/1*tUhgr3m54Qc80GU2BkaOiQ.gif

https://arxiv.org/abs/1710.10196

WGAN

- In short, WGAN (the ‘W’ stands for Wasserstein) proposes a new cost function that has some nice properties that are all the rage for pure mathematicians and statisticians.

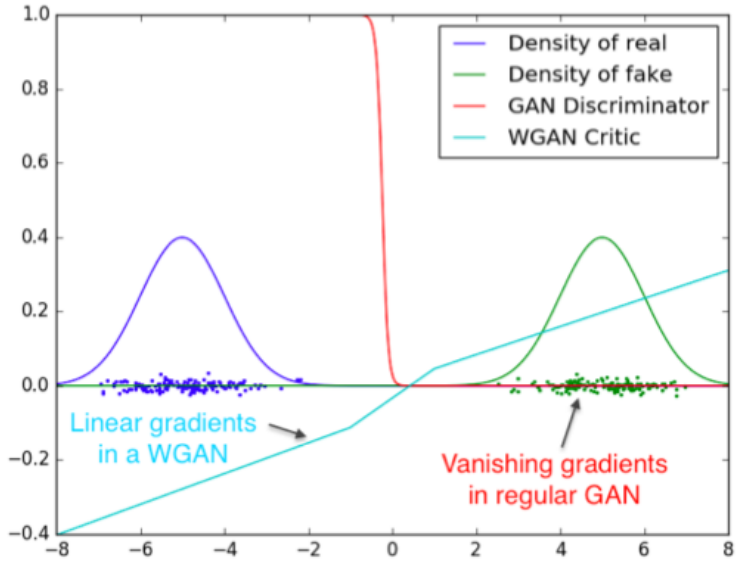


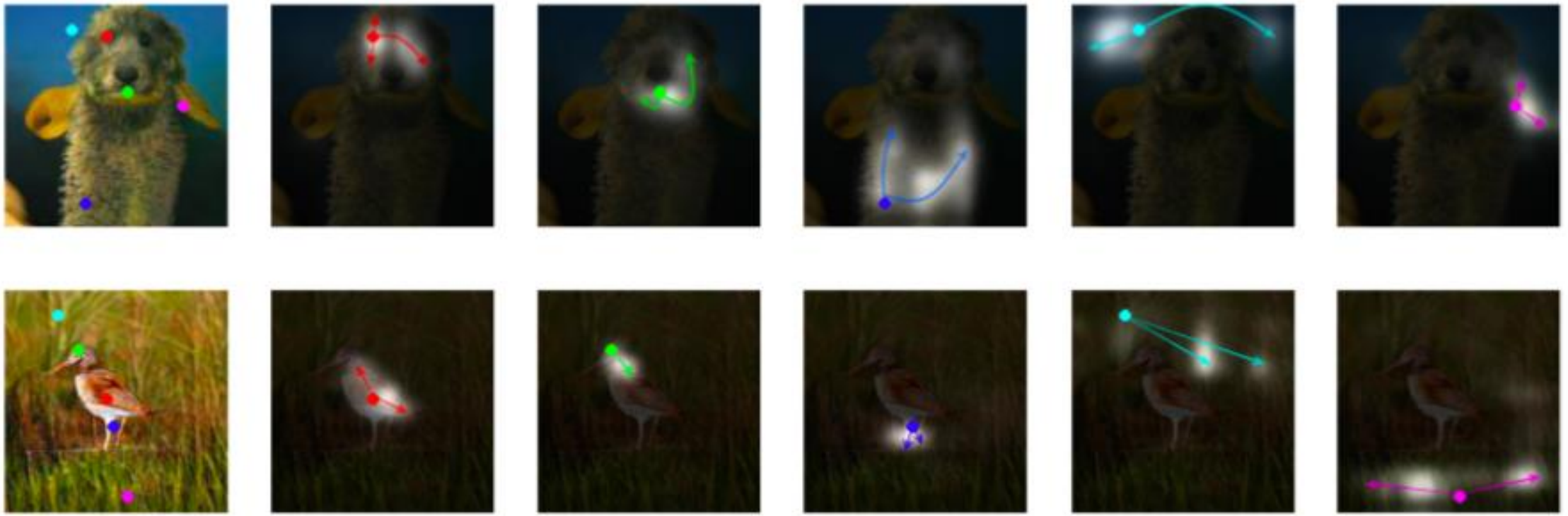
Figure 2: Optimal discriminator and critic when learning to differentiate two Gaussians. As we can see, the discriminator of a minimax GAN saturates and results in vanishing gradients. Our WGAN critic provides very clean gradients on all parts of the space.

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

https://arxiv.org/abs/1701.07875v3

SAGAN

- SAGAN: Self-Attention Generative Adversarial Networks



https://arxiv.org/abs/1805.08318v1

BigGAN

2014

2015

2016

2017

2018

https://twitter.com/goodfellow_ian/status/1084973596236144640?s=20

https://arxiv.org/abs/1809.11096v2

Summary

- **Autoencoders** – best for dimensionality reduction / feature learning. Not good at generating new samples due to discontinuities in latent space.
- **Variational Autoencoders (VAE)** – useful latent representation. Better at generating new samples, but current sample quality not the best.
- **Generative Adversarial Networks (GANs)** - game-theoretic approach, excellent samples! But can be tricky and unstable to train…
- **State-of-the-art: Diffusion model (later)**

Recommended reading

- Overview of generative models
  - https://towardsdatascience.com/deep-generative-models-25ab2821afd3
- Autoencoders
  - https://www.jeremyjordan.me/autoencoders/
- Variational autoencoders
  - https://www.jeremyjordan.me/variational-autoencoders/
  - https://www.youtube.com/watch?v=5WoItGTWV54&feature=youtu.be&t=26m32s
  - https://www.youtube.com/watch?v=uaaqyVS9-rM&feature=youtu.be&t=19m42s
- GANs
  - https://blog.floydhub.com/gans-story-so-far/
  - https://dudeperf3ct.github.io/gan/2019/04/13/Power-of-GAN/
  - https://github.com/soumith/ganhacks
  - https://github.com/hindupuravinash/the-gan-zoo
- Keras
  - https://blog.keras.io/building-autoencoders-in-keras.html
  - https://github.com/eriklindernoren/Keras-GAN