# Deep Learning
# Autoencoder

Renato Assunção - DCC - UFMG

# Plano de aula

- Assuntos das próximas 4 semanas: Classificação não-supervisionada
    - Autoencoders
    - Variational Autoencoders
    - GANs - Generative Adversarial Networks
-

# Classificação não-supervisionada

# Classificação supervisionada

- Vamos entender a diferença entre classificação <u>Supervisionada</u> e <u>Não-supervisionada</u>

- Na classificação supervisionada, existe uma variável com um status especial:
    - a variável resposta Y, o label da observação

- Podemos ter:
    - resposta ou label binário: duas classes: imagem de gato x imagem de cachorro
    - resposta multi-classe: MNIST: 10 classes, os dígitos 0, 1, …, 9
    - resposta contínua (problemas de regressão): preço de apartamento

- Existem várias outras variáveis que não tem o status RESPOSTA:
    - são as features

    - Elas ficam num vetor $\boldsymbol{x} = (x_1, x_2, \ldots, x_p)$

    - Imagens em tons de cinza: valores dos pixels (matriz-imagem colapsada num longo vetor)
    - Preço Aptos: área, ano da construção, número de quartos, número de suítes, etc.

# Classificação supervisionada

- Classif supervisionada → são modelos discriminativos:
  - Discriminar: perceber diferenças; distinguir; discernir

- Como inferir (ou "aprender") os valores da variável RESPOSTA Y num novo exemplo usando as FEATURES $\boldsymbol{x}$ ?

- Inferir como função das features $\boldsymbol{x}$

  - Obtemos uma aproximação para $\mathbb{E}(Y \mid \boldsymbol{x}) = g(\boldsymbol{x})$ = uma função matemática do vetor de features

  - Em geral, $g(\boldsymbol{x})$ é desconhecida e muito complexa

  - Aproximamos $g(\boldsymbol{x})$ por OUTRA função matemática $\hat{g}(\boldsymbol{x})$ muito mais simples que $g(\boldsymbol{x})$

  - $\hat{g}(\boldsymbol{x})$ é uma instância de modelo estatístico (regressão linear, logística, SVM, etc.)

  - $\hat{g}(\boldsymbol{x})$ é aprendida através das amostras de dados

# Classificação supervisionada

- Queremos uma aproximação para $\mathbb{E}(Y \mid \boldsymbol{x}) = g(\boldsymbol{x})$
- Usamos $\hat{g}(\boldsymbol{x}) = \hat{Y}(\boldsymbol{x})$
- g pode ser bem simples:
  - em regressão linear, predizemos o preço médio do apto com features x com uma combinação linear fixa das features:

$$\hat{g}(\boldsymbol{x}) = \hat{Y}(\boldsymbol{x}) = b + w_1 x_1 + \ldots + w_p x_p$$

  - em regressão logística, predizemos a probabilidade de ocorrer Y=1 (quando temos features x) com  uma transformação logística de uma combinação linear das features:

$$\hat{g}(\boldsymbol{x}) = \hat{Y}(\boldsymbol{x}) = \frac{1}{1+e^{-(b+w_1 x_1+\ldots+w_p x_p)}}$$

- g pode ser complexa, com vários passos hierárquicos até chegar em $\hat{Y}(\boldsymbol{x})$
  - Por exemplo, $\hat{Y}(\boldsymbol{x})$ pode ser a saída de uma rede neural

- Modelos discriminativos:

- Inferir RESPOSTA Y como função das features x usando g(x)

- Não nos interessa nada sobre como x se distribui na população.

- Não interessa saber se
  - $\mathbb{P}(X_1 > 3)$
  - $\mathbb{P}(X_1 > X_3)$
  - $\mathbb{P}(X_1 > X_3 | X_5 > 0)$

- Só nos importamos com Y: como a resposta Y responde quando x assume certo valor?

# Classificação não-supervisionada

- Cada exemplo possui p variáveis $\boldsymbol{Y} = (Y_1, \ldots, Y_p)$
- Temos um grande número n de exemplos
- Interesse está na distribuição conjunta das Y's
- Não existe uma dicotomia resposta Y versus features x
- Só existem as Y's:
    - são todas variáveis "resposta"
    - não existe um conjunto de features ao qual elas respondem

# Classificação não-supervisionada

- Interesse é descobrir estruturas não óbvias nos dados $\boldsymbol{Y} = (Y_1, \ldots, Y_p)$
- Tarefas:
  - Redução de dimensionalidade e visualização dos dados
  - Descoberta de clusters / grupos / classes → NÃO existe um rótulo para classe
  - Descoberta de estrutura (probabilistic graphical models, Bayesian networks)
  - Modelos generativos (GANs)

# Alguns dos desafios de alta dimensão

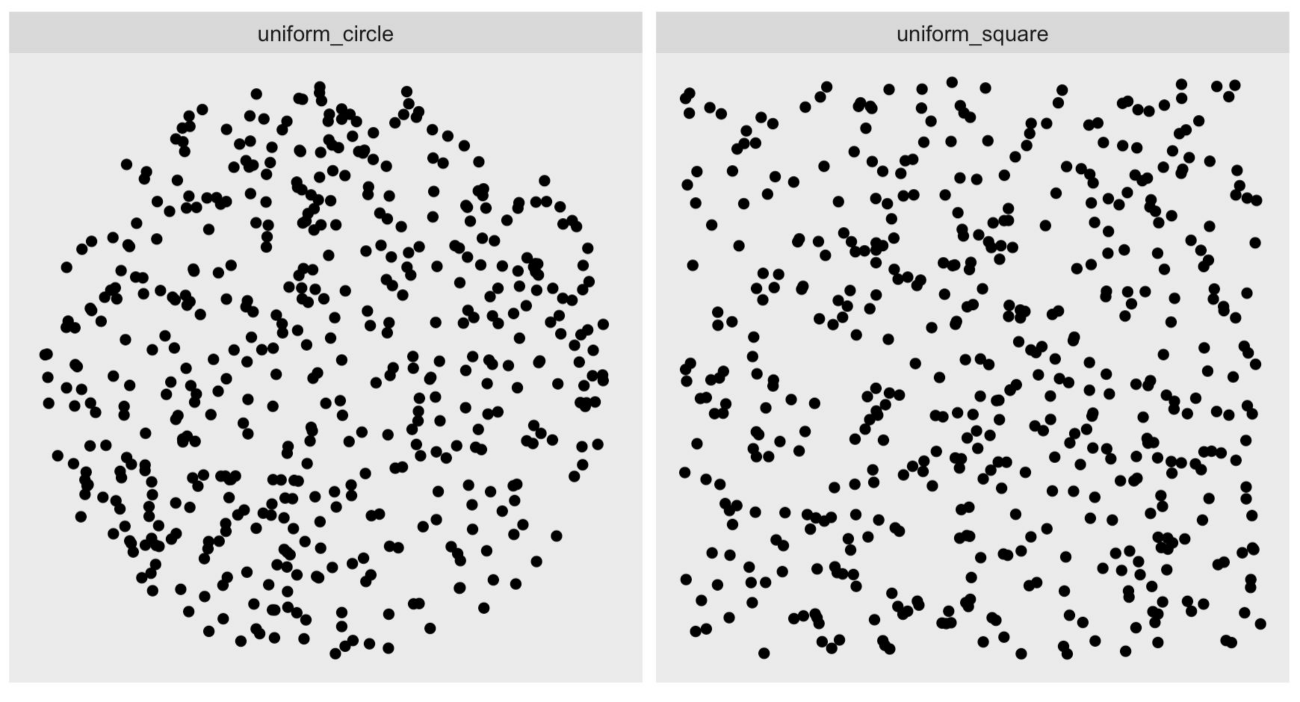
- Maldição da dimensionalidade (Dimensionality curse)

$\mathbb{R}^2$

- Dados de alta dimensão vivem na <u>fronteira</u> do espaço amostral

- <u>Maldição não ocorre</u> em dimensões menores
  - Dados aleatórios com distribuição uniforme no quadrado $[-1,1] \times [-1,1]$
  - Disco S centrado em (0,0) e com raio 1.
  - $\mathbb{P}(\boldsymbol{Y} \in S) = \pi/4 = 0.79$

# Em 2-dim



# Em 3-dim

- Em $\mathbb{R}^3$

$\mathbb{R}^2$

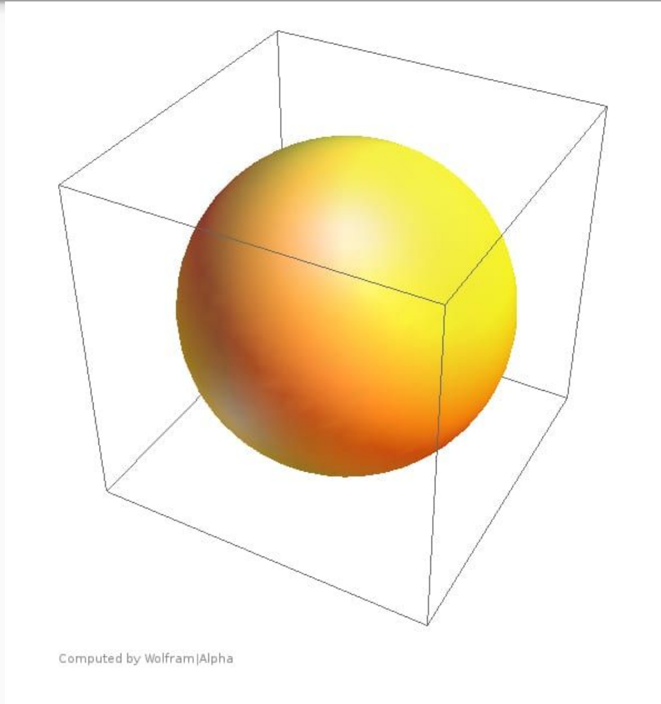
    - Dados aleatórios com distribuição uniforme no cubo tri-dimensional (lado =1)
    - Esfera S centrada em (0,0,0) e com raio 1.
    - $\mathbb{P}(\boldsymbol{Y} \in \boldsymbol{S}) = \mathbf{0.52}$
    -
    - Ao passar de 2-dim para 3-dim baixamos de 0.79 para 0.52

# Esfera S dentro de um cubo em 3-dim

S = esfera de raio 1
Cubo de lado em [-1, 1]

Y = ponto aleatório no CUBO
Y ~ distribuição uniforme no cubo

$$\mathbb{P}(\boldsymbol{Y} \in S) = 0.52$$



# Aumentando a dimensão

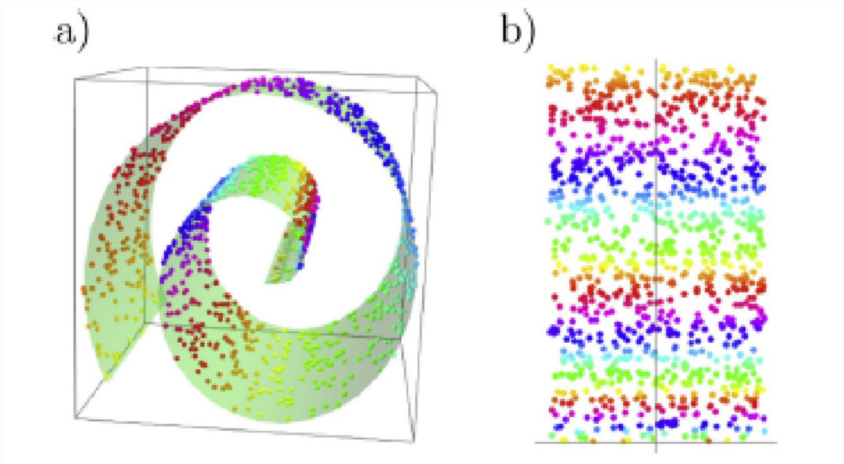
- Dados Uniformes em $[-1, 1]^d$
- Esfera S d-dim centrada em (0,...,0) e de raio 1
- $\mathbb{P}(\boldsymbol{Y} \in S) \sim \left(\frac{1}{2}\right)^d$
- Decai exponencialmente rápido para zero com o aumento da dimensão d
- Por exemplo, com d=10, a probabilidade é apenas 0.001
- Em alta dimensão, os dados uniformemente distribuídos no hiper-cubo estão nos seus “cantos” e não no miolo central.
- Esta é a "dimensionality curse"

# Maldição ou benção?

- Benção da não-uniformidade

- dados reais não são uniformemente distribuídos no espaço amostral.

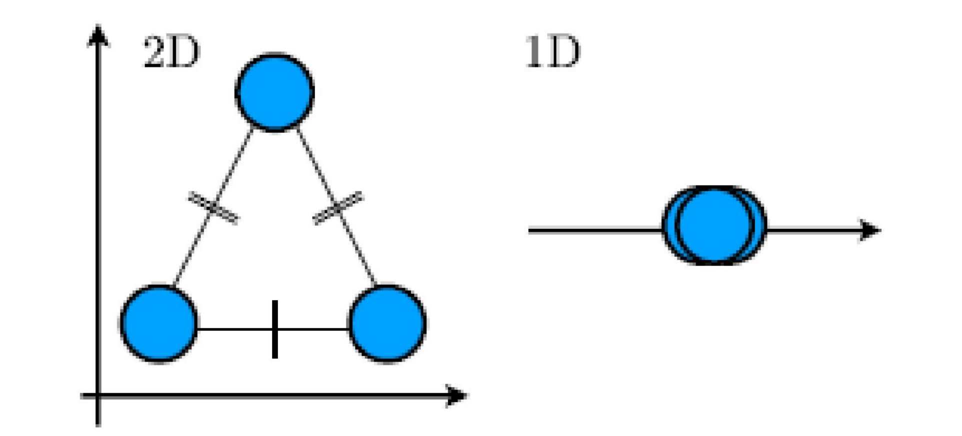
- Localmente, dados estão concentrados em “variedades diferenciáveis” (differentiable manifolds) → equiv. a superfícies em 3 dimensões




- Essas variedades possuem dimensão intrínseca bem menor que a dimensão dos vetores X

# Alguns dos desafios de alta dimensão

- Crowding problem
  - Objetivo recorrente: projetar num espaço menor mas mantendo as distâncias relativas entre os dados
  - Existe uma dimensionalidade intrínseca (mínima)
  - Forçar os dados numa dimensionalidade menor que a intrínseca leva ao colapso



# AutoEncoder (AE)

# Autoencoders

- Autoencoders = redes neurais treinadas com o objetivo de copiar o seu input para o seu output.
- Isto é, a entrada aprox = saída.
- Não é um procedimento supervisionado, com labels.
- O objetivo é reproduzir, da melhor maneira possível, a entrada.
- Estranho?? Dito assim, claro que é...
- O objetivo é aprender representações (encodings) dos dados que sejam “pequenas”.
- Esta representação “econômica” vai servir para:
    - sugerir AUTOMATICAMENTE features que representam bem as entradas
    - redução de dimensionalidade: representar os dados de entrada de maneira econômica
- A essência do processo é um algoritmo em dois passos:
    - 1) represente a entrada de forma comprimida via uma rede encoder
    - 2) recupere a entrada usando a representação comprimida

# Autoencoder

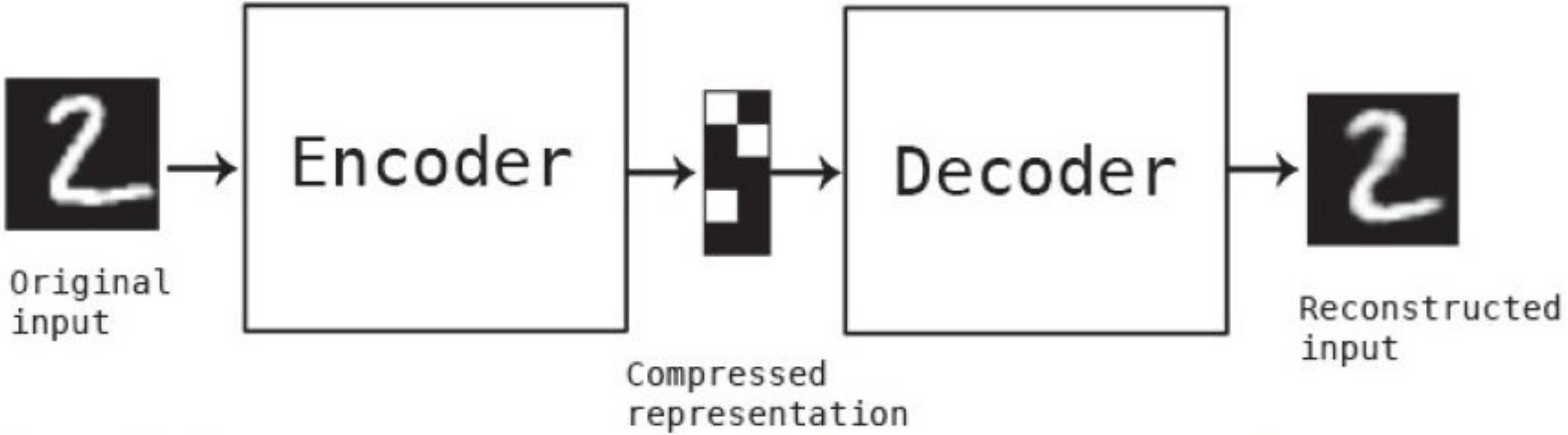
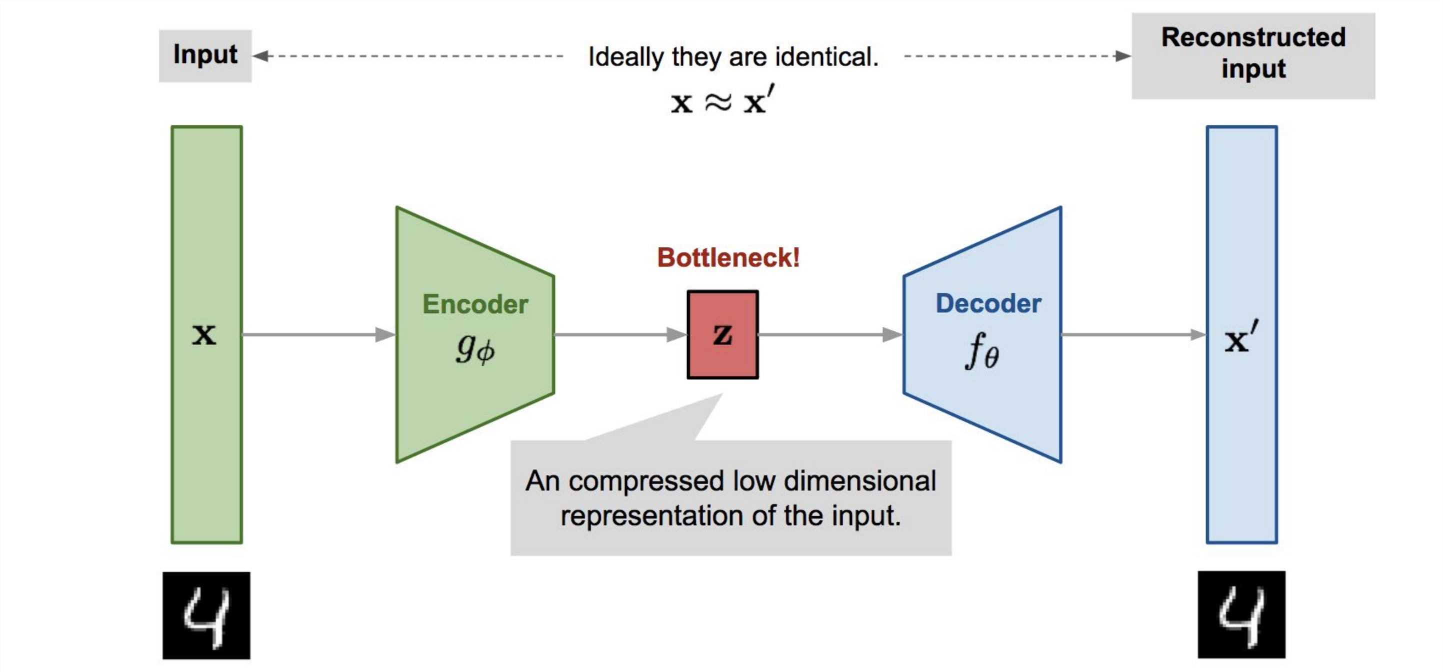


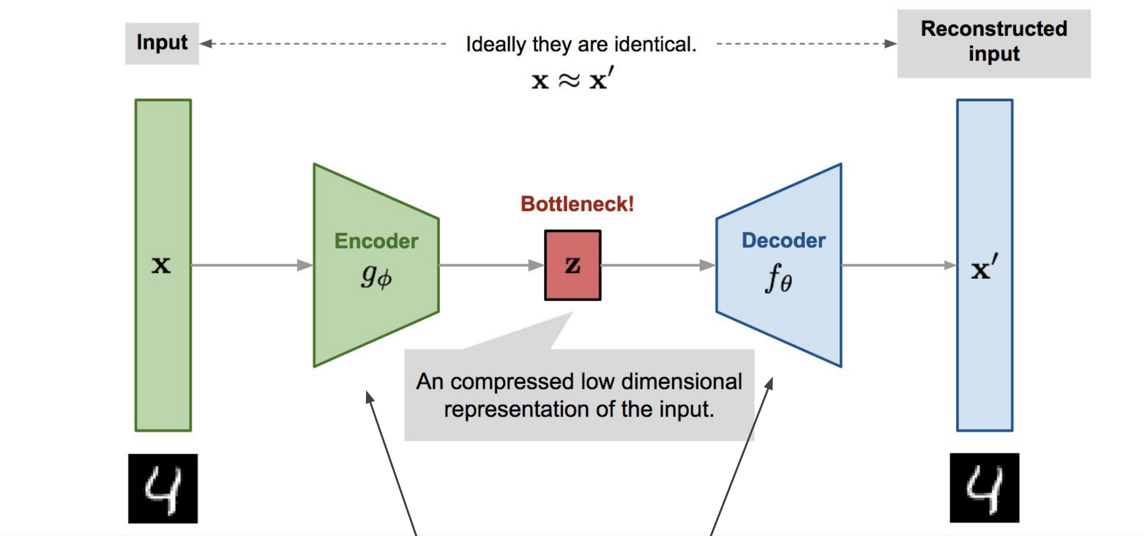

**Figure 8.12 An autoencoder: mapping an input x to a compressed representation and then decoding it back as x'**

From Chollet and Allaire (2017) Deep Learning with R.

# Autoencoder: framework geral



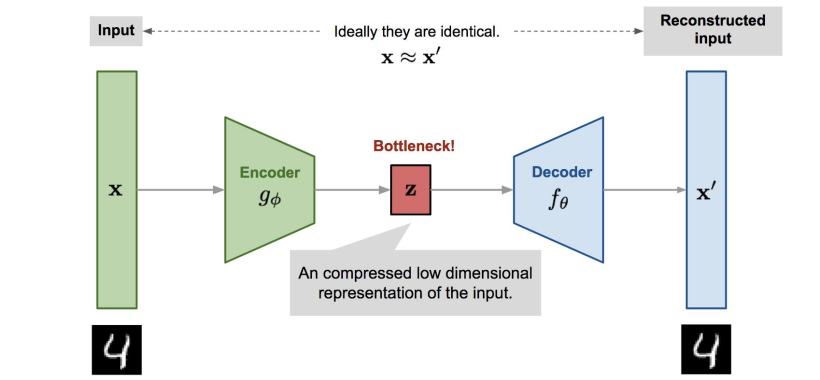
# Autoencoder: framework geral

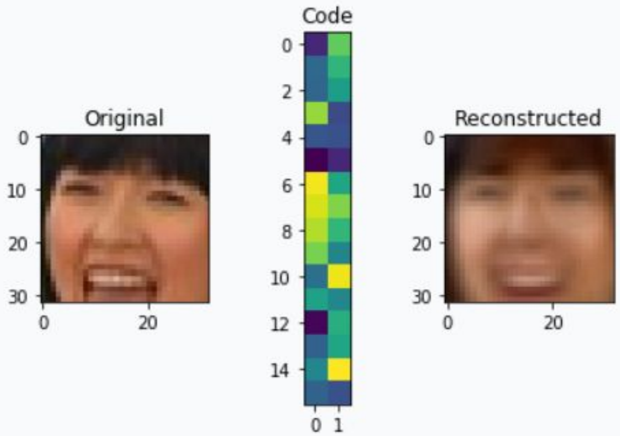



Em geral, o encoder e decoder serão DUAS redes neurais

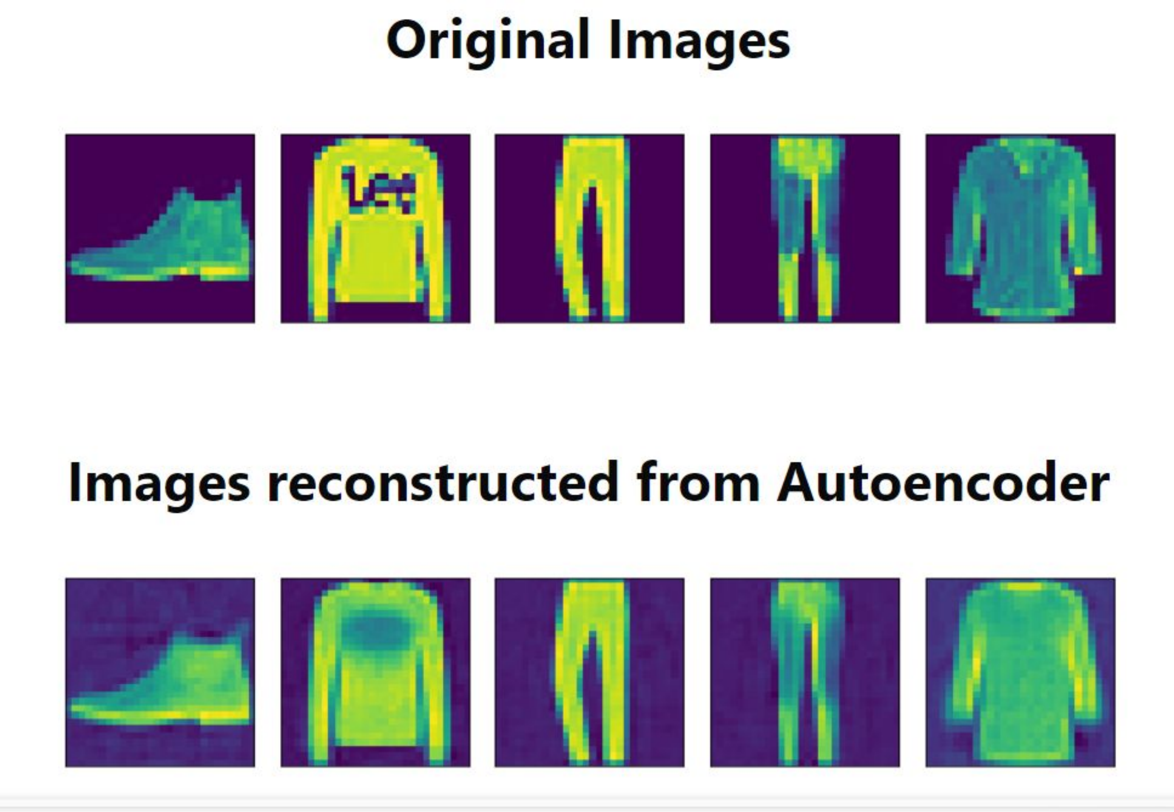
$\phi$ e $\theta$ são os pesos das duas redes

# Autoencoder: framework geral



# Fashion-MNIST



# PCA visto como um (linear) encoder

- PCA pode ser visto como um encoder que usa apenas funções LINEARES

- Seja X o vetor de dimensão n
- Com amostras de X, obtivemos matriz de covariância S (n x n) e os seus **n** autovetores

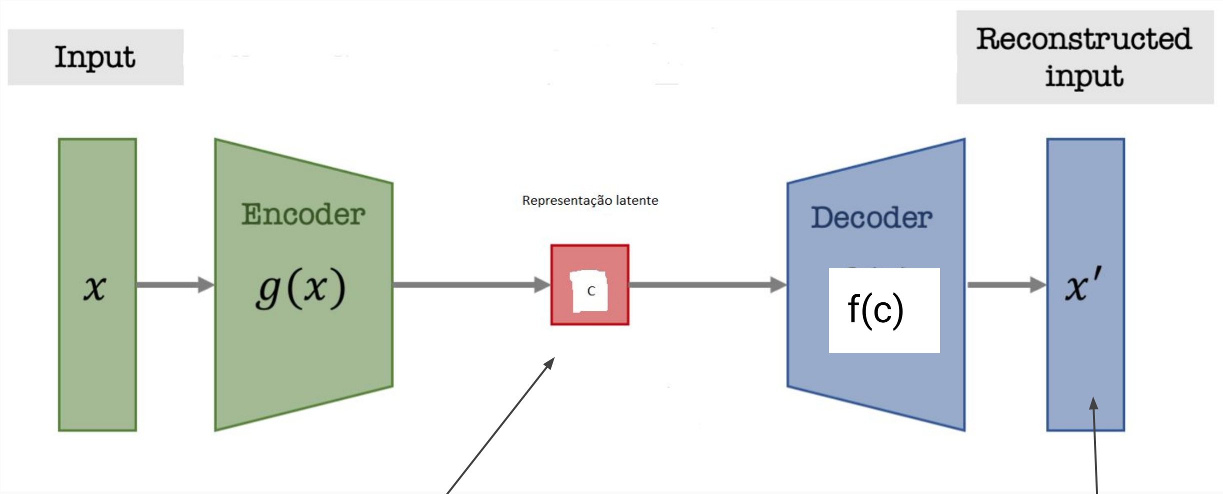
- Dentre estes n, ficamos com K deles (esperamos K << n)
- Escolhemos os autovetores com os maiores autovalores
- Cada um deles é um vetor-coluna de dimensão n

- Monte a matriz ***n x k*** com os K autovetores como colunas: $V = [v_1 \mid v_2 \mid \ldots \mid v_k]$
- A representação de X é o vetor-coluna k-dim

$$\underbrace{\boldsymbol{c}}_{k \times 1} = \begin{bmatrix} c_1 \\ \vdots \\ c_k \end{bmatrix} = \underbrace{\boldsymbol{V}^t \boldsymbol{x}}_{k \times n \;\; n \times 1}$$

# PCA visto como um (linear) encoder

- Monte a matriz ***n x k*** com os **K** primeiros autovetores como colunas (K << n):
- $V = \left[ v_1 \mid v_2 \mid \ldots \mid v_k \right]$

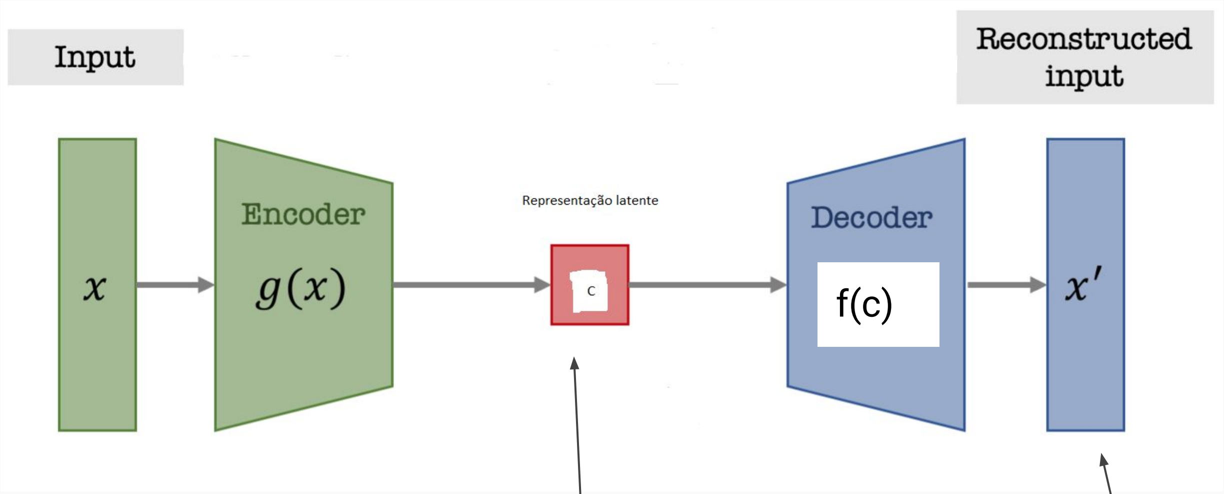



$$g(x) = \underbrace{\boldsymbol{V}^t}_{k \times n} \underbrace{\boldsymbol{x}}_{n \times 1} = \underbrace{\boldsymbol{c}}_{k \times 1}$$

$$f(c) = \underbrace{\boldsymbol{V}}_{n \times k} \underbrace{\boldsymbol{c}}_{k \times 1} = \underbrace{\boldsymbol{x}'}_{n \times 1} \neq \boldsymbol{x}$$

# PCA visto como um (linear) encoder

- Monte a matriz ***n x k*** com os **K** primeiros autovetores como colunas:
- $V = \left[ v_1 \mid v_2 \mid \ldots \mid v_k \right]$



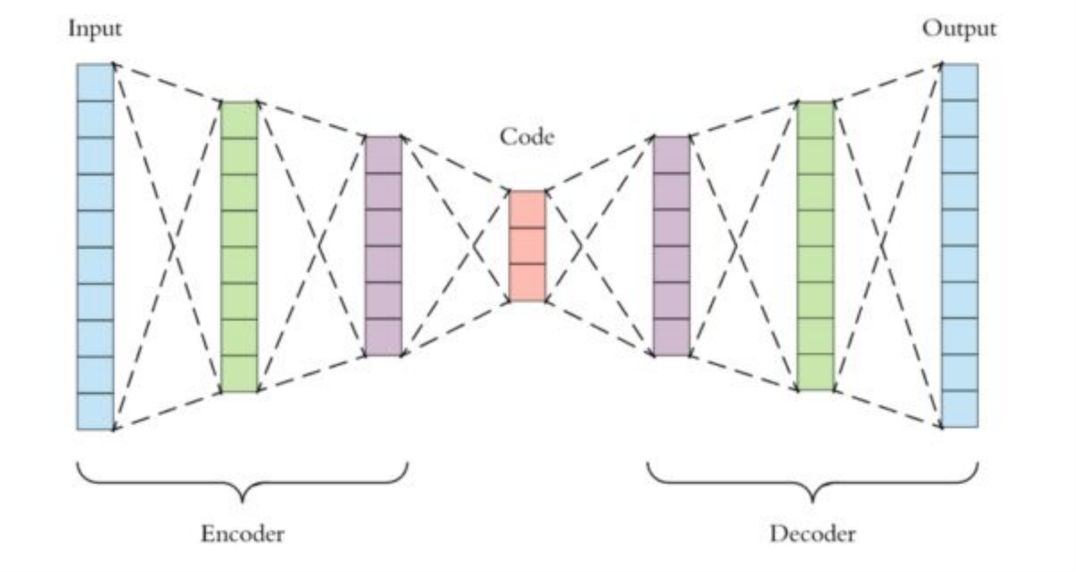

$$\underbrace{\boldsymbol{c}}_{k\times 1} = \begin{bmatrix} c_1 \\ \vdots \\ c_k \end{bmatrix} = \underbrace{\boldsymbol{V}^t \boldsymbol{x}}_{k\times n \;\; n\times 1}$$

$$\underbrace{\hat{\boldsymbol{x}}}_{n\times 1} = \underbrace{\boldsymbol{V}\boldsymbol{c}}_{n\times k \;\; k\times 1}$$

# Arquitetura de autoenconders

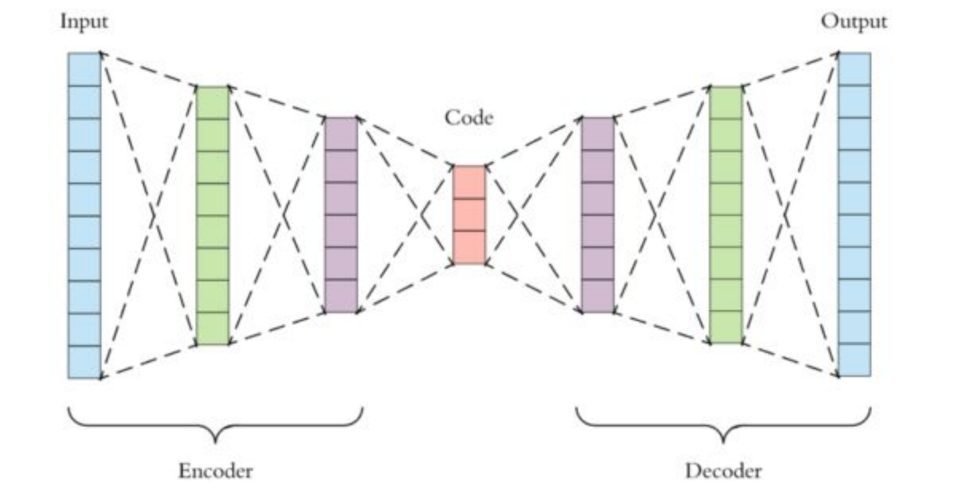
# Arquitetura dos autoencoders gerais

- Um Autoencoder consiste de três camadas:
  - Encoder: rede neural em várias camadas
  - Code: representação de baixa dimensionalidade
  - Decoder: outra rede neural em várias camadas



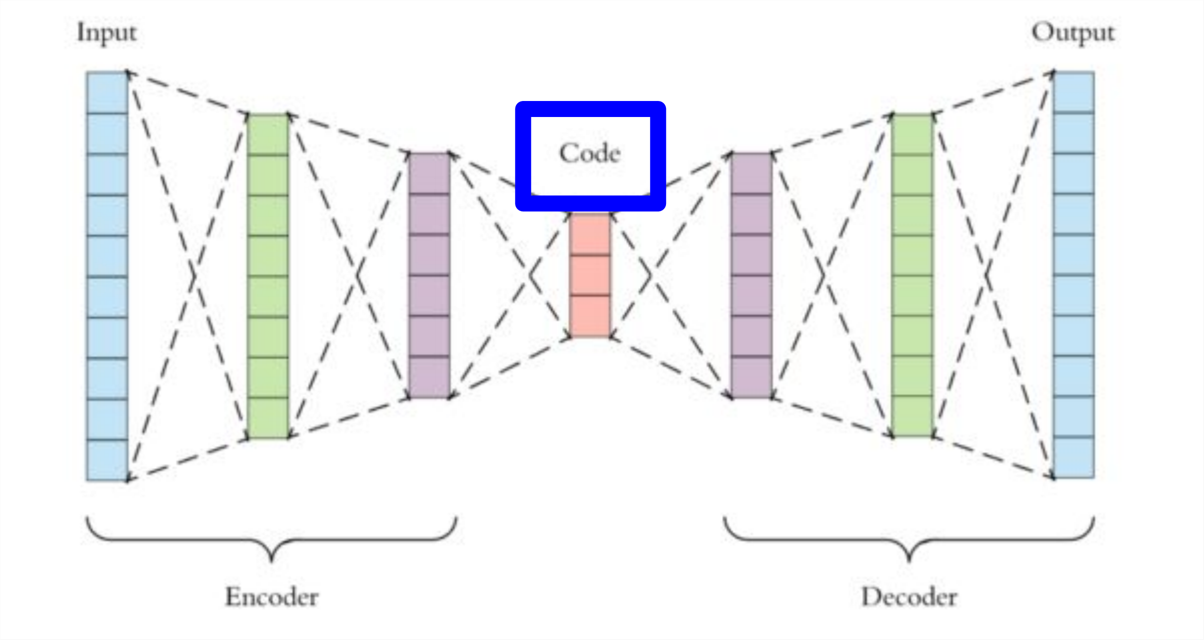
# Arquitetura dos autoencoders

- Camada do Encoder:
  - Comprime a entrada em uma representação de espaço latente (CODE) de dimensão reduzida.
  - A entrada comprimida é uma versão (muito) distorcida da entrada original.



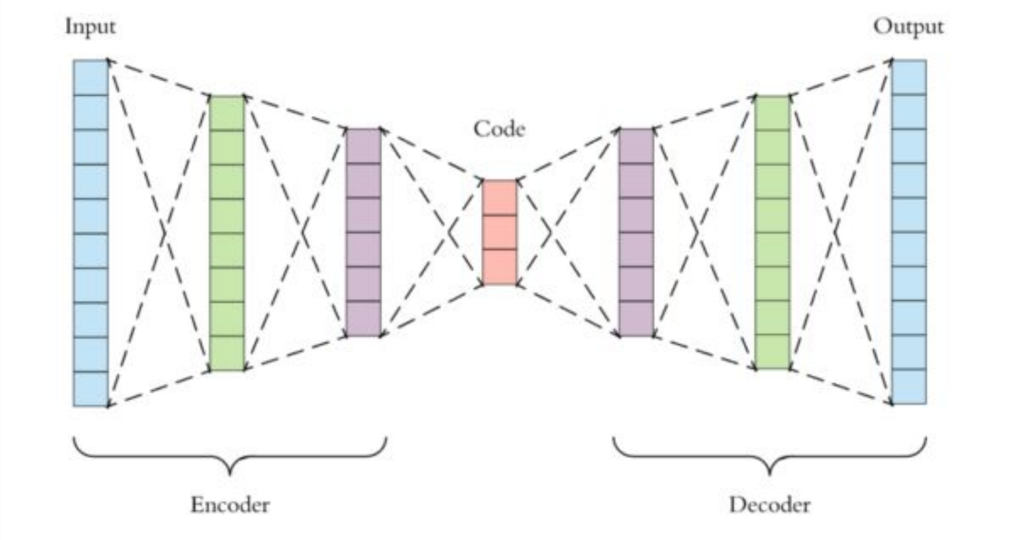
# Arquitetura dos autoencoders

- Camada Code:
  - Representa a entrada comprimida com um vetor de dimensão pequena.
  - Ela será o input da camada decoder



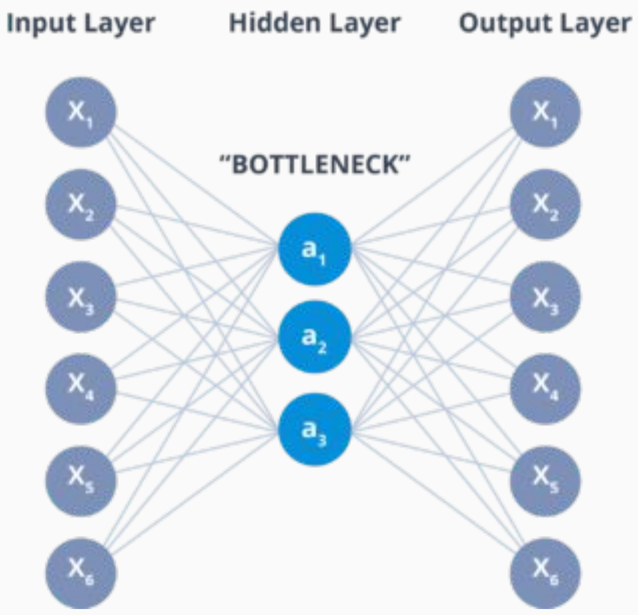
# Arquitetura dos autoencoders

- Camada Decoder
  - Decodifica a representação (a entrada codificada) de volta para a dimensão original.
  - A saída decodificada (OUTPUT) é uma reconstrução com perdas da entrada original (INPUT)
  - É reconstruída a partir da representação do espaço latente (CODE).



# Bottleneck

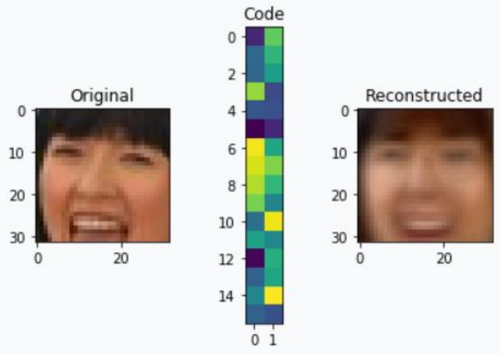
- A camada CODE também é conhecida como gargalo (bottleneck).
- <u>É útil se tiver muito menos neurônios que a entrada (e a saída)</u>
- Rede ENCODER:
  - descobre (aprende) quais aspectos da entrada x são informações relevantes para representá-la
- REDE DECODER:
  - descobre como passar do CODE para um output fiel ao input x
- Isso é feito equilibrando:
  - máximo de compacidade (compressão) de representação.
  - Máximo de fidelidade na saída.

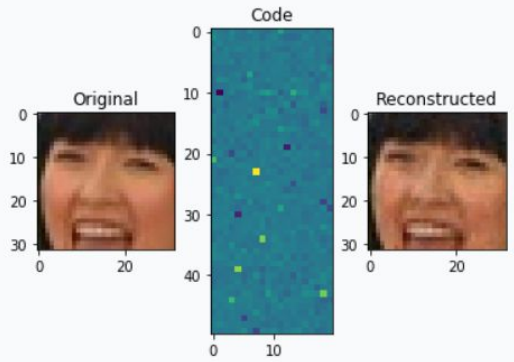


# Bootleneck - Trade-off

- Queremos equilibrar:
  - máximo de compacidade (compressão) de representação.
  - Máximo de fidelidade na saída.
- São objetivos antagônicos: aumentar um diminui o outro

- Veja ao lado o que acontece;
  - aumentamos fidelidade
  - MAS diminuímos compacidade



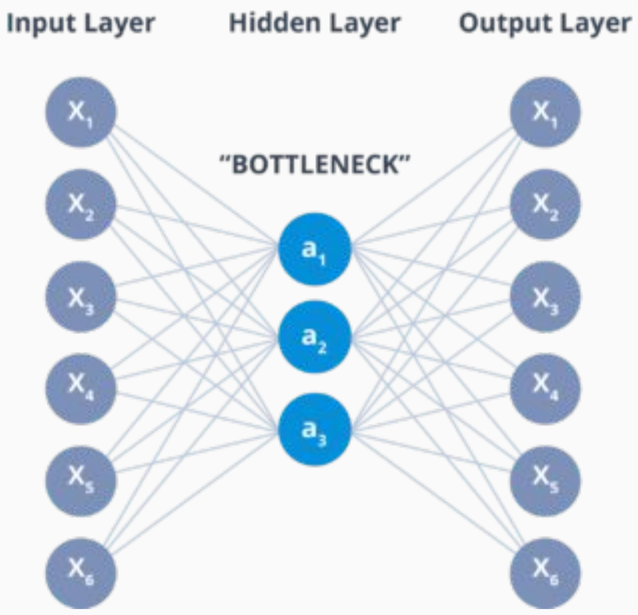



# Bottleneck

- Queremos equilibrar:
  - compressão de representação.
  - fidelidade na saída.

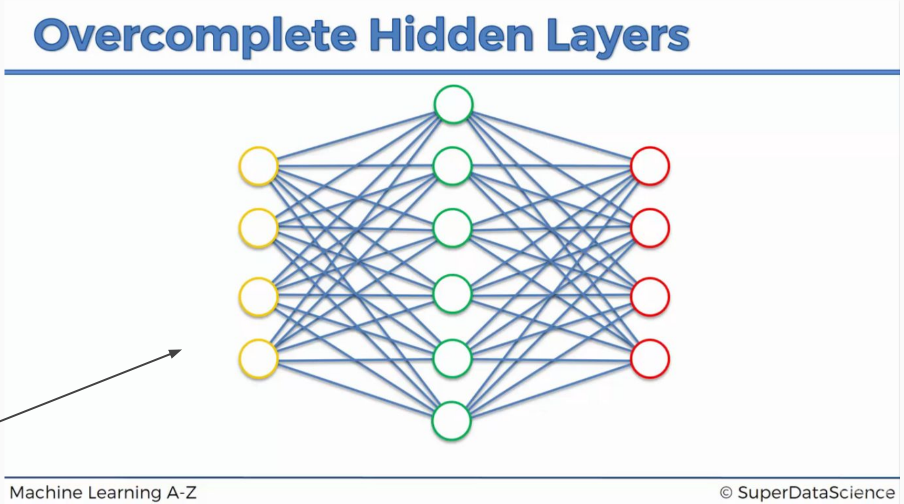
- Compacidade do CODE = dimensão do bottleneck
  - Reduz vetor x de dimensão grande para vetor CODE de dimensão pequena
  - Vai perder informação sobre x no processo.
  - DECODER não recupera x perfeitamente.

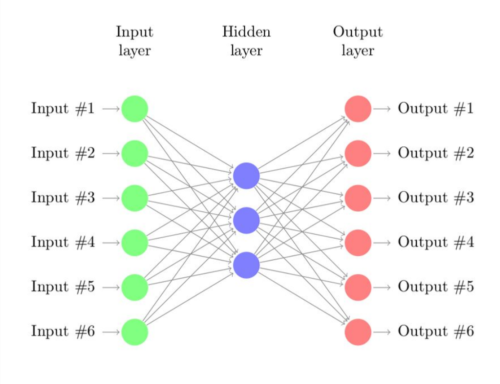
- Trade-off:
  - quanto mais compacto o CODE → mais informação do input é descartada → menos fidelidade na saída



# Overcomplete - subcomplete

- Em princípio, CODE pode ter dimensão maior que a saída. Isto não é útil.
- Rede será tão rica que haverá overfitting aos dados de entrada. Too much capacity.
- Basta zerar certos neurônios e tomar a função identidade com o resto.
- Jargão:
  - Autoencoders overcomplete
  - Autoencoders undercomplete:
    - aka sparse autoencoders



# Propriedades (de https://blog.keras.io/building-autoencoders-in-keras.html)

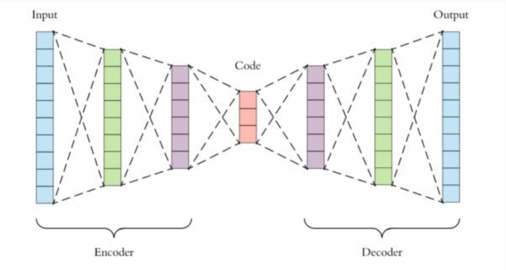
- Data-specific:
  - em geral, são capazes de compactar dados semelhantes aos do treinamento.
  - Autoencoder treinado em imagens de rostos pode ser ruim para compactar fotos de árvores, porque as redes aprendidas seriam específicas para modelar rosto.

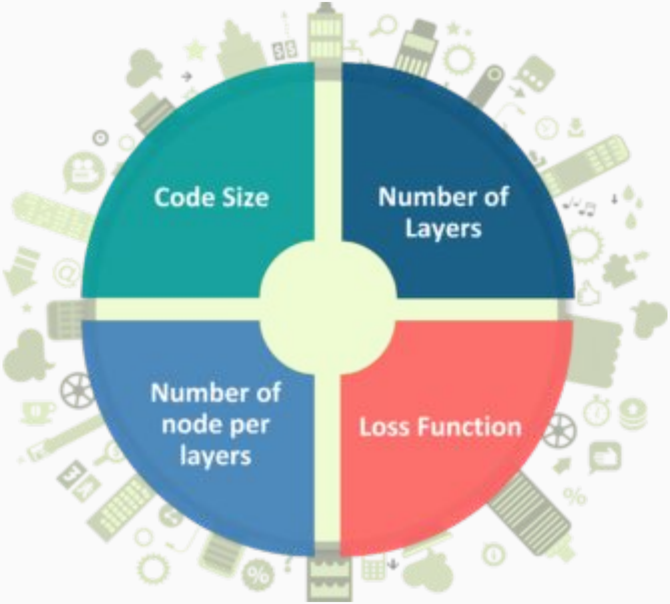
- Lossy:
  - saídas decodificadas serão degradadas em comparação com as entradas originais

- Aprende automaticamente com exemplos:
  - fácil treinar instâncias especializadas do algoritmo que terão um bom desempenho em um tipo específico de entrada.
  - Não requer nenhuma nova engenharia de features, apenas dados de treinamento apropriados.

# 4 Hiperparâmetros dos autoencoders

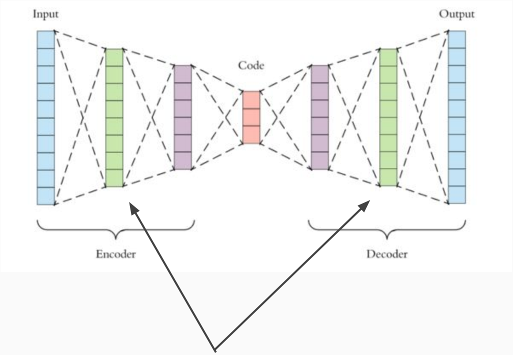
- Tamanho do code = número de nós na camada CODE  (saída do encoder, entrada do decoder).
  - Tamanho menor → mais compactação mas menos fidelidade na saída
- Número de camadas do encoder: arbitrário
- Número de nós por camada: vai diminuindo até chegar no code e depois vai aumentando de novo até a saída.



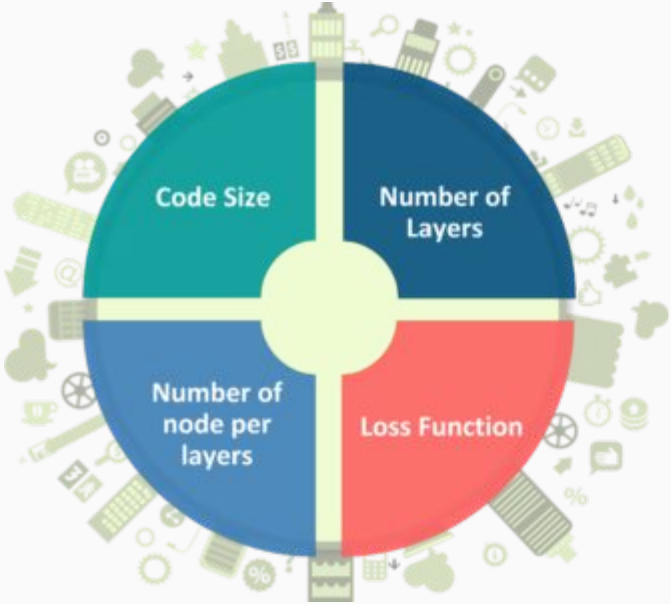



# 4 Hiperparâmetros dos autoencoders

- Podemos usar CNN se for entrada for imagem ou RNN se entrada for texto
- Usualmente, o decodificador é a imagem espelhada do codificador.




uma é a versão “espelhada” da outra

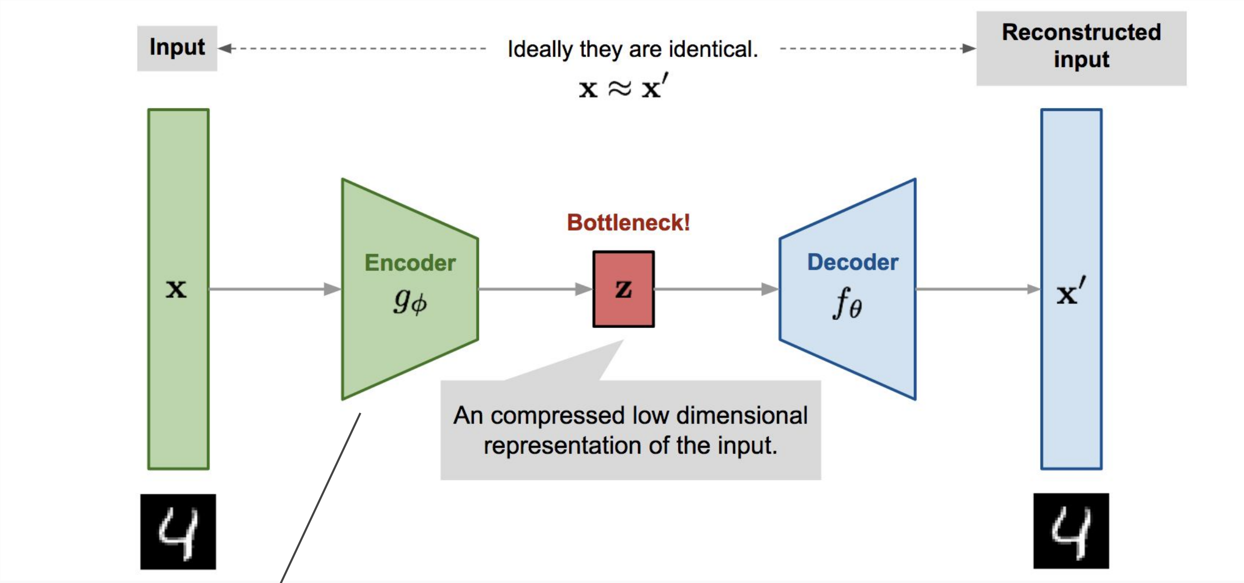



- Função de perda: depende dos dados de entrada(= saída):
  - A seguir...

# Loss functions para Autoencoders

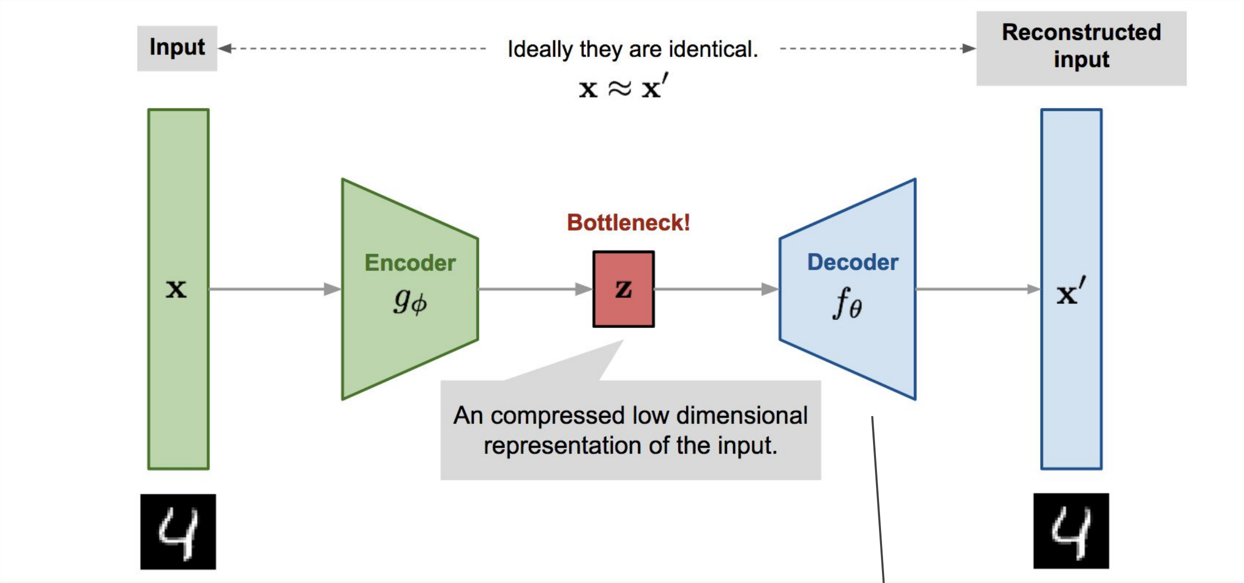
- Idealmente, queremos balancear um trade-off:
  - Deve ser sensível à cada input para obter um bom output (boa reconstrução)
  - Ao mesmo tempo, deve ser insensível aos inputs para que o modelo simplesmente não memorize ou ajuste demais aos dados de treinamento.
- Loss function = L1 + L2
  - L1 = distância entre $x$ e $\hat{x}$ (qualidade da reconstrução: = 0 se $x = \hat{x}$)
  - L2 = regularização para evitar overfitting ao dados de treino

# Relembrando notação




$\phi$

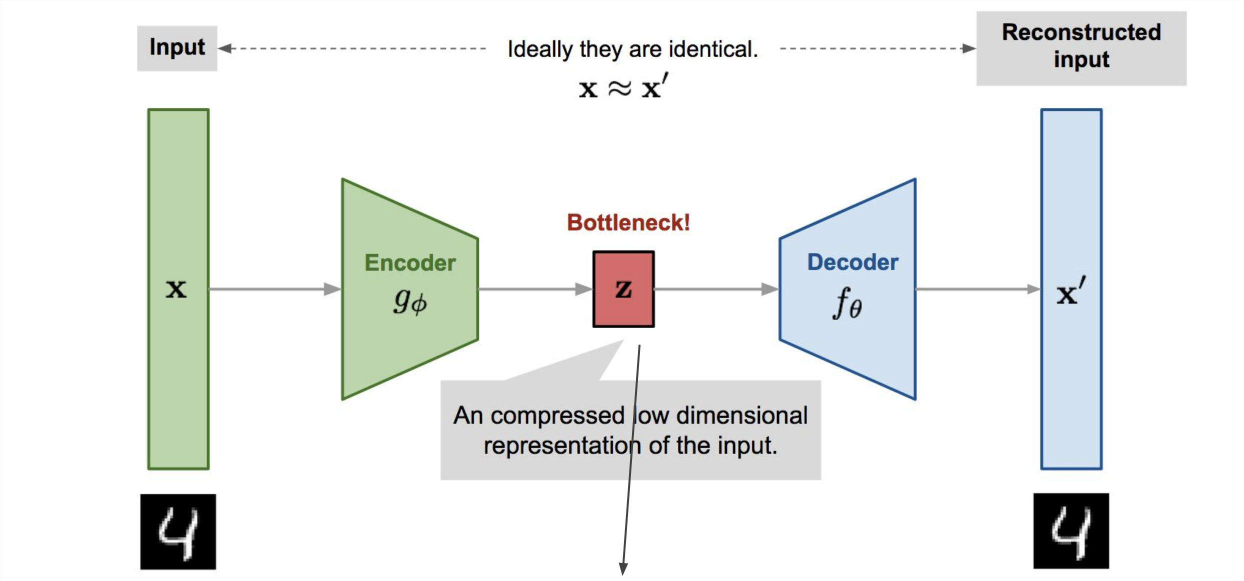
Parâmetros (pesos) da rede neural do encoder




$\boldsymbol{\theta}$

Parâmetros (pesos) da rede neural do decoder

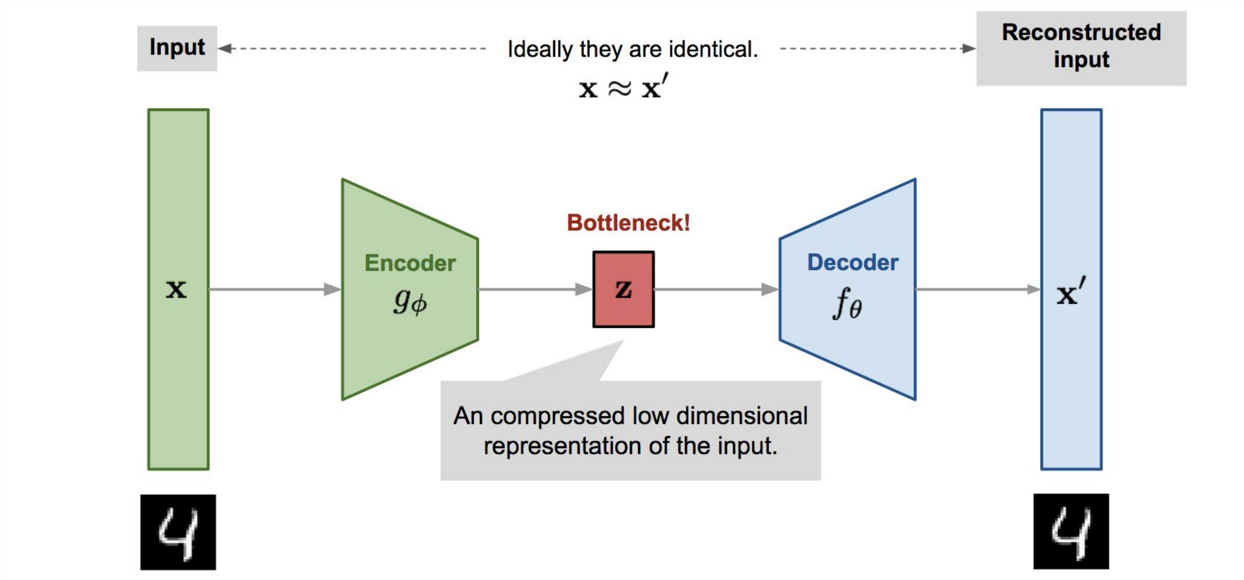
# Relembrando notação




$$\boldsymbol{z} = g_\phi(\boldsymbol{x})$$

Saída do encoder é o vetor z com a representação latente de x

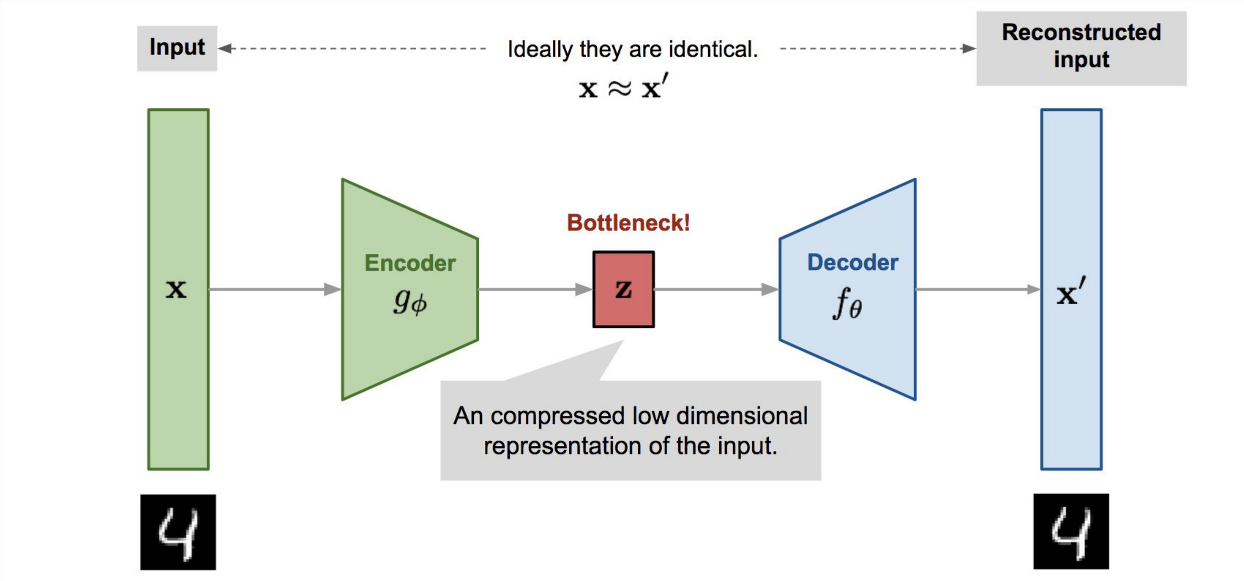
# Relembrando notação




$$\hat{\boldsymbol{x}} = f_{\boldsymbol{\theta}}(\boldsymbol{z})$$

Saída do decoder é a aproximação x-hat

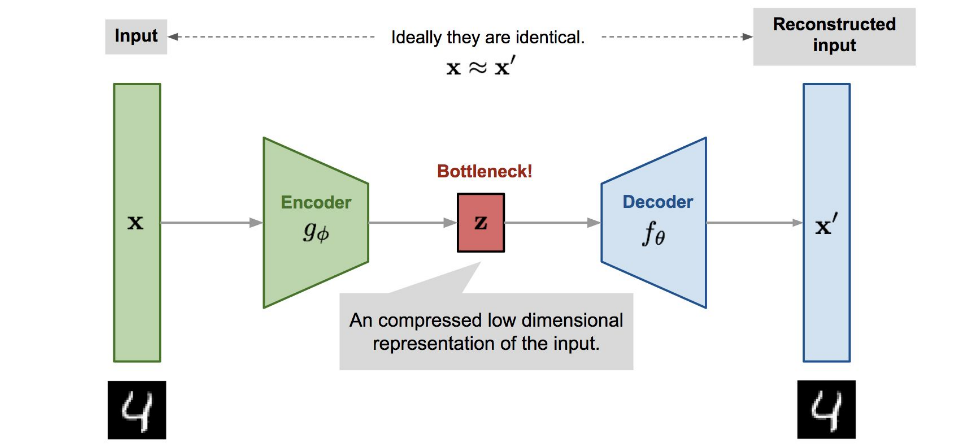
# Perda: primeiro termo




Perda deve incluir a distância entre x e sua reconstrução x-hat

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

# Perda: primeiro termo



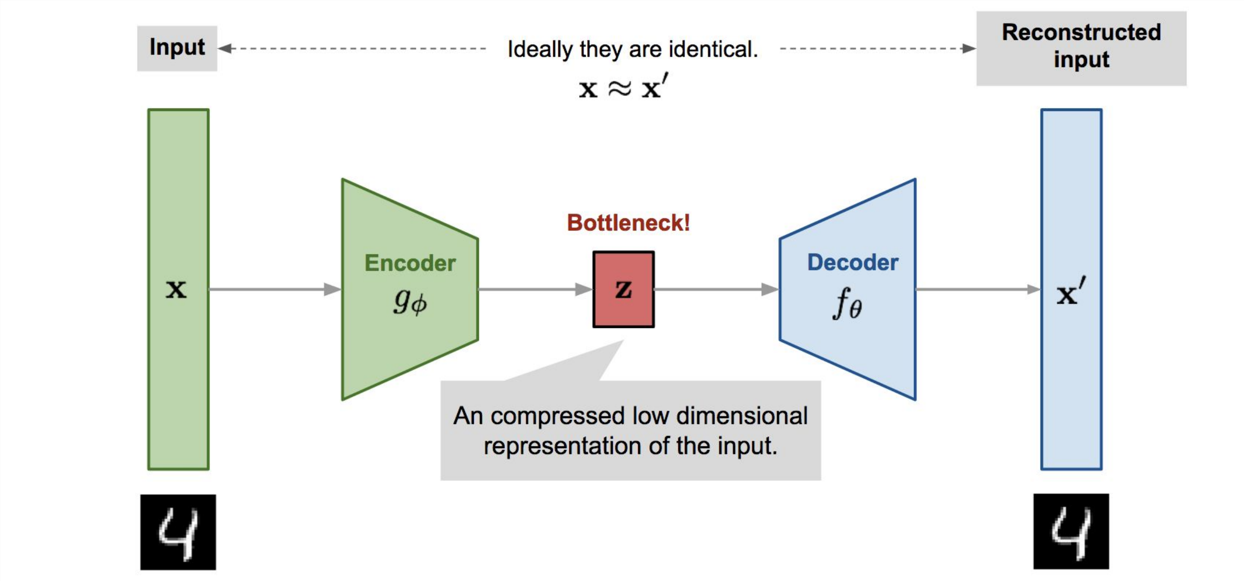

$$\mathcal{L}(x, \hat{x}) = ||x - \hat{x}||^2$$

Como o backpropagation vai atuar aqui?

Ele vai minimizar o custo com relação a que parâmetros?

Onde estão os parâmetros na perda acima? Eles estão escondidos no x-hat...

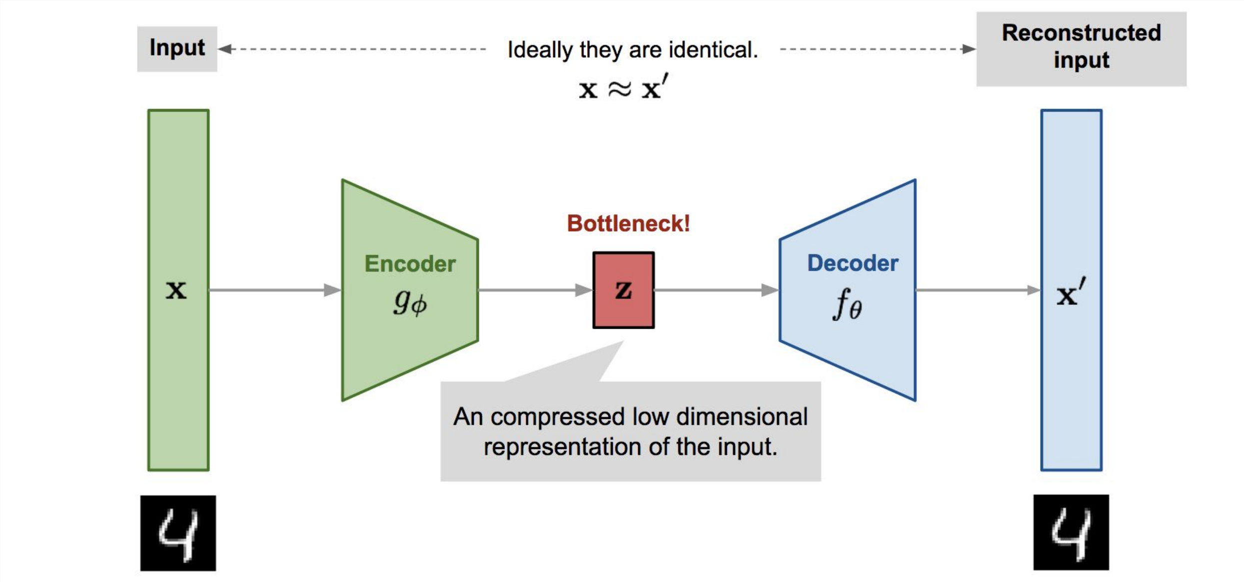
# Perda: primeiro termo




x-hat depende dos parâmetros da rede DECODER e dos fatores latentes z:

$$\mathcal{L}(x, \hat{x}) = ||x - \hat{x}||^2 = ||x - f_{\theta}(z)||^2$$

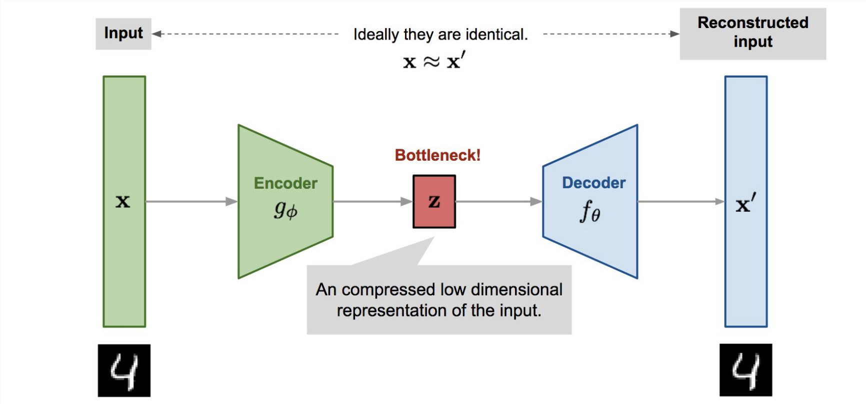
# Perda: segundo termo




Por sua vez, z é função de x E dos parâmetros da REDE ENCODER

$$\mathcal{L}(x, \hat{x}) = ||x - \hat{x}||^2 = ||x - f_\theta(z)||^2 = ||x - f_\theta(g_\phi(x))||^2$$

# Perda: primeiro termo




$$\mathcal{L}(x, \hat{x}) = ||x - \hat{x}||^2 = ||x - f_\theta(z)||^2 = ||x - f_\theta(g_\phi(x))||^2$$

Vamos usar uma amostra de treinamento (vários vetores x) para aprender os pesos $\phi$ e $\theta$

Vamos usar backpropagation (gradient descent) para achar pesos que minimizem o custo esperado (custo médio)

# Loss function: adicionando um segundo termo

- Loss function = L1 + L2
    - L1 = distância entre x e x̂  (mede a qualidade da reconstrução)
    - L2 = regularização para evitar overfitting ao dados de treino
- Acabamos de ver L1:

$$L_1 = \mathcal{L}(x, \hat{x}) = ||x - \hat{x}||^2 = ||x - f_\theta(\boldsymbol{z})||^2 = ||x - f_\theta(g_\phi(\boldsymbol{x}))||^2$$

- Para L2, existem algumas opções.
- A mais popular é penalizar fatores latentes muito grandes, forçando uma contração para zero:

$$L_2 = \lambda \sum_j |z_j| = \lambda ||\boldsymbol{z}||_1$$

- Vamos para obter uma representação esparsa dos dados.
- O enconder seleciona apenas alguns nós no CODE para representar a entrada.
- Esta penalidade é qualitativamente diferente das penalidades L2 ou L1 usadas para regularizar os pesos das redes neurais.
- Em autoencoders, nós forçamos os valores z da camada CODE a irem para zero em vez dos pesos.

# Loss function: adicionando um segundo termo

- Loss function = L1 + L2
  - L1 = distância entre x e x̂  (mede a qualidade da reconstrução)
  - L2 = regularização para evitar overfitting ao dados de treino

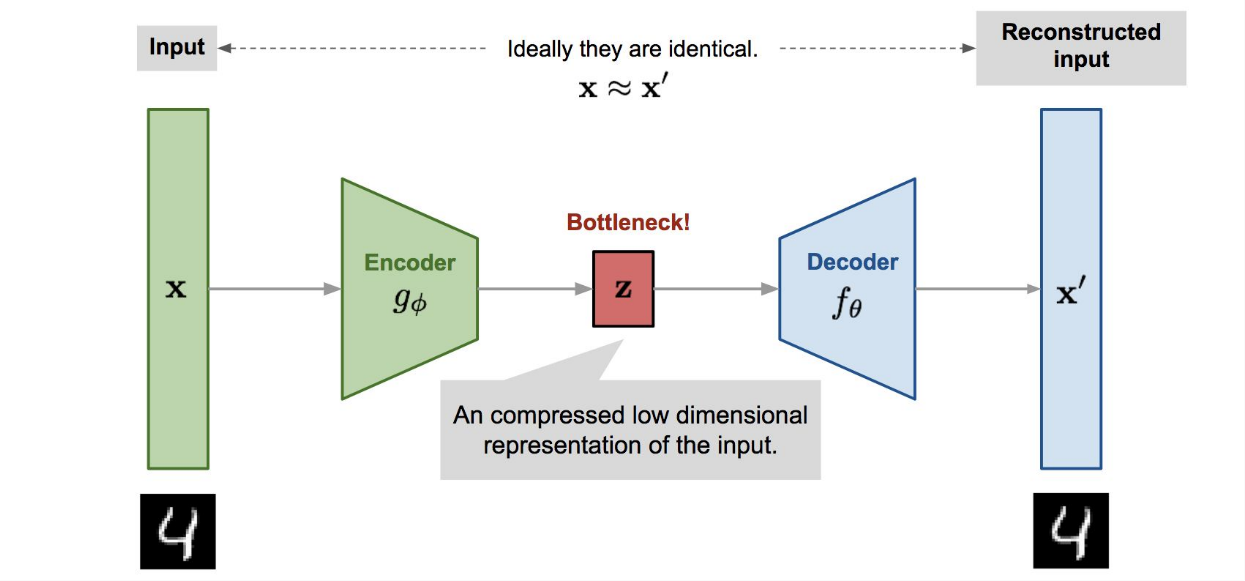
$$L_2 = \lambda \sum_j |z_j| = \lambda ||\boldsymbol{z}||_1$$

Isto é:

$$L_2 = \lambda ||g_\phi(\boldsymbol{x})||_1$$

- Procuramos o vetor de pesos  phi que faça L2 pequeno
- Mas e o parâmetro theta do decoder? Não mexemos nele?
- Se a rede DECODER é simétrica à ENCODER então os pesos de encoder e decoder estarão amarrados.
- Ao otimizar phi estaremos também otimizando theta

# Loss function




$$L = L_1 + L_2 = ||x - f_\theta(g_\phi(\boldsymbol{x}))||^2 + \lambda||g_\phi(\boldsymbol{x})||_1$$

# Para construir um autoencoder

- Precisamos de três coisas:
    - uma função de codificação (ou seja: especificar a arquitetura de uma rede neural),
    - uma função de decodificação (outra rede neural, geralmente simétrica à anterior)
    - uma função de perda a ser minimizada. Existem outra opções além das que mencionamos aqui. Ver algumas nos exemplos.

Rode o backpropagation para obter os pesos das duas redes (ENCODER e DECODER) que minimizam a função de perda.

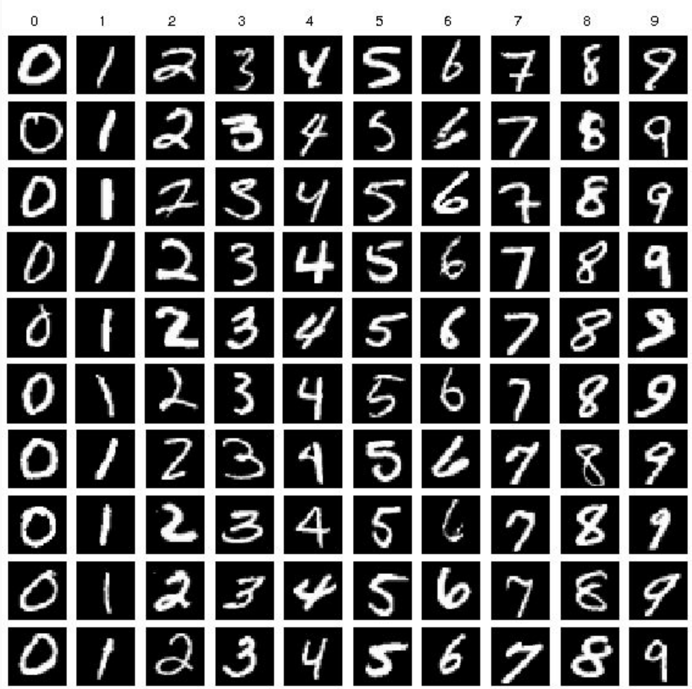
- That’s it!
- Vamos ver um exemplo (você vai reproduzir este exemplo na sessão prática)

# Exemplos com MNIST

https://blog.keras.io/building-autoencoders-in-keras.html

# MNIST

- Imagens de dígitos manuscritos
  - 28 * 28 = 784 pixels
- Não queremos fazer classificação



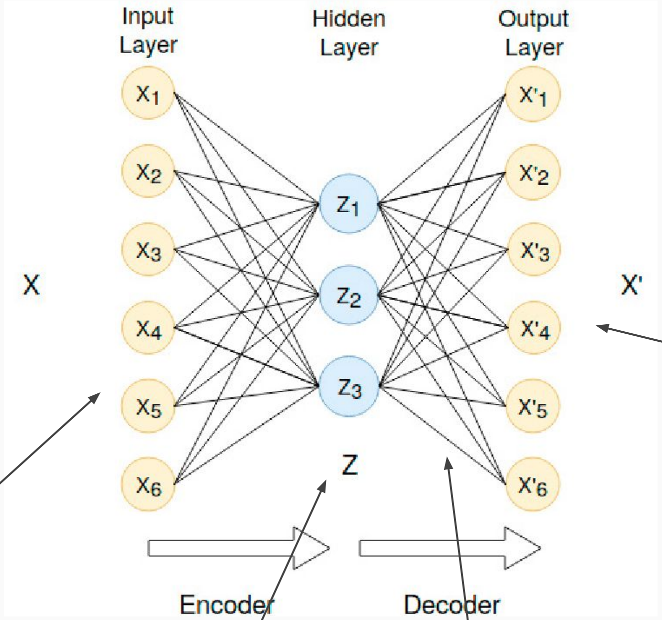
# 1º autoencoder

- Uma <u>única </u>camada escondida com 32 unidades
- Rede rasa (shallow)
- Multi-layer, fully connected

- Ativação RELU (semi-linear, acaba sendo similar a um PCA simples)

- sem regularização na função de custo

# Arquitetura do 1o exemplo




Saída = vetor x-hat 784-dim

Input x = vetor 784-dim

z = vetor 32-dim

Ativação = RELU

# Código em KERAS (+ TensorFlow)

https://blog.keras.io/building-autoencoders-in-keras.html

```
import keras
from keras import layers

# This is the size of our encoded representations
encoding_dim = 32  # 32 floats -> compression of factor 24.5, assuming the input is 784 floats

# This is our input image
input_img = keras.Input(shape=(784,))
# "encoded" is the encoded representation of the input
encoded = layers.Dense(encoding_dim, activation='relu')(input_img)
# "decoded" is the lossy reconstruction of the input
decoded = layers.Dense(784, activation='sigmoid')(encoded)

# This model maps an input to its reconstruction
autoencoder = keras.Model(input_img, decoded)
```

Etc… Veja os detalhes do código na aula prática

# Resultados

- A linha superior tem os dígitos originais.
- A linha inferior são os dígitos reconstruídos.
- Algumas imagens são reconstruídas mais pobremente que outras

- Mas, em geral, estamos perdendo apenas um pouco de detalhes com essa abordagem básica.

# Adicionando uma restrição de esparsidade

- Vamos tornar a representação esparsa acrescentando uma penalização

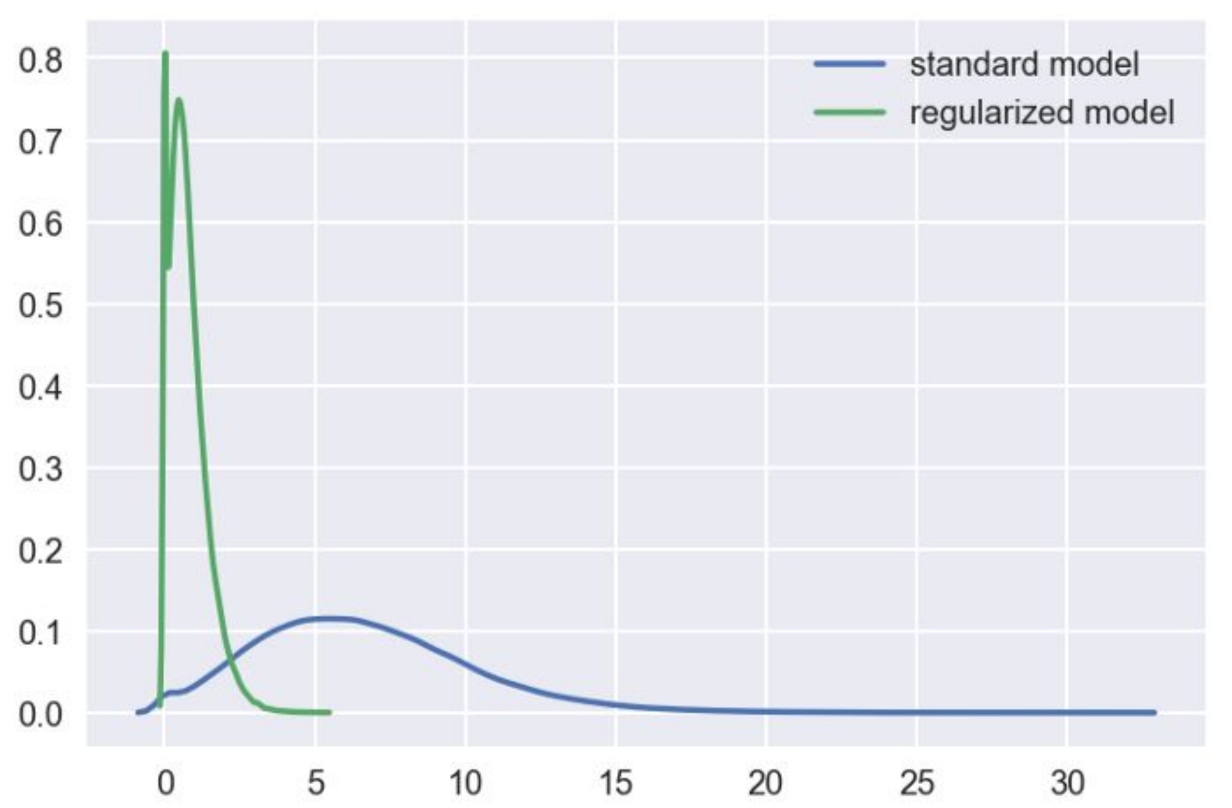
- Continua a mesma arquitetura.

- Em Keras, isso pode ser feito adicionando um activity_regularizer à nossa camada Dense:

# Resultado

Muito similar ao anterior

# Comparação entre os pesos: com e sem regularização



# Deep autoencoder: mais camadas

- Mais camadas ocultas

```
input_img = Input(shape=(784,))
encoded = Dense(128, activation='relu')(input_img)
encoded = Dense(64, activation='relu')(encoded)
encoded = Dense(32, activation='relu')(encoded)

decoded = Dense(64, activation='relu')(encoded)
decoded = Dense(128, activation='relu')(decoded)
decoded = Dense(784, activation='sigmoid')(decoded)
```

Let's try this:

```
autoencoder = Model(input_img, decoded)
autoencoder.compile(optimizer='adadelta', loss='binary_crossentropy')

autoencoder.fit(x_train, x_train,
                epochs=100,
                batch_size=256,
                shuffle=True,
                validation_data=(x_test, x_test))
```

# Resultado

- Um pouquinho melhor em termos da função custo avaliada no conjunto de teste
- Reconstrução parece apenas um pouco melhor

# Autoencoders convolucionais

- Como nossas entradas são imagens, faz sentido usar redes neurais convolucionais (CNN) como codificadores e decodificadores.

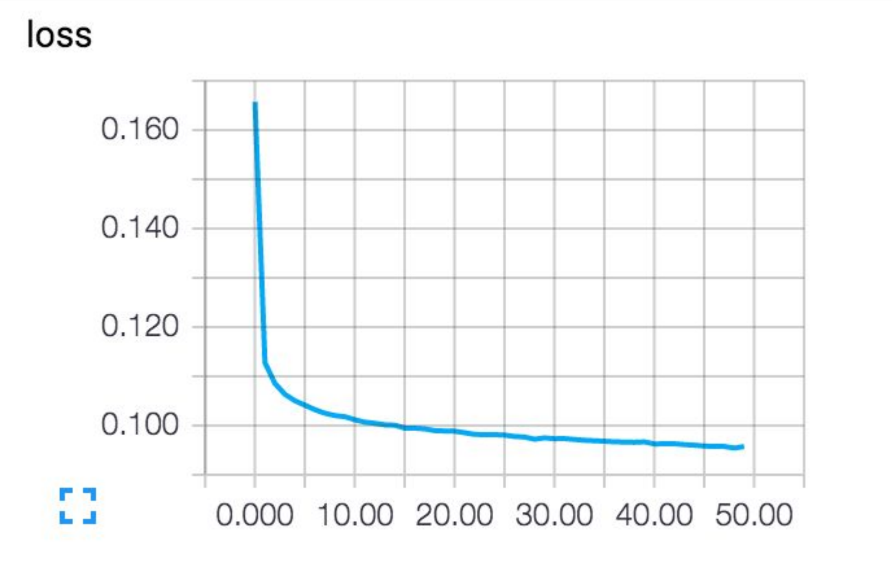
- Autoencoders aplicados às imagens são sempre convolucionais
  - eles funcionam melhor.

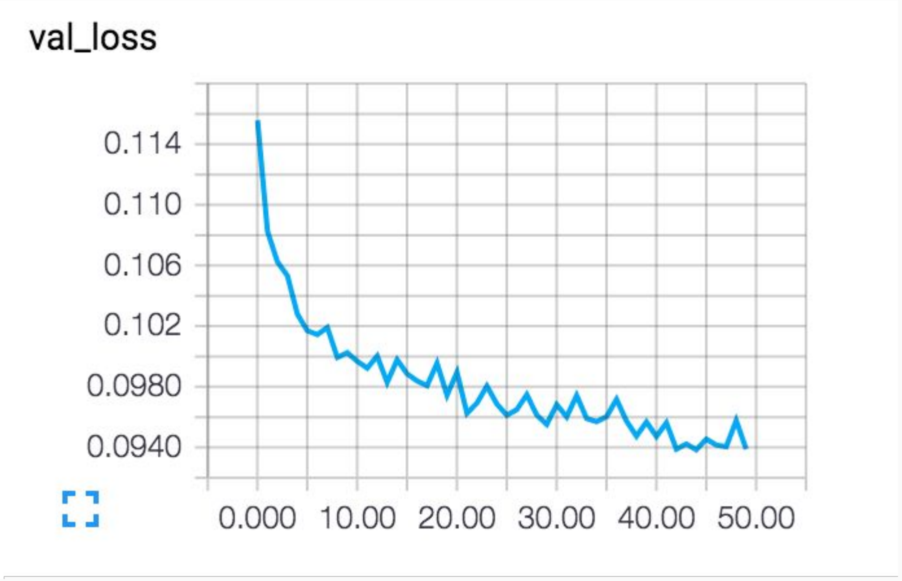
- Vamos implementar um.
- Encoder: uma pilha de camadas Conv2D e MaxPooling2D (maxpool para amostragem espacial)
- Decoder: uma pilha de camadas Conv2D e UpSampling2D
- Loss: sem penalização

# Keras code

```
from keras.layers import Input, Dense, Conv2D, MaxPooling2D, UpSampling2D
from keras.models import Model
from keras import backend as K

input_img = Input(shape=(28, 28, 1))  # adapt this if using `channels_first` image data format

x = Conv2D(16, (3, 3), activation='relu', padding='same')(input_img)
x = MaxPooling2D((2, 2), padding='same')(x)
x = Conv2D(8, (3, 3), activation='relu', padding='same')(x)
x = MaxPooling2D((2, 2), padding='same')(x)
x = Conv2D(8, (3, 3), activation='relu', padding='same')(x)
encoded = MaxPooling2D((2, 2), padding='same')(x)

# at this point the representation is (4, 4, 8) i.e. 128-dimensional

x = Conv2D(8, (3, 3), activation='relu', padding='same')(encoded)
x = UpSampling2D((2, 2))(x)
x = Conv2D(8, (3, 3), activation='relu', padding='same')(x)
x = UpSampling2D((2, 2))(x)
x = Conv2D(16, (3, 3), activation='relu')(x)
x = UpSampling2D((2, 2))(x)
decoded = Conv2D(1, (3, 3), activation='sigmoid', padding='same')(x)

autoencoder = Model(input_img, decoded)
autoencoder.compile(optimizer='adadelta', loss='binary_crossentropy')
```

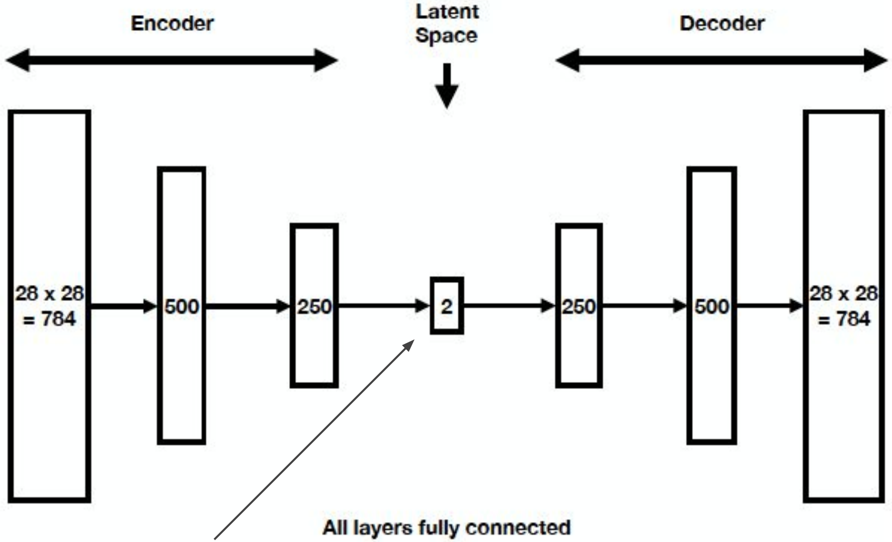
# Perda = 0.094, melhor que anteriores (128 dimensões vs. 32 anteriormente).



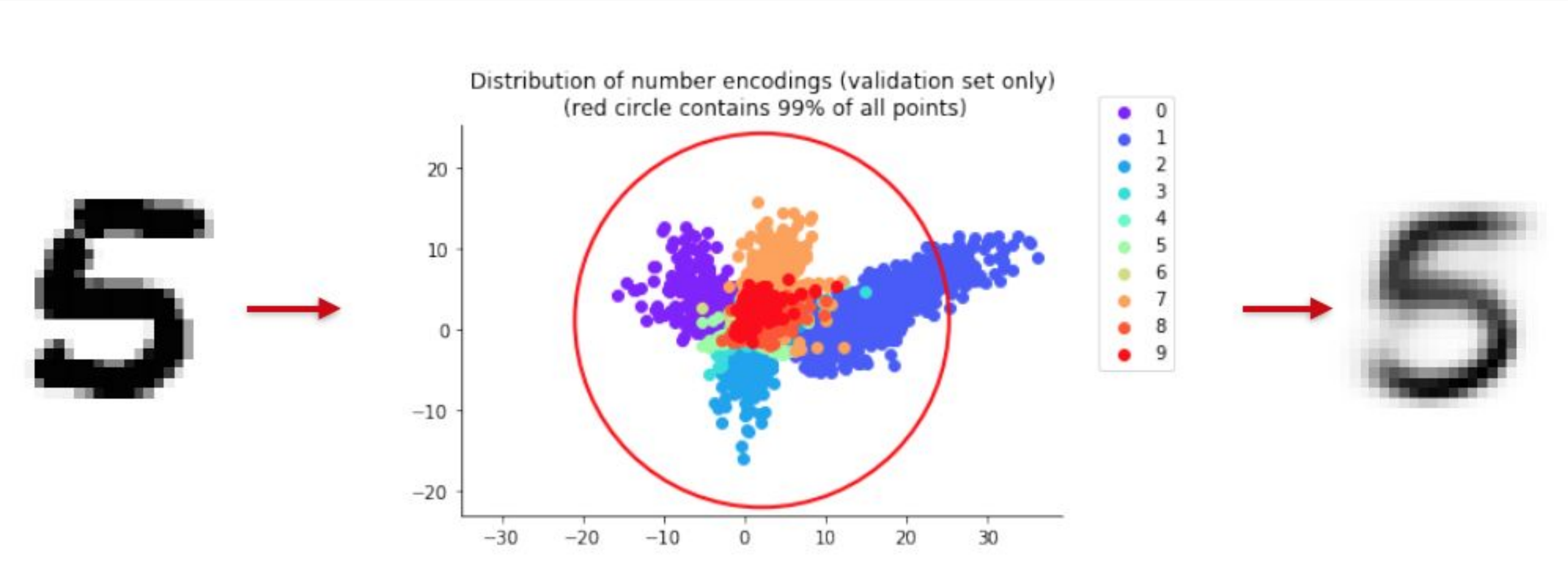

## AE with MNIST

- MNIST is a famous dataset of 70,000 28 x 28 images of handwritten digits.
- Train an AE to compress digits to a 2-D latent space.



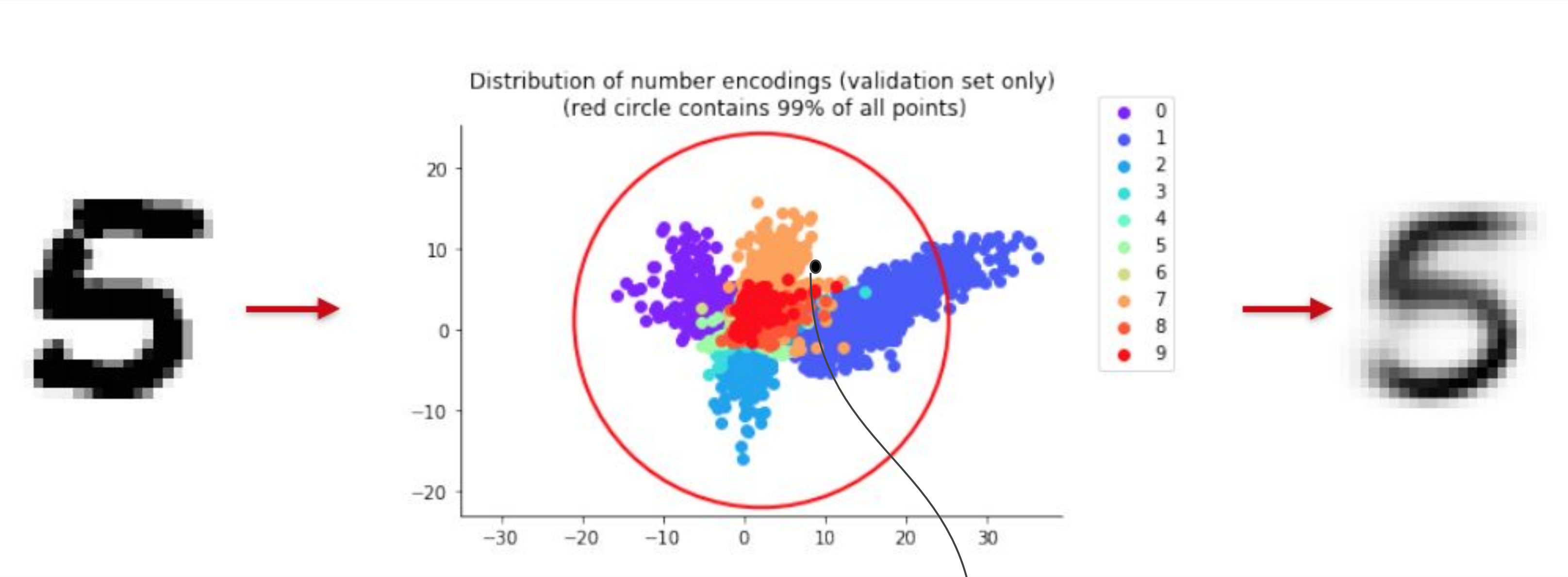

CODE z = (z_1, z_2) , apenas um vetor <u>BI-dimensional</u>

# A representação latente: com os dados de validação apenas



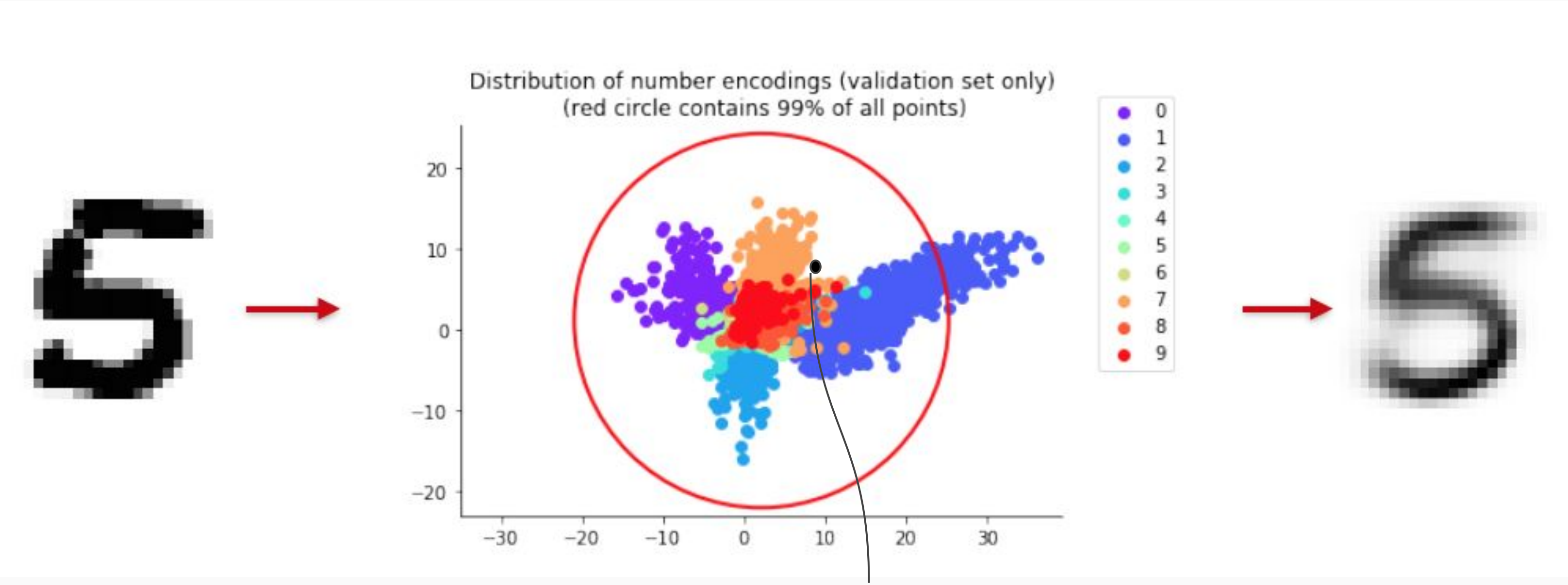

Cada ponto corresponde a uma imagem.
O ponto é o vetor latente z=(z_1, z_2) da imagem.
A cor indica a classe da imagem.
Veja como cada classe forma um cluster no espaço latente
EMBORA a classe <u>não tenha sido</u> <u>usada</u> no autoencoder.

# Podemos gerar novos exemplos??




Selecione um ponto z = (z_1, z_2) ARBITRARIAMENTE
Não é um dos pontos existentes...Por exemplo, selecione ESTE ponto preto

# Podemos gerar novos exemplos??




Por exemplo, selecione ESTE ponto preto

Passe este z=(z_1, z_2) pelo DECODER produzindo uma saída.
O que vai aparecer na saída?
Um 7 aproximado?
Faz sentido CAMINHAR neste espaço Z bi-dim continuamente?
Voltaremos a este tópico com Variational autoencoder (VAE), nosso próximo tópico...

# Denoising autoencoders

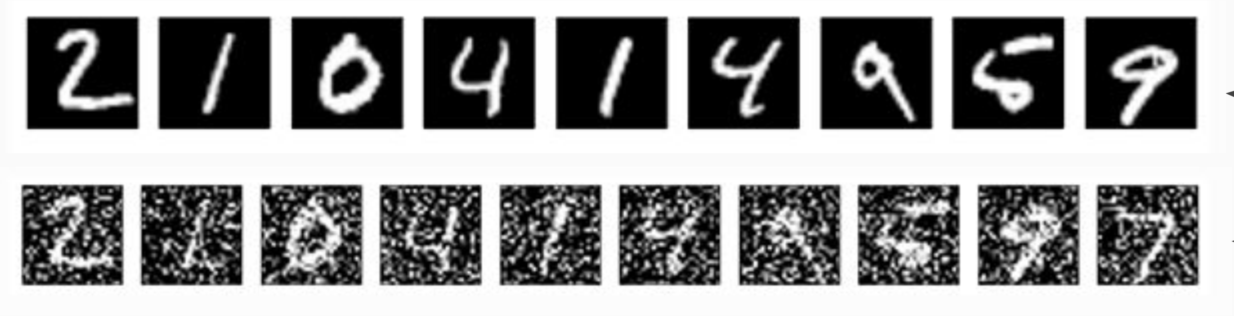
- Outra técnica de regularização.

- Não requer a função L2 na função de perda.

- A ideia é evitar que uma rede muito complexa possa aprender a função de identidade. Se CODE (bootleneck) tiver dimensão muito alta, ele pode apenas aprender os dados de entrada, de modo que a saída seja igual à entrada.
- Não faz um aprendizado de representação que seja útil nem redução de dimensionalidade.
- Denoising autoencoders corrompem os dados de entrada propositalmente.
- Eles adicionam ruído ou mascaram alguns dos valores de entrada.

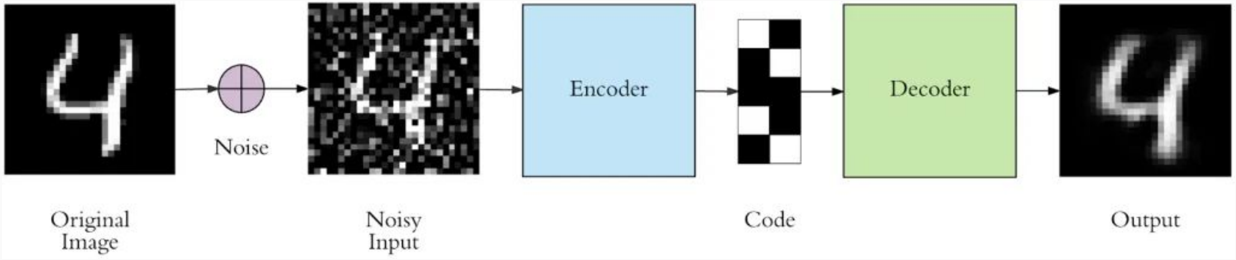
# Denoising autoencoders

- Deseja-se um autoencoder que seja robusto a ruído: que consiga eliminar ruído de imagens
- Geramos ruído e somamos ao input x formando $\tilde{\mathbf{x}} = \mathbf{x} + \varepsilon$



x = dado original

x-til (dado corrompido)



# Denoising autoencoders

- Geramos ruído e somamos ao input x formando $\tilde{\mathbf{x}} = \mathbf{x} + \varepsilon$
- A função de perda agora é:

$$\mathcal{L}(x, \hat{x}) = ||x - \hat{x}||^2 = ||x - f_{\theta}(z)||^2 = ||x - f_{\theta}(g_{\phi}(\tilde{x}))||^2$$

- Ele deve aprender a recuperar x a partir de x-til
- Fornece como entrada imagem com ruído: x-til
- Objetivo é reconstruir imagem sem o ruído: x

dado
original

Input é o dado
corrompido !!

# Como antes, com entrada e saída alteradas

```
input_img = Input(shape=(28, 28, 1))  # adapt this if using `channels_first` image data format

x = Conv2D(32, (3, 3), activation='relu', padding='same')(input_img)
x = MaxPooling2D((2, 2), padding='same')(x)
x = Conv2D(32, (3, 3), activation='relu', padding='same')(x)
encoded = MaxPooling2D((2, 2), padding='same')(x)

# at this point the representation is (7, 7, 32)

x = Conv2D(32, (3, 3), activation='relu', padding='same')(encoded)
x = UpSampling2D((2, 2))(x)
x = Conv2D(32, (3, 3), activation='relu', padding='same')(x)
x = UpSampling2D((2, 2))(x)
decoded = Conv2D(1, (3, 3), activation='sigmoid', padding='same')(x)

autoencoder = Model(input_img, decoded)
autoencoder.compile(optimizer='adadelta', loss='binary_crossentropy')
```

# Resultado

# Uso prático

Ao invés de somar um ruído completamente aleatório, somamos um ruído "estruturado"

Imagem observada é a imagem desejada + ruído "estruturado"

Queremos eliminar o ruído "estruturado"

# Retirando watermarks com denoising autoencoders

- Projeto em https://github.com/tallosan/DeWatermarker


# Retirando watermarks com denoising autoencoders

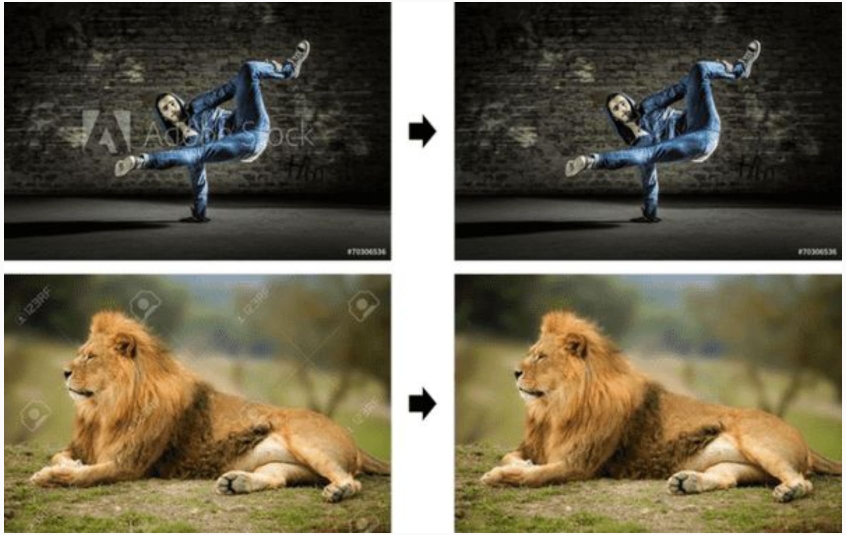
- Pegue uma base de fotos SEM watermark: x
- Adicione a watermark como elas serão encontradas no futuro: x-til



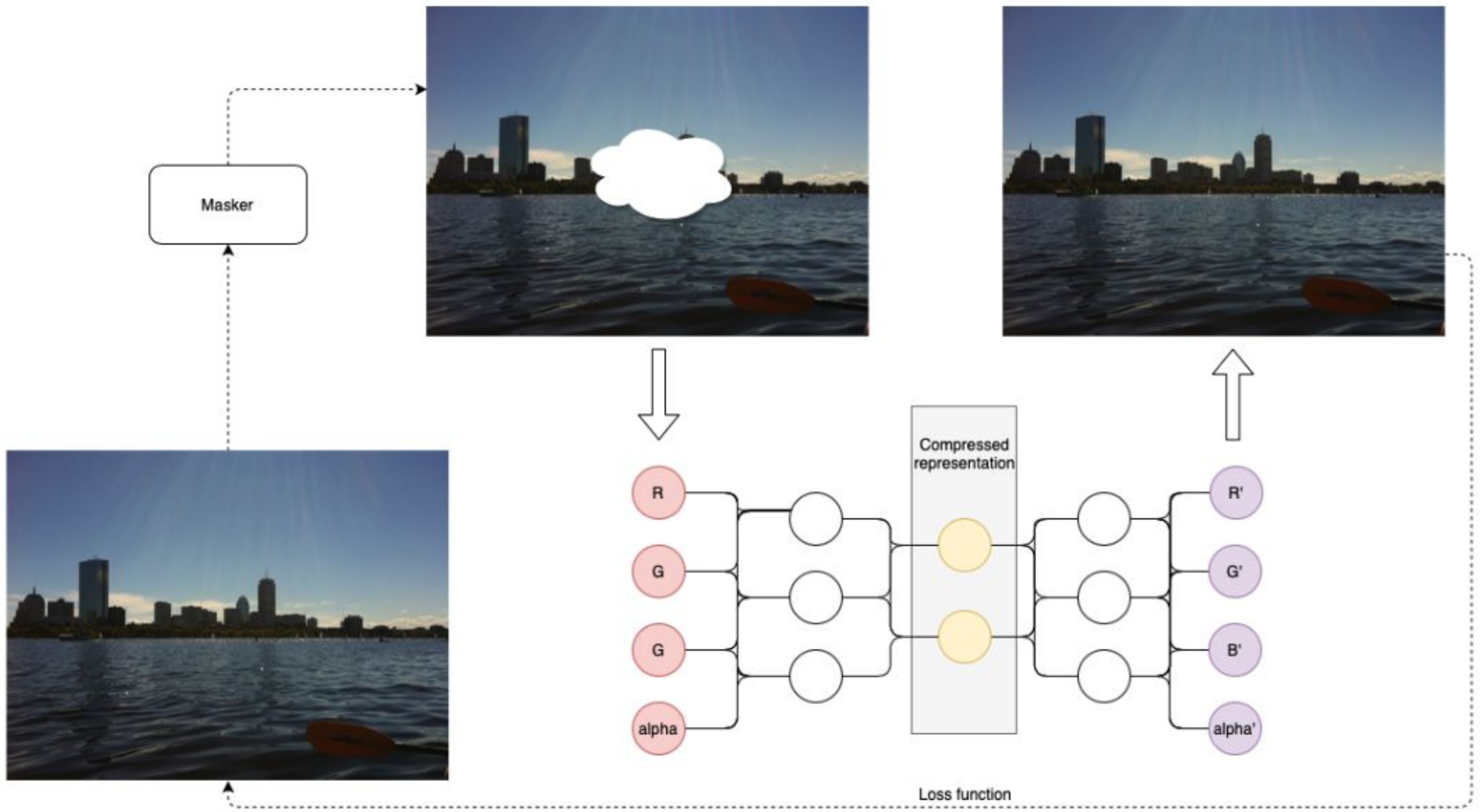
- Temos uma base de imagens em pares ( x , x-til )
- Treine autoencoders para recuperar x a partir de input x-til

# Projetos: retirando watermarks com denoising autoencoders

- Outros projetos:
- https://github.com/tallosan/DeWatermarker
- https://github.com/marcbelmont/cnn-watermark-removal
- https://thenextweb.com/google/2017/08/18/google-watermark-stock-photo-remove/
- https://github.com/udacity/deep-learning/tree/master/autoencoder?source=post_page---------------------------



# Inpainting

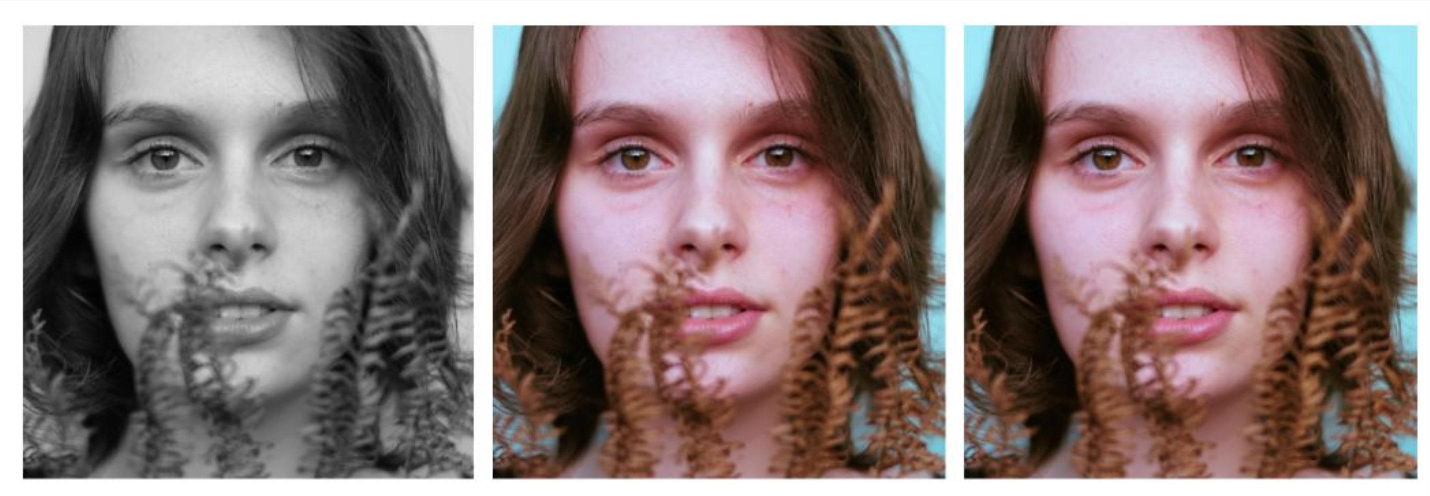


Masker
Compressed representation
R
G
G
alpha
R'
G'
B'
alpha'
Loss function

# Uma ideia muito similar: colorindo imagens preto e branco

Projeto: https://blog.floydhub.com/colorizing-b-w-photos-with-neural-networks/

- Pegue imagens coloridas e transforme em preto-e-banco: pares de (x, x-til)
- Forneça x-til (as imagens p-b) como inputs e compare com a recuperação x-hat da imagem colorida x em 3 canais



P-B Original Recuperada

# Detecção de anomalias

- Autoencoder treinado em um conjunto específico de imagens: apenas gatos
- conjunto D de imagens (ou de treinamento).
- Se x $\in$ D então então o erro (ou perda) de reconstrução $|x - \hat{x}|^2$ deve ser pequeno
- Olhando no conjunto de teste, descobrimos que este erro quase nunca ultrapassa um limiar L
- Mas se $x \notin D$ e é de um tipo muito diferente (paisagem de montanhas) o seu erro de reconstrução deve ser grande
- O autoencoder não pode realizar a reconstrução, pois os atributos latentes não são adaptados para a imagem que nunca foi vista pela rede e que é bem diferente.
- A perda de reconstrução deve ser alta (maior que L)
- Imagem pode ser identificada como uma anomalia

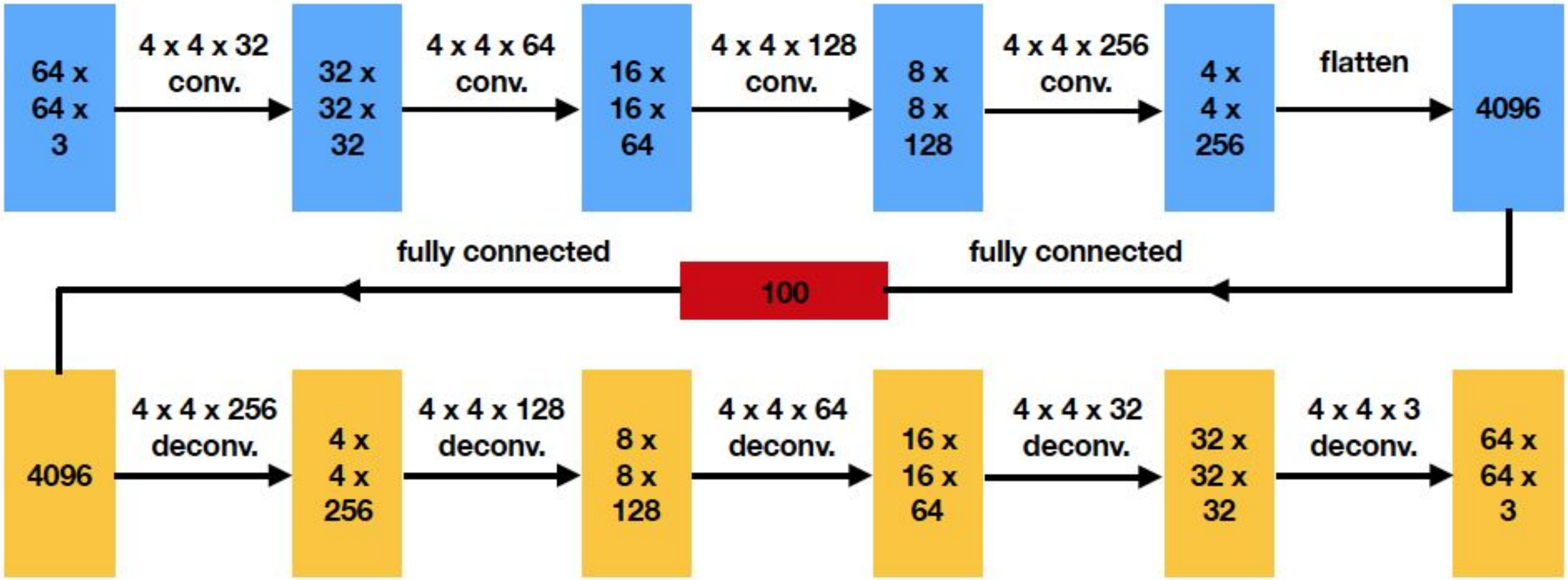
# Autoencoders são bons em compactação de dados?

- Na verdade, não são muito bons.
- Na compactação de imagens, por exemplo, é muito difícil treinar um autoencoder que faça um trabalho melhor do que um algoritmo básico como o JPEG.
- Normalmente, a única maneira de obter isso é restringindo-se a um tipo muito específico de imagem (por exemplo, imagem nas quais JPEG não faz um bom trabalho).
- O fato de os autoencoders serem data-specific os torna geralmente impraticáveis para problemas reais de compactação de dados:
  - você só pode usá-los em dados semelhantes aos que foram usados no treino.
  - Torná-los mais gerais exige dados de treinamento muito distintos, de vários tipos de objetos, situações, contextos, etc.
- Avanços futuros podem mudar isso... quem sabe...

From https://blog.keras.io/building-autoencoders-in-keras.html

# AE with celebA

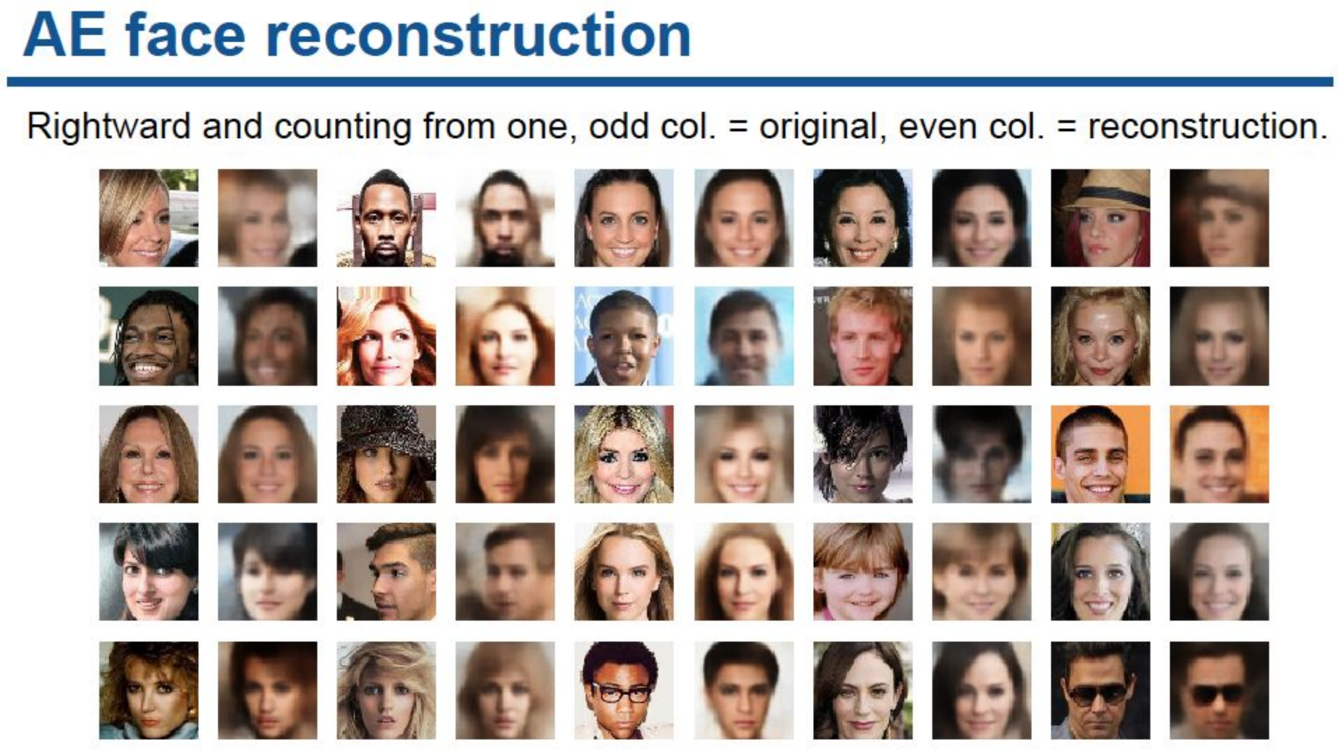
- CelebA is a dataset of over 200,000 images of celebrity faces.
- Train an AE to compress face images to a 100-D latent space.



# Resumo

Criado por Steven Flores

Resultado não é nenhuma maravilha para os dias de hoje



# Resumo

- Na prática, autoencoders não levam a espaços latentes particularmente úteis ou bem estruturados.
    - Ver aula de Variational autoencoder onde isto é discutido com mais detalhes
- Também não são muito bons em redução de dimensionalidade.
- Por estas razões, eles caíram em desuso para reduzir dimensionalidade.
- Entretanto ainda são usados para denoising images.

- Variational encoders (VAE) são uma alternativa mais popular para redução de dim.
- VAE introduzem probabilidade para aprender espaços latentes contínuos e mais estruturados.
- Eles se revelaram uma ferramenta útil para geração de imagens.

FIM da 1a aula de DL unsupervised