[PSI3472 aula 11, Início]

Autoencoders

1. Introdução

Vamos seguir:

https://blog.keras.io/building-autoencoders-in-keras.html
https://www.deeplearningbook.com.br/introducao-aos-autoencoders/
https://www.deeplearningbook.com.br/principais-tipos-de-redes-neurais-artificiais-autoencoders/

Autoencoder não é especialmente útil na prática. Porém, costuma ser apresentado em muitos cursos de aprendizado de máquina como uma alternativa melhor para as técnicas clássicas de redução de dimensionalidade, como PCA. A figura abaixo ilustra bem o funcionamento do PCA, reduzindo os dados de dimensão 3 para 2. Porém, para muitos problemas, provavelmente é possível efetuar transformações mais complexas para reduzir a dimensão dos dados.

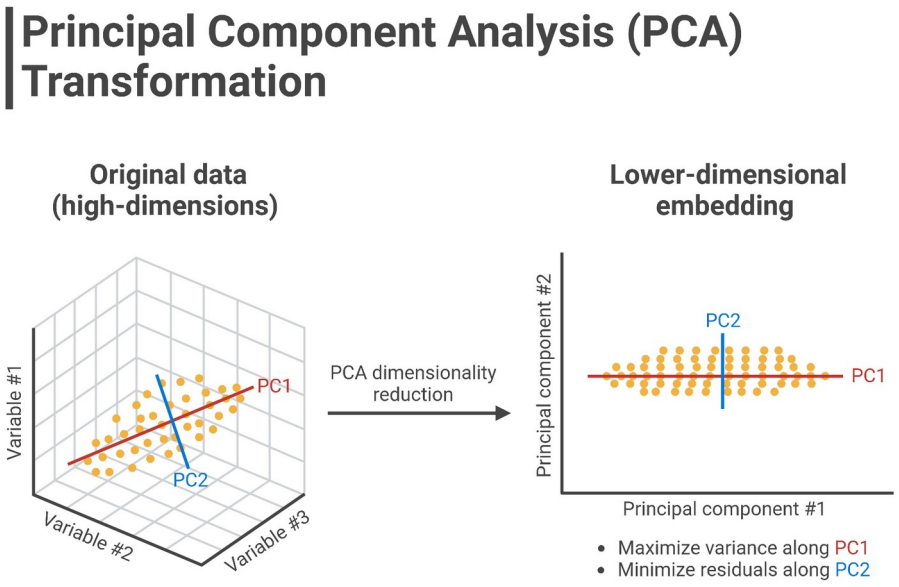


Figura: Funcionamento de PCA [https://www.biorender.com/template/principal-component-analysis-pca-transformation].

Um "autoencoder" ou autocodificador é uma rede neural usada para comprimir e aprender a representação de dados. Ele é composta por duas partes principais:

1. *Codificador (Encoder):* A primeira parte da rede reduz a dimensionalidade dos dados de entrada x, comprimindo-os em uma representação compacta chamada de "vetor latente" z. Essa parte da rede busca identificar as características mais importantes do dado, eliminando redundâncias.
2. *Decodificador (Decoder):* A segunda parte da rede tenta reconstruir os dados originais $\hat{x}$ a partir da representação compacta z. O objetivo é que a saída do decodificador seja o mais próximo possível da entrada original x.

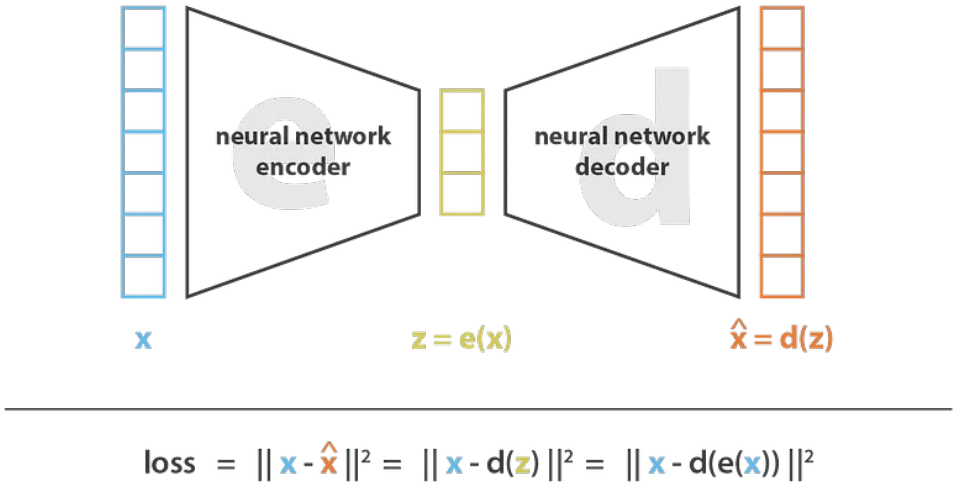


Figura 1: Um autoencoder [https://towardsdatascience.com/understanding-variational-autoencoders-vaes-f70510919f73]

Como Funciona:

- O autoencoder recebe um dado de entrada *x*, como uma imagem ou uma sequência de números.
- O "codificador" *e* transforma esse dado em um vetor latente de dimensão menor $z=e(x)$, comprimindo as informações.
- O "decodificador" *d* usa o vetor comprimido *z* para tentar reconstruir o dado original $\hat{x}=d(z)$.
- O treinamento da rede consiste em minimizar a diferença entre o dado original *x* e o reconstruído $\hat{x}$, usando alguma função de erro, por exemplo, o erro quadrático médio (MSE) ou *binary_crossentropy* (para problemas onde a saída ideal se aproxima da decisão binária).

A compressão de dados pelo autoencoder tem as seguintes características:

1. *Específica aos dados:* Os autoencoders são específicos aos dados, o que significa que eles só serão capazes de comprimir dados semelhantes aos quais foram treinados. Um autoencoder treinado em imagens de rostos não iria funcionar bem para comprimir imagens de árvores.
2. *Com perdas:* Os autoencoders comprimem dado com perdas, isto é, as saídas descomprimidas estarão degradadas em comparação com as entradas (semelhante à compressão MP3 ou JPEG).
3. *Aprendidas automaticamentes:* Os autoencoders são treinados automaticamente a partir de exemplos de treino, o que é uma propriedade útil pois é fácil treinar instâncias especializadas para um tipo específico de dado.

Nota: Os autoencoders normalmente não são bons se forem usado para comprimir dados. Para comprimir imagens, por exemplo, é muito difícil construir um autoencoder que funcione melhor do que um algoritmo básico como JPEG.

Principais Aplicações:

- *Redução de dimensionalidade:* Os autoencoders podem ser usados para reduzir a quantidade de informações sem perder as características importantes dos dados. Os autoencoders podem aprender projeções de dados melhores do que técnicas básicas como PCA.
- *Detecção de anomalias:* Como os autoencoders aprendem a reconstruir apenas dados "normais", eles tendem a ter dificuldades em reconstruir dados "anômalos". Isto pode ser usado para detectar anomalias.
- *Remoção de ruído:* Autoencoders podem ser treinados para remover ruídos de imagens ou sinais.

2. Reprodutibilidade em modelo Keras

https://keras.io/examples/keras_recipes/reproducibility_recipes/

O código abaixo faz o comportamento de Keras ser determinístico, isto é, obtém exatamente os mesmos resultados ao executar o programa várias vezes. Porém, parece que não está funcionando para todos os tipos de *layers* (não funciona em rede convolucional). Também parece que pode diminuir o desempenho de Tensorflow.

```
import tensorflow as tf
import tensorflow.keras as keras
keras.utils.set_random_seed(7)
tf.config.experimental.enable_op_determinism()
```

Programa 1: Este código torna Keras determinística.

3. Autoencoder para MNIST

Vamos tentar representar as imagens de MNIST com 28×28=784 pixels usando um vetor de atributos com dimensão de apenas 32.

Para isso, vamos construir um autoencoder muito simples, que consiste de apenas 2 camadas densas e que codifica as imagens de MNIST, seguindo:

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

A figura 2 mostra o modelo e o programa 2 é a implementação.

As linhas 16-18 do programa 2 descrevem o modelo (figura 2) que consiste em:

- *Encoder* que reduz a imagem 28×28, convertida em vetor de dimensão 784, para vetor latente de dimensão 32.
- *Decoder* que aumenta a dimensão do vetor latente de dimensão 32 para 784.

A figura 3 mostra a saída do programa. Foi obtida erro de teste 0.0916 (binary crossentropy). A média do vetor latente da primeira imagem de teste é 6.5926.

A figura 4 mostra as imagens de teste de entradas e as imagens reconstruídas pelo autoencoder. As saídas estão semelhantes às imagens originais.

Nota: Alterei a linha 29 (vermelho) em relação ao programa original. Para modelos de rede mais complexos, o programa não funciona se deixar esta linha como estava no programa original.

```python
#~/deep/algpi/autoencoder/ae1.py
#https://blog.keras.io/building-autoencoders-in-keras.html
import tensorflow as tf
import tensorflow.keras as keras
from tensorflow.keras import layers
from tensorflow.keras.datasets import mnist
import numpy as np
import matplotlib.pyplot as plt

keras.utils.set_random_seed(7)
tf.config.experimental.enable_op_determinism()

# This is the size of our encoded representations
encoding_dim = 32

input_img = keras.Input(shape=(784,))
encoded = layers.Dense(encoding_dim, activation='relu')(input_img)
decoded = layers.Dense(784, activation='sigmoid')(encoded)
autoencoder = keras.Model(input_img, decoded)

from tensorflow.keras.utils import plot_model
plot_model(autoencoder, to_file='autoencoder.png', show_shapes=True)
autoencoder.summary()

encoder = keras.Model(input_img, encoded)
plot_model(encoder, to_file='encoder.png', show_shapes=True)
encoder.summary()

decoder = keras.Model(encoded, decoded)
plot_model(decoder, to_file='decoder.png', show_shapes=True)
decoder.summary()

autoencoder.compile(optimizer=keras.optimizers.Adam(learning_rate=0.001),
                    loss='binary_crossentropy')

(x_train, _), (x_test, _) = mnist.load_data()
x_train = x_train.astype('float32') / 255
x_test = x_test.astype('float32') / 255
x_train = x_train.reshape((len(x_train), np.prod(x_train.shape[1:])))
x_test = x_test.reshape((len(x_test), np.prod(x_test.shape[1:])))

autoencoder.fit(x_train, x_train, epochs=50, batch_size=256, shuffle=True,
                validation_data=(x_test, x_test))

encoded_imgs = encoder.predict(x_test)
np.set_printoptions(precision=4)
print("encoded_imgs[0]:"); print(encoded_imgs[0])
print("np.mean(encoded_imgs[0]): %.4f"%(np.mean(encoded_imgs[0])))
decoded_imgs = decoder.predict(encoded_imgs)

n = 10  # How many digits we will display
plt.figure(figsize=(20, 4))
for i in range(n):
  ax = plt.subplot(2, n, i + 1)
  plt.imshow(x_test[i].reshape(28, 28), vmin=0, vmax=1)
  plt.gray()
  ax.get_xaxis().set_visible(False)
  ax.get_yaxis().set_visible(False)

  ax = plt.subplot(2, n, i + 1 + n)
  plt.imshow(decoded_imgs[i].reshape(28, 28), vmin=0, vmax=1)
  plt.gray()
  ax.get_xaxis().set_visible(False)
  ax.get_yaxis().set_visible(False)
plt.savefig("ae1_saida.png")
plt.show()
```

Programa 2: Autoencoder com apenas duas camadas densas.

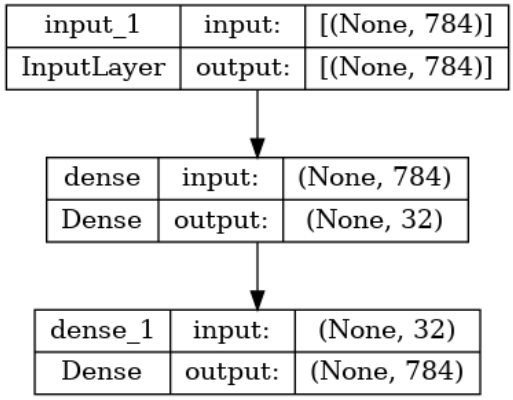
```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 784)]             0

 dense (Dense)               (None, 32)                25120

 dense_1 (Dense)             (None, 784)               25872

=================================================================
Total params: 50992 (199.19 KB)
Trainable params: 50992 (199.19 KB)
Non-trainable params: 0 (0.00 Byte)
```

(a) Autoencoder



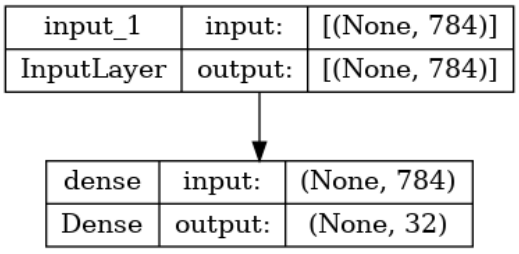
(b) Autoencoder

```
Model: "model_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 784)]             0

 dense (Dense)               (None, 32)                25120

=================================================================
Total params: 25120 (98.12 KB)
Trainable params: 25120 (98.12 KB)
Non-trainable params: 0 (0.00 Byte)
```

(c) Parte encoder



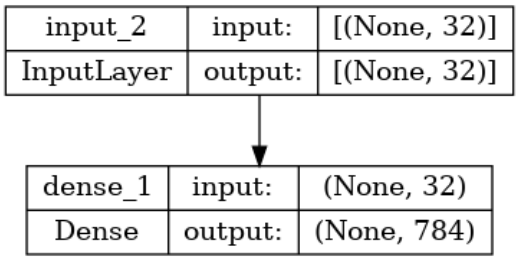
(d) Parte encoder

```
Model: "model_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_2 (InputLayer)        [(None, 32)]              0

 dense_1 (Dense)             (None, 784)               25872

=================================================================
Total params: 25872 (101.06 KB)
Trainable params: 25872 (101.06 KB)
Non-trainable params: 0 (0.00 Byte)
```

(e) Parte decoder



(f) Parte decoder

Figura 2: O modelo autoencoder (a-b) do programa 2 consiste de duas partes: encoder (c-d) e decoder (e-f).

Saída do programa:

```
Epoch  1/50 - 2s 4ms/step - loss: 0.2750 - val_loss: 0.1892
Epoch 10/50 - 1s 3ms/step - loss: 0.0975 - val_loss: 0.0955
Epoch 20/50 - 1s 3ms/step - loss: 0.0934 - val_loss: 0.0921
Epoch 30/50 - 1s 3ms/step - loss: 0.0930 - val_loss: 0.0917
Epoch 40/50 - 1s 3ms/step - loss: 0.0928 - val_loss: 0.0917
Epoch 50/50 - 1s 3ms/step - loss: 0.0926 - val_loss: 0.0916
```

Os 32 atributos e a média da primeira imagem de teste (o dígito “7” da figura abaixo).

```
encoded_imgs[0]:
[ 6.8632 12.2469  8.2113  7.629   4.9538  6.9681  4.7306  6.9359 10.5455
  2.3526  5.4275  2.1784  7.0271  5.8501  4.3305  6.8972  1.3432 13.3574
 11.0648  8.2272  5.1412  5.1101  9.5419 11.6732  2.3109  6.1704  6.6079
  1.277  13.5271  4.4521  2.6978  5.3125]
np.mean(encoded_imgs[0]): 6.5926
```

Figura 3: Saída do programa 2.

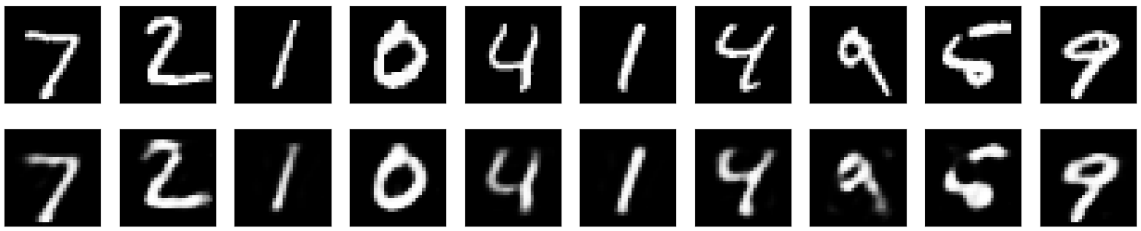


Figura 4: Linha superior: imagens originais na dimensão 28x28=784. Linha inferior: imagens reconstruídos pelo autoencoder do programa 2, usando espaço latente de dimensão 32.

4. Regularização das atividades

Para obter vetor latente esparso (isto é, com vários elementos nulos) e com magnitude pequena, podemos acrescentar regularização de atividade L_1. Isto fará o *backpropagation* preferir redes que geram vetor latente z esparso e com magnitude pequena. Para isso, é necessário fazer uma única alteração nas linhas 17-18 do programa 3 (destacado em amarelo):

```
encoded = layers.Dense(encoding_dim, activation='relu',
            activity_regularizer=regularizers.l1(10e-5)) (input_img)
```

Como vimos em aulas anteriores, regularizador de pesos e vieses calcula a norma L_1 ou L_2 dos parâmetros e a acrescenta à função de custo da rede. Isto ajuda a obter um modelo com parâmetros de pequena magnitude e a evitar *overfitting*. Regularizador de atividades calcula a norma L_1 ou L_2 das saídas de uma camada e a acrescenta à função custo da rede. Isto faz com que aquela camada gere saídas de pequena magnitude.

Nota: Tanto a regularização L_1 como L_2 fazem produzir saídas de pequena magnitude. Porém, somente a regularização L_1 faz a saída ser esparsa (ter muitos elementos nulos). A discussão abaixo mostra por que isto acontece.
https://stats.stackexchange.com/questions/45643/why-l1-norm-for-sparse-models

A figura 5 mostra a saída do programa 3. Note que apareceram 3 atributos nulos na primeira imagem de teste codificada e a média dos atributos diminuiu de 6,6 (figura 3) para 1,5 (figura 5). Por outro lado, o custo de teste aumentou de 0.0916 para 0.0978.

```python
#~/deep/algpi/autoencoder/ae2.py
#https://blog.keras.io/building-autoencoders-in-keras.html
import tensorflow as tf
import tensorflow.keras as keras
from tensorflow.keras import layers, regularizers
from tensorflow.keras.datasets import mnist
import numpy as np
import matplotlib.pyplot as plt

keras.utils.set_random_seed(7)
tf.config.experimental.enable_op_determinism()

# This is the size of our encoded representations
encoding_dim = 32

input_img = keras.Input(shape=(784,))
encoded = layers.Dense(encoding_dim, activation='relu',
            activity_regularizer=regularizers.l1(10e-5)) (input_img)
decoded = layers.Dense(784, activation='sigmoid')(encoded)
autoencoder = keras.Model(input_img, decoded)

from tensorflow.keras.utils import plot_model
plot_model(autoencoder, to_file='autoencoder2.png', show_shapes=True)
autoencoder.summary()

encoder = keras.Model(input_img, encoded)
plot_model(encoder, to_file='encoder2.png', show_shapes=True)
encoder.summary()

decoder = keras.Model(encoded, decoded)
plot_model(decoder, to_file='decoder2.png', show_shapes=True)
decoder.summary()

autoencoder.compile(optimizer=keras.optimizers.Adam(learning_rate=0.001),
                    loss='binary_crossentropy')

(x_train, _), (x_test, _) = mnist.load_data()
x_train = x_train.astype('float32') / 255
x_test = x_test.astype('float32') / 255
x_train = x_train.reshape((len(x_train), np.prod(x_train.shape[1:])))
x_test = x_test.reshape((len(x_test), np.prod(x_test.shape[1:])))

autoencoder.fit(x_train, x_train, epochs=50, batch_size=256, shuffle=True,
                validation_data=(x_test, x_test))
autoencoder.save("ae2.keras")
encoded_imgs = encoder.predict(x_test)
np.set_printoptions(precision=4)
print("encoded_imgs[0]:"); print(encoded_imgs[0])
print("np.mean(encoded_imgs[0]): %.4f"%(np.mean(encoded_imgs[0])))
decoded_imgs = decoder.predict(encoded_imgs)

n = 10  # How many digits we will display
plt.figure(figsize=(20, 4))
for i in range(n):
  ax = plt.subplot(2, n, i + 1)
  plt.imshow(x_test[i].reshape(28, 28), vmin=0, vmax=1)
  plt.gray()
  ax.get_xaxis().set_visible(False)
  ax.get_yaxis().set_visible(False)

  ax = plt.subplot(2, n, i + 1 + n)
  plt.imshow(decoded_imgs[i].reshape(28, 28), vmin=0, vmax=1)
  plt.gray()
  ax.get_xaxis().set_visible(False)
  ax.get_yaxis().set_visible(False)
plt.savefig("ae2_saida.png")
plt.show()
```

Programa 3: Autoencoder com regularização de atividade L_1.

Saída (com regularização de atividade L_1):

```
Epoch  1/50 - 2s 3ms/step - loss: 0.2828 - val_loss: 0.1967
Epoch 10/50 - 1s 3ms/step - loss: 0.1093 - val_loss: 0.1074
Epoch 20/50 - 1s 3ms/step - loss: 0.1031 - val_loss: 0.1018
Epoch 30/50 - 1s 3ms/step - loss: 0.1010 - val_loss: 0.0998
Epoch 40/50 - 1s 3ms/step - loss: 0.0998 - val_loss: 0.0986
Epoch 50/50 - 1s 2ms/step - loss: 0.0989 - val_loss: 0.0978
```

Os 32 atributos da primeira imagem de teste e a sua média (o dígito “7” da figura abaixo, regularização de atividade L_1).

```
encoded_imgs[0]:
[1.5224 3.9411 1.955  1.993  0.7612 1.9671 0.5298 1.7678 3.7771 0.4905
 0.     0.6125 1.2559 0.5273 0.     2.5278 0.3367 4.4388 2.1227 0.2478
 1.094  1.2809 1.9302 3.5686 0.418  1.4615 1.0767 0.5494 2.7701 1.49
 0.     0.3568]
np.mean(encoded_imgs[0]): 1.4616
```

Figura 5: Saída do autoencoder P3. O código gerado tem magnitude pequena e vários elementos nulos.

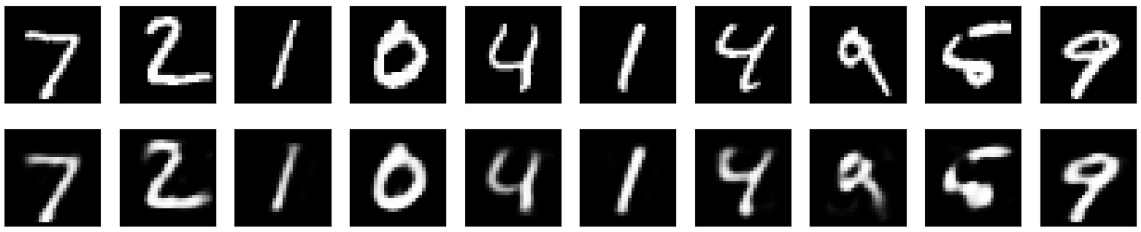


Figura 6: Linha superior: imagens originais. Linha inferior: imagens reconstruídos pelo autoencoder P3 com regularização de atividade L_1.

5. Autoencoder com rede profunda

O programa 4 usa uma rede densa com várias camadas para implementar um autoencoder com espaço latente de dimensão 32. O erro de teste agora diminui de 0,0916 para 0,0817, treinando o modelo durante 100 épocas.

```
#~/deep/algpi/autoencoder/ae3.py deep autoencoder
#https://blog.keras.io/building-autoencoders-in-keras.html
import tensorflow as tf
import tensorflow.keras as keras
from tensorflow.keras import layers, regularizers
from tensorflow.keras.datasets import mnist
import numpy as np
import matplotlib.pyplot as plt

keras.utils.set_random_seed(7)
tf.config.experimental.enable_op_determinism()

encoding_dim = 32

input_img = keras.Input(shape=(784,))
encoded = layers.Dense(128, activation='relu')(input_img)
encoded = layers.Dense(64, activation='relu')(encoded)
encoded = layers.Dense(encoding_dim, activation='relu')(encoded)
decoded = layers.Dense(64, activation='relu')(encoded)
decoded = layers.Dense(128, activation='relu')(decoded)
decoded = layers.Dense(784, activation='sigmoid')(decoded)
autoencoder = keras.Model(input_img, decoded)

from tensorflow.keras.utils import plot_model
plot_model(autoencoder, to_file='autoencoder3.png', show_shapes=True)
autoencoder.summary()

encoder = keras.Model(input_img, encoded)
decoder = keras.Model(encoded, decoded)

autoencoder.compile(optimizer=keras.optimizers.Adam(learning_rate=0.001),
                    loss='binary_crossentropy')

(x_train, _), (x_test, _) = mnist.load_data()
x_train = x_train.astype('float32') / 255
x_test = x_test.astype('float32') / 255
x_train = x_train.reshape((len(x_train), np.prod(x_train.shape[1:])))
x_test = x_test.reshape((len(x_test), np.prod(x_test.shape[1:])))

autoencoder.fit(x_train, x_train, epochs=100, batch_size=256, shuffle=True,
                validation_data=(x_test, x_test))
autoencoder.save("ae3.keras")
encoded_imgs = encoder.predict(x_test)
np.set_printoptions(precision=4)
print("encoded_imgs[0]:"); print(encoded_imgs[0])
print("np.mean(encoded_imgs[0]): %.4f"%(np.mean(encoded_imgs[0])))
decoded_imgs = decoder.predict(encoded_imgs)

n = 10  # How many digits we will display
plt.figure(figsize=(20, 4))
for i in range(n):
  ax = plt.subplot(2, n, i + 1)
  plt.imshow(x_test[i].reshape(28, 28), vmin=0, vmax=1)
  plt.gray()
  ax.get_xaxis().set_visible(False)
  ax.get_yaxis().set_visible(False)

  ax = plt.subplot(2, n, i + 1 + n)
  plt.imshow(decoded_imgs[i].reshape(28, 28), vmin=0, vmax=1)
  plt.gray()
  ax.get_xaxis().set_visible(False)
  ax.get_yaxis().set_visible(False)
plt.savefig("ae3_saida.png")
plt.show()
```

Programa 4: Autoencoder usando rede densa com várias camadas.

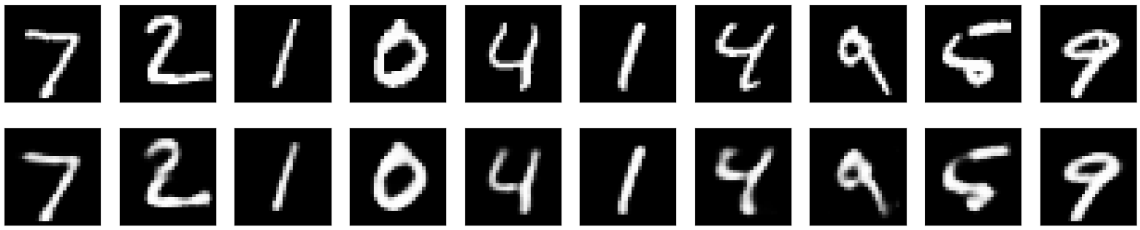


Figura 7: Imagens produzidas pelo autoencoder com rede profunda.

```
Epoch  97/100 - 1s 4ms/step - loss: 0.0819 - val_loss: 0.0817
Epoch  98/100 - 1s 4ms/step - loss: 0.0819 - val_loss: 0.0816
Epoch  99/100 - 1s 5ms/step - loss: 0.0819 - val_loss: 0.0821
Epoch 100/100 - 1s 3ms/step - loss: 0.0819 - val_loss: 0.0817


encoded_imgs[0]:
[ 4.7165 15.6407  0.     13.5595 15.5998 12.4687 10.6236 14.601   3.6524
  0.      6.5526  9.3773  7.773  12.2974 13.42    3.5391  0.      6.1369
 10.6376 13.6025  5.9728  0.      0.      4.2623 14.9408 11.6341  6.4575
 10.2296 14.0448 10.6825 10.5267  9.4736]
np.mean(encoded_imgs[0]): 8.5132
```

Figura 8: Saída do programa 4.

5. Autoencoder com dimensão do espaço latente 2

O programa abaixo usa uma rede profunda para implementar autoencoder usando espaço latente de dimensão 2. O erro de teste agora é 0,1821 (muito maior que 0,0817 do último programa), treinando o modelo durante 100 épocas. A saída fica borrada.

```
#~/deep/algpi/autoencoder/ae3b.py deep autoencoder
#https://blog.keras.io/building-autoencoders-in-keras.html
import tensorflow as tf
import tensorflow.keras as keras
from tensorflow.keras import layers, regularizers
from tensorflow.keras.datasets import mnist
import numpy as np
import matplotlib.pyplot as plt

keras.utils.set_random_seed(7)
tf.config.experimental.enable_op_determinism()

encoding_dim = 2

input_img = keras.Input(shape=(784,))
encoded = layers.Dense(64, activation='relu')(input_img)
encoded = layers.Dense(16, activation='relu')(encoded)
encoded = layers.Dense(encoding_dim, activation='relu')(encoded)
decoded = layers.Dense(16, activation='relu')(encoded)
decoded = layers.Dense(64, activation='relu')(decoded)
decoded = layers.Dense(784, activation='sigmoid')(decoded)
autoencoder = keras.Model(input_img, decoded)

from tensorflow.keras.utils import plot_model
plot_model(autoencoder, to_file='autoencoder3b.png', show_shapes=True)
autoencoder.summary()

encoder = keras.Model(input_img, encoded)
decoder = keras.Model(encoded, decoded)

autoencoder.compile(optimizer=keras.optimizers.Adam(learning_rate=0.001),
                    loss='binary_crossentropy')

(x_train, _), (x_test, _) = mnist.load_data()
x_train = x_train.astype('float32') / 255
x_test = x_test.astype('float32') / 255
x_train = x_train.reshape((len(x_train), np.prod(x_train.shape[1:])))
x_test = x_test.reshape((len(x_test), np.prod(x_test.shape[1:])))

autoencoder.fit(x_train, x_train, epochs=100, batch_size=256, shuffle=True,
                validation_data=(x_test, x_test))
autoencoder.save("ae3b.keras")
encoded_imgs = encoder.predict(x_test)
np.set_printoptions(precision=4)
print("encoded_imgs[0]:"); print(encoded_imgs[0])
print("np.mean(encoded_imgs[0]): %.4f"%(np.mean(encoded_imgs[0])))
decoded_imgs = decoder.predict(encoded_imgs)

n = 10  # How many digits we will display
plt.figure(figsize=(20, 4))
for i in range(n):
  ax = plt.subplot(2, n, i + 1)
  plt.imshow(x_test[i].reshape(28, 28), vmin=0, vmax=1)
  plt.gray()
  ax.get_xaxis().set_visible(False)
  ax.get_yaxis().set_visible(False)

  ax = plt.subplot(2, n, i + 1 + n)
  plt.imshow(decoded_imgs[i].reshape(28, 28), vmin=0, vmax=1)
  plt.gray()
  ax.get_xaxis().set_visible(False)
  ax.get_yaxis().set_visible(False)
plt.savefig("ae3b_saida.png")
plt.show()
```

Programa 5: Autoencoder com espaço latente de dimensão 2.

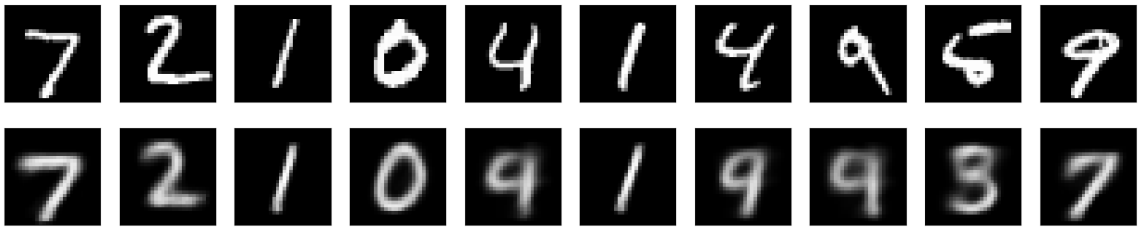


Figura 9: Imagens geradas pelo autoencoder com dimensão do espaço latente 2.

```
Epoch  96/100 - 1s 5ms/step - loss: 0.1793 - val_loss: 0.1823
Epoch  97/100 - 1s 4ms/step - loss: 0.1792 - val_loss: 0.1823
Epoch  98/100 - 1s 4ms/step - loss: 0.1792 - val_loss: 0.1825
Epoch  99/100 - 1s 4ms/step - loss: 0.1792 - val_loss: 0.1822
Epoch 100/100 - 1s 4ms/step - loss: 0.1791 - val_loss: 0.1821


encoded_imgs[0]:
[ 0.      21.3608]
np.mean(encoded_imgs[0]): 10.6804
```

Figura 10: Saída do autoencoder com dimensão do espaço latente 2.

6. Autoencoder com dimensão do espaço latente 2 e regularização de atividade L_2

O programa abaixo usa uma rede profunda para implementar autoencoder usando espaço latente de dimensão 2 e regularização de atividade L_2. Varrendo o espaço latente e reconstruindo as imagens, obtemos a figura F.

```
#~/deep/algpi/autoencoder/ae3c.py deep autoencoder
#https://blog.keras.io/building-autoencoders-in-keras.html
import tensorflow as tf
import tensorflow.keras as keras
from tensorflow.keras import layers, regularizers
from tensorflow.keras.datasets import mnist
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.utils import plot_model
import sys

keras.utils.set_random_seed(7)
tf.config.experimental.enable_op_determinism()

encoding_dim = 2

input_img = keras.Input(shape=(784,))
encoded = layers.Dense(64, activation='relu')(input_img)
encoded = layers.Dense(16, activation='relu')(encoded)
encoded = layers.Dense(encoding_dim, activation='relu',
            activity_regularizer=regularizers.l2(10e-5))(encoded)
decoded = layers.Dense(16, activation='relu')(encoded)
decoded = layers.Dense(64, activation='relu')(decoded)
decoded = layers.Dense(784, activation='sigmoid')(decoded)
autoencoder = keras.Model(input_img, decoded)
plot_model(autoencoder, to_file='ae3c_modelo.png', show_shapes=True)
autoencoder.summary()

encoder = keras.Model(input_img, encoded)
decoder = keras.Model(encoded, decoded)

autoencoder.compile(optimizer=keras.optimizers.Adam(learning_rate=0.001),
                    loss='binary_crossentropy')

(x_train, _), (x_test, _) = mnist.load_data()
x_train = x_train.astype('float32') / 255
x_test = x_test.astype('float32') / 255
x_train = x_train.reshape((len(x_train), np.prod(x_train.shape[1:])))
x_test = x_test.reshape((len(x_test), np.prod(x_test.shape[1:])))

autoencoder.fit(x_train, x_train, epochs=100, batch_size=256, shuffle=True,
                validation_data=(x_test, x_test))
autoencoder.save("ae3c.keras")

encoded_imgs = encoder.predict(x_test)
np.set_printoptions(precision=4)
for i in range(10):
  print("encoded_imgs[%d]:"%(i)); print(encoded_imgs[i])
  print("np.mean(encoded_imgs[%d]): %.4f"%(i,np.mean(encoded_imgs[i])))

decoded_imgs = decoder.predict(encoded_imgs)

n = 10  # How many digits we will display
plt.figure(figsize=(20, 4))
for i in range(n):
  ax = plt.subplot(2, n, i + 1)
  plt.imshow(x_test[i].reshape(28, 28), vmin=0, vmax=1)
  plt.gray()
  ax.get_xaxis().set_visible(False)
  ax.get_yaxis().set_visible(False)

  ax = plt.subplot(2, n, i + 1 + n)
  plt.imshow(decoded_imgs[i].reshape(28, 28), vmin=0, vmax=1)
  plt.gray()
  ax.get_xaxis().set_visible(False)
  ax.get_yaxis().set_visible(False)
plt.savefig("ae3c_saida1.png")
plt.show()

# Display a 2D manifold of the digits
n = 15  # figure with 15x15 digits
digit_size = 28
figure = np.zeros((digit_size * n, digit_size * n))
# We will sample n points within [-15, 15] standard deviations
grid_x = np.linspace(0, 4, n) #Imprime espaco latente [0, 4]
print("grid_x",grid_x)
grid_y = np.linspace(0, 4, n) #Imprime espaco latente [0, 4]
print("grid_y",grid_y)

for i, yi in enumerate(grid_x):
  for j, xi in enumerate(grid_y):
```

```
    z_sample = np.array([[xi, yi]]); print("i, j, z_sample", i, j, z_sample)
    x_decoded = decoder.predict(z_sample,verbose=0); #print("x_decoded",x_decoded)
    digit = x_decoded[0].reshape(digit_size, digit_size); #print("digit",digit)
    figure[i * digit_size: (i + 1) * digit_size,
           j * digit_size: (j + 1) * digit_size] = digit

plt.figure(figsize=(10, 10))
plt.imshow(figure,cmap="Greys"); plt.axis("off")
plt.savefig("ae3c_digitos.png")
plt.show()
```

Programa 6: Autoencoder com espaço latente de dimensão 2 e regularização de atividade L_2.
https://colab.research.google.com/drive/1xx9GpeV-1nR_7Te3M4bU29wL7oU9fpGh

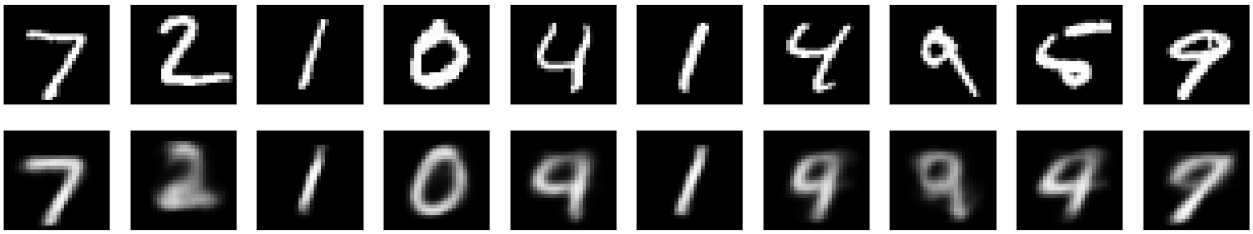


Figura F1.

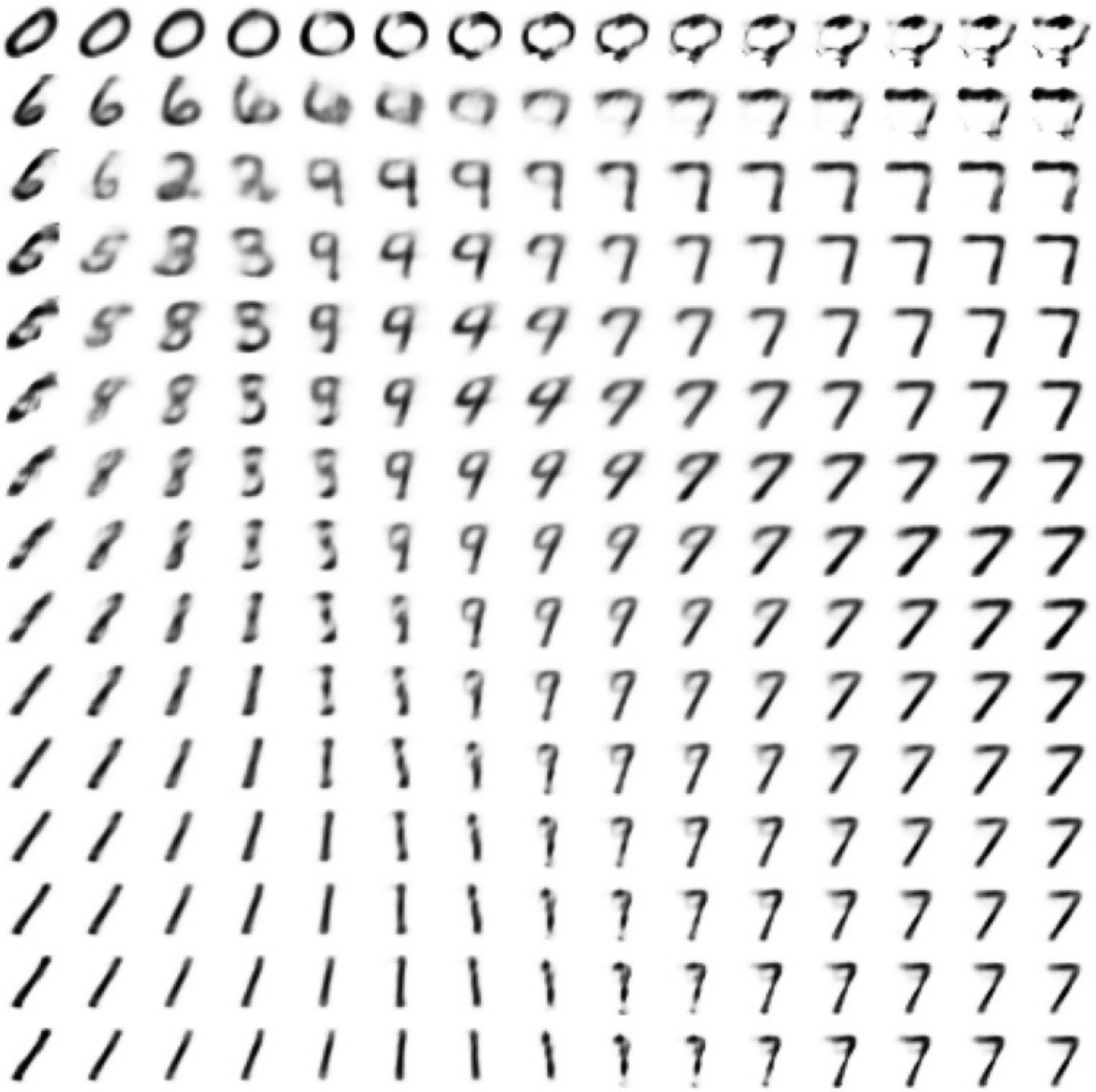


Figura F2: Espaço latente de dimensão 2 de autoencoder com regularização de atividade L_2.

[PSI3472 aula 11/12. Lição de casa #2 de 2. Vale 5,0]
Modifique o programa 6 para processar o conjunto de imagens Fashion MNIST. Gere imagens análogas às figuras F1 e F2.

Variational autoencoder

https://blog.keras.io/building-autoencoders-in-keras.html
https://keras.io/examples/generative/vae/
https://towardsdatascience.com/understanding-variational-autoencoders-vaes-f70510919f73
https://www.youtube.com/watch?v=9zKuYvjFFS8
https://www.deeplearningbook.com.br/variational-autoencoders-vaes-definicao-reducao-de-dimensionalidade-espaco-latente-e-regularizacao/

Código que funciona:
https://colab.research.google.com/drive/1cWeY2EKjWGvXEbzzjIxWNqNvYieXLHMH?hl=pt-br

Pode acontecer que algumas regiões do espaço latente estejam povoadas por atributos que não produzem as imagens de treino. Na figura F, no canto superior direito, há algumas imagens que não representam dígitos. Isto pode ser melhorado usando variational autoencoder.

Outros usos de autoencoders

O blog https://blog.keras.io/building-autoencoders-in-keras.html mostra como eliminar ruído usando autoencoder com CNN.

O vídeo https://www.youtube.com/watch?v=9zKuYvjFFS8 mostra autoencoder usado para fazer inpainting.

[PSI3472 aula 11, Fim]