Autoencoders

Guillermo Ruiz

Los autoencoders son redes neuronales que tienen dos partes, un **codificador** y un **decodificador**. El codificador toma los datos de entrada y los reduce de manera progresiva hasta obtener un tamaño determinado. Al espacio reducido se le llama espacio **latente**. Luego el decodificador toma el espacio latente y lo va aumentando de tamaño hasta recuperar el original. La red debe aprender a generar la entrada durante este proceso. En la siguiente figura se ve un diagrama básico de un autoencoder.

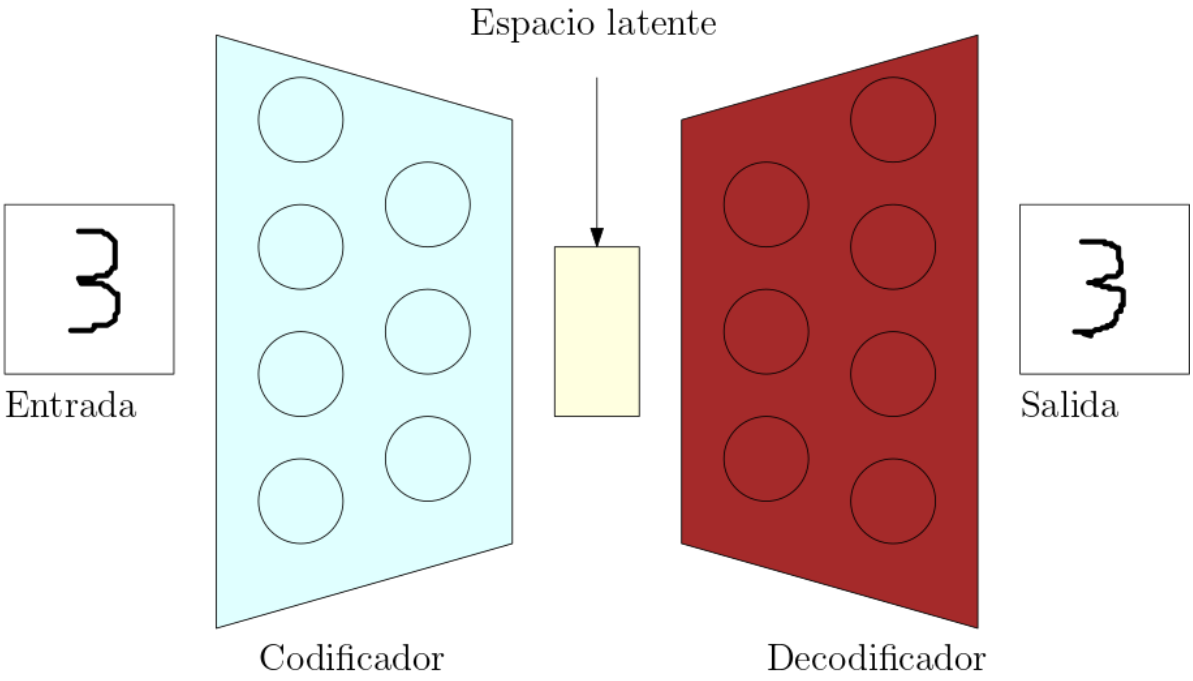


Figure 1: autoencoder

Como podemos observar, la entrada (en este caso una imagen) pasa al codificador que consta de varias capas, con salidas cada vez más pequeñas hasta llegar a un tamaño pequeño (espacio latente). Luego entra al decodificador que tiene las misma canditada de capas que el codificador

pero en orden inverso, lo que hace que sus dimensiones vayan aumentando hasta obtener el tamaño original. El objteivo es que la salida sea igual a la entrada.

En este proceso, la parte importante es el codificador, que se usa para hacer la reducción de dimensión. La idea del autoencoder es que la transformación del codificador condense la información de la entrada original en una cantidad más pequeña de espacio. Mientras más información se capte en la representación del espacio latente, mejor será la reconstrucción del decodificador.

Los autoencoders se pueden usar para reducir la dimensión de los datos, al igual que métodos como PCA. La principal diferencia es que las redes neuronales pueden aprender transformaciones no lineales de los datos. También son útiles para extracción de características que se pueden usar como entradas para otros métodos de parendizaje automático.

A continuación veremos un ejemplo de un autoencoder usando una red neuronal de **tensorflow**.

```
from tensorflow import keras
import matplotlib.pyplot as plt
from tensorflow.keras import layers, models, optimizers
import numpy as np
from sklearn.datasets import fetch_openml
```

Vamos a usar el conjunto de datos MNIST. Note que las imágenes se aplanaron para convertirlas en vectores de 784 dimensiones.

```
mnist = fetch_openml('mnist_784', as_frame=False)

X = mnist.data
y = mnist.target.astype(np.uint8)

X.shape, y.shape
```

```
C:\Users\msubr\Anaconda3\envs\tf28\lib\site-packages\sklearn\datasets\_openml.py:932: FutureW
  warn(
```

```
((70000, 784), (70000,))
```

Aquí definimos la estructura de la red neuronal. Esto se puede hacer usando el modelo `Sequential`. Se aguegan capas densas usando el método `add`. Para más información de cómo usar las redes neuronales, puede ver este material.

Todas las usan la función de activación `relu`. Veamos como definimos el codificador. La primera capa tiene 256 neuronas, el parámetro `input_shape` debe ir siempre en la primera capa e indica el tamaño del vector de entrada. Luego vienen dos capas de 128 y 64 neuronas. Luego viene la capa de donde obtenemos la reducción usando la variable `latent_dim` que en este caso es 8.

Luego viene la parte del decodificador con capas de 64, 128, 256 y 784 neuronas. Note que la entrada y la salida son del mismo tamaño.

```
latent_dim = 8
model = models.Sequential()
model.add(layers.Dense(256, activation='relu', input_shape=(784,)))
model.add(layers.Dense(128, activation='relu'))
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(latent_dim, activation='relu', name='latent_layer'))
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(128, activation='relu'))
model.add(layers.Dense(256, activation='relu'))
model.add(layers.Dense(784, activation=None))
```

Mostramos la estructura final de la red neuronal.

```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 256)               200960

 dense_1 (Dense)             (None, 128)               32896

 dense_2 (Dense)             (None, 64)                8256

 latent_layer (Dense)        (None, 8)                 520

 dense_3 (Dense)             (None, 64)                576

 dense_4 (Dense)             (None, 128)               8320
```

```
 dense_5 (Dense)             (None, 256)               33024

 dense_6 (Dense)             (None, 784)               201488

=================================================================
Total params: 486,040
Trainable params: 486,040
Non-trainable params: 0
_________________________________________________________________
```

Para el entrenamiento usamos el optimizador **Adam** con un `learning_rate` de 0.0005. Puede revisar esto para más información sobre optimizadores. La función de pérdida es la MeanSquaredError (mse). Entrenamos por 20 épocas y un tamaño del lote de 64. Note que los datos de entrada son los mismos que los de la salida.

```
optimizer = optimizers.Adam(learning_rate=0.0005)
model.compile(optimizer=optimizer, loss='mse')
```

```
model.fit(X, X, epochs=20, batch_size=64)
```

```
Epoch 1/20
1094/1094 [==============================] - 11s 6ms/step - loss: 2779.3694
Epoch 2/20
1094/1094 [==============================] - 7s 6ms/step - loss: 2143.0684
Epoch 3/20
1094/1094 [==============================] - 7s 6ms/step - loss: 2002.6642
Epoch 4/20
1094/1094 [==============================] - 7s 6ms/step - loss: 1920.2144
Epoch 5/20
1094/1094 [==============================] - 7s 6ms/step - loss: 1845.8418
Epoch 6/20
1094/1094 [==============================] - 7s 6ms/step - loss: 1795.5627
Epoch 7/20
1094/1094 [==============================] - 7s 6ms/step - loss: 1748.1317
Epoch 8/20
1094/1094 [==============================] - 8s 7ms/step - loss: 1727.4437
Epoch 9/20
1094/1094 [==============================] - 8s 7ms/step - loss: 1709.0842
Epoch 10/20
1094/1094 [==============================] - 7s 7ms/step - loss: 1692.7809
Epoch 11/20
```

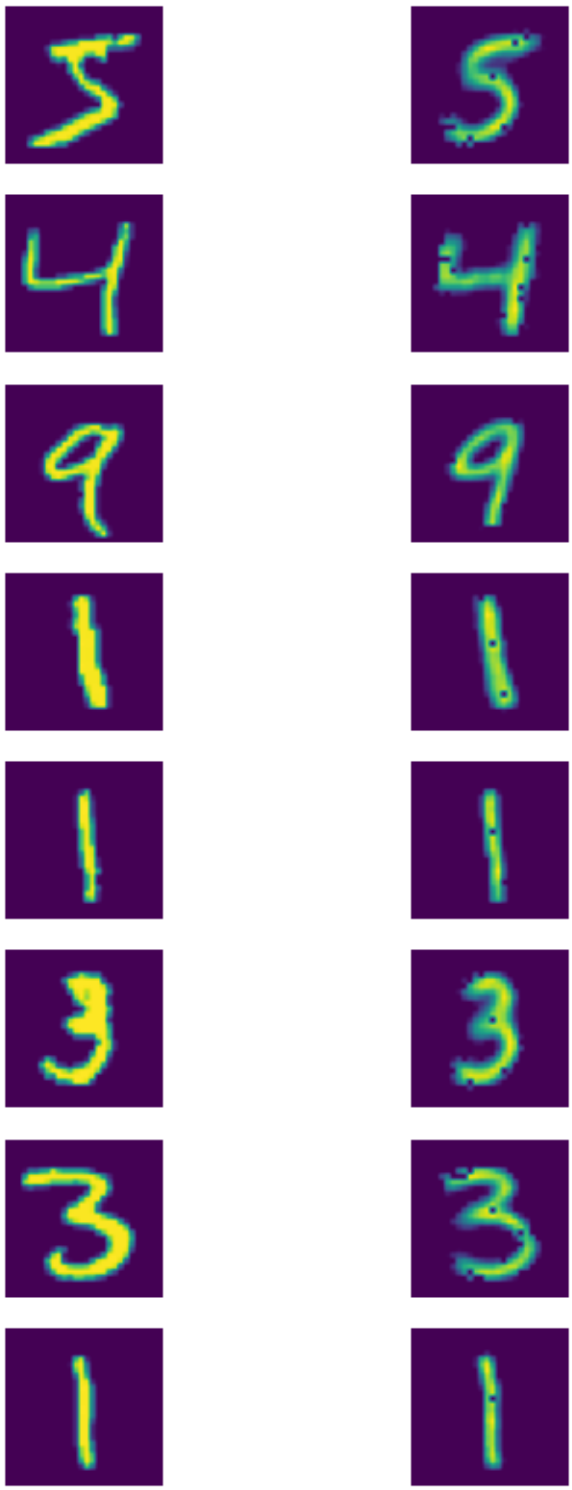
```
1094/1094 [==============================] - 7s 6ms/step - loss: 1668.1685
Epoch 12/20
1094/1094 [==============================] - 7s 6ms/step - loss: 1641.3842
Epoch 13/20
1094/1094 [==============================] - 7s 6ms/step - loss: 1629.3245
Epoch 14/20
1094/1094 [==============================] - 7s 7ms/step - loss: 1620.0317
Epoch 15/20
1094/1094 [==============================] - 8s 7ms/step - loss: 1610.9496
Epoch 16/20
1094/1094 [==============================] - 8s 7ms/step - loss: 1602.6168
Epoch 17/20
1094/1094 [==============================] - 8s 7ms/step - loss: 1581.4395
Epoch 18/20
1094/1094 [==============================] - 8s 7ms/step - loss: 1570.2271
Epoch 19/20
1094/1094 [==============================] - 8s 7ms/step - loss: 1562.6382
Epoch 20/20
1094/1094 [==============================] - 8s 7ms/step - loss: 1556.8164

<keras.callbacks.History at 0x13a3a4a7b80>
```

Predecimos la salida de las primeras imágenes para ver cómo es la reconstrucción.

```
pred = model.predict(X[0:16])
```

```
1/1 [==============================] - 0s 188ms/step
```

```
plt.figure(figsize=(5, 10))
for i in range(0, 16, 2):
    plt.subplot(8, 2, i + 1)
    plt.imshow(X[i].reshape(28, 28), )
    plt.axis('off')
    plt.subplot(8, 2, i + 2)
    plt.imshow(pred[i].reshape(28, 28), )
    plt.axis('off')
plt.show()
```



En la figura, la columna de la izquierda es la imagen original y la de la derecha es la reconstruida usando los datos del espacio latente. Podemos ver que en general coinciden las reconstrucciones con los originales, pero se pueden ver que algunos pixeles se pierden.

Esto nos dice que la información contenida en la reducción es suficiente para una buena reconstrucción.

A continuación vamos a usar una reducción a 2 dimensiones para luego graficarla en el plano. Note que usamos un learning rate diferente y entrenamos por 40 epochs.

```
latent_dim = 2
model = models.Sequential()
model.add(layers.Dense(256, activation='relu', input_shape=(784,)))
model.add(layers.Dense(128, activation='relu'))
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dense(latent_dim, activation='relu', name='latent_layer'))
model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(128, activation='relu'))
model.add(layers.Dense(256, activation='relu'))
model.add(layers.Dense(784, activation=None))
```

```
optimizer = optimizers.Adam(learning_rate=0.001)
model.compile(optimizer=optimizer, loss='mse')
model.fit(X, X, epochs=40, batch_size=256)
```

```
Epoch 1/40
274/274 [==============================] - 3s 7ms/step - loss: 4936.9868
Epoch 2/40
274/274 [==============================] - 2s 7ms/step - loss: 4872.7939
Epoch 3/40
274/274 [==============================] - 2s 7ms/step - loss: 4872.1338
Epoch 4/40
274/274 [==============================] - 2s 8ms/step - loss: 4871.6758
Epoch 5/40
274/274 [==============================] - 2s 8ms/step - loss: 4870.4453
Epoch 6/40
274/274 [==============================] - 2s 7ms/step - loss: 4856.6577
Epoch 7/40
274/274 [==============================] - 2s 7ms/step - loss: 4373.6528
Epoch 8/40
```

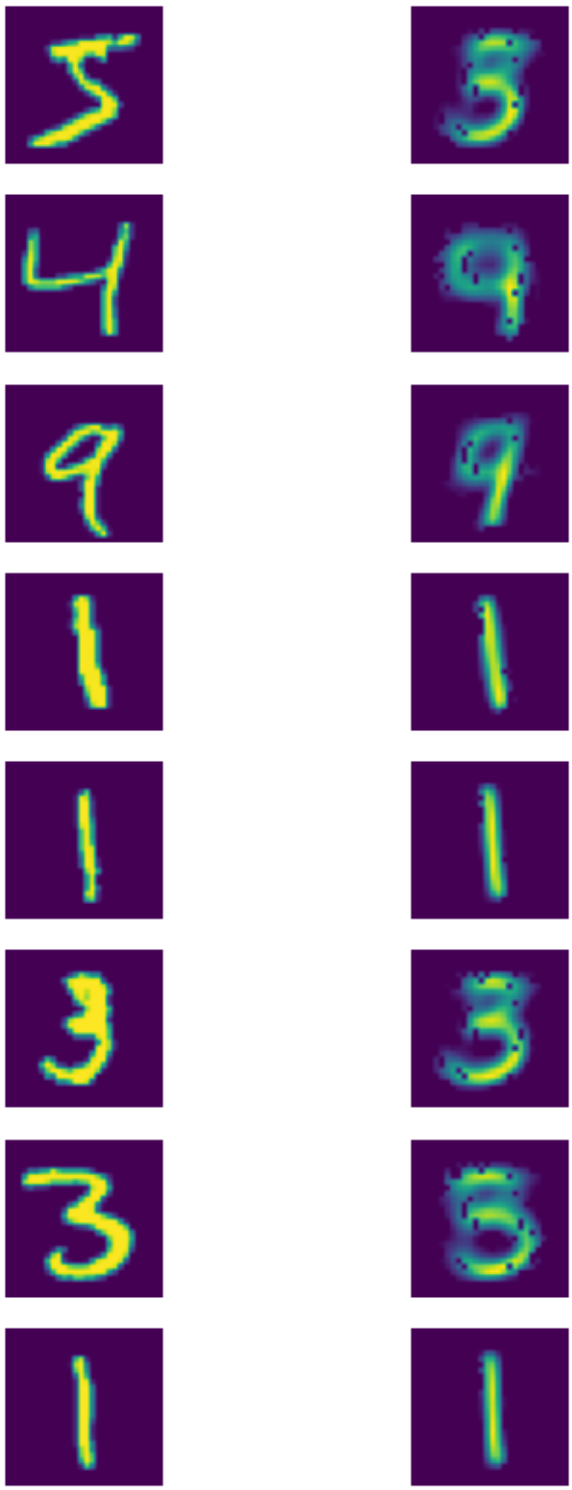
```
274/274 [==============================] - 2s 8ms/step - loss: 3706.4385
Epoch 9/40
274/274 [==============================] - 2s 7ms/step - loss: 3501.2966
Epoch 10/40
274/274 [==============================] - 2s 8ms/step - loss: 3426.7988
Epoch 11/40
274/274 [==============================] - 2s 9ms/step - loss: 3360.5801
Epoch 12/40
274/274 [==============================] - 2s 9ms/step - loss: 3325.4185
Epoch 13/40
274/274 [==============================] - 2s 8ms/step - loss: 3287.2947
Epoch 14/40
274/274 [==============================] - 2s 8ms/step - loss: 3230.2600
Epoch 15/40
274/274 [==============================] - 2s 8ms/step - loss: 3204.4666
Epoch 16/40
274/274 [==============================] - 2s 8ms/step - loss: 3190.5093
Epoch 17/40
274/274 [==============================] - 2s 8ms/step - loss: 3167.5552
Epoch 18/40
274/274 [==============================] - 2s 8ms/step - loss: 3149.0017
Epoch 19/40
274/274 [==============================] - 2s 7ms/step - loss: 3120.2781
Epoch 20/40
274/274 [==============================] - 2s 8ms/step - loss: 3098.8584
Epoch 21/40
274/274 [==============================] - 2s 7ms/step - loss: 3086.4980
Epoch 22/40
274/274 [==============================] - 2s 7ms/step - loss: 3065.4722
Epoch 23/40
274/274 [==============================] - 2s 8ms/step - loss: 3040.5637
Epoch 24/40
274/274 [==============================] - 2s 8ms/step - loss: 3028.0232
Epoch 25/40
274/274 [==============================] - 2s 7ms/step - loss: 3031.9636
Epoch 26/40
274/274 [==============================] - 2s 7ms/step - loss: 3005.3247
Epoch 27/40
274/274 [==============================] - 2s 7ms/step - loss: 2981.2097
Epoch 28/40
274/274 [==============================] - 2s 7ms/step - loss: 2974.6970
Epoch 29/40
274/274 [==============================] - 2s 7ms/step - loss: 2959.8865
```

```
Epoch 30/40
274/274 [==============================] - 2s 7ms/step - loss: 2977.1387
Epoch 31/40
274/274 [==============================] - 2s 7ms/step - loss: 2966.9175
Epoch 32/40
274/274 [==============================] - 2s 7ms/step - loss: 2941.5247
Epoch 33/40
274/274 [==============================] - 2s 7ms/step - loss: 2927.2522
Epoch 34/40
274/274 [==============================] - 2s 7ms/step - loss: 2909.7759
Epoch 35/40
274/274 [==============================] - 2s 8ms/step - loss: 2905.3472
Epoch 36/40
274/274 [==============================] - 2s 8ms/step - loss: 2906.7852
Epoch 37/40
274/274 [==============================] - 2s 7ms/step - loss: 2908.0137
Epoch 38/40
274/274 [==============================] - 2s 8ms/step - loss: 2892.3220
Epoch 39/40
274/274 [==============================] - 2s 7ms/step - loss: 2907.3442
Epoch 40/40
274/274 [==============================] - 2s 7ms/step - loss: 2882.4148

<keras.callbacks.History at 0x13d21e4cd90>
```

Mostramos las imágenes reconstruidas. Se puede apreciar que las reconstrucciones se parecen a las originales.

```
pred = model.predict(X[0:16])
plt.figure(figsize=(5, 10))
for i in range(0, 16, 2):
    plt.subplot(8, 2, i + 1)
    plt.imshow(X[i].reshape(28, 28), )
    plt.axis('off')
    plt.subplot(8, 2, i + 2)
    plt.imshow(pred[i].reshape(28, 28), )
    plt.axis('off')
plt.show()
```

```
1/1 [==============================] - 0s 117ms/step
```



Con la siguiente instrucción creamos el modelo `reductor_model` que lo único que hace es tomar la parte del codificador. Con esto, la salida es la reducción del espacio latente.

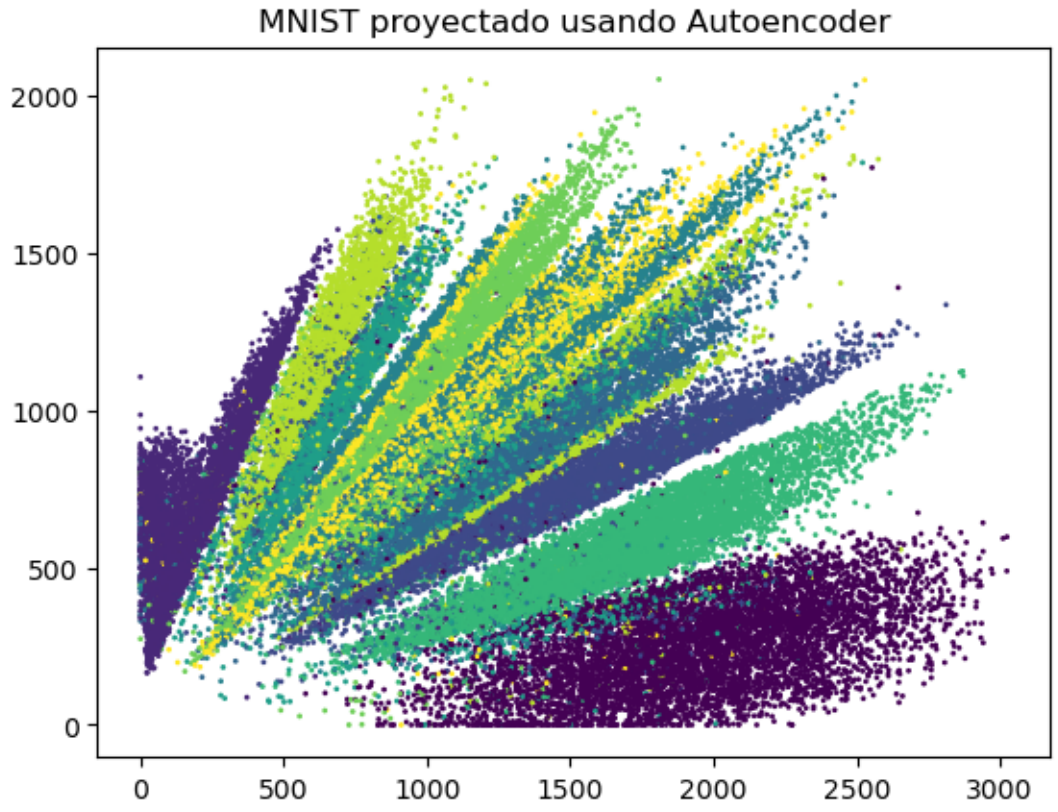
```
reductor_model = models.Model(inputs=model.inputs,
                              outputs=model.get_layer('latent_layer').output)
```

Aplicamos el codificador a los datos para obtener la reducción a dos dimensiones y lo guardamos en `X_t`.

```
X_t = reductor_model.predict(X)
```

```
2188/2188 [==============================] - 4s 2ms/step
```

```
plt.scatter(X_t[:, 0], X_t[:, 1], c=y, s=1)
plt.title("MNIST proyectado usando Autoencoder")
plt.show()
```



Esta es la reducción del conjunto de datos a dos dimensiones. Podemos ver que los datos se agrupan por color pero hay una zona donde se revuelven las imágenes de 2 clases.