Bootcamp Inteligencia Artificial
Nivel Innovador

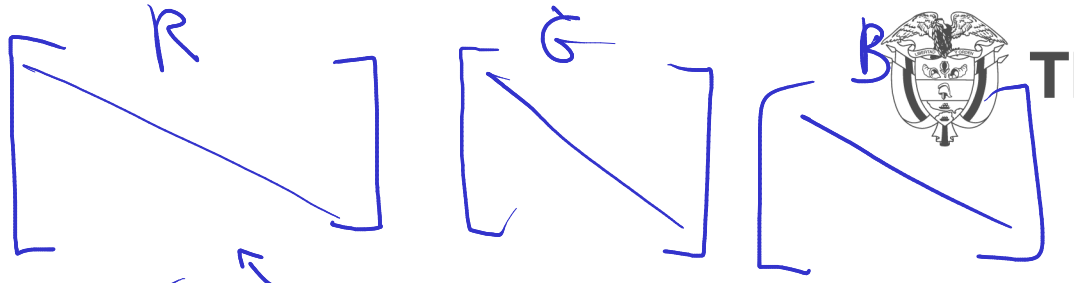
Semana 13:

Aplicaciones de Deep Learning

Agenda

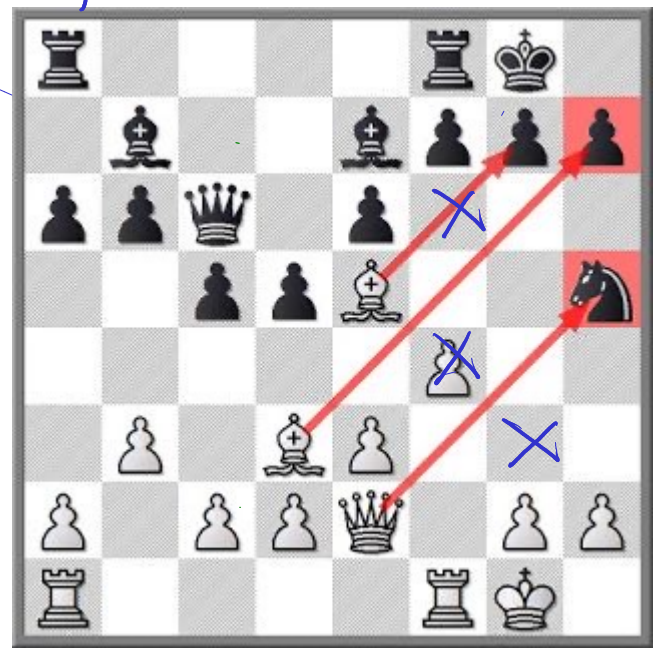
1. Autoencoders y GANs 1

1.1 Introducción



Una computadora no tiene problemas de memoria como nosotros los humanos – las computadoras pueden memorizar secuencias extraordinariamente largas, para eso las usamos.

Pero mediante el autoencoder, obligamos a la computadora a que NO memorice, sino a que busque algún patrón o alguna regla que le permita rearmar los datos desde el principio.

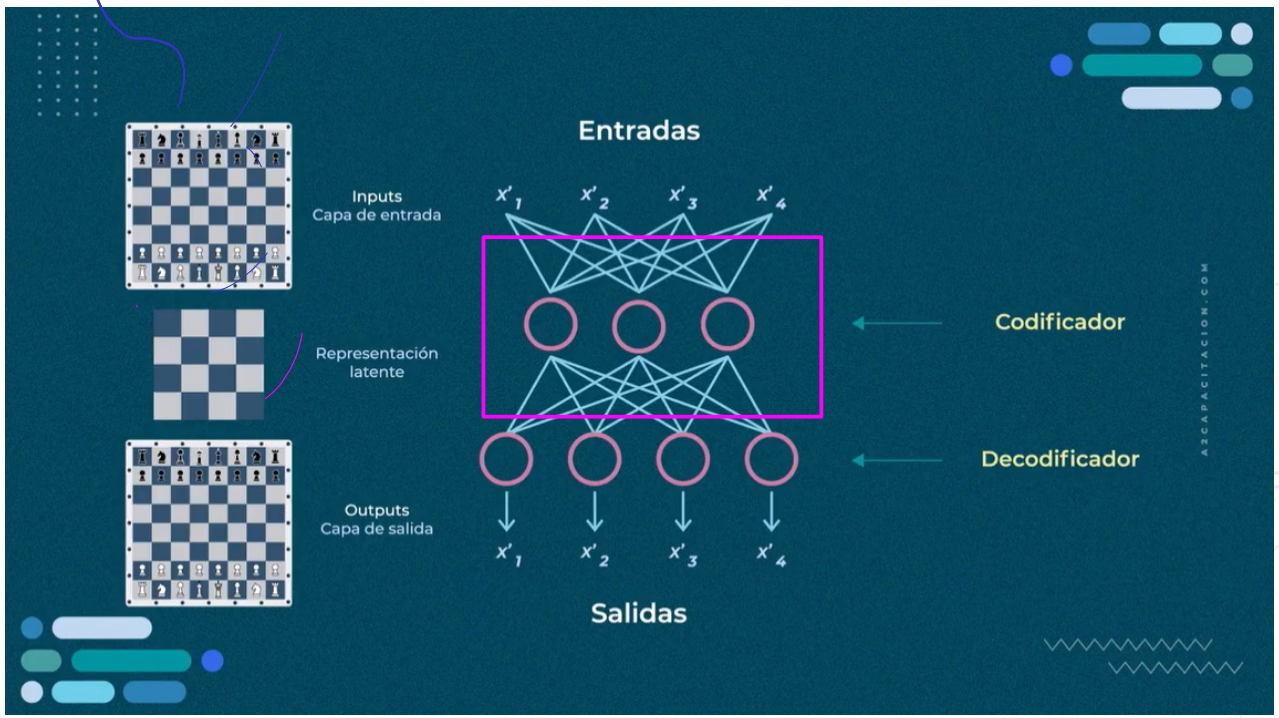


La relación entre patrones y memoria está muy estudiada en el ajedrez. Los jugadores expertos de ajedrez son capaces de memorizar fácilmente las posiciones de todas las piezas en un juego solo de ver el tablero unos segundos.

1.2 Introducción

Un autoencoder generalmente tiene la misma arquitectura que una red neuronal Densa, excepto que el número de neuronas en la capa de salida tiene que ser igual a la cantidad de entradas.

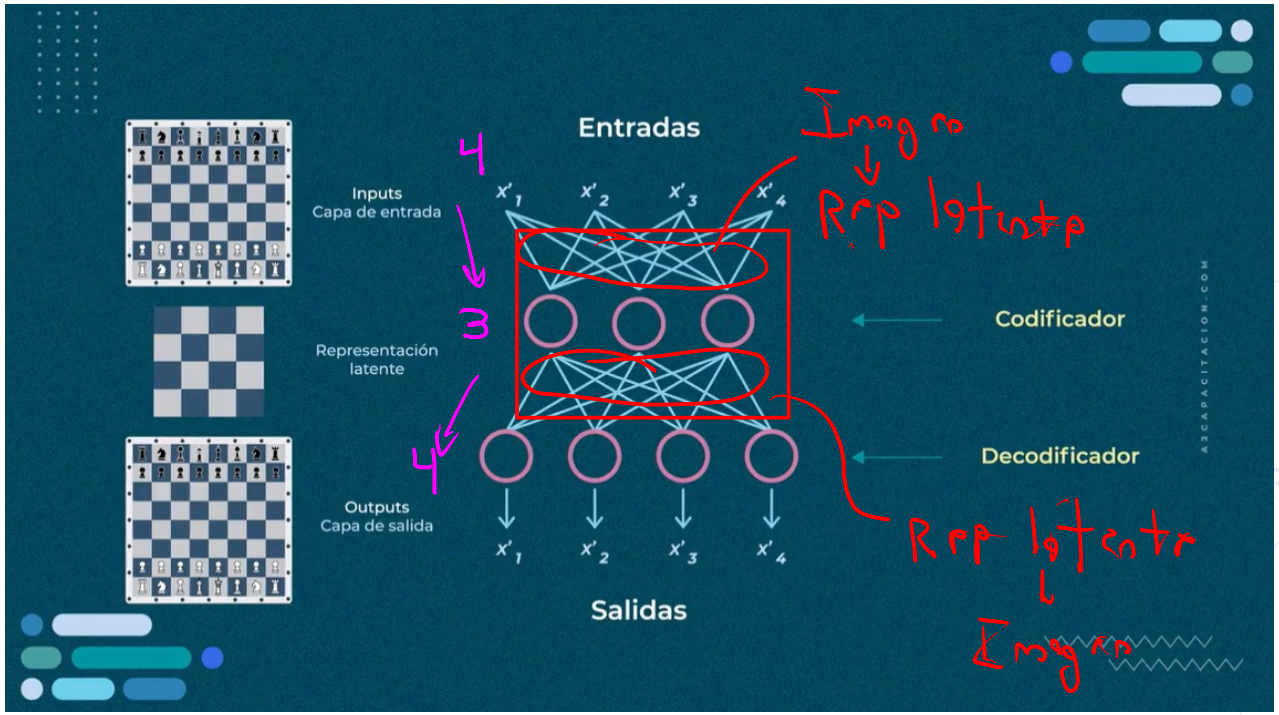
En este ejemplo hay 4 entradas, una capa oculta de 3 neuronas, que sería el encoder, y una capa de salida de 4 neuronas, el decoder. Las salidas se llaman reconstrucciones por que el autoencoder intenta reconstruir las entradas, y la función de costo contiene una pérdida de reconstrucción que penaliza el modelo cuando las reconstrucciones son diferentes de las entradas.



1.3 Introducción

El autoencoder se dice que es sub completo por que la representación interna es de menor dimensión que los datos de entrada.

Un autoencoder sub completo no puede simplemente copiar las entradas y pegarlas como si fueran la salida y ya. Tiene que aprender los patrones de la entrada para poder reconstruirla y escupirte la salida como el autoencoder la entendió. Por lo mismo, se ve obligado a entender las características más importantes de la entrada.



1.4 Armar un PCA con un Autoencoder

Vamos a armar un autoencoder que solo use activaciones lineales y su función de costo sea el Error Promedio Cuadrado.

Aprendizaje no superivisado, no tenemos y

Comenzaremos importando las bibliotecas necesarias.
Traer Scikit.

```
import sklearn
```

Importar tensorflow.

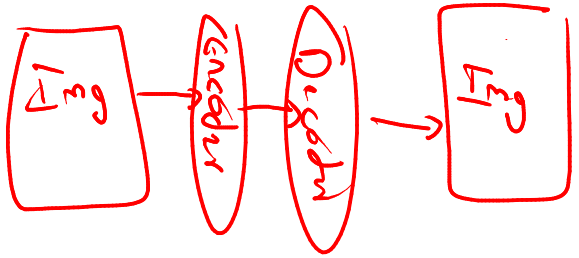
```
import tensorflow as tf
```

Traete Matplotlib.

```
import matplotlib.pyplot as plt
```

Numpy, Os, pandas.

```
import numpy as np
import os
import pandas as pd
```



$$\min\left(\frac{1}{2}\sum_{i=1}^{n}\left(Img_{in} - Img_{out}\right)^2\right)$$

1.5 Armar un PCA con un Autoencoder

Semillas a 42.

```
np.random.seed(42)
tf.random.set_seed(42)
```

Arma un encoder y un decoder.

```
encoder = tf.keras.Sequential([tf.keras.layers.Dense(2)])
decoder = tf.keras.Sequential([tf.keras.layers.Dense(3)])
```

Arma un autonecoder, el modelo que realmente vamos a usar.

```
autoencoder = tf.keras.Sequential([encoder,decoder])
```

Compila tu autoencoder.

```
optimizer = tf.keras.optimizers.SGD(learning_rate=0.5)
autoencoder.compile(loss="mse", optimizer=optimizer)
```

1.6 Armar un PCA con un Autoencoder

El código, como podrán ver, no es nada diferente a los MLPs. No obstante, revisemos:

- Organizamos el autoencoder en 2 componentes – el Codificador y El descodificador. Ambos son modelos secuenciales regulares con una sola capa densa cada uno, y el autocodificador es un modelo secuencial que contiene el codificador seguido del decodificador.

- El número de salidas del autoencoders igual al número de entradas.

- Para hacer un PCA simple, no usamos ninguna función de activación (ósea, todas las neuronas son lineales) y la función de costo es el MSE.

1.7 Armar un PCA con un Autoencoder

Genera un Dataset 3D aleatorio para usarlo con el autoencoder..

```
from scipy.spatial.transform import Rotation

m = 60
X = np.zeros((m, 3))
np.random.seed(42)
angles = (np.random.rand(m) ** 3 + 0.5) * 2 * np.pi
X[:, 0], X[:, 1] = np.cos(angles), np.sin(angles) * 0.5
X += 0.28 * np.random.randn(m, 3)
X = Rotation.from_rotvec([np.pi / 29, -np.pi / 20, np.pi / 4]).apply(X)
X_train = X + [0.2, 0, 0.2]
```

Entrena tu autoencoder.

```
history = autoencoder.fit(X_train, X_train, epochs=500, verbose=False)
```

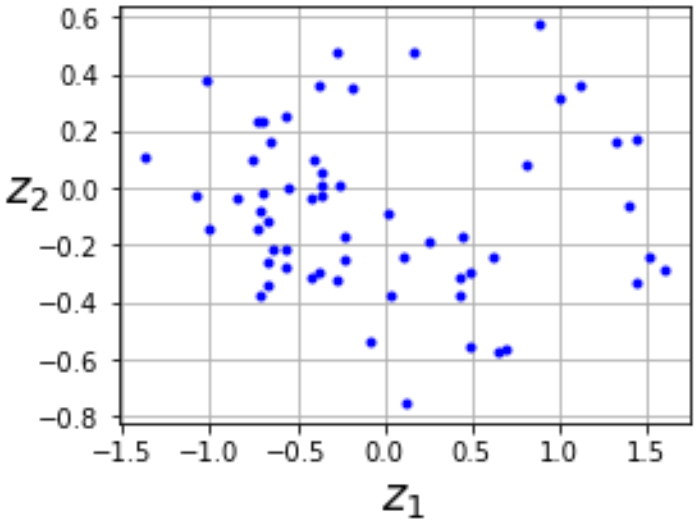
Checa resultados.

```
codings = encoder.predict(X_train)
```

1.8 Armar un PCA con un Autoencoder

Visualiza tu Plot.

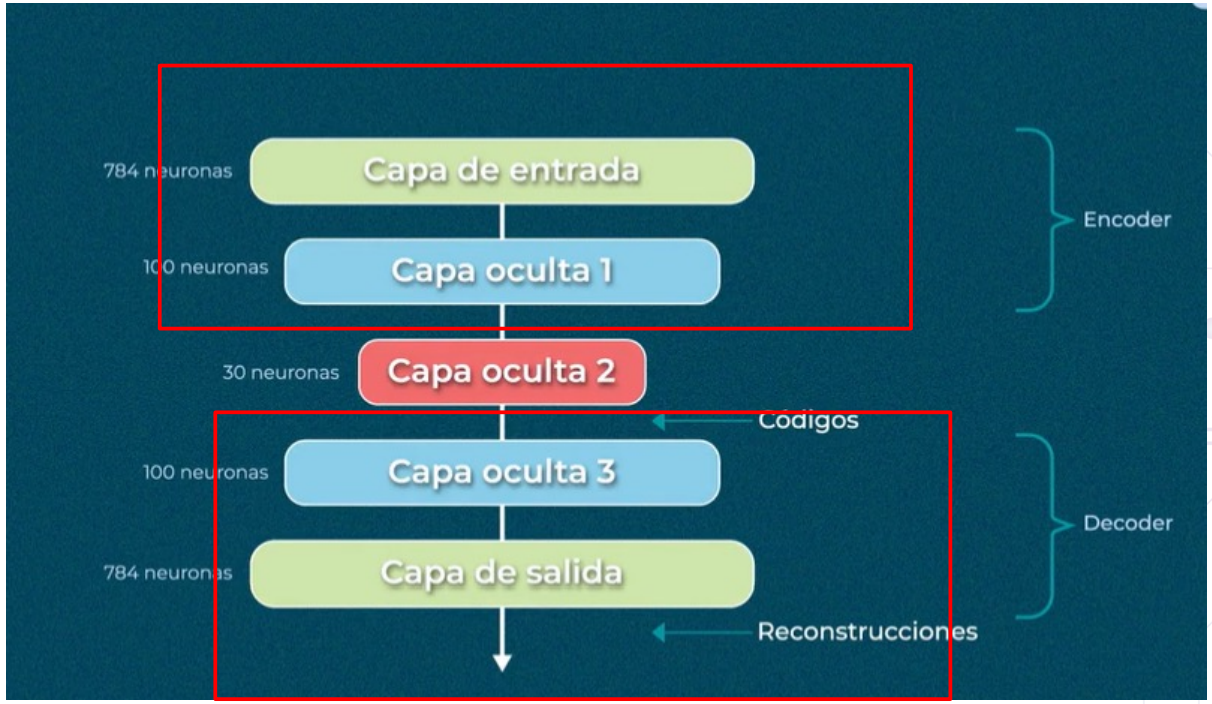
```
fig = plt.figure(figsize=(4,3))
plt.plot(codings[:,0], codings[:, 1], "b.")
plt.xlabel("$z_1$", fontsize=18)
plt.ylabel("$z_2$",          fontsize=18,
rotation=0)
plt.grid(True)
plt.show()
```



1.9 Autoencoders Apilados

Igual que cualquier otra red neuronal que hemos discutido hasta ahora, los autoencoders pueden tener capas ocultas múltiples. Cuando esto ocurre, les llamamos stacked autoencoders o ENCODERS APILADOS (o también, encoders profundos).

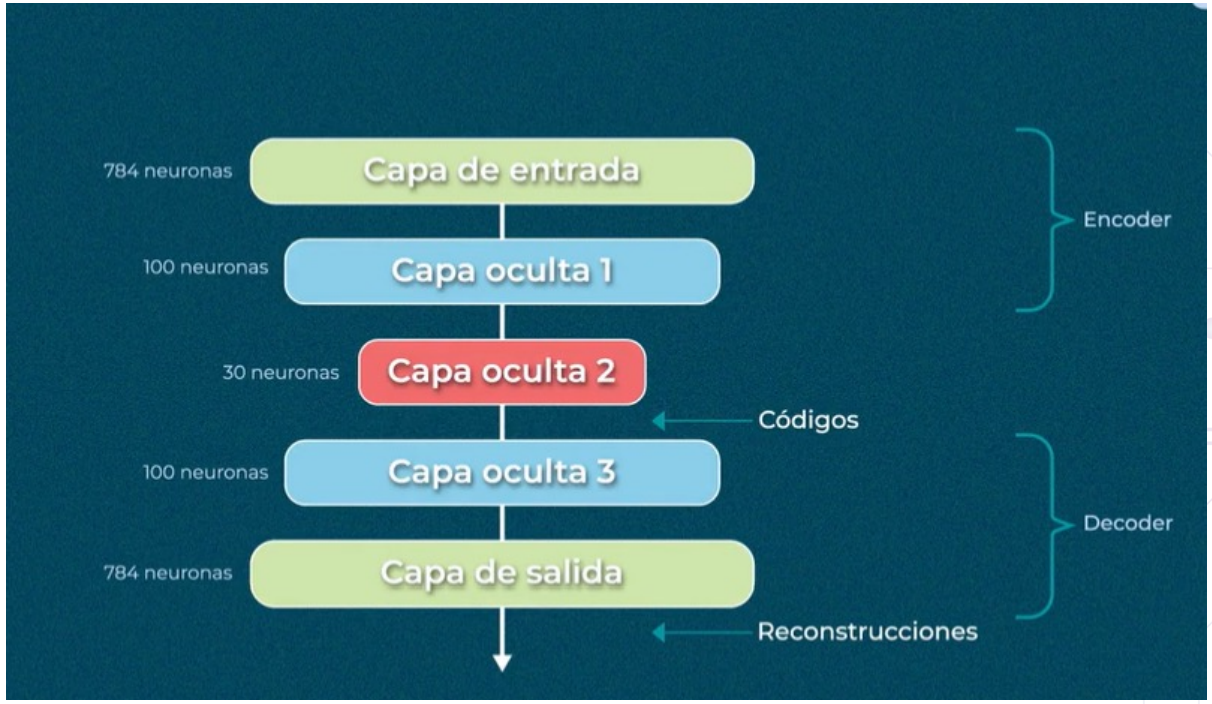
Agregarle más capas ayuda al autoencoder a aprender patrones más complejos. Dicho eso, recuerda que no estamos buscando hacer el autoencoder MUY poderoso. Imagina un encoder tan potente que simplemente mapea cada entrada a un número arbitrario, y el decoder aprende la inversa.

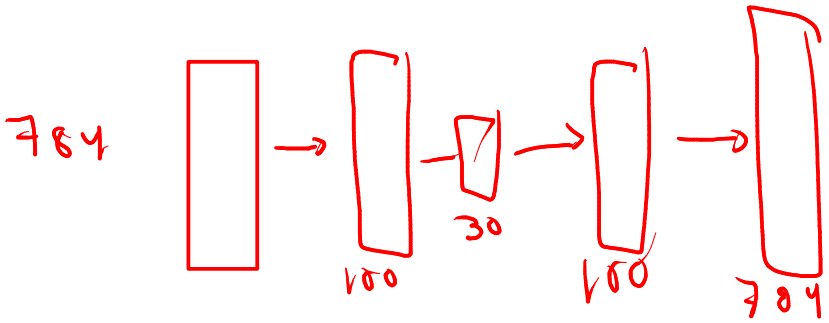


1.10 Autoencoders Apilados

Obviamente un autoencoder va a reconstruir los datos perfectamente, pero no habrá aprendido a hacer generalizaciones útiles en el proceso.

La arquitectura de un autoencoder apilado es típicamente simétrica con respecto a la capa central oculta (la capa de código). Para ponerlo de manera sencilla, se ve como un sándwich. Por ejemplo, el autoencoder para MNIST puede tener 784 entradas, seguido de una capa oculta de 100 neuronas, y luego una capa oculta de 30 neuronas y luego otra capa oculta de 100 neuronaes, y la salida con 784 neuronas.





1.11 Autoencoders Apilados

Implementación en Keras

Puedes implementar un autoencoder apilado mucho como un MLP profundo regular. En particular, la misma técnica que usamos en el capítulo 11 para entrenar redes profundas puede ser aplicada.

Por ejemplo, el siguiente código construye un autoencoder para MNIST de moda:

Carga el dataset de mnist moda, ajustalo/255 y separalo en train, test y valid.

```
fashion_mnist = tf.keras.datasets.fashion_mnist.load_data()
(X_train_full, y_train_full), (X_test, y_test) = fashion_mnist
X_train_full = X_train_full.astype(np.float32) / 255
X_test = X_test.astype(np.float32) / 255
X_train, X_valid = X_train_full[:-5000], X_train_full[-5000:]
y_train, y_valid = y_train_full[:-5000], y_train_full[-5000:]
```

1.12 Autoencoders Apilados

Arma el encoder.

```
stacked_encoder = tf.keras.Sequential([
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(100, activation = "relu"),
    tf.keras.layers.Dense(30, activation = "relu"),
])
```

Arma el decoder.

```
stacked_decoder = tf.keras.Sequential([
    tf.keras.layers.Dense(100, activation = "relu"),
    tf.keras.layers.Dense(28 * 28),
    tf.keras.layers.Reshape([28, 28])
])
```

Arma el Autoencoder Apilado.

```
stacked_ae = tf.keras.Sequential([stacked_encoder, stacked_decoder])
```

Compila el modelo.

```
stacked_ae.compile(loss="mse", optimizer="nadam")
```

Ejecuta.

```
history = stacked_ae.fit(X_train, X_train, epochs=20,
                         validation_data=(X_valid, X_valid))
```

1.13 Autoencoders Apilados

Repasemos el código:

- Primero, hacemos 2 modelos, encoder y decoder.
- El encoder toma imágenes de 28 x 28, las aplana para que cada imagen sea representada como un vector de 784, y luego procesa estos vectores a través de 2 capas densas de tamaños decrecientes. Ambas usan SELU de activación. Para cada imagen de entrada, el encoder saca un tamaño de vector de 30.
- El decoder toma códigos de tamaño 30, y los procesa a través d 2 capas DENSAS usando capas de tamaños crecientes – 100 unidades y luego 784 unidades, y las reformula como vectores finales de 28 x 28 para que tenga la misma figura que las imágenes de entrada.
- Cuando compilamos el autoencoder apilado, usamos la perdida binaria de entropía cruzada en vez del error promedio cuadrado. Estamos tratando la tarea de reconstrucción como un problema de clasificación multi etiqueta.
- Finalmente, vamos a entrenar usando X_Train como las entradas y los objetivos.

1.14 Visualizar las Reconstrucciones

Una manera de asegurarnos que el autoencoder está propiamente entrenado es comparar las entradas y las salidas. La diferencia no debería ser mucha. Vamos viendo la diferencia entre nuestras imágenes de entrada y sus reconstrucciones.

Crea una función para graficar las reconstrucciones

```
def plot_reconstructions(model, images=X_valid, n_images=5):
    reconstructions = np.clip(model.predict(images[:n_images]), 0, 1)
    fig = plt.figure(figsize=(n_images * 1.5, 3))
    for image_index in range(n_images):
        plt.subplot(2, n_images, 1 + image_index)
        plt.imshow(images[image_index], cmap="binary")
        plt.axis("off")
        plt.subplot(2, n_images, 1 + n_images + image_index)
        plt.imshow(reconstructions[image_index], cmap="binary")
        plt.axis("off")
```

Ejecuta tu función y muestra las reconstrucciones

```
plot_reconstructions(stacked_ae)
plt.show()
```

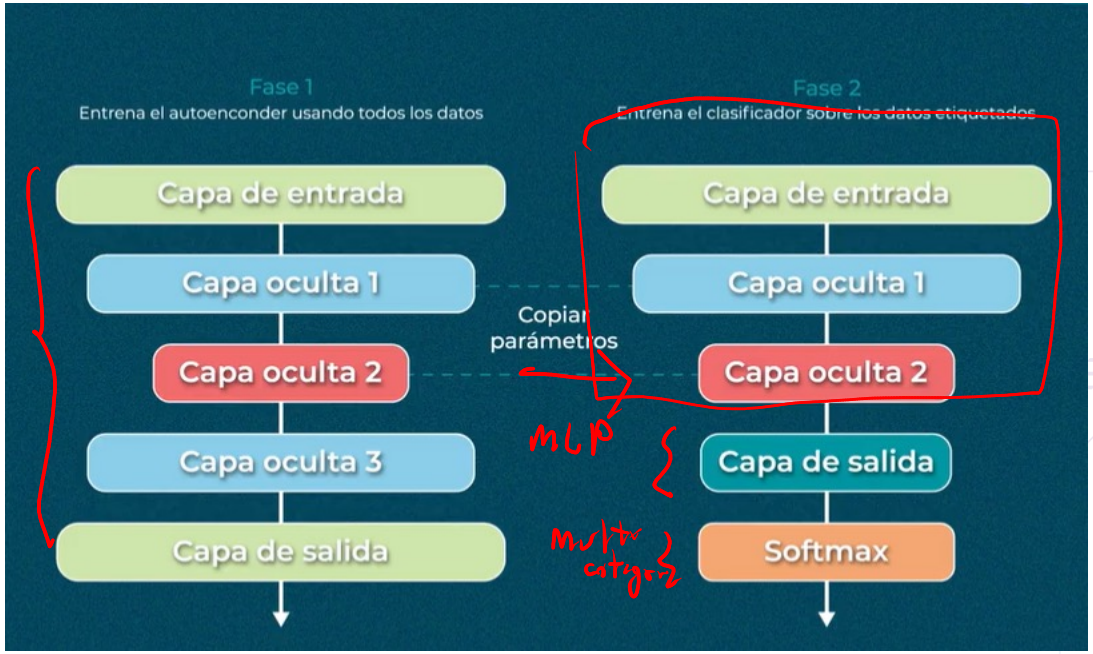
Las reconstrucciones son reconocibles, pero tienen algo de pérdida de calidad. Tal vez tenemos que entrenar el modelo por más tiempo, o hacer el encoder y decoder más profundos, o hacer los codings más largos. Pero si hacemos la red muy poderosa, va a hacer reconstrucciones perfectas sin aprender generalizaciones

1.15 Preentrenamiento Sin Supervisión usando Autoencoders Encimados

Si estas resolviendo una tarea compleja supervisada pero no tienes un montón de datos de entrenamiento etiquetados, una solución es encontrar una red neuronal que realice una tarea similar y reúse sus capas inferiores.

Esto hace posible entrenar un modelo de alto desempeño usando pocos datos de entrenamiento por que tu red neuronal no tiene que aprender todas las variables de bajo nivel; simplemente va a reusar los detectores aprendidos por la red existente.

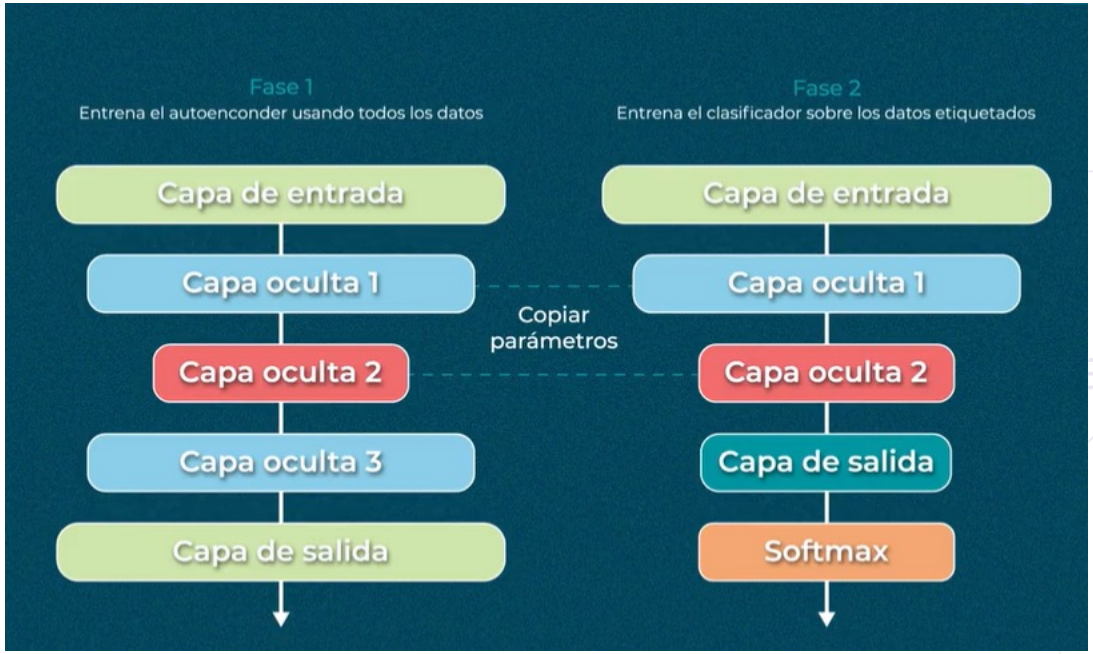
De manera similar, si tienes un dataset grande, pero todo es sin etiqueta, puedes primero entrenar un autoencoder apilado usando todos los datos, y luego reutilizar las capas inferiores para crear una red neuronal para tu tarea de verdad y entrenarla usando los datos etiquetados



1.16 Preentrenamiento Sin Supervisión usando Autoencoders Encimados

Por ejemplo, puedes usar un autoencoder apilado para hacer preentrenamientos sin supervisión para una red neuronal de clasificación. Cuando entrenas un clasificador, si en realidad no tienes muchos datos etiquetados de entrenamiento, tal vez quieras congelar las capas pre entrenadas.

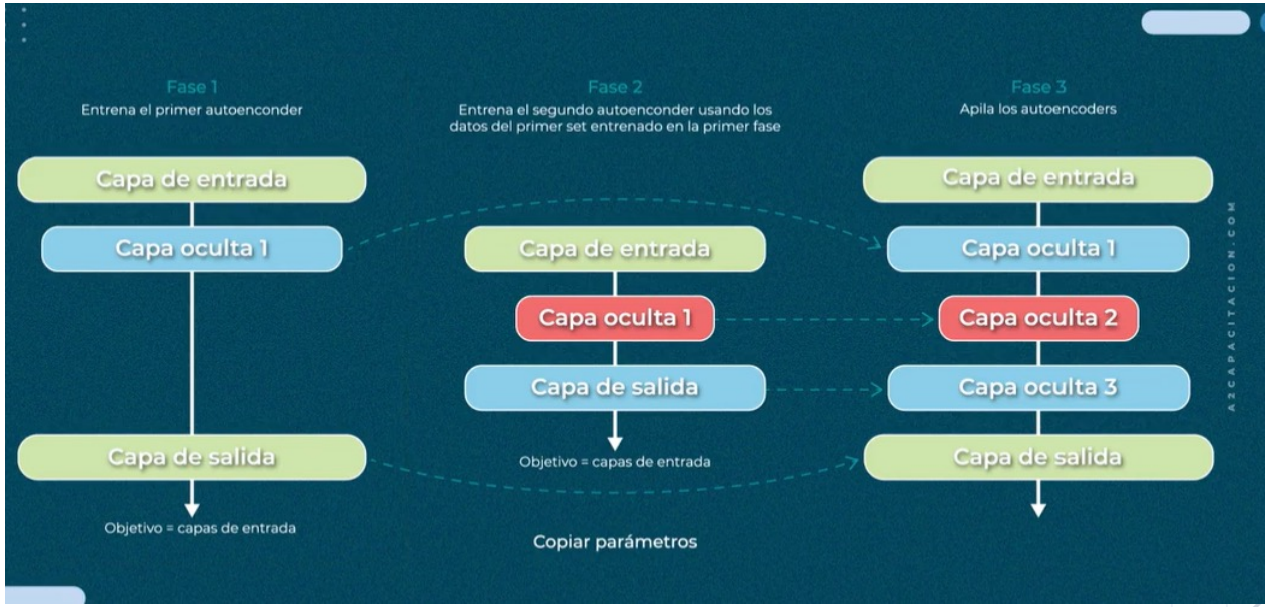
La implementación no tiene nada de chiste – simplemente entrena el autoencoder usando todos los datos de entrenamiento (etiquetados y sin etiquetar), y luego reúsa su capa de codificador para crear una nueva red neuronal.



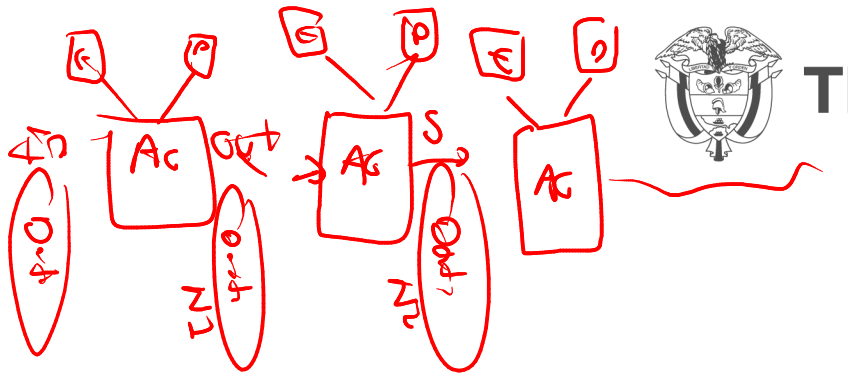
1.17 Un Autoencoder a la vez

En vez de entrenar todo el autoencoder apilado de un solo jalón como le acabamos de hacer, es posible entrenar un autoencoder estrecho a la vez, y luego apilarlos todos en un solo autoencoder apilado. Esta técnica no se usa mucho hoy en día, pero la vas a ver en papers referenciada como “greedy layerwise training”, así que vale la pena saber a qué se refieren.

Durante la primera fase del entrenamiento, el primer autoencoder aprende a reconstruir las entradas. Luego codificamos todo el entrenamiento usando el primer autoencoder, y esto nos da un nuevo set de entrenamiento comprimido.

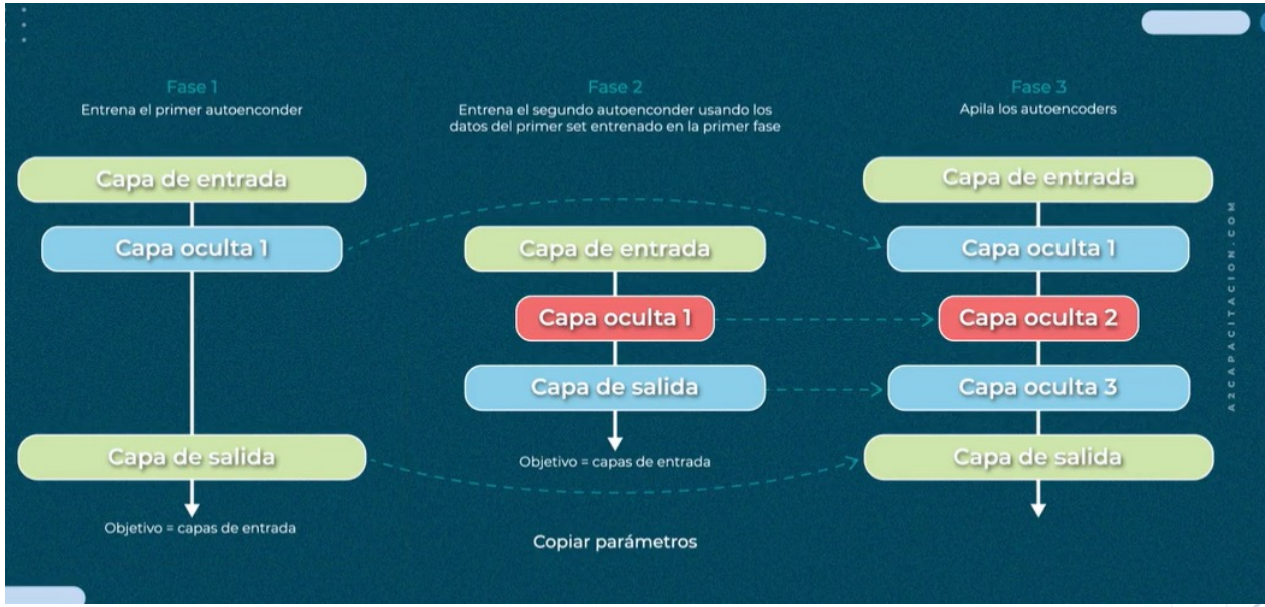


1.18 Un Autoencoder a la vez



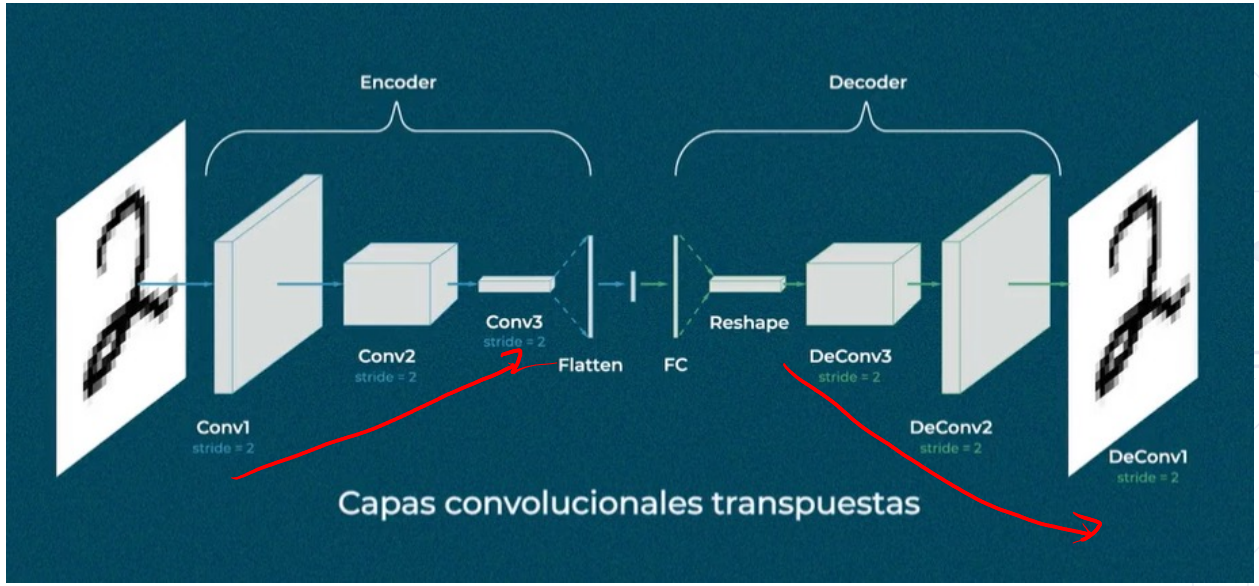
Luego entrenamos un segundo autoencoder en este nuevo dataset. Esta es la segunda fase del entrenamiento. Finalmente, construimos un gran sándwich usando todos estos autoencoders.

Esto nos da un autoencoder apilado final. Podrimos fácilmente entrenar más autoencoders de esta manera, construyendo un autoencoder apilado muy profundo.



1.20 Autoencoders Convolucionales

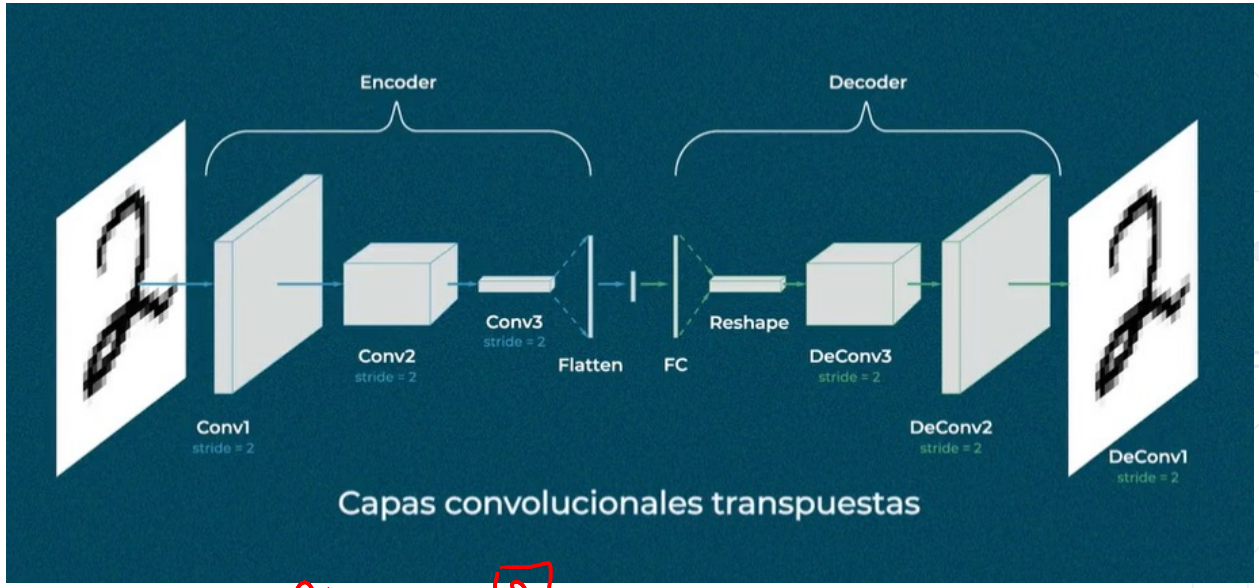
Si estas lidiando con imágenes, entonces los autoencoders que hemos visto hasta ahora no van a funcionar bien (salvo que sean imágenes muy sosas como las de MNIST). Como ya sabemos, la red por excelencia para lidiar con imágenes más complejas son las redes convolucionales.

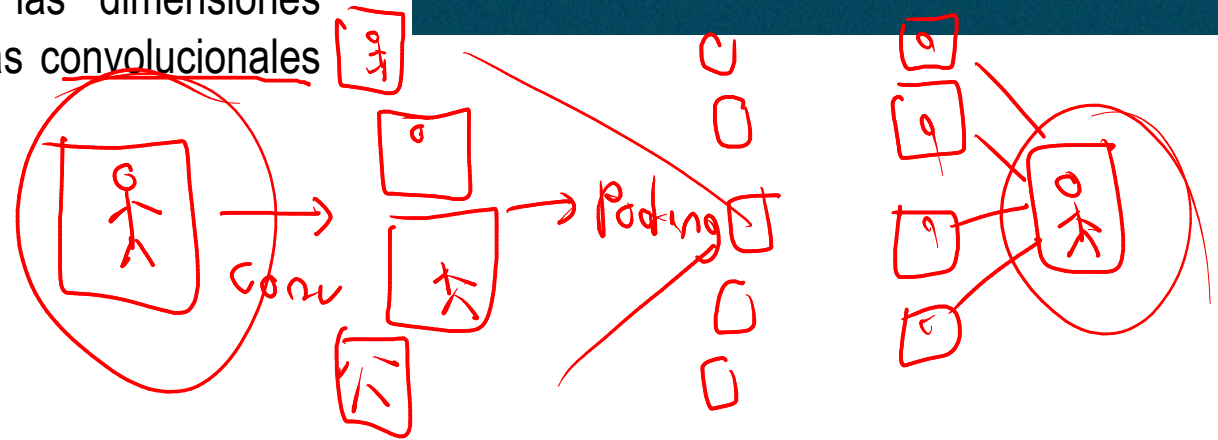


1.21 Autoencoders Convolucionales

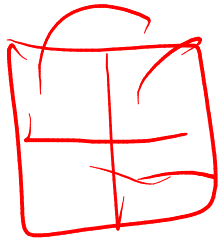
Así que, si vas a construir un autoencoder para imágenes, pues simplemente vas a necesitar construir un autoencoder convolucional. El encoder es un CNN regular compuesto de varias capas convolucionales y capas de pooling.

Reduce la dimensionalidad espacial de las entradas, mientras incrementa la profundidad. El decoder debe de hacer lo inverso (aumentar la resolución de la imagen y reducir su profundidad de regreso a las dimensiones originales), y para esto puedes usar capas convolucionales transpuestas.



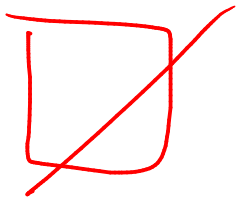


1.22 Autoencoders Convolucionales



Veamos un ejemplo sencillo para MNIST de moda.

A diferencia del pasado, este modelo no tiene capas densas, ahora tendremos capas convolucionales.



Arma el encoder convolucional.

```
conv_encoder = tf.keras.Sequential([
    tf.keras.layers.Reshape([28,28,1]),
    tf.keras.layers.Conv2D(16, 3, padding = "same", activation = "relu"),
    tf.keras.layers.MaxPool2D(pool_size = 2),
    tf.keras.layers.Conv2D(32, 3, padding="same", activation="relu"),
    tf.keras.layers.MaxPool2D(pool_size=2),  # output: 7 × 7 x 32
    tf.keras.layers.Conv2D(64, 3, padding="same", activation="relu"),
    tf.keras.layers.MaxPool2D(pool_size=2),  # output: 3 × 3 x 64
    tf.keras.layers.Conv2D(30, 3, padding="same", activation="relu"),
    tf.keras.layers.GlobalAvgPool2D()  # output: 30
])
```

1.23 Autoencoders Convolucionales

Arma el decoder convolucional.

```
conv_decoder = tf.keras.Sequential([
    tf.keras.layers.Dense(3*3*16),
    tf.keras.layers.Reshape((3,3,16)),
    tf.keras.layers.Conv2DTranspose(32,3, strides = 2, activation = "relu"),
    tf.keras.layers.Conv2DTranspose(16, 3, strides=2, padding="same",
                                    activation="relu"),
    tf.keras.layers.Conv2DTranspose(1, 3, strides=2, padding="same"),
    tf.keras.layers.Reshape([28, 28])
])
```

Arma el autoencoder.

```
conv_ae = tf.keras.Sequential([conv_encoder, conv_decoder])
```

Compila y Ajusta tu modelo.

```
conv_ae.compile(loss="mse", optimizer="nadam")

history = conv_ae.fit(X_train, X_train, epochs=10,
                      validation_data=(X_valid, X_valid))
```

Revisa los resultados visualmente.

```
plot_reconstructions(conv_ae)
plt.show()
```

1.24 Autoencoders Recurrentes

Ahora, si quieres hacer un autoencoder para secuencias como series de tiempo o texto… ¿con que lo harías?

Obviamente con Redes Neuronales Recurrentes, ya que se adaptan mucho mejor que las redes densas.

Construir un autoencoder Recurrente es muy simple, el encoder es un RNN de secuencia a vector que comprime la entrada hasta un solo vector. El decodificador es un RNN vector-a-secuencia que hace lo inverso.

1.25 Autoencoders Recurrentes

Encoder.

```
recurrent_encoder = tf.keras.Sequential([
    tf.keras.layers.LSTM(100, return_sequences = True),
    tf.keras.layers.LSTM(30)])
```

Decoder.

```
recurrent_decoder = tf.keras.Sequential([
    tf.keras.layers.RepeatVector(28),
    tf.keras.layers.LSTM(100, return_sequences = True),
    tf.keras.layers.Dense(28)])
```

Autoencoder.

```
recurrent_ae = tf.keras.Sequential([recurrent_encoder, recurrent_decoder])
```

Compila.

```
recurrent_ae.compile(loss="mse", optimizer="nadam")
```

Historial.

```
history = recurrent_ae.fit(X_train, X_train, epochs=10,
                           validation_data=(X_valid, X_valid))

plot_reconstructions(recurrent_ae)
plt.show()
```

1.26 Autoencoders Recurrentes

Este autoencoder recurrente puede procesar secuencias de cualquier largo, con 28 dimensiones por paso de tiempo. Esto quiere decir que podemos procesar MNIST de moda si tratamos cada imagen como 28 secuencias de filas, en cada paso de tiempo, el RNN va a procesar una sola fila de 28 pixeles.

Obviamente podrías un autoencoder recurrente para cualquier tipo de secuencia.

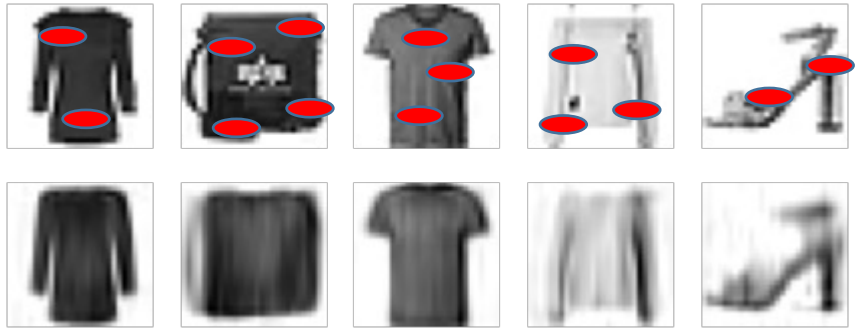
1.27 Autoencoders para quitar Ruido

Una manera diferente de obligar al autoencoder a aprender características útiles es agregar ruido a sus entradas, entrenándolo para recuperarse de las entradas originales que no tenían nada de ruido.

Básicamente estos se llaman “Denoising Autoencoders” que se podría traducir como Autoencoders que quitan ruido.

Este ruido puede ser puro ruido gaussiano agregado a las entradas, o puede ser entradas que simplemente dejamos de considerar, igualito que como se aplica en una capa de dropout.

La implementación es hiper sencilla – es un autoencoder regular apilado con una capa adicional de droput aplicada a las entradas del encoder. También, puedes usar una capa de GaussianNoise. Solo hay que tener en mente que estas capas solo estarán activas durante el entrenamiento.



1.28 Autoencoders para quitar Ruido

Encoder. La parte importante va aquí, en este caso es la línea de Dropout.

```
dropout_encoder = tf.keras.Sequential([
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(100, activation = "relu"),
    tf.keras.layers.Dense(30, activation = "relu")])
```

Decoder.

```
dropout_decoder = tf.keras.Sequential([
    tf.keras.layers.Dense(100, activation="relu"),
    tf.keras.layers.Dense(28 * 28),
    tf.keras.layers.Reshape([28, 28])])
```

Autoencoder.

```
dropout_ae = tf.keras.Sequential([dropout_encoder, dropout_decoder])
```

Compilar.

```
dropout_ae.compile(loss="mse", optimizer="nadam")
```

Historial.

```
history = dropout_ae.fit(X_train, X_train, epochs=10,
                         validation_data=(X_valid, X_valid))
```

Revisa que fue de tus resultados.

```
dropout = tf.keras.layers.Dropout(0.5)
plot_reconstructions(dropout_ae, dropout(X_valid, training=True))
plt.show()
```

1.29 Autoencoders para quitar Ruido

Hay que notar como el autoencoder adivina detalles que no están en la entrada. Como puedes ver, los autoencoders para quitar ruido no solo pueden ser usados para visualización de datos o entrenamiento sin supervisión, también pueden ser usados para quitar ruido de imágenes.

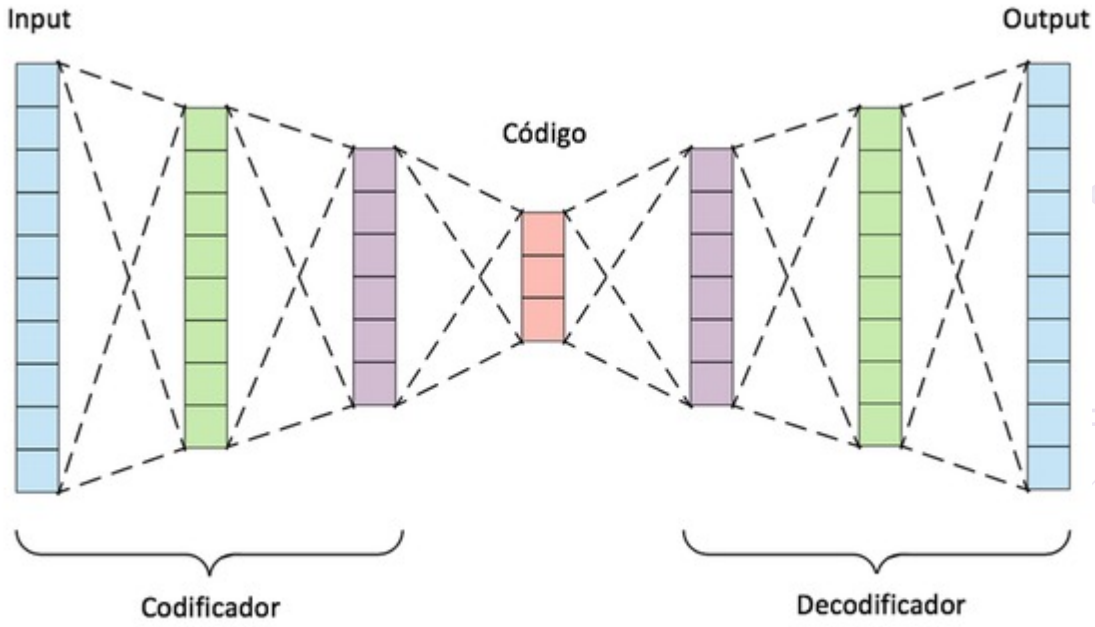
1.29 Autoencoders para quitar Ruido

Hay que notar como el autoencoder adivina detalles que no están en la entrada. Como puedes ver, los autoencoders para quitar ruido no solo pueden ser usados para visualización de datos o entrenamiento sin supervisión, también pueden ser usados para quitar ruido de imágenes.

1.30 Autoencoders Dispersos

Otra clase de cuello de botella que generalmente lleva a una buena extracción de características es la dispersión. Básicamente vamos a agregar un término a la función de costo para que reduzca el número de neuronas activas en a la capa de codificación.

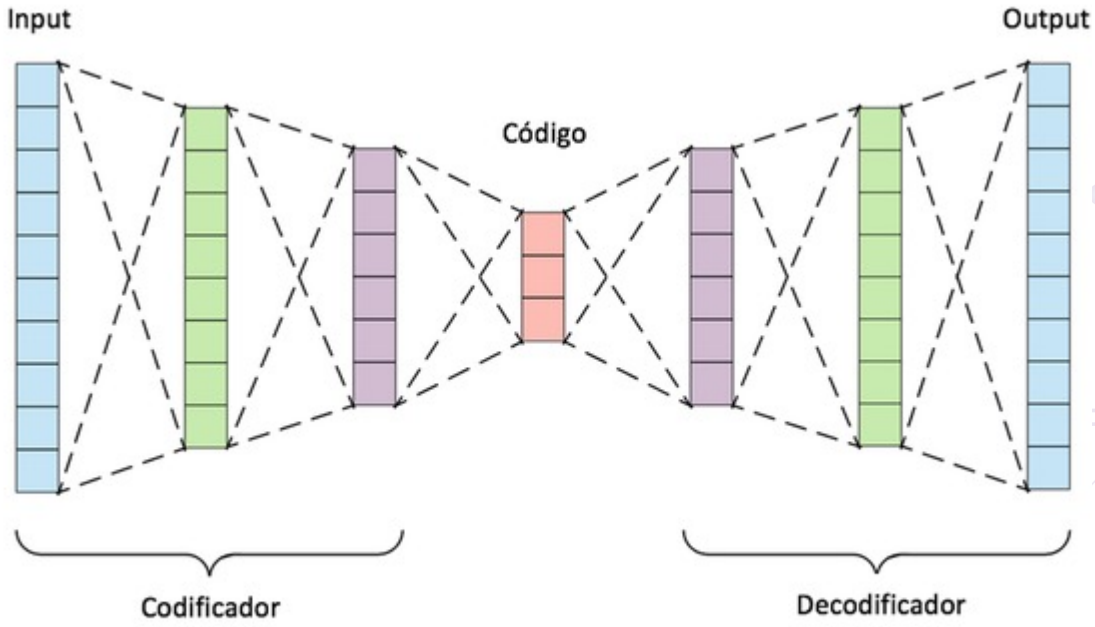
Por ejemplo, podríamos obligarlo a que tenga solo 5% de neuronas significativamente activas en a la capa de código. Esto obliga al autoencoder a representar cada entrada como una combinación de un pequeño número de activaciones.



1.31 Autoencoders Dispersos

Como resultado, cada neurona en la capa de código típicamente acaba representado una característica útil.

Un método simple de echar a andar esto es usar la función de activación sigmoide en la capa de codificación para restringir los valores de código entre 0 y 1. Usar una capa grande de codificación (ejemplo, con 300 unidades) y agregar regularización L1 a las activaciones de la capa de encoding.



1.32 Autoencoders Dispersos

Encoder

```
sparse_l1_encoder = tf.keras.Sequential([
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(100, activation="relu"),
    tf.keras.layers.Dense(300, activation="sigmoid"),
    tf.keras.layers.ActivityRegularization(l1=1e-4)])
```

Decoder, no necesita nada especial.

```
sparse_l1_decoder = tf.keras.Sequential([
    tf.keras.layers.Dense(100, activation="relu"),
    tf.keras.layers.Dense(28 * 28),
    tf.keras.layers.Reshape([28, 28])])
```

Autoencoder

```
sparse_l1_ae = tf.keras.Sequential([sparse_l1_encoder, sparse_l1_decoder])
```

Compilar con mse.

```
sparse_l1_ae.compile(loss="mse", optimizer="nadam")
```

Historial

```
history = sparse_l1_ae.fit(X_train, X_train, epochs=10,
                           validation_data=(X_valid, X_valid))
```

Revisa que fue de tus resultados

```
plot_reconstructions(sparse_l1_ae)
plt.show(
```

1.32 Autoencoders Dispersos

Esta capa de regularización solo regresa sus entradas, pero agrega una pérdida de entrenamiento igual a la suma de los valores absolutos de sus entradas, obvio, solo durante el entrenamiento. Esta penalización va a obligar a la red neuronal a producir códigos cercanos a 0, pero como será penalizado si no reconstruye las entradas correctamente, va a tener que sacar al menos algunos valores que no sean 0.

Usando la norma L1 en vez de la norma L2 va a empujar a la red neuronal a preservar los códigos más importantes mientras eliminan los que no son necesitados para la imagen de entrada.

Preguntas