Autocodificadores

Clase 10

Dra. Wendy Aguilar

Modelos Generativos Profundos

UN ENFOQUE DESDE LA CREATIVIDAD COMPUTACIONAL

Ejecutar

autoencoder_fashion_mnist_version1.ipynb

Cuando termine el anterior:

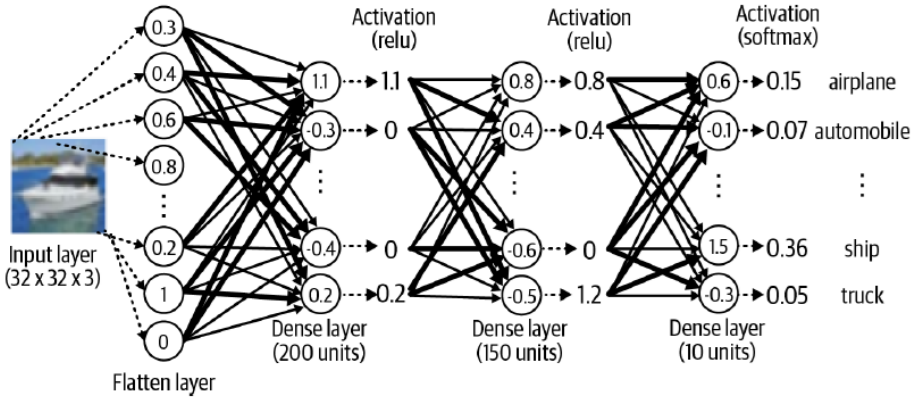
Ejecutar

autoencoder_fashion_mnist_version2.ipynb

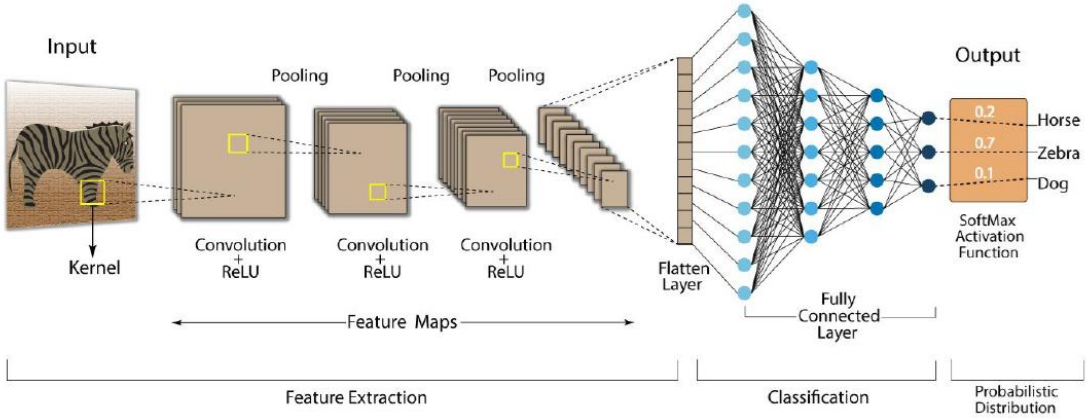
Hasta ahora

nuestras redes profundas han sido discriminativas: aprenden fronteras decisión entre clases

Perceptrones multicapa



Redes convolucionales



$$f_\theta : x \rightarrow y$$

Aprenden una función que predice etiquetas a partir de datos de entrada.

$$p(y|\mathbf{x})$$

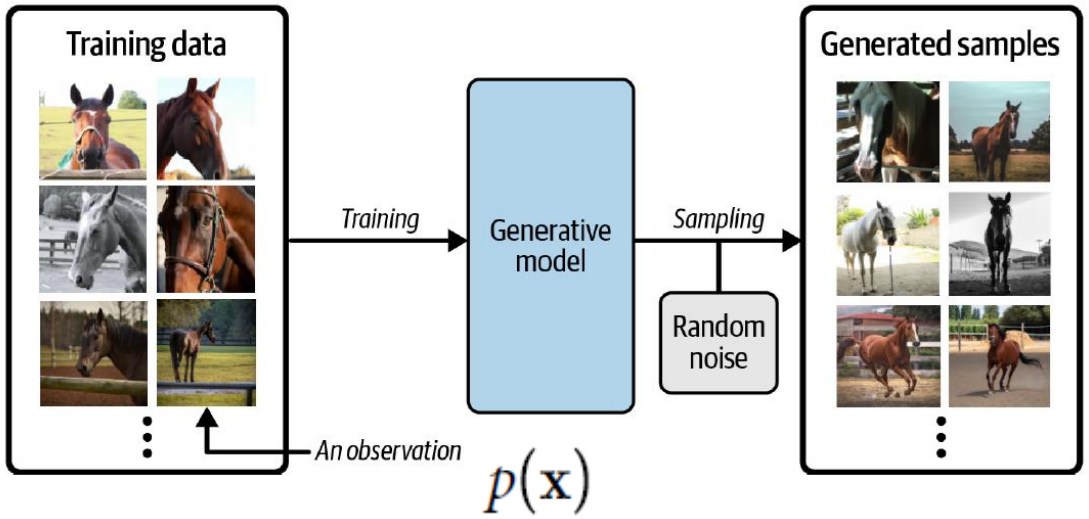
Probabilidad condicional

Tiene como objetivo modelar la probabilidad de una etiqueta *y* dada una observación *x*.

¿Qué pasaría si, en lugar de clasificar datos existentes quisiéramos crear datos nuevos que parezcan reales?

Cambio de paradigma

Modelos generativos



$p(\mathbf{x})$

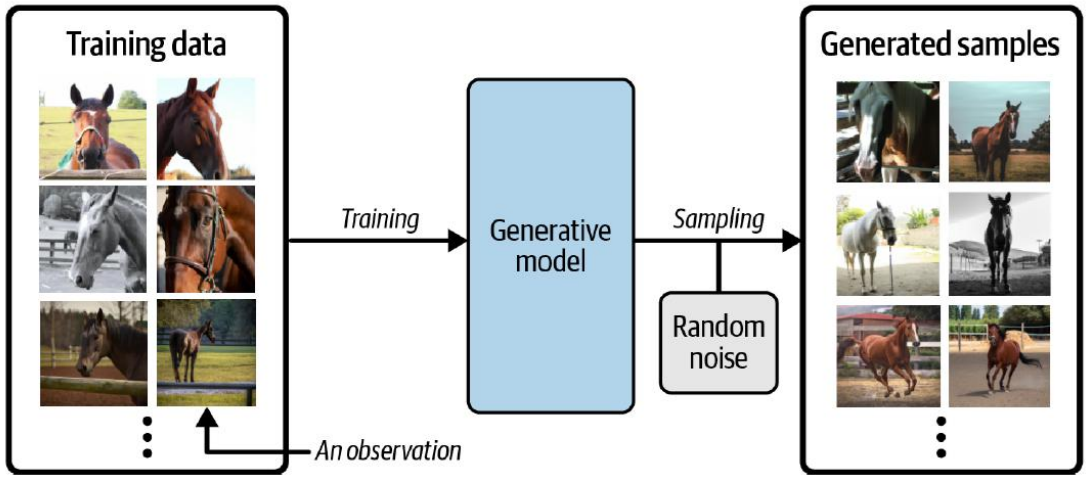
Distribución de probabilidad

Tiene como objetivo modelar la probabilidad de observar un ejemplo *x*.
Muestrear a partir de esta distribución nos permite generar nuevas observaciones.

¿Qué pasaría si, en lugar de clasificar datos existentes quisiéramos crear datos nuevos que parezcan reales?

Cambio de paradigma

Modelos generativos



- Los **mismos bloques de construcción** que ya conocen (capas densas, convoluciones, funciones de activación, optimización con gradiente) se usan también en modelos generativos.

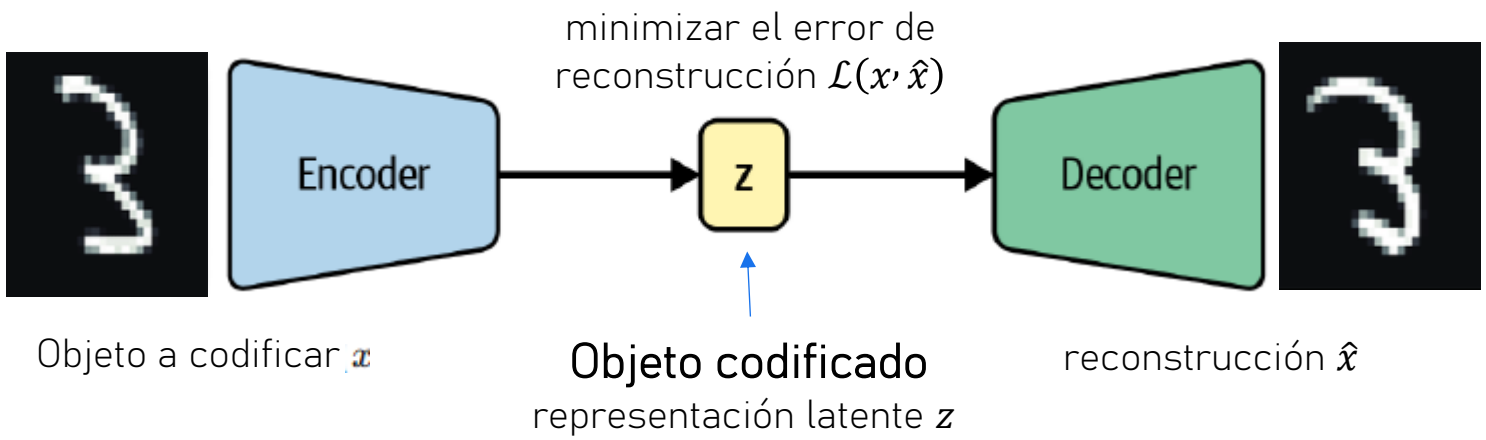
- Lo nuevo no está en la arquitectura básica, sino en cómo se plantea el **objetivo de entrenamiento**.

Una de las primeras propuesta de cómo **aproximar la distribución de los datos** con redes neuronales profundas son los

Autocodificadores

Son una red neuronal profunda que:

- aprende a codificar los datos en un **espacio comprimido** y a decodificarlos para reconstruirlos lo más fielmente posible,

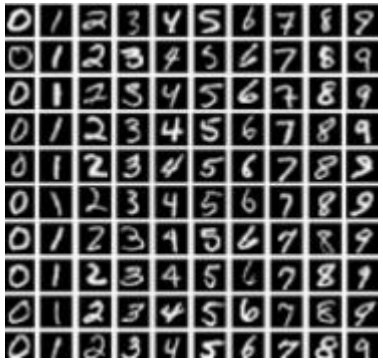


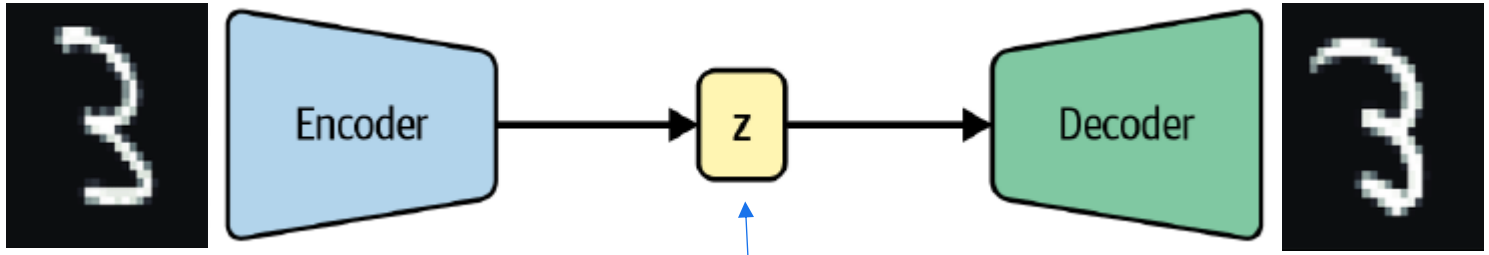
Una de las primeras propuesta de cómo **aproximar la distribución de los datos** con redes neuronales profundas son los

Autocodificadores

Ejemplo:

Supongamos que entrenamos un autocodificador con imágenes de dígitos de 28×28 píxeles (784 dimensiones).





Objeto a codificar x

Objeto codificado
representación latente z

reconstrucción $\hat{x}$

El encoder comprime cada imagen $\boldsymbol{x}$ a un vector latente $\boldsymbol{z}$ de 2 dimensiones (para poder visualizarlo):

$$x \in \mathbb{R}^{784} \longrightarrow z \in \mathbb{R}^2$$

Mapeo

Una imagen del dígito "7" podría mapearse a algo como: $\boldsymbol{z = [-1.23, 2.87]}$

Mientras que una imagen del dígito "3" podría mapearse a algo como: $\boldsymbol{z = [0.45, -0.98]}$

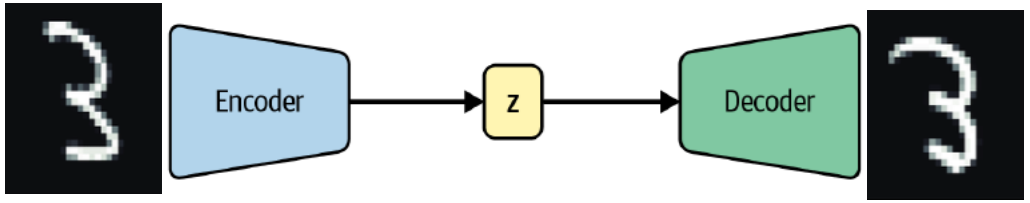
El decodificador toma $\boldsymbol{z}$ y trata de reconstruir la imagen original $\boldsymbol{x}$.

$$z \in \mathbb{R}^2 \longrightarrow x \in \mathbb{R}^{784}$$

Mapeo

Una de las primeras propuesta de cómo **aproximar la distribución de los datos** con redes neuronales profundas son los

Autocodificadores



El autocodificador define dos funciones:

$$z = f_\theta(x) \qquad \text{(encoder)}$$

$$\hat{x} = g_\phi(z) = g_\phi(f_\theta(x)) \qquad \text{(decoder)}$$

Se entrena minimizando el **error de reconstrucción**:

$$\mathcal{L}(\theta, \phi) = \frac{1}{N} \sum_{i=1}^{N} \ell\left(x^{(i)}, \hat{x}^{(i)}\right)$$

Puede ser MSE o cualquier otra función de pérdida punto a punto.

Para lograr baja pérdida, el par (f_θ, g_ϕ) debe:

- **Conservar la información esencial** de x en un vector z de baja dimensión.
- **Eliminar variaciones irrelevantes** que no afectan la reconstrucción.
- **Alinear la geometría de los datos en el espacio latente**, de modo que:
  - Ejemplos parecidos en el espacio de entrada terminen con z cercanos.
  - Así el decodificador pueda aprender una única función g_ϕ continua que los reconstruya.

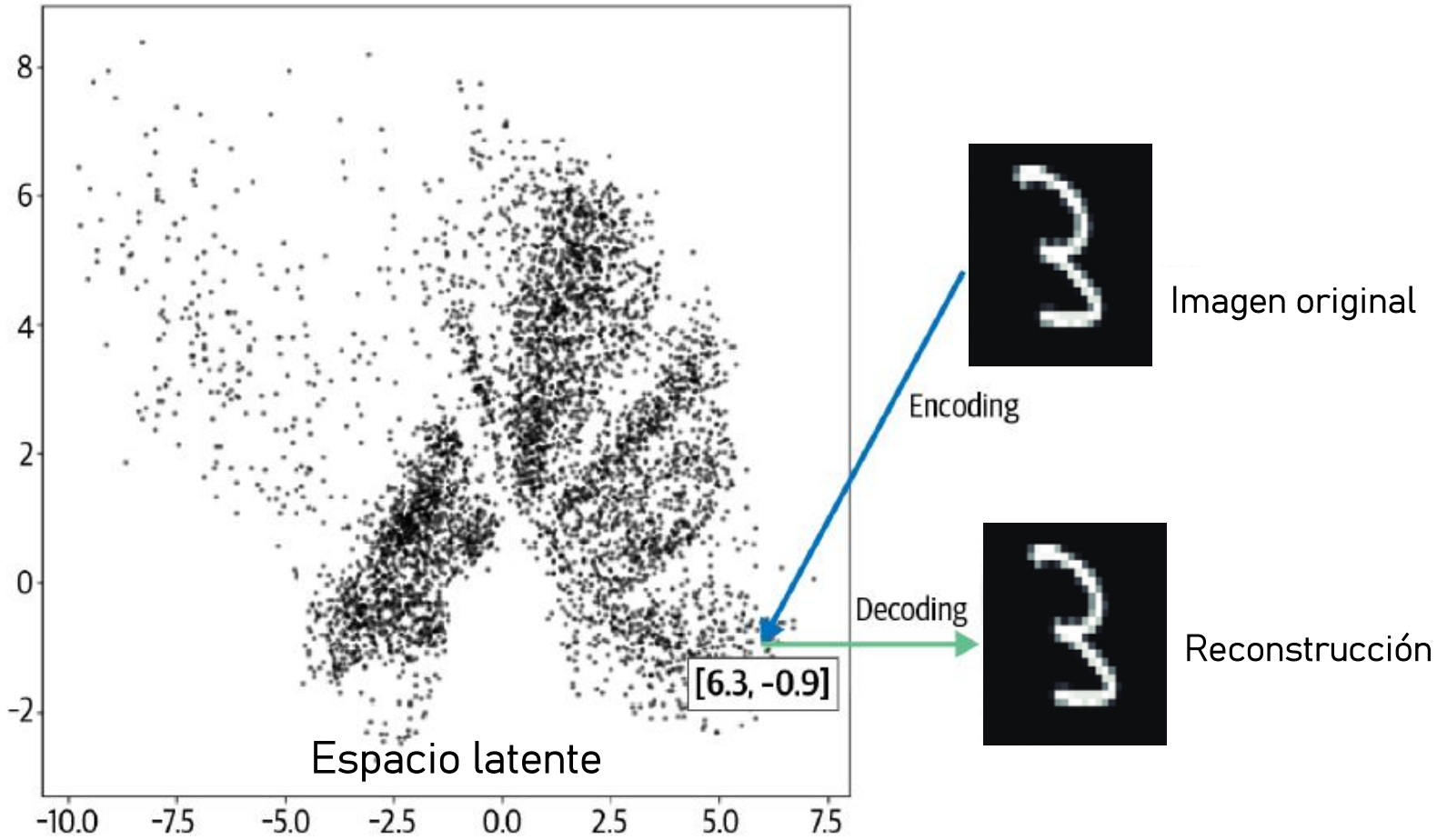
Porque, si dos entradas x_1 y x_2 muy similares quedaran con z muy alejados, el decodificador no podría reconstruir ambas correctamente con la misma función g_ϕ.

Una de las primeras propuesta de cómo **aproximar la distribución de los datos** con redes neuronales profundas son los

Autocodificadores

- Como resultado, cada dato de entrada queda "ubicado" en un punto de un nuevo espacio de características aprendidas.
  Ese conjunto estructurado de puntos $\boldsymbol{z}$ es lo que llamamos **espacio latente.**

- Está organizado de tal forma que ejemplos similares se encuentran en **regiones cercanas** del espacio latente,

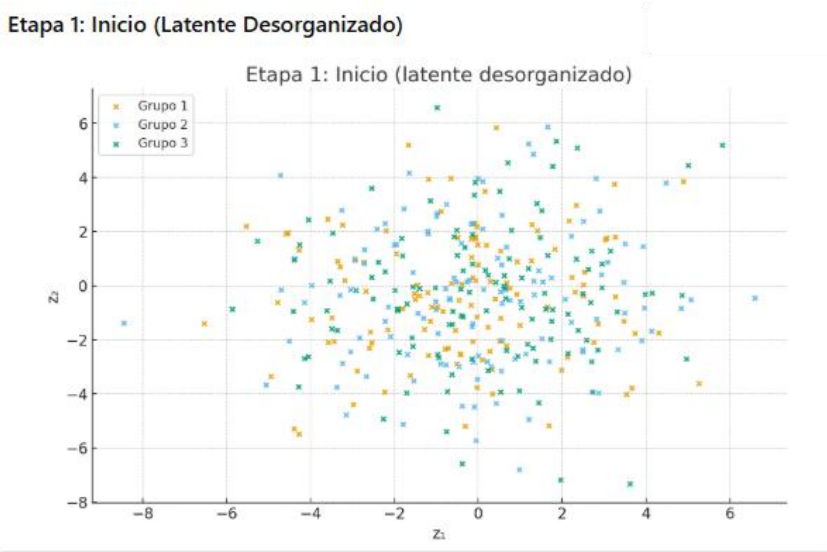


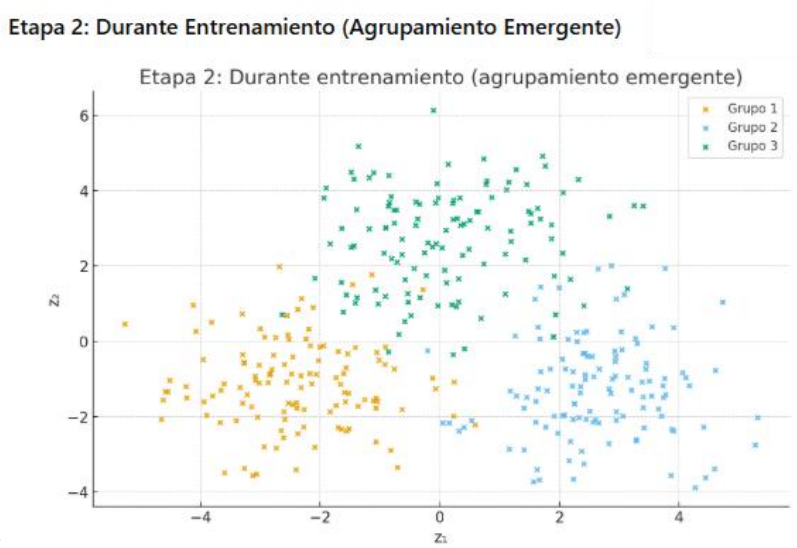
Es como un "mapa interno" que el modelo aprende para organizar los datos según sus características esenciales.

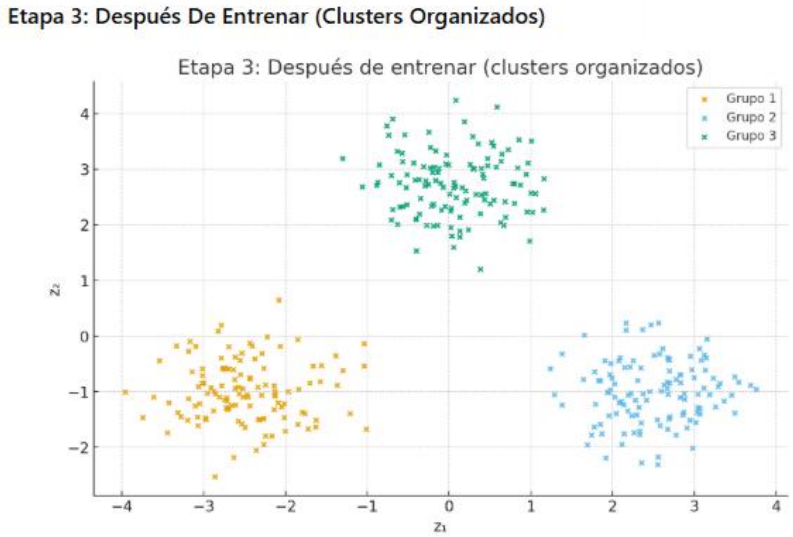
Una de las primeras propuesta de cómo **aproximar la distribución de los datos** con redes neuronales profundas son los

Autocodificadores

- Esta organización del espacio latente **no está programada explícitamente**,
- **emerge** como consecuencia de optimizar la reconstrucción.





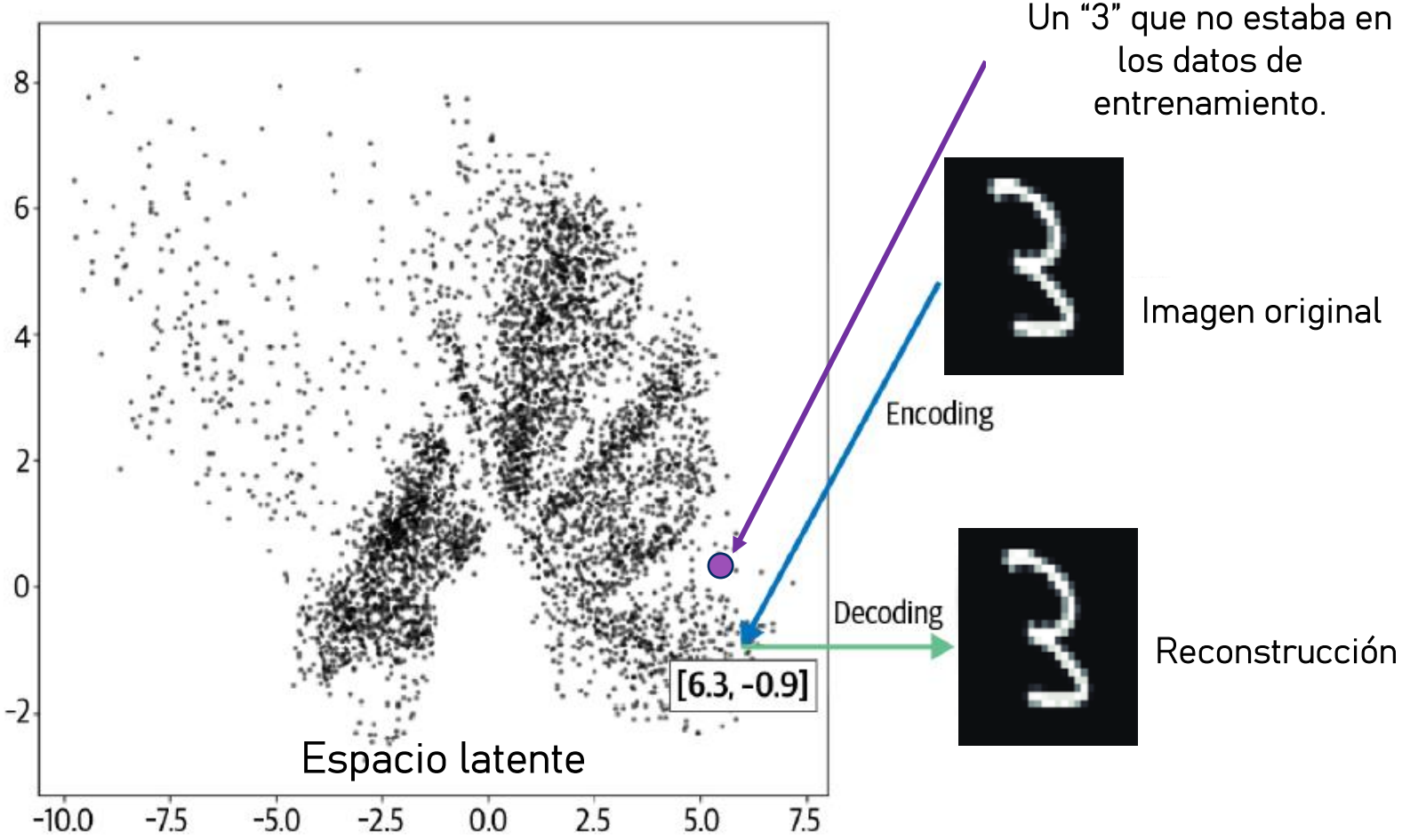


Una de las primeras propuesta de cómo **aproximar la distribución de los datos** con redes neuronales profundas son los

Autocodificadores

- Una vez aprendido este espacio, podemos:
  - **Tomar cualquier punto** z (incluso uno que no provenga de un dato real),
  - Pasarlo al **decodificador** para generar una muestra nueva $\hat{x} = g_\phi(z)$.

Esto convierte al **decodificador en un generador**, y es por eso que el espacio latente es **la puerta de entrada al modelado generativo**.



Construcción de un autocodificador de imágenes

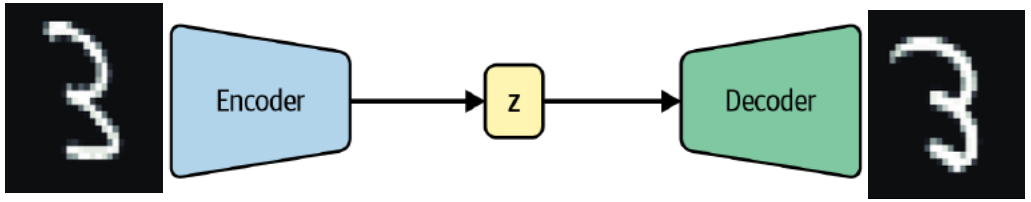
Fashion-MNIST dataset

28x28 pixeles

```
(x_train, y_train), (x_test, y_test) = datasets.fashion_mnist.load_data()


def preprocesa(imgs):
    """
    Normaliza y cambia el tamaño de las imágenes
    """
    imgs = imgs.astype("float32") / 255.0
    imgs = np.pad(imgs, ((0, 0), (2, 2), (2, 2)), constant_values=0.0)
    imgs = np.expand_dims(imgs, -1)
    return imgs


x_train = preprocesa(x_train)
x_test = preprocesa(x_test)
```

Codificador

Su tarea es transformar una entrada de alta dimensión (imagen) en una **representación latente comprimida** z que conserve la información esencial para reconstruirla.



Esto implica lograr **reducción de dimensionalidad + extracción de características**.

¿Qué tipo de capas conocemos que sean buenas haciendo esto?

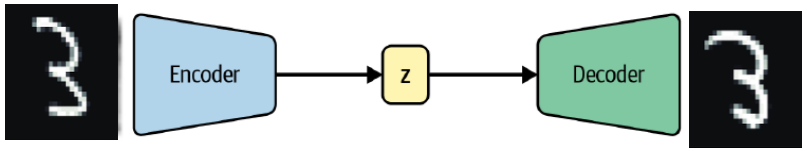
Codificador

Su tarea es transformar una entrada de alta dimensión (imagen) en una **representación latente comprimida** z que conserve la información esencial para reconstruirla.

Esto implica lograr **reducción de dimensionalidad + extracción de características**.

Tipo de datos de entrada	Arquitectura típica de codificador	Observaciones
Imágenes (2D)	CNNs (`Conv2D`)	Muy comunes porque explotan la estructura espacial local (vecindad de píxeles)
Audio, series temporales (1D)	`Conv1D` o RNN (LSTM / GRU)	Las convoluciones 1D capturan patrones locales en el tiempo
Texto (secuencias)	RNN, LSTM, GRU, Transformers (codificador tipo BERT)	Aprenden dependencias secuenciales y semánticas
Datos tabulares / vectores planos	MLPs (capas densas totalmente conectadas)	No hay estructura espacial que aprovechar, por lo que CNN no aporta ventaja
Nubes de puntos / datos 3D	Redes específicas (PointNet, GNNs, etc.)	Capturan estructura geométrica o relacional

Codificador



Componente	Qué aporta	Notas
Capas convolucionales (Conv2D)	Aprenden **filtros espaciales jerárquicos** para extraer patrones locales	Usualmente varias capas apiladas con filtros crecientes (32→64→128...)
Stride > 1 o MaxPooling	**Reducen la resolución espacial** para comprimir	Evitan usar imágenes gigantes en capas densas
Activaciones no lineales (ReLU, LeakyReLU)	Permiten aprender representaciones **no lineales y ricas**	Sin activación solo haría transformaciones lineales
BatchNormalization (opcional)	Estabiliza y acelera el entrenamiento	Especialmente útil en redes profundas
Cuello de botella (reducción progresiva)	Obliga a la red a **comprimir la información**	Sin cuello de botella, podría memorizar
Flatten + Dense(latente)	Colapsan el mapa 2D a un vector 1D latente	Esta es la representación z final
Tamaño del vector latente moderado	Controla el grado de compresión	Si es muy grande, memoriza; si es muy pequeño, pierde info

Codificador

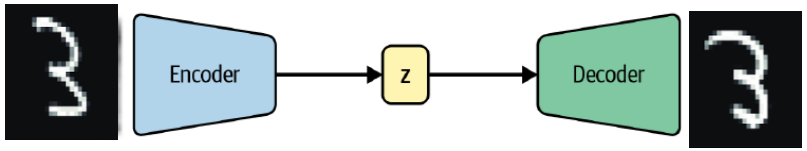
Layer (type)	Output shape	Param #
InputLayer	(None, 32, 32, 1)	0
Conv2D	(None, 16, 16, 32)	320
Conv2D	(None, 8, 8, 64)	18,496
Conv2D	(None, 4, 4, 128)	73,856
Flatten	(None, 2048)	0
Dense	(None, 2)	4,098

Total params	96,770
Trainable params	96,770
Non-trainable params	0

```
IMAGE_SIZE = 32
CHANNELS = 1
EMBEDDING_DIM = 2


# Codificador
encoder_input = layers.Input(
    shape=(IMAGE_SIZE, IMAGE_SIZE, CHANNELS), name="encoder_input"
)
x = layers.Conv2D(32, (3, 3), strides=2, activation="relu", padding="same")(
    encoder_input
)
x = layers.Conv2D(64, (3, 3), strides=2, activation="relu", padding="same")(x)
x = layers.Conv2D(128, (3, 3), strides=2, activation="relu", padding="same")(x)
shape_before_flattening = K.int_shape(x)[1:]  # El decodificador necesitará esta info

x = layers.Flatten()(x)
encoder_output = layers.Dense(EMBEDDING_DIM, name="encoder_output")(x)

encoder = models.Model(encoder_input, encoder_output)
```

Produce un **vector latente z con activación lineal (identidad)**, que es la representación comprimida aprendida por el encoder.

Decodificador

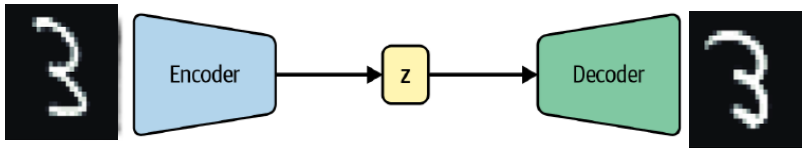


No hay una única manera de definir el decodificador.

Lo único necesario es que sea capaz de **tomar un vector latente** z (de dimensión `EMBEDDING_DIM`) y **reconstruir una imagen de la forma original (ancho, alto, canales)**.

1. **Expandir el vector latente**
   - Tomar `z` ($(batch, EMBEDDING\_DIM)$)
   - Pasarlo por una `Dense` para obtener el número de unidades igual a
     $$(\text{shape\_before\_flattening}) = h \times w \times c.$$
2. **Ir aumentando la resolución espacial**
   - Usar capas `Conv2DTranspose` o `UpSampling2D + Conv2D` para subir gradualmente hasta el tamaño original.
3. **Recuperar la estructura 2D inicial**
   - `Reshape` a $(batch, h, w, c)$.

Decodificador



No hay una única manera de definir el decodificador.

Lo único necesario es que sea capaz de **tomar un vector latente** *z* (de dimensión `EMBEDDING_DIM`) y **reconstruir una imagen de la forma original (ancho, alto, canales)**.

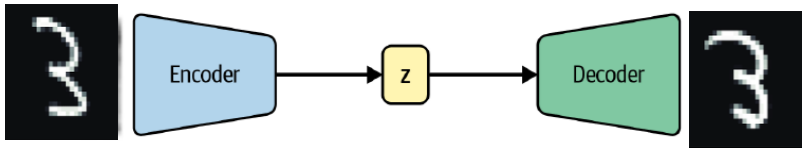
Mientras cumplas esas tres etapas, puedes variar:

- El **número de filtros** en cada capa (p. ej. 128 → 64 → 32 o 256 → 128 → 64).
- El **tipo de capa para subir tamaño** (`Conv2DTranspose` vs `UpSampling2D+Conv2D`).
- Usar o no **BatchNormalization**, **Dropout**, o distintas activaciones.
- Añadir más capas intermedias para mejorar la calidad de reconstrucción.

Esto significa que **hay muchas arquitecturas válidas** para el decodificador de un mismo codificador.

- Es común implementarlo como un espejo del codificador.

Decodificador



Codificador

Layer (type)	Output shape	Param #
InputLayer	(None, 32, 32, 1)	0
Conv2D	(None, 16, 16, 32)	320
Conv2D	(None, 8, 8, 64)	18,496
Conv2D	(None, 4, 4, 128)	73,856
Flatten	(None, 2048)	0
Dense	(None, 2)	4,098

Total params	96,770
Trainable params	96,770
Non-trainable params	0

La **forma del tensor** de **salida** del codificador será la forma del tensor de **entrada** del codificador.

Decodificador

Layer (type)	Output shape	Param #
InputLayer	(None, 2)	0
Dense	(None, 2048)	6,144
Reshape	(None, 4, 4, 128)	0
Conv2DTranspose	(None, 8, 8, 128)	147,584
Conv2DTranspose	(None, 16, 16, 64)	73,792
Conv2DTranspose	(None, 32, 32, 32)	18,464
Conv2D	(None, 32, 32, 1)	289

Total params	246,273
Trainable params	246,273
Non-trainable params	0

Decodificador

1. **Expandir el vector latente**
   - Tomar `z` $((batch, EMBEDDING\_DIM))$
   - Pasarlo por una `Dense` para obtener el número de unidades igual a $(\text{shape\_before\_flattening}) = h \times w \times c$.

Codificador

Layer (type)	Output shape	Param #
InputLayer	(None, 32, 32, 1)	0
Conv2D	(None, 16, 16, 32)	320
Conv2D	(None, 8, 8, 64)	18,496
Conv2D	(None, 4, 4, 128)	73,856
Flatten	(None, 2048)	0
Dense	(None, 2)	4,098

Total params	96,770
Trainable params	96,770
Non-trainable params	0

`Flatten` en el encoder aplana características 2D en un vector.

`Dense + Reshape` en el decoder revierte vector, recuperando la forma de un mapa de características 2D.

Decodificador

Layer (type)	Output shape	Param #
InputLayer	(None, 2)	0
Dense	(None, 2048)	6,144
Reshape	(None, 4, 4, 128)	0
Conv2DTranspose	(None, 8, 8, 128)	147,584
Conv2DTranspose	(None, 16, 16, 64)	73,792
Conv2DTranspose	(None, 32, 32, 32)	18,464
Conv2D	(None, 32, 32, 1)	289

Total params	246,273
Trainable params	246,273
Non-trainable params	0

Decodificador

2. **Ir aumentando la resolución espacial**
   - Usar capas `Conv2DTranspose` o `UpSampling2D + Conv2D` para subir gradualmente hasta el tamaño original.

Codificador

Layer (type)	Output shape	Param #
InputLayer	(None, 32, 32, 1)	0
Conv2D	(None, 16, 16, 32)	320
Conv2D	(None, 8, 8, 64)	18,496
Conv2D	(None, 4, 4, 128)	73,856
Flatten	(None, 2048)	0
Dense	(None, 2)	4,098

Total params	96,770
Trainable params	96,770
Non-trainable params	0

Se va reduciendo el tamaño (ancho y alto) de los mapas de característic as.

En vez de capas convolucionales ocupamos **capas convolucionales transpuestas**

Decodificador

Layer (type)	Output shape	Param #
InputLayer	(None, 2)	0
Dense	(None, 2048)	6,144
Reshape	(None, 4, 4, 128)	0
Conv2DTranspose	(None, 8, 8, 128)	147,584
Conv2DTranspose	(None, 16, 16, 64)	73,792
Conv2DTranspose	(None, 32, 32, 32)	18,464
Conv2D	(None, 32, 32, 1)	289

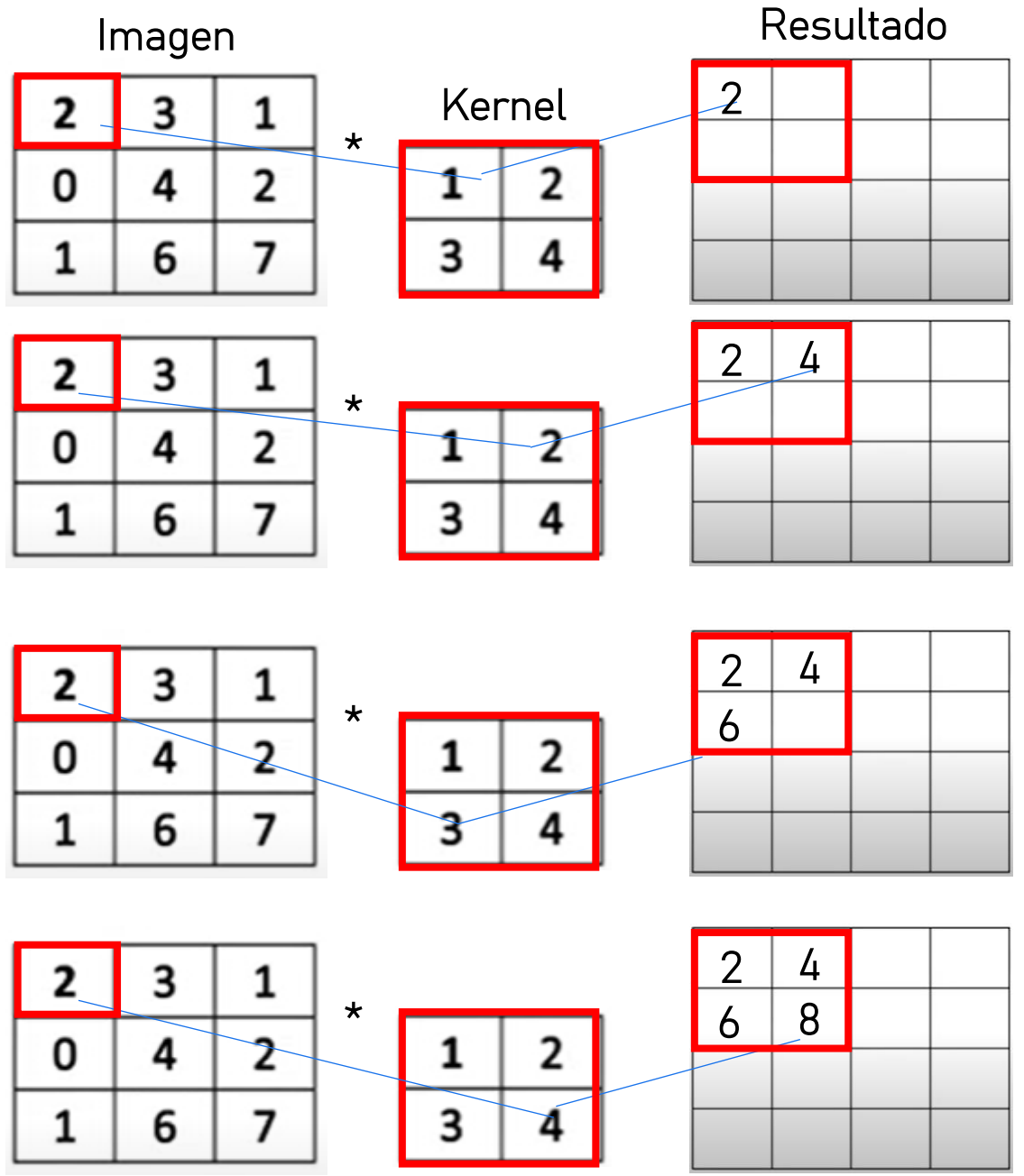
Total params	246,273
Trainable params	246,273
Non-trainable params	0

Se va aumentando el tamaño (ancho y alto) de los mapas de características.

Capas convolucionales transpuestas

Son la versión "inversa" de Conv2D:

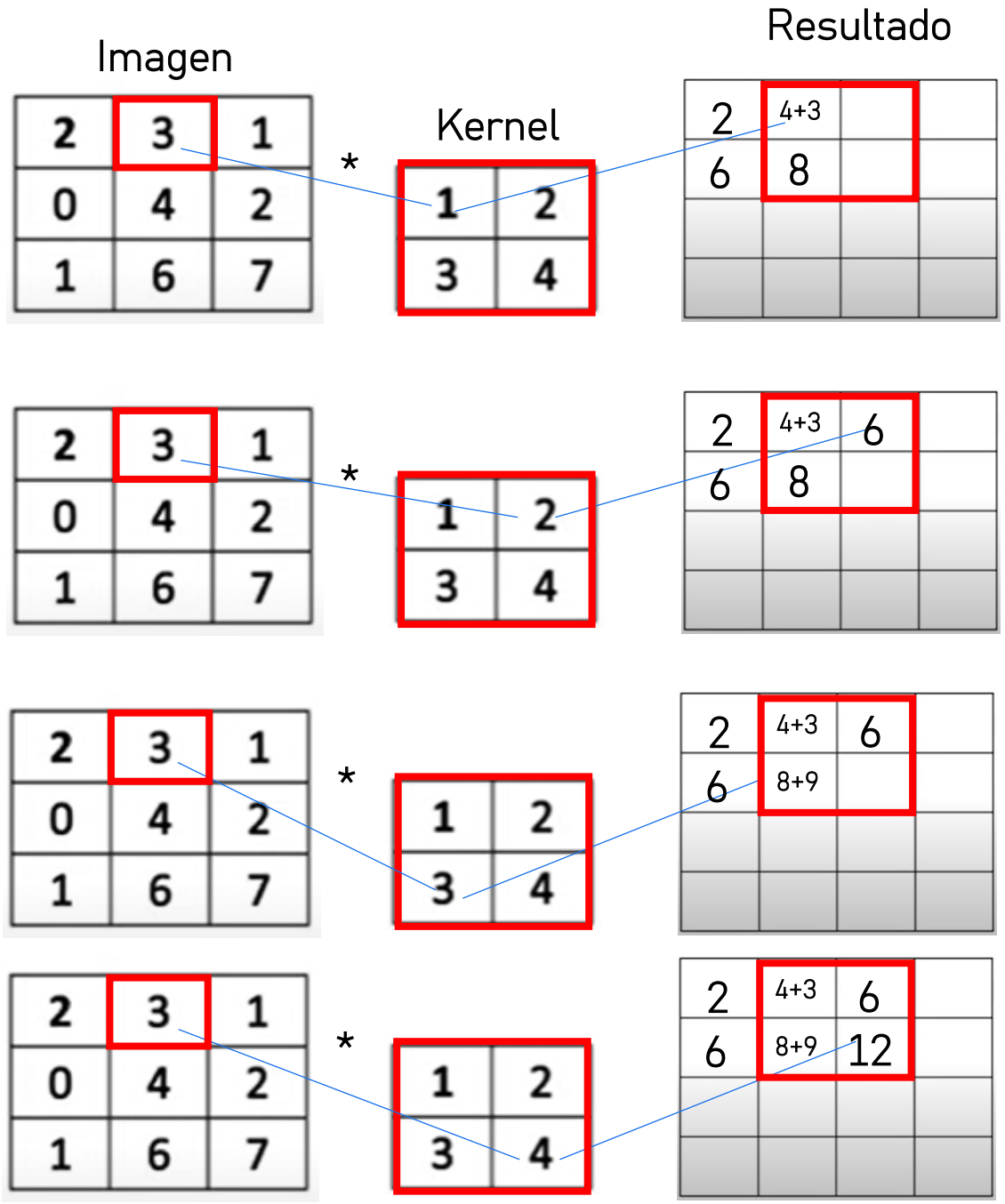
- Conv2D reduce la resolución y extrae características,
- Conv2DTranspose aumenta la resolución y **reconstituye** características espaciales a partir de representaciones comprimidas.



Capas convolucionales transpuestas

Son la versión "inversa" de Conv2D:

- Conv2D reduce la resolución y extrae características,
- Conv2DTranspose aumenta la resolución y **reconstituye** características espaciales a partir de representaciones comprimidas.

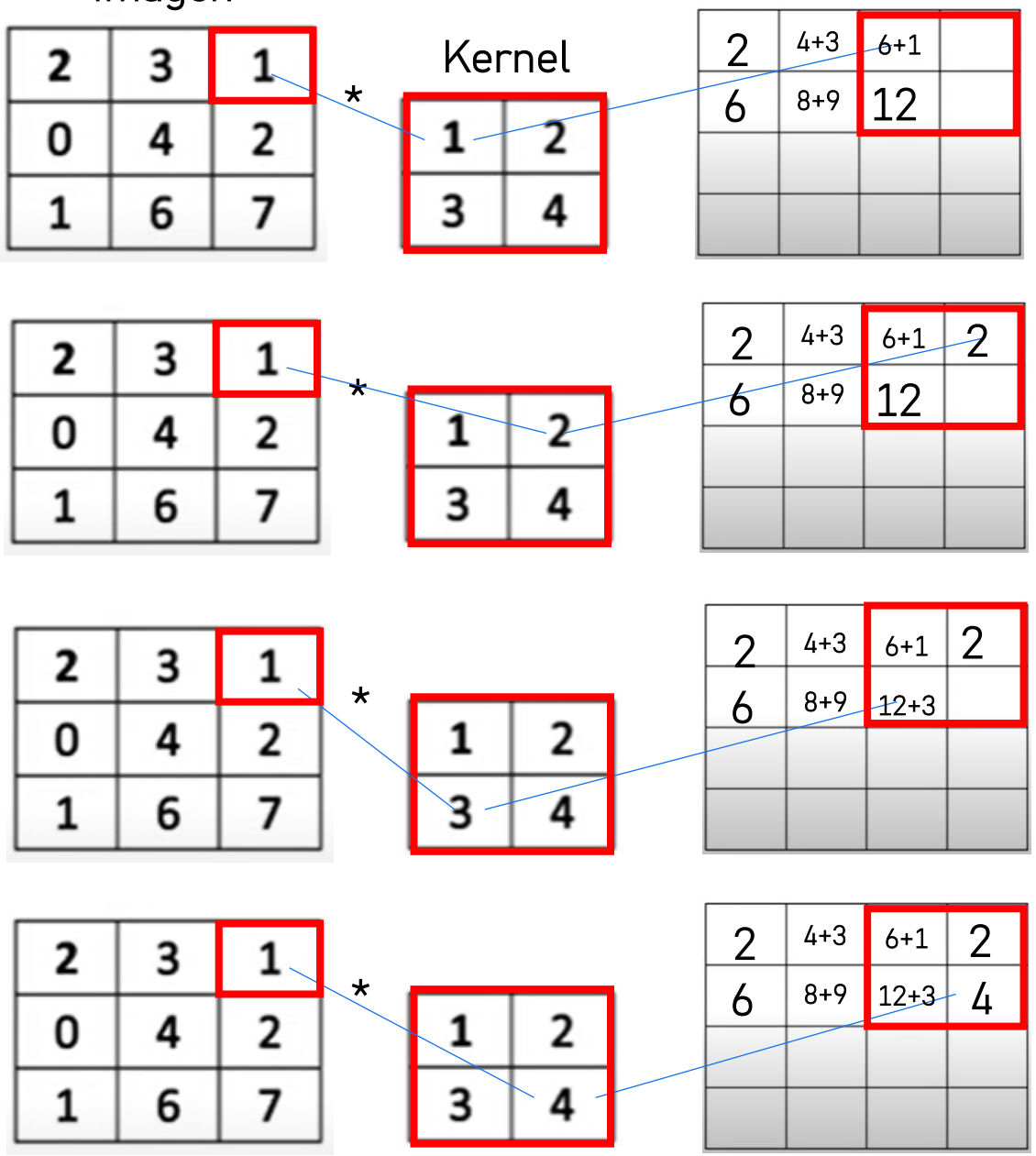


Capas convolucionales transpuestas

Imagen

Kernel

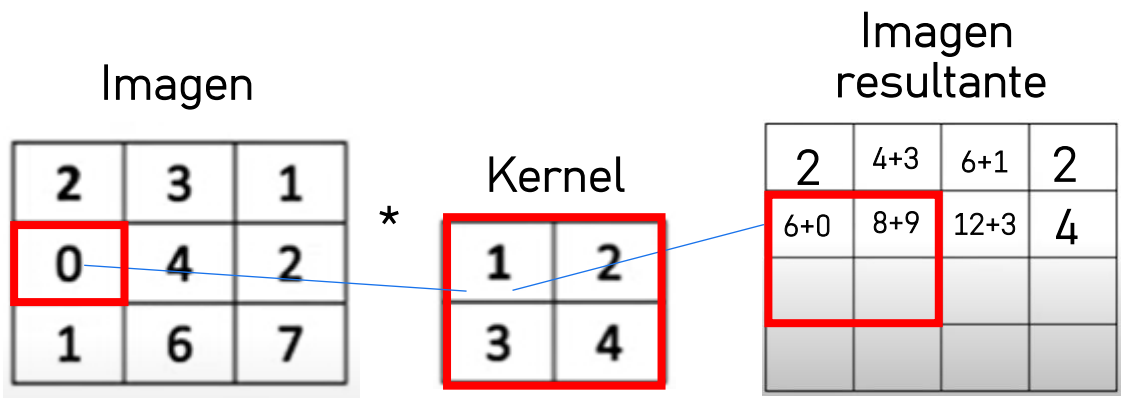
Resultado



Imagen

Kernel

Imagen resultante



•••¿Cómo queda la imagen resultante?

Capas convolucionales transpuestas

Imagen

2	3	1
0	4	2
1	6	7

*

Kernel

1	2
3	4

Resultado

2	4+3	6+1	
6	8+9	12	

2	3	1	
0	4	2	
1	6	7	

*

1	2
3	4

2	4+3	6+1	2
6	8+9	12	

2	3	1	
0	4	2	
1	6	7	

*

1	2
3	4

2	4+3	6+1	2
6	8+9	12+3	

2	3	1	
0	4	2	
1	6	7	

*

1	2
3	4

2	4+3	6+1	2
6	8+9	12+3	4

Imagen

2	3	1
0	4	2
1	6	7

*

Kernel

1	2
3	4

Imagen resultante

2	4+3	6+1	2
6+0	8+9	12+3	4

•••¿Cómo queda la imagen resultante?

2	7	7	2
6	21	25	8
1	20	41	22
3	22	45	28

Capas convolucionales transpuestas con stride = 1

stride=2

Imagen

2	3	1
0	4	2
1	6	7

*

Kernel

1	2
3	4

Imagen resultante

2	4				
6	8				

con stride = 2

Imagen

2	3	1
0	4	2
1	6	7

*

Kernel

1	2
3	4

resultante

2	4	3	6		
6	8	9	12		

Padding

- En la convolución el padding se hace en la entrada.
- En la transpuesta se hace en la salida.

Padding = 0

Imagen

2	3	1
0	4	2
1	6	7

*

Kernel

1	2
3	4

Imagen resultante

2	7	7	2
6	21	25	8
1	20	41	22
3	22	45	28

Padding =1

2	3	1
0	4	2
1	6	7

*

1	2
3	4

2	4	3	6	1	2
6	8	9	12	3	4
0	0	4	8	2	4
0	0	12	16	6	8
1	2	6	12	7	14
3	4	18	24	21	28

8	9	12	3
0	4	8	2
0	12	16	6
2	6	12	7

Descartamos el borde de 1 pixel.

Padding =2

2	3	1
0	4	2
1	6	7

*

1	2
3	4

4	8
12	16

Descartamos el borde de 2 pixeles.

Decodificador

2. **Ir aumentando la resolución espacial**
   - Usar capas `Conv2DTranspose` o `UpSampling2D + Conv2D` para subir gradualmente hasta el tamaño original.

Una alternativa para el decodificador:

Codificador

Layer (type)	Output shape	Param #
InputLayer	(None, 32, 32, 1)	0
Conv2D	(None, 16, 16, 32)	320
Conv2D	(None, 8, 8, 64)	18,496
Conv2D	(None, 4, 4, 128)	73,856
Flatten	(None, 2048)	0
Dense	(None, 2)	4,098

Total params	96,770
Trainable params	96,770
Non-trainable params	0

Usamos el mismo número de filtros, pero en orden inverso.

Decodificador

Layer (type)	Output shape	Param #
InputLayer	(None, 2)	0
Dense	(None, 2048)	6,144
Reshape	(None, 4, 4, 128)	0
Conv2DTranspose	(None, 8, 8, 128)	147,584
Conv2DTranspose	(None, 16, 16, 64)	73,792
Conv2DTranspose	(None, 32, 32, 32)	18,464
Conv2D	(None, 32, 32, 1)	289

Total params	246,273
Trainable params	246,273
Non-trainable params	0

Una alternativa para el decodificador:

Decodificador

Layer (type)	Output shape	Param #
InputLayer	(None, 2)	0
Dense	(None, 2048)	6,144
Reshape	(None, 4, 4, 128)	0
Conv2DTranspose	(None, 8, 8, 128)	147,584
Conv2DTranspose	(None, 16, 16, 64)	73,792
Conv2DTranspose	(None, 32, 32, 32)	18,464
Conv2D	(None, 32, 32, 1)	289

Total params	246,273
Trainable params	246,273
Non-trainable params	0

```
# Decoder
decoder_input = layers.Input(shape=(EMBEDDING_DIM,), name="decoder_input")
x = layers.Dense(np.prod(shape_before_flattening))(decoder_input)
x = layers.Reshape(shape_before_flattening)(x)
x = layers.Conv2DTranspose(
    128, (3, 3), strides=2, activation="relu", padding="same"
)(x)
x = layers.Conv2DTranspose(
    64, (3, 3), strides=2, activation="relu", padding="same"
)(x)
x = layers.Conv2DTranspose(
    32, (3, 3), strides=2, activation="relu", padding="same"
)(x)
decoder_output = layers.Conv2D(
    CHANNELS,
    (3, 3),
    strides=1,
    activation="sigmoid",
    padding="same",
    name="decoder_output",
)(x)

decoder = models.Model(decoder_input, decoder_output)
```

Uniendo el codificador y el decodificador

```
autoencoder = models.Model(
    encoder_input, decoder(encoder_output)
)
```

Autoencoder_fashion_mnist_version1.ipynb

Compilando el autoencoder


```
autoencoder.compile(optimizer="adam", loss="binary_crossentropy"
```

Entrenando el autoencoder

```
checkpoint_path = "/content/drive/MyDrive/checkpoints/model.keras"
log_dir = "/content/drive/MyDrive/logs"

# Crea un checkpint para salvar el modelo
model_checkpoint_callback = callbacks.ModelCheckpoint(
    filepath=checkpoint_path,
    save_weights_only=False,
    save_freq="epoch",
    monitor="loss",
    mode="min",
    save_best_only=True,
    verbose=0,
)

tensorboard_callback = callbacks.TensorBoard(log_dir=log_dir)
```

```
autoencoder.fit(
    x_train,
    x_train,
    epochs=EPOCHS,
    batch_size=BATCH_SIZE,
    shuffle=True,
    validation_data=(x_test, x_test),
    callbacks=[model_checkpoint_callback, tensorboard_callback],
)
```

Entrenamiento

```
Epoch 1/3
600/600 ━━━━━━━━━━━━━━━━━━━━ 15s 12ms/step - loss: 0.3644 - val_loss: 0.2619
Epoch 2/3
600/600 ━━━━━━━━━━━━━━━━━━━━ 13s 9ms/step - loss: 0.2589 - val_loss: 0.2561
Epoch 3/3
600/600 ━━━━━━━━━━━━━━━━━━━━ 5s 8ms/step - loss: 0.2539 - val_loss: 0.2536
```

Decodificador

Una segunda alternativa para el decodificador:

```
# Decodificador alternativo
decoder_input = layers.Input(shape=(EMBEDDING_DIM,), name="decoder_input_alt")

# Expansión del vector latente
x = layers.Dense(np.prod(shape_before_flattening))(decoder_input)
x = layers.Reshape(shape_before_flattening)(x)

# Bloque 1
x = layers.UpSampling2D(size=(2, 2))(x)
x = layers.Conv2D(128, (3, 3), padding="same")(x)
x = layers.BatchNormalization()(x)
x = layers.LeakyReLU(alpha=0.2)(x)

# Bloque 2
x = layers.UpSampling2D(size=(2, 2))(x)
x = layers.Conv2D(64, (3, 3), padding="same")(x)
x = layers.BatchNormalization()(x)
x = layers.LeakyReLU(alpha=0.2)(x)

# Bloque 3
x = layers.UpSampling2D(size=(2, 2))(x)
x = layers.Conv2D(32, (3, 3), padding="same")(x)
x = layers.BatchNormalization()(x)
x = layers.LeakyReLU(alpha=0.2)(x)

# Capa de salida
decoder_output = layers.Conv2D(
    CHANNELS,
    (3, 3),
    activation="sigmoid",
    padding="same",
    name="decoder_output"
)(x)

decoder = models.Model(decoder_input, decoder_output)
```

autoencoder_fashion_mnist_version2.ipynb

Google colab

Layer (type)	Output Shape	Param #
decoder_input_alt (InputLayer)	(None, 2)	0
dense_2 (Dense)	(None, 2048)	6,144
reshape_2 (Reshape)	(None, 4, 4, 128)	0
up_sampling2d_6 (UpSampling2D)	(None, 8, 8, 128)	0
conv2d_9 (Conv2D)	(None, 8, 8, 128)	147,584
batch_normalization_6 (BatchNormalization)	(None, 8, 8, 128)	512
leaky_re_lu_6 (LeakyReLU)	(None, 8, 8, 128)	0
up_sampling2d_7 (UpSampling2D)	(None, 16, 16, 128)	0
conv2d_10 (Conv2D)	(None, 16, 16, 64)	73,792
batch_normalization_7 (BatchNormalization)	(None, 16, 16, 64)	256
leaky_re_lu_7 (LeakyReLU)	(None, 16, 16, 64)	0
up_sampling2d_8 (UpSampling2D)	(None, 32, 32, 64)	0
conv2d_11 (Conv2D)	(None, 32, 32, 32)	18,464
batch_normalization_8 (BatchNormalization)	(None, 32, 32, 32)	128
leaky_re_lu_8 (LeakyReLU)	(None, 32, 32, 32)	0
decoder_output (Conv2D)	(None, 32, 32, 1)	289

Total params: 247,169 (965.50 KB)
Trainable params: 246,721 (963.75 KB)
Non-trainable params: 448 (1.75 KB)

Decodificador

Versión 1

```
Epoch 1/3
600/600 ━━━━━━━━━━━━━━━━━━━━ 15s 12ms/step - loss: 0.3644 - val_loss: 0.2619
Epoch 2/3
600/600 ━━━━━━━━━━━━━━━━━━━━ 13s 9ms/step - loss: 0.2589 - val_loss: 0.2561
Epoch 3/3
600/600 ━━━━━━━━━━━━━━━━━━━━ 5s 8ms/step - loss: 0.2539 - val_loss: 0.2536
```

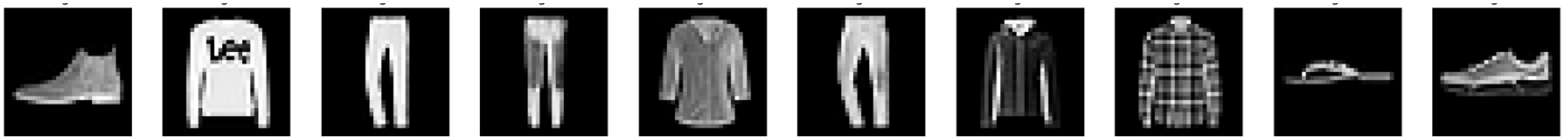
Versión 2

```
Epoch 1/3
600/600 ━━━━━━━━━━━━━━━━━━━━ 19s 17ms/step - loss: 0.2961 - val_loss: 0.2795
Epoch 2/3
600/600 ━━━━━━━━━━━━━━━━━━━━ 11s 12ms/step - loss: 0.2570 - val_loss: 0.2567
Epoch 3/3
600/600 ━━━━━━━━━━━━━━━━━━━━ 7s 12ms/step - loss: 0.2543 - val_loss: 0.2542
<keras.src.callbacks.history.History at 0x7958c0343620>
```

Reconstruyendo imágenes

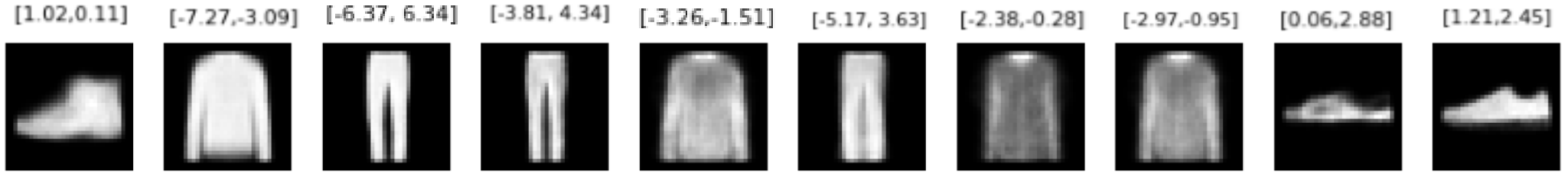
```
n_to_predict = 5000
example_images = x_test[:n_to_predict]
example_labels = y_test[:n_to_predict]

predictions = autoencoder.predict(example_images)
```

Originales



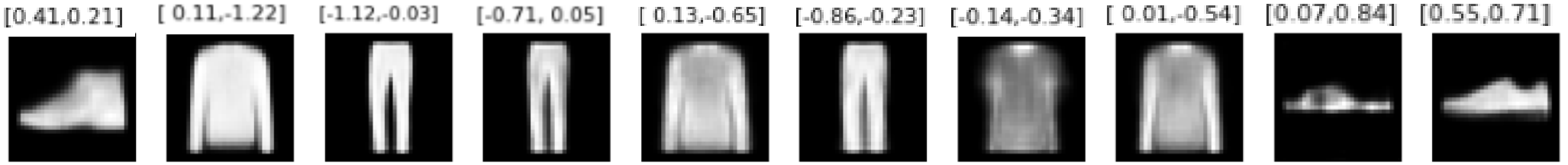
Versión 1

Reconstruidas



Versión 2

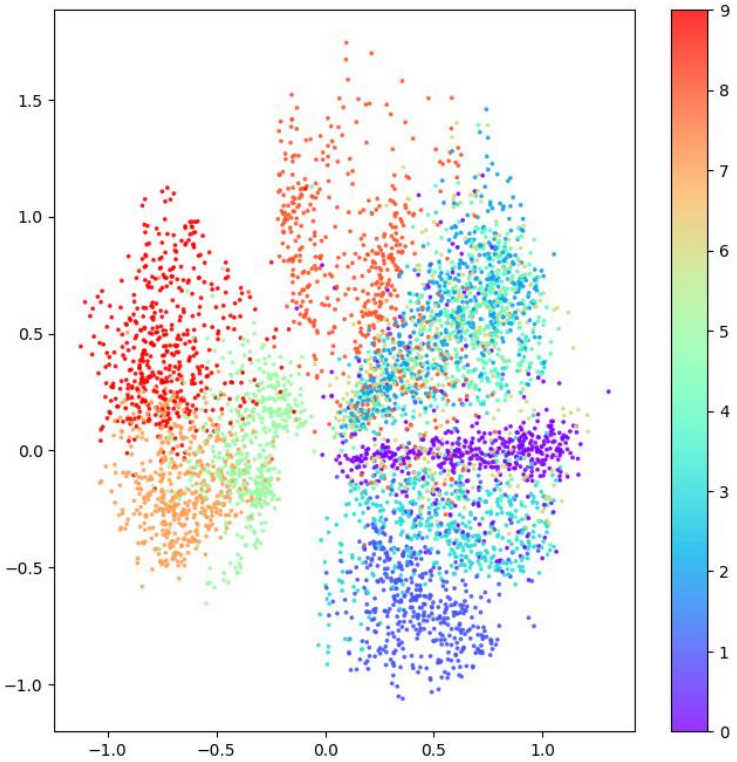
Reconstruidas



Visualizando el espacio latente

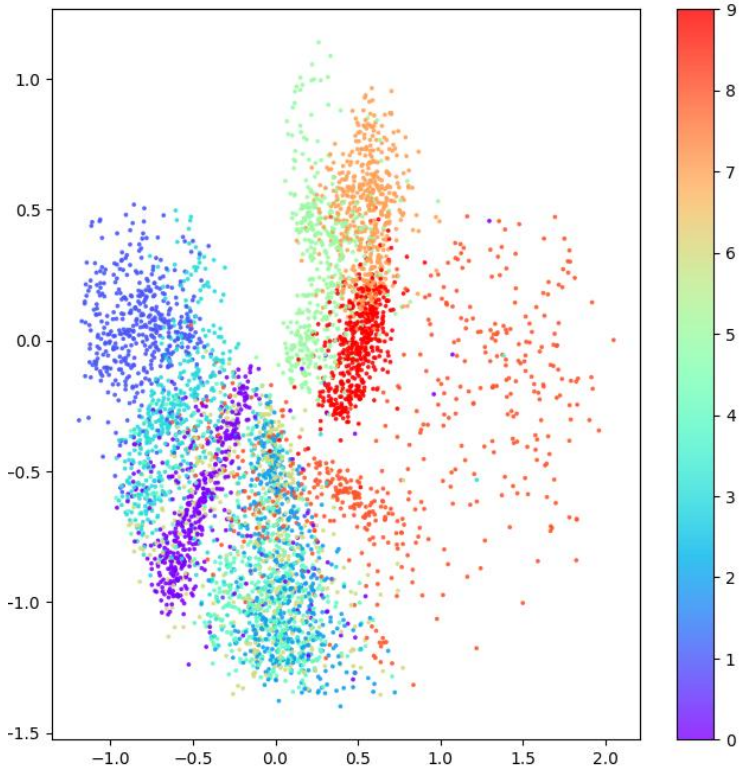
```
embeddings = encoder.predict(example_images)

plt.figure(figsize=(8, 8))
plt.scatter(embeddings[:, 0], embeddings[:, 1], c="black", alpha=0.5, s=3)
plt.show()
```

Versión 1



ID	Clothing label
0	T-shirt/top
1	Trouser
2	Pullover
3	Dress
4	Coat
5	Sandal
6	Shirt
7	Sneaker
8	Bag
9	Ankle boot

Versión 2

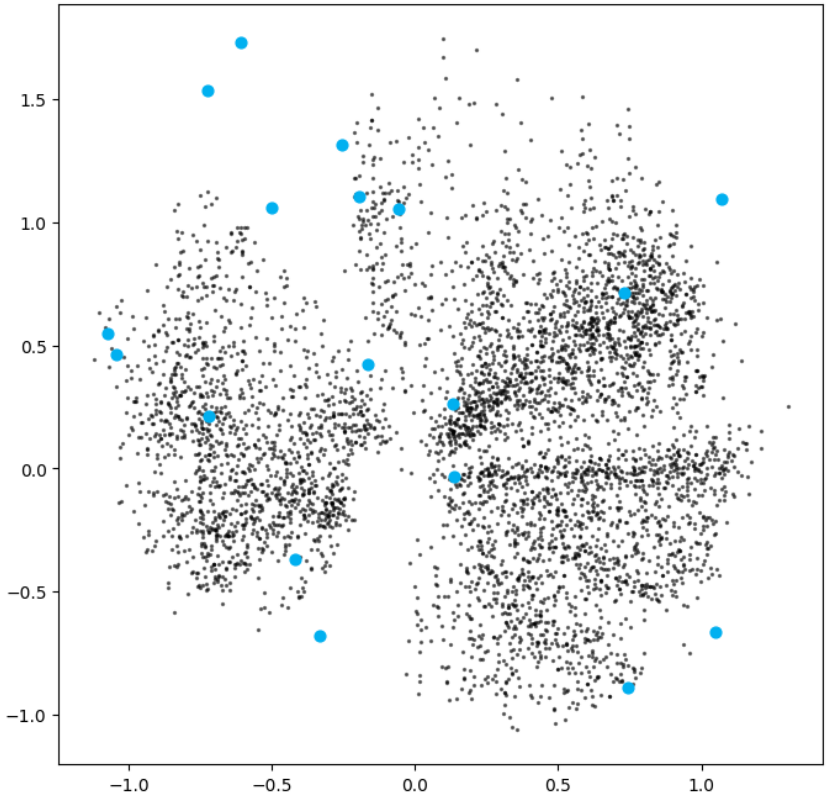


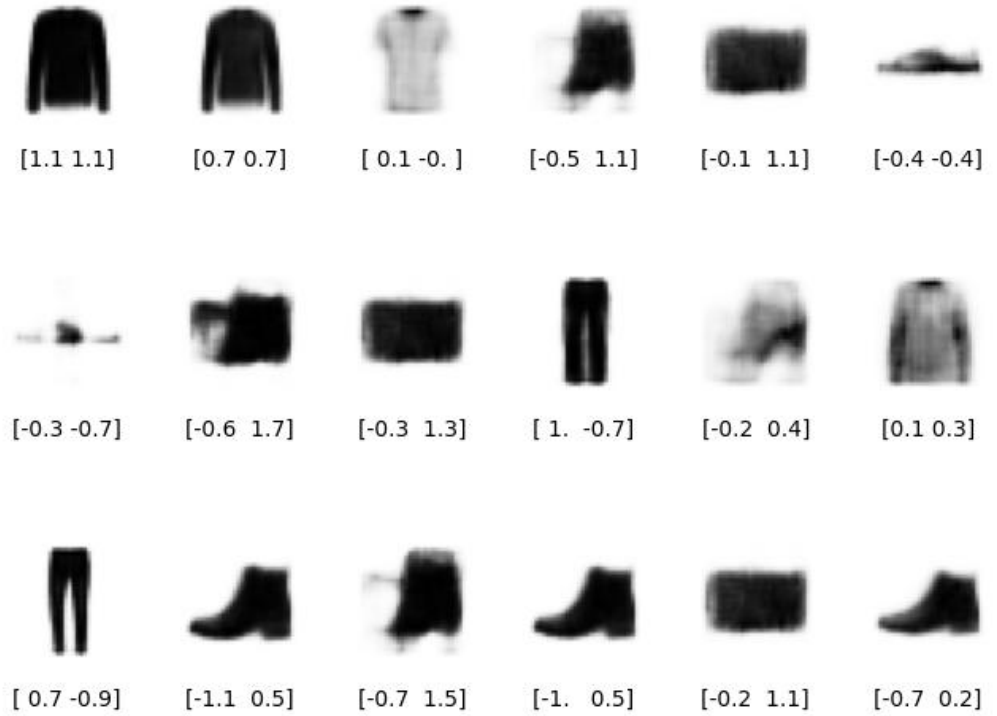
Aunque nunca se le presentaron a la red las etiquetas de las imágenes, el autocodificador ha agrupado aquellas que se parecen en una misma parte del espacio latente.

Generando nuevas imágenes

- Podemos generar imágenes novedosas muestreando algunos puntos en el espacio latente y utilizando el decodificador para convertirlos de nuevo al espacio de píxeles.

```
mins, maxs = np.min(embeddings, axis=0), np.max(embeddings, axis=0)
sample = np.random.uniform(mins, maxs, size=(18, 2))
reconstructions = decoder.predict(sample)
```



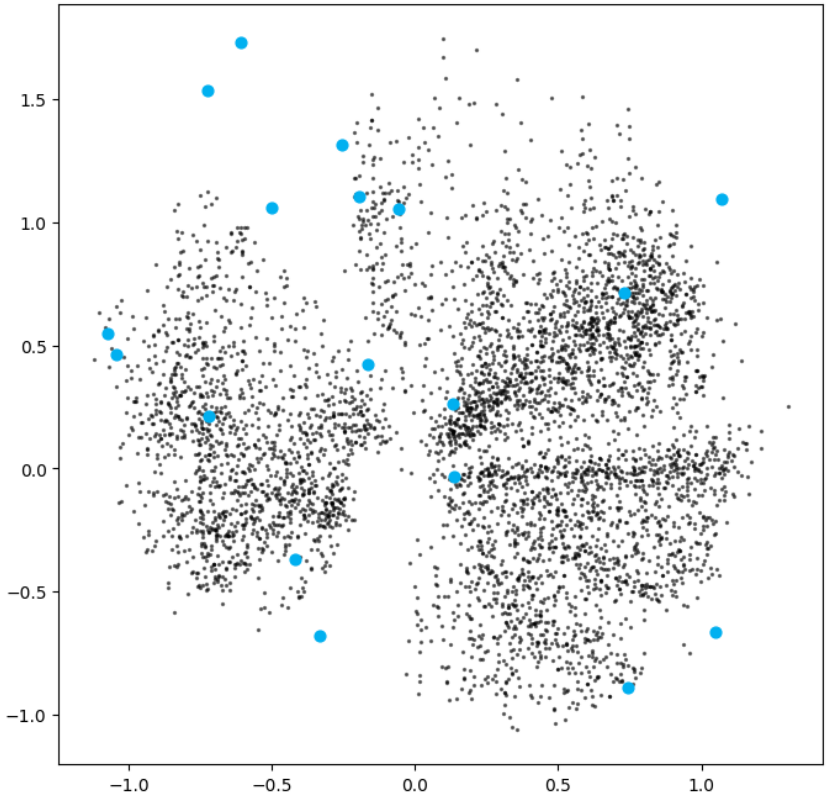


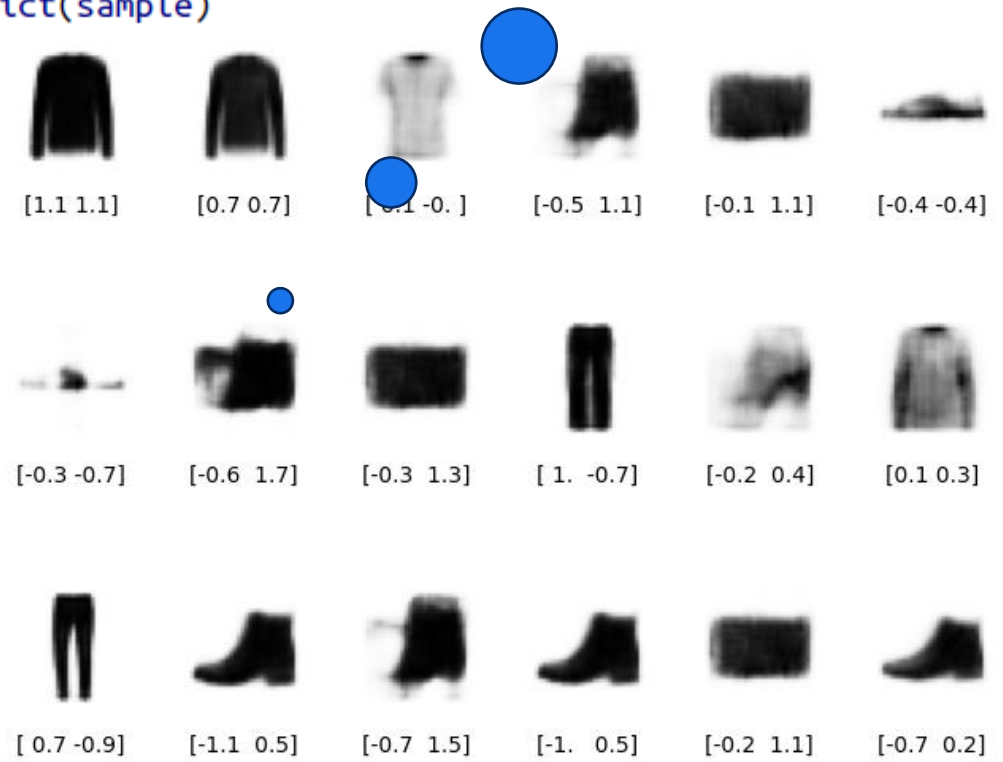
Generando nuevas imág

¿Por qué algunas imágenes generadas son más realistas que otras?

- Podemos generar imágenes novedosas muestreando algunos p
  utilizando el decodificador para convertirlos de nuevo al espac

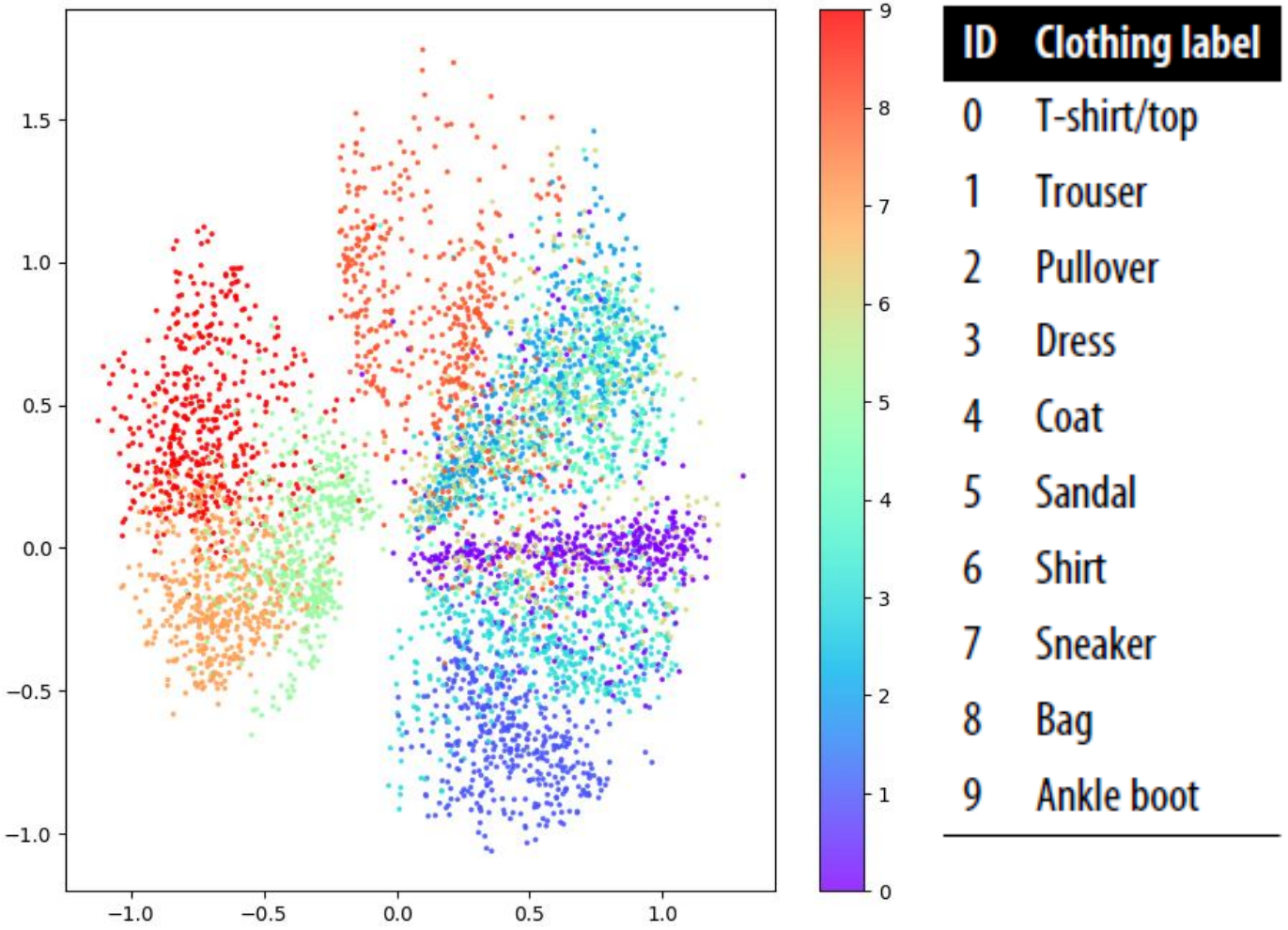
```
mins, maxs = np.min(embeddings, axis=0), np.max(embeddings,
sample = np.random.uniform(mins, maxs, size=(18, 2))
reconstructions = decoder.predict(sample)
```





Para contestarlo, hagamos algunas observaciones acerca de la distribución general de los puntos en el espacio latente.

Distribución general de los puntos en el espacio latente



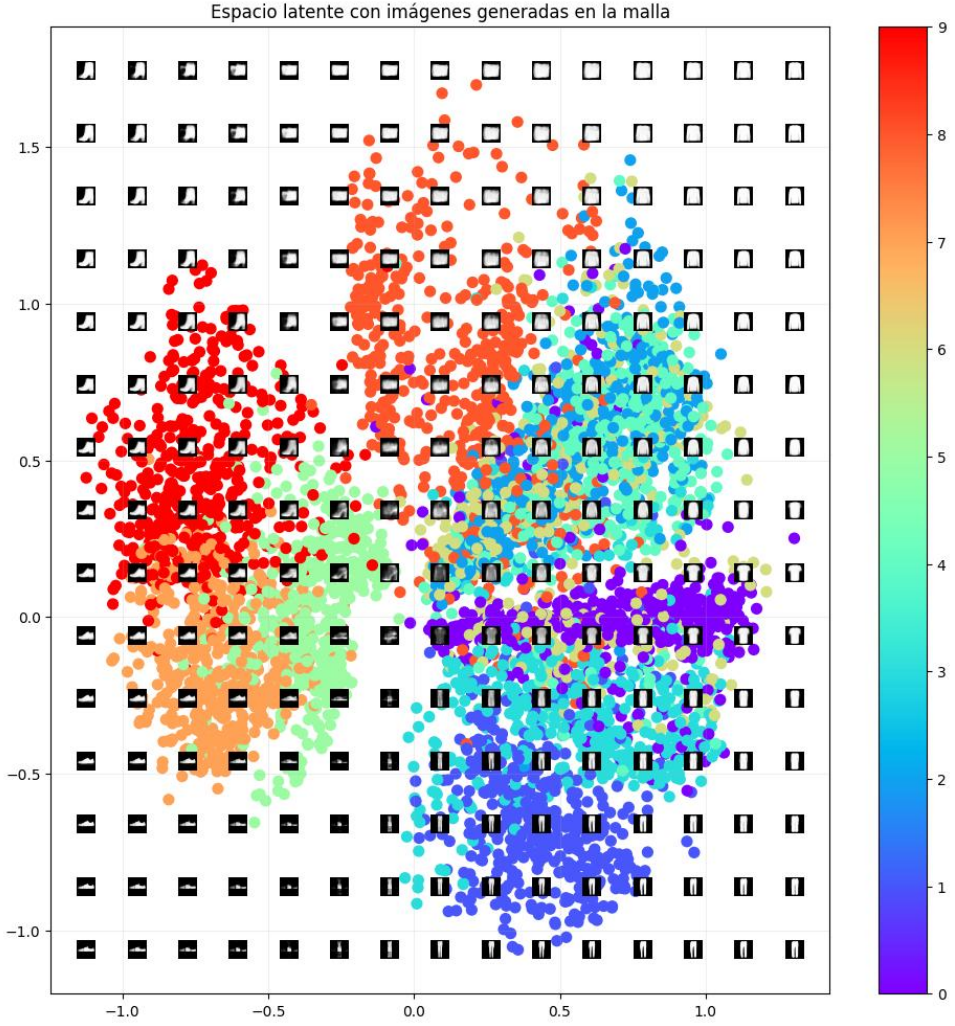
ID	Clothing label
0	T-shirt/top
1	Trouser
2	Pullover
3	Dress
4	Coat
5	Sandal
6	Shirt
7	Sneaker
8	Bag
9	Ankle boot

- Algunas prendas están representadas en un área muy pequeña mientras que otras están representadas en áreas mucho más grandes.
- Esta distribución no es simétrica con respecto al punto (0,0), ni está acotada.
- Existen grandes huecos entre colores que contienen pocos puntos.

Esto causa que el muestrear del espacio latente sea una tarea de mucho reto.

Generando nuevas imágenes

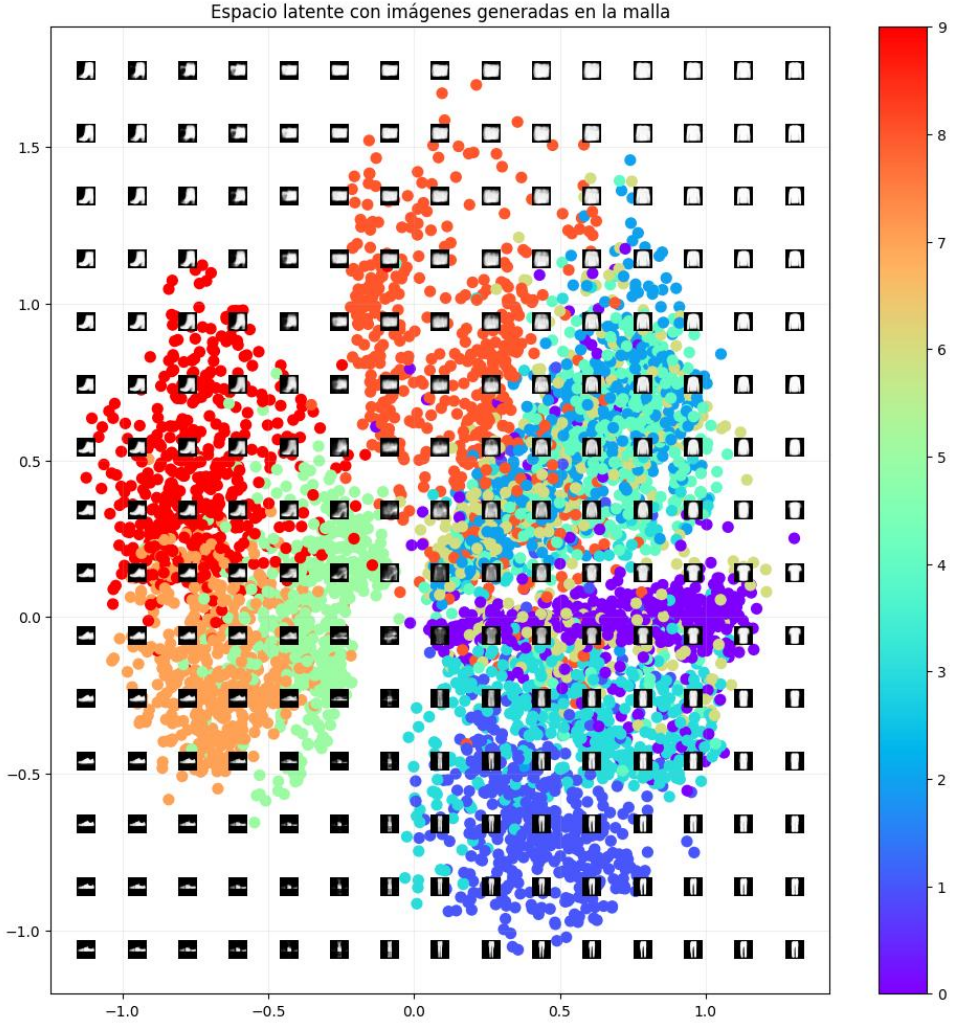
Si encimamos el espacio latente con imágenes de puntos decodificados en una malla, entonces podemos comenzar a entender por qué el decodificador no siempre puede generar imágenes satisfactorias:



1. Si elegimos puntos de manera uniforme en un espacio acotado que definimos, es más probable que muestreamos algo que se decodifique como una bolsa (ID 8) que como una bota tobillera (ID 9), porque la parte del espacio latente asignada a las bolsas (naranja) es más grande que el área de las botas tobilleras (rojo).

Generando nuevas imágenes

Si encimamos el espacio latente con imágenes de puntos decodificados en una malla, entonces podemos comenzar a entender por qué el decodificador no siempre puede generar imágenes satisfactorias:



2. No es evidente cómo deberíamos elegir un punto aleatorio en el espacio latente, ya que la distribución de estos puntos no está definida.

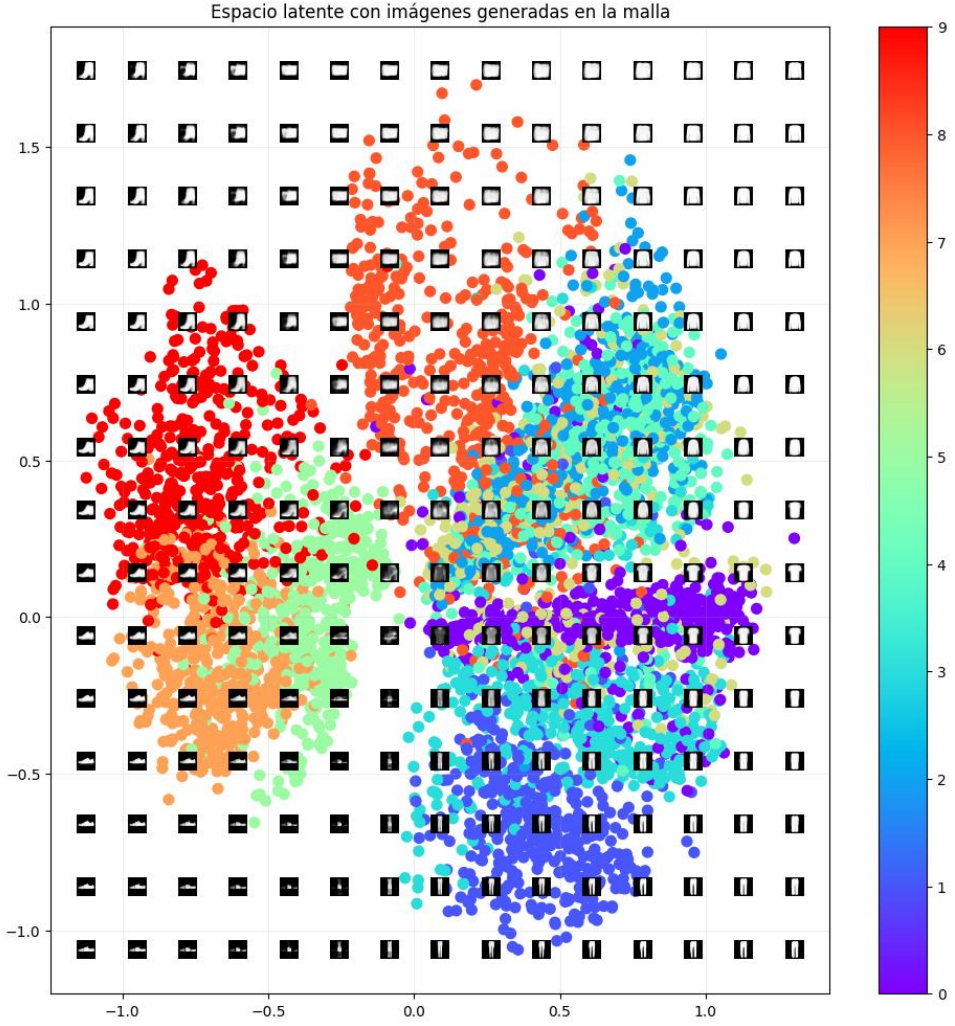
Técnicamente, podríamos justificar la elección de cualquier punto en el plano 2D.

¡Ni siquiera está garantizado que los puntos estén centrados en (0,0)!

Esto hace que el muestreo desde nuestro espacio latente sea problemático.

Generando nuevas imágenes

Si encimamos el espacio latente con imágenes de puntos decodificados en una malla, entonces podemos comenzar a entender por qué el decodificador no siempre puede generar imágenes satisfactorias:



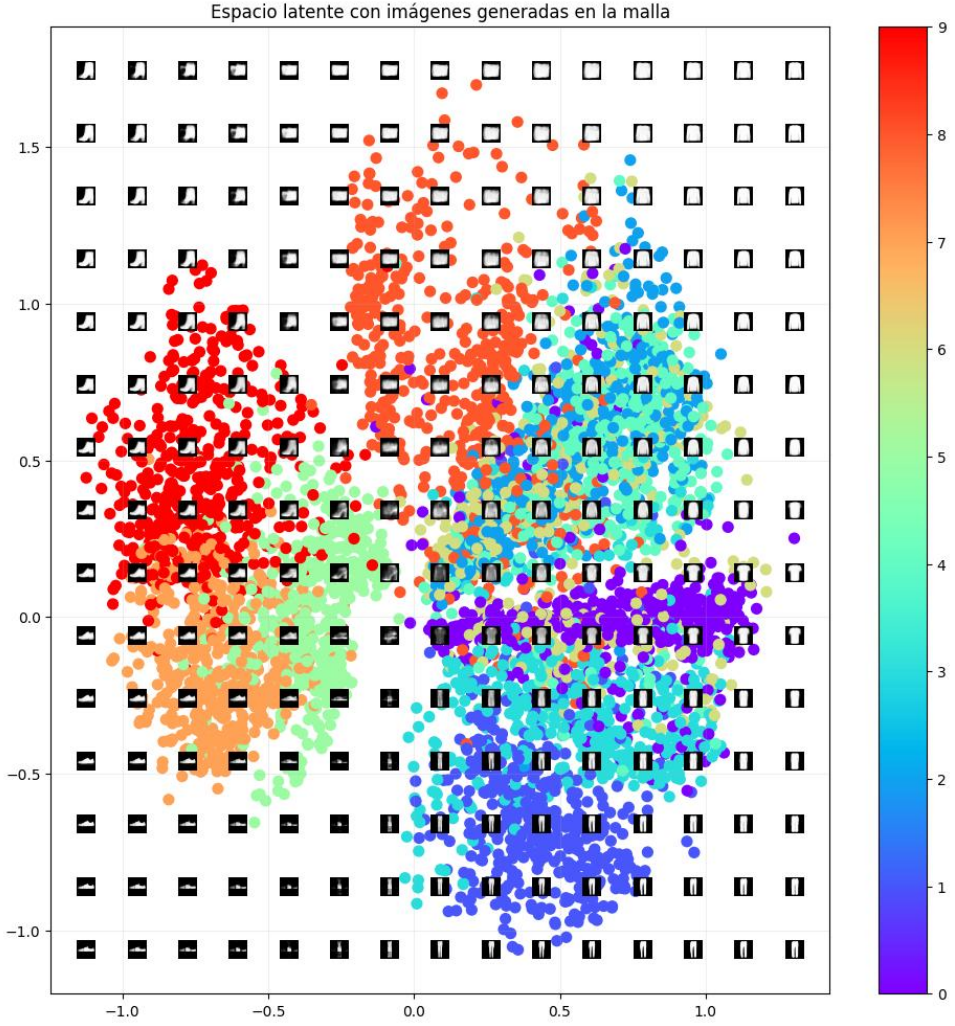
3. Podemos observar vacíos en el espacio latente donde ninguna de las imágenes originales está codificada. Por ejemplo, hay grandes espacios en blanco en los bordes del dominio:

El autocodificador no tiene ninguna razón para garantizar que los puntos en estas áreas se decodifiquen en prendas reconocibles, ya que muy pocas imágenes del conjunto de entrenamiento están codificadas ahí.

Incluso los puntos centrales pueden no decodificarse en imágenes bien formadas. Esto se debe a que el autocodificador no está obligado a garantizar que el espacio sea continuo. Por ejemplo, aunque el punto (-1, -1) podría decodificarse de manera satisfactoria en la imagen de una sandalia, no existe ningún mecanismo para asegurar que el punto (-1.1,-1.1) produzca también una imagen satisfactoria de una sandalia.

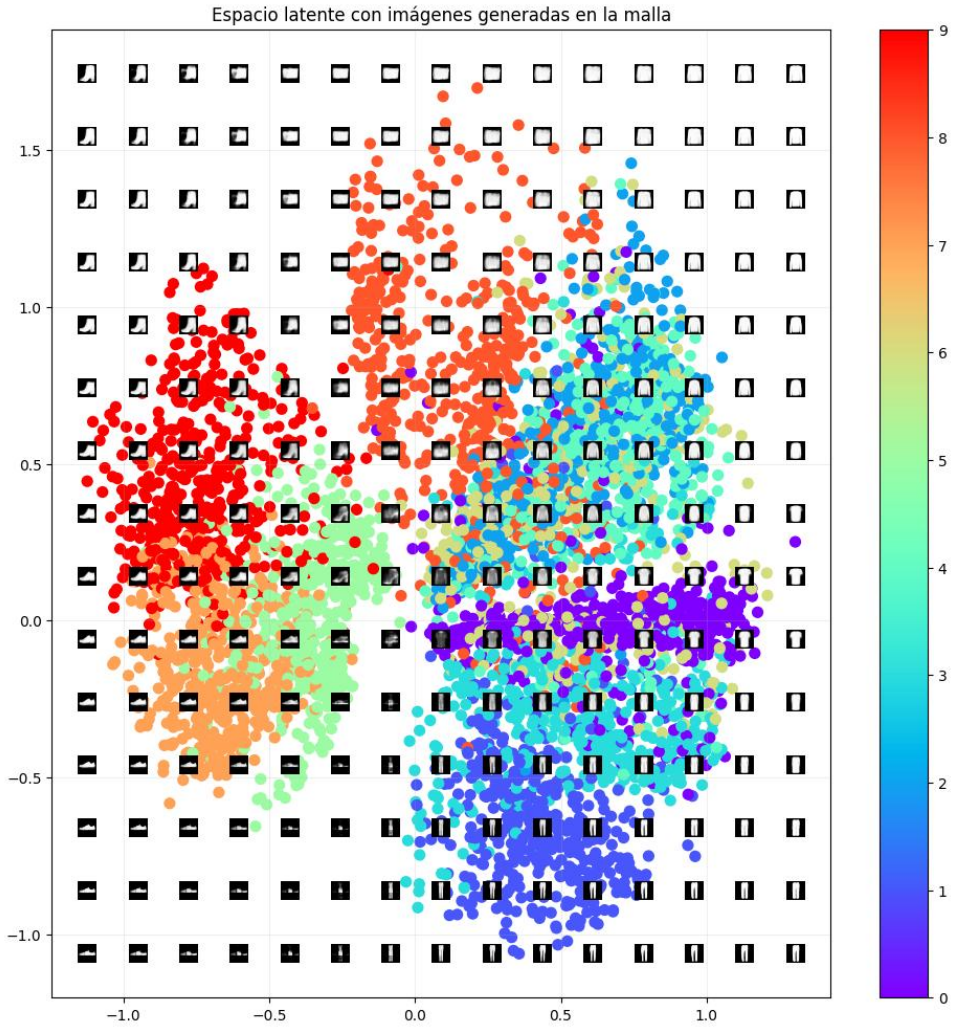
Generando nuevas imágenes

Si encimamos el espacio latente con imágenes de puntos decodificados en una malla, entonces podemos comenzar a entender por qué el decodificador no siempre puede generar imágenes satisfactorias:



- En dos dimensiones este problema es sutil.
- Con más dimensiones este problema se vuelve más aparente.
- Si dejamos que el autocodificador tenga la libertad de codificar las imágenes en el espacio latente como quiera, entonces existirán espacios enormes entre los grupos de puntos similares.

Resumen de los problemas de autocodificador

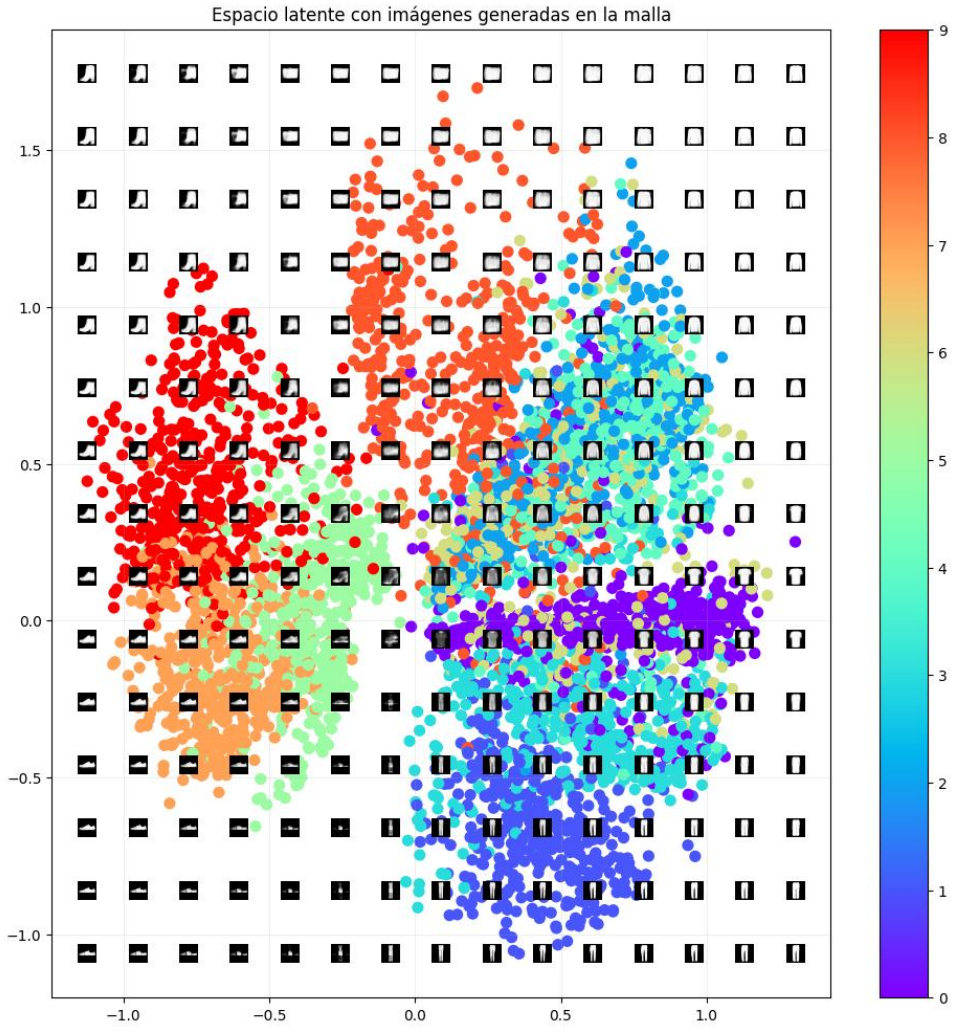


1. Determinismo del espacio latente

- Cada entrada x se mapea a un único vector $z = f_\theta(x)$.
- El espacio latente resultante está formado solo por los puntos de entrenamiento, sin estructura probabilística.

➡ No se puede muestrear "nuevos" z de manera confiable.

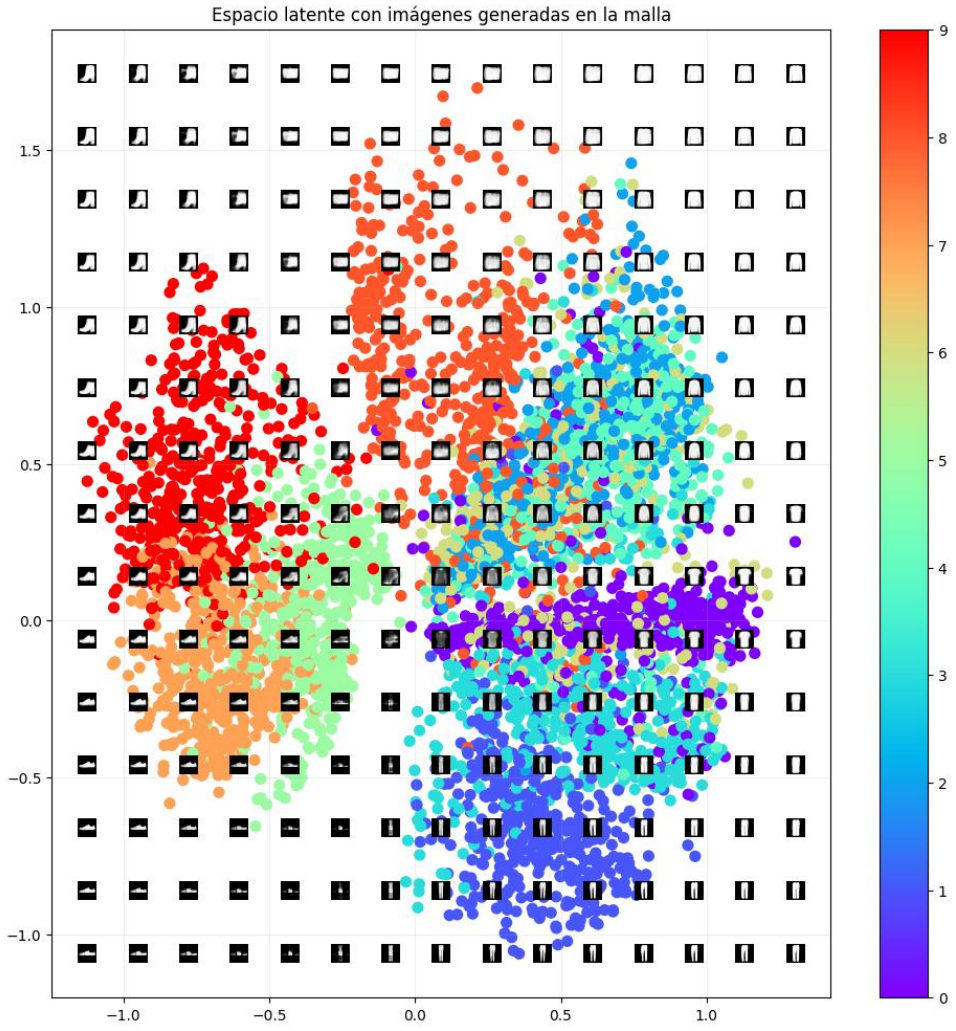
Resumen de los problemas de autocodificador



2. Espacio latente no regularizado

- Los vectores latentes pueden quedar **dispersos, huecos y desorganizados**.
- Si tomas un punto intermedio entre dos z, es probable que no corresponda a un dato válido.
- No garantiza continuidad ni cobertura uniforme del espacio.

Resumen de los problemas de autocodificador



3. Sin modelo explícito de probabilidad

- El autoencoder minimiza solo error de reconstrucción $\mathbf{L}(\mathbf{x}, \hat{\mathbf{x}})$, pero no aprende explícitamente la distribución de los datos $\mathbf{p}(\mathbf{x})$.

➡ Esto limita su uso como modelo generativo en el sentido probabilístico.

Autocodificadores Variacionales

En el 2013
Diederik P. Kingma and Max Welling
Publicaron un artículo que pondría las bases de un nuevo tipo de red neuronal llamada **Autoencoder Variacional**.

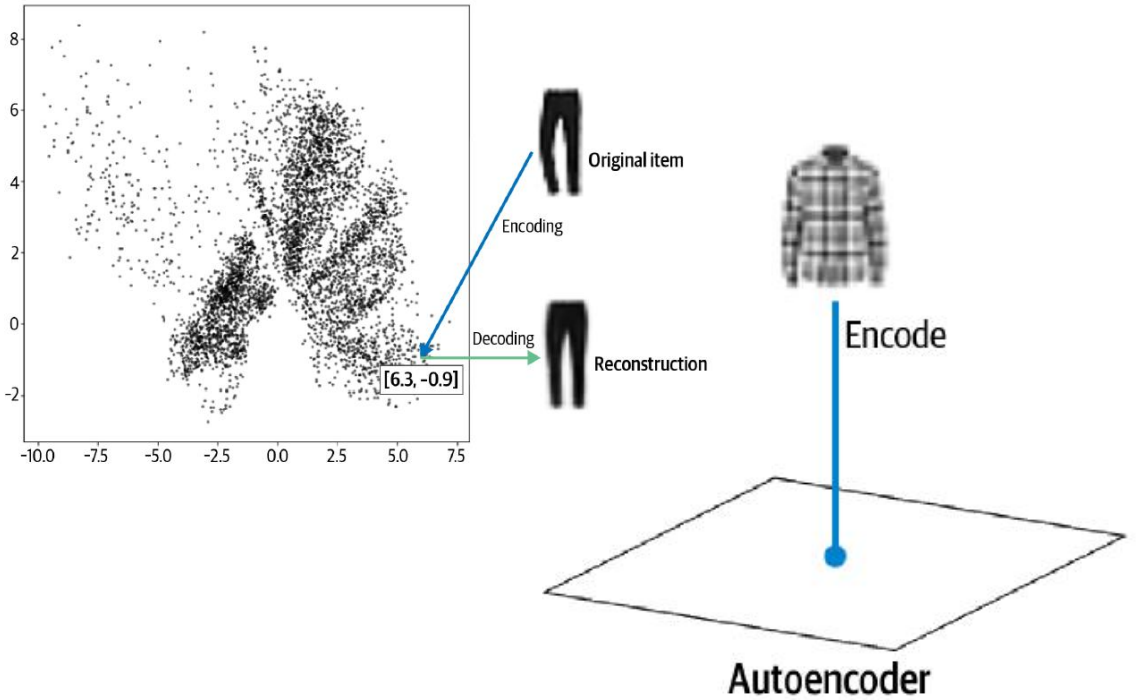
Una formulación **eficiente, diferenciable y escalable con deep learning** que permitiera entrenar con *backpropagation*.

Esta es ahora una de las arquitecturas de aprendizaje profundo más **fundamentales y conocidas para el modelado generativo**.

Kingma, D. P., & Welling, M. (2013). *Auto-encoding variational Bayes*. arXiv. https://arxiv.org/abs/1312.6114

Kingma, D. P., & Welling, M. (2014). *Auto-encoding variational Bayes*. In *Proceedings of the 2nd International Conference on Learning Representations (ICLR 2014)*.

Autocodificadores Variacionales



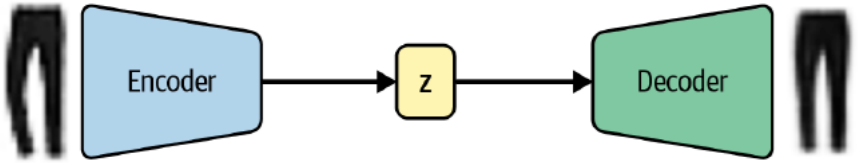
Aprender a mapear cada imagen directamente a un punto en el espacio latente. $x \rightarrow z$

Aprende a mapear cada imagen en una distribución de probabilidad normal multivariada alrededor de un punto en el espacio latente.

Se tiene que modificar el codificador y la función de pérdida

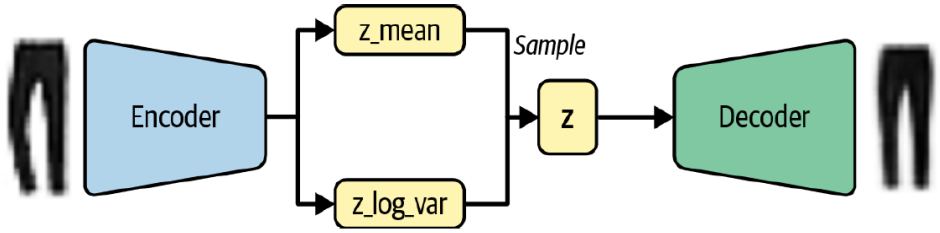
Codificador

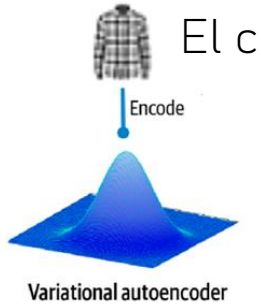
Antes



El codificador produce el vector latente z.

Ahora





El codificador produce los parámetros de una distribución normal

$$\mathcal{N}(\mu, \sigma)$$

El centro de la campana

Mide qué tan extendidos o concentrados están los datos alrededor de la media.

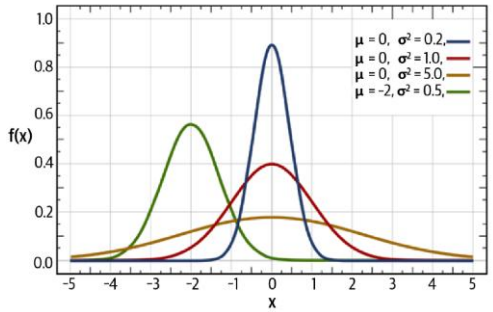
Univariada

(distribuye los puntos sobre una línea)

$$f\left(x \mid \mu, \sigma^2\right) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

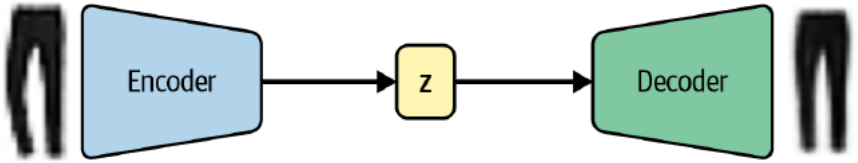
La función de densidad en el punto x, dado que la distribución normal tiene parámetros μ y σ^2.

- Si la varianza es pequeña, la campana será estrecha y picuda.
- Si la varianza es grande, la campana será ancha y achatada.



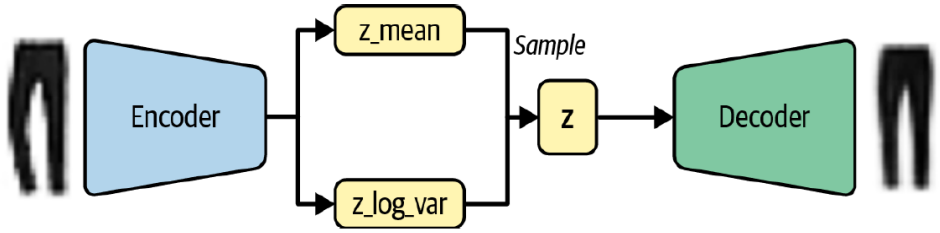
Codificador

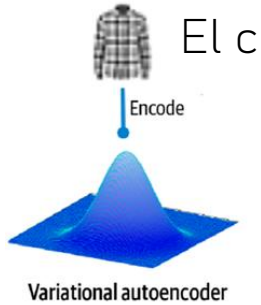
Antes



El codificador produce el vector latente z.

Ahora





El codificador produce los parámetros de una distribución normal

$\mathcal{N}(\mu, \Sigma)$ En **k** dimensiones

Vector de medias.

Matriz de covarianza simétrica

Nos dice no solo la dispersión de cada variable, sino también cómo se mueven en relación unas con otras.

Multivariada

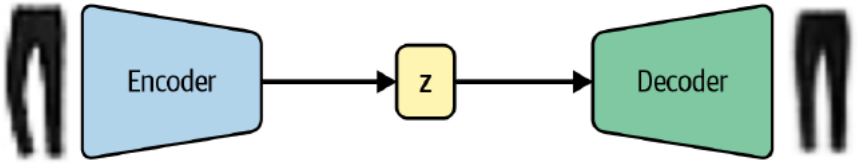
(distribuye los puntos sobre **un plano (2D)**, **un espacio (3D)**, o en general un espacio de muchas dimensiones.

$$f(x_1, ..., x_k) = \frac{\exp\left(-\frac{1}{2}(\mathbf{x}-\mu)^{\mathrm{T}}\Sigma^{-1}(\mathbf{x}-\mu)\right)}{\sqrt{(2\pi)^k|\Sigma|}}$$

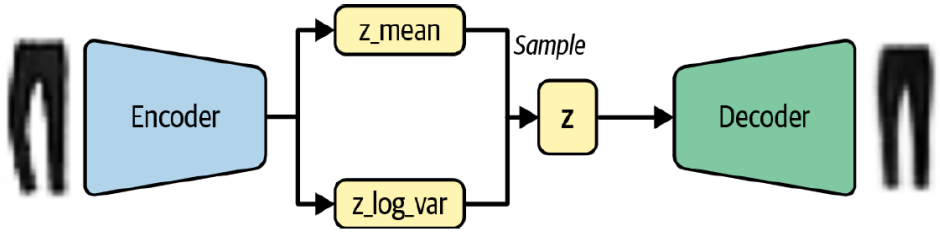
- Típicamente usaremos distribuciones normales cuya matriz de covarianza es diagonal.
- La distribución es independiente en cada dimensión.
- Podemos muestrear un vector en donde cada elemento está distribuido de manera normal con media y varianza independientes.

Codificador

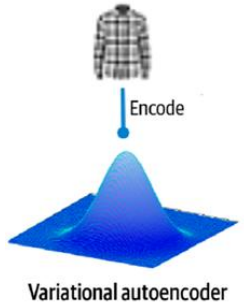
Antes



Ahora



El codificador produce el vector latente z.



Por default usaremos esta distribución multivariada:

$$\mathcal{N}(0, \mathbf{I})$$

Un vector de medias con valores de cero

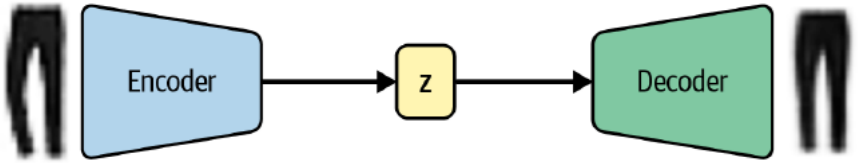
Una matriz identidad de covarianza.

El espacio latente está **centrado en el origen** (alrededor de 0).

- Cada dimensión del vector latente **z** tiene **varianza = 1** (dispersión estándar).
- Las **dimensiones son independientes entre sí** (todas las covarianzas fuera de la diagonal son 0).

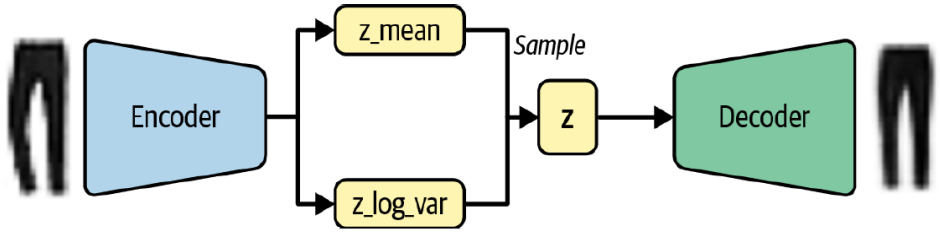
Codificador

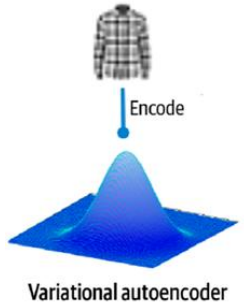
Antes



El codificador produce el vector latente z.

Ahora





El codificador produce:

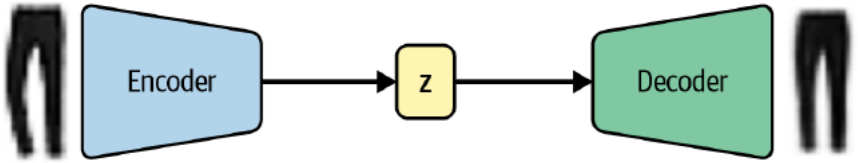
- Un vector de **medias** $\mu(x)$.
- Un vector de **log-varianzas** $\log \sigma^2(x)$.

- Asegura positividad automáticamente.
- Mejora estabilidad numérica.
- Simplifica el cálculo de la pérdida KL.

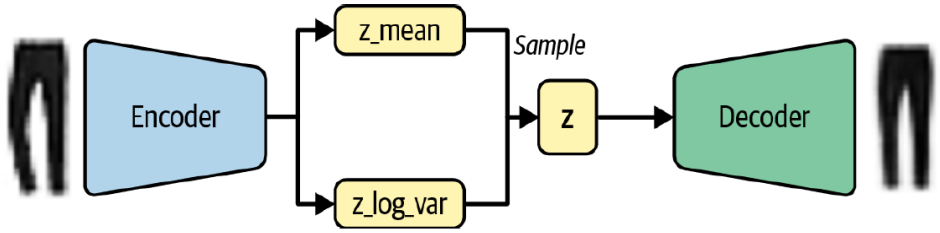
Aprende a reconstruir bien incluso con variaciones aleatorias.

Codificador

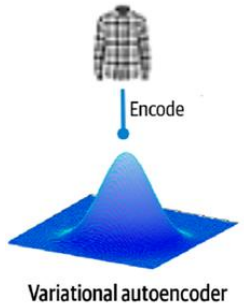
Antes



Ahora



El codificador produce el vector latente z.



De codificador obtenemos

$$z_\mu = \mu(x), \quad z_{\log \sigma^2} = \log \sigma^2(x)$$

Queremos muestrear un vector z de esa distribución y pasarlo al decodificador.

z = random.normal(mean, var)

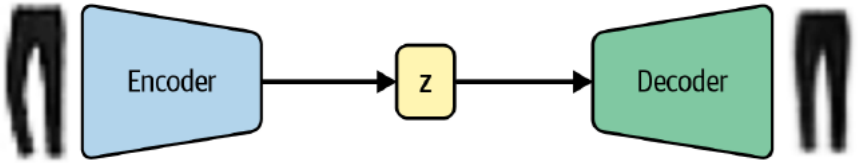
- Pero, en backpropagation necesitamos calcular *cómo cambia la pérdida si modificamos un poquito μ o σ*.
- Y si z es distinto en cada paso (por el azar puro), ese cálculo se vuelve inconsistente porque:

- No hay una derivada definida del muestreo respecto a μ o σ.
- El gradiente se convierte en ruido inservible.

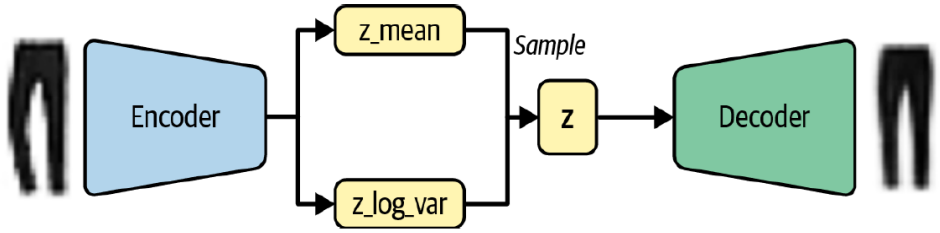
El codificador no recibe una señal clara para ajustar sus pesos → **no aprende**.

Codificador

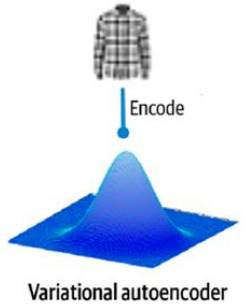
Antes



Ahora



El codificador produce el vector latente z.



z = random.normal(mean, var)

Truco para obtener una muestra *z:*

1. Convertir log-varianza a desviación estándar:

$$\sigma = \exp\left(\frac{1}{2} z_{\log \sigma^2}\right)$$

2. Re-parametrización:

- En vez de muestrear directament$\mathcal{N}(\mu, \sigma^2)$ lo hacemos así;

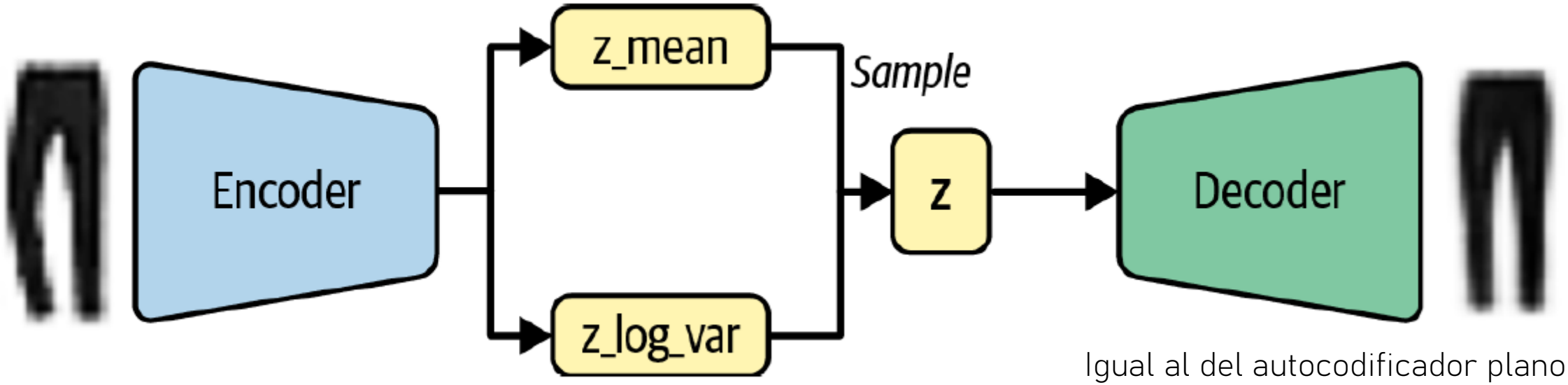
$$z = \mu + \sigma \cdot \epsilon, \quad \epsilon \sim \mathcal{N}(0, I)$$

epsilon = random.normal(mean=0, stddev=1)
z = μ + σ * epsilon

- El azar está en ε, que efectivamente cambia en cada forward pass (sigue habiendo ruido).

- La parte determinista depende de parámetros entrenables que pueden ser diferenciables:
  - Si cambias μ, z cambia suavemente.
  - Si cambias σ, z cambia proporcionalmente.
- Así, aunque z varíe por el ruido, los gradientes hacia μ y σ siguen siendo válidos y útiles.

Arquitectura del VAE



Igual al del autocodificador plano

- Como estamos muestreando un punto aleatorio de un área alrededor de `z_mean`, el decodificador debe de asegurarse de que todos los puntos en el mismo vecindario produzcan imágenes similares cuando se decodifiquen, de tal forma que la pérdida de la reconstrucción permanezca pequeña.

- Esta es una propiedad que asegura que cuando elijamos un punto en el espacio latente que nunca se haya visto por el decoder, sea muy probable que se decodifique en una imagen que esté bien formada.

Construyendo el codificador del VAE

1. Tenemos que construir una capa llamada Sampling que nos permitirá muestrear de la distribución definida por `z_mean` y `z_log_var` .

```
class Sampling(layers.Layer): ❶
    def call(self, inputs):
        z_mean, z_log_var = inputs
        batch = tf.shape(z_mean)[0]
        dim = tf.shape(z_mean)[1]
        epsilon = K.random_normal(shape=(batch, dim))
        return z_mean + tf.exp(0.5 * z_log_var) * epsilon ❷
```

- Creamos una subclase de la clase Layer.
- Hay que definir el método call, el cual describe cómo la capa transforma el tensor.

El truco de reparametrizar

- En vez de muestrear directamente de una distribución normal con los parámetros z_mean y z_log_var, podemos muestrear epsilon de una distribución normal estándar y después ajustar manualmente la muestra para que tenga la media y varianza correctas.

Construyendo el codificador del VAE

```
encoder_input = layers.Input(
    shape=(32, 32, 1), name="encoder_input"
)
x = layers.Conv2D(32, (3, 3), strides=2, activation="relu", padding="same")(
    encoder_input
)
x = layers.Conv2D(64, (3, 3), strides=2, activation="relu", padding="same")(x)
x = layers.Conv2D(128, (3, 3), strides=2, activation="relu", padding="same")(x)
shape_before_flattening = K.int_shape(x)[1:]

x = layers.Flatten()(x)
z_mean = layers.Dense(2, name="z_mean")(x) ❶
z_log_var = layers.Dense(2, name="z_log_var")(x)
z = Sampling()([z_mean, z_log_var]) ❷
encoder = models.Model(encoder_input, [z_mean, z_log_var, z], name="encoder") ❸
```

En vez de conectar la capa Flatten directamente a el espacio latente 2D, la conectamos a las capas z_mean y z_log_var.

Construyendo el codificador del VAE

```
encoder_input = layers.Input(
    shape=(32, 32, 1), name="encoder_input"
)
x = layers.Conv2D(32, (3, 3), strides=2, activation="relu", padding="same")(
    encoder_input
)
x = layers.Conv2D(64, (3, 3), strides=2, activation="relu", padding="same")(x)
x = layers.Conv2D(128, (3, 3), strides=2, activation="relu", padding="same")(x)
shape_before_flattening = K.int_shape(x)[1:]

x = layers.Flatten()(x)
z_mean = layers.Dense(2, name="z_mean")(x) ❶
z_log_var = layers.Dense(2, name="z_log_var")(x)
z = Sampling()([z_mean, z_log_var]) ❷

encoder = models.Model(encoder_input, [z_mean, z_log_var, z], name="encoder") ❸
```

La capa Sampling muestrea un punto z en el espacio latente de la distribución normal definida por los parámetros z_mean y z_log_var.

Construyendo el codificador del VAE

```
encoder_input = layers.Input(
    shape=(32, 32, 1), name="encoder_input"
)
x = layers.Conv2D(32, (3, 3), strides=2, activation="relu", padding="same")(
    encoder_input
)
x = layers.Conv2D(64, (3, 3), strides=2, activation="relu", padding="same")(x)
x = layers.Conv2D(128, (3, 3), strides=2, activation="relu", padding="same")(x)
shape_before_flattening = K.int_shape(x)[1:]

x = layers.Flatten()(x)
z_mean = layers.Dense(2, name="z_mean")(x) ❶
z_log_var = layers.Dense(2, name="z_log_var")(x)
z = Sampling()([z_mean, z_log_var]) ❷
encoder = models.Model(encoder_input, [z_mean, z_log_var, z], name="encoder") ❸
```

El modelo de Keras que define el encoder, un modelo que toma como entrada una imagen y da como salida z_mean, z_log_var y un punto z muestreado de la distribución normal definida por estos parámetros.

Construyendo el codificador del VAE

Layer (type)	Output shape	Param #	Connected to
InputLayer (input)	(None, 32, 32, 1)	0	[]
Conv2D (conv2d_1)	(None, 16, 16, 32)	320	[input]
Conv2D (conv2d_2)	(None, 8, 8, 64)	18,496	[conv2d_1]
Conv2D (conv2d_3)	(None, 4, 4, 128)	73,856	[conv2d_2]
Flatten (flatten)	(None, 2048)	0	[conv2d_3]
Dense (z_mean)	(None, 2)	4,098	[flatten]
Dense (z_log_var)	(None, 2)	4,098	[flatten]
Sampling (z)	(None, 2)	0	[z_mean, z_log_var]

Total params	100,868
Trainable params	100,868
Non-trainable params	0

La función de pérdida

- Anteriormente la función de pérdida solo consistía en la pérdida de reconstrucción entre las imágenes originales y las reconstruidas.
- Aquí agregaremos un componente extra:

Kullback-Leibler (KL) divergence term

- Es una manera de medir qué tanto una distribución de probabilidad difiere de otra.
- En el VAE queremos medir qué tanto nuestra distribución normal con los parámetros ***z_mean*** y ***z_log_var*** difiere de una distribución normal estándar.
- En este caso especial la divergencia KL tiene la siguiente forma cerrada:

```
kl_loss = -0.5 * sum(1 + z_log_var - z_mean ^ 2 - exp(z_log_var))
```

o en notación matemática:

$$D_{KL}[N(\mu, \sigma \parallel N(0,1)] = -\frac{1}{2}\sum\left(1 + log\left(\sigma^2\right) - \mu^2 - \sigma^2\right)$$

- Se minimiza a 0 cuando z_mean=0 y z_log_var=0 para todas las dimensiones.
- Conforma estos dos términos comienzan a alejarse del 0, la pérdida se incrementa.
- Es decir, este término penaliza a la red por codificar observaciones a z_mean y z_log_var que difieren significativamente de los parámetros de una distribución normal estándar, con z_mean = 0 y z_log_var = 0.

La función de pérdida

- Anteriormente la función de pérdida solo consistía en la pérdida de reconstrucción entre las imágenes originales y las reconstruidas.
- Aquí agregaremos un componente extra:

Kullback-Leibler (KL) divergence term

Es como un **imán** que empuja las distribuciones del encoder hacia una normal estándar.

El codificador intentará usar simétricamente y eficientemente el espacio alrededor del origen.

Existe menos probabilidad de que se formen grandes huecos entre clusters de puntos

Esto asegura que el espacio latente sea **ordenado y generativo** en lugar de caótico y disperso.

La función de pérdida

Variaciones

- En el artículo original del VAE, la función de pérdida era simplemente la adición de la pérdida de reconstrucción y el término KL divergence loss.

- Un variante es la β-VAE

    - Incluye un factor que le da un peso al término KL divergence para asegurar que esté bien balanceado con la pérdida de reconstrucción.
    - Si el peso de la pérdida de reconstrucción es muy grande, el término KL divergence no tendrá el efecto regulatorio deseado y veremos el mismo problema que en el autoencoder plano.
    - Si el peso del término KL divergence es muy grande, entonces las imágenes reconstruídas serán de mala calidad.
    - Este peso es uno de los parámetros a tunear cuando entrenamos una VAE.

Entrenando el VAE

```
class VAE(models.Model):
    def __init__(self, encoder, decoder, **kwargs):
        super(VAE, self).__init__(**kwargs)
        self.encoder = encoder
        self.decoder = decoder
        self.total_loss_tracker = metrics.Mean(name="total_loss")
        self.reconstruction_loss_tracker = metrics.Mean(
            name="reconstruction_loss"
        )
        self.kl_loss_tracker = metrics.Mean(name="kl_loss")

    @property
    def metrics(self):
        return [
            self.total_loss_tracker,
            self.reconstruction_loss_tracker,
            self.kl_loss_tracker,
        ]

    def call(self, inputs): ❶
        z_mean, z_log_var, z = encoder(inputs)
        reconstruction = decoder(z)
        return z_mean, z_log_var, reconstruction
```

Esta función describe qué queremos que regrese cuando llamamos al VAE con un imagen de entrada en particular.

Entrenando el VAE

```
def train_step(self, data): ❷
    with tf.GradientTape() as tape:
        z_mean, z_log_var, reconstruction = self(data)
        reconstruction_loss = tf.reduce_mean(
            500
            * losses.binary_crossentropy(
                data, reconstruction, axis=(1, 2, 3)
            )
        ) ❸
        kl_loss = tf.reduce_mean(
            tf.reduce_sum(
                -0.5
                * (1 + z_log_var - tf.square(z_mean) - tf.exp(z_log_var)),
                axis = 1,
            )
        )
        total_loss = reconstruction_loss + kl_loss ❹

    grads = tape.gradient(total_loss, self.trainable_weights)
```

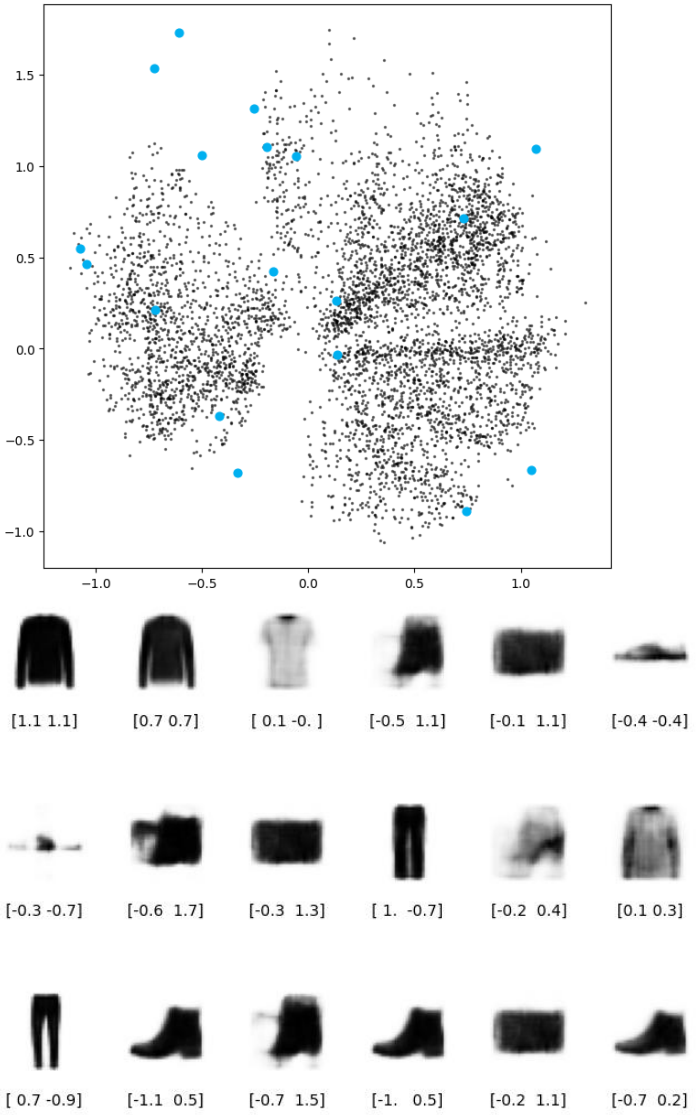
Esta función describe un paso de entrenamiento de la VAE, incluyendo el cálculo de la función de pérdida.

Un valor de beta de 500 se usa en la pérdida de reconstrucción.

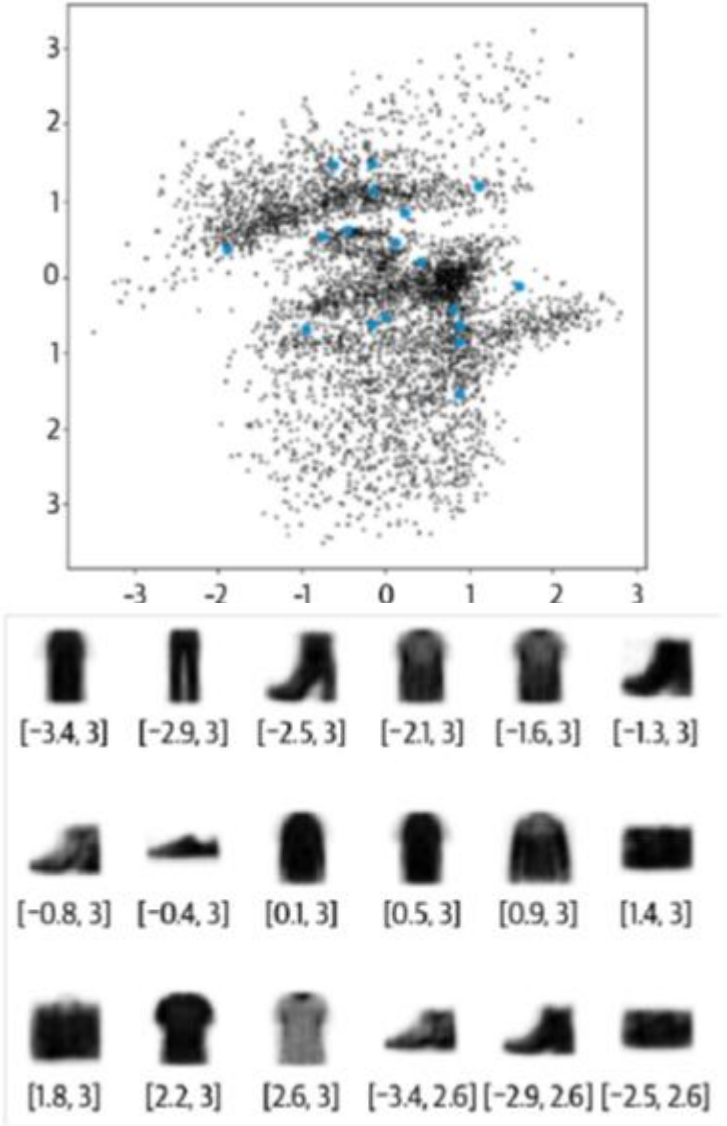
La pérdida total es la suma de la pérdida de reconstrucción y la pérdida de KL divergence.

Análisis del VAE

Autocodificador



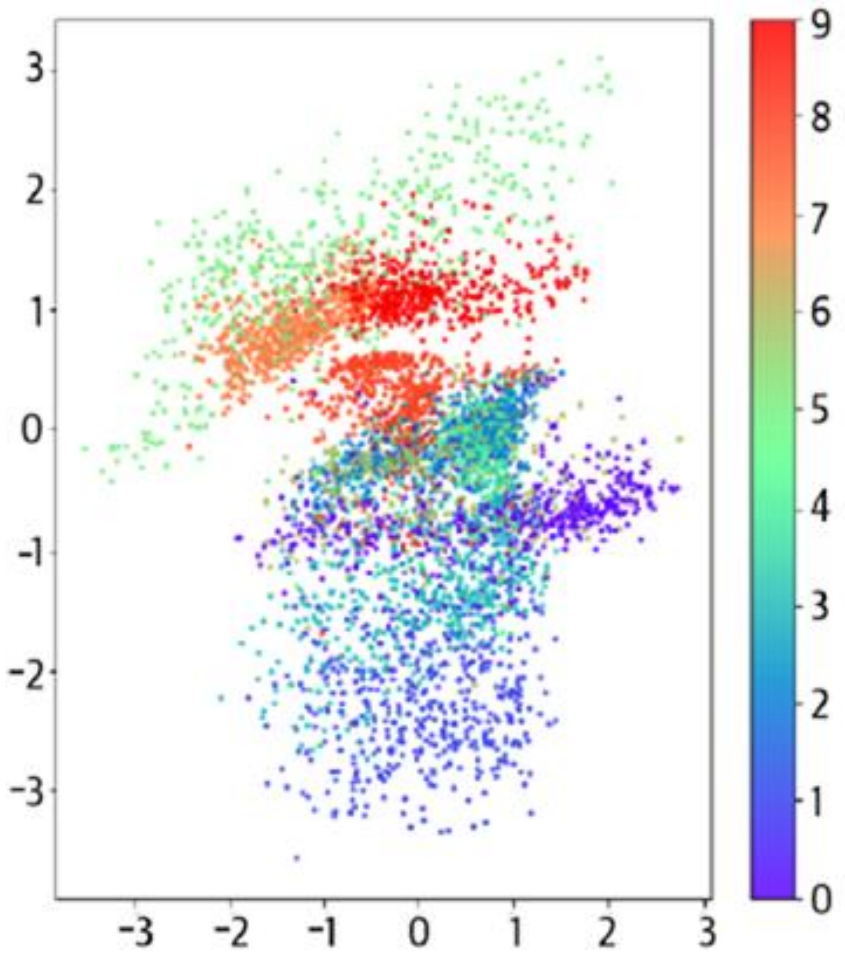
VAE



1. El término de la pérdida KL divergence asegura que los valores las imágenes codificadas no se alejen mucho de una distribución estándar.

2. No hay demasiadas imágenes pobremente formadas ya que el espacio latente es mucho más continuo, debido a que el codificador es ahora estocástico en vez de determinista.

Análisis del VAE



ID	Clothing label
0	T-shirt/top
1	Trouser
2	Pullover
3	Dress
4	Coat
5	Sandal
6	Shirt
7	Sneaker
8	Bag
9	Ankle boot

- Vemos que no hay un trato preferencial a ninguna clase (cada color está aproximadamente igualmente representado.).
- Nota que el VAE no usó las etiquetas.

Explorando el espacio latente

- Hasta ahora el autoencoder y el VAE se limitaron a un espacio latente de dos dimensiones.
- Nos ayudó a visualizar cómo funcionan.

- Veamos ahora un dataset más complejo y veamos las características asombrosas que los VAE pueden llevar a cabo cuando incrementamos la dimensionalidad del espacio latente.

VAE para el CelebA dataset

200,000 imágenes a color de rostros de celebridades.
Cada una anotada con varias etiquetas (usando sombrero, sonriendo, etc.)

Nota: No necesitamos las etiquetas para construir el VAE
Nos serán útiles después para explorar cómo estas características se capturan en el espacio latente

Entrenando el VAE

- La arquitectura del VAE para este dataset es similar al que ocupamos con Fashion-MNIST, con las siguientes pequeñas diferencias:

Codificador

Layer (type)	Output shape	Param #	Connected to
InputLayer (input)	(None, 32, 32, 3)	0	[]
Conv2D (conv2d_1)	(None, 16, 16, 128)	3,584	[input]
BatchNormalization (bn_1)	(None, 16, 16, 128)	512	[conv2d_1]
LeakyReLU (lr_1)	(None, 16, 16, 128)	0	[bn_1]
Conv2D (conv2d_2)	(None, 8, 8, 128)	147,584	[lr_1]
BatchNormalization (bn_2)	(None, 8, 8, 128)	512	[conv2d_2]
LeakyReLU (lr_2)	(None, 8, 8, 128)	0	[bn_2]
Conv2D (conv2d_3)	(None, 4, 4, 128)	147,584	[lr_2]
BatchNormalization (bn_3)	(None, 4, 4, 128)	512	[conv2d_3]
LeakyReLU (lr_3)	(None, 4, 4, 128)	0	[bn_3]
Conv2D (conv2d_4)	(None, 2, 2, 128)	147,584	[lr_3]
BatchNormalization (bn_4)	(None, 2, 2, 128)	512	[conv2d_4]
LeakyReLU (lr_4)	(None, 2, 2, 128)	0	[bn_4]
Flatten (flatten)	(None, 512)	0	[lr_4]
Dense (z_mean)	(None, 200)	102,600	[flatten]
Dense (z_log_var)	(None, 200)	102,600	[flatten]
Sampling (z)	(None, 200)	0	[z_mean, z_log_var]

Total params	653,584
Trainable params	652,560
Non-trainable params	1,024

Ahora tenemos tres canales de entrada (RGB).

Incluimos capas de normalización por lotes después de cada capa convolucional para estabilizar el entrenamiento.

El espacio latente será de 200 dimensiones en vez de 2.

Nota: Usamos un factor de beta para el KL divergence a 2,000.

Entrenando el VAE

Decodificador

Layer (type)	Output shape	Param #
InputLayer	(None, 200)	0
Dense	(None, 512)	102,912
BatchNormalization	(None, 512)	2,048
LeakyReLU	(None, 512)	0
Reshape	(None, 2, 2, 128)	0
Conv2DTranspose	(None, 4, 4, 128)	147,584
BatchNormalization	(None, 4, 4, 128)	512
LeakyReLU	(None, 4, 4, 128)	0
Conv2DTranspose	(None, 8, 8, 128)	147,584
BatchNormalization	(None, 8, 8, 128)	512
LeakyReLU	(None, 8, 8, 128)	0
Conv2DTranspose	(None, 16, 16, 128)	147,584
BatchNormalization	(None, 16, 16, 128)	512
LeakyReLU	(None, 16, 16, 128)	0
Conv2DTranspose	(None, 32, 32, 128)	147,584
BatchNormalization	(None, 32, 32, 128)	512
LeakyReLU	(None, 32, 32, 128)	0
Conv2DTranspose	(None, 32, 32, 3)	3,459

¡Después de aproximadamente **cinco épocas** de entrenamiento, nuestro VAE debería ser capaz de generar imágenes novedosas de rostros de celebridades.!

Análisis del VAE

Ejemplos de rostros reales

Reconstrucciones

El VAE ha capturado con éxito las características clave de cada rostro:

- el ángulo de la cabeza,
- el estilo del cabello,
- la expresión facial.

Algunos de los detalles finos se han perdido, pero es importante recordar que el objetivo de construir VAEs no es el obtener reconstrucciones perfectas, sino el que podamos muestrear del espacio latente para generar nuevos rostros.

Análisis del VAE

Verifiquemos que la distribución de los puntos en el espacio latente se aproximen a una distribución normal multivariada estándar:

```
_, _, z = vae.encoder.predict(example_images)
```

z es el vector latente para muchas imágenes de ejemplo.

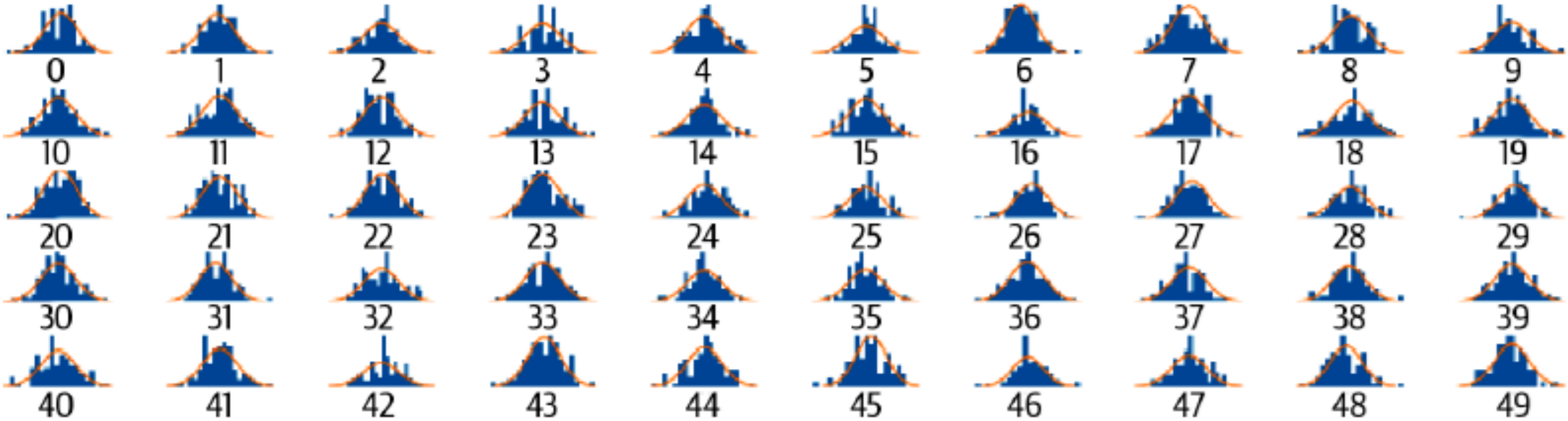
```
for i in range(50):
    ax.hist(z[:, i], density=True, bins=20)  # histograma de la i-ésima dimensión
    ax.plot(x, norm.pdf(x))                   # dibuja la curva de una N(0,1)
```

- Se dibuja el histograma de los valores de **z** en la dimensión **i** (para muchas muestras).
- Se sobrepone la curva de densidad de la normal estándar.
- Así se ve si cada dimensión realmente se comporta como debería.

Análisis del VAE

Verifiquemos que la distribución de los puntos en el espacio latente se aproximen a una distribución normal multivariada estándar:

Las primeras 50 dimensiones



¡No hay ninguna distribución que resalte por tener una distribución significativamente diferente de la normal estándar!

Podemos ahora pasar a generar nuevos rostros.

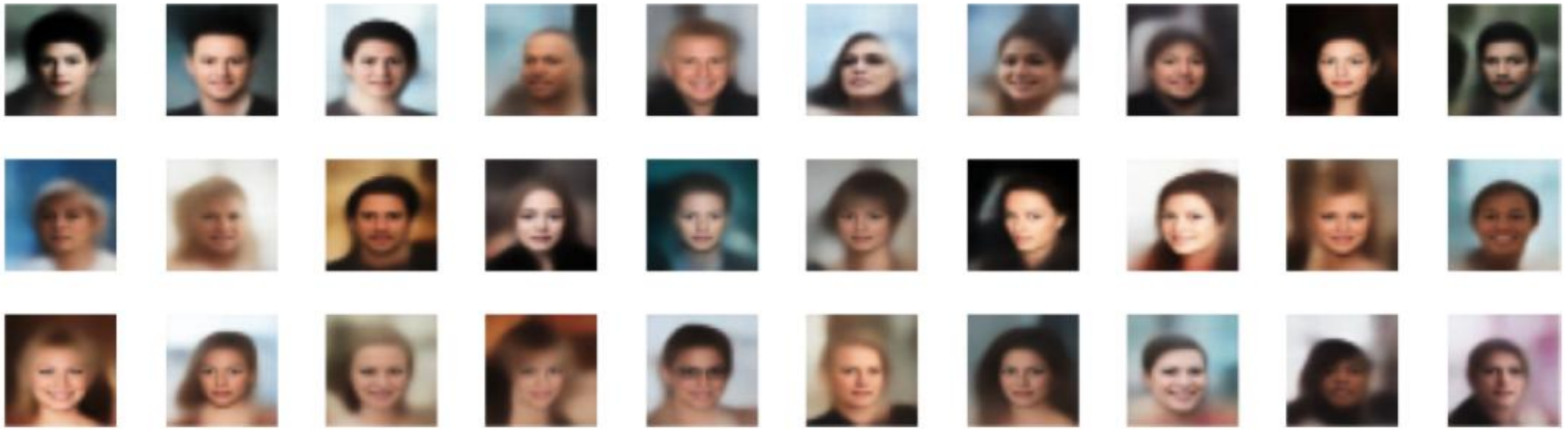
Generando nuevos rostros

```
grid_width, grid_height = (10,3)
z_sample = np.random.normal(size=(grid_width * grid_height, 200)) ❶

reconstructions = decoder.predict(z_sample) ❷

fig = plt.figure(figsize=(18, 5))
fig.subplots_adjust(hspace=0.4, wspace=0.4)
for i in range(grid_width * grid_height):
    ax = fig.add_subplot(grid_height, grid_width, i + 1)
    ax.axis("off")
    ax.imshow(reconstructions[i, :, :]) ❸
```

Muestrea 30 puntos de una distribución normal multivariada estándar de 200 dimensiones.

Decodifica los puntos muestreados.

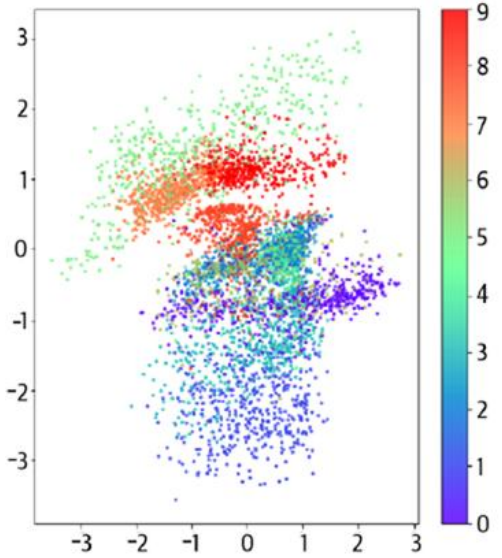
Muestra las imágenes reconstruidas.



Aritmética de los espacios latentes

De triste a feliz

- Necesitamos encontrar un vector en el espacio latente que apunte en la dirección de incrementar la sonrisa.
- Sumamos este vector a la imagen original codificada en el espacio latente.
- Esto nos da un nuevo punto que cuando lo decodifiquemos, nos debería de dar una versión más sonriente de la versión original.



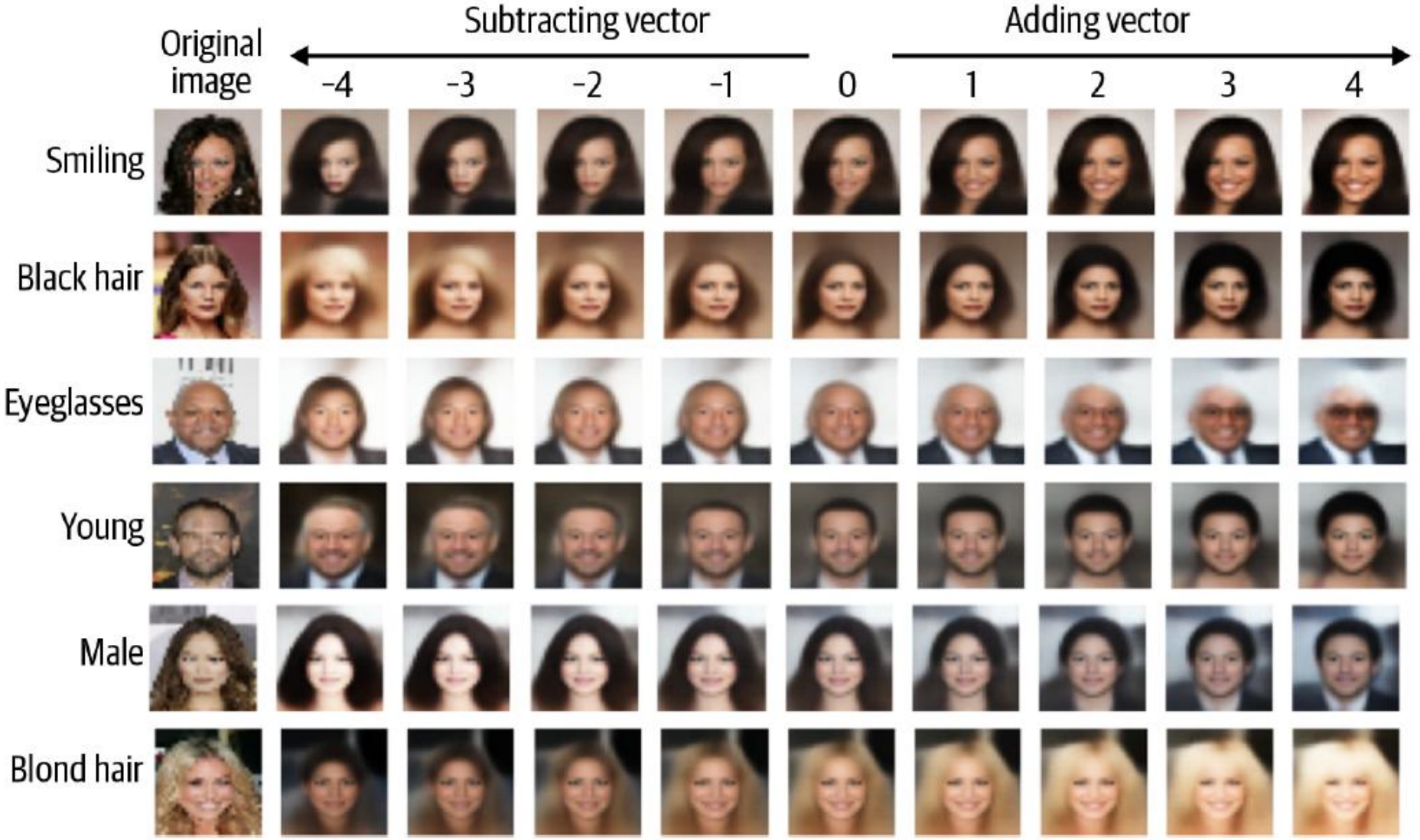
¿Cómo podemos encontrar el vector de sonreir?

Cada una de las imágenes del dataset CelebA está etiquetada con atributos, uno de los cuales es sonriente.
Si calculamos la posición promedio de las imágenes codificadas en el espacio latente con el atributo sonriente y le restamos la posición promedio de las imágenes codificadas que no tienen ese atributo, entonces obtenemos el vector que apunta en la dirección de sonreir.

```
z_new = z + alpha * feature_vector
```

Alpha es un factor que determina qué tanto del vector de la característica se le suma o se le resta.

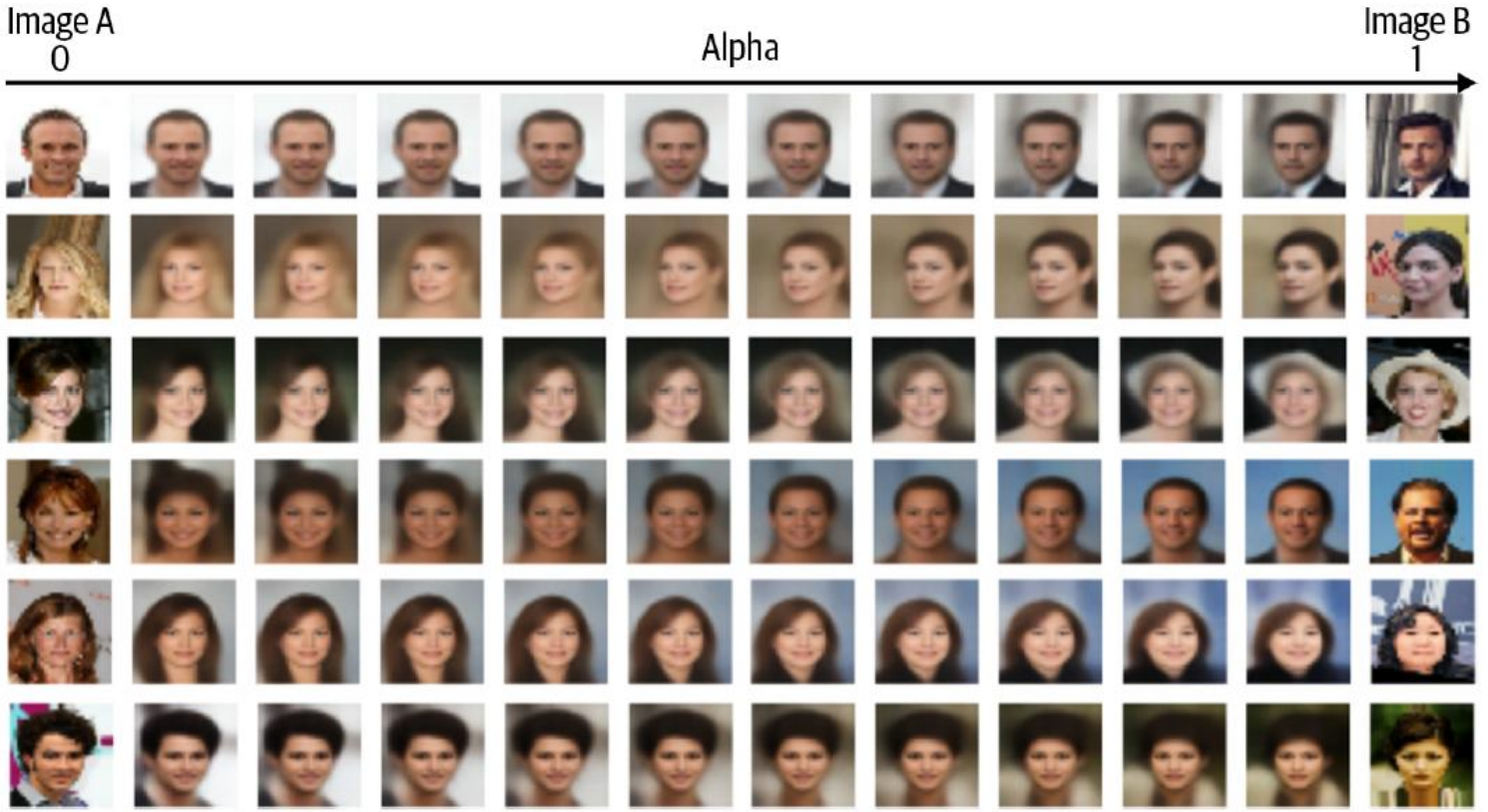
Aritmética de los espacios latentes



Transformación entre rostros

```
z_new = z_A * (1- alpha) + z_B * alpha
```

Alpha es un número entre 0 y 1 que determina qué tan lejos estamos a lo largo de la línea, alejándonos del punto A.



El espacio latente del VAE es realmente un espacio continuo que puede ser recorrido y explorado para generar una multitud de rostros humanos diferentes.

Ejercicios de tarea

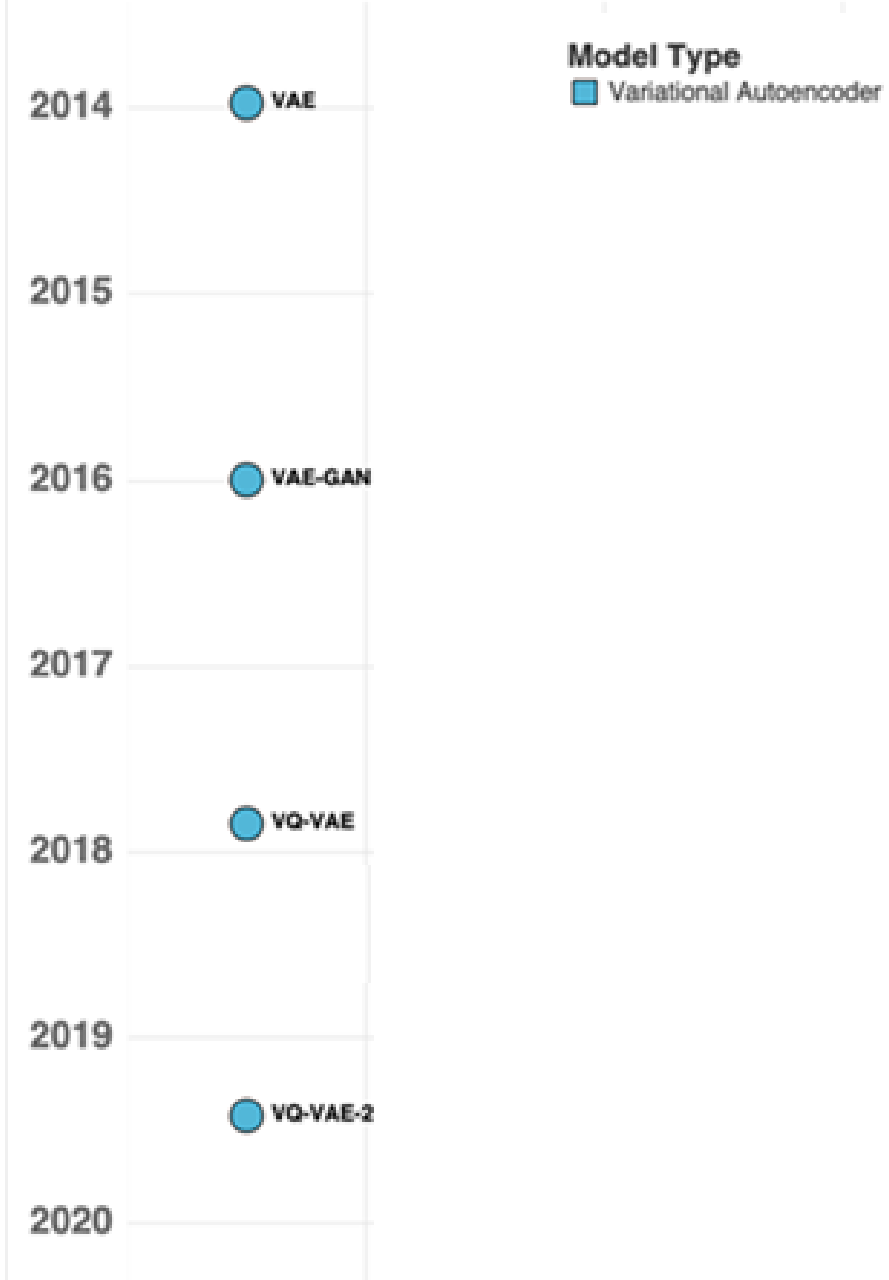
- Crear un VAE para el conjunto de datos MNIST

- Crear un VAE para el conjunto de datos CIFAR10

Google colab

Objetivo: que logre reconstruir lo mejor posible las imágenes de test.

Generative AI Timeline



Proyecto de profundización

- **Proyectos de profundización**

  Tras la sesión introductoria de cada familia de modelos, se asignará a cada estudiante (o equipo) una variante específica, con el objetivo de investigar, implementar y presentar sus particularidades.

  Etapas del proyecto:

  1. **Selección de variante**: El profesor asigna a cada equipo un modelo derivado (por ejemplo, StyleGAN, BigGAN, VQ-VAE, Glow, Stable Diffusion, etc.), procurando cubrir diversidad arquitectónica y de objetivos.
  2. **Trabajo independiente guiado**: Los estudiantes dispondrán de 1 a 2 semanas sin clase presencial para:
     - Revisar la literatura original y documentación técnica.
     - Implementar la variante o adaptar una implementación existente.
     - Comparar resultados con el modelo base estudiado en la sesión introductoria.
  3. **Presentación y discusión**: Cada equipo expondrá sus hallazgos ante la clase, abordando:
     - El problema que motivó la variante.
     - Las modificaciones arquitectónicas o de entrenamiento.
     - Las mejoras alcanzadas.

  Entregables del proyecto de profundización:

  - Presentación (1 hora) con análisis crítico y presentación de resultados.
  - Notebook ejecutable (Jupyter/Colab) con implementación o fine-tuning.

  Nota: Se pondrán en un repositorio a disposición de todos los alumnos.

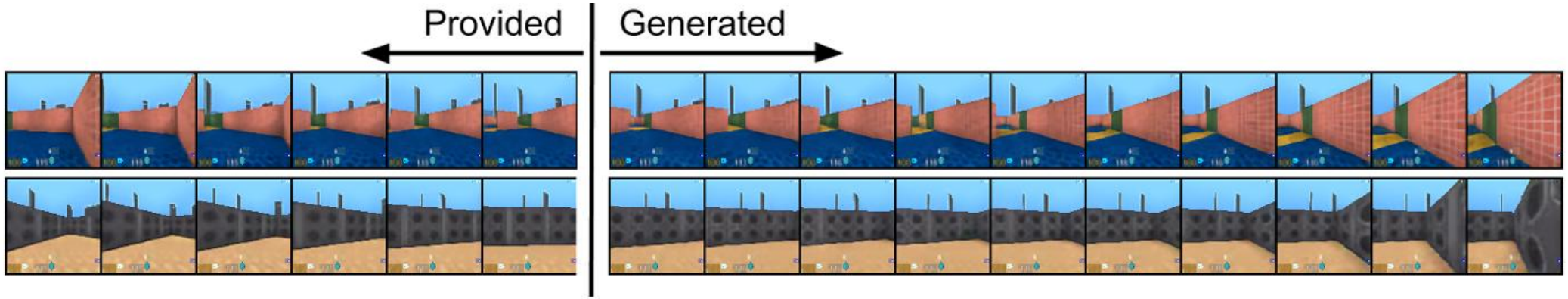
Lunes 22 de septiembre

Explicar el *paper* con énfasis en:

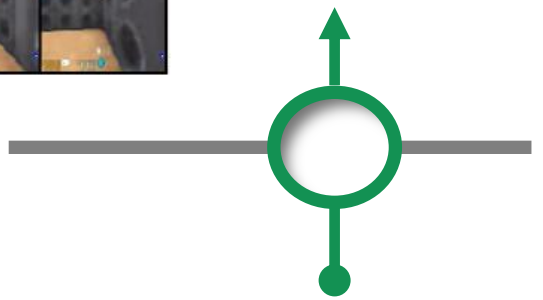
Utilizar el dataset CelebA y otro a elegir

Proyecto de profundización

En equipo



2017



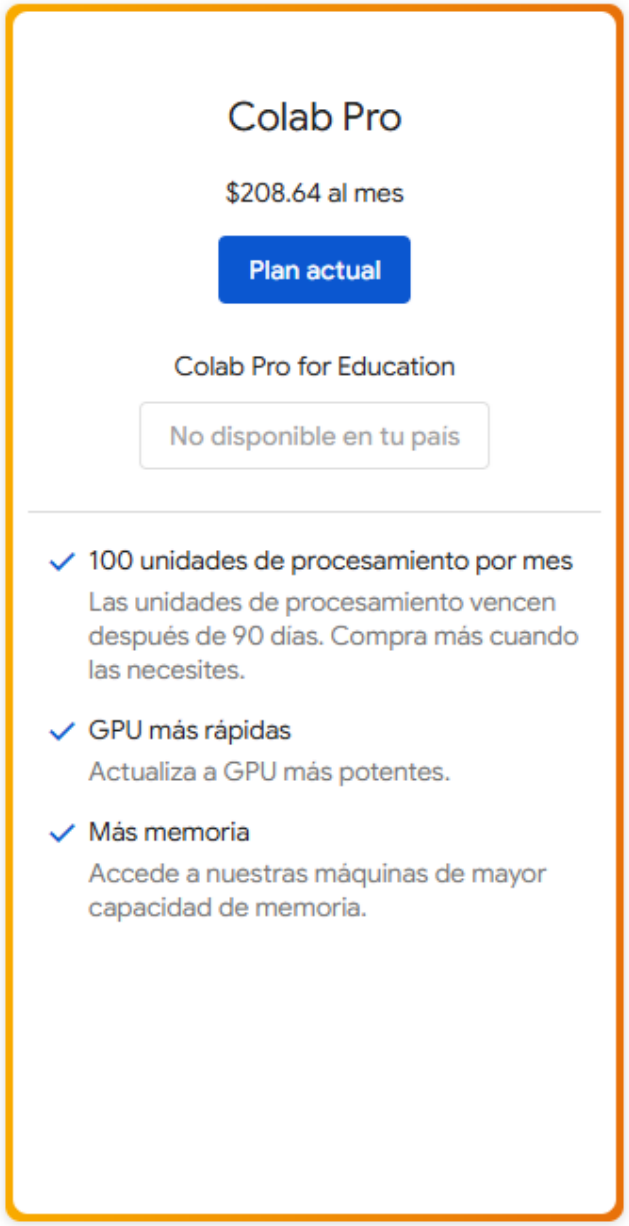
VQ-VAE (Vector Quantized VAE)

Modelo que combina las ideas de los VAEs con una **codificación discreta** basada en **vector quantization**. A diferencia de los VAE tradicionales, que utilizan variables latentes continuas, **VQ-VAE aprende una representación latente discreta**, lo cual es útil para tareas como síntesis de audio, compresión de imágenes y modelado secuencial. Esta representación discreta también permite usar modelos autoregresivos potentes (como PixelCNN) sobre los códigos latentes, mejorando la calidad de las muestras generadas.

van den Oord, A., Vinyals, O., & Kavukcuoglu, K. (2017). *Neural Discrete Representation Learning*. arXiv preprint arXiv:1711.00937.

https://keras.io/examples/generative/vq_vae/

Recomendación



1 cu = 1 hora de gpu

3 meses