



FACULTAD DE INGENIERÍA
DEPARTAMENTO DE OPERACIONES Y SISTEMAS
PROGRAMA DE INGENIERÍA INFORMÁTICA
RNA Y DEEP LEARNING

TRABAJANDO CON AUTOENCODERS

Alumno:

Doffiny S-V, Carlos

carlos.doffiny@uao.edu.co

15 de mayo de 2022

ÍNDICE

Ejercicio N°1	3
Ejercicio N°2	4
Ejercicio N°3	5
Ejercicio N°4	12
Ejercicio N°5	19
Referencias Bibliográficas	26

Ejercicio N°1

Realice una investigación de dos posibles aplicaciones donde se usen Variational Autoencoders (VAE) para la generación de algún tipo información.

Las redes neuronales basadas en una arquitectura Variational Autoencoders (VAE) son entrenadas con métodos no supervisados, y se subdividen en dos sub redes, una que comprime la información y las características de unas imágenes recibidas (encoder), que luego sirven de parámetros de entrada en una red subsiguiente generativa o decodificadora (decoder), que a partir de la data comprimida por el encoder, será capaz de construir la imagen igual al estado inicial antes de ser ingresada en la primera red, o con las modificaciones que deseemos. Adicionalmente, entre estas dos redes se encuentra otra arquitectura conocida como espacio latente, el cual representa el espacio mínimo de representación de la información en todo el autoencoder.

Ahora, la particularidad de los autoencoders variacionales con los originales explicados anteriormente, es que los normales generan un espacio latente no continuo, y por ende discreto, en donde las características extraídas por el encoder son un punto en específico de la imagen, y que por ende causa vacíos en el decoder cuando este intenta reconstruir regiones de la imagen que no poseen puntos en el espacio latente.

Y este problema lo solucionan los autoencoders variacionales [1], en los cuales, para inferir el espacio latente, se hace uso de distribuciones continuas de los datos, por lo cual estos ahora en vez de ser puntos separados, son regiones que ayudan al decoder a reconstruir más fácilmente las imágenes, ya que no hay porciones vacías del espacio latente.

Entre las posibles aplicaciones en donde se pudiera utilizar esta arquitectura para generar algún tipo de información destacan:

- **Reconstrucciones de imágenes a gusto:** como ahora se trabaja con regiones, es más fácil reconstruir las imágenes al placer del desarrollador, ya que se pueden editar un conjunto de características, en vez de puntos específicos, eliminar ruidos, entre muchas otras cosas que deseemos hacer con dichas imágenes, por lo cual a partir de un conjunto de imágenes, se pueden generar nuevas imágenes que combinen estas características de diferentes formas, llegando a crear así modelos no reales y por ende poco comunes [2]. Esta práctica se está haciendo más común en grandes industrias como la automotriz, en donde los motoristas y fabricantes del sector utilizan diseños completamente nuevos e innovadores creados por este tipo de red.
- **Conocer la distribución probabilística de la data de entrada:** estas redes, al usar distribuciones gaussianas para la creación de sus espacios latentes, son

capaces de identificar cualquier tipo de distribución probabilística en cualquier data set de entrada.

Ejercicio N°2

Realice una investigación de dos posibles aplicaciones donde se usen Redes Adversarias Generativas (GAN) para la generación de algún tipo información. ¿Cuál es la diferencia entre un VAE y una GAN cuando se aplica para la generación de información?

Las Redes Adversarias Generativas (GAN) son redes que constan de dos modelos, uno llamado Generador, que se encarga de crear imágenes a partir de ruido, que se parezcan lo más posible a las pertenecientes al data set original, y otro llamado Discriminador que tiene la tarea de, como su nombre lo indica, discriminar las imágenes falsas creadas con el generador, para identificar si esta es verdadera o falsa, en relación a si son como las imágenes del data set original, que no necesariamente son imágenes, ya que puede ser cualquier tipo de información, pero se usan las imágenes como ejemplo.

La principales diferencias entre las GAN's y las VAE's, es que las primeras generan imágenes a partir de datos randómicos, y ambos modelos que forman parte de ella, se retroalimentan constantemente y así el Generador crea información más certera y real, mientras que el Discriminador se va convirtiendo en un mejor detector de información falsa; mientras que en las VAE's, la creación de la nueva información se hace a partir de la compresión de las características de la data original en espacios latentes que sirvan de iniciativa para las generaciones.

Adicionalmente, hablando específicamente de imágenes, las GAN's suelen producirlas con mayor calidad y realismo que su contraparte, pero suelen ser más difíciles de trabajar, entrenar y manejar [3].

Entre dos posibles aplicaciones de las GAN's se pueden encontrar:

- **Generación de informaciones falsas:** estas redes son muy famosas entre los bots que falsican noticias, textos, imágenes, rostros, audios y lo que sea que pueda ser generado artificialmente por estas redes, que además suele ser contenido de muy alta calidad, por lo cual las masas son víctimas fáciles de los ataques de desinformación.
- **Traducciones:** son excelentes para la traducción de una imagen a otra imagen, imagen a emojis, o incluso de un texto a una imagen, entre muchas otras traducciones que son capaces de generar [4].

Ejercicio N°3

Realice una aplicación de clasificación utilizando Autoencoders apilados usando la base de datos MNIST

Para el ejercicio actual se utilizará el siguiente dataset:

<http://yann.lecun.com/exdb/mnist/>

Todo lo correspondiente a la solución del presente ejercicio se encontrará en el archivo Clasificacion_Autoencoders_MNIST_Carlos_Doffiny_SV.ipynb.

El problema de clasificación de imágenes trabajado con MNIST data set [6] consiste en unas 70000 imágenes de 28x28 píxeles en blanco y negro (por lo cual usa 1 canal gris) distribuidas en 10 clases que son los números escritos a mano del 0 al 9. En total, de las 70000 imágenes, 60000 de ellas son imágenes de entrenamiento, mientras que las 10000 restantes son imágenes de testeo, por lo cual no hace falta dividir el data set ya que ya se encuentra dividido.

El primer paso al iniciar el ejercicio, es descargar todos los imports necesarios, más el data set, como se observa en la Fig 1.

```

> Obtención y descarga del data set MNIST

Para el autoencoder solo se necesitarán las imágenes, mientras que para la clasificación si hacen falta los labels

[ ] [(x_train, y_train), (x_test, y_test)] = mnist.load_data() #Descargando el data set, y separando los datos

Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11493376/11490434 [=====] - 0s 0us/step
11501568/11490434 [=====] - 0s 0us/step

> Dimensión de los grupos de datos

[ ] #Mostramos la dimensión de los grupos de datos expresados en:
# (Número de registros, píxeles de largo, píxeles de ancho)
print('Dimensión de la las imágenes de entrenamiento: ', x_train.shape)
print('\nDimensión de la los labels de entrenamiento: ', y_train.shape)
print('\nDimensión de las imágenes de validación: ', x_test.shape)
print('\nDimensión de los labels de validación: ', y_test.shape)

Dimensión de la las imágenes de entrenamiento: (60000, 28, 28)

Dimensión de la los labels de entrenamiento: (60000,)

Dimensión de las imágenes de validación: (10000, 28, 28)

Dimensión de los labels de validación: (10000,)
```

Fig 1. Descargando el data set MNIST

Una vez obtenida esta data, se procede a verificar que la misma ha sido descargada sin problemas, imprimiendo las primeras 25 imágenes del data set de entrenamiento. Como se puede ver en la Fig 2.

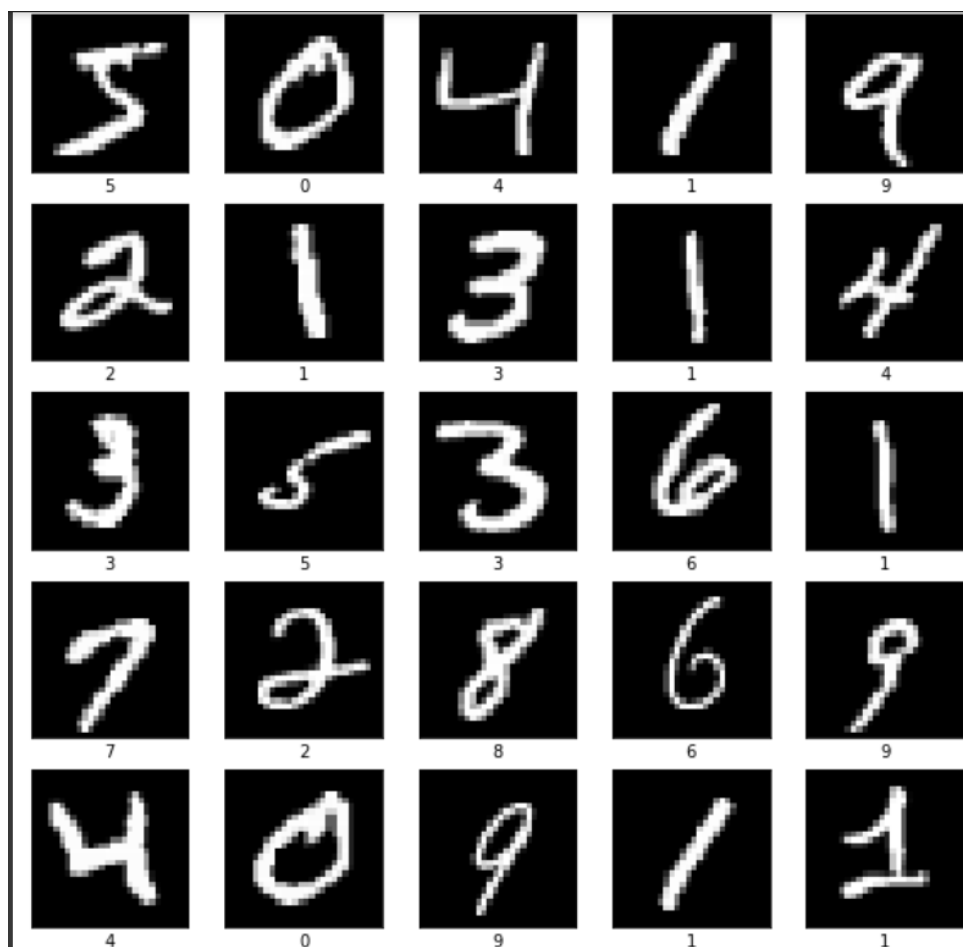


Fig 2. Primeras 25 imágenes del data set de entrenamiento.

Luego, sigue la normalización de los datos de entrada, el reshape de la data de entrada porque los autoencoders solo aceptan vectores unidimensionales, y el one hot encoding para la data de salida, con lo cual ya tendríamos la data preparada para calcular la dimensión de entrada, el número de clases, para iniciar la construcción de la arquitectura del modelo.

Ahora viene la parte interesante, que es diseñar y construir la arquitectura de la red neuronal convolucional que solucione el problema planteado. Para ello se usan 4 arquitecturas, 3 autoencoders, y luego una arquitectura formada por las capas latentes de cada uno de los autoencoders modelados y entrenados anteriormente, por lo cual cada una de estas capas presentes en la última red clasificatoria, ya tendrá sus pesos entrenados y listos para clasificar las imágenes según las 10 clases indicadas por el data set. Las arquitecturas

autoencoders las podemos observar en las Fig 3, 4 y 5, mientras que la de la red clasificadora con las capas latentes entrenadas las podemos observar en la Fig 6. Cabe destacar que aquí se ordenaron las capas latentes de la de mayor cantidad de neuronas, a la de menos, para simular el downsampling de una red convolucional, que son las usadas para la clasificación.

```
▼ Autoencoder #1

[ ] #Haremos ahora un autoencoder de 3 capas de encoder y 3 de decoder
input_img = Input(shape=(784)) #Encoded1
encoded1 = Dense(256, activation='relu')(input_img) #Encoded2
encoded1 = Dense(128, activation='relu')(encoded1) #Encoded3

encoded1 = Dense(64, activation='relu')(encoded1)#Latent layer

decoded1 = Dense(128, activation='relu')(encoded1) #Decoded1
decoded1 = Dense(256, activation='relu')(decoded1) #Decoded2
decoded1 = Dense(784, activation='sigmoid')(decoded1) #Decoded3
```

Fig 3. Arquitectura del primer autoencoder.

```
Autoencoder #2

[ ] #Haremos ahora un autoencoder de 3 capas de encoder y 3 de decoder
input_img = Input(shape=(784)) #Encoded1
encoded2 = Dense(128, activation='relu')(input_img) #Encoded2
encoded2 = Dense(64, activation='relu')(encoded2) #Encoded3

encoded2 = Dense(32, activation='relu')(encoded2)#Latent layer

decoded2 = Dense(64, activation='relu')(encoded2) #Decoded1
decoded2 = Dense(128, activation='relu')(decoded2) #Decoded2
decoded2 = Dense(784, activation='sigmoid')(decoded2) #Decoded3
```

Fig 4. Arquitectura del segundo autoencoder.

Autoencoder #3

```
[ ] #Haremos ahora un autoencoder de 3 capas de encoder y 3 de decoder
input_img = Input(shape=(784)) #Encoded1
encoded3 = Dense(64, activation= 'relu')(input_img) #Encoded2
encoded3 = Dense(32, activation= 'relu')(encoded3) #Encoded3

encoded3 = Dense(16, activation= 'relu')(encoded3) #Latent layer

decoded3 = Dense(32, activation= 'relu')(encoded3) #Decoded1
decoded3 = Dense(64, activation= 'relu')(decoded3) #Decoded2
decoded3 = Dense(784, activation= 'sigmoid')(decoded3) #Decoded3
```

Fig 5. Arquitectura del tercer autoencoder

Modelo clasificador

```
[ ] def classifier(latent_layer1,latent_layer2,latent_layer3):
    model = Sequential() #Creando la caja negra con arquitectura secuencial

    model.add(Flatten(input_shape=(28, 28)))
    model.add(Dense(128, activation='relu'))
    model.add(latent_layer1)
    model.add(latent_layer2)
    model.add(latent_layer3)
    model.add(Dense(num_class, activation = 'softmax'))

    model.summary() #Resumen del modelo

    model.compile(loss = 'categorical_crossentropy', optimizer = 'adam', metrics=['accuracy']) #Co
    # El optimizador que mejor funciona es el adam, y también usaremos métricas

    return model
```

Fig 6. Arquitectura del modelo de clasificación.

Una vez definida la arquitectura, se procede a entrenar la red neuronal, seguido de calcular su score para conocer la pérdida y el acierto tanto del entrenamiento como en la validación, con sus respectivas gráficas en la Fig 7, pero antes de eso, se tiene que hacer un reshape de la data, ya que en los autoencoders la data se entrena con vectores unidimensionales, mientras que en la arquitectura de clasificación, se entrena con matrices.

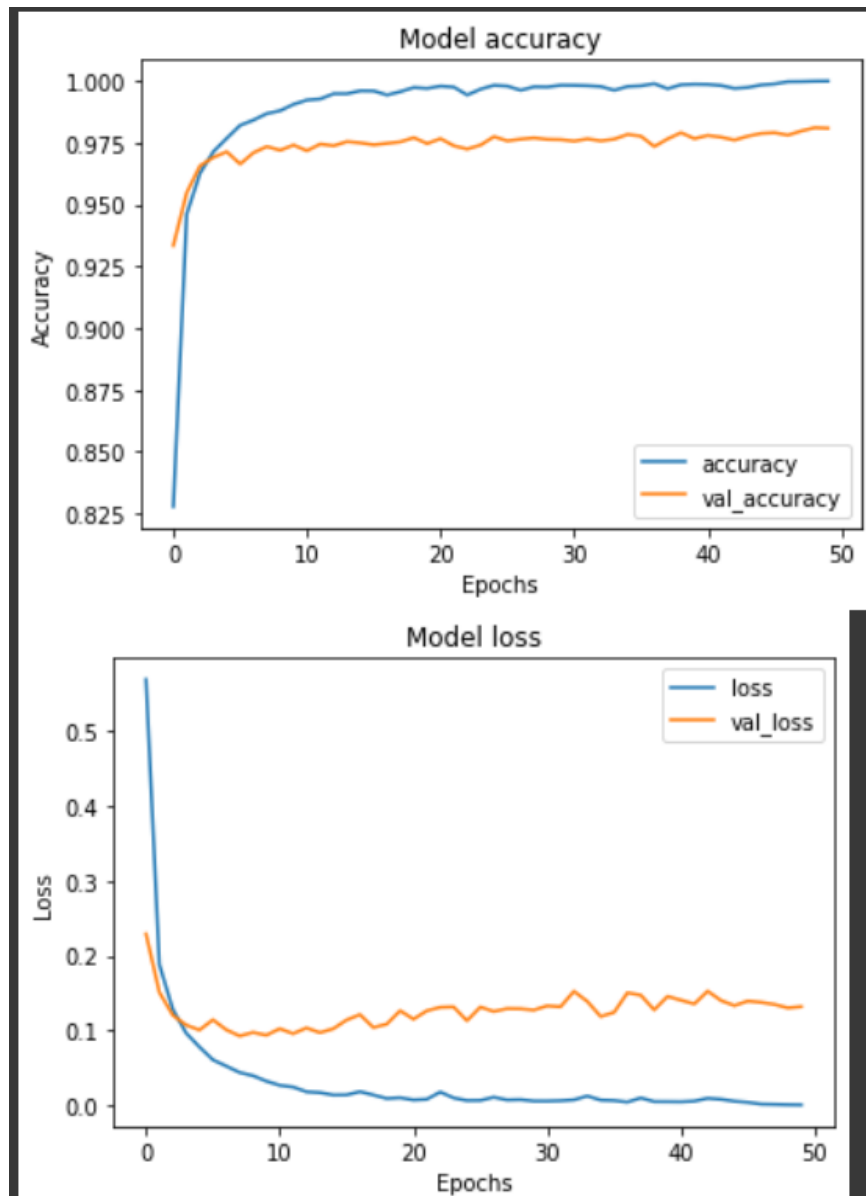


Fig 7. Gráficas de los scores para la data de entrenamiento en azul, y en amarillo la data de validación.

En este caso se observa como los valores de acierto y pérdida de entrenamiento y validación se van manteniendo estables, por lo cual si mejoramos algunos parámetros de la red, la misma podría ser mucho más certera aún. Pero lo positivo es que no se observa un sobre entrenamiento alarmante, considerando las pocas épocas con las que entrenó.

Para visualizar los resultados de la clasificación de las imágenes, se crean unas gráficas de barra para cada imagen que nos ayudan a representar los resultados predichos por la red, ejemplos que podemos ver en la Fig 8.

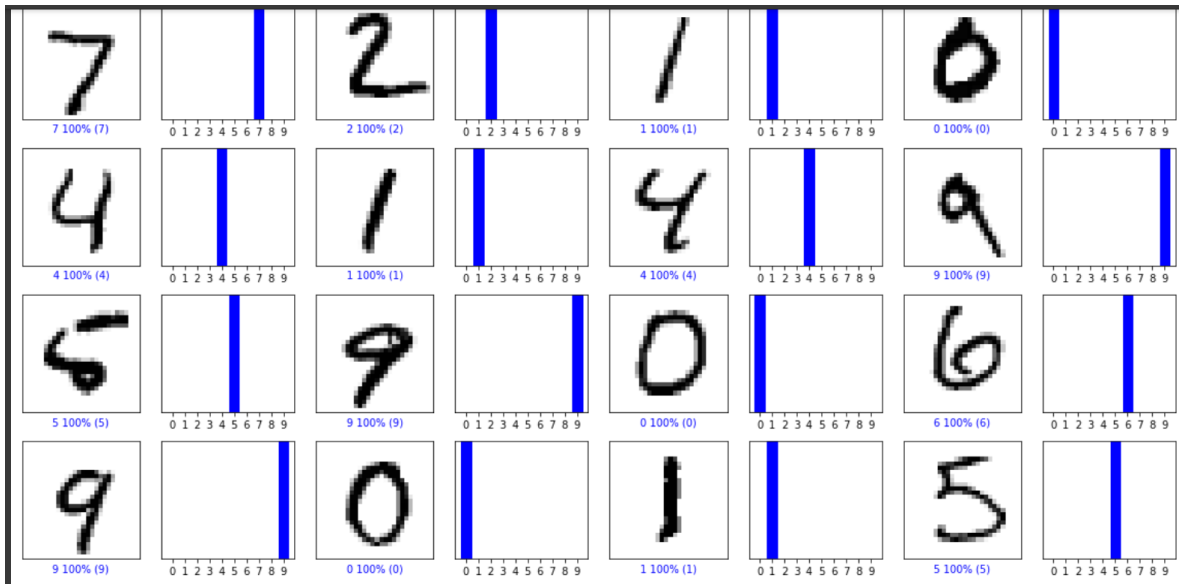


Fig 8. Observando la clasificación resultante de las imágenes.

Ahora, se quiere verificar el acierto de la red pero no en general, sino por clases, y esto se logra construyendo una matriz de confusión, que indicará la cantidad y porcentaje de datos acertados según lo esperado por clases, pero que también indicará a cuál clase fueron a parar aquellos resultados erróneos. Para ello, lo primero que se debe hacer es un model predict con la data de validación, que junto a la salida esperada de validación, generarán la matriz de confusión que indicará las cantidades de aciertos (Fig 9), y que gracias a una función dada en clases, se podrá transformarla en una de porcentaje de aciertos (Fig 10), dando por finalizado el presente ejercicio.

```

from sklearn.metrics import confusion_matrix #Import para la matriz de

cm = confusion_matrix(y_test, outputTest) #NO se pueden ingresar aquí
#real. En el segundo argumento, se extraerá la posición en la cual ese
print(cm)

[[ 972    0    0    1    1    0    2    0    3    1]
 [   0 1124    2    2    0    1    2    1    3    0]
 [   1    2 1011    2    2    1    2    5    6    0]
 [   0    0    1  993    0    2    0    4    5    5]
 [   1    1    2    0  965    0    4    3    1    5]
 [   2    0    0  10    1  872    2    1    3    1]
 [   4    2    2    1    6    6  936    1    0    0]
 [   2    1    6    2    1    0    0 1005    3    8]
 [   3    0    3    3    4    3    2    2  949    5]
 [   1    3    0    4    9    5    0    3    2  982]]

```

Fig 9. Matriz de confusión según cantidades de aciertos y errores.

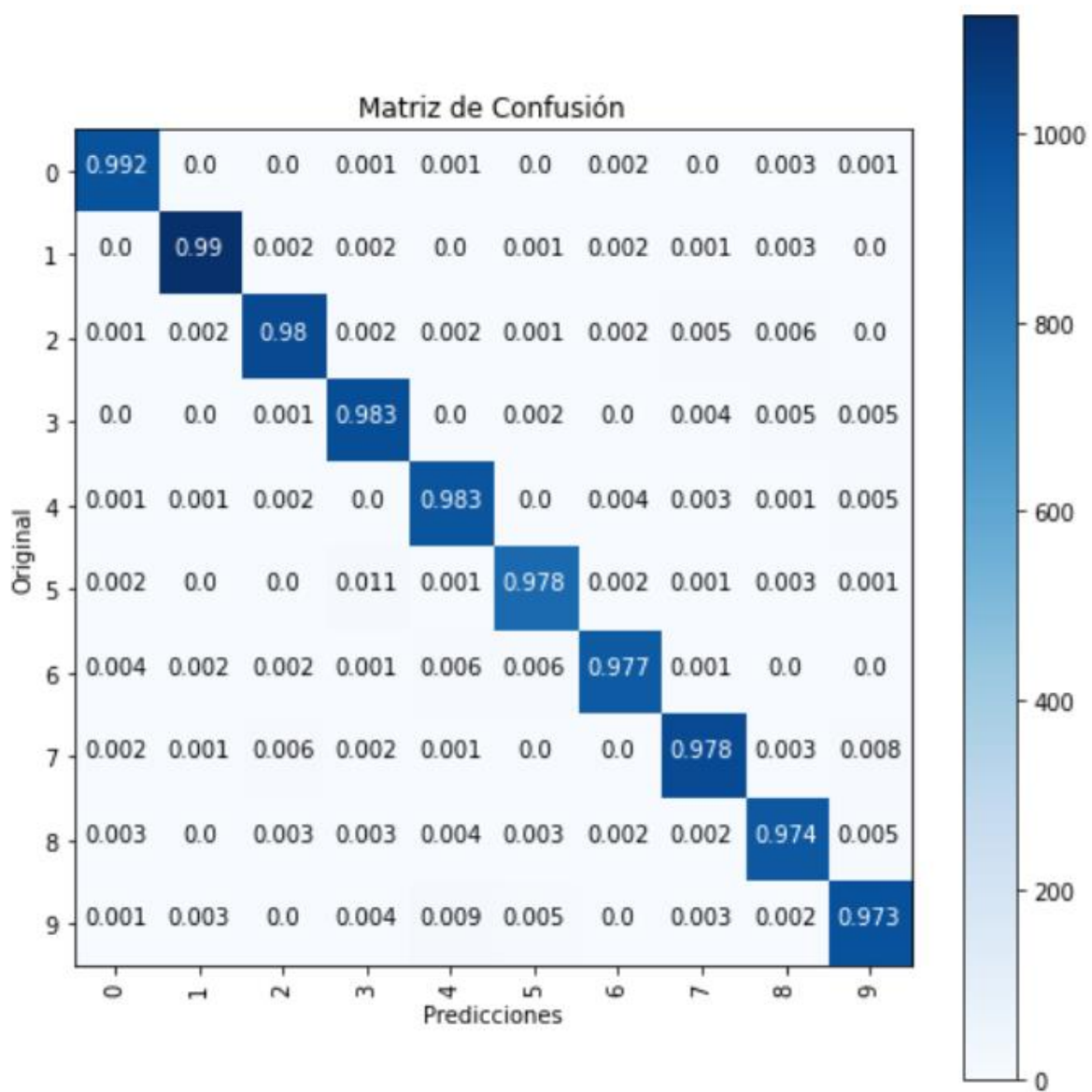


Fig 10. Matriz de confusión según porcentaje de aciertos y errores.

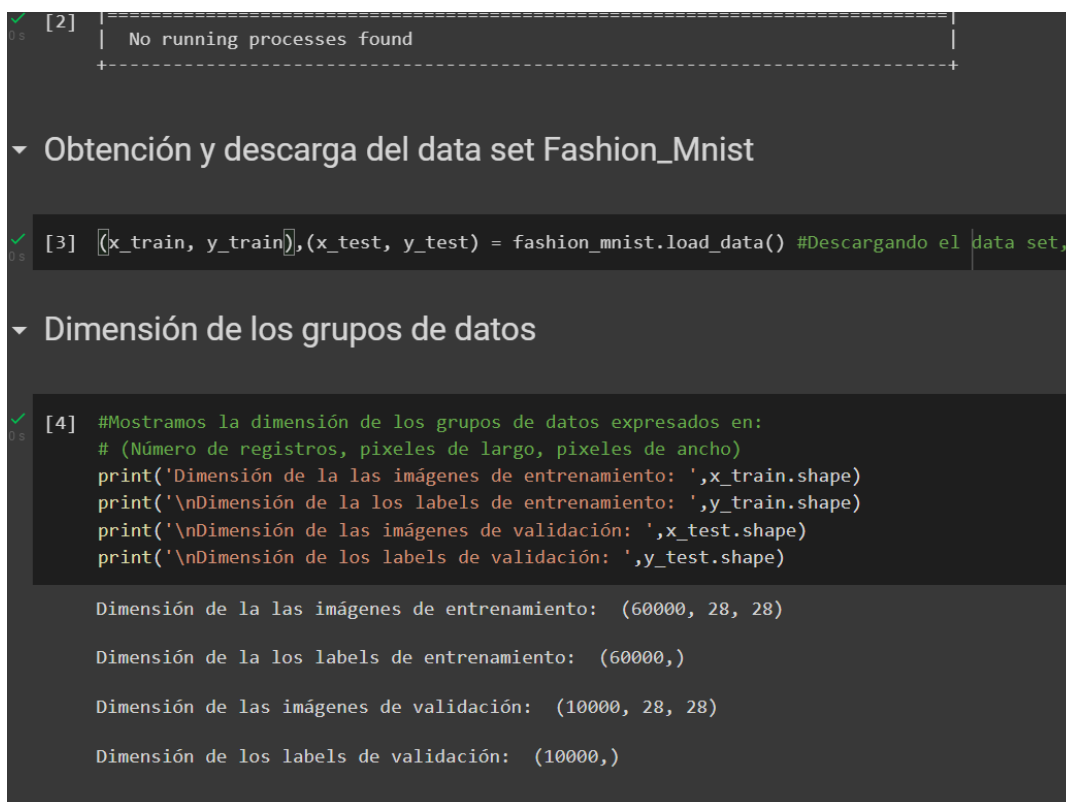
Ejercicio N°4

Entrene un Autoencoder para visualizar en dos dimensiones el comportamiento de la base de datos Fashion MNIST. Realice una visualización similar usando PCA (Análisis de Componentes Principales) y compare los resultados obtenidos. Adicionalmente, verifique la capacidad de generar nuevas imágenes con el Autoencoder entrenado

Todo lo correspondiente a la solución del presente ejercicio se encontrará en el archivo Variational_autoencoder_Fashion_MNIST_Carlos_Doffiny_SV.ipynb

El problema de clasificación de imágenes trabajado con Fashion MNIST data set [5] consiste en unas 70000 imágenes de 28x28 píxeles en blanco y negro (por lo cual usa 1 canal gris) distribuidas en 10 clases que son camisa, pantalón, suéter, vestido, abrigo, sandalia, franela, zapato tipo sneaker, botas y cartera. En total, de las 70000 imágenes, 60000 de ellas son imágenes de entrenamiento, mientras que las 10000 restantes son imágenes de testeo, por lo cual no hace falta dividir el data set ya que ya se encuentra dividido.

El primer paso al iniciar el ejercicio, es descargar todos los imports necesarios, más el data set, como se observa en la Fig 11



```
[2] | No running processes found  
+-----+  
▼ Obtención y descarga del data set Fashion_Mnist  
[3] (x_train, y_train), (x_test, y_test) = fashion_mnist.load_data() #Descargando el data set,  
▼ Dimensión de los grupos de datos  
[4] #Mostramos la dimensión de los grupos de datos expresados en:  
# (Número de registros, píxeles de largo, píxeles de ancho)  
print('Dimensión de la las imágenes de entrenamiento: ', x_train.shape)  
print('\nDimensión de la los labels de entrenamiento: ', y_train.shape)  
print('\nDimensión de las imágenes de validación: ', x_test.shape)  
print('\nDimensión de los labels de validación: ', y_test.shape)  
  
Dimensión de la las imágenes de entrenamiento: (60000, 28, 28)  
  
Dimensión de la los labels de entrenamiento: (60000,)  
  
Dimensión de las imágenes de validación: (10000, 28, 28)  
  
Dimensión de los labels de validación: (10000,)
```

Fig 11. Descargando el data set Fashion MNIST

Una vez obtenida esta data, se procede a verificar que la misma ha sido descargada sin problemas, imprimiendo las primeras 25 imágenes del data set de entrenamiento. Como se puede ver en la Fig 12.

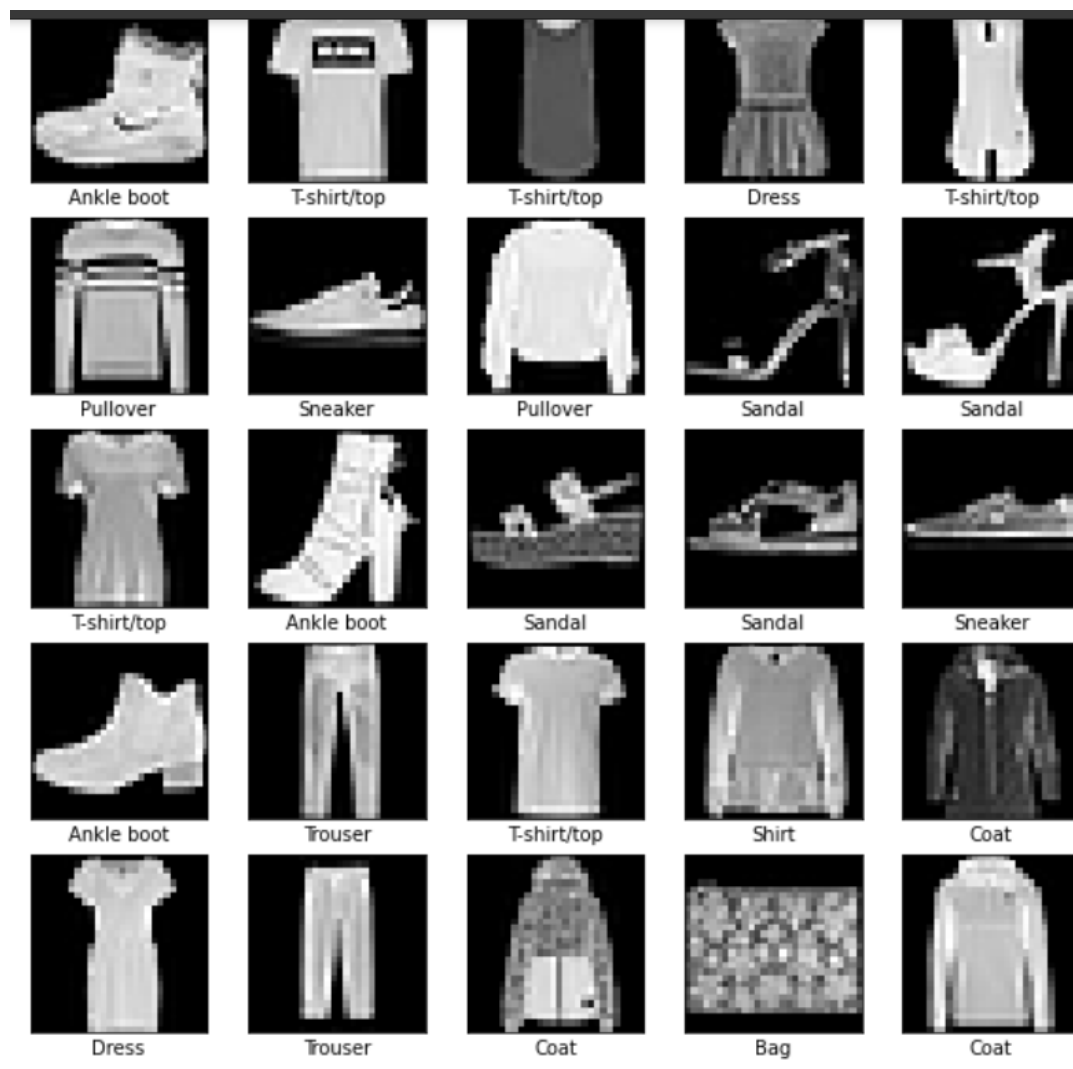


Fig 12. Primeras 25 imágenes del data set de entrenamiento.

Luego, sigue la normalización de los datos de entrada, el reshape de la data de entrada porque los autoencoders solo aceptan vectores unidimensionales, y el one hot encoding para la data de salida, con lo cual ya tendríamos la data preparada para calcular la dimensión de entrada, el número de clases, para iniciar la construcción de la arquitectura del modelo.

Ahora viene la creación de la arquitectura del modelo Autoencoder Variacional. Para ello se construye un encoder que termina en una capa latente bidimensional, que se debe construir con la función de la Fig 13[7]:

```
z = z_mean + exp(z_log_sigma) * epsilon,
```

Fig 13. Función de construcción de los puntos en el espacio latente.

Esta construcción la podemos ver reflejada en la Fig 14:

```

Construcción del encoder

original_dim = input_dim[0] #Dimensión de entrada
intermediate_dim = 64 #Dimensión de la capa intermedia entre la primera del encoder con la latente
latent_dim = 2 #Construiremos un espacio latente de dos dimensiones

inputs = keras.Input(shape=(original_dim,)) #Capa de entrada al encoder
h = layers.Dense(intermediate_dim, activation='relu')(inputs) #Capa intermedia
z_mean = layers.Dense(latent_dim)(h) #Primer parámetro de distribución de la capa latente
z_log_sigma = layers.Dense(latent_dim)(h) #Segundo parámetro de distribución de la capa latente

Construcción de los puntos del espacio latente

[14] def sampling(args): #Función para muestrear los puntos del espacio latente
    z_mean, z_log_sigma = args
    epsilon = K.random_normal(shape=(K.shape(z_mean)[0], latent_dim),
                              mean=0., stddev=0.1) #Epsilon es un tensor
    return z_mean + K.exp(z_log_sigma) * epsilon #Fórmula de formación de los puntos con distribución normal en el espacio latente

z = layers.Lambda(sampling)([z_mean, z_log_sigma]) #Construcción de los puntos en el espacio latente

```

Fig 14. Construcción del Encoder y de la capa latente.

Posteriormente se culmina la construcción del autoencoder creando el decoder y uniéndolo con el encoder, por lo cual se muestra un resumen de cada uno de los 3 modelos: encoder, decoder y autoencoder variacional.

Antes de configurar los parámetros del entrenamiento, se imprime la distribución normal de la data en el espacio latente, como se ve en la Fig 15, en donde se ve cómo todas las clases están compartiendo el mismo espacio, ya que la red aún no sabe diferenciar la región de cada una. Y para ello basta con hacer un model predict de la data de validación sin entrenar aún.

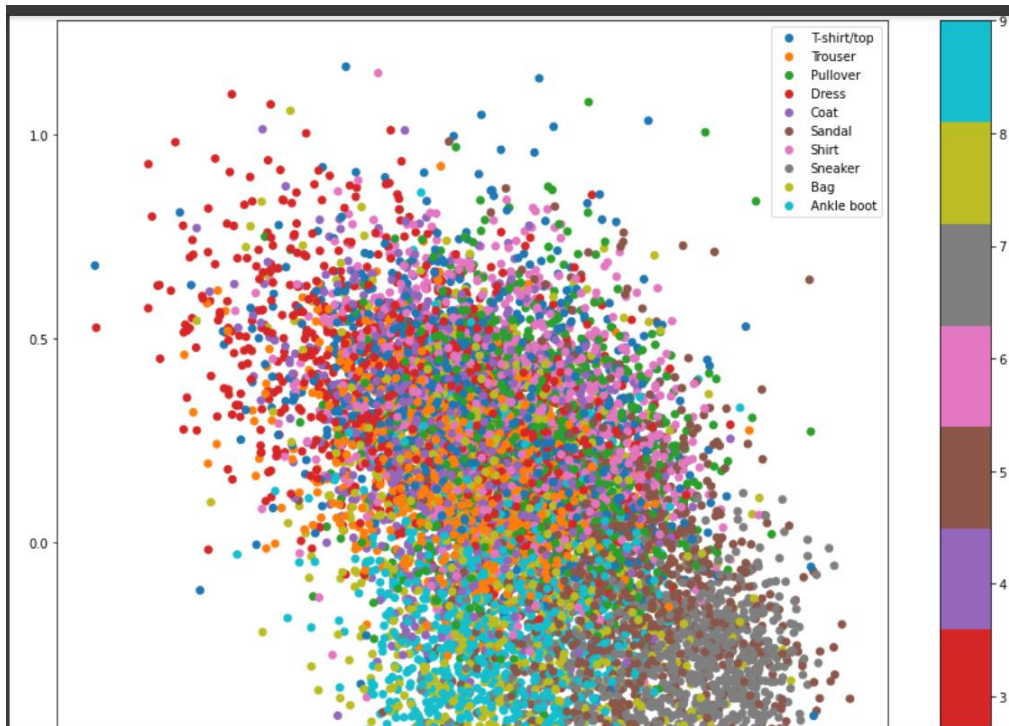


Fig 15. Distribución Normal de la data previa al entrenamiento.

Ahora se procede a configurar los parámetros de entrenamiento, con un optimizador Adam, un error crossentropy binario, 200 épocas y lotes de 64 imágenes, como se observa en la Fig 16.

Estableciendo el optimizador y función de pérdida del modelo

+ Código

+ Texto

```
[21] reconstruction_loss = keras.losses.binary_crossentropy(inputs, outputs) #La pérdida de reconstrucción del modelo
reconstruction_loss *= original_dim #Multiplicándolo por la dimensión de entrada
kl_loss = 1 + z_log_sigma - K.square(z_mean) - K.exp(z_log_sigma) #Pérdida del espacio latente
kl_loss = K.sum(kl_loss, axis=-1)
kl_loss *= -0.5
vae_loss = K.mean(reconstruction_loss + kl_loss) #La pérdida del modelo será la suma de la pérdida del encoder más el deco
vae.add_loss(vae_loss) #Asignando la pérdida del modelo
vae.compile(optimizer='adam') #Asignando el optimizador
```

Entrenamiento del Variational Autoencoder

```
[22] x_train, x_train,
epochs=200,
batch_size=64,
validation_data=(x_test, x_test)) #Asignando las épocas, tamaño del lote y data de entrenamiento y validación, la cual es
```

```
938/938 [=====] - 6s 7ms/step - loss: 253.1512 - val_loss: 256.2716
Epoch 173/200
938/938 [=====] - 5s 6ms/step - loss: 253.1130 - val_loss: 255.7685
Epoch 174/200
938/938 [=====] - 5s 6ms/step - loss: 253.0772 - val_loss: 255.7634
Epoch 175/200
938/938 [=====] - 6s 6ms/step - loss: 253.0931 - val_loss: 256.4754
Epoch 176/200
```

Fig 16. Configurando los parámetros del entrenamiento para luego ejecutarlo.

Una vez finalizado el entrenamiento de la red, se desea observar cómo ha cambiado la distribución de la data en el espacio latente, y se puede ver cómo ahora la red ha ido conociendo e identificando las regiones de cada clase, ya que vemos clases que están muy bien separadas como Bag, Trouser y Ankle Bot, mientras que hay otras que aún necesitan de más entrenamiento para crear regiones de características más definidas entre el pullover, coat y shirt por ejemplo. Esto se observa en la Fig 17, luego de realizar un model predict con la data luego del entrenamiento de la red, cuyas conexiones sinápticas del encoder ya están preparadas para formar estas regiones en el espacio latente.

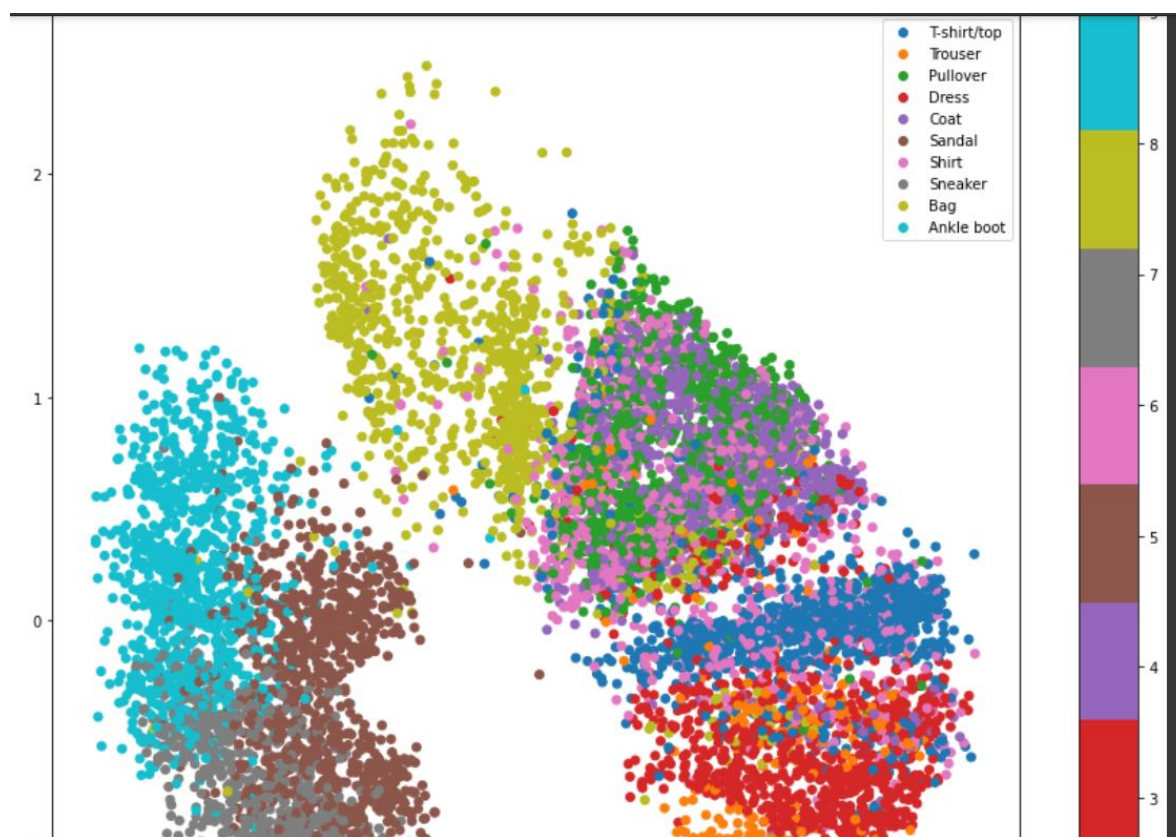


Fig 17. Distribución normal de la data luego del entrenamiento.

Así como el encoder está preparado y entrenado para separar e identificar las diferentes características de las imágenes y representarlas en el espacio latente, el decoder ya es capaz de a partir de este espacio, generar nuevas imágenes, a partir de la reconstrucción para la cual se fue entrenando. En la Fig 18 tenemos un conjunto de imágenes generadas por el decoder, ya que una red VAE es una red generadora también. Sin embargo, a esta red le falta muchísimo entrenamiento para que el encoder separe mejor las regiones en el espacio latente, y para que el decoder genere imágenes más efectivas. Aún así, se pueden distinguir algunos zapatos arriba a la derecha, pantalones arriba a la derecha, sandalias abajo a la izquierda y camisas abajo a la derecha.

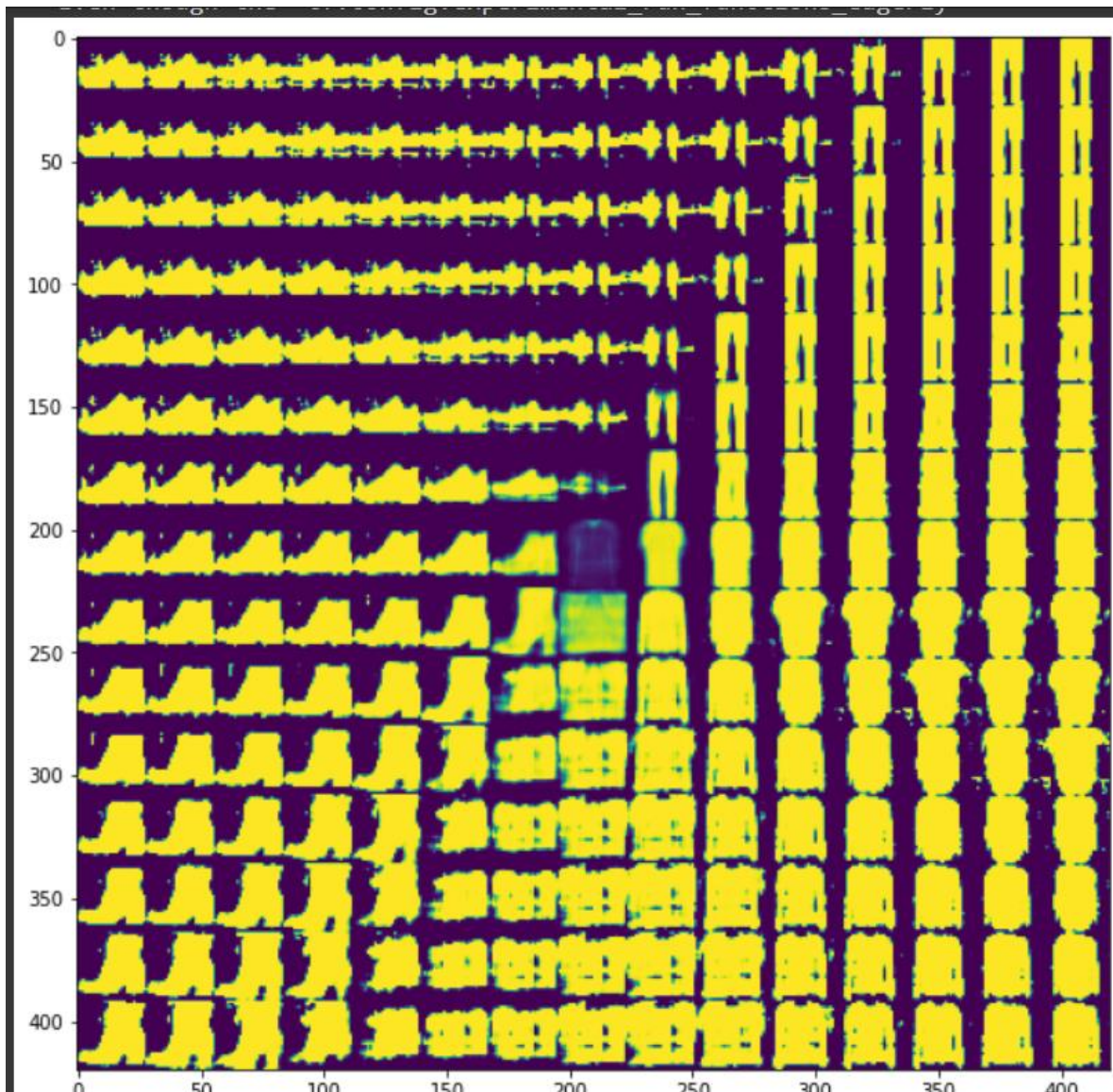


Fig 18. Conjunto de imágenes generadas por el decoder del Autoencoder Variacional.

Por último, se solicita que se compare la región generada por el autoencoder, con la generada por el Análisis de Componentes Principales (PCA), que se puede apreciar en la Fig 19. Ahí se ve como la distribución de los datos es mucho más lineal que con el autoencoder, y eso se debe a que el PCA solo tiene la capacidad de formar regiones lineales sin importar la naturaleza de los datos, ya que su modus operandi para resolver estos problemas es disminuyendo características, mientras que el autoencoder busca extraer las principales características de las imágenes ingresadas, sin importar la linealidad del problema, adaptándose mejor a la separación de las regiones correspondientes a las clases. Además,

vemos como con el PCA, las regiones están unas encima de las otras, con lo cual no está clara la separación de las clases.

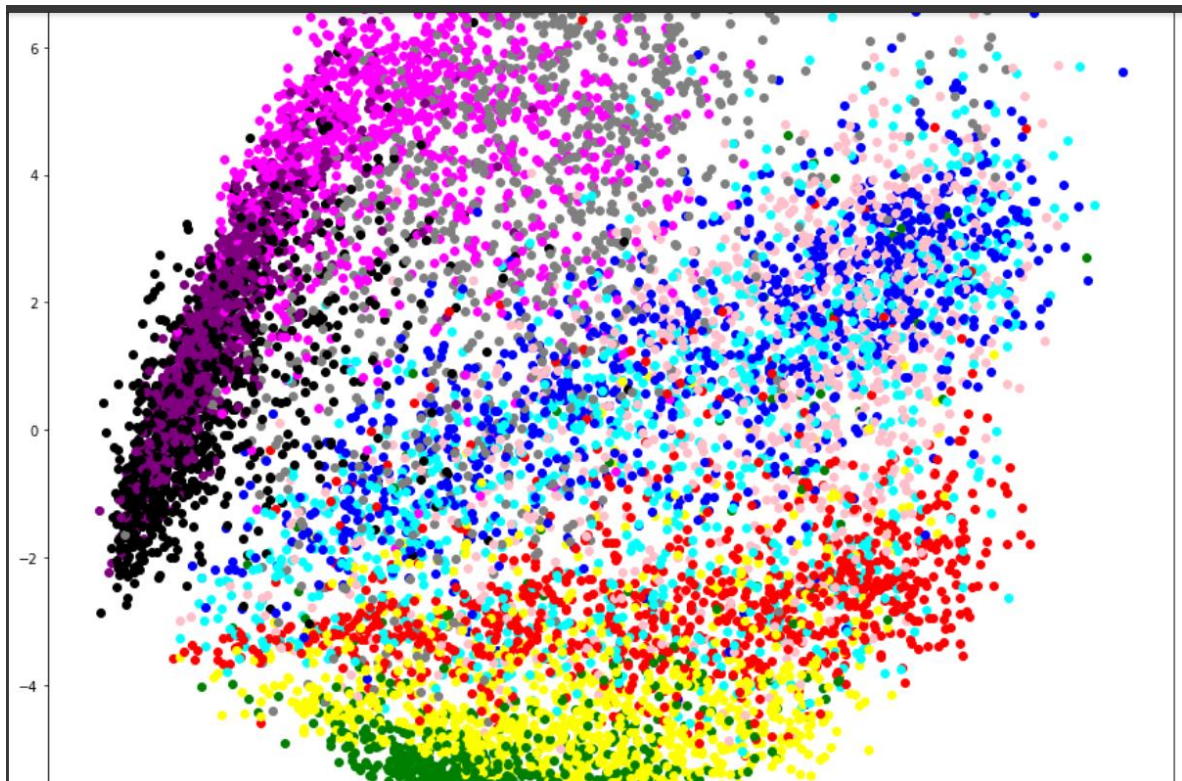


Fig 19. Distribución de la data con el Análisis de Componentes Principales (PCA).

Ejercicio N°5

Realice una aplicación de clasificación utilizando Autoencoders apilados usando la base de datos Fashion_MNIST

Para el ejercicio actual se utilizará el siguiente dataset:

<https://github.com/zalandoresearch/fashion-mnist/tree/master/data/fashion>

Todo lo correspondiente a la solución del presente ejercicio se encontrará en el archivo Clasificacion_Autoencoders_Fashion_MNIST_Carlos_Doffiny_SV.ipynb.

El problema de clasificación de imágenes trabajado con Fashion MNIST data set [5] consiste en unas 70000 imágenes de 28x28 píxeles en blanco y negro (por lo cual usa 1 canal gris) distribuidas en 10 clases que son camisa, pantalón, suéter, vestido, abrigo, sandalia, franela, zapato tipo sneaker, botas y cartera. En total, de las 70000 imágenes, 60000 de ellas son imágenes de entrenamiento, mientras que las 10000 restantes son imágenes de testeo, por lo cual no hace falta dividir el data set ya que ya se encuentra dividido.

El primer paso al iniciar el ejercicio, es descargar todos los imports necesarios, más el data set, como se observa en la Fig x

```
▼ Obtención y descarga del data set Fashion_Mnist
Para el autoencoder solo se necesitarán las imágenes, mientras que para la clasificación si hacen falta los labels

[ ] [(x_train, y_train), (x_test, y_test)] = fashion_mnist.load_data() #Descargando el data set, y separando los datos

▼ Dimensión de los grupos de datos

#Mostramos la dimensión de los grupos de datos expresados en:
# (Número de registros, píxeles de largo, píxeles de ancho)
print('Dimensión de la las imágenes de entrenamiento: ', x_train.shape)
print('\nDimensión de la los labels de entrenamiento: ', y_train.shape)
print('\nDimensión de las imágenes de validación: ', x_test.shape)
print('\nDimensión de los labels de validación: ', y_test.shape)

Dimensión de la las imágenes de entrenamiento: (60000, 28, 28)

Dimensión de la los labels de entrenamiento: (60000,)

Dimensión de las imágenes de validación: (10000, 28, 28)

Dimensión de los labels de validación: (10000,)
```

Fig x. Descargando el data set Fashion MNIST

Una vez obtenida esta data, se procede a verificar que la misma ha sido descargada sin problemas, imprimiendo las primeras 25 imágenes del data set de entrenamiento. Como se puede ver en la Fig XX.

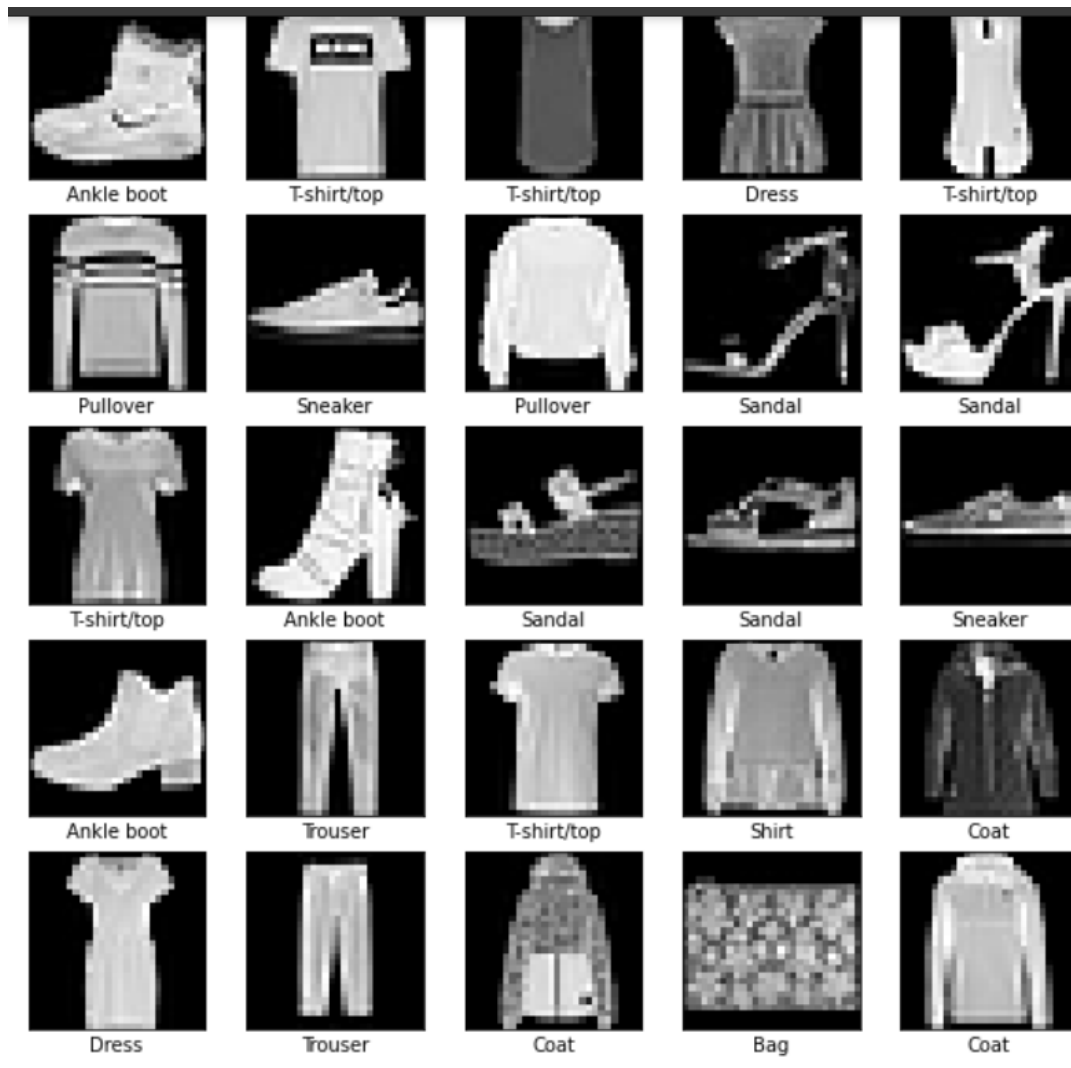


Fig XX. Primeras 25 imágenes del data set de entrenamiento.

Luego, sigue la normalización de los datos de entrada, el reshape de la data de entrada porque los autoencoders solo aceptan vectores unidimensionales, y el one hot encoding para la data de salida, con lo cual ya tendríamos la data preparada para calcular la dimensión de entrada, el número de clases, para iniciar la construcción de la arquitectura del modelo.

Ahora viene la parte interesante, que es diseñar y construir la arquitectura de la red neuronal convolucional que solucione el problema planteado. Para ello se usan 4 arquitecturas, 3 autoencoders, y luego una arquitectura formada por las capas latentes de cada uno de los autoencoders modelados y entrenados anteriormente, por lo cual cada una de estas capas presentes en la última red clasificatoria, ya tendrá sus pesos entrenados y listos para clasificar las imágenes según las 10 clases indicadas por el data set. Las arquitecturas autoencoders las podemos observar en las Fig XX, XX y XX, mientras que la de la red

clasificadora con las capas latentes entrenadas las podemos observar en la Fig XX. Cabe destacar que aquí se ordenaron las capas latentes de la de mayor cantidad de neuronas, a la de menos, para simular el downsampling de una red convolucional, que son las usadas para la clasificación.

```
▼ Autoencoder #1

[ ] #Haremos ahora un autoencoder de 3 capas de encoder y 3 de decoder
input_img = Input(shape=(784)) #Encoded1
encoded1 = Dense(256, activation= 'relu')(input_img) #Encoded2
encoded1 = Dense(128, activation= 'relu')(encoded1) #Encoded3

encoded1 = Dense(64, activation= 'relu')(encoded1)#Latent layer

decoded1 = Dense(128, activation= 'relu')(encoded1) #Decoded1
decoded1 = Dense(256, activation= 'relu')(decoded1) #Decoded2
decoded1 = Dense(784, activation= 'sigmoid')(decoded1) #Decoded3
```

Fig XX. Arquitectura del primer autoencoder.

```
Autoencoder #2

[ ] #Haremos ahora un autoencoder de 3 capas de encoder y 3 de decoder
input_img = Input(shape=(784)) #Encoded1
encoded2 = Dense(128, activation= 'relu')(input_img) #Encoded2
encoded2 = Dense(64, activation= 'relu')(encoded2) #Encoded3

encoded2 = Dense(32, activation= 'relu')(encoded2)#Latent layer

decoded2 = Dense(64, activation= 'relu')(encoded2) #Decoded1
decoded2 = Dense(128, activation= 'relu')(decoded2) #Decoded2
decoded2 = Dense(784, activation= 'sigmoid')(decoded2) #Decoded3
```

Fig XX. Arquitectura del segundo autoencoder.

Autoencoder #3

```
[ ] #Haremos ahora un autoencoder de 3 capas de encoder y 3 de decoder
input_img = Input(shape=(784)) #Encoded1
encoded3 = Dense(64, activation= 'relu')(input_img) #Encoded2
encoded3 = Dense(32, activation= 'relu')(encoded3) #Encoded3

encoded3 = Dense(16, activation= 'relu')(encoded3) #Latent layer

decoded3 = Dense(32, activation= 'relu')(encoded3) #Decoded1
decoded3 = Dense(64, activation= 'relu')(decoded3) #Decoded2
decoded3 = Dense(784, activation= 'sigmoid')(decoded3) #Decoded3
```

Fig XX. Arquitectura del tercer autoencoder

Modelo clasificador

```
[ ] def classifier(latent_layer1,latent_layer2,latent_layer3):
    model = Sequential() #Creando la caja negra con arquitectura secuencial

    model.add(Flatten(input_shape=(28, 28)))
    model.add(Dense(128, activation='relu'))
    model.add(latent_layer1)
    model.add(latent_layer2)
    model.add(latent_layer3)
    model.add(Dense(num_class, activation = 'softmax'))

    model.summary() #Resumen del modelo

    model.compile(loss = 'categorical_crossentropy', optimizer = 'adam', metrics=['accuracy']) #Compilamos el modelo
    # El optimizador que mejor funciona es el adam, y también usaremos métricas

    return model
```

Fig XX. Arquitectura del modelo de clasificación.

Una vez definida la arquitectura, se procede a entrenar la red neuronal, seguido de calcular su score para conocer la pérdida y el acierto tanto del entrenamiento como en la validación, con sus respectivas gráficas en la Fig XX, pero antes de eso, se tiene que hacer un reshape de la data, ya que en los autoencoders la data se entrena con vectores unidimensionales, mientras que en la arquitectura de clasificación, se entrena con matrices.

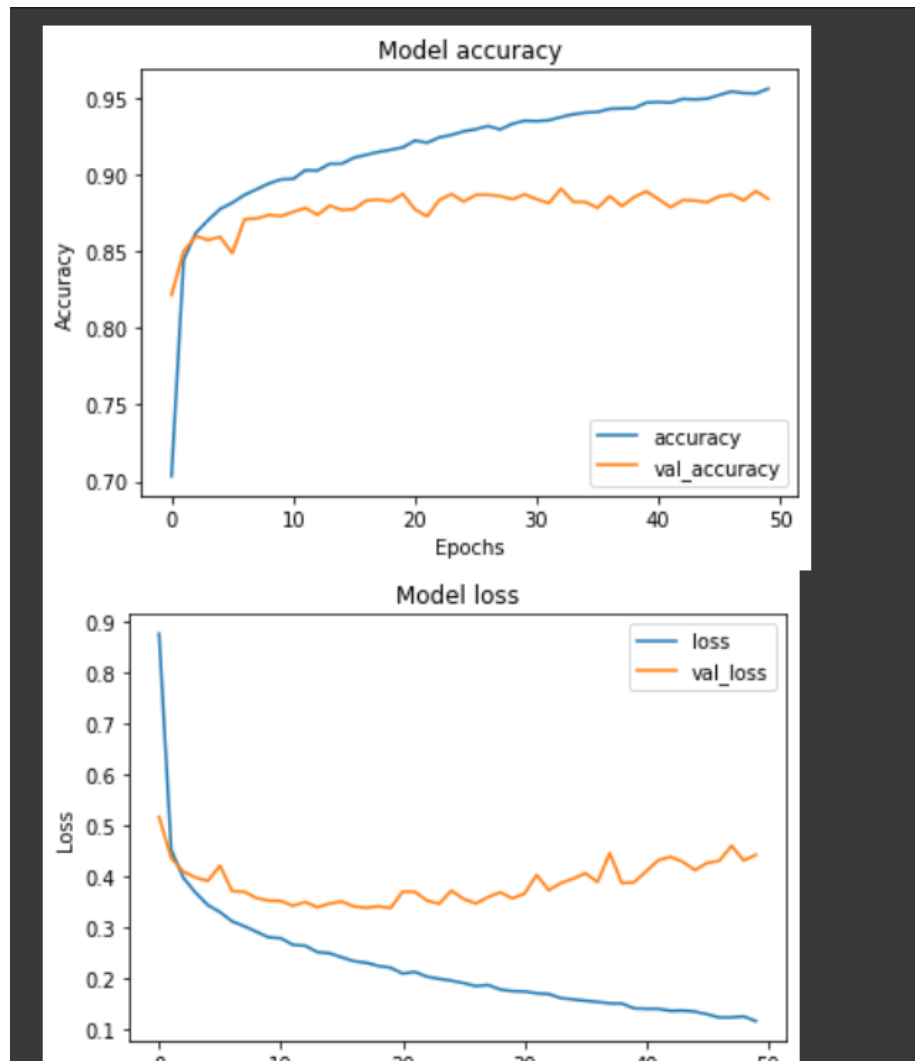


Fig XX. Gráficas de los scores para la data de entrenamiento en azul, y en amarillo la data de validación.

En este caso se observa como los valores de acierto y pérdida de entrenamiento y validación se van separando poco a poco, por lo cual la red empieza a sufrir de sobreentrenamiento, problema que se podría solucionar modificando ciertos parámetros de las diferentes redes.

Para visualizar los resultados de la clasificación de las imágenes, se crean unas gráficas de barra para cada imagen que nos ayudan a representar los resultados predichos por la red, ejemplos que podemos ver en la Fig XX.

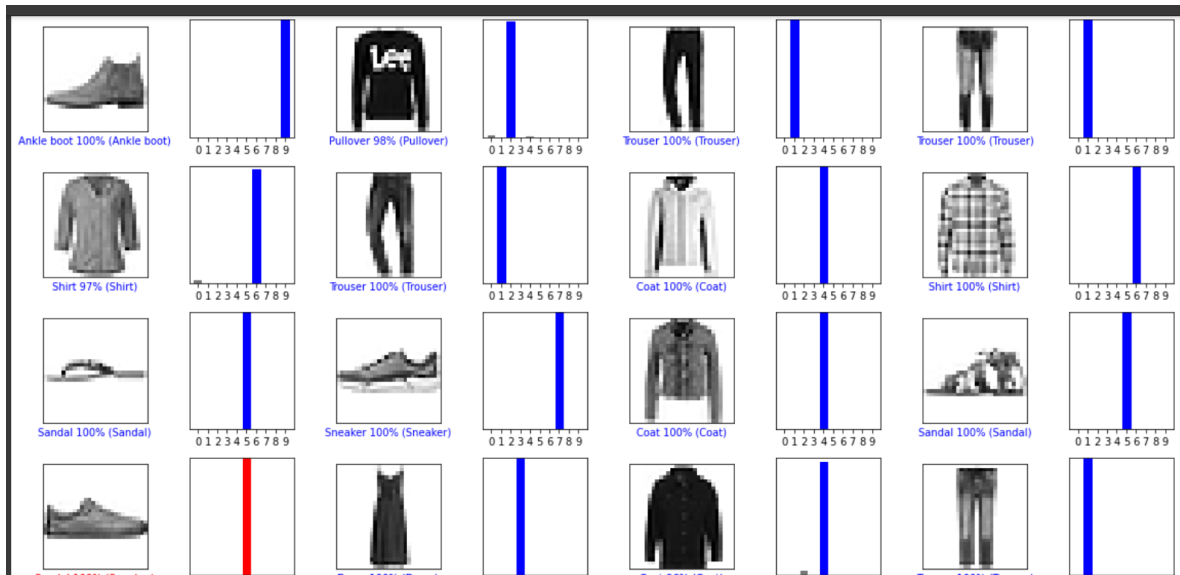


Fig XX. Observando la clasificación resultante de las imágenes.

Ahora, se quiere verificar el acierto de la red pero no en general, sino por clases, y esto se logra construyendo una matriz de confusión, que indicará la cantidad y porcentaje de datos acertados según lo esperado por clases, pero que también indicará a cuál clase fueron a parar aquellos resultados erróneos. Para ello, lo primero que se debe hacer es un model predict con la data de validación, que junto a la salida esperada de validación, generarán la matriz de confusión que indicará las cantidades de aciertos (Fig XX), y que gracias a una función dada en clases, se podrá transformarla en una de porcentaje de aciertos (Fig XX), dando por finalizado el presente ejercicio.

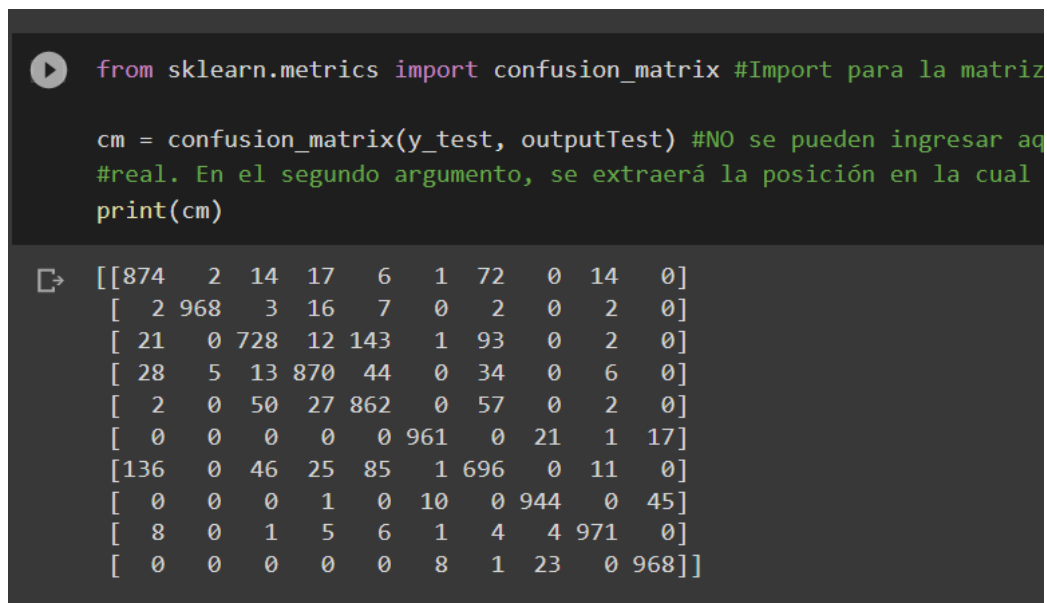


Fig XX. Matriz de confusión según cantidades de aciertos y errores.

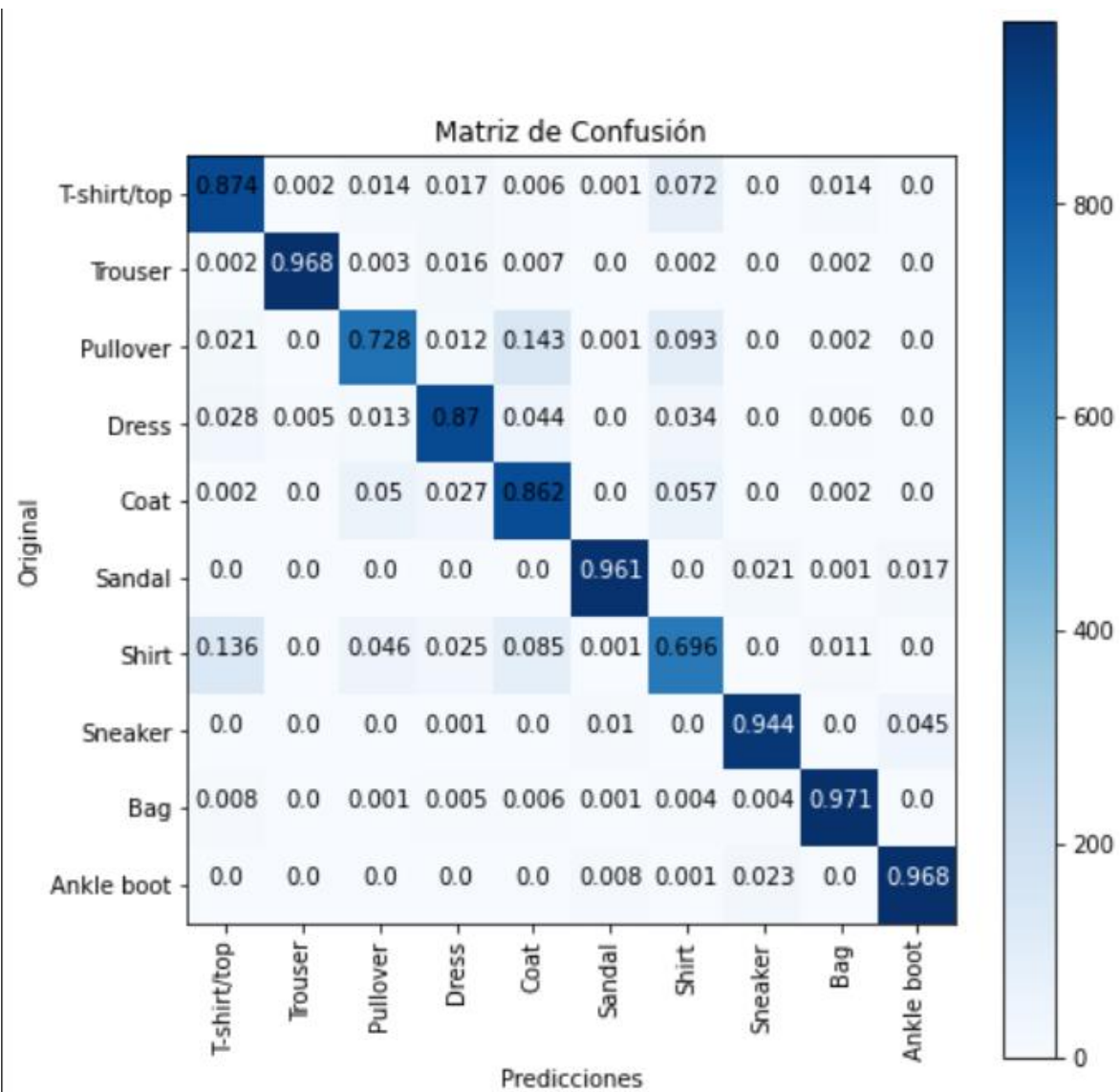


Fig XX. Matriz de confusión según porcentaje de aciertos y errores.

Referencias Bibliográficas

- [1] Rubén Cañadas (2021). ¿Qué son los autoencoders?. 12 de mayo de 2022, de Abdatum. Sitio web: <https://abdatum.com/machine-learning/autoencoders>
- [2] Oscar Contreras Carrasco (2021). Modelos Generativos en el Aprendizaje Automático y su aplicación a la generación de Imágenes Digitales. 12 de mayo de 2022, de Universidad Privada del Valle. Sitio web: <https://revistas.univalle.edu/index.php/ciencias/article/view/110>
- [3] Brian O'Donnell (2021). What are the fundamental differences between VAE and GAN for image generation? 12 de mayo de 2022, de Stack Exchange. Sitio web: <https://ai.stackexchange.com/questions/25601/what-are-the-fundamental-differences-between-vae-and-gan-for-image-generation>
- [4] Jason Brownlee (2019). 18 Impressive Applications of Generative Adversarial Networks (GANs). 12 de mayo de 2022, de Abdatum. Sitio web: <https://machinelearningmastery.com/impressive-applications-of-generative-adversarial-networks/>
- [5] zaladoresearch (2017). Fashion-mnist. 13 de mayo de 2022, de Github. Sitio web: <https://github.com/zaladoresearch/fashion-mnist/tree/master/data/fashion>
- [6] Yann LeCun, Corinna Cortes, Christopher Burges (1998). Mnist Database. 13 de mayo de 2022, de Yann LeCun. Sitio web: <http://yann.lecun.com/exdb/mnist/>
- [7] Francois Chollet (2016). Building Autoencoders in Keras. 15 de mayo de 2022, de The Keras Blog. Sitio web: <https://blog.keras.io/building-autoencoders-in-keras.html>

