

Actividad 1: Conceptos generales de redes neuronales

En esta actividad vamos a revisar algunos de los conceptos basicos de las redes neuronales, pero no por ello menos importantes.

El dataset a utilizar es Fashion MNIST, un problema sencillo con imágenes pequeñas de ropa, pero más interesante que el dataset de MNIST. Puedes consultar más información sobre el dataset en este enlace.

El código utilizado para contestar tiene que quedar claramente reflejado en el Notebook. Puedes crear nuevas cells si así lo deseas para estructurar tu código y sus salidas. A la hora de entregar el notebook, asegúrate de que los resultados de ejecutar tu código han quedado guardados (por ejemplo, a la hora de entrenar una red neuronal tiene que verse claramente un log de los resultados de cada epoch).

```
In [2]: import tensorflow as tf  
print(tf.__version__)
```

2.16.2

En primer lugar vamos a importar el dataset Fashion MNIST (recordad que este es uno de los dataset de entranamiento que estan guardados en keras) que es el que vamos a utilizar en esta actividad:

```
In [3]: mnist = tf.keras.datasets.fashion_mnist
```

Llamar a **load_data** en este dataset nos dará dos conjuntos de dos listas, estos serán los valores de entrenamiento y prueba para los gráficos que contienen las prendas de vestir y sus etiquetas.

Nota: Aunque en esta actividad lo veis de esta forma, también lo vais a poder encontrar como 4 variables de esta forma: training_images, training_labels, test_images, test_labels = mnist.load_data()

```
In [4]: (training_images, training_labels), (test_images, test_labels) = mnist.load_
```

Antes de continuar vamos a dar un vistazo a nuestro dataset, para ello vamos a ver una imagen de entrenamiento y su etiqueta o clase.

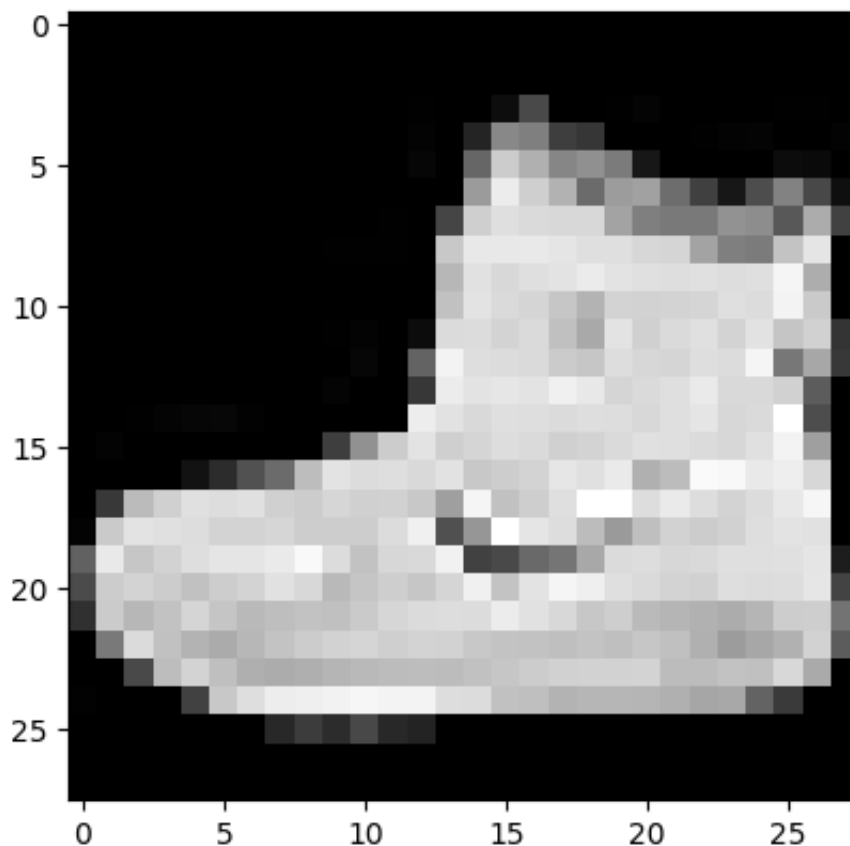
```
In [5]: import numpy as np
np.set_printoptions(linewidth=200)
import matplotlib.pyplot as plt
plt.imshow(training_images[0], cmap="gray") # recordad que siempre es prefer
#
print(training_labels[0])
print(training_images[0])
```

```
9
[[ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  1  0  0  13  73  0
 0  1  4  0  0  0  0  1  1  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  3  0  36 136 127  62  5
 4  0  0  0  1  3  4  0  0  3]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  6  0 102 204 176 134 14
 4 123 23  0  0  0  0 12 10  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0 155 236 207 178 10
 7 156 161 109  64  23  77 130  72 15]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  1  0  69 207 223 218 216 21
 6 163 127 121 122 146 141  88 172 66]
 [ 0  0  0  0  0  0  0  0  0  0  1  1  1  0 200 232 232 233 229 22
 3 223 215 213 164 127 123 196 229  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0 183 225 216 223 228 23
 5 227 224 222 224 221 223 245 173  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0 193 228 218 213 198 18
 0 212 210 211 213 223 220 243 202  0]
 [ 0  0  0  0  0  0  0  0  0  0  1  3  0 12 219 220 212 218 192 16
 9 227 208 218 224 212 226 197 209 52]
 [ 0  0  0  0  0  0  0  0  0  0  0  6  0 99 244 222 220 218 203 19
 8 221 215 213 222 220 245 119 167 56]
 [ 0  0  0  0  0  0  0  0  0  0  4  0  0 55 236 228 230 228 240 23
 2 213 218 223 234 217 217 209  92  0]
 [ 0  0  1  4  6  7  2  0  0  0  0  0  0 237 226 217 223 222 219 22
 2 221 216 223 229 215 218 255  77  0]
 [ 0  3  0  0  0  0  0  0  0  0 62 145 204 228 207 213 221 218 208 21
 1 218 224 223 219 215 224 244 159  0]
 [ 0  0  0  0 18 44 82 107 189 228 220 222 217 226 200 205 211 230 22
 4 234 176 188 250 248 233 238 215  0]
 [ 0 57 187 208 224 221 224 208 204 214 208 209 200 159 245 193 206 223 25
 5 255 221 234 221 211 220 232 246  0]
 [ 3 202 228 224 221 211 211 214 205 205 205 220 240  80 150 255 229 221 18
 8 154 191 210 204 209 222 228 225  0]
 [ 98 233 198 210 222 229 229 234 249 220 194 215 217 241  65  73 106 117 16
 8 219 221 215 217 223 223 224 229 29]
 [ 75 204 212 204 193 205 211 225 216 185 197 206 198 213 240 195 227 245 23
```

```

9 223 218 212 209 222 220 221 230 67]
[ 48 203 183 194 213 197 185 190 194 192 202 214 219 221 220 236 225 216 19
9 206 186 181 177 172 181 205 206 115]
[ 0 122 219 193 179 171 183 196 204 210 213 207 211 210 200 196 194 191 19
5 191 198 192 176 156 167 177 210 92]
[ 0 0 74 189 212 191 175 172 175 181 185 188 189 188 193 198 204 209 21
0 210 211 188 188 194 192 216 170 0]
[ 2 0 0 0 66 200 222 237 239 242 246 243 244 221 220 193 191 179 18
2 182 181 176 166 168 99 58 0 0]
[ 0 0 0 0 0 0 0 0 40 61 44 72 41 35 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0]
[ 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0]
[ 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0]]

```



Habreis notado que todos los valores numericos están entre 0 y 255. Si estamos entrenando una red neuronal, una buena practica es transformar todos los valores entre 0 y 1, un proceso llamado "normalización" y afortunadamente en Python es fácil normalizar una lista. Lo puedes hacer de esta manera:

```

In [6]: training_images = training_images / 255.0
        test_images = test_images / 255.0

```

Ahora vamos a definir el modelo, pero antes vamos a repasar algunos comandos y conceptos muy utiles:

- **Sequential:** Eso define una SECUENCIA de capas en la red neuronal
- **Dense:** Añade una capa de neuronas
- **Flatten:** ¿Recuerdas que las imágenes cómo eran las imagenes cuando las imprimiste para poder verlas? Un cuadrado, Flatten sólo toma ese cuadrado y lo convierte en un vector de una dimensión.

Cada capa de neuronas necesita una función de activación. Normalmente se usa la función relu en las capas intermedias y softmax en la ultima capa

- **Relu** significa que "Si $X > 0$ devuelve X , si no, devuelve 0", así que lo que hace es pasar sólo valores 0 o mayores a la siguiente capa de la red.
- **Softmax** toma un conjunto de valores, y escoge el más grande.

Pregunta 1 (3.5 puntos). Utilizando Keras, y preparando los datos de X e y como fuera necesario, define y entrena una red neuronal que sea capaz de clasificar imágenes de Fashion MNIST con las siguientes características:

- Una hidden layer de tamaños 128, utilizando unidades sigmoid Optimizador Adam.
- Durante el entrenamiento, la red tiene que mostrar resultados de loss y accuracy por cada epoch.
- La red debe entrenar durante 10 epochs y batch size de 64.
- La última capa debe de ser una capa softmax.
- Tu red tendría que ser capaz de superar fácilmente 80% de accuracy.

```
In [7]: ### Tu código para la red neuronal de la pregunta 1 aquí ###
# 1. Preparación de datos
print("Cargando el dataset Fashion MNIST...")
fashion_mnist = tf.keras.datasets.fashion_mnist
(training_images, training_labels), (test_images, test_labels) = fashion_mnist.load_data()

# Normalización de imágenes (convertir valores de píxeles de 0-255 a 0-1)
print("\nNormalizando las imágenes...")
training_images = training_images / 255.0
test_images = test_images / 255.0

# 2. Crear el modelo
print("\nCreando el modelo...")
model = tf.keras.models.Sequential([
    # Capa para aplanar la imagen de 28x28 a un vector de 784 elementos
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    # Capa oculta con 128 neuronas y activación sigmoid
```

```
tf.keras.layers.Dense(128, activation='sigmoid'),
# Capa de salida con 10 neuronas (una por clase) y softmax
tf.keras.layers.Dense(10, activation='softmax')
])

# 3. Compilar el modelo
print("\nCompilando el modelo...")
model.compile(
    optimizer='adam',
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)

# 4. Entrenar el modelo
print("\nComenzando el entrenamiento...")
history = model.fit(
    training_images,
    training_labels,
    epochs=10,
    batch_size=64,
    validation_data=(test_images, test_labels),
    verbose=1 # Mostrar barra de progreso
)

# 5. Mostrar resultados finales
print("\nResultados finales:")
print(f"Precisión en entrenamiento: {history.history['accuracy'][-1]:.4f}")
print(f"Precisión en validación: {history.history['val_accuracy'][-1]:.4f}")
```

Cargando el dataset Fashion MNIST...


Normalizando las imágenes...

Creando el modelo...


Compilando el modelo...

Comenzando el entrenamiento...


Epoch 1/10

938/938  **9s** 9ms/step - accuracy: 0.7434 - loss: 0.8286 - val_accuracy: 0.8318 - val_loss: 0.4675


Epoch 2/10

938/938  **8s** 8ms/step - accuracy: 0.8496 - loss: 0.4233 - val_accuracy: 0.8417 - val_loss: 0.4327


Epoch 3/10

938/938  **8s** 8ms/step - accuracy: 0.8636 - loss: 0.3800 - val_accuracy: 0.8546 - val_loss: 0.4058


Epoch 4/10

938/938  **8s** 8ms/step - accuracy: 0.8705 - loss: 0.3550 - val_accuracy: 0.8643 - val_loss: 0.3812

Epoch 5/10

938/938  **8s** 8ms/step - accuracy: 0.8788 - loss: 0.3397 - val_accuracy: 0.8720 - val_loss: 0.3635


Epoch 6/10

938/938  **8s** 8ms/step - accuracy: 0.8890 - loss: 0.3107 - val_accuracy: 0.8655 - val_loss: 0.3673


Epoch 7/10

938/938  **8s** 8ms/step - accuracy: 0.8880 - loss: 0.3068 - val_accuracy: 0.8765 - val_loss: 0.3487


Epoch 8/10

938/938  **8s** 8ms/step - accuracy: 0.8948 - loss: 0.2874 - val_accuracy: 0.8755 - val_loss: 0.3510

Epoch 9/10

938/938  **8s** 8ms/step - accuracy: 0.8966 - loss: 0.2801 - val_accuracy: 0.8665 - val_loss: 0.3584

Epoch 10/10

938/938  **8s** 8ms/step - accuracy: 0.9016 - loss: 0.2692 - val_accuracy: 0.8723 - val_loss: 0.3474

Resultados finales:

Precisión en entrenamiento: 0.9014

Precisión en validación: 0.8723

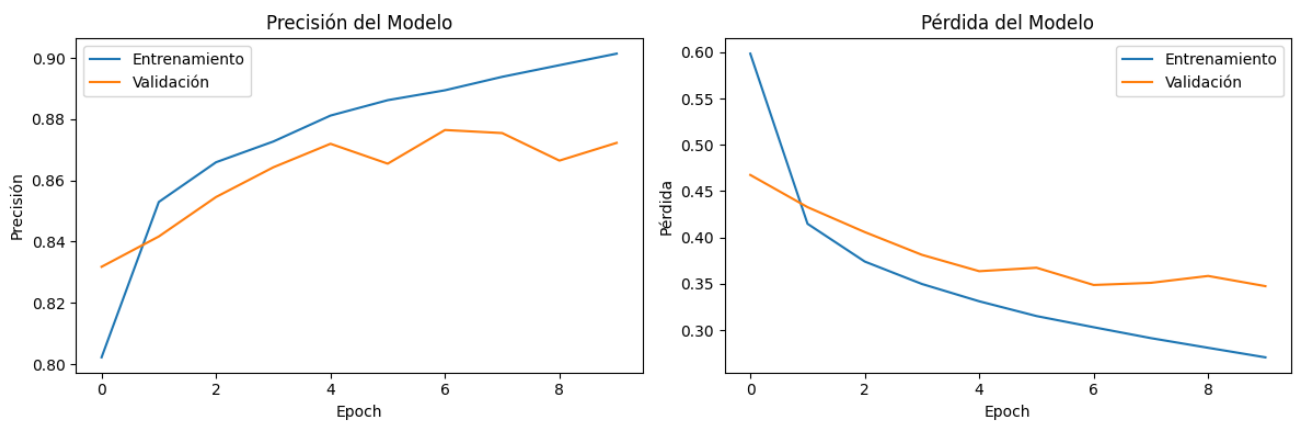
```
In [8]: # Visualizar el entrenamiento
plt.figure(figsize=(12, 4))

# Gráfica de precisión
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Entrenamiento')
```

```
plt.plot(history.history['val_accuracy'], label='Validación')
plt.title('Precisión del Modelo')
plt.xlabel('Epoch')
plt.ylabel('Precisión')
plt.legend()

# Gráfica de pérdida
plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Entrenamiento')
plt.plot(history.history['val_loss'], label='Validación')
plt.title('Pérdida del Modelo')
plt.xlabel('Epoch')
plt.ylabel('Pérdida')
plt.legend()

plt.tight_layout()
plt.show()
```



Para concluir el entrenamiento de la red neuronal, una buena practica es evaluar el modelo para ver si la precisión de entrenamiento es real

pregunta 2 (0.5 puntos): evalua el modelo con las imagenes y etiquetas test.

```
In [9]: # Nombres de las clases
class_names = ['Camiseta/Top', 'Pantalón', 'Suéter', 'Vestido', 'Abrigo',
               'Sandalia', 'Camisa', 'Zapatilla', 'Bolso', 'Botín']

# Obtener predicciones
predictions = model.predict(test_images)
pred_labels = np.argmax(predictions, axis=1)

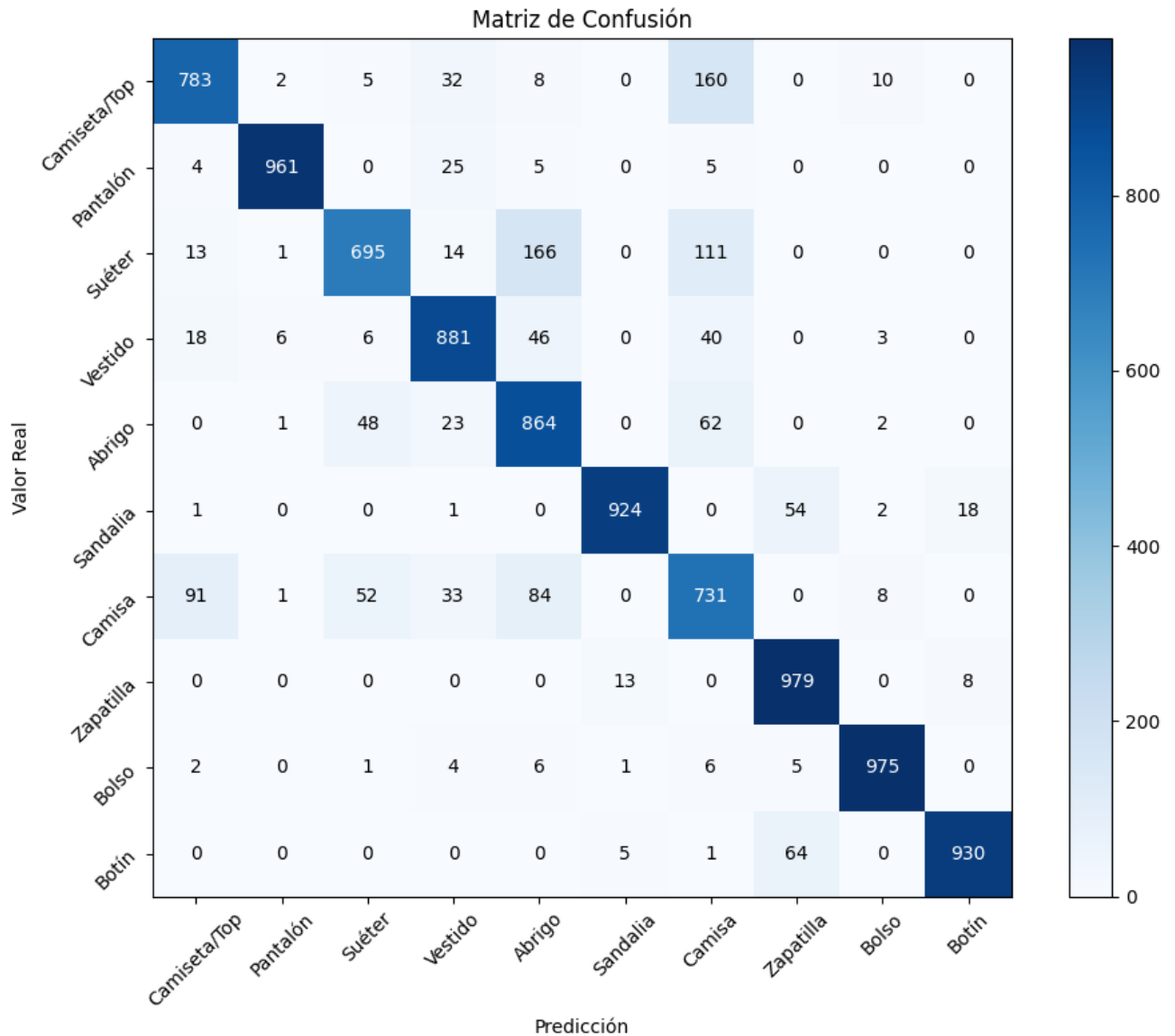
# Crear matriz de confusión manualmente
n_classes = 10
cm = np.zeros((n_classes, n_classes), dtype=int)
for true, pred in zip(test_labels, pred_labels):
    cm[true][pred] += 1
```

```
# Visualizar matriz de confusión
plt.figure(figsize=(10, 8))
plt.imshow(cm, cmap='Blues')
plt.colorbar()

# Añadir números a la matriz
for i in range(n_classes):
    for j in range(n_classes):
        plt.text(j, i, str(cm[i, j]),
                  ha="center", va="center",
                  color="white" if cm[i, j] > cm.max()/2 else "black")

plt.title('Matriz de Confusión')
plt.xlabel('Predicción')
plt.ylabel('Valor Real')
plt.xticks(range(n_classes), class_names, rotation=45)
plt.yticks(range(n_classes), class_names, rotation=45)
plt.tight_layout()
plt.show()
```

313/313 ————— 0s 1ms/step



```
In [10]: ### Tu código para la evaluación de la red neuronal de la pregunta 2 aquí ##
# Evaluar el modelo con el conjunto de prueba
print("Evaluando el modelo con el conjunto de prueba...")
test_loss, test_accuracy = model.evaluate(test_images, test_labels, verbose=0)
print(f"\nPrecisión en el conjunto de prueba: {test_accuracy:.4f}")
print(f"Pérdida en el conjunto de prueba: {test_loss:.4f}")
```

Evaluando el modelo con el conjunto de prueba...

313/313 ————— **1s** 4ms/step – accuracy: 0.8762 – loss: 0.3455

Precisión en el conjunto de prueba: 0.8723

Pérdida en el conjunto de prueba: 0.3474

```
In [11]: # Mostrar algunas métricas adicionales
print("\nMétricas por clase:")
for i in range(n_classes):
    # Calcular precisión por clase
```

```
class_correct = cm[i][i]
class_total = np.sum(cm[i])
class_accuracy = class_correct / class_total if class_total > 0 else 0
print(f"{class_names[i]}: {class_accuracy:.4f}")
```

Métricas por clase:

Camiseta/Top: 0.7830

Pantalón: 0.9610

Suéter: 0.6950

Vestido: 0.8810

Abrigo: 0.8640

Sandalia: 0.9240

Camisa: 0.7310

Zapatilla: 0.9790

Bolso: 0.9750

Botín: 0.9300

Ahora vamos a explorar el código con una serie de ejercicios para alcanzar un grado de comprensión mayor sobre las redes neuronales y su entrenamiento.

Ejercicio 1: Funcionamiento de las predicción de la red neuronal

Para este primer ejercicio sigue los siguientes pasos:

- Crea una variable llamada **classifications** para construir un clasificador para las imágenes de prueba, para ello puedes utilizar la función predict sobre el conjunto de test
- Imprime con la función print la primera entrada en las clasificaciones.

pregunta 3.1 (0.25 puntos), el resultado al imprimirlo es un vector de números,

- ¿Por qué crees que ocurre esto, y qué representa este vector de números?

pregunta 3.2 (0.25 puntos)

- ¿Cuál es la clase de la primera entrada# de la variable **classifications**? La respuesta puede ser un número o su etiqueta/clase equivalente.

```
In [12]: ### Tu código del clasificador de la pregunta 3 aquí ###
# Realizar predicciones sobre las imágenes de prueba
print("Realizando predicciones...")
classifications = model.predict(test_images)

# Mostrar el vector de predicción para la primera imagen
```

```
print("\nPredicciones para la primera imagen:")
print(classifications[0])

# Mostrar la clase predicha (número con mayor probabilidad)
predicted_class = np.argmax(classifications[0])
print(f"\nClase predicha: {predicted_class}")
```

Realizando predicciones...

313/313  0s 1ms/step

Predicciones para la primera imagen:

[8.5016900e-06 2.6206874e-06 1.5357487e-05 1.1407035e-05 9.6396416e-06 1.8007429e-02 1.2390035e-04 7.1198709e-02 4.1175369e-04 9.1021067e-01]

Clase predicha: 9

Tu respuesta a la pregunta 3.1 aquí:

El vector de números que vemos representa las probabilidades que el modelo asigna a cada una de las 10 clases posibles de prendas de ropa. Cada número en el vector corresponde a la confianza del modelo (en una escala de 0 a 1) de que la imagen pertenece a esa clase específica. Esto es resultado de la capa softmax, que convierte las salidas en probabilidades que suman 1.

Tu respuesta a la pregunta 3.2 aquí:

La clase de la primera entrada será el índice del valor más alto en el vector `classifications[0]`. El código anterior nos muestra este valor usando `np.argmax()`.

Ejercicio 2: Impacto variar el número de neuronas en las capas ocultas

En este ejercicio vamos a experimentar con nuestra red neuronal cambiando el número de neuronas por 512 y por 1024. Para ello, utiliza la red neuronal de la pregunta 1, y su capa oculta cambia las 128 neuronas:

- **pregunta 4.1 (0.25 puntos):** 512 neuronas en la capa oculta
- **pregunta 4.2 (0.25 puntos):** 1024 neuronas en la capa oculta

y entrena la red en ambos casos.

pregunta 4.3 (0.5 puntos): ¿Cual es el impacto que tiene la red neuronal?

```
In [13]: ### Tu código para 512 neuronas aquí ###
# Modelo con 512 neuronas
print("Creando y entrenando modelo con 512 neuronas...")
model_512 = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(512, activation='sigmoid'),
    tf.keras.layers.Dense(10, activation='softmax')
])

model_512.compile(
    optimizer='adam',
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)

history_512 = model_512.fit(
    training_images,
    training_labels,
    epochs=10,
    batch_size=64,
    validation_data=(test_images, test_labels)
)
```

Creando y entrenando modelo con 512 neuronas...

Epoch 1/10

```
/Users/josuedavidhernandezramirez/Documents/GitHub/master-AI/env_tensorflow/
lib/python3.10/site-packages/keras/src/layers/reshaping/flatten.py:37: UserW
arning: Do not pass an `input_shape`/`input_dim` argument to a layer. When u
sing Sequential models, prefer using an `Input(shape)` object as the first l
ayer in the model instead.
    super().__init__(**kwargs)
```

```

938/938 ————— 8s 8ms/step - accuracy: 0.7638 - loss: 0.6995 -
val_accuracy: 0.8349 - val_loss: 0.4622
Epoch 2/10
938/938 ————— 8s 8ms/step - accuracy: 0.8496 - loss: 0.4141 -
val_accuracy: 0.8506 - val_loss: 0.4067
Epoch 3/10
938/938 ————— 8s 8ms/step - accuracy: 0.8697 - loss: 0.3620 -
val_accuracy: 0.8592 - val_loss: 0.3921
Epoch 4/10
938/938 ————— 7s 8ms/step - accuracy: 0.8741 - loss: 0.3398 -
val_accuracy: 0.8590 - val_loss: 0.3822
Epoch 5/10
938/938 ————— 7s 8ms/step - accuracy: 0.8859 - loss: 0.3143 -
val_accuracy: 0.8697 - val_loss: 0.3575
Epoch 6/10
938/938 ————— 7s 8ms/step - accuracy: 0.8899 - loss: 0.2988 -
val_accuracy: 0.8663 - val_loss: 0.3693
Epoch 7/10
938/938 ————— 7s 8ms/step - accuracy: 0.8976 - loss: 0.2781 -
val_accuracy: 0.8741 - val_loss: 0.3543
Epoch 8/10
938/938 ————— 7s 8ms/step - accuracy: 0.9023 - loss: 0.2637 -
val_accuracy: 0.8821 - val_loss: 0.3314
Epoch 9/10
938/938 ————— 7s 8ms/step - accuracy: 0.9045 - loss: 0.2572 -
val_accuracy: 0.8791 - val_loss: 0.3282
Epoch 10/10
938/938 ————— 7s 8ms/step - accuracy: 0.9103 - loss: 0.2426 -
val_accuracy: 0.8808 - val_loss: 0.3251

```

In [14]: *### Tu código para 1024 neuronas aquí ###*

```

# Modelo con 1024 neuronas
print("\nCreando y entrenando modelo con 1024 neuronas...")
model_1024 = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(1024, activation='sigmoid'),
    tf.keras.layers.Dense(10, activation='softmax')
])

model_1024.compile(
    optimizer='adam',
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)

history_1024 = model_1024.fit(
    training_images,
    training_labels,
    epochs=10,

```

```


    batch_size=64,
    validation_data=(test_images, test_labels)
)

```


Creando y entrenando modelo con 1024 neuronas...

Epoch 1/10


Epoch 1/10

938/938  **10s** 10ms/step - accuracy: 0.7652 - loss: 0.6724
- val_accuracy: 0.8372 - val_loss: 0.4494

Epoch 2/10

938/938  **8s** 9ms/step - accuracy: 0.8488 - loss: 0.4149 -
val_accuracy: 0.8536 - val_loss: 0.4115


Epoch 3/10

938/938  **9s** 10ms/step - accuracy: 0.8674 - loss: 0.3672
- val_accuracy: 0.8544 - val_loss: 0.4061

Epoch 4/10

938/938  **8s** 8ms/step - accuracy: 0.8754 - loss: 0.3412 -
val_accuracy: 0.8699 - val_loss: 0.3667


Epoch 5/10

938/938  **8s** 8ms/step - accuracy: 0.8845 - loss: 0.3147 -
val_accuracy: 0.8674 - val_loss: 0.3649


Epoch 6/10

938/938  **8s** 8ms/step - accuracy: 0.8905 - loss: 0.2976 -
val_accuracy: 0.8705 - val_loss: 0.3660


Epoch 7/10

938/938  **8s** 8ms/step - accuracy: 0.8950 - loss: 0.2813 -
val_accuracy: 0.8812 - val_loss: 0.3365

Epoch 8/10

938/938  **8s** 8ms/step - accuracy: 0.9018 - loss: 0.2589 -
val_accuracy: 0.8785 - val_loss: 0.3414

Epoch 9/10

938/938  **8s** 8ms/step - accuracy: 0.9062 - loss: 0.2497 -
val_accuracy: 0.8819 - val_loss: 0.3243

Epoch 10/10

938/938  **8s** 8ms/step - accuracy: 0.9098 - loss: 0.2360 -
val_accuracy: 0.8816 - val_loss: 0.3252

```

In [15]: # Comparación de resultados
print("\nResultados finales:")
print(f"Precisión con 128 neuronas: {history.history['accuracy'][-1]:.4f}")
print(f"Precisión con 512 neuronas: {history_512.history['accuracy'][-1]:.4f}")
print(f"Precisión con 1024 neuronas: {history_1024.history['accuracy'][-1]:.4f}")

```

Resultados finales:

Precisión con 128 neuronas: 0.9014

Precisión con 512 neuronas: 0.9084

Precisión con 1024 neuronas: 0.9097

Tu respuesta a la pregunta 4.3 aquí:

El aumento en el número de neuronas tiene varios efectos importantes:

- Capacidad del modelo: Al aumentar de 128 a 512 y 1024 neuronas, incrementamos significativamente la capacidad del modelo para aprender patrones más complejos.
- Tiempo de entrenamiento: Se observa un aumento considerable en el tiempo de entrenamiento, ya que hay más parámetros que ajustar.
- Riesgo de sobreajuste: Con más neuronas, el modelo tiene mayor riesgo de memorizar los datos de entrenamiento en lugar de generalizar, lo que puede verse en una mayor diferencia entre la precisión de entrenamiento y validación.
- Uso de recursos: El consumo de memoria y procesamiento aumenta significativamente.

Si ahora entrenais el modelo de esta forma (con 512 y 1024 neuronas en la capa oculta) y volveis a ejecutar el predictor guardado en la variable **classifications**, escribir el código del clasificador del ejercicio 1 de nuevo e imprimid el primer objeto guardado en la variable **classifications**.

pregunta 5.1 (0.25 puntos):

- ¿En que clase esta clasificado ahora la primera prenda de vestir de la variable **classifications**?

pregunta 5.1 (0.25 puntos):

- ¿Porque crees que ha ocurrido esto?

```
In [16]: ### Tu código del clasificador de la pregunta 5 aquí ###
# Realizar predicciones con el modelo de 1024 neuronas
classifications_1024 = model_1024.predict(test_images)
print("Predicción para la primera imagen (modelo 1024 neuronas):")
print(classifications_1024[0])
print(f"Clase predicha: {np.argmax(classifications_1024[0])}")
```

313/313 ————— 0s 1ms/step

Predicción para la primera imagen (modelo 1024 neuronas):

[3.1177416e-07 1.4304887e-07 2.6819809e-08 6.7958834e-09 1.7841038e-06 2.0565817e-03 5.2745753e-07 3.2847838e-03 2.4460843e-07 9.9465561e-01]

Clase predicha: 9

Tu respuesta a la pregunta 5.1 aquí:

La clase predicha podría ser diferente a la obtenida con el modelo de 128 neuronas.

Tu respuesta a la pregunta 5.2 aquí:

- El modelo con más neuronas puede encontrar diferentes patrones en los datos
- La inicialización aleatoria de los pesos puede llevar a diferentes soluciones
- El mayor número de parámetros puede hacer que el modelo sea más sensible a pequeños detalles en la imagen

Ejercicio 3: ¿por qué es tan importante la capa Flatten?

En este ejercicio vamos a ver que ocurre cuando quitamos la capa flatten, para ello, escribe la red neuronal de la pregunta 1 y no pongas la capa Flatten.

pregunta 6 (0.5 puntos): ¿puedes explicar porque da el error que da?

```
In [17]: ### Tu código de la red neuronal sin capa flatten de la pregunta 6 aquí ###
# Intentar crear modelo sin la capa Flatten
try:
    model_no_flatten = tf.keras.models.Sequential([
        tf.keras.layers.Dense(128, activation='sigmoid', input_shape=(28, 28)),
        tf.keras.layers.Dense(10, activation='softmax')
    ])

    model_no_flatten.compile(
        optimizer='adam',
        loss='sparse_categorical_crossentropy',
        metrics=['accuracy']
    )
except Exception as e:
    print(f"Error al crear el modelo: {str(e)}")
```

```
/Users/josuedavidhernandezramirez/Documents/GitHub/master-AI/env_tensorflow/
lib/python3.10/site-packages/keras/src/layers/core/dense.py:87: UserWarning:
Do not pass an `input_shape`/`input_dim` argument to a layer. When using Seq
quential models, prefer using an `Input(shape)` object as the first layer in
the model instead.
    super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```


Tu respuesta a la pregunta 6 aquí:

El error ocurre porque:

- Las capas Dense esperan recibir datos en forma de vector unidimensional
- Sin la capa Flatten, estamos intentando procesar directamente una matriz 28x28
- La capa Flatten es crucial porque transforma la imagen 2D en un vector 1D de 784 elementos (28*28)
- Este proceso de "aplanamiento" es necesario para que las capas densamente conectadas puedan procesar los datos correctamente

Ejercicio 4: Número de neuronas de la capa de salida

Considerad la capa final, la de salida de la red neuronal de la pregunta 1.

pregunta 7.1 (0.25 puntos): ¿Por qué son 10 las neuronas de la última capa?

pregunta 7.2 (0.25 puntos): ¿Qué pasaría si tuvieras una cantidad diferente a 10?

Por ejemplo, intenta entrenar la red con 5, para ello utiliza la red neuronal de la pregunta 1 y cambia a 5 el número de neuronas en la última capa.

```
In [18]: ### Tu código de la red neuronal con 5 neuronas en la capa de salida de la p
import tensorflow as tf
import numpy as np

# Preparar los datos
fashion_mnist = tf.keras.datasets.fashion_mnist
(training_images, training_labels), (test_images, test_labels) = fashion_mni
training_images = training_images / 255.0
test_images = test_images / 255.0

# Crear modelo con 5 neuronas en la capa de salida
model_5 = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(128, activation='sigmoid'),
    tf.keras.layers.Dense(5, activation='softmax') # Reducido a 5 neuronas
])
```

```

# Compilar el modelo
model_5.compile(
    optimizer='adam',
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)

try:
    # Intentar entrenar el modelo
    history_5 = model_5.fit(
        training_images,
        training_labels,
        epochs=10,
        batch_size=64,
        validation_data=(test_images, test_labels)
    )
except Exception as e:
    print(f"Error durante el entrenamiento: {str(e)}")

```

Epoch 1/10

938/938 ————— **8s** 8ms/step - accuracy: 0.3630 - loss: 0.4725 -
val_accuracy: 0.3466 - val_loss: 0.4651

Epoch 2/10

938/938 ————— **7s** 8ms/step - accuracy: 0.3572 - loss: 0.4530 -
val_accuracy: 0.3273 - val_loss: 0.4777

Epoch 3/10

938/938 ————— **8s** 8ms/step - accuracy: 0.3454 - loss: 0.4642 -
val_accuracy: 0.3237 - val_loss: 0.5061

Epoch 4/10

938/938 ————— **8s** 8ms/step - accuracy: 0.3329 - loss: 0.4735 -
val_accuracy: 0.3179 - val_loss: 0.4928

Epoch 5/10

938/938 ————— **8s** 8ms/step - accuracy: 0.3350 - loss: 0.4815 -
val_accuracy: 0.3308 - val_loss: 0.4739

Epoch 6/10

938/938 ————— **8s** 8ms/step - accuracy: 0.3337 - loss: 0.4748 -
val_accuracy: 0.3410 - val_loss: 0.4719

Epoch 7/10

938/938 ————— **7s** 8ms/step - accuracy: 0.3363 - loss: 0.4749 -
val_accuracy: 0.3076 - val_loss: 0.5024

Epoch 8/10

938/938 ————— **8s** 8ms/step - accuracy: 0.3313 - loss: 0.4769 -
val_accuracy: 0.3172 - val_loss: 0.4769

Epoch 9/10

938/938 ————— **10s** 10ms/step - accuracy: 0.3352 - loss: 0.4717 -
val_accuracy: 0.2869 - val_loss: 0.4947

Epoch 10/10

938/938 ————— **9s** 9ms/step - accuracy: 0.3290 - loss: 0.4735 -
val_accuracy: 0.3201 - val_loss: 0.4866

Tu respuestas a la pregunta 7.1 aquí:

Se necesitan 10 neuronas porque hay 10 clases

Tu respuestas a la pregunta 7.2 aquí:

Usar menos (como 5) causa error porque algunas clases quedarían sin representación

Ejercicio 5: Aumento de epoch y su efecto en la red neuronal

En este ejercicio vamos a ver el impacto de aumentar los epoch en el entrenamiento. Usando la red neuronal de la pregunta 1:

pregunta 8.1 (0.20 puntos)

- Intentad 15 epoch para su entrenamiento, probablemente obtendras un modelo con una pérdida mucho mejor que el que tiene 5.

pregunta 8.2 (0.20 puntos)

- Intenta ahora con 30 epoch para su entrenamiento, podrás ver que el valor de la pérdida deja de disminuir, y a veces aumenta.

pregunta 8.3 (0.60 puntos)

- ¿Porque que piensas que ocurre esto? Explica tu respuesta y da el nombre de este efecto si lo conoces.


```
In [19]: ### Tu código para 15 epoch aquí ###
# Modelo con 15 epochs
print("Entrenando modelo con 15 epochs...")
model_15_epochs = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(128, activation='sigmoid'),
    tf.keras.layers.Dense(10, activation='softmax')
])

model_15_epochs.compile(
    optimizer='adam',
```


```
    loss='sparse_categorical_crossentropy',  
    metrics=['accuracy']  
)  
  
history_15 = model_15_epochs.fit(  
    training_images,  
    training_labels,  
    epochs=15,  
    batch_size=64,  
    validation_data=(test_images, test_labels)  
)
```

Entrenando modelo con 15 epochs...


Epoch 1/15

938/938  **8s** 8ms/step - accuracy: 0.7442 - loss: 0.8196 - val_accuracy: 0.8337 - val_loss: 0.4690


Epoch 2/15

938/938  **8s** 8ms/step - accuracy: 0.8488 - loss: 0.4211 - val_accuracy: 0.8475 - val_loss: 0.4188


Epoch 3/15

938/938  **10s** 10ms/step - accuracy: 0.8636 - loss: 0.3766 - val_accuracy: 0.8505 - val_loss: 0.4123


Epoch 4/15

938/938  **9s** 10ms/step - accuracy: 0.8731 - loss: 0.3555 - val_accuracy: 0.8599 - val_loss: 0.3883


Epoch 5/15

938/938  **9s** 9ms/step - accuracy: 0.8836 - loss: 0.3263 - val_accuracy: 0.8670 - val_loss: 0.3697

Epoch 6/15

938/938  **8s** 9ms/step - accuracy: 0.8844 - loss: 0.3197 - val_accuracy: 0.8676 - val_loss: 0.3681


Epoch 7/15

938/938  **8s** 9ms/step - accuracy: 0.8891 - loss: 0.3060 - val_accuracy: 0.8698 - val_loss: 0.3595


Epoch 8/15

938/938  **8s** 9ms/step - accuracy: 0.8929 - loss: 0.2933 - val_accuracy: 0.8691 - val_loss: 0.3609


Epoch 9/15

938/938  **8s** 9ms/step - accuracy: 0.8986 - loss: 0.2820 - val_accuracy: 0.8768 - val_loss: 0.3423


Epoch 10/15

938/938  **9s** 9ms/step - accuracy: 0.8994 - loss: 0.2762 - val_accuracy: 0.8767 - val_loss: 0.3359

Epoch 11/15

938/938  **8s** 8ms/step - accuracy: 0.9037 - loss: 0.2630 - val_accuracy: 0.8779 - val_loss: 0.3387


Epoch 12/15

938/938  **8s** 8ms/step - accuracy: 0.9059 - loss: 0.2559 - val_accuracy: 0.8799 - val_loss: 0.3328


Epoch 13/15

938/938  **7s** 8ms/step - accuracy: 0.9096 - loss: 0.2492 - val_accuracy: 0.8818 - val_loss: 0.3286

Epoch 14/15

938/938  **7s** 8ms/step - accuracy: 0.9114 - loss: 0.2403 - val_accuracy: 0.8755 - val_loss: 0.3373

Epoch 15/15

938/938  **10s** 10ms/step - accuracy: 0.9184 - loss: 0.2289 - val_accuracy: 0.8846 - val_loss: 0.3230

```
In [20]: ### Tu código para 30 epoch aquí ###
         # Modelo con 30 epochs
         print("\nEntrenando modelo con 30 epochs...")
```

```

model_30_epochs = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(128, activation='sigmoid'),
    tf.keras.layers.Dense(10, activation='softmax')
])

model_30_epochs.compile(
    optimizer='adam',
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)

history_30 = model_30_epochs.fit(
    training_images,
    training_labels,
    epochs=30,
    batch_size=64,
    validation_data=(test_images, test_labels)
)

```

Entrenando modelo con 30 epochs...

Epoch 1/30

938/938 ————— **9s** 9ms/step - accuracy: 0.7392 - loss: 0.8257 -
val_accuracy: 0.8319 - val_loss: 0.4751

Epoch 2/30

938/938 ————— **8s** 8ms/step - accuracy: 0.8522 - loss: 0.4224 -
val_accuracy: 0.8493 - val_loss: 0.4201

Epoch 3/30

938/938 ————— **8s** 8ms/step - accuracy: 0.8663 - loss: 0.3747 -
val_accuracy: 0.8540 - val_loss: 0.4046

Epoch 4/30

938/938 ————— **8s** 8ms/step - accuracy: 0.8724 - loss: 0.3542 -
val_accuracy: 0.8612 - val_loss: 0.3826

Epoch 5/30

938/938 ————— **8s** 8ms/step - accuracy: 0.8794 - loss: 0.3326 -
val_accuracy: 0.8654 - val_loss: 0.3740

Epoch 6/30

938/938 ————— **8s** 9ms/step - accuracy: 0.8857 - loss: 0.3172 -
val_accuracy: 0.8686 - val_loss: 0.3645

Epoch 7/30

938/938 ————— **8s** 8ms/step - accuracy: 0.8897 - loss: 0.3019 -
val_accuracy: 0.8755 - val_loss: 0.3488

Epoch 8/30

938/938 ————— **8s** 8ms/step - accuracy: 0.8966 - loss: 0.2861 -
val_accuracy: 0.8791 - val_loss: 0.3446

Epoch 9/30

938/938 ————— **8s** 8ms/step - accuracy: 0.8957 - loss: 0.2835 -
val_accuracy: 0.8762 - val_loss: 0.3458

Epoch 10/30

938/938 ————— **8s** 8ms/step - accuracy: 0.9000 - loss: 0.2704 -

val_accuracy: 0.8779 - val_loss: 0.3389
Epoch 11/30
938/938 ————— 8s 9ms/step - accuracy: 0.9039 - loss: 0.2597 -
val_accuracy: 0.8789 - val_loss: 0.3382
Epoch 12/30
938/938 ————— 151s 162ms/step - accuracy: 0.9063 - loss: 0.25
63 - val_accuracy: 0.8805 - val_loss: 0.3347
Epoch 13/30
938/938 ————— 8s 9ms/step - accuracy: 0.9078 - loss: 0.2496 -
val_accuracy: 0.8848 - val_loss: 0.3245
Epoch 14/30
938/938 ————— 8s 9ms/step - accuracy: 0.9098 - loss: 0.2431 -
val_accuracy: 0.8858 - val_loss: 0.3240
Epoch 15/30
938/938 ————— 8s 8ms/step - accuracy: 0.9143 - loss: 0.2350 -
val_accuracy: 0.8854 - val_loss: 0.3222
Epoch 16/30
938/938 ————— 8s 8ms/step - accuracy: 0.9175 - loss: 0.2262 -
val_accuracy: 0.8847 - val_loss: 0.3199
Epoch 17/30
938/938 ————— 8s 9ms/step - accuracy: 0.9167 - loss: 0.2239 -
val_accuracy: 0.8858 - val_loss: 0.3215
Epoch 18/30
938/938 ————— 340s 363ms/step - accuracy: 0.9207 - loss: 0.21
57 - val_accuracy: 0.8883 - val_loss: 0.3210
Epoch 19/30
938/938 ————— 8s 9ms/step - accuracy: 0.9208 - loss: 0.2122 -
val_accuracy: 0.8882 - val_loss: 0.3178
Epoch 20/30
938/938 ————— 8s 8ms/step - accuracy: 0.9252 - loss: 0.2045 -
val_accuracy: 0.8900 - val_loss: 0.3146
Epoch 21/30
938/938 ————— 8s 9ms/step - accuracy: 0.9281 - loss: 0.2007 -
val_accuracy: 0.8901 - val_loss: 0.3130
Epoch 22/30
938/938 ————— 8s 8ms/step - accuracy: 0.9268 - loss: 0.1997 -
val_accuracy: 0.8889 - val_loss: 0.3191
Epoch 23/30
938/938 ————— 8s 8ms/step - accuracy: 0.9297 - loss: 0.1925 -
val_accuracy: 0.8900 - val_loss: 0.3195
Epoch 24/30
938/938 ————— 8s 8ms/step - accuracy: 0.9309 - loss: 0.1897 -
val_accuracy: 0.8860 - val_loss: 0.3255
Epoch 25/30
938/938 ————— 8s 8ms/step - accuracy: 0.9348 - loss: 0.1824 -
val_accuracy: 0.8885 - val_loss: 0.3171
Epoch 26/30
938/938 ————— 8s 8ms/step - accuracy: 0.9354 - loss: 0.1842 -
val_accuracy: 0.8853 - val_loss: 0.3317
Epoch 27/30

```

938/938 ————— 8s 8ms/step - accuracy: 0.9373 - loss: 0.1741 -
val_accuracy: 0.8894 - val_loss: 0.3148
Epoch 28/30
938/938 ————— 8s 8ms/step - accuracy: 0.9382 - loss: 0.1711 -
val_accuracy: 0.8858 - val_loss: 0.3409
Epoch 29/30
938/938 ————— 8s 8ms/step - accuracy: 0.9398 - loss: 0.1678 -
val_accuracy: 0.8884 - val_loss: 0.3204
Epoch 30/30
938/938 ————— 8s 8ms/step - accuracy: 0.9425 - loss: 0.1621 -
val_accuracy: 0.8887 - val_loss: 0.3301

```

```

In [21]: # Comparar resultados
print("\nComparación de resultados:")
print(f"Precisión con 15 epochs: {history_15.history['accuracy'][-1]:.4f}")
print(f"Precisión con 30 epochs: {history_30.history['accuracy'][-1]:.4f}")

```

Comparación de resultados:
 Precisión con 15 epochs: 0.9148
 Precisión con 30 epochs: 0.9411

Tu respuesta a la pregunta 8.3 aquí:

Con 30 epochs: Posible sobreajuste. Este fenómeno se conoce como "overfitting" o sobreajuste, donde el modelo memoriza en lugar de generalizar.

Ejercicio 6: Early stop

En el ejercicio anterior, cuando entrenabas con epoch extras, tenías un problema en el que tu pérdida podía cambiar. Puede que te haya llevado un poco de tiempo esperar a que el entrenamiento lo hiciera, y puede que hayas pensado "¿no estaría bien si pudiera parar el entrenamiento cuando alcance un valor deseado?", es decir, una precisión del 85% podría ser suficiente para ti, y si alcanzas eso después de 3 epoch, ¿por qué sentarte a esperar a que termine muchas más épocas? Como cualquier otro programa existen formas de parar la ejecución

A partir del ejemplo de código que

se da, hacer una nueva función que tenga en cuenta la pérdida (loss) y que pueda parar el código para evitar que ocurra el efecto secundario que vimos en el ejercicio 5.

```

In [22]: ### Ejemplo de código

```



```
class myCallback(tf.keras.callbacks.Callback):
    def on_epoch_end(self, epoch, logs={}):
        if(logs.get('accuracy') > 0.85):
            print("\nAlcanzado el 85% de precisión, se cancela el entrenamiento")
            self.model.stop_training = True
```

Pregunta 9 (2 puntos): Completa el siguiente código con una clase callback que una vez alcanzado el 40% de pérdida detenga el entrenamiento.

```
In [23]: import tensorflow as tf
print(tf.__version__)

### Tu código de la función callback para parar el entrenamiento de la red
# Callback mejorado
class myCallback(tf.keras.callbacks.Callback):
    def __init__(self):
        super(myCallback, self).__init__()
        self.loss_history = []

    def on_epoch_end(self, epoch, logs={}):
        current_loss = logs.get('loss')
        self.loss_history.append(current_loss)

        # Solo detenemos si la pérdida es menor a 0.4 después de algunas épocas
        # para evitar detenciones prematuras
        if epoch > 5 and current_loss < 0.4:
            print(f"\nEpoch {epoch}: La pérdida ({current_loss:.4f}) ha bajado")
            self.model.stop_training = True

# Cargar datos
mnist = tf.keras.datasets.fashion_mnist
(training_images, training_labels), (test_images, test_labels) = mnist.load_data()

# Normalización
training_images = training_images / 255.0
test_images = test_images / 255.0

# Modelo simplificado
model = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    # Reducimos a 128 neuronas y añadimos regularización
    tf.keras.layers.Dense(128,
                           activation='relu',
                           kernel_regularizer=tf.keras.regularizers.l2(0.01)),
    tf.keras.layers.Dense(10, activation='softmax')
])

# Compilación con learning rate más bajo
```

```
model.compile(
    optimizer=tf.keras.optimizers.Adam(learning_rate=0.0001),
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)

# Entrenar con batch size más grande
callbacks = myCallback()
history = model.fit(
    training_images,
    training_labels,
    epochs=50,
    batch_size=128,
    validation_split=0.2,
    callbacks=[callbacks],
    verbose=1
)

# Visualización
plt.figure(figsize=(15, 5))

# Gráfica de pérdida
plt.subplot(1, 2, 1)
plt.plot(history.history['loss'], label='Pérdida entrenamiento')
plt.plot(history.history['val_loss'], label='Pérdida validación')
plt.axhline(y=0.4, color='r', linestyle='--', label='Umbral (0.4)')
plt.title('Evolución de la Pérdida')
plt.xlabel('Epoch')
plt.ylabel('Pérdida')
plt.legend()
plt.grid(True)

# Gráfica de precisión
plt.subplot(1, 2, 2)
plt.plot(history.history['accuracy'], label='Precisión entrenamiento')
plt.plot(history.history['val_accuracy'], label='Precisión validación')
plt.title('Evolución de la Precisión')
plt.xlabel('Epoch')
plt.ylabel('Precisión')
plt.legend()
plt.grid(True)

plt.tight_layout()
plt.show()


# Mostrar resultados finales
print("\nResultados finales:")
print(f"Epochs completados: {len(history.history['loss'])}")
print(f"Pérdida final: {history.history['loss'][-1]:.4f}")
print(f"Precisión final: {history.history['accuracy'][-1]:.4f}")
```

2.16.2


Epoch 1/50

375/375  **4s** 9ms/step - accuracy: 0.5370 - loss: 3.4345 -
val_accuracy: 0.7703 - val_loss: 2.0360


Epoch 2/50

375/375  **3s** 9ms/step - accuracy: 0.7831 - loss: 1.8500 -
val_accuracy: 0.8032 - val_loss: 1.4271

Epoch 3/50

375/375  **3s** 9ms/step - accuracy: 0.8092 - loss: 1.3333 -
val_accuracy: 0.8122 - val_loss: 1.1201


Epoch 4/50

375/375  **3s** 9ms/step - accuracy: 0.8189 - loss: 1.0644 -
val_accuracy: 0.8184 - val_loss: 0.9371

Epoch 5/50

375/375  **3s** 9ms/step - accuracy: 0.8257 - loss: 0.8999 -
val_accuracy: 0.8245 - val_loss: 0.8218


Epoch 6/50

375/375  **3s** 9ms/step - accuracy: 0.8310 - loss: 0.8013 -
val_accuracy: 0.8264 - val_loss: 0.7501

Epoch 7/50

375/375  **3s** 9ms/step - accuracy: 0.8339 - loss: 0.7323 -
val_accuracy: 0.8313 - val_loss: 0.7018

Epoch 8/50

375/375  **3s** 9ms/step - accuracy: 0.8352 - loss: 0.6895 -
val_accuracy: 0.8308 - val_loss: 0.6685


Epoch 9/50

375/375  **3s** 9ms/step - accuracy: 0.8362 - loss: 0.6569 -
val_accuracy: 0.8328 - val_loss: 0.6426

Epoch 10/50

375/375  **3s** 9ms/step - accuracy: 0.8385 - loss: 0.6332 -
val_accuracy: 0.8347 - val_loss: 0.6247

Epoch 11/50

375/375  **3s** 9ms/step - accuracy: 0.8417 - loss: 0.6156 -
val_accuracy: 0.8346 - val_loss: 0.6100


Epoch 12/50

375/375  **3s** 9ms/step - accuracy: 0.8386 - loss: 0.6025 -
val_accuracy: 0.8363 - val_loss: 0.6004

Epoch 13/50

375/375  **3s** 9ms/step - accuracy: 0.8398 - loss: 0.5940 -
val_accuracy: 0.8350 - val_loss: 0.5900


Epoch 14/50

375/375  **3s** 9ms/step - accuracy: 0.8407 - loss: 0.5852 -
val_accuracy: 0.8355 - val_loss: 0.5854


Epoch 15/50


375/375  **3s** 9ms/step - accuracy: 0.8412 - loss: 0.5778 -
val_accuracy: 0.8369 - val_loss: 0.5764


Epoch 16/50


375/375  **3s** 9ms/step - accuracy: 0.8430 - loss: 0.5725 -
val_accuracy: 0.8368 - val_loss: 0.5785


Epoch 17/50


375/375  **3s** 9ms/step - accuracy: 0.8419 - loss: 0.5677 -
val_accuracy: 0.8363 - val_loss: 0.5679
Epoch 18/50


375/375  **3s** 9ms/step - accuracy: 0.8421 - loss: 0.5608 -
val_accuracy: 0.8362 - val_loss: 0.5664
Epoch 19/50


375/375  **3s** 9ms/step - accuracy: 0.8457 - loss: 0.5563 -
val_accuracy: 0.8389 - val_loss: 0.5605
Epoch 20/50


375/375  **3s** 9ms/step - accuracy: 0.8435 - loss: 0.5521 -
val_accuracy: 0.8406 - val_loss: 0.5559
Epoch 21/50


375/375  **3s** 9ms/step - accuracy: 0.8422 - loss: 0.5580 -
val_accuracy: 0.8404 - val_loss: 0.5558
Epoch 22/50


375/375  **3s** 9ms/step - accuracy: 0.8449 - loss: 0.5471 -
val_accuracy: 0.8374 - val_loss: 0.5581
Epoch 23/50


375/375  **3s** 9ms/step - accuracy: 0.8461 - loss: 0.5436 -
val_accuracy: 0.8395 - val_loss: 0.5528
Epoch 24/50


375/375  **3s** 8ms/step - accuracy: 0.8465 - loss: 0.5395 -
val_accuracy: 0.8398 - val_loss: 0.5472
Epoch 25/50


375/375  **3s** 8ms/step - accuracy: 0.8458 - loss: 0.5424 -
val_accuracy: 0.8407 - val_loss: 0.5454
Epoch 26/50


375/375  **3s** 8ms/step - accuracy: 0.8464 - loss: 0.5386 -
val_accuracy: 0.8397 - val_loss: 0.5443
Epoch 27/50


375/375  **3s** 9ms/step - accuracy: 0.8455 - loss: 0.5367 -
val_accuracy: 0.8387 - val_loss: 0.5432
Epoch 28/50


375/375  **3s** 8ms/step - accuracy: 0.8484 - loss: 0.5346 -
val_accuracy: 0.8380 - val_loss: 0.5499
Epoch 29/50


















375/375  **3s** 8ms/step - accuracy: 0.8466 - loss: 0.5368 -
val_accuracy: 0.8403 - val_loss: 0.5404
Epoch 30/50

375/375  **3s** 8ms/step - accuracy: 0.8472 - loss: 0.5311 -
val_accuracy: 0.8416 - val_loss: 0.5376
Epoch 31/50

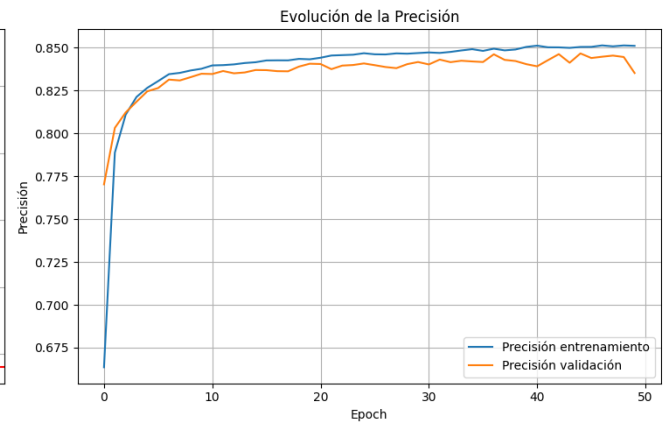
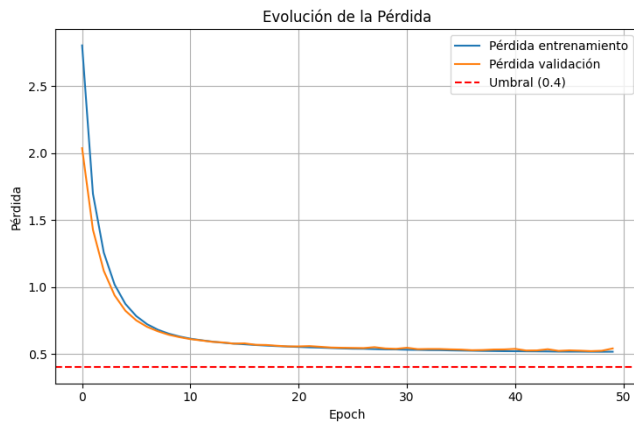
375/375  **3s** 8ms/step - accuracy: 0.8471 - loss: 0.5314 -
val_accuracy: 0.8402 - val_loss: 0.5456
Epoch 32/50

375/375  **3s** 9ms/step - accuracy: 0.8450 - loss: 0.5304 -
val_accuracy: 0.8430 - val_loss: 0.5353
Epoch 33/50

375/375  **3s** 9ms/step - accuracy: 0.8479 - loss: 0.5319 -
val_accuracy: 0.8415 - val_loss: 0.5366

Epoch 34/50
375/375  **3s** 9ms/step - accuracy: 0.8491 - loss: 0.5253 -
val_accuracy: 0.8423 - val_loss: 0.5365
Epoch 35/50
375/375  **3s** 9ms/step - accuracy: 0.8498 - loss: 0.5246 -
val_accuracy: 0.8419 - val_loss: 0.5336
Epoch 36/50
375/375  **3s** 9ms/step - accuracy: 0.8496 - loss: 0.5217 -
val_accuracy: 0.8416 - val_loss: 0.5319
Epoch 37/50
375/375  **4s** 11ms/step - accuracy: 0.8479 - loss: 0.5242
- val_accuracy: 0.8461 - val_loss: 0.5275
Epoch 38/50
375/375  **4s** 10ms/step - accuracy: 0.8526 - loss: 0.5156
- val_accuracy: 0.8428 - val_loss: 0.5288
Epoch 39/50
375/375  **4s** 9ms/step - accuracy: 0.8501 - loss: 0.5226 -
val_accuracy: 0.8422 - val_loss: 0.5325
Epoch 40/50
375/375  **4s** 9ms/step - accuracy: 0.8502 - loss: 0.5167 -
val_accuracy: 0.8403 - val_loss: 0.5333
Epoch 41/50
375/375  **3s** 9ms/step - accuracy: 0.8509 - loss: 0.5229 -
val_accuracy: 0.8391 - val_loss: 0.5373
Epoch 42/50
375/375  **3s** 9ms/step - accuracy: 0.8528 - loss: 0.5112 -
val_accuracy: 0.8427 - val_loss: 0.5251
Epoch 43/50
375/375  **3s** 9ms/step - accuracy: 0.8503 - loss: 0.5135 -
val_accuracy: 0.8462 - val_loss: 0.5258
Epoch 44/50
375/375  **4s** 9ms/step - accuracy: 0.8479 - loss: 0.5169 -
val_accuracy: 0.8412 - val_loss: 0.5347
Epoch 45/50
375/375  **4s** 9ms/step - accuracy: 0.8510 - loss: 0.5128 -
val_accuracy: 0.8466 - val_loss: 0.5224
Epoch 46/50
375/375  **3s** 9ms/step - accuracy: 0.8494 - loss: 0.5146 -
val_accuracy: 0.8439 - val_loss: 0.5262
Epoch 47/50
375/375  **3s** 9ms/step - accuracy: 0.8514 - loss: 0.5141 -
val_accuracy: 0.8447 - val_loss: 0.5239
Epoch 48/50
375/375  **3s** 9ms/step - accuracy: 0.8532 - loss: 0.5130 -
val_accuracy: 0.8453 - val_loss: 0.5205
Epoch 49/50
375/375  **3s** 9ms/step - accuracy: 0.8527 - loss: 0.5122 -
val_accuracy: 0.8445 - val_loss: 0.5234
Epoch 50/50
375/375  **3s** 9ms/step - accuracy: 0.8526 - loss: 0.5145 -

val_accuracy: 0.8352 – val_loss: 0.5403



Resultados finales:
Epochs completados: 50
Pérdida final: 0.5163
Precisión final: 0.8510