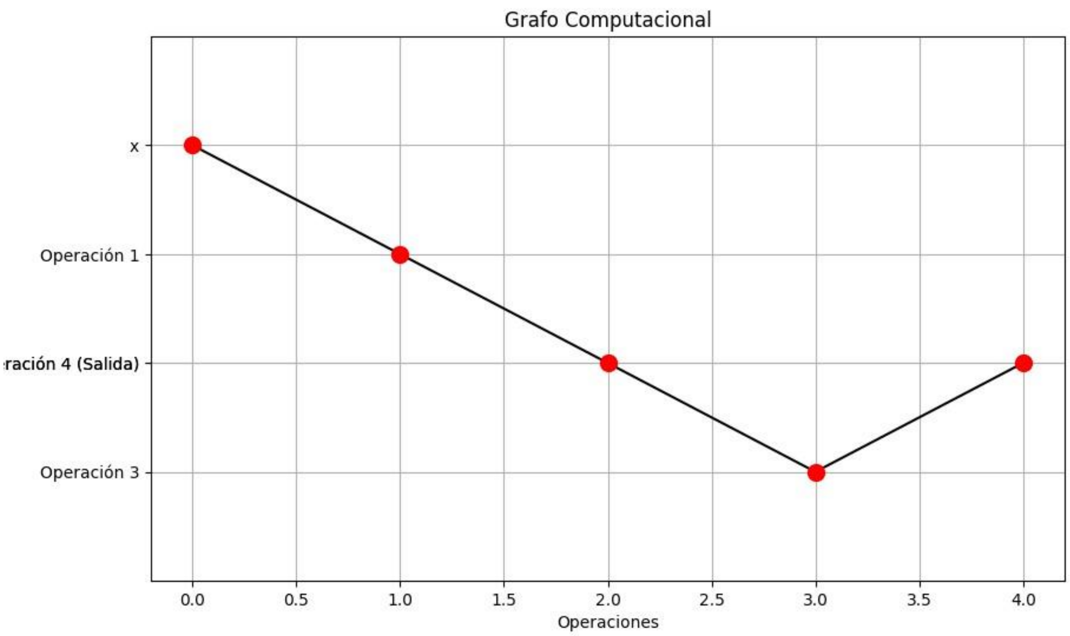
1. Realizar el grafo computacional de las siguientes funciones
    a. Y=sin(x^2+5x+2)
    b. Y=cos(2x+3)-relu(x)
    c. $y = \frac{2x}{e^{x}-3}$

Cada operación se representa como un nodo en el grafo, y la conexion indica la dirección del flujo de datos.

a. *y* = sin (x^2 + 5*x* + 2)

- Operación 1: x^2
- Operación 2: $x^2 + 5x$
- Operación 3: $x^2 + 5x + 2$
- Operación 4: $\sin(x^2 + 5x + 2)$



Código Python:

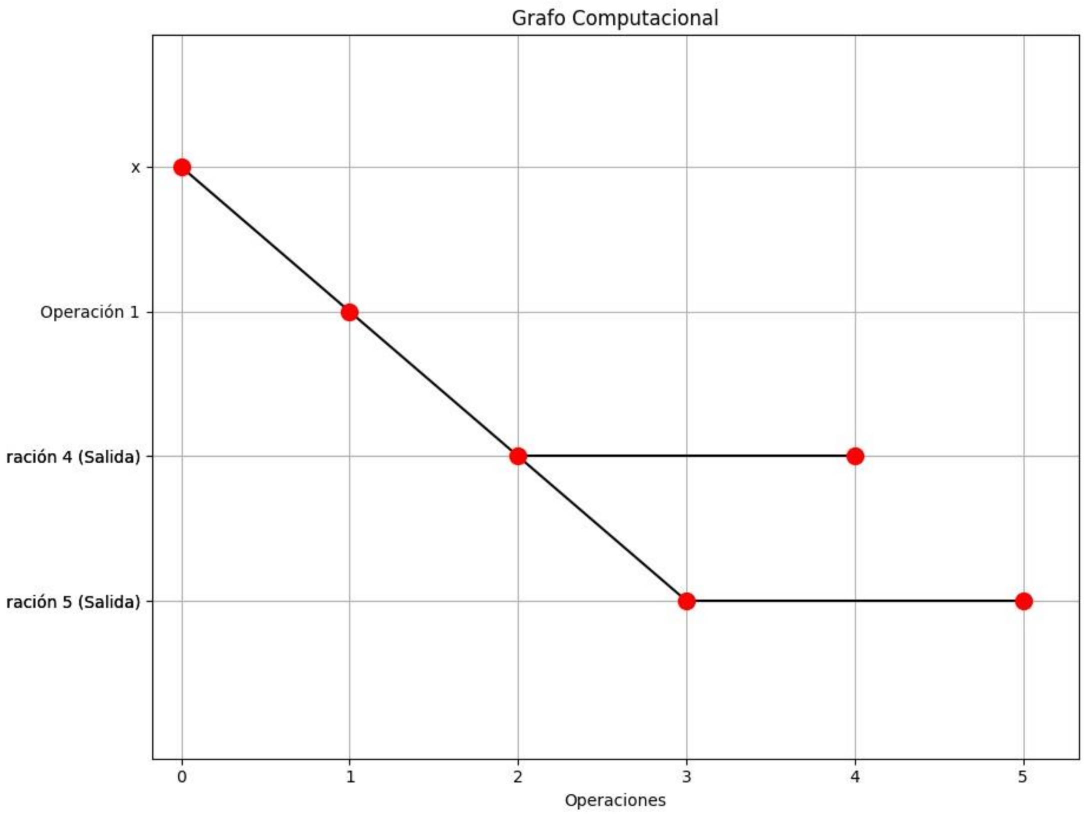
```
import matplotlib.pyplot as plt

# Definir las operaciones
operaciones = ["x", "Operación 1", "Operación 2", "Operación 3", "Operación 4 (Salida)"]

# Definir las conexiones entre las operaciones
conexiones = [("x", "Operación 1"), ("Operación 1", "Operación 2"),
              ("Operación 2", "Operación 3"), ("Operación 3", "Operación 4 (Salida)")]

# Crear el gráfico
plt.figure(figsize=(10, 6))

# Asignar alturas a las operaciones en el eje y
alturas = [4, 3, 2, 1, 2]

for conexion in conexiones:
    # Obtener las coordenadas x e y de los nodos
    x_coords = [operaciones.index(conexion[0]), operaciones.index(conexion[1])]
    y_coords = [alturas[operaciones.index(conexion[0])], alturas[operaciones.index(conexion[1])]]
    plt.plot(x_coords, y_coords, 'k-')  # Dibujar líneas entre nodos

# Dibujar nodos
for op, alt in zip(operaciones, alturas):
    plt.plot(operaciones.index(op), alt, 'ro', markersize=10)

plt.yticks(alturas, operaciones)  # Etiquetas del eje y
plt.title("Grafo Computacional")
plt.xlabel("Operaciones")
plt.ylabel("Nivel")
plt.grid(True)
plt.ylim(0, 5)  # Limitar el rango del eje y
plt.show()
```

b. $y = \cos(2x + 3) - \mathrm{ReLU}(x)$

- Operación 1: $2x$
- Operación 2: $2x + 3$
- Operación 3: $\cos(2x+3)$
- Operación 4: $\mathrm{ReLU}(x)$
- Operación 5: Resta $\cos(2x + 3) - \mathrm{ReLU}(x)$

Código Python:

```
import matplotlib.pyplot as plt

# Definir las operaciones
operaciones = ["x", "Operación 1", "Operación 2", "Operación 3", "Operación 4 (Salida)", "Operación 5 (Salida)"]

# Definir las conexiones entre las operaciones
conexiones = [("x", "Operación 1"), ("Operación 1", "Operación 2"),
              ("Operación 2", "Operación 3"), ("Operación 2", "Operación 4 (Salida)"),
              ("Operación 3", "Operación 5 (Salida)")]

# Crear el gráfico
plt.figure(figsize=(10, 8))

# Asignar alturas a las operaciones en el eje y
alturas = [4, 3, 2, 1, 2, 1]

for conexion in conexiones:
    x_coords = [operaciones.index(conexion[0]), operaciones.index(conexion[1])]
    y_coords = [alturas[operaciones.index(conexion[0])], alturas[operaciones.index(conexion[1])]]
    plt.plot(x_coords, y_coords, 'k-')

# Dibujar nodos
for op, alt in zip(operaciones, alturas):
    plt.plot(operaciones.index(op), alt, 'ro', markersize=10)

plt.yticks(alturas, operaciones)
plt.title("Grafo Computacional")
plt.xlabel("Operaciones")
plt.ylabel("Nivel")
plt.grid(True)
plt.ylim(0, 5)
plt.show()
```

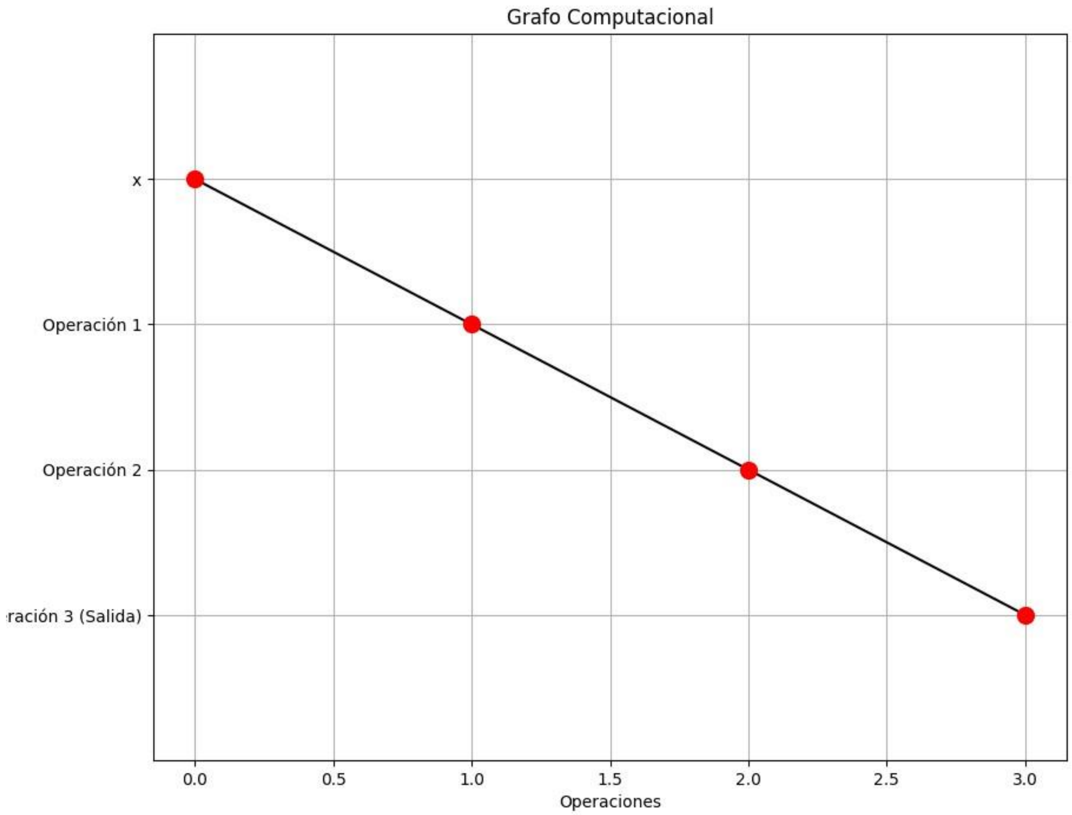

Grafo Computacional
x
Operación 1
ración 4 (Salida)
ración 5 (Salida)
0
1
2
3
4
5
Operaciones

c. $y = 2x / e^x - 3$

- Operación 1: e^x
- Operación 2: $e^x - 3$
- Operación 3: $2x / e^x - 3$

Código Python:

```
import matplotlib.pyplot as plt

# Definir las operaciones
operaciones = ["x", "Operación 1", "Operación 2", "Operación 3 (Salida)"]
conexiones = [("x", "Operación 1"), ("Operación 1", "Operación 2"),
              ("Operación 2", "Operación 3 (Salida)")]

# Crear el gráfico
plt.figure(figsize=(10, 8))

# Asignar alturas a las operaciones en el eje y
alturas = [4, 3, 2, 1]

for conexion in conexiones:
    x_coords = [operaciones.index(conexion[0]), operaciones.index(conexion[1])]
                          y_coords             =             [alturas[operaciones.index(conexion[0])],
alturas[operaciones.index(conexion[1])]]
    plt.plot(x_coords, y_coords, 'k-')

# Dibujar nodos
for op, alt in zip(operaciones, alturas):
    plt.plot(operaciones.index(op), alt, 'ro', markersize=10)

plt.yticks(alturas, operaciones)
plt.title("Grafo Computacional")
plt.xlabel("Operaciones")
plt.ylabel("Nivel")
plt.grid(True)
plt.ylim(0, 5)
plt.show()
```

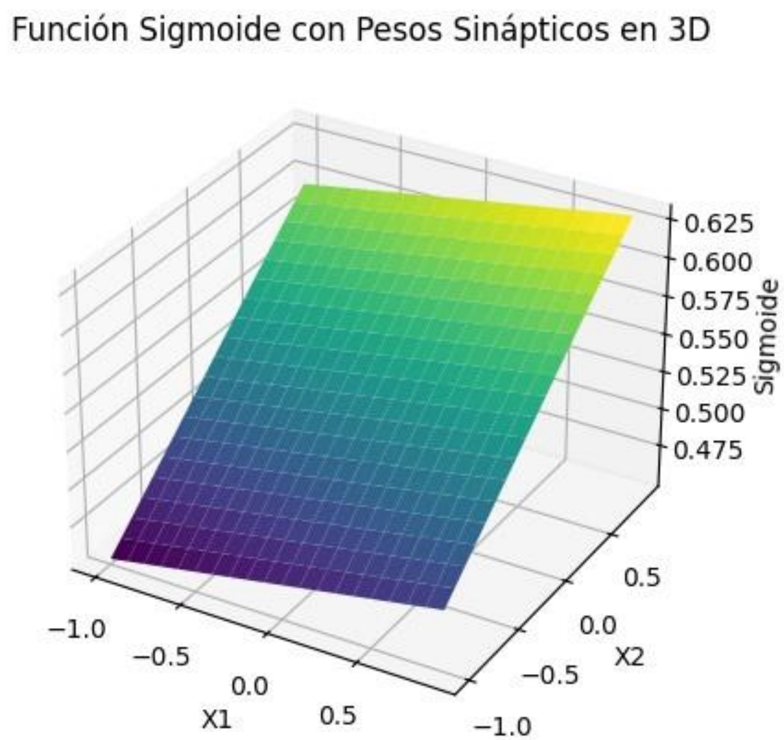


Grafo Computacional
x
Operación 1
Operación 2
ración 3 (Salida)
0.0
0.5
1.0
1.5
2.0
2.5
3.0
Operaciones

2. Para 5 épocas mostrar los pesos sinápticos, de la función sigmoide en 3d donde los pesos iniciales son [0.1,0.3,-0.7] y el bias es 0.25. las entradas del grafo se colocan en la siguiente tabla.

X1	X2	X3	Y
.3	.2	.1	.7
.1	.1	-.1	.0
.5	-.2	-.4	-.5

- Época 1: Pesos = [0.09549605 0.30146127 -0.69594019], Bias = 0.23732116342219975
- Época 2: Pesos = [0.09100841 0.3029286 -0.69187558], Bias = 0.22469082984016928
- Época 3: Pesos = [0.0865375 0.30440185 -0.68780681], Bias = 0.2121107037594517
- Época 4: Pesos = [0.08208374 0.30588089 -0.68373452], Bias = 0.19958244728887575
- Época 5: Pesos = [0.07764753 0.30736562 -0.67965934], Bias = 0.18710767864096078



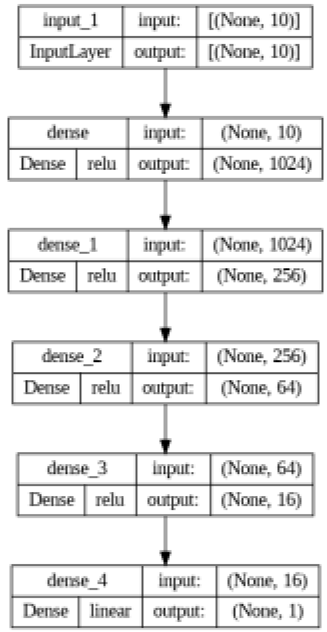
Código Python:

```
import numpy as np
import matplotlib.pyplot as plt

# Función sigmoide
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# Datos de entrada y salida
X = np.array([[0.3, 0.2, 0.1],
              [0.1, 0.1, -0.1],
              [0.5, -0.2, -0.4]])
y = np.array([[0.7],
              [0.0],
              [-0.5]])

# Pesos iniciales y bias
weights = np.array([0.1, 0.3, -0.7])
bias = 0.25

# Parámetros del modelo
learning_rate = 0.1
epochs = 5

# Entrenamiento del modelo
for epoch in range(epochs):
    # Forward pass
    weighted_sum = np.dot(X, weights) + bias
    predictions = sigmoid(weighted_sum)

    # Cálculo del error
    error = y - predictions

    # Cálculo de los gradientes
    d_weights = np.mean(X * error * predictions * (1 - predictions), axis=0)
    d_bias = np.mean(error * predictions * (1 - predictions))

    # Actualización de los pesos y el bias
    weights += learning_rate * d_weights
    bias += learning_rate * d_bias

    # Mostrar los pesos sinápticos
    print(f"Época {epoch + 1}: Pesos = {weights}, Bias = {bias}")

# Calcular Z utilizando los valores de X e Y del meshgrid
```

```python
Z = sigmoid(weights[0] * X[:, 0] + weights[1] * X[:, 1] + weights[2] * X[:, 2] + bias)

# Gráfico de los pesos sinápticos
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
x = np.arange(-1, 1, 0.1)
y = np.arange(-1, 1, 0.1)
X, Y = np.meshgrid(x, y)
Z = sigmoid(weights[0] * X + weights[1] * Y + bias)
ax.plot_surface(X, Y, Z, cmap='viridis')
ax.set_xlabel('X1')
ax.set_ylabel('X2')
ax.set_zlabel('Sigmoide')
plt.title('Función Sigmoide con Pesos Sinápticos en 3D')
plt.show()
```

3. Realizar el código fuente que representa los siguientes modelos.

   a. Para regresión

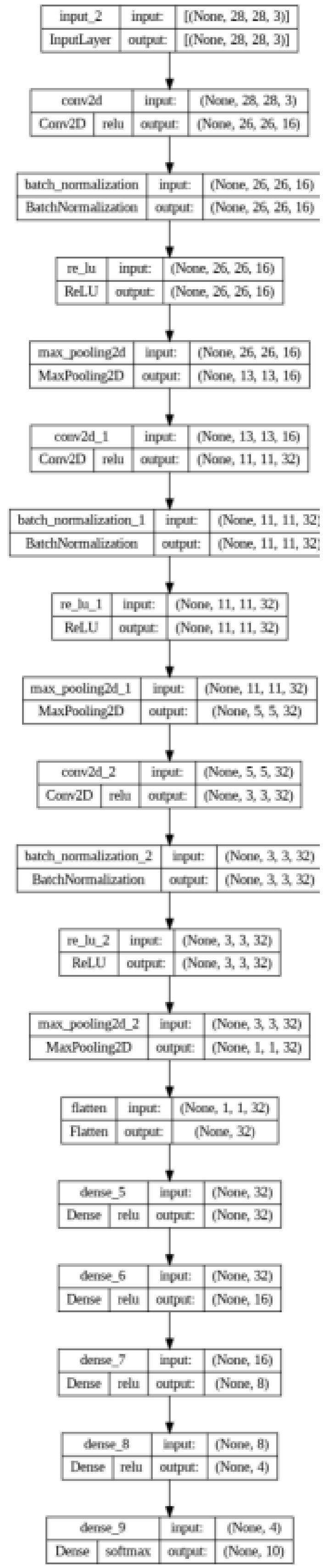


Código Python:

```python
from tensorflow import keras

# Crear el modelo
model = keras.models.Sequential()

# Agregar la capa de entrada
model.add(keras.layers.Input(shape=(10,)))

# Agregar capas densas con activación relu
model.add(keras.layers.Dense(units=1024, activation='relu'))
model.add(keras.layers.Dense(units=256, activation='relu'))
model.add(keras.layers.Dense(units=64, activation='relu'))
model.add(keras.layers.Dense(units=16, activation='relu'))

# Agregar capa densa con activación lineal
model.add(keras.layers.Dense(units=1, activation='linear'))

# Mostrar el resumen del modelo
model.summary()
```

Layer (type)	Output Shape	Param #
dense	(None, 1024)	11,264
dense_1	(None, 256)	262,400
dense_2	(None, 64)	16,448
dense_3	(None, 16)	1,040
dense_4	(None, 1)	17

b. Para clasificación



Código Python:

```python
import tensorflow as tf
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Conv2D, BatchNormalization, ReLU, MaxPooling2D, Flatten, Dense
import pandas as pd

# Define the model
input_layer = Input(shape=(28, 28, 3))
x = Conv2D(16, kernel_size=(3, 3), activation='relu')(input_layer)
x = BatchNormalization()(x)
x = MaxPooling2D(pool_size=(2, 2))(x)
x = Conv2D(32, kernel_size=(3, 3), activation='relu')(x)
x = BatchNormalization()(x)
x = MaxPooling2D(pool_size=(2, 2))(x)
x = Flatten()(x)
x = Dense(32, activation='relu')(x)
x = Dense(16, activation='relu')(x)
x = Dense(8, activation='relu')(x)
x = Dense(4, activation='relu')(x)
output_layer = Dense(10, activation='softmax')(x)
model = Model(inputs=input_layer, outputs=output_layer)

# Compile the model
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

# Crear un DataFrame con los datos de la tabla
data = {
    "Capa": ["Input", "Conv2D-1", "BatchNormalization-1", "MaxPooling2D-1",
             "Conv2D-2", "BatchNormalization-2", "MaxPooling2D-2",
             "Flatten", "Dense-1", "Dense-2", "Dense-3", "Dense-4", "Output"],
    "Tipo": ["Input", "Conv2D", "BatchNormalization", "MaxPooling2D",
             "Conv2D", "BatchNormalization", "MaxPooling2D",
             "Flatten", "Dense", "Dense", "Dense", "Dense", "Dense"],
    "Salida": ["(28, 28, 3)", "(26, 26, 16)", "(26, 26, 16)", "(13, 13, 16)",
               "(11, 11, 32)", "(11, 11, 32)", "(5, 5, 32)",
               "800", "32", "16", "8", "4", "10"],
    "Parámetros": [0, 448, 64, 0, 4640, 128, 0, 0, 25632, 528, 136, 36, 50]
}

df = pd.DataFrame(data)

# Imprimir el modelo
print("Arquitectura del Modelo:")
```

```
print(model.summary())
print("\n")

# Imprimir el DataFrame como tabla
print("Tabla de Capas:")
print(df)
```

Layer (type)	Output Shape	Param #
input_layer (InputLayer)	(None, 28, 28, 3)	0
conv2d (Conv2D)	(None, 26, 26, 16)	448
batch_normalization	(None, 26, 26, 16)	64
(BatchNormalization)		
max_pooling2d (MaxPooling2D)	(None, 13, 13, 16)	0
conv2d_1 (Conv2D)	(None, 11, 11, 32)	4,640
batch_normalization_1	(None, 11, 11, 32)	128
(BatchNormalization)		
max_pooling2d_1 (MaxPooling2D)	(None, 5, 5, 32)	0
	(None, 800)	0
dense (Dense)	(None, 32)	25,632
dense_1 (Dense)	(None, 16)	528
dense_2 (Dense)	(None, 8)	136
dense_3 (Dense)	(None, 4)	36
dense_4 (Dense)	(None, 10)	50

4. Configurar un autoencoder para el conjunto de datos fashion mnist https://keras.io/api/datasets/fashion_mnist/, para ello:
    a. aplanar las imágenes y normalizar.
    b. Configurar para el encoder la siguiente arquitectura [256,128,64,32].
    c. Entrenar el modelo con Early Stoping y BatchNormalization, para 1000 epocas.
    d. Mostrar 12 salidas aleatorias del autoencoder y compararlas con las imágenes originales.

Código Python:

```python
import numpy as np
import matplotlib.pyplot as plt
from keras.datasets import fashion_mnist
from keras.models import Sequential, Model
from keras.layers import Dense, Input
from keras.callbacks import EarlyStopping
# Importar BatchNormalization desde tensorflow.keras.layers
from tensorflow.keras.layers import BatchNormalization

# Cargar el conjunto de datos Fashion MNIST
(x_train, _), (x_test, _) = fashion_mnist.load_data()

# Normalizar y aplanar las imágenes
x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.
x_train = x_train.reshape((len(x_train), np.prod(x_train.shape[1:])))
x_test = x_test.reshape((len(x_test), np.prod(x_test.shape[1:])))

# Configurar la arquitectura del autoencoder
input_dim = x_train.shape[1]
encoding_dim = 32

input_img = Input(shape=(input_dim,))
encoded = Dense(256, activation='relu')(input_img)
encoded = BatchNormalization()(encoded)
encoded = Dense(128, activation='relu')(encoded)
encoded = BatchNormalization()(encoded)
encoded = Dense(64, activation='relu')(encoded)
encoded = BatchNormalization()(encoded)
encoded = Dense(encoding_dim, activation='relu')(encoded)

decoded = Dense(64, activation='relu')(encoded)
decoded = Dense(128, activation='relu')(decoded)
decoded = Dense(256, activation='relu')(decoded)
decoded = Dense(input_dim, activation='sigmoid')(decoded)
```

```python
autoencoder = Model(input_img, decoded)

# Compilar el modelo
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')

# Configurar Early Stopping
early_stopping = EarlyStopping(monitor='val_loss', patience=5, verbose=1)

# Entrenar el modelo
history = autoencoder.fit(x_train, x_train,
                epochs=1000,
                batch_size=256,
                shuffle=True,
                validation_data=(x_test, x_test),
                callbacks=[early_stopping])

# Mostrar 12 salidas aleatorias del autoencoder y compararlas con las imágenes originales
decoded_imgs = autoencoder.predict(x_test)

n = 12
plt.figure(figsize=(20, 4))
for i in range(n):
    # Mostrar la imagen original
    ax = plt.subplot(2, n, i + 1)
    plt.imshow(x_test[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

    # Mostrar la reconstrucción
    ax = plt.subplot(2, n, i + 1 + n)
    plt.imshow(decoded_imgs[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.show()
```

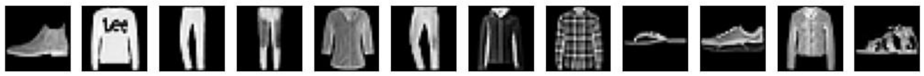
```
Epoch 1/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 15s 33ms/step - loss: 0.4215 - val_loss: 0.3384
Epoch 2/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 8s 33ms/step - loss: 0.3086 - val_loss: 0.3040
Epoch 3/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 9s 40ms/step - loss: 0.3015 - val_loss: 0.2978
Epoch 4/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 8s 35ms/step - loss: 0.2966 - val_loss: 0.2951
Epoch 5/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 9s 39ms/step - loss: 0.2928 - val_loss: 0.2931
Epoch 6/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 9s 37ms/step - loss: 0.2907 - val_loss: 0.2914
Epoch 7/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 8s 33ms/step - loss: 0.2884 - val_loss: 0.2910
Epoch 8/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 8s 35ms/step - loss: 0.2877 - val_loss: 0.2887
Epoch 9/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 8s 33ms/step - loss: 0.2865 - val_loss: 0.2878
Epoch 10/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 9s 36ms/step - loss: 0.2860 - val_loss: 0.2861
Epoch 11/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 8s 32ms/step - loss: 0.2848 - val_loss: 0.2873
Epoch 12/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 8s 33ms/step - loss: 0.2840 - val_loss: 0.2845
Epoch 13/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 7s 31ms/step - loss: 0.2833 - val_loss: 0.2844
Epoch 14/1000
```

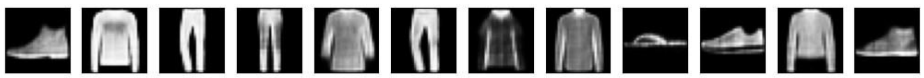
```
235/235 ━━━━━━━━━━━━━━━━━━━━ 8s 32ms/step - loss: 0.2822 -
val_loss: 0.2843
Epoch 15/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 7s 32ms/step - loss: 0.2826 -
val_loss: 0.2834
Epoch 16/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 8s 33ms/step - loss: 0.2818 -
val_loss: 0.2821
Epoch 17/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 10s 40ms/step - loss: 0.2811
- val_loss: 0.2822
Epoch 18/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 8s 32ms/step - loss: 0.2805 -
val_loss: 0.2816
Epoch 19/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 8s 33ms/step - loss: 0.2800 -
val_loss: 0.2818
Epoch 20/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 7s 31ms/step - loss: 0.2794 -
val_loss: 0.2810
Epoch 21/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 8s 34ms/step - loss: 0.2795 -
val_loss: 0.2804
Epoch 22/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 8s 33ms/step - loss: 0.2784 -
val_loss: 0.2815
Epoch 23/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 8s 32ms/step - loss: 0.2786 -
val_loss: 0.2802
Epoch 24/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 8s 33ms/step - loss: 0.2785 -
val_loss: 0.2817
Epoch 25/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 8s 32ms/step - loss: 0.2778 -
val_loss: 0.2789
Epoch 26/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 8s 33ms/step - loss: 0.2781 -
val_loss: 0.2790
Epoch 27/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 9s 37ms/step - loss: 0.2774 -
val_loss: 0.2803
```

Epoch 28/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 10s 43ms/step - loss: 0.2777 - val_loss: 0.2783
Epoch 29/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 8s 35ms/step - loss: 0.2770 - val_loss: 0.2779
Epoch 30/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 8s 33ms/step - loss: 0.2768 - val_loss: 0.2789
Epoch 31/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 12s 50ms/step - loss: 0.2757 - val_loss: 0.2780
Epoch 32/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 11s 45ms/step - loss: 0.2766 - val_loss: 0.2772
Epoch 33/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 8s 32ms/step - loss: 0.2756 - val_loss: 0.2771
Epoch 34/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 14s 59ms/step - loss: 0.2765 - val_loss: 0.2775
Epoch 35/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 13s 54ms/step - loss: 0.2760 - val_loss: 0.2772
Epoch 36/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 11s 48ms/step - loss: 0.2750 - val_loss: 0.2771
Epoch 37/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 10s 42ms/step - loss: 0.2748 - val_loss: 0.2765
Epoch 38/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 10s 42ms/step - loss: 0.2751 - val_loss: 0.2760
Epoch 39/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 12s 49ms/step - loss: 0.2744 - val_loss: 0.2757
Epoch 40/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 11s 45ms/step - loss: 0.2747 - val_loss: 0.2761
Epoch 41/1000

235/235 ━━━━━━━━━━━━━━━━━━━━ 10s 40ms/step - loss: 0.2741 - val_loss: 0.2755
Epoch 42/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 9s 36ms/step - loss: 0.2740 - val_loss: 0.2760
Epoch 43/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 9s 40ms/step - loss: 0.2739 - val_loss: 0.2752
Epoch 44/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 9s 37ms/step - loss: 0.2736 - val_loss: 0.2782
Epoch 45/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 9s 39ms/step - loss: 0.2732 - val_loss: 0.2748
Epoch 46/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 9s 38ms/step - loss: 0.2731 - val_loss: 0.2757
Epoch 47/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 9s 37ms/step - loss: 0.2734 - val_loss: 0.2757
Epoch 48/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 9s 38ms/step - loss: 0.2734 - val_loss: 0.2744
Epoch 49/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 9s 39ms/step - loss: 0.2726 - val_loss: 0.2743
Epoch 50/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 11s 46ms/step - loss: 0.2727 - val_loss: 0.2745
Epoch 51/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 11s 45ms/step - loss: 0.2730 - val_loss: 0.2741
Epoch 52/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 8s 33ms/step - loss: 0.2718 - val_loss: 0.2744
Epoch 53/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 8s 32ms/step - loss: 0.2726 - val_loss: 0.2744
Epoch 54/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 8s 34ms/step - loss: 0.2723 - val_loss: 0.2738

```
Epoch 55/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 8s 36ms/step - loss: 0.2716 -
val_loss: 0.2736
Epoch 56/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 8s 33ms/step - loss: 0.2720 -
val_loss: 0.2737
Epoch 57/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 8s 34ms/step - loss: 0.2717 -
val_loss: 0.2730
Epoch 58/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 8s 34ms/step - loss: 0.2709 -
val_loss: 0.2734
Epoch 59/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 8s 35ms/step - loss: 0.2717 -
val_loss: 0.2739
Epoch 60/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 8s 33ms/step - loss: 0.2714 -
val_loss: 0.2738
Epoch 61/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 8s 34ms/step - loss: 0.2716 -
val_loss: 0.2736
Epoch 62/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 8s 34ms/step - loss: 0.2709 -
val_loss: 0.2738
Epoch 62: early stopping
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step
```


Lee



5. Realizar los mismos puntos anteriores pero con una arquitectura convolucional [32,16,8] padding same.

Código Python:

```
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.datasets import fashion_mnist
from tensorflow.keras.models import Sequential, Model
from tensorflow.keras.layers import Conv2D, MaxPooling2D, UpSampling2D, Input
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.layers import BatchNormalization

# Cargar el conjunto de datos Fashion MNIST
(x_train, _), (x_test, _) = fashion_mnist.load_data()

# Normalizar y cambiar el formato de las imágenes
x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.
x_train = np.reshape(x_train, (len(x_train), 28, 28, 1))  # Añadimos una dimensión para el canal (1 para escala de grises)
x_test = np.reshape(x_test, (len(x_test), 28, 28, 1))

# Configurar la arquitectura del autoencoder
input_img = Input(shape=(28, 28, 1))

# Encoder
x = Conv2D(32, (3, 3), activation='relu', padding='same')(input_img)
x = MaxPooling2D((2, 2), padding='same')(x)
x = BatchNormalization()(x)
x = Conv2D(16, (3, 3), activation='relu', padding='same')(x)
x = MaxPooling2D((2, 2), padding='same')(x)
x = BatchNormalization()(x)
x = Conv2D(8, (3, 3), activation='relu', padding='same')(x)
encoded = MaxPooling2D((2, 2), padding='same')(x)

# Decoder
x = Conv2D(8, (3, 3), activation='relu', padding='same')(encoded)
x = UpSampling2D((2, 2))(x)
x = Conv2D(16, (3, 3), activation='relu', padding='same')(x)
x = UpSampling2D((2, 2))(x)
x = Conv2D(32, (3, 3), activation='relu')(x)
x = UpSampling2D((2, 2))(x)
decoded = Conv2D(1, (3, 3), activation='sigmoid', padding='same')(x)

autoencoder = Model(input_img, decoded)
```

```python
# Compilar el modelo
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')

# Configurar Early Stopping
early_stopping = EarlyStopping(monitor='val_loss', patience=5, verbose=1)

# Entrenar el modelo
history = autoencoder.fit(x_train, x_train,
                          epochs=1000,
                          batch_size=256,
                          shuffle=True,
                          validation_data=(x_test, x_test),
                          callbacks=[early_stopping])

# Mostrar 12 salidas aleatorias del autoencoder y compararlas con las imágenes originales
decoded_imgs = autoencoder.predict(x_test)

n = 12
plt.figure(figsize=(20, 4))
for i in range(n):
    # Mostrar la imagen original
    ax = plt.subplot(2, n, i + 1)
    plt.imshow(x_test[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

    # Mostrar la reconstrucción
    ax = plt.subplot(2, n, i + 1 + n)
    plt.imshow(decoded_imgs[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.show()
```

```
235/235 ━━━━━━━━━━━━━━━━━━━━ 49s 173ms/step - loss:
0.4061 - val_loss: 0.5005
Epoch 2/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 46s 195ms/step - loss:
0.3035 - val_loss: 0.3638
Epoch 3/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 45s 193ms/step - loss:
0.2966 - val_loss: 0.2996
Epoch 4/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 51s 216ms/step - loss:
0.2923 - val_loss: 0.2919
Epoch 5/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 51s 218ms/step - loss:
0.2897 - val_loss: 0.2902
Epoch 6/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 51s 215ms/step - loss:
0.2881 - val_loss: 0.2885
Epoch 7/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 48s 203ms/step - loss:
0.2868 - val_loss: 0.2893
Epoch 8/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 48s 203ms/step - loss:
0.2851 - val_loss: 0.2862
Epoch 9/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 49s 210ms/step - loss:
0.2849 - val_loss: 0.2866
Epoch 10/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 51s 217ms/step - loss:
0.2834 - val_loss: 0.2846
Epoch 11/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 50s 214ms/step - loss:
0.2827 - val_loss: 0.2832
Epoch 12/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 48s 202ms/step - loss:
0.2823 - val_loss: 0.2831
Epoch 13/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 46s 193ms/step - loss:
0.2831 - val_loss: 0.2824
Epoch 14/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 45s 190ms/step - loss:
0.2814 - val_loss: 0.2820
```

```
Epoch 15/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 45s 193ms/step - loss: 0.2799 - val_loss: 0.2835
Epoch 16/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 46s 195ms/step - loss: 0.2799 - val_loss: 0.2808
Epoch 17/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 47s 201ms/step - loss: 0.2799 - val_loss: 0.2804
Epoch 18/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 48s 204ms/step - loss: 0.2791 - val_loss: 0.2799
Epoch 19/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 47s 200ms/step - loss: 0.2787 - val_loss: 0.2802
Epoch 20/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 44s 188ms/step - loss: 0.2784 - val_loss: 0.2793
Epoch 21/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 46s 193ms/step - loss: 0.2782 - val_loss: 0.2798
Epoch 22/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 45s 193ms/step - loss: 0.2782 - val_loss: 0.2794
Epoch 23/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 44s 188ms/step - loss: 0.2770 - val_loss: 0.2808
Epoch 24/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 48s 205ms/step - loss: 0.2771 - val_loss: 0.2832
Epoch 25/1000
235/235 ━━━━━━━━━━━━━━━━━━━━ 46s 197ms/step - loss: 0.2774 - val_loss: 0.2837
Epoch 25: early stopping
313/313 ━━━━━━━━━━━━━━━━━━━━ 7s 19ms/step
```


Lee