Practica de TensorFlow

Autor: German Preciat; Universidad de Guadalajara

Carrera: Ing. Biomédica

Materia: Analisis de datos clínicos

Contacto: german.preciat@academicos.udg.mx

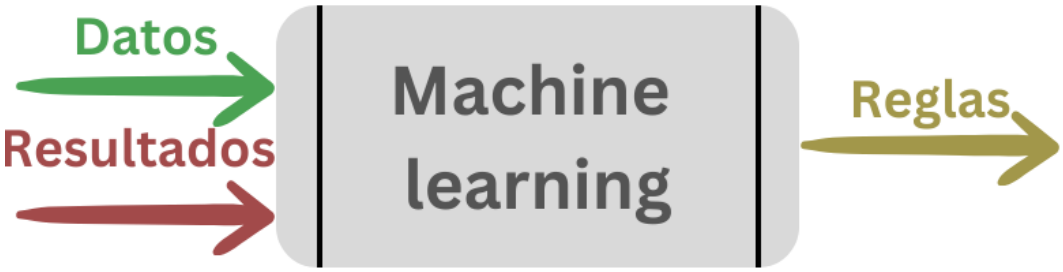
En este cuaderno, crearemos una red neuronal desde cero utilizando Python y TensorFlow. No necesitas tener conocimientos previos, ya que se explicaran todos los conceptos detalladamente.

Con el aprendizaje automático (machine learning), podemos realizar tareas complejas y asombrosas. En este notebook, nos enfocaremos en crear una red neuronal simple para comprender como funciona y para que puedas apreciar las diferencias entre el aprendizaje automático y la programación convencional.

Aprendizaje automático vs programación convencional

El aprendizaje automático opera de manera conceptualmente diferente a la programación convencional. En la programación regular, normalmente escribimos algoritmos que toman entradas y las transforman en resultados. En este enfoque, diseñamos las reglas y lógicas necesarias para lograr esta conversión.

Por otro lado, el aprendizaje automático difiere en que disponemos de un conjunto de entradas y sus respectivos resultados, pero no necesariamente conocemos el algoritmo exacto que realiza la conversión. Nuestro objetivo es crear un modelo que pueda relacionar estas entradas con los resultados deseados y, de manera autónoma, aprender el algoritmo necesario para llevar a cabo la conversión.



Esta distinción fundamental entre la programación tradicional y el aprendizaje automático se hace evidente en la práctica. Para ilustrar este concepto, observemos un escenario de acción que sea simple pero eficaz para comprender cómo funciona.

Convertir de Celsius a Fahrenheit

Imagina que deseas convertir grados Celsius a grados Fahrenheit. La fórmula o algoritmo para hacerlo es el siguiente: para Fahrenheit, igualamos Celsius por 1.8 y luego sumamos 32. Empecemos primero viendo cómo lo haríamos en programación convencional. Para ello, escribiríamos una función como la siguiente:

```
def celsius_a_fahrenheit(celsius):
    fahrenheit = celsius * 1.8 + 32
    return fahrenheit
print(celsius_a_fahrenheit(100))
```

```
212.0
```

Esta función toma como entrada los grados Celsius que deseamos transformar, aplica el algoritmo y devuelve el resultado. Hasta aquí, esto es programación convencional. Ahora, exploremos cómo lograremos esto utilizando el aprendizaje automático.

TensorFlow (redes neuronales)

Una red neuronal grande puede contener millones de neuronas conectadas y es capaz de tomar decisiones muy complejas basadas en múltiples datos y variables. No obstante, en esta ocasión, utilizaremos la red neuronal más sencilla posible para comprender a fondo su funcionamiento.

Las redes neuronales siguen ciertas reglas y conceptos. Se dividen en capas, y cada capa puede constar de una o más neuronas. Toda red neural tiene, al menos, una capa de entrada, donde se reciben los datos de entrada (en este caso, los grados Celsius que deseamos convertir), y una capa de salida, donde obtenemos el resultado calculado. En este caso, buscamos obtener los grados Fahrenheit correspondientes.

Las redes más complejas pueden incluir capas intermedias adicionales, conocidas como capas ocultas. Sin embargo, exploraremos este concepto más adelante.

Recordemos que el aprendizaje automático resulta útil cuando desconocemos la fórmula o algoritmo necesario. Entonces, por un momento, imaginemos que no conocemos la fórmula de conversión. Solo disponemos de datos de entrada en grados Celsius y sus correspondientes resultados en grados Fahrenheit. Nuestra meta es que nuestro modelo aprenda autónomamente el algoritmo. Para lograr esto, utilizaremos `TensorFlow`.

```
import tensorflow as tf
import numpy as np
```

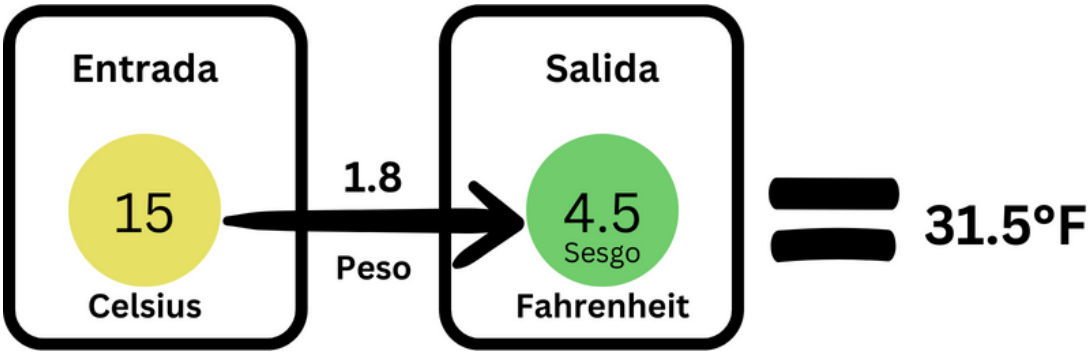
Los siguientes datos contienen una tabla con siete ejemplos, mostrando los grados Celsius y los resultados correspondientes en grados Fahrenheit. Abordaremos esto mediante una red neuronal.

```
celsius = np.array([-40, -10, 0, 8, 15, 22, 38], dtype=float)
fahrenheit = np.array([-40, 14, 32, 46, 59, 72, 100], dtype=float)
```

Para nuestro caso, colocaremos los grados Celsius en la primera neurona. Estos grados Celsius se multiplicarán por el **peso** de la conexión y luego llegarán a la siguiente neurona. Allí se les sumará el **sesgo**, y ese será nuestro resultado. En esencia, así funcionan las redes neuronales de este tipo en escenarios más complejos, que involucran redes con muchas capas, miles de neuronas y millones de conexiones. A grandes rasgos, todos siguen un proceso similar.

Ahora, en nuestro ejemplo, nuestro modelo actual está configurado de la siguiente manera: al inicio, los valores del **peso** y el **sesgo** se inician de manera totalmente **aleatoria**. Por ejemplo, inicialicemos la red con un peso de 1.8 en la conexión y un valor de 4.5 para el sesgo. No esperamos que funcione muy bien, pero lo usaremos como ejemplo.

Imaginemos que tenemos 15 grados Celsius y queremos que la red nos diga cuánto equivalen en grados Fahrenheit, o al menos lo que cree que es. Colocamos estos 15 grados Celsius en la neurona de entrada. Estos grados se multiplican por 1.8, que es el **peso** de la conexión. El resultado es 27. Luego, este valor entra en la siguiente neurona, donde se suma el **sesgo**, que es 4.5. El valor final es de 31.5. Por lo tanto, en este momento, mi red neuronal predice que 15 grados Celsius son 31.5 grados Fahrenheit.



Y con nuestra funcion inicial podemos ver cuanto es en realidad.

```
print(celsius_a_fahrenheit(15))
```

```
59.0
```

Pero, ¿cómo puede la red neuronal aprender los valores más adecuados para los **pesos** y los **sesgos**? No queremos hacerlo manualmente a través de prueba y error. Aquí es donde entra la magia del aprendizaje automático ya que puede aprender a hacer esta conversión de grados Celsius a grados Fahrenheit. Este tipo de redes puede aprender no solo esto, sino también, por ejemplo, calcular el valor de una casa en función de sus características, clasificar un correo como spam o no, detectar transacciones fraudulentas o calcular tiempos de envío, entre otras muchas aplicaciones.

¿Cómo puede lograrlo? En primer lugar, necesitamos obtener suficientes ejemplos de entradas y resultados. Ahora queremos proporcionar a la red neuronal estos ejemplos y que aprenda por sí sola la relación, es decir, que ajuste automáticamente los **pesos** y los **sesgos** para hacer predicciones lo más acertadas posible.

Para lograrlo, la red tomará todos los datos de entrada que le proporcionamos y realizará predicciones para cada uno. Recordemos que inicialmente, los pesos se inicializaron de manera aleatoria. Al principio, le irá muy mal. Dependiendo de qué tan mal le fue, ajustará los pesos y los sesgos. Si le fue muy mal, realizará un ajuste más significativo; si le fue menos mal, hará un ajuste menor. Sin embargo, dado que tenemos solo siete ejemplos, es posible que no comprenda completamente la relación y pueda ajustar el peso y el sesgo adecuadamente solo con estos ejemplos. Veámoslo por nosotros mismos. Sigamos adelante.

Mi primer red neuronal

Ahora diseñamos nuestro modelo de red neuronal, como la que vimos anteriormente. Para hacerlo, además de usar `TensorFlow`, utilizaremos el framework `Keras`. `Keras` simplemente nos permite crear redes neuronales de manera sencilla y ahorra muchas líneas de código. En este caso, podemos especificar las capas de entrada y salida por separado, o bien, ahorrarnos un paso y especificar solo la capa de salida. Veamos cómo hacerlo.

Comencemos creando una variable llamada "`modelo`" y la inicializaremos como una capa de tipo densa. En `Keras`, las capas densas son aquellas que tienen conexiones desde cada neurona hacia todas las neuronas de la capa siguiente. Dado que aquí tenemos solo dos neuronas, no hay mucho más que conectar. Al definir la capa, indicamos las unidades o neuronas de la misma. En este caso, la capa de salida solo tiene una neurona. Además, usemos la variable "`input_shape`" para indicar que tenemos una entrada con una neurona. Esto automáticamente registra la capa de entrada con una neurona.

Una vez que hemos definido estas capas, necesitamos utilizar un modelo de Keras para combinarlas y poder trabajar con él. Existen varios tipos de modelos, pero en este caso, utilizaremos el modelo secuencial.

```
capa = tf.keras.layers.Dense(units=1, input_shape=[1])
modelo = tf.keras.Sequential([capa])
```

Ahora que tenemos el modelo listo, el siguiente paso es compilarlo. La compilación prepara el modelo para el entrenamiento. Recordemos que el entrenamiento en el aprendizaje automático no es magia, sino matemáticas y cálculos. Sin embargo, debemos indicar algunas propiedades para que el modelo procese esas matemáticas de la manera que queremos.

En este momento, indicaremos solo dos propiedades: el optimizador y la función de pérdida. Para el optimizador, usaremos uno llamado "**Adam**". Puedes encontrar detalles sobre este algoritmo en muchos lugares, pero en resumen, permite a la red saber cómo ajustar los pesos y sesgos de manera eficiente para aprender gradualmente en lugar de empeorar. Le indicaremos un valor numérico llamado "tasa de aprendizaje", que determina cuánto ajustar los **pesos** y **sesgos**. Si usamos un valor muy pequeño, la red aprenderá muy lentamente, y si es muy grande, podría pasar del valor esperado sin lograr cambios suficientemente precisos. Usaremos un valor de 0.1, pero puedes probar con otros valores más pequeños.

Para la función de pérdida, utilizaremos una llamada "mean_squared_error" (Error Cuadrático Medio). A pesar de su nombre poco atractivo, esta función considera que es peor tener unos pocos errores grandes que una gran cantidad de errores pequeños.

```
modelo.compile(
    optimizer=tf.keras.optimizers.Adam(0.1),
    loss='mean_squared_error'
)
```

Perfecto, hemos compilado y preparado nuestro modelo. Ahora, vamos a entrenarlo. Para entrenar, utilizamos la función `fit` y le proporcionamos los datos de entrada y los resultados esperados, es decir, los grados Celsius y Fahrenheit. También le decimos cuántas épocas (vueltas) queremos que intente. Recuerda que tenemos solo 7 datos, así que una época significa revisar los 7 datos una sola vez. Necesitamos darle muchas vueltas a los 7 datos para permitir que el modelo se optimice lo mejor posible. Por ahora, indicaremos 1000 épocas. Veremos cómo funciona y luego se puede ajustar. Además, configuraré `verbose` como "False" para evitar una gran cantidad de impresiones durante el entrenamiento.

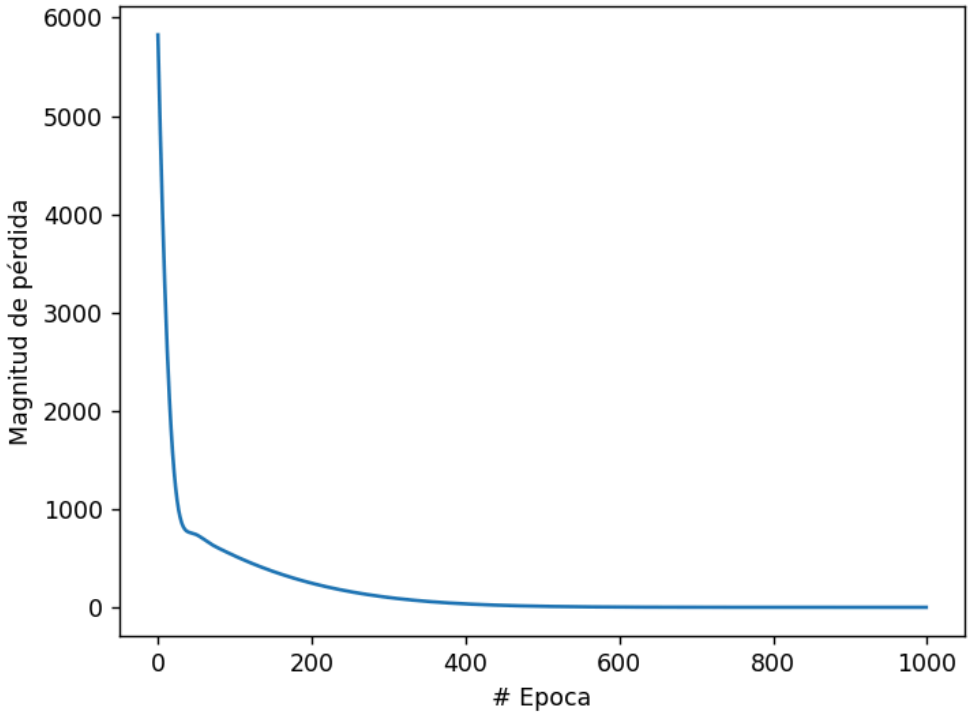
```
print("Comenzando entrenamiento...")
historial = modelo.fit(celsius, fahrenheit, epochs=1000, verbose=False)
print("Modelo entrenado!")
```

```
Comenzando entrenamiento...
Modelo entrenado!
```

¡Y listo! El modelo está entrenado y listo para hacer predicciones.

Antes de intentar hacer predicciones, echemos un vistazo al resultado de la función de pérdida. Esta función básicamente nos dice qué tan mal están los resultados de la red en cada época (vuelta) que dio. Como le dimos 1000 épocas, esos resultados de pérdida se muestran aquí abajo.

```
import matplotlib.pyplot as plt
plt.xlabel("# Epoca")
plt.ylabel("Magnitud de pérdida")
plt.plot(historial.history["loss"])
plt.show()
```



Podemos observar que a medida que la red realiza más épocas, los errores disminuyen. En realidad, por lo que podemos ver, no necesitamos 1000 épocas; 500 o 600 serán suficientes porque después la mejora es mínima, y se mantiene prácticamente en el mismo lugar.

Muy bien, hagamos una predicción de convertir 100 grados Celsius a Fahrenheit.

```
print("Hagamos una predicción!")
resultado = modelo.predict([100.0])
print("El resultado es " + str(resultado) + " fahrenheit!")
```

```
Hagamos una predicción!
1/1 [==============================] - 0s 88ms/step
El resultado es [[211.74075]] fahrenheit!
```

211.74! Es bastante cercano, lo suficiente como para considerarlo un buen resultado. Un punto interesante es examinar la estructura interna de la red y los valores asignados a los **pesos** y **sesgos** después del entrenamiento. Utilizaremos lo siguiente para imprimir los valores.

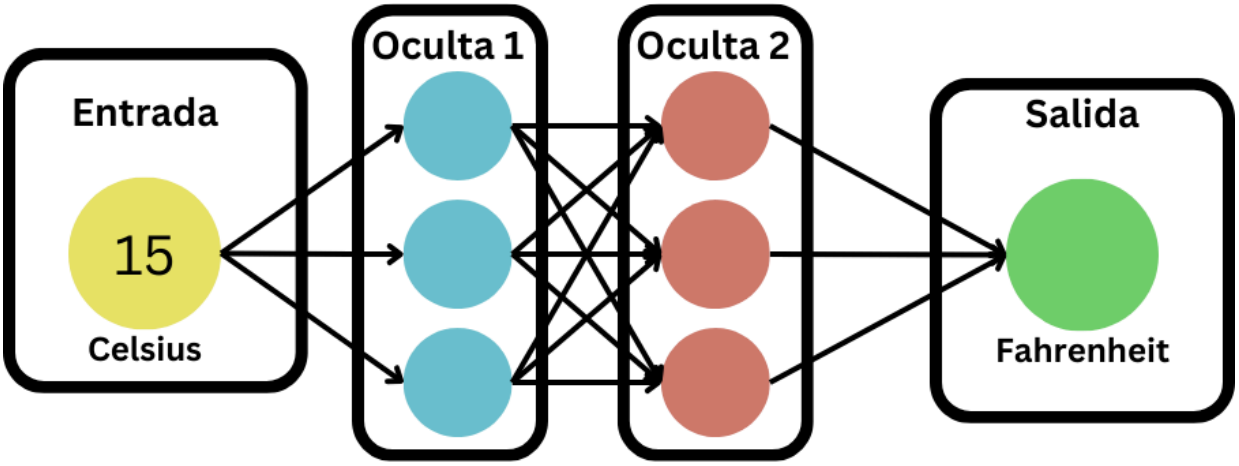
```
print("Variables internas del modelo")
print(capa.get_weights())
```

```
Variables internas del modelo
[array([[1.7983825]], dtype=float32), array([31.902498], dtype=float32)]
```

Solo hay dos valores. El **peso** se asignó como 1.79 y el **sesgo** como 31.9. Sigamos el flujo en nuestra red. Tomamos una entrada de 100 grados. 100 se multiplica por 1.79 y luego se le suma 31.9. Volvamos a revisar la fórmula para convertir de Celsius a Fahrenheit. 1.79 es prácticamente igual a 1.8, y 31.9 es casi 32. La red, sin conocimiento de la fórmula de conversión, llegó a este resultado, que es prácticamente exacto. En este escenario, podríamos decir que se trata de una función lineal. La coincidencia permitió que la fórmula de conversión se ajustara perfectamente a esta pequeña red. En esencia, la red realizó una multiplicación y una suma.

Redes mas complejas

Si agregamos más capas y neuronas a la red, podemos obtener un modelo más complejo y poderoso. Agregar dos capas intermedias (ocultas) con tres neuronas cada una significa que la red tendrá más capacidad para capturar relaciones y patrones en los datos. Aunque esto podría permitir que el modelo se ajuste mejor a los datos de entrenamiento, también puede aumentar el riesgo de sobreajuste si no se tiene suficiente información de entrenamiento.



Para hacer el proceso con esta red más grande, seguiríamos el mismo enfoque que hemos utilizado hasta ahora. Necesitaríamos alimentar al modelo con los datos de entrada y los resultados esperados, compilarlo con las mismas propiedades de optimización y función de pérdida, y luego entrenarlo durante un número adecuado de épocas.

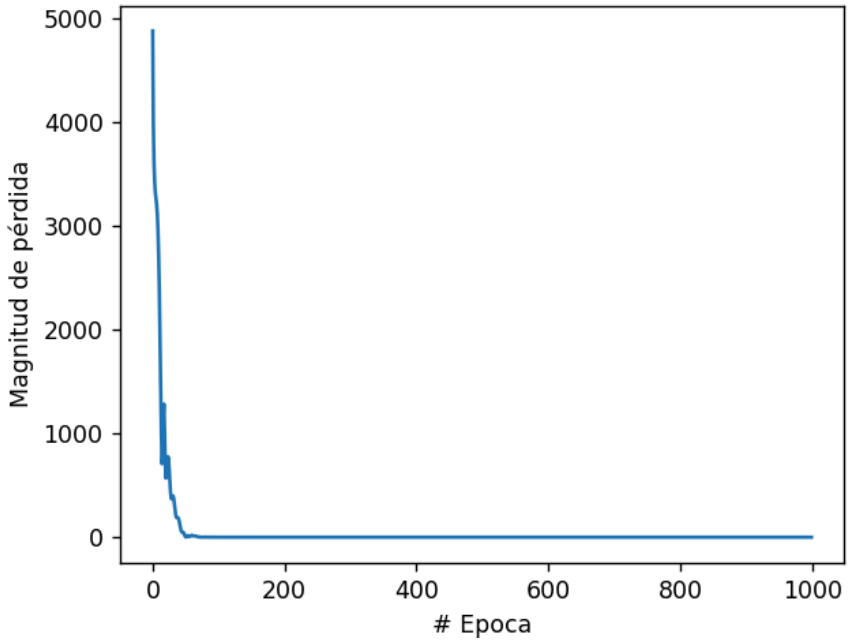
Es importante mencionar que, con una red más grande, el proceso de entrenamiento podría tomar más tiempo y requerir más datos de entrenamiento para obtener buenos resultados. Además, se podría necesitar un ajuste más cuidadoso de los hiperparámetros para evitar el sobreajuste.

En resumen, agregar capas y neuronas a la red puede mejorar su capacidad de aprendizaje, pero también conlleva desafíos adicionales que deben abordarse para obtener un modelo efectivo.

```
oculta1 = tf.keras.layers.Dense(units=3, input_shape=[1])
oculta2 = tf.keras.layers.Dense(units=3)
salida = tf.keras.layers.Dense(units=1)
modelo = tf.keras.Sequential([oculta1, oculta2, salida])

modelo.compile(
    optimizer=tf.keras.optimizers.Adam(0.1),
    loss='mean_squared_error'
)

print("Comenzando entrenamiento...")
historial = modelo.fit(celsius, fahrenheit, epochs=1000, verbose=False)
print("Modelo entrenado!")

print("Hagamos una predicción!")
resultado = modelo.predict([100.0])
print("El resultado es " + str(resultado) + " fahrenheit!")

plt.xlabel("# Epoca")
plt.ylabel("Magnitud de pérdida")
plt.plot(historial.history["loss"])
plt.show()
```

```
Comenzando entrenamiento...
Modelo entrenado!
Hagamos una predicción!
1/1 [==============================] - 0s 72ms/step
```

```
El resultado es [[211.74744]] fahrenheit!
```



Vemos que el entrenamiento se tarda más, sin embargo, podemos ver que el aprendizaje fue mucho más rápido desde la época 50 o 60 y ya no aprendió nada más es decir que en este escenario una red más compleja nos dan mejores resultado. Ademas, a medida que crece su complejidad, se vuelve difícil entender la lógica detrás de cada **peso** y **sesgo**. En redes profundas, el proceso de optimización ajusta automáticamente estos valores para minimizar la pérdida y mejorar la precisión de las predicciones. El enfoque se centra en la capacidad del modelo para hacer predicciones precisas en lugar de comprender cada detalle de sus parámetros internos.

```
print("Variables internas del modelo")
print(oculta1.get_weights())
print(oculta2.get_weights())
print(salida.get_weights())
```

```
Variables internas del modelo
[array([[-0.3596088 , -0.82435125,  0.14392483]], dtype=float32), array([ 0.463765 , -4.1843667, -3.2920249], dtype=float
[array([[-0.06156484, -0.47091213,  0.16123794],
       [ 0.54735357,  1.0710267 ,  1.117031  ],
       [ 0.8025524 ,  0.34987915,  0.35991892]], dtype=float32), array([-4.180308 , -3.9378548, -3.6540153], dtype=float3
[array([[-1.0410886],
       [-1.1721176],
       [-0.7488072]], dtype=float32), array([3.8937552], dtype=float32)]
```

Este ejemplo demuestra que, incluso en problemas simples, tanto redes neuronales simples como complejas pueden aprender algoritmos para resolverlos, aunque pueden seguir caminos muy diferentes para hacerlo. La elección de la complejidad del modelo depende de la naturaleza del problema y de los recursos disponibles.

Ejercicios

Predicciones de Parámetros Biométricos

Imagina que estas en un proyecto de ingeniería biomédica y te propones desarrollar un modelo para predecir el estrés de una persona basado en su frecuencia cardiaca. Tu objetivo es construir una red neuronal artificial que pueda prever respuestas fisiológicas a partir la frecuencia cardíaca.

Conjunto de Datos:

- Entrada (X): La frecuencia cardíaca (ppm)
- Salida (Y): Nivel de estrés.

```
import pandas as pd
```

```
# Datos biometricos (diccionario)
data = {
    'ppm': [75, 82, 88, 95, 100, 105, 92, 78, 110, 85, 98, 115, 80, 93, 102,
            108, 87, 97, 118, 84],
    'stress': [3.1, 3.8, 4.2, 5.5, 6.0, 7.2, 4.5, 3.0, 9.8, 4.2, 5.7, 10, 3.2,
               5.0, 7.5, 8.8, 4.0, 6.5, 10, 4.1]
}

# NOTA:La relación entre la frecuencia cardíaca y el nivel de estrés es inventada y
# no refleja ninguna relación biomédica real.

# Crear DataFrame (tabla)
biometricData_df = pd.DataFrame(data)
```

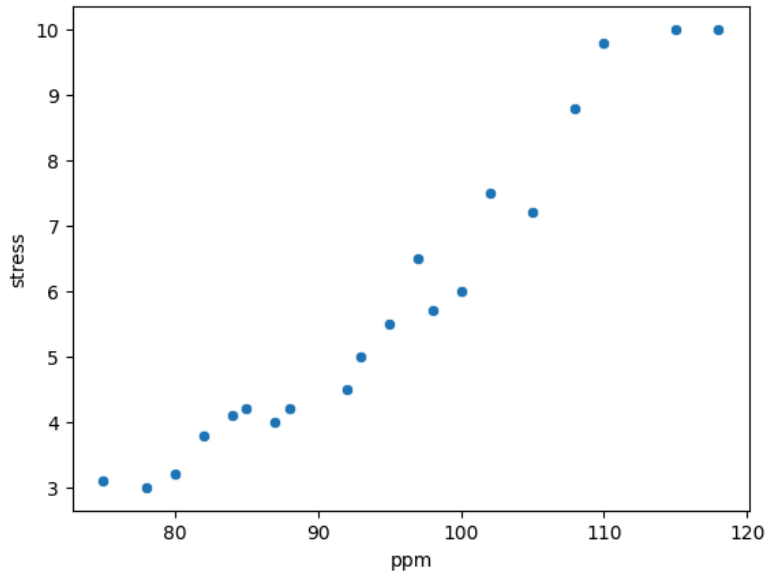
Visualizacion de datos

Seaborn, una biblioteca de visualización de datos basada en Matplotlib, y nos servira para explorar y comprender la relación entre la frecuencia cardíaca y el nivel de estrés en nuestro proyecto de ingeniería biomédica. Seaborn simplifica la creación de visualizaciones atractivas y significativas, proporcionando un conjunto de funciones de alto nivel que complementan la funcionalidad de Matplotlib.

En particular, aprovecharemos las capacidades de Seaborn para crear un scatter plot, una representación gráfica esencial para visualizar la dispersión de datos bidimensionales. Este tipo de visualización nos permitirá examinar de manera efectiva la distribución de puntos en el espacio definido por la frecuencia cardíaca (eje X) y el nivel de estrés (eje Y), proporcionando insights visuales clave sobre cualquier patrón o tendencia presente en nuestros datos.

```
import seaborn as sns
import matplotlib.pyplot as plt

sns.scatterplot(x='ppm', y='stress', data=biometricData_df)
```

```
<Axes: xlabel='ppm', ylabel='stress'>
```



Creando set de entrenamiento

```
x_train = biometricData_df['ppm']
y_train = biometricData_df['stress']
```

Creando modelo secuancial de capas densas, i.e., una neurona con un input y una salida

```
import tensorflow as tf

model = tf.keras.Sequential()
model.add(tf.keras.layers.Dense(units = 1, input_shape = [1]))
```

- `model = tf.keras.Sequential()` : Crea un modelo secuencial. Un modelo secuencial es apropiado para una pila lineal de capas, donde la salida de una capa es la entrada de la siguiente.
- `model.add(tf.keras.layers.Dense(units=1, input_shape=[1]))` : Agrega una capa densa al modelo. La capa densa es una capa completamente conectada, donde cada nodo en la capa recibe una entrada de cada nodo de la capa anterior. Los parámetros clave son:

1. `units=1` : Indica que la capa tiene un solo nodo de salida. En este contexto, el modelo está diseñado para producir una salida unidimensional.
2. `input_shape=[1]` : Especifica la forma de la entrada. Aquí, se espera que la entrada sea un tensor unidimensional con una dimensión de tamaño 1. Esto sugiere que el modelo está destinado a procesar datos univariados.

Para ver la arquitectura del modelo podemos usar

```
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_1 (Dense)             (None, 1)                 2

=================================================================
Total params: 2 (8.00 Byte)
Trainable params: 2 (8.00 Byte)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Compilar el modelo

```
model.compile(optimizer=tf.keras.optimizers.Adam(0.1), loss = 'mean_squared_error')
```

La línea `model.compile(optimizer=tf.keras.optimizers.Adam(0.1), loss='mean_squared_error')` se encarga de configurar el proceso de entrenamiento de un modelo de red neuronal en TensorFlow.

- Optimizer (Optimizador): En este caso, se utiliza el optimizador Adam, que es una variante del descenso de gradiente estocástico. El valor 0.1 especifica la tasa de aprendizaje, que controla la magnitud de los ajustes a los pesos del modelo durante el entrenamiento. Un valor más alto acelera el aprendizaje, pero un valor demasiado alto puede llevar a oscilaciones y convergencia inestable.
- Loss (Pérdida): Se establece la función de pérdida en 'mean_squared_error' (error cuadrático medio). La función de pérdida mide la discrepancia entre las predicciones del modelo y los valores reales. En el caso del error cuadrático medio, se penalizan más fuertemente las diferencias más grandes, proporcionando una medida de cuán bien se ajustan las predicciones del modelo a los datos de entrenamiento.

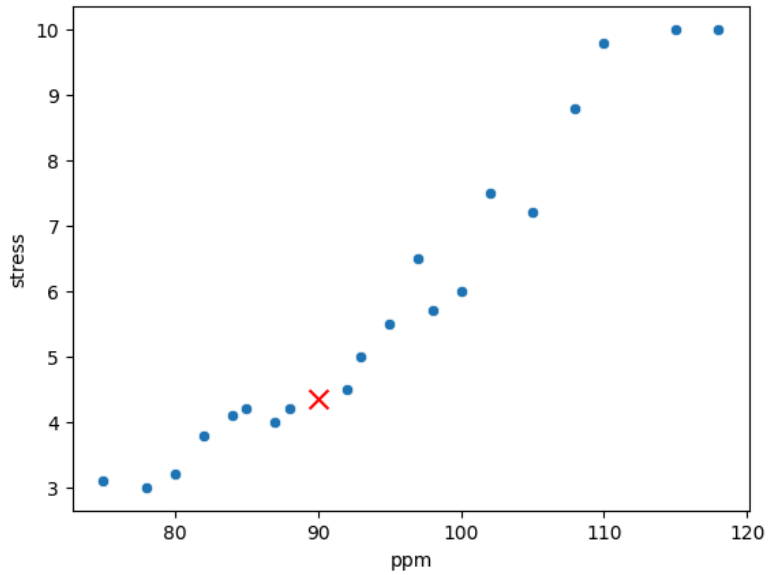
Entrenamiento del modelo

```
epochs_hist = model.fit(x_train, y_train, epochs = 1000)
```

Prediccion

```
# Prediccion
ppm = 90
stress = model.predict([Temp])
print('El nivel de estres segun la Red Neuronal sera de: ', stress)
sns.scatterplot(x='ppm', y='stress', data=biometricData_df)
plt.scatter(ppm, stress, color='red', marker='x', s=100)
# Mostrar el gráfico
plt.show()
```

```
1/1 [==============================] - 0s 66ms/step
El nivel de estres segun la Red Neuronal sera de:  [[4.3480263]]
```




```
# Normalizar imágenes
train_images, test_images = train_images / 255.0, test_images / 255.0

# Convertir etiquetas a categorías
train_labels = tf.keras.utils.to_categorical(train_labels, 10)
test_labels = tf.keras.utils.to_categorical(test_labels, 10)
```

Paso 4: Construcción del Modelo

- Crea un modelo secuencial simple.
- Añade capas convolucionales y de pooling.

```
# Crear modelo
model = models.Sequential()
```

Se crea un modelo secuencial, que es una pila lineal de capas. Los modelos secuenciales son apropiados para una pila simple de capas donde cada capa tiene exactamente un tensor de entrada y un tensor de salida.

```
# Añadir capas
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
```

Paso 5: Compilación y Entrenamiento del Modelo

- Compila el modelo con una función de pérdida y un optimizador.
- Entrena el modelo con el conjunto de datos de entrenamiento.

```
# Compilar modelo
model.add(layers.Flatten())
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))

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

# Entrenar modelo
history = model.fit(train_images[..., tf.newaxis], train_labels, epochs=10, validation_data=(test_images[..., tf.newaxis], te
```

```
Epoch 1/10
1875/1875 [==============================] - 82s 43ms/step - loss: 0.4994 - accuracy: 0.8179 - val_loss: 0.3710 - val_accu
Epoch 2/10
1875/1875 [==============================] - 65s 35ms/step - loss: 0.3176 - accuracy: 0.8833 - val_loss: 0.3481 - val_accu
```

```
Epoch 3/10
1875/1875 [==============================] - 62s 33ms/step - loss: 0.2719 - accuracy: 0.8998 - val_loss: 0.2996 - val_accu
Epoch 4/10
1875/1875 [==============================] - 62s 33ms/step - loss: 0.2438 - accuracy: 0.9111 - val_loss: 0.2647 - val_accu
Epoch 5/10
1875/1875 [==============================] - 64s 34ms/step - loss: 0.2169 - accuracy: 0.9207 - val_loss: 0.2671 - val_accu
Epoch 6/10
1875/1875 [==============================] - 64s 34ms/step - loss: 0.1972 - accuracy: 0.9270 - val_loss: 0.2630 - val_accu
Epoch 7/10
1875/1875 [==============================] - 62s 33ms/step - loss: 0.1819 - accuracy: 0.9327 - val_loss: 0.2831 - val_accu
Epoch 8/10
1875/1875 [==============================] - 62s 33ms/step - loss: 0.1659 - accuracy: 0.9384 - val_loss: 0.2567 - val_accu
Epoch 9/10
1875/1875 [==============================] - 65s 35ms/step - loss: 0.1500 - accuracy: 0.9439 - val_loss: 0.2784 - val_accu
Epoch 10/10
1875/1875 [==============================] - 62s 33ms/step - loss: 0.1377 - accuracy: 0.9473 - val_loss: 0.3193 - val_accu
```

Paso 6: Evaluación del Modelo y Visualización

- Evalúa el modelo con el conjunto de datos de prueba.
- Visualiza las métricas de entrenamiento y validación.

```
# Evaluar modelo
test_loss, test_acc = model.evaluate(test_images[..., tf.newaxis], test_labels, verbose=2)
print('\nExactitud en el conjunto de prueba:', test_acc)

# Visualizar métricas
plt.plot(history.history['accuracy'], label='accuracy')
plt.plot(history.history['val_accuracy'], label = 'val_accuracy')
plt.xlabel('Época')
plt.ylabel('Exactitud')
plt.ylim([0, 1])
plt.legend(loc='lower right')
plt.show()
```

```
313/313 - 3s - loss: 0.3193 - accuracy: 0.9002 - 3s/epoch - 9ms/step

Exactitud en el conjunto de prueba: 0.9002000093460083
```