Anexo 17

Proyecto 17: Regresión Lineal con TensorFlow

Mg. Luis Felipe Bustamante Narváez

En este ejemplo, veremos, a partir de datos aleatorios el comportamiento de un modelo de regresión lineal utilizando TensorFlow para predecir parámetros normales.

Librerías

In […

```
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
from tensorflow.keras.layers import Dense, Input
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam
```

Creamos datos aleatorios

In […

```
# Total de datos
N = 200
# 200 valores aleatorios entre -5 y 4
X = np.random.random(N)*9 - 5
# Función: w=0.5, b=-1, ruido de 0.5 para 200 valores aleatorios con media 0 y desviación 1
y = (0.5*X - 1) + np.random.randn(N)*0.5
```

Explicación

1. Se define la cantidad total de datos a generar:

N = 200

Esto significa que se van a crear **200 muestras** (puntos de datos). Esta variable se usa en las siguientes líneas para definir cuántos valores aleatorios se generarán.

---

2. Se generan **200 valores aleatorios** en el intervalo entre **-5 y 4**:

X = np.random.random(N) * 9 - 5

Aquí está lo que ocurre:

- **np.random.random(N)** genera 200 números aleatorios con distribución uniforme entre 0 y 1.
- *** 9** escala esos valores al rango [0, 9).
- **- 5** desplaza el rango a [-5, 4).

Así que `X` será un arreglo de 200 valores aleatorios distribuidos uniformemente entre -5 y 4.

---

3. Se generan los valores de `y` a partir de una **relación lineal con ruido**:

y = (0.5X - 1) + np.random.randn(N)0.5

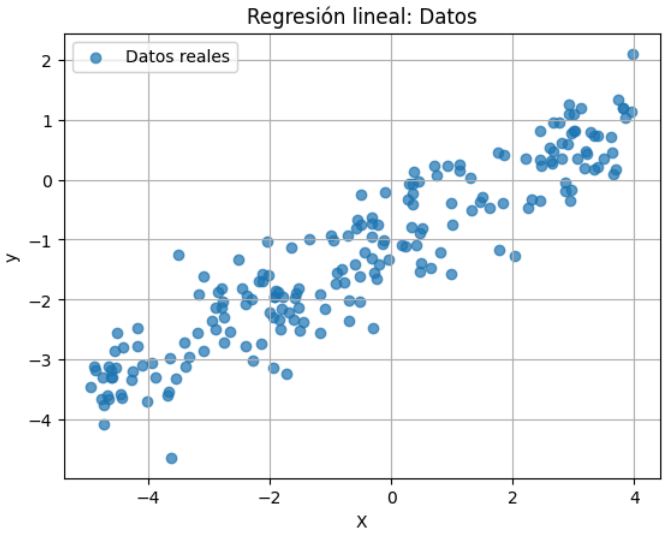
Esto representa una función lineal con pendiente **0.5** y ordenada al origen **-1**, a la que se le suma **ruido aleatorio** para simular datos reales. Desglosemos:

- **0.5 * X - 1** es una función lineal: `y = 0.5x - 1`.
- **np.random.randn(N)** genera 200 valores con distribución normal (media 0, desviación estándar 1).
- **0.5** ajusta el nivel del ruido: lo reduce para que tenga una desviación estándar de 0.5.

Entonces, `y` se compone de:

- Una parte **determinística** (la línea recta).
- Una parte **aleatoria** (el ruido), que simula variaciones naturales o errores de medición.

In [...
```
#Graficamos
plt.scatter(X, y, label='Datos reales', alpha=0.7)
plt.xlabel('X')
plt.ylabel('y')
plt.title('Regresión lineal: Datos')
plt.legend()
plt.grid(True)
plt.show()
```



Modelo

In [...
```
i = Input(shape=(1,))
x = Dense(1)(i)
modelo = Model(i, x)
```

In [...
```
modelo.summary()
```

```
Model: "functional"
```

Layer (type)	Output Shape	Param #
input_layer (InputLayer)	(None, 1)	0
dense (Dense)	(None, 1)	2

Total params: 2 (8.00 B)
Trainable params: 2 (8.00 B)
Non-trainable params: 0 (0.00 B)

In […

```
modelo.compile(
    loss='mse',
    optimizer=Adam(learning_rate=0.1),
    metrics=['mae']
)
```

Explicación

1. Se define la **capa de entrada** del modelo:

i = Input(shape=(1,))

Esto indica que el modelo recibirá **una sola característica** como entrada (por ejemplo, una variable `x` real).

- `shape=(1,)` significa que cada ejemplo de entrada es un número de una dimensión (como un valor de `x`).

---

2. Se agrega una **capa densa (neurona)** que toma la entrada:

x = Dense(1)(i)

- Se crea una **capa densa** con una sola neurona (`Dense(1)`), que recibe la entrada `i`.
- Esta capa aplica una operación lineal:
  (y = w \cdot x + b),
  donde `w` y `b` son los **pesos** y **sesgo** que el modelo aprenderá.

---

3. Se construye el modelo final especificando entrada y salida:

modelo = Model(i, x)

- Se crea un modelo Keras con `i` como entrada y `x` como salida.
- Esto es útil cuando se usa la API **funcional** de Keras (en lugar de la secuencial).

---

4. Se imprime un resumen del modelo:

modelo.summary()

Esto muestra una tabla con:

- Las capas del modelo.
- El tipo de cada capa.

- La cantidad de parámetros entrenables.

Como este modelo tiene solo **una neurona**, el número de parámetros será 2: el peso (`w`) y el sesgo (`b`).

---

5. Se **compila** el modelo, especificando cómo se va a entrenar:

```
modelo.compile(
loss='mse',
optimizer=Adam(learning_rate=0.1),
metrics=['mae']
)
```

- **loss='mse'**: Se usa el error cuadrático medio (*mean squared error*), típico en regresión.
- **optimizer=Adam(learning_rate=0.1)**: Se usa el optimizador **Adam** con una tasa de aprendizaje de **0.1**.
- **metrics=['mae']**: Se añade el **error absoluto medio** (*mean absolute error*) como métrica adicional.

Este modelo está diseñado para **aprender una relación lineal simple**, como por ejemplo los datos generados con `y = 0.5x - 1 + ruido`.

Entrenamiento

In [...

```
r = modelo.fit(
    X.reshape(-1, 1),
    y,
    epochs=800,
    batch_size=32
)
```

Explicación

1. **X.reshape(-1, 1)**

Esto **reorganiza la forma** del array `X` para que tenga una sola columna.

- `X` originalmente tiene forma `(200,)` (200 valores sueltos).
- `X.reshape(-1, 1)` convierte eso en una matriz de **200 filas y 1 columna**: forma `(200, 1)`.

Esto es necesario porque el modelo espera **entradas bidimensionales**, aunque tenga solo una característica.

---

2. **y**

Este es el vector de etiquetas (los valores reales de salida), que usamos para entrenar al modelo. No hace falta cambiar su forma porque Keras acepta vectores 1D como salidas.

---

3. **epochs=800**

Se entrena el modelo durante **800 épocas**.

- Una época significa pasar **todos los datos de entrenamiento una vez** por el modelo.

- Entrenar durante muchas épocas ayuda al modelo a **aprender mejor**, pero también puede provocar **sobreajuste** si es excesivo.

---

4. **batch_size=32**

Durante cada época, los datos se dividen en **lotes (batches)** de 32 ejemplos a la vez.

- Esto hace que el entrenamiento sea más eficiente y estable.
- En cada lote, el modelo actualiza los pesos.

---

5. **r = ...**

La variable `r` guarda el **historial del entrenamiento**.

- `r.history` contiene los valores de la **función de pérdida** (`loss`) y la **métrica** (`mae`) en cada época.
- Puedes usarlo luego para **graficar la evolución del entrenamiento**.

---

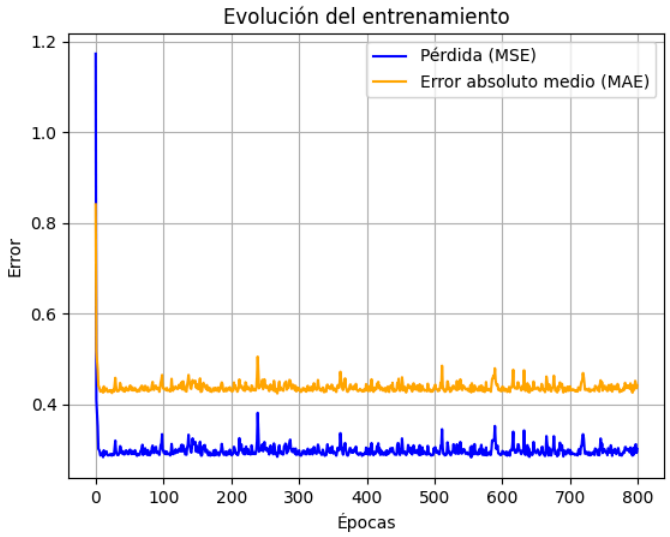
6. **Muestra de la salida**

```
Epoch 796/800
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 0.2973 - mae: 0.4457
Epoch 797/800
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 0.2961 - mae: 0.4433
Epoch 798/800
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - loss: 0.3484 - mae: 0.4778
Epoch 799/800
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 0.2947 - mae: 0.4407
Epoch 800/800
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 0.2842 - mae: 0.4274
```

Gráfica de la función de pérdida

In [...

```python
# Gráfico de la función de pérdida (loss)
plt.plot(r.history['loss'], label='Pérdida (MSE)', color='blue')
plt.plot(r.history['mae'], label='Error absoluto medio (MAE)', color='orange')
plt.xlabel('Épocas')
plt.ylabel('Error')
plt.title('Evolución del entrenamiento')
plt.legend()
plt.grid(True)
plt.show()
```

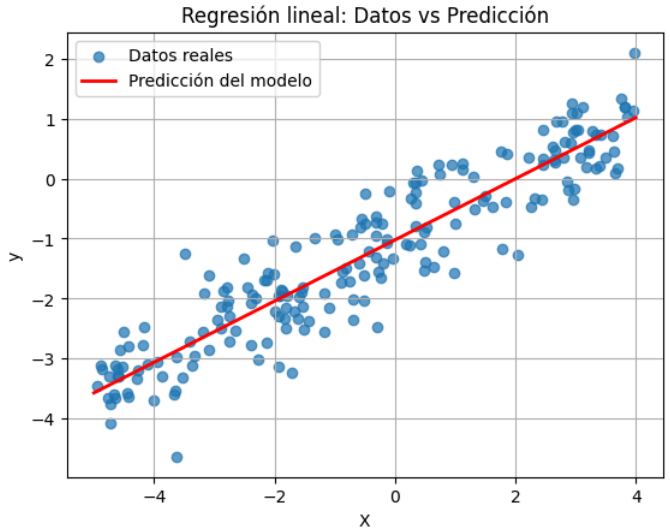


Predicción

In [...
```
X_test = np.linspace(-5, 4, 20).reshape(-1, 1)
P_test = modelo.predict(X_test)
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 53ms/step
```

In [...
```
plt.scatter(X, y, label='Datos reales', alpha=0.7)
plt.plot(X_test, P_test, color='red', label='Predicción del modelo', linewidth=2)
plt.xlabel('X')
plt.ylabel('y')
plt.title('Regresión lineal: Datos vs Predicción')
plt.legend()
plt.grid(True)
plt.show()
```



In [...
```
# Métricas
modelo.layers[1].get_weights()
```

Out[...
```
[array([[0.50980544]], dtype=float32), array([-1.0232817], dtype=float32)]
```

Explicación

Esta línea se usa para **obtener los pesos entrenados** del modelo, específicamente de la **primera capa oculta** (que en este caso es también la única capa densa del modelo).

1. **modelo.layers**

Es una lista que contiene todas las **capas del modelo**.
En este ejemplo, el modelo tiene dos capas:

- `layers[0]` → la capa de entrada: **Input(shape=(1,))**
- `layers[1]` → la capa densa: **Dense(1)**

---

2. **modelo.layers[1]**

Accede específicamente a la **capa densa**, que es donde están los **pesos (`w`) y el sesgo (`b`)** que se ajustan durante el entrenamiento.

---

3. **.get_weights()**

Devuelve una **lista con dos elementos**:

- El primer elemento es un array con el **peso (`w`)** → forma `(1, 1)` porque es una entrada y una neurona.
- El segundo elemento es un array con el **sesgo (`b`)** → forma `(1,)`.

Ejemplo de salida esperada

Si haces un `print()` a eso después de entrenar el modelo, podrías ver algo como:

```
[array([[0.498]]), array([-0.98])]
```

Esto indicaría que el modelo aprendió aproximadamente la función:

$y \approx 0.498x - 0.98$

muy cercana a la **función original** usada para generar los datos:

$y = 0.5x - 1 + \text{ruido}$

Conclusiones

TensorFlow es una biblioteca capaz de implementar redes neuronales para predecir a través de cálculos estadísticos de regresión lineal, cuando un grupo de números, se acerca a los valores reales que se desean predecir. Esta aplicación netamente teórica, puede ser implementada en clasificación de textos para NLP.

Mg. Luis Felipe Bustamante Narváez