IA
REDES NEURONALES CON TENSORFLOW

¿Qué es TensorFlow?

TensorFlow es una biblioteca de código abierto desarrollada por Google para construir y entrenar modelos de machine learning y deep learning. Está diseñada para trabajar con grandes cantidades de datos y modelos complejos de redes neuronales. Su nombre proviene del concepto de "tensores", que son estructuras de datos similares a los arrays multidimensionales.

TensorFlow ofrece:

- Operaciones matemáticas optimizadas.
- Soporte para entrenamiento en CPU y GPU.
- Interfaz de alto nivel (Keras) y bajo nivel (gráficos de flujo de datos).
- Capacidad para guardar, exportar y desplegar modelos.

¿Qué es Keras?

Keras es una API de alto nivel integrada dentro de TensorFlow (tf.keras), que permite construir y entrenar redes neuronales de forma fácil y rápida, sin preocuparse demasiado por los detalles técnicos.

Ventajas de Keras:

- Muy intuitiva y legible.
- Permite construir modelos con pocas líneas de código.
- Fácil de depurar y experimentar.
- Compatible con herramientas de producción (como TensorFlow Serving).

El Modelo: tf.keras.Sequential

En tu programa se usa la clase tf.keras.Sequential, que permite construir un modelo capa por capa, de manera lineal. Es ideal para redes en las que cada capa tiene una sola entrada y una sola salida.

```
modelo = tf.keras.Sequential([
    tf.keras.layers.Dense(units=3, input_shape=[1]),  # Capa oculta 1
    tf.keras.layers.Dense(units=3),                   # Capa oculta 2
    tf.keras.layers.Dense(units=1)                    # Capa de salida
])
```

- **Dense:** indica que cada neurona está completamente conectada a la capa anterior (fully connected).
- **units=3:** significa que esa capa tendrá 3 neuronas.
- **input_shape=[1] :** especifica que la entrada es un número real (un solo valor: el ángulo en grados).

Compilación del Modelo:

modelo.compile() Este paso configura el modelo para el entrenamiento.

Se le especifica:

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

- **El optimizador (optimizer)** se encarga de actualizar los pesos del modelo para reducir el error. En este caso:
  - **Adam(0.1):** Es un optimizador avanzado que combina las ventajas de los métodos momentum y RMSProp. Es rápido y eficiente.
  - **learning rate:** 0.1 es la tasa de aprendizaje, en otras palabras, qué tan grande es el paso que da el algoritmo para ajustar los pesos en cada iteración.
- **loss (Función de pérdida):**
  - **"mean_squared_error":** Esta función calcula el promedio del cuadrado de las diferencias entre los valores reales (radianes) y los valores predichos. El modelo busca minimizar esta función durante el entrenamiento.

Entrenamiento del Modelo

```
historial = modelo.fit(grados, radianes, epochs=1000, verbose=False)
```

Aquí se entrena el modelo durante 1000 épocas, donde:

- **Época:** Una pasada completa por todos los datos de entrenamiento.
- **verbose=False:** No imprime detalles en cada época. Durante el entrenamiento, el modelo ajusta sus pesos para que la salida se aproxime a la función real: radianes = grados $\times \left(\frac{\pi}{180}\right)$

¿Qué aprende el modelo?

Aunque no le damos explícitamente la fórmula de conversión entre grados y radianes, el modelo aprende a aproximarla a partir de los datos de entrenamiento. Al final, se comporta como una función matemática que predice correctamente los valores radianes dados los grados.

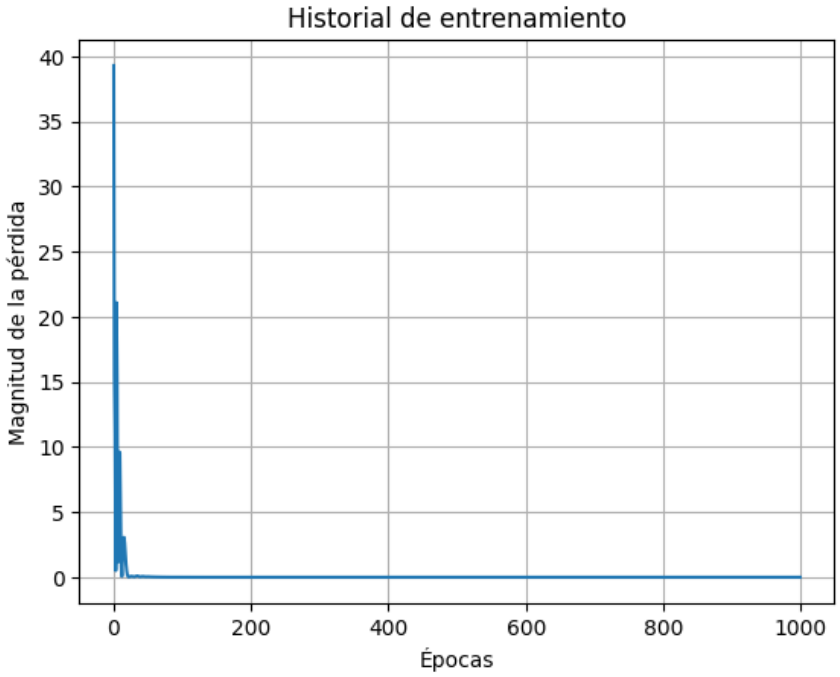
PROGRAMA PYTHON O.O COMPLETO

```
"""
Creado junio 23 de 2025
Nombre: APRENDIZAJE AUTOMÁTICO PARA CONVERSIÓN DE GRADOS A RADIANES
Versión orientada a objetos (POO)

Descripción:
Este programa entrena una red neuronal con TensorFlow para aprender
la relación entre grados y radianes. El modelo se construye usando
dos capas ocultas y una capa de salida.
"""

import tensorflow as tf
import numpy as np
import math
import matplotlib.pyplot as plt

# Definición de la clase principal
class ConversorGradosRadianes:
    def __init__(self):
        self.grados = np.array([0, 30, 45, 60, 90], dtype=float)
        self.radianes = np.array([0, math.pi/6, math.pi/4, math.pi/3, math.pi/2], dtype=float)
        self.modelo = self.construir_modelo()
        self.historial = None

    def construir_modelo(self):
```

```
        # Construcción secuencial del modelo
        modelo = tf.keras.Sequential([
            tf.keras.layers.Dense(units=3, input_shape=[1]),  # Capa oculta 1
            tf.keras.layers.Dense(units=3),                   # Capa oculta 2
            tf.keras.layers.Dense(units=1)                    # Capa de salida
        ])
        # Compilación del modelo
        modelo.compile(
            optimizer=tf.keras.optimizers.Adam(0.1),
            loss="mean_squared_error"
        )
        return modelo

    def entrenar(self, epocas=1000):
        print("Iniciando entrenamiento ...")
        self.historial = self.modelo.fit(self.grados, self.radianes, epochs=epocas, verbose=False)
        print("Entrenamiento finalizado")

    def graficar_perdida(self):
        if self.historial:
            plt.xlabel("Épocas")
            plt.ylabel("Magnitud de la pérdida")
            plt.plot(self.historial.history["loss"])
            plt.title("Historial de entrenamiento")
            plt.grid(True)
            plt.show()

    def predecir(self, valor_en_grados):
        entrada = np.array([valor_en_grados], dtype=float)
        resultado = self.modelo.predict(entrada, verbose=False)
        print(f"El resultado es {resultado[0][0]} radianes")

# Bloque principal
if __name__ == "__main__":
    conversor = ConversorGradosRadianes()
    conversor.entrenar()
    conversor.graficar_perdida()
    print("Hagamos una predicción:")
    conversor.predecir(180.0)
```

Este código genera la salida:

Iniciando entrenamiento ...
Entrenamiento finalizado



Hagamos una predicción:
El resultado es 3.141592502593994 radianes