IA
RED BACKPROPAGATION

La red **Backpropagation**, también conocida como propagación hacia atrás del error, fue desarrollada como un método de entrenamiento para redes neuronales multicapa. Aunque los fundamentos matemáticos fueron establecidos por **Paul Werbos en 1974**, fue popularizada en **1986** por **David Rumelhart, Geoffrey Hinton y Ronald Williams.** Este algoritmo permitió entrenar redes con capas ocultas, superando las limitaciones del modelo perceptrón. Su aparición marcó un hito en el aprendizaje profundo, permitiendo resolver problemas más complejos. Desde entonces, ha sido la base para muchas aplicaciones modernas de inteligencia artificial.

En qué consiste:

El algoritmo de **propagación hacia atrás** consiste en ajustar los pesos de una red neuronal para minimizar el error entre la salida obtenida y la deseada. Funciona en dos fases: primero, se realiza una **propagación hacia adelante** donde se calcula la salida de la red; luego, el error se **retropropaga** desde la salida hacia las capas anteriores. En este proceso, se calculan los gradientes del error respecto a cada peso utilizando la regla de la cadena del cálculo diferencial. Los pesos se actualizan en dirección opuesta al gradiente, reduciendo así el error. Este procedimiento se repite en cada iteración durante el entrenamiento.

Regla Delta:

La **regla generalizada del algoritmo Backpropagation** se refiere al cálculo del **gradiente del error** no solo para la capa de salida, sino también para todas las **capas ocultas** de la red. Esta regla usa la **derivada de la función de activación** y el **error propagado** desde las neuronas posteriores para ajustar cada peso. A cada conexión se le asigna un valor proporcional al impacto que tiene en el error final. De este modo, los pesos se actualizan de forma coordinada en toda la red. Gracias a esta generalización, el algoritmo puede entrenar redes profundas con múltiples capas ocultas.

Luis Fernando Alvarez V. lfalvare@utp.edu.co
Leonardo Alvarez V. lalvarez@utp.edu.co
Hernando Alvarez R. healvare@utp.edu.co

Estructura:

Estructura de la red:

- **Capas:** Capa de entrada, capas ocultas (una o más), capa de salida.
- **Elementos:** Neuronas, pesos w_{ij}, funciones de activación (típicamente sigmoide o ReLU).
- **Datos:** Conjunto de entrenamiento con entradas $\mathbf{x}$ y salidas deseadas $\mathbf{d}$.

Pasos del algoritmo Backpropagation:

1. **Inicialización:**
   - Inicializar pesos w_{ij} con valores pequeños aleatorios.
2. **Propagación hacia adelante:**
   - Calcular las salidas de cada neurona desde la entrada hasta la salida final:
   $$y_j = f\left(\sum_i w_{ij} x_i\right)$$
   donde f es la función de activación.
3. **Cálculo del error en la capa de salida:**
   - Para cada neurona de salida:
   $$\delta_j = (d_j - y_j) \cdot f'(net_j)$$
   donde f' es la derivada de la función de activación, net_j es la entrada neta a la neurona j.
4. **Retro propagación del error a capas ocultas:**
   - Para cada neurona oculta:
   $$\delta_j = f'(net_j) \cdot \sum_k \delta_k \cdot w_{jk}$$
   donde k recorre las neuronas a las que está conectada la neurona j.
5. **Actualización de pesos:**
   - Para cada peso w_{ij}:
   $$w_{ij} \leftarrow w_{ij} + \eta \cdot \delta_j \cdot y_i$$
   donde η es la tasa de aprendizaje y y_i es la salida de la neurona anterior

Luis Fernando Alvarez V. lfalvare@utp.edu.co
Leonardo Alvarez V. lalvarez@utp.edu.co
Hernando Alvarez R. healvare@utp.edu.co

6. **Repetir:**

   - Repetir los pasos 2 a 5 para todas las muestras del conjunto de entrenamiento, durante múltiples épocas hasta minimizar el error total.

Codigo:

```
# -*- coding: utf-8 -*-
"""
Programa: Red neuronal con backpropagation para convertir
grados a radianes(0   - 360  )
Version comentada con explicaciones matematicas
Autor: Usuario
Fecha: Abril 2025
"""
import numpy as np  # numpy: Manejo eficiente de matrices y operaciones matematicas

class NeuralNetwork:
    def __init__(self):
        # Inicializacion de parametros de la red
        self.input_size = 1
        # Una sola entrada:angulo en grados (escalado)
        self.hidden_size = 10
        # Numero de neuronas en la capa oculta
        self.output_size = 1
        # Una sola salida:angulo en radianes (escalado)
        self.learning_rate = 0.01
        # Tasa de aprendizaje eta, valor pequeno para ajustes suaves

        # Inicializacion aleatoria de los pesos y sesgos (bias) con valores entre 0 y 1
        self.weights_input_hidden = np.random.rand(self.input_size, self.hidden_size)
        self.bias_hidden = np.random.rand(1, self.hidden_size)
        self.weights_hidden_output = np.random.rand(self.hidden_size, self.output_size)
        self.bias_output = np.random.rand(1, self.output_size)

    def sigmoid(self, x):
        """
        Funcion de activacion sigmoide:
        f(x) = 1 / (1 + e^(-x))
        Convierte cualquier valor real en un numero entre 0 y 1.
        """
        return 1 / (1 + np.exp(-x))

    def sigmoid_derivative(self, x):
        """
        Derivada de la funcion sigmoide:
        f'(x) = f(x) * (1 - f(x))
        Se usa en el calculo del gradiente durante la retropropagacion.
        """
        return x * (1 - x)


```

Luis Fernando Alvarez V. lfalvare@utp.edu.co
Leonardo Alvarez V. lalvarez@utp.edu.co
Hernando Alvarez R. healvare@utp.edu.co

```
47      def convertir_grados_a_radianes(self, grados):
48          """
49          Entrena la red neuronal para aprender la conversion de grados a radianes
50          usando aprendizaje supervisado y luego realiza la prediccion.
51          """
52          # Datos de entrenamiento:
53          # X: grados normalizados dividiendo entre 360 (para que esten entre 0 y 1)
54          # y: radianes correspondientes, normalizados dividiendo entre 2pi
55          # (tambien entre 0 y 1)
56          X = np.array([[0], [30], [45], [60], [90],
57                        [120], [150], [180],[270], [360]]) / 360.0
58          y = np.array([
59              [0],
60              [np.pi/6],
61              [np.pi/4],
62              [np.pi/3],
63              [np.pi/2],
64              [2*np.pi/3],
65              [5*np.pi/6],
66              [np.pi],
67              [3*np.pi/2],
68              [2*np.pi]
69          ]) / (2*np.pi)
70
71          # Entrenamiento de la red neuronal usando algoritmo de retropropagacion
72          for epoch in range(10000):
73              # ------------------- PROPAGACION HACIA ADELANTE -------------------
74              # Calculo de entradas y salidas en la capa oculta
75              hidden_input = np.dot(X, self.weights_input_hidden) + self.bias_hidden
76              # z^(1) = X  W^(1) + b^(1)
77              hidden_output = self.sigmoid(hidden_input)
78              # a^(1) = sigma(z^(1))
79
80              # Calculo de entradas y salidas en la capa de salida
81              final_input=np.dot(hidden_output, self.weights_hidden_output)+self.bias_output
82              # z^(2)
83              final_output = self.sigmoid(final_input)
84              # a^(2) = salida de la red
85
86              # -------------------- CALCULO DEL ERROR --------------------
87              # Error cuadratico: E = (y - output)^2
88              error = y - final_output  # Error de prediccion (delta)
89
90              # ------------------- RETROPROPAGACION DEL ERROR -------------------
91              # Calculo del gradiente en la salida
92              d_output = error * self.sigmoid_derivative(final_output)
93              # delta^(2) = delta * sigma'(z^(2))
94
95              # Calculo del error que llega a la capa oculta
96              error_hidden = d_output.dot(self.weights_hidden_output.T)
97              # delta^(1) = delta^(2)    (W^(2))^T
98              d_hidden = error_hidden * self.sigmoid_derivative(hidden_output)
99              # delta^(1) = delta^(1) * sigma'(z^(1))
100
101             # ------------------- ACTUALIZACION DE PESOS Y BIAS -------------------
102             # Regla de actualizacion: W = W + eta * gradiente
103             self.weights_hidden_output+=hidden_output.T.dot(d_output)*self.learning_rate
```

```
            self.bias_output+=np.sum(d_output,axis=0,keepdims=True)*self.learning_rate
            self.weights_input_hidden+=X.T.dot(d_hidden)*self.learning_rate
            self.bias_hidden+=np.sum(d_hidden,axis=0,keepdims=True)*self.learning_rate

        # ------------------- PREDICCION FINAL --------------------
        # Conversion del valor ingresado (en grados) a la prediccion de radianes
        entrada = np.array([[grados / 360.0]])
        # Normalizacion de la entrada
        hidden_layer=self.sigmoid(np.dot(entrada,self.weights_input_hidden)+self.bias_hidden)
        salida=self.sigmoid(np.dot(hidden_layer,self.weights_hidden_output)+self.bias_output)
        radianes=salida[0][0]*(2*np.pi)
        # Reescalado de la salida al intervalo original
        return radianes

# ------------------- BLOQUE PRINCIPAL -------------------
if __name__ == "__main__":
    # Instancia de la red neuronal
    convertir = NeuralNetwork()
    # Solicita al usuario ingresar un angulo en grados
    grados = float(input("Ingrese los grados a convertir a radianes: "))
    # Conversion usando la red neuronal entrenada
    radianes = convertir.convertir_grados_a_radianes(grados)
    # Muestra el resultado
    print(f"{grados} grados son {radianes:.6f} radianes")
```

Ventajas y observaciones positivas:

1. **Aprende correctamente la relación grados ↔ radianes:**

   A pesar de ser una red muy simple, el modelo logra captar **de forma aproximada** la función lineal que relaciona los grados con los radianes:

   $$y = \frac{\pi}{180} \cdot x$$

   Esto demuestra que el **backpropagation funciona incluso en una red básica con una sola capa oculta**, lo cual es educativo y útil para comprender el proceso de aprendizaje.

2. **Precisión aceptable en la práctica:** Después de 10,000 iteraciones, los errores de conversión suelen ser **pequeños (del orden de 10^{-3} o menor)**, lo cual es **aceptable para muchas aplicaciones educativas y de ejemplo**.

3. **Sin necesidad de programar la fórmula directamente:** Lo más importante es que la red **aprende sola**, sin que uno le diga explícitamente la fórmula. Esto refleja el poder del aprendizaje automático para aproximar funciones.

Luis Fernando Alvarez V. lfalvare@utp.edu.co
Leonardo Alvarez V. lalvarez@utp.edu.co
Hernando Alvarez R. healvare@utp.edu.co

Limitaciones y precisión matemática:

1. **El modelo nunca será exacto como la fórmula analítica:** La red **aproxima** la función grados → radianes. Incluso después de mucho entrenamiento, **no llega a ser exacta** como el cálculo directo con np.radians().
2. **La precisión depende del conjunto de entrenamiento:** Como solo se usan 10 puntos de entrenamiento (0°, 30°, ..., 360°), la red **no ha visto todos los valores posibles**, y puede fallar o desviarse en los intermedios (por ejemplo, 75° o 123°).
3. **Función objetivo lineal, pero activación sigmoide no lo es:** El uso de la función **sigmoide** introduce **no linealidades**, lo cual puede dificultar una aproximación perfectamente lineal como la de grados → radianes. Una red con activación lineal en la salida, o sin capa oculta, podría ser incluso más precisa en este caso.

Conclusión final:

La red neuronal implementada:

- Tiene un buen **desempeño cualitativo** y es **educativa.**
- Ofrece una **aproximación razonablemente precisa**.
- **No supera a la fórmula matemática directa**, pero **sí demuestra la capacidad de las redes** para aprender relaciones entre variables.

En resumen: no es perfecta, pero para propósitos didácticos y como ejercicio de retropropagación en POO, **es excelente y muy ilustrativa**.

Luis Fernando Alvarez V. lfalvare@utp.edu.co
Leonardo Alvarez V. lalvarez@utp.edu.co
Hernando Alvarez R. healvare@utp.edu.co