UNIVERSIDAD DE GUADALAJARA

CENTRO UNIVERSITARIO DE CIENCIAS EXACTAS E INGENIERIAS

Materia: Seminario de Solución de Problemas de Inteligencia Artificial II

Sección: D05

Profesor: Diego Campos Peña

Practica 1. Ejercicio 3

Daniel Sánchez Zepeda 220286881

Introducción

La retropropagación, también conocida como backpropagation en inglés, es un algoritmo fundamental en el aprendizaje supervisado de redes neuronales artificiales. Fue desarrollado en la década de 1970 y se convirtió en la base del entrenamiento eficiente de redes neuronales profundas.

La idea central detrás de la retropropagación es calcular el gradiente de la función de pérdida con respecto a los pesos de la red neuronal. Esto permite ajustar los pesos en la dirección que minimiza la función de pérdida, lo que conduce a una mejor capacidad de la red para hacer predicciones precisas.

En resumen, la retropropagación es un algoritmo clave para entrenar redes neuronales, ya que permite ajustar los pesos de la red de manera eficiente para minimizar la función de pérdida y mejorar su capacidad predictiva.

Desarrollo/procedimiento

Codigo implementado en pyhon (Comentado)

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

class NeuralNetwork:
    """
    Clase que representa una red neuronal multicapa.
    """

    def _init_(self, layer_sizes):
        """
        Inicializa la red neuronal con capas de pesos y sesgos aleatorios.

        Args:
        layer_sizes (list): Una lista que especifica el número de neuronas en cada capa
de la red.
        """
        self.layer_sizes = layer_sizes
        self.num_layers = len(layer_sizes)
        # Inicialización aleatoria de pesos y sesgos
        self.weights = [np.random.randn(layer_sizes[i], layer_sizes[i+1]) for i in
range(self.num_layers - 1)]
        self.biases = [np.random.randn(1, layer_sizes[i+1]) for i in
range(self.num_layers - 1)]

    def forward(self, X):
        """
        Propaga las entradas hacia adelante a través de la red.

        Args:
        X (array): Entradas de la red neuronal.
```

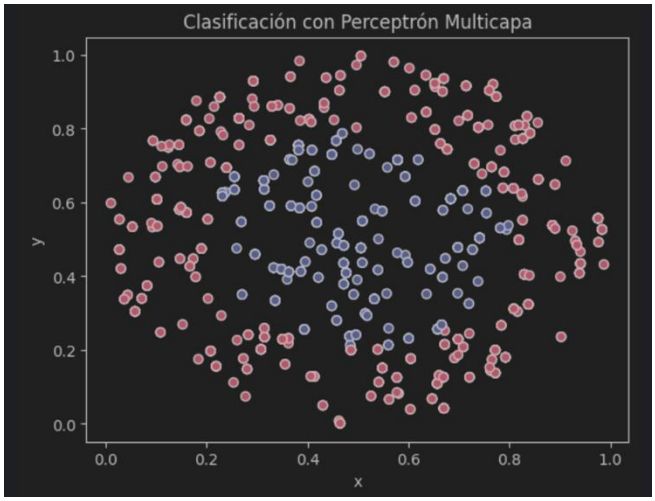
```
        Returns:
        array: Salida de la red neuronal después de la propagación hacia adelante.
        """
        # Propagación hacia adelante
        self.activations = [X]
        for i in range(self.num_layers - 1):
            weighted_input = np.dot(self.activations[-1], self.weights[i]) +
self.biases[i]
            self.activations.append(self.sigmoid(weighted_input))
        return self.activations[-1]

    def backward(self, X, y, output, learning_rate):
        """
        Realiza la retropropagación del error para ajustar los pesos y sesgos.

        Args:
        X (array): Entradas de la red neuronal.
        y (array): Etiquetas reales correspondientes a las entradas.
        output (array): Salida de la red neuronal.
        learning_rate (float): Tasa de aprendizaje para ajustar los pesos y sesgos.
        """
        # Retropropagación
        error = y - output
        deltas = [error * self.sigmoid_derivative(output)]

        for i in range(self.num_layers - 2, 0, -1):
            error = deltas[-1].dot(self.weights[i].T)
            deltas.append(error * self.sigmoid_derivative(self.activations[i]))
        deltas.reverse()

        # Actualización de pesos y sesgos
        for i in range(self.num_layers - 1):
            self.weights[i] += np.dot(self.activations[i].T, deltas[i]) * learning_rate
            self.biases[i] += np.sum(deltas[i], axis=0) * learning_rate

    def train(self, X, y, epochs, learning_rate):
        """
        Entrena la red neuronal utilizando el algoritmo de retropropagación.

        Args:
        X (array): Entradas de entrenamiento de la red neuronal.
        y (array): Etiquetas de entrenamiento correspondientes a las entradas.
        epochs (int): Número de épocas de entrenamiento.
        learning_rate (float): Tasa de aprendizaje para ajustar los pesos y sesgos.
        """
        for epoch in range(epochs):
            output = self.forward(X)
            self.backward(X, y, output, learning_rate)

    def predict(self, X):
        """
        Realiza predicciones para nuevas entradas.

        Args:
        X (array): Entradas para las cuales realizar predicciones.

        Returns:
        array: Salida predicha para las entradas dadas.
```

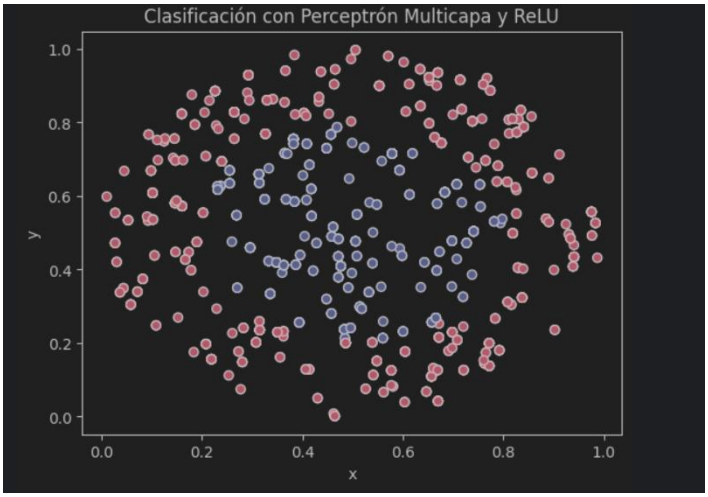
```
        """
        return np.round(self.forward(X))

    @staticmethod
    def sigmoid(x):
        """
        Función de activación sigmoide.

        Args:
        x (array): Entradas a la función sigmoide.

        Returns:
        array: Salida de la función sigmoide.
        """
        return 1 / (1 + np.exp(-x))

    @staticmethod
    def sigmoid_derivative(x):
        """
        Derivada de la función de activación sigmoide.

        Args:
        x (array): Entradas a la derivada de la función sigmoide.

        Returns:
        array: Salida de la derivada de la función sigmoide.
        """
        return x * (1 - x)


# Cargar datos
data = np.genfromtxt('mydataset.csv', delimiter=',')
X = data[:, :-1]
y = data[:, -1]

# Definir la arquitectura de la red neuronal
layer_sizes = [X.shape[1], 5, 1]

# Crear y entrenar la red neuronal
model = NeuralNetwork(layer_sizes)
model.train(X, y.reshape(-1, 1), epochs=8000, learning_rate=0.05)

# Predecir y graficar resultados
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1), np.arange(y_min, y_max, 0.1))
Z = model.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)

plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.coolwarm, edgecolors='k')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Red Neuronal para Clasificación')
plt.show()
```

Resultados

Clasificación de Datos Utilizando Perceptrón Multicapa y Retropropagación

Los resultados obtenidos en la clasificación de datos mediante el Perceptrón Multicapa y la técnica de retropropagación fueron satisfactorios. Sin embargo, al cambiar la función de activación de sigmoidal a ReLU, no se observaron cambios significativos en las gráficas. Es posible que la discrepancia entre las dos funciones de activación no sea lo bastante marcada como para alterar notablemente la apariencia de la frontera de decisión en el espacio de características.





Conclusión

La implementación exitosa del Perceptrón Multicapa mediante el algoritmo de retropropagación destaca la eficacia de esta técnica en el procesamiento y clasificación de datos. Aunque en esta investigación no se detectaron diferencias sustanciales entre el uso de funciones de activación sigmoidal y ReLU, reconocemos la importancia de explorar diversas arquitecturas de red y funciones de activación. Este enfoque experimental permite encontrar la configuración óptima que maximice el rendimiento del modelo y produzca resultados óptimos.

La capacidad de la retropropagación para ajustar los pesos de la red neuronal en función de los errores de predicción es fundamental para su éxito en la clasificación de datos. Su capacidad para aprender de los errores y mejorar la precisión del modelo lo posiciona como una herramienta poderosa en el arsenal del aprendizaje automático. En resumen, concluimos que la retropropagación es una técnica valiosa y versátil que puede desempeñar un papel fundamental en una variedad de aplicaciones de clasificación de datos.