import numpy as np: Importa la biblioteca NumPy para realizar operaciones matemáticas y algebraicas en matrices y arreglos. Se usa la abreviatura np para referirse a NumPy en el código.

import pandas as pd: Importa la biblioteca pandas, que se utiliza para la manipulación y análisis de datos en forma de marcos de datos (DataFrames). Se utiliza la abreviatura pd para referirse a pandas en el código.

from sklearn.preprocessing import OneHotEncoder: Importa la clase OneHotEncoder del módulo sklearn.preprocessing. Se usa para realizar la codificación one-hot de variables categóricas.

import tqdm: Importa la biblioteca tqdm, que proporciona una barra de progreso visual para el seguimiento del progreso de bucles y operaciones largas.

```
import numpy as np #Linear algebra and mathematical operations
import pandas as pd #importing and loading data
from sklearn.preprocessing import OneHotEncoder
import tqdm
```

iris_df = pd.read_csv("Iris.csv"):

pd.read_csv("Iris.csv") es una función de la biblioteca pandas (pd es una abreviatura comúnmente utilizada para pandas) que se utiliza para cargar datos desde un archivo CSV. En este caso, se está cargando un archivo llamado "Iris.csv". El resultado de esta operación es un DataFrame de pandas llamado iris_df, que es una estructura de datos que permite manipular y analizar tablas de datos de manera eficiente.
iris_df = iris_df.sample(frac=1).reset_index(drop=True):

iris_df.sample(frac=1) realiza una operación de muestreo en el DataFrame iris_df. El argumento frac=1 indica que se desea tomar una fracción del DataFrame igual al 100% del tamaño original, lo que es equivalente a reorganizar aleatoriamente todas las filas del DataFrame.

reset_index(drop=True) se utiliza para restablecer el índice del DataFrame después de la operación de muestreo. El argumento drop=True indica que el índice anterior se descartará y no se añadirá como una nueva columna en el DataFrame.

```
iris_df = pd.read_csv("Iris.csv")
iris_df = iris_df.sample(frac=1).reset_index(drop=True) # Shuffle
```

```
iris_df.head()
```

	Id	SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm	Species
0	71	5.9	3.2	4.8	1.8	Iris-versicolor
1	65	5.6	2.9	3.6	1.3	Iris-versicolor
2	81	5.5	2.4	3.8	1.1	Iris-versicolor
3	34	5.5	4.2	1.4	0.2	Iris-setosa
4	68	5.8	2.7	4.1	1.0	Iris-versicolor

X = iris_df[['SepalLengthCm', 'SepalWidthCm', 'PetalLengthCm', 'PetalWidthCm']]:

En esta línea, se crea un nuevo DataFrame X a partir de iris_df, pero solo seleccionando cuatro columnas específicas: 'SepalLengthCm' (Longitud del Sépalo), 'SepalWidthCm' (Ancho del Sépalo), 'PetalLengthCm' (Longitud del Pétalo) y 'PetalWidthCm' (Ancho del Pétalo). El resultado es un DataFrame nuevo que contiene solo estas cuatro características del conjunto de datos de Iris. X = np.array(X):

Luego, este nuevo DataFrame X se convierte en un arreglo NumPy utilizando np.array(X). Esto es útil porque muchos algoritmos de aprendizaje automático y procesamiento numérico en Python trabajan mejor con arreglos NumPy en lugar de DataFrames de pandas. X[:5]:

Por último, se muestra las primeras 5 filas del arreglo X. Esta operación proporciona una vista previa de las primeras 5 filas del conjunto de datos que contiene solo las cuatro características seleccionadas ('SepalLengthCm', 'SepalWidthCm', 'PetalLengthCm' y 'PetalWidthCm').

```
X = iris_df[['SepalLengthCm', 'SepalWidthCm', 'PetalLengthCm', 'PetalWidthCm']]
X = np.array(X)
X[:5]
```

```
array([[5.9, 3.2, 4.8, 1.8],
       [5.6, 2.9, 3.6, 1.3],
       [5.5, 2.4, 3.8, 1.1],
```

```
       [5.5, 4.2, 1.4, 0.2],
       [5.8, 2.7, 4.1, 1. ]])
```

one_hot_encoder = OneHotEncoder(sparse_output=False):

Se crea una instancia de la clase OneHotEncoder del módulo sklearn.preprocessing. Esta clase se utiliza para realizar la codificación one-hot en variables categóricas. El argumento sparse_output=False se utiliza para indicar que no se desea una matriz dispersa como resultado. En su lugar, se prefiere una matriz densa. Y = iris_df.Species:

Se crea una nueva variable Y que contiene la columna "Species" del DataFrame iris_df. Esta columna contiene las etiquetas de las especies de flores en el conjunto de datos Iris. Y = one_hot_encoder.fit_transform(np.array(Y).reshape(-1, 1)):

Primero, se convierte la variable Y en un arreglo NumPy unidimensional utilizando np.array(Y).reshape(-1, 1). Esto es necesario porque fit_transform espera un arreglo bidimensional como entrada. Luego, se utiliza one_hot_encoder.fit_transform para realizar la codificación one-hot en el arreglo unidimensional resultante. Esto crea una representación binaria de las etiquetas de especies, donde cada fila de la matriz resultante corresponde a una etiqueta y cada columna representa una categoría única de especie. Cada celda contiene un 1 si la etiqueta pertenece a esa categoría y un 0 en caso contrario. Y[:5]:

Por último, se muestra las primeras 5 filas de la variable Y. Esto proporciona una vista previa de cómo se ha codificado one-hot una parte de las etiquetas de especies en el conjunto de datos.

```
one_hot_encoder = OneHotEncoder(sparse_output=False)
Y = iris_df.Species
Y = one_hot_encoder.fit_transform(np.array(Y).reshape(-1, 1))
Y[:5]
```

```
array([[0., 1., 0.],
       [0., 1., 0.],
       [0., 1., 0.],
       [1., 0., 0.],
       [0., 1., 0.]])
```

Este código implementa una red neuronal simple en Python utilizando NumPy y define varias funciones útiles relacionadas con la red neuronal y la evaluación de su rendimiento.

1. Importaciones:
   - `import numpy as np`: Importa la biblioteca NumPy para realizar operaciones numéricas y algebraicas eficientes.
2. Funciones de activación y pérdida:
   - `sigmoid(x)`: Define una función de activación sigmoide que toma un valor real `x` y devuelve el resultado de aplicar la función sigmoide a `x`.
   - `sigmoid_derivative(x)`: Define la derivada de la función sigmoide en función de `x`.
   - `mse_loss(y_true, y_pred)`: Define la función de pérdida de error cuadrático medio (MSE) que toma las etiquetas verdaderas `y_true` y las etiquetas predichas `y_pred` y calcula el error cuadrático medio entre ellas.
   - `root_mean_sqrt(y_true, y_pred)`: Calcula la raíz cuadrada del error cuadrático medio (RMSE) entre las etiquetas verdaderas y predichas.
   - `r_squared(y_true, y_pred)`: Calcula el coeficiente de determinación (R^2) entre las etiquetas verdaderas y predichas.
   - `mean_absolute_percentage_error(y_true, y_pred)`: Calcula el error porcentual absoluto medio (MAPE) entre las etiquetas verdaderas y predichas.
3. Clase `NeuralNetwork`:
   - Define una clase llamada `NeuralNetwork` que representa una red neuronal de alimentación hacia adelante (feedforward).
   - `__init__(self, input_dim, output_dim, nodes, learning_rate)`: El constructor de la clase inicializa los parámetros de la red neuronal, incluyendo el número de nodos en cada capa oculta (`nodes`), la tasa de aprendizaje (`learning_rate`), los pesos y sesgos de las capas.
   - `forward(self, x)`: Realiza la propagación hacia adelante en la red neuronal. Calcula las salidas de cada capa y las activaciones utilizando la función sigmoide.
   - `backward(self, x, y_true)`: Realiza la propagación hacia atrás en la red neuronal para actualizar los pesos y sesgos utilizando el algoritmo de retropropagación.

- `train(self, X_train, Y_train, epochs)` : Entrena la red neuronal utilizando los datos de entrenamiento `X_train` e `Y_train` durante el número especificado de épocas (`epochs`). Muestra la pérdida en cada época.
- `predict(self, X_test, y_test)` : Esta función está incompleta y necesita implementarse para realizar predicciones con la red neuronal en un conjunto de datos de prueba.
- `get_stats(self)` : Calcula y muestra estadísticas de evaluación, como RMSE, R^2 y MAPE, para la red neuronal entrenada.

```
import numpy as np

# Activation function (sigmoid)
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# Derivative of sigmoid
def sigmoid_derivative(x):
    return x * (1 - x)

# Mean squared error loss
def mse_loss(y_true, y_pred):
    return np.mean((y_true - y_pred)**2)

def root_mean_sqrt(y_true, y_pred):
    return np.sqrt(np.mean((y_true - y_pred)**2))

def r_squared(y_true, y_pred):
    mean_y_true = np.mean(y_true)
    ss_total = np.sum((y_true - mean_y_true) ** 2)
    ss_residual = np.sum((y_true - y_pred) ** 2)
    return 1 - (ss_residual / ss_total)

def mean_absolute_percentage_error(y_true, y_pred):
    epsilon = 1
    return np.mean(np.abs((y_true - y_pred) / (y_true + epsilon))) * 100


# Neural network class
class NeuralNetwork:
    def __init__(self, input_dim, output_dim, nodes, learning_rate):
        self.weights = []
        self.biases = []
        self.layers = len(nodes)
        self.learning_rate = learning_rate

        prev_nodes = input_dim
        for num_nodes in nodes:
            self.weights.append(np.random.rand(prev_nodes, num_nodes))
            self.biases.append(np.zeros((1, num_nodes)))
            prev_nodes = num_nodes

        self.weights.append(np.random.rand(prev_nodes, output_dim))
        self.biases.append(np.zeros((1, output_dim)))
        self.is_trained = False

    def forward(self, x):
        self.layer_outputs = []
        self.activations = [x]

        for i in range(self.layers + 1):
            output = np.dot(self.activations[-1], self.weights[i]) + self.biases[i]
            self.layer_outputs.append(output)
            activation = sigmoid(output)
            self.activations.append(activation)

        return self.activations[-1]

    def backward(self, x, y_true):
        output_error = y_true - self.activations[-1]
        for i in range(self.layers, -1, -1):
            delta = output_error * sigmoid_derivative(self.activations[i + 1])
            self.weights[i] += self.learning_rate * np.dot(self.activations[i].T, delta)
            self.biases[i] += self.learning_rate * delta
            output_error = np.dot(delta, self.weights[i].T)

    def train(self, X_train, Y_train, epochs):
        for epoch in trange(epochs):
```

```
            for x, y_true in zip(X_train, Y_train):
                x = np.array(x).reshape(1, -1)
                y_true = np.array(y_true).reshape(1, -1)
                y_pred = self.forward(x)
                self.backward(x, y_true)

            if epoch % (epochs/5) == 0:
                loss = mse_loss(y_true, y_pred)
                print(f"Epoch {epoch}, Loss: {loss:.4f}")
        self.is_trained = True
        self.mse = loss


    def predict(self,X_test,y_test):


    def get_stats(self):
        if self.is_trained:
            self.rmse = np.sqrt(self.mse)
            self.r_squared = r_squared(self.y_true, self.y_pred)
            self.mape = mean_absolute_percentage_error(self.y_true, self.y_pred)
            print(f"""
            RMSE = {self.rmse:.2f}
            R_SQUARED = {self.r_squared:.2f}
            MAPE = {self.mape:.2f}
            """)
        else:
            raise Error('HDP entrene el modelo primero')


# Define your data and parameters
from sklearn.model_selection import train_test_split
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.15)
X_train, X_val, Y_train, Y_val = train_test_split(X_train, Y_train, test_size=0.1)

f = len(X_train[0])  # Number of features
o = len(Y_train[0])  # Number of classes
layers = [5, 10]     # Number of nodes in hidden layers
L, E = 0.15, 10000     # Learning rate and epochs

# Create an instance of the NeuralNetwork class
model = NeuralNetwork(input_dim=f, output_dim=o, nodes=layers, learning_rate=L)

# Train the model
model.train(X_train, Y_train, epochs=E)
```

```
  0%|          | 0/10000 [00:00<?, ?it/s]
Epoch 0, Loss: 0.8576
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
Cell In[55], line 15
     12 model = NeuralNetwork(input_dim=f, output_dim=o, nodes=layers, learning_rate=L)
     14 # Train the model
---> 15 model.train(X_train, Y_train, epochs=E)

Cell In[49], line 73, in NeuralNetwork.train(self, X_train, Y_train, epochs)
     71     y_true = np.array(y_true).reshape(1, -1)
     72     y_pred = self.forward(x)
---> 73     self.backward(x, y_true)
     75 if epoch % (epochs/5) == 0:
     76     loss = mse_loss(y_true, y_pred)

Cell In[49], line 62, in NeuralNetwork.backward(self, x, y_true)
     60 output_error = y_true - self.activations[-1]
     61 for i in range(self.layers, -1, -1):
---> 62     delta = output_error * sigmoid_derivative(self.activations[i + 1])
     63     self.weights[i] += self.learning_rate * np.dot(self.activations[i].T, delta)
     64     self.biases[i] += self.learning_rate * delta

KeyboardInterrupt:
```

SEARCH STACK OVERFLOW

El código model.get_stats() invoca un método llamado get_stats en una instancia de la clase NeuralNetwork llamada model. Aquí está la explicación de lo que hace este código:

La clase NeuralNetwork que has definido anteriormente incluye un método llamado get_stats, y este método se utiliza para calcular y mostrar estadísticas de evaluación de un modelo de red neuronal previamente entrenado. Aquí está el análisis de la función get_stats

```
model.get_stats()
```

```
RMSE = 0.01
R_SQUARED = 1.00
MAPE = 0.88
```

Análisis de las métricas

RMSE (Root Mean Squared Error): 0.01

RMSE es una medida de cuán cerca están las predicciones del modelo de las etiquetas verdaderas. Un valor de RMSE cercano a cero indica que el modelo tiene un buen ajuste a los datos. En tu caso, un RMSE de 0.01 es extremadamente bajo, lo que sugiere que las predicciones de tu modelo son muy precisas en términos de la diferencia entre las etiquetas verdaderas y las predicciones. R_SQUARED (R^2): 1.00

R^2 es una medida de la calidad del ajuste del modelo a los datos. Un valor de R^2 de 1.00 indica que el modelo es capaz de explicar el 100% de la variabilidad en los datos de destino. Es decir, el modelo se ajusta perfectamente a los datos. Esto es inusualmente alto y puede ser una señal de sobreajuste si no se ha tenido cuidado al evaluar el modelo. MAPE (Mean Absolute Percentage Error): 0.88

MAPE es una medida del error porcentual promedio absoluto entre las etiquetas verdaderas y las predicciones. Un MAPE de 0.88% significa que, en promedio, las predicciones del modelo tienen un error absoluto del 0.88% en relación con las etiquetas verdaderas. Este valor también es muy bajo y sugiere un buen rendimiento del modelo.

El código model.weights se refiere a un atributo de la instancia de la clase NeuralNetwork llamada model. Este atributo almacena la información sobre los pesos de la red neuronal, es decir, las conexiones entre las neuronas en cada capa de la red.

```
model.weights
```

```
[array([[ 0.0560017 ,  0.31997349, -0.00853139,  0.64103191,  0.16413511],
        [ 0.46903177,  0.23855972,  0.83450365,  0.52179303,  0.68674812],
        [ 0.84105287,  0.74256463,  0.38064473,  0.1748019 ,  0.83342249],
        [ 0.34294358,  0.3981851 ,  0.79359027,  0.11354331,  0.61428416]]),
 array([[1.01283189, 0.80379509, 0.41306963, 0.60659495, 0.38859247,
         0.85359296, 0.71365684, 0.90625517, 0.83221859, 0.88026309],
        [0.77504263, 1.01507917, 0.95167981, 0.48894364, 0.39038631,
         0.58099191, 1.02192837, 0.93636145, 1.01070893, 0.85669174],
        [0.17722826, 0.81165274, 0.56693644, 0.50784662, 0.99042984,
         1.01449569, 0.17685743, 0.42190074, 0.75901571, 0.28710357],
        [0.21390644, 0.9050261 , 0.36599057, 1.02316665, 0.85705284,
         0.59864763, 1.09913984, 1.01235767, 0.33773498, 0.46862377],
        [0.96369666, 0.77331556, 0.70765131, 0.3521905 , 0.69854701,
         0.21262012, 0.6239272 , 0.68700229, 0.30117476, 0.47004483]]),
 array([[-0.24151589,  0.17332751,  0.26591829],
        [ 0.63342267, -0.31024856,  0.06010302],
        [ 0.19536911, -0.12227955,  0.58953861],
        [-0.24593462,  0.26384077, -0.39075717],
        [-0.29117716,  0.22895466, -0.01368985],
        [-0.04865138, -0.48560328,  0.42561441],
        [-0.30922759,  0.05732696, -0.31225031],
        [-0.27837348, -0.4745011 ,  0.01593488],
        [-0.02063119, -0.15062688,  0.17927026],
        [-0.04767289,  0.19082468, -0.26852311]])]
```