Practica 7

Red neuronal multicapa con regresión

Seminario de solución de problemas de inteligencia artificial 2

Clave del curso: I7041

NRC: 124882

Calendario: 2023-A

Sección: D01

Martínez Sepúlveda Alan Jahir

216569127

INCO

02/05/2023

Introducción:

Para la realización de esta práctica, se retoma ya la hecha practica 5 para poder tener las bases de la red neuronal multicapa, pero agregándole a esta la capacidad de que se pueda hacer regresión.

Desarrollo:

Se crea un archivo para definir las funciones necesarias para la red neuronal.

La función "linear" simplemente devuelve la entrada sin ninguna transformación y su derivada es 1.

La función "tanh" devuelve la tangente hiperbólica de la entrada y su derivada es 1 menos el cuadrado de la tangente hiperbólica.

La función "sigmoid" devuelve la función sigmoide de la entrada y su derivada es la función sigmoide multiplicada por uno menos la función sigmoide.

La función "relu" devuelve la función de activación de unidades lineales rectificadas (ReLU) de la entrada y su derivada es 1 para valores positivos y 0 para valores negativos.

La función "activate" toma una cadena que especifica el nombre de la función de activación y devuelve la función correspondiente. Si se proporciona un nombre de función desconocido, se genera una excepción.

```python
import numpy as np

def linear(z, derivative = False):
    a = z
    if derivative:
        da = np.ones(z.shape)
        return a, da
    return a

def tanh(z, derivative = False):
    a = np.tanh(z)
    if derivative:
        da = (1 + a) * (1 - a)
        return a, da
    return a

def sigmoid(z, derivative = False):
    a = 1 / (1 + np.exp(-z))
    if derivative:
        da = a * (1 - a)
        return a, da
    return a

def relu(z, derivative = False):
    a = z * (z >= 0)
    if derivative:
        da = np.array(z >= 0 , dtype=float)
        return a, da
    return a

def activate(function_name):
    if function_name == 'linear':
        return linear
    elif function_name == 'tanh':
        return tanh
    elif function_name == 'sigmoid':
        return sigmoid
    elif function_name == 'relu':
        return relu
    else:
        raise ValueError('function_name unknown')
```

Se define la clase MLP que implementa una red neuronal multicapa para la clasificación y regresión, así como el método constructor de la clase MLP. Recibe los argumentos layers_dim y activations. El primero es una lista de enteros que indica la dimensión de cada capa de la red neuronal, y el segundo es una lista de strings que indica qué función de activación utilizar para cada capa. La lista de activations debe tener la misma longitud que layers_dim menos uno, ya que la primera capa no necesita una función de activación.

```python
class MLP:
    def __init__(self, layers_dim, activations):
        #Atributes
        self.W = [None]
        self.b = [None]
        self.f = [None]
        self.n = layers_dim
        self.L = len(layers_dim) - 1

        #Initialization of sinaptic weights and bias.
        for l in range(1, self.L + 1):
            self.W.append(-1 + 2 * np.random.rand(self.n[l], self.n[l-1]))
            self.b.append(-1 + 2 * np.random.rand(self.n[l], 1))

        #Fill activation functions list
        for act in activations:
            self.f.append(activate(act))
```

El método predict realiza una predicción con la red neuronal. Recibe una matriz de entrada X de forma (n[0], m) y devuelve una matriz de salida a de forma (n[L], m), donde m es la cantidad de muestras.

```python
def predict(self, X):
    a = np.asanyarray(X)
    for l in range(1, self.L + 1):
        z = np.dot(self.W[l], a) + self.b[l]
        a = self.f[l](z)
    return a
```

El método de train implementa el algoritmo de descenso de gradiente estocástico para entrenar una red neuronal de múltiples capas.

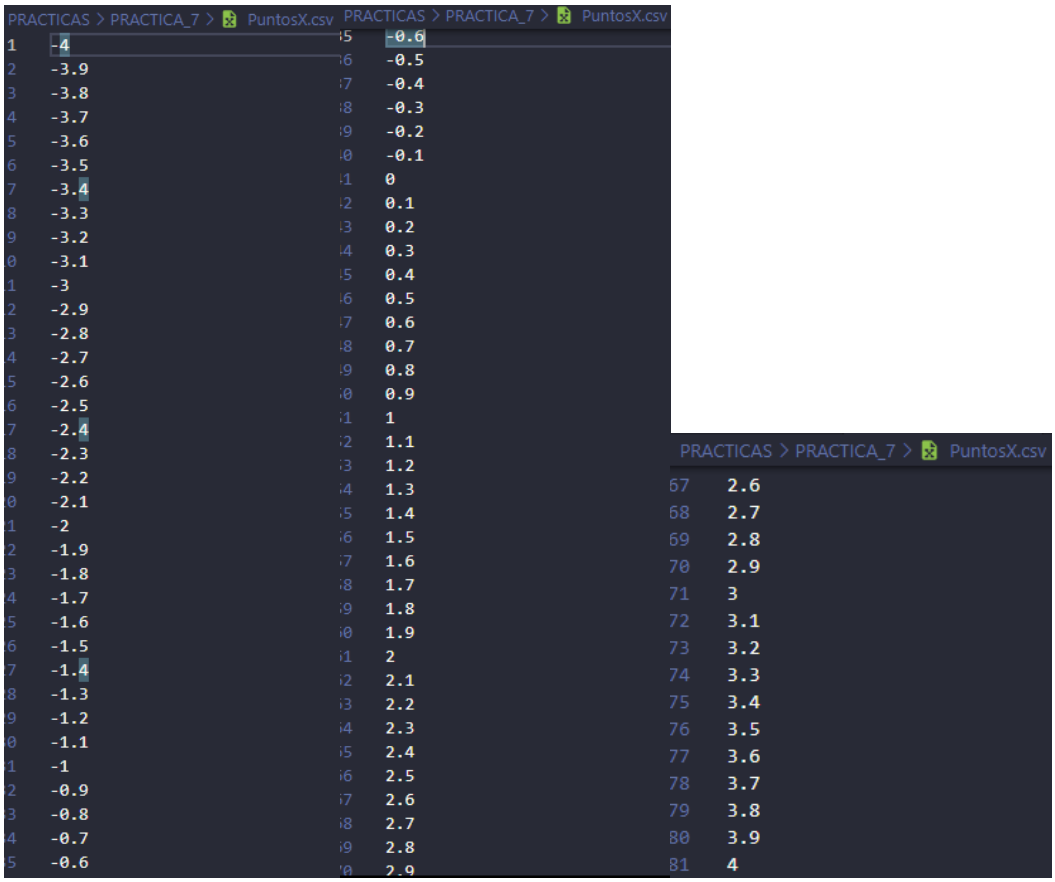
Recibe como entrada un conjunto de datos de entrada X y un conjunto de etiquetas Y correspondientes, el número de épocas de entrenamiento epochs, y la tasa de aprendizaje learning_rate.

El método recorre epochs veces los datos de entrada y etiquetas y en cada iteración actualiza los pesos de la red de acuerdo con el error de la predicción en esa iteración. Para cada par de entrada y etiqueta, realiza la propagación hacia adelante de la red neuronal para obtener las salidas predichas y luego calcula el error entre las salidas predichas y las etiquetas. Luego, realiza la propagación hacia atrás del error para actualizar los pesos y los sesgos de cada capa de la red. Esto se hace mediante el cálculo de los gradientes locales y la aplicación del descenso de gradiente estocástico para actualizar los pesos y sesgos.

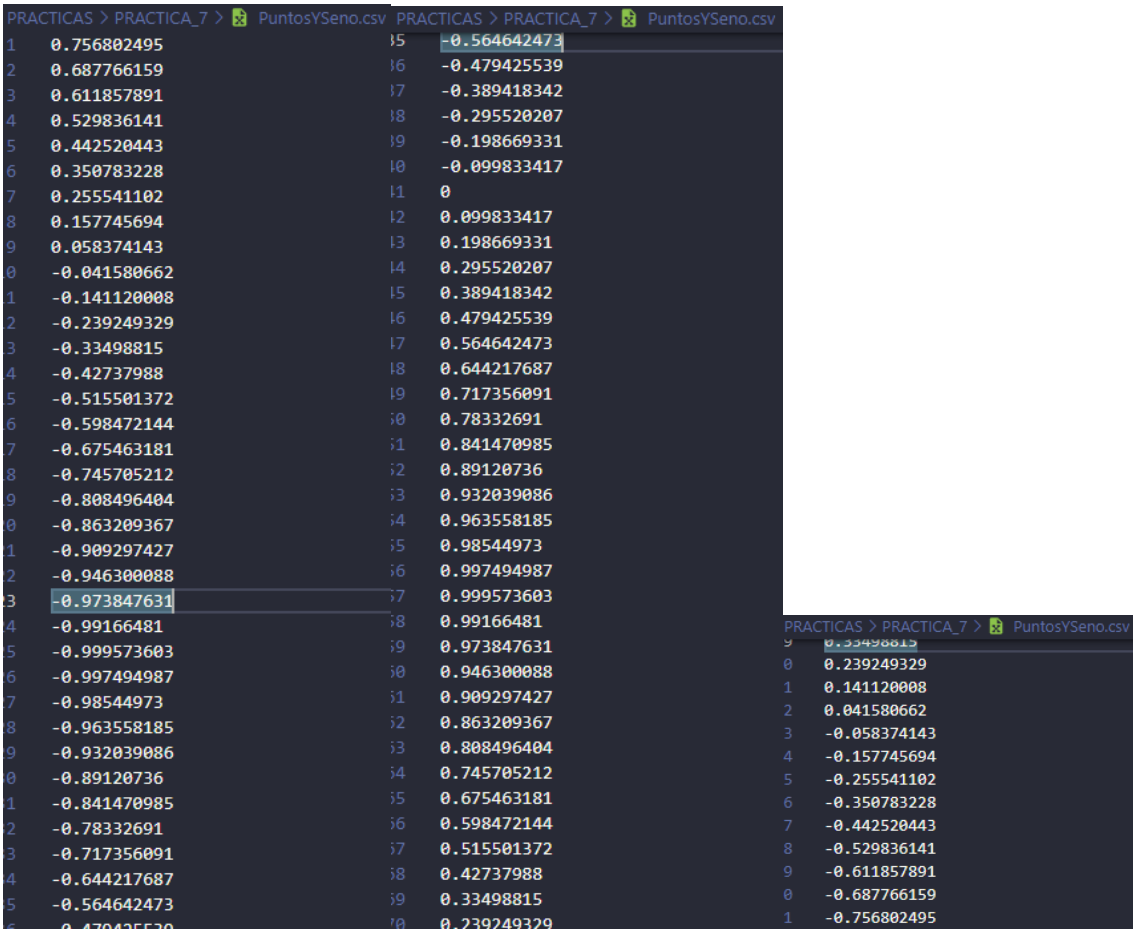
Finalmente, el método devuelve el error de la predicción y la predicción en sí, es decir, las salidas predichas por la red neuronal para el conjunto de entrada X.

```
def train(self, X, Y, epochs, learning_rate):
    X = np.asanyarray(X)
    Y = np.asanyarray(Y).reshape(self.n[-1], -1)
    P = X.shape[1]
    error = 0

    for _ in range(epochs):
        #Stocastic Gradient Descend
        for p in range(P):
            A = [None] * (self.L + 1)
            dA = [None] * (self.L + 1)
            lg = [None] * (self.L + 1)

            #Propagation
            A[0] = X[:,p].reshape(self.n[0], 1)
            for l in range(1, self.L + 1):
                z = np.dot(self.W[l], A[l-1]) + self.b[l]
                A[l], dA[l] = self.f[l](z, derivative=True)

            #regresion
            for l in range(self.L, 0, -1):
                if l == self.L:
                    #lg = Local Gradient
                    lg[l] = (Y[:, p] - A[l]) * dA[l]
                else:
                    lg[l] = np.dot(self.W[l+1].T, lg[l+1]) * dA[l]

            #Weights updates
            for l in range(1, self.L + 1):
                self.W[l] += learning_rate * np.dot(lg[l], A[l-1].T)
                self.b[l] += learning_rate * lg[l]

    predictions = self.predict(X)
    for i in range(len(predictions[0])):
        error += abs((Y[0][i] - predictions[0][i]))
    error /= len(Y[0])
    return error, predictions
```

Para graficar los datos, así como hacer la regresión de la misma se decidió hacer la función de seno, la cual se se grafican desde los puntos -4 a 4 en un archivo csv:



Después se definen los puntos en Y de la función del seno, la cual será nuestra salida deseada:



A continuación se definen las entradas que se le darán en un inicio a la red neuronal, cabe destacar que los datos son menor al 80% de los necesarios para poder resolverse, es por eso que con la regresión se resolverán los faltantes:

PRACTICAS > PRACTICA_7 > Entradas.csv

```
-4
-3.7
-3.4
-3.1
-2.8
-2.5
-2.2
-1.9
-1.6
-1.3
-1
-0.7
-0.4
-0.1
0.2
0.5
0.8
1.1
1.4
1.7
2
2.3
2.6
2.9
3.2
3.5
3.8
4.1
```

Se establecen los hiperparámetros de la red neuronal, como el número de épocas, la tasa de aprendizaje, el número de neuronas en la capa oculta y el número de neuronas en la capa de salida. Se crea una instancia de la clase MLP, que es la red neuronal.

```
epochs = 1500
learning_rate = 0.1
entries = 1  # of columns for the trainingPatternsFileName.
neurons_in_hidden_layer = 8
output_layer_neurons = 1

net = MLP((entries, neurons_in_hidden_layer, output_layer_neurons), ('tanh', 'linear'))

X = []
y = []

x_func = []
y_func = []

X.append(np.array(np.loadtxt(trainingPatternsFileName, delimiter=',', usecols=0)))
y.append(np.array(np.loadtxt(outputValuesFileName, delimiter=',', usecols=0)))

x_func.append(np.array(np.loadtxt(x_funcFile, delimiter=',', usecols=0)))
y_func.append(np.array(np.loadtxt(y_funcFile, delimiter=',', usecols=0)))

# Entrenamiento.

plt.axhline(y=0, color='k')
plt.axvline(x=0, color='k')
plt.xlim([-5, 5])
plt.ylim([-5, 10])

error_list = []
```

Después de definir los parámetros y crear la instancia de la red neuronal, se cargan los datos de entrenamiento y se almacenan en listas. Se utiliza la función train() de la clase MLP para entrenar la red neuronal. Mientras se entrena la red, se almacenan los errores de cada época en una lista y se grafican los resultados de la red neuronal en cada décima época.

```
for i in range(epochs):
    error, pred = net.train(X, y, 1, learning_rate)
    error_list.append(error)
    print("Epoch:", i, "Error:", error)

    if i%10 == 0:
        plt.clf()
        plt.scatter(x_func, y_func, s=40, c='#0404B4')
        plt.plot(X[0], pred[0], color='green', linewidth=3)
        plt.show()
        plt.pause(0.2)
        plt.close()

        # if error < 0.03:
        if error < 0.015: #Para tanh y log.
            break

plt.figure(2)
plt.plot(error_list, color='red', linewidth=3)
plt.pause(0.2)
plt.close()
```

Finalmente, se guardan los resultados de la red neuronal en un archivo csv y se cierra la figura.

```
    results = np.array(net.predict(X)).T
    np.savetxt("Results.csv", results, delimiter=",", fmt='%.4f')

if __name__ == "__main__":
    main()
```

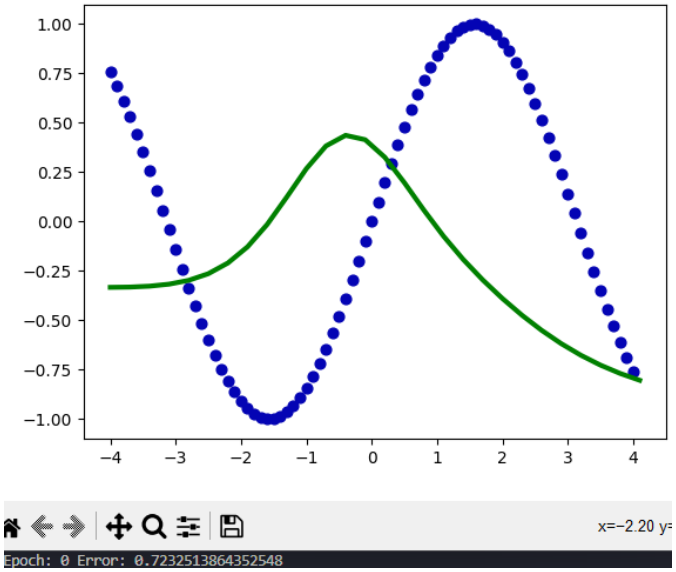
Conclusión:

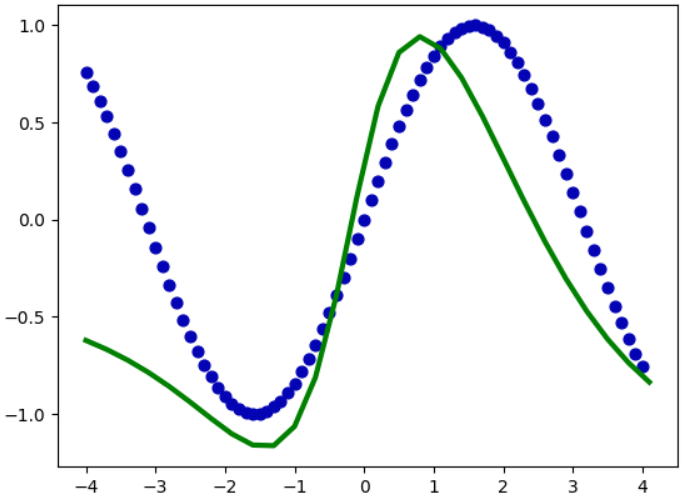
En conclusión, los archivos dados previamente son parte de un proyecto que implementa una red neuronal artificial para realizar regresión. El archivo "MPL.py" contiene la implementación de una red neuronal multicapa con backpropagation, mientras que el archivo "main.py" contiene el código que utiliza la red neuronal para realizar la regresión.

Se cargan los datos de entrenamiento y los datos de gráfica, inicializa una instancia de la red neuronal con una capa oculta y una capa de salida, entrena la red neuronal con los datos de entrenamiento, grafica la salida de la red neuronal en cada epoch, guarda los resultados de la predicción y finalmente grafica el error de entrenamiento a lo largo de las epochs.

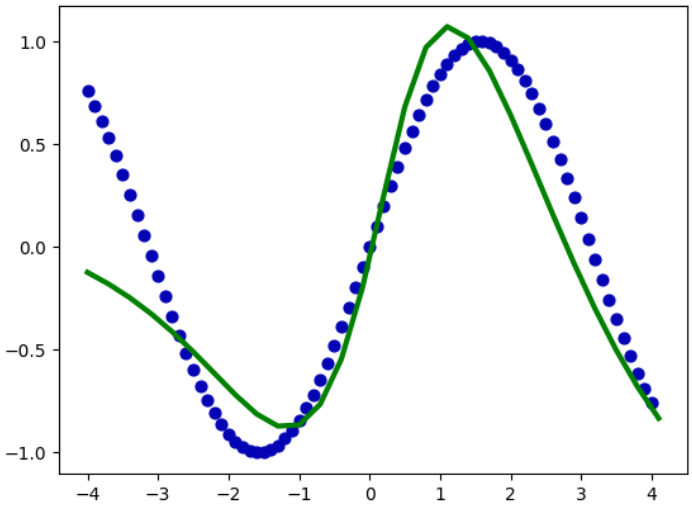
Resultados:

```
klenb@AlanMtz200 MINGW64 /u/8º SEMESTRE/SEMINARIO IA 2/practicas/PRACTICA_7 (master)
$ python Main.py
----------SENO CON REGRESION----------
[!]:
```

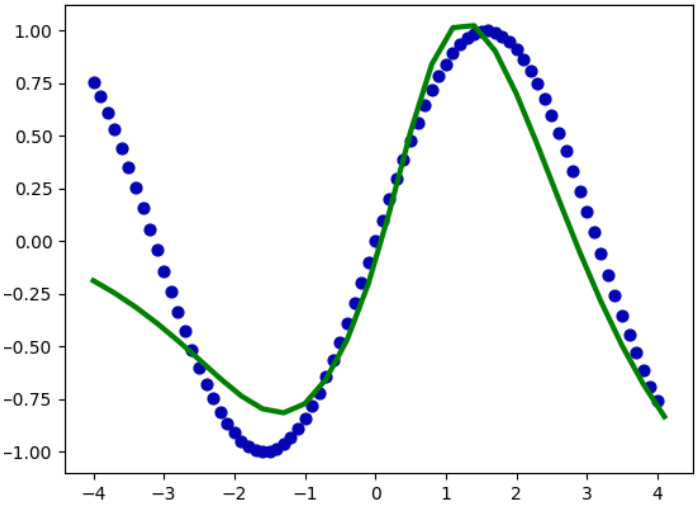




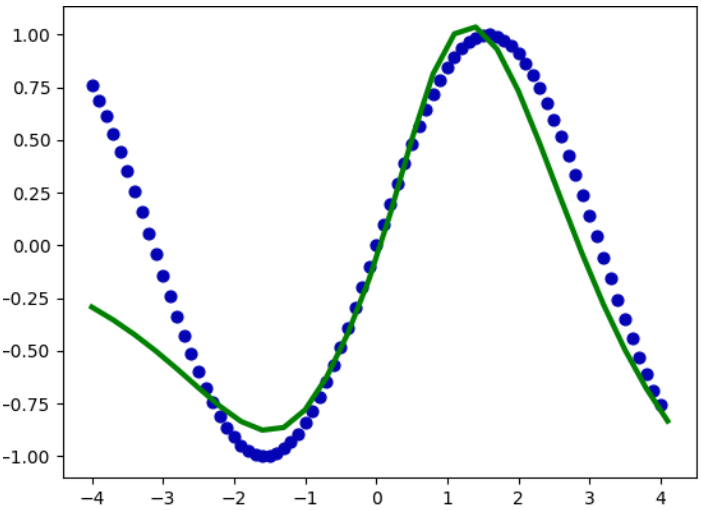

```
Epoch: 0 Error: 0.7232513864352548
Epoch: 1 Error: 0.6507283432564058
Epoch: 2 Error: 0.60820355630489
Epoch: 3 Error: 0.5932719754558934
Epoch: 4 Error: 0.5713205449696525
Epoch: 5 Error: 0.5509501861146547
Epoch: 6 Error: 0.5245568820526233
Epoch: 7 Error: 0.5015961867093767
Epoch: 8 Error: 0.474317611095528
Epoch: 9 Error: 0.4429709086825219
Epoch: 10 Error: 0.4089486299721391
```



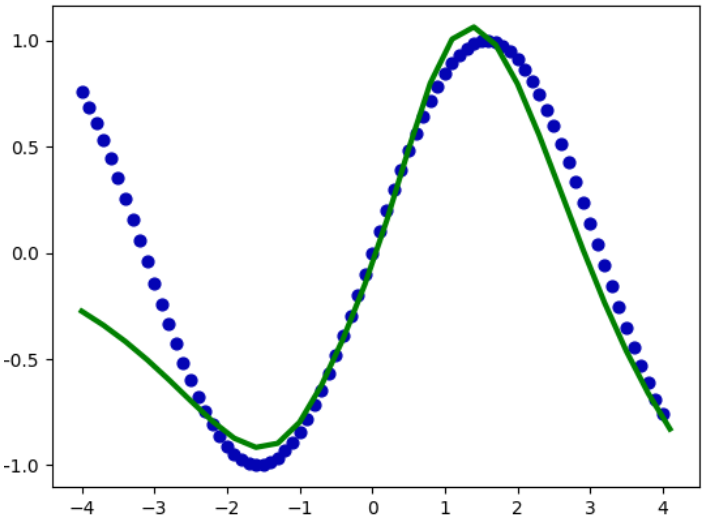

```
Epoch: 11 Error: 0.3787754194174611
Epoch: 12 Error: 0.3480989330847124
Epoch: 13 Error: 0.3172418783705772
Epoch: 14 Error: 0.29026219789254454
Epoch: 15 Error: 0.27124411355791117
Epoch: 16 Error: 0.2549151653723989
Epoch: 17 Error: 0.24373915239116525
Epoch: 18 Error: 0.23598328486830303
Epoch: 19 Error: 0.2292697139757043
Epoch: 20 Error: 0.2232667135085009
```



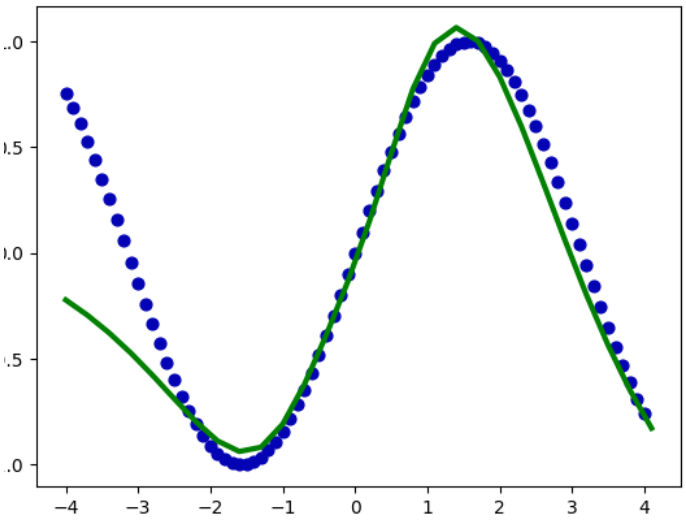

```
Epoch: 20 Error: 0.2232667135085009
Epoch: 21 Error: 0.21800184789197077
Epoch: 22 Error: 0.21409002319666634
Epoch: 23 Error: 0.2120000863512197
Epoch: 24 Error: 0.2106781910869786
Epoch: 25 Error: 0.20966556652961552
Epoch: 26 Error: 0.20893134954362233
Epoch: 27 Error: 0.20843581443831405
Epoch: 28 Error: 0.20813105687984987
Epoch: 29 Error: 0.207962233687515
Epoch: 30 Error: 0.20786957373551068
```



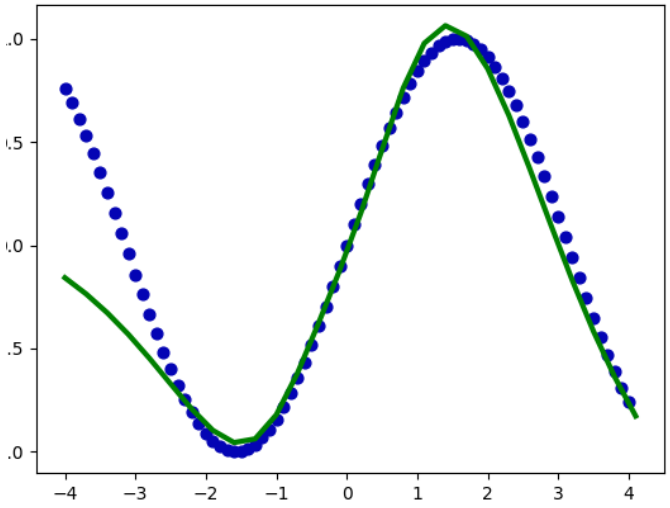

```
Epoch: 30 Error: 0.20786957373551068
Epoch: 31 Error: 0.2077912049492448
Epoch: 32 Error: 0.20772629521871738
Epoch: 33 Error: 0.20816697640454804
Epoch: 34 Error: 0.2089994898186905
Epoch: 35 Error: 0.20953224534122178
Epoch: 36 Error: 0.2097333948567776
Epoch: 37 Error: 0.20958713083060312
Epoch: 38 Error: 0.20909256844114057
Epoch: 39 Error: 0.2082617972164321
Epoch: 40 Error: 0.20711742372225778
```



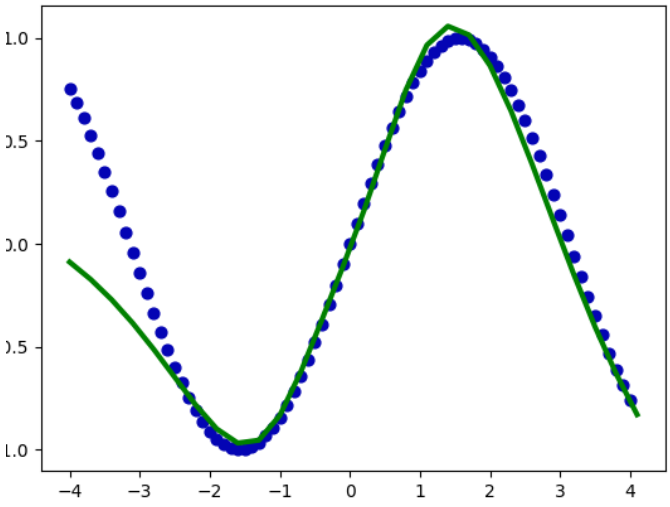

```
Epoch: 40 Error: 0.20711742372225778
Epoch: 41 Error: 0.20568989767245632
Epoch: 42 Error: 0.20401485523749402
Epoch: 43 Error: 0.20213064890222252
Epoch: 44 Error: 0.2000761757937176
Epoch: 45 Error: 0.1978890702294293
Epoch: 46 Error: 0.19560429072976815
Epoch: 47 Error: 0.19325310452776975
Epoch: 48 Error: 0.19086245138879246
Epoch: 49 Error: 0.18845465185713992
Epoch: 50 Error: 0.18604741231639224
```



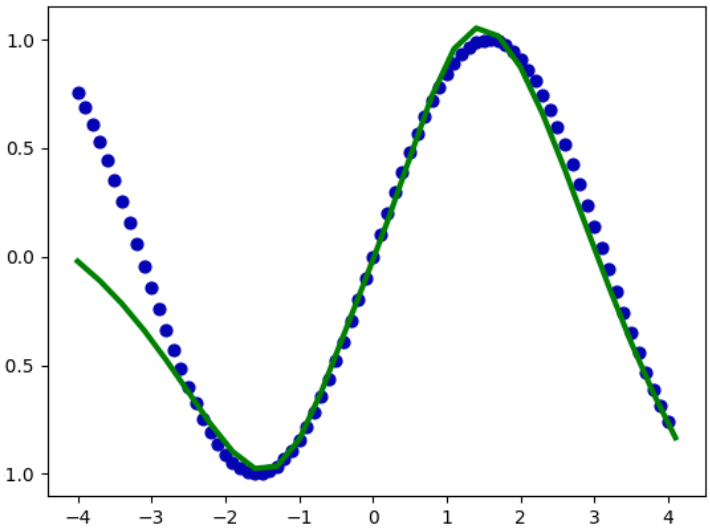

```
Epoch: 50 Error: 0.18604741231639224
Epoch: 51 Error: 0.18365407054942856
Epoch: 52 Error: 0.18128402107712777
Epoch: 53 Error: 0.17894325952606135
Epoch: 54 Error: 0.17666746189536858
Epoch: 55 Error: 0.17462925718517885
Epoch: 56 Error: 0.17264919198016365
Epoch: 57 Error: 0.17067999618930593
Epoch: 58 Error: 0.1687210334840095
Epoch: 59 Error: 0.16677146244608657
Epoch: 60 Error: 0.16487291745779303
```

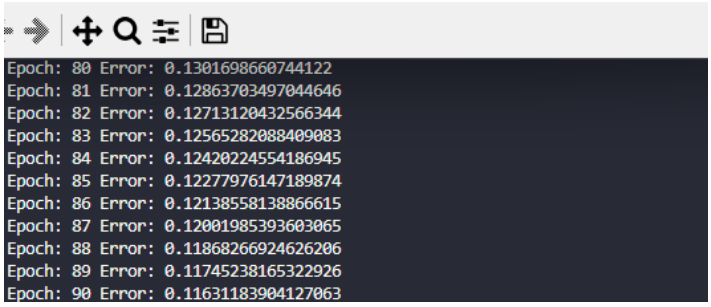


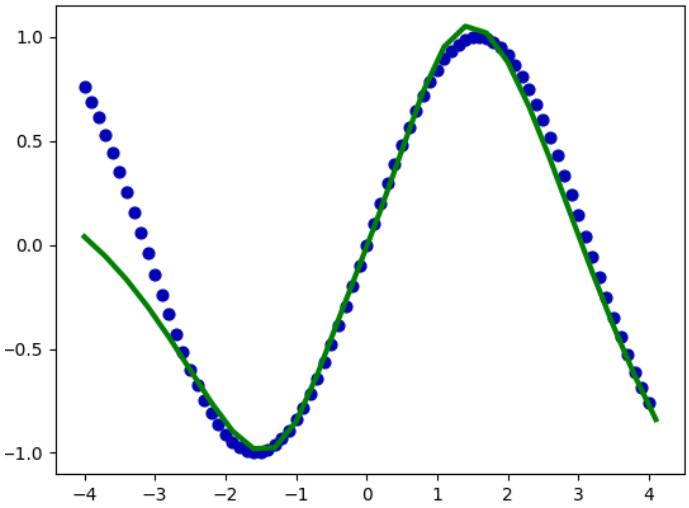
```
Epoch: 60 Error: 0.16487291745779303
Epoch: 61 Error: 0.16304037720346223
Epoch: 62 Error: 0.16121216952749237
Epoch: 63 Error: 0.15938815037053125
Epoch: 64 Error: 0.15756862010235378
Epoch: 65 Error: 0.15575428920207204
Epoch: 66 Error: 0.1539462215797517
Epoch: 67 Error: 0.1521457649382747
Epoch: 68 Error: 0.15035447605385135
Epoch: 69 Error: 0.1485740471928079
Epoch: 70 Error: 0.14680623822035438
```

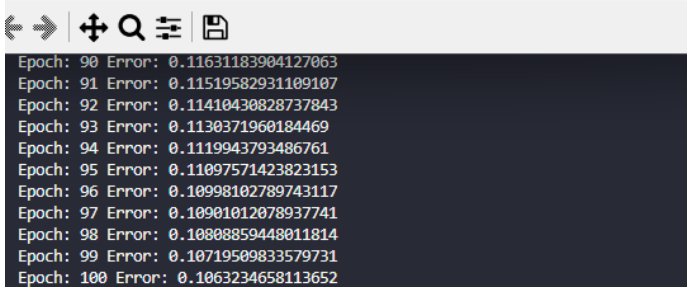


```
Epoch: 70 Error: 0.14680623822035438
Epoch: 71 Error: 0.14505281740367174
Epoch: 72 Error: 0.1433155125449263
Epoch: 73 Error: 0.14159597295073678
Epoch: 74 Error: 0.13989574187307321
Epoch: 75 Error: 0.1382162384430131
Epoch: 76 Error: 0.13655874774084406
Epoch: 77 Error: 0.13492441747059278
Epoch: 78 Error: 0.133314259692798
Epoch: 79 Error: 0.13172915617198394
Epoch: 80 Error: 0.1301698660744122
```


1.0
0.5
0.0
0.5
1.0
−4
−3
−2
−1
0
1
2
3
4


Epoch: 80 Error: 0.1301698660744122
Epoch: 81 Error: 0.12863703497044646
Epoch: 82 Error: 0.12713120432566344
Epoch: 83 Error: 0.12565282088409083
Epoch: 84 Error: 0.12420224554186945
Epoch: 85 Error: 0.12277976147189874
Epoch: 86 Error: 0.12138558138866615
Epoch: 87 Error: 0.12001985393603065
Epoch: 88 Error: 0.11868266924626206
Epoch: 89 Error: 0.11745238165322926
Epoch: 90 Error: 0.11631183904127063


1.0
0.5
0.0
−0.5
−1.0
−4
−3
−2
−1
0
1
2
3
4


Epoch: 90 Error: 0.11631183904127063
Epoch: 91 Error: 0.11519582931109107
Epoch: 92 Error: 0.11410430828737843
Epoch: 93 Error: 0.1130371960184469
Epoch: 94 Error: 0.1119943793486761
Epoch: 95 Error: 0.11097571423823153
Epoch: 96 Error: 0.10998102789743117
Epoch: 97 Error: 0.10901012078937741
Epoch: 98 Error: 0.10808859448011814
Epoch: 99 Error: 0.10719509833579731
Epoch: 100 Error: 0.1063234658113652

Código:

Main.py:

```
import csv
import numpy as np
from MPL import *
import matplotlib.pyplot as plt

def main():
    plt.figure(1)

    print("----------SENO CON REGRESION----------")
    function = int(input("[!]: "))

    trainingPatternsFileName = "Entradas.csv"
    x_funcFile = "PuntosX.csv"

    if function == 1:
        outputValuesFileName = "Salidas.csv"
        y_funcFile = "PuntosYSeno.csv"

    else:
        raise ValueError('Funcion Desconocida')


    epochs = 1500
    learning_rate = 0.1
    entries = 1  # of columns for the trainingPatternsFileName.
    neurons_in_hidden_layer = 8
    output_layer_neurons = 1

    net = MLP((entries, neurons_in_hidden_layer, output_layer_neurons),
('tanh', 'linear'))

    X = []
    y = []

    x_func = []
    y_func = []

    X.append(np.array(np.loadtxt(trainingPatternsFileName, delimiter=',',
usecols=0)))
    y.append(np.array(np.loadtxt(outputValuesFileName, delimiter=',',
usecols=0)))
```

```
    x_func.append(np.array(np.loadtxt(x_funcFile, delimiter=',',
usecols=0)))
    y_func.append(np.array(np.loadtxt(y_funcFile, delimiter=',',
usecols=0)))

    # Entrenamiento.

    plt.axhline(y=0, color='k')
    plt.axvline(x=0, color='k')
    plt.xlim([-5, 5])
    plt.ylim([-5, 10])

    error_list = []

    for i in range(epochs):
        error, pred = net.train(X, y, 1, learning_rate)
        error_list.append(error)
        print("Epoch:", i, "Error:", error)

        if i%10 == 0:
            plt.clf()
            plt.scatter(x_func, y_func, s=40, c='#0404B4')
            plt.plot(X[0], pred[0], color='green', linewidth=3)
            plt.show()
            plt.pause(0.2)
            plt.close()

            # if error < 0.03:
            if error < 0.015: #Para tanh y log.
                break

    plt.figure(2)
    plt.plot(error_list, color='red', linewidth=3)
    plt.pause(0.2)
    plt.close()

    results = np.array(net.predict(X)).T
    np.savetxt("Results.csv", results, delimiter=",", fmt='%.4f')

if __name__ == "__main__":
    main()
```

MPL.py

```python
import numpy as np
from activations import *
import matplotlib.pyplot as plt

class MLP:
    def __init__(self, layers_dim, activations):
        #Atributes
        self.W = [None]
        self.b = [None]
        self.f = [None]
        self.n = layers_dim
        self.L = len(layers_dim) - 1

        #Initialization of sinaptic weights and bias.
        for l in range(1, self.L + 1):
            self.W.append(-1 + 2 * np.random.rand(self.n[l], self.n[l-1]))
            self.b.append(-1 + 2 * np.random.rand(self.n[l], 1))

        #Fill activation functions list
        for act in activations:
            self.f.append(activate(act))

    def predict(self, X):
        a = np.asanyarray(X)
        for l in range(1, self.L + 1):
            z = np.dot(self.W[l], a) + self.b[l]
            a = self.f[l](z)
        return a

    def train(self, X, Y, epochs, learning_rate):
        X = np.asanyarray(X)
        Y = np.asanyarray(Y).reshape(self.n[-1], -1)
        P = X.shape[1]
        error = 0

        for _ in range(epochs):
            #Stocastic Gradient Descend
            for p in range(P):
                A = [None] * (self.L + 1)
                dA = [None] * (self.L + 1)
                lg = [None] * (self.L + 1)

                #Propagation
                A[0] = X[:,p].reshape(self.n[0], 1)
                for l in range(1, self.L + 1):
```

```
                    z = np.dot(self.W[l], A[l-1]) + self.b[l]
                    A[l], dA[l] = self.f[l](z, derivative=True)

                #regresion
                for l in range(self.L, 0, -1):
                    if l == self.L:
                        #lg = Local Gradient
                        lg[l] = (Y[:, p] - A[l]) * dA[l]
                    else:
                        lg[l] = np.dot(self.W[l+1].T, lg[l+1]) * dA[l]

                #Weights updates
                for l in range(1, self.L + 1):
                    self.W[l] += learning_rate * np.dot(lg[l], A[l-1].T)
                    self.b[l] += learning_rate * lg[l]

        predictions = self.predict(X)
        for i in range(len(predictions[0])):
            error += abs((Y[0][i] - predictions[0][i]))
        error /= len(Y[0])
        return error, predictions
```

Funciones de activación:

```
import numpy as np

def linear(z, derivative = False):
    a = z
    if derivative:
        da = np.ones(z.shape)
        return a, da
    return a

def tanh(z, derivative = False):
    a = np.tanh(z)
    if derivative:
        da = (1 + a) * (1 - a)
        return a, da
    return a

def sigmoid(z, derivative = False):
    a = 1 / (1 + np.exp(-z))
    if derivative:
        da = a * (1 - a)
        return a, da
```

```
    return a

def relu(z, derivative = False):
    a = z * (z >= 0)
    if derivative:
        da = np.array(z >= 0 , dtype=float)
        return a, da
    return a

def activate(function_name):
    if function_name == 'linear':
        return linear
    elif function_name == 'tanh':
        return tanh
    elif function_name == 'sigmoid':
        return sigmoid
    elif function_name == 'relu':
        return relu
    else:
        raise ValueError('function_name unknown')
```