

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.

Texto

Descripción generada automáticamente

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.

Texto

Descripción generada automáticamente

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.

Texto

Descripción generada automáticamente

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.

Texto

Descripción generada automáticamente

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:

Imagen que contiene Interfaz de usuario gráfica

Descripción generada automáticamenteImagen en blanco y negro

Descripción generada automáticamente con confianza bajaImagen que contiene Interfaz de usuario gráfica

Descripción generada automáticamente

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

Interfaz de usuario gráfica, Texto

Descripción generada automáticamenteTexto

Descripción generada automáticamenteInterfaz de usuario gráfica, Texto

Descripción generada automáticamente

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:

Pantalla de computadora con letras

Descripción generada automáticamente con confianza baja

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.

Texto

Descripción generada automáticamente

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.

Texto

Descripción generada automáticamente

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

Texto

Descripción generada automáticamente

**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:**

**Imagen que contiene Texto

Descripción generada automáticamente**

**Gráfico, Gráfico de líneas

Descripción generada automáticamente**

**Gráfico

Descripción generada automáticamente**

**Gráfico, Gráfico de líneas

Descripción generada automáticamente**

**Gráfico, Gráfico de líneas

Descripción generada automáticamente**

**Gráfico, Gráfico de líneas

Descripción generada automáticamente**

**Gráfico, Gráfico de líneas

Descripción generada automáticamente**

**Gráfico, Gráfico de líneas

Descripción generada automáticamente**

**Gráfico, Gráfico de líneas

Descripción generada automáticamente**

**Gráfico, Gráfico de líneas

Descripción generada automáticamente**

**Gráfico, Gráfico de líneas

Descripción generada automáticamente**

**Gráfico, Gráfico de líneas

Descripción generada automáticamente**

**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**

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')