Centro Universitario de Ciencias Exactas e Ingenierías

Licenciatura en Ingeniería en Computación

Materia: Seminario de Solución de Problemas de Inteligencia Artificial II. Clave: I7041.

Profesor: Valdés López Julio Esteban

Estudiante: Silva Moya José Alejandro. Código: 213546894.

Práctica VI: MLP – Multi-Layer Perceptron

Problema a resolver: Programar una MLP con una capa oculta y dos neuronas en la misma, para poder resolver el problema de clasificación de la compuerta lógica XOR y XNOR, que previamente eran imposibles con un perceptrón común.

Desarrollo

Lo primero que realizamos fue un documento en donde programamos las funciones de activación más comunes, que tendríamos a nuestra disposición en este trabajo; la lista abarca las funciones: linear, tangente hiperbólica, sigmoide y reLu. Podemos ver su implementación a continuación. Todas son capaces de retornar su forma normal y su forma derivada.

```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
```

```python
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')
```

Una vez hecho este archivo nos podemos dirigir a la creación de la MLP.

```
5   class MLP:
6       def __init__(self, layers_dim, activations):
7           #Atributes
8           self.W = [None] #Matrix with sinaptic weights of each layer. As the
9           self.b = [None] #The same as W, but with the Bias.
10          self.f = [None] #The same as W, but here we will gather every activ
11          self.n = layers_dim
12          self.L = len(layers_dim) - 1 #Max number of layers.
13
14          #Initialization of sinaptic weights and bias.
15          for l in range(1, self.L + 1):
16              self.W.append(-1 + 2 * np.random.rand(self.n[l], self.n[l-1]))
17              self.b.append(-1 + 2 * np.random.rand(self.n[l], 1))
18
19          #Fill activation functions list
20          for act in activations:
21              self.f.append(activate(act))
```

Comenzamos declarando nuestras variables de clase e inicializando nuestros pesos sinápticos y bias, que en este caso funcionarán por capas debido al modelo que estamos trabajando. Inicializamos en None con el propósito de no llenar de basura nuestros primeros registros, y evitar problemas.

La generación de datos iniciales sigue siendo aleatoria.

Finalmente cargamos nuestra lista de funciones de activación, que serán las usadas en las capas ocultas y en la de salida.

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

Nuestra función de predicción funciona basándonos en las salidas de una capa, que se convierten en las entradas de la capa siguiente.

```
31      def train(self, X, Y, epochs, learning_rate):
32          X = np.asanyarray(X)
33          Y = np.asanyarray(Y).reshape(self.n[-1], -1)
34          P = X.shape[1]
35          error = 0
36
37          for _ in range(epochs):
38              #Stocastic Gradient Descend
39              for p in range(P):
40                  A = [None] * (self.L + 1)
41                  dA = [None] * (self.L + 1)
42                  lg = [None] * (self.L + 1)
43
44                  #Propagation
45                  A[0] = X[:,p].reshape(self.n[0], 1)
46                  for l in range(1, self.L + 1):
47                      z = np.dot(self.W[l], A[l-1]) + self.b[l]
48                      A[l], dA[l] = self.f[l](z, derivative=True)
49
50                  #Backpropagation
51                  for l in range(self.L, 0, -1):
52                      if l == self.L:
53                          #lg = Local Gradient
54                          lg[l] = (Y[:, p] - A[l]) * dA[l]
55                      else:
56                          lg[l] = np.dot(self.W[l+1].T, lg[l+1]) * dA[l]
57
58                  #Weights updates
59                  for l in range(1, self.L + 1):
60                      self.W[l] += learning_rate * np.dot(lg[l], A[l-1].T)
61                      self.b[l] += learning_rate * lg[l]
```

Nuestra función de entrenamiento trabaja con base en la Gradiente Descendente Estocástica. Tendremos en una lista los resultados de entrenamiento de las capaz después de haberlas pasado por la función de activación correspondiente, y en otra lista tendremos los resultados con la función de activación en su forma derivada. Finalmente tendremos una tercera lista en donde estaremos almacenando los datos de las gradientes locales.

Realizaremos primero la propagación de información de toda la red, con una operación en Z bastate similar a la que hemos trabajado en el pasado, pero ahora se verá afectada por la posición en la que nos encontremos en el modelo (en qué capa estemos), y después almacenaremos los resultados de la información procesada por las dos formas de las funciones de activación correspondientes a cada capa.

Seguido de esto realizaremos la retropropagación. Para esto manejaremos dos tipos de resultados distintos, que se verán seleccionados dependiendo de en qué parte del modelo estemos ubicados en un tiempo determinado. Mientras estemos procesando datos en capas ocultas tendremos la opción codificada en el else, y si nos encontramos en la capa de salida utilizaremos el primer código.

Finalmente tenemos la actualización de nuestros pesos sinápticos y nuestro bias, que igualmente funciona de manera similar al modelo que veníamos trabajando desde antes. En este caso trabajaremos la gradiente local y la salida de la capa que recién obtenemos.

```
63            predictions = self.predict(X)
64            for i in range(len(predictions[0])):
65                # error += (Y[0][i] - predictions[0][i])
66                error += abs((Y[0][i] - predictions[0][i]))
67            # print(error)
68            error /= len(Y[0])
69            return error
70
```

Añadido a esto tenemos un snipet para el cálculo del error por épocas, para su posterior graficación.

Después de todo esto podemos proceder a ver el main.

```
49      net = MLP((entries, neurons_in_hidden_layer, output_layer_neurons), ('tanh', 'sigmoid'))
50      # First parenthesis:
51          # First position: Number of entrances X to the NN.
52          # Second to n-1 position: Number of neurons in hidden layers.
53          # Last position: Number of neurons in the output layer.
54      # Second parenthesis: Activation functions for the hidden and output layers.
```

Comenzamos inicializando nuestro modelo, que tendrá n entradas, n cantidad de neuronas en **una sola capa oculta que tendremos en este modelo**, y la cantidad de neuronas en la capa de salida. Además de esto tenemos una lista de funciones de activación, que serán las que se usarán en las capas ocultas y en la de salida, por eso es que en este caso solo tenemos dos funciones, y podemos usar las que queramos, solo que en este caso determinamos que eran las mejores opciones.

```
72      plt.figure(1)
73      if logic_gate == "xor":
74          plt.title("XOR with MLP", fontsize=20)
75          plt.plot(0,0,'r*')
76          plt.plot(0,1,'b*')
77          plt.plot(1,0,'b*')
78          plt.plot(1,1,'r*')
79      elif logic_gate == "xnor":
80          plt.title("XNOR with MLP", fontsize=20)
81          plt.plot(0,0,'b*')
82          plt.plot(0,1,'r*')
83          plt.plot(1,0,'r*')
84          plt.plot(1,1,'b*')
85
86      error_list = []
```

A continuación, generamos un plano dependiendo de la compuerta que decidimos resolver. Y generamos una lista de errores para al final poder realizar la gráfica correspondiente.

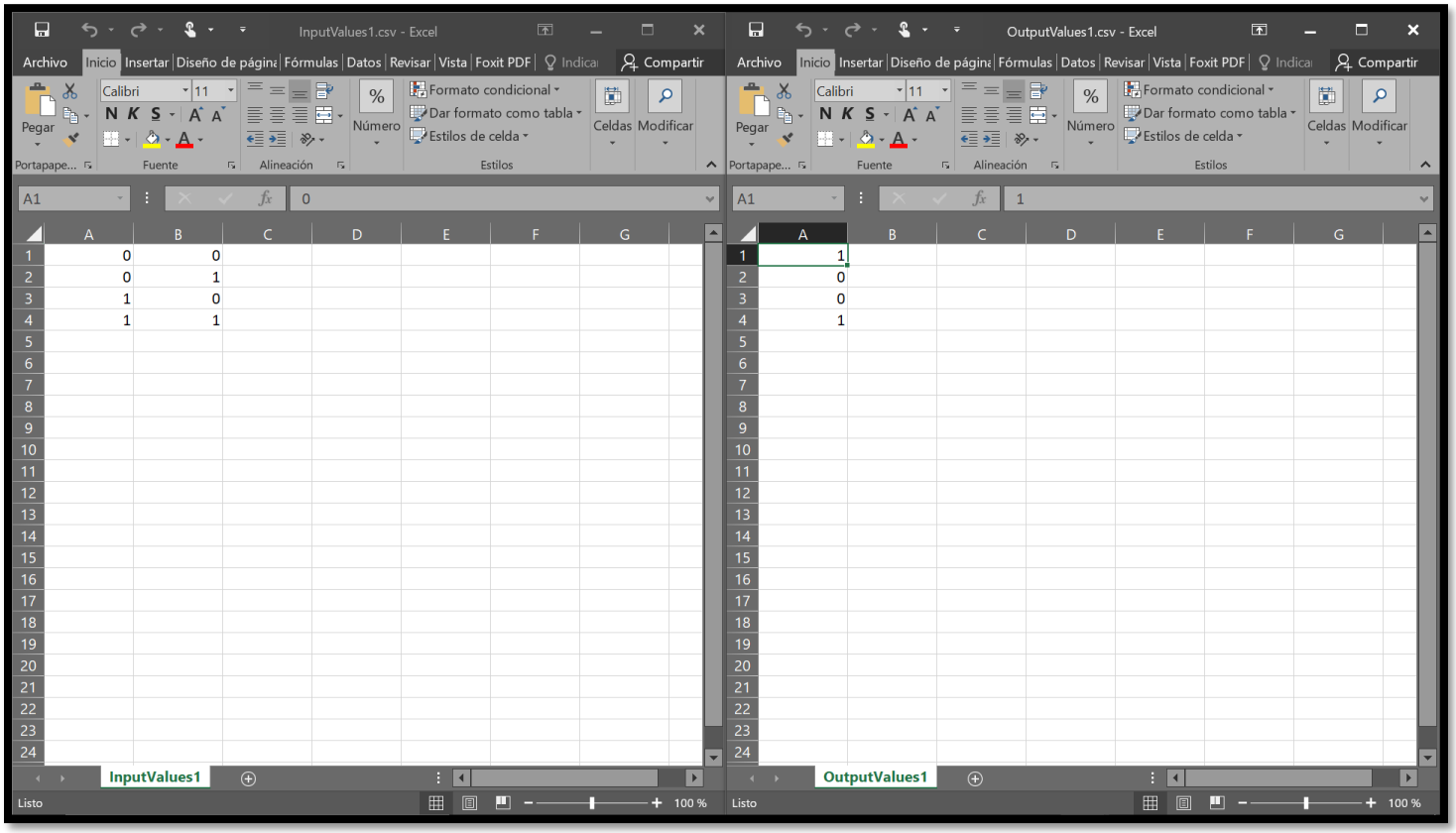
```
    for i in range(epochs):
        error = net.train(X, y, 1, learning_rate)
        error_list.append(error)
        graphLearning(0,0)

        # xx, yy = np.meshgrid(np.arange(-0.1, 1.1, 0.6), np.arange(-0.1, 1.1, 0.6))
        xx, yy = np.meshgrid(np.arange(-1, 2.1, 0.1), np.arange(-1, 2.1, 0.1))
        x_input = [xx.ravel(), yy.ravel()]
        zz = net.predict(x_input)
        zz = zz.reshape(xx.shape)

        plt.contourf(xx, yy, zz, alpha=0.8, cmap=plt.cm.RdBu)

        # plt.xlim([-0.25, 1.25])
        # plt.ylim([-0.25, 1.25])
        plt.xlim([-1, 2])
        plt.ylim([-1, 2])
        plt.grid()
        plt.show()
        if error < 0.15:
            break

    plt.figure(2)

    for i in range(len(error_list)):
        graphError(i, error_list[i])

    results = np.array(net.predict(X)).T
    np.savetxt("Results.csv", results, delimiter=",", fmt='%.0f')
```

Ahora, para el entrenamiento en cuestión y la gráfica del aprendizaje comenzaremos por hacer lo siguiente en un ciclo de acuerdo al límite de épocas o hasta que se alcance el error mínimo:

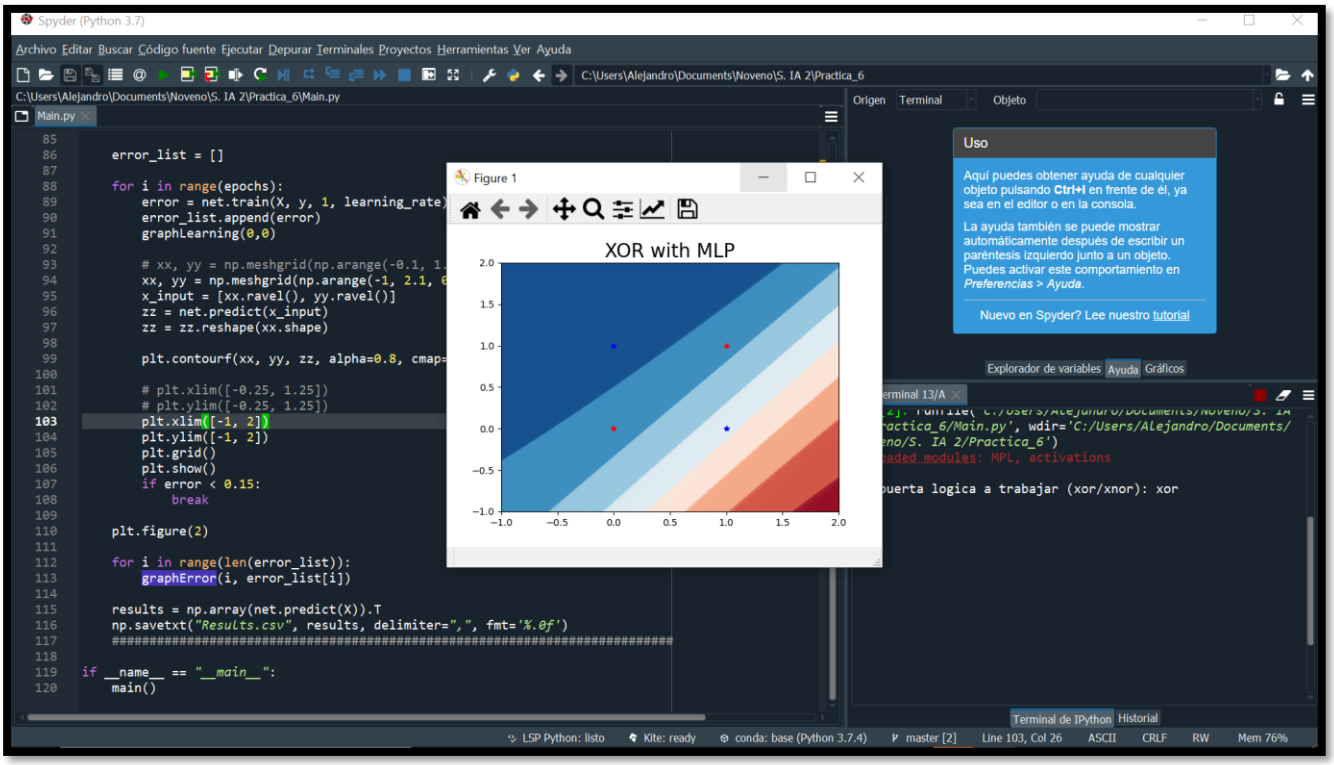
- Entrenar y obtener el error.
- Guardar el error para su posterior graficación.
- Graficar el aprendizaje en el plano de desición.
- Generar un meshgrid para poder dibujar correctamente el entrenamiento,
- Dibujar los resultados.
- Delimitar nuestro modelo en el hiperplano.
- Si nuestro error se alcanza: terminar.

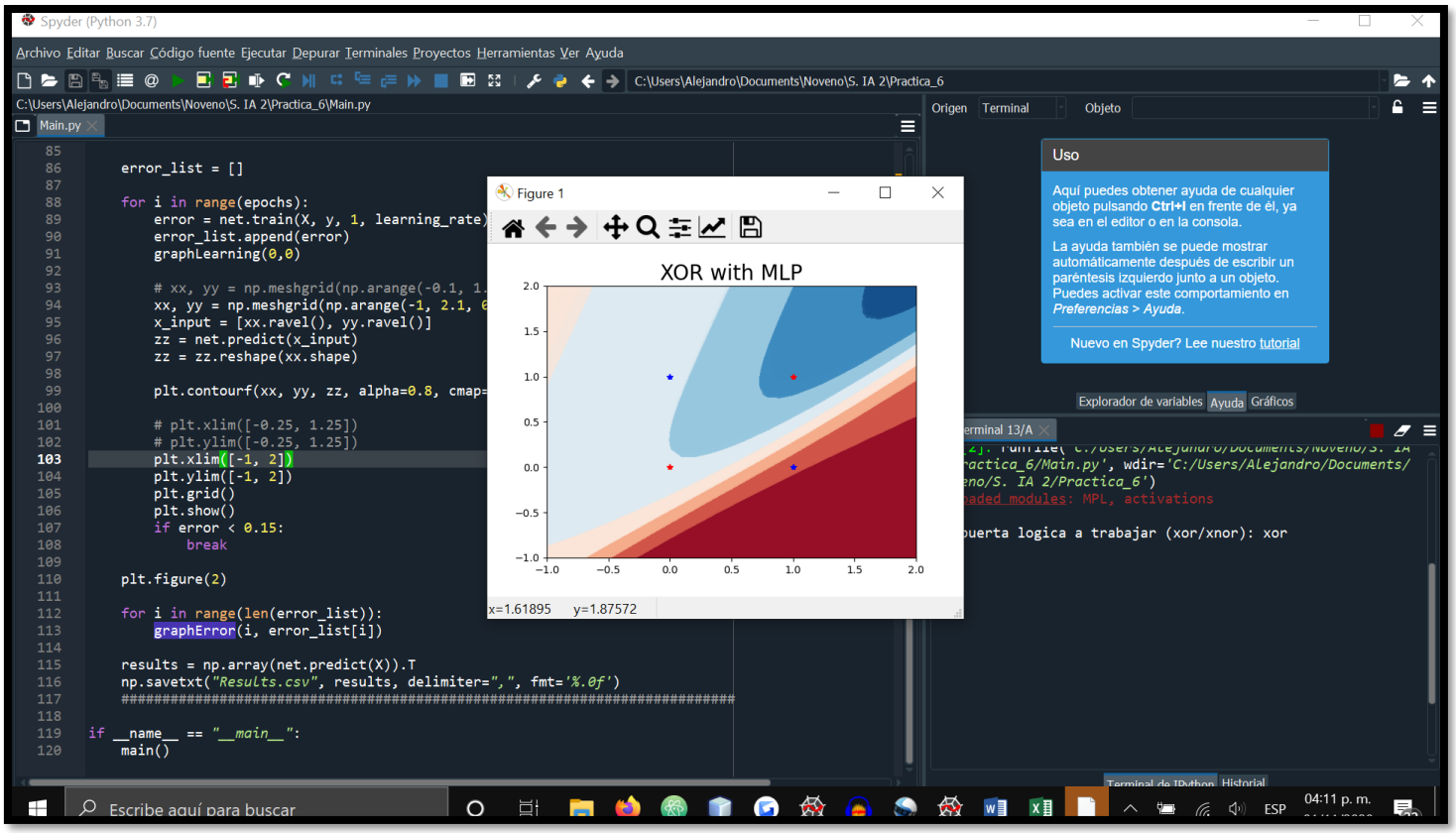
Posteriormente en otro ciclo imprimimos el error, y finalmente obtenemos en un CSV los resultados arrojados por la red neuronal.

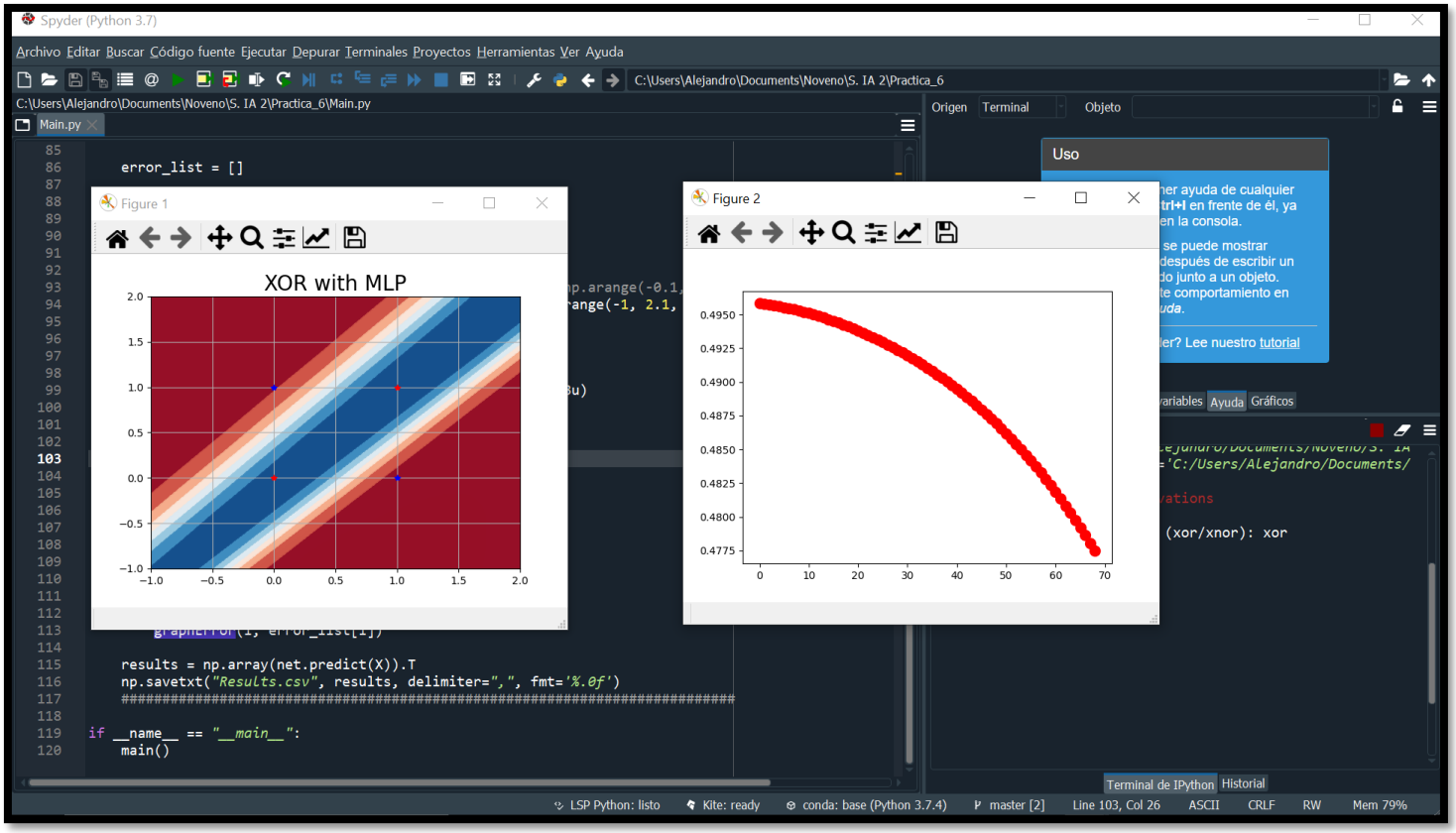
Resultados

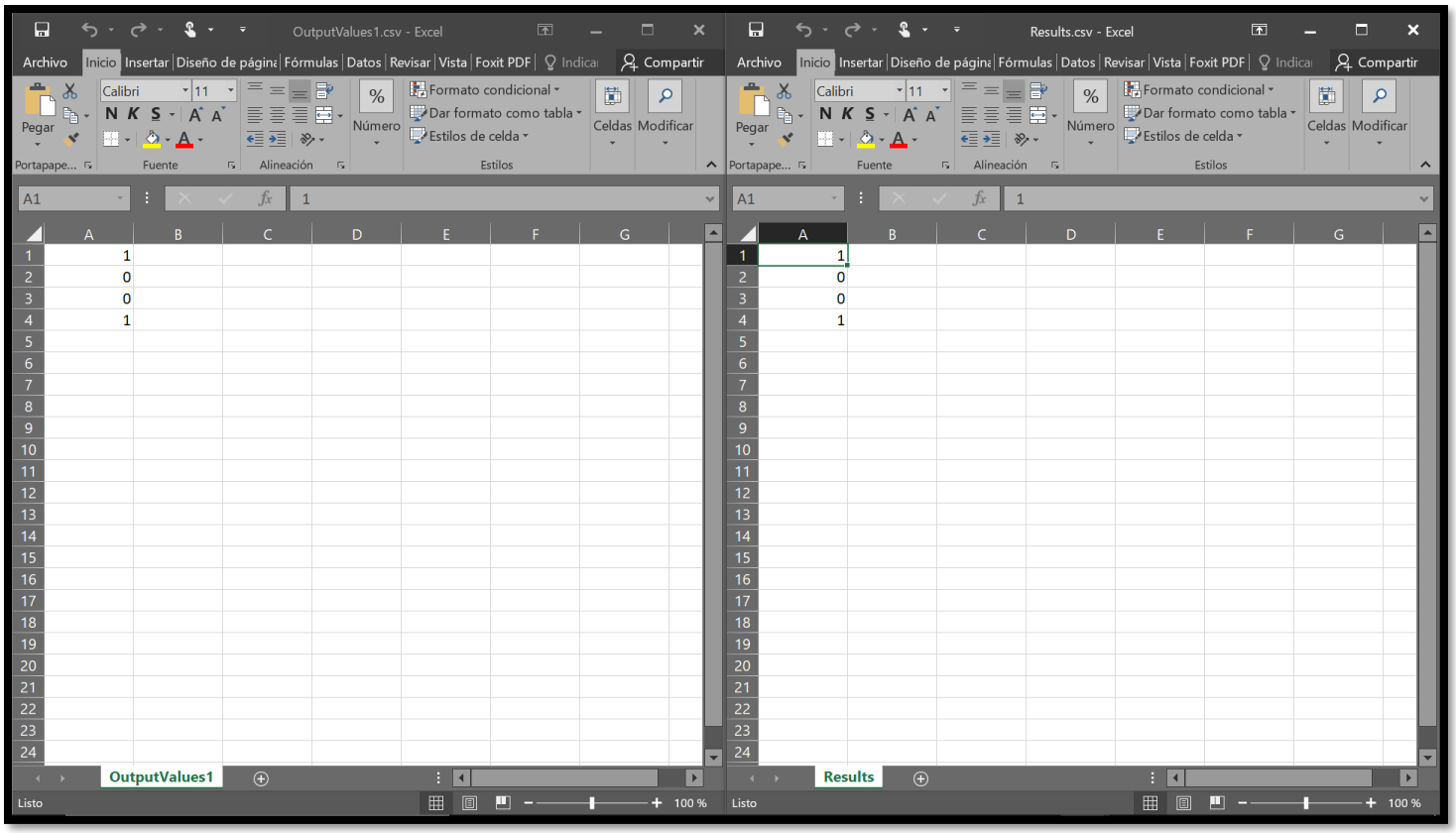


En la imagen anterior podemos observar las entradas para una compuerta lógica, y las salidas deseadas para el caso de la compuerta XOR. Observemos el desempeño y resultados.




Spyder (Python 3.7)
Archivo Editar Buscar Código fuente Ejecutar Depurar Terminales Proyectos Herramientas Ver Ayuda
C:\Users\Alejandro\Documents\Noveno\S. IA 2\Practica_6
Main.py
error_list = []
for i in range(epochs):
error = net.train(X, y, 1, learning_rate)
error_list.append(error)
graphLearning(0,0)
xx, yy = np.meshgrid(np.arange(-0.1, 1.
xx, yy = np.meshgrid(np.arange(-1, 2.1,
x_input = [xx.ravel(), yy.ravel()]
zz = net.predict(x_input)
zz = zz.reshape(xx.shape)
plt.contourf(xx, yy, zz, alpha=0.8, cmap=
# plt.xlim([-0.25, 1.25])
# plt.ylim([-0.25, 1.25])
plt.xlim([-1, 2])
plt.ylim([-1, 2])
plt.grid()
plt.show()
if error < 0.15:
break
plt.figure(2)
for i in range(len(error_list)):
graphError(i, error_list[i])
results = np.array(net.predict(X)).T
np.savetxt("Results.csv", results, delimiter=",", fmt='%.0f')
if __name__ == "__main__":
main()
Figure 1
XOR with MLP
x=1.61895 y=1.87572
Origen Terminal
Objeto
Uso
Aquí puedes obtener ayuda de cualquier objeto pulsando Ctrl+I en frente de él, ya sea en el editor o en la consola.
La ayuda también se puede mostrar automáticamente después de escribir un paréntesis izquierdo junto a un objeto. Puedes activar este comportamiento en Preferencias > Ayuda.
Nuevo en Spyder? Lee nuestro tutorial
Explorador de variables Ayuda Gráficos
Terminal 13/A
Terminal de IPython Historial
Escribe aquí para buscar
ESP
04:11 p. m.


Spyder (Python 3.7)
Archivo Editar Buscar Código fuente Ejecutar Depurar Terminales Proyectos Herramientas Ver Ayuda
C:\Users\Alejandro\Documents\Noveno\S. IA 2\Practica_6
Main.py
error_list = []
results = np.array(net.predict(X)).T
np.savetxt("Results.csv", results, delimiter=",", fmt='%.0f')
if __name__ == "__main__":
main()
Figure 1
XOR with MLP
Figure 2
Origen Terminal
Objeto
Uso
Terminal de IPython Historial
LSP Python: listo
Kite: ready
conda: base (Python 3.7.4)
master [2]
Line 103, Col 26
ASCII
CRLF
RW
Mem 79%



Como podemos observar, el entrenamiento ha sido exitoso.

Ahora veamos el caso del entrenamiento para la compuerta XNOR.

InputValues1.csv - Excel

	A	B
1	0	0
2	0	1
3	1	0
4	1	1

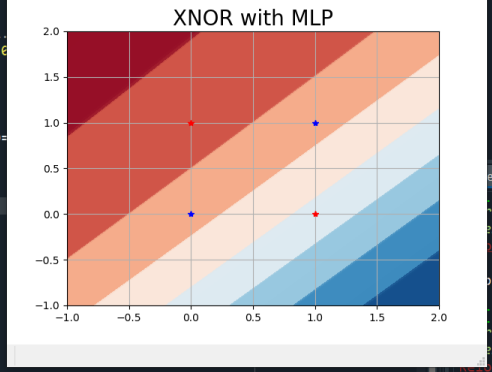
OutputValues2.csv - Excel

	A
1	0
2	1
3	1
4	0

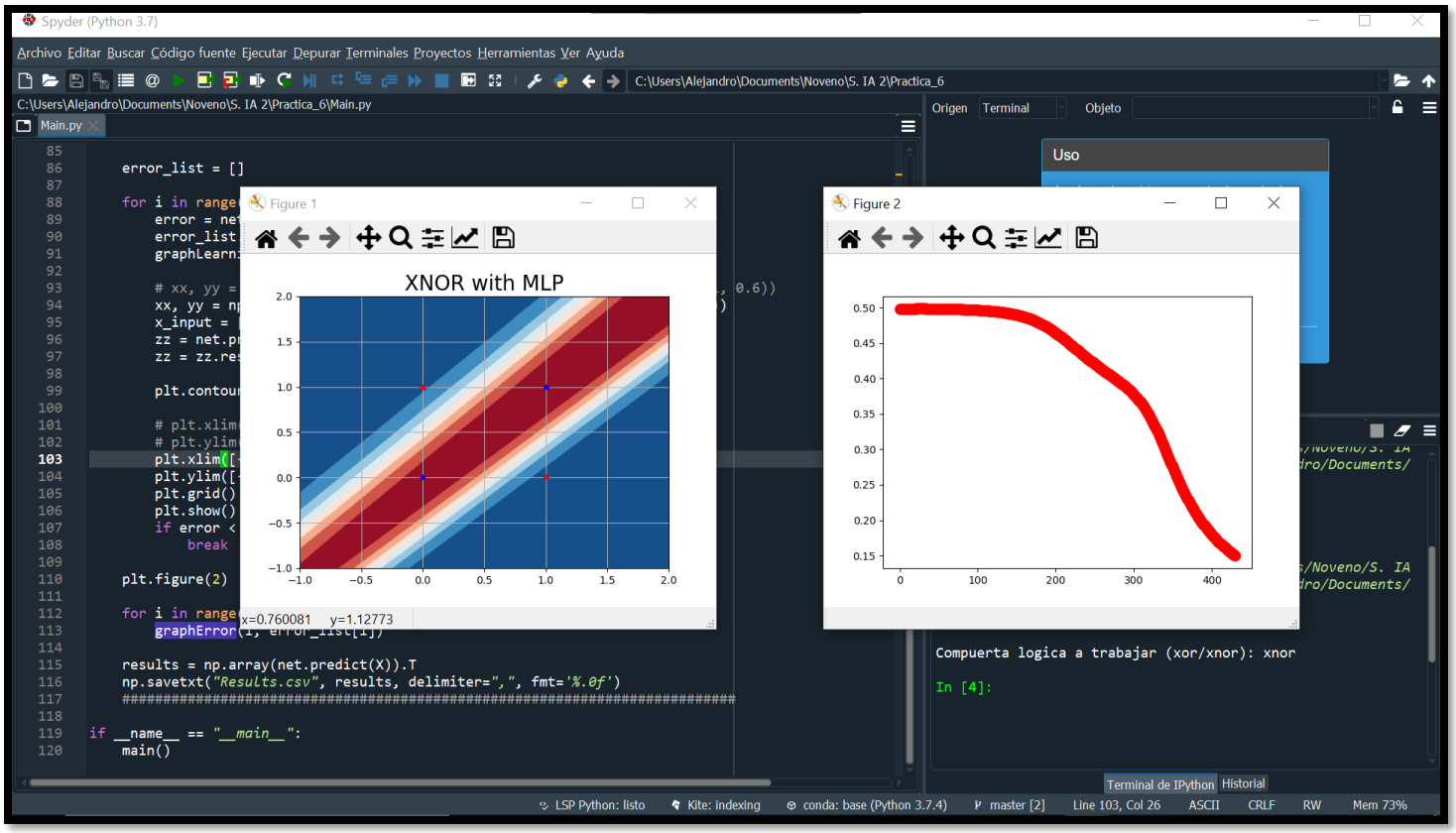
Spyder (Python 3.7)

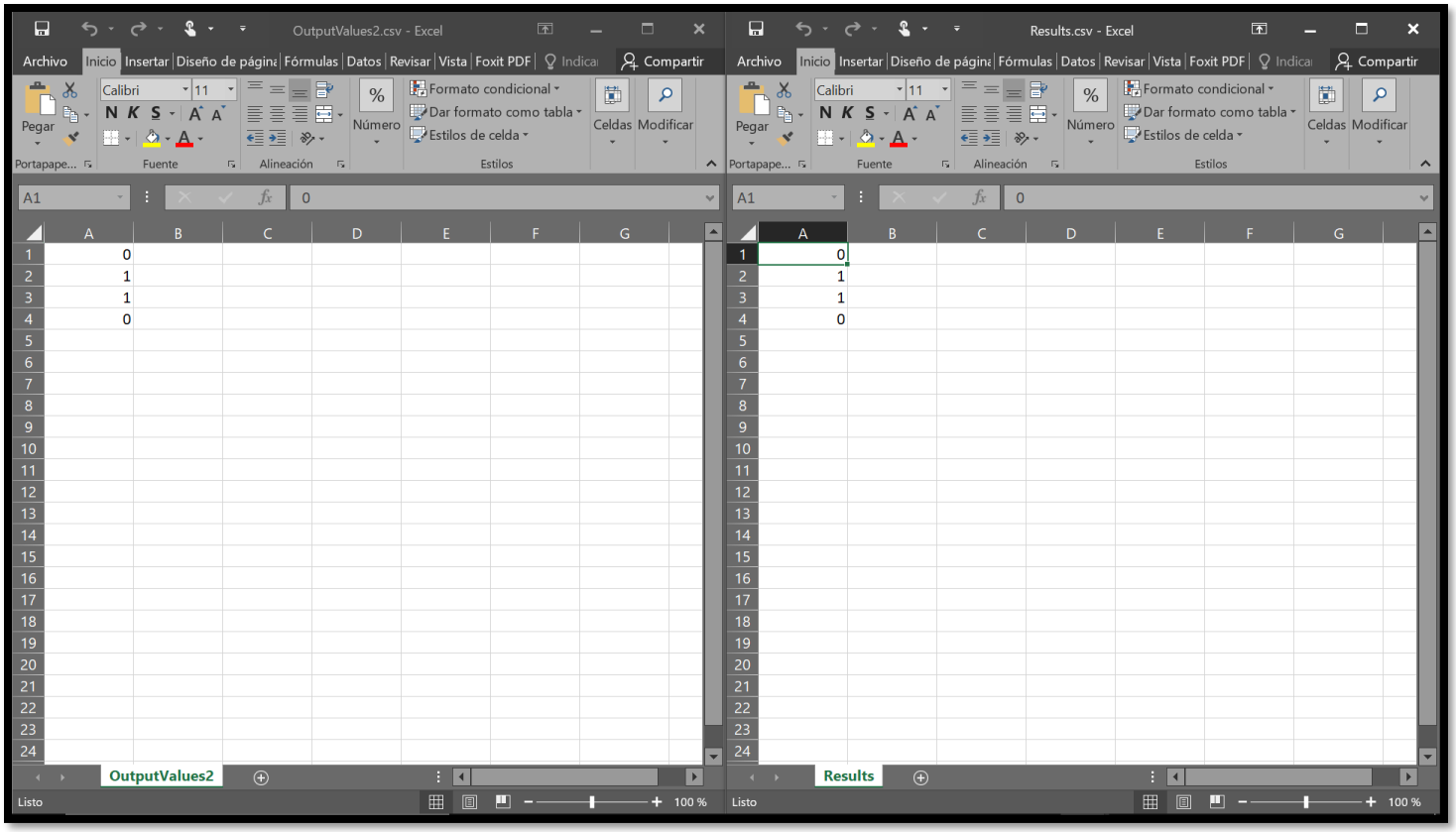
```
    error_list = []

    for i in range(epochs):
        error = net.train(X, y, 1, learning_rate)
        error_list.append(error)
        graphLearning(0,0)

        # xx, yy = np.meshgrid(np.arange(-0.1, 1.
        xx, yy = np.meshgrid(np.arange(-1, 2.1, 0
        x_input = [xx.ravel(), yy.ravel()]
        zz = net.predict(x_input)
        zz = zz.reshape(xx.shape)

        plt.contourf(xx, yy, zz, alpha=0.8, cmap=

        # plt.xlim([-0.25, 1.25])
        # plt.ylim([-0.25, 1.25])
        plt.xlim([-1, 2])
        plt.ylim([-1, 2])
        plt.grid()
        plt.show()
        if error < 0.15:
            break

    plt.figure(2)

    for i in range(len(error_list)):
        graphError(i, error_list[i])

    results = np.array(net.predict(X)).T
    np.savetxt("Results.csv", results, delimiter=",", fmt='%.0f')
    ########################################################################

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

Figure 1




```
Compuerta logica a trabajar (xor/xnor): xor

runfile('C:/Users/ALejandro/Documents/Noveno/S. IA
ractica_6/Main.py', wdir='C:/Users/ALejandro/Documents/
eno/S. IA 2/Practica_6')
Reloaded modules: MPL, activations

Compuerta logica a trabajar (xor/xnor): xnor
```





Una vez más el entrenamiento fue exitoso.

Conclusión

Como podemos observar, la resolución de este problema era llevarlo a más dimensiones, lo que se traduciría como agregar capas ocultas en el modelo, y aunque eso desafortunadamente convierte a la solución y a las matemáticas implicadas en un escenario bastante más complejo, los resultados arrojados son más que favorables, y evidentemente abren las puertas para poder resolver problemas mucho más complejos, y que por ende serían seguramente más cercanos a casos de la vida real.

Código

Activations.py

```
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')
```

MLP.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]                                                    #Matrix with sinaptic weights of
each layer. As the first layer (the entries) doesn't have weights, we use the None to make sure the first space
in the matrix has nothing, but the rest does.
        self.b = [None]                                                    #The same as W, but with the
Bias.
        self.f = [None] #The same as W, but here we will gather every activation function for each individual layer.
        self.n = layers_dim
        self.L = len(layers_dim) - 1 #Max number of layers.

        #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=1000, learning_rate=0.2):
    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)

            #Backpropagation
            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 += (Y[0][i] - predictions[0][i])
            error += abs((Y[0][i] - predictions[0][i]))
        # print(error)
        error /= len(Y[0])
        return error
```

Main.py

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

# Error graphing function.
def graphLearning(x_coordinate, y_coordinate):
    plt.plot(x_coordinate, y_coordinate)
    plt.pause(0.000000001)

def graphError(x_coordinate, y_coordinate):
    plt.plot(x_coordinate, y_coordinate, 'ro', markersize=10)
    plt.pause(0.000001)

def main():
    logic_gate = input("Compuerta logica a trabajar (xor/xnor): ")

    trainingPatternsFileName = "InputValues1.csv"
```

```
    if logic_gate == "xor":
        outputValuesFileName = "OutputValues1.csv"
    elif logic_gate == "xnor":
        outputValuesFileName = "OutputValues2.csv"
    else:
        raise ValueError('Compuerta Desconocida')
    # CSV documents selection.

    epochs = 10000
    learning_rate = 0.3
    neurons_in_hidden_layer = 2

    file = open(trainingPatternsFileName)
    rows = len(file.readlines())
    file.close()
    #To obtain the number of rows from the CSV file

    file = open(trainingPatternsFileName,'r')
    reader = csv.reader(file,delimiter=',')
    entries = len(next(reader))
    file.close()
    #To obtain the number of columns from the CSV file

    file = open(outputValuesFileName,'r')
    reader = csv.reader(file,delimiter=',')
    output_layer_neurons = len(next(reader))
    file.close()
    #To obtain the number of neurons for the program. The number of output columns tells us the number of
neurons.

    net = MLP((entries, neurons_in_hidden_layer, output_layer_neurons), ('tanh', 'sigmoid'))
    # First parenthesis:
        # First position: Number of entrances X to the NN.
        # Second to n-1 position: Number of neurons in hidden layers.
        # Last position: Number of neurons in the output layer.
    # Second parenthesis: Activation functions for the hidden and output layers.

    ###########################################################################
    patterns = []
    y = []

    for i in range(entries):
        x = np.array(np.loadtxt(trainingPatternsFileName, delimiter=',', usecols=i))
        patterns.append(x)
    X = np.array(patterns)

    for i in range(output_layer_neurons):
        y.append(np.array(np.loadtxt(outputValuesFileName, delimiter=',', usecols=i)))
    #Obtaining training patterns in X and output values in y.
    ###########################################################################

    # Training section.
```

```
    plt.figure(1)
    if logic_gate == "xor":
        plt.title("XOR with MLP", fontsize=20)
        plt.plot(0,0,'r*')
        plt.plot(0,1,'b*')
        plt.plot(1,0,'b*')
        plt.plot(1,1,'r*')
    elif logic_gate == "xnor":
        plt.title("XNOR with MLP", fontsize=20)
        plt.plot(0,0,'b*')
        plt.plot(0,1,'r*')
        plt.plot(1,0,'r*')
        plt.plot(1,1,'b*')

    error_list = []

    for i in range(epochs):
        error = net.train(X, y, 1, learning_rate)
        error_list.append(error)
        graphLearning(0,0)

        # xx, yy = np.meshgrid(np.arange(-0.1, 1.1, 0.6), np.arange(-0.1, 1.1, 0.6))
        xx, yy = np.meshgrid(np.arange(-1, 2.1, 0.1), np.arange(-1, 2.1, 0.1))
        x_input = [xx.ravel(), yy.ravel()]
        zz = net.predict(x_input)
        zz = zz.reshape(xx.shape)

        plt.contourf(xx, yy, zz, alpha=0.8, cmap=plt.cm.RdBu)

        # plt.xlim([-0.25, 1.25])
        # plt.ylim([-0.25, 1.25])
        plt.xlim([-1, 2])
        plt.ylim([-1, 2])
        plt.grid()
        plt.show()
        if error < 0.15:
            break

    plt.figure(2)

    for i in range(len(error_list)):
        graphError(i, error_list[i])

    results = np.array(net.predict(X)).T
    np.savetxt("Results.csv", results, delimiter=",", fmt='%.0f')
    ################################################################################

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

Link al repositorio

- https://github.com/TheGenesisX/S_IA_2/tree/master/Practica_6