Practica 5

Red neuronal multicapa

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

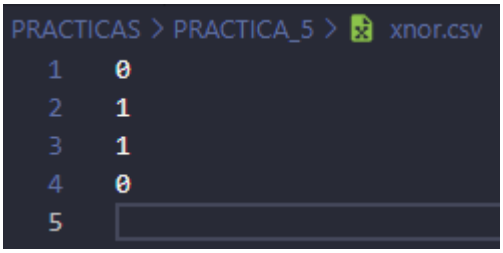
25/04/2023

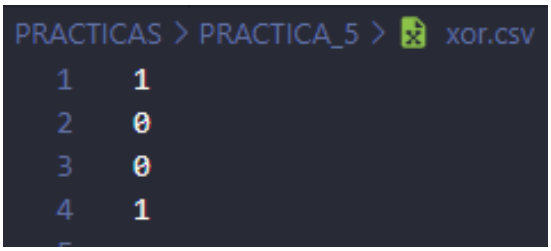
Introducción:

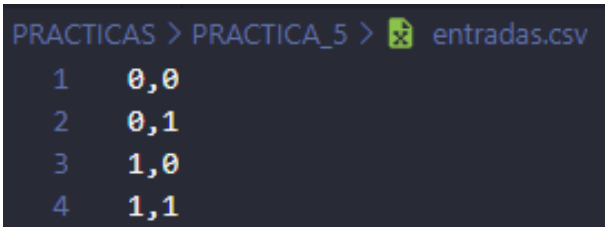
Para la realización de esta práctica se pide la simulación de una red multicapa, en este caso implementando se implementa una utilizando perceptrones, siendo posible poner capas ocultas con las neuronas necesarias para el entrenamiento, teniendo dos archivos csv, uno el cual se le dan las entradas y otro con las salidas esperadas. La red neuronal debe de ser capaz de graficar el error e ir iterando para que las salidas deseadas sean las correctas.

Desarrollo:

Para el desarrollo de esta práctica, y como punto principal se toman los valores que se quieren entrenar, teniéndolos estos en un archivo .csv llamados xor y xnor, así como los datos de entrenamiento, que en este caso serian los necesarios para clasificar estas compuertas.







Se crea un archivo **main.py** para mantener el flujo de los archivos, así como las llamadas a las funciones:

En esta sección se importan las librerías necesarias para la ejecución del programa. csv para la lectura de archivos csv, numpy para la manipulación de matrices y vectores, RedMulticapa es un módulo personalizado que contiene la clase MLP que define una red

neuronal multicapa, matplotlib.pyplot para la creación de gráficos y time para medir el tiempo de ejecución.

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

Estas son dos funciones auxiliares para la creación de gráficos. graphLearning dibuja la curva de aprendizaje de la red, mientras que graphError dibuja los puntos de error en el tiempo.

```
# Error graphing function.
def graphLearning(x_coordinate, y_coordinate):
    plt.plot(x_coordinate, y_coordinate)
    plt.pause(0.2)

def graphError(x_coordinate, y_coordinate):
    plt.plot(x_coordinate, y_coordinate, 'go', markersize=10)
    plt.pause(0.000000001)
```

Esta función main es la función principal del programa. Comienza con un bucle while que imprime un menú de opciones para seleccionar una compuerta lógica para trabajar (XOR o XNOR), o salir del programa. Luego, se solicita al usuario que ingrese el número de la opción deseada. Si la opción es 1 o 2, se asignan los nombres de archivo adecuados para los patrones de entrenamiento y los valores de salida. Si la opción es 3, se imprime un mensaje de despedida y se devuelve de la función.

```python
def main():
    print("Seleccione la compuerta logica a trabajar:")
    print("1. XOR")
    print("2. XNOR")
    print("3. Salir")

    while True:
        try:
            option = int(input("Ingrese el número de la opción que desea: "))
            if option == 1:
                logic_gate = "xor"
                trainingPatternsFileName = "entradas.csv"
                outputValuesFileName = "xor.csv"
                break
            elif option == 2:
                logic_gate = "xnor"
                trainingPatternsFileName = "entradas.csv"
                outputValuesFileName = "xnor.csv"
                break
            elif option == 3:
                print("¡Hasta luego!")
                return
            else:
                print("Opción inválida. Por favor ingrese una opción válida.")
        except ValueError:
            print("Entrada inválida. Por favor ingrese un número.")
```

En esta sección se seleccionan los archivos CSV de entrada y salida y se establecen los parámetros de la red neuronal. También se obtienen las dimensiones de los archivos CSV de entrada y salida. Luego, se crea la red neuronal con la clase MLP definida en el archivo RedMulticapa.py. La red neuronal tiene una capa de entrada con el mismo número de neuronas que la cantidad de salidas.

```python
epochs = 1000
learning_rate = 0.3
neurons_in_hidden_layer = 8

file = open(trainingPatternsFileName)
rows = len(file.readlines())
file.close()

file = open(trainingPatternsFileName,'r')
reader = csv.reader(file,delimiter=',')
entries = len(next(reader))
file.close()

file = open(outputValuesFileName,'r')
reader = csv.reader(file,delimiter=',')
output_layer_neurons = len(next(reader))
file.close()
        You, 4 seconds ago • Uncommitted changes
net = RedMulticapa((entries, neurons_in_hidden_layer, output_layer_neurons), ('tanh', 'sigmoid'))
```

Se está leyendo dos archivos de datos: uno con patrones de entrenamiento y otro con valores de salida esperados. El bucle for se utiliza para recorrer cada columna en ambos archivos y cargar los datos en dos listas separadas llamadas patterns e y, respectivamente. Luego, las listas de patrones se convierten en un array numpy llamado X, que se utilizará para entrenar la red neuronal.

La segunda lista y contiene los valores de salida esperados para cada patrón de entrada. Estos valores se utilizan más adelante en el código para compararlos con las salidas reales de la red neuronal durante el entrenamiento.

```
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)))
```

Se crea una figura de visualización con el título adecuado según la compuerta lógica que se esté utilizando ("XOR" o "XNOR"). Luego, según la compuerta lógica elegida, agrega cuatro puntos de diferentes colores (rojo y negro) a la figura para representar los cuatro posibles pares de entradas binarias. Finalmente, el código pausa la figura por 5 segundos para que el usuario pueda verla antes de continuar.

```
plt.figure(1)
if logic_gate == "xor":
    plt.title("XOR", fontsize=20)
    plt.plot(0,0,'r*')
    plt.plot(0,1,'k*')
    plt.plot(1,0,'k*')
    plt.plot(1,1,'r*')
    plt.pause(5)
elif logic_gate == "xnor":
    plt.title("XNOR", fontsize=20)
    plt.plot(0,0,'k*')
    plt.plot(0,1,'r*')
    plt.plot(1,0,'r*')
    plt.plot(1,1,'k*')
    plt.pause(5)

error_list = []
```

Se itera con un bucle for que entrena la red neuronal en un número determinado de épocas. Durante cada iteración, la red se entrena en el conjunto de patrones de entrada (X) y las salidas esperadas (y), con una tasa de aprendizaje especificada. El error de la red se registra en una lista de errores.

Si el número de iteraciones alcanza un múltiplo de 10, se llama a la función "graphLearning(0,0)", que traza un gráfico de los patrones de entrada. Luego, se crea una cuadrícula de puntos y se llama a la función "net.predict" para predecir la salida para cada punto en la cuadrícula. Estas salidas se muestran en el gráfico con una función de colores. También se muestran los puntos de entrada.

Finalmente, si el error de la red cae por debajo de un valor determinado (0.15 en este caso), se rompe el bucle for.

```python
for i in range(epochs):
    error = net.train(X, y, 1, learning_rate)
    error_list.append(error)
    if i%10 == 0:
        graphLearning(0,0)

        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.YlOrRd)

        plt.xlim([-1, 2])
        plt.ylim([-1, 2])
        plt.grid()
        plt.show()
        # plt.pause(2.5)
        # plt.close()
        print("iteracion", i)
        print("error ", error)

    if error < 0.15:
        break
```

Archivo RedNeuronal.py:

La clase RedMulticapa está diseñada para crear una red neuronal multicapa con un número variable de capas y neuronas por capa, así como con funciones de activación personalizadas. Esta clase tiene tres métodos principales: init(), predict() y train().

En el método init() se inicializan los atributos de la clase, incluyendo la matriz de pesos sinápticos, el vector de sesgos y la lista de funciones de activación para cada capa.

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

class RedMulticapa:
    def __init__(self, layers_dim, activations):
        #Atributes
        self.W = [None] #Matrix with sinaptic weights of each layer. As th
        self.b = [None] #The same as W, but with the Bias.
        self.f = [None] #The same as W, but here we will gather every acti
        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))
```

El método predict() se utiliza para realizar predicciones basadas en las entradas proporcionadas a la red neuronal. En este método, se realiza la propagación hacia adelante a través de la red neuronal.

```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 train() se utiliza para entrenar la red neuronal utilizando un algoritmo de descenso de gradiente estocástico con retropropagación. Este método utiliza un conjunto de datos de entrenamiento para ajustar los pesos sinápticos y los sesgos de la red.

```python
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]
```

```python
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
```

Conclusión:

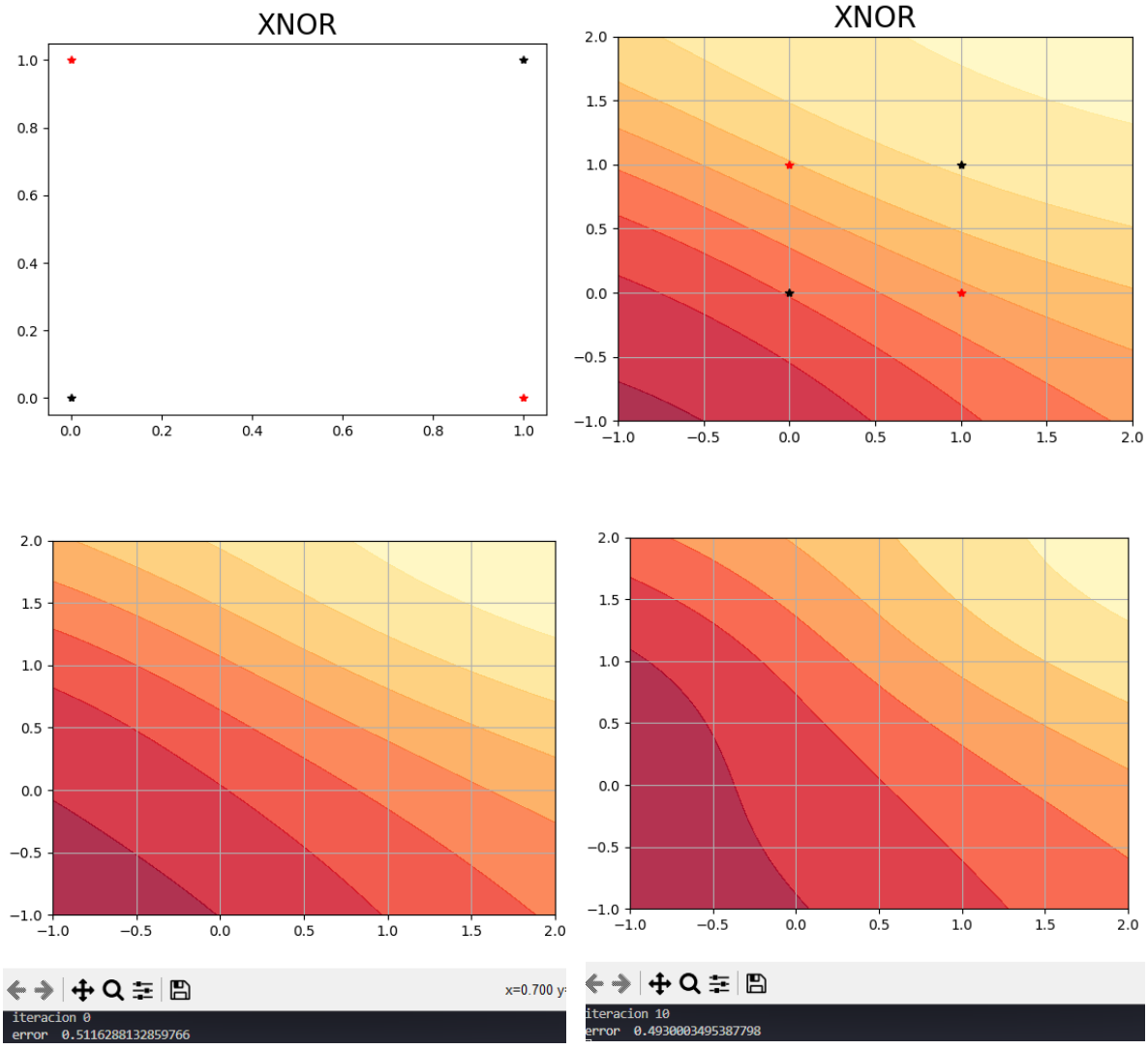
Se implementa un modelo de red neuronal artificial multicapa para tareas de clasificación y/o regresión. El modelo utiliza un algoritmo de descenso de gradiente estocástico con retropropagación para ajustar los pesos sinápticos y los sesgos de la red.

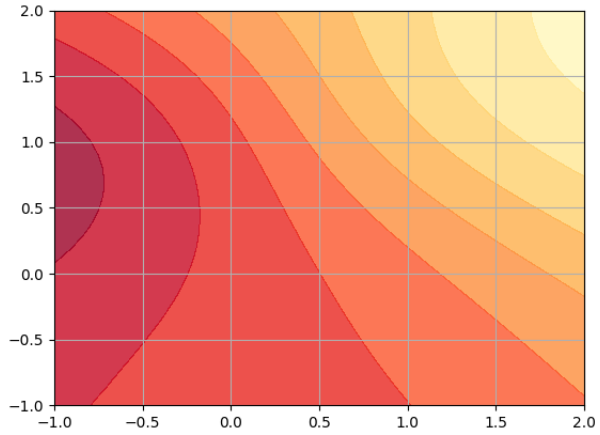
El archivo principal define la clase RedMulticapa y contiene los métodos necesarios para inicializar la red, realizar predicciones y entrenarla con un conjunto de datos de entrenamiento. También se utiliza una biblioteca llamada activations.py para definir las funciones de activación que se utilizarán en la red neuronal.

Resultados:

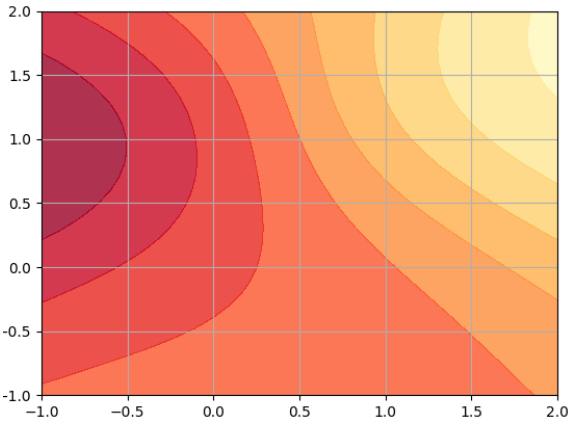
```
klenb@AlanMtz200 MINGW64 /u/8° SEMESTRE/SEMINARIO IA 2/PRACTICAS/PRACTICA_5 (master)
$ python Main.py
Seleccione la compuerta logica a trabajar:
1. XOR
2. XNOR
3. Salir
Ingrese el número de la opción que desea:
```

Compuerta para clasificar:

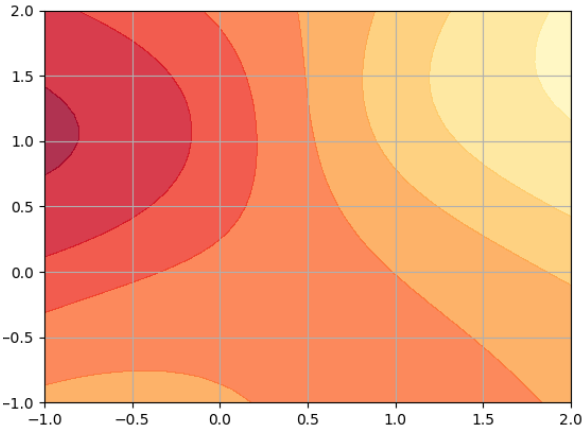




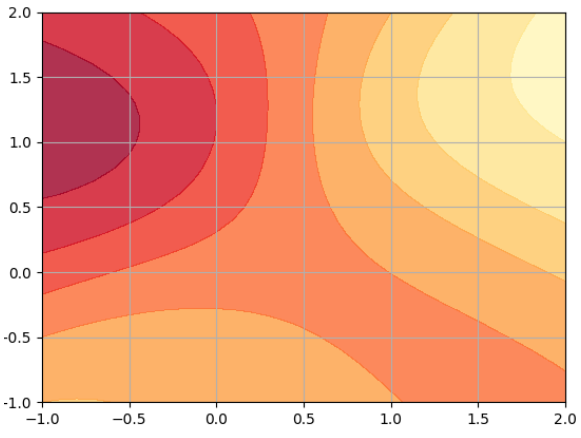
iteracion 20
error 0.4861198004752072



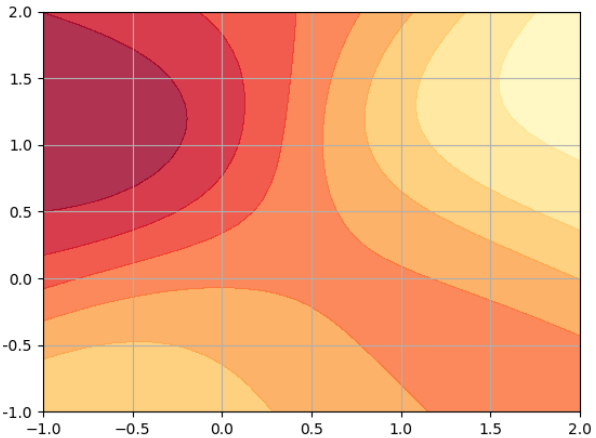
iteracion 30
error 0.47844684101223334



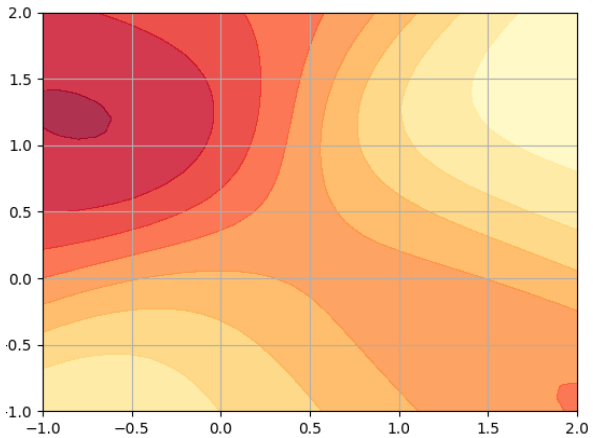
iteracion 40
error 0.46902385274503566



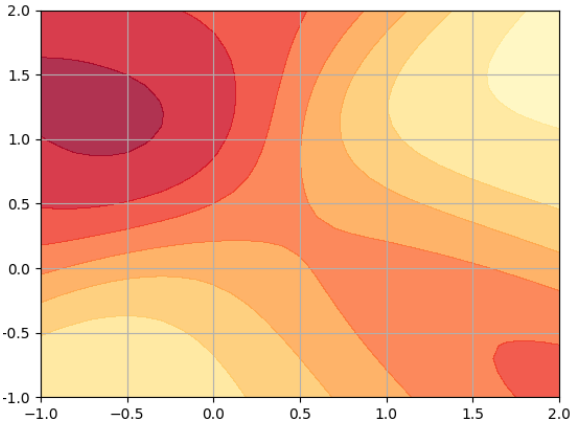
iteracion 50
error 0.45704519468045757



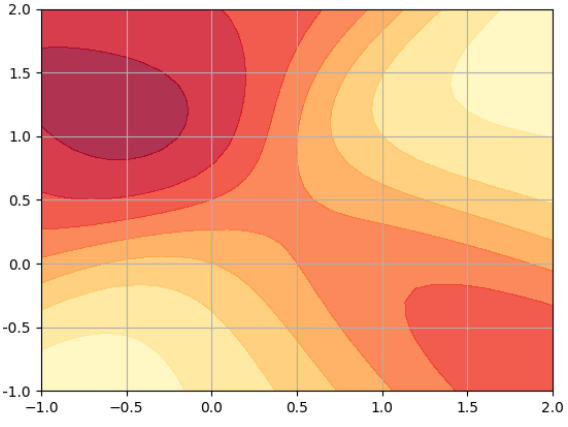
iteracion 60
error 0.4418266949592382



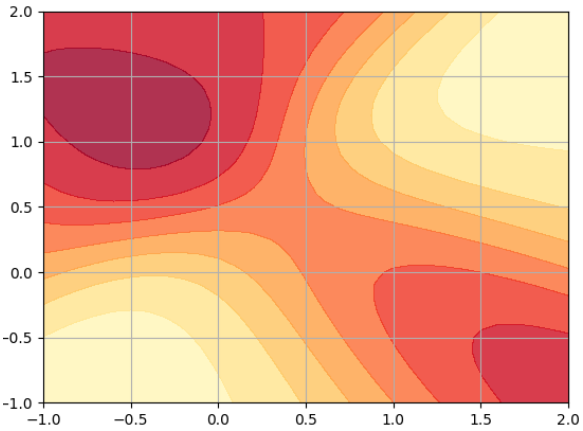
iteracion 70
error 0.42298848969127734



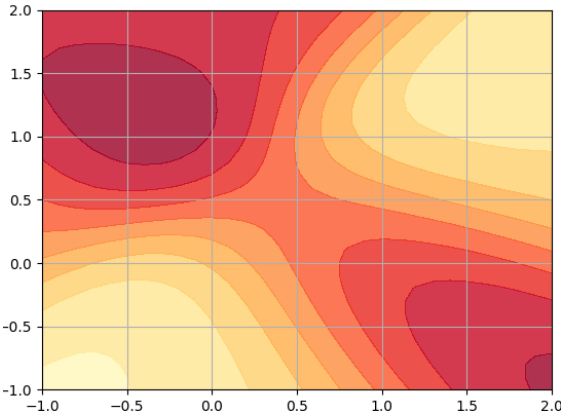

x=0.990 y=
iteracion 80
error 0.4004897291933489



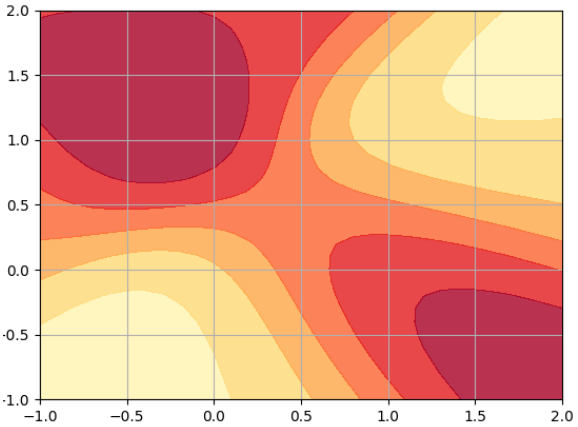

iteracion 90
error 0.3745152469029956



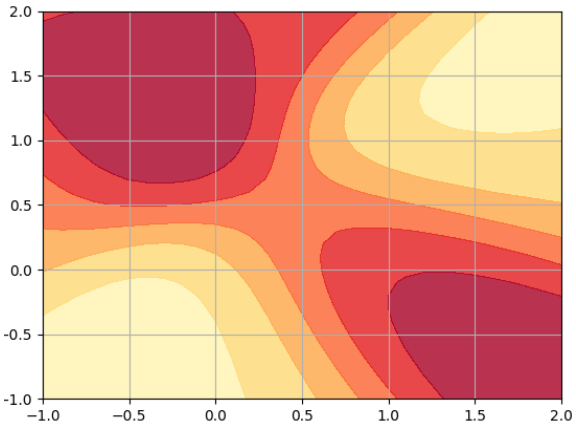

iteracion 100
error 0.3454819457090758



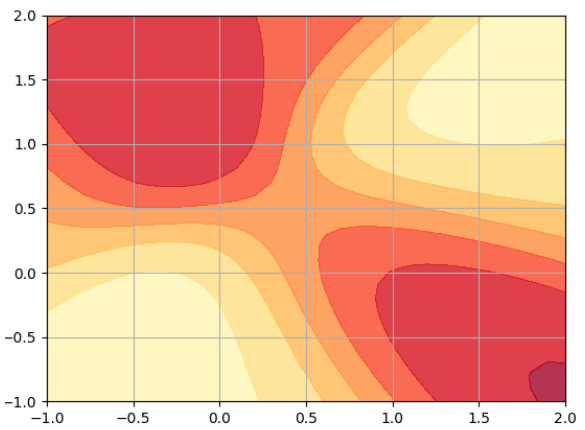

x=1.383 y
iteracion 110
error 0.3143467755232216

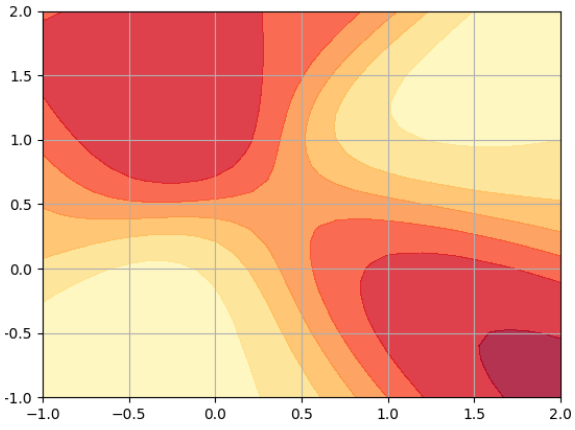


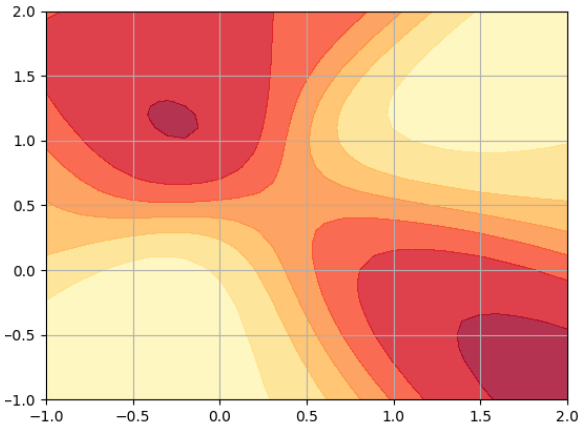

iteracion 120
error 0.28280865938802197

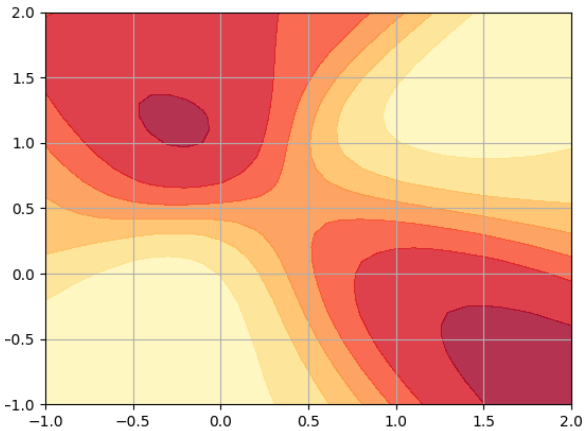


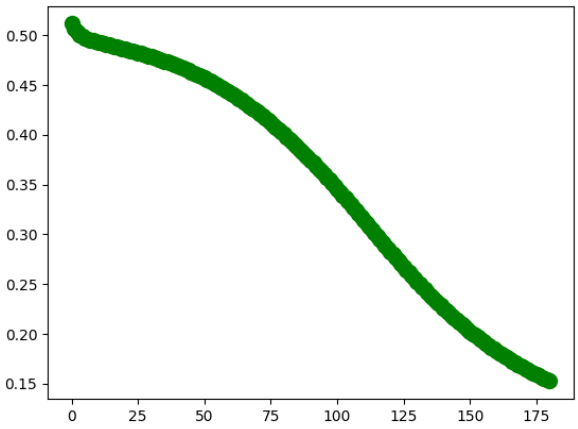

iteracion 130
error 0.2528479036266919











(graficación del error)

Código:

Main.py:

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

# Error graphing function.
def graphLearning(x_coordinate, y_coordinate):
    plt.plot(x_coordinate, y_coordinate)
    plt.pause(0.2)

def graphError(x_coordinate, y_coordinate):
    plt.plot(x_coordinate, y_coordinate, 'go', markersize=10)
    plt.pause(0.000000001)

def main():
    print("Seleccione la compuerta logica a trabajar:")
    print("1. XOR")
    print("2. XNOR")
    print("3. Salir")

    while True:
        try:
            option = int(input("Ingrese el número de la opción que desea:
"))
            if option == 1:
                logic_gate = "xor"
                trainingPatternsFileName = "entradas.csv"
                outputValuesFileName = "xor.csv"
                break
            elif option == 2:
                logic_gate = "xnor"
                trainingPatternsFileName = "entradas.csv"
                outputValuesFileName = "xnor.csv"
                break
            elif option == 3:
                print("¡Hasta luego!")
                return
            else:
                print("Opción inválida. Por favor ingrese una opción
válida.")
```

```
        except ValueError:
            print("Entrada inválida. Por favor ingrese un número.")


    # epochs = 10000
    epochs = 1000
    learning_rate = 0.3
    neurons_in_hidden_layer = 8

    file = open(trainingPatternsFileName)
    rows = len(file.readlines())
    file.close()

    file = open(trainingPatternsFileName,'r')
    reader = csv.reader(file,delimiter=',')
    entries = len(next(reader))
    file.close()

    file = open(outputValuesFileName,'r')
    reader = csv.reader(file,delimiter=',')
    output_layer_neurons = len(next(reader))
    file.close()

    net = RedMulticapa((entries, neurons_in_hidden_layer,
output_layer_neurons), ('tanh', 'sigmoid'))

    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)))

    plt.figure(1)
    if logic_gate == "xor":
        plt.title("XOR", fontsize=20)
        plt.plot(0,0,'r*')
        plt.plot(0,1,'k*')
        plt.plot(1,0,'k*')
```

```
        plt.plot(1,1,'r*')
        plt.pause(5)
    elif logic_gate == "xnor":
        plt.title("XNOR", fontsize=20)
        plt.plot(0,0,'k*')
        plt.plot(0,1,'r*')
        plt.plot(1,0,'r*')
        plt.plot(1,1,'k*')
        plt.pause(5)

    error_list = []

    for i in range(epochs):
        error = net.train(X, y, 1, learning_rate)
        error_list.append(error)
        if i%10 == 0:
            graphLearning(0,0)

            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.YlOrRd)

            plt.xlim([-1, 2])
            plt.ylim([-1, 2])
            plt.grid()
            plt.show()
            # plt.pause(2.5)
            # plt.close()
            print("iteracion", i)
            print("error ", error)

        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()
```

RedMulticapa.py

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

class RedMulticapa:
    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
```