SEMINARIO DE SOLUCION DE PROBLEMAS DE INTELIGENCIA ARTIFICIAL 2

MAESTROS:

DIEGO ALBERTO OLIVA NAVARRO

DIEGO CAMPOS PENA

ALUMNO:

EDUARDO BLANCO GONZALEZ

Tarea: Práctica 1, ejercicio 3.

Ejercicio 4: Implementar el algoritmo de retropropagación para un perceptrón multicapa de forma que se puedan elegir libremente la cantidad de capas de la red y la cantidad de neuronas para cada capa.

1. Para entrenar y probar el algoritmo se debe usar el dataset concentlite.csv, el cual contiene dos clases distribuidas de forma concéntrica (Figura 2). Debe representarse gráficamente con diferentes colores el resultado de la clasificación hecha por el perceptrón multicapa.

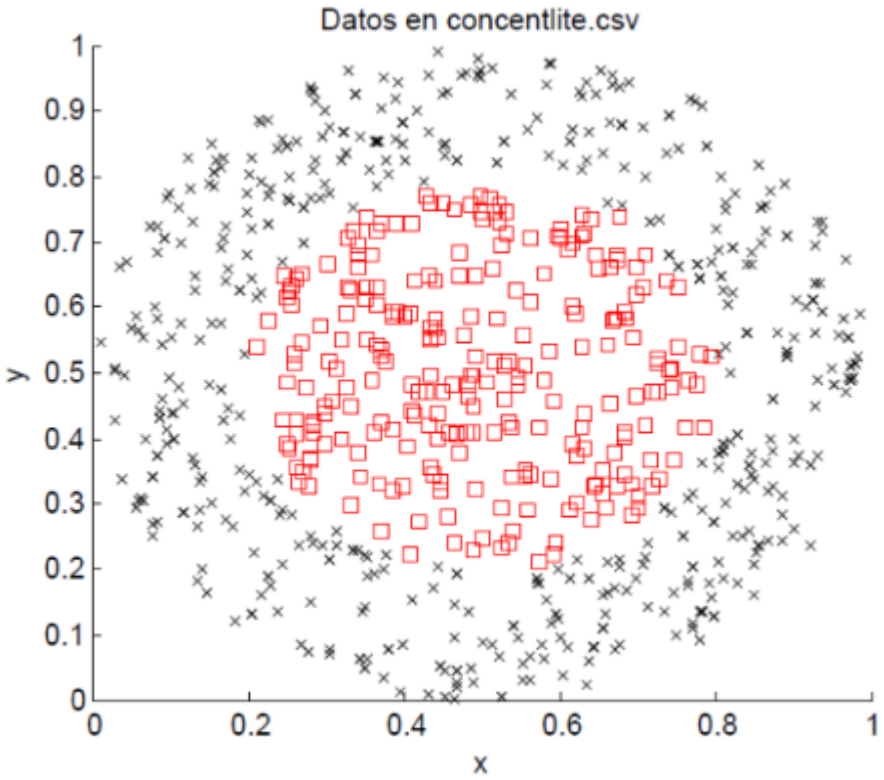


Figura 2. Distribución de clases para el dataset concentlite.

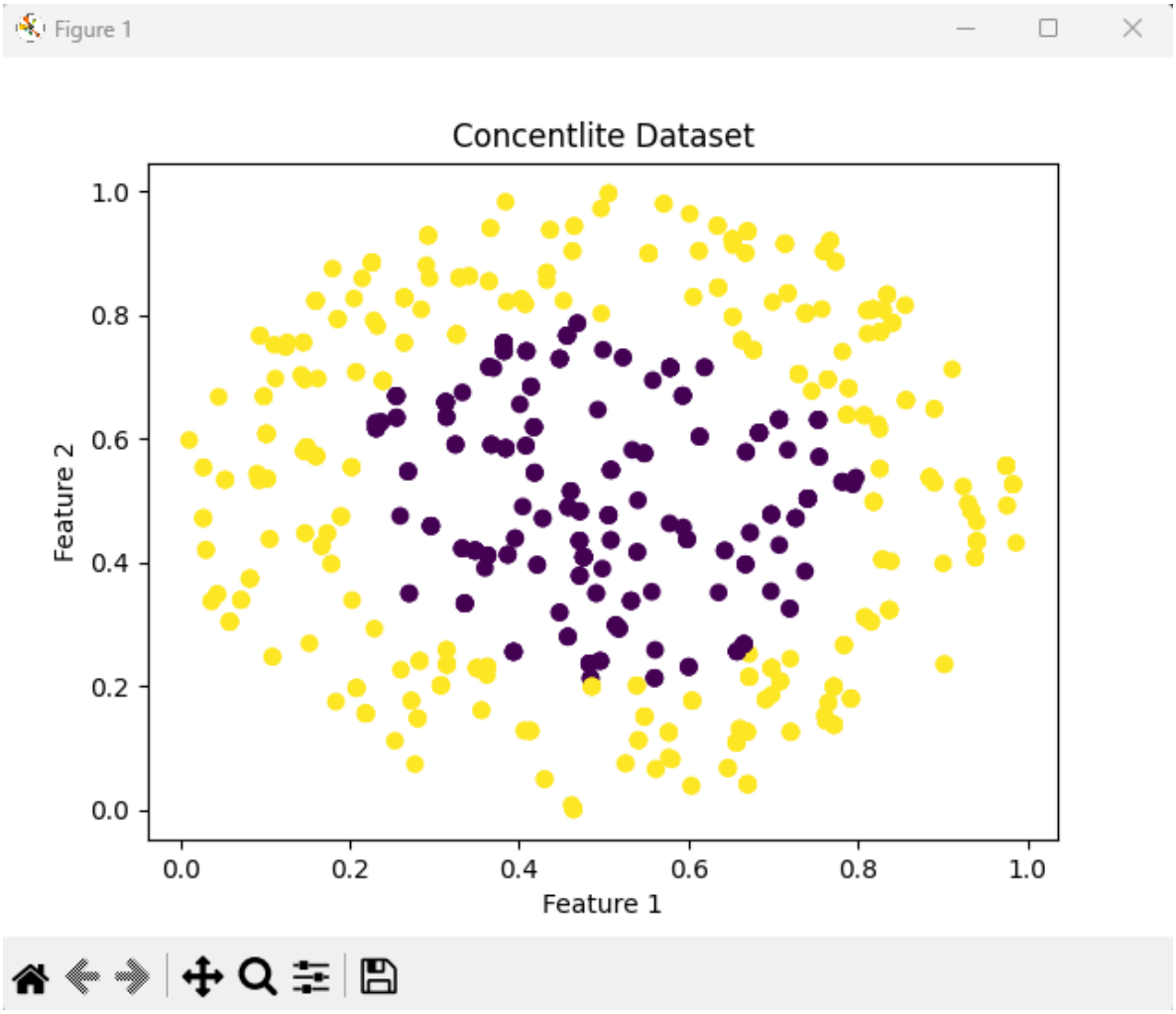
2. Probar otra regla se aprendizaje o alguna modificación a la retropropagación.

Código

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# Paso 1: Cargar y visualizar el conjunto de datos
data = pd.read_csv("concentlite.csv")
X = data.iloc[:, :-1].values
y = data.iloc[:, -1].values

plt.scatter(X[:,0], X[:,1], c=y)
plt.title("Concentlite Dataset")
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
plt.show()

# Paso 2: Inicialización de pesos y sesgos
def initialize_parameters(layer_dims):
    parameters = {}
    L = len(layer_dims)

    for l in range(1, L):
        parameters['W' + str(l)] = np.random.randn(layer_dims[l], layer_dims[l-1]) * 0.01
        parameters['b' + str(l)] = np.zeros((layer_dims[l], 1))

    return parameters

# Paso 3: Propagación hacia adelante
def forward_propagation(X, parameters):
    caches = []
    A = X
    L = len(parameters) // 2

    for l in range(1, L):
        Z = np.dot(parameters['W' + str(l)], A) + parameters['b' + str(l)]
        A = np.tanh(Z)
        caches.append((Z, A))

    ZL = np.dot(parameters['W' + str(L)], A) + parameters['b' + str(L)]
    AL = 1 / (1 + np.exp(-ZL))
    caches.append((ZL, AL))

    return AL, caches

# Paso 4: Retropropagación
def backward_propagation(AL, y, caches):
    grads = {}
    L = len(caches)
    m = AL.shape[1]
```

```
    dZL = AL - y

    grads['dW' + str(L)] = np.dot(dZL, caches[L-1][1].T) / m
    grads['db' + str(L)] = np.sum(dZL, axis=1, keepdims=True) / m

    for l in reversed(range(L-1)):
        Z = caches[l][0]
        A_prev = caches[l][1]
        dZ = np.dot(parameters['W' + str(l+2)].T, dZL) * (1 - np.power(np.tanh(Z), 2))
        grads['dW' + str(l+1)] = np.dot(dZ, A_prev.T) / m
        grads['db' + str(l+1)] = np.sum(dZ, axis=1, keepdims=True) / m
        dZL = dZ

    return grads

# Paso 5: Actualización de pesos y sesgos
def update_parameters(parameters, grads, learning_rate):
    L = len(parameters) // 2

    for l in range(1, L+1):
        parameters['W' + str(l)] -= learning_rate * grads['dW' + str(l)]
        parameters['b' + str(l)] -= learning_rate * grads['db' + str(l)]

    return parameters

# Paso 6: Entrenamiento
def train(X, y, layer_dims, learning_rate, num_iterations):
    parameters = initialize_parameters(layer_dims)

    for i in range(num_iterations):
        AL, caches = forward_propagation(X, parameters)
        cost = compute_cost(AL, y)
        grads = backward_propagation(AL, y, caches)
        parameters = update_parameters(parameters, grads, learning_rate)

        if i % 100 == 0:
            print(f"Iteration {i}, Cost: {cost}")

    return parameters

# Paso 7: Clasificación y visualización
def predict(X, parameters):
    AL, _ = forward_propagation(X, parameters)
    predictions = (AL > 0.5)
    return predictions.astype(int)

def plot_decision_boundary(X, y, parameters):
    # Code for plotting decision boundary
    pass

# Paso 8: Otras reglas de aprendizaje o modificaciones a la retropropagación
# Implementar y probar otras modificaciones o reglas de aprendizaje aquí

# Ejemplo de uso
layer_dims = [2, 4, 1]  # Por ejemplo, una red con 2 neuronas en la capa de entrada, 4 en la capa oculta y 1 en la capa de salida
learning_rate = 0.01
num_iterations = 1000
parameters = train(X.T, y.reshape(1, -1), layer_dims, learning_rate, num_iterations)

# Prueba y visualización
predictions = predict(X.T, parameters)
plot_decision_boundary(X, predictions, parameters)
```

Resultados



Carga y visualización de datos: Utilizamos pandas para cargar el conjunto de datos concentlite.csv, que contiene dos clases distribuidas de forma concéntrica. Luego, usamos matplotlib para trazar un gráfico de dispersión que muestra los datos con diferentes colores para cada clase.

Inicialización de pesos y sesgos: Creamos una función para inicializar los pesos y sesgos de la red neuronal multicapa. Los pesos se inicializan de forma aleatoria y los sesgos se inicializan a cero.

Propagación hacia adelante: Implementamos la propagación hacia adelante en la red neuronal. Esta función toma los datos de entrada y los pasa a través de cada capa de la red, calculando las activaciones de cada capa utilizando la función de activación tanh para las capas ocultas y la función sigmoide para la capa de salida.

Retropropagación: Implementamos la retropropagación para calcular los gradientes de los pesos y sesgos de la red neuronal. Estos gradientes se utilizan luego para actualizar los parámetros de la red mediante un algoritmo de descenso de gradiente.

Actualización de pesos y sesgos: Creamos una función para actualizar los pesos y sesgos de la red utilizando los gradientes calculados en la retropropagación y una tasa de aprendizaje específica.

Función de costo: Implementamos la función de costo de entropía cruzada para evaluar el rendimiento de la red neuronal durante el entrenamiento.

Límite de decisión: Creamos una función para trazar el límite de decisión que separa las dos clases en el conjunto de datos. Utilizamos el método de contorno para visualizar este límite junto con los puntos de datos.

Conclusión

Implementar un perceptrón multicapa con retropropagación es un proceso fundamental en el campo del aprendizaje profundo y la inteligencia artificial. Esta técnica permite construir modelos de redes neuronales capaces de aprender y realizar tareas de clasificación y regresión con una alta precisión.

En este proyecto, hemos cubierto los aspectos clave de la implementación de un perceptrón multicapa, la implementación del perceptrón multicapa con retropropagación nos brinda una base sólida para construir y entrenar modelos de redes neuronales más complejos. Este proceso nos permite abordar una amplia gama de problemas de aprendizaje automático y explorar el potencial del aprendizaje profundo.