INGENIERÍA EN SOFTWARE

POLITÉCNICA SANTA ROSA
Bilingual International Sustainable University

16-3-2025

IMPLEMENTACIÓN DE UNA RED NEURONAL CON DESCENSO DE GRADIANTE

Desarrollado con Python

Cálculo Vectorial

Ingeniería en Software

Apolo Reynoso Ruiz
Erick González Gómez
David Sosa López
Allan Roberto Ortiz Sánchez
Emiliano Pastrana Basaldúa
UPSRJ

Resumen

La presente investigación se enfoca en la implementación de una red neuronal artificial diseñada para clasificar conjuntos de datos mediante el empleo del método de descenso de gradiente. Nuestro objetivo fue desarrollar una herramienta de aprendizaje automático efectiva para manejar problemas de clasificación, aportando una comprensión práctica de los fundamentos teóricos de las redes neuronales y del algoritmo de aprendizaje sugerido. A través de iteraciones y análisis, se logró verificar la mejora en precisión, destacando así los beneficios del uso de este enfoque en tareas de este tipo.

Introducción

Las redes neuronales artificiales son un campo distintivo dentro de la inteligencia artificial que permite solucionar problemas complejos mediante la simulación del funcionamiento del cerebro humano. Nuestro proyecto busca implementar una red neuronal simple pero efectiva para su aplicación en clasificaciones, abordando problemas cuya solución más conservadora exigiría programación explícita y etiquetado detallado de casos especiales. Motivados por la capacidad de aprendizaje automático para optimizar tareas que involucran patrones y datos no claros, este proyecto proporciona una base sólida para el entendimiento y futuros desarrollos en aprendizaje profundo.

Fundamentos Teóricos

Las redes neuronales se componen de capas de neuronas interconectadas, con una capa de entrada que recibe datos, una capa de salida que produce los resultados, y potencialmente una o más capas ocultas para procesar la información. La activación de las neuronas se identifica comúnmente con funciones como la sigmoide, que permite la salida continua dentro de un rango definido.

Descenso de Gradiente

El descenso de gradiente es un algoritmo iterativo utilizado para optimizar funciones. En el contexto de las redes neuronales, se emplea para ajustar los pesos de las conexiones entre neuronas. Esta técnica sigue la regla que consiste en iterativamente mover el vector de pesos en la dirección negativa donde la función de pérdida

disminuye más rápido. Este movimiento se calcula como el producto de la derivada parcial de la función respecto a cada peso con un factor llamado tasa de aprendizaje, que controla la magnitud del movimiento.

El descenso de gradiente es un enfoque crítico del método de optimización utilizado en este proyecto. El objetivo del descenso de gradiente es iterativamente cambiar los pesos de la red de manera que se reduzca la pérdida de coste, encontrando así los mínimos de la función de coste. Este proceso difiere en complejidad dependiendo de la arquitectura de la red y el coste de pericia elegidos, pero en su esencia se basa en las siguientes operaciones:

- Procesar hacia adelante: Hay pasar el conjunto de datos a través de todos los nodos de la neurona, comienza con el cálculo de Z y A usando las ecuaciones y funciones mencionadas previamente.
- Calcular Coste: Utilizarse MSE para comparar la predicción generalizada con el resultado deseado.
- Volver hacia atrás: Invertir el procesamiento para retornar errores y gradientes hasta los pesos y bases, esto involucra el uso de la cadena de la regla de la derivación, invirtiendo las operaciones del modelado hacia delante.
- Actualización de parámetros: Ajustar los pesos y bases usando la fórmula:

$$W := W - \lambda \cdot dW$$

$$b := b - \lambda \cdot db$$

Donde λ es el factor de aprendizaje (learning rate). La elección de λ es crucial para la convergencia y evita problemas tales como oscilaciones o un ajuste excesivamente lento.

Desarrollo del Proyecto

El desarrollo de nuestro proyecto se centró en la implementación práctica de una red neuronal simple dotada con el algoritmo de descenso de gradiente para el entrenamiento. Este enfoque es fundamental en el campo del aprendizaje profundo, donde las técnicas permiten ajustar automáticamente los parámetros de un modelo a partir de un conjunto de datos de entrenamiento, minimizando una función de coste que representa la diferencia entre las predicciones del modelo y los valores

supervisados. Este método nos sirve no solo para resolver problemas de clasificación, sino también para entender cómo funcionan mecánicamente las redes neuronales en situaciones más complejas.

Arquitectura de la red

En las ecuaciones que manejamos, el cálculo vectorial juega un rol pivotal. La proyección de una matriz sobre un conjunto de variables de entrada (x) mediante la multiplicación vectorial permite someter los datos a transformaciones complejas a través de las conexiones neuronales. Esto se representa generalmente como:

$$Z = W \cdot X + b$$

Donde W es la matriz de pesos, X es el vector de entrada y b es el vector de sesgos. La expresión Z representa el resultante de la primera capa de procesamiento de la información antes de aplicar la función de activación.

Una red neuronal típica con una capa oculta tiene la siguiente estructura:

- Entrada (X): datos de entrada.
- Capa oculta: combinación lineal de las entradas seguida por una función de activación.
- Capa de salida: combinación lineal de las salidas de la capa oculta seguida por una función de activación final.

En nuestro caso:

- La entrada (X) es una matriz de tamaño $(nfeatures, m)$, donde:
  - $features$ = número de características de entrada.
  - m = número de ejemplos de entrenamiento.
- La **capa oculta** tiene n_hidden neuronas.
- La **capa de salida** tiene n_output neuronas (en este caso 1 para clasificación binaria).

Fórmulas clave

La entrada a la capa oculta: calculamos la combinación lineal de las entradas y los pesos:

$$Z1 = W1 \cdot X + b1$$

$W1 \rightarrow$ matriz de pesos de la capa de entrada a la capa oculta **(**dimensión: $nhidden \times nfeatures$).

$b1 \rightarrow$ vector de sesgos de la capa oculta (dimensión: $nhidden \times 1$).

$X \rightarrow$ datos de entrada (dimensión: $nfeatures \times m$).

Luego, pasamos el valor de Z1 por una función de activación (en este caso **tanh**):

$$A1 = tanh(Z1)$$

La función tanh (hiperbólica) está definida como:

$$\tanh(z) = ez + e - z$$

De la capa oculta a la capa de salida

Calculamos la combinación lineal de las salidas de la capa oculta y los pesos de la capa de salida:

$$Z2 = W2 \cdot A1 + b2$$

$W2 \rightarrow$ matriz de pesos de la capa oculta a la capa de salida (dimensión: $noutput \times nhidden$).

$b2 \rightarrow$ vector de sesgos de la capa de salida (dimensión: $noutput\ x1$).

La salida final pasa por la función **sigmoide** (porque estamos resolviendo un problema de clasificación binaria):

$$A2 = \sigma(Z2) = 11 + e - Z2$$

La función sigmoide transforma cualquier valor real en un rango entre 0 y 1, lo que permite interpretar el resultado como una probabilidad.

Por lo tanto:

- Si $A2$ es mayor a 0.5, la clase predicha es 1 (positivo).
- Si $A2$ es menor a 0.5, la clase predicha es 0 (negativo).

Coste de Pericia (MSE o Error Cuadrático Medio):

Para medir el rendimiento de la red y guiar su ajuste, empleamos el coste de pericia propuesto, el cual utiliza la pérdida cuadrática o error cuadrático medio (MSE):

$$MSE = m1\sum i = 1m(y(i) - y\hat{}(i))2$$

Donde m es el número de ejemplos de entrenamiento, $y(i)$ es el valor verdadero y $y\hat{}(i)$ es la predicción del modelo. Esta fórmula proporciona una métrica efectiva del error cometido por la red respecto a los datos de entrenamiento, facilitando así el proceso de refinamiento de la red.

Este ciclo iterativo entre procesar hacia adelante y hacia atrás con las actualizaciones de los parámetros, es el corazón del entrenamiento de redes neuronales y es lo que nos permite implementar una solución eficiente y adaptable al problema tratado.

Código

```
# Librerias a utilizar
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import accuracy_score, precision_score,
recall_score, f1_score

# Inicialización de pesos y sesgos

def initialize_parameters(n_input, n_hidden, n_output):

    np.random.seed(42) #Cambiar el valor de la semilla para
inicialización distinta

    w1 = np.random.randn(n_hidden, n_input)

    b1 = np.zeros((n_hidden, 1))

    w2 = np.random.randn(n_output, n_hidden)

    b2 = np.zeros((n_output, 1))

    return w1, b1, w2, b2

# Propagación hacia adelante
```

```python
def forward_propagation(X, W1, b1, W2, b2):
    Z1 = np.dot(W1, X) + b1
    A1 = np.tanh(Z1) # Función de activación en la capa oculta
    Z2 = np.dot(W2, A1) + b2
    A2 = 1 / (1 + np.exp(-Z2)) # Función sigmoide para la salida
    return Z1, A1, Z2, A2

# Función de error cuadrático medio (MSE)
def compute_cost(A2, Y):
    m = Y.shape[1]
    cost = (1 / (2 * m)) * np.sum((A2 - Y) ** 2)
    return cost

# Retropropagación
def backward_propagation(X, Y, Z1, A1, Z2, A2, W1, W2, b1, b2,
learning_rate):
    m = X.shape[1]

    dZ2 = A2 - Y
    dW2 = (1 / m) * np.dot(dZ2, A1.T)
    db2 = (1 / m) * np.sum(dZ2, axis=1, keepdims=True)

    dA1 = np.dot(W2.T, dZ2)
    dZ1 = dA1 * (1 - np.power(A1, 2))
    dW1 = (1 / m) * np.dot(dZ1, X.T)
    db1 = (1 / m) * np.sum(dZ1, axis=1, keepdims=True)

    W1 -= learning_rate * dW1
    b1 -= learning_rate * db1
    W2 -= learning_rate * dW2
    b2 -= learning_rate * db2

    return W1, b1, W2, b2

# Entrenamiento de la red
def train(X, Y, n_hidden, learning_rate, epochs):
    n_input = X.shape[0]
    n_output = Y.shape[0]
```

```
    W1, b1, W2, b2 = initialize_parameters(n_input, n_hidden,
n_output)
    costs = []

    # Función para calcular el costo en cada iteración
    def compute_cost(A2, Y):
        # Compute the cost using the mean squared error (MSE)
formula
        cost = np.mean((A2 - Y) ** 2)
        return cost

    for i in range(epochs):
        Z1, A1, Z2, A2 = forward_propagation(X, W1, b1, W2, b2)
        cost = compute_cost(A2, Y)
        costs.append(cost)
        W1, b1, W2, b2 = backward_propagation(X, Y, Z1, A1, Z2,
A2, W1, W2, b1, b2, learning_rate)
        if i % 100 == 0:
            print(f"Iteración {i}: Error = {cost:.4f}")

    return W1, b1, W2, b2, costs

# Visualización de resultados
def plot_cost(costs):
    plt.plot(costs)
    plt.xlabel('Iteraciones')
    plt.ylabel('Error (MSE)')
    plt.title('Progreso del entrenamiento')
    plt.show()

# Predicción
def predict(X, W1, b1, W2, b2):
    _, _, _, A2 = forward_propagation(X, W1, b1, W2, b2)
    predictions = (A2 > 0.5).astype(int)
    return predictions

# Datos de prueba
np.random.seed(1)
X = np.random.randn(2, 500)
```

```
Y = (X[0] * X[1] > 0).astype(int).reshape(1, 500)

# Parámetros
n_hidden = 5
learning_rate = 0.01
epochs = 1000

# Entrenar el modelo
W1, b1, W2, b2, costs = train(X, Y, n_hidden, learning_rate,
epochs)

# Graficar el error
plot_cost(costs)

# Prueba con datos nuevos
predictions = predict(X, W1, b1, W2, b2)
accuracy = np.mean(predictions == Y) * 100
print(f"Precisión: {accuracy:.2f}%")

# Visualización de la frontera de descisión
def plot_decision_boundary(X, Y, W1, b1, W2, b2):
    x_min, x_max = X[0, :].min() - 1, X[0, :].max() + 1
    y_min, y_max = X[1, :].min() - 1, X[1, :].max() + 1

    xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.01),
                         np.arange(y_min, y_max, 0.01))

    Z = predict(np.c_[xx.ravel(), yy.ravel()].T, W1, b1, W2, b2)
    Z = Z.reshape(xx.shape)

    plt.contourf(xx, yy, Z, alpha=0.7, cmap=plt.cm.Spectral)
    plt.scatter(X[0, :], X[1, :], c=Y.ravel(), edgecolors='k',
cmap=plt.cm.Spectral)
    plt.title('Frontera de Decisión')
    plt.show()

# Datos de prueba
np.random.seed(1)
X = np.random.randn(2, 500)
```

```python
Y = (X[0] * X[1] > 0).astype(int).reshape(1, 500)

# Parámetros
n_hidden = 5
learning_rate = 0.01
epochs = 1000

# Entrenar el modelo
W1, b1, W2, b2, costs = train(X, Y, n_hidden, learning_rate,
epochs)

# Graficar el error y la frontera de decisión
plot_cost(costs)
plot_decision_boundary(X, Y, W1, b1, W2, b2)

# Evaluación
predictions = predict(X, W1, b1, W2, b2)
accuracy = np.mean(predictions == Y) * 100
print(f"Precisión: {accuracy:.2f}%")

# Optimización de hiperparámetros
def optimize_hyperparameters(X, Y, hidden_neurons, learning_rates,
epochs_list):
    best_accuracy = 0
    best_params = {}

    for n_hidden in hidden_neurons:
        for lr in learning_rates:
            for epochs in epochs_list:
                W1, b1, W2, b2, _ = train(X, Y, n_hidden, lr,
epochs)
                predictions = predict(X, W1, b1, W2, b2)
                accuracy = np.mean(predictions == Y) * 100

                if accuracy > best_accuracy:
                    best_accuracy = accuracy
                    best_params = {
                        'n_hidden': n_hidden,
                        'learning_rate': lr,
```

```
                        'epochs': epochs
                    }

                print(f"Neurons: {n_hidden}, Learning Rate: {lr},
Epochs: {epochs}, Accuracy: {accuracy:.2f}%")

    print("\nMejores parámetros encontrados:")
    print(best_params)
    print(f"Mejor precisión: {best_accuracy:.2f}%")
    return best_params

hidden_neurons = [4, 8, 16]
learning_rates = [0.01, 0.05, 0.1]
epochs_list = [500, 1000, 2000]

best_params = optimize_hyperparameters(X, Y, hidden_neurons,
learning_rates, epochs_list)

# Predicción
def predict(X, W1, b1, W2, b2):
    _, _, _, A2 = forward_propagation(X, W1, b1, W2, b2)
    predictions = (A2 > 0.5).astype(int)
    return predictions

# Evaluación de métricas
def evaluate_model(Y_true, Y_pred):
    accuracy = accuracy_score(Y_true.flatten(), Y_pred.flatten())
    precision = precision_score(Y_true.flatten(),
Y_pred.flatten())
    recall = recall_score(Y_true.flatten(), Y_pred.flatten())
    f1 = f1_score(Y_true.flatten(), Y_pred.flatten())

    print(f"Precisión: {accuracy:.4f}")
    print(f"Precisión (Precision): {precision:.4f}")
    print(f"Sensibilidad (Recall): {recall:.4f}")
    print(f"F1-Score: {f1:.4f}")

# Gráfico de precisión
def plot_accuracy(accuracy_list):
```

```
    plt.plot(accuracy_list)
    plt.title('Precisión a lo largo de las épocas')
    plt.xlabel('Épocas')
    plt.ylabel('Precisión')
    plt.show()

from sklearn.datasets import make_moons

# Generamos un conjunto de datos de dos clases no lineales
X, Y = make_moons(n_samples=500, noise=0.2, random_state=42)
X = X.T
Y = Y.reshape(1, -1)

n_input = X.shape[0]
n_hidden = 8
n_output = 1
learning_rate = 0.01
epochs = 1000

W1, b1, W2, b2 = initialize_parameters(n_input, n_hidden,
n_output)

accuracy_list = []

for i in range(epochs):
    Z1, A1, Z2, A2 = forward_propagation(X, W1, b1, W2, b2)
    cost = compute_cost(A2, Y)
    W1, b1, W2, b2 = backward_propagation(X, Y, Z1, A1, Z2, A2,
W1, W2, b1, b2, learning_rate)

    if i % 10 == 0:
        predictions = predict(X, W1, b1, W2, b2)
        accuracy = np.mean(predictions == Y) * 100
        accuracy_list.append(accuracy)
        print(f"Iteración {i}: Error = {cost:.4f} | Precisión =
{accuracy:.2f}%")

# Métricas finales
predictions = predict(X, W1, b1, W2, b2)
```

```
evaluate_model(Y, predictions)

# Gráfico de precisión
plot_accuracy(accuracy_list)
```

Resultados y Análisis

Los resultados obtenidos evidenciaron una disminución sistemática del error cuadrado medio (MSE) a lo largo de las iteraciones de aprendizaje, indicando una mejora continua del modelo en su capacidad de clasificación. Hemos observado un balance entre la complejidad del modelo (a través de la cantidad de neuronas en la capa oculta) y la eficiencia del aprendizaje (ajustando la tasa de aprendizaje y el número de opciones).

Conclusiones y Aprendizajes

A través de este proyecto, se ha comprendido que el descenso de gradiente es efectivo en optimizar pesos en una red neuronal, mejorando con precisión con cada iteración. Los ajustes correctos en hiperparámetros tales como el tamaño de la capa oculta y la tasa de aprendizaje permiten un entrenamiento más eficiente y preciso. Este ejercicio proporciona herramientas para futuros proyectos de inteligencia artificial, resaltando la importancia del aprendizaje automático y el papel de las redes neuronales en su implementación.

Referencias

1. *introducción*. (s/f). Bookdown.org. Recuperado el 16 de marzo de 2025, de https://bookdown.org/victor_morales/TecnicasML/introducci%C3%B3n.html

2. Hay, D. B. (2007). Using concept maps to measure deep, surface and non-learning outcomes. *Studies in Higher Education*, *32*(1), 39–57. https://doi.org/10.1080/03075070601099432

3. LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep Learning. *Nature*, *521*(7553), 436–444.
4. Jordan, M., Kleinberg, J., & Schölkopf, B. (n.d.). *Information Science and Statistics*. https://www.microsoft.com/en-us/research/wp-content/uploads/2006/01/Bishop-Pattern-Recognition-and-Machine-Learning-2006.pdf

5. *Introducción a las redes neuronales*. (n.d.). YouTube. Retrieved March 16, 2025, from http://www.youtube.com/playlist?list=PLVo0_3_ZKpuBeGJQneleWT0vg4aPqTveF