Análisis y Evaluación de una Red Backpropagation: Resultados del Tercer Laboratorio

John Sebastián Galindo Hernández, Miguel Ángel Moreno Beltrán

***Abstract*—The objective of this lab is to analyze and evaluate the performance of a Backpropagation neural network in different classification and prediction tasks. In this experiment, various datasets with numerical, binary, and mixed inputs are used to test the network's adaptability. The results aim to assess the strengths and limitations of the backpropagation algorithm, focusing on convergence rates, error minimization, and the network's ability to generalize across different problem types.**

***Resumen*—El objetivo de este laboratorio es analizar y evaluar el rendimiento de una red neuronal entrenada mediante el algoritmo de retropropagación (Backpropagation) en diversas tareas de clasificación y predicción. En este experimento, se utilizan diferentes conjuntos de datos con entradas numéricas, binarias y mixtas para probar la adaptabilidad de la red. Los resultados tienen como objetivo evaluar las fortalezas y limitaciones del algoritmo de retropropagación, enfocándose en la tasa de convergencia, la minimización del error y la capacidad de la red para generalizar en distintos tipos de problemas.**

***Index Terms*—backpropagation, aprendizaje supervisado, redes neuronales, clasificación, predicción.**

I. Introducción

En este laboratorio, se aborda el estudio del algoritmo de retropropagación (Backpropagation), un modelo de aprendizaje supervisado utilizado en redes neuronales. El enfoque del experimento es evaluar cómo esta técnica maneja tareas de clasificación y predicción en diferentes tipos de entradas: numéricas, binarias y mixtas. A lo largo del análisis, se evaluará la capacidad de la red para minimizar el error, su tasa de convergencia durante el proceso de aprendizaje, y las ventajas y limitaciones del modelo en la generalización de los resultados obtenidos.

II. Fundamentos Teóricos

En esta sección se detallan los aspectos clave del algoritmo de retropropagación (Backpropagation), su funcionamiento y su relevancia en el entrenamiento de redes neuronales.

A. Introducción a la Retropropagación

El algoritmo de retropropagación fue formalizado en 1986 por Rumelhart, Hinton, y Williams, basándose en trabajos previos de Werbos y Parker. Este método permite que una red neuronal aprenda la asociación entre patrones de entrada y clases de salida a través de más capas de neuronas que el perceptrón [1]. La retropropagación se basa en la generalización de la regla delta y ha ampliado significativamente las aplicaciones de las redes neuronales. Su función objetivo es minimizar el error entre la salida deseada y la salida real mediante la actualización iterativa de los pesos.

B. Arquitectura de la Red Neuronal

Una red neuronal entrenada mediante retropropagación consta de una capa de entrada con n neuronas, una capa de salida con m neuronas y al menos una capa oculta. Cada neurona en una capa (excepto la de entrada) recibe entradas de todas las neuronas de la capa anterior y envía su salida a todas las neuronas de la siguiente capa [2]. En la Figura 1, se muestra la estructura general de una red neuronal multicapa.

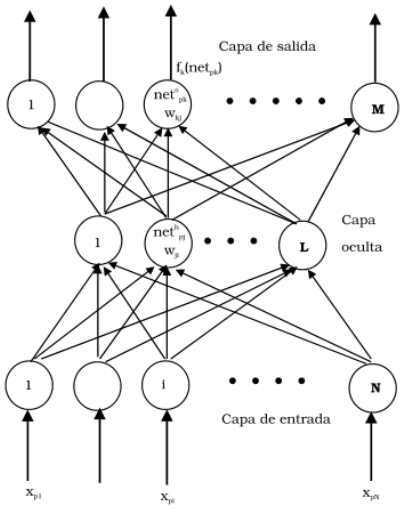

Figura 1. Arquitectura general de una red neuronal con una capa oculta [3].

C. Algoritmo de Retropropagación

El algoritmo de retropropagación opera en dos fases principales:

1) **Propagación hacia adelante**: Los datos de entrada se propagan a través de las capas de la red para generar una salida.
2) **Propagación hacia atrás**: El error se calcula comparando la salida obtenida con el valor deseado. Luego, el error se propaga hacia atrás para ajustar los pesos, comenzando desde la capa de salida hacia las capas anteriores.

El ajuste de los pesos se basa en el método del descenso del gradiente, que permite minimizar la función de costo a través de iteraciones sucesivas [4].

D. Función de Activación

En las redes neuronales entrenadas mediante retropropagación, las funciones de activación son cruciales para introducir

no linealidades y permitir que la red aprenda relaciones complejas. A continuación se presentan las funciones de activación utilizadas en los casos estudiados, junto con sus derivadas, que son esenciales para el ajuste de pesos mediante el descenso del gradiente.

- **Sigmoide**:

$$\sigma(x) = \frac{1}{1+e^{-x}}, \quad \sigma'(x) = \sigma(x)(1-\sigma(x)) \tag{1}$$

  Utilizada en capas ocultas, la sigmoide es adecuada para problemas de clasificación binaria, mapeando cualquier valor de entrada al rango entre 0 y 1. Su derivada permite calcular los gradientes durante la retropropagación.

- **Tanh**:

$$\tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}, \quad \tanh'(x) = 1-\tanh(x)^2 \tag{2}$$

  Similar a la sigmoide, pero mapea los valores al rango entre -1 y 1. Esto puede hacer que las capas ocultas aprendan de manera más eficiente, especialmente cuando los datos tienen entradas con valores negativos.

- **ReLU (Rectified Linear Unit)**:

$$f(x) = \text{máx}(0,x), \quad f'(x) = \begin{cases} 1 & \text{si } x > 0 \\ 0 & \text{si } x \leq 0 \end{cases} \tag{3}$$

  ReLU es una función muy popular en capas ocultas debido a su rápida convergencia. Al ser no lineal, pero simple en su cálculo, permite que las redes neuronales profundas aprendan sin sufrir tanto del problema de desaparición del gradiente.

- **Leaky ReLU**:

$$f(x) = \begin{cases} x & \text{si } x > 0 \\ \alpha x & \text{si } x \leq 0 \end{cases}, \quad f'(x) = \begin{cases} 1 & \text{si } x > 0 \\ \alpha & \text{si } x \leq 0 \end{cases} \tag{4}$$

  Es una variante de ReLU que introduce una pequeña pendiente para los valores negativos de x, lo que evita que las neuronas queden completamente inactivas.

- **Lineal**:

$$f(x) = x, \quad f'(x) = 1 \tag{5}$$

  Usada principalmente en capas de salida cuando se trata de problemas de regresión, la función lineal permite obtener una salida directamente proporcional a la entrada.

- **Softplus**:

$$f(x) = \log(1+e^{x}), \quad f'(x) = \frac{1}{1+e^{-x}} = \sigma(x) \tag{6}$$

  Es una versión suavizada de ReLU. Aunque no es tan popular como ReLU o tanh, se utiliza en algunos casos para evitar los problemas asociados a la activación nula de las neuronas.

E. Función de Costo

La función de costo utilizada en redes de retropropagación suele basarse en el Error Cuadrático Medio (MSE) o la entropía cruzada en problemas de clasificación. La función MSE se define como:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{7}$$

donde y_i es el valor esperado y $\hat{y}_i$ es la salida predicha [5].

F. Desafíos de la Retropropagación

Uno de los principales desafíos de la retropropagación es la **desaparición del gradiente**, donde los gradientes se vuelven demasiado pequeños en redes profundas, dificultando el ajuste adecuado de los pesos. Además, el **sobreajuste** puede ocurrir cuando la red aprende demasiado bien los datos de entrenamiento y no generaliza bien a nuevos datos [6].

G. Mejoras en la Retropropagación

Existen varias técnicas que mejoran el rendimiento de la retropropagación:

- **Momentum**: Ayuda a acelerar el proceso de aprendizaje y a evitar mínimos locales [7].
- **Regularización**: Técnicas como L2 o *dropout* ayudan a reducir el sobreajuste.
- **Algoritmos de optimización avanzados**: Algoritmos como Adam y RMSprop ajustan los pesos de manera más eficiente [8].

H. Aplicaciones de la Retropropagación

La retropropagación tiene una amplia gama de aplicaciones, entre las que destacan:

- **Reconocimiento de patrones**: Se utiliza para identificar formas y objetos en datos visuales [9].
- **Visión por computadora**: Procesamiento y análisis de imágenes.
- **Procesamiento de lenguaje natural**: Análisis de texto y reconocimiento de voz.

I. Comparación de Algoritmos de Aprendizaje Supervisado

En esta sección se presenta una comparación resumida entre el Perceptrón, ADALINE, MADALINE y las redes neuronales con retropropagación.

- **Perceptrón**: Es un modelo de red neuronal de capa única que solo puede resolver problemas de clasificación linealmente separables. Utiliza una función de activación escalonada y ajusta sus pesos cuando hay errores en la clasificación [10].
  - **Ventaja**: Simplicidad y rápida convergencia.
  - **Desventaja**: Incapaz de resolver problemas no lineales, como el XOR, sin capas ocultas.
- **ADALINE**: Similar al Perceptrón, pero utiliza una función de activación lineal y la regla del mínimo error cuadrático medio (MSE) para ajustar los pesos [11].
  - **Ventaja**: Maneja entradas continuas y minimiza el error de manera eficiente.
  - **Desventaja**: Tampoco puede resolver problemas no lineales sin capas ocultas.
- **MADALINE**: Extensión multicapa de ADALINE, capaz de resolver problemas no lineales gracias a su arquitectura con múltiples capas [12].
  - **Ventaja**: Puede resolver problemas no lineales.
  - **Desventaja**: Entrenamiento más complejo que ADALINE.

- **Redes Neuronales con Retropropagación**: A diferencia de los modelos anteriores, estas redes pueden resolver problemas no lineales utilizando múltiples capas y funciones de activación no lineales, ajustando los pesos mediante el descenso del gradiente [1].
  - **Ventaja**: Capacidad para resolver problemas complejos, no lineales.
  - **Desventaja**: Mayor tiempo de entrenamiento y riesgo de sobreajuste.

J. Algoritmo de Retropropagación

1) **Inicialización**
   - **Pesos y bias**:
     - Inicializa los pesos $\mathbf{w}_h$ y $\mathbf{w}_o$ para las conexiones de la capa de entrada a la capa oculta y de la capa oculta a la capa de salida, respectivamente. Los valores iniciales se seleccionan aleatoriamente entre $[0{,}1, 1]$.
     - Inicializa los bias b_h y b_o para las capas oculta y de salida.
   - **Tasa de aprendizaje** (α):
     - Establece α y otros hiperparámetros como el momentum (β) y el número de neuronas en la capa oculta.
   - **Normalización de entradas y salidas**:
     - Asegura que los valores de entrada estén en el rango $[0{,}001, 0{,}999]$ para evitar problemas de saturación.
2) **Entrenamiento**
   a) **Propagación hacia adelante**:
   - Calcula la activación de cada neurona en la capa oculta:

$$Neth_j = \sum_i x_i w_h(j,i) + b_h(j) \tag{8}$$

   - Aplica la función de activación seleccionada (f_h) para obtener la salida de la capa oculta:

$$Yh_j = f_h(Neth_j) \tag{9}$$

   - Calcula la activación de cada neurona en la capa de salida:

$$Neto_k = \sum_j Yh_j w_o(k,j) + b_o(k) \tag{10}$$

   - Aplica la función de activación en la capa de salida (f_o) para obtener la salida de la red:

$$Yk_k = f_o(Neto_k) \tag{11}$$

   b) **Cálculo del error**:
   - Calcula el error de la salida deseada d_k y la salida actual Yk_k:

$$e_k = d_k - Yk_k \tag{12}$$

   - Propaga el error hacia atrás utilizando la derivada de la función de activación:

$$\delta_o(k) = e_k \cdot f_o'(Neto_k) \tag{13}$$

   c) **Retropropagación del error a la capa oculta**:
   - Calcula el error en cada neurona de la capa oculta:

$$\delta_h(j) = f_h'(Neth_j) \cdot \sum_k \delta_o(k) w_o(k,j) \tag{14}$$

   d) **Actualización de pesos y bias**:
   - Actualiza los pesos de la capa de salida:

$$w_o(k,j) \leftarrow w_o(k,j) + \alpha \cdot \delta_o(k) \cdot Yh_j + \beta \tag{15}$$

   - Actualiza los pesos de la capa oculta:

$$w_h(j,i) \leftarrow w_h(j,i) + \alpha \cdot \delta_h(j) \cdot x_i + \beta \tag{16}$$

   - Actualiza los bias en ambas capas:

$$b_o(k) \leftarrow b_o(k) + \alpha \cdot \delta_o(k) \tag{17}$$

$$b_h(j) \leftarrow b_h(j) + \alpha \cdot \sum_i \delta_h(j) \tag{18}$$

   e) **Cálculo del error total**:
   - El error total para el patrón p se calcula como:

$$E_p = \frac{1}{2} \sum_k (d_k - Yk_k)^2 \tag{19}$$

   - Si el error es mayor al umbral de precisión, se continúa con otra época.
3) **Condición de parada**:
   - El proceso de entrenamiento se detiene cuando el error total es menor al umbral de precisión o se alcanza el número máximo de épocas.

III. Metodología

A. Conjunto de Datos

En este experimento, se utilizaron varios conjuntos de datos para abordar problemas de clasificación y predicción, cada uno con diferentes características en términos de entradas y salidas. Los casos se diseñaron para evaluar el rendimiento del algoritmo de retropropagación en redes neuronales multicapa.

- **Caso 0**: Clasificación de dos entradas numéricas con una salida de tres posibles valores (0, 1, -1). Este caso sirvió para una introducción básica al uso de redes neuronales para clasificación.
- **Caso 1**: Clasificación de cuatro entradas binarias con una salida binaria. Este caso permitió la manipulación de los hiperparámetros α (tasa de aprendizaje) y β (momentum) para observar su efecto en el aprendizaje.
- **Caso 2**: Predicción del resultado de la ecuación $\sin(A) + \cos(B) + C = Y$, utilizando tres entradas numéricas y una salida continua. Se generaron 100 patrones de entrenamiento con valores aleatorios para evaluar la capacidad de generalización del modelo.
- **Caso 3**: Predicción de resultados de una lotería utilizando cinco entradas numéricas (día, mes, A, B, C) y 10 salidas con valores 0, 0.5 y 1. Este caso permitió probar la robustez del modelo ante datos ruidosos y aleatorios.
- **Caso 4**: Clasificación de letras a partir de 35 entradas binarias, que representan una matriz de píxeles para formar

letras. La salida es un valor numérico correspondiente al código ASCII de la letra.

Cada caso tiene un conjunto de entradas y salidas específicas, y los datos fueron preparados en archivos JSON, lo que facilitó su carga y manipulación. Esta estructura permite personalizar las pruebas y ajustarlas a las necesidades específicas del experimento.

B. Implementación del Algoritmo

En el desarrollo del algoritmo de retropropagación, se utilizaron varias funciones de activación, como ReLU, Sigmoide, Tanh y Softplus. Se realizaron pruebas combinatorias de estas funciones para determinar cuál de ellas ofrecía un mejor rendimiento en el problema en cuestión. Además, se implementó un ajuste dinámico de la cantidad de neuronas en la capa oculta, incrementándolas cuando fue necesario para mejorar la capacidad de aprendizaje y minimizar el error durante el entrenamiento del modelo.

A continuación, se muestra el código fuente de la implementación en Python:

```
def backpropagation_training(train_data = None,
↪  epoch_label=None, total_error_label=None,
↪  labels_error=None, status_label=None,
↪  download_weights_btn = None, download_training_data_btn
↪  = None, results_btn = None):
    from app import update_training_process,
    ↪  changue_status_training
    # Json para guardar los datos del entrenamiento
    global weights_json, graph_json, stop_training

    # Booleano para saber si se debe aumentar la cantidad de
    ↪  neuronas en la capa oculta
    aumentar_neuronas = False

    # Inicializar listas para almacenar pesos, bias y
    ↪  errores
    bias_h = []
    bias_o = []
    pesos_h_registro = []
    pesos_o_registro = []
    bias_h_registro = []
    bias_o_registro = []
    pesos_h_momentum = []
    pesos_o_momentum = []
    errores_totales = []
    errores_patrones = []
    errores_patrones_registro = []

    funcion_h_nombre = train_data["function_h_name"]
    funcion_o_nombre = train_data["function_o_name"]
    neuronas_h_cnt = train_data["qty_neurons"]
    alpha = train_data["alpha"]
    precision = train_data["precision"]
    iteraciones_max = train_data["max_epochs"]
    MOMENTUM = train_data["momentum"]
    beta = train_data["betha"]
    bias = train_data["bias"]
    entradas = train_data["inputs"]
    salidas_d = train_data["outputs"]

    # Obtener la cantidad de neuronas de salida
    neuronas_o_cnt = len(salidas_d[0])

    # Obtener la función para la capa oculta (h) y la capa
    ↪  de salida
    # También obtener las funciones para la derivada
    funcion_h = switch_function_output(funcion_h_nombre)
    funcion_o = switch_function_output(funcion_o_nombre)
    funcion_h_derivada =
    ↪  switch_function_output(funcion_h_nombre, True)
    funcion_o_derivada =
    ↪  switch_function_output(funcion_o_nombre, True)

    # Normalizar las entradas y las salidas
    entradas, salidas_d, *_ = normalize_data(entradas,
    ↪  salidas_d)

    # Verificar si alguna entrada tiene valor de 1 y volver
    ↪  su valor a 0.999
    # Verificar si alguna entrada tiene valor de 0 y volver
    ↪  su valor a 0.001
    for i in range(len(entradas)):
        for j in range(len(entradas[i])):
            if entradas[i][j] == 1:
                entradas[i][j] = 0.999
            elif entradas[i][j] == 0:
                entradas[i][j] = 0.001

    # Crear los pesos para las conexiones entre entrada y
    ↪  capa oculta
    # Cada neurona tiene una cantidad de pesos igual a la
    ↪  cantidad de entradas
    # También se calcula el bias aparte en otra lista
    pesos_h = [[random.uniform(0.1, 1) for i in
    ↪  range(len(entradas[0]))] for j in
    ↪  range(neuronas_h_cnt)]
    if bias != 0:
        bias_h = [bias for i in range(neuronas_h_cnt)]
    else:
        bias_h = [random.uniform(0.1, 1) for i in
        ↪  range(neuronas_h_cnt)]

    # Crear los pesos para las conexiones entre capa oculta
    ↪  y salida
    # Cada neurona tiene una cantidad de pesos igual a la
    ↪  cantidad de neuronas en la capa oculta
    # También se calcula el bias aparte en otra lista
    pesos_o = [[random.uniform(0.1, 1) for i in
    ↪  range(neuronas_h_cnt)] for j in
    ↪  range(neuronas_o_cnt)]
    if bias != 0:
        bias_o = [bias for i in range(neuronas_o_cnt)]
    else:
        bias_o = [random.uniform(0.1, 1) for i in
        ↪  range(neuronas_o_cnt)]

    # Bucle While que se ejecuta hasta que todos los errores
    ↪  de patrones sean menores a la precisión
    # o hasta que se llegue al máximo de épocas
    epoca = 0
    delta_o = []

    # Valor B para calcular el momentum
    B = beta
    momentum = 0

    # Inicializar los errores de los patrones en un valor
    ↪  igual a la precisión + 5
    for i in range(len(entradas)):
        errores_patrones.append(precision + 0.9)

    while any([error > precision for error in
    ↪  errores_patrones]):

        if epoca % 1000 == 0:
            # Guardar los pesos iniciales en los registros
            pesos_h_registro.append([row[:] for row in
            ↪  pesos_h])
            pesos_o_registro.append([row[:] for row in
            ↪  pesos_o])
            bias_h_registro.append(bias_h[:])
            bias_o_registro.append(bias_o[:])

            ↪  errores_patrones_registro.append(errores_patrones[:])
            error_total = sum(errores_patrones) /
            ↪  len(entradas)
            errores_totales.append(error_total)

            print("Epoca: ", epoca, "Error Total: ",
            ↪  error_total)
            for i in range(len(entradas)):
                if errores_patrones[i] < precision and i % 4
                ↪  != 0:
                    # imprimir con solo 10 decimales y color
                    ↪  verde
                    print("\033[32m#", i, "|E| ",
                    ↪  "{:.10f}".format(errores_patrones[i]),
                    ↪  "\033[0m", end=" ")
                elif errores_patrones[i] <= precision and i
                ↪  % 4 == 0:
```

```
            print("\033[32m#", i, "|E| ",
            ↪  "{:.10f}".format(errores_patrones[i]),
            ↪  "\033[0m")
        elif errores_patrones[i] > precision and i %
        ↪  4 != 0:
            print("\033[91m#", i, "|E| ",
            ↪  "{:.10f}".format(errores_patrones[i]),
            ↪  "\033[0m", end=" ")
        else:
            print("\033[91m#", i, "|E| ",
            ↪  "{:.10f}".format(errores_patrones[i]),
            ↪  "\033[0m")


    update_training_process(epoca, errores_patrones,
    ↪  error_total, precision, epoch_label,
    ↪  total_error_label, labels_error)

# Empezar el recorrido de los patrones
for p in range(len(entradas)):

    if MOMENTUM:
        # Eliminar los pesos anteriores para
        ↪  mantener la lista con solo 1 elemento
        pesos_h_momentum = pesos_h_momentum[-1:]
        pesos_o_momentum = pesos_o_momentum[-1:]
        # Guardar los pesos iniciales en las listas
        ↪  del momentum
        pesos_h_momentum.append([row[:] for row in
        ↪  pesos_h])
        pesos_o_momentum.append([row[:] for row in
        ↪  pesos_o])

    # Sacar las entradas en ese patrón
    x = entradas[p]

    # Sacar las salidas deseadas en ese patrón
    yd = salidas_d[p]

    # Inicializar Neth y Yh
    Nethj = [0 for i in range(neuronas_h_cnt)]
    Yh = [0 for i in range(neuronas_h_cnt)]

    # Realizar la sumatoria entre las entradas y los
    ↪  pesos de la capa oculta
    # Net^h(p,j)=Net^h(p,j)+X(p,i)*Wh(j,i)
    # A esa sumatoria se le suma el bias de la capa
    ↪  oculta
    # Net^h(p,j)=Net^h(p,j)+ Th(j,0)
    for j in range(neuronas_h_cnt):
        for i in range(len(x)):
            Nethj[j] += x[i] * pesos_h[j][i]

        Nethj[j] += bias_h[j]
        # Después se realiza la salida de la capa
        ↪  oculta
        # Yh(p,j)=Funcion_activacion_h(Neth(p,j))
        Yh[j] = funcion_h(Nethj[j])

    # Inicializar Net^o(p,k) y Yk(p,k)
    Netok = [0 for i in range(neuronas_o_cnt)]
    Yk = [0 for i in range(neuronas_o_cnt)]

    # Realizar la sumatoria entre las salidas de la
    ↪  capa oculta y los pesos de la capa de salida
    # Net^o(p,k)=Neto(p,k)+Yh(p,j)*Wo(k,j)
    # A esa sumatoria se le suma el bias de la capa
    ↪  de salida
    # Net^o(p,k)=Neto(p,k)+To(k,0)
    for k in range(neuronas_o_cnt):
        for j in range(neuronas_h_cnt):
            Netok[k] += Yh[j] * pesos_o[k][j]

        Netok[k] += bias_o[k]
        # Después se realiza la salida de la capa de
        ↪  salida
        # Yk(p,k)=Funcion_activacion_o(Neto(p,k))
        Yk[k] = funcion_o(Netok[k])

    # Inicializar la lista de los errores de salida
    delta_o = [0 for i in range(neuronas_o_cnt)]

    # Calcular el error de salida
    # ^o = (d(p,k) - Yk(p,k)) *
    ↪  Funcion_derivada_o(Neto(p,k))
    for k in range(neuronas_o_cnt):
        delta_o[k] = (salidas_d[p][k] - Yk[k]) *
        ↪  funcion_o_derivada(Yk[k])

    # Inicializar la matriz de los errores de la
    ↪  capa oculta
    delta_h = [[0 for i in range(len(x))] for j in
    ↪  range(neuronas_h_cnt)]

    # Calcular el error de la capa oculta
    # ^h(p,j) = Funcion_derivada_h(Neth(p,j)) *
    ↪  (^o(p,k) * Wo(k,j))
    # primero se calcula la sumatoria en una
    ↪  variable llamada backpropagation
    # Después se calcula el error de la capa oculta
    for j in range(neuronas_h_cnt):
        backpropagation = 0
        for k in range(neuronas_o_cnt):
            backpropagation += delta_o[k] *
            ↪  pesos_o[k][j]

        for i in range(len(x)):
            delta_h[j][i] =
            ↪  funcion_h_derivada(Nethj[j]) *
            ↪  backpropagation

    # Actualizar los pesos de la capa de salida
    # W^o_(k,j)(t + 1) = W^o_(k,j)(t) +  * ^o(p,k) *
    ↪  Yh(p,j) + momentum
    for k in range(neuronas_o_cnt):
        for j in range(neuronas_h_cnt):
            # Calcular el momentum B * (w^o(t) -
            ↪  w^o(t - 1))
            momentum = B * (pesos_o[k][j] -
            ↪  pesos_o_momentum[-1][k][j]) if
            ↪  MOMENTUM else 0
            # Calcular el nuevo peso
            pesos_o[k][j] += alpha * delta_o[k] *
            ↪  Yh[j]
            pesos_o[k][j] += momentum

        # Actualizar el bias de la capa de salida
        # Cálculo del momentum para el bias
        # To(k,0)(t + 1) = To(k,0)(t) +  * ^o(p,k) +
        ↪  momentum
        bias_o[k] += alpha * delta_o[k]

    # Actualizar los pesos de la capa oculta
    # W^h_(j,i)(t + 1) = W^h_(j,i)(t) +  * ^h(p,j,i)
    ↪  * X(p,i) + momentum
    for j in range(neuronas_h_cnt):
        for i in range(len(x)):
            # Calcular el momentum B * (w^h(t) -
            ↪  w^h(t - 1))
            momentum = B * (pesos_h[j][i] -
            ↪  pesos_h_momentum[-1][j][i]) if
            ↪  MOMENTUM else 0
            # Calcular el nuevo peso
            pesos_h[j][i] += alpha * delta_h[j][i] *
            ↪  x[i]
            pesos_h[j][i] += momentum

        # Actualizar el bias de la capa oculta
        # Cálculo del momentum para el bias
        # Th(j,0)(t + 1) = Th(j,0)(t) +  *
        ↪  (^h(p,j,i)/len(^h(p,j,i))) + momentum
        bias_h[j] += alpha * sum(delta_h[j]) /
        ↪  len(delta_h[j])

    # Calcular el error total del patrón
    # E^p = 1/2 * (d(p,k) - Yk(p,k))^2
    error_patron = 0.5 * sum((yd[k] - Yk[k]) ** 2
    ↪  for k in range(neuronas_o_cnt))

    # Guardar el error del patrón
    errores_patrones[p] = error_patron

error_total = sum(errores_patrones) / len(entradas)

epoca += 1

if epoca >= iteraciones_max:
    changue_status_training(status_label, "Límite de
    ↪  épocas alcanzado", "red",
    ↪  download_weights_btn,
    ↪  download_training_data_btn, results_btn)
    return
```

```
        if stop_training:
            changue_status_training(status_label,
            ↪  "Entrenamiento detenido", "red",
            ↪  download_weights_btn,
            ↪  download_training_data_btn, results_btn)
            return


    errores_totales.append(error_total)
    pesos_h_registro.append([row[:] for row in pesos_h])
    pesos_o_registro.append([row[:] for row in pesos_o])
    bias_h_registro.append(bias_h[:])
    bias_o_registro.append(bias_o[:])
    errores_patrones_registro.append(errores_patrones[:])

    update_training_process(epoca, errores_patrones,
    ↪  error_total, precision, epoch_label,
    ↪  total_error_label, labels_error)

    graph_json = {
        "training_date":
        ↪  datetime.datetime.now().strftime("%Y-%m-%d
        ↪  %H:%M:%S"),
        "epochs": epoca,
        "max_epochs": iteraciones_max,
        "initial_bias": bias,
        "qty_neurons": neuronas_h_cnt,
        "arquitecture": [len(entradas[0]), neuronas_h_cnt,
        ↪  len(salidas_d[0])],
        "function_h": funcion_h_nombre,
        "function_o": funcion_o_nombre,
        "momentum": MOMENTUM,
        "b": beta,
        "alpha": alpha,
        "precision": precision,
        "totals_errors": errores_totales,
        "patterns_errors": errores_patrones_registro,
        "weights_h": pesos_h_registro,
        "weights_o": pesos_o_registro,
        "bias_h": bias_h_registro,
        "bias_o": bias_o_registro
    }

    weights_json = {
            "weights_h": pesos_h,
            "weights_o": pesos_o,
            "bias_h": bias_h,
            "bias_o": bias_o,
            "qty_neurons": neuronas_h_cnt,
            "function_h_name": funcion_h_nombre,
            "function_o_name": funcion_o_nombre
        }

    if not stop_training:
        changue_status_training(status_label, "Entrenamiento
        ↪  finalizado", "green", download_weights_btn,
        ↪  download_training_data_btn, results_btn)
```

```
def test_neural_network(test_data):

    entradas = test_data["inputs"]
    salidas = test_data["outputs"]
    pesos_h = test_data["weights_h"]
    pesos_o = test_data["weights_o"]
    bias_h = test_data["bias_h"]
    bias_o = test_data["bias_o"]
    funcion_h_nombre = test_data["function_h_name"]
    funcion_o_nombre = test_data["function_o_name"]
    neuronas_h_cnt = test_data["qty_neurons"]


    # Obtener la cantidad de neuronas de salida
    neuronas_o_cnt = len(salidas[0])

    # Obtener la funcion para la capa oculta (h) y la capa
    ↪  de salida
    # También obtener las funciones para la derivada

    funcion_h = switch_function_output(funcion_h_nombre)
    funcion_o = switch_function_output(funcion_o_nombre)

    # Normalizar las entradas y las salidas
    entradas_n, salidas_n, maximo_entrada, minimo_entrada,
    ↪  maximo_salida, minimo_salida =
    ↪  normalize_data(entradas, salidas)

    # Verificar si alguna entrada tiene valor de 1 y volver
    ↪  su valor a 0.999
    # Verificar si alguna entrada tiene valor de 0 y volver
    ↪  su valor a 0.001
    for i in range(len(entradas_n)):
        for j in range(len(entradas_n[i])):
            if entradas_n[i][j] == 1:
                entradas_n[i][j] = 0.999
            elif entradas_n[i][j] == 0:
                entradas_n[i][j] = 0.001

    # Realizar la suma de las entradas y los pesos de la
    ↪  capa oculta
    # Despues se le suma el bias de la capa oculta
    # Realizar la salida de la capa oculta
    # Realizar la suma de las salidas de la capa oculta y
    ↪  los pesos de la capa de salida
    # Despues se le suma el bias de la capa de salida
    # Realizar la salida de la capa de salida
    # Guardar el resultado en una lista
    # Imprimir el resultado de forma -> Patron #, Entradas,
    ↪  Salidas deseadas, Salidas obtenidas
    Y_resultados = []
    for p in range(len(entradas_n)):
        x = entradas_n[p]
        Nethj = [0 for i in range(neuronas_h_cnt)]
        Yh = [0 for i in range(neuronas_h_cnt)]

        for j in range(neuronas_h_cnt):
            for i in range(len(x)):
                Nethj[j] += x[i] * pesos_h[j][i]

            Nethj[j] += bias_h[j]
            Yh[j] = funcion_h(Nethj[j])

        Netok = [0 for i in range(neuronas_o_cnt)]
        Yk = [0 for i in range(neuronas_o_cnt)]

        for k in range(neuronas_o_cnt):
            for j in range(neuronas_h_cnt):
                Netok[k] += Yh[j] * pesos_o[k][j]

            Netok[k] += bias_o[k]
            Yk[k] = funcion_o(Netok[k])

        Y_resultados.append(Yk)

    Y_resultados = [[(Y_resultados[i][j] * (maximo_salida -
    ↪  minimo_salida)) + minimo_salida for j in
    ↪  range(neuronas_o_cnt)] for i in
    ↪  range(len(Y_resultados))]
    errores = []

    return Y_resultados, errores
```

C. Entrenamiento del Modelo

El entrenamiento del modelo se realizó mediante una interfaz gráfica de usuario (ver Figura 2). La interfaz permite configurar una serie de parámetros cruciales que afectan el comportamiento y la eficiencia del modelo de retropropagación. Estos parámetros incluyen:

- **Tasa de aprendizaje** (α): Controla la magnitud de los ajustes en los pesos en cada iteración, afectando la velocidad de convergencia.
- **Bias** (θ): Define el umbral de activación de las neuronas en las capas ocultas. El valor puede ser fijo o aleatorio.
- **Momentum** (β): Ayuda a evitar oscilaciones durante el ajuste de los pesos y acelera el proceso de convergencia, manteniendo un historial del ajuste de pesos previo.
- **Número de neuronas en la capa oculta (H)**: Permite incrementar el número de neuronas para mejorar la capacidad del modelo de aprender patrones complejos.
- **Número de épocas**: Controla la cantidad de iteraciones en las que el modelo ajusta los pesos.

- **Precisión** (ϵ): Criterio de parada basado en la reducción del error. El entrenamiento se detiene si el error cae por debajo de este valor antes de alcanzar el número máximo de épocas.
- **Funciones de activación**: Permite seleccionar funciones de activación como Tangente Hiperbólica, ReLU, o Sigmoide para las capas oculta y de salida, lo que afecta la no linealidad del modelo.

El usuario debe cargar los valores de entrada y salida en la aplicación, utilizando un archivo JSON predefinido. A lo largo del proceso de entrenamiento, los pesos de la red neuronal se ajustan mediante el algoritmo de retropropagación para minimizar el error cuadrático medio (**MSE**), asegurando que el modelo pueda aprender correctamente los patrones del conjunto de entrenamiento.

Durante las pruebas realizadas, se configuraron diferentes combinaciones de las funciones de activación y la cantidad de neuronas en la capa oculta, lo que permitió identificar la configuración óptima que ofrecía los mejores resultados en términos de precisión y convergencia.

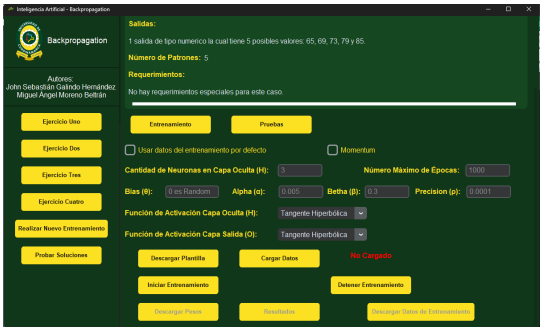

Figura 2. Interfaz de Entrenamiento del Modelo

D. Ejecución del Modelo

Una vez entrenado, el modelo puede ser probado utilizando la sección de **Probar soluciones** de la interfaz gráfica (ver Figura 3). En esta sección, el usuario puede cargar tanto los pesos generados durante el entrenamiento como un nuevo conjunto de entradas para evaluar el rendimiento del modelo en situaciones no vistas durante el entrenamiento.

Además, la aplicación permite cargar conjuntos de datos externos para realizar pruebas personalizadas y medir el error final utilizando métricas como el MSE.

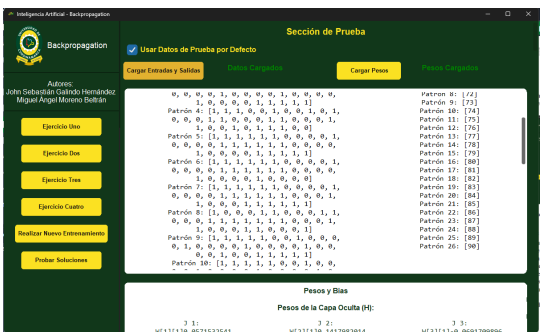

Figura 3. Interfaz para la Ejecución del Modelo

E. Casos de Estudio

Se diseñaron varios casos de estudio para evaluar diferentes aspectos del algoritmo de retropropagación. A continuación se presenta un resumen de los casos abordados en este experimento:

- **Caso 0**: Un caso básico de clasificación con dos entradas numéricas y una salida con tres posibles valores. Este caso se utilizó para validar el correcto funcionamiento del algoritmo en problemas simples.
- **Caso 1**: Un problema de clasificación binaria con cuatro entradas binarias y una salida. Se experimentó con diferentes valores de α y β para observar su efecto en el aprendizaje del modelo.
- **Caso 2**: Un problema de predicción continua basado en la ecuación $\sin(A)+\cos(B)+C=Y$. Se generaron 100 patrones de entrenamiento para garantizar una variedad de datos que representaran diferentes combinaciones de las entradas.
- **Caso 3**: Predicción de resultados en una lotería con cinco entradas numéricas y 10 salidas. Se realizaron pruebas adicionales con datos aleatorios para medir la capacidad de generalización del modelo.
- **Caso 4**: Clasificación de letras a partir de una matriz de 35 píxeles, con una salida que representa el código ASCII de la letra. Este caso permitió evaluar la capacidad del modelo para reconocer patrones visuales simples.

Cada caso de estudio fue diseñado para evaluar tanto la capacidad del modelo para aprender patrones no lineales como su habilidad para generalizar los resultados a datos no vistos previamente. Se llevaron a cabo varios experimentos para ajustar los hiperparámetros y optimizar el rendimiento del modelo.

IV. Resultados

Los experimentos realizados con el algoritmo de retropropagación presentaron resultados diversos según el caso de estudio y las configuraciones de hiperparámetros seleccionadas. A continuación, se describen los resultados obtenidos para cada uno de los casos evaluados.

A. Caso 1: Clasificación Binaria

Este caso presentó ciertos desafíos debido a la naturaleza binaria de las entradas y las salidas. Se evaluaron varias combinaciones de funciones de activación y tasas de aprendizaje (α) para encontrar la configuración más adecuada. A continuación, se destacan los resultados de algunas combinaciones:

- **H = Sigmoidal, O = ReLU**: El modelo completó el entrenamiento en **54,219 épocas**. Mostró una buena capacidad para salir de mínimos locales, ajustando los patrones de manera uniforme hacia el final del entrenamiento.
- **H = Sigmoidal, O = Leaky ReLU**: El entrenamiento se completó en **90,781 épocas**. Aunque esta configuración mostró una mayor dependencia de los pesos iniciales, el modelo logró entrenar. Sin embargo, algunos patrones complicados se ajustaron solo en las últimas 2000 épocas.
- **H = Sigmoidal, O = Lineal**: El modelo entrenó en **69,708 épocas**. Este comportamiento fue similar al de

la combinación con Leaky ReLU, con los mismos patrones problemáticos que necesitaron ajustes adicionales, afectando temporalmente a otros patrones.

- **H = Tanh, O = ReLU**: En los primeros intentos, el modelo no entrenó debido a una actualización de pesos insuficiente. Al ajustar la tasa de aprendizaje a $\alpha = 0{,}06$, el entrenamiento se completó en **5,585 épocas**, mostrando que con Tanh en la capa oculta, un α más alto mejora significativamente la convergencia.
- **H = SoftPlus, O = Lineal**: El entrenamiento fue ineficaz con $\alpha = 0{,}008$, mostrando una actualización de pesos muy lenta. A pesar de aumentar el α, la red seguía teniendo dificultades, sugiriendo que SoftPlus no es adecuada para esta tarea en combinación con activaciones lineales en la capa de salida.

1) Pruebas Adicionales: En las pruebas adicionales con patrones no vistos durante el entrenamiento, el modelo generalizó correctamente en la mayoría de los casos. Sin embargo, algunas combinaciones, especialmente con ReLU y Leaky ReLU, mostraron que ciertos patrones necesitaron más épocas para alcanzar un error aceptable.

2) Conclusión del Caso 1:

- La combinación de Tanh en la capa oculta y un $\alpha = 0{,}06$ proporcionó el mejor rendimiento en términos de convergencia rápida y precisión.
- ReLU y Leaky ReLU mostraron una mayor dependencia de los pesos iniciales, con entrenamientos inestables en algunos casos.
- Algunos patrones presentaron consistentemente mayores dificultades para ajustarse, lo que sugiere la necesidad de analizar la distribución de los datos o los pesos iniciales.
- SoftPlus no fue adecuada para este problema de clasificación cuando se combinó con funciones de activación ReLU o Lineal.

Figura 4. Resultados del entrenamiento del Caso 1.

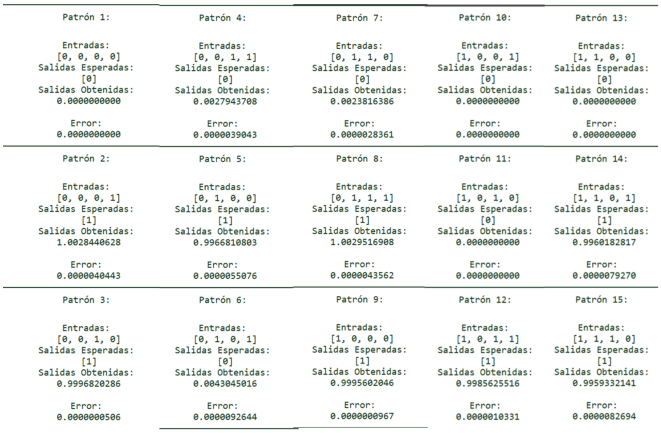

Figura 5. Resultados de las pruebas del Caso 1.

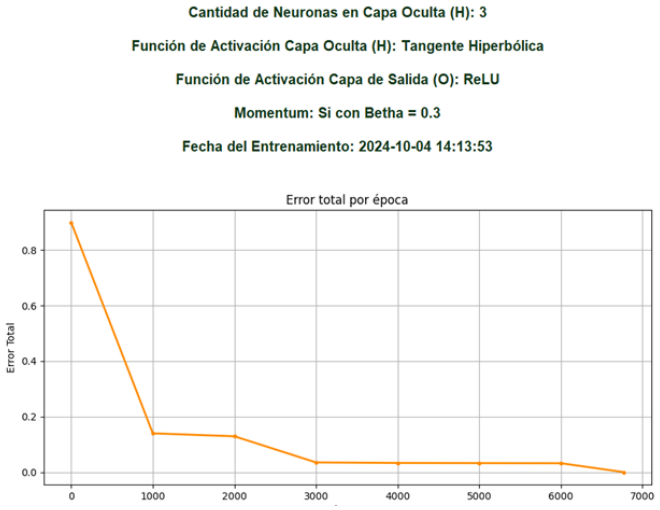

Figura 6. Curvas de error en el entrenamiento del Caso 1.

3) Imágenes de los Resultados:

B. Caso 2: Análisis de Datos con Diferentes Tamaños

1) Entrenamiento 1: 100 Datos: El modelo fue entrenado con un conjunto de 100 datos, donde las pruebas mostraron que, aunque la mayoría de las salidas presentaban errores bajos, algunos patrones tuvieron errores mayores a 0.5 (**Figura 7**).

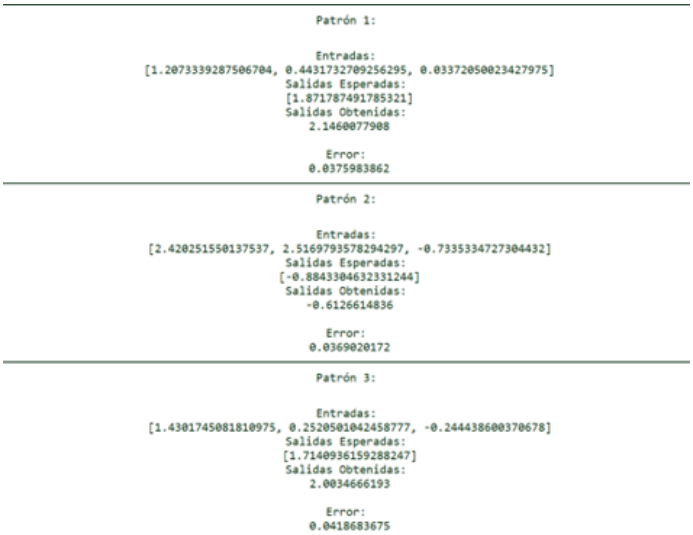

Figura 7. Resultados del entrenamiento con 100 datos.

2) Entrenamiento 2: 1000 Datos: Al aumentar el tamaño del conjunto a 1000 datos, el modelo mostró una mayor precisión, con un 64 % de los patrones presentando errores menores a 0.01 (**Figura 8**).

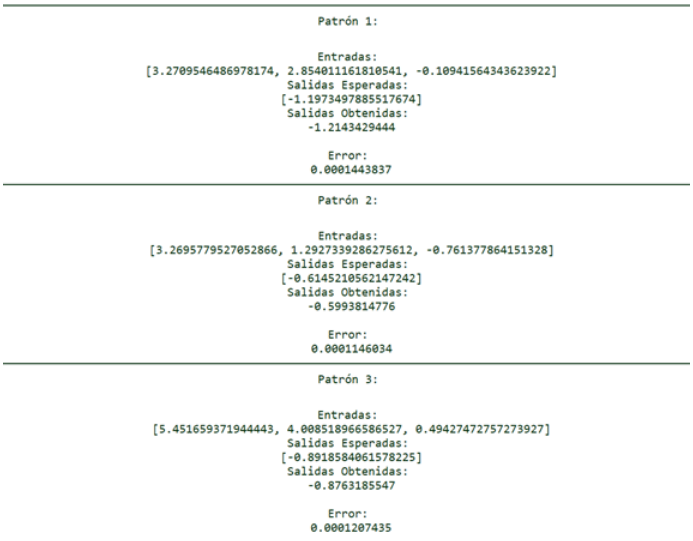

Figura 8. Resultados del entrenamiento con 1000 datos.

3) Entrenamiento 3: 1500 Datos: El tercer entrenamiento, con 1500 datos, mostró mejoras significativas en la precisión, con un 50 % de patrones con errores inferiores a 0.001. Sin embargo, el 24 % de los datos presentaron errores superiores a 0.04 (**Figura 9**).

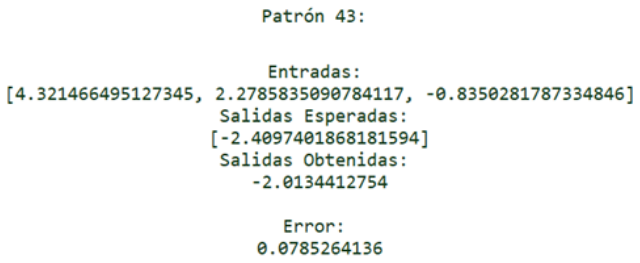

Figura 9. Resultados del entrenamiento con 1500 datos.

4) Conclusión del Caso 2: Aumentar el tamaño del conjunto de datos mejora considerablemente la precisión del modelo. Sin embargo, un porcentaje significativo de patrones todavía presenta errores altos, lo que sugiere la necesidad de ajustar la arquitectura de la red o los hiperparámetros.

C. Caso 3: Pruebas con Entradas Modificadas

1) Entrenamiento: El modelo fue entrenado con 16 entradas de diapositivas, obteniendo una precisión aceptable tras varios intentos. Se logró un entrenamiento eficiente ajustando el número de épocas y neuronas necesarias para la convergencia.

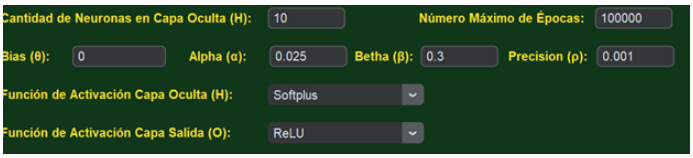

Figura 10. Datos de entrenamiento en el Caso 3.

2) Resultados del Entrenamiento: El modelo clasificó correctamente los datos del entrenamiento. Sin embargo, cuando se realizaron pruebas con 10 datos adicionales, donde se modificaron los valores de día y mes, los resultados no fueron consistentes, mostrando errores significativos.

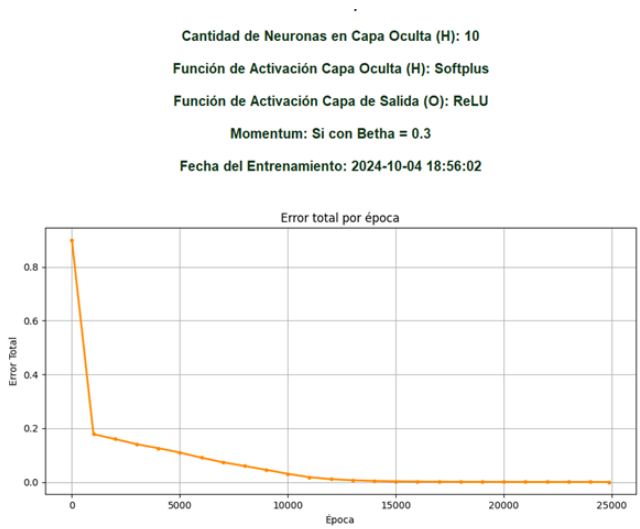

Figura 11. Resultados del entrenamiento en el Caso 3.

3) Conclusión del Caso 3: Aunque el modelo logró una buena clasificación de los datos originales, presentó problemas para generalizar cuando se alteraron las entradas. Esto indica que el conjunto de datos utilizado fue insuficiente para mejorar la capacidad de generalización del modelo.

D. Caso 4: Clasificación de Letras

1) Datos de Entrenamiento: Se entrenó un modelo utilizando matrices binarias de 7x5 que representaban letras del abecedario, donde el valor 1 representaba una casilla negra y el 0 una casilla blanca.

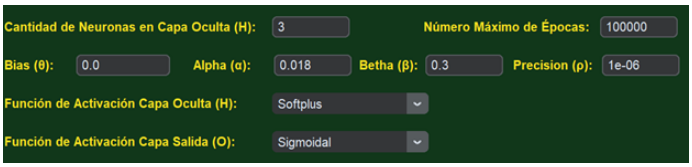

Figura 12. Ejemplo de datos de entrenamiento en el Caso 4.

2) Resultados del Entrenamiento: El modelo fue capaz de asociar correctamente las matrices con sus respectivos códigos ASCII, logrando una alta precisión en la clasificación de las letras.

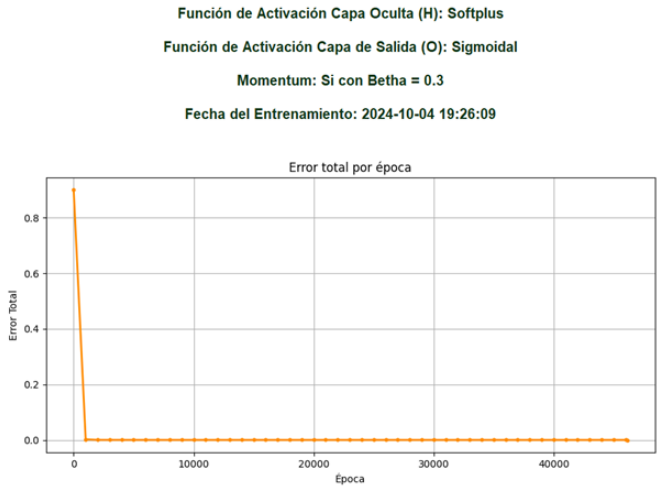

Figura 13. Resultados del entrenamiento en el Caso 4.

3) Resultados de Pruebas: Se realizaron pruebas adicionales para evaluar la capacidad del modelo para asociar las representaciones binarias con las letras correspondientes.

```
Patrón 1:

Entradas:
[1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1,
0, 0, 0, 1]
Salidas Esperadas:
[65] = ['A']
Salidas Obtenidas:
64.9942509359
= ['A']
Error:
0.0000165259
```

```
Patrón 2:

Entradas:
[1, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1,
1, 1, 1, 1]
Salidas Esperadas:
[66] = ['B']
Salidas Obtenidas:
65.9985225716
= ['B']
Error:
0.0000010914
```

Figura 14. Resultados de las pruebas en el Caso 4.

4) Conclusión del Caso 4: El modelo demostró una buena capacidad para aprender las representaciones binarias de las letras y asociarlas correctamente con sus respectivos códigos ASCII. Esto sugiere que el uso de redes neuronales para el reconocimiento de caracteres es efectivo, logrando una alta precisión en la clasificación.

V. Conclusiones

1) John Sebastián Galindo Hernández: En términos generales, el entrenamiento del algoritmo demostró ser efectivo para aprender y ajustar los pesos en los cuatro casos evaluados, aunque se observó que la configuración de las funciones de activación y el tamaño del conjunto de datos limitan su rendimiento. Los modelos mostraron una buena precisión en los conjuntos de datos de entrenamiento, pero la capacidad de generalización varió de un caso a otro. Por ejemplo, en el **Caso 1**, una mejor convergencia y generalización se obtuvo utilizando **Tanh** en la capa oculta y **ReLU** en la capa de salida con una tasa de aprendizaje adecuada. También se observó que algunos patrones específicos presentaron dificultades, lo que resalta la importancia de elegir correctamente la función de activación y los hiperparámetros para lograr un entrenamiento exitoso.

2) Miguel Ángel Moreno Beltrán: El **Caso 2** mostró la importancia crítica del tamaño del conjunto de datos en el rendimiento del modelo. Al aumentar la cantidad de datos, se registró una mejora significativa en la precisión, especialmente en patrones con errores bajos. Sin embargo, un porcentaje de los datos continuó mostrando errores altos, lo que sugiere que, si bien la expansión del conjunto de datos mejora el rendimiento, la arquitectura del modelo también debe ajustarse para reducir aún más los errores en estos casos. El **Caso 3** destacó una limitación del modelo: su capacidad para generalizar cuando se introducen pequeñas variaciones en los datos, como cambios aleatorios en las entradas. Aunque los datos originales se clasificaron fácilmente, las pruebas con valores alterados mostraron inconsistencias, lo que indica que se requiere más datos y una mejor estrategia de entrenamiento para mejorar la robustez del modelo frente a este tipo de variaciones.

3) Conclusión Final: Para concluir, los resultados obtenidos en los cuatro casos confirman que el algoritmo de retropropagación es una herramienta muy poderosa para resolver

problemas de clasificación y predicción, siempre que esté bien configurado y se utilicen conjuntos de datos suficientemente representativos. Sin embargo, la capacidad de generalización del modelo puede verse limitada si se enfrenta a variaciones no vistas durante el entrenamiento. Por lo tanto, es necesario realizar un entrenamiento exhaustivo y utilizar técnicas sofisticadas como el ajuste dinámico de hiperparámetros y la expansión de datos para mejorar el rendimiento. Además, se debe investigar más sobre arquitecturas más profundas y técnicas avanzadas como **Dropout** o la regularización para abordar problemas más complejos, lo que permitiría una generalización más eficiente y un rendimiento robusto en escenarios reales.

REFERENCIAS

[1] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, pp. 533–536, 1986.

[2] G. Hinton and D. E. Rumelhart, *Learning internal representations by error propagation*. MIT Press, 1986, vol. 1.

[3] J. R. Hilera and V. J. Martínez, *Redes neuronales artificiales: fundamentos, modelos y aplicaciones*. RA-MA, 1995.

[4] Y. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller, *Efficient backprop*. Springer, 1998, vol. 1524.

[5] C. M. Bishop, *Pattern recognition and machine learning*. Springer, 2006.

[6] S. Hochreiter and J. Schmidhuber, "The vanishing gradient problem during learning recurrent neural nets and problem solutions," *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, vol. 6, no. 2, pp. 107–116, 1998.

[7] N. Qian, "On the momentum term in gradient descent learning algorithms," *Neural Networks*, vol. 12, no. 1, pp. 145–151, 1999.

[8] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[9] Y. B. Yann LeCun and G. Hinton, "Deep learning," *Nature*, vol. 521, pp. 436–444, 2015.

[10] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain," *Psychological review*, vol. 65, no. 6, pp. 386–408, 1958.

[11] B. Widrow and M. E. Hoff, "Adaptive switching circuits," in *1960 IRE WESCON convention record*. Ieee, 1960, pp. 96–104.

[12] B. Widrow and M. A. Lehr, "30 years of adaptive neural networks: Perceptron, madaline, and backpropagation," *Proceedings of the IEEE*, vol. 78, no. 9, pp. 1415–1442, 1990.