Instituto Tecnológico de Costa Rica

Escuela de Ingeniería en Computación.

Programa de Bachillerato en Ingeniería en Computación.

IC6200 - Inteligencia Artificial

Trabajo Práctico 1: Redes Neuronales

Profesor: Ph. D. Saúl Calderón Ramírez

Resumen

En el presente trabajo práctico se introducirá el concepto de redes neuronales. Consta de 100 puntos distribuidos en 4 secciones. La primera sección tratará sobre la implementación de una red neuronal realimentada, la segunda sección de un clasificador para XOR, la tercera sección de un clasificador de datos en $\mathbb{R}^2$, y la cuarta sección de un clasificador de ataques con datos estructurados.

Estudiantes

José Adrián Amador Ávila	2016101574
Josué Castro Ramírez	2020065036

IS2024

Índice

1. Implementación de una red neuronal realimentada

Esta sección detalla como se realizó la implementación de la red a través de una *clase* en el lenguaje de programación Python, la cuál se hizo de la forma más matricial posible con las funciones que brinda la biblioteca *PyTorch*. La presente sección cubre los siguientes contenidos:

- Concepto de red neuronal
- Clase `Multi_layer_perceptron`
- Implementación de la función *__init__()*
- Propagación hacia adelante: función *forward()*
- Entrenamiento de la red: función *train_mlp()*

1.1. Concepto de red neuronal

Las redes neuronales son sistemas de procesamiento de información inspirados en el sistema nervioso y el cerebro de animales y humanos. Están compuestas por una gran cantidad de unidades simples, conocidas como neuronas, que operan en paralelo y se comunican mediante señales de activación a lo largo de conexiones dirigidas [1]. Las redes neuronales son ensamblajes altamente interconectados de elementos simples que cambian de estado cuando su entrada supera un valor umbral específico, lo que permite modelar comportamientos cooperativos y propiedades computacionales similares a las del sistema nervioso [2].

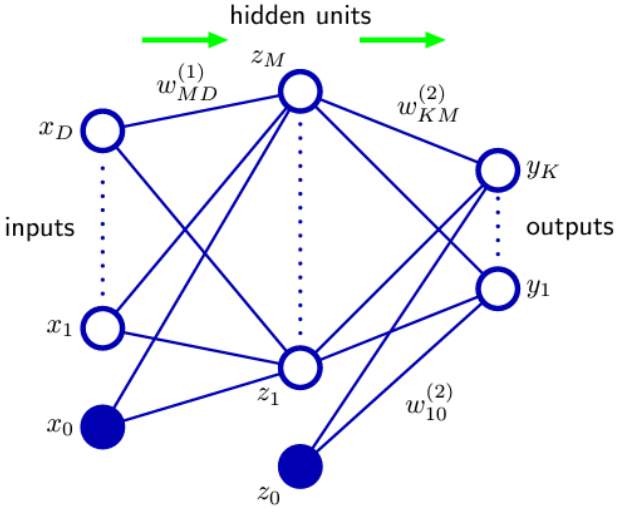


Figura 1: Diagrama de una red neuronal de dos capas (excluyendo la capa de entrada), tomado de [3].

Tal como se muestra en la figura 1, una red neuronal es esencialmente un grafo, usualmente de tres capas. Podemos entonces definir cada una de las capas como:

- La capa de entrada de D nodos, constituida por los valores $x_1, x_2, x_3, ..., x_D$ del arreglo de entrada $\vec{x} \in \mathbb{R}^D$.
- La capa oculta de M nodos, constituida por los valores $y^o_0, y^o_1, y^o_2, ..., y^o_M$, en notación vectorial dada por $\vec{y^o} \in \mathbb{R}^M$. Esta capa controla el nivel de generalización de la red (suavidad de la superficie de decisión).
- La capa de salida, constituida por K nodos $y^s_1, ..., y^s_K$, con $\vec{y^s} \in \mathbb{R}^K$, lo cual corresponde al **número K de clases por discriminar**.

1.1.1. Propagación hacia adelante

Cada una de las capas tiene una matriz de pesos $W^o \in \mathbb{R}^{D\times M}$ y $W^s \in \mathbb{R}^{M\times K}$. La primera matriz de pesos conecta las neuronas de entrada con la capa oculta, y la segunda matriz conecta la capa oculta con la capa de salida. Para cada nodo de la capa oculta, se define el **peso neto o coeficiente de activación** p^o_m, el cual está dado por la combinación lineal de los valores de los nodos de entrada por los pesos de la capa oculta:

$$p^o_m\left(\vec{x}, W^o\right) = \sum_{d=1}^{D} W^o_{d,m} x_d + W^o_{0,m}, \tag{1}$$

donde el peso $W^o_{0,m}$ se refiere comúnmente al **sesgo**, el cual de ignorarse, se puede reescribir la expresión como:

$$p^o_m\left(\vec{x}, W^o\right) = \sum_{d=1}^{D} W^o_{d,m} x_d$$

fijando a $x_0 = 1$, siendo también el **bias** o **sesgo**.

El peso neto p_m es transformado usando una **función de activación** no lineal y diferenciable:

$$y^o_m\left(\vec{x}, W^o\right) = g^o\left(p^o_m\left(\vec{x}, W^o\right)\right), \tag{2}$$

cada uno de los valores de los nodos $y^o_0, y^o_1, y^o_2, ..., y^o_M$, se les llama **salidas de la capa oculta**.

La función de activación g^o denota la condición de una neurona como activada o desactivada ante una entrada específica. Para el caso de este trabajo práctico, se hará uso de la función **tangente hiperbólica** como función de activación, y está dada por:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{3}$$

$$\frac{d}{dx}\tanh(x) = 1 - \tanh^2(x) \tag{4}$$

Ahora bien, para la capa de salida, siguiendo la figura 1, se define el peso neto para la unidad de salida k como:

$$p^s_k(\vec{x}, W^s) = \sum_{m=0}^{M} W^s_{m,k} y^o_m, \tag{5}$$

donde $y^o_0 = 1$, siendo la neurona de sesgos en la capa oculta. El peso neto para una neurona en la capa de salida está dada por la combinación lineal de las salidas de la capa oculta por los pesos de la capa de salida. Y de la misma forma, para obtener las salidas de la capa de salida, se pasa cada uno de los pesos netos p^s_k por una función de activación:

$$y^s_k(\vec{x}, W^s) = g^s(p^s_k(\vec{x}, W^s). \tag{6}$$

Si reemplazamos el funcional y^s_k por su definición en la ecuación 5, se tiene que:

$$y^s_k(\vec{x}, W^s) = g^s\left(\sum_{m=0}^{M} W^s_{m,k} y^o_m\right), \tag{7}$$

y si, de la misma manera reemplazamos y^o_m por su ecuación 2, y dentro de esa misma ecuación el funcional p^o_m por su ecuación 1:

$$y^s_k(\vec{x}, W^o, W^s) = g^s\left(\sum_{m=0}^{M} W^s_{m,k} g^o\left(\sum_{d=1}^{D} W^o_{d,m} x_d + W^o_{0,m}\right) + W^s_{0,k}\right). \tag{8}$$

El proceso de evaluar la ecuación 8 se le conoce como **propagación hacia adelante**.

1.1.2. Retropropagación

Después de realizar la propagación hacia adelante, se debe actualizar los pesos de la capa de salida y la capa oculta. Para encontrar mejores pesos que logren minimizar el error, se usa una ténica llamada **retropropagación** o **backpropagation**. Para esto se va a utilizar el algoritmo del **descenso del gradiente**:

$$\vec{w}(\tau + 1) = \vec{w}(\tau) - \alpha \nabla E(\vec{w}(\tau)). \tag{9}$$

Definimos entonces la ecuación del error como:

$$E(\vec{w}) = \frac{1}{N}\sum_{n=1}^{N} \|y(\vec{x_n}, \vec{w}) - \vec{t_n}\|^2, \tag{10}$$

o bien,

$$E(\vec{w}) = \sum_{n=1}^{N} E_n(\vec{w}),$$

con

$$E_n(\vec{w}) = \frac{1}{N}\|y(\vec{x_n}, \vec{w}) - \vec{t_n}\|^2.$$

Capa de salida

Para actualizar los pesos de la **capa de salida**, W^s, tenemos,

$$W^s_{m,k}(\tau + 1) = W^s_{m,k}(\tau) - \alpha \nabla W^s_{m,k}(\tau), \tag{11}$$

donde,

$$\nabla W^s_{m,k}(\tau) = \frac{d}{dW^s_{m,k}} \left(\sum_{n=1}^{N} E_n(W^s_{m,k}) \right). \tag{12}$$

Podemos transformar de igual manera la ecuación de $E_n(\vec{w})$, de la siguiente forma,

$$E_n(W^s) = \sum_{k=0}^{K} E_{k,n}(W^s),$$

con el error por cada unidad de salida k,

$$E_{k,n}(W^s) = \frac{1}{N} \left(y^s_k(\vec{x_n}, W^s) - t_{k,n} \right)^2 = \frac{1}{N} \left(g^s(p^s_{k,n}) - t_{k,n} \right)^2.$$

Calculando la derivada parcial del error para una muestra n respecto a una entrada de la matriz de pesos $W^s_{m',k'}$, se tiene que:

$$\frac{dE_n}{dW^s_{m',k'}} = \frac{d}{dW^s_{m',k'}} \left(\sum_{k=0}^{K} E_{k,n}(W^s) \right),$$

lo que significa,

$$\frac{dE_{k,n}}{dW^s_{m',k'}} = \frac{d}{dW^s_{m',k'}} \left(\frac{1}{N} \left(g^s(p^s_{k',n}) - t_{k',n} \right)^2 \right). \tag{13}$$

Podemos entonces descomponer la derivada parcial usando la regla de la cadena para obtener lo siguiente:

$$\frac{dE_{k,n}}{dW^s_{m',k'}} = \frac{dE_{k,n}}{dy^s_{k',n}} \frac{dy^s_{k',n}}{dp^s_{k',n}} \frac{dp^s_{k',n}}{dW^s_{m',k'}}. \tag{14}$$

Dado que se está trabajando con la función tangente hiperbólica, $y^s_{k',n} = g^s(p^s_{k',n}) = \tanh(p^s_{k',n})$, para la cual tenemos su derivada en la ecuación 4, $\frac{d}{dp^s_{k',n}} \tanh(p^s_{k',n}) = 1 - \tanh^2(p^s_{k',n})$. Con esto, cada una de las derivadas parciales de la ecuación 14, se desarrolla de la siguiente manera:

$$\frac{dE_{k,n}}{dy^s_{k',n}} = \frac{dE_{k,n}}{dy^s_{k',n}} \left((y^s_{k',n} - t_{k',n})^2 \right) = 2(y^s_{k',n} - t_{k',n}), \tag{15}$$

$$\frac{dy^s_{k',n}}{dp^s_{k',n}} = 1 - \tanh^2(p^s_{k',n}) = 1 - (y^s_{k',n})^2, \tag{16}$$

$$\frac{dp^s_{k',n}}{dW^s_{m',k'}} = \frac{d}{dW^s_{m',k'}} \left(\sum_{m=0}^{M} W^s_{m,k'} y^o_{m,n} \right) = y^o_{m',n}, \tag{17}$$

por lo cual, la derivada de la ecuación 14, se puede expresar como:

$$\frac{dE_{k',n}}{dW^s_{m',k'}} = \left(2(y^s_{k',n} - t_{k',n}) \right) \left(1 - (y^s_{k',n})^2 \right) y^o_{m',n}, \tag{18}$$

y podemos tomar, para una muestra n, el *delta* o *cambio de aprendizaje* δ^s_k como:

$$\delta^s_{k',n} = \left(2(y^s_{k',n} - t_{k',n}) \right) \left(1 - (y^s_{k',n})^2 \right). \tag{19}$$

De esta manera, la actualización del peso $W^s_{m',k'}$ vendría dada, según la ecuación 11 por cada muestra como:

$$W^s_{m',k'}(\tau+1) = W^s_{m',k'}(\tau) - \alpha \nabla W^s_{m',k'}(\tau), \tag{20}$$

con

$$\nabla W^s_{m',k'}(\tau) = \delta^s_{k',n} y^o_{m',n}.$$

Capa oculta

Por otro lado, para actualizar los pesos de la **capa oculta** W^o, se desarrollará de forma similar, la ecuación:

$$W^o_{d',m'}(\tau+1) = W^o_{d',m'}(\tau) - \alpha \nabla W^o_{d',m'}(\tau), \tag{21}$$

donde,

$$W^o_{d',m'}(\tau) = \frac{d}{dW^o_{d',m'}} \left(\sum_{n=1}^{N} E_n(W^o_{d',m'}) \right).$$

Tenemos que el error por cada unidad k de la capa de salida:

$$\begin{aligned} E_{k,n}(W^o) &= \frac{1}{N}(y^s_k(\vec{x}_n, W^o) - t_{k,n})^2 = \frac{1}{N}(y^s_k(p^s_k(\vec{x}_n, W^o)) - t_{k,n})^2 \\ &= \frac{1}{N}\left(g^s\left(\sum_{m=0}^{M} W^s_{m,k} g^o\left(\sum_{d=1}^{D} W^o_{d,m} x_d \right)\right) - t_{k,n} \right)^2. \end{aligned}$$

Por lo tanto, podemos descomponer la derivada del error usando la regla de la cadena, como:

$$\frac{dE_{k,n}}{dW^o_{d',m'}} = \frac{dE_{k,n}}{dy^s_{k,n}} \frac{dy^s_{k,n}}{dp^s_{k,n}} \frac{dp^s_{k,n}}{dy^o_{m',n}} \frac{dy^o_{m',n}}{dp^o_{m',n}} \frac{dp^o_{m',n}}{dW^o_{d',m'}}. \tag{22}$$

Algo importante a tomar en cuenta es que, un cambio en la capa oculta ocasiona un cambio en el error de todas las unidades en la capa de salida, por lo que entonces:

$$\frac{dE_n}{dW^o_{d',m'}} = \sum_{k=0}^{K} \frac{dE_{k,n}}{dW^o_{d',m'}} = \sum_{k=0}^{K} \left(\frac{dE_{k,n}}{dy^s_{k,n}} \frac{dy^s_{k,n}}{dp^s_{k,n}} \frac{dp^s_{k,n}}{dy^o_{m',n}} \right) \frac{dy^o_{m',n}}{dp^o_{m',n}} \frac{dp^o_{m',n}}{dW^o_{d',m'}}.$$

Y sus derivadas parciales dadas por:

$$\frac{E_{k,n}}{dy^s_{k,n}} = 2(y^s_{k,n} - t_{k,n}) \tag{23}$$

$$\frac{dy^s_{k,n}}{dp^s_{k,n}} = 1 - \tanh^2(p^s_{k,n}) = 1 - (y^s_{k,n})^2 \tag{24}$$

$$\frac{dp^s_{k,n}}{dy^o_{m',n}} = W^s_{m',k} \tag{25}$$

$$\frac{dy^o_{m',n}}{dp^o_{m',n}} = 1 - \tanh^2(p^o_{m',n}) = 1 - (y^o_{m',n})^2 \tag{26}$$

$$\frac{dp^o_{m',n}}{dw^o_{d',m'}} = x_{d'}, \tag{27}$$

y así, la derivada parcial del error respecto a un peso específico, para una muestra n, está dada por:

$$\frac{dE_n}{dW^o_{d',m'}} = \sum_{k=0}^{K} \big((2(y^s_{k,n} - t_{k,n}))(1 - (y^s_{k,n})^2)(W^s_{m',k}) \big)(1 - (y^o_{m',n})^2) x_{d'}. \tag{28}$$

Se había definido en la ecuación 19, el $\delta^s_{k,n}$ para la capa de salida, y este término se puede observar en la ecuación 28, por ello, podemos definir el delta en la capa oculta como:

$$\delta^o_{m',k} = \left(\sum_{k=0}^{K} \delta^s_{k,n} W^s_{m',k} \right) (1 - (y^o_{m',n})), \tag{29}$$

y por ende, para la ecuación 21, $W^o_{d',m'}(\tau+1) = W^o_{d',m'}(\tau) - \alpha \nabla W^o_{d',m'}(\tau)$, se define:

$$\nabla W^o_{d',m'} = \delta^o_{m',n} x_{d'}.$$

De esta manera, podemos actualizar los pesos de la capa de salida y oculta, usando las ecuaciones 20 y 21, respectivamente.

1.2. Clase `Multi_layer_perceptron`

Como se detalló al inicio de este informe, se decide definir una *clase* en el lenguaje de programación Python para abordar la implementación de la red neuronal realimentada, a continuación, en la figura 2 se presenta un diagrama de clases que muestra los atributos y métodos asociados a la clase de la red, así como su relación con la clase *TestMultiLayerPerceptron*, la cuál fue definida para realizar las respectivas pruebas unitarias la clase principal; para dicha labor se utilizará la biblioteca `unittest`.

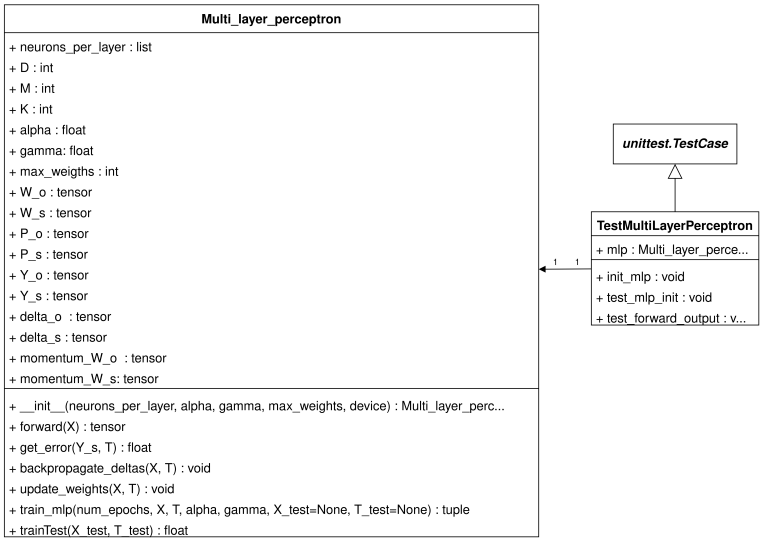


Figura 2: Diagrama de las clases implementadas.

1.3. Implementación de la función `__init__()`

Esta función es la encargada de inicializar un perceptrón multicapa, debe inicializar los pesos de la capa oculta y salida de manera completamente aleatoria en un rango entre -1 y 1. Se debe colocar la neurona de entrada extra que representa el *bias* $(D+1)$, así como la neurona *bias* $(M+1)$ en la capa oculta, en donde sus pesos se representan como *NaN*, por el hecho de no tener conexiones con las neuronas de la capa de entrada. Para su implementación se tomaron en cuenta las siguientes indicaciones:

- `neurons_per_layer` es un arreglo con las neuronas por cada capa, de la forma $[D,\ M,\ K]$, con D las neuronas de entrada, M las neuronas en la capa oculta, y K las neuronas de la capa de salida.
- `alpha` es el coeficiente de aprendizaje α.
- `gamma` es el coeficiente de momentum γ.
- `max_weights` es el valor máximo que puede tomar cualquiera de los pesos en las dos capas.

Otro punto a tomar en cuenta según las indicaciones del trabajo práctico, es que se puede guardar como atributos de clase los pesos netos de la capa oculta y salida (P^o, P^s), los deltas (δ^o, δ^s), así como los tamaños para cada capa (D, M, K), los cuáles son definidos en la figura 2.

1.3.1. Código

Ahora bien, como se puede observar en la figura 3, se tiene la implementación, con algunas omisiones, de lo mencionado anteriormente.

```python
def __init__(self, neurons_per_layer, alpha = 0.01, gamma = 0.9, max_weights = 1):
    # Número de neuronas por capa
    self.neurons_per_layer = neurons_per_layer
    self.D, self.M, self.K = neurons_per_layer

    # Neuronas de sesgo
    self.D += 1
    self.M += 1

    self.alpha = alpha
    self.gamma = gamma
    self.max_weights = max_weights

    # Inicialización de pesos de forma aleatoria dentro del rango permitido
    self.W_o = (torch.rand(self.D, self.M) * 2) - 1
    self.W_o = torch.min(self.W_o, torch.tensor(max_weights))

    self.W_s = (torch.rand(self.M, self.K) * 2) - 1
    self.W_s = torch.min(self.W_s, torch.tensor(max_weights))

    # Colocamos la neurona de sesgos en la capa oculta
    self.W_o[:, 0] = torch.nan

    # Inicialización de los valores de momentum para cada conjunto de pesos
    self.momentum_W_o = torch.zeros_like(self.W_o)
    self.momentum_W_s = torch.zeros_like(self.W_s)
```

Figura 3: Implementación en código de la función *__init__()*

1.3.2. Pruebas unitarias: *__init__()*

A continuación, se detalla la prueba de inicialización de la clase del perceptrón multicapa, la cuál busca comprobar la correcta creación del objeto, validando los siguientes aspectos:

- **Límites válidos:** Validar que la capa de pesos no exceda el límite establecido por el atributo `max_weights`.
- **Dimensiones válidas:** Validar que las dimensiones de las capas de pesos sea la correcta, $D \times M$ para la capa oculta y $M \times K$ para la de salida.
- **Columna de sesgo correcta:** Validar que la columna del sesgo sea correctamente inicializada.

Datos de entrada
Para efectos de esta prueba se definirá un único perceptrón multicapa, se llamará a dicho objeto como `TESTED_MLP`, del cual se muestran algunos de sus atributos a continuación:

- `neurons_per_layer` = {3, 3, 2}.
- `alpha` = 0.01.
- `gamma` = 0.9.
- `max_weights` = 0.5.
- `W_o` = { una matriz aleatoria $W^o \in \mathbb{R}^{D \times M} \mid \forall x \in W^o$: `-max_weights` <= x <= `max_weights`}.
- `W_s` = { una matriz aleatoria $W^s \in \mathbb{R}^{M \times K} \mid \forall x \in W^s$: `-max_weights` <= x <= `max_weights`}.

Resultados esperados
Para esta prueba, se espera que una inicialización correcta del perceptrón, más específicamente, se espera que las capas de peso tengan correctamente asignadas las dimensiones que se establecen en el atributo `neurons_per_layer`. Es decir, de la siguiente forma:

- $W^o \in \mathbb{R}^{4 \times 4}$, recordando que fue agregada la columna de sesgo, por eso se suma 1 a D y M.
- $W^s \in \mathbb{R}^{4 \times 2}$

Además, se debe validar que los valores aleatorios de las capas de peso estén acotados en un rango establecido,, que en este caso es [-0.5, 0.5]. Y por último, se espera que la capa oculta de pesos tenga correctamente inicializada la columna del sesgo con valores NaN en la primer columna.

Resultados obtenidos
A continuación en el *Cuadro 1* se observan los datos obtenidos al inicializar el objeto `TESTED_MLP`, además se detalla en la última columna las dimensiones obtenidas de las respectivas capas de pesos.

Datos de entrada	**W_o obtenido**				**W_s obtenido**		**Dimensiones obtenidas**
`TESTED_MLP`	NaN	**-0.5000**	−0,3958	0,0390	−0,0893	−0,4776	$W^o \in \mathbb{R}^{4\times4}$
	NaN	0,4139	−0,3631	−0,3337	**-0.5000**	−0,2983	$W^s \in \mathbb{R}^{4\times2}$
	NaN	0,1877	0,1377	**0.5000**	**0.5000**	−0,0185	
	NaN	0,3099	0,4860	**-0.5000**	−0,4422	**0.5000**	

Cuadro 1: Resultados obtenidos de la prueba unitaria de la función *__init__()*

De los anteriores resultados obtenidos en el *Cuadro 1*, se puede evidenciar que:

- Se resaltan **en negrita** los valores mínimos y máximos obtenidos en las capas de pesos, los cuales respetan estar en el rango establecido.
- Las dimensiones de las capas de peso son correctas, y respetan el número de neuronas que entra por parámetro de inicialización.
- El `W_o` obtenido cumple que su primer columna es puros datos NaN, por ende, la prueba es aprobada satisfactoriamente.

1.4. Propagación hacia adelante: función *forward(X)*

Partiendo de lo explicado en la sección 1.1 y la ecuación 8, se puede entonces implementar la propagación hacia adelante, no obstante, primero se debe convertir los cálculos a su forma matricial. En primer lugar, se tiene la matriz de entrada X, donde cada fila es un vector de observaciones, de modo $\vec{x}_1, \vec{x}_2, ..., \vec{x}_D \in \mathbb{R}^{NxD}$. Del mismo modo, se definió que la matriz de pesos para la capa oculta está dada por $W^o \in \mathbb{R}^{DxM}$, por lo tanto para obtener la matriz de pesos netos P^o de la capa oculta:

$$P^o = X \cdot W^o, \tag{30}$$

de manera que $P^o \in \mathbb{R}^{NxM}$. Y esta matriz de pesos netos para la capa oculta se pasa a la función de activación **tangente hiperbólica**:

$$Y^o = \tanh(P^o), \tag{31}$$

dando como resultado la salida de la capa oculta $Y^o \in \mathbb{R}^{NxM}$.

Luego de esto, como la primera columna de la matriz de pesos W^o es inicializada con *NanN*, la matriz Y^o también contiene *NaN* en su primer columna, por ello se cambian por 1, y ya se podría continuar con el siguiente paso, el cual es realizar la combinación lineal que genera los pesos netos P^s.

Se sabe que la matriz de pesos para la capa de salida está dada por $W^s \in \mathbb{R}^{MxK}$, y para obtener P^s se debe realizar la combinación lineal entre las salidas de la capa oculta Y^o por los pesos W^s, tal que,

$$P^s = Y^o \cdot W^s, \tag{32}$$

donde $P^s \in \mathbb{R}^{NxK}$. Y por último, se pasan los pesos netos de la capa de salida por la función de activación, dando como resultado las salidas de la capa de salida,

$$Y^s = \tanh(P^s), \tag{33}$$

de tal manera que, $Y^s \in \mathbb{R}^{NxK}$. Por medio de estas ecuaciones se puede entonces hacer la implementación de la función `forward(X)`, tal y como se muestra en la figura 4.

```python
def calculate_tanh(p):
    return (torch.exp(p) - torch.exp(-p)) / (torch.exp(p) + torch.exp(-p))

def forward(self, X):
    # Calcular la salida de la capa oculta
    self.P_o = X.mm(self.W_o)
    self.Y_o = calculate_tanh(self.P_o)

    # Reemplazar valores NaN con 1's
    tmpY_o = self.Y_o
    tmpY_o[torch.isnan(tmpY_o)] = 1.

    # Calcular la salida de la capa de salida
    self.P_s = tmpY_o.mm(self.W_s)
    self.Y_s = calculate_tanh(self.P_s)

    # Retornar la salida de la capa de salida
    return self.Y_s
```

Figura 4: Implementación en código de la función forward

1.4.1. Pruebas unitarias: *forward(X)*

A continuación, se detalla la prueba de la función `forward`, la cuál busca comprobar el correcto comportamiento de la función, validando los siguientes aspectos:

- **Dimensiones correctas:** Validar que las dimensiones de la capa de salida sea la correcta, es decir, $M \times K$.
- **Ausencia de valores NaN:** Validar que la capa de salida no tenga valores en NaN.
- **Rangos correctos:** Validar que la capa de salida tenga sólo valores entre -1 y 1, .

Datos de entrada
Al igual que en la *Sección 1.3.2*, se utilizará el objeto `TESTED_MLP` para realizar las pruebas, teniendo los mismos parámetros de inicialización. Además se utilizará una matriz aleatoria de neuronas de entrada X, la cuál es de la forma $X \in \mathbb{R}^{5\times 2}$.

Resultados esperados
Para esta prueba, se espera el correcto funcionamiento de la función *forward*, más específicamente, se espera que la capa de salida tenga las dimensiones $N \times K$ correctas, las cuáles son en este caso 5×2. Además, se espera que en la capa de salida presente ausencia de valores NaN, y que dichos valores estén en el rango [-1, 1], el cuál es dado por la función de activación tangente hiperbólico.

Resultados obtenidos
A continuación en el *Cuadro 2* se observan los datos obtenidos al inicializar el objeto `TESTED_MLP`, además se detalla en la última columna las dimensiones obtenidas de las respectivas capas de pesos.

Datos de entrada	Y_s obtenido	Dimensiones obtenidas
- `TESTED_MLP` - $X \in \mathbb{R}^{5\times 2}$	$\begin{bmatrix} -0{,}3239 & -0{,}8920 \\ 0{,}3210 & -0{,}0904 \\ 0{,}1181 & -0{,}6024 \\ 0{,}0111 & -0{,}7714 \\ 0{,}0207 & -0{,}8440 \end{bmatrix}$	$Y^s \in \mathbb{R}^{5\times 2}$

Cuadro 2: Resultados obtenidos de la prueba unitaria de la función *forward(X)*

De los anteriores resultados obtenidos en el *Cuadro 2*, se puede evidenciar que:

- Las dimensiones obtenidas en `Y_s` son correctas.
- La capa de salida no cuenta con valores NaN.
- Los valores de la capa de salida están dentro del rango de la función de activación.

1.5. Entrenamiento de la red: función `train_mlp()`

En la presente sección, se explica cómo se logra el entrenamiento de la red. Dicho proceso se realiza a través de la función `train_mlp()`, la cuál se encarga de entrenar al perceptrón multicapa mediante un algoritmo basado en el descenso del gradiente y usará como función de activación la función tangente hiperbólica para las capas. Esta función se construirá a partir de las siguientes funciones:

- `get_error(X, T)`: La cuál calcula el error medio cuadrático para un conjunto de salidas y objetivos.
- `back_propagate_deltas(X, T)`: La cuál calcula los deltas de error para las capas oculta y de salida.
- `update_weights(X, T)`: La cuál actualiza los pesos de las capas según los deltas de la red.

Antes de detallar cómo se realiza el entrenamiento, se procede a explicar cómo se construyen cada una de las funciones que componen a la función `train_mlp()`.

1.5.1. Función *get_error(X, T)*

La función de error de Cuadrados Medios o Mean Squared Error (MSE) es un criterio estadístico usado para medir la calidad de estimadores y modelos predictivos. El MSE calcula la media de los cuadrados de las diferencias entre los valores estimados y los reales, proporcionando una medida cuantitativa de la precisión del modelo. Es especialmente útil en contextos donde las diferencias grandes entre los valores predichos y los reales son particularmente indeseables, ya que el elevar al cuadrado los errores hace que estas diferencias tengan un impacto mayor en el MSE. En los materiales del curso, vimos la siguiente definición de la función de error:

$$E(\vec{w}) = \frac{1}{N}\sum_{n=1}^{N} \left\| y(\vec{x_n},\vec{w}) - \vec{t_n} \right\|^2 \tag{34}$$

En base a ello, se define en código la función de la siguiente manera:

```python
def get_error(self, Y_s, T):
    # Calcular el error utilizando la función de error E(w)
    error = torch.sum(torch.norm(Y_s - T, p=2, dim=1)**2) * (1 / Y_s.shape[0])
    return error.item()
```

Figura 5: Implementación en código de la función de error $E(\vec{w})$

1.5.2. Función *backpropagate_deltas(X, T)*

La retro-propagación, o propagación hacia atrás, es un método fundamental en el entrenamiento de redes neuronales artificiales. Este método consiste en ajustar los pesos de las conexiones de la red de manera eficiente, calculando los gradientes de una función de error respecto a los pesos de la red mediante una propagación hacia atrás desde la capa de salida hasta la de entrada. Este proceso permite que la red aprenda de los errores cometidos en predicciones anteriores, ajustándose para mejorar en iteraciones futuras.

Ecuaciones de los deltas

Tomando como base las ecuaciones 19 y 29, podemos calcular los δ de cada una de las capas. Recordando que el $\delta^s_{k,n} = \left(2(y^s_{k',n} - t_{k',n})\right)\left(1 - (y^s_{k',n})^2\right)$, para un k en específico dado una muestra n, podemos convertir la ecuación a su versión matricial de la siguiente manera:

$$\delta^s = 2(Y^s - T) \times (1 - (Y^s)^2), \tag{35}$$

como se mencionó anteriormente, $Y^s \in \mathbb{R}^{NxK}$, además, $T \in \mathbb{R}^{NxK}$, por lo tanto, $\delta^s \in \mathbb{R}^{NxK}$; y de manera similar, la ecuación $\delta^o_{m',k} = \left(\sum_{k=0}^{K} \delta^s_{k,n} W^s_{m',k}\right)(1 - (y^o_{m',n}))$,

$$\delta^o = (\delta^s \cdot (W^s)^T) \times (1 - (Y^o)^2), \tag{36}$$

como $\delta^s \in \mathbb{R}^{NxK}$ y $W^s \in \mathbb{R}^{MxK}$, se debe tomar la transpuesta de la matriz W^s para dicha operación, dando como resultado una matriz de tamaño NxM, lo cual coincide con el tamaño de Y^o, por lo tanto se tiene que $\delta^o \in \mathbb{R}^{NxM}$.

Código

Usando entonces las ecuaciones 35 y 36, se implementa el código mostrado en la figura 6.

```python
def backpropagate_deltas(self, X, T):
    # Calcular los deltas de la capa de salida
    self.delta_s = (2 * (self.Y_s - T)) * (1 - (self.Y_s**2))

    # Calcular los deltas de la capa oculta
    self.delta_o = self.delta_s.mm(self.W_s.t()) * (1 - (self.Y_o**2))
    return
```

Figura 6: Implementación en código de la función *backpropagate_deltas*

1.5.3. Función *update_weights(X, T)*

Para actualizar los pesos de ambas capas se va hacer uso de las ecuaciones 35 y 36, podemos entonces transformar las ecuaciones de los gradientes para los pesos de las dos capas, definidas en las ecuaciones 20 y 21, en su forma matricial como:

$$W^s(\tau + 1) = W^s(\tau) - \alpha \nabla W^s(\tau), \tag{37}$$

con $\nabla W^s(\tau) = (Y^o)^T \cdot \delta^s$, colocando primera la transpuesta de Y^o, dado que $Y^o \in \mathbb{R}^{NxM} \implies (Y^o)^T \in \mathbb{R}^{MxN}$ y $\delta^s \in \mathbb{R}^{NxK}$, por lo que $\nabla W^s(\tau) \in \mathbb{R}^{MxK}$, y en general dando un resultado correspondiente al tamaño de la matriz de pesos de la capa de salida. De la misma forma, tenemos entonces,

$$W^o(\tau + 1) = W^o(\tau) - \alpha \nabla W^o(\tau), \tag{38}$$

con $\nabla W^o(\tau) = X^T \cdot \delta^o$, donde de manera similar a lo anterior, se coloca primero la transpuesta de la matriz de entradas X, ya que, $X \in \mathbb{R}^{NxD} \implies X^T \in \mathbb{R}^{DxN}$ y $\delta^o \in \mathbb{R}^{NxM}$, por lo que $\nabla W^o(\tau) \in \mathbb{R}^{DxM}$, y en general, siendo el tamaño correspondiente a la matriz de pesos de la capa oculta.

Ahora bien, en el proyecto se hizo la implementación del **gradiente con momentum**, de lo cual la única modificación es usar el parámetro γ. Para ello simplemente se crean dos matrices del mismo tamaño de las matrices de pesos tanto para la capa oculta y salida, se multiplica el coeficiente de momentum γ a esas matrices y se suma el gradiente de los pesos W con el coeficiente de aprendizaje α. Esto y las ecuaciones 37 y 38, se ven reflejadas en la implementación en código de la figura 7.

```
def update_weights(self, X, T):
    # Calcular los gradientes para los pesos
    dW_o = X.t().mm(self.delta_o)
    dW_s = self.Y_o.t().mm(self.delta_s)

    # Aplicar momentum
    self.momentum_W_o = self.gamma * self.momentum_W_o + self.alpha * dW_o
    self.momentum_W_s = self.gamma * self.momentum_W_s + self.alpha * dW_s

    # Actualizar los pesos
    self.W_o -= self.momentum_W_o
    self.W_s -= self.momentum_W_s
    return
```

Figura 7: Implementación en código de la función *update_weights*

1.5.4. Función *train_mlp()*

Para esta función, se hace uso de las funciones explicadas en las secciones anteriores a esta. El objetivo principal es realizar el proceso de entrenamiento de la red neuronal, actualizando los pesos de las capas, y también llevando un historial de los errores por épocas. En la figura 8, se puede observar una versión simplificada de la implementación para esta función. La idea principal es que dado una matriz observaciones $X \in \mathbb{R}^{NxD}$, y sus targets $T \in \mathbb{R}^{NxK}$, se realice el proceso de **propagación hacia adelante** y **retropropagación**, dando como resultado el entrenamiento continuo de la red durante un número determinado de épocas. Dentro de la función se agrega la columna de sesgos a las entradas, se llama a la función *forward(X)* para obtener la salida de la red neuronal, i.e., $Y^s \in \mathbb{R}^{NxK}$, luego se realiza la retropropagación para actualizar los pesos de la capa de salida y oculta, y por último, verificamos el error de la predicción del modelo y si el modelo converge, esto es, si Y^s tiene datos muy cercanos a los presentes en T.

```
def train_mlp(self, num_epochs, X, T):
        # Agregar columna de 1s a las entradas
        X = self.addones(X)
        # Almacenar los errores de entrenamiento en cada época
        errors = []
        epoch_converge = None
        # Ciclo de entrenamiento
        for epoch in range(num_epochs):
            self.forward(X)
            self.backpropagate_deltas(X, T)
            self.update_weights(X, T)
            error = self.get_error(self.Y\_s, T)
            errors.append(error)
            adjusted_Y_s = torch.where(self.Y_s > 0.9, torch.tensor(1.),
                                       torch.where(self.Y_s < -0.9, torch.tensor(-1.), self.Y_s))
            # Verificar si las salidas ajustadas son iguales a los objetivos T
            if epoch_converge is None and torch.equal(adjusted_Y_s, T):
                print(f"Red converge en {epoch} con un error de {error},
                                    el Y_s ajustado es \n{adjusted_Y_s}")
                epoch_converge = epoch

        return errors, epoch_converge
```

Figura 8: Implementación en código de la función *train_mlp()*

2. Clasificación de función XOR

En esta sección se procederá a construir una red neuronal para resolver el problema del XOR. El problema del XOR es un problema no lineal. El XOR es una operación lógica binaria que toma dos entradas y produce una salida basada en la siguiente regla: la salida es verdadera (1), si y solo si, exactamente una de las entradas es verdadera, de lo contrario es falsa (0). En términos más simples, la salida es verdadera si las dos entradas son diferentes. En la tabla 3 podemos ver la tabla de verdad del XOR.

Entrada 1	**Entrada 2**	**Salida**
0	0	0
0	1	1
1	0	1
1	1	0

Cuadro 3: Tabla de verdad del XOR

En la figura 9, se puede ver el XOR como un problema de clasificación no lineal. Es decir, que no se puede resolver usando una única neurona que pueda dividir las dos clases existentes.

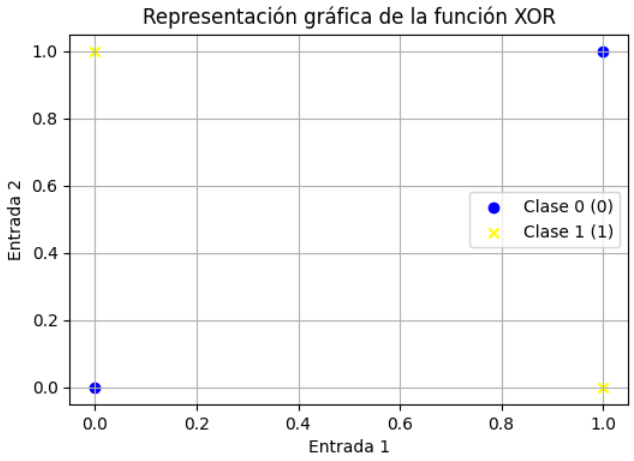


Figura 9: XOR como un problema de clasificación no lineal

Para poder construir la red neuronal que resuelva el problema, vamos a utilizar las entradas del XOR como las muestras $\vec{x} \in \mathbb{R}^2$; y las salidas como el conjunto T u objetivos (*targets*).

2.1. Arquitectura de la red XOR

Para resolver el problema del XOR, se define una arquitectura de $D = 2$ neuronas de entrada; $M = 2$ neuronas en la capa oculta; y $K = 1$ neuronas en la capa de salida. Esto se puede ver representado en la figura 10.

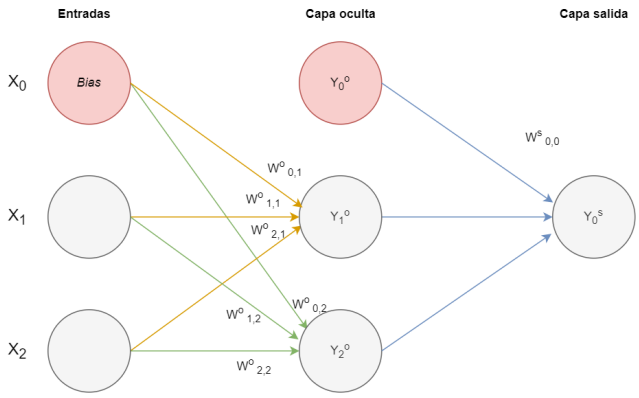


Figura 10: Arquitectura de la red del XOR

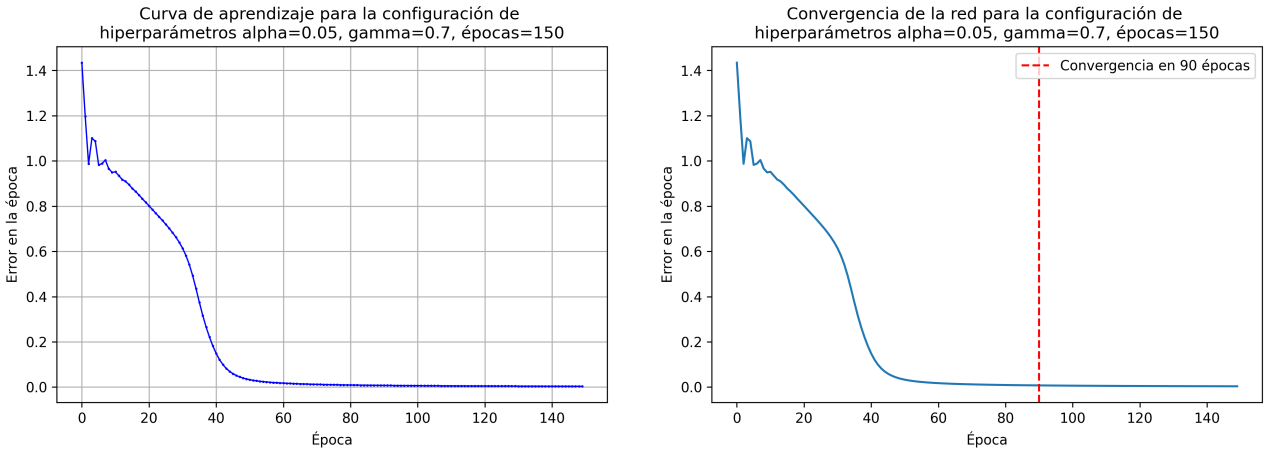
2.2. Entrenamiento de la red XOR y convergencia

Se entrenó la red con distintas combinaciones de parámetros para encontrar cuáles producían la salida esperada. En el cuadro 4, se documentan los parámetros para los cuales la red tuvo una convergencia, es decir, se alcanzó un error de 0 (o al menos cercano a 0).

Configuración	α	γ	**Épocas**
1	0.05	0.7	150
2	0.05	0.9	75
3	0.1	0.0001	200
4	0.1	0.3	150
5	0.1	0.08	200
6	0.1	0.7	50
7	0.1	0.9	50

Cuadro 4: Hiperparámetros probados para la red neuronal del XOR que presentaron convergencia

Para la configuración 1: $\alpha = 0{,}05, \gamma = 0{,}7$; con 150 épocas, se tiene que la red converge alcanzadas las 90 épocas de entrenamiento, tal y como se muestra en la figura 11.

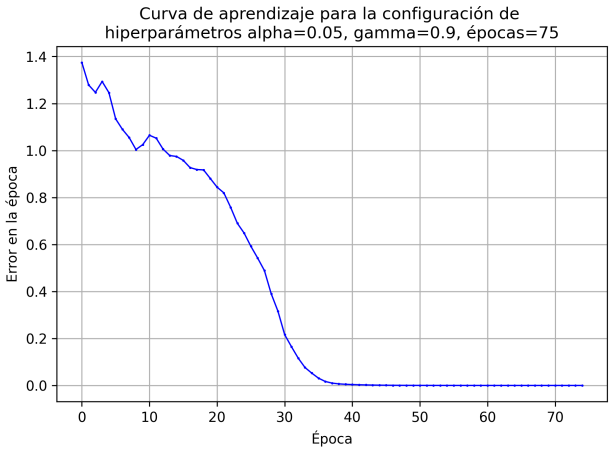


(a) Curva de aprendizaje de la configuración 1

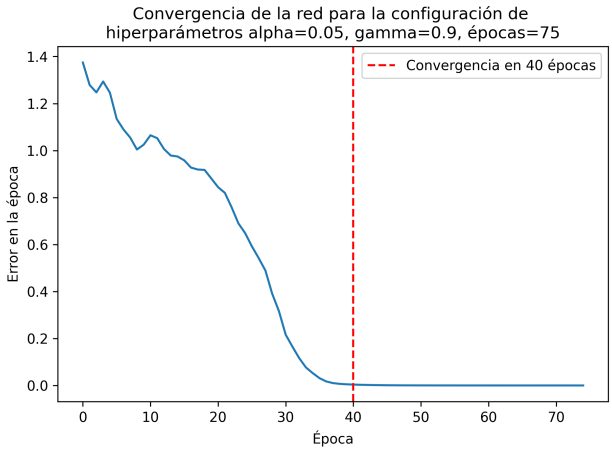
(b) Convergencia de la configuración 1

Figura 11: Curva de aprendizaje y convergencia para la configuración 1

Para la configuración 2: $\alpha = 0{,}05, \gamma = 0{,}9$; con 75 épocas, la red converge alcanzadas las 40 épocas. Si vemos la figura 12, se puede deducir que al modificar el hiperparámetro γ y colocar uno más grande, la red converge en una menor cantidad de épocas para el mismo valor de α.



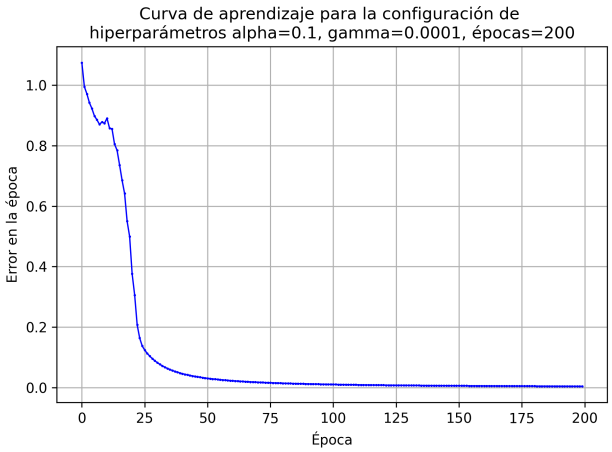
(a) Curva de aprendizaje de la configuración 2



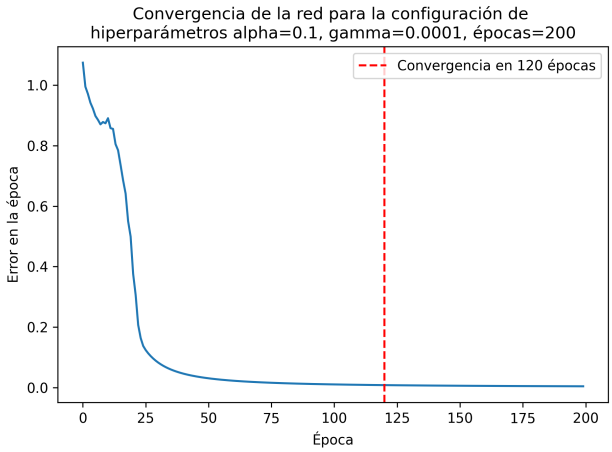
(b) Convergencia de la configuración 2

Figura 12: Curva de aprendizaje y convergencia para la configuración 2

Ahora bien, para la configuración 3: $\alpha = 0{,}1, \gamma = 0{,}0001$; con 200 épocas, se puede observar que con un α un poco más grande, la red de igual forma converge aún si el parámetro γ es muy cercano a 0. Esto se puede apreciar en la figura 13.



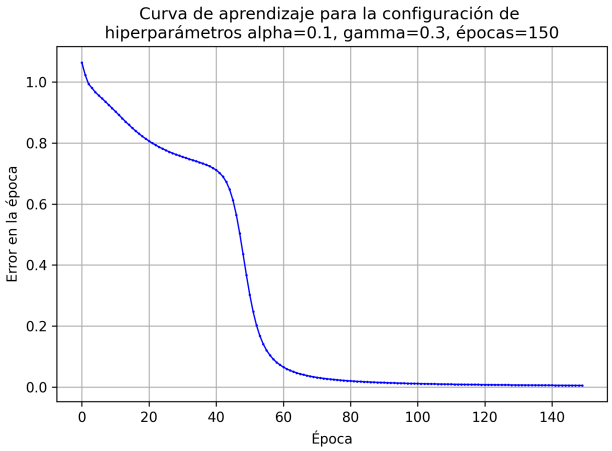
(a) Curva de aprendizaje de la configuración 3



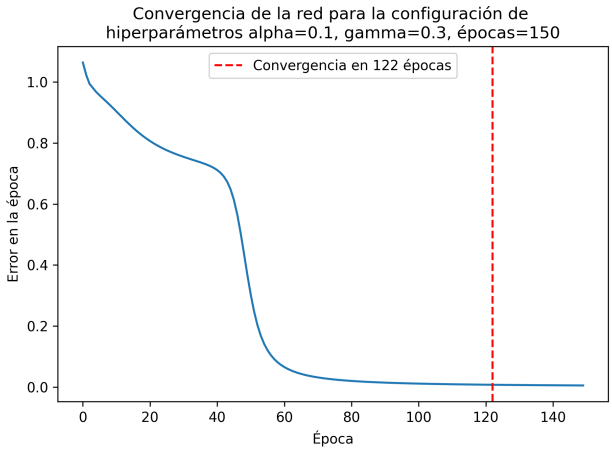
(b) Convergencia de la configuración 3

Figura 13: Curva de aprendizaje y convergencia para la configuración 3

Con la configuración 4: $\alpha = 0{,}1, \gamma = 0{,}3$; y épocas igual a 150, se puede observar en la figura 14 que, pareciera que el hiperparámetro γ no tiene influencia significativa cuando el valor α es igual a 0,1, esto debido a que no se presenta una diferencia importante con lo presentado en la figura 13.



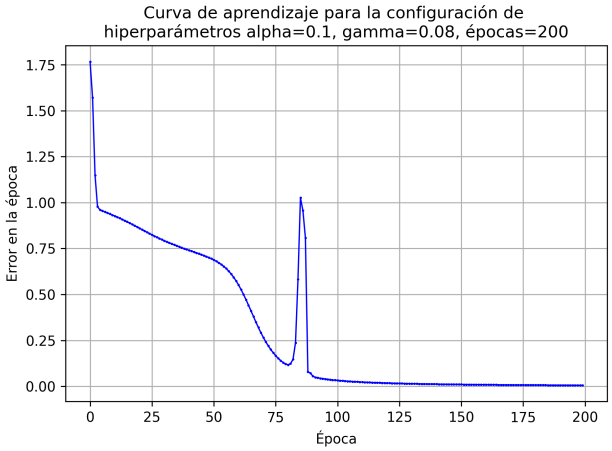
(a) Curva de aprendizaje de la configuración 4



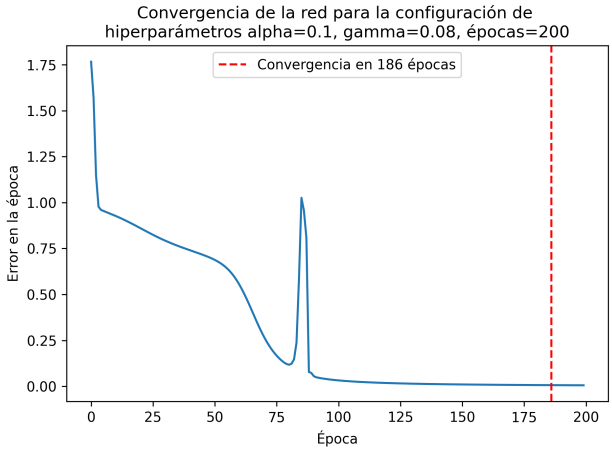
(b) Convergencia de la configuración 4

Figura 14: Curva de aprendizaje y convergencia para la configuración 4

No obstante, como podemos observar en las figuras 15, 16 y 17; para las configuraciones 5, 6 y 7; se determina que, para un $\alpha = 0{,}1$, el presentar un γ más alto, hace que la red converja en una menor cantidad de épocas, más aún con un $\gamma = 0{,}9$, como se presenta en la figura 17.

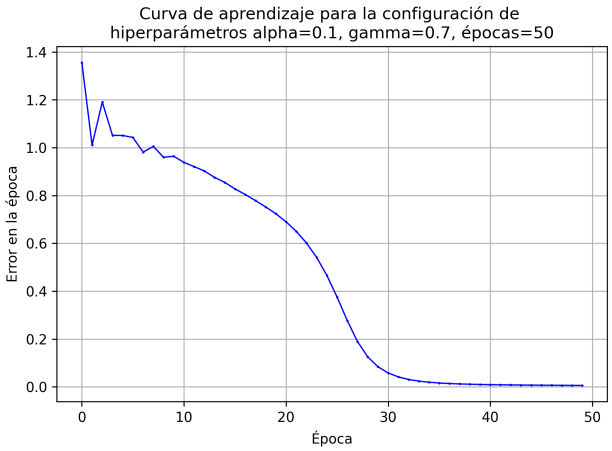


(a) Curva de aprendizaje de la configuración 5

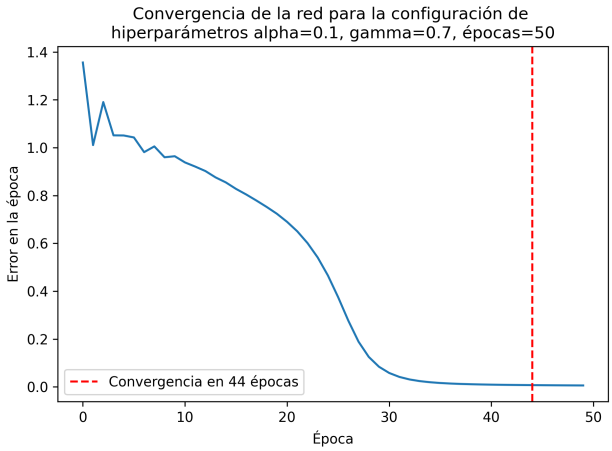


(b) Convergencia de la configuración 5

Figura 15: Curva de aprendizaje y convergencia para la configuración 5

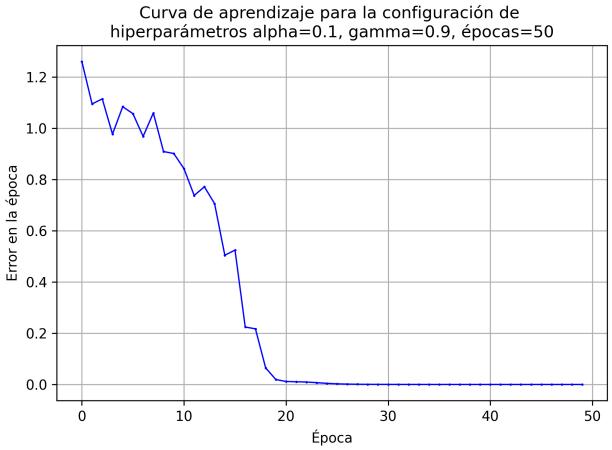


(a) Curva de aprendizaje de la configuración 6

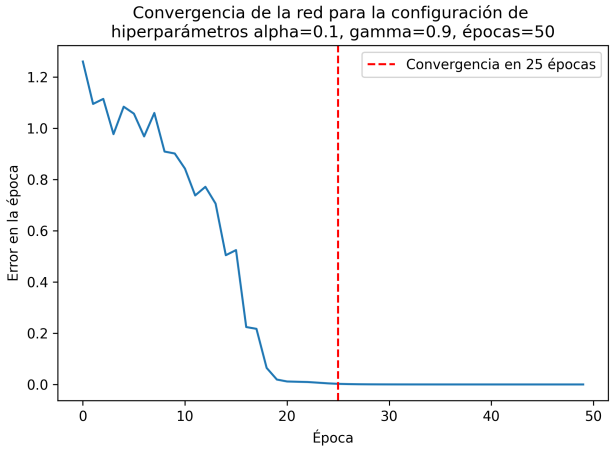


(b) Convergencia de la configuración 6

Figura 16: Curva de aprendizaje y convergencia para la configuración 6



(a) Curva de aprendizaje de la configuración 7



(b) Convergencia de la configuración 7

Figura 17: Curva de aprendizaje y convergencia para la configuración 7

Luego de las pruebas realizadas con los diferentes hiperparámetros mostrado en el cuadro 4, se puede concluir que para el problema del XOR, un $\alpha = 0{,}1$, es un valor bastante bueno para que la red aprenda y disminuya el error eficientemente; y además, probando diferentes configuraciones del γ, se puede controlar qué tan rápido la red converge, donde con un valor mayor, la red tiende a converger en menos de 50 épocas.

3. Clasificador de datos en $\mathbb{R}^2$

La presente sección busca entrenar al perceptrón multicapa para clasificar datos en un espacio $\mathbb{R}^2$. Se abordarán los siguientes dos escenarios: datos que son linealmente separables y aquellos que no lo son; para ello se hará uso de la función provista *generar_datos_R2*, para generar estos dos conjuntos de datos y posteriormente se detallará la configuración de la red para su entrenamiento.

3.1. Generación de los dos conjuntos de datos

Para crear conjuntos de datos sintéticos que se puedan utilizar en el entrenamiento, se emplea un enfoque basado en modelos de mezclas gaussianas (GMM). Este método permite generar datos con características estadísticamente controladas, como la media y la desviación estándar, que definen cómo se distribuyen los puntos de datos en el espacio bidimensional $\mathbb{R}^2$. El código implementado para la generación de datos fue brindado por el profesor del curso en el respectivo cuaderno de jupyter del trabajo práctico.

Para generar el conjunto de datos linealmente separables se usaron los parámetros presentados en el cuadro 5:

Parámetro	Clase 1	Clase 2
Media (μ_1)	$[5, 5]$	$[19, 19]$
Desviación estándar (σ_1)	$[5, 2]$	$[5, 2]$

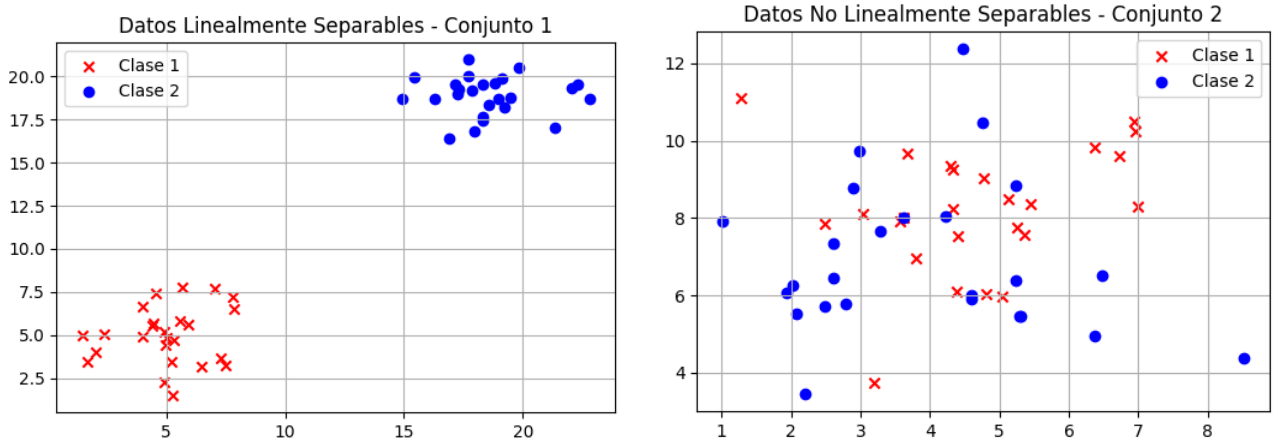
Cuadro 5: Parámetros para generar datos linealmente separables

Y de la misma forma, para generar los datos linealmente no separables, se usaron los parámetros presentados en el cuadro 6. Se debe notar que para ambos conjuntos de datos se usaron 25 observaciones por clase, es decir,

Parámetro	Clase 1	Clase 2
Media (μ_2)	$[5, 8]$	$[4, 7]$
Desviación estándar (σ_2)	$[2, 3]$	$[2, 5]$

Cuadro 6: Parámetros para generar datos no linealmente separables

cada conjunto tiene un tamaño de 50 observaciones. Las gráficas correspondientes a cada conjunto de datos se pueden observar en la figura 18:



(a) Conjunto 1: Datos linealmente separables

(b) Conjunto 2: Datos no linealmente separables

Figura 18: Gráficos de la generación de datos en $\mathbb{R}^2$.

3.2. Construcción de las redes neuronales

Dadas las indicaciones del proyecto, se procede a detallar la construcción de cuatro redes neuronales las cuales comparten los datos: $D = 2$ neuronas de entrada, y $K = 1$ neuronas en la capa de salida. Las distinciones entre cada una se documentan en el cuadro 7,

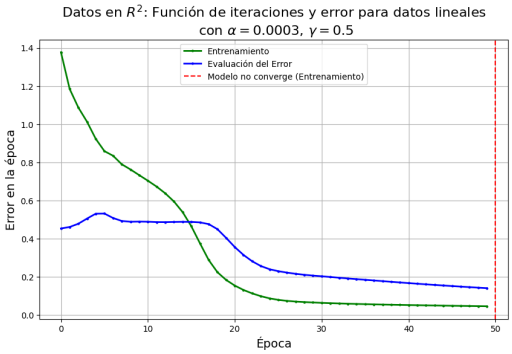
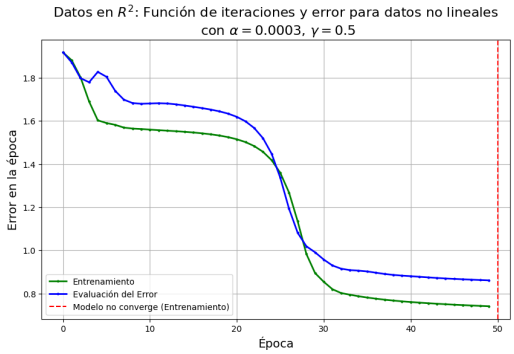
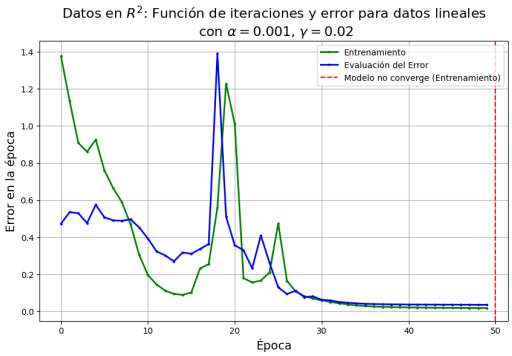
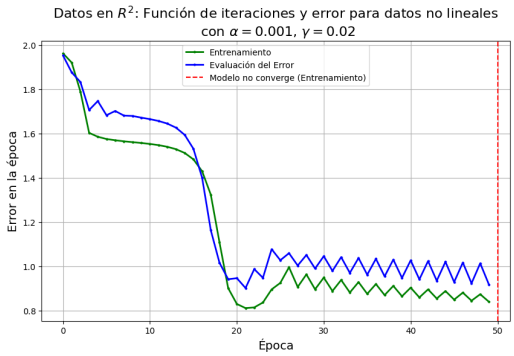
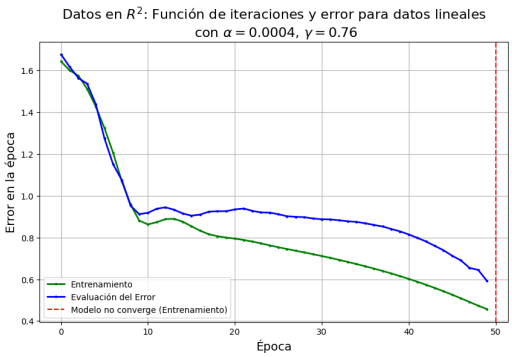
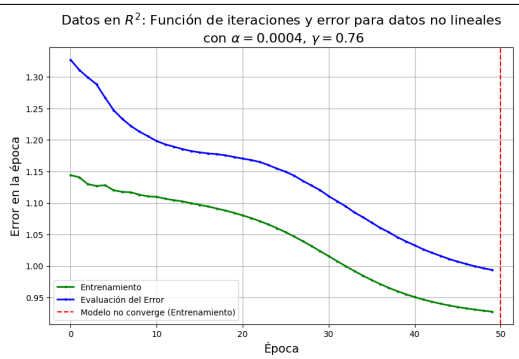
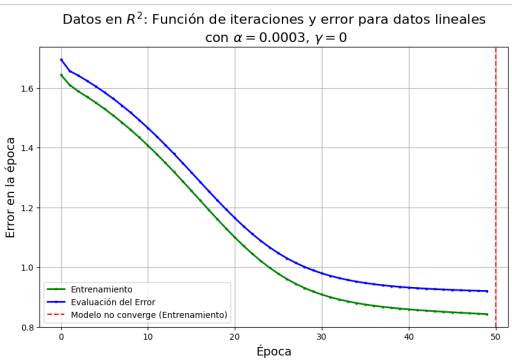
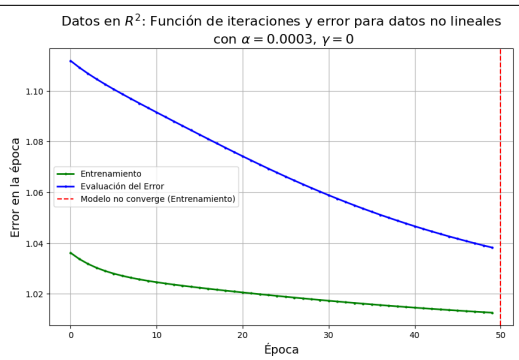
Arquitectura	M	α	γ
1	25	0.0003	0.5
2	25	0.001	0.02
3	4	0.0004	0.76
4	4	0.0003	0

Cuadro 7: Parámetros de las arquitecturas de redes neuronales

3.3. Entrenamiento y convergencia de la red

Dadas las arquitecturas definidas en el cuadro 7, se procede a entrenar cada red neuronal, tanto para el conjunto de datos linealmente separables de la figura 18a, como para el conjunto de datos no linealmente separables de la figura 18b, para $P = 50$ iteraciones. Además, usando *sklearn*, se separa cada conjunto de datos en un 80 % para el entrenamiento de la red, y un 20 % para validación.

En el cuadro 8, se presenta los resultados del entrenamiento para las cuatro arquitecturas creadas. Dentro del cuadro se separan en dos columnas las gráficas de los conjuntos linealmente separables y los no linealmente separables, además, cada fila representa la red neuronal detallada de la misma manera, y en el mismo orden, en el cuadro 7.

Arquitectura	Linealmente separables	No linealmente separables
1		
2		
3		
4		

Cuadro 8: Gráficas de resultado del entrenamiento para los datos en $\mathbb{R}^2$

Ahora bien, en el cuadro 9, se detalla cada arquitectura del cuadro 7, junto con su error final de convergencia.

Neuronas	Configuración	Hiperparámetros	Tipo	Loss
Neuronas [2, 25, 1]	1	$\alpha = 0{,}0003$, $\gamma = 0{,}5$	Lineal	0.0467
			No lineal	0.7416
	2	$\alpha = 0{,}001$, $\gamma = 0{,}02$	Lineal	0.0164
			No lineal	0.8394
Neuronas [2, 4, 1]	3	$\alpha = 0{,}0004$, $\gamma = 0{,}76$	Lineal	0.4572
			No lineal	0.9275
	4	$\alpha = 0{,}0003$, $\gamma = 0$	Lineal	0.8424
			No lineal	1.0126

Cuadro 9: Error final de convergencia para diferentes configuraciones

Luego del entrenamiento y la obtención de resultados, los cuales son detallados tanto en el cuadro 8, como en el cuadro 9, se procederá a realizar un análisis de estos.

En primer lugar, podemos observar que, para las gráficas de la **configuración 1**, tanto para los datos lineales, como los no lineales, la red pareciera que se aproxima a una convergencia, al menos para los datos lineales, los cuales presentan un error final de 0,0467, pero que al mismo tiempo, los datos no lineales siguen una misma tendencia donde su error baja de una manera similar pero un poco más lenta. Para esta configuración se usan 25 neuronas en la capa oculta, junto con un $\alpha = 0{,}0003$ y un $\gamma = 0{,}5$.

La **configuración 2** es un poco similar a la **configuración 1**, no obstante, se usa un α más grande, lo que pareciera que, para los datos lineales, ayuda a mejorar la convergencia, pero que para los no lineal, no existe un cambio muy significativo.

Ahora bien, para la **configuración 3** y **configuración 4**, se usa menos neuronas en la capa oculta, pero de la misma manera, se sigue usando un α muy similar a la **configuración 1**. Con esta combinación se puede notar que el error final de convergencia es mucho mayor para los datos lineales y no lineales.

Con estos resultados se puede decir que la cantidad de neuronas en la capa oculta afecta a la convergencia de la red de una manera bastante significativa, que además, donde a mayor número de neuronas, un α más pequeño es necesario, esto debido a que, si se observa las gráficas del cuadro 8, se puede ver una diferencia muy grande entre las gráficas de las primeras dos filas, con las filas 3 y 4. Donde las primeras dos filas presentan una convergencia más cercana a 0 que las otras dos. Dada la complejidad de los datos no lineales, quizás se presenta lo que se conoce como **under fitting** [4] en la red neuronal. Y de igual manera, el presentar una enorme cantidad de neuronas en la capa oculta, puede llevar a un **sobreajuste** o incluso que se tome una gran cantidad de tiempo para procesar los datos [4].

Para lidiar con los problemas mencionados anteriormente, existen métodos para escoger la cantidad correcta de neuronas en la capa oculta. Un punto de inicio puede ser:

- El número de neuronas de la capa oculta debería estar entre el tamaño de la capa de entrada y el tamaño de la capa de salida.
- El número de neuronas de la capa oculta debería ser 2/3 del tamaño de la capa de entrada, más el tamaño de la capa de salida.
- El número de neuronas de la capa oculta debería ser menor que el doble del tamaño de la capa de entrada. [4]

Si nos basamos en alguno de los puntos anteriores, por ejemplo, segundo punto, podríamos entonces escoger un número de neuronas para la capa oculta de la siguiente manera:

$$M = \frac{2}{3}(D + K),$$

Para nuestro caso, tenemos que, $D = 2$ y $K = 1$, por ende:

$$\begin{aligned} M &= \frac{2}{3}(2 + 1) \\ M &= \frac{2}{3} \cdot 3 \\ M &= 2 \end{aligned}$$

Entonces se puede decir que, existen métodos para poder determinar el número de neuronas en la capa oculta dado el número de neuronas en la capa de entrada. Además, el coeficiente de aprendizaje α es dependiente del número de neuronas en la capa oculta de igual manera, ya que, de usar un menor número de neuronas, se puede tomar un α inclusive más pequeño, esto para que el modelo no realice 'saltos' muy grandes en la búsqueda de un mínimo.

4. Clasificador de ataques usando datos estructurados

La presente sección busca entrenar al perceptrón multicapa para clasificar datos estructurados obtenidos del dataset *UNSW-NB15: A Comprehensive Data set for Network Intrusion Detection systems*, codificando los datos mediante la técnica *One-Hot Vector*. Para ello, fue necesario preparar los datos a través de la biblioteca Pandas, tomando de referencia códigos de ejemplo brindados por el profesor del curso. En la presente sección se cubrirá el procesamiento de los datos y el entrenamiento respectivo.

4.1. Procesamiento de datos para la codificación

Para realizar el procesamiento de los datos, se utilizó la biblioteca `Pandas`, que brinda herramientas para para importar el dataset y transformarlo de manera tal que se utilicen únicamente datos relevantes para el ejercicio, así como modificar la cardinalidad de algunas características con el fin reducir las características codificadas para el método *one hot vector*.

```
1  import numpy as np
2  import pandas as pd
3
4  # Carga de los datos
5  df = pd.read_csv('/content/UNSW_NB15_training-set.csv')
6
7  # Elimina columnas no necesarias
8  list_drop = ['id','attack_cat']
9  df.drop(list_drop, axis=1,inplace = True)
10
11 # Descripción de datos categóricos
12 df.describe(exclude=np.number)
```

Figura 19: Carga de datos del dataset en un dataframe.

En la figura 19 se muestra como se cargaron los datos del dataset completo, además en la línea 12 se realiza un filtro donde se seleccionan únicamente registros categóricos al excluir datos numéricos, a continuación en el cuadro 10 se puede ver una descripción de dichas variables.

	proto	service	state
count	82332	82332	82332
unique	131	13	7
top	tcp	-	FIN
freq	43095	47153	39339

Cuadro 10: Descripción de los datos categóricos del dataframe.

Del cuadro anterior, se puede evidenciar cómo las tres categorías tienen 82332 registros y la categoría "`proto`" (protocolo) es quién más tipos de valores únicos presenta, la intención es reducir la cantidad de valores únicos previo a la codificación *one hot vector*. Para ello se implementará la siguiente estrategia:

- Preservar sólo aquellos registros con valores dentro del top 5 de los valores más frecuentes del dataset. Esto realizado con los métodos `value_counts() y head()`, que retornan, respectivamente; una array ordenado de mayor a menor con la suma de datos por categoría, y los primeros cinco datos del array anterior.
- Reemplazar con "-" si el valor no está entre el top 5 más frecuente

Con estas estrategias se reduce significativamente la cantidad de valores únicos, debido a que se conservan sólo aquellos que sean parte del top 5 más frecuente, a continuación, el código implementado que realiza la reducción:

```
# Filtrar solo pro datos categóricos
df_cat = df.select_dtypes(exclude=[np.number])

for feature in df_cat.columns:
    if df_cat[feature].nunique() > 5:
        df[feature] = np.where(df[feature].isin(top5.index), df[feature], '-')

df.describe(exclude=np.number)
```

Figura 20: Reducción de los valores únicos de las categorías según su top5 de datos más frecuentes.

Una vez hecha la reducción de la figura 20, podemos nuevamente mostrar la descripción de los datos categóricos y evidenciar una notoria reducción de sus valores únicos, la cuál se muestra continuación:

	proto	service	state
count	82332	82332	82332
unique	6	5	6
top	tcp	-	FIN
freq	43095	49275	39339

Cuadro 11: Descripción de los datos categóricos del dataframe con valores únicos reducidos.

Nótese del cuadro 11 que ahora la cantidad de valores únicos para las características categóricas del dataset son mucho menores que en el cuadro 10.

4.2. Codifiación one hot vector

Para realizar codificación se utilizó la biblioteca `scikit-learn`, que brinda herramientas para utilizar un transformador que convierte variables categóricas en una representación numérica, que es más adecuada para modelos de aprendizaje automático, creando una nueva variable binaria para cada categoría.

```
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder

# X representa las categorías a codificar, Y representa los targets
X = df.iloc[:,:-1]
y = df.iloc[:,-1]

# Crear el transformador one-hot encoder y transforma los datos
transform_columns = [1, 2, 3]
ct = ColumnTransformer(transformers=[('encoder',OneHotEncoder(), transform_columns)],
                       remainder='passthrough'
    )
X = np.array(ct.fit_transform(X))
```

Figura 21: Implementación de la codificación One Hot vector

En la figura 21 se crea una instancia de ColumnTransformer, dicha instancia está configurada para aplicar la codificación `OneHotEncoder` a las columnas especificadas (índices 1, 2 y 3 del DataFrame X). Las columnas que no están especificadas para ser transformadas por el OneHotEncoder serán dejadas intactas gracias al parámetro `remainder="passthrough"`. Esto significa que las columnas no listadas en los índices [1, 2, 3] se añadirán sin cambios al resultado final tras la transformación. Luego, el resultado de `ct.fit_transform(X)` se convierte a un array de NumPy para posiblemente facilitar operaciones futuras, dado que muchos algoritmos en scikit-learn esperan datos en este formato.

4.2.1. Procesamiento de datos para entrenar el modelo

Una vez obtenida la codificación, es necesario ajustar los datos para empezar el entrenamiento, estos ajustes constan de convertir a tensores los datos obtenidos DataFrame X, agregar una columna de 1's al inicio para el sesgo y cambiar los valores en 0 por -1. A continuación el código implementado para ello:

```python
from sklearn.model_selection import train_test_split

def add_ones_column(input):
    ones_column = np.ones((input.shape[0], 1))
    return torch.tensor(np.hstack((ones_column, input)), dtype=torch.float)

def swap_zeros(input):
    input = torch.tensor(input.values, dtype=torch.float)
    input[input == 0] = -1
    return input.unsqueeze(1)

# Dividir los datos en conjuntos de entrenamiento y validación
# Se usa el 80% para entrenar el modelo y el 20% para evaluarlo el modelo
data = train_test_split(X, y, test_size = 0.2, random_state = 0, stratify=y)
X_train, X_test, T_train, T_test = data

# Crear una columna de unos
X_train = add_ones_column(X_train)
X_test = add_ones_column(X_test)

# Cambiar 0 por -1
T_train = swap_zeros(T_train)
T_test = swap_zeros(T_test)
```

Figura 22: Procesamiento de datos previo al entrenamiento con el modelo

Para concluir, de la figura 22, puede verse como nuevamente, al igual que en los otros entrenamientos realizado, se utiliza una relación de 80-20 para entrenamiento y pruebas, en la siguiente sección se detalla el entrenamiento a realizar.

4.3. Entrenamiento de la red

Dadas las indicaciones del presente trabajo, es necesario ahora definir la cantidad de neuronas D que tendrá la capa de entrada, así como la cantidad de neuronas K que tendrá la capa de salida, para ello se estudiaron las dimensiones de los datos, y se tomó la decisión de tomar $D = 57$ neuronas de entrada, siendo 57 el número de columnas de los datos, y $K = 1$ fue decido dadas las recomendaciones vistas en clase. A continuación se muestra los parámetros de la red y la magnitud de error mínima alvanzada:

Neuronas [D, M, K]	α	γ	Loss
[57, 17, 1]	$1e^{-7}$	0	0.9929

Cuadro 12: Error final de convergencia para el entrenamiento de datos estructurados

De acuerdo a la tabla 12, se muestra ahora la gráfica de los errores a través de las 50 épocas de entrenamiento:

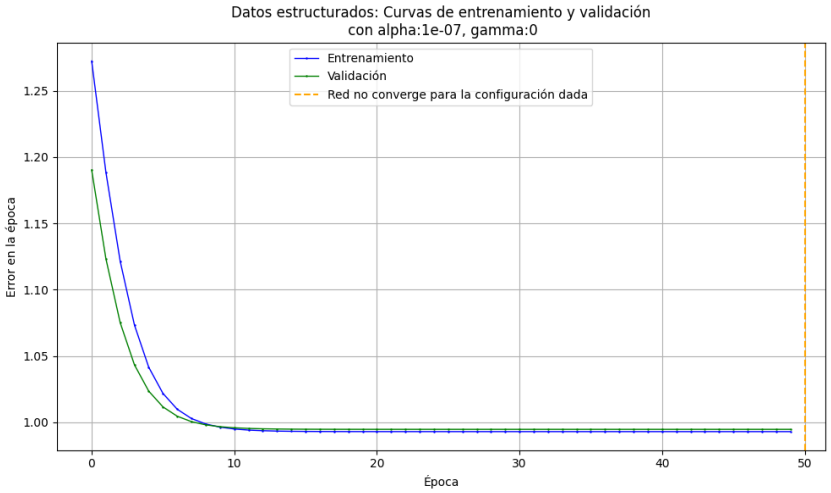


Figura 23: Errores a través de 50 épocas del entrenamiento con datos estructurados

Como se puede evidenciar en la figura 23, los errores parecen acercarse a cero desde la época 10 o inclusive antes, esto muestra como el perceptrón implementado fue capaz de aprender de este conjunto de datos de manera satisfactoria.

4.4. Optimización de hiper-parámetros con optuna

Adicionalmente al entrenamiento se decidió implementar una optimización de hiper-parámetros con la biblioteca optuna, el cuál fue implementado de la siguiente forma:

```
def objective(trial):
    # Generar candidatos de hiperparámetros con Optuna
    D = X_train.shape[1]
    M = trial.suggest_int('M', 1, 20)  # Número de neuronas en la capa oculta
    alpha = trial.suggest_float('alpha', 1e-10, 1e-1, log=True)
    gamm = 0
    params = [M, alpha, gamm]

    # Creamos y entrenamos el modelo con los parámetros seleccionados
    _, errors_train, errors_test, epoch_converge = train_structured_data(50, X_train, T_train,
     X_test, T_test, params, print_e=False)

    # Minimizamos el error de validación final
    return errors_train[-1]  # suponiendo que errors_test contiene el error de validación en cada época

# Definir el estudio de Optuna y número de trials
study = optuna.create_study(direction='minimize')
study.optimize(objective, n_trials=100)

```

Figura 24: Optimización de hiper-parámetros con optuna

Como resultado de haber realizado un entrenamiento por 100 intentos, de la figura 24 se obtuvieron los siguientes mejores parámetros parámetros y pérdida:

Neuronas [D, M, K]	α	γ	Loss
[57, 3, 1]	$1{,}4025e^{-6}$	0	0.9899

Cuadro 13: Error final de convergencia para el entrenamiento de datos estructurados

Nótese que hay una ligera diferencia entre los errores mínimos del cuadro 12 y el cuadro 13. Siendo del segundo cuadro un error menor el resultante de los parámetros con optuna. Cabe mencionar que existe una significativa diferencia entre los valores de M, siendo ahora 14 veces menor, esto repercute en la cantidad de pasos necesarios para calcular los resultados, ya que, al ser menos dimensiones son menos multiplicaciones.

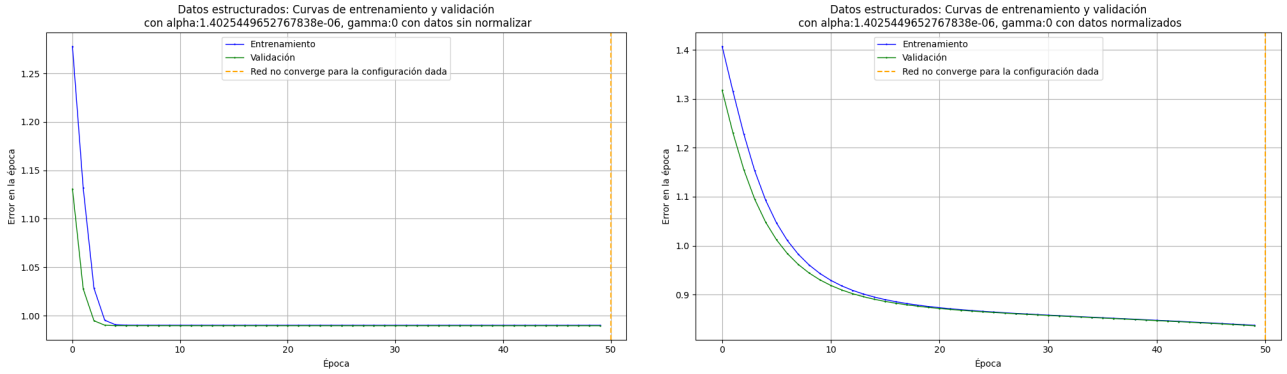
4.5. Normalización de los datos de entrada

Se realizó la normalización de 0 a 1, utilizando el valor máximo de la observación, esto calculado de la siguiente forma:

```
ef normalize_data(X):
    # Asegura que X es un tensor de PyTorch
    X = torch.as_tensor(X, dtype=torch.float32)

    # Calcula el valor máximo por observación (a lo largo de cada fila si X es una matriz 2D)
    X_max = X.abs().max(dim=1, keepdim=True)[0]

    # Evita la división por cero ajustando los valores máximos que sean cero a uno
    X_max[X_max == 0] = 1

    # Normaliza los datos
    return X / X_max

P = 30

# Normalizar los datos de entrenamiento y prueba
X_train_norm = normalize_data(X_train)
X_test_norm = normalize_data(X_test)

M = study.best_trial.params["M"] # número de neuronas en la capa oculta
alph, gamm = study.best_trial.params["alpha"], 0
params = [M, alph, gamm]

train = train_structured_data(P, X_train, T_train, X_test, T_test, params, print_e=False)
train_norm = train_structured_data(P, X_train_norm, T_train, X_test_norm, T_test, params,
    print_e=False)

```

Figura 25: Normalización de los datos de entrada.

De manera tal que se notó el siguiente comportamiento un entrenamiento de 30 corridas del algoritmo:



(a) Gráficas de error con datos sin normalizar

(b) Gráficas de error con datos normalizados

Figura 26: Comparativa de las curvas de error entre datos no normalizados y normalizados.

De la anterior figura se puede evidenciar cómo al normalizar los datos, existe una mayor suavidad en las curvas de aprendizaje de los entrenamientos, así como una disminución en los errores:

Nótese de la figura anterior como los datos normalizados presentan un menor error.

Trial	min(training)	min(testing)
30	0.990004	0.989558

(a) Errores con datos sin normalizar

Trial	min(training)	min(testing)
30	0.859436	0.858770

(b) Errores con datos normalizados

Figura 27: Comparativa de los errores entre datos no normalizados y normalizados.

Referencias

[1] R. Kruse, C. Borgelt, C. Braune, S. Mostaghim, and M. Steinbrecher, "Introduction to neural networks," pp. 9–13, 2016.

[2] H. Sompolinsky, "Statistical mechanics of neural networks," *Physics Today*, vol. 41, pp. 70–80, 1988.

[3] C. M. Bishop, "Pattern recognition and machine learning," *Springer google schola*, vol. 2, pp. 645–678, 2006.

[4] G. Panchal, A. Ganatra, Y. Kosta, and D. Panchal, "Behaviour analysis of multilayer perceptrons with multiple hidden neurons and hidden layers," *International Journal of Computer Theory and Engineering*, vol. 3, no. 2, pp. 332–337, 2011.