

Reporte de Práctica 1

La siguiente práctica consiste en realizar la implementación de un Perceptrón Multicapa (MLP) desde cero y usarlo para resolver problemas de diversa índole como simular la compuerta lógica XOR o diferenciar entre tipos de Vino. Los MLP son una técnica de aprendizaje profundo en Inteligencia Artificial los cuales son capaces de resolver problemas no lineales; la principal limitación de su predecesor el perceptrón. Los MLP simulan las neuronas biológicas mediante el procesamiento de la información en cada neurona y una función de activación que simula los impulsos eléctricos que activan la neurona biológica, esto a su vez se conecta con otras neuronas formando una red neuronal que se conforma de varias capas.

Funcionamiento de un perceptrón multicapa

El funcionamiento de un MLP comienza por la capa de neuronas de entrada realizando el proceso conocido como **Propagación hacia adelante**, esta capa recibe directamente la información que se procesa, sea imagen o texto, siempre representan entradas numéricas al final. Estas entradas son acomodadas en una matriz X de la forma:

$$X = \begin{pmatrix} x_{11} & x_{12} & \dots & x_{1j} \\ x_{21} & x_{22} & \dots & x_{2j} \\ \vdots & \vdots & \ddots & \vdots \\ x_{i1} & x_{i2} & \dots & x_{ij} \end{pmatrix}$$

Donde:

i = Neurona de la capa de entrada

j = Entrada de datos

Luego cada dato de esta matriz es multiplicada por un peso w que representa su importancia en cada neurona, esto también puede ser representado como matriz, que en este caso almacena los parámetros de las neuronas que sirven para determinar su activación.

$$W = \begin{pmatrix} w_{11} & w_{12} & \dots & w_{1j} \\ w_{21} & w_{22} & \dots & w_{2j} \\ \vdots & \vdots & \ddots & \vdots \\ w_{i1} & w_{i2} & \dots & w_{ij} \end{pmatrix}$$

Donde:

i = Neurona de la capa de entrada

j = Entrada de datos

Finalmente es considerado un sesgo (bias) que cada neurona pueda tener, que de igual manera se puede almacenar en una matriz B .

$$B = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_i \end{pmatrix}$$

De manera que los cálculos que realiza toda una capa de la red neuronal son determinados por una sola ecuación.

$$Z = XW^T + B$$

Z representa los cálculos que cada una de las neuronas hace para obtener el valor de pre-activación. Estos valores son finalmente pasado a una función de activación $\sigma(z)$ que determina si la neurona se activa o en qué medida lo hace. Funciones de este tipo existen muchas, pero en esta práctica se utilizó únicamente la función sigmoide, definida a continuación.

$$\sigma(z) = \frac{1}{1+e^{-z}}$$

El valor de esta activación es pasado a las capas siguientes como si fueran las nuevas X , repitiendo el proceso cada vez hasta llegar a la capa final en donde el resultado representa la predicción del modelo; por lo que cada una de las capas tiene sus propias matrices de pesos W y sesgos B donde la capa se denota como un superíndice. Para esta práctica se utilizó una **arquitectura de 2 capas**.

```
def forward(self, X):
    """
    Realiza el Forward pass
    """
    if self.normalize != None:
        self.X = self.normalize(X)
    else:
        self.X = X
    self.z_c1 = self.X @ self.W1.T + self.B1
    self.B1 = np.sum(self.B1, axis=0, keepdims=True)
    self.a_c1 = self.activate(self.z_c1)

    self.z_c2 = self.a_c1 @ self.W2.T + self.B2
    self.B2 = np.sum(self.B2, axis=0, keepdims=True)
    self.y_pred = self.activate(self.z_c2)
    return self.y_pred
```

Nótese que para la implementación de la propagación hacia adelante, por el broadcasting que realiza numpy internamente con las matrices, es necesario “aplanar” las matrices $B^{(k)}$ a un vector.

Al comienzo del proceso la matriz W debe ser inicializada con valores, en esta práctica se probaron 2 métodos distintos de inicialización. El primero es la **inicialización normal** que consiste en tomar valores aleatorios, como si se tratara de un muestreo aleatorio de una distribución normal estándar. El segundo método es llamado **inicialización de Xavier** y solo se diferencia en que la muestra es obtenida de una distribución normal con la siguiente desviación estándar $\sigma = \sqrt{\frac{2}{inputs+outputs}}$, donde inputs y outputs son la cantidad de neuronas de entrada y de salida de la capa de la red neuronal respectivamente. En el siguiente código están ambos métodos de inicialización, es útil usar un flotante de 128 bits pues en las operaciones realizadas la precisión extra es necesaria.

```
def xavier_initialization(input_size, output_size):
```

```

array = np.random.normal(0, np.sqrt(2 / input_size), (input_size, output_size))
return np.float128(array)

def normal_distribution_initialization(input_size, output_size):
    array = np.random.normal(0, 0.1, (input_size, output_size))
    return np.float128(array)

```

Tambien es útil normalizar los datos de entrenamiento pues en problemas complejos los números de entrada de la red neuronal, pueden ser de grados de magnitud muy dispares, lo que empeora su calidad, el método que se usará en esta práctica es la **normalización Z-score** que se define como:

$$z(x) = \frac{x - \mu}{\sigma}$$

Donde:

x = Dato a normalizar

μ = Media de todo el conjunto de datos

σ = Desviación estándar de todo el conjunto de datos

El segundo paso de una red neuronal es el **cálculo del error**, esto indica al modelo cuál era el resultado esperado de las operaciones, este tipo de técnica es denominado *aprendizaje supervisado*. Existen varios métodos para calcular el error, en esta práctica se utilizó el **MSE** (Error cuadrático medio) que se calcula como:

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

Donde:

\hat{y} = Resultado del modelo

y = Resultado real

El error calculado es lo que hace que la red neuronal sea capaz de aprender, pues nos permite determinar cómo ajustar los parámetros de las matrices W y B de la red mediante la **propagación hacia atrás**. Este proceso involucra calcular el gradiente de la función de error; visto de otra forma MSE es una función que depende de la variable \hat{y} , que a su vez es una función que depende de las activaciones a_i , de los pesos w_{ij} y de los sesgos b_i de cada neurona de la penúltima capa. Es posible entonces continuar esta lógica hacia atrás hasta llegar a que la función de error depende de todos los parámetros utilizados en toda la red.

Este hecho es muy importante dado que el gradiente de una función nos permite encontrar el mínimo y en este caso, minimizar el error significa hacer que la red neuronal aprenda a hacer mejores predicciones. Calcular el componente del gradiente para un parámetro de una última capa se define con una regla de la cadena. Cabe destacar que la segunda derivada parcial cuenta con propiedades especiales que reducen el costo computacional.

$$\begin{aligned}
E &= MSE(W, B) \\
\frac{\partial E}{\partial w_{ij}^{(2)}} &= \frac{\partial E}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z_i^{(2)}} \frac{\partial z_i^{(2)}}{\partial w_{ij}^{(2)}} \\
\frac{\partial E}{\partial b_i^{(2)}} &= \frac{\partial E}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z_i^{(2)}} \frac{\partial z_i^{(2)}}{\partial b_i^{(2)}}
\end{aligned}$$

$$\frac{\partial \hat{y}}{\partial z_i^{(2)}} = \frac{\partial \sigma(z_i^{(2)})}{\partial z_i^{(2)}} = \sigma(z_i^{(2)})(1 - \sigma(z_i^{(2)}))$$

Del mismo modo para calcular el componente del gradiente de un parámetro de la primera capa se obtiene con la regla de la cadena, en este caso es posible notar la repetición de algunos términos por lo que es útil almacenarlos en una variable $\delta_i^{(2)}$ para reducir el costo computacional.

$$\begin{aligned} \frac{\partial E}{\partial w_{ij}^{(1)}} &= \frac{\partial E}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z_i^{(2)}} \frac{\partial z_i^{(2)}}{\partial a_i^{(1)}} \frac{\partial a_i^{(2)}}{\partial z_i^{(1)}} \frac{\partial z_i^{(1)}}{\partial w_{ij}^{(1)}} \\ \frac{\partial E}{\partial b_i^{(1)}} &= \frac{\partial E}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z_i^{(2)}} \frac{\partial z_i^{(2)}}{\partial a_i^{(1)}} \frac{\partial a_i^{(2)}}{\partial z_i^{(1)}} \frac{\partial z_i^{(1)}}{\partial b_i^{(1)}} \\ \delta_i^{(2)} &= \frac{\partial E}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z_i^{(2)}} \\ \delta_i^{(1)} &= \delta_i^{(2)} \frac{\partial z_i^{(2)}}{\partial a_i^{(1)}} \frac{\partial a_i^{(2)}}{\partial z_i^{(1)}} \end{aligned}$$

El proceso podría continuar hacia atrás si la red neuronal tuviera más capas. El proceso completo de propagación se ve implementado en la siguiente función:

```
def backward(self, y):
    """
    Realiza la Backpropagation
    """
    self.dE_dy_pred = self.y_pred - y
    self.dy_pred_dz_c2 = self.activation_derivative(self.y_pred)

    self.delta_c2 = self.dE_dy_pred * self.dy_pred_dz_c2
    self.grad_c2 = self.delta_c2.T @ self.a_c1

    self.delta_c1 = self.delta_c2 @ self.W2 * self.activation_derivative(self.a_c1)
    self.grad_c1 = self.delta_c1.T @ self.X
```

Una vez calculado el gradiente ΔE , el último paso consiste en **actualizar los parámetros** dados los valores obtenidos con el gradiente, haciendo que el modelo aprenda a base de modificar los sesgos y los pesos hacia unos que minimicen el error. La velocidad con la que estos cambian es denominada *tasa de aprendizaje* η .

$$\begin{aligned} W_{ij}^{(k)} &\leftarrow W_{ij}^{(k)} - \eta \frac{\partial E}{\partial w_{ij}^{(k)}} \\ b_i^{(k)} &\leftarrow b_i^{(k)} - \eta \frac{\partial E}{\partial b_i^{(k)}} \end{aligned}$$

La implementación de este paso a continuación:

```

def update(self):
    """
        Re-ajusta los parámetros de la red
    """
    self.W2 = self.W2 - self.learning_rate * self.grad_c2
    self.B2 = self.B2 - self.learning_rate * np.sum(self.delta_c2, axis=0,
keepdims=True)

    self.W1 = self.W1 - self.learning_rate * self.grad_c1
    self.B1 = self.B1 - self.learning_rate * np.sum(self.delta_c1, axis=0,
keepdims=True)

```

Entrenamiento y prueba de un perceptrón multicapa

Para entrenar una red neuronal lo único que hay que hacer es repetir los 4 pasos principales: propagación hacia adelante, cálculo del error, propagación hacia atrás y actualización de parámetros. Cada iteración es llamada *época* y cada vez se hace el proceso de entrenamiento con *batch* de datos: una parte pequeña de los datos disponibles para entrenar. Típicamente esto es suficiente, pero en este caso se implementó un paso extra para dividir el batch en *minibatches* para probar si esto tiene algún impacto en el desempeño de la red. La cantidad de épocas se establece arbitrariamente, siempre con el objetivo de que sean suficientes para hacer converger el error a un valor pequeño, en este caso las épocas fueron establecidas en 2000. Otra posibilidad es implementar un criterio de paro, cuando el error represente un valor muy pequeño, pero esto no fue incluido como parte de los experimentos.

```

def train(self, X, Y):
    """
        Entrena a la red neuronal con las features X y el target Y
        1.- Forward Pass
        2.- Cálculo del error
        3.- Backpropagation
        4.- Actualización de parámetros
    """

    self.epochs_error = []
    if self.normalize != None:
        self.X = self.normalize(X)
    else:
        self.X = X
    for epoch in range(self.epochs):
        num_batch = 0
        epoch_error = 0
        for X_batch, y_batch in self.create_minibatches(X, Y, self.batch_size):
            self.y_pred = self.forward(X_batch)
            error = self.loss_function(self.y_pred, y_batch)
            epoch_error += error
            self.backward(y_batch)
            self.update()
            num_batch += 1
        self.epochs_error.append(epoch_error/num_batch)
        # Imprimir el error cada N épocas
        if epoch % 100 == 0:
            print(f"Época {epoch}, Error: {epoch_error/num_batch}")

```

Finalmente una vez completado el entrenamiento, se puede probar la calidad de las predicciones del modelo alimentando entradas a la propagación hacia adelante, idealmente se usan nuevos datos que el modelo no haya visto antes de los cuales se conozca su respuesta correcta; luego se mide su precisión o **accuracy** dividiendo la cantidad de aciertos del modelo entre el total de entradas proporcionadas.

```
def predict(self, X, y):
    """
    Predice como un clasificador binario y establece el accuracy
    """
    self.y_pred = self.forward(X)
    self.y_pred = np.where(self.y_pred >= 0.5, 1, 0)

    errores = np.count_nonzero(self.y_pred - y)
    aciertos = y.shape[0] - errores
    self.accuracy = aciertos / y.shape[0]
    return self.y_pred
```

Para tener todos estos métodos organizados se creó la **clase MultiLayerPerceptron**, la cual se inicializa con la cantidad de neuronas de cada una de las 2 capas; las estrategias de inicialización, normalización, activación y cálculo del error; y parámetros extra como la tasa de aprendizaje y un número para inicializar la semilla de aleatoriedad.

```
class MultiLayerPerceptron:
    def __init__(self,
                 num_entradas,
                 num_neuronas_ocultas,
                 num_salidas, epochs,
                 initializazion_function,
                 activation_function,
                 activation_derivative,
                 loss_function,
                 normalization_function=None,
                 batch_size=128,
                 learning_rate=0.2,
                 random_state=42):
        ...
    def forward(self, X):
        ...
    def backward(self, y):
        ...
    def update(self):
        ...
    def predict(self, X, y):
        ...
    def train(self, X, Y):
        ...
```

Diseño de los experimentos

Se entrenará un perceptrón multicapa por cada combinación de estrategias y parámetros, en total 288 posibilidades que se resumen en la siguiente tabla.

	Parámetro	Posibilidades
Constantes	Épocas	2000
	Métrica de desempeño	Accuracy
	Estrategia de cálculo del error	MSE
	Estrategia de activación	Sigmoide
Variables	Estrategia de inicialización	Inicialización normal, Inicialización de Xavier
	Estrategia de normalización	Normalización Z-score, Sin normalización
	Número de neuronas de la capa oculta	2, 4, 8, 16, 32, 128
	Tasa de aprendizaje	0.01, 0.1, 0.5
	Tamaño de minibatches	8, 16, 32, 64

También se experimentaron con distintos conjuntos de datos conocidos denominados *datasets*, que varían en complejidad:

1. **Compuerta lógica XOR** (2 entradas, 1 salida): El problema más sencillo de todos, aunque no es un dataset como tal, se utilizó para probar el funcionamiento inicial del MLP, consiste en entrenar la red neuronal para predecir el resultado de la compuerta XOR, comprobando que se puede resolver este problema al contrario que con los perceptrones comunes.
2. **Plantas Iris** (4 entradas, 1 salida): Este problema consiste en distinguir entre 2 tipos de plantas o 2 clases, usando datos de 4 características relacionadas con la forma de cada planta. El modelo deberá producir la salida 0 si la planta es Setosa o 1 si es Versicolor.
3. **Cáncer de mama** (30 entradas, 1 salida): Este es el dataset más complejo, se trata de distinguir entre un tumor maligno (clase 0) y un tumor benigno (clase 1); las 30 características describen los tumores en forma,, simetría, textura y más.
4. **Vinos** (13 entradas, 1 salida): Consiste en reconocer 2 tipos de vinos italianos, Barolo representado por 0 y Grignolino representado por 1, utilizando 13 características que describen su composición química.

Para todos los datasets se realizaron los 288 experimentos antes mencionados, cabe destacar que en todos los casos la cantidad de datos de entrenamiento y prueba es distinta, pero siempre están balanceados, es decir, que hay la misma cantidad de ejemplos de cada una de las clases; todos estos datos se almacenan en archivos csv, por lo que leer los datos y ordenarlos en matrices para que sean compatibles con el MLP es un proceso muy sencillo.

```
# -----
# Dataset Iris
# -----
data = np.genfromtxt('./datasets/iris_train.csv', delimiter=',', skip_header=1)
X_iris_train = data[:, :-1]
```

```

Y_iris_train = data[:, -1]
Y_iris_train = Y_iris_train.reshape(len(Y_iris_train),1)
data = np.genfromtxt('./datasets/iris_test.csv', delimiter=',', skip_header=1)
X_iris_test = data[:, :-1]
Y_iris_test = data[:, -1]
Y_iris_test = Y_iris_test.reshape(len(Y_iris_test),1)
...

```

Para mantener el control sobre todos los experimentos, tras entrenar un modelo se guardó un diccionario indicando sus variables particulares y se almacenaron en una lista de tuplas (diccionario, modelo).

```

# -----
# Datos de entrenamiento
# -----
X_train = [X_xor, X_iris_train, X_breast_cancer_train, X_wine_train]
Y_train = [Y_xor, Y_iris_train, Y_breast_cancer_train, Y_wine_train]
X_test = [X_xor, X_iris_test, X_breast_cancer_test, X_wine_test]
Y_test = [Y_xor, Y_iris_test, Y_breast_cancer_test, Y_wine_test]
neuronas_ocultas = [2, 4, 8, 16, 32, 128]
salidas = 1
batch_size = [8, 16, 32, 64]
learning_rate = [0.01, 0.1, 0.5]
epochs = 2000
initialization_functions = [xavier_initialization, normal_distribution_initialization]
normalization_functions = [z_score_normalization, None]
nombres_datasets = ['XOR', 'Iris', 'Breast Cancer', 'Wine']
nombres_inicializacion = ['Xavier', 'Normal distribution']
nombres_normalizacion = ['Z-score', 'Ninguna']

# -----
# Realizar 288 Experimentos por cada Dataset
# -----
lista_clasificadores: list[tuple] = []
for i in range(len(X_train)):
    for i_f_i in range(len(initialization_functions)):
        for i_f_n in range(len(normalization_functions)):
            for n in neuronas_ocultas:
                for lr in learning_rate:
                    for b in batch_size:
                        diccionario = dict()
                        diccionario = {
                            'Dataset': nombres_datasets[i],
                            'Initialization':nombres_inicializacion[i_f_i],
                            'Normalization':nombres_normalizacion[i_f_n],
                            'Activation':'Sigmoid',
                            'Input-size':len(X_train[i][0]),
                            'Output-size':salidas,
                            'Hidden-layer-size':n,
                            'Learning-rate':lr,
                            'Batch-size':b,
                            'Epochs':epochs,

```

```

        }
        print('Entrenando red neuronal con las siguientes
características:')
        print(diccionario)
        clasificador = MultiLayerPerceptron(
            num_entradas=len(X_train[i][0]),
            num_neuronas_ocultas=n,
            num_salidas=salidas,
            inicializazion_function=initialization_functions[i_f_i],
            normalization_function=normalization_functions[i_f_n],
            loss_function=loss_function_MSE,
            activation_function=sigmoid,
            activation_derivative=sigmoid_derivative,
            epochs=epochs,
            batch_size=b,
            learning_rate=lr
        )
        clasificador.train(X_train[i], Y_train[i])
        lista_clasificadores.append((diccionario, clasificador))

```

Resultados y conclusiones

Para el análisis de los resultados se graficaron los errores de los modelos en cada época y se redujeron las épocas graficadas para mejorar la visualización.

```

def graficar_conjunto(listas_modelos, nombres_labels, titulo, caracteristicas,
filename, épocas, opacidad=0.2):
    """
    Grafica un conjunto de redes neuronales en la misma grafica
    """
    colores = ['black', 'darkred', 'darkviolet', 'forestgreen', 'darkorange',
'royalblue']
    marcadores = ['solid', (0, (3, 1, 1, 1)), 'dotted', (0, (5, 1)), 'dashed', (0, (5,
10)), (5, (10, 3)), (0, (3, 5, 1, 5))]
    i = 0
    offset = len(colores) - len(listas_modelos)
    patches = []
    plt.figure(figsize=(16, 9))
    plt.title(titulo)
    plt.figtext(0.13, 0.03, caracteristicas)
    for l in listas_modelos:
        color = colores[i + offset]
        for t in l:
            plt.plot(np.arange(epocas), t[1].epochs_error[:epocas], color=color,
linestyle=marcadores[i], linewidth=1, alpha=opacidad)
            patches.append(mpatches.Patch(color=color, label=nombres_labels[i]))
        i += 1
    plt.legend(handles=patches)
    plt.xlabel('Época')
    plt.ylabel('MSE')

```

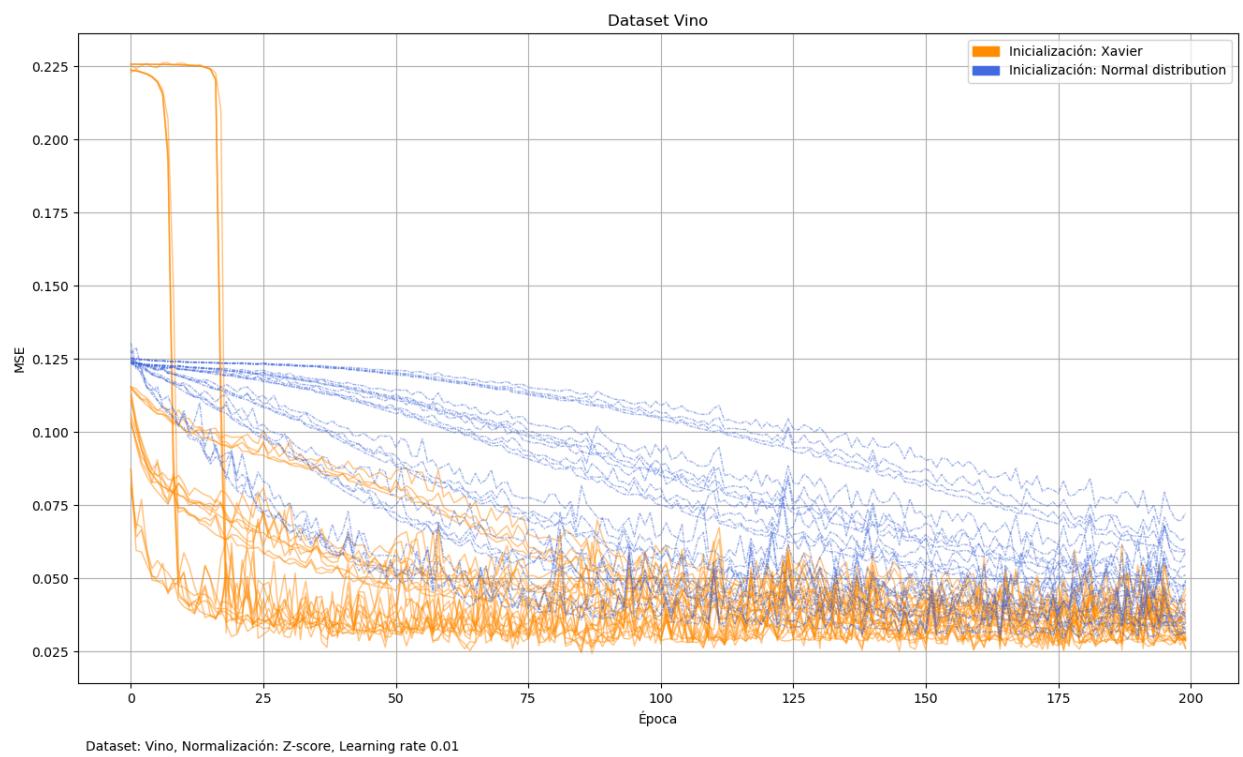
```
plt.grid(True)
plt.savefig(filename)
plt.close()
```

Para comparar uno de los parámetros del experimento se dejaron fijos otros, especialmente los que ofrecen una mejor visualización; para filtrar todos los modelos con un conjunto de características dadas, se hizo uso de las listas por comprensión de Python y de los diccionarios que almacenan esos datos. Las variables específicas de cada gráfica se encuentran en la parte inferior izquierda.

```
# Filtro por cantidad de neuronas de la capa ocultas
Iris_neuronas = []
labels_Iris_neuronas = []
for n in neuronas_ocultas:
    Iris_neuronas.append([t for t in lista_clasificadores if t[0]['Dataset'] == 'Iris'
                           and t[0]['Initialization'] == 'Normal distribution'
                           and t[0]['Normalization'] == 'Z-score'
                           and t[0]['Hidden-layer-size'] == n])
    labels_Iris_neuronas.append(f'{n} neuronas ocultas')
graficar_conjunto(listas_modelos=Iris_neuronas,
                  nombres_labels=labels_Iris_neuronas,
                  titulo='Dataset Iris',
                  características='Dataset: Iris, Inicialización: Distribución normal,
Normalización: Z-score',
                  filename='Iris_neuronas.png',
                  épocas=300,
                  opacidad=0.5)
```

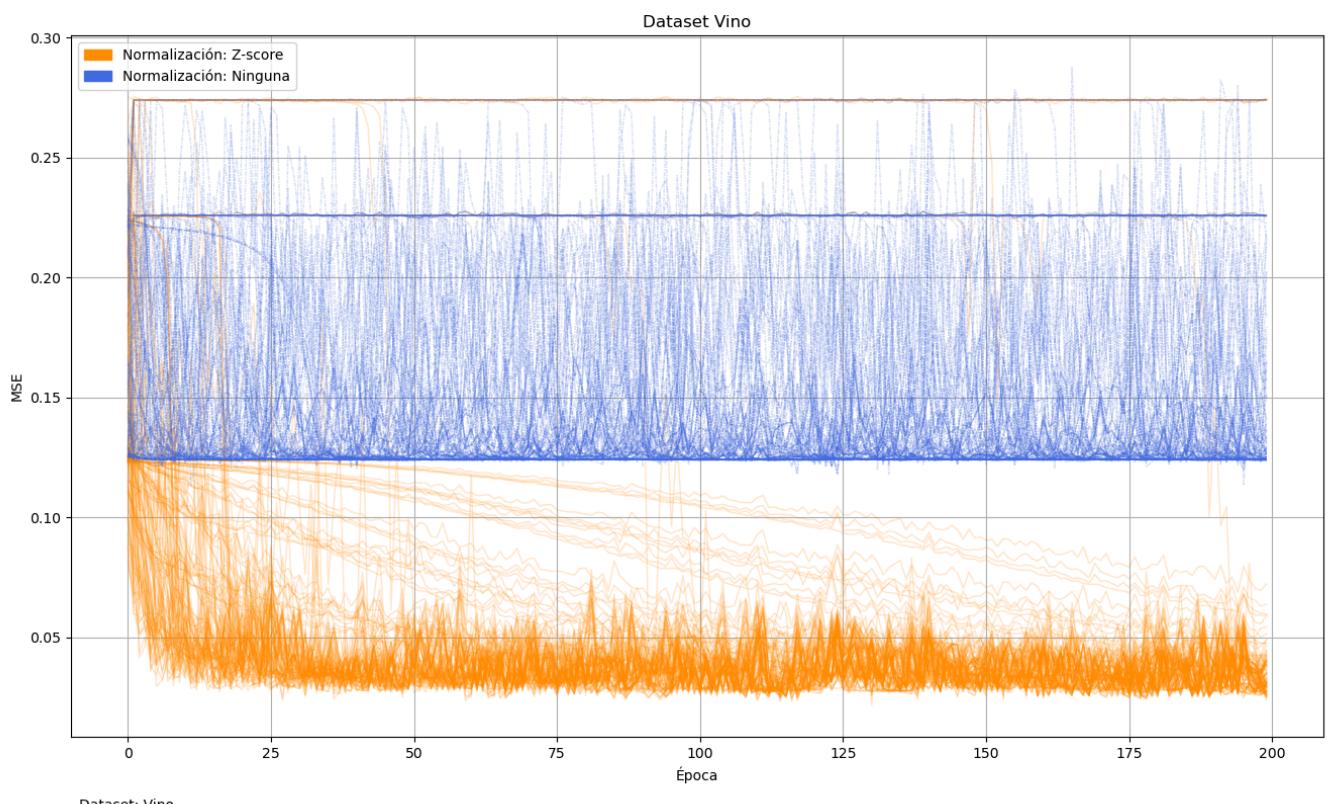
Inicialización

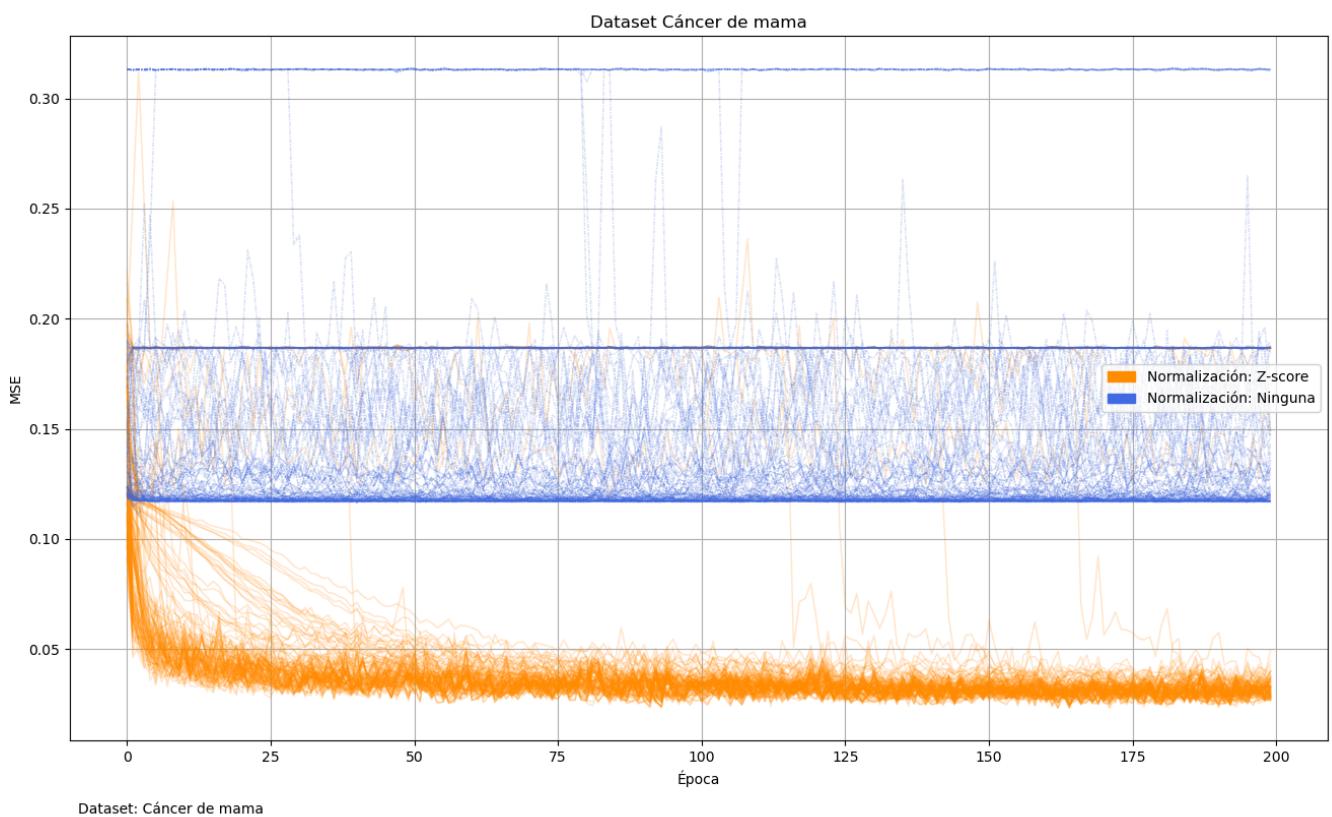
La inicialización de Xavier fue claramente superior a la de una distribución normal, el error decrece notablemente más sobre todo en las primeras épocas, si bien la otra inicialización converge al final en valores similares, la ventaja de la inicialización de Xavier está en que de haber implementado un criterio de paro, se hubiera reducido drásticamente el poder de cómputo necesario para entrenar las redes neuronales. En la gráfica siguiente se muestran los resultados obtenidos para el dataset de vinos, pero en otros datasets se obtuvieron resultados similares.



Normalización

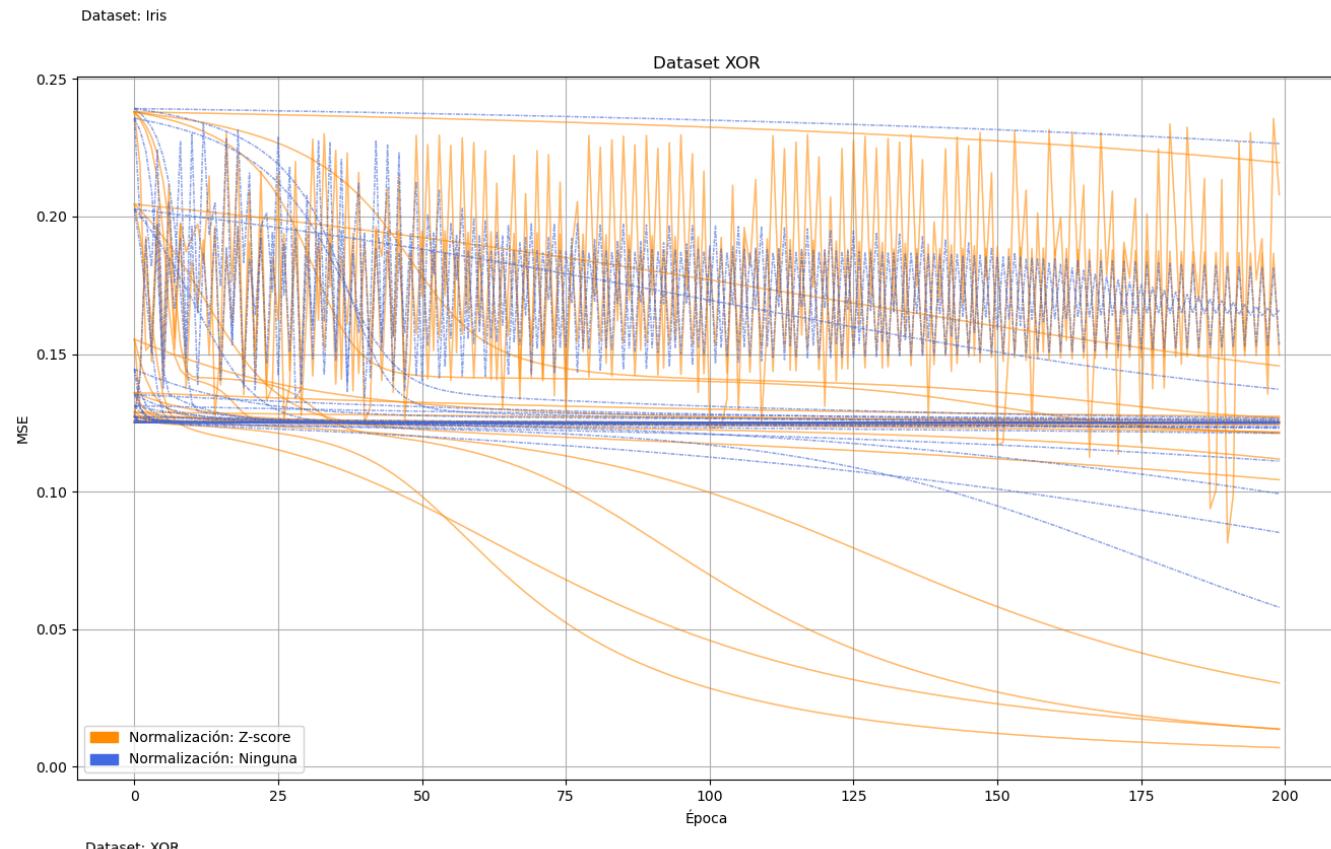
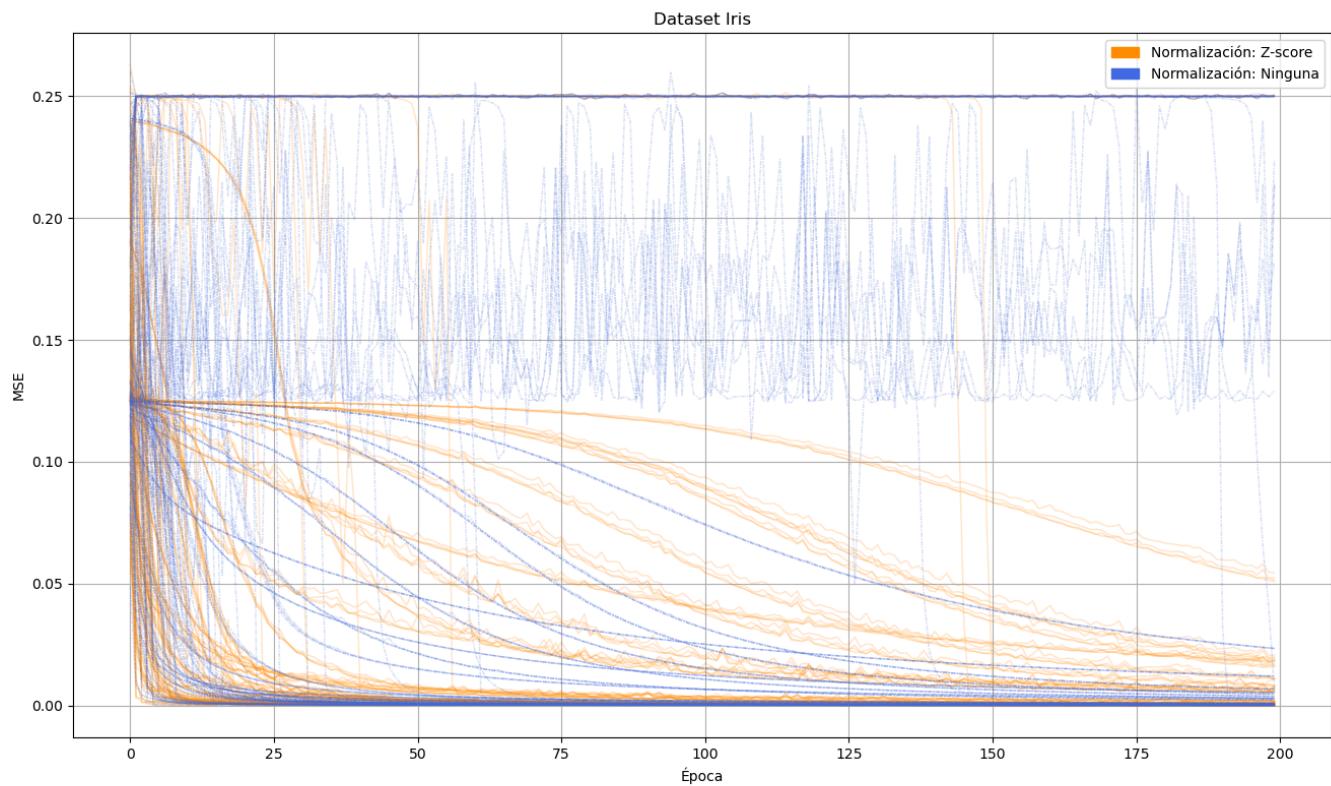
La normalización fue sin duda el parámetro que obtuvo resultados más distintos para cada uno de los datasets, en las siguientes dos gráficas se observa como no normalizar los datos en problemas complejos, resulta en que el modelo no aprende sin importar el número de épocas.





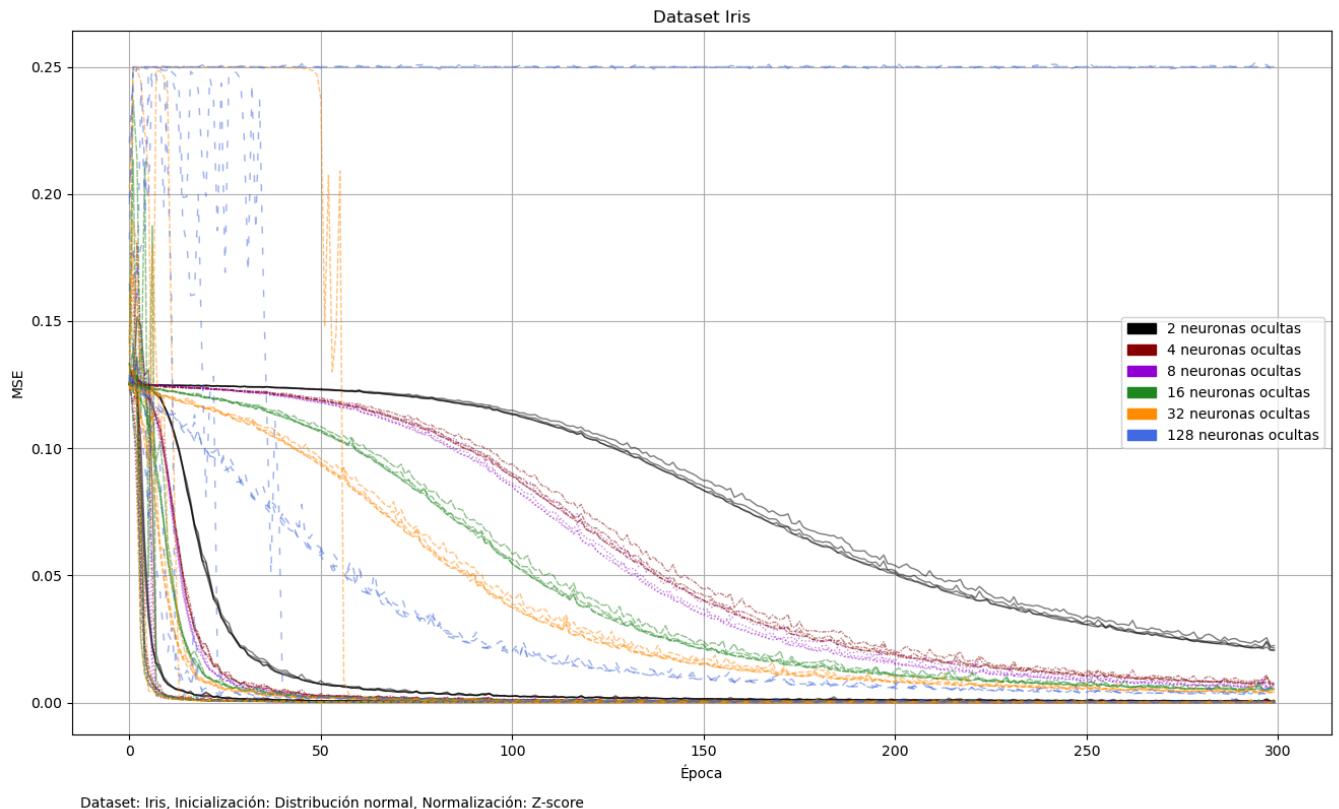
Dataset: Cáncer de mama

Sin embargo para datasets con menos características los resultados son mucho menos claros. Es posible observar una tendencia a que la normalización de los datos reduce las posibilidades de que la red neuronal no converja en un valor, pero en estos casos, los modelos sin datos normalizados también consiguieron converger, aunque en menor medida.



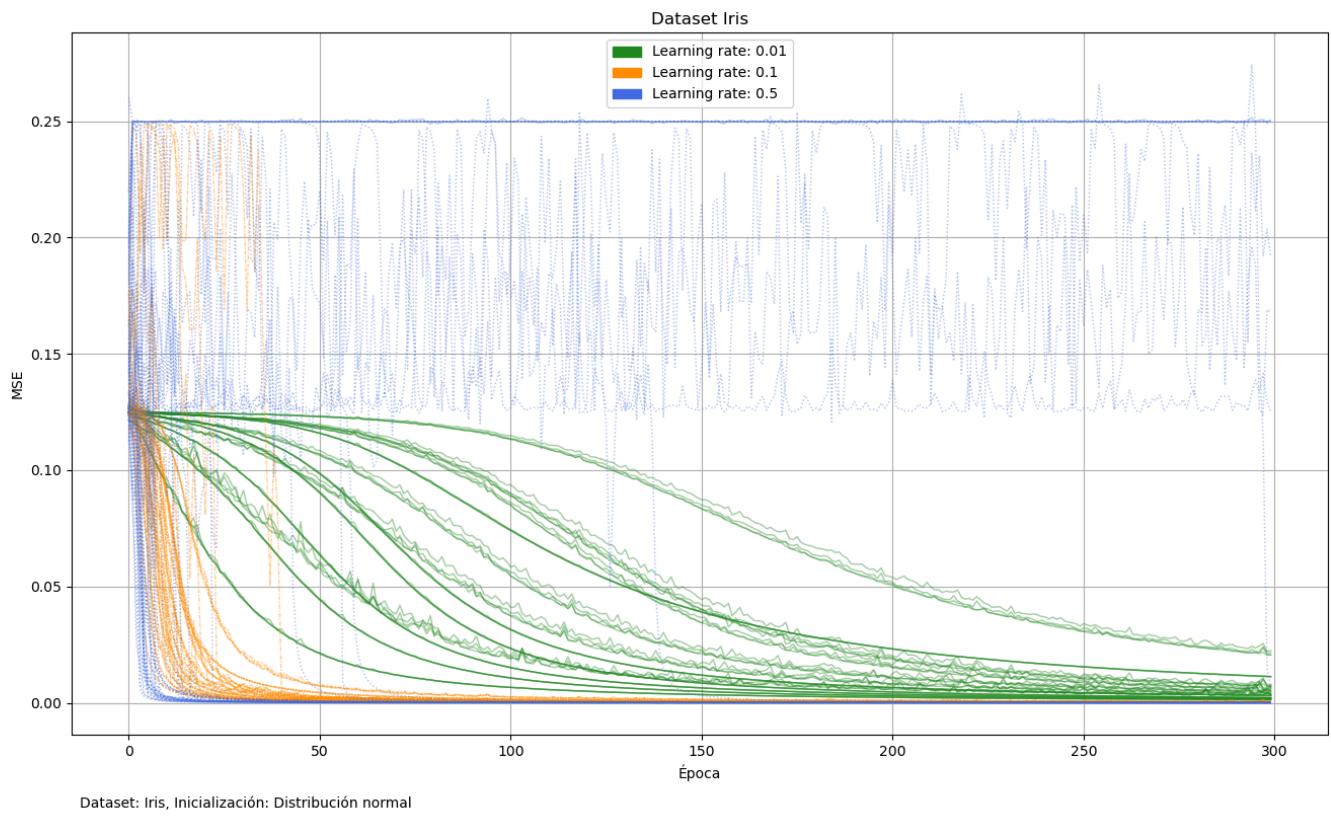
Número de neuronas

En el número de neuronas de la capa oculta se observa una clara tendencia a que los modelos con más neuronas convergen más rápido, esto no deja en claro si son objetivamente mejores pues requieren de menos iteraciones para aprender pero a cambio cada iteración se vuelve más costosa de calcular. En la gráfica se observa el dataset Iris y aunque para otros datasets la tendencia no fue tan clara, por norma general los modelos con más neuronas tienden a producir mejores resultados.



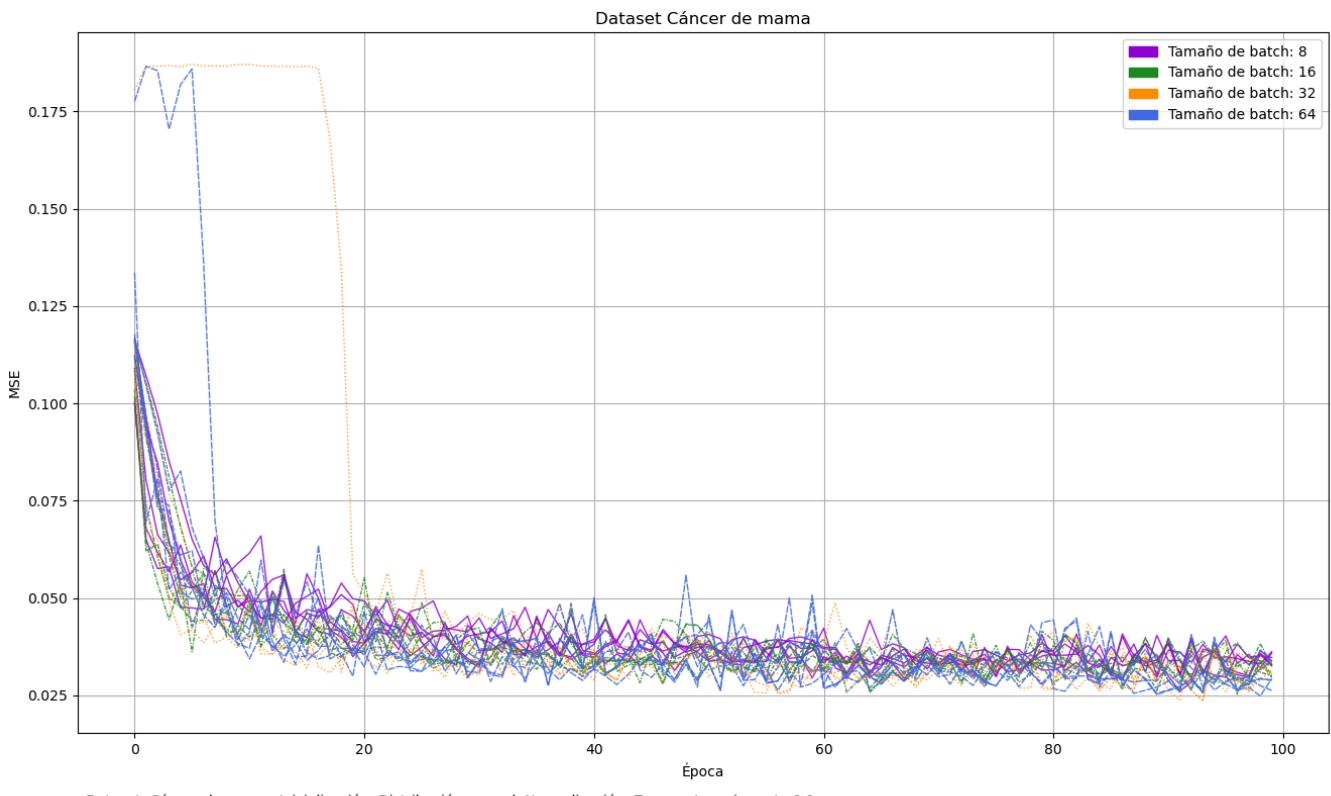
Tasa de aprendizaje

La tasa de aprendizaje es un caso interesante donde no hay un claro ganador, aunque todas las tasas consiguieron modelos que convergen en un valor, no todas lo hacen siempre. La tasa de 0.5 fue la que obtuvo más modelos que no convergen, mientras que la tasa intermedia de 0.1 obtuvo solo algunos modelos que no convergen y continuando con el patrón, la tasa más baja de 0.01 converge en todos los casos experimentados. Esto nos indica que una tasa muy alta puede ser detrimental para la red neuronal, al intentar aprender más rápido puede obtener resultados basura, mientras que una tasa muy baja asegura que esto no ocurra, pero al coste de una convergencia mucho menor, pues en cada época el modelo aprende mucho menos. Es necesaria mayor experimentación con nuevos valores para la tasa de aprendizaje para reconocer un punto de balanceo entre aprender más rápidamente y converger en la mayoría de los casos.



Tamaño de minibatch

El tamaño del minibatch fue sin lugar a dudas la variable menos relevante de todas, consiguiendo resultados sin diferencia significativa en todos los casos.



Mejores modelos por dataset

Finalmente se probaron todos los modelos para medir el accuracy y se obtuvo una lista con los que tuvieron las mejores puntuaciones, pues hubo varios modelos con características distintas que obtuvieron el mismo resultado.

```
mejores_clasificadores = [[], [], [], []]
for d, c in lista_clasificadores:
    i = nombres_datasets.index(d['Dataset'])
    c.predict(X_test[i], Y_test[i])
    d.update({'Accuracy':c.accuracy, 'Last-error':c.epochs_error[-1]})

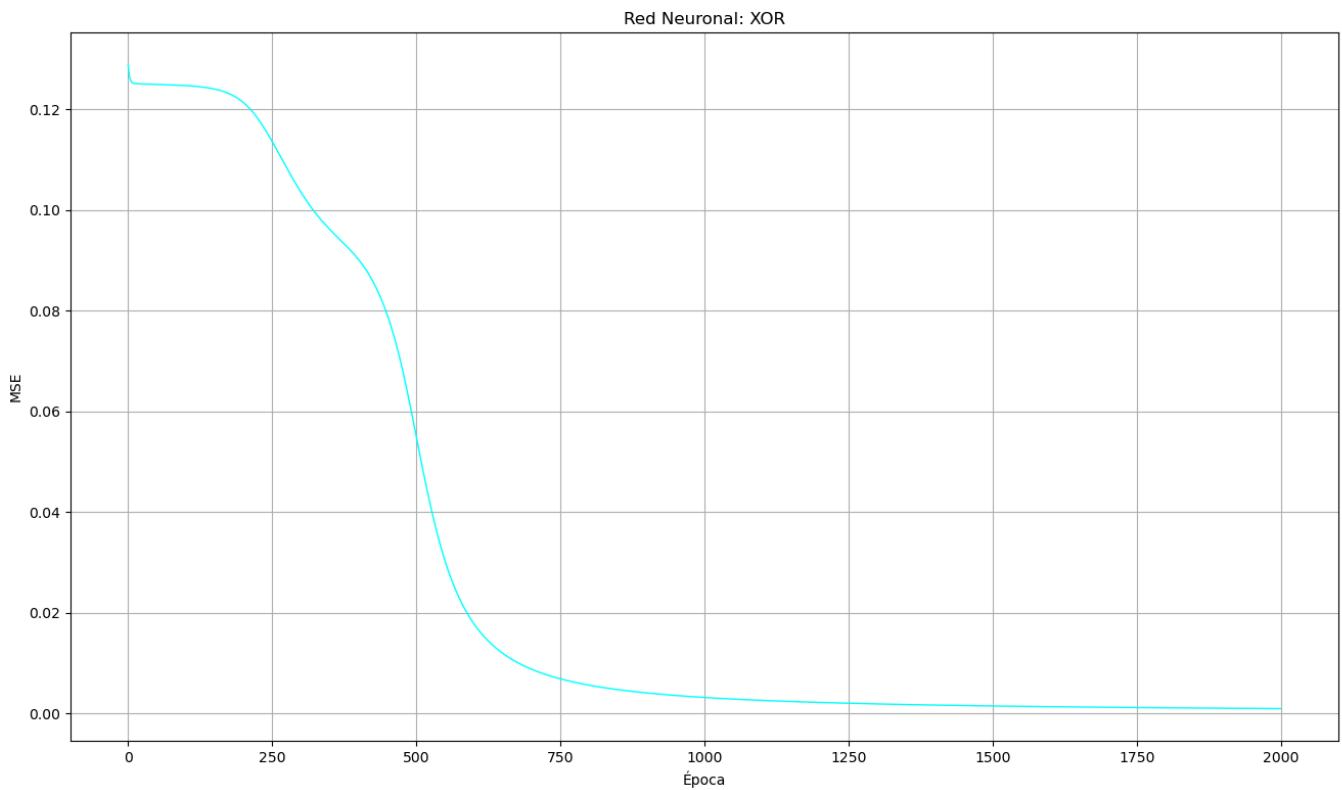
    if len(mejores_clasificadores[i]) != 0:
        if mejores_clasificadores[i][0][0]['Accuracy'] == d['Accuracy']:
            mejores_clasificadores[i].append((d, c))
        elif mejores_clasificadores[i][0][0]['Accuracy'] < d['Accuracy']:
            mejores_clasificadores[i] = [(d, c)]
    else:
        mejores_clasificadores[i] = [(d, c)]


for lista in mejores_clasificadores:
    print(f'\n#####Dataset {lista[0][0]['Dataset']} #####\n')
    print(f'Mejor accuracy: {lista[0][0]['Accuracy']}')
    if len(lista) > 1:
        print(f'Hay {len(lista)} clasificadores con el mismo accuracy')
        for d, c in lista:
            print(d)
    else:
        print(lista[0][0])
```

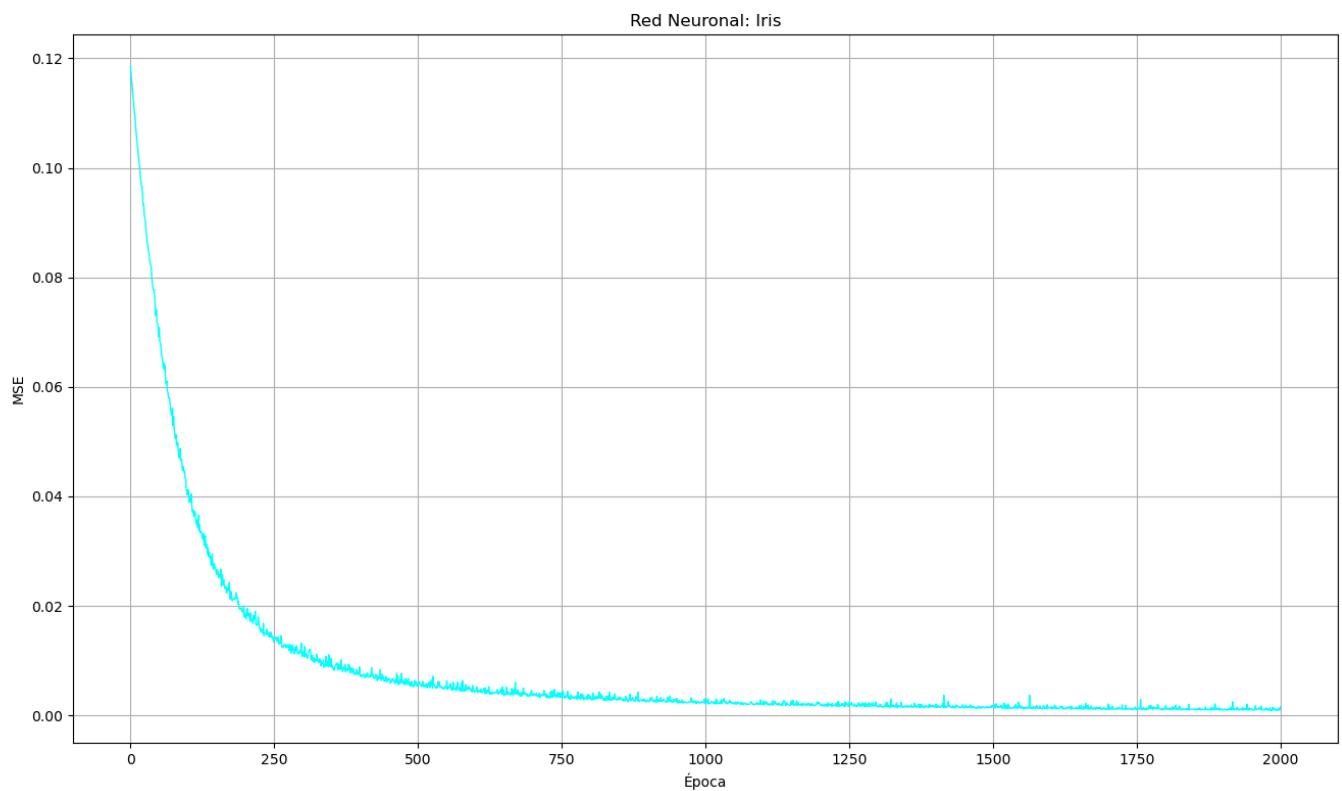
A continuación el resumen de los resultados.

Dataset	Mejor accuracy	Cantidad de modelos con el mejor accuracy
Compuerta lógica XOR	1	112
Plantas Iris	1	264
Cáncer de mama	0.95	8
Vinos	1	3

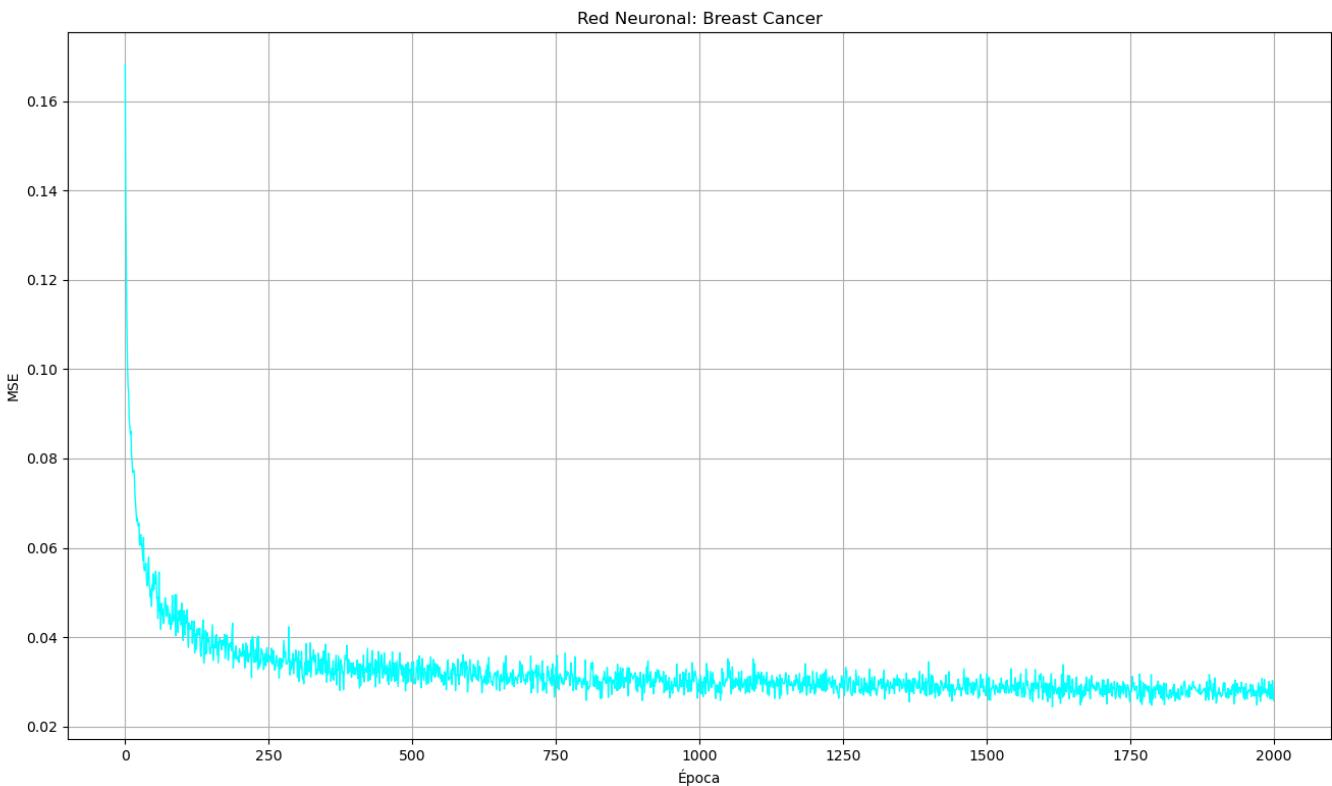
La lista completa con los modelos específicos y sus características se encuentra adjunto a este documento, pero aquí se agregan las gráficas de un ejemplo de algunos modelos con el mejor accuracy para su dataset.



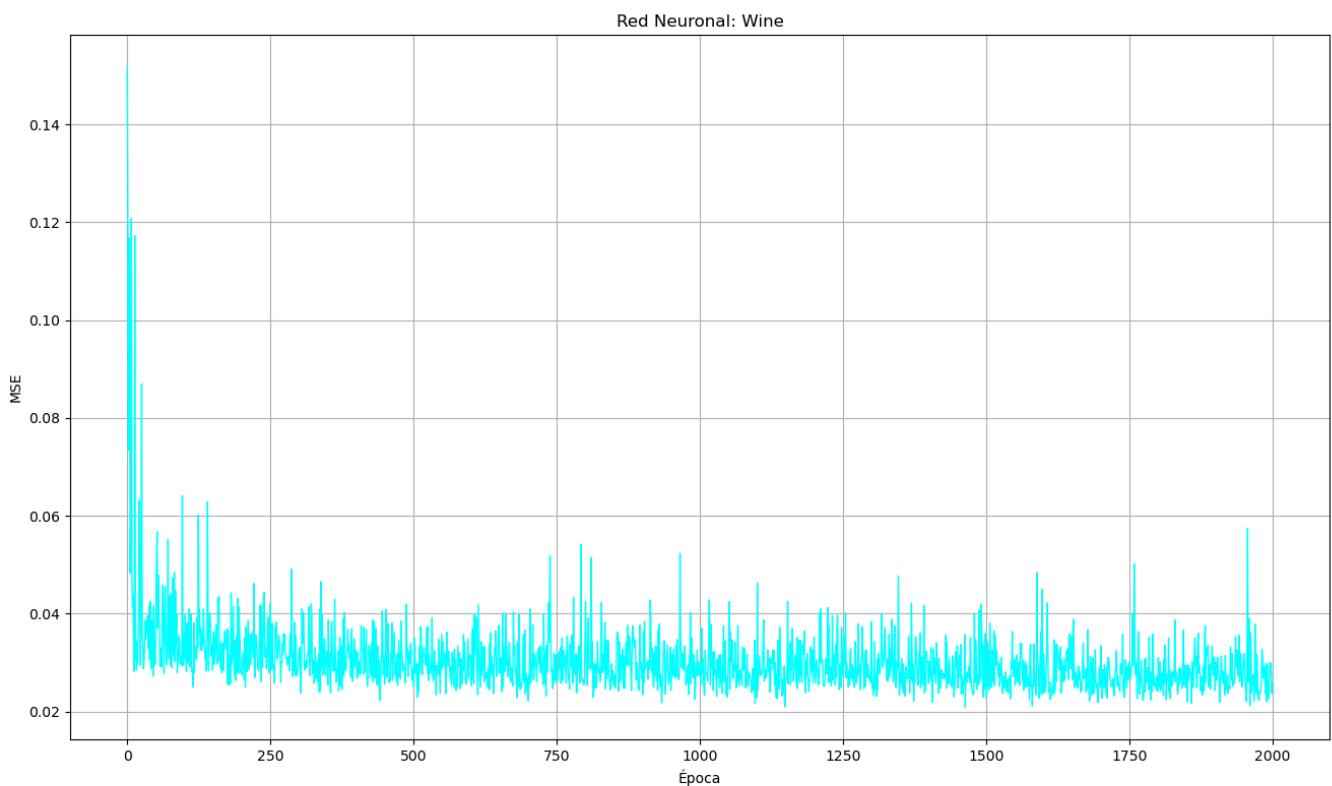
Dataset: XOR, Función de Inicialización: Xavier, Función de Normalización: Z-score, Neuronas ocultas: 2, Learning Rate: 0.5, Batch Size: 8



Dataset: Iris, Función de Inicialización: Xavier, Función de Normalización: Z-score, Neuronas ocultas: 2, Learning Rate: 0.01, Batch Size: 8



Dataset: Breast Cancer, Función de Inicialización: Xavier, Función de Normalización: Z-score, Neuronas ocultas: 8, Learning Rate: 0.01, Batch Size: 8



Dataset: Wine, Función de Inicialización: Xavier, Función de Normalización: Z-score, Neuronas ocultas: 2, Learning Rate: 0.5, Batch Size: 32

Conclusiones

Los perceptrones multicapa son una herramienta de Inteligencia Artificial muy poderosa que es capaz de clasificar objetos según sus características y resolver muchos otros problemas que no fueron explorados en esta práctica. Siempre que se tenga un conjunto de datos lo suficientemente amplio y de calidad, los modelos pueden ser entrenados para predecir de manera sorprendentemente acertada con arquitecturas de muy pocas capas y neuronas.

La infinidad de estrategias que es posible usar en una red neuronal ofrecen una amplia variedad de combinaciones que en ofrecen ventajas y desventajas, muchas veces mejor precisión a coste de mayor costo computacional, pero son raras las estrategias que es siempre es mejor una que otra, sobre todo para conjuntos de datos distintos, siempre se dependerá de la naturaleza del problema. De cualquier modo si es posible observar patrones de comportamiento en las variables de los experimentos más detalladamente descritos en la sección de resultados.

Las redes neuronales son un problema complejo que no es intuitivo y por ello es importante la experimentación, la cantidad de modelos que consiguieron el mejor accuracy resulta antiintuitivo, pues por ejemplo, el problema más simple (XOR) no tiene la mayor cantidad de modelos con accuracy 1 y el problema más complejo (Cáncer de mama) no tiene tampoco la menor cantidad de modelos con el mejor accuracy. Si bien unas estrategias tienden a funcionar mejor que otras, con suficiente suerte y poder de cómputo es posible obtener modelos con buen accuracy sin usar necesariamente las estrategias más complejas.