Taller de Modelado II

LUIS ADRIAN MARTINEZ PEREZ

December 2024

1. Introduction

El objetivo de está practica consiste en implementar una red neuronal con retropagación para clasificar las flores Iris en *Python* y sin el uso de bibliotecas especializadas como *Keras*. Para esta práctica usaremos la biblioteca *numpy* que es un paquete que dispone de múltiples herramientas para manejar matrices de una forma muy eficiente[1].

Esta práctica tiene como base el artículo *Predict on the Iris dataset using a neural network made in numpy* cuyo objetivo es el uso de *numpy* para desarrollar la red neuronal. En esta adaptación optamos por definir funciones para escalar los datos y dividirlos y reservar el uso de *Keras* para obtener el conjunto de datos. Con ello esperamos conseguir el objetivo de usar únicamente *numpy*.

En esta práctica observaremos el rendimiento y la precisición de los modelos al variar: el número de neuronas en la capa oculta y el tamaño del conjunto de entrenamiento. En nuestro caso, no solo dividimos los datos en conjuntos de entrenamiento y prueba sino que separamos y reservamos el 10 % de los datos para evaluar nuestros modelos con nuevos datos.

2. Implementación

En la **Práctica 1** puede consultarse con detalle el proceso de retropropagación que estamos implementando aquí. Vamos a desarrollar nuestros modelos en la plataforma de *Kaggle* y el programa puede revisarse en el siguiente enlace:

https://www.kaggle.com/code/iris-dataset-nn-numpy

Paso 1: Importar las bibliotecas necesarias.

```
[1] import numpy as np
[2] import matplotlib.pyplot as plt
[3] from sklearn.datasets import load_iris
```

Paso 2: Definir las funciones para separar los datos (mezclandolos), de escalamiento, de activación y su derivada.

```
[4] # Función para dividir los datos
[5] def test_split(X, y, test_size, random_state=None):
[6]     np.random.seed(random_state)
[7]     indices = np.arange(X.shape[0])
[8]     np.random.shuffle(indices)
[9]     split_idx = int(X.shape[0] * (1 - test_size))
[10]    train_indices, test_indices = indices[:split_idx], indices[split_idx:]
[11]    return X[train_indices], X[test_indices], y[train_indices], y[test_indices]
```

[1] https://www2.eii.uva.es/fund$_inf/python/notebooks/Bibliotecas/03_Numpy/Numpy.html$

```
[12]
[13] # Función para estandarizar datos
[14] def standard_scaler(X):
[15]     mean = np.mean(X, axis=0)
[16]     std = np.std(X, axis=0)
[17]     return (X - mean) / std
[18]
[19] # Función sigmoide y derivada
[20] def sigmoid(x):
[21]     return 1 / (1 + np.exp(-x))
[22]
[23] def sigmoid_derivative(x):
[24]     return x * (1 - x)
```

Paso 3: Cargar el conjunto de datos iris, codificamos las etiquetas (target) y dividimos los datos reservando el 10 % para validación.

```
[24] # Cargar el conjunto de datos Iris
[25] iris = load_iris()
[26] X = iris.data
[27] y = iris.target
[28]
[29] # Codificar etiquetas en formato one-hot
[30] one_hot_encoded = np.zeros((y.size, y.max() + 1))
[31] one_hot_encoded[np.arange(y.size), y] = 1
[32] y_encoded = one_hot_encoded
[33]
[34] # Dividir datos en conjunto restante y validación
[35] X_restante, X_validacion, y_restante, y_validacion = test_split(
[36]     X, y_encoded, test_size=0.1, random_state=42
[37] )
```

Paso 4: Configurar los hiperparámetros de la red neuronal, los tamaños de entrenamiento a usar en cada ciclo y las listas para guardar el accuracy de entrenamiento y validación.

```
[37] # Configuración de la red neuronal
[38] input_size = X_restante.shape[1]
[39] hidden_size = 5
[40] output_size = y_restante.shape[1]
[41] learning_rate = 0.01
[42] epochs = 1000
[43]
[44] # Lista para guardar los resultados de accuracy
[45] train_sizes = np.arange(0.1, 1, 0.1)  # Tamaños de entrenamiento de 0.1 a 0.9
[46] accuracies_train = []
[47] accuracies_val = []
```

Paso 5: Para cada tamaño de entrenamiento dividir los datos, estandarizar los datos e inicializar pesos y bias.

```
[47] for train_size in train_sizes:
[48]     # Dividir conjunto restante en entrenamiento y prueba
[49]     X_train, X_test, y_train, y_test = test_split(
[50]         X_restante, y_restante, test_size=1 - train_size, random_state=42
[51]     )
[52]
[53]     # Estandarizar los datos
```

```
[54]     X_train = standard_scaler(X_train)
[55]     X_test = standard_scaler(X_test)
[56]     X_validacion = standard_scaler(X_validacion)
[57]
[58]     # Inicializar pesos y bias
[59]     np.random.seed(42)
[60]     weights_input_hidden = np.random.uniform(size=(input_size, hidden_size))
[61]     weights_hidden_output = np.random.uniform(size=(hidden_size, output_size))
[62]     bias_hidden = np.zeros((1, hidden_size))
[63]     bias_output = np.zeros((1, output_size))
```

Paso 6: Entrenar cada modelo con propagación hacia adelante y hacia atrás y actualizar los pesos y bias.

```
[64]     # Entrenamiento de la red neuronal
[65]     for epoch in range(epochs):
[66]         # Propagación hacia adelante
[67]         hidden_layer_input = np.dot(X_train, weights_input_hidden) + bias_hidden
[68]         hidden_layer_output = sigmoid(hidden_layer_input)
[69]         output_layer_input = np.dot(hidden_layer_output, weights_hidden_output) + bias_output
[70]         predicted_output = sigmoid(output_layer_input)
[71]
[72]         # Propagación hacia atrás
[73]         error = y_train - predicted_output
[74]         d_predicted = error * sigmoid_derivative(predicted_output)
[75]         error_hidden_layer = d_predicted.dot(weights_hidden_output.T)
[76]         d_hidden_layer = error_hidden_layer * sigmoid_derivative(hidden_layer_output)
[77]
[78]         # Actualizar pesos y bias
[79]         weights_hidden_output += hidden_layer_output.T.dot(d_predicted) * learning_rate
[80]         weights_input_hidden += X_train.T.dot(d_hidden_layer) * learning_rate
[81]         bias_output += np.sum(d_predicted, axis=0, keepdims=True) * learning_rate
[82]         bias_hidden += np.sum(d_hidden_layer, axis=0, keepdims=True) * learning_rate
```

Paso 7: Evaluar accuracy en los conjuntos de prueba y validación. Decodificar las predicciones en ambos conjuntos y calcular la accuracy. Graficar los resultados

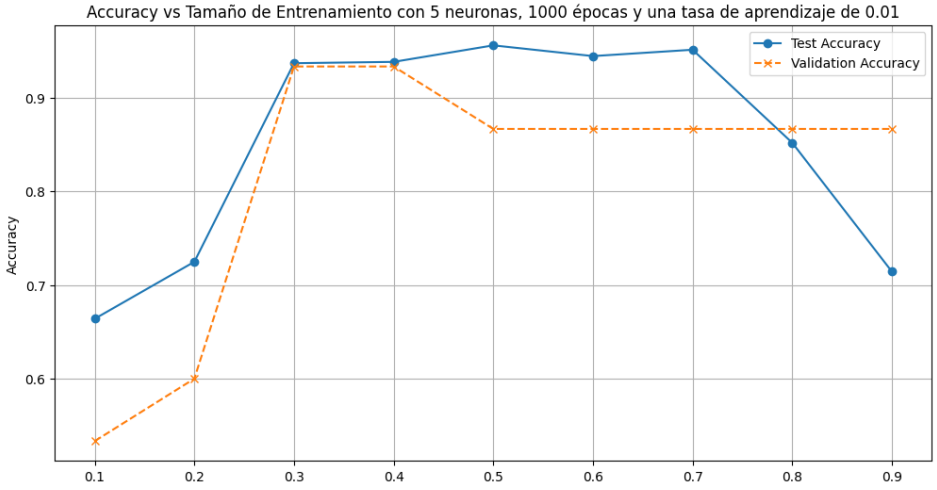
```
[82]     # Evaluar precisión en el conjunto de prueba
[83]     hidden_layer_input_test = np.dot(X_test, weights_input_hidden) + bias_hidden
[84]     hidden_layer_output_test = sigmoid(hidden_layer_input_test)
[85]     output_layer_input_test = np.dot(hidden_layer_output_test, weights_hidden_output) + bias_output
[86]     predicted_output_test = sigmoid(output_layer_input_test)
[87]
[88]     # Decodificar predicciones y calcular precisión en el conjunto de prueba
[89]     y_pred_test = np.argmax(predicted_output_test, axis=1)
[90]     y_true_test = np.argmax(y_test, axis=1)
[91]      accuracy_test = np.mean(y_pred_test == y_true_test)
[92]      accuracies_train.append(accuracy_test)
[93]
[94]      # Evaluar precisión en el conjunto de validación
[95]      hidden_layer_input_val = np.dot(X_validacion, weights_input_hidden) + bias_hidden
[96]      hidden_layer_output_val = sigmoid(hidden_layer_input_val)
[97]      output_layer_input_val = np.dot(hidden_layer_output_val, weights_hidden_output) + bias_output
[98]      predicted_output_val = sigmoid(output_layer_input_val)
[99]
[100]    # Decodificar predicciones en el conjunto de validación y calcular precisión
[101]    y_pred_val = np.argmax(predicted_output_val, axis=1)
[102]    y_true_val = np.argmax(y_validacion, axis=1)
```

```
[103]    accuracy_val = np.mean(y_pred_val == y_true_val)
[104]    accuracies_val.append(accuracy_val)
[105]
[106]    print(f"Train size: {train_size:.1f}, Accuracy (Test): {accuracy_test:.4f},
               Accuracy (Validation): {accuracy_val:.4f}")
[107]
[108]# Graficar los resultados
[109]plt.figure(figsize=(12, 6))
[110]plt.plot(train_sizes, accuracies_train, marker='o', linestyle='-', label='Test Accuracy')
[111]plt.plot(train_sizes, accuracies_val, marker='x', linestyle='--', label='Validation Accuracy')
[112]plt.title(f"Accuracy vs Tamaño de Entrenamiento con {hidden_size} neuronas, {epochs}
               épocas y una tasa de aprendizaje de {learning_rate}")
[113]plt.ylabel("Accuracy")
[114]plt.grid()
[115]plt.legend()
[116]plt.show()
```

3. Resultados.

3.1. Variando el tamaño del conjunto de entrenamiento y prueba.

A continuación podemos observar el comportamiento del perceptron multicapa MLP(4,5,1) con una tasa de aprendizaje de 0.01 y 1000 épocas de entrenamiento:

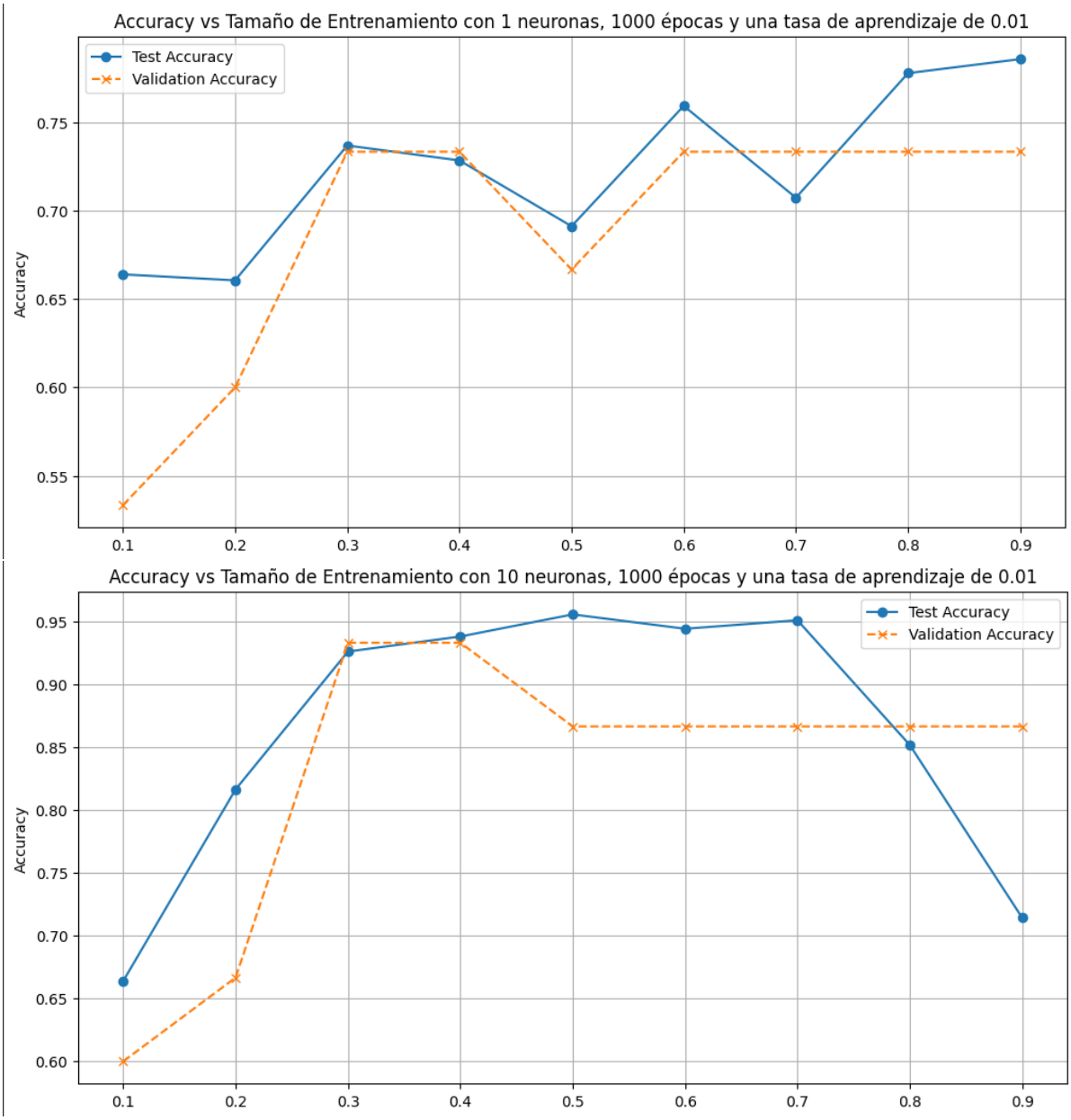


Cuadro 1: Precisión del modelo según el tamaño del conjunto de entrenamiento

Tamaño del conjunto de entrenamiento	Precisión	
	Prueba	Validación
0.1	0.6639	0.5333
0.2	0.7248	0.6000
0.3	0.9368	0.9333
0.4	0.9383	0.9333
0.5	0.9559	0.8667
0.6	0.9444	0.8667
0.7	0.9512	0.8667
0.8	0.8519	0.8667
0.9	0.7143	0.8667

3.2. Variando el número de neuronas en la capa oculta.

A continuación observamos el comportamiento del error cuando varia el número de neuronas en la capa oculta. En particular, vemos el comportamiento de MLP(4,2,1) y MLP(4,10,1). En las gráficas se observa el error en el conjunto de prueba y validación. Como en el caso anterior la tasa de aprendizaje es de 0.01 con 1000 épocas de entrenamiento.

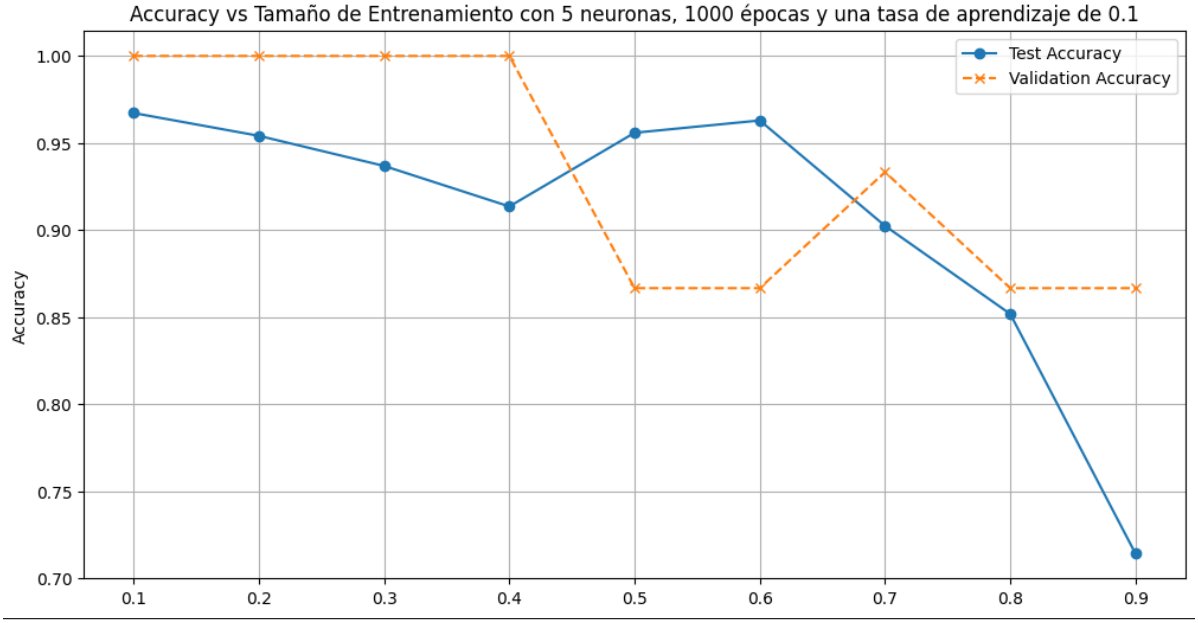


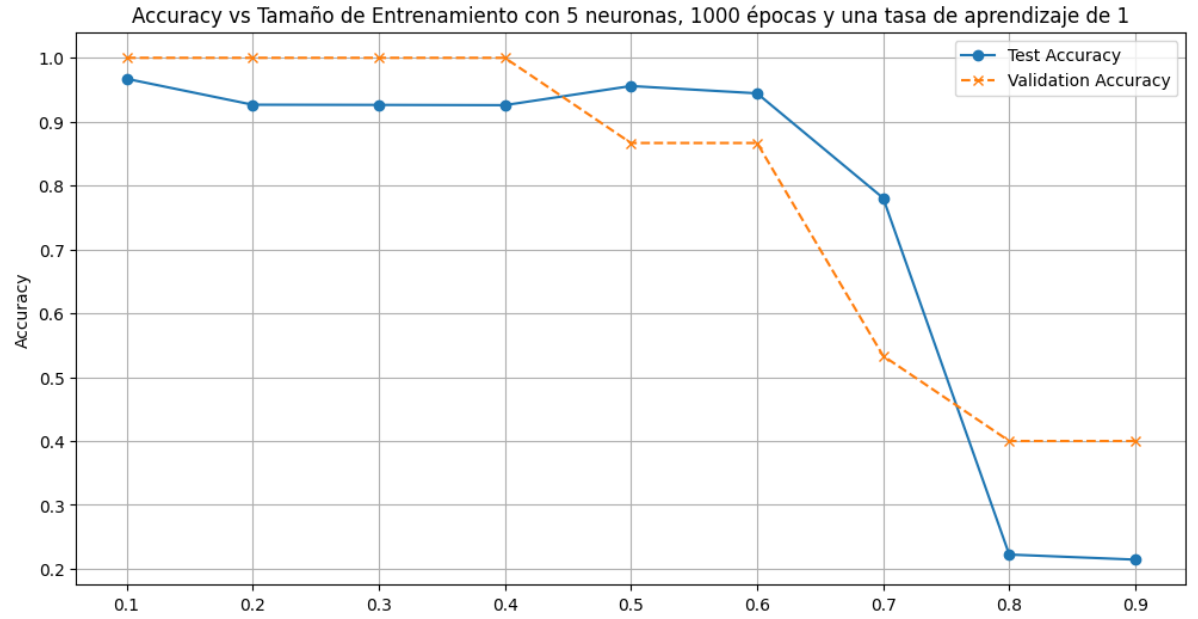
Como puede observarse los modelos con 5 y 10 neuronas con un 70 % de datos de entrenamiento y 30 % de prueba tiene el mejor desempeño.

3.3. Variando la tasa de aprendizaje.

Finalmente podemos observar el comportamiento del error en el conjunto de prueba y validación al variar la tasa de aprendizaje. Se observa al comparar las graficas que con una tasa de aprendizaje 0.1 tiene mejor

rendimiento.





4. Conclusión.

Como podemos observar la variación del tamaño del conjunto de entrenamiento y prueba, el número de neuronas en la capa oculta y la tasa de aprendizaje modifican el rendimiento. A diferencia del modelo que sirvió para desarrollar esta práctica la presición no fue 1. Las razones para esta última observación se deben a que los modelos pueden pueden sobreajustarse por lo que al dividir el conjunto de datos inicialmente y reservar un parte para validación tenemos un comportamiento más realista.