Redes neuronales

Enrique Carro Garrido

I. Introducción (motivación/objetivo de la práctica)

El problema de clasificación es fundamental en multitud de campos. En ciencias de la información, por ejemplo, se presenta en la categorización de textos o en la clasificación de imágenes. Incluso en biología, donde la automatización de la clasificación es una herramienta muy útil en el estudio de las proteínas.

Las máquinas de aprendizaje supervisado son capaces de clasificar conjuntos de elementos de acuerdo a un entrenamiento con un grado de precisión bastante alto, revolucionando los anteriores campos mencionados donde la necesidad de clasificar está muy presente.

En particular, nos centraremos en un tipo de algoritmo de aprendizaje supervisado, las *redes neuronales*, donde se conocen las entradas y las salidas deseadas. La unidad más pequeña de esta estructura es el perceptrón simple, objetivo principal de esta práctica.

II. Material usado (método y datos)

Sean tres elementos con pares de estado $E_1 = (1, 1),\ E_2 = (1, 0),\ E_3 = (0, 1)$, queremos clasificarlos mediante una función $f(x, y) = 3x + 2y > 2$. Para ello, vamos a utilizar un perceptrón simple diseñado por un lado manualmente y por otro mediante la librería `keras` de `tensorflow`.

El método utilizado en el diseño del perceptrón es la **Regla Delta generalizada** que consiste en dos pasos:

- *Propagación.* Dado $E = (e_1, e_2)$ un patrón de entrada y $\omega = (\omega_1, \omega_2)$ unos pesos sipnápticos inicializados **aleatoriamente** entre $[-1, 1]$, se procede de la siguiente manera:
  - Hacer propagación hacia delante de un patrón de entrenamiento para generar la salida, utilizando una función de activación que se aplica sobre el producto escalar de la entrada con los pesos. Es decir, siendo f_a la función de activación, la salida del sensor es $f_a(\omega^T \cdot E)$.
  - Hacer propagación hacia atrás a partir de la salida de la red neuronal, s_p usando la salida real inicial de entrenamiento, s_0, para obtener el error del perceptrón, $\mathcal{E}$, a partir de una función de pérdida.
- *Actualización de los pesos* con el objetivo de minimizar la función de pérdida.

$$\Delta\omega = \mathcal{E} \cdot s_p \cdot (1 - s_p) \cdot E$$
$$\omega = (1 - \theta) \cdot \Delta\omega$$

  donde $\boldsymbol{\theta = 0,5}$ es el factor de aprendizaje.

El perceptrón simple utilizará como función de activación la función **sigmoide**, σ, que se encarga de distinguir si las entradas superan o no un determinado umbral. Esta función viene dada por la siguiente expresión:

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

La función de pérdida computa la diferencia entre la etiqueta real del patrón de entrenamiento y la predicción, $\mathcal{E}$, cuyo valor se utilizará en la actualización de los pesos.

$$\mathcal{E} = s_0 - s_p$$

Para realizar los cálculos matemáticos, tales como el producto escalar de los pesos y el patrón de entrada, y para el manejo de los conjuntos de entrenamiento y prueba se ha utilizado la librería `numpy`; para dibujar las gráficas de comparación, `matplotlib`; para la obtención de datos aleatorios, la librería `random`; y para el tratamiento de los resultados la librería `pandas`. Dentro de la librería `keras` se han utilizado los siguientes módulos:

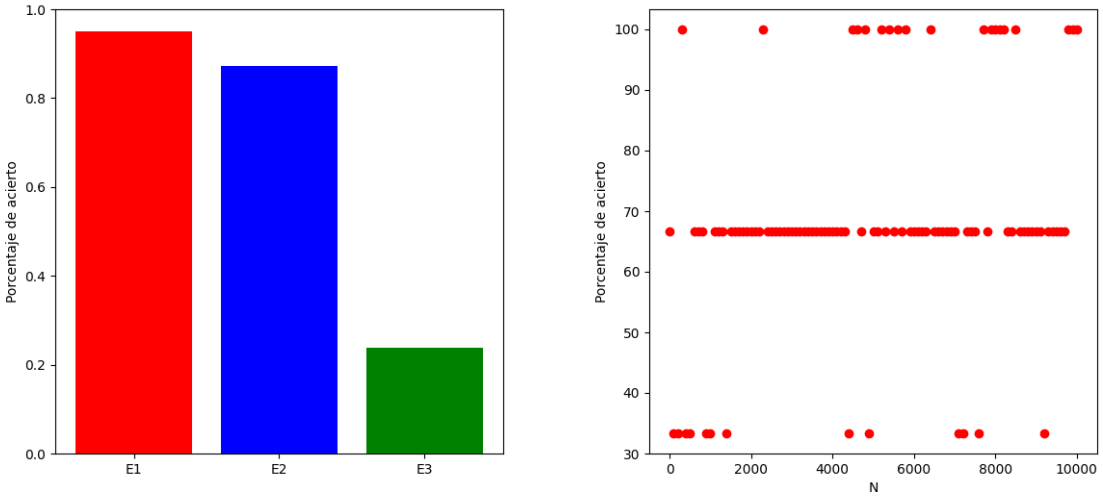
- `models.Sequential`, para definir la topología de la red neuronal. Dada la sencillez de la red construida, ya que sólo necesita un perceptrón, esta es la clase adecuada. Si se quisiera obtener una topología más compleja se debería utilizar la API `Functional`.
- `layers.Dense`, que es una clase que define una capa de neuronas en la red. En este caso, sólo se necesita una capa con una neurona.
- `optimizers.SGD`, optimizador que define la función `fit`, o el algoritmo de entrenamiento, utilizando la Regla Delta generalizada.
- `initializers`, para inicializar los pesos sipnápticos de manera aleatoria entre $[-1, 1]$.

Los datos que se han utilizado son:

- Para el conjunto de entrenamiento, se han cogido de forma aleatoria N muestras en una región $\mathcal{R} := [-500, 500] \times [-500, 500]$. La N viene parametrizada para que la comparación entre ambos modelos sea más justificada.
- Para el cálculo de la precisión en ambos modelos, se ha utilizado como conjunto de prueba los mencionados anteriormente, $E_1 = (1, 1),\ E_2 = (1, 0),\ E_3 = (0, 1)$.

III. Resultados

Se puede ver que los resultados predichos son los mismos para ambos modelos diseñados, por eso las gráficas que mostramos se refieren a los dos.



(a) (b)

Figura 1: Resultados obtenidos variando el número de ejemplos, N

En la Figura 1a, se puede ver que la mayoría de veces acierta E_1 y E_2, sin embargo, falla en clasificar el patrón E_3 más del 70 %. Esto se debe a que el elemento E_3 se encuentra en la frontera del hiperplano definido por f.

Por otro lado, en la Figura 1b, se puede ver cómo va variando el porcentaje de acierto a medida que aumenta el número de ejemplos tomados. Los resultados obtenidos muestran la incapacidad que tiene el perceptrón de clasificar con exactitud y que el número de muestras que se tome no mejora el aprendizaje.

IV. Conclusión

De esta práctica se puede aprender lo siguiente:

- Con la librería keras, se puede modelizar redes neuronales con topologías tan complejas como se quiera que llevaría mucho tiempo programarlo manualmente. Como hemos visto, los resultados son los mismos sin necesidad de implementar ningún algoritmo de entrenamiento.

- Una red neuronal necesita mucho tiempo y espacio en memoria para que tenga resultados con alta precisión. Como hemos podido ver, una sola neurona en una sola iteración acierta en muchas ocasiones pero sigue teniendo problemas en clasificar algo tan sencillo como es una relación lineal. Para aumentar la precisión se deberían añadir más capas con más neuronas, aunque esto conlleva consigo que cada vez se comporte más como una caja negra que no explique el razonamiento que sigue para clasificar.
- El factor de aprendizaje es uno de los parámetros más importantes en una red neuronal. Si se elige un valor muy grande aprende demasiado rápido y tiende a generalizar, mientras que si se elige un valor muy pequeño se produce un sobre-aprendizaje; en ambos casos la precisión del modelo se vería afectada. En este caso se produce el primer caso, el factor es muy alto y por ello tiene problemas en clasificar elementos de la frontera, como ocurre con $E = (0, 1)$.

V. Anexo con el código utilizado

```
import numpy as np
import random
import pandas as pd
import tensorflow as tf
import matplotlib.pyplot as plt
from tensorflow import keras

# Función de separación que pretendemos simular con el perceptrón simple
def f(x, y):
  return 3*x + 2*y > 2

# Diseñamos el perceptrón simple
class Perceptron_simple:
  # Factor de aprendizaje
  e = 0.5

  # Función de activación
  def sigmoid_function(self, x: float) -> float:
    """
    Logistic or sigmoid function, used as the activation function
    """
    return 1./(1 + np.exp(-x))

  def salida_sensor(self, E: np.array) -> float:
    """
    Función de predicción para un punto E, que consiste en aplicar
    la función sigmoide a c(E) = <self.weights, E>.
    """
    # Función de conexión, c(E)
    c = np.dot(self.weights, E)
    # Aplicamos la función sigmoide a c(E)
    output_sensor = self.sigmoid_function(c)
    # Devolvemos la salida del sensor
    return output_sensor


  # Función de aprendizaje
  def fit(self, X_entrenamiento: np.array, S_entrenamiento: np.array, n_it: int = 1):
    """
    Función de aprendizaje basada en la Regla Delta generalizada, donde
    Args:
        - E_entrenamiento: El conjunto de datos de entrenamiento.
        - S_entrenamiento: Salidas reales del conjunto de entrenamiento.
        - n_it: Número de iteraciones con el conjunto de entrenamiento.
    """
    # Asignamos valores aleatorios y pequeños, tanto positivos como negativos
    # a los pesos sinápticos. Normalmente estos pesos están comprendidos entre
    # [-1, 1].
    self.weights = np.random.uniform(-1, 1, X_entrenamiento.shape[1])
    self.b_weights = self.weights.copy()
```

```
    # Para cada uno de los patrones de entrada, realizamos "n_it" iteraciones.
    for it in range(n_it):
      for E, s in zip(X_entrenamiento, S_entrenamiento):
        # Feedforward: Conexión entre la entrada
        s_predecido = self.salida_sensor(E)
        # Backpropagation: Actualizamos los pesos
        error =  s - s_predecido
        delta_w = error * s_predecido * (1 - s_predecido) * E
        self.weights += (1 - self.e)*delta_w

  # Función predicción
  def prediccion(self, X_prueba: np.array) -> np.array:
    """
    Una vez entrenado el perceptrón simple, devuelve salidas posibles para
    un conjunto de prueba que se pasa por argumento
    """
    return np.array([self.salida_sensor(E) for E in X_prueba]) > 0.5

  def beginning_weights(self) -> np.array:
    return self.b_weights

# Definimos el conjunto de prueba que va a ser el mismo siempre
X_prueba = np.array([[1,1], [1,0], [0,1]])
S_prueba = np.array([f(E[0], E[1]) for E in X_prueba])

# Creamos los dataframes donde se recoge la información
columnas = ['N', 'Precision PS', 'Precision Keras', 'Acierta E1', 'Acierta E2', 'Acierta E3']
df = pd.DataFrame(columns=columnas)

# Definimos una función que realiza la comparación entre los dos modelos, fijando la región R donde
# se coge el conjunto de entrenamiento y N el número de ejemplos
def comparacion(N):
    """
    Función que compara el perceptrón simple diseñado en la clase de arriba y el diseñado por
    la librería keras.

    Args:
        R: Región donde se encuentran el conjunto de entrenamiento.
        N: Cardinal del conjunto de entrenamiento.
        df: Dataframe donde se recoge la información de las pruebas.
    """
    # Obtenemos el conjunto de entrenamiento junto a sus etiquetas
    X_entrenamiento = np.array([[random.randint(-R, R), random.randint(-R, R)] for n in range(N)])
    S_entrenamiento = np.array([f(E[0], E[1]) for E in X_entrenamiento])

    # Entrenamos y evaluamos el perceptrón simple
    perceptron = Perceptron_simple()
    perceptron.fit(X_entrenamiento, S_entrenamiento)
    S_predecida = perceptron.prediccion(X_prueba)

    # Entrenamos y evaluamos el perceptrón simple de Keras
    weights = [np.reshape(perceptron.beginning_weights(), (2,1))]
    modelo = keras.models.Sequential()
    modelo.add(keras.layers.Dense(1, input_dim=2, use_bias=False, activation='sigmoid', weights=weights)) # U
    modelo.compile(optimizer=keras.optimizers.SGD(learning_rate=0.5), loss='mean_squared_error') # Regla Del
    modelo.fit(x=X_entrenamiento, y=S_entrenamiento, epochs=1, shuffle=True, verbose=0) # Una iteración
    S_predecida_keras = np.reshape(modelo.predict(x=X_prueba, verbose=0), -1) > 0.5

    # Comparamos los resultados con la salida real
    return {
        'N': N,
        'Precision PS': len(S_predecida[S_predecida == S_prueba]) * 100 / len(S_predecida),
        'Precision Keras': np.all(S_predecida == S_predecida), # Comparacion con Keras
```

```
        'Acierta E1': S_predecida[0] == S_prueba[0],
        'Acierta E2': S_predecida[1] == S_prueba[1],
        'Acierta E3': S_predecida[2] == S_prueba[2]
    }



# Obtención de los datos variando la N
values_N = np.arange(1, 10000 + 100, 100)
R = 700
for N in values_N:
    df = df.append(comparacion(N), ignore_index=True)

# Veces que acierta E1, E2, E3
fig = plt.figure(figsize = (6, 6))
Ei = np.array(['E1', 'E2', 'E3'])
PEi = np.array([len(df[df['Acierta E1']]), len(df[df['Acierta E2']]), len(df[df['Acierta E3']])]) / len(df)
plt.bar(Ei, PEi, color=['r','b', 'g'])
plt.ylabel('Porcentaje de acierto')
plt.ylim(0, 1)
plt.savefig("Diagrama de barras N.png")

# Porcentaje de acierto del perceptrón
fig = plt.figure(figsize = (6, 6))
PN = np.array([df['Precision PS']])
plt.scatter(values_N, PN, color='r')
plt.xlabel('N')
plt.ylabel('Porcentaje de acierto')
plt.savefig("Precision PS N.png")

# Comprobar que predicen lo mismo
print("Porcentaje de acierto de ambos:", len(df[df['Precision Keras']])*100 / len(df))
```