Práctica 1: Arquitecturas supervisadas: Back-propagation neural network

Benearo Semidan Páez Martín benearo.paez101@alu.ulpgc.es
Juan Sebastián Ramírez Artiles juan.ramirez107@alu.ulpgc.es
Alberto Ramos Sánchez alberto.ramos104@alu.ulpgc.es
Iru Nervey Navarro Alejo iru.navarro101@alu.ulpgc.es

Índice

Introducción

En esta memoria se pretende exponer y argumentar la implementación de la red neural ‘back-propagation’. Durante la misma iremos viendo los distintos puntos tratados durante el desarrollo de dicha red.

Tratamiento del dataset:

En primer lugar, mostraremos cómo realizamos la corrección y modificación del conjunto de datos proporcionados para la práctica.

Observando los datos, encontramos que existen filas con una o más columnas vacías (en realidad, contienen múltiples espacios), como mostramos en la ilustración 1.

2	6,1	86	clear	None	208	2,5	Dry	No
2	6,1	86	clear	None	208	2,5	Dry	No
4	6,1	86	clear	None	208	2,5	Dry	No
4	6,1	86	clear	None	208	2,5	Dry	No
2	6,1	86	clear	None	208	2,5	Dry	No
2	6,1	86	clear	None	208	2,5	Dry	No
2	6,1	86	clear	None	208	2,5	Dry	No
2			snow				Dry	No
2			snow				Dry	No
2			snow				Dry	No
4			snow				Dry	No
2			snow				Dry	No
2			snow				Dry	No
2			snow				Dry	No
2			snow				Dry	No
2			snow				Dry	No
2			snow				Dry	No
2			snow				Dry	No
2			snow				Dry	No
2			snow				Dry	No
2			snow				Dry	No
6			snow				Dry	No
2			snow				Dry	No
2			snow				Dry	No
2			snow				Dry	No
2			snow				Dry	No
2			snow				Dry	No
2			snow				Dry	No
2			snow				Dry	No
2			snow				Dry	No
2			snow				Dry	No
2			snow				Dry	No
2			snow				Dry	No

Ilustración 1 - Campos sin contenido

Estas filas carecen de los suficientes datos como para ser tomadas en cuenta, por lo que las eliminamos. Esto lo realizamos mediante la búsqueda y reemplazo de espacios por NaN y posteriormente su eliminado.

Después de esto, procedemos a eliminar la columna completa del ‘timestamp’, ya que no es un dato lo suficientemente relevante para su uso en el entrenamiento.

A continuación, buscamos todos los campos categóricos presentes en el dataset, los cuáles son:

TIPO_PRECIPITACION, INTENSIDAD_PRECIPITACION y ESTADO_CARRETERA y ACCIDENTE.

A excepción de ACCIDENTE, pasamos el resto de las columnas categóricas a ‘one-hot encoding’, es decir, creamos una columna nueva para cada posible valor del campo y le asignamos un 1 al que corresponde y 0 al resto de estos.

Con los valores de ACCIDENTE aplicamos un reemplazamiento para los valores de ‘Yes’ y ‘No’ por 1 y 0 respectivamente. Para aclarar el resultado, mostramos en la ilustración 2 un xls generado tras realizar esta operación.

L	M	N
precipitation_type_clear	precipitation_type_rain	precipitation_type_snow
1	0	0
1	0	0
1	0	0
1	0	0
1	0	0
1	0	0
1	0	0
1	0	0
1	0	0
1	0	0
1	0	0
1	0	0
1	0	0
1	0	0
1	0	0
1	0	0
1	0	0
1	0	0
1	0	0
1	0	0
1	0	0
1	0	0
1	0	0
1	0	0
1	0	0
1	0	0
1	0	0
1	0	0
1	0	0
1	0	0
1	0	0
1	0	0
0	1	0
0	1	0
0	1	0
0	1	0
0	1	0
0	1	0
0	1	0

Ilustración 2 - One-hot encoding

Con esto ya disponemos de un dataset que sabemos que dispone de valores para todas sus columnas y que dispone de datos que puede usar la red neuronal para sus cálculos.

No obstante, el conjunto de datos presenta otro inconveniente a parte de la falta de valores ocasionalmente: la densidad de accidentes con respecto a las de no-accidentes es muy baja. Si dejáramos el dataset tal cuál definimos hasta este momento, la red tendería a sobreajustarse a los casos en los que no se produce accidente, retornando en la mayoría de las predicciones que hiciéramos un 0 (no se producirá accidente).

Por ello, realizamos una selección de datos que coge la misma cantidad de filas con la columna ACCIDENTE a 0 que a 1.

Esto nos quita una gran parte del conjunto de datos (quedando en torno a 6000-7000 filas), pero disponen de una densidad adecuada de casos de accidente para que no se produzca sobreajuste.

Una vez solucionado el punto anterior, realizamos una normalización de los datos haciendo uso de la media y la desviación estándar de los datos. Con esto ponemos todos los valores en rangos estándar entre -2 y 2, permitiendo a la red realizar los cálculos mejor al reducir dichos rangos. ¿Por qué [-2, 2]? Esto es debido a que usamos la función sigmoide para la salida de las neuronas. El valor del sigmoide para -2 es muy cercano a 0 y el de 2, muy cercano a 1. Es importante constatar este rango, ya que, si lo estableciéramos entre 0 y 1, la salida del sigmoide sería entre 0.5 y 0.7 aproximadamente.

Con esto ya tenemos listo el conjunto de datos para ser usados. Solamente nos falta realizar la partición de los datos a ser usados para entrenamientos, pruebas y validación.

En nuestro caso, seleccionamos un 2.5% para pruebas, un 3% para validación y lo restante (un 94.5%) para entrenamiento.

Todo el procedimiento anteriormente descrito se realiza mediante la clase definida en ‘data_cleaner.py’ del código fuente.

Y con esto, concluye el tratamiento del dataset.

Implementación del Backpropagation:

Hemos decidido implementar el Backpropagation de forma matricial. Para ello nos hemos basado en el desarrollo teórico-práctico descrito en el capítulo 7 del libro “Neural Networks - A Systematic Introduction” de Raúl Rojas.

Más concretamente hemos utilizado el algoritmo descrito en el punto 7.3.3. Para una mayor comprensión de nuestra implementación describiremos la correlación entre nuestras variables y las descritas en el libro:

- La matriz $\boldsymbol{D_2}$ es para nosotros la matriz resultante de la expresión en el método ***d_error*** de la clase ***OutputLayer***:

  “D = np.eye(self.sigma_o.shape[0]) * (self.sigma_o * (1 - self.sigma_o))”

  La cual contiene las derivadas del sigmoide y se utilizarán en el cálculo del gradiente.

- El vector $\boldsymbol{o_n^{(2)}}$ es el vector ***sigma_o*** atributo de la clase ***OutputLayer***. Este vector es la salida final de la neurona, siendo el producto de las entradas por los pesos y el resultado pasado por el sigmoide. Para el caso particular de este trabajo el vector es de un solo elemento ya que la salida de la capa output solo tiene una neurona. El código sigue funcionando en los casos en los que la salida de la red tenga más de una neurona.

- La matriz $\boldsymbol{D_1}$ es el resultado de la llamada a la función ***d_error(self)*** de la clase ***HiddenLayer***. El caso básico que describe el libro utiliza una sola capa oculta. El cálculo para más de una capa oculta se escala utilizando una matriz ***D*** por cada capa oculta.

- El vector $\boldsymbol{o_n^{(1)}}$ es, como antes, la salida final de las neuronas de la capa oculta. La dimensión del vector es igual al número de neuronas de la capa.

- El vector ***e***, representa las derivadas del error cuadrático del conjunto de salidas de la capa output. La dimensión del vector es la misma que la de la salida de la red.

- El vector $\boldsymbol{\delta^{(2)}}$, resultado del producto matricial entre $\boldsymbol{D_2}$ y ***e***, es el retorno de la llamada al método ***d_error(self, s_esperada)*** de la clase ***OutputLayer***. Este método se llama desde el método ***fit*** de la clase ***BackPropagation***. Durante el entrenamiento, el valor se guarda en la primera posición de la variable local ***delta***, que es una lista que se utilizará para ir calculando el incremento de pesos de cada capa. La dimensión del vector será igual al número de neuronas de la capa output.

- $\boldsymbol{\delta^{(1)}}$ y los sucesivos deltas, si hubiera más de una capa oculta, se calcularán de forma recurrente mediante el producto de la matriz ***D*** por la matriz de pesos de la capa output, en el caso de la última capa oculta, o la matriz de pesos de la siguiente capa oculta en caso contrario. Finalmente la operación anterior se multiplica por el $\boldsymbol{\delta^{(2)}}$, en el caso de ser la última capa oculta, o el $\boldsymbol{\delta}$ siguiente en el caso de no ser la última.

  Este valor se apilará en la lista de la variable local ***delta***. Los valores de $\boldsymbol{\delta}$ que se vayan calculando, en los casos en los que haya más de una capa oculta, también se apilarán en la lista ***delta***.

- Las matrices de incremento de pesos $\boldsymbol{\Delta W_n^T}$ se irán apilando en la variable local ***delta_pesos***. El incremento de pesos se irá calculando a medida que se vayan obteniendo los valores de $\boldsymbol{\delta}$. Los ***delta_pesos*** se obtienen mediante el producto del ***learning rate*** por el producto de los vectores $\boldsymbol{\delta}$ de la capa y el vector de entradas con el bias $\boldsymbol{\hat{o}}$.

- Por último, se actualizan todos los pesos de la red restando los $\boldsymbol{delta\_pesos^T}$ a las matrices de pesos de las capas ocultas y de salida. La matriz ***delta_pesos*** de cada capa se almacenó transpuesta así que hay que transponerla para que opere correctamente con las matrices de pesos pertinentes.

Mejoras al proyecto:

Hemos considerado conveniente implementar tres mejoras al proyecto.

La primera consiste en la utilización de un ***learning rate*** adaptativo, el cual va disminuyendo a la razón de 0.99995 en cada iteración a partir de un nivel de error que consideramos cerca del óptimo.

Hemos utilizado como error de referencia la norma infinito de la resta vectorial del vector de salida y la salida esperada. Consideramos un nivel de error óptimo a aquel que sea menor a 0.1.

La segunda mejora consiste en la implementación de un sistema de salvado y carga de pesos. De esta manera no es imprescindible entrenar continuamente la red para poder usarla, en el caso de que se nos presenten nuevas muestras a clasificar. El sistema de salvado de pesos tiene en cuenta la arquitectura de la red y el número de iteraciones que se utilizó en el entrenamiento.

La última mejora se puede considerar menor y consiste en el filtrado en los métodos ***predict*** y ***error*** de la clase ***BackPropagation***, de modo que no se permita obtener resultados de ellos si la red no tiene cargados los pesos, fruto del entrenamiento o de la carga de pesos obtenidos en un anterior entrenamiento.

Resultados:

Para el cálculo de la precisión de la red hemos utilizado los valores devueltos por ***predict*** para muestras de prueba y hemos considerado que si el valor devuelto es menor o igual a 0.1 podemos aceptarlo como un 'No accidente', mientras que si el valor devuelto es mayor o igual a 0.9 lo aceptamos como 1 y por lo tanto como 'Accidente'.

Si los valores de etiqueta no coinciden con los de predicción o se produjeron valores fuera de rango se considera como error ya que la red no fue capaz de clasificar. En el primer caso erró por completo y en el segundo no clasificó con seguridad.

Siguiendo este criterio y para valores de prueba no usados ni para entrenamiento o validación hemos obtenido una precisión superior al 90% utilizando una tasa de aprendizaje inicial de 0.01 y realizando en torno a 150 iteraciones.

La red converge para esta tasa de aprendizaje a partir de 150 iteraciones adoptando un comportamiento asintótico más allá de esta cantidad. Ver ilustración 3.

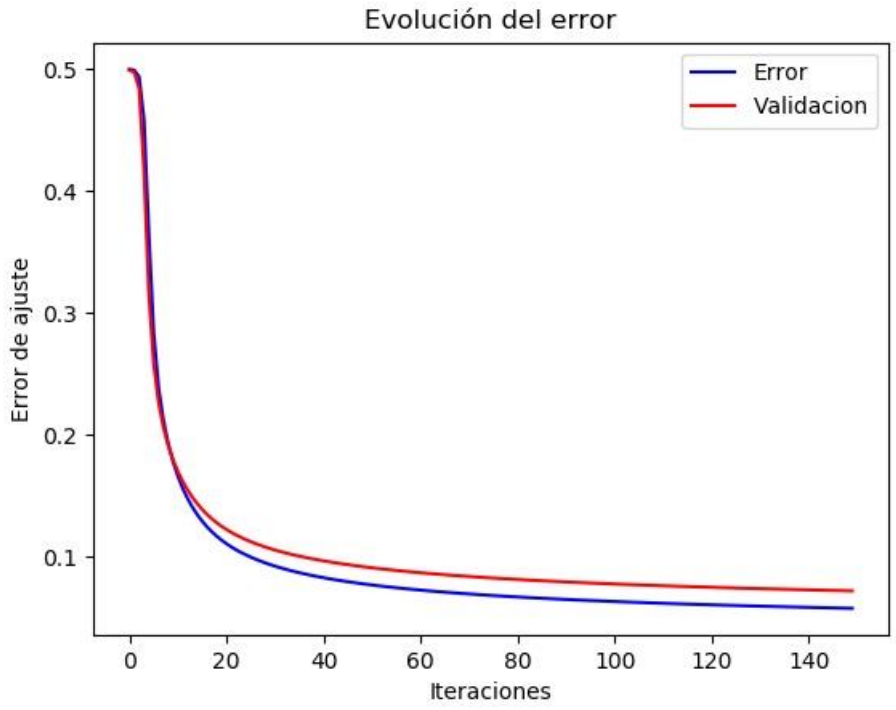

Ilustración 3

Sin embargo, para pruebas con una tasa de aprendizaje de 0.1, la red converge bastante antes, dando una precisión entre 86% y 93% para 25 iteraciones. Véase la ilustración 4.

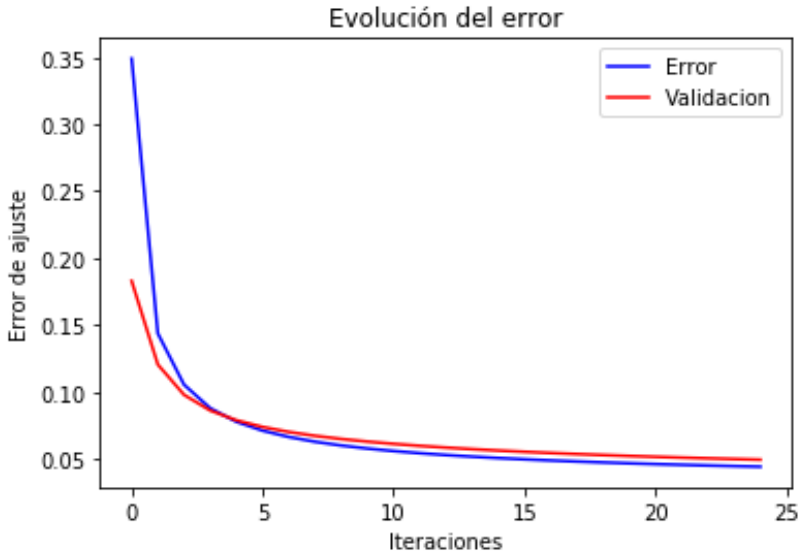

Ilustración 4

Para 100 iteraciones con la tasa de aprendizaje de 0.1, alcanzamos el rango 95 - 97%, siendo este el mejor valor que alcanza nuestra red. En la ilustración 5 mostramos la gráfica de evolución del error para este caso.

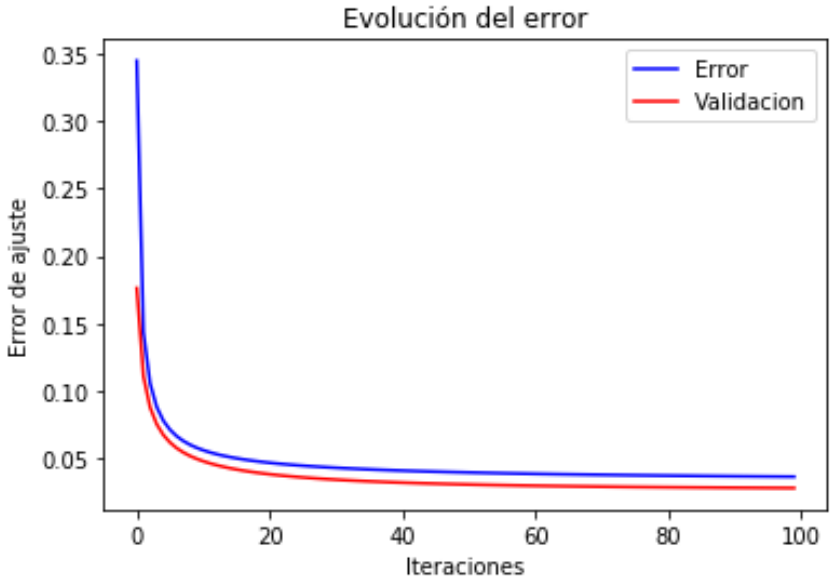

Ilustración 5

Finalmente, queríamos destacar que, bajo la misma tasa de aprendizaje que antes, la red neuronal tiende a dejar de mejorar a partir de 200 iteraciones, como mostramos en la ilustración 6.

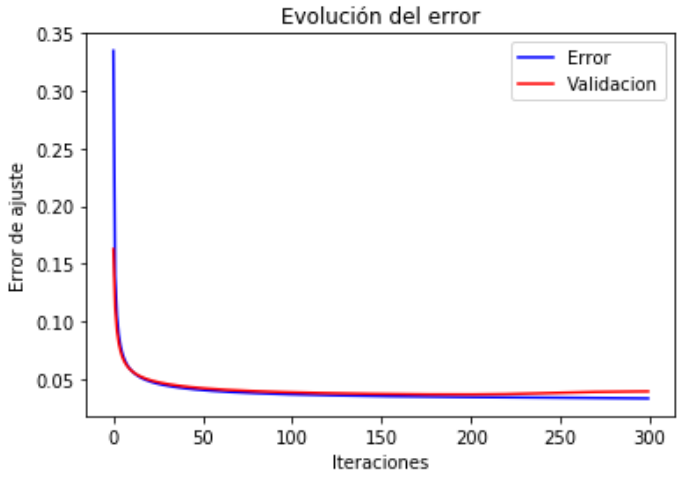

Ilustración 6

Por otra parte, para las predicciones anteriormente mencionadas, devolvemos una salida por pantalla con el siguiente formato:

[Resultado] 'color' => [Valor real]['1 o 0']

Donde '1 o 0' es en función de la etiqueta en el conjunto de datos y 'color' los establecemos según la salida de la red de la siguiente manera:

Verde	Entre 0 y 0.1
Naranja	Entre valores > 0.1 y 0.5
Rojo	Entre valores > 0.5 y 1

Referencias

["Neural Networks - A Systematic Introduction" de Raúl Rojas]