Redes de Neuronas

Práctica 1
Problema de Regresión
Curso 2020-2021

Ramiro Leal Meseguer
100346124
100346124@alumnos.uc3m.es

Claudia Fernández Estebaranz
100366696
100366696@alumnos.uc3m.es

Índice

1. Introducción

El problema planteado en esta práctica consiste en predecir el precio medio de la vivienda para los distintos distritos de California, a partir del censo nacional de 1990. Partimos de un conjunto de 17000 instancias que todas ellas cuentan con los siguientes 8 atributos que determinan el precio medio de la vivienda.

- longitude: Indica cuán al oeste se encuentra una casa.
- latitude: Indica cuán al norte se encuentra una casa.
- housingMedianAge: Antigüedad media (mediana) de una casa dentro de un distrito.
- totalRooms: Cantidad total de habitaciones en las casas de un distrito.
- totalBedrooms: Cantidad total de camas en las casas de un distrito.
- population: Cantidad total de residentes en un distrito.
- households: Cantidad total de grupos familiares (conjunto de personas que reside en una misma casa) en un distrito.
- medianIncome: Ingreso medio (mediana) de los grupos familiares de un distrito (medido en decenas de miles de dólares estadounidenses).

El objetivo de esta práctica es, mediante dos modelos de redes de neuronas supervisados, predecir el precio medio de la vivienda. Los modelos que utilizaremos serán Adaline y perceptrón multicapa. Partiremos de unos datos de entrada que contienen los atributos previamente comentados. A partir de estos procederemos al procesado de datos inicial obteniendo los datos con los que trabajaremos en los dos modelos. Para el modelo de Adaline , se realizará tanto el desarrollo del código como la búsqueda del mejor entrenamiento de la red con estos datos de entrada. Sin embargo, para el perceptrón multicapa partiremos del *script* proporcionado, y a través del *RStudio* realizaremos un análisis de los mejores modelos. Los resultados de ambos modelos se compararán para concluir en las ventajas y desventajas de cada uno.

2. Procesado de datos

Previo a llevar a cabo el aprendizaje de las redes, hay que realizar el procesado de los datos. Para dicha tarea hemos decidido utilizar Python como lenguaje de programación, ya que con el se manejan los datos de manera sencilla y, además, con este desarrollamos el modelo Adaline.

Como nos marca el enunciado de la práctica, el procesado de datos tiene tres fases distintas:

- 1) Normalización de todos los valores en el rango [0,1] teniendo en cuenta los máximos y mínimos de cada variable.
- 2) Aleatorización de los datos para poder llevar el entrenamiento de las redes de manera adecuada.
- 3) Separación de los datos en tres conjuntos distintos.

El fichero con el nombre *"Preparación_Datos.py"* es el encargado de realizar lo que acabamos de mencionar anteriormente. Tras su ejecución, dados unos datos iniciales de entrada, generará en la carpeta *"/DatosProcesados"* los siguientes ficheros:

- ***"DatosEntrada_MaximoMinimo.txt"***: En el se guardarán los valores máximos y mínimos de cada una de las columnas del fichero de datos de entrada. Estos valores se van a utilizar posteriormente para la desnormalización de los datos una vez que se haya entrenado a la red.
- ***"DatosEntrada_[+ conjunto de datos].txt"***: Tres ficheros, uno para cada conjunto de datos (entrenamiento, validación y test). Cada uno de estos ficheros va a tener un porcentaje especifico de los datos ya aleatorizados y normalizados.

2.1. Normalización

La fase de normalización consiste en cambiar los valores de los datos de entrada a una escala común sin distorsionar las diferencias entre los valores de diferentes rangos. Este proceso solo es necesario cuando las entradas están en distintas escalas. Un problema que se puede presentar si no normalizamos es la gran variación de valores que se produce en el descenso del gradiente, ocasionando que la búsqueda de un mínimo global/local sea más complicado. Es por esto que, con valores normalizados, la búsqueda se facilita.

Al ejecutar nuestro fichero *"Preparación_Datos.py"* estamos llevando la normalización de los datos de entrada en un rango de valores [0,1] y, además, generamos un fichero almacenando los máximos y mínimos de cada columna para después poder hacer el proceso inverso y desnormalizarlos.

2.2. Aleatorización

Con la aleatorización se busca evitar cualquier posible sesgo durante el entrenamiento de la red. Al separar nuestros datos en tres conjuntos, queremos que estos sean representativos de la distribución general de los datos, ya que si uno de estos conjuntos tuviera datos muy parecidos, al hacer el entrenamiento sobre la red, no se podría generalizar una predicción para datos que no estuvieran en ese conjunto por el sesgo presente en ellos.

2.3. Conjuntos de datos

Como hemos mencionado anteriormente, tenemos la necesidad de crear una red que generalice bien. Para conseguir eso, necesitamos separar nuestros datos en tres conjuntos.

- **1) Entrenamiento:** Este conjunto contendrá el 60 % de los datos de entrada para el entrenamiento de la red.
- **2) Validación:** Este conjunto contendrá el 20 % de los datos de entrada para una evaluación de la red hasta conseguir un buen modelo que nos pueda servir para luego probar con Prueba/Test.
- **3) Prueba/Test:** Este conjunto contendrá el 20 % de los datos de entrada para una evaluación final de la red.

3. Modelos

3.1. Adaline

El modelo de Adaline es un tipo de red neuronal desarrollada por Widrow y Hoff basado en las neuronas de McCulloch-Pitts. Generalmente, se compone de una sola capa con n neuronas con m entradas cada una. Cada neurona, tiene un vector de pesos asociado y la salida de esta vendrá dada por la siguiente función:

$$y = \sum_{i=1}^{N} x_i * w_i + \theta \tag{1}$$

A diferencia del perceptrón multicapa, Adaline tiene en cuenta la diferencia entre la salida producida (y^p) y la deseada (d^p) de cada entrada, esta diferencia la denominaremos **delta**. La red se apoyará en esta diferencia para ajustar los pesos de la red, y así mejorar sus predicciones, este proceso de ajuste de pesos lo denominamos **descenso del gradiente**. Este último, a su vez, también se verá influenciado por la **razón de aprendizaje** (α), que evitará que la red de cambios bruscos en su búsqueda. Por tanto, por cada entrada aplicaremos la siguiente función:

$$\Delta_p w_j = \alpha * (d^p - y^p) * x_j \tag{2}$$

3.1.1. Implementación

Para la implementación de Adaline hemos utilizado dos ficheros, *main.py* y *Adaline.py*. En la implementación del *main.py* hemos incluido funciones para abrir, leer y escribir ficheros, además de la función que se encargará de gestionar las llamadas a Adaline con las distintas razones de aprendizaje que probaremos.

En el fichero de *Adaline.py* se ha implementado la red completa. A partir de la función *entrenamiento()* se irá llamando al resto de funciones para ir ejecutando la red de Adaline. Partiremos de un modelo inicial, donde los pesos y el umbral tienen asignados valores aleatorios entre (-1,1), la razón de aprendizaje se ha definido en la llamada al método *entrenamiento* y, el número máximo y mínimo de ciclos está predefinido con 3000 y 1000 pero puede ser modificado también a la hora de hacer la llamada.

Una vez tenemos el modelo inicial, comenzamos con el entrenamiento. Recorreremos un bucle, que como máximo se ejecutará el número máximo de ciclos introducido, y dentro de ese bucle, a su vez, recorreremos otro con una longitud igual a la de los datos de entrenamiento. Por cada ciclo, entrenaremos a la red con todos estos datos. Para hacer esto, primero, por cada entrada calcularemos la salida producida con los pesos del modelo actual mediante la función [1]. Y después, ajustaremos los pesos mediante la función *descenso del gradiente*, explicada en [2]. Una vez terminemos un ciclo, procedemos a calcular los errores producidos en este, y los calcularemos tanto para los datos de entrenamiento como para los de validación. En nuestro caso, hemos considerado el cálculo tanto de MSE ($\frac{\sum_{i=1}^{N}(d[i]-x[i])^2}{N}$) como del MAE ($\frac{\sum_{i=1}^{N}|d[i]-x[i]|}{N}$) para tener unas comparaciones más completas. Para calcular estos errores, lo primero que haremos será, con el modelo final obtenido de ese ciclo, volver a calcular todas las salidas producidas y con estas procederemos a la llamada de la función *error*, que nos devolverá ambos. Todo este proceso se realizará n ciclos, llegando en su caso hasta el máximo o hasta que se cumpla la condición de parada. Esta condición consiste en haber superado el número mínimo de ciclos y de que los cambios en los errores de validación no hayan cambiado más de un 0,0001 en los últimos n ciclos. En nuestro caso, hemos considerado que esa n será el número mínimo de ciclos a ejecutar.

Una vez se obtiene el modelo final con esa razón de aprendizaje, volveremos al fichero *main.py* donde se procederá a escribir tales resultados en sus ficheros correspondientes y a ejecutar el fichero test en el modelo final. Por cada razón de aprendizaje, se generará, dentro de la carpeta *salidasProducidas* una nueva carpeta "DatosEntrada - *razonAprendizaje*". Dentro de esta podemos encontrar 3 ficheros:

- errores.txt: Contendrá los errores MSE y MAE producidos con los ficheros de entrenamiento y validación de todos los ciclos y en el último ciclo, se añadirán, además de estos errores, también los del test. En este fichero podemos ver la evolución del entrenamiento.
- pesos.txt: Contendrá los pesos asignados a cada atributo, es decir, el modelo final obtenido.
- testSalidas.txt: Contendrá la salida producida y la salida esperada de cada entrada del fichero test desnormalizada, para poder comparar de una manera más realista los resultados obtenidos.

A su vez, al finalizar la ejecución se imprimirá por pantalla los errores finales obtenidos.

3.1.2. Pruebas

En la parte de pruebas de Adaline, se pueden considerar infinitos casos. Nosotros, a través de los ciclos máximos y mínimos y las razones de aprendizaje hemos tratado de jugar con los distintos modelos hasta obtener el que mejor errores producía. Hemos tenido en cuenta a la hora de comparar los modelos el error MSE del test producido además de cuantos ciclos necesitaba para entrenar. Esto se debe a que podríamos considerar mejor un modelo que tarde menos en ejecutar y produzca un error ligeramente mayor que otro que esté más tiempo ejecutando. Comentar que en Adaline no vamos a jugar con el parámetro *seed* dado que en este solo podemos encontrar un mínimo global sin importancia de donde empecemos la búsqueda.

Todas las pruebas realizadas las podemos encontrar en la carpeta *adaline* dentro de *salidasProducidas*. Aquí encontraremos distintas carpetas diferenciadas por el número de ciclos con el que se han ejecutado y dentro de estas encontramos las distintas pruebas según las razones de aprendizaje aplicadas. Por otra parte, encontramos un .txt denominado *adaline - PruebasReducidas* donde encontramos un resumen de los errores producidos en todas las pruebas, para una visión más sencilla.

En la siguiente tabla, mostraremos la mejor razón de aprendizaje según el número de ciclos con el que se ejecuta.

Mejores modelos por nº ciclos/razón				
Min	**Max**	**Razón**	**TestMSE**	**TestMAE**
2000	10000	5e-05	0.02091	0.10613
1000	3000	0.0002	**0.02080**	**0.10547**
500	5000	0.0001	0.02084	0.10551
200	2000	0.001	0.02103	0.10431

Con los resultados anteriores llegamos a las siguientes conclusiones. El caso (200, 2000) con razón = 0.001, además de tener un error más alto respecto a los demás, en su ejecución no llega hasta el máximo de ciclos, para en el ciclo 1062. Este mayor valor del error podría ser porque se ejecuta en menos ciclos que el resto lo que le podría estar penalizando, ya que en el caso del MAE obtiene el mejor error. Es por esto que consideramos analizar el error de esta razón con otros valores de ciclos. Observamos que el error nunca consigue disminuir en 0.2123, por lo que descartaremos esta razón. En el resto de casos, observamos unos errores muy similares entre todos, sin embargo, (1000, 3000) con razón = 0.0002 es el que tiene menor error en ambos casos. Además, el caso (2000, 10000) con razón = 5e-05, tiene una ejecución muy larga, que debería ser más óptimo debido a que la razón es muy pequeña, sin embargo, no conseguimos unos resultados mucho mejores a pesar de la cantidad de tiempo invertido. Por tanto, como modelos finales podríamos utilizar tanto (1000, 3000) con razón = 0.0002 y (500, 5000) con razón = 0.0001. Por elegir uno de ellos, **nos decantaremos por el de (1000, 3000)** debido a que es aquel con el error un poco más pequeño que el otro.

Mejor modelo Adaline								
Min	**Max**	**Razón**	**TrainMSE**	**TrainMAE**	**ValidMSE**	**ValidMAE**	**TestMSE**	**TestMAE**
1000	3000	0.0002	0.02110	0.10652	0.01970	0.10462	0.02080	0.10547

Podemos observar que el error de validación es ligeramente menor al de entrenamiento lo que nos lleva a pensar que la capacidad de generalización es correcta. El error de validación comienza en 0.13538 y se consigue disminuir 0.01970, lo que muestra una buena dirección del aprendizaje. Sin embargo, si miramos el fichero de salidas del test podemos ver que en algunos casos la predicción no es muy cercana a la esperada por lo que podríamos decir que se trata de una red un tanto pobre debido a su topología. Cuando comparemos con el perceptrón podremos observar estas diferencias.

Por último, nos gustaría comentar ciertos aspectos:

- Cuando aplicábamos razones de aprendizaje muy altas (a partir de 0.1) o muy bajas (e-06) el error aumentaba considerablemente. Esto seguramente se produzca porque la derivada la razón de aprendizaje hace que el error aumente en vez de descender.
- Un beneficio de las razones de aprendizaje altas es que el número de ciclos necesarios, en comparación con una razón baja, es mucho más bajo. Sin embargo, las razones bajas, que requieren más ciclos para tener un buen resultado, obtienen mejores resultados dado que es más difícil que se pasen el mínimo.
- En muchos casos obtenemos modelos cuyos MAE son mejores que en otros modelos, sin embargo el error MSE es peor. Esto se debe a la manera en la que se calculan los errores, dado que si hacemos una predicción muy mala, en MSE, el cuadrado de la diferencia hará que el error sea

aun más destacable llegando a sesgar este error y sobre-estimando la maldad del error. Este tipo de problemas se suelen producir cuando tenemos datos ruidosos, es decir, poco fiables. Por el contrario, lo mismo puede ocurrir pero al revés, si tenemos errores muy pequeños podemos llegar a subestimar la maldad del modelo si nos fijamos en el MSE.

El MAE es un tipo de error que no penaliza de la misma manera los errores, lo hace de una manera más leve, por lo tanto, no es tan sensible a los valores atípicos.

Es por esto, que para analizar nuestros modelos, miraremos ambos errores y trataremos de que en ambos haya un valor coherente que evite que subestimemos o sobre-estimemos la maldad del modelo. Sin embargo, ante una disparidad entre los errores, nos basaremos antes en el MSE porque consideramos que no tenemos datos poco fiables y, que es mejor ser estrictos y penalizar más cuando la diferencia es mayor, pudiendo así mejorar más el modelo.

3.2. Perceptrón multicapa

El perceptrón multicapa es una red neuronal formada por múltiples capas, a diferencia de Adaline, de esta manera tiene la capacidad de resolver problemas que no son linealmente separables. Debido a la complejidad de esta red, para su estudio se nos ha ofrecido un *script* que lo implementa. En este *script* se pueden modificar parámetros de entrada como la seed, el número de ciclos máximos, la topología, la razón de aprendizaje y el shufflePatterns para tratar de encontrar el mejor modelo. Este *script* devuelve el error producido MSE con todos los ficheros de entrada, un gráfico que muestra la evolución del error y cinco ficheros donde se almacenan los distintos resultados. Nosotros además hemos añadido que calcule el error MAE, dado que en Adaline también lo calculamos y así podemos compararlos.

Con el objetivo de encontrar la red que obtenga el mejor modelo, se han realizado múltiples pruebas siguiendo un orden muy concreto, para poder ir avanzando con lógica en los resultados que obteníamos. En la sección de pruebas explicaremos el procedimiento seguido, sin embargo, solo mostraremos los resultados más significativos para evitar extendernos demasiado. Se pueden encontrar todas las pruebas realizadas en el fichero *perceptronMulticapa-Pruebas* de la carpeta *perceptrón multicapa* entregada.

3.2.1. Pruebas

Las primeras pruebas que hicimos fueron, a través de un modelo muy básico, tratar de encontrar la *seed* que mejor se ajustara a los resultados. Decidimos ajustar este parámetro el primero pues de esta manera comenzaremos desde el mejor punto de partida que nos permita encontrar el mínimo global/local. Una *seed* alejada de este mínimo puede producir que este no se encuentre y por ello, que el entrenamiento no sea correcto, obteniendo errores mayores. Tras múltiples pruebas, obtuvimos que aquella donde ***seed=9*** daba los mejores resultados, es por esto, que todas las siguientes pruebas siempre tendrán esa *seed*.

Tras el ajuste de la *seed* procedimos al **análisis de la topología**, donde estudiaremos modelos de una, dos y tres capas con distinto número de neuronas en cada capa. Para estas pruebas siempre hemos utilizado una razón de aprendizaje = 0.2, unos ciclos máximos = 2000 y una seed = 9. Después de realizar distintas combinaciones, hemos seleccionado el mejor modelo obtenido en cada caso:

Mejores modelos por nº de capas						
Topología	**TrainMSE**	**TrainMAE**	**ValidMSE**	**ValidMAE**	**TestMSE**	**TestMAE**
c(20)	0.01369676	0.07859100	0.01260722	0.07772082	0.01330404	0.07830930
c(10, 30)	0.01181417	0.07198496	0.01134853	0.07283097	0.01178809	0.07265878
c(8, 15, 10)	0.01210978	0.07284893	0.01169826	0.07391304	0.01217111	0.07399874

A partir de estas topologías procederemos a realizar un **análisis de las razones de aprendizaje.** Probaremos distintas razones para estas tres topologías y seleccionaremos las tres mejores pruebas de

cada una de estas. Comentar, que al igual que en las pruebas de topología aplicaremos una seed = 9 y un número máximo de ciclos de 2000. Dicho esto, en la siguiente tabla procedemos a mostrar las topologías junto con las razones de aprendizaje que mejores resultados obtuvieron:

Mejores modelos por razón de aprendizaje							
Topología	**Razón**	**TrainMSE**	**TrainMAE**	**ValidMSE**	**ValidMAE**	**TestMSE**	**TestMAE**
c(20)	**0.4**	0.01279152	0.07790416	0.01187199	0.07687270	**0.01301184**	0.07876224
c(20)	0.2	0.01369676	0.07859100	0.01260722	0.07772082	0.01330404	0.07830930
c(20)	0.8	0.01320628	0.07610855	0.01271359	0.07669464	0.01339833	0.07633210
c(10, 30)	**0,2**	0.01181417	0.07198496	0.01134853	0.07283097	**0.01178809**	0.07265878
c(10, 30)	0,4	0.01186794	0.07214903	0.01155062	0.07337529	0.01205120	0.07321817
c(10, 30)	0,7	0.01169927	0.07231058	0.01145786	0.07344299	0.01224731	0.07419796
c(8, 15, 10)	**0,8**	0.01144463	0.07151688	0.01130207	0.07342304	**0.01159215**	0.07383243
c(8, 15, 10)	0,3	0.01203501	0.07229686	0.01152940	0.07315565	0.01192629	0.07307603
c(8, 15, 10)	0,2	0.01210978	0.07284893	0.01169826	0.07391304	0.01217111	0.07399874

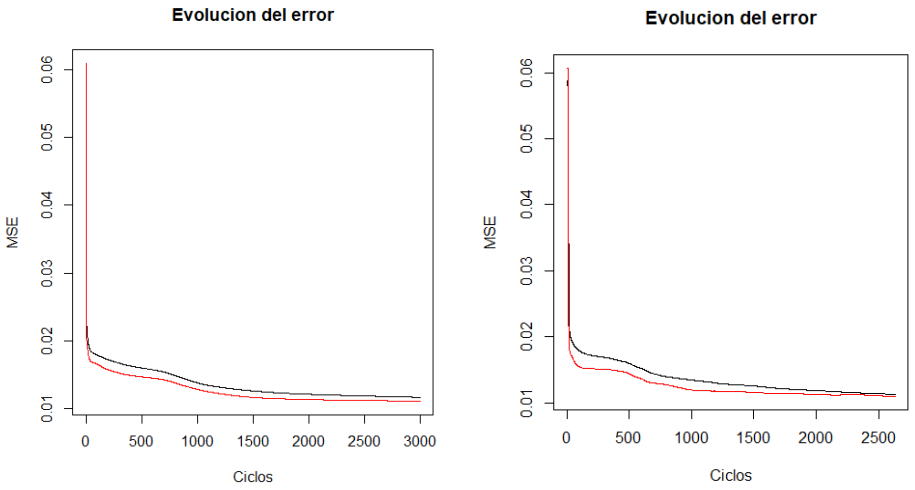
Por cada topología seleccionaremos un modelo basándonos en el resultado de *TestMSE*. En el caso de **c(20) cogeremos la razón de 0.4, con c(10, 30) una razón de 0.2 y con c(8, 15, 10) escogeremos la razón 0.8**. Por tratar de mejorar y seleccionar el mejor modelo de estos tres, probamos a cambiar el número de ciclos máximo, que es el único parámetro que aun no hemos tocado. Obtuvimos los siguientes resultados:

Mejores modelos por nº ciclos máximos							
Topología	**CiclosMax**	**TrainMSE**	**TrainMAE**	**ValidMSE**	**ValidMAE**	**TestMSE**	**TestMAE**
c(20)	2000	0.01279152	0.07790416	0.01187199	0.07687270	0.01301184	0.07876224
c(20)	3000	0.01290388	0.07631259	0.01193047	0.07557891	**0.01285058**	0.07738093
c(20)	4000	0.01537149	0.08436021	0.01437292	0.08406757	0.01486663	0.08427866
c(10, 30)	2000	0.01181417	0.07198496	0.01134853	0.07283097	0.01178809	0.07265878
c(10, 30)	**3000**	0.01140376	0.07110930	0.01113092	0.07267903	**0.01159376**	**0.07221078**
c(10, 30)	4000	0.01133270	0.07092045	0.01112652	0.07272640	0.01159860	0.07219729
c(8, 15, 10)	2000	0.01144463	0.07151688	0.01130207	0.07342304	0.01159215	0.07383243
c(8, 15, 10)	**3000**	0.01079887	0.07007974	0.01097232	0.07209836	**0.01137000**	**0.07256904**
c(8, 15, 10)	4000	0.01063748	0.06910320	0.01076106	0.07156259	0.01154002	0.07324502

Con estos últimos resultados podemos observar que en algunos casos aumentar en exceso el número de ciclos no mejora el error si no que lo empeora. Marcados en negrita tenemos los tres modelos con los que obtenemos mejores resultados para cada topología. Podemos observar que la topología con una capa resulta en un error mayor que las otras dos topologías. Sin embargo, las topologías de dos y tres capas muestran errores muy similares y, ambos son válidos. Podemos observar la tendencia de los errores de ambos modelos en la figura 1.

Tras el análisis del error de ambos modelos miraremos también los datos predichos por las redes. Estos ficheros los podemos encontrar en la carpeta entregada de *perceptrón multicapa* en *Modelo final*. Si en concreto miramos el fichero *netOutputtest* podemos comprobar que las predicciones son bastante similares o cercanas a las esperadas.

Nos gustaría comentar que a la hora de realizar las pruebas tratamos de activar *shufflePatterns*, que se encarga de desordenar en cada ciclo los patrones de entrenamiento y en principio evitaría el *overfitting* ya que no permitiría que la red se adaptara al orden en el que están los datos. Inicialmente, a medida que aumentábamos la razón de aprendizaje el error que se producía cada vez era menor, sin embargo, si realizamos un análisis de los resultados después del entrenamiento, podíamos observar en el gráfico que el error de validación oscilaba demasiado y se volvía inestable, como podemos observar en 2. Es por esto que decidimos descartar todas las pruebas que hicimos con *shufflePatterns* activado.



(a) c(10, 30), ciclosMax=3000, razón = 0.2 **(b)** c(8, 15, 10), ciclosMax=3000, razón = 0.8

Figura 1: Error MSE entrenamiento (negro) y validación (rojo)

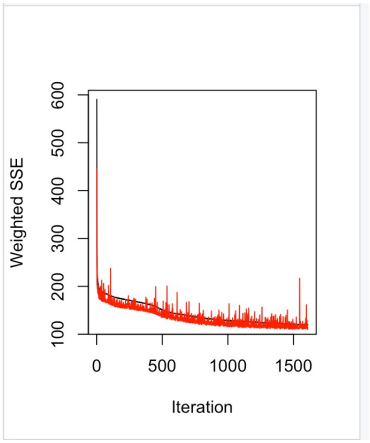


Figura 2: Gráfico errores con shufflePatterns = true

Por último, también comentar que el número máximo de capas que hemos puesto es tres, pues a partir de esta, el perceptrón no funciona correctamente, debido a que la derivada de la función de activación será muy pequeña y, cuando se produzca el back-propagation, a las primeras capas no les afectará casi el aprendizaje.

4. Conclusiones

Adaline posee la ventaja de que su gráfica de error es un hiperparaboloide que posee o bien un único mínimo global, o bien una recta de infinitos mínimos, todos ellos globales. Esto evita la gran cantidad de problemas que da el perceptrón a la hora del entrenamiento debido a que su función de error posee numerosos mínimos locales. Como ventaja del perceptrón sobre Adaline tenemos que al conectar las neuronas a través de funciones de activación no lineales, podemos crear límites de decisión mucho más complejos y no lineales que nos permiten abordar problemas en los que las diferentes clases no son linealmente separables.

De manera general, queremos aclarar que no hay un modelo mejor o peor, si no que este debe seleccionarse en función del problema que se este resolviendo. Si que esta claro, que el perceptrón es un modelo más manejable con el que se puede jugar con muchas variables (la topología de la red, las funciones de activación que aplicamos...), es por esto que suele resolver problemas más haya de los lineales.

En nuestro caso, si comparamos Adaline con el perceptrón podemos ver que será el perceptrón quien nos ofrezca un mejor modelo, dado que el error es mucho menor y además, las predicciones que realiza son mucho más cercanas al real.

Mejor modelo Adaline					
TrainMSE	**TrainMAE**	**ValidMSE**	**ValidMAE**	**TestMSE**	**TestMAE**
0.02110	0.10652	0.01970	0.10462	0.02080	0.10547

Mejores modelos Perceptón					
TrainMSE	**TrainMAE**	**ValidMSE**	**ValidMAE**	**TestMSE**	**TestMAE**
0.01140376	0.07110930	0.01113092	0.07267903	0.01159376	0.07221078
0.01079887	0.07007974	0.01097232	0.07209836	0.01137000	0.07256904

Por último, destacar que en esta práctica no nos hemos encontrado muchos problemas, simplemente es una práctica que requiere mucho trabajo de análisis y comparación de pruebas. Muy a nuestro pesar, en la memoria no hemos mostrado la gran mayoría de las pruebas realizadas para no extenderla y pasarnos del máximo, pero todas estas están incluidas en la carpeta entregada para mostrar el trabajo y estudio que hemos realizado sobre este.

Referencias

[1] J. D. M. Antonio J. Serrano, Emilio Soria. Redes neuronales artificiales. http://ocw.uv.es/ingenieria-y-arquitectura/1-2/libro_ocw_libro_de_redes.pdf, 2009-2010.

[2] C.-T. Chen and W.-D. Chang. A feedforward neural network with function shape autotuning. *Neural Networks*, 9(4):627 – 641, 1996.

[3] D. de Informática. Preparación de datos para el aprendizaje medidas de evaluación. https://aulaglobal.uc3m.es/pluginfile.php/4020144/mod_resource/content/1/PreparacionDatos_Evaluaci%C3%B3n.pdf, 2020.

[4] D. de Informática. Práctica 1 (parte 1): Programación de adaline. https://aulaglobal.uc3m.es/pluginfile.php/4020174/mod_resource/content/2/Programaci%C3%B3n%20de%20Adaline.pdf, 2020.

[5] D. de Informática. Práctica 1: Problema de regresión. predicción del precio medio de la vivienda en california. https://aulaglobal.uc3m.es/pluginfile.php/4020153/mod_resource/content/2/Practica1-2020-21.pdf, 2020.

[6] D. de Informática. Práctica 1: Usando rsnns. https://aulaglobal.uc3m.es/pluginfile.php/4061479/mod_resource/content/2/RSNNS.pdf, 2020.

[7] U. Jaitley. Why data normalization is necessary for machine learning models. https://medium.com/@urvashilluniya/why-data-normalization-is-necessary-for-machine-learning-models-681b65a05029, 2018.

[8] sitiobigdata. Aprendizaje automatico y las metricas de regresión. https://sitiobigdata.com/2018/08/27/machine-learning-metricas-regresion-mse/.

[9] S.-H. Wu. Perceptron, adaline, and optimization. https://nthu-datalab.github.io/ml/labs/04-1_Perceptron_Adaline/04-1_Perceptron_Adaline.html, 2019.