FACULTADE DE MATEMÁTICAS

Traballo Fin de Grao

Optimización de hiperparámetros

Nicolás Álvarez Rodríguez

Septiembre, 2023

UNIVERSIDADE DE SANTIAGO DE COMPOSTELA

GRAO DE MATEMÁTICAS

Traballo Fin de Grao

Optimización de hiperparámetros

Nicolás Álvarez Rodríguez

Septiembre, 2023

UNIVERSIDADE DE SANTIAGO DE COMPOSTELA

Trabajo propuesto

Área de Coñecemento: Matemática aplicada
Título: Optimización de hiperparámetros
Breve descrición do contido
Os hiperparámetros son parámetros que afectan ao comportamento dun algoritmo. Por exemplo, os pesos dunha rede neuronal ou os parámetros de configuración dun algoritmo de optimización. A obtención de valores óptimos para eles é fundamental para que o algoritmo se adapte ao conxunto de problemas nos que se pretende usar. Neste traballo preténdese revisar a bibliografía e analizar as técnicas actuais máis utilizadas neste campo.

Índice

Resumen viii

Introducción xi

1. Generalidades 1

1.1. Neuronas . . . 1

1.2. Redes neuronales . . . 3

1.3. Entrenamiento de una red neuronal . . . 7

1.3.1. Algoritmo del perceptrón . . . 7

1.3.2. Regla delta . . . 12

1.3.3. Retropropagación . . . 13

1.4. Hiperparámetros de una red neuronal . . . 15

1.5. Optimización de hiperparámetros . . . 22

1.6. Métodos . . . 24

1.6.1. Barrido paramétrico . . . 24

1.6.2. Búsqueda aleatoria (*Random Search*) . . . 25

1.6.3. Optimización Bayesiana . . . 25

1.6.4. Otros métodos . . . 28

2. Un ejemplo en R 31

3. Conclusiones **35**

I. Procesos Gaussianos **37**

II. Código **39**

Bibliografía **45**

Resumen

El objetivo de este trabajo es introducir al lector al problema de la optimización de hiperparámetros (HPO) desde las bases del aprendizaje automático, con especial atención al caso de las redes neuronales unidireccionales. Se comienza exponiendo de manera formal el funcionamiento de las neuronas artificiales y redes neuronales, particularizando el caso del perceptrón multicapa. Se continúa exponiendo el proceso de entrenamiento de una red, desde los algoritmos más sencillos hasta el archiconocido algoritmo de retropropagación. Tras esto, se presenta el concepto de hiperparámetro y se realiza una revisión de los principales hiperparámetros de una red. Se continúa definiendo formalmente el problema de HPO y se aportan los métodos más populares para abordarlo en la literatura científica. Por último se implementa una red neuronal sencilla utilizando el software `R` en la que se pondrá en práctica lo visto anteriormente.

Abstract

The objective of this work is to introduce the reader to the hyperparameter optimization (HPO) problem from the basics of machine learning, with special attention to the case of feed-forward neural networks. It begins by formally explaining the functioning of artificial neurons and neural networks, focusing on the case of the multilayer perceptron. The training process of a network is then presented, ranging from simple algorithms to the well-known backpropagation algorithm. After this, the concept of hyperparameter is introduced, and a review of the main hyperparameters of a network is conducted. The HPO problem is formally defined, and the most popular methods to address it in the scientific literature are provided. Finally, a simple neural network is implemented using `R` software to put into practice what has been seen before.

Introducción

Con el desarrollo del *aprendizaje profundo* en la primera década del siglo XXI, los modelos de aprendizaje automático experimentaron un enorme aumento en complejidad y capacidad de resolución de nuevos problemas. Entre las tareas que pueden realizar estos modelos se encuentran la clasificación, regresión, transcripción, traducción, detección de errores, visión por ordenador, etc. [11, pág. 98]. La versatilidad de estos modelos supuso también un aumento de su uso como herramienta complementaria por parte de profesionales e investigadores científicos. En todos estos modelos (ya sea regresión logística, árboles de decisión, SVMs o redes neuronales) aparecen parámetros que deben ser establecidos con anterioridad, y cuya modificación afecta al rendimiento y precisión de los modelos. Sin embargo, como se expone en [21], el aumento de popularidad de los modelos de aprendizaje automático no ha venido acompañado de un tratamiento ajuste adecuado de sus parámetros. En efecto, se observó que en la mayoría de casos estos hiperparámetros se dejan en sus valores por defecto y, aun cuando existe algún tipo de ajuste, este suele ser básico o no se suele incluir explícitamente en los trabajos.

El objetivo de este trabajo es ofrecer una introducción formal al problema del ajuste de hiperparámetros de un modelo de aprendizaje automático. En las Secciones 1.1 y 1.2 se presentan los conceptos de neurona y de red neuronal, en concreto el perceptrón multicapa, como caso particular de modelo de aprendizaje automático en el que se centrará este trabajo. En la Sección 1.3 se expone el proceso de entrenamiento de una red y se incluye el algoritmo de retropropagación. Tras ello, en la Sección 1.4 se revisan los principales hiperpárametros en un MLP. Las secciones principales son las Secciones 1.5 y 1.6. En la primera, se define formalmente el problema de optimización de hiperparámetros. En la segunda, se revisan los principales métodos para resolver dicho problema. Finalmente, en el Capítulo 2 se incluye un ejemplo sencillo de regresión en el que utilizaremos lo visto anteriormente.

Capítulo 1

Generalidades

1.1. Neuronas

Una neurona artificial (en adelante, neurona) es una función matemática que modela el funcionamiento de una neurona natural. Como tal, supone el elemento básico computacional de una red neuronal. Los primeros modelos de neurona artificial fueron el de McCulloch y Pitts en 1943 o el perceptrón introducido por Rosenblatt en 1958 [8]. En esta sección presentaremos el modelo estándar de neurona introducido por Rumelhart y McClelland en 1986 [19], que generaliza los casos anteriores. Los elementos que forman una neurona son:

- Un conjunto de entradas, $\mathbf{x} = (x_1, ..., x_n) \in \mathbb{R}^n$.
- Un conjunto de pesos asociados a las entradas $\mathbf{w} = (w_1, ...w_n) \in \mathbb{R}^n$.
- Una regla de propagación $h(\mathbf{x}, \mathbf{w})$ que integra la información proveniente de las entradas en un único valor que denotaremos. La regla más utilizada, y la que usaremos a partir de aquí, consiste en realizar la suma ponderada de $x_1, ..., x_n$ con los pesos $w_1, ...w_n$ y restarle un parámetro $\theta \in \mathbb{R}$ conocido como *umbral*. Formalmente se expresa como

$$h(\mathbf{x}, \mathbf{w}) = \mathbf{w}^\top \mathbf{x} - \theta = \sum_{i=1}^{n} w_i x_i - \theta, \tag{1.1}$$

- Una función de activación ϕ que devuelve la salida y de la neurona, representado su estado de activación:

$$y = \phi(h(\mathbf{x}, \mathbf{w})) = \phi(\sum_{i=1}^{n} w_i x_i - \theta). \tag{1.2}$$

Observación 1.1. *En muchas ocasiones se obviará la existencia del umbral con el fin de simplificar los cálculos. Esto es posible puesto que podemos tomar dicho umbral como un peso más, incluyéndolo dentro del vector de pesos, y añadiendo a* $\boldsymbol{x}$ *una entrada constante* -1. *De esta manera*

$$\mathbf{w}^\top \mathbf{x} - \theta = \begin{pmatrix} \mathbf{w}^\top & \theta \end{pmatrix} \begin{pmatrix} \mathbf{x} \\ -1 \end{pmatrix}. \tag{1.3}$$

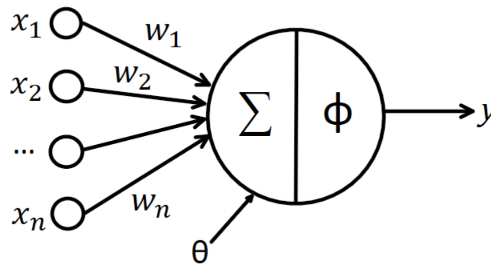

Figura 1.1: Esquema de una neurona con entradas $\mathbf{x}$, pesos $\mathbf{w}$, umbral θ y función de activación ϕ

Las funciones de activación más básicas son la identidad, la función de Heaviside (también llamada escalón unitario o función umbral) y la tangente hiperbólica. Otras funciones comunes son la sigmoide, $\frac{1}{1+\mathrm{e}^{-x}}$, y la ReLU, $\max\{0, x\}$, las cuales estudiaremos en la Subsección 1.4.

El perceptrón

Un ejemplo particular de neurona es el perceptrón introducido por el psicólogo estadounidense Frank Rosenblatt en su trabajo de 1958 llamado *The Perceptron: A probabilistic model for information storage and organization in the brain.* Esta neurona es idéntica al caso general cuyo esquema se corresponde al de la figura 1.1. Utiliza como función de activación ϕ la función de Heaviside. A partir de 1.2 obtenemos que

$$y = \begin{cases} 1 & si \quad \sum_{j=1}^{n} w_j x_j \geq \theta \\ 0 & si \quad \sum_{j=1}^{n} w_j x_j < \theta \end{cases}. \tag{1.4}$$

El perceptrón divide $\mathbb{R}^n$ en dos regiones delimitadas por el hiperplano

$$\sum_{j=1}^{n} w_j x_j = \theta, \tag{1.5}$$

de manera que funciona como un clasificador lineal que asigna el vector de entradas $\boldsymbol{x} = (x_1, ..., x_n)$ a una de las dos clases. En la Sección 1.3 abordaremos el algoritmo de aprendizaje del perceptrón.

La ADALINE

Poco después de la aparición del perceptrón, en 1960, Widrow y Hoff introducen un nuevo modelo de neurona llamado ADALINE (del inglés *ADAptative LInear NEuron*). A diferencia del perceptrón, esta neurona utiliza como función de activación la función identidad, de manera que la salida de la neurona es lineal:

$$y = \sum_{j=1}^{n} w_j x_j - \theta. \tag{1.6}$$

Este hecho, como veremos en la sección 1.3.2, resulta clave a la hora de entrenar la neurona utilizando el descenso de gradiente. De todas formas, es habitual que la salida final de esta neurona siga siendo $\{0, 1\}$, al añadir tras la salida y una función de umbral. Lo importante es ver que la magnitud lineal 1.6 se recoge para aportar información adicional a la hora de ajustar la neurona.

1.2. Redes neuronales

Una red neuronal es un modelo computacional basado en el funcionamiento del cerebro, el cual está formado por neuronas que interactúan entre si por medio de impulsos eléctricos. En textos introductorios [12] se define una red neuronal como un grafo dirigido cuyos nodos son neuronas. Esta definición resulta muy conveniente, puesto que de esta manera una red neuronal queda caracterizada por sus nodos (las neuronas) y la arquitectura de la red [7]:

Definición 1.2. Una red neuronal artificial es un grafo dirigido (V, E) con las siguientes propiedades:

1. Cada nodo i de V es una neurona con umbral θ_i y función de activación ϕ_i
2. A cada conexión $(i, j) \in E$ entre los nodos i y j se le asocia un peso $w_{ij} \in \mathbb{R}$

Comenzaremos definiendo la matriz de pesos de una red neuronal, que resulta de utilidad para distinguir distintos tipos de estructuras de red. Continuaremos viendo como en ciertos casos los nodos se organizan en estructuras llamadas capas. Finalmente daremos una clasificación en base a estos elementos.

Para las definiciones que vienen a continuación, supondremos una red neuronal con nodos numerados del 1 al N.

Definición 1.3. Se denomina *matriz de pesos* a la matriz $\mathbf{W} = (w_{ij}) \in \mathcal{M}_{N\times N}$, donde w_{ij} denota el peso asociado a la arista que conecta el nodo i con el j. Si no hay conexión del nodo i al nodo j entonces $w_{ij} = 0$.

Dentro de una red neuronal, distinguimos tres tipos de nodos con respecto a sus conexiones dentro de la red:

Definición 1.4. Un nodo k se dice *de entrada* si $w_{ik} = 0$ para todo $i \in \{1, ..., N\} \setminus \{k\}$. Esto es, los nodos de entrada no reciben información de otros nodos, sino que reciben los datos o señales del exterior.

Definición 1.5. Un nodo k se dice *de salida* si $w_{ki} = 0$ para todo $i \in \{1, ..., N\} \setminus \{k\}$. Los nodos de salida no traspasan su salida a ningún otro nodo.

Un nodo se dirá *oculto* si no es de entrada ni de salida. Los nodos ocultos no tienen contacto directo con el exterior, puesto que sólo reciben información de otros nodos, la procesan y traspasan la salida a otros nodos.

Podemos clasificar las redes neuronales atendiendo al flujo de información:

Definición 1.6. una *red unidireccional* o *alimentada hacia delante* (del inglés *feed-forward network*) es aquella que no contiene ciclos (caminos que empiezan y terminan en un mismo nodo). En caso contrario, las redes con ciclos se denominan *redes recurrentes.*

En las redes unidireccionales se pueden numerar los nodos de forma que si existe una conexión de i a j, entonces $i < j$ (sin más que comenzar numerando los nodos de entrada y continuar por los nodos conectados a estos, sucesivamente). De esta forma, los N nodos se agrupan en estructuras llamadas *capas* en las que los nodos de la cada capa solo están conectados con los de las capas adyacentes: recogen las salidas de los nodos de la capa anterior, procesan la información y la pasan a los nodos de la capa siguiente.

Definición 1.7. Se denominan capas *de entrada* (resp. *de salida*) a las capas cuyos nodos son de entrada (resp. de salida). Las capas intermedias se denominan *capas ocultas.*

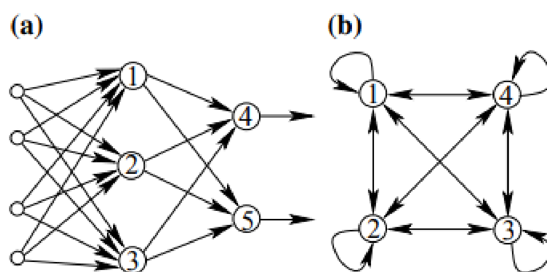

Figura 1.2: **(a)** Red unidireccional. Obsérvese su distribución por capas. Los nodos $1, 2, 3$ son nodos de entrada, mientras que 4 y 5 son nodos de salida **(b)** Red recurrente. Obsérvese que cada nodo está conectado a sí mismo. Obtenido de [7].

Los ejemplos más conocidos de redes unidireccionales son el perceptrón simple introducido por Rosenblatt (1962), el perceptrón multicapa (MLP) (utilizado por primera vez por Werboz en 1974 y popularizado por Rumelhart en 1986). Ejemplos destacables de redes recurrentes son las redes de Hopfield o la máquina de Boltzmann, que no trataremos en este trabajo.

A continuación introduciremos el funcionamiento del perceptrón simple y multicapa. Estos ejemplos serán de utilidad más adelante para explicar los principales hiperparámetros que nos encontraremos en una red neuronal en la Sección 1.4 así como para comprender el funcionamiento del algoritmo de *backpropagation* en la Subsección 1.3.3.

Perceptrón de una sola capa

El *perceptrón de una sola capa* es una red neuronal unidireccional con una capa de entrada de datos y otra de salida formada por neuronas de tipo perceptrón. De manera análoga al caso de un único preceptrón explicado en la Sección 1.1, el perceptrón simple sirve para clasificar el vector de entradas en más clases. Si tiene n nodos de entrada y m nodos de salida, los valores de salida de la red serán:

$$y_i = \phi(red_i) = \phi(\sum_{j=1}^{n} w_{ij}x_j - \theta_i), \quad i = 1, ..., m. \tag{1.7}$$

Si $\mathbf{x} = (x_1, ..., x_n)$, $\mathbf{y} = (y_1, ..., y_m)$, $\boldsymbol{\theta} = (\theta_1, ..., \theta_m)$ y $\mathbf{W} \in \mathcal{M}_{n\times m}$, entonces la formulación vectorial sería

$$\mathbf{y} = \phi(\mathbf{W}^\top \mathbf{x} - \theta). \tag{1.8}$$

Perceptrón multicapa

El *perceptrón multicapa* (MLP) es el ejemplo clásico de red unidireccional en la que existen capas intermedias con funciones de activación continuas, lo que añade complejidad al modelo. De hecho, el MLP es un aproximador universal: un MLP con una sola capa oculta es capaz de aproximar cualquier aplicación continua hasta cualquier orden, con funciones de activación sigmoidales en la capa de salida.

Notación 1.8. *En un MLP con M capas, denotaremos con*

- m_i*, el número de nodos de la* i*-ésima capa.*
- $\mathbf{x} = (x_1, ..., x_{m_1})$*, el vector de entradas de la red.*
- $\mathbf{W^{(i-1)}} \in \mathcal{M}_{m_{i-1} \times m_i}$*, los pesos de las aristas que conectan los nodos de la capa* $i-1$ *e* i*.*
- $\boldsymbol{\theta}^{\mathbf{(i)}}$ *y* $\mathbf{o^{(i)}}$*, los vectores con los umbrales y la salidas de los nodos de la capa* i*, para* $i = 1, ..., M$*.*
- $\boldsymbol{\phi}^{\mathbf{(i)}} : \mathbb{R}^{m_i} \longrightarrow \mathbb{R}^{m_i}$*, la función que aplica las funciones de activación de los nodos de la* i*-ésima capa, componente a componente. Por simplicidad, supondremos que dicha función es la misma para todos los nodos de cada capa.*
- $\mathbf{y} = (y_1, ..., y_{m_M})$*, el vector de salida de la red.*

Las salidas de los nodos de la capa i, $\mathbf{o^{(i)}} \in \mathbb{R}^{m_i}$, se calculan como sigue

$$\mathbf{o^{(i)}} = \boldsymbol{\phi}^{\mathbf{(i)}}([\mathbf{W^{(i-1)}}]^\top \mathbf{o^{(i-1)}} - \boldsymbol{\theta}^{\mathbf{(i)}}), \tag{1.9}$$

Obsérvese que los $\mathbf{o^{(i)}}$ se calculan a partir de $\mathbf{o^{(i-1)}}$ y por lo tanto el flujo de información comienza en la primera capa, donde $\mathbf{o^{(1)}} = \mathbf{x}$. Al llegar a la última capa las salidas de la red serán:

$$\mathbf{y} = \mathbf{o^{(M)}}. \tag{1.10}$$

En el caso del MLP se suele emplear la función de activación sigmoidal para todos los nodos de las $M-1$ primeras capas, lo que permite añadir no-linealidad al modelo. Sin embargo para la capa de salida será habitual utilizar otras funciones de activación dependiendo de la tarea a realizar por la red.

Podemos definir un MLP como una función $\boldsymbol{f}$ que lleva los datos de entrada y devuelve directamente la salida de la red en la 1.10:

$$\begin{aligned} \boldsymbol{f} \colon X \subset \mathbb{R}^{m_1} &\longrightarrow R^{m_M} \\ \mathbf{x} &\longmapsto \boldsymbol{f}(\mathbf{x}; \mathbf{W}) \equiv \mathbf{y} \end{aligned}$$

Esta definición será de utilidad en la Sección 1.5 en la que se define el problema de optimización de hiperparámetros.

1.3. Entrenamiento de una red neuronal

En la sección anterior vimos que las salidas de una red neuronal están estrechamente relacionadas con los pesos w_{ij} y los umbrales θ_i. Estos valores se conocen como *parámetros de la red*. Se denomina *entrenamiento* al proceso de ajuste de los parámetros de la red de manera que las salidas sean adecuadas para la realización de una tarea. Para ello es necesario un conjunto de pares de datos de entrada y salida llamados *conjunto de entrenamiento* en base a los cuales queremos que la red aprenda.

En general, dada una red neuronal con n nodos de entrada y m de salida consideramos un conjunto de *datos de entrenamiento* $\mathcal{D} = \{(\mathbf{x_1}, \mathbf{y_1}), ..., (\mathbf{x_P}, \mathbf{y_P})\} \subset \mathbb{R}^n \times \mathbb{R}^m$. Denotaremos también con $\hat{\mathbf{y}}_\mathbf{p} = f(\mathbf{x_p}; \mathbf{W}) \in \mathbb{R}^m$ al vector de salidas de la red al proporcionarle el vector de entradas $\mathbf{x_p} \in \mathbb{R}^n$.

Observación 1.9. *(Sobre la inicialización de pesos) Para poder comenzar a utilizar la neurona debemos asignar valores a los paramétros. Aunque existen varios métodos para inicializar los pesos de una red que buscan aumentar la rapidez de convergencia, e.g. [9], una opción muy utilizada es la de asignar pesos aleatoriamente. Inicializar todos los pesos a* 0 *o a algún valor constante suele ser desaconsejable puesto que genera simetría en la red, esto es, que todos los nodos de cada capa se comporten igual. En [11, pág. 296] se trata de manera pormenorizada este tema.*

Vamos a estudiar como es el proceso de aprendizaje de los modelos introducidos en la sección anterior.

1.3.1. Algoritmo del perceptrón

En esta sección se describe el llamado *algoritmo de aprendizaje del perceptrón* (en inglés, *Perceptron Learning Algorithm*) para el entrenamiento de una neurona perceptrón. Se trata de un algoritmo iterativo en el que en cada paso del proceso se actualiza el vector de pesos en función de los datos de entrenamiento clasificados erróneamente. El proceso termina cuando el perceptrón es capaz de clasificar correctamente todos los pares de entrenamiento.

Queremos encontrar un algoritmo que, a partir de P pares de datos de entrenamiento $\mathcal{D} \subset \mathbb{R} \times \mathbb{R}^m$, devuelva un conjunto de pesos tales que el perceptrón, con dichos parámetros, clasifique correctamente todos los datos de entrenamiento, es decir:

$$\hat{y}_p = y_p, \quad p = 1, ..., P. \tag{1.11}$$

Dado un vector un vector de entrada $\mathbf{x_p}$ de $\mathcal{D}$, la salida de la red será

$$\hat{y}_p = \begin{cases} 1, & \mathbf{w}^\intercal \mathbf{x_p} \geq 0 \\ -1, & \mathbf{w}^\intercal \mathbf{x_p} < 0 \end{cases} . \tag{1.12}$$

Observación 1.10. *Siguiendo la Observación 1.1, podemos utilizar una expresión abreviada para la ecuación anterior:*

$$\hat{y}_p = sgn(w^\top x_p), \tag{1.13}$$

donde la función $sgn(\cdot)$ es la función signo. Esta forma será de utilidad para expresar cuándo un ejemplo está bien o mal clasificado:

$$\begin{aligned} y_p = \hat{y}_p &\implies y_p \mathbf{w}^\top \mathbf{x_p} > 0 \\ y_p \neq \hat{y}_p &\implies y_p \mathbf{w}^\top \mathbf{x_p} < 0. \end{aligned} \tag{1.14}$$

El *error* del ejemplo p-ésimo, que tomará los valores $\{-2, 0, 2\}$, será

$$e_p = y_p - \hat{y}_p. \tag{1.15}$$

El paso clave del algoritmo consistirá en actualizar el nuevo vector de pesos $\mathbf{w}'$ como sigue:

$$\mathbf{w}' = \mathbf{w} + \eta y_p \mathbf{x_p}, \tag{1.16}$$

o equivalentemente

$$\Delta \mathbf{w} = \eta y_p \mathbf{x_p}, \tag{1.17}$$

donde $\eta > 0$ es un parámetro llamado *paso*. Observamos que existen tres posibilidades:

$$
\begin{aligned}
&y_p = \hat{y}_p \implies e_p = 0 \implies \mathbf{w}' = \mathbf{w}, \\
&y_p = 1 \quad e \quad \hat{y}_p = -1 \implies e_p = 2 \implies \mathbf{w}' = \mathbf{w} + \eta \mathbf{x_p}, \\
&y_p = -1 \quad e \quad \hat{y}_p = 1 \implies e_p = -2 \implies \mathbf{w}' = \mathbf{w} - \eta \mathbf{x_p}.
\end{aligned}
\tag{1.18}
$$

El Algoritmo 1 solo actualiza los pesos cuando $\mathbf{x_p}$ está mal clasificado. Si $\mathbf{x_p}$ sí que está bien etiquetado, $y_p = \hat{y}_p$, se pasa al siguiente par de entrenamiento $(\mathbf{x_{p+1}}, y_{p+1})$. En cada iteración del algoritmo se recorren todos los pares de $\mathcal{D}$, uno por uno, hasta que no queda ninguno mal clasificado.

Algoritmo 1 Algoritmo de aprendizaje del perceptrón

Entrada: $\{\boldsymbol{x_1}, ..., \boldsymbol{x_P}\} \subset \mathbb{R}^n, \quad \{y_1, ..., y_P\} \subset \{0, 1\}, \quad \eta > 0, \quad itmax \in \mathbb{N}$
Salida: $\boldsymbol{w}$

```
w ← 0                                          ▷ Inicialización de los pesos
num_errores ← 0        ▷ Número de errores de clasificación en una iteración
for t = 1, ..., itmax do
    for p = 1, ..., P do
        ŷ = signo(w^t x_p)                          ▷ Salida del perceptrón
        if ŷ ≠ y_p then
            w ← w + η y_p x_p               ▷ Actualización vector de pesos
            num_errores ← num_errores + 1              ▷ Se cuenta el error
        end if
    end for
    if num_errores = 0 then
        stop                                      ▷ Condición de parada
    end if
    num_errores ← 0
end for
```

Cabe preguntarse bajo qué condiciones converge. Para responder a esta pregunta introducimos la definición de conjuntos linealmente separables.

Definición 1.11. Dos conjuntos de puntos X e Y de $\mathbb{R}^n$ se dicen *linealmente separables* si existe un hiperplano que los separa, es decir, existen $a_1, ...a_n, k \in \mathbb{R}$ tal que $\sum_{i=1}^{n} a_i x_i \geq k$ para todo $\mathbf{x} \in X$ y $\sum_{i=1}^{n} a_i y_i < k$, para todo $\mathbf{y} \in Y$.

El siguiente teorema, probado por Rosenblatt en 1962, da una condición necesaria y suficiente para la convergencia del algoritmo. La demostración está basada en las notas [22, 6]:

Teorema 1.12. *(Convergencia del algoritmo del perceptrón) Dado un perceptrón de una única neurona con el vector de entrada* $\mathbf{x} = (x_1, ..., x_n)$ *perteneciente, aleatoriamente, a dos clases de* $\mathbb{R}^n$ *linealmente separables. Entonces, dado un conjunto de entrenamiento* $\mathcal{D}$ *finito, el algoritmo de aprendizaje del perceptrón converge a un hiperplano separando las dos clases en tiempo finito.*

Demostración. Puesto que las dos clases son linealmente separables, existe un vector $\mathbf{w}^*$ unitario ($\|\mathbf{w}^*\| = 1$) que separa correctamente todos los datos de entrenamiento, esto es:

$$y_p \mathbf{w}^{*\top} \mathbf{x_p} > 0, \quad p = 1, ..., P. \tag{1.19}$$

Denotamos con γ el mínimo de los valores anteriores tal que

$$\gamma = \min_{p=1,...,P} y_p \mathbf{w}^{*\top} \mathbf{x_p} > 0. \tag{1.20}$$

Vamos a suponer que el algoritmo no converge y llegaremos a una contradicción. Sin pérdida de generalidad, contaremos solo los pasos en los que se comete un error y el vector de pesos se actualiza. En el k-ésimo paso del algoritmo donde se comete un error, contaremos con un vector de pesos $\mathbf{w_{k-1}}$ y un nuevo par $(x_p, y_p) \in \mathcal{D}$. Sea $\theta(k)$ el ángulo que forman $\mathbf{w}^*$ y $\mathbf{w_k}$,

$$\cos\theta(k) = \frac{\mathbf{w_k}^\top \mathbf{w}^*}{\|\mathbf{w_k}\| \|\mathbf{w}^*\|}. \tag{1.21}$$

A partir del numerador de 1.21 y utilizando 1.16:

$$\begin{aligned} \mathbf{w_k}^\top \mathbf{w}^* &= (\mathbf{w_{k-1}}^\top + \eta y_p \mathbf{x_p}^\top)\mathbf{w}^* \\ &= \mathbf{w_{k-1}}^\top \mathbf{w}^* + \eta y_p \mathbf{w}^{*\top} \mathbf{x_p}. \end{aligned} \tag{1.22}$$

Por como hemos definido γ en 1.20:

$$\mathbf{w_k}^\top \mathbf{w}^* \geq \mathbf{w_{k-1}}^\top \mathbf{w}^* + \eta\gamma. \tag{1.23}$$

Aplicamos esta desigualdad k veces hasta llegar a los pesos iniciales $\mathbf{w_0} \equiv \mathbf{0}$:

$$\mathbf{w_k}^\top \mathbf{w}^* \geq \mathbf{w_0}^\top \mathbf{w}^* + k\eta\gamma = k\eta\gamma. \tag{1.24}$$

Por otra parte, en el denominador de 1.21:

$$\begin{aligned} \|\mathbf{w_k}\|^2 &= \|\mathbf{w_{k-1}} + \Delta\mathbf{w_k}\|^2 \\ &= \|\mathbf{w_{k-1}}\|^2 + \eta^2\|\mathbf{x_p}\|^2 + 2(\mathbf{w_{k-1}^\top}\Delta\mathbf{w_k}) \\ &\leq \|\mathbf{w_{k-1}}\|^2 + \eta^2 D^2, \end{aligned} \tag{1.25}$$

donde $D = \text{máx}_p \|\mathbf{x_p}\|$.

Como antes, aplicando la desigualdad hasta llegar a los pesos iniciales,

$$\|\mathbf{w_k}\|^2 \leq \|\mathbf{w_0}\|^2 + k\eta^2 D^2 = k\eta^2 D^2. \tag{1.26}$$

Finalmente, a partir de las desigualdades 1.24 y 1.26 llegamos a que

$$\cos\theta(k) \geq \frac{\eta k \gamma}{\eta\sqrt{k}D} = \frac{\gamma}{D}\sqrt{k}. \tag{1.27}$$

Hemos llegado a una contradicción: Como hemos supuesto que el algoritmo no converge, entonces $k \longrightarrow \infty$ y por tanto $\cos\theta(n) \longrightarrow \infty$, lo cual no tiene sentido por ser el coseno una función acotada. □

La demostración del teorema también proporciona una cota superior para el número de errores cometidos (equivalentemente, el número de actualizaciones de los pesos) durante el algoritmo:

Corolario 1.13. *En las condiciones del teorema anterior, el algoritmo del perceptrón comete a lo sumo* $\frac{R^2}{\gamma^2}$ *errores.*

Demostración. A partir de la desigualdad 1.24 obtenemos:

$$\|\mathbf{w_k}\|\|\mathbf{w}^*\| \geq \mathbf{w_k^\top}\mathbf{w}^* \geq k\eta\gamma, \tag{1.28}$$

de donde, puesto que $\mathbf{w}^*$ es unitario:

$$\|\mathbf{w_k}\| \geq k\eta\gamma. \tag{1.29}$$

Combinando esta desigualdad con 1.26 llegamos a que

$$k^2\eta^2\gamma^2 \leq \|\mathbf{w_k}\|^2 \leq k\eta^2 D^2 \tag{1.30}$$

y por lo tanto

$$k \leq \frac{D^2}{\gamma^2}. \tag{1.31}$$

□

1.3.2. Regla delta

En 1960 Widrow y Hoff introducen el algoritmo LMS, también llamado regla delta, para el entrenamiento de la ADALINE. El hecho de que la salida de esta neurona fuese lineal y no binaria como en el caso del perceptrón permitió utilizar por primera vez el descenso de gradiente para el aprendizaje de una neurona.

Supongamos entonces que tenemos una única neurona de tipo ADALINE y un conjunto de datos de entrenamiento $\mathcal{D}$ de tamaño P. Definiremos una *función de error* encargada de cuantificar la discrepancia entre las salida real de la neurona, $\hat{y}_p \in \mathbb{R}$, y la etiqueta deseada $y_p \in \{0, 1\}$:

Definición 1.14. La función de error $E \colon \mathbb{R}^{n+1} \longrightarrow \mathbb{R}$ es

$$E(\mathbf{w}) = \frac{1}{2} \sum_{p=1}^{P} (y_p - \hat{y}_p)^2.$$

Puesto que $\hat{y}_p = \mathbf{w}^\top \mathbf{x_p}$, la función E es continua y diferenciable por lo que será posible calcular su gradiente ∇E. Utilizando la regla de la cadena,

$$(\nabla E)_i = \frac{\partial E}{\partial w_i} = \frac{\partial E}{\partial \hat{y}_p} \frac{\partial \hat{y}_p}{\partial w_i} \tag{1.32}$$

$$= \sum_{p=1}^{P} (y_p - \hat{y}_p)(-x_p)_i), \tag{1.33}$$

para $i = 1, ..., n+1$.

Puesto que el gradiente es la dirección de máximo crecimiento, si modificamos el peso w_i en la dirección contraria a $\frac{\partial E}{\partial w_i}$ conseguiremos mejorar el error localmente. De esta forma:

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i} = \eta \sum_{p=1}^{P} (y_p - \hat{y}_p)(x_p)_i \tag{1.34}$$

donde $\eta > 0$ es el paso.

Observamos que, a diferencia del algoritmo del perceptrón las actualizaciones de los pesos son lineales respecto a los errores cometidos, de manera que cuanto mayor sea el error mayor será el cambio. Análogamente, cuando los errores sean pequeños, las actualizaciones de los pesos también lo serán. De esta manera los pesos se acercarán asintóticamente a una solución.

1.3.3. Retropropagación

Una de las piezas clave en el desarrollo del aprendizaje automático es el algoritmo de retropropagación propuesto por Rumelhart et al. en 1986 [20] para el entrenamiento del MLP con funciones de activación de tipo sigmoide. A día de hoy supone la base de los algoritmos de aprendizaje más utilizados, también para redes recurrentes. Es importante remarcar que la retropropagación no supone un algoritmo de entrenamiento como tal, sino que simplemente calcula los gradientes necesarios utilizando la regla de la cadena del cálculo. La retropropagación es una generalización de la regla delta introducida en la sección anterior, de tal manera que también se le conoce originalmente como *regla delta generalizada*.

Como veremos, cada paso del algoritmo de retropropagación esta dividido en dos etapas:

- La primera, el paso *hacia delante*, consiste en computar la salida de la red para un vector de entrada y compararla con la salida deseada para dichos datos de entrada, de manera que se calcula el error cometido por la red.
- La segunda etapa, el paso *hacia atrás*, consiste en calcular las contribuciones de cada nodo de la última capa al error de la red. En base a estas contribuciones, se actualizan los pesos entre la última capa y la anterior utilizando descenso de gradiente. Después los errores se propagan a la capa anterior y se actualizan los pesos entre esta capa y la capa adyacente. El proceso continúa hacia atrás hasta actualizar todos los pesos.

Dado un MLP con M capas y un conjunto de entrenamiento de tamaño P, $\mathcal{D} \subset \mathbb{R}^{m_1} \times \mathbb{R}^{m_M}$. La función objetivo a minimizar es la *función de error*, que es el error cuadrático medio del vector de salidas de la red

$$E = \frac{1}{P}\sum_{p=1}^{P} E_p = \frac{1}{2P}\sum_{p=1}^{P} \|\hat{\mathbf{y}}_\mathbf{p} - \mathbf{y}_\mathbf{p}\|^2, \tag{1.35}$$

donde, para cada $p = 1, ..., P$,

$$E_p = \frac{1}{2}\|\hat{\mathbf{y}}_\mathbf{p} - \mathbf{y}_\mathbf{p}\|^2. \tag{1.36}$$

Recordemos que $\hat{\mathbf{y}}_\mathbf{p} \in \mathbb{R}^{m_M}$ es el vector con las salidas del MLP para los datos de entrada $\mathbf{x}_\mathbf{p} \in \mathbb{R}^{m_1}$, que depende de los pesos de la red, por lo que las función de error E y E_p dependen de la matriz de pesos $\mathbf{W}$.

Al finalizar el primer paso del algoritmo con el ejemplo $(\mathbf{x}_\mathbf{p}, \mathbf{y}_\mathbf{p}) \in \mathcal{D}$ y calcular E_p, la regla para actualizar la matriz de pesos $\mathbf{W}$ será, utilizando descenso de gradiente,

$$\Delta_p \mathbf{W} = -\eta \frac{\partial E_p}{\partial \mathbf{W}}. \tag{1.37}$$

A continuación veremos como podemos aplicar esta regla utilizando la retropropagación de errores que se expone a continuación. Siguiendo la notación del Apartado 1.2, denotamos con $\mathbf{r}_\mathbf{p}^{(\mathbf{i})} = [\mathbf{W}^{(\mathbf{i-1})}]^\top \mathbf{o}_\mathbf{p}^{(\mathbf{i-1})}$ el vector de salidas de las neuronas $v = 1, ..., m_i$ de la capa i antes de aplicar la función de activación y cuando la entrada de la red es el vector $\mathbf{x}_\mathbf{p}$. Denotamos con $\mathbf{o}_\mathbf{p}^{(\mathbf{i})} = \boldsymbol{\phi}^{(\mathbf{i})}(\mathbf{r}_\mathbf{p}^{(\mathbf{i})})$ el vector de salidas finales de las neuronas $v = 1, ..., m_i$ de la capa i cuando la entrada de la red es el vector $\mathbf{x}_\mathbf{p}$.

Utilizando la regla de la cadena, la derivada en 1.37 es

$$\frac{\partial E_p}{\partial w_{uv}^{(i)}} = \frac{\partial E_p}{\partial r_{p,v}^{(i+1)}} \frac{\partial r_{p,v}^{(i+1)}}{\partial w_{uv}^{(i)}}. \tag{1.38}$$

Vamos a desarrollar las dos partes del segundo término. Por una parte

$$\frac{\partial r_{p,v}^{(i+1)}}{\partial w_{uv}^{(i)}} = \frac{\partial}{\partial w_{uv}^{(i)}} \left(\sum_{k=1}^{m_i} w_{kv}^{(i)} o_{p,k}^{(i)} + \theta_v^{(i+1)} \right) = o_{p,u}^{(i)}. \tag{1.39}$$

Teniendo en cuenta 1.9 y utilizando la regla de la cadena:

$$\frac{\partial E_p}{\partial r_{p,v}^{(i+1)}} = \frac{\partial E_p}{\partial o_{p,v}^{(i+1)}} \frac{\partial o_{p,v}^{(i+1)}}{\partial r_{p,v}^{(i+1)}} = \frac{\partial E_p}{\partial o_{p,v}^{(i+1)}} \dot{\phi}^{(i+1)}(r_{p,v}^{(i+1)}). \tag{1.40}$$

donde $\dot{\phi}^{(i+1)}$ es la derivada de la función de activación de los nodos de la capa $i+1$ y la derivada parcial, dependiendo de la capa donde nos encontremos, es

$$\begin{gathered} \frac{\partial E_p}{\partial o_{p,v}^{(M)}} = e_{p,v} \\ \frac{\partial E_p}{\partial o_{p,v}^{(i+1)}} = \sum_{k=1}^{m_{i+2}} \frac{\partial E_p}{\partial r_{p,k}^{(i+2)}} w_{v,k}^{(i+1)}, \quad i = 1, ..., M-2. \end{gathered} \tag{1.41}$$

Definición 1.15. Se denomina *delta* del nodo v de la i-ésima capa al siguiente valor

$$\delta_{p,v}^{(i)} = -\frac{\partial E_p}{\partial r_{p,v}^{(i)}}, \tag{1.42}$$

para $i = 2, ..., M$.

Finalmente, sustituyendo las ecuaciones 1.41 en 1.40 obtenemos

$$\begin{aligned} \delta_{p,v}^{(M)} &= -e_{p,v}\dot{\phi}^{(M)}(r_{p,v}^{(M)}) \\ \delta_{p,v}^{(i+1)} &= -\dot{\phi}^{(i+1)}(r_{p,v}^{(i+1)})\sum_{k=1}^{m_{i+2}} \delta_{p,k}^{(i+2)} w_{v,k}^{(i+1)}, \quad i = 1, ..., M-2. \end{aligned} \tag{1.43}$$

A partir de estas ecuaciones podemos calcular todos los $\delta_{p,v}^{(i+1)}$, sin más que comenzar por la última capa, la de salida, e ir hacia atrás. A medida que se calculan los deltas de la capa $i+1$ podemos actualizar los pesos entre la capa i e $i+1$ como sigue:

$$\Delta_p w_{uv}^{(i)} = -\eta \frac{\partial E_p}{\partial w_{uv}^{(i)}} = \eta \delta_{p,v}^{(i+1)} o_{p,u}^{(i)} \tag{1.44}$$

Observación 1.16. *(Problema de desvanecimiento de gradientes) Durante el proceso de retropropagación, los pesos se actualizan con los deltas calculados a partir de 1.43. Para el cálculo de los deltas de cada capa se multiplican los deltas de la capa adyacente, además de las derivadas de las funciones de activación. Si estos valores son menores que* 1 *entonces los deltas tenderán a cero a medida se retrocede por las capas, causando que los pesos de las primeras capas se actualicen muy lentamente. Este fenómeno dificulta el proceso de entrenamiento, sobre todo a medida que aumentan el número de capas ocultas y el ancho de las capas de una red.*

En la práctica, para el entrenamiento de un MLP se utilizan algoritmos llamados *optimizadores*. En la Subsección 1.4 se introducen los principales.

1.4. Hiperparámetros de una red neuronal

En la sección anterior hemos estudiado distintos algoritmos utilizados para el entrenamiento de una red y hemos visto que los parámetros entrenables (en adelante, simplemente *parámetros*) en este proceso son solamente los pesos y umbrales. Sin embargo, a lo largo de las secciones anteriores nos hemos encontrado con otros parámetros asociados a la red como el número de capas en la Subsección 1.2 o el paso η en la Subsección 1.3.3 que también afectan al desempeño del modelo.

Los *hiperparámetros* son todos aquellos parámetros relacionados con una red neuronal que no se pueden actualizar durante el entrenamiento. Podemos clasificarlos grosso modo en aquellos relacionados con la arquitectura de la red (número de capas ocultas, ancho de capa, funciones de activación) y los relacionados con el entrenamiento de la red (tasa de aprendizaje, optimizador, momento, etc.).

A continuación presentaremos los principales hiperparámetros en una red neuronal.

Número de capas ocultas y ancho de capa

El *número de capas ocultas* y el *ancho de cada capa* (el número de neuronas en una cierta capa) son hiperparámetros importantes puesto que afectan positivamente a la complejidad del modelo al mismo tiempo que influyen negativamente en el tiempo de entrenamiento por medio de la retropropagación: Un número grande de estos hiperparámetros puede hacer al modelo imposible de entrenar en un tiempo razonable y causar sobreajuste, mientras que un número escaso puede hacer que el modelo sea impreciso (subajuste). En general, las redes con más capas son capaces de lograr mejores resultados [11, pág. 198], pero se debe tener en cuenta los medios con los que contamos para entrenar la red. Hasta hace poco más de una década el entrenamiento por medio de redes con más de 2 capas ocultas se consideraba prácticamente inviable [1].

Por norma general, será recomendable comenzar con un ancho de capa igual para todas las capas ocultas. Además, una primera capa oculta mayor que la capa de entrada suele dar mejores resultados [2, pág. 11].

Función de activación

La *función de activación* ϕ de cada neurona se puede interpretar también como un hiperparámetro de tipo categórico. Supone un elemento importante en el funcionamiento de cada neurona y en la trascendencia de las redes neuronales en general: Si las neuronas careciesen de esta función, como la composición de aplicaciones lineales es lineal, la salida de cualquier red neuronal se podría obtener como una suma ponderada de las entradas, perdiendo la complejidad que las hace tan versátiles.

- Tanto la *función sigmoide*

$$\begin{aligned} \mathbb{R} &\longrightarrow (0,1) \\ x &\longmapsto \frac{1}{1+e^{-x}} \end{aligned}$$

como la tangente hiperbólica

$$\begin{aligned} \mathbb{R} &\longrightarrow (-1,1) \\ x &\longmapsto \tanh x \equiv \frac{e^x - e^{-x}}{e^x + e^{-x}} \end{aligned}$$

son dos de las funciones usadas clásicamente como funciones de activación. La función sigmoide se utilizó en los primeros trabajos sobre retropropagación por ser continua y diferenciable con rango acotado $(0,1)$. La tangente hiperbólica comparte estas propiedades y mejora el tiempo de entrenamiento, debido a su mayor rango y su mayor pendiente, como se puede observar en la Figura 1.3.

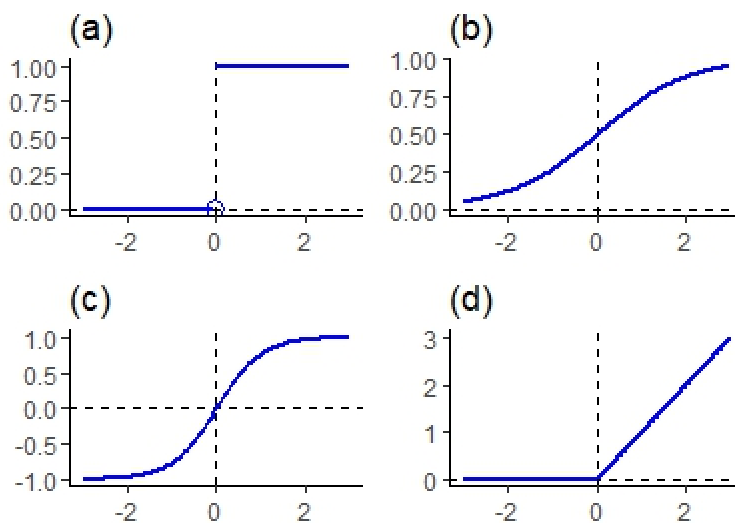

Figura 1.3: Comparación de distintas funciones de activación: **(a)** Función de Heaviside. **(b)** Función sigmoidal. **(c)** Tangente hiperbólica. **(d)** ReLU

Aunque todavía están entre las funciones de activación más utilizadas, no son eficaces para redes con un número de capas o un ancho de capa elevado puesto que causa el desvanecimiento del gradiente expuesto en 1.16: En efecto, la derivada de la función sigmoide es $e^x(1+e^x)^{-2}$, que toma valores en el intervalo $(0, \frac{1}{4})$ y la derivada de $\tanh x$ es $(\cosh x)^{-2}$, con rango $(0,1]$.

- La función ReLU $\max\{0,x\}$ soluciona dicho problema de desvanecimiento puesto que la derivada toma los valores 0 o 1. De esta manera, es preferible sobre las dos anteriores para redes *profundas* [10]. Esta función cuenta con multitud de variantes: LReLU, PReLU, NReLU, etc.

- En [18] se realiza una comparativa entre las funciones de activación más utilizadas, obteniendo que la función propuesta, llamada *Swish*,

$$x \longrightarrow \frac{x}{1+e^{-\beta x}}, \quad \beta \geq 0, \tag{1.45}$$

mejora la precisión de las redes. Obsérvese que para $\beta = 0$ la función es lineal, mientras que para $\beta \longrightarrow \infty$ se asemeja a la función ReLU.

En conclusión, la función ReLU sigue siendo la función más utilizada para las capas ocultas, debido a su mayor eficacia respecto a las funciones de activación sigmoide o tangente hiperbólica [10] y su mayor sencillez respecto de otras como la Swish.

Respecto a la elección de la función en la capa final, la de salida, se debe tener en cuenta el tipo de tarea a realizar por la red: Para los problemas de clasificación se buscarán funciones que tengan rangos pequeños como $(0, 1)$. Se suelen utilizar funciones del tipo sigmoide para clasificación en dos grupos, o su generalización, la función *softmax*

$$\begin{aligned} \mathbb{R}^K &\longrightarrow (0,1)^K \\ \mathbf{z} &\longmapsto \left(\frac{e^{z_1}}{\sum_{i=1}^K e^{z_i}}, \ldots, \frac{e^{z_K}}{\sum_{i=1}^K e^{z_i}} \right), \end{aligned}$$

para la clasificación en más grupos no linealmente separables [16].

Por otra parte, si la tarea es de regresión o aproximación de una función, será conveniente buscar funciones que tengan un rango más amplio. Una opción habitual en este caso es la función identidad, haciendo que la salida de la red sea lineal.

Optimizador

Los *optimizadores* son los algoritmos utilizados para el entrenamiento de las redes neuronales. Estos algoritmos se basan en el descenso de gradiente para actualizar los pesos y en la retropropagación para calcular dichos gradientes, tal como vimos en la Subsección 1.3.3. Sin embargo, este es un proceso lento y costoso computacionalmente, por lo que los optimizadores incluyen distintas variaciones para acelerar el entrenamiento.

A continuación se presentan los optimizadores más utilizados:

- El *descenso de gradiente estocástico* (SGD) reemplaza el cálculo del gradiente en 1.37 por una estimación del mismo utilizando un subconjunto de los pares de entrenamiento. En

cada paso del algoritmo, se escoge aleatoriamente un subconjunto $\mathcal{D}' \subset \mathcal{D}$ de tamaño P', $P' < P$, conocido comúnmente como *minibatch*, a partir del cual se calcula el gradiente estimado como sigue:

$$\mathbf{g}(t) = \frac{1}{P'}\frac{\partial}{\partial \mathbf{W}}\left(\sum_{(\mathbf{x},\mathbf{y})\in\mathcal{D}'} E(\mathbf{x},\mathbf{y},\mathbf{W}(t))\right). \tag{1.46}$$

Tras esto, los pesos se actualizan de la misma forma que antes,

$$\mathbf{W}(t+1) = \mathbf{W}(t) - \eta\mathbf{g}(t). \tag{1.47}$$

Este método tiene la gran ventaja respecto del descenso de gradiente utilizado la Sección 1.3.3 de que el tiempo de computación de cada paso no aumenta con el tamaño del conjunto de entrenamiento. Esto se debe a la segmentación de los datos por lotes (los *minibatch*), en vez de trabajar en cada paso con la totalidad del conjunto de entrenamiento.

El número P' se conoce como *batch size* y es otro hiperparámetro que se suele tomar constante e igual a alguna potencia de 2 como 32, 64, 128, etc. por cuestiones de eficiencia computacional.

- El *método del momento* introduce una nueva variable intermedia v que captura información sobre los gradientes estimados anteriores a través de su media móvil exponencial $\mathbf{v}(\mathbf{t})$. Los pesos se actualizan de la siguiente manera:

$$\mathbf{v}(t+1) = \beta\mathbf{v}(t) + (1-\beta)\mathbf{g}(t), \tag{1.48}$$

$$\mathbf{W}(t+1) = \mathbf{W}(t) - \eta\mathbf{v}(t+1). \tag{1.49}$$

Con este método aparece un nuevo hiperparámetro $\alpha \in (0,1)$ llamado *momento*. Cuanto más cercano a 1 sea α, más afectará la variable $\mathbf{v}$ a la actualización de los pesos. El momento suele tomar valores elevados como 0,5, 0,9 o 0,99.

- El método *Adagrad* busca adaptar el paso η de manera diferente para cada peso, reduciendo el tamaño del paso para aquellos parámetros donde el gradiente es mayor. Para ello, se utiliza la matriz $\mathbf{G}(t)$, que acumula la información de los gradientes hasta la iteración t-ésima:

$$\mathbf{G}(t) = \sum_{\tau=1}^{t}\mathbf{g}(\tau)\mathbf{g}(\tau)^{\intercal}. \tag{1.50}$$

donde los elementos de la diagonal de la matriz son las sumas de cada componente de $\eth(t)$ en las iteraciones $t = 1, ..., \tau$ al cuadrado:

$$(\mathbf{G}(t))_{ii} = \sum_{\tau=1}^{t} (\mathbf{g}(\tau))_i^2. \tag{1.51}$$

De esta forma, añadiendo un ϵ pequeño (normalmente, 10^{-6}) para la estabilidad numérica, la actualización de los pesos es

$$\mathbf{W}(t+1) = \mathbf{W}(t) - \frac{\eta}{\sqrt{(\mathbf{G}(t))_{ii} + \epsilon}} \mathbf{g}(t). \tag{1.52}$$

En la práctica, este método pierde eficacia puesto que al acumularse todos los gradientes desde el inicio del entrenamiento, el paso disminuye demasiado pronto.

- El método *RMSprop* es una variación del Adagrad introducida por Hinton en 2012 que trata de solucionar este problema de acumulación de gradientes. El RMSprop sustituye la matriz $\mathbf{G}(t)$ por una media móvil $\mathbf{M}(t)$ de manera que los gradientes más antiguos dejen de tener efecto sobre el tamaño del paso. La matriz $\mathbf{M}(t)$ se define como

$$\mathbf{M}(t+1) = \gamma \mathbf{M}(t) + (1-\gamma)\mathbf{g}(t)^2, \tag{1.53}$$

donde $\gamma \in (0, 1)$ es un hiperparámetro que ajusta el decaimiento de los gradientes.

- El método *Adam* [14] supone una combinación del método del momento y el RMSprop, resultando en un modelo con dos hiperparámetros $\beta_1, \beta_2 \in [0, 1)$ que sirven de factores de decaimiento para las medias móviles del gradiente y del gradiente al cuadrado, respectivamente. Además, el algoritmo incluye un paso de corrección de los sesgos de estas medias móviles. Se comienza calculando las dos medias móviles

$$\mathbf{v}(t+1) = \beta_1 \mathbf{v}(t) + (1-\beta_1)\mathbf{g}(t), \tag{1.54}$$

$$\mathbf{M}(t+1) = \beta_2 \mathbf{M}(t) + (1-\beta_2)\mathbf{g}(t)^2. \tag{1.55}$$

Después se lleva a cabo la corrección del sesgo

$$\hat{\mathbf{v}}(t+1) = \frac{\mathbf{v}(t+1)}{1-\beta_1^t}, \tag{1.56}$$

$$\hat{\mathbf{M}}(t+1) = \frac{\mathbf{M}(t+1)}{1-\beta_2^t}, \tag{1.57}$$

donde β_1^t y β_2^t son dichos coeficientes elevados a t. Finalmente se actualizan los pesos de manera similar a los casos anteriores:

$$\mathbf{W}(t+1) = \mathbf{W}(t) - \eta \frac{\hat{\mathbf{v}}(t+1)}{\sqrt{\hat{\mathbf{M}}(t+1)} + \epsilon}. \tag{1.58}$$

El método Adam se ha convertido a día de hoy en el optimizador por defecto para redes neuronales profundas debido a su buen desempeño con respecto a otros optimizadores anteriores como Adagrad o RMSprop. Además, el aumento del número de hiperparámetros que supone este método no implica un aumento en la memoria necesaria y su ajuste suele ser sencillo. Los valores por defecto que se dan en el artículo original para β_1^t, β_2^t y ϵ son 0,9, 0,999 y 10^{-8}.

Tasa de aprendizaje

El principal hiperparámetro relacionado con el algoritmo de la retropropagación es el *paso* $\eta > 0$ del descenso de gradiente, conocido en el ámbito del aprendizaje automático como *tasa de aprendizaje* (en inglés, *learning rate*). Se trata de un hiperparámetro crítico para la eficiencia del entrenamiento, por lo que debe tener prioridad a la hora de ajustar los hiperparámetros de una red [2, pág. 8]. Mientras que un paso demasiado pequeño puede hacer que el entrenamiento sea lento, un paso demasiado grande provocará que el que el algoritmo de aprendizaje no converja, por lo que resulta importante ajustar de alguna manera este hiperparámetro. Típicamente, el paso toma valores menores que 1 y mayores que 10^{-6} cuando las entradas de la red están normalizadas [2, pág. 8].

Cuando se utilizan optimizadores como el descenso de gradiente estocástico o derivados en los que el gradiente es una estimación y por lo tanto existe un error en cada actualización, será necesario tomar un paso variable $\eta(t) > 0$ para cada paso t del algoritmo. Como se demuestra en [4, pág. 18-21], una condición suficiente para la convergencia del algoritmo SGD es que $\eta(t)$ cumpla

$$\sum_{t=1}^{\infty} \eta(t) = \infty \quad \text{y} \tag{1.59}$$

$$\sum_{t=1}^{\infty} \eta(t)^2 < \infty. \tag{1.60}$$

Siguiendo esta idea, en [11, pág. 291] se propone un paso con decaimiento lineal entre un paso inicial η_0 y uno final η_τ, tales que $\eta_0 > \eta_\tau > 0$, a lo largo de las τ primeras iteraciones:

$$\eta(t) = (1 - \frac{t}{\tau})\eta_0 + \frac{t}{\tau}\eta_\tau, \quad t = 0, ..., \tau, \tag{1.61}$$

$$\eta(t) = \eta_\tau, \quad t > \tau. \tag{1.62}$$

En [2, pág. 9] se presenta otro tipo de paso variable que comienza a decaer tras la τ-ésima iteración:

$$\eta(t) = \frac{\eta_0 \tau}{\text{máx}\{t, \tau\}}. \tag{1.63}$$

1.5. Optimización de hiperparámetros

Finalmente llegamos a la sección que da nombre al trabajo. La *optimización de hiperparámetros* (HPO) se refiere al ajuste de los parámetros que no se pueden modificar durante el entrenamiento puesto que, como hemos visto, el propio diseño de la red o el optimizador dependen de ellos. De esta forma, el ajuste de los hiperparámetros se lleva a cabo antes del entrenamiento de la red.

Notación 1.17. *Para una red neuronal $\boldsymbol{f}$ con N hiperparámetros, denotamos el dominio del n-ésimo hiperparámetro por Λ_n y el espacio de configuración de los hiperparámetros por $\Lambda = \Lambda_1 \times \Lambda_2 \times ... \times \Lambda_N$. Los vectores con cada combinación de hiperparámetros los denotaremos por $\boldsymbol{\lambda} \in \Lambda$. La red neuronal f con la elección de hiperparámetros λ será $\boldsymbol{f_\lambda}$.*

El problema de HPO se puede expresar formalmente como sigue:

$$\boldsymbol{\lambda^*} = \arg\min_{\boldsymbol{\lambda} \in \Lambda} \quad \mathbb{E}_{\mathbf{x}} \left[\mathcal{L}(\boldsymbol{f_\lambda}, \mathcal{D}_E, \mathcal{D}_V) \right], \tag{1.64}$$

donde $\mathcal{L}$ es la *función de pérdida* [1]. Esta función cuantifica la precisión de la red f_λ, entrenada con el conjunto de entrenamiento $\mathcal{D}_E$, calculando el error respecto a otro conjunto de validación $\mathcal{D}_V$.

Puesto que el conjunto de datos disponibles es finito, la esperanza en 1.64 se estimará con la media muestral utilizando *validación cruzada*:

$$\lambda^* \approx \arg\min_{\lambda \in \Lambda} \quad \Psi(\lambda), \tag{1.65}$$

[1] No se debe confundir con la función de error de la retropropagación en la Subsección 1.3

con

$$\Psi(\boldsymbol{\lambda}) = \frac{1}{|\mathcal{D}_{\mathcal{V}}|} \sum_{x \in \mathcal{D}_V} \mathcal{L}(\boldsymbol{f}_{\boldsymbol{\lambda}}, \mathcal{D}_E, \mathcal{D}_V). \tag{1.66}$$

Hemos llegado a una expresión del problema en términos de una función $\Psi(\boldsymbol{\lambda}) : \Lambda \longrightarrow \mathbb{R}$ que se conoce habitualmente como *superficie de respuesta* en la rama estadística del diseño de experimentos [3].

Observación 1.18. *(Sobre los conjuntos de entrenamiento, validación y prueba) El conjunto de entrenamiento $\mathcal{D}$ introducido en la Sección 1.3 ya es conocido. El conjunto de validación es otro conjunto de pares de datos utilizado para el ajuste de los hiperparámetros y para cuantificar la capacidad de generalización del modelo. Por lo tanto, estos dos conjuntos se deben tomar distintos para evitar sobreajuste. En caso contrario, el modelo podría aprender el conjunto de datos de validación y aparentar un rendimiento óptimo. Por último, el conjunto de prueba se utiliza para comprobar la precisión del modelo una vez ajustados los parámetros e hiperparámetros.*

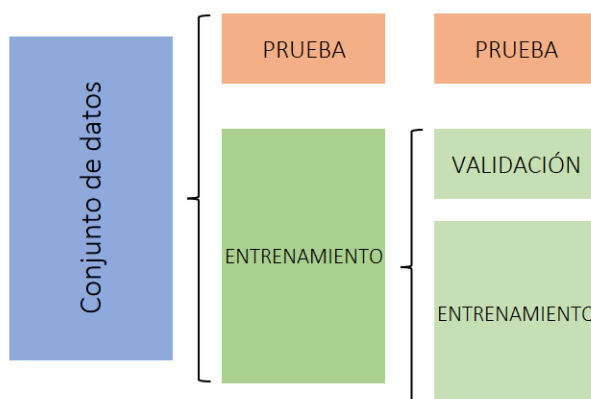

Figura 1.4: Esquema que muestra la división habitual de los datos en el proceso de ajuste de un modelo. En la división de la derecha observamos los conjuntos $\mathcal{D}_E$ y $\mathcal{D}_V$ en verde.

Normalmente, el conjunto total de datos se dividirá en una proporción de 80-20 o 70-30 para los datos de entrenamiento y de prueba, respectivamente [11, pág. 119]. Dentro del conjunto de datos de entrenamiento, se suele realizar algún tipo de validación cruzada, sobre todo cuando el conjunto de datos es pequeño. En el caso más sencillo, se puede realizar otra división en el conjunto de entrenamiento entre los datos de entrenamiento de los parámetros, $\mathcal{D}_E$, y los de validación $\mathcal{D}_V$.

1.6. Métodos

La función $\Psi(\boldsymbol{\lambda})$ que queremos optimizar es, por lo general, desconocida. Mucho menos podemos esperar que tenga propiedades deseables para la optimización como convexidad o diferenciabilidad respecto de $\boldsymbol{\lambda}$. Por lo tanto, el enfoque a seguir será el de tomar una muestra $\{\boldsymbol{\lambda}^{(1)}, ..., \boldsymbol{\lambda}^{(S)}\} \subset \Lambda$ en la que evaluar $\Psi(\cdot)$ y quedarnos con la combinación de hiperparámetros que tenga mejor resultado:

$$\boldsymbol{\lambda}^* \approx \underset{\boldsymbol{\lambda} \in \{\boldsymbol{\lambda}^{(1)}, ..., \boldsymbol{\lambda}^{(S)}\}}{\arg\text{mín}} \Psi(\boldsymbol{\lambda}). \tag{1.67}$$

El problema ahora será tomar esta muestra de manera que se minimize de alguna forma el número de puntos evaluados S, puesto que las evaluaciones de la función Ψ son costosas computacionalmente. En efecto, debemos recordar que cada evaluación de la función Ψ requiere el entrenamiento de una red neuronal como se puede ver en 1.66.

En esta sección introduciremos los principales métodos para abordar este problema. Estos métodos no utilizan la derivada de la función ni ninguna información adicional sobre ella, por lo que se enmarcan dentro de la rama de la optimización *black-box*.

1.6.1. Barrido paramétrico

El *barrido paramétrico* (*Grid Search* en inglés) es la técnica más básica debido a su simplicidad conceptual, por lo que junto al ajuste manual todavía es una de las estrategias más utilizadas, como se puede ver en [3].

Para cada hiperparámetro $\lambda_k \in \Lambda_k$, $k \in \{1, ..., N\}$, se elige un conjunto finito de valores $L_k \subset \Lambda_k$. El barrido paramétrico consiste en evaluar la función Ψ en cada punto de $L = L_1 \times L_2 \times ... \times L_N$ y escoger el $\boldsymbol{\lambda} \in L$ que minimize Ψ. El número de evaluaciones de la función Ψ a realizar será $\prod_{k=1}^{N} |L_k|$.

El talón de Aquiles de esta técnica es la llamada maldición de la dimensionalidad (en inglés, *curse of dimensionality*). A medida que aumenta la cantidad de hiperparámetros a optimizar, el método será más costoso computacionalmente, hasta llegar al punto de que este método resulta inviable para redes profundas con un gran número de hiperparámetros. En efecto, como el número de evaluaciones crece exponencialmente con N, si queremos que cada hiperparámetro explore 10 valores distintos (es decir, $|L_1| = ... = |L_N| = 10$) el número de evaluaciones totales será 10^N.

En conclusión, el barrido paramétrico resulta una buena opción solamente cuando existe conocimiento previo sobre el espacio de hiperparámetros y este es de dimensión baja [24].

1.6.2. Búsqueda aleatoria (*Random Search*)

La *búsqueda aleatoria* consiste en evaluar la función Ψ en puntos al azar de Λ de manera independiente. Mientras que en el barrido parámetrico utilizábamos una malla regular de hiperparámetros, ahora tendremos una malla irregular de puntos donde hemos evaluado la función de coste.

Este método supone una primera mejora al barrido paramétrico: Si contamos con un número máximo de evaluaciones M. Como se puede ver en la Figura 1.5, el primer método nos permite probar solamente $M^{1/N}$ valores de cada uno de los N hiperparámetros, mientras que con la búsqueda aleatoria evaluaremos la función en M valores distintos.

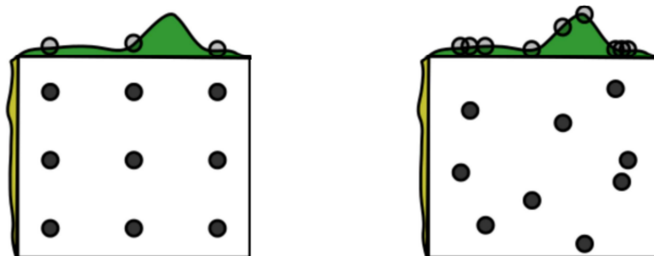

Figura 1.5: Esquema comparativo entre el barrido paramétrico (izquierda) y la búsqueda aleatoria (derecha). Los ejes representan dos hiperparámetros λ_1, λ_2. Sobre cada eje esta la función objetivo respecto de cada hiperparámetro. Obtenido de [3]
.

1.6.3. Optimización Bayesiana

La *optimización bayesiana* (OB) es un método de optimización global e iterativo utilizado para resolver problemas de la forma 1.65 en los que la función objetivo es desconocida. El algoritmo de optimización bayesiana expuesto en el Algoritmo 2 calcula en cada iteración el siguiente punto donde evaluar la función objetivo teniendo en cuenta las evaluaciones anteriores. De esta forma, el algoritmo minimiza el número de evaluaciones de la función para llegar a un resultado razonable.

La optimización bayesiana se basa en dos componentes principales:

- Un modelo estadístico (paramétrico o no paramétrico) sustituto que reemplaza a la función objetivo y cuya distribución es conocida. Los modelos más utilizados para este fin en el ámbito del aprendizaje automático son los procesos gaussianos presentados en el Anexo I.

- Una función de utilidad $u : \Lambda \longrightarrow \mathbb{R}$ que sirve para escoger el siguiente punto de la muestra. Es la encargada del balance entre la explotación (tomar puntos en donde el modelo sustituto toma valores altos) y la exploración (tomar puntos donde el modelo sustituto no tiene información sobre la función objetivo, es decir, donde existe una incertidumbre mayor).

El algoritmo parte de un punto, o conjunto de ellos, $\boldsymbol{\lambda}^{(1)} \in \Lambda$ en el que se evalúa la función Ψ para tener un punto de partida. Denotamos por $L^{(n)} = \{(\boldsymbol{\lambda^{(1)}}, \Psi(\boldsymbol{\lambda^{(1)}})), ..., (\boldsymbol{\lambda^{(n)}}, \Psi(\boldsymbol{\lambda^{(n)}}))\}$ a los puntos evaluados hasta la iteración n-ésima, a los que se añade un nuevo par de puntos en cada iteración. En el paso n del algoritmo, se combina una distribución a priori sobre Ψ, $P(\Psi)$, con la función de verosimilitud $P(L^{(n)}|\Psi)$ para obtener la distribución a posteriori (utilizando el Teorema de Bayes):

$$P(\Psi|L^{(n)}) \propto P(L^{(n)}|\Psi)P(\Psi). \tag{1.68}$$

Esta distribución a posteriori es la que aproxima nuestra función Ψ basándose en los datos hasta esa iteración. En la Figura 1.6 podemos ver la función objetivo (la línea punteada) y la distribución a posteriori $\mu_n(\boldsymbol{\lambda}) \pm \sigma_n(\boldsymbol{\lambda})$, donde $\mu_n(\boldsymbol{\lambda})$ es la media y $\sigma_n(\boldsymbol{\lambda})$ la desviación típica de la distribución a posteriori, como se indica en el Anexo I.

En [23, pág. 7] se demuestra experimentalmente que el uso del algoritmo de optimización bayesiana para el problema de HPO ofrece importantes mejoras con respecto al uso del *random search*. En algunos casos, la eficiencia de las evaluaciones con el uso de la OB llegó a ser 100 veces superior con respecto a la búsqueda aleatoria. Por el contrario, el algoritmo de OB no resulta tan fácil de paralelizar como el RS puesto que utiliza los puntos anteriores para guiar las elecciones.

Funciones de utilidad

La *función de utilidad* u es la pieza clave encargada de guiar el proceso de optimización. En general, son funciones reales definidas de forma que valores altos de u indican posibles valores bajos de la función objetivo Ψ. De esta forma, en el paso n del algoritmo:

$$\boldsymbol{\lambda^{(n+1)}} = \underset{\boldsymbol{\lambda} \in \Lambda}{\arg\max} \quad u(\boldsymbol{\lambda}|L^{(n)}), \tag{1.69}$$

donde la función de utilidad para el paso n-ésimo se denota por $u(\boldsymbol{\lambda}|L^{(n)})$ puesto que esta se construye a partir de la $\mu_n(\boldsymbol{\lambda})$ y $\sigma_n(\boldsymbol{\lambda})$ de la distribución a posteriori, que depende a su vez de la muestra de puntos $L^{(n)}$.

Las dos funciones de utilidad más conocidas son:

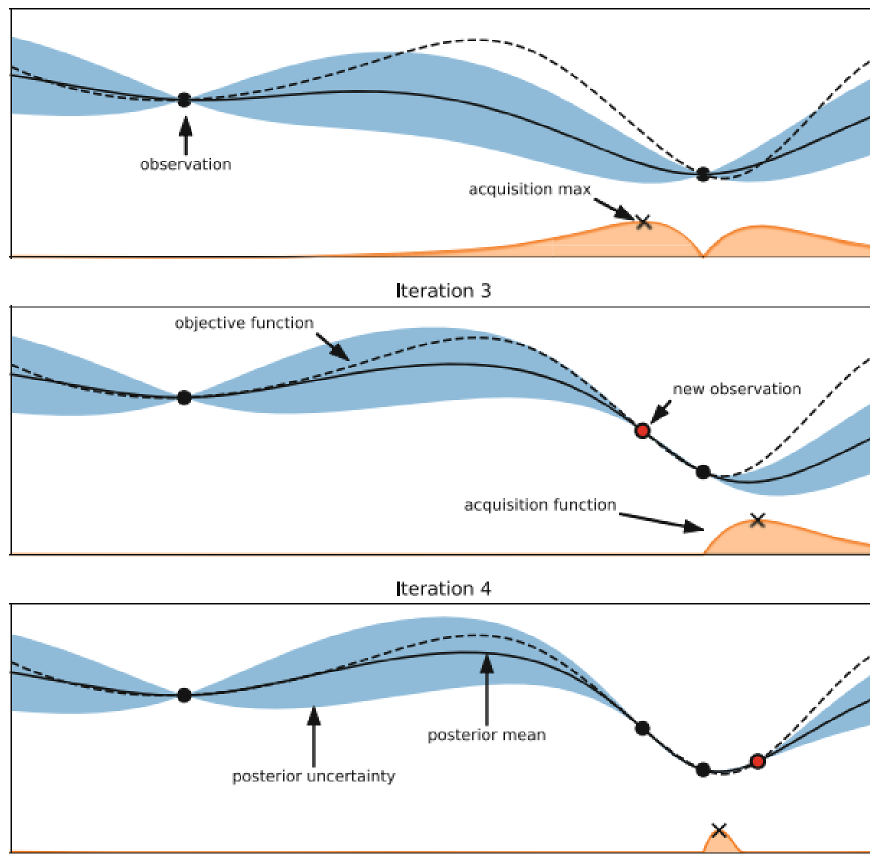

Figura 1.6: Tres pasos del algoritmo de optimización bayesiana. Extraído de [13].

- La *probabilidad de mejora* (en inglés *probability of improvement*) calcula la probabilidad en cada punto de que la observación sea menor que el mejor valor obtenido hasta el momento, $\Psi(\boldsymbol{\lambda}^-)$:

$$PI(\boldsymbol{\lambda}) = P\left(\Psi(\boldsymbol{\lambda}) \leq \Psi(\boldsymbol{\lambda}^-) - \xi\right) = \Phi\left(\frac{\Psi(\boldsymbol{\lambda}^-) + \xi - \mu_n(\boldsymbol{\lambda})}{\sigma_n(\boldsymbol{\lambda})}\right). \tag{1.70}$$

  donde Φ es la función de distribución de la normal estándar y $\xi \geq 0$ es una constante que se añade para mejorar el balance entre explotación-exploración. En efecto, esta función de utilidad tiende fuertemente a la explotación de las zonas con menos incertidumbre.

- La *mejora esperada* (en inglés *expected improvement*) se obtiene a partir de la función de mejora $I(\boldsymbol{\lambda}) = \text{máx}\{0, \Psi(\boldsymbol{\lambda}^-) - \Psi(\boldsymbol{\lambda})\}$. Vemos que esta función tomará valores mayores que 0 en los puntos susceptibles de mejorar. La esperanza de la función $I(\boldsymbol{\lambda})$ se puede evaluar como sigue [5, pág. 13]:

$$EI(\boldsymbol{\lambda}) = \begin{cases} (\Psi(\boldsymbol{\lambda}^+) + \xi - \mu_n(\boldsymbol{\lambda}))\Phi(Z) + \sigma_n(\boldsymbol{\lambda})\phi(Z) & \text{si } \sigma(\boldsymbol{\lambda}) > 0 \\ 0 & \text{si } \sigma(\boldsymbol{\lambda}) = 0 \end{cases}, \tag{1.71}$$

donde

$$Z = \frac{\mu_n(\boldsymbol{\lambda}) + \xi - \Psi(\boldsymbol{\lambda}^-)}{\sigma_n(\boldsymbol{\lambda})} \tag{1.72}$$

y ϕ es la función de densidad de la distribución normal estándar.

Algoritmo 2 Optimización bayesiana

Entrada: itmax ▷ núm. máximo de iteraciones
Salida: $\lambda^{(itmax)}$
  Se parte de $L^{(1)}$ ▷ Una muestra inicial de uno o varios pares de puntos
  for $n = 1, 2, ..., itmax$ **do**
    $\lambda^{(n+1)} = \arg\max_{\lambda \in \Lambda} \quad u(\lambda | L^{(n)})$ ▷ Cálculo del siguiente punto
    Evaluamos $\Psi(\lambda^{(n+1)})$ ▷ Evaluación de la función en el punto
    $L^{(n+1)} = \{L^{(n)}, (\lambda^{(n+1)}, \Psi(\lambda^{(n+1)}))\}$ ▷ Se añade el nuevo dato a la muestra
    Se actualiza el modelo estadístico con $L^{(n+1)}$
  end for

1.6.4. Otros métodos

Los tres métodos clásicos que hemos visto parten de la misma idea reflejada en 1.67: son métodos que se basan en evaluar repetidamente la función objetivo en distintos puntos. En la literatura estos algoritmos se conocen como métodos de búsqueda [24] o métodos de optimización *black-box*. Otro ejemplo de método *black-box* más avanzado son los *algoritmos evolutivos* (EA). En líneas generales, estos métodos parten de un conjunto inicial de puntos que, a través de selección y reproducción (es decir, combinaciones y modificaciones) de los mejores candidatos, evolucionan hacia mejores resultados de la función objetivo.

No obstante también existen otros enfoques a la hora de abordar el problema de HPO. El desarrollo del aprendizaje profundo en los últimos tiempos ha dado lugar lugar a modelos más complejos con una gran cantidad de hiperparámetros y con grandes conjuntos de datos de entrenamiento. Debido a esto, los métodos *black-box* estudiados se vuelven extremadamente lentos o incluso inviables computacionalmente al intentar ajustar estos modelos.

La familia de métodos con *detención temprana* abordan este problema. Muchos de estos métodos utilizan alguno de los métodos ya vistos junto a un criterio de parada. El funcionamiento general de estos algoritmos se basa en entrenar múltiples modelos con distintos hiperparámetros y descartar prematuramente los modelos con peores resultados en base a cierto criterio. De esta manera, no es necesario entrenar todos los modelos hasta la convergencia, sino que la capacidad computacional se reserva a los modelos que se vayan desempeñando mejor. En[24] se realiza una revisión de los principales criterios.

Por último, debemos mencionar que existen métodos recientes que utilizan el descenso de gradiente para optimizar hiperparámetros. En [15] se calculan los gradientes respecto de los hiperparámetros para una red entrenada con descenso de gradiente estocástico con momento.

Capítulo 2

Un ejemplo en R

Vamos a implementar un ejemplo sencillo de red neuronal utilizando las librerías `keras` y `tensorflow` del software estadístico R. El código utilizado se puede consultar en el Anexo II.

Contamos con un conjunto de datos de tamaño 397 extraído del repositorio de datos de la UC Irvine [17]. Se desea conocer el número de cilindros de un coche `cyl` a partir de aceleración del coche `acc` y su peso `weight`. Existen 5 cilindradas distintas, por lo que `cyl` es una variable que toma los valores $\{0, 1, 2, ..., 4\}$. Se divide el 80 % como datos de entrenamiento y el 20 % como datos de prueba, como se puede ver en la Figura 2.1.

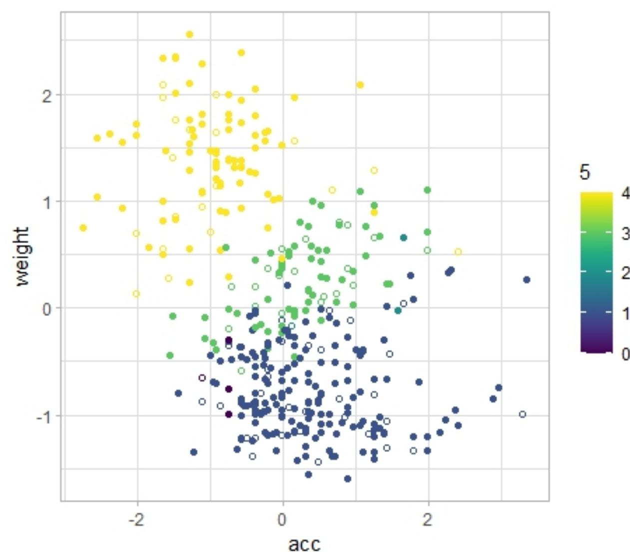

Figura 2.1: Diagrama de dispersión del conjunto de datos. Los puntos sin relleno son los puntos de prueba escogidos aleatoriamente.

Importancia del *learning rate*

Estamos ante un problema de *clasificación multiclase*. Se diseña un modelo sencillo de una única capa oculta con funciones de activación ReLU de tamaño 5 y una capa de salida con función de activación Softmax de tamaño 5, siguiendo las indicaciones en la Subsección 1.4.

La red se entrena utilizando descenso de gradiente estocástico con momento. Se escoge un tamaño de *minibatch* de 32 y un número de pasos 50. En las Figuras 2.2 y 2.3 se observan las curvas de entrenamiento para dos valores de paso distintos. Mientras que con un paso 0,01 se consigue un 0,9211 de precisión al evaluar el modelo con los datos de prueba, con un paso de 2 solo llega al 0,4006 de precisión.

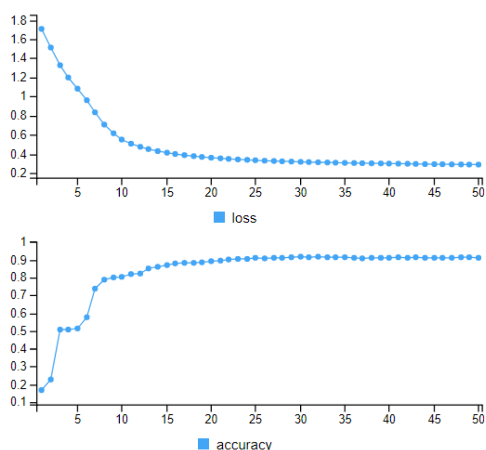

Figura 2.2: $\eta = 0{,}01$ y $\beta = 0{,}9$

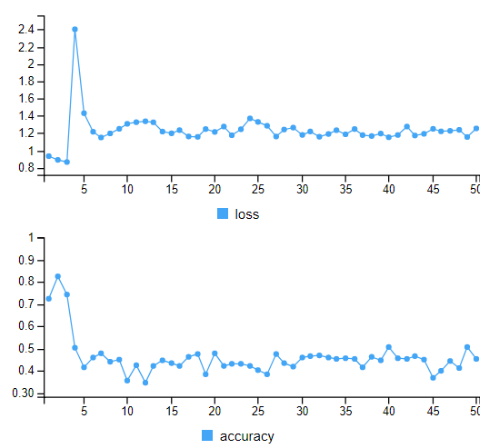

Figura 2.3: $\eta = 2$ y $\beta = 0{,}9$

Grid search y *random search*

El objetivo ahora es implementar los métodos de barrido paramétrico y búsqueda aleatoria sobre un mismo modelo con el fin de compararlos. Realizaremos el ajuste del paso $\eta \in (0, 1]$ y del momento $\beta \in [0, 1)$. Tomamos un modelo todavía más simple que el anterior, con una única capa de salida de 5 neuronas con función de activación softmax. Fijamos un número de pasos, o *epochs*, de 5. Esta es una cantidad muy pequeña, pero al tratarse de un modelo tan sencillo, si añadiesemos más pasos posiblemente todas las opciones de hiperparámetros convergerían a una precisión similar.

Se preparan las mallas para ambos métodos. Para que la comparación tenga sentido las mallas deben ser del mismo tamaño, que será 100. Además, puesto que queremos que la variación del *learning rate* entre 0,001 y 0,002 sea tan importante como de 0,1 a 0,2, utilizaremos un escalado

logarítmico. Para el momento ocurre lo mismo pero en torno a los valores cercanos a 1. Las mallas resultantes aparecen representadas en las Figuras 2.4 y 2.5.

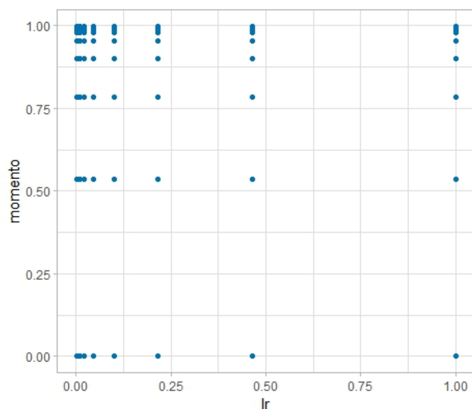

Figura 2.4: Malla regular

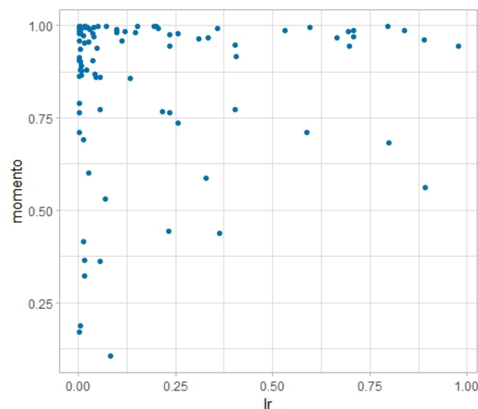

Figura 2.5: Malla aleatoria

Finalmente, se evalúa la precisión y la pérdida con cada combinación de hiperparámetros de las dos mallas. Se utiliza como métrica de pérdida la *entropía cruzada* con los datos de validación del modelo. Los resultados del experimento aparecen en las Tablas 2.1 y 2.2 y en las Figuras 2.6 y 2.7.

	Mín.	Mediana	Media	Máx.
Grid search	0.7319	0.9180	0.9106	0.9306
Random search	0.8801	0.9180	0.9176	0.9274

Cuadro 2.1: Precisión para cada método.

	Mín.	Mediana	Media	Máx.
Grid search	0.2305	0.2358	0.2623	1.1956
Random search	0.2302	0.2318	0.2355	0.2706

Cuadro 2.2: Pérdida para cada método.

Figura 2.6: Comparación de la precisión

Figura 2.7: Comparación de la pérdida

Capítulo 3

Conclusiones

En este trabajo se realizó una revisión general de las dificultades que supone optimizar los hiperparámetros de una red neuronal, presentando los métodos más utilizados en la literatura para abordar dicho problema. Además, se llevó a cabo una revisión particularizada de los principales hiperparámetros, destacando las ventajas y desventajas de las distintas opciones para cada uno. De esta manera, se aportan al lector las bases necesarias para entender este problema y su actualidad debido al desarrollo del aprendizaje profundo. La optimización de hiperparámetros juega un papel todavía más importante si cabe en el *aprendizaje automático automatizado*, conocido en inglés como *AutoML*. Este enfoque pretende hacer posible el uso modelos aprendizaje automático a medida para cualquier usuario independientemente de su conocimiento en el campo. Este hecho es posible, en buena medida, a la automatización del ajuste de hiperparámetros.

En la última parte del trabajo se utiliza `R` para presentar un pequeño ejemplo práctico a partir de un problema de clasificación. Se pusieron a prueba las técnicas de *grid search* y *random search* obteniendo resultados similares con ambos métodos, llegando a precisiones de $\approx 0{,}93$ en ambos casos. Con este ejemplo se mostró también que es posible implementar redes neuronales sencillas en `R`, utilizando las librerías `keras` y `tensorflow`.

Anexo I

Procesos Gaussianos

En esta sección se exponen las nociones básicas relativas a los procesos gaussianos, el tipo de proceso estocástico más utilizado para modelar la función objetivo f.

Definimos $x_t \in A$ como la muestra t-ésima e $y_t = f(x_t) + \epsilon_t$ con $\epsilon_t \sim \mathcal{N}(0, \sigma^2)$.

Definición I.1. Un *proceso gaussiano* (en adelante, PG) es una colección de variables aleatorias satisfaciendo que cualquier subconjunto finito de la colección sigue una distribución normal multivariante.

Un PG queda totalmente determinado por su función de medias y su función de covarianzas,

Definición I.2. Definimos la función de medias de un proceso real $f(x)$ como

$$m(x) = \mathbb{E}[f(x)] = \begin{pmatrix} m(f(x_1)) \\ \vdots \\ m(f(x_d)) \end{pmatrix}, \tag{I.1}$$

y la función de covarianzas como

$$k(x, x') = \mathbb{E}[(f(x) - m(x))(f(x') - m(x'))]. \tag{I.2}$$

Por lo tanto, denotaremos el proceso gaussiano como

$$f(x) \sim GP(m(x), k(x_i, x_j)). \tag{I.3}$$

Pero, ¿cómo utilizamos los PG para optimizar una función f? Supongamos que ya tenemos t muestras $\{x_{1:t}, y_{1:t}\}$ y queremos buscar qué punto tomar a continuación. Denotemos tal punto

por x_{t+1} y el valor de la función en él por f_{t+1}. Se tendrá que (incluir la propiedad de los GP que permite esto antes)

$$\begin{pmatrix} f_{1:t} \\ f_{t+1} \end{pmatrix} \sim N\left(0, \begin{pmatrix} K & k \\ k^t & k(x_{t+1}, x_{t+1}) \end{pmatrix}\right),$$

donde $k = (k(x_{t+1}, x_1), k(x_{t+1}, x_2), \ldots, k(x_{t+1}, x_t))$ y K es la matriz de covarianzas con entradas $k(x_i, x_j)$ para $1 \leq i, j \leq t$. Se tiene el siguiente resultado:

$$P(y_{t+1}|\{x_{1:t}, y_{1:t}\}, x_{t+1}) = N(\mu_t(x_{t+1}), \sigma_t^2(x_{t+1}) + \sigma^2), \tag{I.4}$$

donde

$$\mu_t(x_{t+1}) = k^t(K + \sigma^2 I)^{-1} y_{1:t}, \tag{I.5}$$

$$\sigma_t^2(x_{t+1}) = k(x_{1:t}, x_{1:t}) - k^t(K + \sigma^2 I)^{-1} k. \tag{I.6}$$

Anexo II

Código

En este Anexo se incluye el código en R utilizado para la Sección 2.

```r

set.seed(1502)
rm(list=ls())
setwd('C:/Users/arnic/OneDrive/Escritorio/TFG/Scripts')

library(tensorflow)
library(keras)
library(tidyverse)
library(tidymodels)
library(patchwork)
library(viridis)

###PREPARACION DE LOS DATOS DE INTERES

df<-read.table('auto-mpg.data',header=T,sep='')
df<-datos<-df[,c(6,5,2)]
colnames(df)<-c("acc","weight","cyl")

df[,1]<-scale(df[,1])
df[,2]<-scale(df[,2])

df$cyl[cyl == 3] <- 0
df$cyl[cyl == 4] <- 1
df$cyl[cyl == 5] <- 2
df$cyl[cyl == 6] <- 3
df$cyl[cyl == 8] <- 4

ggplot(df, aes(x = acc, y = weight, color = cyl)) +
  geom_point()+
```

```
30   theme_light()+
31   scale_color_viridis(5)
32
33 ### DIVISION ENTRENAMIENTO/VALIDACION
34
35 split <- initial_split(df, 0.8)
36 train <- training(split)
37 test <- testing(split)
38
39 ggplot() +
40   geom_point(data = test, aes(x = acc, y = weight,color=cyl), shape = 1) +
41   geom_point(data = train, aes(x = acc, y = weight,color=cyl))+
42   scale_color_viridis(5)+
43   theme_light()
44
45
46
47 ### PRIMER EJEMPLO
48
49 modelo1 <- keras_model_sequential()
50 modelo1 %>%
51   layer_dense(units = 5, activation = 'relu', input_shape = c(2)) %>%
52   layer_dense(units = 5, activation = 'softmax') %>%
53   compile(
54     loss = 'sparse_categorical_crossentropy',
55     optimizer = optimizer_sgd(learning_rate = 0.01, momentum = 0.9),
56     metrics = c('accuracy')
57   )
58
59 summary(modelo1) #TABLA DEL DISE O
60
61 ### SE ENTRENA EL MODELO CON EL CJTO. DE ENTRENAMIENTO Y SE EVALUA CON EL CJTO. DE
      PRUEBA
62
63 modelo1 %>% fit(as.matrix(train[,c(1,2)]), as.matrix(train[,3]), epochs = 50,
      batch_size = 32)
64 modelo1 %>% evaluate(as.matrix(test[,c(1,2)]), as.matrix(test[,3]), batch_size =
      32)
65
66
67
68
69
70 ##############################
71 ##############################
72
73
```

```
74
75
76 ### IMPLEMENTACIN DE GRID SEARCH Y RANDOM SEARCH
77
78 ### El modelo en este caso ser :
79
80 modelo2 <- keras_model_sequential()
81 modelo2 %>%
82   layer_dense(units = 5, activation = 'softmax',input_shape = c(2))
83
84 summary(modelo2)
85
86 ### Calculamos las mallas:
87
88 lr_grid<-10**seq(from=-3,to=0,length.out=10)
89 m_grid<-1-10**seq(from=-3,to=0,length.out=10)
90 hp_grid<-expand.grid(lr_grid,m_grid)
91
92 hp_rand<-data.frame(10**runif(100,min=-3,max=0),1-10**runif(100,min=-3,max=0))
93
94 colnames(hp_grid)<-c("lr","momento")
95 colnames(hp_rand)<-c("lr","momento")
96
97 ggplot(hp_grid, aes(x = lr, y = momento)) +
98   geom_point(color="#0072B2")+
99   theme_light()
100
101 ggplot(hp_rand, aes(x = lr, y = momento)) +
102   geom_point(color="#0072B2")+
103   theme_light()
104
105
106 ### FUNCION QUE ENTRENA EL MODELO en funcion de LR y MOMENTO
107
108 npasos<-5
109
110 trainfun<- function(lr,m){
111
112   modelo2 %>% compile(
113     loss = 'sparse_categorical_crossentropy',
114     optimizer = optimizer_sgd(learning_rate = lr, momentum = m),
115     metrics = c('accuracy')
116   )
117
118   foo <- modelo2 %>% fit(as.matrix(train[,c(1,2)]), as.matrix(train[,3]), epochs =
        npasos, batch_size = 32)
119
```

```
  perdida<-foo$metrics$loss[npasos]
  precis<-foo$metrics$accuracy[npasos]

  reset_states(modelo2)

  return(c(perdida, precis))
}

### EVALUAMOS LAS FUNCIONES EN LOS PUNTOS PARA EL GRID SEARCH
### Y guardamos la perdida/precision en cada punto

perdida_grid<-c()
precis_grid<-c()

for (k in 1:dim(hp_grid)[1]) {

  aux<-trainfun(hp_grid[k,1],hp_grid[k,2])

  perdida_grid<-c(perdida_grid,aux[1])
  precis_grid<-c(precis_grid,aux[2])
}

### EVALUAMOS LAS FUNCIONES EN LOS PUNTOS PARA EL RANDOM SEARCH
### Y guardamos la perdida/precision en cada punto

perdida_rand<-c()
precis_rand<-c()

for (k in 1:dim(hp_rand)[1]) {

  aux<-trainfun(hp_rand[k,1],hp_rand[k,2])

  perdida_rand<-c(perdida_rand,aux[1])
  precis_rand<-c(precis_rand,aux[2])
}



### GRAFICAMOS LOS RESULTADOS

grupo = factor(rep(c("Grid", "Random"), each = 100))
perdida<-c(perdida_grid,perdida_rand)
precis<-c(precis_grid,precis_rand)

resultados <- data.frame(grupo,perdida,precis)

ggplot() +
```

```
167   geom_boxplot(aes(x="Malla regular", y=perdida_grid), fill="lightcyan") +
168   geom_boxplot(aes(x= "Malla aleatoria", y=perdida_rand), fill="#FFFACD") +
169   ylab("P rdida") +
170   xlab("")+
171   theme_light()
172 
173 ggplot() +
174   geom_boxplot(aes(x="Malla regular", y=precis_grid), fill="lightcyan") +
175   geom_boxplot(aes(x= "Malla aleatoria", y=precis_rand),fill="#FFFACD") +
176   ylab("Precisi n") +
177   xlab("")+
178   theme_light()
179 
180 ### RESUMEN DE LA PRECISION Y PERDIDA
181 
182 summary(perdida_grid)
183 summary(perdida_rand)
184 
185 summary(precis_grid)
186 summary(precis_rand)
```

prueba2.R

Bibliografía

[1] Y. Bengio. Learning deep architectures for ai. *Foundations*, 2:16, 01 2009.

[2] Y. Bengio. Practical recommendations for gradient-based training of deep architectures. 6 2012. `http://arxiv.org/abs/1206.5533`.

[3] J. Bergstra, J. B. Ca, and Y. B. Ca. Random search for hyper-parameter optimization yoshua bengio, 2012. `http://scikit-learn.sourceforge.net.`

[4] L. Bottou. *On-Line Learning and Stochastic Approximations*, page 9–42. Cambridge University Press, USA, 1999.

[5] E. Brochu, V. M. Cora, and N. de Freitas. A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning, 2010.

[6] M. Collins. Convergence proof for the perceptron algorithm. `http://www.cs.columbia.edu/~mcollins/courses/6998-2012/notes/perc.converge.pdf`.

[7] K. L. Du and M. N. Swamy. *Neural networks and statistical learning, second edition.* 2019.

[8] A. L. Fradkov. Early history of machine learning. *IFAC-PapersOnLine*, 53(2):1385–1390, 2020. 21st IFAC World Congress.

[9] X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, 2010.

[10] X. Glorot, A. Bordes, and Y. Bengio. Deep sparse rectifier neural networks, 2011.

[11] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning.* MIT Press, 2016. `http://www.deeplearningbook.org`.

[12] S. Haykin. *Neural Networks: A Comprehensive Foundation.* Prentice Hall, 1998.

[13] F. Hutter, L. Kotthoff, and J. Vanschoren. *Automated machine learning: Methods, Systems, Challenges.* 2019.

[14] D. Kingma and J. Ba. Adam: A method for stochastic optimization. *International Conference on Learning Representations*, 12 2014.

[15] D. Maclaurin, D. Duvenaud, and R. P. Adams. Gradient-based hyperparameter optimization through reversible learning. 2 2015. `http://arxiv.org/abs/1502.03492`.

[16] S. Narayan. The generalized sigmoid activation function: Competitive supervised learning. *Information Sciences*, 99(1):69–82, 1997.

[17] R. Quinlan. Auto MPG. UCI Machine Learning Repository, 1993. `https://doi.org/10.24432/C5859H`.

[18] P. Ramachandran, B. Zoph, and Q. V. Le. Searching for activation functions. 10 2017. `http://arxiv.org/abs/1710.05941`.

[19] D. Rumelhart, G. Hinton, and J. Mcclelland. A general framework for parallel distributed processing. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, 1, 01 1986.

[20] D. E. Rumelhart and J. L. McClelland. *Learning Internal Representations by Error Propagation*, pages 318–362. 1987.

[21] S. Simon, N. Kolyada, C. Akiki, M. Potthast, B. Stein, and N. Siegmund. Exploring hyperparameter usage and tuning in machine learning research. pages 68–79. IEEE, 5 2023.

[22] H. Sompolinsky. Introduction: The perceptron, 2013. `https://web.mit.edu/course/other/i2course/www/vision_and_learning/perceptron_notes.pdf`.

[23] R. Turner, D. Eriksson, M. McCourt, J. Kiili, E. Laaksonen, Z. Xu, and I. Guyon. Bayesian optimization is superior to random search for machine learning hyperparameter tuning: Analysis of the black-box optimization challenge 2020. *CoRR*, abs/2104.10201, 2021.

[24] T. Yu and H. Zhu. Hyper-parameter optimization: A review of algorithms and applications. 3 2020. `http://arxiv.org/abs/2003.05689`.