TRABAJO FIN DE GRADO

Doble Grado Ingeniería Informática y Matemáticas

Exploración de técnicas de entrenamiento de redes neuronales profundas

Enfoques clásicos y técnicas metaheurísticas

Autor
Eduardo Morales Muñoz

Directores
Pablo Mesejo Santiago
Javier Merí de la Maza

Facultad de Ciencias
Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

—

Granada, diciembre de 2024

Índice

1. Introducción

El aprendizaje automático es una rama de la inteligencia artifical en la que los sistemas son capaces de adquirir conocimiento a partir de datos sin procesar [GBC16]. Se dice que un programa aprende de la experiencia E respecto de alguna tarea T y una medición de rendimiento P si su rendimiento en T, medido por P, mejora con la experiencia E [Mit97]. Nos referimos a este programa como modelo. Existen muchos tipos o subramas de aprendizaje automático dependiendo de la naturaleza de esta tarea T y de su medidor de rendimiento P.

El entrenamiento de un modelo es el proceso de optimizar sus parámetros (equivalentemente pesos), es decir, su representación interna; para minimizar una función de coste (equivalentemente función de error o de pérdida) C que mide el error en el rendimiento. El dominio de dicha función es el espacio de valores que pueden tomar los pesos, normalmente representado de forma tensorial; y su imagen es comúnmente un real no negativo. El objetivo principal del entrenamiento es que el modelo sea capaz de aprender los patrones en un conjunto de datos para luego poder generalizarlos en otros que no ha visto previamente. Diremos que existe un sobreajuste cuando se aprenden los patrones específicos de los datos pero luego no se generaliza bien. La estrategia que usamos para optimizar los pesos es llamada algoritmo de aprendizaje.

El aprendizaje profundo es un paradigma del aprendizaje automático en el que los modelos tienen varios niveles de representación obtenidos a través de la composición de módulos sencillos pero comúnmente no lineales, que transforman la representación de los datos sin procesar hacia un nivel de abstracción mayor [LBH15]. Esta rama comenzó a ganar peso en la década de los 2000 y un punto de inflexión fue el resultado de la competición de ImageNet [1] en 2012 [KSH12]. Actualmente este enfoque es el que mejores resultados consigue, siendo una parte fundamental en la investigación y estructura de las grandes compañías tecnológicas y pudiendo ofrecer aplicaciones comerciales a nivel usuario [Sej18; BLH21].

La mayoría de los modelos en aprendizaje automático se entrenan usando técnicas basadas en el algoritmo de aprendizaje de gradiente descendente (equivalentemente descenso del gradiente), ya que es la estrategia que mejores resultados ofrece actualmente en cuanto a capacidad de generalización del modelo y rendimiento computacional [GBC16; Cau09]. Ésta se basa en la idea de que puedo moverme hacia puntos de menor valor en la función de error del modelo realizando pequeños movimientos en sentido contrario a su gradiente como se esquematiza en la figura 1, con el objetivo de minimizar el valor de salida. Al tratarse de un algoritmo iterativo, es fundamental estudiar su convergencia, que depende de varios factores y se enfrenta a diversas

[1] http://www.image-net.org/challenges/LSVRC/

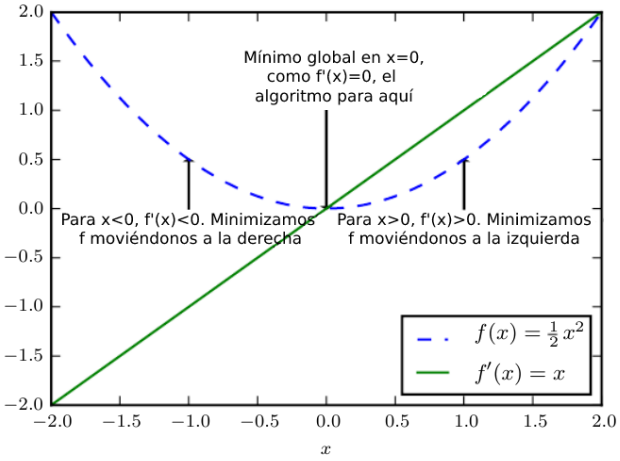

Figura 1: Esquematización de la estrategia de descenso del gradiente en un modelo con un solo parámetro x. El eje horizontal representa los valores que toma éste y el vertical representa el error del modelo en función de x. Imagen obtenida y traducida del libro [GBC16]

dificultades, como veremos en secciones posteriores.

El algoritmo de *backpropagation* (BP) permite transmitir la información desde la salida de la función de coste hacia atrás en un modelo con varios módulos de abstracción para así poder computar el gradiente de una manera sencilla y eficiente [RHW86]. Aunque existen otras posibilidades a la hora de realizar éste cómputo, BP es la más usada y extendida gracias a propiedades como su flexibilidad, eficiencia y escalabilidad, que lo hacen destacar por encima de otras opciones [GBC16].

Dependiendo de la familia de modelos que usemos podremos utilizar una estrategia de aprendizaje distinta, como el caso del *Perceptron* y su *Perceptron Learning Algorithm* [Bis06]. En otros casos como la regresión lineal se usa la estrategia de descenso de gradiente pero el gradiente no tiene por qué calcularse a través de BP. Esto se debe a que en este caso se puede obtener eficientemente a través de librerías matemáticas como *numpy* [2] en el caso del lenguaje *python* [3], ya que esta familia de modelos conllevan menos costo computacional en sus cálculos principalmente debido al escaso número de parámetros en comparación con los de aprendizaje profundo. Para éste sí que es necesario el uso de BP en el caso de que elijamos entrenar mediante gradiente descendente, ya que aunque existen otras alternativas como los métodos numéricos o algunas aproximaciones recientes , no consiguen igualar su rendimiento [LeC+12; GBC16; Nov17; Nøk16].

[2] https://numpy.org/
[3] https://es.python.org/

Otra de las características de este algoritmo para el cálculo del gradiente es que los conceptos en los que se basa son simples: optimización, diferenciación, derivadas parciales y regla de la cadena. Lo cual lo convierte a priori en objeto de estudio accesible. En la práctica, los cálculos que se realizan en esta estrategia se implementan a través de la diferenciación automática, que es una técnica más general que extiende a BP y se usa para el cómputo de derivadas de funciones numéricas de una manera eficiente y precisa [Bay+15].

1.1. Motivación

Tenemos pues que el aprendizaje profundo es el paradigma del aprendizaje automático que mejores resultados obtiene actualmente y más desarrollo e investigación está concentrando, y que basa el entrenamiento (una de las partes fundamentales que determinan el rendimiento del modelo, además de su arquitectura) de los modelos casi por completo en el algoritmo de descenso de gradiente, ya que es el que mejores resultados de generalización y rendimiento ofrece. Éste a su vez depende casi enteramente del algoritmo de BP para calcular el gradiente, ya que aunque existen otras alternativas no son realmente viables. Tanto es así que es muy común la confusión entre éste algoritmo y el de gradiente descentente, que se suelen tomar por la misma cosa. Queda así clara la importancia que tiene BP en el campo del aprendizaje profundo y por extensión también al aprendizaje automático. También conviene destacar la cantidad de veces que se utiliza ésta técnica durante el entrenamiento de un modelo. Cada vez que se actualizan los pesos debemos calcular el gradiente, y teniendo en cuenta la duración de los entrenamientos de los modelos más grandes (con mayor número de parámetros) este algoritmo puede ser usado miles de veces durante un entrenamiento.

Su eficiencia, escalabilidad y flexibilidad lo han convertido en la opción por defecto para el entrenamiento basado en gradiente descendente para modelos de aprendizaje profundo, sin embargo no hay que olvidar que no se trata de una tarea sencilla: la obtención de un mínimo global y la verificación, dado un punto, de que es un mínimo global, se trata de un problema NP-Completo generalmente [MK87], por lo que se buscan estrategias aproximadas capaces de obtener buenas soluciones en tiempos razonables. Uno de los problemas abiertos en el aprendizaje profundo y en el que influye directamente BP es la reducción computacional del entrenamiento: si se ajustan los pesos en un modelo con un número muy alto de parámetros y usando un conjunto de entrenamiento muy grande (que es una tendencia reciente en aprendizaje profundo), los recursos computacionales pueden resultar insuficientes incluso para las grandes compañías, pudiendo requerir de meses para el entrenamiento. Por lo que se necesitan algoritmos más escalables y eficientes para afrontarlo [Dea+12].

Por ello resulta esencial, mientras no existan alternativas viables, poder

ofrecer modificaciones a este algoritmo para mejorar sus cualidades. Atendiendo a la cantidad de uso y su extensión en el campo, una pequeña mejora tendría un alcance enorme. Sin embargo esta línea de investigación no es muy extensa ya que principalmente se buscan alternativas en lugar de mejoras, pudiendo deberse principalmente a que a priori puede parecer una técnica muy enrevesada y compleja. Veremos en el desarrollo de esta parte que esto no es cierto, y que los principios en los que se basa son muy simples. Es clave comprender su base teórica, funcionamiento e implementación práctica para poder proponer mejoras.

1.2. Objetivos

El objetivo principal de esta parte es realizar una investigación sobre los algoritmos de descenso de gradiente y *backpropagation*, proporcionando una visión detallada acerca de los mismos y su implementación. Para ello se divide este objetivo en varios:

1. Definir de manera detallada la base teórica y funcionamiento del algoritmo de descenso de gradiente atendiendo a elementos clave como su convergencia. Enunciar y demostrar resultados teóricos sobre la convergencia y analizar los problemas de ésta.

2. Explorar el uso de BP para el cálculo del gradiente, analizando su implementación a través de la diferenciación automática.

2. Fundamentos previos

A continuación se definirán los conceptos básicos necesarios con los que se trabajará durante el desarrollo de esta parte. Se tratarán los elementos que se usan en el algoritmo de gradiente descendente y BP. Se presenta únicamente el material estrictamente necesario para comprender el trabajo. Se ha usado para la elaboración de esta sección los apuntes en línea del profesor de la UGR Rafael Payá Albert en su curso de Análisis Matemático I [4]. Salvo otras especificaciones, el material de consulta para el desarrollo de esta parte matemática ha sido el curso en línea de Ciencias de Computación de la universidad Bristish Columbia [5] y los libros Probabilistic Machine Learning [Mur22] y Deep Learning [GBC16].

2.1. Cálculo diferencial

Los algoritmos de gradiente descendente y BP se basan principalmente en el cálculo diferencial, y el hecho de que no usen herramientas matemáticas demasiado complejas resulta precisamente una de sus virtudes, ya que

[4] `https://www.ugr.es/~rpaya/docencia.htm#Analisis`
[5] `https://www.cs.ubc.ca/~schmidtm/Courses/5XX-S22/`

gracias a la abstracción y a un diseño ingenioso consiguen obtener grandes resultados a partir de operaciones relativamente sencillas. Empezamos con los conceptos más elementales que subyacen durante todo el trabajo.

En lo que sigue se fijan los abiertos $X \subseteq \mathbb{R}^n$, $Y \subseteq \mathbb{R}^m$ y las funciones $f : X \to Y$ y $h : X \to \mathbb{R}$.

Definición 2.1 (Función diferenciable) *f es diferenciable en el punto $a \in X$ si existe una aplicación lineal y continua $T \in L(X, Y)$ que verifica:*

$$Df(a) = \lim_{x \to a} \frac{\|f(x) - f(a) - T(x-a)\|}{\|x - a\|} = 0$$

Decimos que f es diferenciable si es diferenciable en todo punto del interior de su dominio.

Definición 2.2 (Derivada parcial en un campo escalar) *La derivada parcial de g con respecto a la k-ésima variable x_k en el punto $a = (a_1, \ldots, a_n) \in X$ se define como*

$$\frac{\partial f}{\partial x_k}(a) = \lim_{h \to 0} \frac{f(a_1, \ldots, a_{k-1}, a_k + h, a_{k+1}, \ldots, a_n) - f(a_1, \ldots, a_n)}{h}$$

si existe el límite.

Notamos por $h = (h_1, h_2, \ldots, h_m)$ indicando las m componentes de h que es un campo escalar definido en X, siendo $h_j = \pi_j \circ f$.

Definición 2.3 (Derivada parcial) *f es parcialmente derivable con respecto a la k-ésima variable x_k en $a = (a_1, \ldots, a_n) \in X$ si, y sólo si, lo es f_j $\forall j \in I_m$, en tal caso,*

$$\frac{\partial f}{\partial x_k}(a) = \left(\frac{\partial f_1}{\partial x_k}(a), \ldots, \frac{\partial f_m}{\partial x_k}(a) \right) \in \mathbb{R}^m$$

f es parcialmente derivable en a si, y sólo si, lo es respecto de todas sus variables.

Definimos ahora los elementos clave del algoritmo de entrenamiento: el vector gradiente y la matriz jacobiana. En el algoritmo de descenso de gradiente, lo que se pretende calcular tal como indica el nombre es el vector gradiente, ya que la función de error de los modelos siempre nos devuelve un escalar, es decir que la dimensión de la imagen es 1, y la dimensión de la entrada será el número de parámetros del modelo (número de elementos que tendrá el vector gradiente). Sin embargo las matrices jacobianas también juegan un papel fundamental ya que para calcular ese vector gradiente, el algoritmo de BP necesita realizar cálculos intermedios, que son las matrices jacobianas asociadas a la salida de las capas ocultas (que tienen mayor dimensionalidad) bien con respecto a los parámetros de la capa o bien con respecto al error de la predicción del modelo. En lo que sigue fijamos $x \in X$.

Definición 2.4 (Vector gradiente) *Cuando h es parcialmente derivable en x, el gradiente de h en x es el vector $\nabla h(x) \in X$ dado por*

$$\nabla h(x) = \left(\frac{\partial h}{\partial x_1}(x), \frac{\partial h}{\partial x_2}(x), \ldots, \frac{\partial h}{\partial x_n}(x) \right).$$

Definición 2.5 (Matriz jacobiana) *Si f es diferenciable en x, la matriz jacobiana es la matriz de la aplicación lineal $Df \in L(X,Y)$ y se escribe como J_f. Viene dada por:*

$$\begin{aligned} J_f(x) &= \begin{pmatrix} \frac{\partial f_1}{\partial x_1}(x) & \frac{\partial f_1}{\partial x_2}(x) & \cdots & \frac{\partial f_1}{\partial x_n}(x) \\ \frac{\partial f_2}{\partial x_1}(x) & \frac{\partial f_2}{\partial x_2}(x) & \cdots & \frac{\partial f_2}{\partial x_n}(x) \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1}(x) & \frac{\partial f_m}{\partial x_2}(x) & \cdots & \frac{\partial f_m}{\partial x_n}(x) \end{pmatrix} \\ &= \begin{pmatrix} \nabla f_1(x)^T \\ \vdots \\ \nabla f_m(x)^T \end{pmatrix} = \left(\frac{\partial f}{\partial x_1}(x), \ldots, \frac{\partial f}{\partial x_n}(x) \right) \end{aligned}$$

Si f es de clase C^2 (derivable dos veces con sus derivadas continuas) y derivamos el gradiente obtenemos una matriz cuadrada simétrica con derivadas parciales de segundo orden, a la que llamamos matriz Hessiana.

Definición 2.6 (Matriz Hessiana) *Definimos la matriz Hessiana de h en x como*

$$\nabla^2 h(x) = \begin{pmatrix} \frac{\partial^2 h}{\partial x_1^2}(x) & \frac{\partial^2 h}{\partial x_1 \partial x_2}(x) & \cdots & \frac{\partial^2 h}{\partial x_1 \partial x_n}(x) \\ \frac{\partial^2 h}{\partial x_2 \partial x_1}(x) & \frac{\partial^2 h}{\partial x_2^2}(x) & \cdots & \frac{\partial^2 h}{\partial x_2 \partial x_n}(x) \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 h}{\partial x_n \partial x_1}(x) & \frac{\partial^2 h}{\partial x_n \partial x_1}(x) & \cdots & \frac{\partial^2 h}{\partial x_n^2}(x) \end{pmatrix}$$

Este es un concepto muy importante del cálculo multivariable y la optimización. Cuando hablemos del gradiente descendente, especialmente de la convergencia, usaremos algunas de sus propiedades, como por ejemplo la aproximación cuadrática para desplazamientos pequeños: $h(x + \Delta x) \approx h(x) + \nabla h(x)^T \Delta x + \frac{1}{2} \Delta x^T \nabla^2 h(x) \Delta x$.

Se presenta a continuación una de las reglas más útiles para el cálculo de diferenciales, que afirma que la composición de aplicaciones preserva la diferenciabilidad. Será parte clave en el desarrollo próximo ya que a los modelos de aprendizaje automático basados en capas podemos describirlos como una función que se descompone en una función por cada capa, por tanto será una herramienta que usaremos continuamente para calcular estas matrices jacobianas y gradientes.

Teorema 2.1 (Regla de la cadena) *Sea $Z \subseteq \mathbb{R}^p$ un abierto y $g : Y \to Z$. Entonces si f es diferenciable en x y g es diferenciable en $y = f(x)$ se tiene que $g \circ f$ es diferenciable en x con*

$$D(g \circ f)(x) = Dg(y) \circ Df(x) = Dg(f(x)) \circ f(x)$$

Si $f \in D(X, Y)$ y $g \in D(Y, Z)$, entonces $g \circ f \in D(X, Z)$.

En ocasiones en algunos modelos tenemos que lidiar con funciones que no son diferenciables en un punto, y para poder manejarlas extenderemos el concepto de diferenciabilidad a lo que llamaremos subdiferenciabilidad. Esto se expondrá más adelante ya que son conceptos que no se han explorado a lo largo del grado de matemáticas.

El último concepto, que también resulta de gran importancia en los resultados teóricos sobre la convergencia del gradiente descendente, es el de la condición de Lipschitz, en concreto aplicada al gradiente. No forma parte del cálculo diferencial ya que la condición de Lipschitz no requiere diferenciabilidad, pero lo usaremos en éste ámbito ya que la condición que nos interesa usar está aplicada al gradiente.

Definición 2.7 (Función Lipschitziana) *El campo escalar h es lipschitziano si existe una constante $M \in \mathbb{R}_0^+$ que verifica:*

$$\|h(x) - h(y)\| \leq M\|x - y\| \qquad \forall x, y \in X.$$

La definición nos dice de manera intuitiva que el gradiente de la función no puede cambiar a una velocidad arbitraria. Decimos que la función f tiene gradiente lipschitziano si la condición anterior se aplica a su gradiente, es decir:

$$\|\nabla h(x) - \nabla h(y)\| \leq M\|x - y\| \qquad \forall x, y \in X.$$

La mínima constante $M_0 = L$ que verifica las desigualdades anterior es denominada la constante de Lipschitz de f y viene definida por

$$L = sup\left\{ \frac{\|h(x) - h(y)\|}{\|x - y\|} : x, y \in X, x \neq y \right\}.$$

Para las funciones de clase C^2, es decir las que son diferenciables al menos dos veces con su derivada continua, una equivalencia a que el gradiente de h sea lipschitziano es que $\nabla^2 h(x) \preceq LI \quad \forall x \in X$, esto lo usaremos luego en la demostración 3.2. En esta expresión L es la constante de Lipschitz para el gradiente de h e I es la matriz identidad. El símbolo $\preceq$ denota una desigualdad matricial en términos de semidefinición positiva, es decir que $LI - \nabla^2 h(x)$ es una matriz semidefinida positiva. Equivalentemente para cualquier vector z se tiene $z^T \left(LI - \nabla^2 h(x)\right) z \geq 0$.

2.2. Teoría de la probabilidad

Ahora vamos a introducir los conceptos necesarios para el desarrollo teórico relacionado con la versión estocástica del algoritmo de gradiente descendente. Como su nombre indica, se trata de un proceso estocástico. Desarrollamos las herramientas necesarias para llegar a las definiciones de esperanza condicionada y convergencia casi segura, necesarias para entender las martingalas. Empezamos con las definiciones básicas:

Definición 2.8 (Espacio de probabilidad) *Un espacio de probabilidad es una terna* $(\Omega, \mathcal{A}, P)$ *donde:*

1. Ω *es un conjunto arbitrario.*
2. $\mathcal{A} \subseteq \mathcal{P}(\Omega)$ *tiene estructura de* σ*-álgebra, es decir, verifica que*
   - $\mathcal{A} \neq \emptyset \quad y \quad \Omega \in \mathcal{A}$.
   - $A \in \mathcal{A} \Rightarrow A^c = \Omega \backslash A \in \mathcal{A}$ *(cerrada bajo complementarios).*
   - $A_n \in \mathcal{A} \forall n \in \mathbb{N} \Rightarrow \underset{n\in\mathbb{N}}{\cup} A_n \in \mathcal{A}$ *(cerrada bajo uniones numerables).*
3. $P : \mathcal{A} \to [0, 1]$ *es una función de probabilidad, es decir,*
   - $P(A) \geq, \quad \forall A \in \mathcal{A}$.
   - $P(\Omega) = 1$.
   - *Para cualquier sucesión* $A_{nn\in\mathbb{N}} \subseteq \mathcal{A}$ *de sucesos disjuntos se tiene* $P\left(\underset{n\in\mathbb{N}}{\cup} A_n\right) = \sum_{n\in\mathbb{N}} P(A_n)$.

Decimos que dos sucesos son disjuntos si solo puede ocurrir uno de los dos, es decir, $P(A \cap B) = 0$. Para continuar con las definiciones necesarias, primero debemos asomarnos a la teoría de la medida para poder usar el concepto de función medible, para el que necesitamos con anterioridad el de espacio medible.

Definición 2.9 (Espacio medible) *Un espacio medible es un par* $(\Omega, \mathcal{A})$*, donde* Ω *es un conjunto arbitrario y* $\mathcal{A} \subseteq \mathcal{P}(\Omega)$ *tiene estructura de* σ*-álgebra.*

Vemos por tanto que en la definición 2.8 tenemos un espacio medible. Esto nos servirá para definir las funciones medibles y seguidamente las variables aleatorias.

Definición 2.10 (Función medible) *Una función medible (unidimensional) sobre un espacio medible* $(\Omega, \mathcal{A})$ *es una función* $f : \Omega \to \mathbb{R}$ *que verifica*

$$f^{-1}(B) \in \mathcal{A}, \quad \forall B \in \mathcal{B}$$

o, equivalentemente, que $f^{-1}(\mathcal{B}) \subseteq \mathcal{A}$*, siendo*$\mathcal{B}$ *la* σ*-álgebra de Borel en* $\mathbb{R}$*, es decir, la mínima* σ*-álgebra que contiene a todos los intervalos.*

Definición 2.11 (Variable aleatoria) *Una variable aleatoria sobre un espacio de probabilidad* $(\Omega, \mathcal{A}, \mathcal{P})$ *es una función medible* $X : \Omega \to \mathbb{R}$.

Definición 2.12 (Función de distribución de una variable aleatoria) *Una función* $F_X : \mathbb{R} \to [0, 1]$, *definida por*

$$F_X(x) = PX \leq x) = P_X((-\infty, x]) = P(X \in (-\infty, x]), \quad \forall x \in \mathbb{R},$$

se dice que es función de distribución de la variable aleatoria X.

Una variable aleatoria puede ser discreta o continua. Decimos que es discreta si existe un conjunto numerable $E_X \subseteq \mathbb{R}$, tal que $P_X(X \in E_X) = 1$. Para el objetivo que nos interesa sólo vamos a necesitar del tipo discreto, por lo que nos ceñiremos a ellas y, en caso de no especificar, nos estaremos refiriendo a una variable aleatoria discreta. Pasamos a presentar el último concepto intermedio antes de llegar a la esperanza matemática.

Definición 2.13 (Función masa de probabilidad) *Una variable aleatoria discreta* $X : (\Omega, \mathcal{A}, P) \to (\mathbb{R}, \mathcal{B}, P_X)$ *tiene como función de masa de probabilidad*

$$p_X : E_X \to [0, 1], \quad p_X(x) = P(X = x), \quad \forall x \in E_x.$$

Tenemos entonces que $\sum_{x \in E_X} p_X(x) = 1$.

Definimos ahora la esperanza matemática, que además de servirnos para la posterior definición de martingala, la usaremos en la sección 4.4.2 para intentar controlar la explosión y desvanecimiento del gradiente en BP a través de la inicialización de los pesos.

Definición 2.14 (Esperanza matemática) *Si para la variable aleatoria* X *existe* $\sum_{x \in E_X} |x| p_X(x) < \infty$, *entonces se define la esperanza matemática de* X *como:*

$$E[X] = \sum_{x \in E_X} x p_X(x) = \sum_{x \in E_X} x P(X = x).$$

Dadas dos variables aleatorias x e Y, vamos a ver las dos propiedades más importantes de la esperanza matemática:

- Linealidad: $E[aX + bY] = aE[x] + bE[Y], \quad \forall a, b \in \mathbb{R}$.
- Conservación del orden: $X \leq Y \Rightarrow E[X] \leq E[Y]$.

Ahora vamos a presentar las distribuciones condicionadas para, seguidamente, definir finalmente la esperanza condicionada. OBTENIDO DE https://www.ugr.es/ proman/PyE/Condicionadas.pdf

Definición 2.15 (Distribución condicionada) *Sea $X = (X_1, \ldots, X_n)$ un vector de variables aleatorias discretas sobre el mismo espacio de probabilidad. Sea X_i una componente arbitraria y $x_i \in \mathbb{R}$ tal que $P(X_i = x_i) > 0$. Se define la distribución condicionada de $(X_1, \ldots, X_{i-1}, X_{i+1}, \ldots, X_n)$ a $X_i = x_i$ como la determinada por la función masa de probabilidad*

$$P(X_1 = x_1, \ldots, X_{i-1} = x_{i-1}, X_{i+1} = x_{i+1}, \ldots, X_n = x_n | X_i = x_i) = \frac{P(X_1 = x_1, \ldots, X_{i-1} = x_{i-1}, X_i = x_i, X_{i+1} = x_{i+1}, \ldots, X_n = x_n)}{P(X_i = x_i)}$$
$$\forall (x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n)/(x_1, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_n) \in E.$$

Vamos a definir la esperanza condicionada de una variable a otra (caso bidimensional), pero para más variables sería un razonamiento análogo.

Definición 2.16 (Esperanza condicionada) *Sean X e Y variables aleatorias discretas sobre el mismo espacio de probabilidad. Se define la esperanza matemática condicionada de X a Y como la variable aleatoria, que se denota $E[X|Y]$, que toma el valor $E[X|Y = y]$ cuando $Y = y$, siendo*

$$E[X|Y = y] = \sum_{x \in E_X} xP(X = x|Y = y) \quad si\ P(Y = y) > 0. \tag{1}$$

Ahora definimos el concepto de casi seguridad, necesario también para el mismo desarrollo teórico. Se dice que un suceso es casi seguro cuando la probabilidad de que ocurra es uno. Por tanto dada una sucesión de variables aleatorias $\{X_n\}_{n \in \mathbb{N}}$ definidas sobre un mismo espacio de probabilidad, y X es otra variable aleatoria definida sobre el mismo espacio, se dice que dicha solución converge casi seguramente a X si

$$P\left(\lim_{n \to +\infty} X_n = X\right) = 1$$

Usamos la notación $X_n \overset{c.s}{\to} X$. Como último concepto definimos proceso estocástico, que nos servirá para tratar con las versiones estocásticas del gradiente descendente. Es un proceso aleatorio que evoluciona con el tiempo. Más concretamente, un proceso estocástico es una colección de variables aleatorias X_t indexadas por el tiempo. Si el tiempo es un subconjunto de los enteros no negativos $\{0, 1, 2, \ldots\}$ entonces llamaremos al proceso discreto, mientras que si es un subconjunto de $[0, \infty)$ entonces trataremos con un proceso estocástico continuo. Las variables aleatorias X_t toman valores en un conjunto que llamamos espacio de estados. Este espacio de estados puede ser a su vez discreto, si es un conjunto finito o infinito numerable; o continuo, por ejemplo el conjunto de los números reales $\mathbb{R}$ o un espacio d-dimensional $\mathbb{R}^d$.

3. Gradiente Descendente

Se trata de un algoritmo de aprendizaje iterativo clásico, basado en el método de optimización para funciones lineales de Cauchy. Haskell Curry lo estudió por primera vez para optimización no lineal en 1944 [Cur44], siendo ampliamente usado a partir de las décadas de 1950-1960. Actualmente se trata de la estrategia de entrenamiento de modelos más ampliamente usada, especialmente en los modelos de aprendizaje profundo, siendo la estrategia que mejores resultados consigue en cuanto a capacidad de generalización de los modelos y eficiencia computacional gracias a su aplicación a través del algoritmo de BP. Sin embargo a nivel práctico no se usa en su versión original, sino que a lo largo del tiempo han ido surgiendo numerosas modificaciones con el objetivo de mejorar el algoritmo en diversos ámbitos: aumento de la estabilidad y la velocidad de convergencia, reducción computacional del entrenamiento, capacidad de evitar mínimos locales, etc. Estos métodos modificados del original se conocen como optimizadores. La literatura en este sentido es extensa, es claro que el gradiente descendente sigue siendo la mejor estrategia de optimización de parámetros de un modelo de forma general [Mar+20], aunque la elección del algoritmo de optimización concreto y de su ajuste depende del problema concreto que estemos tratando y generalmente se realiza de manera experimental.

Debemos ver que el entrenamiento de los modelos, está intrínsecamente ligado a la optimización, en concreto a la minimización de la función de coste C. Este no es un problema sencillo, y como se ha mencionado antes se trata de un problema NP-Completo, por tanto de existir algoritmos exactos estos requieren demasiado coste computacional como para utilizarlos en la práctica, por lo que se buscan estrategias aproximadas como el descenso de gradiente para obtener buenas soluciones en un tiempo asequible.

Otros factores a tener en cuenta son la necesidad de escapar de óptimos locales, aún no conociendo de manera explícita la función de error; y la generalización: no es importante únicamente obtener un error bajo en el entrenamiento sino que se mantenga cuando usamos datos de entrada nuevos, ya que nuestro objetivo es ser capaces de encontrar patrones que podamos aplicar en situaciones nuevas y no ajustar el modelo a unos datos dados.

3.1. Gradiente descendente de Cauchy

Procedemos a describir el método original de descenso de gradiente, propuesto en 1847 por Augustin-Louis Cauchy [Cau09]. Es una versión más primitiva y limitada que sus desarrollos posteriores pero que nos permite obtener de forma más sencilla una visión de su funcionamiento.

Fijamos $f : \mathbb{R}^n \to \mathbb{R}_0^+$ una función continua que no toma valores negativos. Sea $x = (x_1, \ldots, x_n) \in \mathbb{R}^n$. Si queremos encontrar los valores de $x_1, \ldots, x_n$ que verifican $f(x) = 0$, que suponemos que existen, bastará con

hacer decrecer indefinidamente los valores de la función f hasta que sean muy cercanos a 0.

Fijamos ahora unos valores concretos $x_0 \in \mathbb{R}^n$, $u = f(x_0)$, $Du = (D_{x_1}u, D_{x_2}u, \ldots, D_{x_n}u)$ y $\epsilon > 0$ con $\epsilon \in \mathbb{R}^n$. Si tomamos $x_0' = x_0 + \epsilon$ tendremos:

$$f(x_0') = f(x_0 + \epsilon) = u + \epsilon Du$$

Sea ahora $\eta > 0$, tomando $\epsilon = -\eta Du$ con la fórmula anterior tenemos:

$$f(x_0') = f(x_0 + \epsilon) = u - \eta \sum_{i=1}^{n} (D_{x_i}u)^2$$

Por tanto hemos obtenido un decremento en el valor de la función f modificando los valores de sus variables en sentido contrario al gradiente, para η suficientemente pequeño. El objetivo de la estrategia es repetir esta operación hasta que se desvanezca el valor de la función f.

3.2. Gradiente descendente en el entrenamiento de modelos

En el caso del entrenamiento de modelos la función que debemos minimizar es la función de coste C, que efectivamente es continua por ser composición de funciones continuas, como se verá más adelante. Esta función no toma valores negativos. Como no podemos realizar un cálculo continuo para comprobar con qué valores de η la función decrece, lo hacemos de manera iterativa, y a este η lo llamamos ratio de aprendizaje o más comúnmente *learning rate*.

Si $C(W)$ es la función de coste del modelo y W representa los parámetros del modelo, entonces la regla iterativa de actualización de los pesos en la estrategia del descenso del gradiente es la siguiente:

$$W_{t+1} = W_t - \eta \nabla C(W) \tag{2}$$

En su descripción original, el gradiente se calcula usando todos los datos de entrenamiento, pero en versiones posteriores se propone dividir el conjunto de entrenamiento en varios subconjuntos disjuntos, denominados lotes. Cada vez que se calcula el gradiente se actualizan los pesos del modelo, y denominamos a esto una iteración. Cada vez que se usan todos los datos de entrenamiento para calcular el gradiente, ya sea tras una sola iteración usando todo el conjunto de entrenamiento o varias si dividimos en lotes, lo denominamos época.

3.2.1. Estrategias de gradiente descendente

En base a los lotes en que dividamos el conjunto de entrenamiento tenemos varios tipos de gradiente descendente [GBC16].

- **Batch Gradient Descent** (BGD): tenemos un único lote, cada iteración se corresponde con una época. Calculamos el gradiente usando todo el conjunto de entrenamiento. Esto ofrece un comportamiento mejor estudiado a nivel teórico, con más resultados demostrados; pero aumenta mucho el coste computacional del entrenamiento hasta el punto que lo vuelve demasiado lento para ser usado en la práctica.
- **Stochastic Gradient Descent** (SGD): Actualiza los pesos calculando el gradiente con sólo un elemento del conjunto de entrenamiento. Cada época tiene tantas iteraciones como número de elementos haya en el conjunto de entrenamiento. Esta estrategia introduce ruido en el entrenamiento ya que el gradiente se calcula de una manera aproximada, aunque esto tiene un efecto positivo ya que al provocar más irregularidad en la trayectoria de convergencia es más probable poder escapar mínimos locales. Además es más eficiente computacionalmente que el anterior y converge más rápido en la práctica.
- **Mini-Batch Gradient Descent** (MBGD): Se divide el conjunto de entrenamiento en M lotes disjuntos de tamaño fijo, y se calcula el gradiente con cada lote, por lo que habrá M iteraciones en cada época. Se consigue una aproximación del gradiente con menos error al usar más datos para su cálculo y además se siguen manteniendo las propiedades que veíamos en la anterior estrategia. Es más eficiente que la anterior al conllevar menos actualizaciones de pesos. Es prácticamente la única estrategia utilizada en la realidad ya que ofrece la mayor eficiencia computacional, estabilidad y rapidez en la convergencia.

En estos dos últimos casos, el conjunto de entrenamiento no permanece fijo, por lo que la regla de actualización de los pesos que hemos visto en 2 quedaría de la siguiente manera:

$$W_{t+1} = W_t - \eta \nabla C(X_{t+1}, W_t) \tag{3}$$

Aunque la política para computar el gradiente sea distinta en estos 3 tipos, los englobaremos dentro de lo que denominaremos el algoritmo de gradiente descendente original, ya que existen varias modificaciones del algoritmo que aportan mejoras a través de modificar la regla de actualización de los pesos y no solo la cantidad de datos con la que se aproxima el gradiente.

3.2.2. *Learning rate*

El elemento η que observamos en la ecuación 2 del gradiente descendente se denomina *learning rate* y lo usamos para controlar la convergencia reduciendo el efecto de la magnitud del gradiente en la actualización de los parámetros. Este valor es positivo y situado en la práctica alrededor de 0.01

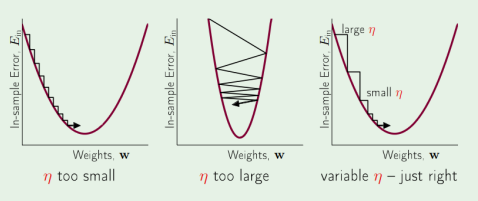

Figura 2: Visualización de cómo afecta el *learning rate* según su adecuación al problema. Imagen obtenida del curso de Caltech [6], tema 9 diapositiva 21

y 0.001 usualmente, aunque para su elección conviene realizar un análisis teórico previo o realizar pruebas prácticas (mucho más común) para elegir un valor adecuado. Este tipo de parámetros, que no son parte del modelo sino del algoritmo de aprendizaje, se denominan hiperparámetros. Dependiendo del tipo de algoritmo o modificación del mismo que usemos habrá diferentes hiperparámetros, siendo el *learning rate* el más importante de manera general, ya que de su valor dependerá la convergencia del algoritmo, pudiendo hacer que converja demasiado lento o que directamente diverja, como podemos observar en la figura 2 o en resultados sobre la convergencia en la sección 3.4.

En cuanto a la selección de los hiperparámetros, no se enfoca como un problema donde se busque el óptimo de estos valores ya que la mayoría no son tan decisivos en la convergencia como el *learning rate*, y se ofrecen valores teóricos en sus papers de presentación que funcionan bien en casos generales. Si bien la convergencia es sensible a los valores iniciales de estos hiperparámetros que se tratan de optimizar a nivel experimental a través del ensayo y error, aunque no se realiza una búsqueda exhaustiva, invirtiéndose muchos más recursos computacionales en el entrenamiento.

Una táctica habitual es usar una política de *learning rate* que decrezca conforme avanza el entrenamiento, de manera que el algoritmo avance con pasos más grandes cuando aún está lejos de un óptimo, con un objetivo explorador, y con pasos más pequeños cuando se va acercando, con un objetivo explotador, procurando una convergencia más estable. [GBC16]. Otro enfoque común es tener un vector de *learning rate* en lugar de un solo escalar, teniendo un valor para cada peso del modelo.

3.3. Subgradientes

Con el objetivo central de calcular el gradiente es lógico pensar que necesitamos ciertas condiciones de diferenciabilidad, aunque sean mínimas, para poder calcular el gradiente que necesitamos. Sin embargo vamos a ver que no necesitamos estrictamente que las funciones sean diferenciables, sino que extendemos al concepto de subdiferenciabilidad.

Podemos pensar en un modelo como una composición de la suma y pro-

ducto de operaciones lineales con operaciones no lineales (funciones de activación), y componiendo ésta con la función de coste del modelo obtendríamos la función $f : X \times \Omega \times Y \to \mathbb{R}^+$, que recibe los pesos del modelo, los datos de entrada y sus etiquetas correctas para proporcionar el error del modelo. Esta es la función que necesitaríamos que fuera diferenciable. Las operaciones lineales preservan la diferenciabilidad, y la composición de funciones diferenciables es diferenciable por lo que si la función de pérdida y las funciones de activación son diferenciables, no tendremos ningún problema a la hora de calcular el gradiente.

Las funciones de coste son diferenciables de manera general, y la más común para problemas de clasificación es *CrossEntropyLoss*, mientras que para regresión son comunes el error cuadrático medio y el error absoluto medio.

- **ECM:** $\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y_i})^2$
- **EAM:** $\frac{1}{N}\sum_{i=1}^{N}|y_i - \hat{y_i}|$
- ***CrossEntropyLoss*:** $-\sum_c \hat{y}_{i,c} log(\frac{e^{y_{i,c}}}{\sum_{c'=1}^{C} e^{y_{i,c'}}})$

En regresión $\hat{y_i}$ es el valor real e y_i es el predicho por el modelo para el dato i que será un real en ambos casos. En clasificación $\hat{y}_{i,c}$ es la etiqueta real del dato i para la clase c, que valdrá 1 en caso de que el dato pertenece a la clase c y 0 en caso contrario, e $y_{i,c} \in [0, 1]$ representa la probabilidad predicha por el modelo de que el dato i pertenezca a la clase c. Finalmente N es el número de datos y C el número de clases.

Hasta el año 2010, las funciones de activación más comunes para las capas ocultas eran la función sigmoide y la tangente hiperbólica. Estas funciones son diferenciables por lo que su uso no suponía ningún problema en la aplicación del descenso de gradiente. Sobre ese año se empezó a popularizar la función de activación ReLU (Rectified Linear Unit), gracias a su simplicidad, reducción de coste computacional y su aparición en modelos ganadores de competiciones de ImageNet como AlexNet en 2012. Desde entonces esta función, junto a algunas de sus variantes que aparecen en la figura 3 son ampliamente usadas y con buenos resultados. Sin embargo salta a la vista que esta función no es diferenciable.

Vamos a presentar entonces el concepto de subgradiente junto con algunas de sus propiedades, obtenidas de [Bub15], para ver que será una extensión del gradiente que nos permitirá usar el método de gradiente descendente con funciones que no sean diferenciables en algunos puntos pero que sí sean subdiferenciables.

Definición 3.1 (Subgradiente) *Sea $A \subset \mathbb{R}^n$ y $f : A \to \mathbb{R}$, $g \in \mathbb{R}^n$ es un subgradiente de f en $a \in A$ si existe un entorno de a U_a tal que $\forall y \in U_a$ se tiene:*

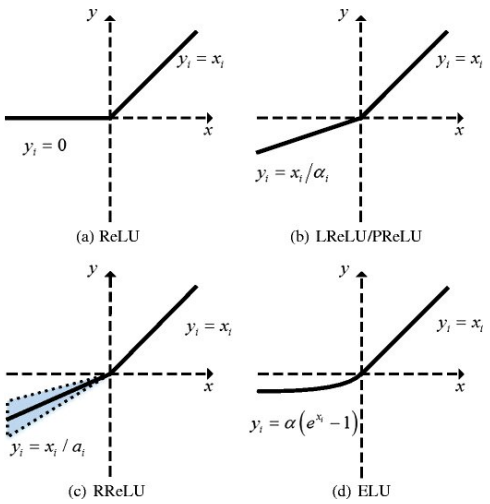

Figura 3: Función ReLU y algunas de sus variantes más usadas como funciones de activación.[7]

$$f(a) - f(y) \leq g^T(a - y)$$

El conjunto de los subgradientes de f en a se denota por $\partial f(a)$. Si existe el subgradiente de f en a, decimos que f es subdiferenciable en a.

Necesitamos también un comportamiento similar al de las funciones diferenciables, en particular necesitamos que las funciones subdiferenciables se preserven a través de las operaciones de suma, multiplicación por escalares y composición.

1. **Multiplicación escalar no negativa**: $\partial(af) = a \cdot \partial f, a \geq 0$

   Por definición g es un subgradiente de f en x_0 si:

   $$f(x) \geq f(x_0) + g^T(x - x_0), \quad \forall x \in dom(f).$$

   Multiplicando la desigualdad por $c \geq 0$:

   $$cf(x) \geq cf(x_0) + cg^T(x - x_0), \quad \forall x \in dom(f).$$

   Por tanto cg es un subgradiente de $h(x) = cf(x)$ en x_0.

2. **Suma**: $\partial(f_1 + f_2)(x) = \partial f_1(x) + \partial f_2(x)$

   Sea g_1 un subgradiente de f_1 y g_2 un subgradiente de f_2, considerando el punto $x_0 \in dom(f_1) \cap dom(f_2)$, por definición tenemos:

   $$f_1(x) \geq f_1(x_0) + g_1^T(x - x_0), \quad \forall x \in U_{x_0},$$

$$f_2(x) \geq g_2(x_0) + g_2^T(x - x_0), \quad \forall x \in U_{x_0}.$$

Sumamos las dos desigualdades para obtener que $g_1 + g_2$ es un subgradiente de $(f_1 + f_2)(x_0)$:

$$f_1(x) + f_2(x) \geq f_1(x_0) + f_2(x_0) + (g_1 + g_2)^T (x - x_0).$$

3. **Composición afín**: Si $h(x) = f(Ax + b) \Rightarrow \partial h(x) = A^T \partial f(Ax + b)$.

   Tenemos que g es un subgradiente de f en y_0:

$$f(y) \geq f(y_0) + g^T(y - y_0), \quad \forall y \in U_{y_0}.$$

   Tomamos $y = Ax + b$ y por tanto $y_0 = Ax_0 + b$. Sustituyendo:

$$f(Ax + b) \geq f(Ax_0 + b) + g^T(Ax + b - (Ax_0 + b)),$$

$$h(x) = f(Ax + b) \geq h(x_0) + g^T A(x - x_0).$$

   Por tanto $A^T g$ es un subgradiente de $h(x) = f(Ax + b)$ en x_0.

Tenemos que comprobar que el subgradiente extiende al gradiente, es decir, que cuando existe gradiente entonces existe un único subgradiente y coincide con él. Además hay funciones que no son diferenciables pero sí subdiferenciables. Esto último se hace evidente con el ejemplo 3.1 de la función ReLU. Vamos a demostrar por tanto que si $f : X \subseteq \mathbb{R}^n \to \mathbb{R}^m$ es diferenciable en el punto $x \in X$ entonces $\partial f(x) = \{\nabla f(x)\}$.

Como f es diferenciable en x, existe un entorno U_x de x donde el gradiente satiface

$$f(y) = f(x) + \nabla f(x)^T(y - x) + o(\|y - x\|) \quad \forall y \in U_x.$$

Por tanto tenemos

$$f(y) \geq f(x) + \nabla f(x)^T(y - x) \quad \forall y \in U_x.$$

Es decir que el gradiente de f en x es también un subgradiente de f en x. Tenemos que $\nabla f(x) \in \partial f(x)$, nos queda demostrar que es único. Para ello vamos a suponer que existe otro subgradiente de f en x, $g \in \partial f(x)$. Sea $u = x + tw$, definimos la función

$$\phi(t) = f(u) - f(x) - g^T(u - x) = f(x + tw) - f(x) - g^T(tw) \geq 0$$

donde se ha usado que $g \in \partial f(x)$ para ver que es no negativa. Derivamos la función para obtener $\phi'(t) = \nabla f(x)^T w - g^T w$. Vemos que para $t = 0$ se tiene que $\phi(0) = 0$, con lo que hay un mínimo en ese punto. Tenemos por tanto que $\phi'(0) = 0$ o equivalentemente $\nabla f(x)^T w = g^T w$. Como w es arbitrario, concluimos que $g = \nabla f(x)$. Como el gradiente es un subgradiente, y todo subgradiente coincide con él, se tiene que es el único subgradiente de f en x, $\partial f(x) = \{\nabla f(x)\}$.

Para presentar una proposición que relaciona los subgradientes con las funciones convexas, las cuales están muy ligadas a la convergencia del gradiente descendente, primero vamos a definir lo que es un conjunto convexo.

Definición 3.2 (Conjunto convexo) *Un subconjunto $E \subseteq \mathbb{R}^n$ es convexo cuando, para cualesquiera dos puntos de E, el segmento que los une está contenido en E:*

$$x, y \in E \Rightarrow \{(1-t)x + ty : t \in [0,1]\} \subset E.$$

Definición 3.3 (Función convexa) *Sea $E \subset \mathbb{R}^n$ un conjunto convexo no vacío y sea $f : E \to \mathbb{R}$, f es una función convexa en E si, y solo si:*

$$f((1-t)y + tx) \leq (1-t)f(y) + tf(x), \quad \forall t \in [0,1], \forall x, y \in E.$$

Proposición 3.1 (Existencia de subgradientes) *Sea $E \subset \mathbb{R}^n$ un conjunto convexo y $f : E \to \mathbb{R}$. Si $\forall x \in E, \partial f(x) \neq \emptyset$ entonces f es una función convexa. Recíprocamente, si f es convexa entonces se tiene que $\forall x \in int(E)$ $\partial f(x) \neq \emptyset$.*

Esta proposición nos asegura que la familia de las funciones ReLU, que son convexas, siempre tienen subgradiente en su interior. Para demostrarla, primero vamos a necesitar de un teorema, en el ámbito de la convexidad:

Teorema 3.1 (Teorema del Hiperplano de apoyo) *Sea $E \subset \mathbb{R}^n$ un conjunto convexo y $x_0 \in Fr(E)$ un puntro de la frontera de E. Entonces, $\exists w \in \mathbb{R}^n, w \neq 0$ tal que*

$$\forall x \in E, \quad w^T x \geq w^T x_0$$

Demostración de la proposición 3.1. Para la primera implicación, queremos probar que si para todo punto $x \in E$ existe al menos un subgradiente de $f(x)$ entonces se verifica

$$f((1-t)x + ty) \leq (1-x)f(x) + tf(y) \quad \forall x, y \in E, t \in [0,1],$$

es decir, que f es convexa. Tomamos $g \in \partial(z)$, $z \in E$ ya que $\forall z \in E, \partial f(z) \neq \emptyset$, y tenemos por la definición de subgradiente:

$$f(z) - f(x) \leq g^T(z - x),$$
$$f(z) - f(y) \leq g^T(z - y), \tag{4}$$

para $x, y \in E$. Tomamos $z = (1-t)x + ty$ y sustituimos:

$$f((1-t)x + ty) - f(x) \leq g^T(((1-t)x + ty) - x),$$
$$f((1-t)x + ty) + g^T(x - ((1-t)x - ty)) \leq f(x),$$
$$f((1-t)x + ty) + g^T(t(x - y)) \leq f(x),$$
$$f((1-t)x + ty) + tg^T(x - y) \leq f(x). \tag{5}$$

Desarrollando en 4 de manera análoga obtenemos

$$f((1-t)x + ty) + (1-t)g^T(y - x)) \leq f(y). \tag{6}$$

Ahora multiplicamos la desigualdad 5 por $(1-t)$ y la 6 por t, y de su suma obtenemos:

$$\begin{aligned}(1-t)f(x) + tf(y) &\geq (1-t)f((1-t)x + ty) + t(1-t)g^T(x - y) \\ &\quad + tf((1-t)x + ty) + t(1-t)g^T(y - x) \\ &= f((1-t)x + ty) + t(1-t)g^T(x - y) + t(1-t)g^T(y - x) \\ &= f((1-t)x + ty)\end{aligned}$$

donde se ha usado que $g^T(x-y) + g^T(y-x) = 0$. Entonces tenemos que $(1-t)f(x) + tf(y) \geq f((1-t)x + ty)$, $\forall x, y \in E$, $t \in [0, 1]$. Por tanto f es convexa, como queríamos probar.

Ahora vamos a probar que f tiene algún subgradiente en $int(E)$ si es convexa. Definimos el epigrafo de una función f como

$$epi(f) = \{(x, t) \in E \times \mathbb{R} : t \geq f(x)\}.$$

Es obvio que f es convexa si y sólo si su epigrafo es un conjunto convexo. Vamos a aprovechar esta propidad y vamos a construir un subgradiente usando un hiperplano de apoyo al epigrafo de la función. Sea $x \in E$, claramente $(x, f(x)) \in Fr(epi(f))$, y $epi(f)$ es un conjunto convexo por ser f convexa. Entonces usando el Teorema del Hiperplano de Apoyo, existe $(a, b) \in \mathbb{R}^n \times \mathbb{R}$ tal que

$$a^T x + bf(x) \geq a^T y + bt, \quad \forall (y, t) \in epi(f). \tag{7}$$

Reordenando tenemos

$$b(f(x) - t) \geq a^T y - a^T x.$$

Como $t \in [f(x), +\infty[$, para que se mantenga la igualdad incluso cuando $t \to \infty$, debe ocurrir que $b \leq 0$. Ahora vamos a asumir que $x \in int(E)$. Entonces tomamos $\epsilon > 0$, verificando que $y = x + \epsilon a \in E$, lo que implica que $b \neq 0$, ya que si $b = 0$ entonces necesariamente $a = 0$. Reescribiendo 7 con $t = f(y)$ obtenemos

$$f(x) - f(y) \leq \frac{1}{|b|} a^T (x - y).$$

Por tanto $\frac{a}{|b|} \in \partial f(x)$, lo que demuestra la otra parte de la implicación.

□

Tenemos entonces que el subgradiente es una extensión del gradiente en aquellos puntos que no son diferenciables. Por ello podríamos decir que existe el método de descenso de subgradiente, que permite usar funciones que no son diferenciables en todos los puntos, y que se usa de manera implícita en el momento en el que en un modelo se usan funciones de la familia ReLU. Conviene destacar esta diferencia para no perder la rigurosidad, aunque solo sea una formalidad, ya que realmente no se hacen diferencias entre uno y otro método, así que nos seguiremos refiriendo al método de descenso de gradiente aunque estemos trabajando con subgradientes. En la práctica simplemente se elige un valor predeterminado para la derivada en el punto que estas funciones no son diferenciables.

Ejemplo 3.1 (Subgradiente de la función ReLU) *La función ReLU es continua en todo el dominio y diferenciable en* $]-\infty, 0[\cup]0, \infty[$. *Su subgradiente es el siguiente:*

$$\nabla ReLU(x) = \begin{cases} 1, & si \quad x \in]0, \infty[\\ c \in [0, 1] & si \quad x = 0 \\ 0 & si \quad x \in]-\infty, 0[\end{cases}$$

En [Ber+23] se analiza la elección del valor que toma el subgradiente en el punto $x = 0$ y se ve su influencia, que no es poca, en la ejecución del algoritmo, y se concluye que el valor 0 es el que ofrece mejor robustez de manera general.

3.4. Convergencia

La convergencia es un factor crucial en el algoritmo de gradiente descendente. Al tratarse de un algoritmo de optimización iterativo, iremos buscando el mínimo global de la función de coste en varios pasos, o en su defecto

un mínimo local que nos ofrezca una solución subóptima. El algoritmo se mueve hacia puntos de menor gradiente por lo que en caso de converger lo hará a puntos donde sea 0. Un factor clave para la convergencia será el hecho de que la función de pérdida sea o no una función convexa.

En caso de que lo sea sólo existirá un punto crítico[8] y será un mínimo global, por lo que no tenemos que preocuparnos de si el algoritmo se queda estancado en un mínimo local, ya que si converge tendremos la solución óptima. Además en este caso el análisis de la convergencia resulta mucho más sencillo, y por eso encontramos más resultados teóricos y más fuertes que en el caso contrario. Desgraciadamente la situación normal es que la función de coste no sea convexa, y de hecho comprobar que una función sea convexa se trata de un problema NP-Hard [Ahm+11], por lo que en la práctia normalmente no realizamos el análisis teórico de la función y la convergencia previo al entrenamiento del modelo. En caso que no sea convexa, podemos converger hacia un punto crítico que no sea un mínimo global, con lo cual el algoritmo parará y puede que hallamos llegado a una solución que aunque sea subóptima no sea lo suficientemente buena.

3.4.1. Convergencia para BGD

Los desarrollos teóricos sobre la convergencia del algoritmo de descenso del gradiente son muchos y variados, sin embargo no son lo suficientemente útiles en la práctica y se presupone que la función no es convexa. Los principales inconvenientes para el desarrollo de un marco teórico práctico son:

- No existen resultados generales que nos permitan conocer el comportamiento de la convergencia del algoritmo en el problema que estemos tratando con un coste asequible. Los resultados son muy específicos y dependen de la función de coste, el valor de los hiperparámetros y la versión del algoritmo de gradiente descendente que estemos utilizando.
- El estudio teórico de la función de coste es muy complejo y requiere muchos recursos computacionales. Por lo tanto la tendencia a nivel experimental es invertir esos recursos en el entrenamiento, ya que ofrece mejores resultados en relación coste/beneficio de manera general que el estudio teórico de los elementos del algoritmo. Además es un procedimiento genérico aplicable en cualquier problema, por lo que resulta más sencillo.

En el caso que la función de coste sea convexa tenemos un caso más sencillo de analizar, principalmente debido a la curvatura que tienen las

[8] Si existiera una región donde la función fuera constante, cada punto de la región sería un punto crítico pero esto sería extraordinariamente extraño

funciones convexas y al hecho de que cualquier punto crítico será un mínimo global.

Teorema 3.2 (Convergencia para funciones convexas) *Suponemos la función $f : \mathbb{R}^n \to \mathbb{R}$ convexa y diferenciable, con su gradiente Lipschitz continuo con constante $L > 0$, $\|\nabla f(x) - \nabla f(y)\|_2 \leq L\|x - y\|_2 \quad \forall x, y \in \mathbb{R}^n$. Si ejecutamos el algoritmo de gradiente descendente k iteraciones con un $\eta < 1/L$ constante, el error disminuirá tras cada iteración, llegando a una solución $x^{(k)}$ que satisface la siguiente desigualdad:*

$$f(x^{(k)}) - f(x^*) \leq \frac{\|x^{(0)} - x^*\|_2^2}{2\eta k}$$

donde x^* es el mínimo global de la función de error.

Demostración.

En el teorema anterior $x \in \mathbb{R}^n$ contiene los pesos del modelo, y suponemos que el conjunto de datos con el que entrenamos es constante, por lo tanto el error del modelo, $f(x)$, sólo dependerá de los parámetros x.

Como el gradiente ∇f es Lipschitz continuo con constante L entonces $\nabla^2 f(x) \preceq LI$. Esto equivale a que $LI - \nabla^2 f(x)$ sea una matriz semidefinida positiva, por lo que $\nabla^2 f(x) - LI$ es una matriz semidefinida negativa. Ahora hacemos un desarrollo cuadrático de f alrededor de $f(x)$ para obtener:

$$\begin{aligned} f(y) &\leq f(x) + \nabla f(x)^T(y - x) + \frac{1}{2}\nabla^2 f(x)\|y - x\|_2^2 \\ &\leq f(x) + \nabla f(x)^T(y - x) + \frac{1}{2}L\|y - x\|_2^2. \end{aligned}$$

Consideramos ahora y como la actualización de los pesos del gradiente descendente, $y = x - \eta\nabla f(x) = x^+$.

$$\begin{aligned} f(x^+) &\leq f(x) + \nabla f(x)^T(x^+ - x) + \frac{1}{2}L\|x^+ - x\|_2^2 \\ &= f(x) + \nabla f(x)^T(x - \eta\nabla f(x) - x) + \frac{1}{2}L\|x - \eta\nabla f(x) - x\|_2^2 \\ &= f(x) - \eta\nabla f(x)^T\nabla f(x) + \frac{1}{2}L\|\eta\nabla f(x)\|_2^2 \\ &= f(x) - \eta\|\nabla f(x)\|_2^2 + \frac{1}{2}L\eta^2\|\nabla f(x)\|_2^2 \\ &= f(x) - (1 - \frac{1}{2}L\eta)\eta\|\nabla f(x)\|_2^2. \end{aligned}$$

Usamos $\eta \leq \frac{1}{L}$ para ver que $-(1 - \frac{1}{2}L\eta) = \frac{1}{2}L\eta - 1 \leq \frac{1}{2}L(\frac{1}{L}) - 1 = -\frac{1}{2}$, y sustituyendo esta expresión en la desigualdad anterior obtenemos

$$f(x^+) \leq f(x) - \frac{1}{2}\eta\|\nabla f(x)\|_2^2. \tag{8}$$

Esta última desigualdad se traduce en que tras cada iteración del algoritmo del descenso de gradiente el valor del error del modelo es estrictamente decreciente, ya que el valor de $\frac{1}{2}\eta\|\nabla f(x)\|_2^2$ siempre es mayor que 0 a no ser que $\nabla f(x) = 0$, en cuyo caso habremos encontrado el óptimo.

Ahora vamos a acotar el valor del error en la siguiente iteración, $f(x^+)$, en términos del valor óptimo de error $f(x^*)$. Como f es una función convexa se tiene

$$f(x) \leq f(x^*) + \nabla f(x)^T(x - x^*).$$

Sustituyendo en 8 obtenemos

$$\begin{aligned}
f(x^+) &\leq f(x^*) + \nabla f(x)^T(x - x^*) - \frac{\eta}{2}\|\nabla f(x)\|_2^2 \\
f(x^+) - f(x^*) &\leq \frac{1}{2\eta}\left(2\eta\nabla f(x)^T(x - x^*) - \eta^2\|\nabla f(x)\|_2^2\right) \\
f(x^+) - f(x^*) &\leq \frac{1}{2\eta}\left(2\eta\nabla f(x)^T(x - x^*) - \eta^2\|\nabla f(x)\|_2^2 - \|x - x^*\|_2^2 + \|x - x^*\|_2^2\right).
\end{aligned}$$

Como $2\eta\nabla f(x)^T(x-x^*)-\eta^2\|\nabla f(x)\|_2^2-\|x-x^*\|_2^2 = \|x-\eta\nabla f(x)-x^*\|_2^2$, se tiene que

$$f(x^+) - f(x^*) \leq \frac{1}{2\eta}\left(\|x - x^*\|_2^2 - \|x - \eta\nabla f(x) - x^*\|_2^2\right).$$

Usamos ahora la definición de x^+ en esta última desigualdad

$$f(x^+) - f(x^*) \leq \frac{1}{2\eta}\left(\|x - x^*\|_2^2 - \|x^+ - x^*\|_2^2\right).$$

Hacemos la sumatoria sobre las k primeras iteraciones y tenemos

$$\begin{aligned}
\sum_{i=1}^{k}\left(fx^{(i)} - f(x^*)\right) &\leq \sum_{i=1}^{k}\frac{1}{2\eta}\left(\|x^{(i-1)} - x^*\|_2^2 - \|x^{(i)} - x^*\|_2^2\right) \\
&= \frac{1}{2\eta}\left(\|x^{(0)} - x^*\|_2^2 - \|x^{(k)} - x^*\|_2^2\right) \\
&\leq \frac{1}{2\eta}\left(\|x^{(0)} - x^*\|_2^2\right).
\end{aligned}$$

El sumatorio de la derecha ha desaparecido ya que es una serie telescópica. Usando que f decrece con cada iteración, e introduciendo la anterior desigualdad, finalmente llegamos a donde queríamos:

$$f(x^{(k)}) - f(x^*) \leq \frac{1}{k}\sum_{i=1}^{k}\Big(f(x^{(i)} - f(x^*)\Big) \leq \frac{\|x^{(0)} - x^*\|_2^2}{2\eta k}.$$

□

Este teorema nos garantiza que bajo las condiciones supuestas el algoritmo del gradiente descendente converge y además lo hace con ratio de convergencia de $O(1/k)$. Es un resultado teórico muy fuerte que por desgracia no puede usarse en la práctica en la gran mayoría de casos: la constante de Lipschitz L es computacionalmente costosa de calcular, por lo que se usan aproximaciones experimentales para el η, además en muy contadas ocasiones la función de error con la que trabajamos es convexa, y tampoco es sencilla de calcular por lo que directamente no se comprueba si lo es o no lo es, y directamente la suponemos no convexa.

3.4.2. Convergencia para SGD y MBGD

Podemos obtener un resultado mucho más práctico, ya que es para SGD y MBGD y además con condiciones más relajadas. A partir de ahora usaremos SGD para referirnos de manera general a las versiones estocásticas del algoritmo de gradiente descendente, tanto SGD como MBGD.Usando la teoría de algoritmos aproximados estocásticos, con el teorema de Robbins-Siegmund tenemos que bajo las siguientes condiciones, cuando la función es convexa se tiene la convergencia casi segura al mínimo global y cuando no lo es hay convergencia casi segura a un mínimo local. Esto nos da un criterio sencillo con el que aseguramos la convergencia, y que no depende de parámetros como la constante de Lipschitz que son complejos de computar. Para una correcta exposición del teorema y su demostración, primero vamos a definir el concepto de supermartingala y casi-supermartingala.

Usamos como referencia el libro `http://www.sze.hu/~harmati/Sztochasztikus%20folyamatok/lawler.pdf`. En primer lugar vamos a introducir los conceptos de martingala, supermartingala y casi supermartingala, que son tipos de procesos estocásticos. Luego enunciaremos un teorema, el de Robbins-Siegmund, que proporciona un fuerte resultado de convergencia para los procesos casi supermartingalas. Demostrando que el algoritmo de SGD es un proceso de este tipo precisamente, enunciaremos el teorema de convergencia para estos algoritmos, demostrándolo en gran parte gracias al teorema de Robbins-Siegmund.

Vamos a hacer la notación un poco más compacta. Si $X_1, X_2, \ldots$ es una sucesión de variables aleatorias usaremos $\mathcal{F}_n$ para denotar "la información contenida en $X_1, \ldots, X_n$". Escribiremos $E[Y|\mathcal{F}_n]$ en lugar de $E[Y|X_1, \ldots, X_n]$.

Una martingala es un modelo de juego justo. Denotamos por $\{\mathcal{F}_n\}$ una sucesión creciente de información, es decir, para cada n tenemos una sucesión de variables aleatorias $\mathcal{A}_n$ tal que $\mathcal{A}_m < \mathcal{A}_n$ si $m < n$. La información que tenemos en el momento n es el valor de todas las variables en $\mathcal{A}_n$. La suposición $\mathcal{A}_m \subseteq \mathcal{A}_n$ implica que no perdemos información. Decimos que una variable aleatoria X es $\mathcal{F}_n$-medible si podemos determinar el valor de X en caso de conocer el valor de todas las variables aleatorias en $\mathcal{A}_n$. A menudo está sucesión creciente de información $\mathcal{F}_n$ se denomina filtración.

Decimos que una secuencia de variables aleatorias $M_0, M_1, M_2, \ldots$ con $E[|M_i|] < \infty$ es una martingala con respecto a $\{\mathcal{F}\}_n$ si cada M_n es medible con respecto a $\mathcal{F}_n$ y para cada $m < n$,

$$E[M_n|\mathcal{F}_m] = M_n, \tag{9}$$

o equivalentemente,

$$E[M_n - M_m|\mathcal{F}_m] = 0. \tag{10}$$

La condición $E[|M_i|] < \infty$ es necesaria para garantizar que las esperanzas condicionadas están bien definidas. Si $\mathcal{F}_n$ es la información en variables aleatorias $X_1, \ldots, X_n$ entonces también diremos que $M_0, M_1, \ldots$ es una martingala con respecto a $X_0, X_1, \ldots$. A veces diremos que $M_0, M_1, \ldots$ es una martingala sin hacer referencia a la filtración $\mathcal{F}_n$. En ese caso significará que la sucesión M_n es una martingala con respecto a sí misma.

Un proceso M_n con $E[|M_n|] < \infty$ es una supermartingala con respecto a $\{\mathcal{F}_n\}$ si para cada $m < n$ se tiene $E[M_n|\mathcal{F}_m) \leq M_m$. En otras palabras, una supermartingala es un juego injusto. Si un proceso no negativo no verifica la desigualdad anterior, pero verifica que

$$E[M_n|\mathcal{F}_m] \leq (1 + \beta_m)V_m + \xi_m + \zeta_m$$

para $m < n$ y $\beta_n, \xi_n, \zeta_n \geq 0$ siendo $\mathcal{F}_n$-medibles, decimos entonces que M_n es una casi supermartingala. Usando el teorema de Robbins-Siegmund, vamos a obtener un poderoso resultado de convergencia para procesos estocásticos no negativos que son casi supermartingalas.

Teorema 3.3 (Teorema de Robbins-Siegmund) *Suponemos que V_n es una casi supermartingala no negativa. Si*

$$\sum_{n=1}^{\infty} \beta_n < \infty \quad y \quad \sum_{n=1}^{\infty} \xi_n < \infty \quad \textit{casi seguro,}$$

entonces existe una variable aleatoria no negativa V_∞ que verifica

$$\lim_{n\to\infty} V_n = V_\infty \quad y \quad \sum_{n=1}^{\infty} \zeta < \infty \quad \textit{casi seguro.}$$

Ahora vamos a comprobar que el algoritmo de gradiente descendente en su versión SGD es una casi supermartingala. Vamos a definir las funciones fuertemente convexas, porque las necesitaremos para este desarrollo.

Definición 3.4 (Función estrictamente convexa) *Sea $E \subset \mathbb{R}^n$ un conjunto convexo no vacío y sea $f : E \to \mathbb{R}$, f es una función fuertemente convexa en E si es estrictamente convexa*[9] *y además se verifica para algún $m \geq 0$:*

$$(\nabla f(x) - \nabla f(y))^T (x - y) \geq m\|x - y\|^2 \quad \forall x, y \in E.$$

Ahora nos fijamos en la minimización, para un conjunto abierto de parámetros $\mathcal{W}$, de la función objetivo

$$H(W) = E[C(X, W)] \tag{11}$$

para una función de pérdida C. Asumimos que $\nabla H(W) = E[\nabla C(X, W)]$ y que

$$G(W) := E[\|\nabla C(X, W)\|^2] \leq A + B\|W\|^2. \tag{12}$$

Asumimos también que $\nabla H(W^*) = 0$ y que

$$(W - W^*)^T \nabla H(W) \geq c\|W - W^*\|^2, \tag{13}$$

lo que implica que W^* es un minimizador único de H, es decir, que es un mínimo global y es único. Esta última condición se mantiene siempre que H sea estrictamente convexa, pero solo necesitamos que se mantenga en W^*.

Añadiendo la tasa de aprendizaje como una sucesión en lugar de una constante y expresando las sucesiones como en 3.3 para facilitar la comprensión, la regla de actualización de los pesos que vimos en 3 quedaría como

$$W_n = W_{n-1} - \eta_{n-1} \nabla C(X_n, W_{n-1}) \tag{14}$$

para una secuencia $X_1, X_2, \ldots$ de variables aleatorias independientes e idénticamente distribuidas. Asumimos que la tasa de aprendizaje η_{n-1} puede depender de $X_1, \ldots, X_{n-1}$ y $W_0, \ldots, W_{n-1}$. De esto obtenemos que

$$\begin{aligned} V_n &= \|W_n - W*\|^2 \\ &= \|W_{n-1} - \eta_{n-1}\nabla C(X_n, W_{n-1}) - W^*\|^2 \\ &= V_{n-1} + \eta_{n-1}^2 \|\nabla C(X_n, W_{n-1})\|^2 - 2\eta_{n-1}(W_{n-1} - W^*)^T \nabla C(X_n, W_{n-1}). \end{aligned}$$

Tomando la esperanza condicionada resulta

[9] La convexidad estricta es igual que la convexidad normal, pero la desigualdad de su definición es una desigualdad estricta

$$\begin{aligned} E[V_n|\mathcal{F}_{n-1}] &= V_{n-1} + \eta_{n-1}^2 E[\|\nabla C(X_n, W_{n-1})\|^2|\mathcal{F}_{n-1}] \\ &\quad - 2\eta_{n-1}(W_{n-1} - W^*)^T E[\nabla C(X_n, W_{n-1})|\mathcal{F}_{n-1}] \\ &= V_{n-1} + \eta_{n-1}^2 G(W_{n-1}) - 2\eta_{n-1}(W_{n-1} - W^*)^T \nabla H(W_{n-1}) \\ &\leq V_{n-1} + \eta_{n-1}^2 (A + B\|W_{n-1}\|^2) - 2c\eta_{n-1}\|W_{n-1} - W^*\|^2. \end{aligned} \tag{15}$$

Ahora observamos que

$$\begin{aligned} \|W_{n-1}\|^2 &= \|W_{n-1} - W^* + W^*\|^2 \\ &\leq (\|W_{n-1} - W^*\| + \|W^*\|)^2 \\ &\leq 2\|W_{n-1} - W^*\|^2 + 2\|W^*\|^2 \\ &= 2V_{n-1} + 2\|W^*\|^2, \end{aligned}$$

e introduciendo esta desigualdad en 15 obtenemos

$$\begin{aligned} E[V_n|\mathcal{F}_{n-1}] &\leq V_{n-1} + \eta_{n-1}^2 (A + 2BV_{n-1} + 2B\|W^*\|^2) - 2c\eta_{n-1}V_{n-1} \\ &= (1 + 2B\eta_{n-1}^2)V_{n-1} + \eta_{n-1}^2 (A + 2B\|W^*\|^2) - 2c\eta_{n-1}V_{n-1}. \end{aligned}$$

Esto demuestra que V_n es una casi supermartingala con $\beta_n = 2B\eta_n^2$, $\xi_n = \eta_n^2(A + 2B\|W^*\|^2)$ y $\zeta_n = c\eta_n V_n$.

Estamos ya en condiciones de enunciar y demostrar el teorema relativo a la convergencia del algoritmo SGD.

Teorema 3.4 (Convergencia de algoritmos SGD) *Con las suposiciones realizadas anteriormente sobre la función de coste C, el proceso V_n converge casi seguro a un límite V_∞ si*

$$\sum_{n=1}^{\infty} \eta_n^2 < \infty \quad \textit{casi seguro.} \tag{16}$$

Si también

$$\sum_{n=1}^{\infty} \eta_n = \infty \quad \textit{casi seguro} \tag{17}$$

entonces el límite es $V_\infty = 0$ y se tiene

$$\lim_{n\to\infty} W_n = W^* \quad \textit{casi seguro.}$$

Demostración.

La primera parte se sigue directamente del teorema de Robbins-Siegmund. Para la segunda, asumimos $V_\infty > 0$ en un conjunto de probabilidad positiva y procedemos por contradicción. Hay entonces una variable aleatoria N tal que en ese conjunto $V_n \geq \frac{V_\infty}{2}$ para $n \geq N$, y

$$\sum_{n=1}^{\infty} \zeta_n = c \sum_{n=1}^{\infty} \eta_n V_n \geq \frac{cV_\infty}{2} \sum_{n=N}^{\infty} \eta_n = \infty$$

con probabilidad positiva. Esto contradice el teorema de Robbins-Siegmund. Concluimos que $V_\infty = 0$ casi seguro, entonces $V_n = \|W_n - W^*\|^2 \to 0$ casi seguro, o

$$W_n \to W^*$$

casi seguro cuando $n \to \infty$.

□

Aunque la suposición de que H es globalmente fuertemente convexa es demasiado fuerte, ya que hace que el teorema no sea útil en la práctica, podemos esperar que el algoritmo tenga un comportamiento similar en el entorno de un minimizador local $W*$ si H es fuertemente convexa en ese entorno.

La conclusión más importante que obtenemos de este resultado es que la tasa de aprendizaje debe tender a cero para asegurarnos la convergencia teórica del algoritmo. En el teorema 3.4 la condición 16 nos dice cómo de rápido debe converger, mientras que la condición 17 nos dice que no debe converger demasiado rápido.

Ejemplo 3.2 *Tomamos como valores de la tasa de aprendizaje la sucesión $\eta_n = e^{-n}$. Entonces tenemos que el proceso V_n, es decir el algoritmo SGD, converge a V_∞ ya que se cumple la primera condición del teorema. Sin embargo la segunda condición no se cumple, lo que quiere decir que no sabemos si $V_\infty = 0$, por tanto no nos aseguramos converger a un minimizador.*

3.4.3. Problemas en la convergencia

En el teorema 3.2 tenemos asegurada la convergencia a un mínimo aunque con unos requisitos que no se suelen encontrar en la práctica. En la proposición **??** por el contrario solo nos garantizamos llegar a un punto crítico, ni siquiera a un mínimo local. Encontramos aquí el mayor problema para la convergencia del algoritmo del gradiente descendente: la convergencia prematura en puntos con gradiente muy cercano a cero que no son soluciones subóptimas.

Cuando el algoritmo se aproxima a un punto crítico, la magnitud del gradiente se aproxima a cero, y teniendo en cuenta la regla de actualización

de los pesos, $W_{t+1} = W_t - \eta \nabla C(W)$, tenemos por tanto que $W_{t+1} - W_t \approx 0$. Es decir que las modificaciones de los pesos con las actualizaciones serán prácticamente nulas, haciendo que el algoritmo se pare o que progrese de manera muy lenta cerca de estos puntos, lo que en un primer momento podría aparentar una falsa convergencia en regiones planas por ejemplo.

Los puntos críticos más comunes son los puntos de silla, que definimos como un punto x_s que verifica que $\nabla f(x_s) = 0$ pero x_s no es ni un mínimo local ni un máximo local. En x_s la matriz Hessiana de f, $\nabla^2 f(x_s)$, tiene valores propios tanto positivos como negativos, lo que indica que la función f se curva hacia abajo en unas direcciones y hacia arriba en otras en el punto x_s.

En espacios de alta dimensionalidad, que son comunes en las redes neuronales, la probabilidad de encontrar puntos de silla es mucho mayor que la de encontrar máximos y minímos locales. Para una función $f : \mathbb{R}^n \to \mathbb{R}$, el número de puntos de silla normalmente crece exponencialmente con respecto a la dimensión n. Esto se debe a que la probabilidad de encontrar valores propios de ambos signos en la matriz Hessiana aumenta con la dimensionalidad del espacio de parámetros [Dau+14].

La manera de solventar estos problemas es utilizar modificaciones en el algoritmo de gradiente descendente que proporcionan mejores propiedades a su convergencia, ya que las estrategias de SGD y MBGD ofrecen una pequeña pero insuficiente solución a este problema. Al calcular el gradiente mediante una aproximación con un subconjunto de los datos, se introduce un ruido ϵ en su cálculo con lo que $W_{t+1} - W_t \approx \epsilon$, que puede servir para conseguir escapar de ese punto de silla. Dichas modificaciones se denominan optimizadores y a diferencia de las versiones vistas en la sección 3.2.1, que variaban solo en la cantidad de datos usados para calcular el gradiente, estos optimizadores cambian la regla de actualización de los pesos añadiendo nuevos cálculos, hiperparámetros y estrategias para conseguir que el algoritmo mejore en estabilidad, robustez y velocidad de convergencia.

Existen otros problemas como la explosión o el desvanecimiento del gradiente, pero están ligados a BP como herramienta para calcularlo, por lo que se abordarán en la sección siguiente junto a la inicialización de pesos del modelo, que es la manera principal de superar estos problemas.

4. BP

Ya conocemos el algoritmo de aprendizaje del gradiente descentente, y en esta sección veremos BP, que como ya hemos mencionado, es el algoritmo más usado a la hora de calcular el gradiente en el entrenamiento de un modelo.

Cuando queremos obtener las predicciones de una red neuronal para unos datos de entrada concretos, la información fluye desde atrás hacia delante,

es decir desde la capa de entrada x, pasando por las capas ocultas hasta producir una salida o, que si es evaluada con la función de coste C produce un escalar E que representa el error del modelo. A esto lo llamamos propagación hacia delante (*forward propagation*).

Para calcular el gradiente de la función de coste respecto a los pesos del modelo necesitamos que la información fluya en sentido contrario, es decir, propagamos el error E, pasando por las capas ocultas, hasta la capa de entrada x. Esto se conoce como propagación hacia atrás (*backpropagation*). El algoritmo de BP toma su nombre de aquí ya que durante su aplicación necesitamos que la información se propage hacia atrás. Si bien, no se trata del mismo concepto, ya que podemos propagar la información hacia detrás sin necesidad de calcular el gradiente, con lo que no estaremos usando BP.

4.1. Diferenciación automática

El algoritmo de BP se implementa en la práctica a través de la diferenciación automática [Bay+15], que es un algoritmo más general para calcular derivadas y que engloba a BP. Se fundamenta en descomponer las funciones en una secuencia de operaciones fundamentales para calcular sus derivadas a través de la regla de la cadena, haciendo este cómputo muy eficiente. Por ello se distingue de la diferenciación simbólica, que manipula las expresiones matemáticas para encontrar derivadas, y de la diferenciación numérica, que calcula las derivadas a través de aproximaciones con diferencias finitas. Esta es la implementación que se usa en las librerías de aprendizaje automático más usadas, como TensorFlow 2 y PyTorch.

En la diferenciación automática existen dos estrategias para calcular un vector gradiente o una matriz jacobiana: diferenciación hacia delante y diferenciación hacia atrás. Su distinción reside principalmente en si realizamos multiplicaciones de un vector por un jacobiano (hacia atrás) o de un jacobiano por un vector (hacia delante). La elección dependerá de las dimensiones de la matriz jacobiana que queramos calcular, en otras palabras, debemos comparar la dimensión de la entrada y de la salida del modelo. Si la dimensión de entrada es mayor que la de salida, necesitaremos menos operaciones para calcular la matriz jacobiana si usamos la estrategia de diferenciación hacia atrás y viceversa.

Debido a la estructura general de una red neuronal donde la dimensión de la entrada es mucho mayor que la de la salida, resulta más eficiente calcular el gradiente con la diferenciación hacia atrás, y esto es lo que entendemos como el algoritmo de BP: la información se propaga hacia atrás en el modelo mientras que se usa la diferenciación hacia atrás con el objetivo de calcular el gradiente del error del modelo con respecto a sus pesos. Si usáramos la diferenciación hacia delante, aunque estuviéramos propagando la información hacia atrás no estaríamos usando el algoritmo de BP, y además el cálculo resultaría mucho más ineficiente. Se suele decir de manera general que el

algoritmo de BP es una aplicación concreta de la diferenciación automática hecha a medida para el entrenamiento de redes neuronales.

Vamos a explorar el algoritmo de BP de manera progresiva: en primer lugar veremos la diferenciación hacia delante y hacia atrás, viendo por qué es más eficiente usar la segunda y acotando este algoritmo general para llegar al algoritmo de BP, para lo que veremos como calcular la matriz Jacobiana de la salida de un perceptrón multicapa (MLP, por sus siglas en inglés) respecto a la entrada, en una situación que no será de entrenamiento, ya que no habrá parámetros entrenables, pero nos servirá para ilustrar el funcionamiento de BP. Luego veremos cómo obtenemos el gradiente del error con respecto a los pesos de cada capa usando el algoritmo de BP en un MLP con parámetros entrenables, concretando con ejemplos para las capas más comúnmente utilizadas. Finalmente vamos a generalizar este concepto hacia modelos más abstractos usando grafos dirigidos acíclicos.

Los MLP son un tipo de red neuronal, divididos en capas compuestas de nodos llamados neuronas, donde cada nodo de cada capa está conectado con todos los nodos de la capa siguiente. Son usados principalmente con conjuntos de entrenamiento tabulares, es decir aquellos que sus datos tienen formato de tabla. Usaremos esta arquitectura para realizar el desarrollo teórico ya que es más sencilla conceptualmente y facilita la notación.

4.2. Diferenciación hacia delante vs hacia atrás en un MLP

Definimos nuestro modelo como $f : \mathbb{R}^n \to \mathbb{R}^m$, $o = f(x)$ con $x \in \mathbb{R}^n$ y $o \in \mathbb{R}^m$. Asumimos que f es una composición de funciones:

$$f = f_k \circ f_{k-1} \circ \cdots \circ f_2 \circ f_1.$$

Donde $k-1$ es el número de capas ocultas del MLP y $k+1$ el total de capas. Cada función f_i representa el cálculo que se realiza en la capa i-ésima. Se tiene para $i \in \{1, \ldots, k\}$

$$f_i : \mathbb{R}^{m_i} \to \mathbb{R}^{m_{i+1}}$$

$$f_i(x_i) = x_{i+1}$$

Donde la entrada del modelo viene representada en la primera capa, $x = x_1$. Además se tiene que $m_1 = n, m_{k+1} = m, x_{k+1} = o$. Para obtener la predicción del modelo, $o = f(x) = f_k(x_k)$, necesitamos calcular el resultado de todas las capas intermedias $x_{i+1} = f_i(x_i)$.

Podemos ver que la matriz jacobiana de la salida con respecto a la entrada $J_f(x) \in \mathbb{R}^{m \times n}$ puede ser calculada usando la regla de la cadena . Esto nos va a servir para ilustrar las diferencias entre la diferenciación hacia atrás y hacia delante

$$\begin{aligned} J_f(x) = \frac{\partial o}{\partial x} &= \frac{\partial o}{\partial x_k}\frac{\partial x_k}{\partial x_{k-1}}\cdots\frac{\partial x_3}{\partial x_2}\frac{\partial x_2}{\partial x_1} \\ &= \frac{\partial f_k(x_k)}{\partial x_k}\frac{\partial f_{k-1}(x_{k-1})}{\partial x_{k-1}}\cdots\frac{\partial f_2(x_2)}{\partial x_2}\frac{\partial f_1(x_1)}{\partial x_1} \\ &= J_{f_k}(x_k)J_{f_{k-1}}(x_{k-1})\cdots J_{f_2}(x_2)J_{f_1}(x_1). \end{aligned}$$

Se discute ahora como calcular el jacobiano $J_f(x)$ de manera eficiente. Recordamos que

$$\begin{aligned} J_f(x_1) = \frac{\partial f(x_1)}{\partial x_1} &= \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{pmatrix} \\ &= \begin{pmatrix} \nabla f_1(x)^T \\ \vdots \\ \nabla f_m(x)^T \end{pmatrix} = \left(\frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_n}\right) \in \mathbb{R}^{m\times n}. \end{aligned}$$

Donde $\nabla f_i(x)^T \in \mathbb{R}^{1\times n}$ es la fila i-ésima y $\frac{\partial f}{\partial x_j} \in \mathbb{R}^m$ es la columna j-ésima de la matriz jacobiana , para $i = 1, \ldots, m$ y $j = 1, \ldots, n$.

Podemos extraer la fila i-ésima del jacobiano usando un producto vector-jacobiano (PVJ) de la forma $e_i^T J_f(x)$, donde $e_i \in \mathbb{R}^m$ es el vector de la base canónica. De manera análoga se puede extraer la columna j-ésima de $J_f(x)$ usando un producto jacobiano-vector (PJV) de la forma $J_f(x)e_j$, donde $e_j \in \mathbb{R}^n$. Se tiene entonces que el cálculo de la matriz jacobiana $J_f(x)$ equivale a n PJV o m PVJ.

Para construir el jacobiano a partir de operaciones PJV o PVJ, podemos suponer que el cálculo del gradiente de $f_i(x)$ tiene el mismo coste computacional que el cálculo de la derivada parcial de f con respecto de alguna de las variables x_j. Por tanto la forma de cálculo más eficiente de la matriz jacobiana depende de qué valor es mayor: si n o m.

Si $n \leq m$ será más eficiente construir el jacobiano $J_f(x)$ usando PJV de derecha a izquierda.

$$J_f(x)v = J_{f_k}(x_k)\cdots J_{f_2}(x_2)J_{f_1}(x_1)v.$$

Donde $J_{f_k}(x_k)$ tiene tamaño $m \times m_{k-1}$, $J_{f_i}(x_i)$ tiene tamaño $m_i \times m_{i-1}$ para $i \in 2, \ldots, k-1$ y $J_{f_1}(x_1)$ tiene tamaño $m_1 \times n$ mientras que el vector columna v será $n \times 1$. Esta multiplicación se puede calcular usando el agoritmo de diferenciación hacia delante definido en 1.

Algorithm 1 Diferenciación hacia delante

$x_1 := x$
for $j \in \{1, \dots, n\}$ **do**
  $v_j := e_j \in \mathbb{R}^n$
end for
for $i \in \{1, \dots, k\}$ **do**
  $x_{i+1} := f_i(x_i)$
  for $j \in \{1, \dots, n\}$ **do**
    $v_j := J_{f_i}(x_i)v_j$
  end for
end for
  return $o = x_{k+1}, (v_1, v_2, \dots, v_n)$

Donde los elementos $v_j, j \in \{1, \dots, n\}$ de la matriz fila v se corresponden con las derivadas parciales de la función del MLP respecto a la entrada de la capa j, es decir la columna j-ésima de la matriz jacobiana, $v_j = \frac{\partial f}{\partial x_j}$.

Si $n \geq m$ es más eficiente calcular $J_f(x)$ para cada fila $i = 1, \dots, m$ usando PVJ de izquierda a derecha. La multiplicación izquierda con un vector fila u^T es

$$u^T J_f(x) = u^T J_{f_k}(x_k) \cdots J_{f_2}(x_2) J_{f_1}(x_1).$$

Donde u^T tiene tamaño $1 \times m$, $J_{f_k}(x_k)$ tiene tamaño $m \times m_{k-1}$, $J_{f_i}(x_i)$ tiene tamaño $m_i \times m_{i-1}$ para $i \in 2, \dots, k-1$ y $J_{f_1}(x_1)$ tiene tamaño $m_1 \times n$. Esto puede calcularse usando la diferenciación hacia atrás (ver algoritmo 2).

Algorithm 2 Diferenciación en modo reverso

$x_1 := x$
for $k \in \{1, \dots, K\}$ **do**
  $x_{k+1} = f_k(x_k)$
end for
for $i \in \{1, \dots, m\}$ **do**
  $u_i := e_i \in \mathbb{R}^m$
end for
for $k \in \{K, \dots, 1\}$ **do**
  for $i \in \{1, \dots, m\}$ **do**
    $u_i^T := u_i^T J_{f_k}(x_k)$
  end for
end for
  return $o = x_{k+1}, \left(u_1^T, u_2^T, \dots, u_m^T\right)$

Donde los elementos u_i^T se corresponden con el gradiente de la función f_i, $u_i^T = \nabla f_i(x)^T$, es decir la fila i-ésima de la matriz jacobiana. Asumimos que $m = 1$ ya que la salida de la función de error del modelo es un escalar, y que

$n = m_i$ $i \in \{2, \ldots, k-1\}$; entonces el coste de computar el jacobiano usando la diferenciación hacia atrás es $O(n^2)$. Este algoritmo aplicado al cálculo del gradiente del error del modelo con respecto a los pesos, propagando la información hacia atrás y con el objetivo de usar gradiente descendente, es lo que conocemos como BP.

Con la notación que estamos empleando, cuando $m = 1$ el gradiente $\nabla f(x)$ tiene la misma dimensión que x. Por tanto es un vector columna mientras que $J_f(x)$ es un vector fila, por lo que técnicamente se tiene que $\nabla f(x) = J_f(x)^T$. Es de vital importancia aclarar esto ya que es el caso en el que nos situamos cuando usamos BP. La dimensión de salida siempre es uno, ya que calculamos la matriz jacobiana de la función de error del modelo con respecto a los pesos, con lo que será un vector gradiente de dimensión igual a la dimensión de los pesos del modelo. La predicción del modelo puede tener dimensión 1 en tareas de regresión, o una dimensión mayor para tareas de clasificación, aunque de manera general no suele ser mayor de 100. La función de error del modelo siempre tendrá como imagen un valor real.

Acabamos de comprobar que para calcular una matriz jacobiana resulta más eficiente usar diferenciación hacia delante si la dimensión de la entrada es menor que la dimensión de la salida; y si la dimensión de la salida es menor que la dimensión de la entrada es preferible usar diferenciación hacia atrás. Por las características de las redes neuronales y los problemas en los que se aplican, siempre vamos a tener que la dimensión de salida es mucho menor que la dimensión de la entrada, por lo que es mucho más eficiente usar la diferenciación hacia atrás.

4.3. BP para MLP

En la sección anterior hemos visto un modelo que no tenía ningún parámetro entrenable. Ahora usaremos uno que sí los tiene y veremos cómo calcular el gradiente de la función de coste con respecto a ellos. Los parámetros son valores reales y tienen la forma $W = W_1 \times W_2 \times \cdots \times W_k \subset \Omega$, con $W_i \in \mathbb{R}^{n_i \times n_{i+1}}$ donde n_i es el número de neuronas de la i-ésima capa. El modelo que tendríamos añadiendo los pesos es $f : \mathbb{R}^n \times \Omega \to \mathbb{R}^m$, $o = f(x, W)$ con $x \in \mathbb{R}^n$ y $o \in \mathbb{R}^m$. Donde las funciones de cada capa son de la forma $f_i(x_i, W_i) = \sigma_i(W_i x_i) = x_{i+1}$ donde σ_i es una función de activación generalmente no lineal. Dependiendo del tipo de problema, la función f_k puede ser distinta: en un problema de regresión usamos la identidad, en clasificación usamos la función *Softmax* que convierte el vector de la predicción del modelo en uno cuyos elementos suman 1 y donde el elemento de la posición i-ésima representa la probabilidad de que la entrada pertenezca a la clase i.

Ahora vamos a considerar la función de coste del modelo como una capa más, a parte de las funciones de las capas ocultas que nos permiten obtener la predicción. Siguiendo con la notación anterior, incluyendo la función de error $\mathcal{L} : \mathbb{R}^n \times \Omega \times \mathbb{R}^m \to \mathbb{R}$, $E = \mathcal{L}((x, W, y) = C(f(x, W), y)$, con $y \in \mathbb{R}^m$

siendo la etiqueta correcta para la entrada x y $C : \mathbb{R}^m \times \mathbb{R}^m \to \mathbb{R}$ la función de coste del modelo. Con esto tenemos que $\mathcal{L} = C \circ f$.

Ejemplo 4.1 *Suponemos un MLP con dos capas ocultas, la salida escalar (problema de regresión) y una función de pérdida* $C(f(x, W), y) = \frac{1}{2}\|f(x, W) - y\|^2$. *Entonces tenemos* $\mathcal{L} : \mathbb{R}^n \to \mathbb{R}$ *y cada capa tiene ecuación*

$$f_1(x_1, W_1) = \sigma_1(W_1 x_1) = x_2$$
$$f_2(x_2, W_2) = W_2 x_2 = x_3 = f(x, W) = o$$
$$C(f(x, W), y) = \frac{1}{2}\|x_3 - y\|^2 = E$$

El objetivo será calcular el gradiente del error con respecto a los parámetros $\frac{\partial E}{\partial W}$ para poder aplicar el entrenamiento a través del gradiente descendente. Buscamos obtener un vector gradiente de la misma dimensión que W, pero el cálculo no es directo, calcularemos progresivamente el gradiente de la función de coste con respecto a los pesos de cada capa, desde la capa final hasta la inicial por lo que buscamos calcular $\frac{\partial E}{\partial W_i}, \forall i = 1, \ldots, k$. Para la última capa $\frac{\partial E}{\partial W_k}$ el cálculo es inmediato, mientras que para el resto podemos usar la regla de la cadena para obtener que

$$\frac{\partial \mathcal{L}}{\partial W_i} = \frac{\partial \mathcal{L}}{\partial x_k} \frac{\partial x_k}{\partial x_{k-1}} \frac{\partial x_{k-1}}{\partial x_{k-2}} \cdots \frac{\partial x_{i+1}}{\partial W_i}$$

Cada $\frac{\partial \mathcal{L}}{\partial W_i} = \left(\nabla_{W_i} \mathcal{L}^T\right)$ es un vector gradiente con el mismo número de elementos que W_i. Estos se calculan propagando hacia atrás la información en el modelo y usando la estrategia de diferenciación hacia atrás a través de PVJ, es decir, el algoritmo de BP que podemos ver en el pseudocódigo 3.

Algorithm 3 BP para MLP con k capas

```
//Propagación hacia delante
x_1 := x
for l ∈ {1, ..., L} do
    x_{l+1} = f_k(x_l, W_l)
end for
//Propagación hacia atrás
u_{L+1} = 1
for l ∈ {L, ..., 1} do
    g_l := u_{l+1}^T ∂f_l(x_l, W_l)/∂W_l
    u_l^T := u_{l+1}^T ∂f_l(x_l, W_l)/∂x_l
end for
    return {∇_{W_l}; l = 1, ..., L}
```

Para tener una idea más profunda y completa acerca del algoritmo de BP, vamos a ver como calcular el PVJ para las capas más comunes en los modelos, para lo que analizaremos sus matrices Jacobianas respecto de la entrada de la capa.

4.3.1. Capa no-lineal

Consideramos primero una capa que aplica una función no lineal, normalmente el caso de las funciones de activación. $z = \sigma(x)$, con $z^{(i)} = \sigma(x^{(i)})$. El elemento en la posición (i, j) del Jacobiano es dado por:

$$\frac{\partial z^{(i)}}{\partial x^{(j)}} = \begin{cases} \sigma'(x^{(i)}) & \text{si } i = j \\ 0 & \text{en otro caso.} \end{cases}$$

Donde $\sigma'(a) = \frac{d\sigma}{da}(a)$. En otras palabras, el Jacobiano con respecto de la entrada es

$$J = \frac{\partial \sigma}{\partial x} = diag(\sigma'(x)).$$

Si tomamos como ejemplo la función $ReLU$, para un vector arbitratio u, podemos calcular su PVJ $u^T J$ a través de la multiplicación de elementos de la diagonal de J con el vector u.

$$\sigma(a) = ReLU(a) = max(a, 0),$$

$$\sigma'(a) = \begin{cases} 0 & \text{si } a < 0 \\ 1 & \text{si } a > 0. \end{cases}$$

Como hemos visto en la sección 3.3 la función $ReLU$ no es diferenciable en el punto 0, pero sí que admite subderivada en todo su dominio, y en el punto $a = 0$ es cualquier valor entre $[0, 1]$, y usualmente en la práctica se toma el valor 0. Por tanto

$$ReLU'(a) = \begin{cases} 0 & \text{si } a \leq 0 \\ 1 & \text{si } a > 0. \end{cases}$$

$$J = \frac{\partial \sigma}{\partial x} = diag(ReLU'(x)).$$

4.3.2. Capa Cross Entropy

Consideramos ahora la capa cuya función es la función de coste, en concreto con una medición del error usando *Cross-Entropy Loss* donde tenemos C clases, que toma las predicciones x y las etiquetas y como entrada y devuelve un escalar. Recordamos que en esta capa la matriz Jacobiana es un vector fila, identificado con el gradiente, ya que la salida es un escalar.

$$\begin{aligned} z = f(x) &= CrossEntropy(x, y) \\ &= -\sum_c y_c log(softmax(x)_c) = -\sum_c y_c log(p_c) \end{aligned}$$

donde $p_c = softmax(x)_c = \frac{e^{x_c}}{\sum_{c'=1}^{C} e^{x_{c'}}}$ son las probabilidades de las clases predichas, e y es la etiqueta correcta con codificación *one-hot encoded vector*, es decir un vector de C elementos que representa la clase real a la que pertenece; si ese elemento pertenece a la clase k, todos las posiciones del vector y serán 0 a excepción de la posición k, que será 1. El Jacobiano con respecto a la entrada es

$$J = \frac{\partial z}{\partial x} = (p - y)^T \in \mathbb{R}^{1 \times C}.$$

Vamos a asumir que la clase objetivo es la etiqueta c:

$$z = f(x) = -log(p_c) = -log\left(\frac{e^{x_c}}{\sum_j e^{x_j}}\right) = log\left(\sum_j e^{x_j}\right) - x_c.$$

Entonces

$$\frac{\partial z}{\partial x_i} = \frac{\partial}{\partial x_i} log \sum_j e^{x_j} - \frac{\partial}{\partial x_i} x_c = \frac{e^{x_i}}{\sum_j e^{x_j}} - \frac{\partial}{\partial x_i} x_c = p_i - \mathbb{I}(i = c).$$

4.3.3. Capa lineal

Consideramos por último una capa lineal $z = f(x, W) = Wx$, donde $W \in \mathbb{R}^{m \times n}$, con $x \in \mathbb{R}^n$ y $z \in \mathbb{R}^m$ son respectivamente la entrada y la salida de esa capa.

Conviene aclarar, para evitar confusiones, que en la descripción previa hemos considerado las capas ocultas como una combinación de las operaciones lineales que aquí se describen con las funciones de activación, aquí sin embargo las analizamos por separado con el objetivo de una descripción más sencilla y un análisis más individualizado. Esta agrupación es una abstracción y por tanto no varía en cuanto a resultados.

Podemos calcular el Jacobiano de la función de la capa con respecto al vector entrada de esa capa, $J = \frac{\partial z}{\partial x} \in \mathbb{R}^{m \times n}$. Como

$$z_i = \sum_{l=1}^{n} W_{il} x_l.$$

El elemento que ocupa la posición (i, j) en la matriz Jacobiana será

$$\frac{\partial z_i}{\partial x_j} = \frac{\partial}{\partial x_j} \sum_{l=1}^{n} W_{il} x_l = \sum_{l=1}^{n} W_{il} \frac{\partial}{\partial x_j} x_l = W_{ij}$$

ya que $\frac{\partial}{\partial x_j} x_l = \mathbb{I}(l = j)$. Por tanto el Jacobiano con respecto a la entrada será

$$J = \frac{\partial z}{\partial x} = W.$$

El PVJ entre $u^T \in \mathbb{R}^{1\times m}$ y $J \in \mathbb{R}^{m\times n}$ es

$$u^T \frac{\partial z}{\partial x} = u^T W \in \mathbb{R}^{1\times n}.$$

Ahora consideramos el Jacobiano con respecto a la matriz de los pesos, $J = \frac{\partial z}{\partial W}$. Esto se puede representar como una matriz de tamaño $m \times (m \times n)$, que resulta compleja de manejar. Por tanto en lugar de eso veremos de manera individual como calcular el gradiente con respecto a un único peso W_{ij}. Esto es más sencillo de calcular ya que $\frac{\partial z}{\partial W_{ij}}$ es un vector. Para su cómputo nos fijamos en que

$$z_l = \sum_{t=1}^{n} W_{lt} x_t, \qquad \text{y}$$

$$\frac{\partial z_l}{\partial W_{ij}} = \sum_{t=1}^{n} x_t \frac{\partial}{\partial W_{ij}} W_{lt} = \sum_{t=1}^{n} x_t \mathbb{I}(i = l \text{ y } j = t).$$

Por tanto

$$\frac{\partial z}{\partial W_{ij}} = (0, \ldots, 0, x_j, 0, \ldots, 0)^T$$

Donde el elemento no nulo ocupa la posición i-ésima. El PVJ entre $u^T \in \mathbb{R}^{1\times m}$ y $\frac{\partial z}{\partial W} \in \mathbb{R}^{m\times(m\times n)}$ se puede representar como una matriz de tamaño $1 \times (m \times n)$. Vemos que

$$u^T \frac{\partial z}{\partial W_{ij}} = \sum_{l=1}^{m} u_l \frac{\partial z_l}{\partial W_{ij}} = u_i x_j.$$

Con lo cual

$$u^T \frac{\partial z}{\partial W} = u x^T \in \mathbb{R}^{m\times n}.$$

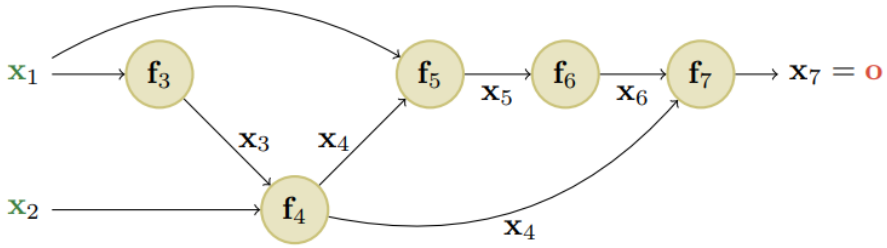

Figura 4: Representación del grafo dirigido acíclico (imagen obtenida de [Mur22])

4.3.4. Grafos computacionales

Los MLP son un tipo de Redes Neuronales Profundas donde cada capa se conecta directamente con la siguiente formando una estructura de cadena. Sin embargo las Redes Neuronales Profundas más recientes combinan componentes diferenciables de forma mucho más compleja, creando un grafo computacional de forma similar a como en la programación se combinan funciones simples para hacer otras más complejas. La restricción es que el grafo resultante debe de ser un grafo acíclico dirigido, donde cada nodo es una función subdiferenciable. Un grafo acíclico dirigido es un tipo de grafo donde cada arista que une dos nodos tiene un sentido específico desde un nodo al otro (dirigido) y en el que no se forman ciclos, es decir, partiendo de un nodo dado no existe una secuencia de aristas dirigidas por la que se pueda volver a él.

Vamos a ver un ejemplo usando la función $f(x_1, x_2) = x_2 e^{x_1} \sqrt{x_1 + x_2 e^{x_1}}$, cuyo grafo se puede ver representado en la figura 4.

Las funciones intermedias que vemos en el grafo son:

$$\begin{aligned} x_3 &= f_3(x_1) = e^{x_1} \\ x_4 &= f_4(x_2, x_3) = x_2 x_3 \\ x_5 &= f_5(x_1, x_4) = x_1 + x_4 \\ x_6 &= f_6(x_5) = \sqrt{x_5} \\ x_7 &= f_7(x_4, x_6) = x_4 x_6. \end{aligned}$$

Ahora no tenemos una estructura de cadena y puede que necesitemos sumar los gradientes a través de diferentes caminos, como es el caso del nodo x_4 que influye en x_5 y x_7. Para asegurar un funcionamiento correcto basta con nombrar los nodos en orden topológico (los padres antes que los hijos) y luego hacer la computación en orden topológico inverso. En general usamos

$$\frac{\partial o}{\partial x_j} = \sum_{k \in Hijos(j)} \frac{\partial o}{\partial x_k} \frac{\partial x_k}{\partial x_j}.$$

En nuestro ejemplo para el nodo x_4:

$$\frac{\partial o}{\partial x_4} = \frac{\partial o}{\partial x_5} \frac{\partial x_5}{\partial x_4} + \frac{\partial o}{\partial x_7} \frac{\partial x_7}{\partial x_4}.$$

En la práctica el grafo computacional se puede calcular previamente, usando una librería que nos permita definir un grafo estático. Alternativamente podemos calcular el grafo en tiempo real, siguiendo la ejecución de la función en un elemento de entrada. Esta segunda opción hace más fácil trabajar con grafos dinámicos cuya forma puede cambiar dependiendo de los valores calculados por la función. Por ejemplo, Tensorflow 1 usaba los grafos estáticos mientras que su versión más reciente TensorFlow 2 y PyTorch usan los grafos en tiempo real.

4.4. Problemas con el cálculo del gradiente

4.4.1. Desvanecimiento y explosión del gradiente

Siguiendo el hilo de la sección 3.4, vamos a ver dos problemas que surgen a la hora de entrenar modelos usando gradiente descendente y que pueden impedir la convergencia, pero con la diferencia de que estos están ligados únicamente al algoritmo de BP, es decir, a cómo se calcula el gradiente y no a cómo se usa en la búsqueda de soluciones.

Cuando entrenamos modelos muy profundos (con muchas capas ocultas), los gradientes tienen tendencia bien a volverse muy pequeños (desvanecimiento del gradiente) o bien a volverse muy grandes (explosión del gradiente) ya que la señal de error es pasada a través de una serie de capas que o lo amplifican o lo mitigan. Esto provoca que o bien se deje de actualizar el peso del cual se desvanece su gradiente o que el gradiente diverja en el otro caso [Hoc+01]. Para ver el problema con detalle, consideramos el gradiente de la función de pérdida con respecto a un nodo en la capa l:

$$\frac{\partial \mathcal{L}}{\partial z_l} = \frac{\partial \mathcal{L}}{\partial z_{l+1}} \frac{\partial z_{l+1}}{\partial z_l} = g_{l+1} J_l$$

donde $J_l = \frac{\partial z_{l+1}}{\partial z_l}$ es la matriz jacobiana, y $g_{l+1} = \frac{\partial \mathcal{L}}{\partial z_{l+1}}$ es el gradiente de la siguiente capa. Si J_l es constante entre capas, es claro que la contribución del gradiente de la capa final g_L a la capa l será $g_L J^{L-l}$. Entonces el comportamiento del sistema dependerá de los valores propios de J.

J es una matriz de valores reales pero generalmente no es simétrica, por lo que sus autovalores y autovectores pueden ser complejos, representando un comportamiento oscilatorio. Sea λ el radio espectral de J, que es el máximo

del valor absoluto de sus autovalores. Si es mayor que 1, el gradiente puede explotar; y si es menor que 1 su gradiente se puede desvanecer.

El problema de la explosión del gradiente se puede resolver de manera rápida y cómoda a través de acotar el gradiente con su magnitud y una constante $c \in \mathbb{R}^+$ en caso de que se vuelva muy grande

$$g' = min(1, \frac{c}{\|g\|}g)$$

De esta manera la norma de g' nunca puede ser mayor que c, pero el vector apunta siempre en la misma dirección que el gradiente.

También existen otras soluciones que además son aplicables al problema del desvanecimiento de gradiente, que no se soluciona de manera tan sencilla:

- Adaptar las funciones de activación para prevenir que el gradiente se vuelva muy grande o muy pequeño.
- Modificar la arquitectura del modelo para estandarizar las funciones de activación en cada capa, para que la distribución de las activaciones sobre el conjunto de datos permanezca constante durante el entrenamiento.
- Elegir cuidadosamente los valores iniciales de los pesos del modelo.

En la siguiente sección veremos detenidamente el último punto, ya que es la práctica más estandarizada. Existen además familias concretas de modelo que mitigan específicamente estos efectos con su arquitectura como son las ResNets, de las cuales hablaremos en la siguiente parte.

4.4.2. Inicialización de los pesos

La manera en la que inicializamos los pesos es una decisión importante a la hora de determinar cómo converge (y si lo hace) un modelo. La convergencia o no, su velocidad y la solución a la que se converge en el entrenamiento de un modelo mediante el algoritmo de gradiente descendente es muy sensible al punto inicial desde el que comenzamos la búsqueda de una solución. Hay que remarcar que esto sucede cuando la función de error no es convexa, pero como es el caso mayoritario, lo asumimos de manera general.

Basándonos en [GB10], donde se observa que muestrear parámetros de una distribución normal con varianza fija puede resultar en el problema de la explosión de gradiente, vamos a ver por qué ocurre esto y, a través de añadir restricciones para evitarlo vamos a llegar a las heurísticas de inicialización de pesos más comunes.

Consideramos el pase hacia delante en una neurona lineal sin capa de activación dada por $o_i = \sum_{j=1}^{n_{in}} w_{ij}x_j$. Suponemos $w_{ij} \sim \mathcal{N}(0, \sigma^2)$, con $\mathbb{E}[x_j] = 0$ y $\mathbb{V}[x_j] = \gamma^2$, donde asumimos x_j independiente de w_{ij}, y n_{in} es

el número de conexiones de entrada que recibe la neurona. La media y la varianza de la salida vienen dadas por

$$\mathbb{E}[o_i] = \sum_{j=1}^{n_{in}} \mathbb{E}[w_{ij}x_j] = \sum_{j=1}^{n_{in}} \mathbb{E}[w_{ij}]\mathbb{E}[x_j] = 0$$

$$\mathbb{V}[o_i] = \mathbb{E}[o_i^2] - (\mathbb{E}[o_i])^2 = \sum_{j=1}^{n_{in}} \mathbb{E}[w_{ij}^2 x_j^2] - 0 = \sum_{j=1}^{n_{in}} \mathbb{E}[w_{ij}^2]\mathbb{E}[x_j^2] = n_{in}\sigma^2\gamma^2.$$

Para evitar que la varianza diverja, necesitamos que $n_{in}\sigma^2$ se mantenga constante. Si consideramos el pase hacia atrás y realizamos un razonamiento análogo vemos que la varianza del gradiente puede explotar a menos que $n_{out}\sigma^2$ sea constante, donde n_{out} son las conexiones de salida de la neurona. Para cumplir con esos dos requisitos, imponemos $\frac{1}{2}(n_{in} + n_{out})\sigma^2 = 1$, o equivalentemente

$$\sigma^2 = \frac{2}{n_{in} + n_{out}}.$$

Esta se conoce como inicialización de Xavier o inicialización de Glorot [GB10]. Si usamos $\sigma^2 = \frac{1}{n_{in}}$ tenemos un caso especial conocida como la inicialización de LeCun, propuesta por Yann LeCun en 1990. Es equivalente a la inicialización de Glorot cuando $n_{in} = n_{out}$. Si usamos $\sigma^2 = \frac{2}{n_{in}}$, tenemos la llamada inicialización de He, propuesta por Kaiming He en [He+15b].

Cabe resaltar que no ha sido necesario usar una distribución Gaussiana. De hecho, las derivaciones de arriba funcionan en términos de la media y la varianza, y no hemos hecho suposiciones sobre si era Gaussiana.

Aunque hemos supuesto que se trataba de una neurona lineal sin función de activación que añada una componente no lineal, se conoce de manera empírica que estas técnicas son extensibles a unidades no lineales. La inicialización que elijamos dependerá mayoritariamente de la función de activación que usemos. Se conoce que para funciones de activación $ReLU$ funciona mejor la inicialización de He, para las funciones $SELU$ se recomienda la inicialización de LeCun y para las funciones lineales, logística, tangente hiperbólica y *Softmax* se recomienda el uso de la inicialización de Glorot [Mur22].

Parte I
Parte informática: enfoque clásico vs técnicas metaheurísticas

5. Introducción

Las redes neuronales profundas han revolucionado el campo de la inteligencia artificial, permitiendo avances significativos en varios campos como el procesamiento del lenguaje natural, reconocimiento de voz o visión por computador. Su capacidad para extraer patrones y representaciones complejas de grandes datasets y a un coste computacional muy eficiente en comparación con otras técnicas las han convertido en piedra angular de los sistemas modernos de aprendizaje automático. En el campo de la visión por computador, las *Convolutional Neural Networks* (ConvNets) o redes convolucionales se han erigido como la familia de modelos que consigue un rendimiento del estado del arte [GBC16].

Las ConvNets son un tipo de red neuronal que procesan datos en forma de grid. Se caracterizan porque tienen al menos una capa donde usan la operación de convolución en lugar de una matriz general de multiplicación. La convolución es un tipo de operación lineal que permite capturar representaciones espaciales aplicando un filtro a la entrada, detectando primero características de bajo nivel como bordes y texturas y aumentando el nivel de complejidad de la representación en las capas sucesivas [GBC16].

Las *Residual Nets* (ResNets) son una subfamilia de ConvNets que atajan el problema del desvanecimiento y explosión de gradiente (ver sección 4.4.1). Cuantas más capas tiene una red neuronal profunda más probable es que sufra este problema, ya que se arrastran más operaciones. Las ResNet crean bloques residuales donde se crea un atajo entre el inicio y el final del bloque en el que se suma la identidad al final del bloque, provocando que el gradiente pueda fluir de manera más efectiva durante el proceso de BP [He+15a].

El gradiente descendente es un algoritmo de aprendizaje que nos permite entrenar este tipo de modelos de forma eficiente, robusta, y con mucho rigor teórico. Sin embargo como ya se ha comentado en la parte anterior tiene algunas limitaciones, y estas se ven incrementadas cuantas más capas y más parámetros tiene el modelo que entrenamos. A parte de desarrollar mejoras en él con los optimizadores, se buscan nuevos algoritmos de aprendizaje que permitan evitar los problemas que presenta el gradiente descendente.

Una de estas aproximaciones son las técnicas metaheuristicas: estrategias de optimización basadas normalmente en componentes bio-inspirados y que son flexibles y adaptables a gran variedad de problemas. Ofrecen una solución cercana a la óptima en un tiempo razonable en muchos problemas cuya solución óptima es computacionalmente inalcanzable, como en problemas NP-Hard. Son técnicas iterativas que no ofrecen una garantía teórica de hallar una buena solución, pero a través de restricciones en el algoritmo se espera que lo haga [Mar+20].

Los más conocidos son los algoritmos evolutivos inspirados en la evolución genética. En ellos se genera una población aleatoria e iterativamente se realizan los procesos de: seleccionar los mejores individuos, recombinarlos

entre ellos, mutarlos para obtener más diversidad genética, y reemplazar los nuevos individuos en la población. Se pueden introducir modificaciones como criterios elitistas, en los que por ejemplo reemplazaríamos la población antigua sólo si fuera peor que la nueva. Los algoritmos evolutivos basados en *Differential Evolution* (DE) se especializan en optimización con parámetros reales y enfatizan la mutación, utilizando el operador de cruce a posteriori de ella. Alcanzan el rendimiento de estado del arte en optimización continua [TBS23].

Los algoritmos meméticos son una hibridación de las técnicas metaheurísticas con algoritmos de búsqueda local, que añaden el uso de información específica del problema. Combinan así la capacidad exploradora del espacio de soluciones que tienen los algoritmos evolutivos con la capacidad explotadora de la búsqueda local. El optimizador local se considera una etapa más dentro del proceso evolutivo y debe incluirse en él.

5.1. Motivación

El ajuste de pesos de un modelo es una de las partes más importantes en su desarrollo y por eso necesitamos de técnicas que nos ofrezcan cada vez mejores resultados a la vez que mayor eficiencia. No se trata de un problema sencillo ya que el número de parámetros de los modelos, es decir la dimensión del problema de optimización, tiene una tendencia que va rápidamente en aumento. Aunque el gradiente descente sea una estrategia muy buena hemos visto sus limitaciones, que nos incitan a intentar encontrar otras estrategias de aprendizaje. Las metaheurísticas toman cada vez un papel más protagonista en la optimización de problemas complejos y de grandes dimensiones a un bajo coste, lo que las sitúa como un posible sustituto.

Para la realización del presente TFG nos basaremos en el reciente paper de Daniel Molina y Francisco Herrera [Mar+20] donde se analiza el papel que juegan actualmente las metaheurísticas tanto en el en entrenamiento de los modelos, como en la selección de los hiperparámetros y de la topología de la red. Nos centraremos únicamente en el primer caso. En él se realiza también un experimento práctico comparativo entre Adam, un optimizador basado en el gradiente descendente, y diferentes versiones de SHADE-ILS, una técnica metaheurística basada en DE que hace uso de búsqueda local (técnica memética) que ofrece los mejores resultados actualmente en el entrenamiento de modelos a través de metaheurísticas.

En dicha publicación se realiza una revisión de la literatura en lo referente a las técnicas metaheurísticas para el entrenamiento de modelos, analizando los resultados de los paper más recientes y criticando de manera general la falta de rigor metodológico en la mayoría de ellos, sumado a que no resulta fácil realizar una comparación totalmente objetiva entre dos ténicas tan distintas como el gradiente descendente y los algoritmos bio-inspirados. Estas son algunas de las principales carencias en la literatura mencionadas

en dicho paper, junto a cómo las afrontaremos:

- Falta de homogeneidad en los *datasets* usados y las tareas a resolver, lo que no permite una comparación objetiva entre diferentes experimentos. Usaremos cuando proceda los mismos *datasets* con las mismas condiciones experimentales.
- Se usan escalas de modelos no realistas para probar algoritmos bioinspirados, que normalmente constan de unos pocos miles de parámetros. En el presente TFG la gama de modelos en función de sus parámetros irá desde los 2 mil parámetros hasta rondar los 1.5M de parámetros, rango elegido en concordancia con el punto anterior.
- Malas prácticas metodológicas para la comparación de algoritmos, por ejemplo, de manera generalizada se suelen comparar varias técnicas metaheurísticas entre ellas, sin que se comparen con el algoritmo de gradiente descendente. En la experimentación usaremos varias técnicas metaheurísticas y varios optimizadores de gradiente descendente.
- Aunque es bien sabido que los algoritmos metaheurísticos requieren de muchos más recursos computacionales que los clásicos, no se realizan análisis de complejidad, de manera que no se establece una equivalencia o comparación objetiva en el rendimiento. En la experimentación asignamos deliberadamente más recursos al entrenamiento con metaheurísticas, y establecemos una equivalencia de manera que podamos realizar más adelante una comparación de la complejidad.

Cabe destacar además que la práctica totalidad de técnicas metaheurísticas se prueban con ConvNets y RNNs, mientras que las hibridaciones de estas técnicas con el gradiente descendente se estudian mayoritariamente en ConvNets.

5.2. Objetivos

Aunque la actual superioridad del entrenamiento de modelos de aprendizaje profundo con el algoritmo del gradiente descendente, en términos de resultados y coste computacional, se evidencie en la literatura, las carencias en la misma que hemos visto en el punto anterior hacen necesaria más experimentación en las condiciones y con el rigor adecuados. El objetivo de este TFG es ofrecer una batería experimental amplia, que permita comparar de manera objetiva las técnicas clásicas, las metaheurísticas y su hibridación, y que además pueda ser comparada de manera objetiva con otros experimentos.

De forma conceptual dividiremos la experimentación en dos partes, siendo común a las dos las técnicas usadas para el entrenamiento de modelos, que

son: Adam, NAG y RMSProp en el caso de las técnicas clásicas y SHADE, SHADE-ILS, SHADE-GD y SHADE-ILS-GD en el caso de las metaheurísticas. Destacamos que estas dos últimas técnicas son propuesta propia, y no se han encontrado experimentos con ellas en la literatura. Aclarar que, a no ser que se especifique explícitamente, cuando nos referimos a técnicas metaheurísticas englobamos las técnicas estándar y las hibridadas. En la primera afrontaremos cuatro tareas con datasets tabulares, entrenando para cada tarea varios modelos MLP de diverso número de capas y de parámetros con cada una de las técnicas antes mencionadas. En la segunda tendremos tres tareas de reconocimiento de imágenes usados en [**y**] el mismo entorno experimental, entrenando para cada tarea tres modelos de ConvNets de distinto número de capas y parámetros, con las técnicas antes mencionadas. De esta manera, definimos las conclusiones que queremos obtener de la experimentación:

- *P1. ¿Cuál es el rendimiento de las técnicas metaheurísticas en los modelos MLP?*. Como la literatura no es extensa en este ámbito, queremos comprobar si existe alguna diferencia al usar técnicas metaheurísticas para entrenar MLP y ConvNets, comparando los resultados de éstas técnicas con las del gradiente descendente en cada familia de modelos.
- *P2. ¿Existe diferencia para tareas de clasificación y regresión?*. Para una comparación objetiva, compararemos los resultados de las tareas asociadas a los cuatro *datasets* tabulares, donde dos son tareas de clasificación y dos de regresión.
- *P3. Análisis del rendimiento de las técnicas metaheurísticas.* Aunque conocemos la superioridad del gradiente descendente, queremos analizar de manera minuciosa cómo influyen las siguientes tres variantes en el rendimiento de estas técnicas en comparación con él:

  1. Complejidad de la tarea
  2. Tamaño del modelo
  3. Tamaño del *dataset*

- *P4. Análisis de la complejidad computacional de las metaheurísticas.* En comparación con el gradiente descendente y a partir de las tres variantes anteriores, en base al concepto de época.
- *P5. Comparativa de las técnicas híbridas propuestas.* Analizamos la hibridación de dos de las técnicas que obtienen mejores resultados en el entrenamiento de modelos con el gradiente descendente. Analizando si mejoran y en qué lo hacen a sus versiones sin gradiente descendente.

Como aclaración, cuando comparemos dos técnicas lo haremos en base a estos tres criterios:

- Minimización de la función de pérdida.
- Generalización del modelo entrenado.
- Tiempo requerido para el entrenamiento.

Destacar que podemos responder a las preguntas P3 y P4 gracias a que sobre cada tarea entrenamos un total de $N_{modelos} \times N_{tecnicas}$ veces, lo que equivaldría a 28 modelos entrenados para cada tarea en la primera parte de la experimentación y 21 modelos por tarea en la segunda.

6. Fundamentos previos

En esta sección se detallan los conceptos, familias de modelos y algoritmos necesarios para la elaboración del trabajo posterior. Se usarán los conocimientos aprendidos en las asignaturas de Aprendizaje Automático, Visión por Computador y Metaheurísticas. Además de información extraída de los artículos de publicación originales cuando corresponda, se usan las siguientes fuentes: [GBC16] y [Fei24] para las secciones 6.1 y 6.2; [Zha+23] para las secciones 6.3 y ; [GP10] y [PSL05] para la sección 6.6; [**numerical˙optimization**] y [Fei24] para la sección **??**; [Zha+23] y [GBC16] para 6.5.

6.1. Redes neuronales y aprendizaje profundo

6.1.1. Red neuronal

Una red neuronal es un modelo computacional inspirado en la manera en la que las neuronas se conectan en el cerebro humano procesando la información. Consiste en capas interconectadas con nodos llamados neuronas, donde cada conexión tiene un peso asociado. Cada neurona normalmente aplica una función no linear, llamada función de activación, a la suma ponderada de las entradas de la capa anterior, permitiendo al modelo aprender relaciones complejas. Este tipo de redes se denominan totalmente conectadas. Sus componentes básicos son:

- Capa de entrada: Recibe la información.
- Capas ocultas: son las capas intermedias, que realizan los cálculos.
- Capa de salida: produce la salida del modelo.

El ejemplo más sencillo es el Perceptrón [Ros58], una red neuronal de una sola capa oculta y una sola neurona como podemos ver en la imagen 5. Es un clasificador lineal, es decir, sólo puede resolver probleams cuyos datos sean linealmente separables. En su versión original su función de activación f es la función signo. Para problemas más complejos que no sean lineales necesitamos usar redes neuronales con varias capas.

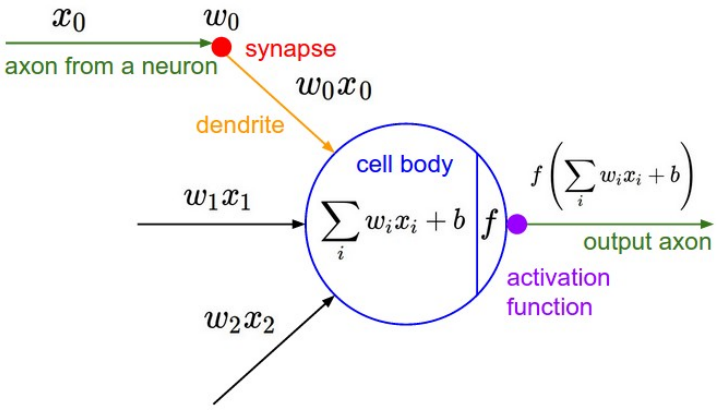

Figura 5: Esquema del modelo de una neurona con tres conexiones de entrada. Obtenida de [Fei24]

6.1.2. Aprendizaje profundo y redes neuronales profundas

Las redes neuronales profundas son redes neuronales que tienen varias capas ocultas, generalmente más de dos, aunque lo normal en este tipo de modelos es que se usen alrededor de 10 o 20. La profundidad de una red viene determinada por el número de capas ocultas que tiene, y ésta permite aprender representaciones jerárquicas de los datos, lo que permite a las redes neuronales profundas capturar patrones más complejos en los datos en comparación con redes menos profundas.

El aprendizaje profundo es una subrama del aprendizaje automático que se centra mayoritariamente en las redes neuronales profundas. Generalmente se usan modelos muy profundos y grandes cantidades de datos para resolver tareas complejas. Esto supone que se requiere de mucho poder computacional y de algoritmos avanzados para optimizar sus parámetros. Este tipo de modelos obtiene un gran rendimiento en tareas como el procesamiento del lenguaje natural, reconocimiento de voz o visión por computador. Los MLP son el ejemplo más clásico.

6.1.3. MLP

Los perceptrones multicapa o *Multilayer Perceptrons* (MLP) son una versión más compleja del Perceptrón, que cuenta con varias capas ocultas y varias neuronas en cada una. Son capaces de procesar datos no linealmente separables ya que pueden aprender información más compleja. Sus capas son totalmente conectadas y la información fluye sólo hacia delante a la hora de hacer una predicción con el modelo.

En su definición original, usan la función signo como función de activación en todas las neuronas y sólo se usan para tareas de clasificación. Sin embargo actualmente son sinónimo de redes profundas totalmente conecta-

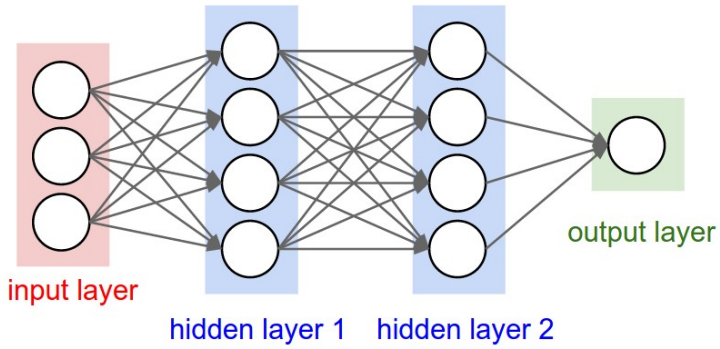

Figura 6: Red neuronal de tres capas (sin contar la capa de entrada) con dos capas ocultas de cuatro neuronas cada una y una capa de salida con una neurona. Destacar que no hay conexiones de una neurona con varias capas. Obtenida de [Fei24]

das, siendo usadas con cualquier tipo de función de activación y para tareas de clasificación o regresión.

6.2. ConvNets

Las ConvNets o redes convolucionales son una familia de modelos de aprendizaje profundo usadas en la visión por computador. Obtienen un rendimiento al nivel del estado del arte en tareas como el reconocimiento de imágenes o la detección de objetos. Se caracterizan por tener una o varias capas (al menos una) basadas en convoluciones para luego tener una o varias capas totalmente conectadas. Las primeras sirven como extractores de características que capturan propiedades espaciales de las imágenes, mientras que las segundas sirven para clasificación. Comenzaron a ganar popularidad con el modelo LeNet-5 [Lec+98] presentado por Yann LeCun en 1998, consiguiendo superar en rendimiento al resto de técnicas hasta la fecha en el reconocimiento de dígitos manuscritos (MNIST).

6.2.1. Operación de convolución

La convolución es una operación matemática que expresa la relación entre la entrada, la salida y la respuesta del sistema a los impulsos. En el contexto del procesamiento de señales, la convolución combina dos señales para producir una tercera. Se define matemáticamente para funciones continuas como

$$(f * g)(t) = \int_{-\infty}^{\infty} f(x)g(t-x)dx.$$

Para funciones discretas se define como

$$(f * g)(t) = \sum_{x=-\infty}^{\infty} f(x)g(t-x).$$

Nos referimos a f como la entrada y a g como el núcleo o filtro. En el aprendizaje automático la entrada suele ser un tensor de datos y el filtro un tensor de parámetros que adaptamos con el algoritmo de aprendizaje. Ambos son de dimensión finita y asumimos que su valor es 0 en todos los puntos donde no almacenamos su valor. Por tanto en la práctica podemos implementar la sumatoria infinita como una suma finita de los elementos de un vector. Si usamos una imagen bidimensional I como entrada, seguramente usaremos un filtro bidimensional K:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n).$$

Las convoluciones cumplen las siguientes propiedades:

- **Conmutatividad**: $f * g = g * f$.
- **Asociatividad**: $f * (g * h) = (f * g) * h$.
- **Distributividad**: $f * (g + h) = (f * g) + (f * h)$.

La propiedad conmutativa es útil a nivel matemático pero no es demasiado práctica en la implementación de una red neuronal. Por ello muchas librerías de aprendizaje automático optan por implementar la función llamada *cross-relation* en lugar de la convolución, volteando el núcleo como podemos ver en 7.

$$C(i,j) = (I \cdot K)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n).$$

Al igual que hacen estas librerías, llamamos a estas dos operaciones indistintamente convolución, ya que en el contexto del aprendizaje de modelos no habrá diferencia, porque el algoritmo de aprendizaje obtendrá los mismos valores para el núcleo y sólo variará su posición. Podemos considerar la convolución como una multiplicación matricial donde la matriz tiene restricciones en muchas posiciones las cuales deben tener el mismo valor.

6.2.2. Capa Convolucional

Las capas convolucionales son las más importantes en la arquitectura de una ConvNet. Las claves de su gran rendimiento en el campo de la visión por computador son: la conectividad local, la disposición espacial y compartir parámetros.

La conectividad local hace referencia a las conexiones de las neuronas. En entradas de alta dimensionalidad como imágenes no es práctico conectar

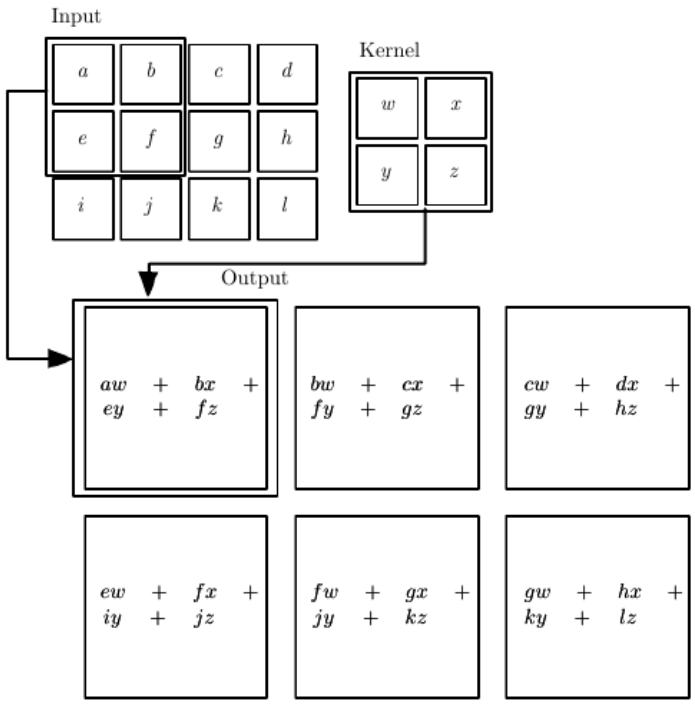

Figura 7: Convolución 2D sin voltear el filtro (*cross-relation*). Obtenida de [GBC16]

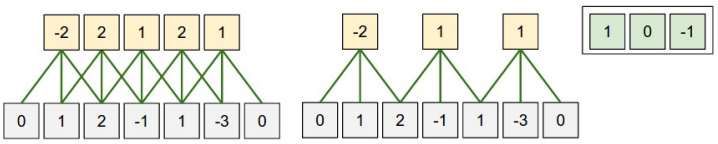

Figura 8: Ilustración de la disposición espacial. El tamaño de la entrada es $W = 5$ (vector gris) y el del filtro $F = 3$ (vector verde), sin usar padding $P = 1$. En el ejemplo de la izquierda se usa *stride* $S = 1$, mientras que en el de la derecha se usa $S = 2$, obteniendo tamaños de salida de 5 y 3, respectivamente. Estos tamaños se pueden calcular según la fórmula 18. Obtenida de [Fei24]

una neurona con todas las neuronas del volumen anterior, por tanto cada neurona se conecta solo a una región local del volumen de entrada. Esto viene determinado por un hiperparámetro llamado campo receptivo, que es el tamaño del filtro que aplicamos. Mientras que las conexiones son locales en el espacio 2D (ancho y altura), siempre abarcan toda la profundidad del volumen de entrada.

Con la disposición espacial nos referimos al tamaño del volumen de salida y cómo están organizadas estas neuronas. Hay tres hiperparámetros con los que controlamos esto:

1. Profundidad del volumen de salida: corresponde a la cantidad de filtros que queremos usar.

2. *Stride*: Indica el número de píxeles (hablando en términos de imágenes) que usamos para desplazar el filtro al realizar la convolución.

3. *Padding*: A veces, para mantener la dimensión de la salida es conveniente rellenar el borde de la entrada con ceros.

Las dimensiones del volumen de salida podemos calcularlas como una función dependiente del tamaño del volumen de entrada W, el tamaño del filtro F, el stride S y el padding P que queramos aplicar. La fórmula es la siguiente:

$$\frac{W - F - 2P}{S + 1}. \tag{18}$$

Esta nos dará las dimensiones en ancho y altura del volumen de salida como podemos ver en la imagen 8, y su profundidad vendrá totalmente determinada por el número de filtros que queramos usar.

Compartir parámetros en una ConvNet nos permite reducir el número de éstos, reduciendo el coste del entrenamiento. Se basa en la suposición de que si una característica es útil en una posición espacial (x, y) tambien lo será en otr (x', y'). En un volumen $W \times H \times D$, en lugar de que cada neurona

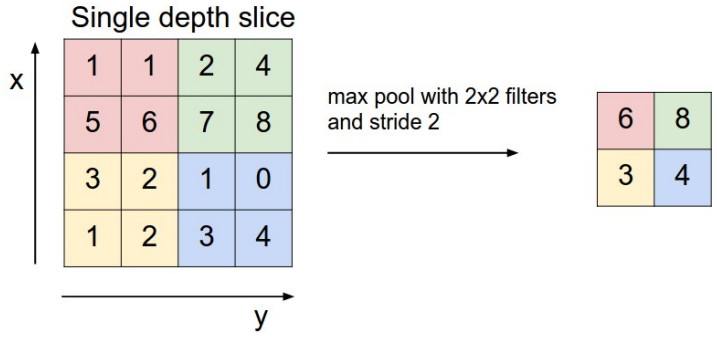

Figura 9: Capa de pooling con un filtro de tamaño 2×2 y stride 2. Obtenida de [Fei24]

tenga su conjuntos de pesos, tenemos D conjuntos de pesos, reduciendo drásticamente su número.

6.2.3. Capa Pooling

Su función es reducir progresivamente el tamaño de la representación para reducir el número de parámetros y la carga computacional en la red, además de controlar el sobreajuste. Opera independientemente a lo largo de la profundidad del volumen, usando la operación máximo. La opción más común es usar filtros 2×2 con un stride de 2 como se observa en la figura 9. La dimensión de la profundidad permanece intacta. Existen otros tipos de pooling, por ejemplo realizar la media entre los elementos, pero esta opción fue dejando paso a la de seleccionar el máximo ya que obtiene mejores resultados en la práctica.

6.2.4. Capa *Batch Normalization*

Batch Normalization (BatchNorm) [IS15] es un método de reparametrización adaptativa motivado por la dificultad de entrenar modelos muy profundos. En una capa de BatchNorm se estandariza la entrada a través de escalarla y trasladarla, lo que ayuda a estabilizar y acelerar el entrenamiento. Para cada mini-batch, se realiza el siguiente proceso:

1. Se calcula la media $\nu_B = \frac{1}{m}\sum_{i=1}^{m} x_i$ y la varianza $\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \nu_B)^2$.

2. Se normaliza la entrada: $\hat{x}_i = \frac{x_i - \nu_B}{\sqrt{\sigma_B^2 + \epsilon}}$, donde ϵ es una constante pequeña para evitar la división por 0.

3. Se escala y se traslada la entrada normalizada: $y_i = \gamma\hat{x}_i + \beta$, donde γ y β son parámetros aprendibles por el modelo.

Normalizando la entrada conseguimos hacer el proceso de entrenamiento más estable y ayuda a evitar el problema de la exlosión o desvanecimiento del gradiente. También proporciona flexibilidad y mejora el rendimiento al reescalar y trasladar la entrada, y que esto dependa de parámetros aprendibles.

6.2.5. Capa FC

Al final de las redes convolucionales lo más común es encontrarnos una o varias capas totalmente conectadas (FC, fully connected), es decir, que cada neurona está conectada a todas las neuronas de la capa anterior, de la misma manera que ocurre en un MLP. Esta parte de la red permite clasificar las características extraídas por las caps convolucionales.

6.3. ResNets

Las ResNets (Residual Networks) [He+15a] son una familia de modelos dentro de las ConvNets. Su característica principal es que usan bloques residuales, que agrupan varias capas en los cuales se suma la identidad (la entrada al bloque) a la salida del bloque. Que las redes neuronales profundas aprendan esta función identidad previene el problema de la degradación, es decir, que el rendimiento de la red decaiga a medida que aumenta el número de capas. Esto puede surgir por varias causas como la inicialización de los pesos, la función de activación o el desvanecimiento/explosión del gradiente.

6.3.1. Bloques residuales

La función identidad se aprende a través de las conexiones residuales, que conectan el inicio y el final de los bloques residuales pasando la identidad. Estas conexiones además permiten aliviar el problema del desvanecimiento de gradiente. Vamos a centrarnos en una red neuronal de manera local, como se muestra en la figura 10.

Si la función identidad $f(x) = x$ es el mapeo subyacente deseado, la función residual equivale a $g(x) = 0$ y, por tanto, es más fácil de aprender: sólo tenemos que llevar a cero los pesos y bias de la últma capa de pesos dentro de la línea de puntos. Con los bloques residuales, las entradas pueden propagarse más rápidamente a través de las conexiones residuales entre capas.

Para ello necesitamos que la entrada y la salida del bloque tengan el mismo tamaño. Si reducimos la dimensionalidad de la entrada o aumentamos el número de filtros entonces deberemos modificar la entrada a través de convoluciones 1×1 para que tenga el mismo tamaño que la salida, como se muestra en la figura 13.

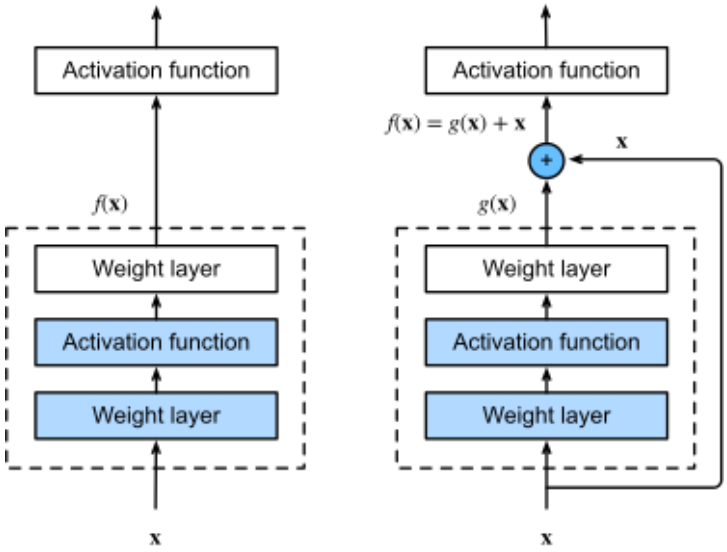

Figura 10: En un bloque convolucional estándar (izquierda), la parte dentro de la línea de puntos debe aprender directamente la función $f(x)$. En un bloque residual (derecha), la parte dentro de la línea de puntos debe aprender la función residual $g(x) = f(x) - x$, haciendo que la función identidad $f(x) = x$ sea más fácil de aprender. Obtenida de [Zha+23]

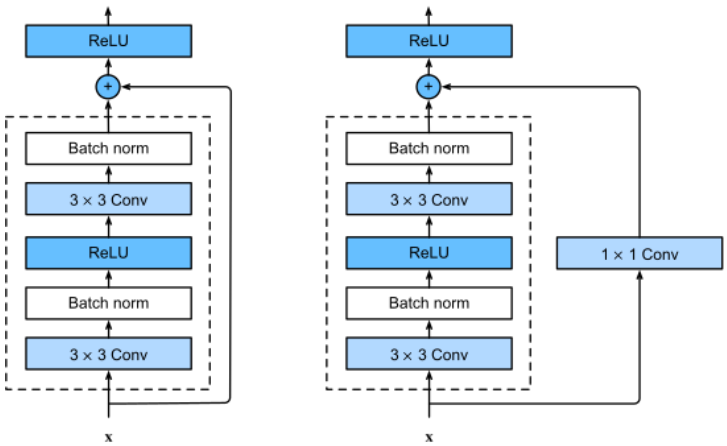

Figura 11: Bloque residual donde la entrada tiene la misma dimensión que la salida (izquierda), y bloque residual donde se transforma el tamaño de la entrada a través de convoluciones 1×1 para que tengan el mismo tamaño (derecha). Obtenida de [Zha+23]

6.3.2. Convoluciones 1x1

Los bloques residuales nos permiten aumentar la profundidad de la red evitando ciertos problemas asociados, pero al añadir más capas estamos aumentando considerablemente el número de parámetros. Las convoluciones con tamaño de filtro 1×1 son una herramienta poderosa para reducir el número de parámetros manteniendo la expresividad de la red. Fueron presentadas en [LCY14]. Este tipo de convoluciones se realizan antes de realizar la convolución requerida, de manera que la dividamos en dos, una con tamaño de filtro 1 y la otra con el tamaño original.

Ejemplo 6.1 *Para ver la diferencia en el número de parámetros al usar esta herramienta, calcularemos los parámetros necesarios para realizar una convolución en el caso de tener* $C = 256$ *canales de entrada,* $O = 512$ *canales de salida y tamaño del filtro* $F = 3$*.*

Si no usamos convoluciones 1×1*, tendríamos* $P = F \times F \times C \times O + O =$ 1.180.160 *parámetros.*

Usándolas debemos elegir un tamaño de filtro intermedio, por ejemplo $C' = 64$*. Aplicamos primero la convolución* 1×1*:* $P'_1 = 1 \times 1 \times C \times C' + C' =$ 16.448 *parámetros. A continuación realizamos la convolución con el tamaño de filtro original:* $P'_2 : F \times F \times C' \times O + O = 295.424$*. En total, sumando las dos capas tendríamos* 311.872 *parámetros, unas cuatro veces menos que en el caso anterior.*

6.4. Política de ciclos de Leslie

En [Smi17] se presenta una estrategia que modifica la tasa de aprendizaje de manera cíclica. En lugar de usar valores fijos o decrecientes, se divide el entrenamiento en ciclos, en los cuales la primera etapa se usa para aumentar la tasa de aprendizaje y la segunda para disminuirla, dentro de unos valores razonables.

Más adelante en [Smi18] el mismo autor propone una extensión de esta estrategia en la que se usa un único ciclo durante todo el entrenamiento. Es decir que la primera mitad del entrenamiento aumentamos la tasa de aprendizaje de manera lineal y en la segunda mitad la hacemos decrecer de igual manera hasta su valor original. Además se propone modificar el momento (en el caso de que se use) de manera inversa a la tasa de aprendizaje. Esto se conoce como la política de ciclos de Leslie o *One Cycle Policy*.

Con esta estrategia se consigue en el entrenamiento una super-convergencia [ST18], es decir se agiliza el entrenamiento un orden de magnitud más rápido que con las estrategias convencionales. Además se comprobó que usar tasas de aprendizajes altas en el valor máximo de la política de ciclos tiene un efecto de regularización en el entrenamiento.

6.5. Optimizadores de gradiente descendente

Con el objetivo de intentar abordar los principales problemas del algoritmo de aprendizaje del gradiente descendente se han propuesto en la literatura diversas modificaciones, añadiendo términos en la regla de actualización de los pesos. Existen optimizadores de primer y segundo orden, en función de si hacen uso sólo de la información del gradiente o también de la matriz Hessiana. Vamos a ver en esta sección únicamente los de primer orden, y veremos un método de segundo orden en la sección siguiente pero como parte de una metaheurística memética. Existen tres enfoques en este ámbito: el uso de momento, learning rates adaptativos y la combinación de los dos anteriores. De cada uno hacemos hincapié en el que vamos a usar en el presente TFG, en orden respectivo: NAG, RMSProp y Adam.

6.5.1. NAG

El algoritmo del gradiente descendente es problemático en regiones de la función de error donde una dimensión tiene mucha más pendiente que otra, que son comunes alrededor de óptimos locales. En estos escenarios el algoritmo oscila y realiza poco progreso real. El momento [Qia99] es un método que acelera al algoritmo en la dirección relevante y compensa las oscilaciones, como podemos ver en la figura **??**. Esto se realiza añadiendo una fracción γ del vector gradiente de la última iteración al vector gradiente actual.

$$\begin{aligned} v_t &= \gamma v_{t-1} + \eta \nabla C(W_t) \\ W_{t+1} &= W_t - v_t. \end{aligned}$$

El valor de γ se sitúa normalmente alrededor de 0.9. El término del momento se incrementa en las dimensiones en las que el gradiente apunta en la misma dirección y se reduce en las que el gradiente cambia de dirección, consiguiendo una convergencia más rápida y estable. Dotamos al algoritmo de cierta inercia para reducir la brusquedad en los cambios de dirección.

El optimizador NAG [Nes83] modifica esta idea de manera que podamos "predecir" a dónde nos lleva esa inercia. A la hora de calcular el gradiente de la función de coste, no lo hacemos respecto a los parámetros, sino respecto a una aproximación de los parámetros tras la iteración actual, de manera que podamos saber de forma aproximada dónde nos encontraremos después de actualizar los pesos. Se puede interpretar como una corrección del método de momento original. El valor del momento se sitúa también alrededor de 0.9.

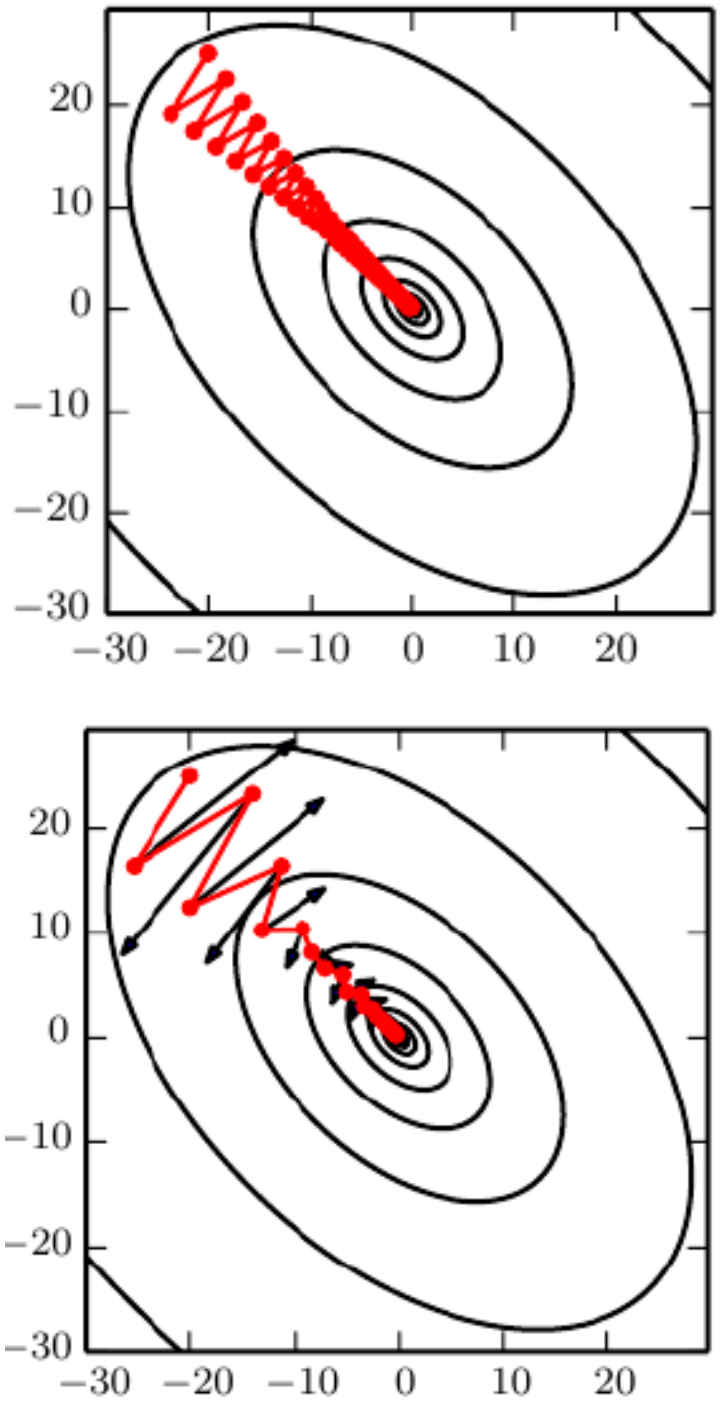

Figura 12: Comparación entre el algoritmo de gradiente descendente sin usar el método del momento (izquierda) y usándolo (derecha). Vemos que en la figura de la derecha se diferencia la dirección del gradiente (flechas negras) y el camino seguido por el algoritmo en rojo. Imágenes obtenidas de [GBC16].

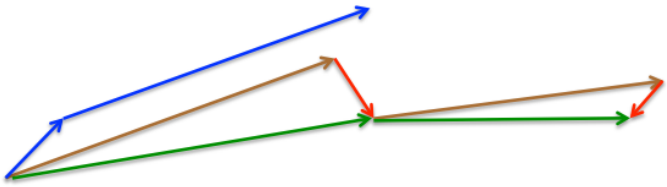

Figura 13: Comparación entre los métodos del momento original (vectores azules) y el momento de Nesterov. En este último, primero realizamos un salto grande en la dirección del gradiente acumulado (vector marrrón) para luego medir el gradiente de la posición al acabar el salto y realizar una corrección (vector rojo). La flecha verde indica la posición final corregida donde acaba una iteración del método NAG. Obtenida de `http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf`

$$v_t = \gamma v_{t-1} + \eta \nabla C(W - v_t)$$
$$W_{t+1} = W_t - v_t.$$

Mientras que usando el optimizador momento original primero calculamos el gradiente y luego realizamos un salto grande en la dirección del gradiente acumulado, NAG primero realiza un salto grande en la dirección del gradiente acumulado, mide el gradiente y despúes realiza una corrección. Esta comparación se ilustra en la figura **??**. Esta estrategia previene al algoritmo de avanzar demasiado rápido.

6.5.2. RMSProp

RMSProp (Root Mean Square Propagation) es un optimizador de primer orden que introduce *learning rates* adaptativos. Presentado por Geoff Hinton en su curso [Hin12], la fórmula de actualización de los pesos es la siguiente:

$$E[\nabla C(W_t)] = 0.9E[\nabla C(W_{t-1})] + 0.1\nabla C(W_t)^2$$
$$W_{t+1} = W - t - \frac{\eta}{\sqrt{E[\nabla C(W_t)^2]_t + \epsilon}} \nabla C(W_t)$$

Donde $E[g^2]_t$ es la media móvil en la iteración t, que depende solamente de la iteración anterior y del gradiente actual. Las operaciones anteriores se realizan elemento a elemento, es decir, cada peso recibe una actualización con un factor personalizado. Actualizando los pesos de esta forma, cuando el gradiente es grande en una dirección entonces el factor de ese peso será pequeño, evitando oscilaciones; mientras que si en otra dirección el gradiente

es relativamente pequeño entonces el valor será grande, acelerando el proceso de entrenamiento.

Existen otros optimizadores que usan tasas de aprendizaje variables como Adagrad [DHS11], sus diferencias residen en la ventana de iteraciones y el cálculo del factor de multiplicación del peso. En este optimizador sólo se tienen en cuenta la iteración pasada y la actual, mientras que el uso de una media exponencial decreciente permite que los *learning rates* no se vuelvan demasiado pequeños.

6.5.3. Adam

Adaptative Moment Estimation (Adam) [KB17] es otro método que calcula tasas de aprendizaje adaptativas para cada parámetro. Además mantiene una media exponencial decreciente de gradientes de iteraciones anteriores m_t similar al momento.

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1)\nabla C(W)$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2)\nabla C(W)^2.$$

m_t y v_t son estimaciones del momento de primer orden (media) y de segundo orden (varianza no centrada) de los gradientes respectivamente. Son inicializados como vectores de 0, por lo que sus autores encontraron que tenían un sesgo al 0, especialmente durante las primeras iteraciones y cuando β_1 y β_2 son próximos a 1. Por tanto se calculan nuevas variables corrigiendo el sesgo:

$$\hat{m}_t = \frac{m_t}{a - \beta_1}$$
$$\hat{v}_t = \frac{v_t}{1 - \beta_2}.$$

Ahora se usan para ajustar los parámetros como hemos visto en el optimizador anterior:

$$W_{t+1} = W_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}\hat{m}_t$$

Los autores proponen valores por defecto de 0.9 para β_1, 0.999 para β_2 y 10^{-8} para ϵ.

6.6. Metaheurísticas

En su definición original las metaheurísticas son métodos que combinan técnicas de mejora local con estrategias de alto nivel para crear un proceso capaz de escapar óptimos locales y realizar una búsqueda robusta del

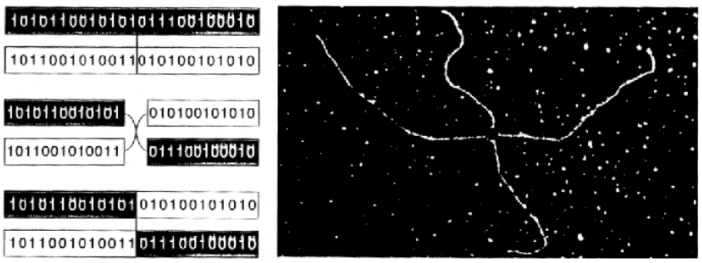

Figura 14: Famosa imagen esquemática del operador de cruce en un punto para dos vectores binarios de soluciones (izquierda), comparando el proceso con la recombinación genética de cromosomas (derecha). Obtenida de [Hol92]

espacio de soluciones. Aunque no hay garantía teórica de que puedan encontrar la solución óptima, su rendimiento es muy superior en algunos casos al de algoritmos exactos que requieren demasiado tiempo para completar su ejecución, especialmente en problemas complejos del mundo real. En problemas NP-Hard por ejemplo se prioriza el uso de metaheurísticas que dan una solución cercana a la óptima en un tiempo mucho menor que algoritmos exactos.

Podemos clasificar a las metaheurísticas en dos grandes grupos en función de cómo se realiza la búsqueda por el espacio de soluciones: basadas en trayectorias y basadas en poblaciones. En las primeras el proceso de búsqueda se caracteriza por realizar una trayectoria en el espacio de búsqueda, que puede ser visto como la evolución en tiempo discreto de un sistema dinámico. En las metaheurísticas basadas en poblaciones, en cada iteración hay un conjunto de soluciones que interactúan entre sí. Nos centraremos en este último tipo ya que es el que vamos a usar.

6.6.1. Metaheurísticas basadas en poblaciones

Son técnicas de optimización probabilística que con frecuencia mejoran a otros métodos clásicos, e intentan imitar el mecanismo de evolución de la naturaleza a través de similitudes con la genética, como se ilustra en la imagen 14. Tiene un conjunto de soluciones denominado población, donde cada solución se llama individuo, y son generados de forma aleatoria. En cada iteración o generación, estos individuos se recombinan entre sí para intentar obtener mejores soluciones cuyo rendimiento es medido con una función objetivo. Las etapas de cada generación se pueden ver esquemáticamente en la figura 15, y son:

- Selección: se elige una parte de la población actual, normalmente con criterios elitistas (se elige a los mejores) aunque introduciendo cierta

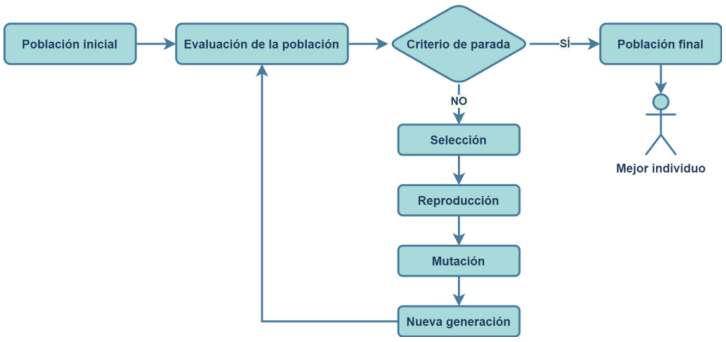

Figura 15: Esquema de la ejecución de un algoritmo basado en poblaciones donde se observan las etapas de cada generación. Obtenida de `https://blogs.imf-formacion.com/blog/tecnologia/`

aleatoriedad. Si el número de individuos elegidos es igual al tamaño de la población, hablamos de un modelo generacional, mientras que si es menor hablamos de un modelo estacionario.

- Cruce: Los individuos seleccionados se agrupan por parejas y se combinan a través del operador de cruce. Las soluciones resultantes se denominan hijos. Operadores comunes son el cruce en un punto, el cruce en dos puntos y el cruce uniforme. Podemos ver un ejemplo gráfico y su semejanza con la genética en la imagen IMAGEN
- Mutación: A los hijos se les aplican cambios aleatorios en sus valores para mantener cierta diversidad genética en la población.
- Reemplazo: Se reemplaza la población actual con la nueva generación. Podemos reemplazarla entera o aplicar criterios elitistas, como reemplazar sólo con los mejores o reemplazar sólo si la nueva generación es mejor que la anterior.
- Terminación: se comprueba si se cumple la condición de parada. Criterios comunes son un número máximo de iteraciones o la convergencia de la población (falta de mejoras entre generaciones).

Los criterios elitistas hacen que la convergencia sea más rápida, pero podemos caer en una convergencia prematura por la falta de diversidad que conllevan estos criterios, de manera que nuestro algoritmo pare antes de encontrar una solución lo suficientemente buena.

6.6.2. Differential Evolution

Los algoritmos de DE [SP97] son modelos basados en poblaciones que son particularmente efectivos para problemas de optimización continuos.

Enfatizan la mutación y la realizan antes de aplicar el operador de cruce. Usan los parámetros factor de mutación F y probabilidad de cruce C_r. Las etapas que varían, descritas en el orden que se realizan en cada generación, son las siguientes:

- Operador de mutación: Para cada solución x_i de la población, se genera un vector mutante v_i a partir de la siguiente expresión:

$$v_i = x_{r1} + F \cdot (x_{r2} - x_{r3}).$$

  Donde x_{r1}, x_{r2} y x_{r3} son individuos seleccinados aleatoriamente con las restricciones de que $x_i \neq x_{rj}$ y $x_{rj} \neq x_{rj'}$ con $j \in \{1, 2, 3\}$. F se suele situar en la práctica ente 0 y 2.

- Cruce: se realiza el cruce entre el vector solución de partida x_i y el vector mutante v_i para generar el vector de prueba u_i. Se usa el cruce binomial:

$$u_{ij} = \begin{cases} v_{ij} & \text{if } \text{rand}_j(0,1) \leq C_r \\ x_{ij} & \text{otherwise} \end{cases}$$

  donde $\text{rand}_j(0, 1)$ es generado aleatoriamente con una distribución uniforme entre 0 y 1 para cada componente j.

- Selección: se compara el valor de la función objetivo de los vectores iniciales con el de los vectores de prueba correspondientes, y el que tenga mayor valor pasa a la generación siguiente.

En el pseudocódigo 4 podemos apreciar el cambio de orden en las etapas de cada generación con respecto al esquema general de los algoritmos basados en poblaciones que veíamos en la figura 15

6.6.3. L-BFGS-B

??

El método L-BFGS-B (Limited-memory Broyden-Fletcher-Goldfarb-Shanno with Box constraints) [Byr+95] es un algoritmo Quasi-Newton, es decir, un algoritmo de optimización iterativo. Los métodos de Newton usan la matriz Hessiana de la función a optimizar para usar más información del problema y ofrecer una convergencia más rápida y estable. Para problemas complejos de dimensionalidad elevada, calcular la Hessiana en cada paso es una tarea computacionalmente inabarcable, y los métodos de Quasi-Newton implementan una aproximación de la Hessiana para rebajar esta carga computacional. Estos métodos se diferencian entre ellos principalmente en la forma de aproximar la matriz Hessiana.

Algorithm 4 Esquema general de DE

```
t := 0
Inicializar Pob_t
Evaluar x   ∀x ∈ Pob_t
while No se cumpla condición de parada do
    t := t + 1
    Mutar Pob_t para obtener Pob'
    Recombinar Pob' y Pob para obtener Pob"
    Evaluar Pob"
    Reemplazar Pob_t a partir de Pob" y Pob_{t-1}
end while
    return x_i ∈ Pob_t : f(x_i) ≤ f(x_j)
```

Uno de los métodos Quasi-Newton más populares es BFGS [DS96], que usa una aproximación de la Hessiana de forma que mantiene su propiedad de definida positiva, lo que asegura una convergencia estable. Sin embargo, aunque se reduce el coste computacional, para almacenar la matriz Hessiana se requiere demasiada memoria. El método L-BFGS [LN89], en lugar de guardar una matriz con $n \times n$ aproximaciones, guarda únicamente un vector de tamaño n que guarda todas las aproximaciones de manera implícita. Esta variante está diseñada para problemas de alta dimensionalidad, y produce resultados similares a su versión sin la memoria limitada.

La última variante L-BFGS-B, que usamos en el presente TFG en el algoritmo SHADE-ILS, es una modificación que maneja restricciones en los valores de las variables, lo que la hace incorporar información sobre el dominio. Al usar información del gradiente y de la Hessiana, es un optimizador de segundo orden, aunque es mucho menos popular que los optimizadores de primer orden. Aunque mejora la rapidez y estabilidad de la convergencia al usar más información del problema y proporciona mejores soluciones, los problemas de aprendizaje automático a día de hoy han adquirido una dimensionalidad demasiado alta para que este tipo de métodos resulten computacionalmente asequibles, y se prefiere usar los de primer orden. Aún así vemos que se implementa dentro del algoritmo de SHADE-ILS con resultados muy satisfactorios, no usándose como método de optimización principal sino de manera complementaria al algoritmo SHADE.

6.6.4. SHADE

SHADE (Success-History based Adaptative Differential Evolution) [TF13] es una variante avanzada del algoritmo original de DE. Consigue mejorar a éste a través de guardar información histórica sobre configuraciones de los parámetros de factor de mutación (F) y el ratio de cruce (CR) que han tenido buenos resultados para poder ajustar de manera adaptativa estos

parámetros y guiar el proceso de evolución, su pseudocódigo puede observarse en 5.

Los mecanismos de cruce y de selección son los mismos que en DE, variando principalmente el mecanismo de mutación. Para generar el vector mutante v_i a partir de la solución x_i, SHADE usa la siguiente estrategia[10]:

$$v_i = x_{r1} + F \cdot (x_p - x_i) F \cdot (x_{r1} - x_{r2}).$$

Donde x_p es un individuo seleccionado aleatoriamente de entre los p mejores de la población y que es distinto a x_i. También se verifica que $x_i \neq x_{rj}$ y $x_{rj} \neq x_{rj'}$ con $j \in \{1, 2\}$. En el algoritmo de SHADE se usa $p = 1$, es decir, se elige al mejor individuo de la población.

El algoritmo inicia los parámetros de factor de mutación y ratio de cruce al valor 0.5, y los va adaptando según se va ejecutando. Para ello mantiene un archivo de memoria, que se actualiza al final de cada generación y en el que se guardan parejas de los valores de los dos parámetros que han dado lugar a mejores soluciones. Al actualizar el archivo se usa la media de Lehmer[11] de manera que se le da más peso a las parejas de parámetros que mejor rendimiento obtienen. Al comienzo de cada generación el algoritmo obtiene valores de F y C_r para cada individuo basándose en el archivo de memoria e introduciendo pequeñas modificaciones.

REVISAR PSEUDOCODIGO

Algorithm 5 Algoritmo SHADE

```
t := 0
Inicializar Pob_t
Inicializar A (archivo externo)
Inicializar M (memoria de parámetros)
Evaluar x   ∀x ∈ Pob_t
while evals < total_evals do
    t := t + 1
    Seleccionar p soluciones para la mutación
    Mutar Pob_t para obtener Pob'
    Recombinar Pob' y Pob para obtener Pob"
    Evaluar Pob"
    Actualizar A y M a partir de Pob" y Pob_{t-1}
    Reemplazar Pob_t a partir de Pob" y Pob_{t-1}
end while
    return x_i ∈ Pob_t : f(x_i) ≤ f(x_j)   ∀j
```

[10] Esta es la estrategia original, existen más modificaciones, aunque basadas en esta propuesta

[11] $Lehmer(X) = \frac{\sum_{x \in X} x^2}{\sum_{x \in X} x}$

6.6.5. Algoritmos meméticos

Los algoritmos meméticos son técnicas de optimización metaheurísticas basadas en el interacción entre componentes de búsqeuda globales y locales, y tienen la explotación de conocimiento específico del problema como uno de sus principios. De manera general se componen principalmente de un algoritmo basado en poblaciones al cual se le ha integrado un componente de búsqueda local.

Su principal diferencia con los algoritmos evolutivos tradicionales es que usan de manera concienzuda todo conocimiento disponible acerca del problema. Esto no es algo opcional sino que es una característica fundamental de los algoritmos meméticos. Al igual que los algoritmos genéticos se inspiran en los genes y la evolución, estas estrategias se inspiran en el concepto de "meme", análogo al de gen pero en el contexto de la evolución cultural. Normalmente se llama "hibridar" a incorporar información del problema a un algoritmo de búsqueda ya existente y que no usaba esta información.

Esta característica de incorporar información del problema está respaldada por fuertes resultados teóricos. En el Teorema *No Free Lunch* [68] se establece que un algoritmo de búsqueda tiene un rendimiento acorde con la cantidad y calidad de información del problema que usa. Más precisamente, el teorema establece que el rendimiento de cualquier algoritmo de búsqueda es indistinguible de media de cualquier otro cuando consideramos el conjunto de todos los problemas.

6.6.6. SHADE-ILS

SHADE-ILS [MLH18] es un algoritmo memético para problemas de optimización continua a gran escala. Combina la exploración del algoritmo basado en poblaciones SHADE, usado en cada generación para evolucionar a la población de soluciones, con la explotación de una búsqueda local que se aplica a la mejor solución que se tenga en esa generación.

En la parte de búsqueda local, en el algoritmo original existe un mecanismo de elección para usar entre varias búsquedas locales, una de ellas L-BFGS-B. En el presente TFG se ha decidido usar sólo esta última, por facilidad de implementación y porque usa más información específica del problema. Por tanto no se detallará este mecanismo de elección entre búsquedas.

Las características fundamentales de esta técnica y que las diferencia con respecto a otros algoritmos meméticos son la elección de los algoritmos empleados (tanto el de búsqueda local como el basado en poblaciones) y su mecanismo de reinicio. Éste se activa cuando a lo largo de tres generaciones el rendimiento de la mejor solución no supera en más de un 5 % al de la anterior. En dicho caso, se elige una solución aleatoria de la población y se le aplica una pequeña perturbación usando una distribución normal y el resto de la población se vuelve a generar aleatoriamente. Cuando ocurre esto

los parámetros adaptativos son reiniciados a los valores por defecto.

Cabe destacar que esto se realiza ya que SHADE-ILS mantiene los parámetros adaptativos del algoritmo SHADE entre generaciones. Esto tiene mucho sentido ya que al finalizar una ejecución de dicho algoritmo, sólo aplicamos búsqueda local a una solución, con lo que la gran mayoría de la población queda intacta y por tanto podemos reutilizar estos parámetros que se han ido adaptando a ella.

SHADE-ILS mantiene un variable para guardar la mejor solución hasta ahora y otra para guardar la mejor solución desde el último reinicio, devolviendo la primera cuando finaliza el algoritmo. En la versión utilizada se ha añadido además un array para guardar el histórico de las mejores soluciones junto con su fitness correspondiente, de manera que podamos analizar y representar las mejoras que realiza el algoritmo.

Algorithm 6 Algoritmo SHADE-ILS

```
t := 0
Inicializar Pob_t
solucion_inicial = (maximo+minimo)/2
mejor_actual = LS(solucion_inicial)
mejor = mejor_actual
while evals < total_evals do
    anterior = mejor_actual.fitness
    mejor_actual, Pob = SHADE(Pob)
    mejor_actual = LS(mejor_actual)
    mejora = anterior - mejor_actual.fitness
    if mejor_actual.fitness < mejor.fitness then
        mejor = mejor_actual
    end if
    if reiniciar then
        Reiniciar y actualizar el mejor_actual
    end if
end while
    return mejor
```

Vemos el pseudocódigo de la implementación realizada en 6 y aclaramos algunas cosas que pueden no haber quedado del todo claras en favor de la claridad del pseudocódigo. Cuando generamos la población inicial, seleccionamos la peor y la mejor solución y la combinamos haciendo una media de sus elementos. A esa solución se le aplica la búsqueda local y se incluye en la población reemplazando a la peor solución. Se guardan los valores de mejora de las últimas 3 generaciones y en caso de que todas estén por debajo del 5 % se activa el mecanismo de reinicio.

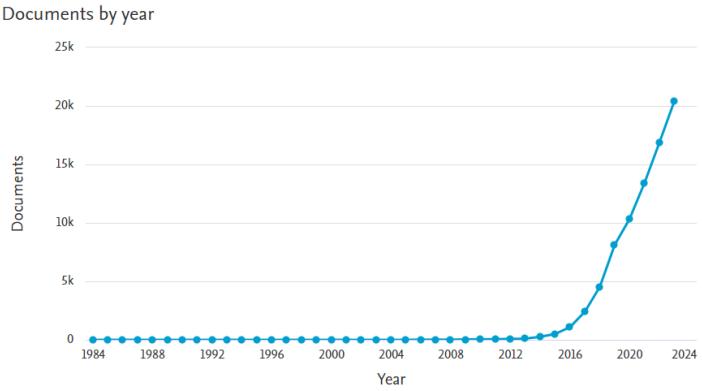

Figura 16: Resultados de la web SCOPUS para la búsqueda TITLE-ABS-KEY (deep AND learning AND training), muestra el número de artículos por año.

7. Estado del arte

En esta sección el objetivo es analizar la literatura reciente y los artículos publicados sobre el entrenamiento de modelos de aprendizaje profundo, tanto a través de técnicas basadas en gradiente descendente como metaheurísticas, enfocándonos en las familias de modelos ConvNets y MLP. Para un mejor contexto, vamos a realizar una búsqueda en la base de datos de referencias bibliográficas y citas SCOPUS, con el fin de conocer el estado actual de la literatura. Para ello la primera búsqeuda que usaremos será simple para conocer de manera general sobre el entrenamiento de modelos de aprendizaje profundo:

```
TITLE-ABS-KEY ( deep  AND learning  AND training )
AND ( LIMIT-TO ( SUBJAREA ,  "COMP" ) )
```

El total de artículos para esta búsqueda asciende a 78.378 resultados pueden apreciarse en la figura 16. En ella se aprecia un punto de inflexión en el año 2012, en el que empiezan a crecer las publicaciones anuales de manera exponencial, siendo estas prácticamente nulas previamente. Este año es en el que AlexNet consigue ganar la competición de ImageNet, con un aumento muy significativo del interés por las redes neuronales profundas a partir de entonces.

Ahora vamos a realizar una búsqueda en el ámbito del entrenamiento de modelos de aprendizaje profundo pero diferenciando entre las técnicas clásicas y las técnicas metaheurísticas, para ello se usan los términos definitorios además de los nombres de las técnicas que se emplean en este TFG, realizando las siguientes búsquedas:

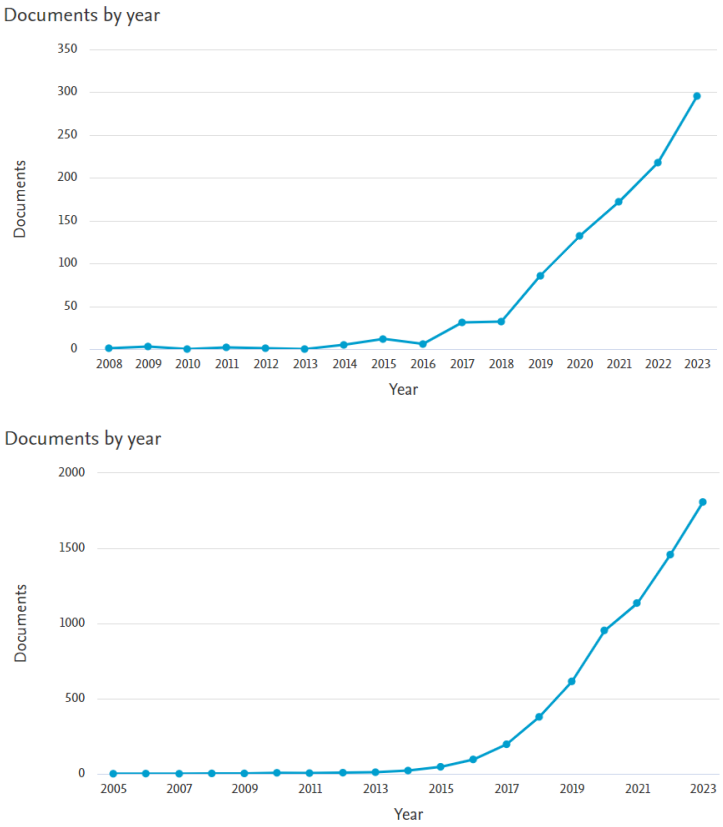

Figura 17: Artículos publicados por año para el entrenamiento de modelos de aprendizaje profundo con metaheurísticas (arriba) y gradiente descendente (debajo) según las búsquedas correspondientes.

```
TITLE-ABS-KEY ( ( deep AND learning ) AND training AND
( metaheuristic OR metaheuristics OR shade OR shade-ils
OR ( differential AND evolution ) OR memetic OR
genetic ) ) AND ( LIMIT-TO ( SUBJAREA , "COMP" ) )
```

```
TITLE-ABS-KEY ( ( deep AND learning ) AND training AND
( gradient OR adam OR optimizer OR rmsprop OR nag ) )
AND ( LIMIT-TO ( SUBJAREA , "COMP" ) )
```

Obtenemos una cantidad de 6753 artículos en lo referente al entrenamiento de modelos de aprendizaje profundo con técnicas basadas en gradiente descendente y 997 para técnicas metaheurísticas. Vemos que la diferencia entre ambas cantidades es grande, siendo la primera 7 veces mayor que la segunda. Destacamos sin embargo que la tendencia en ambos casos es muy similar, como se observa en la figura **??** aumentando prácticamente en la misma proporción desde el año 2012.

7.1. Gradiente descendente y optimizadores

El gradiente descendente es el algoritmo de aprendizaje usado por defecto prácticamente en la totalidad de los modelos de aprendizaje profundo gracias a su eficiencia y buenos resultados. Los problemas que aparecen en su convergencia se buscan evitar en la práctica a través del desarrollo de modificaciones a su algoritmo llamadas optimizadores. La literatura en este sentido es extensa, existiendo multitud de optimizadores. Primero vamos a hacer una distinción clara entre optimizadores de primer y de segundo orden.

Aunque los de segundo orden tienen mejores propiedades teóricas y proporcionan una convergencia más rápida y más estable, ya que usan más información del problema, el hecho de tener que calcular o aproximar la matriz Hessiana aumenta demasiado el poder computacional que se requiere para utilizarlos, ralentizando mucho el entrenamiento. También hay un problema de memoria, ya que para una red neuronal de 1 millón de parámetros se necesitaría almacenar una matriz de tamaño [1.000.000, 1.000.000], ocupando aproximadamente unos 3725GB de memoria RAM. Esto resulta inabarcable más aún viendo que en el top-10 modelos de clasificación de ImageNet ningún modelo baja de los mil millones de parámetros.

Incluso si eliminamos estos inconvenientes de memoria con métodos como L-BFGS (ver sección **??**, tenemos un gran inconveniente con él y es que debemos hacer el cálculo sobre todo el conjunto de entrenamiento. Estos pueden contener del orden de millones de ejemplos (ImageNet tiene más de un millón), haciendo su cálculo inasequible computacionalmente. Conseguir que este tipo de algoritmos como L-BFGS funcionen con mini-batches es más complejo que en MBGD y de hecho es un área abierta de investigación.

En la práctica no es común ver L-BFGS u otros optimizadores de segundo orden aplicados a modelos de apendizaje profundo a gran escala. En su lugar se utilizan variantes de MBGD basadas en el uso de momento y en learning rates adaptativos ya que son mucho más simples y más escalables. Existen varias opciones bastante asentadas, que forman parte de las librerías de aprendizaje automático más usadas como PyTorch y TensorFlow, que son Adam, NAG, RMSProp, AdaGrad o SGD con momento, entre otras. Para atender a las técnicas más novedosas que alcanzan un rendimiento del estado del arte, vamos a analizar la siguiente comparativa: https://akyrillidis.github.io/2020/03/05/algo.html.

En él se hace una comparativa de varios algoritmos con learning rate adaptativo (Adam, AMSGrad, AdamW, QHAdam, Demon Adam) y con learning rate no adaptativo (SGDM, AggMo, QHM, Demon SGDM). Los modelos y datasets usados en este análisis pueden observarse en la tabla 2. Una consideración muy importante que se realiza en dicha comparativa y que es bien sabida en el campo del aprendizaje automático es que el rendimiento de una técnica de entrenamiento está muy ligado al dominio específico del problema (UNIR ESTO A USAR EL MISMO OPT EN GD

Dataset	**Modelo**
CIFAR10	ResNet18
CIFAR100	VGG16
STL10	Wide ResNet 16-8
FMINST	CAPS
PTB	LSTM
MNIST	VAE

Cuadro 1: Tabla con los datasets utilizados con sus respectivos modelos en la experimentación de la comparativa (enlace)

QUE MH), y puede ocurrir que un método que no sea de los mejores en términos generales sea el mejor en un problema específico. Pasamos ahora a describir rápidamente las técnicas más interesantes.

YellowFin [ZM18] es un optimizador con learning rate y momento adaptativo, de manera que mantiene dichos hiperparámetros en un intervalo donde el ratio de convergencia es una constante igual a la raíz del momento. AdamW es una extensión de adam en la que se utiliza penalización en los pesos del modelo de manera que exista un sesgo hacia valores más pequeños durante el entrenamiento, ya que normalmente se asocian valores grandes en los parámetros con el sobreajuste. Aunque Adam ya incorpora esto, AdamW realiza una pequeña modificación a través de desacoplar esta penalización a la actualización del gradiente, resultando en un impacto notable. QHADAM, DEMON ADAM, DEMON MOMENTUM, QHM, AGGMO.

Como conclusión, y atendiendo siempre al dominio específico del problema, se tiene que YellowFin es la mejor opción en caso de no disponer de recursos para ajustar los hiperparámetros, ya que adapta el momento y el learning rate a lo largo del aprendizaje. Si se dispone de recursos pero no demasiados, lo mejor son algoritmos de learning rate adaptativo de manera que sólo se tenga que ajustar el valor del momento, en concreto destacan AdamW, QHAdam y Demon Adam. En cambio si se quiere obtener el mejor rendimiento a toda costa, invirtiendo muchos recursos en el ajuste de parámteros, usar MBGD con momento es la mejor opción, aunque sea un método más clásico. En concreto se recomienda su uso con Demon.

7.2. Metaheurísticas en el entrenamiento de modelos

Aún con el uso de optimizadores, hay ciertos inconvenientes en el entrenamiento que están provocados por el uso de BP como método de cálculo del gradiente, o directamente al uso del gradiente y no a cómo se usa. Los más comunes son el desvanecimiento y explosión del gradiente y la tendencia a quedarse atascado en mínimos locales. Las técnicas metaheurísticas son una gran alternativa ya que sus operadores de búsqueda no dependen de BP evitando así sus problemas.

Uno de los enfoques de aplicación de estas técnicas es la combinación con las técnicas clásicas, con diferentes aproximaciones. Por ejemplo en [Ban19] se usa el algoritmo *Artificial Bee Colony* [Kar05] sobre un conjunto de soluciones aleatorias para generar un conjunto de soluciones al que aplicar el algoritmo de gradiente descendente. En [PKN18] se combina un algoritmo genético con el gradiente descendente, de manera que las nuevas soluciones son generadas con el operador de búsqueda del primero pero son evaluadas tras realizar varias épocas con el segundo. De manera similar en [Kha+17] se usa la técnica metaheurística *Particle Swarm Optimization* [BM17] para entrenar los parámetros de la última capa de una ConvNet, mientras que el resto se entrenan a través del algoritmo de gradiente descendente. La comparación arroja que la hibridación de ambas técnicas alcanza mejores resultados en términos de rapidez de convergencia y de *accuracy*. Prácticamente la totalidad de la literatura referente a esta estrategia está centrada en ConvNets.

El otro enfoque es entrenar el modelo usando exclusivament algoritmos bio-inspirados. En este ámbito destacan los estudios [RFA15] y [Ayu+16] en los que se proponen dos técnicas basadas en *Simulated Annealing* [KGV83] para entrenar los parámetros de una ConvNet, consiguiendo mejor rendimiento y mejor velocidad de convergencia que en el mismo modelo entrenado mediante el algoritmo de gradiente descendente. Al igual que ocurre con el enfoque anterior, la gran mayoría de estos estudios están centrados en ConvNets y RNNs.

Algo importante a destacar en lo referente a la literatura de entrenamiento de modelos con técnicas metaheurísticas es la falta de rigor y de un marco común en los estudios, de manera que no pueden realizarse comparaciones objetivas entre ellos. Estos detalles, que se comentan con más detalle en la sección 5.1, ponen en evidencia la necesidad de más experimentos bajo unas condiciones similares que permitan poder sacar conclusiones objetivas entre ellos.

El rendimiento de estas técnicas todavía no es comparable al de las técnicas clásicas. Si bien es cierto que para tareas con datasets sencillos y modelos con pocos parámetros pueden mejorar al gradiente descendente en minimizar la función de pérdida para el conjunto de entrenamiento, normalmente en generalización su rendimiento lo empeora. Para modelos con muchos parámetros el rendimiento de estas técnicas está por detrás, con resultados no muy distantes pero peores de manera general. Además hay que tener en cuenta la complejidad computacional, ya que para alcanzar un rendimiento similar al gradiente descendente estas técnicas necesitan de mucho más tiempo y recursos computacionales, por lo que no resultan una alternativa viable para este tipo de tareas.

7.2.1. SHADE-ILS

Presentamos ahora una de las técnicas metaheurísticas que mejor resultados ofrece actualmente en el entrenamiento de modelos. En esta sección nos limitaremos a valorar sus resultados en el trabajo [Mar+20], mientras que su definición y funcionamiento se presentan en la sección 6.6.6. En la experimentación de dicho trabajo se atienden tres cuestiones, las tres a través de técnicas metaheurísticas: diseño de la arquitectura, optimización de hiperparámetros y entrenamiento de los parámetros de un modelo.

Nos centraremos en la última. Se utilizan 6 datasets distintos con diferente complejidad, y en base a ésta se elige una arquitectura de modelo concreta dentro de la familia de las ConvNets, de manera que tenga buen rendimiento en su entrenamiento a través de gradiente descendente. Se utiliza el optimizador Adam para esta tarea. En el entrenamiento con SHADE-ILS se utilizan diferentes estrategias que hacen uso de la estructura de capas de los modelos de aprendizaje profundo, realizando el entrenamiento en los pesos de diversas capas, según la estrategia, mientras se mantienen congelados los demás. También se realiza el entrenamiento de todo el modelo a la vez.

Los resultados de la experimentación son claros: solo en uno de las 6 tareas uno de los modelos entrenado con SHADE-ILS minimiza más la función de pérdida que el modelo entrenado con gradiente descendente. Además, en todos los casos, el error de test es mayor. Cabe mencionar que la generalización en los modelos es bastante buena, manteniéndose estos errores en valores cercanos a los que se obtiene en el entrenamiento, y aumentando el error en proporciones similares al modelo entrenado con Adam.

8. Experimentación

Se ha desarrollado una batería de pruebas amplia y diversa que permita una correcta comparación entre el gradiente descendente y las técnicas metaheurísticas atendiendo a los objetivos señalados anteriormente. En la tabla 2 se ofrece un resumen esquemático de las pruebas que se van a realizar. Todas ellas se realizan con los siguientes optimizadores de gradiente descendente: NAG, RMSProp y Adam. Para las metaheurísticas se ha elegido los ya conocidos en la literatura SHADE y SHADE-ILS, y añadiendo una versión híbrida de cada uno con el gradiente descendente. Todas estas elecciones se detallarán a lo largo de esta sección. Para el desarrollo del código se usa el lenguaje Python con las librerías PyTorch y FastAI, entre otras; implementado y ejecutado en Google Colab. El código puede encontrarse en: https://github.com/eedduu/TFG.

Familia	MLP				ConvNets		
Modelo	1,2,5 y 11				LeNet5, ResNet-15 y ResNet57		
Datasets	BHP	BCW	WQ		MNIST	F-MNIST	CIFAR10-G
Tarea	R	C	R	C	Clasificación de imágenes		

Cuadro 2: Resumen de la experimentación. BHP: Boston Housing Price, BCW: Breast Cancer Winsconsin, WQ: Wine Quality. R: regresión, C: clasificación. En el caso de MLP, en la fila modelo se indica el número de capas ocultas.

Capas ocultas	**Neuronas por capa**	**Parámetros**
1	64	2238
2	64, 64	6462
5	64, 128, 256, 128, 64	85k
11	32, 64, 128, 256, 512, 1024, 512, 256, 128, 64 y 32	1.4M

Cuadro 3: Detalles de los modelos MLP

8.1. Modelos

Usaremos dos familias de modelos: MLP y ConvNets. Con los primeros usaremos datasets tabulares para clasificación y regresión, y con los segundos datasets de imágenes para la tarea de clasificación. En la siguiente sección se detallan los datasets con sus características.

La implementación de los MLP se ha realizado a través de la librería FastAI por simplicidad ya que ofrece lo necesario para usarlos directamente. La implementación de las ConvNets se ha realizado desde cero, observando la topología de LeNet5 y las ResNets en sus papers originales, ya que en ellas sí que se han introducido ciertos cambios que se comentan más adelante. Estas modificaciones tienen como objetivo una batería experimental más amplia y acorde a las condiciones que buscamos. Todos los modelos han sido entrenados desde cero.

Para los MLP usaremos 5 modelos, con 1,2,5 y 11 capas ocultas cada uno. El número de neuronas por capa es una potencia de 2 y con estructura piramidal incremental, es decir primero aumentando el número de neuronas por capa y luego disminuyéndolo. Estas son elecciones comunes en la literatura ya que facilitan las operaciones por su estructura (la primera) y el tratamiento de los datos (la segunda). El objetivo es conseguir una variedad experimental que permita medir los efectos de la complejidad del modelo sobre la tarea y el overfitting además de adecuarse a las condiciones del paper de referencia, con modelos desde aproximadamente mil parámetros hasta casi 1.5M como vemos en 3, acercándose al modelo más grande presentado en dicho paper.

Capa	**Dimensión**	**Kernel**	**Canales**
Convolución	28x28	5x5	6
BatchNorm2D	28x28	-	-
ReLU	28x28	-	-
Max Pool	14x14	2x2, stride 2	-
Convolución	10x10	5x5	16
BatchNorm2D	10x10	-	-
ReLU	10x10	-	-
Max Pooling	5x5	2x2	-
Lineal	120	-	-
BatchNorm1D	120	-	-
ReLU	120	-	-
Lineal	84	-	-
BatchNorm1D	84	-	-
ReLU	84	-	-
Lineal	num_classes	-	-

Cuadro 4: Topología de LeNet5 para imágenes 32x32 con un canal de entrada. Las columnas dimensión y canales hacen referencia a la salida de la capa.

Antes de cada capa linal hay una de BatchNorm1D, ya que es la implementación por defecto de FastAI y mejora el rendimiento en el entrenamiento. Los parámetros asociados a este tipo de capa y a los de la capa de salida van incluidos en el cómputo anterior. Se incluye al final del modelo una capa de SoftMax en caso de que la tarea sea clasificación.

Para los modelos basados en convoluciones usamos LeNet5 y dos ResNets, con 15 y 57 capas. El objetivo es de nuevo ofrecer una variedad experimental, con el primer modelo teniendo unos 60k parámetros mientras que el tercero tiene 1.3M, imitando de nuevo las condiciones del paper de referencia. Hay que resaltar que la intención es comparar varios modelos con diferente número de parámetros para ver el efecto sobre éstos, y no hablamos de modelos más o menos potentes, ya que no siempre un incremento en el número de parámetros se traduce en un mejor rendimiento.

LeNet5 es un modelo de sobra conocido presentado por Yann LeCun en [Lec+98], al que sustituimos las funciones de activación por ReLU, ya que en la literatura posterior a la presentación del modelo se han demostrado superiores a las sigmoides y la tangente hiperbólica. También se han sustituido las capas de AveragePool por MaxPool y añadido capas de BatchNorm por los mismos motivos. En la tabla 4 se muestra la topología de este modelo, obviando las capas de Flatten y de SoftMax. Tiene un total de 62 mil parámetros.

Se han diseñado dos modelos de ResNet, uno con 15 capas y otro con 57. A priori puede parecer excesivo 57 capas, pero con la implementación a través de BottleNeckBlocks se aumenta el número de capas considerablemente con un leve incremento del número de parámetros. Sigue además la tendencia en la literatura de aumentar el número de capas antes que el número de filtros. Se han elegido 57 capas con la intención de aproximarse al número de parámetros del modelo más grande del paper de referencia y aprovechar la estructura de esta familia de modelos. Las ResNets se caracterizan por usar bloques convolucionales que agrupan varias capas de convolución donde al final de cada bloque se suma la entrada del mismo, con el objetivo de evitar el problema del desvanecimiento de gradiente (ver sección 4.4.1). En un modelo con pocas capas no se aprecia tan bien este efecto. Su topología se detalla en la tabla 5.

El modelo intermedio, Resnet15, sigue la misma estructura que ResNet57 pero usando menos bloques convolucionales. El objetivo es tener un modelo intermedio, con unos 500 mil parámetros y que nos permita comparar con los otros dos, en términos de entrenamiento y de overfitting, algo muy usual en las redes neuronales con gran número de parámetros. Su topología se detalla en la tabla 6.

Los bloques convolucionales agrupan 3 capas de convolución con sus respectivas capas BatchNorm, y se usan convoluciones 1x1 para hacer cuello de botella, reduciendo así el número de parámetros sin perder expresividad de la red [LCY14; Sze+14]. Se sigue el diseño usual de esta familia de modelos, por ejemplo agrupando más bloques convolucionales en mitad de la red, con una convolución preia a los bloques convolucionales y usando solo una capa lineal. El modelo ResNet57 que se implementa tiene un total de 1.3M de parámetros.

8.2. Datasets

TODO: añadir información sobre los datasets: número de características, target, tamaño, qué contienen, imagenes.

8.2.1. Tabulares

BCW y WQ para clasificación, BHP y WQ para regresión. El objetivo es comparar también el rendimiento de las metaheurísticas en estas tareas, ya que la gran mayoría de la literatura sobre MH se realiza con ConvNets y por tanto con tareas de clasificación. BCW y BHP son dos dataset pequeños (alrededor de 500 instancias) y más fáciles que WQ, que tiene una cantidad de unas 6000 instancias. Por ello hay una tarea sencilla y una difícil para regresión y clasificación.

Para estos datasets se ha realizado un preprocesado de los datos básico y con decisiones comunes basadas en la literatura. Se han eliminado las

Capa	**Dimensión**	**Kernel/Stride**	**Canales**
Convolución	26x26	7x7	64
BatchNorm2d	26x26	-	-
ReLU	26x26	-	-
MaxPool2d	13x13	2x2, stride 2, padding 1	-
BottleneckBlock x3	13x13	1x1, 3x3, 1x1	64
BottleneckBlock x4	7x7	1x1, 3x3, 1x1, stride 2	128
BottleneckBlock x4	4x4	1x1, 3x3, 1x1, stride 2	256
BottleneckBlock x3	2x2	1x1, 3x3, 1x1, stride 2	512
AdaptiveAvgPool2d	512	-	-
BatchNorm1d	512	-	-
Dropout	512	-	-
Lineal	num_classes	-	-

Cuadro 5: Topología de ResNet57 para imágenes 32x32 con un canal de entrada. Las columnas dimensión y canales hacen referencia a la salida de la capa.

Capa	**Dimensión**	**Kernel/Stride**	**Canales**
Convolución	26x26	7x7	64
BatchNorm2d	26x26	-	-
ReLU	26x26	-	-
MaxPool2d	13x13	2x2, stride 2, padding 1	-
BottleneckBlock x1	13x13	1x1, 3x3, 1x1	64
BottleneckBlock x1	7x7	1x1, 3x3, 1x1, stride 2	128
BottleneckBlock x1	4x4	1x1, 3x3, 1x1, stride 2	256
BottleneckBlock x1	2x2	1x1, 3x3, 1x1, stride 2	512
AdaptiveAvgPool2d	512	-	-
BatchNorm1d	512	-	-
Dropout	512	-	-
Lineal	num_classes	-	-

Cuadro 6: Topología de ResNet15 para imágenes 32x32 con un canal de entrada. Las columnas dimensión y canales hacen referencia a la salida de la capa.

variables que tienen menos de un 5 o 10 % (dependiendo de la cantidad de variables del dataset) de correlación con el target. Con las parejas de variables que tienen más de un 90 % de correlación entre sí se elimina una de las dos. Se han eliminado outliers con el método zscore usando un threshold de 3 y se han normalizado los datos de entrada.

El tamaño del batch se ha elegido mediante pruebas experimentales entre los valores 32, 64 y 128. Se divide el conjunto de datos en entrenamiento-validación-test, con un porcentaje 70-10-20.

8.2.2. Imágenes

Para los datasets de imágenes se han elegido tres de los usados en el paper de referencia. Se han elegido MNIST y FMNIST ya que son los más usados para la comparativa en dicho paper, y CIFAR10 ya que a priori es más difícil que los otros, siendo MNIST el más fácil de los tres. Esto proporciona unas pruebas equilibradas, en el mismo marco que el paper y permite una fácil comparación. Al igual que allí, se ha reducido el tamaño de los datasets a 10 mil imágenes para el entrenamiento y 5 mil para el test. Del conjunto de entrenamiento se toma el 30 % para validación.

Se usan las imágenes con una resolución de 32x32 y un solo canal, adaptando las imágenes a estas dimensiones cuando sea necesario. No se usa preprocesamiento de datos ya que se entiende que la propia red a través de las convoluciones los procesa.

8.3. Otras decisiones

Para la reproducibilidad de la experimentación, se fija la semilla 42 en todos las librerías necesarias. No se usa cross validation debido a que las técnicas metaheuristicas requieren de mucho más poder de cómputo que el disponible para poder ejecutarlas varias veces en un tiempo razonable.

Se ha usado la inicialización de pesos Glorot como en el paper a comparar. Para los optimizadores basados en gradiente descendente se han usado los mismos pesos iniciales, mientras que en las técnicas metaheurísticas se ha usado la misma población inicial, para asegurar la igualdad de condiciones en el entrenamiento debido a la gran sensibilidad de éste a los parámetros iniciales.

Se fija 20 como número de épocas en todos los entrenamientos ya que es suficiente para la convergencia de los métodos de gradiente descendente. Siguiendo el criterio del paper de referencia, en las técnicas metaheurísticas una época es algo distinta, siendo en el paper equivalente a $N_eval * N_epochs * N_capas$. En nuestro caso no se ha usado el factor número de capas por diversas razones: no se realiza entrenamiento individualizado por capas, no se dispone de tanto poder de cómputo, usamos modelos con muchas capas. Aunque usemos menos evaluaciones del conjunto de entrenamiento por

época, la comparación sigue en la misma dinámica y las metaheurísticas recibirán mucho más poder de procesamiento que las técnicas clásicas.

Para las tareas de regresión se ha usado el error cuadrático medio como función de coste. Es ampliamente usada en la literatura y aunque es sensible a los outliers, como tenemos preprocesamiento de datos no nos afecta. Se ha usado también la métrica R2 cuadrado para medir la explicación de la varianza con respecto a la media como predicción, para tener un criterio objetivo de comparación ya que en las dos tareas de regresión la escala del objetivo es distinta. Para las tareas de clasificación se ha usado CrossEntropyLoss como función de error y accuracy como métrica, opciones ampliamente usadas en la literatura. En los datasets tabulares se ha usado BalancedAccuracy en lugar de Accuracy, ya que las clases no están balanceadas, y se conoce que en estos casos accuracy no es una métrica representativa, de hecho se hace especial mención a esto en el paper. En los datasets de imágenes las clases están perfectamente balanceadas por lo que se usa accuracy, aunque en este caso su versión balanceada coincidiría con la normal.

8.4. Optimizadores

Existen tres familias de optimizadores del gradiente descendente principales: las que usan el momento, las que autoajustan el learning rate y las que combinan estas dos estrategias. Para la experimentación se elige un optmizador de cada familia con la intención de ver si existe alguna diferencia en el entrenamiento del gradiente descendente debido a éstos. El criterio para la elección del optimizador de cada familia ha sido el número de citas en el paper de presentación y su extensión de uso, analizando cómo de frecuente es su uso en la literatura y su implementación por defecto en las principales librerías de aprendizaje automático. Respectivamente se han elegido: NAG, RMSProp y Adam.

Se ha usado la implementación de estos optimizadores de PyTorch, usandose en el entrenamiento a través de la librería FastAI. Se usa en el entrenamiento la política de ciclos de Leslie, que proporciona una velocidad de convergencia mucho mayor variando el learning rate de forma cíclica. Para elegir el valor máximo del learning rate se usa un buscador de learning rate, que aporta unos resultados muy buenos a un coste bajo. Cabe destacar que la importancia del learning rate es sensiblemente menor en los optimizadores de RMSProp y Adam, ya que de manera automática ajustan estos valores a lo largo del entrenamiento.

Para los hiperparámetros de dichos optimizadores se han usado los valores por defecto de la librería PyTorch. En primer lugar, la intención con la experimentación es establecer una comparativa en igualdad de condiciones sobre el entrenamiento de modelos, y no obtener el máximo rendimiento de cada uno. En segundo lugar estos valores por defecto están basados en los propuestos en los respectivos papers originales y ajustados a través de nu-

merosas pruebas experimentales, de manera que proporcionen los mejores resultados posibles de manera general basandose en la propia implementación de PyTorch.

8.5. Metaheurísticas

Para una correcta comparación con el paper de referencia, que usa el algoritmo SHADE-ILS, se eligen 4 algoritmos metaheurísticos entorno a éste. En primer lugar usamos SHADE-ILS, en su versión completa (entrenando todos los parámetros a la vez). Usamos también el algoritmo de SHADE, sin búsqueda local, para usar un algoritmo puramente metaheurístico que no sea memético. Luego estas dos versiones anteriores las hibridamos con el gradiente descendente, para tener algoritmos meméticos que usen información específica del problema.

El algoritmo de SHADE se implementa a través de la librería pyade-master (LINK), y se le han realizado modificaciones para mantener los parámetros adaptativos del algoritmo SHADE entre ejecuciones distintas y adaptar las estructuras de datos a las del resto del código. A partir de dicho algoritmo se implementan manualmente el resto. Para el algoritmo SHADE-ILS, que combina dos tipos de búsqueda local en su paper original, con una aplicación general, se ha decidido mantener sólo la búsqueda local a través de L-BFGS, ya que al hacer uso del gradiente se maneja información más específica del problema. Esto se realiza a través de la librería scipy.

Se hibridan SHADE y SHADE-ILS con optimizadores basados en gradiente descendente, aunque SHADE-ILS ya usa información del gradiente en su ejecución a través de L-BFGS pero es un optimizador de segundo orden. Para la hibridación se ejecuta el algoritmo y al final de cada 4 épocas se realiza un entrenamiento de gradiente descendente de una época. Esto se realiza para que el peso del entrenamiento lo siga teniendo en mayor medida la parte metaheurística del algoritmo en lugar de la parte clásica. El optimizador que se usa en este caso es el que mejor resultados diera en el entrenamiento basado en gradiente descendente, eligiendo según el modelo que usemos.

Se mantiene la misma división de los datos que en el uso de optimizadores. Se ejecutan los algoritmos evaluando el error sobre el conjunto de entrenamiento y guardando en cada epoch el mejor individuo de la población junto a su error en el entrenamiento, para luego evaluarlos sobre el conjunto de validación y observar qué modelo generaliza mejor a priori, igual que hacemos en el entrenamiento de modelos con gradiente descendente. Luego evaluamos el modelo sobre el conjunto de test, calculando las métricas asociadas a cada tarea.

Como en el paper de referencia no aparecen los hiperparámetros asociados a estos algoritmos, se decide usar el mismo criterio que con los optimizadores y usar los hiperparámetros por defecto de la implementación de la

librería pyade-master, que además son valores comúnes en la literatura.

Referencias

[Ahm+11] Amir Ali Ahmadi et al. "NP-hardness of deciding convexity of quartic polynomials and related problems". En: *Mathematical Programming* 137.1–2 (nov. de 2011), págs. 453-476. ISSN: 1436-4646. DOI: 10.1007/s10107-011-0499-2. URL: http://dx.doi.org/10.1007/s10107-011-0499-2.

[Ayu+16] Vina Ayumi et al. *Optimization of Convolutional Neural Network using Microcanonical Annealing Algorithm*. 2016. arXiv: 1610.02306 [cs.CV]. URL: https://arxiv.org/abs/1610.02306.

[Ban19] Anan Banharnsakun. "Towards improving the convolutional neural networks for deep learning using the distributed artificial bee colony method". En: *International Journal of Machine Learning and Cybernetics* 10 (jun. de 2019). DOI: 10.1007/s13042-018-0811-z.

[Bay+15] Atilim Gunes Baydin et al. *Automatic differentiation in machine learning: a survey*. cite arxiv:1502.05767Comment: 43 pages, 5 figures. 2015. URL: http://arxiv.org/abs/1502.05767.

[Ber+23] David Bertoin et al. *Numerical influence of ReLU'(0) on backpropagation*. 2023. arXiv: 2106.12915.

[Bis06] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Berlin, Heidelberg: Springer-Verlag, 2006. ISBN: 0387310738.

[BLH21] Yoshua Bengio, Yann LeCun y Geoffrey E. Hinton. "Deep learning for AI." En: *Commun. ACM* 64.7 (2021), págs. 58-65. URL: http://dblp.uni-trier.de/db/journals/cacm/cacm64.html%5C#BengioLH21.

[BM17] Mohammad reza Bonyadi y Zbigniew Michalewicz. "Particle Swarm Optimization for Single Objective Continuous Space Problems: A Review". En: *Evolutionary Computation* 25 (mar. de 2017), págs. 1-54. DOI: 10.1162/EVCO_r_00180.

[Bub15] Sébastien Bubeck. "Convex Optimization: Algorithms and Complexity". En: (2015). cite arxiv:1405.4980Comment: A previous version of the manuscript was titled "Theory of Convex Optimization for Machine Learning". URL: http://arxiv.org/abs/1405.4980.

[Byr+95] Richard H. Byrd et al. "A Limited Memory Algorithm for Bound Constrained Optimization". En: *SIAM Journal on Scientific Computing* 16.5 (1995), págs. 1190-1208. DOI: 10.1137/0916069. eprint: https://doi.org/10.1137/0916069. URL: https://doi.org/10.1137/0916069.

[Cau09] Augustin-Louis Cauchy. "ANALYSE MATHÉMATIQUE. – Méthodc générale pour la résolution des systèmes d'équations simultanées". En: 2009. URL: https://api.semanticscholar.org/CorpusID:123755271.

[Cur44] Haskell B. Curry. "The method of steepest descent for non-linear minimization problems". En: *Quarterly of Applied Mathematics* 2 (1944), págs. 258-261. URL: https://api.semanticscholar.org/CorpusID:125304075.

[Dau+14] Yann Dauphin et al. *Identifying and attacking the saddle point problem in high-dimensional non-convex optimization*. 2014. arXiv: 1406.2572 [cs.LG].

[Dea+12] Jeffrey Dean et al. "Large scale distributed deep networks". En: *Advances in neural information processing systems*. 2012, págs. 1223-1231. URL: http://papers.nips.cc/paper/4687-large-scale-distributed-deep-networks.pdf.

[DHS11] John Duchi, Elad Hazan y Yoram Singer. "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization". En: *Journal of Machine Learning Research* 12 (jul. de 2011), págs. 2121-2159.

[DS96] John E. Dennis, Jr. y Robert B. Schnabel. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. Vol. 16. Classics in Applied Mathematics. Philadelphia, PA, USA: SIAM, 1996.

[Fei24] Li Fei-Fei. *CS231n: Convolutional Neural Networks for Visual Recognition*. https://cs231n.stanford.edu/. 2024.

[GB10] Xavier Glorot y Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks." En: *AISTATS*. Ed. por Yee Whye Teh y D. Mike Titterington. Vol. 9. JMLR Proceedings. JMLR.org, 2010, págs. 249-256. URL: http://dblp.uni-trier.de/db/journals/jmlr/jmlrp9.html%5C#GlorotB10.

[GBC16] Ian J. Goodfellow, Yoshua Bengio y Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. Cambridge, MA, USA: MIT Press, 2016.

[GP10] Michel Gendreau y Jean-Yves Potvin, eds. *Handbook of metaheuristics*. 2.ª ed. New York, NY, USA: Springer, 2010.

[He+15a] Kaiming He et al. *Deep Residual Learning for Image Recognition*. 2015. arXiv: 1512.03385 [cs.CV]. URL: https://arxiv.org/abs/1512.03385.

[He+15b] Kaiming He et al. *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification*. cite arxiv:1502.01852. 2015. URL: http://arxiv.org/abs/1502.01852.

[Hin12] Geoffrey Hinton. *Lecture 6e rmsprop: Divide the gradient by a running average of its recent magnitude*. https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf. Neural Networks for Machine Learning, Coursera. 2012.

[Hoc+01] S. Hochreiter et al. "Gradient flow in recurrent nets: the difficulty of learning long-term dependencies". En: *A Field Guide to Dynamical Recurrent Neural Networks*. Ed. por S. C. Kremer y J. F. Kolen. IEEE Press, 2001.

[Hol92] John H. Holland. "Genetic Algorithms". En: *Scientific American* (jul. de 1992).

[IS15] Sergey Ioffe y Christian Szegedy. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. 2015. arXiv: 1502.03167 [cs.LG]. URL: https://arxiv.org/abs/1502.03167.

[Kar05] Dervis Karaboga. "An Idea Based on Honey Bee Swarm for Numerical Optimization, Technical Report - TR06". En: *Technical Report, Erciyes University* (ene. de 2005).

[KB17] Diederik P. Kingma y Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: 1412.6980 [cs.LG].

[KGV83] Scott Kirkpatrick, C. Gelatt y M. Vecchi. "Optimization by Simulated Annealing". En: *Science (New York, N.Y.)* 220 (jun. de 1983), págs. 671-80. DOI: 10.1126/science.220.4598.671.

[Kha+17] Mujahid Khalifa et al. "Particle swarm optimization for deep learning of convolution neural network". En: nov. de 2017, págs. 1-5. DOI: 10.1109/SCCSIT.2017.8293059.

[KSH12] Alex Krizhevsky, Ilya Sutskever y Geoffrey E Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". En: *Advances in Neural Information Processing Systems*. Ed. por F. Pereira et al. Vol. 25. Curran Associates, Inc., 2012. URL: https://proceedings.neurips.cc/paper%5C_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

[LBH15] Yann LeCun, Yoshua Bengio y Geoffrey Hinton. "Deep learning". En: *nature* 521.7553 (2015), pág. 436.

[LCY14] Min Lin, Qiang Chen y Shuicheng Yan. *Network In Network*. 2014. arXiv: 1312.4400 [cs.NE]. URL: https://arxiv.org/abs/1312.4400.

[LeC+12] Yann A. LeCun et al. "Efficient BackProp". En: *Neural Networks: Tricks of the Trade: Second Edition*. Ed. por Grégoire Montavon, Geneviève B. Orr y Klaus-Robert Müller. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, págs. 9-48. ISBN: 978-3-642-35289-8. DOI: 10.1007/978-3-642-35289-8_3. URL: https://doi.org/10.1007/978-3-642-35289-8%5C_3.

[Lec+98] Y. Lecun et al. "Gradient-based learning applied to document recognition". En: *Proceedings of the IEEE* 86.11 (1998), págs. 2278-2324. DOI: 10.1109/5.726791.

[LN89] Dong C. Liu y Jorge Nocedal. "On the limited memory BFGS method for large scale optimization". En: 45 (1989), págs. 503-528.

[Mar+20] Aritz D. Martinez et al. *Lights and Shadows in Evolutionary Deep Learning: Taxonomy, Critical Methodological Analysis, Cases of Study, Learned Lessons, Recommendations and Challenges*. 2020. arXiv: 2008.03620 [cs.NE].

[Mit97] Tom M Mitchell. *Machine learning*. Vol. 1. 9. McGraw-hill New York, 1997.

[MK87] Katta G. Murty y Santosh N. Kabadi. "Some NP-complete problems in quadratic and nonlinear programming". En: *Math. Program.* 39.2 (jun. de 1987), págs. 117-129. ISSN: 0025-5610.

[MLH18] Daniel Molina, Alberto LaTorre y Francisco Herrera. "SHADE with Iterative Local Search for Large-Scale Global Optimization". En: *2018 IEEE Congress on Evolutionary Computation (CEC)*. IEEE, 2018, págs. 1-8. DOI: 10.1109/CEC.2018.8477755.

[Mur22] K.P. Murphy. *Probabilistic Machine Learning: An Introduction*. Adaptive Computation and Machine Learning series. MIT Press, 2022. ISBN: 9780262046824. URL: https://books.google.co.uk/books?id=HLlyzgEACAAJ.

[Nes83] Yurii Nesterov. "A method for solving the convex programming problem with convergence rate O(1/k$\hat{2}$)". En: *Proceedings of the USSR Academy of Sciences* 269 (1983), págs. 543-547. URL: https://api.semanticscholar.org/CorpusID:145918791.

[Nøk16] Arild Nøkland. *Direct Feedback Alignment Provides Learning in Deep Neural Networks*. 2016. arXiv: 1609.01596 [stat.ML].

[Nov17] K. Novak. *Numerical Methods for Scientific Computing*. Lulu.com, 2017. ISBN: 9781365659911. URL: https://books.google.es/books?id=GvXpDQAAQBAJ.

[PKN18] Krzysztof Pawełczyk, Michal Kawulok y Jakub Nalepa. "Genetically-trained deep neural networks". En: jul. de 2018, págs. 63-64. DOI: 10.1145/3205651.3208763.

[PSL05] K.V. Price, R.N. Storn y J.A. Lampinen. *Differential Evolution: A Practical Approach to Global Optimization*. Natural Computing Series. Springer, 2005.

[Qia99] Ning Qian. "On the momentum term in gradient descent learning algorithms". En: *Neural Networks* 12.1 (1999), págs. 145-151. ISSN: 0893-6080. DOI: https://doi.org/10.1016/S0893-6080(98)00116-6. URL: https://www.sciencedirect.com/science/article/pii/S0893608098001166.

[RFA15] L.M. Rere, Mohamad Ivan Fanany y Aniati Arymurthy. "Simulated Annealing Algorithm for Deep Learning". En: vol. 72. Nov. de 2015. DOI: 10.1016/j.procs.2015.12.114.

[RHW86] David E Rumelhart, Geoffrey E Hinton y Ronald J Williams. "Learning representations by back-propagating errors". En: *nature* 323.6088 (1986), págs. 533-536.

[Ros58] F. Rosenblatt. "The perceptron: A probabilistic model for information storage and organization in the brain." En: *Psychological Review* 65.6 (1958), págs. 386-408. ISSN: 0033-295X. DOI: 10.1037/h0042519. URL: http://dx.doi.org/10.1037/h0042519.

[Sej18] Terrence J. Sejnowski. *The Deep Learning Revolution*. Inglés americano. Cambridge, MA: MIT Press, 2018. ISBN: 978-0-262-03803-4.

[Smi17] Leslie N. Smith. *Cyclical Learning Rates for Training Neural Networks*. 2017. arXiv: 1506.01186 [cs.CV]. URL: https://arxiv.org/abs/1506.01186.

[Smi18] Leslie N. Smith. *A disciplined approach to neural network hyper-parameters: Part 1 – learning rate, batch size, momentum, and weight decay*. 2018. arXiv: 1803.09820 [cs.LG]. URL: https://arxiv.org/abs/1803.09820.

[SP97] Rainer Storn y Kenneth V. Price. "Differential Evolution - A Simple and Efficient Heuristic for global Optimization over Continuous Spaces." En: *J. Glob. Optim.* 11.4 (1997), págs. 341-359. URL: http://dblp.uni-trier.de/db/journals/jgo/jgo11.html#StornP97.

[ST18] Leslie N. Smith y Nicholay Topin. *Super-Convergence: Very Fast Training of Neural Networks Using Large Learning Rates*. 2018. arXiv: 1708.07120 [cs.LG]. URL: https://arxiv.org/abs/1708.07120.

[Sze+14] Christian Szegedy et al. *Going Deeper with Convolutions*. 2014. arXiv: 1409.4842 [cs.CV]. URL: https://arxiv.org/abs/1409.4842.

[TBS23] Vinita Tomar, Mamta Bansal y Pooja Singh. "Metaheuristic Algorithms for Optimization: A Brief Review". En: *Engineering Proceedings* 59.1 (2023). ISSN: 2673-4591. DOI: 10.3390/engproc2023059238. URL: https://www.mdpi.com/2673-4591/59/1/238.

[TF13] Ryoji Tanabe y Alex Fukunaga. "Success-history based parameter adaptation for Differential Evolution". En: jun. de 2013, págs. 71-78. ISBN: 978-1-4799-0453-2. DOI: 10.1109/CEC.2013.6557555.

[Zha+23] Aston Zhang et al. *Dive into Deep Learning*. 2023. arXiv: 2106.11342 [cs.LG]. URL: https://arxiv.org/abs/2106.11342.

[ZM18] Jian Zhang y Ioannis Mitliagkas. *YellowFin and the Art of Momentum Tuning*. 2018. arXiv: 1706.03471 [stat.ML]. URL: https://arxiv.org/abs/1706.03471.