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, mes de 201

Índice

Parte I
Parte matemática: gradiente descendente y *backpropagation*

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 2000s 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 **??**, 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/

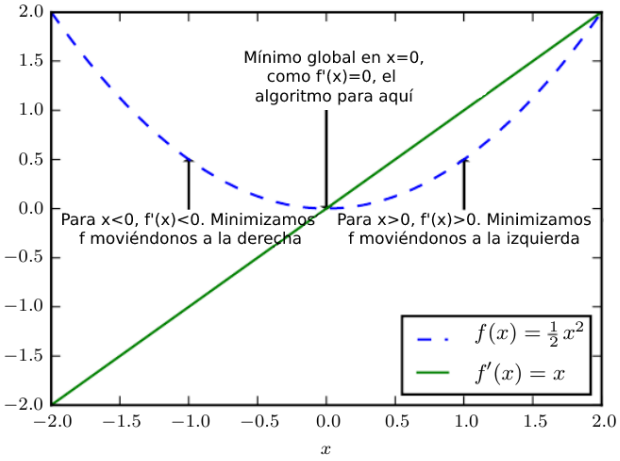

Figura 1: Esquematización de la estrategia de descenso del gradiente en un modelo con un solo parámetro x. El eje horizontal representa los valores que toma éste y el vertical representa el error del modelo en función de x. Imagen obtenida y traducida del libro [GBC16]

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, 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 ofrece. Éste a su vez depende casi enteramente del algoritmo de BP para calcular el gradiente, ya que aunque existan 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 mejoras 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 algo 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.

2. Explorar su implementación a través de BP. Para ello, analizaremos su funcionamiento e implementación.

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 necesarios 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

A continuación se definirán los principales conceptos que se usarán durante el presente TFG. 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

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

de sus virtudes, ya que 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.

Tomaremos X e Y como dos espacios normados no triviales cualesquiera. Se fija una función $f : A \to Y$ donde $\varnothing \neq A \subseteq X$ y un punto $a \in A^\circ$

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

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

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

Definición 2.2 (Derivada parcial) *Sea $f : \Omega \subset \mathbb{R}^n \to Y \subset \mathbb{R}^M$, con Ω un abierto, $f = (f_1, f_2, ..., f_M)$, $a \in \Omega$, $k \in I_n$. Entonces f es parcialmente derivable con respecto a la k-ésima variable en a si, y sólo si, lo es $f_j \forall j \in I_M$, en tal caso,*

$$\frac{\partial f}{\partial x_k}(a) = \left(\frac{\partial f_1}{\partial x_k}(a), ..., \frac{\partial f_M}{\partial x_k}(a) \right) \in \mathbb{R}^M$$

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

Definimos ahora los elementos clave del proceso: 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 de cálculos intermedios, que son las matrices jacobianas asociadas entradas y salidas de las capas ocultas (que tienen mayor dimensionalidad) con respecto a parte de los parámetros (los parámetros de esa capa).

Definición 2.3 (Vector gradiente) *Sea $f : \Omega \subset \mathbb{R}^N \to \mathbb{R}$ un campo escalar, con Ω un abierto y $a \in \Omega$. Cuando f es parcialmente derivable en a, el gradiente de f en a es el vector $\nabla f(a) \in \mathbb{R}^N$ dado por*

$$\nabla f(a) = \left(\frac{\partial f}{\partial x_1}(a), \frac{\partial f}{\partial x_2}(a), ..., \frac{\partial f}{\partial x_N}(a) \right)$$

Fijamos un abierto $\Omega \subset \mathbb{R}^N$, y la función $f : \Omega \to \mathbb{R}^M$. Notamos por $f = (f_1, f_2, ..., f_M)$ indicando las M componentes de f que son campos escalares definidos en Ω, siendo $f_j = \pi_j \circ f$.

Definición 2.4 (Matriz jacobiana) *Si f es diferenciable en $x \in \Omega$, la matriz jacobiana es la matriz de la aplicación lineal $Df \in L\left(\mathbb{R}^N, \mathbb{R}^M\right)$ y se escribe como J_f. Viene dada por:*

$$J_f(x) = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_N} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_N} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_M}{\partial x_1} & \frac{\partial f_M}{\partial x_2} & \cdots & \frac{\partial f_M}{\partial x_N} \end{pmatrix} = \begin{pmatrix} \nabla f_1(x)^T \\ \vdots \\ \nabla f_M(x)^T \end{pmatrix} = \begin{pmatrix} \frac{\partial f}{\partial x_1} \cdots \frac{\partial f}{\partial x_N} \end{pmatrix}$$

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) *Sean X, Y, Z espacios normados, Ω un abierto no vacío de X y U lo es de Y, y las funciones $f : \Omega \to U$ y $g : U \to Z$. Entonces si f es diferenciable en $a \in \Omega$ y g es diferenciable en $b = f(a)$ se tiene que $f \circ f$ es diferenciable en a con*

$$D(g \circ f)(a) = Dg(b) \circ Df(a) = Dg(f(a)) \circ f(a)$$

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

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.

2.2. Lipschitz

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

Definición 2.5 (Función Lipschitziana) *Si E y F son espacios normados, una función $f : E \to F$ es lipschitziana si existe una constante $M \in \mathbb{R}_0^+$ que verifica:*

$$\|f(x) - f(y)\| \leq M\|x - y\| \qquad \forall x, y \in E$$

Decimos que la función f tiene gradiente lipschitziano si la condición anterior se aplica a su gradiente.

$$\|\nabla f(x) - \nabla f(y)\| \leq M\|x - y\| \qquad \forall x, y \in E$$

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

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

La definición nos dice de manera intuitiva que el gradiente de la función no puede cambiar a una velocidad arbitraria.

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 f sea lipschitziano es que $\nabla^2 f(x) \preceq LI \quad \forall x \in E$. Es decir que los valores propios de la matriz Hessiana están mayorados por L. Esta equivalencia la usaremos luego en la demostración **??**

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 los 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, y aunque es bastante claro que el gradiente descendente sigue siendo la mejor estrategia de optimización de parámetros de un modelo de forma general, 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, en su gran mayoría, está intrínsecamente ligado a la optimización, específicamente 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 importancia de la generalización: no es importante únicamente obtener un error bajo sino que ese error se mantenga cuando usamos datos de entrada nuevos, siendo preferible aumentar un poco el error en el entrenamiento si con ello conseguimos mejorar la generalización del modelo.

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.

Definimos $f : \mathbb{R}^N \to \mathbb{R}$ una función que es continua en todas sus variables[6] y las cuales no toman valores negativos. Notamos a las variables por $x = (x_1, ..., x_N) \in \mathbb{R}^N$. Si queremos encontrar los valores de $x_1, ..., x_N$ que verifican $f(x_1, ..., x_N) = 0$, bastará con hacer decrecer indefinidamente los valores de la función f hasta que sean muy cercanos a 0.

Fijamos ahora unos valores concretos $x_0 \in \mathbb{R}^N$, $u = f(x_0)$, $Du = (D_{x_1}u, D_{x_2}u, ..., D_{x_N}u)$ 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. Si aumentamos el valor de η entonces el valor de la función decrecerá hasta cero o en caso contrario hasta alcanzar un mínimo.

3.2. Gradiente descendente en el entrenamiento de modelos

En el caso del aprendizaje 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

[6] Realmente solo necesitamos que sean continuas en el intervalo concreto en el que se moverán.

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*.

En el proceso de entrenamiento de un modelo lo que hacemos es aplicar la idea anterior de manera iterativa en lugar de encontrar un η que minimice en un paso, para ir moviéndonos a puntos de menor valor de la función de coste. Si $C(W)$ es la función de coste del modelo y W representa los parámetros del modelo, entonces la regla de actualización iterativa del descenso del gradiente es la siguiente:

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

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, 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 se puedan 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 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 practicamente la única estrategia utilizada en la realidad ya que ofrece la mayor eficiencia computacional, estabilidad y rapidez en la convergencia.

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 como veremos más adelante, 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 **??** 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 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 modifiación 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 **??** o en resultados sobre la convergencia en la sección **??**.

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 dedican excesivos recursos computacionales a esta búsqueda, invirtiéndose por el contrario 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 del 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 también tener un vector de *learning rate* en lugar de un solo

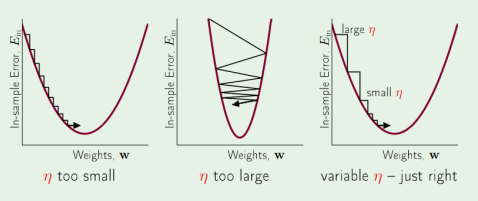

Figura 2: Visualización de cómo afecta el *learning rate* según su adecuación al problema. Imagen obtenida del curso de Caltech [8], tema 9 diapositiva 21

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. Podemos pensar en un modelo como una composición de la suma y producto 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.

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

Las funciones de coste son diferenciables de manera general, y las más comunes son por ejemplo el error cuadrático medio para problemas de regresión y *CrossEntropyLoss* para problemas de clasificación. Donde N es el número de datos, $\hat{y}_i$ es el valor real del dato i, que en ECM será un escalar, y en clasificación binaria será 0 ó 1; y_c es el valor predicho por el modelo, $\hat{y}_c$ es la etiqueta real de la clase c, que valdrá 1 en caso de que el dato pertenece a la clase c y 0 en caso contrario; y y_c es el predicho por el modelo.

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. Sobre ese año empezaron a popularizarse las funciones 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

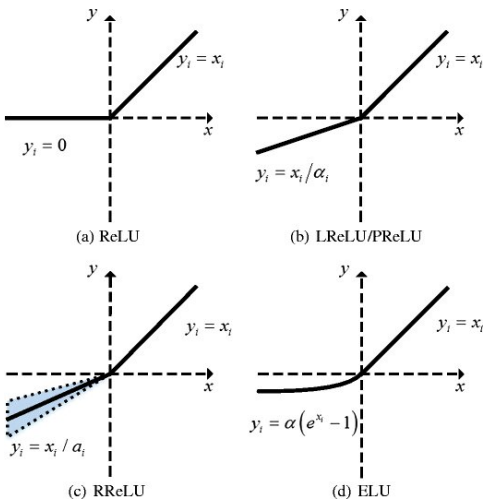

Figura 3: Función ReLU y algunas de sus variantes más usadas como funciones de activación.[10]

variantes que aparecen en la figura **??** 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 $\forall y \in A$ se tiene:*

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

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

Tenemos que comprobar que el subgradiente extiende al gradiente, es decir, que cuando existe gradiente entonces existe el subgradiente y coincide con él, y además hay funciones que no tienen el gradiente pero sí subgradiente. Vamos a definir lo que es un conjunto convexo, ya que resulta elemental tanto en esta sección como en la siguiente y usaremos este concepto para desarrollar otros a partir de él.

Definición 3.2 (Conjunto convexo) *Un subconjunto E de un espacio vectorial X es convexo cuando, para cualesquiera dos puntos de E, el segmento que los une está contenido en E:*

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

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

Para demostrar esta proposición, primero vamos a necesitar de un teorema, en el ámbito de la convexidad:

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

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

Demostración. Para la primera implicación, queremos probar que $f((1-t)x + ty) \leq (1-x)f(x) + tf(y) \forall x, y \in X, \quad t \in [0,1]$, es decir, que f es convexa. Partiendo de que $\forall x \in X, \quad \partial f(x) \neq \emptyset$, tomando cualquier $g \in \partial(z), \quad z \in X$ tenemos por la definición de subgradiente:

$$f(z) - f(x) \leq g^T(z - x),$$
$$f(z) - f(y) \leq g^T(z - y),$$

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

$$f((1-t)x + ty) - f(x) \leq g^T(((1-t)x + ty) - x),$$
$$f((1-t)x + ty) + g^T(x - ((1-t)x - ty) \leq f(x),$$
$$f((1-t)x + ty) + g^T(t(x - y) \leq f(x),$$

$$f((1-t)x + ty) + tg^T(x - y) \leq f(x) \tag{2}$$

Desarrollando en la otra desigualdad de manera análoga obtenemos

$$f((1-t)x + ty) + (1-t)g^T(y - x)) \leq f(y) \tag{3}$$

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

$$(1-t)f(x) + tf(y) \geq$$
$$(1-t)f((1-t)x + ty) + t(1-t)g^T(x - y) + tf((1-t)x + ty) + t(1-t)g^T(y - x)$$
$$= f((1-t)x + ty) + t(1-t)g^T(x - y) + t(1-t)g^T(y - x)$$

Como se tiene que $g^T(x - y) + g^T(y - x) = 0$, entonces tenemos que $(1-t)f(x) + tf(y) \geq f((1-t)x + ty), \quad \forall x, y \in X, \quad t \in [0,1]$. Por tanto f es convexa, como queríamos probar.

Ahora vamos a probar que f tiene algún subgradiente en $int(X)$. Definimos el epigrafo de una función f como $epi(f) = \{(x,t) \in X \times \mathbb{R} : t \geq f(x)\}$. 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 $x \in X$, claramente $(x, f(x)) \in \partial epi(f)$, 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

$$a^T x + bf(x) \geq a^T y + bt, \forall (y,t) \in epi(f) \tag{4}$$

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 $t \to \infty$, necesitamos que $b \leq 0$. Ahora vamos a asumir que $x \in int(X)$. Entonces tomamos $\epsilon > 0$, verificando que $y = x + \epsilon a \in X$, lo que implica que $b \neq 0$, ya que si $b = 0$ entonces necesariamente $a = 0$. Reescribiendo **??** 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 segunda parte de la proposición.

Para la última parte, sea f una función convexa y diferenciable. Entonces por definición para $t \in [0,1]$ y $x, y \in X$

$$tf(y) + (1-t)f(x) \geq f((1-t)x + ty)$$

$$f(y) \geq \frac{f((1-t)x + ty) - (1-t)f(x)}{t}$$

$$= f(x) + \frac{f(x + t(y-x)) - f(x)}{t}$$

$$\xrightarrow[t \to 0]{} \quad f(x) + \nabla f(x)^T (y - x)$$

Lo que demuestra que $\nabla f(x) \in \partial f(x)$

□

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**: $\partial(af) = a \cdot \partial f, a \geq 0$
2. **Suma**: $\partial(f_1 + f_2) = \partial f_1 + \partial f_2$
3. **Composición afín**: Si $g(x) = f(Ax + b) \Rightarrow \partial g(x) = A^T \partial f(Ax + b)$

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 ReLU, por ejemplo. 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 [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.

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:*

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

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 convergerá a puntos donde el gradiente sea 0. Un factor clave para la convergencia será el hecho de que la función de pérdida sea o no una función convexa.

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:*

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

En caso de que la función de coste sea convexa sólo existirá un mínimo y será global, por lo que no tenemos que preocuparnos de si el algoritmo se queda estancado en un mínimo local, ya que si converge tenemos 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. Resultados teóricos para la convergencia del gradiente descendente

Los desarrollos teóricos sobre la convergencia del algoritmo de descenso del gradiente son muchos y variados. Los principales inconvenientes para el desarrollo de un marco teórico que sea útil en la práctica 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 tanto tiempo como 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 genérica que el estudio teórico de los elementos del algoritmo. Además es un procedimiento genérico aplicable en cualquier problema, por lo que resulta más sencillo.
- La mayoría de resultados teóricos aprovechables son basados en el gradiente descendente original (BGD), ya que cuando introducimos el ruido que genera la aproximación del cálculo del gradiente en sus variantes SGD y MBGD se complica el desarrollo teórico. Los resultados que aparecen a continuación son, a menos que se especifique lo contrario, resultados para BGD.

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

Teorema 3.2 (Convergencia para funciones convexas) *Suponemos f una función convexa y diferenciable, con su gradiente Lipschitz continuo*

con constante $L > 0$, teniendo que $\|\nabla f(x) - \nabla f(y)\|_2 \leq L\|x - y\|_2 \quad \forall x, y$. 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 $f^{(k)}$ que satisface la siguiente ecuación:

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

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

Demostración.

En el teorema anterior $x \in \mathbb{R}^n$ son 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 ∇f es Lipschitz continuo con constante L entonces $\nabla^2 f(x) \preceq LI$, donde I es la matriz identidad. Equivalentemente esto significa 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, $y = x - \eta\nabla f(x) = x^+$.

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

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

$$f(x^+) \leq f(x) - \frac{1}{2}\eta\|\nabla f(x)\|_2^2 \tag{5}$$

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

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

La segunda desigualdad se deduce de la primera de manera inmediata. Sustituyendo esa segunda desigualdad en **??** obtenemos

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

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

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

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.

Podemos obtener un resultado mucho más práctico, ya que es para SGD y MBGD y además con condiciones más relajadas. Usando la teoría de algoritmos aproximados estocásticos, en concreto 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 punto crítico. Esto nos da un criterio sencillo para la convergencia, y que no depende de parámetros como la constante de Lipschitz que son complejos de computar.

Proposición 3.2 (Convergencia para SGD) *Sea x_k la secuencia generada por el algoritmo de SGD, sea f la función de coste del modelo y sea η_k el valor del learning rate en la iteración k. Si la sucesión η_k satisface $\sum_{k=1}^{\infty}\eta_k = \infty$ y también $\sum_{k=1}^{\infty}\eta_k^2 < \infty$ entonces $\lim_{k\to\infty}\|\nabla f(x_k)\| = 0$. Es decir, el algoritmo converge hacia un punto crítico de la función.*

Aunque la complejidad de la demostración se escapa al alcance de este TFG, ya que sería necesario demasiadas definicones y resultados previos, en la página 33 de [Gad12] [11] podemos encontrar el enunciado y demostración del teorema de Robbins-Siegmund. En [Han24] podemos observar su uso para la demostracion de la convergencia de SGD. Hay que remarcar que, aunque la demostración de la convergencia se realiza en supuesto de función convexa a un mínimo global, se puede tomar una restricción local de la función de manera que esta sea convexa y se alcance un punto crítico.

3.4.2. Problemas en la convergencia

En el teorema anterior tenemos asegurada la convergencia a un mínimo, sin embargo en el segundo solo nos garantizamos llegar a un punto crítico,

[11] El original 'A convergence theorem for non negative almost supermartingales and some applications' es de pago y no puedo acceder

ni siquiera a un mínimo local. Encontramos aquí el mayor problema del algoritmo del gradiente descendente: la convergencia prematura en puntos con gradiente muy cercano a cero.

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 de una función $f(x)$ 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 solventarlos es utilizar modificaciones en el algoritmo de gradiente descendente que proporcionan mejores propiedades a su comportamiento, 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. Estas modificaciones se denominan optimizadores y a diferencia de las versiones vistas en la sección **??**, 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 de 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. La diferencia 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 sería mucho más ineficiente la computación. Se suele decir de manera general que el algoritmo de BP es una aplicación concreta de la diferenciación automática hecho 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 como 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 comunmente 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 datasets tabulares, es decir aquellos que sus datos vienen en formato de tabla. Usaremos esta arquitectura para realizar el desarrollo teórico ya que es más sencilla conceptualmente y la notación no resulta tan engorrosa.

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 $i \in \{1, ..., k\}$

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

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

Donde la entrada del modelo viene representada en la primera capa, $x = x_1$. Además se tiene que $m_i = n, m_{k+1} = m, x_{k+1} = o$. 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 $J_f(x) = \mathbb{R}^{m \times n}$ 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

$$J_f(x) = \frac{\partial o}{\partial x} = \frac{\partial o}{\partial x_k}\frac{\partial x_k}{\partial x_{k-1}} \cdots \frac{\partial x_3}{\partial x_2}\frac{\partial x_2}{\partial x_1} =$$

$$\frac{\partial f_k(x_k)}{\partial x_k}\frac{\partial f_{k-1}(x_{k-1})}{\partial x_{k-1}} \cdots \frac{\partial f_2(x_2)}{\partial x_2}\frac{\partial f_1(x_1)}{\partial x_1} =$$

$$= J_{f_k}(x_k)J_{f_{k-1}}(x_{k-1}) \cdots J_{f_2}(x_2)J_{f_1}(x_1)$$

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

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

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

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 de la matriz jacobiana que sea más eficiente depende de qué valor es mayor: si n o m.

Si $n < m$ será más eficiente calcular $J_f(x)$ para cada columna $j \in \{1, ..., n\}$ 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 $i \in 2, ..., k-1$ 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, que es un tipo de algoritmo usado en diferenciación automática (ver algoritmo **??**). Asumiendo que $m = 1, m_i = m_j$, el coste de computar $J_f(x)$ es $O(n^2)$.

Donde los elementos de la matriz fila $v_j, j \in \{1, ..., n\}$ 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 $n > m$ es más eficiente calcular $J_f(x)$ para cada fila $i = 1, ..., m$ 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 $i \in 2, ..., k-1$ y $J_{f_1}(x_1)$ tiene tamaño $m_1 \times n$. Esto puede calcularse usando la diferenciación hacia atrás (ver algoritmo **??**). Asumiendo que $m = 1, m_i = m_j$, el coste de computar $J_f(x)$ es $O(n^2)$.

Algorithm 1 Diferenciación hacia delante

```
x_1 := x
for j ∈ {1, ..., n} do
    v_j := e_j ∈ R^n
end for
for i ∈ {1, ..., k} do
    x_{i+1} := f_i(x_i)
    for j ∈ {1, ..., n} do
        v_j := J_{f_i}(x_i) v_j
    end for
end for
    return o = x_{k+1}, (v_1, v_2, ..., v_n)
```

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. 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 que dos

Algorithm 2 Diferenciación en modo reverso

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

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 escalar.

Acabamos de comprobar que para calcular una matriz jacobiana resulta más eficiente usando 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 esos pesos. Los parámetros son valores reales y tienen la forma $W = W_1 \times W_2 \times \cdots \times W_k \subset \Omega$, y $W_i \in \mathbb{R}^{n_i \times n_{i+1}}$. 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.

Ahora vamos a considerar la función de coste del modelo como una capa más, a parte de las funciones de las capas 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 equación*

$$f_1(x_1, W_1) = \sigma_1(W_1 x_1) = x2$$
$$f_2(x_2, W_2) = W_2 x_2 = x_3 = f(x, W) = o$$
$$C(f(x, W), y) = \frac{1}{2}\|x_3 - y\|^2 = E$$

El objetivo, para poder aplicar el entrenamiento a través del gradiente descendente será calcular el gradiente del error con respecto a los parámetros $\frac{\partial E}{\partial W}$. Buscamos obtener un vector gradiente de la misma dimensión que W, pero el calculo 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 $\frac{\partial E}{\partial W_i}, \forall i = 1, ..., k$. 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 **??**.

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 $\sigma'(a) = \frac{d}{da}\sigma(a)$. 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 **??** 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.

$$z = f(x) = CrossEntropy(x, y) = -\sum_c y_c log(softmax(x)_c) = -\sum_c y_c log(p_c)$$

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 (codificado normalmente como un one-hot encoded vector). 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 má 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

$$z_l = \sum_{t=1}^{n} W_{lt} x_t$$

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

Por tanto

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

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

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

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

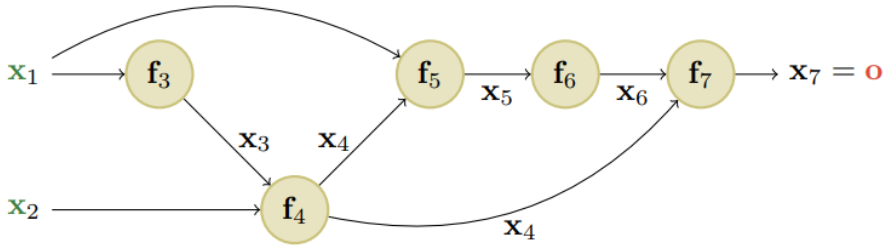

Figura 4: Representación del grafodirigido acíclico (imagen obtenida de **??**)

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. 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 ver reflejado en la figura **??**.

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 API 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 pueden 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 **??**, vamos a ver dos problemas que surgen a la hora de entrenar modelos usando gradiente descendente y que lastran 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 el 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 que 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 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.

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 $w_{ij} \sim \mathcal{N}(0, \sigma^2)$, y $\mathbb{E}[x_j] = 0$ 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+15].

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 usemos 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 II
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) y las Residual Networks (ResNets) se han erigido como las familias de modelos que consiguen un rendimiento del estado del arte.

Las ConvNets son un tipo de red neuronal para procesar 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 linear 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.

Las ResNets son una extensión de las redes neuronales profunda que atajan el problema del desvanecimiento y explosión de gradiente. 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 backpropagation.

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 cuanto 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 esquivar 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.

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 en-

tre ellos, mutarlos para obtener más diversidad genética, y reemplazamiento de los nuevos indiviudos 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 modelos 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.

Los algoritmos meméticos son una hibridación de algoritmos evolutivos con algoritmos de búsqueda local, que añade 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 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 candidato sustituto.

Para la realización del presente TFG nos basaremos en el reciente paper de Daniel Molina y Paco Herrera REF 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 gradiente descendente, y diferentes versiones de SHADE-ILS, una técnica metaheurística basada en DE y con uso de búsqueda local (memética) que ofrece los mejores resultados actualmente en el entrenamiento de modelos.

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, ya que no resulta fácil realizar una comparación totalmente objetiva entre dos ténicas tan distintas como el gradiente descendente y los algoritmos bio-inspirados. Algunas de las principales carencias en la literatura que se intentarán abordar en este TFG son las siguientes:

- De manera general no se suelen comparar las técnicas metaheurísticas con los métodos clásicos del gradiente descendente, sino que se comparan entre sí.
- Falta de homogeneidad en los datasets usados, lo que no permite una comparación objetiva entre papers
- Modelos no realistas que usan unos pocos de miles de parámetros.
- La gran mayoría de hibridaciones entre técnicas metaheurísticas y gradiente descendente son probadas con ConvNets

Por ello aunque la literatura sea extensa se hace necesario tanto la realización de experimentos con las mismas condiciones que en otros papers de referencia como la repetición de los mismos para obtener esa independencia, objetividad y rigor metodológico que son necesarias en el campo. Vamos a reproducir el experimento en modelos MLP y ConvNets de diferente número de parámetros (desde 2 mil hasta 1 millón), usando técnicas clásicas y metaheurísticas reconocidas por su buen funcionamiento y además probando hibridaciones entre ellas. Se intentará reproducir en la medida de lo posible las condiciones de experimentación del paper de referencia con especial hincapié en el uso de algunos de los mismos datasets de imágenes, y entendiendo que existirá una diferencia clara en el poder de cómputo.

5.2. Objetivos

El objetivo principal de este TFG es realizar una comparación experimental de las técnicas de entrenamiento de modelos clásicas basadas en gradiente descendente y las nuevas basadas en metaheurísticas, proponiendo hibridaciones para las más consolidadas. Para ello dividimos en los siguientes objetivos secundarios:

1. Reproducción de la experimentación del paper de referencia en el ámbito del entrenamiento de ConvNets.
2. Realización de pruebas experimentales análogas a las anteriores para el entrenamiento de MLP
3. Hibridación de técnicas metaheurísticas usadas con gradiente descendente, con sus pruebas correspondientes.
4. Análisis de resultados, en comparación con los resultados del paper cuando corresponda.

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.

[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.

[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.

[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 nonlinear 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.

[Gad12] Stéphane Gadat. *Cours d'algorithmes de stockage et de structures de données*. Université de Toulouse. PDF file. 2012. URL: https://perso.math.univ-toulouse.fr/gadat/files/2012/12/cours_Algo_Stos_M2R3.pdf.

[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.

[Han24] Niels Richard Hansen. *Why stochastic gradient descent works*. https://nrhstat.org. 2024. URL: https://nrhstat.org/post/robbins_siegmund/.

[He+15] 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.

[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.

[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.

[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.

[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.

[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.

[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.

[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.

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