Aprendizaje automático con redes neuronales: de los fundamentos a las aplicaciones

Universidad Complutense de Madrid

Facultad de Matemáticas

Doble Grado en Matemáticas y Físicas. Curso Académico 2022-2023.

Ignacio Benito Acedo Blanco

Madrid, Junio de 2023

Resumen

El presente trabajo se centra en el análisis matemático de los cimientos de las redes neuronales así como sus aplicaciones y resultados. Comenzamos examinando los fundamentos teóricos del aprendizaje automático, destacando la importancia de los algoritmos de optimización en la obtención de modelos de alta precisión y viendo con detalle los algoritmos basados en el descenso del gradiente y Backpropagation. Finalizaremos esta introducción repasando brevemente las redes bayesianas.

A continuación, se ahondará en el algoritmo Adam, un método de optimización más sofisticado ampliamente utilizado en el entrenamiento de redes neuronales. Describimos su funcionamiento, destacando su capacidad para adaptar automáticamente la tasa de aprendizaje mediante el empleo de los momentos de primer y segundo orden del gradiente; y se presentarán los resultados obtenidos con él. En este estudio se incluye una demostración- detallada de su orden de convergencia.

El trabajo se concluye con la realización de un código en Python para implementar todas las técnicas estudiadas, empleando únicamente la biblioteca Numpy. Analizaremos sus resultados experimentales y daremos unas conclusiones.

Abstract

This paper focuses on the mathematical analysis of neural networks as well as their applications and results. We begin by examining the theoretical foundations of machine learning, highlighting the importance of optimisation algorithms in obtaining high-precision models and looking in detail at algorithms based on gradient descent and Backpropagation. We will end this introduction by briefly glancing Bayesian networks.

Next, we will delve into the Adam algorithm, a sophisticated optimisation method widely used in neural network training. We describe its functioning, highlighting the ability to automatically adapt the learning rate by using the first and second order moments of the gradient. A detailed demonstration of its convergence order is included in this study.

The work concludes with the development of a Python code to implement all the techniques studied, using only the Numpy library. We will analyse the experimental results and draw some conclusions.

Agradecimientos

Para todos aquellos que me dieron fuerzas para siempre seguir adelante.

Para mis padres que me enseñaron el valor del esfuerzo y a luchar por lo que quiero hasta el final.

Para mis hermanos que me apoyaron y animaron en mis momentos de flaqueza.

A todos mis amigos que supieron estar cuando más los necesitaba.

A Jorge, Jaime, Diego, Melchor y Arturo por siempre sacarme una sonrisa.

A Pablo por enseñarme que no estoy solo.

Para Lausanne y toda la gente que conocí ahí.

A María por confiar tanto en mí y enseñarme una nueva manera de ver la vida.

Muchas gracias. Sin vosotros no hubiera sido posible.

Índice de contenidos

Capítulo 1

Introducción a las redes neuronales

No cabe duda de la revolución informática en la que nos encontramos sumergidos. Con la actual tecnología y la que está por llegar, muchas de las tareas que se antojaban difíciles a los sistemas computacionales se están volviendo cada día más cercanas, o incluso ya resueltas.

Uno de los objetivos desde los inicios de la comunidad informática ha sido la imitación del pensamiento humano, lo que es comúnmente conocido por *Inteligencia artificial*. Si bien no hay una definición clara sobre este término tan de moda, con él se hace referencia a la capacidad de un sistema lógico, como es un ordenador, de razonar por sí mismo. Entre las tecnologías que afrontan este desafío, la más sorprendente y disruptiva de las últimas 2 décadas engloba al tan sonado *Machine Learning*.

Y en especial, dentro esta rama se encuentran las famosas *redes neuronales*, una de las técnicas más empleadas en los últimos años y que mejores resultados están arrojando. Como su propio nombre indica, trata precisamente de imitar la arquitectura de pensamiento del cerebro humano, definiendo unas unidades básicas (neuronas) e interconectándolas en ellas. Vemos multitud de ejemplos de uso cotidianamente en sistemas de recomendación (e.g. YouTube) o chatbots entre los que se incluye Chat GPT.

Este nuevo enfoque rompe el anterior paradigma informático debido al funcionamiento básico del programa. Clásicamente, si se desea que un ordenador realice una tarea, previamente se ha de establecer un protocolo o algoritmo a realizar para cada situación posible, subdividiendo de este modo el problema original en otros más sencillo y manejables. Por contra, el Machine Learning implica un aprendizaje autónomo del propio programa, evitando una programación explícita de la resolución concreta del problema y obteniendo así soluciones no establecidas por el desarrollador.

Como cabe esperar, todas las técnicas de esta nueva tecnología, desde las más actuales a la más antiguas, tienen una fuerte base matemática. El objetivo de este Trabajo de Fin de Grado es precisamente hacer un repaso de las matemáticas presentes en las redes neuronales, así como un análisis riguroso de la eficiencia de sus métodos. Una vez vistas las bases, se estudiará detalladamente Adam, un algoritmo de optimización que ha proporcionado grandes resultados, además de inspirar la creación de otras muchas técnicas y algoritmos. Se proporcionará una demostración de su orden de convergencia. Por último, se realizará un programa escrito en Python que implemente todo lo aprendido para comprobar su potencia experimentalmente haciendo uso exclusivo de la libreria Numpy.

1.1. Machine Learning

El aprendizaje automático, también conocido como Machine Learning, es un campo de estudio dentro de la inteligencia artificial que se enfoca en el desarrollo de algoritmos y modelos computacionales capaces de aprender y mejorar su rendimiento en tareas específicas a partir de la experiencia. Estos algoritmos incluyen técnicas para el análisis de datos, identificación de patrones y realización de predicciones sin la necesidad de

ser programados de una forma explícita.

El proceso fundamental del Machine Learning se compone de tres elementos esenciales: datos de entrenamiento, modelo y algoritmo de optimización. Los *datos de entrenamiento* consiste en conjuntos de información previamente recopilados y etiquetados[1] del problema a resolver, permitiendo al programa identificar los patrones y emular su comportamiento. El *modelo*, por su parte, es una representación matemática del problema en cuestión, cuyo propósito es capturar los patrones y relaciones ocultas presentes en los datos. Los *algoritmos* son los procedimientos y reglas que orientan el entrenamiento y la toma de decisiones del modelo.

En la aplicación del Machine Learning, el tratamiento de los datos consume una gran cantidad de tiempo. Sin embargo, en este trabajo se va a asumir que el conjunto viene ya dado, no haciendo un estudio sobre su preparación y manipulación. En este primer capítulo, estudiaremos el modelo de las redes; mientras que en futuros capítulos entraremos con detalle en los algoritmos de optimización.

1.1.1. Redes neuronales

Como bien es sabido, el cerebro está compuesto de neuronas. Sin entrar en detalles técnicos, estas son las células y componentes principales del sistema nervioso, encargadas de recibir, procesar y transmitir los impulsos nerviosos y químicos a través de descargas eléctricas. El cerebro humano posee alrededor de 10^{11} de estas, que son entrelazadas entre sí para crear un complejo sistema de conexión. De este modo, las neuronas procesan individualmente el impulso recibido y comparten la descarga resultante con otras vecinas, lo que da lugar a la capacidad del razonamiento. Lo que resulta interesante para los propósitos de este trabajo y para la informática es la manera en la que cada neurona individual trata la información recibida; así como el modo en el que es transmitida.

Cada neurona tiene en torno a varios miles de conexiones con el resto de sus compañeras. Cuando una neurona recibe un impulso eléctrico o una señal química de neuronas vecinas, dicha señal puede provocar una excitación en la neurona receptora desencadenando como reacción una transmisión de un nueva descarga. Para ello, los impulsos recibidos se suman entre sí y se integran de una manera específica de modo que, si esta última señal resultante excede una barrera, un voltaje en respuesta es generado y propagado por las conexiones de la neurona.

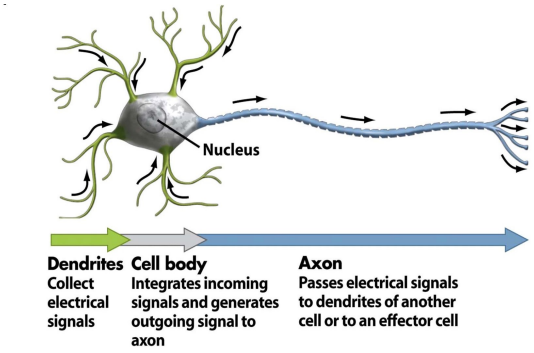
Es esta arquitectura básica la que se propone digitalizar e implementar en nuestras redes neuronales artificiales para dar lugar al "pensamiento del ordenador". Realizando la analogía necesaria entre el sistema biológico y el computacional, se sustituyen las señales eléctricas o químicas provenientes de otras neuronas por simples escalares, denotadas por x_k con $k \in \mathbb{N}$ (pueden representar, por ejemplo, un código de colores, significando x_1 que el color es amarillo, x_2 el naranja...). Cada una de estas entradas son agrupadas en un vector conocido por *input* y representado por $\mathbf{x} = (x_1, x_2, ...)$.

Para comenzar, debido a su símil con la biología y fácil comprensión, se explicará el funcionamiento de las primeras neuronas informáticas que surgieron históricamente: los *perceptrones*, actualmente obsoletas. En dicho modelo cada entrada del vector del input x_k es binario, i.e, $x_k \in \{0, 1\}$.

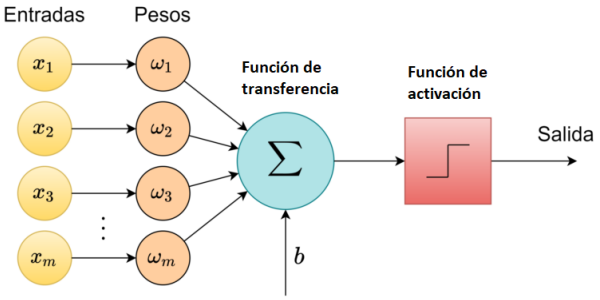
Una vez que un perceptrón intercepta un vector de input, este es capaz de excitar a la neurona resultando en una comunicación de la información. Para decidir qué excita en mayor o menor medida a la neurona, se incorpora para cada una de las componentes del vector de entrada un nuevo escalar definido como *peso* y representando por w_k, que es multiplicado por su correspondiente componente del input. Continuando el anterior ejemplo, podemos controlar si el color amarillo (x_1) excita más a la neurona que el naranja (x_2) tomando por ejemplo $w_1 = 2$ y $w_2 = 1$. Seguido de esto, las excitaciones resultantes son integradas, esto es, se toma el sumatorio $\sum w_i x_i$. Este resultado es definido como *transferencia*. En función de si su magnitud supera una barrera o *sesgo*[2] denotado por b, el perceptrón transmitirá información a la siguiente neurona (el output es 1) o no (el output es 0). Este proceso de flujo de información, tanto como para el caso biológico real como para su modelización matemática (perceptrón), es ilustrado en la figura 1.1.

[1] En ciertos contextos no abordados aquí, estos datos podrían no estar etiquetados, lo cual corresponde al aprendizaje no supervisado.

[2] *bias* en inglés.



(a) M. Mitra (2007). Neurona biológica real junto con el flujo de información



(b) Alulema (2022). Funcionamiento básico de perceptrón.

Figura 1.1: Neurona biológica y neurona matemática.

Pese a ser un modelo verdaderamente intuitivo, el perceptrón presenta una gran carencia: un pequeño cambio de los pesos puede conllevar un cambio drástico sobre la salida -puesto que únicamente se permiten componentes binarias de los vectores de entrada y salida. Consecuentemente, en el caso de neuronas más completas, se permite al output, y por tanto el input, tomar valores continuos. Así, $x_k \in \mathbb{R}$.

Adicionalmente, en el diseño explicado, una neurona no es más que una función lineal con una serie de parámetros w_k. Al realizar las distintas conexiones entre ellas, la composición de funciones resulta en una nueva función lineal. Debido a las limitaciones de modelización que esto conlleva, es necesario añadir una etapa de no-linealización sobre el procesamiento. Esto se consigue, como se ve en la figura 1.1, gracias a la introducción de una función de activación no lineal h previo al retorno de la salida, por lo que el output resultante viene dado por $h(\sum_k w_k x_k)$. Se destaca entre todas las funciones de activación la función sigmoide, que da nombre a las neuronas sigmoides:

$$\sigma(z) = \frac{1}{1 + \mathrm{e}^{-z}} \tag{1.1}$$

La elección de esta función no es única, pudiendo tomar entre otros ejemplos la tangente hiperbólica. Más adelante (sección 1.3) veremos los requisitos que ha tener una función para poder ser elegida como función de activación.

Del mismo modo, podemos reabsorber la barrera de activación para introducirla dentro de la función de transferencia para simplificar el proceso. Esto es, la excitación neuronal se consigue para valores positivos de las activaciones: $h(\sum_j w_i j x_j - b) > 0$. Por otro lado, para aligerar la notación y ecuaciones futuras, el sesgo puede verse como un peso más: $w_0 := -b$. Para ello, se aumenta el tamaño del vector de entrada en una unidad definiendo $x_0 = 1$. De este modo, la salida de la neurona consiste simplemente en $h(\sum_k w_k x_k)$.

Una vez que se han definido las unidades básicas de pensamiento, se procede a realizar la conexión entre estas para formar la *red neuronal* y aumentar su rendimiento. Una primera forma de realizarlo consiste en entrelazar todas las neuronas entre sí. Sin embargo, este enfoque claramente conduce a la aparición de bucles, lo que hace que el resultado de la red sea complejo y definido arbitrariamente. Claramente resulta mucho más eficiente crear varias capas de procesamiento. Por tanto, se seleccionan inicialmente algunas neuronas que realizarán el primer nivel de procesamiento partiendo del input. El resultado de estas neuronas se toma como entrada para la siguiente capa de neuronas, y este proceso se repite recursivamente hasta obtener el resultado final de la red.

Cada uno de los grupos de neuronas procesando secuencialmente la información es denominado *capa*. En especial, las capas intermedias entre el input y el output son denominadas como *capas ocultas*. Así, se puede crear un sistema de capas de neuronas, parecido al que se muestra en la figura 1.2. En este caso solo se presenta una capa oculta. Con esta estructura cada capa extrae características más abstractas y complejas a partir de las características extraídas por las capas anteriores.

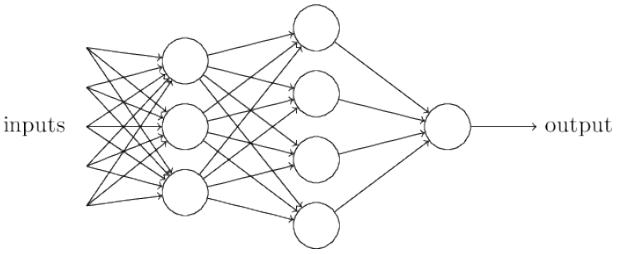


Figura 1.2: M. Nielsen (2019). Ilustración de una red neuronal simple. En este caso solo se tiene una capa oculta. Se trata de una red totalmente conectada, con un output único.

Este sistemas de capas es el que se define por *arquitectura*. Si, tal como en la figura, se asume la existencia de conexiones entre todas las neuronas de una capa con las constituyentes de la capa posterior, la red obtenida es definida como *totalmente conectada*[3].

Se observa que la elección del número de neuronas para la capa de input y output viene de cierta manera predefinidas (el número de neuronas para el input coincide con la dimensionalidad del vector $\mathbf{x}$, mientras que el de ouput coincide con la dimensionalidad deseada del vector resultante. Así, para hacer la clasificación de los datos en m grupos, se tomará una capa final de m neuronas). No obstante, no hay una regla o convención para la elección del número de capas ocultas y su cantidad de neuronas.

Dando una forma más concreta a todo lo explicado hasta el momento, definimos:

Definición 1.1. *Se denota al peso de la conexión que entrelaza la neurona j de la capa $(l-1)$-ésima con la neurona i de la capa l-ésima como $w_{ij}^{(l)}$. El sesgo de la neurona i de la capa l es denotado por $b_i^{(l)} = w_{i0}^{(l)}$.*

Del mismo modo, la matriz de pesos de la capa l es denotada por $\mathbf{w}^{(l)}$. Finalmente, denotamos por $\mathbf{w}$ al conjunto de todos los pesos. Este será tratado usualmente como un vector $\mathbf{w} \in \mathbb{R}^W$, siendo W el número de conexiones total.

Esta definición de índices, aunque a priori puedan parecer un poco confusa, simplifica enormemente las fórmulas. Podemos recapitular y definir las siguientes magnitudes:

$$\text{transferencias} \qquad z_k^{(l)} = \sum_{j=1}^{M} w_{kj}^{(l)} a_j^{(l-1)} + w_{k0}^{(l)} \tag{1.2}$$

$$\text{activaciones} \qquad a_j^{(l)} = h(z_j^{(l)}) \tag{1.3}$$

Estas dos ecuaciones pueden ser reescritas como igualdades matriciales, denotando $\mathbf{a}^{(l)}$ y $\mathbf{z}^{(l)}$ el vector de activaciones y transferencias respectivamente. Dado que $a_j^{(0)}$ coincide con el input de la red, x_j, se puede calcular recursivamente las transferencias y respectivas activaciones hasta alcanzar la capa de output. Por ejemplo, para una red neuronal con una única capa oculta, el vector de salida es:

$$y_k(\mathbf{x},\mathbf{w}) = \tilde{h}_k \left(\sum_{j=1}^{\tilde{M}} w_{kj}^{(2)} h_j \left(\sum_{i=1}^{M} w_{ji}^{(1)} x_i + b_j^{(1)} \right) + b_k^{(2)} \right) \tag{1.4}$$

Como se aprecia, partiendo del vector de inputs $\mathbf{x}$, se hace propagar "hacia adelante" la información por la red, pasando primeramente por la capa de los inputs (función de activación h_j) y después por la capa oculta ($\tilde{h}_k$) para dar lugar al output de la red (y_k). Evidentemente en el caso de tener más de una capa la ecuación es análoga. En el próximo capítulo llamaremos *feedforward propagation*, o propagación hacia adelante, a este proceso.

[3]Aunque hay una gran diversidad de tipos de redes neuronales, en este documento estudiaremos únicamente este tipo, por ser las más generales.

Para terminar la sección, es conveniente ahora resaltar que una red no es más que una composición de funciones; y lo que realmente caracteriza a estas -una vez definidas la arquitectura y funciones de activación- es la magnitud de sus pesos (incluyendo aquí los sesgos). Para obtener un correcto funcionamiento de la red neuronal, por tanto, basta ajustar dichos parámetros; lo que es autónomamente realizado por la red gracias al algoritmo de optimización y conjunto de entrenamiento proporcionados. Hemos conseguido reducir muchísimos problemas informáticos al simple caso de ajuste y evaluación de funciones.

De esta manera, como es habitual en los problemas de optimización, para el ajuste de los parámetros se toma una función de error o de coste que estudie la diferencia existente entre el output proporcionado por la red y el resultado real conocido del conjunto de entrenamiento. Se obtiene un correcto ajuste de los pesos gracias a la minimización de esta diferencia. Un ejemplo habitual resulta la función de mínimos cuadrados:

$$C = \frac{1}{2N} \sum_{i=1}^{N} (a_i^{(L)} - y_i)^2 \tag{1.5}$$

donde y_i es la componente i-ésima del vector de etiquetas $\mathbf{y}$ del conjunto de entrenamiento (compuesto de N puntos) y $a_i^{(L)}$ es la misma componente del output resultante de la red al introducir el input correspondiente a esta etiqueta, $\mathbf{x}$ (siendo L el número de capas de la red). Observamos en este output una dependencia implícita en el valor de los pesos.

El anterior proceso de ajuste de parámetros es lo que se denomina *fase de entrenamiento* y es lo que determina la precisión futura de la red. Se verá con más detalle en el siguiente capítulo los métodos propuestos para ello.

1.2. No Free Lunch Theorem

El *No Free Lunch Theorem* es un resultado fundamental en el campo de la optimización que establece que, en promedio, todos los algoritmos de optimización son igualmente efectivos para encontrar la solución óptima para un problema cualquiera dado. En otras palabras, no hay un algoritmo óptimo común a todos los problemas.

Esto resalta la necesidad de la construcción de diversos métodos, algoritmos y arquitecturas para las redes neuronales. Citando algunos ejemplos, para la clasificación de imágenes es común la aplicación de las redes *convolucionales*. Estas son similares a las totalmente conectadas que hemos descrito, pero se incluyen además unas capas adicionales especializadas en la extracción de características de las imágenes. Por otro lado, las *redes recurrentes* son empleadas para la realización de tareas secuenciales de procesamiento de datos, como puede ser el reconocimiento de voz o procesamiento de lenguaje natural. Su arquitectura básica es esencialmente idéntica a las totalmente conectadas, con la salvedad de permitir la creación de bucles controlados en las conexiones entre las capas.

Por último, las redes totalmente conectadas son las más generales existentes, no estando especializadas en el tipo de datos a tratar o problema a resolver. Aunque no se alcancen los mejores resultados, estas resultan ser polivalentes y por tanto, a falta de información detallada sobre un problema, serán las redes a implementar. Son estas las redes que se analizan en este documento.

1.3. Teorema de aproximación universal

Para concluir el capítulo, vamos a enunciar el resultado fundamental de las redes neuronales, llamado el *teorema de Aproximación Universal*. Este teorema establece la capacidad de una red neuronal con una sola capa oculta, un número suficientemente grande de neuronas y una función de activación adecuada de aproximar cualquier función continua en un conjunto compacto.

Más informalmente, esto se puede entender como la habilidad de imitación de la capacidad de razonamiento humano, objetivo propuesto por la comunidad. Entre las acepciones de inteligencia dentro del diccionario, se encuentra la "habilidad de localizar patrones". Desde una perspectiva matemática, esto puede traducirse como el ajuste de puntos a una función desconocida o como el problema de clasificación de puntos. El resultado afirma que, sea cual sea esta función, una red es capaz de aproximarla.

Veamos por tanto el enunciado de este teorema. Primeramente se debe de dar la siguiente definición [1]:

Definición 1.2 (Función discriminatoria)**.** *Sean $I_n = [0,1]^n$, $M(I_n)$ el espacio finito de medidas de Borel regulares con signo en I_n y $\mathcal{C}(I_n)$ el espacio de funciones continuas en I_n con respecto a la norma del supremo. Entonces se dice que una función es* discriminatoria *si dada una medida $\mu \in M(I_n)$ tal que:*

$$\int_{I_n} \sigma(\mathbf{w}^T\mathbf{x})\, d\mu(\mathbf{x}) = 0 \quad \forall \mathbf{w} \in \mathbb{R}^{n+1} \tag{1.6}$$

Necesariamente implique que $\mu \equiv 0$

Este es el principal requerimiento que debe de cumplir una función para poder ser considerada como una función de activación. Con ella, la función sigmoide y la tangente hiperbólica son discriminatorias. Gracias a este definición podemos enunciar el Teorema de Aproximación:

Teorema 1.1 (Cybenko)**.** *Sea σ cualquier función continua discriminatoria. Entonces las sumas finitas de la siguiente forma:*

$$G(x) = \sum_{j=1}^{N} \alpha_j \sigma(w_j^T x) \quad w_j \in \mathbb{R}^{n+1}, \alpha_j \in \mathbb{R} \tag{1.7}$$

Son densas en $\mathcal{C}(I_n)$

Corolario 1.1. *Usando el teorema 1.1, dado cualquier $\epsilon > 0$ y $f \in \mathcal{C}(I_n)$, existe una suma de la forma 1.7, $G(x)$, que aproxime la función f con una precisión ϵ. Esto es:*

$$|G(x) - f(x)| < \epsilon \quad \forall x \in I_n \tag{1.8}$$

El teorema 1.1 de Cybenko tiene adicionalmente una versión para funciones de clasificación, por lo que además existe una red de este tipo capaz de resolver cualquier problema de clasificación.

De este resultado se concluye que una red neuronal con funciones de activación sigmoides (por ejemplo) con una sola capa puede aproximar cualquier función dada. Téngase en cuenta que solo se asegura la existencia, no ahondando en la arquitectura exacta de la red ni número de neuronas. Con todo, no resulta un resultado extremadamente revelador pero sí tranquilizador, pues ayuda a formar firmemente las bases de las redes neuronales, asegurando que son capaces de cumplir las funcionalidades que motivaron su creación.

Capítulo 2

Entrenamiento de redes neuronales

En el anterior capítulo se han definido y dado las primeras nociones acerca de las redes neuronales. Entre otras cosas, se ha subrayado la importancia de la fase de entrenamiento y en qué esta se trata de un "simple" ajuste de parámetros similar al de otros problemas de optimización. En el presente capítulo se pretende ahondar en dicha cuestión, explicando con detalle las diferentes técnicas que primero surgieron y forman la base de las actuales técnicas más avanzadas.

2.1. Ajuste de parámetros

Los *optimizadores* son algoritmos fundamentales en el entrenamiento de redes neuronales. Su objetivo es el ajuste de los parámetros internos de la red -i.e. los pesos de las conexiones- para minimizar la función de coste. Aunque se está tratando con el simple problema de optimización de una función de coste, en la práctica su solución no es inmediata de implementar debido a la gran dimensionalidad del problema. Como ilustración, para una red neuronal con 2 capas ocultas de 10 neuronas cada una, una capa de de output de 2 neurona y un input de dimensión 100, se requiere realizar el ajuste de 1120 pesos y 122 sesgos: 1242 parámetros en total. Esta red, además, es una de las más sencillas que se pueden encontrar, por no decir que raramente podría ser utilizada para una aplicación real. Esto sumado al hecho de que se usan amplios conjuntos de entrenamiento, consistentes en cientos de miles o incluso millones de puntos para realizar el ajuste, conlleva que se usen métodos de aproximación para localizar el mínimo de la función de coste. Estos son precisamente los optimizadores.

Frecuentemente estos algoritmos presentan unas constantes llamadas *hiperparámetros*. Estos son parámetros externos que no se aprenden directamente del conjunto de datos durante el proceso de entrenamiento de un modelo de aprendizaje automático, sino que son establecidos previamente a la resolución del problema. Por ejemplo, pese a no formar parte de un algoritmo optimizador como tal, el número de capas ocultas o neuronas son parámetros que se definen previo a la construcción de la red neuronal y por tanto pueden ser considerados como hiperparámetros. En general, no se disponen de técnicas precisas para su elección, sino que se escogen mediante métodos heurísticos [2].

2.1.1. Descenso del gradiente

Entre los algoritmos optimizadores se destaca indudablemente por encima del resto al método del *Descenso del gradiente* (o *gradient descent* en inglés) debido a su sencillez y amplio uso en la comunidad. Esta técnica, además, ha servido de inspiración para el desarrollo de otros optimizadores más sofisticados y precisos.

Es bien conocido que el gradiente indica aquella dirección que presenta un cambio más pronunciado y proporciona una magnitud sobre este mismo. Contrariamente, aquella dirección de mayor "descenso" resulta

Algoritmo 2.1: DESCENSO DEL GRADIENTE

Input:
Tamaño de paso: α
Conjunto de entrenamiento: D=($\mathbf{x}$, $\mathbf{y}$):=$\{(x_i, y_i) : i = 1, 2..., N\}$
Función de coste: C
Parámetros iniciales: $\mathbf{w}_0$

Output:
Parámetros óptimos: $\mathbf{w}$

Inicialización:
Iteración: $t \leftarrow 0$

while $\mathbf{w}_t$ *no converge* **do**
$t+1 \leftarrow t$
$\mathbf{w}_t \leftarrow \mathbf{w}_{t-1} - \alpha \nabla_{\mathbf{w}} C(\mathbf{w}_{t-1}; \mathbf{x}, \mathbf{y})$
end
return $\mathbf{w}_t$

Nota: La convergencia de $\mathbf{w}_t$ se puede hacer en base a la diferencia $\mathbf{w}_t - \mathbf{w}_{t-1}$, entre otras muchas opciones.

su dirección opuesta. Con esto en mente, este optimizador propone calcular el gradiente de la función de pérdida para modificar los parámetros "moviéndolos" en la dirección opuesta a $\nabla_{\mathbf{w}} C(\mathbf{w})$. Con esto, la lógica del método de ajuste de parámetros se puede ver en el pseudoalgoritmo 2.1.

Como se ve, la magnitud del avance en la dirección opuesta del gradiente viene determinada por el hiperparámetro α, denominado *tasa de aprendizaje* (o *learning rate* en inglés). Este, al igual que prácticamente cualquier hiperparámetro, es una magnitud sensible que condiciona fuertemente la eficiencia de la red. Una tasa de entrenamiento excesivamente pequeña resulta en una lenta convergencia, e incluso la incapacidad de alcanzar la solución óptima por riesgo al estacionamiento de los pesos en un punto crítico no minino global. Contrariamente, una tasa de aprendizaje elevada puede desestabilizar el proceso de optimización resultando en divergencias. Esto es debido a que este método se fundamenta en la aproximación $\Delta f(x) = f(x_1) - f(x_0) = \nabla f(x_0) \cdot \Delta x + \mathcal{O}((\Delta x)^2)$.

Este algoritmo alivia enormemente el peso computacional gracias a su simplicidad. No obstante, se destaca la necesidad de cálculo de las derivadas parciales respecto de cada uno de los pesos. Esto, pese a no ser una tarea sencilla, se puede realizar mediante el uso de la *propagación hacia atrás* (*Backward Propagation* en inglés), presente en el algoritmo 2.2.

Una vez que se ha visto la intuición y motivación de este optimizador, demostremos su orden de convergencia.

Teorema 2.1. *Sea $C : \mathbb{R}^d \to \mathbb{R}$ una función L-lipschitziana convexa cuyo mínimo se alcanza para $\mathbf{w}^* = argmin_{\mathbf{w}} C(\mathbf{w})$. Entonces, con la regla de actualización del descenso del gradiente definido en el Algoritmo 2.1 con una tasa de aprendizaje verificando $\alpha \leq 1/L$, se tiene la siguiente cota de convergencia para la iteración k-ésima:*

$$C(\mathbf{w}_k) \leq C(\mathbf{w}^*) + \frac{\|\mathbf{w}_0 - \mathbf{w}^*\|_2^2}{2\alpha k} \tag{2.1}$$

En particular, con $\frac{L\|\mathbf{w}_0 - \mathbf{w}^*\|_2^2}{\varepsilon}$ iteraciones se puede encontrar una aproximación ε óptima del valor de $\mathbf{w}$

Demostración: Primeramente, por ser convexa y lipschitziana la función de coste, podemos escribir[1]:

$$C(\mathbf{w}_{i+1}) \leq C(\mathbf{w}_i) + \langle \nabla C(\mathbf{w}_i), \mathbf{w}_{i+1} - \mathbf{w}_i \rangle + \frac{L}{2} \|\mathbf{w}_{i+1} - \mathbf{w}_i\|_2^2 \tag{2.2}$$

[1]En caso de necesidad, se puede ver una prueba de esta igualdad en el apéndice A - lema A.1

Empleando la regla concreta de actualización de los pesos, se puede continuar con la acotación de la función de coste como:

$$C(\mathbf{w}_{i+1}) \leq C(\mathbf{w}_i) - \alpha \|\nabla C(\mathbf{w}_i)\|_2^2 + \frac{L\alpha^2}{2} \|\nabla C(\mathbf{w}_i)\|_2^2 \overset{L\alpha \leq 1}{\leq} C(\mathbf{w}_i) - \frac{\alpha}{2} \|\nabla C(\mathbf{w}_i)\|_2^2 \tag{2.3}$$

Esta desigualdad verifica el no crecimiento de la función de coste con el empleo del descenso del gradiente. Por otro lado, usando la convexidad de C:

$$C(\mathbf{w}_i) \leq C(\mathbf{w}^*) + \langle \nabla C(\mathbf{w}_i), \mathbf{w}_i - \mathbf{w}^* \rangle = C(\mathbf{w}^*) + \frac{1}{\alpha} \langle \mathbf{w}_i - \mathbf{w}_{i+1}, \mathbf{w}_i - \mathbf{w}^* \rangle \tag{2.4}$$

La última igualdad proviene de la relación $\mathbf{w}_{i+1} = \mathbf{w}_i - \alpha \nabla C(\mathbf{w}_i)$. Combinando las dos desigualdades:

$$\begin{aligned}
C(\mathbf{w}_{i+1}) &\leq C(\mathbf{w}^*) + \langle \nabla C(\mathbf{w}_i), \mathbf{w}_i - \mathbf{w}^* \rangle - \frac{\alpha}{2} \|\nabla C(\mathbf{w}_i)\|_2^2 \\
&= C(\mathbf{w}^*) + \frac{1}{\alpha} \cdot \langle \mathbf{w}_i - \mathbf{w}_{i+1}, \mathbf{w}_i - \mathbf{w}^* \rangle - \frac{1}{2\alpha} \|\mathbf{w}_i - \mathbf{w}_{i+1}\|^2 \\
&= C(\mathbf{w}^*) + \frac{1}{2\alpha} \|\mathbf{w}_i - \mathbf{w}^*\|_2^2 - \frac{1}{2\alpha} \left(\|\mathbf{w}_i - \mathbf{w}^*\|_2^2 - 2 \langle \alpha \nabla C(\mathbf{w}_i), \mathbf{w}_i - \mathbf{w}^* \rangle + \|\alpha \nabla C(\mathbf{w}_i)\|_2^2 \right) \\
&= C(\mathbf{w}^*) + \frac{1}{2\alpha} \|\mathbf{w}_i - \mathbf{w}^*\|_2^2 - \frac{1}{2\alpha} \|\mathbf{w}_i - \mathbf{w}^* - \alpha \nabla C(\mathbf{w}_i)\|_2^2 \\
&= C(\mathbf{w}^*) + \frac{1}{2\alpha} \left(\|\mathbf{w}_i - \mathbf{w}^*\|_2^2 - \|\mathbf{w}_{i+1} - \mathbf{w}^*\|_2^2 \right)
\end{aligned} \tag{2.5}$$

Tomando el sumatorio sobre todas las iteraciones $i = 0, \ldots, k-1$ en ambos lados:

$$\sum_{i=0}^{k-1} \left(C(\mathbf{w}_{i+1}) - C(\mathbf{w}^*) \right) \leq \frac{1}{2\alpha} \left(\|\mathbf{w}_0 - \mathbf{w}^*\|_2^2 - \|\mathbf{w}_k - \mathbf{w}^*\|_2^2 \right) \leq \frac{\|\mathbf{w}_0 - \mathbf{w}^*\|_2^2}{2\alpha} \tag{2.6}$$

Como hemos concluido que C es monótona no creciente, se tiene $C(\mathbf{w}_k) - C(\mathbf{w}^*) \leq C(\mathbf{w}_i) - C(\mathbf{w}^*)$ para cualquier iteración $k > i$. Sustituyendo lo anterior se finaliza la demostración.

■

2.1.2. Backpropagation

Como se ha comentado, el método del descenso del gradiente requiere del uso del gradiente de la función de coste respecto de los pesos -incluyendo a los sesgos- de la red en el conjunto de puntos de entrenamiento $D = (\mathbf{x}, \mathbf{y}) = \{(x_i, y_i) : i = 1, 2, 3, ..., N\}$. El objetivo del algoritmo estudiado en esta sección es precisamente el de calcular esta información de una manera eficaz.

La técnica empleada para ello es denominada *Backward Propagation*, o acortando *Backpropagation* -propagación hacia atrás o retropropagación-. Para su aplicación, la función de coste debe de satisfacer las siguientes hipótesis:

1. Se debe de poder descomponer la función de coste como la media de los coste individuales de los distintos inputs. Esto es: $C(\mathbf{x}, \mathbf{y}) = \sum_{i=1}^{N} C(x_i, y_i)/N$
2. Del mismo modo, la función de coste debe de depender exclusivamente de las activaciones de la capa final. En otras palabras, el coste solo depende del output predicho.

Nótese que estas hipótesis no suponen una ligadura altamente restrictiva, siendo la mayoría de las funciones de coste tradicionalmente empleadas aptas para su uso. En concreto, la función de coste de error cuadrático satisface ambas condiciones.

Algoritmo 2.2: BACKPROPAGATION

Input:

Punto de entrenamiento: (x_i, y_i)
Función de coste: C
Derivada de la función de coste: C'
Función de activación: h
Derivada de la función de activación: h'

Output:

Derivada de la función de coste en el punto (x_i, y_i): $\frac{\partial C(x_i, y_i)}{\partial \mathbf{w}}$

Inicialización: obtención de activaciones y transferencias para (x_i, y_i) (feedforward propagation)

$a^{(0)} = x_i$
for $l = 1$ ***to*** L **do**
  $\mathbf{z}^{(l)} \leftarrow \mathbf{w}^{(l)} a^{(l-1)}$
  $\mathbf{a}^{(l)} \leftarrow h(\mathbf{z}^{(l)})$
end

Retropropagación del error y derivadas parciales:

Error en la capa de salida: $\delta^{(L)} \leftarrow \frac{\partial C'(a^{(L)}, y_i)}{\partial a^{(L)}} h'(\mathbf{z}^{(L)})$
for $l = L - 1$ ***to*** 1 **do**
  $\delta^{(l)} \leftarrow h'(\mathbf{z}^{(l)}) \cdot (\mathbf{w}^{(l+1)\top} \cdot \delta^{(l+1)})$
  $\frac{\partial C}{\partial \mathbf{w}^{(l)}} \leftarrow \mathbf{a}^{(l-1)\top} \cdot \delta^{(l)}$
  $\frac{\partial C}{\partial \mathbf{b}^{(l)}} \leftarrow \delta^{(l)}$
end

return $\frac{\partial C}{\partial \mathbf{w}} = \left(\frac{\partial C}{\partial \mathbf{w}^{(l)}}, \frac{\partial C}{\partial b^{(l)}}\right)_{l=0}^{N-1}$

Procedamos finalmente a calcular las derivadas parciales de la función de coste para cada una de las capas y conexiones neuronales. Vamos a separar por conveniencia los pesos de los sesgos. Empleando la regla de la cadena junto con la definición de transferencia:

$$\frac{\partial C_x}{\partial w_{ij}^{(l)}} = \frac{\partial C_x}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{ij}^{(l)}} \quad \text{y} \quad \frac{\partial C_x}{\partial b_j^{(l)}} = \frac{\partial C_x}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial b_j^{(l)}} \tag{2.7}$$

Es conveniente definir el factor común a ambas ecuaciones como el error j-ésimo de la capa l:

$$\delta_j^{(l)} := \frac{\partial C}{\partial z_j^{(l)}} \tag{2.8}$$

Según nuestras hipótesis, la función de coste depende exclusivamente del output de la última capa, denotado por $z^{(L)}$. Con esto se observa que los errores de la última capa son directamente calculables sin necesidad de operaciones intermedias:

$$\delta_j^{(L)} = \frac{\partial C(a^{(L)})}{\partial z_j^{(L)}} \overset{(1,3)}{=} \frac{\partial C(a^{(L)})}{\partial a^{(L)}} \cdot h'(z_j^{(L)}) \tag{2.9}$$

Por el contrario, los errores de las capas ocultas e input, no son tan fácilmente calculables, pues dependen de capas más avanzadas debido al feedforward propagation (1.3). Sin embargo, haciendo uso de la regla de la cadena, podemos recursivamente calcular el error de cada capa moviéndonos desde atrás (última capa de la red) hacia adelante (primera capa de la red), es decir, calcular $\delta_l^{(l)}$ a partir del conocido $\delta_l^{(l+1)}$:

$$\delta_j^{(l)} :\overset{(2,8)}{=} \frac{\partial C}{\partial z_j^{(l)}} = \sum_k \frac{\partial C}{\partial z_k^{(l+1)}} \frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}} = \sum_k \delta_k^{(l+1)} \underbrace{\frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}}}_{(*)} \tag{2.10}$$

Este último término $(*)$, lo calculamos aparte. Teniendo en cuenta la definición explícita de transferencia:

$$z_k^{(l+1)} = \sum_i w_{ki}^{(l+1)} a_i^{(l)} \overset{(1,3)}{=} \sum_i w_{ki}^{(l+1)} h\left(z_i^{(l)}\right) \tag{2.11}$$

Tomando derivadas parciales, podemos retomar la expresión (2.10) para obtener:

$$\delta_j^{(l)} = \sum_k \delta_k^{(l+1)} \cdot w_{kj}^{(l+1)} h'\left(z_j^{(l)}\right) = h'\left(z_j^{(l)}\right) \sum_k \delta_k^{(l+1)} \cdot w_{kj}^{(l+1)} \tag{2.12}$$

Esta ecuación proporciona una manera de realizar el cálculo de los errores de la capa l suponiendo conocidos los mismos para la capa posterior $l+1$. Así, partiendo desde la capa final (2.9), permite el cálculo de todos los errores de la red.

Por otra parte, el cálculo del término restante en (2.7) es inmediato mediante el uso de (1.2) , resultando:

$$\frac{\partial z_j^{(l)}}{\partial w_{ij}^{(l)}} = a_i^{(l)} \qquad \frac{\partial z_j^{(l)}}{\partial b_j^{(l)}} = 1 \tag{2.13}$$

Retomando (2.7) con la información obtenida:

$$\frac{\partial C_x}{\partial w_{ij}^{(l)}} = a_j^{(l-1)} \delta_i^{(l)} \tag{2.14}$$

$$\frac{\partial C_x}{\partial b_i^{(l)}} = \delta_i^{(l)} \tag{2.15}$$

Con esto, hemos probado la veracidad del pseudoalgoritmo 2.2.

2.1.3. Descenso del gradiente estocástico.

El método del descenso del gradiente 2.1.1, aunque eficaz, depende del tamaño del conjunto de entrenamiento, lo que lo convierte en demasiado pesado computacionalmente para su aplicación en la mayoría de situaciones. Recuérdese que es recomendado disponer de un gran conjunto de entrenamiento para poder capturar bien los patrones generales de los puntos de entrenamiento. Debido a la primera de las hipótesis del método de Backward Propagation, para calcular el gradiente total se toma el sumatorio de los gradientes calculados individualmente en cada punto del conjunto de entrenamiento. Por tanto, el coste computación es $\mathcal{O}(NW)$ siendo N el número de puntos del conjunto de entrenamiento y W el número de parámetros a ajustar.

Como solución al problema de escalabilidad del conjunto de entrenamiento, se propone el uso del descenso del gradiente estocástico (DGE). Este se basa en realizar una aproximación del gradiente de la función de coste tomando aleatoriamente un subconjunto del conjunto de entrenamiento de menor tamaño. Es esta estimación la que se empleará en cada iteración para la actualización de parámetros. Es decir, en vez de trabajar con un conjunto inicial dado $D = \{(\mathbf{x}, \mathbf{y})\} = \{(x_i, y_i) | i = 1, 2...N\}$, podemos fijar un *tamaño de lote n* (o *mini-batch size* en inglés) y trabajar con mini conjuntos que son escogidos en cada iteración al azar

Algoritmo 2.3: Descenso del Gradiente Estocástico (DGE)

Input:
Tamaño de paso: α
Número de iteraciones: *epochs*
Tamaño del mini-batch: n
Conjunto de datos de entrenamiento: $D = (\mathbf{x}, \mathbf{y}) := \{(x_i, y_i \ : i \ = 1, 2..., N\}$
Función de coste: C
Parámetros iniciales: $\mathbf{w}_0$

Output:
Parámetros óptimos: $\mathbf{w}$

Inicialización:
Iteración: $t \leftarrow 0$

Descenso:
for *t 1* ***to*** *epochs* **do**
  Seleccionar un mini-batch aleatorio: $(x^{(j)}, y^{(j)})_{j=1}^{n}$
  $\mathbf{w}_t \leftarrow \mathbf{w}_{t-1} - \frac{\alpha}{n} \sum_{j=1}^{n} \nabla_w C(\mathbf{w}_{t-1}; x^{(j)}, y^{(j)})$
end
return $\mathbf{w}_{epochs}$

$S = \{(x_j, y_j) | (x_j, y_j) \in D \ \& \ j = 1, 2...n\}$. De este modo, se intercambia el uso del gradiente de la función de coste por el siguiente estimador:

$$\nabla_{\mathbf{w}} C(\mathbf{x}, \mathbf{y}) = \frac{1}{N} \sum_{(x_j, y_j) \in D} \nabla_{\mathbf{w}} C(x_j, y_j) \simeq \frac{1}{n} \sum_{(x_j, y_j) \in S} \nabla_{\mathbf{w}} C(x_j, y_j) := \boldsymbol{v} \tag{2.16}$$

Dicho proceso se itera un número de veces prefijado denominado *número de épocas*. A cada iteración de este método se le suele denominar *época*. En virtud de la ley de los grandes números, iterando este proceso un número suficientemente grande de veces, el DGE alcanza la misma solución que aquella proporcionada por el método del descenso del gradiente original. De este modo, la lógica del método viene descrita en el pseudoalgoritmo 2.3.

Para terminar, se estudia una prueba del orden de convergencia del DGE. Para simplificar la demostración, se tomará una variante en la cual se toma como estimación de los pesos resultantes del optimizador la media de los pesos obtenidos en las diferentes iteraciones, en lugar de la estimación de la última iteración. Es decir, el resultado final es $\bar{\boldsymbol{w}}_{epochs} = \frac{1}{epochs} \sum_{i=1}^{epochs} \mathbf{w}_i$. En cualquier caso, se observa que la cota que vamos a demostrar en el siguiente resultado se satisface igualmente para aquella estimación $\mathbf{w}_i$ con el menor valor de la función de coste, por lo que no se resta generalidad respecto del método general.

Teorema 2.2 (Convergencia del DGE)**.** *Sea $C : \mathbb{R}^d \to \mathbb{R}$ una función L-lipschitziana convexa cuyo mínimo se alcanza en $\mathbf{w}^* = argmin_{\mathbf{w}} C(\mathbf{w})$. Supongamos que el estimador del gradiente empleado en el DGE tiene una varianza acotada: $Var(\boldsymbol{v}_i) \leq \sigma^2$ para toda iteración i. Entonces, la regla de actualización del descenso del gradiente estocástico definido en el Algoritmo 2.3 modificado como se ha mencionado en el anterior párrafo con una tasa de entrenamiento $\alpha \leq 1/L$ presenta la siguiente cota de convergencia para la iteración k-ésima:*

$$\mathbb{E}\left[C\left(\bar{\boldsymbol{w}}_k\right)\right] \leq C\left(\mathbf{w}^*\right) + \frac{\left\|\mathbf{w}_0 - \mathbf{w}^*\right\|_2^2}{2\alpha k} + \alpha\sigma^2 \tag{2.17}$$

En consecuencia, con $\left(\sigma^2 + L\left\|\mathbf{w}_0 - \mathbf{w}^*\right\|_2^2\right)^2 / \varepsilon^2$ iteraciones se puede alcanzar una solución 2ε-aproximada de la solución óptima usando $\alpha = 1/\sqrt{k}$.

Demostración: Tomando nuevamente el lema A.1, en esta ocasión usando la regla de actualización adaptada

al estimador estocástico:

$$\begin{aligned} C\left(\mathbf{w}_{i+1}\right) &\leq C\left(\mathbf{w}_i\right) + \left\langle \nabla C\left(\mathbf{w}_i\right), \mathbf{w}_{i+1} - \mathbf{w}_i \right\rangle + \frac{L}{2}\left\|\mathbf{w}_{i+1} - \mathbf{w}_i\right\|_2^2 \\ &= C\left(\mathbf{w}_i\right) - \alpha \left\langle \nabla C\left(\mathbf{w}_i\right), \boldsymbol{v_i} \right\rangle + \frac{L\alpha^2}{2}\left\|\boldsymbol{v}_i\right\|_2^2. \end{aligned} \tag{2.18}$$

Tomando esperanzas y teniendo en cuenta que $\mathrm{Var}[\boldsymbol{v}_i] = \mathbb{E}\left[\|\boldsymbol{v}_i\|_2^2\right] - \|\mathbb{E}[\boldsymbol{v}_i]\|_2^2$ y $\mathbb{E}[\boldsymbol{v}_i] = \nabla C\left(\mathbf{w}_i\right)$:

$$\begin{aligned} \mathbb{E}\left[C\left(\mathbf{w}_{i+1}\right)\right] &\leq C\left(\mathbf{w}_i\right) - \alpha\left\|\nabla C\left(\mathbf{w}_i\right)\right\|_2^2 + \frac{L\alpha^2}{2}\left(\left\|\nabla C\left(\mathbf{w}_i\right)\right\|_2^2 + \mathrm{Var}\left(\boldsymbol{v}_\alpha\right)\right) \\ &\leq C\left(\mathbf{w}_i\right) - \alpha\left(1 - \frac{L\alpha}{2}\right)\left\|\nabla C\left(\mathbf{w}_i\right)\right\|_2^2 + \frac{L\alpha^2}{2}\sigma^2 \\ &\overset{L\alpha\leq 1}{\leq} C\left(\mathbf{w}_i\right) - \frac{\alpha}{2}\left\|\nabla C\left(\mathbf{w}_i\right)\right\|_2^2 + \frac{\alpha}{2}\sigma^2, \end{aligned} \tag{2.19}$$

En esta ocasión, de aquí no podemos concluir que la sucesión de valores predichos sea no creciente. Combinando ambas ecuaciones:

$$\mathbb{E}\left[C\left(\mathbf{w}_{i+1}\right)\right] \leq C\left(\mathbf{w}^*\right) + \left\langle \nabla C\left(\mathbf{w}_i\right), \mathbf{w}_i - \mathbf{w}^* \right\rangle - \frac{\alpha}{2}\left\|\nabla C\left(\mathbf{w}_i\right)\right\|_2^2 + \frac{\alpha}{2}\sigma^2 \tag{2.20}$$

Introduciendo $\boldsymbol{v_i}$ gracias a $\mathbb{E}\left[\boldsymbol{v}_i\right] = \nabla C\left(\mathbf{w}_i\right)$ y $\left\|\nabla C\left(\mathbf{w}_i\right)\right\|_2^2 = \mathbb{E}\left[\left\|\boldsymbol{v}_i\right\|_2^2\right] - \mathrm{Var}\left(\boldsymbol{v}_i\right) \leq \mathbb{E}\left[\left\|\boldsymbol{v}_i\right\|_2^2\right] - \sigma^2$:

$$\begin{aligned} \mathbb{E}\left[C\left(\mathbf{w}_{i+1}\right)\right] &\leq C\left(\mathbf{w}^*\right) + \left\langle \mathbb{E}\left[\boldsymbol{v}_i\right], \mathbf{w}_i - \mathbf{w}^* \right\rangle - \frac{\alpha}{2}\mathbb{E}\left[\left\|\boldsymbol{v}_i\right\|_2^2\right] + \alpha\sigma^2 \\ &= C\left(\mathbf{w}^*\right) + \mathbb{E}\left[\left\langle \boldsymbol{v}_i, \mathbf{w}_i - \mathbf{w}^* \right\rangle - \frac{\alpha}{2}\left\|\boldsymbol{v}_i\right\|_2^2\right] + \alpha\sigma^2 \\ &\leq C\left(\mathbf{w}^*\right) + \mathbb{E}\left[\frac{1}{2\alpha}\left(\left\|\mathbf{w}_i - \mathbf{w}^*\right\|_2^2 - \left\|\mathbf{w}_i - \mathbf{w}^* - \alpha\boldsymbol{v}_i\right\|_2^2\right)\right] + \alpha\sigma^2 \\ &= C\left(\mathbf{w}^*\right) + \mathbb{E}\left[\frac{1}{2\alpha}\left(\left\|\mathbf{w}_i - \mathbf{w}^*\right\|_2^2 - \left\|\mathbf{w}_{i+1} - \mathbf{w}^*\right\|_2^2\right)\right] + \alpha\sigma^2 \end{aligned} \tag{2.21}$$

Sumando a todas las iteraciones $i = 0, \ldots, k-1$:

$$\sum_{i=0}^{k-1}\left(\mathbb{E}\left[C\left(\mathbf{w}_{i+1}\right)\right] - C\left(\mathbf{w}^*\right)\right) \leq \frac{1}{2\alpha}\left(\left\|\mathbf{w}_0 - \mathbf{w}^*\right\|_2^2 - \mathbb{E}\left[\left\|\mathbf{w}_k - \mathbf{w}^*\right\|_2^2\right]\right) + k\alpha\sigma^2 \leq \frac{\left\|\mathbf{w}_0 - \mathbf{w}^*\right\|_2^2}{2\alpha} + k\alpha\sigma^2. \tag{2.22}$$

Finalmente, al ser la función de coste convexa:

$$C\left(\sum_{i=0}^{k-1}\frac{y_i}{k}\right) \leq \frac{\sum_{i=0}^{k-1} C(y_i)}{k} \Rightarrow k \cdot C\left(\bar{\mathbf{w}}_k\right) \leq \sum_{i=0}^{k-1} C\left(\mathbf{w}_{i+1}\right) \tag{2.23}$$

Lo que permite obtener:

$$\sum_{i=0}^{k-1}\left(\mathbb{E}\left[C\left(\mathbf{w}_{i+1}\right)\right] - C\left(\mathbf{w}^*\right)\right) = \mathbb{E}\left[C\left(\mathbf{w}_1\right) + \cdots + C\left(\mathbf{w}_k\right)\right] - kC\left(\mathbf{w}^*\right) \geq k\mathbb{E}\left[C\left(\overline{\boldsymbol{w}_k}\right)\right] - kC\left(\mathbf{w}^*\right) \tag{2.24}$$

Reentroduciendo esto en (2.22), se finaliza la prueba:

$$\mathbb{E}\left[C\left(\overline{\boldsymbol{w}_k}\right)\right] \leq C\left(\mathbf{w}^*\right) + \frac{\left\|\mathbf{w}_0 - \mathbf{w}^*\right\|_2^2}{2\alpha k} + \alpha\sigma^2 \tag{2.25}$$

■

Gracias a este resultado, podemos aliviar el peso computacional del ajuste de parámetros. Sin embargo, observamos que para obtener un error de estimación ε, el método original necesita $\mathcal{O}(\varepsilon^{-1})$ iteraciones mientras que para su versión estocástica $\mathcal{O}(\varepsilon^{-2})$.

2.1.4. Limitaciones del descenso del gradiente

Se puede probar que mediante el uso de métodos basados en el empleo del gradiente de la función de coste o estimaciones de la misma como los aquí analizados, para obtener una aproximación ε precisa, se requieren al menos $\mathcal{O}(\sqrt{L/\varepsilon})$ iteraciones [3]. Pese a que otros métodos obtienen mejores resultados, este tipo de estrategias combinan un razonable peso computacional con una buena aproximación de la solución obtenida[2]. Es por esto que gran parte de la comunidad adapta esta estrategia o las combina con otras (tal como veremos en el próximo capítulo).

Sin embargo, este tipo de métodos conllevan algunas problemáticas. La más notoria: el gradiente no se anula únicamente en el mínimo global, sino también para cualquier mínimo local o incluso punto crítico; lo que conlleva un estancamiento del aprendizaje a pesar de no alcanzar una configuración óptima.

No obstante, las demostraciones estudiadas parten de la hipótesis de convexidad sobre la función de coste. Entre otras cosas, esto garantiza la unicidad de los puntos de equilibrio para este tipo de funciones. Por tanto, en la mayoría de aplicaciones del Machine Learning se usan funciones convexas para asegurar un buen rendimiento. A pesar de ello, en ocasiones se escogen como funciones objetivos otro tipo de funciones, pues se alcanzan soluciones más deseables bajo ciertas condiciones [5]. En tal caso, el descenso del gradiente estocástico ayuda a tratar esta problemática pues gracias a la aleatoriedad de la muestra escogida para la estimación del gradiente se introduce una variabilidad en la actualización de parámetros, logrando el escape de puntos críticos.

A pesar de esto, no se asegura la convergencia a una solución óptima en el caso general. Para mejorar la eficiencia, se han desarrollado una multitud de métodos, entre los que se destacan los *rate-scheduler* y la regularización.

Rate schedulers

La idea subyaciente de los rate scheduler (programadores de tasa) consiste en la variación del valor de la tasa de aprendizaje con la evolución del entrenamiento, reduciendo su valor para obtener una mejor precisión en las etapas finales. Esto se consigue a través de una tasa dependiente de la época similar a [4]:

$$\alpha_{\text{epoch}} = \frac{\alpha_0}{1 + \tau \cdot \text{epoch}} \tag{2.26}$$

Donde α_0 es la tasa de aprendizaje inicial, τ es un hiperparámetro de ajuste y epoch es el número de épocas completadas hasta el momento (suponiendo el uso del descenso del gradiente estocástico). En posteriores capítulos se analizará Adam, un algoritmo que combina el uso de programadores con la estimación del gradiente a través de sus momentos de primer y segundo orden para obtener unos resultados experimentales excelentes.

[2]Vamos a comprobar experimentalmente más adelante con un código en Python.

Desvanecimiento del gradiente

Al usar la función sigmoide como función de activación (así como otra gran variedad de funciones), el input de entrada se traduce en la primera de las capas a un vector de escalares cuyas componentes están comprendidas entre 0 y 1, lo que es francamente conveniente para "universalizar" la información entre las neuronas. No obstante, al estudiar su derivada se observa que el intervalo en el cual es no nula o no despreciable es considerablemente pequeño, tomando además un valor máximo de 0,25 (figura 2.1). Tal como se vio en el algoritmo de Backpropagation, se calcula el gradiente de la función de coste a través de los gradientes de las funciones de activación, calculando en esencia su productorio. Por tanto, el gradiente de la función de coste en una red neuronal multicapa de gran dimensionalidad tiende a cero[3]. Es esto lo que se define como el *desvanecimiento del gradiente* [6].

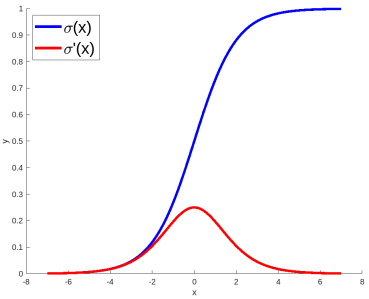


Figura 2.1: Funcion sigmoide y su derivada

Para abordar el problema, la primera solución consiste en reemplazar la función de activación empleada, por ejemplo para la función RELU (función lineal rectificada):

$$\text{RELU} : \mathbb{R} \to \mathbb{R} \tag{2.27}$$

$$x \mapsto xH(x) \tag{2.28}$$

Siendo $H(x)$ es la función escalón de Heavyside. Por tanto, su derivada es constante en el semiplano positivo del eje x con valor 1. Esta función, por contra, conlleva otras problemáticas como tener derivada nula para el semiplano negativo o la no diferenciabilidad. Como se observa que en total las funciones de activación tienen cada una unas problemáticas asociadas, se han creado soluciones más prácticas y especificas, entre ellas la regularización o el uso de LSTMs o *conexiones residuales* [6]. En resumen, estos consisten en conectar las neuronas de las primeras capas con capas posteriores para reactivar el aprendizaje de los pesos.

2.2. Overfitting y Regularización

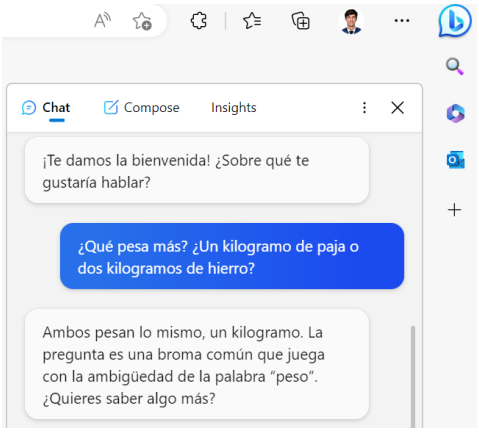


Figura 2.2: Overfitting para Chat Bing. Captura realizada el 24-03.

Cuando se habla de *overfitting* en el contexto del Machine Learning, o sobreajuste en español, se hace alusión a un entrenamiento defectuoso de la red neuronal en el que esta, en lugar de aprender de los patrones generales del conjunto de entrenamiento, aprende del "ruido" presente en sus datos, es decir, de características específicas del conjunto. Por ilustrar lo anterior, para competir con Chat GPT, a día 15 de marzo Microsoft saca al público un chatbot inteligente llamado Bing Chat. Días después de su lanzamiento, podemos encontrar respuestas absurdas a preguntas como aquella de la figura 2.2.

Esto se tiene pues, probablemente, para entrenar a la red se habrá alimentado con un conjunto de entrenamiento que posee la habitual pregunta: *¿Qué es más pesado? ¿1 kilo de paja o 1 kilo de hierro?*. La red aprendió a responder que ambos poseen la misma masa. Sin embargo, aprendió directamente la respuesta y no el sentido de la misma, por lo que si se le cambia ligeramente la pregunta, erra al generalizarla y por tanto al darle respuesta.

Un ejemplo más matemático consiste en el habitual problema de ajuste de puntos experimentales a

[3] Aunque aunque no se han tratado, este problema es más notorio y problemático en las Redes Neuronales Recurrentes. En pocas palabra, se tratan en esencia del mismo tipo de redes aquí estudiando permitiendo la creación de bucles entre las capas existentes, ampliamente usadas en el procesamiento de lenguaje natural.

un modelo desconocido. Sabemos que dados N puntos un polinomio de grado $N-1$ es capaz de ajustarlos exactamente. Sin embargo, como es el caso de la figura 2.3, lo usual es que los datos experimentales presenten un ruido externo. En este caso los puntos negros se han extraído de una variable aleatoria gaussiana cuya media es $2x$, por lo que el mejor ajuste polinómico viene dada por una tendencia lineal -línea azul. Sin embargo, si únicamente se tiene como objetivo la minimización del error de entrenamiento cometido, se puede ajustar la curva con un polinomio de grado $N-1$ -curva roja. De este modo, se ha aprendido del ruido o aleatoriedad presente, por lo que se errará en la generalización y predicción de nuevos puntos.

Distinguimos dos tipos de errores: el de entrenamiento o aprendizaje y de generalización o error "real". Pese a que, en vistas de la consecución de un error de generalización lo menor posibleñd la estrategia general se basa en la obtención de un error mínimo de aprendizaje, no es conveniente que este último sea nulo o excesivamente pequeño, o se correrá el riesgo de aprender directamente del ruido experimental. Es en este contexto donde entra en juego la regularización.

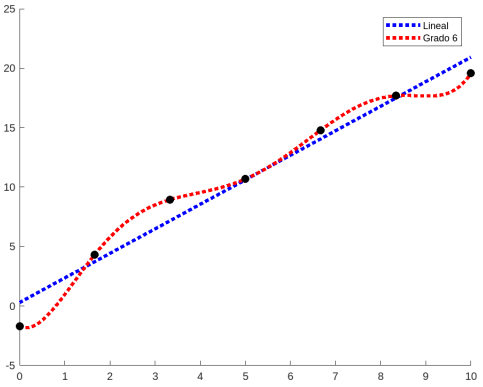


Figura 2.3: Overfitting para el ajuste de puntos

Se define la regularización como "cualquier modificación que se hace sobre un algoritmo de aprendizaje con la intención de reducir su error de generalización y no su error de entrenamiento". Existen varias estrategias, pero sin duda la más frecuente consiste en la adición de un término a la función de coste con el objetivo de penalizar ciertos comportamientos patológicos. De entre ellos, es usual limitar el crecimiento excesivo de los pesos debido a que cuanto mayores sean estos, más pronunciada será la función resultante de la composición de capas en el proceso de feedforward. Esta característica se ilustra en la curva roja mostrada en la Figura 2.3. En otras palabras, esta estrategia busca evitar desviaciones significativas en los resultados obtenidos con inputs similares.

Formalizando esto, si la función de coste original viene dada por $C(\mathbf{x}, \mathbf{y}; \mathbf{w}, \mathbf{b})$, para introducir la regularización, esta es reemplazada por:

$$\tilde{C}(\mathbf{x}, \mathbf{y}; \mathbf{w}, \mathbf{b}) = C(\mathbf{x}, \mathbf{y}; \mathbf{w}, \mathbf{b}) + \eta\Omega(\mathbf{w}) \qquad \eta \in [0, \infty) \tag{2.29}$$

Los sesgos no dependen directamente de las entradas de la red y por lo tanto no están sujetos a las mismas preocupaciones de crecimiento excesivo que los pesos de las conexiones. Si se penalizan los sesgos, se podría restringir innecesariamente la flexibilidad del modelo y limitar su capacidad para ajustarse a los datos de entrenamiento. Esto podría conducir a un subajuste (underfitting), donde el modelo no puede capturar suficientemente las relaciones y complejidades presentes en los datos [7].

Para comprender mejor las consecuencias de la introducción de este término regulatorio, vamos a explorar un ejemplo con profundidad: la conocida por regularización L^2 (probablemente la mas frecuente[4]), consistente en tomar $\Omega(\mathbf{w}) = 1/2||\mathbf{w}||_2^2$, con lo que la función de pérdida se transforma en:

$$\tilde{C}(\mathbf{x}, \mathbf{y}; \mathbf{w}, \mathbf{b}) = C(\mathbf{x}, \mathbf{y}; \mathbf{w}, \mathbf{b}) + \frac{\eta}{2}\mathbf{w}^T\mathbf{w} \tag{2.30}$$

Vamos a ver cómo afecta a nuestro algoritmo de aprendizaje (descenso del gradiente estocástico) respecto del método original. El gradiente de la nueva función de coste es modificado por:

$$\nabla_{\mathbf{w}}\tilde{C} = \eta\mathbf{w} + \nabla_{\mathbf{w}}C \tag{2.31}$$

Con ello, la actualización de pesos en cada una de las épocas toma la forma:

[4]Otra de las más comunes formas de penalización se hace a través de la elección de la norma en L^1

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \alpha\left(\eta\mathbf{w}_k + \nabla C\right) = (\mathbf{w}_k - \alpha\nabla C) - \eta\alpha\mathbf{w}_k \tag{2.32}$$

Se comprueba que el resultado obtenido es la limitación del crecimiento de los pesos en base a un término constante proporcional a ellos ($-\alpha\eta\mathbf{w}_k$). Adicionalmente, se aborda el posible problema de desvanecimiento del gradiente, ya que el gradiente ahora incorpora este término constante que no depende de las funciones de activación como si lo hace el término del gradiente de la función de coste.

Pasemos a estudiar el efecto de la regularización L^2 a nivel global, estudiando el efecto acumulado a lo largo del conjunto total de épocas. Para ello, dada una función de coste, previo a la introducción del término regulatorio sobre C, se puede tomar su aproximación cuadrática en torno al mínimo, denotado por $w^* = \arg\min_{\mathbf{w}} C(\mathbf{w})$ (en el cual el gradiente es nulo):

$$C(\mathbf{w}) = C(\mathbf{w}^*) + \frac{1}{2}(\mathbf{w} - \mathbf{w}^*)^T H(\mathbf{w} - \mathbf{w}^*) \tag{2.33}$$

Donde H es la Hessiana de C evaluada en $\mathbf{w}^*$, que es semidefinida positiva por tratarse $\mathbf{w}^*$ de un mínimo. El mínimo es alcanzado cuando el gradiente es nulo, o lo que equivale en términos de la matriz la hessiana a:

$$\nabla_{\mathbf{w}} C(\mathbf{w}) = H(\mathbf{w} - \mathbf{w}^*) = 0 \tag{2.34}$$

Estudiemos ahora el mínimo de la anterior función de coste aproximada C pero esta vez incluyendo la regularización. Si en esta ocasión el mínimo viene dado por $\tilde{\boldsymbol{w}}$, análogamente debe de cumplir la condición siguiente en este punto óptimo para $\widehat{C}$:

$$\nabla_{\mathbf{w}} \widehat{C}(\mathbf{w}) = \eta\tilde{\boldsymbol{w}} + H(\tilde{\boldsymbol{w}} - \mathbf{w}^*) = 0 \rightarrow \tilde{\boldsymbol{w}} = (H + \eta\mathbb{I})^{-1} H\mathbf{w}^* \tag{2.35}$$

Si se toma $\eta \rightarrow 0$, entonces $\tilde{\boldsymbol{w}} \rightarrow \mathbf{w}^*$ como es esperable. En caso contrario, para estudiar el efecto neto de la regularización L^2, se puede tener en cuenta que H es real y simétrica, lo que la convierte en diagonalizable. De este manera, existen dos matrices Λ y P tales que $H = P\Lambda P^T$. Retomando (2.35) con esta información:

$$\tilde{\boldsymbol{w}} = \left[P(\Lambda + \eta\mathbb{I})P^T\right]^{-1} P\Lambda P^T\mathbf{w}^* = P(\Lambda + \eta\mathbb{I})^{-1}\Lambda P^T\mathbf{w}^* \tag{2.36}$$

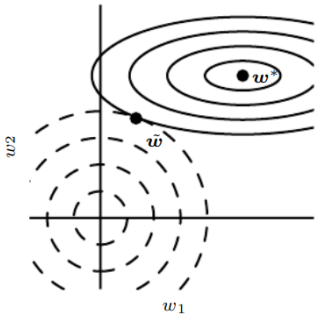


Figura 2.4: I. Goodfellow (2015). Efecto de la Regularización-L^2.

Así, el efecto total es el de reescalar la solución proporcionada por el descenso del gradiente a lo largo de los ejes definidos por los autovectores de H. En concreto, si λ_i son los autovalores que forman Λ, la componente de la anterior solución óptima $\mathbf{w}^*$ alineada con el i-ésimo autovector de H se ve reescalada por un factor de $(\lambda_i + \eta)^{-1}\lambda_i$ según la anterior igualdad. De esta manera, aquellas direcciones para las cuales el autovalor correspondiente sea pequeño, las componente en dicho eje se verán encogidas fuertemente por η; mientras que aquellas direcciones cuyo autovalor sea relativamente grande apenas sufrirán modificaciones.

El anterior fenómeno se puede ver gráficamente en la figura 2.4, donde se representan las curvas de nivel de C con línea discontinua y con línea continua lo equivalente para $\tilde{C}$. En el punto $\mathbf{w}^*$ ambas funciones se encuentran en equilibrio. Dado que en el eje w_1 el autovalor tiene un valor pequeño, la función de coste no aumenta excesivamente cuando uno se aleja de $\tilde{\boldsymbol{w}}$, siendo el resultado de la introducción del término regularizador un gran impacto sobre esta dirección. Así la componente correspondiente correspondiente de $\tilde{\boldsymbol{w}}$ se acerca a 0. Por contra, para la componente restante w_2, el autovalor es grande indicando una gran curvatura; lo que hace que la regularización no tenga un importante papel en esta dirección.

Como conclusión, se observa que solo aquellas direcciones en las que los parámetros tiene una contribución significativa para la reducción de la función de coste se mantienen intactos, como es esperable. Por contra, en las direcciones para las cuales la contribución sea débil, un cambio en dicha dirección no modificará notablemente el gradiente, lo que hace que el término regularizador tienda a anularse.

2.3. Redes Bayesianas

La teoría de probabilidad bayesiana es especialmente interesante para la aplicación de las redes neuronales. En este contexto, se propone el estudio de inferencias sobre los pesos de las conexiones neuronales en lugar de tomarlos como escalares fijos -como hasta ahora se ha hecho en las *redes deterministas*. Así, los pesos $w_{ij}^{(l)}$ son tratados como variables aleatorias.

Esto favorece el rendimiento de la red por dos motivos principalmente. Por una parte, se aborda el problema del overfitting debido a que no solo se trata de minimizar la función de pérdida en los datos de entrenamiento, sino también de maximizar la evidencia de los datos experimentales dada la distribución de los pesos. Esto permite que la red penalice soluciones excesivamente complejas. Por otro lado, se puede realizar un mejor seguimiento de la información en su paso por la red al estudiar probabilidades; contrariamente a las redes deterministas para las cuales dado un input es complicado predecir su resultado. Esto último además permite afrontar los posibles problemas éticos que surgen de los resultados devueltos por la red, pudiendo evitar casos de discriminación o respuestas no deseadas [8].

2.3.1. Ajuste de pesos

Como se ha comentado, se parte de la hipótesis en la que los pesos $w_{ij}^{(l)}$ son variables aleatorias. A partir de esto, vamos a estudiar las probabilidades de hacer una predicción u otra en base a sus valores. En concreto, la nueva estrategia para el ajuste de pesos consiste en escoger aquella configuración que sea más probable conocido el conjunto de entrenamiento D: $\mathrm{P}(\mathbf{w}|D)$. A esta distribución de probabilidad se le denomina *distribución posterior*. Denotamos a su valor máximo como $\mathbf{w}^*$, incluyendo aquí de nuevo los sesgos.

Para ello, se asume conocida una distribución inicial de probabilidad: $\pi(\mathbf{w})$. En virtud del Teorema de Bayes y el Teorema de la Probabilidad Total:

$$\mathrm{P}(\mathbf{w}|D) \overset{\text{Bayes}}{=} \frac{\mathrm{P}(D|\mathbf{w}) \cdot \pi(\mathbf{w})}{\mathrm{P}(D)} \overset{\text{P. total}}{=} \frac{\mathrm{P}(D|\mathbf{w}) \cdot \pi(\mathbf{w})}{\int \mathrm{P}(D|\mathbf{w})\pi\left(\mathbf{w}\right) \mathrm{d}\mathbf{w}} \tag{2.37}$$

Como en tantas ramas de las matemática, debido al gran conocimiento de expresiones concretas, supondremos que las distribución inicial y posterior son ambas normales. En concreto, la distribución inicial, a falta de información a priori, tendrá una media de $\mathbf{0}$ y una varianza dada por $\alpha^{-1}\mathbf{I}$, siendo $\alpha \in \mathbb{R}$ un hiperparámetro. De este modo: $\pi(\mathbf{w}) = \mathcal{N}(\mathbf{w}|\mathbf{0}, \alpha^{-1}\mathbf{I})$.

Análogamente, la media de la función de verosimilitud $\mathrm{P}(D|\mathbf{w})$ será igual al valor que predice una red neuronal determinista con su misma configuración, denotando dicho valor por $\tilde{y}(\mathbf{x})$ [5]. Su varianza, por conveniencia, la estudiaremos como la inversa de un hiperparámetro denominado *precisión*: β. Es decir: $\mathrm{P}(D|\mathbf{y}) = \mathcal{N}(\mathbf{w}|\tilde{y}(\mathbf{x}, \mathbf{w}), \beta^{-1})$.

Por otro lado, se suponen los datos del conjunto de entrenamiento independientes e idénticamente distribuidos. Si $D = \{(x_i, y_i) : \ i = 1, \ldots, N\}$, esto permite tratar cada punto independientemente: $\mathrm{P}(D|\mathbf{w}) = \prod_i \mathcal{N}(y_i|\tilde{y}(x_i, \mathbf{w}), \beta^{-1})$.

Con estas hipótesis, las distribuciones iniciales y la función de verosimilitud son:

[5] Se puede calcular dicho valor gracias al feedforward

$$\pi(\mathbf{w}) = \frac{1}{Z_{\mathbf{w}}(\alpha)} \exp\left(-\frac{\alpha}{2}\sum_{i=1}^{W} w_i^2\right) \quad \text{y} \quad \mathrm{P}\left(D|\mathbf{w}\right) = \frac{1}{Z_D(\beta)} \exp\left(-\frac{\beta}{2}\sum_{n=1}^{N}\left\{\tilde{y}(x_i,\mathbf{w}) - y_n\right\}^2\right) \tag{2.38}$$

Donde $Z_{\mathbf{w}}$ y Z_D son normalizadores, cuyas expresiones concretas son:

$$Z_{\mathbf{w}}(\alpha) = \int \exp\left(-\frac{\alpha}{2}\sum_{i=1}^{W} w_i^2\right) \mathrm{d}\mathbf{w} = \left(\frac{2\pi}{\alpha}\right)^{W/2} \tag{2.39}$$

$$Z_D(\beta) = \int \exp\left(-\frac{\beta}{2}\sum_{n=1}^{N}\left\{\tilde{y}(\mathbf{x},\mathbf{w}) - y_n\right\}^2\right) \mathrm{d}y_1 ... \mathrm{d}y_1 = \left(\frac{2\pi}{\beta}\right)^{N/2} \tag{2.40}$$

Y metiendo estas dos expresiones en (2.37), tenemos que la distribución posterior es:

$$\mathrm{P}(\mathbf{w}|D) = \frac{1}{Z_S} \exp\left(-\frac{\alpha}{2}\sum_{i=1}^{W} w_i^2\right) \exp\left(-\frac{\beta}{2}\sum_{n=1}^{N}\left\{\tilde{y}(\mathbf{x},\mathbf{w}) - y_n\right\}^2\right) \tag{2.41}$$

$$Z_S(\alpha,\beta) = \int \exp\left(-\frac{\alpha}{2}\sum_{i=1}^{W} w_i^2 - \frac{\beta}{2}\sum_{n=1}^{N}\left\{\tilde{y}(\mathbf{x},\mathbf{w}) - y_n\right\}^2\right) \mathrm{d}\mathbf{w} \tag{2.42}$$

Debido a la gran magnitud del conjunto de entrenamiento, la ecuación (2.37) es prácticamente imposible de emplear. Por tanto, vamos a aplicar una aproximación de Laplace sobre la distribución posterior [9]. Realizando las manipulaciones algebraicas convenientes, podemos expresar [10]:

$$\ln \mathrm{P}(\mathbf{w}|D) = -\frac{\alpha}{2}\mathbf{w}^T\mathbf{w} - \frac{\beta}{2}\sum_{n=1}^{N}\left\{\tilde{y}(x_n,\mathbf{w}) - y_n\right\}^2 + cte \tag{2.43}$$

Se observa la gran similaridad con la ecuación de error cuadrática que se introdujo en las redes neuronales deterministas (1.5). Gracias a esta expresión, podemos realizar el problema de maximización, obteniendo una matriz de pesos $\mathbf{w}_{\mathrm{MAP}}$ (MAP significa Máximo A Posteriori). Dicha maximización se puede realizar gracias a técnicas basadas en el uso del gradiente conjunto a Backpropagation.

Para finalizar, una vez la red ha sido entrenada, la obtención de nuevas predicciones $\hat{y}$ se realiza con una marginalizando respecto de $\mathbf{w}$.

$$\mathrm{P}(\hat{y}|\mathbf{x},D) = \int \mathrm{P}(\hat{y}|\mathbf{x},\mathbf{w})\mathrm{P}(\mathbf{w}|D)\mathrm{d}\mathbf{w} \simeq \mathrm{P}(\hat{y}|\mathbf{x},\mathbf{w}_{\mathrm{MAP}}) \tag{2.44}$$

La anterior aproximación es valida en aquellos casos donde la distribución posterior $\mathrm{P}(\mathbf{w}|D)$ está fuertemente concentrada respecto de su máximo $\mathbf{w}_{\mathrm{MAP}}$. Nótese que entonces se obtienen ecuaciones idénticas a aquellas vistas en las redes deterministas.

2.3.2. Elección de hiperparámetros

Para el estudio de la elección de hiperparámetros óptima, se propone el uso de la inferencia. Por tanto, se supone que α y β son variables aleatorias. De este modo, para el cálculo de probabilidades anteriormente descrito, se debe de marginalizar respecto de dichas variables para eliminar la dependencia en las anteriores expresiones. En concreto, la distribución posterior se calcula como:

$$\mathrm{P}(\mathbf{w}|D) = \int\int \mathrm{P}(\mathbf{w},\alpha,\beta|D)\mathrm{d}\alpha\mathrm{d}\beta = \int\int \mathrm{P}(\mathbf{w}|\alpha,\beta,D)\mathrm{P}(\alpha,\beta|D)\mathrm{d}\alpha\mathrm{d}\beta \tag{2.45}$$

Esta integración es intratable de manera analítica, así que nuevamente se toman aproximaciones, siendo habitual el uso de una nueva aproximación de Laplace. Este enfoque es conocido por *empirical Bayes* en el campo de la Estadística; o *evidence approximation* en el contexto del Machine Learning. Si la distribución de probabilidad se concentra especialmente en los máximos α_{MAP} y β_{MAP}, podemos hacer una aproximación análoga a aquella realizada en (2.44):

$$\mathrm{P}(\mathbf{w}|D) \simeq \mathrm{P}(\mathbf{w}|\alpha_{\text{MAP}}, \beta_{\text{MAP}}, D) \tag{2.46}$$

Veamos qué valores de hiperparámetros maximizan la probabilidad posterior. Gracias al Teorema de Bayes y de la Probabilidad Total, obtenemos una expresión equivalente a (2.37):

$$\mathrm{P}(\alpha, \beta|D) = \frac{\mathrm{P}(D|\alpha, \beta)\mathrm{P}(\alpha, \beta)}{\mathrm{P}(D)} \tag{2.47}$$

Donde se ha supuesto conocida una distribución a priori para los hiperparámetros $\mathrm{P}(\alpha, \beta)$. Por tanto, escogeremos los hiperparámetros que maximicen esta última distribución. Se observa que el denominador y la distribución inicial no juegan ningún papel, así que podemos directamente maximizar $\mathrm{P}(D|\alpha, \beta)$, es decir, la función de verosimilitud. Con un poco de manipulación y empleando el teorema de Bayes [11]:

$$\mathrm{P}(D; \alpha, \beta) = \int \mathrm{P}(D|\mathbf{w}, \beta)\pi(\mathbf{w}, \alpha)\mathrm{d}\mathbf{w} \tag{2.48}$$

Para su obtención se ha supuesto la normalidad de las dos distribuciones presentes. De este modo, mediante un desarrollo totalmente análogo al realizado para la obtención de la expresión (2.41), se obtiene:

$$\mathrm{P}(D; \alpha, \beta) = \frac{1}{Z_D(\beta)Z_{\mathbf{w}}(\alpha)} \int \exp\left(\frac{\alpha}{2}\mathbf{w}^T\mathbf{w} - \frac{\beta}{2}\sum_{n=1}^{N}\left\{\tilde{y}(x_n, \mathbf{w}) - y_n\right\}^2\right)\mathrm{d}\mathbf{w} \tag{2.49}$$

Sobre esta expresión se hace nuevamente una aproximación de Laplace. Si $\mathbf{A} = -\nabla\nabla \ln \mathrm{P}(\mathbf{w}|D, \alpha, \beta) = \alpha\mathbb{I} + \beta\mathbf{H}$ es una aproximación local gaussiana de la distribción estudiada de y W el número de configuraciones posibles para $\mathbf{w}$:

$$\ln\left(\mathrm{P}(D; \alpha, \beta)\right) = -\left[\frac{\alpha}{2}\mathbf{w}_{\text{MAP}}^T\mathbf{w}_{\text{MAP}} + \frac{\beta}{2}\sum_{n=1}^{N}\left\{\tilde{y}(x_n, \mathbf{w}_{\text{MAP}}) - y_n\right\}^2\right] - \frac{1}{2}\ln|\mathbf{A}| + \frac{W}{2}\ln\alpha + \frac{N}{2}\ln\beta - \frac{N}{2}\ln 2\pi \tag{2.50}$$

Obsérvese que el término entre corchetes es en esencia (2.43). Derivando respecto de cada parámetros para obtener la elección óptima en cada caso, se obtienen las elecciones óptimas para cada hiperparámetro:

$$\alpha = \frac{\gamma}{\mathbf{w}_{\text{MAP}}^T \cdot \mathbf{w}_{\text{MAP}}} \qquad \beta^{-1} = \frac{\sum_{n=1}^{N}\left\{\tilde{y}(\mathbf{x}_n, \mathbf{w}_{\text{MAP}}) - y_n\right\}^2}{N - \gamma} \tag{2.51}$$

Donde $\gamma = \sum_{i=1}^{W} \lambda_i/(\alpha + \lambda_i)$ es el número efectivo de parámetros, siendo λ_i las soluciones del problema de autovalores de la matriz hessiana $\beta\mathbf{H}\mathbf{u}_i = \lambda_i\mathbf{u}_i$.

Capítulo 3

Adam y Adagrad

En los anteriores capítulos se han estudiado las primeras y más básicas matemáticas desarrolladas para la creación y entrenamiento de las redes neuronales. Son técnicas ilustrativas y otorgan una buena visión de la estrategia ideada para reproducir el procesamiento de la información de una manera similar a la realizada por los seres vivos. No obstante, para la verdadera aplicación de las redes neuronales, se han diseñado técnicas más complejas y elaboradas. Con este espíritu, la presente sección se dedica al estudio de una de estas técnicas: el optimizador Adam (y Adagrad visto como un caso particular).

El algoritmo de optimización empleado determina en gran medida la velocidad de entrenamiento y el rendimiento predictivo final del modelo de la red neuronal. Hasta la fecha, no existe una teoría que guíe adecuadamente dicha elección. En su lugar, la comunidad se basa en estudios empíricos y la evaluación comparativa. Estos estudios arrojan resultados comúnmente favorables para el algoritmo aquí estudiado en situaciones más o menos generales.

Adam (*Adaptive Moment Estimation*) fue propuesto por Diederik P. Kingma y Jimmy Ba en 2014 en el artículo "Adam: A Method for Stochastic Optimization" [12]. Los autores desarrollaron este optimizador combinando características propias de Adagrad y de RMSprop [13]. De esta manera, Adam utiliza estimaciones de los momentos de primer y segundo orden del gradiente de la función de coste para variar la tasa de aprendizaje de manera adaptativa en el entrenamiento de la red neuronal.

Del mismo modo, en este paper además se presenta una prueba de la convergencia del método, que resultó contener errores y fue posteriormente corregida por S. Bock et al [14]. Por último, el año pasado desde Meta AI se generalizó este optimizador y su prueba de convergencia haciendo especial énfasis en el efecto del momento de primer orden [15], obteniendo consigo un mejor orden de convergencia. Es precisamente esta prueba la que aquí se analiza por ser más general y por sus mejores resultados. Asimismo, como ventaja de esta versión, mediante una elección particular de los hiperparámetros se puede recuperar el optimizador Adagrad, lo que puede resultar interesante en ciertos escenarios y en vistas al estudio teórico de los mismos.

3.1. Presentación de los métodos

Adagrad

Adagrad fue propuesto por Duchi, Hazan y Singer en 2011 [16], realizando esencialmente una sofisticación del método del descenso del gradiente mediante la introducción de un rate-scheduler. En particular se sustituye al tasa de entrenamiento del descenso del gradiente α por:

$$\alpha_t = \frac{\alpha_0}{\sqrt{\epsilon + (\sum_{\tau_N=1}^{t} g_{\tau_N}^2)}} \tag{3.1}$$

Algoritmo 3.1: ADAM. Parámetros típicos: $\alpha = 10^3$, $\beta_1 = 0,9$, $\beta_2 = 0,999$ y $\epsilon = 10^{-8}$. Todas las operaciones son a nivel de componentes, y $g_t^2 := g_t \odot g_t$

Input:
Tamaño de paso: α
Tasas exponenciales de actualización de momentos: $0 \leq \beta_1, \beta_2 \leq 1$
Pesos: $\mathbf{w}_0$

Output:
Estimación de los pesos: $\mathbf{w}_N$

Inicialización:
Momento 1$^{\text{er}}$ momento: $m_0 \leftarrow 0$
Momento 2$^{\text{o}}$ momento: $v_0 \leftarrow 0$
Tiempo: $t \leftarrow 0$

while $\boldsymbol{w_t}$ *no converge* **do**

$$
\begin{array}{lcll}
t+1 & \leftarrow & t & \\
g_t & \leftarrow & \nabla_{\mathbf{w}} f_t(\mathbf{w}_{t-1}) & \\
m_t & \leftarrow & \beta_1 m_{t-1} + (1-\beta_1) g_t & (*) \\
\hat{m}_t & \leftarrow & m_t / (1-\beta_1^t) & \\
v_t & \leftarrow & \beta_2 v_{t-1} + (1-\beta_1) g_t^2 & (**) \\
\hat{v}_t & \leftarrow & v_t / (1-\beta_2^t) & \\
\mathbf{w}_t & \leftarrow & \mathbf{w}_{t-1} - \alpha \frac{\bar{m}_t}{\sqrt{\bar{v}_t} + \epsilon} & (***)
\end{array}
$$

end
return $\boldsymbol{w_t}$

Nota: Este algoritmo es compatible con el uso de subconjuntos tomados aleatoriamente de entre D, como se hizo en el Descenso del Gradiente Estocástico.

En tal caso, el algoritmo obtiene un gran rendimiento pues la adaptación de la tasa de aprendizaje se hace de tal manera que los parámetros que tienen gradientes dispersos/actualizaciones más infrecuentes reciben una tasa de aprendizaje superior, lo que permite que se actualicen más rápidamente. Del mismo modo, los gradientes con una mayor frecuencia de actualización (i.e. cuya suma de gradientes es alta) reciben una tasa de aprendizaje inferior para evitar oscilaciones o cambios demasiado rápidos.

El hiperparámetro ϵ es introducido para evitar una división entre 0. De este modo, para no introducir un error externo, a priori es conveniente fijarlo a una cantidad insignificante[1], e.g. 10^{-8}.

Adam

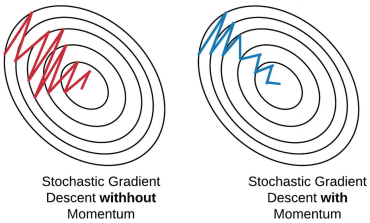


Figura 3.1: H. Ghorbel (2021). Efecto de los momentos sobre el descenso del gradiente

Para estudiar Adam con detalle, y sobre todo con vistas a la demostración, es conveniente aumentar el grado de abstracción y por un momento olvidarse de nuestro estudio sobre redes neuronales. En su lugar pasaremos a estudiar un problema de optimización general. De este modo, el objetivo actual consiste en la minimización de una función genérica $F : \mathbb{R}^d \to \mathbb{R}$. Para ello, se dispone de una función estocástica $f : \mathbb{R}^d \to \mathbb{R}$ tal que $\mathbb{E}[f(\mathbf{w})] = F(\mathbf{w})$ que es muestreada a cada paso de tiempo t para obtener una orientación acerca del error cometido en la presente iteración de entrenamiento f_t [2]. Con esta notación, se define el algoritmo Adam en el pseudoalgoritmo 3.1.

Como se aprecia, está basado en el método del descenso del gradiente: si se toma $\beta_1 = \beta_2 = 0$ y $\epsilon = 1$ se obtiene el mismo optimizador. Sin embargo, ahora se tienen en cuenta los *momentos* de primer y segundo

[1] No obstante, como veremos más adelante, en ciertas ocasiones conviene aumentar este hiperparámetro.
[2] Con esta función f se puede modelizar la aleatoriedad presente en el DGE.

Algoritmo 3.2: GENERALIZACIÓN ADAM. Parámetros típicos: $\alpha = 10^3$, $\beta_1 = 0,9$, $\beta_2 = 0,999$ y $\epsilon = 10^{-8}$. Todas las operaciones son a nivel de componentes, y $g_t^2 := g_t \odot g_t$

Input:
Tamaño de paso: α
Tasas exponenciales de actualización de momentos: $0 \leq \beta_1 < \beta_2 \leq 1$
Output:
Estimación de los pesos: $\mathbf{w}_N$
Inicialización:
Momento 1$^{\text{er}}$ momento: $m_0 \leftarrow 0$
Momento 2$^{\text{o}}$ momento: $v_0 \leftarrow 0$
Tiempo: $t \leftarrow 0$
Pesos: $\mathbf{w}_0$

while $\boldsymbol{w_t}$ *no converge* **do**

$$
\begin{array}{lcll}
t+1 & \leftarrow & t & \\
g_t & \leftarrow & \nabla_{\mathbf{w}} f_t(\mathbf{w}_{t-1}) & \\
\alpha_t & \leftarrow & \alpha(1-\beta_1)\sqrt{\frac{1-\beta_2^t}{1-\beta_2}} & \\
m_t & \leftarrow & \beta_1 m_{t-1} + g_t & (*).2 \\
v_t & \leftarrow & \beta_2 v_{t-1} + g_t^2 & (**).2 \\
\mathbf{w}_t & \leftarrow & \mathbf{w}_{t-1} - \alpha_t \frac{m_t}{\sqrt{v_t+\epsilon}} & (***).2
\end{array}
$$

end
return $\boldsymbol{w_t}$
Nota: Este algoritmo es compatible con el uso de subconjuntos tomados aleatoriamente de entre D, como se hizo en el Descenso del Gradiente Estocástico.

orden m_t y v_t del gradiente (basado en el concepto físico, pues de cierta manera, hace que el gradiente posea "inercia" teniendo en cuenta el historial de sus valores, como se ve en la figura 3.1). Estos momentos se actualizan exponencialmente, controlando su actualización a través de los hiperparámetros β_1 y β_2. Esta característica ayuda a mantener una dinámica de actualización suavizada y a estabilizar la optimización.

Generalización de Adam

El algoritmo Adam es adaptado por Meta AI [15] tomando la forma del pseudoalgoritmo 3.2. Como se ha comentado, permite relacionar sencillamente Adam y Adagrad. En concreto, con esta redefinición, Adagrad resulta ser un caso particular de Adam con la elección de $\beta_1 = 0$ y $\beta_2 = 1$.

Además, se consigue una significante mejoría del orden de convergencia. Cabe mencionar que la expresión de la tasa de aprendizaje adaptativa α_t ahora empleada no es exactamente idéntica a la originalmente propuesta. Realmente, si se quisiera tener en cuenta todos los términos correctivos que Adam incluye, este hiperparámetro tomaría la forma:

$$
\alpha_{t,\text{ adam}} = \alpha \cdot \frac{1-\beta_1}{\sqrt{1-\beta_2}} \cdot \underbrace{\frac{1}{1-\beta_1^t}}_{\text{corrección } m_t} \cdot \underbrace{\sqrt{1-\beta_2^t}}_{\text{corrección } v_t} . \tag{3.2}
$$

No obstante, se decide desechar el término correctivo para m_n pues simplifica la prueba. Por otra parte, esto no implica un error excesivo pues dicho término converge a su valor límite más rápidamente que aquel de v_n, debido al valor que suelen tomar en la aplicación y de su dependencia con el número de iteraciones, pudiendo ser despreciable desde fases tempranas del entrenamiento.

3.2. Corrección de inicialización de momentos

A falta de información previa, los momentos se inicializan ambos a 0 tal como se indica en el pseudoalgoritmo 3.1. No obstante, esto sesgaría el proceso de optimización en torno a dicho valor. Para solventarlo, se debe de hacer una corrección (realizada mediante las variables $\bar{m}_t$ y $\bar{v}_t$) comparando el valor actual del gradiente con la esperanza de los mismos a cada paso de tiempo. Veamos la forma particular de esta relación para el momento de primer momento del gradiente g_t. Directamente de la regla de actualización de m_t (*), se tiene recursivamente:

$$m_t = (1-\beta_1)\left(\sum_{j=1}^{t}\beta_1^t g_{t-j} + m_0\right) \tag{3.3}$$

Tomando esperanzas a ambos lados y teniendo en cuenta que $m_0 = 0$:

$$\mathbb{E}[m_t] = \sum_{j=1}^{t}\mathbb{E}\left[(1-\beta_1)\beta_1^j g_{t-j}\right] \overset{(\triangle)}{=} \mathbb{E}[g_t]\sum_{j=1}^{t}(1-\beta_1)\beta_1^j + \zeta = \beta_1(1-\beta_1^t)\cdot\mathbb{E}[g_t] + \zeta \tag{3.4}$$

En la igualdad ($\triangle$) se toma como factor común la esperanza del gradiente de la suma con el propósito de relacionar ambos extremos de la ecuación fácilmente. Por contra, es necesario entonces introducir un término correctivo ζ donde se englobe el error de la anterior aproximación. Si el gradiente fuera estacionario, claramente $\zeta = 0$. En caso contrario, este error debe de mantenerse pequeño escogiendo β_1 de modo de que la influencia de gradientes lejanos en el tiempo no condicione a m_t excesivamente. Por tanto, suponiendo que ζ puede ser despreciado, despejando de la anterior ecuación se obtiene precisamente la corrección $\hat{m}_t$. Del mismo modo, el término correctivo para v_t es obtenido análogamente (y por ello tienen la misma forma, intercambiando β_1 por β_2 sencillamente).

3.3. Sobre la convergencia de Adagrad y Adam

Para poder enunciar los resultados, tenemos que hacer unas definiciones, hipótesis y comentarios primeramente.

- Continuando con la abstracción comenzada, olvidemos totalmente que se trata de un problema de redes neuronales. Para ello, en vez de pensar en pesos de las conexiones neuronales $\mathbf{w}$, pensemos en variables más generales a ajustar, denotándolas por $x \in \mathbb{R}^d$ (siendo d el número de dimensiones del problema). Recordemos que Adam es un método ideado para optimización, no siendo exclusiva su aplicación a redes neuronales.
- Definimos la esperanza condicionado a f_1, ..., f_n como $\mathbb{E}_n[\cdot]$, siendo n el número de iteraciones. Es importante remarcar que cualquier esperanza condicionada de cualquier función únicamente dependiente de F coincide con su valor (i.e. $\mathbb{E}_n[F] = F$ para cualquier n). Del mismo modo se define

  $$\tilde{v}_{n,i} := \beta_2 v_{n-1,i} + \mathbb{E}_{n-1}\left[(\nabla_i f_n(x_{n-1}))^2\right] \tag{3.5}$$

  Es decir, se trata en esencia del momento de segundo orden sustituyendo el valor de la última iteración por su valor esperado. Por esto mismo $\mathbb{E}_n[\tilde{v}_n] = \tilde{v}_n$.
- Dado un número de iteraciones $N \in \mathbb{N}^*$, se define τ_N una variable aleatoria discreta con dominio en $\{0, ..., N-1\}$ con la siguiente distribución:

  $$\mathbb{P}[\tau_N = j] \propto 1 - \beta_1^{N-j} \qquad \forall j \in \mathbb{N}, j < \mathbb{N} \tag{3.6}$$

Pese que a priori puede parecer una definición extraña, se verá su conveniencia en las demostraciones de los teoremas 3.1 y 3.1.

- Se supondrá que F está acotada superiormente por F_*:

$$F(x) \geq F_* \qquad \forall x \in \mathbb{R}^d \tag{3.7}$$

- Se supone además que existe un $R > \sqrt{\epsilon}$ tal que:

$$||\nabla f(x)||_\infty \overset{\text{c.s.}}{\leq} R - \sqrt{\epsilon} \qquad \forall x \in \mathbb{R}^d \tag{3.8}$$

Esta misma propiedad implica:

$$\sqrt{\epsilon + \tilde{v}_{n,i}} \leq \sqrt{R^2 \sum_{l=1}^{n} \beta_2^{n-l}} = R\sqrt{\sum_{l=1}^{n} \beta_2^{n-l}} \overset{\beta \leq 1}{\leq} R\sqrt{n} \tag{3.9}$$

Esto justifica precisamente la cota sobre el gradiente a primera vista tan poco natural. Nuevamente, la motivación de esta desigualdad será expuesta en las demostraciones de los resultados de convergencia.

- Para terminar, supongamos que el gradiente de F es una función L-lipschitziana en la norma L_2:

$$||\nabla F(x) - \nabla F(y)||_2 \leq L||x - y||_2 \qquad x, y \in \mathbb{R}^d \tag{3.10}$$

En concreto, esto implica gracias al lema A.1 la siguiente igualdad:

$$F(x) \leq F(y) - \nabla F(x)^T (x - y) + \frac{||x - y||_2^2 L}{2} \tag{3.11}$$

3.3.1. Teoremas de convergencia

Teorema 3.1 (Convergencia de Adagrad con momento de primer orden)**.** *Con las anteriores hipótesis, usando las reglas de actualización del Algoritmo 3.2 y con la elección de hiperparámetros $\beta_1 \in [0, 1)$, $\beta_2 = 1$ y $\alpha > 0$, para cualquier número de iteraciones $N \in \mathbb{N}^*$ tal que $N > \beta_1/(1 - \beta_1)$, se tiene la siguiente cota:*

$$\mathbb{E}\left[\|\nabla F\left(x_{\tau_N}\right)\|_2^2\right] \leq 2R\sqrt{N}\frac{F\left(x_0\right) - F_*}{\alpha\tilde{N}} + \frac{\sqrt{N}}{\tilde{N}} E \ln\left(1 + \frac{NR^2}{\epsilon}\right) \tag{3.12}$$

donde $\tilde{N} := N - \frac{\beta_1}{1-\beta_1}$ y $E = \alpha d R L + \frac{12dR^2}{1-\beta_1} + \frac{2\alpha^2 dL^2\beta_1}{1-\beta_1}$

Teorema 3.2 (Convergencia de Adam)**.** *Con las anteriores hipótesis, usando las reglas de actualización del Algoritmo 3.2 y con la elección de hiperparámetros $\beta_1 \in [0, \beta_2)$, $\beta_2 \in (0, 1)$ y $\alpha > 0$, para cualquier número de iteraciones $N \in \mathbb{N}^*$ tal que $N > \beta_1/(1 - \beta_1)$, se tiene la siguiente cota:*

$$\mathbb{E}\left[\|\nabla F\left(x_{\tau_N}\right)\|_2^2\right] \leq 2R\frac{F\left(x_0\right) - F_*}{\alpha\tilde{N}} + E\left(\frac{1}{\tilde{N}} \ln\left(1 + \frac{R^2}{\left(1 - \beta_2\right)\epsilon}\right) - \frac{N}{\tilde{N}} \ln\left(\beta_2\right)\right) \tag{3.13}$$

donde $\tilde{N} := N - \frac{\beta_1}{1-\beta_1}$ *y* $E = \frac{\alpha d R L (1-\beta_1)}{(1-\beta_1/\beta_2)(1-\beta_2)} + \frac{12dR^2\sqrt{1-\beta_1}}{(1-\beta_1/\beta_2)^{3/2}\sqrt{1-\beta_2}} + \frac{2\alpha^2 dL^2\beta_1}{(1-\beta_1/\beta_2)(1-\beta_2)^{3/2}}$

3.3.2. Demostraciones

Con el objetivo de aligerar la notación a lo largo de la demostración se define:

$$G_n := \nabla F(x_{n-1}), \quad g_n := \nabla f(x_{n-1}), \quad u_n := \frac{m_n}{\sqrt{\epsilon + v_n}}, \quad U_n := \frac{g_n}{\sqrt{\epsilon + v_n}}, \quad [M] := \{1, ...M\} \tag{3.14}$$

Lemas técnicos

Lema 3.1 (Cota sobre dirección de actualización)**.** *Dado un punto $x_0 \in \mathbb{R}^d$ y las actualizaciones dadas en el Algoritmo 3.2 con las hipótesis hechas, además de $0 \leq \beta_1 < \beta_2 \leq 1$, para cualquier iteración $n \in \mathbb{N}^*$:*

$$\mathbb{E}\left[\sum_{i\in[d]} G_{n,i}\frac{m_{n,i}}{\sqrt{\epsilon+v_{n,i}}}\right] \geq \frac{1}{2}\left(\sum_{i\in[d]}\sum_{k=0}^{n-1}\beta_1^k\mathbb{E}\left[\frac{G_{n-k,i}^2}{\sqrt{\epsilon+\tilde{v}_{m,k+1,i}}}\right]\right) - \frac{\alpha_n^2 L^2}{4R}\sqrt{1-\beta_1}\left(\sum_{l=1}^{n-1}\|u_{n-l}\|_2^2\sum_{k=l}^{n-1}\beta_1^k\sqrt{k}\right) - \frac{3R}{\sqrt{1-\beta_1}}\left(\sum_{k=0}^{n-1}\left(\frac{\beta_1}{\beta_2}\right)^k\sqrt{k+1}\,\|U_{n-k}\|_2^2\right) \tag{3.15}$$

Demostración: Si se fija la iteración $n \in [N]$:

$$\mathbb{E}\left[\sum_{i\in[d]} G_{n,i}\frac{m_{n,i}}{\sqrt{\epsilon+v_{n,i}}}\right] = \mathbb{E}\left[\sum_{i\in[d]}\sum_{k=0}^{n-1}\beta_1^k G_{n,i}\frac{g_{n-k,i}}{\sqrt{\epsilon+v_{n,i}}}\right] = \underbrace{\mathbb{E}\left[\sum_{i\in[d]}\sum_{k=0}^{n-1}\beta_1^k G_{n-k,i}\frac{g_{n-k,i}}{\sqrt{\epsilon+v_{n,i}}}\right]}_{\mathbb{E}[A]} + \underbrace{\mathbb{E}\left[\sum_{i\in[d]}\sum_{k=0}^{n-1}\beta_1^k\left(G_{n,i}-G_{n-k,i}\right)\frac{g_{n-k,i}}{\sqrt{\epsilon+v_{n,i}}}\right]}_{\mathbb{E}[B]} \tag{3.16}$$

Trataremos ahora los dos sumandos por separado, comenzando primeramente por A.

Se define $\delta^2 = \sum_{j=n-k}^{n}\beta_2^{n-j}g_{j,i}^2$ y $r^2 = \mathbb{E}_{n-k-1}\left[\delta^2\right]$. Con esto podemos expresar $\tilde{v}_{m,k+1,i} - v_{n,i} = r^2 - \delta^2$. Introduciendo estas nuevas definiciones en A sumada a una conveniente manipulación algebraica:

$$\begin{aligned}\mathbb{E}\left[G_{n-k,i}\frac{g_{n-k,i}}{\sqrt{\epsilon+v_{n,i}}}\right] &= \mathbb{E}\left[G_{n-k,i}\frac{g_{n-k,i}}{\sqrt{\epsilon+\tilde{v}_{m,k+1,i}}} + G_{n-k,i}g_{n-k,i}\left(\frac{1}{\sqrt{\epsilon+v_{n,i}}} - \frac{1}{\sqrt{\epsilon+\tilde{v}_{m,k+1,i}}}\right)\right] \\ = \mathbb{E}\left[\mathbb{E}_{n-k-1}\left[G_{n-k,i}\frac{g_{n-k,i}}{\sqrt{\epsilon+\tilde{v}_{m,k+1,i}}}\right]\right] &+ \mathbb{E}\left[G_{n-k,i}g_{n-k,i}\frac{r^2-\delta^2}{\sqrt{\epsilon+v_{n,i}}\sqrt{\epsilon+\tilde{v}_{m,k+1,i}}(\sqrt{\epsilon+v_{n,i}}+\sqrt{\epsilon+\tilde{v}_{m,k+1,i}})}\right] \\ &= \mathbb{E}\left[\frac{G_{n-k,i}^2}{\sqrt{\epsilon+\tilde{v}_{m,k+1,i}}}\right] + \underbrace{\mathbb{E}\left[G_{n-k,i}g_{n-k,i}\frac{r^2-\delta^2}{\sqrt{\epsilon+v_{n,i}}\sqrt{\epsilon+\tilde{v}_{m,k+1,i}}(\sqrt{\epsilon+v_{n,i}}+\sqrt{\epsilon+\tilde{v}_{m,k+1,i}})}\right]}_{A_{II}}\end{aligned} \tag{3.17}$$

Esta última igualdad es debida principalmente a que $\mathbb{E}_{n-k-1}[G_{n-k,i}] = G_{n-k,i}$ y $\mathbb{E}_{n-k-1}[g_{n-k,i}] = G_{n-k,i}$ puesto que f es la función estocástica usada para muestrear F. Volvemos a tratar por separado A_{II}:

$$|A_{II}| \leq \underbrace{|G_{n-k,i}g_{n-k,i}|\frac{r^2}{\sqrt{\epsilon+v_{n,i}}(\epsilon+\tilde{v}_{m,k+1,i})}}_{\kappa} + \underbrace{|G_{n-k,i}g_{n-k,i}|\frac{\delta^2}{(\epsilon+v_{n,i})\sqrt{\epsilon+\tilde{v}_{m,k+1,i}}}}_{\rho} \tag{3.18}$$

Para acotar este término, haremos uso de la conocida *desigualdad de Young*:

$$\forall\lambda>0,\quad \forall a,b\in\mathbb{R} \quad\Rightarrow\quad ab < \frac{\lambda}{2}a^2 + \frac{b^2}{2\lambda} \qquad \text{(Desigualdad de Young)} \tag{3.19}$$

- Para κ escogemos y acotamos:

$$\lambda = \frac{\sqrt{1-\beta_1}\sqrt{\epsilon+\tilde{v}_{m,k+1,i}}}{2}, x = \frac{|G_{n-k,i}|}{\sqrt{\epsilon+\tilde{v}_{m,k+1,i}}}, y = \frac{|g_{n-k,i}|r^2}{\sqrt{\epsilon+\tilde{v}_{m,k+1,i}}\sqrt{\epsilon+v_{n,i}}} \Rightarrow$$
$$\kappa \leq \frac{G_{n-k,i}^2}{4\sqrt{\epsilon+\tilde{v}_{m,k+1,i}}} + \frac{1}{\sqrt{1-\beta_1}} \frac{g_{n-k,i}^2 r^4}{(\epsilon+\tilde{v}_{m,k+1,i})^{3/2}(\epsilon+v_{n,i})} \tag{3.20}$$

Usando ahora que $\epsilon + \tilde{v}_{m,k+1,i} \geq r^2$ y tomando la esperanza condicionada:

$$\mathbb{E}_{n-k-1}[\kappa] \leq \frac{G_{n-k,i}^2}{4\sqrt{\epsilon+\tilde{v}_{m,k+1,i}}} + \frac{1}{\sqrt{1-\beta_1}} \frac{r^2}{\sqrt{\epsilon+\tilde{v}_{m,k+1,i}}} \mathbb{E}_{n-k-1}\left[\frac{g_{n-k,i}^2}{\epsilon+v_{n,i}}\right] \tag{3.21}$$

- Sobre ρ se vuelve a aplicar la desigualdad de Young:

$$\lambda = \frac{\sqrt{1-\beta_1}\sqrt{\epsilon+\tilde{v}_{m,k+1,i}}}{2r^2}, x = \frac{|G_{n-k,i}\delta|}{\sqrt{\epsilon+\tilde{v}_{m,k+1,i}}}, y = \frac{|\delta g_{n-k,i}|}{\epsilon+v_{n,i}} \Rightarrow$$
$$\rho \leq \frac{G_{n-k,i}^2}{4\sqrt{\epsilon+\tilde{v}_{m,k+1,i}}} \frac{\delta^2}{r^2} + \frac{1}{\sqrt{1-\beta_1}} \frac{r^2}{\sqrt{\epsilon+\tilde{v}_{m,k+1,i}}} \frac{g_{n-k,i}^2 \delta^2}{(\epsilon+v_{n,i})^2} \tag{3.22}$$

Cabe destacar que en el caso de ser $r = 0$, se tiene casi seguro que $\delta^2 = 0$, que a su vez implica que $\rho = 0$. Gracias a esto nos aseguramos que en la anterior ecuación no se está tomando una división entre cero. Por otra parte, dado que $\epsilon + v_{n,i} \geq \delta^2$, tomando al esperanza condicionada y haciendo uso que $\mathbb{E}_{n-k-1}\left[\delta^2/r^2\right] = \mathbb{E}_{n-k-1}\left[\delta^2\right]/r^2 = r^2/r^2 = 1$:

$$\mathbb{E}_{n-k-1}[\rho] \leq \frac{G_{n-k,i}^2}{4\sqrt{\epsilon+\tilde{v}_{m,k+1,i}}} + \frac{1}{\sqrt{1-\beta_1}} \frac{r^2}{\sqrt{\epsilon+\tilde{v}_{m,k+1,i}}} \mathbb{E}_{n-k-1}\left[\frac{g_{n-k,i}^2}{\epsilon+v_{n,i}}\right] \tag{3.23}$$

Sumando ambas contribuciones (3.21) y (3.23) en (3.18):

$$\mathbb{E}_{n-k-1}[|A_{II}|] \leq \frac{G_{n-k,i}^2}{2\sqrt{\epsilon+\tilde{v}_{m,k+1,i}}} + \frac{2}{\sqrt{1-\beta_1}} \frac{r^2}{\sqrt{\epsilon+\tilde{v}_{m,k+1,i}}} \mathbb{E}_{n-k-1}\left[\frac{g_{n-k,i}^2}{\epsilon+v_{n,i}}\right] \tag{3.24}$$

Por definición de $\tilde{v}_{m,k+1,i}$, $r \leq \sqrt{\epsilon+\tilde{v}_{m,k+1,i}}$. Si además se usa la cota sobre el gradiente de F (3.9), $r \leq \sqrt{(k+1)}R$, se tiene:

$$\mathbb{E}_{n-k-1}[|A_{II}|] \leq \frac{G_{n-k,i}^2}{2\sqrt{\epsilon+\tilde{v}_{m,k+1,i}}} + \frac{2R}{\sqrt{1-\beta_1}}\sqrt{k+1}\mathbb{E}_{n-k-1}\left[\frac{g_{n-k,i}^2}{\epsilon+v_{n,i}}\right] \tag{3.25}$$

Tomando la esperanza total, y teniendo en cuenta $\epsilon + v_{n,i} \geq \epsilon + \beta_2^k v_{n-k,i} \geq \beta_2^k (\epsilon + v_{n-k,i})$:

$$\mathbb{E}[|A_{II}|] \leq \frac{1}{2}\mathbb{E}\left[\frac{G_{n-k,i}^2}{\sqrt{\epsilon+\tilde{v}_{m,k+1,i}}}\right] + \frac{2R}{\sqrt{1-\beta_1}\beta_2^k}\sqrt{k+1}\mathbb{E}\left[\frac{g_{n-k,i}^2}{\epsilon+v_{n-k,i}}\right] \tag{3.26}$$

Devolviendo esta cota sobre A_{II} al original A:

$$\mathbb{E}[A] \geq \sum_{i\in[d]}\sum_{k=0}^{n-1}\beta_1^k\left(\mathbb{E}\left[\frac{G_{n-k,i}^2}{\sqrt{\epsilon+\tilde{v}_{m,k+1,i}}}\right]-\left(\frac{1}{2}\mathbb{E}\left[\frac{G_{n-k,i}^2}{\sqrt{\epsilon+\tilde{v}_{m,k+1,i}}}\right]+\frac{2R}{\sqrt{1-\beta_1\beta_2^k}}\sqrt{k+1}\mathbb{E}\left[\frac{g_{n-k,i}^2}{\epsilon+v_{n-k,i}}\right]\right)\right)$$
$$=\frac{1}{2}\left(\sum_{i\in[d]}\sum_{k=0}^{n-1}\beta_1^k\mathbb{E}\left[\frac{G_{n-k,i}^2}{\sqrt{\epsilon+\tilde{v}_{m,k+1,i}}}\right]\right)-\frac{2R}{\sqrt{1-\beta_1}}\left(\sum_{i\in[d]}\sum_{k=0}^{n-1}\left(\frac{\beta_1}{\beta_2}\right)^k\sqrt{k+1}\mathbb{E}\left[\|U_{n-k}\|_2^2\right]\right) \tag{3.27}$$

Ahora acotemos el sumando B de (3.16). Fijando $k\in[0,n-1]$ y $i\in[d]$, gracias a la la desigualdad de Young:

$$\lambda=\frac{\sqrt{1-\beta_1}}{2R\sqrt{k+1}},\quad x=|G_{n,i}-G_{n-k,i}|,\quad y=\frac{|g_{n-k,i}|}{\sqrt{\epsilon+v_{n,i}}}\Rightarrow$$
$$\Rightarrow|B|\leq\sum_{i\in[d]}\sum_{k=0}^{n-1}\beta_1^k\left(\frac{\sqrt{1-\beta_1}}{4R\sqrt{k+1}}\left(G_{n,i}-G_{n-k,i}\right)^2+\frac{R\sqrt{k+1}}{\sqrt{1-\beta_1}}\frac{g_{n-k,i}^2}{\epsilon+v_{n,i}}\right). \tag{3.28}$$

En esta ocasión $i\in[d],\epsilon+v_{n,i}\geq\epsilon+\beta_2^k v_{n-k,i}\geq\beta_2^k\left(\epsilon+v_{n-k,i}\right)$, lo que hace:

$$\frac{g_{n-k,i}^2}{\epsilon+v_{n,i}}\leq\frac{1}{\beta_2^k}U_{n-k,i}^2 \tag{3.29}$$

Por otra parte, usando que F tiene gradiente lipschitziano, conjunto con la desigualdad de Jensen[3] y el hecho de que α_n es no decreciente:

$$\|G_n-G_{n-k}\|_2^2\leq L^2\|x_{n-1}-x_{n-k-1}\|_2^2=L^2\left\|\sum_{l=1}^{k}\alpha_{n-l}u_{n-l}\right\|_2^2\leq\alpha_n^2L^2k\sum_{l=1}^{k}\|u_{n-l}\|_2^2 \tag{3.30}$$

Introduciendo (3.29) y (3.30) en (3.28):

$$\begin{aligned}|B|&\leq\left(\sum_{k=0}^{n-1}\frac{\alpha_n^2L^2}{4R}\sqrt{1-\beta_1}\beta_1^k\sqrt{k}\sum_{l=1}^{k}\|u_{n-l}\|_2^2\right)+\left(\sum_{k=0}^{n-1}\frac{R}{\sqrt{1-\beta_1}}\left(\frac{\beta_1}{\beta_2}\right)^k\sqrt{k+1}\,\|U_{n-k}\|_2^2\right)\\&=\sqrt{1-\beta_1}\frac{\alpha_n^2L^2}{4R}\left(\sum_{l=1}^{n-1}\|u_{n-l}\|_2^2\sum_{k=l}^{n-1}\beta_1^k\sqrt{k}\right)+\frac{R}{\sqrt{1-\beta_1}}\left(\sum_{k=0}^{n-1}\left(\frac{\beta_1}{\beta_2}\right)^k\sqrt{k+1}\,\|U_{n-k}\|_2^2\right)\end{aligned} \tag{3.31}$$

Reagrupando la cota para A (3.27 con la cota para B (3.31 en la original (3.16) y combinando términos cuidadosamente se obtiene la desigualdad a demostrar.

■

Lema 3.2 (Cota sobre suma de desviaciones)**.** *Si se tiene $0\leq\beta_1<\beta_2\leq1$ y $(a_n)_{n\in\mathbb{N}^*}$ es una sucesión de reales, si se define $b_j:=\sum_{i=1}^{j}\beta_2^{j-i}a_i^2$ y $c_j=\sum_{i=1}^{j}\beta_1^{j-i}a_i$, se tiene que:*

$$\sum_{j=1}^{N}\frac{c_j^2}{\epsilon+b_j}\leq\frac{1}{(1-\beta_1)(1-\beta_1/\beta_2)}\left(\ln\left(1+\frac{b_N}{\epsilon}\right)-N\ln(\beta_2)\right) \tag{3.32}$$

[3]Desigualdad de Jensen [17]: $\left(\frac{1}{N}\sum_{i=1}^{N}x_i\right)^2\leq\frac{1}{N}\sum_{i=1}^{N}x_i^2$

Demostración: La desigualdad de Jensen permite acotar c_j^2 a j fijo:

$$\frac{c_j^2}{\epsilon + b_j} \leq \frac{1}{1-\beta_1} \sum_{l=1}^{j} \beta_1^{j-l} \frac{a_l^2}{\epsilon + b_j} \overset{(*)}{\leq} \frac{1}{1-\beta_1} \sum_{l=1}^{j} \left(\frac{\beta_1}{\beta_2}\right)^{j-l} \frac{a_l^2}{\epsilon + b_l} \tag{3.33}$$

La desigualdad $(*)$ es debida a que dado un l menor a j (i.e. $l \in [j]$), entonces $b_j \geq \beta_2^{j-l} b_l$. Por tanto sumando sobre todas las $j \in [n]$, entonces:

$$\begin{aligned} \sum_{j=1}^{n} \frac{c_j^2}{\epsilon + b_j} &\leq \frac{1}{1-\beta_1} \sum_{j=1}^{n} \sum_{l=1}^{j} \left(\frac{\beta_1}{\beta_2}\right)^{j-l} \frac{a_l^2}{\epsilon + b_l} = \frac{1}{1-\beta_1} \sum_{l=1}^{n} \frac{a_l^2}{\epsilon + b_l} \sum_{j=l}^{n} \left(\frac{\beta_1}{\beta_2}\right)^{j-l} \\ &\leq \frac{1}{(1-\beta_1(1-\beta_1/\beta_2)} \sum_{l=1}^{n} \frac{a_l^2}{\epsilon + b_l} \end{aligned} \tag{3.34}$$

Ahora manipulemos este último sumatorio aparte:

$$\begin{aligned} \frac{a_l^2}{\epsilon + b_l} &\leq \ln\left(\frac{a_j^2}{\epsilon + b_j}\right) \overset{b_j > a_j^2, a_j^2 > 0}{\leq} \ln\left(\frac{b_j}{\epsilon + b_j - a_j^2}\right) \overset{\epsilon > 0}{\leq} \ln\left(\frac{b_j + \epsilon}{\epsilon + b_j - a_j^2}\right) \leq \ln\left(\frac{b_j + \epsilon}{\epsilon + \beta_2 b_{j-1}}\right) = \\ &= \ln\left(\frac{b_j + \epsilon}{\epsilon + \beta_2 b_{j-1}} \frac{b_{j-1} + \epsilon}{b_{j-1} + \epsilon}\right) = \ln\left(\frac{b_j + \epsilon}{\epsilon + b_{j-1}}\right) + \underbrace{\ln\left(\frac{b_{j-1} + \epsilon}{\epsilon + \beta_2 b_{j-1}}\right)}_{\leq -\ln \beta_2} \end{aligned} \tag{3.35}$$

Al sumar sobre j:

$$\sum_{j=1}^{N} \frac{a_j^2}{\epsilon + b_j} \leq \ln\left(\prod_{j=1}^{N} \frac{\epsilon + b_j}{\epsilon + b_{j-1}}\right) - N \ln \beta_2 = \ln\left(\frac{\epsilon + b_N}{\epsilon + 0}\right) - N \ln(\beta_2) \tag{3.36}$$

Recuperando esta igualdad en (3.34) se obtiene la desigualdad deseada.

■

Recapitulando, estos dos lemas, pese a que su objetivo es facilitar la demostración de los teoremas de acotación del Algoritmo 3.2, son también ilustrativos per se. Por una parte, el lema 3.1 proporciona una cota a la desviación dirección real de descenso para una iteración concreta; mientras que el 3.2 asegura que, pese a que una iteración concreta desvíe de la dirección del mínimo real, la suma de todas las desviaciones siguientes no puede superar el término logarítmico que se encuentra en (3.32).

Para concluir con los resultados previos necesarios, se enuncian ahora unas desigualdades sin demostraciones (en caso de precisar de una demostración del mismo, acuda al apéndice A).

Lema 3.3. *Dado $a \in (0,1)$ y $Q \in \mathbb{N}$:*

$$\sum_{q=0}^{Q-1} a^q \sqrt{q+1} \leq \frac{2}{(1-a)^{3/2}} \tag{3.37}$$

$$\sum_{q=0}^{Q-1} a^q \sqrt{q}(q+1) \leq \frac{4a}{(1-a)^{5/2}} \tag{3.38}$$

Demostraciones de los teoremas 3.1 y 3.2

Demostrar los resultados 3.1 y 3.2, implica la manipulación de términos poco manejables conjunto a la aplicación sucesiva de los lemas anteriormente estudiados. Aprovechando la similaridad existente entre Adagrad y Adam, podemos realizar parte de estas manipulaciones en conjunto, para posteriormente finalizar los resultados especificando según sea necesario.

Primeramente, recordemos que la actualización de los pesos (x_n en nuestra abstracción) toma la forma:

$$x_n - x_{n-1} = \alpha_n u_n \qquad \text{donde} \quad u_n = \frac{m_n}{\sqrt{\epsilon + v_n}} \quad \text{y} \quad \alpha_n = \alpha \cdot (1-\beta_1)\sqrt{\frac{1-\beta_2^n}{1-\beta_2}} \tag{3.39}$$

Siendo $\alpha > 0$, $\beta_1 \in [0,1)$ y $\beta_2 \in [0,1)$, esto significa que α_n es no decreciente.

Manipulaciones comunes a ambos resultados

Usando que el gradiente de F es lipschitziano, sustituyendo en (3.11) los valores $x = x_n$ e $y = x_{n-1}$ (i.e. $x - y = \alpha_n G_n^T u_n$):

$$[F(x_n)] \leq [F(x_{n-1})] - \alpha_n \underbrace{G_n^T u_n}_{(*)} + \frac{\alpha_n^2 L}{2}\left[||u_n||_2^2\right] \tag{3.40}$$

Sobre $(*)$ se puede aplicar el lema 3.1. Con esto, si se toman esperanzas a ambos lados de la desigualdad anterior, se llega a:

$$\begin{aligned}\mathbb{E}\left[F(x_n)\right] \leq \mathbb{E}\left[F\left(x_{n-1}\right)\right] - \frac{\alpha_n}{2}\left(\sum_{i\in[d]}\sum_{k=0}^{n-1}\beta_1^k \mathbb{E}\left[\frac{G_{n-k,i}^2}{2\sqrt{\epsilon + \tilde{v}_{m,k+1,i}}}\right]\right) + \frac{\alpha_n^2 L}{2}\mathbb{E}\left[\|u_n\|_2^2\right] + \\ + \frac{\alpha_n^3 L^2}{4R}\sqrt{1-\beta_1}\left(\sum_{l=1}^{n-1}\|u_{n-l}\|_2^2 \sum_{k=l}^{n-1}\beta_1^k\sqrt{k}\right) + \frac{3\alpha_n R}{\sqrt{1-\beta_1}}\left(\sum_{k=0}^{n-1}\left(\frac{\beta_1}{\beta_2}\right)^k\sqrt{k+1}\,\|U_{n-k}\|_2^2\right).\end{aligned} \tag{3.41}$$

Si se define $\Omega_n := \sqrt{\sum_{j=0}^{n-1}\beta_2^j}$, usando (3.9), para cualquier $k \in \mathbb{N}$ se tiene que para cualquier $k < n$: $\sqrt{\epsilon + \tilde{v}_{n,k+1,i}} \leq R\Omega_n$, lo que conlleva:

$$\begin{aligned}\mathbb{E}\left[F\left(x_n\right)\right] \leq \mathbb{E}\left[F\left(x_{n-1}\right)\right] - \frac{\alpha_n}{2R\Omega_n}\sum_{k=0}^{n-1}\beta_1^k\mathbb{E}\left[\|G_{n-k}\|_2^2\right] + \frac{\alpha_n^2 L}{2}\mathbb{E}\left[\|u_n\|_2^2\right] \\ + \frac{\alpha_n^3 L^2}{4R}\sqrt{1-\beta_1}\left(\sum_{l=1}^{n-1}\|u_{n-l}\|_2^2\sum_{k=l}^{n-1}\beta_1^k\sqrt{k}\right) + \frac{3\alpha_n R}{\sqrt{1-\beta_1}}\left(\sum_{k=0}^{n-1}\left(\frac{\beta_1}{\beta_2}\right)^k\sqrt{k+1}\,\|U_{n-k}\|_2^2\right)\end{aligned} \tag{3.42}$$

Si sumamos sobre $n \in [N]$, y dado que α_n es no decreciente ($\alpha_n \leq \alpha_N$):

$$\mathbb{E}\left[F\left(x_N\right)\right] - F\left(x_0\right) \leq \underbrace{\left(-\frac{1}{2R}\right) \sum_{n\in[N]} \frac{\alpha_n}{\Omega_n} \sum_{k=0}^{n-1} \beta_1^k \mathbb{E}\left[\|G_{n-k}\|_2^2\right]}_{A} + \underbrace{\frac{\alpha_N^2 L}{2} \mathbb{E}\left[\sum_{n\in[N]} \|u_n\|_2^2\right]}_{B} +$$

$$+ \underbrace{\frac{\alpha_N^3 L^2}{4R}\sqrt{1-\beta_1}\left(\sum_{n\in[N]}\sum_{l=1}^{n-1}\mathbb{E}[\|u_{n-l}\|_2^2]\sum_{k=l}^{n-1}\beta_1^k\sqrt{k}\right)}_{C} + \underbrace{\frac{3\alpha_N R}{\sqrt{1-\beta_1}}\left(\sum_{n\in[N]}\sum_{k=0}^{n-1}\left(\frac{\beta_1}{\beta_2}\right)^k\sqrt{k+1}\mathbb{E}[\|U_{n-k}\|_2^2]\right)}_{D} \tag{3.43}$$

Vamos a ocuparnos de cada uno de los 4 sumandos por separado:

- El término A, dada su forma, es conveniente estudiarlo posteriormente usando la naturaleza específica de β_2.
- Podemos aplicar el lema 3.2 para la suma presente en este término B. Fijando $i \in [d]$:

$$\sum_{i=1}^{n} \frac{\overbrace{m_i}^{u_{n,i}}}{\sqrt{\epsilon + v_n}} = \sum_{n=1}^{N} \frac{\left(\sum_{i=1}^{n} \beta_1^{n-i} g_i\right)^2}{\sqrt{\epsilon + \sum_{i=1}^{n} \beta_2^{n-i} g_i^2}} \leq \frac{1}{(1-\beta_1)(1-\beta_1/\beta_2)} \left(\ln\left(1 + \frac{v_{N,i}}{\epsilon}\right) - N\ln(\beta_2)\right) \tag{3.44}$$

  Luego:

$$B < \frac{\alpha_N^2 L}{2(1-\beta_1)(1-\beta_1/\beta_2)} \sum_{i\in[d]} \left(\ln\left(1 + \frac{v_N}{\epsilon}\right) - N\ln(\beta_2)\right) \tag{3.45}$$

- En cuanto a C, conviene realizar un cambio de índice a $j := n-l$, que permite manipular cómodamente:

$$\begin{aligned} C &= \frac{\alpha_N^3 L^2}{4R}\sqrt{1-\beta_1}\left(\sum_{n=1}^{N}\sum_{j=1}^{n-1}\mathbb{E}[\|u_j\|_2^2]\sum_{k=n-j}^{n-1}\beta_1^k\sqrt{k}\right) = \frac{\alpha_N^3 L^2}{4R}\sqrt{1-\beta_1}\sum_{j=1}^{N}\mathbb{E}[\|u_j\|_2^2]\sum_{n=j}^{N}\sum_{k=n-j}^{n-1}\beta_1^k\sqrt{k} \\ &= \frac{\alpha_N^3 L^2}{4R}\sqrt{1-\beta_1}\sum_{j=1}^{N}\mathbb{E}[\|u_j\|_2^2]\sum_{k=0}^{N-1}\beta_1^k\sqrt{k}\sum_{n=j}^{j+k}1 = \frac{\alpha_N^3 L^2}{4R}\sqrt{1-\beta_1}\sum_{j=1}^{N}\mathbb{E}[\|u_j\|_2^2]\sum_{k=0}^{N-1}\beta_1^k\sqrt{k}(k+1) \end{aligned} \tag{3.46}$$

  Observamos que podemos acotar este último sumatorio en virtud del lema 3.3 - desigualdad (3.38):

$$\sum_{k=0}^{N-1}\beta_1^k\sqrt{k}(k+1) \leq \frac{4\beta_1}{(1-\beta_1)^{5/2}} \Rightarrow C \leq \frac{\alpha_N^3 L^2}{R}\frac{\beta_1}{(1-\beta_1)^2}\sum_{j=1}^{N}\mathbb{E}[\|u_j\|_2^2] \tag{3.47}$$

  Sobre cada una de las componentes i de este último sumatorio empleamos nuevamente el lema 3.2, obteniendo análogamente la desigualdad (3.44). Esto proporciona:

$$C \leq \frac{\alpha_N^3 L^2 \beta_1}{R(1-\beta_1)^3(1-\beta_1/\beta_2)} \sum_{i\in[d]} \left(\ln\left(1 + \frac{v_{N,i}}{\epsilon}\right) - N\ln(\beta_2)\right) \tag{3.48}$$

- Para terminar, cambiando el índice al igual que en el caso anterior: $j = n - l$, el término D:

$$\begin{aligned} D &= \frac{3\alpha_N R}{\sqrt{1-\beta_1}} \left(\sum_{n=1}^{N} \sum_{j=1}^{n-1} \left(\frac{\beta_1}{\beta_2} \right)^{n-j} \sqrt{n-j+1} \mathbb{E}[\|U_j\|_2^2] \right) \\ &= \frac{3\alpha_N R}{\sqrt{1-\beta_1}} \sum_{j=1}^{N} \mathbb{E}[\|U_j\|_2^2] \sum_{n=j}^{n-1} \left(\frac{\beta_1}{\beta_2} \right)^{n-j} \sqrt{n-j+1} \end{aligned} \tag{3.49}$$

Podemos acotar el último sumatorio usando el lema 3.3, esta vez con (3.37):

$$\sum_{n=j}^{N} \left(\frac{\beta_1}{\beta_2} \right)^{n-j} \sqrt{(n-j)+1} \overset{a=\beta_1/\beta_2}{\leq} \frac{2}{1-(\beta_1/\beta_2)^{3/2}} \tag{3.50}$$

Es decir:

$$D \leq \frac{6\alpha_N R}{\sqrt{1-\beta_1}\left(1-(\beta_1/\beta_2)^{3/2}\right)} \sum_{n=1}^{N} \mathbb{E}[\|U_j\|_2^2] \tag{3.51}$$

Sobre el último sumatorio podemos usar el lema 3.2 aplicado a cada componente $i \in [d]$. Nótese que la sucesión a sumar no tiene exactamente la misma forma que aquella proporcionada en el enunciado. Sin embargo, observamos cómo podemos transformarlo en la demostración de dicho resultado para tratar con el término ahora presente (concretamente, se corresponde con lo visto en la ecuación (3.34) de la demostración, obteniendo entonces la desigualdad 3.35)) [4] :

$$\sum_{n=1}^{N} U_{j,i} = \sum_{n=1}^{N} \frac{g_{j,i}}{\sqrt{\epsilon + v_{j,i}}} \leq \ln\left(1 + \frac{v_{N,i}}{\epsilon}\right) - N \ln \beta_2 \tag{3.52}$$

Por lo que

$$D \leq \frac{6\alpha_N R}{\sqrt{1-\beta_1}\left(1-(\beta_1/\beta_2)^{3/2}\right)} \sum_{i\in[d]} \left(\ln\left(1 + \frac{v_{N,i}}{\epsilon}\right) - N \ln \beta_2 \right) \tag{3.53}$$

Si reagrupamos las anteriores cotas (3.45), (3.48) y (3.53) en la expresión original (3.43), teniendo además en cuenta la cota sobre F (3.7), se obtiene:

$$\begin{aligned} &F_* - F(x_0) \\ &\leq A + \left[\frac{R\alpha_N^2 L(1-\beta_1)^2 + 2\alpha_N^3 L^2 \beta_1}{2R(1-\beta_1)(1-\beta_1/\beta_2)} + \frac{6\alpha_N R}{\sqrt{(1-\beta_1)(1-(\beta_1/\beta_2)^{3/2})}} \right] \sum_{i\in[d]} \left(\ln\left(1 + \frac{v_{N,i}}{\epsilon}\right) - N \ln \beta_2 \right) \end{aligned} \tag{3.54}$$

Teorema 3.1: Convergencia de Adagrad.

Demostración: Primeramente, por tratarse de Adagrad, por definición $\beta_2 = 1$, lo que transforma la tasa de entrenamiento a $\alpha_n = (1-\beta_1)\alpha$. Es decir, α_n no depende realmente de n. Esto permite reagrupar y simplificar términos en (3.54):

$$F_* - F(x_0) \leq A + \frac{\alpha^2 RL(1-\beta_1)^3 + 2\alpha^3 L\beta_1(1-\beta_1)^2 + 6\alpha R^2}{2R(1-\beta_1)} \sum_{i\in[d]} \ln\left(1 + \frac{v_{N,i}}{\epsilon}\right) \tag{3.55}$$

[4] De otro modo, puede adaptarse el lema 3.2 para obtener el lema A.3, aplicable en esta situación.

Si además se tiene en cuenta (3.9): $v_N < NR^2$.

$$F_* - F(x_0) \leq A + d\frac{\alpha^2 RL(1-\beta_1)^3 + 2\alpha^3 L\beta_1(1-\beta_1)^2 + 6\alpha R^2}{2R(1-\beta_1)} \ln\left(1 + \frac{NR^2}{\epsilon}\right) \tag{3.56}$$

Ahora acotemos A. Usando su definición, $\Omega = \sqrt{\sum_{j=0}^{(n-1)} 1} = \sqrt{n} \leq \sqrt{N}$:

$$\begin{aligned} A &:= \left(-\frac{1}{2R}\right) \sum_{n\in[N]} \frac{\alpha_n}{\Omega_n} \sum_{k=0}^{n-1} \beta_1^k \mathbb{E}\left[\|G_{n-k}\|_2^2\right] \leq -\frac{\alpha(1-\beta_1)}{2R\sqrt{N}} \sum_{j=1}^{N} \mathbb{E}\left[\|G_{n-k}\|_2^2\right] \sum_{n=j}^{N} \beta_1^k \leq \\ &\leq -\frac{\alpha(1-\beta_1)}{2R\sqrt{N}} \sum_{n=1}^{N} \mathbb{E}\left[\|G_{n-k}\|_2^2\right] \frac{1-\beta_1^{N-j+1}}{1-\beta_1} = -\frac{\alpha}{2R\sqrt{N}} \sum_{n=0}^{N-1} (1-\beta_1^{N-j}) \mathbb{E}\left[\|G_{n-k}\|_2^2\right] \end{aligned} \tag{3.57}$$

En la anterior ecuación observamos el término de proporcionalidad que define a τ_N (y de aquí la motivación de esta definición tan poco natural a primera vista). Es interesante ver que al hacer la suma de este término:

$$\sum_{j=0}^{N-1} (1-\beta_1^{N-j}) = N - \sum_{j=0}^{N-1} \beta_1^{N-j} = N - \beta_1 \frac{1-\beta_1^N}{1-\beta_1} \overset{\beta_1 \geq 0}{\geq} N - \frac{\beta_1}{1-\beta_1} := \tilde{N} \tag{3.58}$$

De este modo, si introducimos en la anterior desigualdad la definición de τ_N así como esta última cota, es directo ver que:

$$A \leq -\frac{\alpha\tilde{N}}{2R\sqrt{N}} \mathbb{E}\left[\|\nabla F(x_{\tau_N})\|_2^2\right] \tag{3.59}$$

Con ello, volviendo sobre (3.55):

$$\begin{aligned} &F_* - F(x_0) \\ &\leq -\frac{\alpha\tilde{N}}{2R\sqrt{N}} \mathbb{E}\left[\|\nabla F(x_{\tau_N})\|_2^2\right] + d\frac{\alpha^2 RL(1-\beta_1)^3 + 2\alpha^3 L\beta_1(1-\beta_1)^2 + 6\alpha R^2}{2R(1-\beta_1)} \ln\left(1 + \frac{NR^2}{\epsilon}\right) \end{aligned} \tag{3.60}$$

Reagrupando esta ecuación coincide con el resultado a demostrar.

■

Teorema 3.2: Convergencia de Adam.

Demostración: Análogamente podemos acotar A para Adam, en el cual $\beta_2 \neq 1$:

$$A = -\frac{\alpha}{2R} \sum_{j=0}^{N-1} (1-\beta_1^{N-j}) \mathbb{E}\left[||\nabla F(x_j)||_2^2\right] \overset{(3,58)}{\leq} -\frac{\alpha\tilde{N}}{2R} \mathbb{E}\left[||\nabla F(x_{\tau_N})||_2^2\right] \tag{3.61}$$

En esta ocasión, usando (3.9), se tiene que $v_{N,i} \leq \frac{R^2}{1-\beta_2}$. Introduciendo esta cota en (3.54), sumado a $\alpha_n \leq \alpha_N = \alpha(1-\beta_1)\Omega_n \leq \alpha\frac{1-\beta_1}{\sqrt{1-\beta_2}}$:

$$F_* - F(x_0) \leq -\frac{\alpha\tilde{N}}{2R}\mathbb{E}\left[||\nabla F(x_{\tau_N})||_2^2\right] + d\left[\frac{R\alpha^2 L(1-\beta_1)^4 + 2\alpha^3(1-\beta_1)^3 L^2\beta_1/\sqrt{1-\beta_2}}{2R(1-\beta_1)(1-\beta_2)(1-\beta_1/\beta_2)} + \frac{6\alpha R}{\sqrt{(1-\beta_1)(1-\beta_2)(1-(\beta_1/\beta_2)^{3/2})}}\right] \cdot \cdot \left(\ln\left(1+\frac{R^2}{\epsilon(1-\beta_2)}\right) - N\ln\beta_2\right) \tag{3.62}$$

Es suficiente reorganizar para recuperar el resultado a demostrar.

■

3.4. Resultados experimentales

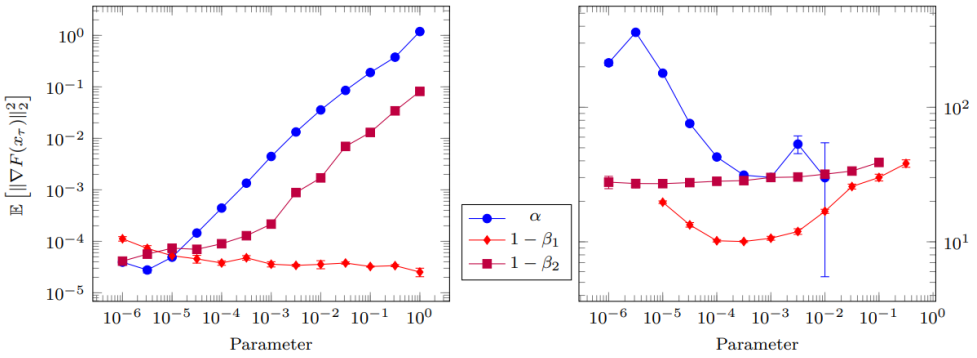


Figura 3.2: Défossez et al (2022). Media experimental observada de la norma del gradiente tras de un número fijo de iteraciones vairando individualmente de los hiperparámetros α, $1-\beta_1$ y $1-\beta_2$, para un problema *toy task* -izquierda- y clasificación de imágenes CIFAR-10 -derecha-. [15]

En primer lugar, se han hecho una serie de hipótesis sobre la función de coste que son bastante restrictivas. En concreto, suponer que su gradiente es lipschitziano así como que el gradiente de la función de muestreo es uniformemente acotada casi seguro (3.8) restringen a muchas de las posibles funciones de coste para cumplir con los resultados posteriormente enunciados. Pese a ello, esta demostración proporciona el mejor orden de convergencia hasta la fecha.

Por una parte, se pueden aligerar estas hipótesis. En la demostración original de Adam (posteriormente corregida), estas dos hipótesis son reemplazadas por la acotación del gradiente de la función de coste (en la norma L_2 así como L_∞) y suposición de convexidad sobre F. Sin embargo, en este caso el orden de convergencia del algoritmo es menor, siendo $\frac{\sum_{n=1}^{N}[f_n(x_n)-f_n(x^*)]}{N} = O(N^{-1/2})$.

Por otra parte, podemos sustituir únicamente la hipótesis sobre la cota del gradiente de f_n (3.8) por una acotación afín de crecimiento [5] [18], obteniendo entonces un decaimiento como $O(\ln(N_{epochs})/N_{epochs}$, siendo N_{epochs} el número de épocas y no iteraciones.

En cuanto la cota concreta demostrada teóricamente, en la figura 3.2 gráfica izquierda se estudia la dependencia experimental con los distintos hiperparámetros, certificando así el resultado obtenido en los teoremas [15]. No obstante, este comportamiento no es verdadero para toda clase de problemas, como se muestra en la gráfica derecha, principalmente a causa de la hipótesis (3.10), observando además una tendencia a la inestabilización del entrenamiento con el crecimiento del hiperparámetro α.

[5] La norma del gradiente estocástico en este caso estará acotada por una función afín de la norma del gradiente esperado.

Fuere cuál fuere el orden de precisión del algoritmo 3.2, lo que es claro y así se manifiesta en su aplicación es un mejor rendimiento que una extensa variedad de optimizadores, entre las cuales se incluye el descenso del gradiente estocástico como se ve en la figura 3.3a). Del mismo modo, en la figura 3.3b) se estudia la rapidez de los mismo para alcanzar una tolerancia prefijada [20]. Aquí vemos como Adam arroja casi siempre unos resultados más satisfactorios que la media, siendo además más rápido. Vemos además que combinado con con los beneficios de momento de Nesterov -NAdam (Nesterov-accelerated Adaptive Moment Estimation)- los resultados son aún mejores en general.

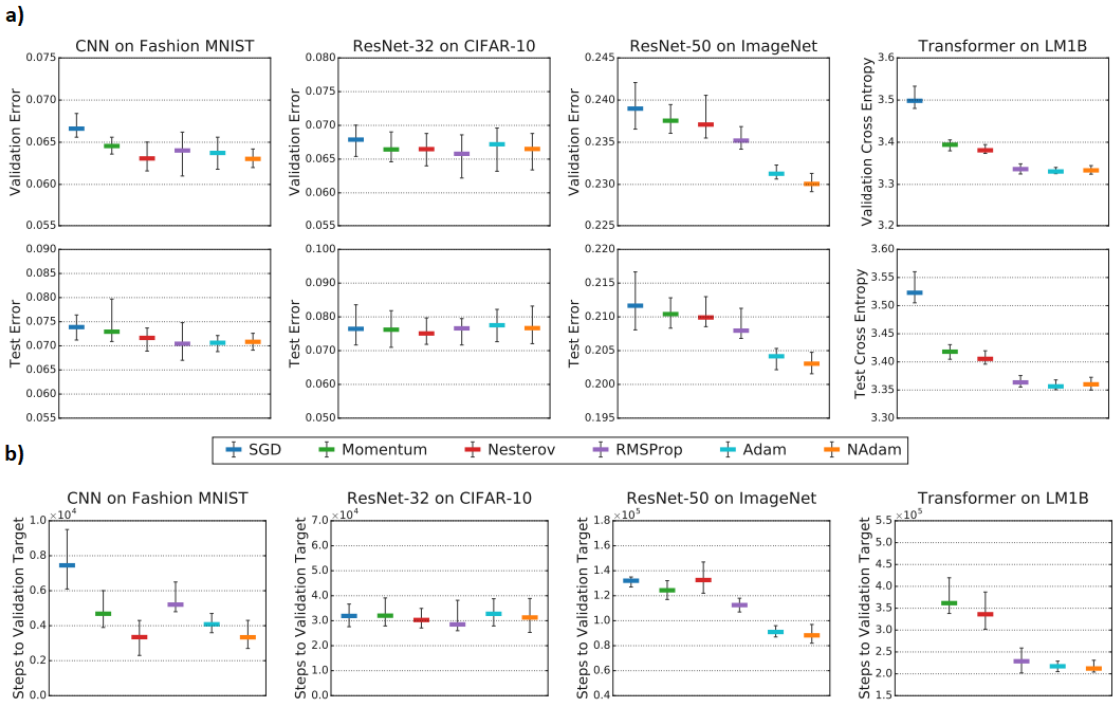


Figura 3.3: Dami Choiet al (2020). a) Comparación del rendimiento de múltiples optimizadores, tanto para el error cometido en el conjunto de validación -1° fila- como para conjunto de test -2° fila-. b) Comparación de la rapidez de múltiples optimizadores para alcanzar un error objetivo común prefijado. [20]

Es importante mencionar que para la obtención de unos resultados precisos es necesario realizar un estudio previo de elección de los hiperparámetros del algoritmo, bien para Adam como para cualquier otro optimizador. En [20] se hace un estudio sobre dicha afirmación, demostrando como debido a una mala sintonización de parámetros se puede observar como Adam puede obtener peores resultados que su competencia. Y es que la elección de parámetros es una cuestión delicada que implica un estudio aparte por sí mismo que aquí no trataremos con detalle.

No se puede pasar nada por alto. Como ilustración, en el caso de los algoritmos 3.1 y 3.2, recordemos que el hiperparámetro ϵ es introducido únicamente para evitar una división entre 0. Sin embargo, no deja de ser un hiperparámetro como los son β_1 o α, y por tanto este se puede escoger de la manera más conveniente posible. Por ello, no se debe suponer desde un primer momento que los mejores resultados son obtenidos para $\epsilon \to 0$. En el artículo previamente mencionado se ve como una elección de incluso $\epsilon = 9475$ tiene mejor rendimiento que otras muchas elecciones a primera vista más intuitivas.

Capítulo 4

Resultados: programación con Python

Para terminar el trabajo, el objetivo de este capítulo es la puesta en práctica de las técnicas estudiadas teóricamente hasta este momento para crear una red neuronal desde cero. Se ha escogido por su optimización y conveniencia Python como lenguaje de programación, utilizando únicamente la biblioteca Numpy[1].

El código tiene como objetivo la clasificación de caracteres manuscritos. Para entrenar y testear la red, emplearemos la base de datos proporcionada por el MNIST (Modified National Institute of Standards and Technology) de caracteres escritos a mano [21]. Esta consta de 70.000 imágenes de caracteres con sus correspondientes clasificaciones.

Para su implementación, tal como se ha comentado anteriormente, dado que el manejo del conjunto de datos es un aspecto sensible que consume mucho tiempo y es ajeno a los propósitos de este trabajo, se ha decidido emplear un conjunto previamente configurado. En concreto, se puede descargar aquí el archivo comprimido con los datos "crudos", mientras que el código para el manejo de este archivo ha sido adaptado partiendo de un paquete creado por M. Nielsen [22] (link a GitHub).

De este modo, las imágenes han sido previamente traducidas para estar contenidas en variables de tipo `tuple` de dimensión 2. En la primera de sus componentes se almacena la imagen codificada como un `Numpy array`[2] de tamaño 784. La segunda de las entradas contiene un `Numpy array`, esta vez con una longitud de 10 entradas. Aquí la entrada i-ésima toma el valor 1, siendo el resto de entradas nulas, indicando que la imagen se trata de un carácter de tipo i.

La base de datos ha sido dividida en 3 subconjuntos: `training_data`, `test_data` y `validation_data`. El primero de ellos consta de 50.000 imágenes y será utilizado para el aprendizaje de la red neuronal. El segundo es empleado para la comprobación del rendimiento de la red en cada una de las épocas del aprendizaje. Con esta información se ha generado la figura 4.1. Para terminar, los datos de validación tienen como objetivo validar la eficiencia de la red una vez terminado el entrenamiento. Esta descomposición es una práctica habitual en la aplicación de Machine Learning.

4.1. Estructura del código

El código fuente del programa se puede encontrar en el Apéndice **??** o descargar en este link. Este ha sido enteramente construido por mí, Ignacio Acedo (a excepción del método de carga del conjunto de datos). Como no puede ser de otra forma, el código se ha basado en la Programación Orientada a Objetos. Esto es así pues permite la encapsulación y organización de los métodos; así como facilita la modificación futura que pueda sufrir el código (corrección de errores o adición de nuevas funcionalidades). El código consta de:

[1] Matplotlib, pickle, gzip y time son usados para tareas auxiliares en vista a la recepción de datos y creación de figuras, no siendo parte del núcleo de la red.

[2] Este `Numpy array` está compuesto a su vez por entradas de tipo `float32`

- Una interfaz[3] `Optimizer`, que representa el algoritmo optimizador a emplear para la minimización de la función de coste. Va a ser implementada por dos nuevas clases `SGD` (descenso del gradiente estocástico) y `Adam`. Esta interfaz obliga a crear un método `train` que será utilizado por la red neuronal para la sintonización de parámetros.

- Una clase `NeuralNetwork`, que representa a las redes neuronales creadas. En este caso, se especifica que es un clasificador de dígitos, pero dada su programación puede ser empleada para cualquier otro programa [4]. Se han creado los siguientes métodos:

  - `__init__`: constructor de la clase para definir los siguientes campos: `input_size`, `hidden_sizes` y `output_sizes` -arquitectura de la red-; y `activation_function_name` y `cost_function_name` -nombres de las funciones de activación y coste que se desean emplear. En caso de no especificar estos dos últimos valores, son tomados por defecto a la función sigmoide como activación y el error cuadrático para la función de pérdida. Finalmente, inicializa los pesos y sesgos con distribuciones normales centradas en torno a 0.

  - `getActivations`: utilizada para la obtención de las activaciones de cada capa así como de sus transferencias correspondientes.

  - `feedforward`: usado para la obtención del vector de ouput de la red. Este consiste en un `Numpy array` de 10 componentes como se ha explicado. Sin embargo, a diferencia del conjunto de entrenamiento, no se tiene un array con un 1 y el resto 0's para determinar la clasificación, sino que se tienen números escalares cualesquiera.

  - `predict`: determina el dígito final predicho por la red, tomando aquella posición del vector de output proporcionado por `feedforward` cuya componente sea máxima (argmax_i $\texttt{output}_i$).

  - `probabilities`: dada la forma del vector de outputs, este método permite la obtención de las probabilidades de cada dígito para una imagen introducida. No obstante, no se han utilizado las técnicas de inferencia explicadas en la sección 2.3.

  - `backward_propagation`: implementación de la técnica del Backpropagation visto en el pseudoalgorimto 2.2 para calcular el gradiente de la función de coste *para una imagen única*.

  - `train`: En sus argumentos, entre otros, es necesario introducir una instancia de una clase que implemente la interfaz `Optimizer`. Agrupa y organiza los anteriores métodos en el orden apropiado para actualizar los pesos gracias al método `train` del propio optimizador. Además, posee un argumento opcional para decidir si se testea y representa el rendimiento de la red respecto del avance de las épocas de entrenamiento.

  - `check_performance` y `plot_performance`: usados para el testeo del rendimiento de la red neuronal. El primero, computa el porcentaje de imágenes bien clasificadas para el conjunto de test y lo muestra en consola a cada finalización de época; mientras que el segundo lo representa en una figura.

- Una clase `Input`, que representa el conjunto de datos con los que se va utilizar para alimentar la red, ya sea para el entrenamiento o para clasificar imágenes. Para ello, posee una serie de métodos estáticos, de modo que son accesibles sin necesidad de crear una instancia. Estos son:

  - `load_mnist_database`: empleado para cargar la base de datos del MNIST desde el archivo `mnist.pkl.gz` usando el código adaptado de M. Nielsen como anteriormente mencionado. Este transforma las imágenes a vectores y los divide en los subconjuntos de entrenamiento, test y validación previamente explicados.

  - `picture2input`: toma como argumento la ruta de una imagen en cualquier formato (`png`, `jpg`...) conteniendo un dígito manuscrito y lo transforma en un `Numpy array` que pueda ser empleado para alimentar la red.

[3] Abstract Base Classes en Python.
[4] Nótese que debido al No Free Lunch Theorem no se puede asegurar un buen rendimiento para otras aplicaciones.

- `mini_batch_data`: descompone el conjunto de entrenamiento en mini batches para ser posteriormente empleados por el optimizador. Permite especificar el tamaño de cada uno de los subconjuntos.

- La clase `__main__`: que contiene el programa principal que lanza la creación del optimizador DGE y Adam y dos instancias idénticas de una red neuronal. Entrena una red con DGE y otra con Adam; y posteriormente representa gráficamente la evolución de la precisión con las épocas en ambos casos.

4.2. Resultados

Para la creación de la red de clasificación de dígitos manuscritos se han escogido los siguientes parámetros: `input_size`=784, `hidden_sizes` =[28] y `output_size`=10, con funciones de activación sigmoide (1.1) y función de coste el error cuadrático (1.5). Se escoge además `mini_batch_size`=20 y `number_epochs`=8. Esto es: una red con una sola capa oculta de 28 neuronas. Vamos a crear dos instancias idénticas de la clase `NeuralNetwork` con el objetivo de entrenar una de ellas con el método del descenso del gradiente y la otra con Adam. Esto permite analizar de primera mano las afirmaciones de los artículos anteriormente estudiados.

Todos los parámetros han sido seleccionados de manera que se obtengan los mejores resultados posibles en ambos casos. Sin embargo, hemos forzado que la capa oculta tenga exactamente 28 neuronas pues las imágenes son descompuestas primeramente en 28 bloques tanto en horizontal como en vertical, que tras reordenar se traducen en el vector de $28^2 = 784$ componentes resultante. De este modo, nuestra capa de neuronas oculta estudia las características de estos bloques. Además se ha verificado posteriormente que con esta capa intermedia se obtienen los resultados óptimos.

Tras este ajuste previo, se toma para DGE $\alpha = 8$. Por parte de Adam, la configuración óptima es $\alpha = 0{,}017$, $\beta_1 = 0{,}86$, $\beta_2 = 0{,}991$ y $\epsilon = 10^{-6}$. Con esto, hemos obtenido para DGE una precisión del 94 % y para Adam un poco más del 95 %. Se observa que, tal como se afirmaba en la figura 3.3, Adam es capaz de obtener una precisión prefijada con menos iteraciones. Sin embargo, se ha de destacar que el entrenamiento con DGE ha durado 29,71 segundos para completar las 8 épocas, mientras que Adam se ha demorado hasta 46,67 segundos. Esto es claramente porque Adam es más sofisticado, implicando el almacenamiento de más variables así como operaciones intermedias.

Por otra parte, es importante remarcar que el código aquí realizado es un mero ejercicio educativo. En la práctica todos estos métodos han sido previamente creados y optimizados de una manera más óptima a la aquí diseñada. Asimismo, otras técnicas más resolutivas son implementadas, lo que asimismo mejoraría el rendimiento apreciablemente. Todo esto es fácilmente accesible con bibliotecas para Python como pueden ser *TensorFlow*, *Keras* o *Pytorch* entre otros. Con esto se alcanza una precisión de hasta el 99.642 %.

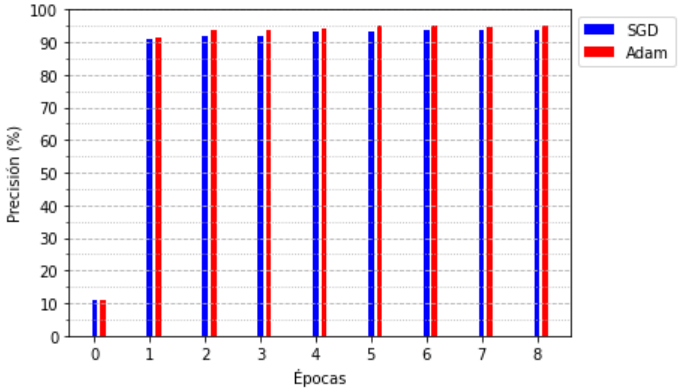


Figura 4.1: Rendimiento de la red creada con Python usando el Descenso del Gradiente Estocástico y Adam como optimizadores (código en anexo **??**).

Capítulo 5

Conclusiones y futuro trabajo

En el documento se han analizado las bases matemáticas de las redes neuronales, además de realizar un sencillo código para ponerlas en práctica. Como se ha visto, con una increíble sencillez matemática se consigue resolver problemas desafiantes como la detección de imágenes u otros más generales como mantener conversaciones lógicas fluidas como hace Chat GPT. Las redes neuronales no son más que la concatenación de funciones no lineales, sin ser estas no extremadamente complejas (convexas en una gran mayoría de aplicaciones). Sin embargo, la principal diferencia con el usual problema de optimización reside en la complejidad del coste computacional de la de minimización la función de coste por la pérdida del rastreo de la información al pasarla por la red y por la gran cantidad de parámetros que deben modelarse.

Una de los puntos claves, por tanto, consiste en el diseño de algoritmos que resuelvan el problema de minimización de la función de coste de una manera precisa a la vez que eficiente. Es decir, una de las claves reside en el optimizador escogido.

Aunque no hemos realizado un especial énfasis, el procesamiento previo del conjunto de entrenamiento es igualmente una tarea necesaria que requiere de una gran cantidad de tiempo. Se ha de asegurar la calidad y adecuación de los datos, siendo necesario realizar una limpieza -eliminación de datos ruidosos, atípicos o incorrectos-, normalización -asegurar que los datos estén en una escala adecuada y comparables entre sí- y selección de las características relevantes para el entrenamiento, logrando de esta manera una reducción de la dimensionalidad. No obstante, para ello se usan técnicas ajenas a las estudiadas, por lo que se ha decidido suponer que se dispone de un conjunto de datos previamente tratados, tal como se ha hecho en el programa de Python construido.

En otro orden de ideas, frecuentemente las redes están desprovistas de matemáticas formales que demuestren rigurosamente los resultados. En su lugar, históricamente el Machine Learning se ha basado en técnicas experimentales más bien del tipo «prueba-error-mejora» guiándose por intuiciones.

Sin embargo, hemos probado en este documento el orden de convergencia de los optimizadores vistos. En concreto, se ve como teóricamente Adam obtiene una mayor precisión en el ajuste de parámetros que aquellos optimizadores que únicamente se basan en el descenso del gradiente. En el proceso de demostración, además, gracias a los lemas desarrollados se comprueba el beneficio del empleo de los momentos del gradiente, tanto a nivel de una iteración individual así como a nivel global. Tras ello, se han estudiado los resultados experimentales obtenidos por varios equipos de trabajo. Todos ellos coinciden en la presente mejoría, tanto en la predicción como en la rapidez de la fase de entrenamiento, que esta técnica proporciona. Estos resultados además muestran problemas para los cuales la convergencia sigue la tendencia esperada según nuestros teoremas. No obstante, esto se ve limitado enormemente a las hipótesis realizadas, observando una desviación de la predicción teórica en otra cantidad de heurísticas.

Por otro lado, se ha destacado la carencia de métodos claros que orienten la elección de la arquitectura (número de capas ocultas y de neuronas así como de funciones a emplear) para la obtención de resultados óptimos. Esto mismo ocurre con la elección de hiperparámetros. No obstante, existen convenciones que sugieren una primera configuración, ajustando esta posteriormente manualmente.

En cuanto al código realizado y sus resultados, se observa como lo predicho en anteriores capítulos acerca de Adam y el DGE se satisface. En particular, el primero obtiene mejores resultados en menos iteraciones. Sin embargo, el tiempo total real de entrenamiento es superior para Adam (5,83 segundos por época frente a 3,71 segundos para DGE). Se tiene que tener en cuenta que la fase de entrenamiento se realiza una única vez, así que no tiene una verdadera relevancia al tratarse de una diferencia total de 17 segundos aproximadamente. Si además se tiene en cuenta que se puede incluso reducir el número de iteraciones a la mitad sin comprometer prácticamente la precisión para Adam, se concluye que Adam es mejor en todos los aspectos. No obstante, se verifica aquí que, pese a que el DGE no es el mejor de los optimizadores, alcanza un buen compromiso entre la ligereza computacional y precisión.

Por otro lado, probablemente una persona más experimentada sea capaz de mejorar el diseño de mi código para Adam de modo que entrene en menos tiempo; o directamente se puede conseguir con librerías como Pytorch o las anteriormente mencionadas.

Finalmente, es necesario reconocer un par de flaquezas técnicas del código. En vista a que el único desarrollador he sido yo, he obviado unas pocas *buenas prácticas* de la programación en Python, como la nomenclatura de algunas variables así como una documentación ligera de algunos métodos. Del mismo modo, no he añadido ningún `try-catch` para capturar los diferentes errores. Como estas modificaciones no afectan al rendimiento de la red, he decidido realizar el código de la manera que me fuera más cómoda.

Futuro Trabajo

Se propone el estudio de redes neuronales más especializadas para problemas concretos. Entre ellas, una de las amplias opciones son las redes convolucionales, mencionadas brevemente a lo largo del documento. Estas han obtenido grandes resultados en problemas de clasificación y detección de imágenes. En breves palabras, incorporan capas de procesamiento del input previo al feedforward, analizando características específicas (por ejemplo, en el caso de imágenes, detección de bordes verticales u horizontales). Estas capas, denominadas capas convolucionales, además permiten además disminuir la dimensionalidad del input, aliviando con ello el peso computacional subyacente en la manipulación de imágenes de alta calidad. Además, estas se pueden combinar con el uso de la inferencia vista de modo que se pueda realizar predicciones más seguras. Se puede ver una introducción con más detalle en [23].

Gracias a este tipo de redes se podría mejorar el código creado para la identificación de caracteres manuscritos. Otra posible mejora sobre el programa realizado consiste en la introducción de inferencias sobre los pesos, cambiando la red neuronal determinista por redes bayesianas; así como la inclusión de un término regularizador en la función de coste. Esto último no se ha realizado pues estamos controlando el overfitting manualmente con el número de épocas y tamaño de lote, pero siguiendo unas "buenas prácticas" se debería de incluir.

Por otro lado, se propone el estudio del manejo de datos de entrenamiento previo necesario. Aunque aquí se parta de un conjunto ya adaptado, como se viene mencionando, es uno de los núcleos pesados y estrictamente necesarios para la realización de la etapa de entrenamiento.

Bibliografía

[1] E. Negrini (2019). *Universal Approximation Theorem, G. Cybenko.* Worcester Polytechnic Institute.

[2] Y. Bengio, N. Léonard, et al (2012). *Practical recommendations for gradient-based training of deep architectures.* CoRR

[3] Y. Nesterov (2005). *Introductory Lectures on Convex Optimization. A Basic Course.* Springer

[4] S. Sudhakar (2017). *Learning Rate Scheduler.*

[5] B. Neyshabur, Z. Li et al (2018). *Towards Understanding the Role of Over-Parametrization in Generalization of Neural Networks.*

[6] S. Hochreiter, J. Schmidhuber (1997). *Long Short-term Memory* - Neural computation.

[7] X. Zhang, J. Shi et al (2018). *Understanding the Role of Biases in Neural Networks for Regularization* - IEEE.

[8] E. Goan, C. Fookes (2020). *Bayesian Neural Networks: An Introduction and Survey - Case Studies in Applied Bayesian Data Science.* Páginas 45–87. Springer.

[9] D. J. C. MacKay (2003). *Information Theory, Inference and Learning Algoritms.* Páginas 89–100. Cambrigde University Press.

[10] I. Goodfellow, Y. Bengio et al (2016). *Deep Learning (Adaptive Computation and Machine Learning series).*

[11] C. M. Bishop *Bayesian Methods for Neural Networks.*

[12] D. P. Kingma, J. Ba (2014). *Deep Learning (Adam: A Method for Stochastic Optimization.*

[13] T. Tieleman, G. Hinton (2012). *Neural networks for machine learning: Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude.*

[14] S. Bock, J. Goppold et al (2018). *An improvement of the convergence proof of the ADAM-Optimizer.*

[15] A. Défossez, L. Bottou et al (2022). *A Simple Convergence Proof of Adam and Adagrad.*

[16] J. Duchi, E. Hazan et al (2011). *Adaptive Subgradient Methods for Online Learning and Stochastic Optimization.*

[17] Stanford Exploration Project (1997). *Jensen's inequality.*

[18] F. Zou, L. Shen et al (2018). *A Unified Analysis of AdaGrad with Weighted Aggregation and Momentum Acceleration.*

[19] T. Szandała (2020) *Review and comparison of commonly used activation functions for deep neural networks.*

[20] D. Choi, C. J. Shallue, et al (2020), *On Empirical Comparisons of Optimizers for Deep Learning.*

[21] Modified National Institute of Standards and Technology - MNIST. *Base de datos de caracteres manuscritos con su correspodiente clasificación*. Acceso a la descarga.

[22] M. Nielsen (2019) *Neural Networks and Deep Learning*. - Libro online.

[23] J. Wu, LAMDA Group (2017). *Introduction to Convolutional Neural Networks*.

[24] C. M. Bishop (2006) *Pattern Recognition and Machine Learning* - Springer.

[25] S. J. Reddi, S. Kale et al (2019), *On The Convergence Of Adam And Beyond*.

[26] C. Szegedy, V. Vanhoucke et al (2016), *Rethinking the inception architecture for computer visions*.

[27] R. Meka (2019). *CS289ML: Notes on convergence of gradient descent* .

Apéndice A

Prueba a los resultados no probados

A.1. Lema de convexidad lipschitziana

Lema A.1. *Si $f : \mathbb{R}^d \to \mathbb{R}$ es una una función convexa L-lipschitziana , entonces para cualesquiera x e y en $\mathbb{R}^d$*

$$f(y) \leq f(x) + \langle \nabla f(x), y - x \rangle + \frac{L}{2}||y - x||_2^2 \tag{A.1}$$

Demostración: Usando el teorema fundamental del cálculo:

$$\begin{aligned}
f(y) &= f(x) + \int_0^1 \langle \nabla f(x + \tau(y - x)), y - x \rangle d\tau \\
&= f(x) + \langle \nabla f(x), y - x \rangle + \int_0^1 \langle \nabla f(x + \tau(y - x)) - \nabla f(x), y - x \rangle d\tau \\
&\leq f(x) + \langle \nabla f(x), y - x \rangle + \int_0^1 \|\nabla f(x + \tau(y - x)) - \nabla f(x)\|_2 \ \|y - x\|_2 d\tau \\
&\leq f(x) + \langle \nabla f(x), y - x \rangle + \int_0^1 L\|\tau(y - x)\|_2 \|y - x\|_2 d\tau \\
&= f(x) + \langle \nabla f(x), y - x \rangle + \frac{L}{2}\|y - x\|_2^2 \qquad \text{(pues } f \text{ es } L\text{-lipschitziana)}
\end{aligned} \tag{A.2}$$

■

A.2. Demostraciones de los lemas técnicos del capítulo 3

Demostración del lema 3.3

Repetimos aquí el enunciado para tener más visión:

Lema A.2. *Dado $a \in (0, 1)$ y $Q \in \mathbb{N}$:*

$$\sum_{q=0}^{Q-1} a^q \sqrt{q + 1} \leq \frac{2}{(1 - a)^{3/2}} \tag{A.3}$$

$$\sum_{q=0}^{Q-1} a^q \sqrt{q}(q+1) \leq \frac{4a}{(1-a)^{5/2}} \tag{A.4}$$

Demostración:

- En cuanto a la primera de las desigualdades, con un poco de manipulación por parte del sumatorio:

$$\begin{aligned}
(1-a)\sum_{q=0}^{Q-1} a^q\sqrt{q+1} &= \sum_{q=0}^{Q-1} a^q\sqrt{q+1} - \sum_{q=1}^{Q} a^q\sqrt{q} \\
&\leq 1 - a^Q\sqrt{Q} + \sum_{q=0}^{Q-1}\frac{a^q}{2\sqrt{q}} \leq 1 + \int_0^\infty \frac{a^x}{2\sqrt{x}}\mathrm{d}x \\
&= 1 + \frac{a^x}{2\sqrt{x}}\mathrm{d}x = 1 + \int_0^\infty \frac{\mathrm{e}^{\ln(a)x}}{2\sqrt{x}}\mathrm{d}x \quad \overset{y:=\sqrt{-2\ln(a)x}}{=} \\
&= 1 + \frac{1}{2\sqrt{-\ln a}}\int_0^\infty \mathrm{e}^{u^2/2}\mathrm{u} = 1 + \frac{\sqrt{\pi}}{2\sqrt{-\ln a}}
\end{aligned} \tag{A.5}$$

  Teniendo en cuenta $\sqrt{-\ln a} \geq \sqrt{1-a}$ y simplificando, se obtiene la cota deseada.

- Por otro lado, para la desigualdad restante, introducimos nuevamente el factor $(1-a)$:

$$\begin{aligned}
(1-a)\sum_{q=0}^{Q-1} a^q\sqrt{q}(q+1) &= \sum_{q=0}^{Q-1} a^q\sqrt{q}(q+1) - \sum_{q=1}^{Q} a^q q\sqrt{q-1} \leq \sum_{q=1}^{Q-1} a^q\sqrt{q}\left[(q+1) - \sqrt{q(q-1)}\right] \\
&\leq \sum_{q=1}^{Q-1} a^q\sqrt{q}\left[(q+1)-(q-1)\right] \leq 2\sum_{q=1}^{Q-1} a^q\sqrt{q+1} \overset{(3,38)}{\leq} \frac{4a}{(1-a)^{3/2}}
\end{aligned} \tag{A.6}$$

■

Lema 3.2 modificado

Lema A.3 (Cota sobre dirección de actualización para $\beta_1 = 0$)**.** *Para todo $n \in \mathbb{N}^*$ y para $i \in \{1, ..., d\}$:*

$$\mathbb{E}_{n-1}\left[\nabla_i F(x_{n-1})\frac{\nabla_i f_n(x_{n-1})}{\sqrt{\epsilon + v_{n,i}}}\right] \geq \frac{(\nabla_i F(x_{n-1}))^2}{2\sqrt{\epsilon+\tilde{v}_{n,i}}} - 2R\mathbb{E}_{n-1}\left[\frac{(\nabla_i f_n(x_{n-1}))^2}{\epsilon + v_{n,i}}\right] \tag{A.7}$$

Demostración: Con las definiciones de arriba (3.14), se tiene que a $i \in [d]$ fijo:

$$\begin{aligned}
\mathbb{E}_{n-1}\left[\frac{G_{n,i}g_{n,i}}{\sqrt{\epsilon+v_{n,i}}}\right] =& \mathbb{E}_{n-1}\left[\frac{G_{n,i}g_{n,i}}{\sqrt{\epsilon+\tilde{v}_{n,i}}}\right] + \mathbb{E}_{n-1}\left[G_{n,i}g_{n,i}\left(\frac{1}{\sqrt{\epsilon+v_{n,i}}} - \frac{1}{\sqrt{\epsilon+\tilde{v}_{n,i}}}\right)\right] \\
=& \frac{G_{n,i}^2}{\sqrt{\epsilon+\tilde{v}_{n,i}}} + \mathbb{E}_{n-1}\underbrace{\left[G_{n,i}g_{n,i}\left(\frac{1}{\sqrt{\epsilon+v_{n,i}}} - \frac{1}{\sqrt{\epsilon+\tilde{v}_{n,i}}}\right)\right]}_{A}
\end{aligned} \tag{A.8}$$

Manipulamos a parte el término A:

$$\begin{aligned} A &= G_{n,i}g_{n,i}\frac{\tilde{v}_{n,i}-v_{n,i}}{\sqrt{\epsilon+v_{n,i}}\sqrt{\epsilon+\tilde{v}_{n,i}}(\sqrt{\epsilon+v_{n,i}}+\sqrt{\epsilon+\tilde{v}_{n,i}})} \\ &\overset{(*)}{=} G_{n,i}g_{n,i}\frac{\mathbb{E}_{n-1}\left[g_{n,i}^2\right]-g_{n,i}^2}{\sqrt{\epsilon+v_{n,i}}\sqrt{\epsilon+\tilde{v}_{n,i}}(\sqrt{\epsilon+v_{n,i}}+\sqrt{\epsilon+\tilde{v}_{n,i}})} \end{aligned} \tag{A.9}$$

En $(*)$ se ha usado que usando la definición $\tilde{v}_{n,i}-v_{n,i} = \beta_2 v_{n,i}+\mathbb{E}[g_{n,i}^2]-v_{n,i} = \beta_2 v_{n,i}+\mathbb{E}[g_{n,i}^2]-\beta_2 v_{n,i}-g_{n,i}^2$

$$|A| \leq \underbrace{|G_{n,i}g_{n,i}|\frac{\mathbb{E}_{n-1}\left[g_{n,i}^2\right]}{\sqrt{\epsilon+v_{n,i}}(\epsilon+\tilde{v}_{n,i})}}_{\kappa} + \underbrace{|G_{n,i}g_{n,i}|\frac{g_{n,i}^2}{(\epsilon+v_{n,i})\sqrt{\epsilon+\tilde{v}_{n,i}}}}_{\rho} \tag{A.10}$$

- Por una parte, podemos acotar κ usando la desigualdad de Young:

  Escogiendo λ apropiadamente así como a y b, se llega a:

$$\lambda = \frac{\sqrt{\epsilon+\tilde{v}_{n,i}}}{2}, x = \frac{|G|_{n,i}}{\sqrt{\epsilon+\tilde{v}_{n,i}}}, y = \frac{|g|_{n,i}\mathbb{E}_{n-1}\left[g_{n,i}^2\right]}{\sqrt{\epsilon+\tilde{v}_{n,i}}\sqrt{\epsilon+v_{n,i}}} \Rightarrow \kappa \leq \frac{G_{n,i}^2}{4\sqrt{\epsilon+\tilde{v}_{n,i}}} + \frac{g_{n,i}^2\mathbb{E}_{n-1}\left[g_{n,i}^2\right]^2}{(\epsilon+\tilde{v}_{n,i})^{3/2}(\epsilon+v_{n,i})} \tag{A.11}$$

  Usando ahora que $\epsilon+\tilde{v}_{n,i} \geq \mathbb{E}_{n-1}[g_{n,i}^2]$ y tomando esperanzas condicionadas, unido a un poco de manipulación algebraica, podemos llevar a:

$$\mathbb{E}_{n-1}[\kappa] \leq \frac{G^2}{4\sqrt{\epsilon+\tilde{v}}} + R\mathbb{E}_{n-1}\left[\frac{g^2}{\epsilon+v}\right] \tag{A.12}$$

- Por otra parte, volviendo a usar la desigualdad de Young sobre ρ esta vez, podemos llegar a:

$$\lambda = \frac{\sqrt{\epsilon+\tilde{v}_{n,i}}}{2\mathbb{E}_{n-1}[g_{n,i}^2]}, x = \frac{|G_{n,i}g_{n,i}|}{\sqrt{\epsilon+\tilde{v}_{n,i}}}, y = \frac{g_{n,i}^2}{\epsilon+v_{n,i}} \Rightarrow \rho \leq \frac{G_{n,i}^2 g_{n,i}^2}{4\sqrt{\epsilon+\tilde{v}_{n,i}}\mathbb{E}_{n-1}[g_{n,i}^2]} + \frac{\mathbb{E}_{n-1}[g_{n,i}^2]g_{n,i}^4}{\sqrt{\epsilon+\tilde{v}_{n,i}}(\epsilon+v_{n,i})^2} \tag{A.13}$$

  Que usando esta vez $\epsilon+v \geq g_{n,i}^2$ más la manipulación algebraica habitual, se llega a:

$$\mathbb{E}_{n-1}[\rho] \leq \frac{G^2}{4\sqrt{\epsilon+\tilde{v}}} + R\mathbb{E}_{n-1}\left[\frac{g^2}{\epsilon+v}\right] \tag{A.14}$$

  Juntando ambos terminos y volviendo a (A.10):

$$[|A|] \leq \frac{G^2}{2\sqrt{\epsilon+\tilde{v}}} + 2R\mathbb{E}_{n-1}\left[\frac{g^2}{\epsilon+v}\right] \tag{A.15}$$

  Metiendo esto último en (A.8) se obtiene el resultado a demostrar.

■