Aplicaciones de los Métodos de Optimización Cuasi-Newton en Aprendizaje Profundo

José García Ventura

Director:
Dr. Javier García Maimó

Área de Ciencias Básicas y Ambientales
Maestría en Matemática Aplicada (MMA)
Instituto Tecnológico de Santo Domingo

Diciembre, 2022

Un conocimiento perfecto de todas las circunstancias que afectan la ocurrencia de un evento cambiaría la expectativa en certeza, y no dejaría espacio ni demanda para una teoría de probabilidades. - George Boole

Para Alan García Ventura

Resumen

La expresión analítica de la segunda derivada en el método de Newton puede volverse complicada e incluso insoluble. Los métodos numéricos para calcular la segunda derivada también son computacionalmente exigentes: si se requieren valores de N para calcular la primera derivada, se deben requerir N^2 para la segunda derivada. De este problema surgen los métodos cuasi-Newton, algoritmos de optimización que nos brindan alternativas menos computacionalmente pronunciadas al método de Newton, al no tener que realizar el cálculo directo de los elementos de segundo orden.

Dado este problema con el método de Newton, otros métodos de primer orden se han vuelto ampliamente disponibles en el aprendizaje profundo, uno de los más comunes siendo el algoritmo de descenso de gradiente. Los métodos alternativos a los de segundo orden, como los métodos cuasi-Newton, tienen capacidades particulares de generalización en la práctica, demostrando ser más efectivos computacionalmente, especialmente cuando se trabaja con grandes matrices de información.

Los avances recientes en el campo de la optimización han demostrado las aplicaciones de los métodos cuasi-Newton dentro del campo de las redes neuronales profundas. Este trabajo proporciona una comparación de estos métodos con los otros más reconocidos, se realizará sobre un problema de reconocimiento de imágenes de la base de datos MNIST y se discutirán métricas de generalización a favor de los métodos cuasi-Newton para problemas a gran escala. Se discutirán las variantes de memoria limitada asociadas a los métodos cuasi-Newton, donde se toman muestras de la matriz del gradiente asociado para facilitar las aproximaciones hessianas ante grandes volúmenes de información. Esto nos proporciona una aproximación eficiente de dicho indicador sin recurrir a calcular una inversa exacta y homogénea de la matriz hessiana.

Palabras clave: Optimización sin restricciones; Optimización numérica; Método de la Secante; Método de Newton; Métodos cuasi-Newton; Redes neuronales.

Abstract

The analytical expression of the second derivative in Newton's method can become complicated and even insoluble. Numerical methods for computing the second derivative are also computationally demanding: if N values are required to compute the first derivative, N^2 must be required for the second derivative. From this problem, quasi-Newton methods emerge, optimization algorithms that provide us with less computationally pronounced alternatives to Newton's method.

Given this problem with the analytical expression of the second derivative in Newton's method, other first-order methods have become widely available in machine and deep learning, one of the most common ones being the gradient descent algorithm. Alternative methods to second-order ones, such as quasi-Newton methods, have special generalization abilities in practice, proving to be more computationally effective, especially when working with large matrices of information.

Recent advances in the field of optimization have shown the applications of quasi-Newton methods within the field of large, deep neural networks. This work provides a demonstration of such an application and compares the methods' generalization indicators against the other best-known optimization methods within the field. The comparison will be made on an image recognition problem from the MNIST database, and the preferred generalization metrics for large-scale problems will be evidenced and discussed. The limited-memory variants associated with quasi-Newton methods will be discussed, where samples are taken from the matrix of the associated gradient to facilitate hessian approximations when faced with high volumes of information. This provides us with an efficient approximation of said indicator without resorting to the computationally expensive calculations required to obtain an exact and homogeneous inverse of the hessian matrix.

Keywords: Unrestrained optimization; Numerical optimization; Secant method; Newton's method; quasi-Newton methods; Neural networks.

Agradecimientos

A mis profesores, ya que he aprendido más de ellos a lo largo de esta maestría que de cualquier otro grupo de profesores a lo largo de mis 25 años de vida. Me han guiado y motivado a querer seguir mejorando siempre.

Índice general

Capítulo 1

Introducción

La resolución de problemas de optimización figura entre los campos de investigación más importantes de las matemáticas aplicadas, así como también de muchas otras ramas de las ciencias, la ingeniería, la informática y la economía [1]. Un vistazo a la bibliografía y la lista de grandes matemáticos que han trabajado en este campo pone de manifiesto un alto nivel de interés contemporáneo en el mismo, debido a la amplia gama de aplicaciones que tiene el campo.

Aunque el rápido desarrollo de las ciencias de la computación llevó a la aplicación efectiva de muchos métodos y algoritmos numéricos, en la realización práctica, es necesario analizar diferentes dimensiones tales como la eficiencia computacional basado en el tiempo usado por el procesador y el diseño de métodos que posean una rápida convergencia a la solución óptima ante problemas de alta complejidad y escala [2]. Dichos problemas constituyen el punto de partida de este trabajo.

Dado el rol clave que juega la optimización en campos de más recién auge como el aprendizaje profundo, se ha identificado un gran potencial en los métodos cuasi-Newton para la optimización de funciones de coste de redes neuronales y poder utilizar estos métodos como una alternativa a los optimizadores más reconocidos como Adam o RMSProp. Esto adquiere aún más relevancia dado que a medida el volumen de datos aumenta, el coste computacional de trabajar con estas funciones basándose en los métodos más convencionales también aumenta, y el incremento de este volumen de datos en los últimos años ha sido exponencial. Los métodos cuasi-Newton, al no acudir al cálculo directo de elementos de segundo orden, representan una viable alternativa para sustituir a los métodos clásicos utilizados, especialmente cuando tratamos problemas con altos volúmenes de información, ya que requieren de un menor coste computacional.

El presente trabajo de tesis de maestría ha de tratar sobre los avances recientes en los métodos de optimización cuasi-Newton, aplicados sobre el campo de las ciencias de la computación y de los datos. El mismo está compuesto de 5 capítulos, resumen de los cuales se pueden encontrar más adelante. Se presentan los antecedentes relevantes sobre optimización, con el objetivo de establecer relaciones referentes a un marco teórico-histórico que ha dado como resultado al estado del campo en la actualidad. La metodología de investigación utilizada es la documental y correlacional. La metodología documental se utiliza al construir un mapa teórico-conceptual del estado del arte de la optimización matemática, y cómo hemos llegado a los avances actuales de las aplicaciones de estos métodos basándonos en las técnicas matemáticas que han ido evolucionando. De igual modo, se emplea una metodología correlacional al realizar la aplicación y exposición de los resultados obtenidos de los algoritmos descritos en la parte documental, de manera tal que se pueda hacer un análisis de la información expuesta y establecer relaciones y diferencias sobre las etapas y metodologías que cada uno de los métodos expuestos emplean. La investigación documental ha de depender fundamentalmente de investigaciones ya realizadas con el fin de aportar un testimonio sobre una parte de la realidad actual del campo de la optimización matemática. Dicho esto, se proponen las siguientes problemáticas a ser tratadas en esta investigación:

1. La pobre resolución de problemas de optimización a gran escala mediante el reconocido método de Newton-Raphson.
2. La veracidad empírica de la superioridad de los optimizadores más famosos dentro del campo del aprendizaje profundo, como lo son Adam y/o descenso de gradiente estocástico con impulso.

En consecuencia, en el Capítulo 2 se discutirá el estado del arte de la optimización matemática, donde veremos los métodos de optimización más estudiados a la hora de aplicarlos al campo del aprendizaje profundo. Adicionalmente, es bueno notar que se ha de hacer referencia a algunos métodos más básicos de optimización sin hacer profunda alusión a las metodologías matemáticas que estos emplean. Se discutirá la teoría de las funciones cuadráticas convexas y su fundamental importancia para la comprensión del nivel de complejidad de un problema de optimización. De igual forma, se desglosarán dos de las principales estrategias a la hora de afrontar un problema de optimización: la búsqueda de línea y las llamadas regiones de confianza. La idea de empezar por aquí es esclarecer algunos conceptos básicos para luego poder estudiar métodos más avanzados que se derivan de los métodos de optimización más concurridos.

En el Capítulo 3 veremos a profundidad las particularidades de los principales métodos cuasi-Newton, qué los hacen especiales y a

qué tipo de problemas pueden aplicarse. Veremos un análisis de convergencia formal de los mismos, así como también un ejercicio de aplicación sobre una función cuadrática objetivo, de manera que se pueda apreciar el comportamiento de algunos de los métodos discutidos. De igual forma, cómo se comparan a nivel de convergencia con algunos de los métodos clásicos discutidos en el Capítulo 2.

En el Capítulo 4 veremos una breve introducción a la temática de redes neuronales y el aprendizaje profundo, y por qué son un área de reciente relevancia ante los métodos cuasi-Newton. Tocaremos los ángulos extrapolables a estos métodos de optimización haciendo hincapié en la aplicación que luego será demostrada, que son las redes neuronales convolucionales, una de las arquitecturas de redes neuronales que son las más aplicables a problemas de reconocimiento de imágenes. Veremos los principales algoritmos de optimización que se aplican a la resolución de estos problemas, así como también el recién auge de los métodos cuasi-Newton para optimizarlas.

En el Capítulo 5 veremos una aplicación de los métodos cuasi-Newton ante el contexto expuesto en el Capítulo 4 de las redes neuronales convolucionales, y esta será comparada contra los otros métodos de optimización más regularmente usados dentro de las redes neuronales también vistos en el Capítulo anterior. Se discutirá un análisis de convergencia de los métodos cuasi-Newton, teniendo como punto de comparación los métodos más utilizados normalmente como lo son Adam y el máximo descenso estocástico con impulso. Muchas de las bibliotecas orientadas a la resolución de problemas de redes neuronales tienen ya integrados los optimizadores más comúnmente usados, los mismos siendo Adam, RMSProp y el descenso de gradiente estocástico (SGD) con impulso. Sin embargo, bajo la ayuda de otros esquemas más recientes, veremos la aplicación de los métodos cuasi-Newton a la hora de compilar un modelo y optimizar su función de coste, todavía haciendo uso de los clásicos esquemas funcionales y secuenciales a nivel computacional.

Por último, en el Capítulo 6 veremos las conclusiones obtenidas tanto de la literatura estudiada como de los resultados empíricos obtenidos. Veremos también posibles nuevas direcciones de investigación que se han visto surgir de los nuevos avances de la aplicación de los métodos cuasi-Newton ante la ciencia de los datos y la computación. Las líneas de investigación planteadas se basan en el trabajo correlacional realizado, así como también basándonos en la revisión de la literatura realizada. Adicionalmente, se presenta en los Anexos (7) el código general de los algoritmos de optimización usados, así como un enlace hacia los mismos, para fines ilustrativos y de reproducción.

Capítulo 2

Estado del arte

La optimización es una herramienta importante en la ciencia de decisiones y en el análisis de sistemas físicos y matemáticos. Para hacer uso de esta herramienta, primero se identifica algún objetivo o una medida cuantitativa del desempeño del problema bajo estudio. Este objetivo puede ser la ganancia, el tiempo, la energía potencial o cualquier cantidad o combinación de cantidades que pueda representarse con un solo número. El objetivo depende de ciertas características del sistema, llamadas variables o incógnitas. Nuestra finalidad es encontrar valores de las variables que optimicen el objetivo. A menudo, las variables están restringidas o restringidas de alguna manera. Por ejemplo, cantidades como la densidad de electrones en una molécula y la tasa de interés de un préstamo no pueden ser negativas [3]. En estos contextos, el problema de optimización no puede ser implementado de modo que libremente encuentre un resultado, sino bajo una región deseada y el método debe restringirse a esta región.

Dentro de lo que tiene que ver con el proceso en sí de la optimización matemática, el proceso de identificar objetivos, variables y restricciones para un problema dado se conoce como el proceso de modelado. La construcción de un modelo apropiado es el primer paso, a veces el paso más importante, en el proceso de optimización. Una vez que se ha formulado el modelo, se puede usar un algoritmo de optimización para encontrar su solución, generalmente con la ayuda de una computadora. Es bueno notar que no existe un algoritmo de optimización universal, sino más bien una colección de algoritmos, cada uno de los cuales se adapta a un tipo particular de problema de optimización. La responsabilidad de elegir el algoritmo apropiado para una aplicación específica a menudo recae en el científico con conocimientos de optimización. Esta elección es importante, ya que puede determinar si el problema se resuelve rápida o lentamente y, de hecho, si se encuentra la solución [3].

Por otro lado, la optimización matemática es un campo de aplicación crucial dentro de campos de más reciente auge como lo son la inteligencia artificial, ya que dentro de este campo se tratan problemas de aprendizaje en donde una máquina necesita una metodología de retroalimentación para saber si esta se encuentra aprendiendo de los datos o no. Por ende, se formula un problema de optimización sobre el error del algoritmo y esto es lo que se optimiza: la función de coste. Esta es una forma muy eficiente de lograr este objetivo, ya que traerá al resultado del algoritmo a asemejarse a los datos indicados (en el caso de aprendizaje supervisado), con una mayor precisión a medida se provea al algoritmo un mayor número de iteraciones, correcciones y de datos con los cuales trabajar. Aquí entran cuestiones técnicas sobre el rendimiento del modelo (como la compensación de sesgo vs. varianza), que se abordan mediante una perspectiva de optimización, como la agregación de parámetros que regularizan y/o controlan la velocidad del ejercicio de la optimización en cada iteración. Para corregir una alta varianza se pueden fabricar datos sintéticos (en caso de imágenes se puede cambiar el ángulo a algunas ya existentes), se agregan más datos al entrenamiento, y se aplican técnicas de regularización para prevenir el sobreajuste, que es cuando existe un alto nivel de varianza dentro del modelo y por lo tanto el mismo no logra generalizar hacia datos nunca antes vistos. Otra técnica que puede contribuir con la reducción de la varianza es el conocido *dropout*, que busca evitar que un modelo se sobreajuste (como resultado de una alta varianza), mediante la eliminación de unidades ocultas en una red neuronal. Esto increíblemente da buenos resultados, especialmente en los casos donde no hay una alta equidad de clases de datos en el entrenamiento.

Una de las metodologías más acertadas al optimizar funciones de coste dentro del aprendizaje automático es el descenso de gradiente (GD), y posiblemente incluso con su contraparte cercana, el descenso de gradiente estocástico (SGD). También existen extensiones más sofisticadas, como el descenso de gradiente con impulso (momentum) y la optimización de Adam, extensión que combina las metodologías con impulso, asegurando una velocidad de búsqueda óptima [4]. Estas metodologías presentan un alto nivel de aplicación ante el campo de la inteligencia artificial, el cual se ha venido desarrollando desde el pasado siglo. Sin embargo, debido a la reciente capacidad de los ordenadores de procesar grandes volúmenes de datos, se han venido desarrollando arquitecturas informáticas que se asemejan a cómo el mismo cerebro humano procesa información. Con estas arquitecturas nos referimos a las redes neuronales, las cuales serán un tema para tratar en el Capítulo 4, y serán también la base de aplicación de este trabajo.

El campo de la optimización es bastante amplio, sin embargo, retratar una representación sucinta del mismo es bastante interesante, como veremos a continuación.

2.1. Funciones cuadráticas convexas

El concepto de convexidad es fundamental en optimización, y determina el nivel de complejidad que conlleva resolver un problema de optimización en teoría [3]. Este implica que una función, en su representación gráfica, permita unir cualquier punto de esta con una línea recta y que esta no se salga del dominio de la función. La convexidad es una propiedad importante de los conjuntos y de las funciones, ya que se puede demostrar que garantiza la existencia y unicidad de un óptimo global.

El término convexo se puede aplicar tanto a conjuntos como a funciones. Un conjunto $S \in \mathbb{R}^n$ es un conjunto convexo si el segmento de línea recta que conecta dos puntos cualesquiera en S se encuentra completamente dentro de S. Formalmente, para dos puntos cualesquiera $x \in S$ y $y \in S$, tenemos $\theta x + (1-\theta)y \in S$ para todo $\theta \in [0,1]$. La función f es una función convexa si su dominio S es un conjunto convexo y si para cualesquiera dos puntos x e y en S, se cumple la siguiente propiedad, denominada la Desigualdad de Jensen:

$$f(\theta x + (1-\theta)y) \leq \theta f(x) + (1-\theta) f(y) \tag{2.1}$$

Ejemplos simples de funciones convexas incluyen la función lineal $f(x) = c^T x + \theta$, para cualquier vector constante $c \in \mathbb{R}^n$ y escalar θ; y la función cuadrática convexa $f(x) = x^T H x$, donde H es una matriz semidefinida positiva simétrica [3].

Decimos que f es estrictamente convexa si la desigualdad en la Ecuación (2.1) es estricta siempre que $x \neq y$ y θ esté en el intervalo abierto $(0,1)$. Se dice que una función f es cóncava si $-f$ es convexa [3]. Por ende, el término opuesto a que una función sea convexa es que ésta sea cóncava. Una simple forma de comprobar que una función es convexa es verificando que su segunda derivada sea positiva. Si este es el caso, podemos asegurar que la Desigualdad de Jensen se cumple y, por tanto, que la función analizada es convexa [5].

Gráficamente, la Desigualdad de Jensen se ve de la siguiente forma, en donde se verifica la condición que la función posee la propiedad en donde se podría dibujar una línea recta entre dos cualesquiera puntos de la gráfica, y esta línea siempre formara parte del dominio de la función. Si este no fuera el caso, la función se denominaría cóncava.

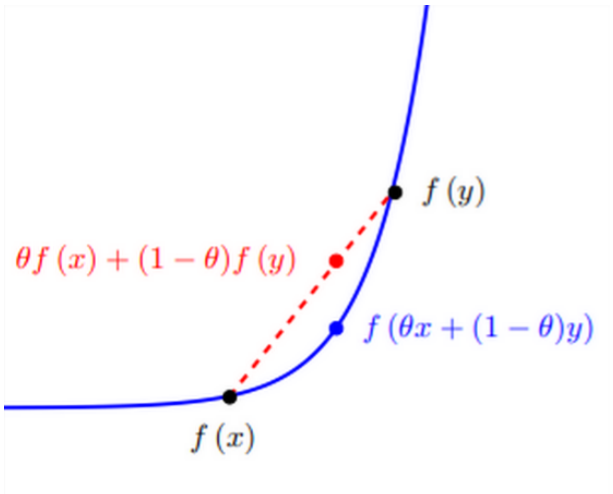

Figura 2.1: Desigualdad de Jensen

Los problemas de optimización aplicados a funciones convexas generalmente son mucho menos complejos de resolver, ya que hallar un mínimo global en una función cóncava se dificulta debido a que métodos como el descenso de gradiente pudieran detenerse en mínimos locales, fallando así en la resolución óptima del problema. Por esta razón, las funciones convexas se consideran mucho más plausibles de poder converger a una solución, y muchas veces funciones que son cóncavas simplemente no poseen solución. Desafortunadamente, muchos de los problemas de optimización de la vida real se basan sobre funciones cóncavas, además de que muchas veces también se tiene de restricciones a la solución buscada, que es donde la solución debe encontrarse bajo un definido dominio, lo que dificulta aún más la resolución del problema.

2.2. Algoritmos de optimización clásicos

Dentro de la optimización matemática sin restricciones, existen numerosos métodos de búsqueda unidimensional, multidimensional, con y sin gradiente, cerrados o abiertos. Cada uno de ellos presenta ventajas y desventajas a la hora de resolver un problema de optimización predeterminado. A esto también se le puede sumar complejidad si tenemos problemas con restricciones, y más aún si estos son no convexos (cóncavos).

Muchos algoritmos para problemas de optimización no lineal buscan solo una solución local, un punto en el que la función objetivo es más pequeña que en todos los demás puntos cercanos factibles. No siempre encuentran la solución global, que es el punto con el valor de función más bajo entre todos los puntos factibles [3]. Este punto, x^*, es el que se denomina minimizador global y generalmente es el objetivo por alcanzar dentro de un problema de minimización.

2.2.1. Método de máximo descenso

El método de máximo descenso es un ejemplo de un algoritmo iterativo con gradiente. Esto significa que el algoritmo genera una secuencia de puntos, cada uno calculado sobre la base de los puntos anteriores. El método es un método de descenso porque a medida que el algoritmo genera cada nuevo punto, el valor correspondiente de la función objetivo disminuye en valor (es decir, el algoritmo posee la propiedad de descenso).

El método de máximo descenso también se conoce como el método de Cauchy (por su autor original, Augustin-Louis Cauchy, quien propuso el método por primera vez en 1847), y es posible emplear esquemas iterativos dentro del mismo método (como por ejemplo el Método de la Bisección o alguna otra estrategia de búsqueda de línea) para también hallar dinámicamente los valores de α óptimos dentro de cada iteración de búsqueda del gradiente.

Una cuestión de interés relacionada con un algoritmo de optimización convergente local o global dado es la tasa de convergencia; es decir, qué tan rápido converge el algoritmo a un punto de solución [5]. Este método utiliza el gradiente de la función objetivo para determinar la dirección del paso a tomar en la siguiente iteración.

El gradiente es una generalización de la derivada para funciones que dependen de más de una variable. El mismo es un vector cuyo modulo representa el ritmo de variación de la función, o derivada direccional, que apunta a la dirección de máximo cambio de la función [5]. Por tanto, tomando el gradiente en negativo, $-\nabla f(x)$, obtendremos la dirección de paso más pronunciada hacia el descenso. En caso de que el problema consista en una maximización, por ende, $\nabla f(x)$ sería la dirección de máximo ascenso. En resumen, las operaciones de suma y resta del gradiente proporcionan las direcciones de paso de máximo cambio, ya sea bajo un contexto de maximización o minimización, respectivamente.

La fórmula iterativa del método de máximo descenso es representada de la siguiente forma:

$$x_{k+1} = x_k - \alpha \nabla F(x) \tag{2.2}$$

donde α representa el tamaño del paso a tomar, y $\nabla F(x)$ la dirección de máximo descenso (asumiendo un problema de minimización) sobre la función objetivo. Por consiguiente, el valor de α seleccionado en la Ecuación (2.2) para iniciar el proceso iterativo, hará que la función converja o diverja (si es demasiado grande el tamaño del paso la solución puede saltar muy lejos de la solución óptima). Para mejor visibilidad del funcionamiento de este método, ver Algoritmo 1.

Algoritmo 1 Método Descenso de Gradiente

Entrada: x_0, f, ∇f, N
Salida: x^n
$H_0 \leftarrow I$, $\epsilon \leftarrow 10^{-3}$ y $k \leftarrow 0$
$g_k \leftarrow \nabla f(x_0)$
Mientras $k < N$ y $||g_{f_k}||_2 > \epsilon$ **Hacer**

1. Definir tamaño del paso
$$\alpha_k \leftarrow \arg\min_{\alpha>0} f(x_k + \alpha g_k)$$
2. Definir nuevo punto
$$x_{k+1} \leftarrow x_k - \alpha_k g_k$$
3. $k \leftarrow k + 1$
4. Actualizar dirección de búsqueda $g_k \leftarrow \nabla f(x_k)$

Fin Mientras
Devolver x_n

2.2.2. Método de gradiente conjugado

Muchos otros algoritmos de optimización mejoran esta misma idea como lo son el método de gradiente conjugado, una aplicación de los métodos de direcciones conjugadas. El método es atribuido a Magnus R. Hestenes y Eduard Stiefel, quienes lo publicaron por primera vez en 1952. El método del gradiente conjugado es considerado como un método intermedio entre el máximo descenso y el método de Newton. Un conjunto de vectores distintos de cero $\{p_0, p_1, \ldots, p_l\}$ se dice que es conjugado con respecto a una matriz definida positiva simétrica A si se cumple que:

$$p_i^T A p_j = 0, \qquad \forall i \neq j. \tag{2.3}$$

Se dice que cualquier conjunto de vectores que satisfagan esta propiedad también es linealmente independiente [3].

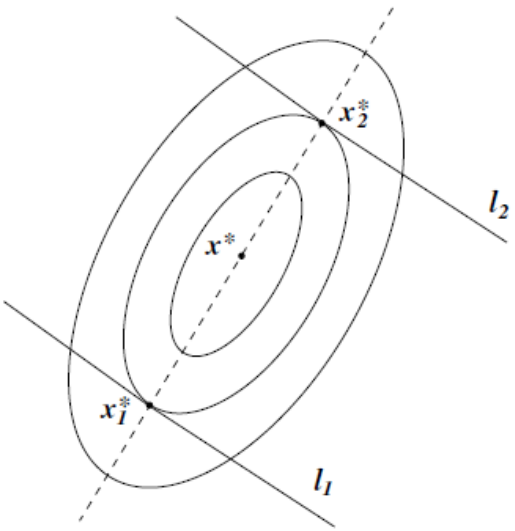

Figura 2.2: Construcción geométrica de las direcciones conjugadas.

Al generar un conjunto de vectores conjugados, se puede calcular un nuevo vector p_k usando solo el vector anterior p_{k-1}. No se necesita entonces conocer todos los elementos anteriores $p_0, p_1, \ldots, p_{k-2}$ del conjunto conjugado, ya que p_k se conjuga automáticamente a estos vectores [3]. Estas direcciones ortogonales proponen que no se debe conocer toda la información de iteraciones anteriores a la

actual, lo cual hace a este método mucho menos computacionalmente costoso que el de máximo descenso. El método del gradiente conjugado se basa en la idea de reducir el número de saltos y asegurarse de que el algoritmo nunca seleccione la misma dirección dos veces.

Por tanto, cada dirección p_k se escoge mediante una combinación entre la dirección de máximo cambio negativo $-r_k$, y la dirección tomada anterior, ρ_{k-1}, como se observa a continuación:

$$r_k = r_{k-1} + \alpha_{k-1} A\rho_{k-1}, \tag{2.4}$$

para que se cumpla la propiedad de conjugación:

$$\rho_{k-1}^T r_k = \rho_{k-1}^T r_{k-1} + \alpha_{k-1}\rho_{k-1}^T A\rho_{k-1} = 0. \tag{2.5}$$

Los métodos de Fletcher-Reeves y Polak–Ribière extienden al método de gradiente conjugado a funciones no lineales, haciendo mejoras al mismo a la hora de implementar cada tamaño de paso y dirección de búsqueda [3].

2.2.3. Método de Newton en optimización

El clásico método de Newton consiste en realizar una aproximación lineal con información de segundo orden de la función para obtener una solución aproximada del próximo paso a tomar para hallar el punto que minimiza la función objetivo.

A nivel histórico, el método de Newton a menudo se identifica como el método de Newton-Raphson. Se le acredita el descubrimiento del método a Isaac Newton en los 1660s. Luego Joseph Raphson (1690) y Thomas Simpson (1740) le agregaron modificaciones al método ya existente. La secuencia exacta es difícil de identificar, dado que Newton no publicó su descubrimiento del çálculo", habiéndose originalmente referido a esto como la teoría de fluctuaciones. Más tarde, es Leibniz quien clama haber inventado el cálculo y lo introduce al mundo en 1684. De hecho, varios métodos iterativos algebraicamente similares al método de Newton existían desde mucho antes, con la derivada siendo reemplazada por diferencias finitas o aproximaciones mediante la fórmula de la Secante [1].

La fórmula iterativa del método de Newton consiste en una generalización de una serie de Taylor multidimensional desarrollada hasta el segundo orden. Por simplicidad, consideremos una función f de una variable. Consultando la fórmula de la Serie de Taylor, sabemos que la aproximación de segundo orden de f sobre un punto $x_{k+\epsilon}$ viene dada por:

$$f(x_k + \epsilon) = f(x_k) + f'(x_k)\epsilon + \frac{1}{2}f''(x_k)\epsilon^2. \tag{2.6}$$

Nuestra aproximación de Taylor se minimiza cuando:

$$\frac{d}{d\epsilon}[f(x_k) + f'(x_k)\epsilon + \frac{1}{2}f''(x_k)\epsilon^2] = f'(x_k) + f''(x_k)\epsilon = 0,$$

lo cual se corresponde con un tamaño de paso de

$$\epsilon = -\frac{f'(x_k)}{f''(x_k)}.$$

Así podemos probar un nuevo esquema iterativo que se corresponderá con:

$$x_{k+1} = x_k - \frac{f'(x_k)}{f''(x_k)}, \tag{2.7}$$

que también tiene en cuenta el comportamiento de segundo orden de la función objetivo [4].

Generalizando a n dimensiones, la primera derivada se reemplaza por el gradiente, ∇f, y la segunda derivada se reemplaza por la hessiana, H. La hessiana de una función escalar f es una matriz cuadrada de las derivadas parciales de segundo orden de f [4].

$$H = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}$$

Con lo cual, decimos que para una función de n dimensiones, el método de Newton se rige bajo el siguiente esquema iterativo:

$$x_{k+1} = x_k - [\nabla^2 F(x_k)]^{-1} \nabla F(x_k). \tag{2.8}$$

En cada iteración k, el método de Newton aproxima la función f en el punto x con un paraboloide, y luego procede a minimizar esa aproximación al llegar al mínimo de ese paraboloide. A diferencia del descenso de gradiente regular, ya no es necesario establecer un parámetro de tasa de aprendizaje α porque ahora nuestro tamaño de paso está determinado exactamente por la distancia al mínimo de la parábola ajustada en ese punto [4].

Para ver la ventaja de calcular el hessiano de la función de pérdida, observemos el Gráfico 2.3 a continuación, comenzando desde el punto (-5, 5). Se puede apreciar cómo el método de Newton converge a la solución óptima en solamente un 2.62 % de los pasos tomados en el descenso de gradiente:

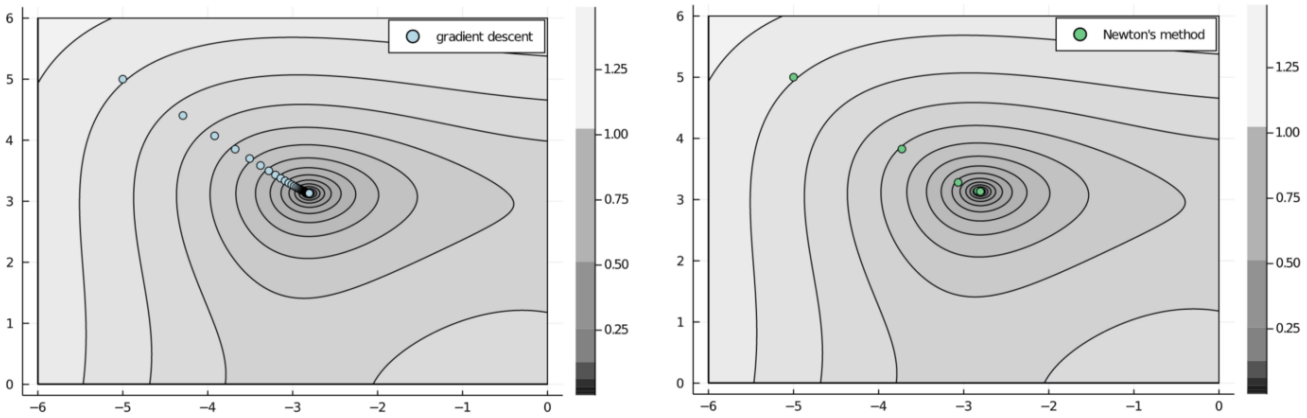

Figura 2.3: Comparativa entre el método de GD y el de Newton.

En la Figura 2.3 se puede apreciar la diferencia en términos de velocidad de convergencia entre el método de Newton y el método planteado anteriormente de máximo descenso (gradient descent). Para mejor visibilidad del funcionamiento de este método, ver Algoritmo 2. Los métodos que usan la dirección de Newton tienen una rápida tasa de convergencia local, típicamente cuadrática. Una vez que se alcanza una vecindad de la solución, la convergencia a una alta precisión a menudo ocurre en solo unas pocas iteraciones. El principal inconveniente de la dirección de Newton es la necesidad de la hessiana, $\nabla^2 f(x)$. Además, el cálculo explícito de esta matriz hessiana de segundas derivadas a veces puede ser un proceso engorroso, propenso a errores y costoso a nivel computacional. Esto último en especial si estamos trabajando con un problema no convexo.

Algoritmo 2 Método de Newton-Raphson

Entrada: x_0, f, ∇f, $\nabla^2 f$, N
Salida: x^n
$\epsilon \leftarrow 10^{-3}$ y $k \leftarrow 0$
$g_k \leftarrow \nabla f(x_0)$
$H_k \leftarrow \nabla^2 f(x_0)$
$S_k \leftarrow H_k^{-1}$
Mientras $k < N$ y $||g_{f_k}||_2 > \epsilon$ **Hacer**

1. Definir nuevo punto
$$x_{k+1} \leftarrow x_k - S_k g_k$$
2. $k \leftarrow k + 1$
3. Actualizar $g_k \leftarrow \nabla f(x_k)$, $H_k \leftarrow \nabla^2 f(x_k)$ y $S_k \leftarrow H_k^{-1}$

Fin Mientras
Devolver x_n

El algoritmo de Levenberg-Marquardt es una simple modificación al Método de Newton, el cual combina el de Cauchy y el de Newton, ayuda a proporcionar direcciones de descenso y reduce problemas de singularidad al calcular la matriz hessiana [5].

¿Por qué entonces nos molestamos en usar cualquier otro método que no sea el de Newton? Este método puede converger mucho más rápido que otros clásicos, sin embargo, no sin costo alguno. Hay dos problemas establecidos en la literatura con el método de Newton:

1. El método es extremadamente sensible a las condiciones iniciales. Esto es especialmente evidente si nuestra función objetivo no es convexa. Por lo que, si nuestra matriz hessiana no es definida positiva, podríamos tener problemas de singularidad al intentar hallar una solución óptima. Los problemas de singularidad ocurren cuando el determinante de la matriz es igual a cero. En consecuencia, la matriz se vuelve no invertible y como resultado tendríamos que recurrir a metodologías de condicionamiento, generalmente ajustando el punto inicial. A diferencia del método de máximo descenso, que asegura que siempre vamos cuesta abajo yendo siempre en la dirección opuesta al gradiente, el método de Newton ajusta un paraboloide a la curvatura local y procede a moverse al punto estacionario de ese paraboloide. Dependiendo del comportamiento local de nuestro punto inicial, el método de Newton podría llevarnos a un máximo o un punto silla en lugar de un mínimo. Este problema se exacerba en funciones no convexas de dimensiones superiores (que son los problemas usualmente planteados dentro del aprendizaje profundo), donde es mucho más probable que ocurran puntos de silla en comparación con puntos mínimos. Vemos entonces que el método de Newton realmente sólo es apropiado para minimizar funciones convexas, y fuera de este tipo de problemas es mejor recurrir a otras metodologías de optimización. Muchos problemas de optimización en el aprendizaje automático son problemas convexos, como la regresión lineal (y de cresta), la regresión logística y los Support Vector Machines. Sin embargo, un ejemplo notorio de un modelo de optimización no convexo serían las redes neuronales.
2. El segundo problema es que el método de Newton es muy costoso de implementar computacionalmente. Mientras que el cálculo del gradiente se aproxima a $O(n)$, el cálculo de la hessiana inversa se aproxima a $O(n^3)$. Por lo que, a medida que aumentan las dimensiones de nuestro problema, la sobrecarga de memoria y tiempo se sale de nuestro control muy rápidamente [4]. Esto representa una de las principales adversidades del método de Newton.

Sin embargo, es muy acertado el crédito que obtiene este método ante otros métodos con gradiente, como el de máximo descenso, y otros métodos sin gradiente como lo son el método de Powell y el método Simplex, por mencionar algunos. Y es que es posible tomar ventaja de la eficiencia ganada al considerar el comportamiento de las segundas derivadas, y al mismo tiempo evitar el coste computacional que suscita calcularlas (como lo hace el método de Newton). En otras palabras, gracias a unos métodos llamados cuasi-Newton, es posible obtener el bajo coste computacional por cada iteración que nos provee el método de máximo descenso, y al mismo tiempo lograr una tasa de convergencia mucho mejor que la del método de Newton.

Existen varios métodos de optimización cuasi-Newton propuestos en la literatura. Cada uno de ellos difiere en cómo definen y construyen las matrices cuasi-Newton ($\nabla^2 F(x_k)$ o B_k), cómo se calculan las direcciones de búsqueda y cómo se actualizan los parámetros del modelo. Sin embargo, la idea sigue siendo la misma: aproximar la matriz de elementos de segundo orden para así favorecer en la resolución de problemas a gran escala y que normalmente son altamente no convexos, basándonos en las metodologías empleadas dentro del método de Newton y solamente cambiando la forma en la que se calcula la matriz hessiana.

Capítulo 3

Métodos estructurados cuasi-Newton

Existe una multitud de algoritmos de "búsqueda de línea", para calcular la longitud de los pasos en los métodos de optimización, y representa una estrategia común para resolver problemas de optimización sin restricciones. Los problemas con restricciones suelen ser resueltos con estrategias de denominadas regiones de confianza, y que serán estrategias discutidas a continuación dentro de este Capítulo. Estas metodologías representan solamente una de las varias estrategias de optimización que puede adoptar un algoritmo. A estos métodos de búsqueda de línea se les denominan métodos sin gradiente. Estos incluyen algoritmos como el método de la Sección Dorada [6], en donde el autor originalmente proponía que el máximo de una función podía encontrarse acudiendo a intervalos de la función, sin imponer condiciones sobre la misma o calcular derivadas. El método opera estrechando sucesivamente el rango de valores en el intervalo especificado hasta encontrar un punto en donde la función llega a un extremo, lo que lo hace relativamente lento, pero muy robusto.

Dentro de lo que tiene que ver con los métodos de primer orden, existen los métodos con gradiente que incluye a los ya mencionados en el Capítulo 2, como el método de máximo descenso y el método del gradiente conjugado. Todos estos métodos y una multitud de otros no mencionados en este trabajo hacen uso de información de primer orden (del gradiente de la función) para construir la siguiente iteración de entrenamiento. Por otro lado, los métodos considerados de segundo orden utilizan información del hessiano de la función, que es la matriz de las segundas derivadas, razón por la cual cualquier método de optimización que utilice información exacta relacionada a la matriz hessiana para calcular la siguiente iteración es considerado un método de segundo orden. Dentro de esta categoría de métodos se encuentra el método de Newton, discutido de igual forma en el Capítulo 2, en donde también se veía que este método no solamente hace uso de información exacta de segundo orden de la función, sino que también procede a calcular la inversa de esta matriz hessiana, y que muchas veces esto puede implicar no solo costes computacionales demasiado altos, sino también problemas de singularidad en el cálculo de esta matriz.

A partir de esta problemática con el método de Newton surge otra categoría de métodos de optimización: los métodos cuasi-Newton. Los métodos cuasi-Newton son métodos muy eficientes para optimizar y encontrar los mínimos o máximos locales de una función sin tener que calcular la inversa de la matriz hessiana. Cualquier método que reemplace el término de la inversa de la hessiana por una aproximación de esta ya se considera un método cuasi-Newton, debido a que no se estaría usando directamente información de segundo orden de la función. Son métodos con gradiente (primer orden) y de búsqueda indirecta y se pueden usar si el jacobiano o el hessiano no están disponibles o son demasiado costosos para calcular en cada iteración. Lo que diferencia a estos métodos de los de primer orden común y corrientes es que estos realizan aproximaciones de la matriz hessiana en cada iteración, emulando de una forma particular a los métodos de segundo orden como el método de Newton. Como recordamos, el método de Newton clásico se define de la siguiente forma:

$$x_{n+1} = x_n - [\nabla^2 f(x_n)]^{-1} \nabla f(x_n), \tag{3.1}$$

donde $[\nabla^2 f(x_n)]^{-1}$ representa la inversa de la matriz hessiana de una función f evaluada en x_n [3]. La forma en la que los métodos cuasi-Newton aproximan la matriz hessiana se corresponde con una generalización del método de la Secante para encontrar la raíz de la primera derivada de problemas multidimensionales. La dirección de búsqueda sería:

$$p_k = -H_k \nabla f(x_k), \tag{3.2}$$

donde H_k es una matriz definida positiva que aproxima la inversa de la matriz hessiana.

En resumen, los métodos cuasi-Newton construyen un modelo cuadrático de la función objetivo usando aproximaciones de la matriz hessiana y utilizan ese modelo para encontrar una secuencia de direcciones de búsqueda que puede resultar en una convergencia superlineal. Dado que estos métodos no requieren derivadas de segundo orden, son más eficientes a nivel computacional que el método

de Newton para problemas de optimización a gran escala [2]. Debido a esta aproximación, los métodos cuasi-Newton suelen tener tasas de convergencia superlineales, lo que puede implicar que la convergencia se logre en una mayor cantidad de iteraciones que el método de Newton, pero logrando una mayor eficiencia computacional al no tener que calcular directamente las matrices exactas de segundo orden.

3.1. Estrategias de optimización

Existen distintas estrategias con las que abordar un problema de optimización. Dos de las estrategias más comunes son la *búsqueda de línea* y la búsqueda mediante *regiones de confianza* (trust-regions), estrategias que serán desarrolladas. Los métodos cuasi-Newton casi siempre emplean una de estas estrategias mencionadas, dependiendo del método tratado, como veremos en lo sucesivo.

3.1.1. Búsqueda de línea

La estrategia de búsqueda de línea representa una forma unidimensional de encontrar un valor óptimo a una función. Usualmente se utiliza para hallar las direcciones de paso óptimas, gobernadas por cada iterado de un algoritmo de optimización, para así no implementar este de forma determinista, sino usar alguna estrategia para hallar el tamaño óptimo de paso.

Cada iteración de un método de búsqueda de línea busca la dirección de búsqueda mínima ρ_k, asumiendo un problema de minimización. Una posible forma de hacer esto es minimizando un modelo cuadrático de la función objetivo,

$$\rho_k = \arg\min_{\rho \in \mathbb{R}^n} \mathcal{L}_k(\rho) \triangleq \frac{1}{2}\rho^T B_k \rho + \nabla f_k^T \rho, \tag{3.3}$$

y luego decide qué tanto moverse en esa dirección [2]. La iteración está dada por $w_{k+1} = w_k + \alpha_k \rho_k$, donde α_k se denomina el tamaño del paso y ρ_k la dirección de búsqueda. La elección ideal para que $\alpha_k > 0$ es el minimizador global de la función univariada $\phi(\alpha) = \mathcal{L}(w_k + \alpha\rho_k)$, pero en la práctica, la elección de α se basa en que se pueda satisfacer una mínima dirección de descenso en la función, así como que nos estemos adentrando en la dirección correcta. Estas condiciones se denominan las condiciones de Wolfe y serán expuestas a continuación.

3.1.1.1. Condiciones de Wolfe

Las condiciones de Wolfe son dos condiciones que se deben cumplir en cada iteración de los métodos cuasi-Newton y también en otros métodos de optimización, que aseguran que el α utilizado en cada iteración proporcione un descenso suficiente de la función objetivo y que asegure que los pasos sean lo suficientemente grandes para proveer de un progreso en el estado de exploración de cada iteración. Estas condiciones se denominan la condición de Armijo y la condición de curvatura, respectivamente [7]. A estas dos condiciones se les conoce como las condiciones de Wolfe y se emplean cuando se está trabajando con una estrategia de búsqueda de línea.

La condición de Armijo es una condición popular de búsqueda de línea inexacta y estipula que α_k debe primero dar una disminución suficiente en la función objetivo f, medida por la siguiente desigualdad:

$$f(x_k + \alpha\rho_k) \leq f(x_k) + c_1 \alpha \nabla f_k^T \rho_k \tag{3.4}$$

para alguna constante $c_1 \in (0, 1)$. En otras palabras, la reducción en f debería ser proporcional tanto a la longitud del paso α_k como a la derivada direccional $\nabla f_k^T \rho_k$.

La condición de disminución suficiente se ilustra en la Figura 3.1. El lado derecho de esta Figura 3.1, que es una función lineal, puede denotarse por $l(\alpha)$. La función $l(\alpha)$ tiene pendiente negativa $c_1 \nabla f_k^T \rho_k$, pero debido a que $c_1 \in (0, 1)$, se encuentra por encima de la gráfica de ϕ para valores positivos pequeños de α. La condición de disminución suficiente establece que α es aceptable solo si $\phi(\alpha) \leq l(\alpha)$. Los intervalos en los que se cumple esta condición se muestran en la Figura 3.1. En la práctica, se elige que c_1 sea bastante pequeño, digamos, $c_1 = 10^{-4}$.

La condición de disminución suficiente (o simplemente condición de Armijo) no es suficiente por sí misma para asegurar que el algoritmo tenga un progreso razonable porque, como vemos en la Figura 3.1, se satisface para todos los valores suficientemente pequeños de α [3]. Para descartar pasos inaceptablemente cortos introducimos un segundo requisito, llamada condición de curvatura, que requiere que α_k satisfaga:

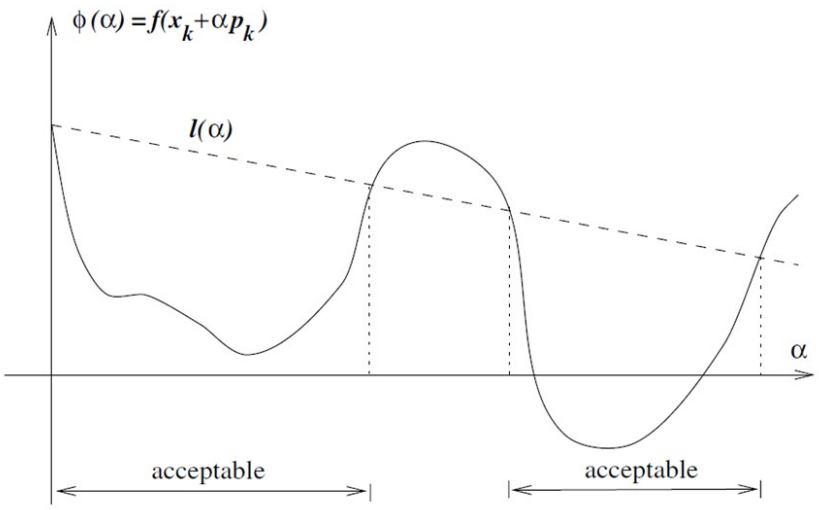

Figura 3.1: Condición de Armijo

$$\nabla f(x_k + \alpha_k \rho_k)^T \rho_k \geq c_2 \nabla f_k^T \rho_k, \tag{3.5}$$

para alguna constante $c_2 \in (c_1, 1)$, donde c_1 es la constante de la Ecuación (3.4). Es decir que, considerando a c_1 en la condición de Armijo (3.4), globalmente las condiciones de Wolfe aseguran que $0 < c_1 < c_2 < 1$ [2]. Hay que tener en cuenta que el lado izquierdo es simplemente la derivada $\phi'(\alpha_k)$, por lo que la condición de curvatura (Figura 3.2) asegura que la pendiente de ϕ en α_k sea mayor que c_2 veces la pendiente inicial $\phi'(0)$ [3]. Esto tiene sentido porque si la pendiente $\phi(\alpha_k)$ es fuertemente negativa, tenemos una indicación de que podemos reducir f significativamente si nos movemos más a lo largo de la dirección elegida.

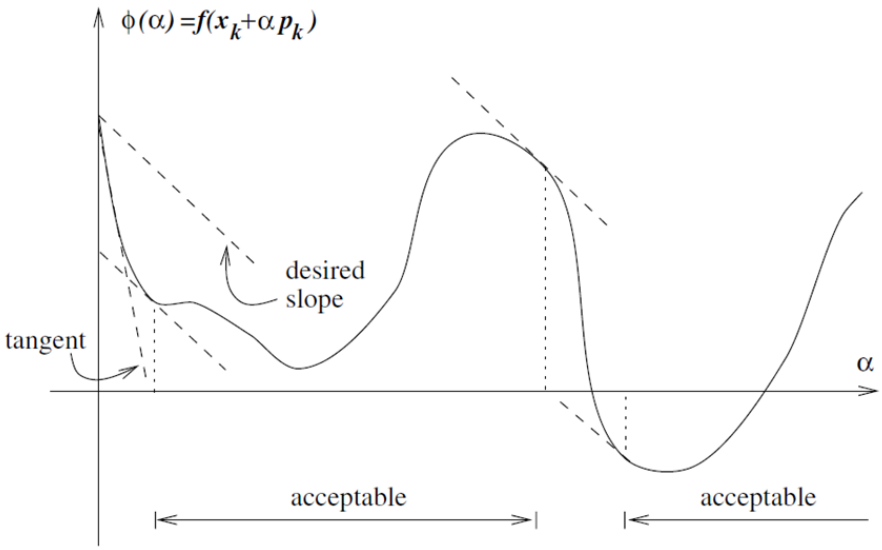

Figura 3.2: Condición de curvatura

Las condiciones de Wolfe son dos condiciones muy arraigadas a los métodos cuasi-Newton. Estas condiciones se presentan para posteriormente tener una mejor idea de a qué nos referimos cuando hablamos de las condiciones que se deben cumplir, al construir un modelo cuadrático de una función f. A nivel de implementación, el lenguaje de programación Python confiere funciones especializadas

que acuden a calcular el α óptimo en de cada iteración, específicamente de manera tal que se cumplan las condiciones de Wolfe.

3.1.2. Regiones de confianza

Las regiones de confianza son estrategias de optimización que sirven para resolver un problema restringido en el cual el siguiente tamaño de paso deberá encontrarse dentro de un determinado radio de búsqueda. Esta es una de las dos estrategias más conocidas en la optimización no lineal, siendo la otra la de búsqueda de línea. Generan una secuencia de iterados $w_{k+1} = w_k + \rho_k$, donde cada paso de búsqueda, ρ_k, es obtenido resolviendo un subproblema de optimización restringido a una región de confianza. Se crea un modelo cuadrático de la función objetivo utilizando una expansión de Taylor de segundo orden [8]:

$$\begin{aligned} \hat{m}(\rho_k) =& f_k + \rho_k^T \nabla f_k + \frac{1}{2}\rho^T B_k \rho_k, \\ \text{sujeto a} \quad & ||\rho_k||_2 \leq \triangle_k, \end{aligned} \tag{3.6}$$

donde $\triangle_k$ representa el radio de la región de confianza, ∇f_k el gradiente de la función objetivo evaluada en el centro de la región de confianza x_k y B_k la aproximación del hessiano [2]. También se selecciona ρ_k para que sea menor que igual a algún valor de radio máximo de la región de confianza $\triangle_k$. A continuación, se presenta gráficamente cómo se va construyendo la secuencia de iterados en un problema de optimización bajo una estrategia de región de confianza.

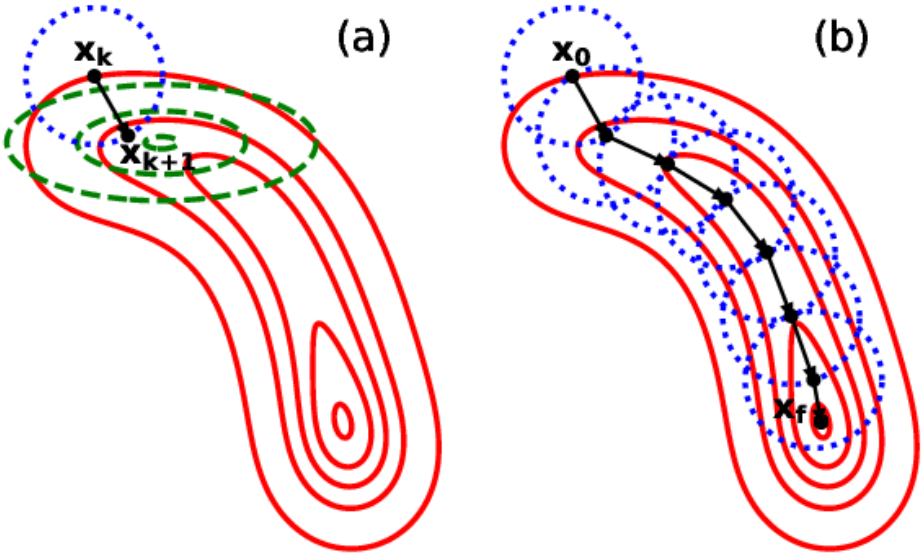

Figura 3.3: Visualización de los métodos de región de confianza.

En la Figura 3.3 se aprecia la idea a simple vista de la trayectoria global que construyen las regiones de confianza para llegar a una solución. En la parte izquierda de la Figura 3.3 (a), se observa cómo partiendo desde un punto inicial x_k, que representa el centro de la región de confianza construida sobre el modelo cuadrático (3.6), se aplica una región de confianza (círculo azul, $\triangle_k$) fuera de la cual el tamaño del paso no puede estar. La función objetivo $f(x)$ se representa en el gráfico de contorno con las líneas rojas. La zona marcada en verde representa el modelo cuadrático de la función objetivo, que es donde se construye el tamaño del paso $\hat{m}(\rho_k)$, restringido a la región de confianza.

La actualización del vector ρ_k se aplica con la Función (3.6). Mientras que, en la estrategia de búsqueda en línea, el tamaño de paso óptimo se selecciona en función de la dirección especificada, en la región de confianza, el tamaño de paso se define por el tamaño máximo de la región de confianza y solo entonces se determina la dirección [8].

El tamaño de la región de confianza es fundamental para la eficacia de todo el algoritmo, ya que las regiones más pequeñas, aunque más precisas, aumentan el número de iteraciones. Si la aproximación es lo suficientemente buena para encontrar el mínimo de la función real, la región de confianza puede aumentarse; de lo contrario, se reduce para seleccionar una dirección más precisa. Podemos validar la aptitud de la aproximación comparando la reducción real de la función y la predicha:

$$\rho_k = \frac{f_k - f(x_k + \rho_k)}{\hat{m}(0) - \hat{m}(\rho_k)}. \tag{3.7}$$

En (3.7), $\hat{m}(0)$ sería igual a f_k, lo que tiene sentido ya que el valor de la aproximación es el mismo que el valor de la función verdadera en el punto central. Si el tamaño del paso ρ_k no es positivo, entonces se debe reducir el radio de la región de confianza. Si ρ_k está cerca de 1, entonces el modelo se aproxima muy bien a la función y la región de confianza puede expandirse [8]:

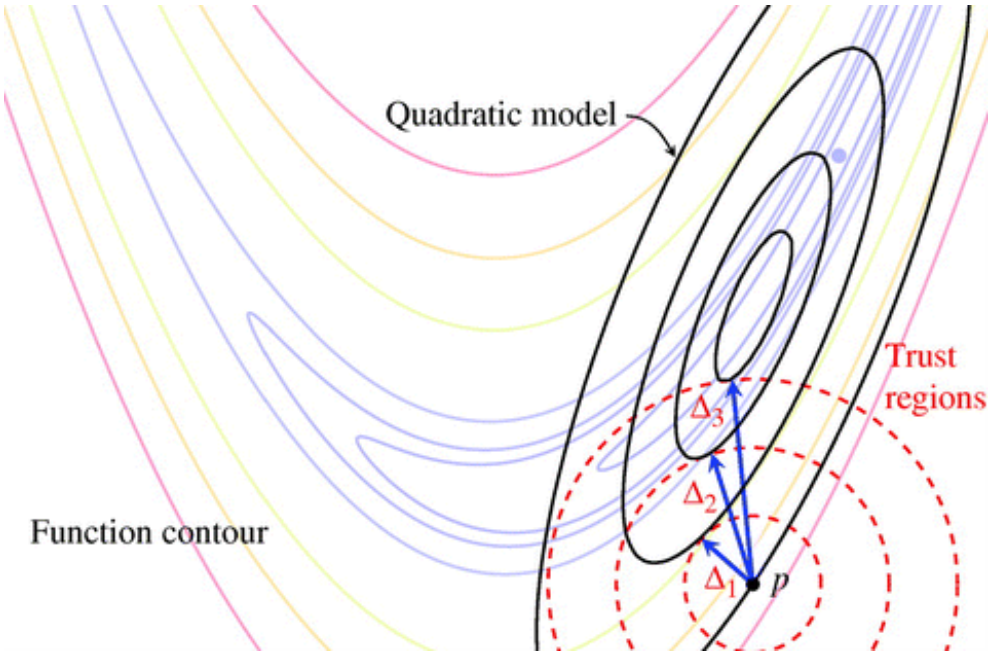

Figura 3.4: Estrategia de región de confianza.

En la Figura 3.4 se observa cómo el tamaño del radio de la región de confianza $\triangle_k$ puede variar dependiendo de cómo nos vayamos acercando al punto óptimo de la función. A medida nos vamos acercando al punto óptimo de la función objetivo x_f, tal y como se observa en la parte derecha de la Figura 3.3 (b), el punto central x_k va a ir variando en cada iteración, y por ende también el modelo cuadrático (3.6) sobre el cual se determina la región de confianza. De esto se puede inferir que en realidad el tamaño de paso ρ_k óptimo será el que en cada iteración haga que el modelo cuadrático (3.6) sea mínimo [2], bajo la restricción expuesta anteriormente:

$$\begin{aligned} \rho_k = \arg \min_{\rho_k \in \mathbb{R}^n} \hat{m}(\rho_k) &\triangleq \frac{1}{2}\rho^T B_k \rho_k + \nabla f_k^T \rho_k \\ \text{sujeto a} \quad ||\rho_k||_2 &\leq \triangle_k, \end{aligned} \tag{3.8}$$

Notar que el símbolo $\triangleq$ en (3.8) representa una definición, o queriendo decir que una expresión es igual a algo por definición, a diferencia de $:=$, que es cuando se define una expresión per se.

3.2. Método de la secante

El método de la secante es un procedimiento de búsqueda de raíces en análisis numérico que utiliza una serie de raíces de líneas secantes para aproximar mejor la raíz de una función f. Tiene la particularidad de que no involucra la derivada de una función no lineal. La fórmula de la secante es una pieza clave sobre el desarrollo de los métodos cuasi-Newton, ya que a partir de sus propiedades teóricas se basan las aproximaciones del hessiano en cada iteración [3].

La fórmula iterativa del método de la secante es la siguiente:

$$y_k = B_{k+1} s_k.$$

Esta relación surge al desarrollar sobre una serie de Taylor un modelo cuadrático de una función objetivo en el actual iterado x_k:

$$m_k(\rho) = f_k + \nabla f_k^T \rho + \frac{1}{2}\rho^T B_k \rho.$$

Aquí B_k es una matriz definida positiva simétrica $n \times n$ que se actualizará en cada iteración. El valor de la función y el gradiente de este modelo en $\rho = 0$ coinciden con f_k y ∇f_k, respectivamente. El minimizador ρ_k de este modelo cuadrático convexo, que podemos escribir explícitamente como

$$\rho_k = -B_k^{-1} \nabla f_k,$$

se utiliza como la dirección de búsqueda, y la nueva iteración sería entonces

$$x_{k+1} = x_k + \alpha_k \rho_k,$$

donde la longitud de paso α_k se elige de manera tal que se satisfagan las condiciones de Wolfe [3]. Esta iteración es bastante similar al método Newton de búsqueda de línea; la diferencia clave es que se usa el hessiano B_k aproximado en lugar del hessiano verdadero.

En lugar de calcular B_k de nuevo en cada iteración, Davidon (uno de los pioneros en el diseño de los métodos cuasi-Newton) propuso actualizarlo de manera sencilla para tener en cuenta la curvatura medida durante el paso más reciente [9]. Supongamos que hemos generado una nueva iteración x_{k+1} y deseamos construir un nuevo modelo cuadrático, de la forma

$$m_{k+1}(\rho) = f_{k+1} + \nabla f_{k+1}^T \rho + \frac{1}{2}\rho^T B_{k+1} \rho.$$

Un requisito propuesto por Davidon, que se debe imponer a B_{k+1}, según el conocimiento adquirido durante el último paso, es que el gradiente de m_{k+1} debe coincidir con el gradiente de la función objetivo f en las últimas dos iteraciones x_k y x_{k+1}. Dado que $\nabla m_{k+1}(0)$ es precisamente ∇f_{k+1}, la segunda de estas condiciones se cumple automáticamente. La primera condición se puede escribir matemáticamente como

$$\nabla m_{k+1}(-\alpha_k \rho_k) = \nabla k_{k+1} - \alpha_k B_{k+1} \rho_k = \nabla f_k$$

Reordenando obtenemos:

$$B_{k+1} \alpha_k \rho_k = \nabla f_{k+1} - \nabla f_k \tag{3.9}$$

Y para simplificar la notación, se implementan los vectores:

$$s_k = x_{k+1} - x_k = \alpha_k \rho_k, \qquad y_k = \nabla f_{k+1} - \nabla f_k.$$

De esta forma, a partir de (3.9), obtenemos la ecuación de la secante [3],

$$B_{k+1} s_k = y_k. \tag{3.10}$$

3.3. Fórmula Simétrica de Rango 1 (SR1)

Uno de los algoritmos más simples entre los denominados cuasi-Newton es el que emplea de la fórmula SR1 (Symmetric Rank 1). En este método, cada sucesiva iteración de la matriz hessiana difiere de su predecesora por una matriz de rango 1 [3]. En consecuencia, el término de corrección es simétrico y tiene la forma:

$$H_{k+1} = H_k + \alpha_k z^{(k)} z^{(k)T}. \tag{3.11}$$

Notar que

$$\operatorname{rank} z^{(k)} z^{(k)T} = \operatorname{rank}\left(\begin{bmatrix} z_1^{(k)} \\ \vdots \\ z_n^{(k)} \end{bmatrix} [z_1^{(k)} \dots z_n^{(k)}] \right) = 1,$$

y debido a esto el nombre de corrección de rango uno (también se denomina algoritmo simétrico de rango único) [5].

Nuestra meta en SR1 consiste en determinar α_k y $z^{(k)}$, tal que dados H_k, $\nabla g^{(k)}$, y $\nabla x^{(k)}$ se satisfaga:

$$H_{k+1} \nabla g^{(k)} = (H_k + \alpha_k z^{(k)} z^{(k)T}) \nabla g^{(k)} = \nabla x^{(k)}. \tag{3.12}$$

Partiendo de aquí se puede determinar la fórmula iterativa de actualización de la aproximación de la inversa de la matriz hessiana que hace cumplir la ecuación de la secante como:

$$H_{k+1} = H_k + \frac{(s_k - H_k y_k)(s_k - H_k y_k)^T}{(s_k - H_k y_k)^T y_k}. \tag{3.13}$$

La fórmula de actualización SR1 ha demostrado ser bastante útil, y su capacidad para generar aproximaciones hessianas indefinidas puede considerarse una de sus principales ventajas. El principal inconveniente de SR1 es que el denominador en (3.13) puede tender a cero. De hecho, incluso cuando la función objetivo es cuadrática y convexa, puede haber pasos en los que simplemente no haya una actualización simétrica de rango 1 que satisfaga la Ecuación (3.10) [3]. Esto puede considerarse como uno de los principales problemas del método cuasi-Newton, SR1.

Como sugiere su nombre, la fórmula de actualización simétrica de rango 1 (SR1) permite satisfacer la ecuación secante y mantener la simetría con una actualización de rango 1 más simple. Sin embargo, a diferencia de BFGS, la actualización SR1 no garantiza que la matriz actualizada mantenga una definición positiva. Conforme a esto, el método SR1 generalmente se implementa con una estrategia de región de confianza [10]. Sin embargo, en la implementación de ejemplo ilustrativo que se hace a continuación el método es aplicado bajo un esquema iterativo de búsqueda de línea (para el tamaño del paso).

En consecuencia, en cada iteración SR1 calcula el nuevo iterado con la fórmula:

$$w_{k+1} = w_k + \rho_k, \tag{3.14}$$

donde ρ_k es el minimizador del siguiente problema de optimización restringido:

$$\begin{aligned} \min_{\rho} m_k(\rho) =& F(w_k) + \nabla F(w_k)^T \rho + \frac{1}{2} \rho^T B_k \rho, \\ \text{s.a.} \quad & ||\rho|| \leq \triangle_k, \end{aligned} \tag{3.15}$$

donde $\triangle_k$ es la región de confianza, y B_k es la aproximación del hessiano correspondiente a la fórmula de actualización SR1 indicada en (3.13) [10].

Otro inconveniente de SR1 es que la matriz H_{k+1} puede ser no definida positiva, y por consiguiente d^{k+1}, o $-H \nabla f_k^{(1)}$, no sería una dirección de descenso. Esto puede pasar aún en el caso de una función cuadrática. Adicionalmente, si cada evaluación del modelo cuadrático es cercana a cero, entonces podríamos tener problemas numéricos al evaluar H_{k+1} [5]. Para mejor visibilidad del funcionamiento de este método, ver Algoritmo 3 en la siguiente Sección.

Afortunadamente, existen mejoras a esta aproximación de rango 1, que igualmente aplican el método de la secante para alinear las evaluaciones H_{k+1}. Estos son los algoritmos DFP y BFGS, que son normalmente los dos métodos cuasi-Newton más conocidos.

3.4. Algoritmo Davidon–Fletcher–Powell (DFP)

Un famoso método cuasi-Newton fue creado por Davidon, más tarde Fletcher y Powell, y se denomina la fórmula Davidon–Fletcher–Powell (DFP) [9]. Davidon en el año 1959 publicó un método para determinar numéricamente mínimos locales de

Algoritmo 3 Método SR1

Entrada: $x_0, f, \nabla f, N$
Salida: x^n
$H_0 \leftarrow I, \epsilon \leftarrow 10^{-3}$ y $k \leftarrow 0$
$\nabla f(x_k) \leftarrow \nabla f(x_0)$
Mientras $k < N$ y $||\nabla f_k||_2 > \epsilon$ **Hacer**

1. Definir la dirección de búsqueda $\rho_k \leftarrow -H\nabla f_k^{(k)}$ y tamaño del paso
$$\alpha_k \leftarrow \arg\min_{\alpha>0} f(x_k + \alpha\rho_k)$$
2. $x_{k+1} \leftarrow x_k + \alpha_k\rho_k$
3. $\nabla f(x_{k+1}) \leftarrow \nabla f(x_{k+1})$
4. Definir $S_k \leftarrow x_{k+1} - x_k$ y $y_k \leftarrow \nabla f(x_{k+1}) - \nabla f(x_k)$
5. Actualizar $H_{k+1}^{SR1} \leftarrow H_k + \frac{(s_k - H_k y_k)(s_k - H_k y_k)^T}{(s_k - H_k y_k)^T y_k}$
6. $k \leftarrow k + 1$

Fin Mientras
Devolver x_n

funciones derivables de varias variables (multidimensional). En el proceso de ubicar cada mínimo, se determina una matriz que caracteriza el comportamiento de la función respecto al mínimo. Para una región en la que la función depende cuadráticamente de las variables, no se requieren más de N iteraciones, donde N es el número de variables. Mediante la elección adecuada de los valores iniciales y sin modificar el procedimiento, se pueden imponer restricciones lineales a las variables. Las aproximaciones del hessiano en DFP, a diferencia de SR1, presentan una diferencia entre sí de una matriz de rango dos.

Luego de la publicación de 1959, en 1963 Fletcher y Powell [11] demostraron que el algoritmo de rango 2 creado por Davidon era mucho más rápido y confiable que los ya existentes, y por eso se le denomina al algoritmo DFP (en honor a sus autores). Se hace una aproximación cuadrática de la función objetivo en la iteración actual x_k para así capturar la información de curvatura [7]. Tal aproximación viene dada por:

$$m_k(\rho) = f_k + \nabla f_k^T \rho + \frac{1}{2}\rho^T B_k \rho,$$

donde B_k es una matriz simétrica y definida positiva de tamaño $n \times n$, la cual se actualiza en cada iteración [7]. Esta B_k es la que representa la aproximación de la matriz hessiana, que luego pasa a ser insumo de la ecuación de Sherman-Morrison para estimar su inversa, ecuación que viene dada por $(A+UV)^{-1} = A^{-1} - A^{-1}U(I+V^TA^{-1}U)^{-1}VA^{-1}$ [12]. La ecuación de Sherman-Morrison aplica correcciones a la matriz original para invertirla evitando el coste computacional de calcularla directamente. Esta ecuación se usa en los métodos cuasi Newton para aproximar la inversa de la aproximación de la matriz hessiana que se obtiene con el algoritmo DFP. De esta forma, se evitan cálculos explícitos de la matriz hessiana inversa, ahorrando tiempo considerable a nivel computacional.

En DFP, buscando B_{k+1}, debemos pedir la condición de que entre todas las matrices simétricas que cumplen con la ecuación secante, $B_{k+1}s_k = y_k$, B_{k+1} debe ser la más cercana a la matriz actual B_k, es decir, se debe resolver el problema:

$$\begin{aligned} &\min_B ||B - B_k||_F \\ \text{s.a.}\quad &B = B^T \\ &Bs_k = y_k, \end{aligned} \tag{3.16}$$

encontrando así la única matriz simétrica de rango uno que cumpla la ecuación de la secante [7].

Partiendo del hecho de que la matriz B_k es definida positiva, por la ecuación de la secante se tiene que

$$S_k^T B_{k+1} S_k = S_k^T y_k > 0.$$

A esto es lo que se le conoce como la condición de curvatura que vimos anteriormente (Figura 3.2). Esta condición no siempre será cierta cuando la función no sea convexa, en este caso tendríamos que forzar a que la desigualdad anteriormente expuesta sea verdadera, lo cual se logra imponiendo las condiciones de Wolfe o las condiciones fuertes de Wolfe en α_k [7]. Por lo que las condiciones de Wolfe han de ser especialmente importantes cuando estemos trabajando con funciones altamente no convexas y donde se maneje un gran volumen de parámetros, que suelen ser los problemas manejados por las redes neuronales. A nivel de implementación esto pudiera significar un problema, ya que forzar una condición de Wolfe que no se está cumpliendo directamente puede hacer el algoritmo bastante lento. Por eso a veces funciona mejor a nivel práctico hacer uso de una estrategia de región de confianza.

La fórmula iterativa de actualización de la matriz hessiana del algoritmo DFP es [13],

$$B_{k+1} = B_k - \frac{1}{s_k^T B_k s_k} B_k s_k s_k^T B_k + \frac{1}{y_k^T s_k} y_k y_k^T + (s_k^T B_k s_k) w_k w_k^T, \tag{3.17}$$

donde:

$$w_k = \frac{1}{y_k^T s_k} y_k - \frac{1}{s_k^T B_k s_k} B_k s_k.$$

La razón detrás de esta aproximación (que no requiere derivadas explícitas de segundo orden) está en que ese hessiano tiende a ser más estable numéricamente porque los términos de segundo orden son más sensibles al ruido. El método de Davidon–Fletcher–Powell (DFP) encuentra la solución de la ecuación secante más cercana a la estimación actual y satisface la condición de curvatura. Fue el primer método cuasi-Newton en generalizar el método de la secante a un problema multidimensional. Esta actualización mantiene la simetría y la definición positiva de la matriz Hessiana. Sin embargo, DFP presenta dificultades cuando se tienen problemas en donde B se acerca a ser singular (especialmente si se trata de un problema a gran escala). Veremos a continuación mejoras que se han diseñado a partir de DFP y SR1.

3.5. Algoritmo Broyden-Fletcher-Goldfarb-Shanno (BFGS)

Quizá el algoritmo más conocido de los cuasi-Newton es el BFGS. Este fue desarrollado en los años 70 por sus 4 autores que trabajaron de forma independiente (Broyden [14], Fletcher [15], Goldfarb [16] y Shanno [17]) sobre el manuscrito que ya existía del algoritmo DFP (Davidon–Fletcher–Powell). La fórmula de actualización BFGS puede ser derivada haciendo un simple cambio en el argumento de la fórmula DFP, ya que la única diferencia entre estos dos algoritmos es que en vez de aproximar la matriz hessiana se parte directamente a aproximar la inversa de ella. Es decir, en lugar de imponer condiciones en las aproximaciones del hessiano B_k, imponemos condiciones sobre el inverso del hessiano H_k, invirtiendo así el planteamiento de la ecuación de la secante en DFP. Estas aproximaciones deben ser simétricas, definidas positivas y satisfacer la ecuación de la secante:

$$H_{k+1} y_k = s_k.$$

De forma análoga a DFP, se tiene:

$$\begin{aligned} &\min_H ||H - H_k|| \\ \text{s.a.} \quad &H = H^T \\ &H y_k = s_k, \end{aligned} \tag{3.18}$$

donde H_k representa la aproximación de la inversa de la matriz hessiana. [7]

La solución para H_{k+1} sería entonces:

$$H_{k+1}^{BFGS} = (I - \rho_k s_k y_k^T) H_k (I - \rho_k y_k s_k^T) + \rho_k s_k s_k^T,$$

donde

$$\rho_k = \frac{1}{y_k^T s_k}$$

Desarrollando, obtenemos la fórmula de actualización BFGS:

$$H_{k+1}^{BFGS} = H_k + \left(1 + \frac{y_k^T H_K y_k}{s_k^T y_k}\right)\frac{s_k s_k^T}{s_k^T y_k} - \left(\frac{s_k y_k^T H_k + H_k y_k s_k^T}{s_k^T y_k}\right) \tag{3.19}$$

A nivel de implementación, dado que una actualización de B^{-1}, depende de su valor anterior, tenemos que inicializar B^{-1} para iniciar el algoritmo. Notar que aquí ya nos referimos a H al hablar de B^{-1}. Hay dos formas naturales de hacer esto. El primer enfoque es igualar B_0^{-1} a la matriz de identidad, en cuyo caso el primer paso será equivalente al descenso de gradiente simple, y las actualizaciones posteriores lo refinarán de forma incremental para acercarse a la hessiana inversa, B^{-1}. Otro enfoque sería calcular e invertir la hessiana exacta únicamente en el punto inicial. Esto iniciaría el algoritmo con pasos más eficientes, pero tiene un costo inicial de calcular la hessiana verdadera e invertirla [4].

Notar que, como ya hemos mencionado, BFGS realmente está construyendo un modelo cuadrático de la función objetivo para así ir determinando los términos de segundo orden. En otras palabras, BFGS usa el impulso o *momentum* implícitamente incorporado en las matrices B_k.

La minimización convencional BFGS emplea una búsqueda de línea, que primero intenta encontrar las direcciones de búsqueda calculando $\rho_k = -B_k^{-1}\nabla\mathcal{L}(w_k)$ y luego decide el tamaño del paso $\alpha_k \in (0, 1]$ basado en condiciones de disminución y curvatura suficientes (las condiciones de Wolfe) para cada iteración k. Luego actualiza los parámetros $w_{k+1} = w_k + \alpha_k\rho_k$. El algoritmo de búsqueda de línea prueba primero la longitud de paso unitario $\alpha_k = 1$ y si no satisface las condiciones de disminución suficiente y de curvatura, reduce recursivamente α_k hasta algún criterio de parada (por ejemplo $\alpha_k < 0.1$) [2].

Notar que ambos métodos DFP y BFGS, a diferencia de SR1, poseen una diferencia con la predecesora aproximación de la hessiana de una matriz de rango 2. También, dado que en BFGS no es necesario calcular la inversa de la matriz hessiana luego de que esta se aproxima (en DFP se aplica la fórmula de Sherman-Morrison para calcular la inversa), el nivel de complejidad computacional solo alcanza O^2 en lugar de O^3 como es el caso de DFP. Para mejor visibilidad del funcionamiento de este método, ver Algoritmo 4.

Algoritmo 4 Método BFGS

Entrada: x_0, f, ∇f, N
Salida: x^n
$H_0 \leftarrow I$, $\epsilon \leftarrow 10^{-3}$ y $k \leftarrow 0$
$\nabla f(x_k) \leftarrow \nabla f(x_0)$
Mientras $k < N$ y $||\nabla f_k||_2 > \epsilon$ **Hacer**

1. Definir la dirección de búsqueda $\rho_k \leftarrow -H_k\nabla f_k^{(k)}$ y tamaño del paso
$$\alpha_k \leftarrow \arg\min_{\alpha>0} f(x_k + \alpha\rho_k)$$
2. $x_{k+1} \leftarrow x_k + \alpha_k\rho_k$
3. $\nabla f_k^{(k)} \leftarrow \nabla f(x_{k+1})$
4. Definir $S_k \leftarrow x_{k+1} - x_k$ y $y_k \leftarrow \nabla f(x_{k+1}) - \nabla f(x_k)$
5. Actualizar $H_{k+1}^{BFGS} \leftarrow H_k + \left(1 + \frac{y_k^T H_K y_k}{s_k^T y_k}\right)\frac{s_k s_k^T}{s_k^T y_k} - \left(\frac{s_k y_k^T H_k + H_k y_k s_k^T}{s_k^T y_k}\right)$
6. $k \leftarrow k + 1$

Fin Mientras
Devolver x_n

3.5.1. Algoritmo L-BFGS

La fórmula de BFGS se puede escribir de la siguiente forma compacta:

$$B_k = B_0 + \psi_k M_k \psi_k^T,$$

donde ψ_k y M_k son definidos como

$$\psi_k = \begin{bmatrix} B_0 S_k & Y_k \end{bmatrix}, \qquad M_k = \begin{bmatrix} -S_k^T B_0 S_k & -L_k \\ -L_k^T & D_k \end{bmatrix}^{-1},$$

y L_k es la parte triangular estrictamente inferior y D_k es la parte diagonal de la matriz $S_k^T Y_k$, es decir, $S_k^T Y_k = L_k + D_k + U_k$, donde U_k es una matriz triangular estrictamente superior [2]. Por cada iteración se busca una aproximación de la matriz hessiana, más específicamente de su inversa. Si la función tiene N variables, la matriz hessiana tiene N^2 elementos. Cuando N es grande, el tiempo necesario para calcular toda la matriz de forma exacta puede ser inviable.

En problemas de gran escala, es común almacenar solo los m pares calculados más recientemente (s_k, y_k), donde normalmente $m \leq 100$. Este enfoque a menudo se denomina BFGS de memoria limitada (L-BFGS) [18]. Aquí L simboliza el parámetro de *limited memory* (memoria limitada). Como su nombre indica, L-BFGS se basa en la fórmula de actualización BFGS (3.19).

Tal como en BFGS, existe una versión de memoria limitada para SR1 (L-SR1), en donde la matriz B_k es definida en cada iteración como resultado de aplicar m actualizaciones a múltiples pares de la matriz identidad, usando m pares de correcciones s_i, y_i, y guardándolos en la memoria [10].

Los métodos cuasi-Newton de memoria limitada son útiles para resolver problemas grandes cuyas matrices hessianas no se pueden calcular a un costo razonable. Estos métodos mantienen aproximaciones simples y compactas de matrices hessianas: en lugar de almacenar aproximaciones $n \times n$ completamente densas, guardan solo unos pocos vectores de longitud m que representan las aproximaciones del hessiano implícitamente. Por lo tanto, el almacenamiento requerido es $O(mn)$ en vez de $O(n^2)$. A pesar de estos modestos requisitos de almacenamiento, a menudo producen una tasa de convergencia aceptable (aunque lineal). Se han propuesto varios métodos de memoria limitada, solo uno de estos es el denominado L-BFGS. La idea principal de este método es usar información de curvatura de sólo las iteraciones más recientes para construir la aproximación hessiana basándonos sólo en esta información. La información de curvatura de iteraciones anteriores, que es menos probable que sea relevante para el comportamiento real de hessiano en la iteración actual, se descarta con el fin de ahorrar almacenamiento y tiempo computacional [3].

3.6. Convergencia de los métodos cuasi-Newton

El método de Newton es uno de los más exitosos a la hora de resolver problemas de optimización. Sin embargo, este puede ser ineficiente a la hora de calcular la inversa de la matriz hessiana cuando la misma tiene una alta dimensión. Por otro lado, la forma directa de cálculo que posee el método de Newton le provee al mismo de una fácil afirmación de que este posee una convergencia cuadrática (lineal, dentro de lo que puede resolver).

A modo ilustrativo y comparativo, consideremos el siguiente problema de optimización sin restricciones de dos dimensiones. La función que tomaremos de prueba se le reconoce como la función multimodal de Himmelblau [19], nombre otorgado en virtud de su creador, el ingeniero químico David Mautner Himmelblau (1924–2011).

$$f(x, y) = (x^2 + y - 11)^2 + (x + y^2 - 7)^2. \tag{3.20}$$

Es posible atribuirle a esta función cuatro puntos de mínimos de forma analítica, estos serían:

1. $f(3.0, 2.0) = 0$
2. $f(-2.8051, 3.1313) = 0$
3. $f(-3.7793, -3.2831) = 0.$
4. $f(3.5844, -1.8481) = 0$

Las combinaciones de puntos fueron obtenidas utilizando el software matemático MATLAB, haciendo una búsqueda de todos los puntos en donde el gradiente de (3.20) se hace cero, obteniendo así el conjunto de los cuatro extremos que minimizan la función. Los códigos en MATLAB se pueden encontrar en el Anexo (Capítulo 7). Inicializamos un número máximo de iteraciones de 5000, y con un mismo punto inicial $x = [1.80, -3.36]$ para cada uno de los cuatro métodos probados. Tomando este planteamiento inicial del problema, se obtuvieron los siguientes resultados.

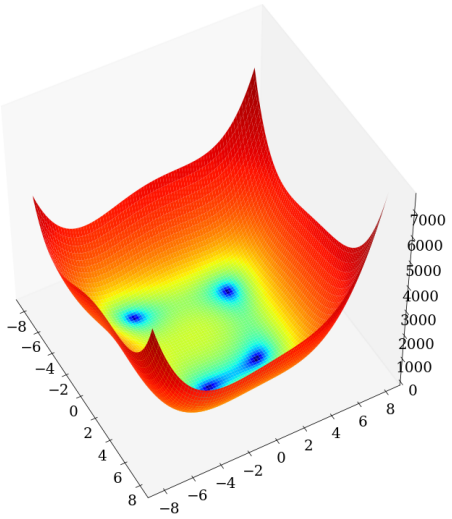

Figura 3.5: Función de Himmelblau

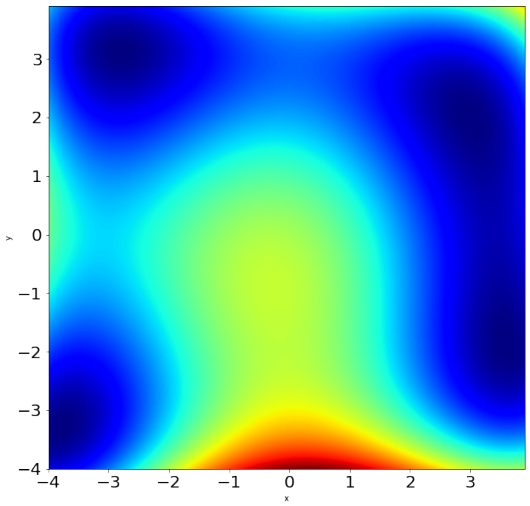

Figura 3.6: Gráfico de contorno

Los dos gráficos anteriores muestran con colores más fríos los valores en donde la función se hace mínima. La idea es plasmar la comparación entre dos de los métodos cuasi-Newton que hemos tratado, contra los métodos de Newton y de máximo descenso, a fin de que se pueda ilustrar la diferencia a nivel de eficiencia computacional que brindan estos métodos.

A continuación, se muestra una tabla en donde se visualizan los resultados obtenidos al minimizar la Función (3.20), haciendo uso del método de Newton, el método del descenso del gradiente y dos de los métodos cuasi-Newton.

Algoritmo	**Cantidad de iteraciones**	**Norma obtenida**
Método Newton	7	0.00008
Método Descenso de Gradiente	21	0.00767
Método BFGS	8	0.00337
Método SR1	9	0.00020

Tabla 3.1: Análisis de la convergencia.

Todos estos algoritmos cuyos resultados se muestran en la Tabla 3.1 parten desde un mismo punto inicial, $x = [1.80, -3.36]$. Se muestran las cantidades de iteraciones de cada algoritmo, así como también la norma vectorial del último gradiente obtenido al final de las iteraciones. Es relevante indicar que tanto el método de Newton como el de descenso de gradiente son bastante sensibles al punto inicial que se escoge, y dependiendo de este punto inicial escogido la convergencia del método puede tanto dirigirse hacia un mínimo distinto como hacer que el método diverja. Adicional a esto, teniendo ambos métodos cuasi-Newton los mismos puntos iniciales, el comportamiento en términos de pasos para llegar a la convergencia es relativamente el mismo al del método de Newton, obteniendo sobre esta misma equivalencia del punto inicial un ahorro computacional que se torna exponencial al trabajar con problemas de mayor escala. Sin embargo, este ahorro computacional arroja una norma final del vector del gradiente superior a la del método de Newton. De esto podemos inferir que la convergencia cuadrática del método de Newton nos provee tanto de un número de iteraciones menor, pero también de un nivel de exactitud mayor.

A continuación, se muestra una representación gráfica de los resultados, en donde se puede ver la trayectoria tomada por cada algoritmo, así como también la solución obtenida. De igual forma, gráficamente se presentan los cuatro mínimos obtenidos haciendo uso de la función de MATLAB *solve*, aplicado al gradiente de la función igualado a cero. Es interesante ver cómo el método de Newton empieza a tomar una trayectoria que pareciera va a divergir de alguno de los puntos de mínimos, sin embargo, la dirección de la trayectoria da un giro de 180 grados en el extremo superior izquierdo de la Figura 3.7, para luego empezar a acercarse a uno de los puntos de mínimos, aunque sea este uno distinto a la elección del resto de métodos probados.

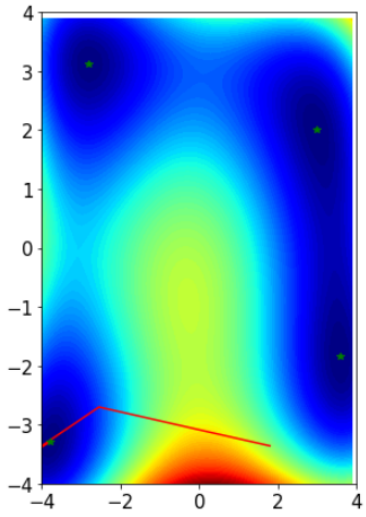

Figura 3.7: Método de Newton

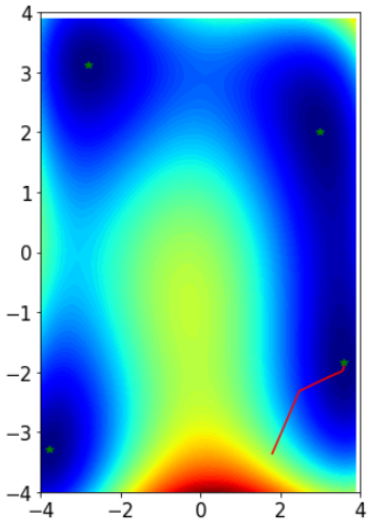

Figura 3.8: Método del Descenso de Gradiente

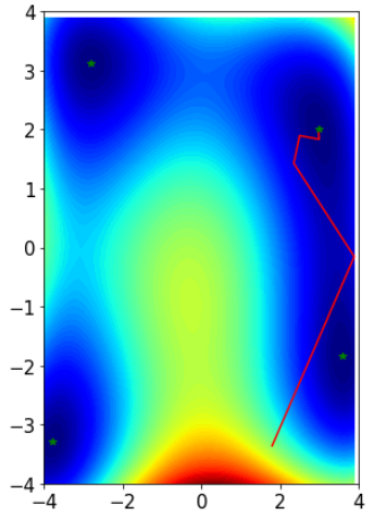

Figura 3.9: Método BFGS

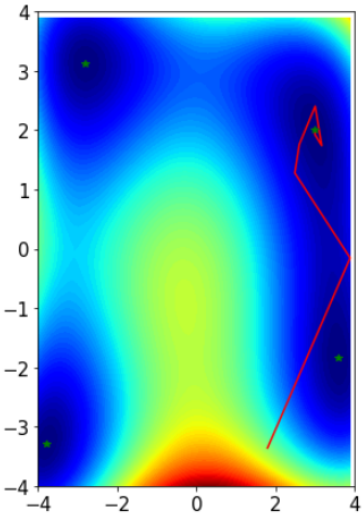

Figura 3.10: Método SR1

En las Figuras anteriores se muestran gráficos de contorno de la Función de Himmelblau, donde los cuatro puntos de estrella representan las combinaciones de puntos de extremos que minimizan la función. El objetivo es que los métodos probados puedan llegar a alguno de estos puntos minimizadores, y poder luego medir la precisión y el rendimiento a nivel de convergencia de estos métodos.

Además, también se aprecia en la Figura 3.9 y Figura 3.10 que debido a la convergencia superlineal en los métodos cuasi-Newton, estos se dirigen de una manera más directa hacia uno de los puntos que minimizan la función dada y lo hacen con menor esfuerzo computacional en cada iteración. También se puede notar cómo cada uno de los métodos posee una tendencia a dirigirse hacia un punto distinto, sin embargo, dado que los cuatro métodos inician desde un mismo punto, las diferencias en las direcciones tomadas han de ser inherentes al método.

Indistintamente, entre todas las combinaciones de puntos iniciales probados, los métodos cuasi-Newton (Figura 3.9 y Figura 3.10) llevan considerable ventaja en términos de la velocidad con la que convergen a alguno de los puntos considerados como solución. Se puede dar el caso de que los métodos convencionales sean más propensos a estar mal condicionados desde un principio, y en consecuencia cuando nos encontramos con problemas altamente no convexos y con grandes cantidades de parámetros, la probabilidad de divergencia, de encontrarse con problemas de singularidad o simplemente de quedarse atrapado en un punto de silla es mucho mayor. Es interesante ver cómo, a pesar de que los métodos cuasi-Newton (Figura 3.9 y Figura 3.10) no hacen uso de información de segundo orden de la función, estos infieren a partir del modelo cuadrático y de gradientes sucesivos anteriores las direcciones óptimas que conllevan a una solución. Estos métodos, al combinarse con otras estrategias de optimización como lo son la búsqueda de línea exacta o estrategias de región de confianza, pueden volverse aún más potentes. En este caso, en la implementación de BFGS y SR1 se hace uso de una estrategia de búsqueda de línea exacta, asegurando así las condiciones de Wolfe expuestas anteriormente para ambos casos. Es notable cómo

el método de Newton es el mejor en términos de cantidad de iteraciones y norma obtenida, debido a su convergencia cuadrática. La uniformidad estructural del método de Newton le proporciona sus propiedades de convergencia cuadrática. Sin embargo, el método de Newton no tiene por qué funcionar siempre, debido a que puede ser imposible invertir la matriz hessiana por problemas de singularidad, o simplemente pudiera divergir. Adicionalmente, las iteraciones dentro del método de Newton se van cada vez haciendo más costosas a nivel computacional porque se necesitan n^2 evaluaciones de la función del hessiano. Es en estos casos en donde los métodos de segundo orden pueden causar problemas, y debido a que los cuasi-Newton los emulan, pueden lograr resultados robustos, comparables al método de Newton, sin acudir a los elementos de segundo orden directamente. Es esto por lo que vemos una similitud en términos de cantidad de iteraciones entre los dos métodos cuasi-Newton probados y el método de Newton, teniendo en cuenta que a pesar de la convergencia cuadrática del método de Newton, este no tiene por qué siempre ser igual de rápido. Aún se debe tener en cuenta que, dados los problemas reales en optimización matemática, no es garantizable que ningún método sea más rápido que los demás en todos los casos. Los métodos más avanzados como SR1 y BFGS tienen una menor cantidad de cálculos, asociados a las matrices que aproximan los elementos de segundo orden, aunque éstas pudieran tener un mayor peso a nivel computacional.

Los métodos cuasi-Newton logran una convergencia superlineal, ahorrando tiempo computacional al calcular las matrices de segundo orden dentro de cada iteración. Sin embargo, vemos cómo ambos métodos cuasi-Newton obtienen una norma más alta que en la del método de Newton, lo que quiere decir que, aunque los dos métodos cuasi-Newton escogieron un set de puntos de mínimos distinto al del método de Newton y máximo descenso, el criterio de parada se logra en una proporción menos eficiente a la del método de Newton. Este último punto también se puede evidenciar en la cantidad de iteraciones que posee el método de Newton contra los cuasi-Newton: el método de Newton obtiene 7 iteraciones mientras que los métodos cuasi-Newton obtienen 8 y 9 para BFGS y SR1, respectivamente. El criterio de parada representa una serie de condiciones donde el algoritmo de búsqueda local se habrá considerado haber alcanzado una solución. El criterio de parada utilizado en los cuatro algoritmos planteados es que la búsqueda ha de parar cuando la norma Euclidiana del gradiente ha alcanzado un valor predeterminado de $\epsilon < 10^{-3}$, también denominado valor de la tolerancia. La norma $L2$ (norma Euclidiana) es simplemente el gradiente negativo normalizado, lo que busca obtener una dirección de paso distinta dentro de cada iteración.

El método ilustrado ha sido aplicado a una función convexa, razón por la cual la diferencia en cantidades de iteraciones no es de una magnitud muy grande, y vemos que todos los métodos probados se acercan considerablemente bien a una de las soluciones. Sin embargo, cuando hablamos de problemas en donde la matriz hessiana pudiera tener cientos de miles de parámetros, la diferencia se vuelve más notable y a nivel computacional vamos viendo cómo el grupo de métodos cuasi-Newton llevan ventaja. El cálculo de la inversa de la matriz hessiana también puede verse propenso a un mal condicionamiento numérico, lo que puede causar que la matriz se encuentre con problemas de singularidad. Los métodos cuasi-Newton son bastante efectivos a la hora de evitar este tipo de situaciones, dado que obtienen no solo una aproximación de la matriz de elementos de segundo orden, sino que también realizan la aproximación de su inversa de forma alternativa, a los mismos fines de mantener una eficiencia a nivel computacional.

De los cuatro métodos probados, el método que obtiene la menor eficiencia computacional, con la mayor cantidad de iteraciones, es el método del máximo descenso, o descenso de gradiente. Esto debido al cálculo en bucle que se realiza sobre la función del vector gradiente en un esquema de búsqueda de línea con un $\alpha = 0.01$ constante. Las evaluaciones sucesivas de los nuevos puntos x_k obtenidos sobre la función del gradiente, para luego realizar el cálculo del nuevo punto usando la Ecuación descrita en (2.2), hacen que el método sea especialmente lento al converger al nivel estipulado de $\epsilon < 10^{-3}$.

Los otros métodos (Newton y cuasi-Newton) se acercan a una diferencia negativa ligeramente mayor al parámetro ϵ, lo que hace que la norma del gradiente obtenido sea menor para estos métodos. Como se ve en las Figuras 3.7, 3.9 y 3.10, se aproximan de forma eficiente a uno de los puntos iniciales escogidos, aún siendo estos puntos distintos a los escogidos por el algoritmo de máximo descenso (Figura 3.8). Esto nos indica que los cuasi-Newton tienen propiedades de convergencia distintas a las de sus contrapartes (superlineales), mientras que los dos métodos que poseen una convergencia cuadrática pueden ser más lentos efecto del ralentizado proceso iterativo que nos resulta de calcular los elementos de segundo orden de manera directa.

Es bueno también notar que la fórmula de actualización simétrica de rango 1 (SR1) se ha aplicado bajo un esquema de búsqueda de línea, a pesar de lo que se explicaba anteriormente de que este algoritmo suele aplicarse bajo un esquema de región de confianza. En algunos problemas, la actualización SR1 no garantiza que la matriz actualizada mantenga una definición positiva. Sin embargo, para el problema planteado en la Ecuación (3.20) nos permite mantener la definición positiva bajo el esquema de actualización de SR1, y se aplicó el esquema de actualización del tamaño del paso usando la estrategia de optimización de búsqueda de línea de manera efectiva.

Los métodos cuasi-Newton son muy utilizados en la resolución de problemas sin restricciones. Sin embargo, también se utilizan en combinación con otros métodos en problemas con restricciones lineales para hallar la dirección de búsqueda en cada iteración. Dentro de las actualizaciones descritas anteriormente, la más utilizada y la más potente es la BFGS [13].

En este mismo sentido, el hecho de que las aproximaciones del hessiano en los métodos cuasi-Newton evolucionen mediante la actualización de fórmulas iterativas, hace que el análisis de convergencia de los métodos cuasi-Newton sea mucho más complejo que el de máximo descenso o del método de Newton [3]. Una de las desventajas de los métodos cuasi-Newton es que pueden proveer resultados pobres si se escogen puntos iniciales que estén muy alejados de una de las soluciones. Esto depende también de cómo se aproxime la matriz hessiana inicial, si se calcula directamente o si se empieza el conteo de iteraciones con la matriz hessiana como la

matriz identidad. Ambos métodos DFP y BFGS básicamente aplican correcciones a una aproximación inicial de la matriz Hessiana, teniendo en cuenta elementos del vector gradiente. De hecho, aún no se sabe si los algoritmos gozan de propiedades de convergencia verdaderamente globales para funciones objetivo no lineales generales. Es decir, no podemos probar que las iteraciones de estos métodos cuasi-Newton se acerquen a un punto estacionario del problema desde cualquier punto de partida y cualquier aproximación adecuada del hessiano inicial [3].

Existen otros métodos estructurados cuasi-Newton, muchos de ellos se consideran híbridos ya que unifican metodologías en función del problema a resolver. Entre ellos se encuentra el método Dennis, Gay y Welsh (DWG) [20], el cual posee propiedades de convergencia q-superlineales, con razonables estimaciones de la matriz hessiana real. Estas aproximaciones de la matriz hessiana real se aplican con algún miembro de la familia convexa de Broyden (BFGS, DFP, PSB o Powell-Symmetric-Broyden). Sin embargo, DWG no garantiza que las actualizaciones de B_k sean definidas positivas (lo que conduce a regiones de confianza dentro del método). Este método en particular no garantiza la convergencia de B_k a la matriz nula en el caso cero residual. Por lo tanto, para estos casos donde el método pierde sus rápidas propiedades de convergencia se controla el residual mediante métodos híbridos. Se utiliza Gauss-Newton en residuales pequeños, y BFGS en residuales grandes, lo cual convierte a este método en uno híbrido, estructurado cuasi-Newton. La política de cambio entre dos algoritmos en un método híbrido es crucial para lograr tasas de convergencia de q-cuadrática a q-superlineales [21]. Por lo tanto, los métodos que combinan múltiples metodologías para acelerar sus tasas de convergencia pueden variar en los niveles de efectividad que alcanzan, dependiendo del problema.

Por otro lado, hay resultados de convergencia superlineal locales bien conocidos que son ciertos bajo supuestos razonables. Tanto los métodos BFGS como SR1 poseen de este tipo de convergencia, no solamente para funciones convexas, sino también para funciones objetivo no lineales.

En el Capítulo 4, comenzaremos a tratar el aprendizaje profundo, que es una de las más recientes aplicaciones de los métodos cuasi-Newton.

Capítulo 4

Aprendizaje profundo

4.1. Aprendizaje profundo

En este Capítulo veremos en qué consiste el aprendizaje profundo, cómo encaja dentro de la optimización, así como dentro de la amplia gama de campos que es la inteligencia artificial. La inteligencia artificial (IA) consiste en cualquier técnica computacional que trate de emular la inteligencia humana, ya sea mediante el aprendizaje profundo u otra técnica. Debido a su efectividad como aprendices no lineales generales [22], el aprendizaje profundo se ha aplicado a un amplio espectro de problemas, incluidos la robótica [23], el reconocimiento visual [24] [25], el procesamiento del lenguaje natural [26], el modelado acústico [27] y muchos otros.

El **aprendizaje automático** nos permite abordar tareas que son demasiado complejos de resolver con programas fijos escritos y diseñados por seres humanos. Desde un punto de vista científico y filosófico, el aprendizaje automático es interesante porque desarrollar nuestra comprensión de este implica desarrollar nuestra comprensión de los principios que subyacen al cerebro humano. Ahora bien, el **aprendizaje profundo** es un tipo específico de aprendizaje automático que hace uso de redes neuronales artificiales de varias capas (3 como mínimo, es decir, $N > 2$). Este resuelve un problema de aprendizaje de representación de datos e influencia de variaciones que suele ser una de las dificultades más prevalentes dentro de la inteligencia artificial. Las aplicaciones de esta requieren que podamos *desenredar* los factores de variación de los datos y podamos descartar los que nos aportan poco [28].

Los algoritmos de aprendizaje profundo intentan resolver problemas de aprendizaje automático a gran escala mediante el aprendizaje de un modelo (o un aproximador de función) a partir de los datos observados para predecir las representaciones latentes (u ocultas) dentro de un modelo. Algunos de los primeros algoritmos de aprendizaje que reconocemos hoy estaban destinados a ser modelos computacionales de aprendizaje biológico, es decir, modelos de cómo podría ocurrir el aprendizaje en el cerebro. Esta es la razón por la que uno de los nombres que el aprendizaje profundo ha sido bautizado con es **redes neuronales artificiales** [28].

En este mismo sentido, existe otro subcampo del aprendizaje automático denominado aprendizaje por refuerzo, que se compara en varias ideas con el aprendizaje profundo, sin embargo, no son exactamente lo mismo. Aunque el aprendizaje profundo tiene una poderosa capacidad de representación de datos, no es suficiente para construir un sistema de IA inteligente. Esto se debe a que un sistema de IA no solo debe poder aprender de los datos proporcionados, sino también de las interacciones con el entorno del mundo real como un ser humano. El **aprendizaje por refuerzo** (RL) es un subconjunto del ML que permite que las computadoras aprendan interactuando con el entorno del mundo real. En resumen, RL separa el mundo real en dos componentes: un entorno y un agente. El agente interactúa con el entorno realizando acciones específicas y recibe retroalimentación del entorno. La retroalimentación generalmente se denomina recompensa en RL. El agente aprende a desempeñarse mejor tratando de obtener recompensas más positivas del entorno. Este proceso de aprendizaje forma un circuito de retroalimentación entre el entorno y el agente, guiando la mejora del agente con algoritmos de RL [29]. Más recientemente, los algoritmos de optimización cuasi-Newton no solo han sido aplicados para optimizar redes neuronales profundas, sino también para optimizar problemas bajo un esquema de aprendizaje por refuerzo [2].

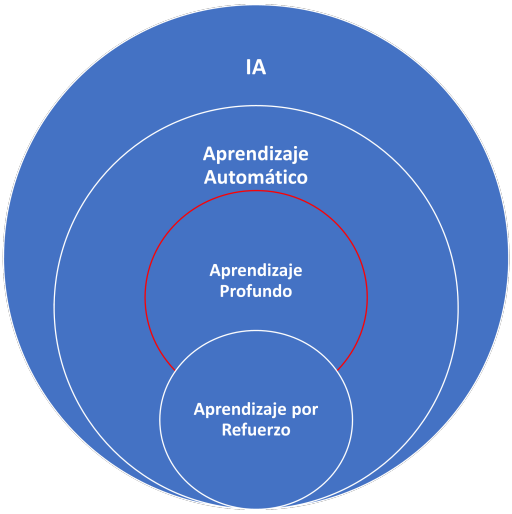

Figura 4.1: Relación entre la IA, Aprendizaje Automático, Aprendizaje Profundo y Aprendizaje por Refuerzo Profundo.

En la Figura 4.1 se puede ver un Diagrama de Venn con algunos de los subcampos que conforman la inteligencia artificial. Se puede observar cómo algunos de estos campos se solapan entre sí, como lo hacen el aprendizaje profundo (DL) y por refuerzo (RL). A la interacción de estos dos campos se le conoce como el aprendizaje por refuerzo profundo. Como se mencionó anteriormente, dentro del aprendizaje profundo se hace uso de redes neuronales artificiales, cuando en el aprendizaje por refuerzo se hace uso de otros esquemas de exploración y explotación de los datos para entrenar un agente hacia un determinado objetivo. Al aprendizaje por refuerzo profundo se le conoce como el uso combinado de esquemas de redes neuronales artificiales, así como también del esquema en forma de circuito del RL.

Dicho esto, amerita plantear brevemente en qué consiste una red neuronal artificial, que es lo que se consideraría como la columna vertebral del aprendizaje profundo, que será el tema a discusión dentro de la siguiente Sección.

4.2. Esquema básico de una red neuronal

La forma en la que el aprendizaje profundo resuelve el problema de representación es introduciendo representaciones que son expresadas en términos de otras representaciones más simples. Aquí es donde entran las redes neuronales artificiales.

4.2.1. Perceptrones

Las **redes neuronales** son una caja de herramientas para abordar problemas de aprendizaje supervisado, desde tareas de procesamiento de lenguaje natural (NLP) hasta tareas de visión computacional o artificial. Para comprender su popularidad, rastreamos su historia, apoyándonos principalmente en el esquema proporcionado por Nielsen [30]. En 1957, Frank Rosenblatt presentó el *perceptrón*, un modelo que imita ampliamente el funcionamiento de una neurona biológica: este recibe entradas y luego toma la decisión de disparar o no la salida en función de estas entradas.

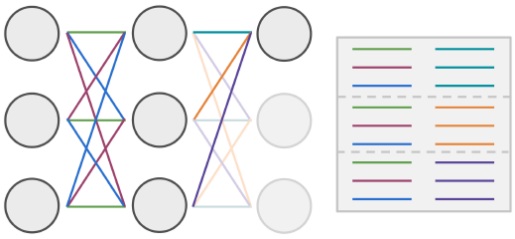

Figura 4.2: Modelo global del perceptrón en una CNN.

El modelo del perceptrón simple se ilustra en la Figura 4.2, en donde los círculos representan las unidades del perceptrón. Este modelo toma como entrada el vector $x = \{x_1, \ldots, x_n\}$, y como salidas ya sea un 0 o un 1, imitando el disparo de una neurona biológica. El disparo se determina usando una serie de pesos $w = \{w_1, \ldots, w_n\}$, números reales que expresan la importancia de las entradas respectivas con la salida, y otro parámetro llamado sesgo o b, que es lo que determina la salida de la neurona. Este modelo puede expresarse como:

$$Salida = \begin{cases} 0, & \text{si } wx + b \leq 0 \\ 1, & \text{si } wx + b > 0, \end{cases}$$

en donde ajustando los pesos w_n y el sesgo b, el perceptrón atribuye a las entradas el valor de la salida (output). Este básicamente es el modelo matemático, viendo que se trata meramente de tomar decisiones basadas en evidencias ponderadas de parámetros. Variando los pesos y el sesgo, se pueden obtener distintos modelos de toma de decisiones. Sin embargo, el perceptrón simple no representa un modelo completo de la toma de decisiones que puede llegar a tomar un humano. Por eso existen niveles de abstracción superiores al perceptrón simple, lo que se conoce como perceptrones multicapa, que es básicamente agregar capas adicionales al modelo visto anteriormente, tomando como motivación el nivel más complejo de toma de decisiones que aplican las neuronas biológicas. De esta forma, una red de perceptrones de muchas capas permite una toma de decisiones más sofisticada.

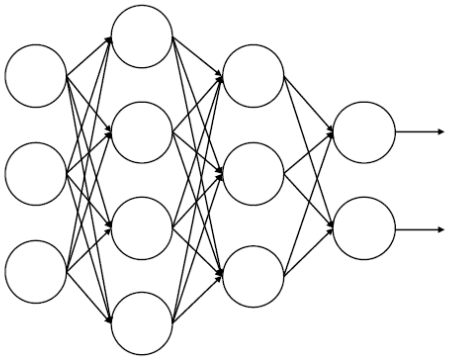

Figura 4.3: Modelo del perceptrón multicapa.

El modelo del perceptrón multicapa se ilustra en la Figura 4.3, en donde las dos capas de neuronas centrales representan las capas añadidas entre las entradas y salida del modelo. Aquí, aunque el modelo presenta varios nodos como parte de la capa de la salida, esto aún se considera una sola salida, en el caso de que esta fuera multiclase. También se puede observar cómo todos los nodos se conectan entre sí representando la interacción completa que tienen los parámetros w y b con las capas de entrada y de salida.

Sin embargo, bajo el esquema actual es muy difícil que algún aprendizaje complejo y capaz de captar relaciones sutiles suceda. Si los pesos y el sesgo de un perceptrón varían levemente es probable que el resultado de la red sea totalmente distinto. Por esto, al cálculo de la decisión se le aplica una función no lineal que se denomina función de activación [30]. De esta forma en vez de tener salidas binarias ahora consideraríamos salidas que se calculan como $f(wx + b)$, donde f es la función de activación no lineal. Esta debe ser no lineal ya que de otra forma la red neuronal se vería reducida a una sola capa de neuronas, ya que se verían cerradas por composición de su función lineal [29]. Existen varias funciones de activación que se usan regularmente en el campo, entre ellas:

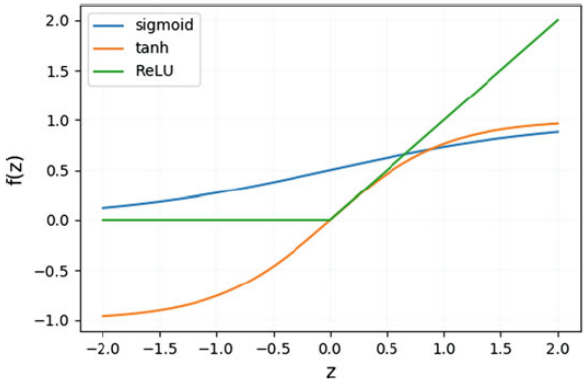

Figura 4.4: Muestra de 3 funciones de activación por elementos.

La función de activación sigmoide fue la primera considerada, de la forma $\sigma(x) := (1 + \exp -x)^{-1}$ [30]. Esta hará que la salida se encuentre entre 0 y 1, como se observa en la Figura 4.4, lo que convertirá a la salida en una suave y continua aproximación del perceptrón. Tanh retornará valores entre -1 y 1, mientras que ReLU retornará 0 cuando la entrada (Z) no sea positiva, y el equivalente a $f(x) = x$ cuando la entrada es positiva.

4.3. Optimización de redes neuronales

Pero, ¿cómo se relacionan las redes neuronales con la optimización matemática? La mayoría de los algoritmos de aprendizaje profundo requieren un paso de optimización [28]. Y es que hasta refiriéndonos a la selección de la cantidad de capas intermedias u ocultas estamos hablando de hiperparámetros que pueden o no afectar la resolución del problema. Para armar el esquema iterativo de optimización se define la función de pérdida, $\mathcal{L}(w, b)$, que sería la diferencia entre el resultado arrojado por la red y el esperado, y se va iterando sobre esta hasta obtener resultados aceptables. A este proceso se le conoce como *backpropagation*, que consiste en ir propagando en dirección contraria de la red neuronal los valores de los parámetros para ir actualizándolos de manera tal que la función de pérdida se vaya minimizando con cada iteración [28]. Esto se hace calculando las derivadas (con las funciones de activación asociadas a cada neurona) de cada una de las funciones matemáticas contenidas en las neuronas, para ir en función de esto cambiando los parámetros. La función de pérdida puede definirse como el error medio cuadrático (MSE), en donde a_i sería la salida de la red para cada entrada x_i:

$$\mathcal{L}(w, b) = \sum_{i=1}^{n} ||y_i - a_i||_2^2. \tag{4.1}$$

Otra forma de construir nuestra función de pérdida es mediante la implementación de una función denominada softmax, que es una variante de la sigmoide en donde se emplea un vector θ para obtener como resultado una distribución de probabilidad. Sobre esta función, se aplica una minimización basada en entropía cruzada (cross-entropy), que minimiza la diferencia de esta distribución de probabilidad contra la distribución original, construyendo así un clasificador de máxima verosimilitud [28].

Por lo general, cuando entrenamos un modelo de aprendizaje automático, como tenemos acceso a un conjunto de datos de entrenamiento, podemos calcular alguna medida de error en el conjunto de entrenamiento llamada error de entrenamiento y esto es lo que minimizamos en la optimización. Hasta ahora, lo que hemos descrito es simplemente un problema de optimización. Lo que separa el aprendizaje automático de la optimización es que se desea que el error de generalización, o error de prueba, también sea bajo. El error de generalización se define como el valor esperado del error en una nueva entrada de datos. Aquí la expectativa se toma a través de diferentes posibles entradas, extraídas de la distribución de entradas que esperamos que el sistema encuentre en la práctica. Por lo general, estimamos el error de generalización de un modelo de aprendizaje automático midiendo su rendimiento en un conjunto de prueba de ejemplos que se recopilaron por separado del conjunto de entrenamiento [30].

Algunos de los problemas que suelen aparecer cuando optimizamos redes neuronales es que podemos tener gradientes inestables o el llamado problema del gradiente explosivo [31], problema que se aprecia en la Figura 4.5.

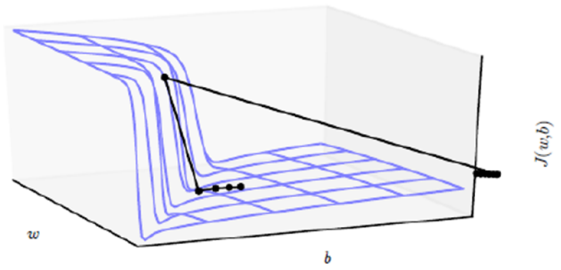

Figura 4.5: Demostración del problema del gradiente explosivo.

Las redes neuronales con muchas capas a menudo tienen regiones extremadamente empinadas que se asemejan a acantilados. En la cara de una estructura de acantilado extremadamente empinada, el paso de actualización de gradiente puede mover los parámetros extremadamente lejos, por lo general saltando fuera de la estructura del acantilado por completo, lo que hace que la actualización del gradiente se vuelva inestable y difícil de realizar de forma óptima [28].

La función objetivo para redes neuronales profundas altamente no lineales a menudo contiene marcadas no linealidades en el espacio de parámetros, que resultan de la multiplicación de varios de ellos. Estas no linealidades dan lugar a derivadas muy altas en algunos lugares. Cuando los parámetros se acercan a tal región de acantilado, una actualización de descenso de gradiente puede catapultar los parámetros muy lejos, posiblemente perdiendo la mayor parte del trabajo de optimización que se había previamente realizado [28]. A medida que se agregan más capas a las redes neuronales, los gradientes de la función de pérdida se aproximan a cero, lo que dificulta el entrenamiento de la red. Por lo tanto, el gradiente disminuye exponencialmente a medida que nos propagamos hacia las capas iniciales. Un gradiente pequeño significa que los pesos y sesgos de las capas iniciales no se actualizarán de manera efectiva con cada época de entrenamiento. Dado que estas capas iniciales a menudo son cruciales para reconocer los elementos centrales de los datos de entrada, pueden conducir a una inexactitud general de toda la red. La solución más sencilla es utilizar funciones de activación, como ReLU, que no provoca una pequeña derivada. De igual forma, este problema se puede reducir utilizando distintas arquitecturas de redes neuronales que normalizan la dirección tomada por el gradiente, usando técnicas adaptativas que veremos a continuación. También, aplicando términos de regularización dentro del algoritmo de optimización seleccionado. Veremos algunos de los algoritmos más comúnmente utilizados a continuación.

4.3.1. Descenso de gradiente estocástico (SGD)

Dentro del aprendizaje profundo, usualmente se usa el algoritmo del descenso de gradiente estocástico (SGD) [32], o alguna de sus variantes para optimizar el problema como tal sobre la función de pérdida. Este algoritmo es una extensión al descenso de gradiente discutido en la Sección 2.2.1, en donde se asume que la función de pérdida es separable y que puede ser descompuesta para solamente depender de una muestra de los datos de entrenamiento, de tamaño m. Esta parte solo se aplica para recalcular el gradiente y evitar altas cargas computacionales del mismo sobre la muestra total de datos. La función de pérdida puede ser reescrita como

$$\mathcal{L}(w,b) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}_i(w,b), \tag{4.2}$$

y lo mismo aplicaría para la función del gradiente. En cada paso, en lugar de usar todos los datos de entrenamiento disponibles para calcular el gradiente, sólo lo calculamos usando un pequeño subconjunto, B, de los ejemplos de entrenamiento elegidos uniformemente al azar. A esta muestra también se le reconoce como *mini-batch*, de tamaño m, y con esto el gradiente de la función es entonces aproximado imparcialmente habiendo tomado el subconjunto de entradas de la distribución original de datos de entrenamiento [30]. Para mejor visibilidad del funcionamiento de este método, ver Algoritmo 5.

En este Algoritmo, los hiperparámetros se corresponden con la tasa de aprendizaje η, el número de épocas, el tamaño del mini-batch m, en conjunto con los hiperparámetros de la arquitectura de la red neuronal.

SGD es una alternativa computacionalmente eficiente al entrenamiento de lotes completos, pero introduce ruido en el gradiente, lo que puede obstruir al proceso de optimización. Sin embargo, ya uno de los algoritmos más concurridos en DL no es el SGD llano, sino el SGD con impulso (que es uno de los que se usarán dentro de este trabajo). También existen muchas otras extensiones a SGD que se basan en esta misma idea, aplicándole modificaciones para hacerlo más escalable y de mayor rendimiento. Algunas de las extensiones más conocidas son RMSProp [33] y Adam [34], que también son algoritmos de optimización estándares a la hora de optimizar redes neuronales.

Algoritmo 5 Descenso de Gradiente Estocástico (SGD)

Entrada: Set de entrenamiento de tamaño N
  Inicializar $v = (w, b)$ y tasa de aprendizaje η
  Para $l = 1, \ldots,$ número de épocas **Hacer**
    El set de entrenamiento se fija sobre la data completa
    Para $i = 1 \ldots n/m$ **Hacer**
      Seleccionar uniformemente m ejemplos del grupo de entrenamiento;
      Calcular el gradiente aproximado $\frac{1}{m} \sum_{j \in B} \nabla \mathcal{L}_j(v)$
      Actualizar $v := v - \eta \frac{1}{m} \sum_{j \in B} \nabla \mathcal{L}_j(v)$
      Restar la muestra tomadas en m del set de entrenamiento
    Fin
  Fin
  Devolver Conjunto de parámetros entrenados v

4.3.2. Variaciones de SGD

La idea de RMSProp (*Root Mean Square Propagation*) [33] se basa en utilizar una tasa de aprendizaje adaptativa, que se encuentra en cada paso usando los gradientes anteriores de la función de pérdida, lo que va reduciendo las oscilaciones presentes en SGD y resulta en tasas de convergencia superiores. El algoritmo aplica un factor de olvido β, mientras va creando una media móvil del cuadrado de los gradientes:

$$V(z_i) := \beta V(z_{i-1}) + (1 - \beta)(\nabla \mathcal{L}(z_i))^2. \tag{4.3}$$

Esto luego se utiliza para adaptar la tasa de aprendizaje para un paso del algoritmo iterativo, usando la regla de actualización dada por:

$$z_{i+1} = z_i - \frac{\eta}{\sqrt{V(z_i)}} \nabla \mathcal{L}(z_i). \tag{4.4}$$

Para el cálculo de todos los gradientes también se utiliza la regla proveniente de SGD, en donde solamente se toma una muestra del set de aprendizaje (un *mini-batch*). Los creadores del algoritmo sugieren utilizar un factor de olvido $\beta = 0.9$, para de esta forma ir adaptando la tasa de aprendizaje de forma tal que se acelere la tasa de convergencia del algoritmo en la dirección horizontal. Este factor tiene el efecto de hacer que la media móvil del cálculo de los gradientes sea decreciente y por ende corrige el sesgo presente en cada brusca iteración en SGD.

Otro algoritmo más regularmente utilizado es Adam [34], que se basa en una técnica adaptada del método del impulso o *momentum*, originalmente introducido en 1986 por Rumelhart et. al. en [35], y básicamente intenta acelerar la tasa de aprendizaje para tomar pasos más grandes en cada iteración. Adam aplica la idea del impulso creando una aceleración a la tasa de aprendizaje, y a la vez corrigiendo el sesgo de esta aceleración mediante el aprendizaje de las oscilaciones con las actualizaciones del gradiente, que es la misma idea presente en RMSProp. Es decir, Adam se consideraría un método híbrido ya que integra la idea de la tasa de aprendizaje adaptativa de RMSProp y la aceleración del paso provista por el método del impulso. Se crean dos vectores iniciales aplicando el factor de olvido β (decay rate), que son luego normalizados antes de calcular el tamaño del paso real:

$$\begin{aligned} v_t &\leftarrow \beta_1 v_{t-1} + (1 - \beta_1) g_t \\ s_t &\leftarrow \beta_2 s_{t-1} + (1 - \beta_2) g_t \odot g_t \end{aligned}$$

$$\widehat{s_t} \leftarrow \frac{s_t}{1 - \beta_2^t} \quad \hat{v_t} \leftarrow \frac{v_t}{1 - \beta_1^t}$$

$$x_t = x_{t-1} - \frac{\eta \hat{v_t}}{\sqrt{\hat{s_t}} + \epsilon} \tag{4.5}$$

Los vectores v_t y s_t representan los vectores de velocidad y de estados, respectivamente, ambos inicializados en cero. El vector de estados hace referencia a la media móvil que se crea a partir del gradiente, buscando un decaimiento exponencial en las oscilaciones. Este hace que el paso se mueva menos en diagonal hacia el mínimo local y más de manera directa, mientras que el vector de la velocidad busca ir ralentizando el tamaño del paso a medida nos acercamos más hacia el final de las iteraciones.

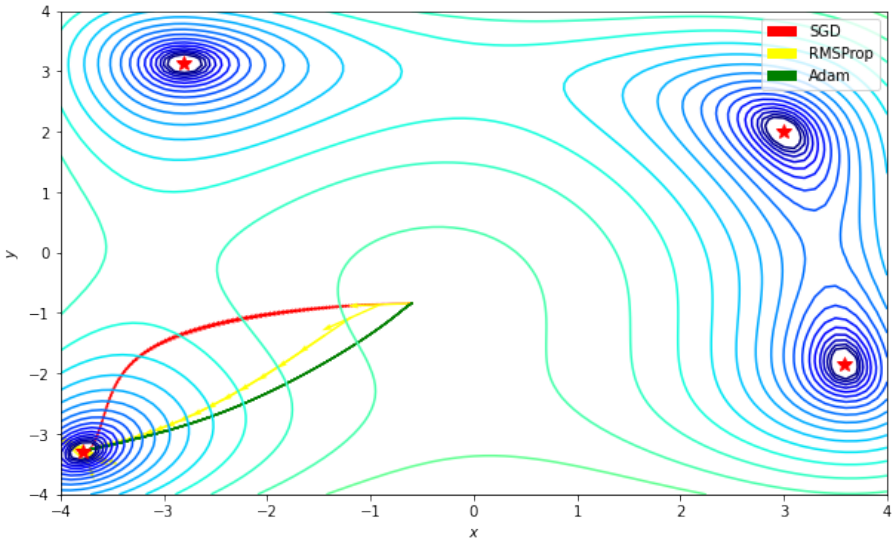

Figura 4.6: Comparación de la trayectoria de los algoritmos discutidos.

En la Figura 4.6 se muestra la Función (3.20) con sus referidos puntos de mínimos, en donde se aplicaron los algoritmos discutidos en este Capítulo, a modo de referencia a la explicación, todos partiendo desde el mismo punto inicial $x = [-0.6, -0.84]$. Se puede observar cómo el sesgo de las oscilaciones del gradiente es corregido en el método de Adam, dirigiéndose de una forma más directa hacia uno de los puntos estrella. También se observa cómo el método RMSProp inicia de forma contundente hacia la misma dirección de búsqueda que SGD, pero luego normaliza su dirección mientras va creando la media móvil del cuadrado de sus gradientes y ralentizando su tasa de aprendizaje con el factor de olvido β. En la línea del optimizador Adam, para corregir el sesgo que se crea de haber inicializado ambos vectores de velocidad y de estados en cero al principio de las iteraciones, estos se normalizan de acuerdo con la iteración actual t que crea el efecto de estar cada vez más cerca de dividir entre 1, lo cual nos deja meramente con el vector de velocidad y de estados. Con lo cual, estas normalizaciones tendrán un efecto más pronunciado al inicio de las iteraciones. Finalmente, como se observa en (4.5), para calcular el tamaño del paso se hace uso de los vectores normalizados, tomando la tasa de aprendizaje η multiplicada por el vector de la velocidad. Se multiplica la normalización de los vectores del gradiente (RMSProp) por un factor ϵ por temas de estabilidad numérica (para evitar dividir entre cero), pero esto también implica un decaimiento exponencial hacia el final de las iteraciones. Como es evidente, Adam es un algoritmo que tiene que ser manipulado a nivel de múltiples parámetros, empezando con la idea de la aceleración del impulso para luego ir haciendo más hincapié del lado del RMSProp, en donde se estaría normalizando la tasa de aprendizaje en base a los gradientes anteriores. Los autores recomiendan usar como hiperparámetros predeterminados: $\eta = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ y $\epsilon = 10^{-8}$. La única diferencia entre RMSProp y Adam es que éste último realiza la corrección del sesgo para el primer (v_t) y segundo (s_t) momento del gradiente, mientras que RMSProp no lo hace. La implementación de la Figura 4.6 a nivel de código se puede hallar en el enlace provisto en el Anexo (Capítulo 7). Para mejor visibilidad del funcionamiento del método Adam, ver Algoritmo 6.

Algoritmo 6 Optimizador Adam

Entrada: Parámetros x_0, tasa de aprendizaje η, no. iteraciones m, $\beta_1 = 0.9$, $\beta_2 = 0.999$ y $\epsilon = 10^{-8}$
Inicializar vectores $v_0 \leftarrow 0$ y $s_0 \leftarrow 0$
Para $t = 1, \ldots,$ número de iteraciones **Hacer**
Calcular gradiente con mini-batch: $\frac{\partial \mathcal{L}}{\partial \theta}$
Actualizar: $v_t \leftarrow \beta_1 v_{t-1} + (1 - \beta_1) g_t$
Actualizar: $s_t \leftarrow \beta_2 s_{t-1} + (1 - \beta_2) g_t \odot g_t$
Calcular media móvil del gradiente cuadrático medio: $\widehat{s_t} \leftarrow \dfrac{s_t}{1 - \beta_2^t}$
Calcular media móvil del gradiente medio: $\hat{v_t} \leftarrow \dfrac{v_t}{1 - \beta_1^t}$
Actualizar: $x_t = x_{t-1} - \dfrac{\eta \hat{v_t}}{\sqrt{\widehat{s_t}} + \epsilon}$
Fin
Devolver x_t, los parámetros entrenados

4.4. Redes neuronales convolucionales

Recapitulando, los algoritmos de optimización más utilizados dentro del aprendizaje profundo son el descenso de gradiente estocástico con impulso, RMSProp y Adam. Sin embargo, hemos visto que el descenso del gradiente falla al no explorar la información de curvatura de la función que se encuentra en la matriz hessiana de la función. Esta información puede servir bastante en la tarea de optimización ya que nos permite alcanzar el punto buscado de una manera más eficiente y directa. Cabe resaltar que SGD funciona bien cuando trabajamos con pequeños lotes de la función de coste, así como también nos provee de la posibilidad de aplicar la técnica del impulso discutida anteriormente. Sin embargo, mientras nos acercamos a funciones de mayor escala, la posibilidad de quedarse en puntos de silla puede exacerbarse, por lo que metodologías alternas a SGD fueron desarrolladas para normalizar los sesgos que presenta SGD como resultado de la trayectoria tomada en la optimización. Estas metodologías son métodos como RMSProp y Adam, que resuelven las principales debilidades de SGD, como aplicar un tamaño del paso adaptativo, de manera tal que este no sea fijo en toda la optimización y que al acercarnos hacia el final de las iteraciones este se haga más lento. Estas metodologías se vuelven indispensables a la hora de trabajar con funciones de costes no convexas, en donde la posibilidad de divergir en la tarea de optimización es mucho más alta, como es el caso en las redes neuronales.

Haremos hincapié especial en una arquitectura particular de redes neuronales, a modo de presentar su contexto para luego en el siguiente Capitulo aplicar tanto dos de los algoritmos discutidos para optimizarla, así como también un método cuasi-Newton. Entre las diversas posibles arquitecturas que se utilizan acorde al problema a resolver, entre ellas están las redes neuronales clásicas tipo feedforward [36], recurrentes (RNN) [35], convolucionales (CNN) [37], redes generativas antagónicas (GAN) [38] y transformadores [39]. Particularmente, en esta Sección nos centraremos en las redes convolucionales, que se aplican para la resolución de problemas de reconocimiento de imágenes y de voz [30], entre otros casos de uso.

Las redes neuronales convolucionales pueden entenderse como un tipo de redes feedforward diseñadas específicamente para procesar datos en forma de múltiples matrices [40]. Las CNN son algoritmos de última generación en los campos de la visión por computadora y el reconocimiento de voz.

Entre algunas de las arquitecturas de redes neuronales convolucionales más ubicuas se encuentran LeNet-5 [40], AlexNet [41], VGG16 [42], ResNet [43], Inception Network [44], MobileNet [45] y EfficientNet [46]. Muchas de estas tienen múltiples versiones, que se han ido desarrollando sobre sí mismas a medida se prueba lo que funciona y lo que no funciona en la práctica, extrapolando así la teoría desde la práctica. Igualmente, se añaden modificaciones e innovaciones a la forma en la que se aborda el problema de escalabilidad de la arquitectura.

Una red convolucional es normalmente utilizada con tres tipos distintos de capas:

1. Las capas *convolucionales*, estas hacen referencia a una serie de filtros que se van activando en secuencia para formar la capa de salida.
2. Las capas *pooling* reducen el tamaño de la siguiente entrada aplicando un filtro de máximo o promedio sobre la matriz completa.
3. Las capas *totalmente conectadas* hacen referencia a las capas clásicas de las redes feedforward en donde todas las entradas están conectadas con todos los nodos de la salida [30].

Todas las capas en una CNN están usualmente entretejidas entre funciones de activación ReLU y/o sigmoides, en donde todas juntas forman a lo que se conoce como una red neuronal convolucional profunda. En este trabajo, se hace uso de una red neuronal convolucional profunda bajo la arquitectura LeNet-5 descrita en [40], generalmente usada para el reconocimiento de patrones visuales, usando funciones de activación ReLU.

A modo de ejemplo ilustrativo, imaginemos que tenemos una red neuronal convolucional que posee de entrada una imagen tridimensional de $32 \times 32 \times 3$. Y así, digamos esto le da una salida a la red de $28 \times 28 \times 6$ dimensiones, luego de aplicarle los filtros correspondientes a la convolución. Esto nos deja con una red neuronal con $3,072$ unidades en una capa y con $4,704$ unidades en la siguiente capa, y si tuviéramos que conectar cada una de estas neuronas para armar la matriz de peso, la cantidad de parámetros en una matriz de peso sería $3,072$ veces $4,704$ que es alrededor de 14 millones. Optimizar iteraciones con una matriz hessiana con este volumen de parámetros se puede volver ineficiente (para no decir imposible) para SGD, aún habiendo reducido considerablemente el tamaño de la salida gracias a la arquitectura convolucional que va comprimiendo la información. A modo ilustrativo, tomando un ejemplo similar, pero en dos dimensiones:

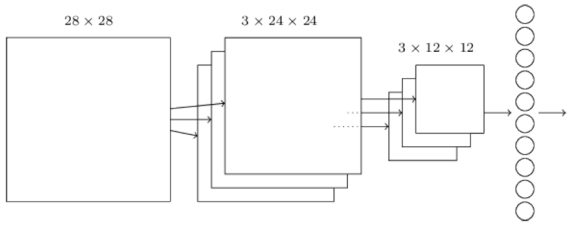

Figura 4.7: Ilustración de una red neuronal convolucional.

En la Figura 4.7 se puede apreciar cómo una imagen se va descomponiendo a medida va entrando en la serie de filtros de una red neuronal convolucional [30]. La capa del medio se corresponde con una de las capas convolucionales, seguida de una capa pooling, en donde se observa se van aplicando filtros bidimensionales para ir reduciendo el tamaño de la capa que entrara en la que estaría totalmente conectada con la salida. En este caso, existen 10 nodos de salida suponiendo la tarea sea la de clasificar dígitos numéricos del 0 al 9. Los filtros van captando distintos tipos de información sobre una determinada imagen para determinar su categoría correspondiente.

Las capas convolucionales son capas totalmente conectadas, con funciones de activación no lineales y utilizadas para la resolución de problemas de reconocimiento de imágenes. Muchas aplicaciones basadas en datos o en objetivos pueden abordarse mediante métodos de aprendizaje profundo. Según la aplicación y los datos, se debe elegir una arquitectura adecuada para el modelo y definir una función de pérdida empírica [2].

En los últimos años, las redes neuronales se han vuelto mucho más profundas, con redes de última generación que han pasado de tener solo unas pocas capas (por ejemplo, AlexNet) a más de cien capas. El principal beneficio de una red muy profunda es que puede representar funciones muy complejas y altamente no lineales. También puede aprender funciones en muchos niveles diferentes de abstracción, desde bordes o patrones específicos (en las capas menos profundas, más cerca de la entrada) hasta funciones muy complejas como el reconocimiento del significado de múltiples patrones en conjunto (en las capas más profundas, más cerca de la salida). Esto hablando específicamente de las redes convolucionales, que se utilizan dentro del reconocimiento de imágenes.

Sin embargo, usar una red más profunda no siempre ayuda. Como ya se mencionaba anteriormente, una gran barrera para entrenarlas son los gradientes que se desvanecen, o el llamado problema del gradiente explosivo [31]. Las redes muy profundas a menudo tienen una señal de gradiente que llega a cero rápidamente o sufren de problemas de desbordamiento numérico, lo que hace que el descenso del gradiente o el descenso de gradiente estocástico sea prohibitivamente lento [30].

Los métodos cuasi-Newton forman una clase alternativa de métodos de primer orden para resolver el problema de optimización no convexa a gran escala en el aprendizaje profundo. Estos métodos, como en SGD, sólo requieren calcular el gradiente de primer orden de la función objetivo. Dado que estos métodos no requieren derivadas directas de segundo orden, son más eficientes que el método de Newton para problemas de optimización a gran escala.

A modo de resumen, para resolver los problemas de optimización con datos de alta dimensión que surgen en las redes neuronales, el descenso de gradiente con *momentum* y sus variantes son los métodos más utilizados en la actualidad, especialmente para el entrenamiento de redes convolucionales. Estas variantes incluyen métodos como RMSprop y Adam, todos los cuales escalan el gradiente estocástico mediante una matriz diagonal basada en estimaciones del primer y segundo momento de los componentes individuales del gradiente. No obstante, se ha realizado un gran esfuerzo para encontrar formas de aprovechar la información de segundo orden para resolver problemas de optimización en el aprendizaje automático. Los enfoques han abarcado toda la gama, desde el uso de un reescalado diagonal del gradiente estocástico, basado en la condición secante asociada a los métodos cuasi-Newton, hasta los métodos de Newton submuestreados, incluyendo aquellos que resuelven el sistema de Newton usando el método del gradiente lineal conjugado [47]. Todas estas estrategias de mejora mencionadas están siendo desarrolladas muy recientemente debido a la gran escala que ha alcanzado el aprendizaje profundo.

Como alternativa al gradiente descendiente, se han implementado algoritmos cuasi-Newton de memoria limitada como L-BFGS con estrategias tanto de búsqueda de línea como de regiones de confianza en problemas de aprendizaje profundo. Estos métodos aproximan la información de la segunda derivada, mejorando la calidad de cada iteración de entrenamiento y evitando la necesidad del ajuste de parámetros (como el tamaño del paso) específicos de la aplicación.

A modo de resumen, se discutieron las principales adversidades y ventajas de los métodos de optimización más ubicuos en el aprendizaje profundo. Se presentó un análisis de trayectoria tomada por algunos de estos algoritmos, y se evidencian las mejoras a SGD que han sido logradas a lo largo del tiempo, siendo algunas de estas mejoras métodos como RMSProp y Adam. En el siguiente Capítulo, se presentará una comparación de los métodos descenso del gradiente estocástico con impulso (SGD), Adam y L-BFGS ante una optimización de una red neuronal convolucional. Se discutirán las preferidas métricas de generalización que presenta el algoritmo cuasi-Newton L-BFGS, así como también los principales hallazgos de posibles mejoras hacia futuras investigaciones.

Capítulo 5

Aplicaciones de los métodos cuasi-Newton en aprendizaje profundo

En este Capítulo se presentará una aplicación del método L-BFGS en una red neuronal, utilizando una estrategia de región de confianza. Se discutirán las ventajas de generalización traídas por las particularidades del método cuasi-Newton al no hacer uso directo de información de segundo orden contra los optimizadores Adam y SGD con impulso, así como también por las variantes de memoria limitada de BFGS discutida en el Capítulo 3. Se realizarán experimentos adicionales sobre el tamaño del lote ("*batch size*") utilizado, el número de lotes con superposición y la variante de memoria m usada. Se trabajará sobre una función de coste de un problema de clasificación de dígitos numéricos escritos a mano (MNIST). Se presentarán resultados robustos a favor de las estrategias de optimización utilizadas por el método cuasi-Newton, así como también posibles líneas de investigación que suponen mejoras adicionales a los planteamientos realizados en este trabajo.

Los algoritmos de aprendizaje profundo a menudo requieren resolver un problema de optimización sin restricciones altamente no lineal. Los métodos más comunes generalmente están restringidos a la clase de algoritmos de primer orden, como el descenso de gradiente estocástico (SGD) [32] con impulso o Adam [34]. El uso de información de curvatura de segundo orden para encontrar direcciones de búsqueda puede ayudar con una convergencia más robusta para problemas de optimización no convexos. Sin embargo, calcular matrices hessianas para problemas a gran escala no es práctico desde el punto de vista computacional [2]. Debido a esto la mayoría de los algoritmos populares que se utilizan en la industria para resolver problemas grandes son de primer orden (como Adam y el descenso del gradiente).

Se ha discutido que el método de Newton utiliza el gradiente y la inversa de la matriz hessiana para encontrar la dirección de búsqueda, a menudo haciendo uso de metodologías de búsqueda de línea para encontrar la longitud del paso a tomar. Sin embargo, el principal cuello de botella en los métodos de segundo orden son los serios desafíos computacionales involucrados en el cálculo de la matriz hessiana, $\nabla^2 L(w)$, para problemas de gran escala en los que no es práctico por el masivo volumen de datos que se maneja. En estas situaciones surge el interés de aplicar otras metodologías de optimización que tomen en consideración la gran cantidad de datos que se maneja, intentando así disminuir el coste computacional sin sacrificar resultados confiables y escalables.

Los métodos cuasi-Newton sólo requieren información de gradiente de primer orden, pero pueden dar como resultado una convergencia superlineal. Esto los convierte en alternativas atractivas a SGD debido a que nos pueden proveer de beneficios similares a los algoritmos de segundo orden, como direcciones de búsqueda más exactas sin tener que aumentar el costo computacional de implementación. El enfoque de memoria limitada Broyden-Fletcher-Goldfarb-Shanno (L-BFGS) es uno de los métodos cuasi-Newton más populares en el aprendizaje profundo, que construyen aproximaciones hessianas definidas positivas siguiendo un esquema conservador en el uso de memoria durante el entrenamiento [2].

5.1. Planteamiento del problema

El problema sobre el cual se ha de comparar a un método cuasi-Newton con sus clásicos predecesores se trata de la clasificación de dígitos numéricos escritos a mano, haciendo uso de la base de datos MNIST [1] [48]. El conjunto de datos del MNIST consta de imágenes de dígitos escritos a mano y etiquetas asociadas que describen qué dígito del 0 al 9 está contenido en cada imagen. Este problema de clasificación es una de las pruebas más elementales en la investigación de aprendizaje profundo y fue originalmente construido y distribuido por Microsoft, la Universidad de Nueva York y Google Research en 1998 [48]. Sigue siendo popular a pesar de ser bastante fácil de resolver para las técnicas modernas. El conjunto de datos del MNIST consta de $70,000$ ejemplos de imágenes escritas a mano

[1] MNIST.

de los dígitos 0–9, con $N = 60,000$ conjuntos de entrenamiento de imágenes (x_i, y_i), y $10,000$ utilizados como conjunto de prueba o validación. Cada imagen x_i es de 28×28 pixeles y cada valor de píxel está entre 0 y 255. Cada imagen x_i en el conjunto de entrenamiento incluye una etiqueta $y_i \in \{0, \ldots, 9\}$ describiendo su clase.

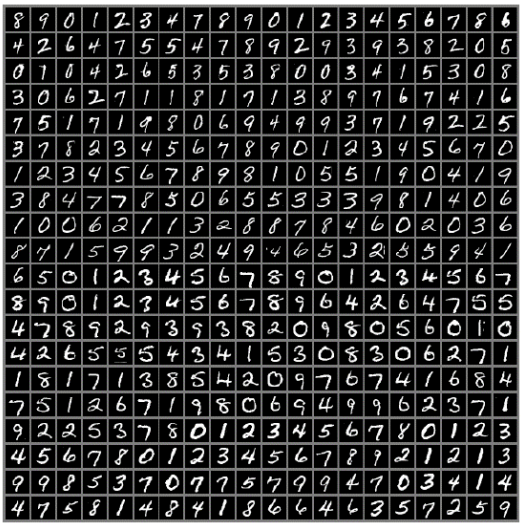

Figura 5.1: Visualización de una parte del dataset MNIST, dígitos escritos a mano del 0 al 9 a ser clasificados por la red neuronal.

Esta red neuronal estará basada en la arquitectura de redes neuronales convolucionales [2], la cual aplicada sobre MNIST tiene $79,510$ parámetros que se deben optimizar.

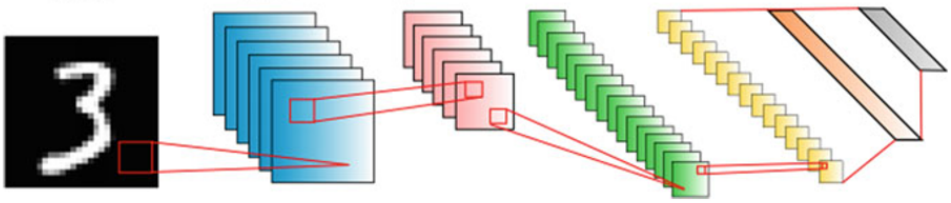

Figura 5.2: Arquitectura LeNet-5.

En la Figura 5.2 se puede observar la distribución de las capas utilizadas en la arquitectura LeNet-5. Consta de una capa de entrada para imágenes de 28×28 pixeles, una capa convolucional con filtros de 5×5, seguida de una función de activación ReLU. A esto le sigue una capa pooling, en donde se toma el valor máximo aplicando un filtro 2×2 sobre la imagen. Esto se repite aumentando la cantidad de filtros a aplicar en la siguiente capa convolucional de 20 a 50, seguida de otra capa de pooling máximo. Para terminar, se agrega una capa totalmente conectada con 500 neuronas, seguida de otra capa totalmente conectada con 10 neuronas, que se corresponde con la cantidad de clases (0-9) que tenemos en el data set. El objetivo de aplicar una red neuronal ante este problema es que la misma pueda generalizar los patrones en las imágenes provistas, y que pueda finalmente predecir la clase de la imagen que se corresponde con el dígito presente en ella [2]. En la Tabla 5.1, se muestran los detalles de la arquitectura LeNet-5:

Capa	**Conexiones**
0: entrada	Imagen 28×28
1	Convolucional, 20 filtros 5×5, función de activación ReLU
2	Max pooling, filtro 2×2
3	Filtros convolucionales, 50 5×5 , función de activación ReLU
4	Max pooling, filtro 2×2
5	Totalmente conectadas, 500 neuronas (sin dropout) seguidas de ReLU
6: salida	Totalmente conectadas, 10 neuronas seguidas de softmax (sin dropout)

Tabla 5.1: Detalles de la arquitectura LeNet-5.

El problema propuesto fue trabajado en el lenguaje de programación Python, versión 3.8.16, con la librería de TensorFlow versión 2.9.2. Se ha trabajado con la plataforma Google Colab, una herramienta desarrollada por Google para otorgar acceso gratuito en la nube a servicios de GPU (Unidad de Procesamiento Gráfico) y TPU (Unidad de Procesamiento Tensorial) a cualquier usuario de Google. Se puede decir que esta plataforma es un paso más avanzado a los cuadernos de Jupyter convencionales, ya que la misma posee integraciones con comandos de terminal desde el portátil y se pueden importar datos desde diversas fuentes remotas como Google Drive y GitHub. No hay que manejar ambientes especializados al trabajar con herramientas de IA como TensorFlow, ya que la plataforma se integra fácilmente con estas librerías de software en sus diferentes versiones, evitando trabajar sobre máquinas virtuales creadas localmente.

Los siguientes gráficos fueron obtenidos al montar el problema de reconocimiento de imágenes en una red neuronal (la arquitectura descrita en la Tabla 5.1), y sucesivamente ir obteniendo los resultados del problema de minimización de la función de coste de ésta. Los sets de datos utilizados se corresponden con uno de entrenamiento y otro de validación. Al correr la data de entrenamiento, los resultados se van guardando en un vector de resultados para luego graficar el vector completo guardado secuencialmente en representación del resultado de la optimización en cada iteración. Los resultados correspondientes a la data de validación fueron de igual forma guardados secuencialmente en un vector aparte, el cual fue representado sobre el mismo gráfico en un color distinto.

La función de pérdida está construida a partir de entropía cruzada con *softmax* entre la variable objetivo y la predicha, que como se mencionó en la Sección 4.3, mide la probabilidad de error en tareas de clasificación discreta en las que las clases son mutuamente excluyentes. Como tenemos un problema de clasificación multiclase, cada entrada pertenece exactamente a una clase, y la función de pérdida lo que hace es convertir a una distribución de probabilidad cada una de las clases y minimizar la diferencia de estas contra las clases reales (un concepto similar a la minimización de la divergencia Kullback-Leibler, que es una medida de qué tan disímil es una distribución de probabilidad ante otra [49]). La representación matemática del cálculo de la pérdida, construida a partir de entropía cruzada con *softmax*, se expresa mediante la separación del cálculo de la pérdida para cada una de las clases asociadas a una observación y se suman los resultados, tal y como se observa en la siguiente expresión:

$$L = -\sum_{c=0}^{M} y_{o,c} \log(p_{o,c}). \tag{5.1}$$

Notar que M en (5.1) representa el número de clases, en este caso 10 dígitos, mientras que p es la probabilidad predicha de la observación o de la clase c [28].

El problema planteado en esta Sección ha sido probado sobre tres algoritmos de optimización, con el objetivo de hacer evidente las métricas favorables de los métodos cuasi-Newton a ser discutidas. Estos métodos son el optimizador Adam [34], el algoritmo del descenso de gradiente estocástico (SGD) con impulso [32] y uno de los métodos cuasi-Newton de memoria limitada, L-BFGS [18]. Para cada uno de los métodos se presentan dos gráficas referentes al desempeño de los algoritmos bajo la tarea de optimización en cuestión, que sería la de minimizar la función de pérdida bajo la tarea de correctamente clasificar dígitos y la de maximizar la precisión de la clasificación. Estas dos gráficas se corresponden con la pérdida obtenida bajo cada iteración del algoritmo, así como también la precisión obtenida bajo la tarea de clasificación en representación de cada una de las iteraciones en el eje horizontal. Los resultados de cada una de las corridas se iban guardando en un archivo pickle, desde donde se cargaron los datos y crearon las gráficas.

El código utilizado para realizar las gráficas a continuación puede ser encontrado en el enlace puesto a disposición en el Anexo (Capítulo 7), a fines ilustrativos y de reproducción del trabajo realizado. El análisis de convergencia de los resultados obtenidos será discutido a profundidad en la Sección 5.2.

5.2. Análisis de la convergencia

El problema de la clasificación de dígitos es inherentemente complejo. Se presentan los resultados a continuación de cada uno de los 3 métodos probados. Cada uno de los gráficos presenta una leyenda como indicador hacia el set de datos utilizado: una métrica se corresponde con el set de datos de entrenamiento y otra con el set de validación, para medir el desempeño del algoritmo en cuestión. Los resultados asociados a cada una de las métricas obtenidas se muestran con colores distintos: el verde representa la data de entrenamiento, mientras que el azul la data de validación. La ubicación horizontal de cada uno de los vectores graficados se corresponde con el número de época o iteración aplicado a la resolución del problema, que serían 100 épocas en cada una de las instancias probadas.

Matemáticamente, una métrica perfecta de pérdida se correspondería con una lo más cercana al cero posible. Por otro lado, una métrica ideal para la precisión de cada algoritmo se correspondería con una lo más cercana al 1 posible. Sin embargo, en el contexto de las redes neuronales, como se mencionaba en la Sección 4.3, este no necesariamente es el caso. Esto es debido a que el problema se extrapola hacia un set de datos de validación, en donde no debería abundar el sobreajuste, de manera tal que el algoritmo pueda luego generalizar hacia datos nunca antes vistos.

Dentro del código utilizado, para obtener los resultados de los algoritmos de optimización más utilizados en el aprendizaje profundo, se utilizaron los esquemas ya definidos en las librerías de software como TensorFlow y Keras, estos algoritmos siendo Adam y SGD. Por otro lado, el algoritmo cuasi-Newton se implementa a partir de un bucle sobre la función de pérdida, construida a partir de la entropía cruzada con softmax. El código primero carga el conjunto de datos MNIST y preprocesa los datos al normalizar los valores de píxeles. Luego define el modelo LeNet-5 utilizando la clase *Sequential* de *tf.keras*, que consta de una serie de capas convolucionales y totalmente conectadas. A continuación, el código define la función de pérdida y el optimizador. La función de pérdida es la pérdida de entropía cruzada categórica y el optimizador es el algoritmo L-BFGS. El algoritmo L-BFGS tiene varios hiperparámetros que se definen en el código, incluido el:

1. Parámetro de memoria limitada, *m*
2. Tamaño del lote (minibatch size)
3. Cantidad de lotes en superposición
4. La estrategia de optimización (búsqueda de línea, región de confianza)
5. Gradiente completo, si el gradiente se ha de calcular con la data completa en cada iteración
6. Máximo número de iteraciones.

En este caso, la función de pérdida es la entropía cruzada categórica dispersa y el optimizador es el algoritmo L-BFGS. Luego, el modelo se compila utilizando estos objetos de pérdida y optimizador. Dentro del bucle, la pérdida y la precisión de la iteración actual se guardan e imprimen mediante una función. A continuación, se evalúa el gradiente de la función de pérdida y se aplican las ecuaciones de actualización de L-BFGS para calcular la dirección de búsqueda, dependiendo de la estrategia de optimización que se esté usando. El ciclo continúa hasta que la norma del gradiente está por debajo de cierta tolerancia o se alcanza el número máximo de iteraciones.

Finalmente, el modelo se entrena usando el método de ajuste (fit) y el rendimiento se evalúa en el conjunto de prueba usando el método de evaluación (test).

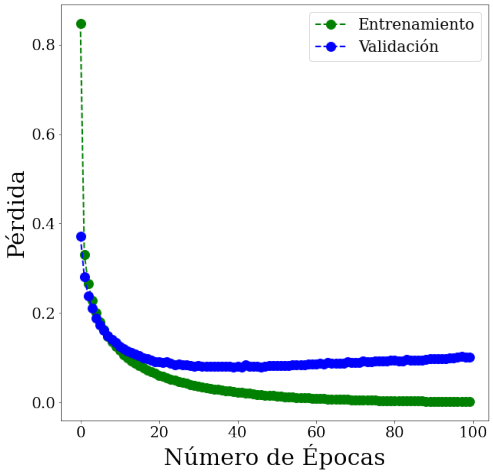

Figura 5.3: Pérdida Adam

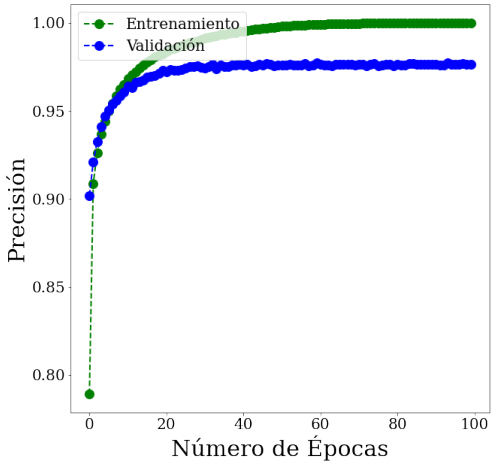

Figura 5.4: Precisión Adam

En las Figuras 5.3 y 5.4 se presentan los resultados obtenidos utilizando el optimizador Adam. Se puede observar cómo el método de Adam diverge tanto en el gráfico de la pérdida como en la precisión cuando comparamos los resultados de validación (línea azul) de los resultados obtenidos con el conjunto de datos de entrenamiento (línea verde). Por ende, se verifica que Adam es el algoritmo con la mayor propensión hacia el sobreajuste de los analizados. Los resultados de entrenamiento son bastante robustos y se acercan hacia la métrica perfecta de un 100 %, sin embargo, cuando observamos los resultados de la data de validación estos se comportan de manera distinta obteniendo métricas más bajas. Esto quiere decir que el nivel de generalización del algoritmo es bajo, por lo que existe una alta varianza aprendida desde los datos de entrenamiento.

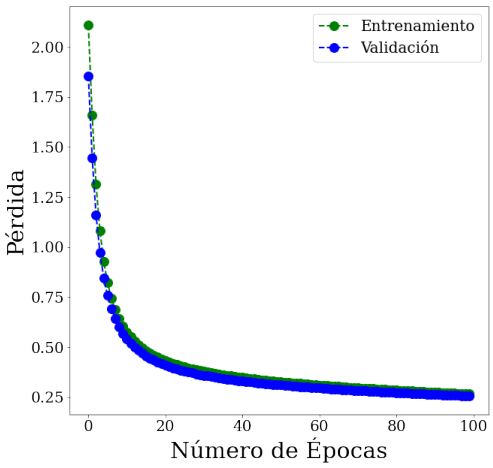

Figura 5.5: Pérdida SGD

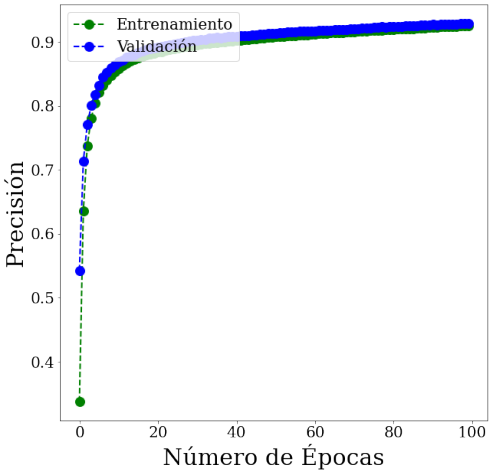

Figura 5.6: Precisión SGD

En las Figuras 5.5 y 5.6 se presentan los resultados obtenidos utilizando el optimizador del descenso de gradiente estocástico con impulso (SGD). Se puede observar cómo los resultados obtenidos no se acercan a las métricas obtenidas con el optimizador Adam (Figura 5.3 y Figura 5.4), debido a que la métrica de la pérdida en SGD se acerca a un 26 % y la de precisión a un 92 %. De esto podemos inferir que al algoritmo le haría falta una mayor cantidad de iteraciones para poder alcanzar precisiones similares a las obtenidas con los otros algoritmos, siendo por ende el menos robusto. Sin embargo, el mismo obtiene niveles de generalización superiores a las de Adam, debido a que las métricas obtenidas tanto en el entrenamiento como en la validación son casi idénticas y por ende hay un nivel menor de varianza obtenida (mayor nivel de generalización).

Puede parecer que solo existe una línea en las imágenes, sin embargo, esto se debe a que la trayectoria tomada por los resultados de la optimización dentro del set de datos de entrenamiento y el set de datos de validación es casi idéntico, por lo que la línea de ambos se superpone a sí misma y a simple vista sólo se logra distinguir uno de los dos vectores de resultados. Sin embargo, si se observa más de cerca se logra identificar otro vector de un color distinto, graficado casi por debajo del vector de resultados más visible, que sería el vector de resultados de los datos de validación. Este es el caso debido al orden en el que se graficaron los datos, primero graficando el vector de resultados de la data de entrenamiento y luego graficando el vector de resultados de la data de validación.

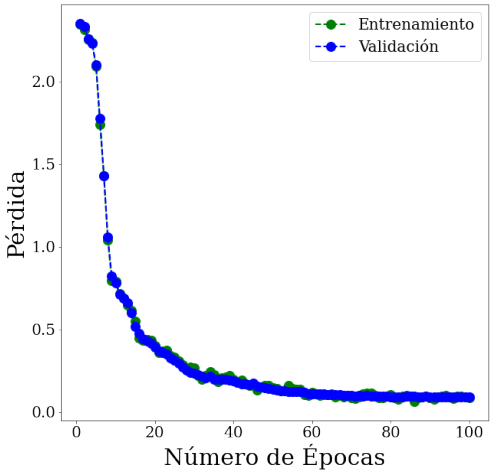

Figura 5.7: Pérdida L-BFGS

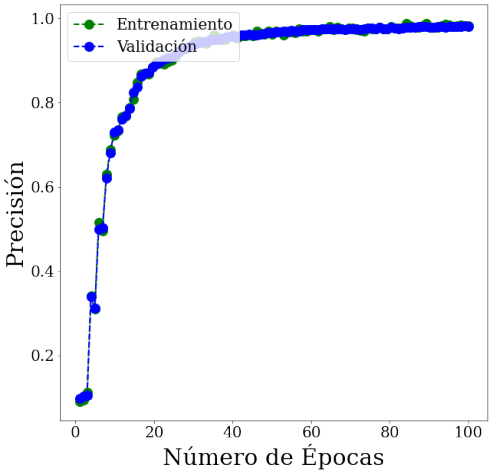

Figura 5.8: Precisión L-BFGS

En las Figuras 5.7 y 5.8 se presentan los resultados obtenidos utilizando el algoritmo cuasi-Newton de memoria limitada L-BFGS. Los resultados muestran una convergencia robusta con características de generalización preferidas, así como un tiempo de entrenamiento mejor al de sus contrapartes, Adam y SGD con impulso. Esto debido a que el mismo obtiene las mejores métricas de generalización bajo la misma cantidad de iteraciones, haciéndolo el más robusto de los tres probados. En consecuencia, dadas las superiores métricas alcanzadas bajo la misma cantidad de iteraciones, los resultados presentan una ventaja inclinándose hacia los métodos cuasi-Newton. Las métricas utilizadas en L-BFGS en los resultados mostrados en las Figuras 5.7 y 5.8 se corresponden con un tamaño del lote del algoritmo de $1,000$, un tamaño del lote en superposición de $n = 50$ y una variante de memoria limitada $m = 25$. Se puede observar cómo al igual que en SGD (Figuras 5.5 y 5.6), L-BFGS tiene resultados de entrenamiento y validación casi idénticos, sin embargo, en este caso las métricas alcanzadas son superiores a SGD.

El esquema de memoria no limitada (BFGS llano) también fue probado ante este mismo conjunto de datos, y se reportan tiempos de entrenamiento mucho más altos que las contrapartes SGD y Adam. Esto sucede ya que la información de curvatura se va haciendo más y más compleja de calcular cuando no tenemos el parámetro de memoria limitada, m. No obstante, esto sigue siendo un modelo bastante robusto y que llegando al final de las iteraciones usadas (100), puede proveer de resultados mucho más precisos que las contrapartes, pero no práctico ante problemas a gran escala, razón por la cual se aplicó el esquema de memoria limitada (L-BFGS). De esta forma, aplicando el parámetro m de la memoria limitada, nos evitamos el tiempo computacional superior al no tener que hacer el cálculo de las matrices hessianas con toda la información de curvatura. Para este parámetro de memoria variante m, se consideró la memoria del conjunto $m \in \{10, 15, 20, 25, 50\}$, y se reporta el mejor desempeño con respecto a los diferentes tamaños de memoria siendo $m = 25$, con un máximo número de iteraciones (en todos los casos) en 100. Más adelante se presentarán resultados experimentales que obtenemos al variar estos parámetros, y veremos que esto arroja informaciones adicionales sobre los métodos cuasi-Newton.

Este es un problema de clasificación supervisado relativamente pequeño. Sin embargo, como se desprende claramente de las Figuras 5.7 y 5.8, los métodos cuasi-Newton de baja memoria son competitivos con las variantes clásicas en términos de ambas métricas de entrenamiento y validación. También se puede observar que, en las etapas iniciales de la optimización, los métodos cuasi-Newton con memoria limitada superan a sus contrapartes clásicas ya que se observa cómo la aceleración hacia el punto de convergencia es superior, acercándose hacia el punto esperado en menos tiempo y de forma más estable.

Los métodos de primer orden (SGD con impulso y Adam) son competitivos contra L-BFGS, sin embargo, requieren un presupuesto computacional superior para lograr un nivel de generalización alto o similar al de L-BFGS. Con L-BFGS, usando el mismo presupuesto computacional, nos acercamos más a una pérdida de cero y una precisión de 100 % de una manera más robusta y estable. Los resultados en la data de validación también muestran que L-BFGS es mejor en poder generalizar ante datos que no ha visto anteriormente, y que pueden proveer resultados más confiables. Dentro de los 3 métodos probados, el cuasi-Newton L-BFGS es el que presenta las métricas de generalización dentro del set de datos de validación más favorables. Esto quiere decir que el algoritmo cuasi-Newton es el que mejor logra el objetivo de poder generalizar de manera robusta cuando se comprueban los resultados ante datos no vistos anteriormente.

Por otro lado, se reportan altos costos computacionales asociados a la implementación de los métodos cuasi-Newton bajo un esquema de búsqueda de línea sobre α_k. Además, cuando las condiciones de Wolfe, específicamente la condición de curvatura, no se cumplen para $\alpha_k \in (0,1]$, la matriz L-BFGS puede no permanecer definida positiva y la actualización se vuelve inestable. Por otro lado, si se rechaza la dirección de búsqueda para preservar la definición positiva de las matrices L-BFGS y seguir buscando una opción viable, el progreso del aprendizaje bajo este se detenía al volverse extremadamente lento. Por esta razón se implementó L-BFGS bajo un esquema de región de confianza, logrando así épocas mucho más rápidas, ya que el problema de minimización con restricciones que este implica es mucho menos engorroso porque no hay condiciones que se deban cumplir de manera simultánea como es el caso de estrategias de búsqueda de línea, como vimos en las Ecuaciones (3.4) y (3.5), que serían las condiciones de Wolfe.

Vale la pena señalar que a medida que aumentamos el tamaño de las redes neuronales, el rendimiento de estos métodos mejora ante sus contrapartes. Por otro lado, los métodos cuasi-Newton tienen un mejor rendimiento en este problema complejo, aunque pequeño, principalmente debido al uso de información de curvatura provista por el cálculo alternativo de la matriz hessiana. Entre los resultados informados, el método cuasi-Newton de memoria limitada supera a los métodos clásicos, debido al uso de información de curvatura local más reciente en las actualizaciones.

A continuación, se presenta una Tabla de los resultados obtenidos bajo los tres algoritmos analizados al final de las 100 iteraciones planteadas:

Método	**Precisión Entrenamiento**	**Precisión Validación**	**Pérdida Entrenamiento**	**Pérdida Validación**
Adam	1.0000	0.9763	0.0015	0.1005
SGD	0.9260	0.9292	0.2660	0.2565
L-BFGS	0.9825	0.9820	0.0362	0.0569

Tabla 5.2: Resultados experimentales.

Específicamente observando los resultados obtenidos en la Tabla 5.2, se puede determinar que los niveles de generalización de L-BFGS

son mucho más estables y robustos que los de Adam y SGD (Figura 5.5 y Figura 5.6). La distancia entre la precisión de entrenamiento y validación en SGD y L-BFGS es similar, sin embargo, la métrica obtenida en L-BFGS es superior. El objetivo de medir la generalización del modelo en un set de datos distinto al entrenamiento es poder medir qué tan acertado ha sido el aprendizaje sobre los parámetros optimizados a los datos durante el entrenamiento, pero tampoco es llegando a la perfección. Se espera tener un margen de error tanto en el set de entrenamiento como en el de validación, debido a que si la precisión obtenida en el entrenamiento es del 100 % muy probablemente estaríamos realizando un sobreajuste del modelo. Este problema surge cuando la varianza en el entrenamiento es tan alta que el modelo logra incorporar de forma abrupta todas las variaciones en el set de datos de entrenamiento (alta varianza), aprendiéndolos de memoria, pero cuando nos enfrentamos a datos nuevos, los parámetros entrenados para resolver el problema a la perfección no logran acertar del todo en el set de validación. Este es el caso que estamos viendo con Adam, muy sutilmente, pero no menos entendible dado el tamaño del problema que estamos tratando, en donde la precisión en el entramiento es de un 100 % y la de la validación casi 3 puntos menor.

5.3. Experimentos adicionales sobre L-BFGS

L-BFGS logra optimizar justo lo necesario sobre el set de entrenamiento de forma tal de no embotellarse las muestras obtenidas y poder en consecuencia generalizar mejor de cara hacia el set de validación. Se puede observar en la Tabla 5.2 cómo la pérdida de validación más baja es la de L-BFGS. Es decir, L-BFGS presenta mejores propiedades de generalización y menores tendencias hacia desarrollar un sobreajuste sobre los datos de entrenamiento. Este problema también es menos evidente en el caso de SGD. Sin embargo, las métricas obtenidas sobre la pérdida (tanto en el entrenamiento como en el set de validación) no son las más acertadas en SGD, obteniendo al final de las 100 iteraciones una pérdida de un 26 % (siendo en teoría el objetivo de la optimización obtener una pérdida de 0 %). Esta sería la métrica de pérdida más alta obtenida, es decir los peores resultados se corresponden con el esquema del descenso de gradiente estocástico con impulso.

Una técnica implementada dentro de las arquitecturas de redes convolucionales es el max pooling, que busca reducir la representación de una región de información en una matriz para reducir la dimensionalidad de las entradas y suavizar los patrones de características. Esto se logra al tomar el valor máximo dentro de una región específica. El resultado es una representación más compacta de los datos de entrada, lo que también reduce el tiempo de entrenamiento y la complejidad del modelo. Esto nos provee de una representación abstracta de zonas ricas en información, y que pueden estar predispuestas al ruido y, por ende, estaríamos predisponiendo a la red neuronal a cometer un sobreajuste. Dentro del método L-BFGS presentado anteriormente, se establece un parámetro que es el que especifica el tamaño del lote en superposición en 50, es decir, el tamaño del lote que podrá formar parte de un muestreo con reemplazo de la muestra original. Este es el muestreo que luego pasa a formar parte del proceso del max pooling. Se corrieron varias pruebas variando este parámetro a fines de observar el comportamiento del algoritmo cuasi-Newton, y luego se graficaron todas las corridas en un sólo gráfico especificando el parámetro utilizado en la leyenda. Los gráficos mostrados a continuación solamente representan la corrida con la data de entrenamiento dentro del algoritmo cuasi-Newton, ya que la representación gráfica de la información corrida sobre la data de validación se comporta de manera casi idéntica. A continuación, se presentan las corridas cambiando el parámetro del tamaño del lote en superposición, dentro del conjunto $n = \{5, 10, 20, 50, 100\}$.

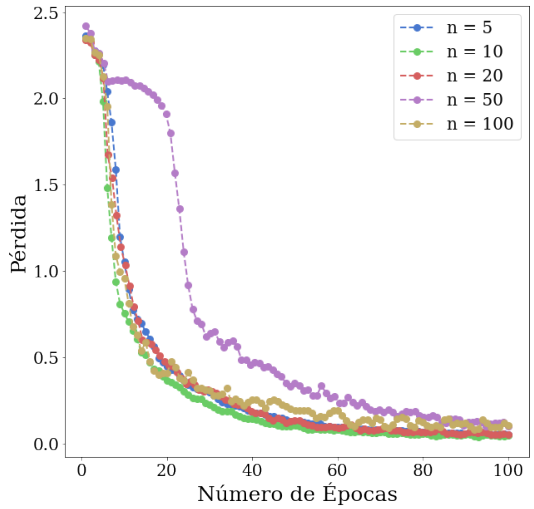

Figura 5.9: Superposición del lote: Métricas de pérdida.

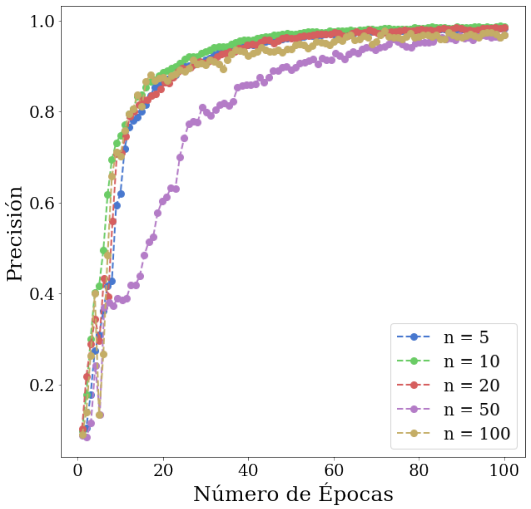

Figura 5.10: Superposición del lote: Métricas de precisión.

Se verificó, para este problema en particular, que no existe ninguna correlación positiva con el parámetro del tamaño del lote en

superposición y el desempeño del algoritmo L-BFGS, tal como se aprecia en la Figura 5.9 y Figura 5.10. Esto debido a que, en base al experimento realizado, se identificó una tendencia de que a mayor es el tamaño del lote que se encontraría en superposición al momento de la selección de la muestra del nuevo gradiente, más pobre es el nivel de generalización del algoritmo. Esto quiere decir que, mientras más pequeño es este parámetro, al momento de seleccionar la parte del lote matriz que estaría entrando en la nueva iteración, y por ende en el nuevo parámetro de memoria limitada para poder recalcular las matrices que representan las aproximaciones de segundo orden, mejor es el desempeño del algoritmo. Esto se debe específicamente a la base de datos con la que estamos trabajando, ya que se puede decir que este problema no implica mucha información espacialmente compleja, o que se debería de estar identificando desde puntos de vista que implican una tercera dimensión. Como estamos trabajando con data de dos dimensiones, y no con imágenes en dos dimensiones con información en representación de más de dos dimensiones, esto no implicaría que hemos de tener objetos con fuentes de ruido adicionales en la imagen como un semáforo (dada la tarea de clasificar vehículos, por ejemplo), que tengamos problemas de luz, o que simplemente haya personas transitando que puedan interferir con el desempeño del algoritmo. Para la tarea en cuestión, simplemente se espera poder identificar los rasgos de un número escrito a mano, que puede o no variar sutilmente a nivel de estructura, y debido a que la información no representa una complejidad significativa a nivel espacial en la imagen, a nivel de cada píxel, se determinó que el tamaño del lote en superposición no aporta ninguna mejora a nivel del desempeño del algoritmo al final de que este haya terminado el entrenamiento. Sin embargo, si tenemos un problema que representa información que viene de una imagen en dos dimensiones, pero representando un espacio de más de dos dimensiones, el tamaño del lote en superposición nos puede traer una mejora significativa, ya que nos proveería de una manera adicional para reducir el ruido y la propensión al sobreajuste de nuestro modelo y poder mantener las favorables métricas de generalización de L-BFGS ante los demás algoritmos de optimización utilizados dentro del aprendizaje profundo. También, este parámetro nos puede ayudar si pudiéramos vernos en la necesidad de poder identificar más de un objeto en una sola imagen o, por ejemplo, que el objeto se encuentre en espacios pequeños, en la esquina de la imagen, o que se encuentre superpuesto por otro objeto delante. En estos debidos casos, el parámetro del tamaño del lote en superposición sí nos pudiera ayudar, debido a que las matrices extraídas a través del max pooling nos estarían trayendo información digamos más "redundante" (muestreo con reemplazo), pero representada de diferentes formas, en determinados espacios de una imagen. Sin embargo, estaríamos tratando con información inherentemente más compleja de analizar, y por ende necesitaríamos de información adicional sobre estos puntos, apoyándonos en este parámetro para poder extraer información más meticulosa de cada uno de los grupos tomados, lo que al final sí nos pudiera estar ayudando a mejorar el desempeño del algoritmo. Sin embargo, para nuestro caso, se identificó que mientras menor es este parámetro, $n = \{5, 10\}$, mejor se comportan las métricas de precisión y pérdida en el algoritmo cuasi-Newton.

Una red convolucional podría fácilmente no aprender a extraer características ricas (que generaría un modelo más generalizable), sino que aprende a extraer características que solo son buenas para clasificar una cierta cantidad de ejemplos en el conjunto de entrenamiento. Entonces, cuando tenemos regiones de agrupación que no se superponen, la información espacial particular se pierde rápidamente y la red sólo identifica los valores de los píxeles dominantes (los que contienen las zonas del max pooling, por ejemplo). Esto aún proporcionaría representaciones jerárquicas de la información muestreada, pero casi siempre estarían dominadas por las regiones más fuertes de la imagen, que luego se propagan a lo largo de la red. Efectivamente, esto crea un sesgo en el aprendizaje que, a su vez, provoca una propensión hacia el sobreajuste. Con regiones superpuestas, hay menos pérdida de información espacial circundante. Sin embargo, la no aplicación de esta no siempre causa un problema en la práctica y las regiones de agrupación superpuestas solo mejoran marginalmente los resultados.

Otro hallazgo dentro de los experimentos realizados es que la variación del parámetro del tamaño de lote (*batch size*) dentro de L-BFGS no tiene ningún efecto particular en los niveles de generalización del algoritmo cuasi-Newton (en la data de validación). Al igual que en Adam y SGD, en L-BFGS este parámetro nos brinda de una forma de decidir el tamaño de la matriz con la que estaríamos trabajando a medida vamos iterando. Inicialmente, para cada uno de los algoritmos se aplicaron valores para el tamaño de lote de este conjunto $Lote = \{5, 50, 100, 500, 1000\}$, y se notó que el tiempo de entrenamiento aumenta al disminuir el tamaño del lote. No obstante, también se verificó que mientras más grande es el tamaño del lote utilizado, más rápido converge el algoritmo a niveles de precisión favorables. Esto nos dice que L-BFGS trabaja mejor con grandes volúmenes de información, y que espera siempre hacerlo de esta forma. Las actualizaciones de las aproximaciones de los elementos de segundo orden parecen hacerse cada vez más difíciles cuando tenemos una matriz de información pequeña, y adicionalmente, tenemos información limitada de la iteración anterior, debido al parámetro de memoria limitada m. Esto hace un hincapié importante en la verdadera utilidad que traen los métodos cuasi-Newton al mundo del aprendizaje profundo, y es que estos algoritmos no serían eficientes si se piensa trabajar con poca información. Las variantes de memoria limitada tienen el efecto de necesitar un alto volumen de información debido a que ya de por sí el algoritmo está estructurado para trabajar sobre este tipo de problemas. En base a esto, el algoritmo presupone tener la necesidad práctica de aplicar un parámetro de memoria limitada a los cálculos de las aproximaciones de elementos de segundo orden. Por ende, hace sentido que se ralentice el proceso de entrenamiento si las matrices que se van construyendo con cada iteración tienen poco volumen de información. Esto provoca que la limitada información de las matrices de primer orden de iteraciones anteriores haga el cálculo de elementos de segundo orden más complejo. En estos casos, las regiones de confianza se van haciendo más pequeñas en cada iteración, lo que hace que el movimiento de la trayectoria del algoritmo sea más lento. A continuación, se muestran los resultados obtenidos al correr varios experimentos variando el parámetro del tamaño del lote, y luego graficando todos los resultados en un mismo gráfico. Al igual que en el experimento anterior, los gráficos mostrados sólo se corresponden con la data de entrenamiento, debido a que los resultados obtenidos dentro de la data de validación son casi idénticos a los de entrenamiento.

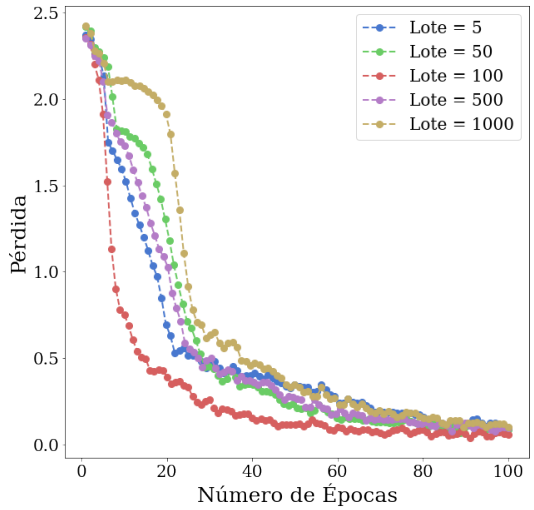

Figura 5.11: Tamaño del lote: Métricas de pérdida.

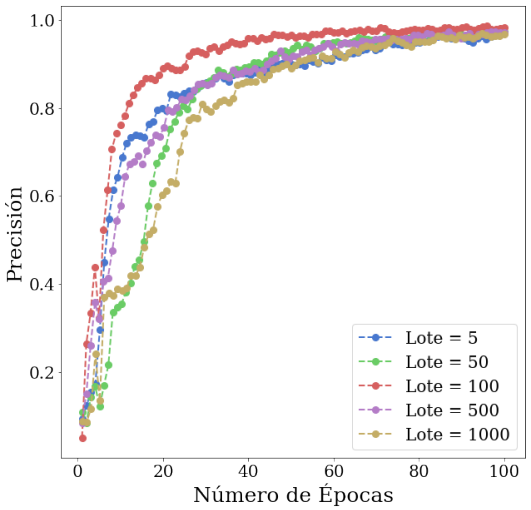

Figura 5.12: Tamaño del lote: Métricas de precisión.

En la Figura 5.11 y Figura 5.12 se observa que los mejores resultados se obtienen en el parámetro probado más alto, cuando el tamaño del lote es igual a 1000. También se puede observar que en ambas figuras no siempre es el tamaño del lote mayor el que se acerca a obtener los mejores resultados. El peor de todos los probados se corresponde con el tamaño del lote igual a 50, sin embargo, el segundo mejor se corresponde con 5. Esto nos indica que el hecho de que este haya sido el punto en donde el entrenamiento se haya hecho más lento no quiere decir que es donde el desempeño haya sido menos favorable, y que por ende debe haber un punto óptimo del tamaño del lote que maximice los niveles de generalización y que haga el entrenamiento más rápido.

El diseño de metodologías para encontrar este punto óptimo del tamaño del lote es una potencial próxima investigación dentro del campo de los métodos cuasi-Newton en el aprendizaje profundo, debido al alto impacto que inicializar este parámetro de manera óptima puede traer en las métricas a ser evaluadas. Disminuir este parámetro también tenía un efecto de ralentizar el entrenamiento, sin embargo, los resultados no se veían tan afectados debido a las superiores propiedades de generalización del algoritmo. Como este se empleó bajo un esquema de región de confianza, se pudo asegurar que el entrenamiento no se iba a detener por tener alguna actualización de la inversa de la matriz hessiana inestable o no definida positiva.

Otra posible dirección para una investigación es aplicar la misma idea para encontrar de forma adaptativa el tamaño del parámetro de memoria limitada m que utilizan los algoritmos cuasi-Newton: adaptando el mismo al tamaño de las matrices asociadas a cada iteración. De esta manera, se estarían optimizando los recursos disponibles de acuerdo con el volumen y/o tiempo estimado para calcular la matriz de aproximación de segundo orden de una iteración dada. Dado que el conjunto de pasadas iteraciones se ha de ir haciendo más y más extendido a lo que nos acercamos hacia el final de las iteraciones y por ende tendríamos más información disponible cada vez menos relevante, un posible punto de partida es que este parámetro tenga un decaimiento exponencial del volumen de información de curvatura a ir guardando en la memoria, con la restricción de que no se comprometa la fiabilidad de la aproximación de la matriz asociada a la iteración y el tiempo de entrenamiento. A continuación, se presentan los resultados obtenidos en el experimento sobre L-BFGS al ir cambiando el parámetro de memoria limitada m, y luego graficando todo en un mismo gráfico. Los resultados mostrados se corresponden con una estrategia de optimización aplicada de región de confianza (RC), cambiando el parámetro m del alcance de la memoria limitada de acuerdo con el siguiente conjunto: $m = \{5, 10, 25, 100, 1000\}$.

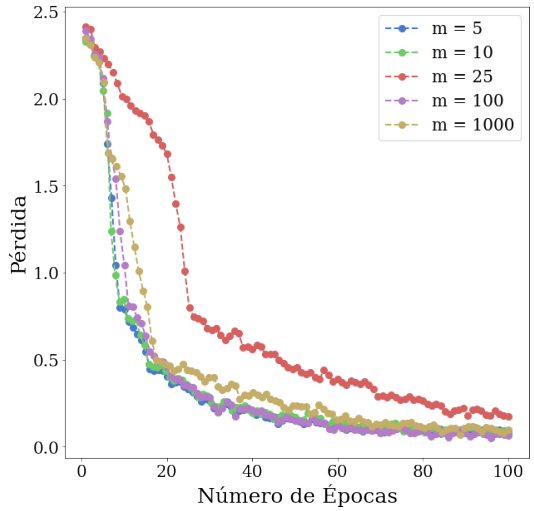

Figura 5.13: Parámetro *m* con RC: Métricas de pérdida.

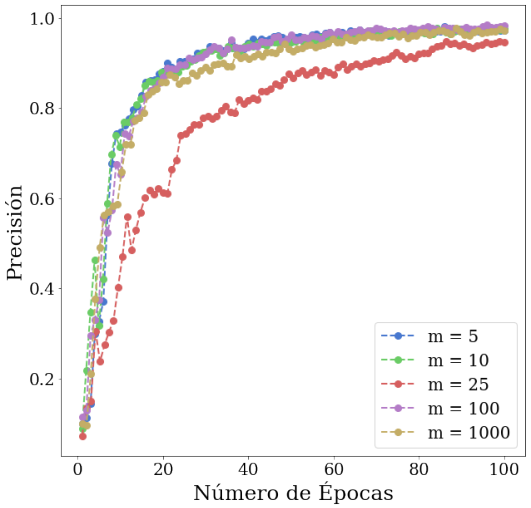

Figura 5.14: Parámetro *m* con RC: Métricas de precisión.

Para el caso de L-BFGS, como se observa en la Figura 5.13 y Figura 5.14, aumentar el parámetro *m* no presenta cambios significativos sobre los resultados obtenidos de la optimización, salvo aumentar el tiempo para llegar al final de las iteraciones, ya que se estaría utilizando información de curvatura de mayor alcance dentro del cálculo de cada una de las aproximaciones de la inversa del hessiano. También se corrió el mismo experimento bajo una estrategia de búsqueda de línea, solo para obtener resultados más inestables tanto en la data de entrenamiento como en la data de validación, debido a la carga computacional agregada de tener que validar dentro de cada iteración que las condiciones de Wolfe se debían de cumplir. Por esta razón, los resultados expuestos se corresponden con regiones de confianza que van variando dentro de cada iteración para determinar la dirección de búsqueda óptima. Sin embargo, las métricas de precisión con la data de validación bajo estrategias de búsqueda de línea tuvieron un comportamiento inestable cuando el parámetro *m* es bajo. Esto nos indica que determinar las condiciones necesarias para satisfacer las condiciones de Wolfe se hace más inestable cuando tenemos un bajo volumen de información de los elementos de primer orden de iteraciones anteriores. Por esta razón también, se recomienda siempre utilizar estrategias de regiones de confianza para aplicar estos algoritmos.

Expuesto todo lo anterior, otra posible línea de investigación es el planteamiento de todas las restricciones dentro del problema de optimización con un método cuasi-Newton bajo regiones de confianza como un problema de optimización distribuida, en el que se definan múltiples objetivos a los cuales optimizar. Estos puntos vendrían siendo la pérdida, el parámetro de memoria limitada, y el parámetro de tamaño del lote con superposición. Esto se haría aplicando una restricción sobre los resultados a partir de cada uno de los objetivos planteados.

Se ha demostrado que los métodos cuasi-Newton se utilizan en Aprendizaje Profundo para mejorar la convergencia de los modelos de redes neuronales. Estos métodos aprovechan la estructura de la función de costo para mejorar el rendimiento y reducir el riesgo de sobreajuste. Los métodos cuasi-Newton también son útiles para evitar los límites numéricos que se presentan al usar el Gradiente Descendente o el método de Newton. En general, los métodos cuasi-Newton de memoria limitada superan a sus variantes clásicas en términos de niveles de generalización y menor número de cálculos. Debemos tener en cuenta que el objetivo de estos experimentos no es obtener un mejor rendimiento que Adam o SGD, sino que el rendimiento de estos pueda verse como un punto de referencia. Para el problema de MNIST, el método Adam puede lograr una precisión superior a la de L-BFGS, pero no sin obtener una varianza que llega al sobreajuste en el entrenamiento. El comportamiento con el set de datos de validación no parece iniciar a asemejarse a la métrica obtenida en el entrenamiento al aumentar el número de iteraciones en el método SGD, situación que no ocurre al aplicar un algoritmo cuasi-Newton. Además, se reporta en [10] que en otros métodos cuasi-Newton como el S-LSR1, en un entorno distribuido, el tiempo para realizar una iteración (una época) es significativamente menor que el tiempo para realizar una época de Adam. S-LSR1 tiene un mejor rendimiento que L-BFGS posiblemente debido a la utilización de curvatura negativa en las actualizaciones del hessiano [10]. De hecho, también ya se han diseñado mejoras a estos métodos básicos cuasi-Newton para mejorar niveles de eficiencia computacional al incrementar el parámetro de memoria limitada *m*: TRMinATR [2].

Los resultados indican que existe potencial para mejoras adicionales, identificando estrategias de búsqueda que simplifiquen cada época, que daría como resultado algoritmos de mejor rendimiento, sin sacrificar los resultados robustos obtenidos. No obstante la posibilidad de mejoras, los resultados obtenidos de los métodos cuasi-Newton ante un problema de clasificación de imágenes muestran que estos pueden fácilmente superar a sus variantes clásicas tanto en la dimensión de entrenamiento como en la de validación, obteniendo resultados mucho más potentes bajo el mismo número de iteraciones.

Capítulo 6

Conclusiones

6.1. Conclusiones

Se ha demostrado que en el método de Newton-Raphson necesitamos computar la matriz hessiana, para luego computar la inversa de esta matriz (también conocida como la matriz jacobiana). En sistemas de ecuaciones donde esta matriz es muy grande, esta evaluación de la inversa de la matriz jacobiana puede llegar a ser bastante demandante a nivel computacional. La idea general de los métodos cuasi-Newton es reemplazar esta matriz jacobiana (su inversa), por una aproximación de ésta. Esta matriz representa la dirección de búsqueda del método, y la misma será evaluada en cada punto o vector obtenido dentro de cada iteración. Se pueden usar técnicas para aproximar la matriz hessiana de una función, basándose en pequeñas perturbaciones alrededor de un punto. Pudiéramos obtener de esta forma una necesidad a mayor cantidad de iteraciones para la convergencia, pero bajo una reducida computación por cada iteración.

Otra forma, que es la utilizada por los métodos cuasi-Newton, es sacar provecho de la fórmula de la secante para hallar una aproximación de la matriz hessiana. Esta fórmula tomará ventaja de los valores de x_k y $\nabla f(x_k)$ que ya hemos calculado en iteraciones anteriores y actuales. Mientras avanzamos en la resolución del problema, los tamaños de paso se irán haciendo cada vez más pequeños mientras nos acerquemos a una solución buscada.

Se ha revisado la literatura de los métodos cuasi-Newton y hemos demostrado que estos utilizan enfoques para aproximar la matriz hessiana sin calcular o almacenar la verdadera matriz hessiana. Esto permite aumentar significativamente la escala del problema, o el tamaño de la data utilizada, dando resultados con tasas de convergencia considerablemente más altas, en áreas que están siendo altamente investigadas dentro de las ciencias de la computación y de datos. Se ha comprobado que los métodos cuasi-Newton pueden ser utilizados para la resolución de problemas de aprendizaje profundo, y requieren menos recursos que los optimizadores populares actuales, como Adam, y aún así proporcionando resultados de generalización mejores que los de las contrapartes. Funcionan excepcionalmente bien para problemas altamente no convexos y con funciones que implican la manipulación de matrices con muchos parámetros. El problema resuelto es interesante porque presenta similitudes con muchas otras aplicaciones, y además se ha usado una arquitectura muy útil para otros muchos problemas dentro del campo de las redes neuronales.

En el Capítulo 2 se presentó que el método de Newton en optimización es un método iterativo que utiliza una aproximación lineal con información de segundo orden de la función para encontrar una solución aproximada del próximo paso para minimizar la función objetivo. El método se basa en una generalización de una serie de Taylor multidimensional desarrollada hasta el segundo orden. Una de las ventajas del método de Newton es que converge más rápido que otros métodos, pero tiene algunos problemas como su sensibilidad a las condiciones iniciales y problemas de singularidad si la matriz hessiana no es definida positiva o es demasiado grande.

En el Capítulo 3 se discutió que los métodos cuasi-Newton son una clase de algoritmos de optimización numérica que buscan aproximar la matriz hessiana de una función en lugar de calcularla directamente. Los métodos cuasi-Newton de memoria limitada son especialmente útiles para resolver problemas grandes, ya que requieren solo un almacenamiento de $O(mn)$ en lugar de $O(n^2)$. El método L-BFGS es un ejemplo de un método de memoria limitada que utiliza información de las iteraciones más recientes para construir una aproximación de la matriz Hessiana.

La representación gráfica de los resultados mostró que los métodos cuasi-Newton, como L-BFGS y SR1, tienden a dirigirse de manera más directa hacia uno de los puntos de mínimo de la función y lo hacen con menor esfuerzo computacional en cada iteración. Además, cada método tiene una tendencia a dirigirse hacia un punto diferente, pero esto es debido a las diferencias inherentes al método y no al punto inicial. Es importante también notar que SR1 se aplicó bajo un esquema de búsqueda de línea, aunque suele utilizarse en un esquema de región de confianza. En algunos casos, SR1 puede no garantizar una definición positiva de la matriz actualizada.

En el Capítulo 4 se discutió el aprendizaje profundo y su papel dentro de la inteligencia artificial. El aprendizaje profundo es una técnica

de aprendizaje automático que utiliza redes neuronales artificiales de varias capas para resolver problemas complejos de optimización. Es útil para desenredar los factores de variación en los datos y descartar aquellos que no son importantes. Los algoritmos de aprendizaje profundo intentan aprender un modelo a partir de los datos observados para predecir las representaciones ocultas dentro de un modelo. El aprendizaje profundo también se conoce como redes neuronales artificiales y se basa en modelos computacionales de aprendizaje biológico. A medida que las redes neuronales se han vuelto más profundas, se han enfrentado problemas de gradientes desvaneciéndose o .explotando". Los métodos de optimización más comunes utilizados en la actualidad son el descenso de gradiente con momentum y sus variantes, como RMSprop y Adam. Sin embargo, también se han desarrollado métodos cuasi-Newton para aprovechar la información de segundo orden en la optimización de aprendizaje automático. Estos métodos son muy recientes debido a la escala cada vez mayor del aprendizaje profundo.

En el Capítulo 5 se presenta una aplicación del método L-BFGS en una red neuronal utilizando una estrategia de región de confianza. Se discuten las ventajas de generalización traídas por las particularidades del método cuasi-Newton al no hacer uso directo de información de segundo orden en comparación con optimizadores como Adam y SGD con impulso, así como también las variantes de memoria limitada de BFGS discutidas en el capítulo anterior. Se realizan experimentos adicionales sobre el tamaño del lote, el número de lotes con superposición y la variante de memoria utilizada. El trabajo se enfoca en una función de costo de un problema de clasificación de dígitos numéricos escritos a mano (MNIST) y se presentaron resultados robustos sobre una red neuronal convolucional en donde se emplea tanto un algoritmo de optimización cuasi-Newton como otros derivados del máximo descenso sobre la función de coste. Se reportan tasas de convergencia más estables y robustas, así como niveles de generalización empíricamente más altos con volúmenes de pérdidas más bajos y niveles de precisión más altos a favor de los métodos cuasi-Newton.

6.2. Líneas futuras

Hay varias líneas de investigación que se pueden explorar en el campo del Aprendizaje Profundo haciendo uso de algoritmos de optimización cuasi-Newton. Por ejemplo, se pueden explorar nuevos métodos para mejorar la convergencia de los modelos de redes neuronales, como la optimización por lotes, la optimización distribuida, así como nuevas técnicas de regularización que permitan un entrenamiento más rápido. También se pueden estudiar los modelos híbridos, en los que se combinan la optimización cuasi-Newton y otros métodos de optimización para lograr mejores resultados.

Actualmente, en herramientas como Python no existen librerías de software especializadas a estos métodos de optimización avanzados para uso dentro de un esquema de aprendizaje profundo. Más bien, las implementaciones de los métodos cuasi-Newton dentro de librerías de Python están meramente orientadas a resolver problemas de optimización matemática, mas no están directamente integradas dentro de esquemas de resolución de problemas de redes neuronales. Esto implicaría tener librerías optimizadas para la multiplicación secuencial de matrices o tensores a gran escala. Debido a esto, el tiempo computacional se hace pesado, sin embargo, los resultados obtenidos son mucho más robustos a medida incrementamos el número de épocas dentro del entrenamiento (para el caso de L-BFGS).

Algunas de las futuras líneas de investigación identificadas dentro de la revisión de los avances actuales se encuentran:

1. Muestreo adaptativo del tamaño del lote (*batch size*) y métodos para escoger la variante de memoria (m), de forma tal que se maximicen métricas dirigidas como la precisión de las aproximaciones de las matrices asociadas a cada iteración, o hacia alguna otra métrica escogida. Esto mejoraría significativamente los tiempos de entrenamiento, así como también la velocidad con la que cada iteración del algoritmo se emplea.
2. Tasas de convergencia locales más rápidas.
3. Varianza reducida sobre las aproximaciones de manera tal que se cumplan las condiciones de Wolfe (cuando se aplican estrategias de búsqueda de línea), ya que esta estrategia de optimización es especialmente compleja de implementar debido a que las condiciones de Wolfe se deben cumplir de manera simultánea, muchas veces impidiendo el avance computacional del cálculo de las matrices de segundo orden.
4. Más resultados numéricos sobre bases de datos no probadas anteriormente para medir niveles de generalización y velocidad de convergencia.
5. Implementaciones de software optimizadas de los métodos cuasi-Newton dentro de las principales librerías de aprendizaje profundo como Pytorch y TensorFlow. Este es uno de los puntos más cruciales para poder seguir avanzando sobre el campo y poder llegar a aplicar los métodos sobre problemas más allá de redes convolucionales, pero también llevarlos a modelos de lenguaje natural (NLP) y de series de tiempo de larga memoria (LSTM), por ejemplo.

Lo más importante es que ya no hay que empezar desde cero para que el campo avance. Este campo se puede tornar bastante complejo a medida entren otras vertientes como la computación distribuida y métodos de optimización híbridos, pero mientras más problemas se vayan resolviendo más interesantes y gratificantes se volverán los que siguen.

El campo del Aprendizaje Profundo haciendo uso de algoritmos de optimización cuasi-Newton tiene un gran potencial para resolver problemas de optimización a gran escala y mejorar la velocidad y precisión de los modelos de redes neuronales. Las investigaciones sobre este campo sólo hacen evidentes resultados altamente robustos, escalables, con favorables métricas de generalización, todo haciendo uso de técnicas de optimización matemáticas de última generación. Además, es importante seguir desarrollando implementaciones optimizadas de los métodos cuasi-Newton dentro de las principales librerías de aprendizaje profundo. Es notable que el campo es complejo y requiere una gran cantidad de recursos computacionales para poder ser implementado de manera eficiente. A pesar de esto, los avances en el campo son prometedores y pueden llevar a soluciones cada vez más precisas y eficientes en el entrenamiento de redes neuronales.

Capítulo 7

Anexos

La implementación a nivel de código en Python puede ser encontrada en el siguiente repositorio de GitHub: MMA-Tesis.

7.1. Función de Himmelblau

A continuación, el código MATLAB utilizado para la resolución analítica del problema de minimización de la Función de Himmelblau (3.20). Define la función de Himmelblau (f), el gradiente de la función (gradiente), el hessiano de la función (hessiano) y los puntos críticos (extremos) de la función. La función se utiliza para encontrar los puntos críticos de la función y verifica si la función es convexa o no utilizando la segunda derivada. Finalmente, se utiliza la función matlabFunction para convertir la función en una función numérica, evaluar en los puntos críticos y verificar si los mismos son minimizadores o no.

```
clc
clear all;
% Funcion de Himmelblau

syms x1 x2
f = (x1^2 + x2 - 11)^2 + (x1 + x2^2 -  7)^2;
gradiente = gradient(f)
hessiano = hessian(f)
hessiano = matlabFunction(hessiano);
S=solve(gradient(f)==0)

temp = struct2cell(S); % Pasar el struct a sym
extremos = vpa(horzcat(temp{:}));
extremos

diff(f, 2) %La funcion es convexa, segunda derivada positiva

f_sym = matlabFunction(f);

vpa(f_sym(extremos(:,1),extremos(:,2))) %Puntos en donde la funcion se hace cero
```

Listing 7.1: Código MATLAB para hallar puntos extremos de la función

7.2. Algoritmos cuasi-Newton en optimización

Este código importa varias librerías para utilizar en el programa, como numpy, scipy, y matplotlib. Luego, define dos funciones que representan la Función objetivo de Himmelblau y su gradiente respectivamente. En seguida, crea una malla de puntos para trazar una superficie de contorno llena en un gráfico de contorno en 2D. Luego se van creando las funciones de cada uno de los métodos a probar para luego comparar los resultados obtenidos.

```
import numpy as np
import numpy.linalg as ln
import scipy as sp
```

```
import scipy.optimize
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
from matplotlib.colors import LogNorm
import matplotlib.pyplot as plt
import numpy as np

# Funcion objetivo
def f(x):
    return (x[0]**2 + x[1] - 11)**2 + (x[0] + x[1]**2 - 7)**2

# Gradiente de funcion objetivo
def f1(x):
    return np.array([2*x[0] + 4*x[0] * (x[0]**2 + x[1] - 11) + 2*x[1]**2 - 14,
                     2*x[1] + 4*x[1] * (x[1]**2 + x[0] - 7) + 2*x[0]**2 - 22])

fig = plt.figure()
fig, axes = plt.subplots(1, 1, sharey=True, figsize=(10, 10))
X = np.arange(-4, 4, 0.1)
Y = np.arange(-4, 4, 0.1)
X, Y = np.meshgrid(X, Y)
Z = (X*X+Y-11)**2 + (X+Y*Y-7)**2
plt.axis('on')

axes.contourf(X, Y, Z, levels=1000, cmap = cm.jet)

axes.tick_params(axis='both', which='major', labelsize=20)
axes.tick_params(axis='both', which='minor', labelsize=20)

plt.xlabel("x")
plt.ylabel("y")
plt.show()
```

Listing 7.2: Gráfico de contorno de la función de Himmelblau

```
from numpy import arange, meshgrid
import matplotlib.cm as cm
import matplotlib.pyplot as plt

def metodo_bfgs(f, fprime, x0,
                maxiter=1000, epsi=10e-3):
    """
    Minimizar una funcion f usando el algoritmo BFGS.

    Parametros
    ----------
    f : f(x)
        Funcion a minimizar.
    x0 : ndarray
        Estimacion inicial.
    fprime : fprime(x)
        El gradiente de `func`.
    """

    # valores iniciales
    k = 0
    gfk = fprime(x0)
    N = len(x0)
    # Matriz de identidad para inicializar h_k
    I = np.eye(N, dtype=int)
    Hk = I
    xk = x0
    xk_vect = [xk]
    alpha_k_list = []
    while ln.norm(gfk) > epsi and k < maxiter: #Criterios de convergencia

        # pk - Direccion de busqueda
        pk = -np.dot(Hk, gfk)

        # Estrategia utilizada: Busqueda de linea
        # B squeda de linea para las condiciones de Wolfe
```

```
        line_search = sp.optimize.line_search(f, f1, xk, pk, maxiter=500)
        #alpha_k - Tamano del paso
        alpha_k = line_search[0]

        xkp1 = xk + alpha_k * pk
        sk = xkp1 - xk
        xk = xkp1

        gfkp1 = fprime(xkp1)
        yk = gfkp1 - gfk
        gfk = gfkp1

        k += 1

        ro = 1.0 / (np.dot(yk, sk))
        A1 = I - ro * sk[:, np.newaxis] * yk[np.newaxis, :]
        A2 = I - ro * yk[:, np.newaxis] * sk[np.newaxis, :]
        Hk = np.dot(A1, np.dot(Hk, A2)) + (ro * sk[:, np.newaxis] *
                                           sk[np.newaxis, :]) #H_{k+1}
        alpha_k_list.append(alpha_k) # Guardar listado de alphas usados
        xk_vect.append(xk)# Guardar listado de puntos obtenidos para el grafico

    return (xk_vect, k, alpha_k_list[-5:], ln.norm(gfk)) #retornar ultimos 5 alphas

punto_inicial = np.array([1.80, -3.36])
result_bfgs, k_bfgs, alpha_k_bfgs, norma_bfgs = metodo_bfgs(f, f1, punto_inicial, maxiter=5000)

print('Resultado del metodo BFGS:')
print('Resultado final (minimizacion): %s' % (result_bfgs[-1]))
print('Conteo de iteraciones: %s' % (k_bfgs))
print('Ultimos 5 valores de alpha_k optimos que satisfacen condiciones Wolfe: %s' % (alpha_k_bfgs))
print('Norma obtenida en BFGS: %s' % (norma_bfgs))



def return_appropriate_result_column(list_of_lists):
    '''
    Funcion que retorna una lista de listas segun puntos resultado del metodo.
    '''
    xvect = [row[0] for row in list_of_lists]
    yvect = [row[1] for row in list_of_lists]
    return [xvect, yvect]

# definir el rango del input
r_min, r_max = -4.0, 4.0
# rango de entrada de muestra uniformemente en incrementos de 0.1
xaxis = arange(r_min, r_max, 0.1)
yaxis = arange(r_min, r_max, 0.1)
# crear una malla a partir del eje
x, y = meshgrid(xaxis, yaxis)
# computar objetivos
results = f([x, y])

# definir set de pasos tomados para llegar al optimo
optima_x = return_appropriate_result_column(result_bfgs)

method = 'BFGS'
fig, axes = plt.subplots(1,2, sharey=True, figsize=(10,7))
for ax, zord in zip(axes, [1,-1]):
    # crear un grafico de contorno lleno con 100 niveles y un esquema de color jet
    ax.contourf(x, y, results, levels=100, cmap='jet')
    ax.autoscale(False) # Para evitar que cambios de limite
    ax.plot(optima_x[0], optima_x[1], zorder=zord, color='red')
    ax.plot(3, 2, '*', color='green') # Punto de optimos
    ax.plot(-2.8051, 3.1313, '*', color='green') # Punto de optimos
    ax.plot(-3.7793, -3.2831, '*', color='green') # Punto de optimos
    ax.plot(3.5844, -1.8481, '*', color='green') # Punto de optimos
    ax.tick_params(axis='both', which='major', labelsize=15)
    ax.tick_params(axis='both', which='minor', labelsize=15)
    ax.set_xlim(r_min, r_max)
```

```
    ax.set_ylim(r_min, r_max)
```

Listing 7.3: BFGS

```
def metodo_sr1(f, fprime, x0,
               maxiter=None, epsi=10e-3):
    '''
    Implementacion del metodo cuasi-Newton SR1 (Symmetric Rank 1)
    '''

    # valores iniciales
    k = 0
    gfk = fprime(x0)
    N = len(x0)
    # matriz de identidad para inicializar p_k
    I = np.eye(N, dtype=int)
    Hk = I # Inicializar Hk con la matriz identidad de tamano N
    xk = x0
    alpha_k_list = []
    xk_vect = [xk]
    while ln.norm(gfk) > epsi and k < maxiter: #Criterios de convergencia

        # pk - Direccion de busqueda
        pk = -np.dot(Hk, gfk)

        #LINE SEARCH
        # Constantes de busqueda de linea para las condiciones de Wolfe.
        line_search = sp.optimize.line_search(f, f1, xk, pk, maxiter=500)
        # alpha_k - Tamano del paso
        alpha_k = line_search[0]

        xkp1 = xk + alpha_k * pk
        sk = xkp1 - xk
        xk = xkp1

        gfkp1 = fprime(xkp1)
        yk = gfkp1 - gfk
        gfk = gfkp1

        k += 1

        A1 = np.dot(Hk, yk) - sk
        Hk = Hk - (np.dot(A1[:, np.newaxis], A1[np.newaxis, :])/np.dot(A1[np.newaxis, :], yk))
        alpha_k_list.append(alpha_k) # Guardar listado de alphas usados
        xk_vect.append(xk) # Guardar listado de puntos obtenidos para el grafico

    return (xk_vect, k, alpha_k_list[-5:], ln.norm(gfk))

punto_inicial = np.array([1.80, -3.36])
result_sr1, k_sr1, alpha_k_sr1, norma_sr1 = metodo_sr1(f, f1, punto_inicial, maxiter=5000)

print('Resultado del metodo SR1:')
print('Resultado final (minimizacion): %s' % (result_sr1[-1]))
print('Conteo de iteraciones: %s' % (k_sr1))
print('Ultimos 5 valores de alpha_k optimos que satisfacen condiciones Wolfe: %s' % (alpha_k_sr1))
print('Norma obtenida en SR1: %s' % (norma_sr1))


# definir set de pasos tomados para llegar al optimo
optima_x = return_appropriate_result_column(result_sr1) #Resultados de SR1 aplicados a la funcion

method = 'SR1'
fig, axes = plt.subplots(1,2, sharey=True, figsize=(10,7))
for ax, zord in zip(axes, [1,-1]):
    # crear un grafico de contorno lleno con 100 niveles y un esquema de color jet
    ax.contourf(x, y, results, levels=100, cmap='jet')
    ax.autoscale(False) # Para evitar que cambios de limite
    ax.plot(optima_x[0], optima_x[1], zorder=zord, color='red')
    ax.plot(3, 2, '*', color='green') # Punto de optimos
    ax.plot(-2.8051, 3.1313, '*', color='green') # Punto de optimos
    ax.plot(-3.7793, -3.2831, '*', color='green') # Punto de optimos
```

```
    ax.plot(3.5844, -1.8481, '*', color='green') # Punto de optimos
    ax.tick_params(axis='both', which='major', labelsize=15)
    ax.tick_params(axis='both', which='minor', labelsize=15)
    ax.set_xlim(r_min, r_max)
    ax.set_ylim(r_min, r_max)
```

Listing 7.4: SR1

```
def hessian_function(x):
    '''
    Predefinir el hessiano de la funcion.
    '''
    return np.array([[12*x[0]**2+4*x[1]-42, 4*x[0]+4*x[1]],
                     [4*x[0]+4*x[1],        12*x[1]**2+4*x[0]-26]])


def metodo_newton(f, fprime, hessian,
                  x0, maxiter=5000, epsi=10e-2):
    '''
    Implementacion del metodo de Newton-Raphson.
    '''

    k = 0 # Iniciar conteo de iteraciones
    gfk = fprime(x0) #jacobiano
    Hk = hessian(x0) #hessiano
    xk = x0
    xk_vect = [xk]
    S = np.linalg.inv(Hk) # Direccion de busqueda

    while ln.norm(gfk) > epsi and k < maxiter:

        xk = xk - np.dot(S, gfk)
        k+=1
        xk_vect.append(xk)
        gfk = fprime(xk)
        Hk = hessian(xk)
        S = np.linalg.inv(Hk)

    return xk_vect, k, ln.norm(gfk)

punto_inicial = np.array([1.80, -3.36])
#punto_inicial = np.array([2,2]) #set de puntos iniciales
result_newton, k_newton, norma_newton = metodo_newton(f, f1, hessian_function, punto_inicial)



print('Resultado del metodo de Newton:')
print('Resultado final (minimizacion): %s' % (result_newton[-1]))
print('Conteo de iteraciones: %s' % (k_newton))
print('Norma obtenida en Newton: %s' % (norma_newton))


optima_x = return_appropriate_result_column(result_newton) #Resultados de Newton aplicados a la
    funcion

method = 'Newton'
fig, axes = plt.subplots(1,2, sharey=True, figsize=(10,7))
for ax, zord in zip(axes, [1,-1]):
    # crear un grafico de contorno lleno con 100 niveles y un esquema de color jet
    ax.contourf(x, y, results, levels=100, cmap='jet')
    ax.autoscale(False) # Para evitar que cambios de limite
    ax.plot(optima_x[0], optima_x[1], zorder=zord, color='red')
    ax.plot(3, 2, '*', color='green') # Punto de optimos
    ax.plot(-2.8051, 3.1313, '*', color='green') # Punto de optimos
    ax.plot(-3.7793, -3.2831, '*', color='green') # Punto de optimos
    ax.plot(3.5844, -1.8481, '*', color='green') # Punto de optimos
    ax.tick_params(axis='both', which='major', labelsize=15)
    ax.tick_params(axis='both', which='minor', labelsize=15)
    ax.set_xlim(r_min, r_max)
```

```
    ax.set_ylim(r_min, r_max)
```

Listing 7.5: Método de Newton

```
def metodo_gradient_descent(f, fprime, x0,
                            maxiter=5000, epsi=10e-3):
    '''
    Implementacion del metodo del maximo descenso.
    '''
    k = 0 # Iniciar conteo de iteraciones
    gfk = fprime(x0)
    xk = x0
    xk_vect = [xk]
    alpha = 0.01 #alpha busqueda de linea fijo
    while ln.norm(gfk) > epsi and k < maxiter:
        xk = xk - alpha * gfk
        k+=1
        xk_vect.append(xk)
        gfk = fprime(xk)
    return xk_vect, k, ln.norm(gfk)

punto_inicial = np.array([1.80, -3.36])
#punto_inicial = np.array([-6,0])
result_GD, k_GD, norma_gd = metodo_gradient_descent(f, f1, punto_inicial)

print('Resultado del metodo de GD:')
print('Resultado final (minimizacion): %s' % (result_GD[-1]))
print('Conteo de iteraciones: %s' % (k_GD))
print('Norma obtenida en Gradient Descent: %s' % (norma_gd))


optima_x = return_appropriate_result_column(result_GD) #Resultados de GD aplicados a la funcion

method = 'Descenso de Gradiente'
fig, axes = plt.subplots(1,2, sharey=True, figsize=(10,7))
for ax, zord in zip(axes, [1,-1]):
    # crear un grafico de contorno lleno con 100 niveles y un esquema de color jet
    ax.contourf(x, y, results, levels=100, cmap='jet')
    ax.autoscale(False) # Para evitar que cambios de limite
    ax.plot(optima_x[0], optima_x[1], zorder=zord, color='red')
    ax.plot(3, 2, '*', color='green') # Punto de optimos
    ax.plot(-2.8051, 3.1313, '*', color='green') # Punto de optimos
    ax.plot(-3.7793, -3.2831, '*', color='green') # Punto de optimos
    ax.plot(3.5844, -1.8481, '*', color='green') # Punto de optimos
    ax.tick_params(axis='both', which='major', labelsize=15)
    ax.tick_params(axis='both', which='minor', labelsize=15)
    ax.set_xlim(r_min, r_max)
    ax.set_ylim(r_min, r_max)
```

Listing 7.6: Método de descenso de gradiente

7.3. Variaciones de SGD

Haciendo uso de la misma función de Himmelblau, se aplican otros algoritmos de optimización como Adam y RMSProp para análisis comparativo.

```
import matplotlib.pyplot as plt
import autograd.numpy as np

from mpl_toolkits.mplot3d import Axes3D
from matplotlib.colors import LogNorm
from matplotlib import animation
from IPython.display import HTML

from autograd import elementwise_grad, value_and_grad
from scipy.optimize import minimize
```

```
from collections import defaultdict
from itertools import zip_longest
from functools import partial



# Funcion de Himmelblau
f  = lambda x, y: (x**2 + y - 11)**2 + (x + y**2 - 7)**2

xmin, xmax, xstep = -4, 4, .1
ymin, ymax, ystep = -4, 4, .1
x, y = np.meshgrid(np.arange(xmin, xmax + xstep, xstep), np.arange(ymin, ymax + ystep, ystep))

z = f(x, y)

minima1 = np.array([3, 2])
minima2 = np.array([-2.8051, 3.1313])
minima3 = np.array([-3.7793, -3.2831])
minima4 = np.array([3.5844, -1.8481])

# Verificar que los puntos son minimizadores en la funcion:
print(f(*minima1))
print(f(*minima2))
print(f(*minima3))
print(f(*minima4)) #valores muy cercanos a cero

minima_1 = minima1.reshape(-1, 1)
minima_2 = minima2.reshape(-1, 1)
minima_3 = minima3.reshape(-1, 1)
minima_4 = minima4.reshape(-1, 1)

print(f(*minima_1))
print(f(*minima_2))
print(f(*minima_3))
print(f(*minima_4))

fig = plt.figure(figsize=(8, 5))
ax = plt.axes(projection='3d', elev=50, azim=-50)

ax.plot_surface(x, y, z, norm=LogNorm(), rstride=1, cstride=1,
                edgecolor='none', alpha=.8, cmap=plt.cm.jet)
ax.plot(*minima_1, f(*minima_1), 'r*', markersize=10)
ax.plot(*minima_2, f(*minima_2), 'r*', markersize=10)
ax.plot(*minima_3, f(*minima_3), 'r*', markersize=10)
ax.plot(*minima_4, f(*minima_4), 'r*', markersize=10)

ax.set_xlabel('
')
ax.set_ylabel('
')
ax.set_zlabel('
')

ax.set_xlim((xmin, xmax))
ax.set_ylim((ymin, ymax))

plt.show() # Grafico 3D

dz_dx = elementwise_grad(f, argnum=0)(x, y)
dz_dy = elementwise_grad(f, argnum=1)(x, y)

fig, ax = plt.subplots(figsize=(10, 6))

ax.contour(x, y, z, levels=np.logspace(0, 5, 35), norm=LogNorm(), cmap=plt.cm.jet)
ax.quiver(x, y, x - dz_dx, y - dz_dy, alpha=.5)
ax.plot(*minima_1, 'r*', markersize=18)
ax.plot(*minima_2, 'r*', markersize=18)
ax.plot(*minima_3, 'r*', markersize=18)
ax.plot(*minima_, 'r*', markersize=18)
```

```
ax.set_xlabel('
')
ax.set_ylabel('
')

ax.set_xlim((xmin, xmax))
ax.set_ylim((ymin, ymax))

plt.show() # Grafico 2D

# Colocando el punto inicial para todos los algoritmos a probar:
x0 = np.array([-0.6, -0.84])
func = value_and_grad(lambda args: f(*args))
res = minimize(func, x0=x0, method='BFGS',
               jac=True, tol=1e-20, callback=print)

def make_minimize_cb(path=[]):

    def minimize_cb(xk):
        # creamos un deep copy de xk
        path.append(np.copy(xk))

    return minimize_cb

path_ = [x0]
res = minimize(func, x0=x0, method='BFGS',
               jac=True, tol=1e-20, callback=make_minimize_cb(path_))
path = np.array(path_).T
path.shape

fig, ax = plt.subplots(figsize=(10, 6))

ax.contour(x, y, z, levels=np.logspace(0, 5, 35), norm=LogNorm(), cmap=plt.cm.jet)
ax.quiver(path[0,:-1], path[1,:-1], path[0,1:]-path[0,:-1], path[1,1:]-path[1,:-1], scale_units='xy'
    , angles='xy', scale=1, color='k')
ax.plot(*minima_1, 'r*', markersize=18)
ax.plot(*minima_2, 'r*', markersize=18)
ax.plot(*minima_3, 'r*', markersize=18)
ax.plot(*minima_4, 'r*', markersize=18)

ax.set_xlabel('
')
ax.set_ylabel('
')

ax.set_xlim((xmin, xmax))
ax.set_ylim((ymin, ymax))
ax.set_title("BFGS") # Grafico de trayectoria de prueba

import numpy as np
from scipy.optimize import OptimizeResult
# SGD

def sgd(
    fun,
    x0,
    jac,
    args=(),
    learning_rate=0.001,
    mass=0.9,
    startiter=0,
    maxiter=5000,
    callback=None,
    **kwargs
):
    """‘‘scipy.optimize.minimize‘‘ compatible implementation of stochastic
    gradient descent with momentum.
    Adapted from ‘‘autograd/misc/optimizers.py‘‘.
    """
    x = x0
```

```
    velocity = np.zeros_like(x)

    for i in range(startiter, startiter + maxiter):
        g = jac(x)

        if callback and callback(x):
            break

        velocity = mass * velocity - (1.0 - mass) * g
        x = x + learning_rate * velocity

    i += 1
    return OptimizeResult(x=x, fun=fun(x), jac=g, nit=i, nfev=i, success=True)


# RMSProp
def rmsprop(
    fun,
    x0,
    jac,
    args=(),
    learning_rate=0.1,
    gamma=0.9,
    eps=1e-8,
    startiter=0,
    maxiter=5000,
    callback=None,
    **kwargs
):
    """``scipy.optimize.minimize`` compatible implementation of root mean
    squared prop: See Adagrad paper for details.
    Adapted from ``autograd/misc/optimizers.py``.
    """
    x = x0
    avg_sq_grad = np.ones_like(x)

    for i in range(startiter, startiter + maxiter):
        g = jac(x)

        if callback and callback(x):
            break

        avg_sq_grad = avg_sq_grad * gamma + g**2 * (1 - gamma)
        x = x - learning_rate * g / (np.sqrt(avg_sq_grad) + eps)

    i += 1
    return OptimizeResult(x=x, fun=fun(x), jac=g, nit=i, nfev=i, success=True)

# Adam
def adam(
    fun,
    x0,
    jac,
    args=(),
    learning_rate=0.001,
    beta1=0.9,
    beta2=0.999,
    eps=1e-8,
    startiter=0,
    maxiter=5000,
    callback=None,
    **kwargs
):
    """``scipy.optimize.minimize`` compatible implementation of ADAM -
    [http://arxiv.org/pdf/1412.6980.pdf].
    Adapted from ``autograd/misc/optimizers.py``.
    """
    x = x0
    m = np.zeros_like(x)
    v = np.zeros_like(x)
```

```
    for i in range(startiter, startiter + maxiter):
        g = jac(x)

        if callback and callback(x):
            break

        m = (1 - beta1) * g + beta1 * m  # estimacion del primer momento.
        v = (1 - beta2) * (g**2) + beta2 * v  # estimacion del segundo momento.
        mhat = m / (1 - beta1**(i + 1))  # correccion de sesgo.
        vhat = v / (1 - beta2**(i + 1))
        x = x - learning_rate * mhat / (np.sqrt(vhat) + eps)

    i += 1
    return OptimizeResult(x=x, fun=fun(x), jac=g, nit=i, nfev=i, success=True)

# Implementacion

methods = [
    sgd,
    rmsprop,
    adam
#     "BFGS",
#     "Newton-CG",
#     "L-BFGS-B",
#     "TNC",
#     "SLSQP"
]

path_ = [x0]
res = minimize(func, x0=x0, method=methods[0],
               jac=True, tol=1e-20, callback=make_minimize_cb(path_))
path0 = np.array(path_).T

path_ = [x0]
res = minimize(func, x0=x0, method=methods[1],
               jac=True, tol=1e-20, callback=make_minimize_cb(path_))
path1 = np.array(path_).T
path_ = [x0]
res = minimize(func, x0=x0, method=methods[2],
               jac=True, tol=1e-20, callback=make_minimize_cb(path_))
path2 = np.array(path_).T

fig, ax = plt.subplots(figsize=(10, 6))

ax.contour(x, y, z, levels=np.logspace(0, 5, 35), norm=LogNorm(), cmap=plt.cm.jet)
ax.quiver(path0[0,:-1], path0[1,:-1], path0[0,1:]-path0[0,:-1], path0[1,1:]-path0[1,:-1],
          scale_units='xy', angles='xy', scale=0.5, color='red', label="SGD")
ax.quiver(path1[0,:-1], path1[1,:-1], path1[0,1:]-path1[0,:-1], path1[1,1:]-path1[1,:-1],
          scale_units='xy', angles='xy', scale=0.5, color='yellow', label="RMSProp")
ax.quiver(path2[0,:-1], path2[1,:-1], path2[0,1:]-path2[0,:-1], path2[1,1:]-path2[1,:-1],
          scale_units='xy', angles='xy', scale=0.5, color='green', label="Adam")
ax.plot(*minima_1, 'r*', markersize=10)
ax.plot(*minima_2, 'r*', markersize=10)
ax.plot(*minima_3, 'r*', markersize=10)
ax.plot(*minima_4, 'r*', markersize=10)

ax.set_xlabel('
')
ax.set_ylabel('
')

ax.set_xlim((xmin, xmax))
ax.set_ylim((ymin, ymax))

ax.legend(loc='best') # Grafico de comparacion final
```

Listing 7.7: Análisis de variaciones del algoritmo SGD

7.4. Comparación de la optimización cuasi-Newton en aprendizaje profundo

En esta Sección se ilustra el código utilizado para la construcción de la red neuronal convolucional bajo el esquema tradicional de optimización, usando los algoritmos Adam y SGD.

```
#MNIST Dataset

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, Conv2D, MaxPool2D
from keras.utils.np_utils import to_categorical
from keras.utils import np_utils
from keras.datasets import mnist
import matplotlib.pyplot as plt
import numpy as np

# cargando el conjunto de datos
(X_train, y_train), (X_test, y_test) = mnist.load_data()
# Aplanando las imagenes de los 28x28 pixeles a 1D 787 pixeles
X_train = X_train.reshape(60000, 784)
X_test = X_test.reshape(10000, 784)
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')

# normalizando los datos para ayudar con el entrenamiento
X_train /= 255
X_test /= 255

# codificacion one-hot utilizando las utilidades relacionadas con numpy de keras
n_classes = 10
print("Forma antes de la codificacion one-hot: ", y_train.shape)
Y_train = np_utils.to_categorical(y_train, n_classes)
Y_test = np_utils.to_categorical(y_test, n_classes)
print("Forma despues de la codificacion one-hot: ", Y_train.shape)

# construyendo una pila lineal de capas con el modelo secuencial
model = Sequential()
# capa oculta
model.add(Dense(100, input_shape=(784,), activation='relu'))
# capa de salida
model.add(Dense(10, activation='softmax'))

# mirando el resumen del modelo
model.summary()

# compilando el modelo secuencial
model.compile(loss='categorical_crossentropy', metrics=['accuracy'], optimizer='Adam')
#SGD / RMSprop / Adadelta / Adagrad / Adam / Adamax / Ftrl / Nadam
# entrenando el modelo por 100 epocas
hist = model.fit(X_train, Y_train, batch_size=1000, epochs=100, validation_data=(X_test, Y_test))


# Graficos de validacion Adam

plt.rcParams.update({'font.size': 18})
plt.rc('font', family='serif')
plt.style.use('seaborn-muted')

#imprimir loss y accuracy
loss, acc = model.evaluate(X_train, Y_train, batch_size=1000, verbose=0)
print(f'Perdida en train set: {loss:.4f}')
loss, acc = model.evaluate(X_test, Y_test, batch_size=1000, verbose=0)
print(f'Perdida en test set:  {loss:.4f}')

maxiter=100 # Maximo numero de iteraciones establecido anteriormente

plt.subplots(figsize = (10, 10))
plt.plot(hist.history['loss'],
         color='green', marker='o', linestyle='dashed', linewidth=2, markersize=12)
plt.plot(hist.history['val_loss'],
```

```
        color='blue', marker='o', linestyle='dashed', linewidth=2, markersize=12)

plt.yscale("linear")
#plt.title('Perdida Adam', fontsize=35)
plt.ylabel('Perdida', fontsize=30)
plt.xlabel('Numero de epocas', fontsize=30)
plt.legend(['Entrenamiento', 'Validacion'], loc='upper right', fontsize=20)
plt.tick_params(axis='both', which='major', labelsize=20)
plt.tick_params(axis='both', which='minor', labelsize=20)


#imprimir loss y accuracy
loss, acc = model.evaluate(X_train, Y_train, batch_size=1000, verbose=0)
print(f'Precision en train set: {acc:.4f}')
loss, acc = model.evaluate(X_test, Y_test, batch_size=1000, verbose=0)
print(f'Precision en test set:  {acc:.4f}')

plt.subplots(figsize = (10, 10))
plt.plot(hist.history['accuracy'],
        color='green', marker='o', linestyle='dashed', linewidth=2, markersize=12)
plt.plot(hist.history['val_accuracy'],
        color='blue', marker='o', linestyle='dashed', linewidth=2, markersize=12)
plt.yscale("linear")
#plt.title('Precision Adam', fontsize=35)
plt.ylabel('Precision', fontsize=30)
plt.xlabel('Numero de epocas', fontsize=30)
plt.legend(['Entrenamiento', 'Validacion'], loc='upper left', fontsize=20)
plt.tick_params(axis='both', which='major', labelsize=20)
plt.tick_params(axis='both', which='minor', labelsize=20)



# construyendo una pila lineal de capas con el modelo secuencial
model2 = Sequential()
# capa oculta
model2.add(Dense(100, input_shape=(784,), activation='relu'))
# capa de salida
model2.add(Dense(10, activation='softmax'))

# compilando el modelo secuencial
model2.compile(loss='categorical_crossentropy', metrics=['accuracy'], optimizer='SGD')
#SGD / RMSprop / Adadelta / Adagrad / Adam / Adamax / Ftrl / Nadam
# entrenando el modelo por 100 epocas
hist2 = model2.fit(X_train, Y_train, batch_size=1000, epochs=100, validation_data=(X_test, Y_test) )

#imprimir loss y accuracy
loss2, acc2 = model2.evaluate(X_train, Y_train, batch_size=1000, verbose=0)
print(f'Perdida en train set: {loss2:.4f}')
loss2, acc2 = model2.evaluate(X_test, Y_test, batch_size=1000, verbose=0)
print(f'Perdida en test set:  {loss2:.4f}')

plt.subplots(figsize = (10, 10))
plt.plot(hist2.history['loss'],
        color='green', marker='o', linestyle='dashed', linewidth=2, markersize=12)
plt.plot(hist2.history['val_loss'],
        color='blue', marker='o', linestyle='dashed', linewidth=2, markersize=12)

plt.yscale("linear")
#plt.title('Perdida SGD', fontsize=35)
plt.ylabel('Perdida', fontsize=30)
plt.xlabel('Numero de epocas', fontsize=30)
plt.legend(['Entrenamiento', 'Validacion'], loc='upper right', fontsize=20)
plt.tick_params(axis='both', which='major', labelsize=20)
plt.tick_params(axis='both', which='minor', labelsize=20)

#imprimir loss y accuracy
loss3, acc3 = model2.evaluate(X_train, Y_train, batch_size=1000, verbose=0)
print(f'Precision en train set: {acc3:.4f}')
loss3, acc3 = model2.evaluate(X_test, Y_test, batch_size=1000, verbose=0)
print(f'Precision en test set:  {acc3:.4f}')
```

```
plt.subplots(figsize = (10, 10))
plt.plot(hist2.history['accuracy'],
         color='green', marker='o', linestyle='dashed', linewidth=2, markersize=12)
plt.plot(hist2.history['val_accuracy'],
         color='blue', marker='o', linestyle='dashed', linewidth=2, markersize=12)
plt.yscale("linear")
#plt.title('Precision SGD', fontsize=35)
plt.ylabel('Precision', fontsize=30)
plt.xlabel('Numero de epocas', fontsize=30)
plt.legend(['Entrenamiento', 'Validacion'], loc='upper left', fontsize=20)
plt.tick_params(axis='both', which='major', labelsize=20)
plt.tick_params(axis='both', which='minor', labelsize=20)
```

Listing 7.8: Esquema tradicional de optimización en una red neuronal

En esta parte se construye la misma arquitectura, usando el mismo set de datos MNIST, pero se aplica el algoritmo cuasi-Newton L-BFGS. Esta parte debe ser reproducida en un cuaderno de Google Colab, ya que se utilizan comandos de terminal que pueden no funcionar en alguna otra herramienta como un Jupyter Notebook.

```
import os
import pickle
import matplotlib.pyplot as plt
import numpy as np
plt.rcParams.update({'font.size': 18})
plt.rc('font', family='serif')
plt.style.use('seaborn-muted')

!git clone https://github.com/josegarciav/MMA-Thesis #Clonar repositorio de la tesis

#Cambiar el directorio al folder del git clone
!pwd
os.chdir('/content/MMA-Thesis')
!pwd

!pip install tensorflow #==1.15.00  #Cambiar la version de tensorflow actual

!mkdir results #Crear carpeta dentro del directorio actual

!python LBFGS_TR.py -m=25 -minibatch=1000 -num-batch=50 -method='trust-region' -maxiter=100
# Correr el modelo convolucional en MNIST con L-BFGS, con una estrategia de region de confianza

DATA_PATH = "/content/MMA-Thesis/results/results_experiment_FEB_23_trust-
    region_m_25_n_50_minibatch_1000.pkl"
infile = open(DATA_PATH, 'rb')
best_model1 = pickle.load(infile)
# Modelo L-BFGS bajo la estrategia de region de confianza con batch size=1000

maxiter=100 # Maximo numero de iteraciones establecido anteriormente

plt.subplots(figsize = (10, 10))
plt.plot(np.linspace(1, maxiter, len(best_model1[0])), best_model1[0],
         color='green', marker='o', linestyle='dashed', linewidth=2, markersize=12)
plt.plot(np.linspace(1, maxiter, len(best_model1[1])), best_model1[1],
         color='blue', marker='o', linestyle='dashed', linewidth=2, markersize=12)

plt.yscale("linear")
plt.ylabel('Perdida', fontsize=30)
plt.xlabel('Numero de Epocas', fontsize=30)
plt.legend(['Entrenamiento', 'Validacion'], loc='upper right', fontsize=20) # , 'Prueba'
plt.tick_params(axis='both', which='major', labelsize=20)
plt.tick_params(axis='both', which='minor', labelsize=20)


plt.subplots(figsize = (10,10))
plt.plot(np.linspace(1, maxiter, len(best_model1[3])), best_model1[3],
         color='green', marker='o', linestyle='dashed', linewidth=2, markersize=12)
plt.plot(np.linspace(1, maxiter, len(best_model1[4])), best_model1[4],
         color='blue', marker='o', linestyle='dashed', linewidth=2, markersize=12)
```

```
plt.yscale("linear")
plt.ylabel('Precision', fontsize=30)
plt.xlabel('Numero de Epocas', fontsize=30)
plt.legend(['Entrenamiento', 'Validacion'], loc='upper left', fontsize=20)
plt.tick_params(axis='both', which='major', labelsize=20)
plt.tick_params(axis='both', which='minor', labelsize=20)
```

Listing 7.9: Esquema cuasi-Newton en una red neuronal

Bibliografía

[1] J. D. Cloyd. *Data Mining with Newton's Method*, East Tennessee State University (2002).

[2] J. Rafati, R. F. Marcia. *Quasi-Newton Optimization Methods for Deep Learning Applications*, Springer (2020).

[3] J. Nocedal, S. Wright. *Numerical Optimization*, Springer (2006).

[4] A. Lam. *BFGS in a Nutshell: An Introduction to Quasi-Newton Methods*, Towards Data Science (2020).

[5] E. Chong, S. H. Zak. *Introducción a la Optimización*, John Wiley & Sons, Inc. (2013).

[6] J. Kiefer. *Sequential minimax search for a maximum*, Proceedings of the American Mathematical Society (1953).

[7] A. Vázquez, J. A. *Implementación de los Métodos cuasi-Newton*, Benemérita Universidad Autónoma de Puebla (2019).

[8] V. Zahorui. *Trust region*, Unlocking the power of data (2020).

[9] W. C. Davidon. *Variable Metric Method for Minimization*, Argonne National Laboratory (1959).

[10] A. S. Berahas, M. Jahani, P. Richtárik, M. Takác. *Quasi-Newton Methods for Machine Learning: Forget the Past, Just Sample*, University of Michigan (2021).

[11] R. Fletcher, M. J. D. Powell. *A rapidly convergent descent method for minimization*, The Computer Journal (1963).

[12] J. Sherman, W. J. Morrison. *Adjustment of an Inverse Matrix Corresponding to a Change in One Element of a Given Matrix*, Courant Institute, Annals of Mathematical Statistics (1950).

[13] I. P. Lence. *Técnicas Paralelas Aplicadas a Optimización No Lineal en Sistemas de Memoria Distribuida*, Universidad Santiago de Compostela (2007).

[14] C. G. Broyden. *The convergence of a class of double-rank minimization algorithms*, IMA Journal of Applied Mathematics (1970).

[15] R. Fletcher. *A New Approach to Variable Metric Algorithms*, The Computer Journal (1970).

[16] D. Goldfarb. *A family of variable-metric methods derived by variational means*, Mathematics of Computation (1970).

[17] D. F. Shanno, P. C. Kettler. *Optimal Conditioning of Quasi-Newton Methods*, Mathematics of Computation (1970).

[18] D. C. Liu, J. Nocedal . *On the limited memory BFGS method for large scale optimization*, Mathematical Programming (1989).

[19] D. Himmelblau. *Programación no lineal aplicada*, McGraw-Hill (1972).

[20] J. Dennis, D. Gay, R. Welsch. *An Adaptive Nonlinear Least-squares Algorithm*, ACM Transactions on Mathematical Software (1981).

[21] I. Abelló Ugalde. *Región de Confianza en Métodos Estructurados Secantes para Mínimos Cuadrados*, Facultad de Ciencias - UNAM, México (2009).

[22] Y. Bengio. *Learning Deep Architectures for AI.*, Université de Montréal (2009).

[23] I. Lenz, R. Knepper, A. Saxena. *Deepmpc: learning deep latent features for model predictive control.*, Robotics: Science and Systems XI (2015).

[24] G. E. Hinton, R. R. Salakhutdinov. *Reducing the Dimensionality of Data with Neural Networks*, Science (2006).

[25] H. Lee, R. Grosse, R. Ranganath, A. Y. Ng. *Convolutional Deep Belief Networks for Scalable Unsupervised Learning of Hierarchical Representations*, Stanford University (2009).

[26] R. Collobert, J. Weston. *A Unified Architecture for Natural Language Processing: Deep Neural Networks with Multitask Learning*, ICML (2008).

[27] A.-R. Mohamed, G. E. Dahl, G. Hinton. *Acoustic Modeling Using Deep Belief Networks*, IEEE Transactions on Audio, Speech, and Language Processing (2012).

[28] I. J. Goodfellow, Y. Bengio, A. Courville. *Deep Learning*, MIT Press, Cambridge (2016).

[29] H. Dong, Z. Ding, S. Zhang. *Deep Reinforcement Learning: Fundamentals, Research and Applications*, Springer (2020).

[30] M. Nielsen. *Neural Networks and Deep Learning*, Determination Press (2015).

[31] A. Orvieto, J. Kohler, D. Pavllo, T. Hofmann, A. Lucchi. *Vanishing Curvature and the Power of Adaptive Methods in Randomly Initialized Deep Networks*, ETH Zurich (2021).

[32] H. Robbins, S. Monro. *A Stochastic Approximation Method*, The Annals of Mathematical Statistics (1951).

[33] G. Hinton, N. Srivastava, K. Swersky. *Lecture 6.5 - rmsprop: Divide the gradient by a running average of its recent magnitude*, University of Toronto (2012).

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

[35] D. Rumelhart et. al. *Learning Representations by Back-Propagating Errors*, Nature (1986).

[36] M. Minsky, S. A. Papert. *Perceptrons: An Introduction to Computational Geometry*, MIT Press (1969).

[37] K. Fukushima. *Neocognitron: A Self-organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position*, Biological Cybernetics (1980).

[38] I. J. Goodfellow et al. *Generative Adversarial Nets*, Université de Montréal (2014).

[39] A. Vaswani et al. *Attention Is All You Need*, Google Research (2017).

[40] Y. LeCun et al. *Gradient-based learning applied to document recognition*, IEEE (1998).

[41] A. Krizhevsky, I. Sutskever, G. Hinton. *ImageNet Classification with Deep Convolutional Neural Networks*, University of Toronto (2012).

[42] K. Simonyan, A. Zisserman. *Very Deep Convolutional Networks for Large-Scale Image Recognition*, University of Oxford (2015).

[43] K. He, X. Zhang, S. Ren, J. Sun. *Deep Residual Learning for Image Recognition*, Microsoft Research (2015).

[44] C. Szegedy et al. *Going Deeper with Convolutions*, Google Inc. (2014).

[45] A. G. Howard et al. *MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications*, Google Inc. (2017).

[46] M. Tan, Q. Le. *EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks*, Google Inc. (2019).

[47] D. Goldfarb, Y. Ren, A. Bahamou. *Practical Quasi-Newton Methods for Training Deep Neural Networks*, Columbia University (2020).

[48] Y. LeCun et al. *The MNIST database of handwritten digits*, Courant Institute, NYU (1998).

[49] I. Csiszar. *I-Divergence Geometry of Probability Distributions and Minimization Problems*, Annals of Probability, Institute of Mathematical Statistics (1975).