UNIVERSITAT DE BARCELONA

Facultat de Matemàtiques i Informàtica

GRADO DE MATEMÁTICAS

Trabajo final de grado

# Introducción y optimización estocástica de redes neuronales profundas MLP

Autor: Rubén Deulofeu Gomez

Director: Dr. Oriol Pujol Vila
Realizado en: Departamento de Matemática Aplicada y Análisis

Barcelona, 21 de junio de 2020

## Abstract

Since technology has stressed so much our society, human beings have always dreamed about to achieve the most. Among all those dreams, some more reachables than others, there is the ability to create machines that think by themselves. This chimera, despite of being quite different from how our ancestors ever imagined, is nowadays at the summit of its history. The massive increase of data, through digitalization, and the constant technological improvements, in this case, in the form of progress in high performance hardware production, have been the main drivers of this change.
This grade project covers a specific area that is, sometimes, a part of the aforementioned creation process, known as *Artificial Intelligence*. This specific area is called *Deep Learning*. Deep learning techniques can be regarded as the algorithms that exploit data using models based on non-linear function compositions. In this respect, optimization plays a crucial role as it is the driver that transform the information in data into the model parameters.
The project aims at understanding the mathematical basis of optimization, namely *stochastic optimization theory* and its application to deep learning.

## Resumen

Desde que la tecnología ha irrumpido en nuestra sociedad con tanta firmeza, el ser humano siempre ha soñado con llevarla a su máximo exponente. De entre todos esos sueños, algunos más factibles que otros, se haya la capacidad de crear maquinas que piensen por sí mismas. Esta quimera, pese a ser diferente a como la imaginaron nuestros antepasados, se haya en el punto álgido de su historia. El aumento masivo de datos, debido a la digitalización, y la mejora tecnológica constante, en este caso, a nivel de hardware, han sido los impulsores principales de este hecho.
Este trabajo de final de grado abarcará un área concreta que, ocasionalmente, forma parte del proceso de creación anteriormente mencionado, conocido por *Inteligencia Artificial*. Esta área se denomina *Deep Learning* o *Aprendizaje profundo*. El aprendizaje profundo se conoce como el conjunto de algoritmos que aprovechan los datos utilizando modelos de composición de funciones no lineales. En este aspecto, la optimización juega un papel fundamental, ya que es el controlador que transforma la información de los datos en los parámetros del modelo.
El trabajo estará centrado en entender la base matemática de esta optimización, llamada *Optimización Estocástica* y su aplicación en Deep Learning.

2010 Mathematics Subject Classification. 68T01, 68T05, 68T20, 68T30

*"The rise of powerful AI will either be the best or the worst thing ever to happen to humanity"*

— Stephen Hawking

## Agradecimientos

*Dedicado a Teodora y Rosa. Siempre conmigo.*

*En primer lugar, agradezco enormemente la ayuda y compresión recibida por parte de mi tutor del trabajo, Oriol.*

*En segundo lugar, agradezco a mi familia y amigos por todo el apoyo durante toda mi etapa universitaria. Han sido un pilar inquebrantable que me ha soportado todos estos años, que no han sido fáciles. Con especial mención a mi pareja, que ha sido la persona que más me ha aguantado y más cariño me ha brindado.*

*Para acabar, a todo el personal de la universidad, gracias por la educación y el servicio dado durante estos años. Ha sido un placer.*

# Índice

**1. Introducción** **1**

**2. Machine Learning** **3**

2.1. Conceptos básicos . . . . . 3
2.2. Usos del aprendizaje automático . . . . . 3
2.3. Experiencia . . . . . 4
2.4. Criterios de medición . . . . . 5
2.5. Capacidad y ajustes . . . . . 5
2.6. Regularización . . . . . 7
2.7. Estructura de los algortimos en Machine Learning . . . . . 7
2.7.1. Espacio de hipótesis . . . . . 7
2.7.2. Función de coste . . . . . 8
2.7.3. Búsqueda del método de optimización . . . . . 10
2.8. Deep Learning . . . . . 10
2.8.1. Arquitectura del perceptrón multicapa . . . . . 15
2.8.2. Función de activación . . . . . 16
2.8.3. Back propagation . . . . . 18
2.8.4. Teorema de aproximación universal . . . . . 22

**3. Optimización** **24**

3.1. Introducción . . . . . 24
3.1.1. Aplicaciones . . . . . 25
3.1.2. Soluciones frente a la optimización . . . . . 25
3.2. Optimización de mínimos cuadrados . . . . . 26
3.2.1. Usos y variantes de la optimización de mínimos cuadrados . . . . . 26
3.3. Programación lineal . . . . . 27
3.3.1. Usos de la programación lineal . . . . . 28
3.4. Optimización convexa . . . . . 29
3.4.1. Usos de la optimización convexa . . . . . 29
3.5. Optimización estocástica . . . . . 30

**4. Stochastic subgradient methods** **31**

4.1. Subgradientes . . . . . 31
4.1.1. Existencia . . . . . 32
4.1.2. Unicidad . . . . . 32
4.1.3. Algunas formas de cálculo de subgradientes . . . . . 32

4.2. Función de valor óptimo en optimización convexa . . . . . . . . . . . . . . 34
4.3. Métodos con subgradientes . . . . . . . . . . . . . . . . . . . . . . . . . . 35
4.3.1. Introducción . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
4.3.2. Método con subgradiente negativo básico . . . . . . . . . . . . . . 36
4.3.3. Reglas para el paso de optimización (step size) . . . . . . . . . . . 36
4.3.4. Convergencia de los métodos con subgradientes . . . . . . . . . . . 37
4.3.5. Cota óptima y criterio de parada . . . . . . . . . . . . . . . . . . . 39
4.4. Métodos con subgradientes estocásticos . . . . . . . . . . . . . . . . . . . 40
4.4.1. Subgradiente insesgado con ruido . . . . . . . . . . . . . . . . . . . 40
4.4.2. Métodos con subgradientes estocásticos básicos . . . . . . . . . . . 40
4.4.3. Convergencia en métodos con subgradientes estocásticos . . . . . . 41
4.4.4. Aplicaciones en Machine Learning y variantes . . . . . . . . . . . . 42

**5. Práctica: Reconocimiento de dígitos 45**

**6. Conclusiones 47**

**7. Anexos 48**

7.1. Código en Python de una red MLP fully conncected para reconocimiento de dígitos . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48

# 1. Introducción

Antes que nada, se introducirán algunas nociones básicas sobre Inteligencia Artificial, Machine Learning y Deep Learning. Pese a no existir definiciones formales, se intentará hacer entender al lector las diferencias entre ellas.

La **Inteligencia Artificial**, o IA, como su nombre indica, es la inteligencia llevada a cabo por máquinas. Como es lógico, al ser la inteligencia un concepto abstracto, dar una definición exacta es imposible.

En lo que nos atañe, podemos considerar la inteligencia artificial como una rama de las ciencias de la computación, la cual estudia las diferentes técnicas o procesos para crear máquinas que sean capaces de interpretar ciertos datos y, mediante la experiencia, ir mejorando los resultados de los objetivos para las que fueron creadas. Esto de manera menos formal, es equivalente a máquinas que simulen funciones cognitivas de algunos seres vivos, tales como percibir, razonar, aprender y solventar problemas.

No obstante, cabe destacar que pese a que la inteligencia humana o animal ha influido en la forma en la que diseñan estas máquinas inteligentes, hay casos en los que el parecido entre inteligencia e inteligencia artificial es nulo.

Asimismo, encontramos el **Machine Learning** o **Aprendizaje Automático**. El Machine Learning se define como un subcampo de la inteligencia artificial, el cual estudia los diferentes algoritmos que permiten a las máquinas aprender.

Técnicamente, una máquina aprende cuando, utilizando diferentes herramientas matemáticas, consigue ir mejorando los resultados esperados cuanta mayor es su experiencia. Es decir, es capaz de aumentar sus probabilidades de éxito en su cometido, a medida que va ejecutándose y recibiendo más datos de entrada. Evidentemente, la forma en la que se valoran los resultados y su mejoría, depende del tipo de problema y propósito que tengamos.

En el apartado 2, nos centraremos en introducir las nociones básicas de Machine Learning, así como la estructura de los algoritmos y las características de los modelos para, a continuación, dar paso a hablar extensamente sobre Deep Learning.

El **Deep Learning**, o aprendizaje profundo, no es más que un caso particular de Machine Learning. Se le conoce como *deep*, o *profundo*, porque los algoritmos que se utilizan están compuestos de varias capas, que desglosan un tipo de problema complejo en varios problemas más simples, que son más fáciles de resolver para la máquina.

Para acabar de entender esto último, utilizaremos un pequeño ejemplo. Imaginemos un programa que tenga que determinar si, dada una fotografía, en ella se encuentra un animal o una persona. Una imagen, para un ordenador, es un conjunto de píxeles. Dar una respuesta (con un error suficientemente pequeño) a partir únicamente de los datos de los píxeles, es muy complicado. Ahí es donde entra el Deep Learning. Utilizando capas intermedias u ocultas de información, como por ejemplo: limites, donde el píxel cambia bruscamente de brillo; bordes o contornos, que son conjuntos de límites; objetos, que son conjuntos cerrados de bordes; se transforma el problema en otros más sencillos que, encadenados, conseguirán dar una respuesta estimada, con un índice de acierto sustancialmente alto. Todo esto, dependiendo siempre, de las características de la imagen.

La profundidad del modelo se define como la cantidad de capas de información que tiene. Como las capas de información no tienen una definición fija, el mismo modelo puede tener diferente profundidad según las capas que se consideren. La definición de las capas o

de que manera están conectadas entre sí, depende del problema propuesto y el propósito que queramos llevar a cabo. Como se ha mencionado, el potencial del Deep Learning se basa en su capacidad de resolver problemas no lineales, mediante el uso de la composición de funciones. Hablaremos del modelo más tradicional de Deep Learning, el perceptrón multicapa o MLP. Veremos como la información fluye en el algoritmo y como se llega a optimizar.

La optimización, sobretodo estocástica, debido a la enorme cantidad de datos a tratar, está muy presente en este tipo de modelos.

En la sección 3, se introducirán brevemente los conceptos relacionados con la optimización matemática. Se verán los diferentes tipos, las soluciones que tenemos en este tema, y sus diferentes usos prácticos hoy en día. La idea es dar un base mínima sobre que es la optimización, que variantes existen, y como podemos utilizarla para lo que nos atañe, en este caso, la relación entre optimización y aprendizaje automático.

Para finalizar la teoría, en el apartado 4, veremos detalladamente el método de optimización por excelencia en aprendizaje profundo: el método con subgradientes estocásticos. Se darán las definiciones oportunas y se tratarán las diferentes propiedades asociadas al mismo. Hablaremos también de alguna variante del método y de su aplicación directa a los modelos de aprendizaje profundo.

Para acabar, en la sección 5, veremos un ejemplo de un algoritmo de Deep Learning que es capaz de resolver un problema concreto. Durante el proceso, se detallará el código de programación utilizado y se verá como los conceptos principales que se han explicado en el trabajo, se utilizan en la práctica para resolver, con soltura y precisión, un problema complejo.

# 2. Machine Learning

## 2.1. Conceptos básicos

El aprendizaje profundo es un subcampo del aprendizaje automático. Por lo tanto, para entender el primero hace falta tener un conocimiento amplio del segundo.
Como se ha mencionado antes, dar una definición precisa de inteligencia artificial no es posible. Así como tampoco lo es dar una definición unánime sobre qué entendemos al decir que una máquina aprende. A grandes rasgos, citando a Tom M. Mitchell (1997): «Decimos que un programa de ordenador aprende de una experiencia $E$, respecto a un ejercicio $T$, medido con un criterio $C$, cuando su desempeño en el ejercicio $T$, mejora con la experiencia $E$, si lo medimos con el criterio $C$.» Esta definición, pese a no pretender darla como única válida, se utilizará para profundizar un poco más en los diferentes tipos de ejercicios, experiencias, y criterios de medición que encontramos hoy día, y sirven para construir los algoritmos del Machine Learning.

## 2.2. Usos del aprendizaje automático

Empezaremos con los diferentes tipos de acciones o ejercicios más comunes, con sus respectivos ejemplos, para ver los diferentes usos que pueden tener los algoritmos en Machine Learning:

- **Clasificación**: Consiste en determinar a que conjunto pertenecen los datos de entrada. Para este objetivo, se suele utilizar una función $f : \mathbb{R}^n \to \mathbb{N}$, definida por $f(x) = y$, donde a partir de unos datos $x \in \mathbb{R}^n$, el modelo determina un número natural $y$, que está asociado a una clasificación concreta. En otros casos, en vez de devolver valores naturales, la función nos devuelve una distribución de probabilidad sobre clasificaciones o clases.
  Un ejemplo de algoritmo de clasificación sería aquel que hemos nombrado anteriormente, determinar si dada una imagen, en ella aparece una persona o un animal. Evidentemente, todo tipo de programa que sirva para reconocer objetos a partir de cualquier tipo de imagen, entra dentro de esta categoría. En este tipo de programas, el aprendizaje profundo está muy presente.
  Cabe destacar, que en muchos casos de clasificación la entrada no está completa. Es decir, faltan cierto tipo de datos. Esto se debe a la imposibilidad o rentabilidad de obtener éstos. Para esquivar este problema, en vez de utilizar una única función como la vista anteriormente, se utilizan varias y se juega con su distribución en probabilidad en las variables comunes más relevantes.

- **Transcripción**: Se trata de algoritmos que, a partir de algún tipo parcialmente desestructurado de datos, devuelven los datos de forma textual. Por ejemplo, un algoritmo que dada una imagen de un texto, devuelve el texto en sí. Otro ejemplo serían los algoritmos de reconocimiento de voz, que a partir de la forma de las ondas de una archivo de audio, devuelven en texto lo que han interpretado.

- **Traducción**: El más famoso es *Google Translate*. Son algoritmos que a partir de caracteres en un lenguaje concreto, devuelven los caracteres en otro lenguaje. Últimamente el Deep Learning ha tenido una gran participación en este tipo de algoritmos con tal de mejorar las traducciones dadas.

- **Regresión**: Es similar a la tarea de *Clasificación*. La gran diferencia es que en vez de clasificar sobre $\mathbb{N}$, lo hace sobre $\mathbb{R}$. Por lo tanto, en la gran mayoría de los casos, el algoritmo recibe de entrada un vector de datos de dimensión $n$ y devuelve un valor numérico real. Muy útil para previsiones de gastos o cálculo de precios de aseguradoras, entre otros.

- **Detección de anomalías**: Se le pide al programa que analice una serie de datos y encuentre si hay alguno que no cuadra. El ejemplo más conocido es la detección de movimientos inusuales en tarjetas de crédito.

- **Salida estructurada**: Son los algoritmos a los que se les pide una salida, normalmente un vector, que tenga una fuerte correlación entre las variables que devuelve. Los programas de análisis sintáctico de oraciones y de segmentación de imágenes son algunos de los ejemplos de esta categoría.

- **Recreación**: A partir de los datos, el algoritmo debe recrear nuevos ejemplos similares. Muy utilizado en la industria del videojuego para generar texturas de objetos, ya que hacer píxel a píxel sería una pesadilla.

- **Estimadores de funciones de densidad o de funciones de probabilidad**: En este tipo de tarea, el algoritmo debe determinar, a partir de los datos de entrada, una función de densidad o una función de probabilidad que sintetice el comportamiento de los datos. En la gran mayoría de ejemplos nombrados anteriormente, es necesario este proceso para facilitar el aprendizaje de la máquina y mejorar su cometido. Por otra parte, en muchos casos, los cálculos necesarios para realizar este proceso son intratables a nivel computacional. Para superar este obstáculo, se suelen reajustar las dimensiones implicadas en el proceso, para hacer computacionalmente posible el cálculo necesario. Como uno puede esperar, el uso de estimadores está muy presente en el proceso.

Hay varias tareas más en las que el Machine Learning está presente, pero el objetivo de este trabajo no es darlas todas, sino nombrar las principales y poner un poco en contexto al lector. Evidentemente, los algoritmos pueden resolver y abarcar, parcial o totalmente, varias de estas tareas a la vez. Todo depende siempre del objetivo que queramos alcanzar.

## 2.3. Experiencia

En cuanto a experiencia en Machine Learning, los algoritmos pueden ser categorizados, a grosso modo, en dos tipos: supervisados o no supervisados.
Un modelo o algoritmo supervisado es aquel al que se le da un resultado esperado por cada entrada, a partir del cual irá aprendiendo. De manera más coloquial, se le está diciendo al programa cuándo lo está haciendo bien o mal cada vez que da una salida. Por otra parte, los algoritmos no supervisados sacan conclusiones según los patrones que extraen de los datos y no necesitan de la intervención de un resultado esperado para aprender. Sobretodo en este último caso, hay una fuerte presencia de la teoría de la probabilidad. A grandes rasgos, digamos que los algoritmos no supervisados intentan obtener, implícitamente o explícitamente, una función de probabilidad (o de densidad) a partir de una cantidad masiva de ejemplos, normalmente aleatorios. Hay casos que en vez de obtener directamente la función, los algoritmos utilizan algunas propiedades derivadas de los datos para su posterior análisis.

De nuevo nos encontramos con definiciones poco formales, por lo tanto, la clasificación entre estos dos tipos de algoritmos suele ser variable, llegando a ver incluso casos mixtos o semisupervisados.

Por otra parte, la forma en la que llegan los datos es muy variada. La forma más común que tiene un conjunto de datos de entrada es en forma de matriz. Dependiendo de si las matrices de datos son cuadradas o no, se deben reajustar de diferentes formas con tal de optimizar el cálculo. Un ejemplo de conjunto de datos heterogéneos seria una serie de imágenes con tamaños diferentes. La idea general es facilitar el trabajo de codificar una información compleja y multivariable.

## 2.4. Criterios de medición

Hablemos ahora sobre los criterios de medición en los algoritmos del Machine Learning. Para evaluar cuando un modelo está funcionando como debe, hay diferentes maneras. En algoritmos sencillos, se puede evaluar el desempeño de la máquina mediante la precisión del modelo. La precisión del modelo puede calcularse mediante, por ejemplo, un índice de error. Es decir, casos correctos entre casos totales.

Para tareas más complejas, se tiene que recurrir a estrategias más enrevesadas para determinar la eficacia del modelo, siempre dependiendo del objetivo y del algoritmo que tengamos enfrente. Se debe pensar que forma de medición se ajusta mejor a nuestro objetivo. En muchos casos, este es uno de los grandes retos a la hora de crear el modelo, determinar como medimos que éste esté yendo bien. En el apartado 2.7.2, se hablará sobre las funciones de coste, que son la quintaesencia de los criterios de medición.

## 2.5. Capacidad y ajustes

Uno de los principales objetivos que tiene un algoritmo en Machine Learning, es la capacidad de actuar correctamente delante de escenarios no vistos. Esta habilidad se denomina *generalización*.

Normalmente, cuando entrenamos a un modelo de Machine Learning, lo hacemos mediante una serie de datos de entrenamiento. A raiz de estos datos, se minimiza el error. Posteriormente, se utilizan unos datos de muestreo para comprobar el funcionamiento del algoritmo delante de nuevos datos. El error esperado por cada entrada de nuevos datos se conoce como error de generalización. Esta es la principal diferencia entre Machine Learning y optimización: el tratamiento del error de generalización.

Un algoritmo basado en aprendizaje automático debe afrontar dos objetivos: minimizar el error de entrenamiento y hacer lo más pequeño posible la brecha entre el error de testeo y el error de entrenamiento. Estos procesos producen, si no se llevan correctamente a cabo, subajuste (underfitting) y sobreajuste (overfitting).

Si un algoritmo tiene overfitting, quiere decir que depende demasiado del entrenamiento (sobreentrenamiento) y ante nuevos datos no procederá bien. Por otro lado, si un modelo tiene underfitting es que el modelo no ha minimizado lo suficiente el error de entrenamiento, dando lugar a predicciones poco precisas. Que un algoritmo esté ajustado (sin sobreajuste ni subajuste) se logra modificando su capacidad. Informalmente, la capacidad de un modelo es su habilidad de ajuste a una amplia serie de funciones. La capacidad se modifica reduciendo, o ampliando, el espacio hipotético de soluciones. En un problema de regresión por ejemplo, considerar funciones de grado superior seria un ejem-

plo de aumento de capacidad. A continuación, veremos como en un ejemplo práctico, una función de grado 1 produce underfitting, y una de grado muy grande produce overfitting.

| | Underfitting | Just right | Overfitting |
|---|---|---|---|
| Symptoms | • High training error<br>• Training error close to test error<br>• High bias | • Training error slightly lower than test error | • Very low training error<br>• Training error much lower than test error<br>• High variance |
| Regression illustration | | | |
| Classification illustration | | | |
| Deep learning illustration | Error, Validation, Training, Epochs | Error, Validation, Training, Epochs | Error, Validation, Training, Epochs |
| Possible remedies | • Complexify model<br>• Add more features<br>• Train longer | | • Perform regularization<br>• Get more data |

Figura 1: Ejemplo de cuando en un modelo se produce overfitting y underfitting. Como podemos observar, la recta no es capaz de minimizar bien el error de entrenamiento. En cambio, la función de grado alto se adapta demasiado a los datos de entrenamiento y no es capaz de hacer pequeño el error de generalización (la diferencia entre error de entrenamiento y error del test es demasiado grande). También vemos el comportamiento del error en modelos de aprendizaje profundo.
Fuente: stanford.edu

Como principales soluciones a estos problemas, para un modelo sobreajustado, debemos obtener más datos y regularizar (2.6) el algoritmo. En cambio, en un modelo subajustado, debemos entrenar más al modelo o darle más complejidad, aumentado su capacidad. Para acabar la explicación sobre el ajuste de los algoritmos, pondremos un ejemplo simple que facilitará el entendimiento del lector.

Imaginemos que tenemos un algoritmo que debe detectar si en una imagen hay un perro o no. Si en el entrenamiento, de entrada sólo obtiene imágenes de un único tipo de raza, al introducir una imagen de un perro de otra raza, el algoritmo fallará. Se produce underfitting. Ahora, si entrenamos a la máquina con imágenes de perros de diferentes razas pero todas con un rasgo común, por ejemplo el color marrón, al pedirle que identifique una imagen con un perro de otro color, el algoritmo fallará. Esto sería overfitting. Evidentemente, con fallar nos referimos a que la eficiencia del modelo será muy baja y errará muchas veces.

## 2.6. Regularización

En términos generales, regularizar un modelo consiste en, mediante unas penalizaciones, definir alguna preferencia de una función sobre otra en el espacio hipotético de soluciones. Por ejemplo, si nuestro espacio hipotético de soluciones contiene únicamente (por las características de los datos) funciones polinómicas, una forma de regularizar seria ponerle una penalización mayor a las funciones con grado mayor. Para que el algoritmo escoja una función de grado mayor, ésta tendría que tener, dependiendo de la penalización, un error significativamente más bajo que una de grado menor. Vulgarmente, regularizar es poner determinadas preferencias en las soluciones que queremos obtener. Como se ha mencionado antes, esto nos sirve para corregir modelos con overfitting.
En el ejemplo de la figura 1, la regularización nos permitiría hacer que el algoritmo escogiera una función de grado 2, por encima de una de grado superior que produciría sobreajuste. Como es lógico, existen todo tipo de regulaciones en función del objetivo del modelo y la complejidad que se pueda alcanzar, computacionalmente hablando.

## 2.7. Estructura de los algortimos en Machine Learning

Después de haber introducido los principales conceptos derivados de Machine Learning, vamos a pasar a la práctica. Los algortimos en Machine Learning, generalmente, comparten la misma estructura. Esta estructura se sostiene en tres pilares principales: el espacio de hipótesis, la función de coste y el método de optimización.

### 2.7.1. Espacio de hipótesis

Conocemos como espacio de hipótesis, o espacio de parámetros, al conjunto de hipótesis que son consistentes con los datos. Básicamente, son las hipótesis que tiene sentido considerar teniendo en cuenta de dónde proceden los datos.

Por ejemplo, si nuestros datos son imágenes desde un satélite, que queremos que sirvan para hacer previsiones meteorológicas, considerar si la imagen puede contener un animal, no tiene sentido. Este escenario quedaría claramente fuera de nuestro espacio de hipótesis. Cuando estamos delante de un problema concreto, se escoge el espacio de hipótesis a raíz de los resultados que queramos obtener.

Un punto del espacio de hipótesis sería un modelo con unos parámetros específicos que, evidentemente, tenga sentido considerar. Como es lógico, la cantidad de espacios de parámetros es muy variada y es un concepto más bien abstracto sin definición muy firme.

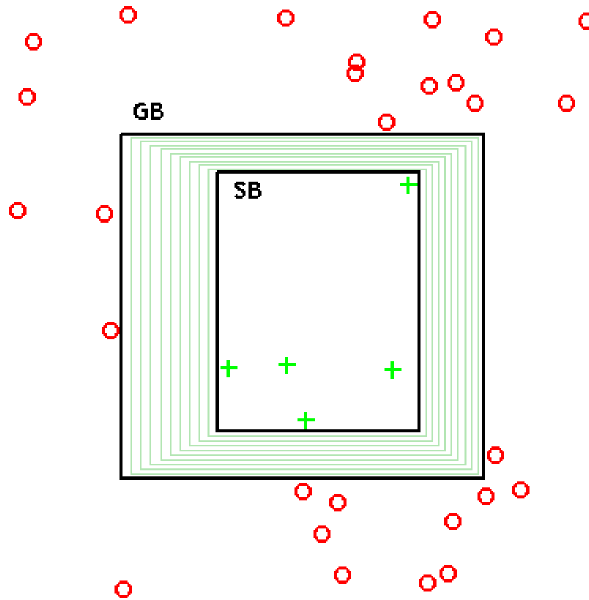


Figura 2: En este ejemplo se puede observar un espacio de hipótesis en forma de rectángulo. Normalmente, se define una cota superior, llamada *cota de máxima generalidad de hipótesis positivas* $GB$, y una inferior, definida como *cota de máxima especificidad de hipótesis positivas* $SB$. Como se puede intuir, la cota de máxima generalidad engloba todos los ejemplos positivos y cubre el máximo espacio posible, sin llegar nunca a incluir casos negativos. Por otro lado, la cota de máxima especificidad cubre el mínimo espacio posible, incluyendo siempre todos los casos positivos.
Fuente: Wikipedia

### 2.7.2. Función de coste

La función de coste o pérdida es el método principal para medir el desempeño de un modelo en Machine Learning. Es una función que a partir de los datos pertinentes, nos devuelve un valor, que representa un coste intuitivo al evento que estamos considerando. Es la función que tenemos que minimizar para entrenar al modelo.

En este trabajo, usaremos esta notación para referirnos a una función de coste:

$$\mathcal{L}(y_i, f(x_i, \theta))$$

donde $y_i$ es el valor esperado por cada ejemplo $x_i$ que nos devuelve la función $f$ y $\theta$ es el espacio de parámetros. Tanto $x_i$ como $y_i$, pueden ser vectores, o matrices, de dimensión cualquiera.

La idea principal es dar un coste asociado a todos los datos de entrenamiento del algoritmo. Hay que tener en cuenta, que después vamos a querer minimizar esta función mediante algún método de optimización, por lo tanto, la elección de ésta es muy importante en el proceso de creación del algoritmo.
Veamos unos ejemplos de función de coste.

- **Error cuadrático medio**

$$\mathcal{L}(y_i, f(x_i, \theta)) = \frac{1}{n}\sum_{i=1}^{n}(y_i - f(x_i, \theta))^2$$

Penaliza mucho las predicciones $f(x_i, \theta)$ que están muy alejadas del valor esperado, además su interpretación no suele ser sencilla. No obstante, sus propiedades permiten calcular gradientes más fácilmente. Es de las más utilizadas.

- **Raíz cuadrada del error cuadrático medio**

$$\mathcal{L}(y_i, f(x_i, \theta)) = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - f(x_i, \theta))^2}$$

Ídem que su símil anterior. Muy usadas en modelos de regresión.

- **Error absoluto medio**

$$\mathcal{L}(y_i, f(x_i, \theta)) = \frac{1}{n}\sum_{i=1}^{n}|y_i - f(x_i, \theta)|$$

Su convergencia y diferenciación son más complicadas de calcular pero penaliza menos los valores alejados de los esperados y su interpretación suele ser más sencilla que las anteriores. Suele usarse en modelos más simples.

- **Error absoluto medio escalado o corregido**

$$\mathcal{L}(y_i, f(x_i, \theta)) = \frac{\frac{1}{n}\sum_{i=1}^{n}|y_i - f(x_i, \theta)|}{\frac{n}{n-1}\sum_{i=2}^{n}|y_{n-1} - f(x_n, \theta)|}$$

Es muy similar a la anterior, forzando la simetría. Se suele usar cuando tenemos una sola variable.

- **Entropía cruzada**
Es la función de coste por excelencia en tareas de clasificación en aprendizaje profundo. Vamos a ajustar un poco la notación para facilitar el entendimiento. Supongamos que tenemos un conjunto $A = \{1, 2, \cdots, M\}$ de clases. Cada nombre natural está asociado con una clasificación, por ejemplo, 1 = "perro", 2 = "gato", etc. La entropía cruzada (aplicada a funciones de coste) se define como:

$$-\sum_{c=1}^{M} y_{i,c}\log(p_{i,c})$$

donde $y_{i,c}$ es la función indicatriz que devuelve 1 si la clase $c$ es la clasificación correcta para la observación $i$, y 0 si es incorrecta. Por otro lado, $p_{i,c}$ es la probabilidad de que la observación $i$ sea de la clase $c$.

Normalmente, se suele considerar la función logística $p_{i,c}(x) = \frac{1}{1+e^{-w_i \cdot x_i}}$ como probabilidad del modelo. En este caso, $w_i$ son los pesos del modelo que optimizará el método de optimización correspondiente para minimizar la función de coste. Cabe destacar, que esta función de coste penaliza mucho una predicción de confianza (con probabilidad alta) que falla. Además, la entropía cruzada tiene una fuerte relación con la función de log-verosimilitud. Recordemos que la función de verosimilitud de un modelo se define como

$$\prod_i q_i^{Mp_i}$$

con $p_i$ siendo la probabilidad observada y $q_i$ la probabilidad esperada de una observación $i$. Por lo tanto, la log-versimiliud media se define como:

$$\frac{1}{M}\log\left(\prod_i q_i^{Mp_i}\right) = \sum_i p_i \log(q_i),$$

que se corresponde con la función de coste de la entropía cruzada con signo cambiado.

- **Función de coste Hinge**

$$\mathcal{L}(y_i, f(x_i, \theta)) = \sum_i \text{máx}(0, 1 - y_i f(x_i, \theta)),$$

Es muy utilizada para tareas de clasificación. Al ser una función no diferenciable, las técnicas con gradientes no son válidas para minimizarla. No obstante, los métodos con subgradientes, que veremos más adelante, si que sirven para minimizar este tipo de funciones.

Hay diferentes variantes y más funciones de coste de las que acabamos de ver. No obstante, se ha considerado que éstas eran las principales, ya que se son las más utilizadas en los algoritmos de aprendizaje automático.

### 2.7.3. Búsqueda del método de optimización

Una vez tenemos planteado el problema, el objetivo que queremos alcanzar, el espacio de hipótesis y la función de coste asociada, viene posiblemente la parte más delicada del proceso de creación del algoritmo: la elección del método de optimización. En cualquier modelo de Machine Learning, debemos utilizar la optimización, del tipo que haga falta, para minimizar nuestra función de coste.

Normalmente, los algoritmos de Machine Learning suelen tratar con una enrome cantidad de información. Por este motivo, la optimización estocástica se utiliza frecuentemente, ya que nos permite reducir el tamaño de los datos, garantizando que la precisión del modelo sea suficientemente alta.

No obstante, otros tipos de optimización pueden ser utilizados si fuera necesario, siempre dependiendo de la función a minimizar. En el apartado 3, introduciremos brevemente los diferentes tipos de optimización existentes.

Debido al objetivo de este trabajo, daremos más enfásis a la optimización estocástica por su mayor uso en los algoritmos de Machine Learning. En particular, en la sección 4, veremos los métodos de optimización por excelencia en aprendizaje automático, los métodos con subgradientes (o gradientes) estocásticos, o stochastic subgradient methods.

## 2.8. Deep Learning

Los algoritmos simples de Machine Learning son capaces de resolver una amplia cantidad de problemas importantes. No obstante, no son especialmente eficaces en algunos pilares de la inteligencia artificial, como por ejemplo, reconocimientos de voz o de objetos. El desarrollo del aprendizaje profundo ha sido motivado en gran parte para solventar estos problemas.

A medida que la dimensión de los datos aumentan, los algoritmos más simples del Machine Learning, como la optimización a través de los estimadores de máxima verosimilitud; el uso de la probabilidad condicionada de la teoría de Bayes; las máquinas de vectores de soporte de Vladimir Vapnik; se enfrentan a un gran obstáculo: generalizar.

La generalización para nuevos ejemplos aumenta exponencialmente en dificultad a medida que los datos son de dimensión mayor. A su vez, los algoritmos simples antes nombrados no son capaces de aprender funciones en espacios muy grandes, que producen un coste computacional masivo. A este problema se le conoce generalmente como *Maldición de la dimensión.* Los algoritmos de Deep Learning nos permiten representar funciones de gran complejidad, descomponiendo éstas en capas y pasar por encima de los obstáculos de los que hemos hablado y otros varios.

Las redes neuronales profundas, o simplemente redes neuronales, son la base del Deep Learning. El objetivo de estas redes es aproximar alguna función $f^*$, que normalmente no es lineal. Para realizar esta aproximación, un modelo de Deep Learning define una serie de capas y unidades (elementos dentro de capas) los cúales permiten aproximar funciones muy complejas de manera simplificada utilizando la composición de funciones. Más formalmente, se trata de encontrar una aproximación de $f^*(x)$ mediante una función compuesta $f(x)$, definida por una composición de funciones $\{f_n\}_{n\in\mathbb{N}}$ tales que

$$f(x, \theta) = f_n(f_{n-1}(f_{n-2}(\cdots f_1(x, \theta))))$$

Cada una de estas funciones $f_i$ se conoce como capa, y a cada elemento dentro de una capa como unidad, neurona o perceptrón. El nombre *Redes Neuronales* proviene de la inspiración en un modelo simple del cerebro a la hora de crear este tipo de algoritmos. Más concretamente, vemos que la creación de las redes neuronales, de varios tipos, viene influenciada por diferentes funciones cognitivas de partes de un cerebro biológico. No obstante, como ya hemos mencionado, hay casos en que el parecido o influencia es nula.

La última capa, en este caso $f_n$, se denomina capa de salida, y la primera $f_1$, capa de entrada. El resto de funciones se conocen como capas ocultas o escondidas.

La profundidad de un algoritmo es la cantidad de capas que tiene. La definición de las capas depende siempre del problema a resolver, por lo tanto, un modelo puede tener más de una profundidad dependiendo de la elección de las capas. En la figura 5, podemos ver un ejemplo visual de las capas y neuronas en un modelo de Deep Learning, concretamente una red perceptrón multicapa, o MLP por sus siglas en inglés, multilayer perceptron.

En este caso, todas las neuronas de una capa, están conectadas con todas las neuronas de la capa anterior y siguiente, exceptuando la de entrada y salida, que solo conectan por un lado. Esto se conoce como red fully connected o completamente conectada.
Otra cualidad de la figura 5, es que es una red feedforward o prealimentada. Una red feedforward, es aquella donde la información fluye directamnte desde la capa de entrada a la de salida, pasando (o no) por capas ocultas, pero sin ningun tipo de bucle ni ciclo. Son el tipo de redes neuronales más utilizadas hoy día.

Existe una basta variedad de modelos de redes de aprendizaje profundo. Al ser diseños pensados para programación, se pueden diseñar modelos muy variados y hacer muchas mezclas entre ellos. Por ejemplo, la mejor manera que se ha visto hasta de ahora de un programa de reconocimiento de dígitos, como el que se hará más adelante en este trabajo, es haciendo una red con parte MLP y parte convolucional, que veremos en seguida que significa.

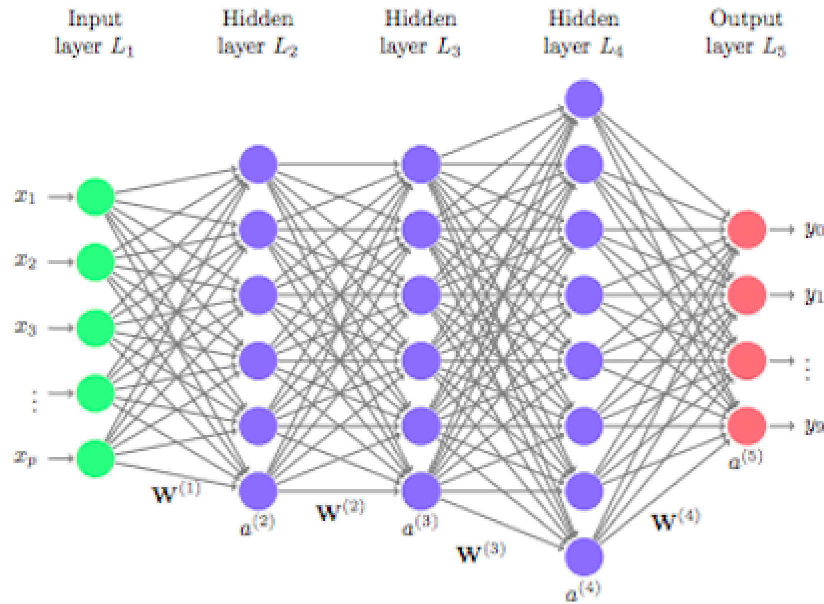


Figura 3: Grafo representativo de un algoritmo de Deep Learning. Como se puede observar, el modelo cuenta con cinco capas, incluyendo la de entrada y la de salida. Entre capa y capa, se puede observar una sucesión $W^{(i)}$, $i \in \{1,2,3,4\}$. Esta $W$ corresponde los pesos de cada capa y $a^{(i)}$ son las funciones de activación en cada capa. Más adelante veremos que significan exactamente. Fuente: UC Business Analytics R Programming Guide.

Básicamente, la base de todos los modelos de Deep Learning es la composición de funciones. Ahora bien, qué funciones componemos determina qué tipo de modelo vamos a tener. Por ejemplo, en los modelos MLP, como el que hemos visto, la información de cada neurona se transmite en forma de regresión lineal. Es decir, $g(x,w) = w^T x$, donde $x \in \mathbb{R}^n$ y $w \in \mathbb{R}^n$ es un parámetro que optimizará el método de optimización. En estos casos, es habitual añadir sesgos $b \in \mathbb{R}^n$, que también forman parte del espacio de parámetros, dando lugar a que las neuronas transmitan la información como $g(x,w,b) = w^T x + b$. Más adelante, veremos más en detalle la transmisión de información en este tipo de red.

Todo nuestro trabajo estará centrado en las redes MLP completamente conectadas. No obstante, vamos a ver brevemente otros tipos de redes que también se utilizan para resolver problemas de aprendizaje automático. Esta clasificación no es única, ya que la forma de las redes es muy variada según su propósito, y esto da lugar a muchas clasificaciones posibles. Simplemente, se quiere dar un poco de introducción al lector sobre otro tipo de redes neuronales, que no se trabajarán en este trabajo.

- **Redes neuronales convolucionales**: Este tipo de redes se han empezado a utilizar mucho últimamente. Se usan, básicamente, en reconocimiento de imágenes por su capacidad de detección de patrones visuales, inspiradas en una función biológica del córtex visual primario de un cerebro biológico. Una red neuronal convolucional es cualquier red neuronal que utilice una operación matemática llamada convolución en al menos una de sus capas. La convolución de dos funciones $f$ y $g$ y denotada como $f * g$ se define como

$$(f * g)(x) = \int_{-\infty}^{\infty} f(z) g(x - z) dz,$$

donde intervalo de integración depende del dominio de las funciones $f$ y $g$. Ahora, suponiendo que $f$ y $g$ están definidas para un entero $t$, podemos definir la convolución discreta como:

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

En el caso particular de las redes neuronales, la función $f$ se corresponde a la entrada que recibe la neurona, y $g$ se conoce como núcleo convolucional o kernel. Lo más habitual, es que la entrada sea una matriz de datos de la capa anterior, y el núcleo sea una matriz de parámetros que son ajustados por el algoritmo de optimización.

Si tenemos una imagen $I$ en dos dimensiones como input, lo más usual es utilizar el núcleo $K$ también de dos dimensiones. En este caso la definición de convolución seria:

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

donde la segunda igualdad se debe a que la convolución es una operación conmutativa.
La forma en la que aplicamos la convolución en las redes neuronales es bastante distinta a su aplicación en ingeniería y matemáticas. Se requiere de un proceso basado en tres partes principales para definir una típica capa convolucional.

En primer lugar, la capa realiza varias convoluciones para generar un conjunto de elementos lineales. En segundo lugar, estos elementos lineales se pasan por una función de activación no lineal. Finalmente, usando una función de agrupamiento o pooling, se modifica la salida para la siguiente capa.

Sea $x_j^{(L)}$ la $j$-ésima neurona de la capa $L$. Por definición, la salida que producirá esta neurona es:

$$x_j^{(L)} = g\left(b_j^{(L)} + \sum_k (K_{kj}^{(L)} * x_k^{(L-1)})(i, i_1)\right),$$

donde g es una función de activación no lineal, $K_{kj}^{(L)}$ es el núcleo convolucional asociado a la neurona $j$ de la capa $L$ y la $b_j^{(L)}$ es el sesgo asociado a la neurona $j$ de la capa $L$ y $k = 1, \cdots, \dim\{x^{(L-1)})\}$.

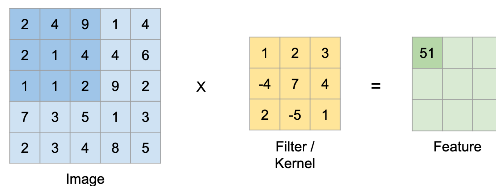


Figura 4: Aquí podemos ver gráficamente la convolución sobre una imagen. Como se puede observar, la matriz de datos se va multiplicando parcialmente (con las dimensiones adecuadas) por el núcleo convolucional dando como resultado la salida de la neurona, que es otra matriz.
Fuente: mc.ai.

En la fórmula que hemos dado antes, sobre la salida de una neurona convolucional, no se ha tenido en cuenta el proceso de pooling. Básicamente la función de pooling

sirve para reducir la dimensión de la matriz por motivos computacionales, mediante algun tipo de operación. Un ejemplo sería, suponiendo que tenemos una matriz cuadrada $A$ de dimensión 6, transformarla en una 2x2 usando el máximo valor dentro de los menores naturales de 3x3. Es decir:

$$A = \begin{pmatrix} a_{1,1} & \cdots & a_{1,6} \\ \vdots & \ddots & \vdots \\ a_{6,1} & \cdots & a_{6,6} \end{pmatrix} \to \hat{A} = \begin{pmatrix} \max\limits_{i,j \in \{1,2,3\}} \{a_{i,j}\} & \max\limits_{i \in \{1,2,3\},\, j \in \{4,5,6\}} \{a_{i,j}\} \\ \max\limits_{i \in \{4,5,6\},\, j \in \{1,2,3\}} \{a_{i,j}\} & \max\limits_{i,j \in \{4,5,6\}} \{a_{i,j}\} \end{pmatrix}.$$

El uso de una función de agrupamiento no está presente en todos los modelos convolucionales, pero es muy útil a la hora de aumentar la velocidad de cálculo del modelo mediante la reducción de las dimensiones de las imágenes.

Si el lector está interesado en indagar más en redes neuronales convolucionales, dejamos en las referencias [23] una web que se considera muy interesante, y amplia los temas que acabamos de introducir.

- **Redes neuronales de base radial**: Una red neuronal de base radial es un tipo de red, en la cual su salida viene determinada por una función de distancia a un punto central. La arquitectura típica de estas redes consta de tres capas: la de entrada, una oculta y la de salida. La capa oculta es donde se aplica la función de distancia, y la capa de salida se calcula de forma lineal. En términos formales, dada una entrada $x \in \mathbb{R}^n$, la salida de la red viene dada por la función $\phi : \mathbb{R}^n \to \mathbb{R}$ definida por

$$\phi(x) = \sum_{i=1}^{N} a_i \rho(||x - c_i||)$$

donde $N$ en la cantidad de neuronas en la capa oculta, $c_i$ es el centro asociado a la neurona $i$ y $a_i$ son los pesos asociados a la neurona $i$. Se suele escoger como norma $|| \cdot ||$ la distancia euclidiana, y como función de base radial

$$\rho(||x - c_i||) = \exp[-\beta ||x - c_i||^2].$$

Dado que $\lim\limits_{||x|| \to \infty} \rho(||x - c_i||) = 0$, cambiar los parámetros de una neurona tiene un impacto pequeño para valores de entrada muy alejados del centro.

- **Redes neuronales recurrentes**: Una red neuronal recurrente es aquella en que alguna capa contiene ciclos. La información de entrada a la red es devuelta una cantidad finita de veces a ella misma para, al acabar la recurrencia, devolver una salida.

Este tipo de redes están inspiradas en la capacidad de un cerebro orgánico de recordar información a través del tiempo. Básicamente, la red almacena la información de iteraciones anteriores y la usa para mejorar los parámetros. Las iteraciones se suelen denominar bucle temporal.

Las arquitecturas de las redes neuronales recurrentes, dependen de la cantidad de entradas y de salidas que tienen. Un ejemplo de una entrada simple y una salida múltiple es que, dada una imagen, el algoritmo devuelva un texto con sentido de que se está mostrando en ella. Aquí, la combinación entre red convolucional y recurrente mejora mucho el desempeño del algoritmo. Un ejemplo contrario, donde la entrada es múltiple y la salida simple, es que dado una oración, se le pida al algoritmo decir si es más sentimentalmente positiva o negativa. Finalmente, una red neuronal recurrente de varias entradas y varias salidas serviría, por ejemplo, para mejorar las traducciones de un idioma a otro.

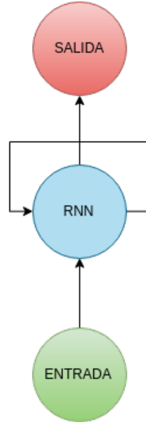


Figura 5: Grafo simplificado de una red neuronal recurrente. Como se puede observar, la red neuronal recurrente opera sobre si misma una cantidad finita de veces para posteriormente dar una salida. Fuente: meduim.com.

### 2.8.1. Arquitectura del perceptrón multicapa

Después de haber visto un poco los diferentes tipos de redes neuronales, empezaremos a hablar sobre el perceptrón multicapa, que será la red neuronal escogida para este trabajo.

Consideramos una red neuronal MLP fully connected de $L+1$ capas.

La notación $x^{(j)} \in \mathbb{R}^{p_j}$ con $p_j = \dim\{x^{(j)}\}$ nos servirá para referirnos a la capa oculta $j$-ésima de la red, o equivalentemente a la capa $j+1$ de la red, ya que consideramos que la capa $x^{(0)}$ es la de entrada, y $x^{(L)}$ la de salida. Para las neuronas o perceptrones, utilizaremos $x_i^{(j)}$ para referirnos a la $i$-ésima neurona de la $j$-ésima capa oculta.

Como hemos avanzado antes, la información en los modelos perceptrón multicapa se transmite mediante una regresión lineal. Usando la notación anterior, tenemos que el valor de cada neurona de la red será:

$$
\begin{array}{ccc}
x_1^{(1)} = F\left(\sum_{k=1}^{p_0} x_k^{(0)} w_{1,k}^{(1)} + b_1^{(1)}\right) & \cdots & x_1^{(L)} = F\left(\sum_{k=1}^{p_{(L-1)}} x_k^{(L-1)} w_{1,k}^{(L)} + b_1^{(L)}\right) \\
\vdots & & \vdots \\
x_{p1}^{(1)} = F\left(\sum_{k=1}^{p_0} x_k^{(0)} w_{p_1,k}^{(1)} + b_{p_1}^{(1)}\right) & \cdots & x_{p_L}^{(L)} = F\left(\sum_{k=1}^{p_{(L-1)}} x_k^{(L-1)} w_{p_L,k}^{(L)} + b_{p_L}^{(L)}\right).
\end{array}
$$

La idea detrás de esto es que cada neurona de una capa es la suma ponderada (con unos pesos $w_{i,j}$) de todas las neuronas de la capa anterior más un sesgo, aplicando posteriormente la función de activación $F$, que normalmente no es lineal.

Cabe destacar, que cada neurona de la red puede tener una función de activación diferente. Por simplificar un poco la notación, suponemos que todas las neuronas de la red tienen la misma función de activación.

Las ponderaciones, o pesos, y el sesgo de cada neurona son diferentes al resto de neuronas de la misma capa. Es decir, la diferencia entre la neurona $a$ y la neurona $b$ de la misma capa, solo está en los pesos, en el sesgo y, en pocas ocasiones, en la función de activación. Estos pesos y sesgos, normalmente se escogen aleatoriamente y es lo que se varia durante el aprendizaje de la red, para minimizar la función de coste.

Otra forma de escribirlo, es en forma matricial, que nos será muy útil a la hora de expresar posteriormente cálculos con la función de coste. Queremos expresar todas las neuronas de la primera capa oculta por ejemplo, es decir, queremos expresar el vector $x^{(1)}$. Entonces tenemos:

$$x^{(1)} = F\left(W^{(1)}x^{(0)} + b^{(1)}\right) = F\left(\begin{pmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,p_0} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,p_0} \\ \vdots & \vdots & \ddots & \vdots \\ w_{p_1,1} & w_{p_1,1} & \cdots & w_{p_1,p_0} \end{pmatrix} \begin{pmatrix} x_1^{(0)} \\ x_2^{(0)} \\ \vdots \\ x_{p_0}^{(0)} \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_{p_1} \end{pmatrix}\right).$$

Para facilitar el entendimiento, se ha resumido un poco la notación. En este caso, cuando hablamos de $w_{i,j}$ y de $b_l$ en realidad nos estamos refiriendo a $w_{i,j}^{(1)}$ y $b_l^{(1)}$ porque los pesos y el sesgo están asociados a la capa oculta 1. Esto se hace para diferenciar pesos y sesgos de neuronas de capas distintas. Aquí se ha vuelto a suponer que todas las neuronas de la red tienen la misma función de activación $F$.

### 2.8.2. Función de activación

En los modelos de Deep Learning, como por ejemplo la red fully connected que acabamos de ver, para transmitir la información, a cada neurona se le aplica una función $F$, que se conoce como función de activación. Esta función determina la salida de una neurona a partir de los datos de entrada, después de haber realizado ciertos cálculos. Principalmente, estas funciones nos permiten controlar la salida de las neuronas según nuestro objetivo.

Normalmente, las funciones de activación de una misma capa son todas iguales, a veces, incluso las de la red entera. No obstante, hay casos donde sobretodo la capa de entrada y salida tienen diferentes funciones de activación que el resto de capas. Todo esto depende del formato con el que llegan los datos y el tratamiento que queremos darle a éstos. Evidentemente, si tenemos una red neuronal que mezcla varios de los tipos que hemos visto antes, la elección de la función de activación varia según el tipo de capas.

La elección de una función de activación adecuada, según la red que tengamos, viene determinada por una serie de propiedades principales de las funciones de activación.

Si la función de activación es no lineal, entonces está demostrado que una red con dos capas es una aproximador universal, como veremos en la sección 2.8.4.

En cuanto a rango, si la función tiene rango finito, los métodos de optimización basados en gradientes suelen ser más estables. Por otro lado, si el rango de la función es infinito, el entrenamiento suele ser más eficiente, pero se suele necesitar un step size, o learning rate, (se explicará más adelante) más pequeño.

Evidentemente, si la función de activación es de clase $C^1$, nos permite usar mas fácilmente los algoritmos de optimización basados en gradientes, ya que éste es calculable directamente.

Si la función es aproximable a la identidad cerca del origen, la red neuronal procederá eficientemente con pesos iniciados aleatoriamente con valores pequeños.

En la siguiente tabla, podemos ver algunas de las funciones de activación que se utilizan para los modelos de aprendizaje profundo, así como su gráfica, su derivada, y su rango.

| Nombre | Ecuación | Derivada | Rango |
|---|---|---|---|
| Identidad | $f(x) = x$ | $f'(x) = 1$ | $(-\infty, \infty)$ |
| Paso binario | $f(x) = \begin{cases} 0 & \text{si } x < 0 \\ 1 & \text{si } x \geq 0 \end{cases}$ | $f(x) = \begin{cases} 0 & \text{si } x \neq 0 \\ ? & \text{si } x = 0 \end{cases}$ | $\{0, 1\}$. |
| Sigmoid | $f(x) = \sigma(x) = \frac{1}{1+e^{-x}}$ | $f'(x) = f(x)(1 - f(x))$ | $(0, 1)$ |
| Tanh | $f(x) = tanh(x) = \frac{(e^x - e^{-x})}{(e^x + e^{-x})}$ | $f'(x) = 1 - f(x)^2$ | $(-1, 1)$ |
| Arctan | $f(x) = tan^{-1}(x)$ | $f'(x) = \frac{1}{x^2}$ | $(-\pi/2, \pi/2)$ |
| Arcsenh | $f(x) = sinh^{-1}(x) = ln(x + \sqrt{x^2+1})$ | $f'(x) = \frac{1}{\sqrt{x^2+1}}$ | $(-\infty, \infty)$ |
| Softsign | $f(x) = \frac{x}{1+\lvert x \rvert}$ | $f'(x) = \frac{1}{(1+\lvert x \rvert)^2}$ | $(-1, 1)$ |
| ISRU | $f(x) = \frac{x}{\sqrt{1+\alpha x^2}}$ | $f'(x) = \left(\frac{1}{\sqrt{1+\alpha x^2}}\right)^3$ | $(-\frac{1}{\sqrt{\alpha}}, \frac{1}{\sqrt{\alpha}})$ |
| ISRLU | $f(x) = \begin{cases} \frac{x}{\sqrt{1+\alpha x^2}} & \text{si } x < 0 \\ x & \text{si } x \geq 0 \end{cases}$ | $f'(x) = \begin{cases} \left(\frac{1}{\sqrt{1+\alpha x^2}}\right)^3 & \text{si } x < 0 \\ 1 & \text{si } x \geq 0 \end{cases}$ | $(-\frac{1}{\sqrt{\alpha}}, \infty)$ |
| ReLU | $f(x) = max(0, x)$ | $f'(x) = \begin{cases} 0 & \text{si } x \leq 0 \\ 1 & \text{si } \lvert x \rvert > 0 \end{cases}$ | $[0, \infty)$ |
| Leaky ReLU | $f(x) = \begin{cases} 0{,}01x & \text{si } x < 0 \\ x & \text{si } x \geq 0 \end{cases}$ | $f'(x) = \begin{cases} 0{,}01 & \text{si } x < 0 \\ 1 & \text{si } x \geq 0 \end{cases}$ | $(-\infty, \infty)$ |
| PReLU | $f(x) = \begin{cases} \alpha x & \text{si } x < 0 \\ x & \text{si } x \geq 0 \end{cases}$ | $f'(x) = \begin{cases} \alpha & \text{si } x < 0 \\ 1 & \text{si } x \geq 0 \end{cases}$ | $(-\infty, \infty)$ |
| RReLU | $f(x) = \begin{cases} \beta x & \text{si } x < 0 \\ x & \text{si } x \geq 0 \end{cases}$ | $f'(x) = \begin{cases} \beta & \text{si } x < 0 \\ 1 & \text{si } x \geq 0 \end{cases}$ | $(-\infty, \infty)$ |
| SoftPlus | $f(x) = \ln(1 + e^x)$ | $f'(x) = \frac{1}{1+e^{-x}}$ | $(0, \infty)$ |
| Gaussiana | $f(x) = e^{-x^2}$ | $f'(x) = -2xe^{-x^2}$ | $(0, 1]$ |
| Seno | $f(x) = \sin(x)$ | $f'(x) = \cos(x)$ | $[-1, 1]$ |
| Identidad de Bent | $f(x) = \frac{\sqrt{x^2+1}-1}{2} + x$ | $f'(x) = \frac{x}{2\sqrt{x^2+1}} + 1$ | $(-\infty, \infty)$ |
| Softmax | $f_i(\vec{x}) = \frac{e^{x_i}}{\sum_{j=1}^{J} e^{x_j}}, \ i = 1, \cdots, J.$ | $\frac{\partial f_i(\vec{x})}{\partial x_j} = f_i(\vec{x})(\delta_{ij} - f_j(\vec{x}))$ | $(0, 1)$ |
| Maxout | $f(\vec{x}) = \max_i x_i$ | $\frac{\partial f}{\partial x_j} = \begin{cases} 1 & \text{si } j = \underset{i}{\operatorname{argmax}}\, x_i \\ 0 & \text{si } j \neq \underset{i}{\operatorname{argmax}}\, x_i \end{cases}$ | $(-\infty, \infty)$ |

Donde $\alpha > 0$ manteniendo el rango cierto, y $\delta_{ij}$ es la delta de Kronecker [24].
Como se pude observar, las dos últimas funciones de activación están definidas para toda la capa, mientras que el resto, están definidas para cada neurona.

### 2.8.3. Back propagation

Cuando en una red neuronal, la información fluye desde la entrada $x$ a la salida $\hat{y}$ pasando por todas las capas intermedias, este fenómeno se denomina propagación hacia adelante o forward propagation.

Por otra parte, durante el entrenamiento de una red neuronal, la función de coste se ha de ir reduciendo, buscando un mínimo local. Para eso, hacemos uso de métodos de optimización, de los que hablaremos más adelante. Normalmente, los métodos que utilizamos para entrenar a nuestra red, involucran el cálculo de gradientes o subgradientes. Ahora bien, calcular el gradiente de una función, que es una composición de muchas funciones, con normalmente cientos de variables, con métodos usuales, es computacionalmente muy caro. Para superar este problema, se utiliza la propagación hacia atrás. La propagación hacia atrás, o back propagation, es el método más utilizado en Deep Learning para calcular los gradientes que necesitamos durante el entrenamiento de la red.

Antes de nada, debemos recordar un resultado previo. La regla de la cadena es la fórmula que estipula el resultado de derivar una función que es una composición de dos o más funciones.

**Teorema 2.1.** ***Regla de la cadena.*** *Sea $g$ una función diferenciable en $x$ y $f$ una función diferenciable en $g(x)$, entonces:*

$$\frac{d}{dx}f(g(x)) = f'(g(x))g'(x).$$

*Alternativamente tenemos, por la notación de Leibniz:*

$$\frac{df}{dx} = \frac{df}{dg}\frac{dg}{dx}.$$

*Ahora, considerando la composición $f_1 \circ (f_2 \circ \cdots \circ (f_{n-1} \circ f_n))$, donde se supone que cada $f_i$ es diferenciable en su punto inmediato, aplicando la regla de la cadena reiteradamente, obtenemos, en notación de Leibniz, que la derivada es*

$$\frac{df_1}{dx} = \frac{df_1}{df_2}\frac{df_2}{df_3}\cdots\frac{df_n}{dx}.$$

Si el lector está interesado en la demostración, se pueden encontrar varias en las referencias [25].

Volviendo a las redes neuronales, la propagación hacia atrás es simplemente aplicar reiteradamente la regla de la cadena, para obtener un cálculo del gradiente de la función de coste. Vamos a entrar en detalle.

Empezaremos con un pequeño ejemplo muy básico, para entender el principio de la propagación hacia atrás. Supongamos que tenemos una red neuronal MLP fully connected, donde todas las capas tienen únicamente una neurona y que el número de capas de la red es $L + 1$.

Suponemos también que nuestra función de coste $\mathcal{L}$ es un error cuadrático medio $\frac{1}{n}\sum_{i=1}^{n}(y_i - f(x_i, \theta))^2$, donde $f(x_i, \theta))$ es la salida de nuestra red, que en nuestro caso es un simple valor real, y $y_i$ es valor esperado para cada ejemplo de entrenamiento $i$. Como vamos a considerar solo 1 ejemplo para el entrenamiento, y todas las capas tienen una única neurona, podemos considerar nuestra función de coste como $\mathcal{L}(y, x^{(L)}) = (y - x^{(L)})^2$,

donde siguiendo la notación de la sección 2.8.1, $x^{(L)}$ es la capa de salida de la red, que en este caso, es solo una neurona.

Recordemos que

$$x^{(L)} = F(w^{(L)}x^{(L-1)} + b^{(L)})$$

siendo $F$ una función de activación, $w^{(L)}$ los pesos y $b^{(L)}$ los sesgos. Para simplificar, supondremos que toda la red tiene la misma función de activación $F$. Ahora, sea $z^{(L)} = w^{(L)}x^{(L-1)} + b^{(L)}$. Es decir, $x^{(L)} = F(z^{(L)})$. Queremos computar el gradiente de la función de coste para un caso del entrenamiento, en este caso

$$\nabla_{(w,b)}\mathcal{L} = \begin{pmatrix} \frac{\partial \mathcal{L}}{\partial w^{(1)}} & \frac{\partial \mathcal{L}}{\partial b^{(1)}} \\ \vdots & \vdots \\ \frac{\partial \mathcal{L}}{\partial w^{(L)}} & \frac{\partial \mathcal{L}}{\partial b^{(L)}} \end{pmatrix}.$$

Cabe recordar, que en un caso práctico normal, la función de coste, en este caso el error cuadrático medio, se calcula utilizando la media de todos los errores cuadráticos calculados a partir de cada uno de los ejemplos. Es decir, si en vez de tener un ejemplo de entrenamiento, tenemos $n$ ejemplos (o $n$ inputs de información), el gradiente total se calcularía haciendo la media de los gradientes para cada ejemplo. Es decir,

$$\nabla_\theta \mathcal{L} = \frac{1}{n}\sum_{i=1}^{n} \nabla_\theta \mathcal{L}_i$$

donde $\theta$ son todos los parámetros del modelo y $\mathcal{L}_i$ es la función de coste asociada a cada ejemplo, en nuestro caso, $\mathcal{L}_i = (y_i - f(x_i, \theta))^2$ para cada ejemplo de entrenamiento $x_i$ que pasamos por la red $f$, y cada salida esperada, o target, $y_i$.

Prosiguiendo con nuestro particular ejemplo, vamos a ver como calcular, mediante propagación hacia atrás, todos los elementos del gradiente $\nabla\mathcal{L}$ suponiendo que solo hay un único ejemplo de entrenamiento.

Empezamos por la última capa. En nuestro caso, solo contiene un elemento $x^{(L)} = F(w^{(L)}x^{(L-1)} + b^{(L)})$, con $w^{(L)}, b^{(L)} \in \mathbb{R}$. Ahora bien, por la regla de la cadena sobre $\mathcal{L}(F(z^{(L)}))$:

$$\frac{\partial \mathcal{L}}{\partial w^{(L)}} = \frac{\partial \mathcal{L}}{\partial x^{(L)}} \cdot \frac{\partial x^{(L)}}{\partial z^{(L)}} \cdot \frac{\partial z^{(L)}}{\partial w^{(L)}},$$

$$\frac{\partial \mathcal{L}}{\partial b^{(L)}} = \frac{\partial \mathcal{L}}{\partial x^{(L)}} \cdot \frac{\partial x^{(L)}}{\partial z^{(L)}} \cdot \frac{\partial z^{(L)}}{\partial b^{(L)}}.$$

Recordemos que, $z^{(L)} = w^{(L)}x^{(L-1)} + b^{(L)}$, por lo tanto:

$$\frac{\partial z^{(L)}}{\partial w^{(L)}} = x^{(L-1)}, \quad \frac{\partial x^{(L)}}{\partial z^{(L)}} = F'(z^{(L)}), \quad \frac{\partial \mathcal{L}}{\partial x^{(L)}} = 2(x^{(L)} - y),$$

donde la última igualdad viene de que hemos escogido una función de coste de error cuadrático. Por otra parte, $\frac{\partial z^{(L)}}{\partial b^{(L)}} = 1$, por lo tanto:

$$\frac{\partial \mathcal{L}}{\partial w^{(L)}} = 2(x^{(L)} - y)F'(z^{(L)})x^{(L-1)},$$

$$\frac{\partial \mathcal{L}}{\partial b^{(L)}} = 2(x^{(L)} - y)F'(z^{(L)}).$$

Una vez calculada la primera iteración, aplicaremos la propagación hacia atrás viendo la relación entre dos capas consecutivas.

Para la capa $L-1$, aplicando de nuevo la regla de la cadena, tenemos que:

$$\frac{\partial \mathcal{L}}{\partial w^{(L-1)}} = \frac{\partial \mathcal{L}}{\partial x^{(L)}} \cdot \frac{\partial x^{(L)}}{\partial z^{(L)}} \cdot \frac{\partial z^{(L)}}{\partial x^{(L-1)}} \cdot \frac{\partial x^{(L-1)}}{\partial z^{(L-1)}} \cdot \frac{\partial z^{(L-1)}}{\partial w^{(L-1)}},$$

$$\frac{\partial \mathcal{L}}{\partial b^{(L-1)}} = \frac{\partial \mathcal{L}}{\partial x^{(L)}} \cdot \frac{\partial x^{(L)}}{\partial z^{(L)}} \cdot \frac{\partial z^{(L)}}{\partial x^{(L-1)}} \cdot \frac{\partial x^{(L-1)}}{\partial z^{(L-1)}} \cdot \frac{\partial z^{(L-1)}}{\partial b^{(L-1)}}.$$

Sea ahora $\delta^{(L)} = \frac{\partial \mathcal{L}}{\partial x^{(L)}} \cdot \frac{\partial x^{(L)}}{\partial z^{(L)}} = 2(x^{(L)} - y)F'(z^{(L)})$, que ya hemos calculado antes. Por definición de $z^{(L)}$, $\frac{\partial z^{(L)}}{\partial x^{(L-1)}} = w^{(L)}$, y razonando como antes, $\frac{\partial z^{(L-1)}}{\partial b^{(L-1)}} = 1$, $\frac{\partial z^{(L-1)}}{\partial w^{(L-1)}} = x^{(L-2)}$. Nos queda entonces que

$$\frac{\partial \mathcal{L}}{\partial w^{(L-1)}} = \delta^{(L)} w^{(L)} F'(z^{(L-1)}) x^{(L-2)},$$

$$\frac{\partial \mathcal{L}}{\partial b^{(L-1)}} = \delta^{(L)} w^{(L)} F'(z^{(L-1)}).$$

Si ahora definimos $\delta^{(L-1)} = \delta^{(L)} w^{(L)} \cdot F'(z^{(L-1)})$, tenemos la iteración para calcular el gradiente de todas las capas definida por:

$$\begin{aligned}
\delta^{(L)} &= \frac{\partial \mathcal{L}}{\partial x^{(L)}} \cdot \frac{\partial x^{(L)}}{\partial z^{(L)}} = 2(x^{(L)} - y)F'(z^{(L)}) \\
\frac{\partial \mathcal{L}}{\partial b^{(L)}} &= \delta^{(L)} \\
\frac{\partial \mathcal{L}}{\partial w^{(L)}} &= \delta^{(L)} \cdot x^{(L-1)}, \\
\\
\delta^{(L-1)} &= \delta^{(L)} \cdot w^{(L)} \cdot \frac{\partial x^{(L-1)}}{\partial z^{(L-1)}} = \delta^{(L)} \cdot w^{(L)} \cdot F'(z^{(L-1)}) \\
\frac{\partial \mathcal{L}}{\partial b^{(L-1)}} &= \delta^{(L-1)} \\
\frac{\partial \mathcal{L}}{\partial w^{(L-1)}} &= \delta^{(L-1)} \cdot x^{(L-2)} \\
&\vdots \\
\delta^{(J)} &= \delta^{(J+1)} \cdot w^{(J+1)} \cdot \frac{\partial x^{(J)}}{\partial z^{(J)}} = \delta^{(J+1)} \cdot w^{(J+1)} \cdot F'(z^{(J)}) \\
\frac{\partial \mathcal{L}}{\partial b^{(J)}} &= \delta^{(J)} \\
\frac{\partial \mathcal{L}}{\partial w^{(J)}} &= \delta^{(J)} \cdot x^{(J-1)}
\end{aligned}$$

para $J = L-2, \cdots, 1$.

Recordemos que este caso es el más sencillo, con solo una neurona por capa. Para extender la propagación hacia atrás para más neuronas por capa, simplemente hemos de ajustar la notación un poco y tener en cuenta que en vez de multiplicar reales, estaremos

multiplicando matrices y vectores.
Consideramos ahora el caso de una red neuronal, fully connected, con $L+1$ capas, las cuales tienen dimensión variable $p_i \in \mathbb{N}$, según la capa $i$ en la que estemos. Para simplificar, nuevamente suponemos que en toda la red, la función de activación será $F$. Recordemos que estos cálculos corresponden a un único caso de entrenamiento. Procedemos como en el ejemplo simple, y empezamos por la última capa. Ahora, en notación matricial:

$$x^{(L)} = F\left(W^{(L)}x^{(L-1)} + b^{(L)}\right) = F\left(\begin{pmatrix} w_{1,1}^{(L)} & w_{1,2}^{(L)} & \cdots & w_{1,p_{(L-1)}}^{(L)} \\ w_{2,1}^{(L)} & w_{2,2}^{(L)} & \cdots & w_{2,p_{(L-1)}}^{(L)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{p_L,1}^{(L)} & w_{p_L,2}^{(L)} & \cdots & w_{p_L,p_{(L-1)}}^{(L)} \end{pmatrix}\begin{pmatrix} x_1^{(L-1)} \\ x_2^{(L-1)} \\ \vdots \\ x_{p_{(L-1)}}^{(L-1)} \end{pmatrix} + \begin{pmatrix} b_1^{(L)} \\ b_2^{(L)} \\ \vdots \\ b_{p_L}^{(L)} \end{pmatrix}\right) =$$

$$= \begin{pmatrix} x_1^{(L)} \\ x_2^{(L)} \\ \vdots \\ x_{p_L}^{(L)} \end{pmatrix}$$

Sea $\mathcal{L} = (y_i - x_i^{(L)})^2,\ i \in \{1, \cdots, p_n\}$, la función de coste asociada a la red. En este caso, $y$ hace referencia a la entrada esperada por el único ejemplo de entrenamiento que estamos considerando. No hay que confundirlo con las $y_i$ mencionadas en las funciones de coste. Cada una de esas $y_i$ era un vector, en este caso $y$ es el único vector a considerar. Hay que tener en cuenta que ahora, $Z^{(L)} = W^{(L)}x^{(L-1)} + b^{(L)}$, donde $W^{(L)} \in \mathbb{R}^{p_L \times p_{(L-1)}}$ y $b^{(L)} \in \mathbb{R}^{p_L}$. Aplicando la regla de la cadena sobre $\mathcal{L}(F(Z^{(L)}))$ tenemos:

$$\nabla_{W^{(L)}}\mathcal{L} = \nabla_{W^{(L)}}Z^{(L)} \cdot \nabla_{Z^{(L)}}x^{(L)} \cdot \nabla_{x^{(L)}}\mathcal{L} = F'(Z^{(L)}) \cdot 2(y-x) \cdot (x^{(L-1)})^T$$
$$\nabla_{b^{(L)}}\mathcal{L} = \nabla_{b^{(L)}}Z^{(L)} \cdot \nabla_{Z^{(L)}}x^{(L)} \cdot \nabla_{x^{(L)}}\mathcal{L} = F'(Z^{(L)}) \cdot 2(y-x).$$

Como se puede observar, esencialmente es la misma idea que hemos visto en nuestro ejemplo simplificado. Lo único que hay que tener en cuenta al hacer el gradiente, es que las matrices implicadas en una composición de funciones se transponen y el orden de multiplicación se invierte [22]. Definiendo esta vez $\delta^{(L)} = \nabla_{Z^{(L)}}x^{(L)} \cdot \nabla_{x^{(L)}}\mathcal{L} = F'(Z^{(L)}) \cdot -2(y-x)$ que en este caso es un vector de $p_n$ componentes y aplicando la regla de la cadena nuevamente:

$$\nabla_{W^{(L-1)}}\mathcal{L} = \nabla_{W^{(L-1)}}Z^{(L-1)} \cdot \nabla_{Z^{(L-1)}}x^{(L-1)} \cdot \nabla_{x^{(L-1)}}Z^{(L)} \cdot \delta^{(L)},$$
$$\nabla_{b^{(L-1)}}\mathcal{L} = \nabla_{b^{(L-1)}}Z^{(L-1)} \cdot \nabla_{Z^{(L-1)}}x^{(L-1)} \cdot \nabla_{x^{(L-1)}}Z^{(L)} \cdot \delta^{(L)}$$

y teniendo en cuenta que

$$\nabla_{b^{(L-1)}}Z^{(L-1)} = \mathrm{Id}, \quad \nabla_{W^{(L-1)}}Z^{(L-1)} = x^{(L-2)}, \quad \nabla_{x^{(L-1)}}Z^{(L)} = (W^{(L)})^T,$$

obtenemos:

$$\begin{aligned}
\delta^{(L)} &= F'(Z^{(L)}) \cdot \nabla_{x^{(L)}}\mathcal{L}, \\
\nabla_{W^{(L)}}\mathcal{L} &= \delta^{(L)} \cdot (x^{(L-1)})^T, \\
\nabla_{b^{(L)}}\mathcal{L} &= \delta^{(L)}, \\
\\
\delta^{(L-1)} &= F'(Z^{(L-1)}) \cdot (W^{(L)})^T \cdot \delta^{(L)}, \\
\nabla_{W^{(L-1)}}\mathcal{L} &= \delta^{(L-1)} \cdot (x^{(L-2)})^T, \\
\nabla_{b^{(L-1)}}\mathcal{L} &= \delta^{(L-1)}, \\
&\vdots \\
\delta^{(J)} &= F'(Z^{(J)}) \cdot (W^{(J+1)})^T \cdot \delta^{(J+1)}, \\
\nabla_{W^{(J)}}\mathcal{L} &= \delta^{(J)} \cdot (x^{(J-1)})^T, \\
\nabla_{b^{(J)}}\mathcal{L} &= \delta^{(J)}
\end{aligned}$$

para $J = L-2, \cdots, 1$. Es muy importante tener en cuenta que en los productos para calcular las diferentes $\delta^{(i)}$ de dimensión $p_i$, el producto se realiza de derecha a izquierda.

Con este proceso, obtenemos el gradiente para un caso del entrenamiento. Por lo tanto, el proceso se debe repetir para cada caso de entrenamiento que tengamos, para, después de calcular la función de coste total, minimizarla utilizando los métodos de optimización con gradientes o subgradientes, que veremos en el apartado 4.4.

### 2.8.4. Teorema de aproximación universal

En aprendizaje profundo, la consideración principal sobre la arquitectura del modelo, es la profundidad y la dimensión de las capas. A destacar, un modelo de una única capa oculta es suficiente para entrenar unos datos de entrenamiento. No obstante, modelos más profundos suelen permitir dimensiones de capa más pequeñas y menos parámetros, además, son frecuentemente generalizables con los datos de prueba. En cambio, suelen ser más difíciles de optimizar. Por ahora, la manera para encontrar la arquitectura más óptima es mediante la experimentación, monitorizando el error de testeo. Un modelo completamente lineal puede representar, por definición, únicamente funciones lineales. No obstante, se suelen desear sistemas que no tengan porque ser lineales. Para resolver este problema, existe un teorema muy útil: *el teorema de aproximación universal.*

El teorema establece que una red neuronal con al menos una capa oculta, que contiene un número finito de neuronas, puede aproximar cualquier función continua en subconjuntos compactos de $\mathbb{R}^n$, bajo ciertas condiciones en la función de activación (definida en 2.8.2) que veremos en seguida.

**Teorema 2.2.** *Sea $\varphi : \mathbb{R} \longrightarrow \mathbb{R}$ una función de activación acotada, continua y no constante. Sea $I_m$ el hipercubo $[0,1]^m$ de dimensión $m$.*
*Consideramos el espacio $\mathbb{C}(I_m)$, definido como el espacio de funciones continuas reales sobre $I_m$. Entonces, para cualquier $\epsilon > 0$ y cualquier función $f \in \mathbb{C}(I_m)$, existen un entero $N \in \mathbb{Z}$, unas constantes $v_i, b_i \in \mathbb{R}$ y unos vectores $w_i \in \mathbb{R}^m$, $i \in \{1, 2, \cdots, N\}$ tales que podemos definir:*

$$F(x) = \sum_{i=1}^{N} v_i \varphi(w_i^T x + b_i)$$

*como una aproximación de $f$ tal que:*

$$|F(x) - f(x)| < \epsilon, \quad \forall x \in I_m.$$

*En otras palabras, funciones de la forma de $F(x)$ son densas en $\mathbb{C}(I_m)$.*
*El teorema también es aplicable sustituyendo $I_m$ por cualquier subconjunto compacto de $\mathbb{R}^m$.*

Existen ciertas variantes a este teorema, una de ellas estipula que:

**Teorema 2.3.** *Para cualquier función intergable Lebesgue $f : \mathbb{R}^n \longrightarrow \mathbb{R}$ y cualquier $\epsilon > 0$ existe una red neuronal completamente conectada ReLU [18] $\mathcal{B}$, con dimensión $d \leq n + 4$ tal que su función asociada $F_{\mathcal{B}}$ cumple:*

$$\int_{\mathbb{R}^n} |f(x) - F_{\mathcal{B}}(x)| dx < \epsilon$$

En resumen, este teorema y sus variantes nos permiten afirmar que una gran variedad de funciones pueden ser representadas y aproximadas mediante una red neuronal perceptrón multicapa con ciertas funciones de activación, como son, por ejemplo, las ReLU y las sigmoid. No obstante, no podemos afirmar que el algoritmo de optimización del modelo sea capaz de tratar correctamente la función, ni tampoco que vaya a tener un coste computacional admisible.

Si el lector está interesado en indagar más en las variantes y demostraciones de este teorema, dejamos en las referencias [16] información que extiende los temas que acabamos de hablar.

# 3. Optimización

## 3.1. Introducción

Un problema de optimización matemática, o simplemente problema de optimización, consiste en minimizar (o maximizar) una función $f_0(x) : \mathbb{R}^n \longrightarrow \mathbb{R}$ llamada función objetivo o de coste, con unas funciones de restricción $f_i(x) : \mathbb{R}^n \longrightarrow \mathbb{R}$, tales que

$$f_i(x) \leq b_i, \quad i = 1, \cdots, m.$$

Esto se escribe normalmente de la forma siguiente:

$$\begin{aligned} &\text{minimizar } f_0(x) \\ &\text{con retricciones } f_i(x) \leq b_i \quad i = 1, \cdots, m. \end{aligned}$$

El vector $x = (x_1, \cdots, x_n)$ se denomina variable de optimización y las constantes $b_1, b_2, \cdots, b_m$ condiciones o restricciones. Para evitar confusiones, en las anteriores secciones nos referíamos a $x$ como los datos de entrada de los modelos pero ahora estamos considerando a $x$ como la variable a optimizar. Es decir, ahora estamos pensando en $x$ como elemento del espacio de parámetros.

Se dice que un vector $x^*$ es solución del problema de optimización o vector óptimo si tiene el menor (o mayor) objetivo de entre todos los vectores que cumplen las restricciones antes mencionadas. Esto es, $\forall z$ tal que $f_1(z) \leq b_1, \cdots, f_m(z) \leq b_m$ tenemos $f_0(x^*) \leq f_0(z)$.

Existen distintas categorías dentro de los problemas de optimización según las características de las funciones que aparecen en ellos. Por ejemplo, si las funciones $f, f_1, \cdots, f_m$ son lineales, es decir,

$$f_i(\alpha x + \beta y) = \alpha f_i(x) + \beta f_i(y),$$

para todo $x, y \in \mathbb{R}^n$, y todo $\alpha, \beta \in \mathbb{R}$, donde $i = 0, \cdots, m$, el problema se denomina de programación lineal. Si una o más funciones no son lineales, el problema se denomina de programación no lineal.

Otro ejemplo a considerar, son los problemas de optimización convexa, que suceden cuando tanto la función a minimizar y las funciones restrictivas son convexas. Una función $f$ es convexa cuando cumple:

$$f(\alpha x + \beta y) \leq \alpha f(x) + \beta f(y), \quad \forall x, y \in \mathbb{R}^n, \quad \forall \alpha, \beta \in \mathbb{R}, \alpha + \beta = 1, \ \alpha, \beta \geq 0.$$

Como es fácilmente deducible, toda función lineal es convexa, por lo tanto, la programación lineal es un caso concreto de la programación convexa.

Por otro lado, la optimización estocástica es un tipo de optimización en la cual, la función objetivo i/o algunas (o todas) las funciones de restricción tienen presente la aleatoriedad. Es decir, se usan y generan variables aleatorias dentro de los métodos de optimización.

Para entrenar nuestros modelos de aprendizaje automático y particularmente de aprendizaje profundo, debemos minimizar la función de coste asociada al modelo. Hay una gran variedad de funciones de coste elegibles para los modelos, y muchas veces, son funciones bastante complicadas de tratar, y por ende, de minimizar. La optimización matemática nos brinda las herramientas para tratar de resolver el problema de minimzar la función de coste.

Introduciremos brevemente los tipos más generales de optimización, cómo resolver los problemas que plantean éstos y cómo los utilizamos hoy en día.

En los algoritmos de aprendizaje profundo, la cantidad de datos suele ser masiva. Por este motivo, se utiliza la optimización estocástica que nos permite reducir la dimensión de los datos asegurando que el algoritmo de optimización sea lo suficientemente efectivo. Como este trabajo está centrado en Deep Learning, hablaremos extensamente del método estocástico más utilizado para tratar funciones de coste asociadas a redes neuronales, el método con subgradientes estocásticos y algunas variantes, que sirven principalmente para acelerar la convergencia.

### 3.1.1. Aplicaciones

Un problema de optimización es, básicamente, una abstracción matemática a una situación cotidiana, como es escoger de entre muchos candidatos, el mejor. Volviendo a las matemáticas, esto es, hacer la mejor elección posible de un vector de $\mathbb{R}^n$ de entre un conjunto de otros vectores (que también cumplan ciertas restricciones). La variable $x$ representa la elección, $f_i(x) \leq b_i$ las restricciones, y $f_0(x)$ el coste que tiene hacer la elección $x$. Una solución del problema de optimización es entonces, la elección que tenga el mínimo coste (o el máximo beneficio) de entre todos los demás candidatos.

La optimización tiene una gran variedad de usos en el mundo en el que vivimos. Determinar la mejor inversión en un activo; escoger la colocación óptima en componentes electrónicos; infraestructuras que sean óptimas en cuanto a economía y espacio; tareas de ingeniería de todo tipo; mejoras en procesos químicos; determinar a partir de unos datos el mejor modelo que se ajuste a ellos antes y después de ir añadiendo información; son algunos de los ejemplos de aplicaciones de la optimización.

La optimización está muy presente en nuestro día a día, por el hecho que siempre nos interesa hacer las cosas lo mejor posible en todos los aspectos.

### 3.1.2. Soluciones frente a la optimización

Como hemos dicho, una solución a un problema de optimización es aquella que minimiza la función objetivo y cumple las restricciones del problema. Para lograr este objetivo, durante las últimas décadas, se ha puesto un gran ímpetu en el desarrollo de una gran variedad de algoritmos, que nos permiten encontrar soluciones (con cierta precisión y error) según a que tipo de problema nos estemos enfrentando.

La efectividad de los algoritmos, es decir, su capacidad de resolver el problema de optimización, varia mucho de un algoritmo a otro. Estos modelos dependen de muchos factores, como su objetivo, sus funciones de restricción, la cantidad de variables, si tienen una estructura especial, etc. Aunque en un problema de optimización, las funciones de restricción o la función objetivo sean relativamente sencillas, como por ejemplo, polinomios que son $\mathbb{C}^\infty$, la dificultad de resolución puede ser abismal. Puede ocurrir que el cálculo sea computacionalmente intratable o simplemente, que no se encuentre solución. Esto sucede en muchas ocasiones.

No obstante, tenemos algunas excepciones importantes. Hay tipos de problemas de optimización que, pese a tener enormes cantidades de variables, o restricciones, existen algoritmos efectivos que son capaces de resolverlos.
La programación lineal es un caso muy conocido de esta excepción, así como los problemas

de mínimos cuadrados. Por otro lado, no es tan sabido que la programación convexa también forma parte de la excepción. Efectivamente, hay algoritmos que son capaces de resolver problemas de optimización con funciones convexas.

## 3.2. Optimización de mínimos cuadrados

Un problema de mínimos cuadrados es un problema de optimización matemática sin restricciones (i.e $m = 0$) y que tiene la forma

$$\text{minimizar } f_0(x) = ||Ax - b||_2^2 = \sum_{i=1}^{k} (a_i^T x - b_i)^2$$

donde $A \in \mathbb{R}^{k \times n}$ con $k \geq n$, $a_i^T$ son las filas de $A$, y el vector $x \in \mathbb{R}^n$ es la variable de optimización.

La solución $x^*$ a este tipo de problemas pasa por resolver el sistema de ecuaciones lineales sobredeterminado (más ecuaciones que incógnitas),

$$(A^T A)x^* = A^T b,$$

por lo tanto, tenemos la solución $x^* = (A^T A)^{-1} A^T b$, dado que $A^T A$ es cuadrada, simétrica, definida positiva (o semidefinida positiva si tiene columnas linealmente dependientes) y por tanto, invertible.

Veamos los cálculos:

$$min_x \, f_0(x) \Rightarrow \frac{\partial f_0(x)}{\partial x} = 0 \Leftrightarrow \frac{\partial ||Ax - b||_2^2}{\partial x} = \frac{\partial (Ax - b)^T (Ax - b)}{\partial x} = \frac{\partial (b^T b - 2x^T A^T b + x^T A^T Ax)}{\partial x} = 0.$$

Derivando obtenemos:

$$\frac{\partial f_0(x)}{\partial x} = 0 \Leftrightarrow -2A^T b + 2(A^T A)x = 0 \Leftrightarrow (A^T A)x = A^T b.$$

Existen una gran cantidad de algoritmos fiables y preciosos que resuelven el problema anterior. El número de operaciones necesarias para resolver un problema de mínimos cuadrados es proporcional a $n^2 k$ con constante conocida. Esto es equivalente a unos pocos segundos para cualquier ordenador de escritorio. Evidentemente, cuanto mayor sea la potencia de cálculo, menor será el tiempo que necesitaremos para realizar el cálculo. Si además, la matriz tiene características que nos permitan reducir operaciones, como por ejemplo, que sea una matriz hueca (con menos de $kn$ elementos no nulos) esto permite que, pese a ser un matriz masiva, de decenas o cientos de miles de elementos, el tiempo de cálculo sea muy pequeño. Para problemas aun más extensos, de millones de variables por ejemplo, el tema se complica y los problemas pasan a ser un desafío más complicado para la tecnología actual. No obstante, se puede afirmar que en la gran mayoría de los casos, los algoritmos que resuelven problemas de optimización de mínimos cuadrados, son muy efectivos y muy fiables.

### 3.2.1. Usos y variantes de la optimización de mínimos cuadrados

El problema de mínimos cuadrados es la base del análisis de la regresión, del control óptimo y de la estimación paramétrica en procesos de ajuste de datos. Reconocer

que un problema de optimización como de mínimos cuadrados es relativamente sencillo. Simplemente, hemos de comprobar que la función objetivo puede expresarse como una función cuadrática y probar que la forma cuadrática $A^T A$ asociada es semidefinida positiva. Dentro del problema de mínimos cuadrados tenemos algunas variantes:

- *Mínimos cuadrados ponderados*:
  La función de coste tiene unos pesos asociados $w_i, \; i \in \{1, 2, \cdots, k\}$:

  $$f_0(x) = \sum_{i=1}^{k} w_i (a_i^T x - b_i)^2$$

  Estos pesos nos sirven para controlar los tamaños de los elementos $a_i^T x - b_i$, o simplemente para influenciar el resultado que queremos obtener. En optimización estocástica, los pesos nos permiten determinar las probabilidades asociadas a cada elemento.

- *Mínimos cuadrados regularizados*:
  Ya hemos hablado anteriormente sobre la regularización. En este caso, las penalizaciones que ponemos son de esta forma:

  $$f_0(x) = \sum_{i=1}^{k} (a_i^T x - b_i)^2 + \rho \sum_{i=1}^{n} x_i^2, \quad p > 0$$

  Los términos extra añadidos, penalizan los valores más grandes de $x$, por si fuera necesario realizare un ajuste al problema base. El parámetro $\rho$ es a elección nuestra y la idea es encontrar el equilibrio para no hacer demasiado grande a $\rho \sum_{i=1}^{n} x_i^2$ mientras hacemos pequeña a la función inicial $\sum_{i=1}^{k} (a_i^T x - b_i)^2$. En optimización estocástica, la regularización puede verse como una estimación estadística, teniendo en cuenta que el vector $x$ a estimar, venga previamente con una distribución asociada.

Si hacemos un poco la reflexión, vemos que las diferentes técnicas que se utilizan en optimización de mínimos cuadrados, pueden aplicarse en modelos de aprendizaje automático si la función de coste es parecida al error cuadrático medio.

## 3.3. Programación lineal

Como ya hemos mencionado, un problema de optimización se denomina de programación lineal cuando la función objetivo y las restricciones son funciones lineales. En este caso, a diferencia de los mínimos cuadrados, no existe una fórmula analítica para la solución. Por otra parte, tenemos una variedad de algoritmos muy efectivos para resolverlos. Por ejemplo el método simple de Dantzig [31] o los métodos de punto interior [27].

Otra diferencia respecto a los métodos de mínimos cuadrados, es que no se puede dar el nombre exacto de operaciones aritméticas requeridas para resolver un problema de programación lineal. No obstante, si que podemos establecer cotas bastante precisas sobre el número de operaciones necesarias, dada una cierta precisión, mediante un método de punto interior.
La complejidad en la práctica es del orden de $n^2 m$ (suponiendo $m \geq n$) pero con constante

$m$ no conocida sino aproximada. A grandes rasgos, estos algoritmos de resolución de la programación lineal son bastante fiables, pero no tanto como los de mínimos cuadrados.

Podemos resolver fácilmente un problema de programación lineal, de miles de variables y miles de restricciones, en un ordenador de sobremesa normal y corriente, en cuestión de varios segundos. Como pasaba antes, si se cumplen ciertas características (por ejemplo si la matriz asociada es hueca), podemos aumentar a decenas o cientos de miles de variables y restricciones sin aumentar casi nada el tiempo de cálculo. Para problemas más extensos, de millones de variables, vuelve a ser un reto para nuestra tecnología actual resolver un problema de programación lineal.

Generalmente, podemos afirmar que los métodos de resolución de problemas de programación lineal existentes, resuelven la gran mayoría de problemas de una manera muy fiable y muy precisa.

### 3.3.1. Usos de la programación lineal

La programación lineal se usa básicamente para resolver problemas del estilo

$$\begin{aligned} &\text{minimizar } c^T x \\ &\text{con retricciones } a_i^T x \leq b_i \quad i = 1, \cdots, m. \end{aligned}$$

siendo $c, a_1, \cdots, a_m \in \mathbb{R}^n$ y $b_1, \cdots, m \in \mathbb{R}$ parámetros fijos del problema y $x$ el vector de optimización. En otros casos, el problema original no se presenta de la forma de una programación lineal, pero podemos transformarlo en un equivalente a ésta y, por lo tanto, resolverlo con los algoritmos existentes.

Vamos a ver un ejemplo sencillo: el problema de aproximación de Chebyshev. Este problema consiste en minimizar

$$\max|a_i^T x - b_i|$$

donde $x \in \mathbb{R}^n$ es la variable de optimización y $a_1, \cdots, a_k \in \mathbb{R}^n$, $b_1, \cdots, b_k \in \mathbb{R}$ son parámetros fijados por el problema.

Este problema tiene cierto parecido con los mínimos cuadrados ya que ambos tratan de medir los términos $a_i^T x - b_i$. En mínimos cuadrados, lo hacemos con la suma de los cuadrados y en este caso, con el máximo de los valores absolutos. Otra diferencia notable, es que la función objetivo, en este caso no es diferenciable, cosa que en mínimos cuadrados sí lo es.

El problema de aproximación de Chebyshev se puede transformar en un problema de programación lineal de la forma

$$\begin{aligned} &\text{minimizar } t \\ &\text{con retricciones } a_i^T x - t \leq b_i \quad i = 1, \cdots, m. \\ &\qquad - a_i^T x - t \leq -b_i \quad i = 1, \cdots, m, \end{aligned}$$

con variables $x \in \mathbb{R}^n$ y $t \in \mathbb{R}$. Dado que todos los problemas de programación lineal se pueden resolver (excepto si tienen muchísimas variables y restricciones), el problema de aproximación Chebyshev también se puede. Como curiosidad, a día de hoy existen programas automáticos que son capaces de detectar problemas de optimización que pueden llegar a ser transformados a problemas de programación lineal.

## 3.4. Optimización convexa

Un problema de optimización convexa tiene la forma

$$\begin{gathered}\text{minimizar } f_0(x)\\ \text{con retricciones } f_i(x) \leq b_i \quad i = 1, \cdots, m.\end{gathered}$$

donde las funciones $f_0, \cdots, f_m : \mathbb{R}^n \to \mathbb{R}$ son convexas. Es decir, se cumple que para todo $f_i,\ i = 0, \cdots, m$:

$$f_i(\alpha x + \beta y) \leq \alpha f_i(x) + \beta f_i(y), \quad \forall x, y \in \mathbb{R}^n, \quad \forall \alpha, \beta \in \mathbb{R},\ \alpha + \beta = 1,\ \alpha, \beta \geq 0.$$

Hay que destacar, que tanto la programación lineal como la optimización de mínimos cuadrados, son casos particular de la optimización convexa.

No existe ninguna fórmula analítica para resolver un problema de optimización convexa. No obstante, tal y como pasaba con la programación lineal, hay métodos muy efectivos capaces de resolverlos.

Los métodos de punto interior [27], actúan muy bien en la práctica y, en algunos casos, se puede demostrar que resuelven el problema con una precisión específica, y con un número de operaciones no mayor a un polinomio de las dimensiones del problema. En la gran mayoría de los casos, los métodos de punto interior resuelven el problema con entre 10 y 100 iteraciones. Sin tener en cuenta la estrucura del problema, el número de operaciones necesario en cada paso es

$$\max\{n^2 m, n^3, C\}$$

donde $C$ es el coste de calcular la primera y segunda derivada de las funciones $f_0, \cdots, f_m$.

Como sucedía con la progrmación lineal y los mínimos cuadrados, los métodos de resolución de problemas de optimización convexa pueden resolver rápidamente problemas con cientos de variables y miles de restricciones. Además, si la estructura del problema es favorable, por ejemplo si la matriz asociada es hueca, los métodos pueden resolver problemas de varios miles de variables y restricciones.

### 3.4.1. Usos de la optimización convexa

Si somos capaces de formular el problema que tenemos delante en forma de problema de optimización convexa, prácticamente ya hemos acabado. Como hemos dicho antes, existen métodos muy efectivos delante de la optimización convexa.

Ahora bien, la gran dificultad es ser capaz de interpretar y reformular un problema que pueda ser de optimización convexa. Existen trucos para reconocer este tipo de problemas, pero generalmente es un desafío reconocer y reformular el problema.

Para esto, se está realizando hoy en día, una fuerte investigación para encontrar formas de identificar problemas que puedan ser de optimización convexa, y cómo reformularlos para que los métodos existentes puedan resolverlos.

Este tema podría dar para un libro entero, pero nuestro objetivo no es hablar extensamente de este tema, sino tener presente que cuando tenemos delante una función de coste asociada a un algoritmo de aprendizaje automático, tenemos muchas herramientas para tratar el problema de minimizarla.

## 3.5. Optimización estocástica

En lo que nos atañe, los algoritmos de aprendizaje automático y particularmente, las redes neuronales profundas, necesitan de las herramientas de la optimización para minimizar la función de coste. En el caso de redes neuronales, solemos tener delante problemas donde las funciones de activación no son lineales.

Por otra parte, como ya hemos visto en el apartado de 2.8.3, la cantidad de cálculos necesarios aun teniendo relativamente pocas capas ocultas, es masivo. Ahora imaginemos que además de tener todos esos cálculos, queremos entrenar a la red con una cantidad muy grande de ejemplos. El coste computacional sería enorme, solamente tratable por ordenadores muy potentes, y a veces ni eso.

A día de hoy ya hay empresas, que alquilan sus servidores y ordenadores, para que diferentes algoritmos de aprendizaje automático puedan ejecutarse. Evidentemente, Big Data y Machine Learning van de la mano. Es lógico que con una gran cantidad de datos, los modelos y las predicciones que nos brindan éstos, sean más precisas. El problema principal es a la hora de tratar con estos datos.

La optimización estocástica, nos permite reducir el coste computacional de los modelos, escogiendo muestras aleatorias de ejemplos de entrenamientos y aproximando gradientes, en vez de realizar el cálculo directo. A grandes rasgos, la optimización estocástica se define como la rama de la optimización que genera y trata con variables aleatorias en sus métodos.

A continuación, veremos el método de optimización estocástica más utilizado en aprendizaje automático, el método de subgradientes estocásticos. Este método es una variante del método de descenso del gradiente [20]. Las diferencias principales son, que en vez de ver el caso general, cuando se calcula todo el gradiente de la función objetivo, veremos que utilizando una pequeña muestra aleatoria de ejemplos de entrenamiento, y utilizando subgradientes con ruido, en vez de gradientes, el método funciona y reduce el coste computacional.

Básicamente, el método de subgradientes estocásticos nos permite encontrar un mínimo de la función de coste, pese a que pudiera haber funciones no diferenciables y escogiendo muestras aleatorias de ejemplos de entrenamiento, en vez de todo el conjunto de datos. Esto reduce el coste computacional del algortimo en gran medida.

# 4. Stochastic subgradient methods

## 4.1. Subgradientes

En esta sección, se introducirán las nociones básicas de los subgradientes, así como sus principales propiedades y formas de cálculo. La idea es introducir un poco de teoría básica antes de hablar sobre los métodos con subgradientes.

**Definición 4.1.** *Un vector $g \in \mathbb{R}^n$ es un subgradiente de $f : \mathbb{R}^n \longrightarrow \mathbb{R}$ en $x \in Dom(f)$ si para todo $z \in Dom(f)$:*

$$f(z) \geq f(x) + g^T(z - x).$$

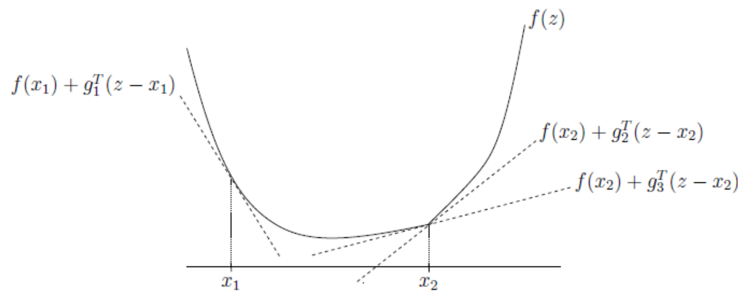


Figura 6: En $x_1$, la función convexa $f$ es diferenciable y $g_1$ (que es la derivada de $f$ en $x_1$) es el único subradiente en $x_1$. En cambio, en el punto $x_2$, $f$ no es diferenciable y tiene varios subgradientes. Se muestran dos de ellos: $g_2$ y $g_3$. Fuente: Stanford University.

**Definición 4.2.** *Una función $f$ es subdiferenciable en $x$ si existe algún subgradiente de $f$ en $x$. El conjunto de todos los subgradientes de $f$ en $x$ se denomina subdiferencial de $f$ en $x$ y se denota como $\partial f(x)$. Si una función $f$ es subdiferenciable en todo su dominio $Dom(f)$, $f$ es una función subdiferenciable.*

**Proposición 4.3.** *El subdiferencial $\partial f(x)$ es siempre un conjunto convexo cerrado, incluso siendo $f$ no convexa.*

*Demostración.* Esto se deduce del hecho de que $\partial f(x)$ se puede expresar como la intersección infinita de subespacios cerrados :

$$\partial f(x) = \bigcap_{z \in Dom(f)} \{g | f(z) \geq f(x) + g^T(z - x)\}$$

□

**Proposición 4.4.** *Además, si $f$ es continua en $x$, entonces $\partial f(x)$ está acotado.*

*Demostración.* Sea $\epsilon > 0$ tal que:

$$-\infty < \underline{f} \leq f(y) \leq \overline{f} < \infty \qquad \forall y \in \mathbb{R}^n \quad t.q \quad ||y - x||_2 < \epsilon.$$

Ahora si $\partial f(x)$ no está acotado, entonces existe una sucesión $g_n \in \partial f(x)$ con $||g_n||_2 \to \infty$. Tomando $y_n = x + \frac{\epsilon g_n}{||g_n||_2}$, tenemos $f(y_n) \geq f(x) + g_n^T(y_n - x) = f(x) + e||g_n||_2 \to \infty$ que contradice el hecho de que $f(y_n)$ es acotada. Por lo tanto, hemos demostrado que si $f$ es continua en $x$, $\partial f(x)$ está acotado. □

### 4.1.1. Existencia

**Proposición 4.5.** *Si $f$ es convexa y $x \in int(Dom\,(f))$, entonces $\partial f(x)$ es no vacío y acotado.*

*Demostración.* Para demostrar que $\partial f(x) \neq \emptyset$ utilizaremos el teorema del hiperplano [28] de soporte para el conjunto convexo (es convexo por ser $f$ convexa) llamado epígrafo:

$$epi\, f = \{(x, y) : \; x \in \mathbb{R}^n, y \in \mathbb{R}, f(x) \leq y\} \subseteq \mathbb{R}^{n+1}.$$

El teorema se aplica en el punto de frontera $(x, f(x))$ para probar la existencia de $a \in \mathbb{R}^n$ y $b \in \mathbb{R}$ no nulos, tales que:

$$\begin{pmatrix} a \\ b \end{pmatrix}^T \left( \begin{pmatrix} z \\ t \end{pmatrix} - \begin{pmatrix} x \\ f(x) \end{pmatrix} \right) = a^T(z - x) + b(t - f(x)) \leq 0$$

para todo $(z, t) \in epi\, f$. Cosa que implica que $b \leq 0$ y que

$$a^T(z - x) + b(f(z) - f(x)) \leq 0$$

para todo $z$. Si $b \neq 0$, podemos dividir $b$ para obtener que

$$f(z) \geq f(x) - (a/b)^T(z - x),$$

que demuestra que $-a/b \in \partial f(x)$. Que $b \neq 0$ significa que el hiperplano de soporte no puede ser vertical. Ahora bien, si $b = 0$ entonces $a^T(z - x) \leq 0$, $\forall z \in Dom\,(f)$, cosa que contradice el hecho que $x \in int(Dom\,(f))$. Por lo tanto, concluimos que $b \neq 0$ y que $\partial f(x) \neq \emptyset$. □

Existen algunas funciones convexas que no tienen subgradientes en algunos puntos, no obstante, en este trabajo supondremos que todas las funciones convexas que tratamos, son subdiferenciables en todo su dominio.

### 4.1.2. Unicidad

Si $f$ es convexa y diferenciable en $x$, entonces su gradiente en $x$ es un subgradiente y $\partial f(x) = \{\nabla f(x)\}$. Esto es un si solo si, es decir, también se cumple que si $f$ es convexa y $\partial f(x) = \{g\}$, entonces $f$ es diferenciable en $x$ y $g = \nabla f(x)$.
No obstante, hemos visto que un subgradiente puede existir pese a que $f$ no sea diferenciable en todo su dominio. En este caso, el subgradiente en $x$ puede no ser único. En la figura 6, tenemos un ejemplo de que sucede con el subdiferencial, en puntos donde $f$ es diferenciable y no diferenciable respectivamente.

### 4.1.3. Algunas formas de cálculo de subgradientes

En esta sección, veremos brevemente algunas propiedades de cálculo para subgradientes de funciones convexas. Cabe destacar, que existen dos tipos de cálculo de subgradientes. Si queremos calcular el subdiferencial $\partial f(x)$ completamente, el cálculo se denomina fuerte. En cambio, si solo es necesario el cálculo de un subgradiente (independientemente de que haya más) el cálculo se denomina débil. En la mayoría de aplicaciones, con el cálculo débil ya es suficiente, dado que solo necesitamos un subgradiente para la gran mayoría de procesos donde éstos aparecen.

- *Escalado no negativo*:
  Para $\alpha \geq 0$ , $\partial f(\alpha x)(x) = \alpha \partial f(x)$.

- *Sumas e Integrales* :
  Suponemos que $f = f_1 + f_2 + \cdots + f_n$, donde $f_1, \cdots, f_m$ son funciones convexas. Entonces:

  $$\partial f(x) = \partial f_1(x) + \cdots + \partial f_n(x).$$

  Esto se extiende a sumas infinitas, integrales, y esperanzas matemáticas (siempre que existan).

- *Transformaciones afines*:
  Sea $f$ convexa, y $h(x) = f(Ax + b)$. Entonces $\partial h(x) = A^T \partial f(Ax + b)$.

- *Máximos*:
  Sea $f$ el máximo de una serie de funciones, $f_1, f_2, \cdots, f_m$, es decir:

  $$f(x) = max\{f_i(x)\}_{i=1,\cdots,m},$$

  donde $f_i$ son funciones subdiferenciables. Veamos como hallar un subgradiente de $f$ en $x$. Sea $j$ cualquier índice tal que $f_j(x) = f(x)$ y $g \in \partial f_j(x)$. Entonces $g \in \partial f(x)$. En otras palabras, para encontrar un subgradiente de un máximo de funciones en el punto $x$, basta con hallar una de las funciones que alcance el máximo en el punto $x$ y escoger uno de sus subgradientes. Esto se debe a

  $$f(z) \geq f_j(z) \geq f_j(x) + g^T(z - x) = f(x) + g^T(y - x).$$

  Generalmente tenemos

  $$\partial f(x) = \mathbf{Co}(\bigcup_{i=1,\cdots,m} \{\partial f_i(x) \mid f_i(x) = f(x)\}),$$

  donde **Co** es la envolvente convexa. Se conoce como envolvente convexa [29] de un conjunto $C$, a la intersección de todos los conjuntos convexos que contienen a C. En nuestro caso, podemos decir que el subdiferencial de un máximo de funciones en el punto $x$, es la envolvente convexa de la unión de subdiferenciales de funciones $f_i$ que cumplen el requisito $f_i(x) = f(x)$.

- *Supremos*
  Sea $f$ el sup

  $$f(x) = \underset{\alpha \in \mathcal{A}}{sup}\, f_\alpha(x),$$

  donde $f_\alpha$ son funciones subdiferenciables. Solo veremos el cálculo débil.
  Suponiendo que el supremo exista, procediendo como en el caso anterior, tenemos que si $\beta \in \mathcal{A}$ es un índice tal que $f_\beta(x) = f(x)$ y $h \in \partial f_\beta(x)$, obtenemos que $h \in \partial f(x)$. Si el supremo no existe, puede ser que la función no sea subdiferenciable en $x$, dependiendo del índice. No obstante, si asumimos que $\mathcal{A}$ es compacto y que la función $\alpha \to f_\alpha(x)$ es semi-continua superiormente para cada $x$, entonces:

  $$\partial f(x) = \mathbf{Co}(\bigcup_{\alpha \in \mathcal{A}} \{\partial f_\alpha(x) \mid f_\alpha(x) = f(x)\}).$$

- *Minimización en multivariables*

Supongamos ahora que nuestra función $f$ es de la forma

$$f(x) = \inf_y F(x, y),$$

donde F(x,y) es subdiferenciable y convexa en $(x, y) \in \mathbb{R}^n \times \mathbb{R}^m$. Supongamos también, que el ínfimo sobre $y$ se alcanza en el subconjunto $Y_x \subset \mathbb{R}^m$, donde $Y_x$ es no vacío. Entonces, $F(x, y) = f(x)$ para $y \in Y_x$. Por definición, $g \in \mathbb{R}^n$ es un subgradiente de $f$ en $x$ si y solo si

$$f(x') \geq f(x) + g^T(x' - x) = F(x, y) + g^T(x' - x)$$

para todo $x' \in \mathbb{R}^n$ y cualquier $y \in Y_x$. Equivalentemente

$$F(x', y') \geq F(x, y) + g^T(x' - x) = F(x, y) + \begin{pmatrix} g \\ 0 \end{pmatrix}^T \left( \begin{pmatrix} x' \\ y' \end{pmatrix} - \begin{pmatrix} x \\ y \end{pmatrix} \right)$$

para todo $(x', y') \in \mathbb{R}^n \times \mathbb{R}^m$ y para $x, y \in Y_x$. En particular, hemos visto que:

$$\partial f(x) = \{g \in \mathbb{R}^n |\ (g, 0) \in \partial F(x, y) \text{ para algún } y \in Y_x\}.$$

Esto es, existe un $g \in \mathbb{R}^n$ tal que $(g, 0) \in F(x, y)$ para algún $y \in Y_x$, siendo $g$ un subgradiente de $f$ en $x$ (siempre y cuando el ínfimo exista y $x \in int\, Dom\, (f)$).

## 4.2. Función de valor óptimo en optimización convexa

Sea $f : \mathbb{R}^m \times \mathbb{R}^p \longrightarrow \mathbb{R}$ la función de valor óptimo de un problema de optimización convexa, con $z \in \mathbb{R}^n$ como variable de optimización,

$$\begin{aligned} \text{minimizar } & f_0(z) \\ \text{con retricciones } & f_i(z) \leq x_i, \quad i = 1, \cdots, m \\ & Az = y \end{aligned}$$

En otras palabras, $f(x, y) = inf_z\, F(x, y, z)$ donde

$$F(x, y, z) \begin{cases} f_0(x), & \text{si} \quad f_i \leq x_i,\ i = 1, \cdots, m,\ Az = y \\ +\infty & \text{en otro caso} \end{cases}$$

que es convexa en $(x, y, z)$.
Supongamos que queremos subdiferenciar $f$ en un punto $(\hat{x}, \hat{y})$. Podemos expresar el problema de optimización como

$$\begin{aligned} \text{minimizar } & g(\lambda) - x^T\lambda - y^T\sigma \\ \text{con retricciones } & \lambda \geq 0 \end{aligned}$$

donde

$$g(\lambda) = \inf_z \left( f_0(z) + \sum_{i=1}^m \lambda_i f_i(z) + \sigma^T Az \right).$$

Suponemos que la dualidad fuerte [30] se mantiene para el primer y último problema de optimización en $(x, y) = (\hat{x}, \hat{y})$ y que el valor dual óptimo se alcanza en $\lambda^*, \sigma^*$. Entonces, por las inecuaciones de pertubación global tenemos:

$$f(x, y) \geq f(\hat{x}, \hat{y}) - (\lambda^*)^T(x - \hat{x}) - (\sigma^*)^T(y - \hat{y}).$$

En otras palabras, obtenemos un subgradiente:

$$-(\lambda^*, \sigma^*) \in \partial f(\hat{x}, \hat{y}).$$

## 4.3. Métodos con subgradientes

### 4.3.1. Introducción

Los métodos con subgradientes son una serie de algoritmos para minimizar principalmente problemas de optimización convexa donde las funciones son no diferenciables. Como es lógico, su estructura es parecida a un método de gradiente ordinario.

Recordemos como son los métodos de gradientes. Un método con gradientes es un algoritmo que resuelve un problema de optimización, obviamente diferenciable, utilizando el gradiente de la función en cada punto. Es decir, resuelve un problema del tipo

$$\min_{x \in \mathbb{R}^n} f(x)$$

calculando directamente $\nabla f(x)$ y iterando de alguna determinada manera. El ejemplo más utilizado se conoce como *método de descenso del gradiente*. La idea detrás de este algoritmo, es que se ha observado que para una función multivariable y diferenciable $F(x)$ en un entorno de un punto $x_0$, ésta decrece lo más rápido posible si lo hace desde $a$ con dirección $-\nabla F(a)$. Es decir, si

$$a_{n+1} = a_n - \gamma \nabla F(a_n)$$

con $\gamma \in \mathbb{R}^+$ suficientemente pequeño, entonces $F(a_n) \geq F(a_{n+1})$.

Extrapolando, el método consiste en, a partir de una estimación inicial $x_0$ para un mínimo local de $F(x)$, se considera la iteración $x_{n+1} = x_n - \gamma_n \nabla F(x_n)$. Tenemos entonces una sucesión monótona decreciente $F(x_0) \geq F(x_1) \geq \cdots$, que puede converger a un mínimo local. Cabe destacar, que $\gamma_n$ es variable en cada iteración y se le conoce como *step size*, o si aplicamos el algortimo a Machine Learning, como *learning rate*.

Bajo ciertas suposiciones sobre $F(x)$, como por ejemplo, $F$ convexa y $\nabla F(x)$ Lipschitz, y eligiendo una $\gamma_n$ adecuada, la convergencia hacia un mínimo local puede estar garantizada. Teniendo en cuenta que si $F$ es convexa, sus mínimos locales son también mínimos globales, el método del gradiente descendiente, en este caso, tendería hacia una solución global.

Ahora bien, hablemos ahora de las diferencias entre los métodos de subgradientes y los de gradientes. La más obvia de todas, es que los métodos con subgradientes permiten trabajar directamente con funciones no diferenciables. El cálculo de cada valor $\gamma_i$ también es diferente en los dos tipos de métodos. En los métodos con gradientes, normalmente para determinar cada $\gamma_i$ se usa lo que llamamos búsqueda en línea [26].
No obstante, esto no sucede en los métodos de subgradientes. En la mayoría de los casos, los valores $\gamma_i$ vienen fijados antes de cada iteración. Además, los métodos de subgradientes no son de descenso, como si lo es el método de descenso del gradiente. El valor de la función en un método con subgradiente puede, y suele, crecer en alguna iteración.

Ambos métodos, tanto el de subgradiente como el de descenso del gradiente son los principales algoritmos de minimización de funciones de coste en Machine Learning. En cuanto a redes neuronales, se combina el backpropagation que hemos visto anteriormente, con cualquiera de los dos métodos, según si la función de coste es diferenciable o no.

### 4.3.2. Método con subgradiente negativo básico

El caso más básico es un problema sin restricciones, donde el objetivo es minimizar una función convexa $f : \mathbb{R}^n \longrightarrow \mathbb{R}$ que tiene dominio $\mathbb{R}^n$. Como uno se puede imaginar, el método consiste en

$$x^{(k+1)} = x^{(k)} - \alpha_k g^{(k)}$$

donde $g^{(k)}$ es cualquier subgradiente de $f$ en $x^{(k)}$ y $\alpha_k > 0$ es el $k$-ésimo step size. Si $f$ es diferenciable en $x^{(k)}$, entonces el único valor para $g^{(k)}$ es $g^{(k)} = \nabla f(x^{(k)})$, y, en este caso, el método del subradiente es el mismo que el del gradiente, excepto en la elección de $\alpha_k$. Como hemos mencionado, estos algoritmos no tienen porque ser de descenso. Esto se debe a que $-g(k)$ puede no ser una dirección descendiente, es decir, que suceda $f'(x; -g^{(k)}) > 0$ que implicaría que $f(x^{(k+1)}) > f(x^{(k)})$. Incluso yendo $-g(k)$ en dirección decreciente en $x^{(k)}$, puede suceder que $f(x^{(k+1)}) > f(x^{(k)})$ debido al step size $\alpha_k$. En conclusión, que los métodos de subgradientes no tienen que porque ser decrecientes en cada iteración. Debido a este hecho, tenemos que ir calculando en cada iteración, el mejor punto posible, es decir, el punto con el valor de función más pequeño. Para esto, definimos en cada iteración

$$f^{(k)}_{\text{óptimo}} = \min\{f^{(k-1)}_{\text{óptimo}}, f(x^{(k)})\}$$

y fijamos $i^{(k)}_{\text{óptimo}} = k$ si $f^{(k)}_{\text{óptimo}} = f(x^{(k)})$, es decir, si $x^{(k)}$ es el punto óptimo. Esto no es necesario en el método con gradiente ya que, al ser de descenso, el punto actual es siempre el punto óptimo de avance. Ahora, iterando tenemos

$$f^{(k)}_{\text{óptimo}} = \min\{f(x^{(1)}), \cdots, f(x^{(k)})\},$$

es decir, el mejor valor objetivo en $k$ iteraciones. Como $f^{(k)}_{\text{óptimo}}$ sí que es decreciente, tiene un límite, pese a que puede ser $-\infty$.

### 4.3.3. Reglas para el paso de optimización (step size)

Como hemos mencionado antes, la elección del step size es otra diferencia notable entre un método con subgradientes y otro con gradientes. Veamos los ejemplos más básicos de elección de un step size en un método con subgradientes:

- *Step size constante.* $\alpha_k = \alpha$ donde $\alpha$ es una constante positiva independiente a $k$.
- *Step length constante.* $\alpha_k = \gamma/||g^{(k)}||_2$ donde $\gamma > 0$. Esto implica: $||x^{(k+1)} - x^{(k)}||_2 = \gamma$.
- *De cuadrado sumable pero no sumable.* En este caso, el step size cumple

  $$\alpha_k \geq 0 \qquad \sum_{k=1}^{\infty} \alpha_k^2 < \infty \qquad \sum_{k=1}^{\infty} \alpha_k = \infty.$$

  Un ejemplo de cuando sucede esto seria $\alpha_k = \frac{a}{b+k}$ con $a > 0$ y $b \geq 0$.
- *Decreciente no sumable.* El step size satisface

  $$\alpha_k \geq 0 \qquad \lim_{k\to\infty} \alpha_k = 0 \qquad \sum_{k=1}^{\infty} \alpha_k = \infty.$$

  Por ejemplo, esto sucede cuando $\alpha_k = \frac{a}{\sqrt{k}}$ con $a > 0$.

- *Step length decreciente no sumable.* Ahora $\alpha_k = \gamma_k / ||g(k)||_2$, donde

$$\gamma_k \geq 0 \qquad \lim_{k\to\infty} \gamma_k = 0 \qquad \sum_{k=1}^{\infty} \gamma_k = \infty.$$

Como hemos comentado, en los métodos de subgradientes lo más interesante es que se escoge el step size antes de iniciar el algoritmo, por lo tanto, el step size no depende de los datos que se vayan computando. Cosa que si suele suceder en los métodos con gradientes.

### 4.3.4. Convergencia de los métodos con subgradientes

Vamos a demostrar la convergencia de un típico método con subgradiente bajo ciertas suposiciones. Suponemos que existe un minimizador $x^*$ de $f$. También suponemos que la norma de los subgradientes está acotada, es decir, que existe $M$ tal que $||g^{(k)}||_2 \leq M$, $\forall k$. Esto sucede, por ejemplo cuando $f$ es Lipschitz

$$|f(u) - f(v)| \leq M||u - v||_2,$$

para todo $u, v$, porque entonces $||g||_2 \leq M$ para cualquier $g \in \partial f(x)$, y cualquier $x$.

Por otra parte, supondremos que existe un número $R$ conocido cumpliendo

$$R \geq ||x^{(1)} - x^*||_2.$$

Esta $R$ puede ser interpretada como una cota superior de la distancia entre el punto inicial y el conjunto óptimo. En los métodos con subgradientes, a diferencia de los de gradientes, para medir la convergencia no se utiliza el valor de la función en cada iteración, ya que esta puede decrecer o crecer. Es la distancia euclídea al conjunto óptimo el que utilizamos como cuantificador. Recordemos que $x^*$ es un punto que minimiza $f$, por lo tanto, $x^*$ es un punto arbitrario del conjunto óptimo. Tenemos

$$\begin{aligned} ||x^{(k+1)} - x^*||_2^2 &= ||x^{(k)} - \alpha_k g^{(k)} - x^*||_2^2 \\ &= ||x^{(k)} - x^*||_2^2 - 2\alpha_k g^{(k)T}(x^{(k)} - x^*) + \alpha_k^2 ||g^{(k)}||_2^2 \\ &\leq ||x^{(k)} - x^*||_2^2 - 2\alpha_k (f(x^{(k)}) - f^*) + \alpha_k^2 ||g^{(k)}||_2^2, \end{aligned}$$

donde $f^* = f(x^*)$. La última línea se obtiene directamente de la definición de subgradiente

$$f(x^*) \geq f(x^{(k)}) + g^{(k)T}(x^* - x^{(k)}).$$

Si se aplica la inecuación anterior reiteradamente, obtenemos

$$||x^{(k+1)} - x^*||_2^2 \leq ||x^{(1)} - x^*||_2^2 - 2\sum_{i=1}^{k} \alpha_i (f(x^{(i)}) - f^*) + \sum_{i=1}^{k} \alpha_i^2 ||g^{(i)}||_2^2.$$

Sabemos que $||x^{(k+1)} - x^*||_2^2 \geq 0$ y por hipótesis tenemos $||x^{(1)} - x^*||_2 \leq R$. Aplicando esto último obtenemos

$$2\sum_{i=1}^{k} \alpha_i (f(x^{(i)}) - f^*) \leq R^2 + \sum_{i=1}^{k} \alpha_i^2 ||g^{(i)}||_2^2.$$

Que combinado con

$$\sum_{i=1}^{k} \alpha_i (f(x^{(i)}) - f^*) \geq \left( \sum_{i=1}^{k} \alpha_i \right) \min_{i=1,\cdots,k} (f(x^{(i)}) - f^*) = \left( \sum_{i=1}^{k} \alpha_i \right) (f^{(k)}_{\text{óptimo}} - f^*),$$

nos da la inecuación

$$f^{(k)}_{\text{óptimo}} - f^* = \min_{i=1,\cdots,k} (f(x^{(i)}) - f^*) \leq \frac{R^2 + \sum_{i=1}^{k} \alpha_i^2 ||g^{(i)}||_2^2}{2 \sum_{i=1}^{k} \alpha_i}$$

Finalmente, bajo la suposición $||g^{(k)}||_2 \leq M$, tenemos

$$f^{(k)}_{\text{óptimo}} - f^* \leq \frac{R^2 + M^2 \sum_{i=1}^{k} \alpha_i^2}{2 \sum_{i=1}^{k} \alpha_i}.$$

Vamos a sacar algunas conclusiones de este resultado aplicándolo a los diferentes tipos de step size.

- *Step size constante.* Cuando tenemos $\alpha_k = \alpha$, obtenemos

$$f^{(k)}_{\text{óptimo}} - f^* \leq \frac{R^2 + M^2 \alpha^2 k}{2\alpha k}.$$

  El lado derecho de la inecuación converge a $M^2\alpha/2$ cuando $k \to \infty$. Por lo tanto, para el método de subgradiente con constante fija $\alpha$, $f^{(k)}_{\text{óptimo}}$ converge con $M^2\alpha/2$ cómo óptimo. También concluimos que $f^{(k)}_{\text{óptimo}} - f^* \leq M^2\alpha$ con como máximo $\frac{R^2}{M^2\alpha}$ pasos.

- *Step length constante.* Con $\alpha_k = \gamma_k/||g^{(k)}||_2$, obtenemos

$$f^{(k)}_{\text{óptimo}} - f^* \leq \frac{R^2 + \gamma^2 k}{2 \sum_{i=1}^{k} \alpha_i} \leq \frac{R^2 + \gamma^2 k}{2\gamma k/M},$$

  donde hemos usado que $\alpha_i \geq \gamma/M$. En este caso, el lado derecho converge a $M\gamma/2$ cuando $k \to \infty$. Por lo tanto, el método converge con $M\gamma/2$ como óptimo.

- *De cuadrado sumable pero no sumable.* En este caso teníamos

$$\alpha_k \geq 0 \qquad ||\alpha||_2^2 = \sum_{k=1}^{\infty} \alpha_k^2 < \infty \qquad \sum_{k=1}^{\infty} \alpha_k = \infty.$$

  Por lo tanto

$$f^{(k)}_{\text{óptimo}} - f^* \leq \frac{R^2 + M^2 ||\alpha||_2^2}{2 \sum_{i=1}^{k} \alpha_i},$$

  que converge a 0 cuando $k \to \infty$ ya que el numerador tiende a $R^2 + M^2||\alpha||_2^2$ pero el denominador tiende a $+\infty$. Entonces, el método converge con $f^{(k)}_{\text{óptimo}} \to f^*$.

- *Decreciente no sumable.* Recordemos las propiedades de este caso.

$$\alpha_k \geq 0 \qquad \lim_{k \to \infty} \alpha_k = 0 \qquad \sum_{k=1}^{\infty} \alpha_k = \infty.$$

Vamos a ver que si estas propiedades se cumplen entonces

$$\frac{R^2 + M^2 \sum_{i=1}^k \alpha_i^2}{2\sum_{i=1}^k \alpha_i} \longrightarrow 0$$

cosa que implica que el método de sugradiente converge. Sea $\epsilon > 0$, entonces existe un entero $N_1$ tal que $\alpha_i \leq \epsilon/M^2$, $\forall i > N_1$. También existe un entero $N_2$ cumpliendo

$$\sum_{i=1}^{N_2} \geq \frac{1}{\epsilon}\left(R^2 + M^2 \sum_{i=1}^{N_1} \alpha_i^2\right),$$

ya que $\sum_{i=1}^{\infty} \alpha_i = \infty$. Sea $N = \max\{N_1, N_2\}$. Entonces, para $k > N$ tenemos

$$\begin{aligned}
\frac{R^2 + M^2 \sum_{i=1}^k \alpha_i^2}{2\sum_{i=1}^k \alpha_i} &\leq \frac{R^2 + M^2 \sum_{i=1}^{N_1} \alpha_i^2}{2\sum_{i=1}^k \alpha_i} + \frac{M^2 \sum_{i=N_1+1}^k \alpha_i^2}{2\sum_{i=1}^{N_1} \alpha_i + 2\sum_{i=N_1+1}^k \alpha_i} \\
&\leq \frac{R^2 + M^2 \sum_{i=1}^{N_1} \alpha_i^2}{(2/\epsilon)(R^2 + M^2 \sum_{i=1}^{N_1} \alpha_i^2)} + \frac{M^2 \sum_{i=N_1+1}^k (\epsilon \alpha_i/M^2)}{2\sum_{N_1+1}^k \alpha_i} \\
&= \frac{\epsilon}{2} + \frac{\epsilon}{2} = \epsilon.
\end{aligned}$$

- *Step length decreciente no sumable.* Si $\alpha = \gamma_k/||g^{(k)}||_2$, y $\gamma_k$ es no sumable y converge a 0, tenemos que

$$f_{\text{óptimo}}^{(k)} - f^* \leq \frac{R^2 + \sum_{i=1}^k \gamma_k^2}{2\sum_{i=1}^k \alpha_i} \leq \frac{R^2 + \sum_{i=1}^k \gamma_k^2}{(2/M)\sum_{i=1}^k \gamma_i}$$

que converge a 0 cuando $k \to \infty$.

### 4.3.5. Cota óptima y criterio de parada

Una vez hablado de convergencia, surge una cuestión interesante. ¿Podemos escoger $\alpha_1, \cdots, \alpha_k$ positivos de tal manera que

$$\frac{R^2 + M^2 \sum_{i=1}^k \alpha_i^2}{2\sum_{i=1}^k \alpha_i}.$$

sea mínimo? Al ser la función anterior, simétrica y convexa de $\alpha_1, \cdots, \alpha_k$, podemos concluir que el óptimo se alcanza cuando $\alpha_1 = \cdots = \alpha_k = \alpha$. Para algún $\alpha$. Esto reduce el problema a minimizar

$$\frac{R^2 + M^2 k\alpha^2}{2k\alpha}.$$

Derivado respecto a $\alpha$ e igualando a 0, obtenemos que el mínimo se alcanza cuando $\alpha = \frac{R}{M\sqrt{k}}$. En otras palabras, la elección de $\alpha_1, \cdots, \alpha_k$ viene dada por

$$\alpha_i = \frac{R}{M\sqrt{k}},\ i = 1, \cdots, k.$$

Esto implica

$$f_{\text{óptimo}}^{(k)} - f^* \leq RM/\sqrt{k}.$$

Dicho de otra manera, sea cual sea nuestra elección del step size, la cota óptima $\frac{R^2+M^2\sum_{i=1}^{k}\alpha_i^2}{2\sum_{i=1}^{k}\alpha_i}$ tiene que ser cómo mínimo tan grande como $RG/\sqrt{k}$.

En los métodos de subgradientes, el criterio de parada no es calculado de ninguna forma en particular. Esto es debido a que cualquier cota mejor que $\frac{R^2+M^2\sum_{i=1}^{k}\alpha_i^2}{2\sum_{i=1}^{k}\alpha_i}$ y que no dependa de $M$, converge muy lentamente a 0. Por lo tanto, no existe un cálculo formal para el criterio de parada.

Generalmente, el método básico con subgradientes o gradientes, converge bastante lento. Por ese motivo, se suelen utilizar métodos de aceleración de la convergencia del algoritmo que veremos más adelante.

## 4.4. Métodos con subgradientes estocásticos

Acabamos de ver como hacer frente al problema de que nuestra función de coste asociada al algortimo de Machine Learning, sea no diferenciable en algún punto de su dominio. Ahora bien, como ya hemos mencionado, existe otro gran problema, la dimensión de los datos. En Machine Learning y, sobretodo en Deep Learning, la cantidad de parámetros y ejemplos de entrenamiento hacen que la cantidad de cálculos necesaria para calcular todos los gradientes o subgradientes sea muy grande.

Para esquivar este problema, surgieron los métodos estocásticos con subgradientes, o stochastic gradient (subgradient) methods. En estos métodos, no se calcula el gradiente o subgradiente total de la función, sino que se calcula una estimación insesgada del gradiente (o subgradiente) tomando la media de los gradientes de una muestra reducida de ejemplos, elegida al azar. Está tecnica, la de escoger una muestra aleatoria de ejemplos de entranimiento en vez de considerar todos los datos de entrada, se conoce como minibatch.

### 4.4.1. Subgradiente insesgado con ruido

Sea $f : \mathbb{R}^n \to \mathbb{R}$ una función convexa. Decimos que un vector aleatoria $g' \in \mathbb{R}^n$ es un subgradiente insesgado con ruido de $f$ en $x \in Dom(f)$ si $g = \mathbb{E}(g') \in \partial f(x)$, es decir,

$$f(z) \geq f(x) + (\mathbb{E}(g'))^T(z-x)\ \forall z.$$

Si $g'$ es un subgradiente insesgado con ruido, $g'$ se puede escribir como $g' = g + v$ donde $g$ es un subgradiente de $f$ en $x$ y $v$ tiene media cero y es lo que se denomina ruido. Si $x$ es un variable aleatoria, entonces definimos $g'$ como subgradiente con ruido de $f$ en $x$ si

$$\forall z,\ f(z) \geq f(x) + \mathbb{E}(g'|x)^T(z-x)$$

casi seguramente.

### 4.4.2. Métodos con subgradientes estocásticos básicos

El método con subgradiente estocástico es esencialmente un método con subgradientes parecido a los que hemos visto anteriormente. No obstante, hay algunos matices diferentes. Las principales diferencias son a la hora de escoger el step size, y que trabajaremos con subgradientes insesgados con ruido. Tanto los métodos con subgradientes estocásticos como los métodos con gradientes estocásticos son la quintaesencia del Deep Learning.

Consideramos el caso más básico, la optimización sin restricciones de una función convexa $f : \mathbb{R}^n \to \mathbb{R}$. El método tiene como iteración

$$x^{(k+1)} = x^{(k)} + \alpha_k g'^{(k)},$$

donde $x^{(k)}$ es la k-ésima iteración, $\alpha_k > 0$ es el step size en la k-ésima iteración, y $g'^{(k)}$ es un subgradiente insesgado con ruido de $f$ en $x^{(k)}$, es decir,

$$\mathbb{E}(g'^{(k)}|x^{(k)}) = g^{(k)} \in \partial f(x^{(k)}).$$

Como estamos en el caso de un método con subgradiente simple, podemos ir evaluando los puntos óptimos y el valor asociado a la función

$$f^{(k)}_{\text{óptimo}} = \min\{f(x^{(1)}), \cdots, f(x^{(k)})\}.$$

Evidentemente, $x^{(k)}, g'^{(k)}$ y $f^{(k)}_{\text{óptimo}}$ son procesos estocásticos.

### 4.4.3. Convergencia en métodos con subgradientes estocásticos

Veamos una demostración de convergencia del método con subgradientes estocásticos utilizando, por ejemplo, un step size de cuadrado sumable pero no sumable. Recordemos que esto quiere decir que

$$\alpha_k \geq 0 \qquad ||\alpha||_2^2 = \sum_{k=1}^{\infty} \alpha_k^2 < \infty \qquad \sum_{k=1}^{\infty} \alpha_k = \infty.$$

Suponemos que existe un $x^*$ que minimiza $f$, y una $M$ tal que $\mathbb{E}(||g^{(k)}||_2^2) \leq M^2, \forall k$. También suponemos que existe una $R$ tal que $\mathbb{E}(||x^{(1)} - x^*||_2^2) \leq R^2$. Veremos que

$$\mathbb{E}(f^{(k)}_{\text{óptimo}}) \to f^*$$

cuando $k \to \infty$. También veremos la convergencia en probabilidad: para cualquier $\epsilon > 0$,

$$\lim_{k\to\infty} \mathbb{P}(f^{(k)}_{\text{óptimo}} \geq f^* + \epsilon) = 0.$$

Para ver la convergencia casi segura, habría que recurrir a métodos más complejos que quedan fuera del objetivo de este trabajo. Tenemos entonces

$$\begin{aligned}
\mathbb{E}(||x^{(x+1)} - x^*||_2^2 \ \mid \ x^{(k)}) &= \mathbb{E}(||x^{(k)} - \alpha_k g'^{(k)} - x^*||_2^2 \ \mid \ x^{(k)}) \\
&= ||x^{(k)} - x^*||_2^2 - 2\alpha_k \mathbb{E}(g'^{(k)T}(x^{(k)} - x^*) \ \mid \ x^{(k)}) + \alpha_k^2 \mathbb{E}(||g'^{(k)}||_2^2 \ \mid \ x^{(k)}) \\
&= ||x^{(k)} - x^*||_2^2 - 2\alpha_k \mathbb{E}(g'^{(k)} \ \mid \ x^{(k)})^T (x^{(k)} - x^*) + \alpha_k^2 \mathbb{E}(||g'^{(k)}||_2^2 \ \mid \ x^{(k)}) \\
&\leq ||x^{(k)} - x^*||_2^2 - 2\alpha_k (f(x^{(k)}) - f^*) + \mathbb{E}(||g'^{(k)}||_2^2 \ \mid \ x^{(k)}),
\end{aligned}$$

donde la desigualdad se cumple casi seguramente, ya que $\mathbb{E}(g'^{(k)} \mid x^{(k)}) \in \partial f(x^{(k)})$. Aplicando esperanzas a banda y banda de la desigualdad y usando que $\mathbb{E}(||g'^{(k)}||_2^2) \leq M^2$ obtenemos

$$\mathbb{E}(||x^{(k+1)} - x^*||_2^2) \leq \mathbb{E}(||x^{(k)} - x^*||_2^2) - 2\alpha_k(\mathbb{E}(f(x^{(k)}) - f^*) + \alpha_k^2 M^2.$$

Aplicando recursivamente esta desigualdad:

$$\mathbb{E}(||x^{(k+1)} - x^*||_2^2) \leq \mathbb{E}(||x^{(1)} - x^*||_2^2) - 2\sum_{i=1}^{k} \alpha_i(\mathbb{E}(f(x^{(i)}) - f^*) + M^2 \sum_{i=1}^{k} \alpha_i^2.$$

Usando ahora que $\mathbb{E}(||x^{(1)} - x^*||_2^2) \leq R^2$, $\mathbb{E}(||x^{(k+1)} - x^*||_2^2) \geq 0$ y que $\sum_{i=1}^{k} \alpha_i^2 \leq ||\alpha||_2^2$, tenemos

$$2\sum_{i=1}^{k} \alpha_i(\mathbb{E}(f(x^{(i)})) - f^*) \leq R^2 + M^2||\alpha||_2^2.$$

Por lo tanto se cumple que

$$\underset{i=1,\cdots,k}{min}(\mathbb{E}(f(x^{(i)})) - f^*) \leq \frac{R^2 + M^2||\alpha||_2^2}{2\sum_{i=1}^{k} \alpha_i} \underset{k\to\infty}{\longrightarrow} 0,$$

ya que el step size no es sumable. Hemos probado que $\min_{i=1,\cdots,k} \mathbb{E}(f(x^{(i)})$ converge a $f^*$. Usando la desigualdad de Jensen y la concavidad de la función mínimo, se tiene

$$\mathbb{E}(f^{(k)}_{\text{óptimo}}) = \mathbb{E}\left(\min_{i=1,\cdots,k} f(x^{(i)})\right) \leq \min_{i=1,\cdots,k} \mathbb{E}(f(x^{(i)})),$$

que, junto con lo que hemos visto antes, prueba la convergencia de $\mathbb{E}(f^{(k)}_{\text{óptimo}})$ a $f^*$. Aplicando ahora la desigualdad de Markov, obtenemos que para $\epsilon > 0$,

$$\mathbb{P}(f^{(k)}_{\text{óptimo}} - f^* \geq \epsilon) \leq \frac{\mathbb{E}(f^{(k)}_{\text{óptimo}} - f^*)}{\epsilon}.$$

Cosa que prueba la convergencia en probabilidad, ya que cuando $k \to \infty$ la banda derecha de la desigualdad tiende a 0. Por lo tanto, también lo hace la banda izquierda.

### 4.4.4. Aplicaciones en Machine Learning y variantes

Acabamos de ver un poco la teoría detrás de todo el método de subgradientes estocásticos. Ahora bien, necesitamos precisar como se utiliza este método para nuestro cometido, minimizar la función de coste. Sea $\mathcal{L}(y_i, f(x_i))$ nuestra función de coste. Como ya hemos visto, las funciones de coste en aprendizaje automático son de la forma

$$\mathcal{L}(y_i, f(x_i, \theta)) = \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}_i$$

donde $\mathcal{L}_i$ es el coste de cada ejemplo de entrenamiento, y $\theta$ es el espacio de parámetros. Suponemos que la cantidad de ejemplos de entrenamiento es $n$. El método de subgradientes estocásticos se aplicaría sobre este función de la siguiente manera:

$$\theta_k = \theta_{(k-1)} - \alpha_k \frac{1}{j}\sum_{i=1}^{j} \nabla_{\theta_{k-1}} \mathcal{L}_i.$$

donde $\alpha_k$ es el step size, también conocido como learning rate, y $j < n$ es la dimensión de la muestra aleatoria de ejemplos de entrenamiento. Normalmente $\theta_0$ se inicializa con valores aleatorios pequeños.

La convergencia de este método depende mucho de la elección del learning rate $\alpha_k$. Es una de las principales diferencias entre aplicar un método de subgradientes con o sin minibatch. Una condición suficiente para garantizar la convergencia de este método, es que el learning rate sea de cuadrado sumable pero no sumable, como ya hemos demostrado en el apartado 4.4.3.

En Deep Learning, los gradientes $\nabla\theta_k\mathcal{L}_i$ se calculan con el método de backpropagation. Este método varia según en que tipo de red neuronal se aplique. No es lo mismo aplicarlo en una red convolucional que en una red perceptrón multicapa.

El stochastic subgradient method es el método más popular a la hora de minimizar funciones de coste en algoritmos de Machine Learning. Pese a eso, a veces, la convergencia es algo lenta y puede acarrear problemas numéricos. Para hacer frente a este problema, se han creado variantes del método, que aceleran la convergencia y estabilidad del mismo.

- **Momentum**: Su nombre proviene de la relación con la cantidad de movimiento en física. Se define un parámetro $\beta \in [0, 1)$, que determina qué tan rápido decaen exponencialmente los gradientes calculados previamente. El método consiste en:

$$\begin{aligned} v_k &= \beta v_{k-1} - \alpha_k \frac{1}{j}\sum_{i=1}^{j} \nabla_{\theta_{k-1}}\mathcal{L}_i. \\ \theta_k &= \theta_{k-1} + v_k \end{aligned}$$

  La variable $v$ acumula $\frac{1}{j}\sum_{i=1}^{j} \nabla_{\theta_{k-1}}\mathcal{L}_i$ y según $\beta$ los gradientes previos afectarán más o menos a la dirección actual.

- **AdaGrad**: Este algoritmo adapta individualmente los learning rates con un escalado inversamente proporcional a la raiz cuadrada de la suma de todos los cuadrados de los gradientes anteriores. Los parámetros con derivada parcial sobre la función de coste más alta, decrecen más rápidamente su learning rate. El método se puede expresar como:

$$\begin{aligned} r_0 &= 0, \quad \delta \approx 10^{-7} \\ r_k &= r_{k-1} + \frac{1}{j}\sum_{i=1}^{j} \nabla_{\theta_{k-1}}\mathcal{L}_i \odot \frac{1}{j}\sum_{i=1}^{j} \nabla_{\theta_{k-1}}\mathcal{L}_i. \\ \hat{\theta}_k &= -\frac{\alpha_k}{\delta + \sqrt{r_k}} \odot \frac{1}{j}\sum_{i=1}^{j} \nabla_{\theta_{k-1}}\mathcal{L}_i. \\ \theta_k &= \theta_{k-1} + \hat{\theta}_k. \end{aligned}$$

  donde la operación $\odot$ denota el producto de Hadamard (elemento a elemento), y en la tercera igualdad, tanto la división como la raiz cuadrada también se aplica elemento a elemento. El valor de $\delta$ se suele escoger cercano a $10^{-7}$ por motivos de estabilidad numérica.

- **RMSProp**: Es muy parecido a AdaGrad con una pequeña ponderación a la hora

de actualizar el cuadrado de los gradientes anteriores. El método consiste en:

$$\begin{aligned}
&r_0 = 0, \quad \delta \approx 10^{-6}, \quad \rho \in [0,1) \\
&r_k = \rho r_{k-1} + (1-\rho)\frac{1}{j}\sum_{i=1}^{j} \nabla_{\theta_{k-1}}\mathcal{L}_i \odot \frac{1}{j}\sum_{i=1}^{j} \nabla_{\theta_{k-1}}\mathcal{L}_i. \\
&\hat{\theta}_k = -\frac{\alpha_k}{\delta + \sqrt{r_k}} \odot \frac{1}{j}\sum_{i=1}^{j} \nabla_{\theta_{k-1}}\mathcal{L}_i. \\
&\theta_k = \theta_{k-1} + \hat{\theta}_k.
\end{aligned}$$

AdaGrad está pensado para una convergencia rápida con funciones convexas. No obstante, en funciones no convexas puede entrañar problemas. En cambio, RMSProp está pensado para actuar mejor en funciones no convexas gracias a la ponderación hecha en los gradientes. En este caso, $\delta$ suele ser cercano a $10^{-6}$ por estabilización de la división hecha en la tercera igualdad.

- **Adam**: Su nombre proviene del término ingles adaptive moments. A grandes rasgos, es como una mezcla entre el método del momentum y el método RMSProp con algunas variaciones que corrigen el sesgo de $v_k$ y $r_k$. El método consiste en:

$$\begin{aligned}
&r_0 = 0, \quad v_0 = 0, \quad \delta \approx 10^{-8}, \quad \rho_1, \rho_2 \in [0,1) \\
&v_k = \rho_1 v_{k-1} + (1-p_1)\frac{1}{j}\sum_{i=1}^{j} \nabla_{\theta_{k-1}}\mathcal{L}_i \\
&r_k = \rho_2 r_{k-1} + (1-\rho_2)\frac{1}{j}\sum_{i=1}^{j} \nabla_{\theta_{k-1}}\mathcal{L}_i \odot \frac{1}{j}\sum_{i=1}^{j} \nabla_{\theta_{k-1}}\mathcal{L}_i. \\
&\hat{v}_k = \frac{v_k}{1-\rho_1^k} \\
&\hat{r}_k = \frac{r_k}{1-\rho_2^k} \\
&\hat{\theta}_k = -\alpha_k \frac{\hat{v}_k}{\delta + \sqrt{\hat{r}_k}} \odot \frac{1}{j}\sum_{i=1}^{j} \nabla_{\theta_{k-1}}\mathcal{L}_i. \\
&\theta_k = \theta_{k-1} + \hat{\theta}_k.
\end{aligned}$$

  Los valores recomendados para las constantes son $\rho_1 = 0{,}9$, $\rho_2 = 0{,}999$ y $\delta \approx 10^{-8}$.

Actualmente los métodos más populares a la hora de resolver problemas de Deep Learning son: el método con subgradientes estocásticos, con y sin momentum, el RMSProp, con y sin momentum, el AdaGrad con alguna varitante y el Adam.

En estos momentos, no hay consenso sobre cuál de los métodos vistos es más efectivo a la hora de enfrentarse a un problema. La elección suele basarse en los conocimientos del usuario sobre los métodos y con cuál de ellos se siente más cómodo.

## 5. Práctica: Reconocimiento de dígitos

Después de haber visto un poco la teoría detrás de los métodos con subgradientes estocásticos, y cómo funciona la propagación hacia atrás, vamos a ver un ejemplo práctico de una red neuronal profunda fully connected, donde se aplican ambos conceptos para solucionar un problema concreto.

Este es uno de los problemas más clásicos de Deep Learning. De entrada, tenemos una imagen de 8x8 píxeles que contiene un dígito del 0 al 9. Cada entrada viene con su valor esperado. Es decir, la entrada consiste en una matriz $A \in \mathbb{R}^{8\times 8}$ y un natural $n \in \{0, \cdots, 9\}$, que indica que número se nos está mostrando en la imagen. Mediante estas entradas, entrenaremos nuestra red neuronal para que sea capaz de detectar que dígitos se muestran en las imágenes.

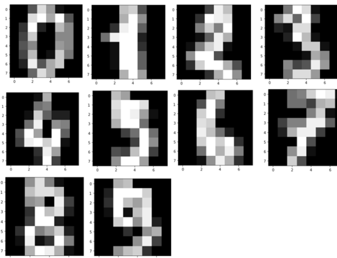

Figura 7: Aquí tenemos unos ejemplos de los datos de entrada que le hacemos llegar al algoritmo. Se ha utilizado la librería *sklearn.datasets*. Fuente: creación propia.

En este caso, hemos escogido una red neuronal perceptrón multicapa, con una capa ocultas de 30 neuronas. La salida es una única neurona que será nuestra predicción del dígito. Esto, junto al valor esperado que viene junto al ejemplo, nos sirve para ir calculando la función de coste.

La función de coste escogida es el error cuadrático medio, del que ya hemos hablado anteriormente.

En cuanto a la función de activación, se ha utilizado la función ReLU, en toda la red. En la última capa oculta, se han de hacer unos pequeños ajustes para dar la predicción como un valor simple.

En el código, podemos ver como se aplica el método con subgradientes estocásticos. En primer lugar, se escogen aleatoriamente muestras de ejemplos de entrenamiento. En segundo lugar, se calcula su subgradiente (como la función de coste es diferenciable, en este caso, el gradiente es el único subgradiente) y se aplica el método de subgradientes estocásticos para minimizar la función de coste. Por otro lado, en las propias definiciones de las capas, se puede ver bajo los nombres de backward y gradient, el proceso de backpropagation, teniendo en cuenta que la función de activación escogida es una ReLU [18].

En este algoritmo no se han utilizado los sesgos.

Al ser un algoritmo muy simple, con una función de coste no muy adecuada para clasificaciones, el error es bastante alto. No obstante, podemos ver que el algoritmo está aprendiendo con los datos. También ha influido el hecho de escoger unos datos tan simplificados y el no haber utilizado capas convolucionales en el proceso.

El objetivo de este trabajo no era crear el mejor algoritmo, sino ver en la práctica los conceptos de los que hemos hablado durante el trabajo.

Modificar la capacidad del modelo, buscar otras funciones de coste, cambiar la función de activación o modificar la arquitectura del algoritmo serían algunas de las formas de intentar mejorar el algoritmo.

Para hacernos a la idea hasta donde podemos llegar si mejoramos nuestro algoritmo, el MNIST (Modified National Institute of Standards and Technology database) cuenta con una base de datos de dígitos escritos a mano de 28x28 píxeles, y a día de hoy se han conseguido algoritmos con una precisión más alta que el 99 %.

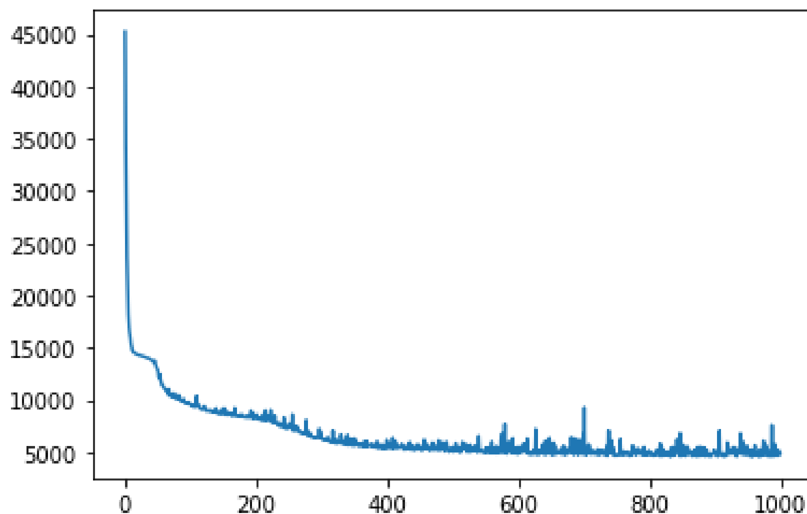


Figura 8: Impresión por pantalla del programa. El eje $X$ se corresponde con las iteraciones y el eje $Y$ con el valor de la función de coste. Se puede apreciar el comportamiento estocástico del algoritmo por las ondulaciones de la función de coste.

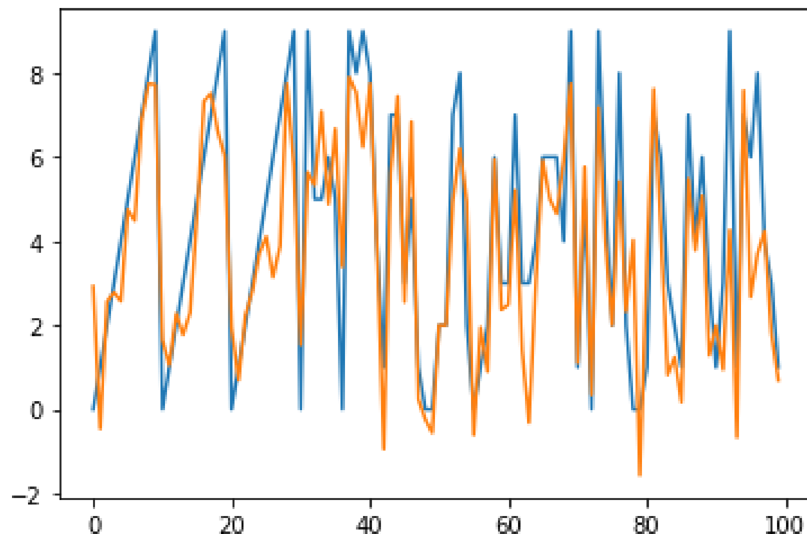


Figura 9: Impresión por pantalla del programa. En azul, el valor esperado que viene con la entrada. En naranja, el valor que hemos obtenido de salida del algoritmo. La precisión del modelo es del 26 %, un 16 % más alta que un modelo completamente aleatorio.

# 6. Conclusiones

Mi objetivo principal al realizar este trabajo era aprender y mostrar cómo las matemáticas actúan en los modelos de Machine Learning y, particularmente, en los de Deep Learning.

Para lograr este objetivo, se han dado las nociones básicas de los algoritmos de aprendizaje automático, haciendo énfasis en Deep Learning y concretamente, en las redes neuronales perceptrón multicapa.

Por otro lado, se ha introducido el tema de la optimización matemática, describiendo sus variantes, para dar paso a explicar con rigor matemático el método de optimización más utilizado en Machine Learning: el método de subgradientes estocásticos.

Considero que el trabajo ha quedado bastante completo y permite a un lector, con previos conocimientos de matemáticas, a introducirse en el campo del aprendizaje profundo, y a entender cómo funcionan la mayoría de algoritmos de este campo.

En cuanto al código, pese a ser un algoritmo muy básico y obviamente no ser perfecto, sirve para tener una idea de la estructura básica que todo algoritmo de Deep Learning debe tener, en un lenguaje de programación moderno, como es en nuestro caso, Python.

A nivel personal, este trabajo me ha servido para indagar profundamente en un tema en el cual estoy muy interesado, como es la Inteligencia Artificial, y su relación con las matemáticas. Espero continuar poder aprendiendo más sobre el tema, y en su debido momento, poder dedicarme a ello como profesional.

Considero que es una rama de las ciencias de la computación con mucho potencial en los próximos años, y que permitirá al ser humano a alcanzar unos recursos tecnológicos que a día de hoy no somos capaces de imaginar.

En resumen, estoy muy satisfecho con mi trabajo y mi aprendizaje a raíz del mismo, y espero que el lector haya sido capaz de disfrutar y entender lo que he intentado transmitir y explicar en él.

# 7. Anexos

## 7.1. Código en Python de una red MLP fully conncected para reconocimiento de dígitos

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt

#Definición de la función de coste
class loss:
def __init__(self):
self.y = 0
def evaluate(self,model,x,t):
#Evalua la función de coste L(y,t)
self.y = model.forward(x)
return (t-self.y)*(t-self.y) #Error cuadrático
def gradient(self,model,x,t):
self.y = model.forward(x)
return -2.*(t-self.y)

#Definición genérica de capa
class layer:
def __init__(self):
pass
def forward(self): # Evalua la capa
pass
def backward(self): # Gradiente respecto a los inputs
pass
def gradient(self): # Gradiente respecto a los parámetros
pass

#Definición de una capa lineal con función de activación ReLU
class reluLayer(layer):
def __init__(self,input_dim,n_neurons):
self.z = np.zeros((1,n_neurons))
self.w = np.random.randn(input_dim+1,n_neurons)
self.y = np.zeros((1,n_neurons))
def forward(self,x): # Evalua la capa
self.xext = np.concatenate((x,np.ones((x.shape[0],1))),axis=1)
self.z = np.dot(self.xext, self.w)
self.y = np.maximum(0,self.z)
return self.y
def backward(self): #Gradiente respecto a los inputs
dydz = np.tile(np.where(self.z>0.,1.,0.),(self.w.shape[0],1))
tw = dydz*self.w
return tw[:-1,:]
def gradient(self): # Gradiente respecto a los parámetros
dydz = np.where(self.z>0.,1.,0.)
return np.dot(dydz.T,self.xext)

#Definición de una capa lineal con función de activación ReLU
class linearLayer(layer):
def __init__(self,input_dim,n_neurons):
self.w = np.random.randn(input_dim+1,n_neurons)
self.y = np.zeros((1,n_neurons))
def forward(self,x): # evaluate the layer
self.xext = np.concatenate((x,np.ones((x.shape[0],1))),axis=1)
self.y = np.dot(self.xext, self.w)
return self.y
```

```
def backward(self): # Gradiente respecto a los inputs
return self.w[:-1,:]
def gradient(self): # Gradiente respecto a los parámetros
return self.xext


#Definición del modelo
class model:
def __init__(self):
self.architecture = []
self.y_ = []
def addLayer(self,layer):
self.architecture.append(layer)
def forward(self,x):
#Evalua f(x,w)
self.y_= x
for layer in self.architecture:
self.y_=layer.forward(self.y_)
return self.y_

#Definición del algoritmo de optimización
class optimize:
def __init__(self, debug = False, plot_convergence = 10.):
self.debug_ = debug
self.pc = plot_convergence

def run(self, data, target, model, loss, num_iter, eta):
#Dado una función de coste y un modelo,
#Encuentra los parámetros óptimos para un conjunto concreto de datos
self.l = np.zeros((int(np.ceil(num_iter/self.pc)),1))
i=0

N_samples = data.shape[0]
for t in range(num_iter):
#Paso 1.-Escogemos aletoriamente una muestra de datos de entramiento
idx = np.random.randint(N_samples)
xi = data[idx]
xi = xi[np.newaxis,:]
yi = target[idx]
yi = yi[:,np.newaxis]
if (t*1.)%self.pc==0:
self.l[i] = np.sum(loss.evaluate(model,data,target))
i=i+1

#Paso 2.- Método con gradientes estocásticos para optimizar los parámetros
dLdx= loss.gradient(model,xi,yi)
for layer in reversed(model.architecture):
tmp = layer.w - eta * np.dot(dLdx,layer.gradient()).T
dLdx = np.dot(dLdx,layer.backward().T)
layer.w = tmp
def plot(self):
plt.plot(self.l)
```

Listing 1: Perceptrón multicapa con función de activación ReLU

Los datos se han extraído de la librería *sklearn.datasets*, mediante los siguientes comandos. La última linea de esta parte del código, muestra en formato 8x8 la imagen de entrada según el dígito que le pidamos. En la figura 7, podemos ver el aspecto de los dígitos.

```
#Red Neuronal fully connected para reconocimiento de dígitos
from sklearn.datasets import load_digits
data = load_digits()
x=data.data/16.
y=data.target
y=y[:,np.newaxis]

plt.imshow(x[9,:].reshape(8,8),cmap='gray')
```

Una vez creada la red, queda marcar los parámetros que necesita para ejecutarse, y ejecutarla. En las siguientes líneas de código podemos ver:

- El número de iteraciones.
- El learning rate o step size (en este caso constante) igual a 0.001.
- La definición de las capas.
- La inicialización de la función de coste $L$
- Ejecutar el programa.
- Imprimir por pantalla el gráfico iteraciones-función de coste.

```
num_iter = 100000
eta = 0.001 #optimization step/learning rate
nn = model()
nn.addLayer(reluLayer(64,30))
nn.addLayer(linearLayer(30,1))
L = loss()
opt = optimize(plot_convergence=100)
x_mod = opt.run(x,y,nn,L,num_iter,eta)
opt.plot()
plt.plot(y[0:100])
plt.plot(nn.forward(x[0:100]))
```

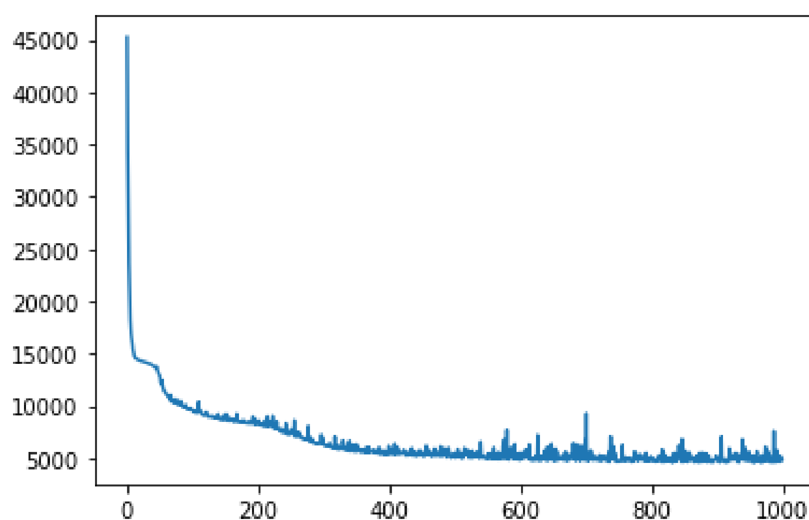


Figura 10: Impresión por pantalla del programa. El eje $X$ se correspoende con las iteracions y el eje $Y$ con el valor de la función de coste.

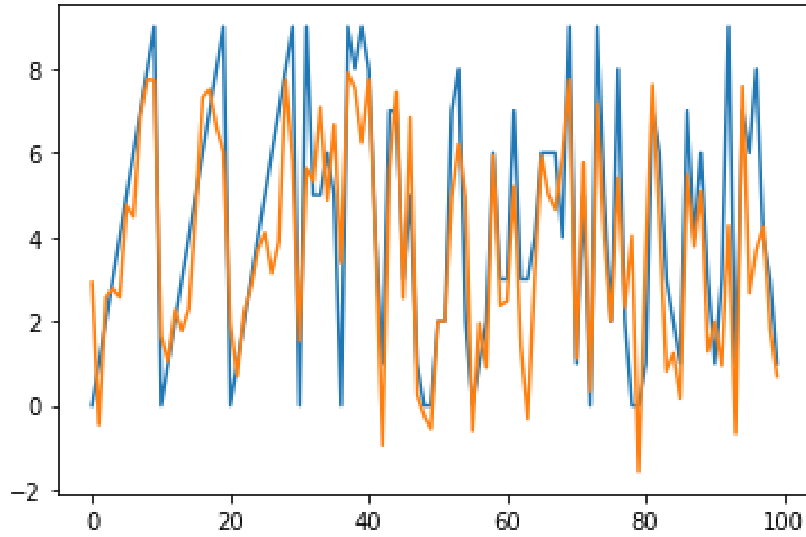


Figura 11: Impresión por pantalla del programa. En azul, el valor esperado que viene con la entrada. En naranja, el valor que hemos obtenido de salida del algoritmo. La precisión del modelo es del 26 %, un 16 % más alta que un modelo completamente aleatorio.

La estadística nos permite tratar los posibles problemas que surgen en los procesos de creación de los algoritmos. Las propiedades estadísticas de los datos y de los modelos nos permiten saber cuando y en que medida, estamos haciendo las cosas correctamente.

Haremos un repaso de las principales definiciones y propiedades que servirán si el lector quiere hacer un breve repaso de estadística básica.

Sea $\Omega$ el conjunto al que pertenecen las observaciones, denominado como espacio muestral. Este conjunto $\Omega$ tiene asociado una familia de subconjuntos $\mathbb{A}$ con estructura de $\sigma$-álgebra. Por lo tanto, la dupla $(\Omega, \mathbb{A})$ es un espacio de medida. En la gran mayoría de los casos, $\Omega \subset \mathbb{R}$ y $\mathbb{A}$ será la $\sigma$-álgebra de Borel.

**Definición 7.1.** *Un modelo estadístico es una terna* $(\Omega, \mathbb{A}, \mathbb{P})$*, dónde* $(\Omega, \mathbb{A})$ *es un espacio de medida y* $\mathbb{P}$ *es una familia de probabilidades en* $(\Omega, \mathbb{A})$*. La asignación de* $\mathbb{P}$ *se hace según la información que se tiene sobre las observaciones y con las hipótesis razonables que se pueden hacer sobre ellas. En nuestro caso, las observaciones son los datos de entrada.*

**Definición 7.2.** *Un modelo estadístico* $(\Omega, \mathbb{A}, \mathbb{P})$ *se conoce como paramétrico si* $\mathbb{P} = \{P_\theta, \theta \in \Theta\}$ *donde* $\Theta$ *es un subconjunto de* $\mathbb{R}^d$ *con* $d \geq 1$*. El subconjunto* $\Theta$ *se conoce como espacio de parámetros.*

Es nuestro caso, el parámetro $\theta$ es desconocido y debemos estimarlo a partir de los datos de entrada, que supondremos finitos. Nuestro objetivo principal consiste en encontrar a partir de los datos un valor aproximado del parámetro $\theta$ o de alguna función del parámetro $g(\theta) : \Theta \longrightarrow \mathbb{R}^k$.

**Definición 7.3.** *Un estimador de* $g(\theta)$ *es cualquier estadístico* $T : \Omega \longrightarrow g(\Theta)$ *que nos sirva para el objetivo que hemos mencionado.*

**Definición 7.4.** *La esperanza matemática de una variable aleatoria discreta* $X$ *es:*

$$\mathbb{E}(X) = \sum_{1}^{n} x_i p(x_i)$$

*siempre y cuando X tiene esperanza finita, es decir si:*

$$\sum_{1}^{n} |x_i| p(x_i) < \infty$$

*donde $p(X)$ es la función de probabilidad asociada a la variable X.*

**Definición 7.5.** *La esperanza matemática de una variable aleatoria absolutamente continua es*

$$\mathbb{E}(X) = \int_{-\infty}^{\infty} x f(x) dx$$

*siempre y cuando X es una variable absolutamente continua con esperanza finita, es decir si:*

$$\int_{-\infty}^{\infty} |x| f(x) < \infty$$

*donde $f(x)$ es la función de densidad asociada a la variable X.*

**Definición 7.6.** *Un estadístico $T$ es integrable si tiene esperanza finita para todo $\theta$, por lo tanto,*

$$\mathbb{E}_\theta(|T|) < \infty, \forall \theta \in \Theta.$$

**Definición 7.7.** *El sesgo de un estimador $T$ integrable respecto a $g(\theta)$ es la función*

$$b_T(\theta) = \mathbb{E}_\theta(T) - g(\theta).$$

**Definición 7.8.** *Un estimador es insesgado o no tiene sesgo cuando*

$$b_T(\theta) = 0 \Leftrightarrow \mathbb{E}_\theta(T) = g(\theta).$$

**Definición 7.9.** *La varianza de un conjunto de datos $X = \{x_1, x_2, ..., x_n\}$ se define como:*

$$Var(X) = \sigma_X^2 = \frac{1}{n} \sum_{i=1}^{n} (x_i - \bar{x})^2 = \mathbb{E}[(X - \bar{x})^2] = \mathbb{E}(X^2) - \mathbb{E}(X)^2$$

*La desviación típica es la raiz cuadrada de la varianza. Por lo tanto:*

$$\sigma_X = \sqrt{\frac{1}{n} \sum_{i=1}^{n} (x_i - \bar{x})^2}$$

**Definición 7.10.** *La varianza corregida se define como:*

$$\hat{\sigma}_X^2 = \frac{1}{n-1} \sum_{i=1}^{n} (x_i - \bar{x})^2 = \frac{n}{n-1} \sigma_X^2$$

Dependiendo del objetivo que queramos alcanzar, nos interesará más utilizar estimadores con o sin sesgo. Las varianzas y la desviación típica nos permiten saber cuanta dispersión tenemos en los datos. El error de generalización se suele calcular mediante el error estándar, que por definición es:

$$SE = \sqrt{Var(\bar{x})} = \frac{\sigma}{\sqrt{n}}$$

Donde $n$ es la dimensión de la muestra.

**Definición 7.11.** *Decimos que una sucesión de estimadores* $\{T_n, n \geq 1\}$ *es consistente si para todo* $\epsilon > 0$ *y para todo* $\theta \in \Theta$ *tenemos que:*

$$\lim_{n\to\infty} P_\theta(|T_n - g(\theta)|) > \epsilon) = 0$$

*Esto es equivalente a que la sucesión* $T_n$ *converge en probabilidad a* $g(\theta)$*, que solemos escribir así:*

$$T_n \xrightarrow[n\to\infty]{P_\theta} g(\theta), \forall\theta \in \Theta$$

Gracias a la consistencia podemos determinar si, a medida que los datos de entrenamiento aumentan, nuestra estimación tiende al resultado esperado de los correspondientes parámetros. En otras palabras, nos garantiza que, a mayor cantidad de datos, menos sesgo. En algunos casos, se utilizan los estimadores fuertemente consistentes. La diferencia con los anteriores es que en vez de exigir la convergencia en probabilidad, exigimos la convergencia casi segura:

$$T_n \xrightarrow[n\to\infty]{P_\theta - c.s} g(\theta) \qquad o \qquad \lim_{n\to\infty} \mathbb{E}_\theta(|T_n - g(\theta)|^2) = 0, \quad \forall\theta \in \Theta$$

Para más información sobre este tema, recomendamos el libro [15], que consideramos de gran calidad educativa en cuanto a estadística.

## Referencias


[1] Ian Goodfellow and Yoshua Bengio and Aaron Courville. *Deep Learning.* The MIT Press, 2016.
http://www.deeplearningbook.org

[2] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization.* Cambridge University Press, USA, 2004.
https://web.stanford.edu/~boyd/cvxbook/bv_cvxbook.pdf

[3] B. Polyak. *Introduction to Optimization.* Optimization Software, Inc., 1987.

[4] Stephen Boyd and Almir Mutapcic, with additions by John Duchi. *Stochastic Subgradient Methods.* Notes for EE364b, Stanford University, Spring 2017-2018.

[5] Stephen Boyd (with help from Jaehyun Park). *Subgradient Methods.* Notes for EE364b, Stanford University, Spring 2013-14. fbxg

[6] S. Boyd, J. Duchi, and L. Vandenberghe. *Subgradients.* Notes for EE364b, Stanford University, Spring 2014-15.
https://web.stanford.edu/class/ee364b/lectures/subgradients_notes.pdf

[7] K. Marti. *Stochastic Optimization Methods.* Springer, 2005.

[8] J. Hiriart-Urruty and C. Lemaréchal. *Convex Analysis and Minimization Algorithms I & II.* Springer, New York, 1993.

[9] Jean-Baptiste Hiriart-Urruty and Claude Lemaréchal. *Fundamentals of Convex Analysis.* Springer, 2001.

[10] P. Camerini, L. Fratta, and F. Maffioli. *On improving relaxation methods by modifying gradient techniques.* Math. Programming Study, 1975.

[11] A. Nedić and D. Bertsekas. *Incremental subgradient methods for nondifferentiable optimization.* SIAM J. on Optimization, 2001.

[12] Y. Nesterov. *Introductory Lectures on Convex Optimization: A Basic Course.* Kluwer Academic Publishers, 2004.

[13] A. Nemirovski and D. Yudin. *Problem Complexity and Method Efficiency in Optimization.* Wiley-Interscience, 1983.

[14] N. Shor. *Minimization Methods for Non-differentiable Functions.* Springer Series in Computational Mathematics. Springer, 1985.

[15] Olga Julià i David Márquez-Carreras. *Un primer curs d'estadística.* Publicacions I Edicions de la Universitat de Barcelona, 2011.

[16] L. Ferreira Guilhoto. *An Overview Of Artificial Neural Networks for Mathematicians.*
https://math.uchicago.edu/~may/REU2018/REUPapers/Guilhoto.pdf

[17] Loss functions:
https://ml-cheatsheet.readthedocs.io/en/latest/loss_functions.html

[18] Rectifier in neural networks:
https://en.wikipedia.org/wiki/Rectifier_(neural_networks)

[19] Universal approximation theorem:
https://en.wikipedia.org/wiki/Universal_approximation_theorem#cite_note-cyb-2

[20] Gradient descent:
https://en.wikipedia.org/wiki/Gradient_descent

[21] Stochastic gradient method :
https://en.wikipedia.org/wiki/Stochastic_gradient_descent

[22] Backpropagation:
https://en.wikipedia.org/wiki/Backpropagation

[23] Convolutional networks:
https://towardsdatascience.com

[24] Delta de Kronecker:
https://en.wikipedia.org/wiki/Kronecker_delta

[25] Chain Rule:
https://en.wikipedia.org/wiki/Chain_rule

[26] Line search:
https://en.wikipedia.org/wiki/Kronecker_delta

[27] Interior point methods:
https://en.wikipedia.org/wiki/Interior-point_method

[28] Teorema del hiperplano de soporte:
https://en.wikipedia.org/wiki/Supporting_hyperplane

[29] Envolvente convexa:
https://en.wikipedia.org/wiki/Convex_hull

[30] Dualidad fuerte:
https://inst.eecs.berkeley.edu/~ee227a/fa10/login/l_dual_strong.html

[31] Método simplex:
https://es.wikipedia.org/wiki/Algoritmo_s%C3%ADmplex