Trabajo monográfico

Fundamentos matemáticos del aprendizaje profundo

Aplicaciones en el procesamiento del lenguaje natural

Christian Fachola Garagorry

Orientadoras:

Paola Bermolen
María Inés Fariello
Instituto de Matemática y Estadística Rafael Laguardia

Licenciatura en Matemática
Universidad de la República
Montevideo, Uruguay

Índice general

CAPÍTULO 1

Introducción

1.1. Motivación

La inferencia estadística no es un campo nuevo, la podemos entender como el estudio de métodos que usan conjuntos de observaciones conocidas para inferir información sobre nuevas observaciones desconocidas. Sin embargo, en estas últimas décadas se ve una proliferación (o resurgimiento) de métodos estadísticos a los que podemos referirnos en conjunto como 'aprendizaje automático' o *machine learning* en inglés. Esta situación se debe a la investigación y experimentación continua en el área, de actores públicos o privados, del aumento sostenido en la capacidad de cómputo y almacenamiento ofrecidos por la tecnología, y por la abundancia de datos fruto natural de la digitalización del mundo. Regresiones, árboles de decisión, *support vector machines*, redes neuronales, son algunos de los métodos que pertenecen a la categoría de 'aprendizaje automático' .

Las redes neuronales, particularmente, se ubican en el estado del arte de muchas aplicaciones. En los años que preceden a la fecha en la que se escribe este trabajo, se ve un uso generalizado de herramientas de *inteligencia artificial* (IA) como la serie GPT (OpenAI), Claude (Anthropic), Gemini (Google), DeepSeek, entre otras. Herramientas con las que se puede tener una conversación fluida como si fuera una persona quien responde, que soportan procesar datos en distintos formatos (texto, imágenes, audio), y que pueden realizar tareas muy diversas. Es en este contexto que se escribe este trabajo, en un área donde ya, probablemente por su alta aplicabilidad, las implementaciones han avanzado más rápido que la teoría.

La situación motiva a abordar el tema buscando fundamentos teóricos para el área, por eso hablaremos de optimización y del teorema de aproximación universal para redes neuronales. A su vez, al tratarse de matemática aplicada, describiremos algunos algoritmos y varias cuestiones de implementación.
No se pretende llegar a comentar en detalle cómo funcionan los 'modelos'[1] en el estado del arte en aprendizaje profundo, como los que soportan la reciente explosión de la IA que enmarca a las herramientas mencionadas más arriba. Dicho esto, en el Capítulo 5 se describe la arquitectura del Transformer, que funciona como base de los modelos avanzados actuales. Se da un marco teórico robusto al aprendizaje automático en general y a las redes neuronales en particular, que permite seguir investigando en el área, pero no es un trabajo que busque formalizar totalmente la teoría del aprendizaje automático, para algo así podemos referirnos a [26].

[1]Palabra genérica que engloba a todas esas herramientas para *modelar* la realidad definida por los datos. Desde un punto de vista matemático, son funciones.

1.2. Estructura de la monografía

Este trabajo se divide en 6 capítulos, siendo el primero la presente Introducción. En el Capítulo 2 estudiaremos métodos de procesamiento de lenguaje natural (PLN) independientes de la teoría del aprendizaje profundo, que son útiles en sí mismos, pero sus limitaciones nos motivan a estudiar métodos más complejos. Allí también introduciremos algunos conceptos básicos de PLN y de *machine learning*.

En el Capítulo 3 daremos un marco teórico al concepto de optimización que nos servirá para abordar el Capítulo 4, donde estudiaremos en detalle la regresión logística y las redes neuronales *feedforward*.

Teniendo los fundamentos desarrollados, abordaremos nuevamente las aplicaciones a PLN en el Capítulo 5, donde veremos arquitecturas de aprendizaje profundo con una mirada particular en el dominio de los textos, aunque con suficiente generalidad como para entender cómo se aplican en otros campos.

1.3. Público objetivo

El trabajo puede ser leído por alguien con experiencia práctica en aprendizaje automático pero sin formación formal en matemática y que tenga curiosidad sobre los fundamentos del área, en este caso podría interesarle más el Capítulo 3 y las partes más teóricas del Capítulo 4 como la formalización de las redes neuronales, el teorema de aproximación universal y el algoritmo de *backpropagation*.

En contrapartida, quien tenga una formación matemática, pero que se acerque por primera vez al aprendizaje automático, no le será difícil entender el Capítulo 3 y puede centrarse más en los Capítulos 2, 4 y 5. Se trata de acompañar las menciones de arquitecturas concretas y técnicas con referencias; de esta manera, el texto es un punto de entrada para investigar el área en mayor profundidad, en diferentes direcciones.

CAPÍTULO 2

Principios de IA en PLN

En este capítulo veremos maneras en las que se pueden procesar textos escritos en lenguaje natural. Por 'procesar' nos referimos a extraer información de los textos, clasificarlos o a cualquier otro uso inteligente de los mismos. A cualquiera de estos métodos los catalogamos como IA, porque involucran la automatización de tareas que de otra manera requieren una inteligencia humana para realizarlas.

En la Sección 2.1 veremos dos ejemplos que utilizan 'reglas duras', es decir, lógicas deterministas. En cambio, en la sección 2.2 veremos cómo los datos, en este caso los mismos textos, se pueden usar como fuente de información, *aprendiendo* de los patrones que se encuentran en ellos.

2.1. Métodos basados en reglas

Este tipo de métodos fueron los primeros en desarrollarse. Son aquellos donde el diseñador del algoritmo tiene más control y son, además, la base para algunas etapas del procesamiento de los textos, realizado incluso hoy en día.

No son herramientas que puedan por sí solas resolver los problemas difíciles en PLN, como generar textos coherentes, traducciones fehacientes, etc. En cambio, son suficientes en contextos acotados y sus falencias en tareas complejas nos dan la motivación para hablar de otras formas de abordar esas problemáticas.

2.1.1. Expresiones regulares.

Una expresión regular (ER) es un formalismo para describir un conjunto de cadenas de caracteres o *strings*. Por ejemplo, la ER definida como

$$r := a*$$

representa el conjunto de cadenas formadas por cero o más `a`. Allí, la letra `a` tiene su significado literal, mientras que el carácter `*` es un símbolo especial que denota que el símbolo inmediatamente anterior puede llegar a repetirse cero o más veces.

En la teoría de los lenguajes formales, las expresiones regulares definen al conjunto de los lenguajes regulares y se prueba (ver, por ejemplo, Capítulo 3 de [10]) que a toda ER `r` le corresponde un autómata finito (una máquina de estados) F_r que puede validar si un string es parte del lenguaje regular definido por r.
Es mediante la mencionada equivalencia que las ER no solo son meros descriptores de conjuntos, también sirven para validar el formato de strings.

Ejemplo 2.1.1.
Una dirección de una calle en Montevideo puede describirse como un texto que empiece por las palabras `Calle`, `Av.` o `Blvar.`, seguido de una palabra y un número. Esta misma definición es capturada por la siguiente ER:

```
r := (Calle|Av.|Blvar.) ([A-Z][a-z])+ [0-9]+.
```

Los paréntesis `(,)` sirven para agrupar, son un carácter especial como el `*`. Lo encerrado en los primeros paréntesis quiere decir que el string debe empezar por alguna de las tres palabras que mencionamos antes, el carácter `|` es el *OR* lógico. Después, `[A-Z]` representa a una letra mayúscula del abecedario, mientras que `[a-z]` representa a cualquier letra minúscula. El símbolo `+` indica que lo inmediatamente anterior debe estar una o más veces. Por último, `[0-9]+` representa al menos un dígito del 0 al 9.

Las ER se pueden usar de forma directa para algunas aplicaciones de PLN, por ejemplo, clasificación de texto.

Ejemplo 2.1.2. **Análisis de sentimientos usando expresiones regulares**
El siguiente es un ejemplo rudimentario de análisis de sentimiento, que es la clasificación de textos en categorías positiva y negativa, por ejemplo, para reseñas de productos en venta en una página web. Usamos la ER

```
r1 := (excelente | muy bueno | me encantó),
```

identificando la presencia de subconjuntos de palabras normalmente asociadas con sentimientos positivos como "excelente", "muy bueno" y "me encantó". Del mismo modo, habría que tener una ER con palabras negativas, por ejemplo:

```
r2 := (espantoso | muy malo | defectuoso)
```

.
El algoritmo de clasificación consiste en identificar la presencia de estas palabras; si hay más de la lista positiva, se considera la reseña como positiva, de lo contrario, se considera negativa.

Hay dos problemas evidentes en la metodología del ejemplo anterior: tenemos que confeccionar una lista exhaustiva de palabras positivas o negativas, y, además, una misma palabra puede pertenecer a una clase o a otra dependiendo del *contexto*. Meramente identificar la presencia de una palabra sin tener en cuenta las otras palabras que la rodean no es suficiente. Profundizaremos en esta cuestión de darle un contexto a las palabras en el Capítulo 5 donde veremos el enfoque utilizado para las redes neuronales.

A pesar de sus limitaciones, ERs como las del ejemplo se pueden utilizar para generar *features*[1] para usar luego en modelos estadísticos. Además, son la base para realizar una de las tareas básicas de preprocesamiento de texto, que se conoce como *tokenization*.

Definición 2.1.1. **Tokenization**
Por 'tokenization' nos referimos al proceso de dividir un texto, o secuencia, para no limitarnos solo al PLN, en elementos indivisibles llamados *tokens*. En textos, los tokens pueden ser palabras, caracteres o hasta partes de palabras.

[1]Variables dependientes, también llamadas características, en un problema de modelado estadístico.

El enfoque más sencillo para definir los tokens es considerar cada una de las secuencias separadas por un espacio, u otros separadores, como un token. La tarea de definir los separadores (como espacios, puntos, etc.) se especifica con expresiones regulares.
En Python, la librería `re` permite trabajar con ERs, no solo para validar formato o detectar ciertos patrones en un texto, sino también para transformar el texto mediante sustituciones. Un ejemplo conocido de uso más avanzado de ERs es el de ELIZA[4], un programa de los años 60 que intentaba simular la conversación con un psicólogo.

2.1.2. Gramáticas Libres de Contexto.

Supongamos que queremos programar una herramienta para escribir automáticamente textos con un estilo literario en particular, por ejemplo, en base a un texto de Borges:

> *El universo (que otros llaman la Biblioteca) se compone de un número indefinido, y tal vez infinito, de galerías hexagonales, con vastos pozos de ventilación en el medio, cercados por barandas bajísimas. Desde cualquier hexágono se ven los pisos inferiores y superiores: interminablemente. La distribución de las galerías es invariable. Veinte anaqueles, a cinco largos anaqueles por lado, cubren todos los lados menos dos; su altura, que es la de los pisos, excede apenas la de un bibliotecario normal.*
>
> – Jorge Luis Borges, *La Biblioteca de Babel*

Una manera para atacar el problema, basada en reglas duras, son las gramáticas libres de contexto (GLC). Las GLC se parecen a las expresiones regulares, en el sentido de que también son un formalismo para identificar elementos y patrones en un texto, con la diferencia de que dan una manera sistemática de generar los mismos. Una GLC consta de variables, símbolos terminales, un símbolo inicial y reglas de producción que nos dicen cómo sustituir variables por otras variables o por símbolos terminales. Veamos un ejemplo simple:

Ejemplo 2.1.3. **Lenguaje de paréntesis**

```
Variables: S
Símbolos terminales: (,)
Símbolo inicial: S
Reglas de producción:
    1. S -> SS
    2. S -> (S)
    3. S -> v
```

Allí, el símbolo `v` representa la cadena vacía. Esta gramática puede generar textos como `()`, `(())`, `()()`, `((()))`, etc. Por ejemplo, así se obtiene la secuencia de caracteres `(())`:

$$S \xrightarrow{\text{regla 2}} (S)$$
$$(S) \xrightarrow{\text{regla 2}} ((S))$$
$$((S)) \xrightarrow{\text{regla 3}} (()).$$

Para resolver el problema planteado al principio, nos apoyamos en el conocimiento generado por el campo de la linguística, escribiendo reglas acordes a la gramática del idioma español. La siguiente es una GLC que intenta hacer justamente esto.

EJEMPLO 2.1.4. **Gramática libre de contexto para modelar el texto de Borges.**

```
Variables:
    S: Oración
    NP: Frase nominal (sujeto)
    VP: Frase verbal
    ADJ: Adjetivo
    N: Sustantivo
    V: Verbo
    PP: Frase preposicional
    CONJ: Conjunción

Símbolos terminales:
    Palabras específicas del texto (e.g., ‘‘universo’’, ‘‘galerías’’,
    ‘‘hexagonales’’, ‘‘indefinido’’, etc.)
    Signos de puntuación (e.g., comas, puntos)

Símbolo inicial: S

Reglas de producción:
    1. S -> NP VP
    2. NP -> ADJ N | ART N | N
    3. VP -> V NP | V PP NP
    4. ART -> el | la | los | las
    5. ADJ -> indefinido | infinito | hexagonales |
                vastos | bajísimas
    6. N -> universo | Biblioteca | número | galerías | pozos |
           ventilación | barandas | pisos | anaqueles | bibliotecario
    7. V -> se compone | excede | cubren | ve
    8. PP -> de | en | por
    9. CONJ -> y | o | pero
    10. S -> S CONJ S.
```

Vemos que una GLC para el texto de Borges modela algunas reglas de la gramática del español y tiene como símbolos terminales los sustantivos que identificamos en el extracto del cuento. El siguiente es un ejemplo de oración construida con esta GLC:

$$
\begin{aligned}
S &\xrightarrow{\text{regla 1}} NP\ VP \\
&\xrightarrow{\text{regla 2}} ART\ N\ VP \\
&\xrightarrow{\text{regla 3}} ART\ N\ V\ PP\ NP \\
&\xrightarrow{\text{regla 2}} ART\ N\ V\ PP\ ADJ\ N \\
&\xrightarrow{\text{reglas 5,6,7,4,8}} \text{El universo se compone de infinitas galerías.}
\end{aligned}
$$

Como se puede ver, construir una GLC que modele siquiera una parte del extracto de Borges requiere de conocimiento experto para confeccionar reglas que generen todas las secuencias vistas, pero con cuidado de no generar secuencias sin sentido, por lo tanto, requiere de mucho esfuerzo y tiempo. Aún más trabajo tomaría modelar el cuento completo. Y ni hablar del esfuerzo que requeriría definir una GLC que modele el idioma español en su totalidad.

Esta tarea es irrealizable desde su concepción, aun escribiendo reglas de la forma más cuidadosa posible, usando todo lo que se sabe sobre la sintaxis del español, agregando símbolos terminales para todas las palabras en el diccionario, hay un problema irresoluble con este enfoque: el lenguaje está en constante evolución, cambia, se crean nuevas palabras y nuevas formas de usarlas en combinación con otras todo el tiempo. Esto implica que cualquier conjunto de reglas debería estar en constante escrutinio para ser actualizado y adaptado a las realidades del idioma en cada contexto. Ese enfoque no escala, lo cual no quiere decir que las GLC no sirvan para nada, se utilizan en lenguajes más acotados y con sintaxis rígida, como lo son los lenguajes de programación.

Es necesario cambiar la estrategia, no utilizar reglas creadas por expertos sino utilizar un enfoque *estadístico* que se pueda adaptar rápido a la evolución del lenguaje. Veremos un método de este tipo en la siguiente sección.

2.2. Métodos estadísticos

En esta sección cambiamos el enfoque, vemos a los textos como elementos de un espacio de probabilidad, cada texto (o palabra) es una *observación* que tenemos a disposición. Un problema central en PLN usando este enfoque es el de conseguir un buen *modelo de lenguaje*. Un modelo de lenguaje es una función de probabilidad cuyo dominio son textos (formalmente una secuencia de *tokens*) y retorna la probabilidad de que ese texto exista de manera natural en el lenguaje. Esta probabilidad se puede estimar de forma frecuentista, que es lo que se busca con el método presentado en la siguiente sección.

2.2.1. Modelo de N-gramas.

Vamos a entender cómo usar un enfoque frecuentista para modelar el lenguaje y por qué el uso de los N-gramas, que dan nombre al enfoque y definimos más abajo, ayudan en su implementación en la práctica.

Consideremos un texto constituido por una secuencia $t = w_1, w_2, \ldots, w_i, \ldots, w_n$, definimos la variable aleatoria X_j que toma como valor el token en la posición j, así que la probabilidad de que en la posición j esté el token w_j es $P(X_j = w_j)$, que en un abuso de notación escribiremos $P(w_j)$. La probabilidad del texto completo t es una probabilidad conjunta:

$$P(X_1 = w_1, X_2 = w_2, \ldots, X_n = w_n) = P(w_1, w_2, \ldots, w_n) = P(w_{1:n}),$$

donde introdujimos la notación $w_{1:n}$ para denotar la secuencia de tokens $w_1, \ldots, w_n$. La probabilidad de t es la probabilidad de que el último token sea w_n condicionada a que los anteriores fueron $q = w_{1:n-1}$. Ahora, $P(q)$ es la probabilidad de que el último token sea w_{n-1} dado que los anteriores fueron $w_{1:n-2}$. Esta idea está resumida en la siguiente propiedad, que se puede probar fácilmente por inducción:

PROPIEDAD 2.2.1. *Dados $A_1, \ldots, A_n$ sucesos en un espacio de probabilidad, se tiene que:*

$$\begin{aligned} P(A_1, \ldots, A_n) &= P(A_1)P(A_2|A_1)\ldots P(A_n|A_1, \ldots, A_{n-1}) \\ &= P(A_1)\prod_{k=2}^{n} P(A_k|A_1, \ldots A_{k-1}). \end{aligned} \tag{1}$$

Usamos esta propiedad para calcular la probabilidad de toda la secuencia t:

$$P(w_{1:n}) = P(w_1)\prod_{k=2}^{n} P(w_k|w_{1:k-1}) = P(w_1)P(w_2|w_1)P(w_3|w_1, w_2)\ldots P(w_n|w_{1:n-1}). \tag{2}$$

¿Cómo se estima esta probabilidad en la práctica? Mediante frecuencias medidas en un *corpus*[2] grande. Dada una secuencia de tokens $t = w_{1:n}$, estimamos la probabilidad condicional por su definición mediante frecuencias:

$$P(w_n|w_{1:n-1}) = \frac{count(w_{1:n})}{count(w_{1:n-1})}, \tag{3}$$

donde $count(\cdot)$ es una función que cuenta la cantidad de apariciones de la secuencia argumento en el corpus. El corpus donde se calculan estas frecuencias, es, en el lenguaje del *machine learning*, el corpus de *entrenamiento*. Es de entrenamiento porque es donde vamos a estimar la probabilidad de cada secuencia. Una vez hecho esto, se puede calcular la probabilidad de secuencias nuevas.

El problema con las secuencias nuevas es que puede que no estén en el corpus de entrenamiento, es decir, si la secuencia $q = v_{1:n}$ no está, entonces la ecuación (3) nos da 0, pero sabemos que debería ser mayor a 0 porque q es una secuencia del lenguaje. Para mitigar este problema se asume la hipótesis del N-grama, que consiste en solo considerar los últimos N tokens del texto.

DEFINICIÓN 2.2.1. **N-grama**
Un N-grama es una secuencia de N tokens consecutivos dentro del mismo texto. Por ejemplo, los 2-gramas (o *bigramas*) de "El árbol alto" son `[El, árbol]` y `[árbol, alto]`

[2]Conjunto de textos.

Con la consideración de trabajar con N-gramas, la ecuación (3) nos queda:

$$P(w_n|w_{1:n-1}) \approx P(w_n|w_{n-N+1:n-1}) = \frac{count(w_{n-N+1:n})}{count(w_{n-N+1:n-1})}. \tag{4}$$

Esta simplificación resuelve en buena medida el problema de las secuencias no vistas, ya que no necesitamos encontrar toda la secuencia q en el corpus, sino solamente los N-gramas de q. Así entonces, podemos calcular de manera relativamente fiable la probabilidad de cualquier secuencia, siempre y cuando el corpus sea lo suficientemente grande y variado como para contener N-gramas de la secuencia.

El enfoque frecuentista con uso de los N-gramas también nos sirve para generar texto: dada una palabra w elegimos la siguiente palabra v muestreando v del vocabulario[3] con probabilidad $P(v|w)$, si estamos trabajando con bigramas.

A diferencia de las GLC o las ER, este enfoque no utiliza reglas creadas por expertos. Dado un corpus, los parámetros del modelo (las probabilidades condicionales) se pueden calcular automáticamente. Esto mismo es lo que se busca cuando se ajustan parámetros de un modelo estadístico, ya sea un modelo relativamente simple como el de N-gramas, o una regresión logística, o más complejo, como una red neuronal. Hablaremos en detalle sobre estos últimos dos modelos en el Capítulo 4, que, a diferencia de lo que se hace al trabajar con N-gramas, no se estiman sus parámetros mediante conteo de tokens, sino que se encuentran resolviendo un problema de optimización.

[3]Conjunto de tokens del corpus.

CAPÍTULO 3

Optimización

En muchos algoritmos de aprendizaje automático, como en la regresión logística o redes neuronales del Capítulo 4, la manera en la que se encuentran los parámetros óptimos es mediante un proceso de optimización. En este capítulo daremos un marco teórico al problema.

Sea $f : \mathbb{R}^n \to \mathbb{R}$. Un problema de optimización está dado por encontrar

$$x^* = \operatorname*{argmin}_{x \in C} f(x).$$

Si $C = \mathbb{R}^n$ decimos que es un problema de optimización sin restricciones, de lo contrario, C es un subconjunto propio de $\mathbb{R}^n$ y decimos que el problema es de optimización con restricciones. A veces es posible arribar a una solución analíticamente, pero si esto no es posible, existen métodos numéricos.
En lo que sigue estudiaremos problemas sin restricciones a resolver con métodos numéricos, cuya aplicabilidad y condiciones de convergencia varían según el método y las hipótesis que cumpla la función f.

3.1. Rol de la convexidad y el gradiente

Sea f una función diferenciable, $f : \mathbb{R}^n \to \mathbb{R}$. En esta sección vamos a probar que en estas condiciones, el problema de hallar el mínimo de f es equivalente a encontrar puntos críticos de f. Para eso, primero, vamos a probar el siguiente lema:

LEMMA 3.1.1. *Sea $f : \mathbb{R}^n \to \mathbb{R}$ diferenciable y convexa, entonces $f(z) \geq f(x) + \langle \nabla f(x), z - x \rangle$ para todo $x, y \in \mathbb{R}^n$.*

PRUEBA. Por definición, f es convexa si $\forall t \in [0,1], \forall x, y \in \mathbb{R}^n$ se tiene $f(tx + (1-t)y) \leq tf(x) + (1-t)f(y)$. Dados $x, z \in \mathbb{R}^n$ y $\alpha \in (0,1]$, definimos $v = x + \alpha(z-x) \forall \alpha \in (0,1]$. Ese vector v es un punto del segmento que une a x y z. Reescribiendo se tiene $v = \alpha z + (1-\alpha)x$, por lo tanto, por convexidad de f se tiene que:

$$\begin{aligned} f(x + \alpha(z-x)) \leq \alpha f(z) + (1-\alpha)f(x) \Rightarrow & \frac{f(x + \alpha(z-x))}{\alpha} \leq f(z) + \frac{f(x)}{\alpha} - f(x) \\ \Rightarrow & \frac{f(x + \alpha(z-x)) - f(x)}{\alpha} \leq f(z) - f(x) \\ \Rightarrow & \lim_{\alpha \to 0^+} \frac{f(x + \alpha(z-x)) - f(x)}{\alpha} \leq f(z) - f(x). \end{aligned}$$

El límite es la definición de derivada de f en la dirección $z-x$. Como f es diferenciable, este es igual a $\langle \nabla f(x), z-x\rangle$, entonces:

$$\langle \nabla f(x), z-x\rangle \leq f(z)-f(x).$$

Como x, z son arbitrarios, esto prueba que $\forall x, z \in \mathbb{R}^n$ se tiene $f(z) \geq f(x)+\langle \nabla f(x), z-x\rangle$. ■

Ahora veremos que si la función es convexa y diferenciable, ser punto crítico es equivalente a ser mínimo global.

PROPOSICIÓN 3.1.2. *Sea $f : \mathbb{R}^n \to \mathbb{R}$ diferenciable y convexa con x^* mínimo local, entonces: $\nabla f(x^*) = 0$ si, y solo si, x^* es mínimo global.*

PRUEBA. ($\Leftarrow$) La prueba consiste en encontrar dos vectores opuestos para los que la derivada direccional en el punto x^* en cada una de esas dos direcciones da cero.

Sea $v \in \mathbb{R}^n$ cualquiera. Por un lado como x^* es mínimo local (porque es global), existe $\delta > 0$ tal que $\forall x \in B(x^*, \delta)$ se tiene $f(x) \geq f(x^*)$.
Entonces si tomamos un real $h \to 0$, de forma tal que $x^* + hv \in B(x^*, \delta)$, se va a tener $f(x^* + hv) \geq f(x^*)$, entonces:

$$\lim_{h\to 0^+} \frac{f(x^*+hv)-f(x^*)}{h} \geq 0 \Rightarrow \frac{\partial f(x^*)}{\partial v} \geq 0 \Rightarrow \langle \nabla f(x^*), v\rangle \geq 0,$$

donde en la última igualdad se usa que f es diferenciable, por lo tanto la derivada direccional con respecto v es el producto interno con el gradiente en el punto x^*. Del mismo modo, razonando igual pero sustituyendo v por $-v$ se obtiene $\langle \nabla f(x^*), v\rangle \leq 0$. Cumplir ambas desigualdades a la vez implica que el producto interno es cero, es decir, $\langle \nabla f(x^*), v\rangle = 0$, y esto para todo $v \in \mathbb{R}^n$, pues habíamos tomado v arbitrario.

El único vector con producto interno nulo con cualquier otro es el vector nulo, por lo tanto $\nabla f(x^*) = 0$.

($\Rightarrow$) Suponemos ahora que $\nabla f(x^*) = 0$, queremos ver que x^* es mínimo global. Como f es diferenciable y convexa, vale el Lema 3.1.1 tomando $x = x^*$, obteniendo:

$$\forall z \in \mathbb{R}^n, f(z) \geq f(x^*) + \langle \nabla f(x^*), z-x^*\rangle .$$

Ahora, por hipótesis $\nabla f(x^*) = 0$, entonces $\forall z \in \mathbb{R}^n, f(z) \geq f(x^*)$, es decir x^* es mínimo global. ■

Si el gradiente de f se conoce y se puede resolver la ecuación $\nabla f(x) = 0$, entonces, vía la proposición anterior, basta con evaluar f en los puntos del conjunto solución y quedarse con el mínimo, esto si f es convexa. En el caso no convexo, el mínimo absoluto de f podemos encontrarlo primero calculando los puntos críticos, descartando los que no sean mínimos relativos (para esto necesitamos poder calcular $\nabla^2 f(x)$ y sus valores propios en cada punto crítico x) y hallando el mínimo dentro de ese conjunto restante.

En otro contexto más general, donde no se pueda (o sea muy costoso) calcular ∇f, igual se puede utilizar la información brindada puntualmente por el gradiente para guiarnos

al mínimo y resolver el problema de forma numérica. En esta idea se basan los métodos descritos en la siguiente sección.

3.2. Métodos de gradiente

Una regla intuitiva para encontrar el mínimo de una función de manera numérica es iterativa y consiste en empezar en un punto inicial y, en cada paso, moverse en una dirección que haga disminuir el valor de la función. Formalmente, definimos una sucesión de puntos $\{x_k\}_{k\in\mathbb{N}}$ de forma tal que

$$x_{k+1} = x_k + \alpha_k d_k, \tag{5}$$

donde $\alpha_k \in \mathbb{R}$ es el paso y $d_k \in \mathbb{R}^n$ es la dirección de búsqueda.

Definición 3.2.1. Dado d_k tal que $\langle \nabla f(x_k), d_k\rangle < 0, \forall k \in \mathbb{N}$, decimos que el método definido por la ecuación (5) es un método de gradiente y que d_k es una dirección de descenso.

El nombre se debe a que en ese caso existe $\delta > 0$ tal que $f(x_k+\alpha d_k) < f(x_k), \forall \alpha \in (0, \delta)$. El pseudocódigo general para un método de gradiente es el siguiente:

Result: Sucesión x_k
Inicializar en x_0
Repetir:
    Elegir dirección d_k
    Elegir paso α_k
    Asignar $x_{k+1} = x_k + \alpha_k d_k$
Hasta que se cumpla un criterio de parada
return x_k

Algoritmo 1: Pseudocódigo para un método de gradiente.

Elección de la dirección d_k
Nos restringimos a elegir direcciones $d_k = -D_k \nabla f(x_k)$, donde D_k es una matriz simétrica definida positiva, de este modo:

$$\nabla f(x_k)^T D_k \nabla f(x_k) > 0 \Leftrightarrow \nabla f(x_k)^T(-D_k \nabla f(x_k)) < 0 \Leftrightarrow \nabla f(x_k)^T d_k < 0.$$

Así, definir la matriz D_k nos define automáticamente una dirección de descenso. Dentro de esta familia de direcciones, tomar $D_k = I$ da lugar al método de gradiente más simple definido a partir de la ecuación (5) como $x_{k+1} = x_k - \alpha_k \nabla f(x_k)$, allí α_k es una sucesión, cuyas alternativas para definirla veremos más adelante. Este método para definir las iteradas lo llamaremos descenso por gradiente a secas, pero también se le conoce como *steepest descent*.

Para lo anterior solo se requiere información de primer orden, lo cual es atractivo en contextos donde seguir derivando para obtener información de mayor orden es costoso, por ejemplo, en aprendizaje profundo.

Mal condicionamiento
Recordemos que si $f \in C^2$ la matriz Hessiana $H(\theta)$ en un punto θ del dominio está formada por las derivadas segundas de la f:

$$H(\theta) = \left(\frac{\partial^2 f}{\partial \theta_i \, \partial \theta_j} \right)_{i,j}.$$

El número de condición de la matriz $H(\theta)$ tiene que ver con la curvatura de la función en el punto θ y se define como:

$$\kappa(H(\theta)) = \frac{\lambda_{max}}{\lambda_{min}},$$

donde λ_{max} y λ_{min} son el valor propio máximo y mínimo de $H(\theta)$, respectivamente. Siempre $\lambda_{max} \geq \lambda_{min}$, por lo que $\kappa(H(\theta)) \geq 1$, si es mucho mayor que 1, decimos que la matriz $H(\theta)$ está mal condicionada.

Si durante la ejecución del algoritmo de descenso por gradiente, nos topamos con una región del dominio de la función donde la matriz Hessiana está mal condicionada, el algoritmo es menos eficiente, puede tomar una mayor cantidad de pasos en terminar que si fuera la matriz no estuviera mal condicionada (ver Figura 1).

Una modificación conocida al descenso por gradiente usual, que intenta aplacar el problema del mal condicionamiento, es el *método de Newton*, que consiste en tomar $D_k = (\nabla^2 f(x_k))^{-1}$ en la ecuación (5). Este método converge rápidamente (cerca del punto crítico), pero se paga el costo computacional de calcular la Hessiana y su inversa. Ahora vemos cómo este método ayuda a contrarrestar ese efecto que causa la curvatura de la función a través de un ejemplo.

Ejemplo 3.2.1. **Método de Newton aplicado en una forma cuadrática.**
Consideramos una función cuadrática 'estirada' en un eje, definida por $f(x, y) = \frac{a}{2}x^2 + \frac{b}{2}y^2$ con $a \gg b > 0$. Entonces la matriz Hessiana en un punto (x, y) cualquiera es la matriz constante:

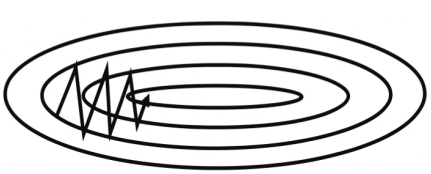
$$H = \begin{bmatrix} a & 0 \\ 0 & b \end{bmatrix} = \nabla f(x, y).$$

La función crece mucho más en la dirección del eje x que en la del eje y, y las curvas de nivel son elipses. Esto es un problema para el algoritmo de descenso por gradiente porque desvía al vector $-\nabla f(x, y)$ del mínimo, que es el $(0, 0)$. El número de condición $\kappa(H) = \frac{a}{b} \gg 1$ captura esta situación.

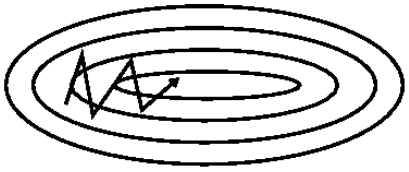
Al multiplicar al gradiente por $H(x, y)^{-1}$, corregimos por la curvatura de la función f. En al Figura 1 ilustramos cómo puede dar lugar a un camino más directo al mínimo.

En contrapartida, si consideramos a $g(x, y) = x^2 + y^2$, entonces las curvas de nivel son circunferencias y en todo punto el vector $-\nabla g(x, y)$ apunta al mínimo global, entonces vamos a arribar al mínimo mucho más rápido. Incluso, si usamos el paso adecuado (ver

método de *line search* más abajo), encontramos el mínimo en un paso. En el caso de g, el número de condición de $\nabla^2 g(x, y)$ es exactamente 1 en todo punto.



Trayectoria de los puntos $\{x_k\}$ usando el descenso por gradiente



Trayectoria de los puntos $\{x_k\}$ usando el método de Newton

FIGURA 1. Curvas de nivel de la función $f(x, y) = ax^2 + by^2$ con $a \gg b$. La matriz $\nabla^2 f(x, y)$ está mal condicionada.

Elección del paso α_k

Una vez definida la dirección de descenso, existen diferentes alternativas para elegir el paso. A continuación describimos algunas de ellas:

1. **Line search:**
Consiste en resolver el problema de optimización unidimensional $\min_{\alpha>0} f(x_k + \alpha d_k)$, es decir, encontrar el paso que minimiza la función en la semirrecta que parte de x_k en la dirección d_k. Esto se puede hacer analíticamente cuando es posible o con un método numérico aplicado a $\mathbb{R}$.

2. **Limited line search:**
Es igual al line search, pero acotamos el espacio de búsqueda del α a un intervalo $[0, s]$, donde $s \in \mathbb{R}$ es un parámetro a definir.

3. **Regla de Armijo:**
Las dos propuestas anteriores implican resolver un problema de optimización unidimensional en cada paso del método, lo cual puede resultar en un costo computacional acumulado muy alto. La regla de Armijo es una alternativa en la que también se busca un paso dentro de la dirección d_k, pero no necesariamente el óptimo, sino que se hace a base de ensayo y error. La forma más simple de hacer esto es empezar probando con cierto valor de α y evaluar si se cumple que $f(x_k + \alpha d_k) < f(x_k)$, en caso contrario reducir el α multiplicando por algún factor menor a 1. Este método es natural, pero no siempre se puede encontrar tal α (ver Figura 1.2.6 en [8]).

La regla de Armijo funciona bajo ese principio, pero se pide una condición particular que en definitiva se traduce en que el decrecimiento en cada paso sea sustancial; se definen $s, \beta, \theta \in \mathbb{R}$ con $\beta, s > 0$, $\theta < 1$ y se toma como paso $\alpha = \beta^m s$, donde $m \in \mathbb{N}$ es el menor natural que cumple:

$$f(x_k) - f(x_k + \beta^m s d_k) \geq -\theta \beta^m s \nabla f(x_k)^T d_k. \tag{6}$$

Notar que la cantidad en la parte derecha de la desigualdad es positiva si la dirección d_k es de descenso. De (6) se desprende que siguiendo la regla de Armijo obtenemos un método de descenso (el valor funcional disminuye en cada paso), veremos más adelante además que en caso de que el algoritmo converja, lo hace a un punto crítico.

4. **Paso fijo:**
Este es el método más simple de todos, donde sencillamente se elige $\alpha_k = \alpha$ para todo $k \in \mathbb{N}$, donde α es un valor fijo. El algoritmo producido de esta manera no necesariamente es de descenso.
5. **Paso decreciente:**
Aquí se pide que $\alpha_k \to 0$. El método no es necesariamente de descenso, además, tiene el peligro de estancarse rápidamente si el paso se hace muy pequeño. Para impedir esto último, se impone que $\sum_{k=0}^{\infty} \alpha_k = \infty$, es decir, pedimos que el paso disminuya pero a una tasa lenta, de manera que su serie diverja.

Condición de parada
Los algoritmos iterativos no tienen por qué alcanzar exactamente el mínimo. Hay distintos criterios para definir que ya alcanzamos un punto lo suficientemente bueno. Presentamos los mas usuales:

1. Criterio de tolerancia en el gradiente: $\|\nabla f(x_k)\| \leq \epsilon$.
2. Criterio de tolerancia en el valor funcional: $|f(x_k) - f(x_{k-1})| \leq \epsilon$.
3. Criterio de tolerancia en los puntos: $\|x_k - x_{k-1}\| \leq \epsilon$.

3.2.1. Resultados de convergencia.

En esta sección estudiamos condiciones para que un método de gradiente converja a un punto crítico. Los resultados y definiciones se encuentran en [8]. Empezamos dando una definición técnica de una propiedad que debe cumplir la dirección d_k en algunos teoremas que veremos:

Definición 3.2.2. Decimos que la sucesión de direcciones $\{d_k\}$ es relacionada por gradiente a $\{x_k\}$ si para toda subsucesión $\{x_k\}_{k\in I}$ que converge a un punto no crítico, se tiene que la subsucesión de direcciones $\{d_k\}_{k\in I}$ correspondiente tiene norma acotada y además satisface que:

$$\limsup_{k\to\infty, k\in I} \langle \nabla f(x_k), d_k \rangle < 0. \tag{7}$$

El siguiente es un resultado sobre la convergencia de una sucesión generada según la regla de Armijo.

Teorema 3.2.1. *Sea $f : \mathbb{R}^n \to \mathbb{R}$ diferenciable, $\{x_k\}$ una sucesión generada por un método de gradiente $x_{k+1} = x_k + \alpha_k d_k$, donde $\{d_k\}$ es relacionada por gradiente a $\{x_k\}$ y α_k está dado por la regla de Armijo. Entonces todo punto de aglomeración de $\{x_k\}$ es un punto crítico de f.*

Prueba. Supongamos que x^* es un punto de aglomeración de $\{x_k\}$, es decir que existe una subsucesión $\{x_k\}_{k\in I}$ tal que $\lim_{k\to\infty, k\in I} x_k = x^*$, queremos ver que x^* es

punto crítico. Supongamos que no lo es, o sea $\nabla f(x^*) \neq 0$.
Por continuidad de la f tenemos que $f(x^*) = \lim_{k\to\infty, k\in I} f(x_k)$, por lo tanto:

$$\lim_{k\to\infty, k\in I} f(x_k) - f(x_k + \alpha_k d_k) = 0.$$

Ahora, por regla de Armijo existen β, σ, s con $\beta, s > 0$, $\theta < 1$ tales que $\alpha_k = \beta^{m_k} s$, siendo m_k el menor natural tal que

$$f(x_k) - f(x_k + \beta^{m_k} s d_k) \geq -\sigma \beta^{m_k} s \nabla f(x_k)^T d_k.$$

Entonces también $\alpha_k \nabla f(x_k)^T d_k \to 0$ y como el producto interno es acotado, ya que d_k está acotado por ser relacionada por gradiente con $\{x_k\}$, tenemos $\alpha_k \to 0$ si $k \in I$. Notemos que para cada k, por definición del m_k, se tiene:

$$\begin{aligned} & f(x_k) - f(x_k + \beta^{m_k - 1} s d_k) < -\sigma \beta^{m_k - 1} s \nabla f(x_k)^T d_k \\ \Leftrightarrow \quad & f(x_k) - f(x_k + \frac{\alpha_k}{\beta} d_k) < -\sigma \frac{\alpha_k}{\beta} \nabla f(x_k)^T d_k, \end{aligned} \tag{8}$$

lo anterior vale para todo k, en particular para todo $k \in I$. El resto de la prueba consiste en verificar la dirección p tal que $\nabla f(x_k)^T p > 0$, para arribar a una contradicción. Empezamos definiendo

$$p_k = \frac{d_k}{\|d_k\|}, \qquad \overline{\alpha}_k = \alpha_k \frac{\|d_k\|}{\beta},$$

de donde:

$$d_k = \|d_k\| \, p_k, \qquad \alpha_k = \overline{\alpha}_k \frac{\beta}{\|d_k\|}.$$

Sustituyendo en (8) nos queda:

$$f(x_k) - f(x_k + \overline{\alpha}_k p_k) < -\sigma \overline{\alpha}_k \nabla f(x_k)^T p_k.$$

Por lo tanto

$$\frac{f(x_k) - f(x_k + \overline{\alpha}_k p_k)}{\overline{\alpha_k}} < -\sigma \nabla f(x_k)^T p_k. \tag{9}$$

El término de la izquierda es un cociente incremental y se relaciona directamente con ∇f. Consideramos la función $g : [0, 1] \to \mathbb{R}$ dada por $g(t) = f(x_k + t p_k)$, por el teorema del valor medio, existe $\hat{\alpha}_k \in [0, \overline{\alpha}_k]$ tal que:

$$f(x_k + \overline{\alpha}_k p_k) - f(x_k) = g'(\hat{\alpha}_k) \overline{\alpha}_k.$$

Por regla de la cadena es $g'(\hat{\alpha}_k) = \nabla f(x_k + \hat{\alpha}_k p_k)^T p_k$, si además multiplicamos ambos lados de la igualdad por -1 obtenemos:

$$f(x_k) - f(x_k + \overline{\alpha}_k p_k) = -\nabla f(x_k + \hat{\alpha}_k p_k)^T p_k \overline{\alpha}_k.$$

Sustituyendo en (9) nos queda:

$$-\nabla f(x_k + \hat{\alpha}_k p_k)^T p_k < -\sigma \nabla f(x_k)^T p_k. \tag{10}$$

Por otro lado, como $\|p_k\| = 1$, tenemos que $\{p_k\}_{k\in I}$ es acotada por lo tanto tiene una subsucesión convergente, es decir: existe $J \subset I \subset \mathbb{N}$ tal que $\{p_k\} \to \overline{p}_k$.

A su vez, puesto que $\|d_k\|$ es acotado y $\beta < 1$, vale el siguiente límite:

$$\overline{\alpha}_k = \alpha_k \frac{\|d_k\|}{\beta} = \beta^{m_k - 1} \|d_k\| \to_{k \in I} 0.$$

Como $\hat{\alpha_k} \in [0, \overline{\alpha_k}]$ tenemos que $\hat{\alpha_k} \to 0$ si $k \in I$. Entonces $\hat{\alpha_k} \to 0$ si $k \in J$.
Tomando límite en (10) con $k \in J$, utilizando que tanto f como el gradiente y el producto interno son funciones continuas, obtenemos:

$$-\nabla f(\overline{x})^T \overline{p} < -\sigma \nabla f(\overline{x})^T \overline{p} \Leftrightarrow 0 < (1-\sigma) \nabla f(\overline{x})^T \Leftrightarrow \nabla f(\overline{x})^T \overline{p} > 0. \tag{11}$$

En la última implicancia utilizamos que $\sigma < 1$. Ahora, por otro lado, por definición de p_k tenemos

$$\nabla f(x_k)^T p_k = \frac{\nabla f(x_k)^T d_k}{\|d_k\|}.$$

Si tomamos límite en la expresión anterior, llegamos a una contradicción usando que d_k está relacionada por gradiente con x_k:

$$\begin{aligned}
\nabla f(\overline{x})^T \overline{p} &= \lim \nabla f(x_k)^T p_k \\
&= \lim \frac{\nabla f(x_k)^T d_k}{\|d_k\|} \\
&\leq \limsup \frac{\nabla f(x_k)^T d_k}{\|d_k\|} \\
&\leq \frac{\limsup \nabla f(x_k)^T d_k}{\limsup \|d_k\|} \\
&< 0.
\end{aligned}$$

Esto es absurdo porque contradice (11). El absurdo proviene de suponer que existe un punto de aglomeración de $\{x_k\}$, a saber, el x^*, que no es punto crítico. ■

El resultado anterior se puede extender a los métodos de *line search* y de *limited line search*, y a cualquier regla que defina un paso con una reducción en el valor funcional en cada iteración menor al que se consigue con la regla de Armijo [8].

A continuación estudiamos el caso del paso fijo, que si bien no es un método de descenso, imponiendo que el gradiente de f sea Lipschitz, junto con otras condiciones, podemos garantizar convergencia a un punto que cumpla la condición de optimalidad. Previo a esto necesitamos del siguiente lema:

LEMMA 3.2.2. ***(Descent Lemma)*** *Sea $f : \mathbb{R}^n \to \mathbb{R}$ diferenciable. Sean $x, y \in \mathbb{R}^n$, $L \in \mathbb{R}^+$ tales que*

$$\|\nabla f(x + ty) - \nabla f(y)\| \leq Lt \|y\|, \text{ para todo } t \in [0, 1].$$

Entonces se cumple:

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

PRUEBA. Definimos $g(t) = f(x+ty)$, entonces $g'(t) = \langle y, \nabla f(x+ty)\rangle$. La siguiente cadena de desigualdades demuestra la tesis:

$$\begin{aligned} f(x+y) - f(x) =& g(1) - g(0) = \int_0^1 g'(t)dt = \int_0^1 \langle y, \nabla f(x+ty)\rangle \, dt \\ &\overset{(1)}{=} \int_0^1 \langle y, \nabla f(x)\rangle \, dt + \int_0^1 \langle y, \nabla f(x+ty) - \nabla f(x)\rangle \, dt \\ &\overset{(2)}{\leq} \int_0^1 \langle y, \nabla f(x)\rangle \, dt + \int_0^1 |\langle y, \nabla f(x+ty) - \nabla f(x)\rangle| dt \\ &\overset{(3)}{\leq} \langle y, \nabla f(x)\rangle + \int_0^1 \|y\| \, \|\nabla f(x+ty) - \nabla f(x)\| \, dt \\ &\overset{(4)}{\leq} \langle y, \nabla f(x)\rangle + \|y\| \int_0^1 \|y\| \, Ltdt \\ &= \langle y, \nabla f(x)\rangle + \frac{L}{2} \|y\|^2 . \end{aligned}$$

En (1) sumamos y restamos $\nabla f(x)$ y utilizamos la linealidad del producto interno y la integral; en (2) aplicamos que $\int_a^b h(t)dt \leq |\int_a^b h(t)|dt$ junto con la desigualdad triangular; en (3) usamos la desigualdad de Cauchy-Schwarz y que $\int_0^1 \langle y, \nabla f(x)\rangle \, dt = \langle y, \nabla f(x)\rangle$ y en (4) aplicamos la hipótesis. En la última desigualdad calculamos la integral $\int_0^1 tdt = \frac{1}{2}$. ■

Ahora sí, el teorema referente al método de paso fijo:

TEOREMA 3.2.3. *Sea $\{x_k\}$ una sucesión generada por un método de gradiente dado por: $x_{k+1} = x_k + \alpha_k d_k$, con $\alpha_k = \alpha$ fijo para todo k y d_k relacionado por gradiente a x_k. Supongamos también que ∇f es Lipschitz de constante L, además, que para todo k se cumple que $d_k \neq 0$ y que existe ϵ tal que $\epsilon \leq \alpha_k \leq (2-\epsilon)\frac{|\nabla f(x_k)^T d_k|}{L\|d_k\|^2}$.*
Entonces se cumple que todo punto de aglomeración de $\{x_k\}$ es un punto crítico de f.

PRUEBA. Por hipótesis existe $L > 0$ tal que para todo $x, y \in \mathbb{R}^n$:

$$\|\nabla f(x) - \nabla f(y)\| \leq L \|x - y\| .$$

Si escribimos $x = x_k + \alpha_k^2 d_k$ y $y = x_k$ nos queda, para todo k, que:

$$\left\|\nabla f(x_k + \alpha_k^2 d_k) - \nabla f(x_k)\right\| \leq L \left\|\alpha_k^2 d_k\right\| \Rightarrow \|\nabla f(x_k + \alpha_k(\alpha_k d_k)) - \nabla f(x_k)\| \leq \alpha_k L \|\alpha_k d_k\| ,$$

lo que nos permite usar el *descent lemma* para $x = x_k$ e $y = \alpha_k d_k$, obteniendo:

$$\begin{aligned} & f(x_k + \alpha_k d_k) \leq f(x_k) + \langle \nabla f(x_k), \alpha_k d_k\rangle + \frac{L}{2} \|\alpha_k d_k\|^2 \\ \Rightarrow\ & f(x_k + \alpha_k d_k) - f(x_k) \leq \alpha_k (\nabla f(x_k)^T d_k) + \frac{L}{2}\alpha_k^2 \|d_k\|^2 \\ \Rightarrow\ & f(x_k + \alpha_k d_k) - f(x_k) \leq \alpha_k \left(\frac{L}{2}\alpha_k \|d_k\|^2 - |\nabla f(x_k)^T d_k|\right) . \end{aligned}$$

En la última implicancia usamos que $\nabla f(x_k)^T d_k < 0$, por lo que es igual al opuesto de su valor absoluto. Por hipótesis $\epsilon \leq \alpha_k \leq (2-\epsilon)\frac{|\nabla f(x_k)^T d_k|}{L\|d_k\|^2}$, sustituyendo α_k en el segundo factor del miembro derecho de la desigualdad, obtenemos:

$$\begin{aligned}\frac{L}{2}\alpha_k \|d_k\|^2 - |\nabla f(x_k)^T d_k| &\leq \frac{\|d_k\|^2 L}{2}(2-\epsilon)\frac{|\nabla f(x_k)^T d_k|}{L\|d_k\|^2} - |\nabla f(x_k)^T d_k| \\ &= \left(1-\frac{\epsilon}{2}\right)|\nabla f(x_k)^T d_k| - |\nabla f(x_k)^T d_k| \\ &= -\frac{\epsilon}{2}|\nabla f(x_k)^T d_k|.\end{aligned}$$

Por lo tanto:

$$\frac{L}{2}\alpha_k \|d_k\|^2 - |\nabla f(x_k)^T d_k| \leq -\frac{\epsilon}{2}|\nabla f(x_k)^T d_k|.$$

Volviendo a la desigualdad inicial nos queda:

$$\begin{aligned}f(x_k+\alpha_k d_k) - f(x_k) &\leq \alpha_k(-\frac{\epsilon}{2}|\nabla f(x_k)^T d_k|) \\ (\Rightarrow) f(x_k) - f(x_k+\alpha_k d_k) &\geq \frac{\epsilon}{2}\alpha_k|\nabla f(x_k)^T d_k| \\ &\geq -\frac{\epsilon^2}{2}|\nabla f(x_k)^T d_k|.\end{aligned}$$

Para la última desigualdad usamos que $\alpha_k \geq \epsilon$. Ahora estamos en condiciones de probar lo que queremos: partimos de un punto p de aglomeración de $\{x_k\}$, entonces hay una subsucesión $\{x_{k_j}\}$ que converge a p y queremos probar que p es punto crítico. Supongamos por absurdo que p no lo fuera, como f es continua, también tenemos $f(x_{k_j}) \to f(p)$, entonces

$$f(x_{k_j}) - f(x_{k_j} + \alpha_{k_j} d_{k_j}) \to f(p) - f(p) = 0,$$

y usando la desigualdad que probamos antes, junto con la continuidad del gradiente se tiene que:

$$\frac{\epsilon^2}{2}|\nabla f(x_{k_j})^T d_{k_j}| \to 0 \Rightarrow |\nabla f(x_{k_j})^T d_{k_j}| \to 0.$$

Esto último no puede ser, ya que $\limsup \nabla f(x_{k_j})^T d_{k_j} < 0$, por ser d_k relacionado por gradiente, que vale porque supusimos por absurdo que p no es punto crítico.

Entonces p es un punto crítico de f. Elegimos p arbitrario, así que probamos que todo punto de aglomeración de $\{x_k\}$ es un punto crítico de f. ■

También existe un resultado del mismo tipo para el caso del paso decreciente, que utiliza técnicas parecidas a las de los teoremas anteriores. El lector interesado puede ver el enunciado y su prueba en el Apéndice A.

3.3. Descenso por gradiente estocástico

El descenso por gradiente estocástico (SGD, por sus siglas en inglés) es una variante de descenso por gradiente que cobra relevancia cuando no tenemos acceso al gradiente exacto de la función a optimizar, o no es eficiente calcularlo, y en su lugar decidimos estimarlo. Este algoritmo y sus variantes son muy utilizadas en en toda el área del aprendizaje automático. La referencia principal para esta sección es el libro 'Optimization for Modern Data Analysis' [21], Capítulos 2, 3 y 5.

3.3.1. Concepto general y ejemplos.

La situación es similar a la que teníamos cuando empezamos a hablar de optimización, se tiene una función diferenciable $f : \mathbb{R}^n \to \mathbb{R}$ que queremos minimizar. La diferencia es que en lugar del valor exacto de $\nabla f(x)$, tenemos un valor estimado $g(x, \xi) \in \mathbb{R}$, donde ξ es una variable aleatoria de distribución P que toma valores en cierto espacio Ξ, por ejemplo, $\Xi = I \subseteq \mathbb{N}$, que podrían ser índices de un conjunto indexado de datos de entrenamiento en un problema de aprendizaje automático.

Lo que se exige es que $g(x, \xi)$ sea un estimador insesgado de $\nabla f(x)$, es decir, que

$$\mathbb{E}[g(x, \xi)] = \nabla f(x). \tag{12}$$

El método de SGD es igual al de descenso por gradiente clásico, pero sustituyendo el gradiente por el estimador. La regla de actualización queda:

$$x_{k+1} = x_k - \alpha_k g(x_k, \xi_k). \tag{13}$$

Ejemplo 3.3.1. **Gradiente con ruido**
Un caso simple de SGD se da cuando el estimador $g(x, \xi)$ es el valor real del gradiente más un ruido:

$$g(x, \xi) = \nabla f(x) + \xi, \tag{14}$$

donde ξ es una variable aleatoria conocida, por ejemplo una normal. Cumplir con la condición (12) implica que $\mathbb{E}(\xi) = 0$

El siguiente ejemplo es la base para el uso del SGD en el aprendizaje automático.

Ejemplo 3.3.2. **Método de gradiente incremental**
Sea f una suma finita de la forma

$$f(x) = \frac{1}{n} \sum_{i=1}^{n} f_i(x). \tag{15}$$

Entonces, por linealidad del gradiente se tiene

$$\nabla f(x) = \frac{1}{n} \sum_{i=1}^{n} \nabla f_i(x). \tag{16}$$

Como se ve, calcular ∇f implica el cálculo de n gradientes ∇f_i. Esto puede ser costoso si n es grande. En su lugar, el método de gradiente incremental es un método iterativo

que propone, en cada paso k, seleccionar un índice i_k al azar, de acuerdo a una variable aleatoria ξ_k, y usar solo la información del índice i en la regla de actualización:

$$x_{k+1} = x_k - \alpha_k \nabla f_{i_k}(x_k). \tag{17}$$

Aquí entonces ξ_k toma valores en el espacio $\Xi = \{1, \ldots, n\}$ y $g(x, \xi) = \nabla f_\xi(x) = \nabla f_{i_k}(x)$. La distribución es en principio la uniforme, o sea $P(\xi_k = i) = P(i) = \frac{1}{n}$ para todo $i \in \{1, \ldots, n\}$. Este estimador cumple con la condición (12), ya que por definición de esperanza:

$$\mathbb{E}_\xi(g(x, \xi)) = \frac{1}{n} \sum_{i=1}^{n} \nabla f_i(x) = \nabla f(x). \tag{18}$$

3.3.1.1. *Riesgo empírico.* Nos desviamos brevemente del SGD en abstracto para entender por qué el ejemplo anterior engloba a la optimización en la mayoría de los problemas de aprendizaje automático. En un problema de clasificación o regresión, la eficacia del modelo se mide calculando el error esperado sobre un conjunto de datos, donde ese error se calcula con una función de costo $L(u, v)$.

Si $p(x, y)$ es la distribución que genera los datos y $m(x)$ la salida generada por el modelo para un dato x, el error esperado es:

$$R[m] = \mathbb{E}_p[L(m(x), y)]. \tag{19}$$

El objetivo es minimizar este error. En el caso donde m depende de ciertos parámetros θ, el problema de optimización es encontrar $\theta^* = \operatorname{argmin}_\theta R[m(\theta)]$. Encontrar este mínimo puede ser muy costoso, además, depende de conocer la distribución $p(x, y)$, lo que en la práctica no se conoce. Lo que se hace es tomar una muestra de datos $\{(x_i, y_i)\}_{i=1}^n$ que sea *i.i.d.* con la que se calcula el riesgo empírico:

$$R_{emp}[m] = \frac{1}{n} \sum_{i=1}^{n} L(m(x_i), y_i). \tag{20}$$

Cambiamos el objetivo inicial de minimizar $R[m]$ por minimizar $R_{emp}[m]$. Si la muestra es representativa del problema a resolver, minimizar $R_{emp}[m]$ se acerca a minimizar $R[m]$. La ecuación (20) se puede formular como una instancia de gradiente incremental, con $f_i(x) = L(m(x_i), y_i)$.

3.3.2. Convergencia del SGD.

Supongamos que tenemos la función $f : \mathbb{R}^n \to \mathbb{R}$ diferenciable que se minimiza en un x^*, que no conocemos y queremos aplicar el algoritmo de SGD para hallarlo. Hay tres hipótesis importantes necesarias para la convergencia del algoritmo a ese mínimo:

H1: $\mathbb{E}[g(x, \xi)] = \nabla f(x)$.

H2: Existen $L, B \geq 0$ t.q. para $x \in \mathbb{R}^n$ se cumple

$$\mathbb{E}[\|g(x, \xi)\|^2] \leq L \|x - x^*\|^2 + B^2.$$

H3: La sucesión de v.a $\{\xi_k\}$ es independiente e idénticamente distribuida.

H1 es la condición (12) que ya exigíamos al principio y **H3** en el contexto del aprendizaje automático es que la muestra de datos sea tomada de manera *i.i.d.* El análisis sustancial radica en entender cuándo se cumple **H2**, que es una cota para el valor esperado para el estimador del gradiente, en función de los valores L y B y de la distancia al mínimo. A continuación, vemos dos ejemplos importantes donde las hipótesis se cumplen y determinamos qué valores toman L y B en cada caso.

Ejemplo 3.3.3. **Clasificación binaria con regresión logística**
Consideramos un problema de clasificación binaria en el que cada observación i está compuesta por un predictor x_i (vector) y la variable de respuesta $y_i \in \{0, 1\}$.
Para este ejemplo usamos la regresión logística, que introduciremos en detalle en el Capítulo 4, primera sección. Consideramos una muestra $\{(x_i, y_i)\}_{i=1,\dots,n}$, la función a minimizar es la entropía cruzada, cuya formula derivamos de diferentes maneras en la Sección 4.1.1 y es:

$$f(w) = \frac{1}{n}\sum_{i=1}^{n} -y_i \log \sigma(w^T x_i) - (1 - y_i)\log(1 - \sigma(w^T x_i)). \tag{21}$$

El vector $w \in \mathbb{R}^n$ representa a los parámetros de la regresión logística, donde, para simplificar, no estamos considerando el término b de sesgo. Lo anterior se puede formular como un caso de gradiente incremental con $f_i(w) = -y_i \log \sigma(w^T x_i) - (1 - y_i)\log(1 - \sigma(w^T x_i))$, pensando nuevamente que hay una sucesión de v.a. ξ_k subyacente que toma valores en los índices $\{1, \dots, n\}$ de manera independiente con una distribución uniforme (se cumple H3) y el estimador del gradiente es $g(w, \xi_k) = \nabla f_{i_k}$.
Escribimos $x_i = (x_i^1, \dots, x_i^p)$. Queremos calcular el gradiente de cada f_i, ahora, en la Sección 4.1.2 vemos que las derivadas direccionales de las f_i son de la forma:

$$\frac{\partial f_i}{\partial w^j}(w) = (\sigma(w^T x) - y_i)x_i^j. \tag{22}$$

Esta es una derivada direccional, solo una entrada del vector gradiente. Calculando cada entrada nos queda $\nabla f_i(w) = ((\sigma(w^T x) - y_i)x_i^j)_{j=1,\dots,p}$. Para estimar ∇f según el método de gradiente incremental, utilizamos la v.a. ξ para sortear un índice i de manera que el gradiente estocástico g queda definido como:

$$g(w, i) = \nabla f_i(w) = ((\sigma(w^T x_i) - y_i)x_i^j)_{j=1,\dots,p}.$$

Se cumple H1, porque $\mathbb{E}[g(w, \xi)] = \nabla f(w)$. Nos queda la hipótesis H2, empecemos por calcular la esperanza de $\|g(w, \xi)\|^2$, se tiene:

$$\|g(w, i)\|^2 = \sum_{i=1}^{p}(\sigma(w^T x_i) - y_i)^2(x_i^j)^2 < \sum_{i=1}^{p}(x_i^j)^2 = \|x_i\|^2 .$$

Donde usamos que $(\sigma(w^T x_i) - y_i)^2 < 1$ para todo i, esto porque $\sigma : \mathbb{R} \to (0, 1)$ e $y_i \in \{0, 1\}$. Entonces, si $M = \operatorname{argmax}_i \|g(w, i)\|^2$, usando la desigualdad anterior, la esperanza $\mathbb{E}[\|g(w, i)\|^2]$ tiene que ser menor a $\|x_M\|^2$:

$$\mathbb{E}[\|g(w,i)\|^2] = \frac{1}{n}\sum_{i=1}^{n}\|g(w,i)\|^2 \leq \max_i \|g(w,i)\|^2 < \|x_M\|^2 .$$

Por lo que se cumple H2 para $B = \|x_M\|$ y $L = 0$.

El ejemplo anterior es un caso particular de gradiente incremental usando la regresión logística, veamos ahora el caso general.

Ejemplo 3.3.4. **Caso general de gradiente incremental**
Sea $f(x) = \sum_{i=1}^{n} f_i(x)$ donde además las f_i cumplen que ∇f_i son Lipschitz de constante L_i. Como antes, ξ es una v.a. aleatoria que toma valores en $\{1,\ldots,n\}$ y el estimador de $\nabla f(x)$ es $g(x,\xi) = \nabla f_\xi(x)$. Las hipótesis H1 y H3 se cumplen por ser este un caso de gradiente incremental, nos enfocamos en H2.

Definimos $(x^*)^i$ como un punto crítico para f_i, o sea son puntos tales que $\nabla f_i((x^*)^i) = 0$. Dado x, usando la condición de Lipschitz de ∇f_i para x y $(x^*)^i$ se tiene:

$$\|\nabla f_i(x)\| \leq L_i \left\|x - (x^*)^i\right\| .$$

Usando esto junto con la definición de $g(x,\xi)$ obtenemos:

$$\mathbb{E}\left[\|g(x,\xi)\|^2\right] = \mathbb{E}\left[\|\nabla f_i(x)\|^2\right] \leq \mathbb{E}\left[L_i^2\left\|x - (x^*)^i\right\|^2\right]. \tag{23}$$

Donde la aleatoreidad viene del sorteo del índice i dada por la v.a. ξ. Por otro lado, usando que dados $a, b \in \mathbb{R}^n$ se cumple $\|a+b\|^2 \leq 2\|a\|^2 + 2\|b\|^2$ tenemos que:

$$\left\|x - (x^*)^i\right\|^2 = \left\|x - x^* + x^* - (x^*)^i\right\|^2 \leq 2\|x - x^*\|^2 + 2\left\|x^* - (x^*)^i\right\|^2 . \tag{24}$$

Y usando la monotonía y linealidad de la esperanza obtenemos:

$$\begin{aligned}\mathbb{E}[L_i^2\left\|x - (x^*)^i\right\|^2] &\leq \mathbb{E}[L_i^2(2\|x - x^*\|^2 + 2\left\|x^* - (x^*)^i\right\|^2)] \\ &= \mathbb{E}[2L_i^2\|x - x^*\|^2] + \mathbb{E}[2L_i^2\left\|x^* - (x^*)^i\right\|^2] \\ &= \frac{2}{n}\sum_{i=1}^{n} L_i^2\|x - x^*\|^2 + \frac{2}{n}\sum_{i=1}^{n} L_i^2\left\|x^* - (x^*)^i\right\|^2 .\end{aligned} \tag{25}$$

Combinando esto último con la primera ecuación obtenemos:

$$\mathbb{E}[\|g(x,i)\|^2] \leq (\frac{2}{n}\sum_{i=1}^{n} L_i^2)\|x - x^*\|^2 + \frac{2}{n}\sum_{i=1}^{n} L_i^2\left\|x^* - (x^*)^i\right\|^2 . \tag{26}$$

Entonces definiendo $L = \frac{2}{n}\sum_{i=1}^{n} L_i^2$ y $B = \sqrt{\frac{2}{n}\sum_{i=1}^{n} L_i^2\|x^* - (x^*)^i\|^2}$ se tiene

$$\mathbb{E}[\|g(x,i)\|^2] \leq L\|x - x^*\|^2 + B^2,$$

y se cumple H2.

3.3.2.1. *Análisis de la convergencia del SGD.*
Ya vimos dos ejemplos donde se cumplen las hipótesis H1, H2 y H3. Estas junto con otras hipótesis servirán para responder dos preguntas de importancia para el uso del SGD en los ejemplos anteriores, estas son:

- ¿Qué tan cerca del óptimo nos deja el algoritmo?
- ¿Cómo tiene que ser la sucesión de pasos α_k?

Para responderlas, analizamos la distancia entre el punto x_k en el paso k del SGD y el óptimo x^*. Dado que el algoritmo es aleatorio lo que vamos a medir es una esperanza para entender qué tan bueno es *en promedio*. Hay dos formas naturales de hacerlo: medir el error en el dominio, esto es $r = \mathbb{E}[\|x_k - x^*\|]$ o medirlo en los valores funcionales, es decir $r = \mathbb{E}[|f(x_k) - f(x^*)|]$. Vamos a concentrarnos ahora en la primera. Denotamos $a_k = \mathbb{E}[\|x_k - x^*\|^2]$, que sería el error cuadrático medio en el paso k.

$$\begin{aligned} \|x_{k+1} - x^*\|^2 &= \|x_k - \alpha_k g(x_k, \xi_k) - x^*\|^2 \\ &= \langle x_k - \alpha_k g(x_k, \xi_k) - x^*, x_k - \alpha_k g(x_k, \xi_k) - x^* \rangle \\ &= \|x_k - x^*\|^2 - 2\alpha_k \langle x_k - x^*, \alpha_k g(x_k, \xi_k) \rangle + \alpha_k^2 \|g(x_k, \xi_k)\|^2 . \end{aligned} \tag{27}$$

Tomando esperanza de ambos lados, obtenemos una ecuación en recurrencia para a_k:

$$a_{k+1} = a_k - 2\alpha_k \mathbb{E}[\langle x_k - x^*, g(x_k, \xi_k) \rangle] + \alpha_k^2 \mathbb{E}[\|g(x_k, \xi_k)\|^2]. \tag{28}$$

Ahora operamos un poco más sobre las esperanzas que ahí aparecen. En primer lugar, el valor $\langle x_k - x^*, g(x_k, \xi_k) \rangle$ depende del valor que tome ξ_k y de x_k, este a su vez depende de todas las v.a. anteriores $\xi_1, \ldots, \xi_{k-1}$, teniendo esto en cuenta, podemos reescribir la esperanza usando la ley de la esperanza iterada (LEI) que nos dice que dadas dos v.a. X e Y, se tiene que $\mathbb{E}[X] = \mathbb{E}[\mathbb{E}[X|Y]])$. En nuestro caso tenemos $X = \xi_k$ e $Y = (\xi_1, \ldots, \xi_{k-1})$.

$$\begin{aligned} \mathbb{E}[\langle x_k - x^*, g(x_k, \xi_k) \rangle] &= \mathbb{E}[\mathbb{E}_{\xi_k}[\langle x_k - x^*, g(x_k, \xi_k) \rangle \, | \xi_1, \ldots, \xi_{k-1}]] \\ &= \mathbb{E}[\langle \mathbb{E}_{\xi_k}[g(x_k, \xi_k) | \xi_1, \ldots, \xi_{k-1}], x_k - x^* \rangle] \\ &= \mathbb{E}[\langle \mathbb{E}[g(x_k, \xi)], x_k - x^* \rangle] \\ &= \mathbb{E}[\langle \nabla f(x_k), x_k - x^* \rangle]. \end{aligned} \tag{29}$$

Para la primera igualdad usamos la LEI, tomando esperanza solo con respecto a la variable ξ_k condicionada a las v.a. anteriores, en la segunda utilizamos la linealidad de la esperanza, en la tercera la LEI y por último que $g(x_k, \xi)$ es un estimador insesgado de $\nabla f(x_k)$.

También podemos usar la esperanza condicional para acotar $\mathbb{E}[\|g(x_k, \xi_k)\|^2]$, que es el otro término en (28):

$$\mathbb{E}[\|g(x_k, \xi_k)\|^2] = \mathbb{E}[\mathbb{E}_{\xi_k}[\|g(x_k, \xi_k)\|^2 \, | \xi_1, \ldots, \xi_{k-1}]] \leq \mathbb{E}[L^2 \|x_k - x^*\|^2 + B^2].$$

Volviendo a la recurrencia (28) para a_{k+1}, usando los cálculos anteriores nos queda:

$$a_{k+1} = a_k - 2\alpha_k \mathbb{E}[\langle \nabla f(x_k), x_k - x^* \rangle] + \alpha_k^2 \mathbb{E}[L^2 \|x_k - x^*\|^2 + B^2].$$

Por linealidad de la esperanza, el tercer término nos queda $\alpha_k^2 L^2 a_k + \alpha_k^2 B^2$, entonces la recurrencia se simplifica a:

$$a_{k+1} \leq a_k - 2\alpha_k \mathbb{E}[\langle \nabla f(x_k), x_k - x^* \rangle] + \alpha_k^2 L^2 a_k + \alpha_k^2 B^2. \tag{30}$$

Hemos llegado a una expresión para ver cómo se reduce el error esperado en cada paso, con respecto al error anterior. Discutiendo según L y B damos con distintos resultados.

Caso $L = 0$

Asumimos que f es convexa y diferenciable, tal como sucede en el ejemplo 3.3.3 de la regresión logística,. Como asumimos $L = 0$, la recurrencia (30) se simplifica a:

$$a_{k+1} \leq a_k - 2\alpha_k \mathbb{E}[\langle \nabla f(x_k), x_k - x^* \rangle] + \alpha_k^2 B^2. \tag{31}$$

Defininendo

$$\lambda_k = \sum_{j=0}^{k} \alpha_j, \text{ y} \qquad \overline{x_k} = \frac{\sum_{j=0}^{k} \alpha_j x_j}{\sum_{j=0}^{k} \alpha_j} = \lambda_k^{-1} \sum_{j=0}^{k} \alpha_j x_j,$$

λ_k es la suma de los pasos y $\overline{x_k}$ se puede entender como la suma de todas las iteraciones normalizada por el tamaño total de los pasos. Vamos a estudiar cómo se desvía $f(\overline{x}_k)$ de $f(x^*)$. Para esto nos interesa tener información de $\mathbb{E}[f(\overline{x}_k) - f(x^*)]$. Se cumple que:

$$\begin{aligned} \mathbb{E}[f(\overline{x}_k) - f(x^*)] &= \mathbb{E}[f(\lambda_k^{-1} \sum_{j=0}^{k} \alpha_j x_j) - f(x^*)] \\ &\leq \mathbb{E}[\lambda_k^{-1} \sum_{j=1}^{k} \alpha_j f(x_j) - f(x^*)] \\ &= \lambda_k^{-1} \sum_{j=1}^{k} \alpha_j \mathbb{E}[f(x_j) - f(x^*)], \end{aligned} \tag{32}$$

donde en la primera desigualdad utilizamos la convexidad de f, ya que $\overline{x}_k$ es una combinación lineal convexa de los x_j, es decir, una suma ponderada donde cada uno de los pesos es $\frac{\alpha_j}{\lambda_k}$, no negativos y suman todos 1, por lo tanto $f(\sum_{j=0}^{k} \frac{\alpha_j}{\lambda_k} x_j) \leq \sum_{j=0}^{k} \frac{\alpha_j}{\lambda_k} f(x_j)$. En la última igualdad usamos la linealidad de la esperanza.

Ahora, como f es convexa y diferenciable, se cumple el Lema 3.1.1, de donde se sigue que $\forall x, z \in \mathbb{R}^n : f(x) - f(z) \leq -\langle \nabla f(x), x - z \rangle$. En particular, se cumple $f(x_j) - f(x^*) \leq -\langle \nabla f(x_j), x_j - x^* \rangle$. Aplicando esto en la desigualdad (32) obtenemos:

$$\lambda_k^{-1} \sum_{j=1}^{k} \alpha_j \mathbb{E}[f(x_j) - f(x^*)] \leq -\lambda_k^{-1} \sum_{j=1}^{k} \alpha_j \mathbb{E}[\langle \nabla f(x_j), x_j - x^* \rangle]. \tag{33}$$

Entonces, con lo visto hasta ahora sabemos que

$$\mathbb{E}[f(\overline{x_k}) - f(x^*)] \leq -\lambda_k^{-1} \sum_{j=1}^{k} \alpha_j \mathbb{E}[\langle \nabla f(x_j), x_j - x^* \rangle]. \tag{34}$$

Lo cual vincula el error $\mathbb{E}[f(\overline{x_k}) - f(x^*)]$ con una suma en la que aparece el gradiente de f, igual que en la recurrencia para a_k. Trabajamos sobre la expresión (31) para seguir acotando el error esperado. Para un $j \in \{1, \dots, k\}$ se cumple:

$$\begin{aligned} a_{j+1} &\leq a_j - 2\alpha_j \mathbb{E}[\langle \nabla f(x_j), x_j - x^* \rangle] + \alpha_j^2 B^2 \\ \Rightarrow a_{j+1} - a_j - \alpha_j^2 B^2 &\leq -2\alpha_j \mathbb{E}[\langle \nabla f(x_j), x_j - x^* \rangle]. \end{aligned}$$

Por lo tanto:

$$\frac{1}{2}(a_j - a_{j+1}) + \frac{1}{2}\alpha_j^2 B^2 \geq \alpha_j \mathbb{E}[\langle \nabla f(x_j), x_j - x^* \rangle].$$

Si sumamos en $j = 1, \dots, k$ y multiplicamos por λ_k^{-1}, obtenemos:

$$\begin{aligned} \lambda_k^{-1} \sum_{j=1}^{k} \alpha_j \mathbb{E}[\langle \nabla f(x_j), x_j - x^* \rangle] &\leq \lambda_k^{-1} \sum_{j=1}^{k} \frac{1}{2}(a_j - a_{j+1}) + \frac{1}{2}\alpha_j^2 B^2 \\ &= \lambda_k^{-1} \frac{1}{2}[(\sum_{j=1}^{k} a_j - a_{j+1}) + B^2 \sum_{j=1}^{k} \alpha_j^2] \\ &= \lambda_k^{-1} \frac{1}{2}(a_1 - a_{k+1} + B^2 \sum_{j=1}^{k} \alpha_j^2). \end{aligned}$$

Ahora, $a_{k+1} = \mathbb{E}[\|x_{k+1} - x^*\|^2] \geq 0$ y por lo tanto $a_1 - a_{k+1} \leq a_1$, entonces:

$$\begin{aligned} \lambda_k^{-1} \frac{1}{2}(a_1 - a_{k+1} + B^2 \sum_{j=1}^{k} \alpha_j^2) &\leq \lambda_k^{-1} \frac{1}{2}(a_1 + B^2 \sum_{j=1}^{k} \alpha_j^2) \\ &= \frac{a_1 + B^2 \sum_{j=1}^{k} \alpha_j^2}{2\lambda_k} \\ &= \frac{\mathbb{E}[\|x_1 - x^*\|^2] + B^2 \sum_{j=1}^{k} \alpha_j^2}{2\lambda_k} \quad = \frac{D_0^2 + B^2 \sum_{j=1}^{k} \alpha_j^2}{2\sum_{j=1}^{k} \alpha_j}. \end{aligned}$$

Donde denotamos $D_0 = \mathbb{E}[\|x_1 - x^*\|]$. Siguiendo la cadena de desigualdades hemos probado entonces que:

$$\mathbb{E}[f(\overline{x_k}) - f(x^*)] \leq \frac{D_0^2 + B^2 \sum_{j=1}^{k} \alpha_j^2}{2\sum_{j=1}^{k} \alpha_j}. \tag{35}$$

Teniendo este resultado, es inmediato probar el siguiente

TEOREMA 3.3.1. *Sea $f : \mathbb{R}^n \to \mathbb{R}$ convexa y diferenciable. Supongamos que corremos el algoritmo de SGD por T iteraciones con paso constante $\alpha > 0$, cumpliendo las condiciones H1, H2, H3, y que además $L = 0$. Definimos:*

$$\alpha_{opt} = \frac{D_0}{B\sqrt{T}}, \text{ y} \qquad \theta = \frac{\alpha}{\alpha_{opt}}.$$

Entonces se cumple la siguiente cota:

$$\mathbb{E}[f(\overline{x}_T) - f(x^*)] \leq \left(\frac{1}{2}\theta + \frac{1}{2}\theta^{-1}\right) \frac{B\mathbb{E}[\|x_1 - x^*\|]}{\sqrt{T}}. \tag{36}$$

PRUEBA. Tenemos $\alpha_{\text{opt}} = \frac{D_0}{B\sqrt{T}}$ y $\theta = \frac{\alpha}{\alpha_{opt}} = \frac{\alpha B\sqrt{T}}{D_0}$. De la definición de θ también tenemos que $\alpha = \theta\alpha_{opt}$. Sustituyendo en la ecuación (35) obtenemos:

$$\begin{aligned}
\mathbb{E}[f(\overline{x}_T) - f(x^*)] &\leq \frac{D_0^2 + B^2 \sum_{j=1}^k \alpha_j^2}{2\sum_{j=1}^k \alpha_j} \\
&= \frac{D_0^2 + B^2 T(\theta\alpha_{opt})^2}{2T\theta\alpha_{opt}} \\
&= \frac{D_0^2}{2T\theta\alpha_{opt}} + \frac{1}{2}B^2\theta\alpha_{opt} \\
&\overset{\text{sust. } \alpha_{opt}}{=} \frac{D_0^2}{2T\theta}\frac{B\sqrt{T}}{D_0} + \frac{B^2\theta}{2}\frac{D_0}{B\sqrt{T}} \\
&= \frac{D_0 B}{2\sqrt{T}\theta} + \frac{B\theta D_0}{2\sqrt{T}} \\
&= \left(\frac{1}{2}\theta^{-1} + \frac{1}{2}\theta\right)\frac{D_0 B}{\sqrt{T}}.
\end{aligned}$$

■

Si bien el resultado refiere al promedio $\overline{x}_T$, al ser un resultado asíntotico, se puede extender a $f(x_T)$. Dando lugar a una cota para $\mathbb{E}[f(\overline{x}_T) - f(x^*)]$. No presentaremos este desarrollo aquí.

Esta condición nos dice qué tan grande puede ser el error esperado en función del α, de la distancia inicial promedio al óptimo, y de la cantidad de pasos T. La cota es más pequeña cuando $\theta = 1$, es decir, cuando $\alpha = \alpha_{opt}$, por eso lo denominamos como óptimo. A su vez, aumentar la cantidad de pasos T disminuye la cota, pero el retorno no es lineal, si aumentamos 100 veces la cantidad de pasos, a lo sumo disminuimos el error esperado en 10 veces. La entropía cruzada como función de coste para el modelo de regresión logística cumple las hipótesis H1, H2 y H3, como vimos en el ejemplo 3.3.3 y además es convexa (ver subsección en el capítulo siguiente 4.1.2) y diferenciable, por lo que se encuentra en las hipótesis del teorema anterior n y obtenemos una cota para el error esperado al ejecutar el algoritmo del SGD.

Caso general

Aquí, en principio, solo pedimos que $B, L > 0$. Además pedimos que f sea fuertemente convexa y diferenciable. La definición de convexidad fuerte nos dice que existe $m > 0$ tal que, para todo x, y:

$$f(y) \geq f(x) + \langle \nabla f(x), y - x \rangle + \frac{m}{2} \|y - x\|^2 .$$

Esto, a su vez, implica que

$$\langle \nabla f(x_k), x_k - x^* \rangle \geq m \|x_k - x^*\|^2 , \forall k.$$

Si combinamos esto con la condición (30) obtenemos:

$$a_{k+1} \leq a_k - 2\alpha_k m \mathbb{E}[\|x_k - x^*\|^2] + \alpha_k^2 L^2 a_k + \alpha_k^2 B^2. \tag{37}$$

En lo que sigue, nos centramos en el caso de α fijo, por lo que podemos escribir $\alpha_k \equiv \alpha$. Siguiendo con la deducción, sacamos $a_k = \mathbb{E}[\|x_k - x^*\|^2]$ de factor común y obtenemos:

$$a_{k+1} \leq (1 - 2m\alpha + \alpha^2 L^2) a_k + \alpha^2 B^2. \tag{38}$$

La ecuación anterior podemos escribirla como $a_k \leq A a_{k-1} + D$, siendo

$$A = 1 - 2m\alpha + \alpha^2 L^2 \text{ y } D = \alpha^2 B^2.$$

Si desarrollamos la recurrencia, obtenemos que $a_k \leq A^k a_0 + D \sum_{i=0}^{k-1} A^i$. Ahora, la suma $\sum_{i=0}^{k-1} A^i$ es igual a $\frac{A^k - 1}{A - 1}$, entonces:

$$a_k \leq A^k a_0 + D \frac{A^k - 1}{A - 1}. \tag{39}$$

Nos gustaría que, a medida que el número k de iteraciones crezca, la cantidad A^k tienda a 0, para que la cota sea lo más pequeña posible. Para esto, vamos a imponer como condición adicional que $0 < A < 1$. Esto implica que

$$0 < 1 - 2m\alpha + \alpha^2 L^2 < 1.$$

La desigualdad derecha implica que

$$\alpha(\alpha L^2 - 2m) < 0,$$

y como $\alpha, L > 0$, esto, a su vez, implica:

$$\alpha < \frac{2m}{L^2}. \tag{40}$$

Esta es la condición que tiene que cumplir α. Por otro lado, la desigualdad izquierda implica que el polinomio $p(\alpha) = \alpha^2 L^2 - \alpha 2m + 1$ sea positivo. Esto es, que el discriminante sea positivo (de esta manera, no tiene raíces reales):

$$4m^2 - 4L^2 < 0.$$

Como $m, L > 0$, esto implica que

$$m < L.$$

En resumen, si L es mayor a la constante m de convexidad fuerte de f y α cumple que $\alpha < \frac{2m}{L^2}$, entonces $0 < A < 1$ y, por lo tanto, $A^k \to 0$.

Retomando la ecuación (39), asumiendo que se cumplen las condiciones para que $A^k \to 0$, se tiene que mientras más iteremos, menor será el error esperado $a_k = \mathbb{E}[\|x_k - x^*\|^2]$. Si k es lo suficientemente grande, la cota es equivalente a:

$$a_k \leq -\frac{D}{A-1} = \frac{\alpha^2 B^2}{2m\alpha - \alpha^2 L^2}. \tag{41}$$

Resumimos este desarrollo en el siguiente teorema:

TEOREMA 3.3.2. *Sea $f : \mathbb{R}^n \to \mathbb{R}$ diferenciable con mínimo global en $x^* \in \mathbb{R}^n$ y que cumple las condiciones H1, H2 (con L, B cualquiera) y H3. Además f es fuertemente convexa con constante de convexidad m. Supongamos que corremos el algoritmo de SGD por T iteraciones con paso $\alpha > 0$ fijo, entonces tenemos la siguiente cota para el error esperado $a_T = \mathbb{E}[\|x_T - x^*\|]$:*

$$a_T \leq \frac{\alpha^2 B^2}{2m\alpha - \alpha^2 L^2}. \tag{42}$$

Así que a lo sumo podemos decir que teóricamente el error cae dentro de una bola centrada en el óptimo, pero puede que oscile dentro de ella sin acercarse nunca a él. Notar que el error en el teorema anterior está dado en el domino, habla de cuanto dista x_T del óptimo x^*, mientras que el Teorema 3.3.1 se refiere a un error en los valores funcionales.

Según lo que vimos en el ejemplo 3.3.4, un riesgo empírico donde cada uno de los sumandos f_i tenga gradiente ∇f_i Lipschitz cumple las condiciones H1, H2 y H3, así que si además es fuertemente convexo (ya suponemos diferenciable), entonces podemos aplicar el teorema anterior y conseguir una cota para el error en el resultado del SGD.

3.4. La teoría en práctica

Es importante conocer los teoremas para asegurarse de no estar utilizando un algoritmo inadecuado, para entender qué probar y qué no probar. Las librerías de aprendizaje de datos en Python, por ejemplo, suelen tener valores por defecto para los hiperparámetros del SGD, pero cuando estos fallan hay que conocer la teoría para saber en qué parte del espacio de hiperparámetros es razonable experimentar.

Aunque vale la pena mencionar que, en la experiencia de quién escribe, no es estrictamente necesario verificar que se cumplen las condiciones de los teoremas que vimos para realizar un buen trabajo como científico de datos. En muchos casos, el desempeño final del modelo está más influenciado por la calidad de los datos de entrenamiento que por la elección del paso óptimo exacto.

Por otro lado, los teoremas no siempre aplican: tanto el Teorema 3.3.1 como 3.3.2 requieren condiciones de convexidad para aplicarlos. Sin embargo, cabe mencionar que existen otros teoremas de convergencia local del SGD que no exigen convexidad (ver resultados relacionados con las condiciones de Robbins-Monro [1]).

En el capítulo siguiente estudiaremos la regresión logística y veremos que la función de costo de 'entropía cruzada' que ya mencionamos en el Ejemplo 3.3.3, es convexa para este modelo; en este caso, sí podríamos aplicar el Teorema 3.3.1. También hablaremos en detalle de las redes neuronales y veremos que no suelen ser convexas, por lo que no

aplican los teoremas vistos, y que, más allá de eso, hay varios detalles a tener en cuenta a la hora de optimizar.

CAPÍTULO 4

Regresión Logística y Redes Neuronales

En la primera sección de este capítulo hablaremos de la regresión logística (RL) que es un modelo estadístico clásico. Daremos su definición, cómo usarla para resolver problemas de clasificación y de la función de costo que se minimiza en este caso para poder llegar a los pesos adecuados para resolver el problema.
En la segunda sección hablaremos de redes neuronales *feedforward*, que son modelos de aprendizaje profundo. Daremos su definición con cada uno de sus componentes, comentaremos las dificultades que existen en el proceso de encontrar sus parámetros óptimos y el algoritmo de *backward propagation* para calcular gradientes de redes de manera eficiente. También probaremos el teorema de aproximación universal [6], un resultado que nos dice, en esencia, que las redes son buenos aproximadores de funciones. La RL no es un modelo de aprendizaje profundo, pero veremos que se puede escribir como un caso particular de red neuronal.

4.1. Regresión Logística

Una regresión es una función $l : \mathbb{R}^n \to \mathbb{R}$ que busca predecir $y \in \mathbb{R}$ a partir de valores en $\mathbb{R}^n$. La regresión logística es una regresión que compone una transformación afín (transformación lineal más traslación) con una función no lineal, que es la función logística, también llamada **sigmoide**.

Definición 4.1.1. **(Regresión logística)** si $x \in \mathbb{R}^n$, se define la regresión logística $l : \mathbb{R}^n \to (0, 1)$ tal que:

$$l(x) = \sigma(\langle w, x\rangle + b), \tag{43}$$

donde $w \in \mathbb{R}^n$ es un vector de **pesos** o *weights*, $b \in \mathbb{R}$ es el **sesgo** o *bias* y $\sigma : \mathbb{R} \to (0, 1)$ es la función sigmoide (ver Figura 1) dada por:

$$\sigma(z) = \frac{1}{1 + e^{-z}}. \tag{44}$$

La RL se puede utilizar para resolver problemas de clasificación, donde l se compone con una función de umbral o *threshold* $T_\alpha : (0, 1) \to \{0, 1\}$, que depende de un parámetro α (el umbral), con $0 < \alpha < 1$:

$$T_\alpha(t) = \begin{cases} 1 & \text{si } t \geq \alpha, \\ 0 & \text{si } t < \alpha. \end{cases}$$

Se dice que el modelo predice la clase positiva si $T_\alpha(l(x)) = 1$ y la negativa en otro caso.

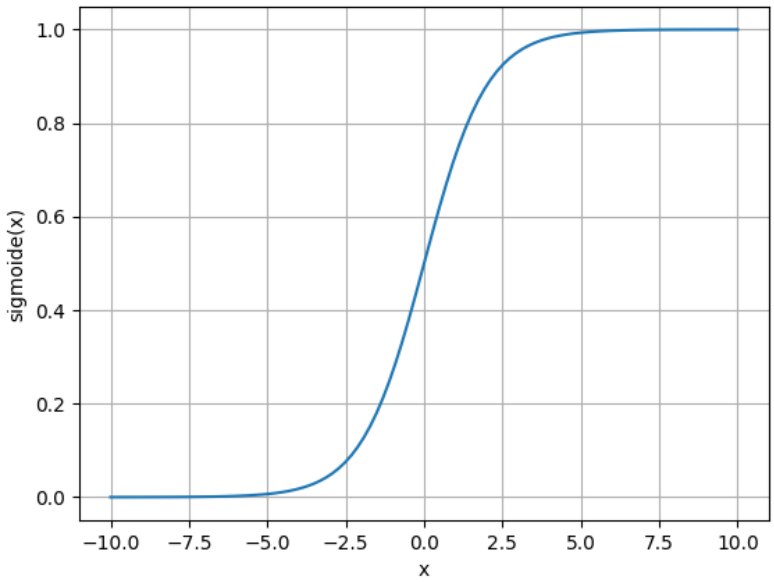


FIGURA 1. Función sigmoide en [-10, 10]

OBSERVACIÓN 4.1.1.
Como ejercicio vamos a formalizar la salida de la RL como una distribución de probabilidad, esto nos será útil luego. Si la clasificación es binaria, con una clase positiva, `A`, y una negativa, `B`, entonces, fijada una observación x, la RL define una probabilidad decretando que:

$$P(A|x) := l(x) \text{ y } P(B|x) = 1 - l(x).$$

Como $l(x) \in (0, 1)$ nos queda que $P(A|x), P(B|x) \in (0, 1)$ y además $P(A|x) + P(B|x) = 1$. Por lo que $P_x = P(\cdot|x)$ define una probabilidad sobre el espacio muestral constituido por las clases del problema (i.e. $\Omega = \{A, B\}$).

Aplicación a PLN
La RL tiene como punto de entrada un vector, entonces para aplicarla al PLN se deben mapear los textos a vectores. Cada entrada del vector es una característica del texto o *feature*.

Las features se pueden definir a mano y en ese caso pueden variar mucho dependiendo del problema a tratar. Por ejemplo, si el problema es etiquetar textos con emociones positivas o negativas, podríamos tener un diccionario de palabras asociadas a cada clase, y a cada texto se le asigna un vector en $\mathbb{R}^2$, siendo la primera entrada la cantidad de palabras positivas y la segunda la cantidad de palabras negativas. Para tareas como esta se pueden usar las expresiones regulares del Capítulo 2.

Otra alternativa para definir las features es usar *representation learning*, es decir, usar

algoritmos para calcular las features automáticamente; aprender a mapear textos a vectores. Un algoritmo conocido para esto es Word2Vec. A esos vectores se les llama *embeddings*. Profundizaremos en los embeddings en el Capítulo 5.

4.1.1. Función de costo.

La RL definida en (43) tiene como dominio a $\mathbb{R}^n$ pero depende de los parámetros w y b. Se deben ajustar los parámetros a los datos proporcionados en un conjunto de entrenamiento. Ese ajuste se da minimizando una función de costo o *loss*, que cuantifica, para todas las observaciones x, las diferencias entre el valor predicho por el modelo $\hat{y}$ y el valor verdadero y. Queremos que esas diferencias sean lo más pequeñas posible. La función de costo es el riesgo empírico $R_{emp}[m]$ que mencionamos en el capítulo de optimización, en este caso el modelo m es la RL. En general vamos a denotar por L a la función de costo.

La primera función de costo que se nos podría ocurrir podría ser el valor absoluto, o sea $L_{abs}(\hat{y}, y) = |\hat{y} - y|$. El problema con L_{abs} es que no es diferenciable en todo punto, propiedad que nos gustaría tener ya que pretendemos obtener los parámetros adecuados para la RL (el w y el b) optimizando. Por el motivo anterior se suele utilizar el error cuadrático medio para problemas donde la variable a predecir es real, es decir en una regresión, y lo que llamaremos 'entropía cruzada' si la variable es categórica (en otras palabras, un conjunto finito), que corresponde se corresponde con un problema de clasificación. Nos enfocaremos en el caso de clasificación binaria y llegaremos a la fórmula de la entropía cruzada de distintas maneras, viendo cómo se relaciona con otros conceptos de matemática presentes en teoría de la información y en estadística.

Representaremos las clases como 1 (clase positiva) o 0 (clase negativa). Dada una observación x, buscamos que $T(l(x)) = y$. Es decir, queremos una RL que retorne una probabilidad $\hat{y} = p(y|x)$ que sea cercana a 0 si $y = 0$ y cercana a 1 si $y = 1$.

4.1.1.1. *Fórmula de la entropía cruzada a partir de variables Bernoulli.*

Por la observación 4.1.1, dada una entrada x y su etiqueta y, la función $l(x)$ define una distribución de probabilidad $\hat{y} := P(y|x)$. Esta probabilidad se puede modelar con una v.a. Bernoulli: los resultados pueden ser 1 con probabilidad $\hat{y}$ o 0 con probabilidad $1-\hat{y}$. La fórmula para la función de probabilidad de una v.a. $Z \sim Ber(p)$ es:

$$P(Z = z) = p^z(1-p)^{1-z} = \begin{cases} p & \text{si } z = 1, \\ 1-p & \text{si } z = 0. \end{cases}$$

En nuestro caso z es la etiqueta verdadera y p es $\hat{y}$. Nos queda entonces:

$$P(y|x) = \hat{y}^y(1-\hat{y})^{1-y} = \begin{cases} \hat{y} & \text{si } y = 1, \\ 1-\hat{y} & \text{si } y = 0. \end{cases}$$

Ahora aplicamos logaritmo a ambos miembros de la igualdad. El logaritmo es conveniente, porque al aplicarlo a un producto lo que nos quedará es una suma, más fácil de derivar, y, por lo tanto, de optimizar. El logartimo también ayuda desde un punto de vista computacional: al tratar con probabilidades, todas serán menores a 1, si no usamos

el logaritmo entonces vamos a tener un producto de cantidades menores a 1, que va a tender a cero, y esto puede terminar en un problema de *underflow*[1].

$$\begin{aligned}\log P(y|x) &= \log \hat{y}^y (1-\hat{y})^{1-y} \\ &= \log \hat{y}^y + \log(1-\hat{y})^{1-y} \\ &= y \log \hat{y} + (1-y)\log(1-\hat{y}).\end{aligned}$$

Queremos maximizar la probabilidad de obtener la etiqueta real y dado x, o sea, maximizar $P(y|x)$. Esto equivale a maximizar $\log P(y|x)$ que a su vez equivale a minimizar $-\log P(y|x)$. La función de costo a minimizar queda entonces:

$$L(\hat{y}, y) := -\log P(y|x) = -y \log \hat{y} - (1-y)\log(1-\hat{y}). \tag{45}$$

La fórmula anterior se le llama entropía cruzada o *cross entropy*. Sustituyendo $\hat{y} = l(x)$ según (43) nos queda una función que depende de los los pesos w y b:

$$L(w, b) = -y \log \sigma(\langle w, x\rangle + b) - (1-y)\log(1-\sigma(\langle w, x\rangle + b)). \tag{46}$$

Minimizar la función anterior con respecto a w y b equivale a encontrar los parámetros que mejor se ajusten a la distribución Bernoulli con la que modelamos el problema al principio.

4.1.1.2. *Relación con la entropía.* La entropía es un concepto que proviene del mundo de la física y la química, en particular relacionado con el área de la termodinámica, que se refiere a cómo la energía de un sistema interactúa con la energía de otros sistemas y con el ambiente. En el dominio de la probabilidad y la teoría de la información, se define como una cantidad asociada a una distribución de probabilidad.

Intuición
Antes de dar la definición de entropía, trataremos de dar una intuición de dónde viene la misma, mediante un ejemplo. La idea que tenemos que tener es que queremos dar una medida de qué tan *incierta* es una distribución de probabilidad.

Ejemplo 4.1.1.
Supongamos que trabajamos en una estación de monitoreo climático y nuestro trabajo es enviar un mensaje a la población con el pronóstico del clima para el día siguiente.

Diremos que nuestro mensaje contiene 1 *bit* de información si reduce la incertidumbre del problema a la mitad. De esta manera, si los pronósticos de clima posibles son *soleado* o *lluvioso* con probabilidades de ocurrir $p_1 = p_2 = \frac{1}{2}$, entonces dar el pronóstico reduce la incertidumbre a la mitad, transmite 1 bit de información. Si tuvieramos 8 estados posibles de clima, igualmente probables, dar el pronóstico transmite $\log_2 8 = 2$ bits de información, que es $\log_2(\frac{1}{p_i})$, siendo p_i la probabilidad del evento i, en este caso $p_i = 1/8$ para $i = 1, \ldots, 8$. Entonces

[1]Problema que surge al tratar de manejar números más pequeños de los que la computadora puede

$$\log_2 \frac{1}{p_i} = -\log_2 p_i$$

es la información dada al pronosticar el evento i. Ahora, si la p no es uniforme, por ejemplo, si $p_1 = \frac{3}{4}$ y $p_2 = \frac{1}{4}$ en el caso de dos estados, entonces la información de cada evento es $-\log_2 \frac{3}{4} = 0.41$ y $-\log_2 \frac{1}{4} = 2$ bits, respectivamente. Pronosticar que va a estar soleado nos da menos información que pronosticar lluvia, tiene sentido, ya que es un evento más probable. Nuestro mensaje promedio tiene

$$p_1(-\log_2 p_1) + p_2(-\log_2 p_2)$$

bits de información.

Entropía

El concepto que surge al final del ejemplo anterior de *información dada por el evento promedio* se formaliza con la definición de entropía de una distribución.

Definición 4.1.2. **Entropía**

Sea p una distribución de probabilidad discreta que toma valores con probabilidad $p_i = p(i)$. Se define la entropía $H(p)$ como:

$$H(p) = -\sum_i p_i \log(p_i). \tag{47}$$

Es equivalente trabajar con el logaritmo natural que con el logaritmo en base 2, por la propiedad de cambio de base. Esta definición se corresponde con la intuición de la entropía como medida de incertidumbre. Si la masa de probabilidad está concentrada en un solo evento i con $p_i = 1$ y $p_j = 0$, si $j \neq i$, es decir, no hay incertidumbre, entonces la entropía es igual a 0. Por el contrario, la entropía es máxima cuando la distribución es uniforme, es decir, cuando la incertidumbre es máxima (todos los eventos son igualmente probables).

Entropía cruzada

La entropía cruzada es una variación de la fórmula (47) adaptada para considerar dos distribuciones de probabilidad.

Definición 4.1.3. **Entropía cruzada**

Sean dos distribuciones de probabilidad discretas p y q, que toman valores con probabilidad $p_i = p(i)$ y $q_i = q(i)$. La entropía cruzada se define como:

$$H(p, q) = -\sum_i p_i \log(q_i). \tag{48}$$

Si $q = p$, entonces $H(p, q) = H(p)$. Otra cantidad relacionada es la diferencia

$$D_{KL}(p||q) = H(p, q) - H(p),$$

conocida como divergencia de Kullback-Leibler.

Aplicación en nuestro problema

Para el contexto de un problema de clasificación de n clases $\{1, \dots, n\}$ definimos p como la probabilidad de que la etiqueta verdadera sea i. En el caso binario p toma valores $p_1 = y$ y $p_0 = 1 - y$.
Por otro lado, definimos q_i como la probabilidad asignada por el modelo de regresión logística para la etiqueta i. Entonces $q_1 = \hat{y} = l(x)$ para cierta observación x, y $q_0 = 1 - \hat{y}$. Aplicando la ecuación (48) nos queda que la entropía cruzada entre la distribución real p y la modelada q es:

$$\begin{aligned} H(p, q) &= -\, p_0 \log(q_0) - p_1 \log(q_1) \\ &= -\, (1 - y) \log(1 - \hat{y}) - y \log \hat{y}. \end{aligned} \tag{49}$$

Esta fórmula es igual a la función de costo definida en (45)

4.1.1.3. *Relación con el principio de máxima verosimilitud.*
En estadística, es usual querer estimar cantidades desconocidas, como una esperanza o parámetros de una distribución, a partir de una muestra de datos. Existen diferentes métodos para hacer esto, uno de ellos es el estimador de máxima verosimilitud (EMV), que definimos a continuación. Veremos que maximizar la verosimilitud es equivalente a minimizar la función de costo de entropía cruzada.

Definición 4.1.4. **Verosimilitud**
Sean $X_1, \dots X_n$ variables aleatorias i.i.d. con distribución $q(x; \theta)$ que depende de un parámetro θ. La función de verosimilitud es otra v.a. que se define como

$$\mathcal{L}(\theta) = \prod_{i=1}^{n} q(X_i; \theta), \tag{50}$$

por conveniencia se suele utilizar $\ell(\theta) = \log \mathcal{L}(\theta) = \sum_{i=1}^{n} \log q(X_i; \theta)$.

El estimador de máxima verosimilitud (EMV) es el θ donde se maximiza esa función:

$$\hat{\theta} = \operatorname*{argmax}_{\theta} \mathcal{L}(\theta) = \operatorname*{argmax}_{\theta} \ell(\theta). \tag{51}$$

La explicación para esta definición es que si $x_1, \dots, x_n$ es la muestra observada de las X_i, como las v.a. son independientes entonces la probabilidad de efectivamente realizar esa muestra es un producto $\prod_{i=1}^{n} q(x_i; \theta)$, es decir $\mathcal{L}(\theta)$ donde cada v.a. X_i toma el valor x_i. Así que el EMV se puede entender como los parámetros θ donde se maximiza la probabilidad de observar lo observado.

Yendo a nuestro caso, tenemos una muestra $(x_1, y_1), \dots, (x_N, y_N)$ de observaciones etiquetadas. Definimos como k_i la cantidad de veces que se repite el par (x_i, y_i).
La distribución empírica está dada por $p_i := p(Y = y_i | X = x_i) = \frac{k_i}{N}$. Donde X, Y son v.a. de cuya distribución conjunta se obtuvo la muestra. Escribimos $\Delta = \Omega_x \times \Omega_y$ el recorrido de X, Y, que incluye la muestra.

Por otro lado, consideramos q_i como la probabilidad que el modelo le asigna a la etiqueta y_i, para la observación x_i. Es decir $q_i := q(Y = y_i|X = x_i|\theta)$. La función de verosimilitud para la muestra con la distribución q es:

$$\mathcal{L}(\theta) = \prod_{i}^{N} q(Y = y_i|X = x_i|\theta) = \prod_{(x_i,y_i)\in\Delta} q(Y = y_i|X = x_i|\theta)^{k_i}.$$

Por lo tanto, tomando logaritmo nos queda:

$$\begin{aligned}\ell(\theta) &= \sum_{(x_i,y_i)\in\Delta} \log q(Y = y_i|X = x_i|\theta)^{k_i} \\ &= \sum_{(x_i,y_i)\in\Delta} k_i \log q(Y = y_i|X = x_i|\theta) \\ &= \sum_{(x_i,y_i)\in\Delta} Np(Y = y_i|X = x_i) \log q(Y = y_i|X = x_i|\theta) \\ &= -NH(p, q).\end{aligned}$$

Así que $\ell(\theta) = -NH(p, q)$. Esta es la relación entre la función de verosimilitud y la entropía cruzada. Al minimizar $H(p, q)$, que ya vimos que es equivalente a minimizar la función de costo de la RL, estamos encontrando el máximo de la función de verosimilitud.

4.1.2. Convexidad de la función de costo.

La entropía cruzada en el caso de la regresión logística resulta ser una función convexa con respecto a los parámetros w y b. Así que cualquier mínimo local será global, facilitando el proceso de optimización, como vimos en el Capítulo 3.

PROPOSICIÓN 4.1.1. *La función de costo definida en* (46) *es convexa con respecto a los parámetros w y b.*

PRUEBA.
Queremos probar que

$$L(w, b) = -y \log \sigma(\langle w, x\rangle + b) - (1 - y) \log(1 - \sigma(\langle w, x\rangle + b))$$

es convexa con respecto a w y b. Para esto vamos a probar que la matriz Hessiana de L es semidefinida positiva.
Para empezar, necesitaremos la derivada de $\sigma(t) = \frac{1}{1+e^{-t}}$, que es

$$\sigma'(t) = \frac{e^{-t}}{(1 + e^{-t})^2} = \sigma(t)(1 - \sigma(t)).$$

Denotamos $w = (w_1, \ldots, w_n), x = (x_1, \ldots, x_n)$ y $z = \langle w, x\rangle + b$. El primer paso de la prueba es calcular las derivadas primeras $\frac{\partial L}{\partial w_i}$ para cada peso w_i y para el sesgo b.

$$\begin{aligned}\frac{\partial L(w,b)}{\partial w_j} =& -y\frac{\partial}{\partial w_j}\log\sigma(\langle w,x\rangle+b)-(1-y)\frac{\partial}{\partial w_j}\log(1-\sigma(\langle w,x\rangle+b))\\ =&\left(\frac{-y}{\sigma(z)}+\frac{1-y}{1-\sigma(z)}\right)\frac{\partial\sigma(z)}{\partial w_j}\\ =&\frac{\sigma(z)-y}{\sigma(z)(1-\sigma(z))}\frac{\partial\sigma(z)}{\partial z}\frac{\partial z}{\partial w_j}\\ =&\frac{\sigma(z)-y}{\sigma(z)(1-\sigma(z))}\sigma(z)(1-\sigma(z))x_j\\ =&(\sigma(z)-y)x_j.\end{aligned}$$

Donde usamos que $\frac{\partial z}{\partial w_j}=x_j$. Del mismo modo calculamos la derivada con respecto a b:

$$\begin{aligned}\frac{\partial L(w,b)}{\partial b} =& -y\frac{\partial}{\partial b}\log\sigma(\langle w,x\rangle+b)-(1-y)\frac{\partial}{\partial b}\log(1-\sigma(\langle w,x\rangle+b))\\ =&\left(\frac{-y}{\sigma(z)}+\frac{1-y}{1-\sigma(z)}\right)\frac{\partial\sigma(z)}{\partial b}\\ =&\frac{\sigma(z)-y}{\sigma(z)(1-\sigma(z))}\frac{\partial\sigma(z)}{\partial z}\frac{\partial z}{\partial b}\\ =&\frac{\sigma(z)-y}{\sigma(z)(1-\sigma(z))}\sigma(z)(1-\sigma(z))\\ =&\sigma(z)-y.\end{aligned}$$

Donde usamos que $\frac{\partial z}{\partial b}=1$. Ahora podemos calcular las derivadas segundas, que son $(n+1)^2$, pero nos basta con entender la forma que tienen $\partial w_i\partial w_j, \partial w_i\partial b, \partial b\partial w_j, \partial b\partial b$. Primero calculamos $\partial w_i\partial w_j$:

$$\begin{aligned}\frac{\partial}{\partial w_j}\left(\frac{\partial L(w,b)}{\partial w_i}\right) =&\frac{\partial}{\partial w_j}(\sigma(z)-y)x_i\\ =&\frac{\partial}{\partial w_j}\sigma(z)x_i-\frac{\partial}{\partial w_j}yx_i\\ =&x_i\frac{\partial(\sigma(z))}{\partial z}\frac{\partial z}{\partial w_j}-0\\ =&\sigma(z)(1-\sigma(z))x_ix_j.\end{aligned}$$

Ahora las derivadas cruzadas:

$$\begin{aligned}\frac{\partial}{\partial w_j}\left(\frac{\partial L(w,b)}{\partial b}\right) =&\frac{\partial}{\partial w_j}(\sigma(z)-y)\\ =&\sigma(z)(1-\sigma(z))x_j.\end{aligned}$$

$$\begin{aligned}\frac{\partial}{\partial b}\left(\frac{\partial L(w,b)}{\partial w_i}\right) =& \frac{\partial}{\partial b}(\sigma(z)-y)x_i \\ =& \frac{\partial}{\partial b}\sigma(z)x_i - \frac{\partial}{\partial b}yx_i \\ =& x_i\frac{\partial(\sigma(z))}{\partial z}\frac{\partial z}{\partial b} - 0 \\ =& \sigma(z)(1-\sigma(z))x_i.\end{aligned}$$

Por último la derivada $\frac{\partial^2}{\partial b}$ nos queda:

$$\begin{aligned}\frac{\partial}{\partial b}\left(\frac{\partial L(w,b)}{\partial b}\right) =& \frac{\partial}{\partial b}(\sigma(z)-y) \\ =& \sigma(z)(1-\sigma(z)).\end{aligned}$$

Entonces la matriz hessiana de L es:

$$\begin{aligned}\nabla^2 L(w,b) =& \begin{bmatrix} \sigma(z)(1-\sigma(z))x_1^2 & \dots & \sigma(z)(1-\sigma(z))x_1x_n & \sigma(z)(1-\sigma(z))x_1 \\ \vdots & \ddots & \vdots & \vdots \\ \sigma(z)(1-\sigma(z))x_nx_1 & \dots & \sigma(z)(1-\sigma(z))x_n^2 & \sigma(z)(1-\sigma(z))x_n \\ \sigma(z)(1-\sigma(z))x_1 & \dots & \sigma(z)(1-\sigma(z))x_n & \sigma(z)(1-\sigma(z)) \end{bmatrix} \\ =& \sigma(z)(1-\sigma(z))A.\end{aligned}$$

Donde $A \in \mathbb{R}^{(n+1)\times(n+1)}$ es la matriz definida como:

$$A = \begin{bmatrix} x_1^2 & \dots & x_1x_n & x_1 \\ \vdots & \ddots & \vdots & \vdots \\ x_nx_1 & \dots & x_n^2 & x_n \\ x_1 & \dots & x_n & 1 \end{bmatrix}.$$

Queremos ver que $\nabla^2 L(w,b)$ es semidefinida positiva, por definición esto implica estudiar si $u\nabla^2 L(w,b)u \geq 0$, para un vector $u \in \mathbb{R}^{n+1}$ cualquiera. Sea $u = (u_1, \dots, u_{n+1})$, se tiene:

$$u\nabla^2 L(w,b)u^t = u\sigma(z)(1-\sigma(z))Au^t = \sigma(z)(1-\sigma(z))uAu^t.$$

Como $\sigma(t) \in (0,1)$, y, por lo tanto, $\sigma(z)(1-\sigma(z)) \geq 0$, para ver que $u\nabla^2 L(w,b)u \geq 0$ nos basta con ver que $uAu^t \geq 0$:

$$\begin{aligned}uAu^t =& \sum_{i=1}^{n}\sum_{j=1}^{n} x_ix_ju_ju_i + 2u_{n+1}\sum_{i=1}^{n} x_iu_i + u_{n+1}^2 \\ =& \sum_{i=1}^{n} x_i^2u_i^2 + \sum_{i=1}^{n}\sum_{j\neq i} x_ix_ju_ju_i + 2u_{n+1}\sum_{i=1}^{n} x_iu_i + u_{n+1}^2 \\ =& \left(u_{n+1} + \sum_{i=1}^{n} x_iu_i\right)^2 \geq 0.\end{aligned}$$

Esto prueba que la matriz Hessiana es semidefinida positiva en todo punto, ya que w y b son parámetros arbitrarios, de manera que la función de costo es convexa. ■

La convexidad de la función de costo la vuelve una función sencilla de optimizar, ya que el mínimo es el único punto crítico. De hecho, no solo tenemos asegurado acercarnos al mínimo, sino que la convexidad es una de las hipótesis del Teorema 3.3.1 que nos habla de la tasa de convergencia a ese mínimo.

La regresión logística es un modelo útil en muchos casos, es fácil de explicar y de optimizar. Si la RL tiene buen desempeño para nuestro problema, podemos conformarnos con ella o usarla como línea base para otros experimentos con modelos más complejos. En caso de necesitar mayor poder de modelado del que ofrece la regresión logística, podemos utilizar la familia de modelos que conocemos como redes neuronales.

4.2. Redes neuronales *feedforward*

4.2.1. Definición.

Una red neuronal se puede representar de varias maneras, pero, en primer lugar, es una función $f : \mathbb{R}^n \to \mathbb{R}^m$. Esa función está parametrizada, tal como la regresión logística. La f es una composición de funciones lineales y no lineales.

Es usual representar a las redes como un grafo, ver la Figura 2. A los nodos se les llama 'neurona', representando valores reales, y a las aristas 'pesos', palabra que ya venimos usando para referirnos a los parámetros de un modelo. La red de la Figura mencionada representa a la función $f(x) = a(\langle w, x\rangle + b)$. Si la función a es la sigmoide, lo usual en un problema de clasificación binaria, entonces nos queda la misma ecuación que en (43). Las redes neuronales generalizan a la regresión logística.

Las redes que estudiaremos en esta sección se dice que son de tipo *feedforward*, abreviadas FNN, por su nombre completo en inglés (*feedforward neural network*); la información fluye en un solo sentido, desde la entrada a la salida, de ahí el nombre. También se les puede llamar *multi-layer perceptron* (MLP). En estas redes no hay ciclos, como sí hay en otras arquitecturas, como las recurrentes (ver Capítulo 5).

4.2.2. Arquitectura.

Una red tiene capas o *layers*, una en sucesión de la otra, representando la composición de funciones vectoriales, donde las neuronas individuales son el resultado de la función escalar en cada coordenada. En la Figura 3 vemos una representación. Allí cada neurona h_i^j representa la combinación lineal de las salidas de la capa anterior compuesta con una función de activación: $h_i^k = a_k(\sum_{j=1}^{n_{k-1}} w_{ji} h_t^{k-1})$. El cálculo neurona a neurona entre dos capas, es decir, cada función escalar, se puede realizar en simultáneo usando matrices.

Denotamos a las capas como $l_0, l_1, \ldots, l_{k-1}, l_k$, donde l_0 alberga al vector de entrada x, las capas l_1 hasta l_{k-1} son las llamadas 'capas ocultas' y l_k es la capa de salida. Cada capa l_i tiene n_i neuronas. Definimos W_i de dimensiones $n_{i+1} \times n_i$ como la matriz de pesos entre las capas l_i y l_{i+1}, $b_i \in \mathbb{R}^{n_{i+1}}$ es el sesgo de la capa i-ésima, a_i es la función de activación e $y_i \in \mathbb{R}^{n_i}$ es la salida de la capa l_i. Usando la notación introducida, la

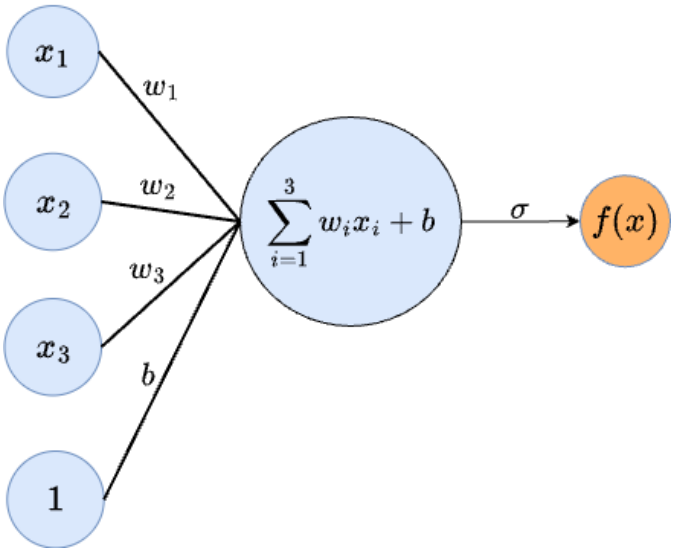


FIGURA 2. Representación de una red neuronal simple con una capa de entrada y una neurona de salida. Aquí $x = (x_1, x_2, x_3)$, $w = (w_1, w_2, w_3)$ es un vector de pesos, $b \in \mathbb{R}$ es el sesgo, y σ es una función no lineal llamada función de activación. El vector w pondera la influencia de cada entrada de x en el resultado final y el sesgo se computa concatentando a la entrada el valor real 1

función f que define la red está dada por recurrencia según las siguientes ecuaciones:

$$\begin{cases} y_0 = x, \\ y_i = a_i(W_{i-1}y_{i-1} + b_{i-1}) \text{ donde } i = 1, \ldots, k-1, \\ y_k = a_k(W_{k-1}y_{k-1} + b_{k-1}). \end{cases} \tag{52}$$

En general, la primera capa, la entrada, no tiene función de activación (a_0 es la identidad), la capa final tiene una función especial según el problema y las capas ocultas usan la misma activación. Esto no es estrictamente necesario por la teoría, pero es lo usual en la práctica.

4.2.3. Funciones de activación.

Las funciones de activación agregan el elemento de no linealidad al modelo, aumentando así el poder expresivo de la red (ver Sección 4.2.4). Sin ellas, la red degeneraría en una composición de funciones lineales, que termina siendo también una función lineal. A continuación enumeramos algunas de las funciones más utilizadas como función de activación.

- **Función sigmoide:**

$$\sigma(z) = \frac{1}{1 + e^{-z}}.$$

  Usada en clasificación binaria, ya la conocíamos de la regresión logística. Ver gráfico en Figura 1.

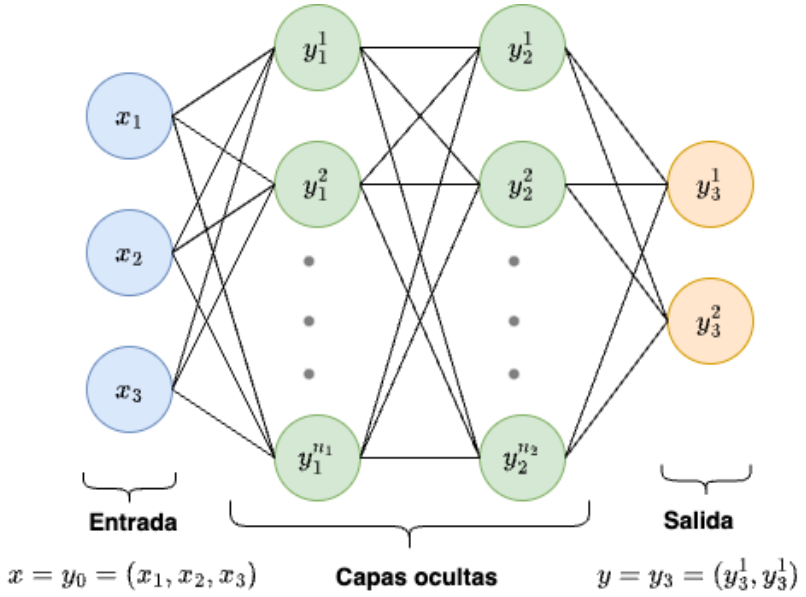


FIGURA 3. Diagrama de una red neuronal con una capa de entrada, 2 capas ocultas y una capa de salida. Omitimos representar el sesgo pero se puede recuperar agregando una neurona constante de valor 1 en cada capa.

- **Función softmax:**

$$\text{softmax}(z) = \left(\frac{e^{z_1}}{\sum_{i=1}^{k} e^{z_i}}, \ldots, \frac{e^{z_k}}{\sum_{i=1}^{k} e^{z_i}} \right).$$

  Donde $z = (z_1, \ldots, z_k)$. Es una generalización de la σ para varias dimensiones.
- **Función ReLU (Rectified Linear Unit):**

$$\text{ReLU}(z) = \max(0, z).$$

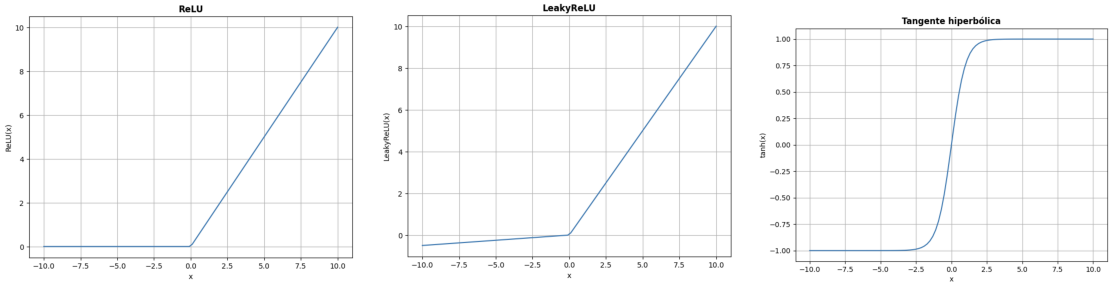
  Es una función a tramos que no modifica los valores positivos pero anula los negativos. Ver gráfico en Figura 4a.
- **Función Leaky ReLU:**

$$\text{LeakyReLU}(z) = \begin{cases} z & \text{si } z \geq 0. \\ \alpha z & \text{si } z < 0. \end{cases}$$

  Donde típicamente α es un valor pequeño como 0.01. Ver gráfico en Figura 4b.
- **Función tangente hiperbólica**

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}.$$

  Ver gráfico en Figura 4c.



(A) ReLU (B) LeakyReLU (C) tanh

FIGURA 4. Funciones de activación: ReLU, LeakyReLU y tanh .

Las funciones sigmoide y softmax suelen usarse exclusivamente en la capa de salida en problemas de clasificación, porque generan una distribución de probabilidad sobre las etiquetas. Las otras se utilizan usualmente en las capas ocultas. En cuanto a la implementación: no hay una función de activación que sirva para todos los casos de uso; a la hora de elegir, se construye sobre lo ya probado o se experimenta con varias funciones y se elige la que consiga mejores resultados.

Nada impide que la sigmoide sea usada en las capas ocultas, pero ha caído en desuso, en [14] se menciona que esto puede deberse a que su derivada lejos de 0 es casi nula (la función se vuelve casi constante) lo que dificulta la optimización de los parámetros mediante métodos de gradiente, fenómeno que también ocurre con la tangente hiperbólica. Se dice que son funciones que saturan sobre 0 y 1.

4.2.4. Expresividad: Teorema de Aproximación Universal.

El problema general que quiere resolver el aprendizaje automático como disciplina se puede resumir en querer aproximar de forma correcta una función que explique el comportamiento de un conjunto de datos de interés. La práctica ha comprobado que las redes neuronales son una herramienta muy poderosa para dicha aproximación.

En esta sección estudiaremos las condiciones teóricas para que las redes neuronales sean buenos aproximadores, en particular, el resultado clásico presentado en "Approximation by Superpositions of a Sigmoidal Function" [6], donde se prueba que funciones continuas pueden ser aproximadas por redes neuronales *feedforward* con una capa oculta, una neurona de salida y funciones de activación como la sigmoide. No es el único teorema de este tipo, variando la arquitectura de la red o las funciones de activación se pueden hacer estudios similares [7] [15].

Denotamos por $I = [0,1]^n$ al cubo en $\mathbb{R}^n$ y por $M(I)$ al espacio de medidas con signo en I. Al espacio $C(I)$ lo definimos como el conjunto de funciones continuas de I en $\mathbb{R}$ y lo dotamos con el producto interno $\langle f, g\rangle = \int_I f(x)g(x)dx$, con la norma asociada a este (la norma L^2) y con la topología inducida por la norma.

Las pruebas de los teoremas que veremos utilizan resultados de teoría de la medida

y análisis funcional, como el resultado de extensión de funcionales lineales continuos de Hanhn-Banach (H-B, para abreviar, es el Teorema 3.6 en [5]). En particular, utilizaremos dos resultados relacionados que también se pueden encontrar en [5] y enunciamos sin demostración a continuación, el primero:

TEOREMA 4.2.1. *(Teorema 3.3 en [5]). Sea X un espacio vectorial y M un subespacio en él, p una semi-norma en X y $\alpha : M \to \mathbb{R}$ tal que:*

$$|\alpha(x)| \leq p(x), \ \forall x \in M.$$

Entonces α se extiende a un funcional lineal $L : X \to \mathbb{R}$ que satisface

$$|L(x)| \leq p(x), \ \forall x \in X.$$

En nuestro contexto la semi-norma p es la norma L^2. El segundo resultado es una forma conveniente de escribir la extensión de H-B, usando como hipótesis que el espacio ambiente sea localmente convexo.

TEOREMA 4.2.2. *(Teorema 3.5 en [5]). Sea X un espacio vectorial normado y localmente convexo, $M \subset X$ subespacio y sea $x_0 \in X$. Si x_0 no está en la clausura de M, entonces existe L funcional lineal tal que $L(x_0) = 1$ pero $L(x) = 0$ para todo $x \in M$.*

También necesitamos la siguiente definición para expresar la condición importante que tienen que cumplir las funciones de activación de las redes neuronales.

DEFINICIÓN 4.2.1. Una función $\sigma : \mathbb{R} \to \mathbb{R}$ se dice *discriminatoria*, si, dada $\mu \in M(I)$, sucede que si

$$\int_I \sigma(\langle y, x \rangle + \theta) d\mu(x) = 0, \ \forall y \in \mathbb{R}^n, \forall \theta \in \mathbb{R},$$

entonces $\mu = 0$

Probamos el siguiente lema que nos será útil:

LEMMA 4.2.3. *Sea $h \in C(I)$ y $\lambda(x)$ la medida de Lebesgue en I. Entonces $\mu(x) = h(x)d\lambda(x)$ es una medida con signo en I.*

PRUEBA. Sea $A \subset I$ medible Lebesgue, escribir $\mu(x) = h(x)d\lambda(x)$ quiere decir que la medida de A según μ se calcula como:

$$\mu(A) = \int_A h(x)d\lambda(x).$$

Ahora, ¿está bien definida esa integral? Por hipótesis h es una función continua y su dominio es un intervalo, por lo tanto su imagen es un intervalo. Entonces h es acotada (y también $|h|$), es decir:

$$\exists M \in \mathbb{R} / \forall x \in I : |h(x)| \leq M.$$

Además, $|h|$ es medible Lebesgue por ser continua, así que $|h|$ es no negativa, medible y acotada con soporte de medida finita (el soporte a lo sumo mide $\lambda(I) = 1$), por lo tanto es integrable Lebesgue. Su integral se define mediante dos funciones no negativas y acotadas; $h^+(x) = \max(h(x), 0)$ y $h^-(x) = \max(-h(x), 0)$. Se tiene $h(x) = h^+(x) + h^-(x)$ y definimos la integral de h como:

$$\int h(x)d\lambda(x) = \int h^+(x)d\lambda(x) + \int h^-(x)d\lambda(x).$$

Entonces μ está bien definida, ya que la integral de h está bien definida. Para afirmar que es una medida, basta con verificar la condición de σ-aditividad de la medida, esto se desprende de las propiedades de la integral, dado un conjunto numerable $\{E_n\}_{n\in\mathbb{N}}$ de conjuntos medibles disjuntos, se tiene:

$$\mu\left(\bigcup_{n=1}^{\infty} E_n\right) = \int_{\bigcup_{n=1}^{\infty} E_n} h(x)d\lambda(x) = \sum_{n=1}^{\infty}\int_{E_n} h(x)d\lambda(x) = \sum_{n=1}^{\infty}\mu(E_n).$$

■

Ahora sí, ya tenemos los ingredientes necesarios para probar el siguiente teorema, que es, junto con 4.2.5, uno de los resultados principales en [6], y nos dice que si la función de activación es discriminatoria, entonces una red neuronal con una capa oculta puede aproximar cualquier función continua en I.

TEOREMA 4.2.4. *(Aproximación Universal) Sea σ continua y discriminatoria. Entonces, las funciones de la forma $g(x) = \sum_{j=1}^{N} \alpha_j\sigma(\langle w_j, x\rangle + b_j)$ con $N \in \mathbb{N}$ son densas en $C(I)$.*

PRUEBA. Sea $G = \{\sum_{j=1}^{N} \alpha_j\sigma(\langle w_j, x\rangle + b_j) : w_j \in \mathbb{R}^N, \alpha_j, b_j \in \mathbb{R}, N \in \mathbb{N}\}$, queremos ver que $\overline{G} = C(I)$. Utilizaremos los teoremas 4.2.1 y 4.2.2, para esto hay que verificar que G es un subespacio vectorial de $C(I)$:
En primer lugar, $C(I)$ es un espacio vectorial puesto que si $f, g \in C(I)$ entonces $\alpha f + \beta g$ es también una función continúa con dominio I.
Ahora, si $g \in G$, entonces es $g(x) = \sum_{j=1}^{N} \alpha_j\sigma(\langle w_j, x\rangle + b_j)$, es decir, que es combinación lineal de las funciones $\{\sigma(\langle w_j, x\rangle + b_j)\}_{j=1}^{N}$. Estas últimas son funciones continuas, puesto que $x \to \langle w_j, x\rangle + b_j$ es lineal (por lo tanto continua) y σ es continua. Así que, en definitva, G es igual al espacio generado por las $\sigma(\langle w_j, x\rangle + b_j) \in C(I)$, por lo que es un subespacio vectorial de $C(I)$.

Supongamos por absurdo que $\overline{G} \subsetneq C(I)$, entonces existe $f_0 \in C(I) - \overline{G}$. Por el Teorema 4.2.2, existe $L : C(I) \to \mathbb{R}$, lineal en el espacio vectorial $(C(I), \mathbb{R})$, tal que $L(f_0) = 1$ y $L(g) = 0$ para todo $g \in G$.
Además, el funcional L cumple que $0 = |L(f)| \leq \|f\|$ para todo $f \in G$, entonces, por el Teorema 4.2.1, se extiende a un $\hat{L}$ en todo $C(I)$ que cumple $|\hat{L}(x)| \leq \|f\|$.
Ahora, como I es compacto y las f son continuas, esto garantiza que $\|f\| < \infty$, por lo tanto $\left\|\hat{L}\right\| < \infty$, donde por $\left\|\hat{L}\right\|$ notamos el máximo de los valores funcionales alcanzados por $\hat{L}$. Es decir que $\hat{L}$ es acotado. Entonces $\hat{L}$ es un funcional lineal acotado definido sobre un espacio métrico, el teorema de representación de Riesz (Proposición 5.4 en [9]) nos dice que existe $h \in C(I)$ tal que $\hat{L}(f) = \langle f, h\rangle$ para todo $f \in C(I)$. Así que nos

queda:

$$\hat{L}(f) = \langle f, h\rangle = \int_I f(x)h(x)dx = \int_I f(x)h(x)d\lambda(x),$$

con la última igualdad hicimos explícito que estamos usando la medida de Lebesgue $\lambda(x)$. Definimos $\mu = h(x)d\lambda(x)$, entonces μ es una medida con signo en I (ver lema 4.2.3) y nos queda $\hat{L}$ definida para toda f como:

$$\hat{L}(f) = \int_I f(x)d\mu(x).$$

Ahora que encontramos una definición de $\hat{L}$ como integral con respecto a una medida, podemos utilizar la hipótesis de que σ es discriminatoria para llegar al absurdo. Observamos que $g(x) = \sigma(\langle w, x\rangle + b) \in G$, entonces, como $\hat{L}(G) = L(G) = 0$, debe ser $\hat{L}(g) = 0$, lo que nos queda:

$$\int_I \sigma(\langle w, x\rangle + b)d\mu(x) = 0 \ \forall w \in \mathbb{R}^n, b \in \mathbb{R}.$$

Ahora, como σ es discriminatoria, esto implica que $\mu = 0$, pero entonces $\hat{L}(f) = 0 \ \forall f \in C(I)$, puesto que integramos con la medida nula. Esto es absurdo, contradice que $\hat{L}(f_0) = L(f_0) = 1$.
El absurdo proviene de suponer que G es un subespacio propio de $C(I)$. Por lo tanto debe ser $\overline{G} = C(I)$ y con esto hemos probado que las funciones de la forma

$$g(x) = \sum_{j=1}^{N} \alpha_j \sigma(\langle w_j, x\rangle + b_j)$$

son densas en $C(I)$. ■

De poco interés sería el resultado anterior si no pudiéramos dar ejemplos de funciones discriminatorias. A continuación vemos un teorema que nos describe toda una familia de este tipo de funciones, que incluye en particular a la sigmoide. Antes de esto, algunas definiciones:

Definición 4.2.2. Una función $\sigma : \mathbb{R} \to \mathbb{R}$ continua es *sigmoidal* si

$$\lim_{t\to-\infty} \sigma(t) = 0 \text{ y } \lim_{t\to\infty} \sigma(t) = 1.$$

Definición 4.2.3. Sea μ una medida en $\mathbb{R}^n$, definimos la transformada de Fourier de μ como $\hat{\mu}$ definida por $\hat{\mu}(m) = \int_{\mathbb{R}} e^{-i\langle m,x\rangle}d\mu(x)$ para todo $m \in \mathbb{R}^n$.

Se puede probar que la transformada de Fourier de una medida es única, así que si dos medidas tienen la misma transformada de Fourier, entonces son iguales. En particular, si $\mu = 0$ es la medida nula, su transformada $\hat{\mu} = 0$.
Ahora veamos que las sigmoidales son discriminatorias, para la prueba necesitamos usar el teorema de convergencia acotada para calcular la integral de funciones acotadas a partir de funciones simples, este teorema se puede encontrar en [9].

Teorema 4.2.5. *Sea σ acotada, medible y sigmoidal. Entonces σ es discriminatoria.*

PRUEBA. Sea σ como en las hipótesis, y supongamos que

$$\int_I \sigma(\langle \alpha, x\rangle + \beta) d\mu(x) = 0,\ \forall \alpha \in \mathbb{R}^n, \beta \in \mathbb{R},$$

para alguna μ medida con signo, queremos ver que $\mu = 0$. Definimos $\sigma_\lambda(x) = \sigma(\lambda(\langle y, x\rangle + \theta) + \phi)$, donde $x, y \in \mathbb{R}^n, \lambda, \theta, \phi \in \mathbb{R}$. Observamos que σ_λ converge en λ a la función $\psi(x)$ dada por:

$$\psi(x) = \begin{cases} 1 & \text{si } \langle y, x\rangle + \theta > 0. \\ 0 & \text{si } \langle y, x\rangle + \theta < 0. \\ \sigma(\phi) & \text{si } \langle y, x\rangle + \theta = 0. \end{cases}$$

Denotamos el hiperplano $H_{y,\theta} = \{x \in \mathbb{R}^n : \langle y, x\rangle + \theta = 0\}$, al semiplano positivo $H^>_{y,\theta} = \{x \in \mathbb{R}^n : \langle y, x\rangle + \theta > 0\}$ y $H^<_{y,\theta}$ al negativo. La función ψ vale 1 en $H^>$, 0 en $H^<$ y $\sigma(\phi)$ en H.
El teorema de convergencia acotada nos permite calcular el límite de la integral de las σ_λ como la integral de ψ, esta última la podemos calcular explícitamente por ser una función simple.

$$\lim_{\lambda\to\infty} \int_I \sigma_\lambda(x) d\mu(x) = \int_I \lim_{\lambda\to\infty} \sigma_\lambda(x) d\mu(x) = \int_I \psi(x) d\mu(x) = \sigma(\phi)\mu(H_{y,\theta}) + \mu(H^>_{y,\theta}).$$

Supusimos que $\int_I \sigma(\langle \alpha, x\rangle + \beta) d\mu(x) = 0$, para todo $\alpha \in \mathbb{R}^n, \beta \in \mathbb{R}$, entonces en particular $\int_I \sigma_\lambda(x) d\mu(x) = 0$, tomando $\alpha = \lambda y$, $\beta = \lambda\theta + \lambda\phi$. El resultado es que la integral de la izquierda en la ecuación anterior da cero y entonces:

$$\sigma(\phi)\mu(H_{y,\theta}) + \mu(H^{>0}_{y,\theta}) = 0\ \forall \phi, \theta, y. \tag{53}$$

Ahora fijamos $y \in \mathbb{R}^n$ y dada una función $h(p)$ acotada y medible, definimos

$$F(h) = \int_I h(\langle y, x\rangle) d\mu(x).$$

Entonces F es un funcional lineal y acotado, definido sobre el espacio de funciones acotadas y medibles. Si h es la función indicatriz del intervalo $[\theta, \infty)$, entonces:

$$\begin{aligned} F(h) &= \int_I h(\langle y, x\rangle)\, d\mu(x) \\ &= \mu\left(\{x \in I : \langle y, x\rangle \geq \theta\}\right) \\ &= \mu\left(\{x \in I : \langle y, x\rangle - \theta > 0\}\right) + \mu\left(\{x \in I : \langle y, x\rangle - \theta = 0\}\right) \\ &= \mu(H^>_{y,-\theta}) + \mu(H_{y,-\theta}) \\ &= 0. \end{aligned}$$

Donde utilizamos la aditivdad de μ y lo probado antes en (53). Del mismo modo, $F(h) = 0$ si h es la indicatriz del intervalo abierto (θ, ∞). Por linealidad de F, $F(h) = 0$ para toda h indicatriz de un intervalo. Por lo tanto, también se anula en toda función simple (combinación lineal finita de indicatrices de intervalos).

En general, si $f : \mathbb{R}^d \to \mathbb{R}$ podemos aproximarla por una sucesión de funciones simples $h_n / \lim_{n\to\infty} h_n = f$ y por convergencia acotada $\lim_{n\to\infty} F(h_n) = F(f)$. Como $F(h_n) = 0$ para todo n, ya que las h_n son funciones simples, entonces $F(f) = 0$ para toda f. Entonces $F = 0$.

Ahora, dado $y \in \mathbb{R}^n$, la transformada de Fourier de μ es, $\hat{\mu}(y) = \int_I e^{-i\langle y,x\rangle} d\mu(x) = F(h)$, donde $h(p) = e^{-ip}$. Esta h es una función acotada y medible, por lo que $F(h) = 0$. Entonces $\hat{\mu} = 0$, lo cual prueba que $\mu = 0$, por unicidad de la transformada. Entonces σ es discriminatoria. ■

OBSERVACIÓN 4.2.1. La sigmoide $\sigma(t) = \frac{1}{1+e^{-t}}$ es acotada, continua (por lo tanto medible) y sigmoidal, por lo que es discriminatoria según el teorema anterior. Por lo tanto, vía el Teorema de Aproximación Universal, redes de una capa que utilicen la sigmoide como función de activacion son buenos aproximadores de funciones continuas en I. Más aún, podemos aproximar cualquier función de *decisión*, es decir, resolver problemas de clasificación.

DEFINICIÓN 4.2.4. Sean $P_1, \ldots, P_k \subset I$ una partición de I. Una función $f : I \to \{1, \ldots, k\}$ es de decisión si $f(x) = j \iff x \in P_j$.

El siguiente resultado nos dice que podemos implementar funciones de decisión con redes de una capa. Usamos el teorema de Lusin que nos habla de cómo aproximar conjuntos medibles por cerrados sin perder demasiada medida en el proceso, también se puede encontrar en [9].

TEOREMA 4.2.6. *Sea σ sigmoidal y continua. Sea f la función de decisión de una partición finita de I. Entonces, para todo $\epsilon > 0$, existe una suma finita de funciones de la forma*

$$g(x) = \sum_{j=1}^{N} \alpha_j \sigma(\langle w_j, x\rangle + b_j)$$

y un conjunto $D \subset I$, tal que $\lambda(I - D) < \epsilon$ y

$$|g(x) - f(x)| < \epsilon, \ \forall x \in D.$$

PRUEBA. Sea $P_1, \ldots, P_k$ una partición de I, es decir que $P_i \cap P_j = \emptyset$ para $i \neq j$ y $\bigcup_{i=1}^{k} P_i = I$. Sea $f : I \to \{1, \ldots, k\}$ tal que $f(x) = j \iff x \in P_j$ una función de decisión.

Notar que f es medible y es finita. Sea $\epsilon > 0$. Por el teorema de Lusin, existe $D \subset I$ tal que $\lambda(I - D) < \epsilon$ y la restricción de $h = f_{|_D}$ es continua.

Entonces $h \in C(I)$, aplicando el teorema de aproximación universal, existe $g(x) = \sum_{j=1}^{N} \alpha_j \sigma(\langle w_j, x\rangle + b_j)$ tal que $|g(x) - h(x)| < \epsilon \forall x \in I$, pues las g son densas en $C(I)$. Por lo tanto, dado $x \in D$ se tiene

$$|g(x) - f(x)| = |g(x) - h(x)| < \epsilon.$$

■

Estos teoremas no mencionan otras arquitecturas, como RNNs o Transformers, tampoco redes *feedforward* más profundas. Además, las variantes que vimos aquí solo sirven para funciones de activación sigmoidales. Sin embargo, aportan argumentos para justificar por qué el aprendizaje profundo es tan útil en la práctica.

Aún más importante es mencionar que estos teoremas solo nos garantizan que si tenemos un problema que puede ser modelado mediante una función continua, entonces existe una instancia de red neuronal de una capa con activación sigmoide que aproxima esa solución tanto como queramos, pero no dicen cómo encontrarla. Este es el problema al que se enfrenta un científico de datos o quién quiera usar una red neuronal en la práctica; y el éxito en el mismo dependerá de varios factores como la calidad de los datos, su cantidad, los hiperparámetros de entrenamiento, etc.

El proceso de encontrar los pesos correctos para definir esa red neuronal que mejor aproxima la que resuelve el problema es justamente lo que se busca con el proceso de optimización que veremos en la siguiente sección.

4.2.5. Optimización en una red neuronal.

La arquitectura de la red por sí sola se puede entender como una plantilla para aproximar una función teórica f^* que no conocemos. Para medir qué tan cerca estamos de tener una buena aproximación se utiliza la función de costo. Para encontrar los parámetros que mejor aproximen se hace como con la RL: se minimiza el costo.

Lo que hacemos es medir qué tan lejos está la salida producida de la salida esperada, denotando a esta cantidad como $L(f(x;\theta), y)$, donde x es la entrada, y es la salida esperada y f representa a la red con parámetros θ. Esa cantidad es una variable aleatoria, donde la aleatoriedad está dada por (x, y).

El problema de optimización es encontrar los parámetros θ que minimizan la esperanza de la función de costo. Esto es exactamente lo planteado en la Sección 3.3.1.1 sobre riesgo empírico en un contexto general. En este caso el riesgo empírico del modelo f escrito como función de los parámetros θ es:

$$R_{emp}[f](\theta) = \frac{1}{n}\sum_{i=1}^{n} L(f(x_i;\theta), y_i). \tag{54}$$

Como ya vimos, esta cantidad es susceptible de ser minimizada mediante el algoritmo de SGD. Las funciones de costo usuales son la entropía cruzada para clasificación y el error cuadrático para la regresión. Por ahora no hay nada nuevo, ya hablamos de optimización en general y para la regresión logística en particular, la diferencia sustancial es la función de modelado: la f definida por una red neuronal es típicamente muy compleja. Por lo que el riesgo empírico termina siendo mucho más difícil de minimizar que en el caso de la RL.

Esta complejidad ha dado lugar al estudio y experimentación de cómo mejorar el proceso

de optimización, manipulando los hiperparámetros de entrenamiento, que no son valores a optimizar, sino ciertas decisiones que se toman acerca de cómo realizar el descenso por gradiente. Cambios en estos hiperparámetros pueden afectar la convergencia del algoritmo, la calidad final del modelo, el tiempo de entrenamiento, etc. Describimos algunos de ellos a continuación.

4.2.5.1. *Elección de hiperparámetros.*

Batch size

Al utilizar SGD estimamos en cada paso el gradiente $\nabla f(x)$ por un gradiente estocástico $g(x, \xi)$. Según lo planteado en la sección sobre SGD en el capítulo 3, la variable aleatoria ξ consiste en tomar aleatoriamente un dato del conjunto de entrenamiento. Una extensión del método anterior es seleccionar aleatoriamente un subconjunto de cardinal m y calcular g como promedio de gradientes en cada punto.
El primer método se conoce como *online learning* y el segundo como *mini-batch learning*. Se suele utilizar el segundo a menos que aprender durante la ejecución del algoritmo sea crítico para el problema (de ahí que se le denomine *online* al primero).

La razón principal para tomar $m < N$ en (54) y no tomar toda la muestra, es que así se ahorra en cómputo sin perder demasiada precisión. Para entender por qué no se pierde demasiada precision, pensemos lo siguiente: asumamos que la distribución de los datos fuera normal, entonces el promedio calculado en (54) representa la media muestral y el error estándar es $\frac{\sigma}{\sqrt{m}}$ siendo σ el desvío estándar verdadero (que desconocemos). Aumentar m disminuye el desvío pero a una razón peor que la lineal, es decir, si aumentamos m 100 veces, solo podemos esperar en el orden de 10 veces menos desvío.

Por otro lado, un m muy pequeño puede hacer que la estimación $g(x, \xi)$ sea muy ruidosa, aunque vale mencionar que este ruido puede funcionar como elemento de regularización que ayude a evitar el *overfitting* durante el entrenamiento. Balancear estas dos cuestiones es parte del trabajo a realizar al entrenar. Al enfrentarse a un problema por primera vez, lo usual es utilizar valores ya probados en la literatura para arquitecturas similares a la trabajada y experimentar.

Por último, algunos componentes de hardware donde se ejecutan los algoritmos de optimización se benefician de ciertos tamaños específicos de *batch size*, especialmente utilizando hardware específico para cómputo matricial como GPUs o TPUs[2], donde es común utilizar tamaños de batch que sean potencias de 2. Así, valores típicos pueden oscilar entre 32 y 256. El hardware también impone limitaciones como la memoria con la que se cuenta al momento de entrenar; si se trabaja con datos pesados, por ejemplo imágenes, puede ser que no se puedan tener en memoria una cantidad grande como 256 datos a la vez, junto con todos los pesos de la red, esto obligaría a disminuir el *batch size*.

[2]Las GPUs fueron inicialmente diseñadas para procesar gráficos en computadora y por eso son tan buenas operando con matrices. Las TPUs (Tensor Processing Unit) son un hardware específico para operar con modelos de aprendizaje automático diseñado por Google.

Learning rate

Vamos a estudiar la elección del paso α en el algoritmo de SGD. En la Sección 3.2.1 estudiamos teoremas de convergencia y hablamos sobre qué sucede según el valor de α. Un estudio teórico más detallado sobre las redes podría intentar probar algún teorema que vincule el α con la tasa de convergencia o la estimación del error en el resultado del SGD, similares a 3.3.1 y 3.3.2, que no aplican porque suponen convexidad. Un estudio así no lo haremos en este trabajo, pero es una línea de trabajo futuro interesante.

Lo que podemos decir es que son funciones continuas (las presentadas en este trabajo al menos) y son diferenciables en casi todo punto[3], por lo que podemos aplicar métodos basados en gradiente como el SGD.

Más allá de lo teórico, en la práctica el α es un hiperparámetro más y en muchos casos se encuentra experimentalmente. La intuición que hay que tener es que si α es pequeño vamos a tardar más en converger, pero una vez dentro del entorno de un mínimo local, es difícil que salgamos de él. En cambio, si el α es grande vamos a avanzar rápido hacia un posible mínimo pero corremos el peligro de 'pasarlo de largo' o de quedar cerca pero un siguiente paso irse lejos.

Para tener un balance entre la precisión de un α pequeño y la velocidad de convergencia de uno grande, se suele usar una estrategia de α decreciente, que se disminuye en cada batch o cada cierta cantidad de batches. Lo que se intenta es disminuir rápido el valor inicial del costo con la esperanza de entrar en la cuenca de atracción de un mínimo, momento para el cual la precisión del paso pequeño es más deseada.

La regla de Armijo es otro criterio posible, son también comunes métodos adaptativos como Adagrad [11] y Adam [13]. Este tipo de algoritmos cambian el paso dinámicamente, de ahí que sean *adaptativos*, y pueden llegar a usar información del historial de gradientes, no solo del gradiente en el batch actual, en ese caso se denominan métodos con memoria o de *momentum*.

Epochs

Si el *batch size* es igual a m, entonces la red toma m datos en cada paso del SGD, y si el conjunto de entrenamiento tiene N datos, entonces la red ve todos los datos en N/m pasos. A esa pasada completa por el conjunto de entrenamiento se le llama época o *epoch*. Como una sola época no suele ser suficiente para que todos los pesos de la red se ajusten, se hacen varias pasadas. La elección de la cantidad de épocas es un hiperparámetro más. Si son muy pocas nos perdemos de modelar de mejor manera la distribución de datos (*underfitting*) pero si son muchas corremos el riesgo de sobreajustar el modelo al conjunto de entrenamiento y perder capacidad de generalización (*overfitting*).

Es una práctica común monitorear alguna métrica, medida sobre un conjunto de datos no visto por el modelo, al final de cada época y parar si la métrica no mejora luego de cierta cantidad t de épocas, donde t es también un hiperparámetro. A esto se le llama

[3]Depende de las funciones de activación, que pueden tener puntos de no diferenciabilidad, como la ReLU en 0.

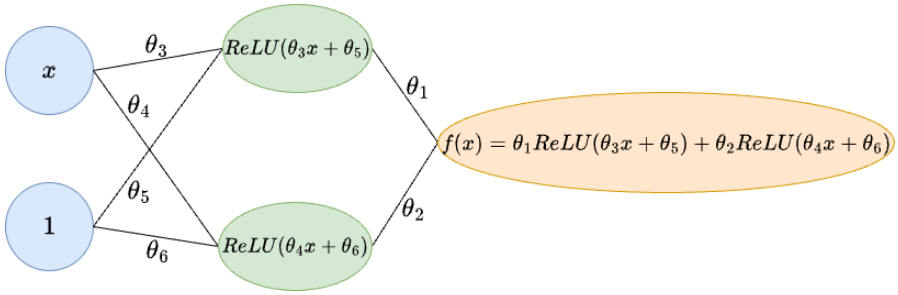


FIGURA 5. Representación de la red $f(x) = \theta_1 ReLU(\theta_3 x + \theta_5) + \theta_2 ReLU(\theta_4 x + \theta_6)$. La segunda capa no tiene sesgo.

early stopping y es una técnica de regularización, puesto que combate el sobreajuste.

4.2.5.2. *Desafíos en la optimización de redes neuronales.*
Entrenar exitosamente[4] una red neuronal tiene dificultades, para empezar la cantidad de parámetros suele ser muy grande y la función de costo en general no es convexa.

A continuación mencionamos problemas que pueden surgir al optimizar, algunos son propios de estar tratando con redes y otros, como la explosión del gradiente o el mal condicionamiento, podrían suceder en otros contextos de optimización.

No convexidad
Vamos a ver mediante un ejemplo que el riesgo empírico (54) no es convexo en general.

EJEMPLO 4.2.1. Sea $f : \mathbb{R} \to \mathbb{R} / f(x) = \theta_1 ReLU(\theta_3 x + \theta_5) + \theta_2 ReLU(\theta_4 x + \theta_6)$. Esta función representa una red neuronal como la de la Figura 5, tiene una capa oculta con funciones de activación ReLU en ella y la identidad como activación en la capa de salida.

Supongamos que los datos a modelar corresponden a un problema de regresión, unidimensional en la entrada como en la salida, por ejemplo predecir el precio de una casa en función de sus metros cuadrados. Una función de costo válida para este caso es el error cuadrático:

$$L(f(x;\theta), y) = \|f(x;\theta) - y\|^2 = |f(x;\theta) - y|^2.$$

El riesgo empírico a minimizar en función de $\theta = \{\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6\}$ es:

$$R_{emp}[f](\theta) = \frac{1}{n} \sum_{i=1}^{n} |f(x_i;\theta) - y_i|^2.$$

Consideramos ahora dos puntos en el espacio de parámetros, $\overline{\theta} = (1, -1, 1, 1, 1, 0)$ y $\hat{\theta} = (-1, 1, 1, 1, 0, 1)$. Se puede chequear rápidamente que $f(x;\overline{\theta}) = f(x;\hat{\theta}) = ReLU(x +$

[4]La definición de 'éxito' puede variar de un problema a otro pero digamos que nos referimos a tener un costo bajo en conjuntos de test, representativos del problema, que la red no haya visto durante entrenamiento

$1) - ReLU(x)$. Es decir que la red neuronal realizada en cada uno de estos puntos es la misma.

Por otro lado, $\frac{\overline{\theta}+\hat{\theta}}{2} = (0, 0, 1, 1, \frac{1}{2}, \frac{1}{2})$ y también es fácil ver que $f(x; \frac{\overline{\theta}+\hat{\theta}}{2}) = 0$. Es decir que la red neuronal para este vector de parámetros es la función idénticamente nula. El punto se encuentra en el segmento que une a $\overline{\theta}$ y $\hat{\theta}$ incluido en $\mathbb{R}^6$. Lo que vamos a hacer ahora es dar un conjunto de datos que viola la definición de convexidad para $R_{emp}[f]$.

Supongamos que $S = \{(-1, 0), (1, 1)\}$ es el conjunto de datos. Entonces se tiene que:

$$R_{emp}[f](\overline{\theta}) = \frac{1}{2}(f(-1; \overline{\theta}) - 0)^2 + (f(1; \overline{\theta}) - 1)^2 = 0.$$

En esto usamos que $f(-1; \overline{\theta}) = 0$ y $f(1; \overline{\theta}) = 1$, que se calcula directamente usando la definición de la ReLU. De la misma forma se puede calcular que $R_{emp}[f](\hat{\theta}) = 0$. Sin embargo, en el punto intermedio se tiene que:

$$R_{emp}[f]\left(\frac{\overline{\theta} + \hat{\theta}}{2}\right) = \frac{(-1)^2}{2} = \frac{1}{2}.$$

De esta manera encontramos dos puntos $\overline{\theta}$ y $\hat{\theta}$ tales que

$$R_{emp}[f]\left(\frac{1}{2}\overline{\theta} + \frac{1}{2}\hat{\theta}\right) = \frac{1}{2} > 0 = \frac{1}{2}R_{emp}[f](\overline{\theta}) + \frac{1}{2}R_{emp}[f](\hat{\theta}),$$

esto contradice la definición de convexidad.

Alguien podría decir que el ejemplo anterior es de juguete, y tendría razón: las aplicaciones reales tienen mucho más que 6 parámetros. Sin embargo, dentro del ejemplo está escondido un argumento que basta para mostrar en general por qué las redes no son convexas, o al menos una razón por la cual no lo son. Si nos fijamos, la única diferencia entre $\overline{\theta}$ y $\hat{\theta}$ es que se intercambiaron las neuronas de la capa oculta, se dio una permutación de parámetros $(1 \leftrightarrow 2, 3 \leftrightarrow 4, 5 \leftrightarrow 6)$. Ahora, esta acción de intercambiar dos neuronas siempre da lugar a la misma red *feedforward*. Por lo tanto, si un punto θ es mínimo local para $R_{emp}[f]$, entonces también lo es un parámetro θ' que represente la permutación de neuronas en f, entonces $R_{emp}[f](\theta) = R_{emp}[f](\theta')$, pero no necesariamente $R_{emp}[f](t\theta + (1-t)\theta') \leq tR_{emp}[f](\theta) + (1-t)R_{emp}[f](\theta')$.

Mal condicionamiento

Es el problema que ya presentamos en detalle en la sección de optimización, cuando hablamos de la elección de la dirección de descenso. Lo que nos interesa es ver qué pasa con $\kappa(H(\theta))$ donde $H(\theta)$ es la matriz Hessiana de la función de costo en el punto θ del espacio de parámetros. Recordar que $\kappa(H(\theta)) = \frac{\lambda_{max}}{\lambda_{min}}$, donde λ_{max} y λ_{min} son el valor propio máximo y mínimo de $H(\theta)$, respectivamente.
Si ese número es muy grande, entonces da un indicio de que los valores propios son muy dispares y esto resulta en una reducción lenta de la función de costo, como comentamos en el ejemplo 3.2.1.

En el contexto general de optimización, dijimos que el método de Newton era una solución posible al problema. En las redes neuronales no disponemos de información de segundo orden, o en todo caso, suele ser muy costoso calcularla, entonces no podemos utilizar exactamente las mismas técnicas. La búsqueda de algoritmos de optimización eficientes para redes neuronales es un área de investigación activa.

Explosión del gradiente
Cambios bruscos en el valor de la función de costo también son un problema: si en un entorno de θ tenemos valores funcionales mayores a $R_{emp}[f](\theta)$, imaginemos una elevación abrupta en el gráfico, corremos el peligro de arribar allí en el paso siguiente.

Este fenómeno se puede volver más frecuente en algunas arquitecturas de red, como redes recurrentes, donde las dependencias temporales hacen que una misma matriz de pesos W se multiplique por sí misma repetidas veces. Lo cual termina haciendo que el gradiente de toda la red explote o se desvanezca, en función de los valores propios de W. Ver la Sección 5.2 y el capítulo de redes recurrentes en [14] para más detalles.

Para mitigar estos problemas, una técnica burda pero efectiva es la de acotar el gradiente, es decir, acotamos $\|g(x,\xi)\|$ para que en ningún paso del algoritmo sea mayor a cierto M. La constante M no tiene por qué ser global a todo el entrenamiento, puede ser calculada en cada batch como un porcentaje de $\|g(x,\xi)\|$. A esta técnica se le llama *gradient clipping*.

Estancamiento
El SGD intentará llevarnos a un mínimo local, pero que puede no ser global, en ese caso nuestro proceso de optimización quedaría estancado en ese punto subóptimo. En principio, hay que acostumbrarse a esta dificultad, la existencia de mínimos locales no globales es consecuencia de la no convexidad, que ya vimos que es una propiedad común en $R_{emp}[f]$ si f es una red. Sin embargo, se menciona en [14] que los mínimos locales en funciones de alta dimensionalidad son muy raros; de hecho, la mayoría de los puntos críticos van a ser puntos silla.

La intuición explicada allí es que si pensamos a la función de costo como una $L : \mathbb{R}^n \to \mathbb{R}$ aleatoria[5], entonces para que un punto θ^* sea mínimo, la Hessiana $H(\theta^*)$ debe ser definida positiva, es decir, todos los valores propios deben ser positivos. Este va a ser intuitivamente un evento muy raro en una matriz aleatoria. Por otro lado, para un punto silla nos basta con que haya tanto valores propios negativos como positivos, que es un evento mucho más probable a medida que la dimensión n crece.

Pasaje local-global
El SGD es un método que dispone de información local al punto θ dentro del espacio de parámetros en el que estamos optimizando. En este sentido se puede ver también como

[5]Lo es, ya que su valor depende de los parámetros pero también del valor de la red en cada punto aleatorio de la muestra de datos.

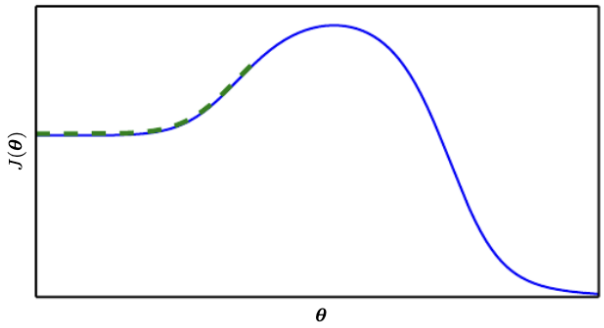


FIGURA 6. Ejemplo de un problema de pasaje de un mínimo local a uno global. Si nuestro punto inicial está donde empieza la linea punteada, nunca vamos a alcanzar el mínimo.

un algoritmo *greedy*, toma decisiones a corto plazo que son las mejores con la información que se tiene en el momento. El problema es que nada garantiza que la dirección opuesta al gradiente en el punto θ nos lleve a un mínimo local de bajo costo, ni mucho menos a uno global. Ver un ejemplo sencillo como el de la Figura 6.
El punto en el que inicializamos el algoritmo juega un papel importante. Sin embargo, los algoritmos funcionan en la práctica aunque el punto inicial se elija de manera aleatoria. En [14] se explica que la alta dimensionalidad del espacio de parámetros puede ser una razón para que el SGD sea efectivo a pesar del problema de pasaje local-global, intuitivamente porque siempre tenemos alguna dirección por la cual seguir descendiendo, entonces no se da el problema de la figura donde la función representada es de una dimensión.

4.2.6. Cálculo del gradiente.

Hemos hablado del gradiente de la función de costo, que implica el cálculo del gradiente de la función f que define la red. Siguiendo la ecuación (52), vemos que una red es una composición de funciones, por lo tanto, podemos utilizar la regla de la cadena para calcular la derivada de f con respecto a cada parámetro. Matemáticamente no hay nada más que decir, es una función vectorial definida a partir de la composición de funciones continuas, diferenciables y podemos diferenciar de la manera usual.

Sin embargo, en la práctica, el uso *naive* de la regla de la cadena nos lleva a cómputo redundante, como muestra el siguiente ejemplo sencillo extraído de [14].

EJEMPLO 4.2.2. Sean $f : \mathbb{R} \to \mathbb{R}$ una función diferenciable y $w \in \mathbb{R}$. Definimos $x = f(w)$, $y = f(x)$, $z = f(y)$. Si aplicamos la regla de la cadena para calcular $\frac{dz}{dw}$ obtenemos:

$$\begin{aligned}\frac{dz}{dw} &= \frac{dz}{dy}\frac{dy}{dx}\frac{dx}{dw} \\ &= f'(y)f'(x)f'(w) \\ &= f'(f(f(w)))f'(f(w))f'(w).\end{aligned}$$

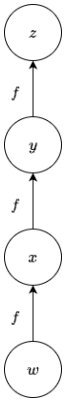


FIGURA 7. Grafo computacional del ejemplo 4.2.2.

En la última igualdad vemos que $f(w)$ se calcula dos veces, lo cual es un gasto de computo innecesario si consideramos que podemos guardar el valor de $f(w)$ y reutilizarlo cada vez que sea necesario.

Esto motiva el uso de un algoritmo particular para calcular el gradiente de una red neuronal, llamado *backward propagation*, que se basa en la observación de que podemos guardarnos resultados intermedios en la aplicación de la regla de la cadena para reutilizarlos. De esta manera, gastamos memoria para ahorrar en cómputo. Para darle un marco teórico al algoritmo, introducimos el concepto de grafo computacional.

4.2.6.1. *Grafo computacional.*
Un grafo computacional (GC) es una manera de representar una función f como composición de otras funciones. En este grafo, los nodos son variables, que pueden ser de distintos dominios: reales, vectores, matrices. Y los conjuntos de aristas representan operaciones entre estas variables. Las operaciones pueden ser complejas, como una función de activación, o elementales, como la suma o el producto. En la Figura 7 se encuentra el GC del ejemplo 4.2.2 y en la Figura 8 hay un ejemplo más complejo.
Estos grafos son dirigidos. Para computar la función f que representa un GC, tomamos los nodos sin aristas entrantes como nodos iniciales y recorremos el grafo aplicando sucesivamente las funciones representadas en las aristas, hasta llegar a los nodos finales, definidos por no tener aristas salientes.

A continuación pasamos a describir formalmente como se computa la función que representa la GC y cómo usar el grafo para calcular el gradiente de la función.

Forward propagation
Consideremos un grafo computacional G que representa una función f con nodos $\{u^i\}_{i=1,\dots,n}$ siendo $u^1, \dots, u^p$ los nodos que corresponden a las variables de entrada, y el resto de los nodos son variables intermedias y de salida. Los argumentos de un nodo u^i serán todos los nodos con aristas salientes que incidan en u^i, denotamos a este conjunto como A^i. Denotamos como f^i la transformación que lleva A^i en u^i, es decir que $u^i = f^i(A^i)$.

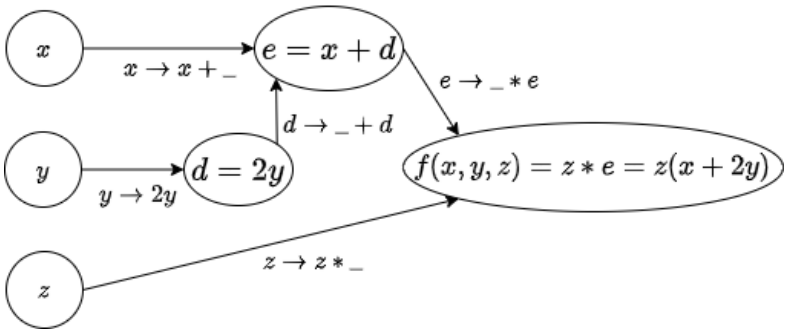


FIGURA 8. Grafo computacional de la función $f(x, y, z) = z(x + 2y)$. Los nodos son valores reales y las aristas representan funciones de una variable. El producto y la suma se representan mediante dos aristas. En general, una función n-aria se representa por n aristas.

Supongamos que la notación está hecha de tal forma que los argumentos para calcular u^i están indexados antes que i, o sea:

$$A^i \subset \{u^j : j < i\}. \tag{55}$$

El cálculo de $f(x)$ se realiza con el siguiente algoritmo.

Result: f(x)
for $i = 1, \ldots, p$ **do**:
    $u^i = x_i$
endfor
for $i = p + 1, \ldots, n$ **do**:
    $u^i = f^i(A^i)$
endfor
return u^n

Algoritmo 2: Forward propagation

Backward propagation

El grafo G tiene su grafo $\hat{G}$ inverso que se obtiene invirtiendo el orden de las aristas. Este grafo sirve para representar las derivadas de f con respecto a las distintas variables representadas en cada nodo. Si en G el cálculo se realiza en un sentido, en $\hat{G}$ se realiza en el sentido opuesto. El grafo $\hat{G}$ será útil para calcular el gradiente de f mediante el algoritmo de *backpropagation*.

Supongamos que queremos calcular el gradiente de una f con codominio en $\mathbb{R}$, entonces su GC tiene un solo nodo de salida, que siguiendo la convención de notación, es el nodo u^n. Vamos a mantener un registro de las derivadas parciales ya calculadas, para así evitar redundancia. Definimos una lista `grads` donde `grads[i]` guarda el valor $\frac{\partial u^n}{\partial u^i}$, es decir, la derivada de la salida con respecto a la variable en el nodo i. El algoritmo es el

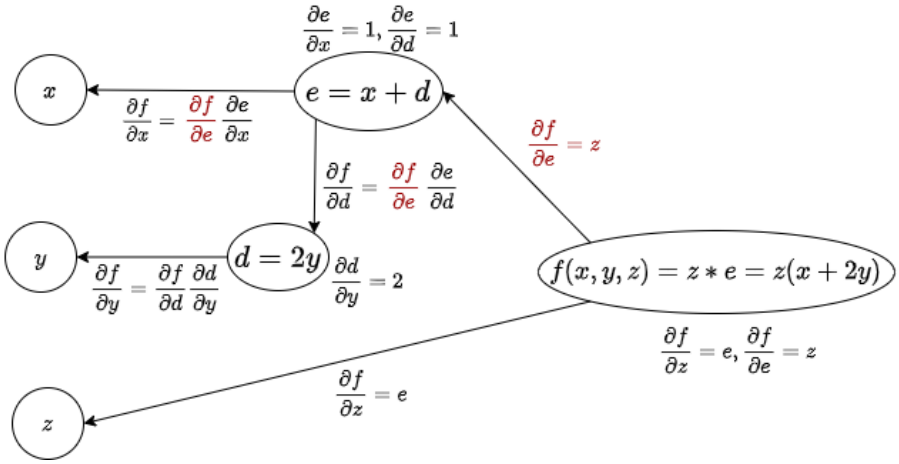


FIGURA 9. Grafo computacional inverso de la funcion $f(x) = z(x + 2y)$ representando el computo de las derivadas parciales. Notar que $\frac{\partial f}{\partial e} = z$ se necesita tanto para el cálculo de $\frac{\partial f}{\partial x}$ como para el de $\frac{\partial f}{\partial y}$, al usar *backpropagation* se calcula una sola vez y se guarda el resultado.

siguiente:

Correr el algoritmo de forward propagation para obtener u^n
Inicializar `grads[n]:=1`
for $j = n - 1$ down to 1 **do**:
    `grads[j]` := $\sum_{u^i / j \in Parents(u^i)} \text{grads}[i] \frac{\partial u^i}{\partial u^j}$
endfor
return *grads*

Algoritmo 3: Backward propagation

La sección `grads[0:p]` de la lista es $(\frac{u^n}{u^1}, \ldots, \frac{u^n}{u^p})$, que es exactamente $\nabla f(x)$ porque $u^n = f(x)$ y $u^i = x_i$. En la Figura 9 visualizamos el cálculo de las derivadas parciales en el GC de la Figura 8.

Las redes neuronales tienen muchos parámetros y una misma derivada puede usarse muchas veces en el cálculo del gradiente, por lo que el uso de estos algoritmos es fundamental para el realizar un cálculo eficiente. El grafo computacional de una red es extenso, pero se construye de igual manera que en los ejemplos presentados. Referimos a [14] para ver ejemplos completos de grafos computacionales para una red neuronal sencilla.

CAPÍTULO 5

Aprendizaje profundo aplicado al PLN

5.1. Redes Neuronales Feedforward

En esta sección vamos a profundizar en aspectos de implementación de redes FNN para problemas específicos de PLN. Dependiendo del problema a resolver, la definición de la tarea a aprender por el modelo (la red) cambia. Nosotros nos centraremos en dos problemas en particular, que son muy usuales: clasificación y generación de texto.
El tipo de problema influye en cuestiones prácticas de la arquitectura como la cantidad de neuronas en la capa de salida o en qué funciones de activación usar. Por ejemplo, si hay que clasificar, habría que tener una neurona por clase en la capa de salida y usar entropía cruzada como función de costo. Si, en cambio, hablamos de generar texto, una posibilidad es modelar el problema como uno de clasificación sobre todos los tokens vocabulario, y en ese caso tendríamos una neurona por token, con función de activación softmax y entropía cruzada como función de costo.
Más allá del problema a resolver y la arquitectura elegida para hacerlo, debemos primero procesar el texto y llevarlo a un dominio que la red pueda entender, es decir, vectores.

5.1.1. Mapear textos a vectores.

Lo primero que hay que hacer al enfrentarse a un problema de PLN es lo que se llama *preprocesamiento*. Se pueden normalizar palabras (llevarlas a una raíz en común), eliminar *stopwords* (palabras muy comunes), eliminar signos de puntuación, etc. Cuáles de estas tareas se realizan depende de si tienen sentido para el problema o no.

Luego se transforma el texto en *tokens*, es decir, lo dividimos en elementos atómicos, pueden ser palabras enteras, *subwords*[1] o hasta caracteres individuales. Aquí las expresiones regulares del Capítulo 2, Sección 2.1, juegan un papel importante, pero también se utilizan métodos puramente estadísticos para identificar tokens relevantes automáticamente.

Por último, hay que asignarle a cada token un vector en $\mathbb{R}^n$ que lo represente y que sea una entrada válida para la red. Estos vectores de *features* se obtienen de varias maneras; la manera más 'artesanal' implica pensar en definiciones de features para computarlas, como hablamos también en el Capítulo 2 en la sección de expresiones regulares.

Otra manera, y que evita el trabajo de definir features particulares para cada problema, es usar métodos de conteo de tokens. Supongamos que tenemos documentos (textos) y que el vocabulario V, que incluye a todos los tokens de todos los documentos, está

[1]Tokens más pequeños que una palabra o signo, no tienen significado por si mismo pero son elementos que se repiten con frecuencia. Como los sufijos 'endo' o 'ismo' en español.

indexado. El más simple de los métodos de conteo se conoce como *one-hot encoding*, donde a cada documento se le asigna un vector cuya entrada i-ésima es 1 si el i-ésimo token del vocabulario está presente en el texto, y es 0 en otro caso. Una mejora sobre lo anterior es codificadar no solo la presencia del token, sino la cantidad de veces que aparece en cada documento, en ese caso hablamos de un *bag of words* (BoW). Un paso más de complejidad es relativizar el conteo del BoW dividiendo cada entrada i por la cantidad de documentos donde esté presente el i-ésimo token. Ver Capítulo 6 de [29] para más detalles.

El tercer método para representar textos utiliza el concepto de *representation learning*, que se refiere a utilizar las mismas features del problema para 'aprender' una representación de las mismas. No es un enfoque *programable*, como las features artesanales o de conteo. Convertimos la tarea de generar la representación en parte del problema de aprendizaje automático. Es un concepto transversal al aprendizaje automático, y para el caso de textos en particular, vale la pena comentar el algoritmo de Word2Vec [12]; este método consiste en entrenar una red neuronal para que prediga un token a partir de los tokens que lo rodean (su contexto). Como resulado del entrenamiento de esta red se producen vectores que representan a cada token y que capturan la semántica del mismo, en función de su uso en el lenguaje (ver también Capítulo 6 de [29] para más detalles). Esta idea de 'dime con quién andas y te diré quién eres', pero aplicada a palabras, existe desde mediados del siglo pasado y se le llama 'distributional hypothesis' [2][3] y lo que dice es que las palabras que aparecen en contextos similares tienen significados similares.

La función que mapea tokens a vectores se le llama *embedding*, palabra que también se puede usar para hablar de la representación particular de cada token: al token w le corresponde el embedding e_w que será un vector de $\mathbb{R}^n$. Podemos arribar a un embedding de forma manual, por conteo o con representation learning, pero en general la palabra se reserva para los vectores generados por el último método; seguiremos esta convención. Una ventaja de métodos dentro de esta última categoría es que sus representaciones son densas y de baja dimensión, en oposición a las dadas por conteo de tokens, cuya dimensión es igual a $|V|$ (el cardinal del vocabulario) y son muy ralas (usualmente un documento tiene solo una fracción de todas las palabras en el corpus).

Un embedding $E : V \to \mathbb{R}^n$, siendo V el vocabulario, que capture correctamente la semántica de las palabras puede tener propiedades interesantes, por ejemplo, se cumplen ecuaciones algebraicas como $E(\text{rey}) - E(\text{hombre}) + E(\text{mujer}) \approx E(\text{reina})$ (ver Figura 1). La función E logra asociar ciertas direcciones del espacio con conceptos como género o particularidades como en este caso la propiedad de 'ser de la realeza', capturada por la dirección $E(\text{rey}) - E(\text{hombre})$.

5.1.2. Clasificación de texto.

Este es un problema que ya hemos visitado cuando hablamos de métodos basados en reglas y de la regresión logística, y consiste en asignar una etiqueta a un texto de un conjunto de etiquetas pre-establecidas. La FNN utilizada para resolver estos problemas

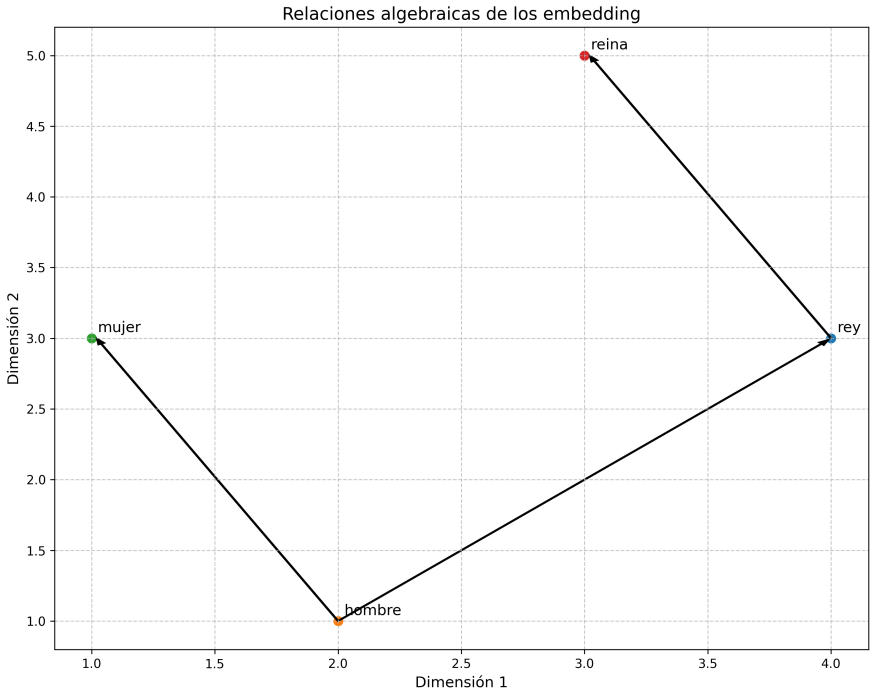


FIGURA 1. Representación en 2 dimensiones del algebra de embeddings. El vector $E(\text{rey}) - E(\text{hombre}) + E(\text{mujer})$ apunta a $E(\text{reina})$. En el ejemplo las magnitudes de $E(\text{mujer}) - E(\text{hombre})$ y $E(\text{reina}) - E(\text{rey})$ son iguales pero podrían solo ser vectores colineales. Los embedding suelen tener una cantidad de dimensiones mucho mayor a 2 (por ejemplo, 100).

tiene una neurona por cada etiqueta posible en la capa de salida, con una función softmax de activación para que la salida sea una probabilidad entre las clases, excepto para el caso binario donde podemos usar una sola neurona con una sigmoide.

Ya tenemos todas las herramientas para que la red pueda consumir un texto s, solo hay que aplicarlas en el orden adecuado. Primero, normalizar textos y tokenizar, con esto vamos a tener s definido como una secuencia de tokens $s = w_1, \ldots, w_k$. Luego, usando una función de embedding, s se convierte en una secuencia $s' = e_1, \ldots, e_k$ de vectores. La función que mapea textos en vectores podría tenerse de antemano (p. ej. un Word2Vec *pre-entrenado*) o entrenarse junto con la red que resuelve el problema de PLN que nos interesa. La secuencia s' es recibida por la red en forma de un único vector, producto de la concatenación de todos los e_i. La capa de entrada queda entonces de largo $k \times d$, siendo d la dimensión del embedding.

El problema con el enfoque descripto anteriormente es que el largo de cada texto s, es decir la cantidad de tokens k, no está fija en la mayoría de los problemas reales. Queremos soportar textos de largo variable. Sin embargo, la capa de entrada de una

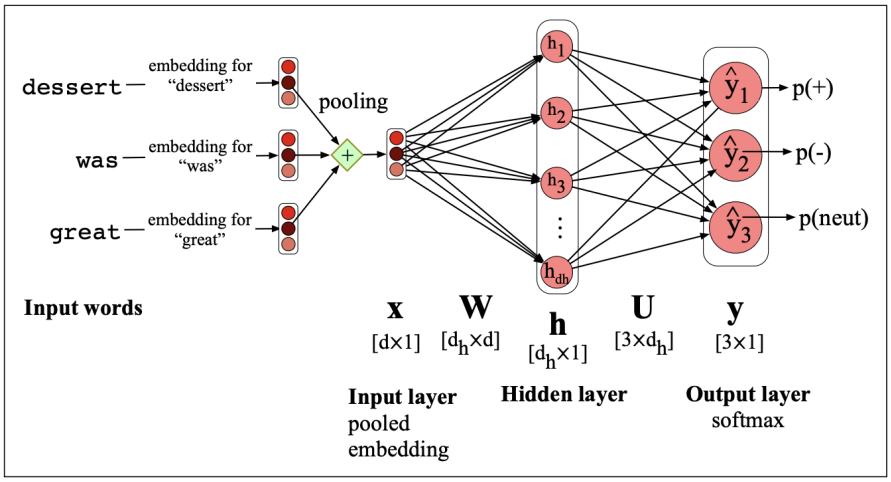


FIGURA 2. FNN de una capa oculta para clasificación de reseñas de restaurantes en positivas, negativas o neutras. Utiliza un embedding preentrenado que genera vectores de dimensión d y una capa de pooling para manejar textos de largo variable. W y U son matrices de pesos. Imagen de [29].

FNN tiene una dimensión fija.

Para solucionar el problema de la dimensión fija se pueden utilizar dos enfoques. El primero es decretar que los textos tienen un tamaño fijo, definiendo k como un hiperparámetro más. Los textos con más de k tokens se truncan, mientras que los que son más cortos se 'rellenan' con un token especial reservado para este propósito hasta conseguir el largo requerido (a esta acción se le llama *padding*). El segundo enfoque consiste en utilizar alguna función de agregación, también llamada de *pooling*, como pueden ser el promedio o la suma de los embeddings y alimentar así a la red con un vector de tamaño d fijo. En la Figura 2 se ilustra una FNN de una capa oculta utilizando la técnica de pooling. El problema es que en cualquiera de los dos enfoques, ya sea padding o pooling, se pierde información, y en el caso de pooling también se pierde el orden de los tokens. La arquitectura de redes recurrentes, que veremos más adelante, intenta solucionar este problema (aunque da lugar a otros).

Ahora sí, preprocesando, vectorizando y aplicando padding o pooling, podemos alimentar a una red con la representación de un texto. Transformamos el dominio de textos a vectores en $\mathbb{R}^d$ y estamos en condiciones de entrenar la red y posteriormente usarla para inferencia. Al momento de inferir, es decir, aplicar la red en nuevos textos no vistos durante el entrenamiento, hay que aplicar los mismos pasos de procesamiento a los textos para obtener los embeddings previo a aplicar la red.

5.1.3. Generación de texto.

Otro problema central en PLN es el de generación de texto, que ya lo estudiamos cuando

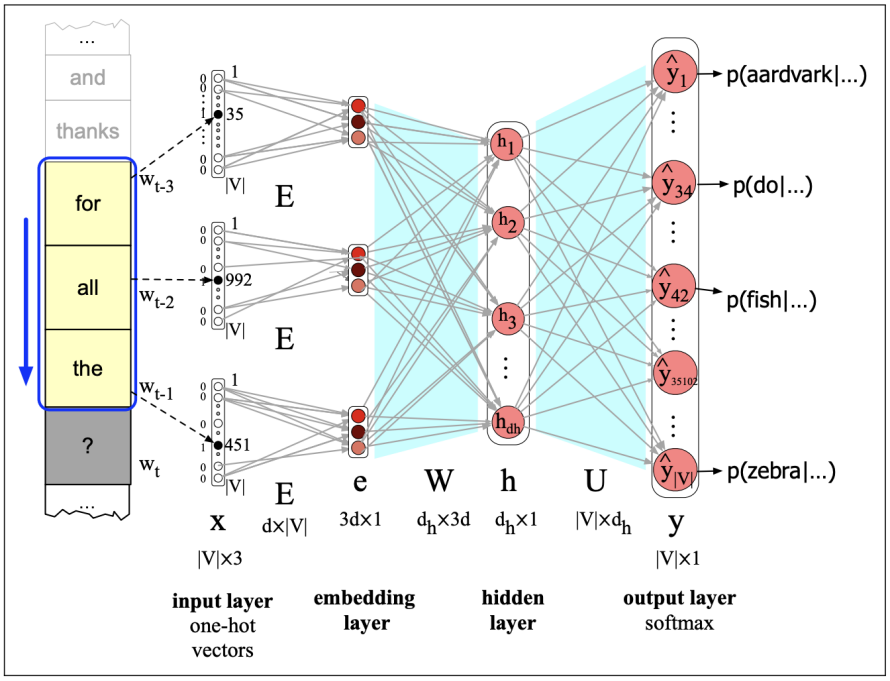


FIGURA 3. FNN de una capa oculta para generación de texto en inglés. Utiliza una matriz de embeddings E pre-calculada, y una ventana de contexto que comprende a los N=3 últimos tokens. La capa de salida es una función softmax y tiene una neurona por token en el vocabulario. Imagen de [29].

vimos el método de N-Gramas. Para resolverlo con una FNN vamos a tratar el problema como uno de clasificación sobre el vocabulario utilizando una capa de salida con activación softmax, cada token posible es una clase diferente.

¿De qué manera recibe la red su entrada? Imaginemos que tenemos una tira de tokens conocidos y queremos que la red prediga el siguiente token. La predicción la haremos en base a los N últimos tokens, también pasando por un preprocesamiento y capa de embedding, de manera que la capa de entrada de la red tiene tamaño total $N \times d$, donde d es la dimensión del embedding. Ilustramos este flujo en la Figura 3 que muestra una FNN de una capa oculta para generación de textos en inglés.
Si planteamos la red de esta manera, no hay pooling, no se pierde información en un proceso de agregación, pero, en contrapartida, la 'memoria' de la red es fija, solo podemos tener en cuenta las últimas N palabras. Además, el embedding es el mismo para cada palabra, sin importar el contexto en el que se encuentre. Las arquitecturas más avanzadas que veremos a continuación se desarrollan en respuesta a estas limitaciones.

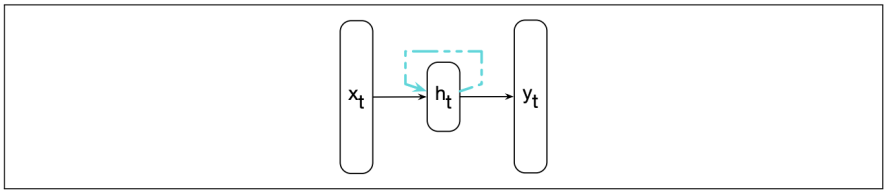


FIGURA 4. Esquema de una RNN simple de una sola capa oculta. En el tiempo t la capa oculta h recibe el valor de entrada x_t pero además su valor anterior el tiempo $t-1$. Imagen de [29].

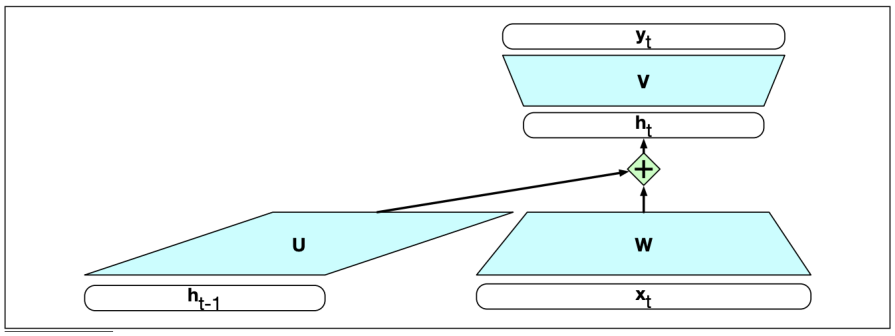


FIGURA 5. Pesos de una RNN simple de una sola capa oculta. La capa oculta h recibe el valor de entrada x_t y el estado oculto anterior h_{t-1}, ambos ponderados por matrices de pesos W y U respectivamente, los resultados se suman y se dan como entrada a la capa h_t. La salida es un vector y que se pondera según los pesos en la matriz V, como en la capa de salida de las FNN ya vistas. Imagen de [29].

5.2. Redes neuronales recurrentes

Una red neuronal recurrente (RNN) es una arquitectura pensada para procesar secuencias de vectores. Son redes donde las salidas de capas intermedias se reutilizan; si pensamos en el grafo que representa la red, estamos diciendo que admitimos ciclos, a diferencia de una FNN donde la información siempre fluye en una sola dirección, sin ciclos. Si notamos a la secuencia como $x_1, \ldots, x_k$, en el momento t la red recibe el vector x_t y lo procesa igual que haría una FNN, pero con el agregado de que las capas ocultas reciben información del estado que tenían en el tiempo $t-1$. Ver la Figura 4 para una representación esquemática del asunto.

El h_t o 'estado oculto' en el tiempo t es el vector representado por la capa oculta de la red, recibe como entrada la capa anterior pero también el estado oculto h_{t-1} del tiempo anterior, cada uno de estos vectores está ponderado por pesos a aprender. De esta manera, tenemos conjuntos de pesos diferentes para ponderar la información del token actual y el resumen de toda la información anterior. Ver la Figura 5 para entender cómo encajan estos componentes en el caso simple de una RNN de una sola capa oculta h.

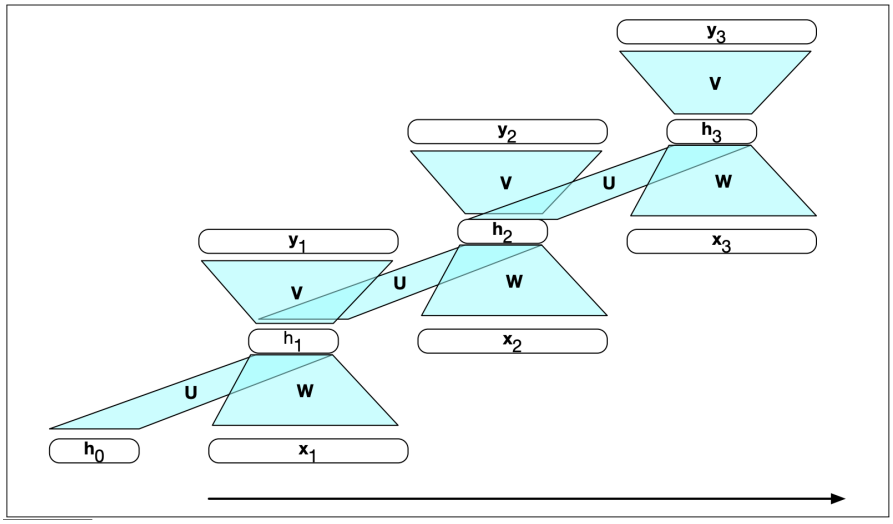


FIGURA 6. Desenrollado de una RNN simple de una sola capa oculta. La red se puede ver como una FNN con la dimensión temporal agregada. La capas de la red se recalculan en cada tiempo t mientras que los pesos U, V, W se comparten a lo largo del tiempo. Imagen de [29].

Formalicemos la arquitectura: sea a la función de activación de la capa h y sea f la función de activación de la capa de salida (típicamente una sigmoide o una softmax), entonces la red de la Figura 5 se puede escribir como:

$$\begin{cases} h_t = a(Wx_t + Uh_{t-1}). \\ y_t = f(Vh_t). \end{cases} \tag{56}$$

No incluimos el sesgo en la ecuación (56) porque consideramos que está integrado en las matrices de pesos como una columna más, concatenando el valor real 1 al vector de entrada y de estados ocultos. Si hubiera más capas tendríamos una recurrencia en la profundidad de la red como variable, igual que en (52). Lo que hacemos en una RNN es sumarle otra dimensión, que es el tiempo. Si 'desenrollamos' la ecuación (56) en el tiempo, notamos que una RNN puede ser vista como una FNN con el agregado de la dimensión temporal, ver Figura 6, esto implica que todo lo visto sobre optimización de la red y el cálculo del gradiente mediante backpropagation podemos aplicarlo aquí también.

Las RNNs superan en desempeño a las FNN en varios problemas de PLN por el hecho de que tienen memoria potencialmente infinita, no hay un parámetro que define cuántas palabras hacia atrás en la secuencia está viendo la red. Sin embargo, en la práctica, pierden desempeño en secuencias muy largas.

Esa pérdida de desempeño se debe primero a que a los estados ocultos h se les pide que sean multipropósito: deben guardar información de toda la secuencia, y por otro

lado proveer de información útil para el token que está siendo procesado, esto hace que la información a largo plazo tienda a diluirse.
En segundo lugar, existe un problema que tiene que ver con la optimización y la arquitectura particular de las RNNs. Si vemos la Figura 6 vemos que la matriz U se multiplica por el estado oculto anterior h_{t-1} en cada tiempo t, por lo tanto, durante el proceso de backpropagation durante el cual se calcula el gradiente de la red, esta matriz aparece multiplicandose repetidamente, tantas veces como tenga de largo la secuencia. Veámoslo en términos de fórmulas; tomemos la ecuación (56) y quedémonos solo con la recurrencia de los h_t, así simplificamos la recurrencia a una de tipo lineal simple y podemos escribir:

$$h_t = Wh_{t-1},$$

que si la secuencia tiene largo t nos da:

$$h_t = W^t h_0.$$

Esto nos da la pauta de que el valor del estado oculto depende de las condiciones iniciales de la matriz W. De hecho, si W es diagonalizable y admite una descomposición $W = PDP^{-1}$ entonces con D diagonal y P matriz de cambio de base, entonces:

$$h_t = PD^t P^{-1} h_0,$$

y los valores propios λ con $|\lambda| > 1$ crecen exponencialmente con t, mientras que si $|\lambda| < 1$, decrecen a 0 rápidamente. Esto surge de la recurrencia durante el proceso de backpropagation y da lugar a los problemas de *exploding gradient* y *vanishing gradient*, respectivamente. El primero hace que el aprendizaje sea muy inestable, mientras que el segundo lo ralentiza o directamente lo detiene.

El problema de falta de memoria se puede mitigar con extensiones de las RNNs que vimos, que tendrán estados (vectores como el h_t) con roles específicos para almacenar información relevante a lo largo del tiempo; una muy popular es la LSTM (Long Short Term Memory). El problema de los gradientes que crecen exponencialmente puede mitigarse con *gradient clipping*, que consiste en truncar los gradientes que superen un cierto umbral. La LSTM y otras arquitecturas, así como más técnicas para mitigar problemas con el gradiente, pueden encontrarse con más detalle en el Capítulo 10 de [14].

5.2.1. Redes recurrentes para PLN.

Una RNN puede tratar cualquier tipo de secuencia, no necesariamente texto, pero si nos centramos en ese tipo de dato, podemos definir la arquitectura según la tarea a resolver.

Si se necesita etiquetar cada elemento de la secuencia, entonces la entrada y salida de la red tendrán la misma dimensión. Un ejemplo clásico es el de asignarle a cada palabra su categoría gramatical, que se conoce como el problema de *POS-Tagging*. En la Figura 7 se muestra una implementación con una RNN.

En cambio, para clasificar la secuencia como un todo, la entrada de la red serán los tokens (sus embedding, para ser precisos) y la salida serán las clases. Casos de uso abundan, uno típico, que ya hemos mencionado, es clasificar los textos como de sentimiento positivo o negativo (ver Figura 8).

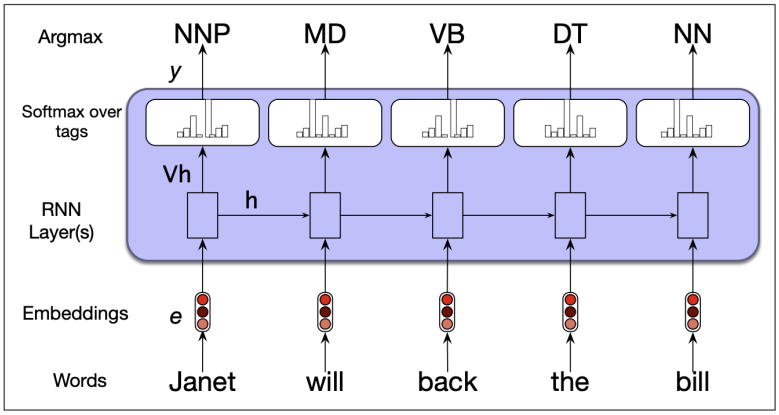


FIGURA 7. Part of Speech (POS) Tagging. Se entrena a la red para que aprenda la categoría gramatical de cada token. Una secuencia de largo n tiene n salidas. El estado oculto h se propaga en el tiempo. Imagen de [29].

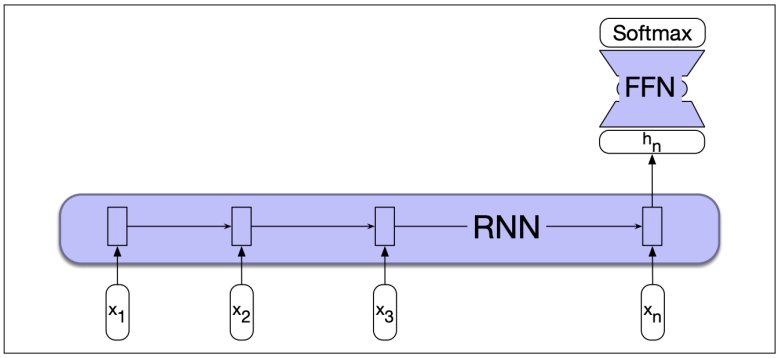


FIGURA 8. Esquema de una RNN para clasificación de un texto. La secuencia se procesa de manera recurrente propagando el estado oculto h en el tiempo, se usa el último estado oculto h_n, que tiene información de toda la secuencia, como entrada para una FNN que realiza la clasificación final. Imagen de [29].

Como última aplicación de RNNs en PLN mencionamos la generación de secuencias, al igual que en las FNN lo que se hace es una clasificación sobre el vocabulario. La diferencia es que no hay ventana de contexto, sino que nos valemos de los estados ocultos para mantener esa información. Ver esquema de implementación en la Figura 9.

5.3. Arquitectura Encoder-Decoder

La arquitectura que vamos a ver se puede usar para problemas donde la entrada es una secuencia y la salida es también una secuencia, como traducción, resumen de textos, modelos de pregunta y respuesta, etc. El problema de POS-Tagging está dentro de esa

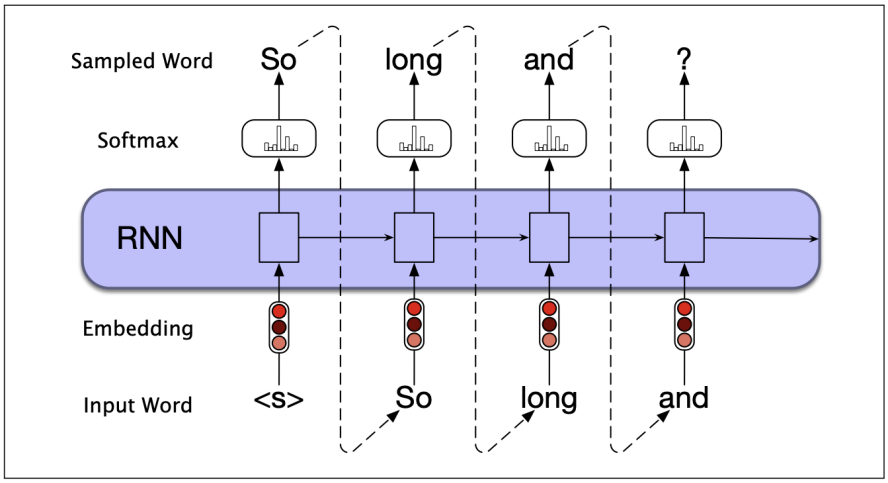


FIGURA 9. RNN para generación de texto. Se empieza la generación ingresando un token reservado para marcar el principio de un texto. En cada paso se genera un token nuevo que es usado como entrada para el paso siguiente. El estado oculto se propaga a lo largo del tiempo. Imagen de [29].

categoría, allí la secuencia de entrada y la de salida tienen el mismo largo. En ese caso, la arquitectura de RNN de la Figura 7 es suficiente, pero hay muchos casos donde los tokens de entrada y salida no tienen una correspondencia uno a uno.

Por ejemplo, en la traducción de un texto de un idioma a otro, las secuencias pueden tener distinto largo y orden; un verbo al principio de la frase en el idioma de entrada puede estar al final en el de salida, por decir algo. Se necesita tener una comprensión global de toda la secuencia de entrada antes de generar la de salida. Esta condición no es exclusiva de la traducción, sino que surge naturalmente en muchas tareas de transformación de secuencias, como en los ejemplos ya nombrados al inicio de la sección.

Para manejar entrada y salida de diferente largo surge la arquitectura Encoder-Decoder. Conceptualmente consta de dos partes: un *Encoder* que toma una secuencia de entrada y la transforma en un vector $\mathbf{c}$ que representa a la secuencia, llamado vector de *contexto* o vector en el espacio latente, y un *Decoder* que toma ese vector y lo utiliza para generar una secuencia de salida. Ver la Figura 10 para una representación esquemática de esta arquitectura.

La implementación del Encoder y del Decoder puede ser realizada por cualquier arquitectura de red neuronal que soporte secuencias, como una RNN, una red convolucional o un Transformer (ver Sección 5.4). Ahora vamos a estudiar la implementación basada en RNNs, donde los estados ocultos jugarán el papel del contexto. Empezamos ilustrando la implementación con la Figura 11, para el caso de traducción de secuencias.

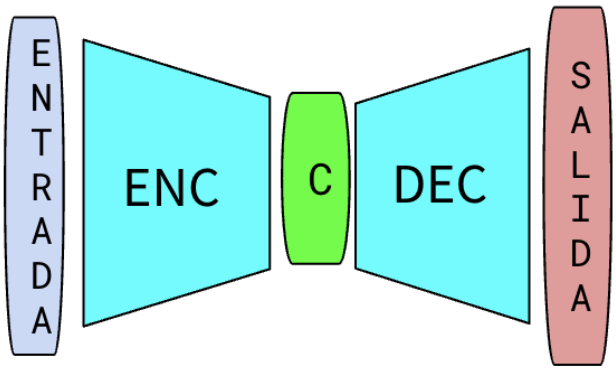


FIGURA 10. Esquema de una arquitectura Encoder-Decoder. El Decoder y el decoder son modelos que soportan secuencias, **C** es un vector llamado contexto.

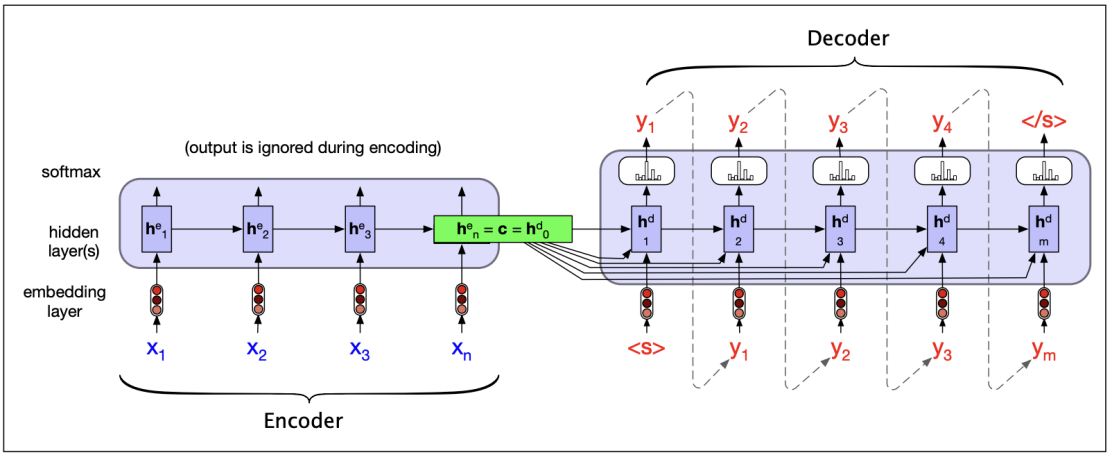


FIGURA 11. Arquitectura Encoder-Decoder basado en RNNs para traducción de secuencias. El Encoder recibe la secuencia de entrada y genera un vector de contexto **c** que es la entrada del Decoder, este es el último estado oculto del Encoder y el primero del Decoder. El Decoder genera la secuencia de salida. Cada estado h_i^d del Decoder tiene como entrada el token anterior, el estado oculto h_{i-1}^d y el contexto. Imagen de [29].

En la Figura 11 se parte de una secuencia $x_1, \ldots, x_n$ que es la entrada del Encoder, la RNN procesa esta entrada de a un token a la vez, generando una salida y un estado oculto en cada paso, que ignoramos (por ahora). Lo único que rescatamos de este proceso es el último estado oculto h_n^e del Encoder, pues mantiene información de toda la secuencia. Este estado oculto es el contexto que se le pasa al Decoder, que también es una RNN, en este caso de generación de texto como la de la Figura 9. En un primer paso el Decoder recibe un token especial `<s>` que indica el inicio de una secuencia de salida y el contexto $h_n^e = c$, que puede pensarse como un estado oculto h_0^d inicial del Decoder.

En cada paso el Decoder genera un token de salida y_i y un estado oculto h_i^d que se usa como entrada en el paso siguiente, además de la salida y_{i-1}, que es el token generado en el paso anterior, junto con el contexto c. La salida del Decoder es una secuencia $y_1, \ldots, y_m$ que es la traducción de la secuencia de entrada.

Podríamos haberle pasado el contexto solo al primer estado oculto del Decoder, pero esto limitaría la influencia del texto de entrada en la generación de los tokens posteriores. Por eso hay una redundancia de información y se le pasa el contexto a cada estado oculto del Decoder. Las fórmulas para el proceso de *decoding* nos quedan:

$$\begin{cases} c = h_n^e = h_0^d, \\ h_i^d = RNN(y_{t-1}, h_{t-1}^d, c), \\ \hat{y}_t = softmax(h_t^d), \end{cases} \tag{57}$$

donde por RNN denotamos a alguna red recurrente que toma como entrada el último token generado, el estado oculto anterior y al contexto, generando un estado oculto nuevo. La salida es un vector $\hat{y}_t$ de probabilidades sobre el vocabulario, representando la probabilidad que cada palabra tiene de ser la siguiente. El token efectivamente generado puede ser el más probable de todos, pero puede haber otras estrategias de muestreo sobre esta distribución, por ejemplo para generar textos más creativos y menos repetitivos (a este tipo de control se le puede encontrar por el nombre de *temperatura*). Durante el proceso de entrenamiento, la salida generada se usa junto con la salida real conocida para calcular la función de costo para optimizar.

5.3.1. Mecanismos de atención.

Desde que empezamos a hablar de RNNs, en más de una oportunidad mencionamos que la influencia de los tokens iniciales de la secuencia se pierde a medida que avanzamos en la misma, hay una pérdida de memoria. La raíz de este problema ya la hemos comentado y es que se le pide al estado oculto, que es un vector de cierta dimensión fija, que mantenga pesos para ponderar toda la información histórica junto con la información de la entrada al momento de predicción. Esto es un problema para las RNN en general y está presente al usarlas como implementación de un Encoder-Decoder, ese último estado oculto del Encoder es un cuello de botella. Por más bueno que sea el Decoder, solo puede trabajar con la información que disponibilice el Encoder.

En realidad tenemos más información a disposición, ya que podríamos no usar solamente el último estado oculto, sino todos los estados intermedios generados por el Encoder. Una opción sería concatenarlos todos, a priori no viable, pues el vector final tendría dimensión variable, aunque se resolvería con algún tipo de padding. Otra opción sería resumirlos de alguna manera igual que comentamos en el caso de las FNNs, mediante una estrategia de pooling. Estrategias válidas pero que no tienen en cuenta una cuestión crucial: no todos los estados ocultos h_i^e son igual de importantes en cada momento de la generación.

Por ejemplo, si en la frase de entrada hay un nombre propio en la posición i, el h_i^e es el estado oculto del Encoder correspondiente a esa posición y quizás deba tener más

peso a la hora de generar el nombre en la traducción. Para capturar este tipo de relaciones entre los estados ocultos, en vez de solo sumar o promediar los h_i^e, vamos a hacer una suma ponderada, donde los pesos de la suma no son fijos sino que dependen del momento de la generación. La forma de calcular estos pesos es lo que se conoce como mecanismo de atención.

Concretamente en el i-ésimo paso de generación lo que haremos es computar un *score* entre h_{i-1}^d, el último estado oculto del Decoder, que se busca que resuma toda la semántica de la frase generada hasta el momento, y cada uno de los estados ocultos h_j^e del Encoder. Una forma de hacer esto es mediante el producto interno entre los vectores:

$$score(h_{i-1}^d, h_j^e) = \left\langle h_{i-1}^d, h_j^e \right\rangle. \tag{58}$$

Este score es una medida de la similitud, si es alto significa que la información que aporta h_j^e es relevante para la generación del token i. Luego se aplica una función softmax sobre estos scores para llevarlos al intervalo $[0, 1]$ y que sumen 1.

$$softmax\left((score(h_{i-1}^d, h_j^e))_j\right) = \left(\frac{e^{\langle h_{i-1}^d, h_1^e \rangle}}{\sum_{k=1}^{n} e^{\langle h_{i-1}^d, h_k^e \rangle}}, \ldots, \frac{e^{\langle h_{i-1}^d, h_n^e \rangle}}{\sum_{k=1}^{n} e^{\langle h_{i-1}^d, h_k^e \rangle}} \right). \tag{59}$$

Obtenemos así los pesos de la suma ponderada:

$$\alpha_{ij} = \frac{e^{\langle h_{i-1}^d, h_j^e \rangle}}{\sum_{k=1}^{n} e^{\langle h_{i-1}^d, h_k^e \rangle}}. \tag{60}$$

Con estos ingredientes, podemos calcular en cada paso i de la generación un vector de contexto c_i definido como la suma ponderada por α_{ij} de los estados ocultos h_j^e del Encoder:

$$c_i = \sum_{j=1}^{n} \alpha_{ij} h_j^e.$$

En la Figura 12 se puede ver una implementación de una arquitectura Encoder-Decoder con RNNs usando mecanismos de atención.

Es posible usar una función de scoring más compleja que el producto interno, de hecho, podemos parametrizar el cálculo, de manera que también se pueda *aprender*, generalizando la ecuación (58) a una forma bilineal definida por una matriz de pesos W_s:

$$score(h_{i-1}^d, h_j^e) = (h_{i-1}^d)^T W_s h_j^e. \tag{61}$$

Esta generalización permite que los estados ocultos sean de distinta dimensión y de aprender pesos particulares en cada instancia de entrenamiento para cada problema

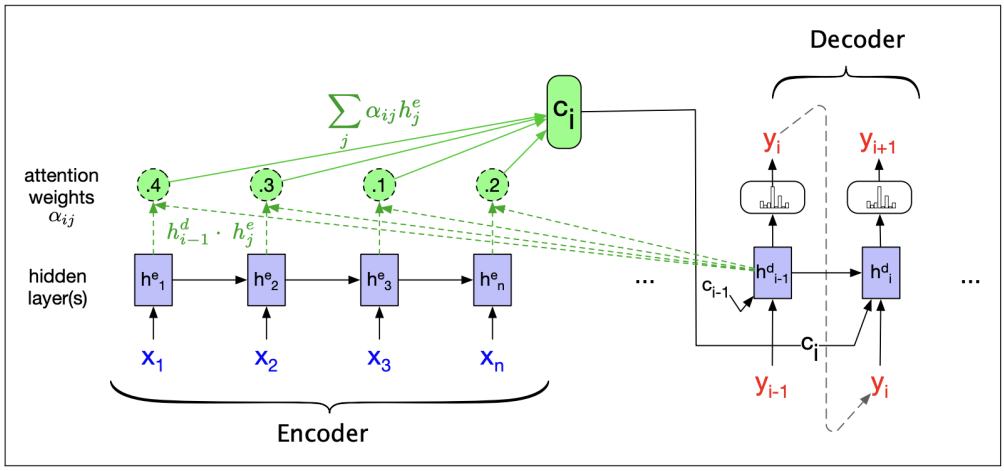


FIGURA 12. Arquitectura Encoder-Decoder con mecanismos de atención. El Encoder genera una secuencia de estados ocultos h^e que son usados para calcular un vector de contexto c_i en cada paso de generación del Decoder. La atención se calcula mediante una función de scoring (por ejemplo producto interno) entre los estados ocultos del Encoder y el último estado oculto del Decoder. Imagen de [29].

particular.

5.4. Transformers

Los mecanismos de atención no tienen por qué ser usados exclusivamente en el contexto de estados ocultos de una RNN. Si tenemos vectores y una forma de compararlos, tenemos un mecanismo de atención.

Nada impide comparar embeddings de palabras con otras, más aún sabiendo que los embeddings capturan la semántica de las palabras y de alguna forma las direcciones en el espacio de embedding capturan conceptos.

Cuando se habla de *attention* en este sentido, se refiere a comparar la influencia o relevancia de un token respecto a otro en otra secuencia (ver Figura 13). En cambio, si se escucha hablar de *self-attention*, el concepto se refiere al caso en el que comparamos tokens de una misma secuencia entre ellos mismos. Esto lo que permite es, dado un token w_i y su contexto (el resto de la frase), entender qué tokens son más relevantes, cuáles modifican la semántica de w_i.
Si $w_1, \ldots, w_i, \ldots w_n$ constituyen una secuencia con embeddings $e_1, \ldots, e_i, \ldots e_n$, calcular los scores de atención entre w_i y el resto de los tokens y realizar la suma ponderada como en (62) nos da un nuevo embedding contextualizado de w_i:

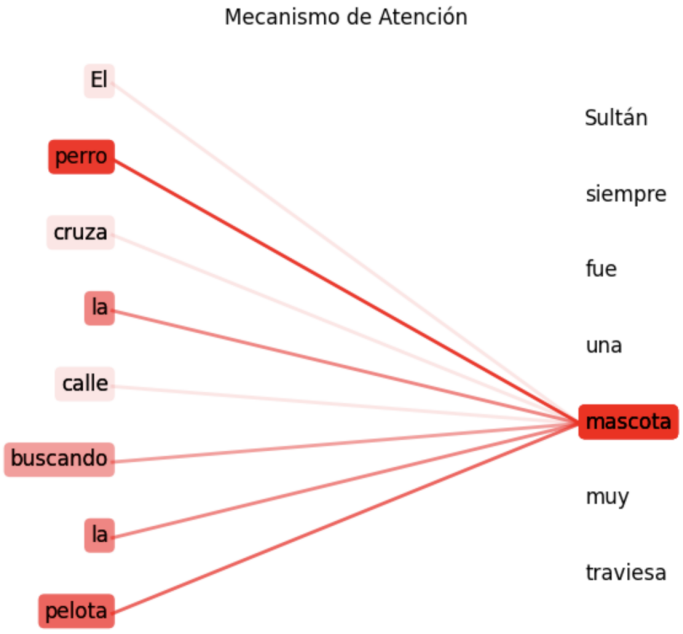


FIGURA 13. Atención entre dos secuencias de tokens. Cada token de la secuencia de entrada es comparado con cada token de la secuencia de salida. El grosor de las líneas indica un valor de atención más fuerte. La palabra 'mascota' de la segunda secuencia guarda una relación semántica fuerte con 'perro' (los vectores tienen un producto interno alto), pero no con 'calle'

$$\hat{e}_i = \sum_{j=1}^{n} \alpha_{ij} e_j,$$

solo que en este caso los pesos α_{ij} se definen como:

$$\alpha_{ij} = \frac{\exp(\langle e_i, e_j \rangle)}{\sum_{k=1}^{n} \exp(\langle e_i, e_k \rangle)}. \tag{62}$$

Este nuevo embedding modifica la semántica capturada por e_i según el contexto en el que se encuentre. Es una manera *dinámica* de modificar los embeddings para adecuarse mejor a la situación particular de cada secuencia. Todos los tokens se benefician de esto, pero un ejemplo que ilustra la utilidad de la técnica es pensar en palabras con más de un significado; un 'banco' puede ser una entidad financiera o un lugar para sentarse, con embeddings estáticos el vector de 'banco' es el mismo no importa si la oración es 'Fuí al banco a retirar mis ahorros' o 'Me senté en el banco a contemplar el paisaje'. Usando *self-attention* obtenemos dos vectores distintos para cada instancia de la palabra.

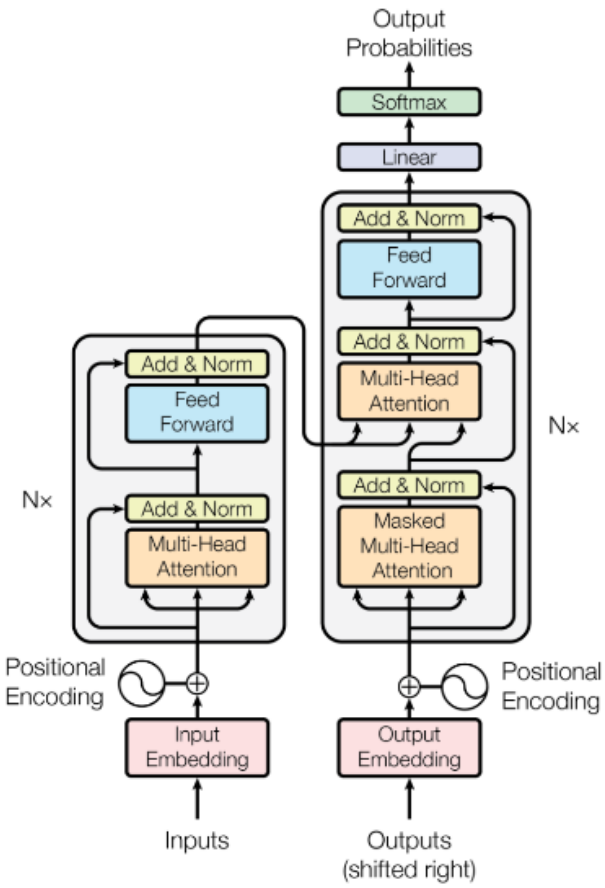


FIGURA 14. Arquitectura del Transformer original de Vaswani et al. La arquitectura se compone de un Encoder y un Decoder, cada uno con bloques de atención y feedforward. El 'bloque' compuesto por el cabezal de atención y la red feedforward se compone consigo mismo N veces. Imagen de [17]

En 2017 Vaswani et al. construyeron sobre estas ideas para desarrollar la arquitectura del Transformer en su artículo 'Attention is All You Need' [17]. La arquitectura original se puede ver en la Figura 14. En principio pensada para el problema de traducción de secuencias, usando una estructura de Encoder-Decoder, se cimentó como la arquitectura base de muchos modelos de PLN, como los de la clase GPT[2] de OpenAI. También se extendieron a otras modalidades, es decir, otros tipos de datos, como imágenes (ver la arquitectura ViT [3])[24], grafos o tablas de datos.

[2]Las siglas GPT significan 'Generative Pre-Trained Transformer'
[3]Vision Transformer

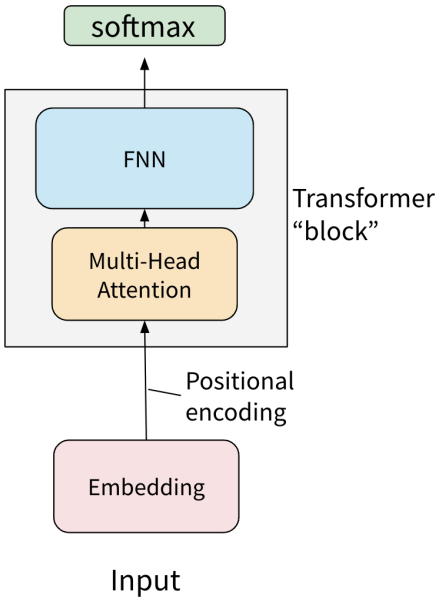


FIGURA 15. Esquema simplificado de un Transformer para generación de texto. El Encoder recibe una secuencia de embeddings de tokens y genera una secuencia de salida. Ignoramos operaciones de suma y normalización, así como *skip-connections*.

Si bien la arquitectura original es un Encoder-Decoder, que tiene que ver con el problema particular de traducción, el Transformer se puede usar en cualquier tarea que requiera procesar secuencias. Comentaremos un esquema más simplificado de la arquitectura que solo se enfoca en lo que sería el Decoder, que puede ser usado, por ejemplo, para generación de texto, ver la Figura 15.

5.4.1. Bloque de atención.

La sección de la red dedicada al mecanismo de atención computa los pesos de atención entre los tokens de la secuencia de entrada, es un caso de self-attention. Lo novedoso en [17] es que un mismo token tiene un embedding distinto según el rol que esta cumpliendo en el cálculo. La idea es que si w_i es un token cuyo embedding queremos contextualizar, entonces ese token se pregunta qué otros tokens lo influencian y debemos usar su embedding de *query* para realizar esta pregunta. En cambio si estamos evaluando si w_i influye en otros tokens, usamos su embedding de *key*. Finalmente, para saber cuál es la magnitud de esa influencia, debemos usar su embedding de *value*.

En la primera definición de atención que vimos con los Encoder-Decoder calculamos la atención del vector u al w como $score(u, w) = \langle u, w \rangle$. En el bloque de atención del Transformer queremos usar esta misma fórmula, pero utilizando los vectores q_u (query) y k_w (key), en la dirección del vector v_w (value), esto es:

$$att_T(u, v) = \langle q_u, k_w \rangle \, v_w. \tag{63}$$

En lugar de realizar cada cálculo por separado, ya que el cálculo consiste en realizar productos internos, es más eficiente calcular la atención de todos los tokens entre sí a la vez utilizando matrices.

Sean $Q, K \in \mathbb{R}^{d_k \times r}$ donde los vectores q_i y k_i son las columnas i-ésimas de cada matriz, respectivamente, y $V \in \mathbb{R}^{d_v \times r}$ con v_i como su columna i-ésima. Las dimensiones d_k y d_v son hiperparámetros que definen el espacio donde queremos definir los embeddings, mientras que r es el tamaño de la ventana de contexto, es la cantidad de tokens que vamos a considerar para generar el siguiente. Calculamos los productos internos multiplicando Q^T por K, obteniendo una nueva matriz:

$$(Q^T K)_{ij} = (\langle q_i, k_j \rangle)_{ij}.$$

Aplicamos una función softmax a cada vector fila de la matriz anterior, esto nos da una nueva matriz:

$$S = softmax(Q^T K).$$

La entrada (i, j) de la matriz S es el número

$$s_{ij} = \frac{e^{\langle q_i, k_j \rangle}}{\sum_{t=1}^{r} e^{\langle q_i, k_t \rangle}}.$$

La suma de los v_j ponderada por los s_{ij} (la fila i-ésima de S) da lugar a un vector $\hat{q}_i$, que es el embedding contextualizado de q_i:

$$\hat{q}_i = \sum_{j=1}^{r} s_{ij} v_j = \begin{pmatrix} | & & | \\ v_1 & \cdots & v_r \\ | & & | \end{pmatrix} \begin{pmatrix} s_{i1} \\ \vdots \\ s_{ir} \end{pmatrix} = V \begin{pmatrix} s_{i1} \\ \vdots \\ s_{ir} \end{pmatrix}, \tag{64}$$

donde hicimos explícito el cálculo de $\hat{q}_i$ en función de las columnas de V, usando el hecho de que multiplicar una matriz por un vector resulta en una combinación lineal de las columnas de la matriz. Esto nos indica que podemos calcular todas las representaciones $\hat{q}_i$ para $i = 1, ..., r$ multiplicando V por S^T:

$$\hat{Q} := VS^T = \begin{pmatrix} | & & | \\ v_1 & \cdots & v_r \\ | & & | \end{pmatrix} \begin{pmatrix} s_{11} & \cdots & s_{i1} & \cdots & s_{r1} \\ s_{12} & \cdots & s_{i2} & \cdots & s_{r2} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ s_{1r} & \cdots & s_{ir} & \cdots & s_{rr} \end{pmatrix} = \begin{pmatrix} | & & | \\ \hat{q}_1 & \cdots & \hat{q}_r \\ | & & | \end{pmatrix}. \tag{65}$$

Los embeddings contextualizados se encuentran en las columnas de $\hat{Q} = VS^T$. La formulación matricial de la capa de atención nos queda entonces:

$$att_T(Q, K, V) = \hat{Q} = VS^T = V softmax(Q^T K), \tag{66}$$

donde recordamos que Q, K, V son matrices de pesos a aprender y la función softmax se aplica por separado para cada fila.

Esta es casi la forma presentada en el paper [17], a menos de una normalización por $\sqrt{d_k}$. La razón que allí se da para esta normalización es que el producto interno crecerá en valor absoluto con la dimensión de los embeddings, llevando la softmax a valores muy cercanos a 0 o 1, donde los gradientes son más pequeños, dificultando la optimización. La fórmula final queda entonces:

$$att_T(Q, K, V) = \hat{Q} = VS^T = Vsoftmax\left(\frac{Q^T K}{\sqrt{d_k}}\right). \tag{67}$$

Observación 5.4.1. En el artículo original, la notación elegida usa matrices fila. La fórmula presentada para el mecanismo de atención allí es

$$att_{vaswani}(A, B, C) = softmax\left(\frac{AB^T}{\sqrt{d_k}}\right)C,$$

que es equivalente a nuestra notación para el mecanismo de atención, basta con trasponer. La equivalencia es $A = Q^T, B = K^T, C = V^T$ y se cumple $att_{vaswani}(Q^T, K^T, V^T) = att_T(Q, K, V)^T$.

La multiplicación de matrices es muy eficiente en una computadora, donde existe hardware dedicado para esto. Además, la fórmula es paralelizable, cada token puede ser comparado con los demás por separado, utilizando el hardware de la manera más eficiente. Esto último supone una ventaja crucial en la práctica con respecto a las RNNs, donde los cálculos se realizan secuencialmente.

Una limitante del bloque de atención, y por extensión del Transformer, es que las secuencias que se pueden procesar tienen un tamaño fijo, no podemos potencialmente almacenar información de secuencias de tamaño arbitrario como en las RNNs (aunque en la práctica esto tampoco sucedía). Sin embargo, el problema en la práctica se mitiga si se usan ventanas de atención grandes, como se usan actualmente, con grandes modelos de lenguaje (LLMs) con ventanas de más de 128.000 tokens.

5.4.2. Multi-Head Attention.

Lo que describimos anteriormente se puede entender como **un** bloque de atención. En la arquitectura del Transformer se utilizan varios bloques a la vez, en lugar de ejecutar una sola instancia de atención siguiendo la ecuación (67), se ejecutan varias, con diferentes valores de queries, keys y values. De esta manera, la red aprende distintas maneras de contextualizar el token. Esto se hace en paralelo para cada token. Luego se agregan los resultados de todos los bloques, que se puede realizar concatenando o sumando las salidas de cada uno.

5.4.3. Bloque Feedforward.

La salida de la capa de atención se pasa por una FNN usual. Puede entenderse que la capa de atención se encarga de contextualizar los embeddings de los tokens y la FNN de realizar la tarea específica de PLN, como clasificación o generación de texto. Esta

división es útil desde un punto de vista didáctico, pero la forma en la que los pesos entre capas interactúan entre sí depende de cada instancia particular de optimización de los pesos. En los comentarios finales hablamos un poco sobre este tema de la interpretabilidad de los modelos. Aún más compleja es la interpretación de los componentes si se tienen conexiones que saltean capas, llamadas *skip-connections*, que están presentes en el diagrama de la Figura 14.

5.4.4. Positional Encoding.

Los tokens dentro de una secuencia tienen un orden. Hay muchos contextos en los que esto es importante; intuitivamente, es más probable empezar una oración con un artículo o un sujeto que con un verbo, por ejemplo. Las RNNs utilizan la posición de forma implícita, ya que procesan la secuencia de a un token a la vez. En cambio, en los Transformers, los mecanismos descritos hasta ahora no se nutren de la información de la posición del token.

Para agregar esta información importante es que se utiliza lo que se conoce como *positional encoding*. La idea es agregar a los embeddings de los tokens información de la posición, esto se hace sumándoles una cantidad que depende justamente de esa posición. Hay distintas formas de implementarlo, se pueden tener reglas fijas o usar un encoding aprendido. En el artículo de Vaswani et al. se utilizan senos y cosenos de distintas frecuencias para codificar la posición de los tokens. La fórmula es la siguiente, aplicada en cada entrada par $2i$ e impar $2i+1$ del embedding:

$$\begin{cases} PE(pos, 2i) = \sin\left(\frac{pos}{10000^{2i/d}}\right), \\ PE(pos, 2i+1) = \cos\left(\frac{pos}{10000^{2i/d}}\right), \end{cases} \tag{68}$$

donde pos es la posición del token en la secuencia y d la dimensión del embedding. A cada posición se le suma un vector particular de sinusoides, fijo. El modelo aprende a reconocer esta información como una marca de la posición del token. Dicho de otra forma, la información de la posición está codificada en ese vector de sinusoides y el modelo aprende a reconocerla.

5.4.5. ¿Por qué los Transformers?

Desde su aparición en 2017, los Transformers han pasado a dominar el estado del arte en PLN y otras áreas del aprendizaje automático, ejemplos de modelos basados en Transformers que son o fueron estado del arte pueden ser [19][20][25]. Vamos a mencionar algunas razones por las que esto ha sucedido así.

5.4.5.1. *Mantiene dependencias a largo plazo.*

Los Transformer parecen resolver el problema de mantener relación entre tokens distantes entre sí en una secuencia larga, es decir, que no sufren de pérdida de memoria como las RNNs. El último token de la secuencia tiene la misma información sobre el token inmediatamente anterior que la que tiene sobre el inicial, eso sí, siempre y cuando la ventana de contexto abarque a toda la secuencia. ¿Cuál es la diferencia con arquitecturas de tipo FNN para secuencias como la de la Figura 3? Una diferencia es el uso

de mecanismos de atención para generar embeddings contextualizados en vez de embeddings fijos.

5.4.5.2. *Scaling laws.*
Por otro lado, son muy eficientes en cuanto a tiempo y capacidad de cómputo, son altamente paralelizables porque el procesamiento de cada token es independiente del resto. Esta ventaja en eficiencia no es menor y puede que sea la razón principal del éxito de la arquitectura, ya que vuelve viable entrenar con más datos y más parámetros. Esto ha probado ser crucial para obtener modelos en el estado del arte, que realmente pueden realizar tareas que modelos con arquitecturas similares pero de escala menor no pueden [23]. La escala del modelo refiere tanto al tamaño del conjunto de entrenamiento, como a la cantidad de parámetros y al tiempo de cómputo usado para el entrenamiento.

Como ejemplo de las llamadas *scaling laws* podemos tomar modelos de tipo GPT que OpenAI desarrolló desde 2017: primero GPT ([18] 2018, 1.17×10^8 parámetros), GPT-2 ([20], 2019 con 1.5×10^9 parámetros) y GPT-3 ([22], 2020 con 1.75×10^{11} parámetros). Al aumentar los parámetros también se recomienda aumentar la cantidad de datos y el tiempo de entrenamiento, de lo contrario se puede caer en sobreajuste [23]. Una versión de GPT-3 fue la usada en el lanzamiento de ChatGPT a fines de 2022[4].

Estas reglas de escalado parecen cumplirse en general pero vale la pena mencionar que no es *siempre* el caso. El uso eficiente tanto de los datos de entrenamiento como de los recursos de cómputo es un área de investigación en sí misma, el lector interesado puede leer el artículo [28] de Sara Hooker, donde se dan ejemplos donde la regla de escalado se rompe y razones por las que esto podría estar pasando.

5.4.5.3. *Aplicabilidad.*
Hay que destacar también que los Transformers no son solo útiles para procesar texto, son máquinas que procesan secuencias y, por lo tanto, cualquier dato que sea *tokenizable*, es decir, que pueda ser descompuesto en elementos atómicos y representado como secuencia de estos tokens, puede ser procesado por un Transformer. Una imagen se puede dividir en tokens, audio también. Incluso existen técnicas para hacer coexistir las representaciones vectoriales de tokens que provienen de distintos dominios, como es el caso de CLIP [25], dando lugar a lo que se conoce como modelos *multimodales*.

Más aún, no solo hay que pensar en procesar contenido como texto, audio, imágenes o video. Se han utilizado con éxito arquitecturas del tipo Transformer para la detección de fraude en pagos electrónicos (Nubank) o sistemas de recomendación (Netflix [31]). Uno de los desafíos a resolver para adoptar esta arquitectura es cómo obtener representaciones útiles del dominio de interés (los embeddings) y, crucialmente, cuál es la mejor manera de darle un orden a esas representaciones.

[4]Cabe mencionar que para un producto como ese no basta solo con las scaling laws, al preentrenamiento hay que sumarle la tarea de hacer que el texto generado se asemeje al tipo de texto esperado en una conversación de mensajería instantánea, con técnicas como RLHF (*reinforcement learning with human feedback*).

CAPÍTULO 6

Comentarios finales

En este trabajo hemos tratado los fundamentos matemáticos claves para entender un modelo de aprendizaje automático; cómo definimos un modelo, qué alcance tiene cada método y que debemos entender el 'entrenamiento' como la resolución de un problema de optimización. Usando el PLN como eje conductor, hemos visto que un mismo problema se puede atacar con métodos de distinta complejidad, cada uno con sus ventajas y desventajas.
Desde métodos basados en reglas como las expresiones regulares, que son útiles en problemas acotados y bien definidos pero que fallan en problemas más abiertos; hasta redes neuronales que tienen una capacidad expresiva impresionante, afirmación sustentada hasta cierto punto por el teorema de aproximación universal 4.2.4, pero su entrenamiento puede presentar desafíos y requiere de muchos datos.

Como mencionamos en la introducción, las aplicaciones en el aprendizaje automático a veces avanzan más rápido que la teoría. En este sentido, el pragmatismo es ley, lo que funciona se adopta. No sería coherente con la realidad pretender que solo se adopten modelos totalmente entendidos y probados con garantías matemáticas. Se puede argumentar que la labor de la investigación es generar conocimiento que acompañe y guíe esa adopción, además de dar explicaciones cuando las cosas parecen no funcionar. Esta monografía me ha dado algunas herramientas para tomar ese camino, y también intenta ser un punto de partida para otros que quieran hacer lo mismo.

Utilizamos el PLN como eje conductor del trabajo para fijar ideas, pero mucho de lo visto, tanto a nivel teórico como a nivel de implementación, puede adaptarse a otros tipos de datos: problemas con datos en $\mathbb{R}^n$ de forma 'nativa' (tabulares), imágenes, audio, datos en formato de grafos. Si hablamos concretamente del Transformer, la arquitectura puede soportar cualquier conjunto de datos en forma de secuencia. De hecho, el punto clave de la traducción para pasar de un dominio de datos a otro es el de la vectorización: basta con tener un vector en $\mathbb{R}^n$ para que la red pueda interpretarlo, no importa su origen. Hay métodos de vectorización como el mencionado CLIP que se pueden usar para que vectores representando tanto textos como imágenes convivan en un mismo espacio, gozando de las mismas características deseables de semántica de embeddings que describimos en la sección 5.1.1 del Capítulo 5.

6.1. Trabajo a futuro

Hay muchas áreas en el aprendizaje profundo en las que se puede profundizar, a continuación se mencionan algunas.

6.1.1. Garantías en el proceso de optimización en aprendizaje automático.

En el Capítulo 3 probamos el Teorema 3.3.1, que nos da el paso óptimo para que el algoritmo de SGD tenga el menor error esperado acotado en cierto número de iteraciones y vimos que la entropía cruzada como función de costo para clasificación binaria con la RL se encuentra en las hipótesis del mencionado teorema (ver el ejemplo 3.3.3 y la prueba de convexidad en la sección 4.1.2).

Sin embargo, vimos con el ejemplo 4.2.1 que las redes neuronales no suelen ser convexas, por lo que no aplica el Teorema 3.3.1 ni el 3.3.2 en ellas. Esta carencia motiva a estudiar qué garantías hay para el uso del SGD en el contexto del aprendizaje profundo, ¿podemos dar resultados similares a los vistos, que acoten el error estimado al querer hallar el mínimo mediante SGD en función de los hiperparámetros del algoritmo, por ejemplo, del paso α?

El problema no es sencillo y posiblemente no tenga una respuesta única que sirva para las funciones de costo de cualquier red neuronal. Una línea de trabajo posible es profundizar en el entendimiento de los resultados de Robbins-Monro [1] .

6.1.2. Modelos fundacionales.

Los últimos años han visto un cambio de paradigma en el abordaje de los problemas de PLN favoreciendo el uso de modelos generales (como los LLMs) que pueden realizar tareas diversas con mínimo o nulo entrenamiento particular en ellas, aplicando lo que se conoce como *few-shot learning* y *zero-shot learning*, respectivamente. Este contexto lleva a preguntarse si ese enfoque generalista puede aplicarse en dominios diferentes. Mencionamos los ejemplos de sistemas de detección de fraude o de recomendación, pero puede haber muchos. El atractivo de los así llamados 'modelos fundacionales' es que no implican un desarrollo de un modelo específico para cada tarea particular, son versátiles, adaptandose a necesidades variables y muchas veces es más simple mantener funcionando correctamente en una aplicación real un solo modelo que mantener una cantidad variable de modelos expertos.

Partiendo de lo aprendido en PLN, parecería prometedor usar arquitecturas basadas en el Transformer para encontrar modelos fundacionales en más áreas. Para esto parece ser clave poder plantear el problema a resolver de una manera *auto-supervisada*, es decir, como una clasificación donde el valor a predecir (la etiqueta), viene dada por el problema mismo. Por ejemplo, en la predicción de la siguiente palabra en un texto, ya sabemos la *verdadera* palabra que le sigue a cada palabra de entrenamiento, puesto que son parte del conjunto de textos conocidos. Así el modelo puede nutrirse de mucha cantidad de

datos, solo posible si no hace falta etiquetado manual.

6.1.3. Interpretabilidad.

Otra área interesante es la de interpretabilidad; dado un vector x de entrada a una red, no es evidente cómo las distintas *features* de x influencian la salida final del modelo, la red actúa como caja negra. Hay trabajos en esta área que han aportado mucho, como [16], donde se toma prestado de la teoría de juegos el concepto de *shapley values* para cuantificar la influencia de cada feature, una bibliografía excelente para este tipo de interpretabilidad es el libro 'Interpretable Machine Learning' de Christoph Molnar [27].

Un paso más allá está el poder entender los comportamientos internos de la red ¿qué parámetros responden más a qué features? ¿se puede entender entonces qué neuronas o conjunto de neuronas 'aprenden' más de un concepto que de otro? Cuestiones de este tipo se tratan en lo que en inglés se conoce como *mechanistic interpretability*, el interesado puede referirse a trabajos desarrollados en la empresa Anthropic [30]. Esta investigación es relevante en los LLMs para controlar la salida de estos modelos, más allá de lo que se puede hacer con técnicas de *finetuning* (post-entrenamiento de un modelo para una tarea específica) o de *prompting*. Si entendemos las representaciones internas de la red, la esperanza es que también podrían ser modificadas a gusto para dirigir con más precisión el modelo, estamos hablando de poder 'programarlo' en un nuevo sentido de la palabra.

APÉNDICE A

Pruebas

Aquí se detallan teoremas y pruebas que no se incluyeron en el cuerpo principal del trabajo.

TEOREMA A.0.1. *Sea $\{x_k\}$ una sucesión generada por un método de gradiente $x_{k+1} = x_k + \alpha_k d_k$ con $\alpha_k \to 0$ y $\sum_{k=0}^{\infty} \alpha_k = \infty$. Donde el gradiente de f es L-Lipschitz y además existen $c_1, c_2 > 0$ tales que*

$$c_1 \|\nabla f(x_k)\|^2 \leq -\|\nabla f(x_k)\|^t d_k, \quad (69) \qquad \|d_k\|^2 \leq c_2 \|\nabla f(x_k)\|^2 . \quad (70)$$

Entonces se tiene que $f(x_k) \to -\infty$ o $f(x_k)$ converge a un punto finito y $\nabla f(x_k) \to 0$. Además, todo punto límite de $\{x_k\}$ es un punto crítico de f.

PRUEBA. Al igual que en la prueba de 3.2.3 usamos el hecho de que ∇f es L-Lipschitz junto con el lema 3.2.2 para obtener:

$$f(x_k + \alpha_k d_k) - f(x_k) \leq \alpha_k (\frac{1}{2}\alpha_k L \|d_k\|^2 - |\nabla f(x_k)^T d_k|). \quad (71)$$

Ahora usando (70) sobre $\frac{1}{2}\alpha_k L \|d_k\|^2 - |\nabla f(x_k)^T d_k|$ obtenemos:

$$\frac{1}{2}\alpha_k L \|d_k\|^2 - |\nabla f(x_k)^T d_k| \ \leq \frac{1}{2}\alpha_k L c_2 \|\nabla f(x_k)\|^2 - |\nabla f(x_k)^T d_k|.$$

Además $\nabla f(x_k)^T d_k < 0$ y por lo tanto $-|\nabla f(x_k)^T d_k| = \nabla f(x_k)^T d_k$, usando (69) obtenemos que:

$$\nabla f(x_k)^T d_k \leq -c_1 \|\nabla f(x_k)\|^2$$
$$(\Rightarrow) - |\nabla f(x_k)^T d_k| \leq -c_1 \|\nabla f(x_k)\|^2 .$$

Entonces

$$\begin{aligned}\frac{1}{2}\alpha_k L \|d\|^T - |\nabla (x_k)^T d_k| \leq & \frac{1}{2}\alpha_k L c_2 \|\nabla f(x_k)\|^2 - c_1 \|\nabla f(x_k)\|^2 \\ = & f(x_k) + (\frac{1}{2}\alpha_k c_2 L - c_1) \|\nabla f(x_k)\|^2 \\ = & f(x_k) - \alpha_k c_1 \|\nabla f(x_k)\|^2 + \alpha_k^2 \frac{1}{2} c_2 L \|\nabla f(x_k)\|^2 .\end{aligned}$$

Ahora, el termino cuadrático en α_k en la ecuación anterior tiende a 0 más rápido que el lineal, entonces eventualmente podemos despreciarlo. Formalmente decimos que existe una constante $c > 0$ y un índice k_0 tal que si $k \geq k_0$ entonces:

$$f(x_{k+1}) \leq f(x_k) - \alpha_k c \|\nabla f(x_k)\|^2 .$$

Esto prueba que f es (eventualmente) decreciente, así que $f(x_k) \to -\infty$ o $f(x_k) \to \overline{p}$. Nos centramos en el resto de la prueba en el segundo caso. La ecuación anterior nos dice que $\alpha_k c_1 \|\nabla f(x_k)\|^2 \leq f(x_k) - f(x_{k+1})$ para todo $k \geq k_0$. Sumando en desde k_0 hasta $k_0 + m$ obtenemos:

$$\sum_{k=k_0}^{k_0+m} \alpha_k c_1 \|\nabla f(x_k)\|^2 \leq \sum_{k_0}^{k_0+m} f(x_k) - f(x_{k+1}) = f(x_{k_0}) - f(x_{k_0+m+1}),$$

donde en la última igualdad usamos que la suma es telescópica. Si ahora hacemos tender m a infinito nos queda:

$$c_1 \sum_{k=k_0}^{\infty} \alpha_k \|\nabla f(x_k)\|^2 \leq f(x_{k_0}) - \lim_{k\to\infty} f(x_k) < \infty.$$

Pudimos acotar el miembro de la derecha porque $f(x_k)$ converge a $\overline{p}$. Probamos así que la serie de la izquierda converge, es decir que $\sum_{k=k_0}^{\infty} \alpha_k \|\nabla f(x_k)\|^2 < \infty$. El resto de la prueba consiste en probar que $\lim \|\nabla f(x_k)\| = 0$, la estrategia será probar que tanto el límite inferior como el superior son 0.

Empezamos por probar que $\|\nabla f(x_k)\|^2 \to 0$, supongamos que la norma nunca alcanzase el 0, o sea que existe $\epsilon > 0$ tal que $\|\nabla f(x_k)\|^2 \geq \epsilon$ para todo $k \geq k_0$ pero entonces multiplicando por α_k obtenemos:

$$\alpha_k \|\nabla f(x_k)\|^2 \geq \alpha_k \epsilon, \text{ para todo } k \geq k_0,$$

y sumado en $k \geq k_0$ tendríamos:

$$\sum_{k=k_0}^{\infty} \alpha_k \|\nabla f(x_k)\|^2 \geq \epsilon \sum_{k=k_0}^{\infty} \alpha_k = \infty$$

lo cual contradice la convergencia de la serie, que acabamos de establecer. Esto prueba que $\forall \epsilon > 0, \exists k \geq k_0 t.q. \|\nabla f(x_k)\|^2 < \epsilon$ y esto prueba, por definición, que $\liminf \|\nabla f(x_k)\|^2 = 0$, que implica $\liminf \|\nabla f(x_k)\| = 0$.

Nos resta probar que $\limsup \|\nabla f(x_k)\| = 0$, supongamos por absurdo que no, es decir que existe $\epsilon > 0$ tal que $\limsup \|\nabla f(x_k)\| \geq \epsilon > 0$. Entonces por un lado tenemos que, dado un índice, $\|\nabla f(x_k)\|$ está tan cerca de 0 como queramos, para algún k mayor a ese índice, esto por la definición de límite inferior. Por otro lado se cumple lo análogo para el límite superior, solo que en vez de estar dentro de un entorno de 0, podemos asegurar que la norma está a una distancia mayor a ϵ de 0. Usando estos hechos, podemos construir dos subsucesiones $\{n_j\}$ y $\{m_j\}$ tales que:

1) $m_j < n_j < m_{j+1}$.
2) $\frac{\epsilon}{3} < \|\nabla f(x_k)\|, \forall m_j \leq k < n_j$.
3) $\|\nabla f(x_k)\| \leq \frac{\epsilon}{3}, \forall n_j \leq k < m_{j+1}$.

Ahora, la cola de una serie convergente converge a 0, así que $\sum_{k=k_0+n}^{\infty} \alpha_k \|\nabla f(x_k)\|^2 \to 0$ si $n \to \infty$. Entonces tomando un $\bar{j}$ lo suficientemente grande, esto es, tal que $m_{\bar{j}} >> k_0$, podemos hacer que la serie sea tan chica como queramos. Entonces para tal $\bar{j}$ tenemos:

$$\sum_{k=m_{\bar{j}}}^{\infty} \alpha_k \|\nabla f(x_k)\|^2 < \frac{\epsilon^2}{9L\sqrt{c_2}}. \tag{72}$$

Consideremos entonces $j \geq \bar{j}$ y sea m tal que $m_j \leq m < n_j$, se cumple la siguiente cadena de igualdades y desigualdades:

$$\begin{aligned}
\|\nabla f(x_{n_j}) - \nabla f(x_m)\| &\overset{\text{telescópica}}{=} \left\| \sum_{k=m}^{n_j-1} \nabla f(x_{k+1}) - \nabla f(x_k) \right\| \\
&\overset{\text{triangular}}{\leq} \sum_{k=m}^{n_j-1} \|\nabla f(x_{k+1}) - \nabla f(x_k)\| \\
&\overset{\nabla f \text{ Lipschitz}}{\leq} L \sum_{k=m}^{n_j-1} \|x_{k+1} - x_k\| \\
&\overset{\text{def } x_k}{=} L \sum_{k=m}^{n_j-1} \alpha_k \|d_k\| \\
&\overset{(1)}{\leq} L\sqrt{c_2} \sum_{k=m}^{n_j-1} \alpha_k \|\nabla f(x_k)\| \\
&\overset{(2)}{\leq} \frac{3L\sqrt{c_2}}{\epsilon} \sum_{k=m}^{n_j-1} \alpha_k \|\nabla f(x_k)\|^2 \\
&\leq \frac{3L\sqrt{c_2}}{\epsilon} \frac{\epsilon^2}{9L\sqrt{c_2}} \\
&= \frac{\epsilon}{3}.
\end{aligned}$$

Donde en (1) usamos (70) y para (2) usamos que como $m_j \leq m < n_j$ vale que $\|\nabla f(x_k)\| > \frac{\epsilon}{3}$ y por lo tanto $\frac{3}{\epsilon} \|\nabla f(x_k)\| > 1$ para todo k en el rango de la sumatoria, entonces lo que hicimos en el punto (2) aumentar cada sumando multiplicando por esa cantidad, que es mayor a 1 así que mantenemos la desigualdad de menor o igual. El último paso es solo usar la cota que elegimos con ese propósito para la serie. Lo probado junto con (69) nos permite decir que:

$$\|\nabla f(x_m)\| \leq \left\|\nabla f(x_{n_j}) - \nabla f(x_m)\right\| + \left\|\nabla f(x_{n_j})\right\| \leq \frac{2\epsilon}{3}, \forall j \geq \bar{j}, m_j \leq m < n_j,$$

donde para la primera desigualdad usamos la propiedad triangular y para la segunda acotamos por un lado $\left\|\nabla f(x_{n_j}) - \nabla f(x_m)\right\|$ según lo acabamos de probar y para acotar $\left\|\nabla f(x_{n_j})\right\|$ usamos (69), ya que vale para n_j. Utilizando de vuelta (69) pero ahora para

los $n_j \leq m < m_{j+1}$ se tiene que $\|\nabla f(x_m)\| \leq \frac{\epsilon}{3} < \frac{2\epsilon}{3}$. Combinando todo hemos probado que existe $\overline{j}$ tal que para todo $m \geq m_{\overline{j}}$ se cumple:

$$\|\nabla f(x_m)\| \leq \frac{2\epsilon}{3}.$$

O sea, encontramos un índice, tal que para todo m mayor a este, $\|\nabla f(x_m)\|$ está tan cerca de 0 como queramos, pero esto contradice que $\limsup \|\nabla f(x_k)\| \geq \epsilon$. Por lo tanto $\limsup \|\nabla f(x_k)\| = 0$ y junto con lo propio probado antes para $\liminf$, deducimos que $\lim \|\nabla f(x_k)\| = 0$.

Finalmente entonces, si $\overline{p}$ es un punto de aglomeración de x_k, entonces por continuidad de f y de f tenemos que $\nabla f(\overline{p}) = \lim_k \nabla f(x_k) = 0$, por lo tanto $\overline{p}$ es un punto crítico de f. ■

Bibliografía

[1] Herbert Robbins and Sutton Monro. "A stochastic approximation method". In: *The annals of mathematical statistics* (1951), pp. 400–407.

[2] Zellig S Harris. "Distributional structure." In: *WORD* (1954).

[3] John Rupert Firth. "A synopsis of linguistic theory 1930-1955". In: *Studies in Linguistic Analysis, Special Volume/Blackwell* (1957).

[4] Joseph Weizenbaum. "ELIZA—A Computer Program For the Study of Natural Language Communication Between Man and Machine". In: *Communications of the ACM* 9.1 (1966), pp. 36–45.

[5] Walter Rudin. *Functional analysis*. Vol. 12. McGraw-Hill, 1976.

[6] George Cybenko. "Approximation by superpositions of a sigmoidal function". In: *Mathematics of Control, Signals, and Systems* 2.4 (1989), pp. 303–314.

[7] Kurt Hornik, Maxwell B. Stinchcombe, and Halbert L. White. "Multilayer feedforward networks are universal approximators". In: *Neural Networks* 2 (1989), pp. 359–366.

[8] Dimitri P. Bertsekas. *Nonlinear Programming*. 2nd. Athena Scientific, 1999. ISBN: 978-1886529007.

[9] Elias M. Stein and Rami Shakarchi. *Real Analysis: Measure Theory, Integration, and Hilbert Spaces*. Princeton University Press, 2005. ISBN: 9780691113869.

[10] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. 3rd ed. Boston, MA: Addison-Wesley, 2006.

[11] John Duchi, Elad Hazan, and Yoram Singer. "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization". In: *Journal of Machine Learning Research (JMLR)* (2011).

[12] Tomas Mikolov et al. *Efficient Estimation of Word Representations in Vector Space*. 2013. arXiv: `1301.3781 [cs.CL]`.

[13] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *International Conference on Learning Representations (ICLR)* (2015).

[14] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[15] Zhou Lu et al. "The Expressive Power of Neural Networks: A View from the Width". In: *NeurIPS* (2017).

[16] Scott Lundberg and Su-In Lee. *A Unified Approach to Interpreting Model Predictions*. 2017.

[17] Ashish Vaswani et al. *Attention Is All You Need*. 2017.

[18] Alec Radford et al. *Improving Language Understanding by Generative Pre-Training*. 2018.
[19] Jacob Devlin et al. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *Nations of the Americas Chapter of the Association for Computational Linguistics (NAACL)* (2019).
[20] Alec Radford et al. "Language models are Unsupervised Multitask earners". In: *OpenAI blog* (2019).
[21] Benjamin Recht and Stephen J. Wright. *Optimization for Modern Data Analysis*. 2019.
[22] Tom B Brown et al. "Language Models are Few-Shot Learners". In: *NeurIPS* (2020).
[23] Jared Kaplan et al. *Scaling Laws for Neural Language Models*. 2020.
[24] Alexey Dosovitskiy et al. "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale". In: *International Conference on Learning Representations (ICLR)* (2021).
[25] Alec Radford et al. "Learning Transferable Visual Models From Natural Language Supervision". In: *International Conference on Machine Learning (ICML)* (2021).
[26] Julius Berner et al. "The Modern Mathematics of Deep Learning". In: *Mathematical Aspects of Deep Learning*. Cambridge University Press, Dec. 2022, pp. 1–111. ISBN: 9781316516782. DOI: 10.1017/9781009025096.002.
[27] Christoph Molnar. *Interpretable Machine Learning*. Fecha de acceso: 2025-06-27. Leanpub, 2022. URL: https://christophm.github.io/interpretable-ml-book/.
[28] Sara Hooker. *On the Limitations of Compute Thresholds as a Governance Strategy*. 2024. arXiv: 2407.05694 [cs.AI].
[29] Daniel Jurafsky and James H. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition with Language Models*. 3rd. 2024.
[30] Adly Templeton et al. *Scaling Laws for Monosemanticity*. 2024.
[31] Netflix Technology Blog. *Foundation Model for Personalized Recommendation*. Fecha de acceso: 2025-06-23. 2025. URL: https://netflixtechblog.com/foundation-model-for-personalized-recommendation-1a0bd8e02d39.

tuid FIRMA DIGITAL
Guillermo Jose Moncecchi Giordano

Christian Emanuel Fachola Garagorry

tuid FIRMA DIGITAL
Alejandro Pablo Cholaquidis Noblia

tuid FIRMA DIGITAL
Marcelo Fiori Schiavo