Capítulo 2

Fundamentos teóricos

Todas las teorías son legítimas y ninguna
tiene importancia. Lo que importa
es lo que se hace con ellas.

Jorge Luis Borges

Resumen: Este capítulo pretende refrescar conocimientos, e introducir otros, para entender las bases teóricas utilizadas en todo el trabajo realizado. Inicialmente, se exhibe la noción de aprendizaje maquinal en sistemas de regresión y clasificación, desde la forma supervisada hasta la no supervisada, y cómo las redes neuronales se utilizan para componer dichos sistemas. Finalmente se profundiza en aspectos del aprendizaje profundo, lo cual en conjunto con todos los fundamentos presentados abarcan las características implementadas para este trabajo.

En los últimos años, el "aprendizaje maquinal" (mejor conocido en inglés como *machine learning*) adquirió bastante popularidad, presentando algoritmos que aproximan en diversas tareas el concepto de inteligencia artificial [6]. Este campo estudia técnicas para construir sistemas capaces de aprender a partir de datos a realizar diversos tipos de tareas sin requerir que se les indique cómo hacerlas. Esto se emplea en una gran cantidad de aplicaciones en donde no es factible diseñar y programar de forma explícita algoritmos para realizar tareas complejas, como visión computacional, motores de búsqueda, reconocimiento de voz, predicción de fraudes, etc.

Los tipos de sistemas que se diseñan para realizar estas tareas mencionadas por lo general siguen dos tipos de aprendizaje: supervisado, y no supervisado [9]. A su vez, dichas tareas a realizar sobre datos se suelen clasificar típicamente en las siguientes categorías: a) clasificación, donde el sistema aprende de forma supervisada a asignar clases; b) regresión, aprendiendo supervisadamente a predecir una variable continua; c) agrupamiento o *clustering*, para dividir los datos en grupos pero de forma no supervisada; d) reducción de dimensiones, mapeando los datos de entrada en un espacio de menor dimensión. En la siguiente sección de este capítulo se detallan los contenidos referidos a la construcción de estos sistemas, para conocer cómo se implementan los algoritmos de aprendizaje maquinal a partir de un conjunto de datos.

2.1. Aprendizaje supervisado

A continuación se procede a detallar la implementación de un sistema con aprendizaje maquinal en forma supervisada para realizar tareas de regresión y clasificación, con el fin de hacer familiar el tratamiento de funciones objetivo, computando sus gradientes y optimizando los objetivos sobre un conjunto de parámetros. Estas herramientas básicas van a formar la base para los algoritmos implementados en el presente trabajo.

2.1.1. Sistemas de regresión y clasificación

En una *regresión lineal* el objetivo es predecir una variable deseada y (la cual puede ser un vector o un escalar, dependiendo de la naturaleza del problema) partiendo de un vector de entrada $\mathbf{x} \in \Re^d$ que por lo general representa las "características" que describen el fenómeno analizado. Para ello, lo usual es contar con un conjunto de N patrones o ejemplos donde cada uno de ellos tiene asociado un vector con características $\mathbf{x}^{(i)}$ y su valor de predicción correspondiente o "etiqueta" $y^{(i)}$, y con ello se busca modelar de forma supervisada (i.e. explicitando la salida deseada) una función $y = h(\mathbf{x})$ tal que $y^{(i)} \approx h(\mathbf{x}^{(i)})$ para cada i-ésimo ejemplo de entrenamiento. Teniendo una cantidad suficiente de patrones, se espera que $h(\mathbf{x})$ sea un buen predictor incluso cuando se le presente un nuevo vector de características donde su correspondiente etiqueta no se conoce.

Para modelar la hipótesis $h(\mathbf{x})$, en cualquier tipo de sistema, se deben definir dos cuestiones: i) cómo se representa la hipótesis y ii) cómo se mide su error de aproximación respecto a la función deseada y. En el caso de la regresión lineal, se define: $h_\theta(\mathbf{x}) = \theta^\top \mathbf{x}$, donde $h_\theta(\mathbf{x})$ representa una gran familia de funciones parametrizadas por θ. Por lo tanto, para encontrar una elección de θ que aproxime $h_\theta(\mathbf{x}^{(i)})$ lo mayor posible a un vector $\mathbf{y}^{(i)}$, se busca minimizar una función de "costo" o "penalización", la cual mide el error cometido en la predicción. Un ejemplo de esta función es utilizar como medida el *Error Cuadrático Medio* (o MSE, del inglés *Mean Squared Error*), la cual se define como:

$$L_i(\theta) = \frac{1}{2} \sum_j (h_\theta(x_j^{(i)}) - y_j^{(i)})^2 \tag{2.1}$$

El valor resultante de esta función corresponde al error de aproximación de $h_\theta(\mathbf{x}^{(i)})$ para un i-ésimo ejemplo dado, y para conocer el error total sobre todo el conjunto de N ejemplos disponibles se promedian todos los errores cometidos tal que $L(\theta) = \frac{1}{N} \sum_i^N L_i(\theta)$. La elección de θ que minimiza esta función de costo se puede encontrar mediante algoritmos de optimización (e.g. gradiente descendiente) que, por lo general, requieren que se compute tanto $L_i(\theta)$ como su gradiente $\nabla_\theta \mathbf{L_i}(\theta)$ (detallado más adelante en la Sección 2.1.2). En este caso, al diferenciar la función de costo respecto al parámetro θ, al aplicar la regla de la cadena el gradiente queda el siguiente producto elemento a elemento:

$$\nabla_\theta \mathbf{L_i}(\theta) = \mathbf{x}^{(i)} \odot (h_\theta(\mathbf{x}^{(i)}) - \mathbf{y}^{(i)}) \tag{2.2}$$

Notar que la constante $\frac{1}{2}$ utilizada en la Ecuación 2.1 es agregada para que sea cancelada al ser derivada dicha función, y con ello se ahorra el cómputo de multiplicar todos los valores del vector dado en la Ecuación 2.2.

En el caso de la regresión lineal, los valores a predecir son continuos. Cuando se tratan problemas de clasificación, la predicción se realiza sobre una variable discreta que puede tomar sólo determinados valores. Para ello, lo que se intenta es predecir la probabilidad de que un ejemplo dado pertenezca a una clase contra la posibilidad de que pertenezca a otra/s. En una *regresión logística* la clasificación se realiza mediante la predicción de etiquetas binarias, por lo cual $y^{(i)} \in \{0, 1\}$. Llamando $z = \theta^\top \mathbf{x}$ al resultado de una función lineal, específicamente se trata de aprender otra función de la forma:

$$\begin{aligned} P(y = 1 \mid \mathbf{x}) &= h_\theta(\mathbf{x}) = \frac{1}{1 + \exp(-z)} \equiv \sigma(z), \\ P(y = 0 \mid \mathbf{x}) &= 1 - P(y = 1 \mid \mathbf{x}) = 1 - h_\theta(\mathbf{x}) \end{aligned} \tag{2.3}$$

La función $\sigma(z)$ es denominada "función logística" o "sigmoidea" (desarrollada en la Sección 2.2.2), y su imagen cae en el rango $[0, 1]$ por lo que $h_\theta(\mathbf{x})$ puede interpretarse como una probabilidad. Por lo general este tipo de regresión se asocia con otra función de costo denominada *Entropía Cruzada*, la cual se define mediante la siguiente expresión:

$$L_i(\theta) = -\left(y^{(i)} \log(h_\theta(\mathbf{x}^{(i)})) + (1 - y^{(i)}) \log(1 - h_\theta(\mathbf{x}^{(i)}))\right)$$

A partir de ello, se puede clasificar un nuevo patrón chequeando cuál de las dos clases resulta con mayor probabilidad. Cuando se deben manejar más de dos clases, la *regresión softmax* es utilizada como clasificador para tratar con etiquetas $y^{(i)} \in \{1, \ldots, K\}$, donde K es el número de clases.

Dado un vector de entrada $\mathbf{x}$, se busca estimar la probabilidad que $P(y = k \mid \mathbf{x})$ para cada valor de $k = 1, \ldots, K$. Entonces, la hipótesis debe resultar en un vector de dimensión K cuyos elementos sean las probabilidades estimadas (y sumen uno). Concretamente, la función $h_\theta(\mathbf{x})$ toma la forma:

$$h_\theta(\mathbf{x}) = \begin{bmatrix} P(y = 1|\mathbf{x}; \theta) \\ P(y = 2|\mathbf{x}; \theta) \\ \vdots \\ P(y = K|\mathbf{x}; \theta) \end{bmatrix} = \frac{1}{\sum_{j=1}^{K} \exp(z^{(j)})} \begin{bmatrix} \exp(z^{(1)}) \\ \exp(z^{(2)}) \\ \vdots \\ \exp(z^{(K)}) \end{bmatrix}$$

Notar que por cada $k \in \{1, \ldots, K\}$ existe un $z^{(k)} = \theta^{(k)\top} \mathbf{x}$ para cada parámetro $\theta^{(k)}$ del modelo, y el término $\frac{1}{\sum_{j=1}^{K} \exp(z^{(j)})}$ normaliza la distribución para que sume uno en total. En cuanto a la función de costo, es similar a la definida para la regresión logística excepto que se debe sumar sobre los K diferentes valores de las clases posibles. Para ello, se puede expresar cada etiqueta $y^{(i)}$ de forma "binarizada" como un vector de dimensión K, que esté compuesto de ceros excepto en la posición correspondiente a la clase apuntada por $y^{(i)}$. Llamando $\mathbf{t}_k^{(i)}$ a este vector, donde para un patrón i está compuesto de ceros excepto en la posición k que vale 1, y recordando que $h_\theta(\mathbf{x}^{(i)})$ es un vector también de dimensión K (por ser la salida de la función *softmax*), se tiene la suma sobre un producto elemento a elemento entre vectores tal que:

$$L_i(\theta) = -\sum_k \left(\mathbf{t}_k^{(i)} \odot \log(h_\theta(\mathbf{x}^{(i)}))\right) \tag{2.4}$$

Notar que esto compone un clasificador lineal, ya que las predicciones sólo se basan en el logaritmo de una distribución de probabilidades [64]. En cuanto al gradiente de dicha función, se puede demostrar que mediante la regla de la cadena se llega a la siguiente expresión simple:

$$\nabla_{\mathbf{z}}\mathbf{L_i}(\theta) = h_\theta(\mathbf{x}^{(i)}) - \mathbf{t}_K^{(i)} \tag{2.5}$$

Recordar que $\nabla_{\mathbf{z}}\mathbf{L_i}(\theta)$ resulta en un vector por ser la derivada de la función de costo $L_i(\theta)$ (que es un escalar) respecto a z, y como ya se dijo ambos se utilizan para la optimización del parámetro θ mediante algoritmos basados en gradientes. También es preciso aclarar que a esta función se le pueden adicionar otros términos que sirvan para agregar penalizaciones al modelo a optimizar, como pueden ser algunas normas regularizadoras (que más adelante se detallan en la Sección 2.3.2.2) las cuales deben tener un gradiente asociado para la optimización mencionada.

2.1.2. Optimización

Una vez que se tiene un modelo parametrizado por un conjunto θ, y una función de costo para medir el error de aproximación a la función deseada y, se procede a optimizarlo para mejorar dicha aproximación en base a un conjunto de datos. Para ello se emplean algoritmos de optimización que, por cada ejemplo del conjunto de datos, utilizan el valor obtenido por la función de costo para actualizar los parámetros del modelo. Este procedimiento se realiza sobre todos los datos disponibles para minimizar el error producido por el modelo, y generalmente se aplica de forma iterativa a través de "épocas".

La idea es que buscar el mejor conjunto de parámetros se puede hacer más fácilmente mediante un refinamiento iterativo, por el cual se parte de un conjunto inicial (designado de alguna forma, como aleatoriamente) que luego se refina iterativamente de forma que en cada pasada éste se mejora un poco para minimizar la función de costo asociada. Este proceso puede interpretarse como recorrer paso a paso un espacio de búsqueda (el de parámetros) donde en cada uno de ellos se busca dirigirse a la región que asegura el mínimo costo posible. Concretamente, se empieza con un θ aleatorio y luego se computan modificaciones $\delta\theta$ sobre el mismo tal que al actualizar a $\theta + \delta\theta$ el costo sea menor.

Dada una función de costo u objetivo $L(\theta)$ y la capacidad de calcular su gradiente respecto a los parámetros $\theta \in \mathbb{R}^d$, el procedimiento de evaluar iterativamente sus valores para actualizar θ se denomina *gradiente descendiente* (GD) [9]. Esta actualización se efectúa en la dirección opuesta del gradiente de la función objetivo, y mediante una tasa de aprendizaje η se define el tamaño de los pasos a realizar hasta el mínimo de esta función, lo cual se resume como:

$$\theta = \theta - \eta\nabla_\theta\mathbf{L}(\theta) \tag{2.6}$$

Aunque existen otras formas de realizar la optimización basada en gradientes (e.g. el método quasi-Newton L-BFGS [11]), el gradiente descendiente es actualmente el más común y estándar para optimizar la función de costo en un modelo (especialmente en redes neuronales).

Cuando el conjunto de datos a utilizar en la optimización es realmente grande, resulta ineficiente computar la función de costo y su gradiente sobre el con-

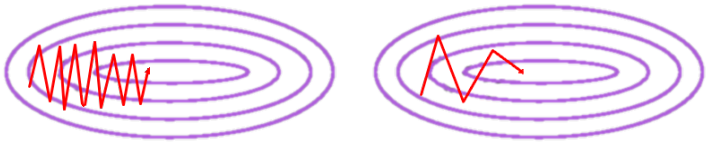


Figura 2.1: Influencia del *momentum* en la optimización por gradiente descendiente (GD). Izquierda, GD sin *momentum*. Derecha, GD con *momentum*.

junto entero por cada actualización a realizar. Es por ello que se suele implementar un enfoque del gradiente descendiente denominado *mini-batch*, en donde cada actualización se computa sobre un subconjunto o "batch" muestreado del conjunto original. En la práctica, el gradiente obtenido del *mini-batch* es una buena aproximación del gradiente total, y con ello se obtiene una convergencia más rápida mediante una actualización de parámetros más frecuente [13].

El caso extremo del *mini-batch* es cuando se utiliza un único ejemplo como batch, y en ese caso el proceso se denomina *gradiente descendiente estocástico* (conocido en inglés como *Stochastic Gradient Descent* o SGD). Aunque esa es su definición técnica, en la práctica se suele referir como SGD al gradiente descendiente con *mini-batch* ya que, en la mayoría de las herramientas de optimización, resulta más eficiente evaluar los gradientes para un subconjunto de datos que para una única entrada a la hora de computar una actualización. En cuanto al tamaño de batch a utilizar, se debe tener en cuenta las restricciones de memoria que existan, aunque por lo general suele ser de entre 10 y 100 ejemplos.

Para acelerar la optimización por GD en una dirección dada, se suele agregar a la Ecuación 2.6 un término de *momentum* que básicamente aplica una fracción γ de la actualización hecha en la época anterior sobre la actual, lo cual resulta:

$$\mathbf{v_t} = \gamma \mathbf{v_{t-1}} + \eta \nabla_\theta \mathbf{L}(\theta) \tag{2.7}$$

$$\theta_t = \theta_{t-1} - \mathbf{v_t} \tag{2.8}$$

En la Figura 2.1 se puede apreciar el efecto de aplicar *momentum* en la optimización, por el cual se disminuyen las oscilaciones al acelerarse el proceso en la dirección relevante [20]. El término γ suele fijarse en 0.9 o un valor similar.

Existe otra forma de aplicar un *momentum*, distinta a la de computar el gradiente sobre el θ actual para aplicar el término en la actualización. El gradiente acelerado de Nesterov (en inglés, conocido como *Nesterov Accelerated Gradient* o NAG) va más allá al calcular el gradiente no sobre el parámetro actual sino sobre una aproximación de su valor futuro de la siguiente forma:

$$\mathbf{v_t} = \gamma \mathbf{v_{t-1}} + \eta \nabla_\theta \mathbf{L}(\theta_{\mathbf{t-1}} + \gamma \mathbf{v_{t-1}}) \tag{2.9}$$

$$\theta_t = \theta_{t-1} - \mathbf{v_t} \tag{2.10}$$

Esto tiene una garantía teórica de convergencia para funciones objetivo convexas [50], y en la práctica suele funcionar bastante mejor respecto al uso del *momentum* estándar.

Todos los enfoques explicados manipulan la tasa de aprendizaje η de forma global y siempre igual para todos los parámetros. *Adagrad* es un algoritmo de

optimización que calcula de forma adaptativa su tasa respecto a los parámetros. Para ello utiliza una tasa de aprendizaje distinta para cada parámetro θ en cada paso, y no una misma actualización para todos a la vez, lo cual se resume como:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\mathbf{G_t} + \epsilon}} \odot \nabla_\theta \mathbf{L}(\theta)_\mathbf{t} \tag{2.11}$$

donde $\odot$ se define como una multiplicación elemento a elemento entre la diagonal de la matriz resultante de operar $\mathbf{G_t}$ con η y ϵ, y los correspondientes elementos del vector gradiente de la función objetivo. Aquí ϵ es un término de suavizado que evita divisiones por cero (usualmente pequeño, en el orden de 1e-8), y $\mathbf{G_t}$ es una matriz diagonal cuya diagonal corresponde a la suma de los gradientes cuadrados actuales (i.e. $\nabla_\theta \mathbf{L}(\theta)_t^2$). Uno de los beneficios principales de esta técnica es que elimina la necesidad de ajustar manualmente la tasa η, y se ha mostrado en estudios que mejora importantemente la robustez de SGD especialmente en redes neuronales profundas [17].

Adadelta es una extensión de *Adagrad* que busca reducir la forma agresiva y monotónicamente decreciente de obtener la tasa de aprendizaje [71]. En lugar de acumular todos los gradientes cuadrados pasados, Adadelta restringe una ventana con un tamaño fijo para acumular estos valores en una suma que se define recursivamente como una media móvil decreciente. Esta media móvil exponencial $E[\nabla_\theta \mathbf{L}(\theta)^2]_t$ en la época t depende sólo del promediado anterior y el gradiente actual tal que:

$$E[\nabla_\theta \mathbf{L}(\theta)^2]_t = \gamma E[\nabla_\theta \mathbf{L}(\theta)^2]_{t-1} + (1-\gamma)\nabla_\theta \mathbf{L}(\theta)_t^2 \tag{2.12}$$

Los gradientes acumulados en la media móvil se deben inicializar en 0 para hacer posible esta actualización recursiva. A partir de ello, respecto a la Ecuación 2.11 de *Adagrad* se cambia la matriz G_t por esta media móvil de forma tal que:

$$\theta_{t+1} = \theta_t + \mathbf{\Delta}\theta, \qquad \mathbf{\Delta}\theta = -\frac{\eta}{\sqrt{E[\nabla_\theta \mathbf{L}(\theta)^2]_t + \epsilon}} \nabla_\theta \mathbf{L}(\theta)_t \tag{2.13}$$

De allí se puede ver que el denominador de la actualización corresponde a la raíz del error cuadrático medio del gradiente (i.e. $RMS[\nabla_\theta \mathbf{L}(\theta)]_t$).

Dado que las unidades en esta actualización no coinciden (ya que deberían tener las mismas hipotéticas unidades que el parámetro en cuestión), los autores del método definieron otra media móvil exponencial pero aplicada sobre el cuadrado de las actualizaciones en lugar del cuadrado de los gradientes. Por lo tanto, ahora el término $RMS[\mathbf{\Delta}\theta]_t$ es desconocido, por lo cual se aproxima con el RMS de las actualizaciones hasta el paso anterior. A partir de esto, la actualización definida para *Adadelta* resulta:

$$\theta_{t+1} = \theta_t + \mathbf{\Delta}\theta, \qquad \mathbf{\Delta}\theta = -\frac{RMS[\Delta\theta]_{t-1}}{RMS[\nabla_\theta L(\theta)]_t} \nabla_\theta L(\theta)_t \tag{2.14}$$

$$RMS[\mathbf{\Delta}\theta]_t = \sqrt{E[\mathbf{\Delta}\theta^2]_t + \epsilon} \tag{2.15}$$

$$E[\mathbf{\Delta}\theta^2]_t = \gamma E[\mathbf{\Delta}\theta^2]_{t-1} + (1-\gamma)\mathbf{\Delta}\theta_t^2 \tag{2.16}$$

Notar que la constante η desaparece de la ecuación original ya que no resulta necesario definir una tasa de aprendizaje para la actualización, aunque en algunas implementaciones se incorpora para tener otro control de la optimización.

Se puede encontrar una buena referencia de estos métodos explicados en distintos artículos y tutoriales de optimización [47] [20] [57] [5].

2.1.3. Métricas de evaluación

Para conocer el comportamiento del modelo optimizado en la tarea asignada, se establecen métricas para medir su desempeño sobre un conjunto de datos y a partir de ello poder ajustar el mismo para mejorar sus resultados. Dichas métricas son específicas del tipo de problema tratado, por lo que se distinguen para tareas de *clasificación* y *regresión*.

2.1.3.1. Clasificación

En problemas supervisados de clasificación, cada patrón de un conjunto de datos tiene asignada una etiqueta de la clase que debe predecir el modelo. A raíz de ello, los resultados para cada patrón corresponden a cuatro categorías:

- Verdadero Positivo (VP): la etiqueta es positiva y la predicción también.
- Verdadero Negativo (VN): la etiqueta es negativa y la predicción también.
- Falso Positivo (FP): la etiqueta es negativa pero la predicción es positiva.
- Falso Negativo (FN): la etiqueta es positiva pero la predicción es negativa.

La forma más sencilla de medir el desempeño de una clasificación es calcular su exactitud mediante la cantidad de aciertos obtenidos para la clase dada sobre el total de las predicciones hechas (medida conocida como *accuracy* o ACC). No obstante, esta medida no es buena especialmente cuando se tratan conjuntos de datos no balanceados por clases. Si se quiere modelar un predictor de anomalías, es muy probable que los datos utilizados tengan más ejemplos de comportamiento normal que de algo anómalo, y si se obtiene un 95 % de predicciones correctas sobre el total no hay seguridad de que el modelo sea bueno si el 5 % restante abarca muchas o todas las anomalías no predichas.

Para evitar esto, se definen medidas de *precisión* y *sensibilidad* (mejor conocidas en inglés como *precision* y *recall*) que tienen en cuenta el "tipo" de error cometido. En tareas de clasificación, el valor de *precision* P determina para una clase la cantidad de Verdaderos Positivos dividido por el número total de elementos clasificados como pertenecientes a dicha clase (i.e. la suma de Verdaderos Positivos y Falsos Positivos), mientras que el valor de *recall* R representa la cantidad de Verdaderos Positivos predichos sobre el total de elementos que realmente pertenecen a la clase en cuestión (i.e. la suma de Verdaderos Positivos y Falsos Negativos) [53]. Por lo tanto, sus expresiones quedan dadas por:

$$P = \frac{VP}{VP + FP} \qquad R = \frac{VP}{VP + FN} \tag{2.17}$$

En un contexto de recuperación de información, la *precision* determina la cantidad de elementos relevantes del total que fueron recuperados, mientras que el *recall* representa la fracción de elementos recuperados del total que son relevantes. Una medida que combina a estas dos mediante una media armónica es el Valor-F (mejor conocida e inglés como F–Score o F–Measure). La misma

ofrece un balance entre *precision* y *recall* ajustado por un parámetro β, y un caso muy utilizado de esta medida es el F1-Score donde $\beta = 1$, tal que:

$$F(\beta) = (1 + \beta^2)(\frac{PR}{\beta^2 P + R}) \qquad F1 = \frac{2PR}{P + R} \tag{2.18}$$

En caso de que el sistema esté diseñado para tratar problemas multi-clase (dos o más clases), las métricas deben ajustarse para soportar esta característica [70]. Para ello, se procede a calificar positiva o negativa a una predicción respecto a la etiqueta en base al contexto de una clase en particular. Cada tupla de etiqueta y predicción se evalúa para cada una de las clases, y se considera como positiva para dicha clase o negativa para el resto de las clases. Así, un verdadero positivo ocurre cuando la predicción y la etiqueta coinciden, mientras que un verdadero negativo se da cuando ni la predicción ni la etiqueta corresponden a la clase tomada en cuenta. Es por ello que para un simple patrón resultan múltiples verdaderos negativos en problemas de más de dos clases (y la misma idea se extiende para caracterizar FNs y VNs). Para seguir este enfoque de múltiples etiquetas posibles, se derivan las medidas ya definidas para evaluar la clasificación respecto a todas las clases en dos formas posibles [60]:

I) Computar el promedio de las mismas medidas calculadas por cada una de las clases (*macro-promediado*).

II) La suma de cuentas para obtener VP, FP, VN, FN acumulativos, y a ello aplicar una métrica de evaluación (*micro-promediado*).

En la Tabla 2.1 se exponen las medidas de evaluación explicadas utilizando estas aproximaciones, donde el subíndice μ representa a aquella medidas con *micro-promediado* y el subíndice M a aquellas con *macro-promediado*. Notar que esta última aproximación para evaluar una clasificación abarca y generaliza las métricas explicadas para problemas de dos clases.

Una forma visual de representar el desempeño del modelo en la clasificación hecha es mediante una *matriz de confusión*, en la cual se presentan los resultados de las predicciones en cantidades por cada una de las clases en cuestión. Dicha matriz cuadrada tiene dimensión igual a la cantidad de clases tratada, y cada columna corresponde a las predicciones hechas mientras que cada fila representa las instancias de la clase actual u original a predecir. Esta disposición de los resultados facilita su interpretación (de allí proviene el nombre, ya que se conoce rápidamente en qué clase/s se confunde el predictor) y además permite derivar rápido los cálculos de las métricas explicadas. En el caso de dos clases, es sencillo expresar las categorías de resultados sobre cada celda de la matriz, y para el caso multi-clase se representan de la forma mencionada anteriormente tal como se puede apreciar en la Tabla 2.2 cuando se tienen 3 clases. De allí se puede notar que una clasificación perfecta resulta en una matriz de confusión diagonal.

Tabla 2.1: Medidas de evaluación en problemas de clasificación multi-clases.

Medida	Fórmula	Sentido de la evaluación
ACC	$\frac{\sum_{i=1}^{C} \frac{VP_i+VN_i}{VP_i+FN_i+FP_i+VN_i}}{C}$	Promedio de la efectividad por clase del clasificador
P_μ	$\frac{\sum_{i=1}^{C} VP_i}{\sum_{i=1}^{C}(VP_i+FP_i)}$	Suma de la precisión lograda en la clasificación por cada clase.
R_μ	$\frac{\sum_{i=1}^{C} VP_i}{\sum_{i=1}^{C}(VP_i+FN_i)}$	Suma de la sensibilidad lograda en la clasificación por cada clase.
$F_\mu(\beta)$	$(1+\beta^2)(\frac{P_\mu R_\mu}{\beta^2 P_\mu + R_\mu})$	Balance entre precisión y sensibilidad basada en la suma de decisiones realizadas por clase.
P_M	$\frac{\sum_{i=1}^{C} \frac{VP_i}{VP_i+FP_i}}{C}$	Promedio de los cálculos de precisión lograda por cada clase.
R_M	$\frac{\sum_{i=1}^{C} \frac{VP_i}{VP_i+FN_i}}{C}$	Promedio de los cálculos de sensibilidad lograda por cada clase.
$F_M(\beta)$	$(1+\beta^2)(\frac{P_M R_M}{\beta^2 P_M + R_M})$	Balance entre precisión y sensibilidad basada en un promedio sobre el total de clases.

Tabla 2.2: Esquema de matriz de confusión para evaluar predicciones sobre 3 clases.

		Predicción		
Actual		VP_1	FN_1 FP_2	FN_1 FP_3
		FP_1 FN_2	VP_2	FN_2 FP_3
		FP_1 FN_3	FP_2 FN_3	VP_3

2.1.3.2. Regresión

Cuando la variable a predecir es de naturaleza continua, el modelo a construir compone un sistema de regresión. Para ello, la forma de conocer la precisión en la predicción no debe hacerse por cantidad de aciertos sino midiendo un error en la salida. Las medidas más utilizadas para ello son el *Error Cuadrático Medio* (en inglés, *Mean Squared Error* o MSE), el *Error Absoluto Medio* (en inglés, *Mean Absolute Error* o MAE) y el coeficiente de determinación R^2 [46].

$$MSE = \frac{\sum_{i=0}^{N-1}(y_i - \hat{y}_i)^2}{N} \qquad RMSE = \sqrt{\frac{\sum_{i=0}^{N-1}(y_i - \hat{y}_i)^2}{N}} \tag{2.19}$$

$$MAE = \frac{\sum_{i=0}^{N-1}|y_i - \hat{y}_i|}{N} \qquad RMAE = \sqrt{\frac{\sum_{i=0}^{N-1}|y_i - \hat{y}_i|}{N}} \tag{2.20}$$

$$R^2 = 1 - \frac{MSE}{VAR(y) \cdot (N-1)} = 1 - \frac{\sum_{i=0}^{N-1}(y_i - \hat{y}_i)^2}{\sum_{i=0}^{N-1}(y_i - \bar{y_i})^2} \tag{2.21}$$

El término error aquí representa la diferencia entre el valor verdadero y_i y el predicho $\hat{y}_i$. Para ello se calcula una suma de los residuos, dividida por el número de grados de libertad N. Esto pueder verse como la media de las desviaciones de cada predicción respecto a los verdaderos valores, generadas por un modelo estimado durante un espacio de muestra particular. Valores de error bajos significan que el modelo es más preciso en sus predicciones, y un error total de 0 indica que el modelo se ajusta a los datos perfectamente. Tanto el cuadrado como el valor absoluto del error medido en cada suma captura la magnitud total del mismo, ya que algunas diferencias pueden ser negativas. Se suele utilizar la raíz cuadrada de el error calculado para hacerlo independiente de la escala para la comparación de distintos modelos.

El coeficiente de determinación R^2 provee una medida de cuán bien se ajusta el modelo a los datos. El mismo puede ser interpretado como la proporción de la variación explicada por el modelo. Cuanto mayor es dicha proporción, mejor es el modelo en sus predicciones, siendo que el valor 1 indica un ajuste perfecto.

2.1.4. Ajuste de hiperparámetros

Como puede notarse, durante el diseño de un modelo existen diversos parámetros y configuraciones que se deben especificar en base a los datos tratados y la tarea asignada para dicho modelo. Estos pueden ser valores continuos de los que se puede tener una idea de rango posible (e.g. tasa de aprendizaje para el algoritmo de optimización) o categorías particulares de algún componente (e.g. algoritmo de clasificación a utilizar). La configuración elegida es crucial para que el proceso de optimización resulte en un modelo con buen desempeño en la tarea asignada, y en el caso de los hiperparámetros (así se les denomina a los valores a ajustar en una configuración) elegir un valor determinado puede ser difícil especialmente cuando son sensibles en su variación.

Una forma asistida para realizar determinar una buena configuración es implementar un algoritmo de búsqueda de hiperparámetros, el cual realiza elecciones particulares tomando muestras sobre los rangos o categorías posibles que se definan, así luego se optimiza un modelo por cada configuración especificada. Opcionalmente, también se puede utilizar *validación cruzada* [44] para estimar la generalización del modelo obtenido con la configuración y su independencia del conjunto de datos tomado. Resumiendo, una búsqueda consta de:

- Un modelo estimador (de regresión o clasificación).

Figura 2.2: Tipos de ajuste que puede lograr un modelo sobre los datos que utiliza para su entrenamiento.

- Un espacio de parámetros.
- Un método para muestrear o elegir candidatos.
- Una función de evaluación del modelo.
- (Opcional) Un esquema de validación cruzada.

Una forma de muestrear el espacio delimitado de parámetros es tomando de manera exhaustiva los valores que caen en una grilla, la cual generalmente se determina por una discretización equiespaciada del espacio tratado. A este método se le denomina "búsqueda por grilla" o *grid search* en inglés, y se caracteriza por su simplicidad al ser fácil de implementar aunque por ello es más propenso a sufrir un problema denominado *maldición de la dimensionalidad*. Este último señala que a medida que el espacio de búsqueda es mayor, la estrategia se hace menos eficiente ya que requiere una cantidad mucho mayor de muestras necesarias para obtener mejores candidatos [21].

Otra forma que evade buscar exahustivamente sobre el espacio de parámetros (lo cual también es potencialmente costoso si dicho espacio es de una dimensión alta), es la de muestrear una determinada cantidad de veces el espacio delimitado, en forma aleatoria y no sobre una grilla determinada. Este método se denomina "búsqueda aleatoria" o *random search* en inglés, y es tan fácil de implementar como el *grid search* aunque se considera más eficiente especialmente en espacios de gran dimensión [8].

2.1.5. Control de la optimización

El ajuste de hiperparámetros y la mejora de las configuraciones y elecciones hechas en el diseño buscan que el modelo a construir tenga el mejor desempeño posible. Sin embargo, esto no se puede efectuar sobre el conjunto de datos con el cual se ajusta el modelo (llamado "conjunto de entrenamiento") ya que así no puede garantizarse que el modelo generalice y se desempeñe aproximadamente igual con otro conjunto de datos que no se le presentó nunca. Esta cuestión introduce un problema denominado "sobreajuste" (mejor conocido en inglés como *overfitting*), por el cual un modelo optimizado se desempeña muy bien con los datos que utilizó en su ajuste, pero ocurre lo contrario sobre otro conjunto de

datos que no haya "visto" o usado jamás. Este problema siempre trata de ser evitado ya que el desempeño obtenido no es representativo sobre casos en los que se pretenda utilizar el modelo en un escenario real. La Figura 2.2 esquematiza cómo se desempeña el modelo sobre los datos de entrenamiento cuando ocurre el fenómeno de sobreajuste.

Para mitigar el *overfitting* se debe contar con un "conjunto de validación", construido a partir del total de datos disponibles al separar una porción para este propósito de validación. Este conjunto no se utiliza para entrenar el modelo, sino que durante dicho proceso sirve para evaluar y monitorear que el modelo está generalizando y no se está ajustando demasiado a los datos de entrenamiento. También se necesita definir un "conjunto de prueba", el cual no debe ser usado nunca durante el modelado ya que representa datos que se le van a presentar al modelo luego de que ya haya sido construido y que seguramente no son iguales a los que el mismo utilizó durante su entrenamiento. En cuanto a la magnitud a definir para cada porción elegida, en términos del conjunto total de datos, lo que se suele realizar es partir en 70 % para entrenamiento, 15 % para validación y el 15 % restante para prueba.

Finalmente, durante la optimización se precisa la información del desempeño obtenido en el modelo para conocer cuándo es lo suficientemente bueno como para frenar el proceso. (ya que generalmente no se obtienen niveles óptimos) . Es por eso que suelen establecerse ciertas reglas o criterios de corte para que la optimización se realice hasta lograrse un nivel de desempeño definido, o bien frenarla cuando no se está obteniendo un ajuste deseable.

Concluyendo, el ajuste de parámetros (en forma manual o asistida con algoritmos de búsqueda) se realiza para optimizar el modelo sobre los datos de entrenamiento, y el desempeño obtenido con dicha configuración se evalúa con un conjunto de validación (que no debe ser utilizado para ajustar el modelo). Una vez que se encuentra la mejor configuración, se establece como fija y allí se evalúa sobre los datos de prueba para conocer finalmente el desempeño del modelo construido.

2.2. Redes Neuronales Artificiales

Dentro del campo científico de la Inteligencia Artificial, las *Redes Neuronales Artificiales* (RNA) comprenden una rama antigua pero destacada, especialmente en la actualidad luego del surgimiento del aprendizaje profundo. Se entiende como RNA a un sistema con elementos procesadores de información de cuyas interacciones locales depende su comportamiento en conjunto [30]. Dicho sistema trata de emular el comportamiento del cerebro humano, adquiriendo conocimiento de su entorno mediante un proceso de aprendizaje y almacenándolo para disponer de su uso [31].

Las RNAs son implementadas en computadoras para imitar la estructura neuronal de un cerebro, tanto en programas de software como en arquitecturas de hardware, por lo cual se utilizan para componer un sistema de aprendizaje maquinal. No obstante, sólo consiste en una aproximación debido a las diferencias significativas que se presentan en la Tabla 2.3 [48]. Por lo general, las computadoras presentan una arquitectura de tipo Von Neumann basada en un microprocesador muy rápido capaz de ejecutar en serie instrucciones complejas

Tabla 2.3: Diferencias entre cerebro humano y computadora convencional

Características	Cerebro humano	Computadora convencional
Velocidad de proceso	Entre 10^{-3} y 10^{-2} seg	Entre 10^{-9} y 10^{-8} seg
Nivel de procesamiento	Altamente paralelo	Poco o nulo paralelizado
Número de procesadores	Entre 10^{11} y 10^{14}	Entre 4 y 8
Conexiones	10.000 por procesador	Pocas
Almacenamiento del conocimiento	Distribuido	En posiciones precisas
Tolerancia a fallos	Amplia	Poca o nula
Consumo de energía en una operación/seg	10^{-16} Julios	10^{-6} Julios

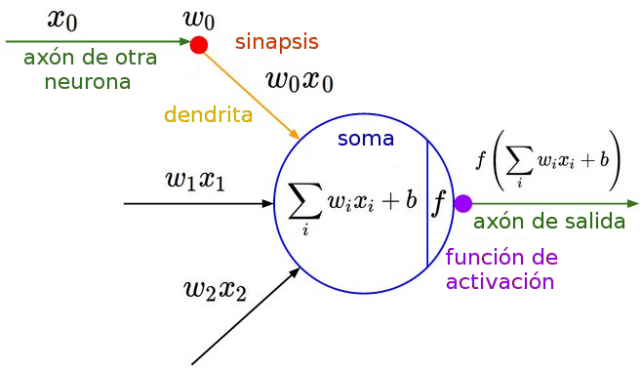


Figura 2.3: Modelo matemático de una neurona.

de forma fiable, mientras que el cerebro está compuesto por millones de procesadores elementales o neuronas, que se interconectan formando redes. Además, las neuronas biológicas no adquieren conocimiento por ser programadas sino que lo hacen a partir de estímulos que reciben de su entorno, y operan mediante un esquema masivamente paralelo distinto al serializado o poco paralelo de la computadoras convencionales.

2.2.1. Arquitectura

La unidad básica de cómputo en el cerebro es una neurona, la cual recibe señales de entrada desde sus dendritas y las procesa en su cuerpo, llamado soma, para producir señales de salida mediante un único axón. Estas últimas a su vez interactúan por sinápsis con las dendritas de otras neuronas y con ello se logra la comunicación de estímulos en todo el cerebro.

Para modelar el comportamiento de las neuronas, la idea es que las sinápsis que producen pueden controlarse mediante *pesos sinápticos* que definan una magnitud de la influencia que ejerce una con otra (y también dirección, al poder excitar o inhibir mediante pesos positivos o negativos, correspondientemente).

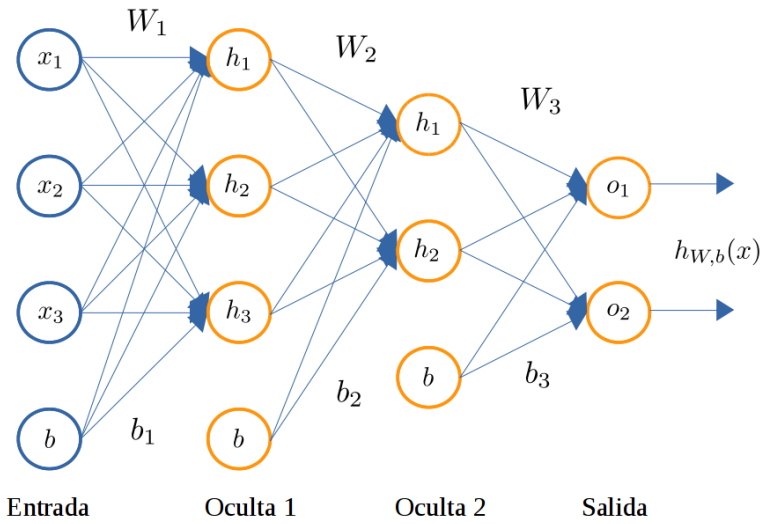


Figura 2.4: Arquitectura básica de un MLP con 4 capas.

Los pesos sinápticos w_0 multiplican las señales de entrada x_0 que llegan por las dendritas para ejercer una suma ponderada en el soma, y si el resultado supera un cierto umbral la neurona se "activa" enviando un estímulo a lo largo de su axón. Dicha suma puede incorporar además un término de sesgo b que participa sin multiplicarse por la entrada. En este modelo se considera que no interesa conocer el preciso tiempo en que se activa la neurona sino la simple ocurrencia de su activación, lo cual es representado mediante una *función de activación* [31]. En la Figura 2.3 se representa gráficamente el modelo explicado, el cual constituye lo que se denomina como "perceptrón simple'.

Para modelar una red neuronal artificial las neuronas se agrupan en capas, de forma tal que todas las unidades de una capa se conectan con todas las neuronas de sus capas próximas para formar una estructura interconectada. En la forma más simple, se tiene una capa de entrada que proyecta la señal entrante en una capa de salida. Cuando se incorporan capas intermedias (denominadas capas ocultas), la red adquiere más niveles de procesamiento entre su entrada y salida, y lo que se forma es un "perceptrón multicapa" (conocido en inglés como *Multi Layer Perceptron* o MLP) [31]. En esta configuración, para producir la salida de la red se computan sucesivamente todas las activaciones una capa tras otra, donde puede notarse que una red con n capas equivale a tener n perceptrones simples en cascada, donde la salida del primero es la entrada del segundo y así sucesivamente. Además, cada capa puede tener diferente número de neuronas, e incluso distinta función de activación.

Siguiendo el modelo matemático de un perceptrón simple, los pesos sinápticos ahora pueden expresarse en conjunto de forma vectorial como matrices, así como también los pesos del sesgo se representa como un vector. Por lo tanto, dada un vector de entrada $\mathbf{x}$, la suma ponderada efectuada en una capa se expresa como un producto entre la matriz de pesos sinápticos $\mathbf{W}$ y dicha entrada $\mathbf{x}$, en la cual también se suma el vector de sesgo o *bias* $\mathbf{b}$. Al resultado de esto se le

aplica la función de activación, produciendo así la salida final de la capa. En la Figura 2.4 se representa la arquitectura de un perceptrón multicapa, mostrando las interacciones que tienen sus unidades desde la entrada hasta su salida.

2.2.2. Funciones de activación

Una función de activación determina cómo se transforman las entradas a través de la red neuronal, lo cual es determinante para que logre su capacidad de aprender funciones complejas. En general, se caracterizan por ser *no lineales* ya que incrementan el poder de expresión y con ello se pueden obtener interesantes representaciones de las entradas que ayuden a la tarea designada para la red. La razón por la cual no es conveniente que sean lineales es porque de esa forma no tendría sentido que la red posea más de una capa, ya que la combinación de funciones lineales tiene un resultado lineal. Además, las redes neuronales están pensadas principalmente para tratar tanto problemas de clasificación en donde las clases no son linealmente separables como problemas en donde no se logra una precisión deseable mediante un modelo lineal.

Como se anticipó anteriormente en la arquitectura de una red, cada unidad o neurona de una capa recibe una entrada que es ponderada por sus pesos sinápticos para luego producir una salida activada que sirve como entrada a todas las neuronas de la capa siguiente (o en el caso de ser la última capa, que es el resultado final de la activación completa de la red). Dado un vector $\mathbf{x}$ de entrada para una capa de la red, se le aplica una transformación afín (i.e. una transformación lineal por la matriz de pesos $\mathbf{W}$, seguido de una traslación por el vector $\mathbf{b}$) cuyo resultado es la salida lineal $\mathbf{z}$. Dicha salida es la que recibe la función de activación f para producir la salida final de la capa $\mathbf{a}$, lo cual se expresa como :

$$\begin{aligned} \mathbf{z} &= \mathbf{W}\mathbf{x} + \mathbf{b} \\ \mathbf{a} &= f(\mathbf{z}) \end{aligned} \tag{2.22}$$

Originalmente las funciones de activación más utilizadas eran las sigmoideas, las cuales son la *sigmoidea* (σ) y la *tangente hiperbólica* ($tanh$). Las mismas tienen una inspiración biológica y están delimitadas por un mínimo y un máximo valor, lo cual causa que las neuronas se saturen en las últimas capas de la red neuronal [64]. Ambas funciones se definen analíticamente como.

$$\sigma(z) = \frac{1}{1+e^{-z}}, \qquad tanh(z) = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}} \tag{2.23}$$

Otra función de activación que ha sido muy utilizada en muchas aplicaciones es la *lineal rectificada* o ReLU (*Rectifier Linear Unit*), la cual comprende una no linealidad simple: resulta en 0 para entradas negativas, y para valores positivos se mantiene intacta, por lo cual no tiene valores límites como las funciones sigmoideas. El gradiente de la ReLU es 1 para todos los valores positivos y 0 para los negativos, lo cual hace que, durante la optimización de la red, los gradientes negativos no sean usados para actualizar los pesos sinápticos correspondientes. Además, el hecho de que el gradiente sea 1 para cualquier valor positivo hace que el entrenamiento sea más rápido que con otras funciones de activación no lineales. Por ejemplo, la función sigmoidea tiene muy pequeños gradientes para

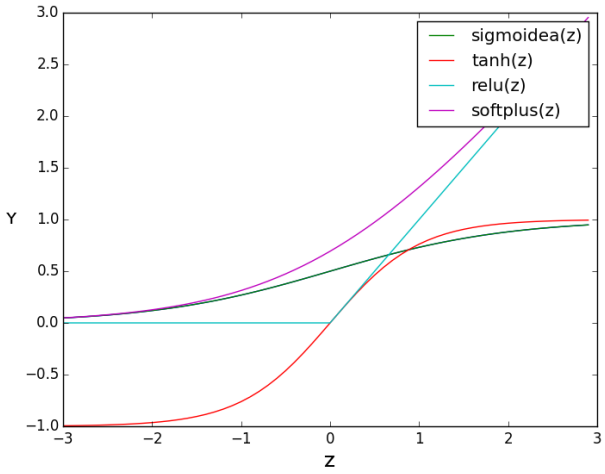


Figura 2.5: Visualización de las funciones de activación para $-3 \leq z \leq 3$.

grandes valores positivos y negativos, por lo que el aprendizaje practicamente se frena o "estanca" en dichas regiones [19]. Es preciso notar que las ReLUs poseen una discontinuidad en 0, por lo cual no es derivable allí la función. No obstante se fuerza a que allí la derivada sea igual a 0, y el hecho de que allí la activación sea 0 otorga buenas propiedades de ralez a la red [64]. Una función que aproxima a la ReLU es la *softplus*, que además de ser continua su derivada es la función *sigmoidea*. Ambas funciones entonces se expresan como:

$$relu(z) = max(0, z), \qquad softplus(z) = \log\left(1 + e^{z}\right) \tag{2.24}$$

En la Figura 2.5 se visualizan las funciones de activación explicadas en un dominio definido, mientras que en la Tabla 2.4 se presentan todas las funciones de activación mencionadas con la respectiva derivada de cada una.

Tabla 2.4: Funciones de activación desarrolladas, detallando para cada una tanto su expresión la de su respectiva derivada analítica.

Nombre	**Función**	**Derivada**
Sigmoidea	$f(z) = \dfrac{1}{1+e^{-z}}$	$f'(z) = \dfrac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$
Tangente Hiperbólica	$f(z) = \dfrac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$	$f'(z) = 1 - tanh^{2}(z)$
ReLU	$f(z) = max(0, z)$	$f'(z) = \begin{cases} 1 & z > 0 \\ 0 & z \leq 0 \end{cases}$
Softplus	$f(z) = \log\left(1 + e^{z}\right)$	$f'(z) = \sigma(z) = \dfrac{1}{1+e^{-z}}$

En el caso de que la red neuronal tratada esté diseñada para realizar tareas de clasificación, se debe tener en cuenta que la última capa tenga una salida

conveniente para ello. Es por eso que la función allí debe ser de clasificación, y para ello se suele utilizar la función *softmax* explicada en la Sección 2.1.

2.2.3. Retropropagación

La propagación hacia atrás de errores o retropropagación (en inglés, *back-propagation*) es un algoritmo de aprendizaje utilizado para efectuar el entrenamiento de redes neuronales. Fue introducido originalmente en la década del 1970, pero cobró realmente importancia y utilidad en el 1986 mediante una publicación que describía el trabajo con distintas redes neuronales donde este algoritmo alcanzaba un aprendizaje bastante más rápido que otros enfoques [68]. A raíz de ello se logró resolver problemas que antes no estaban resueltos, y actualmente es casi un estándar para la optimización de redes neuronales.

El procedimiento consiste en que, dado un patrón (x, y), primero se realiza un "paso hacia adelante" para computar todas las activaciones de cada capa a través de la red hasta calcular la salida final de la misma. A partir de esto se puede utilizar la salida deseada y para computar el valor de la función objetivo y su gradiente respecto a la salida, los cuales son necesarios para conocer cuánto deben variar todos los parámetros de la red. Para ello, se calcula en cada unidad i de cada capa l un "término de error" que mide cuánto afectó cada una en las salidas calculadas. Para la capa de salida, dicho término se calcula en base al gradiente ya computado, y a partir de ello se efectúa el "paso hacia atrás" del algoritmo de la siguiente forma: desde la penúltima capa hasta la primera (sin contar la de entrada), se computa cada término de error en base al correspondiente de la capa siguiente (usado para multiplicar los pesos sinápticos dados) y al gradiente de la activación dada, y a partir de ello se puede computar el gradiente de la función objetivo respecto a los parámetros de la red tratados. Se puede notar que el término de error se va propagando desde el final de la red hasta el principio para poder computar el gradiente de la función objetivo respecto a cada parámetro de la red, y en dicho cálculo se aplica la regla de la cadena sucesivamente para derivar estos valores desde las activaciones obtenidas.

Finalmente, el resultado de este algoritmo es el valor de la función de costo y su gradiente respecto a todos los parámetros de la red, lo cual es de utilidad en algoritmos de optimización basados en gradientes como los descriptos en la Sección 2.1.2. Detalles específicos de la implementación se pueden encontrar en tutoriales de redes neuronales [52] [51], y en el Algoritmo 2 del Apéndice A se resume este proceso explicado.

2.3. Aprendizaje profundo

A partir de la introducción sobre *deep learning* realizada en la sección Resumen de este trabajo, así como los antecedentes de sus aplicaciones exitosas mencionados en la Sección 1.1, en los siguiente apartados se procede a profundizar acerca de las características que ofrece en el modelado a diferencia de las redes neuronales básicas ya detalladas.

2.3.1. Redes Neuronales Profundas

Existen ciertas particularidades en las redes profundas que despertaron su interés e incrementaron su estudio. Principalmente, se puede demostrar que hay funciones que una red de n capas puede representar de forma compacta (con un número de unidades ocultas que es polinomio del número de entradas) pero una red de $n-1$ capas no puede representar a menos que tenga una gran cantidad exponencial de unidades ocultas, y esto quiere decir que las redes profundas pueden representar significativamente más conjuntos de funciones que las redes de una o pocas capas ocultas [51]. Además, una red profunda puede aprender a representar los datos mediante descomposiciones por partes.

La profundidad definida para una red neuronal en la práctica es arbitraria, y depende mucho de la tarea a realizar y los datos disponibles para el ajuste: si la red es poco profunda (e.g. 2 o 3 capas), la misma tendrá menor poder de representación y además se corre el riesgo de *overfitting* si la cantidad de unidades en la capa oculta es grande respecto a la dimensión de entrada; si la red es bastante profunda (e.g. 10 o más capas), se tiene mayor capacidad para el aprendizaje, aunque la gran cantidad de niveles en la red puede ocasionar un problema típico de este modelado denominado *vanishing gradient*. Este último ocurre en el entrenamiento de redes neuronales mediante aprendizaje basado en gradiente y retropropagación, y afecta no sólo a las del tipo multicapa sino también a aquellas del tipo recurrente [37].

El *vanishing gradient* se debe a que la señal de error a retropropagar para el ajuste de parámetros decrece exponencialmente con la cantidad de capas, por lo cual las capas que estén más cerca de la entrada se entrenan muy lentamente. Las funciones de activacion utilizadas influyen bastante en este problema: si la imagen del gradiente abarca valores chicos (e.g. sigmoidea, tangente hiperbólica), se corre mayor riesgo de que se "desvanezcan" las actualizaciones para las primeras capas; si dicha imagen comprende valores altos (e.g. ReLU), existe el riesgo de que las actualizaciones sean inestables y dificulten la convergencia de la optimización (problema denominado *exploding gradient*) [52].

A raíz de esto, se originaron varias propuestas para mitigar este problema:

1. El "pre-entrenamiento" de las redes neuronales mediante aprendizaje no supervisado para inicializar los pesos sinápticos capa-por-capa posibilitó arquitecturas de múltiples niveles que sólo requerían de un pequeño ajuste en forma supervisada para obtener buenos resultados [6] [22].

2. Las redes recurrentes LSTM (*Long short-term memory*) componen una arquitectura específicamente diseñada para combatir el *vanishing gradient* [38], y actualmente son implementadas a nivel industrial en sistemas de visión computacional y reconocimiento de voz debido a la gran precisión que obtiene en dichas tareas.

3. El aprendizaje residual compone una metodología propuesta para entrenar redes de gran profundidad (de cientos o miles de capas) disminuyendo importantemente el problema mencionado y mostrando resultados competitivos en tareas de visión computacional [33].

En las siguientes secciones se profundiza acerca de la primer propuesta mencionada, pero primero se procede a detallar un procedimiento realizado en cualquier tipo de red neuronal para mejorar la calidad del ajuste de parámetros.

2.3.2. Tratamiento sobre los pesos sinápticos

Para mitigar el problem de *overfitting* mencionado, especialmente cuando la red tiene tantos parámetros libres (i.e. pesos sinápticos y sesgo) que pueden ajustarse demasiado a los datos de entrenamiento, siempre resulta conveniente que estos parámetros reciban un tratamiento apropiado desde que se instancian hasta que se optimizan. A continuación se describen dos formas de realizar esto, las cuales adquirieron especial importancia con el origen del aprendizaje profundo debido a la cantidad de parámetros que poseen las redes de ese tipo.

2.3.2.1. Inicialización

La inicialización de los pesos sinápticos en una red neuronal influye mucho en su desempeño y el tiempo que se requiere para optimizarlo. Lo deseable es que los pesos sinápticos se inicialicen con valores cercanos (pero no iguales) a 0, por lo cual puede pensarse en que dichos valores se obtengan de un muestreo sobre una distribución de probabilidades que tenga media igual a 0 y una varianza pequeña para que los valores sean cercanos a 0. Dicha distribución puede ser normal o uniforme, y se ha comprobado que en la práctica la elección de una u otra tiene relativamente poco impacto en el desempeño final. En cuanto a que los valores sean pequeños, se debe tener cuidado ya que eso implica también que, durante la retropropagación, los gradientes utilizados para actualizar los pesos también sean pequeños (ya que son proporcionales) y con ello las actualizaciones se "desvanezcan" en la propagación, especialmente con redes profundas.

Por lo tanto para inicializar los pesos de esta forma se debe controlar la varianza de la distribución a muestrear, y que además su valor tenga relación con la dimensión de entrada que tienen los pesos sinápticos. Una recomendación es la de escalar la varianza a $\frac{1}{\sqrt{n}}$, siendo n el número de entradas que tiene la matriz de pesos [47]. En la práctica, es muy utilizado que la varianza sea $\sqrt{\frac{2}{n}}$, lo cual muestra buen comportamiento en redes neuronales profundas (especialmente cuando la función de activación es una ReLU) [32]. Otra forma de inicializar los pesos, recomendada para las funciones de activación sigmoideas, es la de utilizar para el muestreo una distribución uniforme que esté en el rango $\pm\sqrt{\frac{6}{n_{in}+n_{out}}}$ para la función Tanh, y en el rango $\pm 4{,}0\sqrt{\frac{6}{n_{in}+n_{out}}}$ para la sigmoidea [29]. En cuanto al vector de sesgo, por lo general se suelen inicializar todos sus valores iguales (o muy aproximado, según algunos trabajos) a 0. No se requiere ninguna técnica de muestreo ya que, según muchos estudios, el mayor impacto en la inicialización de los parámetros está dado por los pesos sinápticos [47].

2.3.2.2. Regularización

Como se ha dicho anteriormente, es deseable que las redes neuronales sean capaces de generalizar las aptitudes adquiridas durante el entrenamiento para tener un buen desempeño al presentarse patrones nunca vistos. Para prevenir el problema del *overfitting*, un buen tratamiento es incoporar términos de regularización sobre los pesos sinápticos. Con ello se penaliza la complejidad del modelo en términos del ajuste a los datos de entrenamiento, de forma que pueda obtenerse generalización sobre los datos de prueba. Entre las formas de regula-

rización más utilizadas, se destacan tres técnicas:

Norma L_1

Término que se agrega a la función objetivo a optimizar en la red neuronal, y que tiene la particularidad de conducir a que la matriz de pesos sea "rala" (es decir, que algunos valores sean muy cercanos o iguales a 0). Esta propiedad puede ser deseable para que se utilicen sólo un subconjunto ralo de las entradas más importantes y se produzca robustez ante entradas con ruido [47]. Agregando este término, la función de costo y su gradiente quedan:

$$\begin{aligned} L &= L_0 + \lambda_1 \sum_l \sum_i \sum_j |W_{ji}^{(l)}| \\ \nabla L &= \nabla L_0 + \lambda_1 \sum_l \sum_i \sum_j sign(W_{ji}^{(l)}) \end{aligned} \tag{2.25}$$

Aquí, se define a $sign(x)$ como una función que retorna 1 si x es positivo, -1 si es negativo ó 0 en caso que x sea nulo.

Norma L_2

Es la regularización más común que se incorpora en los pesos de una red neuronal, y tiene el efecto de penalizar fuertemente las matrices de pesos con picos o diferencias importantes entre valores, con lo cual se fuerza a que los pesos sean pequeños [52]. Al incorporar este término en el costo de la red resulta:

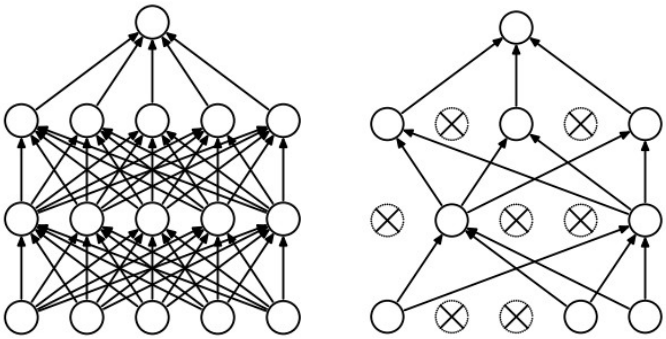
$$\begin{aligned} L &= L_0 + \lambda_2 \sum_l \sum_i \sum_j \frac{1}{2} (W_{ji}^{(l)})^2 \\ \nabla L &= \nabla L_0 + \lambda_2 \sum_l \sum_i \sum_j W_{ji}^{(l)} \end{aligned} \tag{2.26}$$

Notar que a partir de ello, durante la actualización de los pesos sinápticos en la optimización, la regularización por norma L_2 produce que cada peso decaiga linealmente a 0 (i.e. $\mathbf{W} = \mathbf{W} - \lambda_2 \mathbf{W}$) [47].

Dropout

Es un algoritmo extremadamente simple y muy efectivo para lograr la propiedad de generalización sobre redes neuronales [61]. A diferencia de las normas L_1 y L_2 no se basa en modificar la función de costo para penalizarla durante la optimización de la red, sino que consiste en modificar la red para regularizarla y hacerla más robusta a información faltante o corrupta.

Dado un patrón de entrada en el entrenamiento (ya que nunca se debe usar durante la etapa de prueba o para hacer predicciones), se permite la activación de una neurona con una cierta probabilidad p definida como parámetro, o de lo contrario se le asigna valor 0 a la salida de la misma. Esto provoca que sólo una fracción del total de neuronas produzca una activación en la salida, con lo que el proceso puede entenderse como que en cada iteración de la optimización se toma un muestreo de la red neuronal completa, y se actualizan sólo los parámetros de dicha red muestreada [4]. Durante la etapa de prueba no debe aplicarse



(a) Red neuronal estándar (b) Luego de aplicar Dropout

Figura 2.6: Figura tomada de , donde se representa la anulación de las salidas de cada neurona donde se aplicó dropout.

dropout para que la evaluación sea total en la red. A su vez, en esta etapa es importante realizar un escalado de los pesos en cada capa por p ya que se quiere que las salidas generadas sean idénticas a las salidas esperadas en la etapa de entrenamiento. Para ello, es recomendado hacerlo durante el entrenamiento de forma que el procedimiento para realizar predicciones quede inalterado [47], por lo que se deben dividir por p las activaciones producidas en cada capa luego de aplicar Dropout. En ese mismo procedimiento se debe retornar además la máscara binaria producida para saber exactamente cuáles fueron las unidades "tiradas" con el Dropout, así no se actualizan durante el proceso de optimización. En la Figura 2.6 se puede apreciar gráficamente cómo afecta el Dropout a las conexiones de una red neuronal, y el Algoritmo 3 del Apéndice A refleja el procedimiento a realizar para lograr esto durante el entrenamiento de una red.

A partir de todas estas técnicas explicadas, se consideran la siguientes recomendaciones prácticas para obtener resultados buenos en la optimización de una red neuronal:

- En la práctica, es mayormente utilizada la regularización mediante la norma L_2, aunque se suele incorporar también en menor proporción la norma L_1 para lograr también ciertas propiedades de "raleza" sobre los pesos sinápticos. Esta combinación constituye lo que se denomina *regularización de red elástica*[72].

- Como se puede notar, la regularización nunca afecta al vector de sesgo $\mathbf{b}$. Esto se debe a que el mismo no interactúa con los datos de forma multiplicativa, y como sólo produce una traslación en el espacio de soluciones no se considera que regularizar dicho vector produzca una moderación importante sobre la solución respecto al ajuste [47].

- Por lo general, por cada norma se utiliza la misma constante de penalización λ en todas las capas de la red.

- Como regla general, el Dropout se suele aplicar con $p = 0{,}5$ para todas las capas ocultas, aunque también se puede probar sobre la entrada con $p = 0{,}2$ [2] [36].

2.3.3. Aprendizaje no supervisado

En las anteriores secciones, se presentaron técnicas para modelar sistemas con aprendizaje maquinal siguiendo únicamente un enfoque supervisado. A diferencia de ello, el aprendizaje no supervisado busca modelar la función hipótesis basándose únicamente en la entrada, lo cual puede expresarse como $h(\mathbf{x}) \approx \mathbf{x}$. En el caso de las redes neuronales, se traduce en que no requieren otra información más que el vector de entrada para ajustar los pesos de las conexiones entre neuronas (i.e. se prescinde de entradas etiquetadas). En algunos casos, la salida representa el grado de similitud entre la información que se le está ingresando y la que ya se le ha mostrado anteriormente. En otro caso podría realizar una codificación de los datos de entrada, generando a la salida una versión codificada de la entrada (e.g. con menos bits, pero manteniendo la información relevante de los datos), y también algunas redes pueden lograr un mapeo de características, obteniéndose en la salida una disposición geométrica o representación topográfica de los datos de entrada [30].

Como se mencionó al principio del presente capítulo, este enfoque del aprendizaje maquinal se utiliza generalmente en dos tipos de aplicaciones: en *clustering*, y en reducción de dimensiones. Para lograr esto último, un método muy popular que se utiliza es el análisis de componentes principales (en inglés, *Principal Component Analysis* o PCA) que se basa en proyectar los datos en un espacio de dimensión menor tratando de maximizar la varianza de estos en cada una de las componentes obtenidas [9]. Dichas componentes resultan de aplicar una descomposición en valores singulares (SVD) a la matriz formada con los datos, y cada componente es un autovector que se caracteriza por la varianza que retiene de la proyección mediante su correspondiente autovalor. Por lo tanto, para lograr la reducción de dimensiones se toman las componentes que mayor varianza producen (ordenadas por autovalor) y además se puede conocer la proporción de varianza que retuvo la reducción mediante la proporción total de autovalores retenidos respecto al total de la proyección. También se utiliza para extracción de características sobre un conjunto de datos, ya que las componentes principales se pueden como los vectores que "mayor información" proveen sobre dichos datos (en términos de variabilidad) y por ende aportan mayor discriminación para otros algoritmos clasificadores o de regresión [31]. Cuando se aplica *whitening* a la salida del PCA, cada una de las componentes es escalada al dividir sus dimensiones por el respectivo autovalor, y al resultado de esto se lo suele denominar ZCA [47].

En los algoritmos de aprendizaje profundo, el enfoque no supervisado es frecuentemente considerado crucial para obtener un buen desempeño con las redes neuronales entrenadas. Esto se debe a que puede ayudar a lograr la generalización buscada en la red, ya que gran parte de la información que definen sus parámetros provienen de modelar los datos de entrada. Luego la información de las etiquetas puede ser usada para ajustar los parámetros obtenidos, los cuales ya descubrieron las características importantes de forma no supervisada.

La ventaja del pre-entrenamiento no supervisado como regularizador respecto a una inicialización aleatoria de parámetros ha sido claramente demostrada en distintas comparaciones estadísticas [22] [6] [29]. De esta forma, las redes pueden aprender a extraer características por sí solas, por lo cual sus entradas suelen ser datos crudos sin mucho pre-procesamiento, y la idea de inyectar una señal de entrenamiento no supervisado por cada capa puede ayudar a guiar a

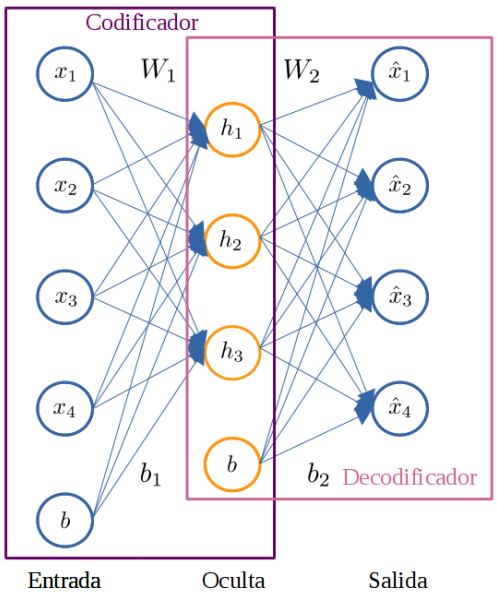


Figura 2.7: Arquitectura básica de un autocodificador.

sus respectivos parámetros hacia mejores regiones en el espacio de búsqueda [6].

2.3.4. Autocodificadores

Un autoasociador o autocodificador (en inglés, conocido como *AutoEncoder* o AE) es un tipo de red neuronal de tres capas, donde su entrada y salida tienen igual dimensión y se fuerza a que sean iguales (es decir, que la red aprenda a reconstruir la entrada en la salida). Esto constituye un esquema no supervisado ya que se trata de aproximar la función identidad (i.e. $\mathbf{y} = \mathbf{x}$), pero además se imponen ciertas restricciones en la configuración que permiten capturar una estructura de los datos que la ajustan. Estas restricciones se hacen sobre términos que penalicen la red (e.g. normas de regularización) y sobre la dimensión de la capa oculta, que por lo general se dispone que sea distinta a la de entrada.

Un autocodificador entonces consiste de dos partes: el codificador, que produce la transformación de la entrada en la dimensión dada por la capa oculta, y el decodificador que vuelve a reconstruir la entrada a partir de la representación codificada. Para lograr la reconstrucción, esta red se entrena mediante retropropagación para minimizar el error de reconstrucción (generalmente medido con MSE), por lo cual supone un sistema de regresión. En la Figura 2.7 se puede apreciar la arquitectura de un autocodificador tal como fue detallada.

Una aplicación muy estudiada de los autocodificadores consiste en la reducción de dimensiones sobre un conjunto de datos. En comparación con PCA, los autocodificadores se asemejan a dicho método cuando la dimensión de salida en la red es menor a la de entrada, pero se diferencian en que la transformación producida es no lineal, lo cual en muchos estudios produce mejores representaciones de los datos a reducir [35].

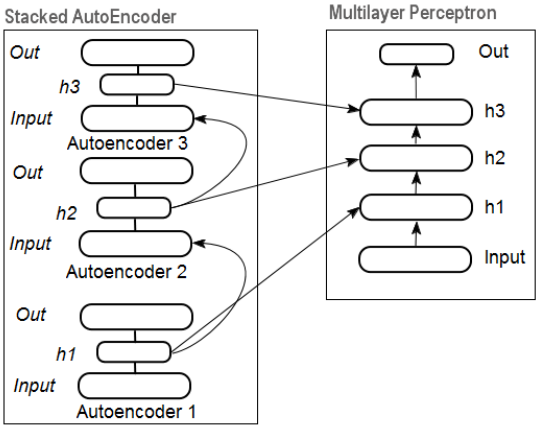


Figura 2.8: Construcción de un autocodificador apilado.

Existen ciertas formas de extender el diseño de un autocodificador para asegurar que capture una representación útil de la entrada. Una es agregar "raleza" (en inglés, *sparsity*) que significa forzar a que muchas unidades ocultas sean iguales o cercanas a cero, lo cual ha sido explotado en muchas aplicaciones exitosas [49]. Otra forma es agregar aleatoriedad en la codificación de la entrada a reconstruir, como en los *denoising autoencoders* que adicionan ruido a la entrada para que la red aprenda a anularlo o limpiarlo en la salida [65]. También existe un enfoque distinto definido por *variational autoencoders*, en el que la representación latente aprendida compone un modelo generativo con el cual se puede realizar un muestreo a partir de una entrada dada [43].

Autocodificadores apilados

Una vez que el autocodificador se entrenó de forma no supervisada, se pueden utilizar las características que aprendió (i.e. la codificación de la entrada) para realizar una tarea supervisada de regresión o clasificación. En ese caso, suele resultar conveniente ajustar la red ya entrenada utilizando patrones etiquetados de datos para mejorar el desempeño en dicha tarea. De esta forma, el entrenamiento de una red neuronal se puede componer en dos etapas:

a) Un *pre-entrenamiento* de forma no supervisada, para extraer características de los datos y obtener una representación codificada de ellos.

b) Un *ajuste fino* de forma supervisada, para modificar los parámetros de forma que mejore la tarea asignada a la red en base a ello.

Para extraer distintos niveles de representación sobre los datos, los AEs son combinados en otro tipo de red denominada "autocodificador apilado" (más conocida en inglés como *Stacked AutoEncoder* o SAE). A partir de ello es que se pueden construir redes neuronales profundas, compuestas de múltiples capas para extraer características de distintos niveles sobre los datos.

Dado un autocodificador ya entrenado, se puede apilar éste con otro para conformar un autocodificador apilado. No obstante, no se utiliza en su totalidad

sino que se aprovecha sólo la parte de codificación. Es decir que la activación producida en la capa oculta de un autocodificador (i.e. las características detectadas) alimentan la entrada del autocodificador siguiente que es agregado a la pila, como se esquematiza en la Figura 2.8. Esto quiere decir que cada AE de esta pila trata de reconstruir la salida producida por el AE precedente, y a partir de ello las representaciones de los datos adquieren distintos niveles a lo largo de esta estructura. A este proceso de entrenar un autocodificador a partir del otro se lo suele denominar en inglés como *greedy layer-wise*, y se considera crucial para pre-entrenar redes profundas de forma no supervisada asegurando que cada nivel de las mismas reciba actualizaciones adecuadas durante la optimización [6].

Una vez ejecutado el pre-entrenamiento de un SAE, se puede realizar el ajuste fino mencionado con datos etiquetados en forma supervisada. Esto equivale a inicializar los parámetros de un perceptrón multicapa en forma no supervisada, lo cual mejora importantemente el desempeño del modelo en muchas aplicaciones estudiadas respecto a la inicialización estándar [6]. Con ello se procede a ajustar el modelo para una tarea en particular, y la combinación de estas dos etapas es muy explotada en diversas aplicaciones de aprendizaje profundo para obtener modelos con buen desempeño en tareas de gran complejidad.

Bibliografía

Cuando bebas agua, recuerda la fuente.
Proverbio chino

[1] T. Abaitua et al. Procesado de señales eeg para un interfaz cerebro-máquina (bci). 2012.

[2] A. Arora, A. Candel, J. Lanford, E. LeDell, and V. Parmar. Deep learning with h2o, 2015.

[3] S. Bahrampour, N. Ramakrishnan, L. Schott, and M. Shah. Comparative study of caffe, neon, theano, and torch for deep learning. *arXiv preprint arXiv:1511.06435*, 2015.

[4] P. Baldi and P. Sadowski. The dropout learning algorithm. *Artificial intelligence*, 210:78–122, 2014.

[5] J. Bayer, C. Osendorfer, S. Diot-Girard, T. Rückstiess, and S. Urban. climin - a pythonic framework for gradient-based function optimization. *TUM Tech Report*, 2016.

[6] Y. Bengio. Learning deep architectures for ai. *Foundations and trends® in Machine Learning*, 2(1):1–127, 2009.

[7] Y. Bengio. Practical recommendations for gradient-based training of deep architectures. In *Neural networks: Tricks of the trade*, pages 437–478. Springer, 2012.

[8] J. Bergstra and Y. Bengio. Random search for hyper-parameter optimization. *The Journal of Machine Learning Research*, 13(1):281–305, 2012.

[9] C. M. Bishop. Pattern recognition. *Machine Learning*, 128, 2006.

[10] B. Blankertz, G. Curio, and K.-R. Muller. Classifying single trial eeg: Towards brain computer interfacing. *Advances in neural information processing systems*, 1:157–164, 2002.

[11] R. H. Byrd, P. Lu, J. Nocedal, and C. Zhu. A limited memory algorithm for bound constrained optimization. *SIAM Journal on Scientific Computing*, 16(5):1190–1208, 1995.

[12] M. Clifton. What is a framework? `http://www.codeproject.com/Articles/5381/What-Is-A-Framework`. Accedido: 04-01-2016.

[13] A. Cotter, O. Shamir, N. Srebro, and K. Sridharan. Better mini-batch algorithms via accelerated gradient methods. In *Advances in neural information processing systems*, pages 1647–1655, 2011.

[14] G. Coulouris, J. Dollimore, and T. Kindberg. Distributed systems: Concepts and design, 1994.

[15] M. D' Zmura, S. Deng, T. Lappas, S. Thorpe, and R. Srinivasan. Toward eeg sensing of imagined speech. In *International Conference on Human-Computer Interaction*, pages 40–48. Springer, 2009.

[16] C. S. DaSalla, H. Kambara, M. Sato, and Y. Koike. Single-trial classification of vowel speech imagery using common spatial patterns. *Neural Networks*, 22(9):1334–1339, 2009.

[17] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, A. Senior, P. Tucker, K. Yang, Q. V. Le, et al. Large scale distributed deep networks. In *Advances in Neural Information Processing Systems*, pages 1223–1231, 2012.

[18] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[19] T. Dettmers. Deep learning in a nutshell: History and training. `https://devblogs.nvidia.com/parallelforall/deep-learning-nutshell-history-training/`. Accedido: 22-02-2016.

[20] D. D. Dive. Optimization of gradient descent. `http://dsdeepdive.blogspot.com/2016/03/optimizations-of-gradient-descent.html`. Accedido: 19-10-2016.

[21] D. L. Donoho et al. High-dimensional data analysis: The curses and blessings of dimensionality. *AMS Math Challenges Lecture*, pages 1–32, 2000.

[22] D. Erhan, Y. Bengio, A. Courville, P.-A. Manzagol, P. Vincent, and S. Bengio. Why does unsupervised pre-training help deep learning? *Journal of Machine Learning Research*, 11(Feb):625–660, 2010.

[23] L. A. Farwell and E. Donchin. Talking off the top of your head: toward a mental prosthesis utilizing event-related brain potentials. *Electroencephalography and clinical Neurophysiology*, 70(6):510–523, 1988.

[24] L. A. Farwell and E. Donchin. The truth will out: Interrogative polygraphy ("lie detection") with event-related brain potentials. *Psychophysiology*, 28(5):531–547, 1991.

[25] M. Fayad and D. C. Schmidt. Object-oriented application frameworks. *Communications of the ACM*, 40(10):32–38, 1997.

[26] L. Ferrado and M. Cuenca-Acuna. Filtrando eventos de seguridad en forma conservativa mediante deep learning. pages 94–101. 45 JAIIO, 2016. ISSN: 2451–7585.

[27] P. Forslund. A neural network based brain-computer interface for classification of movement related eeg. 2003.

[28] M. Fowler. Continous integration. `http://martinfowler.com/articles/continuousIntegration.html`. Accedido: 05-10-2016.

[29] X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In *International conference on artificial intelligence and statistics*, pages 249–256, 2010.

[30] J. R. H. González and V. J. M. Hernando. *Redes neuronales artificiales: fundamentos, modelos y aplicaciones*. Ra-ma, 1995.

[31] S. Haykin and N. Network. A comprehensive foundation. *Neural Networks*, 2(2004), 2004.

[32] K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1026–1034, 2015.

[33] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016.

[34] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *Signal Processing Magazine, IEEE*, 29(6):82–97, 2012.

[35] G. E. Hinton and R. R. Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, 2006.

[36] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*, 2012.

[37] S. Hochreiter, Y. Bengio, P. Frasconi, and J. Schmidhuber. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies, 2001.

[38] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[39] U. Hoffmann, J.-M. Vesin, T. Ebrahimi, and K. Diserens. An efficient p300-based brain–computer interface for disabled subjects. *Journal of Neuroscience methods*, 167(1):115–125, 2008.

[40] H. Karau, A. Konwinski, P. Wendell, and M. Zaharia. *Learning Spark: Lightning-Fast Big Data Analysis*. O 'Reilly Media, Inc., 2015.

[41] K. Kaur and A. K. Rai. A comparative analysis: Grid, cluster and cloud computing. *International Journal of Advanced Research in Computer and Communication Engineering*, 3(3):5730–5734, 2014.

[42] S. S. Khan and M. G. Madden. A survey of recent trends in one class classification. In *Irish conference on Artificial Intelligence and Cognitive Science*, pages 188–197. Springer, 2009.

[43] D. P. Kingma and M. Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[44] R. Kohavi et al. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Ijcai*, volume 14, pages 1137–1145, 1995.

[45] Q. V. Le. Building high-level features using large scale unsupervised learning. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 8595–8598. IEEE, 2013.

[46] E. L. Lehmann and G. Casella. *Theory of point estimation*. Springer Science & Business Media, 2006.

[47] F.-F. Li and A. Karpathy. Cs231n: Convolutional neural networks for visual recognition, 2015.

[48] R. F. López and J. M. F. Fernandez. *Las redes neuronales artificiales*. Netbiblo, 2008.

[49] C. P. MarcÁurelio Ranzato, S. Chopra, and Y. LeCun. Efficient learning of sparse representations with an energy-based model. In *Proceedings of NIPS*, 2007.

[50] Y. Nesterov. A method for unconstrained convex minimization problem with the rate of convergence o (1/k2). In *Doklady an SSSR*, volume 269, pages 543–547, 1983.

[51] A. Ng, J. Ngiam, C. Y. Foo, Y. Mai, and C. Suen. Ufldl tutorial, 2012.

[52] M. A. Nielsen. Neural networks and deep learning. *URL: http://neuralnetworksanddeeplearning.com*, 2015.

[53] D. M. Powers. Evaluation: from precision, recall and f-measure to roc, informedness, markedness and correlation. 2011.

[54] G. Pressel and L. Rufiner. Diseño y elaboración de una base de datos pública de registros electroencefalográficos orientados a la clasificación de habla imaginada. *Proyecto Final de Bioingeniería, FIUNER, Oro Verde, Entre Ríos*, 2016.

[55] B. Recht, C. Re, S. Wright, and F. Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems*, pages 693–701, 2011.

[56] O. Rojo. Introducción a los sistemas distribuidos, 2003.

[57] S. Ruder. An overview of gradient descent optimization algorithms. `http://sebastianruder.com/optimizing-gradient-descent/`. Accedido: 19-10-2016.

[58] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[59] M. Snir. *MPI–the Complete Reference: The MPI core*, volume 1. MIT press, 1998.

[60] M. Sokolova and G. Lapalme. A systematic analysis of performance measures for classification tasks. *Information Processing & Management*, 45(4):427–437, 2009.

[61] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[62] P. Suppes, Z.-L. Lu, and B. Han. Brain wave recognition of words. *Proceedings of the National Academy of Sciences*, 94(26):14965–14969, 1997.

[63] Y. Taigman, M. Yang, M. Ranzato, and L. Wolf. Deepface: Closing the gap to human-level performance in face verification. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1701–1708, 2014.

[64] J. van Doorn. Analysis of deep convolutional neural network architectures. 2014.

[65] P. Vincent, H. Larochelle, Y. Bengio, and P.-A. Manzagol. Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th international conference on Machine learning*, pages 1096–1103. ACM, 2008.

[66] D. Wang and J. Huang. Tuning java garbage collection for spark applications. `https://databricks.com/blog/2015/05/28/tuning-java-garbage-collection-for-spark-applications.html`. Accedido: 18-12-2015.

[67] T. White. *Hadoop: The definitive guide*. .º 'Reilly Media, Inc.", 2012.

[68] D. R. G. H. R. Williams and G. Hinton. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986.

[69] J. R. Wolpaw and D. J. McFarland. Control of a two-dimensional movement signal by a noninvasive brain-computer interface in humans. *Proceedings of the National Academy of Sciences of the United States of America*, 101(51):17849–17854, 2004.

[70] Y. Yang and X. Liu. A re-examination of text categorization methods. In *Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval*, pages 42–49. ACM, 1999.

[71] M. D. Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.

[72] H. Zou and T. Hastie. Regularization and variable selection via the elastic net. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 67(2):301–320, 2005.