REDES NEURONALES

INTRODUCCIÓN

Antes del surgimiento de las redes neuronales, la resolución de problemas en inteligencia artificial se basaba principalmente en la representación explícita del conocimiento, utilizando reglas definidas por expertos y modelos probabilísticos. Sin embargo, con el rápido avance de la inteligencia artificial comenzaron a aparecer problemas cada vez más complejos, para los cuales este enfoque resultaba poco práctico o insuficiente. En estos casos, definir de manera explícita todas las reglas y relaciones necesarias se volvía inviable, por lo que el paradigma tradicional dejó de ser adecuado.

Como se mencionó anteriormente, la regresión logística puede considerarse la unidad mínima de una red neuronal, es decir, una regresión logística puede interpretarse como una neurona artificial básica. Este concepto surge como una forma de aproximarse al funcionamiento del cerebro humano, un órgano extremadamente complejo cuyo proceso de manejo de la información aún no se comprende por completo. Hasta hoy, los científicos no han logrado definir con certeza un modelo matemático que describa exactamente su funcionamiento, si es que tal modelo existe.

Durante la década de los años 50, científicos e ingenieros comenzaron a introducir los primeros conceptos de inteligencia artificial. No obstante, el hardware disponible en esa época no contaba con la capacidad necesaria para ejecutar algoritmos complejos como los modelos neuronales. Es importante destacar que las redes neuronales artificiales no son una simulación exacta del cerebro humano, sino una aproximación inspirada en su comportamiento.

A pesar de ser aproximaciones, estos modelos han demostrado ser lo suficientemente potentes como para permitir que las máquinas realicen tareas tradicionalmente asociadas a los seres humanos, tales como la clasificación de datos mediante el reconocimiento de patrones (por ejemplo, clasificación de animales), la detección de anomalías (como en

sistemas de videovigilancia), el procesamiento de señales (imágenes, video, audio y voz) y la aproximación de funciones objetivo.

En este tema aprenderás el principio de funcionamiento de una red neuronal artificial, una de las herramientas más utilizadas actualmente en inteligencia artificial, gracias a su gran capacidad para reconocer patrones y extraer información relevante a partir de los datos.

El sistema visual humano es una verdadera maravilla de la naturaleza, perfeccionada a lo largo de millones de años de evolución. Gracias a esta capacidad, para los seres humanos resulta muy sencillo reconocer la secuencia de números mostrada en la Figura 1, la cual pertenece al conjunto de datos MNIST.

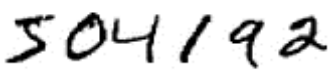

Figura 1.Secuencia de digitos tomados del conjunto de datos MNIST

Sin embargo, si se analiza con mayor detalle el proceso involucrado en el reconocimiento de los dígitos 504192, se puede apreciar que se trata de una tarea altamente compleja. Este proceso comienza en la corteza visual primaria, también conocida como V_1, la cual contiene aproximadamente 140 millones de neuronas con decenas de billones de conexiones entre ellas. Posteriormente, el procesamiento de la información visual se realiza de manera progresiva a través de otras áreas especializadas del cerebro, como las cortezas V_2, V_3, V_4 y V_5. Esto pone en evidencia que, aunque el reconocimiento de dígitos parece trivial para un humano, en realidad involucra un sistema extremadamente sofisticado.

Por ejemplo, para una persona identificar el número 8 es algo inmediato: se percibe simplemente como dos círculos, uno sobre otro. No obstante, diseñar un algoritmo que permita a una computadora reconocer este mismo dígito resulta una tarea considerablemente más difícil. Afortunadamente, existen las redes neuronales, que permiten desarrollar sistemas capaces de realizar este tipo de reconocimiento, aunque siguiendo un enfoque distinto al del cerebro humano.

La idea fundamental detrás de las redes neuronales consiste en presentar a la computadora una gran cantidad de ejemplos de dígitos escritos a mano. A esta colección se le conoce como

conjunto de entrenamiento, y es a partir de estos ejemplos que el modelo, a través de un conjunto de capas ordenadas jerárquicamente (como las cortezas visuales humanas), aprende a identificar patrones y a reconocer correctamente los dígitos, como se ilustra en la Figura 2.

Figura 2. Muestras de dígitos escritos a mano del conjunto de datos MNIST

Una red neuronal es capaz de aprender a partir de los ejemplos de entrenamiento e inferir de manera automática las reglas necesarias para reconocer dígitos escritos a mano. En general, mientras mayor sea la cantidad de ejemplos de entrenamiento, mejor será el desempeño de la red en el reconocimiento de los dígitos, ya que tendrá más información para identificar patrones relevantes.

Actualmente, las redes neuronales son una de las herramientas más utilizadas en la Inteligencia Artificial, principalmente debido a su gran capacidad de reconocimiento. No obstante, su aplicación no se limita únicamente a esta tarea, sino que también pueden emplearse en una amplia variedad de problemas, entre los que destacan:

- Clasificación de datos mediante reconocimiento de patrones
    - Las redes neuronales pueden utilizarse para determinar si un objeto específico aparece dentro de una imagen digital, identificando patrones visuales característicos.
- Detección de anomalías o novedades

    - Una red neuronal puede identificar comportamientos o características inusuales en los datos, como detectar si el conductor de un automóvil presenta señales de fatiga y corre el riesgo de quedarse dormido mientras conduce.
- Procesamiento de señales
    - Las redes neuronales permiten realizar tareas como compresión, separación, filtrado o codificación de señales, incluyendo imágenes, audio y otros tipos de datos.
- Aproximación de una función objetivo
    - Estas redes también pueden utilizarse para predecir eventos futuros, por ejemplo, estimar si una tormenta tiene probabilidades de convertirse en un huracán.

Las redes neuronales forman parte de los algoritmos de Aprendizaje Automático (Machine Learning), una rama fundamental de la inteligencia artificial. En la literatura es común encontrar comparaciones directas entre las redes neuronales artificiales y el cerebro humano, como si estas redes fueran un modelo exacto del funcionamiento de la corteza visual humana. Sin embargo, esta idea no es del todo correcta. Las redes neuronales artificiales no son una simulación precisa del cerebro humano, sino modelos matemáticos inspirados en principios generales del funcionamiento de los sistemas biológicos. Sus respuestas se asemejan a la forma en que podría responder el cerebro de cualquier mamífero, incluyendo el ser humano, pero sin replicar fielmente su estructura o complejidad.

El cerebro humano está compuesto por aproximadamente 10^{11} neuronas, las cuales reciben señales electroquímicas provenientes de otras neuronas a través de uniones sinápticas. Estas uniones conectan el axón (salida) de la neurona emisora con las dendritas (entradas) de la neurona receptora. En función de los impulsos recibidos, cada neurona realiza una serie de procesos internos y, posteriormente, genera su propia señal de salida.

La emisión de esta señal está determinada por el potencial interno de la neurona. Cuando dicho potencial supera un cierto valor umbral, se produce un impulso eléctrico que se transmite a través del axón; en caso contrario, no se genera ninguna señal. Además, en el cerebro humano existe un proceso continuo de aprendizaje, mediante el cual la efectividad de

las sinapsis se modifica con el tiempo, permitiendo que la influencia de una neurona sobre otra aumente o disminuya.

Las redes neuronales artificiales están inspiradas en estas características neurofisiológicas. Por esta razón, se componen de un gran número de unidades de procesamiento llamadas neuronas artificiales, organizadas en capas y conectadas entre sí de manera jerárquica mediante conexiones ponderadas. En este modelo artificial, el aprendizaje se lleva a cabo ajustando automáticamente los pesos de dichas conexiones, con el objetivo de reproducir un conjunto representativo de patrones asociados al problema que se desea resolver.

MODELO

El objetivo principal de las redes neuronales es encontrar los valores óptimos de sus parámetros para producir una salida deseada a partir de una entrada determinada. Para lograrlo, el funcionamiento de una red neuronal se basa, de manera general, en dos procesos fundamentales: la propagación hacia adelante y la propagación hacia atrás.

Durante el proceso de propagación hacia adelante, cada neurona calcula su respuesta a partir de las entradas que recibe. Esta operación consiste en realizar una combinación lineal de las entradas con sus respectivos pesos, agregar un término de sesgo y aplicar posteriormente una función de activación. Matemáticamente, este proceso puede expresarse como:

$$\mathrm{y} = f(\mathbf{w} \cdot \mathbf{x} + b) \tag{1}$$

donde $\mathbf{x}$ representa el vector de entradas, $\mathbf{w}$ el vector de pesos asociados, b el sesgo y $f(\cdot)$ una función de activación. Este mecanismo permite a la red transformar la información de entrada y generar una salida que será utilizada por las siguientes capas o, en el caso de la última capa, como la predicción final del modelo.

Utilizaremos x_k para referirnos a la entrada k; w_{jk}^{l} para referirnos al peso de la conexión de la neurona k en la capa l a la neurona j de la capa $l+1$; del mismo modo, para el bias de la capa l que conecta la neurona j de la capa $l+1$ se representa como b_j^l. Por ejemplo, la figura

1 muestra el peso $w_{1,2}^{1}$, el cual pondera la conexión de la neurona 2 (x_2) de la capa de entrada a la neurona 1 (a_1^1) de la capa 1.

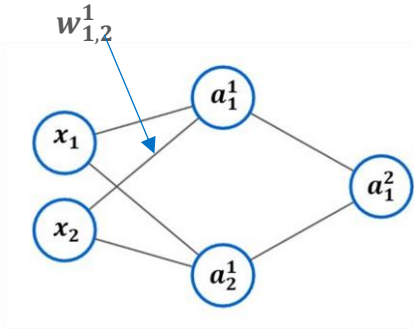

Fig. 1. Visualización de peso

De manera similar, representaremos las activaciones de la red; utilizaremos z_k^l para la respuesta lineal (antes de ser sometida a una función de activación) de la neurona k en la capa l y a_k^l para referirnos a la respuesta no lineal (después de ser sometida a una función de activación) de la neurona k en la capa l.

Por ejemplo, en la figura 1 se observan tres activaciones, dos en la capa 1 (a_1^1 y a_2^1) y una en la capa 2 (a_1^2). Así, la activación de una neurona viene dada por la siguiente ecuación:

$$a_j^l = f\left(\sum_k^K w_{j,k}^l \times a_k^{l-1} + b_j^l\right) \quad (2)$$

La función de activación permite agregar no linealidad a la suma ponderada de las entradas de cada neurona. Existen diferentes tipos de funciones de activación, sin embargo, nos enfocaremos solo en dos, en la función sigmoide y la función ReLU (Rectified Linear Unit). La función sigmoide (Figura 2) es una función monótona acotada que da una salida gradual no lineal para las entradas; se define como:

$$f(x) = \frac{1}{1+e^{-x}} \quad (3)$$

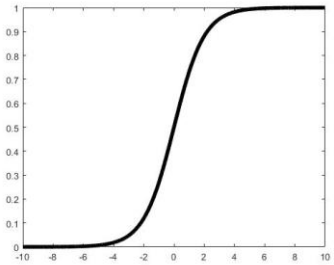

Fig. 2. Función sigmoide

Por otro lado, la función ReLU (Figura 3) está definida como:

$$f(x) = max(0, x) \tag{4}$$

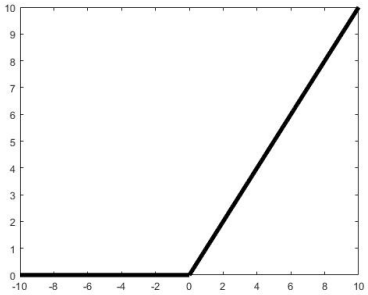

Fig 3. Función ReLU

Para visualizar de manera más clara el proceso de propagación hacia adelante, utilizaremos como ejemplo el entrenamiento de una red neuronal con función de activación sigmoide para simular el funcionamiento de una compuerta lógica XOR.

Las redes neuronales pertenecen a la categoría de algoritmos de aprendizaje supervisado, lo que implica que requieren un conjunto de entrenamiento. Dicho conjunto está formado por pares de datos (x, y), donde $x$representa las entradas y $y$las salidas deseadas.

En el caso particular de la compuerta lógica XOR, el conjunto de entrenamiento estará conformado por la tabla de verdad de dicha compuerta. Esta tabla define explícitamente la relación entre las entradas y la salida esperada, y será utilizada por la red neuronal para aprender el comportamiento lógico de la compuerta. A continuación, se define formalmente dicho conjunto de entrenamiento:

Tabla 1. Conjunto de Entrenamiento

x		y
0	0	0
0	1	1
1	0	1
1	1	0

Cada vector columna de x representa una pareja de entradas binarias, mientras que cada elemento del vector $y$corresponde a la salida deseada para esa pareja de entradas. Por ejemplo, la pareja de valores ubicada en la columna 3 de x es $[1; 0]$; dado que la operación lógica $1 \oplus 0$(XOR) produce como resultado 1, este es precisamente el valor que aparece en la posición 3 del vector y.

Supongamos ahora que, para este proceso de entrenamiento, utilizaremos la arquitectura de red neuronal que se muestra en la figura 4, la cual permitirá modelar el comportamiento no lineal de la compuerta lógica XOR.

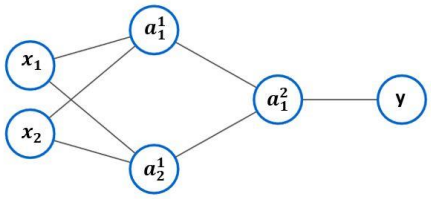

Fig. 4. Arquitectura neuronal

Ahora supongamos que los parámetros están definidos como:

$$\boldsymbol{w}^1 = \begin{bmatrix} 20 & 20 \\ -20 & 20 \end{bmatrix} \qquad \boldsymbol{w}^2 = [20 \quad 20]$$

$$\boldsymbol{b}^1 = \begin{bmatrix} -10 \\ 30 \end{bmatrix} \qquad \boldsymbol{b}^2 = [-30]$$

Para calcular la salida de la primera neurona de la primera capa oculta, utilizando un vector renglón (posición 4) de $\boldsymbol{x}$ como entrada haremos:

$$z_1^1 = w_{1,1}^1 \times x_1 + w_{1,2}^1 \times x_2 + b_1^1 \tag{5}$$

$$z_1^1 = 20 \times 1 + 20 \times 1 - 10$$

$$z_1^1 = 30$$

$$a_1^1 = \sigma(z_1^1) \tag{6}$$

$$a_1^1 = \sigma(30)$$

$$a_1^1 = 1$$

Hacemos lo mismo para la neurona 2 de la primera capa oculta:

$$z_2^1 = w_{2,1}^1 \times x_1 + w_{2,2}^1 \times x_2 + b_2^1 \tag{7}$$

$$z_2^1 = -20 \times 1 + 20 \times 1 + 30$$

$$z_2^1 = 30$$

$$a_2^1 = \sigma(z_2^1) \tag{8}$$

$$a_2^1 = \sigma(30)$$

$$a_2^1 = 1$$

Para calcular la salida de la neurona a_1^2 hacemos:

$$z_1^2 = w_{1,1}^2 \times a_1^1 + w_{1,2}^2 \times a_2^1 + b_1^2 \tag{9}$$

$$z_1^2 = 20 \times 1 + 20 \times 1 - 30$$

$$z_1^2 = 10$$

$$a_1^2 = \sigma(z_1^2) \tag{10}$$

$$a_1^2 = \sigma(10)$$

$$a_1^2 = 1$$

Realizar los cálculos arriba descritos de manera individual neurona por neurona es muy caro, hablando computacionalmente, por lo que se opta por vectorizar las variables para tener un cálculo más eficiente. A continuación, se presentan las ecuaciones vectorizadas capa por capa de la red de la figura 2.

$$\boldsymbol{z}^1 = \boldsymbol{w}^1 \times \boldsymbol{x} + \boldsymbol{b}^1 \tag{11}$$

$$\boldsymbol{a}^1 = \sigma(\boldsymbol{z}^1) \tag{12}$$

$$\boldsymbol{z}^2 = \boldsymbol{w}^2 \times \boldsymbol{a}^1 + \boldsymbol{b}^2 \tag{13}$$

$$\boldsymbol{a}^2 = \sigma(\boldsymbol{z}^2) \tag{14}$$

Por último, la salida y de la red sería la última activación de la red, que en este caso es $\boldsymbol{a}^2$

$$\boldsymbol{y} = \boldsymbol{a}^2 \tag{15}$$

En una red neuronal no se controlan directamente ni las entradas ni las salidas; sin embargo, sí es posible controlar sus parámetros de aprendizaje, ajustándolos de tal manera que la red produzca una salida coherente y consistente para una entrada dada. Estos parámetros, principalmente los pesos y sesgos, se optimizan a través de un proceso iterativo de entrenamiento.

Durante este proceso, la red modifica gradualmente sus parámetros con el objetivo de encontrar los valores óptimos que minimicen las diferencias entre la salida generada por el modelo y la salida deseada. En las etapas iniciales del entrenamiento, es común que la red produzca salidas incorrectas, lo que implica la existencia de un error.

Para cuantificar este error se utiliza una función de error o función de costo, la cual mide qué tan lejos se encuentra la predicción de la red respecto al valor esperado. De manera intuitiva, un valor alto de la función de costo indica que la red no está realizando un buen trabajo, mientras que un valor bajo sugiere que el modelo está aprendiendo correctamente y presenta un mejor desempeño. Regresando

a nuestro ejemplo, calculamos la función de costo utilizando el error cuadrático medio de nuestro modelo:

Tabla 2. Predicción de la Red

x_1	x_2	y	a_1^2
0	0	0	0
0	1	1	1
1	0	1	1
1	1	0	1

$$C(w,b) = \frac{1}{2n}\sum_x \big(\boldsymbol{a} - \boldsymbol{y}(x)\big)^2$$

$$C(w,b) = \frac{1}{2\times 4}\sum_x ([0\ 1\ 1\ 1] - [0\ 1\ 1\ 0])^2$$

$$C(w,b) = \frac{1}{2\times 4} \times [1]$$

$$C(w,b) = \frac{1}{8}$$

El error para la primera iteración es igual a $\frac{1}{8}$; una vez que se ha calculado la función de costo $C(w,b)$, la cual está en función de los pesos y el bias, se procede a propagar el error hacía atrás mediante el proceso llamada *retro propagación* o *propagación hacia atrás*, el cual estaremos detallando a continuación.

El proceso de propagación hacía atrás nos permitirá ir propagando el error desde la salida hasta las entradas de tal manera que la red tenga una idea de lo distante que esta la salida real con la salida deseada y así poder modificar los parámetros para que éstas sean lo más cercano posible. Cuando el error de la red tiende a cero, quiere decir que las salidas reales y deseadas son muy similares, por lo que se dice que la red ha convergido.

La propagación hacía atrás se trata de entender cómo cambia la función de costo con respecto a los cambios que ocurren en los parámetros (pesos y bias). Esto significa que debemos calcular las derivadas parciales $\frac{\partial C}{\partial w_{jk}^{l}}$ y $\frac{\partial C}{\partial b_{j}^{l}}$. En primer lugar, calcularemos la derivada parcial de la función de costo (error) con respecto a los pesos; como estas variables no están directamente relacionadas, utilizaremos la regla de la cadena para realizar la operación

$$\frac{\partial C}{\partial w_{jk}^{l}} = \frac{\partial C}{\partial a_{k}^{l}} \times \frac{\partial a_{k}^{l}}{\partial z_{k}^{l}} \times \frac{\partial z_{k}^{l}}{\partial w_{jk}^{l}} \tag{16}$$

Para visualizar mejor la derivada de la función de costo con respecto al error, calcularemos qué tanto afecta un cambio en el peso $w_{1,1}^{w}$ en la salida de la red:

$$\frac{\partial C}{\partial w_{1,1}^{2}} = \frac{\partial C}{\partial a_{1}^{2}} \times \frac{\partial a_{1}^{2}}{\partial z_{1}^{2}} \times \frac{\partial z_{1}^{2}}{\partial w_{11}^{2}}$$

En la figura 5 se puede apreciar el recorrido que toma el error propagándose hasta el peso $w_{1,1}^{2}$. La visualización de la neurona fue modificada un poco; dentro de la neurona se incluyeron las dos operaciones, tanto la respuesta z_{k}^{l} como la activación a_{k}^{l}

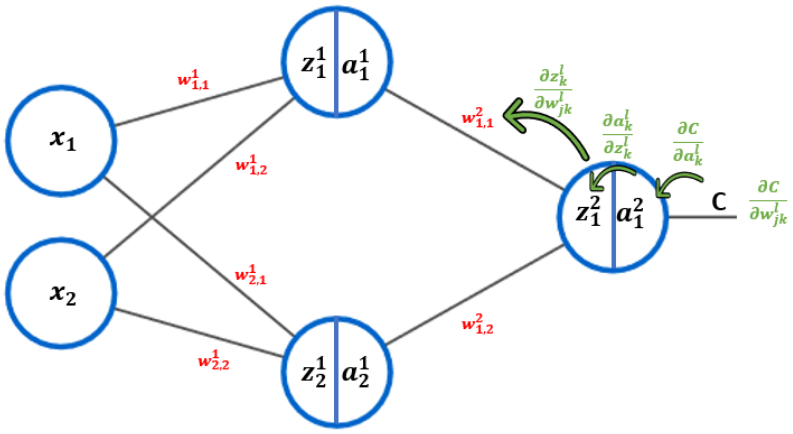

Fig. 5. Propagación hacia atrás

Ahora resolveremos cada una de las derivadas parciales, comenzando con la derivada parcial de la función de costo de la ecuación 13 con respecto a la activación:

$$\frac{\partial C}{\partial a_1^2} = \frac{\partial\left(\frac{1}{2n}\sum_x\left(\boldsymbol{a}_1^2-\boldsymbol{y}(x)\right)^2\right)}{\partial a_1^2}$$

$$\frac{\partial C}{\partial a_1^2} = \frac{1}{n}\left(\boldsymbol{a}_1^2 - \boldsymbol{y}(x)\right) \times \frac{\partial\left(\boldsymbol{a}_1^2-\boldsymbol{y}(x)\right)}{\partial a_1^2}$$

$$\frac{\partial C}{\partial a_1^2} = \frac{1}{n}\left(\boldsymbol{a}_1^2 - \boldsymbol{y}(x)\right) \tag{17}$$

Ahora procederemos con el cálculo de la derivada parcial de la activación con respecto a la respuesta:

$$\frac{\partial a_1^2}{\partial z_1^2} = \frac{\partial\left(\sigma(z_1^2)\right)}{\partial z_1^2}$$

$$\frac{\partial a_1^2}{\partial z_1^2} = \sigma'(z_1^2) = \sigma(z_1^2)\left(1 - \sigma(z_1^2)\right)$$

$$\frac{\partial a_1^2}{\partial z_1^2} = \sigma(z_1^2) \times \left(1 - \sigma(z_1^2)\right) \tag{18}$$

A continuación, procederemos a calcular la derivada de la respuesta con respecto a un peso:

$$\frac{\partial z_1^2}{\partial w_{11}^2} = \frac{\partial\left(w_{1,1}^2 \times a_1^1 + w_{1,2}^2 \times a_2^1 + b_1^2\right)}{\partial w_{11}^2}$$

$$\frac{\partial z_1^2}{\partial w_{11}^2} = a_1^1 \tag{19}$$

Sustituyendo las ecuaciones 16, 17 y 18 en 15, tenemos que una modificación en el peso $w_{1,1}^2$ afecta la función de costo como sigue

$$\frac{\partial C}{\partial w_{1,1}^2} = \frac{1}{n}\left(\boldsymbol{a}_1^2 - \boldsymbol{y}(x)\right) \times \sigma(z_1^2) \times \left(1 - \sigma(z_1^2)\right) \times a_1^1 \tag{20}$$

El mismo procedimiento se utiliza para encontrar los efectos de las modificaciones en los pesos restantes, así como también para conocer el efecto que causa una modificación en los bias.

RETRO PROPAGACIÓN Y OPTIMIZACIÓN

Como se mencionó anteriormente, el objetivo de los algoritmos de optimización es minimizar la función de costo; en este caso trataremos de minimizar la función de costo del error cuadrático medio $C(w,b)$, la cual está en función de dos variables, los pesos y los bias. La función de costo la podemos visualizar como una señal en tres dimensiones, w, b y $C(w,b)$, como se puede observar en la figura 6.

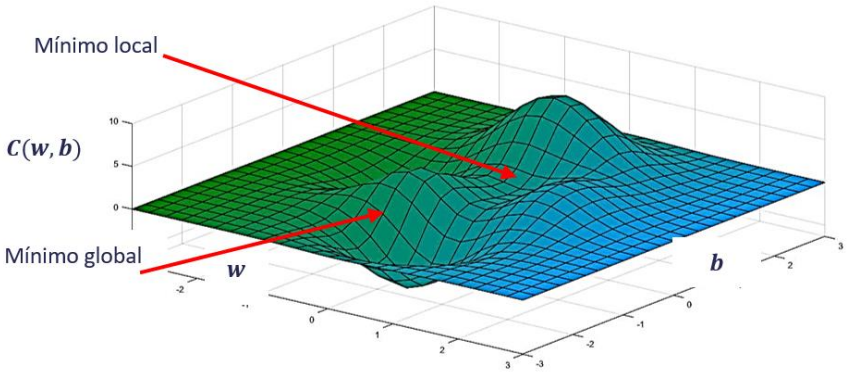

Fig. 6. Función de Costo

En un proceso iterativo, el algoritmo gradiente descendente modificará los parámetros de tal manera que se alcance un mínimo global, es decir, que el error de la red sea casi cero (convergencia de la red). Haciendo una analogía, supongamos que nosotros somos el gradiente descendente y estamos parados sobre la cima de una montaña, y buscamos caminar hasta una zona donde su altitud con respecto a nivel del mar sea la mínima, en donde podremos decir que encontramos el mínimo global.

Matemáticamente hablando, necesitamos calcular derivadas parciales con respecto a cada uno de los parámetros y utilizar una constante α conocida como factor de aprendizaje, la cual definirá que tan rápido aprende la red, o, visto de otro punto de vista, que tan grande son los pasos que debo dar para caminar hacia la zona de mínimo global (tomando en cuenta la analogía hecha anteriormente). El gradiente descendente se define como:

$$w_{nuevo} = w_{actual} - \alpha \times \nabla C(w,b) \tag{21}$$

En donde w_{nuevo} es el valor que tomará el nuevo peso para asegurar un menor error y w_{actual} es el peso que la red está utilizando actualmente. α es el factor de aprendizaje y $\nabla(.)$ es el gradiente de la función de costo.

Como resumen, la manera en cómo funciona este algoritmo es que iterativamente calcula el gradiente de la función de costo, y se mueve en la dirección opuesta, siempre hacia abajo, buscando el mínimo global (de ahí el nombre de gradiente descendente) como se muestra en la figura 7.

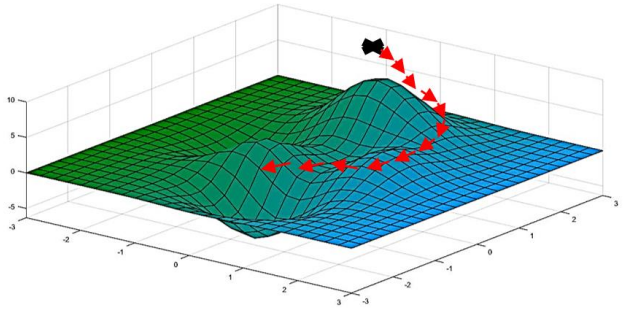

Fig. 7. Gradiente descendente

Gradiente descendente estocástico (SGD)

Este algoritmo, a diferencia del gradiente simple, actualiza los parámetros para cada ejemplo de entrenamiento. El gradiente descendente estocástico es mucho más rápido, ya que realiza las actualizaciones una por una, y no con todo el conjunto de entrenamiento:

$$w_{nuevo} = w_{actual} - \alpha \times \nabla C(w, b; x(i); y(i)) \tag{22}$$

Donde $\{x(i), y(i)\}$ corresponden a una entrada y una salida del i-ésimo ejemplo de entrenamiento, respectivamente. Una de las desventajas de hacer muchas actualizaciones (una para cada ejemplo de entrenamiento), es que se genera un fenómeno de alta varianza, lo que hace que la función de costo fluctúe en diferentes intensidades haciendo muy difícil la convergencia. Para resolver este problema, se introdujo una variante llamada gradiente descendente en mini lotes.

Gradiente descendente en mini lotes

El gradiente descendente en mini lotes corrige los inconvenientes del gradiente descendente y del SGD, sacando lo mejor de los dos algoritmos. Este algoritmo actualiza parámetros en pequeños lotes de ejemplos de entrenamiento, reduciendo la varianza en las actualizaciones.

Momento

El algoritmo de gradiente produce muchas oscilaciones, lo que hace que la convergencia sea muy difícil, para solucionar esto, el algoritmo momento fue inventado, acelerando el gradiente descendente

hacia la dirección correcta y suavizando las oscilaciones en direcciones incorrectas. El algoritmo consiste en calcular las derivadas parciales con respecto a los pesos y los bias en cada iteración y agregar un coeficiente β como se muestra en la ecuación 3 y 4

$$v_{dw}(t) = \beta \times v_{dw}(t-1) + (1-\beta) \times \nabla C(w,b) \tag{23}$$

$$v_{db}(t) = \beta \times v_{db}(t-1) + (1-\beta) \times \nabla C(w,b) \tag{24}$$

Y posteriormente se actualizan los parámetros de la siguiente manera:

$$w_{nuevo} = w_{actual} - \alpha \times v_{dw} \tag{25}$$

$$b_{nuevo} = b_{actual} - \alpha \times v_{db} \tag{26}$$

El algoritmo momento ve el problema de la convergencia desde la perspectiva de la física clásica. Suponga que tenemos una pelota en cierto terreno montañoso (curva del error) en donde el error de la red se puede interpretar como la altura sobre el nivel del mar a la cual se encuentra la pelota. Inicializar los parámetros con números aleatorios es equivalente a inicializar la velocidad inicial de la pelota a cero (existe una energía potencial $U = mgh$). Este algoritmo de optimización se puede ver como la creación de un vector de velocidad con el cual la pelota se desplaza colina abajo hasta alcanzar el punto de convergencia (altura sobre nivel del mar igual a cero).

La fuerza de la partícula está relacionada con el gradiente de la energía potencial ($F = -\nabla U$), en donde la fuerza ejercida sobre la partícula es precisamente el gradiente de la función de costo. Ya que la fuerza, según Isaac Newton es $F = ma$, el gradiente es entonces proporcional a la aceleración de la partícula. En el algoritmo SGD ocurre algo diferente; el gradiente directamente afecta la posición de la partícula, mientras que la inspiración de la física clásica sugiere que la optimización esté relacionada directamente con le velocidad lo cual hará que indirectamente se afecte la posición.

Adam

El algoritmo Adam (Adaptive Moment Estimation, por sus siglas en inglés) es otro método de optimización. Este algoritmo mantiene un decaimiento exponencial de gradientes v_{dw} y v_{db} muy similar al método momento. Mientras que momento puede ser visto como una pelota que cae sobre una pendiente, Adam se comporta como una pelota pesada con fricción. Podemos calcular los decaimientos promedios de los gradientes anteriores y anteriores cuadrados de la siguiente manera:

$$v_{dw} = \beta_1 v_{dw} + (1 - \beta_1) v_{dw} \tag{27}$$

$$v_{db} = \beta_1 v_{db} + (1 - \beta_1) v_{db} \tag{28}$$

$$s_{dw} = \beta_2 s_{dw} + (1 - \beta_2) s_{dw}^2 \tag{29}$$

$$s_{db} = \beta_2 s_{db} + (1 - \beta_2) s_{db}^2 \tag{30}$$

En este método, cuando los pesos y lo bias son inicializados en ceros, existe un sesgo hacia cero, especialmente durante las primeras iteraciones, para corregir esta deficiencia, se realiza una corrección del bias somo sigue:

$$v_{dw}^{corregida} = \frac{v_{dw}}{(1-\beta_1^t)} \tag{31}$$

$$v_{db}^{corregida} = \frac{v_{db}}{(1-\beta_1^t)} \tag{32}$$

$$s_{dw}^{corregida} = \frac{s_{dw}}{(1-\beta_2^t)} \tag{33}$$

$$s_{db}^{corregida} = \frac{s_{db}}{(1-\beta_2^t)} \tag{34}$$

Posteriormente, se actualizan los parámetros:

$$w_{nuevo} = w_{actual} - \alpha \times \frac{v_{dw}^{crregida}}{\sqrt{s_{dw}^{corregida}} + \varepsilon} \tag{35}$$

$$b_{nuevo} = b_{actual} - \alpha \times \frac{v_{db}^{crregida}}{\sqrt{s_{db}^{corregida}} + \varepsilon} \tag{36}$$

Para escoger los parámetros, α necesita ser obtenida heurísticamente, $\beta_1 = 0.9$, $\beta_2 = 0.999$ y $\varepsilon = 10^{-8}$

REGULARIZACIÓN

Suponga que estamos entrenando una red neuronal para el reconocimiento de dígitos escritos a mano, esta red contiene 30 capas ocultas, las cuales equivalen a cerca de 23,860 parámetros. De las 50,000 imágenes que contiene la base de datos MNIST, utilizaremos solamente 1,000 para el entrenamiento.

Si realizamos el entrenamiento con una red que contiene muchos parámetros, utilizando solamente un pequeño número de imágenes, la red tiene un buen desempeño con el set de entrenamiento, pero un pobre desempeño con el resto de las imágenes, este fenómeno se conoce como sobreajuste

En otras palabras, la red es capaz de aprender todas las diferencias que existen dentro del set de entrenamiento (se dice que la red tiene mucha varianza), de tal manera que memoriza las imágenes que fueron utilizadas para entrenar y su poder de generalización es muy pobre, es por ello que, para nuevos ejemplos de entrenamiento, su desempeño es muy pobre.

Una de las soluciones al sobreajuste es aumentar el set de entrenamiento para que la red neuronal pueda generalizar la información, sin embargo, esto no es nada plausible con el proceso biológico, ya que, si se le muestra una imagen de un elefante a un niño de tres años por vez primera, éste es capaz de generalizar la información y poder reconocer elefantes en imágenes que nunca ha visto. El objetivo de la regularización es darle el poder a la red de poder generalizar la información, y esto se logra utilizando los siguientes algoritmos de regularización.

Regularización L_1

Esta forma de regularización es relativamente común, la cual consiste en por cada w agregamos el término $\lambda|w|$ en la función de costo que se requiera minimizar, donde λ es el parámetro de regularización.

$$C(w,b) = \frac{1}{2n}\sum_x (y(x) - a)^2 + \frac{\lambda}{n}|w| \tag{37}$$

Este tipo de regularización tiene la propiedad de hacer que el vector de pesos tienda a cero, en otras palabras, las neuronas que utilizan la regularización L_1, terminan utilizando solo un subconjunto escaso de sus entradas más importantes y se vuelven casi invariantes a las entradas con ruido. Es posible combinar la regularización L_1 y L_2, agregando los términos $\lambda_1|w| + \frac{1}{2}\lambda_2 w^2$, donde λ_1 y λ_2 son los parámetros de regularización L_1 y L_2 respectivamente. A esta combinación se le conoce como regularización de red elástica

Regularización L_2

La regularización L_2 es la forma más común de regularización, puede ser implementada penalizando la magnitud cuadrada de todos los parámetros en la función de costo. Para cada parámetro w en la red, agregamos el término $\frac{1}{2}\lambda w^2$, donde λ es el parámetro de regularización:

$$C(w,b) = \frac{1}{2n}\sum_x (y(x) - a)^2 + \frac{1}{2n}\lambda^2 w^2 \quad (38)$$

Por la naturaleza de las redes neuronales, existe una interacción multiplicativa entre neuronas, lo que hace que los pesos cada vez sean más grandes en magnitud, como consecuencia, el producto de las entradas por los pesos tiene una magnitud enorme. La regularización L_2 penaliza los pesos muy grandes, prefiriendo vectores con pesos pequeños, y así, la red podrá utilizar sus entradas con una magnitud muy parecida a la magnitud original de ellas

Descarto de neuronas (dropout)

Este algoritmo fue propuesto por Sirvastava para prevenir el sobreajuste de las redes neuronales. El funcionamiento de este algoritmo es, mientras se está entrenando, la red solamente va a utilizar ciertas neuronas activas utilizando una probabilidad p, es decir, solamente utilizará un porcentaje de las neuronas y las demás se descartarán

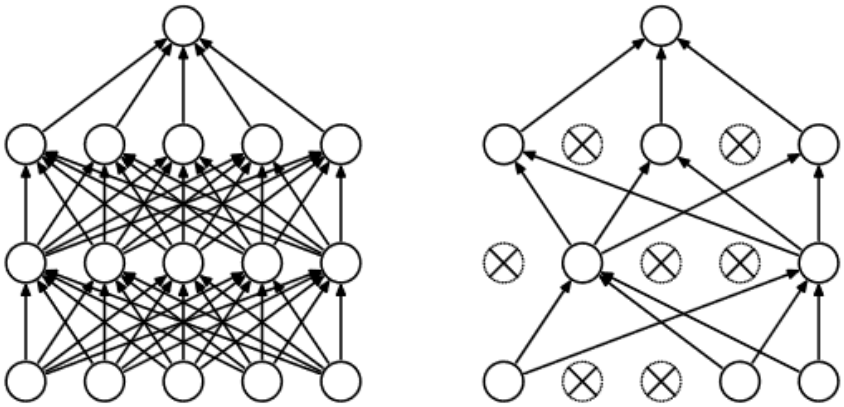

Fig. 8. Dropout

En la figura 8 se puede observar cómo se van descartando neuronas en cada capa en cada iteración, en donde las neuronas que se descartan son diferentes, por lo que se puede intuir que en cada iteración se tiene una red diferente. Para hacer una analogía, suponga que viajamos a un lugar donde no conocen los gatos, y queremos enseñarles cómo son, entonces, empezamos a repartir imágenes de gatos a personas diferentes, de esta manera, podremos crear una generalización de cómo se ve un gato utilizando diferentes puntos de vista; lo mismo pasa con las redes neuronales, cada una de ellas va a entregar una opinión diferente y como resultado, se obtiene una regularización de los datos.

En la práctica, es muy común utilizar un dropout de 0.2, es decir, descartar el 20% de las neuronas y quedarse con el 80% restante

Aumento Artificial de Datos

Las redes neuronales tienden a sobre ajustarse con la presencia de un número pequeño de ejemplos de entrenamiento, esto es porque no son suficientes para poder generalizar toda la base de datos. El método de aumento artificial de datos se utiliza cuando ya no se cuenta con más ejemplos de entrenamiento en la base de datos y se procede a crear variaciones de los ejemplos ya existentes para crear de manera artificial más datos de entrenamiento