Curso de Inteligencia Artificial

Fundamentos de aprendizaje profundo

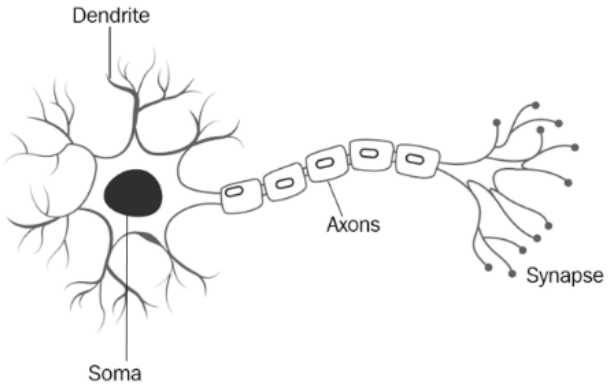
El aprendizaje profundo es un subconjunto del aprendizaje automático y se trata de redes neuronales. Este existe desde hace una década, pero la razón por la que es tan popular en este momento es por los avances informáticos y la disponibilidad de grandes volúmenes de datos. Con este enorme volumen de datos, los algoritmos de aprendizaje profundo pueden superar a los algoritmos clásicos de aprendizaje automático.

Neuronas biológicas y artificiales

Antes de continuar, primero exploraremos qué son las neuronas y cómo funcionan realmente las neuronas en nuestro cerebro, y luego aprenderemos sobre las neuronas artificiales. Una neurona se puede definir como la unidad computacional básica del cerebro humano. Las neuronas son las unidades fundamentales de nuestro cerebro y sistema nervioso. Nuestro cerebro abarca aproximadamente 100 mil millones de neuronas. Todas y cada una de las neuronas están conectadas entre sí a través de una estructura llamada sinapsis, que es responsable de recibir información del entorno externo a través de los órganos sensoriales, de enviar instrucciones motoras a nuestros músculos y de realizar otras actividades.

Una neurona también puede recibir información de otras neuronas a través de una estructura ramificada llamada dendrita. Estos insumos se fortalecen o debilitan; es decir, se ponderan según su importancia y luego se suman en el cuerpo celular llamado soma. Desde el cuerpo celular, estas entradas sumadas se procesan y se mueven a través de los axones y se envían a las otras neuronas.

La figura muestra una única neurona biológica básica:



Ahora, veamos cómo funcionan las neuronas artificiales. Supongamos que tenemos tres entradas x_1, x_2 y x_3 para predecir la salida y. Estas entradas se multiplican por los pesos w_1, w_2 y w_3 y se suman de la siguiente manera:

$$x_1.w_1 + x_2.w_2 + x_3.w_3$$

Pero, ¿por qué estamos multiplicando estos insumos por pesos? Porque todas las entradas no son igualmente importantes para calcular la salida y. Digamos que x_2 es más importante para calcular la salida en comparación con las otras dos entradas. Luego, asignamos un valor más alto a w_2 que a los otros dos pesos. Entonces, al multiplicar pesos con entradas, x_2 tendrá un valor mayor que las otras dos

entradas. En términos simples, los pesos se utilizan para fortalecer las entradas. Después de multiplicar las entradas con los pesos, las sumamos y sumamos un valor llamado sesgo, b:

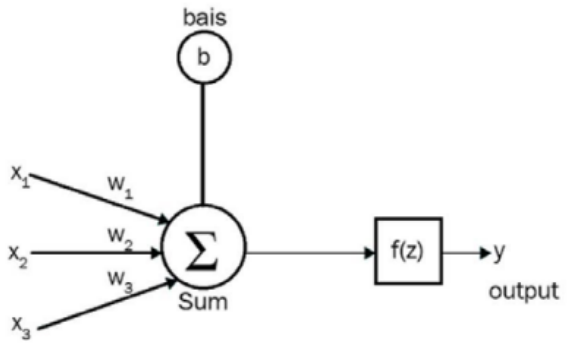
$$z = (x_1.w_1 + x_2.w_2 + x_3.w_3) + b$$

Si observas detenidamente la ecuación anterior, puede parecerte familiar. ¿No se parece z a la ecuación de regresión lineal? ¿No es simplemente la ecuación de una línea recta?

Bueno, sí. Entonces, ¿cuál es la diferencia entre las neuronas y la regresión lineal?
En las neuronas, introducimos la no linealidad en el resultado, z, aplicando una función f(.) llamada función de activación o transferencia. Por lo tanto, la salida se convierte en:

y = f(z)

La figura muestra una sola neurona artificial:



Entonces, una neurona toma la entrada, x, la multiplica por pesos, w, y agrega sesgo, b, forma z, y luego aplicamos la función de activación en z y obtenemos la salida, y.

ANN y sus capas

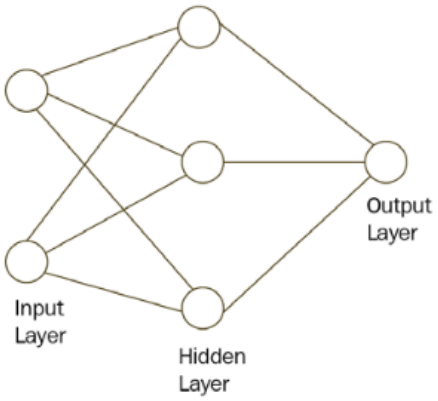
Si bien las neuronas son realmente geniales, no podemos usar una sola neurona para realizar tareas complejas. Esta es la razón por la que el cerebro tiene miles de millones de neuronas, apiladas en capas, formando una red. De manera similar, las neuronas artificiales están dispuestas en capas. Todas y cada una de las capas están conectadas de tal manera que la información pase de una capa a otra.

Una ANN típica consta de las siguientes capas:

- Capa de entrada
- Capa oculta
- Capa de salida

Cada capa tiene una colección de neuronas, y las neuronas de una capa interactúan con todas las neuronas de las otras capas. Sin embargo, las neuronas en la misma capa no interactuarán entre sí. Esto se debe simplemente a que las neuronas de las capas adyacentes tienen conexiones o bordes entre ellas, sin embargo, las neuronas en la misma capa no tienen ninguna conexión. Usamos el término nodos o unidades para representar las neuronas en la ANN.

La figura muestra una ANN típica:



Capa de entrada

La capa de entrada es donde alimentamos la entrada a la red. El número de neuronas en la capa de entrada es el número de entradas que alimentamos a la red. Cada entrada tendrá alguna influencia en la predicción de la salida. Sin embargo, no se realiza ningún cálculo en la capa de entrada; solo se usa para pasar información del mundo exterior a la red.

Capa oculta

Cualquier capa entre la capa de entrada y la capa de salida se denomina capa oculta. Procesa la entrada recibida de la capa de entrada. La capa oculta es responsable de derivar relaciones complejas entre la entrada y la salida. Es decir, la capa oculta identifica el patrón en el conjunto de datos. Es principalmente responsable de aprender la representación de datos y de extraer las características.

Puede haber cualquier número de capas ocultas, sin embargo, tenemos que elegir una cantidad de capas ocultas según nuestro caso de uso. Para un problema muy simple, podemos usar solo una capa oculta, pero mientras realizamos tareas complejas como el reconocimiento de imágenes, usamos muchas capas ocultas, donde cada capa es responsable de extraer características importantes. La red se llama red neuronal profunda cuando tenemos muchas capas ocultas.

Capa de salida

Después de procesar la entrada, la capa oculta envía su resultado a la capa de salida. Como sugiere el nombre, la capa de salida emite la salida. La cantidad de neuronas en la capa de salida se basa en el tipo de problema que queremos que resuelva la red. Si es una clasificación binaria, entonces el número de neuronas en la capa de salida es uno y nos dice a qué clase pertenece la entrada. Si se trata de una clasificación multiclase, por ejemplo, con cinco clases, y si queremos obtener la probabilidad de cada clase como salida, entonces el número de neuronas en la capa de salida es cinco, cada una de las cuales emite la probabilidad. Si es un problema de regresión, entonces tenemos una neurona en la capa de salida.

Explorando las funciones de activación

Una función de activación, también conocida como función de transferencia, juega un papel vital en las redes neuronales. Se utiliza para introducir la no linealidad en las redes neuronales. Como aprendimos

antes, aplicamos la función de activación a la entrada, que se multiplica por pesos y se suma al sesgo, es decir, f(z), donde z = (entrada * pesos) + sesgo y f(.) es la función de activación.

Si no aplicamos la función de activación, entonces una neurona simplemente se parece a la regresión lineal. El objetivo de la función de activación es introducir una transformación no lineal para aprender los complejos patrones subyacentes en los datos.

La función sigmoidea

La función sigmoidea es una de las funciones de activación más utilizadas. Escala el valor entre 0 y 1. La función sigmoidea se puede definir de la siguiente manera:

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

Es diferenciable, lo que significa que podemos encontrar la pendiente de la curva en dos puntos cualesquiera. Es monótono, lo que implica que es completamente no creciente o no decreciente. La función sigmoidea también se conoce como función logística. Como sabemos que la probabilidad se encuentra entre 0 y 1, y dado que la función sigmoidea aplasta el valor entre 0 y 1, se usa para predecir la probabilidad de salida.

La función tanh

Una función de tangente hiperbólica (tanh) genera el valor entre -1 y +1 y se expresa de la siguiente manera:

$$f(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}}$$

A diferencia de la función sigmoidea, que está centrada en 0,5, la función tanh está centrada en 0.

La función Unidad Lineal Rectificada (ReLU)

RELU, es otra de las funciones de activación más utilizadas. Emite un valor de cero a infinito. Es básicamente una función por partes y se puede expresar de la siguiente manera:

f(x) = max(0, x)

Es decir, f(x) devuelve cero cuando el valor de x es menor que cero y f(x) devuelve x cuando el valor de x es mayor o igual a cero. Cuando alimentamos cualquier entrada negativa a la función ReLU, convierte la entrada negativa a cero.

La función softmax

La función softmax es básicamente la generalización de la función sigmoidea. Por lo general, se aplica a la capa final de la red y al realizar tareas de clasificación de múltiples clases. Da las probabilidades de que cada clase sea de salida y, por lo tanto, la suma de los valores softmax siempre será igual a 1.

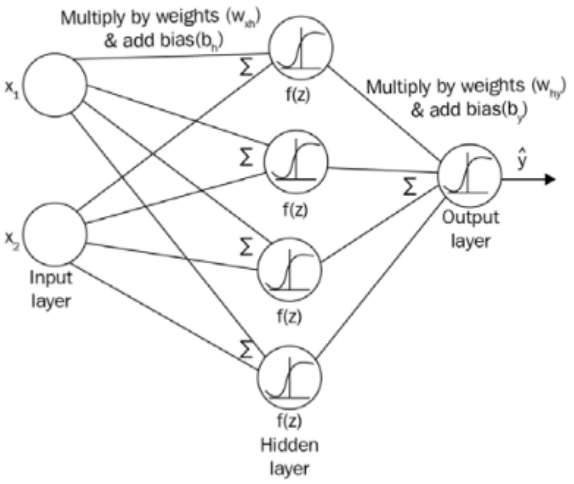
Se puede representar de la siguiente manera:

$$f(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

Como se muestra en la figura, la función softmax convierte sus entradas en probabilidades:

Propagación directa en ANN

En esta sección, veremos cómo una ANN aprende dónde se apilan las neuronas en capas. El número de capas en una red es igual al número de capas ocultas más el número de capas de salida. No tenemos en cuenta la capa de entrada cuando calculamos el número de capas en una red. Considera una red neuronal de dos capas con una capa de entrada, x, una capa oculta, h, y una capa de salida, y, como se muestra en el siguiente diagrama:



Consideremos que tenemos dos entradas, x_1 y x_2, y tenemos que predecir la salida $\hat{y}$.

Como tenemos dos entradas, el número de neuronas en la capa de entrada es dos. Establecemos el número de neuronas en la capa oculta en cuatro y el número de neuronas en la capa de salida en uno. Ahora, las entradas se multiplican por pesos, y luego agregamos sesgo y propagamos el valor resultante a la capa oculta donde se aplica la función de activación.

Antes de eso, necesitamos inicializar la matriz de peso. En el mundo real, no sabemos qué entrada es más importante que la otra para poder ponderarlos y calcular la salida. Por lo tanto, inicializamos aleatoriamente los valores de ponderación y sesgo. El peso y el valor de sesgo entre la entrada a la capa oculta están representados por W_{xh} y b_h, respectivamente. ¿Qué pasa con las dimensiones de la matriz de peso? Las dimensiones de la matriz de pesos deben ser el número de neuronas en la capa actual x el número de neuronas en la siguiente capa. ¿Porqué es eso?

Porque es una regla básica de multiplicación de matrices. Para multiplicar dos matrices, AB, el número de columnas en la matriz A debe ser igual al número de filas en la matriz B. Entonces, la dimensión de la matriz de peso, W_{xh}, debe ser el número de neuronas en la capa de entrada x el número de neuronas en la capa oculta, es decir, 2 x 4:

$$z_1 = XW_{xh} + b_h$$

La ecuación anterior representa z_1 = entrada x pesos + bias. Ahora, esto se pasa a la capa oculta. En la capa oculta, aplicamos una función de activación a z_1. Usemos la función de activación sigmoide y podemos escribir:

$$a_1 = \sigma(z_1)$$

Después de aplicar la función de activación, nuevamente multiplicamos el resultado a_1 por una nueva matriz de peso y agregamos un nuevo valor de sesgo que fluye entre la capa oculta y la capa de salida. Podemos denotar esta matriz de peso y sesgo como W_{hy} y b_y respectivamente. La dimensión de la matriz de peso W_{hy}, será el número de neuronas en la capa oculta x el número de neuronas en la capa de salida. Dado que tenemos cuatro neuronas en la capa oculta y una neurona en la capa de salida, la dimensión de la matriz W_{hy} será 4 x 1. Entonces, multiplicamos a_1 por la matriz de peso, W_{hy} y agregamos sesgo b_y y pasamos el resultado z_2 a la siguiente capa, que es la capa de salida:

$$z_2 = a_1 W_{hy} + b_y$$

Ahora, en la capa de salida, aplicamos la función sigmoide a z_2, lo que dará como resultado un valor de salida:

$$\hat{y} = \sigma(z_2)$$

Todo este proceso desde la capa de entrada hasta la capa de salida se conoce como **propagación directa o propagación hacía adelante**. Por lo tanto, para predecir el valor de salida, las entradas se propagan desde la capa de entrada a la capa de salida. Durante esta propagación, se multiplican por sus respectivos pesos en cada capa y se les aplica una función de activación encima. Los pasos completos de propagación hacia adelante se dan de la siguiente manera:

$$z_1 = XW_{xh} + b_h$$
$$a_1 = \sigma(z_1)$$
$$z_2 = a_1 W_{hy} + b_y$$
$$\hat{y} = \sigma(z_2)$$

¿Cómo sabemos si la salida generada por la red neuronal es correcta? Definimos una nueva función llamada función de costo (J), también conocida como función de pérdida (L), que nos dice qué tan bien está funcionando la red neuronal. Hay muchas funciones de costo diferentes. Usaremos el error cuadrático medio como una función de costo, que se puede definir como la media de la diferencia cuadrática entre la salida actual y la prevista:

$$J = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

Aquí, n es el número de muestras de entrenamiento, y es el resultado real y

$\hat{y}$

es el resultado previsto.

Aprendimos que se usa una función de costo para evaluar nuestra red neuronal, es decir, nos dice qué tan buena es la red neuronal para predecir la salida. Pero la pregunta es ¿dónde está aprendiendo realmente la red? En la propagación directa, la red solo intenta predecir la salida. Pero, ¿cómo aprende a predecir la salida correcta?

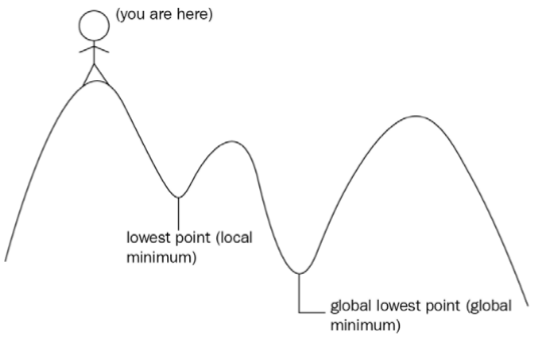
¿Cómo aprende una ANN?

Si el costo o la pérdida es muy alto, significa que la red no está prediciendo la salida correcta. Entonces, el objetivo es minimizar la función de costo para que las predicciones de redes neuronales sean mejores. ¿Cómo podemos minimizar la función de costo? Es decir, ¿cómo podemos minimizar la pérdida/costo? Aprendimos que la red neuronal hace predicciones utilizando la propagación directa. Entonces, si podemos cambiar algunos valores en la propagación directa, podemos predecir la salida correcta y minimizar la pérdida. Pero, ¿qué valores podemos cambiar en la propagación directa? Obviamente, no podemos cambiar la entrada y la salida.
Ahora nos quedan los pesos y los valores de sesgo. Recuerda que acabamos de inicializar las matrices de peso al azar. Dado que los pesos son aleatorios, no serán perfectos.
Actualizaremos estas matrices de peso (W_{xh} y W_{hy}) de tal manera que la red neuronal proporcione una salida correcta. ¿Cómo actualizamos estas matrices de peso? Aquí viene una nueva técnica llamada **descenso de gradiente**.

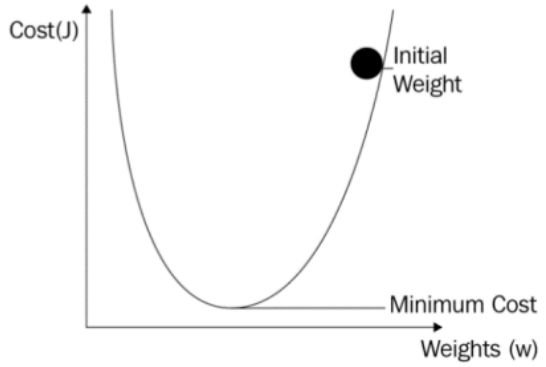
Con el descenso de gradiente, la red neuronal aprende los valores óptimos de las matrices de peso inicializadas aleatoriamente. Con los valores óptimos de pesos, la red puede predecir la salida correcta y minimizar la pérdida. Ahora, exploraremos cómo se aprenden los valores óptimos de los pesos mediante el descenso de gradiente.

El descenso de gradiente es uno de los algoritmos de optimización más utilizados. Se utiliza para minimizar la función de coste, lo que nos permite minimizar el error y obtener el menor valor de error posible. Pero, ¿cómo encuentra el descenso de gradiente los pesos óptimos?

Comencemos con una analogía. Imagina que estamos en la cima de una colina, como se muestra en el siguiente diagrama, y queremos llegar al punto más bajo de la colina. Puede haber muchas regiones que parezcan los puntos más bajos de la colina, pero tenemos que llegar al punto que en realidad es el más bajo de todos. Es decir, no deberíamos quedarnos atrapados en un punto creyendo que es el punto más bajo cuando existe el punto más bajo global:



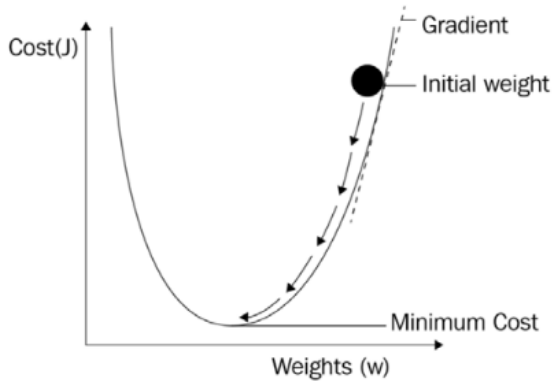
De manera similar, podemos representar la función de costo de la siguiente manera. Es una gráfica de costo contra pesos. El objetivo es minimizar la función de coste. Es decir, tenemos que llegar al punto más bajo donde el costo sea el mínimo. El punto oscuro sólido en el siguiente diagrama muestra los pesos inicializados aleatoriamente. Si movemos este punto hacia abajo, entonces podemos llegar al punto donde el costo es el mínimo:



¿Cómo podemos mover este punto (peso inicial) hacia abajo? ¿Cómo podemos descender y llegar al punto más bajo? Los gradientes se utilizan para moverse de un punto a otro. Entonces, podemos mover este punto (peso inicial) calculando un gradiente de la función de costo con respecto a ese punto (pesos iniciales), que es:

$$\frac{\partial J}{\partial W}$$

Los gradientes son las derivadas que en realidad son la pendiente de una recta tangente, como se ilustra en el siguiente diagrama. Entonces, al calcular el gradiente, descendemos (movemos hacia abajo) y alcanzamos el punto más bajo donde el costo es mínimo. El descenso de gradiente es un algoritmo de optimización de primer orden, lo que significa que solo tenemos en cuenta la primera derivada al realizar las actualizaciones:



Así, con descenso de gradiente, movemos los pesos a una posición donde el costo es mínimo. Pero aún así, ¿cómo actualizamos los pesos?

Como resultado de la propagación directa, estamos en la capa de salida. Ahora **propagaremos hacia atrás** [1]la red desde la capa de salida a la capa de entrada y calcularemos el gradiente de la función de

[1] backpropagate

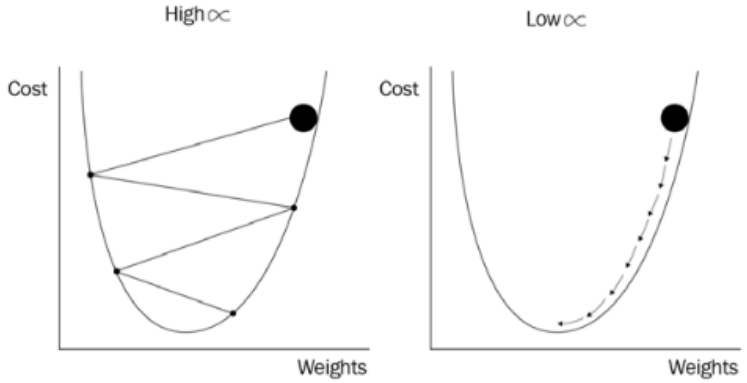
costo con respecto a todos los pesos entre la capa de salida y la de entrada para que podamos minimizar el error. Después de calcular los gradientes, actualizamos nuestros pesos anteriores usando la regla de actualización de peso:

$$W = W - \alpha \frac{\partial J}{\partial W}$$

Esto implica pesos = pesos - α x gradientes.

¿Qué es α ? Se llama **tasa de aprendizaje**. Como se muestra en el siguiente diagrama, si la tasa de aprendizaje es pequeña, entonces damos un pequeño paso hacia abajo y el descenso de gradiente puede ser lento.

Si la tasa de aprendizaje es grande, entonces damos un gran paso y el gradiente de descenso será rápido, pero es posible que no alcancemos el mínimo global y nos quedemos estancados en un mínimo local. Entonces, la tasa de aprendizaje debe elegirse de manera óptima:



Todo este proceso de propagación hacía atrás de la red desde la capa de salida a la capa de entrada y la actualización de los pesos de la red mediante el descenso de gradiente para minimizar la pérdida se denomina **retropropagación** [2]. Ahora que tenemos una comprensión básica de la retropropagación, fortaleceremos la comprensión aprendiendo sobre esto en detalle, paso a paso.

Tenemos dos pesos, uno W_{xh}, que es la entrada para los pesos de las capas ocultas, y el otro W_{hy}, que es el peso de las capas ocultas para la salida. Necesitamos encontrar los valores óptimos para estos dos pesos que nos darán la menor cantidad de errores. Entonces, necesitamos calcular la derivada de la función de costo J con respecto a estos pesos. Dado que estamos propagando hacia atrás, es decir, pasando de la capa de salida a la capa de entrada, el primer peso será W_{hy}. Entonces, ahora necesitamos calcular la derivada de J con respecto a W_{hy} ¿Cómo calculamos la derivada? Primero, recordemos la función de costo, J:

$$J = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

[2] backpropagation

No podemos calcular la derivada directamente de la ecuación anterior ya que no hay un término W_{hy}. Entonces, en lugar de calcular la derivada directamente, calculamos la derivada parcial. Recordemos la ecuación de propagación directa:

$$\hat{y} = \sigma(z_2)$$

$$z_2 = a_1 W_{hy} + b_y$$

Primero calcularemos una derivada parcial con respecto a

$$\hat{y}$$

y a partir de ahí calcularemos la derivada parcial con respecto a z_2. A partir de z_2, podemos calcular directamente la derivada W_{hy}. Es básicamente la regla de la cadena. Entonces, la derivada de J con respecto a W_{hy} queda de la siguiente manera:

$$\frac{\partial J}{\partial W_{hy}} = \frac{\partial J}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z_2} \cdot \frac{dz_2}{dW_{hy}} \qquad (1)$$

Ahora, calcularemos cada uno de los términos de la ecuación anterior:

$$\frac{\partial J}{\partial \hat{y}} = (y - \hat{y})$$

$$\frac{\partial \hat{y}}{\partial z_2} = \sigma'(z_2)$$

Aquí, está la derivada de la función de activación sigmoidea. La derivada de la función sigmoidea es:

$$\sigma'(z) = \frac{e^z}{(1 + e^{-z})^2}.$$

A continuación tenemos:

$$\frac{dz_2}{dW_{hy}} = a_1$$

Así, sustituyendo todos los términos anteriores en la ecuación (1) podemos escribir:

$$\frac{\partial J}{dW_{hy}} = (y - \hat{y}).\sigma'(z_2).a_1 \qquad (2)$$

Ahora necesitamos calcular una derivada de J con respecto a nuestro próximo peso, W_{xh}.

De manera similar, no podemos calcular la derivada de W_{xh} directamente de J ya que no tenemos ningún término de W_{xh} en J. Entonces, necesitamos usar la regla de la cadena. Recordemos los pasos de propagación hacia adelante nuevamente:

$$\hat{y} = \sigma(z_2)$$

$$z_2 = a_1 W_{hy} + b_y$$

$$a_1 = \sigma(z_1)$$

$$z_1 = X W_{xh} + b_h$$

Ahora, de acuerdo con la regla de la cadena, la derivada de J con respecto a W_{xh} se da como:

$$\frac{\partial J}{\partial W_{xh}} = \frac{\partial J}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z_2} \cdot \frac{dz_2}{\partial a_1} \cdot \frac{\partial a_1}{\partial z_1} \cdot \frac{dz_1}{dW_{xh}} \qquad (3)$$

Ya hemos visto cómo calcular los dos primeros términos de la ecuación anterior, ahora, veremos cómo calcular el resto de los términos:

$$\frac{dz_2}{\partial a_1} = W_{hy}$$

$$\frac{\partial a_1}{\partial z_1} = \sigma'(z_1)$$

$$\frac{dz_1}{dW_{xh}} = X$$

Así, sustituyendo todos los términos anteriores en la ecuación (3), podemos escribir:

$$\frac{\partial J}{\partial W_{xh}} = (y - \hat{y}).\sigma'(z_2).W_{hy}\,.\sigma'(z_1)\,.x \qquad (4)$$

Una vez que hayamos calculado los gradientes para ambos pesos, W_{hy} y W_{xh}, actualizaremos los pesos iniciales de acuerdo con la regla de actualización de peso:

$$W_{hy} = W_{hy} - \alpha \frac{\partial J}{\partial W_{hy}} \qquad (5)$$

$$W_{xy} = W_{xy} - \alpha \frac{\partial J}{\partial W_{xh}} \qquad (6)$$

Así actualizamos los pesos de una red y minimizamos la pérdida.

Aparte de esto, existen diferentes variantes de los métodos de descenso de gradiente, como el descenso de gradiente estocástico, el descenso de gradiente de mini lotes, Adam, RMSprop y más.

Antes de continuar, debes familiarizarte con algunas de las terminologías más utilizadas en las redes neuronales:

- Paso hacia adelante [3]: el paso hacia adelante implica la propagación hacia adelante desde la capa de entrada a la capa de salida.
- Paso hacia atrás [4]: el paso hacia atrás implica propagar hacia atrás desde la capa de salida a la capa de entrada.
- Época: la época especifica la cantidad de veces que la red neuronal ve todos los datos de entrenamiento. Entonces, podemos decir que una época es igual a un paso hacia adelante y un pase hacia atrás para todas las muestras de entrenamiento.
- Tamaño del lote: el tamaño del lote especifica la cantidad de muestras de entrenamiento que usamos en un paso hacia adelante y un paso hacia atrás.
- Número de iteraciones: el número de iteraciones implica el número de pasos donde un pase = un pase hacia adelante + un pase hacia atrás.

Digamos que tenemos 12.000 muestras de entrenamiento y el tamaño de lote es de 6.000. Entonces nos llevará dos iteraciones completar una época. Es decir, en la primera iteración, pasamos las primeras 6000 muestras y realizamos un pase hacia adelante y otro hacia atrás, en la segunda iteración, pasamos las siguientes 6000 muestras y realizamos un paso hacia adelante y otro hacia atrás. Después de dos iteraciones, la red neuronal verá las 12.000 muestras de entrenamiento completas, lo que lo convierte en una época.

[3] Forward pass
[4] Backward pass