Introducción a las redes neuronales artificiales

Emilio Barragán Rodríguez - 2020

1. Introducción

Las redes neuronales artificiales a día de hoy puede ser lo que más escuchemos en el mundo de la inteligencia artificial pero, ¿son una novedad? La respuesta es no. Llevan mucho tiempo existiendo, pero ahora es cuando tenemos el poder computacional para poder construirlas y ejecutarlas.

Las redes neuronales artificiales pertenecen a un campo de la inteligencia artificial llamado Machine Learning. Cuando hablamos de redes neuronales profundas, redes neuronales recurrentes o redes neuronales convolucionales, nos estamos metiendo en un subcampo del Machine Learning, llamado Deep Learning.

Para tener un concepto claro, antes de meternos en materia vamos a definir qué es una red neuronal artificial (y qué son los perceptrones que las forman).

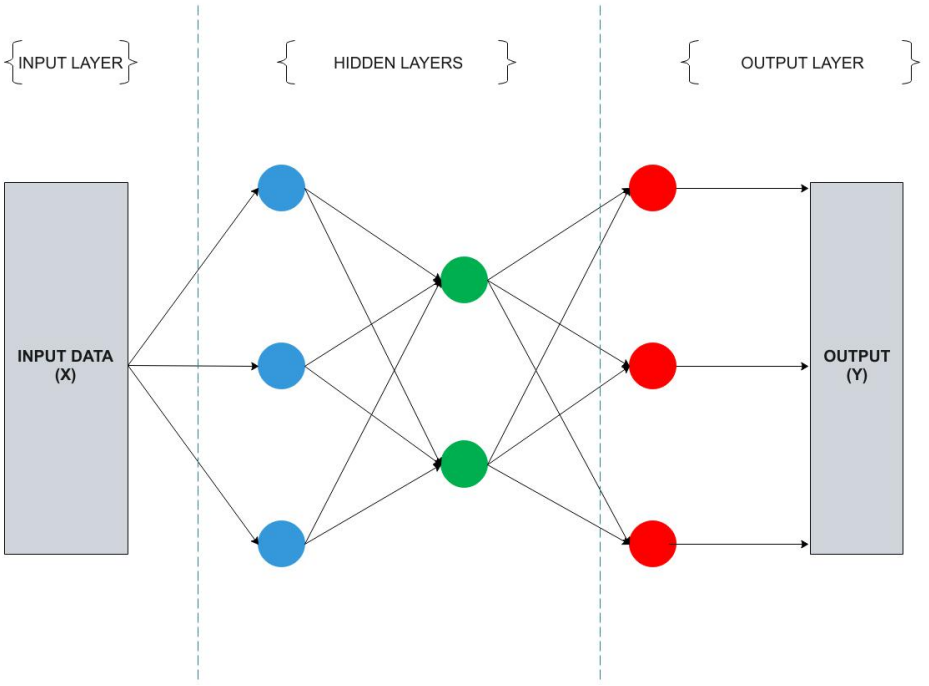
1.1. ¿Qué es una red neuronal artificial?

Una red neuronal artificial es algo así como un "cerebro", que puede aprender ciertas cosas (como clasificar fotos de gatitos o predecir el precio de una casa) a partir de unos datos.

Se compone de unas neuronas (perceptrones) y unos enlaces sinápticos con cierta calidad de conexión (pesos).

Las redes neuronales profundas pueden tener muchas capas. A la capa de entrada se le llama **Input Layer**, a las capas ocultas (se llaman así porque no sabemos lo que pasa en ellas) las llamamos **Hidden Layers**, y a la capa de salida (la última) **Output Layer**.

A continuación vemos un diagrama que muestra un ejemplo de la estructura de una red neuronal:



Viendo este diagrama, nos damos cuenta que a la red neuronal le pasamos unos datos y, al final, nos devuelve algo.

Las líneas que unen los círculos son los **enlaces sinápticos**, que tienen asociados unos **pesos** (calidad de la conexión, biológicamente hablando) que determinan cuánta información pasará por ese enlace.

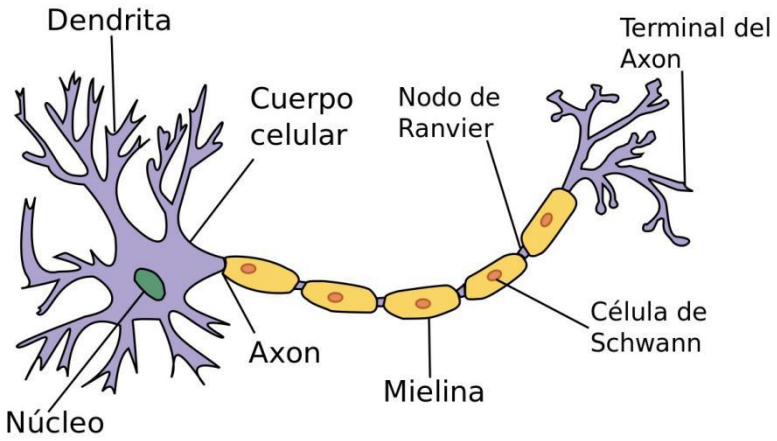
Cada círculo que vemos en el diagrama es un **perceptrón**. Lo definimos a continuación.

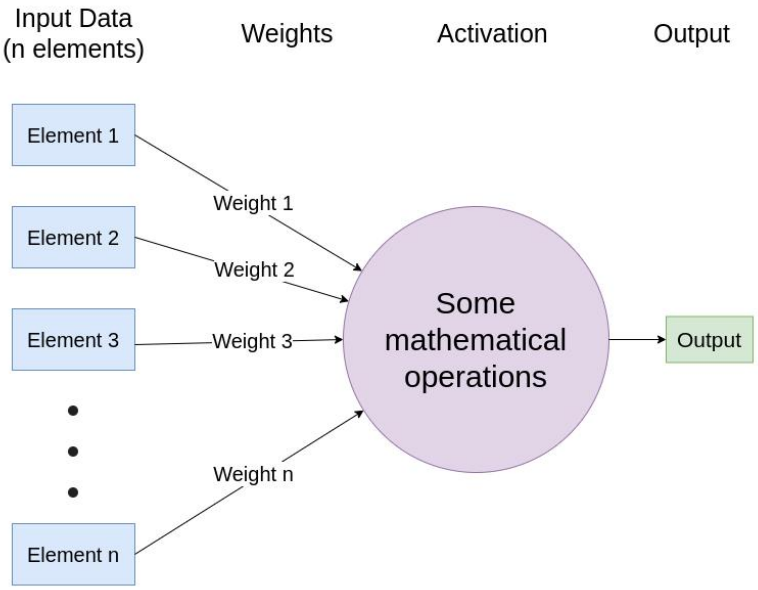
1.2. ¿Qué es un Perceptrón?

Los perceptrones (neuronas) son los **"ladrillos"** que forman las redes neuronales, es decir, son la base de toda red neuronal. Pero antes de nada, vamos a describir qué hace una neurona.

Una neurona recibe un pulso eléctrico a través de unos enlaces llamados dendritas. Si la calidad del enlace es lo suficientemente buena, dicho pulso eléctrico activa la neurona y, a través del axón hasta llegar a la terminal del axón, pasa el impulso eléctrico a otras neuronas con las que esté conectada.

Vamos a hacer una comparativa entre un perceptrón y una neurona para tener una idea más clara de lo que es un perceptrón:





Vemos claras similitudes. Las **dendritas** de la neurona son nuestros **enlaces**, por donde nos llega la información, y su **calidad de conexión** serían nuestros **pesos**. El **núcleo** lo podríamos equiparar a nuestra **activación** (más tarde veremos qué se esconde en esas operaciones matemáticas), donde se decide si la neurona/perceptrón se activa y da una salida, o no. Por último nuestra **salida** equivaldría a la **terminal del axón**, donde se conectan más neuronas (en nuestro caso más perceptrones).

Debemos tener en cuenta que el dato que pasamos al perceptrón es solo uno, que puede estar compuesto de distintos elementos. Esto lo veremos detalladamente más tarde.

Teniendo ya una idea clara del concepto de perceptrón y de red neuronal, vamos a ver cómo se han desarrollado las redes neuronales a lo largo del tiempo.

1.3. Historia de las redes neuronales en IA

La base de las redes neuronales (biológicas) como las conocemos hoy día, se remontan a **1873** y **1890** cuando **Alexander Bain** (en 1873) y **William James** (en 1890), filósofo y psicólogo respectivamente, en sus respectivos trabajos, propusieron que los pensamientos y las actividades que realizaba el cuerpo eran resultado de interacciones entre las neuronas que había en el cerebro. Tras esto otros psicólogos y científicos empezaron a investigar en este campo.

En **1943 Warren McCulloch** y **Walter Pitts**, neurofisiólogo y lógico respectivamente, crearon un modelo computacional basado en las matemáticas y en algoritmos. Lo llamaron **threshold logic.** Fueron los que introdujeron la idea de las redes neuronales en la computación.

En **1949**, **Donald Hebb** (psicólogo) creó una hipótesis sobre el aprendizaje, basándose en la plasticidad de las neuronas. Hoy en día esta hipótesis se conoce como **aprendizaje de Hebbian**.

En **1958**, **Frank Rosenblatt** (psicólogo), creó el perceptrón.

En **1969** las redes neuronales se estancaron, ya que se dieron cuenta de que en esa época no había ordenadores capaces de computar las redes neuronales. Otro problema por el que se estancó fue por las limitaciones del perceptrón, que no era capaz de resolver el problema de la **operación XOR** (exclusive-or). Estos problemas se solucionaron cuando los ordenadores aumentaron su potencia y cuando se creó el algoritmo de **propagación hacia atrás** (backpropagation), que ponía fin al problema XOR.

El procesamiento paralelo distribuido se hizo popular a **mitad de los 80** bajo el nombre de conexionismo. Tras esto, en **1986**, se expuso el uso del conexionismo para simular los procesos neuronales.

Tras todo esto se siguió avanzando en el campo, hasta que en **2012 Andrew Ng** y **Jeff Dean** crearon una red neuronal capaz de **reconocer imágenes de gatos**. Este acontecimiento, sumado al incremento en el poder computacional de las GPU y la computación distribuida, dieron

lugar al boom del término **Deep Learning**, aunque ya existía dicho término desde **1986**, introducido por Rina Dechter.

Después de poner un poco en contexto la historia de las redes neuronales, vamos a ver detalladamente (y matemáticamente) cómo son los perceptrones y las redes neuronales.

2. *Las matemáticas de las Redes Neuronales*

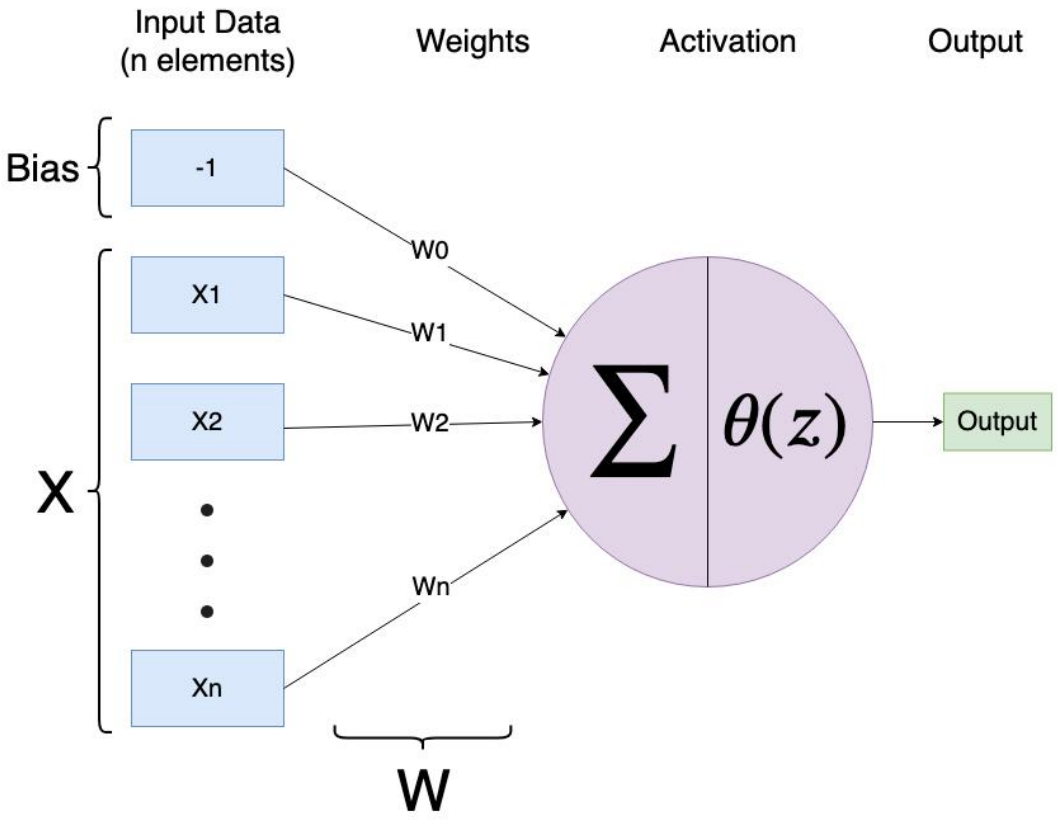
Las redes neuronales artificiales se basan en las matemáticas para funcionar. Todo lo que está detrás de las redes y los perceptrones son matemáticas, por lo tanto si queremos entender bien qué hacen y cómo hacen las cosas tenemos que entender las matemáticas que hay detrás.

Empezaremos con el perceptrón y luego pasaremos a las redes neuronales, ya que estas últimas están compuestas por perceptrones, lo que nos da una ligera idea que tendrá más o menos una estructura parecida.

En general, lo que vamos a ver es cómo el perceptrón (y la red neuronal) genera una salida.

2.1. Perceptrón

Vamos a ver un diagrama actualizado con "las cosas matemáticas":



Como vemos, hemos añadido y modificado algunos elementos para hacer la explicación más sencilla. A continuación vamos a explicar en detalle en qué consiste cada elemento del perceptrón.

- **X** : Es el dato que le llega al perceptrón. Consiste en un vector de uno o más elementos con el que vamos a generar la salida del perceptrón (más tarde también lo entrenaremos y haremos predicciones).

- **W** : Son los pesos de las conexiones del perceptrón. Al igual que X, es un vector. Su tamaño depende de la cantidad de elementos que tenga el vector X.

- **Bias** : En nuestro caso usaremos $(-1 * W_0)$, pero puede ser $(1 * W_0)$. Matemáticamente, la bias nos permite deplazar la salida de la función de activación hacia la derecha o hacia la izquierda (si lo viésemos en una gráfica de dos dimensiones). Conceptualmente, es el umbral que tienen que superar el sumatorio para que el perceptrón se active.

- **Sumatorio** (**∑**) : Es parte del "núcleo" del perceptrón. Es la suma de todos los productos de cada elemento de X y su peso correspondiente en W. A todo esto habría que sumarle la Bias, es decir, al final tendríamos: $(\sum_{i=1}^{n} X_i * W_i) + (-1 * W_0)$

- **Función de activación** (θ(z)) : Suele ser una función no linear ya que nos proporcionan un mayor y mejor aprendizaje. El concepto es que cogemos el resultado del sumatorio (z) y le aplicamos una función (θ). El resultado de esta función es el Output de nuestro perceptrón. Las funciones de activación más usadas son la función sigmoide, la función ReLu y la función Tanh. La elección de la función de activación depende del resultado que queramos obtener.

Después de estas definiciones podríamos intuir que al final tendríamos:

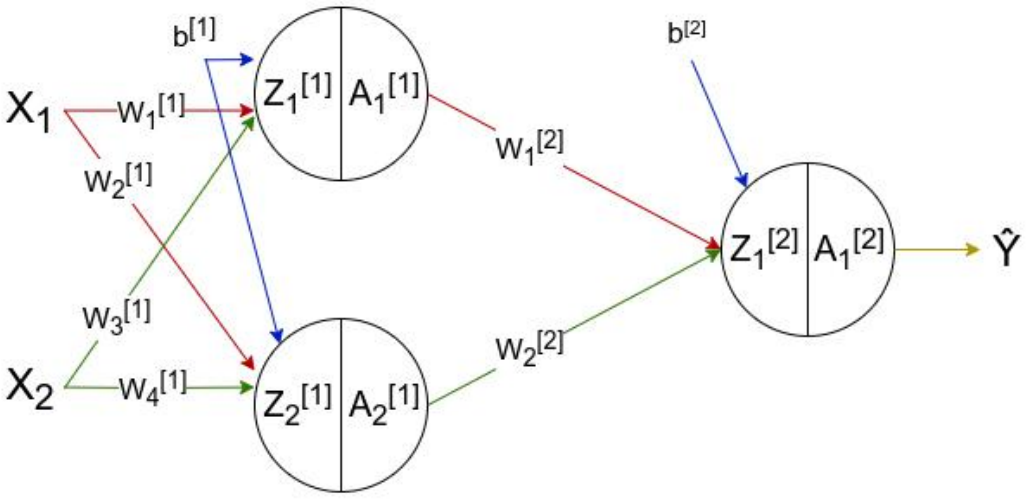
$$\mathbf{Output} = \theta(z)$$

$$\mathbf{z} = (\sum_{i=1}^{n} X_i * W_i) + (-1 * W_0)$$

2.2. Redes Neuronales

Como hemos visto, es una "combinación" de perceptrones. Para hacer los cálculos más sencillos vamos a usar productos escalares, por lo que vamos a tener que hacer uso de nociones de álgebra (muy presente en las redes neuronales y en el machine learning en general).

Ahora vamos a ver un esquema de como quedaría una red neuronal artificial añadiéndole las "cosas matemáticas", como hicimos con el perceptrón (vamos a usar una más simple que la que usamos en el apartado 1.1.) :



Vamos a explicar detalladamente cada parte de esta nueva red:

- $\boldsymbol{X_i}$: Es el elemento *i* de nuestro dato de entrada X.

- $\boldsymbol{W_j^{[i]}}$: Es el peso *j* de la matriz de pesos de la capa *i* .

- $\boldsymbol{b^{[i]}}$: Es el vector de bias de la capa *i*. Un elemento por perceptron. Sería $(-1 * W_0)$ en nuestro caso, como explicamos en el perceptrón.

- $\boldsymbol{Z_j^{[i]}}$: Es el resultado del sumatorio del perceptrón *j* de la capa *i* .

- $\boldsymbol{A_j^{[i]}}$: Es el resultado de aplicar la función de activación a $Z_j^{[i]}$.

- Ŷ : Es la salida de la red, el Output.

Una vez detallados los elementos de la red, vamos a ver, al igual que hicimos en el perceptrón, cómo se computa la salida (Ŷ). Primero veremos unos cálculos generalizados, y luego veremos cómo computar la salida específicamente para nuestra red.

Generalizando, tenemos:

$$Z^{[i]} = W^{[i]} \bullet A^{[i-1]} + b^{[i]}$$

$$A^{[i]} = \theta^{[i]}(Z^{[i]})$$

$$\hat{Y} = \theta^{[L]}(Z^{[L]}) = A^{[L]}$$

Al aplicar estas ecuaciones, debemos tener en cuenta algunas cosas:

- La operación • es el producto escalar.

- $A^{[0]} = Z^{[0]} = X$.

- L es el número de capas de la red, por lo que la capa L es la última capa.

- $\theta^{[i]}$ es la función de activación de la capa i .

Llegados a este punto, vamos a ver cómo quedarían las fórmulas para computar el Output de nuestra red neuronal.

Para la primera capa, tenemos:

$$Z^{[1]} = W^{[1]} \bullet A^{[0]} + b^{[1]} = \begin{bmatrix} W_1^{[1]} & W_3^{[1]} \\ W_2^{[1]} & W_4^{[1]} \end{bmatrix} \bullet \begin{bmatrix} X_1 \\ X_2 \end{bmatrix} + \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \end{bmatrix} =$$

$$= \begin{bmatrix} W_1^{[1]} * X_1 + W_3^{[1]} * X_2 + b_1^{[1]} \\ W_2^{[1]} * X_1 + W_4^{[1]} * X_2 + b_2^{[1]} \end{bmatrix} = \begin{bmatrix} Z_1^{[1]} \\ Z_2^{[1]} \end{bmatrix}$$

$$A^{[1]} = \theta^{[1]}(Z^{[1]}) = \begin{bmatrix} \theta^{[1]}(Z_1^{[1]}) \\ \theta^{[1]}(Z_2^{[1]}) \end{bmatrix}$$

Para la segunda, y última capa, tenemos:

$$Z^{[2]} = W^{[2]} \bullet A^{[1]} + b^{[2]} = \begin{bmatrix} W_1^{[2]} \\ W_2^{[2]} \end{bmatrix} \bullet \begin{bmatrix} Z_1^{[1]} \\ Z_2^{[1]} \end{bmatrix} + \begin{bmatrix} b_1^{[2]} \end{bmatrix} =$$

$$= \begin{bmatrix} W_1^{[2]} * Z_1^{[1]} + W_2^{[2]} * Z_2^{[1]} + b_1^{[2]} \end{bmatrix} = \begin{bmatrix} Z_1^{[2]} \end{bmatrix}$$

$$A^{[2]} = \theta^{[2]}(Z^{[2]}) = \begin{bmatrix} \theta^{[2]}(Z_1^{[2]}) \end{bmatrix}$$

$$\hat{Y} = \theta^{[2]}(Z^{[2]}) = A^{[2]}$$

Tras ver este ejemplo, deberíamos tener un concepto bastante claro de cómo se computa la salida en una red neuronal. Pero esto solo no basta para tener una red que nos sirva. Para que la red sea funcional debemos hacer que **aprenda**. Esto nos lleva al siguiente punto.

3. ¿Cómo aprenden las Redes Neuronales?

Para que las redes neuronales aprendan necesitamos tener datos de entrenamiento (los que vamos a usar para predecir) y sus respectivos valores "predichos" o "targets" de antemano.

A grandes rasgos "aprender", para una red neuronal, significa ir actualizando los pesos en cada iteración del aprendizaje adaptándose a los datos de entrenamiento para, al final, obtener una salida adecuada a lo que esperamos (minimizar la función de coste, lo que es lo mismo, acercar nuestro resultado a los targets).

Vamos a ver cómo aprenden las redes neuronales mediante una técnica llamada descenso por el gradiente, ya que es muy usada en la realidad y es una de las mejores maneras de empezar a entender cómo aprenden estas redes.

3.1. Función de pérdida y de coste

Es la función (en concreto el valor que devuelva la función) que la red va a tratar de minimizar. Se llama función de pérdida cuando se le aplica a un solo ejemplo y función de coste cuando se calcula la media de todas las funciones de pérdida de todos los ejemplos.

Vamos a ver una función de pérdida llamada regresión logística, que sirve para clasificación binaria:

$$L(\hat{y}, y) = -(y * \log(\hat{y}) + (1 - y) * \log(1 - \hat{y}))$$

Los elementos que aparecen en esta función son:

- $\hat{y}$: Es el output de nuestro perceptrón, lo que hemos predicho.

- **y** : Es el valor real que se espera, nuestro "target".

Es difícil ver la utilidad de esta función para usarla como función de pérdida, vamos a explicarla:

Imaginemos que $y = 1$ (es decir, el resultado que se espera es 1), entonces si sustitumimos y en la función, tenemos:

$$L(\hat{y}, 1) = -(1 * \log(\hat{y}) + (1 - 1) * \log(1 - \hat{y})) =$$
$$= -(1 * \log(\hat{y}))$$

Para minimizar $-(1 * \log(\hat{y}))$ queremos que $\hat{y}$ (nuestra predicción) sea lo más grande posible, es decir, 1. Si nos damos cuenta, 1 era lo que valía nuestro target, por lo que nos aproximamos a él.

Imaginemos ahora que $y = 0$ (es decir, el resultado que se espera es 0), entonces si sustitumimos y en la función, tenemos:

$$L(\hat{y}, 0) = -(0 * \log(\hat{y}) + (1 - 0) * \log(1 - \hat{y})) = \\ = 1 * \log(1 - \hat{y})$$

Para minimizar $1 * \log(1 - \hat{y})$ queremos que $\hat{y}$ (nuestra predicción) sea lo más pequeña posible, es decir, 0. Tal como pasaba en el ejemplo anterior, 0 es el valor de nuestro target

Como dijimos al principio, esta función de pérdida es solo para un ejemplo. Si queremos obtener la función de coste (la que calcula la media de todas las funciones de pérdida de todos los ejemplos) tenemos que usar:

$$J(w, b) = \frac{1}{m} * \sum_{i=1}^{m} L(\hat{y}_i, y_i)$$

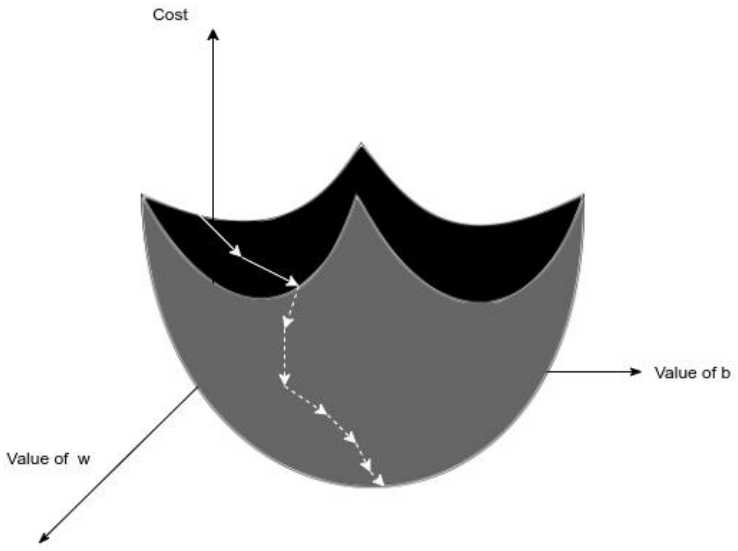
Usamos "w"(los pesos) y "b"(la bias) como argumentos de la función porque son los elementos que vamos a ir actualizando para obtener la salida que queremos ($\hat{y}$). Debemos tener en cuenta que "m" es el número de ejemplos que tenemos.

Otra función de coste a la que hacer mención es la función MAE (Mean Absolute Error, se usa por ejemplo en regresión):

$$MAE = \frac{1}{m} \sum_{i=1}^{m} |y_j - \hat{y_j}|$$

3.2. Descenso por el Gradiente

La idea del descenso por el gradiente es sencilla, vamos bajando por la pendiente de la función hasta llegar al mínimo (mínimo local, aunque a veces global). Veamos el siguiente diagrama:



El eje y indica el valor que devuelve la función de coste (lo que queremos minimizar) y en el eje x vamos a poner el valor del peso y en el eje z el valor de la bias. Como vemos, vamos bajando por la pendiente de la función (vamos cambiando los pesos y la bias) hasta llegar al valor mínimo del coste. El tamaño de los pasos que damos por la pendiente se llama **tasa de aprendizaje** (learning rate).

Debemos tener en mente que la dirección en la que tenemos que ir bajando por la función la marca la derivada de la función en ese punto.

Los pesos y la bias se actualizan de la siguiente manera:

$$W_i = W_i - \alpha * dW_i$$

$$b = b - \alpha * db$$

Donde α es el learning rate y dW_i y db_i son las derivadas de W_i y de b_i respectivamente.

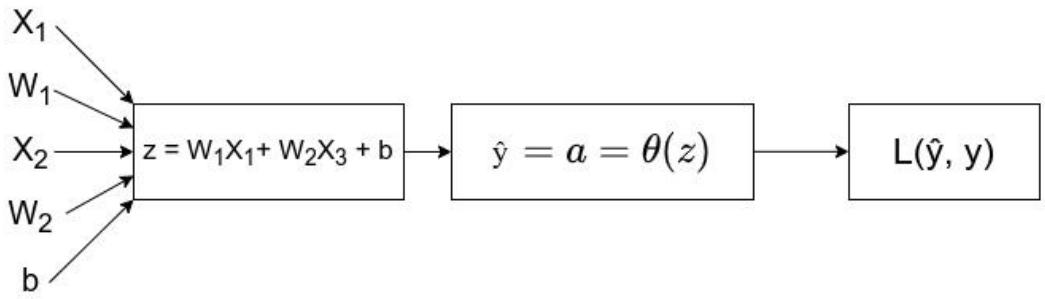
La actualización de los pesos la llevaremos a cabo al final de la propagación hacia detrás.

Teniendo todo esto en cuenta, vamos a ver ahora cómo propagar hacia delante y hacia detrás

3.3. Propagación hacia delante

La propagación hacia delante es, básicamente, computar la salida de la red (o del perceptrón).

Vamos a ver un ejemplo para un perceptrón con un ejemplo de dos elementos:

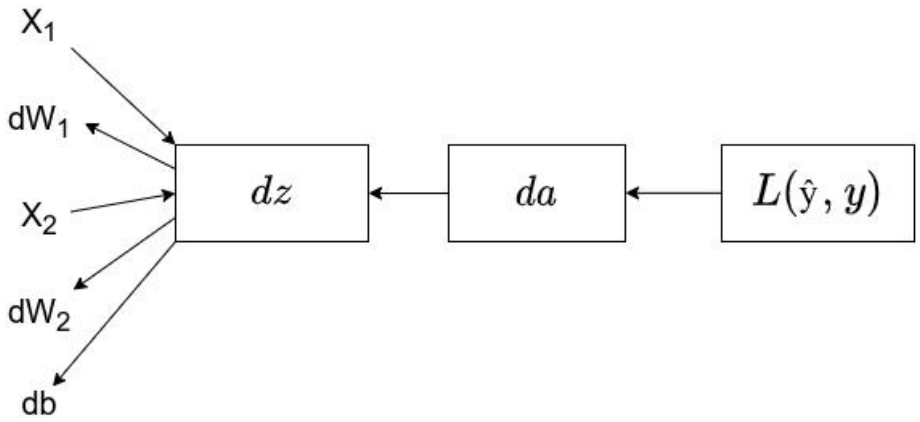


Así habríamos calculado hacia delante todos los componentes de este grafo de propagación hacia delante.

Ahora, teniendo todo esto calculado, procederemos a propagar el resultado hacia detrás para poder actualizar los pesos.

3.4. Propagación hacia detrás

En la propagación hacia detrás vamos computando las derivadas de los distintos elementos del grafo anterior. Para verlo claro miramos el siguiente grafo:



Vemos que es parecido, pero ahora las flechas van hacia detrás y vamos calculando las derivadas. A continuación vemos cómo calcular esas derivadas :

$$\mathbf{da} = \frac{dL}{da}$$

$$\mathbf{dz} = \frac{\mathrm{dL}}{\mathrm{dz}} = \frac{\mathrm{dL}}{\mathrm{da}} * \frac{\mathrm{da}}{\mathrm{dz}}$$

$$\mathbf{dW_i} = \frac{\mathrm{dL}}{\mathrm{dW_i}} = \frac{\mathrm{dL}}{\mathrm{da}} * \frac{\mathrm{da}}{\mathrm{dz}} * \frac{\mathrm{dz}}{\mathrm{dW_i}}$$

Tras obtener las derivadas podemos actualizar los pesos tal y como vimos en el punto 3.2:

$$\boldsymbol{W_1} = W_1 - \alpha * dW_1$$

$$\boldsymbol{W_2} = W_2 - \alpha * dW_2$$

$$\boldsymbol{b} = b - \alpha * db$$

Acabamos de hacer una iteración en el aprendizaje de un perceptrón.

3.5. Aprendiendo una red neuronal

Usando la misma notación que en el apartado 2.2 vamos a ver cómo se computa la propagación hacia delante (que en realidad es lo mismo que

computar la salida de la red) y la propagación hacia detrás para actualizar los pesos:

Propagación hacia delante:

$$\boldsymbol{Z^{[i]}} = W^{[i]} \bullet A^{[i-1]} + b^{[i]}$$

$$\boldsymbol{A^{[i]}} = \theta^{[i]}(Z^{[i]})$$

$$\hat{\mathrm{Y}} = \theta^{[\mathrm{L}]}(\mathrm{Z}^{[\mathrm{L}]}) = \mathrm{A}^{[\mathrm{L}]}$$

Debemos tener en cuenta un caso especial ,como vimos anteriormente, en el que:

$$\boldsymbol{A^{[0]}} = Z^{[0]} = X$$

Este es el caso de la primera capa, que son los ejemplos.

Propagación hacia detrás:

$$\boldsymbol{dZ^{[i]}} = dA^{[i]} * \theta^{[i]\,'}(Z^{[i]})$$

$$\mathbf{dA^{[i-1]}} = \mathrm{W}^{[\mathrm{i}]\,\mathrm{T}} \bullet \mathrm{dZ}^{[\mathrm{i}]}$$

$$\boldsymbol{dW^{[i]}} = \frac{1}{m} * (dZ^{[i]} \bullet A^{[i-1]\,T})$$

$$\mathbf{db^{[i]}} = \frac{1}{m} * \mathrm{sumrows}(\mathrm{Z}^{[\mathrm{i}]})$$

Donde $\boldsymbol{\theta^{[i]\,'}}$ es la derivada de la función de activación de la capa *i* y $\mathbf{sumrows}(\mathbf{Z^{[i]}})$ *devuelve un vector en la que cada fila es la suma de cada fila de* $\mathrm{Z}^{[\mathrm{i}]}$.

Debemos tener en cuenta un caso especial en el que:

$$\boldsymbol{dZ^{[L]}} = A^{[L]} - Y$$

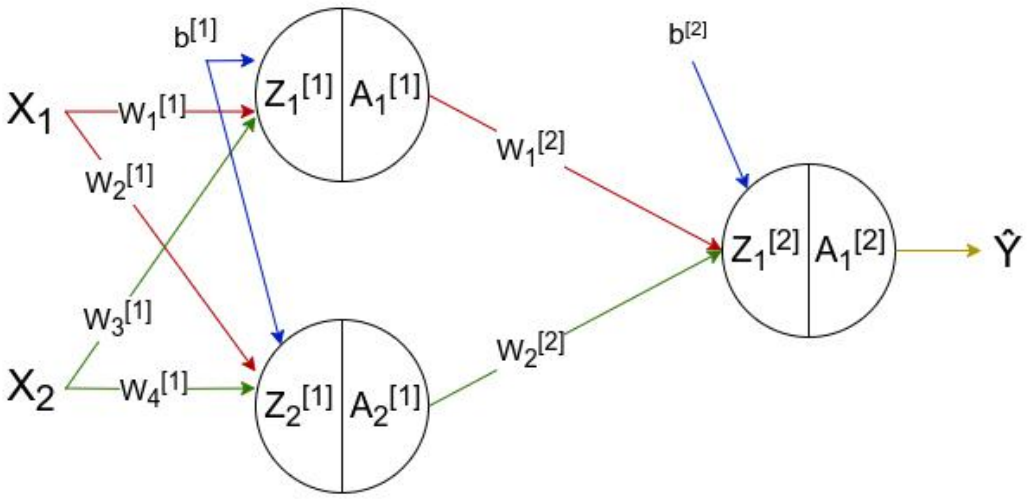
Este es el caso de la última capa, en la que debemos ver cúanto nos hemos equivocado entre la salida de nuestra red y lo que debería salir.

Actualización de pesos:

$$\boldsymbol{W^{[i]}} = W^{[i]} - \alpha * dW^{[i]}$$

$$\boldsymbol{b^{[i]}} = b^{[i]} - \alpha * db^{[i]}$$

Ahora vamos a ver un ejemplo de propagación hacia delante y hacia detrás con la red que vimos en el apartado 2.2. Vamos a recordarla:



Según lo que hemos visto, tendríamos primero que propagar hacia delante computando la salida de la red, y luego, propagar hacia detrás para actualizar los pesos. Vamos a ver cómo sería:

Propagación hacia delante:

$$\boldsymbol{Z^{[1]}} = W^{[1]} \bullet X + b^{[1]}$$

$$\boldsymbol{A^{[1]}} = \theta^{[1]}(Z^{[1]})$$

$$\boldsymbol{Z^{[2]}} = W^{[2]} \bullet A^{[1]} + b^{[2]}$$

$$\hat{\mathrm{Y}} = A^{[2]} = \theta^{[2]}(Z^{[2]})$$

Llegados a este punto habríamos computado la salida de nuestra red, es decir, hemos propagado hacia delante. Ahora nos quedaría propagar el error cometido hacia detrás para poder actualizar los pesos correctamente.

Propagación hacia detrás:

$$\boldsymbol{dZ^{[2]}} = A^{[2]} - Y = \hat{Y} - Y$$

$$\mathbf{dA^{[1]}} = \mathrm{W}^{[2]\,\mathrm{T}} \bullet \mathrm{dZ}^{[2]}$$

$$\boldsymbol{dW^{[2]}} = \frac{1}{m} * (dZ^{[2]} \bullet A^{[1]\,T})$$

$$\mathbf{db}^{[2]} = \frac{1}{m} * \text{sumrows}(\text{Z}^{[2]})$$

$$\boldsymbol{dZ}^{[1]} = dA^{[1]} * \theta^{[1]\,'}(Z^{[1]})$$

$$\mathbf{dA}^{[0]} = \text{W}^{[1]\,\text{T}} \bullet \text{dZ}^{[1]}$$

$$\boldsymbol{dW}^{[1]} = \frac{1}{m} * (dZ^{[1]} \bullet X^{T})$$

$$\mathbf{db}^{[1]} = \frac{1}{m} * \text{sumrows}(\text{Z}^{[1]})$$

Ahora estamos preparados para actualizar los pesos de nuestra red ya que tenemos todas las derivadas necesarias para ello.

Actualización de pesos:

$$\boldsymbol{W}^{[2]} = W^{[2]} - \alpha * dW^{[2]}$$

$$\boldsymbol{b}^{[2]} = b^{[2]} - \alpha * db^{[2]}$$

$$\boldsymbol{W}^{[1]} = W^{[1]} - \alpha * dW^{[1]}$$

$$\boldsymbol{b}^{[1]} = b^{[1]} - \alpha * db^{[1]}$$

Hemos llegado al final de la primera iteración del aprendizaje. Esto lo podríamos repetir tantas veces como queramos para que nuestra red aprendiese más y más, pero hay que tener cuidado con el overfitting y, en contraparte, el underfitting.

3.6. Overfitting y underfitting

El **overfitting** ocurre cuando nuestra red neuronal se ajusta demasiado a los datos de entrenamiento y al darle otros datos sobre los que hacer una predicción no lo hace bien. Esto es, en esencia, que tiene un buen rendimiento sobre los datos de entrenamiento pero no generaliza, o no tiene un buen rendimiento, sobre otros datos nuevos.

El **underfitting**, por otra parte, ocurre cuando no entrenamos lo suficiente a nuestra red. Esto puede ocurrir porque no tengamos los datos suficientes o porque no hayamos iterado las veces que la red necesita para aprender. Por ejemplo puede pasar si entrenamos una red para que reconozca perros y solo lo entrenamos con perros de color marrón. Cuando le pasemos una foto de un perro de color blanco la red probablemente no sea capaz de clasificar bien la imagen como un perro ya que no tiene ejemplos previos de perros blancos.

En el siguiente diagrama vemos gráficamente cómo sería cada uno de los casos que se nos pueden presentar: