Contenido

[2. Cosas que hacer en el documento. 4](#_Toc160450406)

[3. Estructura del documento. 4](#_Toc160450407)

[4. Conceptos básicos transversales. 4](#_Toc160450408)

[4.1 Introducción. 4](#_Toc160450409)

[4.2 Datos estructurados vs Datos no estructurados. 4](#_Toc160450410)

[4.3 Representación de una imagen. 5](#_Toc160450411)

[4.4 Librería H5 para volúmenes de datos. 7](#_Toc160450412)

[4.5 Decisiones que se deben tomar al construir una red neuronal. 7](#_Toc160450413)

[4.6 Vectorización. 8](#_Toc160450414)

[4.1 Broadcasting en Python. 9](#_Toc160450415)

[5. Introducción al deep learning 11](#_Toc160450416)

[5.1 Introducción. 11](#_Toc160450417)

[5.2 ¿Que es una neurona? 11](#_Toc160450418)

[5.3 Una red neuronal 12](#_Toc160450419)

[5.4 Red neuronal densamente conectada. 13](#_Toc160450420)

[6. Base matemática para entender funcionamiento de una neurona. 13](#_Toc160450421)

[6.1 Aquí repaso de ¿Qué es una regresión lineal? 13](#_Toc160450422)

[6.2 Función logística de regresión. 13](#_Toc160450423)

[6.3 Función de coste de la regresión logística. 14](#_Toc160450424)

[6.4 Descenso de gradiente: Optimización de funciones. 15](#_Toc160450425)

[6.5 Seudocódigo descenso de gradiente en Python. 17](#_Toc160450426)

[6.6 Versión vectorizada en Python del foward-propagatio. 18](#_Toc160450427)

[6.7 Versión vectorizada en Python de backward-propagation. 20](#_Toc160450428)

[6.8 Algoritmo vectorizado de descenso de gradiente en Python. 21](#_Toc160450429)

[7. Funciones de activación 22](#_Toc160450430)

[7.1 Introducción. 22](#_Toc160450431)

[7.2 Función logística. 23](#_Toc160450432)

[7.3 Función hiperbólica tangente. 23](#_Toc160450433)

[7.4 ReLu function 24](#_Toc160450434)

[7.5 Leaky ReLu 25](#_Toc160450435)

[7.6 Criterios de seleccion de una función de activación. 25](#_Toc160450436)

[7.7 Derivadas de funciones de activación. 26](#_Toc160450437)

[8. Red neuronal de dos capas. 26](#_Toc160450438)

[8.1 Representación de una red neuronal de dos capas. 26](#_Toc160450439)

[8.2 Cálculo de salidas de las capas para un ejemplo. 27](#_Toc160450440)

[8.3 Calculo para “m” ejemplos. 29](#_Toc160450441)

[8.4 Descenso de gradiente en una red neuronal de dos capas. 32](#_Toc160450442)

[8.5 Inicialización de pesos y bias a números aleatorios. 33](#_Toc160450443)

[8.6 Implementación de algoritmo de descenso de gradiente en una red neuronal de dos capas. 34](#_Toc160450444)

[9. Red neuronal profunda. 35](#_Toc160450445)

[9.1 Introducción. 35](#_Toc160450446)

[9.2 Principales elementos de una red neuronal profunda. 35](#_Toc160450447)

[9.3 Propagación hacia delante. 36](#_Toc160450448)

[9.4 Dimensiones de redes neuronales. 37](#_Toc160450449)

[9.5 Redes neuronales profundas. 38](#_Toc160450450)

[9.6 Flujo básico de una red neuronal. 39](#_Toc160450451)

[9.7 Propagación hacia delante y hacia atrás. 40](#_Toc160450452)

[9.8 Parámetros e hiperparámetros. 41](#_Toc160450453)

[9.9 Implementación en Python de algoritmo de descenso de gradiente en redes neuronales multicapa. 42](#_Toc160450454)

[10. Aspectos prácticos del machine learning. 44](#_Toc160450455)

[10.1 Naturaleza iterativa 44](#_Toc160450456)

[10.2 División del conjunto de datos de entrenamiento. 44](#_Toc160450457)

[10.3 Bias y varianza. 46](#_Toc160450458)

[10.4 Enfoque sistemático de acción según Bias y varianza. 47](#_Toc160450459)

[11. Técnicas de regularización. 48](#_Toc160450460)

[11.1 Regularización con la regresión Ridge (Penalización L2). 48](#_Toc160450461)

[11.2 Regularización de Ridge en Python. 50](#_Toc160450462)

[11.3 Regularización “Drop-out” 51](#_Toc160450463)

[11.4 Aumentación de datos. 53](#_Toc160450464)

[11.5 Early Stopping. 54](#_Toc160450465)

[12. Técnicas de optimización. 55](#_Toc160450466)

[12.1 Normalización de inputs. 55](#_Toc160450467)

[12.2 Desaparición o explosión de gradientes. 56](#_Toc160450468)

[12.3 Inicialización de pesos. 56](#_Toc160450469)

[12.4 Aproximación numérica de gradientes. 57](#_Toc160450470)

[13. Algoritmos de optimización. 59](#_Toc160450471)

[13.1 Introducción 59](#_Toc160450472)

[13.2 Mini-batch Descenso de gradiente. 59](#_Toc160450473)

[13.3 El termino de época. 60](#_Toc160450474)

[13.4 Promedios Ponderados Exponenciales, PPE. 61](#_Toc160450475)

[13.5 Corrección del sesgo. 63](#_Toc160450476)

[13.6 Descenso de gradiente con momentum. 63](#_Toc160450477)

[13.7 RMSprop 65](#_Toc160450478)

[13.8 ADAM 65](#_Toc160450479)

[13.9 Learning Rate decay. 67](#_Toc160450480)

# Cosas que hacer en el documento.

* ~~Buscar donde poner funciones de activación.~~
* Definir previamente a Función logística de regresión, que es una función de activación y que es una función de activación.
* Refina introducción de apartado 7.
* En el final del apartado 7 buscar añadir la explicación de código que hay en las carpetas anexas a este documento.
* Explicar por algún lado lo de la g como función versátil de activación.
* Insertar por algún lado lo de inicialización de pesos adecuada.

# Estructura del documento.

El objetivo de este documento es que personas con cierta inquietud con el mundo del machine learning puedan introducirse en el campo con mayor facilidad.

En el epígrafe Introducción al machine learning se da respuesta superficial de que es el machine Learning, su taxonomía y el flujo de trabajo para abordar problemas de machine learning, así como conceptos transversales del machine learning y las librerías que se utilizan en Python en el contexto de machine learning.

# Conceptos básicos transversales.

## Introducción.

En esta sección se verán conceptos básicos que son transversales a toda la materia de aprendizaje profundo.

## Datos estructurados vs Datos no estructurados.

En el contexto de las redes neuronales y el aprendizaje automático, los datos se pueden categorizar ampliamente en dos tipos: estructurados y no estructurados. La diferencia entre ellos es importante porque determina cómo se preparan y procesan los datos antes de usarlos para entrenar un modelo de aprendizaje automático.

* **Datos Estructurados:** Son aquellos que están organizados en un formato predecible, por lo general en tablas con filas y columnas, como las bases de datos o las hojas de cálculo. Cada fila representa un registro (por ejemplo, una casa), y cada columna representa una característica específica de ese registro (por ejemplo, el número de habitaciones, el tamaño de la casa, etc.). Estos datos son fáciles de buscar, manipular y analizar porque siguen un esquema definido. En el caso de las redes neuronales, los datos estructurados se utilizan para entrenar modelos que pueden hacer predicciones precisas, como predecir el precio de una casa basándose en características como la ubicación, el tamaño, y la cantidad de habitaciones.
* **Datos No Estructurados:** En contraste, los datos no estructurados no siguen un formato predecible o un esquema fácil de manejar. Incluyen todo tipo de formatos como texto libre (emails, tweets, documentos), imágenes, audio y video. Estos datos son más complejos y requieren preprocesamiento adicional para extraer características que puedan ser utilizadas por una red neuronal. Por ejemplo, una red neuronal que se utiliza para el reconocimiento de imágenes primero debe aprender a identificar características relevantes de las imágenes crudas, como bordes, texturas y patrones de color, antes de poder clasificar la imagen o entender su contenido.

\*\*Aplicación en Redes Neuronales:\*\*

Las redes neuronales se pueden diseñar y entrenar para manejar ambos tipos de datos, pero la arquitectura del modelo y el proceso de entrenamiento pueden ser significativamente diferentes.

- Para \*\*datos estructurados\*\*, se utilizan redes neuronales tradicionales o redes neuronales profundas (deep learning) que pueden manejar entradas numéricas y realizar tareas como clasificación, regresión o predicción de series temporales.

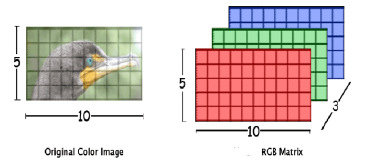
- Para \*\*datos no estructurados\*\*, se utilizan arquitecturas especializadas como redes neuronales convolucionales (CNNs) para imágenes, redes neuronales recurrentes (RNNs) para texto o secuencias de tiempo, y redes neuronales de procesamiento de lenguaje natural (NLP) para el análisis de texto.

En resumen, mientras que los datos estructurados se ajustan bien a las redes neuronales "estándar" y son más directos para el análisis, los datos no estructurados requieren un enfoque más complejo y redes neuronales especializadas para descubrir patrones ocultos y obtener información valiosa.

## Representación de una imagen.

Normalmente una imagen estará representada por tres matrices, cada una de las matrices tendrá dimensión “n x m”, siendo “n” el número de filas de la matriz y “m” el número de columnas. Además, en este caso, tendremos tantas filas como pixeles tenga la imagen de altura, y tantas columnas como pixeles de ancho tenga la imagen, es decir, que, si tenemos una imagen HD, de 1920x1080 pixeles, cada matriz tendrá 1080 filas y 1920 columnas.

¿Por qué son necesarias tres matrices? Púes bien, cada matriz representa el nivel de intensidad de un determinado color, si usamos RGB, los 3 colores que nos interesan son Red, Green and Blue, de tal forma que la representación de una imagen podría quedar como se representa en la siguiente imagen:



Ahora bien, para transformar esto en una estructura de información manejable en la programación tenemos varias opciones, la primera es meter todos los valores de las 3 matrices en un único array unidimensional

La longitud de “” será el ancho de la imagen en pixeles por la altura de la imagen en pixeles por 3, es decir:

La forma más común de aplanar estas matrices en un vector es seguir un orden específico, como de izquierda a derecha y de arriba hacia abajo, pasando por cada píxel y añadiendo sus componentes RGB al vector. Existen principalmente dos convenciones para hacer esto:

* **Por filas:** Se comienza con la fila superior de la imagen, se recorre cada píxel de izquierda a derecha, y se añaden primero el valor rojo, luego el verde y finalmente el azul. Una vez completada la fila, se pasa a la siguiente fila hasta cubrir toda la imagen.
* **Por columnas:** Similar al método anterior, pero en vez de recorrer las filas, se recorre cada columna de arriba hacia abajo, y por cada píxel, se añaden sus componentes RGB al vector.

En ambos casos, el resultado es un vector largo donde cada conjunto de tres valores consecutivos representa los colores (rojo, verde, azul) de un píxel en la imagen.

Una vez tenemos nuestra imagen un vector , podemos usar esa información para un problema de clasificación binaria, de tal forma que para entrenar un algoritmo de clasificación binaria tendremos un conjunto de “ejemplos” de entrenamiento, que tendrán cada una, una entrada (imagen en forma de vector) y una salida y que podrá ser 0 o 1, más formalmente:

nº ejemplos

Ejemplo n-ésimo

Además, tendremos un conjunto de test, que tendrán exactamente la misma forma que un ejemplo de entrenamiento, pero son casos específicos para comprobar que el algoritmo hace predicciones correctas, este conjunto se denotará como .

Finalmente, tendremos una matriz que tendrá tantas columnas como ejemplos de entrenamiento hay, es decir, tendrá dimensión “m”, y tendrá tantas filas como el valor de , es decir, que tendrá tantas filas como longitud tenga el vector , y también tendremos una para denotar un vector de salidas, tal que:

Nota: La manera de diferenciar entre una entrada o un conjunto de entradas radica en si es mayúscula o minúscula la “x”, lo mismo sucede con la salida “y”.

## Librería H5 para volúmenes de datos.

## Decisiones que se deben tomar al construir una red neuronal.

* Elegir función de activación que más se adecue a nuestro problema:

## Vectorización.

La vectorización es el proceso de conversión de operaciones que normalmente se ejecutarían a través de bucles en operaciones sobre vectores o matrices completas, permitiendo que se realicen de manera simultánea y eficiente, aprovechando las capacidades de hardware como los procesadores vectoriales o las instrucciones SIMD (**S**ingle Instruction, Multiple Data). Esto mejora significativamente el rendimiento al reducir el número de instrucciones de programación necesarias y aumentar la velocidad de ejecución.

En este contexto y con fines didácticos la manera que tendremos de implementar la vectorización será con la librería Numpy, la cual, nos abstrae de toda la complejidad que implica a nivel de código construir instrucciones SIMD y nos facilita la vida a los programadores dándonos una serie de funciones a las que podemos llamar sin preocuparnos por toda la configuración que requiere la vectorización.

Imaginemos que queremos una tarea muy sencilla, dado dos vectores queremos obtener un tercer vector que sea la suma de ambos, podemos aplicar dos estrategias para conseguir esto, la primera es con un for-loop, siendo esta la menos eficiente y quedando como sigue:

Una captura de pantalla de un celular

Descripción generada automáticamente

Este mismo problema vectorizado quedaría como sigue:

Captura de pantalla de un celular

Descripción generada automáticamente

Internamente Numpy transformará esto en instrucciones que se puede ejecutar en paralelo con diferentes datos (SIMD), evitando un estilo de programación secuencial (uno detrás del otro) típico de un bucle for, normalmente ejecutar varias tareas en paralelo siempre será mucho más rápido que ejecutar varias tareas de manera secuencial. Ahora bien, si tenemos la siguiente función a nivel matemático:

**Nota:** Recordar que es el número de características que tiene el ejemplo de entrada, por ejemplo, una imagen de 1024x1024 pixeles representada en formato RGB, se traducirá en que existen 1024x1024x3 características.

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamenteUna manera de transformar la anterior fórmula matemática en código Python con dos versiones (la vectorizada y la no vectorizada) sería la que sigue:

Texto

Descripción generada automáticamente

La versión vectorizada será en torno a 400 veces más rápida que la versión vectorizada, aunque esto también depende del contexto especifico (hardware, SO, etc). Aunque al principio parezca un reto vectorizar código, la estrategia no es memorizar y aplicar cuando sea necesario, sino que dado un código escrito de manera “tradicional” intentar buscar en Google, ChatGPT, Stackoverflow, Google, etc maneras de vectorizarlo y estudiar ese caso particular, por ejemplo, si queremos calcular el logaritmo neperiano en base dos, podemos solicitar a chatGPT que nos haga una transformación de un código no-vectorizado a un código vectorizado:

Texto

Descripción generada automáticamenteTexto

Descripción generada automáticamente

Vistos algunos ejemplos, nuestro objetivo a partir de ahora será crear el algoritmo de descenso de gradiente en su versión vectorizada, para poder hacer el entrenamiento de una neurona mucho más rápido.

## Broadcasting en Python.

El **broadcasting** es un concepto poderoso en Python, especialmente utilizado en bibliotecas como NumPy y Pandas, que permite realizar operaciones aritméticas entre arrays de diferentes formas (o tamaños) sin necesidad de duplicar los datos en la memoria. Este mecanismo hace que las operaciones sean más eficientes y rápidas, ya que evita la necesidad de usar bucles explícitos para iterar sobre los elementos de los arrays.

Broadcasting traducido al español se entiende como transmisión, un array más pequeño se "transmite" a lo largo de un array más grande de manera que sus elementos se extienden para coincidir con la forma del array más grande, permitiendo operaciones elemento a elemento. Por ejemplo, imaginemos que queremos suma el numero 1 a todos los elementos de una matriz, el código queda como sigue:

Texto

Descripción generada automáticamente

Como podemos apreciar en lugar de tener que haber creado una matriz de 3 x 3 con todos sus valores igual a 1 y sumárselo a la variable “matriz”, gracias al broadcasting conseguimos que esta operación sea mucho más sencilla y eficiente que tener que crear la matriz y usar bucles for. Veamos algunos ejemplos:

* El primer caso es uno en el que vemos como si sumamos a un vector de (n,m) un único elemento, automáticamente en el proceso de suma, se crea un tensor que se ajusta al tamaño del sumando 1.

Tip para trabajo: No es necesario memorizarse estas propiedades, con que te suenen es suficiente, luego a la hora de la realidad, utilizaras buscadores de internet, para encontrar la solución al problema que tengas, la clave aquí es quedarse con que si vas a hacer operaciones entre tensores (matrices, vectores, etc) de diferentes dimensiones utilizando bucles for y teniendo que hacer adaptaciones, preguntante antes si puedes utilizar broadcasting.

# Introducción al deep learning

## Introducción.

En este apartado estudiaremos de manera superficial que es una neurona y como la conexión entre varias de ellas dan lugar a una red neuronal, será en apartados más avanzados cuando veremos al detalle a nivel matemático y de programación que es una red neuronal.

## ¿Que es una neurona?

Una neurona en el contexto del deep learning y la inteligencia artificial es una unidad fundamental de procesamiento que simula el funcionamiento de una neurona biológica en el cerebro humano. Estas "neuronas artificiales" se utilizan para realizar cálculos y tomar decisiones en modelos de aprendizaje automático, como redes neuronales.

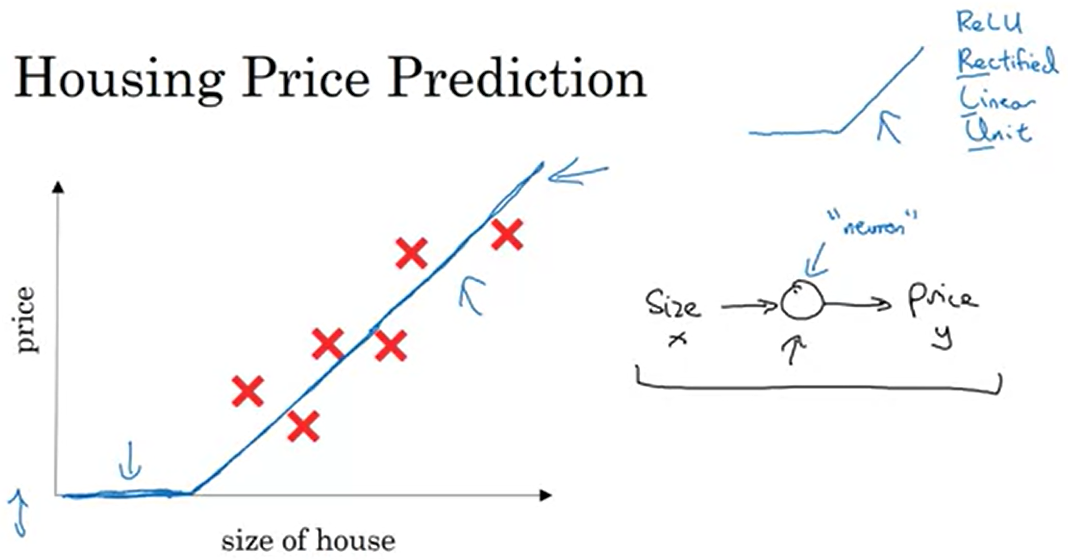


Ilustración —https://www.coursera.org/learn/neural-networks-deep-learning/lecture/eAE2G/what-is-a-neural-network

En este caso más simple, tenemos una grafica que captura la relacion entre el precio de una casa y el tamaño de la casa, claramente se ve que la relación entre estas dos caracteristicas es directamente proporcional y que se puede capturar con una funcion clasica de tipo , sin embargo, si solo dejaramos esta función existirian valores del tamaño de la casa que darían un precio negativo y esto no tendría mucho sentido, es por esto que para tener una representación más fideligna de la realidad podemos usar lo que se denomina una función de activación que envuelva nuestra función original ).

Una función de activación apropiada para este caso es la función ReLU, que tiene la forma , o contextualizado a nuestro caso quedaría como , de esta manera ya tendríamos una manera simple de representar nuestra realidad con una función matemática.

Púes bien, en este caso, una neurona podría definirse como un ente, que dada una entrada “x” (tamaño), genera una salida “y”(precio de la casa), la neurona actua como una función matemática.

## Una red neuronal

Claramente el precio de una vivienda no solo depende de su tamaño, hay otros actores como el numero de habitaciones, numero de parques, numero de edificios de servicios publicos (educación, sanida, policia), numero de transportes publicos ofrecidos en la zona, etc.

Como podemos apreciar, al tener varios factores que delimitarán el precio de una vivienda el problema se vuelve más complejo y es más dificil representarlo con una neurona

Diagrama

Descripción generada automáticamente

Como podemos apreciar, cada neurona esta especializada en hacer diferntes cosas y la union de varias hace que se llegue a la tarea más compleja de predecir el precio de una casa. En este contexto, **la entrada de la red neuronal sería X** que es un vector de datos de entrada, es decir:

[Tamaño de lacasa, numero de habitaciones, numero de colegios, numero de institutos, número de bibliotecas, paradas de autobus, estación de tren]

Un ejemplo de como sería esta entrada: [200m², 3, 5, 2, 1, 8, 2]

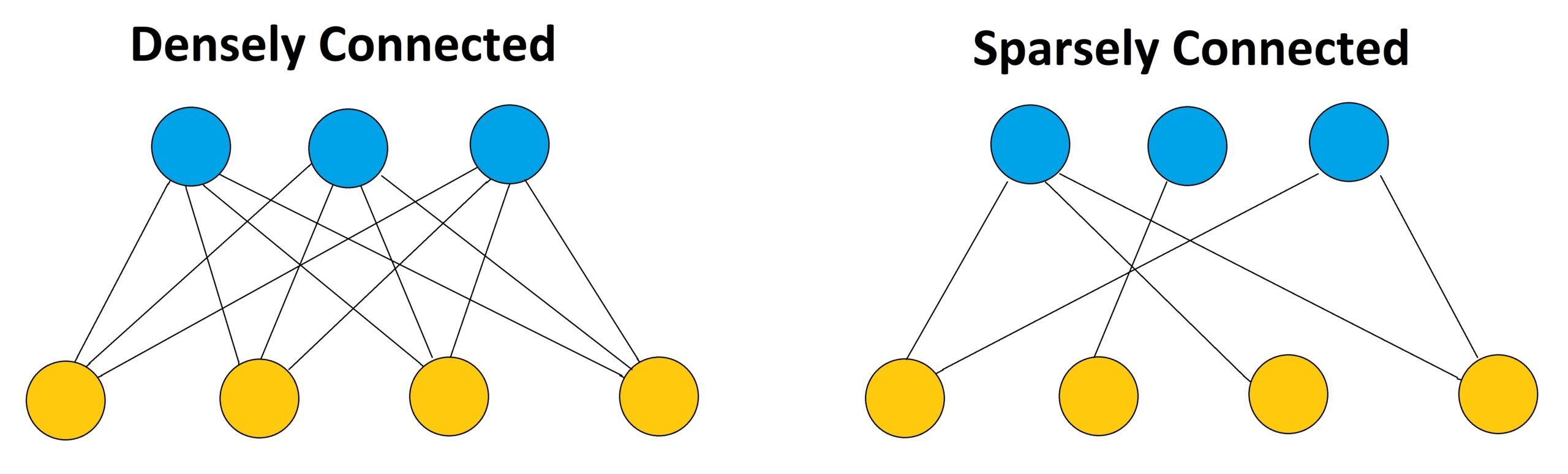
**La salida sería el precio de la casa**, un aspecto importante a reseñar, es que normalmente, tendrémos que proveer a la red neuronal, un conjunto de entrenamiento que consiste en ristras que tengan valores de caracteristicas y un precio de la realidad, es decir:

[200m², 3, 5, 2, 1, 8, 2] => 230.000€

[120m², 1, 1, 0, 0, 0, 1] => 150.000€

Cuando introducimos estos datos, la red neuronal aprende a relacionar las caracteristicas con el precio de la casa ella sola, realmente lo que hay de manera intermedia, no suele ser nuestro trabajo aprenderlo, sino de la red neuronal.

## Red neuronal densamente conectada.



Una red neuronal densamente conectada es aquella en la que todas las neuronas estan conectadas con todas las neuronas.

# Base matemática para entender funcionamiento de una neurona.

## Aquí repaso de ¿Qué es una regresión lineal?

## Función logística de regresión.

La función logistica de regresión, es una función de clasificación binaria. La fórmula de la regresión logística viene dada por la siguiente expresión:

El objetivo es definir los valores de los pesos ( y del sesgo , para que dada cualquier entrada se predizca un valor de optimo. La anterior función más expandida quedaría como sigue:

Si lo convertimos en un problema más genérico con matrices quedaría de la siguiente manera:

La función logística fuerza a que el resultado final de sea un valor entre 0 y 1, de tal forma de que para todos aquellos valores de las características ( que hagan que la función logística de como resultado un valor menor que 0.5 sean clasificados en una clase y para el caso de que sean mayores que 0.5 sean clasificados en la clase opuesta.

Gráficamente hablando la regresión logística para clasificación binaria se vería como una línea ligeramente curvada para problemas de una dimensión (una entrada y una salida), un plano para problemas de dos dimensiones o un hiperplano para problemas con dimensiones mayores, tanto si es una línea, un plano como una figura geométrica más abstracta y difícil de visualizar todas ellas separarían la nube de datos del conjunto de entrenamiento en dos espacios, uno para una clase y otro para la otra clase, visualmente, en el caso de dos dimensiones sería como la siguiente gráfica:

[Gráfico, Gráfico de dispersión

Descripción generada automáticamente](https://machine-learning.paperspace.com/wiki/logistic-regression)

## Función de coste de la regresión logística.

Nuestro objetivo es que nuestra función de regresión logistica sea capaz de clasificar en dos clases posibles determinada información que se le da como entrada , tal que “(i)” denota el ejemplo i-esimo. Una manera de poner a prueba nuestra función logistica es darle entradas para las que ya conocemos cual es la salida correcta, es decir, sabemos en cual de las dos clases debemos clasificarla y comprobar si la salida predicha por nuestra función se aleja mucho de la salida correcta .

Lo más normal es que un modelo no sea capaz de acertar al 100% el valor que debería tener “y”, pero si que se aproxime, la diferencia entre lo que se ha predicho y lo que es realmente se conoce como perdida o margen de error. Sin embargo, hay que retocar un poco esta funcion para que los numeros negativos no le afecte quedando como sigue:

En la regresión logística, el objetivo es predecir variables categóricas, generalmente en clasificación binaria. Aquí, se desea que la salida sea una probabilidad que varía entre 0 y 1. La función sigmoide, como mencionamos antes, es ideal para esto porque transforma los valores de entrada en probabilidades.

El problema con la función de pérdida cuadrática en este contexto es que puede conducir a problemas en el entrenamiento del modelo, como no converger a un mínimo global debido a la forma de la función de costo cuando se aplica a la función sigmoide. Esto es particularmente problemático debido a cómo las predicciones de probabilidad interactúan con el error cuadrático. Para solucionar estos problemas es más optimo utilizar la función de perdida logística, la cual viene dada por:

Ahora bien, este anterior caso esta aplicado para un caso especifico (un ejemplo), para aplicar la función de coste a nivel global, es decir, teniendo en cuenta todo el conjunto de entrenamiento y no solo un ejemplo, tenemos la siguiente expresión:

Tal que denota el ejemplo i-esimo.

## Descenso de gradiente: Optimización de funciones.

Tal y como vimos en el epigrafe anterior, la función define el coste o el error medio que comete la red neuronal con sus prediciones, claramente, nuestro objetivo es minimizar esta función, de tal manera que si lo hacemos estaremos consiguiendo minimizar el error que comete nuestra red neuronal.

La manera en la que minimizamos esta función es **ajustando los valores de “w” y de “b”** hasta encontrar el **valor minimo de la función**, aquí es donde entra el metodo del **descenso de gradiente**, el cual, utiliza los gradientes (derivadas parciales) que se calculan mediante un metodo que denominaremos **backpropagation** para saber cómo ajustar los parámetros (pesos y bias) y así minimizar la función de coste.

Para verlo más claro veamos un poco más en profundidad los diferentes pasos del metodo del descenso de gradiente:

1. Inicializar “w” y “b” a unos determinados valores aleatorios, este es el punto de partida para empezar la optimización.
2. Se realiza un proceso denominado “forward propagation”, el cual, introduce unas entradas en la red neuronal de un determinado ejemplo y propaga esas entradas desde la primera capa (capa de entrada) hasta la última capa (capa de salida), para obtener una predicción.
3. Se calculan las derivadas (o gradientes) parciales de la función de coste con respecto a cada parámetro (w y b), de esta forma, obtenemos información de como cada uno de los parámetros afecta a la función de coste, es decir, como una pequeña variación en los pesos afecta afecta a la función de coste objetivo a optimizar, que al fin y al cabo es la definición en esencia de lo que es una derivada.

Además, si la derivada parcial de la función de coste con respecto a un parámetro es positiva, significa que aumentar ese parámetro incrementará el valor de la función de coste. Si la derivada es negativa, aumentar ese parámetro disminuirá el valor de la función de coste. Por lo tanto, para minimizar la función de coste, queremos movernos en la dirección opuesta al signo de la derivada parcial.

**El backpropagation** producirá un vector gradiente que contiene todas las derivadas parciales de la función de coste con respecto a cada uno de los parámetros de la red (cada peso y el bias). En una red neuronal, estos parámetros suelen ser los pesos y los sesgos de todas las conexiones entre las neuronas.

1. El vector gradiente es crítico para el proceso de optimización, ya que indica la dirección en la cual la función de coste aumenta más rápidamente. Por lo tanto, para minimizar la función de coste, se ajustan los parámetros en la dirección opuesta al gradiente (obtenido tras el back-prograpagation).

Además, e**l tamaño del paso en esa dirección está determinado por el learning rate**. Si el learning rate es demasiado pequeño, el entrenamiento de la red puede ser muy lento, ya que los pasos hacia el mínimo de la función de coste son muy pequeños. Por otro lado, si el learning rate es demasiado grande, podríamos "saltarnos" el mínimo y hacer que el entrenamiento sea inestable. Las funciones que definen esta actualización de los pesos vienen dada por:

También en el caso del “bias”, se definirá de la siguiente forma:

1. Se establece un criterio de detención que determina cuándo se considera que el entrenamiento ha convergido o ha alcanzado un estado deseado. Esto puede ser un número máximo de iteraciones, un umbral de error, o cualquier otro criterio específico para la tarea. Si se ha cumplido el criterio de detección se para el entrenamiento de la red neuronal, en caso de que no, se vuelve al paso 2.

## Seudocódigo descenso de gradiente en Python.

Antes de entrar en el código Python, primero debemos observar el pseudocodigo del método de descenso de gradiente para un problema con dos características y una única neurona, con una estrategia más clásica con bucle-for, el cual, quedaría como sigue:

**def** gradient\_descent\_algoritmh()

# inicialización de variables

**for** i = 1 **in** m:

# Va acumulando costo de entrenamiento

El problema de esta estrategia es que es muy ineficiente dado que cada vuelta del bucle for se ejecuta de manera secuencial, una detrás de otra, implicando un coste computacional muy alto, para solucionar este problema recurrimos a algo **denominado “vectorización”**, que lo que busca es cambiar esa estrategia de “uno detrás de otro” y pasar a una estrategia de “todos a la vez en paralelo”, esta estrategia la veremos con más detalle en el siguiente epígrafe.

## Versión vectorizada en Python del foward-propagatio.

Tal y como vimos en el epígrafe 5.3, el seudocódigo del descenso de gradiente para “m” ejemplos de 2 características, quedaría como sigue:

Texto

Descripción generada automáticamente

Con el objetivo de hacer un algoritmo vectorizado en Python que implemente el algoritmo de descenso de gradiente de manera genérica, vamos a extender el problema para “n” características, quedando como sigue la representación matricial del conjunto de ejemplos:

Tal que “” es el número de características de cada ejemplo () y “m” es el número de ejemplos, es decir, que si invocamos el método X.shape en Python tendremos como resultado la tupla .

El Foward propagation, recordando es el caso de aplicar unas entradas a nuestra neurona, propagarlas hacia delante y obtener una salida predicha por la neurona. En definitiva, sería obtener el resultado de la función:

Para ello, primero calculamos “z” y después aplicamos la función sigmoidea a z quedándonos:

Tras esta operación tendremos un vector con todas las regresiones lineales calculadas para cada ejemplo y bastaría calcular la función sigmoide para cada uno de los ejemplos, dándonos como resultado un vector “”

Esto en código Python quedaría como sigue:

Texto

Descripción generada automáticamente

Ilustración — Propagación hacia delante

**Nota:** Elegimos “y\_predicted” como valor porque no podemos escribir en Python su notación matemática .

Además, debemos tener en cuenta que en la línea 16 de este código se esta u

## Versión vectorizada en Python de backward-propagation.

Tras haber obtenido el valor de las salidas para todos los ejemplos, en forma de vector pasamos a calcular la diferencia entre lo predicho y lo real, y así estudiar el error del problema.

Recordemos que dZ representa la derivada de la función de coste con respecto de Z, es un paso intermedio (por la regla de la cadena de derivadas) para poder calcular las derivadas con respecto de los pesos y el bias. Ahora con el fin de ir calculando esos gradientes acumulados para todos los ejemplos antes teníamos este fragmento de seudocódigo:

Texto

Descripción generada automáticamente

Nuestro objetivo ahora es convertirlo en matrices para evitar que sea necesario iterar con un bucle for a lo largo de todos los ejemplos, para ello:

Recordando que será la derivada de la función de coste con respecto de los pesos. Lo mismo ocurre con tal que:

Ahora si ponemos todo junto en código Python

Texto

Descripción generada automáticamente

## Algoritmo vectorizado de descenso de gradiente en Python.

Finalmente, si unimos lo aprendido en el epígrafe 5.6 y 5.7 y añadimos la actualización de los pesos y el bias en dirección opuesta a lo que indiquen los gradientes, podremos apreciar que quedará lo siguiente:

Texto

Descripción generada automáticamenteTexto

Descripción generada automáticamente

Tras la ejecución de este algoritmo, podemos utilizar los valores de los pesos y el bias que optimizan la función de coste calculados en el algoritmo de descenso de gradiente y utilizarlos para predecir nuevos casos,

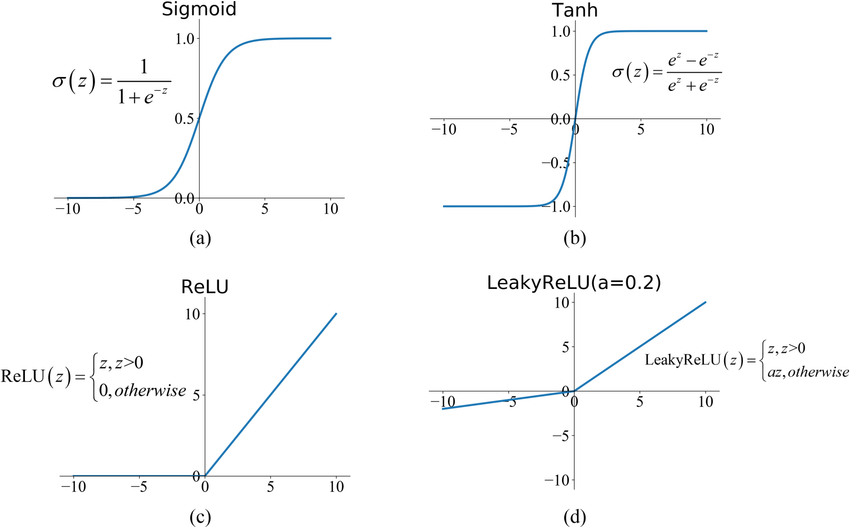
# Funciones de activación

## Introducción.

Hasta este momento hemos visto que, al querer calcular la propagación hacia delante de una neurona, o lo que es lo mismo, la salida de la neurona, utilizábamos la función:

La función o función logística actúa como algo que denominaremos “función de activación”. Una función de activación es una función matemática que se le aplica a la salida de una neurona o capas de neuronas, sus propósitos claves son:

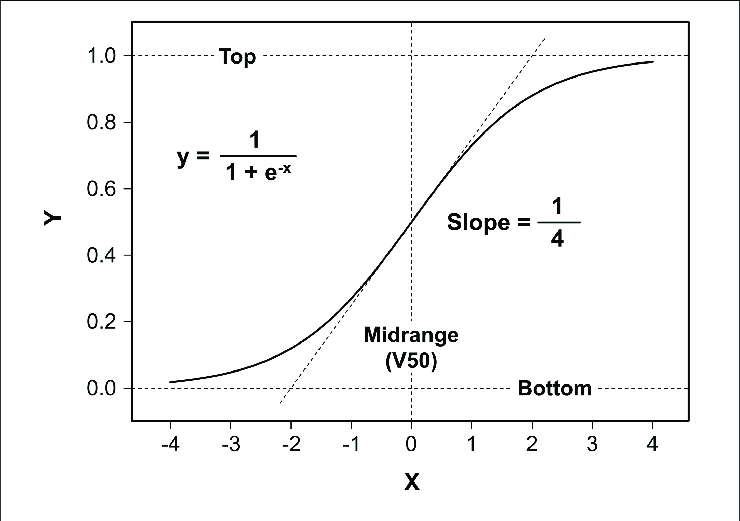
* **No linealidad:** Las funciones de activación introducen no linealidades en el modelo, lo que permite a la red neuronal aprender y representar relaciones más complejas entre los datos de entrada y salida
* **Transformación de datos:** Convierte la entrada neta de la neurona en una salida que pueda ser consumida por otras neuronas.
* **Control de Rango de Salida:** Controlar que la salida genere un rango de valores
* **Decisión de Fuego:** Se decide si la neurona se activa o no se activa y transmite información a las siguientes neuronas.

A continuación, procederemos a analizar varios tipos de Función de activación

## Función logística.

La función logística, también conocida como la función sigmoide, es una de las funciones de activación más fundamentales y ampliamente utilizadas en el campo del aprendizaje automático y las redes neuronales. Es **especialmente prevalente en problemas de clasificación binaria** debido a su capacidad para mapear predicciones a probabilidades que varían entre 0 y 1. A continuación, exploramos las características y aplicaciones de la función logística, así como su representación gráfica, que nos proporciona una visión intuitiva de cómo transforma los valores de entrada en un rango limitado y manejable. Su expresión matemática queda definida por:

Y tiene a nivel grafico la siguiente forma:



## Función hiperbólica tangente.

La función tangente hiperbólica, comúnmente abreviada como , es otra función de activación crucial utilizada en redes neuronales artificiales. Similar a la función logística en cuanto a que es una función sigmoidal, la función tiene la ventaja de centrar su salida en torno a cero, lo que a menudo **conduce a una convergencia más rápida durante el entrenamiento**. La salida de la función varía entre -1 y 1, y su forma simétrica permite que maneje datos negativos de manera más efectiva. Esta característica la hace especialmente útil en capas ocultas de redes neuronales **profundas donde la normalización de los datos es beneficiosa**. Su formula matemática viene definida por:

Diagrama

Descripción generada automáticamente

Y su representación gráfica queda como sigue:

## ReLu function

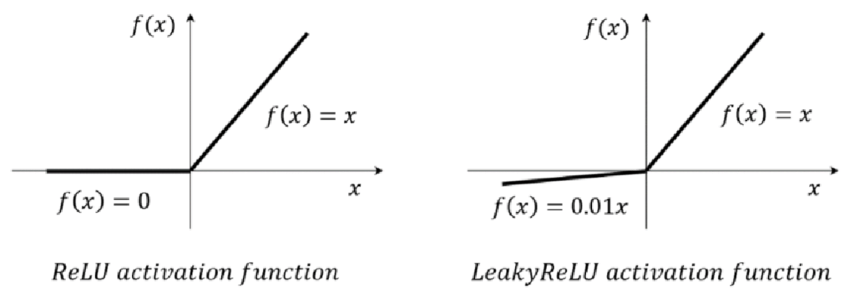
La función ReLU (Rectified Linear Unit) es una función de activación ampliamente utilizada en las redes neuronales y el aprendizaje profundo. Esta función es especialmente popular en el campo del procesamiento de imágenes y el reconocimiento de patrones, pero también se aplica en diversas otras tareas de aprendizaje automático y deep learning debido a su simplicidad y eficacia. La función ReLU se define matemáticamente de la siguiente manera:

En palabras simples, la función ReLU toma un valor de entrada x y devuelve x si x es mayor o igual a cero, o devuelve cero si x es negativo. Esto significa que la función ReLU es lineal para valores positivos y se aplana en cero para valores negativos.

Sin embargo, la función ReLU también tiene algunas limitaciones, como el problema de "neuronas muertas" (dead neurons). Esto ocurre cuando una neurona siempre produce una salida cero para cualquier entrada, lo que significa que no contribuye al aprendizaje de la red. Para abordar este problema, se han desarrollado variantes de la función ReLU, como **Leaky ReLU y Parametric ReLU (PReLU)**, que introducen ligeras modificaciones para permitir un flujo de gradiente incluso para valores negativos pequeños.

La función ReLU es una función de activación fundamental en el aprendizaje profundo que aporta no linealidad, eficiencia computacional y resuelve problemas de desvanecimiento del gradiente. A pesar de sus limitaciones, sigue siendo una elección popular en muchas arquitecturas de redes neuronales.

## Leaky ReLu



## Criterios de seleccion de una función de activación.

|  |  |  |
| --- | --- | --- |
| Función de Activación | Ventajas | Desventajas |
| Tanh | - Rango de salida simétrico alrededor de 0 que puede ayudar a la convergencia.  - Puede mejorar la eficiencia del entrenamiento en comparación con la sigmoide. | - Puede sufrir de desvanecimiento de gradientes en redes profundas.  - **No es ideal** para redes **con entradas no normalizadas.** |
| Logística (Sigmoide) | - Rango de salida entre 0 y 1, natural para probabilidades.  - Históricamente popular para clasificación binaria. | - Sufre de desvanecimiento de gradientes.  - Las salidas no están centradas alrededor de 0.  - No es eficiente para redes muy profundas.  - Nunca usar esta, excepto para la capa de salida si se está haciendo clasificación binaria, porque Tanh es una versión mejorada de esta. |
| ReLU | - Muy eficiente computacionalmente.  - Acelera la convergencia en la práctica en comparación con otras funciones.  - Es la más utilizada | - Neuronas pueden "morir" durante el entrenamiento si la activación es siempre negativa (dead ReLU problem). |
| Leaky ReLU | - Intenta solucionar el problema de las neuronas muertas en ReLU permitiendo una pequeña pendiente para valores negativos. | - El coeficiente de la "fuga" necesita ser elegido cuidadosamente.  - No hay un estándar establecido para el valor de este coeficiente. |

Tip trabajo: Normalmente es muy dificil saber a ciencia cierta que funcion de activación aplicar a nuestro problema, tenemos unas guías generales que nos permiten intentar cercar cual de ellas deberiamos utilizar, sin embargo, es interesante adquirir una mentalidad cientifica y de experimentación e ir probando diferentes conbinaciones de uso de funciones de activación y ver cual es la que mejor resultados nos da.

## Derivadas de funciones de activación.

Tal y como hemos apreciado en el algoritmo del descenso de gradiente, las derivadas son importantes en el proceso de optimización de una red neuronal, es por ello, que en este apartado se ven cada una de las derivadas de las funciones de activación que se han visto en este capitulo.

|  |  |  |
| --- | --- | --- |
| **Nombre** | Función original | Función derivada |
| Tanh |  | Versión simplificada: |
| Logística (Sigmoide) |  | Versión simplificada: |
| ReLU |  |  |
| Leaky ReLU |  |  |

# Red neuronal de dos capas.

## Representación de una red neuronal de dos capas.

Diagrama

Descripción generada automáticamente

Los elementos clave que podemos identificar en el anterior diagrama son los siguientes:

* **Capa de entrada ():** Está compuesta por las características que definen un problema, por ejemplo, si nuestra red neuronal quiere predecir precios de vivienda, nuestro problema esta caracterizado por “tamaño de vivienda en metros cuadrados”, “número de habitaciones”, etc. Cada representa unas características que representa el problema.
* **Capa oculta :** Constituyen todas las neuronas que están entre la capa de entrada y la capa de salida, es decir, está compuesta por neuronas que procesan las señales de la capa de entrada y las convierten en señales para la capa de salida. El término “oculto” viene del hecho de que en el proceso de entrenamiento de una red neuronal “no observamos” los valores intermedios que hay en la red neuronal, nosotros siempre observaremos la entrada y la salida que se le da a la red neuronal, pero rara vez necesitaremos observar lo que pasa en los niveles intermedios.
* **Capa de salida :** Es responsable de calcular el valor predicho o la salida de la red neuronal para un determinado input. Por ejemplo, en el caso de la red neuronal para la predicción de precios de vivienda, la capa de salida podría producir un valor numérico que representa el precio predicho de la vivienda.
* **Consideraciones adicionales**: La capa oculta y la capa de salida tendrán un conjunto de pesos y bias, por cada neurona habrá un bias diferente y tantos pesos como características haya en el problema. Por ejemplo, los pesos de la capa 1 denotados como se representarán con una matriz de dimensión (4,3), es decir, 4 filas y 3 columnas, el cuatro lo deducimos porque en la capa 1 hay 4 nodos o neuronas y el 3 porque nuestro problema tiene 3 características. Adicionalmente, el bias de la capa 1 denotado como será un vector de dimensión (4,1) dado que tiene 4 nodos esa capa y por cada uno de ellos habrá un bias asociado. De manera más general:

## Cálculo de salidas de las capas para un ejemplo.

Dada la siguiente imagen:

Diagrama

Descripción generada automáticamente

Podremos apreciar que las salidas de las neuronas de la capa oculta vienen dadas por:

En este contexto, representa el peso asociado con la capa ‘i’ y la neurona ‘j’, donde la numeración de las neuronas se realiza de arriba hacia abajo. Adicionalmente, cada neurona posee un término de sesgo correspondiente, denotado como . Que es especifico de la neurona ‘j’ en la secuencia vertical de la red.

Claramente para una persona este tipo de notación matemática se vuelve muy compleja de leer y de poder seguir si vamos añadiendo capas o complejizamos una capa, es por ello, que toda esta información se compacta siguiendo una notación matricial y vectorial de la siguiente manera:

En este caso tenemos que , denota el vector de pesos transpuesto de la capa ‘i’, y la neurona ‘j’, contando de arriba hacia abajo, denota el vector de características y denota el bias de la capa ‘i’ y la neurona ‘j’.

A nivel de programación, normalmente el cálculo de salida de una neurona o estado de activación de una neurona vendrá dado por dos pasos:

* Cálculo de regresión lineal:
* Aplicación de función de activación sobre el cálculo previo:

Diagrama, Diagrama de Venn

Descripción generada automáticamente

Es decir, que en este ejemplo especifico tendremos casi 8 instrucciones:

|  |  |
| --- | --- |
| **Nº instrucción** | **Operación** |
| 1 |  |
| 2 |  |
| 3 |  |
| 4 |  |
| 5 |  |
| 6 |  |
| 7 |  |
| 8 |  |

Todo esto complejiza mucho el problema, porque tendríamos que usar un bucle for, donde en cada vuelta ejecutamos los pasos de dos en dos, en una red neuronal de este tamaño es asumible, pero normalmente, las redes neuronales tienen miles de neuronas que implicarían grandes costes computaciones de cálculo, es por esto, que **nuestro objetivo es vectorizar el problema**, aprovechándonos de las propiedades de las matrices.

De manera aún más compacta y resumida tendremos que la anterior formula viene determinada por:

Finalmente tendremos que:

El motivo por el que hemos separado en dos pasos el calculo de la salida de una capa de una red neuronal es debido a que la función de activación que aplicaremos puede cambiar, es decir, no siempre será la función sigmoide, sino que habrá otras alternativas como la función tangente o ReLu, entre otras muchas que veremos más adelante.

## Calculo para “m” ejemplos.

Observando la siguiente imagen:

Diagrama

Descripción generada automáticamente

Tenemos las siguientes formulas:

Señalando lo siguiente:

* Para el cálculo de la salida de la capa 2, tendremos una matriz de peso W con dimensiones (1,4) dado que tenemos una única neurona que recibe 4 entradas (características) y que tendrá dimensión (1,1), dado que solo tenemos una neurona.
* Este caso es para un único ejemplo.

Nuestro objetivo ahora es preparar nuestra red neuronal para que pueda ser entrenada con múltiples ejemplos de manera eficiente, es decir, de manera vectorizada, para ello, debemos denotar de alguna manera a que ejemplo nos referimos, para ello, utilizamos parentensis redondeados, es decir, que nuestras anteriores formulas quedaría como sigue:

Tal que “i” denota el ejemplo “i-esimo” con el que se entrenará nuestra red neuronal. Entonces, si tenemos “m” ejemplos tendríamos que hacer algo así:

For i = 1 to m:

Y esto lo consideramos ineficiente, es por ello que trabajaremos en lugar de con un vector “x” con una matriz “X” que tendrá todos los ejemplos del conjunto de entrenamiento, y además, en lugar de decir que tenemos un estado de activación “a” para un determinado ejemplo, pondremos en mayúsculas la “a” para denotar el estado de activación para todos los ejemplos, es decir, nuestra notación pasaría a ser:

Tal que viene definido por:

Y que y vienen definidas por:

Podemos entender que las salidas de las regresiones lineales y los estados de activaciones están organizadas horizontalmente para indicarnos en que ejemplo nos encontramos y verticalmente para decirnos en que “nodo” o “neurona especifica” nos encontramos, por poner un ejemplo, tendremos que:

* : El estado de activación o salida generada por la neurona número 2, en la capa 1, de con el ejemplo de entrenamiento 4.
* : El estado de activación o salida generada por la neurona número 4, en la capa 2, de con el ejemplo de entrenamiento 4.

## Descenso de gradiente en una red neuronal de dos capas.

Diagrama

Descripción generada automáticamente

Para aplicar el algoritmo de descenso de gradiente de la anterior imagen, tenemos las siguientes partes:

* **Parámetros**: En este caso nuestros parámetros son:
  + con dimensión de (4,3) es decir, **4** filas por cada nodo de la capa uno y **3** columnas, porque cada nodo tendrá como máximo **3** entradas (hay 3 características x1, x2 y x3).
  + con dimensión de (4,1) dado que habrá **1** bías por cada neurona de la primera capa, que son **4**
  + con dimensión (1,4) porque solo existe 1 nodo en la segunda capa y existen 4 entradas que vienen de la anterior capa.
  + con dimensión (1,1) porque solo existe 1 neurona en la capa 2.
* **Función de coste**:
* **Inicialización de pesos y bias:** Se suele inicializar a un numero aleatorio pequeño en lugar de a cero. lo cual es una práctica estándar para evitar simetría en el aprendizaje y permitir un rompimiento de simetría que facilita el aprendizaje.
* **Repetir el siguiente seudocódigo tantas iteraciones como se indiquen:**
  + **Foward propagation**: Calcular predicciones, tal y como vimos en los anteriores epígrafes quedaría como sigue:
  + Backpropagation:
  + Actualizar pesos y bias según un learning rate y las derivadas de los pasos anteriores.
  + Realizamos lo mismo para

## Inicialización de pesos y bias a números aleatorios.

Inicializar los pesos y sesgos de una red neuronal con números pequeños y aleatorios es una práctica importante por varias razones:

* **Romper la Simetría**: Si todos los pesos se inicializan con el mismo valor, como ceros, entonces todas las neuronas en una capa aprenderán de la misma manera durante el entrenamiento. Esto significa que no importa cuántas neuronas tenga la capa, todas se comportarán de manera idéntica y la capa en su conjunto actuará efectivamente como una sola neurona. La inicialización aleatoria rompe esta simetría, permitiendo que las neuronas aprendan diferentes características de los datos.
* **Mejorar la Eficiencia del Aprendizaje:** Los números aleatorios pequeños ayudan a que el algoritmo de aprendizaje (descenso de gradiente) funcione mejor. Si los pesos son demasiado grandes o pequeños, los gradientes (esencialmente la información utilizada para actualizar los pesos) pueden ser demasiado grandes o pequeños, llevando a problemas como la explosión o desvanecimiento de gradientes, donde los pesos se actualizan de manera ineficiente y el aprendizaje se estanca o es inestable.
* **Fomentar la Diversidad en la Red:** La inicialización aleatoria asegura que cada neurona en la red tiene una perspectiva inicial única sobre el problema, lo que conduce a una diversidad de características aprendidas y a una red más poderosa y capaz de generalizar bien.
* **Evitar Puntos de Saturación:** En el caso de funciones de activación no lineales como la sigmoide o la tanh, si los pesos se inicializan con valores muy grandes, las neuronas pueden comenzar en un estado de saturación, donde cambios en la entrada tienen poco o ningún efecto en la salida de la función de activación. Esto puede ralentizar el aprendizaje o atrapar la red en un estado subóptimo.

## Implementación de algoritmo de descenso de gradiente en una red neuronal de dos capas.

# Red neuronal profunda.

## Introducción.

Tanto en el caso de

## Principales elementos de una red neuronal profunda.

Las redes neuronales profundas son aquellas que tienen varias capas ocultas, veamos una imagen de ejemplo y posteriormente analicemos sus diferentes partes:

Diagrama

Descripción generada automáticamente

Veamos nuevas notaciones genéricas para redes neuronales profundas:

* Con “L” determinamos el número de capas de una red neuronal, es este caso son 4,
* Utilizaremos para indicar el número de neuronas de un determinado nivel L, por ejemplo:
* Con determinaremos el estado activación de la capa L, es decir, será un vector columna, de dimensión ( donde cada elemento representará la salida de una neurona tras haber hecho propagación hacia delante. Tengamos en cuenta también que:
* : Denotan los pesos y el bias del nivel L respectivamente.
* denotará la capa de entrada o lo que es lo mismo, el input de la red neuronal, es decir, que , lo mismo sucede con la última capa , en este caso, L será el valor numérico de la ultima capa o el tamaño de la red neuronal. Para nuestro ejemplo L = 4.

## Propagación hacia delante.

Diagrama

Descripción generada automáticamente

Volviendo a la anterior red neuronal anterior, la salida de activación de la capa 1 implicaría los siguientes cálculos:

Claramente vemos que estas 8 instrucciones siguen un patrón cíclico, por lo que podemos abstraer la siguiente fórmula para cualquier nivel, teniendo que:

Además, si lo vectorizamos, tendremos que:

Donde lo único que hemos hecho es poner en mayúscula la A y la Z para indicar que se tratan de varios ejemplos, es decir que:

Nota: Aunque nuestra filosofía siempre será eliminar bucles “for” siempre que podamos, para el foward propagation tendremos que iterar sobre un bucle for para calcular cada una de las salidas, esto se debe a que la salida de una capa depende de la salida de la anterior capa, esto define un problema de naturaleza secuencial (uno detrás de otro) y no de una manera paralela (todos a la vez).

## Dimensiones de redes neuronales.

Diagrama

Descripción generada automáticamente

Observando el anterior ejemplo de red neuronal, si vamos a estudiar las dimensiones de cada capa tendremos lo siguiente:

**Nivel 1**

En el caso del nivel 1, tenemos que el vector de pesos tendrá dimensiones (3,2) dado que hay 3 neuronas (3 filas) y 2 características de entradas (2 columnas) que provienen de la capa cero o capa de entrada, además habrá un bias que será un vector de (3,1) dado que hay 3 neuronas.

La operación intermedia antes de aplicar la función de activación tendrá una dimensión propia de la multiplicación de los pesos con la salida de la capa previa, en este caso, como estamos en el nivel 1, es salida de la capa previa corresponde con la dimensión de la red neuronal, es decir que tendremos que al multiplica la matriz de pesos con dimensión (3,2) contra el vector entrada (2,1), nos dará como resultado un con dimensiones (3,1).

**Nivel 2**

En el caso del nivel 2, tenemos que el vector de pesos tendrá dimensiones (5,3) dado que hay 5 neuronas (5 filas) y 3 entradas de la capa anterior (3 columnas), además habrá un bias que será un vector de (5,1) dado que hay 5 neuronas.

La operación intermedia antes de aplicar la función de activación tendrá una dimensión propia de la multiplicación de los pesos con la salida de la capa previa, en este caso, como estamos en el nivel 2, es salida de la capa previa corresponde con la dimensión de la red neuronal, es decir que tendremos que al multiplica la matriz de pesos con dimensión (5,3) contra el vector entrada (3,1), nos dará como resultado un con dimensiones (5,1).

**Caso genérico**

El caso genérico vendrá a decirnos que:

* La dimensión de será .
* La dimensión del bias será .
* Tendrá una dimensión de donde “m” es el número de ejemplos.
* tendrá dimensiones resultado de la multiplicación de los pesos con las salidas de la capa anterior, es decir que será
* y tendrán las mismas dimensiones.

Adicionalmente, las derivadas tendrán las siguientes dimensiones:

## Redes neuronales profundas.

Las redes neuronales profundas se han convertido en herramientas fundamentales para resolver una amplia gama de problemas complejos debido a su capacidad para construir representaciones jerárquicas de los datos. La profundidad de estas redes, es decir, la cantidad de capas ocultas que poseen es un factor crucial que contribuye a su eficacia. A continuación, se ofrece un resumen detallado en español de los puntos clave mencionados en el texto original, con una mejora en la claridad y la inclusión de información relevante.

¿Qué calculan las redes profundas?

En el contexto de la detección o reconocimiento facial, una red neuronal profunda inicia su proceso con la detección de características simples en la primera capa, como los bordes en una imagen, utilizando unidades ocultas que funcionan como detectores de características. Estas unidades se representan visualmente como pequeñas cajas que buscan patrones específicos en la imagen, como bordes verticales u horizontales.

A medida que la información avanza a través de las capas de la red, las representaciones se vuelven más complejas. Por ejemplo, la detección de bordes permite identificar partes específicas de un rostro, como los ojos o la nariz. Finalmente, la combinación de estas partes facilita el reconocimiento de rostros completos. Este proceso ilustra cómo las capas tempranas de la red se enfocan en funciones simples, que se componen en capas posteriores para aprender funciones más complejas. Este enfoque jerárquico también se aplica en otros dominios, como el reconocimiento de voz, donde la red puede aprender desde características básicas del sonido hasta unidades más complejas de lenguaje, como fonemas, palabras y frases.

Intuición sobre la eficacia de las redes profundas

Una analogía útil proviene de la teoría de circuitos, donde se observa que ciertas funciones matemáticas se pueden calcular de manera más eficiente con redes profundas que con redes poco profundas. Por ejemplo, la función de exclusión o "XOR" de un conjunto de características de entrada puede implementarse de manera más compacta y eficiente en una red profunda. En una red poco profunda, para lograr el mismo cálculo, se requeriría una cantidad exponencialmente mayor de unidades ocultas, lo que ilustra la capacidad de las redes profundas para representar funciones complejas de manera más eficiente.

Perspectiva histórica y tendencias

Aunque la noción de "aprendizaje profundo" ha ganado popularidad en parte debido a su atractivo como término de marketing, la eficacia de las redes neuronales profundas es bien fundamentada. Históricamente, se ha observado una tendencia hacia el uso de redes cada vez más profundas para ciertas aplicaciones, encontrando que modelos con docenas de capas pueden ser óptimos para problemas específicos. Sin embargo, es importante comenzar con modelos más simples y aumentar la complejidad gradualmente, ajustando la profundidad de la red como un hiperparámetro en función del problema específico.

En resumen, las redes neuronales profundas ofrecen una metodología poderosa para construir representaciones complejas y jerárquicas de los datos, permitiendo abordar tareas que van desde el reconocimiento de imágenes hasta el procesamiento del lenguaje natural. La profundidad de estas redes es clave para su capacidad de aprender y modelar la complejidad inherente a los datos y las tareas a las que se enfrentan.

## Flujo básico de una red neuronal.

Imagen de la pantalla de un celular con la imagen de una caricatura

Descripción generada automáticamente con confianza baja

El diagrama proporciona una representación visual del flujo de operaciones en una red neuronal durante el proceso de aprendizaje, específicamente durante el algoritmo de descenso de gradiente.

**Entrada y Propagación Hacia Adelante**

* La red neuronal comienza con la entrada de datos (denotada por 'x').
* Estos datos pasan a través de múltiples capas (denotadas por 'a'), donde cada capa transforma la entrada mediante una combinación de pesos ('w') y sesgos ('b'). Estos pesos y sesgos son ajustables y se modifican durante el entrenamiento.
* Cada capa produce una salida que se convierte en la entrada para la siguiente capa. Este proceso continúa hasta que se alcanza la capa de salida.
* Además de la salida también produce información que se guarda en caché como el vector de pesos, el bias y la salida previa, esto se usará luego en el backpropagation de esa misma capa.
* Dentro de cada capa, después de combinar la entrada con los pesos y sesgos, se aplica una función de activación (no mostrada explícitamente en la imagen). Esta función introduce no linealidades en el modelo, lo que permite que la red capture patrones complejos.

**Cálculo del error**

* Una vez que los datos han pasado a través de todas las capas y se ha obtenido una salida final, se calcula un error (denotado por 'L' de Loss en la imagen), que es la diferencia entre la predicción de la red y el valor real o esperado.

**Retroprapagación y Descenso del Gradiente**

* Con el objetivo de mejorar las predicciones, la red necesita ajustar sus pesos y sesgos. Para hacerlo, utiliza un proceso llamado retropropagación.
* La retro propagación calcula el gradiente del error con respecto a cada peso y sesgo, indicando cómo deben ajustarse para reducir el error. Esto se representa en la imagen con términos como 'dL/w' o 'dL/b', que son las derivadas parciales del error respecto a los pesos y sesgos.
* Estos gradientes se utilizan luego para actualizar los pesos y sesgos en la dirección que minimiza el error. Este proceso se realiza repetidamente y es conocido como descenso del gradiente.

El proceso de propagación hacia adelante y retropropagación se repite muchas veces, con cada iteración ajustando los pesos y sesgos para mejorar gradualmente la precisión de la red.

Con suficientes iteraciones y datos de entrenamiento adecuados, la red neuronal "aprende" y se vuelve mejor en la tarea específica que se le ha asignado, ya sea reconocer rostros, entender el habla, o cualquier otro problema complejo.

## Propagación hacia delante y hacia atrás.

En primer lugar, cada capa recibirá como entrada el estado de activación de la capa anterior y genera la salida de la capa actual y cacheará el resultado de la regresión lineal sin haberla pasado por la función de activación con el objetivo de utilizarla más adelante en la propagación hacia atrás. Recordemos las operaciones, las cuales quedan como siguen:

En el caso de la propagación hacia atrás tendremos:

## Parámetros e hiperparámetros.

* **Parámetros:** Son las variables que el modelo utiliza para hacer predicciones y que se ajustan automáticamente durante el proceso de entrenamiento. En una red neuronal, los parámetros típicos incluyen:
  + **Pesos (Weights):** Valores que se multiplican con las entradas para determinar la importancia de cada característica.
  + **Sesgos (Biases):** Valores que se suman a la entrada ponderada antes de pasarla a través de la función de activación para permitir que la red ajuste la salida.

Estos parámetros son ajustados mediante el entrenamiento para minimizar una función de pérdida, que mide qué tan bien el modelo está realizando su tarea.

* **Hiperparámetros**: Son configuraciones que gobiernan cómo se entrena el modelo y que deben ser establecidas antes de iniciar el proceso de entrenamiento. No se aprenden a partir de los datos durante el entrenamiento, sino que son establecidos por el programador. Incluyen:
  + **Tasa de Aprendizaje (Learning Rate):** Determina cuánto se ajustan los parámetros en cada paso del proceso de optimización.
  + **Número de iteraciones:** ¿Cuántas veces quieres que se actualicen los pesos y el bias?
  + **Número de Capas y Número de Unidades en cada Capa:** La arquitectura de la red, incluyendo cuántas capas ocultas hay y cuántas unidades tiene cada una.
  + **Elección de función de activación**

Nota: Estos son solo lo más básicos, mas adelante se verán otros hiperparámetros más avanzados.

## Implementación en Python de algoritmo de descenso de gradiente en redes neuronales multicapa.

La inicialización adecuada de los parámetros, incluyendo los pesos W y los sesgos b, es un paso crítico antes de comenzar el entrenamiento de una red neuronal profunda. Aunque podría parecer razonable iniciar todos los parámetros en cero, esto no es recomendable en redes con múltiples capas. El ideal **es inicializar los pesos con valores aleatorios pequeños**. Esto es esencial para evitar un problema conocido como "simetría de pesos", donde neuronas en la misma capa aprenden de forma idéntica, lo que reduce la eficiencia del aprendizaje.El código a continuación muestra una función que inicializa los parámetros de una red neuronal profunda en Python:

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

Diagrama

Descripción generada automáticamenteEn este caso podemos apreciar que recibimos un “array” denominado layer\_dims, este arreglo tendrá las dimensiones de las diferentes capas de la red neuronal, es decir, el número de neuronas que hay en cada capa de izquierda a derecha, por ejemplo, para la siguiente imagen:

Tendremos que layer\_dims = [3,5,5,3,1]. Una vez entendido esto se crea un diccionario denominado parameters que contendrá todas las matrices de pesos y los vectores de bias inicializados e indexados por nombre, tal que Wi, indicará, la matriz de pesos de la capa “i” y lo mismo pasará con bi, que indicara el vector columna bias de la capa i.

Tras esto para cada una de las capas creamos una matriz de pesos, las cuales se **inicializarán a números aleatorios pequeños**, en el caso del vector bias se inicializará a cero y ya toda esta información se almacenará en nuestro diccionario, el cual será devuelto en el lugar donde se invoque está función.

Tras esto nuestro paso será implementar la propagación hacia delante

Captura de pantalla de computadora

Descripción generada automáticamente

Como podemos apreciar, este código implementa la propagación hacia delante a lo largo de todas las capas, en la arquitectura de nuestra red neuronal hemos decidido que todas las capas excepto la capa de salida tenga una función de activación ReLu, en la última capa o capa de salida utilizaremos la función sigmoide.

Texto

Descripción generada automáticamente

Además, las cachés en redes neuronales se utilizan para guardar ciertos valores que se calculan durante la propagación hacia adelante (**forward propagation**) de la red. Estos valores se utilizan más tarde durante la retro propagación (**backpropagation**), que es el proceso por el cual la red aprende ajustando los pesos y sesgos en base al error de la salida. En el código almacenamos dos tipos de caché:

* **Linear\_cache:** Que guarda los valores A, W y b después de realizar la operación, con el objetivo de utilizarlos más adelante en la retro propagación (backpropagation) para el caculo de los gradientes de los pesos y sesgos.
* **Activation\_cache**: Guarda el valor Z, que es el resultado de la operación lineal antes de aplicar la función de activación. Necesario para calcular el gradiente de la función de activación durante la retropropagación.

Cuando se ejecuta la función linear\_activation\_forward, combinas estos dos cachés en una tupla (cache = (linear\_cache, activation\_cache)) para cada capa de la red. Esto se hace para mantener organizada la información necesaria para la retropropagación. Al final de la propagación hacia adelante, tienes una lista de cachés, cada una correspondiendo a una capa de la red.

Texto

Descripción generada automáticamente

# Aspectos prácticos del machine learning.

## Naturaleza iterativa

La naturaleza del machine learning aplicado tiene un enfoque muy experimental e iterativo, con iterativo queremos decir que seguimos un ciclo que se repite hasta alcanzar los resultados deseados.

En este proceso iterativo nuestro trabajo consiste en decidir cuantas capas tiene nuestra red neuronal, cuantas neuronas tiene cada capa, ajustar la tasa de aprendizaje, decidir que funciones de activación utilizar, etc. En este proceso, la red neuronal entrenada nos puede dar buenos resultados o malos resultados, de ahí viene la palabra “experimental”, no tiene un enfoque tan determinista en el que decimos “si hago esto”, “pasara esto otro”, sino que es más impredecible y probabilístico, haciendo que sea más de probar ciertas configuraciones y ver si da resultados mejores o peores. El diagrama que define el flujo básico es el siguiente:

Imagen que contiene Diagrama

Descripción generada automáticamente

Donde tenemos una idea, la ponemos en práctica implementándola en código y experimentamos viendo resultados, que nos pueden dar otras ideas y así sucesivamente.

## División del conjunto de datos de entrenamiento.

Normalmente los datos se dividirán entres conjuntos:

* **Conjunto de entrenamiento (60%-70% de los datos totales):** Se utiliza para entrenar el modelo de aprendizaje automático. Durante el proceso de entrenamiento, el modelo aprende a reconocer patrones, características y asociaciones en estos datos, ajustando sus parámetros internos para minimizar el error entre las predicciones y los resultados reales.
* **Conjunto de desarrollo (15%-20% de los datos totales) también denominado conjunto test:** También conocido como el conjunto de validación, se utiliza para afinar y ajustar los hiperparámetros del modelo y para tomar decisiones sobre la arquitectura del modelo y otras variables de diseño. Permite evaluar el rendimiento del modelo en datos que no ha visto durante el entrenamiento, pero sin utilizar el conjunto de prueba final. Si tenemos “n” algoritmos o modelos diferentes, utilizamos este conjunto de desarrollo o validación para ver cual se acerca mejor a la solución a nuestro problema. Permite realizar iteraciones rápidas en el diseño del modelo, proporcionando una retroalimentación inmediata sobre cómo los cambios en los hiperparámetros o en la estructura del modelo afectan su rendimiento.
* **Conjunto de prueba (15%-20% de los datos totales):** Debe ser representativo de los datos reales sobre los que se espera que el modelo opere y, a menudo, se extrae de la misma distribución que el conjunto de entrenamiento. No se utiliza para el entrenamiento directo, pero sí para validar la eficacia del modelo y para evitar el sobreajuste.

Sin embargo, cabe señalar que cuando decidimos que una parte de los datos sean para evaluar el modelo o para el conjunto de desarrollo, también estamos decidiendo no usarlos para el conjunto de entrenamiento, lo cual, puede derivar en la perdida del uso de datos valiosos que podrían ayudar a entrenar la red neuronal, es por ello, que siempre que se pueda se intenta ampliar el conjunto de entrenamiento.

Normalmente, cuanto más grande es el conjunto de datos inicial, más podemos reducir el porcentaje de datos que se usaran como desarrollo o como prueba.

Otro aspecto muy importante es que tanto el **conjunto de entrenamiento como el conjunto de prueba tenga la misma distribución de representación de los datos**, por ejemplo, si tenemos un problema de machine learning que busca clasificar pasos de una receta en tres posibles clases (preparando ingredientes, cocinando y presentación), si en el conjunto de entrenamiento tenemos que el 33% de los datos son preparando, un 40% son cocinando y un 27% son presentación, entonces buscaremos esa misma proporción en el conjunto de prueba o test.

Esta congruencia entre los conjuntos asegura que el modelo esté expuesto a una representación equitativa de cada clase durante el entrenamiento y que su capacidad de generalización sea evaluada de manera justa durante la prueba, evitando así sesgos que podrían comprometer la precisión y la fiabilidad de las predicciones del modelo en escenarios reales.

## Bias y varianza.

En el aprendizaje automático, el sesgo se refiere a un error sistemático debido a suposiciones simplistas en el modelo. Un modelo con alto sesgo no captura adecuadamente la complejidad de los datos, llevando a predicciones que están consistentemente desviadas de los valores reales. Un sesgo elevado puede ser indicativo de que el modelo es demasiado simple, conocido como subajuste o underfitting, lo que implica que no logra aprender bien la estructura subyacente de los datos y, por ende, suele tener un rendimiento predictivo pobre.

El underfitting puede compararse con generalizaciones excesivamente simplistas en el análisis sociológico. Por ejemplo, aseverar que "todos los alemanes son hiperproductivos mientras que todos los españoles no lo son", constituye una generalización que ignora las variaciones individuales y las excepciones, perdiendo así la riqueza y la precisión en la descripción de la realidad.

En contraste, la varianza se refiere a la sensibilidad de un modelo a las fluctuaciones específicas en el conjunto de entrenamiento. Un modelo con alta varianza es aquel que se ajusta demasiado a los datos específicos con los que ha sido entrenado, incluyendo el ruido y las anomalías, lo que puede resultar en un sobreajuste o overfitting. Esto significa que, aunque el modelo puede tener un rendimiento excelente con los datos de entrenamiento, su capacidad para generalizar bien a nuevos datos es limitada.

Diagrama

Descripción generada automáticamente

En el gráfico, se ilustra cómo un modelo con underfitting no sigue el verdadero patrón de los datos (izquierda), mientras que un modelo con overfitting sigue los datos de entrenamiento demasiado de cerca, capturando incluso el ruido (derecha). Idealmente, buscamos un punto medio donde el modelo es lo suficientemente complejo para capturar la verdadera relación subyacente en los datos, pero no tan complejo que no pueda generalizar bien a nuevos ejemplos.

La clave es encontrar un equilibrio entre sesgo y varianza para lograr un modelo que no solo se ajuste bien a los datos conocidos, sino que también sea capaz de realizar predicciones acertadas con datos nuevos. Este equilibrio óptimo permitirá que el modelo sea robusto frente a diferentes conjuntos de datos y situaciones reales.

## Enfoque sistemático de acción según Bias y varianza.

Estudiaremos diferentes casos que nos darán pistas de como actuar en los siguientes ciclos iterativos de la obtención del modelo ideal para nuestro problema:

* Si apreciamos un **Bias alto**, esto indica que nuestro modelo simplifica demasiado la realidad, una manera de solventarlo es **hacer la arquitectura de nuestra red neuronal más compleja**, para que capte más detalles de los datos que está estudiando, es decir, añadir más capas y más neuronas, permitiendo así que se aprenda de manera “más profunda” de la realidad objeto de estudio.

Además**, ampliar el conjunto de datos de entrenamiento** puede ser beneficioso, ya que proporciona a la red neuronal una base de conocimiento más amplia y diversa, mejorando su capacidad para generalizar a partir de los ejemplos proporcionados. Esta estrategia puede ayudar a minimizar el sesgo al asegurarse de que el modelo tenga acceso a una representación más completa y variada de los datos.

* Cuando nos enfrentamos a **una alta varianza**, el modelo puede estar sobre ajustándose a los datos de entrenamiento, capturando ruido en lugar de las relaciones subyacentes genuinas. Para mitigar este problema, una de las técnicas más efectivas es la **inclusión de más datos de entrenamiento**, si están disponibles. Esta adición puede ayudar al modelo a aprender las tendencias generales en lugar de sobre ajustarse a las particularidades o al ruido en el conjunto de datos original.

Otra solución es la aplicación de técnicas de regularización. La regularización ayuda a prevenir el sobreajuste al penalizar los pesos de la red neuronal que se vuelven demasiado grandes, fomentando así un modelo más simple y generalizable. Existen diferentes métodos de regularización, como L1, L2, y dropout, cada uno con sus propias características y aplicaciones, los cuales exploraremos en detalle en secciones posteriores.

# Técnicas de regularización.

## Regularización con la regresión Ridge (Penalización L2).

La regresión de Ridge es una técnica usada en estadística para analizar datos que sufren de ciertos problemas que la regresión lineal no puede manejar bien. Imagina que estás intentando predecir el precio de una casa basándote en diferentes características como el tamaño de la casa, el número de habitaciones, y la edad de la casa. Si el tamaño de la casa y el número de habitaciones están muy relacionados (porque normalmente las casas grandes tienen muchas habitaciones), puede ser complicado para un modelo de regresión lineal estándar entender cómo cada una de estas características influye por separado en el precio.

La regresión de Ridge ayuda a resolver este problema añadiendo un tipo de "costo" o "penalización" por tener coeficientes (que son los pesos que multiplican a cada característica) muy grandes. Esto es Porque pesos grandes a menudo significan que la red **está poniendo demasiada atención a detalles muy específicos de los datos de entrenamiento.** el modelo podría empezar a pensar que esa característica es lo único importante, ignorando las demás.

Para poner un ejemplo sencillo, digamos que queremos predecir las calificaciones finales de estudiantes basándonos en el tiempo de estudio y el número de tareas entregadas. Si estos dos están muy relacionados (porque los estudiantes que estudian más también tienden a entregar más tareas), la regresión de Ridge ayudaría a asegurarse de que nuestro modelo no dependa solamente de una de estas características, sino que use ambas de una manera equilibrada para hacer una predicción.

La penalización de Ridge obliga al modelo a no depender demasiado de ninguna característica individual, balanceando la influencia de todas ellas. El resultado es un modelo que, aunque pueda no ajustarse perfectamente a los datos de entrenamiento, debería ser mejor al hacer predicciones sobre datos nuevos y desconocidos.

Tal que “m” indica el número de ejemplos con lo que la red neuronal fue entrenada, L es el número de capas de la red neuronal.

Una manera de entender esto es que, si nuestro objetivo es optimizar la función de coste, en esta nueva versión, **tendremos en cuenta en esta optimización la suma de todos los pesos de la red neuronal**, por lo que, si nuestro objetivo es minimizar la función de coste, también ese proceso de minimización buscaremos reducir lo máximo posible el tamaño de los pesos, evitando sobre ajustes a ciertos datos y dando mayor generalización.

Además, el parámetro controla la intensidad de esta penalización: un más grande implica una penalización más fuerte, y un α de 0 reduciría la regresión Ridge a la regresión lineal ordinaria, dado que el sumando derecho de la nueva función de coste se anulará. El valor por defecto de es 1.5

En esencia, la regresión Ridge busca **encontrar un equilibrio entre la adaptación fiel a los datos y la simplicidad del modelo**, mejorando así su capacidad de generalización. Además, cabe señalar que esta regresión tiene un doble objetivo:

* Ajustarse bien a los datos (minimizando el error residual)
* Mantener los coeficientes relativamente pequeños (minimizando la penalización).

La penalización evita que los coeficientes alcancen valores excesivamente grandes, lo cual puede ser un signo de sobreajuste o de inestabilidad debido a problemas como la multicolinealidad (que unas características o variables independientes dependan de otras)

Interfaz de usuario gráfica, Gráfico, Gráfico de líneas

Descripción generada automáticamente

Una de las características más interesantes que se aprecian en la gráfica es que los coeficientes nunca llegan a ser cero, por lo cual, no se eliminan características.

Los cambios significativos sobre el algoritmo de descenso de gradiente clásico tras aplicar la regularización en L2, son los que siguen:

Aunque los tres sumandos pueden ser confusos lo que viene a decir:

* Se elevan al cuadrado todos los elementos de la matriz de pesos de cada capa . Esto se hace para cada peso individualmente, resultando en una matriz del mismo tamaño donde cada elemento es el cuadrado del correspondiente peso.
* Luego, se suma el valor de todos los elementos cuadrados dentro de esta matriz. En términos matemáticos, esto se denomina la norma de Frobenius de la matriz , representada por
* Finalmente, se acumulan estas sumas de todos los elementos cuadrados para todas las capas l de la red neuronal.

El objetivo es minimizar ese valor, por lo que en la función de coste se buscaría minimizar todos los pesos.

Tras esto, la actualización de los pesos quedaría como sigue:

Nuevamente, esto derivaría en que ahora se suavizará y armonizarán los valores de los pesos haciendo que ningún peso adquiera demasiada importancia y así evitando el sobreajuste. Con este nuevo enfoque tenemos un nuevo hiperparámetros que indicará con que fuerza aplicamos la regresión Ridge y es nuestro trabajo saber cual es el valor idóneo de esta variable para nuestro caso de uso especifico.

## Regularización de Ridge en Python.

Visto la parte teórica de la regularización L2 podemos ver una implementación en código python:

Texto

Descripción generada automáticamente

Si apreciamos la siguiente formula ya vista en el epígrafe anterior:

Podemos concluir en el código Python que la hemos dividido en dos partes, la primera parte la entropía cruzada y la segunda parte es la nueva parte que se añade relacionada con la regularización L2, que no es más que la implementación matemática de la fórmula matemática expuesta.

A modo de síntesis hemos añadido como factor determinante de la función de coste la suma del cuadrado de todos los pesos de toda la red neuronal, por lo que buscaremos minimizar esa función de coste y por ende el valor de los pesos, evitando que ningún peso se desmesure hacia cantidades muy grandes.

La segunda parte de la aplicación de la regularización L2 es la actualización de cálculos de gradientes para tener en cuenta que la nueva función de coste tiene ahora el objetivo de minimizar los pesos también, para el caso de una red neuronal de 3 capas tenemos lo siguiente:

Pantalla de computadora con letras

Descripción generada automáticamente con confianza media

## Regularización “Drop-out”

La regularización "Dropout" es una técnica muy eficaz para prevenir el sobreajuste en redes neuronales. La idea detrás de Dropout es sorprendentemente simple y se asemeja a un tipo de entrenamiento robusto en el mundo real.

Imagina que tienes un equipo de trabajadores en una empresa. Si solo algunos de ellos son los que siempre resuelven los problemas más complicados, el equipo en general se vuelve muy dependiente de esos pocos trabajadores. Si un día faltan, el equipo podría tener problemas para funcionar correctamente porque no han practicado lo suficiente con los desafíos. En el contexto de una red neuronal, esto es análogo a tener ciertas neuronas (o nodos) que se vuelven "especialistas" en la tarea y activan muy fuertemente, mientras que otras contribuyen poco al resultado final.

Dropout soluciona esto de una manera bastante ingeniosa: durante el entrenamiento, en cada paso (o iteración**), se "apaga" aleatoriamente** un porcentaje de neuronas en la red. Esto significa que, en cada paso de entrenamiento, una serie aleatoria de neuronas se ignora o se "descarta". Esto fuerza a que otras neuronas "intervengan" y aprendan a manejar la parte de la tarea que normalmente hubieran dejado a las neuronas ahora descartadas. Como resultado, la red se vuelve menos sensible a los pesos específicos de las neuronas, y esto conduce a un modelo más robusto que generaliza mejor cuando se expone a nuevos datos.

Volviendo al ejemplo de los trabajadores, sería como si cada día, algunos trabajadores se toman el día libre de manera aleatoria, obligando a los demás a aprender diferentes partes del proceso de trabajo. Con el tiempo, el equipo se vuelve más flexible y menos dependiente de cualquier trabajador individual.

La belleza de Dropout radica en su simplicidad y efectividad. Se ha demostrado que mejora el rendimiento de las redes neuronales en una amplia gama de tareas y es una de las técnicas estándar en el arsenal de herramientas de un científico de datos para regularizar redes neuronales.

Hay muchas maneras de implementar Dropout, la que veremos aquí será la de “inverter Drop-out”, en está técnica, se siguen los siguientes pasos:

1. Se genera una matriz de Dropout: Que consiste en una matriz aleatoria del mismo tamaño que la activación de una determinada capa L. Cada elemento de esta matriz es el resultado de una comparación entre un número aleatorio y el keep\_prob, que es la probabilidad de mantener una neurona activa. Si el número aleatorio es menor que keep\_prob, el elemento correspondiente en la matriz es True (o 1), lo que significa que la neurona se mantendrá; de lo contrario, es False (o 0), y la neurona se "apagará" o se descartará.
2. A continuación, se hará una multiplicación elemento a elemento entre la matriz de activación del nivel L y la matriz de Dropout del nivel L, poniendo a cero las neuronas que han sido descartadas.
3. Tras esto, se escalan las activaciones que no fueron puestas a cero dividiéndolas por “keep\_prob”, este paso es el que diference el inverted Dropout del “Dropout tradicional”.

Por poner un ejemplo, si se tienen 50 unidades (neuronas) y keep\_prob es 0.8, entonces en promedio se "apagarían" 10 unidades y el resto se escalaría para compensar esta desactivación.

Este método ayuda a prevenir el sobreajuste ya que cada neurona no puede confiar en la presencia de las otras; debe aprender a ser útil en conjunto con muchas diferentes subredes aleatorias de las neuronas existentes. Esto lleva a que la red desarrolle una representación más robusta de los datos.

En este nuevo contexto, aparece un nuevo hiperparámetro “keep\_prob” que nos dice la probabilidad que hay de que se mantenga activa una neurona. esta probabilidad de conservación puede variar entre las capas de la red. Por ejemplo, en capas con más parámetros, es posible que desees un keep\_prob más bajo para aplicar una regularización más fuerte y prevenir el sobreajuste. En capas con menos riesgo de sobreajuste, podrías usar un keep\_prob más alto o incluso igual a 1, lo que significa que no estás aplicando Dropout en esa capa.

En la práctica, a menudo se aplica Dropout solo a las capas ocultas y no a la capa de entrada. Esto es porque eliminar características de entrada puede no ser deseable, especialmente si no tienes muchas para comenzar. Dropout funciona de manera similar a la regularización L2, en el sentido de que evita que los pesos de las conexiones se vuelvan demasiado grandes. Sin embargo, a diferencia de L2, que aplica la misma penalización a todos los pesos, Dropout adapta su penalización a la escala de las activaciones de las neuronas.

Una desventaja de Dropout es que complica la verificación del descenso del coste durante el entrenamiento, porque al estar desactivando unidades aleatoriamente, la función de coste no es fija y es más difícil de seguir. Una práctica común es desactivar Dropout temporalmente para verificar que la función de coste disminuye de manera consistente antes de volver a activar Dropout para continuar con el entrenamiento.

## Aumentación de datos.

Data augmentation es una técnica de regularización que consiste en aumentar el tamaño y la diversidad del conjunto de entrenamiento mediante la generación de nuevas muestras a partir de las existentes. Esta técnica se basa en la idea de que si la red neuronal aprende a generalizar bien a partir de un conjunto de datos aumentado, también será capaz de generalizar bien a nuevos datos que no haya visto antes.

Existen diversas técnicas de data augmentation que se pueden aplicar a diferentes tipos de datos. Algunas de las técnicas más comunes son:

|  |  |
| --- | --- |
| **Imágenes:**   * Rotaciones * Volteos * Escalas * Recortes * Cambios de brillo y contraste * Adición de ruido | **Texto:**   * Sinónimos * Paraphraseo * Traducción * Adición de ruido |
| **Audio:**   * Cambios de tono * Adición de ruido * Reverberación |  |

Beneficios del data augmentation:

* **Reduce el sobreajuste:** al aumentar el tamaño del conjunto de entrenamiento, la red neuronal tiene menos probabilidades de aprender las características específicas del conjunto de entrenamiento y más probabilidades de aprender las características generales que le permitirán generalizar bien a nuevos datos.
* **Mejora la generalización:** al aumentar la diversidad del conjunto de entrenamiento, la red neuronal se expone a una mayor variedad de datos y, por lo tanto, es más capaz de generalizar a nuevos datos que no haya visto antes.
* **Puede ayudar a encontrar una solución más robusta:** al aumentar el tamaño del conjunto de entrenamiento, la red neuronal es menos sensible a los valores atípicos.

Desventajas del data augmentation:

* Puede aumentar el tiempo de entrenamiento: al aumentar el tamaño del conjunto de entrenamiento, el tiempo de entrenamiento también aumenta.
* Puede ser difícil elegir las técnicas de data augmentation adecuadas: la elección de las técnicas de data augmentation adecuadas depende del tipo de datos y del problema específico.

## Early Stopping.

Early stopping es una técnica de regularización que consiste en detener el entrenamiento de una red neuronal antes de que comience a sobreajustarse. El sobreajuste ocurre cuando una red neuronal aprende demasiado del conjunto de entrenamiento y comienza a memorizar las características específicas de este conjunto, en lugar de aprender las características generales que le permitirían generalizar bien a nuevos datos.

El early stopping funciona monitorizando el rendimiento de la red neuronal en un conjunto de validación. El conjunto de validación es un conjunto de datos independiente del conjunto de entrenamiento que se utiliza para evaluar el rendimiento de la red neuronal durante el entrenamiento.

Si el rendimiento de la red neuronal en el conjunto de validación comienza a disminuir, se detiene el entrenamiento. Esto evita que la red neuronal aprenda las características específicas del conjunto de entrenamiento y le permite generalizar mejor a nuevos datos.

Existen diferentes estrategias para implementar el early stopping:

* **Monitorear la precisión:** Se monitorea la precisión de la red neuronal en el conjunto de validación y se detiene el entrenamiento cuando la precisión comienza a disminuir.
* **Monitorear la pérdida:** Se monitorea la pérdida de la red neuronal en el conjunto de validación y se detiene el entrenamiento cuando la pérdida comienza a aumentar.
* **Utilizar un callback:** Se puede utilizar un callback para detener el entrenamiento automáticamente cuando se cumpla una condición específica.

Beneficios del early stopping:

* **Reduce el sobreajuste:** al detener el entrenamiento antes de que comience el sobreajuste, la red neuronal es menos probable que aprenda las características específicas del conjunto de entrenamiento.
* **Mejora la generalización:** al evitar el sobreajuste, la red neuronal es más capaz de generalizar bien a nuevos datos que no haya visto antes.
* **Puede ahorrar tiempo de entrenamiento:** al detener el entrenamiento antes de que comience el sobreajuste, se puede ahorrar tiempo de entrenamiento.

Desventajas del early stopping:

* **Puede ser difícil elegir el momento adecuado para detener el entrenamiento:** si se detiene el entrenamiento demasiado pronto, la red neuronal puede no haber aprendido lo suficiente. Si se detiene el entrenamiento demasiado tarde, la red neuronal puede comenzar a sobreajustarse.
* **Puede ser sensible al ruido en el conjunto de validación:** si el conjunto de validación es pequeño o contiene ruido, el rendimiento de la red neuronal en el conjunto de validación puede ser variable, lo que puede dificultar la elección del momento adecuado para detener el entrenamiento.

# Técnicas de optimización.

## Normalización de inputs.

La normalización de las entradas en redes neuronales es una técnica para acelerar el entrenamiento y mejorar la convergencia del algoritmo de descenso del gradiente. Esto se logra transformando las características de entrada a un rango similar, generalmente entre 0 y 1 o con media cero y desviación estándar igual a 1.

EL motivo por el que se utiliza está técnica es que mejora el rendimiento del descenso del gradiente: Cuando las características tienen escalas muy diferentes, los pesos y sesgos también necesitan rangos distintos para actualizarse correctamente. La normalización asegura que el algoritmo de descenso del gradiente opere en un espacio con escalas similares, permitiendo pasos más grandes y una convergencia más rápida.

Imagina dos características, una entre 0 y 1000 y otra entre 0 y 1. Sin normalización, el algoritmo priorizará la actualización de la segunda característica debido a su menor escala. La normalización los pone en un rango similar, permitiendo actualizaciones equilibradas para ambas.

La técnica general para normalizar es la que si

1. **Calcular la media (μ):** Suma todos los valores de una característica y divide por el número de entradas.
2. **Calcular la desviación estándar (σ):** Calcula la desviación de cada valor respecto a la media y luego la raíz cuadrada de la media de esos cuadrados.
3. **Restar la media:** Para cada valor, resta la media calculada en el paso 1.
4. **Dividir por la desviación estándar:** Para cada valor, divide por la desviación estándar calculada en el paso 2.

Algunos aspectos que destacar son los que siguen:

* Aplica la misma normalización al conjunto de entrenamiento y prueba.
* Si las escalas son similares, la normalización puede no ser necesaria.
* Considera técnicas alternativas como Min-Max Scaling para rangos específicos.

## Desaparición o explosión de gradientes.

El problema de los gradientes que desaparecen o explotan es un desafío significativo en el entrenamiento de redes neuronales profundas. Este fenómeno ocurre debido a la propagación de gradientes a través de las múltiples capas de la red durante el proceso de retropropagación. Cuando los gradientes son muy pequeños (desvanecimiento) o muy grandes (explosión), dificultan la actualización efectiva de los pesos de la red, lo que puede llevar a una convergencia muy lenta o incluso a la incapacidad de aprender.

Una estrategia clave para mitigar estos problemas es la **inicialización cuidadosa de los pesos.** Por ejemplo, la inicialización de Xavier (o Glorot) y He son técnicas populares que consideran el tamaño de las capas anterior y siguiente para escalar los pesos iniciales. Esto ayuda a mantener los gradientes en una escala razonable, evitando que se desvanezcan o exploten a medida que se propagan a través de la red.

Otra técnica relevante es el uso de funciones de activación no lineales, como ReLU (Rectified Linear Unit) y sus variantes (Leaky ReLU, Parametric ReLU, etc.). Estas funciones ayudan a mitigar el problema del desvanecimiento de gradientes al proporcionar gradientes no saturados para valores positivos, lo que facilita el flujo de gradientes a través de la red.

Además, técnicas avanzadas como la normalización por lotes, donde se normaliza la salida de una capa antes de pasarla a la siguiente, pueden ayudar a estabilizar la distribución de las activaciones y los gradientes a lo largo de la red, haciendo el entrenamiento más estable y rápido.

En resumen, los problemas de gradientes que desaparecen o explotan pueden obstaculizar significativamente el entrenamiento de redes neuronales profundas. Sin embargo, mediante una inicialización cuidadosa de los pesos, la selección adecuada de funciones de activación y la aplicación de técnicas de normalización, es posible aliviar estos problemas y entrenar redes más profundas y efectivas.

## Inicialización de pesos.

Cuando creamos una red neuronal, cada conexión entre las neuronas tiene un "peso" asociado. Estos pesos determinan la importancia de las entradas a cada neurona. Antes de empezar a entrenar la red, necesitamos dar un valor inicial a estos pesos, y este proceso se llama "inicialización de pesos".

Imagina que estás tratando de equilibrar una balanza. Si partes de un punto muy desequilibrado, te tomará mucho tiempo encontrar el equilibrio. De manera similar, si los pesos de la red neuronal no se inicializan adecuadamente, el proceso de aprendizaje puede ser muy lento o incluso fallar. Los pesos iniciales incorrectos pueden llevar a problemas como:

* **Pesos que Explotan:** Los valores se hacen demasiado grandes muy rápidamente, como una bola de nieve cuesta abajo.
* **Pesos que Desvanecen:** Los valores se hacen tan pequeños que prácticamente desaparecen, como una huella en la arena que se borra.
* **Problema de simetría de pesos:** ocurre cuando todos los pesos y los bias son inicializados a ceros, como todas las neuronas parten del mismo punto de partido aprenden exactamente igual, entonces todas las neuronas estarían aprendiendo lo mismo, o lo que es lo mismo son todas iguales produciendo la misma salida, a este concepto se le ha denominado problema de simetría de pesos.

**Estrategias de Inicialización**

Lo primero que hay que tener claro es que los pesos no pueden ser inicializados a cero para evitar el problema de simetría de pesos, ni deben ser inicializados al mismo número todos los pesos, es decir, que deben tener diferentes valores, tras esto, evitamos el problema de simetría de pesos y podemos enfocarnos más en como solucionar los problemas derivados de los pesos que se vuelven muy grandes o pequeños.

De manera genérica, para solucionar estos problemas, la matriz de pesos de cada capa de la red neuronal será inicializada dependiendo del número de entradas provenientes de la capa anterior. Quedándonos que la inicialización quedaría como sigue en seudocódigo Python:

Dependiendo de la función de activación que utilicemos cambiara ligeramente el radicando de la raíz cuadrada. Veamos que particularidades tienen los siguientes casos:

* **Inicialización Uniforme**: Se eligen al azar, pero dentro de un rango específico para mantenerlos controlados. Suele ser entre el rango
* **Inicialización de Xavier/Glorot:** Especialmente buena para las funciones de activación "tanh" (una función matemática). Se basa en el número de entradas y salidas de la neurona para decidir el rango de los pesos iniciales. Se usa como radicando.
* **Inicialización de He:** Recomendada para cuando se usa la función de activación "ReLU" (otra función matemática). Similar a Xavier, pero utiliza una fórmula diferente que funciona mejor con ReLU. Se usará como radicando

## Aproximación numérica de gradientes.

Cuando entrenas una red neuronal con retropropagación, calculas los gradientes, que son las derivadas de la función de coste con respecto a los pesos de la red. Estos gradientes te dicen cómo cambiar los pesos para mejorar el rendimiento de la red. La verificación del gradiente es una forma de comprobar si estos cálculos de gradientes están bien hechos.

La idea es comparar el gradiente calculado durante la retropropagación con una aproximación del gradiente calculado numéricamente usando definiciones de cálculo y ver que ambas coinciden. Calcular la derivada exacta de una función puede ser complejo. La aproximación numérica ofrece una forma más sencilla de obtener un valor aproximado del gradiente.

Texto

Descripción generada automáticamente

La comprobación de gradientes con aproximación numérica de dos lados es una herramienta valiosa para asegurar la implementación correcta del algoritmo de retropropagación en redes neuronales. Aunque requiere un poco más de cálculo, la **precisión adicional compensa el esfuerzo**.

**¿Cómo Funciona?**

* Unificar los Parámetros: Primero, imagina que tomas todos esos tornillos y perillas y los alineas en una sola fila grande. En términos de programación, esto significa convertir todas las matrices y vectores de parámetros en un gran vector lineal, que llamaremos "theta".
* Unificar los Ajustes (Gradientes): Luego, haces lo mismo con las cantidades que te dicen cuánto ajustar cada tornillo, que son los gradientes de la función de coste con respecto a cada parámetro (estos son los "dW" y "dB").
* Calcular Aproximadamente los Ajustes: Para cada parámetro, calculas cuánto cambia la "función de coste" (una medida de qué tan bien está funcionando la máquina) si ajustaras ese tornillo un poco hacia adelante y un poco hacia atrás (esto es añadir y restar un pequeño valor "épsilon"). Este cálculo te da una aproximación del gradiente para ese parámetro.
* Comparar Aproximaciones con los Gradientes Reales: Después de hacer estos cálculos para todos los parámetros, tienes una lista de aproximaciones. Comparas esta lista con los ajustes que te dijo que hicieras la retropropagación.
* Evaluar la Exactitud: Para ver si estas dos listas son similares, calculas la diferencia entre ellas. Si la diferencia es muy pequeña (por ejemplo, menor que un valor muy pequeño como 10^-7), puedes estar bastante seguro de que la retropropagación está ajustando los tornillos correctamente. Si la diferencia es grande, probablemente hay un error en la manera en que la retropropagación está calculando los ajustes.

En resumen:

* La verificación de gradientes es una herramienta de depuración poderosa. Cuando construyes una red neuronal y no estás seguro si la retropropagación está funcionando como debería, esta técnica te puede ayudar a encontrar errores. Es como un control de calidad para asegurarte de que los cálculos complejos que estás haciendo en el fondo están bien hechos antes de confiar en que tu red neuronal está aprendiendo de la manera correcta.
* Compara las derivadas calculadas por tu código con aproximaciones numéricas.
* Si las diferencias son pequeñas, tu implementación es probablemente correcta.
* Si hay diferencias grandes, revisa los cálculos para encontrar el error.

# Algoritmos de optimización.

## Introducción

Las redes neuronales normalmente requieren el procesamiento de conjuntos de datos muy grandes y esto suele derivar en costes altos a nivel computacional y temporal, es por ello, que la optimización de los algoritmos es fundamental. En esta sección veremos:

## Mini-batch Descenso de gradiente.

Habíamos visto anteriormente que nuestro algoritmo de descenso de gradiente esta implementado para trabajar con “m” ejemplos, esto lo conseguimos con la vectorización de nuestro problema usando matrices y vectores.

Si el número de ejemplos “m” fuera muy grande por el orden de 1-5 millones de ejemplos, tendríamos matrices muy grandes, pero ¿Qué pasaría si dividiéramos el conjunto de “m” ejemplos en pequeños lotes a los que denominaremos “mini-batchs”?

Si consideramos que el tamaño de nuestros “mini-batchs” es de 1000 y que “m” es el numero de muestras totales. Una visión muy superficial de nuestro algoritmo de descenso de gradiente quedaría como sigue:

Para cada lote (mini-batch) de nuestro conjunto de mini lotes debemos hacer:

* Propagación hacia delante con el mini-batch “t” esimo.
* Cálculo de función de coste.
* Cálculo de gradientes.
* Actualización de pesos.

## El termino de época.

**El término "época"** en el contexto del aprendizaje de máquinas y redes neuronales se refiere a una pasada completa a través de todo el conjunto de datos de entrenamiento. Entonces, una época se completa cuando se han procesado todos los mini-batches que componen el conjunto de datos.

En el entrenamiento de redes neuronales, típicamente se realizan múltiples épocas de entrenamiento, donde cada época intenta refinar los pesos y sesgos de la red para mejorar su precisión. El número de épocas necesarias para entrenar una red neuronal puede variar dependiendo del tamaño del conjunto de datos, la complejidad de la red y otros factores como la tasa de aprendizaje y la función de coste.

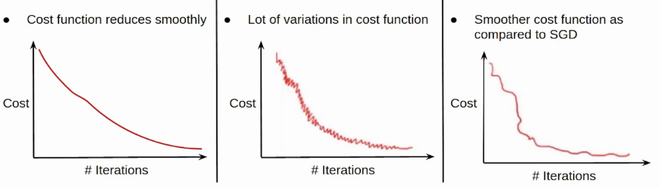
Por lo tanto, si tu conjunto de datos de entrenamiento se divide en mini-batches de, por ejemplo, 1000 ejemplos cada uno, y tienes 10,000 ejemplos en total, entonces tendrías 10 mini-batches por época. Después de procesar estos 10 mini-batches, habrías completado una época de entrenamiento.

Con respecto al tamaño del mini-batch tendremos las siguientes conclusiones:

* Si el tamaño del mini lote es igual a “m”, estaremos tratando el algoritmo de descenso de gradiente denominado “Batch gradient descent”, que es el que hemos estado viendo hasta ahora. Este algoritmo 𝑠uele implicar que cada iteración se vuelve muy lenta, porque encada iteración tenemos que procesar los millones de ejemplos que haya. Este **lo usamos si tenemos un conjunto de ejemplos relativamente pequeños.**
* Si el tamaño del mini lote es igual a 1, estaremos hablando de algoritmo especial denominado algoritmo de descenso de gradiente estocástico (Stochastic Gradient descent). Esta estrategia hace que perdamos las ventajas propias de vectorización de un problema en términos de eficiencia.

Lo ideal es encontrar un punto intermedio entre estos dos extremos, para aprovechar las ventajas de la vectorización, pero no hacer muy larga la computación durante una iteración. Además, cabe señalar que para los tamaños de **los mini-batch se suele poner potencias de 2 por motivos de eficiencia y uso de memoria**.

Otro aspecto para destacar es que ocurre con la función de coste cuando fragmentamos en mini-batchs nuestro entrenamiento, para ello, observemos la siguiente gráfica.



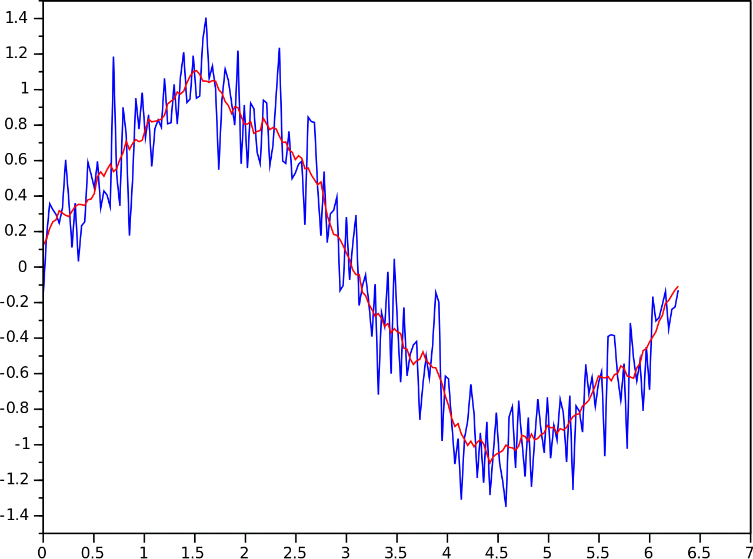
Una de las primeras cosas que podemos apreciar cuando implementamos el algoritmo del descenso de gradiente con la estrategia de “mini lotes” es que la función de coste no va reduciéndose de manera suave y lineal, sino que vemos que tiene continúas subidas y bajadas abruptas pero que, de media, de forma global va descendiendo.

Esto se debe a que cuando entrenamos con mini-batches, las estimaciones del gradiente de la función de pérdida se basan solo en un subconjunto de datos, lo que puede llevar a estimaciones ruidosas y a actualizaciones de los parámetros que oscilan considerablemente. Esto puede hacer que la trayectoria de optimización sea muy irregular, como se muestra en la gráfica de la derecha.

## Promedios Ponderados Exponenciales, PPE.

En la ultima parte del anterior epígrafe vimos como la función de coste experimentaba cambios disruptivos que no nos permitían ver de manera muy clara la tendencia general de la función de coste. Los PPE ayudan a suavizar estas oscilaciones al proporcionar una media móvil de los gradientes. En lugar de utilizar el gradiente calculado en cada iteración directamente, los PPE toman en cuenta los gradientes anteriores con un factor de suavizado que disminuye exponencialmente.

Así, los gradientes más recientes tienen más peso que los antiguos, pero todos contribuyen al valor actual del gradiente "suavizado". Esto lleva a actualizaciones de parámetros más estables y a una disminución más suave y consistente de la función de costo a lo largo del tiempo.ar de manera abrupta por “casos que generan ruido”.



El algoritmo de "Exponentially Weighted Averages" (Promedios Ponderados Exponenciales, PPE) es un método utilizado en la optimización de redes neuronales, entre otras aplicaciones, para suavizar las series temporales de datos, como las métricas de rendimiento durante el entrenamiento. Aunque no es un algoritmo de optimización en sí mismo, como lo serían el Descenso del Gradiente, Adam o RMSprop, juega un papel crucial en el contexto de la optimización, especialmente en la adaptación de tasas de aprendizaje y el momentum.

Cabe señalar de que actúan como un complemento o una técnica auxiliar dentro de los algoritmos de optimización para redes neuronales. **No son algoritmos de optimización por sí mismos**, sino más bien un **método para mejorar y estabilizar el proceso de optimizació**n que llevan a cabo algoritmos como el Descenso del Gradiente, Adam, RMSprop, entre otros.

De manera más coloquial lo que hace es darles más relevancia a los datos más recientes e ir olvidando la importancia de los datos más antiguos.

El concepto básico detrás de los PPE es que cada punto en el promedio ponderado está influenciado más significativamente por los puntos de datos más recientes, con la influencia disminuyendo exponencialmente para los puntos más antiguos. Esto se logra mediante la aplicación de un factor de ponderación que disminuye exponencialmente para los puntos de datos más antiguos. Su fórmula general viene dada por:

Tal que:

* : es el valor del PPE en el tiempo t
* es el factor de ponderación, con un valor entre 0 y 1. Un valor más alto da más peso a los valores anteriores, suavizando más la serie. Poniendo un ejemplo, si estamos midiendo el promedio de temperaturas de una ciudad un alto puede estar indicando que tengamos en cuenta los últimos 50 días, mientras que un bajo puede indicar que solo tengamos en cuenta los últimos 4 días.
* : Es el valor actual de la serie temporal en el tiempo
* : Es el valor del PPE en el tiempo t-1

Sus ventajas son:

* **Suavización:** Los PPE pueden ayudar a suavizar las fluctuaciones en los datos o en las métricas de rendimiento, lo que facilita la identificación de tendencias a largo plazo.
* **Sensibilidad a la Tendencia Reciente:** Al dar más peso a los datos más recientes, los PPE pueden adaptarse rápidamente a los cambios en las tendencias de los datos.
* La clave del éxito de los PPE en estos contextos radica en su capacidad para equilibrar la influencia de los gradientes pasados y recientes, permitiendo una transición más suave y estable hacia los mínimos de la función de perdida.

## Corrección del sesgo.

La corrección de sesgo (bias correction) en los Promedios Ponderados Exponenciales (PPE) es un ajuste que se realiza para compensar el sesgo que se introduce al inicializar las estimaciones del promedio ponderado.

Cuando comenzamos a calcular un PPE, la estimación inicial está basada solamente en el primer dato, ya que aún no se han acumulado suficientes datos para que el promedio ponderado refleje la realidad de la distribución de los datos. Esto es especialmente cierto en las primeras etapas del cálculo del promedio ponderado cuando se ha visto una cantidad muy pequeña de datos. Como resultado, el PPE puede estar sesgado hacia los valores iniciales, que posiblemente no sean representativos del conjunto de datos completo.

Para corregir este sesgo inicial, se aplica un factor de corrección que ajusta el PPE hacia lo que sería si hubiera estado operando sobre toda la historia de datos desde el inicio. En el contexto de los algoritmos de optimización como Adam, donde los PPE se utilizan para estimar momentos de los gradientes (tanto el primer momento que es el promedio de los gradientes, como el segundo momento que es el promedio de los cuadrados de los gradientes), esta corrección es crucial para asegurarse de que las estimaciones sean precisas lo más pronto posible. La fórmula de corrección de sesgo generalmente se ve así:

Donde:

* es el valor corregido del PPE en el tiempo t.
* es el valor calculado del PPE en el tiempo t sin corrección.
* es el factor de suavizado utilizado en el PPE.
* es el número de iteraciones (o el paso de tiempo en el contexto de la serie temporal).

Esta corrección de sesgo asegura que en las primeras iteraciones, cuando aún no se han acumulado suficientes datos para hacer una estimación robusta, el PPE no subestime la verdadera magnitud de los momentos del gradiente. Con el tiempo, a medida que se acumulan más datos, el impacto de la corrección de sesgo disminuye y el PPE refleja más precisamente el comportamiento real de los datos.

## Descenso de gradiente con momentum.

El descenso de gradiente con “momentum” es una variante del algoritmo de descenso de gradiente, utilizado en la optimización de funciones, especialmente en el contexto del aprendizaje automático y la inteligencia artificial. Tal y como hemos visto anteriormente, la idea detrás del descenso de gradiente es minimizar una función de coste o pérdida, ajustando iterativamente los parámetros del modelo en la dirección opuesta al gradiente de la función de coste con respecto a esos parámetros.

El "momentum" se introduce en este proceso para acelerar la convergencia y evitar quedarse atrapado en mínimos locales**. Funciona agregando una fracción del vector de actualización del paso anterior al vector de actualización actual**, similar a cómo un objeto en movimiento tiende a mantenerse en movimiento debido a su momento. Esto permite al algoritmo "acumular velocidad" a medida que se desplaza por la superficie de la función de coste, lo que puede resultar en convergencias más rápidas y suaves.

Aquí hay una descripción detallada de cómo funciona:

* **Inicialización:** Seleccionas valores iniciales para los parámetros que deseas optimizar. Además**, inicializas un vector de momentum**, generalmente con ceros, que tiene la **misma dimensión que los parámetros (pesos y bias).**
* **Cálculo del gradiente:** En cada iteración, calculas el gradiente de la función de coste con respecto a los parámetros actuales. Este gradiente indica la dirección de mayor incremento de la función de coste.
* **Actualización del momentum:** Antes de ajustar los parámetros, actualizas el vector de momentum multiplicando el momentum anterior por un factor de momentum (generalmente un valor entre 0 y 1) y luego sumando el gradiente actual multiplicado por la tasa de aprendizaje. Esto hace que el vector de momentum acumule una dirección ponderada de los gradientes anteriores. Las formulas quedaría como siguen:

Nota: La “v” la podemos interpretar como “velocidad”.

* **Actualización de parámetros:** Finalmente, ajustas los parámetros restando este vector de momentum actualizado de los parámetros actuales. Esto efectivamente mueve los parámetros en la dirección opuesta al gradiente ponderado acumulado, **teniendo en cuenta tanto el gradiente actual como la tendencia general de los gradientes anteriores**.
* **Iteración:** Repites los pasos 2 a 4 hasta que se cumpla algún criterio de parada, como un número máximo de iteraciones o un cambio mínimo en la función de coste.

El efecto del momentum es particularmente útil para superar los valles planos o las regiones donde el gradiente es pequeño, así como para **salir de los mínimos locales que no son el mínimo global.** Al acumular gradientes de pasos anteriores, el algoritmo puede **mantenerse en movimiento** y potencialmente atravesar estas regiones difíciles más eficientemente.

El momentum acumula gradientes pasados, donde el gradiente más reciente tiene más peso, y los pesos de los gradientes anteriores disminuyen exponencialmente, según el factor de momentum . El valor de este momentum suele ser 0.9.

## RMSprop

**RMSprop** es una modificación del método de descenso de gradiente y fue desarrollado como una solución a algunos de los problemas encontrados con el uso del descenso de gradiente estocástico, especialmente en contextos donde el gradiente puede ser muy diferente para diferentes parámetros o puede cambiar abruptamente.

La idea clave detrás de **RMSprop** es adaptar la tasa de aprendizaje para cada parámetro, dividiendo la tasa de aprendizaje para un parámetro por una medida de la reciente magnitud de los gradientes para ese parámetro. Esto ayuda a mitigar el problema del aprendizaje oscilante o demasiado rápido en pendientes pronunciadas y permite un aprendizaje más estable y rápido en superficies planas. La manera como Funciona RMSprop:

* **Acumulación de Gradientes Cuadrados:** RMSprop mantiene un promedio móvil del cuadrado de los gradientes para cada parámetro. Esto le da una medida de la variabilidad o la magnitud reciente de los gradientes para cada parámetro.
* **Ajuste de la Tasa de Aprendizaje:** La tasa de aprendizaje para cada actualización de parámetro se divide por la raíz cuadrada del promedio móvil mencionado, con un pequeño valor añadido para evitar la división por cero (esto se conoce como "suavizado" o "smoothing term", y suele ser un número muy pequeño

## ADAM

**Adam (Adaptive Moment Estimation)** es otro algoritmo de optimización popular que se utiliza ampliamente para entrenar redes neuronales. Es una combinación de las técnicas utilizadas en el descenso de gradiente con momentum y la técnica RMSprop.

Adam mantiene estimaciones separadas de los primeros momentos (el promedio de los gradientes, **similar al momentum**) y los segundos momentos (el promedio de los gradientes al cuadrado, **similar a RMSprop**) de los gradientes.

Estas estimaciones se utilizan para adaptar la tasa de aprendizaje para cada parámetro, lo que permite hacer ajustes más finos durante el entrenamiento. Además, Adam incluye un mecanismo de corrección de sesgos para los primeros y segundos momentos estimados, lo que lo hace particularmente efectivo durante las primeras iteraciones y cuando los momentos estimados están cerca de cero.

La estrategia general es inicializar a cero los variables que ya habíamos visto de los algoritmos de optimización de “momentum” y de “RMSprop”.

Tras esto, cada vez que se hayan calculados los gradientes con respectos de los pesos y el bias, antes de la actualización de los pesos y el bias, calculamos el “momentum 1” propio del algoritmo Momemtum y el “momentum 2” propio del algoritmo de RMSProp. Primero hacemos los cálculos propios del algoritmo de momentum:

Aplicamos corrección del sesgo explicada en anteriores epígrafes:

Tras esto hacemos los cálculos propios del algoritmo de “MSRProp”:

Y también aplicamos la corrección de sesgo:

Finalmente aplicamos la actualización de pesos:

Los valores normales para son 0.9 y para es 0.999, y suele estar en

Texto

Descripción generada automáticamente

## Learning Rate decay.

El concepto de "learning rate decay" se refiere a la estrategia de ir disminuyendo la tasa de aprendizaje a medida que avanza el entrenamiento. La idea es comenzar con pasos más grandes para hacer progresos rápidos al principio, cuando estás lejos del punto óptimo, y luego hacer los pasos más pequeños a medida que te acercas a la solución, evitando así el riesgo de "saltarse" la mejor solución debido a pasos demasiado grandes.

Esta estrategia es útil porque al principio, cuando los parámetros del modelo están muy lejos de sus valores óptimos, es seguro y eficiente hacer grandes ajustes. Pero a medida que te acercas a la solución óptima, quieres hacer ajustes más finos para afinar los parámetros y llegar lo más cerca posible al mejor conjunto de valores.

Ha varias alternativas para ir haciendo decrecer el “learning rate” a medida que pasan las épocas:

# Tuneo de hiperparámetros

## Introducción

En el entrenamiento de una red neuronal en números de hiperparámetros a configurar puede ser abrumador, por ejemplo, solo para el contenido que hemos visto hasta ahora podemos tener hiperparámetros como:

* Tasa de aprendizaje (): Determinan en que proporción se da un salto en cada iteración del algoritmo de descenso de gradiente. Es el más importante.
* Momentum (
* Hiperparámetros propios del algoritmo de optimización ADAM como , y . Normalmente se suele dejar en sus valores por defecto.
* Número de capas de la red neuronal
* Numero de neuronas por cada capa.
* Hiperparámetros para determinar como decae el learning rate.
* El tamaño de cada mini-batch.

Cuando trabajamos con modelos de aprendizaje automático, especialmente redes neuronales, tenemos que tomar decisiones sobre varios "hiperparámetros". Estos son parámetros que no se aprenden directamente del proceso de entrenamiento, sino que se configuran antes de iniciar el entrenamiento y pueden influir significativamente en el rendimiento del modelo. Ejemplos comunes de hiperparámetros incluyen el número de unidades ocultas en una capa de una red neuronal, el número total de capas en la red, la tasa de aprendizaje, etc.

**Muestreo Aleatorio vs. Escala Adecuada**

Una forma de encontrar buenos valores para estos hiperparámetros es mediante la "búsqueda de hiperparámetros", que puede incluir técnicas como el muestreo aleatorio. El muestreo aleatorio simplemente selecciona valores al azar para cada hiperparámetro dentro de un rango definido y evalúa el rendimiento del modelo con esos valores. Sin embargo, no todos los hiperparámetros deben muestrearse de la misma manera. Algunos, como el número de capas en una red, podrían muestrearse uniformemente dentro de un rango pequeño y discreto (por ejemplo, 2 a 4 capas). Pero para otros hiperparámetros, especialmente aquellos que pueden variar en órdenes de magnitud, como la tasa de aprendizaje, el muestreo uniforme no es eficiente.

Tomemos como ejemplo la tasa de aprendizaje (denotada como "alpha" en el video). Si tenemos un rango que va desde 0.0001 hasta 1 y muestreamos uniformemente, la mayoría de nuestros valores muestreados se agruparán en el extremo superior del rango, lo que nos dará muy poca información sobre el rendimiento del modelo en tasas de aprendizaje más bajas. Para abordar este problema, usamos una escala logarítmica para muestrear estos valores. Esto significa que en lugar de muestrear directamente dentro del rango de 0.0001 a 1, transformamos este rango a una escala logarítmica y luego muestreamos uniformemente dentro de este rango logarítmico. De esta manera, dedicamos más recursos a explorar el rango completo de valores posibles de manera más equitativa.

Para implementar esto en Python, podríamos generar un número aleatorio r entre -4 y 0, y luego usar como nuestro valor muestreado para la tasa de aprendizaje. Esto nos dará una distribución de valores que es uniforme en la escala logarítmica, permitiéndonos explorar eficientemente el rango completo de posibles tasas de aprendizaje.