https://medium.com/metadatos/todo-lo-que-necesitas-saber-sobre-el-descenso-del-gradiente-aplicado-a-redes-neuronales-19bdbb706a78

**Descenso del Gradiente Aplicado a Redes Neuronales**

Unared neuronal se compone de “neuronas” organizadas en capas:

Diagram

Description automatically generated

Cada neurona (exceptuando la capa de entrada) es un sumatorio de todas sus entradas; que no son más que las salidas de las capas anteriores multiplicadas por unos **pesos.**

A esta suma se le añade un término adicional llamado sesgo o **bias**.

Y al resultado se le aplica una función no lineal conocida como función de activación.

Diagram

Description automatically generated

Los **parámetros**de la red son (**pesos, bias**)

Los parametros son ajustamos con **entrenamiento**

Se entrena con sets con etiquetas, como un problema de M.L. supervisado

El resultado es un modelo que debe ser capaz de hacer predicciones con muestras futuras.

**Inicialización de los parámetros**

Se elige una arquitectura

Se inicializa sus parametros

Si partimos de cero (sin un modelo pre-entrenado) es bastante común proceder de la siguiente forma:

* Inicializaremos los pesos aleatoriamente para ayudar a la red, *rompiendo su simetría*.
  + El fin es evitar que todas las neuronas de una capa acaben aprendiendo lo mismo.
  + Se suelen generar los pesos de cada capa usando una [distribución normal](https://es.wikipedia.org/wiki/Distribuci%C3%B3n_normal" \t "_blank) de media cero y varianza **1/n** ó **2/n**(siendo ***n***el número de entradas).
  + Este valor de la varianza depende un poco de la función de activación que se coloque a la salida de la neurona (usaremos [2/n para ReLU](https://arxiv.org/abs/1502.01852)).
* Inicializaremos los parámetros de bias a cero.

A partir de aquí procederemos iterativamente siguiendo un algoritmo de **optimización**, que tratará de minimizar la diferencia entre la salida real y la estimada por la red.

**Descenso del gradiente**

El algoritmo más utilizado para entrenar redes neuronales es el **descenso del gradiente**.

El gradiente es un cálculo que nos permite saber cómo ajustar los parámetros de la red de tal forma que se minimice su desviación a la salida.

**Distintas versiones**

El algoritmo cuenta con varias versiones dependiendo del número de muestras que introduzcamos a la red en cada iteración:

* Descenso del gradiente **en lotes**(o *batch*): todos los datos disponibles se introducen de una vez. Esto supondrá problemas de estancamiento, ya que el gradiente se calculará usando siempre todas las muestras, y llegará un momento en que las variaciones serán mínimas. Importante: La entrada a una red neuronal tenga algo de aleatoriedad.
* Descenso del gradiente **estocástico**: se introduce una única muestra aleatoria en cada iteración. El gradiente se calculará para esa muestra concreta, lo que supone la introducción de la deseada aleatoriedad, dificultando así el estancamiento. El problema de esta versión es su lentitud, ya que necesita de muchas más iteraciones, y además no aprovecha los recursos disponibles.
* Descenso del gradiente (estocástico) **en mini-lotes**(o *mini-batch*): en lugar de alimentar la red con una única muestra, se introducen *N*muestras en cada iteración; conservando las ventajas de la segunda versión y consiguiendo además que el entrenamiento sea más rápido debido a la paralelización de las operaciones. Nos quedamos pues con esta modificación del algoritmo, eligiendo un valor de *N*que nos aporte un buen balance entre aleatoriedad y tiempo de entrenamiento (también que no sea demasiado grande para la memoria de la GPU disponible).

Chart, line chart

Description automatically generated

[Comparación de tiempos](https://www.d2l.ai/chapter_optimization/minibatch-sgd.html)

**El Algoritmo**

Estas son las iteraciones que lleva a cabo el algoritmo del Descenso del Gradiente, en su tercera versión:

**1…** Introducimos un **mini-lote** de entrada con N muestras aleatorias provenientes de nuestro dataset de entrenamiento, previamente etiquetado (lo que significa que conocemos la salida real).

**2…** Después de los cálculos en cada capa de la red, obtenemos como resultado las predicciones a su salida. A este paso se le conoce como *forward propagation*(de las entradas).

**3…** Evaluamos la **función de coste** (también llamada función de pérdida, MSE si fuera regresion) para dicho mini-lote. Se trata de una función [elegida previamente](https://towardsdatascience.com/deep-learning-which-loss-and-activation-functions-should-i-use-ac02f1c56aa8" \t "_blank) en base al tipo de problema concreto, para poder evaluar de la forma más adecuada la diferencia entre las predicciones de nuestra red y las salidas reales. El valor de esta función de coste es lo que se trata de minimizar en todo momento mediante el algoritmo, y hacia ello se orientan los siguientes pasos.

Table

Description automatically generated

[Funciones de coste para los problemas más comunes](https://towardsdatascience.com/deep-learning-which-loss-and-activation-functions-should-i-use-ac02f1c56aa8)

**4…** Calculamos el **gradiente**como la **derivada multivariable**de la función de coste con respecto a todos los parámetros de la red. Gráficamente sería la **pendiente**de la tangente a la función de coste en el punto donde nos encontramos (evaluando los pesos actuales). Matemáticamente es un **vector** que nos da la dirección y el sentido en que dicha función aumenta más rápido, por lo que deberíamos movernos en sentido contrario si lo que tratamos es de minimizarla.

Si tuviéramos una red con sólo 2 parámetros, podríamos representar la función de coste en 3 dimensiones (plano XY para los valores de los parámetros, plano Z para el valor del coste).

El gradiente en este caso será un vector de 2 componentes (una en el eje X y otra en el eje Y), y nos marcará un sentido en el plano XY hacia donde deberíamos mover los parámetros para aumentar más rápido el coste.

Su módulo será mayor cuanto mayor sea la pendiente de la tangente en el punto donde se calcula, y por consiguiente será menor cuanto más cerca estemos de un mínimo de la función de coste (en un mínimo la pendiente es 0).

La idea por tanto es movernos en sentido contrario al del gradiente, tratando de alcanzar el mínimo global:

Chart, surface chart

Description automatically generated

[Ejemplo de ensueño](https://eli.thegreenplace.net/2016/understanding-gradient-descent/)

Por desgracia

* Las funciones de coste son mucho más complejas.
* No tendremos 2 parámetros o dimensiones, sino posiblemente millones;
* y además la superficie de la función presentará mínimos locales que durante el entrenamiento pueden confundirse con el mínimo global.

Chart, surface chart

Description automatically generated

[La vida es así de dura](https://docs.scipy.org/doc/scipy/reference/tutorial/optimize.html)

El cálculo del vector gradiente en una red neuronal profunda no es para nada algo trivial. Se complica bastante debido a la gran cantidad de parámetros y a su disposición en múltiples capas. *¿Cómo saber lo que influye realmente la variación de un parámetro de la primera capa en el coste final si esa variación repercute en todas las neuronas de todas las capas sucesivas?*

Disponemos del algoritmo ***back-propagation*** (*o propagación hacia atrás*) este calcula las derivadas parciales de la función de coste a la salida con respecto únicamente a los parámetros de la última capa (que no influyen sobre ningún otro parámetro de la red).

Una vez obtenidas estas derivadas, pasamos a la capa anterior, y calculamos nuevamente las derivadas parciales de la función de coste, pero ahora con respecto a los parámetros de esta capa, algo que en parte ya tenemos resuelto precisamente por la regla de la cadena.

Y así seguiremos progresando hacia atrás, hasta llegar al inicio de la red.

Esto lo haran los frameworks que ya existen, pero teniamos que conocer la teoria.

**5…** Una vez obtenido el vector gradiente, actualizaremos los parámetros de la red restando a su valor actual el valor del gradiente correspondiente, multiplicado por una **tasa de aprendizaje** que nos permite ajustar la magnitud de nuestros pasos. El término de actualización se resta porque como ya hemos visto antes queremos avanzar en el sentido contrario al del gradiente, para que la función de coste disminuya. Según nos acerquemos al mínimo global, los pasos serán en teoría más pequeños porque la pendiente de la función de coste será menor, pero mejor dejemos de pensar en el ejemplo irreal en tres dimensiones: se suele optar igualmente por ir disminuyendo la tasa de aprendizaje con el tiempo.

Repetiremos todos los pasos mientras el valor de la función de coste y las métricas de salida no empiecen a empeorar de forma sostenida.

**A tener en cuenta**

Seguiremos el proceso anterior tratando de alcanzar el **mínimo global** de la función de coste, con la esperanza de no acabar *rebotando*en torno al mismo, hundidos en un mínimo local, o estancados en un “punto silla” (mínimo local en una dimensión, máximo local en otra).

En realidad lo del punto silla es improbable que ocurra si hemos elegido una versión estocástica del descenso del gradiente, y también será más fácil escapar de las proximidades de un mínimo local en dicho supuesto.

Igualmente para evitar un final no deseado del algoritmo será muy importante una elección adecuada de la **tasa de aprendizaje**; pudiendo decidir sobre lo siguiente:

* Su valor. No deberá ser ni muy alto (posibles oscilaciones) ni muy bajo (lentitud, más iteraciones)

Diagram

Description automatically generated with medium confidence

[Cosas que pueden ocurrir si no acertamos con la tasa de aprendizaje](http://www.bdhammel.com/learning-rates/)

* Si será fija o variable con el tiempo. Podemos hacer por ejemplo que vaya disminuyendo con el tiempo, para evitar oscilaciones cuando estemos cerca del mínimo de la función de coste. O también podemos aplicar una tasa cíclica que aumente y disminuya para tratar de esquivar los mínimos locales.
* Si será la misma para toda la red, o distinta por capa o grupos de capas. Cuando usamos [Transferencia de Aprendizaje](https://en.wikipedia.org/wiki/Transfer_learning" \t "_blank) (escribiré un artículo entero sobre el tema) conviene elegir una tasa baja para entrenar la parte del modelo pre-entrenado, y otra más alta para las capas que añadimos nosotros.

El sobreajuste ocurrirá si entrenamos durante demasiadas *epochs*

Unaa ***epoch***es un ciclo del algoritmo en el que la red ve todas las muestras disponibles una vez.

Ejecutar demasiadas *epochs*implica procesar las mismas muestras demasiadas veces, y de ahí el sobreajuste.

Esto puede venir provocado por un dataset pequeño en comparación con el tamaño de la red (si el número de muestras de entrenamiento es mucho menor que el número de parámetros o grados de libertad de la red), pero también por una tasa de aprendizaje demasiado pequeña que no nos permita optimizar los parámetros con la suficiente rapidez.

Para luchar contra el sobreajuste aplicaremos técnicas usadas ya con otros algoritmos de Machine Learning (destinar parte de las muestras para **validación** del modelo, obtener más datos, etc) pero también técnicas específicas para Deep Learning, como son las técnicas de **regularización**; que tratan de compensar de diversas maneras la exagerada cantidad de parámetros en la red.

Lo bueno es que estas técnicas de regularización también nos pueden ayudar a salvar los mínimos locales, matando dos pájaros de un tiro. De nuevo es algo que espero explicar en más detalle dentro de otro artículo que tengo pendiente.

**Optimizando el optimizador**

El descenso del gradiente es un método de optimización de primer orden, ya que toma las primeras derivadas de la función de coste.

Eso nos da información puntual sobre la pendiente de la función, pero no sobre su curvatura, lo que hace que nos falte el contexto por así decirlo.

Podríamos calcular las segundas derivadas, de forma que pudiéramos conocer también cómo varía el gradiente, pero eso supondría un elevado coste computacional.

Existen algunas técnicas de aproximación donde se estiman esas segundas derivadas con un uso de memoria limitado, como es el caso de [LBFGS](https://es.wikipedia.org/wiki/L-BFGS); pero lo más utilizado son optimizaciones sobre el descenso del gradiente estocástico. A continuación se detallan algunas de las más populares:

**Momentum**

Actualizar los parámetros de la red añadiendo un término adicional que tenga en cuenta el valor de la actualización aplicada en la iteración anterior, de tal forma que se estarán teniendo en cuenta los gradientes anteriores además del actual. El gradiente actual se multiplicará por la tasa de aprendizaje (*η*) y el valor de la actualización anterior por una constante conocida como **coeficiente del momentum** (*γ*), con un valor típico de 0,9.

A picture containing text, watch, night sky

Description automatically generated

Lo que obtenemos con esto es una [media móvil ponderada exponencialmente](https://es.wikipedia.org/wiki/Media_m%C3%B3vil#Media_m%C3%B3vil_exponencial) del gradiente, de tal forma que el avance sea más rápido cuando nos movemos en la dirección correcta, y las posibles oscilaciones se atenúen cuando se *rebote*en la función de coste. Ambas características hacen que la convergencia se produzca más rápido.

**RMSProp (Root Mean Square Propagation)**

La tasa de aprendizaje se adapta para cada parámetro; igual que en otro método conocido como **Adagrad**. RMSProp mejora a este último precisamente por incluir la media móvil exponencial del gradiente **al cuadrado** (igual que otro algoritmo similar llamado **Adadelta**). También aparece una constante*ρ*, que se conoce como **factor de olvido:**

A picture containing text, watch, clock

Description automatically generated

(actualización particular para cada parámetro)

**Adam (Adaptive Moment Estimation)**

Una combinación de RMSProp con el Momentum. Por un lado tendremos la media móvil exponencial del gradiente al cuadrado, y por otro la media móvil exponencial de los pasos anteriores. Suele ser el más rápido de los vistos hasta ahora, y el que se usa por defecto en *[fastai](https://github.com/fastai/fastai" \t "_blank)*a día de hoy.

A picture containing text, watch, clock

Description automatically generated

(actualización particular para cada parámetro)

A continuación un ejemplo a modo de comparativa (hay que tener en cuenta que no siempre Adam es la mejor opción):

A picture containing chart

Description automatically generated

[Ejemplo](https://medium.com/@alexrachnog/optimization-cookbook-1-262aa5555b61)

**NOTA**: En el momento de escribir este artículo se está hablando de nuevos métodos que mejoran lo que ya conocíamos: **RAdam**, **LookAhead**, y [una combinación de ambos](https://medium.com/@lessw/new-deep-learning-optimizer-ranger-synergistic-combination-of-radam-lookahead-for-the-best-of-2dc83f79a48d). ¡Quizá es buen momento para echarles un ojo!