Artificial Intelligence for ICT professionals

Entrenamiento de redes neuronales

valgrai.eu

Valencian Graduate School and Research Network of Artificial Intelligence

Índice

1. Introducción

Bueno, entrenar una NN no es fácil

- Vamos a tratar de introducir los **elementos básicos** en el proceso de entrenamiento de una red neuronal
- Los dos componentes principales serán el algoritmo de descenso de gradiente (ya introducido en regresión lineal y regresión logística) y el algoritmo de retropropagación (**backpropagation**), el elemento básico y fundamental en el entrenamiento de redes neuronales profundas
- La retropropagación se describió por primera vez en la década de 1970, pero se hizo popular por un artículo en 1986 de David Rumelhart, Geoffrey Hinton y Ronald Williams

2. La función de coste/pérdida

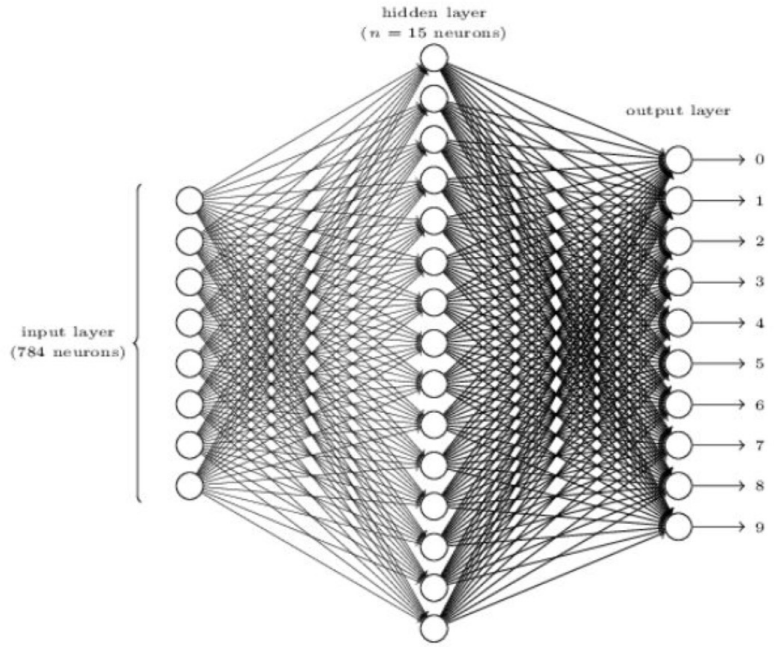
Objetivo: minimizarla

- Como contexto de algunos de nuestros ejemplos en esta unidad, utilizaremos el famoso conjunto de datos **MNIST** (60000 dígitos de entrenamiento, 10000 dígitos de prueba), todos de 0 a 9 dígitos escritos a mano
- 28x28 pixels

2. La función de coste/pérdida

Objetivo: minimizarla

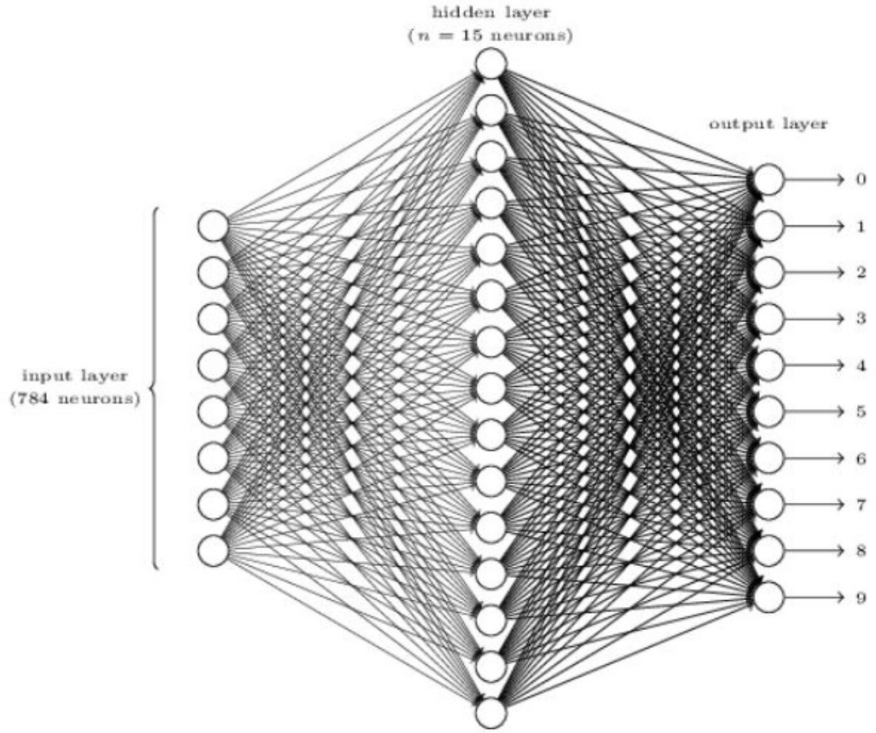
- **Input layer:** Imágenes 28x28 (en realidad no son neuronas, solo entradas)
- Un problema de clasificación en este caso, de 0 a 9, totalmente conectado y feed-forward
- Concatenamos todos los píxeles en una fila de 784 elementos, donde en la capa de entrada tendremos como entrada un valor de 0 a 1 (escala de grises)
- Usaremos solo 1 capa oculta con 15 neuronas (hiperparámetros para experimentar)
- **The output layer**: 10 neuronas con valores de 0 a 1 (sigmoide), donde se seleccionará la de máximo valor de activación



2. La función de coste/pérdida

Objetivo: minimizarla

- **¿Cuántas neuronas hay en la capa oculta? ¿Cuántas capas ocultas?** Son hiperparámetros sin reglas exactas, tal vez pautas dependiendo de la arquitectura de la red neuronal, por lo que tendremos que entrar en un bucle de 'entrenar/evaluar'



2. La función de coste/pérdida

Objetivo: minimizarla

- **¿Qué está pasando en la capa oculta?** Generalmente, las representaciones intermedias que tenemos en una red neuronal son difíciles de entender y representar (modelo de caja negra)
- Está claro que cuantas más neuronas y capas ocultas más potente será la red neuronal, y los patrones más complejos serán capaces de detectar a partir de nuestro conjunto de datos de entrada, aunque desde un punto específico más neuronas no van necesariamente siempre a ayudar (la NN tiene suficiente potencia para expresar la variabilidad de los datos de entrada, e incluso podría ser una fuente de sobreajuste)
- En nuestra pequeña red neuronal para MNIST, el pensamiento es un poco más intuitivo, y es posible ver qué neuronas se activan delante de unos dígitos y podemos deducir qué formas o características están aprendiendo nuestras neuronas en la capa oculta

2. La función de coste/pérdida

Objetivo: minimizarla

- **Notación**
  - x: Ejemplo de entrada o entrenamiento (en nuestro caso un vector de 784 píxeles)
  - y(x): valor deseado de nuestra red, representado como un vector de 10 componentes
  - Ej: y(x) = (0, 0, 1, 0, 0, 0, 0, 0, 0) representa que la salida deseada para un ejemplo dado es 3
- y(x) es la función que queremos aprender (la función real u óptima), que tiene que asociar el vector de 784 entradas para el vector de 10 salidas
- La red neuronal **no es más que una función** que nos dará el vector resultado (10 componentes) a partir de un vector de entrada x (una imagen) y la colección de valores de todos los pesos y bias de las neuronas tras un proceso de entrenamiento
- Recuerda que comenzaremos con valores aleatorios para todas las **w's** y **b's**, y el proceso de aprendizaje consistirá en encontrar los mejores valores de las mismas a partir del proceso de entrenamiento
- Definiremos la función de la red neuronal como **a(x, w, b)**, siendo x la entrada, w todos los pesos, b todos los sesgos; o simplemente para simplificar.

2. La función de coste/pérdida

Objetivo: minimizarla

- **Objetivo:** Obtener (aprender) el conjunto de parámetros **w** y **b** de la red neuronal que mejor se aproxima a la función **y(x)** para todos los valores **x** (del conjunto de datos de entrenamiento)
- Necesitamos, entonces, medir qué tan bien **a(x, w, b)** se aproxima a **y(x)**
- Hacemos esto definiendo (como hicimos en la regresión lineal y logística) una función de costo:

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

MSE y norma vectorial, ya que en este caso la salida es un vector

- donde:

1. **a** es la salida de la red para **a(x, w, b)**
2. **n** en el número total de muestras de nuestro conjunto de datos de entrenamiento
3. **w** y **b** son todos los parámetros de la red neuronal
4. La suma de **x** es la suma de todas las muestras de entrenamiento de nuestro conjunto de datos de entrenamiento

2. La función de coste/pérdida

Objetivo: minimizarla

$$C(w,b) \equiv \frac{1}{2n}\sum_{x}\|y(x)-a\|^2$$

- La **función de coste** es la suma de los valores que mide el **error** (pérdida) que está cometiendo nuestra red a al aproximar la función real **y**
- Estamos obteniendo el cuadrado de la distancia vectorial, un valor siempre positivo, entre la salida real y el valor estimado de nuestra red

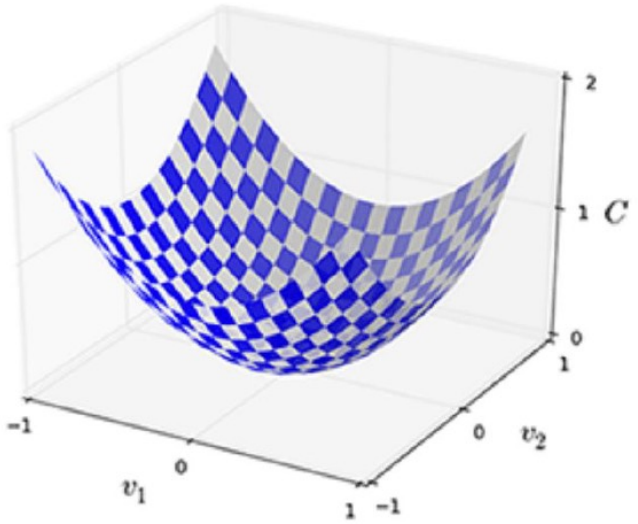
  - Por ejemplo, podríamos tener:
    - **a(x, w, b)** = (0.9, 0.1, 0.1, 0.2, 0.1, 0.1, 0, 0, 0, 0)
    - **y(x)** = (1, 0, 0, 0, 0, 0, 0, 0, 0, 0)
    - En este caso, el error será: $(1-0.9)^2+(0-0.1)^2+(0-0.1)^2+(0-0.2)^2+...$
    - En la función de coste, sumamos todos los errores de todos los **x** del conjunto de datos de entrenamiento

2. La función de coste/pérdida

Objetivo: minimizarla

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

- Nuestro objetivo es entrenar la red neuronal para conseguir que nuestra aproximación al objetivo real sea la mejor posible, es decir, **minimizar lo máximo posible C(w,b)**
- Este problema es un problema de **optimización**: queremos obtener los valores de **w** y **b** que minimicen el valor de C



Un ejemplo de **C** con solo 2 parámetros y convexo. Como veremos, tendremos miles (incluso más) de parámetros y la función de coste será no convexa, es decir, con muchos mínimos locales.

2. La función de coste/pérdida

Objetivo: minimizarla

loss function

cost function

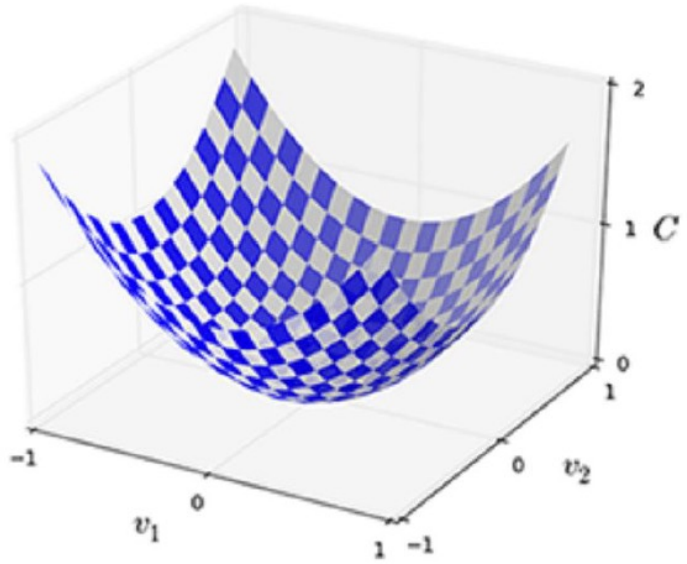
$$C(w,b) \equiv \frac{1}{2n}\sum_{x}\|y(x)-a\|^2$$

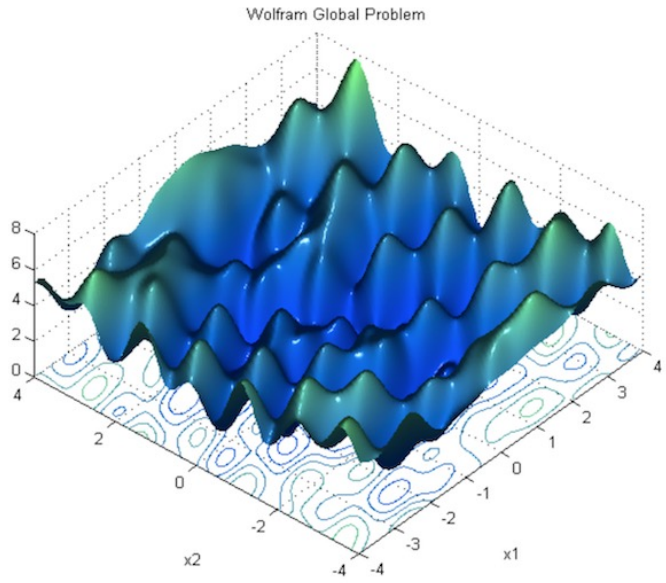
- Además del **MSE** (Error Cuadrático Medio) existen diferentes funciones de pérdida para medir el error
- En el caso, por ejemplo, de la **clasificación**, normalmente usamos la pérdida de **entropía cruzada** (con la última capa de neuronas usando una función de activación softmax para obtener valores de probabilidad de 0 a 1)
- Es habitual utilizar la función de pérdida o de coste, aunque técnicamente la función de pérdida es el error de una muestra y la función de coste es la suma de todas las muestras del conjunto de datos de entrenamiento

3. Gradient descent

Un algoritmo para minimizar una función

- Para simplificar nuestra comprensión de la aplicación del gradiente de descenso en el contexto de las redes neuronales, veremos primero cómo minimizar una función general **C(v)**, donde **v** puede ser un vector con varios componentes (de nuevo, para simplificar, usaremos 2)
- Una forma de calcular su mínimo es hacerlo analíticamente usando la derivada: **C'(v) = 0**.
- Sin embargo, en el caso de la complejidad y el elevado número de parámetros en una red neuronal, esto no va a ser posible

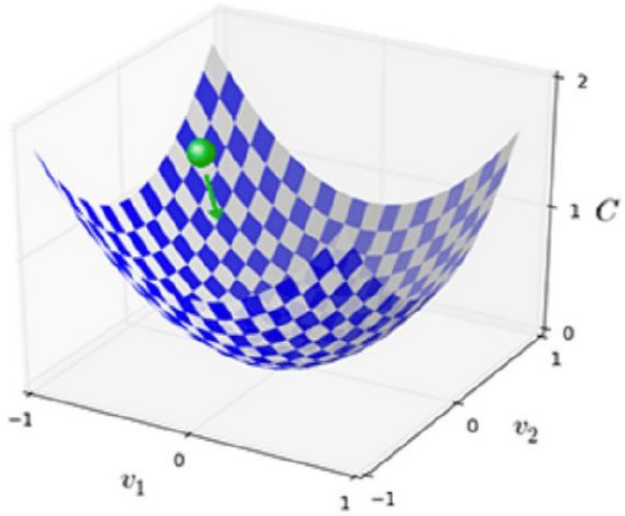




3. Gradient descent

Un algoritmo para minimizar una función

- Comenzamos con un aleatorio **(v1, v2)**
- Encontramos la dirección de la pendiente/cambio máximo hacia el **mínimo** (hacia abajo)
- Damos un **pasito** en esta dirección
- Repetimos este proceso desde el nuevo punto generado hasta llegar a un **mínimo** (local)



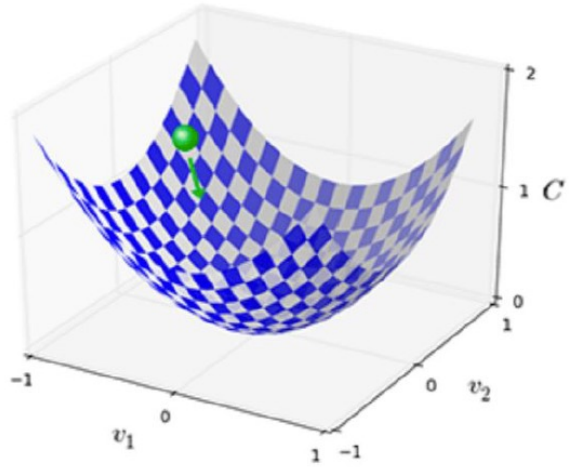
3. Gradient descent

Un algoritmo para minimizar una función

- Definimos el vector de la variación de **v**

$$\nabla C \equiv \left(\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2}\right)^T$$

- Y el **gradiente** de **C**, que define la dirección de la **pendiente máxima**, como



3. Gradient descent

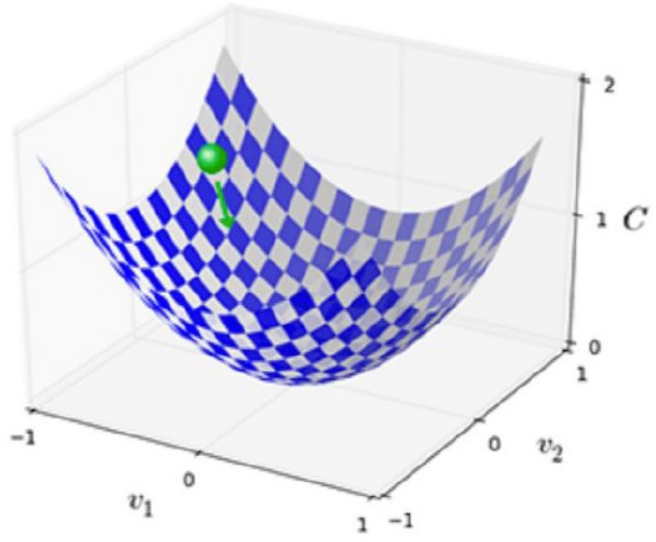
Un algoritmo para minimizar una función

- Podemos definir la regla de actualización para pasar de un punto **v** a un punto **v'**, donde **C** tiene un valor menor, como:

$$v' = v - \eta \nabla C$$

Repetimos este proceso de forma iterativa

- Esta es la regla de actualización del descenso de gradiente



3. Gradient descent

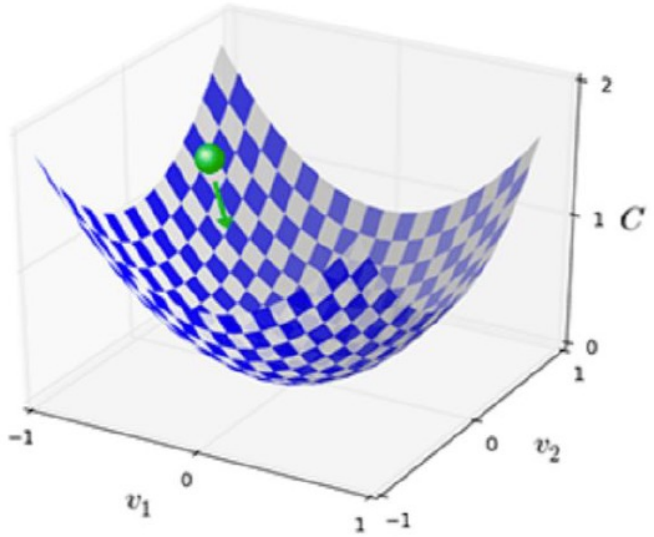
Un algoritmo para minimizar una función

- Aplicamos la regla de actualización

$$v' = v - \eta \nabla C$$

hasta que lleguemos a un mínimo

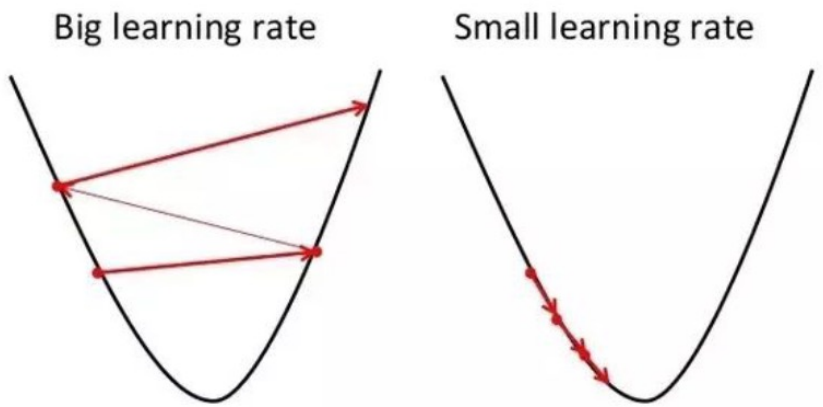
- **Intuición:** nos ”dejamos caer” en la dirección de la máxima variación ya que llegamos a un **mínimo** de la función



4. Learning rate

El hiperparámetro para cambiar un poco las w y b de las neuronas

- La tasa de aprendizaje η, como su nombre lo indica, define de alguna manera la velocidad en la que funciona el descenso de gradiente
- Tiene que ser lo suficientemente pequeña para que la aproximación local que define la derivada sea correcta
- Si η es demasiado grande, podemos no encontrar el mínimo e incluso divergir a un valor mayor de C
- Por otro lado, un valor muy pequeño de la misma haría que el algoritmo avanzara de una manera muy lenta



4. Learning rate

El hiperparámetro para cambiar un poco las w y b de las neuronas

- En nuestro caso, los parámetros de la función a minimizar son los pesos **w** y los bias **b** de la red

- Por lo tanto, la regla de actualización es así:

$$w'_k = w_k - \eta \frac{\partial C}{\partial w_k}$$

$$b'_l = b_l - \eta \frac{\partial C}{\partial b_l}$$

para cada uno de los valores de $\mathbf{w_k}$ y $\mathbf{b_l}$

- La función de coste tiene que ser **diferenciable**, ya que necesitamos calcular las derivadas.

5. SGD

Stochastic Gradient Descent

- **En la práctica**, las redes neuronales **no se entrenan calculando el gradiente completo**, sino que obtenemos una estimación del mismo a partir de una muestra aleatoria de ejemplos de entrenamiento
- En lugar del gradiente completo,, por ejemplo 128, que es una gran diferencia para conjuntos de datos con miles o millones de eje**promediamos los gradientes de una pequeña muestra de ejemplos**mplos de entrenamiento
- Este algoritmo se denomina Descenso de Gradiente Estocástico o **SGD**

5. SGD

Stochastic Gradient Descent

- Elegimos **m** ejemplos para el conjunto de datos al **azar**: X1, X2, ..., Xm
- Este conjunto de muestras se denomina lote o minilote (**batch**)
- Tamaños comunes para un batch: 16, 32, 64, 128, 256, 512... Cuanto mayor sea el batch, mejor será la aproximación al gradiente real, pero más operaciones
- El uso de batches también ayuda a llevar a cabo operaciones vectoriales/matriciales

- **Estimación de la pendiente**:

$$\nabla C \approx \frac{1}{m}\sum_{j=1}^{m} \nabla C_{X_j}$$

- Por lo tanto, la regla de actualización con SGD es, con j de 1 a m:

$$w'_k = w_k - \frac{\eta}{m}\sum_{j} \frac{\partial C_{X_j}}{\partial w_k}$$

$$b'_l = b_l - \frac{\eta}{m}\sum_{j} \frac{\partial C_{X_j}}{\partial b_l}$$

For 1 batch

5. SGD

Stochastic Gradient Descent

- El entrenamiento se lleva a cabo **batch por batch** (selección aleatoria) hasta completar (considerar) todos los elementos del conjunto de datos de entrenamiento
- Cuando todos los valores del conjunto de datos se han utilizado en los batches, entonces decimos que hemos entrenado un **epoch**
- Proceso:

```
for epoch=1 to num_epochs:
    while there are training examples left to be considered in the epoch:
        1. choose a batch of m elements not used in the epoch
        2. calculate gradients of the parameters and apply SGD (update rule)
```

$$w'_k = w_k - \frac{\eta}{m}\sum_j \frac{\partial C_{X_j}}{\partial w_k}$$

$$b'_l = b_l - \frac{\eta}{m}\sum_j \frac{\partial C_{X_j}}{\partial b_l}$$

Nos estamos moviendo más rápido al mínimo, ya que cambiamos el peso y los bias con más frecuencia (más pasos).
Sin embargo, el **gradiente** es una **estimación**, por lo que, si es demasiado pequeño, este movimiento o cambio en los parámetros puede ser más arbitrario y ser problemático para la convergencia al mínimo.

6. Backpropagation

Todo el proceso de formación de NN

Repasemos lo que hemos aprendido y agreguemos una pieza que falta: el algoritmo de retropropagación

- Una red neuronal consta de capas interconectadas de nodos o ”neuronas”, y cada capa aprende a extraer características de los datos de entrada. Cada nodo de una capa está conectado a cada nodo de la siguiente capa mediante ’pesos’, y cada nodo también tiene un ‘bias’. Estos pesos y bias son los parámetros que la red aprende durante el entrenamiento.
- Los datos de entrada se pasan a través de la red y cada nodo aplica una transformación lineal (multiplicando la entrada por el peso y añadiendo el bias), seguida de una función de activación no lineal. La salida de la última capa es la predicción de la red.
- Para medir qué tan buenas son estas predicciones, usamos una **función de pérdida L**. Una opción común para **L** es el error cuadrático medio para los problemas de regresión, o la entropía cruzada para los problemas de clasificación:

```
L = 1/N Σ (y_true - y_pred)^2   # Mean Squared Error

L = -1/N Σ y_true * log(y_pred) # Cross Entropy
```

6. Backpropagation

Todo el proceso de formación de NN

- El objetivo del entrenamiento es **ajustar** los **pesos** y los **bias** para **minimizar esta función de pérdida**.
- Hacemos esto usando una técnica llamada **descenso de gradiente**.
- El **gradiente** de una función nos da la dirección de **ascenso más pronunciado**.
- Para encontrar el **mínimo**, queremos **ir en la dirección opuesta**, es decir, la dirección de descenso más pronunciado.
- Calculamos el gradiente de la función de pérdida con respecto a cada parámetro (peso o sesgo), y actualizamos el parámetro restando este gradiente, escalado por un factor llamado tasa de aprendizaje **η**:

```
weight = weight - η * ∂L/∂weight
bias = bias - η * ∂L/∂bias
```

6. Backpropagation

Todo el proceso de formación de NN

- Calcular el **gradiente de L** con respecto a cada parámetro no es solo para la capa de salida, **tenemos que hacerlo también para cualquier parámetro de las neuronas de las capas anteriores**.
- Para calcular eficientemente estos gradientes, usamos un algoritmo llamado **retropropagación**.
- La **retropropagación** primero **realiza un paso hacia adelante a través de la red, calculando la salida y la pérdida**, y luego atraviesa la red hacia atrás, calculando **el gradiente de la pérdida con respecto a cada parámetro**.
- Aplica la regla de la **cadena de las derivadas** para desglosar el cálculo del gradiente en partes manejables (resuelve la determinación de los gradientes de cada nodo teniendo en cuenta que una NN es una composición de funciones)

6. Backpropagation

Todo el proceso de formación de NN

```
# Forward pass
for each layer in network:
    output = activation(weight * input + bias)

# Backward pass
for each layer in reversed(network):
    ∂L/∂weight = ∂L/∂output * ∂output/∂weight
    ∂L/∂bias = ∂L/∂output
```

6. Backpropagation

Todo el proceso de formación de NN

- En la práctica, calcular el gradiente en todo el conjunto de datos puede ser costoso desde el punto de vista computacional, especialmente para conjuntos de datos grandes.
- En su lugar, utilizamos una variante de descenso de gradiente llamada descenso de **gradiente estocástico (SGD)**.
- En SGD, estimamos el gradiente utilizando un pequeño subconjunto aleatorio o "**batch**" del conjunto de datos.
- Calculamos el gradiente para este **batch**, actualizamos los parámetros y luego pasamos al siguiente **batch**. Este proceso se repite hasta que hayamos cubierto todo el conjunto de datos, lo que constituye un "**epoch**".
- Todo el proceso se repite durante varios **epochs** hasta que el rendimiento de la red es satisfactorio (convergencia del error)

```
# Stochastic Gradient Descent
for epoch in number_of_epochs:
    for batch in dataset:
        # Forward pass
        for each layer in network:
            output = activation(weight * input + bias)

        # Calculate loss
        loss = loss_function(y_true, output)

        # Backward pass
            for each layer in reversed(network):
                ∂L/∂weight = ∂L/∂output * ∂output/∂weight
                ∂L/∂bias = ∂L/∂output

        # Update parameters
        for each layer in network:
            weight = weight - η * ∂L/∂weight
            bias = bias - η * ∂L/∂bias
```

6. Backpropagation

Todo el proceso de formación de NN

```
for epoch=1 to num_epochs:
    while there are training examples left to be considered in the epoch:
        1. choose a batch of m elements not used in the epoch
        2. calculate gradients of the parameters and apply SGD (update rule)
```

Backpropagation algorithm

$$w'_k = w_k - \frac{\eta}{m}\sum_j \frac{\partial C_{X_j}}{\partial w_k}$$

$$b'_l = b_l - \frac{\eta}{m}\sum_j \frac{\partial C_{X_j}}{\partial b_l}$$

Valencian Graduate School and Research Network of Artificial Intelligence

valgrai.eu

valgrai.eu

Valencian Graduate School and Research Network of Artificial Intelligence