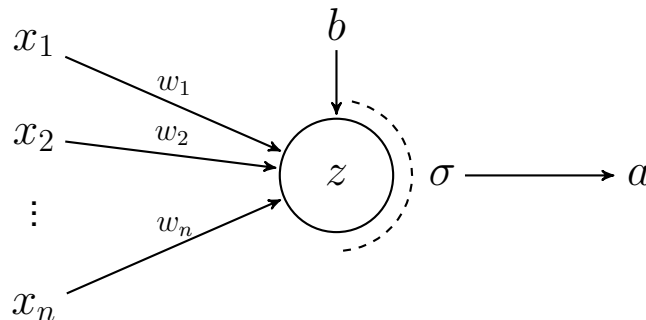


1. Concepto de Neurona y Red Neuronal

1.1. Concepto de Neurona:

Componentes característicos de una neurona:

- **Inputs** (x_i): Datos de entrada de la neurona.
- **Output** (a): Resultado o valor de la neurona.
- **Pesos** (w_i) y **bias** (b): son los parámetros de aprendizaje, en función de estos se optimizará el sistema.
- **Función de Activación** (σ): es un elemento no-lineal de la neurona encargado de dotarla de mayor complejidad. Más en (B).



El output de la neurona vendrá dado por la aplicación de la función de activación a una suma ponderada de los inputs:

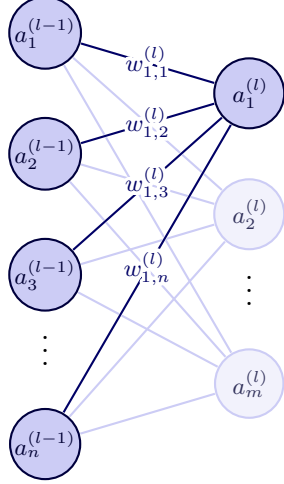
$$a = \sigma \left(\sum_{i=1}^n (w_i x_i) + b \right)$$

Cada neurona representa, en un espacio n-dimesional, una recta de regresión lineal 'distorsionada', la cual tiene asociada unos pesos (w_1, w_2, \dots, w_n) y un bias (b) que actúa como término independiente. Este concepto de neurona presenta la limitación que la composición de regresiones lineales da lugar a otra regresión lineal, es por eso que la función de activación σ es un elemento no-lineal necesario para dotar de mayor complejidad a la red.

Por cuestiones de uniformidad en la notación, a veces se omite el término independiente b integrandolo como un peso más, i.e., se asume $x_0 = 1$ y $w_0 = b$.

1.2. Concepto de Capa:

La composición de neuronas organizadas en capas da lugar a una red neuronal, pudiendo expresar matricialmente el valor de una capa en función del valor de la capa anterior. Están interconectadas entre sí del siguiente modo:



$$a^{(l)} = \sigma(\underbrace{W^{(l)}a^{(l-1)} + b^{(l)}}_{=: z^{(l)}})$$

donde identificamos

- Vector de valores de la capa l y $l - 1$

$$a^{(l)} = \begin{pmatrix} a_1^{(l)} \\ a_2^{(l)} \\ \vdots \\ a_m^{(l)} \end{pmatrix}, a^{(l-1)} = \begin{pmatrix} a_1^{(l-1)} \\ a_2^{(l-1)} \\ \vdots \\ a_n^{(l-1)} \end{pmatrix} \text{ con } a_i^{(l)} \text{ el valor de la } i\text{-ésima neurona de la capa } l$$

- Matriz de pesos de la capa l

$$W^{(l)} = \begin{pmatrix} w_{1,1}^{(l)} & w_{1,2}^{(l)} & \cdots & w_{1,n}^{(l)} \\ w_{2,1}^{(l)} & w_{2,2}^{(l)} & \cdots & w_{2,n}^{(l)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m,1}^{(l)} & w_{m,2}^{(l)} & \cdots & w_{m,n}^{(l)} \end{pmatrix} \text{ con } w_{i,j}^{(l)} \text{ es el peso que conecta } a_j^{(l-1)} \text{ con } a_i^{(l)}$$

- Vector de biases de la capa l

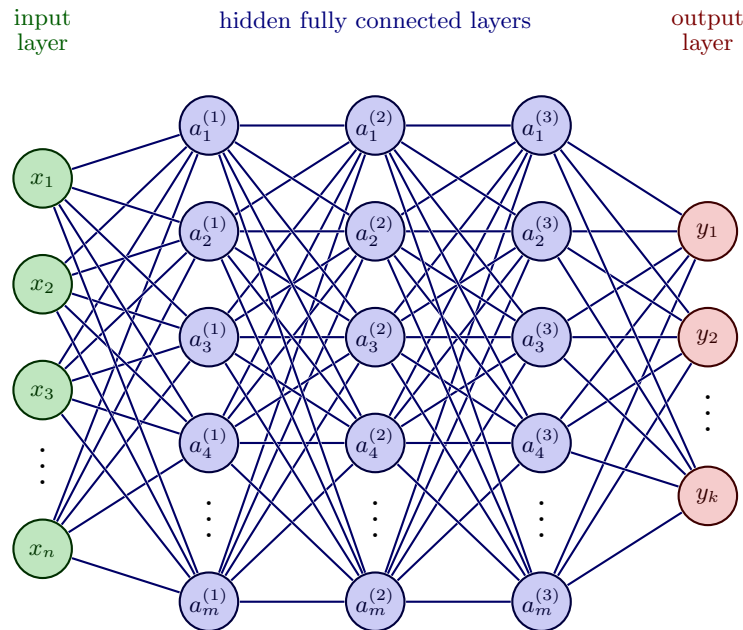
$$b^{(l)} = \begin{pmatrix} b_1^{(l)} \\ b_2^{(l)} \\ \vdots \\ b_m^{(l)} \end{pmatrix} \text{ con } b_i^{(l)} \text{ el bias de la } i\text{-ésima neurona de la capa } l$$

1.3. Concepto de Red Neuronal:

Existen 3 tipos de capas principales:

- **Capa de entrada**¹: Datos de entrada.
- **Capas ocultas**: Procesan los datos a través de una serie de transformaciones no lineales. Puede haber una o más de estas capas.
- **Capa de salida**: Resultado final de la red.

Tanto el número de neuronas en la capa de entrada como en la capa de salida está sujeto a la naturaleza de los datos y el target esperado, por lo que podemos considerarlas como 'fijas', mientras que el n^o de capas ocultas y el n^o de neuronas en dichas capas es un hiperparámetro 'modificable'.



Las redes neuronales multicapa (MLP) son un tipo de red neuronal artificial que se caracteriza por tener múltiples capas de neuronas. En ella, la información se propaga de forma directa (forward propagation) pasando a través de la red, capa por capa, aplicando funciones de activación en cada neurona para producir una salida final. Este proceso determina las predicciones de la red antes de cualquier ajuste de pesos durante el entrenamiento.

Componentes característicos de una Red Neuronal:

- **Inputs (x_i)**: Datos de entrada de la red.
- **Output (y_i)**: Resultado o valor de la red.
- **Parámetros**:
 - **Parámetros entrenables - pesos ($w_{i,j}^{(l)}$) y biases ($b_i^{(l)}$)**: Parámetros de aprendizaje en función de los cuales se optimizará la red. Se suele referir a ellos simplemente como 'parámetros'.
 - **Hiperparámetros**: Configuraciones que se establecen antes del entrenamiento del modelo y no se modifican durante este. Algunos ejemplos son: función de activación (σ), función de coste (\mathcal{C}), estructura y número de neuronas y capas, n^o de épocas o iteraciones, tasa de rendimiento (η), tamaño y n^o de mini-lotes, parámetro de regularización (λ), parámetros de optimizadores (β, ϵ)...

¹Por cuestiones de notación, se suele expresar la capa de entrada como $a^{(0)}$ y la de salida como $a^{(L)}$

2. Descenso de Gradiente

En el proceso de entrenamiento de una red neuronal se disponen de unos casos de prueba consistentes en unos datos de entrada y unos datos de salida o **targets** esperados. El objetivo de una red neuronal es ajustar sus pesos y biases para que su output se acerque a los targets. Para medir el grado de exactitud o precisión de una red neuronal con respecto a los casos de prueba se utiliza una función de coste que devuelve la 'distancia' entre el target esperado y el output de la red neuronal.

- Denotaremos a la función de coste como $\mathcal{C}(\theta)$ donde θ es el conjunto de parámetros entrenables, es decir, los pesos y biases de la red.

Las funciones más utilizadas son la 'cross-entropy function' (clasificación) o la 'Mean Squared Function' (regresión). Más en (B).

El gradiente (∇) de una función indica la dirección de máxima pendiente en un punto, es decir, la dirección de más rápido ascenso. El descenso de gradiente consiste en ir actualizando los parámetros de la siguiente forma:

$$\theta_{t+1} = \theta_t - \eta \nabla \mathcal{C}(\theta_t)$$

donde $\eta > 0$ es un hiperparámetro conocido como la tasa de aprendizaje (learning rate), la cual suele ajustarse para que (1) converja y (2) lo haga lo más rápido posible.

Para calcular $\nabla \mathcal{C}(\theta_t)$ utilizaremos el algoritmo de Backpropagation, el cual veremos en el siguiente apartado.

2.1. Descenso de Gradiente Estocástico (SGD)

Las funciones de coste se definen para evaluar el 'error' entre vectores, es decir, entre el output de la red y el output esperado de **una sola muestra**. El descenso de gradiente regular tiene en cuenta todos los datos de entrenamiento en todas las iteraciones, esto es, calcula el gradiente de todas las muestras y luego hace la media. Esto hace el entrenamiento muy lento y poco eficiente, por eso se plantea el Descenso de Gradiente Estocástico (SGD) donde se dividen los datos de entrenamiento en muestras y en cada iteración se utiliza una de ellas.

SGD

El procedimiento^a consiste en varios ciclos donde:

1. Se dividen los datos de entrenamiento en mini-lotes aleatoriamente.
2. Por cada mini-lote se hace una iteración donde:
 - a) Se calcula el gradiente de la función de coste restringida al mini-lote haciendo la media de los gradientes de cada dato.

$$\nabla \mathcal{C}_{batch}(\theta_t) = \frac{1}{len(batch)} \left(\sum_{data \in batch} \nabla \mathcal{C}_{data}(\theta_t) \right)$$

- b) Se actualizan los parámetros: $\theta_{t+1} = \theta_t - \eta \nabla \mathcal{C}_{batch}(\theta_t)$

^aTodas las operaciones son elemento a elemento

3. Backpropagation

Llegados hasta este punto es fundamental introducir el algoritmo de backpropagation o retropropagación. Esta técnica es clave para entrenar redes neuronales y consiste en calcular el gradiente de la función de coste respecto de cada peso propagando el error desde la salida hacia todas las capas de la red.

Calculamos las derivadas parciales de la función de coste respecto de los parámetros de la siguiente forma

$$\begin{aligned} \frac{\partial C}{\partial w_{i,j}^{(l)}} &= \delta_i^{(l)} a_j^{(l-1)} \\ \frac{\partial C}{\partial b_i^{(l)}} &= \delta_i^{(l)} \end{aligned} \quad \text{con} \quad \begin{cases} \text{Caso base } (l = L): & \delta_i^{(L)} = \frac{\partial C}{\partial a_i^{(L)}} \sigma'(z_i^{(L)}) \\ \text{Caso inductivo } (l < L): & \delta_i^{(l)} = \sum_k (\delta_k^{(l+1)} w_{k,i}^{(l+1)}) \sigma'(z_i^{(l)}) \end{cases}$$

$\delta_i^{(l)}$ se puede ver como la responsabilidad que tiene la i -ésima neurona de la capa l en la función de coste. Es por esto que se le suele llamar el **error imputado** por la neurona.

Dem.:

Recordemos la regla de la cadena multivariable en derivadas parciales: supongamos que $f(x_1, x_2, \dots, x_m)$ y $x_i = x_i(t_1, t_2, \dots, t_n)$ $i = 1, \dots, m$, son funciones diferenciables. Entonces

$$\frac{\partial f}{\partial t_j} = \frac{\partial f}{\partial x_1} \frac{\partial x_1}{\partial t_j} + \frac{\partial f}{\partial x_2} \frac{\partial x_2}{\partial t_j} + \dots + \frac{\partial f}{\partial x_m} \frac{\partial x_m}{\partial t_j}$$

Por la regla de la cadena obtenemos que

$$\frac{\partial C}{\partial w_{i,j}^{(l)}} = \frac{\partial C}{\partial z_i^{(l)}} \cdot \frac{\partial z_i^{(l)}}{\partial w_{i,j}^{(l)}} \quad \text{y} \quad \frac{\partial C}{\partial b_i^{(l)}} = \frac{\partial C}{\partial z_i^{(l)}} \cdot \frac{\partial z_i^{(l)}}{\partial b_i^{(l)}} \quad (1)$$

Definimos el error imputado por la i -ésima neurona de la capa l , $\delta_i^{(l)} := \frac{\partial C}{\partial z_i^{(l)}}$,

Además, es claro que $\frac{\partial z_i^{(l)}}{\partial w_{i,j}^{(l)}} = a_j^{(l-1)}$ y $\frac{\partial z_i^{(l)}}{\partial b_i^{(l)}} = 1$. Luego, sustituyendo en (1) queda

$$\boxed{\frac{\partial C}{\partial w_{i,j}^{(l)}} = \delta_i^{(l)} \cdot a_j^{(l-1)} \quad \text{y} \quad \frac{\partial C}{\partial b_i^{(l)}} = \delta_i^{(l)}} \quad (2)$$

De nuevo por la regla de la cadena calculamos el valor de $\delta_i^{(l)}$

$$\delta_i^{(l)} = \frac{\partial C}{\partial a_i^{(l)}} \cdot \frac{\partial a_i^{(l)}}{\partial z_i^{(l)}} = \frac{\partial C}{\partial a_i^{(l)}} \cdot \sigma'(z_i^{(l)}) \quad (3)$$

donde $\frac{\partial C}{\partial a_i^{(l)}}$ depende de la capa en la que estemos

- Caso base ($l = L$): Depende de la función de coste escogida
- Caso inductivo ($l < L$): $\frac{\partial C}{\partial a_i^{(l)}} = \sum_k \left(\frac{\partial C}{\partial z_k^{(l+1)}} \cdot \frac{\partial z_k^{(l+1)}}{\partial a_i^{(l)}} \right) = \sum_k (\delta_k^{(l+1)} w_{k,i}^{(l+1)})$

Por tanto,

$$\boxed{\delta_i^{(L)} = \frac{\partial C}{\partial a_i^{(L)}} \cdot \sigma'(z_i^{(L)}) \quad \text{y} \quad \delta_i^{(l)} = \sum_k (\delta_k^{(l+1)} w_{k,i}^{(l+1)}) \cdot \sigma'(z_i^{(l)}) \text{ para } l < L}$$

3.1. Extensión Matricial

Habiendo comprendido el funcionamiento del algoritmo de backpropagation, resulta conveniente extender este concepto mediante una representación matricial que nos permitirá expresar las operaciones a nivel de capas de manera más compacta.

Para ello debemos definir previamente la extensión matricial de las derivadas parciales

$$\frac{\partial C}{\partial A} := \begin{pmatrix} \frac{\partial C}{\partial a_{1,1}} & \frac{\partial C}{\partial a_{1,2}} & \dots & \frac{\partial C}{\partial a_{1,n}} \\ \frac{\partial C}{\partial a_{2,1}} & \frac{\partial C}{\partial a_{2,2}} & \dots & \frac{\partial C}{\partial a_{2,n}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial C}{\partial a_{m,1}} & \frac{\partial C}{\partial a_{m,2}} & \dots & \frac{\partial C}{\partial a_{m,n}} \end{pmatrix}$$

el error imputado por una capa

$$\delta^{(l)} := \begin{pmatrix} \delta_1^{(l)} \\ \delta_2^{(l)} \\ \vdots \\ \delta_{n_l}^{(l)} \end{pmatrix}$$

y la extensión vectorial de la función de activación

$$\sigma \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{pmatrix} := \begin{pmatrix} \sigma(v_1) \\ \sigma(v_2) \\ \vdots \\ \sigma(v_n) \end{pmatrix}$$

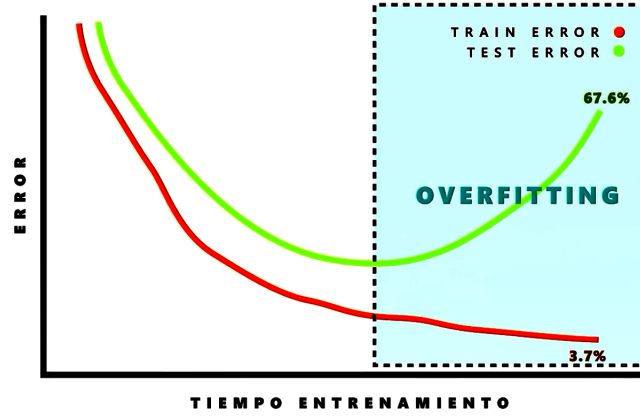
$$\begin{cases} \frac{\partial C}{\partial W^{(l)}} = \delta^{(l)} \cdot (a^{(l-1)})^t \\ \frac{\partial C}{\partial b^{(l)}} = \delta^{(l)} \end{cases} \quad \text{con} \quad \begin{cases} \text{Caso base } (l = L): & \delta^{(L)} = \frac{\partial C}{\partial a^{(L)}} \odot \sigma'(z^{(L)}) \\ \text{Caso inductivo } (l < L): & \delta^{(l)} = (W^{(l+1)})^t \cdot \delta^{(l+1)} \odot \sigma'(z^{(l)}) \end{cases}$$

donde la operación \odot es el producto de Hadamard o 'element-wise product', i.e., el producto elemento a elemento.

4. Optimización de Redes Neuronales

4.1. Overfitting

El Overfitting es una consecuencia directa del exceso de iteraciones en el entrenamiento, sucede cuando el modelo se sobreajusta a los datos de entrenamiento modelizando el ruido y evitando una correcta generalización.



La identificación del overfitting suele hacerse con una división de los datos (datos de entrenamiento y datos de prueba) y la observación de una reversión de la curva de error.

Hay distintos métodos para evitarlo:

- **Regularización:** Estas regularizaciones se basan en una modificación de la función de coste de forma que penalicen los datos extremos, i.e., fuerza a que los pesos sean pequeños. Esto se interpreta como una preferencia por modelos menos complejos que evitan ajustarse excesivamente a los datos de entrenamiento, lo que podría incluir ruido o particularidades irrelevantes. En términos simples, la regularización ayuda a que los modelos aprendan patrones más generales y robustos, mejorando así su capacidad para predecir con precisión sobre datos no vistos, *a pesar de no tener una explicación teórica completa de por qué funciona de manera consistente en todas las situaciones.*

- Regularización $L1$ o 'Lasso': Mitigación del efecto de datos de entrada irrelevantes

$$C' = C + \lambda \sum_{w \in W_t} |w| = C + \lambda \cdot \|W_t\|_1$$

$$\nabla C'(\theta_t) = \nabla C(\theta_t) + \lambda \cdot \text{sign}(W_t)$$

- Regularización $L2$ o 'Ridge': Mitigación del efecto de la correlación entre datos

$$C' = C + \frac{\lambda}{2} \sum_{w \in W_t} w^2 = C + \frac{\lambda}{2} \cdot \|W_t\|_2^2$$

$$\nabla C'(\theta_t) = \nabla C(\theta_t) + \lambda \cdot W_t$$

- Regularización Lm : Triplazo mío

$$C' = C + \frac{\lambda}{m} \sum_{w \in W_t} |w|^m = C + \frac{\lambda}{m} \cdot \|W_t\|_m^m$$

$$\nabla C'(\theta_t) = \nabla C(\theta_t) + \lambda \cdot \text{sign}(W_t) \cdot |W_t|^{m-1}$$

- *WeightDecay*: Afecta directamente a la actualización de los pesos donde de ser

$$W_{t+1} = W_t + \Delta W_t$$

pasa a ser

$$W_{t+1} = (1 - \lambda)W_t + \Delta W_t$$

Utilizando SGD la regularización *L2* es equivalente a un *Weight Decay* reparametrizado. Sin embargo, esta última técnica es más extrapolable a variantes más avanzadas. como Adam y sus derivados.

Nota: W_t es θ_t pero cambiando todos los biases por 0.

En todas las regularizaciones, el factor $\lambda > 0$ determina cuán regularizado estará el coste.

- **Dropout**: Eliminación/apagado de neuronas aleatoriamente en cada época del entrenamiento para evitar la dependencia de unos con otras y hacer la red más robusta. Durante el proceso de entrenamiento existe una diferencia en la velocidad de convergencia de los distintos pesos, este fenómeno (llamado *coadaptación*) es en gran parte mitigado por el dropout.

Durante el entrenamiento, se aplica un dropout en cada neurona con una probabilidad p . Esto se puede representar utilizando una variable de máscara $r_j^{(l)}$ (que se actualiza en cada iteración) para cada neurona j en la capa l , donde:

$$r_j^{(l)} \sim \text{Bernoulli}(p)^2$$

La activación de la neurona de la capa l , ya en forma matricial, con dropout se define como:

$$\tilde{a}^{(l)} = r^{(l)} \odot a^{(l)}$$

De esta forma, el algoritmo de backpropagation quedaría igual que la versión sin dropout pero escalado en un factor de $r_j^{(l)}$. Así, la media/esperanza matemática quedaría escalada por un factor de p .

Durante la fase de inferencia no se aplica dropout (todas las neuronas activas). Para que esta fase sea coherente con el entrenamiento, debemos mantener la expectativa de las activaciones similares a las del entrenamiento escalándola por el factor p , que es la media o esperanza matemática de la variable $r_j^{(l)}$ ($\mathbb{E}[r_j^{(l)}] = p$)

$$a_i^{(l)} \leftarrow p a_i^{(l)}$$

De esta manera, la salida de cada neurona en la fase de inferencia se multiplica por la probabilidad de no aplicarle dropout durante el entrenamiento.

Se suele escoger $p = 0,5$ ³ en las capas ocultas y $p = 0,8$ o mayor en las de entrada. No se recomienda en ningún caso un $p < 0,5$.

- **Aumento del nº de datos**: Aumento del nº de datos de entrenamiento, ya sea consiguiendo nuevos datos (lo cual suele ser complicado) o extendiéndolos artificialmente (modificación de los datos originales mediante translaciones, distorsiones, rotaciones... para aumentar la flexibilidad de la red ante variaciones). Con suficientes datos de entrenamiento es complicado que la red, por muy grande que sea, los 'overfittee'.

²La notación $r_i^{(l)} \sim \text{Bernoulli}(p)$ indica que $r_i^{(l)}$ es una variable aleatoria que sigue una distribución de Bernoulli con parámetro p . Esto significa que $r_i^{(l)}$ puede tomar el valor 0 con probabilidad $P(r_i^{(l)} = 1) = p$ o el valor 1 con probabilidad $P(r_i^{(l)} = 0) = 1 - p$

³Porque la virtud está en el punto medio y porque $p = 0,5$ maximiza la varianza $\text{Var}[r_j^{(l)}] = p \cdot (1 - p)$, lo que produce el mayor número de combinaciones posibles de neuronas activas, obligando a la red a aprender de muchas configuraciones posibles y, así, generalizando mejor

- **Early Stopping:** Detección del punto en el que la curva del error/precisión (del conjunto de validación) empeora a medida que transcurren las iteraciones. No es una técnica para evitar el reverso de la curva, sino para detectarlo y parar a tiempo. Por ejemplo, la técnica 'no-improvement-in- n ' consiste en llevar la cuenta del mejor resultado y parar cuando se lleve un tiempo sin mejorarse. Este método introduce un nuevo hiperparámetro n . Existen estrategias más agresivas y otras más moderadas.
- **División de los datos:** Correcta división (aleatoria e independiente) de los datos en suficientes conjuntos (entrenamiento, verificación, validación...).

4.2. Batch Normalization

Normalmente, al entrenar NN aplicamos una preparación/normalización a los datos de entrada, no obstante, las activaciones intermedias (outputs de cada capa antes de pasar por la función de activación) pueden tener distribuciones que cambian durante el entrenamiento. Este fenómeno se conoce como *shift de covarianza interno*. Estos cambios en las distribuciones de las activaciones pueden dificultar el proceso de entrenamiento, haciendo que el modelo necesite más épocas para converger y aumentando la sensibilidad a la inicialización de los pesos. Para ello ideamos el Batch Normalization que **aumenta la velocidad de convergencia**, aporta simetría y estabilidad a la función de coste que **permite inicializaciones de pesos sub-óptimas** y actúa como un **pequeño regularizador**. *Todo esto está respaldado por técnicas experimentales y análisis teóricos estimativos, pero caracemos de una explicación sólida que nos ayude a entender totalmente porqué esta técnica funciona tan bien.* Por cada iteración, se normaliza cada capa de activaciones intermedias $z^{(l)}$ de la siguiente forma:

1. **Cálculo de la media y varianza** de cada capa a lo largo de un minibatch: Sea $z_{data}^{(l)}$ el vector de activaciones intermedias de la capa l resultado de la propagación directa del dato *data* del minilote *batch*,

$$\mu_{batch}^{(l)} = \frac{1}{m} \sum_{data \in batch} z_{data}^{(l)} \quad \sigma_{batch}^{2(l)} = \frac{1}{m} \sum_{data \in batch} (z_{data}^{(l)} - \mu_{batch}^{(l)})^2$$

2. **Normalización de las activaciones:**

$$\hat{z}_{data}^{(l)} = \frac{z_{data}^{(l)} - \mu_{batch}^{(l)}}{\sqrt{\sigma_{batch}^{2(l)} + \epsilon}}$$

donde ϵ es un pequeño valor constante añadido para evitar la división por cero. Así, la activación de cada neurona a lo largo de los datos de un minilote forma una distribución de probabilidad normal $\mathcal{N}(\mu = 0, \sigma^2 = 1)$.

3. **Escala y Desplazamiento:** Finalmente, se aplica una transformación lineal a las activaciones normalizadas para permitir que la red recupere la capacidad de representar las distribuciones originales si es necesario,

$$\text{BN}(z_{data}^{(l)}) = \gamma^{(l)} \hat{z}_{data}^{(l)} + \beta^{(l)}$$

Estos parámetros son también entrenables.

Influencia en la Propagación Directa

La normalización es un paso intermedio entre el cálculo de la combinación lineal de las activaciones de capas anteriores y la aplicación de la función de activación, quedaría de la siguiente forma:

$$a^{(l)} = \sigma(\text{BN}(\underbrace{W^{(l)} a^{(l-1)} + b^{(l)}}_{=: z^{(l)}}))$$

Para poder normalizar correctamente el cómputo debe hacerse en el siguiente orden: recorrer la red capa a capa (bucle externo) simultáneamente con todos los datos del minibatch (bucle interno).

Influencia en el Backpropagation

Habría que ver como cambia las derivadas respecto de pesos/bias y respecto de los parámetros estos nuevos.

4.3. Selección heurística de hiperparámetros

Cuando no existe una teoría sólida y robusta en la que basarse, las heurísticas sobre redes neuronales desempeñan un papel crucial en la investigación ayudando a formular hipótesis y diseñar experimentos que, con el tiempo, pueden conducir a una comprensión más precisa y detallada del funcionamiento de las redes neuronales.

Enfrentar un problema de redes neuronales por primera vez puede generar dudas sobre varios aspectos. La depuración de redes neuronales es compleja y requiere habilidad y mucho ensayo y error, similar a la programación, muchas veces resultando en frustración. Es crucial aprender a depurar y desarrollar heurísticas para seleccionar buenos hiperparámetros y arquitecturas.

Para seleccionarlos es común reservar un conjunto de casos de prueba al que podemos llamar '*casos de validación*'.

- **Estructura de la red:** Como ya hemos comentado, el número de neuronas de las capas de entrada y salida suelen considerarse fijos, mientras que la estructura relativa a las capas ocultas es variable. Para las redes neuronales con pocas capas ocultas, aumentar el número de neuronas en dichas capas puede resultar beneficioso para el rendimiento de la red, ya que puede permitir modelos patrones más complejos. No obstante, el exceso de neuronas ocultas aumenta el coste computacional y puede favorecer el overfitting, por lo que hay que encontrar un equilibrio.
- **Número de Iteraciones:** Es bastante fácil de optimizar intentando no quedarse cortos y utilizar la técnica dinámica del early stopping.
- **Inicialización de los pesos:** La inicialización de los pesos/bias de forma aleatoria (campana de Gauss Normalizada) supone una limitación y un posible estancamiento en mínimos locales. Existen distintos criterios para inicializar los parámetros, ya sea una modificación de la distribución de probabilidad (p.ej., acentuar la campana por el centro evitando así la saturación) o copiando una los pesos de una red ya entrenada para realizar una tarea relacionada pero más general que la de tu red.
- **Tasa de aprendizaje:** Una tasa de aprendizaje alta puede acelerar el proceso de entrenamiento y ayudar a salir de mínimos locales. Sin embargo, esto también puede provocar inestabilidad en el entrenamiento, haciendo que la red oscile y no converja adecuadamente. Por otro lado, una tasa de aprendizaje baja asegura una convergencia más estable y precisa, pero el entrenamiento puede volverse extremadamente lento, y existe el riesgo de quedarse atrapado en mínimos locales o en un progreso muy lento.
También cabe destacar que la tasa de aprendizaje depende fuertemente de la función de coste ⁴ y está altamente relacionada con el número de iteraciones (como norma general a menor η mayores han de ser las iteraciones para obtener un resultado comparable).
- **Función de Coste y Activación:** Para tareas de regresión se suele usar MSE, MAE, Huber... Para tareas de clasificación se suele usar Cross Entropy + Softmax. Por otro lado, la función sigmoide y *tanh* tienen la capacidad de universalidad pero presentan el problema de 'Vanishing Gradient', en cambio la función ReLU y sus variantes solucionan este problema. Para más mirar Apéndice B.
- **Parámetro de early stopping n :** Generalmente es un hiperparámetro fácil de ajustar y depende de la naturaleza de los datos y el problema a modelar. A menor n más agresiva es la estrategia.
- **Parámetro de regularización λ :** Tal como está implementado en este documento el valor de λ oscilará entre 0,0001 y 0,1, variará en función de la técnica utilizada (*L1*, *L2*, *Weight Decay*...).

⁴Por ejemplo, toscamente podemos decir que con la función de Cross Entropy Variante aprende aproximadamente 6 veces más rápido que la MSE. Este es un resultado muy poco riguroso pero que puede tomarse como punto de partida.

- **Tamaño del mini-lote:** Este es un hiperparámetro que suele afectar al tiempo de entrenamiento y no tanto al rendimiento del modelo, es por eso que se considera bastante independiente al resto, por lo que se recomienda encontrar algunos valores aceptables (no óptimos) del resto de hiperparámetros, optimizar el tamaño del mini-lote y después optimizar los demás.

Todas estas pautas de selección son manuales y con una estrategia 'a ojo', pero también existen técnicas automáticas que exploran el espacio de hiperparámetros o métodos, si bien manuales, más sistemáticos.

4.4. Optimizadores Avanzados - Variantes del SGD

El algoritmo de descenso de gradiente suele presentar la limitación de que al principio 'avanza' rápidamente pero se va ralentizando con el tiempo, lo que se conoce como el '*Vanishing Gradient Problem*'. Recordemos su fórmula:

$$\theta_{t+1} = \theta_t - \eta \nabla C(\theta_t)$$

El objetivo de estos optimizadores es modificar el algoritmo de descenso de gradiente para aumentar la velocidad de converjencia, evitar mínimos locales mediante técnicas basadas en considerar la variación del gradiente y adaptar la tasa de aprendizaje a las necesidades de cada parámetro.

- **Gradiente Conjugado:**
- **Técnica de la Hessiana:** Sea C la función de coste y $\Delta\theta_t = (\Delta w_1, \Delta w_2, \dots)$ un incremento arbitrario en los parámetros (pesos y biases), definimos el polinomio de Taylor

$$\begin{aligned} C(\theta_t + \Delta\theta_t) &= C(\theta_t) + \sum_j \frac{\partial C}{\partial w_j} \Delta w_j + \frac{1}{2} \sum_{j,k} \Delta w_j \frac{\partial^2 C}{\partial w_j \partial w_k} \Delta w_k + \dots \\ &= C(\theta_t) + \nabla C(\theta_t) \cdot \Delta\theta_t + \frac{1}{2} \Delta\theta_t^T H \Delta\theta_t + \dots \end{aligned}$$

donde H es la matriz Hessiana con $[H]_{i,j} = \frac{\partial^2 C}{\partial w_j \partial w_i}$. Truncamos por el segundo término

$$C(\theta_t + \Delta\theta_t) \approx C(\theta_t) + \nabla C(\theta_t) \cdot \Delta\theta_t + \frac{1}{2} \Delta\theta_t^T H \Delta\theta_t$$

Viendolo como una función respecto de $\Delta\theta_t$, si la matriz Hessiana es definida positiva (formalmente es que $x^T H x > 0 \forall x \in \mathbb{R}^n$, intuitivamente es que tenga forma convexa, i.e., de valle y no de montaña ni de silla) podemos hallar el valor de $\Delta\theta_t$ que minimice $C(\theta_t + \Delta\theta_t)$.

$$\frac{\partial C(\theta_t + \Delta\theta_t)}{\partial \Delta\theta_t} = \nabla C(\theta_t) + H \Delta\theta_t = 0 \implies \Delta\theta_t = -\nabla C(\theta_t) \implies \Delta\theta_t = -H^{-1} \nabla C(\theta_t)$$

donde H^{-1} existe por ser H definida positiva y se ha utilizado la propiedad $\frac{\partial}{\partial x_i} (\frac{1}{2} x^T H x) = \frac{\partial}{\partial x_i} (\frac{1}{2} \sum_j \sum_k x_j H_{jk} x_k) = \frac{1}{2} (\sum_j x_j H_{ji} + \sum_k H_{ik} x_k) = \sum_j H_{ji} x_j = (Hx)_i$ por ser H simétrica.

Esto es que $\Delta\theta_t = -H^{-1} \nabla C(\theta_t)$ es el incremento que más hará disminuir la función de coste. Por tanto, actualizaremos los parámetros de la forma $\theta_{t+1} = \theta_t + \Delta\theta_t$, i.e.,

$$\theta_{t+1} = \theta_t - H^{-1} \nabla C(\theta_t)$$

Normalmente la técnica Hessiana alcanza los mínimos en menos iteraciones que el SGD, esto es porque tiene en cuenta cómo varían los gradientes (segundas derivadas). Sin embargo, el coste de operar con matrices (sobre todo el cálculo de la inversa) tan grandes hace que no sea rentable utilizar esta técnica en redes neuronales complejas.

- **Momento:** La intuición subyacente a este optimizador se basa en que múltiples descensos previos en la misma dirección deberían conducir a largos descensos posteriores, i.e., que debería existir cierta inercia (equivalente al momento físico que adquiere una bola al caer) que nos de una idea sobre cómo cambian los gradientes, al igual que con la Hessiana pero con una implementación más sencilla.

La formulación matemática del algoritmo de Momento es la siguiente:

Inicializacion : $v_0 = 0$

$$\begin{aligned} v_{t+1} &= \beta v_t + \eta \nabla \mathcal{C}(\theta_t) \\ \theta_{t+1} &= \theta_t - v_{t+1} \end{aligned}$$

donde

- v_t es la velocidad/momento que 'memoriza' los gradientes de iteraciones anteriores.
- $0 < \beta < 1$ es un hiperparámetro llamado coeficiente de momento o de decaimiento exponencial que regula lo influyente que es la velocidad/momento en el descenso, intuitivamente podríamos pensar en él como la fricción del movimiento, *típicamente toma valores entre 0,9 y 0,999*.

También es común encontrarlo de la siguiente forma

$$\begin{aligned} v_{t+1} &= \beta v_t + (1 - \beta) \nabla \mathcal{C}(\theta_t) \\ \theta_{t+1} &= \theta_t - \eta v_{t+1} \end{aligned}$$

simplemente habría que reescalar η en $1/(\beta - 1)$. Así, una posible interpretación es la descomposición del descenso en 2 'pasos', uno donde se avanza en dirección al gradiente de la iteración actual y otro donde se tiene en cuenta la 'inercia' de anteriores gradientes. Los factores de ponderación de ambos pasos (β y $\beta - 1$) limitan la magnitud del descenso.

Si sustituimos sucesivamente el factor v_t observamos que realmente estamos tomando una **media ponderada (exponencialmente) de los gradientes** de iteraciones anteriores (de ahí proviene el nombre de β), donde a mayor lejanía menor factor de ponderación.

$$\begin{aligned} v_{t+1} &= \beta v_t + (1 - \beta) \nabla \mathcal{C}(\theta_t) \\ v_{t+1} &= \beta (\beta v_{t-1} + (1 - \beta) \nabla \mathcal{C}(\theta_{t-1})) + (1 - \beta) \nabla \mathcal{C}(\theta_t) \\ &= \beta^2 v_{t-1} + \beta(1 - \beta) \nabla \mathcal{C}(\theta_{t-1}) + (1 - \beta) \nabla \mathcal{C}(\theta_t) \\ &\vdots \\ v_{t+1} &= \sum_{k=0}^t \beta^k (1 - \beta) \nabla \mathcal{C}(\theta_{t-k}) \end{aligned}$$

Sin embargo, este optimizador puede descontrolarse un poco, de hecho, existen alternativas que, basándose en la misma idea, intentan controlar el descenso.

- **RMSprop**: Muchas veces ocurre que hay una gran diferencia entre los gradientes de distintos parámetros, provocando que primero se optimicen unos y luego otros, siendo esta segunda fase muy lenta, sería ideal poder dar 'largos pasos' en dirección de unos parámetros y pequeños en dirección de otros. RMSprop (Root Mean Squared Propagation) permite adaptar la tasa de aprendizaje para cada parámetro separadamente, haciéndola vectorial. Esto último es lo que se conoce como **tasa de aprendizaje adaptativa**.

La formulación matemática del algoritmo RMSprop es la siguiente

Inicializacion : $v_0 = 0$

$$\begin{aligned} v_{t+1} &= \beta v_t + (1 - \beta) (\nabla \mathcal{C}(\theta_t))^2 \\ \theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{v_{t+1} + \epsilon}} \nabla \mathcal{C}(\theta_t) \end{aligned}$$

donde todas las operaciones son elemento a elemento y

- v_t es la velocidad/momento.

- $0 < \beta < 1$ es el coeficiente de momento o de decaimiento exponencial, *típicamente toman valores entre 0,9 y 0,999 respectivamente.*
- $\epsilon > 0$ es el coeficiente de amortiguamiento para evitar la división por 0, *típicamente entre 10^{-8} y 10^{-10} .*
- $\frac{\eta}{\sqrt{v_{t+1} + \epsilon}}$ es la tasa de aprendizaje efectiva.

Así,

$\nabla \mathcal{C}(\theta_t)$ grande $\rightarrow v_{t+1}$ grande \rightarrow tasa de aprendizaje efectiva pequeña

y de forma contraria ocurre con los gradientes pequeños.

- **Adam:** Adam (Adaptive Moment Estimation) combina las ideas de momento y RMSprop, manteniendo un promedio exponencialmente decaído de los gradientes anteriores y sus cuadrados, **ajustando así la tasa de aprendizaje** para cada parámetro.

Inicialización : $v_0 = m_0 = 0$

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1 - \beta_1) \nabla \mathcal{C}(\theta_t) \\ v_t &= \beta_2 v_{t-1} + (1 - \beta_2) (\nabla \mathcal{C}(\theta_t))^2 \\ \hat{m}_t &= \frac{m_t}{1 - \beta_1^t} \\ \hat{v}_t &= \frac{v_t}{1 - \beta_2^t} \\ \theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{\hat{v}_t + \epsilon}} \hat{m}_t \end{aligned}$$

donde todas las operaciones son elemento a elemento y

- m_t y v_t son los vectores promedios móviles de primer y segundo momento de los gradientes.
- \hat{m}_t y \hat{v}_t son los vectores de las estimaciones corregidas de los momentos.
- β_1 y $\beta_2 > 0$ son los parámetros de decaimiento exponencial para los momentos, típicamente entre 0,9 y 0,999.
- $\epsilon > 0$ es el coeficiente de amortiguamiento, típicamente entre 10^{-8} y 10^{-10} .

La actualización de los parámetros θ en Adam consiste en calcular una media ponderada de los gradientes (m_t) y una media ponderada de los cuadrados de los gradientes (v_t), ajustando ambos promedios para tener en cuenta el hecho de que al inicio los gradientes están sesgados hacia 0 (\hat{m}_t y \hat{v}_t), sobre todo en las primeras iteraciones. Luego, se ajustan los parámetros θ utilizando estas medias corregidas, permitiendo un paso de aprendizaje adaptativo y eficiente.

Unos inconvenientes de este optimizador son que es el más espacialmente ineficiente de los que hemos visto y que, si bien los valores de $\beta_1 = 0,9$ y $\beta_2 = 0,999$ funcionan bien en la mayoría de los casos, es muy sensible a la tasa de aprendizaje η .

También destacar que no es recomendable utilizarlo conjuntamente a la regularización $L2$, pues suele llevar a tasas de aprendizaje efectivas extremas. Como alternativa el *Weight Decay* da mejores resultados, quedando el último paso tal que así

$$\theta_{t+1} = \theta_t - \left(\frac{\eta}{\sqrt{\hat{v}_t + \epsilon}} \hat{m}_t + \lambda \cdot W_t \right)$$

4.5. Learning Rate Warmup

El 'learning rate warmup' es una técnica utilizada en el entrenamiento de redes neuronales que consiste en aumentar gradualmente la tasa de aprendizaje desde un valor pequeño hasta el valor deseado η durante las primeras etapas del entrenamiento. Esta estrategia se implementa para mitigar problemas como el 'early over-fitting' en conjuntos de datos altamente diferenciados, esto es que cuando los datos incluyen un grupo de observaciones relacionadas con características fuertes, el modelo puede sesgarse hacia esas características al inicio del entrenamiento, o peor aún, hacia características incidentales no

relacionadas. El warmup reduce este efecto de primacía de los ejemplos iniciales, lo que permite que el modelo no necesite varias épocas adicionales para 'desaprender' esos sesgos iniciales, sino unas pocas (generalmente 1) de calentamiento para directamente no aprenderlos.

En un proceso de warmup de n épocas la tasa de aprendizaje se incrementa linealmente siguiendo la secuencia $\{\eta_n\} = \frac{\eta}{n}, \frac{2\eta}{n}, \dots, \frac{(n-1)\eta}{n}, \eta$. Es decir, en la i -ésima época de las n primeras la tasa de aprendizaje será

$$\frac{i \cdot \eta}{n}$$

para luego, a partir de la época n , estabilizarse en η . Esto significa que la(s) primera(s) iteración(es) recibe(n) solo una fracción n -ésima del efecto del sesgo inicial, balanceando su influencia de manera efectiva.

5. Preparación y carga de los datos

A la hora de entrenar una red neuronal una parte fundamental son los datos de entrenamiento y verificación. Usualmente se obtienen de una fuente en la que no tienen porqué tener el formato adecuado para tu red neuronal, es por ello que conviene cargarlos y preprocesarlos y formatearlos para que se ajusten a tu red. Generalmente se divide el dataset en dos grupos: los datos de entrenamiento y los datos de verificación (se suele usar la proporción 80 % - 20 %). No obstante, también es interesante hacer dividirlos en 3 grupos: datos de entrenamiento, datos de verificación y datos de validación, donde estos últimos se utilizarán para elegir los hiperparámetros del modelo. La razón por la que no utilizamos un nuevo conjunto de datos es para no 'overfittear' los hiperparámetros basandose en los datos de verificación. Siguiendo esta lógica necesitaríamos una secuencia infinita de conjuntos de datos, pero en la práctica 3 suelen ser suficientes.

El efecto de unos buenos datos de entrenamiento preprocesados es enorme, al fin y al cabo la complejidad aprendida por la red viene dada por sus datos de entrenamiento

$$\text{Algoritmo sofisticado} \leq \text{Algoritmo simple} + \text{Datos de entrenamiento de calidad}$$

6. Redes Neuronales Convolucionales (CNN)

Las redes neuronales convolucionales (CNN) son una particularización de una red neuronal general utilizadas para el procesamiento de imágenes y basadas en tres ideas: Campos Receptivos Locales, Pesos Compartidos y Agrupamiento.

6.1. Campos Receptivos Locales y Pesos Compartidos

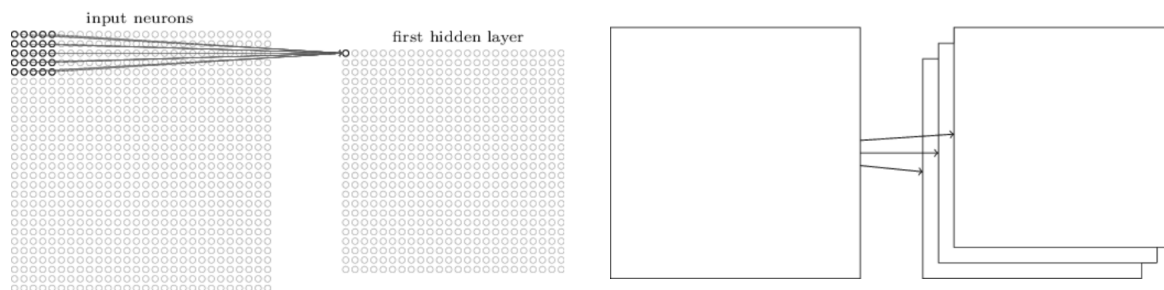
La idea novedosa que introducen las redes neuronales convolucionales es la correlación espacial de los píxeles de una imagen: antes se veían las capas como un vector lineal, cada neurona estaba conectada a todas las neuronas de la capa anterior y cada conexión tiene un peso único, ahora les damos un significado espacial viéndolas como una **matriz**, las neuronas están conectadas con una pequeña región de las neuronas de la capa anterior y los pesos de una capa a otra son compartidos.

Este conjunto de pesos compartidos (junto con el bias se conoce como **filtro o núcleo**) representados como una matriz se aplica a todas las posiciones de la imagen y representa el concepto de campo receptivo local.

Matemáticamente, sea $a^{(l-1)}$ la matriz de la capa de entrada de tamaño $m \times n$ y F la matriz filtro de tamaño $f \times f$, la salida $a^{(l)}$ de la capa convolucional (mapa de característica) de tamaño $(m - f + 1) \times (n - f + 1)$ es

$$[a^{(l)}]_{i,j} = \sigma(b + (a^{(l-1)} * F)_{i,j}) = \sigma \left(b + \sum_{m=0}^{f-1} \sum_{n=0}^{f-1} a_{i+m,j+n}^{(l-1)} \cdot F_{m,n} \right)$$

Por cada capa convolucional, esta operación se repite k veces, cada vez con un filtro distintos, dando lugar a k 'capas de salida' distintas a las que llamamos **mapas de característica**.

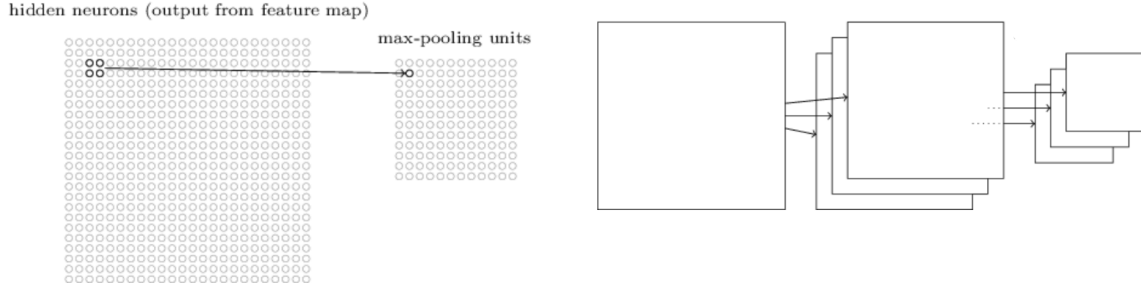


Estas características presentan las siguientes ventajas:

- **Reducción de la Complejidad Computacional:** Al limitar el número de conexiones y compartir pesos y biases, las CNN reducen significativamente la cantidad de parámetros que necesitan ser entrenados, lo que a su vez reduce la complejidad computacional y el riesgo de sobreajuste.
- **Detección de Características Locales:** Los campos receptivos permiten a las CNN detectar características locales (bordes, texturas...) de manera más eficiente. Estas características locales se combinan en capas posteriores para formar patrones y representaciones más complejas y de alto nivel de la imagen.
- **Invariancia a la Traducción:** Debido a que las mismas características pueden aparecer en diferentes partes de una imagen, las CNN son inherentemente invariantes a la traducción. Esto significa que pueden reconocer un objeto sin importar dónde aparezca en la imagen.
- **Reducción del Overfitting:** Los pesos compartidos hacen que los filtros se vean obligados a aprender de toda la imagen, esto hace menos probable que se sobreajusten patrones locales de los datos de entrenamiento, haciendo que *no sea tan necesario aplicar técnicas de regularización* como el dropout a las capas convolucionales.

6.2. Agrupamiento (Pooling)

El agrupamiento tiene como objetivo reducir la dimensionalidad de los mapas de características, controlar el sobreajuste y facilitar el posterior tratamiento. Este proceso se realiza típicamente después de las capas convolucionales tomando cada mapa de característica y condensándolo. Para ello, se divide cada mapa de característica en 'regiones de agrupación' (normalmente no superpuestas) cada una de las cuales dará lugar a una neurona.



Hay varios tipos de agrupamiento, pero los más comunes son el agrupamiento máximo (max pooling) y el agrupamiento promedio (average pooling).

- **Agrupamiento Máximo:** toma el valor máximo de cada región del mapa de características, lo que permite conservar las características más prominentes. Matemáticamente, para una región de entrada R de tamaño $r \times r$ en el mapa de característica A , el valor agrupado P es

$$P = \max_{(i,j) \in R} A_{i,j}$$

- **Agrupamiento Promedio,** se calcula el promedio de los valores en cada región, suavizando las características. Matemáticamente, para una región de entrada R de tamaño $r \times r$ en el mapa de característica A , el valor agrupado P es

$$P = \frac{1}{|R|} \sum_{(i,j) \in R} A_{i,j}$$

6.3. Longitud de Paso (Stride) y Relleno (Padding)

Realmente tanto las capas convolucionales básicas como las de agrupamiento son instancias de un tipo general de capa caracterizada por el factor de **longitud de paso (stride)** $S = (S_H, S_W)$, el cual hace referencia al 'desplazamiento', tanto horizontal como vertical, del *kernel*.

Otra limitación de las CNN tal como están expuestas hasta ahora es la pérdida de información de los píxeles de los bordes, sobre todo a medida que concatenas diversas capas. Para solucionar esto se propone el **relleno (padding)** $P = (P_H, P_W)$ consistente en añadir píxeles blancos (a 0) en los bordes de la imagen.

Tal como se ha explicado, las capas convolucionales básicas tendrían una longitud de paso $S = (1, 1)$ y las capas de agrupamiento $S = (r, r)$. Ambas tendrían un relleno de $P = (0, 0)$.

6.4. Estructura General

Estructura *in-layer*

Suponiendo una **capa convolucional simple** con k_I mapas de característica de entrada, la aplicación de cada uno de los k_O kernels/filtros se realiza de forma conjunta a través de todas los mapas de característica, considerando un kernel de dimensión $f_H \times f_W \times k_I$, i.e., los filtros tienen una profundidad igual al nº de mapas de características de la capa de entrada. Así, cada filtro da lugar a un mapa de característica diferente, resultando en k_O de estos.

Por otro lado, suponiendo una **capa convolucional de agrupamiento** con k_I mapas de característica de entrada, existe un único kernel de tamaño $f_H \times f_W$ que se aplica independientemente a todas los mapas, dando lugar a k_I mapas de característica de salida. En el fondo es una particularización del caso anterior donde el filtro i -ésimo es un tensor con la matriz i -ésima con contenido y el resto a 0.

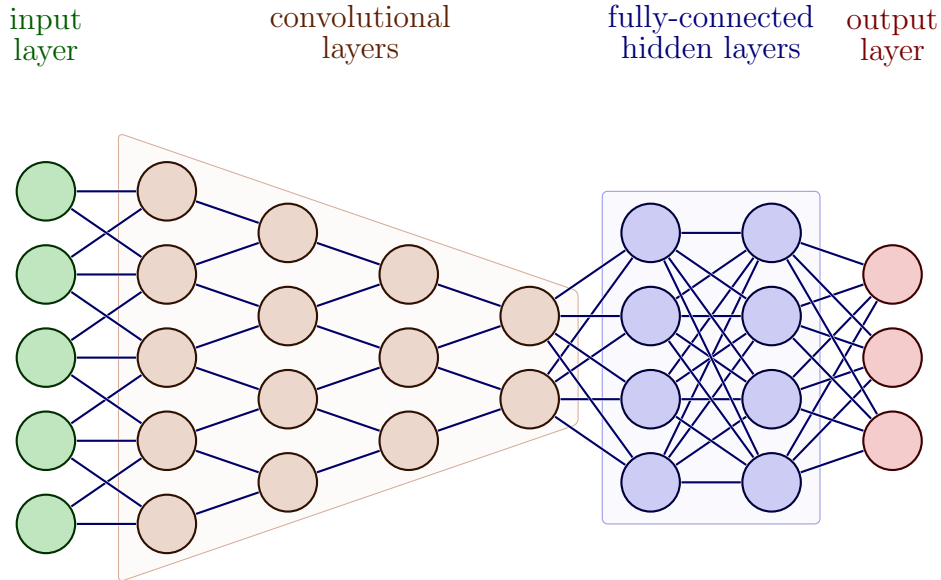
Resumiendo, una capa convolucional genérica se caracteriza por: una capa de entrada de tamaño $m \times n \times k_I$ a la que le aplicas k_O filtros de tamaño $f_H \times f_W \times k_I$ con una longitud de paso $S = (S_H, S_W)$ y un relleno de $P = (P_H, P_W)$ resulta en un tensor de salida de tamaño

$$\left(\frac{m + 2 \cdot P_H - f_H}{S_H} + 1 \right) \times \left(\frac{n + 2 \cdot P_W - f_W}{S_W} + 1 \right) \times k_O$$

i.e., k_O mapas de característica.

Estructura *out-layer*

La capa de entrada de una CNN suele ser una (b/n) o tres (color) matrices, posteriormente se estructuran una secuencia de capas convolucionales simples seguidas de unas de agrupamiento. Por último se implementan capas 'fully-connected' que adecuan la salida de la red a la estructura deseada. Podemos entender las capas convolucionales como detectores de patrones espaciales y las capas 'fully-connected' detectores de patrones más abstractos.



A. Gradientes Inestables

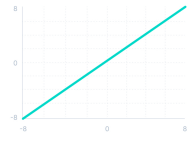

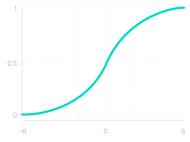
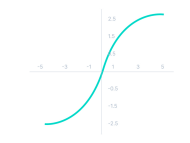
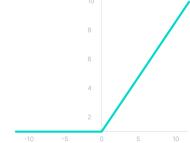
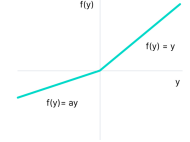
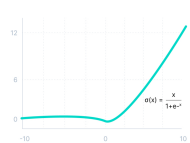
Cuando utilizamos redes neuronales profundass con muchas capas, podemos comprobar experimentalmente la diferencia de gradientes entre distintas capas. Como ya sabemos, $\delta_j^{(l)} = \partial C / \partial b_j^{(l)}$, así, tomamos $||\delta^{(l)}||$ como la 'velocidad de convergencia' o 'tamaño del gradiente' de la capa l y observamos que a mayor l mayor es el gradiente.

$$Unstable Gradient Problem = Vanishing Gradient Problem + Exploding Gradient Problem$$

es decir, esta inestabilidad se traduce en dos principales cuestiones a tratar: el *Vanishing Gradient Problem* en las primeras capas y el *Exploding Gradient Problem* en las últimas capas.

B. Funciones de Activación y de Coste

B.1. Funciones de Activación

Funciones Lineales			
Nombre	Fórmula	Gráfica	Ventajas y Desventajas
Lineal	$f(x) = x$		Ventajas: Sencilla, útil en regresión Desventajas: No introduce no linealidades. No apta para capas ocultas (colapso) ni para la retropropagación.
Step	$H(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$		Ventajas: Simple, clara separación binaria. Útil para tareas de clasificación binaria. Desventajas: No diferenciable, no apta para retropropagación
Funciones No Lineales			
Nombre	Fórmula	Gráfica	Ventajas y Desventajas
Sigmoid	$\sigma(x) = \frac{1}{1+e^{-x}}$		Ventajas: Suaviza valores, útil para probabilidades Desventajas: Gradientes pequeños, problema de 'Vanishing Gradient' y activaciones solo positivas.
Softmax	$\sigma(x)_i = \frac{e^{x_i}}{\sum_j e^{x_j}}$	Composición de sigmoides	Ventajas: Salidas probabilísticas, útil en clasificación multiclase Desventajas: Computacionalmente costosa, no adecuada para capas ocultas
Tanh	$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$		Ventajas: Mejora sobre Sigmoide con gradientes más fuertes y simétricos en cuanto al signo. Desventajas: Aún puede sufrir de 'Vanishing Gradient'.
ReLU	$ReLU(x) = \max(0, x)$		Ventajas: Computacionalmente eficiente, alta velocidad de convergencia Desventajas: Problema de 'Dead Neurons'.
Parametric ReLU	$P-ReLU(x) = \max(ax, x)$		Ventajas: Mitiga el problema de 'Dead Neurons' (una particularización es el Leaky-ReLU con a = 0,1) Desventajas: Introduce un hiperparámetro
Swish	$Swish(x) = x \cdot \sigma(x)$		Ventajas: Suaviza la función de activación y no quita relevancia a los valores negativos (a los extremos, $x \rightarrow \infty$, los satura a 0) Desventajas: Puede ser computacionalmente más costoso

Cuadro 1: Funciones de activación y sus propiedades

Muchas veces, sobre todo en tareas de clasificación, resulta conveniente que las activaciones de la última capa formen una distribución de probabilidad. Para ello se utiliza, en la última capa, una función de activación un poco especial⁵: la función **Softmax**, la cual viene descrita por la siguiente ecuación

$$\sigma(z_j^{(L)}) = \frac{e^{z_j^{(L)}}}{\sum_k e^{z_k^{(L)}}}$$

Tiene la capacidad de transformar $a^{(L)}$ en una distribución de probabilidad por las siguientes propiedades:

- **Positividad**: elevar los valores $z_k^{(L)}$ a la potencia de e garantiza la positividad de las salidas.
- **Suma de 1**: $\sum_{j=1}^{n_L} \sigma(z_j^{(L)}) = \frac{\sum_j e^{z_j^{(L)}}}{\sum_k e^{z_k^{(L)}}} = 1$
- **Sensibilidad coherente**: Aumentar de forma aislada una activación $z_j^{(L)}$ aumenta $\sigma(z_j^{(L)})$ (por la monotonía creciente de σ) y disminuye el resto de $\sigma(z_k^{(L)})$.

¿De donde viene el nombre 'softmax'?

Por simplificar omitiremos el $^{(L)}$ en la notación:

Imaginemos una generalización

$$f_c(z_j) = \frac{e^{cz_j}}{\sum_k e^{cz_k}}$$

donde con $c = 1$ obtenemos la función softmax. Se demuestra fácilmente que sigue teniendo la propiedad de transformar conjuntos de valores en distribuciones de probabilidad.

Calculamos como paso previo el límite del denominador cuando $c \rightarrow \infty$ teniendo en cuenta que $z_i := \max(z) = z_{i_1} = \dots = z_{i_m}$

$$\begin{aligned} \lim_{c \rightarrow \infty} \sum_k e^{cz_k} &= \lim_{c \rightarrow \infty} e^{cz_i} \cdot \sum_k e^{c(z_k - z_i)} = \lim_{c \rightarrow \infty} e^{cz_i} \cdot \left(\sum_{z_k = z_i} 1 + \sum_{\cancel{z_k \neq z_i}} e^{c(z_k - z_i)} \right) = \\ &= \lim_{c \rightarrow \infty} e^{cz_i} \cdot (m + 1) \end{aligned}$$

Ahora tomemos el límite de la función

$$\lim_{c \rightarrow \infty} f_c(z_j) = \lim_{c \rightarrow \infty} \frac{e^{cz_j}}{\sum_k e^{cz_k}} = \lim_{c \rightarrow \infty} \frac{e^{c(z_j - z_i)}}{m + 1} = \begin{cases} \frac{1}{m+1} & \text{si } z_i = z_k \\ 0 & \text{si } z_i < z_k \end{cases}$$

Con esto deducimos que la función crea una distribución de probabilidad distribuyéndola entre los valores máximos. Con esto cobra sentido que llamemos a la función con $c = 1$ como la forma 'soft' de la función máximo, i.e., softmax.

El parámetro c regula lo 'afilada' que es la transformación, de hecho, el factor $\tau = 1/c$ se le denomina *temperatura*:

- Mayor c / menor $\tau \rightarrow$ preferencia por valores máximos \rightarrow menor riesgo.
- Menor c / mayor $\tau \rightarrow$ distribución igualitaria \rightarrow mayor riesgo.

La no-linealidad es un requisito fundamental de las neuronas ubicadas en capas ocultas (si no lo fueran toda la red colapsaría a una neurona), las cuales suelen compartir función de activación. Por otro lado, las neuronas de la capa de salida pueden ser lineales y la de la capa de entrada siempre lo es, i.e., la función identidad, por lo que se suele omitir.

⁵Decimos que es especial porque depende de las activaciones de todas las neuronas de la capa final, algo que no suele ocurrir con el resto de activaciones. Esta propiedad recibe el nombre de *no-localidad*

B.1.1. Elección de Función de Activación

Capas de salida	<ul style="list-style-type: none">Regresión - LinealClasificación Binaria - Step function o SigmoideClasificación Multiclase - SoftmaxClasificación Multietiqueta - SigmoideLa función ReLU no suele ajustarse correctamente.	Capas ocultas	<ul style="list-style-type: none">ReLU - Es la opción base, suele empezarse con este y luego ir ajustando o cambiando (útil en CNN).Tanh o Sigmoide - También típica pero especialmente la sigmoide presenta el problema de 'Vanishing Gradient' (útil en RNN).
------------------------	---	----------------------	--

B.2. Funciones de Coste

Para definir las funciones de coste usaremos la siguiente notación: $a^{(L)}$ el output de la red, \hat{y} el target esperado y n_L el nº de neuronas de la última capa.

Durante todo este apéndice se definirán las funciones de coste para una sola muestra x_i . La función de coste para un conjunto de muestras $x = \{x_1, x_2, \dots, x_n\}$ se calculará promediando todas las muestras.

$$C_x = \frac{1}{n} \sum_{x_i} C_{x_i}$$

De cada funcion poner su formula y la formula de la derivada parcial con respecto a $a_j^{(L)}$.

B.2.1. Funciones para Regresión

Al utilizarse en tareas de regresión, la funcion de activación de la última capa suele ser lineal.

■ Mean Squared Error (MSE)

El error de una sola muestra sería

$$\frac{1}{n_L} \sum_{j=1}^{n_L} (a_j^{(L)} - \hat{y}_j)^2$$

No obstante, para simplificar el cálculo de las derivadas es común añadir un 2 al denominador y eliminar el $1/n_L$. Además, lo expresamos de forma más compacta como norma euclidiana ⁶:

$$C_{x_i} := \frac{1}{2} \|a^{(L)} - \hat{y}\|_2^2$$

Así, las derivadas parciales quedan como

$$\frac{\partial C_{x_i}}{\partial a^{(L)}} = a^{(L)} - \hat{y}$$

Observaciones:

- Simplicidad y fácil implementación.
- Sensibilidad a los datos atípicos o extremos.

■ Mean Absolute Error (MAE)

El error de una sola muestra sería

$$\frac{1}{n_L} \sum_{j=1}^{n_L} |a_j^{(L)} - \hat{y}_j|$$

⁶Se pierde el 'Mean' del nombre pero bueno ns

De forma similar al MSE modificamos la expresión quitando el $1/n_L$

$$C_{x_i} := \|a^{(L)} - \hat{y}\|_1$$

Así, las derivadas parciales quedan como

$$\frac{\partial C_{x_i}}{\partial a^{(L)}} = \text{sgn}(a^{(L)} - \hat{y})$$

Observaciones:

- Soluciona el problema de los datos atípicos
- Puede presentar errores en el cálculo del gradiente cuando nos aproximemos a la solución, pues el valor absoluto no es diferenciable en $x = 0$.

■ **Huber Loss**

La función de Huber sobre un escalar viene dada por la siguiente definición

$$L_\delta(a_j^{(L)}, \hat{y}_j) := \begin{cases} \frac{1}{2}(a_j^{(L)} - \hat{y}_j)^2 & \text{si } |a_j^{(L)} - \hat{y}_j| \leq \delta \\ \delta \cdot (|a_j^{(L)} - \hat{y}_j| - \frac{1}{2}\delta) & \text{resto} \end{cases}$$

donde δ es el punto de transición de un coste cuadrático a uno lineal.

Ahora, definimos su aplicación sobre un vector como la suma de las aplicaciones sobre cada uno de los componentes

$$C_{x_i} = L_\delta(a^{(L)}, \hat{y}) := \sum_{j=1}^{n_L} L_\delta(a_j^{(L)}, \hat{y}_j)$$

Así, las derivadas parciales quedan como

$$\frac{\partial C_{x_i}}{\partial a^{(L)}} = \begin{cases} a^{(L)} - \hat{y} & \text{si } |a^{(L)} - \hat{y}| \leq \delta \\ \delta \cdot \text{sgn}(a^{(L)} - \hat{y}) & \text{resto} \end{cases}$$

NOTA: Todas las operaciones y condiciones en las funciones a trozos están entendidas elemento a elemento.

Observaciones:

- En cuanto a robustez, el coste de Huber supone una mejora respecto de las anteriores funciones pues le resta importancia a los datos extremos sin perder la continuidad y diferenciable (cuadrático como MSE entorno a su mínimo en $a^{(L)} - \hat{y}$ pero lineal como MAE más allá).

B.2.2. Funciones para Clasificación

En las redes neuronales de clasificación, se suele aplicar a la última capa la función de activación softmax, lo que asegura que el output de la red represente una distribución de probabilidad discreta de n_L sucesos e \hat{y} la distribución de probabilidad esperada (suele ser 1 en un suceso y 0 en el resto).

■ **Divergencia de Kullback–Leibler**

Sean P y Q dos distribuciones/funciones de probabilidad sobre una variable aleatoria discreta en un espacio de sucesos $\Omega = \{1, \dots, n\}$, definimos la divergencia KL como

$$D_{KL}(P||Q) := \sum_{i=1}^n P(i) \cdot \ln \frac{P(i)}{Q(i)}$$

donde $P(i)$ y $Q(i)$ indican la probabilidad del suceso i para P y Q respectivamente. ⁷ Veamos de donde surge esta fórmula.

⁷Si la expresión $0 \cdot \ln(0)$ aparece en la fórmula, se interpreta como 0

Intuitivamente, cuando pensamos en la distancia entre dos distribuciones de probabilidad pensamos en qué medida se asignan probabilidades similares a sucesos similares. Dicho de otro modo, podemos comparar las probabilidades de ambas distribuciones de que sucedan unas mismas observaciones.

$$\frac{P(\text{Observaciones})}{Q(\text{Observaciones})}$$

Cuanto más se acerque este ratio a 1, mayor será la similitud entre probabilidades. Pensemos ahora en un conjunto de $N = N_1 + \dots + N_n$ observaciones genéricas generadas por la distribución P , donde N_i es el número de veces que ha ocurrido el suceso i , y relacionemoslo con la probabilidad de que la distribución Q genere esas mismas observaciones. Así, tenemos que

$$\frac{P(\text{Observaciones})}{Q(\text{Observaciones})} = \frac{\sum_{i=1}^n P(i)^{N_i}}{\sum_{i=1}^n Q(i)^{N_i}}$$

Normalizamos elevando a $1/N$ (para medir similitud por observación, no por secuencia de observaciones) y aplicamos el logaritmo (notemos que no modifica la intuición del procedimiento pues es una función monótona)

$$\ln \left(\frac{\sum_{i=1}^n P(i)^{N_i}}{\sum_{i=1}^n Q(i)^{N_i}} \right)^{\frac{1}{N}}$$

Aplicando propiedades del logaritmo llegamos a

$$\sum_{i=1}^n \frac{N_i}{N} \cdot \ln \left(\frac{P(i)}{Q(i)} \right)$$

Es claro que si $N \rightarrow \infty$, entonces $\frac{N_i}{N} \rightarrow P(i)$, luego la expresión nos queda como

$$\sum_{i=1}^n P(i) \cdot \ln \left(\frac{P(i)}{Q(i)} \right)$$

Así, concluimos que la divergencia KL es una medida natural de 'distancia'⁸ entre distribuciones de probabilidad motivada por la probabilidad de que la segunda distribución genere observaciones producidas por la primera.

Aplicando esta divergencia como una función de coste quedaría

$$C_{x_i} := D_{KL}(\hat{y} || a^{(L)}) = \sum_{j=1}^{n_L} \hat{y}_j \cdot \ln \left(\frac{\hat{y}_j}{a_j^{(L)}} \right)$$

Así, las derivadas parciales quedan como

$$\frac{\partial C_{x_i}}{\partial a^{(L)}} = - \frac{\hat{y}}{a^{(L)}}$$

■ Cross Entropy Cost

La función Cross Entropy se define del siguiente modo

$$C_{x_i} := H(\hat{y}, a^{(L)}) = - \sum_{j=1}^{n_L} \hat{y}_j \cdot \ln(a_j^{(L)})$$

Así, las derivadas parciales quedan como

$$\frac{\partial C_{x_i}}{\partial a^{(L)}} = - \frac{\hat{y}}{a^{(L)}}$$

⁸Coloquialmente hablando, no es una distancia matemática

y $\delta^{(L)}$ queda

$$\delta^{(L)} = \frac{\partial C_{x_i}}{\partial a^{(L)}} \odot \sigma'(z^{(L)}) = -\frac{\hat{y}}{a^{(L)}(1-a^{(L)})} \odot \sigma(z^{(L)}) = -\hat{y} \odot (1 - a^{(L)})$$

Esta función de coste es una sencilla derivación de la divergencia KL . Partiendo de $D_{KL}(\hat{y}||a^{(L)})$, podemos aplicar propiedades de logaritmos hasta llegar a

$$D_{KL}(\hat{y}||a^{(L)}) = \underbrace{\sum_{j=1}^{n_L} \hat{y}_j \cdot \ln(\hat{y}_j)}_{\text{independiente de } \theta} - \sum_{j=1}^{n_L} \hat{y}_j \cdot \ln(a_j^{(L)}) = k + H(\hat{y}, a^{(L)})$$

Como vemos, la divergencia KL y la función *Cross Entropy* únicamente difieren en una constante, por lo que podemos afirmar que

$$\operatorname{argmin}_{\theta} D_{KL}(\hat{y}||a^{(L)}) = \operatorname{argmin}_{\theta} H(\hat{y}, a^{(L)})$$

es decir, que minimizar una es equivalente a minimizar otra, de hecho, podemos ver en las expresiones de sus derivadas parciales que sus gradientes son iguales.

Cuando el número de clases/categorías es 2 se le llama Binary Cross Entropy, y se aplica cuando la capa de salida tiene una única neurona.

Observaciones:

- Rinde especialmente bien cuando la capa de salida forma una distribución de probabilidad.
- Necesidad de aplicar la función softmax para que la capa de salida cumpla con la propiedad requerida.

■ Cross Entropy Cost Variation

A la hora de evaluar el coste de una red neuronal, las activaciones de la capa de salida ($a^{(L)}$) rara vez van a formar una distribución de probabilidad, por lo que, como alternativa a aplicar la función softmax en la última capa, se propone una variación donde la función de coste es la suma de varias funciones de entropía cruzada unineuronales donde cada neurona de salida se trata como si formara una distribución de probabilidad de dos elementos: su activación $a_j^{(L)}$ y su complemento $1 - a_j^{(L)}$.

$$C_{x_i} := - \sum_{j=1}^{n_L} \hat{y}_j \cdot \ln(a_j^{(L)}) + (1 - \hat{y}_j) \cdot \ln(1 - a_j^{(L)})$$

Aunque en la práctica nuestras redes neuronales no contienen elementos probabilísticos reales y las activaciones no son probabilidades en un sentido estricto, esta interpretación permite generalizar el concepto de entropía cruzada. La fórmula se convierte en una suma de las contribuciones de cada neurona, capturando cómo de bien cada neurona de salida se ajusta a su valor objetivo. Así, las derivadas parciales quedan como

$$\frac{\partial C_{x_i}}{\partial a^{(L)}} = \frac{a^{(L)} - \hat{y}}{a^{(L)} \cdot (1 - a^{(L)})}$$

Este resultado es especialmente interesante porque al calcular $\delta^{(L)}$ nos queda

$$\delta^{(L)} = \frac{\partial C_{x_i}}{\partial a^{(L)}} \odot \sigma'(z^{(L)}) = \frac{a^{(L)} - \hat{y}}{a^{(L)} \cdot (1 - a^{(L)})} \odot \sigma(z^{(L)}) = a^{(L)} - \hat{y}$$

Observaciones:

- No es necesario aplicar la función softmax y elimina el término $\sigma'(z^{(L)})$ en $\delta^{(L)}$, lo que evita el problema de que ralentización del aprendizaje cuando se satura en valores de $\sigma'(z^{(L)})$ pequeños.

■ **Distancia de Hellinger**

La distancia de Hellinger entre 2 vectores se define como sigue

$$C_{x_i} = h(a_j^{(L)}, \hat{y}) := \frac{1}{\sqrt{2}} \cdot \sqrt{\sum_{j=1}^{n_L} (\sqrt{a_j^{(L)}} - \sqrt{\hat{y}_j})^2}$$

$$C_{x_i} := \frac{1}{\sqrt{2}} \cdot \|\sqrt{a^{(L)}} - \sqrt{\hat{y}}\|_2$$

Vemos que es una distancia por herencia de la norma euclídea. Es idóneo para tareas de clasificación donde el output está acotado entre 0 y 1 porque, si no lo estuviera, las raíces cuadradas no existirían.

Así, las derivadas parciales quedan como

$$\frac{\partial C_{x_i}}{\partial a^{(L)}} = \frac{\sqrt{a^{(L)}} - \sqrt{\hat{y}}}{\sqrt{2a^{(L)}} \cdot \|\sqrt{a^{(L)}} - \sqrt{\hat{y}}\|_2}$$