Apuntes De Clase Semana 8
Inteligencia Artificial

Perez Picado Esteban, *2021046572*

***Abstract*—This document presents detailed lecture notes on key concepts in neural networks, focusing on activation functions and the backpropagation algorithm. The notes cover recent developments in open-source AI models, such as Yama 4 by Meta, and practical tools like Hugging Face and LM Studio for running local models. In terms of neural network theory, we explore the importance of activation functions, including sigmoid, tanh, ReLU, Leaky ReLU, and PReLU, discussing their roles in introducing non-linearity and addressing issues like vanishing gradients. The document then delves into the backpropagation algorithm, detailing the forward and backward passes, using computational graphs for visualization. It includes a comprehensive derivation of gradients for both single-layer and multi-layer networks, emphasizing the computational complexity as networks grow deeper. These insights lay the groundwork for understanding optimization techniques in deep learning, preparing for future discussions on gradient optimization methods.**

I. NOTICIAS Y NOVEDADES EN INTELIGENCIA ARTIFICIAL

A. Lanzamiento de Yama 4

Meta, el equipo detrás de los modelos *Llama*, recientemente publicó el *Yama 4*, el cual sigue siendo *open source*, permitiendo descargarlo y utilizarlo sin restricciones. Esta nueva generación introduce un enfoque de *mixture of experts*, lo que significa que el modelo está compuesto por varias redes neuronales especializadas (expertos), y un router se encarga de dirigir las entradas hacia el experto más adecuado según la tarea.

El *Yama 4* tiene tres versiones:

- **Scout** y **Maveric**: ya disponibles.
- **Vegemocop**: aún en entrenamiento.

Especificaciones destacadas:

- 16 expertos activos por modelo.
- Hasta 109 billones de parámetros en total (Scout/Maveric).
- El *Vegemocop* proyecta alcanzar 288 billones de parámetros activos y 2 terabytes de parámetros totales.

Estos modelos son capaces de ajustarse para funcionar en una sola GPU *Nvidia H100*, lo que, aunque es un hardware de gama alta, permite ejecutar modelos de esta magnitud.

B. Características de Mixture of Experts

El enfoque de *mixture of experts* permite dividir la carga de procesamiento entre diferentes modelos especializados. Un *router* analiza la entrada y decide cuál experto es el más adecuado para procesarla, reduciendo así el costo computacional y mejorando la eficiencia.

¿Qué es *distillation*?

Es un proceso donde un modelo grande (el modelo padre) enseña a uno más pequeño (modelo hijo), transfiriendo su conocimiento. De este modo, el modelo más pequeño aprende a realizar tareas complejas sin necesitar tanta capacidad computacional. En este contexto, **Maveric** y **Scout** serían los modelos hijos, optimizados mediante *distillation* a partir de un modelo mayor.

Ranking en LM Arena:

En la plataforma *LM Arena*, donde se comparan modelos de lenguaje, *Yama 4* ya se encuentra rankeado junto a modelos como *GPT-4*, *GPT-5*, *Grok 3*, entre otros, diferenciando los *open source* de los propietarios.

C. Herramientas de Hugging Face y LM Studio

Hugging Face: comunidad donde se publican modelos y datasets listos para descargar y utilizar. Ahí ya están disponibles las versiones recientes de *Yama 4*.

LM Studio: aplicación que permite descargar y correr modelos localmente, sin necesidad de conexión a internet.

Ejemplo de uso:

- Puedes correr un modelo con *4 bits quantization*, lo cual reduce el tamaño del modelo (por ejemplo, de 4 GB), aunque sacrificando algo de precisión.
- Puedes ajustar cuánto del modelo cargas en *GPU* o *CPU*, dependiendo de tus recursos.

El profesor compartió su experiencia personal: al correr modelos quantizados a 4 bits en su laptop, pudo manejarlos sin problemas, mientras que modelos con mayor cantidad de bits (más pesados) le resultaron lentos.

Nota: Si tienes una *GPU* decente, puedes experimentar con estos modelos y tener algo similar a un *ChatGPT* local, lo cual es bastante útil si estás sin conexión.

II. ASPECTOS ADMINISTRATIVOS DEL CURSO

A. Cambio en fechas de entrega del proyecto final

La fecha de entrega del proyecto final se movió al **martes 29 de abril**, debido a la **Semana Compu**, en la cual no se pueden realizar evaluaciones.

Además, esa semana no habrá **clases presenciales**, únicamente **clases virtuales**.

This paper was produced by the IEEE Publication Technology Group. They are in Piscataway, NJ.

Manuscript received April 19, 2021; revised August 16, 2021.

B. Modalidad de clases virtuales en Semana Compu

Durante la Semana Compu, el curso se mantendrá en modalidad **virtual**, asegurando que las actividades académicas continúen sin interrupciones a pesar de las restricciones para evaluaciones presenciales.

C. Consideraciones sobre Scikit-Learn y visualización de curvas

Se discutió un aspecto técnico importante: **Scikit-Learn**, la librería recomendada para el proyecto, no permite graficar fácilmente las **curvas de entrenamiento y validación** para monitorear el **overfitting**.

Aunque existen funciones como *learning curves*, estas se enfocan en métricas globales (por ejemplo, **accuracy**) y no permiten seguir detalladamente el progreso por época como en otros frameworks.

Alternativa: En el segundo proyecto se utilizará **PyTorch**, que permite este tipo de visualización de manera más flexible.

Importante: Aunque *Scikit-Learn* es recomendado, puedes utilizar cualquier otra librería que prefieras si encuentras alguna que se ajuste mejor a tus necesidades.

III. FUNCIONES DE ACTIVACIÓN EN REDES NEURONALES

A. Importancia de las funciones de activación

En una red neuronal, después de calcular la combinación lineal de las entradas $(x_1, x_2, ..., x_n)$ y sus respectivos pesos $(w_1, w_2, ..., w_n)$, más un sesgo (b), obtenemos la pre-activación (z):

$$z = \sum_{i=1}^{n} x_i w_i + b$$

Este valor z puede estar en cualquier punto del eje real, desde $-\infty$ hasta $+\infty$. Sin embargo, para que la red aprenda comportamientos complejos, necesitamos **controlar el rango de salida** y añadir **no linealidad**. Sin funciones de activación, la red se comportaría como una simple función lineal, sin importar cuántas capas tenga.

B. ¿Por qué no usar solo funciones lineales?

Si todas las capas aplican solo combinaciones lineales, el resultado final sería otra función lineal:

$$\sum_{i=1}^{n} x_i w_i + b$$

$$w = w - \alpha \frac{\partial L}{\partial w}$$

$$w_1 x + b_1 = h_1$$

$$w_2 h_1 + b_2 = h_2$$

$$w_3 x + b$$

Esto equivale a tener una regresión lineal disfrazada de red neuronal. Por eso es fundamental introducir **funciones de activación no lineales** entre las capas.

C. Función Sigmoide

La función **sigmoide** es una de las más clásicas:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Características:

- Rango de salida: $[0, 1]$
- Siempre positiva y estrictamente creciente.

Problema: Vanishing Gradient. En los extremos (cercanos a 0 o 1), la pendiente es casi cero, lo que hace que las actualizaciones de los pesos sean mínimas y la red aprenda muy lento o se estanque.

D. Tangente hiperbólica (tanh)

La función **tangente hiperbólica** es otra opción popular:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{1}$$

Características:

- Rango de salida: $[-1, 1]$
- Estrictamente creciente, con gradientes más pronunciados en el centro.

Se utiliza mucho en modelos de lenguaje, como las **LSTM** (*Long Short-Term Memory*), redes diseñadas para recordar o olvidar información al procesar secuencias, como en tareas de series temporales o procesamiento de texto.

E. Problema del Vanishing Gradient

Cuando las derivadas de las funciones de activación son muy pequeñas (especialmente en los extremos), el gradiente se va perdiendo capa tras capa. Esto afecta el proceso de **descenso de gradiente**, ya que los pesos apenas se actualizan. A medida que las redes se hacen más profundas, este problema se agrava.

¿Cómo mitigar el Vanishing Gradient?

Batch Normalization: Técnica que **normaliza las salidas de cada capa**, ajustándolas a un rango estándar (cercano a cero). Esto ayuda a mantener los gradientes en un rango manejable durante el entrenamiento, reduciendo los problemas de estancamiento.

F. ReLU (Rectified Linear Unit)

La **ReLU** es probablemente la función de activación más popular en *deep learning*. Su fórmula es simple:

$$ReLU(x) = \max(0, x) \tag{2}$$

¿Cómo funciona?

- Si $x < 0$, devuelve 0.
- Si $x > 0$, devuelve x.

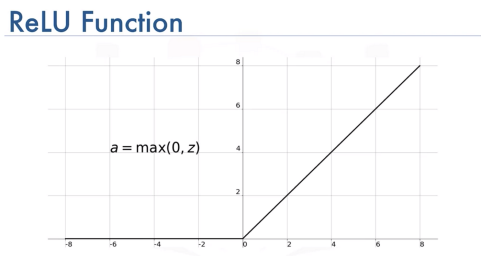

Fig. 1. Gráfico de la función ReLU.

Ventajas:

- Es muy eficiente computacionalmente.
- Introduce **no linealidad** de forma sencilla.
- Se adapta bien a **redes neuronales profundas**, donde la cantidad de cálculos es enorme.

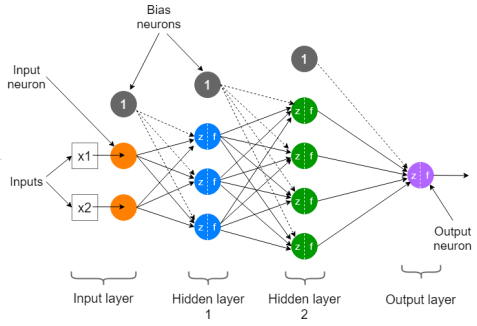

Fig. 2. Diagrama de neuronas ocultas.

Problema de ReLU: Neuronas muertas
Cuando el valor de entrada es negativo, **ReLU** "mata" la activación, propagando ceros hacia las siguientes capas. Esto significa que:

- Las neuronas dejan de aprender si siempre reciben valores negativos.
- Durante el **backpropagation**, las derivadas también son 0, por lo que los pesos de esas neuronas no se actualizan:

$$\frac{\partial L}{\partial W} = 0 \Rightarrow W_{\text{nuevo}} = W_{\text{anterior}}$$

Nota: En $x = 0$, **ReLU** no es derivable, pero en la práctica esto se soluciona eligiendo uno de los lados (izquierda o derecha) arbitrariamente.

G. Leaky ReLU

Para evitar las **neuronas muertas**, surge **Leaky ReLU**, una variante que permite que las entradas negativas tengan una pequeña pendiente:

$$LeakyReLU(x) = \begin{cases} x & \text{si } x > 0 \\ \alpha x & \text{si } x \leq 0 \end{cases} \tag{3}$$

Donde α es un valor pequeño, típicamente 0.01.

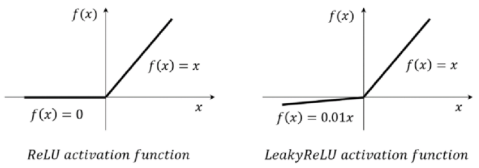

Fig. 3. Grafico de la función Leaky ReLU.

¿Qué gana la red? Aunque la activación sea negativa, los pesos se siguen actualizando, aunque sea ligeramente.

H. Parametric ReLU (PReLU)

PReLU mejora la idea de **Leaky ReLU** al permitir que ese parámetro α se **aprenda durante el entrenamiento**, en lugar de fijarlo manualmente:

$$PReLU(x) = \begin{cases} x & \text{si } x > 0 \\ \alpha x & \text{si } x \leq 0 \end{cases}$$

Ventaja: La red ajusta automáticamente la pendiente en las zonas negativas, adaptándose mejor al problema específico.

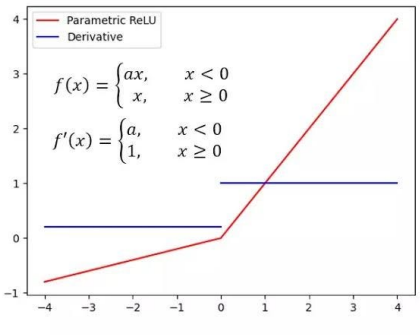

Fig. 4. Grafico de la función Parametric ReLU

I. Softmax y clasificación multiclase

Cuando tenemos una **salida multiclase**, usamos **Softmax** para convertir los resultados de la red (llamados **logits**) en probabilidades:

$$Softmax(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

¿Qué hace?

- Toma los **logits** $(z_1, z_2, ..., z_K)$ y los transforma en valores entre 0 y 1.
- La suma total es 1, representando una **distribución de probabilidad** entre las clases.
- Identifica cuál clase tiene mayor probabilidad (la más alta después de aplicar softmax).

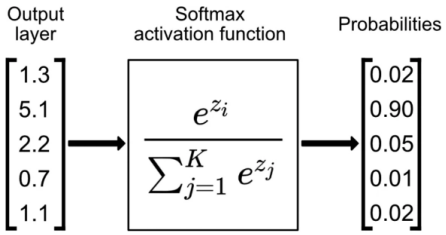

Fig. 5. Diagrama del funcionamiento de Softmax.

J. Resumen práctico: ¿Qué función usar?

- **ReLU**: Primera opción en **capas ocultas** por su simplicidad y eficiencia.
- **Leaky ReLU** o **PReLU**: Úsalas si notas problemas de **neuronas muertas**.
- **Sigmoide** y **tanh**: Fueron populares en los 90, pero ya no se recomiendan tanto por problemas de **vanishing gradient**.
- **Softmax**: Solo en la **capa de salida** para problemas de **clasificación multiclase**.
- Para **problemas de regresión**, en la **capa de salida** se usa una neurona sin función de activación, permitiendo predecir valores continuos (desde $-\infty$ hasta $+\infty$).

K. Función de pérdida: Cross-Entropy Loss

Para entrenar modelos de clasificación multiclase (cuando usamos **Softmax**), la función de pérdida adecuada es **Cross-Entropy Loss**:

$$L_i = -\log\left(P(Y = y_i, X = x_i)\right)$$

$$L_i = -\log\left(\frac{e^{s_k}}{\sum e^{s_j}}\right), \text{ con } j \text{ desde } 1, \ldots, C$$

Donde:

- y_i es la **etiqueta real** (0 o 1).
- $\hat{y}_i$ es la **probabilidad predicha** por la red.

¿Por qué usar Cross-Entropy?

- Penaliza fuertemente las predicciones incorrectas.
- Es **compatible con Softmax**, ya que ambas son funciones estrictamente crecientes, lo que evita comportamientos anómalos en el entrenamiento.

IV. BACKPROPAGATION EN REDES NEURONALES

A. Introducción

1) Forward Propagation: Es el proceso que utilizamos para hacer **predicciones**.

Consiste en tomar la entrada del modelo y pasarla a través de todas las capas, de izquierda a derecha, hasta obtener la salida final (ya sea una etiqueta, una probabilidad, un valor real, etc.).

2) Backpropagation: Este es el proceso de **ajustar los pesos** del modelo calculando gradientes.

Se realiza de derecha a izquierda, propagando el error desde la salida hasta las primeras capas.

El objetivo es saber cuánto influye cada peso en el error total, y así poder ajustarlo para mejorar el rendimiento del modelo.

$$\frac{\partial L}{\partial W_{ji}}$$

Calculamos la derivada parcial de la función de pérdida (L) respecto a cada peso W_{ji}.

B. Representación de operaciones mediante grafos computacionales

Para entender mejor este proceso, podemos representar las operaciones de la red como un **grafo computacional**.

Ejemplo simple:

$$f(x, y, z) = (x + y) \cdot z$$

Se descompone en:

- Suma: $q = x + y$
- Multiplicación: $f = q \cdot z$

Esto permite ver paso a paso cómo fluye la información, y facilita el cálculo de derivadas usando la **regla de la cadena**.

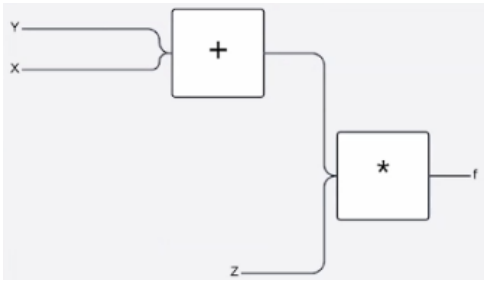

Fig. 6. Diagrama de grafo computacional.

C. Aplicación a una neurona (última capa)

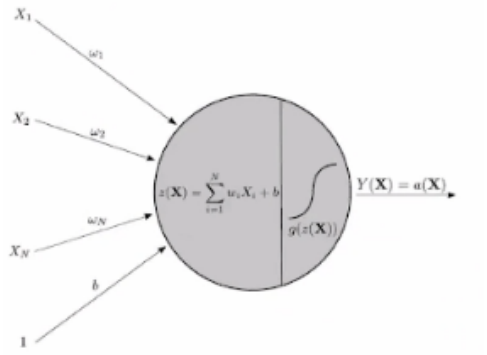

Fig. 7. Función de activación.

Consideramos una sola neurona en la capa de salida:

Con la función:

$$g(x) = \frac{1}{1 + e^{-x}}$$

Pre-activación:

$$z^l = wa^{l-1} + b^l$$

Activación:

$$a^l = g(z^l)$$

Función de costo:

$$L_i = (a^l - y_i)^2$$

1) Backpropagation paso a paso (regla de la cadena): Queremos actualizar $W^{(L)}$ y $b^{(L)}$. Para eso calculamos las derivadas parciales encadenadas:

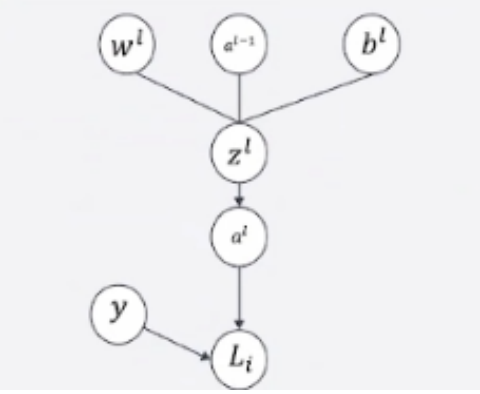

Fig. 8. Diagrama del proceso.

Desglose de cada término:

$$\frac{\partial L_i}{\partial a^l} = 2(a^l - y)$$

$$\frac{\partial a^l}{\partial z^l} = g(z^l) \cdot \big(1 - g(z^l)\big)$$

$$\frac{\partial z^l}{\partial b^l} = 1$$

$$\frac{\partial L_i}{\partial b^l} = \frac{\partial z^l}{\partial b^l} \cdot \frac{\partial a^l}{\partial z^l} \cdot \frac{\partial L_i}{\partial a^l}$$

$$\frac{\partial L_i}{\partial b^l} = 1 \cdot \big(g(z^l) \cdot \big(1 - g(z^l)\big)\big) \cdot 2(a^l - y)$$

2) ¿Qué sigue después?: Este procedimiento se realiza neurona por neurona, capa por capa, empezando desde la salida hacia las primeras capas. Lo interesante es que los mismos cálculos se **reutilizan** (se almacenan en **caché**), por lo que no hay que recomputar todo desde cero cada vez, optimizando el entrenamiento.

Nota práctica: Aunque hemos usado **sigmoide** en este ejemplo, el proceso es igual para cualquier función de activación (**ReLU**, **tanh**, etc.). Lo único que cambia es la fórmula de su derivada.

D. Derivación específica por neurona

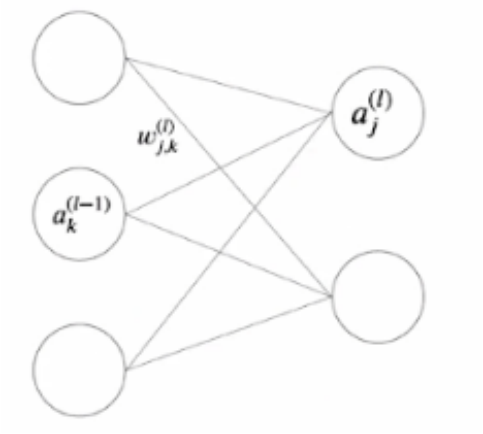

Fig. 9. Multiples neuronas.

Cuando calculamos las derivadas en redes **multicapa**, hay que tener en cuenta que:

- Solo nos interesa la **neurona específica** sobre la que derivamos.

Por ejemplo, si estamos derivando respecto a la neurona 2 de la capa L, todos los demás términos de la sumatoria son constantes. Solo se considera el término donde $j = 2$.

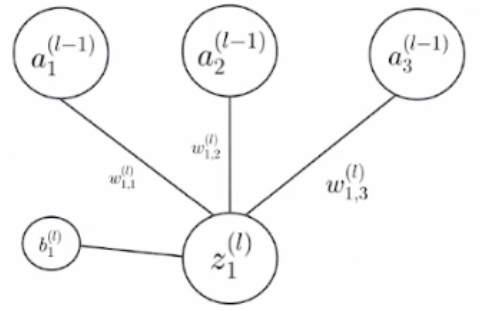

Fig. 10. Neurona Zoom in.

Preactivación:

$$z_j^{(l)} = b_j^{(l)} + \sum_{k=1}^{n^{(l-1)}} w_{j,k}^{(l)} a_k^{(l-1)}$$

Activación:

$$a_j^{(l)} = g\left(z_j^{(l)}\right)$$

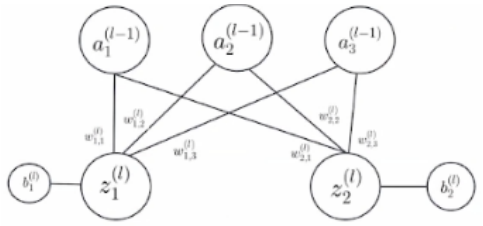

Fig. 11. Segundo ejemplo de neurona Zoom in.

Preactivación:

$$z_j^{(l)} = b_j^{(l)} + \sum_{k=1}^{n^{(l-1)}} w_{j,k}^{(l)} a_k^{(l-1)}$$

Activación:

$$a_j^{(l)} = g\left(z_j^{(l)}\right)$$

E. Desglose de derivadas parciales por parámetros

$$\frac{\partial L_i}{\partial w_{j,k}^{(l)}} = \frac{\partial z_j^{(l)}}{\partial w_{j,k}^{(l)}} \cdot \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \cdot \frac{\partial L_i}{\partial a_j^{(l)}}$$

Aplicando la **regla de la cadena**, las derivadas se calculan paso a paso:

- **Derivada de la función de pérdida respecto a la activación**:

$$L_i = \sum_{j=1}^{n_l} \left(a_j^{(l)} - y_j^{(l)}\right)^2$$

- **Calculo preactivación de una neurona**:

$$z_j^{(l)} = b_j^{(l)} + \sum_{k=1}^{n_{l-1}} w_{j,k}^{(l)} a_k^{(l-1)}$$

- **Activación de una neoruna**:

$$a_j^{(l)} = g\left(z_j^{(l)}\right)$$

- **Derivada de la preactivación respecto a la activación**:

$$L_i = \sum_{j=1}^{n_l} \left(a_j^{(l)} - y_j\right)^2$$

$$\frac{\partial L_i}{\partial a_j^{(l)}} = \left((a_1^{(l)} - y_1)^2 + (a_2^{(l)} - y_2)^2 + \cdots + (a_n^{(l)} - y_n)^2\right)'$$

$$\frac{\partial L_i}{\partial a_j^{(l)}} = 2(a_j^{(l)} - y_j)$$

- **Derivada de la activación con respecto a preactivación**:

$$\frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} = g\left(z_j^{(l)}\right)\left(1 - g\left(z_j^{(l)}\right)\right)$$

- **Derivada de la preactivación con respecto a W**:

$$z_j^{(l)} = b_j^{(l)} + \sum_{k=1}^{n_{l-1}} w_{j,k}^{(l)} a_k^{(l-1)}$$

El peso $w_{jk}^{(L)}$ conecta la neurona k de la capa anterior con la neurona j de la capa actual.

F. Extensión al cálculo en capas anteriores

Hasta ahora hemos trabajado en la última capa, pero ¿qué pasa cuando queremos ajustar los pesos de capas anteriores?

El error de una neurona en una capa previa depende de todas las neuronas a las que está conectada en la capa siguiente. Esto introduce una **sumatoria** en la regla de la cadena:

$$\frac{\partial\left(L_i\right)}{\partial a_k^{(l-1)}} = \sum_{j=1}^{n_l} \frac{\partial\left(z_j^{(l)}\right)}{\partial a_k^{(l-1)}} \cdot \frac{\partial\left(a_j^{(l)}\right)}{\partial z_j^{(l)}} \cdot \frac{\partial\left(L_i\right)}{\partial a_j^{(l)}}$$

La derivada de la pre-activación de la siguiente capa respecto a la activación de la capa anterior es:

$$L_i = \sum_{j=1}^{n_l} \left(a_j^{(l)} - y_j\right)^2$$

$$z_j^{(l)} = b_j^{(l)} + \sum_{k=1}^{n_{l-1}} w_{j,k}^{(l)} a_k^{(l-1)}$$

$$a_j^{(l)} = g\left(z_j^{(l)}\right)$$

Esto significa que cada neurona en una capa previa recibe retroalimentación desde **todas las neuronas conectadas en la siguiente capa**. Esto explica por qué el número de cálculos crece exponencialmente con el tamaño de la red.

G. Complejidad computacional

Cada vez que avanzamos hacia **capas más profundas** (anteriores) en el backpropagation, la cantidad de cálculos crece debido a las múltiples conexiones entre neuronas.

Por cada **peso** y **bias**, se debe calcular la derivada correspondiente, y muchas de estas derivadas requieren **sumatorias** que recorren todas las neuronas conectadas.

Nota: Aquí es donde se **reutilizan cálculos intermedios** (como en *caché*), para que el proceso no se vuelva ineficiente.

H. Cierre de la clase

Conclusión clave: El cálculo de los **gradientes** en una red neuronal no es complicado conceptualmente, pero crece en **complejidad computacional** cuando aumentan las capas y neuronas.

En la próxima clase, se discutirá cómo **optimizar estos cálculos**, posiblemente usando técnicas como **vectores Jacobianos** o **automatización de derivadas** (como hace TensorFlow o PyTorch).