

APUNTES DE CLASE

Inteligencia Artificial — Semana 9 — 02 de Octubre

Priscilla Jiménez Salgado

Escuela de Ingeniería en Computación, Tecnológico de Costa Rica

Cartago, Costa Rica — 2021022576@estudiantec.cr

Abstract—Este documento hace un repaso general y claro sobre las funciones de activación más utilizadas en las redes neuronales, además de explicar conceptos importantes sobre cómo están diseñadas y cómo han evolucionado las redes neuronales artificiales. Se presentan funciones como ReLU, Sigmoide y Softmax, entre otras, con su base matemática. También se repasa el concepto de perceptrón y las redes multicapa, y se comentan algunos retos clásicos en el área, como el problema del XOR y la llamada “maldición de la dimensionalidad”.

I. REVIEW DE LA LECTURA

En clase el profesor comentó de forma muy básica la lectura From Language to Action: A Review of Large Language Models as Autonomous Agents and Tool Users. Señaló que lo importante para el próximo quiz del martes es entender lo esencial: los modelos de lenguaje (LLMs) ya no solo generan texto, sino que también funcionan como agentes autónomos capaces de razonar, planificar, usar memoria e interactuar con herramientas externas. La lectura diferencia entre sistemas de un solo agente y sistemas multi-agente, donde varios modelos colaboran para resolver problemas más complejos. Además, se destacan sus aplicaciones en investigación, programación, salud, robótica y simulaciones, así como los principales retos, entre ellos la memoria limitada, la seguridad, la ética y la necesidad de mejores evaluaciones.

A. Noticias de la semana

En clase se habló del lanzamiento de Sora 2, el nuevo modelo de generación de video creado por OpenAI como respuesta al Nano Banana de Google.

A diferencia del primer Sora, que tenía resultados poco realistas, esta nueva versión produce videos más naturales y coherentes, además de incluir audio gracias a su capacidad multimodal. El profesor mostró un ejemplo hecho con la herramienta y explicó que incluso podría usarse para presentaciones académicas. También se mencionó la nueva aplicación “Sora by OpenAI”, una plataforma donde las personas pueden crear y compartir videos con inteligencia artificial a partir de simples descripciones o prompts.



Fig. 1: Sora by OpenAI

II. ASPECTOS ADMINISTRATIVOS

El profesor compartió las notas de los trabajos pendientes y brindó retroalimentación individual a cada grupo de trabajo. Sin embargo, aún queda por entregar la calificación del quiz 4 realizado y de la tarea presentada el pasado miércoles, que están pendientes de revisión. Además, se indicó que la próxima semana se asignará el proyecto del curso.

A. Repaso

– *El perceptrón*: Puede entenderse de forma similar a una regresión logística, aunque se diferencia en la función de pérdida que utiliza. Durante la historia de la inteligencia artificial se produjo el llamado “invierno de la IA”, en parte debido al problema del **XOR**, ya que este no podía ser representado

adecuadamente por un modelo de regresión logística ni por un perceptrón simple.

– Predicción de compuertas lógicas:

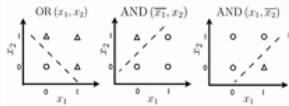


Fig. 2. Compuertas lógicas

En la figura se ilustran las compuertas lógicas **OR** y **AND** mediante gráficos bidimensionales.

- **OR(X_1, X_2):** Los triángulos indican la salida 1 y los círculos la salida 0. Esta compuerta devuelve 1 siempre que al menos una de las entradas sea igual a 1.
- **AND(\bar{X}_1, X_2):** Corresponde a una compuerta AND donde la primera entrada está negada. La salida es 1 (triángulo) únicamente cuando la primera entrada es 0 y la segunda es 1.
- **AND(X_1, \bar{X}_2):** Representa la compuerta AND con la segunda entrada negada. El resultado es 1 (triángulo) solo cuando la primera entrada es 1 y la segunda es 0.

En cada gráfico, la línea punteada marca el **límite de decisión** que distingue entre las dos clases de salida (0 y 1). Esta representación facilita la comprensión de cómo las compuertas lógicas realizan la clasificación de sus entradas en un espacio bidimensional.

– Problema del XOR:

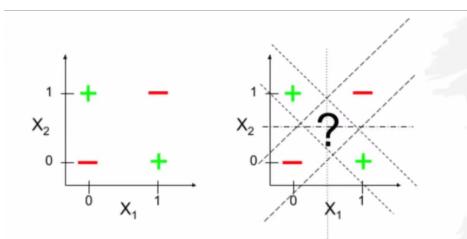


Fig. 3. Problema del XOR

El principal inconveniente es que el problema no es linealmente separable, por lo que el algoritmo del perceptrón simple no podía ofrecer una solución adecuada. Es en este punto donde surgen las redes neuronales o perceptrones multicapa, ya que estos sí

tienen la capacidad de abordar problemas no lineales. Gracias a ello, se amplía significativamente el rango de problemas que pueden resolverse con este método.

– Inspiración Biológica:

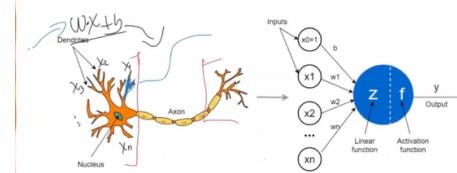


Fig. 4. Inspiración biológica

Las redes neuronales se inspiran en cómo funcionan las neuronas en nuestro cerebro. Cada neurona está conectada con otras a través de sus dendritas, y en el núcleo es donde se procesa la información.

Si lo comparamos con una regresión logística, las dendritas serían como las entradas de datos (inputs), y el núcleo representaría la función lineal que procesa esa información. Al final, la neurona decide si deja pasar o no esa señal.

– *Funciones de activación:* En la regresión logística esa transformación se conoce como función no lineal, específicamente la sigmoide. Según la señal recibida, la neurona se activa o no, permitiendo que la información continúe, la transforme o la bloquee.

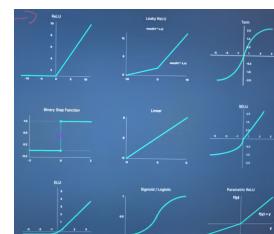


Fig. 5. Funciones de activación

- **Función ReLU:** La función $g(x) = \max(0, x)$ está limitada por debajo de cero y es estrictamente creciente. Es muy eficiente en modelos de *Deep Learning*, pero presenta el problema de las llamadas *neuronas muertas*, ya que no es derivable en todos los puntos y, en algunos casos, el gradiente puede llegar a ser cero, impidiendo la actualización de los pesos.

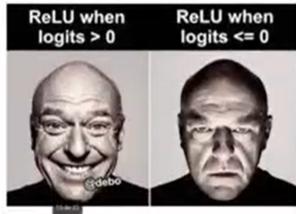


Fig. 6. Ejemplo ReLU

- **Leaky ReLU:** Esta función asigna una pequeña constante al valor mínimo permitido, lo que ayuda a evitar el problema de las neuronas muertas. Aunque representa una mejora respecto a la ReLU original, no se considera la solución definitiva.

$$g(x) = \begin{cases} 0.01x, & x < 0 \\ x, & x \geq 0 \end{cases}$$

$$\frac{\partial g(x)}{\partial x} = \begin{cases} 0.01, & x < 0 \\ 1, & x \geq 0 \end{cases}$$

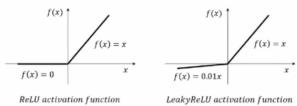


Fig. 7. Ejemplo Leaky ReLU

- **Parametric ReLU (PReLU):** Esta función permite aprender un parámetro que controla si la señal continúa en la parte negativa. Dicho parámetro se entrena junto con el resto de la red, lo que brinda mayor flexibilidad al modelo.

$$g(x) = \begin{cases} wx, & x < 0 \\ x, & x \geq 0 \end{cases}$$

$$\frac{\partial g(x)}{\partial x} = \begin{cases} w, & x < 0 \\ 1, & x \geq 0 \end{cases}$$

- **Función Tanh:** Tiene una forma parecida a la sigmoide, pero su salida está acotada en el rango $(-1, 1)$, lo que permite manejar valores positivos y negativos.
- **Binary Step Function:** Devuelve 1 si la entrada es mayor que cero y 0 si es menor o igual a cero.
- **Función lineal:** Básicamente deja pasar la salida sin aplicar ninguna transformación adicional.

- **Funciones SELU y ELU:** Son de la misma familia. Aunque requieren mayor costo computacional, ofrecen un rendimiento muy eficiente.
- **Función Sigmoid:** Convierte la entrada en un valor entre 0 y 1. Es muy usada cuando se necesita interpretar las salidas como probabilidades.

– Perceptrón multicapa (MLP):

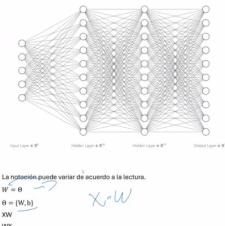


Fig. 8. Perceptrón

El **Perceptrón Multicapa (MLP)** es una evolución del perceptrón simple que permite resolver problemas más complejos, especialmente aquellos que no son linealmente separables.

El profesor lo explicó de manera sencilla con la imagen: en la *capa de entrada* (Input Layer) se encuentran los datos originales, representados como X_i , que no cambian porque son las entradas del sistema. Luego aparecen las *capas ocultas* (Hidden Layers), que son las responsables de realizar los cálculos, transformaciones y operaciones internas, dándole a la red la capacidad de aprender relaciones más complejas. Finalmente, está la *capa de salida* (Output Layer), que entrega el resultado final y cuyo tamaño depende del problema que se esté resolviendo.

La gran ventaja del MLP es que, gracias a sus múltiples capas y funciones de activación, introduce **no linealidad**, lo que le permite resolver problemas que el perceptrón simple no podía. Además, se entrena utilizando la *propagación del error* (backpropagation), que consiste en calcular cuánto se equivocó la red y ajustar los pesos mediante descenso de gradiente, mejorando así el rendimiento del modelo.

Ahora nos preguntamos, ¿cómo se calcula una pasada en la red?

El proceso comienza con la expresión $h(0) = \text{sigmoid}(XW_0 + b_0)$, donde $h(0)$ corresponde a la primera capa oculta. Lo que se hace es calcular primero la regresión lineal $XW_0 + b_0$, luego aplicar la función sigmoide al resultado, y con eso se obtiene el valor del primer *Hidden Layer*.

Después, para la siguiente capa oculta, el procedimiento es prácticamente el mismo: $h(1) = \text{sigmoid}(h(0)W_1 + b_1)$. En este caso, el valor de $h(0)$ pasa a ser la entrada de la siguiente capa.

Este mismo proceso se repite hasta llegar a la última capa, que se expresa como $h(n) = \text{sigmoid}(h(n-1)W_n + b_n)$.

En otras palabras, cada capa oculta toma como entrada el resultado de la capa anterior, y mediante una combinación lineal más la activación, se van construyendo paso a paso los valores hasta la salida final de la red.

– *Salida independiente y distribución*: Cada salida puede asociarse a una variable distinta. Según el caso, la distribución puede ser de tipo categórica (como en el uso de *softmax*) o continua (como en una regresión).

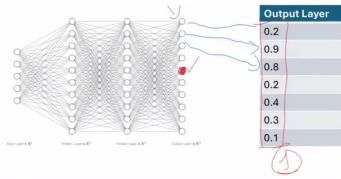


Fig. 9. Salida independiente

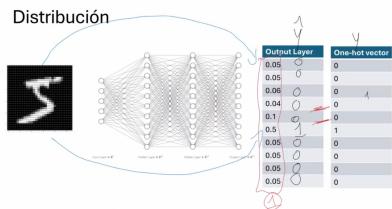


Fig. 10. Distribución

– *Capa de salida*: Es la parte final de la red y se calcula con la fórmula $h(n) = g(h(n-1)W^n + b^n)$. Básicamente, lo que hace es tomar la salida de la última capa oculta, multiplicarla por los pesos, sumarle un sesgo y luego pasarl por una función de activación. Esta función $g(x)$ no siempre es la sigmoide, puede ser otra dependiendo del tipo de

tarea: *softmax* si se trata de una clasificación múltiple, o lineal si es un problema de regresión. Lo importante es que sea una función no lineal, ya que eso es lo que le da a la red la capacidad de resolver

Capa de salida

- $h(n) = g(h(n-1)W^n + b^n)$
- Donde $g(x)$ no es necesariamente la función sigmoid.
- Función no-lineal.

Fig. 11. Capa de salida

– *Función costo*: Es una **función matemática** que calcula el nivel de error del modelo, y cuyo objetivo principal es minimizar dicho error durante el proceso de entrenamiento.

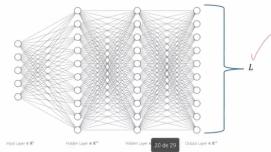


Fig. 12. Función de costo

– *Maldición de dimensionalidad*: Pasa cuando trabajamos con datos que tienen muchísimas variables o dimensiones. Al ir aumentando esas dimensiones, los datos empiezan a dispersarse y quedan muy separados entre sí, lo que hace más difícil encontrar patrones claros. En otras palabras, el modelo tiene que calcular en un espacio cada vez más grande y con menos densidad de información, lo que complica el aprendizaje.

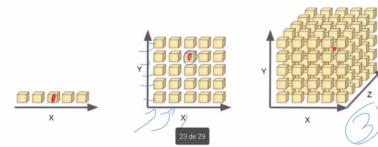


Fig. 13. Maldición de la dimensionalidad

– *Comportamiento jerárquico*: Se utiliza este enfoque porque imita la forma en que los humanos aprenden: comienzan con conceptos simples y luego los combinan para formar ideas más complejas. Esto permite generar mejoras exponenciales en las funciones y aprovechar mejor el aprendizaje.

- Permite construir funciones polinómicas.

- Utiliza la composición de funciones, reutilizando funciones simples para crear otras de mayor nivel.
 - Ofrece una representación compacta, donde con pocos pesos se pueden modelar funciones complejas.
 - **Ejemplo:** una red neuronal puede aproximar otra función.

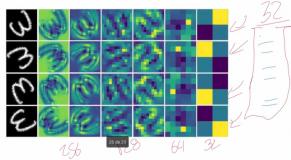


Fig. 14. Comportamiento jerárquico

– *Mapas de características en CNN*: En una red neuronal convolucional (CNN), las capas no trabajan solo con los píxeles, sino que van aprendiendo representaciones cada vez más complejas de la imagen. Al inicio, en las primeras capas, se detectan cosas muy básicas como bordes o líneas. Luego, en las capas intermedias, ya aparecen formas un poco más claras como partes de ojos o bocas. Finalmente, en las últimas capas, la red es capaz de reconocer objetos completos, por ejemplo un rostro.

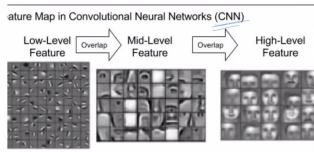


Fig. 15. Extracción progresiva de características en una CNN

- *Representaciones Vectoriales:* En procesamiento de lenguaje natural, las palabras se representan como vectores de alta dimensión, esto permite que palabras con funciones similares se agrupen en el espacio vectorial.

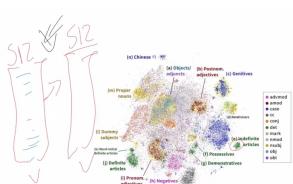


Fig. 16. Visualización

III. CONTINUACIÓN DE FUNCIONES DE ACTIVACIÓN

Las funciones de activación son un elemento fundamental en las redes neuronales, ya que permiten introducir la no linealidad necesaria para representar relaciones complejas en los datos. A continuación, se presentan las funciones más importantes junto con sus principales características matemáticas.

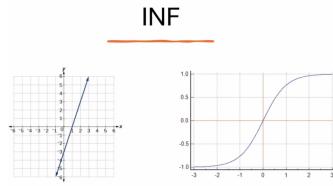


Fig. 17. Ejemplos de funciones de activación: a la izquierda la función lineal y a la derecha la función tangente hiperbólica (\tanh), usada en redes neuronales para introducir no linealidad.

A. Función Lineal

La función lineal se define como $f(x) = x$. La derivada es constante, por lo que el modelo no puede usar el descenso del gradiente ni aprender de los datos.

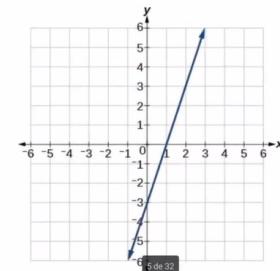


Fig. 18. Función lineal

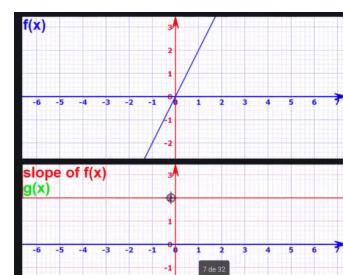


Fig. 19. Ejemplo

B. Sigmoide

Tiene una activación que varía entre 0 y 1, siempre positiva, acotada y estrictamente creciente. Sin embargo, presenta el problema de que su derivada se approxima a cero en los extremos de la función, lo que

provoca gradientes muy pequeños. Esto hace que el entrenamiento se vuelva lento o se detenga, fenómeno conocido como *vanishing gradient*.

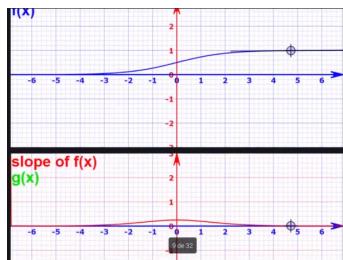


Fig. 20. Ejemplo

C. Tangente Hiperbólica

La función **TanH** tiene un rango de valores entre -1 y 1. Su comportamiento es similar al de la función sigmoide, con la diferencia de que está centrada en el origen, lo que permite que los valores negativos también sean considerados. Sin embargo, al igual que la sigmoide, presenta el problema del *gradiente desvanecido* en los extremos, lo que puede dificultar el entrenamiento de redes profundas.

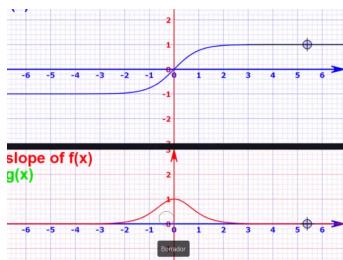


Fig. 21. Ejemplo

D. Función Softmax

La función **Softmax** convierte la capa de salida (*output layer*) en una distribución de probabilidad, ya que normaliza los valores mediante una sumatoria. Su definición es la siguiente:

$$\sigma(x)_j = \frac{e^{x_j}}{\sum_{k=1}^K e^{x_k}}$$

Es comúnmente utilizada en problemas de clasificación, donde el vector de entrada se conoce como *logits*. Además, se emplea junto con la función de pérdida **Cross-Entropy Loss**.

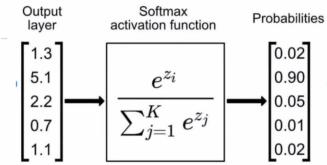


Fig. 22. Ejemplo softmax

- **¿Por qué usar e^x ?** Porque es una función estrictamente creciente y evita valores negativos en la salida.
- **Cross-Entropy Loss:** También llamada *Log-Loss* o *Logistic Loss*, se utiliza como función de pérdida en Softmax. Representa probabilidades en un espacio logarítmico dentro del rango [0, 1] y es numéricamente estable.

La pérdida se define como:

$$L = \log(P(Y = y_i | X = x_i))$$

y en el caso de clasificación multiclas:

$$L = -\log\left(\frac{e^{s_k}}{\sum_{j=1}^C e^{s_j}}\right)$$

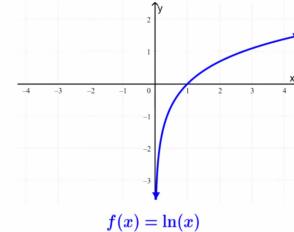


Fig. 23. Ejemplo

E. ¿Cuál función de activación utilizar?

La elección de la función de activación depende del tipo de problema que se esté resolviendo. Las funciones **Sigmoid** y **Tanh** suelen presentar el inconveniente del *vanishing gradient*, lo que dificulta el entrenamiento en redes profundas. Por ello, se recomienda iniciar con la función **ReLU**, ya que es rápida de calcular y ampliamente utilizada en *Deep Learning*. En caso de que no funcione adecuadamente, se pueden emplear variantes como **Leaky ReLU** o **Parametric ReLU**, que buscan superar estas limitaciones.

IV. BACKPROPAGATION

Permite calcular cuánto contribuye cada peso al error final de la red, actualizando los parámetros en dirección opuesta a la propagación hacia adelante. Este proceso es esencial para que la red aprenda y mejore su desempeño durante el entrenamiento.

A. Procesos del Entrenamiento

- **Forward Propagation:** Consiste en calcular la salida de la red enviando los datos desde la capa de entrada hacia las capas siguientes, hasta obtener el resultado final.
- **Backpropagation:** Implica propagar el error desde la capa de salida hacia las capas anteriores, calculando las derivadas parciales con respecto a los pesos y sesgos para ajustar los parámetros del modelo.

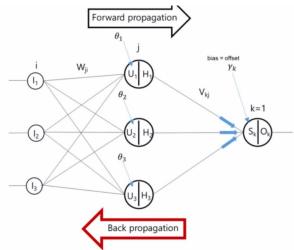


Fig. 24. Forward y Back Propagation

B. Optimización del grafo

En este ejemplo se considera una red neuronal en la que cada capa contiene únicamente una neurona, suponiendo que la función de activación utilizada es la *Sigmoide*, como se muestra en la Figura ??.

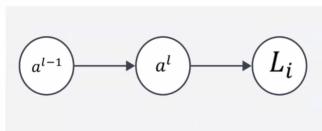


Fig. 25. Grafo de la red neuronal

- Denominamos a las capas antes de L_i , a^l hasta a^{l-n} .
- Definimos el MSE como:

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

- Dividimos la neurona en 2 capas:

- 1) **Entrada:** $z^l = w^l a^{l-1} + b^l$ donde a^{l-1} corresponde a los inputs x .

- 2) **Salida:** $a^l = g(z^l)$ donde g es nuestra función de activación.

Vamos a actualizar los parámetros de z^l , que son w^l y b^l . Para esto emplearemos la **regla de la cadena**, usando la salida de la activación de la capa anterior. Profundizando a nivel de neurona, se muestra la siguiente figura.

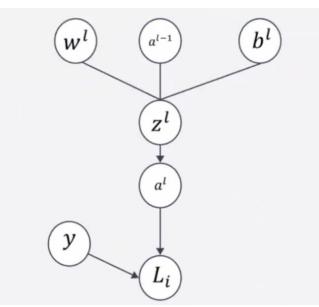


Fig. 26. Grafo de la capa al y Li a detalle

C. Vector gradiente

El vector gradiente se define como el conjunto de derivadas parciales de los parámetros (pesos y sesgos) de la red neuronal. Al calcularlo, es común encontrar operaciones repetidas, lo que se aprovecha en el algoritmo de backpropagation para optimizar los cálculos.

$$\nabla L = \begin{bmatrix} \frac{\partial(L)}{\partial w^{(1)}} \\ \frac{\partial(L)}{\partial b^{(1)}} \\ \vdots \\ \frac{\partial(L)}{\partial w^{(n)}} \\ \frac{\partial(L)}{\partial b^{(n)}} \end{bmatrix}$$

Fig. 27. Vector Gradiente

D. Múltiples neuronas

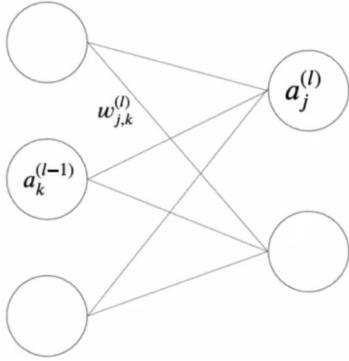


Fig. 28. Grafo con mayor dimensionalidad

- Superíndice:** Señala la capa a la que pertenece una variable. Ejemplo: $a^{(l)}$ corresponde a la capa l .
- Subíndice:** Identifica el número de neurona dentro de una capa específica. Ejemplo: $a_j^{(l)}$ se refiere a la j -ésima neurona en la capa l .
- Pesos:** Se representan con dos subíndices: el primero indica la neurona destino y el segundo la neurona de origen. Ejemplo: $w_{j,k}^{(l)}$ representa el peso que conecta la neurona $a_k^{(l-1)}$ con la neurona $a_j^{(l)}$.

A continuación, en la siguiente figura se ilustra cómo una neurona de la capa l recibe entradas desde varias neuronas de la capa anterior ($l - 1$). Este proceso se puede dividir en dos pasos:

- Preactivación:**

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

donde $b_j^{(l)}$ es el sesgo de la neurona y $w_{j,k}^{(l)}$ los pesos de conexión.

- Activación:**

$$a_j^{(l)} = g(z_j^{(l)})$$

donde g representa la función de activación aplicada.

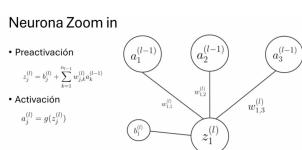


Fig. 29. Ejemplo

E. Cálculo de función de perdida

Para esta sección, el *loss* global se obtiene sumando las diferencias entre la salida de cada neurona en la capa de activación j y su valor esperado y_j , recorriendo todas las neuronas de la capa l .

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

En la siguiente figura se muestra un ejemplo donde se evalúa la salida de una capa de activación utilizando esta función de pérdida.

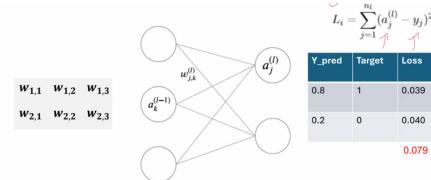


Fig. 30. Ejemplo

Cambios a la Regla de la Cadena

Como las funciones L_i , $z_j^{(l)}$ y $a_j^{(l)}$ han sido modificadas, es necesario plantear nuevas derivadas que permitan actualizar los parámetros de cada neurona j .

En la ecuación se observa que, para ajustar un peso específico $w_{j,k}^{(l)}$, debemos calcular sus derivadas parciales. Sin embargo, gracias al concepto de *caché*, la única derivada que cambia al actualizar un peso diferente es $\frac{\delta z_j^{(l)}}{\delta w_{j,k}^{(l)}}$, mientras que el resto permanece constante para toda la capa.

Las derivadas se expresan de la siguiente forma:

$$\frac{\delta L_i}{\delta a_j^l} = ((a_1^l - y_1)^2 + (a_2^l - y_2)^2 + \dots + (a_n^l - y_n)^2)$$

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

$$\frac{\delta a_j^{(l)}}{\delta z_j^{(l)}} = g(z_j^{(l)}) (1 - g(z_j^{(l)}))$$

$$\frac{\delta z_j^{(l)}}{\delta w_{j,k}^{(l)}} = a_k^{(l-1)}$$

Con esto se logra actualizar los pesos de la capa l , aunque las derivadas no cambian, sí deben manejarse más índices a medida que la red crece en complejidad.

Capa $l - 1$

Cuando el cálculo debe extenderse hacia una capa anterior, como la capa $l - 1$, el procedimiento se vuelve más complejo. Esto ocurre porque, según el tamaño de la siguiente capa, el algoritmo requiere combinar más conexiones y parámetros, lo que incrementa la dificultad del cálculo.

$$\frac{\delta L_i}{\delta a_k^{(l-1)}} = \sum_{j=1}^{n^l} \frac{\delta z_j^{(l)}}{\delta a_k^{(l-1)}} \frac{\delta a_j^{(l)}}{\delta z_j^{(l)}} \frac{\delta L_i}{\delta a_j^{(l)}}$$