Inteligencia Artificial

Machine Learning - (Aprendizaje Automático)

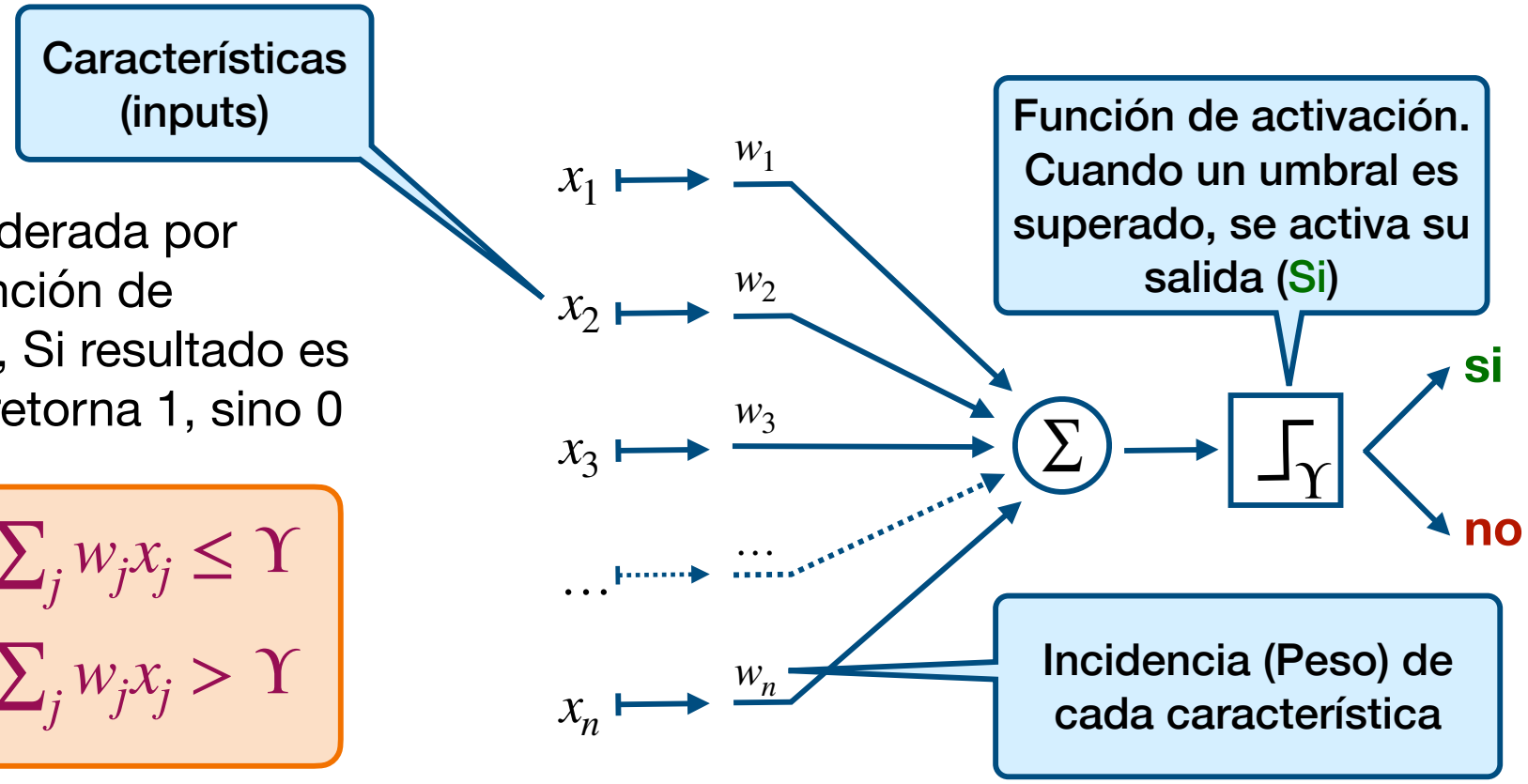
Redes Neuronales

Redes Neuronales - “Perceptron”

Retomemos la idea del **perceptrón**, que es el modelo más simple de una red neuronal artificial, propuesto por **Frank Rosenblatt en 1958**.

Calcula una suma ponderada por pesos, y aplica una función de activación al resultado, Si resultado es superior a un umbral, retorna 1, sino 0

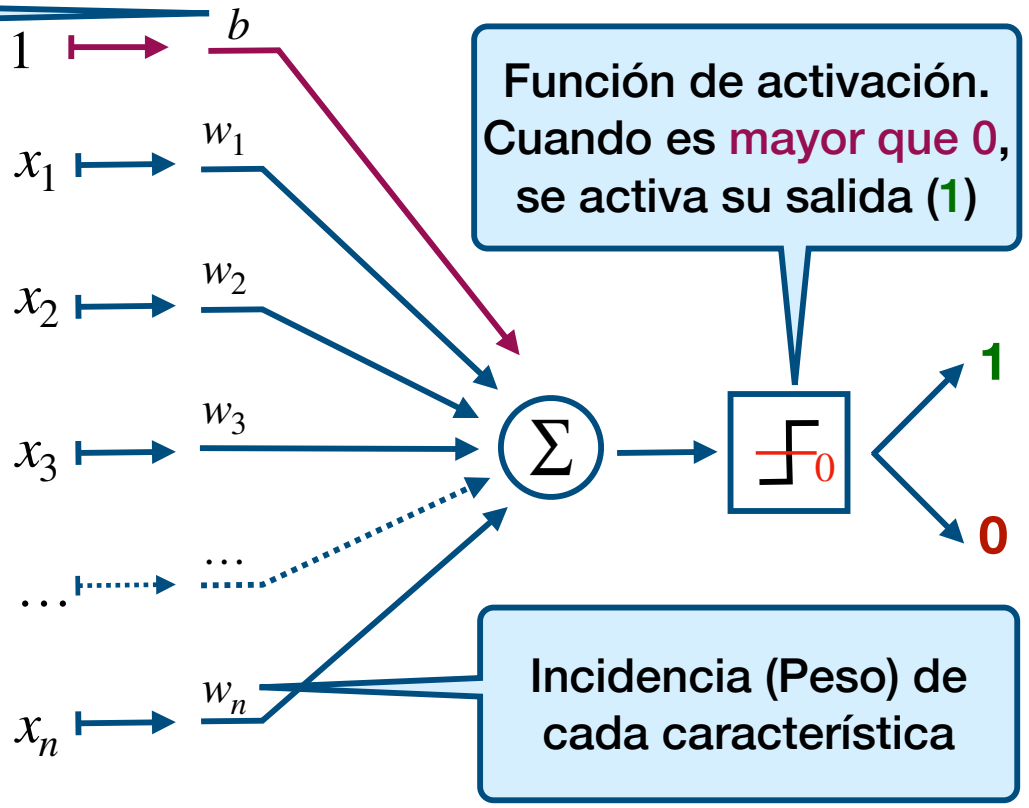
$$output = \begin{cases} \text{no} & \sum_j w_j x_j \leq \Upsilon \\ \text{si} & \sum_j w_j x_j > \Upsilon \end{cases}$$



Redes Neuronales - “Perceptron”

Retomemos la idea del **perceptrón**, que es el modelo más simple de una red neuronal artificial, propuesto por **Frank Rosenblatt en 1958**.

Agregamos la noción de bías



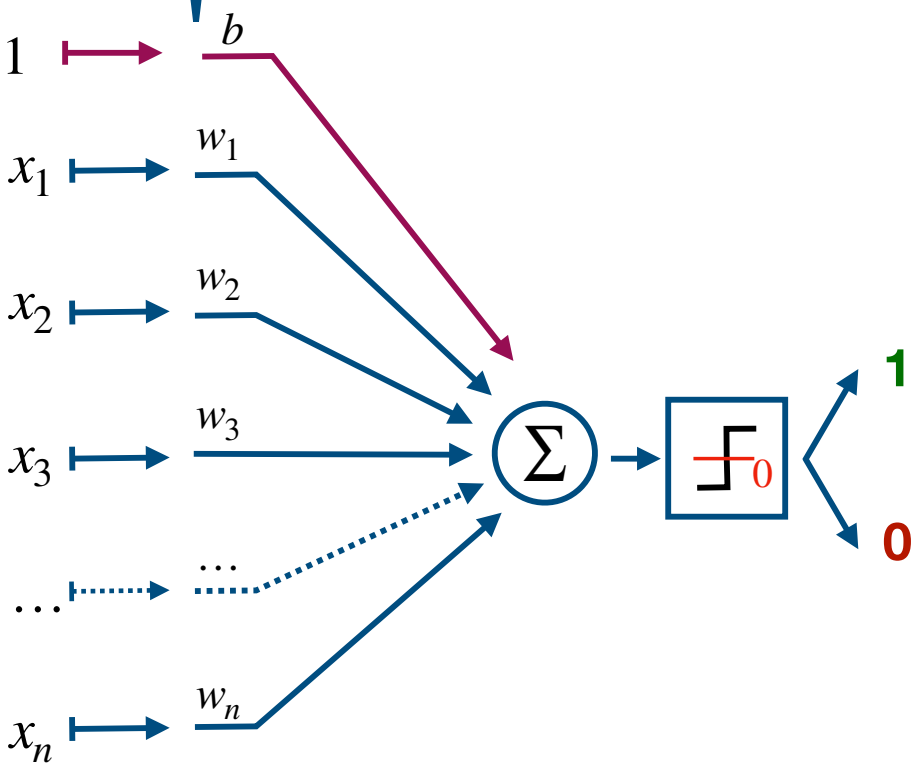
Remplazamos la notación $\Sigma_j w_j x_j$ por $w \cdot x$.

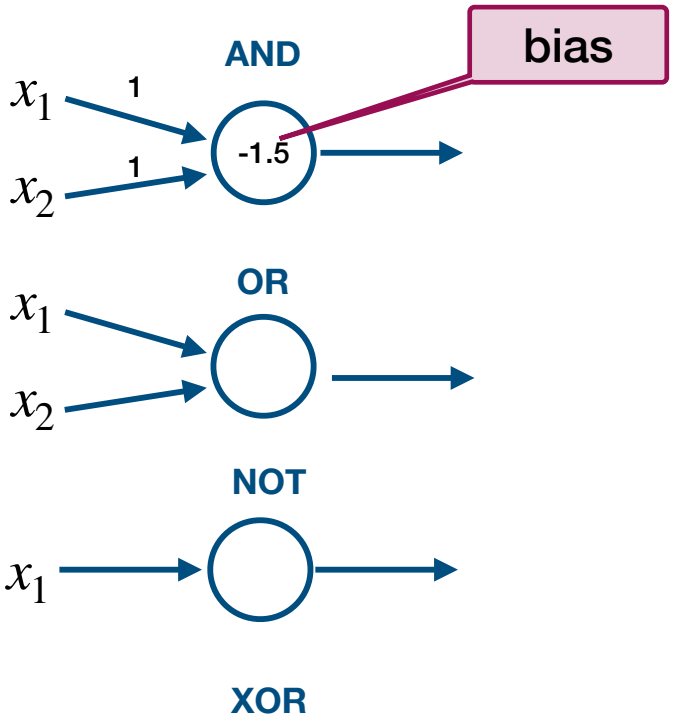
$$output = \begin{cases} 0 & w \cdot x + b \leq 0 \\ 1 & w \cdot x + b > 0 \end{cases}$$

Redes Neuronales - “Perceptron”

¿ Qué pesos (w) y bias (b) elegimos para que el resultado coincida con la función que queremos aprender de manera generalizada ?

$$output = \begin{cases} 0 & w \cdot x + b \leq 0 \\ 1 & w \cdot x + b > 0 \end{cases}$$

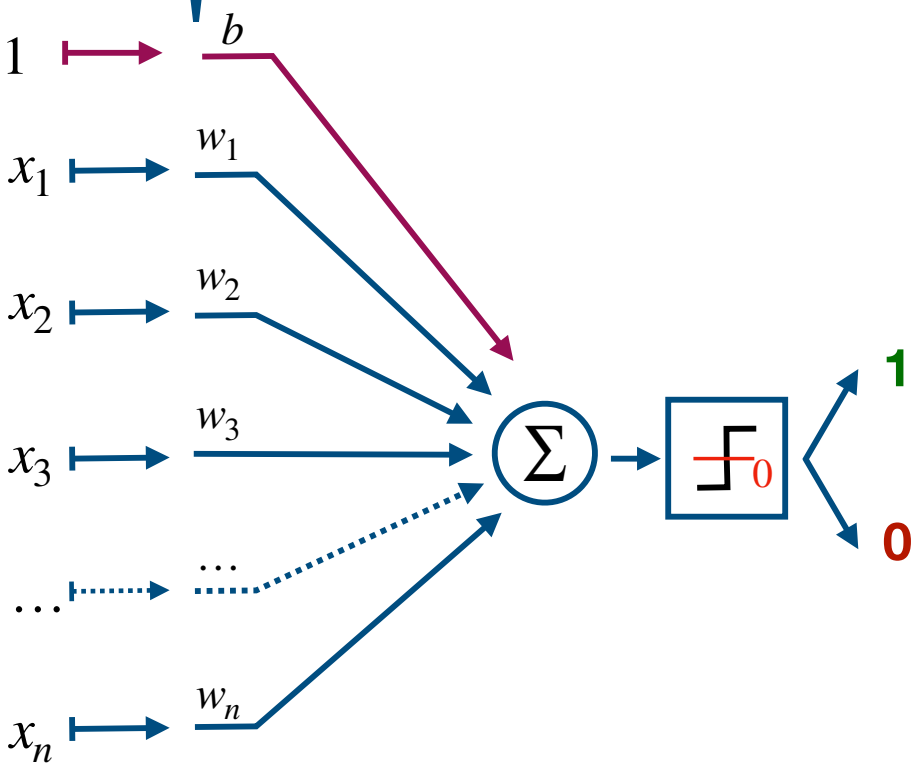


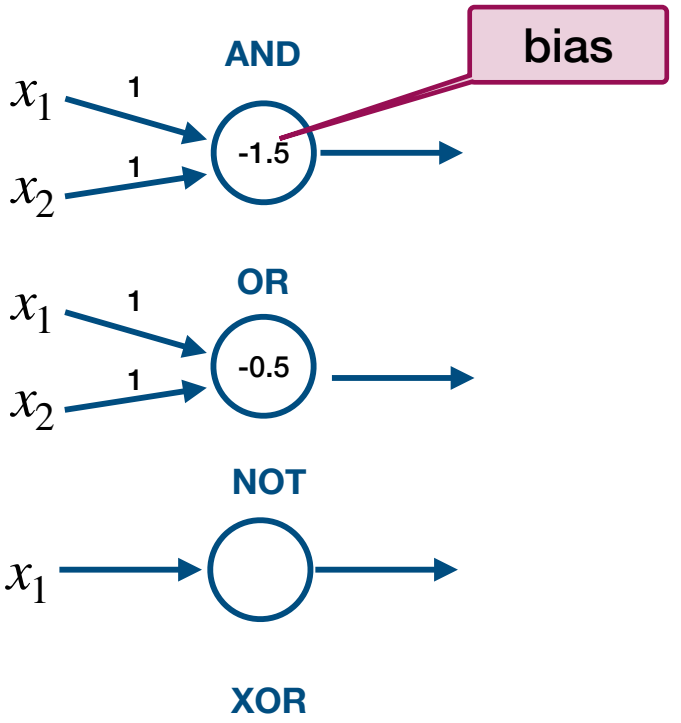


Redes Neuronales - “Perceptron”

¿ Qué pesos (w) y bias (b) elegimos para que el resultado coincida con la función que queremos aprender de manera generalizada ?

$$output = \begin{cases} 0 & w \cdot x + b \leq 0 \\ 1 & w \cdot x + b > 0 \end{cases}$$

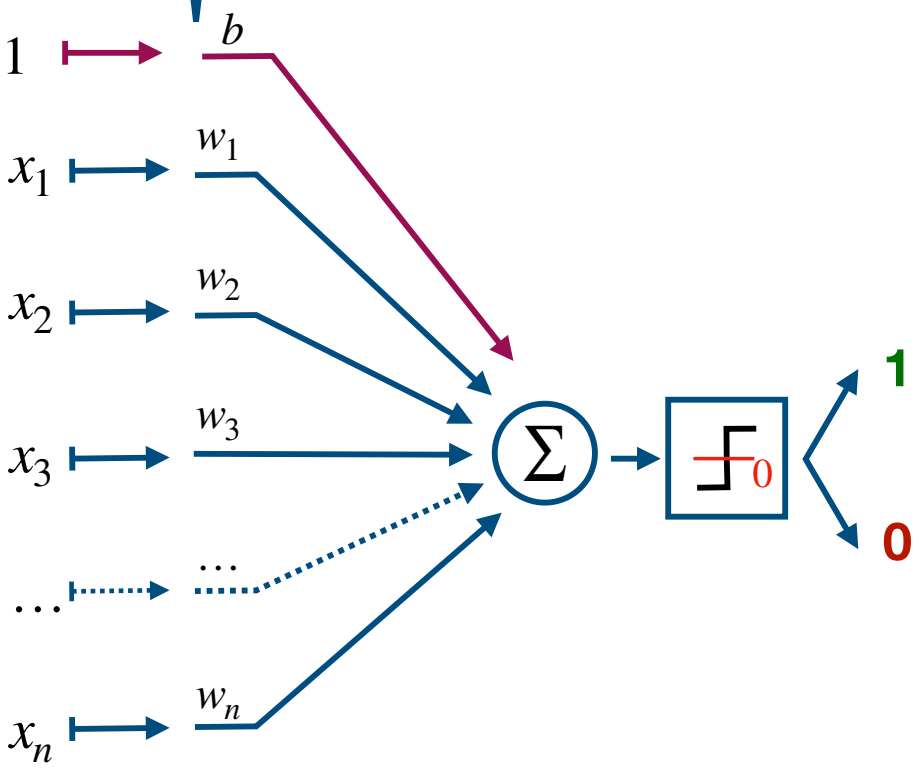


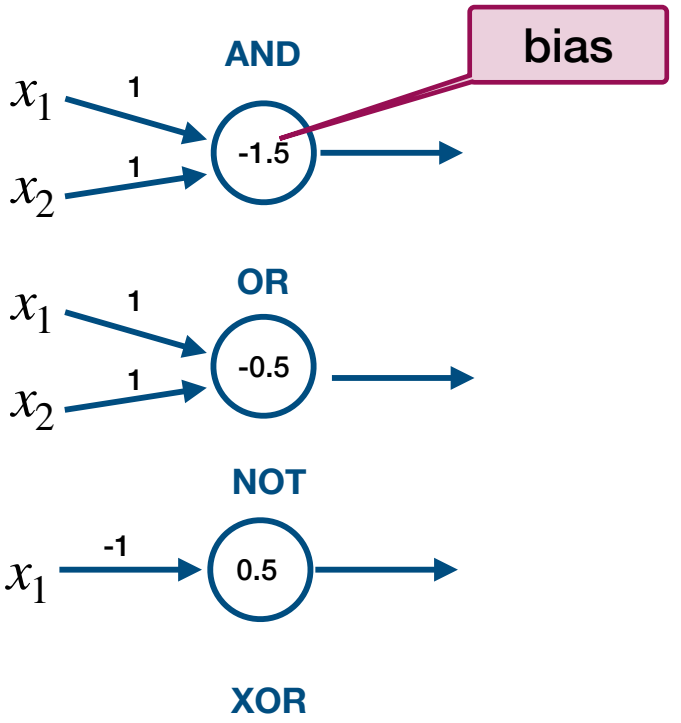


Redes Neuronales - “Perceptron”

¿ Qué pesos (w) y bias (b) elegimos para que el resultado coincida con la función que queremos aprender de manera generalizada ?

$$output = \begin{cases} 0 & w \cdot x + b \leq 0 \\ 1 & w \cdot x + b > 0 \end{cases}$$

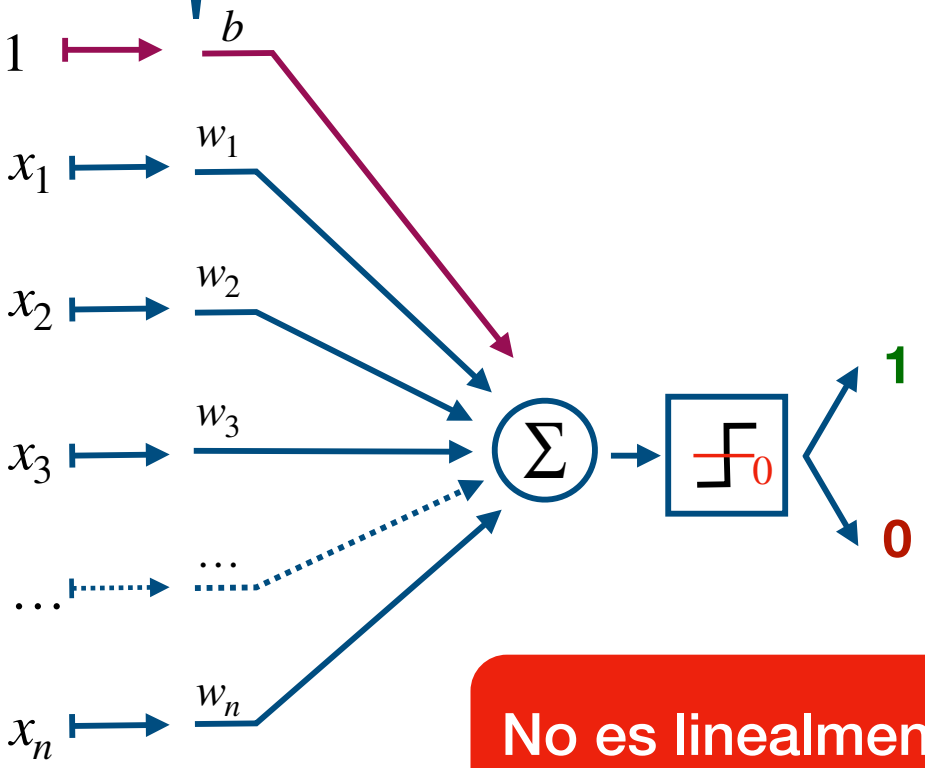


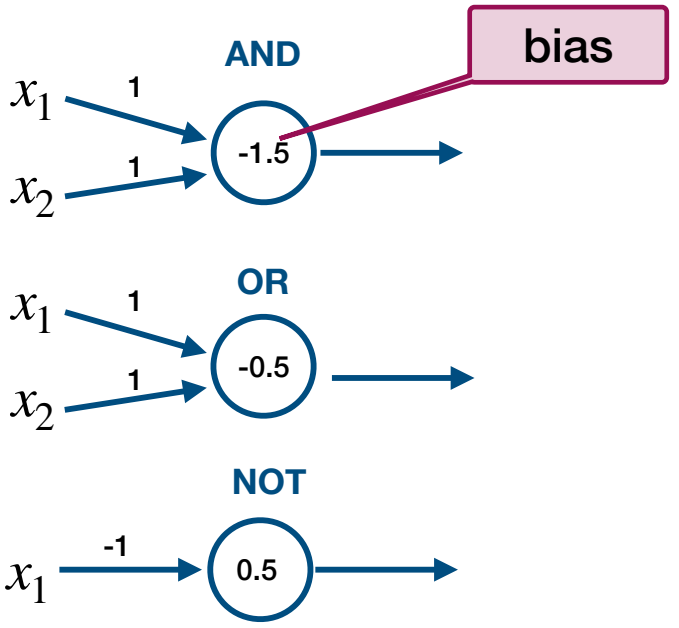


Redes Neuronales - “Perceptron”

¿ Qué pesos (w) y bias (b) elegimos para que el resultado coincida con la función que queremos aprender de manera generalizada ?

$$output = \begin{cases} 0 & w \cdot x + b \leq 0 \\ 1 & w \cdot x + b > 0 \end{cases}$$



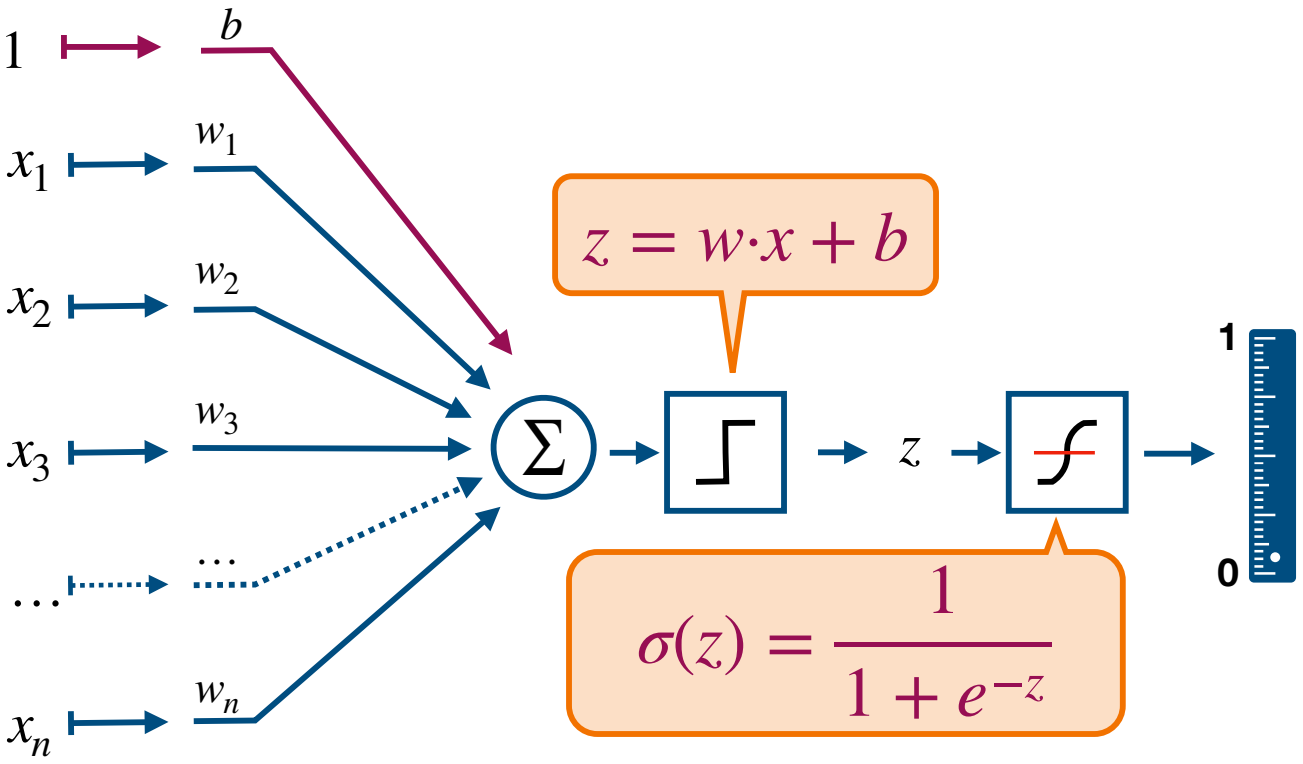


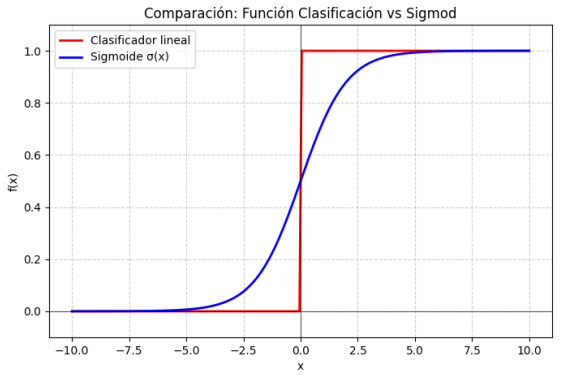
No es linealmente separable

XOR

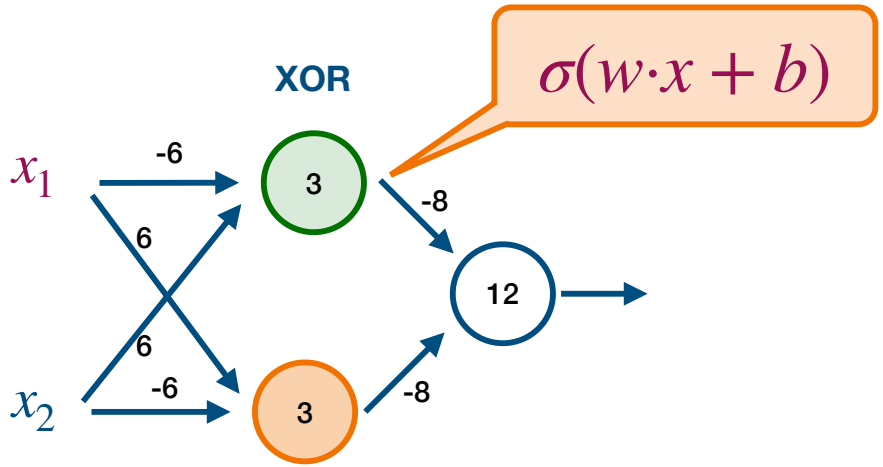
Redes Neuronales - Respuesta probabilística

Con el mismo objetivo que para clasificación, en lugar de utilizar una decisión discreta (>0,<=0), utilizamos una función que nos proyecte la salida en una probabilidad en 0 y 1.





Redes Neuronales - XOR



Combinando varios perceptrones (neuronas) podemos crear una **Red Neuronal** que resuelve el problema de computar la función lógica XOR. Podemos entender su comportamiento como la composición del cómputo de las neuronas.

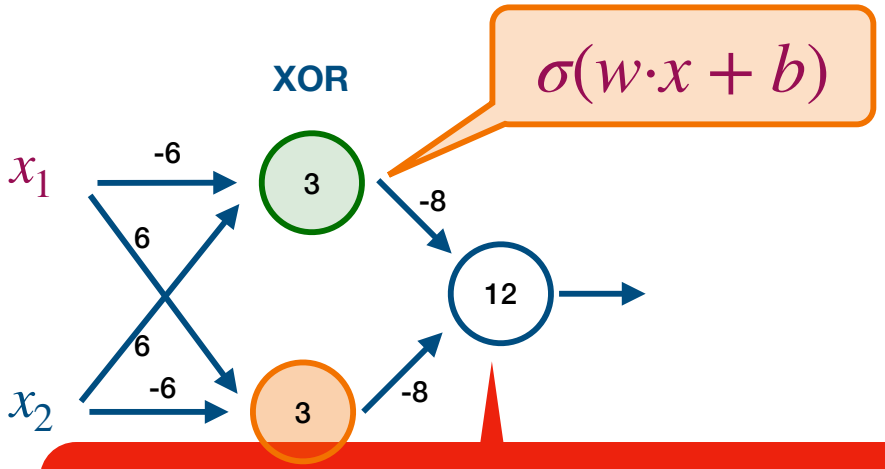
[0,0] = σ (σ(0 * -6 + 0 * 6 + 3) * -8 + σ(0 * 6 + 0 * -6 + 3) * -8 + 12) = 0.0376 = False

[1,0] = σ (σ(1 * -6 + 0 * 6 + 3) * -8 + σ(1 * 6 + 0 * -6 + 3) * -8 + 12) = 0.9740 = True

[0,1] = σ (σ(0 * -6 + 1 * 6 + 3) * -8 + σ(0 * 6 + 1 * -6 + 3) * -8 + 12) = 0.9740 = True

[1,1] = σ (σ(1 * -6 + 1 * 6 + 3) * -8 + σ(1 * 6 + 1 * -6 + 3) * -8 + 12) = 0.0376 = False

Redes Neuronales - XOR



Combinando varios perceptrones (neuronas) podemos crear una **Red Neuronal** que resuelve el problema de computar la función lógica XOR. Podemos entender su comportamiento como la composición del cómputo de las neuronas.

¿ cómo averiguamos (aprendemos) los valores de los pesos y bias de cada neurona ?

[0,0] = σ (σ(0 * -6 + 0 * 6 + 3) * -8 + σ(0 * 6 + 0 * -6 + 3) * -8 + 12) = 0.0376 = False

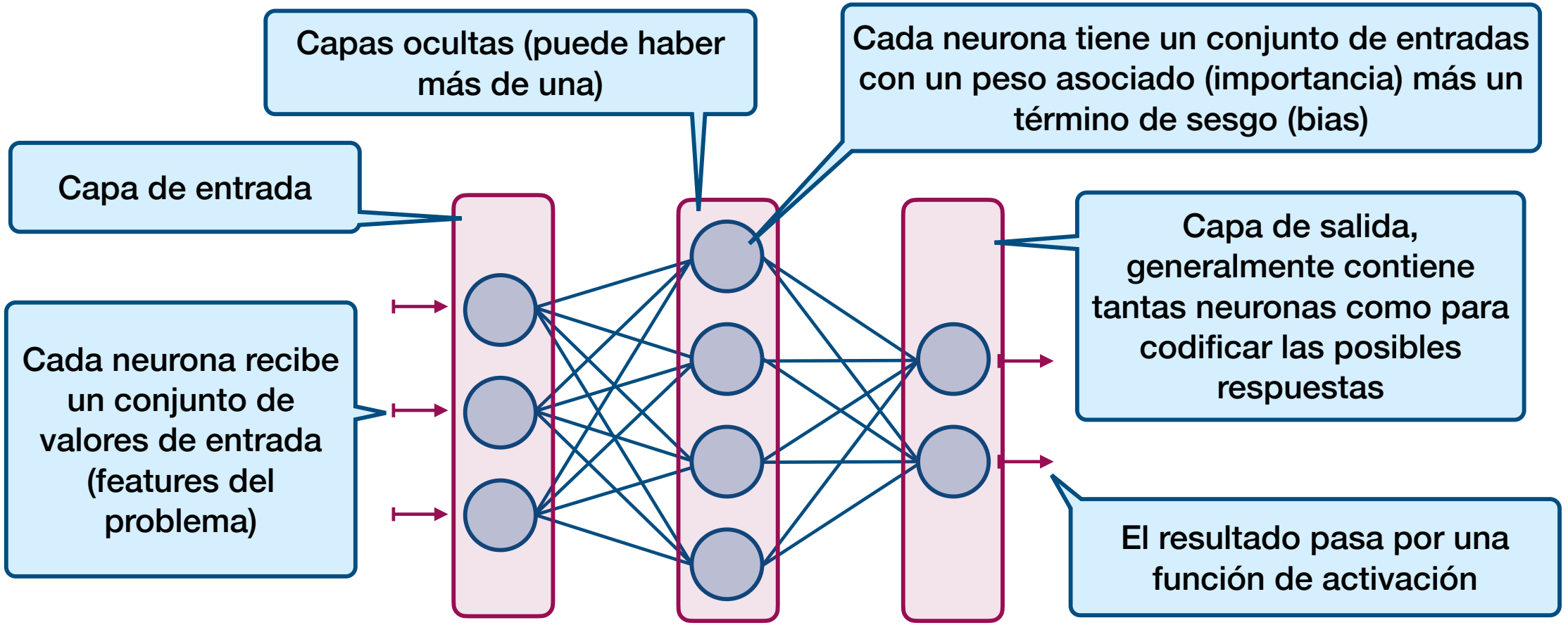
[1,0] = σ (σ(1 * -6 + 0 * 6 + 3) * -8 + σ(1 * 6 + 0 * -6 + 3) * -8 + 12) = 0.9740 = True

[0,1] = σ (σ(0 * -6 + 1 * 6 + 3) * -8 + σ(0 * 6 + 1 * -6 + 3) * -8 + 12) = 0.9740 = True

[1,1] = σ (σ(1 * -6 + 1 * 6 + 3) * -8 + σ(1 * 6 + 1 * -6 + 3) * -8 + 12) = 0.0376 = False

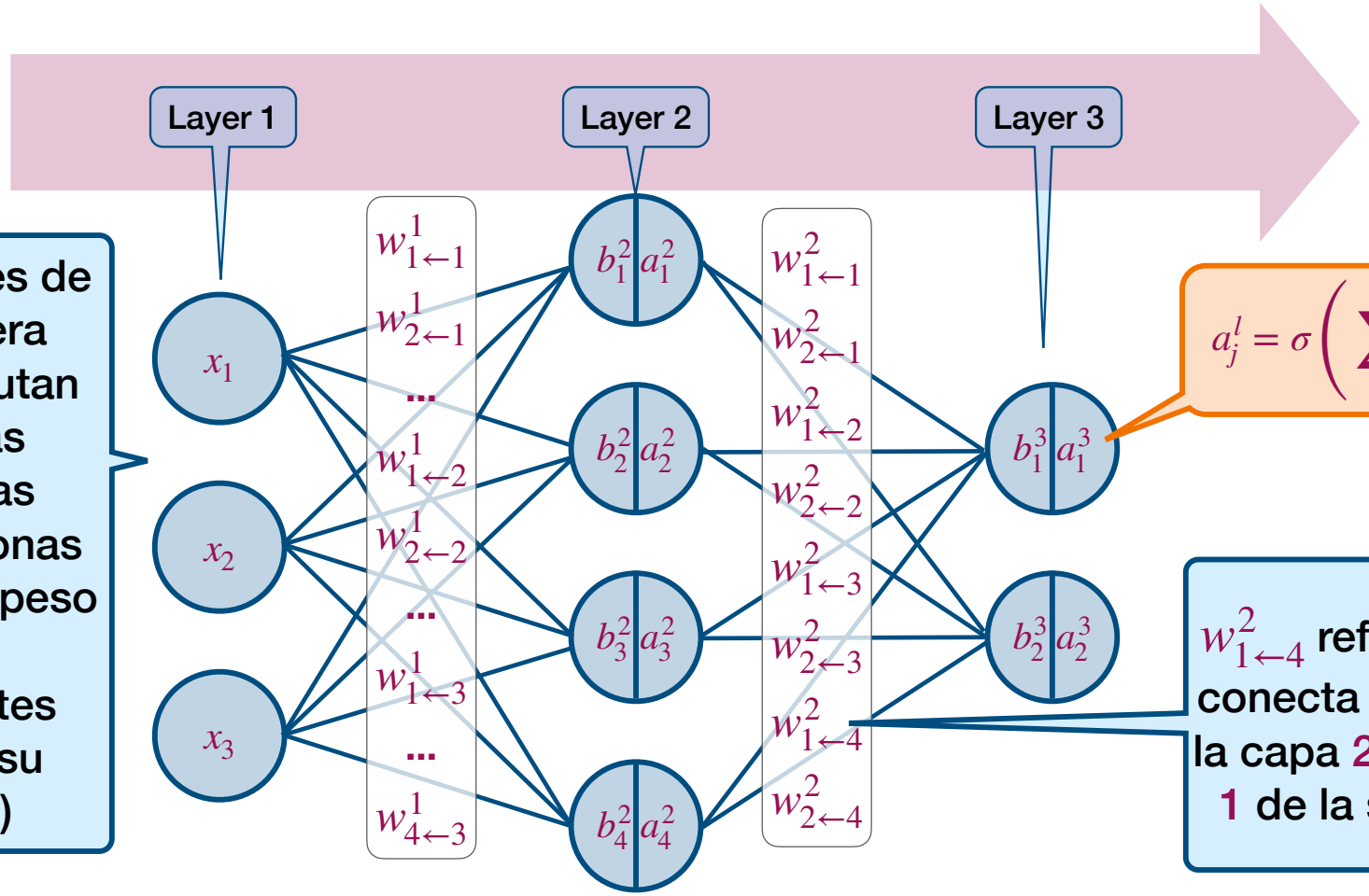
Redes Neuronales - Estructura

Una red neuronal artificial (ANN) es un modelo computacional compuesto por unidades llamadas **neuronas artificiales**, organizadas en **capas** (entrada, ocultas y salida), que están **conectadas entre sí mediante pesos**.



Redes Neuronales - Funcionamiento

El funcionamiento de las ANN se puede intuir como la combinación de funcionamiento de varios perceptrones avanzando sobre las diferentes capas (layers).

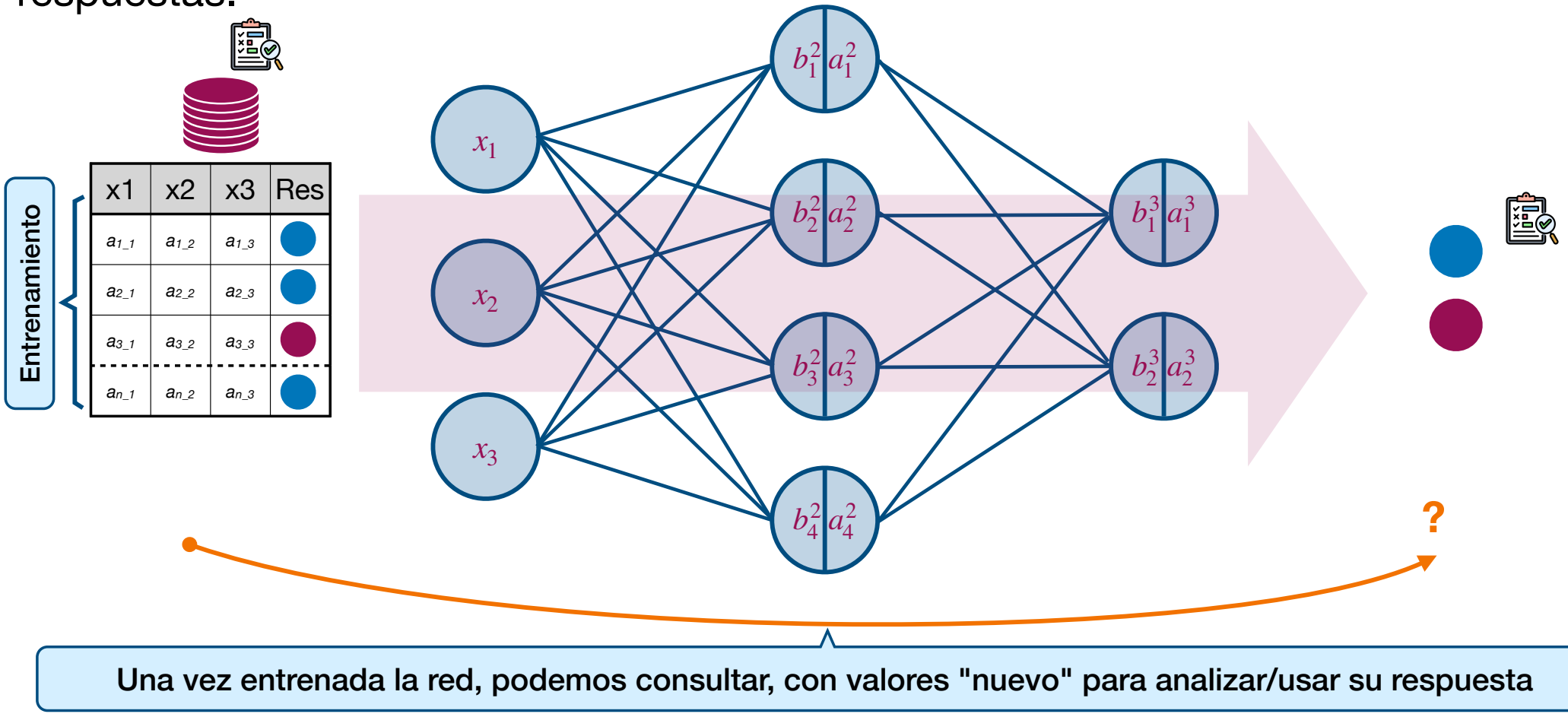


Dados los valores de entrada (primera capa), se computan (alimentan) las entradas de las siguientes neuronas considerando el peso de las correspondientes conexiones y su sesgos (bias)

$w^2_{1\leftarrow 4}$ refiere a peso que conecta la neurona 4 de la capa 2 con la neurona 1 de la siguiente capa

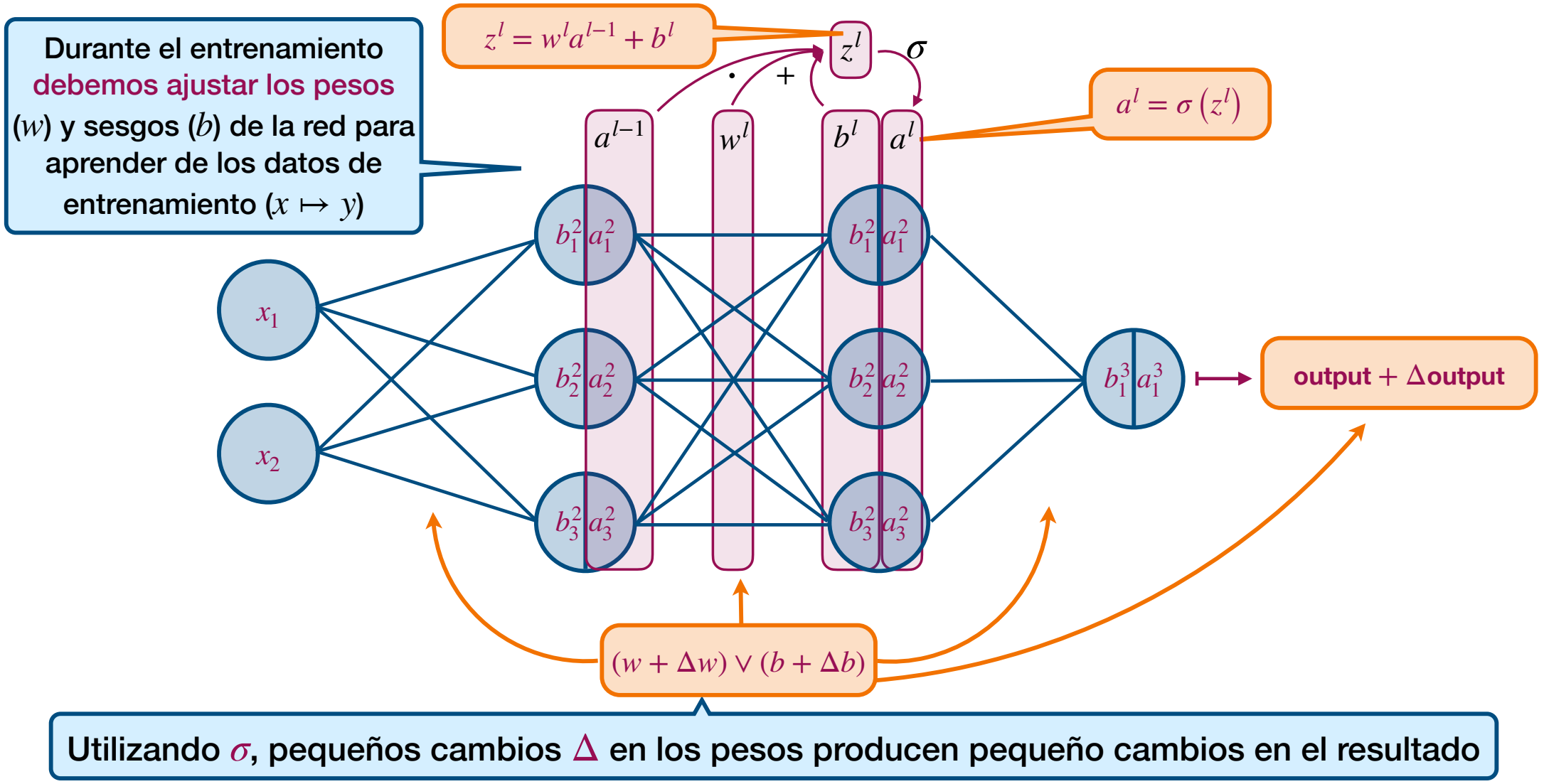
Redes Neuronales - Entrenamiento

El principal de desafío de las redes neuronales es su entrenamiento, es decir, cómo "aprender" el comportamiento de un conjunto de datos con sus correspondientes respuestas.



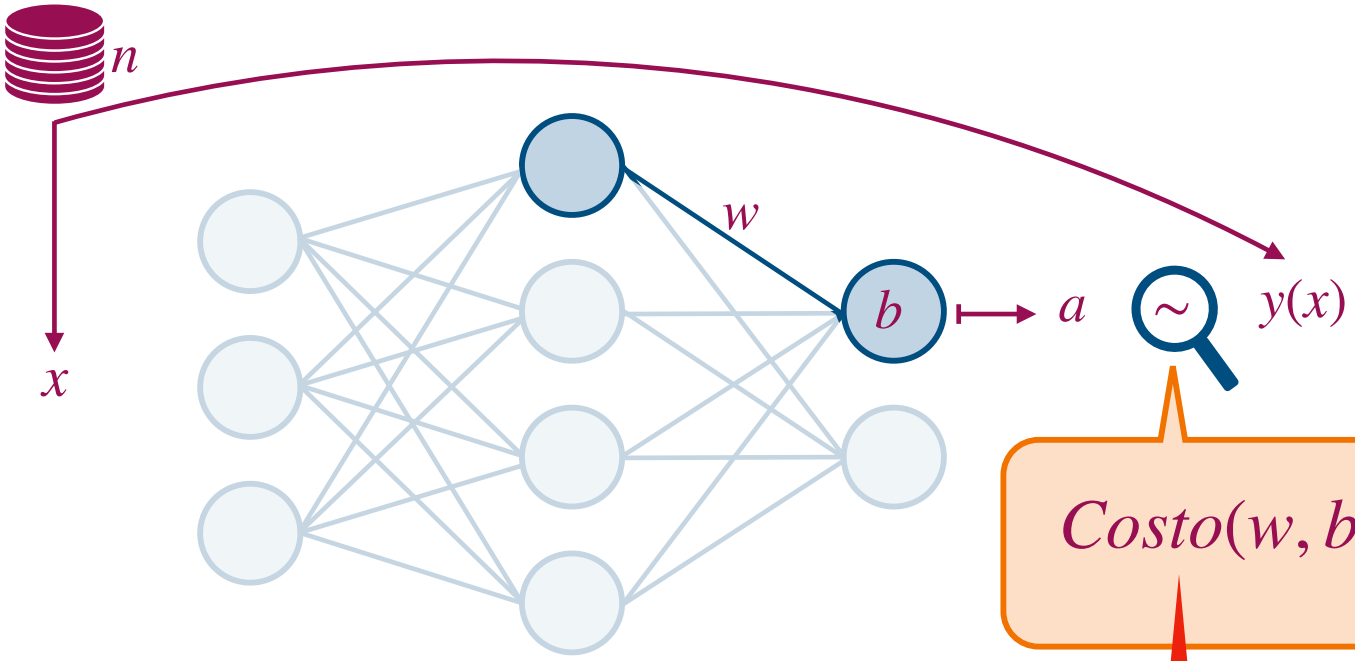
Una vez entrenada la red, podemos consultar, con valores "nuevo" para analizar/usar su respuesta

Redes Neuronales - Cómo entrenamos una ANN



Redes Neuronales - Cómo entrenamos una ANN

De manera similar que con otras técnicas de aprendizaje, podemos utilizar el MSE como una métrica para medir el **costo** entre el resultado actual aprendido de la ANN **vs el valor que debería retornar** en base a dataset de entrenamiento.

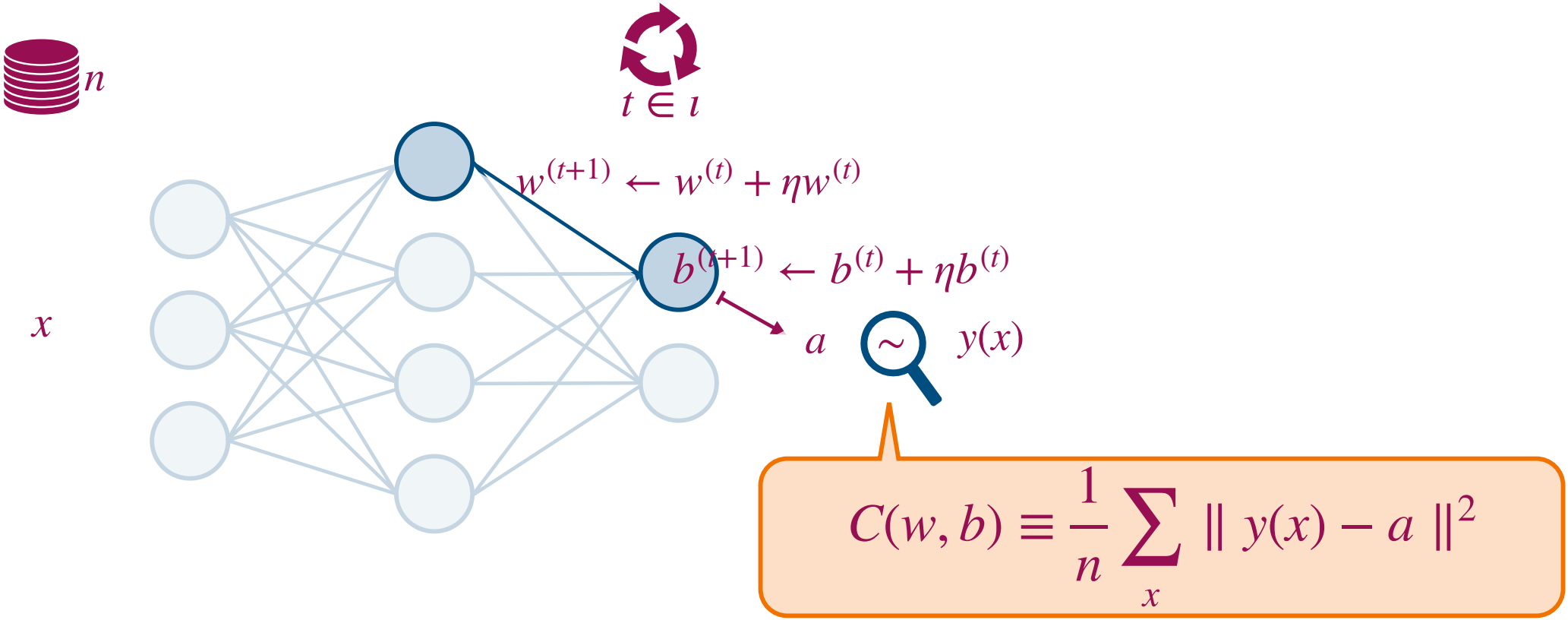


$$Costo(w,b) \equiv \frac{1}{n}\sum_{x} \| y(x) - a \|^2$$

La red "aprende" en la medida que podamos minimizar la funcion de Costo C

Redes Neuronales - Cómo entrenamos una ANN

Para minimizar la función de costo podemos ir alterando (actualizando) iterativamente durante ι (épocas) los pesos w y sesgos b de la red de manera contemplando un grado de aprendizaje η (learning rate).

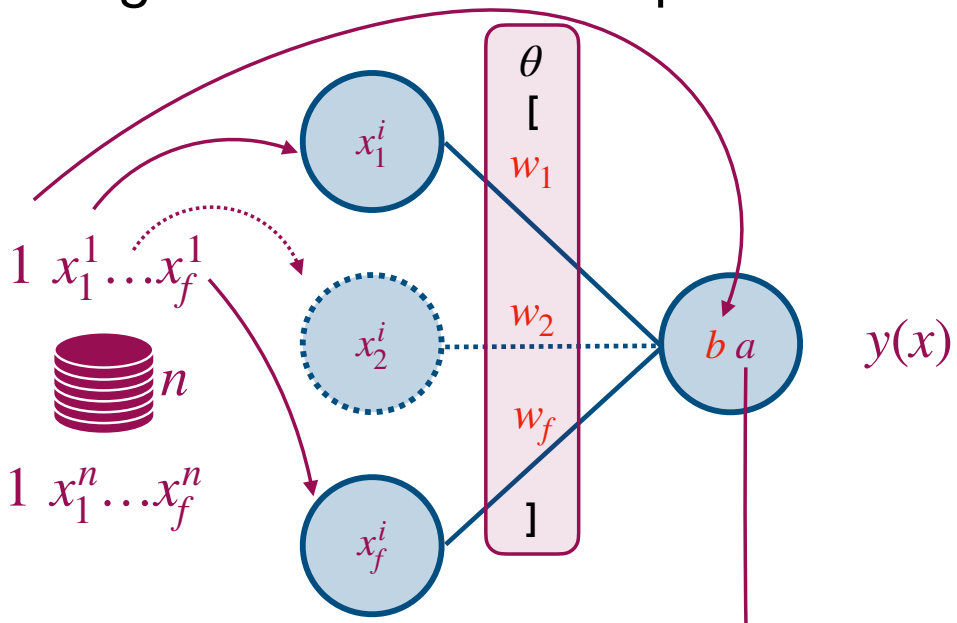


Redes Neuronales - Cómo entrenamos una ANN

Una técnica que ayuda a converger más rápidamente es la técnica del descenso del gradiente, podemos utilizarla. Teniendo en cuenta la fórmula del costo que adoptamos para el entrenamiento, los respectivos gradientes a contemplar durante el entrenamiento son:

$$w^{(t+1)} \leftarrow w^{(t)} + \eta \frac{\delta C}{\delta w^{(t)}} \qquad -\frac{2}{n}\sum_{i=1}^{n} x_i(y_i - (wx_i + b))$$

$$b^{(t+1)} \leftarrow b^{(t)} + \eta \frac{\delta C}{\delta b^{(t)}} \qquad -\frac{2}{n}\sum_{i=1}^{n} y_i - (wx_i + b)$$

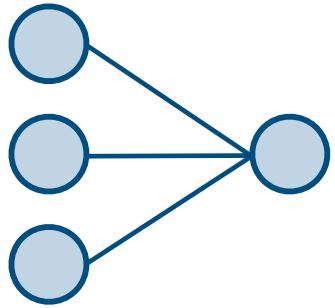


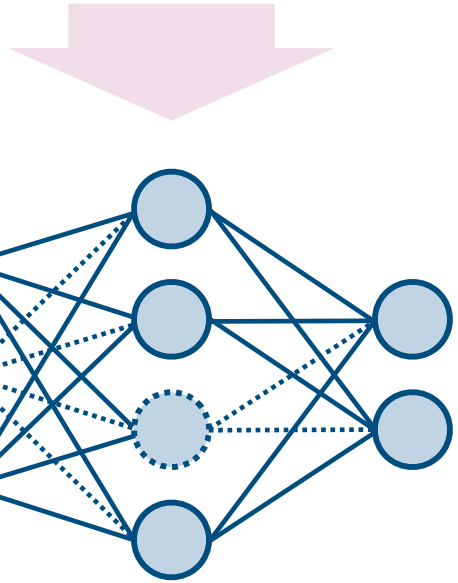

```
for iteration in range(n_iter):
    gradients = (1/m) * X_b.T @ (X_b @ theta - y)  # derivadas parciales
    theta = theta - eta * gradients
```

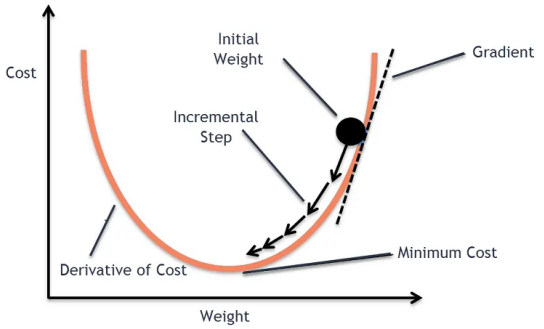
$$\theta = \theta - \eta\, 2\mathbf{X}^{\mathbf{T}}(\mathbf{X}\theta - y)$$

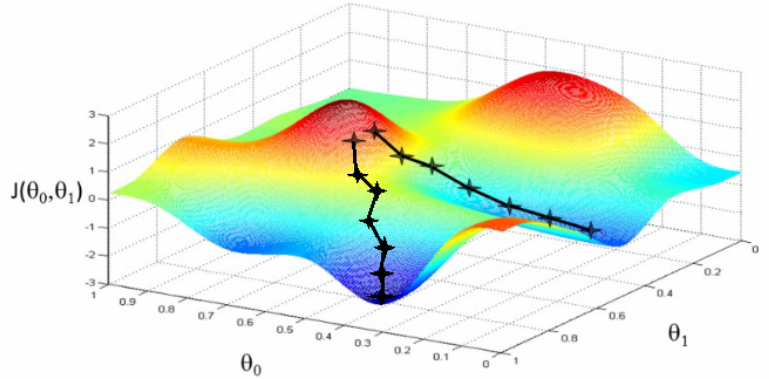
Redes Neuronales - Cómo entrenamos una ANN

¿ Cómo podemos trasladar la idea a una ANN con L capas y múltiples salidas ?



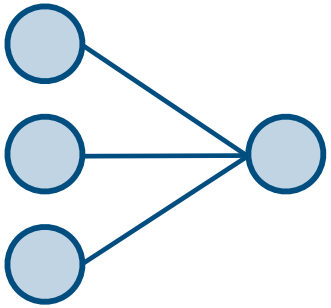


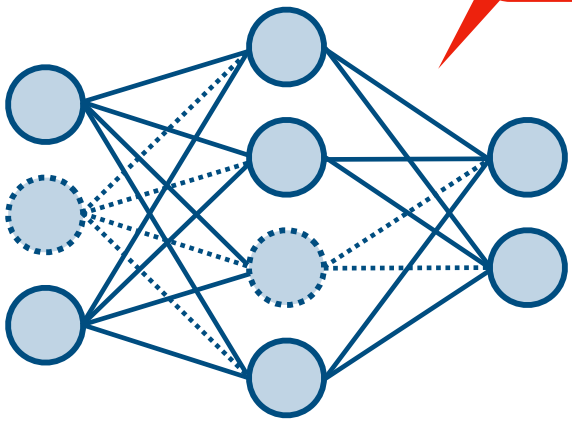




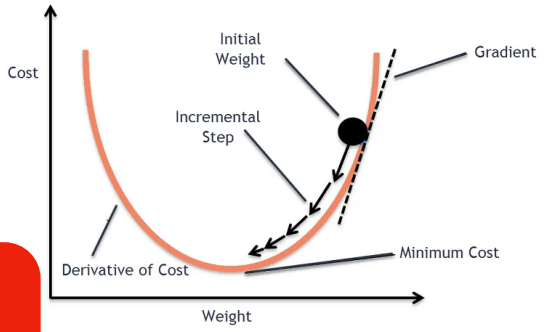
Redes Neuronales - Cómo entrenamos una ANN

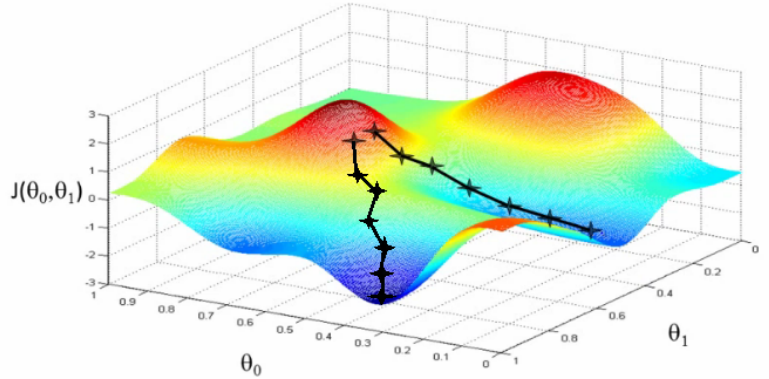
¿ Cómo podemos trasladar la idea a una ANN con L capas y múltiples salidas ?





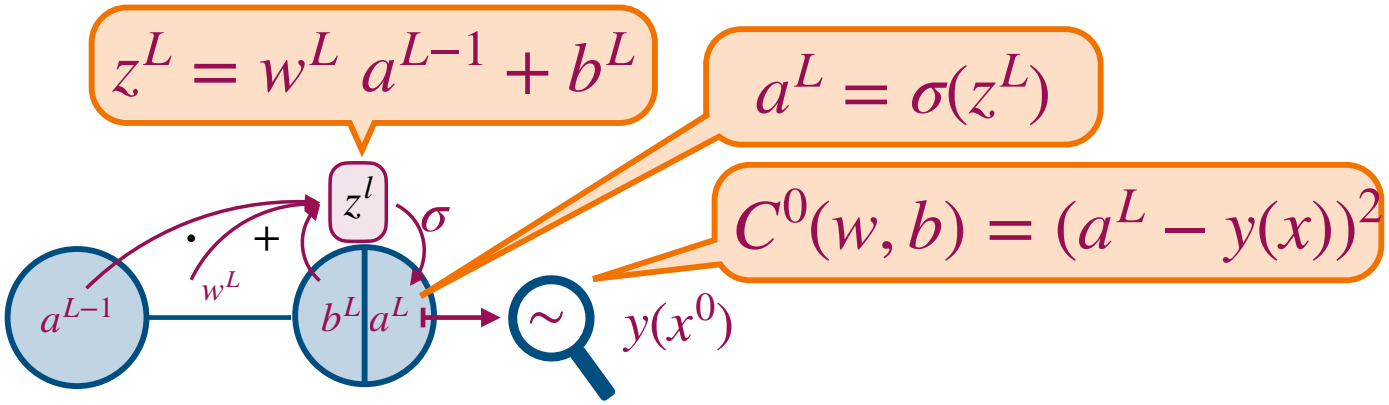
Backpropagation (retropropagación)





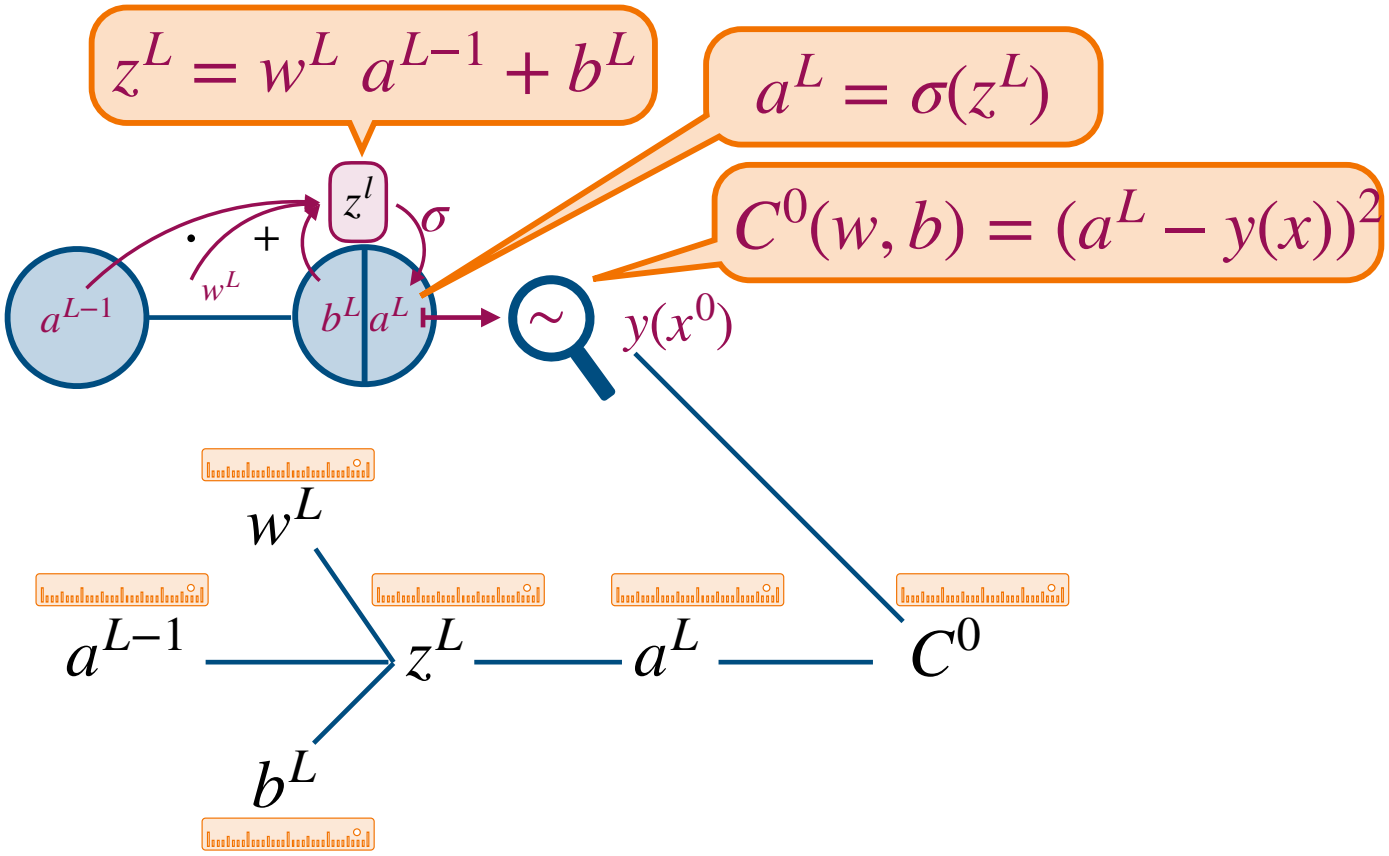
Redes Neuronales - Backpropagation

Para entender cómo funciona comencemos con entendiendo la relación de los componentes de una red respecto del costo para una red simple y con un solo caso de entrenamiento x^0.



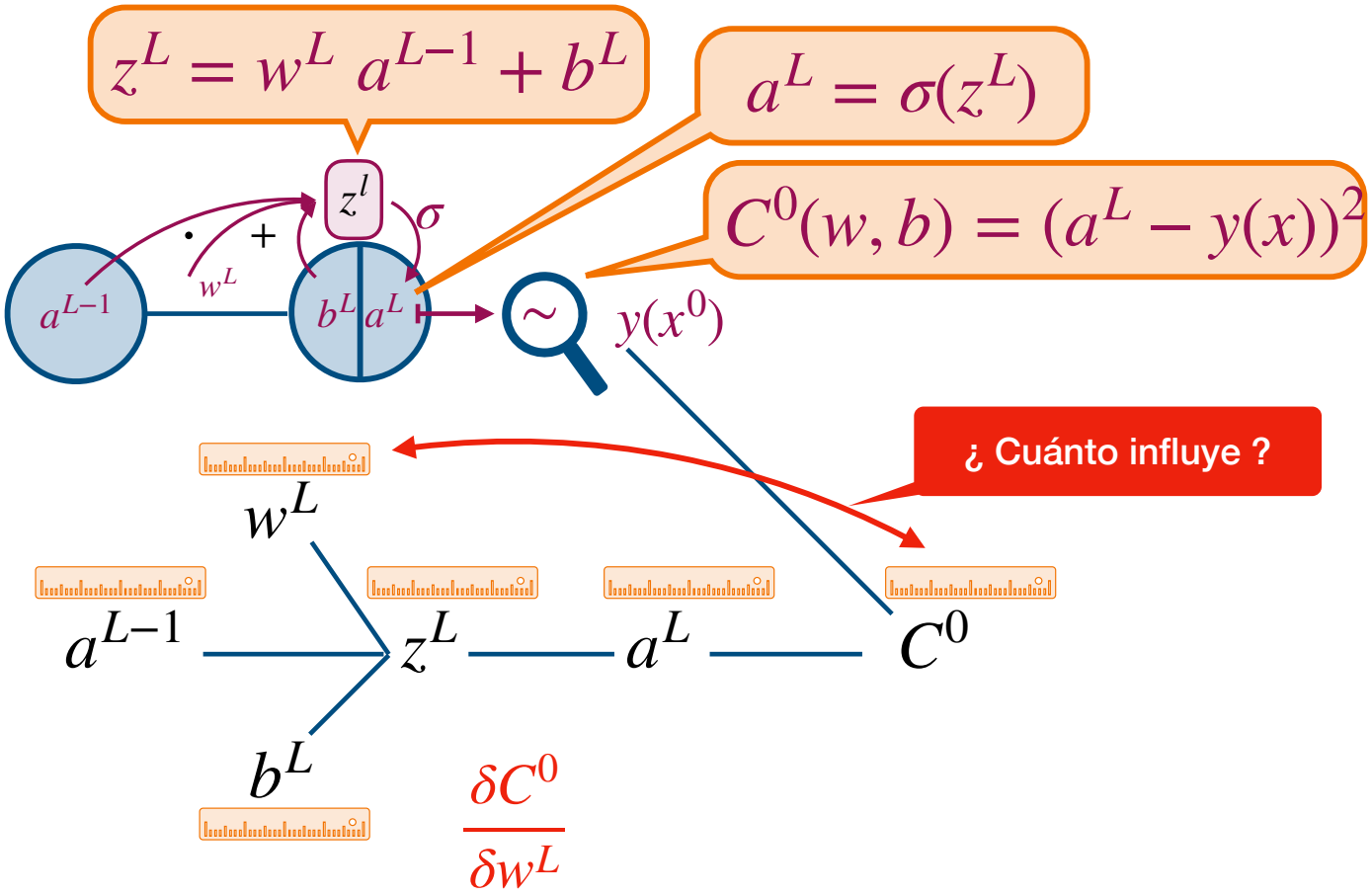
Redes Neuronales - Backpropagation

Para entender cómo funciona comencemos con entendiendo la relación de los componentes de una red respecto del costo para una red simple y con un solo caso de entrenamiento x^0.



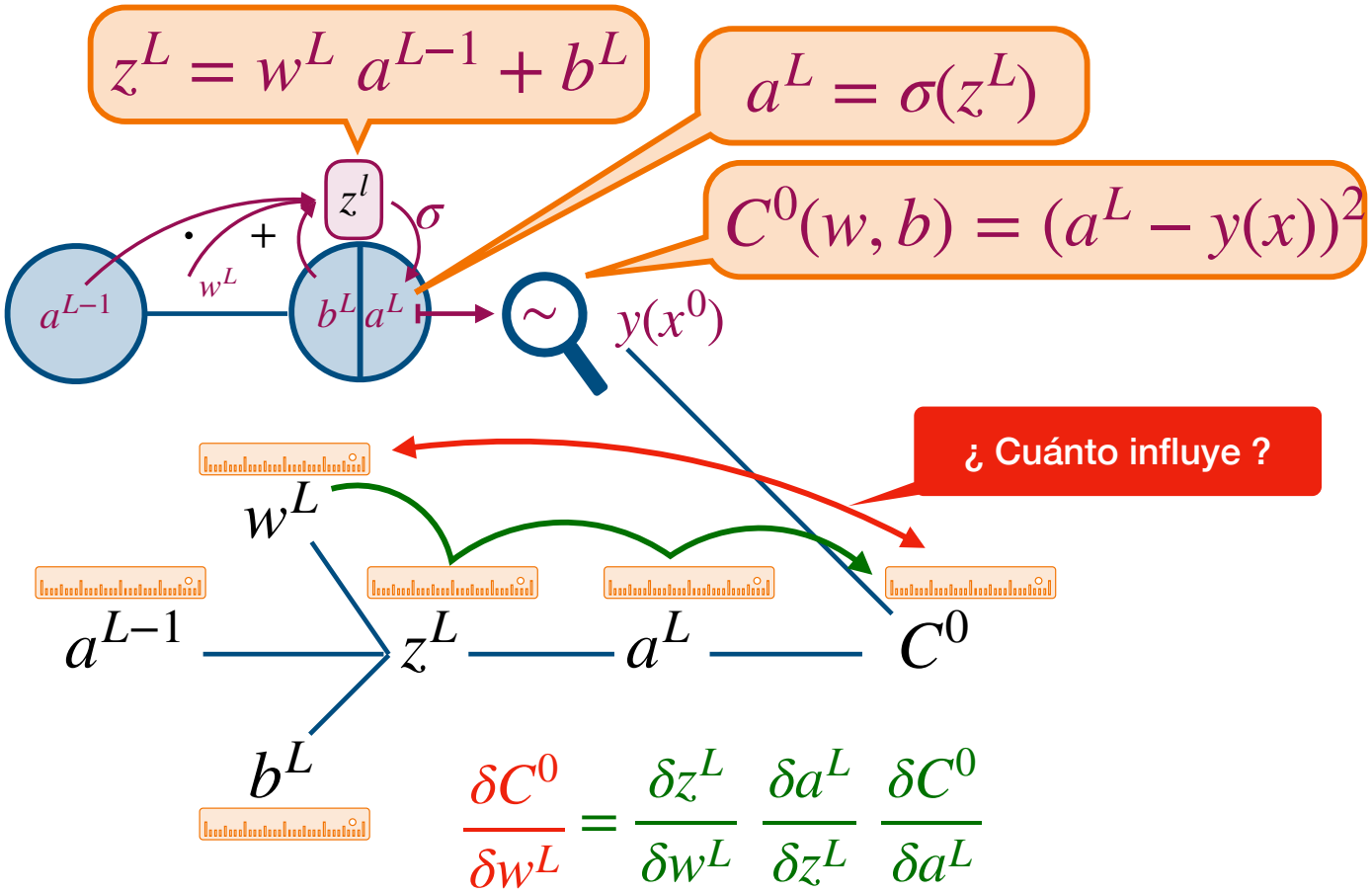
Redes Neuronales - Backpropagation

Para entender cómo funciona comencemos con entendiendo la relación de los componentes de una red respecto del costo para una red simple y con un solo caso de entrenamiento x^0.



Redes Neuronales - Backpropagation

Para entender cómo funciona comencemos con entendiendo la relación de los componentes de una red respecto del costo para una red simple y con un solo caso de entrenamiento x^0.



Redes Neuronales - Backpropagation

Para entender cómo funciona comencemos con entendiendo la relación de los componentes de una red respecto del costo para una red simple y con un solo caso de entrenamiento x^0.

$$\frac{\delta C^0}{\delta a^L} = 2(a^L - y(x^0))$$

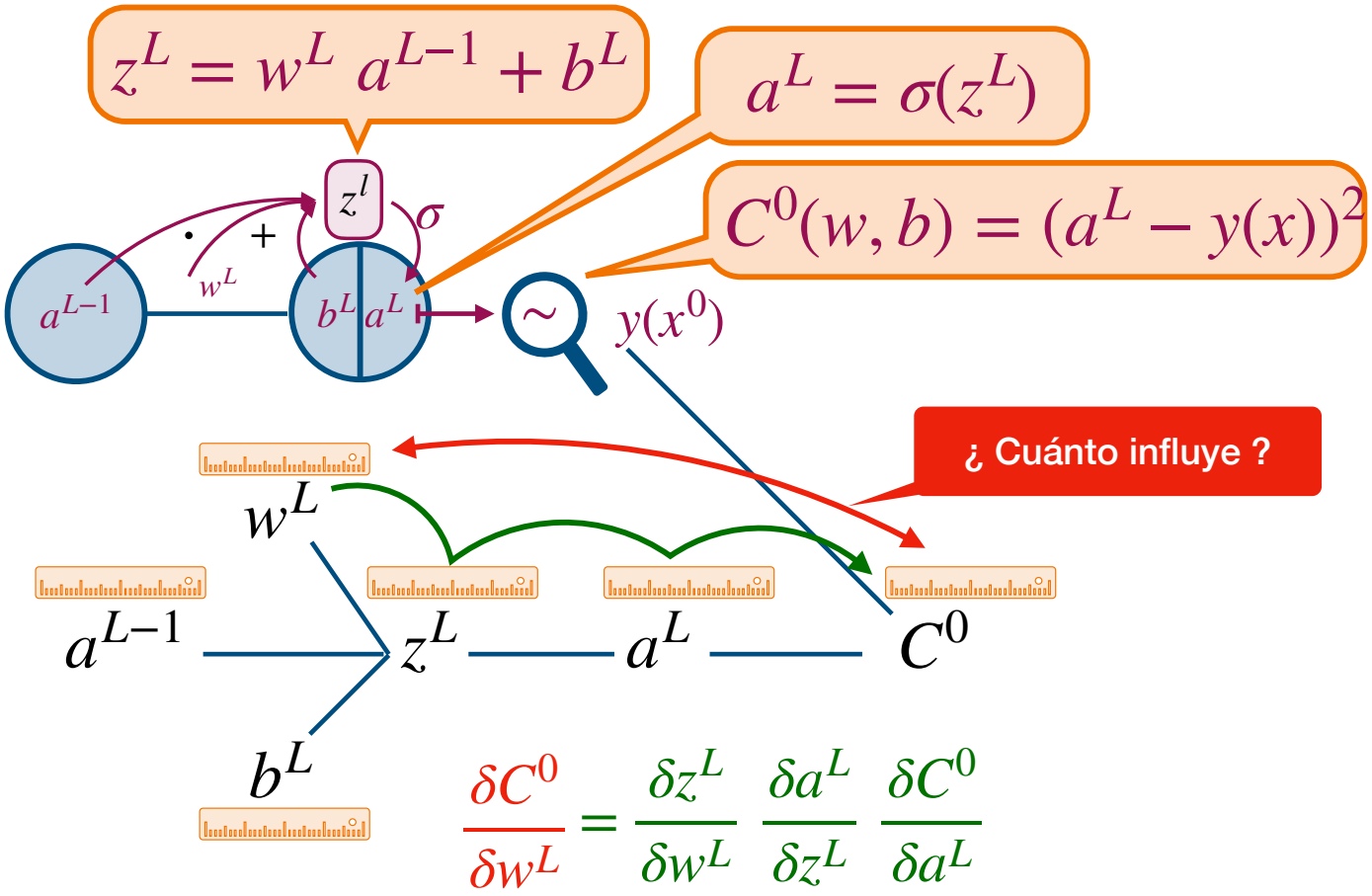
$$\frac{\delta a^L}{\delta z^L} = \sigma'(z^L)$$

$$\frac{\delta z^L}{\delta w^L} = a^{L-1}$$

$$\frac{\delta C^0}{\delta w^L} = a^{L-1}\ \sigma'(z^L)\ 2(a^L - y(x^0))$$

$$\frac{\delta C^0}{\delta b^L} = 1\ \sigma'(z^L)\ 2(a^L - y(x^0))$$

$$\frac{\delta C^0}{\delta a^{L-1}} = w^l\ \sigma'(z^L)\ 2(a^L - y(x^0))$$

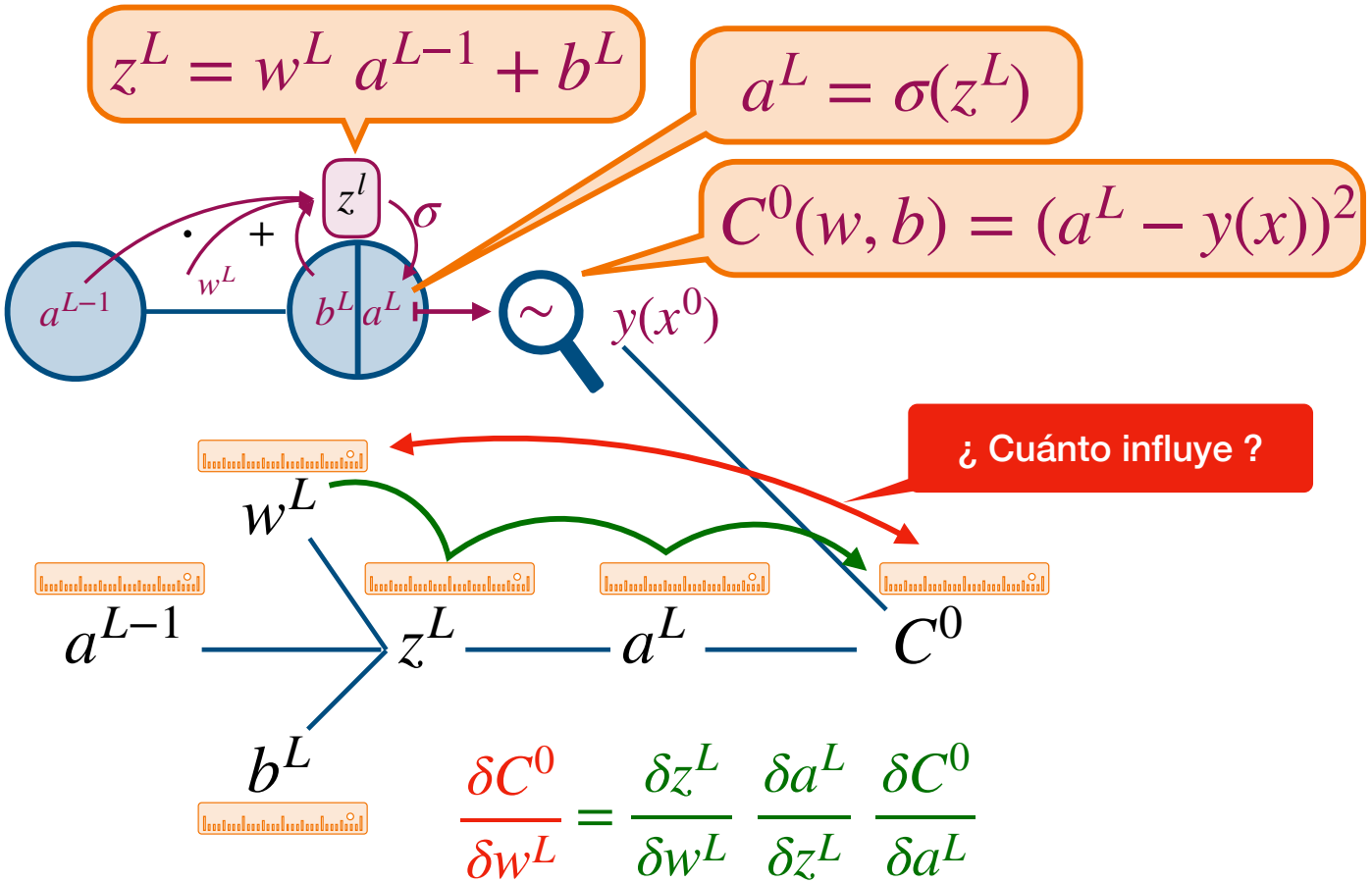


Redes Neuronales - Backpropagation

Para entender cómo funciona comencemos con entendiendo la relación de los componentes de una red respecto del costo para una red simple y con un solo caso de entrenamiento.

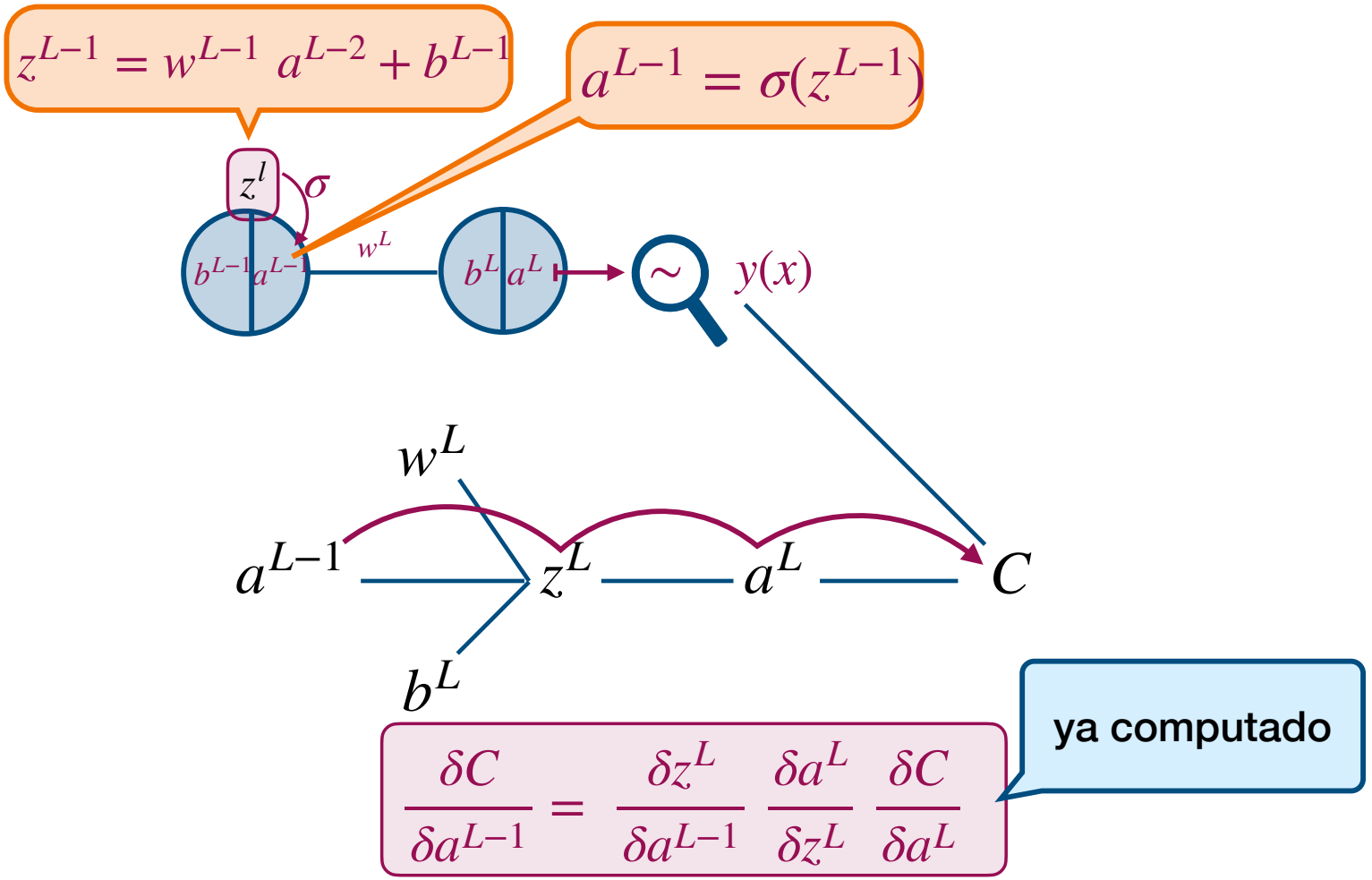
$$C(w,b) = \frac{1}{n}\sum_{i=1}^{n}(a^L - y(x^i))^2$$

$$\frac{\delta C}{\delta w^L} = \frac{1}{n}\sum_{i=1}^{n}\frac{\delta C^i}{\delta w^L}$$



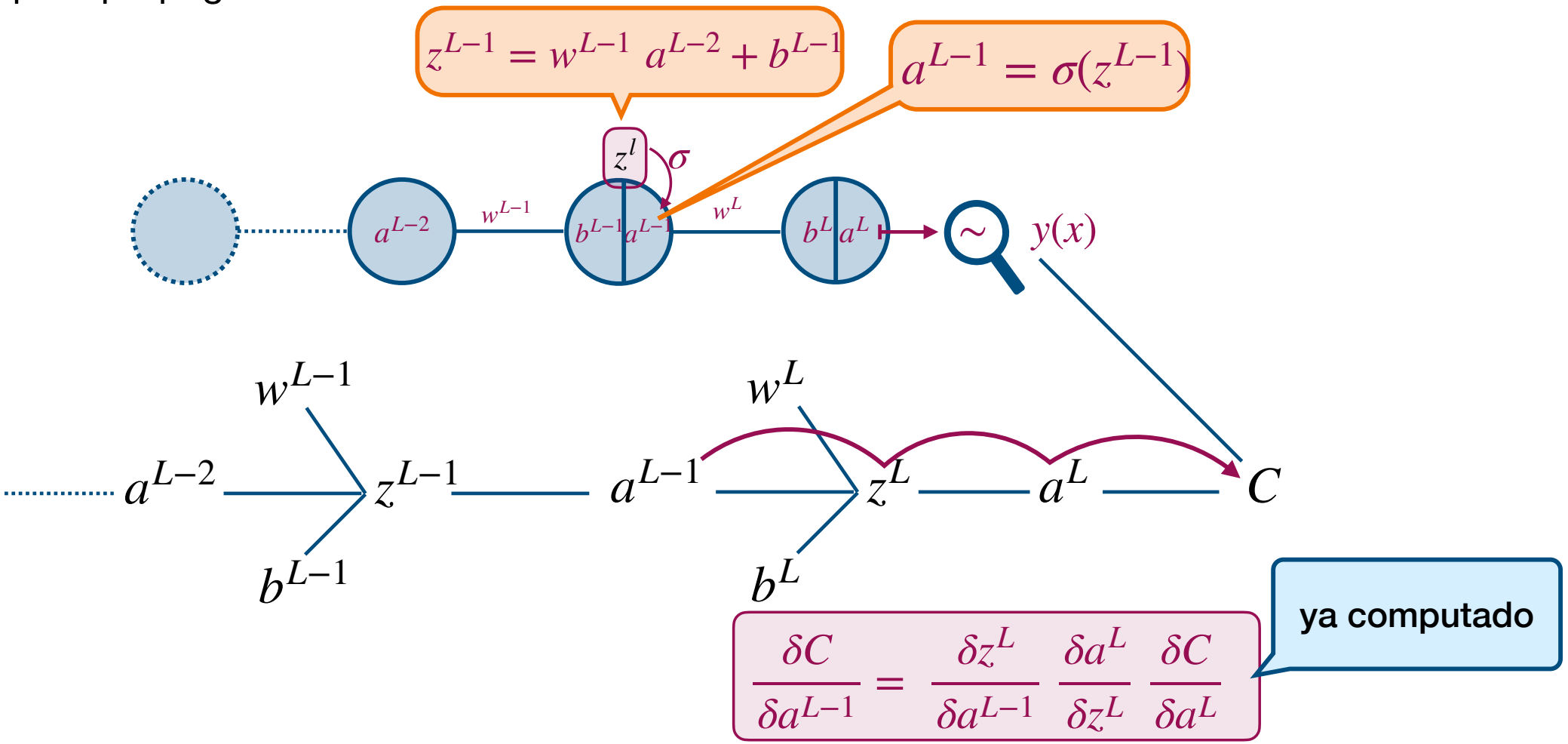
Redes Neuronales - Backpropagation

Una vez que computamos y ajustamos las activación de la capa previa, ahora utilizamos el valor para propagar hacia atrás



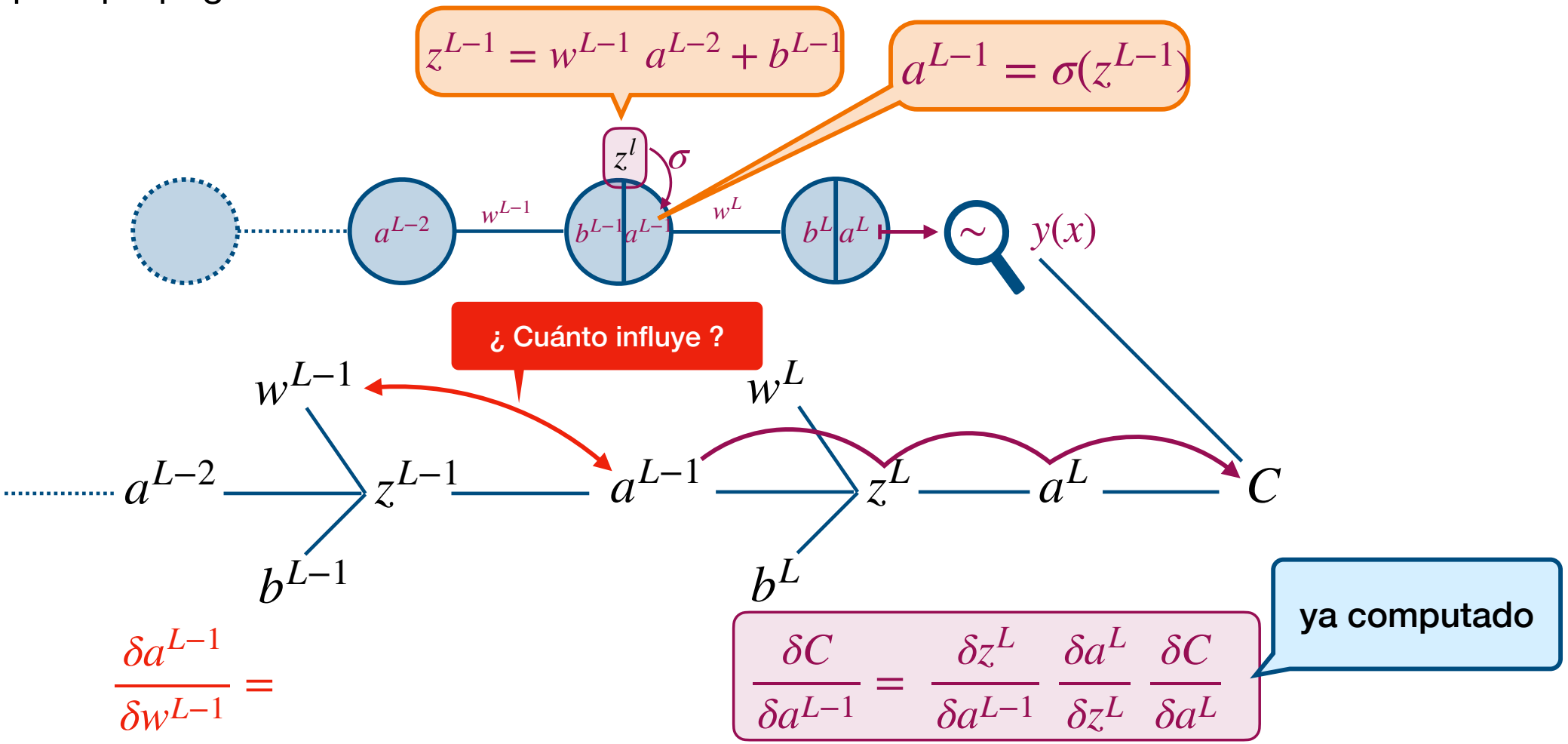
Redes Neuronales - Backpropagation

Una vez que computamos y ajustamos las activación de la capa previa, ahora utilizamos el valor para propagar hacia atrás



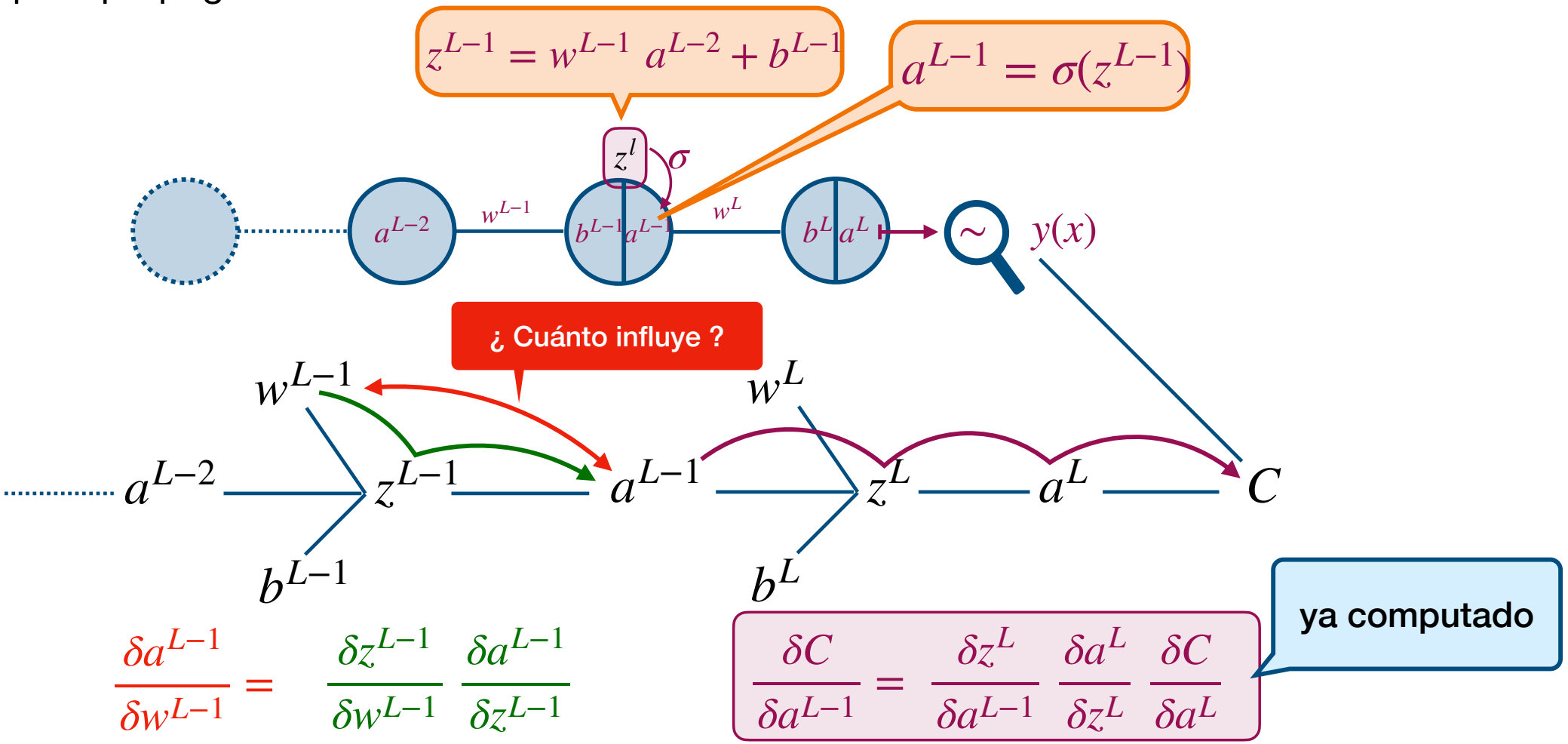
Redes Neuronales - Backpropagation

Una vez que computamos y ajustamos las activación de la capa previa, ahora utilizamos el valor para propagar hacia atrás



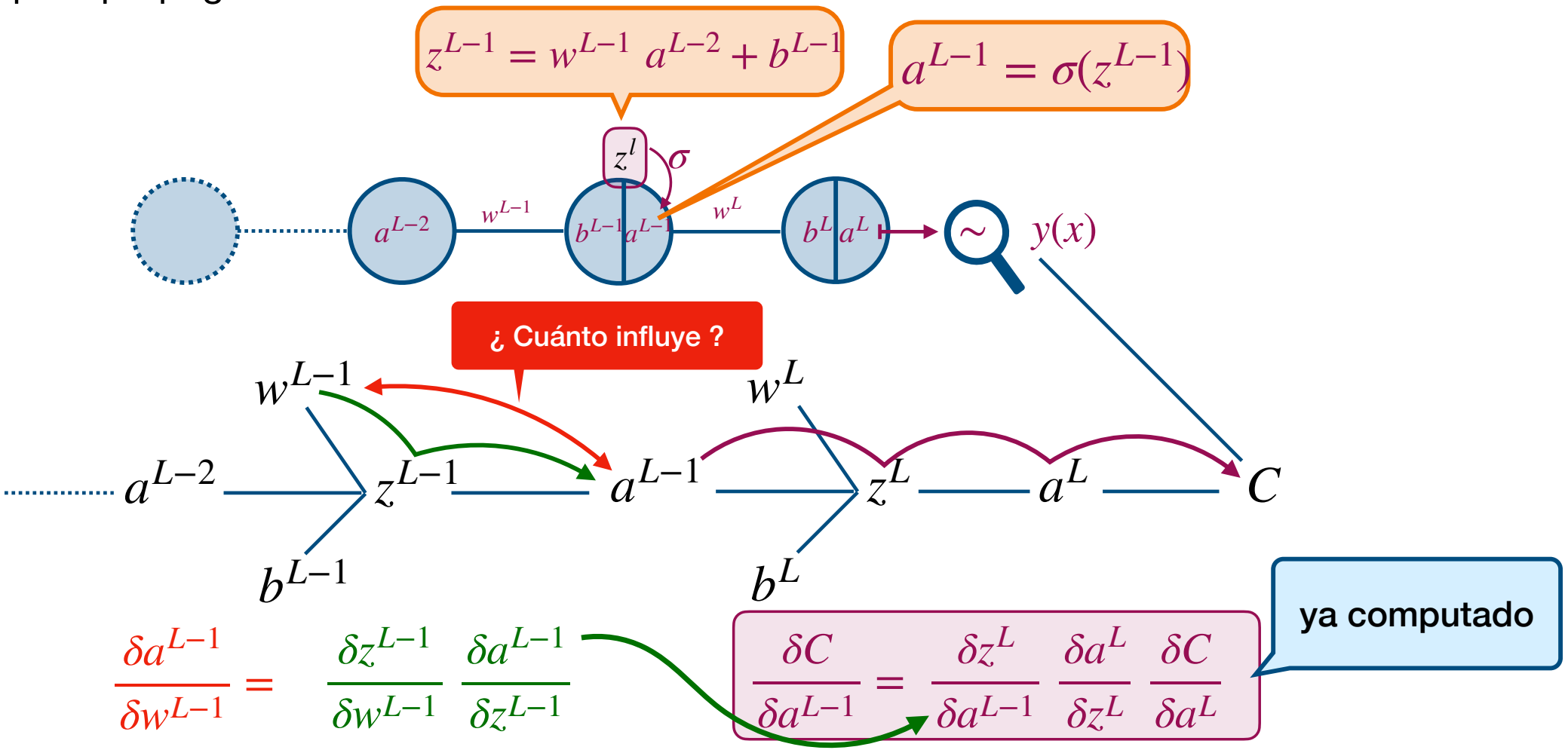
Redes Neuronales - Backpropagation

Una vez que computamos y ajustamos las activación de la capa previa, ahora utilizamos el valor para propagar hacia atrás



Redes Neuronales - Backpropagation

Una vez que computamos y ajustamos las activación de la capa previa, ahora utilizamos el valor para propagar hacia atrás

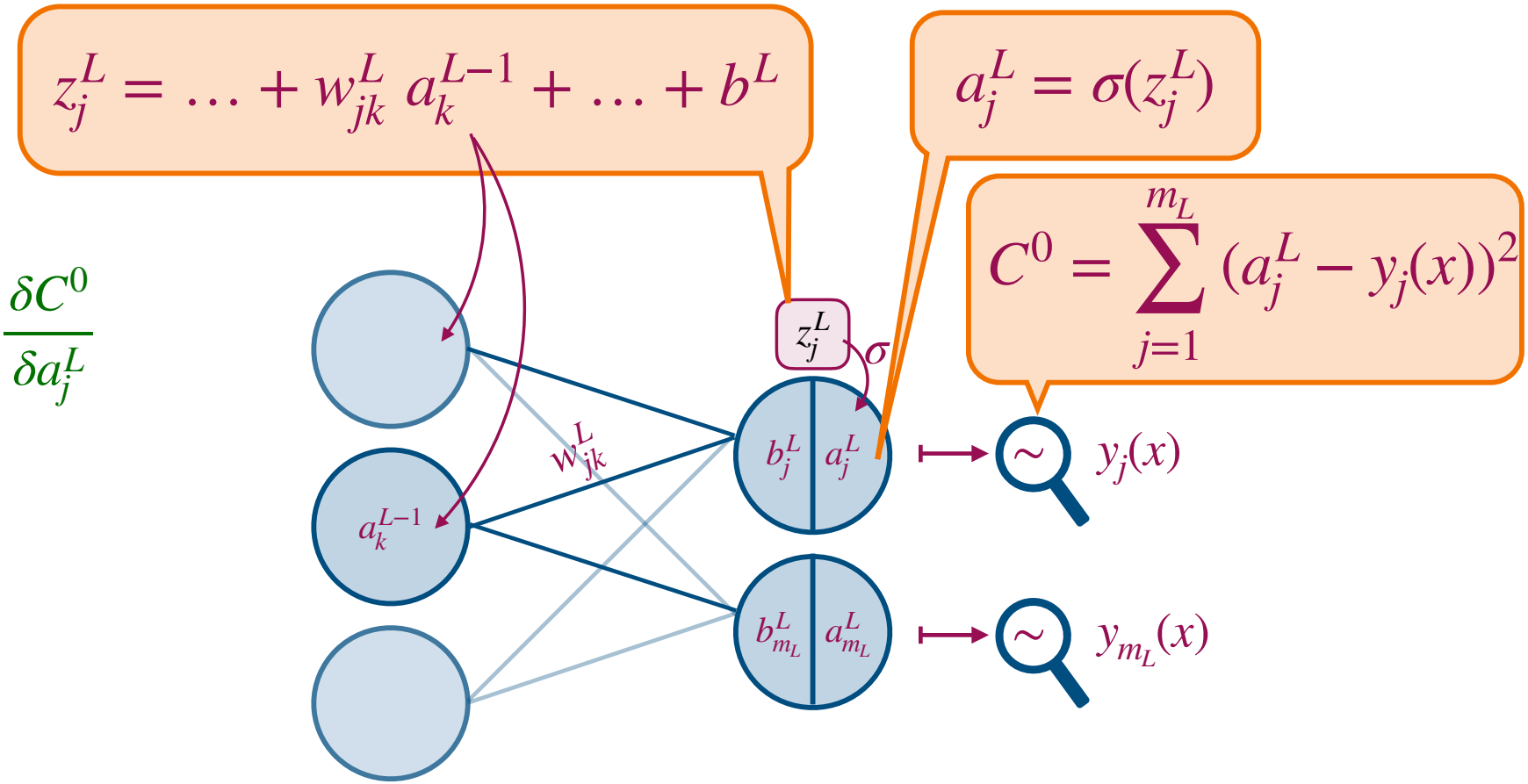


Redes Neuronales - Backpropagation

Cómo podemos generalizar la idea para redes con más de una neurona por layer

$$\frac{\delta C^0}{\delta w^L_{jk}} = \frac{\delta z^L_j}{\delta w^L_{jk}} \frac{\delta a^L_j}{\delta z^L_j} \frac{\delta C^0}{\delta a^L_j}$$

$$\frac{\delta C^0}{\delta a^{L-1}_k}$$



Redes Neuronales - Backpropagation

Cómo podemos generalizar la idea para redes con más de una neurona por layer

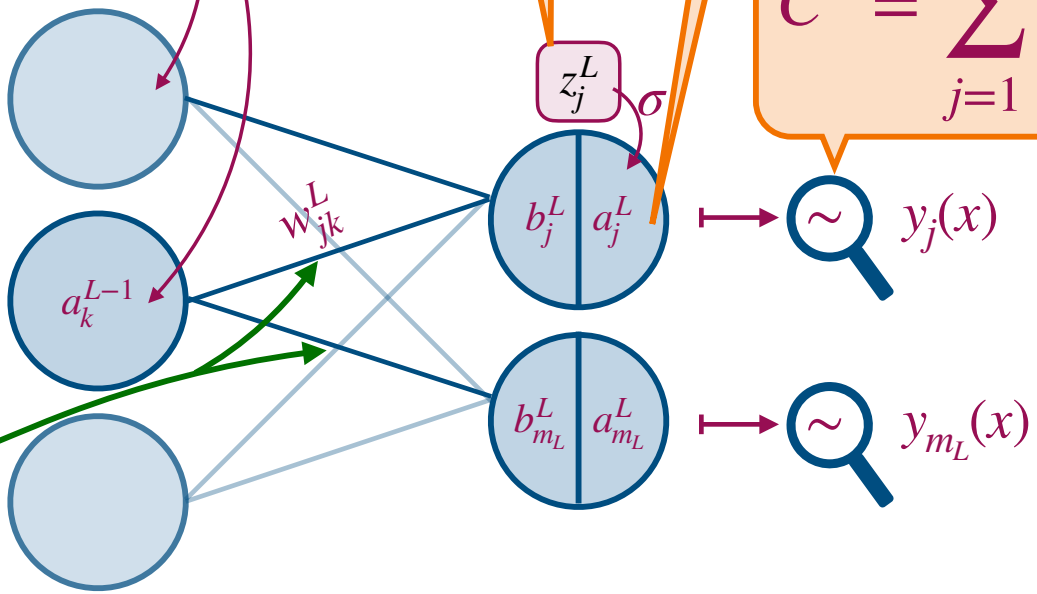
$$z_j^L = \ldots + w_{jk}^L \, a_k^{L-1} + \ldots + b^L$$

$$a_j^L = \sigma(z_j^L)$$

$$C^0 = \sum_{j=1}^{m_L} (a_j^L - y_j(x))^2$$

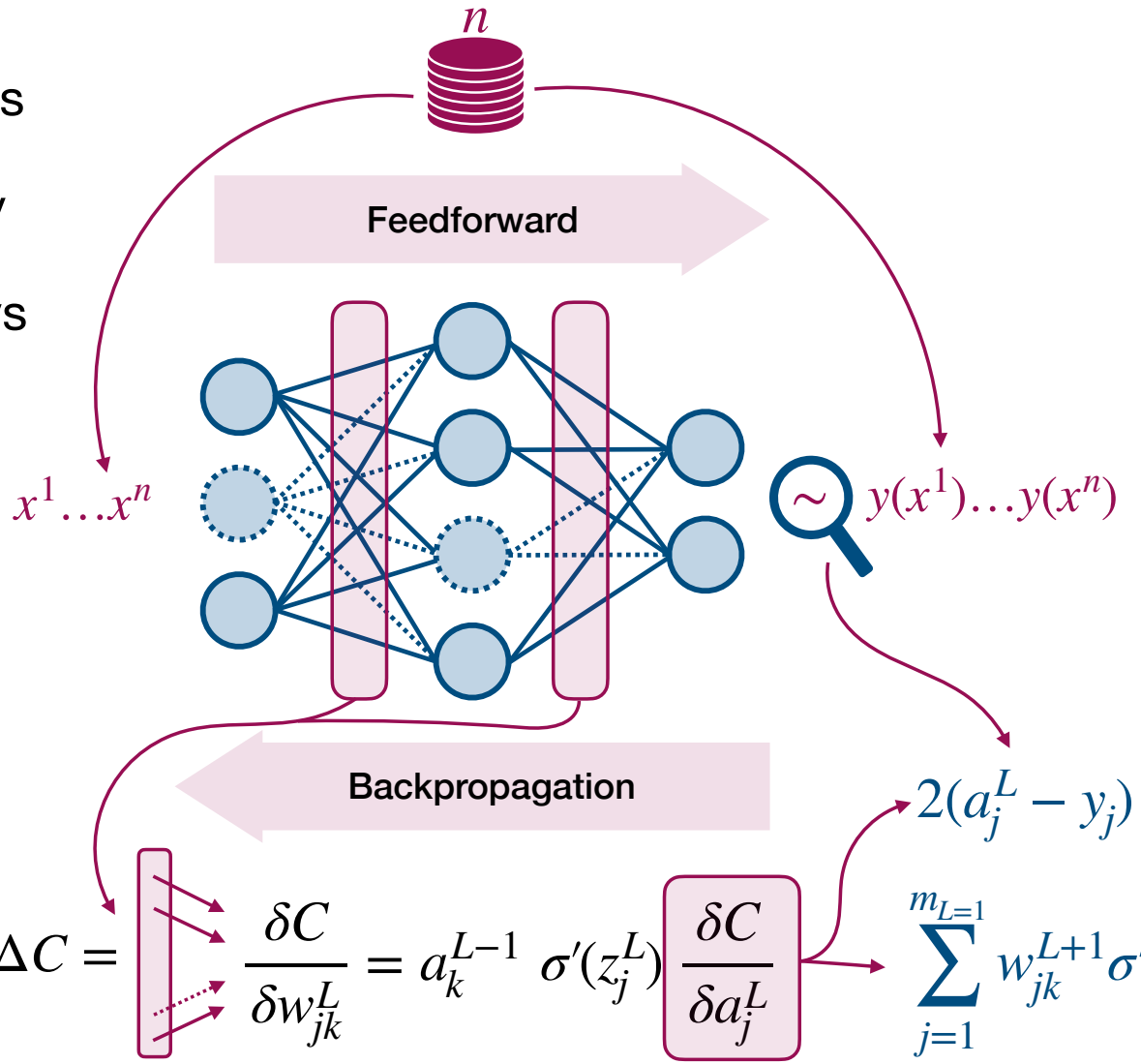
$$\frac{\delta C^0}{\delta w_{jk}^L} = \frac{\delta z_j^L}{\delta w_{jk}^L} \frac{\delta a_j^L}{\delta z_j^L} \frac{\delta C^0}{\delta a_j^L}$$

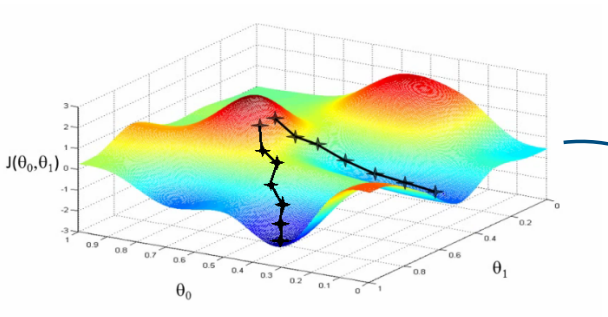
$$\frac{\delta C^0}{\delta a_k^{L-1}} = \sum_{j=1}^{m_L} \frac{\delta z_j^L}{\delta a_k^{L-1}} \frac{\delta a_j^L}{\delta z_j^L} \frac{\delta C^0}{\delta a_j^L}$$



Redes Neuronales - Entrenamiento

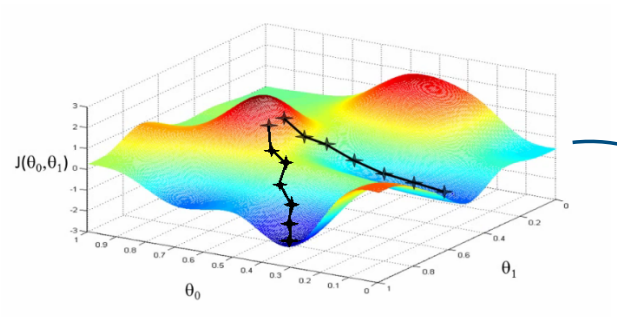
Para entrenar la red, calculamos la salida para cada caso de entrenamiento (**Feedforward**) y luego, en contraste con el resultado del caso de estudio vs el de la red, computamos los pesos y sesgos mediante la retropropagación del error (**Backpropagation**).

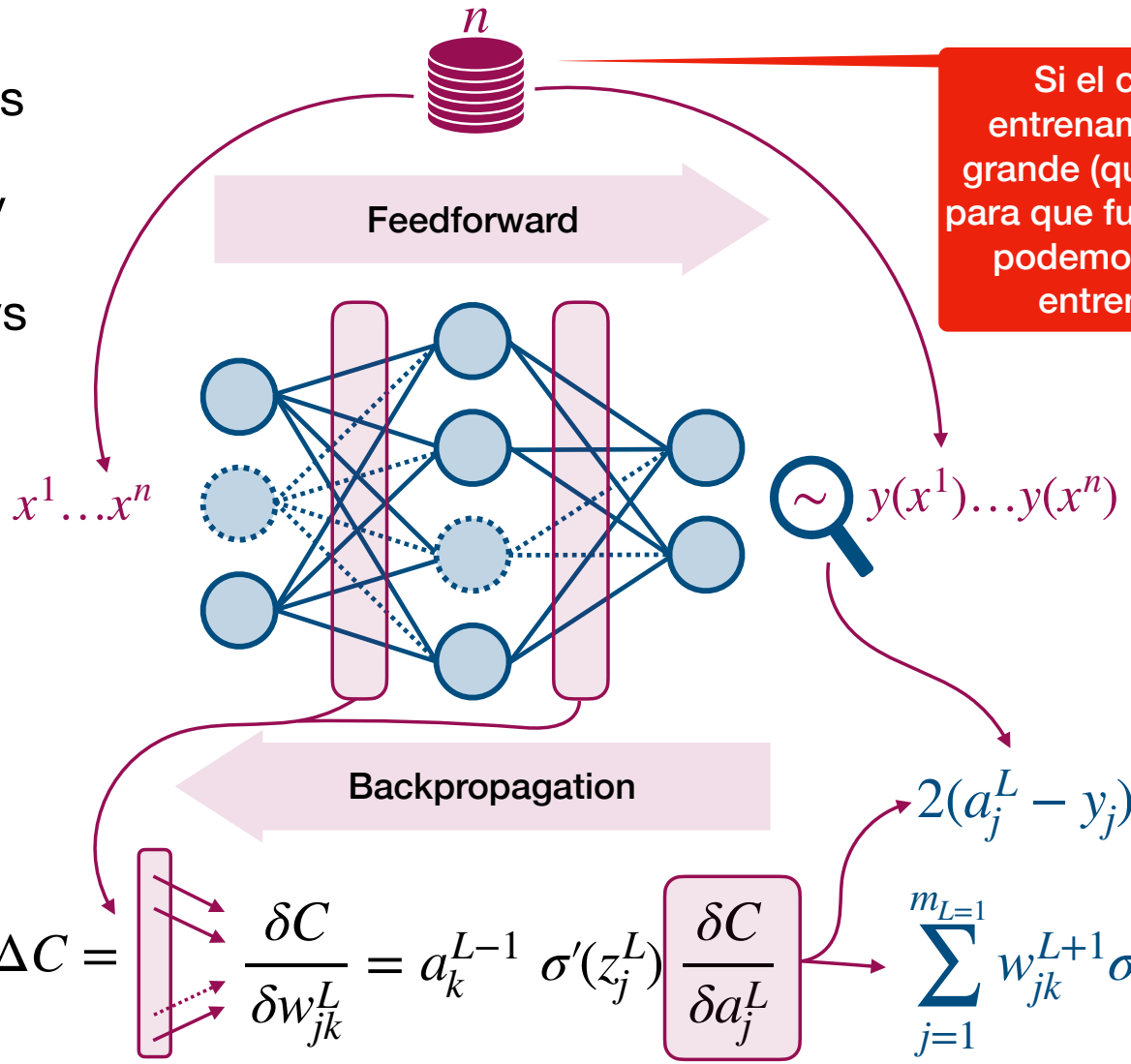




Redes Neuronales - Entrenamiento

Para entrenar la red, calculamos la salida para cada caso de entrenamiento (**Feedforward**) y luego, en contraste con el resultado del caso de estudio vs el de la red, computamos los pesos y sesgos mediante la retropropagación del error (**Backpropagation**).





Si el conjunto de entrenamiento es muy grande (que es necesario para que funcione), ¿ cómo podemos optimizar el entrenamiento ?

Redes Neuronales - Entrenamiento estocástico

Convergencia	Gradual directa (poca varianza entre iteraciones)	Gradual (varianza media entre iteraciones)	Gradual (alta varianza entre iteraciones)
Hiperparámetros	Tasa de aprendizaje	Tasa de aprendizaje, tamaño de batch, etc.	Tasa de aprendizaje, etc.
Uso de datos de entrenamiento	exhaustivo	reducido (ajustable)	mínimo
Comportamiento con muchas instancias	Lento	Rápido	Rápido

Redes Neuronales - Entrenamiento - Inicialización y Learning rate

Es conveniente inicializar los parámetros (w) de la red con valores aleatorios. Comúnmente se usan distribuciones uniformes en $[-a, a]$ o una distribución normal $\mathcal{N}(0, s^2)$. En en éste último caso la elección de s es crítica. Es común usar He-initialization: $s = \sqrt{\frac{2}{m}}$, donde m en número de inputs de cada neurona.

La elección del parámetro η (learning rate es fundamental para lograr convergencia y evitar inestabilidad (saltos entre valles de la función de error)

- Un valor de η demasiado pequeño el aprendizaje demorará más
- Un valor grande de η podrá generar inestabilidad en el proceso
- Una estrategia común es comenzar con un η grande e ir decrementándolo en los próximos ciclos

Redes Neuronales - Entrenamiento - Normalización

La normalización de los datos (llevarlos a magnitudes similares) evita que se tengan que manejar datos con valores extremadamente grandes o extremadamente pequeños:

- Permite una convergencia más rápida, mejora el flujo de gradientes y reduce el sesgo
- Normalización de los datos de entrada:
  - Mini-max scaling: $xn = (x - xmin) / (xmax - xmin)$
  - Estandarización: $xn = (x - mean(X)) / stddev(x)$
- Batch normalization: Normalización aplicada en cada capa oculta

Redes Neuronales - Entrenamiento - Regularización

Regularización L1 y L2 (Weight Penalties): Se agregan términos adicionales a la función de pérdida para penalizar pesos grandes

- **L1 (Lasso)**: penaliza la suma de valores absolutos de los pesos (algunos pesos quedan en 0)

$$L_{reg} = L + \lambda \sum |w_i|$$

- **L2 (Ridge)**: penaliza la suma de los cuadrados de los pesos. Evita pesos demasiado grandes y distribuye la carga entre ellos. Útiles para redes pequeñas.

$$L_{reg} = L + \lambda \sum w_i^2$$

Dropout: Durante el entrenamiento, **se apaga aleatoriamente un porcentaje de neuronas** en cada forward pass.

- Evita que las neuronas dependan demasiado entre sí (**co-adaptación**).
- Ejemplo: con dropout=0.5, cada neurona tiene 50% de probabilidad de “apagarse”.
- Al testear, se usan todas las neuronas pero se escalan los pesos.

Redes Neuronales - Funciones de activación

Las funciones de activación transforman la salida de cada neurona antes de pasarla a la siguiente capa. Permiten **no linealidad** y que la red aprenda relaciones complejas. Se pueden configurar diferentes activadores en las diferentes capas de una ANN.

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

Rango: (0, 1). Ventajas: útil en probabilidades binarias. Desventajas: problema de *vanishing gradient* (los gradientes pequeños desaparecen en la retropropagación). Generalmente se utiliza capa de salida en clasificación binaria.

Tanh (Tangente hiperbólica): $\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$

Rango: (-1, 1). Ventajas: centrada en 0 (mejora convergencia respecto a sigmoide). También sufre *vanishing gradient*.

ReLU (Rectified Linear Unit): $f(x) = \max(0,x)$

Rango: [0, ∞). Ventajas: computación sencilla, evita saturación en parte positiva. Desventaja: *dying ReLU* (neuronas se apagan si reciben valores negativos constantemente). Generalmente se utilizan capas ocultas en redes profundas.

Redes Neuronales - Funciones de activación (cont.)

Leaky ReLU: $f(x) = \begin{cases} x & x > 0 \\ \alpha x & x \leq 0 \end{cases}$

Introduce una pendiente pequeña en valores negativos (ej. α=0.01). Evita el problema de *dying ReLU*.

Softmax: $\textbf{softmax}(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}$

Convierte un vector de scores en una **distribución de probabilidad**. Rango: (0,1) y suma total = 1. Se utiliza generalmente en la capa de salida en clasificación multiclase.

ELU (Exponential Linear Unit): similar a ReLU pero suaviza la parte negativa.

Swish: $f(x) = x \cdot \sigma(\beta x) = \frac{x}{1 + e^{-\beta x}}$

Mejora suavidad y propagación de gradientes. Generalmente utilizada en transformes y CNN.

GELU (Gaussian Error Linear Unit): variante más avanzada de ReLU usada en Transformers. Funciona mejor en modelos grandes, muy utilizada en NLP.

Redes Neuronales - Optimizadores

Los optimizadores son los algoritmos que ajustan los pesos de la red en función del error que se comete en cada iteración:

Gradient Descent (GD): es la versión básica que actualiza los pesos en la dirección del gradiente negativo. Funciona bien en problemas simples, pero es lento y sensible a la tasa de aprendizaje.

Stochastic Gradient Descent (SGD): mejora al GD usando muestras individuales o pequeños lotes, lo que acelera el entrenamiento y aporta cierta aleatoriedad que ayuda a escapar de mínimos locales. Sin embargo, puede oscilar mucho.

SGD con Momentum: añade una "inercia" a las actualizaciones, acumulando gradientes pasados para suavizar los cambios y acelerar la convergencia, especialmente en superficies irregulares.

Redes Neuronales - Optimizadores (cont.)

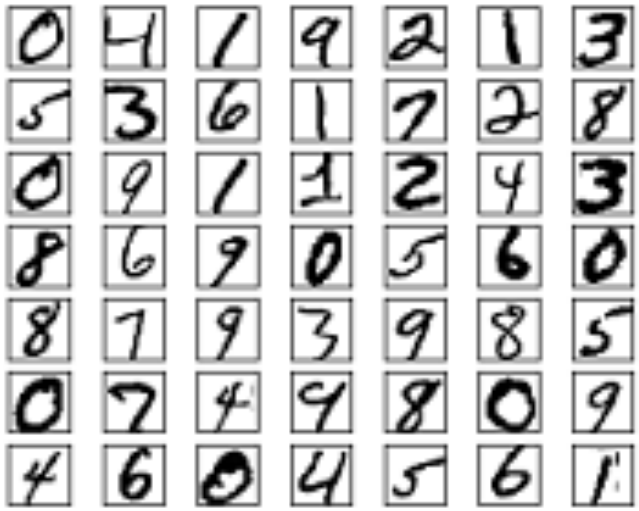
Adagrad: adapta la tasa de aprendizaje para cada parámetro, reduciéndola automáticamente en función de su frecuencia de actualización. Es muy útil para datos dispersos, pero su desventaja es que la tasa puede disminuir demasiado y frenar el aprendizaje.

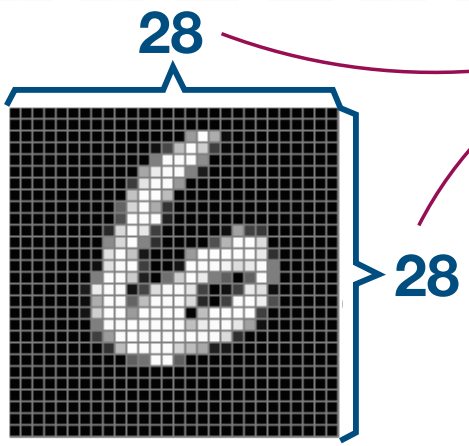
RMSProp: corrige el problema de Adagrad manteniendo un promedio móvil de los gradientes recientes, lo que permite entrenar de manera más sostenida. Es muy usado en redes profundas y en problemas como RNNs.

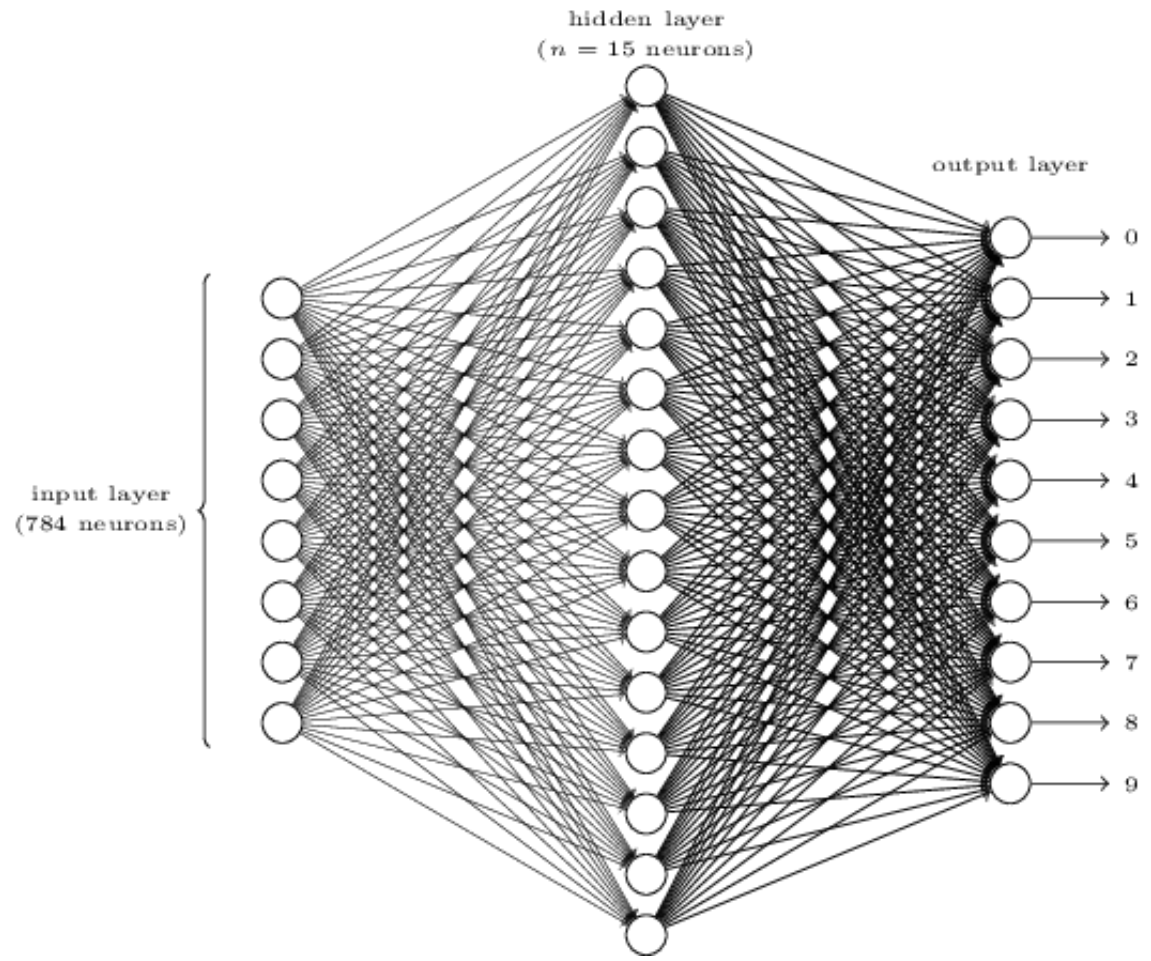
Adam (Adaptive Moment Estimation): combina la idea de momentum y RMSProp, guardando tanto la media como la varianza de los gradientes. Es uno de los optimizadores más populares por su rapidez y robustez.

Redes Neuronales - From Scratch con ejemplo MNIST

https://neuralnetworksanddeeplearning.com/chap1.html







Redes Neuronales - Usando Scikitlear

```python
from sklearn.datasets import fetch_openml
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import classification_report, accuracy_score
import matplotlib.pyplot as plt
# obtener el dataset de openml
mnist = fetch_openml('mnist_784', version=1, as_frame=False)
X, y = mnist["data"], mnist["target"].astype(int)
# Normalizar datos (0-1) (están en esacala de grises
X = X / 255.0
# Dividir en entrenamiento y prueba
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
# Crear y entrenar red neuronal (Multilayer perceptron)
mlp = MLPClassifier(
    hidden_layer_sizes=(128, 64),  # 2 capas ocultas
    activation="relu",
    solver="adam",
    alpha=1e-4,  # regularización L2
    learning_rate_init=0.001,
    max_iter=20, # cantidad de iteraciones
    verbose=True,
    random_state=42
)
mlp.fit(X_train, y_train)
# Evaluación
y_pred = mlp.predict(X_test)
print("\nAccuracy:", accuracy_score(y_test, y_pred))
print("\nReporte de clasificación:\n", classification_report(y_test, y_pred))
# Mostrar algunas predicciones
fig, axes = plt.subplots(1, 5, figsize=(10, 3))
for i, ax in enumerate(axes):
    ax.imshow(X_test[i].reshape(28, 28), cmap="gray")
    ax.set_title(f"Pred: {y_pred[i]}")
    ax.axis("off")
plt.show()
```