



OTRAS ESPECIFICACIONES PARA REDES



1. FUNCIONES DE ACTIVACIÓN

Sigmoid

Salida en el rango (0,1)

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

Se interpretaba como probabilidad → usada en la salida de redes para clasificación binaria.

Problemas:

- Saturación en extremos (derivada ≈ 0)
- **Vanishing gradients**



Hoy día casi no se usa en capas ocultas.

¿Vanishing Gradient?

El gradiente de cada capa incluye el **producto de muchas derivadas de funciones de activación**.

$$\delta^{[l]} = \left(W^{[l+1]\top} \delta^{[l+1]} \right) \circ \sigma'(z^{[l]})$$

Si estas derivadas son menores que 1, el producto total **tiende a cero** a medida que la red se profundiza.

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

$$\sigma'(z) = \sigma(z)(1 - \sigma(z)) \in (0, 0.25)$$

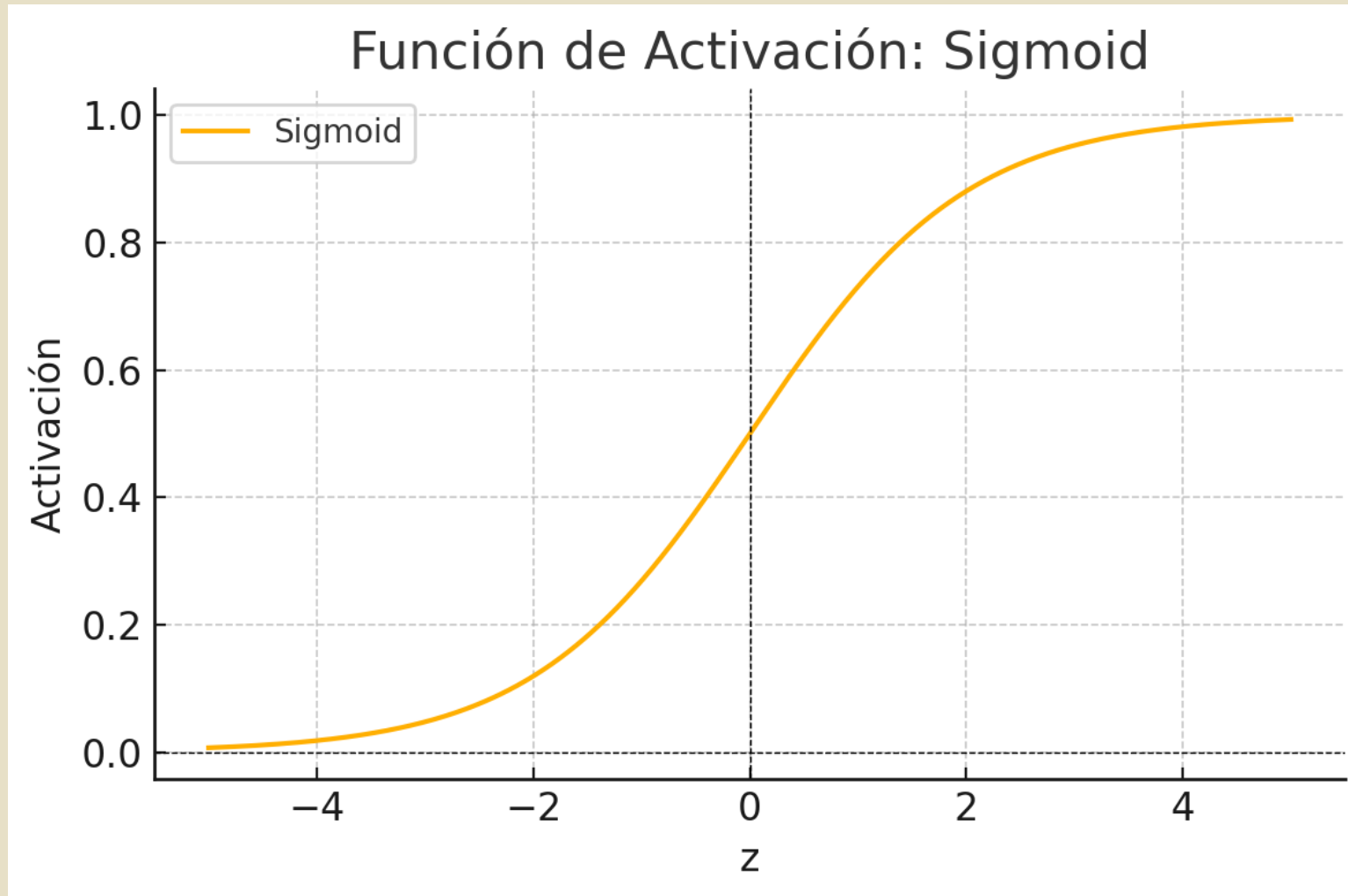
Entonces, si tienes una red de 20 capas y multiplicas derivadas pequeñas:

$$0.25^{20} \approx 9.1 \times 10^{-13}$$

Consecuencias del vanishing gradient:

- Redes profundas no aprenden bien.
- Mucho tiempo de entrenamiento, poca mejora.
- Inicialización de pesos se vuelve crítica.
- Con funciones como sigmoide o tanh, **es difícil entrenar más de 4–5 capas sin técnicas adicionales.**

Sigmoid – se satura fácil




Tanh

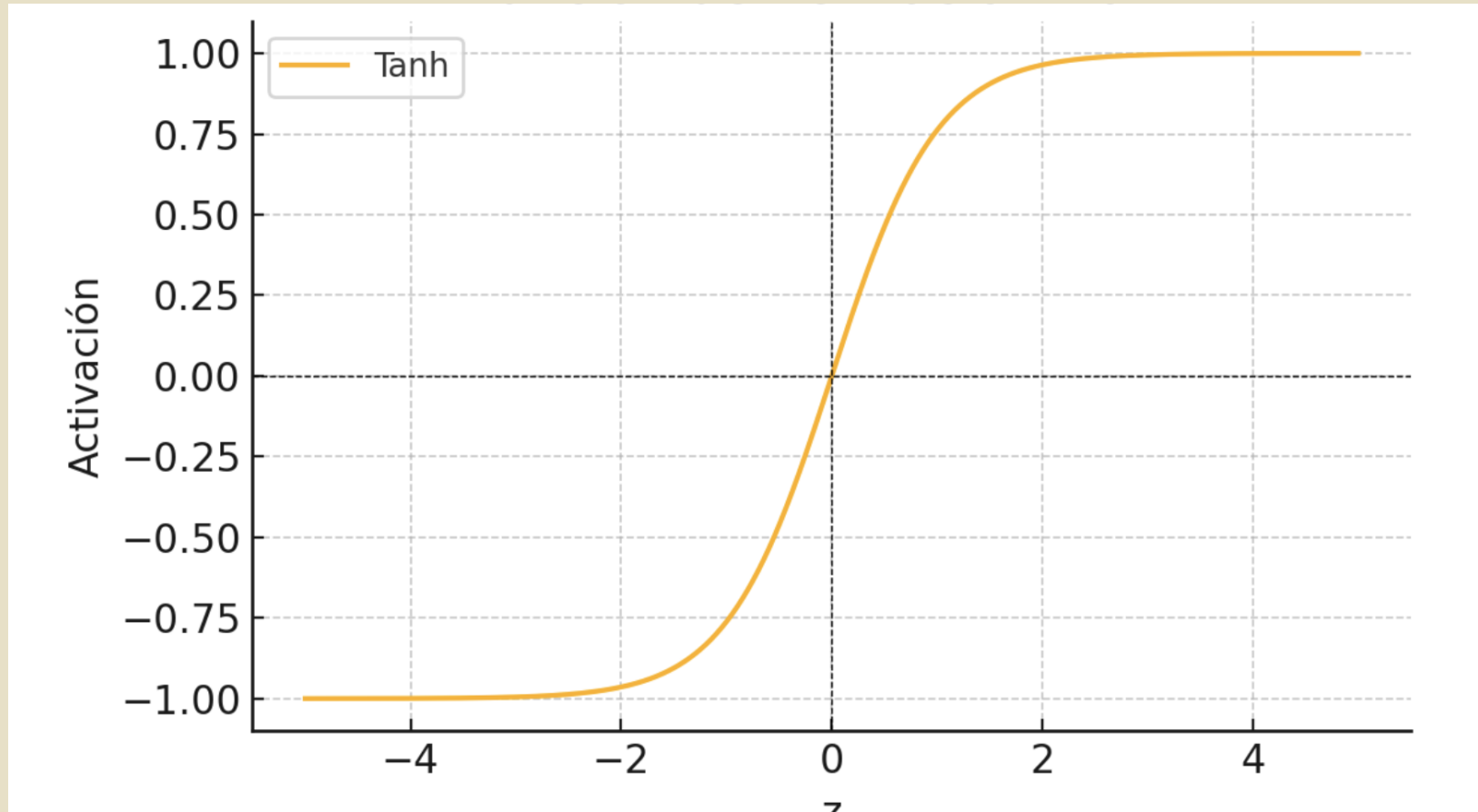
(tangente
hiperbólica)

Salida en el rango $(-1,1)$

$$\sigma(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

- **Ventaja:** centrada en cero \rightarrow mejor que sigmoid en muchas tareas
- **Desventaja:** también puede saturar \rightarrow *vanishing gradient*
-  Se usa aún en algunas RNNs (aunque se prefiere ReLU en general).

Tanh – También se satura fácil



ReLU (Rectified Linear Unit)

Rango: 0 y positivos

$$\sigma(z) = \max(0, z)$$

$$\sigma'(z) = \begin{cases} 1 & z > 0 \\ 0 & z \leq 0 \end{cases}$$

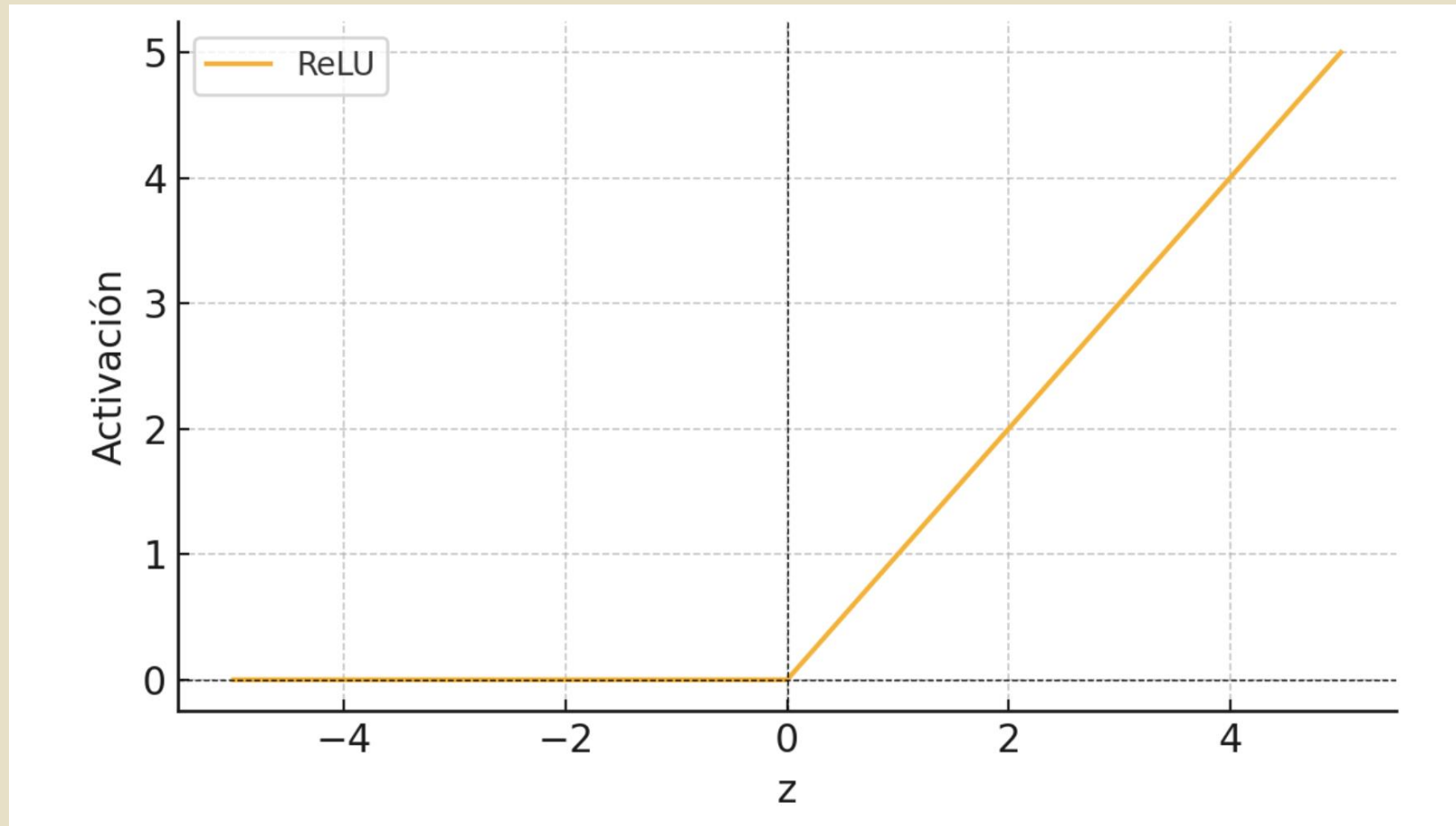
Hace que la red sea **sparse** (muchas neuronas con salida 0), menor costo computacional, menor co-adaptación de neuronas.

- **Ventaja:** Computación muy eficiente, no satura para valores grandes → no hay vanishing gradient.
- **Problema:** Neuronas “muertas” si $z < 0$ para siempre.



Muy usada en CNNs y redes profundas modernas.

ReLU, la saturación es solo para $z < 0$



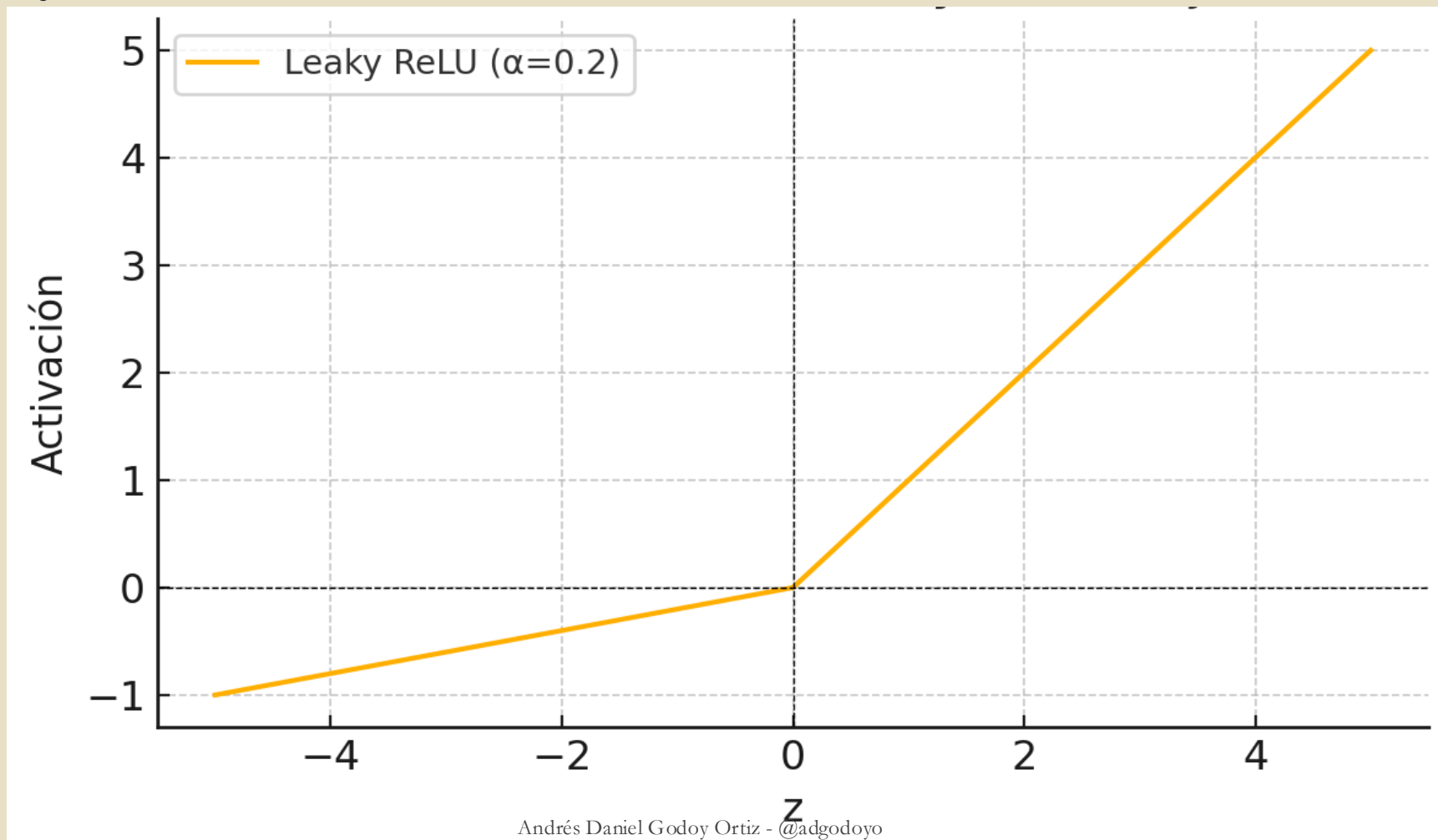
Leaky ReLU

$(-\infty, \infty)$

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

- Variante de ReLU que **no mata** neuronas del todo.
- α suele ser 0.01.
- Mejora la robustez del entrenamiento.

Leaky ReLU



Parametric Leaky ReLU

$(-\infty, \infty)$

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

Como Leaky ReLU, pero α se aprende durante el entrenamiento.



Aumenta flexibilidad → pero también riesgo de overfitting.

GELU (Gaussian Error Linear Unit)

$(-\infty, \infty)$

$$\text{GELU}(z) = z \cdot \Phi(z)$$

$$\Phi(z) = \int_{-\infty}^z \frac{1}{\sqrt{2\pi}} e^{-t^2/2} dt$$

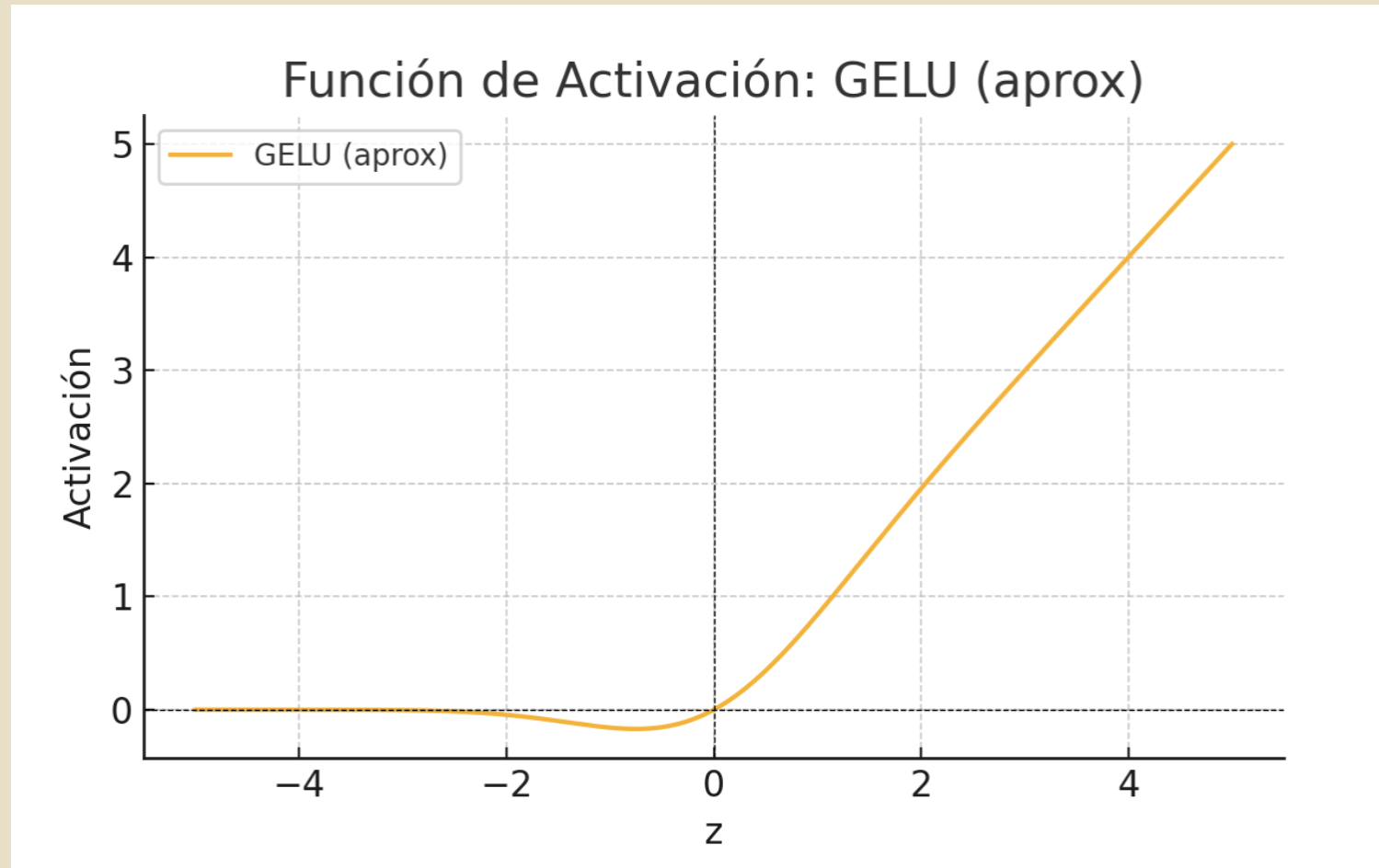
multiplica el valor z por la probabilidad de que una variable normal estándar sea menor que z .

Intuitivamente: **activa proporcionalmente a cuán probable es que esa neurona "importe"** bajo una distribución gaussiana.

Aproximación eficiente:

$$\text{GELU}(z) \approx 0.5z \left(1 + \tanh \left[\sqrt{\frac{2}{\pi}} (z + 0.044715z^3) \right] \right)$$

GELU



¿Por qué se usa en Transformers?

- Aunque más compleja que la de ReLU, GELU tiene una derivada continua, lo que hace que el **descenso del gradiente sea más suave**
- Centrado en 0 → no sesga el flujo hacia positivo como ReLU
- Probabilidad → permite decisiones más informadas a través de activaciones suaves

Activación	Rango	Centrada en 0	Saturación	Vanishing Gradients	Comentarios
Sigmoid	$(0, 1)$	No	Sí	Sí	Solo para salida
Tanh	$(-1, 1)$	Sí	Sí	Sí	Mejor que sigmoid
ReLU	$[0, \infty)$	No	Maso	Maso	Rápida, efectiva, simple
Leaky ReLU	$(-\infty, \infty)$	No	No	No	Suaviza ReLU
GELU	$(-\infty, \infty)$	Sí	Suave	No	Más caro computacionalmente
Softmax	$(0, 1)$	No	No aplica	No aplica	Solo salida multiclase



2. ESCALAR INPUTS

¿Por qué?

¿Cuál es el
problema si no?

Para una red densa con activaciones $a^{[l]} = \sigma(z^{[l]})$, donde $z^{[l]} = W^{[l]}a^{[l-1]} + b^{[l]}$, las dos expresiones clave que usamos en la retropropagación son:

Propagación del error (delta):

$$\delta^{[l]} = \left(W^{[l+1]\top} \delta^{[l+1]} \right) \circ \sigma'(z^{[l]})$$

Gradiente respecto a los pesos:

$$\frac{\partial \mathcal{L}}{\partial W^{[l]}} = \delta^{[l]} \cdot (a^{[l-1]})^\top$$

¿Cómo se relaciona esto con el escalamiento de los inputs?

Observa que en la expresión del gradiente:

$$\frac{\partial \mathcal{L}}{\partial W^{[1]}} = \delta^{[1]} (a^{[0]})^\top$$

... el término $a^{[0]}$ **corresponde directamente a los inputs escalados x** , porque:

$$a^{[0]} = x$$

Entrenamiento inestable si x tiene rangos de valores muy diferentes

Propagación del error y escalamiento

La fórmula de $\delta[l]$ también se ve afectada:

$$\delta^{[l]} = \left(W^{[l+1]\top} \delta^{[l+1]} \right) \circ \sigma'(z^{[l]})$$

Si las activaciones previas $a[l-1]$ (y por ende $z[l]$) no están en un rango razonable (por ejemplo, con media 0 y varianza 1), entonces:

Las derivadas $\sigma'(z[l])$ pueden ser extremadamente pequeñas o cercanas a cero.

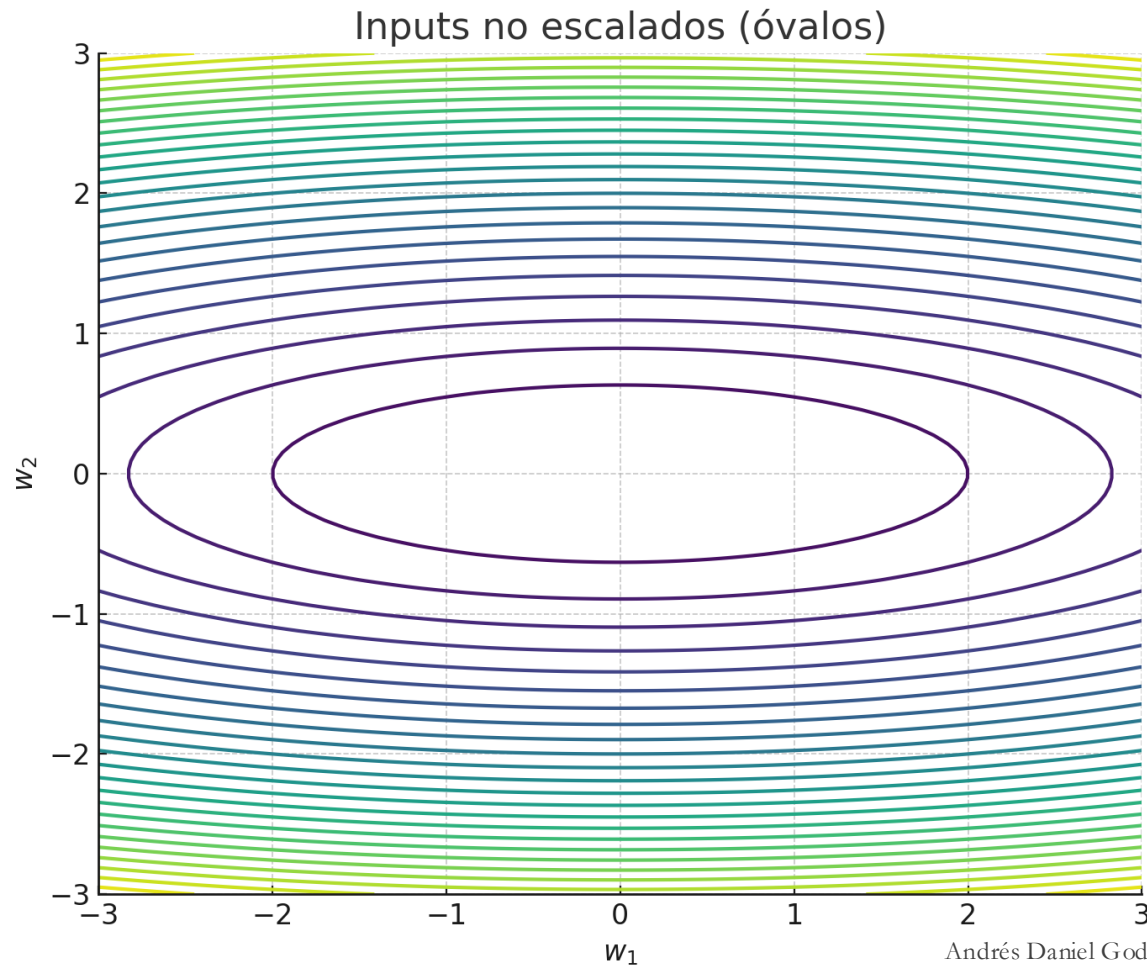
La convergencia se hace más lenta

Supón que uno de tus inputs es muy grande (por ejemplo, en miles) y otro es muy pequeño (por ejemplo, en décimas).

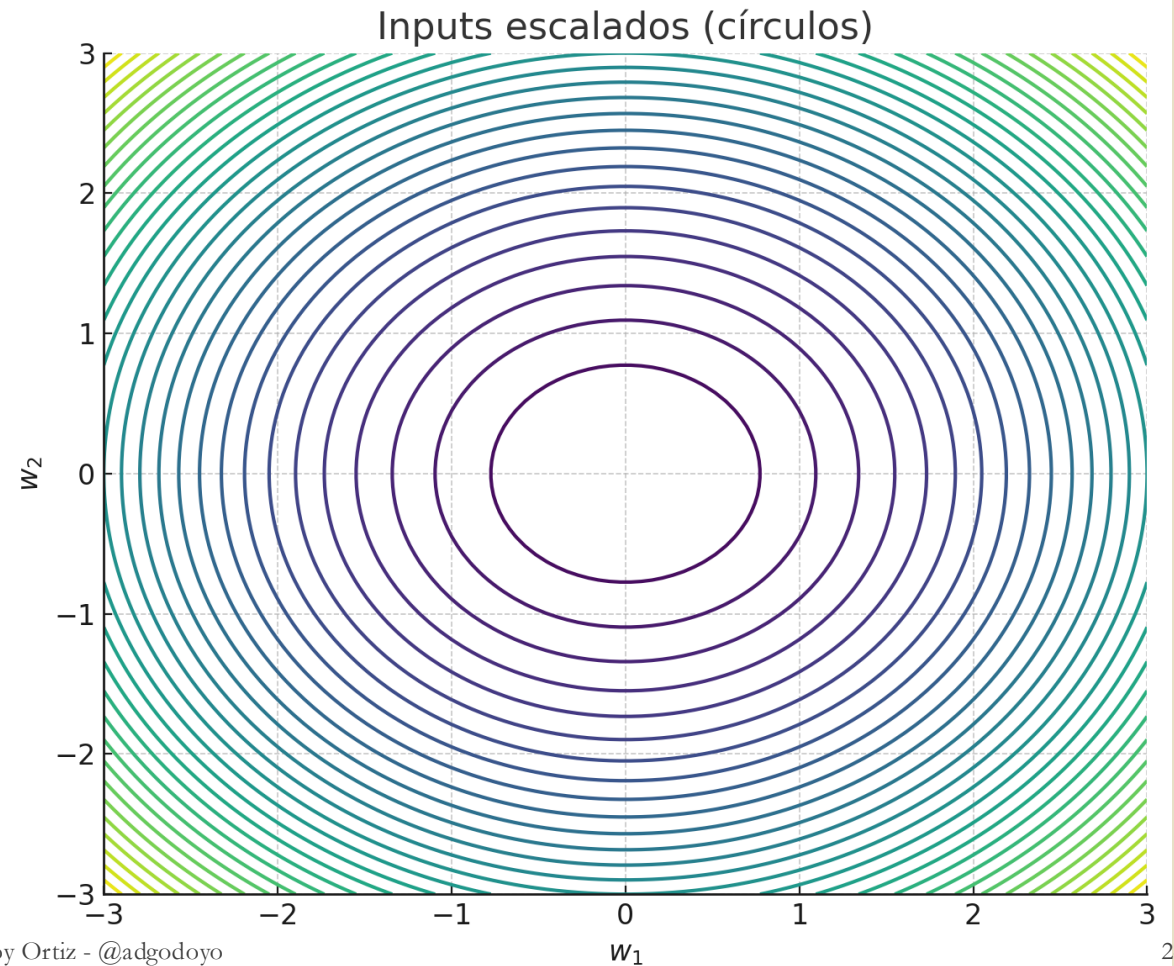
Entonces, para que la red funcione, los pesos asociados también van a crecer o encogerse para compensar:

- El peso asociado al input grande tendrá que ser pequeño para no disparar la activación.
- El peso del input pequeño tendrá que ser grande para que no desaparezca su efecto.

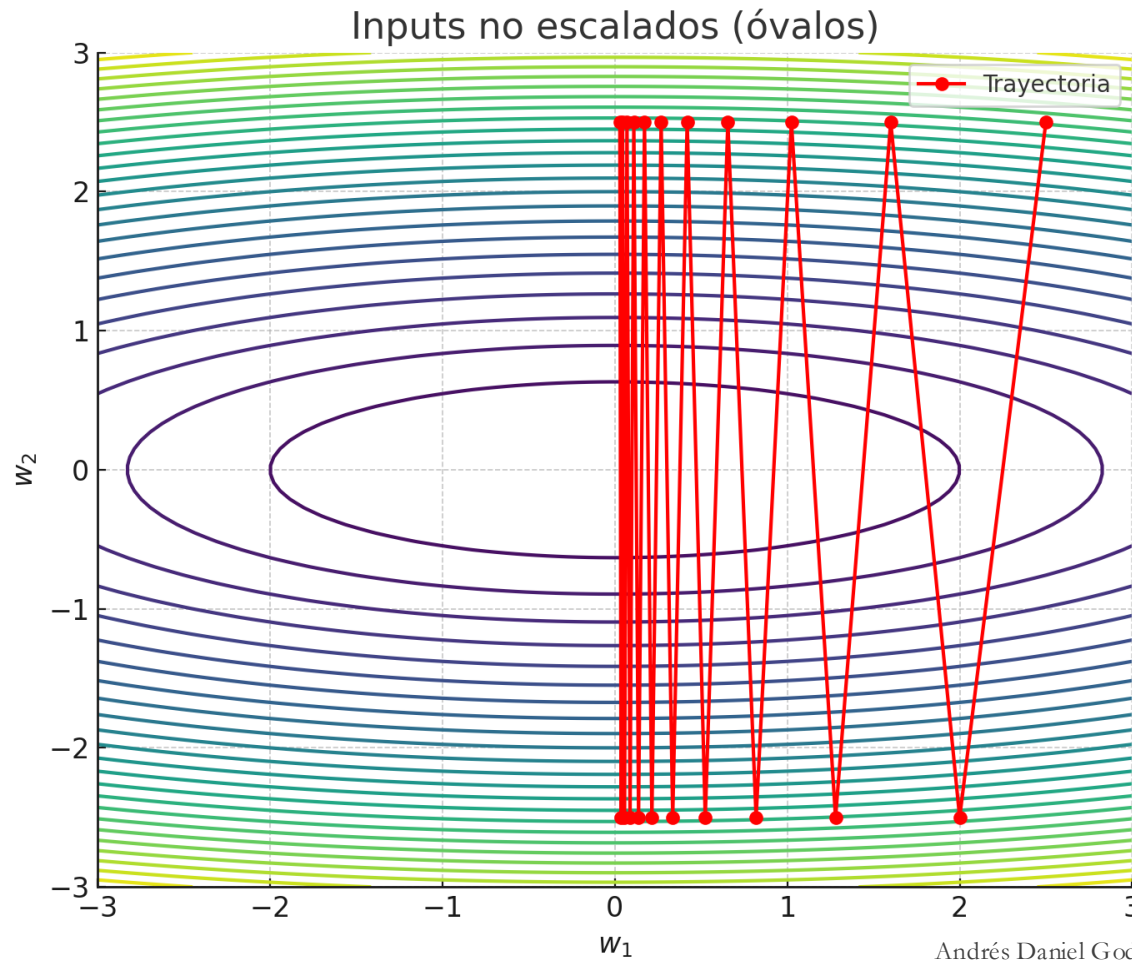
Loss Landscape



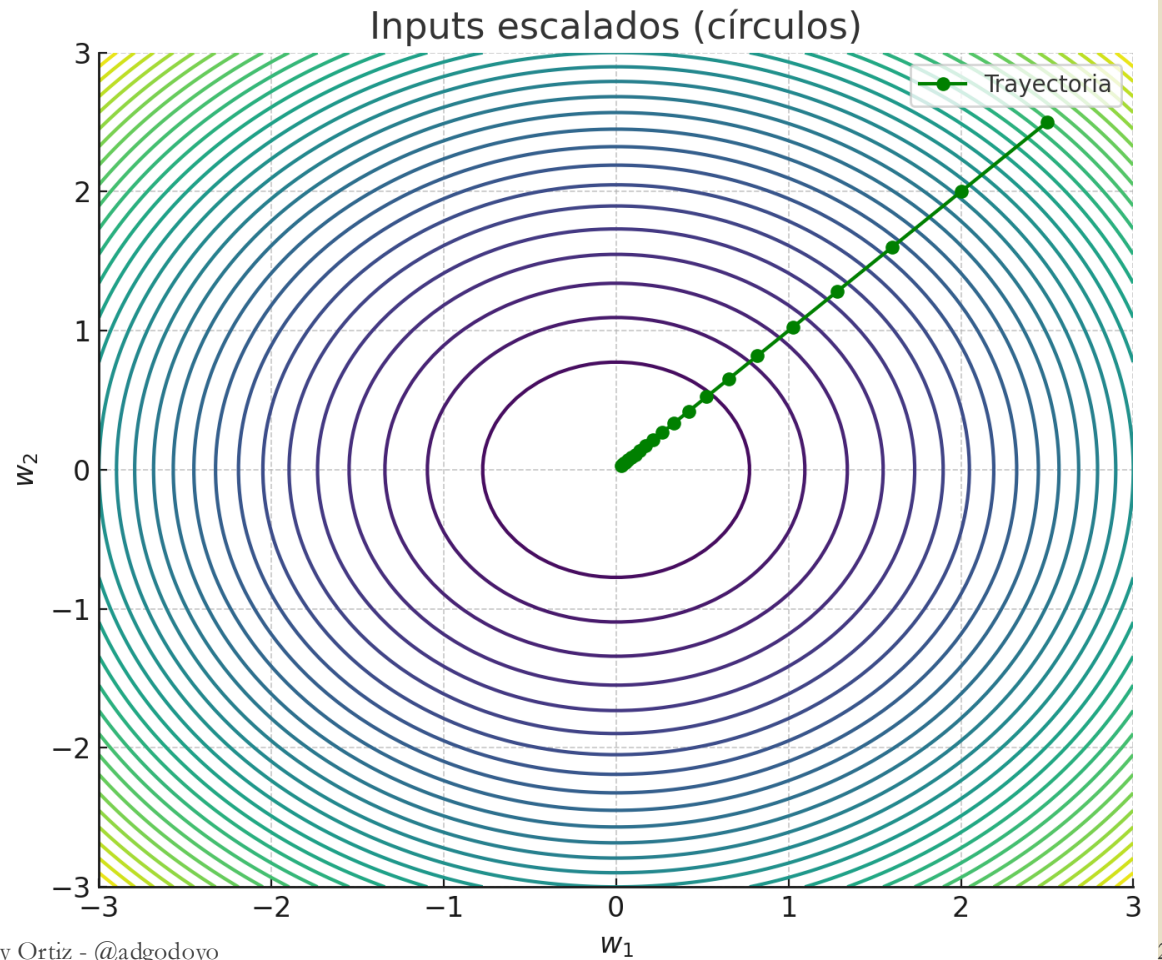
Andrés Daniel Godoy Ortiz - @adgodoyo



Trayectoria gradiente (misma lr y steps)



Andrés Daniel Godoy Ortiz - @adgodoyo



Estandarización (Z-score normalization)

$$x' = \frac{x - \mu}{\sigma}$$

Centra los datos en cero y los escala para que tengan varianza unitaria.

Cuándo usarla:



- ✓ Cuando usas funciones de activación como **tanh** o **sigmoid**.
- ✓ Cuando no conoces los límites superiores/inferiores de los datos.
- ✓ Cuando los datos tienen outliers leves (aunque no extremos).

Normalización Min-Max

$$x' = \frac{x - x_{\min}}{x_{\max} - x_{\min}}$$

Transforma los valores al rango [0,1]

Cuándo usarla:


-  Cuando conoces los límites de tus datos (como imágenes RGB).
-  Cuando usas activaciones como **ReLU**, que no saturan.

Dividir por una constante

$$x' = \frac{x}{255}$$

Simplemente escala el rango $[0, 255] \rightarrow [0, 1]$.

Cuándo usarla:

-  Imágenes en escala de grises o RGB, 8 bits por canal.

El escalado es tan importante que para redes profundas surgen otras opciones..

Batch Normalization

Normaliza el output $z[l]$ de cada capa antes de aplicar la activación. La media y la varianza se extraen del batch.

$$z^{[l]} = W^{[l]}a^{[l-1]} + b^{[l]} \xrightarrow{\text{BatchNorm}} \tilde{z}^{[l]} \xrightarrow{\sigma} a^{[l]}$$

1. Computar la media y varianza del batch:

$$\mu_B = \frac{1}{m} \sum_{i=1}^m z^{[l](i)}, \quad \sigma_B^2 = \frac{1}{m} \sum_{i=1}^m (z^{[l](i)} - \mu_B)^2$$

donde m es el tamaño del mini-batch.

2. Normalizar:

$$\hat{z}^{[l](i)} = \frac{z^{[l](i)} - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

donde ϵ es un valor pequeño (como 10^{-5}) para evitar división por cero.

Batch Normalization

Normaliza el output $z[l]$ de cada capa antes de aplicar la activación. La media y la varianza se extraen del batch.

3. Escalar y desplazar (aprendidos por la red):

$$\tilde{z}^{[l](i)} = \gamma^{[l]} \cdot \hat{z}^{[l](i)} + \beta^{[l]}$$

- γ : controla la **varianza aprendida** del output.
- β : controla el **valor medio aprendido** del output.

Estas son **variables aprendibles** como cualquier otro peso de la red.

4. Aplicar función de activación:

$$a^{[l]} = \sigma(\tilde{z}^{[l]})$$

¿Por qué el paso 3?

Después de que normalizas tus activaciones, **la red pierde la libertad de decidir cuál debería ser la escala o la media más adecuada** para una capa.

Entonces, BatchNorm **reintroduce esa flexibilidad** mediante dos nuevos parámetros que la red **aprende**.

¿Qué pasa durante inferencia (modo evaluación-testeo) con las media y varianza?

- Durante el entrenamiento, usa la media y varianza del batch actual.
- Durante inferencia, usa la media y varianza acumuladas durante entrenamiento (moving averages).

¿Qué hace que BatchNorm sea útil?

1. **Permite usar learning rates mayores**
2. **Acelera la convergencia**
3. **Permite inicializaciones menos cuidadosas**
4. **Reduce overfitting (ligeramente) incluso sin dropout**
5. **Actúa como regularizador al añadir ruido estocástico (por el cálculo batch-wise)**

Contexto	¿BatchNorm útil?	Alternativas o consideraciones
Redes profundas (CNN, MLP)	Re sí	Estándar de facto
Redes muy pequeñas	Neh	La varianza por batch es ruido
Modelos con batch size muy bajo	Mejor otras alternativas	Usar LayerNorm o GroupNorm
Transformers / NLP	Nunca	Usan LayerNorm

Buenas prácticas claves



1. Nunca escales train y test por separado.

Siempre: `scaler.fit(X_train)` y luego `scaler.transform(X_test)`. Si no, introduces *data leakage*.

2. Haz el escalado antes de dividir los datos en batches o entrenar.

3. En pipelines reales, guarda el objeto scaler para usarlo en producción o inferencia.

4. Para datos por canal (e.g., imágenes RGB), escalar por canal, no globalmente. Es decir, Canal R se escalar con su propia media y varianza, y así para G y B.



3. FUNCIONES DE COSTO

REGRESIÓN

Error Cuadrático Medio (MSE)

$$\mathcal{L}_{\text{MSE}} = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$$

Penaliza fuertemente errores grandes.

Derivada simple

Sensible a outliers

REGRESIÓN

Error Absoluto Medio (MAE)

$$\mathcal{L}_{\text{MAE}} = \frac{1}{n} \sum_{i=1}^n |y_i - \hat{y}_i|$$

Penaliza todos los errores por igual.

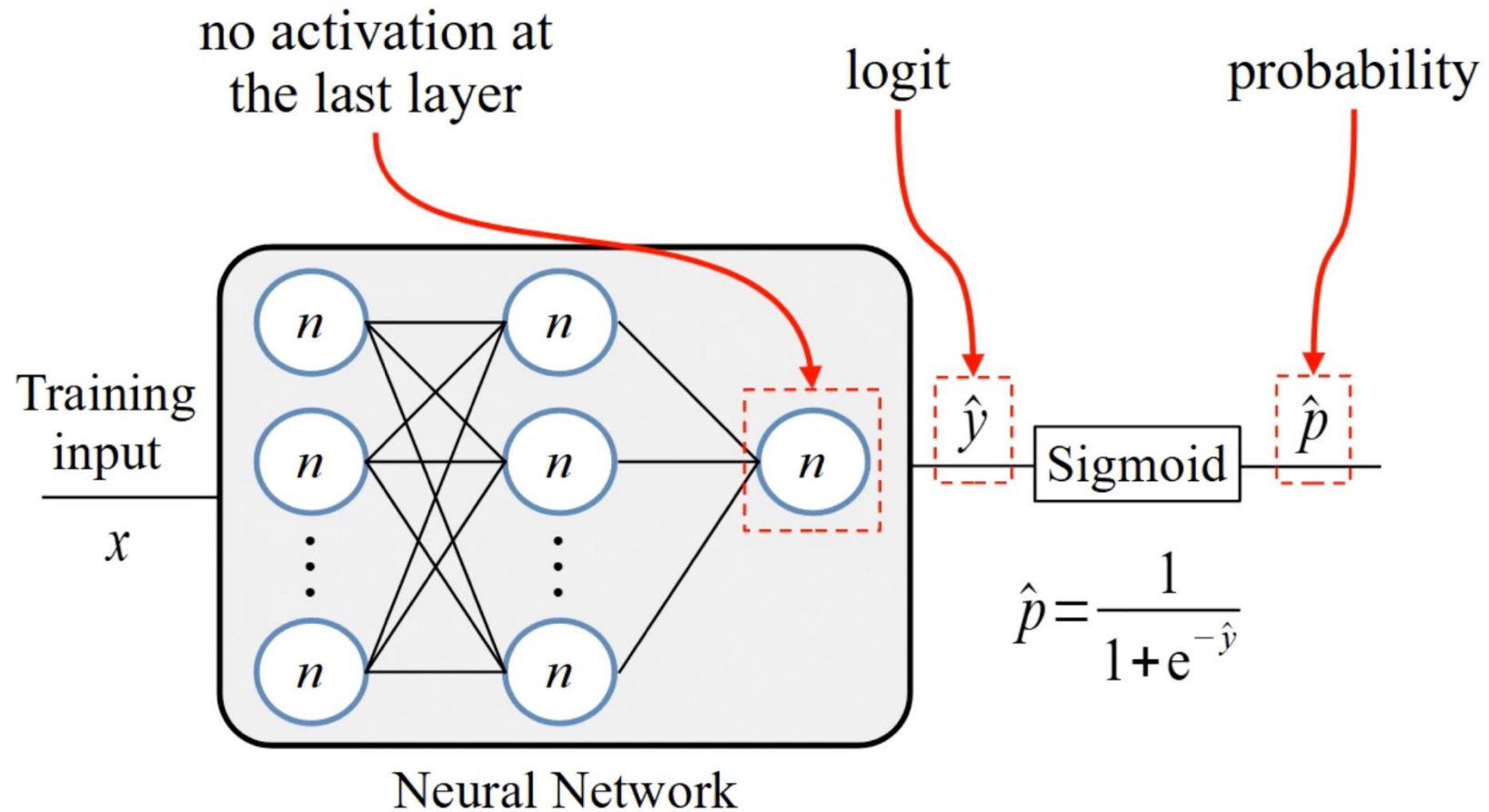
Más robusta a outliers, pero **menos suave** para optimización

REGRESIÓN

Huber Loss

$$\mathcal{L}_{\delta} = \begin{cases} \frac{1}{2}(y - \hat{y})^2 & \text{si } |y - \hat{y}| \leq \delta \\ \delta \cdot (|y - \hat{y}| - \frac{1}{2}\delta) & \text{si no} \end{cases}$$

Combina lo mejor de MSE (en errores pequeños) y MAE (en errores grandes).
Muy usada en práctica cuando hay outliers.



Clasificación Binaria

BINARY CLASSIFICATION

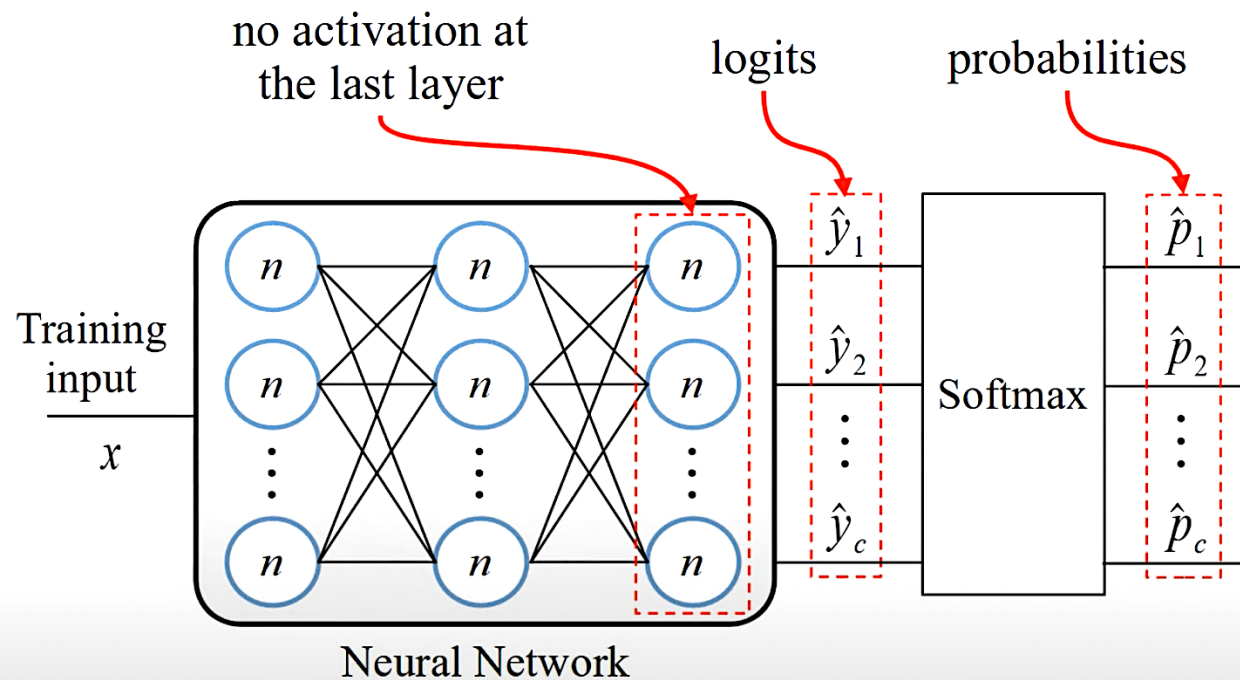
Binary Cross-Entropy (BCE)

$$\mathcal{L}_{\text{BCE}} = - [y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})]$$

- y es *one-hot*, \hat{y} es una distribución de probabilidad (sigmoid)

En PyTorch: `nn.BCELoss()` (requiere que ya hayas hecho sigmoid), por eso lo más seguro es:

`nn.BCEWithLogitsLoss()` # incluye sigmoid + BCE



Softmax equation

$$\hat{p}_i = \frac{e^{\hat{y}_i}}{\sum_{j=1}^c e^{\hat{y}_j}}$$

$$\sum_{i=1}^c \hat{p}_i = 1$$

Clasificación multiclase

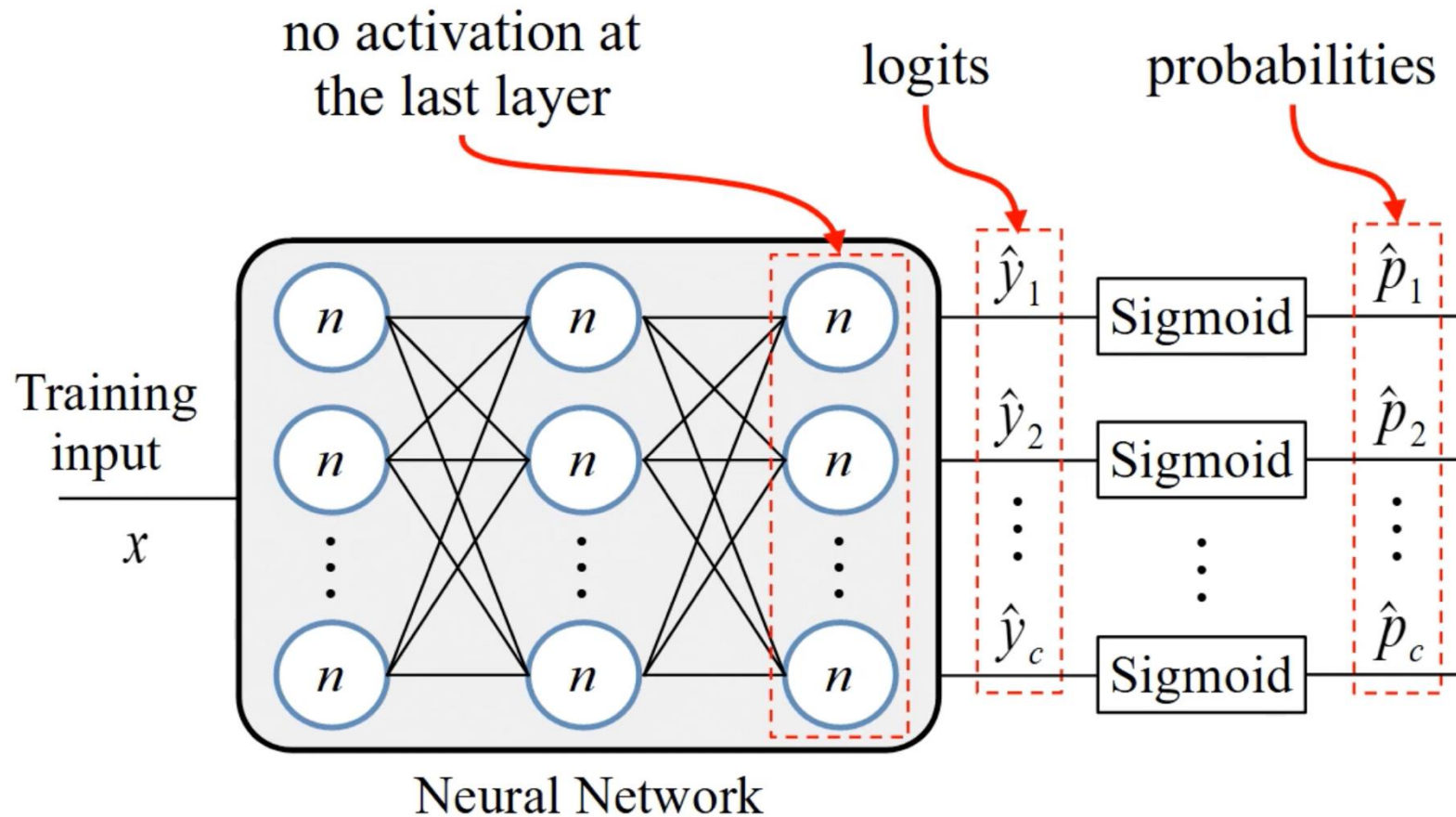
MULTICLASS CLASSIFICATION

Categorical Cross-Entropy (CCE)

$$\mathcal{L}_{\text{CCE}} = - \sum_{i=1}^K y_i \log(\hat{y}_i)$$

- y es *one-hot*, \hat{y} es una distribución de probabilidad (softmax)
- Penaliza con fuerza si la clase correcta tiene baja probabilidad

`nn.CrossEntropyLoss()` ##incluye softmax



Clasificación
multi-etiqueta
(multilabel)

MULTICLASS CLASSIFICATION

Categorical Cross-Entropy (CCE)

$$\mathcal{L} = - \sum_{i=1}^K [y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)]$$

- Usa **sigmoid por neurona** de salida.
- Cada neurona predice si su clase está o no presente.

Tipo de tarea	Etiquetas y	Neuronas de salida	Activación final	Función de pérdida (Pytorch)
Regresión univariada	Escalar (real)	1	Ninguna	MSELoss, MAELoss, HuberLoss
Regresión multivariada	Vector continuo de dim d	d	Ninguna	MSELoss
Clasificación binaria	0 o 1	1	sigmoid	BCELoss / BCEWithLogitsLoss
Clasificación multiclase	Entero en $\{0, \dots, K-1\}$	K	softmax (implícito)	CrossEntropyLoss
Clasificación multilabel	Vector binario de dim K	K	sigmoid (por salida)	BCEWithLogitsLoss (por neurona)



4. REGULARIZACIÓN

¿Qué es regularización?

Es cualquier técnica que **modifica el entrenamiento para reducir la complejidad del modelo**, buscando que aprenda patrones generales y no ruido específico del conjunto de entrenamiento.



Señales de que necesitas regularización

- Mucha diferencia entre pérdida de entrenamiento y validación.
- Accuracy de entrenamiento muy alta pero test bajo.
- La red memoriza ruido o ejemplos atípicos.

L2 Regularization

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{original}} + \lambda \sum_i w_i^2$$

Penaliza pesos grandes → modelo más "suave"

Se llama "weight decay" porque hace que los pesos tiendan a encogerse con el tiempo.

Muy común en redes profundas, también en Transformers.

Se añade el término λw al **gradiente** de la pérdida:

$$g_t = \nabla_w \mathcal{L}(w_t) + \lambda w_t$$

Este gradiente **se pasa por el mecanismo adaptativo de Adam**:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \quad , \quad v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

Y luego el peso se actualiza así:

$$w_{t+1} = w_t - \eta \cdot \frac{m_t}{\sqrt{v_t} + \epsilon}$$

🔍 **Problema:** el término de L2 **pasa por la adaptación**, se mezcla con el gradiente de datos y **no actúa como un verdadero decay puro**.

Ejemplo: L2 Regularización + Adam

Weight decay

Modifica directamente la actualización de pesos

$$w \leftarrow w - \eta \cdot \left(\frac{\partial \mathcal{L}_{\text{data}}}{\partial w} + \lambda w \right)$$

Primero se computa el **gradiente puro** de la pérdida:

$$g_t = \nabla_w \mathcal{L}(w_t)$$

Este se adapta como en Adam:

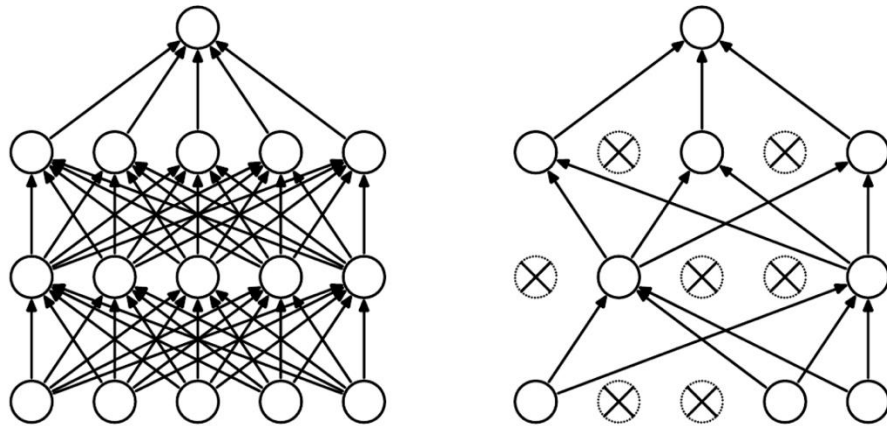
m_t, v_t como antes

Y **luego**, en el paso de actualización, se aplica el weight decay **fuera del mecanismo adaptativo**:

$$w_{t+1} = w_t - \eta \cdot \left(\frac{m_t}{\sqrt{v_t} + \epsilon} + \lambda w_t \right)$$

Ejemplo:
Weight decay
+ Adam

Dropout



Durante el entrenamiento, **se ponen a cero las neuronas** con una probabilidad μ (típicamente $\mu=0.5$).

Esto se hace de forma **independiente** para cada batch.

Se crea un **ensamble “en tiempo real”** a partir de una sola red con parámetros compartidos.

¿Por qué es una buena idea?

1. **Obliga a la red a aprender una representación redundante \Rightarrow regularización**
2. **Reduce la capacidad efectiva del modelo \Rightarrow modelos más grandes, entrenamiento más largo**
3. **Previene la co-adaptación de características (las unidades no pueden aprender a "deshacer" la salida de otras)**

En pytorch

- `import torch.nn as nn`
- `drop = nn.Dropout(p=0.5)`
- `drop.train()` # activa dropout
- `drop.eval()` # desactiva dropout (no escalará en test)

Arquitectura	Técnicas de regularización más usadas	Observaciones clave
MLP (redes densas)	L2 (Weight Decay), Dropout, Early Stopping	Dropout es muy efectivo, especialmente con capas grandes
CNN (convolutional)	L2, Data Augmentation, BatchNorm, (poco Dropout)	Dropout es menos común en capas convolucionales; Data Augmentation es clave
RNN / LSTM / GRU	L2, Dropout (pero solo en conexiones no recurrentes), LayerNorm	Dropout debe aplicarse con cuidado → nn.Dropout vs nn.Dropout2d
Transformers (BERT, GPT)	Weight Decay (AdamW), Dropout, LayerNorm	No usan BatchNorm. Dropout está en múltiples puntos: atención, MLP, embeddings
Autoencoders / GANs	L2, Dropout (en codificadores), BatchNorm (en generadores)	En GANs: BatchNorm ayuda mucho a estabilizar el entrenamiento