Aprendizaje profundo

Redes neuronales y optimización

Resumen : Perceptrón -> MLP -> Retropropagación -> Regularización -> Optimizadores -> RNN/LSTM -> Atención

Objetivo

Entender qué cambia al pasar de modelos lineales a redes profundas, cómo entrenarlas de forma estable y por qué la atención es clave en secuencias largas.

Temario

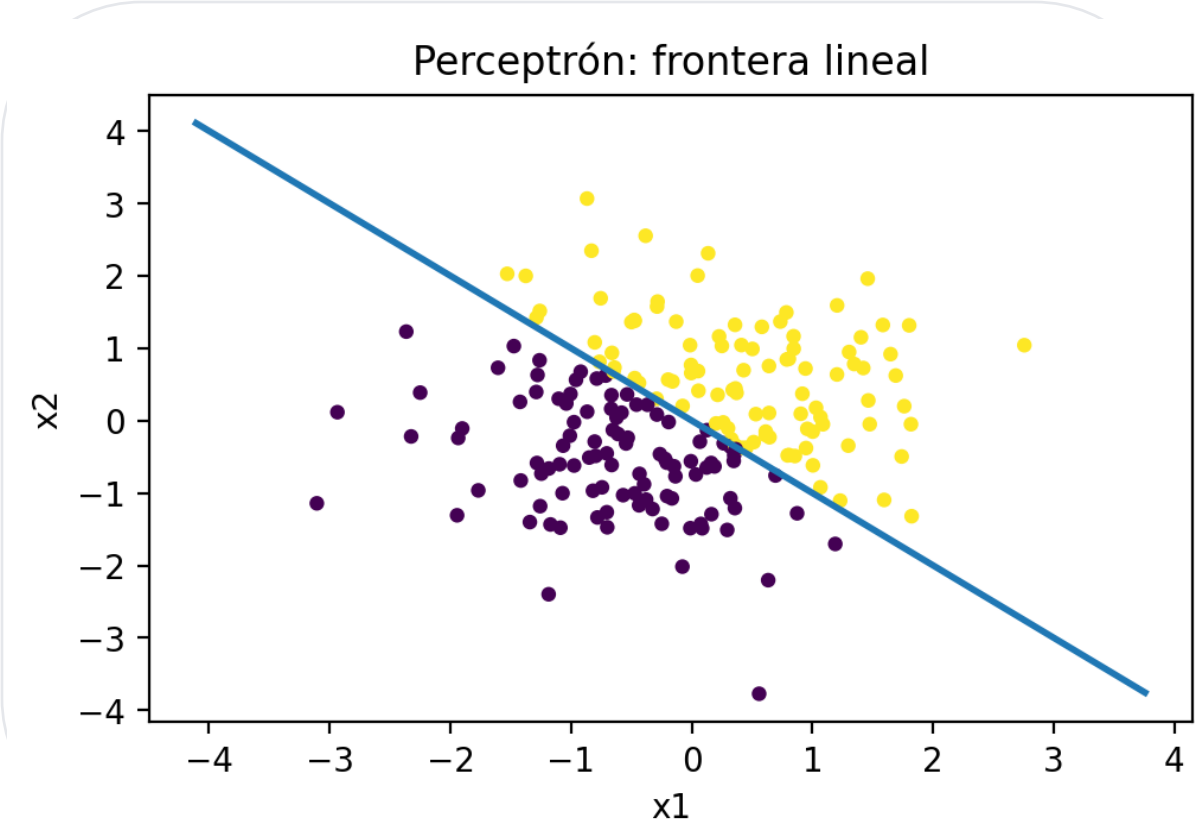
- Perceptrón, MLP, funciones de activación y función de pérdida
- Entrenamiento y retropropagación: papel de los gradientes
- Regularización y generalización: L2, dropout, batch normalization, early stopping
- Optimizadores modernos: SGD+momentum, Adam/AdamW, programación de tasa de aprendizaje
- Limitaciones de RNN/LSTM en secuencias largas y motivación para la atención

Idea central

La práctica consiste en "ablación": cambiar una sola cosa (activación / regularización / optimizador) y observar estabilidad y generalización.

1. Perceptrón

Modelo lineal con activación: frontera de decisión



Definición:

$y = \text{sign}(w \cdot x + b)$ (o $\sigma(w \cdot x + b)$ en variante probabilística)

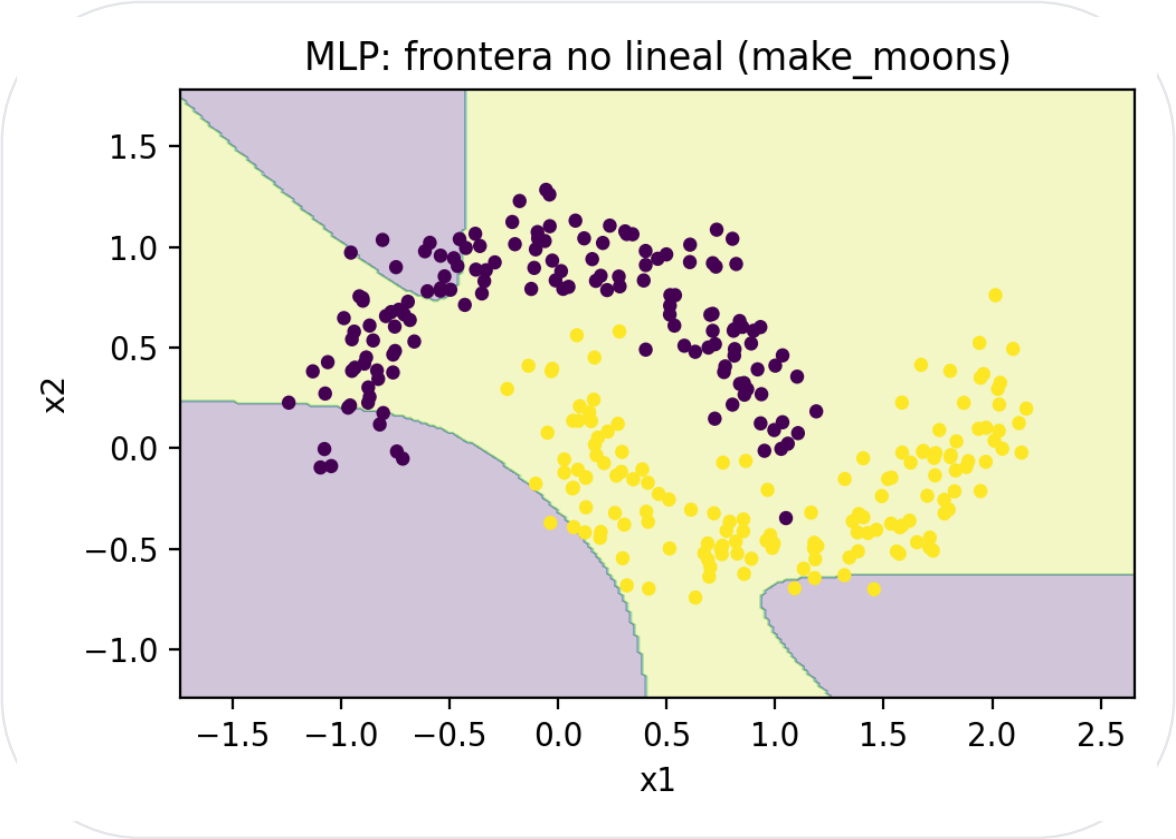
- Aprende una separación lineal (hiperplano).
- Base para entender capas lineales y el papel del sesgo (b).
- Limitación: no resuelve fronteras no lineales (por ejemplo, XOR / "moons").

Conexión con el cuaderno

Incluye un perceptrón "a mano" (NumPy) para ver el update paso a paso.

1.2 MLP (Multilayer Perceptron)

No linealidad = composición de capas + activaciones



Arquitectura típica:

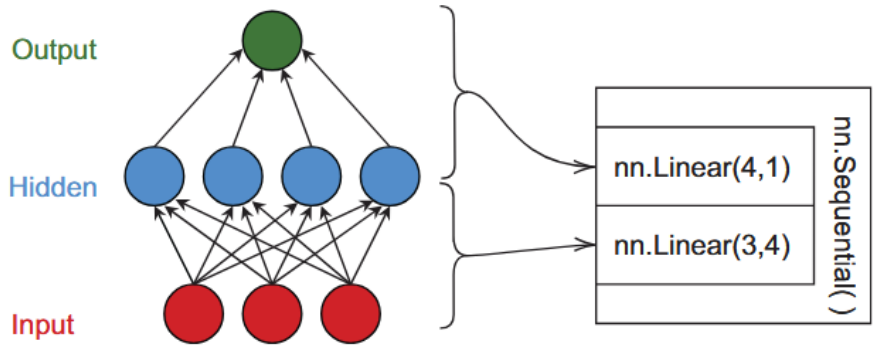
$h_1 = \phi(W_1x + b_1) \rightarrow h_2 = \phi(W_2h_1 + b_2) \rightarrow \hat{y}$

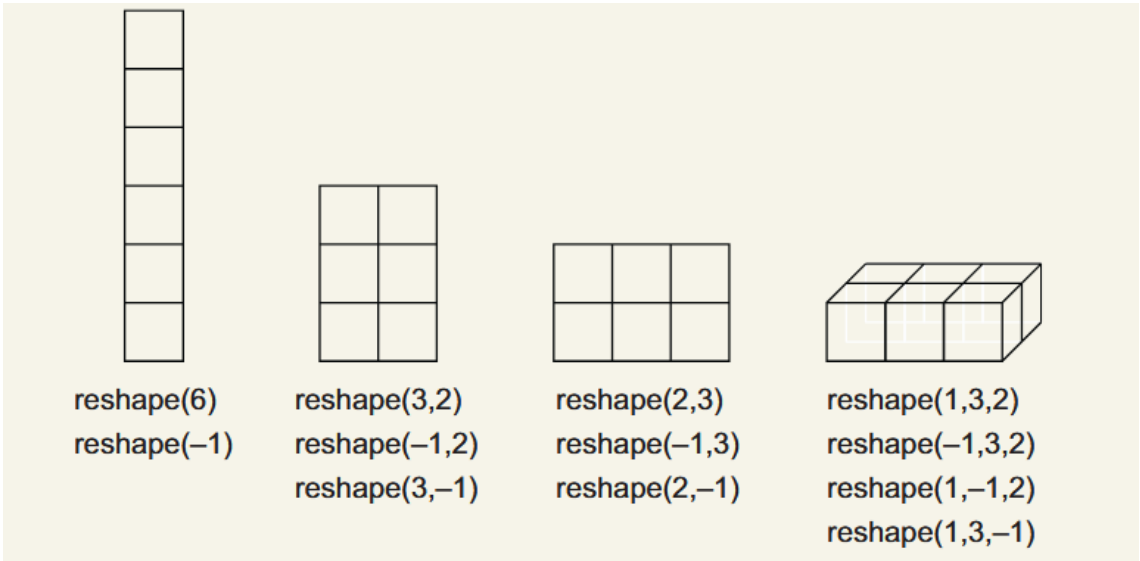
- La activación ϕ rompe la linealidad (sin ϕ, todo colapsa a una sola capa lineal).
- Capacidad ↑ con profundidad/ancho; riesgo de sobreajuste ↑.
- Entrenamiento con descenso por gradiente sobre una pérdida (loss).

En el cuaderno
MLP en PyTorch sobre make_moons, con métricas train/val.

1.3 MLP: arquitectura y shapes (PyTorch)

De tensores a nn.Sequential: entradas, batches y reshape

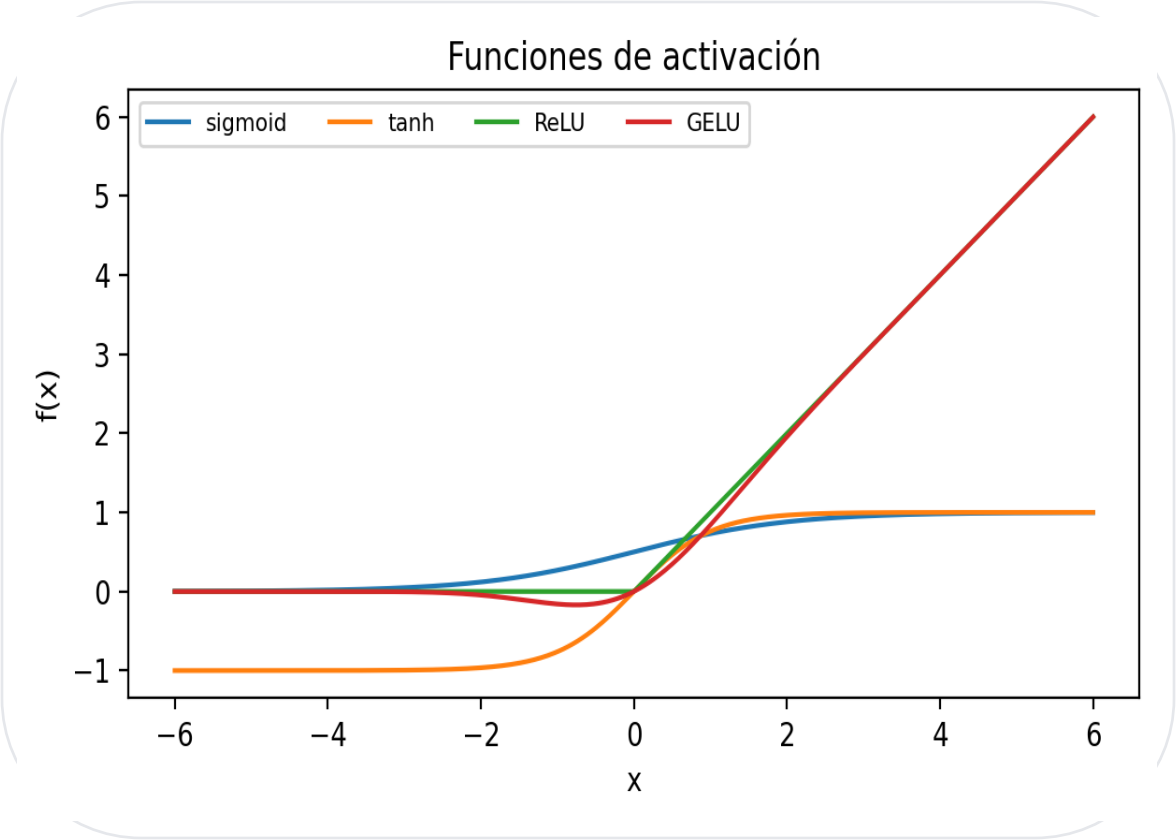


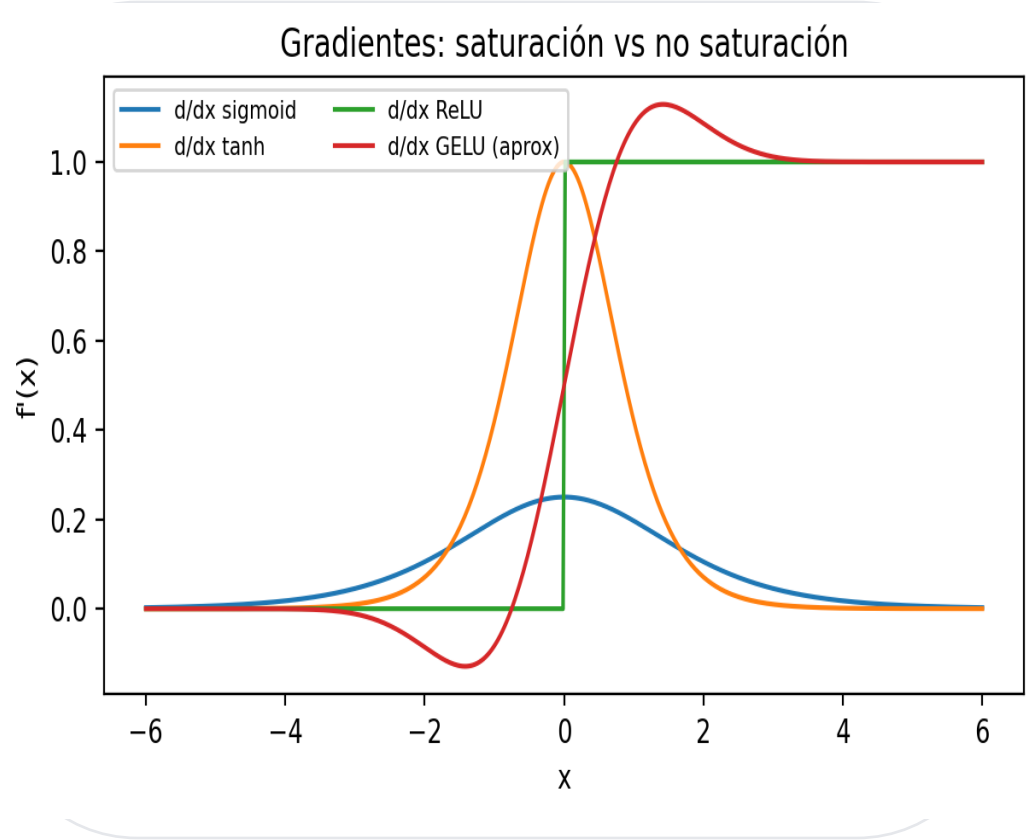


- Convención típica: x ∈ R^{batch×d} (o [batch, channels, H, W] en visión)
- nn.Linear(in_features, out_features) espera el último eje como 'features'
- Usa reshape/view para aplanar: [batch, C, H, W] -> [batch, C·H·W]
- Revisa siempre dimensiones con x.shape antes/después de cada bloque
- En clasificación: última capa -> #clases; en regresión: última capa -> 1

2. Activaciones

Forma de la no linealidad y sus efectos



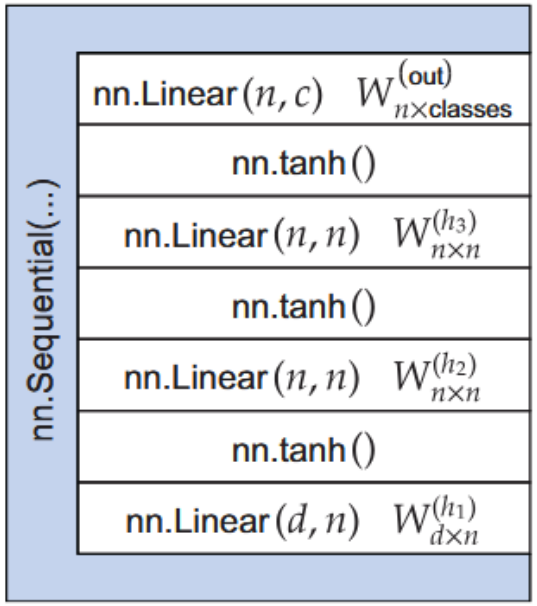


Punto clave

Sigmoid/tanh pueden saturar (gradiente pequeño). ReLU/GELU suelen facilitar la propagación del gradiente en redes profundas.

2.1 Activaciones en MLP profundo (composición)

Activación repetida (tanh) y su impacto en el flujo de gradiente



$$\hat{y} = \tanh\left(h_3^\top W^{(\text{out})}_{n\times\text{classes}}\right)$$
$$h_3 = \tanh\left(h_2^\top W^{(h_3)}_{n\times n}\right)$$
$$h_2 = \tanh\left(h_1^\top W^{(h_2)}_{n\times n}\right)$$
$$h_1 = \tanh\left(x^\top W^{(h_1)}_{d\times n}\right)$$

$$f(x) = \tanh\left(\tanh\left(\tanh\left(x^\top W^{(h_1)}_{d\times n}\right) W^{(h_2)}_{n\times n}\right) W^{(h_3)}_{n\times n}\right) W^{(\text{out})}_{n\times\text{classes}}$$

- Un MLP es una composición de capas lineales + no linealidades.
- No linealidades saturantes (p.ej., tanh/sigmoid) pueden hacer que los gradientes se atenúen.
- ReLU/GeLU suelen entrenar mejor en redes profundas; BatchNorm/LayerNorm estabilizan.

Función de pérdida (loss)

Qué optimizamos y por qué

Ejemplos comunes:

- Clasificación multiclase: Cross-Entropy
- Clasificación binaria: BCE / BCEWithLogits
- Regresión: MSE / MAE

Intuición

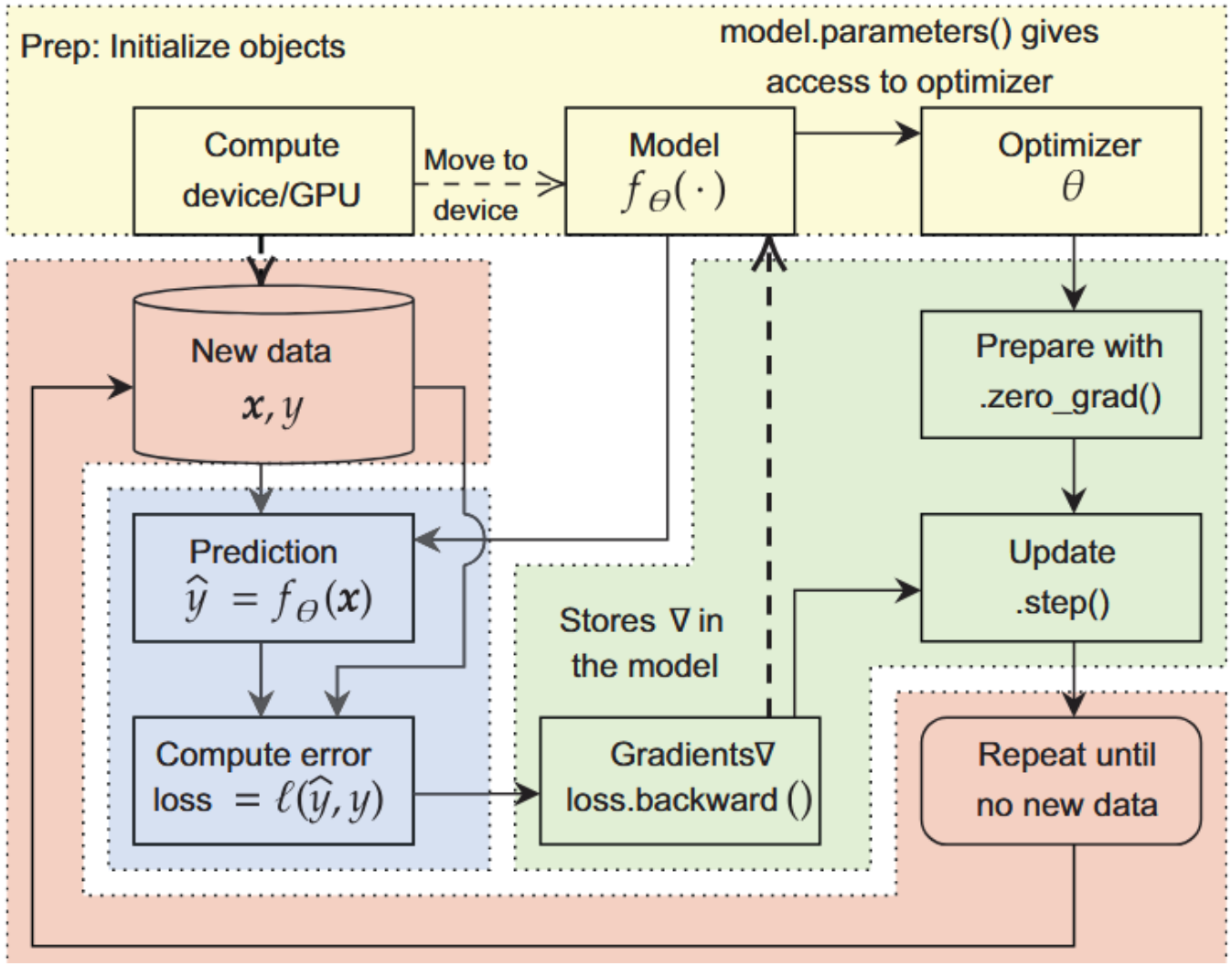
La pérdida define qué significa "equivocarse" y cómo se penalizan los errores. En redes, el gradiente de la pérdida guía la actualización de los pesos.

Loop de entrenamiento (conceptual)

1) $\hat{y}$ = model(x)
2) L = loss($\hat{y}$, y)
3) optimizer.zero_grad()
4) L.backward() -> gradientes
5) optimizer.step() -> update

3. Entrenamiento: ciclo en PyTorch

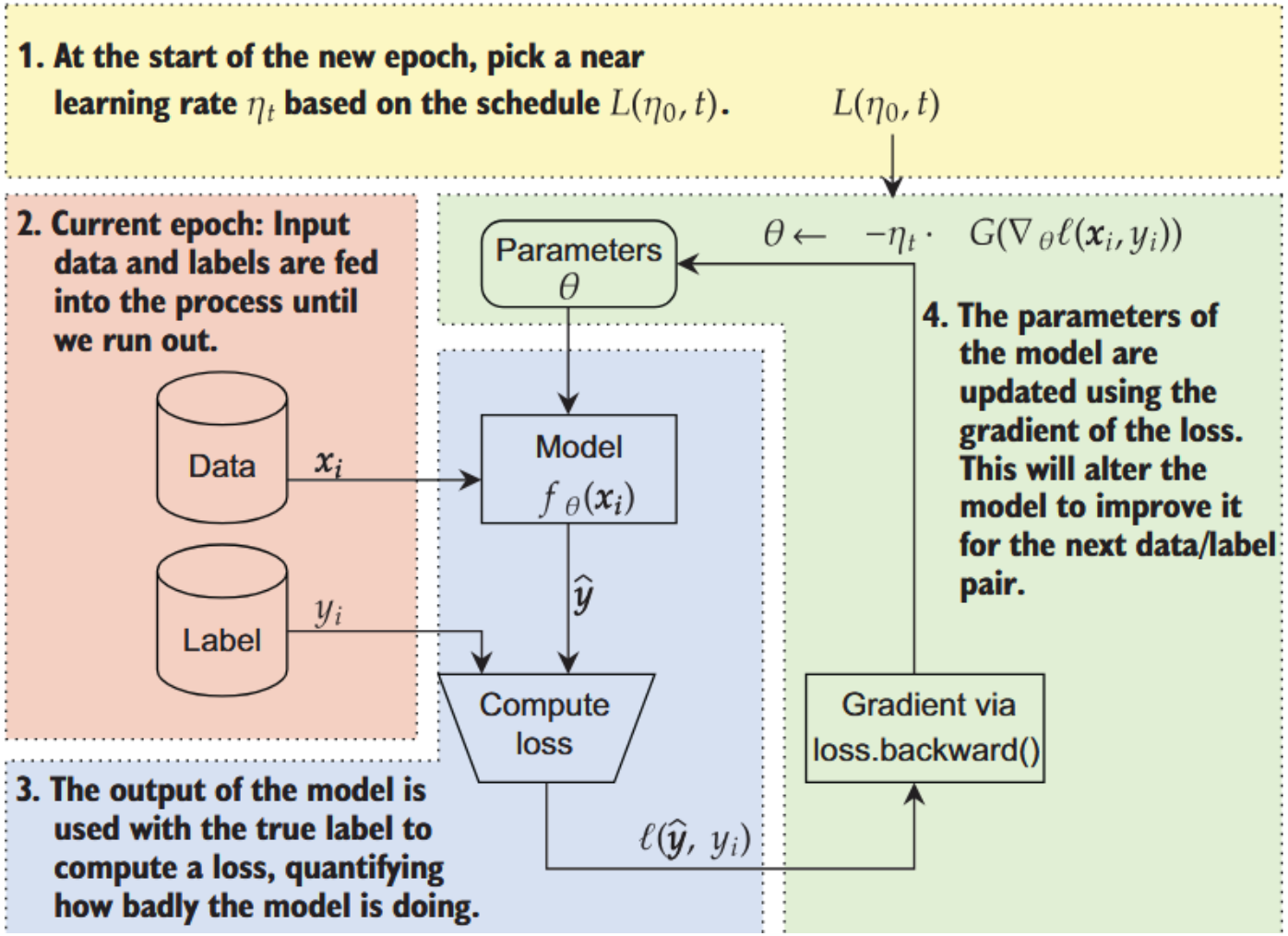
Forward -> loss -> backward -> step (y el rol de zero_grad)



- model.train() activa dropout/batchnorm en modo entrenamiento
- optimizer.zero_grad() limpia gradientes acumulados
- loss.backward() calcula $\nabla\theta\ell$ y los guarda en param.grad
- optimizer.step() actualiza parámetros (SGD/Adam/…)
- model.eval() desactiva dropout y fija BatchNorm para validación

3.1 Entrenamiento con schedule de LR

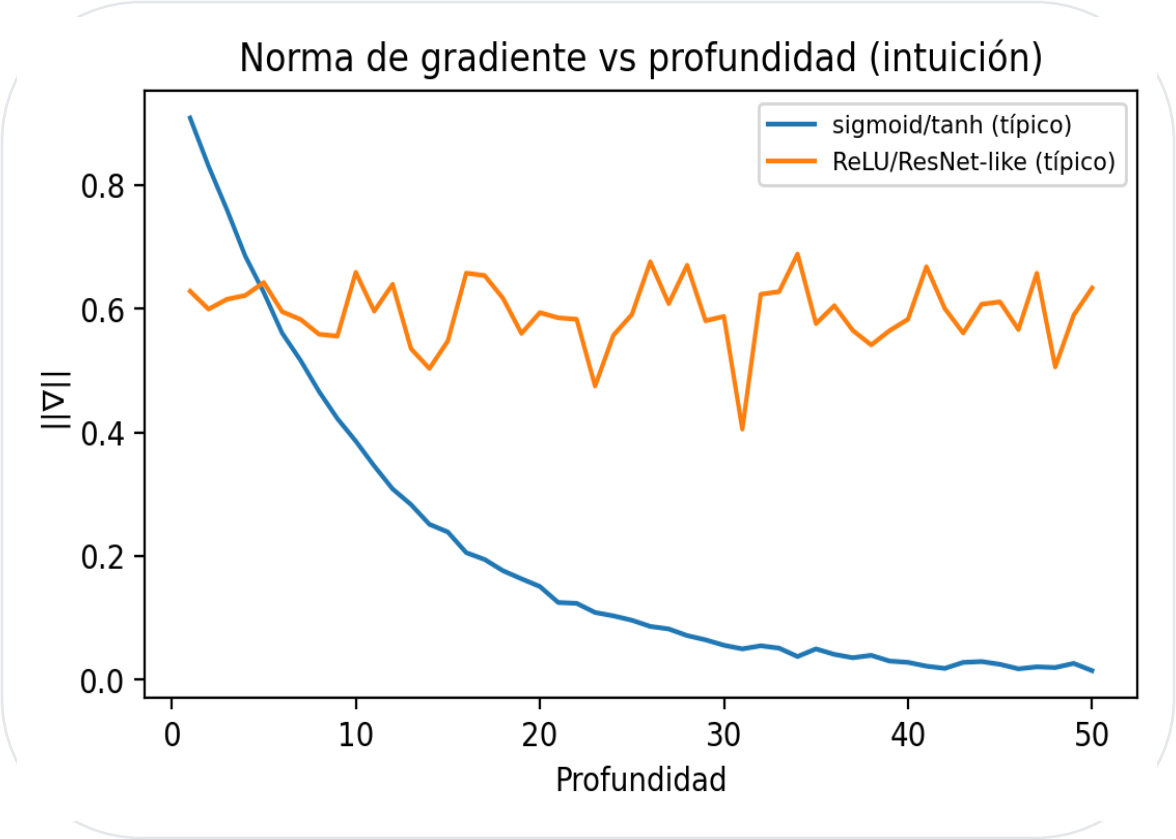
Scheduler dentro del loop (epoch ->batches)



Idea: η_t = L(η0, t) se elige al inicio de la época; luego se repite (forward -> loss -> backward -> step) sobre los mini-batches.

4. Gradientes y retropropagación

Por qué importan: estabilidad del entrenamiento



Qué mirar en práctica:

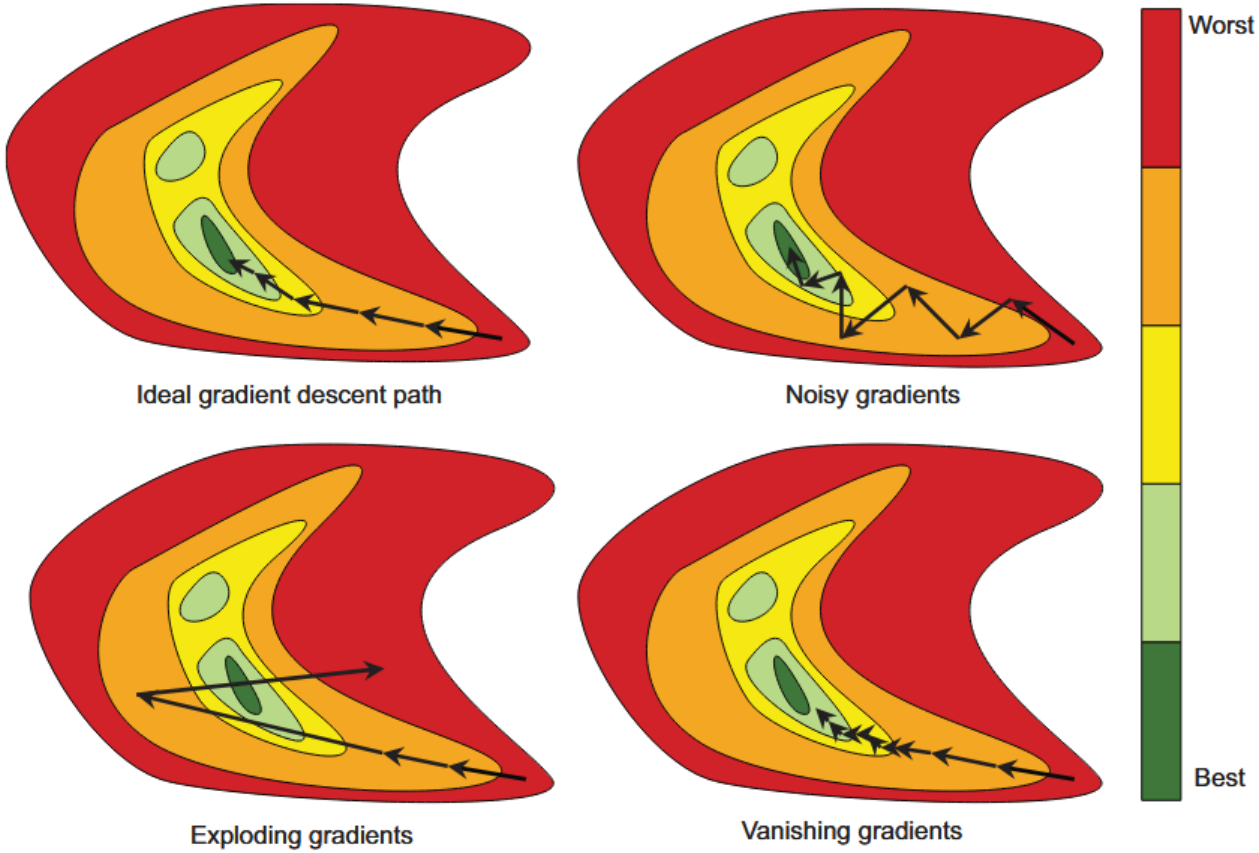
- Vanishing gradients: la señal se hace demasiado pequeña en capas tempranas.
- Exploding gradients: la señal crece y desestabiliza (picos / NaN).
- Activación, inicialización, normalización y LR interactúan fuertemente.

En el cuaderno

Se estudia la norma del gradiente y se conecta con saturación de activaciones.

3.2 Gradientes: ruido, desaparición y explosión

Qué se observa en entrenamiento profundo/secuencial



- **Desaparición/explosión: multiplicación repetida de Jacobianos (profundidad o tiempo).**
- **Mitigación: ReLU/GELU, inicialización, normalización, clipping, residual/skip connections.**

5. Verificación de la retropropagación

Diferencias finitas como "prueba de humo"

Objetivo

Validar que el gradiente computado por autograd coincide (aprox.) con un gradiente numérico.

Fórmula

$$\partial L/\partial \theta \approx (L(\theta+\varepsilon) - L(\theta-\varepsilon)) / (2\varepsilon)$$

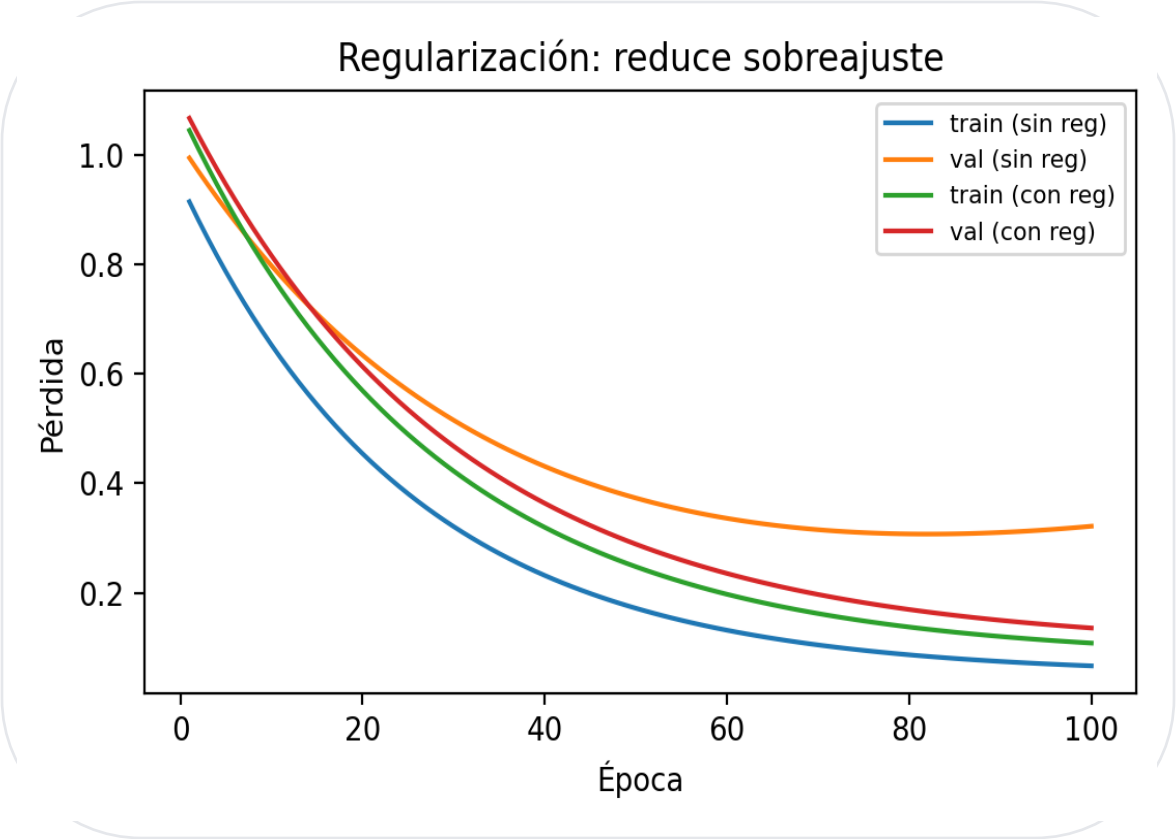
- Útil para depurar
- Sensible a ε y precisión
- Se aplica a pocos θ aleatorios

Buenas prácticas

- Desactivar dropout (modo eval)
- Usar float64 para el check
- Comparar error relativo
- Probar varios θ
- Fijar seed

6. Regularización y generalización

Controlar sobreajuste sin "matar" la señal



Técnicas importantes

- L2 / weight decay: penaliza pesos grandes.
- Dropout: regulariza por ensamble implícito.
- Batch Normalization: estabiliza escalas (y a veces regulariza).
- Early stopping: parar cuando validación deja de mejorar.

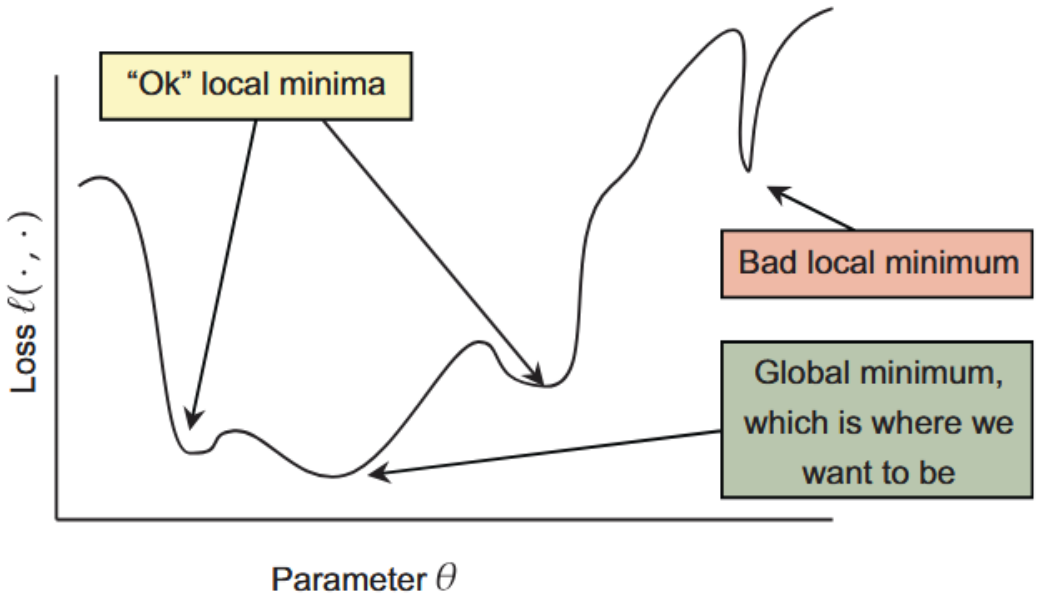
Métrica útil

Brecha train-val (generalization gap) + curva val-loss.

7. Intuición de optimización: gradiente y mínimos locales

Descenso por gradiente y mínimos locales (intuición)

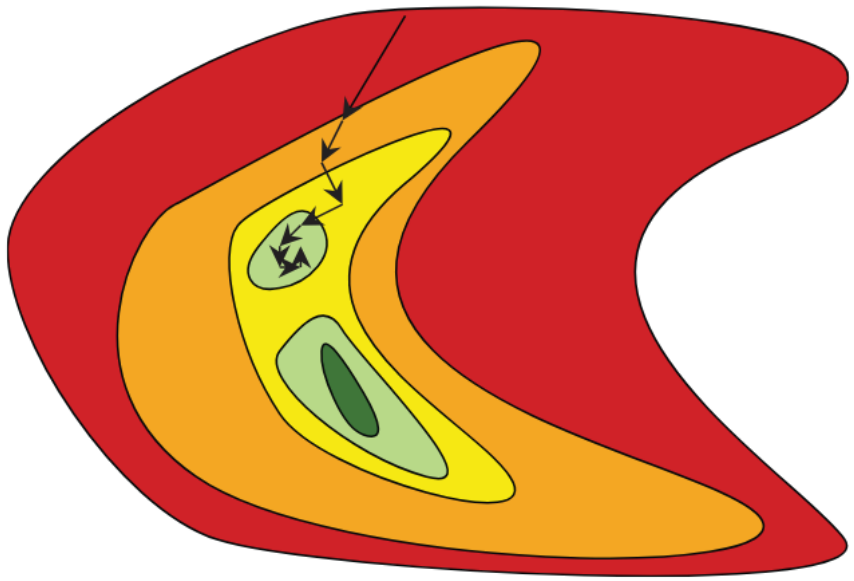
$$\Theta_{t+1} = \Theta_t - \eta \cdot \underbrace{\nabla_{\Theta_t} \ell(f_{\Theta_t}(\boldsymbol{x}), y)}_{\text{the gradient } g^t}$$



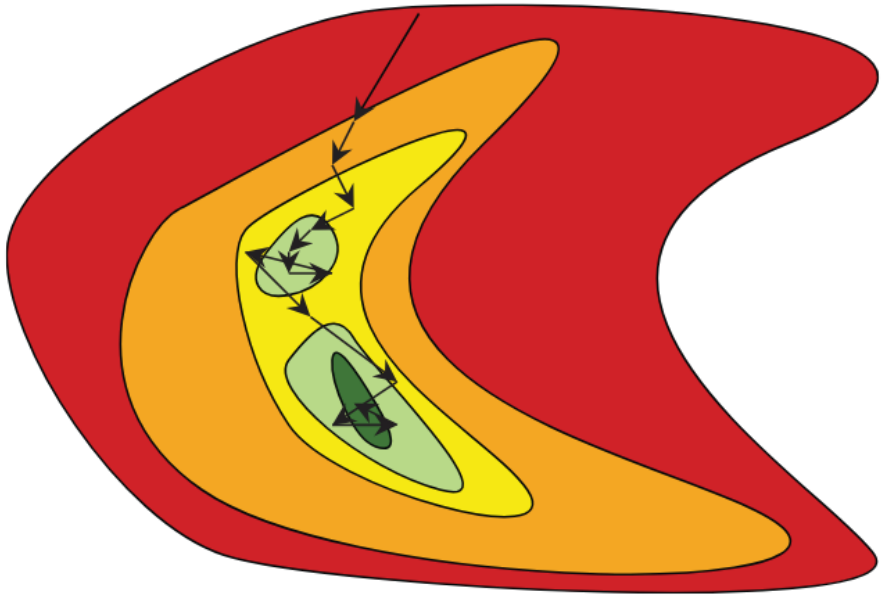
- En aprendizaje profundo, la superficie de pérdida suele ser no convexa.
- Momentum ayuda a atravesar valles y a suavizar ruido del gradiente.
- Adam/AdamW adaptan el paso por parámetro; AdamW separa weight decay (L2).
- Learning rate schedule: clave para convergencia + generalización.

7.1 SGD: ruido y exploración

Mini-batch introduce ruido útil para salir de "valles" estrechos



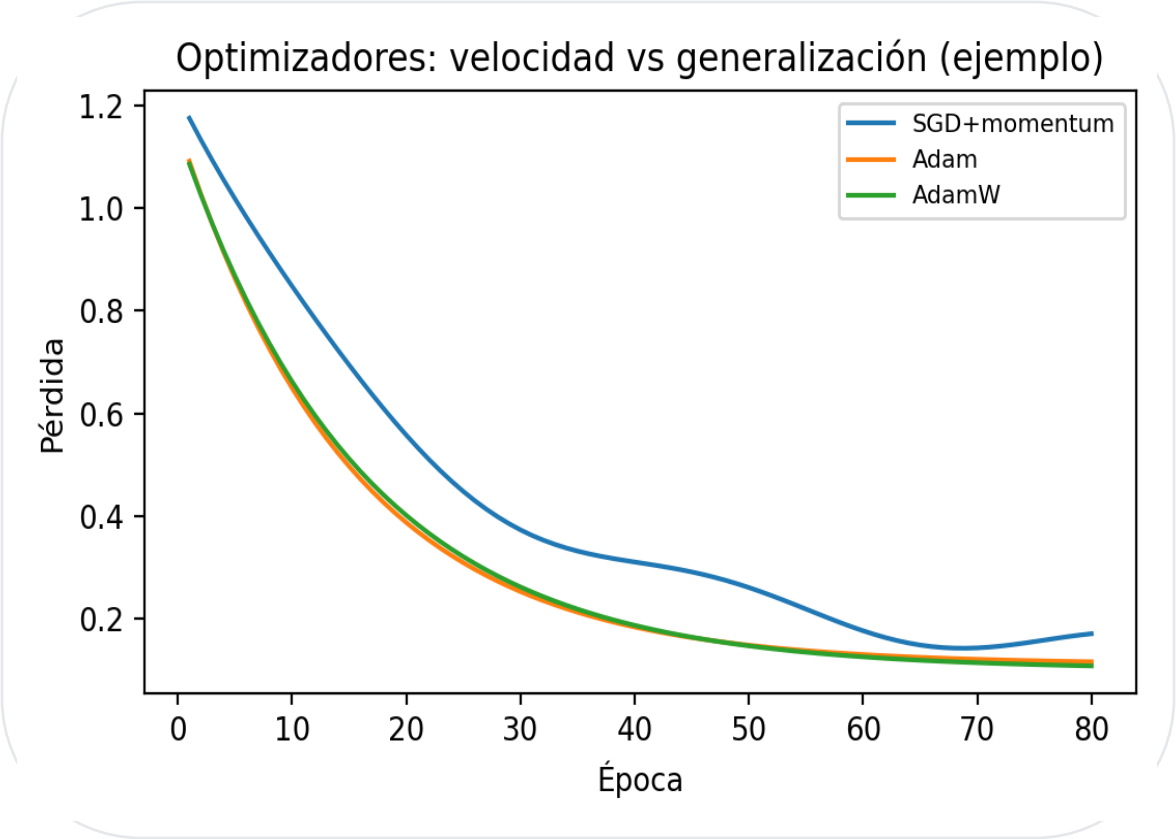
Caso A: se queda en un mínimo local



Caso B: explora y cambia de valle

7.2 Optimizadores modernos

SGD+momentum, Adam y AdamW



Resumen:

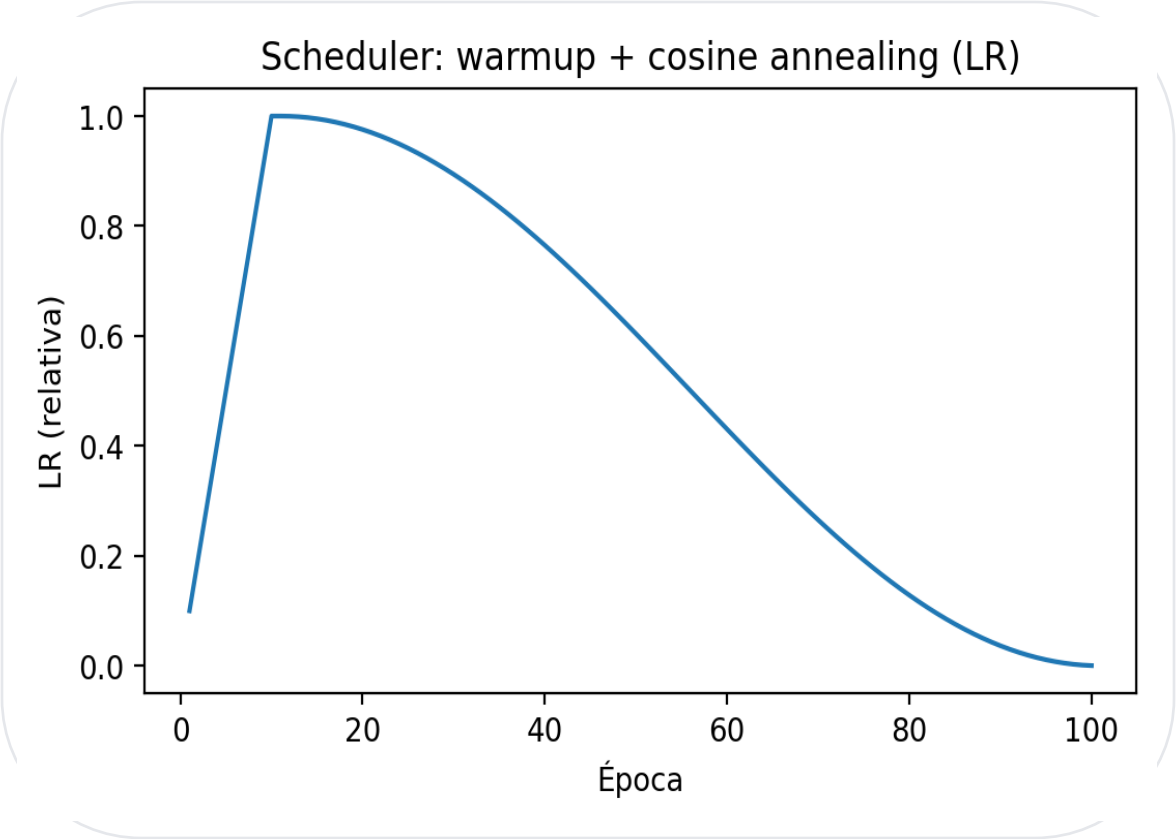
- SGD+momentum: simple, fuerte en generalización, requiere tuning de LR.
- Adam: adaptativo por parámetro, converge rápido.
- AdamW: separa weight decay del término adaptativo (mejor práctica actual).

Regla práctica

Compara "a igualdad de presupuesto": mismas épocas/batches y una búsqueda mínima de LR.

8. Scheduler de tasa de aprendizaje

Warmup + cosine como baseline sólido



¿Por qué usar scheduler?

- Warmup: evita pasos grandes cuando los gradientes son inestables al inicio.
- Cosine annealing: reduce LR gradualmente para refinar mínimos.
- Alternativas: step decay, one-cycle, plateau-based.

En el cuaderno

Incluye un ejemplo de scheduler cosine sobre un entrenamiento de clasificación.

8.1 Actualización con el schedule de tasa de aprendizaje

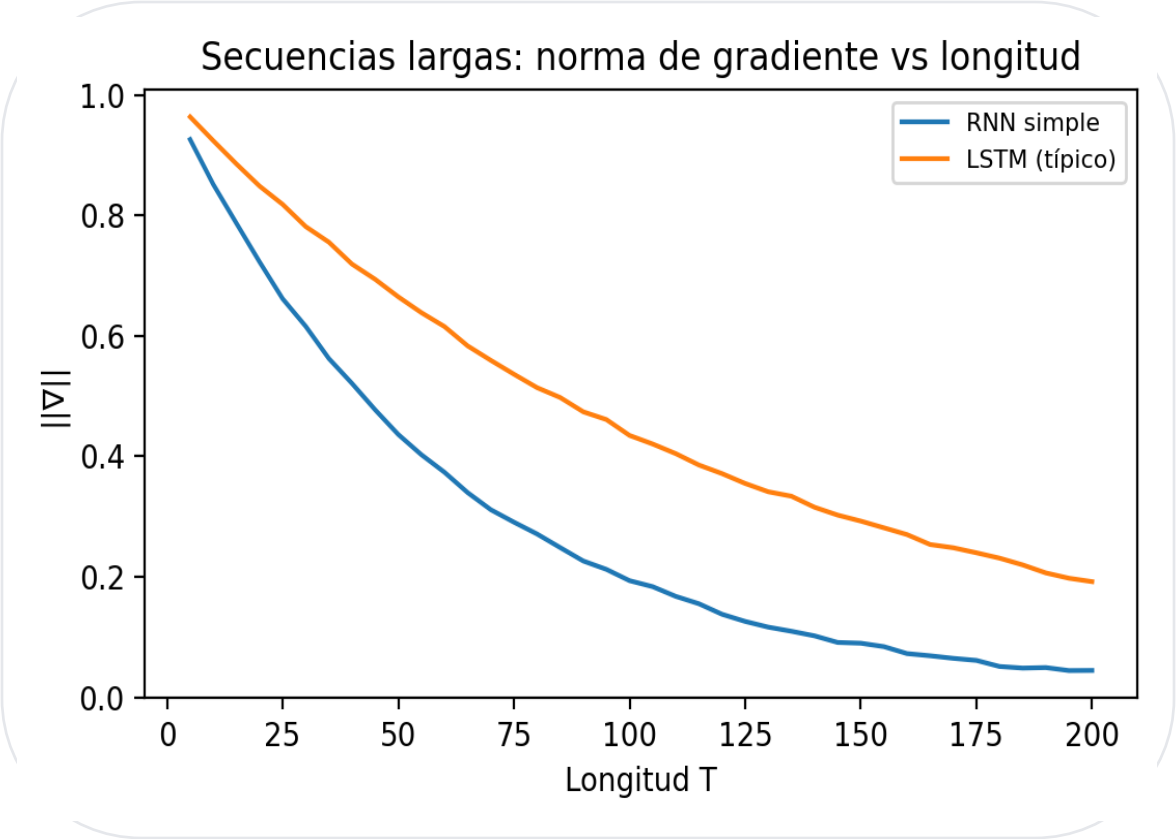
La tasa η_t cambia con t (warmup, cosine, step, plateau, …)

$$\underbrace{\Theta_{t+1}}_{\text{new parameters}} = \underbrace{\Theta_t}_{\text{old parameters}} - \underbrace{L(\eta_0, t)}_{\text{learning rate schedule}} \cdot \underbrace{g^t}_{\text{gradient}}$$

- L(η0, t) define η_t (por ejemplo: CosineAnnealingLR, StepLR, ReduceLROnPlateau).
- Intuición: η grande al inicio para explorar; η pequeña al final para refinar.
- En PyTorch: se usa típicamente scheduler.step() (por epoch o por batch, según el scheduler).

9. Secuencias largas

Limitaciones de RNN/LSTM y señal de gradiente



Problemas típicos:

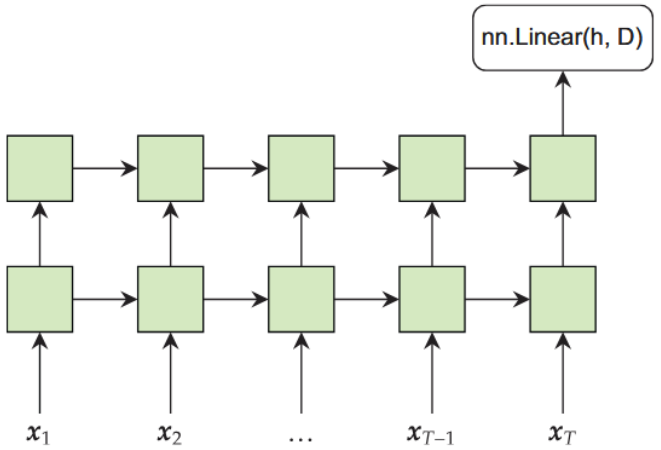
- Desaparición/explosión de gradientes al crecer la longitud T.
- Bottleneck: comprimir todo en un estado oculto limita capacidad.
- Cómputo secuencial: peor paralelismo y latencia.

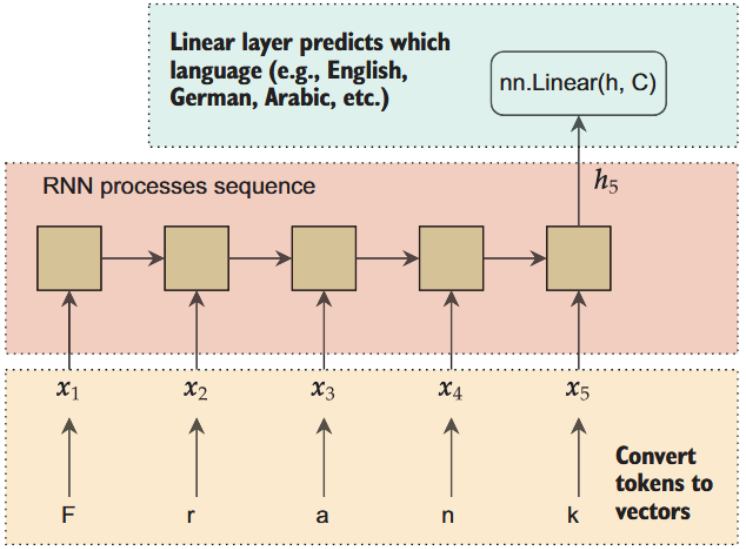
En el cuaderno

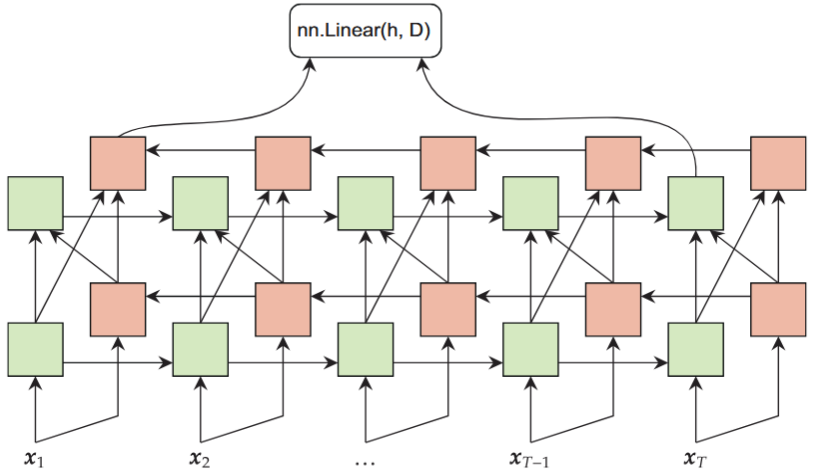
Experimento: norma de gradiente vs longitud + puente a traducción Seq2Seq.

9.1 RNN: desenrollado temporal y profundidad efectiva

Profundidad temporal y motivación para mecanismos de atención



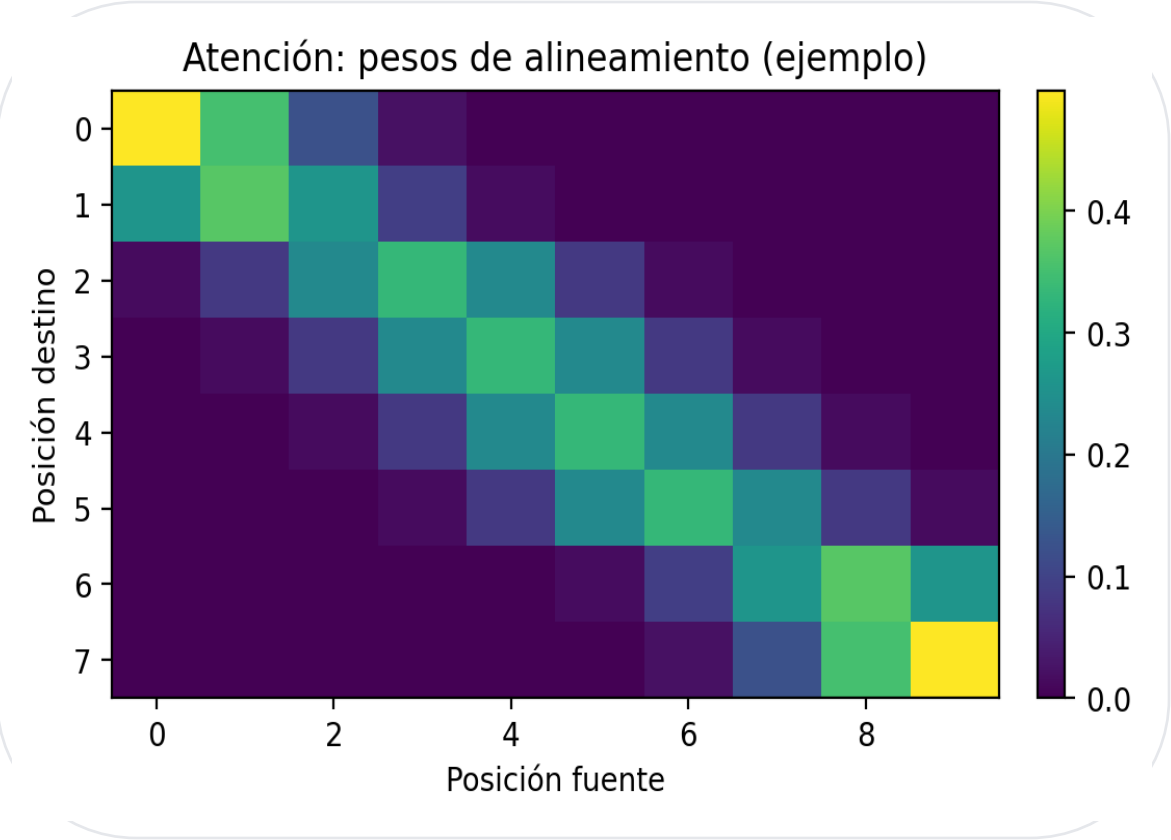




- Backpropagation Through Time (BPTT) crea rutas de gradiente de longitud T.
- A medida que T crece: vanishing/exploding gradients, y costo O(T).
- Atención permite accesos directos (paths cortos) a tokens lejanos.

10. Motivación para la atención

Consultar todos los estados relevantes, no solo el último



Idea:

context = $\Sigma_i \alpha_i \cdot h_i$ (α_i dependen del estado del decodificador)

- Evita el bottleneck: el decodificador “mira” toda la fuente.
- Mejora dependencias long-range (señal directa a posiciones tempranas).
- Interpretable: α forma un alineamiento (heatmap).

En el cuaderno

Atención aditiva (tipo Bahdanau) en un modelo Seq2Seq de traducción.