UNIVERSIDAD EAFIT
Escuela de Ciencias Aplicadas e Ingeniería

Lecture 03

Optimización para redes neuronales

Optimización

UNIVERSIDAD EAFIT | Escuela de Ciencias Aplicadas e Ingeniería

Objetivo: minimizar la función de pérdida $\mathcal{L}(\mathbf{w})$

Algoritmo:

1. Inicializar los pesos: $\mathbf{w}^{(0)} \sim$ aleatorio
2. Para cada iteración $t = 0, 1, 2, \ldots$:
   - Calcular el gradiente:

     $$\nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w}^{(t)})$$

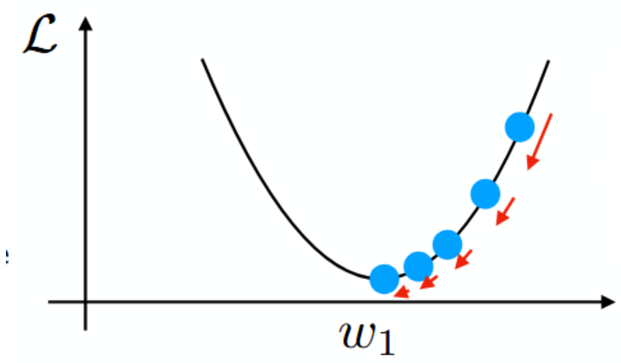
   - Actualizar los pesos:

     $$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta \nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w}^{(t)})$$

     donde $\eta > 0$ es la tasa de aprendizaje.
3. Repetir hasta convergencia

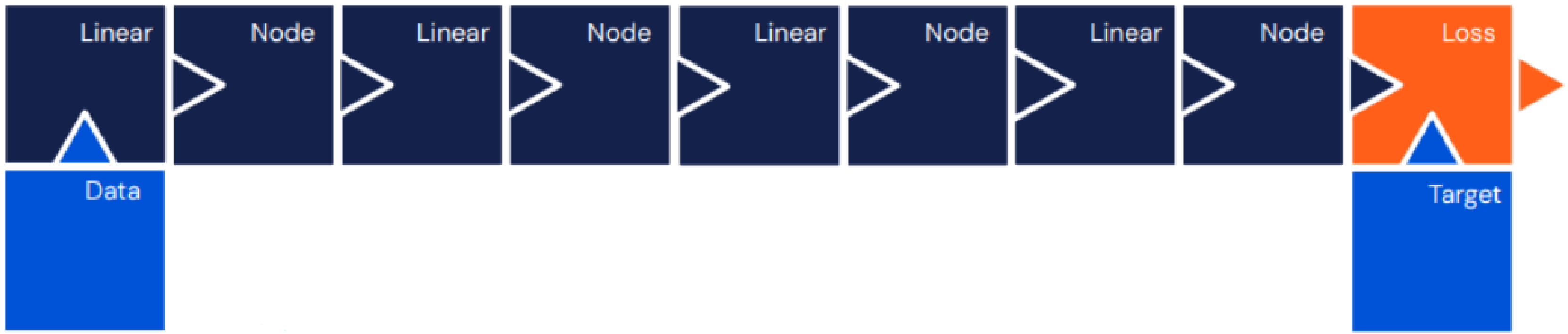
Input: Dataset $D = \{(\mathbf{x}^{(i)}, y^{(i)})\}$, learning rate η
Output: Parameters $\mathbf{W}, \mathbf{b}$
Initialize $\mathbf{W}, \mathbf{b}$;
for *epoch* $t = 1$ **to** T **do**
  foreach *mini-batch* $\mathcal{B} \subset D$ **do**
    Compute gradients $\nabla_{\mathbf{W}} \mathcal{L}_{\mathcal{B}}, \ \nabla_b \mathcal{L}_{\mathcal{B}}$;
    Update parameters:;
    $\mathbf{W} \leftarrow \mathbf{W} - \eta \nabla_{\mathbf{W}} \mathcal{L}_{\mathcal{B}}$;
    $\mathbf{b} \leftarrow \mathbf{b} - \eta \nabla_{\mathbf{b}} \mathcal{L}_{\mathcal{B}}$;
  end
end

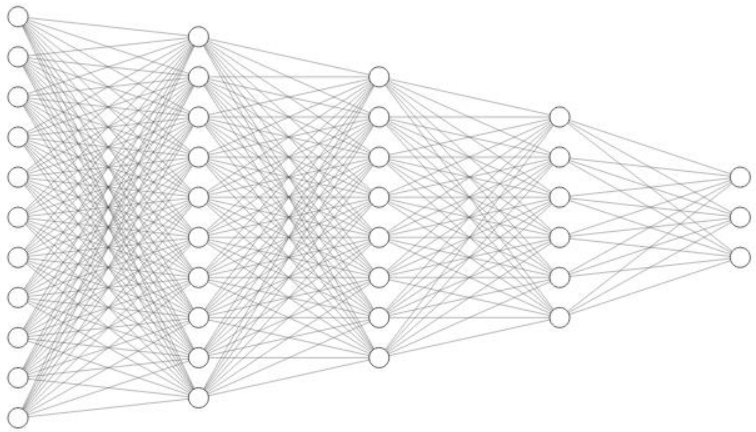
Mini-Batch Gradient Descent

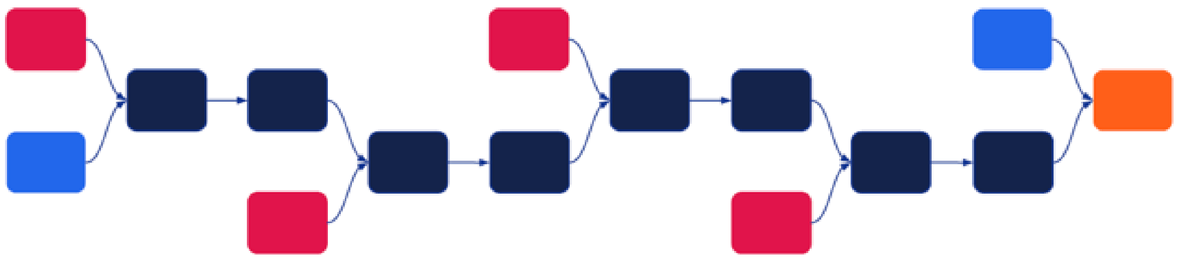


Grafos computacionales

UNIVERSIDAD EAFIT
Escuela de Ciencias Aplicadas e Ingeniería

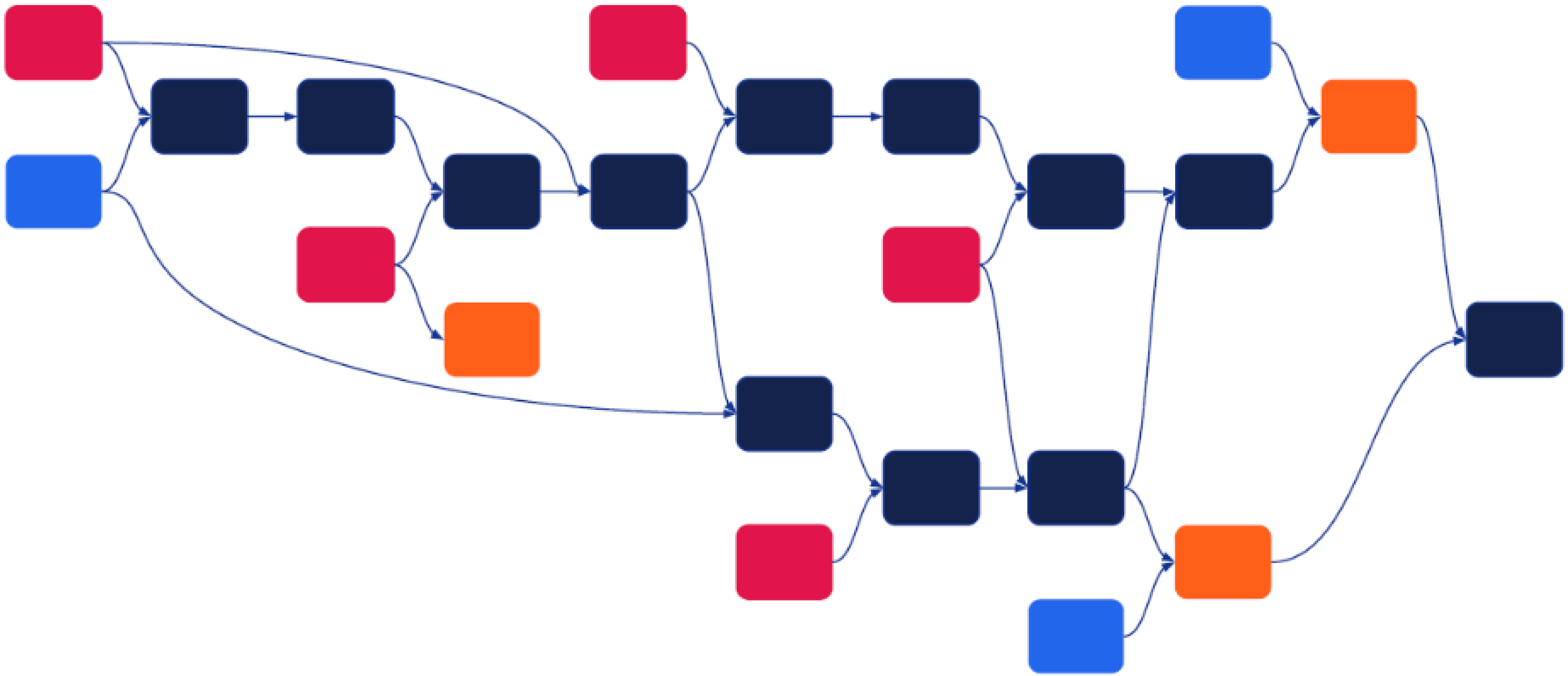




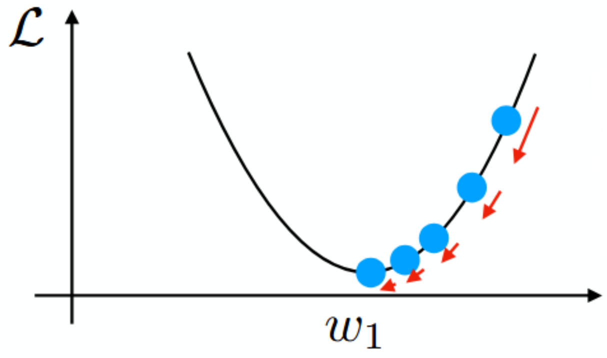


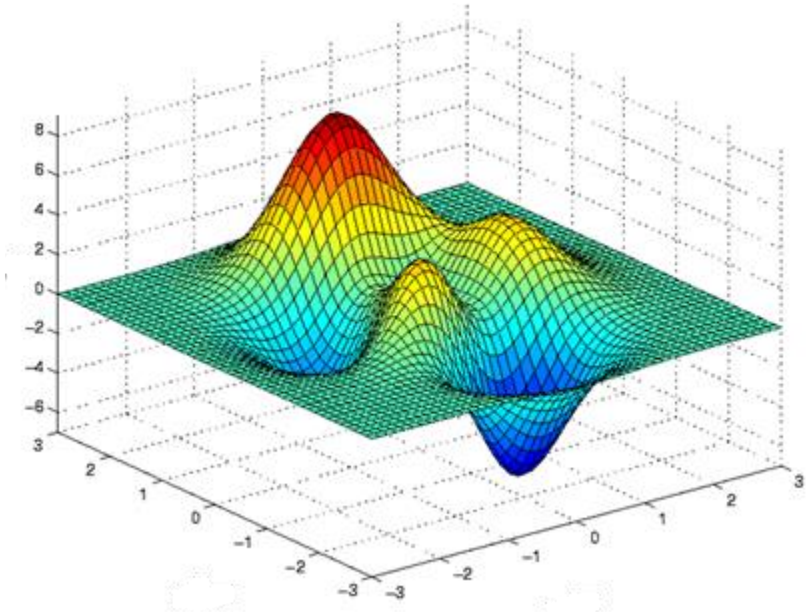
Grafos computacionales

Escuela de
Ciencias Aplicadas
e Ingeniería



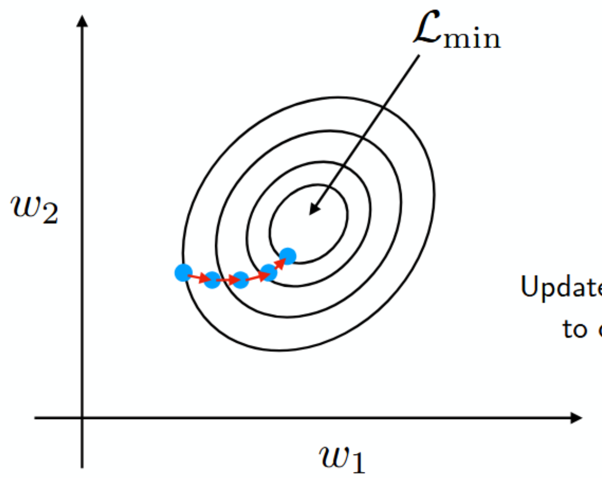
Derivadas y Gradientes





Optimización - SGD

Input: Dataset $D = \{(\mathbf{x}^{(i)}, y^{(i)})\}$, learning rate η
Output: Parameters $\mathbf{W}, \mathbf{b}$
Initialize $\mathbf{W}, \mathbf{b}$;
for *epoch* $t = 1$ **to** T **do**
    foreach *mini-batch* $\mathcal{B} \subset D$ **do**
        Compute gradients $\nabla_{\mathbf{W}} \mathcal{L}_{\mathcal{B}},\ \nabla_{b} \mathcal{L}_{\mathcal{B}}$;
        Update parameters:;
        $\mathbf{W} \leftarrow \mathbf{W} - \eta \nabla_{\mathbf{W}} \mathcal{L}_{\mathcal{B}}$;
        $\mathbf{b} \leftarrow \mathbf{b} - \eta \nabla_{\mathbf{b}} \mathcal{L}_{\mathcal{B}}$;
    end
end

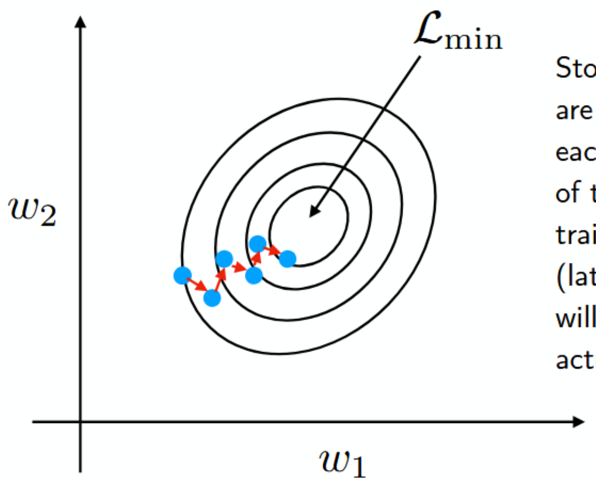


Updates perpendicular to contour lines

Nuestros gradientes provienen de *minibatches*, ¡por eso pueden ser ruidosos!

$$L(W) = \frac{1}{N} \sum_{i=1}^{N} L_i(x_i, y_i, W)$$

$$\nabla_W L(W) = \frac{1}{N} \sum_{i=1}^{N} \nabla_W L_i(x_i, y_i, W)$$

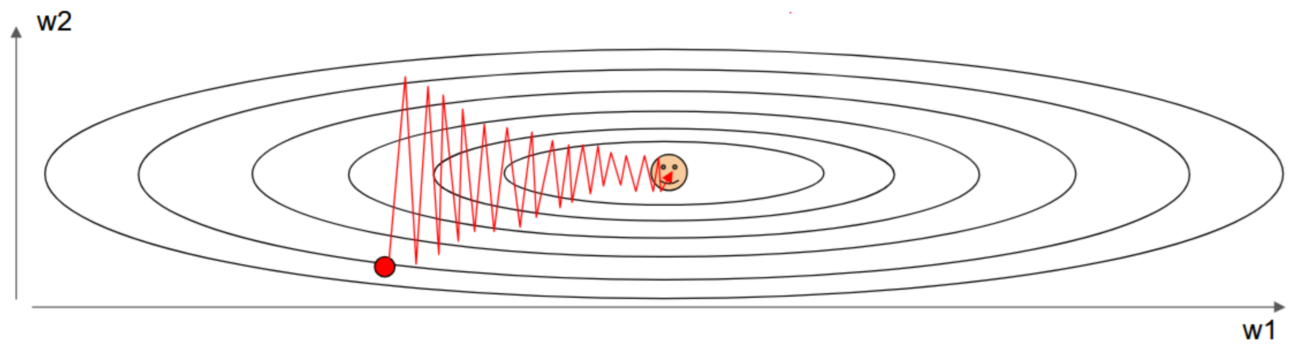


Stochastic updates are a bit noisier, because each batch is an approximation of the overall loss on the training set (later, in deep neural nets, we will see why noisier updates are actually helpful)

Optimización - SGD

¿Qué pasa si la función de pérdida cambia rápidamente en una dirección y lentamente en otra?

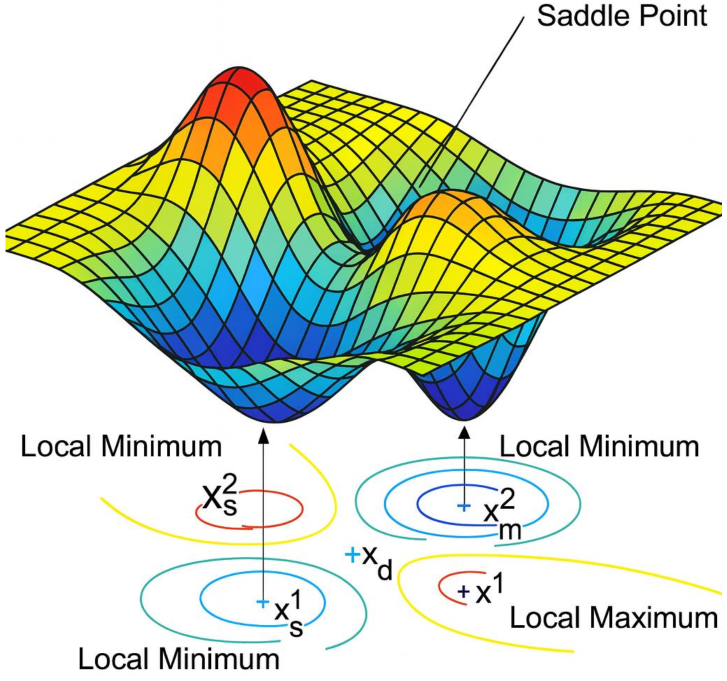
- Progreso muy lento en la dimensión poco inclinada
- Inestabilidad en la dirección empinada



Optimización - SGD

¿Qué pasa si la función de pérdida tiene mínimos locales o puntos de silla?

- Puede quedar atrapado en un **mínimo local** o cerca de un **punto de silla**
- En espacios de alta dimensión, los **puntos de silla** son mucho más frecuentes que los mínimos locales



Optimización - SGD

$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

python

```python
while True:
    dx = compute_gradient(x)
    x -= learning_rate * dx
```

SGD + Momentum

SGD + Momentum (forma 1)

$$v_{t+1} = \rho v_t - \alpha \nabla f(x_t)$$

$$x_{t+1} = x_t + v_{t+1}$$

```python
vx = 0
while True:
    dx = compute_gradient(x)
    vx = rho * vx - learning_rate * dx
    x += vx
```

SGD + Momentum (forma 2)

$$v_{t+1} = \rho v_t + \nabla f(x_t)$$

$$x_{t+1} = x_t - \alpha v_{t+1}$$

```python
vx = 0
while True:
    dx = compute_gradient(x)
    vx = rho * vx + dx
    x -= learning_rate * vx
```

RMSProp

- Mantiene un promedio exponencial de los gradientes al cuadrado.
- Escala cada componente del gradiente de forma independiente.
- **Tasa de aprendizaje adaptativa por parámetro**.
- Reduce oscilaciones en superficies con curvatura desigual.
- Ampliamente usado en redes profundas.

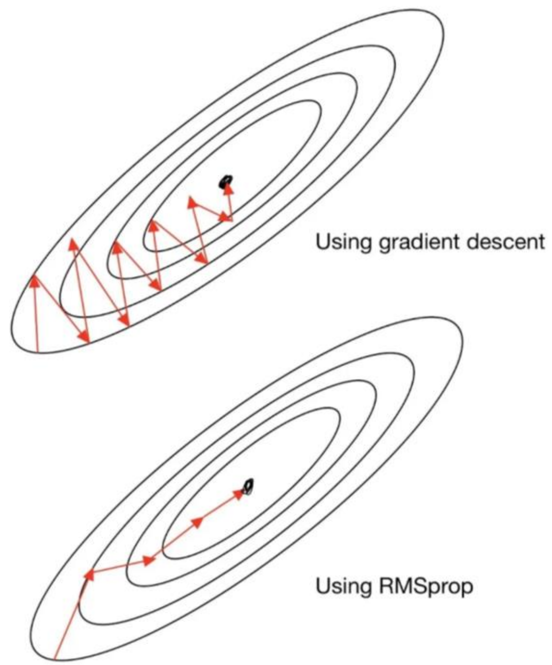
$$g_t = \nabla f(x_t)$$

$$s_t = \rho \cdot s_{t-1} + (1-\rho) \cdot g_t^2$$

$$x_{t+1} = x_t - \frac{\alpha}{\sqrt{s_t} + \varepsilon} \cdot g_t$$

python

```python
grad_squared = 0
while True:
    dx = compute_gradient(x)
    grad_squared = decay_rate * grad_squared + (1 - decay_rate) * dx * dx
    x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```



Adam (90%)

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\nabla f(x_t)$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2)\left(\nabla f(x_t)\right)^2$$

$$x_{t+1} = x_t - \frac{\alpha \cdot m_t}{\sqrt{v_t} + \varepsilon}$$

- Combina momentum (1er momento) y RMSProp (2do momento).
- Ajusta la tasa de aprendizaje adaptativamente por parámetro.
- Estable, eficiente y muy utilizado en deep learning.
- Popular en tareas con gradientes ruidosos o escasos.

```python
first_moment = 0
second_moment = 0
while True:
    dx = compute_gradient(x)
    first_moment = beta1 * first_moment + (1 - beta1) * dx
    second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
    x -= learning_rate * first_moment / (np.sqrt(second_moment) + 1e-7)
```

Momentum

RMSProp

Adam

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\nabla f(x_t)$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2)\left(\nabla f(x_t)\right)^2$$

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

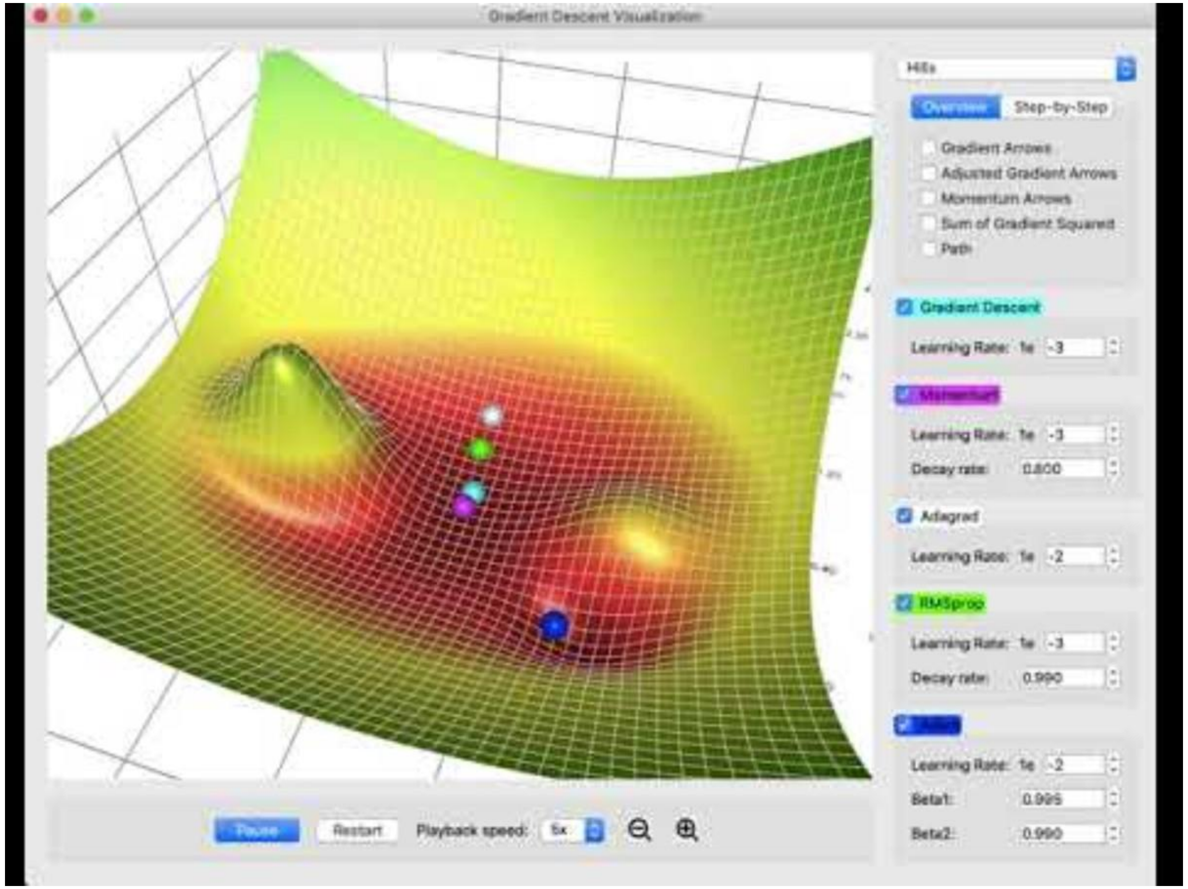
$$x_{t+1} = x_t - \frac{\alpha \cdot \hat{m}_t}{\sqrt{\hat{v}_t} + \varepsilon}$$

python

```python
first_moment = 0
second_moment = 0
for t in range(1, num_iterations):
    dx = compute_gradient(x)
    first_moment = beta1 * first_moment + (1 - beta1) * dx
    second_moment = beta2 * second_moment + (1 - beta2) * dx * dx

    first_unbias = first_moment / (1 - beta1 ** t)
    second_unbias = second_moment / (1 - beta2 ** t)

    x -= learning_rate * first_unbias / (np.sqrt(second_unbias) + 1e-7)
```

- Combina **momentum** y **RMSProp** con corrección de sesgo.
- Usa estimaciones corregidas de primer y segundo momento $(\hat{m}_t, \hat{v}_t)$.
- Corrige el sesgo inicial porque $m_0, v_0 = 0$.
- Parámetros recomendados:
  - $\beta_1 = 0.9$
  - $\beta_2 = 0.999$
  - $\alpha = 10^{-3}$ o $5 \cdot 10^{-4}$

Comparación



https://github.com/lilipads/gradient_descent_viz

AdamW

Escuela de Ciencias Aplicadas e Ingeniería

Adam estándar:

- Aplica L2 dentro del gradiente:

$$\nabla f(x) \leftarrow \nabla f(x) + \lambda x$$

- El término de regularización se mezcla con el gradiente.
- Puede tener un efecto no deseado al ser escalado por momentos.

python

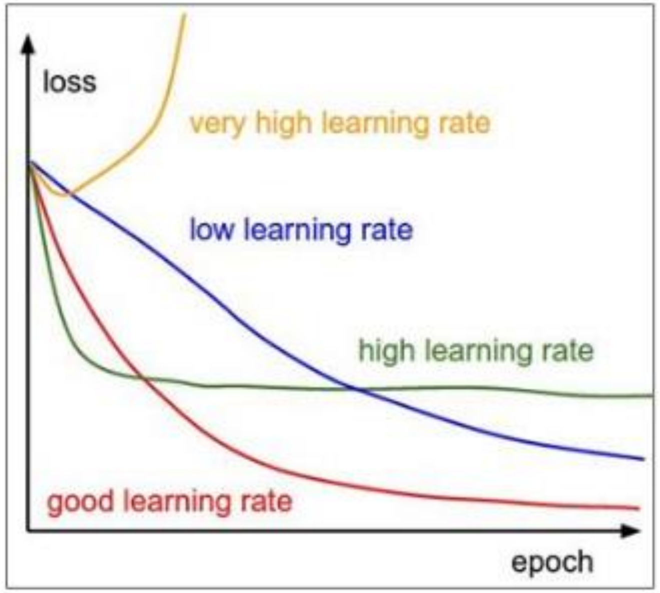
```python
first_moment = 0
second_moment = 0
for t in range(1, num_iterations):
    dx = compute_gradient(x)

    # Regularización L2 desacoplada (weight decay)
    x *= (1 - learning_rate * weight_decay)

    first_moment = beta1 * first_moment + (1 - beta1) * dx
    second_moment = beta2 * second_moment + (1 - beta2) * dx * dx

    first_unbias = first_moment / (1 - beta1 ** t)
    second_unbias = second_moment / (1 - beta2 ** t)

    x -= learning_rate * first_unbias / (np.sqrt(second_unbias) + 1e-7)
```

AdamW (Weight Decay desacoplado):

- Separa explícitamente la regularización.
- Actualiza parámetros así:

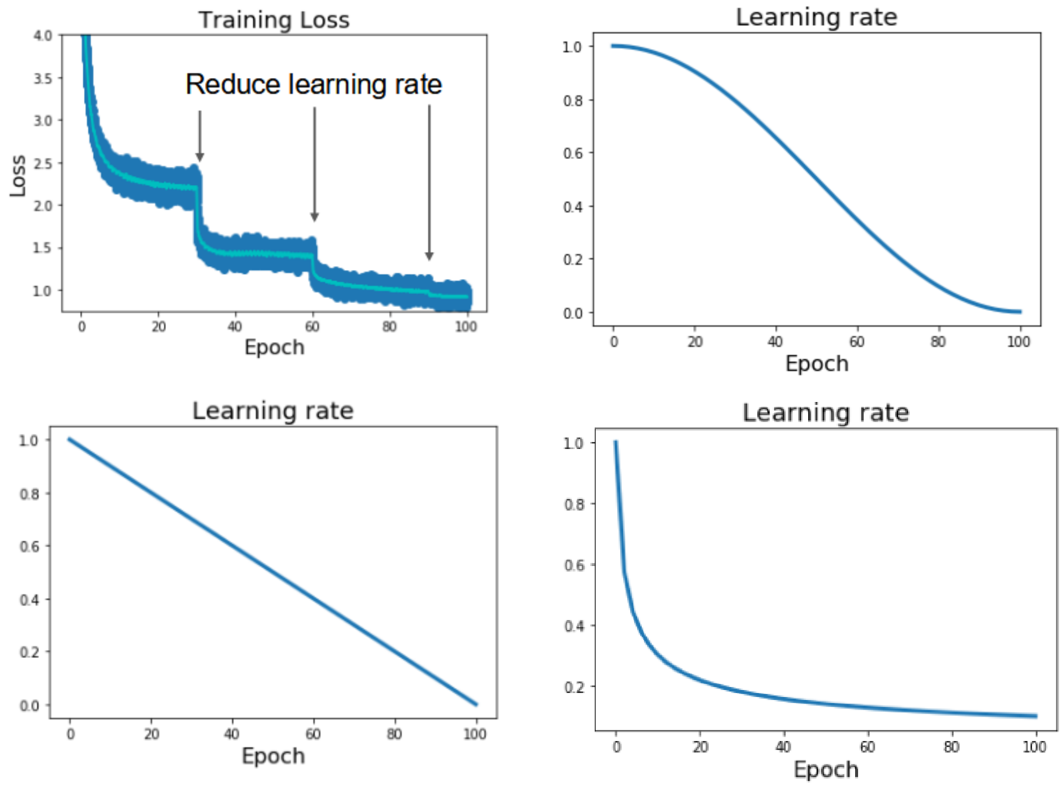
$$x \leftarrow x \cdot (1 - \alpha \cdot \lambda)$$

Learning rate schedulers



- **P:** ¿Cuál de estas tasas de aprendizaje es mejor usar?
- **R:** En realidad, todas podrían ser buenas tasas de aprendizaje.

Learning rate schedulers



Step: Reducir la tasa de aprendizaje en ciertos puntos fijos. Ejemplo: multiplicar por 0.1 en las épocas 30, 60 y 90.

Cosine:

$$\alpha_t = \frac{1}{2}\alpha_0\left(1+\cos\left(\frac{t\pi}{T}\right)\right)$$

Linear:

$$\alpha_t = \alpha_0\left(1-\frac{t}{T}\right)$$

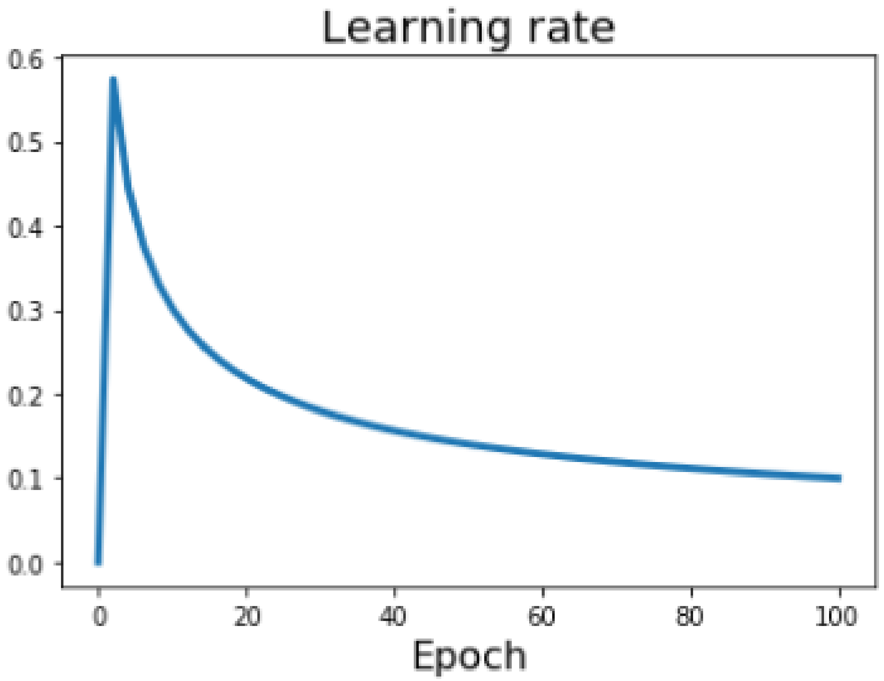
Inverse sqrt:

$$\alpha_t = \frac{\alpha_0}{\sqrt{t}}$$

- α_0: tasa de aprendizaje inicial
- α_t: tasa de aprendizaje en la época t
- T: número total de épocas

- Loshchilov and Hutter, ICLR 2017
- Radford et al., 2018 (GPT)
- Feichtenhofer et al., arXiv 2018
- Child et al., arXiv 2019 (Sparse Transformers)

Linear Warmup



- Las tasas de aprendizaje altas al inicio pueden hacer que la pérdida explote.
- El **warmup lineal** evita esto aumentando LR desde 0 de forma progresiva.
- Comúnmente usado en los primeros 5,000 pasos.
- Regla empírica: si aumentas el batch en N, aumenta LR en N.

Goyal et al., "Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour", arXiv 2017.