Deep Learning con **Pytorch**

Juan Pablo Morales
@juanpamf

SGD y Backpropagation

Una analogía

Heurísticas

- Seguir la dirección del gradiente
- Tener un componente aleatorio para evitar mínimos locales
- Ajustar el largo del paso (learning rate)

Bajada de gradiente

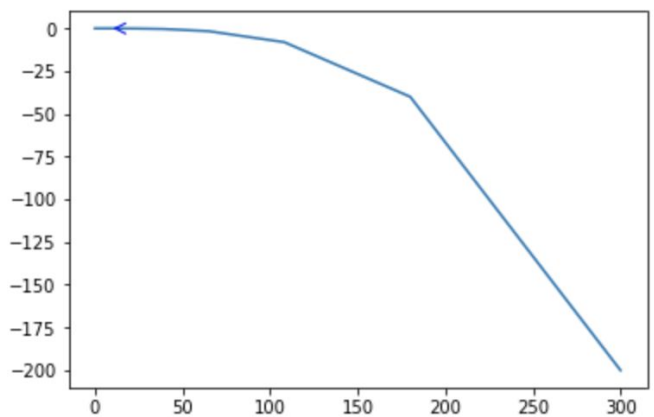
```
step = 0.2
point=(300,-200)
epsilon = 1e-20
iter = 0
decay = 0.999

def sum_square(x):
  return x[0]*x[0] + 2*x[1]*x[1]

def gradient(x):
  return (2*x[0], 4*x[1])

def tuple_norm(x):
  return x[0]*x[0] + x[1]*x[1]

def compute_gradient_step(gradient,step):
  return tuple(x*step for x in gradient)

def compute_new_point(point, gradient):
  return (point[0] - gradient[0], point[1] - gradient[1])



while tuple_norm(gradient(point)) > epsilon :
  grad = gradient(point)
  point = compute_new_point(point, compute_gradient_step(grad, step))
  step = step*decay
  if iter % 3 == 1:
    print(point)
  iter = iter + 1

print(f"iterations : {iter}\nfinal value :\n({point}, {sum_square(point)})")
```

$$f(x,y) = x^2 + 2y^2$$

$$\frac{\partial f}{\partial x} = 2x$$

$$\frac{\partial f}{\partial y} = 4y$$

$$\nabla f = (\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}) = (2x, 4y)$$

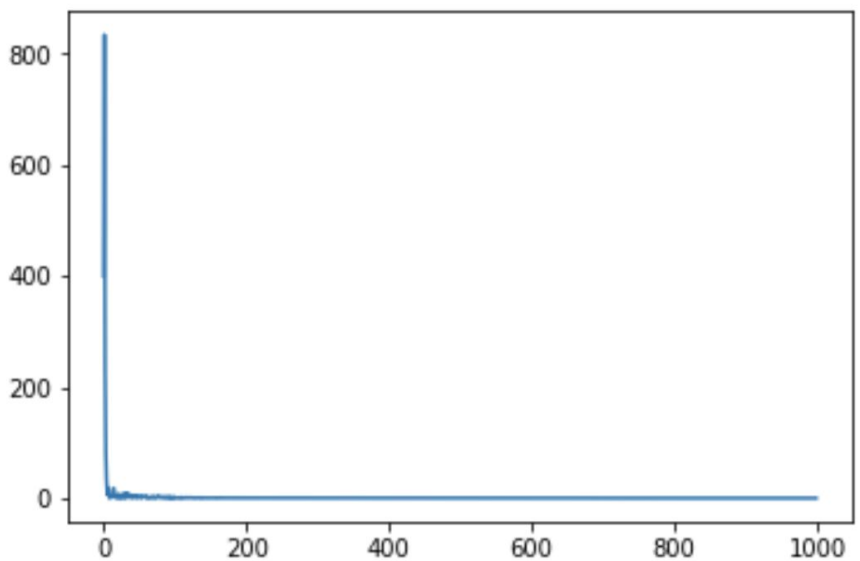


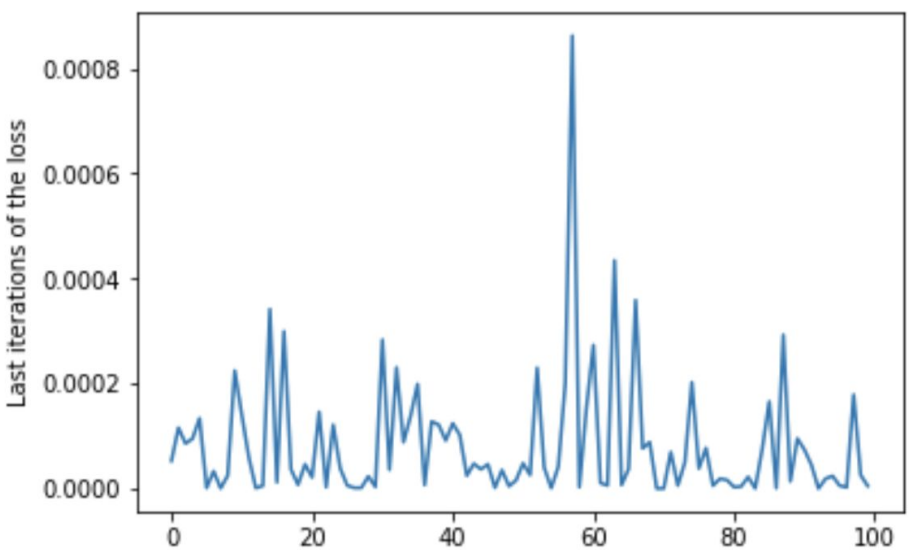
SGD

- La pérdida es una suma sobre todo el dataset
  - Costo de cómputo alto
- Introducimos aleatoriedad eligiendo al azar un dato sobre el cual calcular la pérdida
- Aplicamos un paso de la bajada de gradiente

SGD

Excelentes propiedades teóricas





Mini-batch SGD

- **Bajada de gradiente (dataset completo)**
  - Rápido, puede bloquearse en mínimos locales
- **SGD (1 dato)**
  - Converge más lentamente, evita mínimos locales
- **Mini-batch SGD (k datos)**
  - Buen compromiso entre velocidad y aleatoreidad.

Backpropagation

- Una red neuronal es un grafo de cómputo

  Compuesto de entradas, salidas, parámetros, y funciones lineales y de activación

- Su estructura nos permite calcular el gradiente de forma más simple

  Algoritmo de backpropagation

- Librerias como pytorch implementan de forma automática este algoritmo

  Autograd (reverse mode automatic differentiation)

Backpropagation

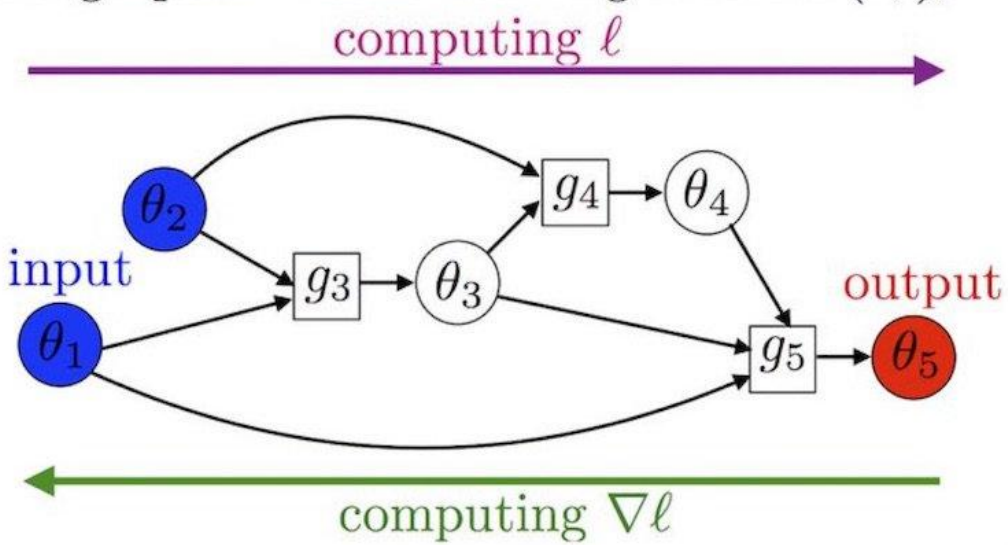
Computer program ⇔ directed acyclic graph ⇔ linear ordering of nodes $(\theta_r)_r$

forward

```
function ℓ(θ_1, ..., θ_M)
  for r = M + 1, ..., R
    θ_r = g_r(θ_Parents(r))
  return θ_R
```

backward

```
function ∇ℓ(θ_1, ..., θ_M)
  ∇_R ℓ = 1
  for r = R − 1, ..., 1
    ∇_r ℓ = Σ_{s∈Child(r)} ∂_r g_s(θ) ∇_s ℓ
  return (∇_1 ℓ, ..., ∇_M ℓ)
```



$$\text{“}\frac{\partial \ell}{\partial \theta_r} = \sum_{s \in \text{Child}(r)} \frac{\partial \ell}{\partial \theta_s}\frac{\partial \theta_s}{\partial \theta_r}\text{”}$$

$\nabla_r \ell(\theta)$ $\quad$ $\nabla_s \ell(\theta)$ $\quad$ $\partial_r g_s(\theta)$

Las piezas del puzzle encajan

```
[ ]  net = Net().to(device)
     loss_fn = nn.NLLLoss()
     optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)

     #bloque clave para entrenar una red neuronal
     for inputs, targets in trainloader:
         optimizer.zero_grad()

         outputs = net(inputs)
         loss = loss_fn(outputs, targets)

         loss.backward()
         optimizer.step()
```