Autograd — Rétropropagation depuis Zéro

Jour 3 — Matin

Julien Rolland

Formation M2 Développement Fullstack

Jour 3

Plan du module

1. Limite du Linéaire
2. La Non-Linéarité
3. Credit Assignment
4. Graphe de Calcul
5. Forward & Backward Pass
6. Micro-Autograd en Python
7. Vers PyTorch

La Limite du Linéaire : Le Problème du XOR

Rappel J2 : Un neurone = une droite

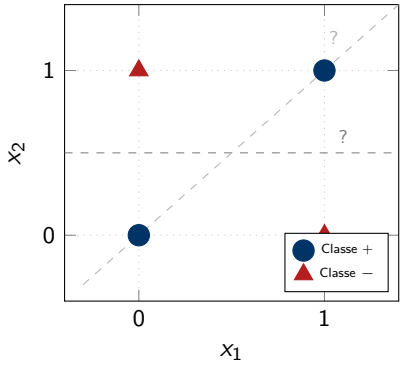
- $y = \text{sign}(\boldsymbol{w} \cdot \boldsymbol{x} + b)$
- Sépare l'espace en deux **demi-plans**

XOR : Non linéairement séparable

- Classe • : $(0, 0)$ et $(1, 1)$
- Classe ▲ : $(0, 1)$ et $(1, 0)$
- **Aucune droite** ne les sépare

Solution

« Tordre l'espace » $\Rightarrow$ **Couches cachées** (*Hidden Layers*)



L'Ingrédient Secret : La Non-Linéarité

Sans activation : réseau inutile

$$W_2(W_1 \cdot \mathbf{X}) = \underbrace{(W_2 W_1)}_{W_{\text{comb}}} \cdot \mathbf{X}$$

1000 couches = 1 seule couche linéaire

Avec activation σ : expressivité

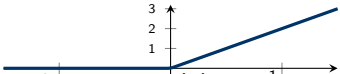
$h = \sigma(W_1 \cdot \mathbf{X})$ puis $y = W_2 h$

$\Rightarrow$ **Non réductible** à une seule couche

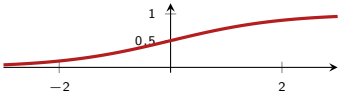
ReLU : La Norme Industrielle

$f(x) = \max(0, x)$ Dérivée : 0 ou 1

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



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



Le « Credit Assignment Problem »

Dans un réseau profond

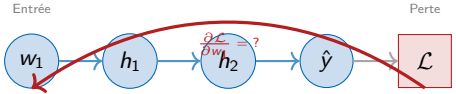
- On calcule $\mathcal{L}$ à la **sortie**
- Les poids $w_1, w_2, \ldots$ sont au **début**
- **Question :** de combien modifier w_i pour réduire $\mathcal{L}$?

Le problème fondamental

L'influence d'un poids précoce passe par **toutes les couches suivantes**

Solution : Rétropropagation

Propager $\frac{\partial \mathcal{L}}{\partial w_i}$ **à l'envers** le long du graphe



Le Graphe de Calcul (DAG)

Modélisation

- Toute fonction ⇒ graphe **orienté acyclique**
- **Nœuds** : opérations élémentaires ($+$, $\times$, exp...)
- **Arêtes** : variables / tenseurs

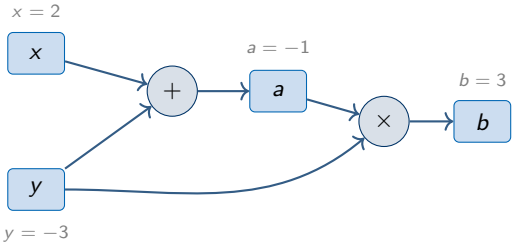
Intérêt

Chaque nœud calcule sa **dérivée locale** ⇒ les gradients se chaînent vers l'entrée

Notre exemple

$f(x, y) = (x + y) \times y$

$a = x + y, \quad b = a \times y$



Le Forward Pass

Mécanique

1. Parcourir le graphe de **gauche à droite**
2. Calculer chaque nœud
3. **Stocker** les valeurs intermédiaires

Pourquoi stocker ?

Les valeurs intermédiaires sont **nécessaires** lors du backward pass pour calculer les dérivées locales

```python
x = 2.0
y = -3.0

# Noeud 1 : addition
a = x + y    # a = -1.0

# Noeud 2 : multiplication
b = a * y    # b = 3.0  <- Sortie

print(f"Sortie b = {b}")
# Sortie b = 3.0
```

La Chain Rule (Règle de la Chaîne)

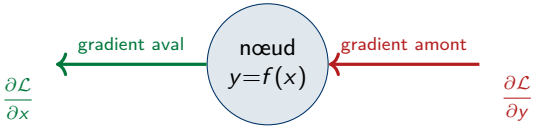
Formule

Si $z = f(y)$ et $y = g(x)$, alors :

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} \cdot \frac{\partial y}{\partial x}$$

Pour les graphes de calcul

- **Gradient amont** $\frac{\partial \mathcal{L}}{\partial y}$ arrive de la sortie
- Nœud calcule sa **dérivée locale** $\frac{\partial y}{\partial x}$
- **Gradient aval** = amont $\times$ local



locale : $\frac{\partial y}{\partial x}$

$$\frac{\partial \mathcal{L}}{\partial x} = \frac{\partial \mathcal{L}}{\partial y} \cdot \frac{\partial y}{\partial x}$$

Le Backward Pass (Reverse Mode)

Algorithme

1. Initialiser $\frac{\partial \mathcal{L}}{\partial \mathcal{L}} = 1$
2. Parcourir **de droite à gauche**
3. Chaque nœud : grad aval $\times$ dérivée locale

Résultat clé

Un seul backward

$\Rightarrow$ gradients de **tous** les paramètres

```
# b = a * y   =>   db/da = y,   db/dy = a
# a = x + y   =>   da/dx = 1,   da/dy = 1

# Init : gradient de la sortie
db = 1.0

# Noeud * : b = a * y
da   = db * y     # 1.0 * (-3.0) = -3.0
dy_2 = db * a     # 1.0 * (-1.0) = -1.0

# Noeud + : a = x + y
dx   = da * 1.0   # -3.0
dy_1 = da * 1.0   # -3.0

# Gradients finaux (accumulation)
grad_x = dx              # -3.0
grad_y = dy_2 + dy_1     # -4.0
```

Encapsuler le `float`

- `data` : valeur numérique
- `grad` : gradient (init. à 0)
- `_prev` : parents dans le graphe
- `_backward` : gradient local

Principe

Code mathématique **normal** + construction du graphe **automatique**

```python
class Value:
    def __init__(self, data, _children=()):
        self.data = data
        self.grad = 0.0
        self._prev = set(_children)
        self._backward = lambda: None

    def __repr__(self):
        return (f"Value(data={self.data:.3f}, "
                f"grad={self.grad:.3f})")

# Utilisation
x = Value(2.0)
y = Value(-3.0)
print(x)  # Value(data=2.000, grad=0.000)
```

Dunder methods

- `__add__` → opérateur $+$
- `__mul__` → opérateur $\times$
- Graphe construit **implicitement**

Règle pour $+$

$$\frac{\partial(a+b)}{\partial a} = 1, \quad \frac{\partial(a+b)}{\partial b} = 1$$

Règle pour $\times$

$$\frac{\partial(a \cdot b)}{\partial a} = b, \quad \frac{\partial(a \cdot b)}{\partial b} = a$$

```python
def __add__(self, other):
    out = Value(self.data + other.data,
                (self, other))
    def _backward():
        self.grad  += 1.0 * out.grad
        other.grad += 1.0 * out.grad
    out._backward = _backward
    return out

def __mul__(self, other):
    out = Value(self.data * other.data,
                (self, other))
    def _backward():
        self.grad  += other.data * out.grad
        other.grad += self.data  * out.grad
    out._backward = _backward
    return out
```

Ce matin : Micro-Autograd scalaire

- Autograd sur des **scalaires** ($x \in \mathbb{R}$)
- Graphe construit manuellement
- Gradients : nombres simples

PyTorch : Même logique, Tenseurs

- Variables : **tenseurs** $X \in \mathbb{R}^{m \times n}$
- Graphe construit **automatiquement**
- Gradients : matrices (Jacobiennes)

	Scalaire	**Tenseur**
Valeur	$x \in \mathbb{R}$	$X \in \mathbb{R}^{m \times n}$
Gradient	$\partial \mathcal{L} / \partial x$	$\partial \mathcal{L} / \partial X$
Type	scalaire	Jacobienne
Règle	Chaîne	Chaîne vectorielle
Classe	`Value`	`torch.Tensor`

Cet après-midi : PyTorch

```
x = torch.tensor(..., requires_grad=True)
```