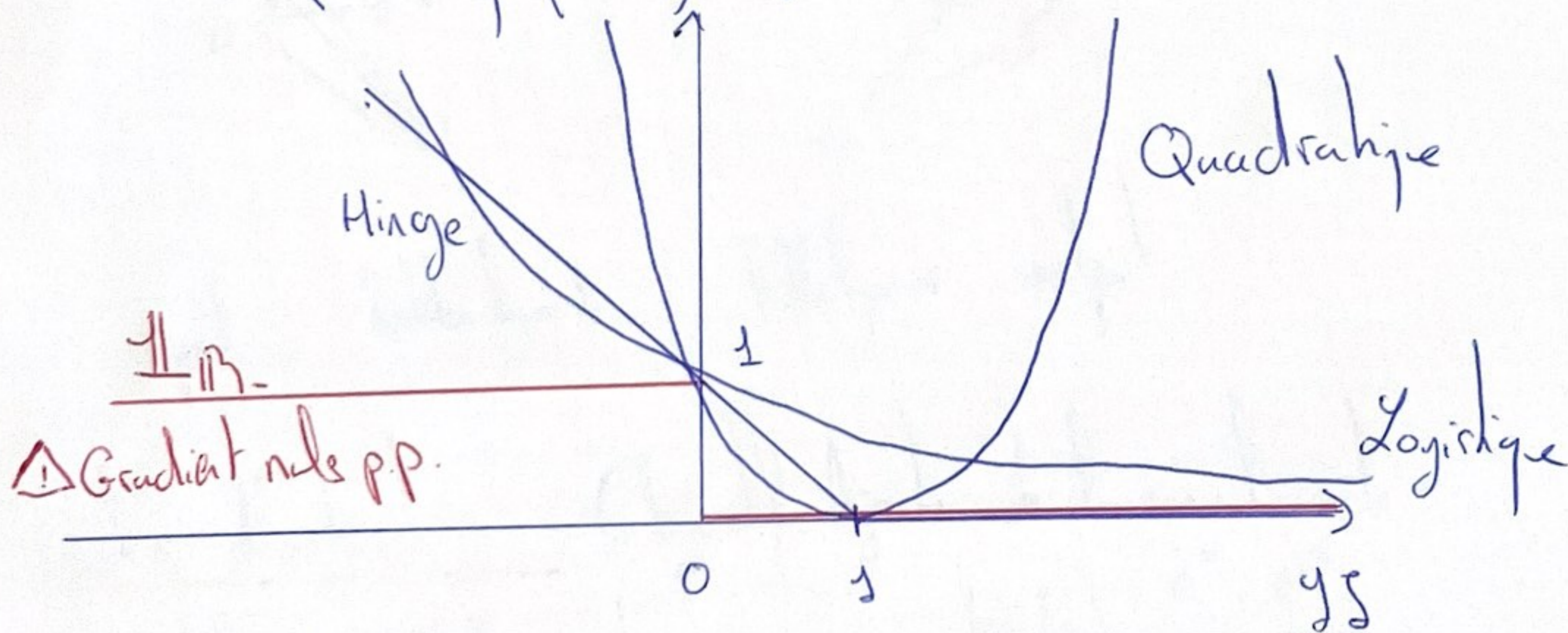


Réseaux de Neurones

Contexte: \hat{f} est argmin $\frac{1}{n} \sum_{i=1}^n P(y_i, f(x_i)) + \Omega(f)$
 où f est la fonction de perte. $\Omega(f)$ est la régularisation.

Fonction de perte:

- L1 $P(y, s) = |y - s|$ Moy.
- Quadratique $P(y, s) = (y - s)^2$ Moy.
- Logistique $P(y, s) = \log(1 + e^{-ys})$ Class.
- Exponentielle $P(y, s) = e^{-ys}$ Class.
- Hinge $P(y, s) = (1 - ys)_+$ Class.
- Quadratique (class) $P(y, s) = (1 - ys)^2$ Class.



$$P(y, s) = \phi(ys)$$

Le signe dit si la classification est bonne.

Réseaux de neurones :

Une classe de fonctions F qui :

- Est expressive
- Permet de calculer les gradients efficacement
- Est hautement parallélisable.

$$\text{output} = A_{\text{out}} \sigma (A_N \dots \sigma (A_2 \sigma (A_1 \text{input} + b_1) + b_2) \dots + b_N) + b_{\text{out}}$$



...



σ : fonction d'activation.

input hidden layer 1 hidden layer n output

Fonctions d'activation : Objectif : Ajouter de la non-linéarité

- Sigmoid (x) = $\frac{1}{1 + e^{-x}}$

- tan (x) = $\frac{e^x - e^{-x}}{e^x + e^{-x}}$

- ReLU (x) = $x_+ = \max(x, 0)$


- Softmax (x)_i = $\frac{e^{x_i}}{\sum_j e^{x_j}}$

Apprentissage de feature maps :

(3)

$$M(z) = A_{out} \text{hidden_layer}_n(z) + b_{out}$$

Ainsi, $M(\cdot)$ est linéaire en $\text{hidden_layer}_n(\cdot)$. Il est possible de voir un réseau de neurones comme un modèle linéaire à sa dernière couche dont le but est d'extraire des features.

 Comme $\text{hidden_layer}_n(\cdot)$ est une fonction apprise, le problème n'est pas vraiment linéaire ou convexe!

Homogénéité des réseaux Meta.

Lorsque la fonction d'activation est une fonction Meta, chaque neurone qui n'est ni sur la couche input, ni sur la première couche cachée s'écrit

$$\text{neurone} = \sigma \left(\sum_j a_j (w_j^T \text{hidden_layer}_{-2}(z) + b_j) \right)_+$$

et $\forall (a_j) > 0$, cette quantité est invariante par la transformation

$$\begin{cases} a_j \leftarrow a_j d_j \\ w_j \leftarrow w_j / d_j \\ b_j \leftarrow b_j / d_j \end{cases}$$

Il existe donc beaucoup de paramètres dont le même
réseau

(9)

\Rightarrow Paramètres \mapsto Mise en n'est pas injective

II. Optimisation et règle de la chaîne

(5)

$$F(\theta) = \frac{1}{n} \sum_{i=1}^n P(y_i; \rho_\theta(x_i)) + \Omega(\theta).$$

$$\text{d'où: } \nabla_{\theta} F(\theta) = \frac{1}{n} \sum_{i=1}^n \nabla_{\theta} P(y_i; \rho_\theta(x_i)) + \nabla_{\theta} \Omega(\theta).$$

Déscente de gradient:

• Commencer à θ_0

• Tant que (critère à vérifier pour continuer)

$$\theta_t \leftarrow \theta_{t-1} - \gamma_t \nabla_{\theta} F(\theta_{t-1})$$

↑
"Learning rate" à l'itération t .



Lorsque n est grand, calculer $\nabla_{\theta} F (= \frac{1}{n} \sum_{i=1}^n \dots)$ peut être trop cher, il est alors possible de remplacer $\nabla_{\theta} F$ par un estimateur calculé sur un sous-ensemble du jeu de données uniquement

Soit $B \subseteq \{1, \dots, n\}$

$$\nabla_{\theta}^{(B)} F(\theta) = \frac{1}{|B|} \sum_{b \in B} \nabla_{\theta} P(y_b; \rho_\theta(x_b)) + \nabla_{\theta} \Omega(\theta)$$

Déscent de gradient stochastique:

- Commencer à θ_0
- Tant que (critère à vérifier pour continuer)

• Sélectionner $B \subseteq \{1, \dots, n\}$ selon la règle choisie (aléa ou déterministe)

• $\theta_t \leftarrow \theta_{t-1} - \alpha_t \nabla_{\theta}^{(B)} f(\theta_{t-1})$

B s'appelle un "batch".

1) Algorithme de backpropagation



Comment calculer le gradient en les paramètres de

$f(y, h(z))?$

↑
fonction encodée par le réseau.

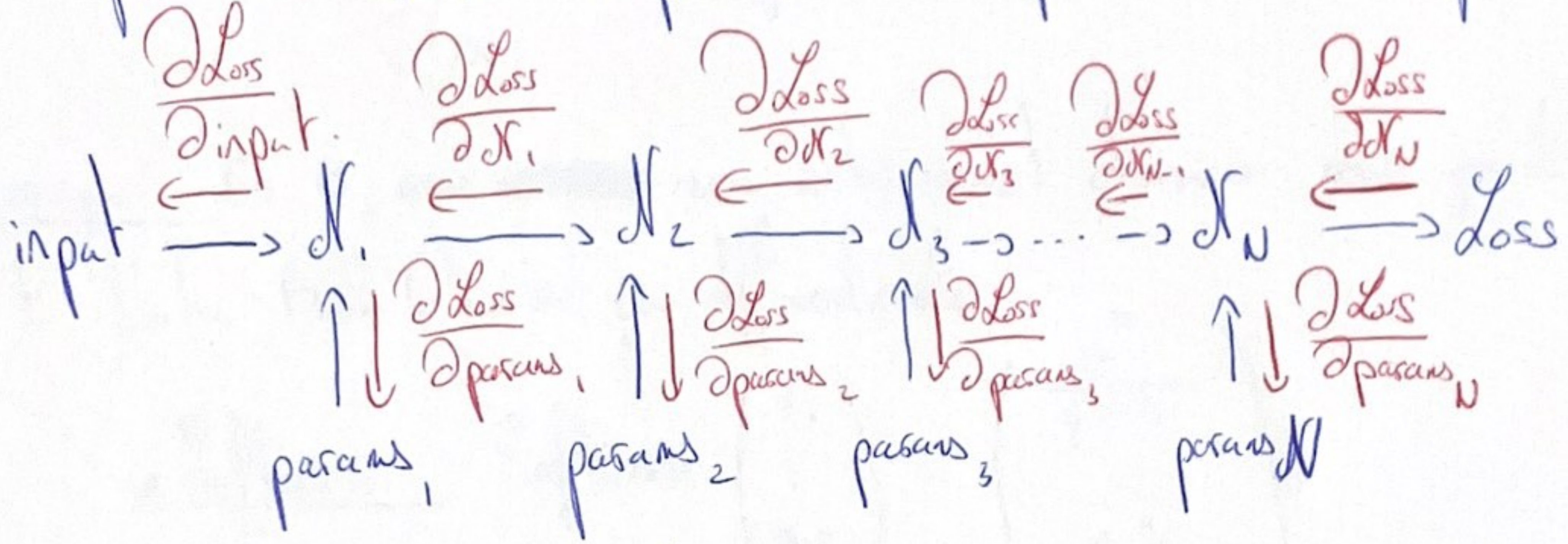
Solution: règle de la chaîne!

Si f et g sont différentiables, $d(g \circ f) = (dg \circ f) \circ df$

Ce qui donne, en notation informelle :

Si $N = N(\text{input}, \text{params})$, et $\text{Loss} = \text{Loss}(N)$,

(*) $\frac{\partial \text{Loss}}{\partial \text{params}} = \frac{\partial \text{Loss}}{\partial N} \frac{\partial N}{\partial \text{params}}$; $\frac{\partial \text{Loss}}{\partial \text{input}} = \frac{\partial \text{Loss}}{\partial N} \frac{\partial N}{\partial \text{input}}$ (**)



Algorithme Forward Backward

• Forward : Calculer $N(x, \theta)$ en gardant les activations dans chaque nœud

• Backward : En remontant à partir de la Loss, pour chaque nœud N

• Calculer $\frac{\partial \text{Loss}}{\partial \text{params}(N)}$ (avec *)

• Calculer $\frac{\partial \text{Loss}}{\partial \text{input}(N)}$ (avec (**))

2) Calculs des gradients à la main

(8)

• ReLU $\text{ReLU} \begin{pmatrix} x_1 \\ \vdots \\ x_c \end{pmatrix} = \begin{pmatrix} (x_1)_+ \\ \vdots \\ (x_c)_+ \end{pmatrix}$

$$\text{donc } \frac{\partial \text{ReLU}(x)_j}{\partial x_i} = \delta_{ij} \mathbb{1}_{x_i > 0}$$

Remarque: En \mathbb{B} , nous ~~avons~~ avons arbitrairement choisi un sous-gradient car la fonction $\text{ReLU}(\cdot)$ n'est pas différentiable.

• Softmax $\text{Softmax} \begin{pmatrix} x_1 \\ \vdots \\ x_p \end{pmatrix} = \begin{pmatrix} \frac{e^{x_1}}{\sum_j e^{x_j}} \\ \vdots \\ \frac{e^{x_p}}{\sum_j e^{x_j}} \end{pmatrix}$

$$\frac{\partial \text{Softmax}(x)_j}{\partial x_i} = \text{Softmax}(x)_j (\delta_{ij} - \text{Softmax}(x)_j)$$

• Affine: $\text{aff}(x) = Ax + b$

$$\frac{\partial \text{aff}(x)_j}{\partial x_i} = A_{ji}$$

$$\frac{\partial \text{aff}(x)_j}{\partial b_i} = 1$$

$$\frac{\partial \text{aff}(x)_j}{\partial A_{ij}} = x_j$$

III Approximation Universelle

Nous allons prouver que les réseaux de neurones ont la capacité d'approximer les fonctions continues aussi proche que l'on veut.

Mise en œuvre à une couche cachée

$$f(x) = \sum_{j=1}^m \eta_j (w_j^T x + b_j)_+$$

Étape 1: Les fonctions continues sont approchables par des fonctions continues affines par morceaux en 1-1.2.

Soit $g \in C^0([a, b], \mathbb{R})$.

Alors d'après le théorème de Heine, g est absolument continue.

en effet, sinon, $\exists \epsilon > 0 \forall \eta, \exists a_\eta, b_\eta \in [a, b]$ $\text{t.q. } |a_\eta - b_\eta| < \eta$ $\text{et } |g(a_\eta) - g(b_\eta)| \geq \epsilon$.

$\eta_n = \frac{1}{n}$ donne deux suites $(a_n)_{n \geq 1}$ et $(b_n)_{n \geq 1}$

Comme $[a, b]$ est compact, on extrait $a_{\varphi(n)} \rightarrow a_\infty \in [a, b]$

De même, on extrait $b_{\psi(n)} \rightarrow b_\infty \in [a, b]$

Alors $a_{\varphi(n)} \rightarrow a_\infty$; $b_{\psi(n)} \rightarrow b_\infty$

Comme $\forall n, |a_{\varphi(n)} - b_{\psi(n)}| \leq \frac{1}{\varphi(n)} \leq \frac{1}{n}$, $a_\infty = b_\infty$.

Par continuité, $|g(a_{\varphi(n)}) - g(b_{\psi(n)})| \xrightarrow{n \rightarrow +\infty} 0$ ce qui est absurde.

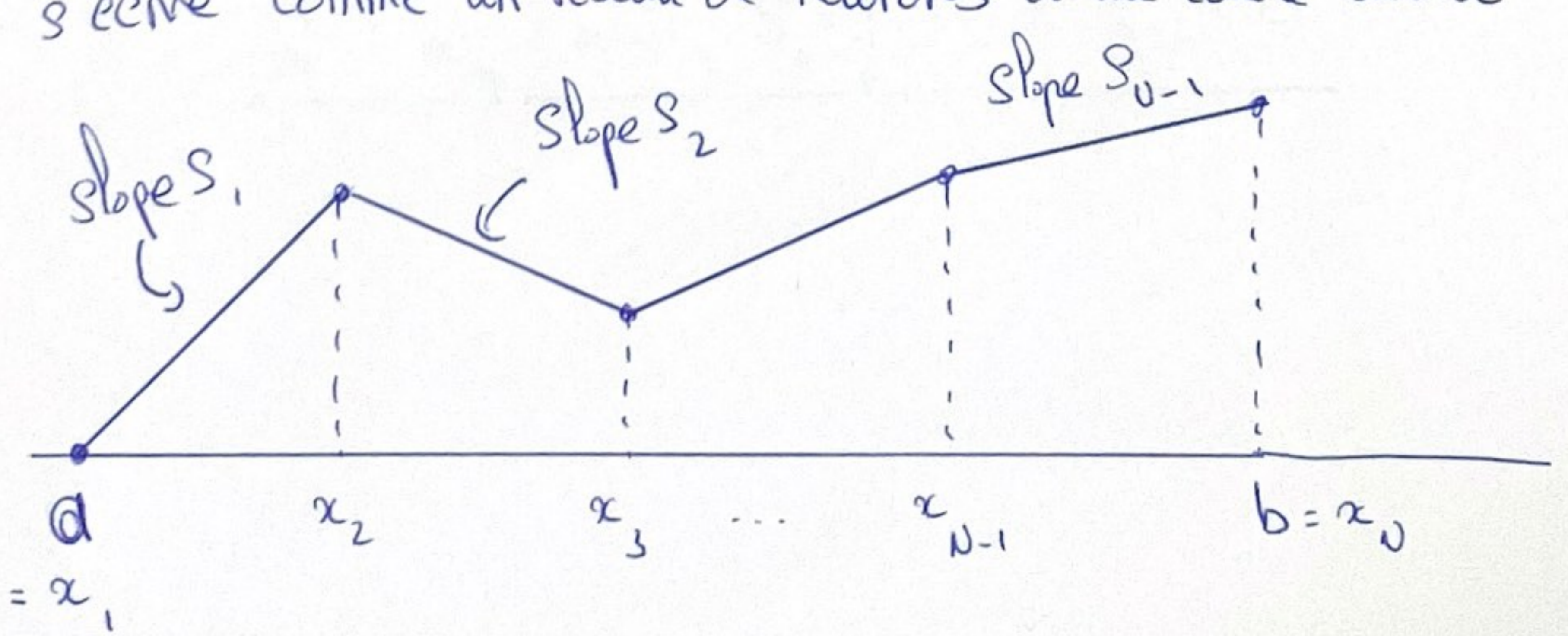
Comme g est absolument continue, $\forall \epsilon > 0$, il existe $\eta > 0$ tel que $|x, y| < \eta \Rightarrow |g(x) - g(y)| < \epsilon$.

Considérons la fonction (a=0 pour simplifier)

$$g_\eta(x) = \begin{cases} g(k\eta) + (x - k\eta) \frac{g((k+1)\eta) - g(k\eta)}{\eta} & \text{si } x \in [k\eta, (k+1)\eta) \\ g(k\eta) + (x - k\eta) \frac{g(b) - g(k\eta)}{b - k\eta} & \text{si } k\eta \in [a, b], (k+1)\eta > b, x \in [k\eta, b) \\ g(b) & \text{si } x = b \end{cases}$$

Alors g_η est continue et satisfait $\forall x \in [a, b] |g(x) - g_\eta(x)| \leq 2\epsilon$

Étape 2: Toutes les fonctions continues et affines par morceaux sur $[a, b]$ peuvent s'écrire comme un réseau de neurones à une couche cachée



Cas n° 1: $f(a) = 0$

$$f(x) = S_1(x - x_1)_+ \text{ sur } [x_1, x_2]$$

$$f(x) = S_1(x - x_1)_+ + (S_2 - S_1)(x - x_2)_+ \text{ sur } [x_1, x_3]$$

...

$$f(x) = \sum_{i \geq 1} (S_i - S_{i-1})(x - x_i)_+ \text{ sur } [x_i, x_{i+1}]$$

avec la convention $S_0 = 0$.

Cas général:

On voit f sur $[a, b]$ comme la restriction d'une fonction f' affine par morceaux et continue sur $[a-1, b]$.

$$f(x) = \sum_{i \geq 0} (S_i - S_{i-1})(x - x_i)_+$$

où $x_0 = a-1$, $S_0 = f(a)$ et $S_{-1} = 0$