UNIVERSITÉ
DE GENÈVE

Projet de Bachelor d'informatique

Graphes computationnels et différentiation automatique

Giulio GANCI

Année 2024-2025

Table des matières

1 Introduction

L'apprentissage automatique (*Machine Learning*, ML) a révolutionné de nombreux domaines, tels que la vision par ordinateur, le traitement du langage naturel et la robotique. Au cœur des algorithmes de ML, notamment ceux basés sur l'apprentissage supervisé comme les réseaux de neurones, se trouve un processus clé : l'**optimisation**.

Pour optimiser les paramètres d'un modèle, les méthodes de descente de gradient (et ses variantes, comme la rétropropagation) nécessitent le calcul précis des **dérivées** de la fonction de coût. Cependant, calculer ces dérivées manuellement pour des modèles complexes devient rapidement impraticable. La **différentiation automatique** (Automatic Differentiation, AD) offre une solution élégante en calculant les dérivées de manière algorithmique via des **graphes computationnels**.

Dans ce projet, j'implémente un système de différentiation automatique basé sur les graphes computationnels, permettant :

— La représentation d'expressions mathématiques sous forme de graphes
— Le calcul automatique des dérivées partielles de fonctions arbitraires (saisies sous forme de chaînes de caractères)

Ce projet explore ainsi les fondements mathématiques de la AD tout en fournissant une implémentation opérationnelle.

2 Théorie

2.1 Introduction

Représenter un calcul sous la forme d'un graphe peut sembler, à première vue, une complication inutile. Après tout, pourquoi s'encombrer d'une structure de graphe pour évaluer une expression, alors qu'un simple calcul direct suffirait en théorie ?

Pourtant, cette représentation graphique révèle toute sa puissance lorsqu'il s'agit non seulement de calculer la valeur d'une expression, mais surtout d'en optimiser le calcul des dérivées. C'est ici qu'intervient la **différenciation automatique**, une technique bien plus efficace pour calculer la dérivée en un point précis que les méthodes symboliques ou numériques traditionnelles.

Le graphe computationnel, en structurant les dépendances entre les opérations, permet :

— De décomposer un calcul complexe en une séquence d'opérations élémentaires,
— D'appliquer systématiquement la règle de la chaîne pour le calcul des gradients,
— D'optimiser l'ordre des calculs pour une évaluation rapide des dérivées.

Ainsi, ce qui pourrait paraître comme une abstraction superflue devient en réalité un outil indispensable pour l'apprentissage automatique, la physique numérique et tout domaine nécessitant des gradients précis et efficaces.

2.2 Graphes computationnels

Définition formelle d'un graphe computationnel

Un **graphe computationnel** est un graphe orienté acyclique noté $G = (V, E)$, où :

— V est un ensemble de **nœuds** représentant :
  — des **variables** (entrées, sorties ou intermédiaires),
  — des **opérations** (fonctions mathématiques),
  — ou des **constantes**.
— $E \subseteq V \times V$ est un ensemble d'**arêtes orientées** représentant les dépendances de calcul. Une arête $(u, v) \in E$ signifie que le résultat de u est utilisé comme entrée pour v. Les noeuds racines représentent les variables et constantes, et le noeud feuille représente le résultat final de l'expression.

2.2.1 Exemple

Considérons le calcul $y = (x_1 + x_2) \times x_3 + 5$. Son graphe computationnel associé est :

$$G = (V, E)$$

avec :

$$V = \{x_1, x_2, x_3, +_1, +_2, \times, 5, y\}$$

$$E = \{(x_1, +_1), (x_2, +_1), (+_1, \times), (x_3, \times), (\times, +_2), (+_2, y)\}$$

On remarque que si une opération revient plusieurs fois, il est nécessaire de la numéroter afin d'en distinguer les instances.

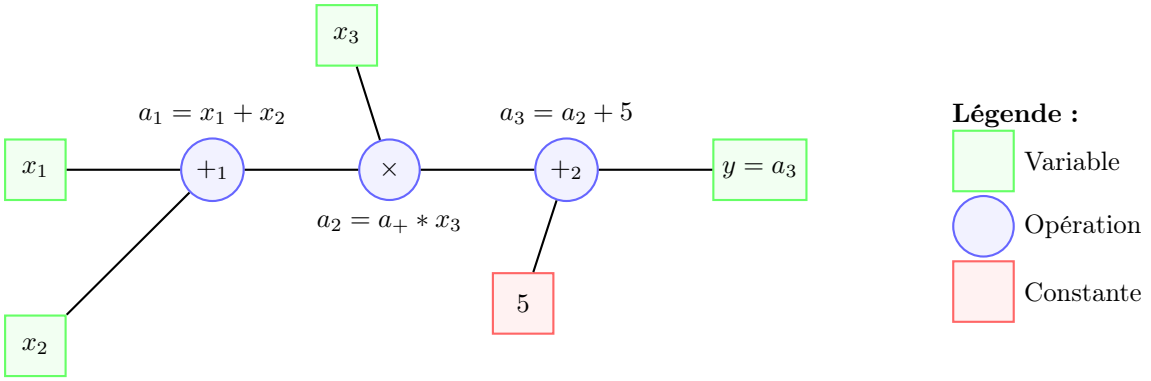


FIGURE 1 – Graphe computationnel de $y = (x_1 + x_2) \times x_3 + 5$ (orientation gauche-droite)

2.2.2 Fonction d'évaluation

Soit $G = (V, E)$ un graphe computationnel

On peut définir la fonction `eval`(x) qui prend en entrée un nœud $x \in V$ et retourne sa valeur calculée en propageant les dépendances du graphe. Formellement :

$$\texttt{eval}(x) = \begin{cases} c & \text{si } x \text{ est une constante de valeur } c, \\ v & \text{si } x \text{ est une variable d'entrée avec } v \text{ sa valeur donnée}, \\ f(\texttt{eval}(u_1), \ldots, \texttt{eval}(u_n)) & \text{si } x \text{ est une opération } f \text{ avec parents } u_1, \ldots, u_n. \end{cases}$$

2.3 Différentiation automatique par rétropropagation

Principe général

La différentiation automatique calcule les dérivées en décomposant le calcul en opérations élémentaires et en appliquant systématiquement la **règle de la chaîne**.

Mécanisme des gradients

Chaque nœud a_i du graphe contient :
- `value` : résultat du calcul direct (*forward pass*)
- `grad` : $\frac{\partial f}{\partial a_i}$ calculé lors de la *backpropagation*, où f est l'expression à dériver (souvent l'erreur qu'on souhaite minimiser)

Algorithme en 2 phases

1. **Forward pass** : Évaluer le graphe en propageant les valeurs des variables vers la sortie
2. **Backward pass** : Calculer les gradients en partant de la sortie vers les entrées

2.3.1 Forward Pass

Pour une fonction y=f(x), on calcule la valeur de chacun des noeuds du graphe computationnel correspondant, jusqu'à atteindre le noeud y. On part avec des valeurs attribuées à chaque variable x_i à l'aide de eval(x_i), pour "propager vers l'avant" l'information.

Voici une visualisation du processus :

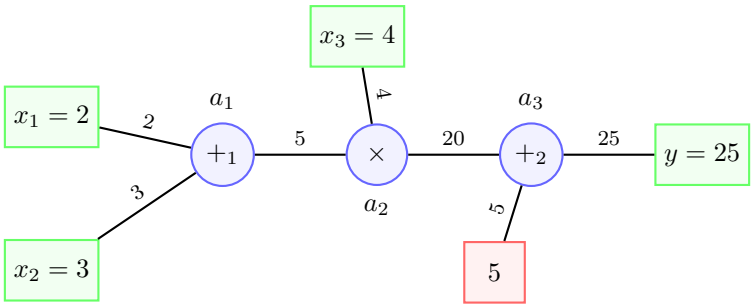


FIGURE 2 – Propagation des valeurs

Cela nous donne :

TABLE 1 – Calcul des valeurs par propagation

Nœud	**Valeur**
y	25
a_3	25
a_2	20
5	5
a_1	5
x_3	4
x_1	2
x_2	3

Nous avons désormais les valeurs de chaque noeud, qui permettront de calculer chaque dérivé partielle lors de la prochaine étape.

2.3.2 Backward Pass

Le backward pass consiste à calculer les dérivées partielles de manière récursive en partant de la sortie vers les entrées, en utilisant la règle de dérivation en chaîne. Pour chaque nœud a_i du graphe computationnel :

$$\frac{\partial f}{\partial a_i} = \sum_{a_j \in \text{Enfants}(a_i)} \frac{\partial f}{\partial a_j} \cdot \frac{\partial a_j}{\partial a_i} \tag{1}$$

où f est la fonction finale représentée par le graphe computationnel.

Dans le contexte spécifique du machine learning, f correspond généralement à la fonction de perte $\mathcal{L}$ qu'on cherche à minimiser. La rétropropagation calcule alors $\nabla\mathcal{L}$ par rapport aux paramètres du modèle.

Le processus se déroule en 3 étapes :

1. Initialiser avec $\frac{\partial f}{\partial a_{\text{final}}} = 1$ (si $f = a_{\text{final}}$, cas général)
2. Pour chaque nœud a_i dans l'ordre inverse du forward pass :
   — Calculer $\frac{\partial a_j}{\partial a_i}$ pour chaque enfant a_j
   — Accumuler les gradients : $\frac{\partial f}{\partial a_i} += \frac{\partial f}{\partial a_j} \cdot \frac{\partial a_j}{\partial a_i}$
3. Répéter jusqu'à atteindre les variables d'entrée

Calcul des dérivées locales

Le backward pass repose sur une décomposition récursive du gradient via la règle de chaîne. Pour l'implémenter efficacement, chaque opération du graphe computationnel doit fournir son gradient local, c'est-à-dire le jacobien de sa sortie par rapport à ses entrées.

Soit $a_j = op(a_i)$ un nœud enfant de a_i, où op est une opération élémentaire. La dérivée partielle $\frac{\partial a_j}{\partial a_i}$ se calcule selon la nature de l'opération :

— **Opérations élémentaires** : Pour les opérations arithmétiques de base (+, -, ×, /), les fonctions usuelles (exp, log, sin, cos, etc.) ou les opérations tensorielles (produit matriciel, convolution), les gradients locaux sont bien connus et se calculent analytiquement.

— **Opérations composées** : Certains nœuds peuvent représenter des opérations plus complexes (softmax, couches de réseau de neurones, etc.), mais leur gradient peut toujours se décomposer en sous-opérations différentiables.

La "simplicité" des opérations dépend en réalité du niveau d'abstraction choisi. Une opération considérée comme élémentaire à haut niveau (comme une convolution) peut en réalité cacher des centaines d'opérations sous-jacentes. En pratique, les frameworks de différenciation automatique implémentent ces gradients locaux via :

— Des formules analytiques pré-codées pour les opérations de base
— Des règles de décomposition automatique pour les opérations composées
— Des approximations numériques pour les cas pathologiques

Exemple détaillé du backward pass

Appliquons ce mécanisme à $f(x_1, x_2, x_3) = (x_1 + x_2) \times x_3 + 5$ (cf. Fig. 1) calculons $\frac{\partial f}{\partial x_i}\Big|_{(x_1,x_2,x_3)=(2,3,4)}$:

Étape 1 : Initialisation du gradient

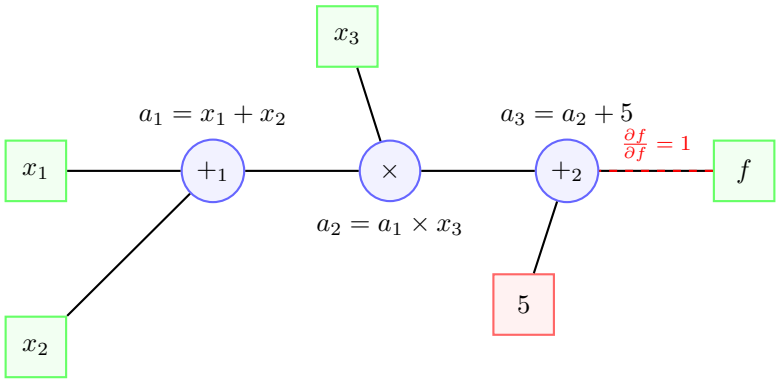


FIGURE 3 – Backward pass - Étape 1 : Initialisation du gradient de la sortie à 1

Étape 2 : Propagation vers a_2 et 5

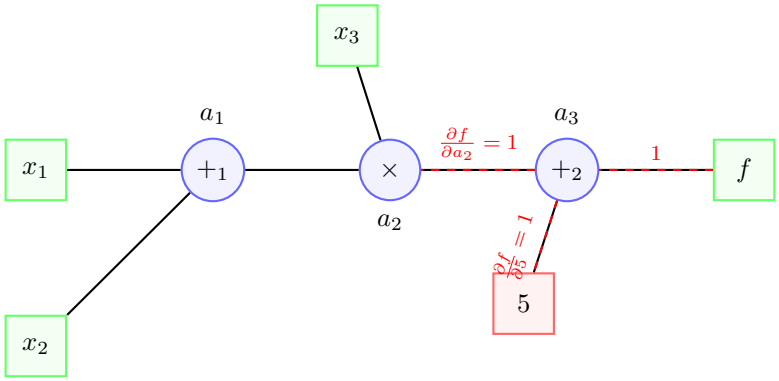


FIGURE 4 – Backward pass - Étape 2 : Calcul des gradients de a_2 et 5

Étape 3 : Propagation vers a_1 et x_3

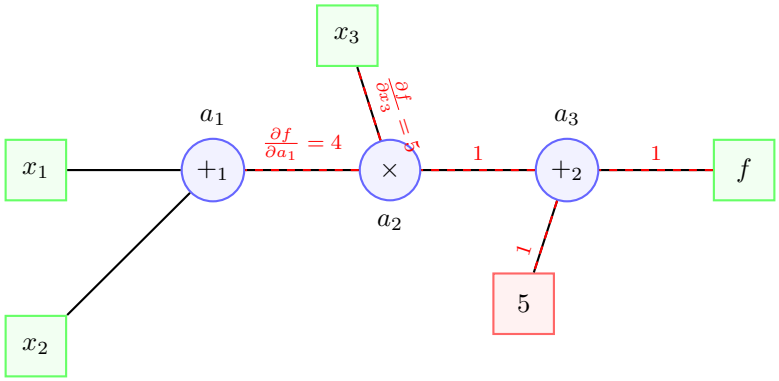


FIGURE 5 – Backward pass - Étape 3 : Calcul des gradients de a_1 et x_3

Étape 4 : Propagation finale vers x_1 et x_2

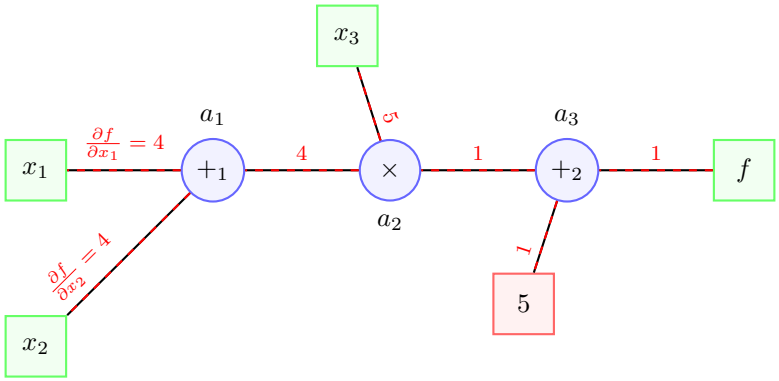


FIGURE 6 – Backward pass - Étape 4 : Calcul des gradients finaux des variables d'entrée

TABLE 2 – Calcul des gradients par rétropropagation

Nœud	**Valeur**	**Gradient**	**Dérivée locale**	**Calcul**
$a_3 = f$	25	$\frac{\partial f}{\partial a_3} = 1$	$\frac{\partial a_3}{\partial a_3} = 1$	Initialisation
a_2	20	$\frac{\partial f}{\partial a_2} = 1$	$\frac{\partial a_3}{\partial a_2} = 1$	$\frac{\partial f}{\partial a_3} \times \frac{\partial a_3}{\partial a_2} = 1 \times 1$
5	5	$\frac{\partial f}{\partial 5} = 1$	$\frac{\partial a_3}{\partial 5} = 1$	$\frac{\partial f}{\partial a_3} \times \frac{\partial a_3}{\partial 5} = 1 \times 1$
a_1	5	$\frac{\partial f}{\partial a_1} = 4$	$\frac{\partial a_2}{\partial a_1} = x_3 = 4$	$\frac{\partial f}{\partial a_2} \times \frac{\partial a_2}{\partial a_1} = 1 \times 4$
x_3	4	$\frac{\partial f}{\partial x_3} = 5$	$\frac{\partial a_2}{\partial x_3} = a_1 = 5$	$\frac{\partial f}{\partial a_2} \times \frac{\partial a_2}{\partial x_3} = 1 \times 5$
x_1	2	$\frac{\partial f}{\partial x_1} = 4$	$\frac{\partial a_1}{\partial x_1} = 1$	$\frac{\partial f}{\partial a_1} \times \frac{\partial a_1}{\partial x_1} = 4 \times 1$
x_2	3	$\frac{\partial f}{\partial x_2} = 4$	$\frac{\partial a_1}{\partial x_2} = 1$	$\frac{\partial f}{\partial a_1} \times \frac{\partial a_1}{\partial x_2} = 4 \times 1$

3 Implémentation

3.1 Guide

Le code permet de calculer les dérivées partielles de toute fonction par rapport à ses variables. Le système offre également la possibilité de visualiser le graphe de la fonction.

3.1.1 Utilisation

Deux méthodes sont possibles :

Méthode 1 : via `main.py`

1. Modifier la fonction dans le fichier `main.py` en éditant la variable `fonction`
2. Exécuter le script `main.py`
3. Entrer les valeurs des variables pour lesquelles on souhaite calculer les dérivées
4. Le système retourne les dérivées partielles par rapport à chaque variable
5. Afficher le graphe si souhaité

Méthode 2 : via `script.sh`

1. Modifier la fonction dans `config.cfg` **ou** passer la fonction en argument avec `-f`
2. Exécuter le script `script.sh` avec les options désirées :
   — `./script.sh` — exécute avec la fonction par défaut dans `config.cfg`
   — `./script.sh -f "x^2 + y"` — exécute avec une fonction donnée
   — `./script.sh -g` — affiche également le graphe de calcul
   — `./script.sh -v` — mode verbeux
   — `./script.sh -c autre.cfg` — spécifie un fichier de configuration alternatif

Restrictions syntaxiques

— **Opérations supportées** :
   — Addition : $a + b$
   — Soustraction : $a - b$
   — Multiplication : $a * b$
   — Division : a/b
   — Puissance : a^b (écrit `a^b` dans le code)
— **Fonctions élémentaires supportées** :

$$\sin(a), \cos(a), \text{sqrt}(a), \exp(a), \ln(a), \text{neg}(a)$$

Notez que le code ne supporte pas la négation directe (par exemple dans `(`$-y + x$`)`), il faut utiliser `neg(`y`)` : `(neg(`y`) + `x`)`.

Priorité des opérateurs

Lors de l'analyse des expressions, chaque opérateur est associé à un niveau de priorité :
— Priorité 0 (implicite) : fonctions élémentaires (`sin`, `exp`, `neg`)
— Priorité 1 : parenthèses `( )`
— Priorité 2 : addition `+`, soustraction `-`
— Priorité 3 : multiplication `*`, division `/`
— Priorité 4 : puissance `^`

Grammaire formelle

La grammaire acceptée par le système est définie comme suit :

$$\begin{aligned} S &\to S + S \mid S - S \mid S * S \mid S/S \mid \texttt{S ^ S} \mid \text{op}(S) \mid \text{var} \mid \text{const} \\ \text{op} &\in \{\sin, \cos, \text{sqrt}, \exp, \ln\} \\ \text{var} &:= \text{chaîne de caractères (nom de variable)} \\ \text{const} &:= \text{nombre réel} \end{aligned}$$

Note : Certaines écritures mathématiques ne sont pas prises en charge. Veuillez vous référer à la grammaire ci-dessus pour une utilisation optimale.

3.1.2 Structure du code

Le projet est organisé en différents modules :

`fonctions.py` — Implémente les opérations mathématiques supportées
- — Contient les règles de dérivation pour chaque opération
- — Définit les fonctions élémentaires et leurs dérivées

`Graphe.py` — Classe principale `Graphe` représentant le graphe de calcul
- — Méthodes pour :
  - — Construire la représentation graphique de la fonction
  - — Calculer les dérivées partielles
  - — Parcourir la structure du graphe

`main.py` — Point d'entrée du programme
- — Interface utilisateur pour :
  - — Spécifier la fonction à analyser
  - — Entrer les valeurs des variables
  - — Afficher les résultats des dérivées
  - — Afficher le graphe computationnel correspondant
- — Utilise les fonctionnalités de `Graphe.py`

`script.sh` — Script shell permettant d'automatiser l'exécution du calcul
- — Gère les options : fonction personnalisée, affichage du graphe, configuration externe, etc.
- — Crée un script Python temporaire et exécute le traitement

`config.cfg` — Fichier de configuration contenant la fonction par défaut à analyser
- — Peut être modifié manuellement ou remplacé via l'option `-c`

`Tests.py` — Teste les fonctions de `Graphe.py`

`Tableau.py` — Affiche les performances de `Graphe.py`

3.2 Fonctionnement du code

3.2.1 Représentation des nœuds des graphes

Chaque nœud contient :
- — Une valeur `value` représentant la valeur prise par le noeud
- — Un gradient `grad` (initialisé à 0) représentant le gradient de la fonction par rapport au noeud actuel
- — Une fonction d'opération `op` correspondant à l'opération du noeud (sauf pour les variables)
- — Une liste de parents `parents` correspondant aux noeuds parents du noeud actuel.

Le pseudo-code ci-dessous illustre la procédure d'initialisation du graphe de calcul.

Algorithm 1 Initialisation du graphe

procedure ADD_VARIABLE(name, value)
    Définir un nouveau nœud **v** tel que :
- — `v.value` ← `value`
- — `v.grad` ← 0.0
- — `v.op` ← `None`
- — `v.parents` ← `[]`

    nodes[name] ← **v**
end procedure

procedure ADD_OPERATION(name, op, parents, grad_fn)
    Pour chaque parent **p** dans `parents`, récupérer la valeur `p.value`.
    Calculer `result` ← `op`({p.value pour chaque p dans parents}).
    Définir un nouveau nœud **n** avec :
- — `n.value` ← `result`
- — `n.grad` ← 0.0
- — `n.op` ← `grad_fn`
- — `n.parents` ← `parents`

    nodes[name] ← **n**
end procedure

3.2.2 Création du graphe computationnel

Pour construire dynamiquement un graphe de calcul à partir d'une expression mathématique, nous utilisons un parseur associé à la notation polonaise inverse (postfixée, **Annexe A**). Cette représentation permet de traiter l'expression sous forme de pile, facilitant l'analyse syntaxique et la génération du graphe.

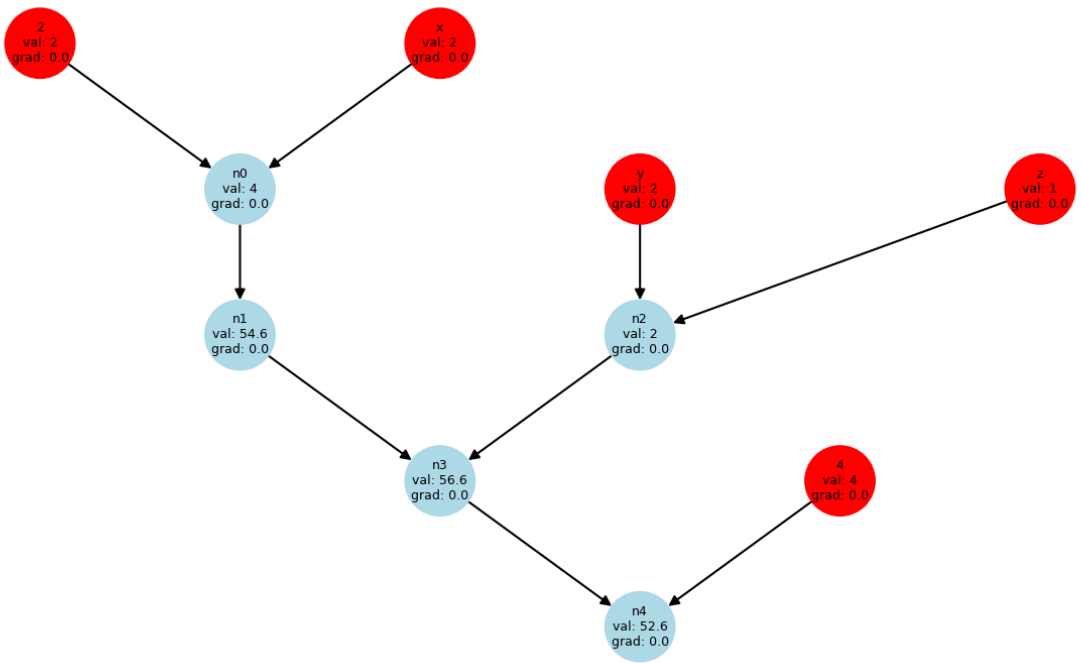
Ce processus constitue la phase de **forward pass** défini formellement plus tôt, durant laquelle les valeurs des nœuds sont calculées à partir des entrées et des opérations successives.

Lors de l'appel à la méthode `create_graph`, une chaîne de caractères telle que `"exp(2*x)+y*z-4"` est convertie en une liste postfixée : `['2', 'x', '*', 'exp', 'y', 'z', '*', '+', '4', '-']`. Cette liste est ensuite traitée pour créer les nœuds correspondants en appelant les fonctions `add_variable` et `add_operation` précédemment définies.

Algorithm 2 Création du graphe à partir d'une expression postfixée

```
procedure CREATE_GRAPH(postfix_expression, valeurs)
    Initialiser une pile vide
    for chaque token dans l'expression postfixée do
        if token est une fonction unaire (ex. sin, exp) then
            Extraire l'opérande depuis la pile
            Créer un nœud d'opération unaire avec cette opérande
        else if token est un opérateur binaire (ex. +, *, -) then
            Extraire deux opérandes depuis la pile
            Créer un nœud d'opération binaire avec ces opérandes
        else
            token est une variable ou une constante
            if non déjà définie then
                Ajouter un nœud variable avec sa valeur correspondante
            end if
        end if
        Empiler le nom du nouveau nœud
    end for
    Définir l'indice du dernier nœud créé
end procedure
```

Une fois le graphe initialisé, voici sa structure pour la fonction `exp(2*x) + y * z - 4` :



On constate que tous les gradients sont initialisés à 0. Il faut attendre l'étape de la backpropagation pour avoir des gradients cohérents.

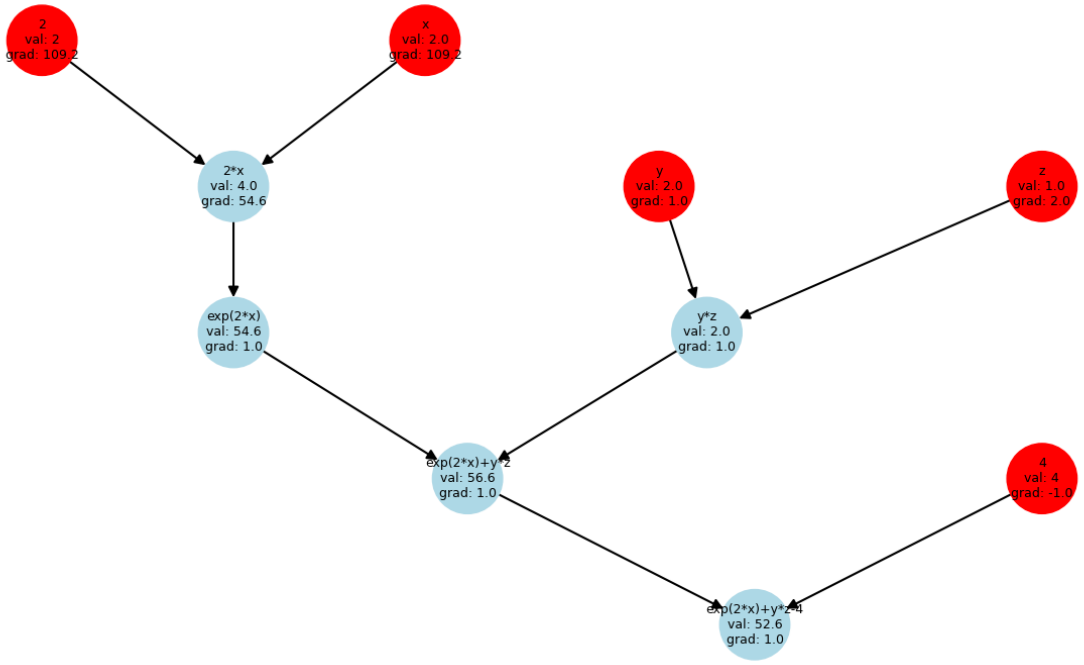
3.2.3 Backpropagation/Backward Pass

Ce processus correspond au backward pass défini plus tôt. Pour rappel, il propage à reculons les dérivées locales et utilise la regle de chainage pour calculer les derivées globales.

Algorithm 3 Backpropagation du gradient

```
target ← "a_max"                                   ▷ Par défaut, le noeud max(la fonction).
nodes[target].grad ← 1.0                           ▷ Le gradient du nœud cible est 1 par définition
sorted_nodes ← liste des nœuds triée en ordre inverse
for name dans sorted_nodes do                      ▷ Parcours inverse des nœuds
    node ← nodes[name]
    if node.op existe then                         ▷ Si le nœud a des parents, calculer les gradients
        parent_grads ← node.op(nodes[parent].value) ∀ parent
        for p, grad dans zip(node.parents, parent_grads) do
            nodes[p].grad ← nodes[p].grad + node.grad × grad      ▷ Règle de chainage
        end for
    end if
end for
```

Après la backpropagation des gradients, on obtient ceci :



Comparons les résultats obtenus avec les résultats calculés à la main :
Soit la fonction :

$$f(x, y, z) = e^{2x} + y \cdot z - 4$$

Évaluons les dérivées au point $(x, y, z) = (2, 2, 1)$:

3.2.4 Dérivée par rapport à x

$$\begin{aligned}\frac{\partial f}{\partial x} &= \frac{\partial}{\partial x}\left[e^{2x}\right] + \frac{\partial}{\partial x}\left[y \cdot z\right] + \frac{\partial}{\partial x}\left[-4\right] \\ &= e^{2x} \cdot \frac{\partial}{\partial x}\left[2x\right] + 0 + 0\end{aligned}$$

$$= 2e^{2x}$$

$$\left.\frac{\partial f}{\partial x}\right|_{(2,2,1)} = 2e^4 \approx \boxed{109.1963}$$

3.2.5 Dérivée par rapport à y

$$\frac{\partial f}{\partial y} = \frac{\partial}{\partial y}\left[e^{2x}\right] + \frac{\partial}{\partial y}\left[y \cdot z\right] + \frac{\partial}{\partial y}\left[-4\right]$$
$$= 0 + z + 0$$
$$= z$$
$$\left.\frac{\partial f}{\partial y}\right|_{(2,2,1)} = 1 = \boxed{1}$$

3.2.6 Dérivée par rapport à z

$$\frac{\partial f}{\partial z} = \frac{\partial}{\partial z}\left[e^{2x}\right] + \frac{\partial}{\partial z}\left[y \cdot z\right] + \frac{\partial}{\partial z}\left[-4\right]$$
$$= 0 + y + 0$$
$$= y$$
$$\left.\frac{\partial f}{\partial z}\right|_{(2,2,1)} = 2 = \boxed{2}$$

Les résultats calculés correspondent bien aux résultats obtenus grâce à la différentiation automatique.

4 Résultats de l'implémentation

4.1 Véracité des dérivées calculées

Voici un tableau montrant la différence entre les dérivées calculées et celle exact pour plusieurs fonctions :

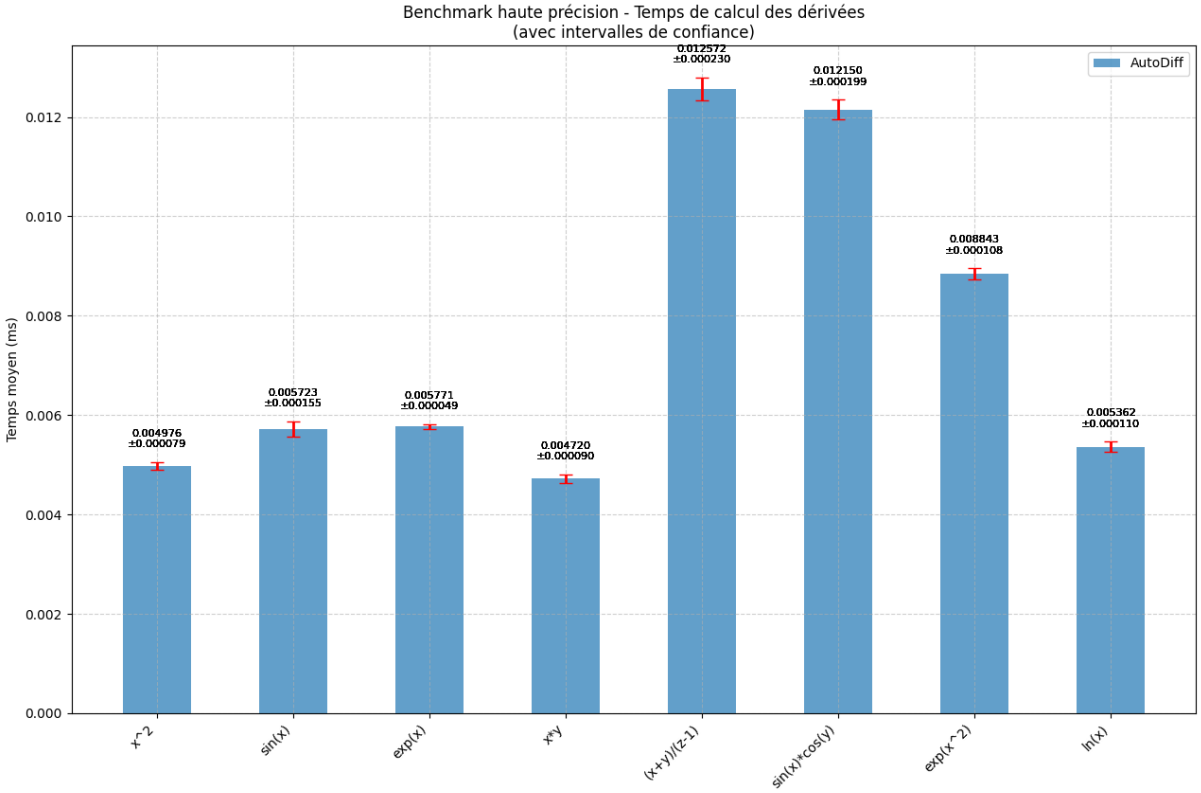
Fonction	**Dérivée**	**Variable**	**Valeur AD**	**Valeur réelle**	**Erreur relative**
x^2	$2x$	$x = 2$	4.000000	4.000000	0
$\sin(x)$	$\cos(x)$	$x = \pi/4$	0.707107	0.707107	0
$\exp(x)$	$\exp(x)$	$x = 1$	2.71828	2.71828	0
$x \cdot y$	y	$x = 3, y = 4$	4.000000	4.000000	0
$x \cdot y$	x	$x = 3, y = 4$	3.000000	3.000000	0
$x + y$	1	$x = 5, y = 7$	1.000000	1.000000	0
$x + y$	1	$x = 5, y = 7$	1.000000	1.000000	0
$\frac{x+y}{z-1}$	$\frac{1}{z-1}$	$x = 2, y = 3, z = 4$	0.333333	0.333333	0
$\frac{x+y}{z-1}$	$\frac{1}{z-1}$	$x = 2, y = 3, z = 4$	0.333333	0.333333	0
$\frac{x+y}{z-1}$	$-\frac{x+y}{(z-1)^2}$	$x = 2, y = 3, z = 4$	-0.555556	-0.555556	0
$\sin(x)\cos(y)$	$\cos(x)\cos(y)$	$x = \pi/4, y = \pi/3$	0.353553	0.353553	0
$\sin(x)\cos(y)$	$-\sin(x)\sin(y)$	$x = \pi/4, y = \pi/3$	-0.612372	-0.612372	0
$\exp(x^2)$	$2x\exp(x^2)$	$x = 1$	5.43656	5.43656	0
$\ln(x)$	$\frac{1}{x}$	$x = 2$	0.500000	0.500000	0

TABLE 3 – Comparaison des dérivées obtenues par différentiation automatique (AD) et des valeurs réelles, avec les expressions des dérivées et les valeurs des variables utilisées

On constate que pour toutes les fonctions testées, les dérivées calculées sont justes.

4.2 Complexité de calcul pour plusieurs fonctions

Voici la différence des vitesse d'execution entre différentes fonctions :



5 Conclusion

Ce projet a permis d'explorer en profondeur les mécanismes fondamentaux de la différentiation automatique, pierre angulaire des systèmes modernes d'apprentissage automatique (**Annexe B**). L'implémentation d'un graphe computationnel fonctionnel a démontré :

— La puissance de la règle de chaîne appliquée de manière systématique à travers un graphe de calcul
— L'élégance mathématique derrière les algorithmes de rétropropagation
— L'efficacité pratique de cette approche pour le calcul précis de dérivées complexes

Les applications potentielles de ce travail s'étendent bien au-delà du cadre académique. Un système de différentiation automatique robuste ouvre la porte à :

— L'optimisation de paramètres dans des modèles de machine learning
— La résolution numérique d'équations différentielles
— L'analyse de sensibilité dans des simulations scientifiques

En pratique, plusieurs bibliothèques implémentent efficacement ces concepts :

— **PyTorch** et **TensorFlow** : Les frameworks dominants en deep learning, utilisant des graphes computationnels dynamiques/statiques
— **JAX** : Combinaison puissante de différentiation automatique et de transformations de fonctions
— **Autograd** (prédécesseur de JAX) : Différentiation automatique pure en Python/Numpy
— **CasADi** : Spécialisé dans l'optimisation non-linéaire et le contrôle optimal

Ce projet souligne également l'importance cruciale des structures de données appropriées (ici, les graphes computationnels) pour résoudre efficacement des problèmes mathématiques complexes. La différentiation automatique, en combinant élégance théorique et efficacité pratique, représente un parfait exemple de symbiose entre les mathématiques et l'informatique.

A Notation postfixe (polonaise inverse) et graphes computationnels

A.1 Notation Polonaise Inverse (NPI)

- Définition : forme postfixée où l'opérateur suit ses opérandes
- Propriétés clés :
  - Suppression des parenthèses (priorité implicite)
  - Ordre d'évaluation univoque
  - Implémentation naturelle avec pile LIFO
- Exemple avec pile :
  - Expression : $\sin(x^2) + \frac{y}{3}$
  - NPI : `x 2 ^ sin y 3 / +`

A.2 Arbre syntaxique (Représentation intermédiaire)

- Structure arborescente des expressions :
  - Feuilles : opérandes (constantes/variables)
  - Nœuds internes : opérateurs
  - Priorité opératoire préservée
- Construction algorithmique :
  1. Analyse lexicale → tokens
  2. Parsing → hiérarchie des opérations
  3. Résolution des parenthèses
- Exemple : $\sin(x^2) + \frac{y}{3}$

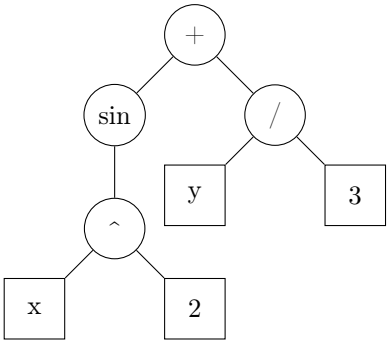


FIGURE 7 – Arbre syntaxique annoté

A.3 Lien NPI ↔ Arbre syntaxique

- Correspondance biunivoque :
  - NPI = parcours post-ordre de l'arbre
  - Reconstruction unique possible
- Algorithme de conversion :

```
1: function TreeToNPI(node :root)
2:     if node.left ≠ null then TreeToNPI(node.left)
3:     end if
4:     if node.right ≠ null then TreeToNPI(node.right)
5:     end if
6:     output(node.value)
7: end function
```

- Illustration :

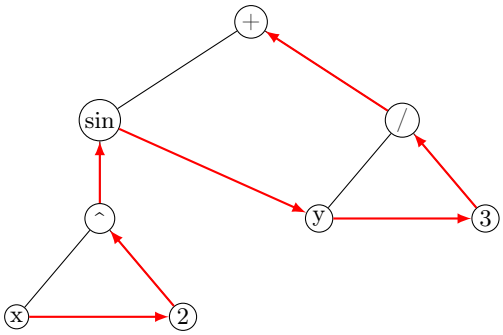


FIGURE 8 – Parcours post-ordre générant la NPI

A.4 Conclusion : Unification des représentations

— Correspondance exacte :

$$\text{NPI} \rightleftharpoons \text{Arbre syntaxique} \rightleftharpoons \text{Graphe computationnel}$$

— Graphe = Arbre syntaxique + Sémantique :
  — Assignation des valeurs aux feuilles
  — Mécanisme de propagation des valeurs
— Exemple unifié :

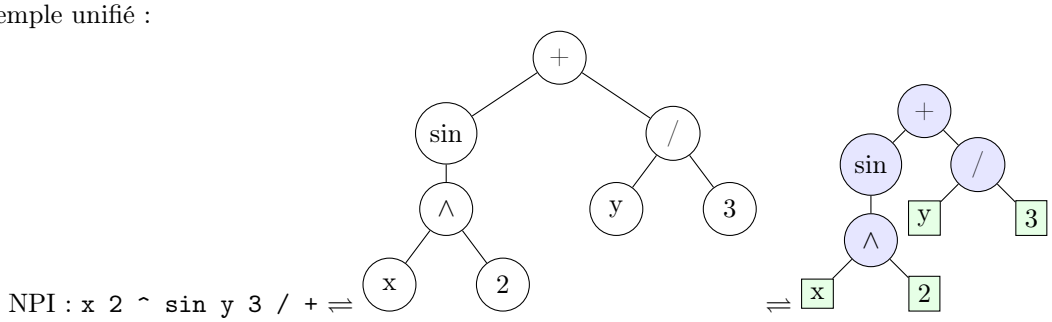


FIGURE 9 – Équivalence entre NPI, arbre syntaxique et graphe pour l'expression $\sin(x^2) + y/3$

— Propriété fondamentale :

Théorème. Pour toute expression mathématique bien formée, il existe :

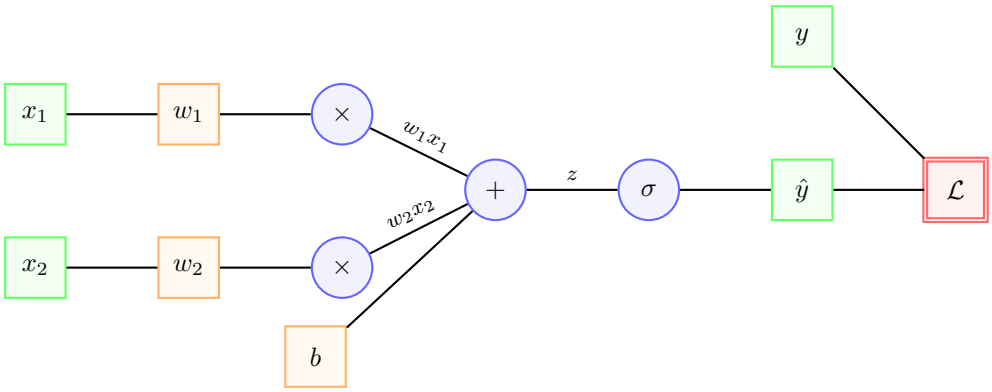
1. Une unique NPI (à l'ordre des opérandes près)
2. Un seul arbre syntaxique canonique
3. Un graphe équivalent

B Lien avec les réseaux de neurones

Les réseaux de neurones artificiels peuvent être interprétés comme des graphes computationnels particuliers où les opérations sont typiquement des combinaisons linéaires suivies de fonctions d'activation non-linéaires. Cette perspective permet d'appliquer directement les mécanismes de différentiation automatique étudiés précédemment.

B.1 Analogie graphe computationnel et réseau de neurone

Un réseau de neurones peut être modélisé comme un graphe computationnel dirigé où :
— Les nœuds feuilles représentent les données d'entrée ($\mathbf{x}$) et les paramètres apprenables ($\mathbf{W}$, $\mathbf{b}$)
— Les nœuds internes correspondent aux opérations (produit matriciel, fonctions d'activation, etc.)
— Le nœud final représente la fonction de perte $\mathcal{L}(\mathbf{y}, \hat{\mathbf{y}})$



Légende :
$z = w_1x_1 + w_2x_2 + b \quad \hat{y} = \sigma(z)$
$\mathcal{L}$: fonction de perte (comparaison entre $\hat{y}$ et y)

FIGURE 10 – Graphe computationnel simplifié d'un neurone artificiel

Ce graphe computationnel peut être plus simplement et généralement représenté comme ci dessous :

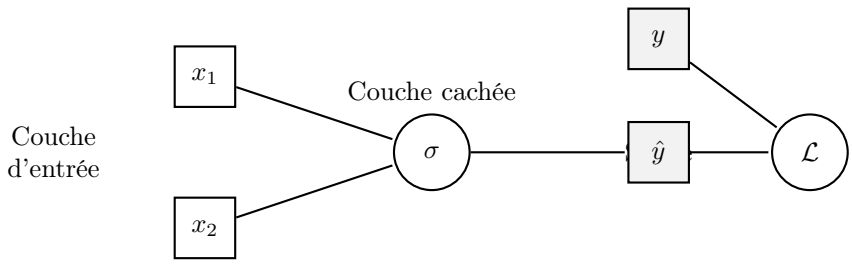


FIGURE 11 – Représentation classique d'un perceptron simple

B.2 Calcul des gradients dans un réseau de neurones

La rétropropagation est une application des règles de chaînage dans les graphes computationnels. Considérons une fonction de coût $\mathcal{L}$ mesurant l'écart entre la prédiction $\hat{y}$ et la vraie valeur y.

B.2.1 Fonction de coût

Pour une perte MSE (Mean Squared Error) :

$$\mathcal{L} = \frac{1}{2}(y - \hat{y})^2$$

Le facteur $\frac{1}{2}$ simplifie le calcul des dérivées. Notre objectif est de minimiser $\mathcal{L}$ via la descente de gradient.

B.2.2 Dérivation des gradients

Le gradient pour un poids w_i s'obtient par :

$$\frac{\partial \mathcal{L}}{\partial w_i} = \underbrace{\frac{\partial \mathcal{L}}{\partial \hat{y}}}_{\text{(1) Dérivée de la perte}} \cdot \underbrace{\frac{\partial \hat{y}}{\partial z}}_{\text{(2) Dérivée de l'activation}} \cdot \underbrace{\frac{\partial z}{\partial w_i}}_{\text{(3) Contribution du poids}}$$

Avec pour chaque terme :
— (1) $\frac{\partial \mathcal{L}}{\partial \hat{y}} = \hat{y} - y$ (pour MSE)
— (2) $\frac{\partial \hat{y}}{\partial z} = \sigma'(z)$ (dépend de la fonction d'activation)
— (3) $\frac{\partial z}{\partial w_i} = a_i$ (activation du neurone précédent)

Pour un réseau profond, la formule générale pour un poids $w_{ij}^{(l)}$ à la couche l est :

$$\frac{\partial \mathcal{L}}{\partial w_{ij}^{(l)}} = \delta_j^{(l)} a_i^{(l-1)}$$

où $\delta_j^{(l)}$ est l'erreur propagée au neurone j de la couche l.

B.3 Algorithme complet

Pour un réseau profond avec L couches :

1. **Forward pass** :
   — Calculer toutes les activations $a^{(l)}$ et entrées $z^{(l)}$ pour $l = 1$ à L
2. **Backward pass** :
   — Initialiser $\delta^{(L)} = \nabla_a \mathcal{L} \odot \sigma'(z^{(L)})$ (couche sortie)
   — Pour $l = L - 1$ à 1 :
   $$\delta^{(l)} = ((w^{(l+1)})^T \delta^{(l+1)}) \odot \sigma'(z^{(l)})$$
   — Calculer les gradients :
   $$\frac{\partial \mathcal{L}}{\partial w^{(l)}} = \delta^{(l)} (a^{(l-1)})^T$$
3. **Mise à jour des paramètres(Descente de gradients)** :
   — Pour chaque poids :
   $$w^{(l)} \leftarrow w^{(l)} - \eta \frac{\partial \mathcal{L}}{\partial w^{(l)}}$$
   — Pour chaque biais :
   $$b^{(l)} \leftarrow b^{(l)} - \eta \delta^{(l)}$$

B.4 Formules importantes

Pour une couche l :
— **Entrée** : $z^{(l)} = w^{(l)} a^{(l-1)} + b^{(l)}$
— **Activation** : $a^{(l)} = \sigma(z^{(l)})$
— **Erreur** : $\delta^{(l)} = \begin{cases} \nabla_a \mathcal{L} \odot \sigma'(z^{(L)}) & \text{(couche sortie)} \\ ((w^{(l+1)})^T \delta^{(l+1)}) \odot \sigma'(z^{(l)}) & \text{(couche cachée)} \end{cases}$
— **Gradient poids** : $\nabla_{w^{(l)}} \mathcal{L} = \delta^{(l)} (a^{(l-1)})^T$
— **Gradient biais** : $\nabla_{b^{(l)}} \mathcal{L} = \delta^{(l)}$

Références

— Baydin, A. G. et al. (2018). *Automatic Differentiation in Machine Learning : a Survey*. Journal of Machine Learning Research, 18(153), 1-43.
— Cours Intelligence Artificielle (Université de Genève) : `https://pgc.unige.ch/main/teachings/details/13X005?year=2024&fac=default-value`
— Cours Compilateur et Interpréteur (Université de Genève) : `https://pgc.unige.ch/main/teachings/details/2024-13X001`
— Wikipédia : `https://fr.wikipedia.org/wiki/D%C3%A9rivation_automatique`
— Vidéo explicative 3Blue1Brown : `https://www.youtube.com/watch?v=Ilg3gGewQ5U`
— Vidéo explicative Gabriel Turinici : `https://www.youtube.com/watch?v=2P4lslpg1-w&t=182s`
— Vidéo explicative deepmath : `https://www.youtube.com/watch?v=xQKw1xqNtvE&t=776s`
— Vidéo explicative Alexandre TL : `https://www.youtube.com/watch?v=MmsFAlGIsEg`