

Rapport de Stage

Différentiation Automatique en Python

Implémentation des modes Forward et Backward

Concepts abordés :

- Différentiation automatique
- Mode forward et backward
- Surcharge d'opérateurs
- Graphe computationnel
- Rétropropagation

21 juin 2025

Table des matières

1 Lectures et ressources personnelles

Durant ce stage, j'ai consulté plusieurs ressources scientifiques et techniques qui m'ont aidé à comprendre les concepts clés liés à la différentiation automatique et à leur mise en œuvre en Python, notamment :

Ressources bibliographiques

Baydin, A.G., Pearlmutter, B.A., Radul, A.A., & Siskind, J.M. (2018). **Automatic Differentiation in Machine Learning : A Survey**. *Journal of Machine Learning Research*, 18(153), 1–43.

Ce travail m'a permis de mieux comprendre les fondements théoriques de la différentiation automatique, les différences entre les modes forward et backward, ainsi que les principales approches d'implémentation.

Victor Martin – Machine Learning Engineer

J'ai également tiré profit d'un article / tutoriel rédigé par Victor Martin, qui présente de manière pédagogique l'implémentation d'un système de différentiation automatique par surcharge d'opérateurs. Cette ressource m'a aidé à structurer l'architecture de mon propre module.

2 Introduction à la Différentiation Automatique

2.1 Principes fondamentaux

Définition

La différentiation automatique (DA) est une technique pour calculer les dérivées de manière précise et efficace en utilisant un programme informatique. Contrairement à :

- La différentiation symbolique (comme celle que l'on fait à la main), qui produit des expressions mathématiques complexes.
- La différentiation numérique (comme les différences finies), qui introduit des erreurs dues aux approximations.

La DA est basée sur une idée simple : tout programme informatique est une séquence d'opérations élémentaires (addition, multiplication, etc.) et de fonctions mathématiques (exp, sin, cos, etc.). En appliquant la règle de la chaîne à ces opérations, on peut calculer la dérivée de manière automatique.

Important

La différentiation automatique (DA) est une méthode qui permet de calculer les dérivées de manière automatique pendant l'exécution d'un programme. Au lieu de produire des formules complexes de dérivées, la DA calcule directement les valeurs numériques des dérivées avec une grande précision (au niveau de la machine). Cette méthode est rapide et efficace, sans nécessiter de réorganiser le code en formules

mathématiques complexes, comme c'est le cas avec la différentiation symbolique.

La DA est essentielle dans les réseaux de neurones, notamment pour le calcul efficace et précis des gradients nécessaires à l'entraînement. Dans les réseaux de neurones, l'apprentissage repose sur la minimisation d'une fonction de perte, qui évalue les erreurs entre les prédictions du modèle et les valeurs réelles. Cette minimisation est généralement effectuée par des algorithmes d'optimisation basés sur la descente de gradient, qui nécessitent les dérivées partielles de la fonction de perte par rapport aux paramètres du modèle.

La DA permet de calculer ces dérivées de manière exacte, sans les erreurs d'approximation associées aux méthodes de différentiation numérique. Contrairement à la différentiation symbolique, qui produit des expressions complexes difficiles à gérer, la DA génère directement les valeurs des dérivées lors de l'exécution du code, tout en garantissant une précision au niveau machine. Cela est particulièrement important dans les réseaux de neurones profonds, où les modèles comportent des milliers, voire des millions de paramètres.

Grâce à la DA, les frameworks d'apprentissage automatique modernes (comme TensorFlow, PyTorch et JAX) peuvent optimiser efficacement les réseaux de neurones, même pour des architectures complexes. En conséquence, la DA est devenue une composante fondamentale de l'apprentissage profond, rendant possible l'entraînement rapide et stable des modèles de réseaux de neurones.

2.2 La Règle de la Chaîne et les Modes de DA

La DA repose sur la règle de la chaîne, qui décrit comment les dérivées se propagent dans une composition de fonctions. Pour une fonction composée comme :

$$y = f(g(h(x)))$$

La dérivée de y par rapport à x est donnée par :

$$\frac{dy}{dx} = \frac{dy}{dw_2} \cdot \frac{dw_2}{dw_1} \cdot \frac{dw_1}{dx}$$

Où :

$$w_1 = h(x) \tag{1}$$

$$w_2 = g(w_1) \tag{2}$$

$$y = f(w_2) \tag{3}$$

2.2.1 Exemple de Différentiation Automatique

Exemple

Fonction d'exemple

Considérons la fonction suivante :

$$f(x_1, x_2) = x_1^2 \cdot x_2 + \cos(x_2) = y$$

où $f : \mathbb{R}^2 \rightarrow \mathbb{R}$ avec $n = 2$ et $m = 1$.

Décomposition en opérations élémentaires

On définit les variables suivantes :

- $v_{i-n} = x_i$ avec $i = 1, \dots, n$ sont les variables d'entrée ;
- v_i avec $i = 1, \dots, l$ sont les variables intermédiaires ;
- $y_{m-i} = v_{l-i}$ avec $i = m - 1, \dots, 0$ sont les variables de sortie.

Notations :

$$v_{-1} = x_1 \quad (4)$$

$$v_0 = x_2 \quad (5)$$

$$v_1 = v_{-1}^2 \quad (6)$$

$$v_2 = v_1 \cdot v_0 \quad (7)$$

$$v_3 = \cos(v_0) \quad (8)$$

$$v_4 = v_2 + v_3 \quad (9)$$

Graphiquement, nous pouvons aussi représenter cette fonction sous la forme d'un graphe acyclique dirigé, où les variables sont les nœuds du graphe, et les arêtes représentent les relations algébriques.

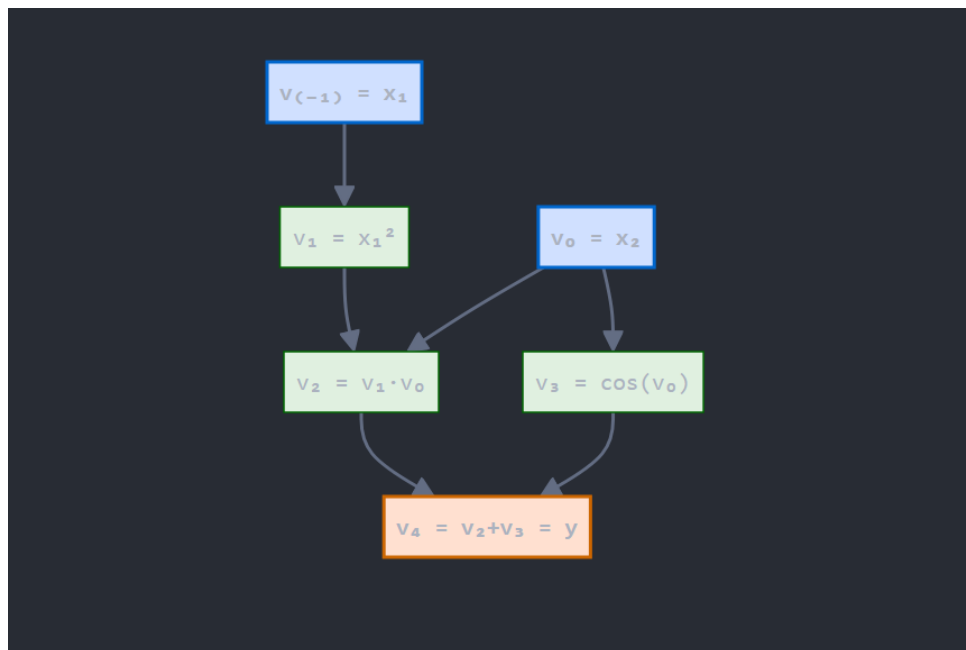


FIGURE 1 – Graphe computationnel de la fonction $f(x_1, x_2) = x_1^2 \cdot x_2 + \cos(x_2)$

2.3 Modes de Différentiation Automatique

2.3.1 Mode avant (Forward Mode)

Mode Forward

Dans le mode avant, on propage les dérivées dans le même sens que le calcul de la fonction. Pour chaque variable intermédiaire v_i , on calcule sa valeur et sa dérivée directionnelle dans la direction d'une entrée donnée.

Si on considère la direction $\dot{x} = (1, 0)$ (c'est-à-dire la dérivée par rapport à x_1), nous avons :

$$\dot{v}_{-1} = 1 \quad (\text{car } v_{-1} = x_1) \quad (10)$$

$$\dot{v}_0 = 0 \quad (\text{car } v_0 = x_2) \quad (11)$$

$$\dot{v}_1 = 2v_{-1} \cdot \dot{v}_{-1} = 2x_1 \cdot 1 = 2x_1 \quad (12)$$

$$\dot{v}_2 = v_0 \cdot \dot{v}_1 + v_1 \cdot \dot{v}_0 = x_2 \cdot 2x_1 + x_1^2 \cdot 0 = 2x_1x_2 \quad (13)$$

$$\dot{v}_3 = -\sin(v_0) \cdot \dot{v}_0 = -\sin(x_2) \cdot 0 = 0 \quad (14)$$

$$\dot{v}_4 = \dot{v}_2 + \dot{v}_3 = 2x_1x_2 + 0 = 2x_1x_2 \quad (15)$$

Donc $\frac{\partial f}{\partial x_1} = \dot{v}_4 = 2x_1x_2$

Si on considère la direction $\dot{x} = (0, 1)$ (c'est-à-dire la dérivée par rapport à x_2), nous avons :

$$\dot{v}_{-1} = 0 \quad (\text{car } v_{-1} = x_1) \quad (16)$$

$$\dot{v}_0 = 1 \quad (\text{car } v_0 = x_2) \quad (17)$$

$$\dot{v}_1 = 2v_{-1} \cdot \dot{v}_{-1} = 2x_1 \cdot 0 = 0 \quad (18)$$

$$\dot{v}_2 = v_0 \cdot \dot{v}_1 + v_1 \cdot \dot{v}_0 = x_2 \cdot 0 + x_1^2 \cdot 1 = x_1^2 \quad (19)$$

$$\dot{v}_3 = -\sin(v_0) \cdot \dot{v}_0 = -\sin(x_2) \cdot 1 = -\sin(x_2) \quad (20)$$

$$\dot{v}_4 = \dot{v}_2 + \dot{v}_3 = x_1^2 + (-\sin(x_2)) = x_1^2 - \sin(x_2) \quad (21)$$

Donc $\frac{\partial f}{\partial x_2} = \dot{v}_4 = x_1^2 - \sin(x_2)$

2.3.2 Mode inverse (Mode Backward)

Mode Backward

Dans le mode inverse (ou rétropropagation), on propage les dérivées dans le sens inverse du calcul de la fonction. Pour chaque variable intermédiaire v_i , on calcule sa contribution à la dérivée de la sortie.

Si on note $\bar{v}_i = \frac{\partial y}{\partial v_i}$ la dérivée de la sortie y par rapport à v_i , alors en appliquant la règle de la chaîne en sens inverse :

$$\bar{v}_4 = \frac{\partial y}{\partial v_4} = 1 \quad (\text{car } y = v_4) \quad (22)$$

$$\bar{v}_3 = \frac{\partial y}{\partial v_3} = \bar{v}_4 \cdot \frac{\partial v_4}{\partial v_3} = 1 \cdot 1 = 1 \quad (\text{car } v_4 = v_2 + v_3) \quad (23)$$

$$\bar{v}_2 = \frac{\partial y}{\partial v_2} = \bar{v}_4 \cdot \frac{\partial v_4}{\partial v_2} = 1 \cdot 1 = 1 \quad (\text{car } v_4 = v_2 + v_3) \quad (24)$$

$$\bar{v}_1 = \frac{\partial y}{\partial v_1} = \bar{v}_2 \cdot \frac{\partial v_2}{\partial v_1} = 1 \cdot v_0 = x_2 \quad (\text{car } v_2 = v_1 \cdot v_0) \quad (25)$$

$$\bar{v}_0 = \frac{\partial y}{\partial v_0} = \bar{v}_2 \cdot \frac{\partial v_2}{\partial v_0} + \bar{v}_3 \cdot \frac{\partial v_3}{\partial v_0} \quad (26)$$

$$= 1 \cdot v_1 + 1 \cdot (-\sin(v_0)) = x_1^2 - \sin(x_2) \quad (27)$$

$$\bar{v}_{-1} = \frac{\partial y}{\partial v_{-1}} = \bar{v}_1 \cdot \frac{\partial v_1}{\partial v_{-1}} = x_2 \cdot 2v_{-1} = x_2 \cdot 2x_1 = 2x_1x_2 \quad (28)$$

Les dérivées partielles de y par rapport aux entrées sont donc :

$$\frac{\partial f}{\partial x_1} = \bar{v}_{-1} = 2x_1x_2 \quad (29)$$

$$\frac{\partial f}{\partial x_2} = \bar{v}_0 = x_1^2 - \sin(x_2) \quad (30)$$

Ces résultats correspondent exactement à ceux obtenus par le mode avant et confirment les dérivées partielles analytiques de la fonction $f(x_1, x_2) = x_1^2 \cdot x_2 + \cos(x_2)$.

2.4 Avantages de la Différentiation Automatique

Avantages clés de la DA

- **Précision** : La DA calcule les dérivées avec une précision machine, contrairement aux méthodes numériques qui introduisent des erreurs d'approximation.
- **Efficacité** : Le mode inverse est particulièrement efficace pour les fonctions avec beaucoup d'entrées et peu de sorties, comme en apprentissage profond.
- **Flexibilité** : La DA peut être appliquée à des programmes arbitrairement complexes, incluant des boucles et des conditions.

2.5 Implémentations

Il existe trois approches principales pour implémenter la différenciation automatique (DA) : (1) transformation de code source, (2) surcharge d'opérateurs, et (3) cadres à graphe computationnel. Chaque approche a ses avantages et inconvénients, en fonction du langage de programmation et des besoins de l'utilisateur.

1. Transformation de code source

Cette méthode consiste à analyser le code source d'un programme pour y insérer les calculs des dérivées. Elle produit un code optimisé, mais nécessite un analyseur

spécifique au langage et peut être complexe pour les programmes à flux de contrôle dynamique.

2. Surcharge d'opérateurs

Cette approche repose sur la redéfinition des opérateurs arithmétiques et des fonctions mathématiques pour propager les dérivées. Elle est particulièrement adaptée aux langages orientés objets et permet une intégration transparente avec le code existant. Cependant, elle peut introduire un surcoût à l'exécution dû à la gestion des objets dérivés.

3. Cadres à graphe computationnel

Les calculs sont représentés sous forme de graphes dirigés, où les nœuds représentent les opérations et les arêtes les dépendances de données. Cette méthode est couramment utilisée en apprentissage automatique pour sa flexibilité, mais elle peut être moins efficace pour les petits calculs.

Important

Dans ce projet, nous nous concentrerons sur la surcharge d'opérateurs, car nous concevrons et implémenterons tout en Python.

En parlant de surcharge d'opérateurs, je vais donner une petite explication ici. La surcharge d'opérateurs repose sur la redéfinition des opérateurs arithmétiques (+, -, *, /) et des fonctions mathématiques (sin, cos, exp) pour propager automatiquement les dérivées lors des calculs.

3 Architecture modulaire en Python pour la surcharge d'opérateur, le traçage d'opérations mathématiques et le calcul de dérivées en mode forward

3.1 Implémentation du mode forward pour la différenciation automatique

3.1.1 Structure du projet

Pour cette partie, j'ai défini trois modules (`forward.py`, `utile.py` et `exempleforward.py`).

Module `forward.py`

À l'intérieur de ce module, j'ai défini la classe `AutoDiff` qui possède deux attributs principaux :

- `valeur` : pour renvoyer la valeur de la fonction évaluée à un point
- `derive` : pour renvoyer l'évaluation de la dérivée à ce même point

À l'intérieur de cette classe, j'ai redéfini les opérations mathématiques de base (+, -, *, /) ainsi que quelques fonctions mathématiques fréquemment utilisées (cos, sin, log). Dans ces méthodes, j'ai directement inclus les règles de dérivation mathématique correspondantes.

Module `utile.py`

Ce module contient deux fonctions importantes :

- `variable()` : pour définir les variables par rapport auxquelles on dérive
- `constant()` : pour définir les variables qui restent constantes lors de la dérivation

Ces fonctions permettent de distinguer les variables actives des constantes lors de l'évaluation des dérivées partielles.

Module `exemple.py`

Ce module est dédié aux tests avec des assertions (`assert`) pour vérifier la validité des résultats.

3.1.2 Exemple d'application

Prenons une fonction (fonction définie plus haut) à deux variables :

$$f(x_1, x_2) = x_1^2 \cdot x_2 + \cos(x_2)$$

où $f : \mathbb{R}^2 \rightarrow \mathbb{R}$ avec $n = 2$ et $m = 1$.

Exemple**Calcul de la dérivée partielle par rapport à x_1**

Dans ce cas, nous considérons x_1 comme variable et x_2 comme constante.

La dérivée analytique est :

$$\frac{\partial f}{\partial x_1} = 2x_1x_2$$

En évaluant en $x_1 = 1$ et $x_2 = 2$:

$$\frac{\partial f}{\partial x_1} = 2 \cdot 1 \cdot 2 = 4$$

En exécutant notre implémentation dans VS Code, nous obtenons bien cette valeur.

Exemple**Calcul de la dérivée partielle par rapport à x_2**

Maintenant, nous considérons x_2 comme variable et x_1 comme constante.

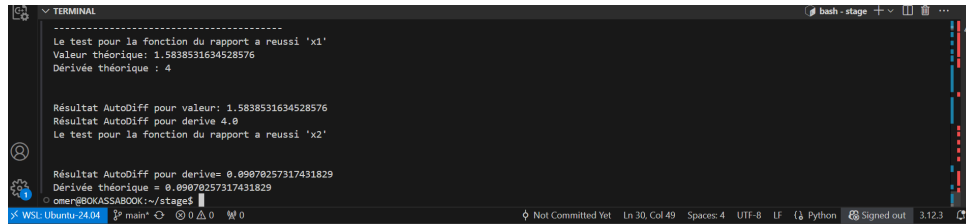
La dérivée analytique est :

$$\frac{\partial f}{\partial x_2} = x_1^2 - \sin(x_2)$$

En évaluant en $x_1 = 1$ et $x_2 = 2$:

$$\frac{\partial f}{\partial x_2} = 1^2 - \sin(2) \approx 0.090$$

Notre implémentation retourne également cette même valeur.



```
-----  
Le test pour la fonction du rapport a réussi 'x1'  
Valeur théorique: 1.583851634528576  
Dérivée théorique : 4  
  
Résultat AutoDiff pour valeur: 1.583851634528576  
Résultat AutoDiff pour derive 4.0  
Le test pour la fonction du rapport a réussi 'x2'  
  
Résultat AutoDiff pour derive= 0.89070257317431829  
Dérivée théorique = 0.89070257317431829  
omer@BOKASSABOOK:~/stage5
```

FIGURE 2 – Capture d'écran des résultats d'exécution du mode forward

3.2 Implémentation du mode backward pour la différenciation automatique

3.2.1 Structure du projet

Architecture du mode backward

Pour implémenter le **mode backward de la différenciation automatique**, j'ai conçu une classe centrale `Tensor`, dans un fichier unique `backward.py`, afin d'optimiser l'organisation et la performance du code. Cette classe encapsule une valeur scalaire, son gradient (`grad`), les fils (ou nœuds enfants du graphe de calcul), et l'opération associée (addition, multiplication, exponentielle, etc.).

La méthode clé `backward()` implémente la rétropropagation récursive à travers un graphe computationnel. Lorsqu'une opération est effectuée (comme `+`, `*`, `sin`, etc.), un nouvel objet `Tensor` est créé avec un lien vers ses parents (fils) et l'opération utilisée. Le gradient est propagé en appelant `backward()` sur les parents, selon les règles de dérivation spécifiques à chaque opération (par exemple, pour $z = x \times y$, on propage $\frac{dz}{dx} = y$ et $\frac{dz}{dy} = x$).

Les opérations arithmétiques de base ont été redéfinies (`__add__`, `__mul__`, `__pow__`, etc.), ainsi que plusieurs fonctions usuelles (`sin`, `cos`, `log`, `sqrt`, etc.), pour permettre une construction fluide du graphe computationnel.

Important

J'ai choisi de regrouper tout dans un seul module au lieu de répartir en plusieurs fichiers (comme `Tensor.py`, `grad.py`, `operation.py`) pour réduire la complexité du chargement et améliorer la performance, notamment en évitant les imports multiples et les risques de circularité dus à l'appel récursif dans `backward()`.

Un second fichier `exemplebackward.py` contient des fonctions exemples sur lesquelles j'ai validé la justesse des gradients calculés.

3.2.2 Exemple détaillé : Construction du graphe computationnel et rétropropagation

Pour illustrer le fonctionnement de l'algorithme de différenciation automatique en mode backward, nous analysons pas à pas l'exécution sur un exemple concret.

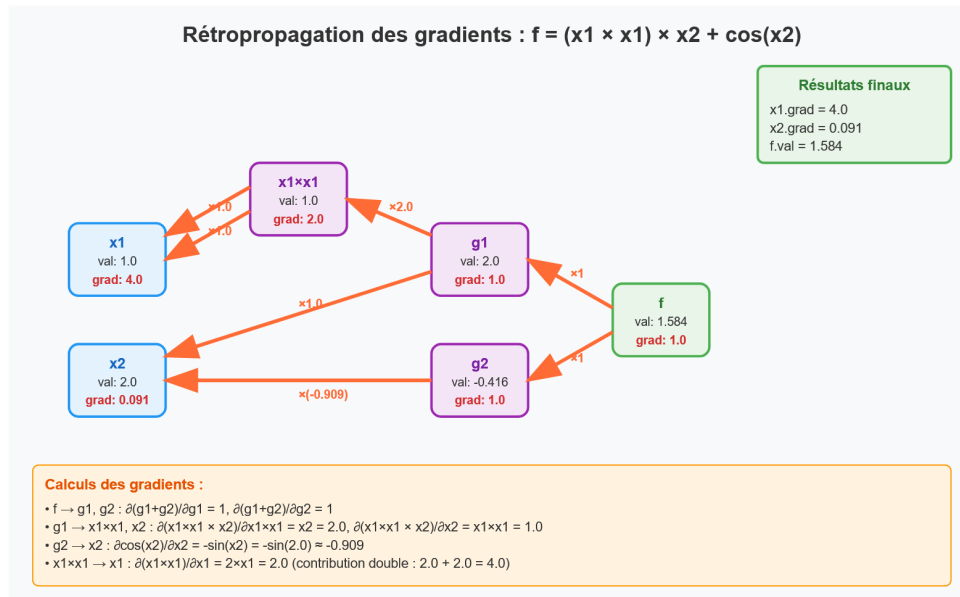


FIGURE 3 – Graphe computationnel de la fonction $f(x_1, x_2) = x_1^2 \cdot x_2 + \cos(x_2)$

Suite

1 Benchmark

1.1 Objectif et outils

L'objectif est d'analyser les performances des modes d'autodifférenciation (forward et backward) sur la fonction :

$$f = (x_1 \cdot \sin(x_2) + \log((x_3^2 + 1)^2) + \exp(x_1 \cdot x_3)) \cdot \cos(x_2) \quad (1)$$

Important

Outils de profilage :

- **cProfile avec SnakeViz** : Analyse du temps via une interface web intuitive.
- **Scalene** : Mesures détaillées (CPU, mémoire, GPU).
- **memory_profiler** : Suivi mémoire pour le mode backward.

1.2 Mode Backward

Observations avec cProfile + SnakeViz (10 000 000 appels à `.backward()`) :

- Temps concentré sur les opérations mathématiques et `__init__` de Tensor.
- Optimisations : Mémorisation des sinus/cosinus et suppression des gradients inutiles, réduisant un peu le temps, car avant le temps d'exécution était 98,5 s.

Scalene :

- Utilisation CPU : 95%.
- Mémoire stable, sans fuites.
- Exécution sur CPU (pas de GPU).

memory_profiler :

- Pic mémoire : 37,1 MiB (\approx 38,8 Mo).

Définition

Conclusion : Les optimisations réduisent le coût CPU, mais l'implémentation Python reste limitée par la récursivité. Une réécriture en C/C++ via Cython est envisageable.

1.3 Mode Forward

- Pas de graphe, dérivées calculées en direct.
- Utilisation de numpy pour calculs vectorisés (optimisés en C).
- Temps : 0,4 s pour 10 millions d'appels.
- Remplacement de math par numpy pour vecteurs.

1.4 Comparaison Forward, Backward et PyTorch

Résultats :

Mode	Temps (10M appels)	Mémoire
Forward (bobodiff)	0,4 s	38,8 Mo
Backward (bobodiff)	98,5 s	38,8 Mo
Backward (PyTorch)	1114 s (≈ 18 min)	450,6 Mo

TABLE 1 – Comparaison des performances entre les différents modes

Analyse :

- **Forward** : Rapide grâce à l'absence de graphe et numpy.
- **Backward** : Coûteux (récursivité, création d'objets Tensor).
- **PyTorch** : Lent ici à cause de son infrastructure lourde, mais optimisé pour des modèles complexes.

Définition

Conclusion : bobodiff excelle en mode forward pour cas simples, mais backward est moins performant. PyTorch est adapté aux grands modèles.

1.5 Benchmark Bobodiff vs PyTorch (Réseau de neurones)

Configuration : Réseau avec 1 à 1000 entrées, couche cachée (même nombre de neurones), sortie unique, activation sigmoïde.

Résultats :

- **Bobodiff** : Temps de 2228,68 ms (600 entrées) à 6622,13 ms (1000 entrées), croissance quadratique (calculs non vectorisés).
- **PyTorch** : Temps de 27,83 ms (600 entrées) à 57,20 ms (1000 entrées), croissance linéaire (`torch.matmul`).
- **Facteur d'accélération** : PyTorch ≈ 100 fois plus rapide.

Observations :

- bobodiff fonctionne jusqu'à 1000 entrées avec `sys.setrecursionlimit(10000)`, mais la récursivité reste un goulot d'étranglement.

Important

Recommandations :

- Remplacer la récursivité par une approche itérative dans `backward()`.
- Utiliser numpy pour calculs vectorisés.

1.6 Benchmark Mode Forward (Bobodiff vs PyTorch)

Configuration : Réseau avec 1 à 1000 entrées, couche cachée (`tanh`), sortie unique.

Résultats :

Définition

Conclusion : bobodiff est fonctionnel en mode forward, mais moins performant que PyTorch. Des calculs vectorisés via numpy sont recommandés.

Framework	Temps (700 entrées)	Temps (1000 entrées)	Croissance
Bobodiff	444,57 ms	891,61 ms	Quadratique
PyTorch	29,19 ms	50,39 ms	Linéaire
Speedup	PyTorch 11 à 17 fois plus rapide		

TABLE 2 – Comparaison des performances en mode Forward

2 Annexes

2.1 Images

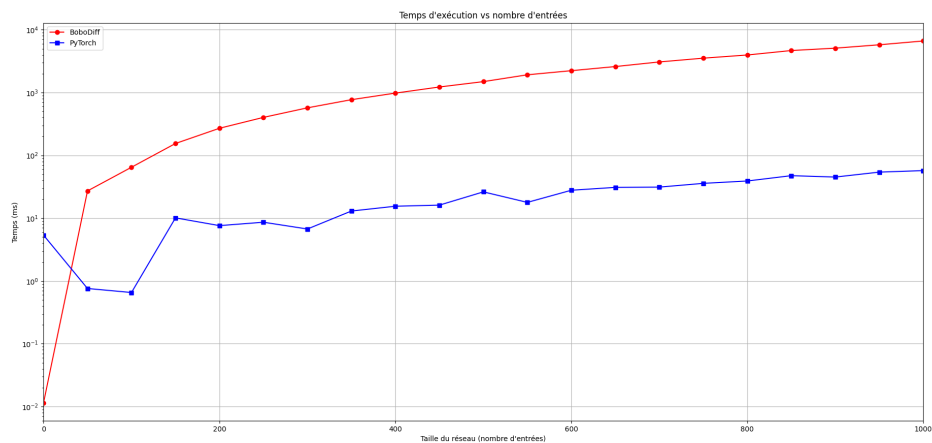


FIGURE 1 – Mode Backward

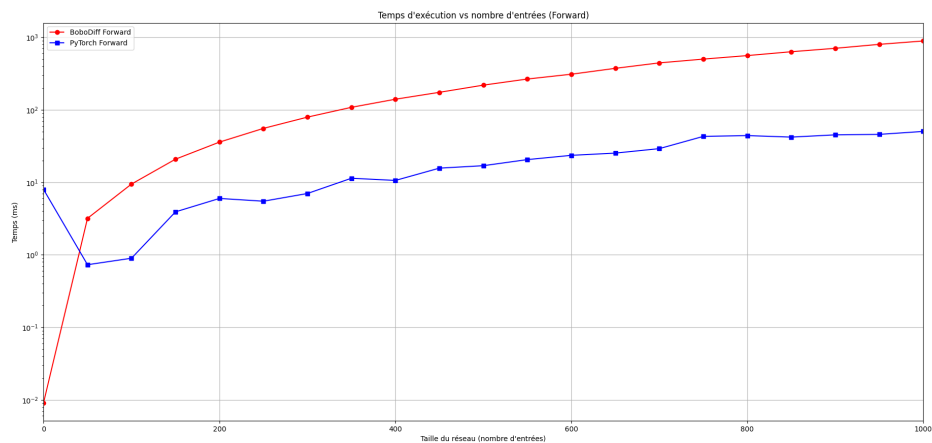


FIGURE 2 – Mode Forward