

Dériver un programme

La règle de la chaîne, des listes de Wengert à PyTorch

règle de la chaîne

mode inverse

différences finies

bande

Comment un programme peut-il calculer la pente de sa propre sortie, sans formule, et presque sans surcoût ?

Un cadeau de fin de cours, à lire par curiosité : rien n'est à rendre. Le projet « Micrograd » reste décrit par sa fiche sur le site et par le README de son squelette ; ce fascicule raconte les idées qui l'entourent et ce qu'elles deviennent au-delà du cours.

1 Pourquoi ce fascicule

Dans le projet Micrograd, vous avez écrit les trois fonctions qui manquaient à un petit moteur de différentiation automatique : les dérivées locales du produit et de $\tanh$, puis la propagation inverse qui remonte du résultat vers les feuilles en additionnant les contributions. Le graphe, son ordre de parcours, le réseau 2-4-1 et la boucle d'apprentissage étaient fournis ; une comparaison avec les différences finies a servi de juge de paix, et les quatre points XOR ont, si tout s'est bien passé, fini par être bien classés.

Ce fascicule ne revient pas sur ce code. Il raconte ce qu'il y a autour : pourquoi la règle de la chaîne, appliquée nœud par nœud, suffit à dériver n'importe quel programme numérique ; pourquoi il existe deux sens de parcours, et pourquoi celui que vous avez écrit, le mode inverse, coûte si peu quel que soit le nombre de paramètres ; qui a eu ces idées, et quand ; comment vérifier un gradient sans se faire confiance ; enfin comment PyTorch, JAX ou Nx s'y prennent à grande échelle, et ce que cela coûte en mémoire. Rien n'est à rendre : lisez-le un soir, un crayon à la main pour refaire le tableau de la section suivante.

2 La règle de la chaîne, nœud par nœud

La règle de la chaîne dit que si f dépend de v et v de x , alors $\frac{df}{dx} = \frac{df}{dv} \times \frac{dv}{dx}$. Le chapitre 8 du poly l'appliquait déjà à la parabole $q(u) = (u - 3)^2$. Toute la différentiation automatique repose sur une observation simple : un programme numérique, aussi long soit-il, n'est qu'une suite d'opérations élémentaires dont chacune a une dérivée connue. Il suffit donc d'appliquer la règle de la chaîne à chaque opération, dans le bon ordre, pour obtenir la dérivée de l'ensemble, sans jamais écrire la formule développée.

Fixons le vocabulaire. Un *graphe de calcul* est l'ensemble des valeurs intermédiaires d'un programme, reliées par les opérations qui les produisent : les *feuilles* sont les entrées et les paramètres, la *racine* est le résultat, et chaque nœud intérieur connaît ses *parents*, c'est-à-dire ses arguments. La *dérivée locale* d'un nœud par rapport à l'un de ses parents est la dérivée de la seule opération qui le produit, prise isolément : pour $v = u + x$, elle vaut 1 dans les deux directions ; pour $u = xy$, elle vaut y selon x et x selon y . Enfin, l'*adjoint* d'un nœud k , que l'on note $\bar{k}$, est la dérivée du résultat

final par rapport à ce nœud : $\bar{k} = \frac{\partial f}{\partial k}$. C'est lui que la passe arrière calcule, et c'est lui que le squelette appelait simplement « gradient ».

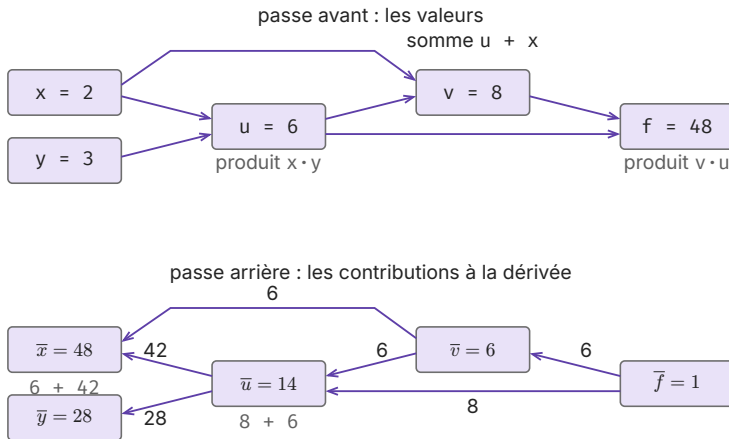
2.1 Un exemple avec une valeur réutilisée

Prenons un calcul voisin de celui du projet, choisi pour qu'une valeur intermédiaire serve deux fois. Avec $x = 2$ et $y = 3$:

$$u = xy, \quad v = u + x, \quad f = vu.$$

La *passe avant* évalue de gauche à droite : $u = 6$, $v = 8$, $f = 48$. La *passe arrière* part de la racine, dont la dérivée par rapport à elle-même vaut $\bar{f} = 1$, et remonte. Le nœud $f = vu$ transmet $\bar{f} \times u = 6$ à v et $\bar{f} \times v = 8$ à u . Le nœud $v = u + x$ reçoit donc $\bar{v} = 6$ et le transmet tel quel à u et à x , puisque ses dérivées locales valent 1. Voici le point délicat : u a reçu 8 directement de f et 6 par l'intermédiaire de v , et son adjoint est la somme, $\bar{u} = 14$. Il transmet ensuite $14 \times y = 42$ à x et $14 \times x = 28$ à y . Au total, $\bar{x} = 6 + 42 = 48$ et $\bar{y} = 28$.

Vérifions sur papier : $f = (xy + x)xy = x^2y^2 + x^2y$, d'où $\frac{\partial f}{\partial x} = 2xy^2 + 2xy = 36 + 12 = 48$ et $\frac{\partial f}{\partial y} = 2x^2y + x^2 = 24 + 4 = 28$. Les deux méthodes concordent, et la seconde n'a eu besoin d'aucune formule développée : seulement des dérivées locales et des additions.



En haut, les valeurs circulent de gauche à droite le long des dépendances. En bas, les mêmes flèches sont parcourues à rebours ; le nombre porté par chaque flèche est le produit de l'adjoint du nœud d'arrivée par la dérivée locale. Le nœud u reçoit deux contributions, 8 et 6, que l'on additionne.

Le tableau ci-dessous rassemble les deux passes. Lisez-le de haut en bas pour les valeurs, puis de bas en haut pour les adjoints : chaque ligne du bas de la colonne de droite n'utilise que les lignes situées au-dessous d'elle.

Nœud	Calcul	Valeur	Dérivées locales	Adjoint
x	feuille	2		$6 + 42 = 48$
y	feuille	3		28
u	xy	6	$y = 3$ selon x , $x = 2$ selon y	$8 + 6 = 14$
v	$u + x$	8	1 et 1	6
f	vu	48	$u = 6$ selon v , $v = 8$ selon u	1

2.2 Pourquoi additionner les contributions

Pourquoi la somme, et pas autre chose ? Parce que la règle de la chaîne à plusieurs variables le dit : si le résultat f dépend de k à travers plusieurs nœuds $c_1, \dots, c_m$ qui utilisent k comme argument, alors

$$\bar{k} = \sum_{i=1}^m \bar{c}_i \times \frac{\partial c_i}{\partial k}.$$

Chaque chemin qui part de k vers la racine apporte sa part, et les parts s'ajoutent. Un moteur qui en oublierait une, par ex. en transformant la liste des parents $[x, x]$ d'un carré en ensemble, calculerait exactement la dérivée d'une autre fonction. C'est le souci que les tests du projet guettaient avec le parent présent deux fois.

INVARIANT DE LA PASSE ARRIÈRE

Si l'on traite les nœuds dans l'ordre inverse d'un ordre topologique, c'est-à-dire chaque nœud après tous ceux qui l'utilisent, alors au moment où un nœud est traité, son adjoint est complet. En effet, la racine n'a aucun utilisateur et son adjoint vaut 1 dès le départ. Un utilisateur c de k dépend de k : il apparaît donc après k dans l'ordre topologique, avant lui dans l'ordre inverse, et il a déjà ajouté sa contribution $\bar{c} \times \frac{\partial c}{\partial k}$ quand son tour est venu. Par récurrence sur la position dans l'ordre, chaque adjoint reçoit toutes ses contributions avant d'être lui-même distribué à ses parents.

C'est exactement ce que garantit l'ordre fourni par le squelette, et c'est la raison pour laquelle toute la passe arrière tient dans un seul parcours de cet ordre, sans jamais revenir sur un nœud déjà traité. Notez aussi ce que l'invariant ne dit pas : il ne

promet rien sur les nœuds qui ne mènent pas à la racine, lesquels restent simplement absents de la map des gradients.

3 Deux sens de parcours, et un théorème rassurant

Le graphe se parcourt dans les deux sens, et les deux sens calculent des dérivées exactes. Ils ne coûtent cependant pas le même prix, et la différence explique pourquoi l'apprentissage profond a choisi l'un plutôt que l'autre.

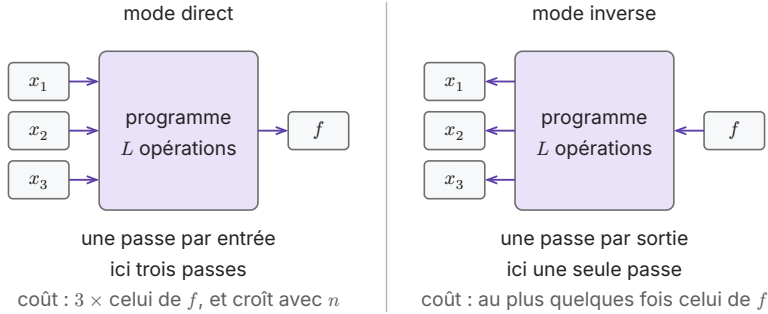
3.1 Le mode direct

Le *mode direct* (en anglais *forward mode*) propage les dérivées dans le même sens que les valeurs. On choisit une direction de dérivation, par ex. « selon x », on attache à chaque feuille sa dérivée dans cette direction, soit $\dot{x} = 1$ et $\dot{y} = 0$, et chaque nœud calcule, en même temps que sa valeur, sa dérivée à partir de celles de ses parents. Sur notre exemple, $\dot{u} = \dot{x}y + x\dot{y} = 3$, puis $\dot{v} = \dot{u} + \dot{x} = 4$, enfin $\dot{f} = \dot{v}u + v\dot{u} = 24 + 24 = 48$. On retrouve bien $\bar{x} = 48$, mais en une seule passe avant un peu alourdie, sans rien mémoriser. L'écriture la plus élégante de ce mode est celle des *nombreux* : on remplace chaque nombre a par le couple $(a, \dot{a})$ et l'on redéfinit les opérations pour que le second membre suive la règle de la chaîne, un peu comme la calculatrice à pile d'AP1 transportait son état d'une instruction à la suivante.

Le souci du mode direct est son coût en fonction du nombre d'entrées : une passe ne donne la dérivée que dans une direction. Pour les dix-sept paramètres de notre réseau, il faut dix-sept passes ; pour le milliard et demi de paramètres de GPT-2, un milliard et demi. C'est rédhibitoire pour l'apprentissage, où l'on veut la dérivée d'une seule perte par rapport à tous les paramètres.

3.2 Le mode inverse et le prix du gradient

Le *mode inverse* (*reverse mode*), celui du projet, procède à l'envers : une passe avant mémorise toutes les valeurs intermédiaires, puis une passe arrière part de la racine et distribue les adjoints. Une seule passe arrière donne les dérivées du résultat par rapport à toutes les feuilles à la fois, quel que soit leur nombre. En contrepartie, une passe arrière ne traite qu'une sortie : pour une fonction à m sorties, il en faudrait m . L'apprentissage, avec sa perte scalaire et ses millions de paramètres, est le cas idéal du mode inverse, et c'est pour cela que la rétropropagation s'est imposée.



Trois entrées, une sortie. Le mode direct propage une direction de dérivation à la fois, de gauche à droite ; le mode inverse propage l'adjoint de la sortie, de droite à gauche, et obtient les trois dérivées d'un coup.

Combien coûte cette passe arrière ? Chaque nœud du graphe est visité une fois, et chaque visite effectue quelques multiplications et additions, une par parent. Le coût total est donc proportionnel au nombre d'opérations du calcul lui-même, et non au nombre de paramètres. Walter Baur et Volker Strassen l'ont démontré en 1983 dans un cadre plus exigeant que le nôtre, celui des programmes arithmétiques : si une fonction rationnelle de n variables se calcule avec L multiplications et divisions, alors la fonction et ses n dérivées partielles se calculent ensemble avec au plus $3L$ multiplications et divisions. La constante ne dépend pas de n . Andreas Griewank a repris ce résultat en 1989 pour la différentiation automatique, et le livre qu'il a écrit avec Andrea Walther en 2008 le nomme *principe du gradient bon marché* ; dans leur modèle de coût, qui compte aussi les additions et les accès mémoire, le temps du gradient reste borné par environ quatre fois celui de la fonction.

À RETENIR

Dériver une fonction à n variables par le mode inverse coûte au plus quelques fois son évaluation, que n vaille dix-sept ou un milliard. Le prix se paie ailleurs : en mémoire, puisqu'il faut conserver les valeurs intermédiaires jusqu'à la passe arrière. Le mode direct ne stocke rien, mais son coût croît avec n .

4 Une histoire en quatre dates

L'idée a été trouvée plusieurs fois, par des gens qui ne se lisaient pas. En 1964, Robert Wengert publie dans les *Communications of the ACM* un court article qui décrit le

mode direct : décomposer un calcul en une liste d'opérations élémentaires, et faire suivre chacune de sa dérivée. Cette liste porte depuis son nom, *liste de Wengert* ; nous y reviendrons, car c'est aussi la « bande » des bibliothèques modernes. À la même époque, des ingénieurs du contrôle optimal, Henry Kelley en 1960 puis Arthur Bryson peu après, calculent des gradients de trajectoires en remontant les étapes d'un système dynamique, ce qui est déjà le mode inverse, mais dans un cadre continu et sans l'idée d'un programme quelconque.

Le mode inverse tel que nous l'avons écrit apparaît en 1970 dans le mémoire de master de Seppo Linnainmaa, à l'université d'Helsinki. Son but n'était pas l'apprentissage : il voulait estimer comment les erreurs d'arrondi se propagent dans un long calcul, ce qui demande la dérivée du résultat par rapport à chaque étape intermédiaire, c'est-à-dire tous les adjoints. Il publie la méthode en 1976 dans la revue *BIT*. En 1980, Bert Speelpenning, dans sa thèse à l'université de l'Illinois, en donne la première implémentation réellement automatique, qui transforme un programme ordinaire sans intervention de son auteur. De son côté, Paul Werbos l'applique aux réseaux de neurones dans sa thèse de 1974, puis dans un article de 1982, sans écho immédiat.

L'écho vient en 1986 : David Rumelhart, Geoffrey Hinton et Ronald Williams publient dans *Nature* « Learning representations by back-propagating errors ». L'algorithme est le même, mais l'article montre expérimentalement que les unités cachées d'un réseau entraîné ainsi apprennent des représentations utiles, ce que le perceptron ne pouvait pas faire. Le mot *rétropropagation* s'impose, et avec lui l'idée que l'on peut entraîner des réseaux à plusieurs couches. Enfin, la synthèse de référence est le livre d'Andreas Griewank et Andrea Walther, *Evaluating Derivatives*, dont la seconde édition paraît en 2008 : on y trouve les deux modes, le théorème du gradient bon marché, les points de reprise et bien d'autres raffinements.

EN PRATIQUE

Vous rencontrerez trois noms pour la même chose : *différentiation automatique en mode inverse* chez les numériciens, *rétropropagation* chez les spécialistes des réseaux de neurones, *autograd* dans les bibliothèques. La rétropropagation est le mode inverse appliqué à la perte d'un réseau ; autograd est le nom du module de PyTorch, devenu nom commun.

5 Vérifier un gradient sans le croire sur parole

Un moteur de dérivation se trompe silencieusement : une dérivée locale fausse ou une contribution oubliée produit des nombres plausibles, et l'apprentissage semble parfois marcher quand même, en plus lent. Il faut donc un juge indépendant. Les *différences finies* en sont un : elles approchent la dérivée par un taux d'accroissement, avec deux évaluations ordinaires de la fonction et aucune connaissance de sa structure.

Deux formules sont possibles. La différence *avant*, $\frac{f(x+h)-f(x)}{h}$, et la différence *centrée*, $\frac{f(x+h)-f(x-h)}{2h}$. Pourquoi la seconde est-elle meilleure ? Écrivons le développement de Taylor, $f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \frac{h^3}{6}f'''(x) + \dots$. En reportant dans la différence avant, il reste $f'(x) + \frac{h}{2}f''(x) + \dots$: l'erreur est proportionnelle à h . Dans la différence centrée, les termes en h^2 de $f(x+h)$ et de $f(x-h)$ sont égaux et se soustraient, et il reste $f'(x) + \frac{h^2}{6}f'''(x) + \dots$: l'erreur est proportionnelle à h^2 . Diviser h par dix divise donc l'erreur par dix dans un cas, par cent dans l'autre. Vérifions-le sur $f(x) = x^3$ en $x = 2$, dont la dérivée exacte vaut 12 :

```
1 cube = fn x -> x * x * x end
2 exact = 12.0
3
4 for h <- [0.1, 0.01, 0.001] do
5   avant = (cube.(2.0 + h) - cube.(2.0)) / h
6   centree = (cube.(2.0 + h) - cube.(2.0 - h)) / (2 * h)
7   IO.puts("h = #{h}  erreur avant : #{Float.round(avant - exact,
8     6)}" <>
9     "  centrée : #{Float.round(centree - exact, 9)}")
9 end
```

```
CONSOLE  h = 0.1  erreur avant : 0.61  centrée : 0.01
h = 0.01  erreur avant : 0.0601  centrée : 0.0001
h = 0.001  erreur avant : 0.006001  centrée : 1.0e-6
```

Les deux colonnes suivent exactement la prédiction : ici $f'' = 12$ et $f''' = 6$, donc l'erreur avant vaut environ $6h$ et l'erreur centrée exactement h^2 . Faut-il alors prendre h aussi petit que possible ? Prédisez ce que donne la différence centrée pour $h = 10^{-13}$ avant de lire la suite.

```

1 cube = fn x -> x * x * x end
2 sci = fn e -> :erlang.float_to_binary(e, scientific: 1) end
3
4 for h <- [1.0e-5, 1.0e-8, 1.0e-13] do
5   centree = (cube.(2.0 + h) - cube.(2.0 - h)) / (2 * h)
6   IO.puts("h = #{sci.(h)}   erreur centrée : #{sci.(centree - 12.0)}")
7 end

```

```

CONSOLE   h = 1.0e-05   erreur centrée : 2.1e-10
          h = 1.0e-08   erreur centrée : -7.3e-08
          h = 1.0e-13   erreur centrée : -9.6e-03

```

L'erreur diminue jusqu'à $h = 10^{-5}$, puis remonte. En clair, deux erreurs se disputent h . L'erreur de troncature, celle de Taylor, décroît comme h^2 . L'erreur d'arrondi, elle, croît quand h diminue : $f(x + h)$ et $f(x - h)$ sont deux flottants voisins, chacun connu à 10^{-16} près en valeur relative (cf. le chapitre 6 du poly), et leur différence perd d'autant plus de chiffres significatifs qu'ils sont proches ; cette perte est ensuite divisée par $2h$, donc amplifiée. Pour la différence centrée, le compromis se situe vers $h \approx 10^{-5}$ ou 10^{-6} en double précision, la racine cubique de la précision machine. Avec $h = 10^{-16}$, $x + h$ et $x - h$ sont le même flottant que x et la « dérivée » vaut zéro.

PIÈGE

Les différences finies ne remplacent pas le mode inverse : pour n paramètres, elles demandent $2n$ évaluations complètes du réseau, autant que le mode direct, avec en prime une erreur d'approximation. Elles sont un outil de test, précieux parce qu'indépendant, et c'est ainsi que le projet les utilisait : comparer vos dix-sept dérivées à une approximation grossière mais étrangère à votre code. Une tolérance trop serrée dans ce test rejette un moteur juste ; trop lâche, elle accepte une dérivée locale fausse d'un facteur deux.

6 Une bande ou un graphe : comment font les bibliothèques

Toutes les bibliothèques appliquent l'algorithme que vous avez écrit. Elles diffèrent par la manière d'obtenir le graphe, et cette différence a structuré dix ans d'outils d'apprentissage profond.

La première famille construit le graphe à l'avance, avant tout calcul. Theano, présenté en 2010 par l'équipe de Montréal, demandait d'assembler une expression symbolique, la dérivait, puis la compilait ; TensorFlow a suivi ce modèle en 2015, avec un graphe statique exécuté ensuite dans une session. C'est aussi le choix de Nx, lancé en 2021 par José Valim et Sean Moriarity : le corps d'un `defn` n'est pas exécuté, il est traduit en une expression que le compilateur inspecte, dérive et optimise, comme nous l'avons observé au chapitre 7 avec le compilateur de débogage. JAX, présenté par Google en 2018, procède par *tracage* : il exécute une fois la fonction Python avec des valeurs abstraites qui enregistrent les opérations rencontrées, obtient ainsi le graphe, puis le transforme ; `grad`, `jit` et `vmap` y sont des transformations de ce graphe.

La seconde famille enregistre les opérations au moment où elles s'exécutent, sur une *bande (tape)* : chaque opération réellement effectuée ajoute une ligne, avec ses arguments et sa dérivée locale, et la passe arrière relit la bande à l'envers. C'est la liste de Wengert de 1964, et c'est le modèle de Chainer en 2015, puis de PyTorch, dont le module autograd est décrit par Adam Paszke et ses collègues en 2017. L'avantage est la liberté : des boucles et des conditions ordinaires du langage hôte décident du graphe, qui peut changer à chaque appel. Le micrograd d'Andrej Karpathy, publié en 2020 en une centaine de lignes de Python, est la version minimale de cette famille : chaque valeur garde ses enfants et une petite fonction de rétropropagation, puis un tri topologique ordonne la passe arrière. Une bande a d'ailleurs une propriété agréable : l'ordre d'enregistrement est déjà un ordre topologique, puisqu'une opération ne peut utiliser que des valeurs calculées avant elle. Votre graphe immuable, où chaque identifiant est plus grand que ceux de ses parents, est une bande de ce genre, écrite avec les maps d'Elixir.

Pour relier ce fascicule au chapitre 8, voici notre exemple confié à Nx dans un carnet d'AP5, après la cellule `Mix.install` habituelle. `grad` dérive la fonction par rapport au tuple de ses deux arguments et rend les mêmes nombres que notre tableau :

```

1  defmodule Reuse do
2    import Nx.Defn
3
4    defn f(x, y) do
5      u = x * y
6      v = u + x
7      v * u
8    end
9
10   defn gradient(x, y), do: grad({x, y}, fn {a, b} -> f(a, b) end)
11 end
12
13 {dx, dy} = Reuse.gradient(2.0, 3.0)
14 {Nx.to_number(dx), Nx.to_number(dy)}

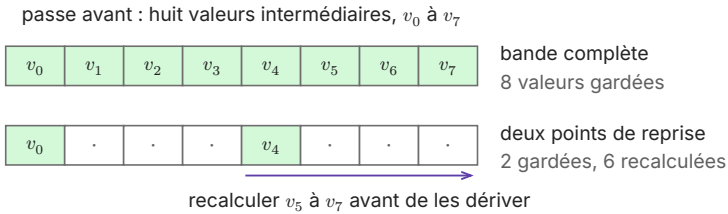
```

```
VALEUR {48.0, 28.0}
```

6.1 La mémoire du mode inverse

Le théorème de Baur et Strassen parle de temps. En mémoire, le mode inverse a un coût que le mode direct n'a pas : toutes les valeurs intermédiaires de la passe avant doivent survivre jusqu'à ce que la passe arrière les utilise, puisque les dérivées locales en dépendent. Dans notre projet, le graphe entier tient dans une map ; pour un grand réseau, ces valeurs, les *activations*, occupent la plus grande partie de la mémoire du GPU, bien plus que les paramètres eux-mêmes, car elles sont multipliées par la taille du lot et par la longueur des séquences.

La parade classique s'appelle *points de reprise (checkpointing)*. Au lieu de tout garder, on ne conserve que quelques valeurs jalons ; pendant la passe arrière, quand on atteint un tronçon dont les valeurs ont été oubliées, on les recalcule à partir du jalon précédent. On échange ainsi de la mémoire contre du temps. Andreas Griewank a montré en 1992 qu'un placement récursif des jalons limite à la fois la mémoire et le temps supplémentaire à un multiple du logarithme de la longueur du calcul. En 2016, Tianqi Chen et ses collègues ont appliqué l'idée aux réseaux profonds : en plaçant un jalon toutes les $\sqrt{n}$ couches d'un réseau à n couches, la mémoire des activations tombe de n à environ $\sqrt{n}$, au prix d'une passe avant supplémentaire. C'est ce que font aujourd'hui les grands entraînements sous le nom de *recalcul des activations*.



Huit étapes de calcul. Une bande complète garde les huit valeurs intermédiaires. Avec deux points de reprise, on ne garde que v_0 et v_4 : pour dériver les étapes 5 à 7, on les recalcule d'abord depuis v_4 , puis on fait de même pour les étapes 1 à 3 depuis v_0 .

7 Au-delà du projet : la programmation différentiable

Une fois que l'on sait dériver un programme, on est tenté de dériver autre chose qu'un réseau de neurones. Un simulateur physique, un moteur de rendu d'images, un solveur d'équations différentielles sont des programmes composés d'opérations élémentaires dérivables : le mode inverse leur donne un gradient, et le gradient permet d'ajuster leurs paramètres pour reproduire une observation, ou d'entraîner un réseau à travers eux. Ce programme de recherche porte le nom de *programmation différentiable*, popularisé en 2018 ; Mike Innes et ses collègues l'ont formulé en 2019 pour Julia, où le compilateur lui-même produit la passe arrière, et JAX en est l'incarnation la plus répandue en Python. Les *équations différentielles neuronales* de Ricky Chen et ses collègues, en 2018, en donnent un bel exemple : la passe arrière y est elle-même une équation différentielle, résolue à rebours, ce que le contrôle optimal appelait déjà méthode de l'adjoint dans les années 1960.

Ce qui change à l'échelle n'est pas l'algorithme, qui reste celui de votre `Enum.reduce`, mais ce qui l'entoure : la mémoire, que l'on vient de voir ; la précision, car les grands modèles calculent en flottants de 16 bits et doivent protéger les gradients de l'extinction ou du débordement ; la répartition du graphe sur des milliers de processeurs, où la passe arrière devient un problème de communication. Ce qui ne change pas non plus, ce sont les limites. Une condition `if` se dérive sur chaque branche, mais la condition elle-même n'a pas de dérivée ; un arrondi, un tri, un `argmax` ont une dérivée nulle presque partout et ne transmettent rien ; une fonction comme `tanh` saturée renvoie des dérivées locales minuscules, et le gradient s'évanouit en traversant beaucoup de couches. Ces limites ne sont pas des bogues du moteur : il calcule exactement la dérivée de ce qu'on lui donne. La question, à grande échelle

comme dans votre réseau 2-4-1, est de savoir si cette dérivée dit quelque chose d'utile sur la direction à prendre.

8 Pour aller plus loin

Automatic differentiation in machine learning: a survey, Atılım Güneş Baydin, Barak Pearlmutter, Alexey Radul et Jeffrey Siskind (2018).

<https://arxiv.org/abs/1502.05767>

Le panorama le plus lisible : les deux modes, l'histoire, les familles d'outils. À lire en premier.

Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation, Andreas Griewank et Andrea Walther (2008).

<https://doi.org/10.1137/1.9780898717761>

Le livre de référence ; le chapitre sur le coût du gradient et celui sur les points de reprise prolongent directement ce fascicule.

Learning representations by back-propagating errors, David Rumelhart, Geoffrey Hinton et Ronald Williams (1986).

<https://doi.org/10.1038/323533a0>

Quatre pages de *Nature* : l'article qui a donné son nom et son public à la rétropropagation.

Taylor expansion of the accumulated rounding error, Seppo Linnainmaa (1976).

<https://doi.org/10.1007/BF01931367>

Le mode inverse publié pour analyser des erreurs d'arrondi : une lecture d'historien, pour voir l'idée naître loin des réseaux.

The complexity of partial derivatives, Walter Baur et Volker Strassen (1983).

[https://doi.org/10.1016/0304-3975\(83\)90110-X](https://doi.org/10.1016/0304-3975(83)90110-X)

Le théorème du facteur trois, en quatorze pages de théorie de la complexité. Exigeant, mais l'énoncé et l'idée de la preuve se lisent.

micrograd, Andrej Karpathy (2020).

<https://github.com/karpathy/micrograd>

Le dépôt qui a inspiré le projet, avec sa vidéo « The spelled-out intro to neural networks and backpropagation » : comparez son `Value` mutable à votre graphe immuable.

Autograd mechanics, documentation de PyTorch.

<https://pytorch.org/docs/stable/notes/autograd.html>

Comment une bibliothèque industrielle enregistre sa bande, et ce que signifient `requires_grad`, `detach` et les hooks.

Nx.Defn, documentation de Nx 1.0.

<https://hexdocs.pm/nx/1.0.0/Nx.Defn.html>

La page de `defn`, `grad` et `value_and_grad` : la version Elixir du graphe construit à l'avance, celle du chapitre 7.

9 Une question pour la route

Notre perte était un scalaire, et le mode inverse obtenait ses dix-sept dérivées en une passe. Imaginez maintenant une fonction à dix-sept entrées et mille sorties, par ex. un réseau qui prédit mille pixels : combien de passes faut-il au mode inverse pour obtenir toutes les dérivées de toutes les sorties, et combien au mode direct ? Lequel choisiriez-vous, et que devient la réponse si l'on inverse les nombres, mille entrées pour dix-sept sorties ? Enfin, peut-on mélanger les deux modes pour obtenir, en une passe avant suivie d'une passe arrière, la dérivée seconde d'une perte dans une direction donnée ?

Pour toute question, vous pouvez me contacter par email.