17.4 Programmation différentiable et apprentissage d'EDO

Ici on va introduire une alternative pour l'apprentissage des réseaux de type OdeNet. Cela va nous permettre d'introduire la programmation différentiable qui va bien au dela de l'apprentissage d'EDO.

17.4.1 OpD vs DpO

L'approche OdeNet consiste à écrire le gradient d'une fonction cout qui dépend de la solution au temps final ou a un ensemble de temps par rapport aux paramètres de l'EDO. En pratique cela fait intervenir le gradient de la solution par rapport aux paramètres. Sachant que la fonction qui relie les paramètres à la solution n'étant pas explicite le calcul de gradient est complexe. Pour effectuer ce calcul les Odenet propose de calculer le gradient au niveau continue avec ce qu'on appelle la ***La rétro-propagation continue*** (celle introduite au chapitre précédent) ou ***la méthode d'adjoint***. Plus globalement on parle de méthode de type ***Optimiser puis Discrétiser*** car le calcul de gradient se fait au niveau continue de l'EDO avant qu'on utilise un flot numérique. Une autre approche consiste à ***Discrétiser puis Optimiser***. Il s'agit de discrétiser l'EDO puis de calculer son gradient. Une fois discrétisée et codée la résolution d'une EDO revient a un programme informatique enchainant les fonctions simples. Dans ce cadre pour calculer le gradient de la solution par rapport aux paramètres il faut être capable de différentier automatiquement à travers notre programme. Pour cela l'outil essentiel est la ***différentiation automatique*** qui est l'outils caché derrière les logiciels d'apprentissage comme Pytorch, Tensorflow ou Jax. Cela permet de calculer la dérivée automatiquement une sucession de fonction dans un programme. Coder de façon a ce que cette différentiation automatique soit utilisable dans l'ensemble du code s'appelle la ***Programmation différentiable***. Cela permet notamment d'optimiser n'importe qu'elle partie du code ou d'étudier sa sensibilité aux entrées.

On va dans un premier introduire les concepts important pour la différentiation automatique de programme puis on discutera plus précisement le cas des EDO. Ces notes sont principalement basés sur le superbe document [1.45].

17.4.2 Rappels d'algèbre et d'analyse

On va commencer par des rappels d'outils sur la différentiation et d'algèbre linéaires fondamentaux pour ce chapitre.

Définition 17.37. Dérivées directionnelle et partielle d'une fonction scalaire. On se donne une fonction $f : \mathbb{R}^d \to \mathbb{R}$, la dérivée directionnelle de f en $\boldsymbol{x}$ dans la direction $\boldsymbol{v}$ est donnée par

$$\partial f(\boldsymbol{x})[\boldsymbol{v}] := \lim_{\delta \to 0} \frac{f(\boldsymbol{x} + \delta \boldsymbol{v}) - f(\boldsymbol{x})}{\delta}$$

sous condition que la limite existe. On parle de dérivée partielle notée $\partial_i f(\boldsymbol{x})$ si $\boldsymbol{v} = \boldsymbol{e}_i$ le ième vecteur de la base canonique.

Définition 17.38. Gradient. On se donne une fonction $f : \mathbb{R}^d \to \mathbb{R}$ différentiable en chaque point $\boldsymbol{x}$. Le ***gradient*** est le vecteur des dérivées partielles

$$\nabla f(\boldsymbol{x}) := \begin{pmatrix} \partial_1 f(\boldsymbol{x}) \\ \vdots \\ \partial_d f(\boldsymbol{x}) \end{pmatrix} = \begin{pmatrix} \partial f(\boldsymbol{x})\left[\boldsymbol{e}_1\right] \\ \vdots \\ \partial f(\boldsymbol{x})\left[\boldsymbol{e}_d\right] \end{pmatrix}$$

On peut relier la dérivée directionnelle et le gradient par la relation:

$$\partial f(\boldsymbol{x})[\boldsymbol{v}] = \sum_{i=1}^{d} v_i \partial f(\boldsymbol{x})\left[\boldsymbol{e}_i\right] = \langle \boldsymbol{v}, \nabla f(\boldsymbol{x}) \rangle$$

Maintenant on va définir les notions équivalentes pour les fonctions à valeurs vectorielles.

Définition 17.39. Dérivées directionnelle et partielle d'une fonction vectorielle. On se donne une fonction $\boldsymbol{f} : \mathbb{R}^d \to \mathbb{R}^p$ définie par $\boldsymbol{f}(\boldsymbol{x}) = (f_1(\boldsymbol{x}), \ldots, f_p(\boldsymbol{x}))$ la dérivée directionnelle de $\boldsymbol{f}$ en $\boldsymbol{x}$ dans la direction $\boldsymbol{v}$ est donnée par

$$\partial \boldsymbol{f}(\boldsymbol{x})[\boldsymbol{v}] := \lim_{\delta \to 0} \begin{pmatrix} \frac{f_1(\boldsymbol{x}+\delta \boldsymbol{v}) - f_1(\boldsymbol{x})}{\delta} \\ \vdots \\ \frac{f_p(\boldsymbol{x}+\delta \boldsymbol{v}) - f_p(\boldsymbol{x})}{\delta} \end{pmatrix}$$

sous condition que la limite existe. On parle de dérivée partielle notée $\partial_i f(\boldsymbol{x})$ si $\boldsymbol{v} = \boldsymbol{e}_i$ le ième vecteur de la base canonique.

Définition 17.40. Jacobienne. La ***Jacobienne*** d'une fonction vectorielle

différentiable $f : \mathbb{R}^d \to \mathbb{R}^p$ en $\boldsymbol{x}$ est donnée par la matrice des dérivées partielles de chaque fonction scalaire:

$$\boldsymbol{\partial f}(\boldsymbol{x}) := \begin{pmatrix} \partial_1 f_1(\boldsymbol{x}) & \dots & \partial_d f_1(\boldsymbol{x}) \\ \vdots & \ddots & \vdots \\ \partial_1 f_p(\boldsymbol{x}) & \dots & \partial_d f_p(\boldsymbol{x}) \end{pmatrix} \in \mathbb{R}^{p \times d}$$

On a aussi

$$\partial \boldsymbol{f}(\boldsymbol{x})[\boldsymbol{v}] = \sum_{i=1}^{q} v_i \partial_i f(\boldsymbol{x}) = \boldsymbol{\partial f}(\boldsymbol{x})\boldsymbol{v} \in \mathbb{R}^p$$

Le produit entre la Jacobienne et une vecteur $\boldsymbol{v}$ permet de mesure la variation de la fonction dans la direction $\boldsymbol{v} \in \mathbb{R}^d$ d'entrée. On peut aussi mesurer la variation de la fonction dans une direction de sortie $\boldsymbol{u} \in \mathbb{R}^p$. Cela revient à calculer le gradient de la fonction

$$\langle \boldsymbol{u}, \boldsymbol{f} \rangle(\boldsymbol{x}) := \langle \boldsymbol{u}, \boldsymbol{f}(\boldsymbol{x}) \rangle \in \mathbb{R}$$

En développant $\boldsymbol{u}$ sur une base $\boldsymbol{u} = \sum_{j=1}^{p} u_j \boldsymbol{e}_j \in \mathbb{R}^p$ on peut, en utilisant la linéarité du gradient par

$$\nabla \langle \boldsymbol{u}, \boldsymbol{f} \rangle(\boldsymbol{x}) = \sum_{j=1}^{p} u_j \nabla f_j(\boldsymbol{x}) = \boldsymbol{\partial f}(\boldsymbol{x})^\top \boldsymbol{u} \in \mathbb{R}^d$$

Si on détaille le calcul composante par composante on obtient

$$\nabla_i \langle \boldsymbol{u}, f \rangle(\boldsymbol{x}) = \left[\boldsymbol{\partial} f(\boldsymbol{x})^\top \boldsymbol{u} \right]_i = \lim_{\delta \to 0} \frac{\langle \boldsymbol{u}, f\left(\boldsymbol{x} + \delta \boldsymbol{e}_i\right) - f(\boldsymbol{x}) \rangle}{\delta}$$

On voit que la ième composante $\nabla \langle \boldsymbol{u}, \boldsymbol{f} \rangle(\boldsymbol{x})$ revient à la variation dans la direction $\boldsymbol{e}_i$ projeté dans la direction $\boldsymbol{u}$.

La règle de la différentiation en chaine est un des outils les plus fondamentaux en différentiation automatique. On a déjà vu son utilisation dans le calcul du gradient des réseaux de neurones.

Pour construire nos outils de programmation différentiable on a besoin de pouvoir mesurer les variations infinitésimales le long des directions d'entrées ou de sortie. On a vu rapidement que ces notions étaient encodées par des applications linéaires reliées à la Jacobienne. On va maintenant formaliser ses notions dans les espaces Euclidiens

Pour les vecteurs on va se murir du produit scalaire usuel. Pour les matrices on se muni du produit scalaire suivant:

$$\langle A, B \rangle_{\mathcal{M}_{n,m}(\mathbb{R})} = Tr(A^t B).$$

On va maintenant introduire une notion fondamentale pour la suite.

Définition 17.41. Opérateur adjoint d'une application linéaire. On se donne deux espaces Euclidiens $(\mathcal{E}, \langle \cdot, \cdot \rangle_{\mathcal{E}})$ et $(\mathcal{F}, \langle \cdot, \cdot \rangle_{\mathcal{F}})$ ***l'adjoint*** d'une application linéaire $l : \mathcal{E} \to \mathcal{F}$ est l'unique application linéaire $l^* : \mathcal{F} \to \mathcal{E}$ tel que $\boldsymbol{v} \in \mathcal{E}$ et $\boldsymbol{u} \in \mathcal{F}$ satisfasse:

$$\langle l(\boldsymbol{v}), \boldsymbol{u} \rangle_{\mathcal{F}} = \langle \boldsymbol{v}, l^*(\boldsymbol{u}) \rangle_{\mathcal{E}}$$

Dans le cas $l(\boldsymbol{v}) = A\boldsymbol{v}$ on obtient $l^*(\boldsymbol{u}) = A^t \boldsymbol{u}$ par dualité du produit scalaire.

On va pouvoir introduire les deux outils qui seront très utilisés dans la suite.

Définition 17.42. Produit Jacobienne-Vecteur (JVP). Pour $\boldsymbol{f} : \mathcal{E} \to \mathcal{F}$, l'application linéaire $\partial \boldsymbol{f}(\boldsymbol{x}) : \mathcal{E} \to \mathcal{F}$, qui relie $\boldsymbol{v}$ à

$$\partial \boldsymbol{f}(\boldsymbol{x})[\boldsymbol{v}] = \boldsymbol{\partial f}(\boldsymbol{x})\boldsymbol{v}$$

est appelé ***Produit Jacobienne Vecteur (JVP)***. La fonction:

$$\partial \boldsymbol{f} : \mathcal{E} \to (\mathcal{E} \to \mathcal{F})$$

est une fonction qui va $\mathcal{E}$ à l'espace des applications linéaire $\mathcal{L}(\mathcal{E}, \mathcal{F})$.

Définition 17.43. Produit Vecteur-Jacobienne (VJP). Pour $\boldsymbol{f} : \mathcal{E} \to \mathcal{F}$, l'adjoint $\partial \boldsymbol{f}^*(\boldsymbol{x}) : \mathcal{F} \to \mathcal{E}$, du JVP est appelé ***Produit Vecteur-Jacobienne (VJP)***. Il satisfait:

$$\nabla \langle \boldsymbol{u}, \boldsymbol{f} \rangle_{\mathcal{F}}(\boldsymbol{w}) = \boldsymbol{\partial f}^*(\boldsymbol{w})\boldsymbol{u}$$

avec $\langle \boldsymbol{u}, \boldsymbol{f} \rangle_{\mathcal{F}}(\boldsymbol{x}) := \langle \boldsymbol{u}, \boldsymbol{f}(\boldsymbol{x}) \rangle_{\mathcal{F}}$ La fonction:

$$(\partial \boldsymbol{f})^* : \mathcal{E} \to (\mathcal{F} \to \mathcal{E})$$

est une fonction qui va $\mathcal{E}$ à l'espace des applications linéaire $\mathcal{L}(\mathcal{F}, \mathcal{E})$.

Maintenant faut étendre les précédentes définitions aux cas ou les entrées et

sorties sont multiples. Par exemple pour les réseaux de neurones ce sont des fonctions qui dépendent de leurs entrées et des poids du réseaux.

Définition 17.44. JVP et VJP pour les fonctions à plusieurs entrées. On considère une fonction différentiable $\boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{f}(\boldsymbol{x}_1, \dots, \boldsymbol{x}_M)$ avec $\boldsymbol{f} : \mathcal{E} = \mathcal{E}_1 \times \cdots \times \mathcal{E}_M \to \mathcal{F}$, L'application JVP dans la direction $\boldsymbol{v} = (\boldsymbol{v}_1, \dots, \boldsymbol{v}_M) \in \mathcal{E}$ est donnée par

$$\begin{aligned}\partial f(\boldsymbol{x})[\boldsymbol{v}] &= \partial \boldsymbol{f}(\boldsymbol{x}_1, \dots, \boldsymbol{x}_M)[\boldsymbol{v}_1, \dots, \boldsymbol{v}_M] \in \mathcal{F} \\ &= \sum_{i=1}^{M} \partial_i f(\boldsymbol{x}_1, \dots, \boldsymbol{x}_M)[\boldsymbol{v}_i]\end{aligned}$$

L'application VJP dans la direction $\boldsymbol{u}$ est donnée par

$$\begin{aligned}\partial \boldsymbol{f}(\boldsymbol{x})^*[\boldsymbol{u}] &= \partial \boldsymbol{f}(\boldsymbol{x}_1, \dots, \boldsymbol{w}_M)^*[\boldsymbol{u}] \in \mathcal{E} \\ &= \left(\partial_1 f(\boldsymbol{x}_1, \dots, \boldsymbol{x}_M)^*[\boldsymbol{u}], \dots, \partial_M f(\boldsymbol{x}_1, \dots, \boldsymbol{x}_M)^*[\boldsymbol{u}]\right)\end{aligned}$$

Définition 17.45. JVP et VJP pour des fonctions a plusieurs sorties. On considère une fonction différentiable $\boldsymbol{f}(\boldsymbol{x}) = (\boldsymbol{f}_1(\boldsymbol{w}), \dots, \boldsymbol{f}_T(\boldsymbol{w}))$, de la forme $\mathcal{E} \to \$\$\mathcal{F}$ et $f_i : \mathcal{E} \to \mathcal{F}_i$ avec $\mathcal{F} := \mathcal{F}_1 \times \cdots \times \mathcal{F}_T$. L'application JVP dans la direction $\boldsymbol{v} \in \mathcal{E}$ est donnée par

$$\partial \boldsymbol{f}(\boldsymbol{x})[\boldsymbol{v}] = (\partial \boldsymbol{f}_1(\boldsymbol{x})[\boldsymbol{v}], \dots, \partial \boldsymbol{f}_T(\boldsymbol{x})[\boldsymbol{v}]) \in \mathcal{F}$$

L'application VJP dans la direction $\boldsymbol{u} = (\boldsymbol{u}_1, \dots, \boldsymbol{u}_T)$ est donnée par

$$\begin{aligned}\partial \boldsymbol{f}(\boldsymbol{x})^*[\boldsymbol{u}] &= \partial \boldsymbol{f}(\boldsymbol{x})^*[\boldsymbol{u}_1, \dots, \boldsymbol{u}_T] \in \mathcal{E} \\ &= \sum_{i=1}^{T} \partial \boldsymbol{f}_i(\boldsymbol{x})^*[\boldsymbol{u}_i]\end{aligned}$$

Définition 17.46. Dérivée en chaine. On se donne deux fonctions $\boldsymbol{f} : \mathcal{E} \to \mathcal{F}$ et $\boldsymbol{g} : \mathcal{F} \to \mathcal{G}$ globalement différentiable. La composition $\boldsymbol{g} \circ \boldsymbol{f}$ est différentiable. Le ***produit Jacobienne-Vecteur (JVP)*** de la composition est donnée par:

$$\boldsymbol{\partial}(\boldsymbol{g} \circ \boldsymbol{f})(\boldsymbol{x}) = \boldsymbol{\partial g}(\boldsymbol{f}(\boldsymbol{x}))\boldsymbol{\partial} f(\boldsymbol{x}) \tag{17.13}$$

Le ***produit Vecteur-Jacobienne (VJP)*** de la composition est donnée par:

$$\boldsymbol{\partial}(\boldsymbol{g} \circ \boldsymbol{f})^*(\boldsymbol{x}) = \boldsymbol{\partial f}^*(\boldsymbol{x})\boldsymbol{\partial g}^*(\boldsymbol{f}(\boldsymbol{x})) \tag{17.14}$$

Si $\mathcal{G} = \mathbb{R}$ on obtient

$$\nabla(\boldsymbol{g} \circ \boldsymbol{f})(\boldsymbol{x}) = \boldsymbol{\partial f}(\boldsymbol{x})^\top \nabla \boldsymbol{g}(\boldsymbol{f}(\boldsymbol{x})) \tag{17.15}$$

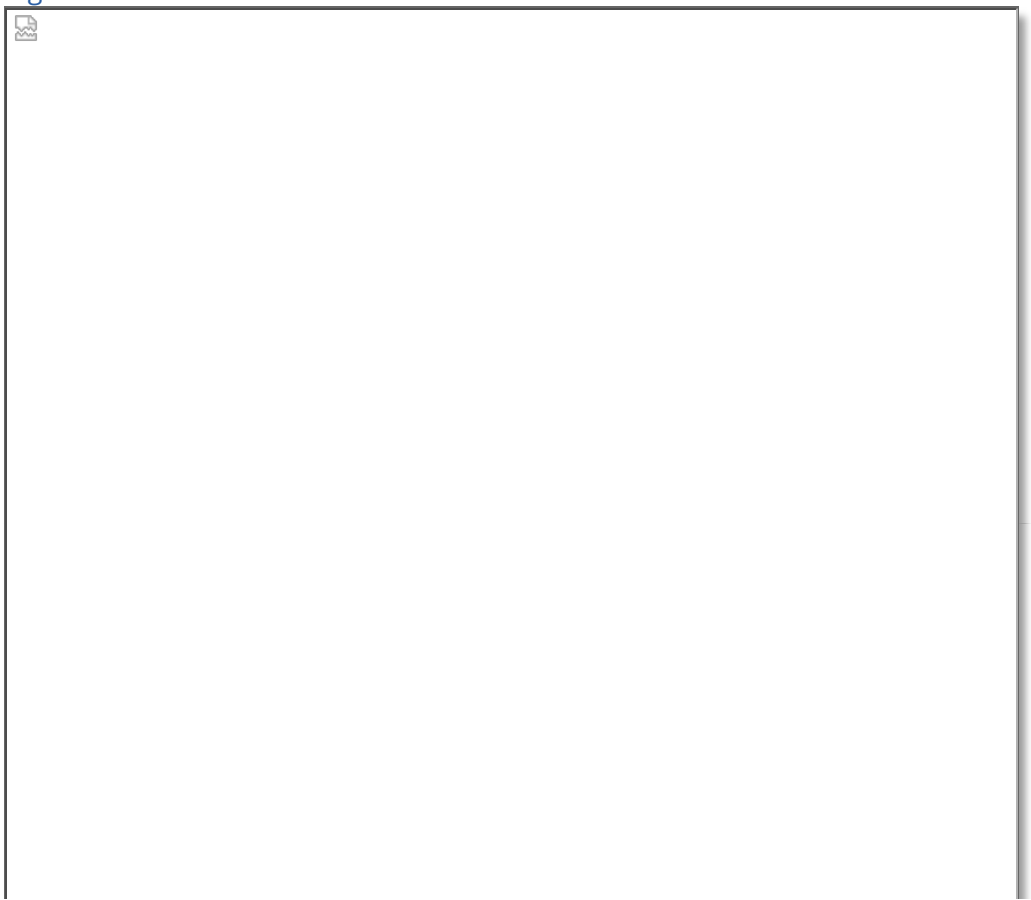
La dérivée à la chaine est l'outil central de la rétro-propagation et de la différentiation automatique. Cette définition générale sera très utile par la suite.

17.4.3 Programmes paramétrisés et différentiation

17.4.3.1 Programme paramétrisés et représentation

Comme dit précédemment, l'outil central pour la différentiation automatique va être la dérivée en chaine qui permet de dériver une composition de fonction. Avant de l'introduire son application a un programme informatique on va regarder comme on représente un programme mathématiquement.

Si on considère un programme qui est une succession de fonction informatique et donc une composition de fonction mathématique comme sur l'image Figure 17.47



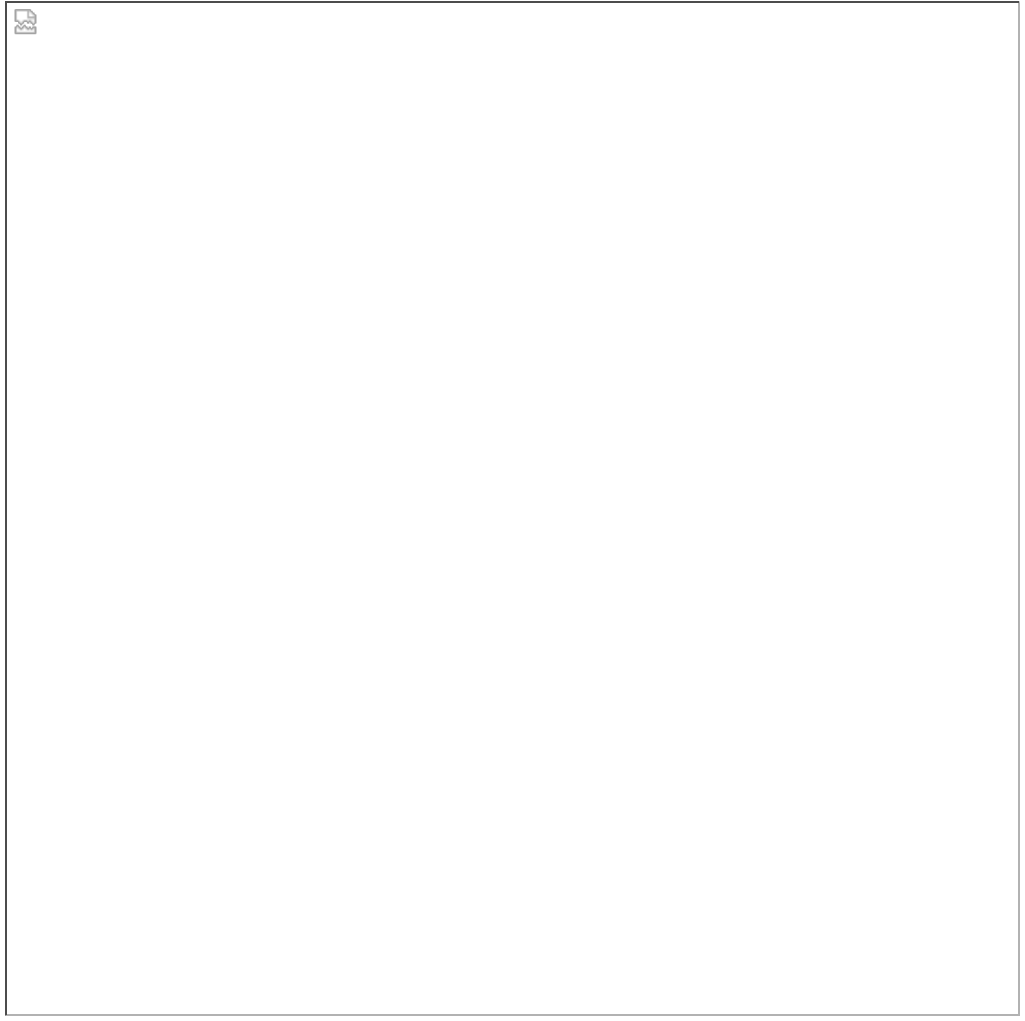


Figure 17.47. Exemple de programme séquentiel. Image issue de [1.45].

En pratique, un programme ne suit pas une structure aussi simple. En effet on peut avoir une fonction a plusieurs entrées dont les données sont calculées par plusieurs fonctions antérieures. On peut donner l'exemple Figure 17.48.

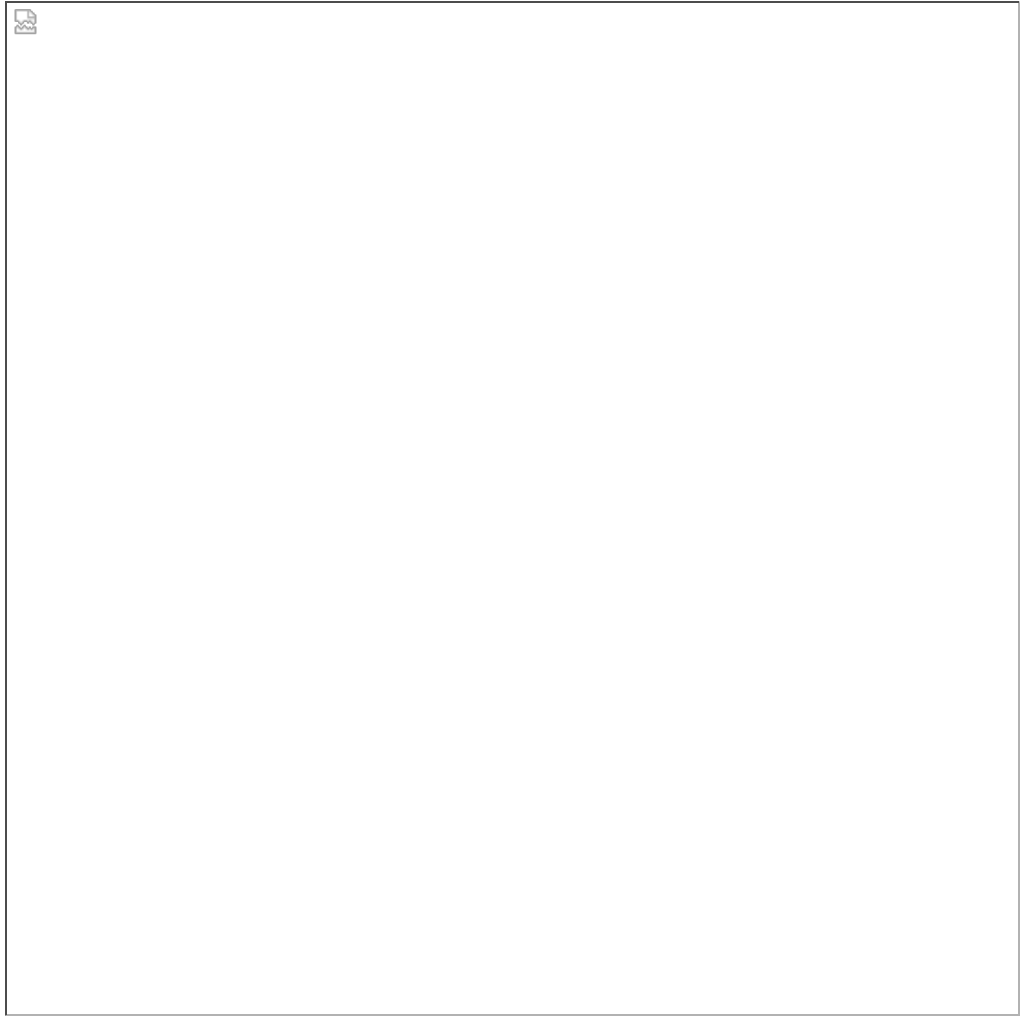


Figure 17.48. Exemple de programme non séquentiel. Image issue de [1.45].

En pratique pour représenter un programme on va utiliser un ***graphe orienté acyclique***. Cette structure a été introduite dans le cours de rappel Section 2.4. Chaque noeud représentera uen fonction du programme chaque arête (i, j) représentera la dépendance entre les sorties de la fonction i et les entrée de la fonction j. Afin de définir l'ordre dans lequel les fonctions doivent être évaluées on a besoin choisir une relation d'ordre sur les noeuds du graphe. En général on utilise l'ordre ***topologique***. Il est définit par $i \leq j$ si il existe aucun chemin de j à i. Cela revient à dire que ***chaque sommet apparaît bien avant ses successeurs***. Cela permet d'assurer qu'aucune fonctio n'est appelée avant que ses sonnées d'entrées soit calculées. En prenant cette façon de représenter un programme, l'execution d'un programme revient à l'algorithme suivant.

Algorithm 17.49. Execution d'un programme.

- ***Données :*** *les fonctions* $f_1, .. f_K$ *dans l'ordre topologique du graphe*
- ***Entrées*** *: état* $\boldsymbol{x}_0$.
- $\forall k = 1 < K$:
  - *On récupère la liste des noeuds parents du noeud* k:

$$i_1, ... i_{p_k} = Parent(k)$$

  - *On récupère les données associées* $\boldsymbol{x}_{i_1}, \ldots, \boldsymbol{x}_{p_k}$.
  - *On calcul* $\boldsymbol{x}_k = \boldsymbol{f}_k(\boldsymbol{x}_{i_1}, \ldots, \boldsymbol{x}_{p_k})$.
- *On renvoit* $\boldsymbol{x}_K$.

Ce formalisme peut aussi fonctionner si on a des fonctions qui ont des multiples entrées ou sorties il suffit de les concatener toutes ensemble et ensuite les fonctions filtreront utiles ou non.

Cette description des programmes nous permettra de mettre en place la différentiation automatique. Avant cela on va montrer comme différentier les briques élementaires des programmes informatiques.

17.4.3.2 Différentiation des structure des contrôles

Les structure de contrôles (instruction conditionnelle et boucle) sont la bases des programmes informatiques, on va donc commencer par regarder comme différentier ses outils.

On va commencer par les opérateurs de comparaison comme $<$. On peut l'écrire comme des opérateur binaire de la forme

$$y = O(x_1, x_2)$$

avec $y \in \{0, 1\}$. Pour donner une définition mathématique de ses opérateurs on va rééintroduire la fonction de Heaviside.

Définition 17.50. Fonction de Heaviside. La fonction de Heaviside est donnée par

$$H(x) = \begin{cases} 1, & x \geq 0 \\ 0, & x < 0 \end{cases}$$

La fonction de Heaviside nous permet de redéfinir les opérateurs de comparaison.

- Opérateur $\geq$ donné par $Ge(x_1, x_2) = H(u_1 - u_2)$
- Opérateur $\leq$ donné par $Le(x_1, x_2) = H(u_2 - u_1)$
- Opérateur $=$ donné par $Eq(x_1, x_2) = Ge(x_1, x_2)Ge(x_2, x_1) = H(u_1 - u_2)H(u_2 - u_1)$
- Opérateur ! $=$ donné par $Neq(x_1, x_2) = 1.0 - H(u_1 - u_2)H(u_2 - u_1)$

La fonction de Heaviside est différentiable partout sans en zero et sa dérivée est nulle partout. Par conséquent la dérivée ne va pas apporter des informations. Pour palier a ses deux défauts l'idée de remplacer la fonction de heaviside par une fonction régularisée.

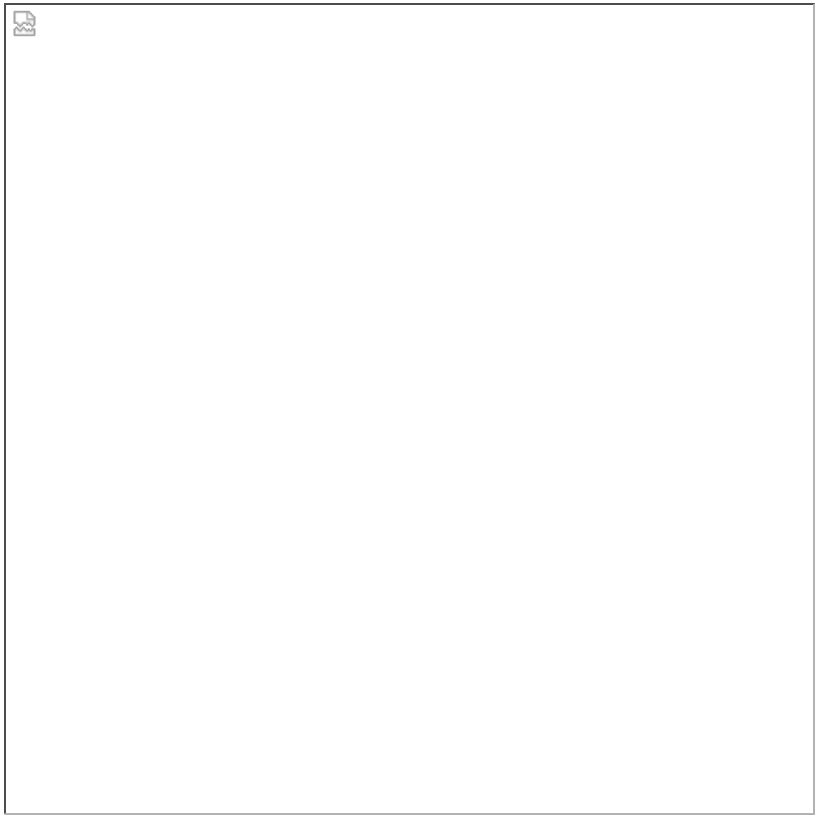


Figure 17.51. fonction sigmoide pour plusieurs σ

La fonction la plus naturelle est la fonction ***sigmoide*** très utilisée en apprentissage. Elle est donnée par

$$sig_\theta(x) = \frac{1}{1 + e^{-\frac{x}{\sigma}}}$$

Sur la figure Figure 17.51 on voit que lorsque $\sigma \to 0$ la fonction sigmoide tends vers la fonction de Heaviside. En pratique on peut donc définir des opérateurs de comparaison régularisés en renplacant dans les définitions précédentes par une ***fonction sigmoide***.

On va maintenant introduire l'opérateur d'égalité. On voit que la stratégie pour obtenir un un opérateur différentiable on relaxe l'opérateur en lui demandant de renvoyer $y \in [0,1]$ au lieu de $y \in \{0,1\}$. La première question est donc de savoir comment relaxer l'opération d'égalité. L'égalité peut être vue comme la limite d'une mesure de ***similarité***.

Définition 17.52. Fonction de similarité. une fonction de similarité S entre deux données x_1 et x_2 est une fonction maximale en $x_1 - x_2$. On peut définir une fonction de similarité assez générale à partir de fonction noyau Définition 3.68 par la formule

$$S(x_1, x_2) = \frac{k(x_1, x_2)}{\sqrt{k(x_1, x_1)}\sqrt{k(x_2, x_2)}} \tag{17.16}$$

Si on prend un noyau invariant par translation $k(x - y)$ on obtient

$$S(x_1, x_2) = \frac{k(x_1 - x_2)}{k(0)} \tag{17.17}$$

Afin de régulariser l'opérateur d'égalité on va donc le remplace par une fonction de similarité S_σ qui va tendre vers l'opérateur égal lorsque $\sigma \to 0$

▸ **Exemple 17.53.**

On donne ici deux noyaux qui peuvent définir des fonctions de similarités:

$$k_\sigma(t) = exp\left(-\frac{-t^2}{\sigma}\right)$$

et

$$k_\sigma(t) = sech\left(\frac{t}{\sigma}\right)$$

avec $sech(x) = \frac{2}{e^x + e^{-x}}$. On les retrouve sur les figures Figure 17.54 et Figure 17.55.

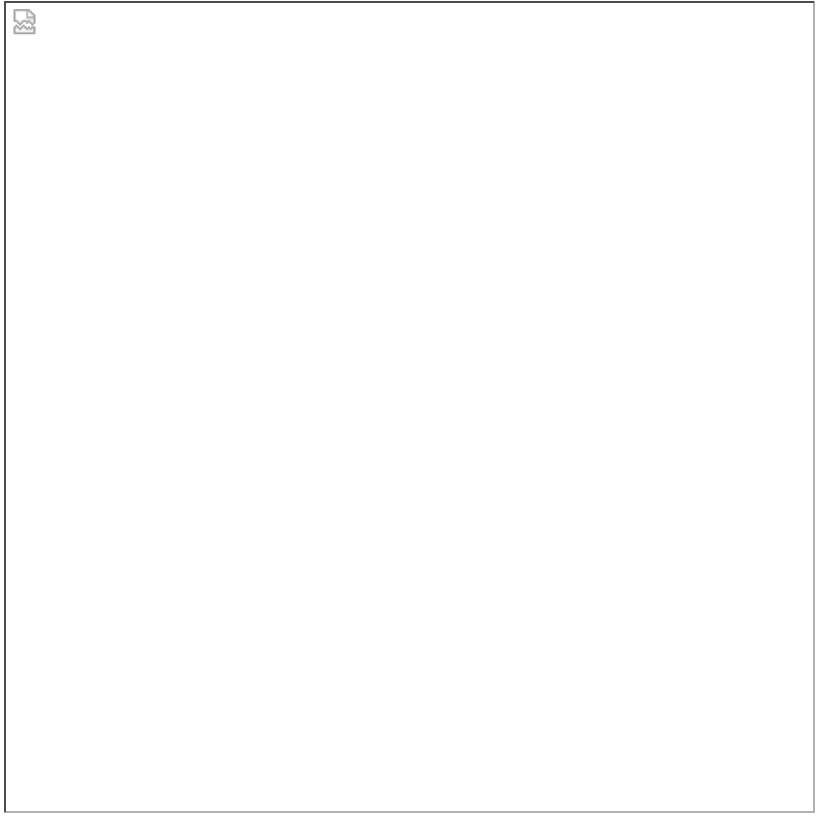


Figure 17.54. fonction de similarité Gaussienne.

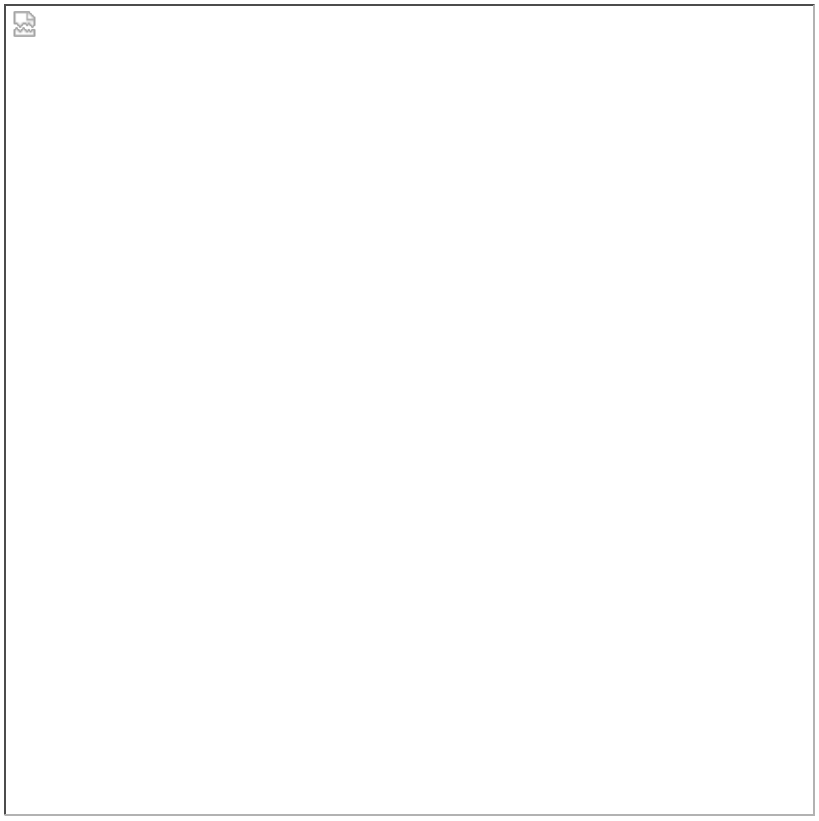


Figure 17.55. fonction de similarité Sech.

Evidemment une fois qu'on a définir une version régularisée de l'opérateur égal par une fonction de similarité de la forme (17.17) on peut définir aussi l'opérateur ***différent !=***.

Après les opérateurs de comparaison on va maintenant considérer les opérateurs logiques: ***et, ou, non*** ainsi que les généralisations au cas non binaire. Il s'agit d'opérateur travaillant sur des booléens $y \in \{0,1\}$. On va commencer par les rappeler

$$\begin{aligned}
\text{and}\,(y_1,y_2) &:= \begin{cases} 1 & \text{si } y_1 = y_2 = 1 \\ 0 & \text{sinon} \end{cases} \\
\text{or}\,(y_1,y_2) &:= \begin{cases} 1 & \text{si } 1 \in \{y_1,y_2\} \\ 0 & \text{sinon} \end{cases} \\
\text{not}(y) &:= \begin{cases} 0 & \text{si } y = 1 \\ 1 & \text{si } y = 0 \end{cases}
\end{aligned}$$

les opérateurs de comparaison retourne normalement des booléens donc dans $\{0,1\}$. Pour dériver les opérateurs de comparaisons on utiliser des versions régularisées qui renvoit des doubles dans $[0,1]$. Cela revient donc a dire que un boolean devient une réel entre $[0,1]$ très probablement proche de zéro ou de 1. Cela permet d'obtenir facilement des versions régulières de nos opérateurs logiques

Définition 17.56. Opérateurs logiques régularisés. Soit $y_1, y_2 \in [0,1]$. alors les opérateurs logiques deviennent des opérateurs de la forme:

$$[0,1] \times [0,1] \to [0,1]$$

définis par

$$\text{and}\,(y_1,y_2) = y_1 y_2, \quad \text{or}\,(y_1,y_2) = y_1 + y_2 - y_1 y_2$$

$$\text{not}(y) = 1 - y$$

et

$$\text{any}(y_1,\dots y_n) = \Pi_{i=1}^n y_i, \quad \text{any} = 1 - \Pi_{i=1}^n (1-y_i)$$

Ses opérateurs de comparaisons et logiques ont des interpretation probabiliste. En effet on remplace nos boolean par des lois de probabilités logistique. Maintenant qu'on a introduit ses outils fondamentaux de comparaison et logiques on va pouvoir introduire la dérivation des instuction conditionnnelles comme mes "if".

une structure conditionnelle "ifelse" est définie par la fonction mathématique $ifeslse(y, \boldsymbol{x}_1, \boldsymbol{x}_2) : \{0,1\} \times E \times E \to E$ donnée par

$$\begin{aligned}
\text{ifelse}\ (y, \boldsymbol{x}_1, \boldsymbol{x}_2) &:= \begin{cases} \boldsymbol{x}_1 & \text{si } y = 1 \\ \boldsymbol{x}_2 & \text{si } y = 0 \end{cases} \\
&= y\boldsymbol{x}_1 + (1-y)\boldsymbol{x}_2
\end{aligned}$$

Cette fonction n'est pas différentielle car le comportement dépend d'un booléen. Par contre la fonction est dérivable dans chaque branche. Elle est donc facile de calculer les dérivées par rapport à $\boldsymbol{x}_1$ et $\boldsymbol{x}_2$. Elles sont données par

$$\partial_{\boldsymbol{x}_1} \text{ifelse}\ (y, \boldsymbol{x}_1, \boldsymbol{x}_2) = yI_d, \quad \partial_{\boldsymbol{x}_1} \text{ifelse}\ (y, \boldsymbol{x}_1, \boldsymbol{x}_2) = (1-y)I_d$$

Si $\boldsymbol{x}_1$, \bx$x_2$ sont donnés par des fonctions de $\boldsymbol{x}_1 = \boldsymbol{f}_1(\boldsymbol{z}_1), \boldsymbol{x}_2 = \boldsymbol{f}_2(\boldsymbol{z}_2)$ on remplace juste l'identité par les jacobiennes de ses fonctions. Le problème vient de la différentiation par rapport à y. On repart d'un cas particulier

$$\text{ifelse}\ (g(\boldsymbol{z}_1), \boldsymbol{z}_1, \boldsymbol{z}_2) := \begin{cases} \boldsymbol{f}_1(\boldsymbol{z}_1) & \text{si } g(\boldsymbol{z}_1) \geq 0 \\ \boldsymbol{f}_2(\boldsymbol{z}_2) & \text{sinon} \end{cases}$$
$$= H(g(\boldsymbol{z}_1))\boldsymbol{f}_1(\boldsymbol{z}_1) + (1 - H(g(\boldsymbol{z}_1)))\boldsymbol{f}_2(\boldsymbol{z}_2)$$

Maintenant calculons la dérivée de cette fonction par rapport à $\boldsymbol{z}_1$:

$$\partial_{\boldsymbol{x}_1} \text{ifelse } = (\partial_{\boldsymbol{z}_1} g)(\boldsymbol{z}_1)H^{'}(g(\boldsymbol{z}_1))\boldsymbol{f}_1(\boldsymbol{z}_1) + H(g(\boldsymbol{z}_1))\partial_{\boldsymbol{z}_1}\boldsymbol{f}_1(\boldsymbol{z}_1) + (1 - (\partial_{\boldsymbol{z}_1} g)(\boldsymbol{z}_1)H$$

Puisque la dérivée de la fonction de Heaviside H est nulle on obtient

$$\partial_{\boldsymbol{x}_1} \text{ifelse } = H(g(\boldsymbol{z}_1))\partial_{\boldsymbol{z}_1}\boldsymbol{f}_1(\boldsymbol{z}_1) + \boldsymbol{f}_2(\boldsymbol{z}_2)$$

et donc on voit que la dérivée de g n'influe nullement la dérivée $\partial_{\boldsymbol{x}_1}$ ifelse ce qui pose problème. Pour cette raison il faut aussi régulariser le ***"ifelse"***. En pratique il suffit d'utiliser des opérateurs de comparaison régularisés (utilisant la fonction sigmoide), des opérateurs logiques régularisés Définition 17.56 permettra d'obtenir des ***"ifelse"*** différentiable si on utilise

$$y\boldsymbol{x}_1 + (1 - y)\boldsymbol{x}_2$$

avec $y \in [0, 1]$. Par exemple dans l'exemple précédent is suffit de remplacer $H(g(\boldsymbol{z}_1))$ par $sig_\theta(g(\boldsymbol{z}_1))$.

Notamment dans le cadre des opérateurs logique il peut être assez fréquent d'utiliser les fonctions ***Argmax*** et ***Argmin***. De la mêmem façon on utilise des versions régularisées appelées ***SoftAgrmax*** et ***SoftAgrmin***

Définition 17.57. Fonction SoftArgmax et SoftArgmmin. La fonction Softargmax d'un vecteur $\boldsymbol{x} \in \mathbb{R}^d$ est donnée par

$$\text{softargmax}(\boldsymbol{x}) := \frac{\exp(\boldsymbol{x})}{\sum_{i=1}^{d} \exp(x_i)}$$

et le Softargmin est donné par

$$\text{softargmin}(\boldsymbol{x}) = \text{softargmax}(-\boldsymbol{x})$$

Résumé.
On a principalement étudier des opérateurs qui travaillent sur des booléens $\{0, 1\}$ ce qui ses opérations non différentiables. Globalement la stratégie consite a remplacer les booléen par des réels dans $[0, 1]$ avec des fonction qui ne produise pas (comme le $\geq$) ui ne produisent pas des booleen mais des réels dans $[0, 1]$ qui se concentrent vers 0 ou vers 1.

Maintenant qu'on a traiter les opérateurs de comparaison et logique et les structures conditionnelles on va introduire le traitement des boucles. Pour les boucles ***for*** la situation est assez simple. En effet une boucle "for" peut se voir comme une composition de fonctions. Si chaque fonction est différentiable la boucle sera différentiable. Différentier une boucle "for" est donc similaire a différentier un programme. Ce sera introduit dans la sous section suivante. Pour les boucles ***while*** la situation est plus compliqué car le nombre d'itération depend d'une structure conditionnelle. On expliquera pas en détails les difficultés mais pour différentier à travers une boucle "while" on va régulariser la structure conditionnelle de test et dans le test on va mettre une condition d'arrêt suplémentaire avec un nombre maximal d'opérateur. On peut le faire avec opérateur ***égal*** régularisé.

17.4.3.3 Différentiation automatique

Maintenant qu'on a vu comment on modélisait un programme avec un graphe puis comment on différentiait les structure de bases comment les opérateurs binaires, les structures conditionnelles et les boucles. Maintenant on va introduire les outils fondamentaux de différentiation automatique qui vont permettre de différentier à travers un programme donc un ensemble de fonction.

On part d'une séquence simple de fonctions:

$$\begin{aligned} \boldsymbol{x}_0 &\in \mathcal{E}_0 \\ \boldsymbol{x}_1 &= \boldsymbol{f}_1(\boldsymbol{x}_0) \in \mathcal{E}_1 \\ &\vdots \\ \boldsymbol{x}_K &= \boldsymbol{f}_K(\boldsymbol{x}_{K-1}) \in \mathcal{E}_K \\ f(\boldsymbol{x}_0) &= \boldsymbol{x}_K \end{aligned}$$

En utilisant la règle de la dérivée en chaine (17.13) on obtient:

$$\boldsymbol{\partial f}(\boldsymbol{x}_0) = \boldsymbol{\partial f}_K(\boldsymbol{x}_{K-1})\boldsymbol{\partial f}_{K-1}(\boldsymbol{x}_{K-2})\ldots\boldsymbol{\partial f}_2(\boldsymbol{x}_1)\boldsymbol{\partial f}_1(\boldsymbol{x}_0)$$

On voit donc la que la jacobienne de la transformation totale est le produit des jacobiennes de chacune des transformations. Ce calcul peut être très couteux car on va effectuer K produits entre des matrices. En pratique on utilise rarement la Jacobienne. On veut souvent calculer la jacobienne dans une direction donc multiplier par un vecteur d'entrée $\boldsymbol{v}$. La formule va nous donner une approche ou on calcul, au fur et à mesure, nos fonctions f_i on va aussi calculer leurs jacobiennes et appliquer l'application linéaire associée au résultat

de l'application linéaire de la fonction précedente.

Algorithm 17.58. Différentiation automatique en mode avant (forward).

- ***Données :*** *les fonctions* $f_1, .. f_K$.
- ***Entrées*** *: état* $\boldsymbol{x}_0 \in \mathcal{E}_0$ *et direction* $\boldsymbol{v} \in \mathcal{E}_0$
- *On initialise* $\boldsymbol{t}_0 = \boldsymbol{v}$
- $\forall k = 1 < K$
  - *On calcul* $\boldsymbol{x}_k = \boldsymbol{f}_k(\boldsymbol{x}_{k-1}) \in \mathcal{E}_k$
  - *On calcul* $\boldsymbol{t}_k = \partial \boldsymbol{f}_k(\boldsymbol{x}_{k-1})[\boldsymbol{t}_{k-1}] \in \mathcal{E}_k$
- *On renvoit* $\boldsymbol{f}(\boldsymbol{x}_0) = \boldsymbol{x}_K$ *et* $\partial \boldsymbol{f}(\boldsymbol{x}_0)[\boldsymbol{v}] = \boldsymbol{t}_K$.

On retrouve sur la figure Figure 17.59 un schéma expliquant l'algorithme.

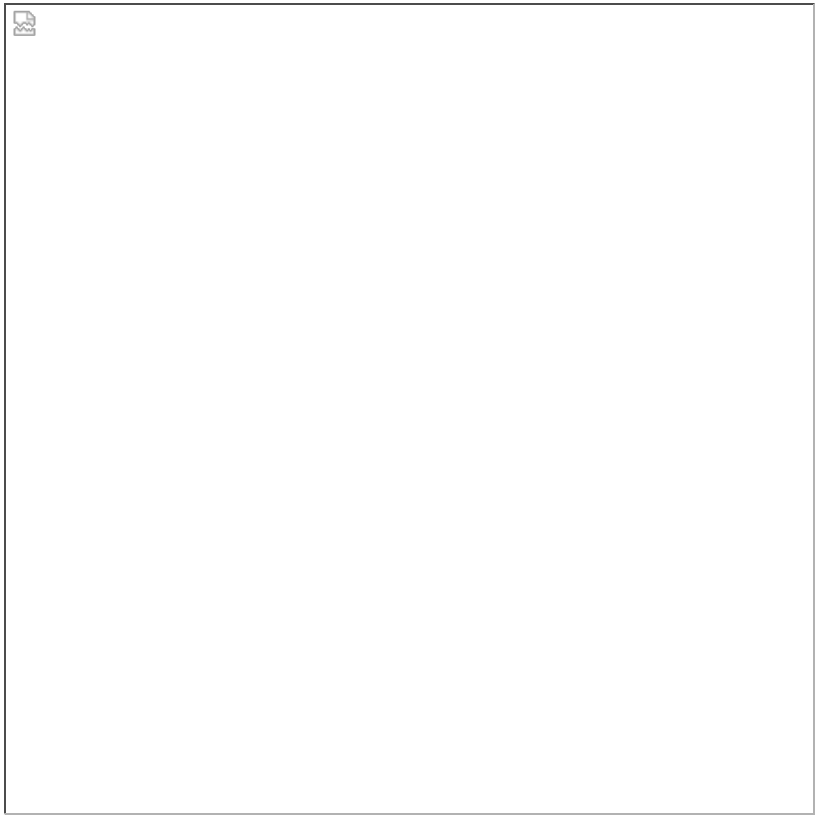


Figure 17.59. principe de l'autodifférentiation avant. Image issue du livre [1.45].

On souhaite maintenant essayer d'estimer l'empreinte mémoire de cet algorithme c'est à dire la quantité d'informations maximal qui sera stocker lors de l'auto-différentation avant. Si on regarde juste l'évaluation de l'empreinte mémoire de la fonction $\boldsymbol{f}$ donnée par la composition de nos K fonctions elle est juste celle du plus grand $\boldsymbol{x}_k$ car chaque fonction utilise uniquement le résultat de la précédente. Maintenant on va regarder ce qui se passe pour notre algorithme. En pratique on voit qu'on stocke uniquement deux objets $\boldsymbol{x}_k$ et $\boldsymbol{t}_k$ donc la l'empreinte maximal en mémoire est celle des plus gros objets de la séquence.

Maintenant on va regarder l'exemple ou on veut calculer le gradient d'une fonction de cout appliquée à la sortie de notre fonction composée.

$$\nabla_{\boldsymbol{x}_0} L\left(\boldsymbol{f}\left(\boldsymbol{x}_0\right)\right)$$

avec $L : \mathcal{E}_K \to \mathbb{R}$. Ce cadre apparait quand lorsqu'on veut calculer le gradient d'une fonction de coût appliquée au résultat de f. La formule de la dérivée en chaine nous donne:

$$\nabla_{\boldsymbol{x}_0} L\left(\boldsymbol{f}\left(\boldsymbol{x}_0\right)\right) = \left(\partial \boldsymbol{f}^*\left(\boldsymbol{x}_0\right)\right) \nabla_{\boldsymbol{x}} L\left(\boldsymbol{f}\left(\boldsymbol{x}_0\right)\right)$$

En pratique on voit donc qu'on cherche à calculer le plus souvent un terme de type VJP:

$$\partial \boldsymbol{f}^*(\boldsymbol{x})[\boldsymbol{u}] = \partial \boldsymbol{f}^*(\boldsymbol{x})\boldsymbol{u}$$

avec $\boldsymbol{v} \in \mathcal{E}$ un vecteur. En utilisant la formule de la dérivée en chaine on va donc chercher à calculer

$$\partial \boldsymbol{f}^*\left(\boldsymbol{x}_0\right) = \partial \boldsymbol{f}_1^*\left(\boldsymbol{x}_0\right) \partial \boldsymbol{f}_2^*\left(\boldsymbol{x}_1\right) \ldots \partial \boldsymbol{f}_{K-1}^*\left(\boldsymbol{x}_{K-2}\right) \partial \boldsymbol{f}_K^*\left(\boldsymbol{x}_{K-1}\right)$$

Ce calcul va donner lieu a un deuxième algorithme appelé l'auto-différentiation rétrograde.

Algorithm 17.60. Différentiation automatique en mode rétrograde (backward).

- ***Données :*** *les fonctions* $f_1, .. f_K$.
- ***Entrées*** *: état* $\boldsymbol{x}_0 \in \mathcal{E}_0$ *et direction* $\boldsymbol{u} \in \mathcal{E}_K$
- $\forall k = 1$ *à* K
  - *On calcul* $\boldsymbol{x}_k = \boldsymbol{f}_k(\boldsymbol{x}_{k-1}) \in \mathcal{E}_k$
- *On ininialise* $\boldsymbol{r}_K = \boldsymbol{u}$
- $\forall k = K$ *vers* 1
  - *On calcul* $\boldsymbol{r}_{k-1} = \partial \boldsymbol{f}_k^*(\boldsymbol{x}_{k-1})[\boldsymbol{r}_k] \in \mathcal{E}_{k-1}$

- *On renvoit* $\boldsymbol{f}(\boldsymbol{x}_0) = \boldsymbol{x}_K$ *et* $\partial \boldsymbol{f}(\boldsymbol{x}_0)[\boldsymbol{u}] = \boldsymbol{r}_0$.

La quantité $\boldsymbol{r}_k$ qui décrit la variation de notre fonction $\boldsymbol{f}_k$ par raport a une direction de sortie est appelé ***l'état adjoint*** de $\boldsymbol{x}_k$. On retrouve sur la figure Figure 17.61 un schéma expliquant l'algorithme.

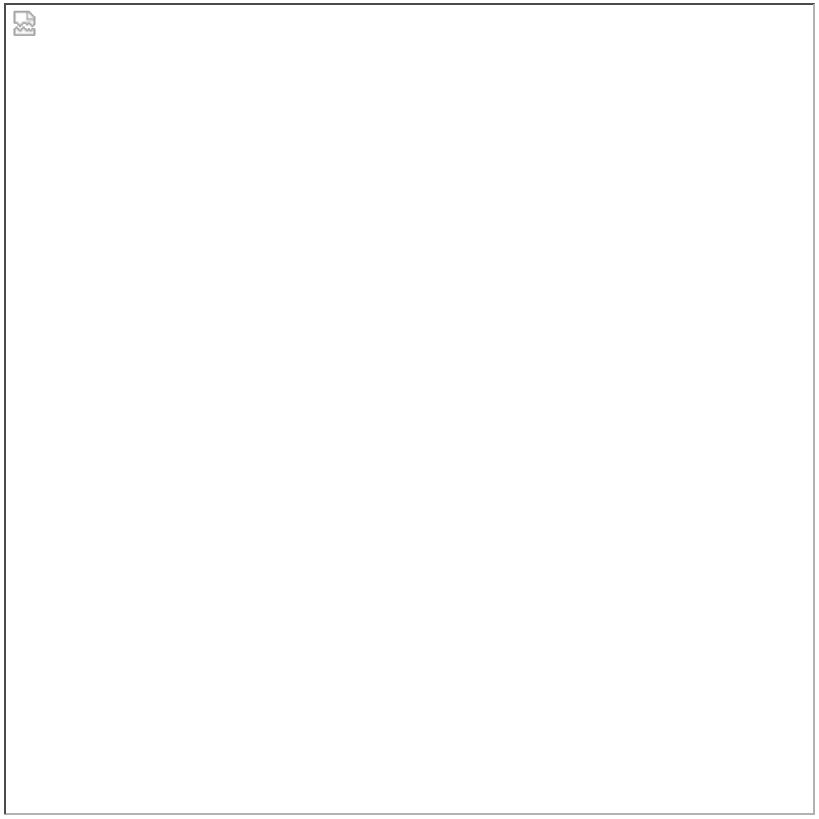
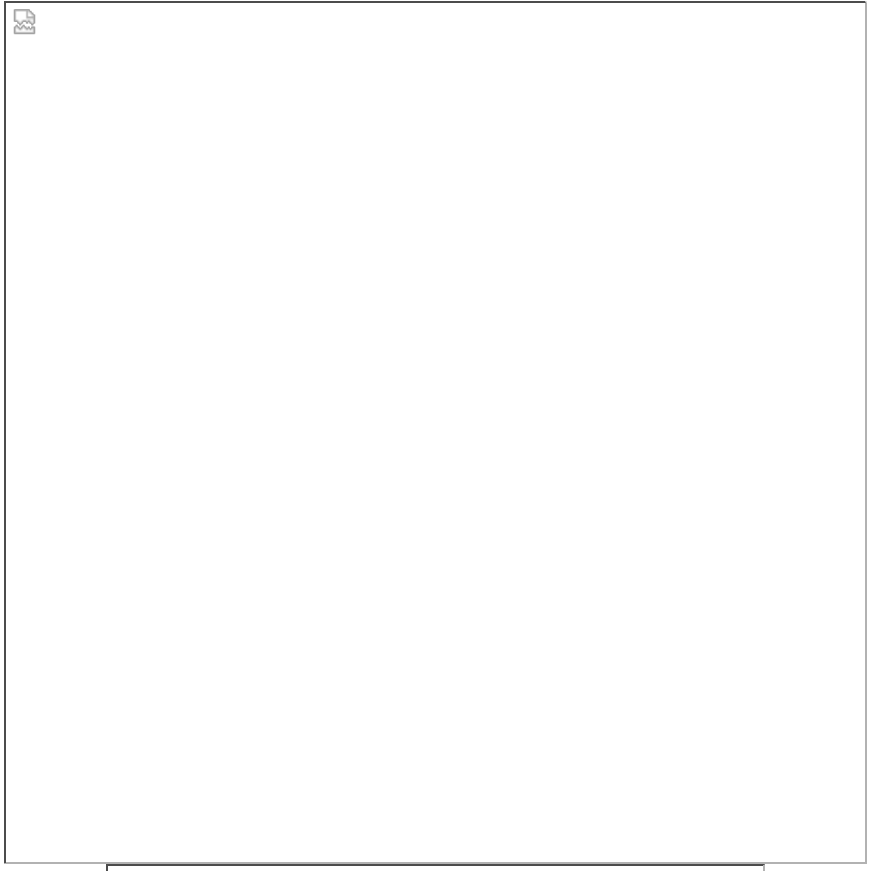


Figure 17.61. principe de l'auto-différentiation rétrograde. Image issue du livre [1.45].

L'algorithme rétrograde est celui qui est populairement utilisé pour les réseaux de neurones et pour l'apprentisage en général car il permet de calculer naturellement le gradient d'une fonction cout (17.15). Cependant la consomation mémoire est différente et plus importante que dans le cas en avant. On voit la différence sur la figure Figure 17.62



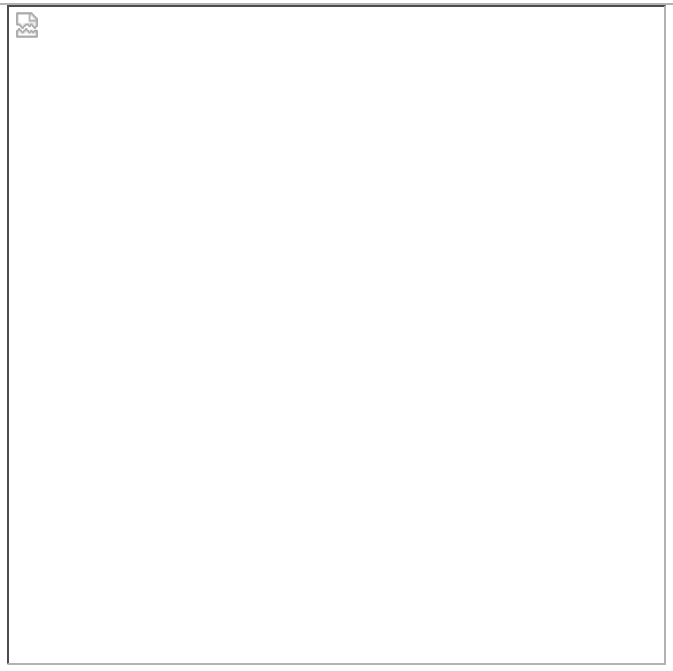


Figure 17.62. Comparaison de l'empreinte mémoire entre les deux approches

d'auto-différentiation. Gauche: en avant. Droite: rétrograde. Image issue du livre [1.45].

Maintenant on va comparer l'empreinte mémoire des deux approches pour le calcul de la jacobienne complète et pas seulement d'un produit jacobienne vecteur. On suppose qu'on a $\boldsymbol{f} : \mathbb{R}^d \to \mathbb{R}^m$. On suppose que les dimension interméfaire sont tous de taille d Dans le cas avant pour calculer la jacobienne on va calculer le produit $\boldsymbol{\partial f v}$ en prenant comme valeur de $\boldsymbol{v}$ tout les vecteurs de base $\boldsymbol{e}_i$ de $\mathbb{R}^d$. On va donc devoir faire d calcule de type JVP. Chaque appel de type JVP a comme empreinte mémoire $O(\max(d, m))$ et le cout de calcul est donné par le cout d'un produit matrice vecteur donc $\sum_{1=1}^{k-1} d^2 + dm$ ce qui donne donc un cout total pur la Jacobienne entière de de l'ordre $O(md^2 + Kd^3)$. Dans le cas arrière il faudra aussi multiplier par tous les vecteurs de base de $\mathbb{R}^m$. On devrait faire donc m calculs de type VJP. Pour la complexité de calcul elle est la même par VJP que par JVP donc on obtient $O(m^2 d + Kmd^2)$. Par contre le cout mémoire est de l'ordre $O(Kd + m)$.

▸ **Exemple 17.63.**

Si on prend un réseau avec 10 entrées et 10 neurones par couche avec 6 couches. Le mode avant nous donne une complexité mémoire en $O(10)$ et de calcul en $O(6100)$ la ou le mode rétrograde nous donnera une complexité mémoire en $O(60)$ et de calcul en $O(610)$.

Utilisation pratique.

En général on conseille le mode rétrograde si $m < d$ (fréquent en apprentissage) et le mode en avant si $m \geq d$. En effet pour le cas $d = m = 1$ les deux ont un coût CPU en $O(K)$ mais la mémoire de la métthode en avant est en $O(1)$.

Le cas d'un programme séquentiel introduit ici est suffisant pour décrire précisement l'algorithme de retro-propagation du gradient. Il suffit s'utiliser la définition de VJP pour le cas à plusieurs entrées. On va juste écrire l'algoritme de la rétropropagation du gradient d'un modèle paramétrique f_θ à plusieurs couches. Il s'agit donc d'une application du mode rétrograde.

Algorithm 17.64. Rétropropagation du gradient.

- ***Données :*** *les fonctions* $f_1, .. f_K$ *et une fonction de* L
- ***Entrées*** *: Données entrée-sortie* $(\boldsymbol{x}, \boldsymbol{y})$ *et vecteur de paramètre* $\theta = (\theta_1, \ldots, \theta_K)$
- *on initialise* $\boldsymbol{x}_0 = \boldsymbol{x}$
- $\forall k = 1$ *à* K
  - *On calcul* $\boldsymbol{x}_k = \boldsymbol{f}_k(\boldsymbol{x}_{k-1}, \theta_k) \in \mathcal{E}_k$
- *On calcule* $L(\boldsymbol{x}_k, \boldsymbol{y})$
- *On ininialise* $\boldsymbol{r}_K = \nabla_1 L(\boldsymbol{x}_k, \boldsymbol{y})$
- $\forall k = K$ *vers* 1
  - *On calcul* $(\boldsymbol{r}_{k-1}, \boldsymbol{g}_k) = \partial \boldsymbol{f}_k^*(\boldsymbol{x}_{k-1}, \theta_k)[\boldsymbol{r}_k] \in \mathcal{E}_{k-1} \times \Theta_k$
- *On renvoit la fonction de cout total et son gradient* $L(\boldsymbol{x}_K, \boldsymbol{y})$ *et* $\partial L(\boldsymbol{x}_K, \boldsymbol{y}) = (\boldsymbol{g}_1, \ldots, \boldsymbol{g}_K)$.

Dan cet algorithme le terme $\boldsymbol{g}_k$ correspond à l'adjoint des poids donc à la variation de la fonction $\boldsymbol{f}_k$ par rapport aux variations de θ_k.

Jusqu'a présent on a traité le cas d'une séquence simple. En pratique on a vu que les programmes étaient plutôt définis par des graphes que par des séquences linéaires. On va donc juste introduire les modes avant et rétrograde dans le cas plus général d'un programme défini par un graphe.

Algorithm 17.65. Différentiation automatique en mode avant pour un graphe.

- ***Données :*** *les fonctions* $f_1, .. f_K$ *dans l'ordre topologique du graphe*
- ***Entrées*** *: état* $\boldsymbol{x}_0 \in \mathcal{E}_0$ *et direction* $\boldsymbol{v} \in \mathcal{E}_0$
- *On initialise* $\boldsymbol{t}_0 = \boldsymbol{v}$
- $\forall k = 1 < K$:
  - *On récupère les noeuds parents* $i_1, \ldots i_{p_k} = pa(k)$
  - *On calcul* $\boldsymbol{x}_k = \boldsymbol{f}_k(\boldsymbol{x}_{i_1}, \ldots, \boldsymbol{x}_{i_{p_k}})$
  - *On calcul* $\boldsymbol{t}_k = \sum_{j=1}^{p_k} \partial_j \boldsymbol{f}_k(\boldsymbol{x}_{i_1}, \ldots, \boldsymbol{x}_{i_{p_k}})[\boldsymbol{t}_{i_j}]$
- *On renvoit* $\boldsymbol{f}(\boldsymbol{x}_0) = \boldsymbol{x}_K$ *et* $\partial \boldsymbol{f}(\boldsymbol{x}_0)[\boldsymbol{v}] = \boldsymbol{t}_K$.

Algorithm 17.66. Différentiation automatique en mode rétrograde pour un graphe.

- ***Données :*** *les fonctions* $f_1, .. f_K$ *donnée dans l'ordre topologique de graphe.*
- ***Entrées*** *: état* $\boldsymbol{x}_0 \in \mathcal{E}_0$ *et direction* $\boldsymbol{u} \in \mathcal{E}_K$

- $\forall k = 1$ *à* K
  - *On récupère les noeuds parents* $i_1, \dots i_{p_k} = pa(k)$
  - *On calcul* $\boldsymbol{x}_k = \boldsymbol{f}_k(\boldsymbol{x}_{i_1}, \dots, \boldsymbol{x}_{i_{p_k}})$
  - *On instancie* $l_k = \partial \boldsymbol{f}_k^*(\boldsymbol{x}_{i_1}, \dots, \boldsymbol{x}_{i_{p_k}})$
- *On ininialise* $\boldsymbol{r}_K = \boldsymbol{u}$ *et* $\boldsymbol{r}_k = 0, \quad \forall k \in \{0, K-1\}$
- $\forall k = K$ *vers* 1
  - *On récupère les noeuds parents* $i_1, \dots i_{p_k} = pa(k)$
  - *On calcule*

$$(\boldsymbol{\delta}_{i_1}, \dots, \boldsymbol{\delta}_{i_{p_k}}) = l_k[\boldsymbol{r}_k]$$

    avec $(\boldsymbol{\delta}_{i_1}, \dots, \boldsymbol{\delta}_{i_{p_k}})$ *les sorties de VJP avec plusieurs entrées définies dans Définition 17.44.*
  - *On calcule* $\boldsymbol{r}_{i_j} \leftarrow \boldsymbol{r}_{i_j} + \boldsymbol{\delta}_{i_j}, \quad \forall j \in \{1, \dots, p_k\}$
- *On renvoit* $\boldsymbol{f}(\boldsymbol{x}_0) = \boldsymbol{x}_K$ *et* $\partial \boldsymbol{f}(\boldsymbol{x}_0)[\boldsymbol{u}] = \boldsymbol{r}_0$.

Si une valeur intermédiaire $\boldsymbol{s}_k$ est utilisées par les fonctions $f_{j_1}, \dots, f_{j_{c_k}}$ ou $j_1, \dots, j_{c_k} = \text{enfant}(k)$, les dérivées par rapport à $\boldsymbol{x}_k$ va donc utiliser la somme des variations de toute les fonctions. Cela nous donne

$$\boldsymbol{r}_k := \sum_{j \in \{j_1, \dots, j_{c_k}\}} \boldsymbol{\delta}_{k,j}$$

avec $\boldsymbol{\delta}_{k,j}$ Dans l'algorithme lorqu'on a la variation d'un noeud on va la distribué dans toutes les variations $\boldsymbol{r}$ des enfants de ceux noeuds.

Une fois que vous avez une mode différentiation automatique il suffit de le coupler avec des fonctions de bases différentiable (on peut parler de fonctions primitive) pour obtenir un code différentiable. Les fonctions primitives sont les fonctions mathématiques usuelles ou les fonctions regularisées introduites précédemment comme le "et", le "=" etc. Il est important que pur ces fonctions la les modes JVP ou VJP soient définis. On a besoin aussi que les compisitions de ses fonctions primitives restent des fonctions dérivables.

Le mode retrograde qui est principalement utilisé en apprentissage et en optimisation est assez consommateur en mémoire. En effet la complexité mémoire est de l'ordre de K le nombre de fonction (noeuds du graphe de calcul). Pour limiter il existe plusieurs approches. La première approche est simple mais limité én pratique elle consiste a construire des fonctions inversible. Cela permet de calculer l'état dont on a besoin pour le mode-rétrograde juste avec l'évaluation de l'inverse.

Algorithm 17.67. Différentiation automatique en mode rétrograde avec fonctions inversibles.

- ***Données :*** *les fonctions* $f_1, .. f_K$.
- ***Entrées*** *: état* $\boldsymbol{x}_0 \in \mathcal{E}_0$ *et direction* $\boldsymbol{u} \in \mathcal{E}_K$
- *On initialise* $\boldsymbol{r}_K = \boldsymbol{u}$
- *On calcul* $\boldsymbol{x}_K = \boldsymbol{f}_K \circ \dots \circ \boldsymbol{f}_1(\boldsymbol{x}_0)$
- $\forall k = K$ *vers* 1
  - *On calcul* $\boldsymbol{x}_{k-1} = \boldsymbol{f}_{k-1}^{-1}(\boldsymbol{x}_k)$
  - *On calcul* $\boldsymbol{r}_{k-1} = \partial \boldsymbol{f}_k^*(\boldsymbol{x}_{k-1})[\boldsymbol{r}_k]$
- *On renvoit* $\partial \boldsymbol{f}(\boldsymbol{x}_0)[\boldsymbol{u}] = \boldsymbol{r}_0$.

Bien que des architectures de réseaux peuvent être inversible cette stratégie reste limitée en pratique. L'approche la plus utilisé est celle des points de contrôle.

Définition 17.68. Méthode de point de contrôle pour le mode rétrograde. La méthode des points de contrôle fonctionne en ne stockant sélectivement qu'un sous-ensemble de nœuds intermédiaires, appelés points de contrôle, et en recalculant les autres.

On va commencer par introduire le cas extrême ou l'on ne stocke aucune valeur dans le cadre d'un programme séquentiel.

Algorithm 17.69. Différentiation automatique en mode rétrograde sans stockage mémoire.

- ***Données :*** *les fonctions* $f_1, .. f_K$.
- ***Entrées*** *: état* $\boldsymbol{x}_0 \in \mathcal{E}_0$ *et direction* $\boldsymbol{u} \in \mathcal{E}_K$
- *On ininialise* $\boldsymbol{r}_K = \boldsymbol{u}$
- $\forall k = K$ *vers* 1
  - *On calcul* $\boldsymbol{x}_{k-1} = \boldsymbol{f}_{k-1} \circ \dots \circ \boldsymbol{f}_1(\boldsymbol{x}_0)$
  - *On calcul* $\boldsymbol{r}_{k-1} = \partial \boldsymbol{f}_k^*(\boldsymbol{x}_{k-1})[\boldsymbol{r}_k]$
- *On renvoit* $\partial \boldsymbol{f}(\boldsymbol{x}_0)[\boldsymbol{u}] = \boldsymbol{r}_0$.

A ajouter: méthode de checkpointing

17.4.4 Différentier à travers des fonctions implicites

Jusqu'à présent, nous avons appris à différentier à travers des fonctions explicites. En pratique, il est possible de se retrouver avec des fonctions implicites à différentier. Par exemple, si on prend un schéma d'Euler implicite en temps, la détermination du temps $n+1$ va nécessiter la résolution d'un système linéaire ou nonlinéaire dépendant de $\boldsymbol{f}_\theta$. On va donc potentiellement vouloir differentier par rapport à \theta la fonction $S : \theta \to \boldsymbol{x}^{n+1}$ ou $S : \boldsymbol{x}^{n+1}$ est la solution du schéma d'Euler implicite. Cette fonction S n'est pas explicite par rapport a θ. Pour cette raison, on va voir comme différentier des fonctions implicites.

Il peut exister plusieurs exemples:

- Les problèmes d'optimisations ou on a:

$$h(\theta) = g(\boldsymbol{x}^*(\theta), \theta), \qquad \text{avec } \boldsymbol{x}^*(\theta) = \operatorname{argmax}_{\boldsymbol{x}\in\mathcal{E}} f(\boldsymbol{x}, \theta) \quad (17.18)$$

  donc on souhaite différentier une fonction qui dépend du résultat d'un processus d'optimisation qui dépend de paramètre. Il y a donc une dépendance implicite. Prenons le cas de d'une régression linéaire Ridge de la forme

$$\theta^* = \operatorname{argmax} \left(\| \ Y - \theta X \ \|^2 + \lambda \ \| \ \theta \ \|^2 \right)$$

  Les paramètres optimaux θ^* sont une fonction implicite du paramètre de régularisation λ on peut donc l'écrire $\theta^*(\lambda)$. Si ensuite on a un crètire pour juger du meilleur paramètre λ on peut vouloir le dériver par rapport à λ pour ensuite le déterminer par une méthode de gradient. On se retrouve typiquement dans notre cas.

- La recherche de zéro d'équations nonlinéaires de la forme:

$$f(\boldsymbol{x}, \theta) = 0$$

  qui englobent notamment les schémas implicites dont on a parler précédemment.

La solution d'un problème d'optimisation ou la recherche de zéro d'une fonction est effectué en général en utilisant un algorithme itératif. Cet algorithme peut s'interpréter comme une composition de fonctions simples qui sont ou qu'on peut rendre dérivable. On peut donc a priori dériver ses algorithmes avec de la différentiation automatique (typequement rétrograde). On parle de ***Dérouler l'algorithme***.On peut voir cette approche comme une variante de ***discrétiser puis optimiser***. En pratique cela peut être très couteux car on va devoir sauvegarder les états intermédiaires comme montré précédemment. Cela peut devenir très couteux en mémoire. Une autre approche est de voir si on peut calculer directement le VJP. On pourrait voir cela comme l'approche ***optimiser puis discretiser***. On va introduire ses méthodes pour les problèmes implicites.

17.4.4.1 Différentier à travers des problèmes d'optimisation

Repartons du problème (17.18) avec $\boldsymbol{f}, \boldsymbol{g} : \mathcal{E} \times \Theta \to \mathbb{R}$. Si on dérive notre fonction h on obtient

$$\partial_\theta h(\boldsymbol{\theta}) = \partial_1 g\left(\boldsymbol{x}^\star(\boldsymbol{\theta}), \boldsymbol{\theta}\right) \partial_\theta \boldsymbol{x}^\star(\boldsymbol{\theta}) + \partial_2 g\left(\boldsymbol{x}^\star(\boldsymbol{\theta}), \boldsymbol{\theta}\right)$$

Le seul point bloquant de ce calcul est de déterminer $\partial_\theta \boldsymbol{x}^\star(\boldsymbol{\theta})$. On va maintenant introduire un cas simplifier ou le problème peut être résolu plus facilement.

On va se restreindre ici au cas $g(\boldsymbol{x}, \theta) = f(\boldsymbol{x}, \theta)$. Dans ce cas le problème devient

$$h(\theta) = f(\operatorname{argmax}_{\boldsymbol{x}\in\mathcal{E}} f(\boldsymbol{x}, \theta), \theta).$$

Hors évaluer une fonction en son argmax est équivalent au maximum donc notre problème ici devient un problème de recherche de maximum:

$$h(\theta) = \max_{\boldsymbol{x}\in\mathcal{E}} f(\boldsymbol{x}, \theta)$$

ce qui a priori sera plus simple à différentier. On cherche donc le gradient d'un problème d'optimisation paramètrique par rapport à ses paramètres.

Définition 17.70. Théorème de Danskin. Soit $\boldsymbol{f} : \mathcal{E} \times \Theta \to \mathbb{R}$ avec Θ un ensemble convexe. Posons

$$h(\theta) = \max_{\theta\in\Theta} f(\boldsymbol{x}, \theta)$$

et

$$\boldsymbol{x}^\star(\theta) := \operatorname*{argmax}_{\boldsymbol{x}\in\mathcal{E}} f(\boldsymbol{x}, \theta)$$

Si f est concave en $\boldsymbol{x}$, convexe en θ, et atteint son unique maximum en $\boldsymbol{x}^\star(\theta)$ alors la fonction h est différentiable de gradient:

$$\nabla_\theta h(\theta) = \nabla_2 f\left(\boldsymbol{x}^\star(\theta), \theta\right)$$

Le Théorème de Danskin dit finalement qu'on peut dériver simplement f comme si $\boldsymbol{x}^\star$ ne dépendait pas de θ ce qui rend le calcul très facile. Le théorème de Rockafellar nous donne le même résultat avec des hypothèses la fonction f différente.

Maintenant on va revenir a un cadre plus géénral d'optimisation sous contrainte et rappeler un résultat qui nous permettra de construire la différentation de ce problème d'optimisation par rapport aux paramètres

Proposition 17.71. *On considère le problème suivant avec* $f, g, h : \mathcal{E} \times \Theta \to \mathbb{R}$:

$$\begin{aligned} \boldsymbol{x}^\star(\theta) = \underset{\boldsymbol{x}}{\operatorname{argmin}} f(\boldsymbol{x}, \theta) \\ \text{sous les conditions que } g(\boldsymbol{x}, \theta) \leq 0 \\ h(\boldsymbol{x}, \theta) = 0 \end{aligned}$$

La solution de ce problème est donnée celle de $G(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\nu}; \theta) = 0$ *avec*

$$G(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\nu}; \theta) = \begin{bmatrix} \nabla_{\boldsymbol{x}} f(\boldsymbol{x}; \theta) + \partial_{\boldsymbol{x}} g(\boldsymbol{x}; \theta)^T \boldsymbol{\lambda} + \partial_{\boldsymbol{x}} h(\boldsymbol{x}; \theta)^T \boldsymbol{\nu} \\ \boldsymbol{\lambda} \circ g(\boldsymbol{x}; \theta) \\ h(\boldsymbol{x}; \theta) \end{bmatrix}$$

▸ **Preuve.**

On commence par appliquer les conditions de Karush-Kuhn-Tucker. \\ A compléter

Ce résultat nous montre donc que dériver la solution d'un problème d'optimisation va tout naturellement se transformer en un problème de dérivation d'une fonction implicite liée au gradient (cas sans contrainte) ou au Lagrangien (cas avec contrainte). On va maintenant traiter le cas des fonctions implicites qui était un des enjeux de cette section.

17.4.4.2 Différentier à travers des fonctions implicites

On considère une équation nonlinéaire $F(\boldsymbol{x}; \theta) = \boldsymbol{0}$ avec $\boldsymbol{F} : \mathcal{E} \times \Theta \to \mathcal{E}$. Si on considère $\boldsymbol{x}^*(\theta)$ la solution de l'équation. On cherche donc à calculer $\partial_\theta \boldsymbol{x}^*(\theta)$. Cela nous permettra de traiter un certains nombres de problème comme les produits d'optimisation introduit précédemment. On va introduire formellement la solution avant de la justifier.

On part de $\boldsymbol{F}(\boldsymbol{x}^*(\theta); \theta) = \boldsymbol{0}$ et on dérive par rapport à θ ce qui donne:

$$\partial_1 \boldsymbol{F}\left(\boldsymbol{x}^\star(\theta), \theta\right) \partial_\theta \boldsymbol{x}^\star(\theta) + \partial_2 \boldsymbol{F}\left(\boldsymbol{x}^\star(\theta), \theta\right) = \boldsymbol{0}$$

Donc la dérivée que l'on cherche doit satisfaire:

$$-\partial_1 \boldsymbol{F}\left(\boldsymbol{x}^\star(\theta), \theta\right) \partial_\theta \boldsymbol{x}^\star(\theta) = \partial_2 \boldsymbol{F}\left(\boldsymbol{x}^\star(\theta), \theta\right)$$

Pour calculer la dérivée il nous faut donc inverser $\partial_1 \boldsymbol{F}\left(\boldsymbol{x}^\star(\theta), \theta\right)$. Ceci est assuré sous certaines conditions par le théorème des fonctions implicites que l'on rappelle.

Définition 17.72. Théorème des fonctions implicites. Soit $\boldsymbol{F} : \mathcal{E} \times \Theta \to \mathcal{E}$. On suppose $\boldsymbol{F}(\boldsymbol{x}, \theta)$ une fonction continument différentiable dans le voisinage de $(\boldsymbol{x}_0, \theta_0)$ tel que $\boldsymbol{F}\left(\boldsymbol{x}_0, \theta_0\right) = \boldsymbol{0}$ et $\partial_1 \boldsymbol{F}\left(\boldsymbol{x}_0, \theta_0\right)$ soit inversible. Alors il existe un voisinage U_0 de θ_0 dans lequel il existe une fonction $\boldsymbol{x}^\star(\theta)$ tel que:

- $\boldsymbol{x}^\star\left(\theta_0\right) = \boldsymbol{x}_0$,
- $\boldsymbol{F}\left(\boldsymbol{x}^\star(\theta), \theta\right) = \boldsymbol{0}, \forall \theta \in U_0$,
- On a:

$$\partial_\theta \boldsymbol{x}^\star(\theta) = -\partial_1 \boldsymbol{F}(\boldsymbol{x}^\star(\theta), \theta)^{-1} \partial_2 \boldsymbol{F}\left(\boldsymbol{x}^\star(\theta), \theta\right).$$

Cela nosu permet d'obtenir facilement le JVp et VPJ pour les fonctions implicites.

Proposition 17.73. JVP et VJP pour les fonctions implicites. *Soit* $\boldsymbol{x}^*(\theta)$ *la solution d'une équation implicite de type* $F\left(\boldsymbol{x}, \theta\right) = \boldsymbol{0}$ *avec* $\boldsymbol{F} : \mathcal{E} \times \Theta \to \mathcal{E}$. *On pose*

$$A = -\partial_1 \boldsymbol{F}\left(\boldsymbol{x}^\star(\theta), \theta\right), \quad B = \partial_2 \boldsymbol{F}\left(\boldsymbol{x}^\star(\theta), \theta\right)$$

Le JVP $\boldsymbol{t} = \partial \boldsymbol{x}^\star(\theta) \boldsymbol{v}$ *est solution de*

$$A\boldsymbol{t} = B\boldsymbol{v}$$

et le VJP $\partial \boldsymbol{x}^\star(\theta)^* \boldsymbol{u}$ *est donné par*

$$\partial \boldsymbol{x}^\star(\theta)^* \boldsymbol{u} = B^* \boldsymbol{r}$$

avec $\boldsymbol{r}$ *solution de:*

$$A^* \boldsymbol{r} = \boldsymbol{u}$$

▸ **Preuve.**

Le JVP s'obtient immédiatement en multipliant le résultat du théorème des fonctions implicites par $\boldsymbol{v}$.\\ TO DO: Ajouter VJP

L'approche dit "déroulée" peut se voir comme l'approche DpO appliquée aux fonctions implicites la ou l'utilisation du mode VJP introduite juste au dessus peut être vu comme l'approche OpD. Dans ce cadre l'approche OpD est plus efficace car elle évite un coût mémoire important lié au "dérouler" des algorithmes de résolution comme les méthodes de point fixes ou de Newton.

17.4.4.3 Application en algébre linéaire

Dans cette section on va proposer des exemples en algébre linéaire. On va suppposer qu'on va avoir ici des matrices ou des systèmes linéaires qui dependent de paramètres θ et une fonction de coût qui va dépendre de calcul utilisant ces matrices et membres de droites. On souhaite donc dériver notre fonction de cout par rapport à θ en utilisant la différentiation de fonctions implicites.

Dans un premier exemple, on va supposer qu'on souhaite résoudre

$$\min_\theta J(\boldsymbol{x}^s), \quad \text{sous contrainte } A(\theta)\boldsymbol{x}^* = \boldsymbol{b}(\theta)$$

On souhaite donc calculer le VJP

$$\nabla_\theta J(\boldsymbol{x}^*) = (\partial^*_\theta \boldsymbol{x}^s)\nabla_{\boldsymbol{x}} J$$

ce qui est égal à la chaine de VJP

$$\nabla_\theta J(\boldsymbol{x}^*) = (\partial^*_\theta (A, \boldsymbol{b}))(\partial^*_{A,\boldsymbol{b}} \boldsymbol{x}^s)\nabla_{\boldsymbol{x}} J$$

Le VJP associé à $(\partial^*_\theta (A, \boldsymbol{b}))$ sera naturellement calculer par l'autodifférentiation appliqué aux fonctions qui construise $A, \boldsymbol{b}$ en fonction de θ. Par contre pour le calcul VJP associé à la solution $\boldsymbol{x}^s$ la situation est différente. Si on code un algorithme d'inversion comme une méthode de gradient Conjugé ou une décomposition LU vous allez devoir différentier à travers des centaines voir milliers d'appels de fonctions. Il est donc plus efficace de dériver le VJP par la méthode des fonctions implicites directement.

Proposition 17.74. VJP d'une résolution de système linéaire. *Soit une système linéaire $A\boldsymbol{x} = \boldsymbol{b}$. On nomme la solution $\boldsymbol{x}^s$ du solution du système. le vecteur de sortie est nommé $\boldsymbol{u}$. Le VJP associé à la solution est donné par*

$$\frac{\partial^* \boldsymbol{x}^s}{\partial A}\boldsymbol{u} = -\boldsymbol{r} \otimes \boldsymbol{x}^s$$

et

$$\frac{\partial^* \boldsymbol{x}^s}{\partial b}\boldsymbol{u} = \boldsymbol{r}$$

avec $A^t\boldsymbol{r} = \boldsymbol{u}$ le système adjoint.

▸ **Preuve.**

On commence par écrire notre problème sous la forme d'une équation implicite:

$$F(\boldsymbol{x}, A, \boldsymbol{b}) = A\boldsymbol{x} - \boldsymbol{b} = 0$$

Il nous fait maintenant calculer les différentielle. Pour la première c'est facile. Il est en effet immédiat que

$$\partial_1 F(\boldsymbol{x}, A, \boldsymbol{b}) = A$$

Maintenant on va chercher les différentielles par rapport à $\boldsymbol{b}$ et A.

- Différentielle par rapport à $\boldsymbol{b}$:

  $$F(\boldsymbol{x}, A, \boldsymbol{b} + \delta\boldsymbol{b}) = A\boldsymbol{x} - \boldsymbol{b} - \delta\boldsymbol{b}$$

  donc

  $$(\partial_3 F)\delta\boldsymbol{b} = F(\boldsymbol{x}, A, \boldsymbol{b} + \delta\boldsymbol{b}) - F(\boldsymbol{x}, A, \boldsymbol{b})$$

  ce qui donne

  $$(\partial_3 F)\delta\boldsymbol{b} = -\delta\boldsymbol{b}$$

  donc $(\partial_3 F) = -I_d$ et donc $(\partial_3 F)^* = -I_d$
- Différentielle par rapport à $\boldsymbol{A}$. On écrit dans un premier temps:

  $$F(\boldsymbol{x}, A + \delta A, \boldsymbol{b}) = (A + \Delta A)\boldsymbol{x} - \boldsymbol{b}$$

  On chercher donc a déterminer l'application linéaire:

  $$(\partial_2 F)\delta A : \mathcal{M}_{n,n}(\mathbb{R}) \to \mathbb{R}^n$$

  On a donc

  $$(\partial_2 F)\delta A = F(\boldsymbol{x}, A + \delta A, \boldsymbol{b}) - F(\boldsymbol{x}, A, \boldsymbol{b}) = \delta A\boldsymbol{x}$$

On va maintenant calculer le dual dont on a besoin pour appliquer les résultats sur les fonctions implicites. On cherche donc à déterminer:

$$(\partial_2 F)^* \delta A : \mathbb{R}^n \to \mathcal{M}_{n,n}(\mathbb{R}).$$

On va donc calculer

$$\langle (\partial_2 F)\delta A, \boldsymbol{y} \rangle_{\mathbb{R}^n} = \langle \delta A \boldsymbol{x}, \boldsymbol{y} \rangle_{\mathbb{R}^n}$$

On veut maintenant se ramener à un produit scalaire dans $\mathcal{M}_{n,n}(\mathbb{R})$ pour obtenir l'application linéaire duale Définition 17.41:

$$\langle \delta A \boldsymbol{x}, \boldsymbol{y} \rangle_{\mathbb{R}^n} = Tr(\delta A \boldsymbol{x} \boldsymbol{y}^t) = \langle \delta A, \boldsymbol{y} \otimes \boldsymbol{x} \rangle_{\mathcal{M}_{n,n}(\mathbb{R})}$$

on a donc

$$(\partial_2 F)^* : \boldsymbol{y} \to \boldsymbol{y} \otimes \boldsymbol{x}$$

Maintenant qu'on a obtenu nos différentielles on peut conclure avec la proposition Proposition 17.73. L'état adjoint $\boldsymbol{r}$ est solution de

$$-\partial_1 F(\boldsymbol{x}, A, \boldsymbol{b})^* \boldsymbol{r} = -A^t \boldsymbol{r} = \boldsymbol{u}$$

Ensuite on va obtenir le résultat en appliquant la suite de la formule d'une VJP Proposition 17.73:

$$\frac{\partial^* \boldsymbol{x}^s}{\partial b} \boldsymbol{u} = (\partial_3 F)^* \boldsymbol{r} = -\boldsymbol{r}$$

et

$$\frac{\partial^* \boldsymbol{x}^s}{\partial A} \boldsymbol{u} = (\partial_2 F)^* \boldsymbol{r} = \boldsymbol{r} \otimes \boldsymbol{x}.$$

En posant l'adjoint solution de $= A^t \boldsymbol{r} = \boldsymbol{u}$ on obtient le résultat final.

17.4.5 Régularisation

En construction

17.4.6 Différentiation et ODE

Maintenant que nous avons introduits les outils de différentiation automatique on peut considérer le cas des EDO. Comme introduit au début du chapitre on a vu que l'approche OdeNet revient à la méthode OpD pour le calcul du gradient d'une EDO. Utiliser la programmation différentiable pour calculer ce même gradient peut être vu comme l'apporche DpO. On va maintenant discuter l'intérêt des deux.

Un premier avantage de l'approche DpO est une fois l'ensemble du code programmé de façon différentiable on peut calculer le gradient de l'EDO, modifié le type de fonction coût, différentier les solveurs sans travail supplémentaire. Cependant si on se restreint au cadre des EDO l'approche OpD semble a première vu meilleure puisqu'on a pas a stocker la solution de l'EDO primal grace au dual augmenté de la méthode OdeNet on se retrouve avec un cout mémoire en $O(m)$ au lieu $O(mT)$ pour la méthode DpO. On va cepencent montrer avec des exemples que l'approche OpD ne marche pas toujours et donc qu'une approche DpO avec des "points de contrôles" peut être totalement pertinente.

Le fait qu'on est pas besoin de stocker la solution de l'EDO dans l'approche vient du fait qu'on peut résoudre l'EDO dans le sens inverse car le flot d'une EDO est l'inversible ce qu'on ne peut a prior pas faire avec le mode DpO car les schémas ne pas a priori réversible. Cependant la réversabilité de l'EDO n'est pas automatique. En pratique il y a des conditions de régularité sur f_θ pour avoir de la réversibilité dans un ouvert. Un première exemple est l'EDO:

$$\frac{d}{dt} z(t) = z^3(t)$$

qui admet une solution $\varphi^0(t, z_0) = \frac{z_0}{\sqrt{1-2z_0^2 t}}$ on a donc une solution qui est bien posé jusqu'au temps $t < \frac{1}{2z_0^2}$. On voit donc que l'existence de la solution et donc la reversibilité va dépendre du temps final et de l'ensemble des conditions initiales. Un second exemple est l'EDO

$$\frac{d}{dt} z(t) = -\lambda z(t)$$

dont la solution existe sur $\mathbb{R}^+$. Si on résout l'EDO dans le sens inverse on se retrouve a résoudre

$$\frac{d}{dt} z(t) = \lambda z(t)$$

Si λ est grande il s'agit d'une exponentielle fortement croissante. Pour ce problème les schémas seront instable sauf à utiliser des pas de temps infiniment petit. Ces deux exemples montrent qu'en pratique résoudre l'EDO dans le sens inverse peut s'avérer compliquer. Dans [1.46] les auteurs montrent

aussi la présence d'instabilité si $\boldsymbol{f}_\theta$ est un réseau convolutif.

Résumé.

Pour ces raisons les auteurs de [1.46] concluent que l'approche DpO couplé avec des "points de contrôles" est une solution efficace.

Une approche alternative qui pas forcement toujours efficace est d'utiliser des schémas réversibles. En effet ces schémas nous disent que

$$\Phi_{\Delta t}^{-1} = \Phi_{-\Delta t}$$

Cela permet de pouvoir calculer rapidement l'état $\boldsymbol{z}_{k-1}$ a partir de l'état $\boldsymbol{z}_k$ et donc de ne pas stocker les solutions de l'équation primale. Cette approche peut être notamment intéressante pour les EDO conservatives ou les schémas réversibles sont plus fréquents.