# Programmation différentiable et EDO

Emmanuel Franck[*],

3 octobre, 2024
**Master CMSI, M2**, Strasbourg

[*] MACARON project-team, Université de Strasbourg, CNRS, Inria, IRMA, France


# Outline

# Programmation différentiable

**Odenet**

L'approche OdeNet consiste à écrire le gradient d'une fonction cout qui dépend de la solution a un ensemble de temps par rapport aux paramètres de l'EDO.

- On doit calculer 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.
- Les Odenet permettent de calculer le gradient au niveau continue (on parle de rétro-propagation continue ou de la méthode d'adjoint).

**Optimiser puis discrétiser**

On calcul le gradient à partir de l'EDO et de l'EDO adjointe puis on discrétise les deux avec le schéma de notre choix. Méthode utilisée par OdeNet.

**Discrétiser puis optimiser**

On discrétise l'EDO puis on calculera le gradient.

# OdeNet II

- Comment différentier un schéma ?
- Dans les cas simples un schéma est une large composition de fonction simple

$$f(x;\theta) = (I_d + \Delta t \mathbf{F}_\theta) \circ .... \circ (I_d + \Delta t \mathbf{F}_\theta)$$

- Dans le pire des cas on cette composition peu contenir des fonctions implicites (issue de schémas implicites par exemple).

**Besoin**

Etre capable de différentier a travers une composition de fonctions explicites et implicites le plus variées possible

**Solution**

Programmation différentiable et différentation automatique.

- Frameworks différentiable: Torch, Jax, Tensorflow, Taichi, package Julia etc

## Dérivées directionnelle, partielle et gradient d'une fonction scalaire

On se donne une fonction $f : \mathbb{R}^d \to \mathbb{R}$, la dérivée directionnelle de $f$ en $\mathbf{x}$ dans la direction $\mathbf{v}$ est donnée par

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

sous condition que la limite existe. On parle de dérivée partielle notée $\partial_i f(\mathbf{x})$ si $\mathbf{v} = \mathbf{e}_i$ le ième vecteur de la base canonique. Le **gradient** est le vecteur des dérivées partielles

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

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

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

## Dérivées directionnelle, partielle et jacobienne

Soit $f : \mathbb{R}^d \to \mathbb{R}^p$ définie par $f(\mathbf{x}) = (f_1(\mathbf{x}), ...., f_p(\mathbf{x}))$ la dérivée directionnelle de $f$ en $\mathbf{x}$ dans la direction $\mathbf{v}$ est donnée par

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

On parle de dérivée partielle notée $\partial_i f(\mathbf{x})$ si $\mathbf{v} = \mathbf{e}_i$. La **Jacobienne** de $f : \mathbb{R}^d \to \mathbb{R}^p$ en $\mathbf{x}$ est donnée par la matrice des dérivées partielles de chaque fonction scalaire:

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

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

# Différentiabilité: adjoint et dualité

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

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

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

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

- Si on détaille on obtient:

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

# Différentiabilité: adjoint et dualité

- On a besoin de pouvoir mesurer les variations infinitésimales le long des directions d'entrées ou de sortie.
- Ces notions étaient encodées par des applications linéaires reliées à la Jacobienne. On va maintenant formaliser ses notions dans les espaces euclidiens.

**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 $\forall \mathbf{v} \in \mathcal{E}$ et $\forall \mathbf{u} \in \mathcal{F}$ satisfasse:

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

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

- A partir de la on va pouvoir introduire les deux outils essentiels.

# Différentiabilité: JVP et VJP

## Produit Jacobienne-Vecteur (JVP)

Pour $\mathbf{f} : \mathcal{E} \to \mathcal{F}$, l'application linéaire $\partial\mathbf{f}(\mathbf{x}) : \mathcal{E} \to \mathcal{F}$, qui relie $\mathbf{v}$ à la dérivée directionnelle de $\mathbf{f}$

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

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

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

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

## Produit Vecteur jacobienne (VJP)

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

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

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

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

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

**JVP/VJP multiple entrées**

On considère une fonction différentiable $\mathbf{f}(\mathbf{x}) = \mathbf{f}(\mathbf{x}_1, \dots, \mathbf{x}_M)$ avec $\mathbf{f} : \mathcal{E} = \mathcal{E}_1 \times \cdots \times \mathcal{E}_M \to \mathcal{F}$, L'application JVP dans la direction $\mathbf{v} = (\mathbf{v}_1, \dots, \mathbf{v}_M) \in \mathcal{E}$ est donnée par

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

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

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

**JVP/VJP multiple sorties**

On considère une fonction différentiable $\mathbf{f}(\mathbf{x}) = (\mathbf{f}_1(\mathbf{w}), \dots, \mathbf{f}_T(\mathbf{w}))$, 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 $\mathbf{v} \in \mathcal{E}$ est donnée par

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

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

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

# Différentiabilité: dérivée en chaine

## Dérivée en chaine

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

$$\partial(\mathbf{g} \circ \mathbf{f})(\mathbf{x}) = \partial\mathbf{g}(\mathbf{f}(\mathbf{x}))\partial f(\mathbf{x})$$

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

$$\partial(\mathbf{g} \circ \mathbf{f})^*(\mathbf{x}) = \partial\mathbf{f}^*(\mathbf{x})\partial\mathbf{g}^*(\mathbf{f}(\mathbf{x}))$$

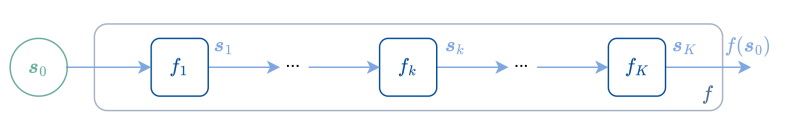
Si $\mathcal{G} = \mathbb{R}$ on obtient

$$\nabla(\mathbf{g} \circ \mathbf{f})(\mathbf{x}) = \partial\mathbf{f}(\mathbf{x})^\top \nabla\mathbf{g}(\mathbf{f}(\mathbf{x}))$$

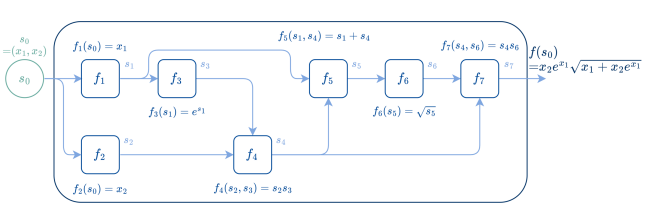
- La dérivée à la chaine est l'outil central de la rétro-propagation et de la différentiation automatique.

# 1er représentation d'un programme

- L'outil central pour la différentiation automatique va être la dérivée en chaine qui permet de dériver une composition de fonction.
- On va formaliser un programme de façon mathématique.



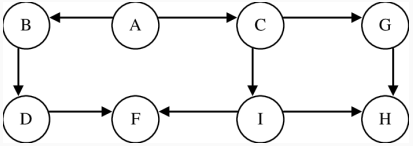

- En pratique c'est plutôt:



**Graphe**

Soit $V$ un nombre fini de sommets. Un graphe est définie par une paire $G = [V, E]$ avec $V$ l'ensembles des noeuds et $E$ l'ensemble des arêtes ou un élément de $E$ est une paire d'élément de $V$ noté $(x, y)$ avec $x, y \in V$.

- Un graphe est dit orienté si on peut aller d'un sommet $i$ vers un sommet $j$ sans que l'inverse soit automatiquement possible.
- Un chemin entre $i$ et $j$ est l'ensemble de arêtes qui permet de un noeud $i$ avec un noeud $j$.
- Un graphe acyclique est un graphe ou il existe aucun chemin entre un noeud et lui même.



# Execution programme et différentiation

**Programme**

On peut raisonnablement supposé que un programme est un Graphe Acyclique Orienté (DAG). Les noeuds sont les fonctions et les arêtes représente le lien entre une fonction qui renvoie une valeur et celle qui va utiliser cette valeur en entrée.

- On a besoin d'une relation d'ordre sur le graphe pour définir un ordre d'execution.

**Ordre topologique**

Il est définit par $i \leqslant j$ si il existe aucun chemin de $j$ à $i$. Cela revient à dire que chaque sommet apparaît bien avant ses successeurs. Pas unique

- Exemple:
  `https://www.techiedelight.com/fr/topological-sorting-dag/`

# Représentation d'un programme

**Algorithm 14.49. 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* $k$:

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

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

- Avant de voir comment différentier un programme, on va regarder comment différentier les briques de bases.
- Pour les fonctions mathématiques pas de soucis.
- Structure de contrôle (boucle ? if ? operator logique ou de comparaison ?)

# Opérateur de comparaison

- 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 $y \in \{0, 1\}$ un booléen.

**Fonction de Heaviside**

La fonction de Heaviside est donnée par

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

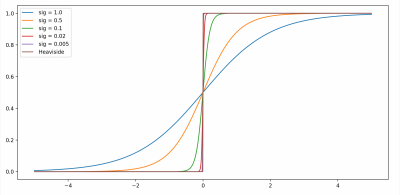
- La fonction de Heaviside permet de définir des opérateurs de comparaison
  - Opérateur $\geqslant$ donné par $Ge(x_1, x_2) = H(u_1 - u_2)$
  - Opérateur $\leqslant$ donné par $Le(x_1, x_2) = H(u_2 - u_1)$
  - Opérateur $=$ donné par $Eq(x_1, x_2) = Ge(x_1, x_2)Ge(x_2, x_1) = H(u_1 - u_2)H(u_2 - u_1)$
  - Opérateur $!=$ donné par $Neq(x_1, x_2) = 1.0 - H(u_1 - u_2)H(u_2 - u_1)$

# Régularisation I

- 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.
- On utilise une version régularisée: La sigmoide donnée par

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

- $\lim_{\sigma \to 0} sig_\theta(x) = H(x)$




- On a plus un booléen en sortie $y \in \{0, 1\}$ par un réel $y \in [0, 1]$ .
- En pratique on peut donc définir des opérateurs de comparaison régularisés en renplacant dans les définitions précédentes par une fonction sigmoide.

# Régularisation II

- Maintenant on va passer à l'opérateur d'égalité $=$ qui renvoi un booléen.
- On peut voir l'égalité comme une limite d'une fonction de similarité.

**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 invariante par translation par la formule

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

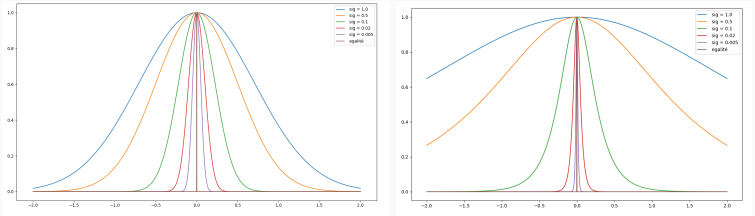
- Exemple:

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

avec $sech(x) = \frac{2}{e^x + e^{-x}}$.

# Opérateurs logiques

- Les noyaux:




- On va donc remplacer le "=" par une similarité avec $\sigma << 1$.
- Opérateurs logiques: et, ou, non. Il s'agit d'opérateur travaillant sur des booléens $y \in \{0, 1\}$. On va commencer par les rappeler

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

# Opérateurs logiques II

**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

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

- On voit que l'idée est de remplacer les booléens par des des réels dans $[0,1]$ ce qui est possible avec nos opérateurs $<, >, =$ régularisés.

# Structure conditionnelle

- La structure conditionnelle classique "ifelse" est une fonction de la forme $ifeslse(y, \mathbf{x}_1, \mathbf{x}_2) : \{0,1\} \times E \times E \to E$ donnée par

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

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

- Cette fonction n'est pas différentielle car le comportement dépend d'un booléen. Par contre la fonction est dérivable dans chaque branche. On a

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

- Le problème vient de la différentiation par rapport à $y$. On repart d'un cas particulier

$$\text{ifelse}\ (g(\mathbf{x}_1), \mathbf{x}_1, \mathbf{x}_2) := \begin{cases} \mathbf{f}_1(\mathbf{x}_1) & \text{si } g(\mathbf{x}_1) \geqslant 0 \\ \mathbf{f}_2(\mathbf{x}_2) & \text{sinon} \end{cases}$$

$$= H(g(\mathbf{x}_1))\mathbf{f}_1(\mathbf{x}_1) + (1 - H(g(\mathbf{x}_1)))\mathbf{f}_2(\mathbf{x}_2)$$

# Structure conditionnelle II

- Maintenant calculons la dérivée de cette fonction par rapport à $\mathbf{x}_1$:

  $$\partial_{\mathbf{x}_1} \text{ ifelse } = (\partial_{\mathbf{x}_1} g)(\mathbf{x}_1) H'(g(\mathbf{x}_1))\mathbf{f}_1(\mathbf{x}_1) + H(g(\mathbf{x}_1))\partial_{\mathbf{x}_1}\mathbf{f}_1(\mathbf{x}_1) + (1 - (\partial_{\mathbf{x}_1} g)(\mathbf{x}_1) H'(g(\mathbf{x}_1)))\mathbf{f}_2(\mathbf{x}_2)$$

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

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

- La dérivée de $g$ n'influe nullement la dérivée $\partial_{\mathbf{x}_1}$ ifelse ce qui pose problème.
- **Solution**: Remplacer les Heaviside par la Sigmoide.

**Argmax et régularisation**

La fonction Softmax (régularisation argmax) d'un vecteur $\mathbf{x} \in \mathbb{R}^d$ est donnée par

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

- Les boucles sont une composition de fonctions dont on peut traiter ca comme un sous problème. Par contre pour le "while", il faut régularisé la condition de test et ajouter un nombre d'itération maximale.

# Différentiation automatique

- On part d'une séquence simple de fonctions:

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

- En utilisant la règle de la dérivée en chaine on obtient:

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

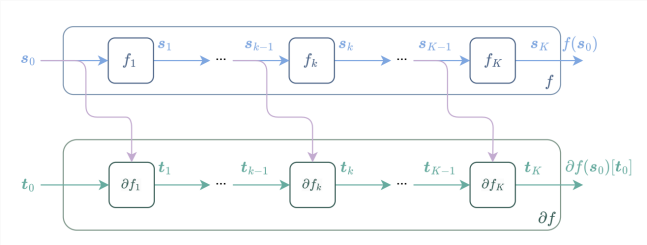
- On voit donc la que la Jacobienne de la transformation totale est le produit des Jacobiennes de chacune des transformations.
- On veut souvent calculer la Jacobienne dans une direction donc multiplier par un vecteur d'entré $\mathbf{v}$.
- Cela va nous donner un premier algorithme de différentiation automatique.

# Différentiation avant

- Algorithme et schéma

**Algorithm 14.58. 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* $\partial \boldsymbol{f}(\boldsymbol{x}_0)[\boldsymbol{v}] = \boldsymbol{t}_K$.



# Différentiation rétrograde

- 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.

$$\nabla_{\mathbf{x}_0} L(\mathbf{f}(\mathbf{x}_0))$$

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 $f$. On a:

$$\nabla_{\mathbf{x}_0} L(\mathbf{f}(\mathbf{x}_0)) = (\partial \mathbf{f}^*(\mathbf{x}_0)) \nabla_{\mathbf{x}} L(\mathbf{f}(\mathbf{x}_0))$$

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

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

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

$$\partial \mathbf{f}^*(\mathbf{x}_0) = \partial \mathbf{f}_1^*(\mathbf{x}_0) \partial \mathbf{f}_2^*(\mathbf{x}_1) \dots \partial \mathbf{f}_{K-1}^*(\mathbf{x}_{K-2}) \partial \mathbf{f}_K^*(\mathbf{x}_{K-1})$$

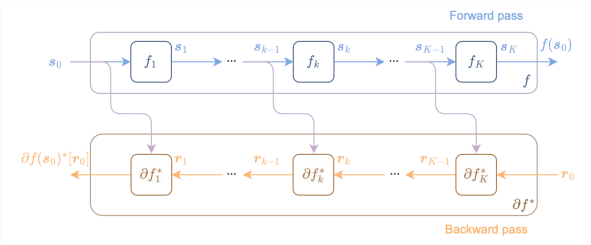
Ce calcul va donner lieu a un deuxième algorithme appelé l'auto-différentiation rétrograde.

# Différentiation rétrograde 2

- Algorithme et schéma

**Algorithm 14.60. 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* $\partial \boldsymbol{f}(\boldsymbol{x}_0)[\boldsymbol{u}] = \boldsymbol{r}_0$.



# Comparaison

- Comparons l'empreinte mémoire des deux approches pour le calcul de la Jacobienne complète.
- On a $\mathbf{f} : \mathbb{R}^d \to \mathbb{R}^m$. On suppose que les dimensions intermédiaire sont de taille $d$.
- Cas avant:
  - On va calculer le produit $\partial \mathbf{fv}$ en prenant comme valeur de$\mathbf{v}$ tout les vecteurs de base $\mathbf{e}_i$ 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 des produits matrice-vecteur $\sum_{1=1}^{K-1} d^2 + dm$
  - Cela donne en temps de calcul $O(md^2 + Kd^3)$.
- Cas arrière:
  - On multiple par tous les vecteurs de base de $\mathbb{R}^m$.
  - On devrait faire donc $m$ calculs de type VJP. Pour la complexité de calcul on a $O(m^2 d + Kmd^2)$. Par contre le cout mémoire est de l'ordre $O(Kd + m)$.

**En pratique**

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

# Retro-propapation de gradient

- A partir de du mode rétrograde on peut retrouver la rétro-propagation du gradient

**Algorithm 14.64. 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* $\partial L(\boldsymbol{x}_K, \boldsymbol{y}) = (\boldsymbol{g}_1, \ldots, \boldsymbol{g}_K)$.

- Les algorithmes précédents se généralise au cas ou le programme est un DAG et pas une séquence linéaire.
- Si on compose notre code de fonctions primitives différentiables on peut dériver toute composition avec le mode avant ou rétrograde.

## Défaut du mode rétrograde

Le coût mémoire est en $O(K)$ avec $K$ le nombre de noeud du graphe de calcul.

- première approche: couche réversible. Il s'agit de fonction dont on connait explicitement l'inverse ou bien des fonctions dont l'inverse est rapidement calculable.

**Algorithm 14.67. 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* $\boldsymbol{x}_K = \boldsymbol{f}_K \circ \ldots \circ \boldsymbol{f}_1(\boldsymbol{x}_0)$
- $\forall k = K$ *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* $\partial \boldsymbol{f}(\boldsymbol{x}_0)[\boldsymbol{u}] = \boldsymbol{r}_0$.

# Point ce contrôle

## Point de contrôle

Le point de contrôle fonctionne en ne stockant sélectivement qu'un sous-ensemble de nœuds intermédiaires, appelés points de contrôle, et en recalculant les autres.

- Comment choisir les points de contrôle ?
  - heuristique du programmeur,
  - Division récursive en deux,
  - Programmation dynamique,

**Algorithm 14.69. 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* $\boldsymbol{r}_K = \boldsymbol{u}$
- $\forall \boldsymbol{k} = K$ *vers* $1$
  - *On calcul* $\boldsymbol{x}_{k-1} = \boldsymbol{f}_{k-1} \circ \ldots \circ \boldsymbol{f}_1(\boldsymbol{x}_0)$
  - *On calcul* $\boldsymbol{r}_{k-1} = \partial \boldsymbol{f}_k^*(\boldsymbol{x}_{k-1})[\boldsymbol{r}_k]$
- *On renvoit* $\partial \boldsymbol{f}(\boldsymbol{x}_0)[\boldsymbol{u}] = \boldsymbol{r}_0$.

# Différentier les fonctions implicites

- Jusqu'à présent, nous avons appris à différentier à travers des fonctions explicites.
- Il est possible de se retrouver avec des fonctions implicites à différentier.
- Exemple: on a une EDP

$$-\Delta u + g(u, \beta) = f$$

- On chercher un $\beta$ optimal tel que $j(u) = \mid u - u_{ref} \mid^2$ soit minimal.

**Objectif**

Différentier à travers des fonctions implicites et le combiner aux modes avant et rétrograde.

- Deux solutions:
  - Dérouler (unrolling) l'algorithme comme composition de fonctions simples et différentier avec le mode avant/rétrograde. Très couteux en mémoire
  - Déterminer un algorithme capable de calculer le VJP final.

# Problèmes d'optimisation

- **1er exemple**: différentier des problèmes d'optimisation

**Problème d'optimisation**

On considère

$$h(\theta) = g(\mathbf{x}^*(\theta), \theta)$$

avec

$$\mathbf{x}^*(\theta) = \text{argmax}_{\mathbf{x} \in \mathcal{E}} f(\mathbf{x}, \theta)$$

- On cherche a différentier $h$. On obtient:

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

- Le point problématique est de calculer $\partial_\theta \mathbf{x}^\star(\theta)$.
- Comment faire ?

# Problèmes d'optimisation II

- Cas simplifié: $g(\mathbf{x}, \theta) = f(\mathbf{x}, \theta)$.
- Dans ce cas le problème devient:

$$h(\theta) = f(\mathbf{x}^*(\theta), \theta), \quad \mathbf{x}^*(\theta) = \text{argmax}_{\mathbf{x} \in \mathcal{E}} f(\mathbf{x}, \theta)$$

- C'est équivalent à résoudre:

$$\max_{\mathbf{x} \in \mathcal{E}} f(\mathbf{x}, \theta)$$

**Théorème de Danskin**

Soit $\mathbf{f} : \mathcal{E} \times \Theta \to \mathbb{R}$ avec $\times$ un ensemble convexe. Posons

$$h(\theta) := \max_{\theta \in \times} f(\mathbf{x}, \theta)$$

et

$$\mathbf{x}^\star(\theta) := \underset{\mathbf{x} \in \mathcal{E}}{\text{argmax}} f(\mathbf{x}, \theta)$$

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

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

# Problèmes d'optimisation III

- Le Théorème de Danskin dit finalement qu'on peut dériver $f$ comme si $\mathbf{x}^\star$ ne dépendait pas de $\theta$.
- Il existe d'autres théorème qui énonce le résultat avec des conditions différentes.
- Maintenant que nous avons traiter un premier cas, repassons a un cas plus général.

**Cas général: optimisation avec contraintes**

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

$$\mathbf{x}^\star(\theta) = \underset{\mathbf{x}}{\operatorname{argmin}} f(\mathbf{x}, \theta)$$
$$\text{sous les conditions que } g(\mathbf{x}, \theta) \leqslant 0$$
$$h(\mathbf{x}, \theta) = 0$$

# Problèmes d'optimisation IV

- Les conditions KTT nous permette de nous ramener a un problème de recherche de zéro sous contraintes.
- Soit

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

- La solution du problème d'optimisation est donnée par

$$G(\mathbf{x}^\star, \boldsymbol{\lambda}^\star, \boldsymbol{\nu}^\star, \theta) = 0, \quad g(\mathbf{x}^\star, \theta) \leqslant 0, \quad \boldsymbol{\lambda}^\star \geqslant 0$$

- Sous certains arguments, on peut se débarrasser des contraintes. On doit résoudre une équation nonlinéaire.
- **Conclusion**: Pour dériver un problème d'optimisation on doit dériver une fonction implicite

# Fonctions implicites I

**Problème considéré**

Soit une équation nonlinéaire $\mathbf{F}(\mathbf{x}; \theta) = \mathbf{0}$. Comment calculer:

$$\partial_\theta \mathbf{x}^*(\theta)$$

avec $\mathbf{x}^*(\theta)$ la solution de l'équation.

- On dérive $\mathbf{F}(\mathbf{x}^*(\theta); \theta) = 0$.
- On obtient:

$$\partial_1 \mathbf{F}(\mathbf{x}^\star(\theta), \theta) \partial \mathbf{x}^\star(\theta) + \partial_2 \mathbf{F}(\mathbf{x}^\star(\theta), \theta) = \mathbf{0}$$

- Donc la dérivée que l'on cherche doit satisfaire:

$$-\partial_1 \mathbf{F}(\mathbf{x}^\star(\theta), \theta) \partial \mathbf{x}^\star(\theta) = \partial_2 \mathbf{F}(\mathbf{x}^\star(\theta), \theta)$$

- Un théorème nous donne les conditions d'existence: le théorème des fonctions implicites.

# Fonctions implicites II

**Théorème des fonctions implicites**

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

- $\mathbf{x}^\star(\theta_0) = \mathbf{x}_0$,
- $\mathbf{F}(\mathbf{x}^\star(\theta), \theta) = \mathbf{0}, \forall \theta \in U_0$
- on est:

$$\partial \mathbf{x}^\star(\theta) = -\partial_1 \mathbf{F}(\mathbf{x}^\star(\theta), \theta)^{-1} \partial_2 \mathbf{F}(\mathbf{x}^\star(\theta), \theta).$$

- Le théorème des fonctions implicites nous donne la solution pour calculer les JVP et VJP d'une fonction implicite.

# Fonctions implicites III

**JVP et VJP pour les fonctions implicites**

Soit $\mathbf{x}^*(\theta)$ la solution d'une équation implicite de type $F(\mathbf{x}, \theta) = \mathbf{0}$ avec $\mathbf{F} : \mathcal{E} \times \Theta \to \mathcal{E}$. On pose

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

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

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

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

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

avec $\mathbf{r}$ solution de:

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

Programmation différentiable et EDO

## Programmation différentiable et EDO

**DpO. vs OpD**

Pour les problèmes de type fonction implicites l'approche DpO ou de déroulement parait moins avantageuse que l'approche OpD (qui calcul directement le VJP). Qu'en est t'il pour les EDO ?

**DpO. vs OpD**

L'approche OdeNet est équivalent à OpD. Comment ce comporte l'approche DpO ?

- Exemple: on veut résoudre:

$$\min_{\theta \in \Theta} L(\mathbf{x}_K) \text{ avec } \mathbf{x}_{k+1} = \Phi_{\Delta t}(\mathbf{x}_{k+1}; \theta)$$

  ou $\Phi_{\Delta t}$ est un flot discret. Il s'agit de la discrétisation du problème de contrôle optima d'EDO.
- Le gradient est calculable avec le mode rétrograde de la différentiation automatique. Même le schéma est implicite ou complexe on a tous les outils pour le différentier.

# OpD vs DpO

- DpO:
  - Cours important en mémoire ($O(K)$ avec $K$ le nombre d'étape en temps). Possible de le réduire avec des "points de contrôle".
  - Si l'ensemble du code est écrit de façon différentiable, On peut traiter tout les cas (type de fonctions de coût, présence d'un réseau, optimisation d'un solveur) directement.
- OpD:
  - Moins couteux en mémoire. En effet la résolution du dual augmenté (cours précédent) recalcule l'état primal et évite donc le stockage de l'ensemble des pas de temps.
  - On peut changer le schéma du backward (plus simple par exemple) mais ajoute de l'erreur numérique.
- Une étude conclue a une efficacité plus grande de l'approche DpO avec point de contrôle car elle souffre moins d'instabilité de gradient.

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

- En effet dans l'approche OpD rien ne garantie la stabilité du schéma utilisé pour l'EDO rétrograde.

# Schémas réversibles

- On rappelle que le flot d'une EDO satisfait: $\boldsymbol{\varphi}^t(t+\Delta t,.)^{-1} = \boldsymbol{\varphi}^{t-\Delta t}(t,.)$
- Si un schéma satisfait $\boldsymbol{\Phi}_{\Delta t}^{-1} = \boldsymbol{\Phi}_{-\Delta t}$ on parle de schéma réversible.

- Les EDO sont naturellement réversible c'st pour ca que OpD n'as pas besoin de stocker les valeurs de l'état mais peut les recalculer en remontant le flot.
- Les schémas ne sont pas forcement réversibles. Exemples: les schémas d'Euler.

**Schémas réversibles et VJP**

Si vous avez un schéma réversible vous pouvez calculer $\mathbf{x}_k$ à partir de $\mathbf{x}_{k+1}$ et donc éviter le stockage inhérent au mode rétrograde.

- Exemple: le schéma $\Phi_{\Delta t} = \Phi^E_{\frac{1}{2}\Delta t} \circ \Phi^I_{\frac{1}{2}\Delta t}$ avec les flots des schémas d'Euler implicite et explicite est réversible.