# AMAL - TP 1
# Définition de fonctions en `pyTorch`

Nicolas Baskiotis - Stéphane Rivaud - Benjamin Piwowarski - Laure Soulier

2024-2025

**Important**

Sur votre *propre ordinateur*, vous pouvez reproduire l'environnement de travail en tapant la commande suivante (attention, mettre à jour avant chaque TP)

```
pip install master_dac
```

Sur les machines des salles de TPs, cet environment est déjà présent.

```
source /users/nfs/Enseignants/piwowarski/venv/deepdac/bin/activate
```

Si vous utilisez un éditeur (VS Code), n'oubliez pas spécifier l'environment pour pouvoir développer plus facilement.

## Notations

| | |
|---|---|
| $x$ | un scalaire |
| $\mathbf{x}$ | un vecteur dans $\mathbb{R}^n$ |
| $\mathbf{X}$ | une matrice dans $\mathbb{R}^{n\times p}$ |
| $\frac{\partial f}{\partial x}$ | La dérivée partielle de $f$ par rapport à $x$. |
| $\frac{\partial f}{\partial \mathbf{x}}$ ou $\nabla_{\mathbf{x}} f$ | Vecteur de dérivée partielles de $f$ par rapport à $\mathbf{x}_i$. |
| $\frac{\partial f}{\partial \mathbf{X}}$ | Vecteur de dérivée partielles de $f$ par rapport à $\mathbf{X}_{ij}$. |

## 1 Graphes de fonction

Un élément central de `PyTorch` est le graphe de calcul : lors du calcul d'une variable, l'ensemble des opérations qui ont servi au calcul sont stockées sous la forme d'un graphe de

calcul. Ce graphe est *acyclique*. Les nœuds internes du graphe représentent les opérations, le noeud terminal le résultat et les racines les variables d'entrées. Ce graphe sert en particulier à calculer les dérivées partielles de la sortie par rapport aux variables d'entrées – en utilisant les règles de dérivations chainées des fonctions composées.

**Graphe de fonctions** Dans ces TPs, nous allons considérer que tout réseau de neurones peut s'exprimer sous la forme d'une composition de fonctions de base (transformation – e.g. linéaire ; activation – e.g. sigmoïde, softmax ; erreur – e.g. erreur quadratique).

Un exemple est donné figure 1 – pour l'instant, nous ne nous intéressons pas à la nature de ce graphe, mais plus à sa formalisation : nous avons une entrée ($\mathbf{x}$), des paramètres ($\theta$) et trois fonctions (linéaire, transposée, erreur quadratique). La sortie est un scalaire $\ell \in \mathbb{R}$. Le graphe de calcul réalise les opérations suivantes : $\mathbf{y} = \theta\mathbf{x} + b_1$, $\hat{\mathbf{x}} = \theta^\top\mathbf{y} + b_2$, $\ell = \Delta(\mathbf{x}, \hat{\mathbf{x}})$ qui calcule un coût.

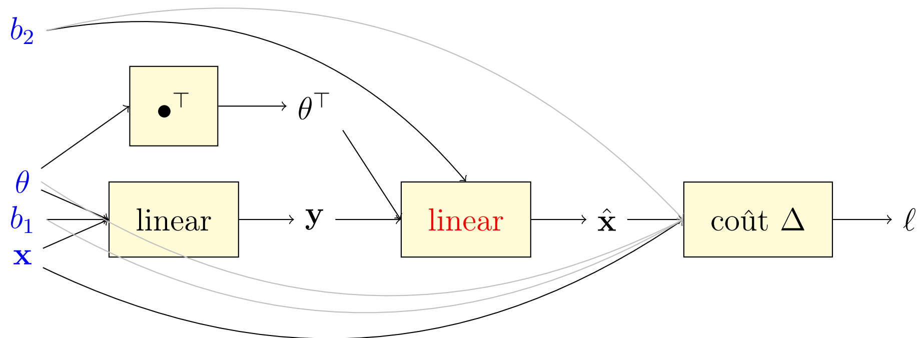



FIGURE 1 – Un graphe de calcul (auto-encodeur)

La fonction objectif qui donne la sortie $\ell$ est $L(\mathbf{x}, b_1, \theta, b_2)$ – elle dépend donc de quatre tenseurs (ici des scalaires, vecteurs et matrices). Pour apprendre, il faut savoir calculer le gradient du risque $\nabla_L$ par rapport aux paramètres (ici $\theta$, $b_1$ et $b_2$).

**Fonction** Afin d'abstraire ce graphe de calcul, nous nous focalisons sur la fonction linéaire en rouge dans la figure 1 : nous nous plaçons dans le cadre général où nous considérons toutes les opérations qui ont lieu en aval comme étant produit par une fonction $f$ et toutes celles en amont par une fonction $L$ comme illustré par la figure 2.

On a donc les correspondances de notations suivantes pour la figure 2 :

$\mathbf{m}$ correspond aux entrées initiales $(b_2, \theta, b_1, \mathbf{x})$

$\mathbf{n}$ correspond aux entrées de linear $(b_2, \theta^\top, \mathbf{y})$

$\mathbf{z}$ correspond à $\hat{\mathbf{x}}$

$f$ correspond à $(b_2, \theta, b_1, \mathbf{x}) \mapsto (b_2, \theta^\top, \mathbf{y})$, soit $\mathbf{n} = f(\mathbf{m})$

i.e. toutes les transformations qui précèdent la fonction linear

$g$ correspond à linear, soit $\mathbf{z} = g(\mathbf{n})$

$L$ correspond à coût, soit $\ell = L(\mathbf{m}, \mathbf{z})$

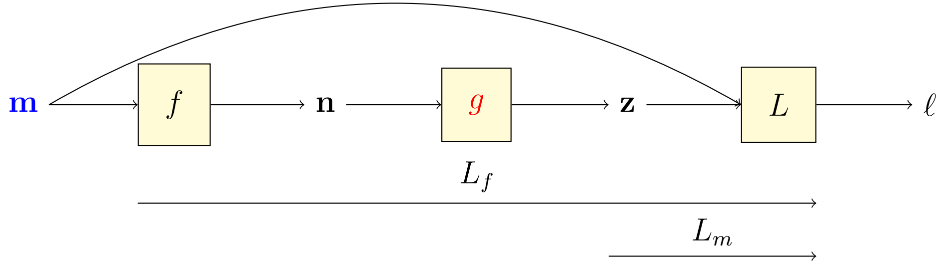



FIGURE 2 – Une fonction $g$ dans le graphe de calcul. $f$ et $L$ représentent l'ensemble des fonctions qui viennent avant/après dans le graphe de calcul.

Afin de comprendre dans la pratique le mécanisme de rétro-propagation, nous nous concentrons sur la fonction $g$ (figure 2). Les entrées $\mathbf{m}$ correspondent aux observations et à des paramètres, et $\mathbf{n} = f(\mathbf{m}), \mathbf{z} = g(\mathbf{n}) = g \circ f(\mathbf{m})$. On note par ailleurs $L_{\mathbf{m}}(\mathbf{z}) = L(\mathbf{m}, \mathbf{z})$.

On rappelle que quand pour une fonction $L(\mathbf{u}, \mathbf{v})$ et des fonction $g(\mathbf{t})$, $h(\mathbf{t})$, les dérivées partielles composées de $L(g(\mathbf{t}), h(\mathbf{t}))$ sont [1] :

$$\frac{\partial L}{\partial \mathbf{t}_k}(g(\mathbf{t}), h(\mathbf{t})) = \sum_i \frac{\partial L}{\partial \mathbf{u}_i}(g(\mathbf{t})) \times \frac{\partial g_i}{\partial \mathbf{t}_k}(\mathbf{t}) + \sum_j \frac{\partial L}{\partial \mathbf{v}_j}(h(\mathbf{t})) \times \frac{\partial h_j}{\partial \mathbf{t}_k}(\mathbf{t})$$

On suppose que l'on connaît $\frac{\partial L}{\partial \mathbf{u}_k}$ et $\frac{\partial L}{\partial \mathbf{v}_k}$. En utilisant le théorème de dérivation des fonctions composées, nous avons :

$$\frac{\partial L_f}{\partial \mathbf{m}_k}(\mathbf{m}) = \frac{\partial L}{\partial \mathbf{m}_k}(\mathbf{m}, \mathbf{z}) + \sum_i \frac{\partial L_{\mathbf{m}} \circ g}{\partial \mathbf{n}_i}(\mathbf{n}) \times \frac{\partial f_i}{\partial \mathbf{m}_k}(\mathbf{m})$$

$$\frac{\partial L_{\mathbf{m}} \circ g}{\partial \mathbf{n}_k}(\mathbf{n}) = \sum_i \frac{\partial L_m}{\partial \mathbf{z}_i}(\mathbf{z}) \times \frac{\partial g_i}{\partial \mathbf{n}_k}(\mathbf{n}) = \sum_i \frac{\partial L}{\partial \mathbf{z}_i}(\mathbf{m}, \mathbf{z}) \times \frac{\partial g_i}{\partial \mathbf{n}_k}(\mathbf{n})$$

1. $\frac{\partial L(\mathbf{u}, \mathbf{v})}{\partial \mathbf{u}_k}$ n'est qu'une notation pour indiquer qu'on prend la dérivée partielle par rapport à la $k$-ième dimension de la première variable de $L$

En regardant les équations ci-dessus, nous observons que pour calculer les dérivées partielles par rapport à l'ensemble des entrées (en vert), il suffit de remonter l'information depuis les fonctions filles vers leurs parentes et savoir calculer les dérivées partielles des fonctions par rapport à leurs entrées (les dérivées signalées en rouge que l'on peut calculer analytiquement).

En particulier, pour la fonction $g$, il suffit de connaître $\frac{\partial L_m}{\partial \mathbf{z}_i}$ (transmis par la fonction $L$) et la forme analytique des dérivées partielles $\frac{\partial g_i}{\partial \mathbf{n}_k}$ de $g$ par rapport à une de ses entrées. **Autograd** (dans `PyTorch`) permet de faire cela de manière automatique en enregistrant les fonctions parentes pour chaque calcul effectué.

# 2 Différenciation automatique : autograd

Toute opération sous `PyTorch` hérite de la classe `torch.nn.Function` et doit définir :

- une méthode `forward(context, *args)` : passe avant, calcule le résultat de la fonction appliquée aux arguments
- une méthode `backward(context, *args)` : passe arrière, calcule les dérivées partielles par rapport aux entrées. Les arguments de cette méthode correspondent aux valeurs des dérivées suivantes dans le graphe de calcul. En particulier, il y a autant d'arguments à backward que de sorties pour la méthode forward et autant de sorties que d'arguments dans la méthode forward. Le calcul se fait sur les valeurs du dernier appel de forward.

En pratique, ce ne sont pas ces fonctions qui sont directement utilisées, mais un encapsulage de ces fonctions dans les tenseurs. A chaque fois qu'une opération est exécutée sur un tenseur, le graphe de calcul est calculé à la volée et stocké dans le tenseur résultant. La méthode `backward` d'un tenseur permet de rétropropager le calcul du gradient sur toutes les variables qui ont servies à son calcul ; la valeur du gradient pour chaque dérivée partielle se trouve dans l'attribut `grad` de la variable concernée.

Toutefois, comme le graphe de calcul est coûteux en ressource, il faut spécifier manuellement que l'on souhaite le calculer. Ceci est fait par l'intermédiaire de l'attribut booléen `requires_grad` des tenseurs (il suffit de le spécifier pour les variables racines pour que l'information soit propagée). De plus, les gradients intermédiaires ne sont pas conservés par défaut pour des raisons d'optimisation mémoire. Si on souhaite les conserver, il faut appeler la méthode `retain_grad()` sur la variable concernée.

Attention ! vu les propriétés additives du gradient, l'appel à `backward()` met à jour par addition le gradient attaché à la variable. Il faut explicitement demander sa remise à zéro lors d'un nouveau calcul, sinon le résultat est additioné à l'ancien.

Par ailleurs, il est possible de spécifier que pour un bloc d'exécution donné le gradient n'aura pas à être calculé et ainsi désactiver le graphe de calcul grâce à l'instruction `with torch.no_grad()`.

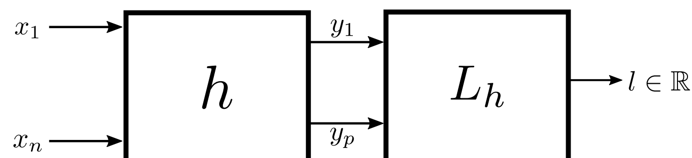
```
a = torch.rand((1,10),requires_grad=True)
b = torch.rand((1,10),requires_grad=True)
c = a.mm(b.t())
```

```
d = 2 * c
c.retain_grad() # on veut conserver le gradient par rapport à c
d.backward()  ## calcul du gradient et retropropagation
              ##jusqu'aux feuilles du graphe de calcul
print(d.grad) #Rien : le gradient par rapport à d n'est pas conservé
print(c.grad) # Celui-ci est conservé
print(a.grad) ## gradient de d par rapport à a qui est une feuille
print(b.grad) ## gradient de d par rapport à b qui est une feuille

d = 2 * a.mm(b.t())
d.backward()
print(a.grad) ## 2 fois celui d'avant, le gradient est additioné
a.grad.data.zero_() ## reinitialisation du gradient pour a
d = 2 * a.mm(b.t())
d.backward()
print(a.grad) ## Cette fois, c'est ok

with torch.no_grad():
    c = a.mm(b.t())  ## Le calcul est effectué sans garder le graphe de calcul
c.backward() ## Erreur
```

**Question 1**

1. Implémenter un algorithme de descente du gradient batch pour la régression linéaire en utilisant les fonctionnalités de la différenciation automatique.
2. Tester votre implémentation avec les données de California Housing. Tracer la courbe du coût en apprentissage et celle en test. Utiliser pour cela de préférence `tensorboard`. Utiliser pour l'instant seulement la fonction `add_scalar` après avoir créé un fichier de log grâce à la commande `SummaryWriter(path)`.
3. Implémenter la descente de gradient en partant du fichier `tp1_descente.py`.
4. Implémenter une descente de gradient stochastique et une mini-batch. Comparer la vitesse de convergence et les résultats obtenus.

## 3 Fonction dans un graphe



On suppose qu'on a une fonction $h : \mathbb{R}^n \mapsto \mathbb{R}^p$ et que sa sortie sert à calculer une fonction objectif $L$. Les entrés $\mathbf{x}_1, \ldots, \mathbf{x}_n$ peuvent provenir d'autre fonctions – mais ceci n'a pas d'importance. La sortie de la fonction $h$ est un vecteur $h(\mathbf{x}) = \mathbf{y} \in \mathbb{R}^p$.

**Question 2**

Caluler la dérivée partielle

$$\frac{\partial L \circ h}{\partial \mathbf{x}_i}(\mathbf{x})$$

en fonction de $\frac{\partial h_j}{\partial \mathbf{x}_i}(\mathbf{x})$ et de $(\nabla L)_j = \frac{\partial L}{\partial \mathbf{y}_j}(h(\mathbf{x}))$ où $(\nabla L)_j$ permet de simplifier la notation. Pour une fonction $h$ connue (ce qui est toujours le cas pour nous), on peut calculer explicitement sa dérivée par rapport à $\mathbf{x}_i$.

Le calcul précédent permet d'intuiter un résultat sur lequel nous reviendrons dans les TPs suivants : pour calculer le gradient d'une fonction composée de manière analytique, il suffit de connaître le gradient de chaque fonction par rapport à chacune de ses entrées et appliquer une dérivation chaînée.

Les plateformes modernes d'apprentissage profond exploitent ce résultat pour modulariser et rendre très efficace la programmation de nouvelles architectures. Pour cela, une *fonction* doit implémenter non seulement le résultat de son application - appelé passe *forward* - mais également la dérivée partielle par rapport à chacune de ses entrées - appelée passe *backward*. En particulier,

**Forward** Calcul de $y = h(\mathbf{x})$ en fonction de $\mathbf{x}$

**Backward** Calcul de $\frac{\partial h}{\partial \mathbf{x}}(\mathbf{x})$ et retour en prenant en compte les $\nabla L(\mathbf{y})$ passés en paramètres et de données dérivées de la phase forward $(\mathbf{x})$. Certains calculs partiels peuvent être également conservés pour accélérer les calculs.

# 4 Application : Régression Linéaire

## 4.1 Calcul du gradient (scalaire)

Le modèle de régression linéaire peut être vu comme la composition de deux éléments :

— une fonction linéaire, responsable du calcul de la prédiction :

$$\hat{y} = f(\mathbf{x}, \mathbf{w}, b) = \mathbf{x} \cdot \mathbf{w} + b$$

— un coût aux moindres carrés (MSE) : $mse(\hat{y}, y) = (\hat{y} - y)^2$

Afin d'effectuer une descente de gradient, il faut calculer le gradient du coût par rapport aux paramètres $\mathbf{w}$ et $b$. Nous allons calculer ce gradient grâce au chaînage des dérivées partielles. Afin de traiter le cas général, nous supposons que le coût est défini de la manière suivante :

$$C(\mathbf{x}, \mathbf{w}, b, y) = mse(f(\mathbf{x}, \mathbf{w}, b), y)$$

où $\mathbf{x}$ est l'entrée, $y$ la sortie désirée dans $\mathbb{R}$, et $\mathbf{w} \in \mathbb{R}^n$ et $b \in \mathbb{R}$ les paramètres du modèle linéaire.

Afin de calculer le gradient de $C$ par rapport à $\mathbf{w}$ et $b$, ce qui permet d'optimiser les paramètres via une descente de gradient, nous devons tout d'abord nous intéresser aux gradients des fonctions $f$ et $mse$.

**Question 3**

Appliquer le résultat trouvé dans la question 1 pour calculer

— Les dérivées pour la fonction de coût MSE

$$\frac{\partial L \circ mse}{\partial y}(\hat{y}, y) \text{ et } \frac{\partial L \circ mse}{\partial \hat{y}}(\hat{y}, y)$$

— Les dérivées de la fonction linéaire $f$ par rapport à ses entrées $\mathbf{x}$ ($\mathbf{w}$ et $b$ sont obtenus de manière similaire)

$$\frac{\partial L \circ f}{\partial \mathbf{x}_j}(\mathbf{x}, \mathbf{w}, b)$$

La fonction $L$ introduite ici est générique et est présente uniquement pour bien montrer le fonctionnement du chaînage des dérivées. Par rapport au problème qui nous intéresse, dans le premier cas comme la fonction $mse$ est la dernière de étape de calcul, on peut considérer que $L$ correspond à l'identité ; dans le deuxième cas il s'agit bien sûr de la fonction $mse(\cdot, \hat{y})$.

## 4.2 Calcul du gradient (matriciel)

On considère maintenant un cas plus général, qui sera celui utilisé en PyTorch, où nous travaillerons avec un lot (*batch*) de $q$ exemples (au lieu d'un) et $p$ sorties (au lieu d'une).

— Nous avons toujours une fonction linéaire, responsable du calcul de la prédiction :

$$\hat{\mathbf{Y}} = f(\mathbf{X}, \mathbf{W}, \mathbf{b}) = \mathbf{X}\mathbf{W} + \begin{pmatrix} \mathbf{b} \\ \vdots \\ \mathbf{b} \end{pmatrix}$$

où $\mathbf{X} \in \mathbb{R}^{q \times n}$ sont les entrées, $\mathbf{W} \in \mathbb{R}^{n \times p}$ est la matrice de poids, et $\mathbf{b} \in \mathbb{R}^{1 \times p}$ est le biais[2]. La sortie $\mathbf{Y}$ est donc une matrice dans $\mathbb{R}^{q \times p}$.

2. Avec des tenseurs PyTorch (similaires aux tenseurs numpy), cela s'écrit `X @ W + b` en utilisant la multiplication matricielle et le *broadcast*

— un coût aux moindres carrés (MSE) que l'on généralise à un lot d'exemples et plusieurs sorties : $mse(\hat{\mathbf{Y}}, \mathbf{Y}) = \frac{1}{q}\|\hat{\mathbf{Y}} - \mathbf{Y}\|^2$. Notez que l'on divise par le nombre d'exemple, ce qui permet d'avoir une magnitude du gradient qui ne varie pas en fonction du nombre d'exemples.

Le coût $C$ est alors défini comme

$$C(\mathbf{X}, \mathbf{W}, \mathbf{b}, \mathbf{Y}) = mse(f(\mathbf{X}, \mathbf{W}, \mathbf{b}), \mathbf{Y})$$

**Question 4**

Étendre les résultats obtenus dans la question précédente pour calculer

— Les dérivées pour la fonction de coût MSE

$$\frac{\partial L \circ mse}{\partial \mathbf{Y}_{ij}}(\hat{\mathbf{Y}}, \mathbf{Y})$$

et

$$\frac{\partial L \circ mse}{\partial \hat{\mathbf{Y}}_{ij}}(\hat{\mathbf{Y}}, \mathbf{Y})$$

— Les dérivées de la fonction linéaire $f$ par rapport à ses entrées

$$\frac{\partial L \circ f}{\partial \mathbf{X}_{ij}}(\mathbf{X}, \mathbf{W}, \mathbf{b}),\ \frac{\partial L \circ f}{\partial \mathbf{W}_{ij}}(\mathbf{X}, \mathbf{W}, \mathbf{b}) \text{ et } \frac{\partial L \circ f}{\partial \mathbf{b}_i}(\mathbf{X}, \mathbf{W}, \mathbf{b})$$

Afin d'optimiser les calculs, et d'exploiter au mieux les architectures parallèles (GPU), il faut ensuite essayer d'exprimer au maximum les calcul de manière matricielle.

**Question 5**

Utiliser une écriture matricielle pour exprimer les résultats précédents, i.e. calculer

— Les dérivées pour la fonction de coût MSE

$$\frac{\partial L \circ mse}{\partial \mathbf{Y}}(\hat{\mathbf{Y}}, \mathbf{Y})$$

et

$$\frac{\partial L \circ mse}{\partial \hat{\mathbf{Y}}}(\hat{\mathbf{Y}}, \mathbf{Y})$$

— Les dérivées de la fonction linéaire $f$ par rapport à ses entrées

$$\frac{\partial L \circ f}{\partial \mathbf{X}}(\mathbf{X}, \mathbf{W}, \mathbf{b}),\ \frac{\partial L \circ f}{\partial \mathbf{W}}(\mathbf{X}, \mathbf{W}, \mathbf{b}) \text{ et } \frac{\partial L \circ f}{\partial \mathbf{b}}(\mathbf{X}, \mathbf{W}, \mathbf{b})$$

### 4.3 Calcul du gradient de $C$

Maintenant que nous avons calculé toutes les dérivées partielles, il faut montrer comment calculer la dérivée de $C$ par rapport à chacune de ses entrées.

**Question 6**

Donner les formules qui permettent de calculer

$$\frac{\partial C}{\partial \mathbf{W}} \text{ et } \frac{\partial C}{\partial \mathbf{b}}$$

Vous pouvez utiliser la fonction $mse_{\mathbf{Y}}(\hat{\mathbf{Y}}) = mse(\hat{\mathbf{Y}}, \mathbf{Y})$.

## 5 Implémentation des fonctions

`PyTorch` utilise une classe abstraite `Function` dont sont héritées toutes les fonctions et qui nécessite l'implémentation de ces deux méthodes :

- méthode `forward(ctx, *inputs)` : calcule le résultat de l'application de la fonction
- méthode `backward(ctx, *grad_outputs)` : calcule le gradient partiel par rapport à chaque entrée de la méthode `forward` ; le nombre de `grad_outputs` doit être égale aux nombre de sorties de `forward` (pourquoi ?) et le nombre de sorties doit être égale aux nombres de `inputs` de `forward`.

Pour des raisons d'implémentation, les deux méthodes doivent être statiques. Le premier paramètre `ctx` permet de sauvegarder un contexte lors de la passe `forward` (par exemple les tenseurs d'entrées) et il est passé lors de la passe `backward` en paramètre afin de récupérer les valeurs. **Attention :** le contexte doit être unique pour chaque appel de `forward`.

**Question 7**

Télécharger le code fourni sur le site de l'UE. Il contient un squelette pour l'implémentation des fonctions, une classe `Context` pour faire fonctionner votre code et un exemple d'utilisation.

1. Implémenter en `PyTorch` les classes nécessaires pour la régression linéaire : la fonction linéaire et la fonction de coût MSE. Utiliser bien les outils propres à `pyTorch`, en particulier des `Tensor` et pas des matrices `numpy`. Assurez vous que vos fonctions prennent en entrée des batchs d'exemples (matrice 2D) et non un seul exemple (vecteur). N'hésiter pas à prendre un exemple et de déterminer les dimensions des différentes matrices en jeu.

2. `PyTorch` vous permet de vérifier le calcul de vos dérivées grâce à la fonction `gradcheck` qui effectue une approximation numérique par différence finie. Un exemple d'utilisation est donné dans le code source. Tester vos fonctions avec cet outil, en utilisant `tp1_gradcheck.py` que vous devez compléter.

## Descente de gradient

La fonction $f$ correspond au prédicteur (ou classifieur) que l'on souhaite apprendre, i.e. trouver les paramètres $\mathbf{W}$ et $\mathbf{b}$ qui minimisent le risque d'erreur sur les prédictions. Pour chaque ensemble d'exemples $\mathbf{X}$ donné, la sortie du prédicteur $\hat{\mathbf{Y}} =$ est comparée à la sortie attendue $\mathbf{Y}$ grâce à la fonction de coût $L(\hat{\mathbf{y}}, \mathbf{y})$ qui permet de quantifier l'erreur. Pour les paramètres $\mathbf{W}$ et $\mathbf{b}$, le coût associé à l'ensemble d'apprentissage est

$$C(\mathbf{W}, \mathbf{b}) = mse(f(\mathbf{X}, \mathbf{W}, \mathbf{b}), \mathbf{Y})$$

.

Dans le contexte de la minimisation du risque empirique, la formalisation du problème d'apprentissage est la suivante : trouver le paramètre optimal $\mathbf{w}^*$ qui minimise le coût de prédiction sur l'ensemble des données d'apprentissage

$$\mathbf{W}^*, \mathbf{b}^* = argmin_{\mathbf{w},\mathbf{b}} L(\mathbf{W}, \mathbf{b}) = argmin_{\mathbf{W},\mathbf{b}}\ mse(f(\mathbf{X}, \mathbf{W}, \mathbf{b}), \mathbf{Y})$$

.

Un algorithme d'apprentissage classique pour optimiser le paramètre est l'algorithme de descente de gradient. Il consiste à mettre à jour itérativement le paramètre $\mathbf{w}$ selon la formule :

$$\mathbf{w} \leftarrow \mathbf{w} - \epsilon \nabla_{\mathbf{W}} L(\mathbf{W}, \mathbf{b})$$

$$\mathbf{b} \leftarrow \mathbf{b} - \epsilon \nabla_{\mathbf{b}} L(\mathbf{W}, \mathbf{b})$$

avec $\epsilon$ un paramètre appelé le pas de gradient.