Automatic differentiation

Maximilien

June 2020

Notions MIASHS fonctions, $\mathbb{R}^n$, calcul de dérivés, gradient

1 Introduction

Ces dernières années ont vu exploser les usages du *deep learning* ou apprentissage profond ainsi que des nombreux *frameworks* dédiés. On citera notamment *Pytorch* et *TensorFlow*. Bien que ces derniers soient généralement vus comme des *frameworks de deep learning*, ils sont en réalité des outils de différentiation automatique.

Une tâche classique en apprentissage automatique consiste à chercher une fonction $h : \mathcal{X} \mapsto \mathcal{Y}$ où $\mathcal{X}$ pourrait être par exemple vu comme l'ensemble des photos de chiens et de chats et $\mathcal{Y} = \{0, 1\}$ comme les étiquettes chien ou chat. Les réseaux de neurones, qui représentent les fonctions types utilisées en *deep learning*, n'échappent pas à cette règle. La fonction h est en particulier recherchée dans un ensemble $\mathcal{H}$. Cela se fait en fixant une forme paramétrique à la fonction qu'on considère h_θ où θ fait référence aux paramètres. Ainsi, notre ensemble $\mathcal{H}$ regroupe toute les fonctions ayant la même forme paramétrique h_θ, $\forall \theta \in \mathbb{R}^m$. Une manière traditionnelle de rechercher la bonne paramétrisation s'appelle la "minimisation du risque empirique". On commence ici par collecter un jeu de données sur $\mathcal{X} \times \mathcal{Y}$ généralement noté $\mathcal{D} = \{(x_i, y_i)\}_{i \leq N}$ et à définir une notion d'erreur, ou *loss*. Cette loss regardera les "prédictions" de notre fonction pour chaque élément de notre jeu de données $\{h_\theta(x_i)\}$ et comparera ces dernières aux vraies valeurs $\{y_i\}$. Cette loss est vue comme une fonction de $\mathbb{R}^m$ dans $\mathbb{R}$ et non de $\mathcal{X}$. En effet, la quantité avec laquelle nous allons jouer est θ. Notons $\mathcal{L}_\mathcal{D}(\theta)$, ou simplement $\mathcal{L}(\theta)$, l'erreur faite par notre modèle pour la paramétrisation θ sur le jeu de données que nous avons collecté. Trouver la valeur optimale $\theta^\star$ peut se faire algorithmiquement en suivant ce qu'on appelle la descente de gradient à partir du moment où $\mathcal{L}(\theta)$ satisfait certaines régularités (différentiable, convexe et coercive). Cependant, cette idée d'optimisation en suivant le flot de gradient peut également être appliquée sur des scénarios satisfaisant moins d'hypothèses (non convexe, non coercif, différentiable presque partout voire sous-différentiable). C'est le cas particulier des réseaux de neurones. Ne pas trouver la solution optimale $\theta^\star$ n'est pas gênant à partir du moment où la solution obtenue $\hat{\theta}$ permet d'avoir suffisamment peu d'erreurs. Ces

notes n'aborderont pas les aspects de généralisation et donc de régularisation liés à l'optimisation des réseaux de neurones mais la manière dont le gradient est calculé de manière générique pour des fonctions quelconques (sous certaines conditions).

2 Descente de gradient

Rappelons tout d'abord qu'un champ de vecteurs associe à un vecteur de "coordonnées" une direction dans l'espace : $v : \mathbb{R}^d \mapsto \mathbb{R}^d$. Reprenons notre fonction $\mathcal{L}(\theta)$. Le flot de gradient associe à chaque coordonnée θ le vecteur de plus forte pente $\nabla\mathcal{L}(\theta)$ tel que $d\mathcal{L}(\theta) = \langle \nabla\mathcal{L}(\theta), x\rangle$. Il s'agit donc d'un champ de vecteurs :

$$\begin{aligned} G : \mathbb{R}^m &\mapsto \mathbb{R}^m \\ \theta &\to \nabla\mathcal{L}(\theta) \end{aligned}$$

Soit un point de départ $\theta(0)$ (on voit $\theta(t)$ comme une fonction du temps résultant du processus d'optimisation) dans l'espace des paramètres $\mathbb{R}^m$. Idéalement, on souhaiterait suivre le flot de gradient G dans la direction inverse jusqu'à ce qu'on atteigne un minimum local :

$$\frac{d\theta}{dt} = -\nabla\mathcal{L}(\cdot)$$

Étant donné un point de départ (une condition initiale) $\theta(0)$ et si $\mathcal{L}$ est localement Lipschitz en θ, alors le théorème de Cauchy-Lipschitz nous dit que la solution du problème de Cauchy ci-dessus est unique et on la note $\theta(t)$. De plus, si $\mathcal{L}$ est fortement convexe et coercive alors $\theta(t)$ convergera vers un minimum global $\theta^\star$ qui ne dépend pas de la condition initiale. Si la fonction est seulement convexe et coercive, alors la solution optimale n'est pas nécessairement unique (et dépend le cas échéant de la condition initiale) bien qu'une solution locale soit nécessairement une solution optimale globale. Si la fonction n'est pas coercive, alors il n'existe peut-être tout simplement pas de solution optimale et si la fonction n'est pas convexe, alors toute solution locale n'est pas forcément une solution globale. Qu'une solution optimale $\theta^\star$ n'existe pas voire ne soit pas globale n'est pas gênant en soit si à partir du moment où l'algorithme d'optimisation est interrompu, la quantité d'erreurs $\mathcal{L}$ est suffisamment faible – ce critère dépend de l'application utilisateur. L'absence de solution optimale limite cependant l'interprétation des résultats. À noter que dans ce cas de figure, une condition d'arrêt de notre algorithme de descente de gradient doit être calculée sur les valeurs de $\mathcal{L}$ et non sur deux itération susccessives de θ.

Dans le cas particulier des réseaux de neurones, $\mathcal{L}$ n'est pas convexe et il est toujours possible de trouver une condition initiale telle que l'optimisation diverge à l'infini.

En sautant certaines notions telles que les opérateurs proximaux, etc. allons directement à la formulation classique de l'algorithme de descente de gradient. Ce dernier est vu comme une discrétisation du problème continu présenté

précédemment. Il existe beaucoup de variantes de l'algorithme de descente de gradient. La version la plus standard est présentée par l'algorithme 1.

Algorithm 1: La descente de gradient

input : $f : \mathbb{R}^m \mapsto \mathbb{R}$, $x^{(0)} \in \mathbb{R}^m$, $\eta > 0$.
output: $\hat{x}$ un minimum local ou garantissant certaines propriétés.
$t \leftarrow 0$;
while *not stop_condition* **do**
  $x^{(t+1)} \leftarrow x^{(t)} - \eta \nabla f(x^{(t)})$;
  $t \leftarrow t + 1$;
end
output$\leftarrow x^{(t)}$;

À partir du moment où l'algorithme devient discret, la notion de stabilité apparaît. On peut ainsi contrôler la vitesse de déplacement avec le paramètre η. De trop grande valeur du paramètre η risque de rendre le problème instable et des valeurs trop faibles favoriseraient les situations où l'optimisation reste coincée dans de mauvais minimum locaux dans les cas non convexes. Ainsi, on se déplace dans le sens inverse du gradient avec une vitesse qui dépend bien entendu de la norme du gradient, mais également de ce paramètre de "pas" η.

Lorsqu'une personne "s'attaque" au *deep learning*, l'algorithme qui lui est présenté est souvent appelé SGD ou *stochastic gradient descent* au lieu de *gradient descent*. Cela vient du fait que la fonction f représente une quantité d'erreur sur notre jeu de donnée et qu'on l'estime en ne prenant en compte que quelques éléments du jeu de données qu'on échantillonne à chaque itération. Cette estimation par échantillon introduit une dose de stochasticité à chaque pas. Ce bruit est fondamental dans le succès des réseaux de neurones profonds dans son rôle de régularisation. Ces notes n'aborderont pas ce point.

Concernant la condition d'arrêt, dans un problème d'optimisation convexe, il est possible de comparer la valeur de la fonction entre deux étapes : $f(x^{(t)}) - f(x^{(t+1)})$. Cependant, dans le cadre des réseaux de neurones, ce n'est plus vrai et l'évolution de la fonction f au cours du temps peut ralentir puis accélérer à nouveau. En général, on fixe à la main le nombre d'itérations en fonction des performances de notre modèle sur un ensemble de validation (i.e. qui ne servira jamais à calculer le gradient).

On observe que le cœur de ce processus d'optimisation est le calcul de ce gradient qui donne la direction et la vitesse d'optimisation. La section suivante introduit la notion de différentiation automatique permettant son calcul.

3 Différentiation automatique

3.1 Les concepts

Soit $f : \mathbb{R}^m \mapsto \mathbb{R}$ et $x \in \mathbb{R}^m$. Un pas d'optimisation consiste à "déplacer" x de manière à minimiser au maximum notre fonction f. Pour cela on suit l'inverse du gradient qui nous donne la direction de plus forte pente. Le calcul de ce

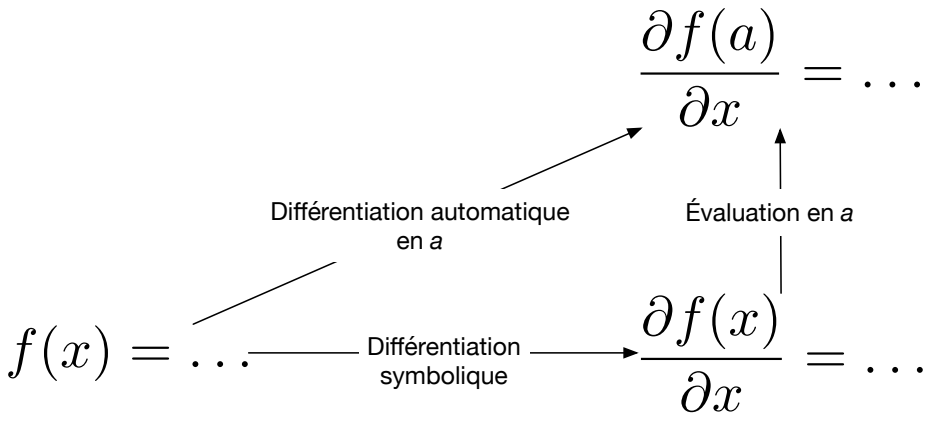

FIGURE 1 – Différentiation automatique et symbolique

gradient est la clé. Bien que la plupart des librairies classiques de statistiques possèdent des implémentations propres à chaque modèle, celles liées au *deep learnign* s'accompagnent d'une infinité d'architectures de réseaux différentes. Il convient de proposer à l'utilisateur une stratégie qui ne l'oblige pas à calculer lui même son gradient : la différentiation automatique.

Rappelons que le gradient est le vecteur des dérivées partielles :

$$\nabla f(a) = \left[\frac{\partial f(a)}{\partial x_1}, \ldots, \frac{\partial f(a)}{\partial x_m}\right]^T.$$

Il est facile de démontrer que $\nabla f(a)$ est la direction de plus forte pente au point a et est orthogonale ($\langle \nabla f(a), \cdot \rangle = 0$) aux lignes de niveaux passant par a.

La différentiation automatique se différencie du calcul symbolique en ce sens que la forme analytique de la solution n'intéresse pas, mais uniquement son évaluation en un point. La figure 1 illustre cette idée. La différentiation automatique est ainsi beaucoup plus efficace à calculer. De plus, la différentiation automatique n'est pas une stratégie de différentiation numérique et ne souffrent pas ainsi d'approximation (autre que celles liées à la représentation des nombres sur un ordinateur).

Soit $f : \mathbb{R}^m \mapsto \mathbb{R}^n$ et $g : \mathbb{R}^n \mapsto \mathbb{R}$ et leur composition $(g \circ f)$. La base de la différentiation automatique consiste à observer qu'à partir du moment où on connaît les dérivées partielles de f, de g et d'une composition de fonctions, alors le reste n'est qu'une répétition de tâches de dérivations élémentaires :

$$\frac{\partial (g \circ f)(a)}{\partial x_j} = \sum_{i=1}^{n} \frac{\partial g}{\partial f_i}(f(x)) \frac{\partial f_i}{\partial x_j}(x) \tag{1}$$

où $\frac{\partial g}{\partial f_i}$ est la dérivée partiel de g par rapport à sa i^{eme} entrée et $\frac{\partial f_i}{\partial x_j}$ la dérivée partielle de f par rapport à sa j^{eme} entrée. Toutes les autres opérations sont

élémentaires. Si ces dérivées partielles sont elle-mêmes des constructions, il suffit de répéter l'opération.

3.2 L'algorithme de rétro-propagation du gradient

L'algorithme de rétro-propagation du gradient ou *backpropagation* est un outil de différentiation automatique. L'algorithme sera détaillé au travers de sommes, mais le problème pourrait être formulé de manière matricielle (i.e. Jacobienne).

La structure de données de base de l'algorithme de rétro-propagation est un graphe dirigé $G = (E, V)$ sans cycle qui décrit une fonctionnelle $f : \mathbb{R}^m \mapsto \mathbb{R}$. Notons juste qu'en *machine learning*, le gradient est calculé dans l'espace des paramètres et non dans l'espace des données elles-même. Finalement, lors de l'apprentissage on voit nos modèles comme des fonctions des paramètres dans $\mathbb{R}$ et lors de l'utilisation de notre modèle calculé, on le voit comme une fonction des données dans $\mathcal{Y}$. Chaque nœud (ou *vertex*) de notre graphe représente une opération élémentaire (e.g. multiplication, addition, sinus). On dira qu'une opération est élémentaire si ses dérivées partielles sont connues. Les arêtes entrantes sont les arguments de la fonction et l'unique arête sortante est le résutlat de notre fonction. Une fonction dans $\mathbb{R}^n$ est vue comme n nœuds représentant chacun une des dimensions de sortie.

L'algorithme de rétro-propagation du gradient se structure en deux étapes : le *forward* et le *backward*. La première étape calcule un certain nombre de dérivées partielles et l'évaluation de la fonction. La seconde remonte le graphe pour agréger ces calculs et obtenir les dérivées partielles.

3.2.1 Forward

Soit une fonction $f : \mathbb{R}^m \mapsto \mathbb{R}$ que l'on peut représenter sous la forme d'un graphe dirigé sans cycle. Soit un nœud de ce graphe représentant une fonction $g : \mathbb{R}^n \mapsto \mathbb{R}$. La figure 2 illustre ce nœud de calcul. Cette fonction est un nœud de calcul élémentaire dont on sait calculer les dérivées partielles.

La première phase de différentiation automatique consiste à remonter le graphe en partant des arguments de notre fonction f ainsi que de ses paramètres jusqu'à sa sortie et en suivant les arêtes dirigées. Les nœuds sont traités à partir du moment où l'ensemble de leurs nœuds parents le sont. Chaque nœud représente une fonction calculée et on suppose que les dérivées partielles de ces fonctions sont connues. On pourrait illustrer ce point par une fonction implémentée dans la librairie utilisée.

Lorsqu'un nœud est traité, l'ensemble de ses dérivées partielles $(\partial g/\partial x_i)(x)$, où x est bien l'entrée du nœud de calcul et non de la fonction f, sont calculées et stockées sur le nœud lui-même. La figure 3 représente cette idée. À ce stade, la seule information connue et calculée est l'ensemble des dérivées partielles d'un nœud par rapport à chacune de ses entrées qui sont soit les arguments de la fonction f, soit un paramètre de la fonction f, soit un calcul issu d'un nœuf parent. On remarquera qu'en réalité, seuls les dérivées partielles permettant

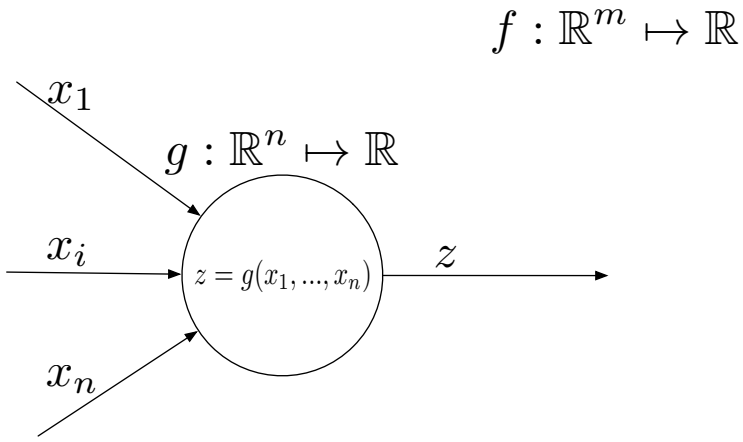

FIGURE 2 – Une fonction f et un "nœud" de cette fonction.

de remonter jusqu'aux arguments pour lesquels on veut la dérivée partiel sont calculées.

3.2.2 Backward

La seconde étape appelée *backward* exploite les informations calculées précédemment afin de produire une ou des dérivées partielles (et donc le gradient). Cette fois-ci, le parcours part de la sortie de la fonction et remonte jusqu'à ses arguments. À chaque nœud, l'opération suivante est répétée. Soit z la sortie du nœud courant, et $\partial f/\partial z$ la dérivée partielle de notre fonction par rapport à cette sortie. Il suffit de multiplier par les dérivées partielles calculées lors du forward pour obtenir les dérivées partielles de notre fonction f par rapport aux arguments de notre nœud courant et ainsi remonter au fur et à mesure le graphe de notre fonction. En remontant récursivement, on arrive aux arguments mêmes de la fonction et le gradient est calculé. Le point de départ est bien entendu l'ensemble des dérivées partielles du dernier nœud de notre graphe par rapport à ses arguments.

Notons que la sortie d'un nœud peut être utilisée par plusieurs autres nœud de calcul. Ce n'est pas gênant et il suffit d'additionner l'ensemble des dérivées partielles avant "d'exécuter" le nœud courant. En effet, on a par exemple :

$$\frac{\partial f(u(z), v(z))}{\partial z} = \frac{\partial f}{\partial u}\frac{\partial u}{\partial z} + \frac{\partial f}{\partial v}\frac{\partial v}{\partial z}.$$

La figure 4 illustre ce phénomène.

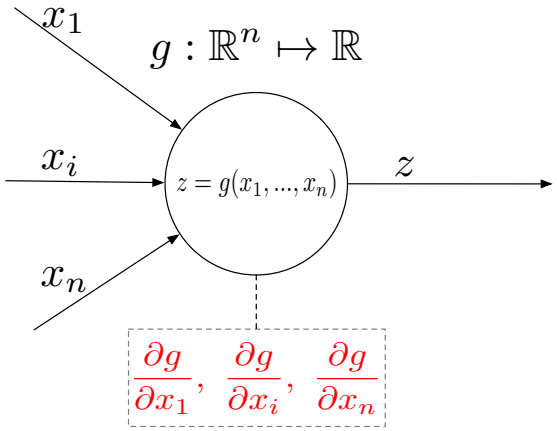

FIGURE 3 – Phase forward, les dérivées partielles sont calculées et stockées.

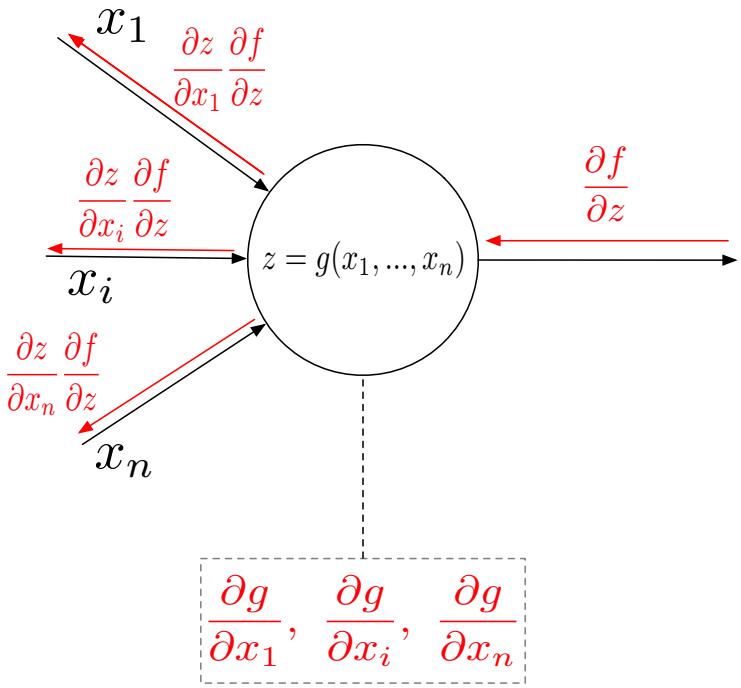

FIGURE 4 – L'opération de backward.

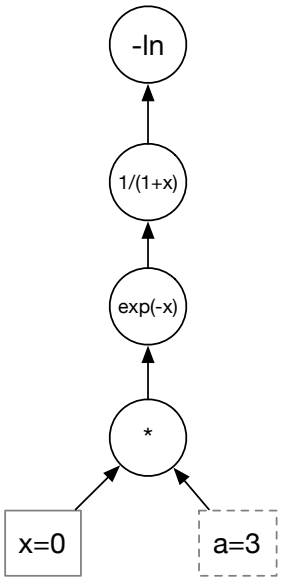

FIGURE 5 – Un graphe représentant une fonction.

3.2.3 Exemple simple

L'approche de différentiation automatique se construit autour de deux étapes : le forward et le backward. Illustrons ce mécanisme avec la fonction :

$$f(x) = -\ln\left(\frac{1}{1+e^{-ax}}\right).$$

Supposons que nous disposons des fonctions $-ln(x)$, $\exp(-x)$ et $1/(1+x)$ notamment. Notons que ces fonctions pourraient très bien se décomposer de manière encore plus élémentaire. Ces fonctions étant fournies par notre "librairie", leur dérivée est connue :

$$\begin{aligned}(-\ln(x))' &= -\frac{1}{x}\\ (\exp(-x))' &= -\exp(-x)\\ \left(\frac{1}{1+x}\right)' &= -\frac{1}{(1+x)^2}\end{aligned}$$

Le graphe de notre fonction est donné par la figure 5. Cette fonction fait évidemment référence à la régression logistique et à sa log-vraisemblance (i.e. entropie croisée).

Prenons $a = 3$ et calculons $\frac{\partial f}{\partial x}(0)$. La phase de forward consiste à exécuter chaque nœud de calcul et pour chacun à calculer ses dérivées partielles par rapport à ses entrées (on se restreint à celles qui nous intéresse). Ce calcul

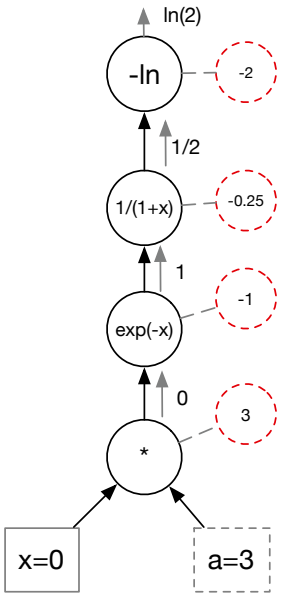

FIGURE 6 – Le calcul du forward est l'accumulation des dérivées partielles. Le calcul de ces dernières est représenté par les cercles en pointillés rouges. Les flèches grises et les valeurs associées représentent les exécutions de nos nœud de calcul.

est illustré par la figure 6. Les flèches grises représentent l'évaluation de notre fonction et les cercles en pointillés rouges, l'accumulation des dérivées partielles.

La seconde phase consiste à partir du haut et à remonter le graphe en multipliant les dérivées partielles (et éventuellement en sommant) jusqu'à arriver aux arguments de notre fonction. Cela est illustré par la figure 7.

Le résultat de cette exécution nous montre qu'on a :

$$\frac{\partial f}{\partial x}(0) = -\frac{3}{2}.$$

4 Exemple avec un réseau de neurones

Un réseau de neurones n'est rien d'autres qu'une grosse fonction composée de petites fonctions élémentaires. Ce sont ces fonctions élémentaires qu'on appelle souvent "neurone". De plus, la manière dont ces dernières sont reliées entre elles, par combinaison linéaire et composition fait penser à un réel réseau de neurones. Un neurone est ce qu'on appelle une fonction *Ridge*, c'est-à-dire une fonction paramétrique de la forme :

$$g_{\theta,b} : \mathbb{R}^m \mapsto \mathbb{R},\ \theta \in \mathbb{R}^m, b \in \mathbb{R}$$
$$x \to \sigma(\langle \theta, x \rangle + b)$$

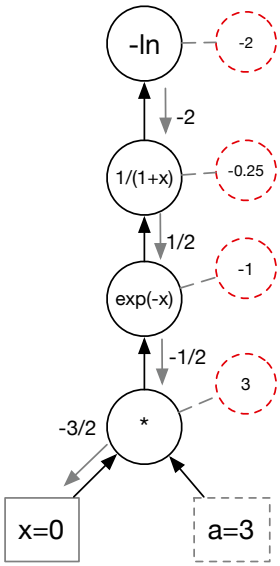

FIGURE 7 – L'opération de backward. Les flèches grises et leur valeur associée représente le calcul des dérivées partielles résultat du produit entre les dérivées partielles locales à un nœud et celles parvenant de la sortie de notre fonction. Ainsi, la valeur $1/2$ entre $1/(1+x)$ et exp(x) est le résultat du calcul entre -2 et $-1/4$.

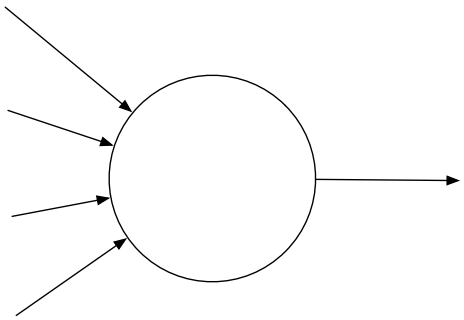

FIGURE 8 – Un neurone.

où σ est une fonction quelconque de $\mathbb{R}$ dans $\mathbb{R}$. Quelconque dans un premier temps mais certaines régularités (e.g. différentiable) seront nécessaires afin de pouvoir optimiser notre réseau de neurones. On a donc une fonction qui prend une quantité multidimensionnelle en argument et retourne une quantité scalaire. C'est cette idée qui est à la base du schéma présenté en figure 8.

Concentrons-nous sur le cas du réseau à une seule couche cachée. C'est-à-dire une fonction de la forme :

$$f(x) = \gamma(\langle w, \phi(x)\rangle + b_w)$$

où γ est la fonction sigmoid (qui tend vers 1 lors que x tend vers $+\infty$ et vers 0 lorsque x tend vers $-\infty$) et :

$$\phi(x) = [g_{\theta_1,b_1}(x), ..., g_{\theta_\omega,b_\omega}(x)]^T$$

ϕ est donc une fonction de $\mathbb{R}^d$ (dimension des données) dans $\mathbb{R}^\omega$ taille de notre couche cachée. Plus la couche cachée est grande, plus nous pouvons calculer des fonctions compliquées.

Soit une donnée x et son label y (par exemple une photo de chat et $y = 1$ pour indiquer "chat"), nous souhaitons évaluer la qualité de la prédiction :

$$l(f(x), y) = -\big(y \ln(f(x)) + (1-y) \ln(1-f(x))\big).$$

On voit ici $f(x)$ comme la probabilité que x soit associé au label $y = 1$ et si c'est le cas, on veut maximiser son logarithme et sinon, on veut maximiser $\ln(1-f(x))$. Il s'agit donc d'une métrique de qualité qui vaut 0 si notre fonction prédit bien le label correct. Calculer cette fonction pour un jeu de données se fait en moyennant les prédictions pour chacun des éléments de notre jeu de données. À partir du moment où on sait calculer le gradient pour chacun des éléments de notre somme, le généraliser à notre moyenne devient trivial. Concentrons-nous donc sur le cas où nous n'avons qu'un élément dans notre jeu de données. On souhaite minimiser notre fonction (- le log) en appliquant notre algorithme de descente de gradient. On cherche à trouver la paramétrisation de notre réseau de neurones qui fait le moins d'erreur possible (qui prédit 1 pour les chats et 0 pour les chiens). On a donc besoin de savoir calculer le gradient.

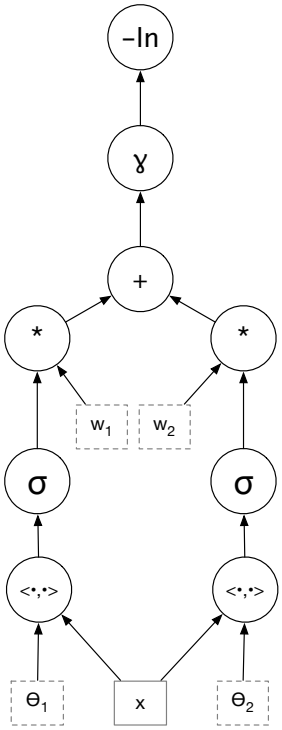

FIGURE 9 – Un réseau de neurones de largeur 2.

Supposons que tous les b_i valent 0 et supposons que $x \in \mathbb{R}^2$ et $\omega = 2$ (nous n'avons que 2 neurones). Nous avons donc $\theta_1, \theta_2 \in \mathbb{R}^2$ et $w \in \mathbb{R}^2$.

Soit $\sigma(x) = \max(0; x)$ et supposons un échantillon de la classe positif ($y = 1$). On remarque que σ est dérivable presque partout. Cela est suffisant pour optimiser un réseau de neurones. La fonction à optimiser est donc la suivante :

$$\mathcal{L}(\cdot) = -\ln(1/(1 + \exp(\langle w, [\sigma(\langle \theta_1, x\rangle, \langle \theta_2, x\rangle]\rangle))).$$

Les quantités optimisables sont les paramètres de notre modèle et on cherche en particulier : $\partial\mathcal{L}/\partial\theta_{11},\ \partial\mathcal{L}/\partial\theta_{12},\ \partial\mathcal{L}/\partial\theta_{21},\ \partial\mathcal{L}/\partial\theta_{22},\ \partial\mathcal{L}/\partial w_1,\ \partial\mathcal{L}/\partial w_1$.

Construisons le graphe de notre fonction. La figure 9 illustre ce dernier. La différentiation automatique calcule les dérivées partielles pour une paramétrisation donnée. Initialisons donc notre réseau aux valeurs illustrées par la figure 10. Les figures 11 et 12 illustrent les étapes de forward et de backward. On obtient comme attendu nos dérivées partielles à la fin de notre étape de backward.

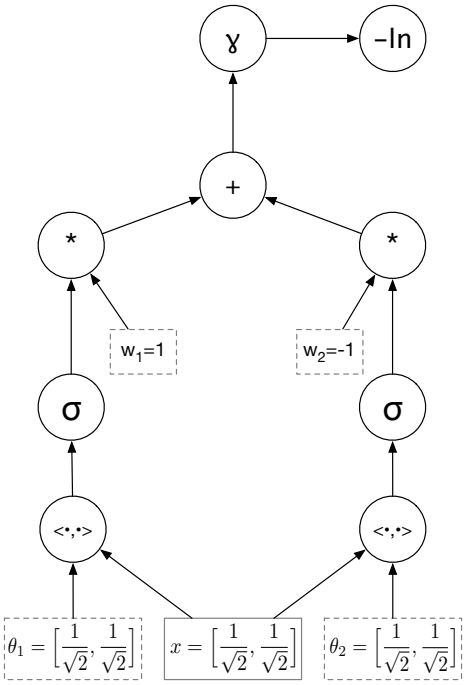

FIGURE 10 – On intialise notre réseau de neurones dans $\mathbb{R}^2$

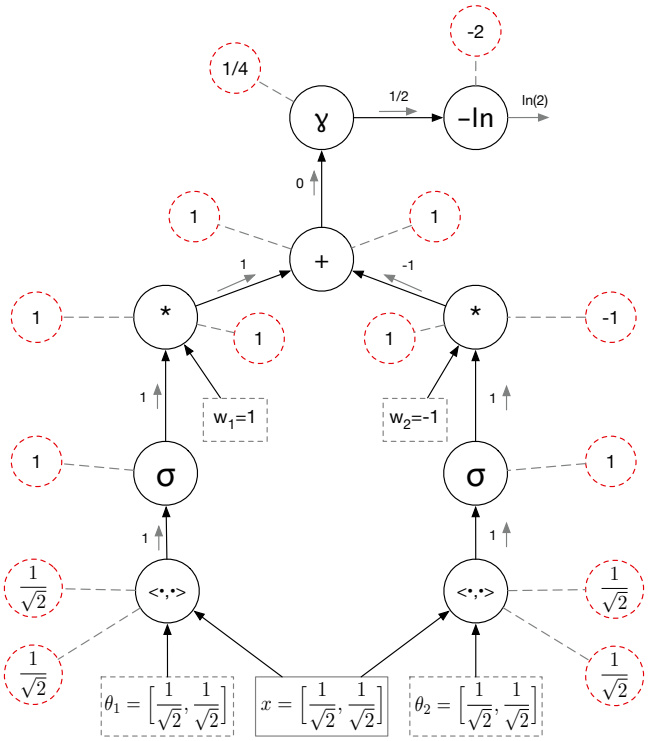

FIGURE 11 – Étape de forward permettant de calculer la *loss* de notre réseau.

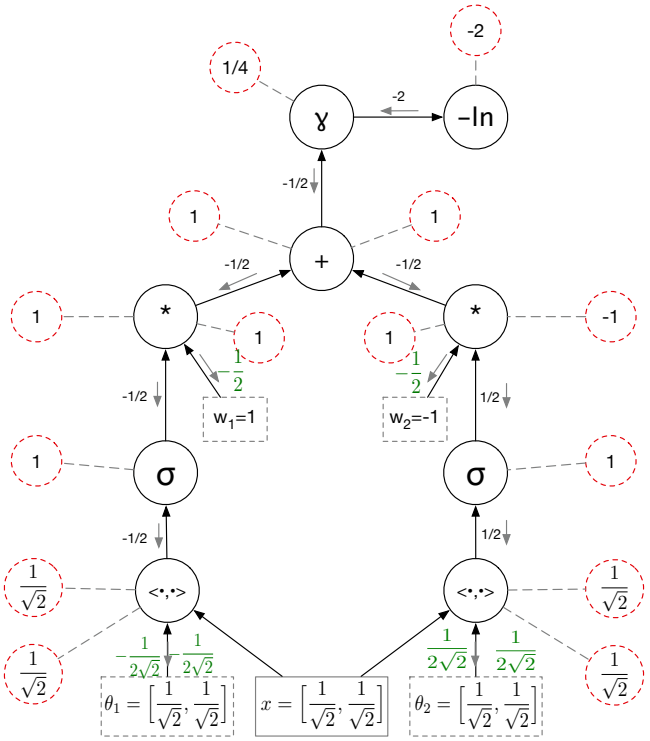

FIGURE 12 – L'étape de backward permettant d'obtenir les dérivées partielles.

Les dérivées partielles sont les suivantes :

$$\frac{\partial \mathcal{L}}{\partial \theta_{11}} = -\frac{1}{2\sqrt{2}}, \frac{\mathcal{L}}{\partial \theta_{12}} = -\frac{1}{2\sqrt{2}},$$
$$\frac{\partial \mathcal{L}}{\partial \theta_{21}} = \frac{1}{2\sqrt{2}}, \frac{\mathcal{L}}{\partial \theta_{22}} = \frac{1}{2\sqrt{2}},$$
$$\frac{\partial \mathcal{L}}{\partial w_1} = -\frac{1}{2}, \frac{\mathcal{L}}{\partial w_2} = -\frac{1}{2}.$$

Intuitivement, le poids associé à chaque filtre θ_i a la même importance à un signe près. Ainsi, le filtre θ_1 sera mis à jour à la même "vitesse" que le filtre θ_2 mais dans une direction opposée car son effet est inversé par le poids de w_2.

On remarque de plus que la symétrie de l'intialisation implique une symétrie des dérivées partielles. Si on avait choisi $w_1 = w_2$ alors les filtres θ_1 et θ_2 auraient eu la même mise à jour et auraient donc toujours calculé la même fonction ! C'est pour cela (entre autre) qu'un réseau de neurones doit être initialisé aléatoirement. Cela est bien entendu possible car notre fonction admet plusieurs minimum locaux.

Cette logique se retrouve de manière très prégnantes dans le framework de deep learning pytorch. En effet, une itération d'optimisation se résume de la manière suivante :

1. loss = $\mathcal{L}(y, f(x))$ où on obtient la sortie de la loss après une étape de forward,
2. loss.backward() où on calcule nos dérivées partielles,
3. gradient_descent.step() où on demande à notre descente de gradient de faire un pas dans la direction du gradient.