# Compte-rendu

Vincent Auriau – Laurent Beaughon – Marc Belicard
Yaqine Hechaichi – Thaïs Rahoul – Pierre Vigier

5 janvier 2017

# Table des matières

# Chapitre 1

# Réseaux de neurones

## 1.1 Apprentissage automatique

Avant de parler de neurones et de réseaux de neurones, nous commencerons par rappeler rapidement ce qu'est l'apprentissage automatique, ses différentes formes et fixerons certaines notations pour la suite de ce document.

L'apprentissage automatique au sens général peut-être défini comme un processus permettant à une machine d'évoluer afin de résoudre une tâche ou un problème. Il en existe plusieurs formes parmi lesquelles l'apprentissage supervisé, l'apprentissage non-supervisé et l'apprentissage par renforcement.

L'apprentissage supervisé est celui qui nous intéressera le plus, il sera donc davantage détaillé. Dans un problème d'apprentissage il est donné un ensemble de taille finie $N$ de couples d'entrée, sortie que nous noterons $(x^{(1)}, y^{(1)}), ..., (x^{(N)}, y^{(N)})$. Ces couples sont appelés exemples. Les entrées proviennent d'un espace $\mathcal{X}$ de dimension $n$ et les sorties d'un espace $\mathcal{Y}$ de dimension $m$. Une hypothèse classique de l'apprentissage automatique est de supposer que les entrées sont générées de manière indépendante selon une certaine loi de probabilité sur $\mathcal{X}$. Plus formellement, les $(x^{(i)})_{i\in\{1,...,N\}}$ sont les réalisations de variables aléatoires $(X_i)_{i\in\{1,...,N\}}$ avec les $X_i$ indépendantes et identiquement distribuées selon une probabilité $p$. Pour plus de commodités, notons $X = \{x^{(i)}, \forall i \in \{1, ..., N\}\}$ et $Y = \{y^{(i)}, \forall i \in \{1, ..., N\}\}$.

L'objectif de l'apprentissage supervisé est alors de découvrir le processus générateur des sorties à partir des entrées. Si ce processus est déterministe, il s'agira d'une fonction $f$ générant les sorties à partir des entrées. L'objectif de l'apprentissage sera donc de trouver à partir des exemples un modèle $\hat{f}$ proche de f. Si le processus est non déterministe, il s'agira alors de retrouver la loi de $p(y|x)$.

Si les sorties prennent leurs valeurs dans un espace $\mathcal{Y}$ infini, le problème est dit de régression. Dans le cas contraire, on parle de problème de classification. Par convention, on choisit les valeurs des classes telles que $\mathcal{Y} = \{0, ..., M-1\}$ avec $M$ le nombre de classes. Si $M = 2$, il s'agit de classification binaire.

Quelque soit le type de problème à résoudre, l'objectif est toujours de trouver une fonction ou une distribution de probabilité proche de celle générant les sorties. Cependant, cette notion de "proche" est très relative. Soit $e : \mathbb{R}^m \times \mathbb{R}^m \to \mathbb{R}$ la mesure de performance sur un exemple. $e(y, \hat{f}(x))$ représentera l'écart de notre modèle avec la sortie attendue. Une mesure courante est l'erreur quadratique $e(x, y) \mapsto \|x - y\|_2^2$. On définit alors l'erreur sur l'ensemble des exemples :

$$E_{in} = \frac{1}{N} \sum_{i=1}^{N} e(y^{(i)}, f(x^{(i)}) = \mathbb{E}_{x,y \sim \hat{p}_{data}}(e(y, f(x))) \tag{1.1}$$

Et l'erreur dans l'absolue, sur toutes les valeurs possibles :

$$E_{out} = \mathbb{E}_{x,y \sim p}(e(y, f(x))) \tag{1.2}$$

Où $\hat{p}_{data}$ est la distribution empirique dans l'ensemble des exemples et $p$ la distribution réelle des entrées et des sorties.

L'objectif d'être proche peut alors se traduire par la minimisation de $E_{out}$. Or l'estimation de $E_{out}$ nécessite de connaître $p(x, y)$ pour tout $(x, y)$ dans $\mathcal{X} \times \mathcal{Y}$, ce qui n'est pas le cas. Cependant, il est aisé de calculer $E_{in}$, qui est une estimation de $E_{out}$ sur les données disponibles. Malheureusement, les exemples servant à calculer $E_{in}$ seraient les mêmes que ceux utilisés pour choisir notre modèle. Par conséquent, le modèle serait choisi pour bien fonctionner sur ces exemples. $E_{in}$ sous-estimerait la valeur réelle de $E_{out}$ et ne serait plus un bon proxy pour $E_{out}$.

Une solution simple pour régler ce problème est de découper nos exemples en deux ensembles distincts : l'ensemble d'apprentissage et l'ensemble de test que nous noterons respectivement $(X_{train}, Y_{train})$ et $(X_{test}, Y_{test})$. Nous aurons alors :

$$E_{train} = \frac{1}{N_{train}} \sum_{i=1}^{N_{train}} e(y_{train}^{(i)}, f(x_{train}^{(i)})) \tag{1.3}$$

qui servira pour entraîner le modèle pendant l'étape d'apprentissage et :

$$E_{test} = \frac{1}{N_{test}} \sum_{i=1}^{N_{test}} e(y_{test}^{(i)}, f(x_{test}^{(i)})) \tag{1.4}$$

qui servira comme proxy de $E_{out}$ et qui permettra d'estimer si notre modèle se généralise bien à des données inconnues.

Nous utiliserons cette dernière solution dans la suite du document. Finalement il est à noter que nous parlerons indifféremment de fonction d'erreur ou de fonction de coût.

## 1.2 Inspiration biologique

On distingue principalement deux approches lorsque l'on souhaite développer une intelligence artificielle : la première consiste à analyser de façon logique les tâches relevant de la cognition humaine pour ensuite les restituer par programme, il s'agit du cognitivisme. La seconde approche quant à elle présuppose que la pensée est produite par le cerveau ou en est une propriété. Il s'agit alors d'étudier ce dernier pour bien saisir son fonctionnement et ensuite le reproduire : c'est le connexionnisme.

C'est précisément cette approche qui a conduit à la définition et l'étude des réseaux de neurones, organisations complexes d'unités de calcul élémentaires interconnectées. Si la pertinence biologique n'est pas une priorité dans le présent rapport, ce projet étant principalement motivé par la réalisation d'algorithmes efficaces, il est tout de même intéressant d'examiner quelques propriétés neurophysiologiques élémentaires pour établir des liens entre neurones réels et neurones formels. Les différentes études menées sur le cerveau montrent que celui-ci est constitué de milliards de neurones connectés entre eux. Ces derniers reçoivent des impulsions électriques par des extensions très ramifiées de leur corps cellulaire (appelées dendrites) et les transmettent par de longs prolongements (les axones). Les contacts entre deux entités se font par l'intermédiaire des synapses, chaque neurone intégrant en permanence jusqu'à un millier de signaux synaptiques. Ces influx nerveux n'opèrent pas de manière linéaire, on constate en effet la présence d'un effet de seuil. A la terminaison d'un axone, des médiateurs chimiques sont libérés et se lient à des récepteurs présents sur les dendrites : il y aura alors un effet excitateur ou un effet inhibiteur selon le message délivré.

Comment est-il donc possible de rendre compte des processus cognitifs à partir d'un ensemble d'unités ayant une faible puissance de calcul et interconnectées en réseau ?

## 1.3 Un neurone

Un neurone est modélisé par une fonction $f$ de $\mathbb{R}^n$ dans $\mathbb{R}$. Elle est déterminée par trois paramètres : un vecteur de poids $w \in \mathbb{R}^n$, un biais $b \in \mathbb{R}$ et une fonction d'activation $g$. La fonction $f$ se réécrit alors :

$$\forall x \in \mathbb{R}^n, y = f(x) = g(w^T x + b) = g(\sum_{i=1}^{n} w_i x_i + b) \tag{1.5}$$

Il est usuel de rajouter une composante $x_0$ égale à 1 à chaque entrée afin de pouvoir considérer le biais comme un simple poids. Avec cette convention, la formule 1.5 devient :

$$\forall x \in \mathbb{R}^n, y = f(x) = g(w^T x) = g(\sum_{i=0}^{n} w_i x_i) \tag{1.6}$$

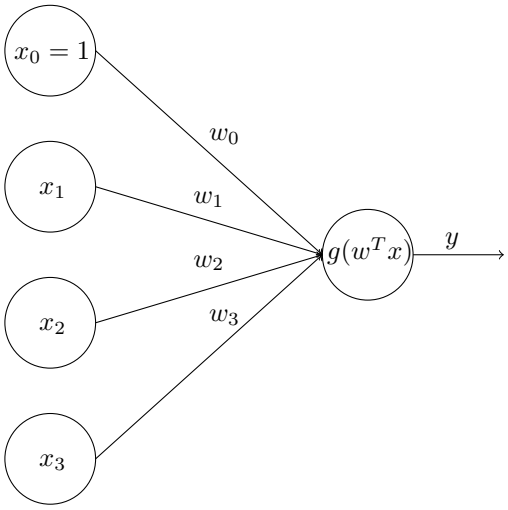


FIGURE 1.1 – Représentation graphique d'un neurone. Le biais est ici considéré comme le poids associé à une entrée constante égale à 1.

L'analogie avec le neurone biologique est ici claire. La fonction prend plusieurs entrées que l'on peut assimiler aux dendrites, les pondère et renvoie une sortie scalaire qui correspond à l'axone.

La figure 1.1 montre une représentation graphique d'un neurone.

Les principales fonctions d'activation sont :

— $Id : x \mapsto x$ dite identité ;

— $u : x \mapsto 1_{x \geq 0}$ dite échelon ou fonction de Heavyside ;

— $\tanh : x \mapsto \tanh(x)$ dite tangente hyperbolique ;

— $\sigma : x \mapsto \frac{1}{1+\exp(-x)}$ dite sigmoïde de part de sa forme en S ;

— $ReLU : x \mapsto max(0, x)$ dite ReLU (rectified linear unit).

La figure montre les graphes de ces différentes fonctions et permet de visualiser les différences entre celles-ci.

Il est possible de se servir d'un neurone seul pour résoudre des problèmes de classification ou de régression. Détaillons quelques exemples importants.

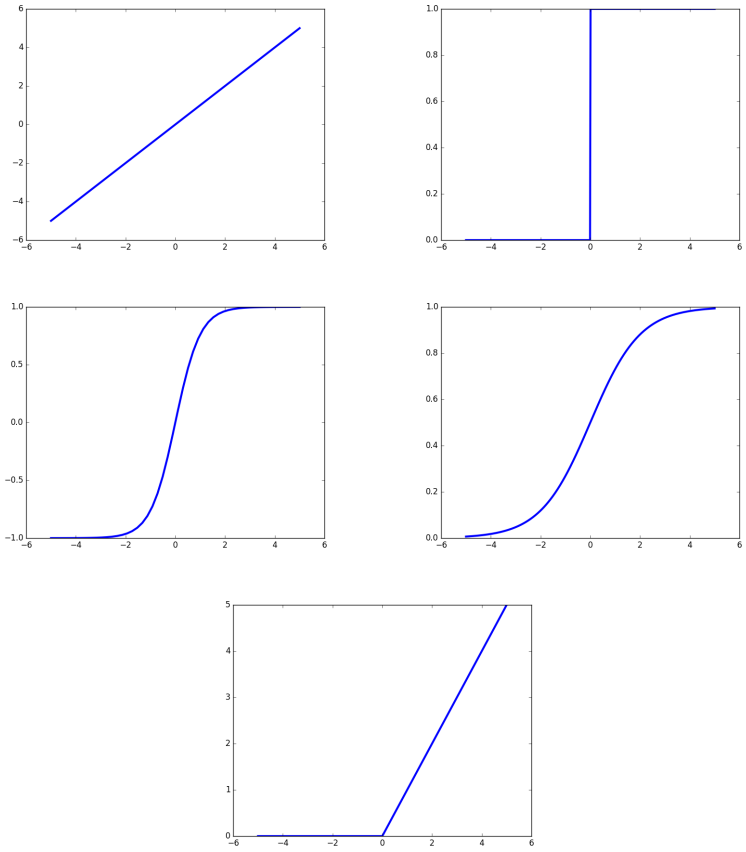

FIGURE 1.2 – Graphe entre -5 et 5 de la fonction identité (en haut à gauche), de l'échelon (en haut à droite), de la tangente hyperbolique (au milieu à gauche), de la sigmoïde (au milieu à droite) et de ReLU (en bas).

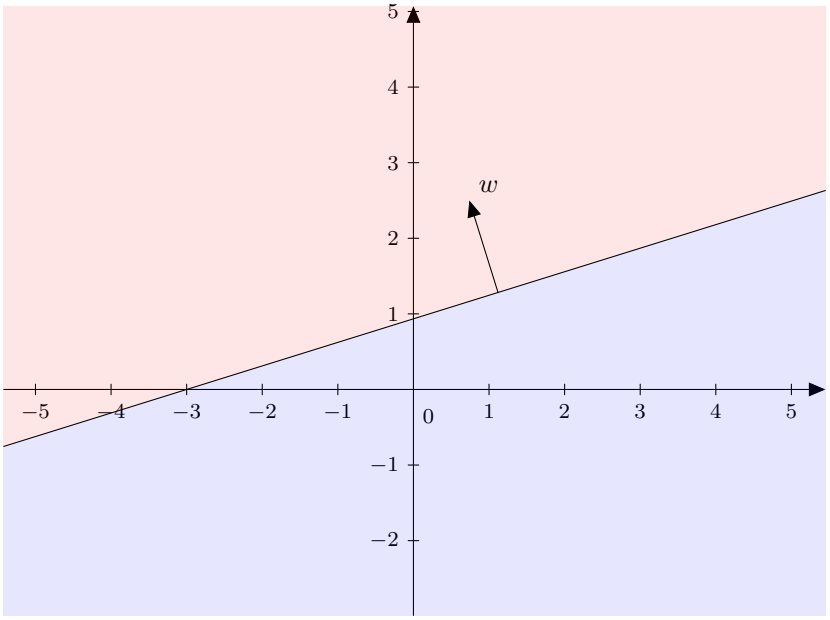


FIGURE 1.3 – Hyperplan séparateur dans un espace à deux dimensions correspondant à un certain vecteur de poids $w$. La partie rouge correspond aux entrées classifiées 1 et en bleue celles qui sont classifiées 0.

Un neurone avec comme fonction d'activation un échelon sépare l'espace en deux demis espaces séparés par un hyperplan de vecteur normal $w$. En effet :

— $\forall x \in \mathbb{R}^n, w^T x < 0 \Rightarrow f(x) = 0$

— $\forall x \in \mathbb{R}^n w^T x \geq 0 \Rightarrow f(x) = 1$

La figure 1.3 illustre cette séparation. Ce cas particulier de neurone est couramment appelé perceptron.

Si la fonction d'activation est une sigmoïde, la sortie du neurone peut-être interprétée comme la probabilité $p(y = 1|x)$. Par conséquent, le neurone classifie un exemple dans la classe 1 si la sortie est supérieure à 0.5 et dans la classe 0 sinon. On parle ici de régression logistique. L'espace est toujours séparée en deux parties par un hyperplan, $H = \{x, f(x) = 0.5\}$. Cependant maintenant les valeurs associées aux entrées sont continues entre 0 et 1. La figure 1.4 permet de visualiser cette différence. Dans la suite de ce document, des graphes de ce type seront utilisés à plusieurs reprises afin de pouvoir visualiser l'action d'un réseau de neurones.

Finalement, si la fonction d'activation est la fonction identité, le neurone sera capable d'atteindre toutes les valeurs réelles. Il pourra donc résoudre un problème de régression. Il s'agit de la régression linéaire.

Le point commun entre tous les neurones est qu'ils sont seulement capable de modéliser des interactions linéaires entre les entrées. Cependant, à partir

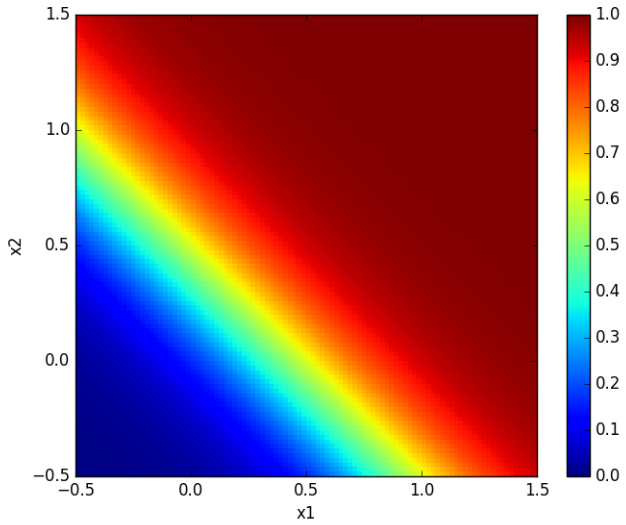


FIGURE 1.4 – Sortie d'une régression logistique prenant en entrée un vecteur de $\mathbb{R}^2$ sur $[-0.5, 1.5] \times [-0.5, 1.5]$. L'axe $x_1$ et l'axe $x_2$ correspondent respectivement à la première et à la deuxième coordonnée de l'entrée.

d'un ensemble de données, il est possible d'utiliser des neurones pour modéliser des interactions non linéaires entre les différentes composantes des entrées. Pour cela, il faut transformer les entrées afin de créer de nouvelles composantes. Par exemple, prenons un ensemble de données tel que les points ont le label 1 s'ils sont dans le disque de rayon 0.5 et de centre $(0; 0)$ et 0 sinon. Un exemple d'un tel ensemble de donnée est visible sur la figure **??**. En l'état, cet ensemble n'est pas linéairement séparable et donc un neurone simple ne peut pas le classifier correctement. Cependant si la transformation $x \mapsto (x_1^2, x_2^2)$ est appliquée aux exemples. Les points sont maintenant linéairement séparables dans ce nouvel ensemble. Il est alors possible d'utiliser un neurone afin de classifier correctement l'ensemble. Ces résultats sont présents sur la figure **??**.

Figure manquante !

En conclusion, un neurone est une fonction relativement simple permettant de modéliser des interactions linéaires entre ses entrées. Dans la partie suivante, nous verrons comment associer ces neurones afin de créer des modèles plus complexes.

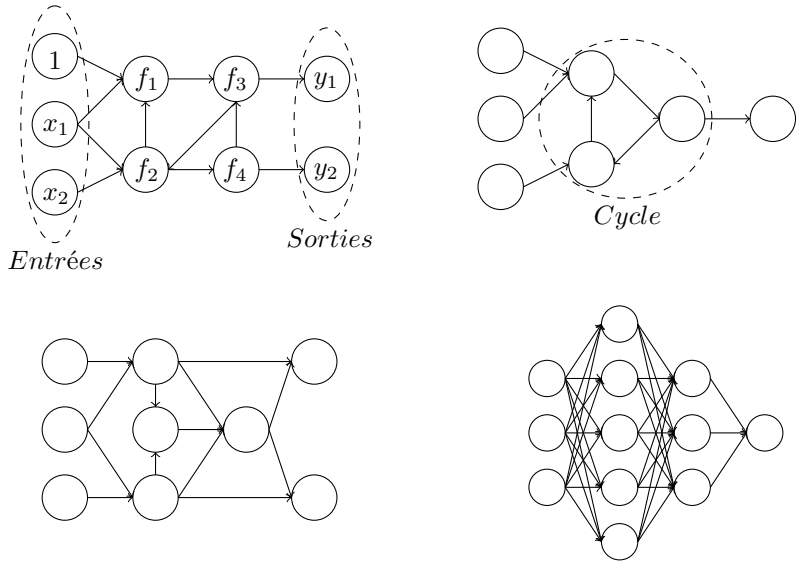


FIGURE 1.5 – Exemples de réseaux de neurones. Un réseau de neurones quelconque (en haut à gauche), récurrent (en haut à droite), feedforward (en bas à gauche) et multicouche (en bas à droite).

## 1.4 Un réseau de neurones

Un réseau de neurones est défini par un graphe orienté $\mathcal{G}(V, A)$ où les nœuds sont des neurones et les arêtes des liens entre les neurones et par un ensemble de neurones d'entrée $V_{in} \subset V$ et de neurones de sorties $V_{out} \subset V$. Une arête partant d'un neurone $i$ vers un neurone $j$ signifie que la sortie du neurone $i$ est une entrée pour le neurone $j$. Notons $f_j$ la fonction représentant le neurone $j$. Finalement, nous noterons un réseau $\mathcal{N}(V, A, V_{in}, V_{out}, f)$ où $f$ est l'ensemble des fonctions des neurones.

Si nous notons $Pa$ l'application qui à un neurone renvoie les indices de ses parents et $Ch$ l'application qui à un neurone renvoie les indices de ses enfants. Alors la sortie du neurone $j$ est :

$$y_j = f_j((y_i)_{i \in Pa(j)}) \tag{1.7}$$

Il existe certaines classes particulière de réseau de neurones. Tout d'abord, s'il contient des cycles, il sera dit récurrent. Dans le cas contraire, on parlera de réseau *feedforward*. Puis s'il est possible de l'organiser sous forme de couches où les sorties des neurones d'une couche sont les entrées de la couche suivante, on parle de réseau de neurones multicouche dit aussi MLP (multilayer perceptron). Des exemples des différents types de réseaux sont présents sur la figure 1.5.

Dans les sections suivantes, seront étudiés les algorithmes pour évaluer un réseau de neurones puis pour l'entraîner. Cependant, seulement les algorithmes dans le cadre d'un réseau de neurones feedforward seront abordés. Les algorithmes concernant les réseaux de neurones récurrent seront abordés dans la partie ? ?.

## 1.5 Propagation

L'algorithme de propagation dans le cadre des réseaux feedforward est très simple. Il suffit de mettre à jour les neurones d'entrée puis d'appliquer la formule 1.7 récursivement en partant des neurones dans $V_{out}$.

Ainsi, un premier algorithme naïf est :

**Algorithm 1** Algorithme d'évaluation d'un réseau de neurone feedforward utilisant la mémoïsation afin de ne pas recalculer plusieurs fois la sortie d'un neurone.

```
procedure evaluer_reseau(N(V, A, V_in, V_out, f), x)
    function evaluer_neurone(j)
        if j ∈ V_in then
            return x_j
        else
            t ← (evaluer_neurone(i), i ∈ Pa(j))
            return f_j(t)
        end if
    end function
    for j ∈ V_out do
        y_j ← evaluer_neurone(j)
    end for
end procedure
```

Cette version n'est pas du tout efficace, elle calcule plusieurs fois la sortie de chaque neurone. Si le graphe est densément connecté, l'algorithme aura un coût temporel extrêmement élevé. Une amélioration simple consiste à rajouter de la mémoïsation afin de garantir que la sortie de chaque neurone sera calculée au plus une fois. Le pseudo-code de cet algorithme corresponde à l'algorithme 3.

Cet algorithme calcule au plus une fois la sortie de chaque neurone et il appelle *evaluer_neurone* au plus $|A|$ fois, il a donc un coût temporel en $O(|V|+|A|)$. Ce qui est a priori le coût optimal.

## 1.6 Fonctions de coût

Avant de parler des algorithmes d'apprentissage et de la manière de modifier les poids avant d'approcher une fonction, nous allons revenir sur les fonctions de coût. Dans la suite, nous justifierons l'utilisation et le bien-fondé des fonctions de coût les plus courantes.

**Algorithm 2** Algorithme naïf d'évaluation d'un réseau de neurone feedforward. Il prend en entrée un réseau de neurone et un vecteur d'entrée pour le réseau de neurone.

**procedure** $evaluer\_reseau(\mathcal{N}(V, A, V_{in}, V_{out}, f), x)$
  **function** $evaluer\_neurone(j)$
    **if** $déjàCalculé_j$ est $faux$ **then**
      $t \leftarrow (evaluer\_neurone(i), i \in Pa(j))$
      $y_j \leftarrow f_j(t)$
      $déjàCalculé_j \leftarrow vrai$
    **end if**
    **return** $y_j$
  **end function**
  Initialiser un tableau $dejaCalculé$ de longueur $|V|$ à $faux$.
  **for** $i \in V_{in}$ **do**
    $y_i \leftarrow x_i$
    $déjàCalculé_i \leftarrow vrai$
  **end for**
  **for** $j \in V_{out}$ **do**
    $evaluer\_neurone(j)$
  **end for**
**end procedure**

Il sera détaillé trois fonctions de coût : une pour la régression, une pour la classification binaire et une pour la classification à plusieurs classes. La même méthode sera utilisée à chaque fois : un modèle sera proposé puis la fonction de coût sera déduite en utilisant le principe du maximum de vraisemblance.

### 1.6.1 Régression

Dans un problème de régression, il est courant de choisir comme modèle :

$$\hat{p}(y|x) \sim \mathcal{N}(\hat{y}(x,\theta), \sigma) \Leftrightarrow \hat{p}(y|x) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(\frac{-(y-\hat{y})^2}{2\sigma^2}\right) \tag{1.8}$$

Où $\theta$ est l'ensemble des poids de notre réseau, $\hat{y}(x,\theta)$ est la sortie de notre réseau quand l'entrée est $x$ et $\sigma$ une constante. On se permettra de noter seulement $\hat{y}(x)$ au lieu de $\hat{y}(x,\theta)$ pour plus de légèreté.

La fonction de vraisemblance est alors :

$$L(y^{(1)}, ..., y^{(N)}, x^{(1)}, ..., x^{(N)}; \theta) = \prod_{i=1}^{N} \hat{p}(y^{(i)}|x^{(i)}) \tag{1.9}$$

Au lieu de maximiser la vraisemblance, nous allons minimiser l'opposé du logarithme de la vraisemblance (*negative log-likelihood* en anglais). On obtient alors :

$$NLL(y^{(1)}, ..., y^{(N)}, x^{(1)}, ..., x^{(N)}; \theta) = -\sum_{i=1}^{N} \log \hat{p}(y^{(i)}|x^{(i)}) \tag{1.10}$$

Cette forme est préférée car la somme est plus facile à dériver que le produit. En outre, l'opposé du logarithme de la vraisemblance est pris pour se ramener à un problème de minimisation.

En remplaçant $\hat{p}$ par l'expression 1.8, on obtient :

$$NLL(y^{(1)}, ..., y^{(N)}, x^{(1)}, ..., x^{(N)}; \theta) = \sum_{i=1}^{N} (y^{(i)} - \hat{y}(x^{(i)}))^2 + C \tag{1.11}$$

où $C$ est une constante.

Dans le problème de minimisation, la constante peut-être négligée et il est possible de multiplier par une constante positive sans changer le problème. On déduit que sous cette modélisation, la fonction de coût obtenue par le principe de maximum de vraisemblance est :

$$E = \frac{1}{N} \sum_{i=1}^{N} (y^{(i)} - \hat{y}(x^{(i)}))^2 = \frac{1}{N} \sum_{i=1}^{N} e(y^{(i)}, \hat{y}(x^{(i)})) \tag{1.12}$$

avec $e : (x, y) \mapsto (x - y)^2$. Cette fonction est appelée erreur quadratique moyenne (*mean squared error* en anglais).

### 1.6.2 Classification binaire

Dans le cas de la classification binaire, à $x$ fixé, $p(y|x)$ suit une loi de Bernoulli. Un modèle possible est donc :

$$\hat{p}(y|x) \sim \mathcal{B}(\hat{y}(x, \theta)) \Leftrightarrow \hat{p}(y|x) = \begin{cases} \hat{y}(x) \text{ si } y = 1 \\ 1 - \hat{y}(x) \text{ sinon} \end{cases} \tag{1.13}$$

La formule 1.13 peut se réécrire de manière plus compacte :

$$2 + 2 = 4$$

$$\hat{p}(y|x) = \hat{y}(x)^y (1 - \hat{y}(x))^{1-y} \tag{1.14}$$

En utilisant cette dernière forme, on obtient que l'expression de la *negative log-likelihood* est :

$$NLL(y^{(1)}, ..., y^{(N)}, x^{(1)}, ..., x^{(N)}; \theta) = -\sum_{i=1}^{N} y^{(i)} \log \hat{y}(x^{(i)}) + (1 - y^{(i)}) \log(1 - \hat{y}(x^{(i)})) \tag{1.15}$$

On en déduit de même que précédemment que notre fonction de coût est :

$$E = \frac{1}{N}\sum_{i=1}^{N} -(y^{(i)} \log \hat{y}(x^{(i)}) + (1 - y^{(i)}) \log(1 - \hat{y}(x^{(i)}))) = \frac{1}{N}\sum_{i=1}^{N} e(y^{(i)}, \hat{y}(x^{(i)})) \tag{1.16}$$

avec $e : (x, y) \mapsto x \log y + (1 - x) \log(1 - y)$.

### 1.6.3 Classification à plusieurs classes

Finalement, dans le cas d'une classification à plusieurs classes, $p(y|x)$ est une loi Multinoulli à $x$ fixé. Si la sortie $\hat{y}$ du réseau est un vecteur de longueur $M$ avec $M$ le nombre de classes, une sortie possible est :

$$\hat{p}(y|x) \sim \mathcal{M}(\hat{y}(x, \theta)_0, ..., \hat{y}(x, \theta)_{M-1}) \Leftrightarrow \hat{p}(y|x) = \hat{y}(x)_y \tag{1.17}$$

La *negative log-likelihood* associée est donc :

$$NLL(y^{(1)}, ..., y^{(N)}, x^{(1)}, ..., x^{(N)}; \theta) = -\sum_{i=1}^{N} \log \hat{y}(x^{(i)})_{y^{(i)}} \tag{1.18}$$

Posons pour tout $i \in \{0, ..., M - 1\}$, $\tau^{(i)} \in \mathbb{R}^M$ le vecteur tel que :

$$\tau_j^{(i)} = \begin{cases} 1 \text{ si } i = j \\ 0 \text{ sinon} \end{cases}$$

Il est alors possible de réécrire la formule 1.18 en utilisant ces vecteurs :

$$NLL(y^{(1)}, ..., y^{(N)}, x^{(1)}, ..., x^{(N)}; \theta) = \sum_{i=1}^{N} -\sum_{j=0}^{M-1} \tau_j^{(y^{(i)})} \log \hat{y}(x^{(i)})_j \tag{1.19}$$

Enfin en introduisant l'entropie croisée $H$ définie de manière générale par :

$$H(p, q) = -\sum_x p(x) \log q(x)$$

On obtient que :

$$NLL(y^{(1)}, ..., y^{(N)}, x^{(1)}, ..., x^{(N)}; \theta) = \sum_{i=1}^{N} H(\tau^{(y^{(i)})}, \hat{y}(x^{(i)})) \tag{1.20}$$

On a remplacé $y$ par un vecteur prenant un 1 à la position $y$, une telle transformation est couramment appelé encodage *one-hot*.

Notre fonction de coût sera donc :

$$E = \frac{1}{N}\sum_{i=1}^{N} H(\tau^{(y^{(i)})}, \hat{y}(x^{(i)})) = \frac{1}{N}\sum_{i=1}^{N} e(y^{(i)}, \hat{y}(x^{(i)})) \tag{1.21}$$

Avec $e : (x, y) \mapsto H(\tau^x, y)$. Cette fonction de coût est naturellement appelée entropie croisée (*cross-entropy* en anglais).

## 1.7 Descente du gradient

Étant donné une fonction de coût $E$, l'objectif de l'apprentissage est de déterminer les poids permettant de minimiser $E$. Dans l'idéal, il faudrait pouvoir exprimer $E$ en fonction des poids, dériver $E$ par rapport à chaque poids puis chercher les valeurs des poids atteignant les minimas. Malheureusement, dans la plupart des cas, il est impossible d'avoir de telles solutions analytiques.

Une réponse à ce problème est la descente du gradient. Le principe est simple, il s'agit d'un procédé itératif consistant à modifier légèrement les valeurs des poids afin qu'à chaque étape $E$ décroisse. En notant, $\theta$ le vecteur contenant les poids de tous les neurones du réseau. La valeur de ces poids changera à chaque itération, par conséquent, notons $\theta(t)$ la valeur de $\theta$ à l'itération $t$. La direction selon laquelle $E$ augmente le plus est $\frac{\partial E}{\partial \theta}$, la direction selon laquelle $E$ décroît le plus est donc $-\frac{\partial E}{\partial \theta}$. Si les poids sont modifiés en se déplaçant dans cette direction d'un pas $\eta(t)$ assez petit, le coût diminuera. L'équation d'évolution est donnée par 1.22.

$$\theta(t+1) = \theta(t) - \eta(t)\frac{\partial E}{\partial \theta} \tag{1.22}$$

Il existe plusieurs stratégies afin de choisir le pas $\eta(t)$. Dans un premier temps et pour plus de simplicité, nous choisirons un pas $\eta(t) = \eta$ constant.

## 1.8 Rétropropagation du gradient

L'application de l'algorithme précédant nécessite le calcul des dérivées partielles par rapport à chacun des poids du réseau de neurones. Prenons le neurone $j$ du réseau de neurone. Son entrée est notée $x_j$, sa sortie $y_j$, sa fonction d'activation $g_j$ et ses poids $w_j$. De plus, définissons la quantité intermédiaire $s_j = w_j^T x_j$. La formule 1.6 appliquée à ce neurone est donc $y_j = g_j(w_j^T x_j) = g(s_j)$. En utilisant la règle de la chaîne, on obtient que :

$$\frac{\partial E}{\partial w_j} = \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial s_j}\frac{\partial s_j}{\partial w_j} \tag{1.23}$$

On peut remarquer que le terme $\frac{\partial s_j}{\partial w_j}$ est égal à $x_j$ et que $\frac{\partial y_j}{\partial s_j} = g'(s_j)$ d'où l'égalité 1.23 devient :

$$\frac{\partial E}{\partial w_j} = \frac{\partial E}{\partial y_j} g_j'(s_j) x_j \tag{1.24}$$

Afin de calculer $\frac{\partial E}{\partial y_j}$, utilisons le fait que la sortie du neurone $j$ est une des entrées de ses enfants. En notant $x_{i,j}$ la coordonnée correspondant à la sortie du neurone $j$ dans l'entrée du neurone $i$, on a :

$$\frac{\partial E}{\partial y_j} = \sum_{i \in Ch(j)} \frac{\partial E}{\partial x_{i,j}} \tag{1.25}$$

La formule 1.25 montre que si on calcule $\frac{\partial E}{\partial x_i}$, la dérivée du coût par rapport aux entrées d'un neurone, il est aisé de calculer la dérivée du coût par rapport aux poids pour les parents de ce neurone. Or la dérivée du coût par rapport aux entrées d'un neurone est facilement calculable en utilisant l'égalité suivante :

$$\frac{\partial E}{\partial x_j} = \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial s_j}\frac{\partial s_j}{\partial x_j} = \frac{\partial E}{\partial y_j} g_j'(s_j) w_j \tag{1.26}$$

Finalement, en utilisant les formules 1.24, 1.25 et 1.26, il est possible de rétropropager le gradient et de calculer $\frac{\partial E}{\partial w_j}$ pour tout neurone $j$. On en déduit l'algorithme suivant :

**Algorithm 3** Algorithme de rétropropagation du gradient dans un réseau de neurone feedforward.

**procedure** $retropropager\_gradient(\mathcal{N}(V, A, V_{in}, V_{out}, f), x, y)$
  **function** $calculer\_gradient(j)$
    **if** $déjàCalculé_j$ est $faux$ **then**
      $\frac{\partial E}{\partial y_j} \leftarrow \sum_{i \in Ch(j)} calculer\_gradient(i)_j$
      $\frac{\partial E}{\partial x_j} \leftarrow \frac{\partial E}{\partial y_j} g_j'(s_j) w_j$
      $\frac{\partial E}{\partial w_j} \leftarrow \frac{\partial E}{\partial y_j} g_j'(s_j) x_j$
      $déjàCalculé_j \leftarrow vrai$
    **end if**
    **return** $\frac{\partial E}{\partial x_j}$
  **end function**
  Appeler $evaluer\_reseau(\mathcal{N}(V, A, V_{in}, V_{out}, f), x)$ et récupérer les entrées et sorties de chaque neurone
  Initialiser un tableau $dejaCalculé$ de longueur $|V|$ à $faux$.
  **for** $i \in V_{out}$ **do**
    Calculer $\frac{\partial E}{\partial y_i}$
    $déjàCalculé_i \leftarrow vrai$
  **end for**
  **for** $i \in V$ **do**
    $calculer\_gradient(i)$
  **end for**
**end procedure**

Cet algorithme utilise la mémoïsation afin de ne pas calculer plusieurs fois les dérivées pour un neurone. On peut remarquer que cet algorithme est très similaire à celui de la propagation. Pour des raisons similaires, le coût temporel de la rétropropagation est aussi en $O(|V| + |A|)$.

# Chapitre 2

# Une première implémentation

## 2.1 Motivation

La première implantation a été faite sous Python avec pour but principal de rester le plus proche possible de l'architecture neuronale du réseau afin de pouvoir bien étudier le fonctionnement de l'algorithme d'apprentissage. Quitte à perdre en rapidité de calcul, nous avons ainsi décidé de créer des éléments neurones et un réseau composé de plusieurs de ces neurones. Cette approche permet une bonne compréhension des concepts de base des réseaux de neurones. Nous avons alors pu appliquer cette implémentation sur des cas simples (XOR notamment, cf Résultats), mais aussi obtenir un aperçu des optimisations possibles afin d'accélérer les calculs. Cela s'est effectivement rapidement révélé nécessaire.

## 2.2 Diagramme UML

Suivant cette volonté de créer une première implémentation simple et intuitive, le diagramme UML comporte ainsi deux classes principales : une classe Neuron et une classe Network. Ainsi un réseau (network) sera composé de plusieurs neurones (neurons).

Le neurone a été défini comme une entité autonome, qui comporte des entrées et une sortie et est caractérisé par des poids ainsi que ses relations avec d'autres neurones (parents ou enfants). Il peut alors calculer la sortie si les sorties de ses parents ont préalablement été évaluées. Pour déterminer le gradient au niveau de chaque poids, il a tout d'abord besoin de ceux de ses enfants.

On répartit les neurones en différentes catégories selon leurs fonctions d'activation (sigmoïde, tangente hyperbolique, Softmax ou ReLu par exemple).

Ainsi la classe Neuron possède de nombreuses sous-classes correspondant à

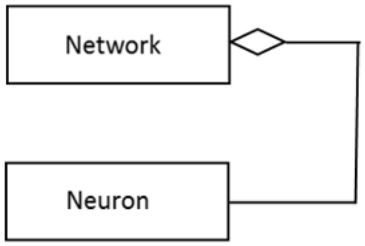


FIGURE 2.1 – UML simplifié (à changer)

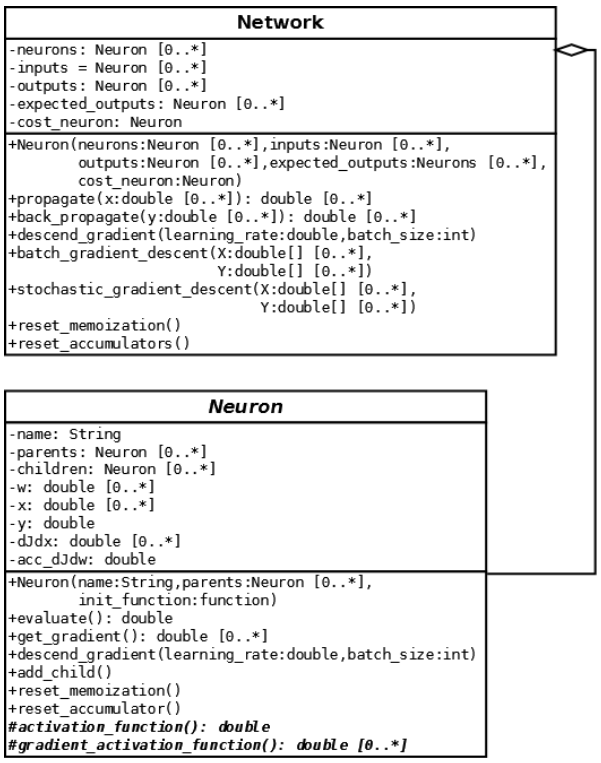


FIGURE 2.2 – UML simplifié maispastrop

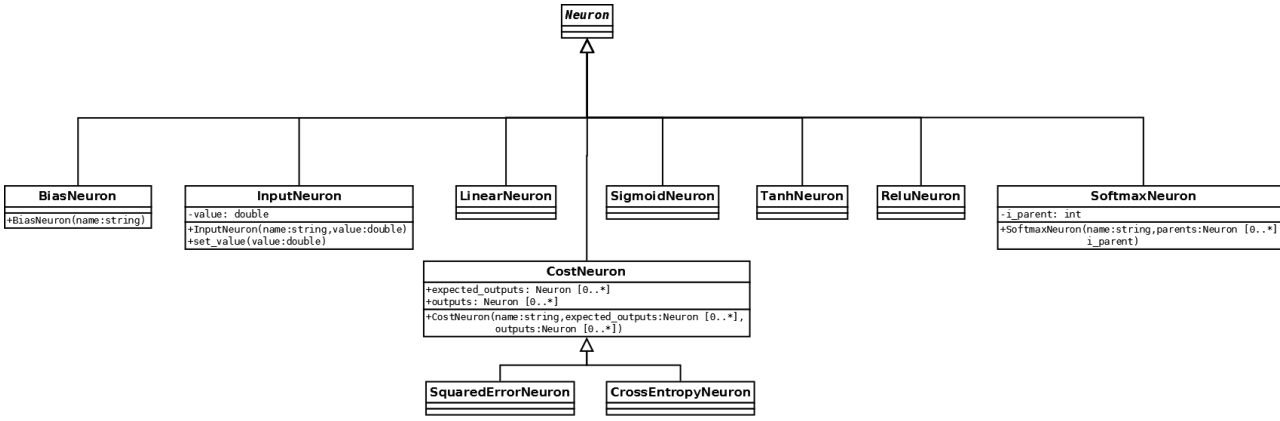


FIGURE 2.3 – Déclinaison de la classe neurone

ces fonctions. En outre, il existe plusieurs sous-classes destinées aux neurones ayant un comportement particulier. On distingue ainsi BiasNeuron, qui permet d'ajouter un biais au niveau des entrées d'un autre neurone, InputNeuron correspondant simplement aux neurones d'entrées. L'ajout d'un neurone de coût (cost neuron) à la fin du réseau permet de calculer directement l'erreur lors de la propagation d'une entrée. Il faudra ainsi spécifier pour chaque input d'entrée la sortie attendue pour calculer le coût (avec un InputNeuron).

Les liens entre neurones ne seront pas mémorisées par le réseau. Cette tâche sera réalisée par les neurones eux-mêmes. Ainsi, chacun possédera en attribut une liste de parents et une liste d'enfants, ce qui lui permettra de se situer dans le réseau. Ces deux listes sont indispensables afin de propager le résultat de sortie du neurone et afin de rétropropager le gradient lors de l'algorithme d'apprentissage.

## 2.3 Principe de fonctionnement

Pour utiliser le programme, il suffit de créer le réseau de neurones voulu. On crée pour cela différents neurones (InputNeuron, SigmoidNeuron, CrossEntropyNeuron, ...) en spécifiant les parents à chaque fois. Le programme mettra lui même à jour les listes de parents et d'enfants de chaque neurone afin de créer les différentes relations entre neurones. On crée finalement le réseau (Network) en spécifiant les entrées, sorties, le neurone de coût et les neurones intermédiaires. On peut alors appliquer deux fonctions principales sur le réseau. "Propagate" permet de calculer la sortie du réseau pour une entrée fournie en paramètre. "Batch_propagation_descent" permet d'appliquer l'algorithme de d'apprentissage basée sur la backpropagation du gradient pour un ensemble d'entrées, de sorties attendues et un learning rate $\eta$ donnés. Pour réaliser cet algorithme d'apprentissage, le programme sélectionne une entrée x et une sortie attendue

y_expected. Il applique ensuite un "propagate" afin d'obtenir la sortie y et le coût correspondant. Un backpropagate permet alors de faire remonter le gradient jusqu'à chaque neurone où il sera accumulé dans une variable acc_dJdw. On réitère ce processus pour tous les couples x et y_expected. Enfin, on met à jour les poids grâce à un "descent_gradient" à l'aide de l'équation 2.1 (valable pour un batch) :

$$w(t+1) = w(t) - \frac{\eta}{batch\_size} acc\_dJdw \tag{2.1}$$

Nous avons utilisé diverses astuces afin d'améliorer l'efficacité de notre programme. Par exemple, la sigmoide est une fonction d'activation souvent utilisée. Elle est définie par $f(x) = \frac{1}{1+\exp(-x)}$. Au lieu d'entrée directement la formule complète de la dérivée, nous la simplifions en l'écrivant sous la forme $f'(x) = f(x) * (1 - f(x))$. Nous utilisons deux variables acc_dJdw et dJdx dans chaque neurone. La première permet d'accumuler les corrections à apporter aux poids que l'on applique à la fin du batch. dJdx sert de mémoization afin d'optimiser les calculs et de ne pas en faire d'inutiles. En effet, pour calculer son gradient, chaque neurone a besoin des gradients de ses enfants. Si nous ne mémorisions pas le gradient de chaque neurone dans dJdx, nous devrions le recalculer à chaque fois qu'un des parents le demande. Cela alourdirait énormément les calculs et ferait augmenter significativement le temps d'exécution. Ces deux variables doivent évidemment être réinitialisées à la fin du passage du batch de données.

## 2.4 Résultats

Afin de tester le fonctionnement de cette première application, nous avons commencé par le faire fonctionner sur un modèle simple : le XOR. Le but était donc de réaliser un réseau neuronal à deux entrées et une sortie qui fonctionne comme un XOR : il renvoie zéros si les entrées sont semblables (égales à un ou à zéro) et il renvoie un si elles sont différentes (l'une égale à un et l'autre à zéro). On entraine alors le réseau par le batch définition du XOR : les quatre couples (0 ;0), (0 ;1), (1 ;0) et (1 ;1). Le gradient est alors calculé en moyennant les résultats du réseau sur ces quatre entrées. Cela a permis d'étudier et d'assurer le bon fonctionnement de l'implémentation.

Des premiers tests ont été réalisés avec un réseau avec une couche cachée de deux neurones. Les résultats sont alors plutôt mauvais : alors que théoriquement le XOR est réalisable avec cette architecture, nous avons pu observer que lors de l'exécution de l'algorithme, la descente du gradient a tendance à se bloquer dans un minimum local de la fonction de coût.

Nous avons alors pu remarquer que même en modifiant différents paramètres, (valeurs initiales des poids, learning rate ou les fonctions d'activation), cela restait inefficace, et les résultats obtenus ne correspondaient pas à la fonction xor que l'on attendait (voir figure 2.5).

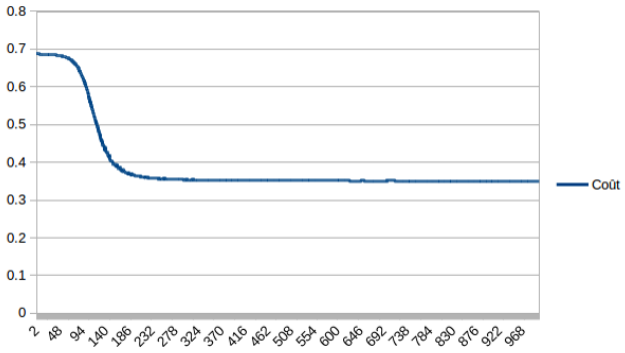


FIGURE 2.4 – fonction de coût bloquée dans un minimum local

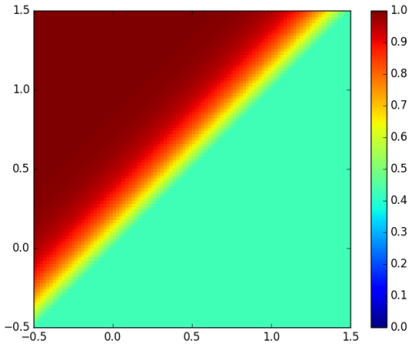


FIGURE 2.5 – XOR bloqué dans un minimum local

Nous sommes alors passés sur une seconde architecture avec cette fois quatre neurones dans la couche intermédiaire cachée. On obtient cette fois de très bons résultats comme celui de la figure 2.6.

On peut remarquer avec cette présentation des résultats que le réseau de neurones renvoie les bonnes réponses du XOR pour les entrées définies pour l'entrainement. Pour toutes les autres valeurs, le réseau "interprète" alors avec son apprentissage. On peut remarquer que cette interprétation varie selon la fonction d'activation. Ainsi pour la tangente hyperbolique les zones définies sont beaucoup plus courbées que pour la ReLu. On peut lier cela avec les représentations graphiques de ces fonctions. En effet la Relu est en fait deux demi-droites alors que la tangente hyperbolique a une courbe représentative beaucoup plus "arrondie".

L'apprentissage s'est donc bien réalisé pour le XOR, les résultats obtenus

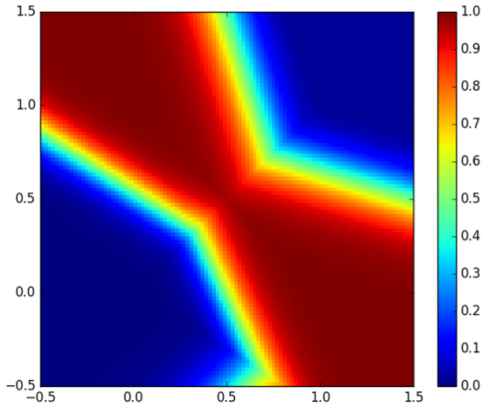


FIGURE 2.6 – 2 couches cachées de 4 neurones-ReLu

sont prometteurs pour la suite. Nous avons alors décidé de faire fonctionner l'algorithme sur les données MNIST.

MNIST est une base de données de chiffres écrits à la main réalisée par Yann Lecun. Cette base de donnée est constituée d'un ensemble de données d'apprentissage de 60.000 exemples et un ensemble de test constitué de 10.000 exemple. L'intersection de ces deux ensembles est nulle. Chaque exemple est donc une image d'une taille fixe d'un chiffre écrit à la main, centré. Le but est donc que notre algorithme puisse reconnaître les chiffres écrits.

Nous avons réalisé un premier apprentissage des données MNIST sur un réseau sans couche cachée totalement connecté (fully connected). Ce réseau dispose d'une entrée par pixel des images et de dix sorties, une par chiffre. Un premier apprentissage est réalisé sur le réseau avec un calcul du gradient moyenné sur des batchs de 128 exemples. On calcule alors la précision du réseau de neurones sur l'ensemble complet d'apprentissage ainsi que sur l'ensemble de test tous les 2.000 exemples.

On peut remarquer sur ces premiers résultats, que la précision progresse très vite avant de plafonner autour des 90% dès les 10.000 exemples utilisés. Nous avons obtenu une précision maximale de 90.83% sur cette architecture neuronale, très simpliste. Une précision plus importante pourrait être obtenue en ajoutant au moins une couche cachée au réseau. Cependant, le temps d'exécution avec cette première architecture (21.565 secondes soit plus de 6 heures), nous a montré que cela serait impossible avec de réaliser un apprentissage en un temps raisonnable avec des architectures plus compliquées.

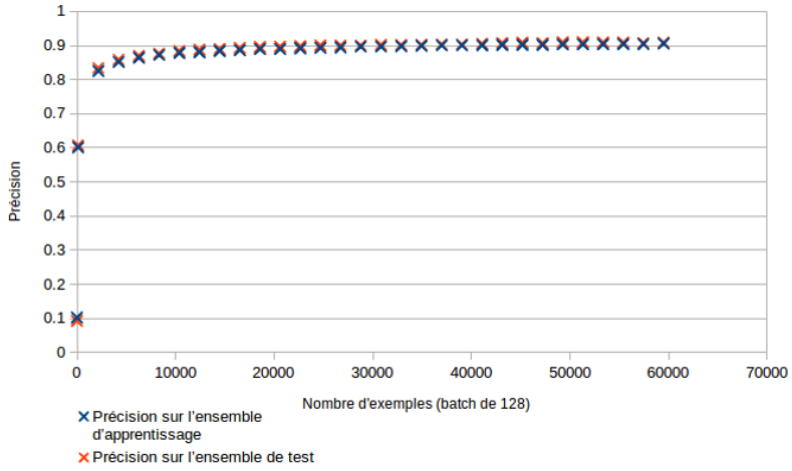


FIGURE 2.7 – Précision en fonction du nombre d'exemples appris

## 2.5 Conclusion

Cette première implémentation intuitive permet ainsi d'obtenir des résultats très satisfaisants, allant jusqu'à 90% de réussite sur MNIST. De plus, elle met en évidence le fonctionnement d'un réseau de neurones. Cependant, on remarque que les calculs ne sont pas du tout optimisés. Cela explique que le temps d'exécution devient rapidement très long. Dans l'exemple de l'application à l'ensemble de données MNIST, entraîner plusieurs fois le réseau sur l'ensemble d'apprentissage permettrait d'obtenir de bien meilleurs résultats, mais cela prendrait alors beaucoup trop de temps pour être véritablement envisageable. Afin d'améliorer le temps de calcul et d'optimiser l'algorithme, nous nous sommes intéressés à une nouvelle approche des réseaux neuronaux basée sur les Computational Graphs, ou Graphes de calculs.

# Chapitre 3

# Graphe de calculs

## 3.1 Définition

Un graphe de calcul est la représentation de fonctions composées comme un réseau d'entités interconnectés. Chacune de ces entités est une fonction ou une opération, cela permet alors une meilleure visualisation de la tâche globale effectuée par le réseau ainsi que les relations entre deux neurones successifs. On appellera par la suite nœud parent un nœud situé en amont du réseau (i.e. plus proche des inputs) par rapport à un autre (appelé alors nœud enfant) et en lien direct avec lui.

L'intérêt principal de cette représentation est la rapidité et la facilité de l'implémentation de l'algorithme de backpropagation. En effet, le caractère modulaire de cette dernière rend aisé le calcul de gradient en appliquant la règle de la chaîne, comme le montre l'exemple suivant :

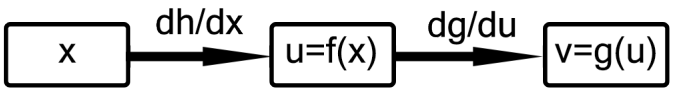


FIGURE 3.1 – Décomposotion de la rétropropagation de l'erreur dans un graphe de calcul simple.

Supposons que l'on veuille sur cet exemple simple obtenir la dérivée de g par rapport à x, il suffira alors de faire le produit des dérivées indiquées sur chaque branche connectant deux nœuds différents.

## 3.2 Diagramme UML

Là encore, l'architecture retenue contient 2 classes mères : Node et Graph, un graph étant constitué de plusieurs nodes qui connaissent chacune leurs parents et leurs enfants.

La différence principale avec l'architecture précédente est dans la modularité de la classe Node. Bien qu'elle joue un rôle similaire à celui de la classe Neuron dans le modèle Neuron/Network, elle est bien plus souple : une node peut représenter une entrée du graph, un vecteur de poids, une opération élémentaire (addition, soustraction, produit scalaire), ou une fonction (fonction d'activation d'un neurone ou fonction de coût du réseau). Une node prend pour entrée une matrice (ou 2 dans le cas des opérateurs élémentaires), et donne en sortie ses résultats sous la forme d'une matrice de taille adaptée. De même elle peut calculer le gradient par rapport à ses entrées étant donné celui de ses enfants toujours sous forme matricielle. Cette capacité des nodes à gérer les matrices est primordiale car elle permet de créer un réseau ne possédant qu'une seule node d'entrée et de propager à travers lui tout un lot d'exemples en une seule fois, diminuant ainsi grandement le nombre d'appels objet en les remplaçant par des opérations matricielles. Les résultats obtenus avec cette architecture permettront par la suite de confirmer que les appels objets et non les opérations occupaient la majeure partie du temps de calcul dans l'implémentation présentée en chapitre 1.

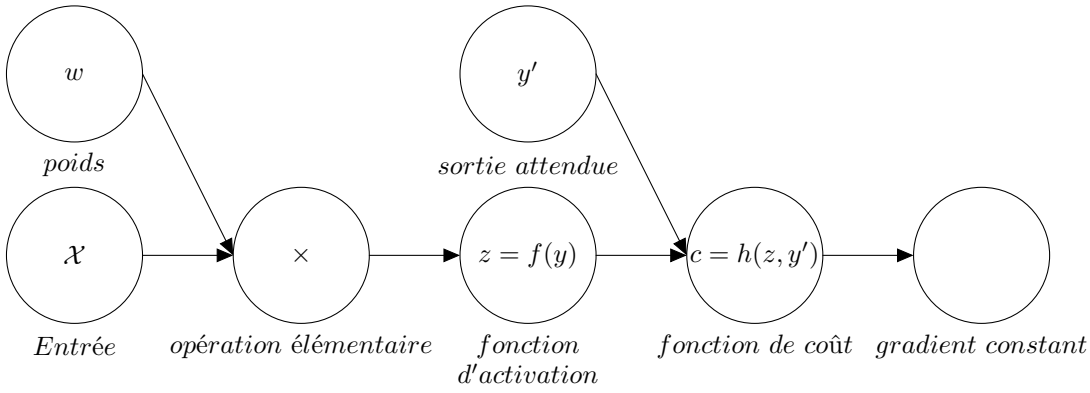


FIGURE 3.2 – Schéma d'un graphe de calcul pour un neurone simple

Un graph commence par une node d'entrée, suivie d'un ensemble de nodes de calcul dont la dernière est qualifiée de node de sortie. Une ou plusieurs node(s) de fonction de coût prennent pour parent la node de sortie, une node d'entrée qu'on actualisera aux valeurs des sorties attendues lors de l'entrainement et une node de régularisation. Enfin on clôt le graph par une node qui possède un gradient unitaire constant et aucune sortie. Elle permettra d'initier la descente du gradient dans la pile des appels récursifs des nodes depuis les entrées vers la fin du graph. C'est le graph qui gère la propagation des exemples et la descente du gradient à travers les nodes mais son rôle est très limité de par la grande autonomie qui est donnée à celles-ci. Les fonctions de propagation et rétropropagation de chaque node sont en effet codée de sorte à ce qu'un unique appel de leur fonction de propagation (respectivement rétropropagation) sur la node d'entrée (respectivement de sortie) donne lieu à une cascade d'appel jusqu'à l'obtention du résultat voulu. Dans une propagation par exemple, on appelle la méthode « propager » sur la dernière node du réseau qui l'appellera elle-même sur ses parents pour calculer sa sortie, ce qui actualisera les sorties de toutes les nodes nécessaires au calcul du résultat initialement demandé.

## 3.3 Résultats

L'architecture en Graphe de Calcul donne de très bons résultats du point de vue de la rapidité de calcul. On observe par exemple une réduction d'un facteur 1000 du temps de calcul pour l'entrainement du réseau simple sur l'ensemble MNIST par rapport à l'implémentation "naïve" du chapitre 1. Ces performances sont atteintes grâce à la diminution du nombre d'appels objets, remplacés par des calculs matriciels sur des matrices de grande taille gérées par des bibliothèques efficaces.

# Chapitre 4

# Études des paramètres

## 4.1 Approche du problème

L'implémentation basée sur les graphes de calcul nous a permis de réaliser des tests sur les deux exemples introduits précédemment : XOR et MNIST. À travers de nombreux tests, nous avons souhaité étudier l'influence des paramètres de l'algorithme sur ses performances, c'est-à-dire sur sa précision et sa rapidité.

Il existe sept paramètres que l'on peut définir pour un réseau de neurones :
— l'architecture du réseau
— le prétraitement des données
— l'initialisation des poids
— le choix des fonctions d'activations
— le choix de la fonction de coût
— la taille des batchs et le nombre de passage
— le taux d'apprentissage

L'influence d'un paramètre n'étant bien évidemment pas indépendante ni des autres paramètres ni du problème considéré, il devient rapidement délicat d'obtenir des résultats robustes. En effet, il est impossible de réaliser des mesures en faisant varier sept variables en même temps, tout en effectuant des répétitions à chaque fois pour s'assurer de la précision des relevés. Pour s'affranchir de cette difficulté, nous avons choisi de ne faire varier qu'un seul paramètre à la fois en l'intégrant dans une configuration apportant des résultats acceptables. L'influence du paramètre étudié sur plusieurs configurations permet alors d'interpoler une estimation de son comportement général. C'est donc ainsi que nous avons pu déterminer des éléments permettant de comprendre les rôles de ces paramètres et de les choisir pour optimiser l'efficacité d'un réseau de neurones. Ce sont ces éléments qui vont être présentés dans la suite de cette partie.

## 4.2 Étude du XOR

Dans un premier temps, nous avons effectués plusieurs tests sur un exemple très simple : XOR. Quelques résultats ont déjà été présentés en partie 2.4. Ils vont être rapelés et détaillés dans cette partie.

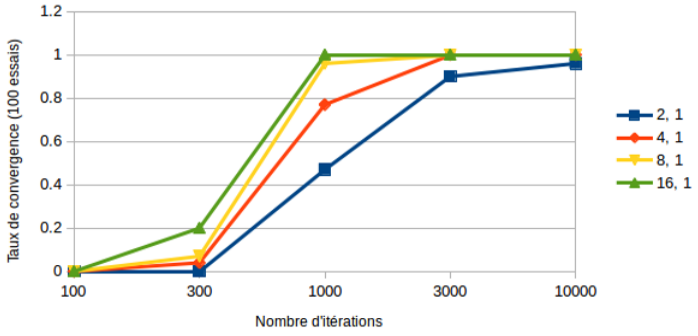


FIGURE 4.1 – Convergence en fonction de l'architecture avec tanh

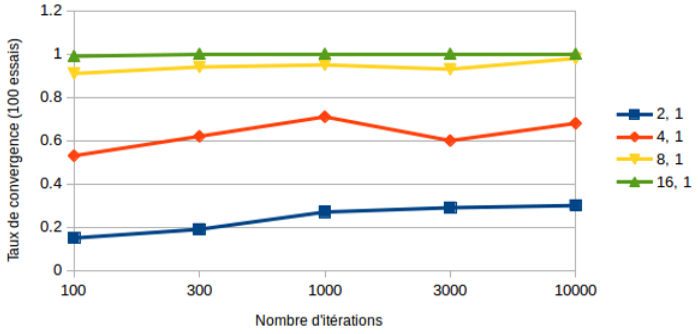


FIGURE 4.2 – Convergence en fonction de l'architecture avec relu

## 4.3 Étude de MNIST

### 4.3.1 Architecture du réseau

Le nombre de couches cachées d'un réseau de neurone ainsi que leur composition constitue l'architecture de ce réseau.

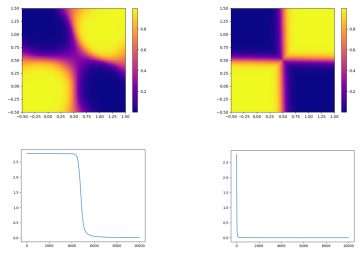

FIGURE 4.3 – Résultat après centrage des entrées pour tanh (à gauche) et pour relu (à droite)

### 4.3.2 Prétraitement des données

### 4.3.3 Initialisation des poids

Avant le lancement de l'algorithme sur un réseau de neurone, il est nécessaire d'initialiser les poids. L'usage le plus courant est de les initialiser aléatoirement. Cependant, le choix de la distribution aléatoire n'est pas forcément évident. En effet, on peut choisir d'utiliser une distribution uniforme, une distribution gaussienne ou tout autre type de distribution aléatoire avec des paramètres variables.

Nous avons donc testé trois types de fonction d'initialisation des poids : une répartition uniforme, une répartition uniforme centrée en 0 et une répartition gaussienne centrée en 0. Si les initialisations centrées en 0 permettent une convergence un petit peu plus rapide, ces trois initialisations produisent des résultats similaires. Cependant, l'amplitude des poids change beaucoup les résultats quelle que soit la fonction d'initialisation. En effet, les poids convergeant lors de l'apprentissage vers de petites valeurs, une initialisation avec de petites amplitudes va permettre une convergence plus rapide.

C'est ce que l'on peut voir sur les deux figures 4.4 et 4.5. Dans les deux cas, les poids sont initialisés selon une loi normale centrée de variance 0,1. Cependant, sur la première, un coefficient multiplicatif de 0,1 leur est affecté alors que celui-ci est de 10 sur la deuxième. On peut ainsi bien observer que dans le cas du facteur multiplicatif de 10, la convergence est plus lente au début de l'apprentissage, même si l'on atteint sensiblement les mêmes valeurs finales de précision.

### 4.3.4 Choix des fonctions d'activations

### 4.3.5 Choix de la fonction de coût

### 4.3.6 Taille des batchs et nombre de passages

### 4.3.7 Taux d'apprentissage

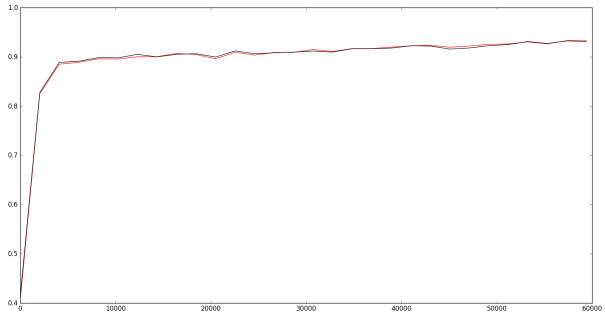


FIGURE 4.4 – Précision sur les ensembles de test (en noir) et d'apprentissage (en rouge) en fonction du nombre d'exemples utilisés pour l'apprentissage pour une initialisation des poids d'amplitude 0.1

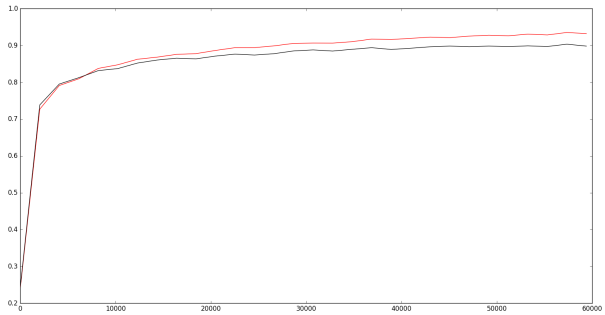


FIGURE 4.5 – Précision sur les ensembles de test (en noir) et d'apprentissage (en rouge) en fonction du nombre d'exemples utilisés pour l'apprentissage pour une initialisation des poids d'amplitude 0.1

# Chapitre 5

# Réseaux de neurones récurrents

## 5.1 Motivation

Nous avons pu voir que les réseaux feedforward peuvent être utilisés afin d'appendre à générer la sortie voulu en fonction de l'entrée du réseau. Néanmoins, ces réseaux sont limités à des entrées et des sorties de tailles fixes. De plus il ne peuvent gérer des réseaux présentant des boucles. On utilise alors des réseaux récurrents qui permettent un traitement plus efficace des séquences de données. On pourra ainsi prévoir la suite d'une séquence.

Nous nous intéresserons à deux algorithmes permettant l'apprentissage des réseaux récurrents : Real Time Recurrent Learning (RTRL) et Back Propagation Through Time (BPTT).

Dans un premier temps, nous comparerons ces algorithmes sur l'apprentissage de la grammaire de Reber, ayant un nombre d'états fini. Nous utiliserons la grammaire simple présentée sur la figure 5.1.

A l'aide de cette grammaire, nous créons un automate permettant de générer deux ensembles de mots afin d'entraîner et de tester notre futur réseau. L'objectif du réseau sera alors d'apprendre cette grammaire afin de prédire les caractères possibles après un caractère donné.

L'étape suivante consistera à tester les algorithmes sur l'apprentissage de la grammaire de Reber symmétrique présentée sur la figure 5.2.

Dans cette situation, le réseau devra être capable de se souvenir d'un état passé (T ou P) afin de prédire le nouvel état (T ou P). Cet état passé pourra être plus ou moins lointain en fonction de la taille de la grammaire de Reber insérée au milieu.

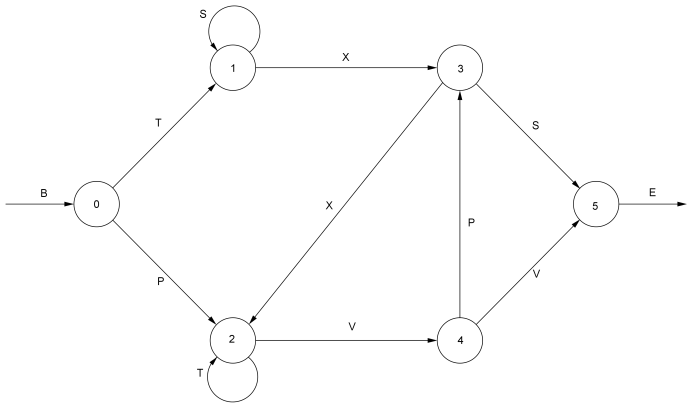


FIGURE 5.1 – Grammaire de Reber simple

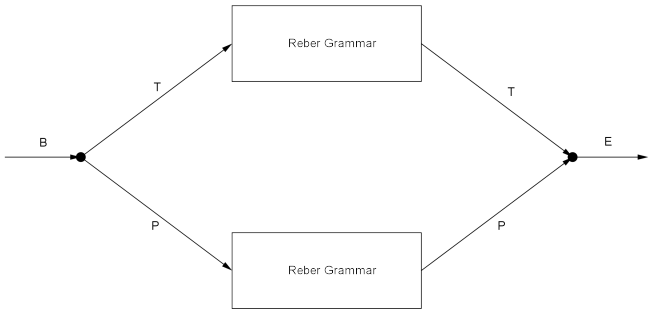


FIGURE 5.2 – Grammaire de Reber symmétrique

Enfin, nous appliquerons ces deux algorithmes à l'apprentissage de textes de Shakespeare. L'objectif sera alors de générer un texte anglais cohérent.

## 5.2 Deux algorithmes

### 5.2.1 Real Time Recurrent Learning (RTRL)

### 5.2.2 Back Propagation Through Time (BPTT)

## 5.3 Implémentation de RTRL

### 5.3.1 Principe de fonctionnement

### 5.3.2 Résultats

## 5.4 Implémentation de BPTT

### 5.4.1 Principe de fonctionnement

### 5.4.2 Résultats

## 5.5 Comparaison des deux algorithmes