

# Compte-rendu

Vincent AURIAU – Laurent BEAUGHON – Marc BELICARD  
Yaqine HECHAICHI – Thaïs RAHOUL – Pierre VIGIER

10 juin 2017

# Table des matières

<b>1</b>	<b>Une première implémentation</b>	<b>3</b>
1.1	Motivation . . . . .	3
1.2	Diagramme UML . . . . .	3
1.3	Principe de fonctionnement . . . . .	5
1.4	Résultats . . . . .	6
1.5	Conclusion . . . . .	9
<b>2</b>	<b>Études des paramètres</b>	<b>10</b>
<b>3</b>	<b>Graphe de calculs</b>	<b>11</b>
3.1	Motivations . . . . .	11
3.2	Définitions . . . . .	13
3.3	Dérivation automatique . . . . .	13
3.4	Nœuds . . . . .	14
3.4.1	La classe Node . . . . .	14
3.4.2	Les opérations mathématiques . . . . .	16
3.4.3	Dérivation matricielle . . . . .	17
3.5	La classe Graph . . . . .	21
3.6	Diagramme UML . . . . .	21
3.7	Conclusion . . . . .	21
3.8	Approche du problème . . . . .	23
3.9	Étude du XOR . . . . .	23
3.10	Étude de MNIST . . . . .	23
3.10.1	Architecture du réseau . . . . .	23
3.10.2	Prétraitement des données . . . . .	24
3.10.3	Initialisation des poids . . . . .	24
3.10.4	Choix des fonctions d'activations . . . . .	26
3.10.5	Choix de la fonction de coût . . . . .	26
3.10.6	Taille des batchs et nombre de passages . . . . .	26
3.10.7	Taux d'apprentissage . . . . .	26
<b>4</b>	<b>Réseaux de neurones récurrents</b>	<b>28</b>
4.1	Motivation . . . . .	28
4.2	Dépliage . . . . .	28

4.3	RTRL . . . . .	29
4.4	BPTT . . . . .	29
4.5	Évaluation . . . . .	29
4.5.1	Grammaire de Reber . . . . .	29
4.5.2	Shakespeare . . . . .	30
4.6	Comparaison des deux algorithmes . . . . .	30
<b>5</b>	<b>Long Short-Term Memory (LSTM)</b>	<b>31</b>
5.1	Motivation . . . . .	31
5.2	Principe de fonctionnement . . . . .	32
5.3	Première Implémentation . . . . .	35
5.4	Implémentation avec des nœuds . . . . .	35
5.5	Résultats . . . . .	35
5.6	Autres applications . . . . .	35
<b>6</b>	<b>Application : génération de musique</b>	<b>37</b>
6.1	Introduction . . . . .	37
6.2	Génération en abc . . . . .	38
6.3	Génération en midi . . . . .	38
6.3.1	Présentation du format . . . . .	38
6.3.2	Principe . . . . .	38
6.3.3	Résultats . . . . .	39
6.4	Génération de partitions . . . . .	41
6.5	Génération de signaux . . . . .	41

# Chapitre 1

## Une première implémentation

### 1.1 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.

### 1.2 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

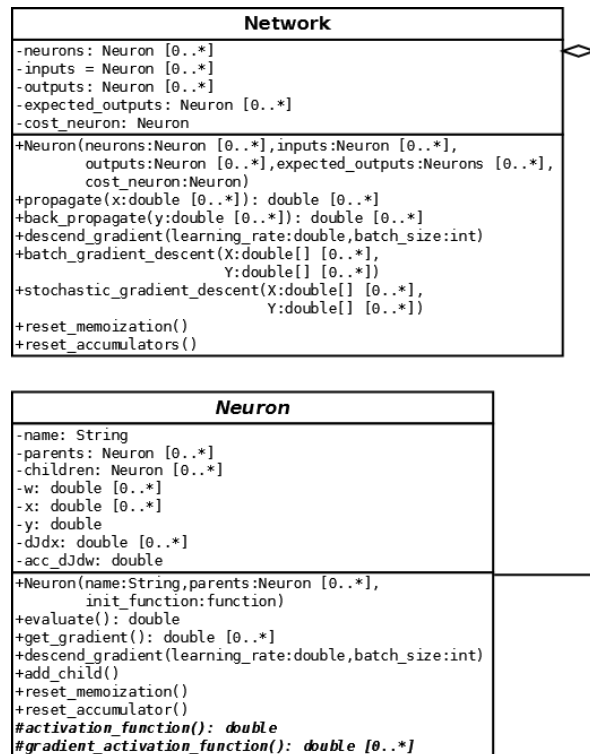


FIGURE 1.1 – Diagramme UML des classes Neuron et Network

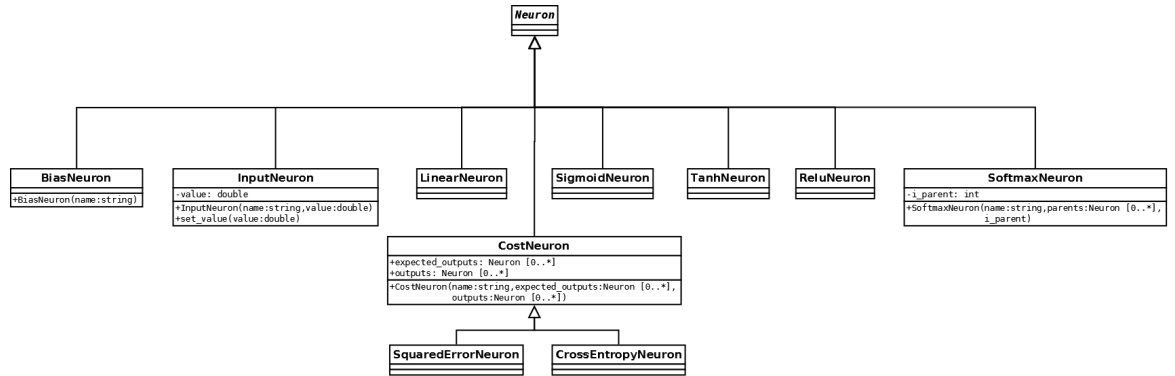


FIGURE 1.2 – Déclinaison de la classe neurone

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 mémorisés 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.

### 1.3 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 1.1 (valable pour un batch) :

$$w(t+1) = w(t) - \frac{\eta}{batch\_size} acc\_dJdw \quad (1.1)$$

Nous avons utilisé diverses astuces afin d'améliorer l'efficacité de notre programme. Par exemple, la sigmoïde est une fonction d'activation souvent utilisée. Elle est définie par  $f(x) = \frac{1}{1+\exp(-x)}$ . Au lieu d'entrer 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 mémorisation 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 mémorisons 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.

## 1.4 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 entraîne 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 1.4).

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

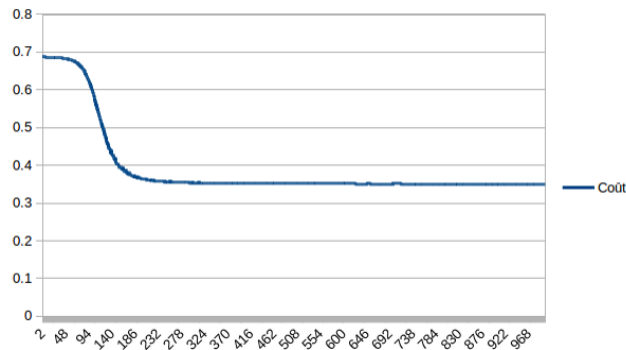


FIGURE 1.3 – fonction de coût bloquée dans un minimum local

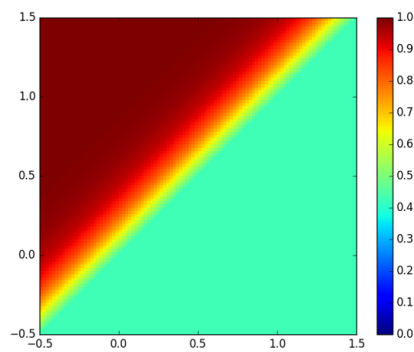


FIGURE 1.4 – XOR bloqué dans un minimum local

résultats comme celui de la figure 1.5.

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



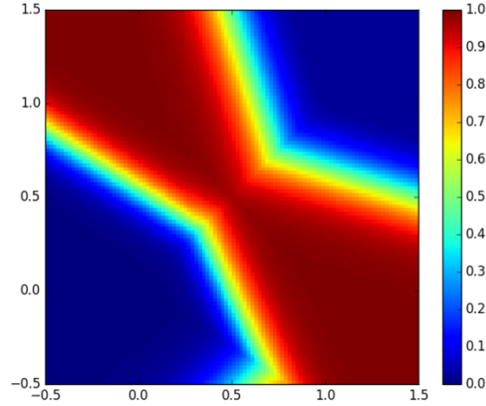


FIGURE 1.5 – 2 couches cachées de 4 neurones-ReLu

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. Chaque entrée associe au pixel une valeur entre 0 et 255 correspond à la nuance de gris du pixel (du blanc au noir). 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.

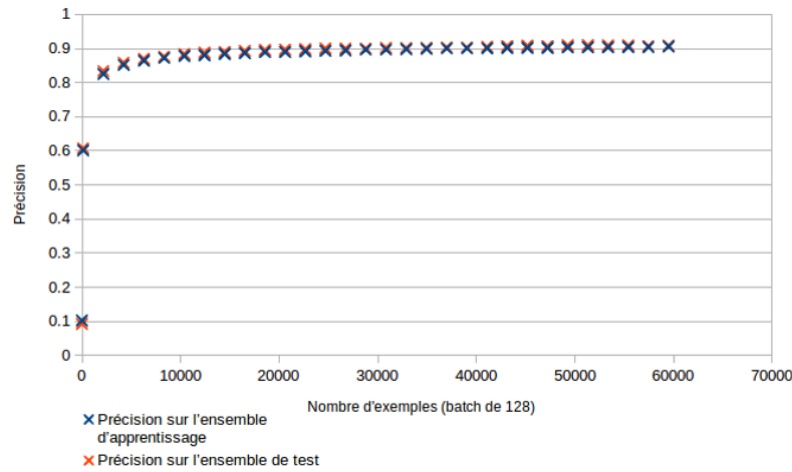


FIGURE 1.6 – Précision en fonction du nombre d'exemples appris

## 1.5 Conclusion

Cette première implémentation intuitive permet ainsi d'obtenir des résultats très satisfaisants, allant jusqu'à 90% de réussite sur le problème de 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 2

# Études des paramètres

## Chapitre 3

# Graphe de calculs

### 3.1 Motivations

Nous avons vu précédemment que l'architecture du réseau de neurones peut être lourde autant du point de vue modélisation que du point de vue calcul. Pour motiver l'apparition des graphes de calcul prenons l'exemple d'un réseau de neurones à  $m$  entrées, une couche cachée de  $n$  neurones et  $p$  sorties. Tous les neurones ont pour fonction d'activation une sigmoïde  $\sigma$ . Un tel réseau est représenté sur la figure 3.1.

Nous pourrions modéliser ce réseau de neurone en définissant  $n + p$  neurones ayant chacun son vecteur poids mais une façon équivalente de modéliser ce système est de rassembler les poids des deux couches dans deux matrices  $W_1 \in \mathbb{R}^{m \times n}$  et  $W_2 \in \mathbb{R}^{n \times p}$ . En notant  $x \in \mathbb{R}^m$  le vecteur ligne des entrées du réseau, on a alors que la sortie est :

$$y = \sigma(\sigma(xW_1)W_2) \quad (3.1)$$

où la fonction  $\sigma$  s'applique terme à terme.

Sur la figure 3.1, en bas, est représentée cette formule de manière graphique. Un tel graphe sera appelé graphe de calculs.

Cette modélisation matricielle a de nombreux avantages. Tout d'abord, elle est plus compacte et simple à manipuler et à visualiser. Puis, elle a un avantage certain pour notre implémentation en Python car elle permet d'utiliser pleinement la librairie d'algèbre linéaire `numpy` ce qui accélérera grandement les calculs. Enfin, il est facile d'étendre la formule précédente au calcul de la sortie de plusieurs vecteurs d'entrées. En effet si  $X \in \mathbb{R}^{N \times m}$  est une matrice contenant  $N$  vecteurs d'entrées alors la sortie du réseau s'écrit :

$$Y = \sigma(\sigma(XW_1)W_2)$$

On aurait donc envie de s'affranchir de la structure du neurone pour seulement modéliser les opérations mathématiques effectuées par le réseau.

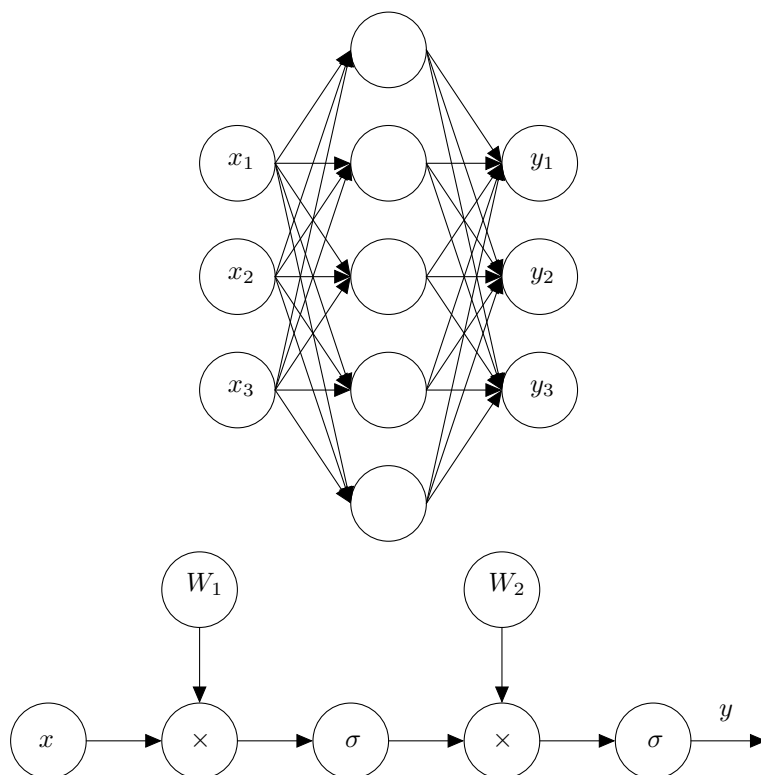


FIGURE 3.1 – Réseau de neurones à 3 entrées, 5 neurones cachés et 3 neurones de sorties (en haut). La représentation équivalente en utilisant un graphe de calculs (en bas).

## 3.2 Définitions

**Définition 1** (Graphe de calcul). Un graphe de calcul est un graphe représentant une fonction mathématique. Un nœud de ce graphe représente soit une variable, des poids ou une opération mathématique. Formellement, nous noterons  $\mathcal{G}(V, A, V_{var}, (V_{op}, f), (V_{poids}, \theta))$  le graphe  $\mathcal{G}$  dont les nœuds sont  $V$ , les arêtes  $A$ ,  $V_{var} \subset V$  les nœuds représentant une variable,  $V_{op} \subset V$  ceux représentant une opération et  $f$  les opérations en question, et  $V_{poids} \subset V$  ceux représentant des poids et  $\theta$  les poids associés.

Dans cette définition, nous considérons les variables comme les arguments du graphe de calculs. En ce qui concerne, les réseaux de neurones, les entrées seront un nœud représentant une variable. Les poids sont les paramètres de la fonction. Ce seront les poids des neurones par exemple. Nous n'imposons pour l'instant aucune limite aux opérations mathématiques pouvant être effectuées au sein des graphes de calculs.

Dans le graphe de calculs de la figure 3.1, le nœud  $x$  est une variable, les nœuds  $W_1$  et  $W_2$  sont des poids et les deux nœuds  $\sigma$  représente l'opération sigmoïde appliquée terme à terme. Nous remarquerons que la sortie  $y$  n'est pas explicitée par un nœud.

Les graphes de calculs offrent une très grande liberté. Il est possible de représenter avec ceux-ci une classe de fonctions bien plus grandes qu'avec des réseaux de neurones classiques.

Écrivons dès maintenant, l'algorithme de propagation. Afin de pouvoir être concis, nous devons ajouter des notations. Nous noterons l'arc allant de la sortie  $k$  du nœud  $n_i$  vers l'entrée  $l$  du nœud  $n_j$  comme un quadruplet  $(n_i, k, n_j, l)$ . De plus, nous noterons  $in(n)$  le nombre d'entrées du nœud  $n$  et  $out(n)$  le nombre de sorties du nœud  $n$ .

Nous remarquons que l'algorithme de propagation est très similaire à celui décrit pour les réseaux de neurones. En effet, les graphes de calculs en sont une généralisation.

Nous avons ainsi modéliser nos réseaux de manière plus efficace. Il reste cependant un problème à résoudre. Notre objectif est d'optimiser les paramètres de notre fonction. Pour cela, nous avons besoin de calculer la dérivée du coût  $E$  par rapport aux poids avant de pouvoir utiliser l'algorithme de la descente du gradient.

## 3.3 Dérivation automatique

Afin de réaliser cela, nos nœuds représentant des opérations mathématiques ne seront pas seulement responsable de calculer une sortie mais aussi de propager le gradient. En effet prenons un nœud représentant une opération mathématique  $f$  à  $m$  entrées  $x_1, \dots, x_m$  et à  $n$  sorties  $y_1, \dots, y_n$ . La règle de la chaîne, nous permet d'exprimer la dérivée du coût par rapport aux entrées en fonction du coût par

---

**Algorithm 1** Algorithme d'évaluation d'un graphe de calculs.

---

```

procedure evaluer_graphe( $\mathcal{G}(V, A, V_{var}, (V_{op}, f), (V_{poids}, \theta)), x$ )
  function evaluer_noeud( $n_j$ )
    if déjàCalculé[ $j$ ] est faux then
      // 1. On récupère les valeurs des entrées.
       $t \leftarrow (evaluer\_noeud(i)_k, (n_i, k, l)$  tel que  $(n_i, k, n_j, l) \in A$ )
      // 2. On calcule les valeurs de sortie.
       $y_j \leftarrow f_j(t)$ 
      // 3. On mémorise.
      déjàCalculé[ $j$ ]  $\leftarrow$  vrai
    end if
    return  $y_j$ 
  end function
  Initialiser un tableau déjàCalculé de longueur  $|V|$  à faux.
  for  $n_i \in V_{var}$  do
     $y_i \leftarrow x_i$ 
    déjàCalculé[ $i$ ]  $\leftarrow$  vrai
  end for
  for  $n_j \in V$  do
    evaluer_noeud( $n_j$ )
  end for
end procedure

```

---

rapport aux sorties du nœud. En effet, on a :

$$\forall i \in \{1, \dots, m\}, \frac{\partial E}{\partial x_i} = \sum_{j=1}^n \frac{\partial y_j}{\partial x_i} \frac{\partial E}{\partial y_j} = \sum_{j=1}^n \frac{\partial f_j}{\partial x_i}(x) \frac{\partial E}{\partial y_j} \quad (3.2)$$

Rappelons que si  $x_i \in \mathbb{R}^p$  et  $y_j \in \mathbb{R}^q$  alors  $\frac{\partial E}{\partial y_j} \in \mathbb{R}^q$  et  $\frac{\partial y_j}{\partial x_i} = (\frac{\partial y_{j,l}}{\partial x_{i,k}})_{k,l} \in \mathbb{R}^{p \times q}$ .

La formule 3.2 nous permet de décrire un algorithme de rétropropagation du gradient très similaire à celui des réseaux de neurones :

## 3.4 Nœuds

Maintenant que les graphes de calculs ont été présentés de manière théorique, nous allons dans la suite présenter de manière plus pratique comment nous les avons implémentés.

### 3.4.1 La classe Node

Un nœud aura deux missions : calculer sa sortie et rétropropager le gradient. Nous représentons ces deux opérations sur la figure 3.2.

---

**Algorithm 2** Algorithme de rétropropagation du gradient dans un graphe de calculs.

---

```

procedure retropropager_gradient( $\mathcal{G}(V, A, V_{var}, (V_{op}, f), (V_{poids}, \theta)), x, y)$ 
  function calculer_gradient( $n_i$ )
    if déjàCalculé[ $i$ ] est faux then
      // 1. Pour chaque sortie, on récupère la dérivée du coût par rapport
      // à celle-ci.
      for  $k$  dans  $\{1, \dots, out(n_i)\}$  do
         $\frac{\partial E}{\partial y_{i,k}} \leftarrow \sum_{(n_j, l) \text{ tel que } (n_i, k, n_j, l) \in A} \text{calculer\_gradient}(n_j)_l$ 
      end for
      // 2. Pour chaque entrée, on calcule la dérivée du coût par rapport
      // à celle-ci.
      for  $l$  dans  $\{1, \dots, in(n_i)\}$  do
         $\frac{\partial E}{\partial x_{i,l}} \leftarrow \sum_{k \in \{1, \dots, out(n_i)\}} \frac{\partial f_{i,k}}{\partial x_{i,l}}(x) \frac{\partial E}{\partial y_{i,k}}$ 
      end for
      // 3. On mémorise.
      déjàCalculé[ $i$ ]  $\leftarrow$  vrai
    end if
    return  $\frac{\partial E}{\partial x_j}$ 
  end function

  Appeler evaluer_graphe( $\mathcal{N}(V, A, V_{in}, V_{out}, f), x$ ) et récupérer les entrées
  et sorties de chaque noeud
  Initialiser un tableau déjàCalculé de longueur  $|V|$  à faux.
  for  $n_i \in V_{out}$  do
    Calculer  $\frac{\partial E}{\partial y_i}$ 
    déjàCalculé $_i \leftarrow$  vrai
  end for
  for  $n_i \in V$  do
    calculer_gradient( $n_i$ )
  end for
end procedure

```

---

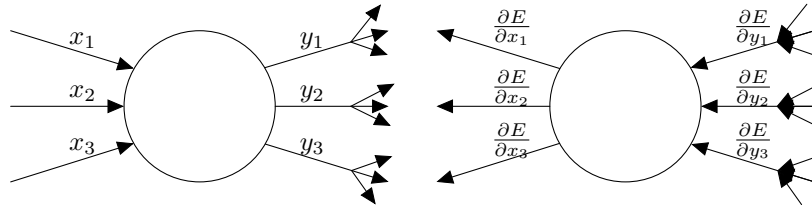


FIGURE 3.2 – Représentation de la propagation (à gauche) et de la rétropropagation (à droite) au sein d'un nœud.

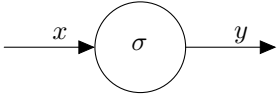
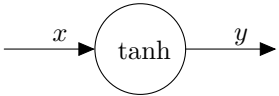


Lors de la propagation, un nœud demande ses entrées à ses parents puis calcule sa sortie et la mémorise. De façon analogue, lors de la rétropropagation, un nœud demande à chacun de ses enfants le gradient par rapport à la sortie qu'ils utilisent puis s'en sert pour calculer le gradient par rapport à ses entrées et mémorise .

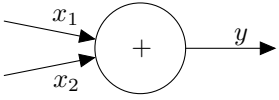
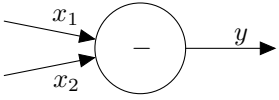
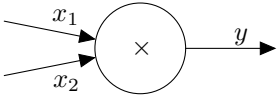
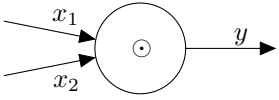
### 3.4.2 Les opérations mathématiques

Dans cette sous-section, nous allons décrire les symboles ainsi que les formules des nœuds que nous avons implémentés.

Nous commençons d'abord par décrire les nœuds à une entrée et une sortie :

Symbole	Formule de propagation	Formule de rétropropagation
	$\sigma(x) = \frac{1}{1+\exp(-x)}$	$\frac{\partial E}{\partial y} \odot y \odot (1 - y)$
	$\tanh(x)$	$\frac{\partial E}{\partial y} \odot (1 - y \odot y)$

Puis ceux à deux entrées et une sortie :

Symbole	Formule de propagation	Formule de rétropropagation
	$x_1 + x_2$	$(\frac{\partial E}{\partial y}, \frac{\partial E}{\partial y})$
	$x_1 - x_2$	$(\frac{\partial E}{\partial y}, -\frac{\partial E}{\partial y})$
	$x_1 \times x_2$	$(\frac{\partial E}{\partial y} x_2^T, x_1^T \frac{\partial E}{\partial y})$
	$x_1 \odot x_2$	$(\frac{\partial E}{\partial y} \odot x_2, \frac{\partial E}{\partial y} \odot x_1)$

### 3.4.3 Dérivation matricielle

Afin d'avoir des performances correctes en utilisant Python, nous avons comme contrainte d'optimiser notre utilisation de la bibliothèque `numpy`. Pour cela, il faut privilégier les appels à cette librairie qui exécute du code compilé plutôt que d'écrire nous-même les calculs en Python qui seraient interprétés et donc exécutés bien plus lentement.

Or afin d'utiliser la descente du gradient, il faut calculer la dérivée du coût qui est scalaire par rapport aux poids qui sont stockés dans une matrice. Le moyen le plus efficace de calculer cette quantité est d'obtenir une formule faisant intervenir directement des matrices.

Au début du projet, nous n'étions pas très à l'aise avec les dérivées matricielles. De plus, nous n'avons pas trouvé dans la littérature de document expliquant clairement le concept et donnant un formulaire utile pour les réseaux de neurones. Dans cette sous-section nous allons donc rappeler le concept et les notations. Puis nous démontrerons les formules énoncées ci-dessus pour expliciter la méthode permettant de d'obtenir de telles formules.

**Définition 2** (Dérivée tensorielle). Soit une fonction  $f : \mathbb{R}^{m_1 \times \dots \times m_p} \rightarrow \mathbb{R}^{n_1 \times \dots \times n_q}$ .

Nous appellerons dérivée tensorielle de  $f$  en  $x$  notée  $\frac{\partial f}{\partial x}(x)$ , le tenseur  $(\frac{\partial f_{j_1, \dots, j_q}}{\partial x_{i_1, \dots, i_p}}(x))_{i_1, \dots, i_p, j_1, \dots, j_q} \in$

$$\mathbb{R}^{m_1 \times \dots \times m_p \times n_1 \times \dots \times n_q}$$

Dans notre cas, nous calculerons toujours les dérivées d'une fonction à valeur dans  $\mathbb{R}$ , le coût et à arguments matriciels, les poids. Nous utiliserons donc le cas particulier suivant.

**Définition 3** (Dérivée matricielle). Soit une fonction  $f : \mathbb{R}^{m \times n} \rightarrow \mathbb{R}$ . Nous appellerons dérivée matricielle de  $f$  en  $x$  notée  $\frac{\partial f}{\partial x}(x)$ , la matrice  $(\frac{\partial f}{\partial x_{i,j}}(x))_{i,j} \in \mathbb{R}^{m \times n}$ .

*Remarque 1.* Dans la suite du document, si  $y = f(x)$ , nous noterons  $\frac{\partial f}{\partial x}(x)$ , la dérivée de  $f$  par rapport à  $x$  évaluée en  $x$ ,  $\frac{\partial y}{\partial x}$ .

*Remarque 2.* Nous avons choisi la convention  $\frac{\partial f}{\partial x}(x) = (\frac{\partial f}{\partial x_{i,j}}(x))_{i,j}$  cependant il est possible que d'autres auteurs utilisent la convention  $\frac{\partial f}{\partial x}(x) = (\frac{\partial f}{\partial x_{i,j}}(x))_{j,i}$ . Les deux conventions ont des avantages et des inconvénients, aucune ne fait a priori consensus. Nous avons choisi la première car la dérivée a la même taille que l'argument. Par conséquent pour la descente du gradient, nous pouvons simplement écrire :

$$w(t+1) = w(t) - \eta \frac{\partial E}{\partial w}$$

ce qui est fort élégant.

Nous donnerons tous les résultats en suivant notre convention. Cependant, ce n'est pas une fatalité puisqu'il suffit de transposer les résultats pour les obtenir selon l'autre convention.

Donnons dès à présent, le résultat qui va nous être le plus utile, la règle de la chaîne en version matricielle.

**Proposition 1** (Règle de la chaîne). Soit  $f : \mathbb{R}^{m \times n} \rightarrow \mathbb{R}^{p \times q}$ ,  $g : \mathbb{R}^{p \times q} \rightarrow \mathbb{R}$ , et  $x \in \mathbb{R}^{m \times n}$ . Notons  $y = f(x)$  et  $z = g(y)$  pour simplifier, on a alors :

$$\forall i \in \{1, \dots, m\}, j \in \{1, \dots, n\}, (\frac{\partial z}{\partial x})_{i,j} = \sum_{k \in \{1, \dots, p\}, l \in \{1, \dots, q\}} \frac{\partial z}{\partial y_{k,l}} \frac{\partial y_{k,l}}{\partial x_{i,j}}$$

Nous remarquons donc que dans le cas général, il n'y a pas de formule simple permettant de calculer la dérivée. Cependant pour les opérations que nous utilisons, cette formule se simplifie.

Nous allons dans la suite démontrer toutes les formules énoncées dans la sous-section 3.4.2. Cette partie pourra servir d'exercices pour le lecteur voulant s'initier à la dérivation matricielle ou de mémo pour ses rédacteurs.

Afin de rester dans le cadre de l'apprentissage automatique, nous appellerons comme dans les sections précédentes  $E$  notre fonction à valeurs dans  $\mathbb{R}$ .

La démarche pour dériver chaque formule sera toujours la même :

1. On exprime  $\frac{\partial E}{\partial x_{i,j}}$  en fonction des  $(\frac{\partial E}{\partial y_{k,l}})_{k,l}$  en utilisant la règle de la chaîne énoncée ci-dessus. On ne manipule ici que des dérivées scalaires, on peut donc utiliser toutes les opérations habituelles sans prendre de précautions.

2. On reconnait dans l'expression obtenue de  $\frac{\partial E}{\partial x_{i,j}}$  une opération connue qu'il est possible d'exprimer en notation matricielle.

**Proposition 2.** Si  $y = \sigma(x)$  alors  $\frac{\partial E}{\partial x} = \frac{\partial E}{\partial y} \odot y \odot (1 - y)$ .

Afin de démontrer ce résultat, nous utiliserons le lemme suivant donnant le résultat dans le cas scalaire.

**Lemme 1.** Soit  $\sigma : x \in \mathbb{R} \mapsto \frac{1}{1+\exp(-x)}$ , on a :

$$\forall x \in \mathbb{R}, \sigma'(x) = \sigma(x)(1 - \sigma(x))$$

*Démonstration.* Soit  $x \in \mathbb{R}$ , en utilisant les règles de dérivation, on a :

$$\sigma'(x) = \frac{\exp(-x)}{(1 + \exp(-x))^2} = \frac{1}{1 + \exp(-x)} \frac{\exp(-x)}{1 + \exp(-x)}$$

d'où :

$$\sigma'(x) = \sigma(x) \frac{(1 + \exp(x)) - 1}{1 + \exp(-x)} = \sigma(x)(1 - \sigma(x))$$

□

Nous pouvons alors démontrer la proposition 2.

*Démonstration.* L'opération  $\sigma$  étant appliquée terme à terme, on a que :

$$\forall i \in \{1, \dots, m\}, j \in \{1, \dots, n\}, \frac{\partial E}{\partial x_{i,j}} = \frac{\partial E}{\partial y_{i,j}} \frac{\partial y_{i,j}}{\partial x_{i,j}}$$

Or d'après le lemme 1,  $\frac{\partial y_{i,j}}{\partial x_{i,j}} = y_{i,j}(1 - y_{i,j})$ . Donc :

$$\forall i \in \{1, \dots, m\}, j \in \{1, \dots, n\}, \frac{\partial E}{\partial x_{i,j}} = \frac{\partial E}{\partial y_{i,j}} y_{i,j}(1 - y_{i,j})$$

On reconnait alors des produits terme à terme et on en conclut que :

$$\frac{\partial E}{\partial x} = \frac{\partial E}{\partial y} \odot y \odot (1 - y)$$

□

**Proposition 3.** Si  $y = \tanh(x)$  alors  $\frac{\partial E}{\partial x} = \frac{\partial E}{\partial y} \odot (1 - y \odot y)$ .

Rappelons sans démonstration, le résultat bien connu sur la dérivée de  $\tanh$ .

**Lemme 2.**

$$\forall x \in \mathbb{R}, \tanh'(x) = 1 - \tanh(x)^2$$

À l'aide de ce résultat, prouvons la proposition 3.

*Démonstration.* De même que pour  $\sigma$ , l'opération  $\tanh$  étant appliquée terme à terme, on a que :

$$\forall i \in \{1, \dots, m\}, j \in \{1, \dots, n\}, \frac{\partial E}{\partial x_{i,j}} = \frac{\partial E}{\partial y_{i,j}} \frac{\partial y_{i,j}}{\partial x_{i,j}}$$

Or d'après le lemme 2,  $\frac{\partial y_{i,j}}{\partial x_{i,j}} = 1 - y_{i,j}^2$ . Donc :

$$\forall i \in \{1, \dots, m\}, j \in \{1, \dots, n\}, \frac{\partial E}{\partial x_{i,j}} = \frac{\partial E}{\partial y_{i,j}} (1 - y_{i,j}^2)$$

On reconnaît alors, ici encore, des produits terme à terme et on en conclut que :

$$\frac{\partial E}{\partial x} = \frac{\partial E}{\partial y} \odot (1 - y \odot y)$$

□

**Proposition 4.** Si  $y = x_1 + x_2$  alors  $\frac{\partial E}{\partial x_1} = \frac{\partial E}{\partial x_2} = \frac{\partial E}{\partial y}$ .

*Démonstration.* Comme l'addition est faite terme à terme, pour tout  $i \in \{1, \dots, m\}, j \in \{1, \dots, n\}$ ,  $y_{i,j} = x_{1,i,j} + x_{2,i,j}$  ne dépend que de  $x_{1,i,j}$  et de  $x_{2,i,j}$  et donc :

$$\frac{\partial E}{\partial x_{1,i,j}} = \frac{\partial E}{\partial y_{i,j}} \frac{\partial y_{i,j}}{\partial x_{1,i,j}} = \frac{\partial E}{\partial y_{i,j}}$$

On procède de même pour  $\frac{\partial E}{\partial x_2}$ .

□

**Proposition 5.** Si  $y = x_1 - x_2$  alors  $\frac{\partial E}{\partial x_1} = \frac{\partial E}{\partial y}$  et  $\frac{\partial E}{\partial x_2} = -\frac{\partial E}{\partial y}$ .

*Démonstration.* Même preuve que pour l'addition. La seule différence est que pour tout  $i \in \{1, \dots, m\}, j \in \{1, \dots, n\}$ ,  $\frac{\partial y_{i,j}}{\partial x_{2,i,j}} = -1$ . □

**Proposition 6.** Si  $y = x_1 \times x_2$  alors  $\frac{\partial E}{\partial x_1} = \frac{\partial E}{\partial y} x_2^T$  et  $\frac{\partial E}{\partial x_2} = x_1^T \frac{\partial E}{\partial y}$ .

*Démonstration.* On a pour tout  $i \in \{1, \dots, m\}, k \in \{1, \dots, p\}, y_{i,k} = \sum_{j=1}^n x_{1,i,j} x_{2,j,k}$ .

Donc si  $i \in \{1, \dots, m\}, j \in \{1, \dots, n\}$ ,  $x_{1,i,j}$  intervient seulement dans le calcul de  $y_{i,k}$  pour  $k \in \{1, \dots, p\}$ . On en déduit que :

$$\frac{\partial E}{\partial x_{1,i,j}} = \sum_{k=1}^p \frac{\partial E}{\partial y_{i,k}} \frac{\partial y_{i,k}}{\partial x_{1,i,j}} = \sum_{k=1}^p \frac{\partial E}{\partial y_{i,k}} x_{2,j,k}$$

On reconnaît la l'expression du produit matriciel entre  $\frac{\partial E}{\partial y}$  et  $x_2^T$ , d'où :

$$\frac{\partial E}{\partial x_1} = \frac{\partial E}{\partial y} x_2^T$$

Procédons de même pour  $\frac{\partial E}{\partial x_2}$ . Soit  $j \in \{1, \dots, n\}, k \in \{1, \dots, p\}, x_{2_{j,k}}$  intervient seulement dans le calcul de  $y_{i,k}$  pour  $i \in \{1, \dots, m\}$ . On en déduit que :

$$\frac{\partial E}{\partial x_{2_{j,k}}} = \sum_{i=1}^m \frac{\partial E}{\partial y_{i,k}} \frac{\partial y_{i,k}}{\partial x_{2_{j,k}}} = \sum_{i=1}^m \frac{\partial E}{\partial y_{i,k}} x_{1_{i,j}}$$

On reconnaît la l'expression du produit matriciel entre  $x_1^T$  et  $\frac{\partial E}{\partial y}$ , d'où :

$$\frac{\partial E}{\partial x_2} = x_1^T \frac{\partial E}{\partial y}$$

□

**Proposition 7.** Si  $y = x_1 \odot x_2$  alors  $\frac{\partial E}{\partial x_1} = \frac{\partial E}{\partial y} \odot x_2$  et  $\frac{\partial E}{\partial x_2} = \frac{\partial E}{\partial y} \odot x_1$ .

*Démonstration.* Comme la multiplication est faite terme à terme, pour tout  $i \in \{1, \dots, m\}, j \in \{1, \dots, n\}, y_{i,j} = x_{1_{i,j}} x_{2_{i,j}}$  ne dépend que de  $x_{1_{i,j}}$  et de  $x_{2_{i,j}}$  et donc :

$$\frac{\partial E}{\partial x_{1_{i,j}}} = \frac{\partial E}{\partial y_{i,j}} \frac{\partial y_{i,j}}{\partial x_{1_{i,j}}} = \frac{\partial E}{\partial y_{i,j}} x_{2_{i,j}}$$

On reconnaît la formule du produit terme à terme entre  $\frac{\partial E}{\partial y}$  et  $x_2$ .

On procède de même pour  $\frac{\partial E}{\partial x_2}$ .

□

## 3.5 La classe Graph

## 3.6 Diagramme UML

Afin de résumer, les propos de cette partie, nous présentons le diagramme de classe de notre implémentation en Python.

## 3.7 Conclusion

Nous avons décrit les graphes de calculs, donnés les algorithmes permettant de calculer efficacement la sortie de celui-ci et de calculer la dérivée du coût par rapport aux paramètres. Ensuite, nous sommes rentrés davantage dans les détails en donnant les formules de propagation et de rétropropagation pour les nœuds les plus utiles dans le cadre de l'apprentissage automatique. Finalement nous avons décrit notre implémentation en donnant le diagramme de classes.

Dans cette partie, nous avons développé un outil bien plus puissant que les réseaux de neurones. En effet, nous avons donné un cadre et une méthode permettant d'optimiser théoriquement n'importe quelle fonction pour peu que nous puissions la décomposer en opérations dont nous sachons calculer la dérivée. Cette grande puissance théorique vient de plus avec une grande facilité d'implémentation et d'utilisation.

Finalement, nous verrons dans la partie suivante que notre implémentation graphes de calculs est bien plus performante que notre précédente modélisation en neurones. En outre, la modularité des graphes de calculs, nous permettra d'exécuter de nombreux tests sur l'architecture et les paramètres des réseaux de neurones, ce que nous détaillerons dans la partie suivante.

## 3.8 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.

## 3.9 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 1.4. Ils vont être rapelés et détaillés dans cette partie.

## 3.10 Étude de MNIST

### 3.10.1 Architecture du réseau

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



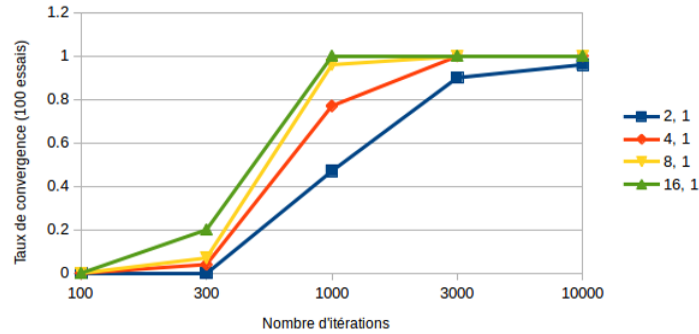


FIGURE 3.3 – Convergence en fonction de l'architecture avec tanh

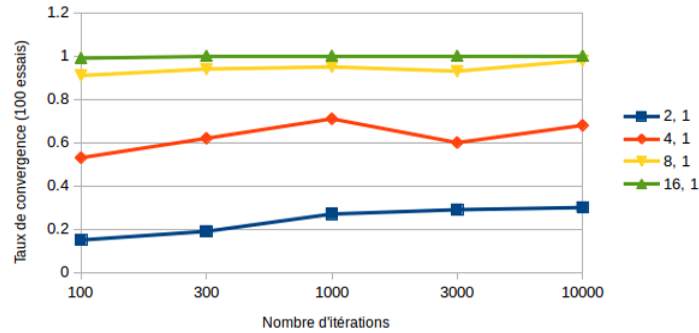


FIGURE 3.4 – Convergence en fonction de l'architecture avec relu

### 3.10.2 Prétraitement des données

### 3.10.3 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

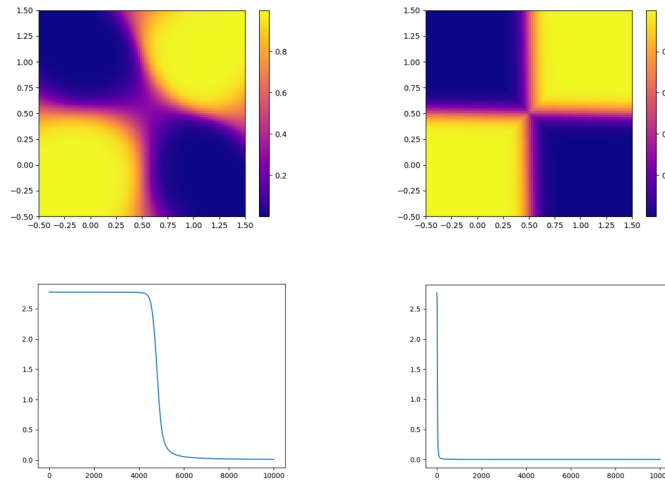


FIGURE 3.5 – Résultat après centrage des entrées pour tanh (à gauche) et pour relu (à droite)

lors de l'apprentissage vers de petites valeurs, une initialisation avec de petites amplitudes va permettre une convergence plus rapide.

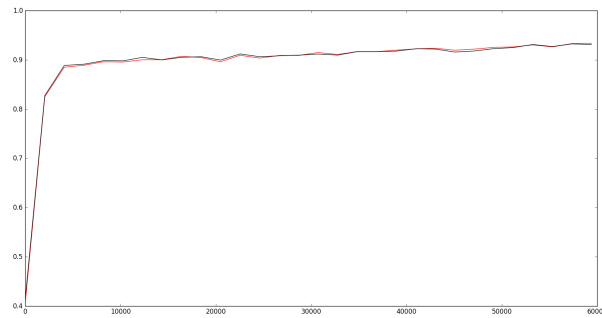


FIGURE 3.6 – Précision sur les ensembles de test (en noir) et d'apprentissage (en rouge) en fonction du nombre d'exemples utilisés pour l'apprentissage pour une initialisation des poids d'amplitude 0.1

C'est ce que l'on peut voir sur les deux figures 3.6 et 3.7. 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

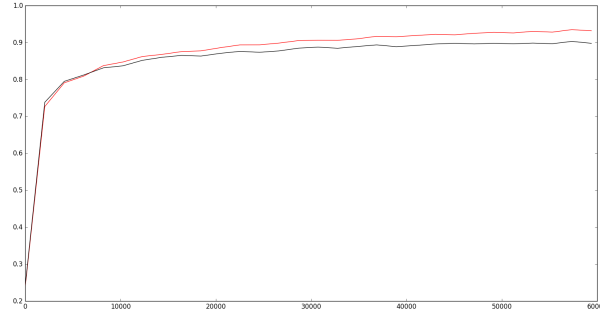


FIGURE 3.7 – Précision sur les ensembles de test (en noir) et d'apprentissage (en rouge) en fonction du nombre d'exemples utilisés pour l'apprentissage pour une initialisation des poids d'amplitude 0.1

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.

#### 3.10.4 Choix des fonctions d'activations

#### 3.10.5 Choix de la fonction de coût

#### 3.10.6 Taille des batchs et nombre de passages

#### 3.10.7 Taux d'apprentissage

Nous avons aussi étudié l'influence du taux d'apprentissage ou learning rate. En effet celui-ci influe sur la mise à jour des poids : plus il est grand et plus chaque l'importance donnée au calcul de la dérivée est important. Avoir un taux d'apprentissage grand permet d'obtenir rapidement des performances satisfaisantes en termes de précision. Au contraire, des phénomènes d'oscillations sont observés pour un apprentissage conséquent, et cela fournit en retour de moins bons résultats, par rapport à des taux d'apprentissage plus faibles. Cela a été illustré sur la figure 3.8.

L'architecture du réseau a été choisie sans couche cachée avec 10 neurones de sortie sous forme de Softmax. Le neurone de coût est un SoftmaxCrossEntropy, et l'apprentissage se fait avec des lots de 128 exemples. Nous avons alors étudié l'apprentissage du réseau pour des taux d'apprentissage allant de 0.01 à 100. Nous pouvons alors distinguer trois cas reflétant les principaux comportements :

- Pour un taux d'apprentissage de 0.01, la convergence est très lente mais sans oscillations. Si la précision obtenue peut-être à hauteur des autres taux d'apprentissage en utilisant plus d'exemples, on peut considérer celui-ci trop petit.

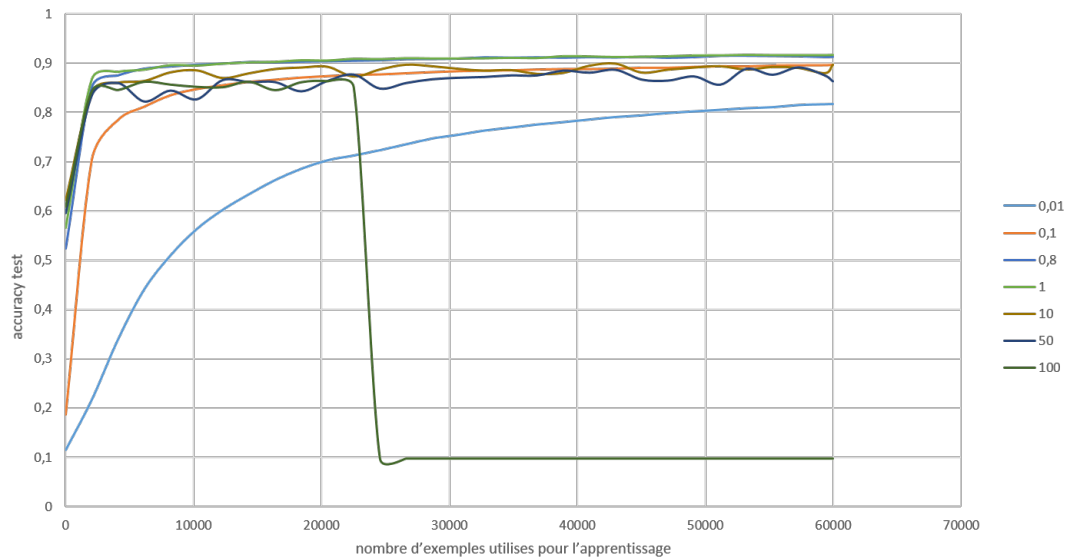


FIGURE 3.8 – Influence du learning rate sur la précision sur l'ensemble de test en fonction du nombre d'exemple utilisés

- Pour un taux d'apprentissage de 100, la convergence est très rapide au début, mais oscille beaucoup et va finalement bloquer l'apprentissage vers un autre minimum local de coût non désiré. On peut donc considérer ce taux d'apprentissage comme trop grand.
- Pour un taux d'apprentissage de 0.1, il semble apparaître un bon compromis entre stabilité et vitesse de la convergence. C'est cette valeur qui en général sera choisie pour les tests, bien que d'autres soient satisfaisants (0.8 ou 1 par exemple).

## Chapitre 4

# Réseaux de neurones récurrents

### 4.1 Motivation

Nous avons pu voir que les réseaux feedforward peuvent être utilisés afin d'apprendre 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 cycles. 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).

### 4.2 Dépliage

Nous avons dit précédemment qu'un réseau de neurones récurrent est un réseau possédant des cycles. Néanmoins, il est difficile d'imaginer que la valeur de sortie d'un neurone au temps  $t$  dépende de la sortie de ce même neurone au temps  $t$ . Pour résoudre ce problème, nous allons ajouter une dimension temporelle à nos réseaux et permettre aux neurones au temps  $t$  de dépendre des valeurs d'autres neurones au temps  $t - 1$ .

Afin de mettre en évidence cette dépendance temporelle sur nos graphes, les arêtes reliant la sortie d'un neurone au temps  $t - 1$  à un neurone au temps  $t$  vont être annotées par un carré comme sur la figure ?? . On les appellera arête "retard".

On a alors de nouvelles équations pour la sortie du neurone  $j$  :

$$\begin{cases} t = 0, & y_j(t) = 0 \\ \forall t \geq 1, & y_j(t) = g_j(\sum_{i \in Pa_t(j)} w_{ji} y_i(t) + \sum_{i \in Pa_{t-1}(j)} w'_{ji} y_i(t-1)) \end{cases} \quad (4.1)$$

où  $Pa_t(j)$  est l'ensemble des neurones ayant une arête allant vers  $j$  au temps  $t$  et  $Pa_{t-1}(j)$  l'ensemble des neurones ayant une arête "retard" allant vers  $j$ .

Une manière commode de mettre en évidence cette dépendance temporelle est de déplier le graphe.

On peut même pour une séquence d'entrée donnée modéliser tous les calculs du réseau sans aucun cycle si on déplie le réseau un nombre suffisant de fois.

### 4.3 RTRL

### 4.4 BPTT

## 4.5 Évaluation

### 4.5.1 Grammaire de Reber

Dans un premier temps, nous comparerons ces algorithmes sur l'apprentissage de la grammaire de Reber, qui est régulière. Nous utiliserons la grammaire simple présentée sur la figure 4.1.

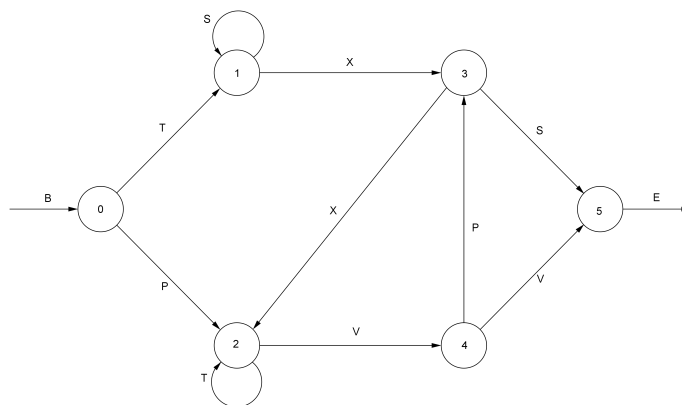


FIGURE 4.1 – Automate reconnaissant la grammaire de Reber simple

A l'aide de cette automate, nous générerons deux ensembles de mots respectivement 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 symétrique présentée sur la figure 4.2.

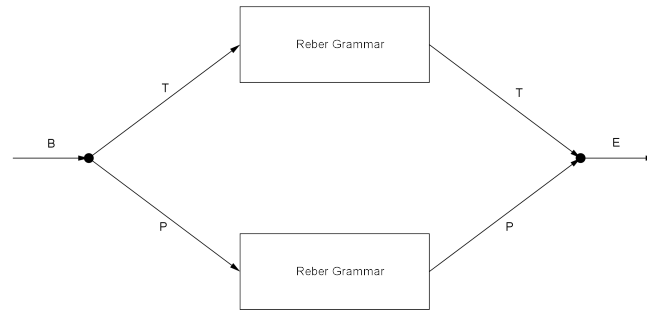


FIGURE 4.2 – Automate reconnaissant la grammaire de Reber symétrique

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.

#### 4.5.2 Shakespeare

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

### 4.6 Comparaison des deux algorithmes

## Chapitre 5

# Long Short-Term Memory (LSTM)

### 5.1 Motivation

Les réseaux de neurones récurrents étudiés jusqu'alors permettent effectivement d'apprendre des suites de séquences. Néanmoins, on observe un phénomène d'oubli se caractérisant par une faible influence des plus vieilles informations sur la sortie actuelle.

Cela est dû à la rétropropagation du gradient. En effet, celle-ci est basée sur la règle de la chaîne qui fait donc apparaître un produit de dérivées de fonctions d'activation. Or si ces dérivées sont supérieures à 1, cette partie du gradient rétropropagé risque d'exploser. À l'inverse, si elles sont inférieures à 1, comme c'est le cas pour la sigmoïde dont la dérivée a une valeur maximale de 0.25, cette partie du gradient tend à disparaître lorsque l'on remonte loin dans le temps. Ainsi, on perd la dépendance à long terme.

Nous travaillons jusque là avec des réseaux récurrents comme celui présenté sur la figure 5.1 composé d'une simple couche de tangentes hyperboliques.

Dans le cadre des LSTM, nous utilisons une nouvelle cellule de base afin d'éviter cette disparition du gradient lors de la rétropropagation.



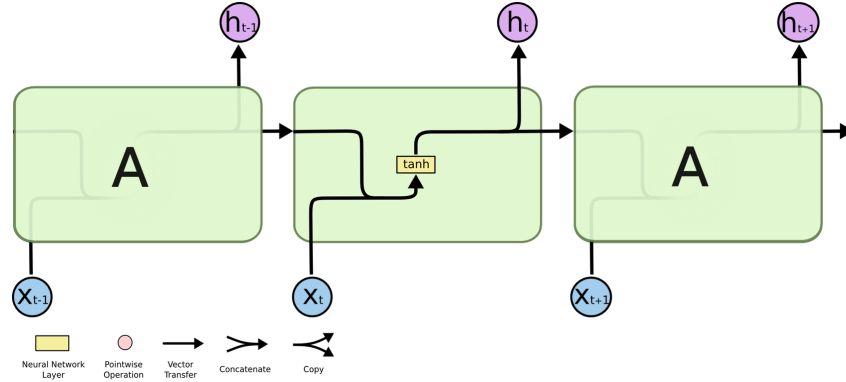


FIGURE 5.1 – Dépliage d'un RNN classique

## 5.2 Principe de fonctionnement

Comme pour les réseaux récurrents, on peut déplier les LSTM afin de se ramener à une cellule de base qui se répète. Le principe des LSTM repose sur l'existence d'un état qui apparaît tout en haut de la cellule et qui subit seulement quelques modifications linéaires. Cela permettra ainsi, lors de la rétro-propagation du gradient, de ne pas perdre la dépendance avec les informations lointaines.

La cellule de base des LSTM peut donc être représentée par la figure suivante.

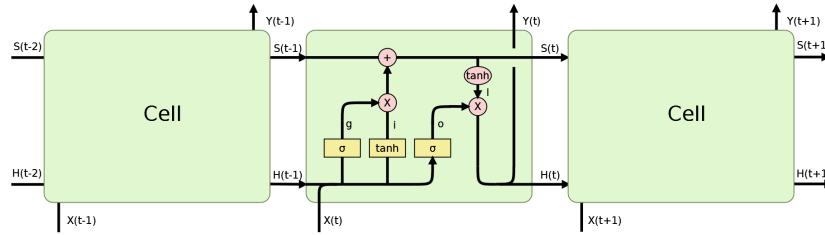


FIGURE 5.2 – Cellule élémentaire des LSTM

Cette figure permet de représenter facilement le réseau en utilisant la structure d'un graphe de calcul. Il faut cependant bien distinguer les fonctions tanh et sigmoïde qui symbolisent une couche entière de neurones (dans les rectangles jaunes) et la fonction tanh (dans le cercle rose) qui s'applique à chaque élément du vecteur en entrée. De même, les multiplications dans les cercles roses se font terme à terme.

Les poids à régler lors de l'apprentissage se situent au niveau de chaque couche de neurones sigmoïde et tanh. Il est à noter que lorsque l'on déplie la

cellule LSTM dans le temps, les poids sont les mêmes d'une cellule à l'autre. Le fait qu'ils soient ainsi partagés sera important pour l'implémentation.

On peut distinguer plusieurs parties dans la cellule LSTM qui possèdent chacune une fonction particulière.

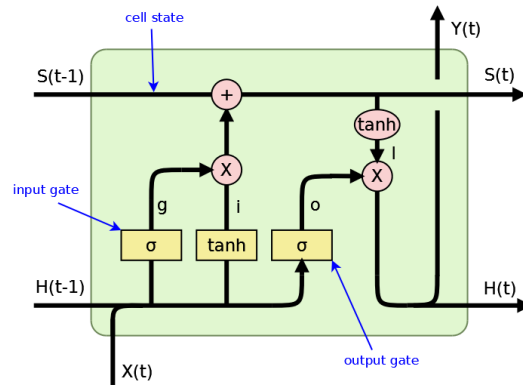


FIGURE 5.3 – Différentes parties d'une cellule LSTM

Le cell state  $S$  correspond à la mémoire de la cellule. Celle-ci subit peu de modification au cours du temps, correspondant à l'ajout d'information dans la mémoire.

Les sigmoïdes permettent d'obtenir des sorties comprises entre 0 et 1. On peut donc pondérer l'importance d'une valeur en la multipliant par la sortie d'une sigmoïde. Si la sortie de la sigmoïde est proche de 1, cela signifie que l'on garde la valeur, en revanche, si celle-ci est proche de 0, on oublie la valeur calculée.

Ainsi on calcule un vecteur  $i$  à partir de l'entrée et de la sortie précédente à l'aide d'une couche de  $\tanh$ . Puis, on le multiplie terme à terme par le vecteur de sortie  $g$  d'une couche de sigmoïdes appelée input gate afin de sélectionner les informations que l'on souhaite conserver. Enfin, on ajoute ces informations sélectionnées dans le cell state.

La sortie de la cellule est obtenue à partir du cell state auquel on applique une  $\tanh$ . Enfin, on sélectionne les informations que l'on veut garder en sortie à l'aide d'une couche de sigmoïdes appelée output gate appliquée à l'entrée.

En pratique, on n'est pas sûr qu'une cellule LSTM suive effectivement le principe décrit précédemment. Celui-ci est plutôt une illustration afin de comprendre de manière générale le fonctionnement des LSTM.

La structure d'une cellule LSTM n'est pas non figée. En effet, il est possible de l'adapter en ajoutant, enlevant certains éléments, gates. Par exemple, on peut ajouter une forget gate composée d'une couche de sigmoïdes au début de la cellule. On multiplie la sortie de cette forget gate afin de garder, supprimer, réduire certaines composantes du cell state.

Dans le cas d'une cellule LSTM, les équations de propagation sont simples à calculer. En effet, comme pour le graphe de calcul, les équations sont directement données par le schéma.

Pour la backpropagation du gradient, il est toutefois nécessaire de calculer les formules à utiliser dans l'algorithme. La figure 5.4 fait apparaître le gradient à chaque endroit de la cellule.

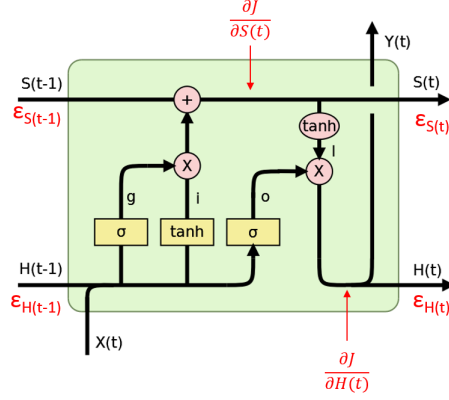


FIGURE 5.4 – Cellule LSTM

En utilisant, BPTT pour rétropropager le gradient sur le graphe du LSTM déplié, on obtient les formules suivantes :

$$\varepsilon_{S(t-1)} = \frac{\partial J}{\partial S(t)} = \varepsilon_{S(t)} + F'_l \text{diag}(o) \frac{\partial J}{\partial H(t)} \quad (5.1)$$

$$\varepsilon_{H(t-1)} = \left( (W_g^T F'_g \text{diag}(i) + W_i^T F'_i \text{diag}(g)) \frac{\partial J}{\partial S(t)} + W_o^T F'_o \text{diag}(l) \frac{\partial J}{\partial H(t)} \right)_{1, \dots, n} \quad (5.2)$$

avec :

$$F'_l = \text{diag}((1 - l_i^2)_{i=1, \dots, n})$$

$$F'_i = \text{diag}((1 - i_i^2)_{i=1, \dots, n})$$

$$F'_g = \text{diag}((g_i(1 - g_i))_{i=1, \dots, n})$$

$$F'_o = \text{diag}((o_i(1 - o_i))_{i=1, \dots, n})$$

Il est alors possible de calculer le gradient relatif à chaque matrice de poids.

$$\frac{\partial J}{\partial W_o} = \begin{pmatrix} H(t-1) \\ x(t) \end{pmatrix} \left( F'_o \text{diag}(l) \frac{\partial J}{\partial H(t)} \right)^T \quad (5.3)$$

$$\frac{\partial J}{\partial W_i} = \begin{pmatrix} H(t-1) \\ x(t) \end{pmatrix} \left( F'_i \text{diag}(g) \frac{\partial J}{\partial S(t)} \right)^T \quad (5.4)$$

$$\frac{\partial J}{\partial W_g} = \begin{pmatrix} H(t-1) \\ x(t) \end{pmatrix} \left( F'_g \text{diag}(i) \frac{\partial J}{\partial S(t)} \right)^T \quad (5.5)$$

On rétropropage le gradient dans tout le graphe déplié avec les formules précédentes. Celui-ci est à chaque fois accumulé au niveau des poids. Les poids  $W_o$ ,  $W_i$  et  $W_g$  sont alors mis à jour avec la formule :

$$W = W - \eta \sum \frac{\partial J}{\partial W} \quad (5.6)$$

### 5.3 Première Implémentation

Une première implémentation des LSTM a été réalisé en conservant le plus possible le concept de cellule indépendante. Ainsi une classe `Weights` contient tous les poids de la cellule et la classe `Lstmcell` se contentera elle de calculer la propagation et la rétropropagation (Voir Figure 4.4 de la cellule LSTM pour situer les poids).

### 5.4 Implémentation avec des nœuds

Cette seconde implémentation a pour but de réutiliser les composantes `Nodes` (utilisées dans les réseaux simples) afin d'optimiser les calculs et de faciliter la mise en pipeline de plusieurs cellules LSTM les unes derrières les autres.

### 5.5 Résultats

### 5.6 Autres applications

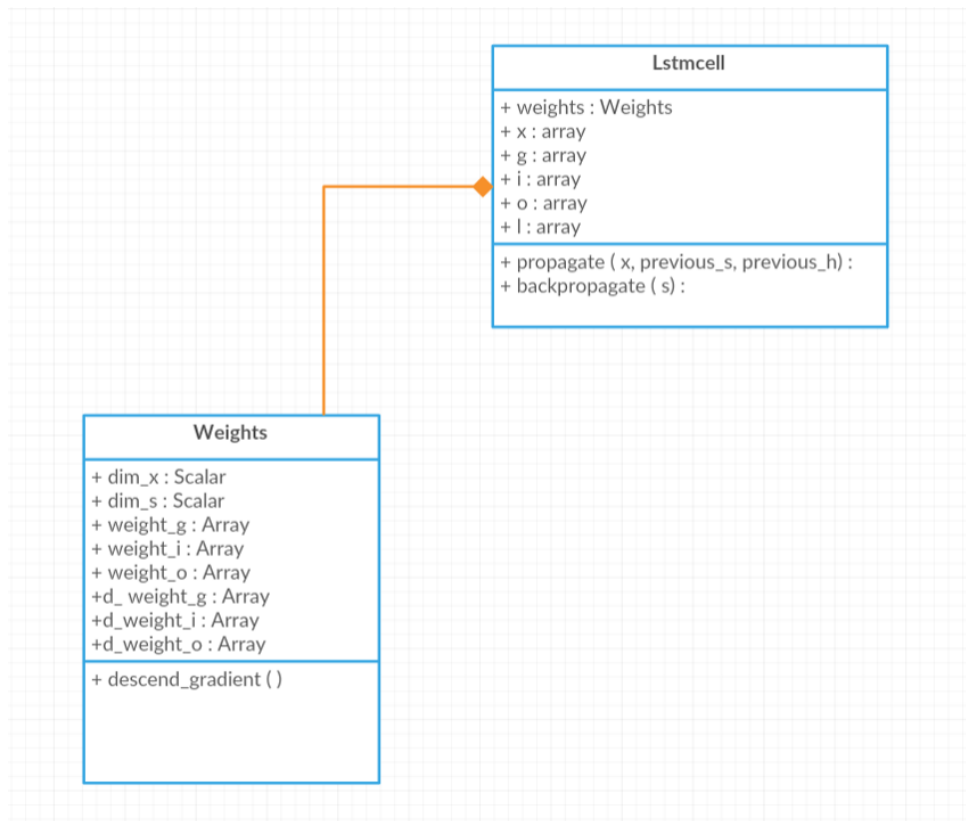


FIGURE 5.5 – Diagramme UML de la cellule LSTM

## Chapitre 6

# Application : génération de musique

### 6.1 Introduction

La deuxième application des LSTM à laquelle nous nous sommes intéressés consistait en la génération de musique. Pour cela, nous avons envisagé quatre approches. La première n'est pas directement liée à la musique. En effet, on utilise alors le format abc qui représente un morceau de musique par un texte. La génération se faisait alors simplement en appliquant ce que l'on avait fait avec Shakespeare. La deuxième méthode consiste à utiliser le format midi. Celui-ci permet de représenter simplement l'état des notes à chaque instant. La troisième méthode s'appuie sur l'utilisation de partitions qui permettent de retranscrire efficacement un morceau de musique. Dans ce cas-là, on génère directement une partition. Enfin, la dernière méthode consiste à exploiter directement un signal sonore. Cela permet d'utiliser tout type de son.

Quelle que soit la stratégie utilisée, il convient de trouver une base de données suffisamment fournie afin de pouvoir entraîner le réseau dessus. Il est assez difficile de trouver des fichiers abc sur différents styles de musique. En revanche il existe plusieurs bases de données réunissant de nombreux fichiers midi. Enfin, l'utilisation de signaux permet de s'affranchir de ces problèmes puisqu'il est alors possible d'utiliser n'importe quel fichier sonore (wav, mp3, ...). Des convertisseurs existent afin de passer d'un format à l'autre, mais ceux-ci sont plus ou moins efficaces.

## 6.2 Génération en abc

## 6.3 Génération en midi

### 6.3.1 Présentation du format

Le format midi utilise une représentation binaire. Un ensemble d'événements permet de décrire entièrement le morceau. On distingue les MetaMessages qui permettent de donner des informations d'ordre général sur la mélodie telles que le titre, la durée, le tempo, ... Il existe aussi les Messages qui décrivent la succession des événements tout au long du morceau, comme l'activation ou la désactivation de notes. Un morceau peut être décomposé en plusieurs tracks. Ceux-ci se partagent alors les différents voix : mélodie, accompagnement, ...

On utilisera la bibliothèque *mido* sous Python qui permet de représenter le format midi sous la forme d'une succession d'événements. Chaque message s'écrit alors de la manière suivante *événement paramètres time*. *time* permet de définir après quel délai par rapport à l'événement précédent l'événement courant a lieu. Dans notre situation, on s'intéressera à l'exploitation des événements *note\_on note velocity time* et *note\_off note velocity time*. Ceux-ci permettent d'activer ou de désactiver une note. On peut aussi préciser l'intensité de la note comprise entre 0 et 127. Dans notre cas, on considère que l'intensité vaut 0 lorsque l'on éteint la note. En réalité, elle permet de préciser de quelle manière on relâche la note (rapidement ou si on laisse un peu durer). Les notes sont elles aussi numérotées de 0 à 127. Le tableau 6.3.1 donne la correspondance entre les notes et leur numéro dans la représentation midi.

Octave Number	Hauteurs											
	C	C#	D	D#	E	F	F#	G	G#	A	A#	B
0	0	1	2	3	4	5	6	7	8	9	10	11
1	12	13	14	15	16	17	18	19	20	21	22	23
2	24	25	26	27	28	29	30	31	32	33	34	35
3	36	37	38	39	40	41	42	43	44	45	46	47
4	48	49	50	51	52	53	54	55	56	57	58	59
5	60	61	62	63	64	65	66	67	68	69	70	71
6	72	73	74	75	76	77	78	79	80	81	82	83
7	84	85	86	87	88	89	90	91	92	93	94	95
8	96	97	98	99	100	101	102	103	104	105	106	107
9	108	109	110	111	112	113	114	115	116	117	118	119
10	120	121	122	123	124	125	126	127				

tableCorrespondance entre les notes et leur représentation en midi

### 6.3.2 Principe

Comme dit précédemment, on s'intéresse simplement à l'activation ou non de notes. Pour cela, nous allons représenter un morceau de musique par une matrice. Celle-ci possède 128 lignes (une pour chaque note) et à un nombre de colonnes dépendant de la longueur du morceau. En midi, un morceau est décomposé en ticks d'horloge. L'état du morceau variant peu entre deux ticks, il convient de le rééchantillonner afin d'avoir une matrice exploitable. Chaque instant sera ainsi décrit par une colonne de la matrice. En regardant chaque ligne, on pourra savoir si une note est éteinte (0) ou si elle est activée, on a alors directement accès à sa vélocité (entre 1 et 127). Dans notre représentation, nous décrirons tous les tracks dans une seule matrice.

Nous testerons notre implémentation sur les jigs déjà utilisées en abc après les avoir converties en midi (en utilisant `abc2midi`) ainsi que des morceaux de Mozart au piano. Les figures 6.3.2 et 6.3.2 représentent les matrices d’une jig et d’un morceau de Mozart.

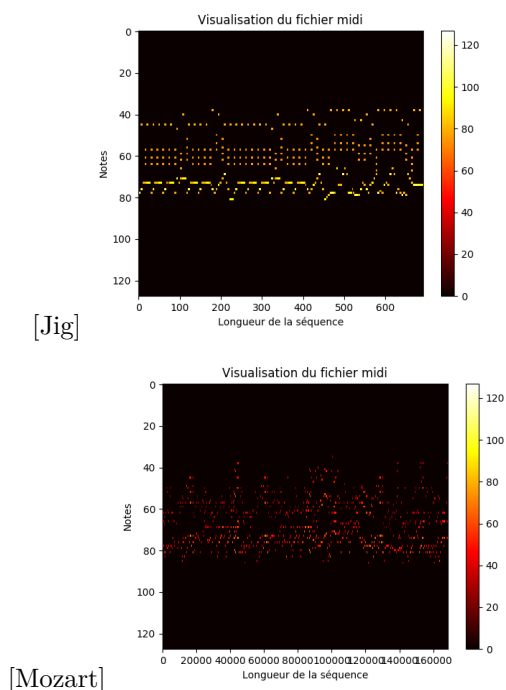


FIGURE 6.1 – Visualisation de fichiers midi

Lors de l’apprentissage, on envoie en entrée du réseau une colonne de la matrice. Celui-ci doit alors prévoir la prochaine. Lors de la génération, on donne un vecteur ou une séquence de notes en entrée du réseau et on lui demande de générer un morceau de musique. On définit un seuil (à régler, souvent entre 10 et 20) qui permet de décider si une note est active ou non. Cela permet de jouer plusieurs notes en même temps ainsi que des accords. Après la génération d’un vecteur, on ne le remet pas directement en entrée. On sélectionne les notes actives grâce au seuil et on met les autres à 0. On obtient finalement un ensemble de vecteurs décrivant les états successifs du morceau généré. On les concatène pour en faire une matrice. Il ne reste plus qu’à faire la conversion de cette matrice en midi.

### 6.3.3 Résultats

On applique dans un premier temps l’apprentissage sur les jigs. On utilise deux cellules LSTM en série avec un état caché de 512. On met une couche



de neurones pour adapter les données en sortie. Avec une activation linéaire on obtient ainsi des vecteurs de taille 128 en sortie où chaque élément représente l'intensité de la note correspondante. On lui donne en entrée des séquences de 10 vecteurs, cela signifie que l'on va déplier le réseau 10 fois à chaque fois. On fait des batchs de 50 séquences. On utilise RMSprop comme algorithme d'optimisation avec un learning rate de 0.95. La figure 6.2 permet de visualiser le fichier généré après le passage de 38 800 batchs lors de l'apprentissage.

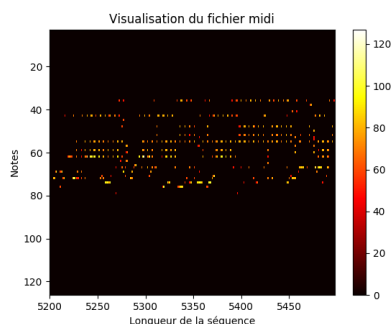


FIGURE 6.2 – Visualisation du fichier midi généré à partir des jigs après 38 800 batchs

Le résultat obtenu possède bien le style des jigs avec un accompagnement répétitif à l'arrière. Cet accompagnement, souvent commun aux jigs, est très rapidement appris par le réseau. Certaines séquences mélodiques rappellent énormément certaines jigs de l'apprentissage.

L'apprentissage de Mozart se fait sur plusieurs symphonies pour piano trouvées dans une base de données de midi. On utilise une fois de plus 2 cellules LSTM en série avec un état caché de 512. Cette fois on fait l'apprentissage sur des séquences de longueur 120. Une longueur plus grande est importante car les morceaux de Mozart sont moins répétitifs que les jigs. Cela permet au réseau de remonter plus loin dans les vecteurs passés et donc d'apprendre la structure de séquences plus longues. On passe des batchs comportant 10 séquences. On utilise là aussi l'algorithme RMSprop avec un learning rate de 0.95.

La difficulté principale lors de l'apprentissage de Mozart est que la structure commune des morceaux est moins évidente que dans les jigs. De plus, il y a souvent des changements de rythmes/tempo dans les fichiers midi, ce qui est difficile à retranscrire dans la représentation matricielle avec l'implémentation actuelle. La figure 6.3 représente le morceau généré après 79 500 passés.

Il est nécessaire de passer un grand nombre de batchs avant d'avoir des

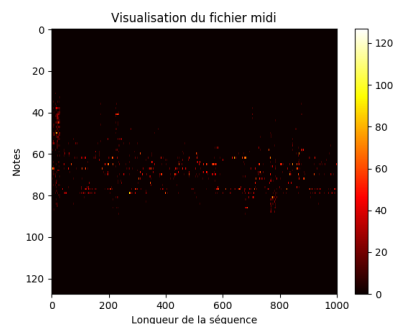


FIGURE 6.3 – Visualisation du fichier midi généré à partir des morceaux de Mozart après 79 500 batchs

résultats satisfaisants. Le morceau généré a une forme similaire aux morceaux du datasets. Néanmoins, l'écoute ne rend pas toujours aussi bien que les originaux. Un réseau plus complexe, un apprentissage plus long et une base de données plus importante pourraient permettre d'améliorer les résultats.

## 6.4 Génération de partitions

## 6.5 Génération de signaux