

Compte-rendu

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

13 juin 2017

Table des matières

1	Réseaux de neurones	4
1.1	Apprentissage automatique	4
1.2	Inspiration biologique	6
1.3	Un neurone	7
1.4	Un réseau de neurones	12
1.5	Propagation	14
1.6	Fonctions de coût	14
1.6.1	Régression	16
1.6.2	Classification binaire	16
1.6.3	Classification à plusieurs classes	17
1.7	Descente du gradient	18
1.8	Rétropropagation du gradient	18
2	Une première implémentation	20
2.1	Motivation	20
2.2	Diagramme UML	20
2.3	Principe de fonctionnement	22
2.4	Résultats	23
2.5	Conclusion	25
3	Graphes de calculs	27
3.1	Motivations	27
3.2	Définitions	29
3.3	Dérivation automatique	29
3.4	Nœuds	30
3.4.1	La classe Node	30
3.4.2	Les variables	32
3.4.3	Les poids	32
3.4.4	Les opérations mathématiques	32
3.4.5	Dérivation matricielle	35
3.5	La classe Graph	40
3.6	Diagramme UML	40
3.7	Conclusion	40

4	Études des paramètres	43
4.1	Approche du problème	43
4.2	Étude du XOR	44
4.3	Étude de MNIST	45
4.3.1	Introduction	45
4.3.2	Architecture du réseau	45
4.3.3	Initialisation des poids	46
4.3.4	Choix des fonctions d'activations et de coût	47
4.3.5	Taille des batchs et nombre de passages	48
4.3.6	Taux d'apprentissage	49
5	Réseaux de neurones récurrents	51
5.1	Motivation	51
5.2	Dépliage	51
5.3	BPTT	54
5.4	RTRL	54
5.5	Comparaison des deux algorithmes	56
5.6	Évaluation	58
5.6.1	Grammaire de Reber	58
5.6.2	Résultats	60
5.7	Problème des dépendances temporelles	60
6	Long Short-Term Memory (LSTM)	64
6.1	Motivation	64
6.2	Principe de fonctionnement	65
6.3	Première Implémentation	68
6.4	Implémentation avec des nœuds	68
6.5	Résultats	70
6.6	Autres applications	70
7	Application : génération de texte	71
7.1	Introduction	71
7.2	Principe	71
7.2.1	Architecture du réseau	71
7.2.2	Entraînement	72
7.2.3	Génération	72
7.3	Génération à partir d'œuvres de Shakespeare	73
7.3.1	Résultats	73
7.3.2	tests complémentaires sur la forget gate	74
7.4	Conclusion	75
8	Algorithmes d'optimisation	76
8.1	Motivation	76
8.2	Descente de gradient et momentum	76
8.3	Algorithmes à learning rate adaptatif	77
8.3.1	Adagrad	77

8.3.2	RMSProp	77
8.3.3	Adadelta	78
8.3.4	Adam	78
8.4	Comparaison des algorithmes	79
8.4.1	Application à l'apprentissage du XOR	79
8.4.2	Application à l'apprentissage des oeuvres de Shakespeare	80
8.5	Conclusion	80
9	Application : génération de musique	81
9.1	Introduction	81
9.2	Génération en abc	81
9.2.1	Présentation du format	81
9.2.2	Principe	82
9.2.3	Résultats	83
9.3	Génération en midi	84
9.3.1	Présentation du format	84
9.3.2	Principe	84
9.3.3	Résultats	85
9.4	Génération de partitions	87
9.4.1	Génération de notes	87
9.4.2	Génération de notes pour plusieurs instruments en série	88
9.4.3	Génération de notes pour plusieurs instruments en parallèle	88
9.5	Génération de signaux	88
9.5.1	Présentation du format	88
9.5.2	Principe	90
9.5.3	Résultat	91

Introduction

Nous sommes un groupe de six étudiants de CentraleSupélec, cursus Ingénieur Supélec. Dans le cadre d'un projet long, nous avons été amenés à étudier les réseaux de neurones. Ce projet était encadré par deux enseignants-chercheurs de notre école, Mme Joanna Tomasik et M. Arpad Rimmel. L'objectif était d'étudier et d'implémenter différents réseaux de neurones. Ainsi, le projet était décomposé en plusieurs étapes. La première consistait à étudier le fonctionnement d'un réseaux de neurones simples et à l'implémenter. Puis, nous nous sommes intéressés aux réseaux de neurones récurrents offrant de nouvelles opportunités dans l'apprentissage. Deux algorithmes d'apprentissage furent ainsi étudiés : RTRL et BPTT. Enfin, nous sommes entrés dans le vif du sujet en nous penchant sur l'étude des LSTM. La fin du projet était destinée à appliquer les LSTM sur différents problèmes. Tout au long du projet, la démarche fut de rechercher les travaux déjà existants afin de s'appropriier les différents aspects du problème et d'étudier les solutions proposées. Puis, nous implémentions nos propres réseaux de neurones sous Python afin de comparer leurs performances respectives.

Tout au long du projet, plusieurs outils furent mis en place afin d'organiser et de faciliter le travail de groupe. GitHub (<https://github.com/supelec-lstm>) a été utilisé afin de partager le code entre les différents membres. Zotero permettait de tenir à disposition une bibliographie de tous les articles trouvés sur le sujet. Enfin, une réunion hebdomadaire avec nos professeurs et l'autre équipe permettait de faire le point sur les avancées de la semaine, de clarifier certaines incertitudes et de discuter de problèmes d'implémentation.

Ce compte-rendu fait la synthèse du travail effectué lors de ce projet. On y trouvera des apports théoriques sur le fonctionnement des réseaux de neurones. De plus, les implémentations choisies seront explicitées et les différentes méthodes d'apprentissage seront comparées.

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 $(x^{(1)}, y^{(1)}), \dots, (x^{(N)}, y^{(N)})$. 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 $(x^{(i)})_{i \in \{1, \dots, N\}}$ sont les réalisations de variables aléatoires $(X_i)_{i \in \{1, \dots, N\}}$ avec les X_i indépendantes et identiquement distribuées selon une probabilité p . Pour plus de commodités, notons $X = \{x^{(i)}, \forall i \in \{1, \dots, N\}\}$ et $Y = \{y^{(i)}, \forall i \in \{1, \dots, N\}\}$.

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 $\mathcal{Y} = \{0, \dots, M-1\}$ 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.

Définition 1 (Erreur sur un exemple). Soit $e : \mathbb{R}^m \times \mathbb{R}^m \rightarrow \mathbb{R}$ la mesure de performance sur un exemple.

$e(y, \hat{f}(x))$ représentera l'écart de notre modèle avec la sortie attendue.

Exemple 1 (Erreur quadratique). 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 :

Définition 2 (Erreur empirique).

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

avec \hat{p}_{data} est la distribution empirique dans l'ensemble des exemples.

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

Définition 3 (Erreur absolue).

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

avec 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 est généralement impossible. Cependant, il est aisé de calculer E_{in} , qui est une estimation de E_{out} sur les données disponibles.

En effet, on a la proposition suivante :

Proposition 1. E_{in} est un estimateur non biaisé de E_{out} .

Démonstration.

$$\mathbb{E}_{(x_1, y_1), \dots, (x_N, y_N) \sim p} (E_{in}) = \frac{1}{N} \sum_{i=1}^N \mathbb{E}_{(x_i, y_i) \sim p} (e(y^{(i)}, f(x^{(i)}))) = \mathbb{E}_{x, y \sim p} (e(y, f(x)))$$

□

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 :

$$E_{train} = \frac{1}{N_{train}} \sum_{i=1}^{N_{train}} e(y_{train}^{(i)}, f(x_{train}^{(i)})) \quad (1.1)$$

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

$$E_{test} = \frac{1}{N_{test}} \sum_{i=1}^{N_{test}} e(y_{test}^{(i)}, f(x_{test}^{(i)})) \quad (1.2)$$

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

C'est à partir de l'hypothèse que le comportement intelligent humain est le résultat de la structure et des éléments de bases du système nerveux central (que sont les neurones), que l'on a développé les réseaux de neurones artificiels. A la suite des observations biologiques, les scientifiques ont schématisé un neurone en trois parties : les dendrites qui constituent les entrées de l'information créée par un stimulus, un corps où l'information est traitée, et l'axone qui représente la voie de sortie de l'information vers d'autres unités neuronales. Les synapses permettent le contact entre 2 neurones, chacun d'entre eux intégrant en permanence jusqu'à un millier de signaux synaptiques. Ces signaux n'opèrent pas de manière linéaire : il y a un effet de seuil. Les réseaux de neurones artificiels ont été développés à partir de cette vision et leur but est de modéliser le mécanisme d'apprentissage et de traitement de l'information qui se produit dans le cerveau humain.

1.3 Un neurone

Définition 4 (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 :

$$\forall x \in \mathbb{R}^n, y = f(x) = g(w^T x + b) = g\left(\sum_{i=1}^n w_i x_i + b\right) \quad (1.3)$$

Remarque 1. 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 1.3 devient :

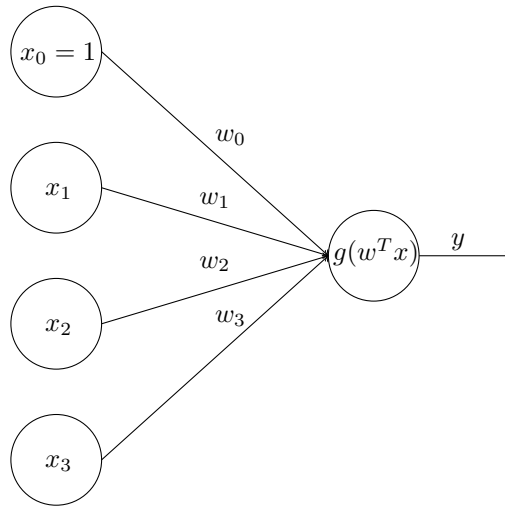


FIGURE 1.1 – Représentation graphique d’un neurone. Le biais est ici considéré comme le poids associé à une entrée constante égale à 1.

$$\forall x \in \mathbb{R}^n, y = f(x) = g(w^T x) = g\left(\sum_{i=0}^n w_i x_i\right) \quad (1.4)$$

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 1.2 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.

Exemple 2 (Perceptron). 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.

Exemple 3 (Régression logistique). 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 classe 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.

Exemple 4 (Régression linéaire). 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 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 1.5. En l'état, cet ensemble n'est pas linéairement séparable et donc un neurone simple ne peut pas le classifier correctement. Cependant si une transformation telle que $x \mapsto (x_1, x_2, x_1x_2, x_1^2, x_2^2)$ est appliquée aux exemples. Les points sont maintenant linéairement séparables dans ce nouvel espace. Il est alors possible d'utiliser un neurone afin de classifier correctement l'ensemble. Ces résultats sont présents sur la figure 1.5.

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.

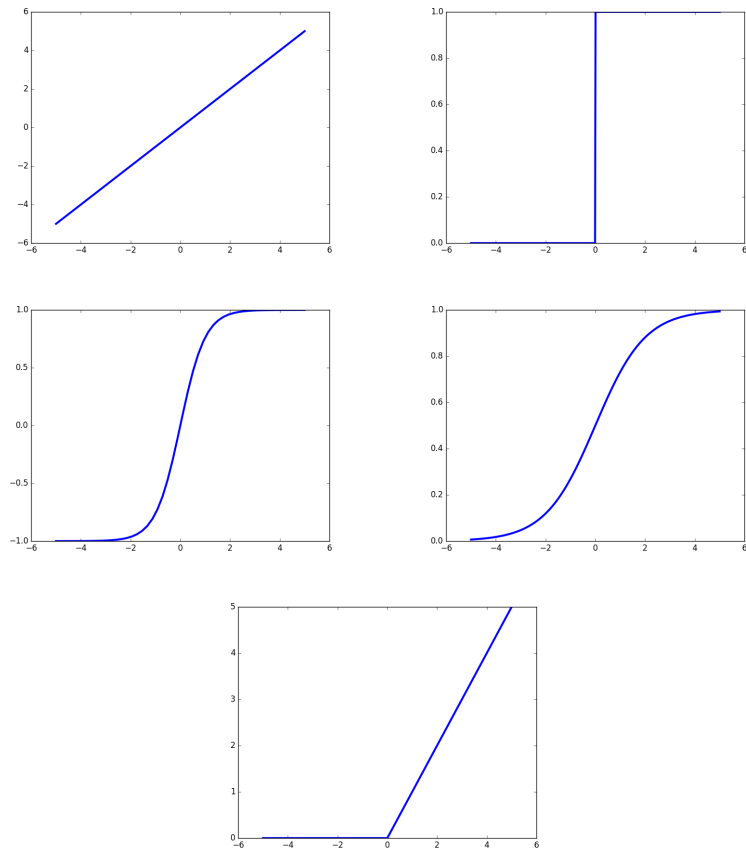


FIGURE 1.2 – Graphe entre -5 et 5 de la fonction identité (en haut à gauche), de l'échelon (en haut à droite), de la tangente hyperbolique (au milieu à gauche), de la sigmoïde (au milieu à droite) et de ReLU (en bas).

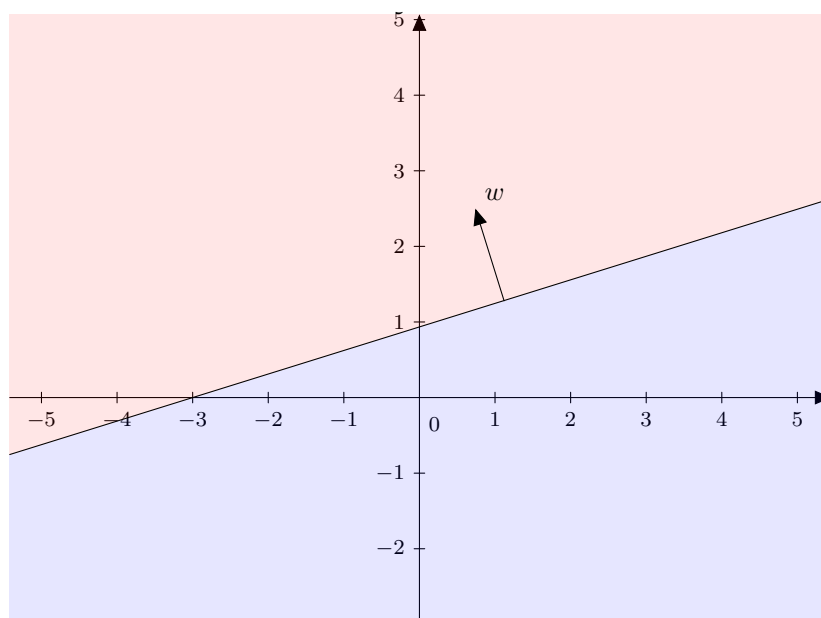


FIGURE 1.3 – Hyperplan séparateur dans un espace à deux dimensions correspondant à un certain vecteur de poids w . La partie rouge correspond aux entrées classifiées 1 et en bleue celles qui sont classifiées 0.

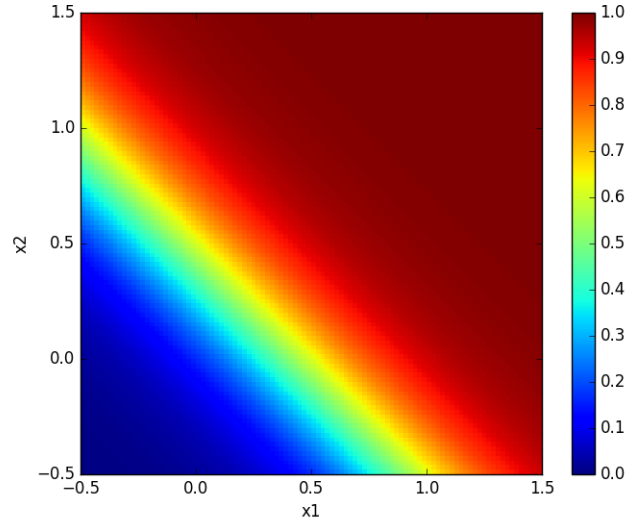


FIGURE 1.4 – Sortie d’une régression logistique prenant en entrée un vecteur de \mathbb{R}^2 sur $[-0.5, 1.5] \times [-0.5, 1.5]$. L’axe x_1 et l’axe x_2 correspondent respectivement à la première et à la deuxième coordonnée de l’entrée.

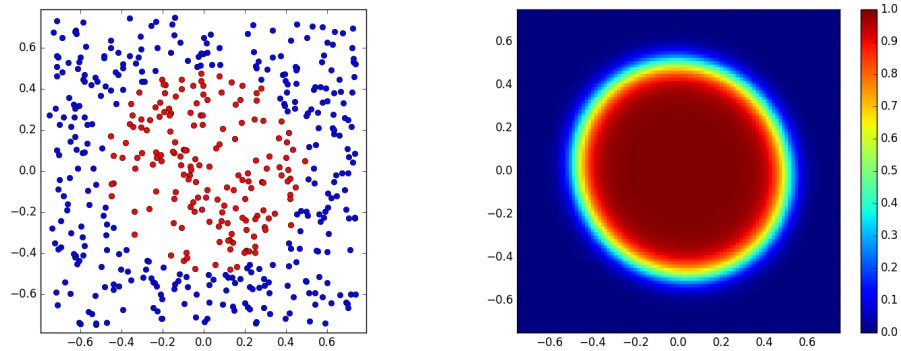


FIGURE 1.5 – À gauche un ensemble de données de 500 points. Les points à l’intérieur du cercle de rayon 0.5 et de centre $(0; 0)$ sont en rouge, les autres en bleu. À droite, la représentation du disque apprise par le neurone.

1.4 Un réseau de neurones

Définition 5 (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 :

$$y_j = f_j((y_i)_{i \in Pa(j)}) \quad (1.5)$$

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 1.6.

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 1.5 récursivement en partant des neurones dans V_{out} .

Ainsi, un premier algorithme naïf est :

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 mémorisation afin de garantir que la sortie de chaque neurone sera calculée au plus une fois. Le pseudo-code de cet algorithme correspond à l'algorithme 4.

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.

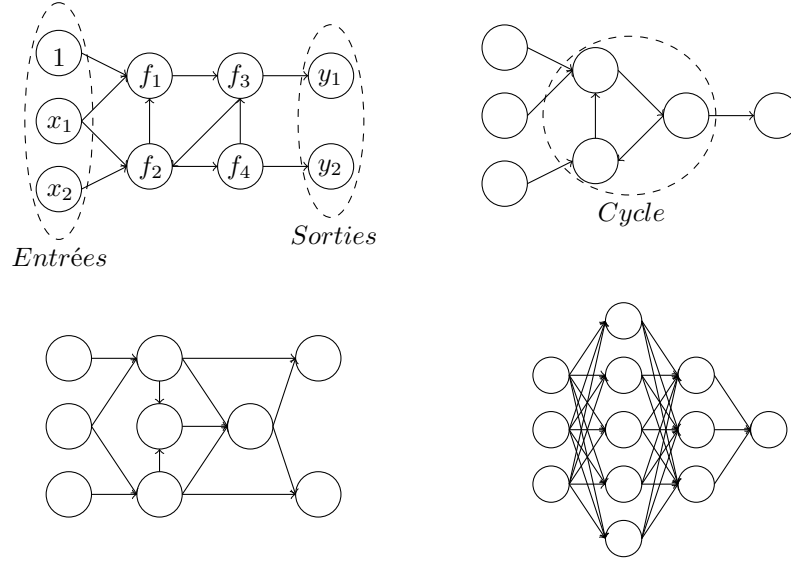


FIGURE 1.6 – Exemples de réseaux de neurones. Un réseau de neurones quelconque (en haut à gauche), récurrent (en haut à droite), feedforward (en bas à gauche) et multicouche (en bas à droite).

Algorithm 1 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  $j \in V_{in}$  then
      return  $x_j$ 
    else
       $t \leftarrow (evaluer\_neurone(i), i \in Pa(j))$ 
      return  $f_j(t)$ 
    end if
  end function
  for  $j \in V_{out}$  do
     $y_j \leftarrow evaluer\_neurone(j)$ 
  end for
end procedure

```

Algorithm 2 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( $\mathcal{N}(V, A, V_{in}, V_{out}, f), x$ )
  function evaluer_neurone( $j$ )
    if déjàCalculé[ $j$ ] est faux then
       $t \leftarrow (\text{evaluer\_neurone}(i), i \in Pa(j))$ 
       $y_j \leftarrow f_j(t)$ 
      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

```

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.

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.

Définition 6 (Fonction de vraisemblance). Si x_1, \dots, x_n sont des échantillons générés indépendamment par une probabilité p_θ la vraisemblance de l'échantillon est alors :

$$L(\theta) = \prod_{i=1}^N p_\theta(x^{(i)})$$

Remarque 2. Dans notre cas, les échantillons $(x_1, y_1), \dots, (x_n, y_n)$ sont générés par la loi $p_\theta(y|x)$ où θ sont les paramètres du modèle, la vraisemblance est donc :

$$L(\theta) = \prod_{i=1}^N p_\theta(y^{(i)}|x^{(i)})$$

Définition 7 (Maximum de vraisemblance). Le paramètre θ maximise la vraisemblance si $\theta \in \operatorname{argmax} L(\theta')$.

Remarque 3. Maximiser la vraisemblance revient donc à rechercher le maximum d'une fonction.

Remarque 4. Au lieu de maximiser la vraisemblance, nous allons minimiser l'opposé du logarithme de la vraisemblance (*negative log-likelihood* en anglais). On obtient alors :

$$NLL(\theta) = - \sum_{i=1}^N \log p_{\theta}(y^{(i)}|x^{(i)}) \quad (1.6)$$

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.

1.6.1 Régression

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

$$p_{\theta}(y|x) \sim \mathcal{N}(\hat{y}(x, \theta), \sigma) \Leftrightarrow p_{\theta}(y|x) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(y - \hat{y})^2}{2\sigma^2}\right) \quad (1.7)$$

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

En remplaçant p_{θ} par l'expression 1.7 dans l'équation 1.6, on obtient :

$$NLL(\theta) = \sum_{i=1}^N (y^{(i)} - \hat{y}(x^{(i)}))^2 + C \quad (1.8)$$

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, une fonction de coût obtenue par le principe de maximum de vraisemblance est :

$$E = \frac{1}{N} \sum_{i=1}^N (y^{(i)} - \hat{y}(x^{(i)}))^2 = \frac{1}{N} \sum_{i=1}^N e(y^{(i)}, \hat{y}(x^{(i)})) \quad (1.9)$$

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 :

$$p_\theta(y|x) \sim \mathcal{B}(\hat{y}(x, \theta)) \Leftrightarrow p_\theta(y|x) = \begin{cases} \hat{y}(x) & \text{si } y = 1 \\ 1 - \hat{y}(x) & \text{sinon} \end{cases} \quad (1.10)$$

La formule 1.10 peut se réécrire de manière plus compacte :

$$p_\theta(y|x) = \hat{y}(x)^y (1 - \hat{y}(x))^{1-y} \quad (1.11)$$

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

$$NLL(\theta) = - \sum_{i=1}^N y^{(i)} \log \hat{y}(x^{(i)}) + (1 - y^{(i)}) \log(1 - \hat{y}(x^{(i)})) \quad (1.12)$$

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

$$E = \frac{1}{N} \sum_{i=1}^N -(y^{(i)} \log \hat{y}(x^{(i)}) + (1 - y^{(i)}) \log(1 - \hat{y}(x^{(i)}))) = \frac{1}{N} \sum_{i=1}^N e(y^{(i)}, \hat{y}(x^{(i)})) \quad (1.13)$$

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 :

$$p_\theta(y|x) \sim \mathcal{M}(\hat{y}(x, \theta)_0, \dots, \hat{y}(x, \theta)_{M-1}) \Leftrightarrow p_\theta(y|x) = \hat{y}(x)_y \quad (1.14)$$

La *negative log-likelihood* associée est donc :

$$NLL(\theta) = - \sum_{i=1}^N \log \hat{y}(x^{(i)})_{y^{(i)}} \quad (1.15)$$

Posons pour tout $i \in \{0, \dots, M-1\}$, $\tau^{(i)} \in \mathbb{R}^M$ 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 1.15 en utilisant ces vecteurs :

$$NLL(\theta) = \sum_{i=1}^N - \sum_{j=0}^{M-1} \tau_j^{(y^{(i)})} \log \hat{y}(x^{(i)})_j \quad (1.16)$$

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 :

$$NLL(\theta) = \sum_{i=1}^N H(\tau^{(y^{(i)})}, \hat{y}(x^{(i)})) \quad (1.17)$$

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 :

$$E = \frac{1}{N} \sum_{i=1}^N H(\tau^{(y^{(i)})}, \hat{y}(x^{(i)})) = \frac{1}{N} \sum_{i=1}^N e(y^{(i)}, \hat{y}(x^{(i)})) \quad (1.18)$$

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, θ 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 θ 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 1.19.

$$\theta(t+1) = \theta(t) - \eta(t) \frac{\partial E}{\partial \theta} \quad (1.19)$$

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 1.4 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 :

$$\frac{\partial E}{\partial w_j} = \frac{\partial E}{\partial y_j} \frac{\partial y_j}{\partial s_j} \frac{\partial s_j}{\partial w_j} \quad (1.20)$$

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 1.20 devient :

$$\frac{\partial E}{\partial w_j} = \frac{\partial E}{\partial y_j} g'_j(s_j) x_j \quad (1.21)$$

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 $x_{i,j}$ l'entrée du neurone i correspondant à la sortie du neurone j , on a :

$$\frac{\partial E}{\partial y_j} = \sum_{i \in Ch(j)} \frac{\partial E}{\partial x_{i,j}} \quad (1.22)$$

La formule 1.22 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 :

$$\frac{\partial E}{\partial x_j} = \frac{\partial E}{\partial y_j} \frac{\partial y_j}{\partial s_j} \frac{\partial s_j}{\partial x_j} = \frac{\partial E}{\partial y_j} g'_j(s_j) w_j \quad (1.23)$$

Finalement, en utilisant les formules 1.21, 1.22 et 1.23, 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 :

Cet algorithme utilise la mémorisation 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 $O(|V| + |A|)$.

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

```

procédure retropropager_gradient( $\mathcal{N}(V, A, V_{in}, V_{out}, f), x, y$ )
  function calculer_gradient( $j$ )
    if déjàCalculéj est faux then
       $\frac{\partial E}{\partial y_j} \leftarrow \sum_{i \in Ch(j)} \text{calculer\_gradient}(i)_j$ 
       $\frac{\partial E}{\partial x_j} \leftarrow \frac{\partial E}{\partial y_j} g'_j(s_j) w_j$ 
       $\frac{\partial E}{\partial w_j} \leftarrow \frac{\partial E}{\partial y_j} g'_j(s_j) x_j$ 
      déjàCalculéj  $\leftarrow$  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 déjàCalculé de longueur  $|V|$  à faux.
  for  $i \in V_{out}$  do
    Calculer  $\frac{\partial E}{\partial y_i}$ 
    déjàCalculéi  $\leftarrow$  vrai
  end for
  for  $i \in V$  do
    calculer_gradient( $i$ )
  end for
end procédure

```

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

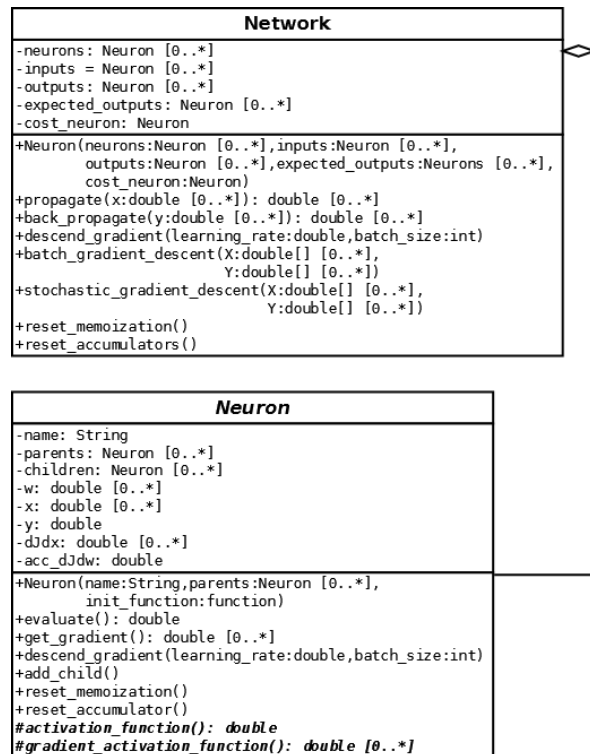


FIGURE 2.1 – Diagramme UML des classes Neuron et Network

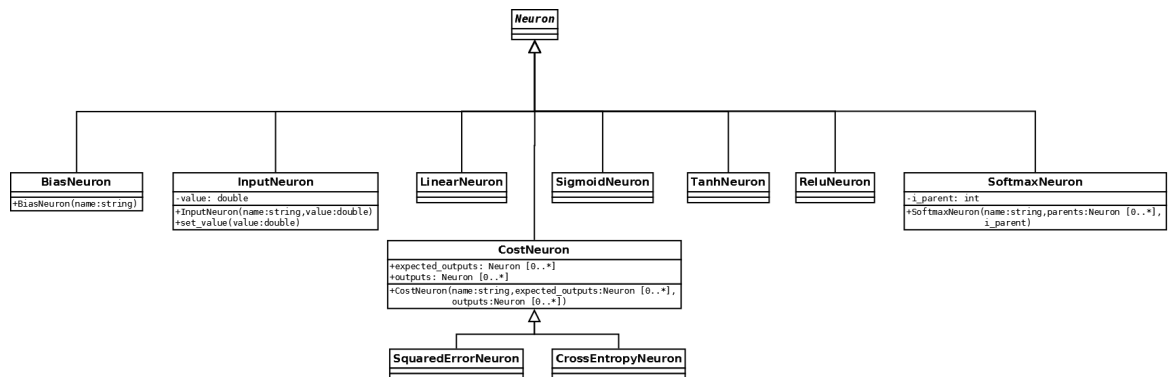


FIGURE 2.2 – Déclinaison de la classe neurone

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 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 η 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) :

$$w(t+1) = w(t) - \frac{\eta}{batch_size} acc_dJdw \quad (2.1)$$

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.

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 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 2.4).

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

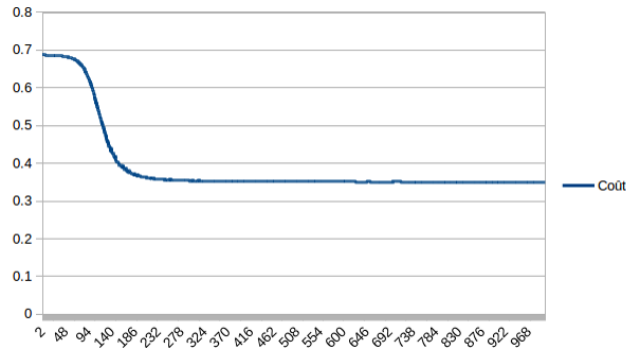


FIGURE 2.3 – fonction de coût bloquée dans un minimum local

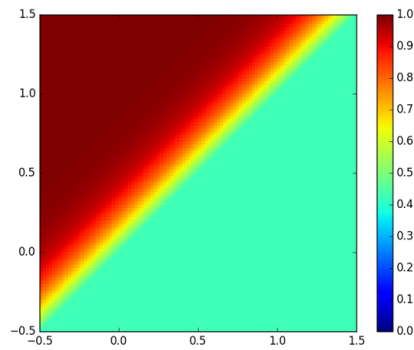


FIGURE 2.4 – XOR bloqué dans un minimum local

résultats comme celui de la figure 2.5.

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.

MNIST est une base de données de chiffres écrits à la main réalisée par

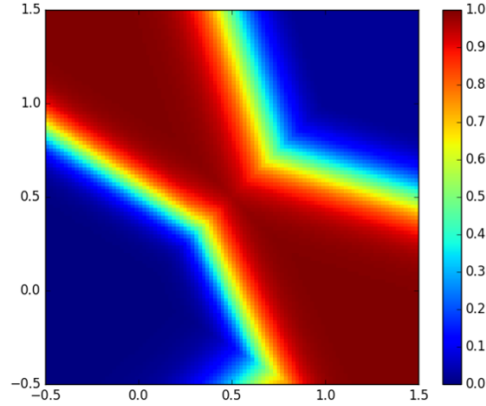


FIGURE 2.5 – 2 couches cachées de 4 neurones-ReLu

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.

2.5 Conclusion

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

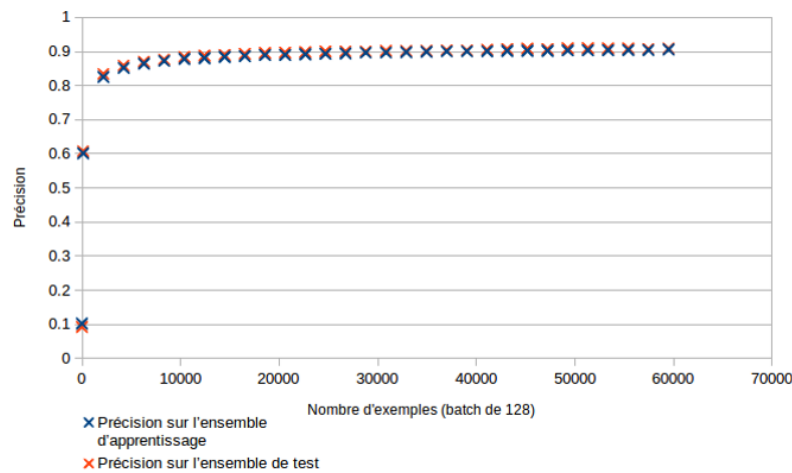


FIGURE 2.6 – Précision en fonction du nombre d'exemples appris

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 de neurones basée sur les graphes de calculs.

Chapitre 3

Graphes 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 σ . 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 σ 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.

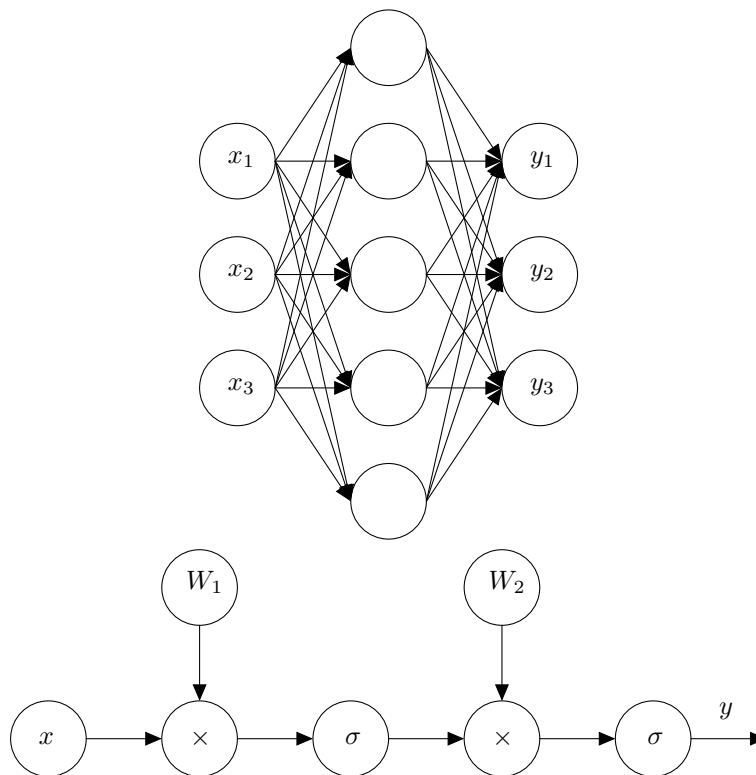


FIGURE 3.1 – Réseau de neurones à 3 entrées, 5 neurones cachés et 3 neurones de sorties (en haut). La représentation équivalente en utilisant un graphe de calculs (en bas).

3.2 Définitions

Définition 8 (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 θ 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 σ 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 4 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 dejaCalculé 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.

Un nœud a deux méthodes principales :

Algorithm 5 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

```

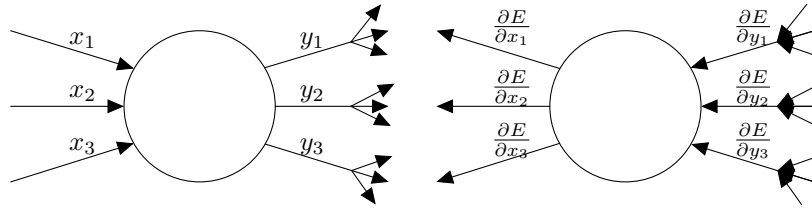


FIGURE 3.2 – Représentation de la propagation (à gauche) et de la rétropropagation (à droite) au sein d'un nœud.

- `get_output` : utilisée lors la propagation, le nœud demande ses entrées à ses parents puis calcule sa sortie et la mémoire ;
- `get_gradient` : utilisée lors de la rétropropagation, 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 la mémoire.

3.4.2 Les variables

Les nœuds variables sont des nœuds auxquels il est possible d'assigner une valeur via la méthode `set_value`. Il renvoie ensuite cette valeur lorsqu'un de ses enfants lui demande sa sortie.

Ils sont utilisés dans deux cas principaux. Le premier est évidemment les entrées du graphe. Le deuxième cas est plus subtile. Nous avons choisi comme pour la première implémentation avec des neurones d'inclure la fonction de coût dans nos graphe de calculs. Or cette fonction de coût prend comme argument la sortie du réseau \hat{y} , mais aussi la sortie attendue y . On utilise donc des nœuds variables pour spécifier la sortie attendue.

3.4.3 Les poids

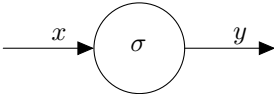
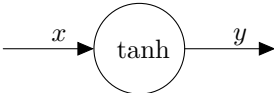



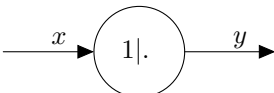
Un nœud poids est un nœud auquel est associé une matrice qui représente des poids. Le nœud poids renvoie cette matrice lorsqu'un de ses enfants lui demande sa sortie.

En plus de sa matrice de poids, le nœud poids possède un accumulateur qui sert à accumuler le gradient sur plusieurs batches. Ainsi à chaque fois que le nœuds poids calcule la dérivée du coût par rapport à ses poids, il ajoute cette valeur à son accumulateur.

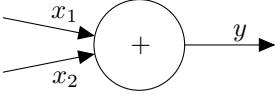
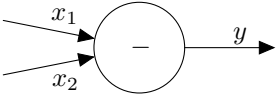
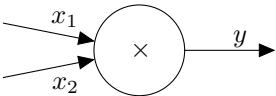
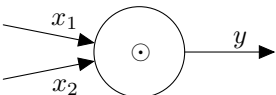
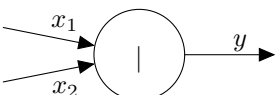


3.4.4 Les opérations mathématiques

Dans cette sous-section, nous allons décrire les symboles ainsi que les formules de propagation et de rétropropagation des nœuds opérations 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
	$\sigma(x) = \frac{1}{1+\exp(-x)}$	$\frac{\partial E}{\partial y} \odot y \odot (1 - y)$
	$\tanh(x)$	$\frac{\partial E}{\partial y} \odot (1 - y \odot y)$
	$ReLU(x)$	$\frac{\partial E}{\partial y} \odot (\mathbf{1}_{x_{i,j} \geq 0})_{i,j}$
	$softmax(x)$	$\left(\sum_{k=1}^n y_{i,j} (\delta_{k,j} - y_{i,k}) \frac{\partial E}{\partial y_{i,k}} \right)_{i,j}$
	$\ x\ _2^2$	$2 \frac{\partial E}{\partial y} x$
	$1 x = \begin{pmatrix} 1 & x \end{pmatrix}$	$\left(\frac{\partial E}{\partial y_{i,j}} \right)_{i \in \{1, \dots, m\}, j \in \{2, \dots, n+1\}}$

Puis ceux à deux entrées et une sortie :

Symbole	Formule de propagation	Formule de rétropropagation
	$x_1 + x_2$	$(\frac{\partial E}{\partial y}, \frac{\partial E}{\partial y})$
	$x_1 - x_2$	$(\frac{\partial E}{\partial y}, -\frac{\partial E}{\partial y})$
	$x_1 \times x_2$	$(\frac{\partial E}{\partial y} x_2^T, x_1^T \frac{\partial E}{\partial y})$
	$x_1 \odot x_2$	$(\frac{\partial E}{\partial y} \odot x_2, \frac{\partial E}{\partial y} \odot x_1)$
	$x_1 x_2 = (x_1 \quad x_2)$	$((\frac{\partial E}{\partial y_{i,j}})_{i \in \{1, \dots, m\}, j \in \{1, \dots, n_1\}}, (\frac{\partial E}{\partial y_{i,j}})_{i \in \{1, \dots, m\}, j \in \{n_1+1, \dots, n_1+n_2\}})$
	$\text{sigmoidCE}(x_1, x_2)$	$(-\frac{\partial E}{\partial y} \odot (\frac{x_2}{1-x_2}), -\frac{\partial E}{\partial y} \odot (\frac{x_1}{x_2} - \frac{1-x_1}{1-x_2}))$
	$\text{softmaxCE}(x_1, x_2)$	$(-\frac{\partial E}{\partial y} \odot \log(x_2), -\frac{\partial E}{\partial y} \odot (\frac{x_1}{x_2}))$

3.4.5 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 d'obtenir de telles formules.

Définition 9 (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 10 (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 5. 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 6. 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 2 (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 :

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

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 3.4.4. 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 reconnaît 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 3. 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 3.

Démonstration. L'opération σ 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 4. 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 4.

Démonstration. De même que pour σ , 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 reconnait 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 5. Si $y = \text{ReLU}(x)$ alors $\frac{\partial E}{\partial x} = \frac{\partial E}{\partial y} \odot (\mathbb{1}_{x_{i,j} \geq 0})_{i,j}$.

Démonstration.

$$\forall i \in \{1, \dots, m\}, j \in \{1, \dots, n\}, \frac{\partial E}{\partial x_{i,j}} = \frac{\partial E}{\partial y_{i,j}} \frac{\partial y_{i,j}}{\partial x_{i,j}} = \frac{\partial E}{\partial y_{i,j}} \mathbb{1}_{x_{i,j} \geq 0}$$

On reconnait alors une multiplication terme à terme d'où le résultat annoncé. □

Proposition 6. Si $y = \text{softmax}(x)$ alors $\frac{\partial E}{\partial x} = \left(\sum_{k=1}^n y_{i,j} (\delta_{k,j} - y_{i,k}) \frac{\partial E}{\partial y_{i,k}} \right)_{i,j}$.

Lemme 3. Si $y = \text{softmax}(x) = \left(\frac{e^{x_j}}{\sum_{k=1}^n e^{x_k}} \right)_j$ alors :

$$\forall i \in \{1, \dots, n\}, \forall j \in \{1, \dots, n\}, \frac{\partial y_i}{\partial x_j} = y_i(\delta_{i,j} - y_j)$$

Démonstration. Pour tout $i, j \in \{1, \dots, n\}$:

$$\frac{\partial y_i}{\partial x_j} = \begin{cases} \frac{e^{x_i} \sum_{k=1}^n e^{x_k} - (e^{x_i})^2}{(\sum_{k=1}^n e^{x_k})^2} & \text{si } i = j \\ \frac{-e^{x_i} e^{x_j}}{(\sum_{k=1}^n e^{x_k})^2} & \text{sinon} \end{cases} = \begin{cases} y_i(1 - y_i) & \text{si } i = j \\ -y_i y_j & \text{sinon} \end{cases} = y_i(\delta_{i,j} - y_j)$$

□

Utilisons ce lemme pour démontrer la proposition 6 :

Démonstration.

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

□

Proposition 7. Si $y = \|x\|_2^2$ alors $\frac{\partial E}{\partial x} = 2 \frac{\partial E}{\partial y} x$.

Démonstration.

$$y = \|x\|_2^2 = \sum_{i,j} x_{i,j}^2$$

d'où

$$\forall i \in \{1, \dots, m\}, j \in \{1, \dots, n\}, \frac{\partial E}{\partial x_{i,j}} = \frac{\partial E}{\partial y} \frac{\partial y}{\partial x_{i,j}} = 2 \frac{\partial E}{\partial y} x_{i,j}$$

et donc :

$$\frac{\partial E}{\partial x} = 2 \frac{\partial E}{\partial y} x$$

□

Proposition 8. Si $y = 1|x = (1 \ x)$ alors $\frac{\partial E}{\partial x} = (\frac{\partial E}{\partial y} \frac{\partial y}{\partial x_{i,j}})_{i \in \{1, \dots, m\}, j \in \{2, \dots, n+1\}}$.

Démonstration.

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

□

Démontrons maintenant les formules de rétropropagation pour les nœuds à deux entrées.

Proposition 9. 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 $\frac{\partial E}{\partial x_2}$. \square

Proposition 10. 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 $i \in \{1, \dots, m\}, j \in \{1, \dots, n\}$, $\frac{\partial y_{i,j}}{\partial x_{2,i,j}} = -1$. \square

Proposition 11. 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 là l'expression du produit matriciel entre x_1^T et $\frac{\partial E}{\partial y}$, d'où :

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

\square

Proposition 12. 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 $\frac{\partial E}{\partial x_2}$. \square

Proposition 13. Si $y = x_1|x_2$ alors $\frac{\partial E}{\partial x_1} = (\frac{\partial E}{\partial y_{i,j}})_{i \in \{1, \dots, m\}, j \in \{1, \dots, n_1\}}$ et $(\frac{\partial E}{\partial y_{i,j}})_{i \in \{1, \dots, m\}, j \in \{n_1+1, \dots, n_1+n_2\}}$.

Démonstration.

$$\forall i \in \{1, \dots, m\}, j \in \{1, \dots, n_1\}, \frac{\partial E}{\partial x_{1,i,j}} = \frac{\partial E}{\partial y_{i,j}} \frac{\partial y_{i,j}}{\partial x_{1,i,j}} = \frac{\partial E}{\partial y_{i,j}}$$

$$\forall i \in \{1, \dots, m\}, j \in \{1, \dots, n_2\}, \frac{\partial E}{\partial x_{2,i,j}} = \frac{\partial E}{\partial y_{i,n_1+j}} \frac{\partial y_{i,n_1+j}}{\partial x_{2,i,j}} = \frac{\partial E}{\partial y_{i,n_1+j}}$$

□

Proposition 14. Si $y = \text{sigmoidCE}(x_1, x_2)$ alors $\frac{\partial E}{\partial x_1} = -\frac{\partial E}{\partial y} \odot \left(\frac{x_2}{1-x_2}\right)$ et $\frac{\partial E}{\partial x_2} = -\frac{\partial E}{\partial y} \odot \left(\frac{x_1}{x_2} - \frac{1-x_1}{1-x_2}\right)$.

Proposition 15. Si $y = x_1 \odot x_2$ alors $\frac{\partial E}{\partial x_1} = -\frac{\partial E}{\partial y} \odot \log(x_2)$ et $\frac{\partial E}{\partial x_2} = -\frac{\partial E}{\partial y} \odot \left(\frac{x_1}{x_2}\right)$.

3.5 La classe Graph

La classe graphe est une classe utilitaire. Toute la logique de la propagation et de la rétropropagation est contenue dans les nœuds. Elle sert à stocker les nœuds créés auparavant et leur fonction : entrée, poids, coût ou sortie. Il est ensuite possible d'utiliser les méthodes suivantes afin d'interagir facilement avec les nœuds :

- **propagate** : met à jour les entrées des nœuds variables et calcule la sortie du graphe ;
- **backpropagate** : calcule la dérivée du coût par rapport à chaque poids.

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

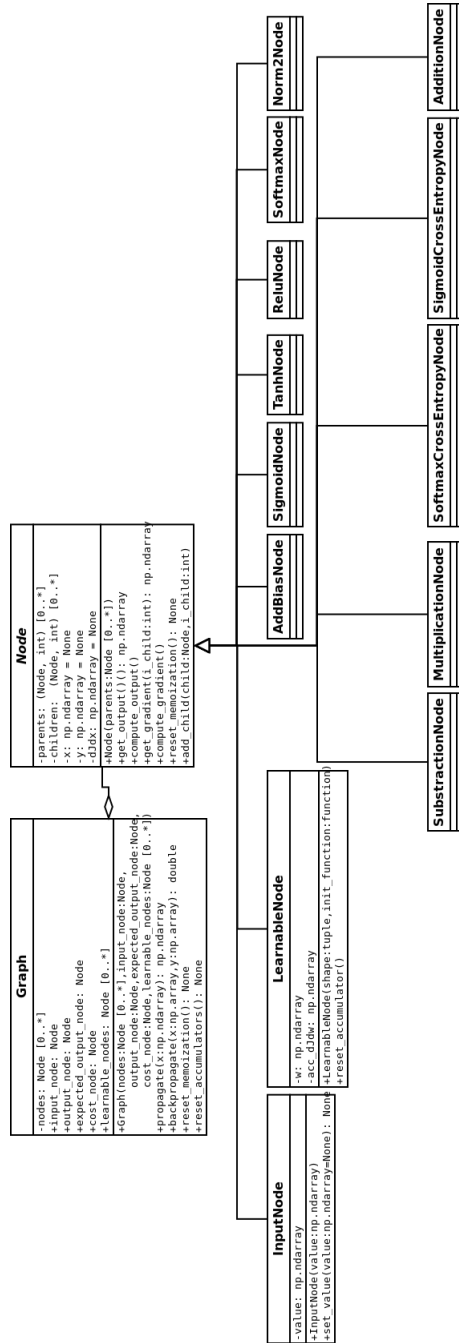


FIGURE 3.3 – Diagramme UML de pychain

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.

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.

Finalement, comme nous le verrons, cette implémentation restera valide pour la modélisation des réseaux de neurones récurrents.

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 rappelés et détaillés ici.

Nous avons principalement étudié l'influence de l'architecture du réseau sur la rapidité de l'apprentissage.

La figure 4.1 présente la précision obtenue pour des structures d'une seule couche cachée avec un nombre variable de neurones de fonction d'activation tanh. Moins le réseau possède de neurones sur sa couche cachée, plus il a besoin d'apprendre sur un nombre important d'itérations. Cependant, avec seulement deux neurones sur la couche cachée, l'apprentissage se fait déjà relativement rapidement et le gain obtenu en ajoutant des états cachés n'est pas vraiment significatif puisqu'il ajoute un coût temporel de calcul.

Cette même étude réalisée dans les mêmes conditions en remplaçant simplement la fonction d'activation tangente hyperbolique par une ReLu donne des résultats complètement différents visible en figure 4.2. Ici, la taille de la couche cachée prend beaucoup plus d'importance puisqu'elle devient le paramètre principal pour obtenir une précision parfaite rapidement. En effet, même si augmenter le nombre d'itérations permet à des architectures avec peu d'états cachés permet d'améliorer la précision du réseau, cette amélioration est très lente. À l'inverse le réseau apprend presque instantanément lorsque l'on augmente le nombre d'états cachés.

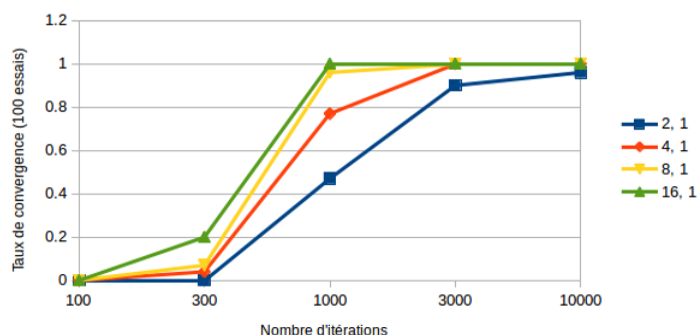


FIGURE 4.1 – Convergence en fonction de l'architecture avec tanh

L'exemple de ces deux situations représente clairement les problèmes rencontrés pour étudier les influences des paramètres d'un réseau de neurone. En effet, l'influence de la taille de la couche cachée sur notre réseau n'est plus du

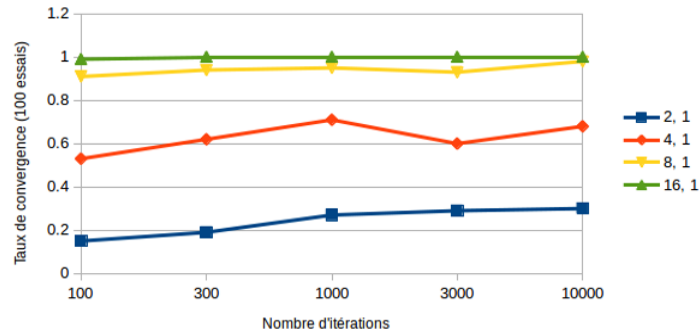


FIGURE 4.2 – Convergence en fonction de l'architecture avec relu

tout la même selon le choix de la fonction d'activation appliquée à chaque neurone. Ainsi, il n'est souvent pas possible de définir une règle générale à appliquer concernant un paramètre pris indépendamment des autres.

4.3 Étude de MNIST

4.3.1 Introduction

Dans cette partie, mis à part les paramètres explicitement précisés pour un exemple, toutes les mesures qui vont être présentées furent réalisées en utilisant les paramètres suivants :

- Pas de couche cachée, les neurones d'entrée sont directement reliés aux sorties
- Les entrées sont centrées et normalisées : chaque pixel est alors représenté par une valeur de $[-1; 1]$
- Les poids sont initialisés aléatoirement avec une fonction Gaussienne centrée de variance 0,1
- On utilise une sigmoïde en sortie
- On utilise l'erreur quadratique pour la fonction de coût
- Les exemples sont traités pas batch de 128
- On fixe un learning rate de 0,1

4.3.2 Architecture du réseau

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

De façon intuitive, il paraît évident qu'une plus grande architecture de réseau, possédant plus de neurones par couches cachées ou plus de couches cachées apprendra avec plus de précision l'ensemble de données. Cependant cela a un coût en temps et rendra l'apprentissage beaucoup plus lent. Ainsi, alors qu'un

apprentissage effectué sans couche cachée sur 30 passages du set de données mnist aura une précision d'environ 93%, un réseau composé d'une couche cachée de 400 neurones ayant pour fonction d'activation tangente hyperbolique pourra avoir sur ce même ensemble presque 98% de précision. Le résultat de cette seconde situation est visible en figure 4.3. On y remarque que l'ensemble de données est très rapidement complètement appris.

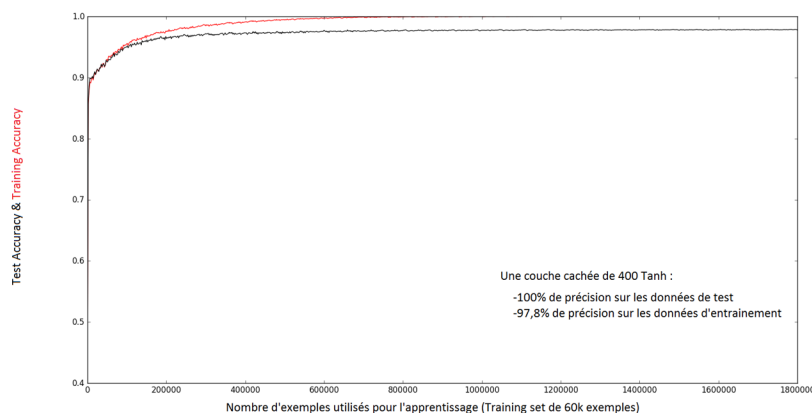


FIGURE 4.3 – Précision pour un réseau avec une couche cachée de 400 neurones

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

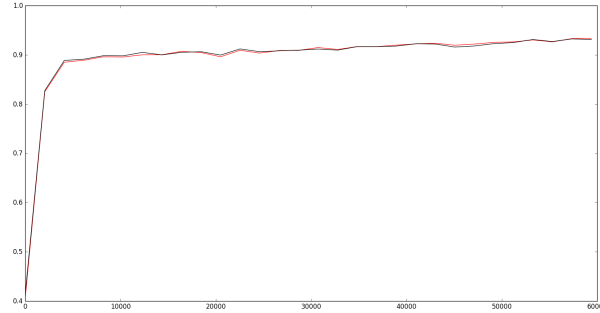


FIGURE 4.4 – Précision sur les ensembles de test (en noir) et d'apprentissage (en rouge) en fonction du nombre d'exemples utilisés pour l'apprentissage pour une initialisation des poids d'amplitude 0.1

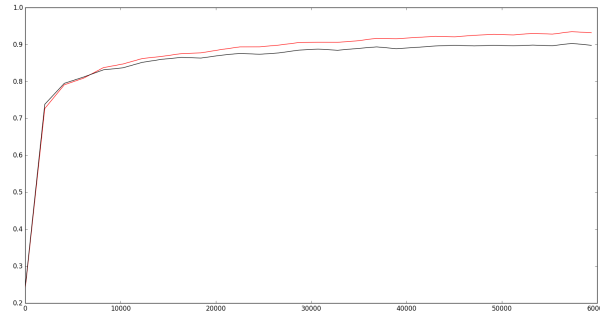


FIGURE 4.5 – Précision sur les ensembles de test (en noir) et d'apprentissage (en rouge) en fonction du nombre d'exemples utilisés pour l'apprentissage pour une initialisation des poids d'amplitude 0.1

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 et de coût

Le tableau 4.6 présente quelques résultats sur la précision obtenu par le réseau selon les fonctions de coût et les fonctions en sortie utilisées. Le choix de la fonction de coût ne semble avoir qu'un impact très mineur sur les résultats obtenus par le réseau. L'erreur quadratique présente des performances légèrement supérieures à l'entropie croisée pour ce problème mais cette différence est très faible.

Sortie	Softmax		Sigmoïde	
Fonction de coût	Entropie croisée	Erreur quadratique	Entropie croisée	Erreur quadratique
Moyenne	0,92303	0,92667	0,9149	0,91708
Ecart type	0,0003335	0,000565784	0,000258199	0,000285968
Intervalle de confiance	0,000206702	0,00035067	0,00016003	0,000177242

FIGURE 4.6 – Influence de la fonction de coût sur la précision obtenue par le réseau

En revanche il apparait que l'utilisation de la fonction Softmax en sortie donne de meilleurs résultats que celle d'une sigmoïde. L'emploi de Softmax entrainant un cout en temps plus important que celui de Sigmoïde, cet avantage est à relativiser.

4.3.5 Taille des batchs et nombre de passages

La taille d'un batch est le nombre d'exemples que l'on insère dans le réseau entre chaque rétropropagation du gradient.

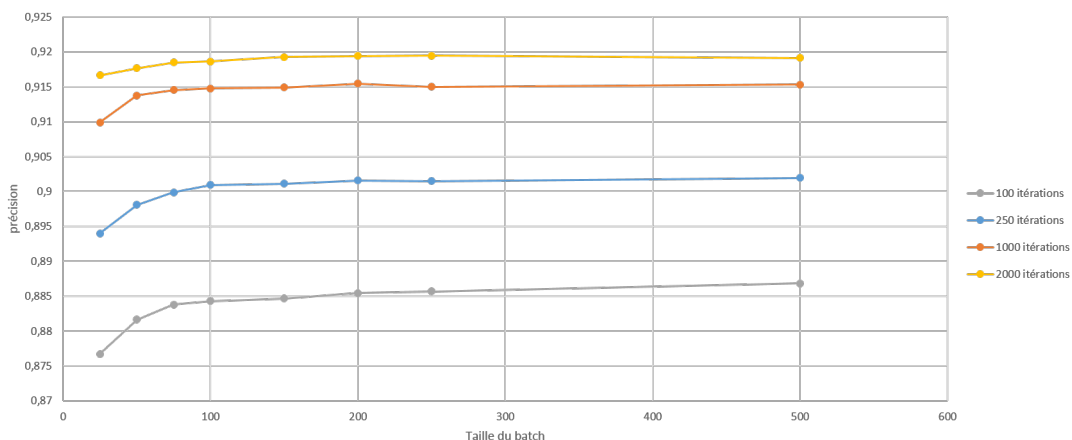


FIGURE 4.7 – Influence de la taille du batch

La figure 4.7 présente la précision obtenue pour différentes tailles de batch. On remarque qu'au delà de 150, augmenter la taille du batch n'améliore presque plus les performances du réseau. La figure 4.8 présente les mêmes données que la figure 4.7 mais réparties en fonction du temps de calcul. Utiliser des batchs plus petits semble être plus efficace pour l'apprentissage des données mnist.

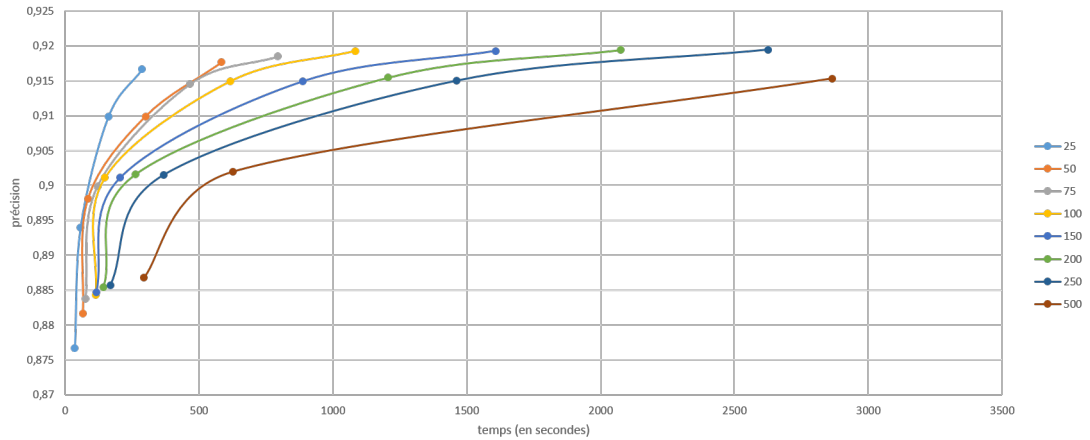


FIGURE 4.8 – Influence de la taille du batch sur le temps de calcul

4.3.6 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 4.9.

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.
- 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).

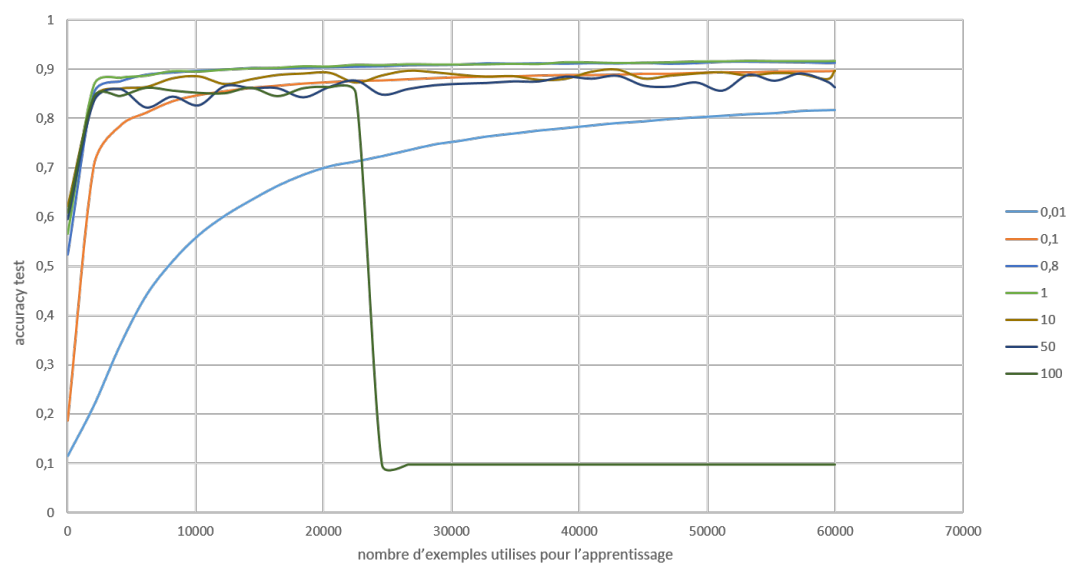


FIGURE 4.9 – Influence du learning rate sur la précision sur l'ensemble de test en fonction du nombre d'exemple utilisés

Chapitre 5

Réseaux de neurones récurrents

5.1 Motivation

Nous avons pu voir que les réseaux feedforward peuvent être utilisés afin d'apprendre à prédire la sortie voulue 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 faire de la prédiction et de la génération de séquence.

Dans un premier, nous verrons comment modéliser ces réseaux récurrents. Puis nous nous intéresserons à deux algorithmes d'optimisation permettant l'apprentissage des réseaux récurrents : real time recurrent learning (RTRL) et backpropagation through time (BPTT).

5.2 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 5.1. On les appellera arête "retard".

On peut maintenant remarquer que pour une séquence x_1, \dots, x_n le réseau récurrent de la figure 5.1 est équivalent au réseau feedforward de la figure 5.2.

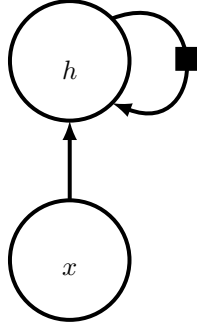


FIGURE 5.1 – Réseau de neurones récurrent où l'arête avec un carré signifie que le nœud h prend sa valeur au temps $t - 1$ comme entrée au temps t .

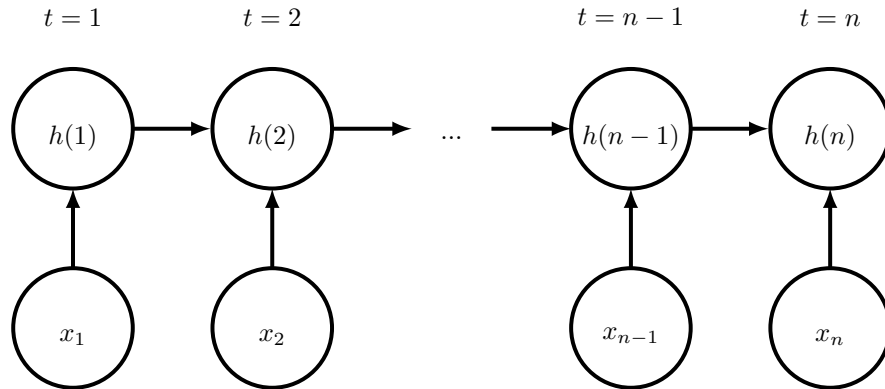


FIGURE 5.2 – Réseau de neurones feedforward équivalent au réseau récurrent de la figure 5.1 pour la séquence d'entrée x_1, \dots, x_n .

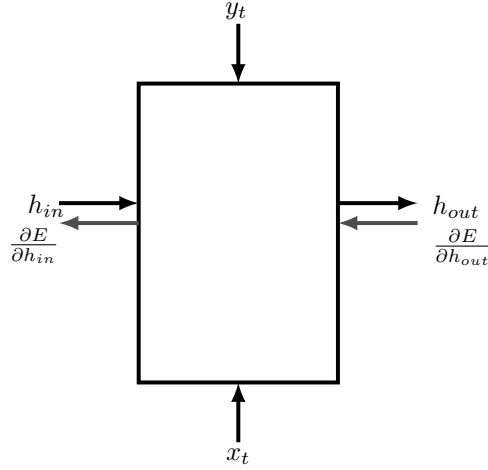


FIGURE 5.3 – Représentation d'un layer. Il contient les nœuds du graphe à un temps t . Il prend en entrée la sortie de l'état caché du temps $t - 1$ appelée h_{in} et l'entrée x_t . Il a comme sortie l'état caché h_{out} ainsi que \hat{y}_t qui sera comparé avec y_t la sortie attendue au temps t . Le *layer* sert aussi lors de la phase de rétropropagation.

Le processus de passage d'un réseau récurrent à un réseau feedforward s'appelle dépliage.

Une fois le réseau déplié obtenu, on peut utiliser l'algorithme de propagation classique pour les graphes de calculs afin de calculer la sortie du réseau récurrent.

Nous avons utilisé trois méthodes pour générer des réseaux dépliés. La première prend en entrée un réseau récurrent avec des arêtes retards et obtient le dépliage à partir de celui-ci.

La seconde prend en entrée un *layer*, représenté figure 5.3, c'est-à-dire le sous-graphe qui va être copié à chaque pas de temps et le clone autant de fois qu'il le faut.

Enfin la dernière méthode est de ne pas utiliser d'algorithme "général" permettant de passer d'un graphe récurrent à un graphe acyclique. Mais plutôt d'écrire pour chaque architecture de réseau une fonction dépliant ce réseau. Une telle fonction contiendra principalement une boucle créant les nœuds nécessaires pour chaque temps.

Devant la diversité des architectures des réseaux récurrents, la dernière méthode est la plus sensée et la plus pratique. Elle offre une liberté totale quant à l'architecture du graphe tout en étant relativement simple.

5.3 BPTT

Contrairement, à ce que nous avons fait lors du projet, nous allons d'abord présenter l'algorithme *backpropagation through time* (BPTT) car il s'agit d'une simple généralisation de la rétropropagation déjà présentée pour les réseaux feedforward.

L'algorithme consiste en deux étapes :

1. Déplier le réseau récurrent pour avoir un réseau feedforward.
2. Appliquer l'algorithme de rétropropagation à ce réseau feedforward.

La seule difficulté conceptuelle réside donc dans le dépliage.

En pratique, le dépliage est un processus coûteux en temps. Lors de l'apprentissage, on évite donc de déplier un graphe pour chaque exemple d'apprentissage. Pour cela, si cela est possible, on fixe la taille des séquences d'entrée lors de l'apprentissage. Nous pourrions ainsi utiliser le même réseau pour tous les exemples. En outre, comme tous les exemples auront la même taille, nous pourrions les placer dans une matrice comme pour les réseaux feedforward et passer un batch d'exemples en même temps ce qui permet d'accélérer encore les calculs en utilisant pleinement la puissance de `numpy`.

5.4 RTRL

Détaillons maintenant l'algorithme RTRL qui a une philosophie complètement différente. Au lieu de rétropropager le gradient, nous allons le propager.

Nous nous plaçons dans le cas particulier où le réseau est fully connected comme sur la figure 5.4.

Nous notons I l'ensemble des neurones d'entrée et U l'ensemble des autres neurones. Posons alors une notation unifiant ces deux ensembles :

$$z_k(t) = \begin{cases} x_k(t) & \text{si } k \in I \\ y_k(t-1) & \text{si } k \in U \end{cases} \quad (5.1)$$

avec $y_k(0) = 0$ pour tout $k \in U$.

Posons aussi, $s_k(t)$ la somme pondérée des entrées du neurone k au temps t :

$$s_k(t) = \sum_{l \in U \cup I} w_{k,l} z_l(t) \quad (5.2)$$

On a alors :

$$\forall k \in U, y_k(t) = f_k(s_k(t))$$

La fonction de coût total est la somme des coûts partiels que l'on calcule à chaque temps t :

$$E = \sum_{t=1}^n E(t)$$

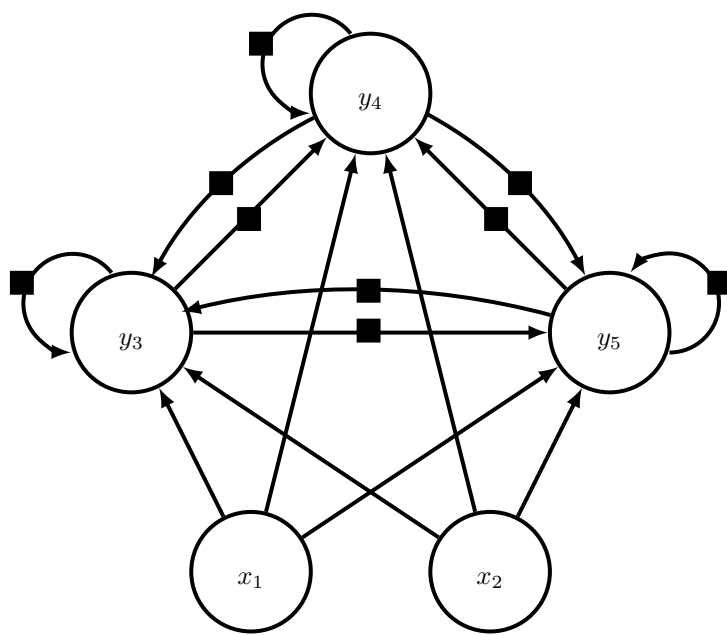


FIGURE 5.4 – Réseau de neurones récurrent fully connected avec 2 entrées et 3 autres neurones.

Nous pouvons alors calculer la dérivée du coût partiel au temps t , $E(t)$, par rapport à un poids $w_{i,j}$:

$$\frac{\partial E(t)}{\partial w_{i,j}} = \sum_{k \in U} \frac{\partial E(t)}{\partial y_k(t)} \frac{\partial y_k(t)}{\partial w_{i,j}} \quad (5.3)$$

Le terme $\frac{\partial E(t)}{\partial y_k(t)}$ est connu, il dépend de la fonction de coût utilisée. Pour une fonction de coût quadratique $E(t) = \sum_{k \in U} (y_k(t) - d_k(t))^2$ avec $d_k(t)$ la valeur attendue pour $y_k(t)$, on a $\frac{\partial E(t)}{\partial y_k(t)} = 2(y_k(t) - d_k(t))$.

Il nous reste donc le terme $\frac{\partial y_k(t)}{\partial w_{i,j}}$ à calculer. Déterminons une relation de récurrence entre $\frac{\partial y_k(t+1)}{\partial w_{i,j}}$ et les $(\frac{\partial y_l(t)}{\partial w_{i,j}})_{l \in U}$:

$$\frac{\partial y_k(t+1)}{\partial w_{i,j}} = \frac{\partial y_k(t+1)}{\partial s_k(t+1)} \frac{\partial s_k(t+1)}{\partial w_{i,j}} = f'_k(s_k(t+1)) \frac{\partial s_k(t+1)}{\partial w_{i,j}}$$

En utilisant la formule 5.2, on a alors :

$$\begin{aligned} \frac{\partial y_k(t+1)}{\partial w_{i,j}} &= f'_k(s_k(t+1)) \sum_{l \in I \cup U} \frac{\partial s_k(t+1)}{\partial z_l(t+1)} \frac{\partial z_l(t+1)}{\partial w_{i,j}} + \frac{\partial s_k(t+1)}{\partial w_{k,l}} \frac{\partial w_{k,l}}{\partial w_{i,j}} \\ &= f'_k(s_k(t+1)) \left(\sum_{l \in I \cup U} w_{k,l} \frac{\partial z_l(t+1)}{\partial w_{i,j}} + \delta_{i,k} z_j(t+1) \right) \end{aligned}$$

Puis en utilisant la formule 5.1, on a finalement :

$$\frac{\partial y_k(t+1)}{\partial w_{i,j}} = f'_k(s_k(t+1)) \left(\sum_{l \in U} w_{k,l} \frac{\partial y_l(t)}{\partial w_{i,j}} + \delta_{i,k} z_j(t+1) \right) \quad (5.4)$$

Utilisons les formules 5.3 et 5.4 pour écrire l'algorithme de propagation du gradient. Dans l'algorithme 6, p représente les $(\frac{\partial y_k(t)}{\partial w_{i,j}})_{k,i,j}$ et ϵ les $(\frac{\partial E}{\partial w_{i,j}})_{i,j}$. On effectue la propagation des entrées en même temps que la propagation du gradient.

On parle d'apprentissage en temps réel car il est possible d'apprendre après élément de la séquence x_1, \dots, x_n . Ou bien d'accumuler le gradient sur toute la séquence x_1, \dots, x_n comme présenté dans l'algorithme 6. Nos expériences ont montré qu'il était mieux d'accumuler le gradient sur une séquence en entière voire sur des batches de plusieurs séquences.

5.5 Comparaison des deux algorithmes

La première remarque à faire est que si on prend le réseau décrit dans la section précédente et que l'on accumule le gradient sur une séquence alors les algorithmes RTRL et BPTT calculent la même quantité : $\frac{\partial E}{\partial w}$.

On pourra vérifier en pratique que les deux algorithmes calculent bien la même quantité dans la section suivante où les résultats sont identiques.

Intéressons, nous maintenant aux complexités de ces deux algorithmes. Le tableau ci-dessous résume leurs complexités en espace et en temps où l'on considère un réseau à n neurones et une séquence taille L .

Algorithm 6 Algorithme RTRL.

```
procedure propager_gradient( $I, U, w, f, x, d$ )
  Initialiser un tableau  $z$  à 2 dimensions de taille  $(n + 1, |I \cup U|)$  à 0.
  Initialiser un tableau  $s$  de taille  $|U|$  à 0.
  Initialiser un tableau  $p$  à 3 dimensions de taille  $(|U|, |U|, |I \cup U|)$  à 0.
  Initialiser un tableau  $\epsilon$  à 2 dimensions de taille  $(|U|, |I \cup U|)$  à 0.
  for  $t \in \{1, \dots, n\}$  do
    // Propagation
    for  $k \in I$  do
       $z[t - 1][k] \leftarrow x[t][k]$ 
    end for
    for  $k \in U$  do
       $s[k] \leftarrow \sum_{l \in U \cup I} w_{k,l} z[t - 1][l]$ 
       $z[t][k] \leftarrow f_k(s[k])$ 
    end for
    // Propagation de l'erreur
    for  $k \in U$  do
      for  $i \in U$  do
        for  $j \in I \cup U$  do
           $p[k][i][j] \leftarrow f'_k(s[k]) (\sum_{l \in U} w_{k,l} p[l - 1][i][j] + \delta_{i,k} z[t][j])$ 
        end for
      end for
    end for
    for  $i \in U$  do
      for  $j \in I \cup U$  do
         $\epsilon[i][j] \leftarrow \epsilon[i][j] + \sum_{k \in U} 2(z[t][k] - d[t][k]) p[k][i][j]$ 
      end for
    end for
  end for
end procedure
```

	Complexité en temps	Complexité en espace
RTRL	$O(n^4)$	$O(n^3)$
BPTT	$O(Ln^2)$	$O(Ln)$

Justifions rapidement ces résultats. Pour RTRL :

- Complexité en espace : besoin de stocker un tableau p de dimension 3 d'où un coût en $O(n^3)$.
- Complexité en temps : la mise à jour de chaque élément du tableau nécessite le calcul d'une somme de n termes d'où un coût en $O(n^4)$.

Et pour BPTT :

- Complexité en espace : valeurs de sortie des Ln neurones (après dépliage).
- Complexité en temps : chacun des Ln neurones met à jour ses poids en $O(n)$ ($O(1)$ pour chacun de ses n poids) d'où un coût en $O(Ln^2)$.

L'algorithme BPTT est donc bien plus intéressant du point de vue temporel et même spatial si la taille de la séquence n'est pas trop élevée.

L'algorithme BPTT a donc de nombreux avantages par rapport à RTRL :

- conceptuellement, il s'agit de la continuité naturelle de la rétropropagation des réseaux feedforward pour les réseaux récurrents ;
 - il possède une meilleure complexité temporelle et spatiale que RTRL ;
 - il peut s'appliquer à n'importe quelle architecture de réseaux récurrents, et donc en particulier à l'architecture LSTM que nous verrons plus tard.
- Alors que RTRL lui n'a été développé que pour les réseaux fully connected, même s'il doit être possible de déterminer les formules de propagation pour d'autres architectures.

5.6 Évaluation

Nous avons implémenté les algorithmes BPTT et RTRL avec comme objectif d'apprendre à des réseaux récurrents la grammaire de Reber.

Dans la première sous-section est décrit avec plus de détails le problème à résoudre. Dans la deuxième sous-section, sont exposés les résultats obtenus par les deux algorithmes.

5.6.1 Grammaire de Reber

Dans un premier temps, nous avons comparé ces algorithmes sur l'apprentissage de la grammaire de Reber, qui est régulière. Nous utiliserons la grammaire simple présentée sur la figure 5.5.

À l'aide de cette automate, nous avons généré 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.

Plus précisément, si $w_1w_2...w_n$ est un mot reconnu par le grammaire de Reber. Nous associons à chaque lettre de l'ensemble $\{B, T, P, S, X, V, E\}$ un nombre dans $\{1, ..., 7\}$. Puis nous créons une séquence $(x_1, ..., x_n)$ où pour tout

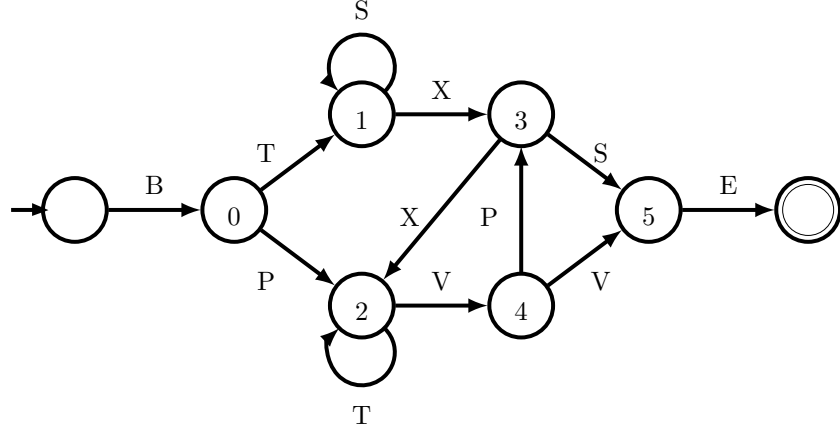


FIGURE 5.5 – Automate reconnaissant la grammaire de Reber simple

$i, x_i \in \mathbb{R}^7$ est la lettre w_i *one hot encoded*. Par exemple si w_1 est B et que B est associé au nombre 1 alors $x_1 = (1, 0, \dots, 0)$. À chaque temps $t \in \{1, \dots, n - 1\}$, nous voulons que le réseau prédise les transitions possibles pour de l'automate après qu'il est vu $w_1 \dots w_t$.

Les réseaux que nous utiliserons, émettront à chaque temps t un vecteur de \mathbb{R}^7 où chaque coordonnée sera comprise entre 0 et 1. Nous dirons que le réseau a prédit une transition utilisant la lettre w au temps t si la coordonnée correspondante du vecteur a une valeur supérieure à un seuil défini. Nous avons utilisé 0.3 comme valeur seuil.

Lors des phases de test, nous calculons un score pour évaluer le réseau. Ce score est la proportion des mots pour lesquels il a prédit correctement toutes les transitions.

Ce premier problème permet d'évaluer la capacité des réseaux récurrents à traiter des séquences de taille variable.

L'étape suivante consistera à tester les algorithmes sur l'apprentissage de la grammaire de Reber symétrique présentée sur la figure 5.6.

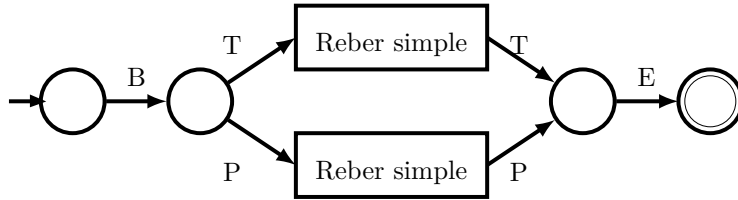


FIGURE 5.6 – Automate reconnaissant la grammaire de Reber symétrique

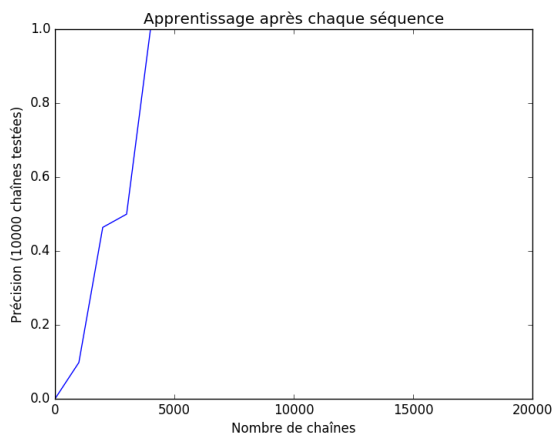
Cette fois le problème qui nous intéresse est de savoir si le réseau est capable

de retenir une information sur le long terme. Ainsi si nous évaluerons le réseau sur sa capacité à bien se souvenir que la deuxième lettre est la même que l'avant dernière lettre d'un mot de la grammaire de Reber symétrique.

Cette fois nous utilisons un score différent. Le score sera la proportion des mots pour lesquels le réseau a correctement prédit que la deuxième lettre était la même que l'avant dernière.

5.6.2 Résultats

Caractère présenté	B	T	P	S	X	V	E
B	0.01	0.48	0.51	0.01	0.01	0.01	0.01
T	0.00	0.02	0.00	0.56	0.43	0.02	0.01
X	0.00	0.01	0.00	0.53	0.47	0.02	0.00
X	0.01	0.47	0.01	0.03	0.02	0.50	0.02
V	0.01	0.01	0.51	0.00	0.00	0.47	0.03
V	0.00	0.00	0.01	0.02	0.02	0.00	0.97



5.7 Problème des dépendances temporelles

Pour conclure cette partie, nous allons effectuer une analyse théorique montrant les limitations des réseaux récurrents fully-connected. Il en ressortira la nécessité d'une architecture plus complexe permettant la conservation des dépendances temporelles à long terme.

Nous allons prendre un réseau et des notations similaires à celui utilisé dans la partie 5.4 sur l'algorithme RTRL. On retrouvera une visualisation de l'architecture considérée dans la figure 5.7. On prend un réseau avec n neurones cachés numérotés de 1 à n , $m - n$ neurones de sortie numérotés de $n + 1$ à m et $p - m$ neurones d'entrée numérotés de $m + 1$ à p .

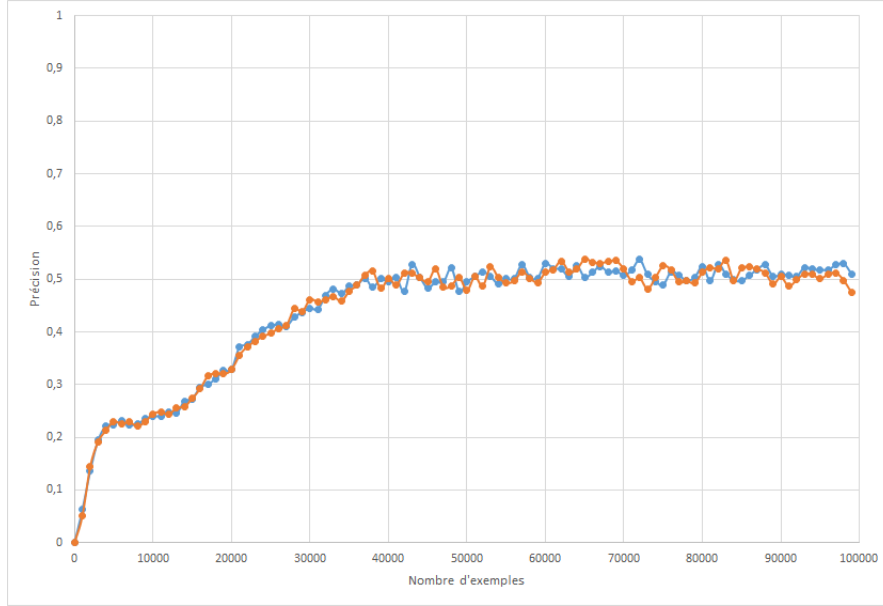


FIGURE 5.7 – Résultats des algorithmes BPTT (rouge) et RTRL (bleu) sur l'apprentissage d'une grammaire de Reber symétrique, moyenné sur 100 essais ??

Les neurones cachés prennent comme entrées au temps t les entrées du réseau au temps t ainsi que les sorties des neurones cachés au temps $t - 1$. Tandis que les neurones de sortie prennent comme entrées les sorties des neurones cachés au temps t .

On note alors pour $i \in \{1, \dots, m\}$:

$$\begin{aligned}
 - & s_i(t) = \begin{cases} \sum_{j=1}^n w_{ij} y_j(t-1) + \sum_{j=m+1}^p w_{ij} y_j(t) & \text{si } i \in \{1, \dots, n\} \\ \sum_{j=1}^n w_{ij} y_j(t) & \text{si } i \in \{n+1, \dots, m\} \end{cases} \\
 - & y_i(t) = f_i(s_i(t)) \\
 - & E = \sum E(t) \\
 - & \epsilon_i(t) = \frac{\partial E}{\partial s_i(t)}
 \end{aligned}$$

Notre objectif est de montrer que la dépendance temporelle entre un neurone caché du temps t et d'un neurone du temps $t + q$ diminue rapidement avec q . Ce qui montrera que cette architecture n'est pas adaptée pour apprendre des dépendances à long terme.

$\epsilon_i(t)$ représente le gradient de l'erreur par rapport à l'activation d'un neurone i du temps t et $\epsilon_j(t+q)$ le gradient de l'erreur par rapport à l'activation d'un neurone j du temps $t+q$. On voudrait montrer que ces deux quantités s'influencent de moins en moins lorsque q augmente. En effet cela montrerait que lors de est difficile d'apprendre des dépendances à long terme. Pour cela, nous allons donc étudier la quantité $\frac{\partial \epsilon_i(t)}{\partial \epsilon_j(t+q)}$ et montrer que celle-ci décroît

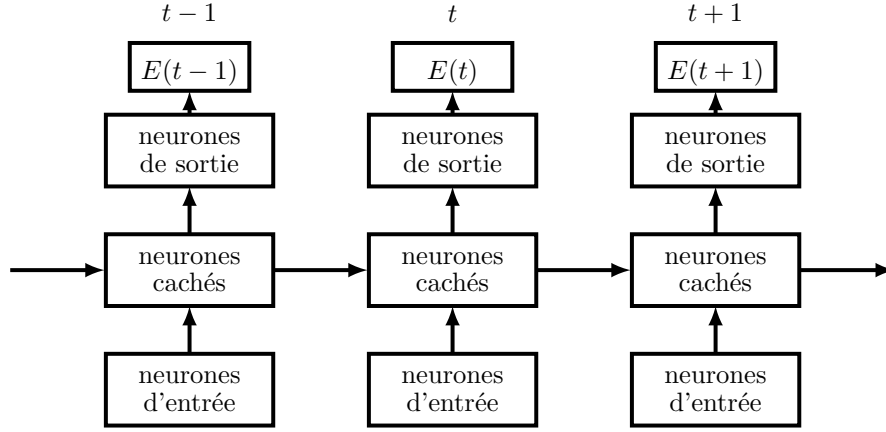


FIGURE 5.8 – Architecture considérée dans la sous-section 5.7

rapidement avec q .

Pour un neurone de sortie, on a :

$$\epsilon_i(t) = \frac{\partial E}{\partial y_i(t)} \frac{\partial y_i(t)}{\partial s_i(t)} = \frac{\partial E}{\partial y_i(t)} f'_i(s_i(t))$$

De même pour un neurone caché, d'après la règle de la chaîne :

$$\epsilon_i(t) = \sum_{j=0}^n \frac{\partial E}{\partial s_j(t+1)} \frac{\partial s_j(t+1)}{\partial s_i(t)} + \sum_{j=n+1}^m \frac{\partial E}{\partial s_j(t)} \frac{\partial s_j(t)}{\partial s_i(t)}$$

D'où :

$$\epsilon_i(t) = \sum_{j=0}^n \epsilon_j(t+1) f'_i(s_i(t)) w_{ji} + \sum_{j=n+1}^m \epsilon_j(t) f'_i(s_i(t)) w_{ji}$$

On en déduit :

$$\frac{\partial \epsilon_i(t)}{\partial \epsilon_j(t+1)} = f'_i(s_i(t)) w_{ji}$$

D'où par récurrence :

$$\frac{\partial \epsilon_i(t)}{\partial \epsilon_j(t+q)} = \sum_{l_1=0}^n \sum_{l_2=0}^n \dots \sum_{l_{q-1}=0}^n \prod_{i=1}^q f'_{l_i}(t+i-1) w_{l_{i+1}l_i}$$

Cette quantité représente un flux local.

Réécrivons cette quantité en utilisant une notation matricielle :

$$\frac{\partial \epsilon_i(t)}{\partial \epsilon_j(t+q)} = W_j^T F'(t+q-1) \left(\prod_{k=2}^{q-1} W F'(t+q-k) \right) W_i f'_i(s_i(t))$$

avec :

- $W = (w_{ij})_{i,j}$
- $F'(t) = \text{diag}(f'_1(s_1(t)), \dots, f'_n(s_n(t)))$

Soit une norme matricielle $\|\cdot\|_A$ compatible avec une norme vectorielle $\|\cdot\|_x$ telle que $\max_{1,\dots,n}(|x_i|) \leq \|x\|_x$.

L'hypothèse sur la norme $\|\cdot\|_x$ donne :

$$|x^T y| = \left| \sum_{i=1}^n x_i y_i \right| \leq n \max(|x_i|) \max(|y_i|) \leq n \|x\|_x \|y\|_x$$

On a alors :

$$\left| \frac{\partial \epsilon_i(t)}{\partial \epsilon_j(t+q)} \right| \leq n \|W_j\|_x \|F'(t+q-1) \left(\prod_{k=2}^{q-1} W F'(t+q-k) \right) W_i f'_i(s_i(t))\|_x$$

Par compatibilité :

$$\left| \frac{\partial \epsilon_i(t)}{\partial \epsilon_j(t+q)} \right| \leq n \|W_j\|_x \|W\|_A^{q-2} \|W_i\|_x (f'_{max})^q$$

avec $f'_{max} = \max_{1,\dots,n}(\|F'(i)\|_A)$

On remarque que :

$$\forall i, \|W_i\|_x = \|W e_i\|_x \leq \|W\|_A \|e_i\|_x \leq \lambda \|W\|_A$$

avec $e_i = (1_{i=j})_j$ et $\lambda = \max_{1,\dots,n}(\|e_i\|_x)$

Finalement :

$$\left| \frac{\partial \epsilon_i(t)}{\partial \epsilon_j(t+q)} \right| \leq \lambda n \|W\|_A^q (f'_{max})^q$$

On prend $\|A\|_A = \max_i(\sum_j |a_{ij}|)$, $\|x\|_x = \max_i(|x_i|)$ et $f_i(t) = \sigma(t) = \frac{1}{1+\exp(-t)}$. On a alors $f'_{max} = \frac{1}{4}$, $\lambda = 1$ et $\|W\|_A \leq n w_{max}$ avec $w_{max} = \max_{i,j}(|w_{ij}|)$, d'où :

$$\left| \frac{\partial \epsilon_i(t)}{\partial \epsilon_j(t+q)} \right| \leq n \left(\frac{n w_{max}}{4} \right)^q$$

On en conclut que si $w_{max} < \frac{4}{n}$ alors on a une décroissance exponentielle.

Ce résultat dit donc que si les poids sont inférieurs à une certaine valeur alors, on peut être sûr que le réseau ne pourra pas modéliser convenablement les dépendances à long terme.

Afin de contourner ce problème, nous introduisons dans la partie suivante une architecture de réseaux de neurones récurrents, le LSTM, qui a été développée pour résoudre ce problème spécifiquement.

Chapitre 6

Long Short-Term Memory (LSTM)

6.1 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 6.1 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.

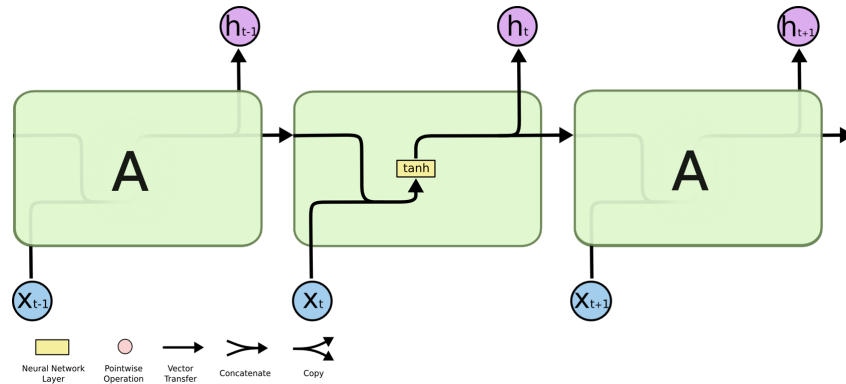


FIGURE 6.1 – Dépliage d'un RNN classique

6.2 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.

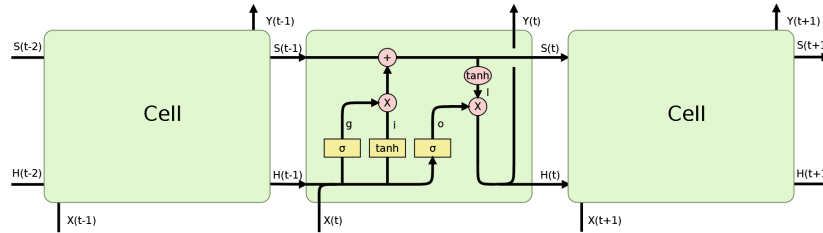


FIGURE 6.2 – Cellule élémentaire des LSTM

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.

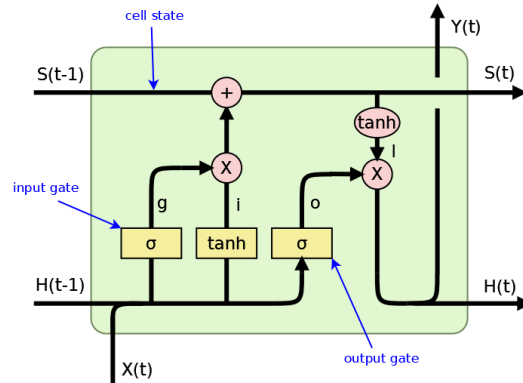


FIGURE 6.3 – Différentes parties d'une cellule LSTM

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 6.4 fait apparaître le gradient à chaque endroit de la cellule.

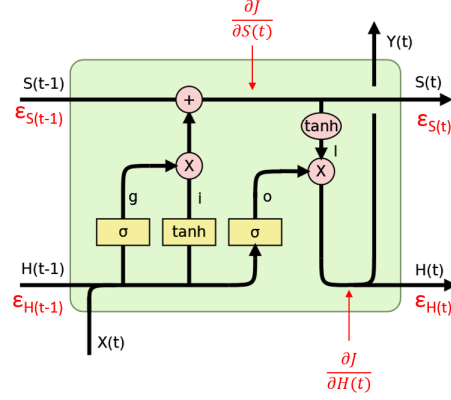


FIGURE 6.4 – Cellule LSTM

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

$$\varepsilon_{S(t-1)} = \frac{\partial J}{\partial S(t)} = \varepsilon_{S(t)} + F'_l \text{diag}(o) \frac{\partial J}{\partial H(t)} \quad (6.1)$$

$$\varepsilon_{H(t-1)} = \left((W_g^T F'_g \text{diag}(i) + W_i^T F'_i \text{diag}(g)) \frac{\partial J}{\partial S(t)} + W_o^T F'_o \text{diag}(l) \frac{\partial J}{\partial H(t)} \right)_{1, \dots, n} \quad (6.2)$$

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.

$$\frac{\partial J}{\partial W_o} = \begin{pmatrix} H(t-1) \\ x(t) \end{pmatrix} \left(F'_o \text{diag}(l) \frac{\partial J}{\partial H(t)} \right)^T \quad (6.3)$$

$$\frac{\partial J}{\partial W_i} = \begin{pmatrix} H(t-1) \\ x(t) \end{pmatrix} \left(F'_i \text{diag}(g) \frac{\partial J}{\partial S(t)} \right)^T \quad (6.4)$$

$$\frac{\partial J}{\partial W_g} = \begin{pmatrix} H(t-1) \\ x(t) \end{pmatrix} \left(F'_g \text{diag}(i) \frac{\partial J}{\partial S(t)} \right)^T \quad (6.5)$$

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 :

$$W = W - \eta \sum \frac{\partial J}{\partial W} \quad (6.6)$$

6.3 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).

Si cette implémentation intuitive fonctionne et donne des résultats satisfaisants, elle n'est cependant pas très adapté à la création de réseaux de cellules LSTM. C'est pour cela que nous nous sommes lancés dans une seconde implémentation, plus pratique et flexible.

6.4 Implémentation avec des nœuds

Cette seconde implémentation a pour but de réutiliser `pychain`, la librairie de graphes de calculs que nous avons développée et déjà appliquée aux réseaux feedforward et aux réseaux récurrents. L'utilisation des graphes de calculs nous permettra d'optimiser les calculs mais surtout de gagner en flexibilité. En effet, on pourra alors facilement changer l'architecture de la cellule LSTM mais aussi très simplement combiner plusieurs cellules entre elles. Ou même combiner des cellules LSTM avec des réseaux feedforward.

L'objectif est de créer un nœud à trois entrées et à deux sorties comme décrit sur la figure 6.6.

Deux solutions s'offraient à nous. On pouvait réutiliser les équations de propagation et de rétropropagation décrite dans la section précédente. Cette solution a l'inconvénient qu'il est nécessaire de dériver à la main les formules de propagation et de rétropropagation à chaque fois que l'on veut changer l'architecture de la cellule LSTM. On n'atteint donc pas l'objectif de flexibilité.

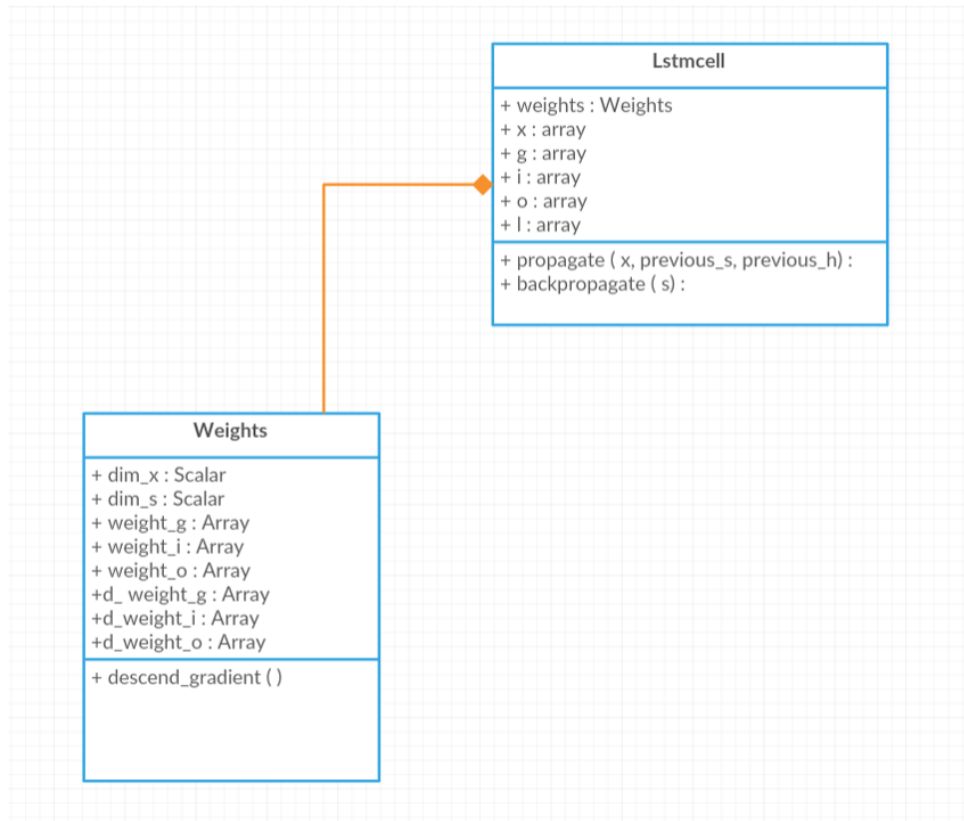


FIGURE 6.5 – Diagramme UML de la cellule LSTM

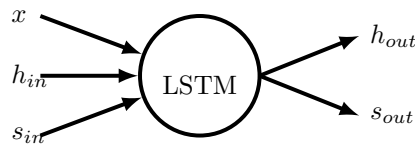


FIGURE 6.6 – Architecture macroscopique de la cellule LSTM.

L'autre solution consiste à exprimer le nœud LSTM en fonction d'autres nœuds élémentaires que nous avons définis dans la partie 3.4. Comme cela, le gradient sera calculé automatiquement par les nœuds. Afin de créer un tel nœuds, nous avons défini une classe *CompositeNode* qui est basée sur le design pattern Composite. Il s'agit d'une classe fille de la classe *Node* qui a comme attributs d'autres nœuds. La classe *LSTMNode* est alors une classe fille de la classe *CompositeNode*.

Grâce à cette architecture nous avons pu tester différentes variantes de la cellule LSTM : avec ou sans forget gate. Mais surtout, nous avons considérablement augmenté la puissance de notre librairie en permettant de créer des nœuds à partir d'autres nœuds ce qui permet de créer des modèles beaucoup plus complexes.

Nous avons donc pu soumettre des cellules LSTM à l'apprentissage de la grammaire symétrique de Reber.

Les LSTM peuvent donc apprendre parfaitement une grammaire de Reber au contraire des deux algorithmes appliqués à des réseaux récurrents simples. On peut aussi noter que que l'apprentissage se fait très rapidement et de manière efficace. Les oscillations sont moins conséquentes et surtout il n'y a pas de palier,

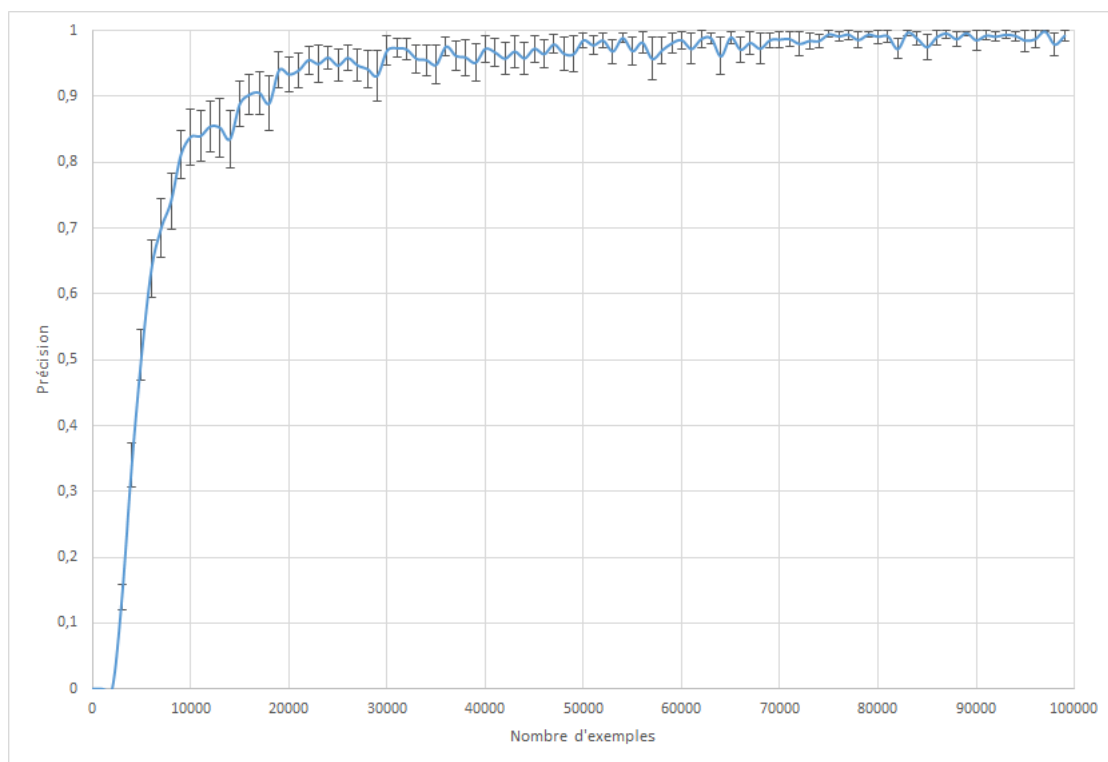


FIGURE 6.8 – Résultats de l'apprentissage de la grammaire de Reber symétrique pour une cellule LSTM, moyenné sur 100 essais ??

ce qui montre l'efficacité de l'apprentissage.

Cela montre bien la force des LSTM qui peuvent donc prendre en compte de manière efficace le contexte afin de prédire une suite à ce contexte. C'est dans ce cadre là que nous nous sommes penchés vers la génération de séquence afin de pouvoir vraiment tester les possibilités offertes par celles-ci.

Chapitre 7

Application : génération de texte

7.1 Introduction

La première application des LSTM à laquelle nous nous sommes intéressés est la génération de séquences. Le principe est assez explicite : après apprentissage sur un ensemble de séquences, on veut pouvoir en générer une nouvelle qui répond aux mêmes règles grammaticales que les séquences de l'ensemble d'apprentissage.

7.2 Principe

On entraîne un réseau à prédire le prochain caractère d'une séquence, puis on lui fait générer des lettres selon cet apprentissage.

7.2.1 Architecture du réseau

Le réseau retenu comporte 2 cellules LSTM dont les états cachés sont de taille 128.

Les caractères sont convertis au format "One Hot Encoded" : un caractère est représenté par un vecteur ne contenant que des 0 sauf un 1 à la ligne correspondant à sa nature dans un dictionnaire prédéfini. La sortie du réseau est un softmax donnant une distribution de probabilité pour le caractère suivant.

Afin de pouvoir fixer le nombre d'état caché indépendamment du format des entrées, il est nécessaire d'ajouter un étage feed forward d'adaptation en sortie du réseau. Ce réseau permettra de passer d'un vecteur de la taille de l'état caché à un vecteur de la taille de la sortie (qui sera aussi la taille de l'entrée dans notre cas).

7.2.2 Entraînement

Les caractères sont présentés en entrée par séquence de longueur fixée. La taille de la séquence correspond à la taille du dépliage que l'on décide de réaliser lors de l'application de l'algorithme BPTT. Cette longueur est donc choisie afin de ne pas produire un dépliage trop volumineux qui demanderait plus de mémoire que ne peut en fournir l'ordinateur.

La figure 7.1 montre un tel dépliage sur un exemple simpliste.

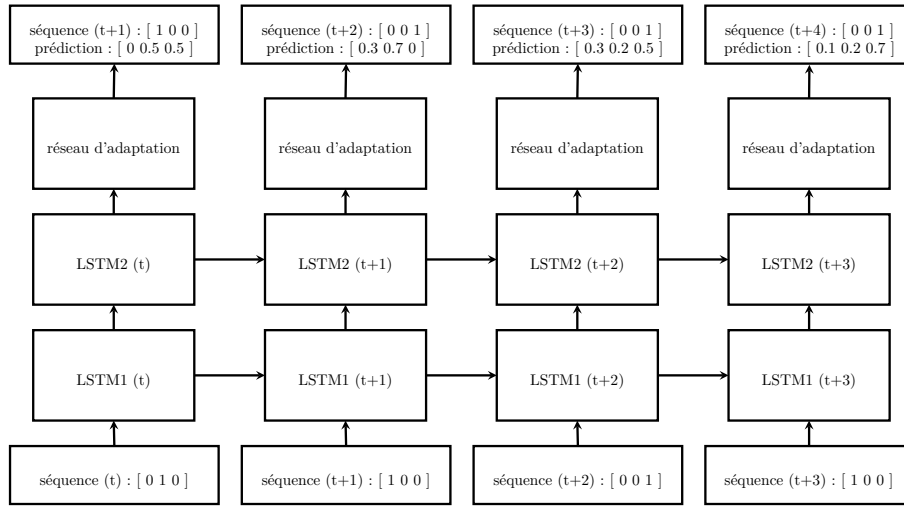


FIGURE 7.1 – Exemple de dépliage de longueur 4 pour l'apprentissage de séquence

Dans notre implémentation l'apprentissage est accéléré en traitant plusieurs séquences en parallèle. On met alors non plus un vecteur représentant un caractère en entrée mais une matrice représentant les caractères au temps t de chacune des séquences dont on parallélise l'apprentissage. La diminution du temps de calcul provient de l'optimisation des calculs matriciels sous Numpy.

7.2.3 Génération

Pour générer un extrait de texte, on donne en entrée du réseau au temps $t+1$ un caractère obtenue par tirage sur la distribution probabiliste obtenue en sortie du dernier étage au temps t . Le premier caractère est choisie arbitrairement parmi l'ensemble des caractères de l'ensemble d'apprentissage.

La figure 7.2 développe un exemple simpliste de génération.

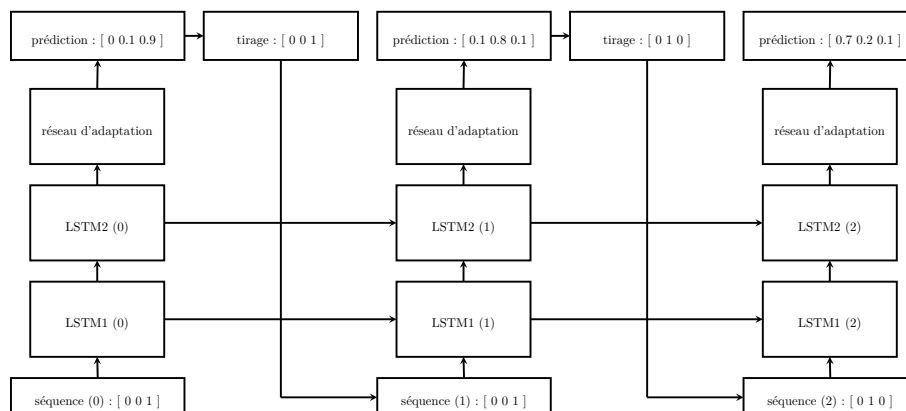


FIGURE 7.2 – Exemple de génération d'un extrait de longueur 3

7.3 Génération à partir d'œuvres de Shakespeare

L'objectif de départ était de générer des textes, l'ensemble d'apprentissage type pour cet exercice est l'ensemble des pièces de théâtre de shakespeare. Cet ensemble possède 3 avantages majeurs. Tout d'abord la langue anglaise possède moins de caractères distincts que le français. Ensuite ils possèdent tous la structure de dialogue *nomdupersonnage : texte*, qui est facilement repérable et constituera donc un critère d'évaluation du résultat. Enfin l'anglais tel qu'il est écrit par shakespeare est très reconnaissable. L'ensemble de textes ainsi constitué comporte 40 000 lignes de texte.

7.3.1 Résultats

On entraîne le réseau par batchs de 50 séquences de 50 caractères. Les résultats obtenus avec l'algorithme classique de descente du gradient n'étaient pas satisfaisants et nous ont poussé à implémenter d'autres algorithmes d'optimisation qui sont décrits au chapitre 8. On utilise alors RMSProp qui donne des résultats bien plus probants bien plus rapidement tel que le montre la figure 7.3.

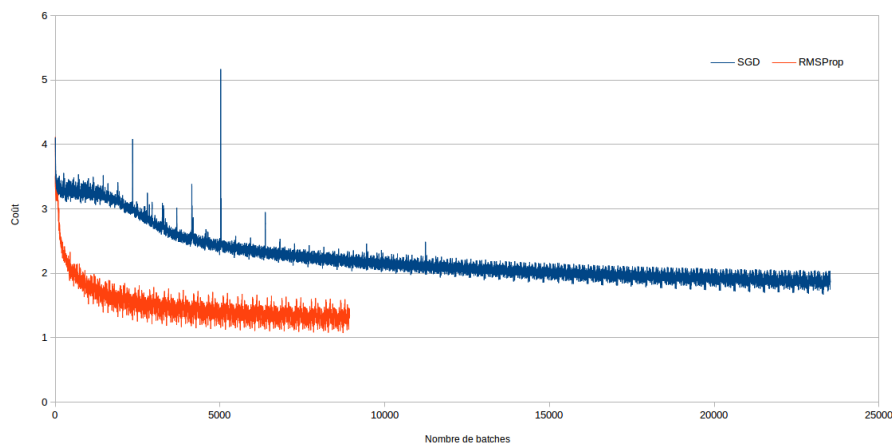


FIGURE 7.3 – Courbe du coût lors de l'entraînement selon l'algorithme d'optimisation utilisé

Voici un exemple de texte généré :

Third Servan :
Of many bald with him fire, read now ?

Second Murderer :
Out ! where he wal'd apt thou, myself!
O brother's maliss and trunks and Caubble subject.
Now i' the fill in thy noble devart wagains to argon me thy commanded ?

LADY ANNE :
Sir, af you have fellow's their eyes live ?

Ces résultats, bien que durs à évaluer sont jugés satisfaisants étant donné leur ressemblance avec l'anglais shakespearien et leur structure de pièce de théâtre. Il est intéressant de noter que certaines chaînes de caractères comme les noms des personnages suivis de deux points et d'un retour à la ligne est copiée exactement sur l'ensemble d'apprentissage, et dépendent même de la pièce qui a été apprise en dernier. Ici "Third Servant" et "Second Murderer" sont des personnages du Roi Lear.

7.3.2 tests complémentaires sur la forget gate

Le problème de la génération de texte a été l'occasion de questionner l'utilité de la forget gate. Cette partie de la cellule LSTM ne figurait que dans une partie des sources décrivant la composition de la cellule. La question est de savoir si retirer cette porte diminue notablement l'efficacité du réseau, et ce même en ajoutant des neurones à la couche cachée pour garder un nombre de neurone

constant par cellule.

Le tracé de la courbe de coût lors de l'apprentissage (figure 7.4 montre que la forget gate apporte effectivement un gain en performances.

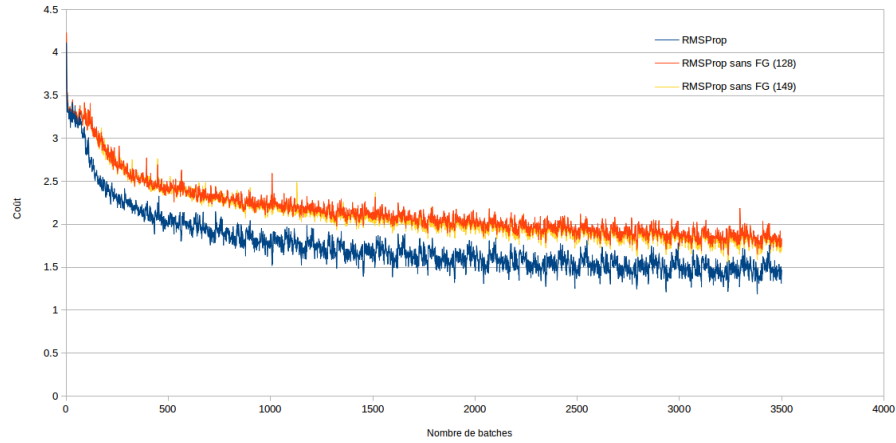


FIGURE 7.4 – Courbe du coût lors de l'entraînement avec avec ou sans forget gate

7.4 Conclusion

Une bonne utilisation des LSTM permet donc de générer un texte qu'on peut qualifier d'acceptable. Cependant mesurer la réussite de la génération est un problème à part entière qu'on ne traite pas ici, et les résultats sont donc difficilement quantifiables. De plus le texte produit semble en partie très proche du texte passé lors de la phase d'apprentissage.

Chapitre 8

Algorithmes d'optimisation

8.1 Motivation

Il existe une grande variété d'algorithmes permettant la minimisation de la fonction de coût en plus de la descente de gradient présentée précédemment. Ce chapitre aura pour but de présenter les plus populaires d'entre eux et d'en donner les principales caractéristiques. Nous les comparerons ensuite pour retenir les solutions les plus efficaces.

8.2 Descente de gradient et momentum

$$W_{t+1} = W_t - \eta \frac{\partial J}{\partial W} \quad (8.1)$$

Les poids sont changés en fonction de la dérivée partielle de la fonction de coût par rapport à W . Comme vu précédemment, η représente le learning rate. Si celui-ci est trop faible, la convergence sera lente, à l'inverse une valeur trop élevée entraînera la divergence des poids.

Cependant, le gradient de la fonction de coût change rapidement à chaque itération à cause de la diversité de chaque exemple d'apprentissage. On approchera alors le minimum par petits pas mais de façon désordonnée (en "zigzag"). En effet, si la direction moyenne obtenue sur plusieurs exemples d'apprentissage est la bonne ce n'est pas le cas si l'on ne prend en compte qu'un échantillon, celui-ci étant bruité. Cette approche consistant à calculer le gradient à chaque exemple est appelée *descente de gradient stochastique*.

Pour pallier ce problème, deux solutions sont possibles. On peut moyenner le gradient sur des batchs d'exemples de longueur fixée, au risque d'avoir de la redondance et de perdre en vitesse de calcul ou en espace mémoire. On parle alors de *descente de gradient par batchs*. La *descente de gradient par mini-batchs* consiste à prendre les avantages des deux méthodes, en trouvant un compromis sur la taille des batchs de sorte que le temps de calcul de chaque étape ne

soit pas trop long mais que les pas soient suffisamment précis pour assurer une convergence efficace.

Il est également possible d'introduire un terme de momentum. Cette opération consiste à prendre en compte l'information des itérations précédentes pour savoir dans quelle direction il est préférable de progresser. En introduisant à cet effet un nouvel hyperparamètre μ , on obtient alors les équations de mise à jour des poids ci-dessous.

$$\begin{aligned}v_{t+1} &= \mu v_t - \eta \frac{\partial J}{\partial W} \\ W_{t+1} &= W_t + v_t\end{aligned}\tag{8.2}$$

La trajectoire est alors plus lisse et la convergence plus efficace.

8.3 Algorithmes à learning rate adaptatif

8.3.1 Adagrad

L'objectif de cet algorithme est de ne plus avoir de learning rate constant mais d'adapter celui-ci en fonction de la valeur du gradient aux étapes précédentes. Cela se traduit dans l'équation de mise à jour des poids par une division par un terme g_t , somme des carrés des gradients des itérations précédentes. On obtient finalement les équations suivantes :

$$\begin{aligned}g_{t+1} &= g_t + \left(\frac{\partial J}{\partial W}\right)^2 \\ W_{t+1} &= W_t - \frac{\eta \frac{\partial J}{\partial W}}{\sqrt{g_{t+1}} + \epsilon}\end{aligned}\tag{8.3}$$

Le terme ϵ sert à initialiser l'algorithme et permet de s'assurer que le dénominateur ne sera jamais nul. On observe que si la norme du gradient est grande, le learning rate sera faible et vice versa. On n'a donc plus besoin de changer le learning rate manuellement, celui-ci est simplement fixé au départ puis adapté automatiquement. Cependant, comme le terme au dénominateur va augmenter au fur et à mesure, les termes g_t étant tous positifs, le learning rate aura tendance à devenir de plus en plus faible jusqu'à devenir infinitésimal. Les algorithmes suivants permettront de résoudre ce problème.

8.3.2 RMSProp

La principale différence avec Adagrad est que le terme g_t est calculé grâce à une moyenne mobile exponentielle : on utilise alors une pondération des termes qui décroît exponentiellement, ce qui donne plus d'importance aux valeurs récentes du gradient sans pour autant supprimer complètement les anciennes contributions.

$$\begin{aligned}
g_{t+1} &= \gamma g_t + (1 - \gamma) \left(\frac{\partial J}{\partial W} \right)^2 \\
W_{t+1} &= W_t - \frac{\eta \frac{\partial J}{\partial W}}{\sqrt{g_{t+1}} + \epsilon}
\end{aligned} \tag{8.4}$$

En raison du terme quadratique présent lors du calcul de g_t , ce dernier est appelé **moment d'ordre 2** de $\frac{\partial J}{\partial W}$. On trouve dans certaines implémentations l'ajout d'un moment d'ordre 1, qui se définit de façon très similaire par :

$$m_{t+1} = \gamma m_t + (1 - \gamma) \left(\frac{\partial J}{\partial W} \right) \tag{8.5}$$

Il est même possible de rajouter un terme de momentum comme pour la descente du gradient stochastique, les équations de mise à jour finales sont alors :

$$\begin{aligned}
v_{t+1} &= \mu v_t - \frac{\eta \frac{\partial J}{\partial W}}{\sqrt{g_{t+1} - m_{t+1}^2 + \epsilon}} \\
W_{t+1} &= W_t + v_{t+1}
\end{aligned} \tag{8.6}$$

8.3.3 Adadelta

Cet algorithme utilise également une moyenne mobile exponentielle pour le calcul de g_t , moment d'ordre 2 du gradient. Cependant, au lieu d'utiliser un hyperparamètre η fixe comme précédemment, il introduit à la place x_t , moment d'ordre 2 de v_t .

$$\begin{aligned}
g_{t+1} &= \gamma g_t + (1 - \gamma) \left(\frac{\partial J}{\partial W} \right)^2 \\
x_{t+1} &= \gamma x_t + (1 - \gamma) v_{t+1}^2
\end{aligned} \tag{8.7}$$

Les équations finales sont alors, si l'on se limite aux moments d'ordre 2 et sans momentum :

$$\begin{aligned}
v_{t+1} &= - \frac{\sqrt{x_t + \epsilon} \frac{\partial J}{\partial W}}{\sqrt{g_{t+1} + \epsilon}} \\
W_{t+1} &= W_t + v_{t+1}
\end{aligned} \tag{8.8}$$

8.3.4 Adam

Le dernier algorithme présenté consiste à reprendre les moments d'ordre 1 et 2 du gradient, m_t et g_t , mais en les faisant cette fois-ci décroître dans le temps à l'aide de coefficients γ_1 et $\gamma_2 \in [0,1[$

$$\hat{m}_{t+1} = \frac{m_{t+1}}{1 - \gamma_1^{t+1}} \hat{g}_{t+1} = \frac{g_{t+1}}{1 - \gamma_2^{t+1}} \tag{8.9}$$

Cela aura donc pour effet de réduire la taille des pas à l'approche du minimum et donc d'assurer une bonne convergence grâce à une adaptation efficace du learning rate. On obtient finalement l'équation de mise à jour des poids suivante :

$$W_{t+1} = W_t - \frac{\alpha \hat{m}_{t+1}}{\sqrt{\hat{g}_{t+1} + \epsilon}} \quad (8.10)$$

8.4 Comparaison des algorithmes

Nous avons eu l'occasion d'implémenter ces différents algorithmes et de les appliquer aux problèmes précédents. S'il n'y a pas à l'heure actuelle de consensus sur le meilleur d'entre eux, certaines solutions se sont révélées être bien plus efficaces que d'autres dans notre cas.

8.4.1 Application à l'apprentissage du XOR

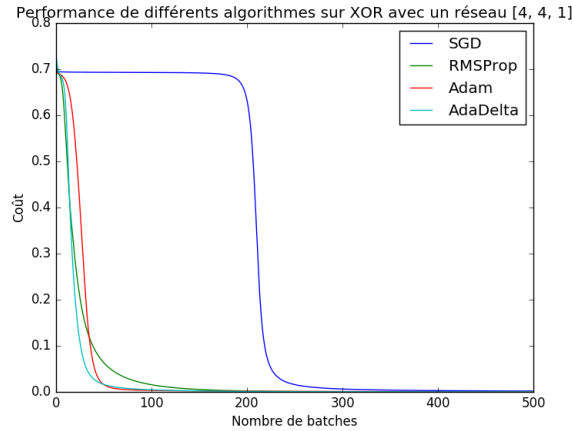


FIGURE 8.1 – Comparaison des fonctions de coût pour les différents algorithmes appliqués au XOR

On observe que les fonctions de coût pour les algorithmes à learning rate adaptatif ont des profils très similaires et ne présentent pas le pallier observé pour la descente de gradient stochastique. L'apprentissage est presque 4 fois plus rapide pour des temps de calcul du même ordre de grandeur, ce sont donc de meilleures implémentations en tout point pour ce type de problème.



FIGURE 8.2 – Comparaison des fonctions de coût pour les différents algorithmes appliqués à Shakespeare aux côtés de l’implémentation d’Andrej Karpathy

8.4.2 Application à l’apprentissage des oeuvres de Shakespeare

La différence entre la descente de gradient stochastique et les autres algorithmes est encore plus flagrante dans ce cas : la convergence est bien plus rapide, nécessitant bien moins d’exemples d’apprentissage. Nous avons donc par la suite totalement délaissé la descente de gradient classique au profit de RMSProp ou Adam, beaucoup plus performants.

8.5 Conclusion

Si la descente du gradient stochastique est l’algorithme le plus répandu dans la littérature, il existe de nombreuses autres méthodes d’optimisation pour la minimisation de la fonction de coût. Nous avons présenté les plus populaires d’entre elles dans ce chapitre et avons réalisé qu’elles permettaient d’avoir de bien meilleures performances pour les applications que nous avons considérées.

Il existe également une autre famille d’algorithmes du second ordre que nous n’avons pas eu le temps d’implémenter (méthode de Newton, BFGS, conjugate gradient...). Cependant, si nous avons trouvé des implémentations des algorithmes de ce chapitre dans des projets disponibles sur Internet, ce n’est pas le cas des méthodes du second ordre. Il est donc possible que ces dernières soient moins efficaces, que ce soit au niveau du temps de calcul ou de l’espace mémoire, bien que cela reste à vérifier.

Chapitre 9

Application : génération de musique

9.1 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 sous forme de texte. La génération se fait alors simplement en appliquant ce que l'on a 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 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.

9.2 Génération en abc

9.2.1 Présentation du format

Le langage abc permet de représenter un morceau de musique en format ASCII. Il s'est rapidement développé grâce à sa facilité de représentation et de

compatibilité entre les différents systèmes d'exploitation. De plus, ce langage ne nécessite pas l'utilisation d'un ordinateur pour être lu, contrairement au midi. Ainsi, les morceaux sont représentés de manière simple, efficace et compacte. Un fichier abc est composé de deux parties : un entête et le corps de la musique. L'entête permet de définir les caractéristiques globales du morceau :

- X : numéro du morceau dans le fichier
- T : titre
- M : valeur de la mesure
- L : valeur de la note de référence
- K : tonalité
- P : ordre dans lequel les sections sont jouées

Le corps correspond à l'écriture du morceau. Les notes sont représentées dans le système anglo-saxon : a=la ; b=si ; c=do ; d=ré ; e=mi ; f=fa ; g=sol. Le choix des majuscules/minuscules et des symboles "," et "" permettent de choisir les octaves utilisées. Les guillemets servent à écrire les accords. Les séparateurs sont utilisés pour les mesures et on peut spécifier les parties répétées. La figure 9.1 est l'exemple d'une jig en format abc qui sera utilisée lors de l'apprentissage.

```
X: 3
T:The American Dwarf
% Nottingham Music Database
S:FTB, via EF
M:6/8
K:D
P:A
A|"D" def fed|"G" BdB AFD| "D"DFA "G"B2 A|"Em" cee "A7" e2 A|
"D" def fed|"G" BdB "D"AFD|"D" DFA "G"B2 A|"A7" Add "D" d2:|
"B"e|"D"fga agf|"G" gab "A7"bag|"D"fga "D"agf|"Em" gfg "A7"e2 g|
"D"fga agf|"G"gab "A7"bag|"D" fga "A7"efg|"D" fdd d2 :|
```

FIGURE 9.1 – Exemple d'un fichier ABC

9.2.2 Principe

Étant donné que les morceaux de musique sont représentés sous forme de texte, on adopte un système d'apprentissage similaire à celui utilisé avec Shakespeare.

On utilise une cellule LSTM que l'on fait apprendre sur des séquences de longueur 50. Cela revient à déplier le réseau 50 fois à chaque apprentissage. Pour améliorer la rapidité et la convergence, on travaille avec des batchs de 50 séquences. L'algorithme d'optimisation choisi est RMSprop qui nous a donné les meilleurs résultats sur Shakespeare.

9.2.3 Résultats

Dans un premier temps, on réalise l'apprentissage sur un ensemble de 340 jigs trouvé sur une base de données de morceaux en abc. Le réseau arrive rapidement à apprendre la structure du format abc. De plus, les caractéristiques communes des jigs sont bien apprises ; mélodie, accompagnement. La figure 9.2 est un exemple de jig générée avec notre réseau au format abc. Il existe des convertisseurs abc/midi qui nous permettent de vérifier la cohérence du morceau généré. On voit ainsi que le réseau a très bien appris la structure des jigs, peut-être même trop bien, puisque certaines parties des morceaux générés sont quasiment identiques à des passages de la base de données d'apprentissage. Il arrive parfois que le texte généré présente quelques erreurs dans le format abc. Les convertisseurs sont en général capables de les corriger automatiquement voire d'ignorer les séquences absurdes.

```
X: 66
T:Pevim Dew's Music Database
% Nottingham Music Database
S:Trae, arr Phil Rowe
M:6/8
K:G
D|"D"F2A fed|c2A "F"A2A|"G"g2a "F#m"a2a|"Am"a3 -F#7"B2c|
"C"c2c "G7"G2G|"C"EFG EFG|"D7"AGF c2d|"C7"e2c g2e|
"F"afA fed|"Dm" B2A c3| ^g=c "E7"Bcd|
"F"M3 c3|"D7"AFA dcA|"D"d2f a2f|"A7"agf abg|"D"baf d2d|"C"ede "G"dcB|
"D7"d2B c2A|"Gm"d2g "A7ec|edc BcA|"D""D"FAF "A"AFA|"D"FFD d2::
c|"Dm"d2e e3|"D7"dcB "F"dcf|
"G"f2d "A7"=g^c2|"D"d3 "A7"gef|
"A"f2e "A7"AFA|"D"def f2f|"D"a2d "A7"edc|"D"fdd cdd::
/2c/2|"D"a2f "G"d2f|"D"f2f "A7"gaf|"D"a2d "E7"cBA|A6|"D"Adc d2C|
[1"B7"BADFFAFA|"D"d, D "Cm"=c2c: [G"G2B-d2e|"F"a2e "C7/g""A7"^fe||
"D" "Dm"def "G/b"gge "D"f2B|"G"gfg gfg|edc AGG|"G"Bdg efd|"Am"eAA
"D7"ABc|"G"B3 -B2B|Bdd G3:|
```

FIGURE 9.2 – Exemple d'une jig générée

Cette méthode de génération est assez efficace et facile à mettre en oeuvre. Cependant, elle est assez limitée puisque l'on n'utilise qu'une représentation textuelle de la musique. Pour des morceaux plus complexes, comme Mozart, le réseau a bien plus de mal à apprendre la structure de base du format. On essaiera par la suite de proposer des méthodes d'apprentissage et de représentation des morceaux plus proches de la musique.

9.3 Génération en midi

9.3.1 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 9.1 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				

TABLE 9.1 – Correspondance entre les notes et leur représentation en midi

9.3.2 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 9.3a et 9.3b représentent les matrices d’une jig et d’un morceau de Mozart.

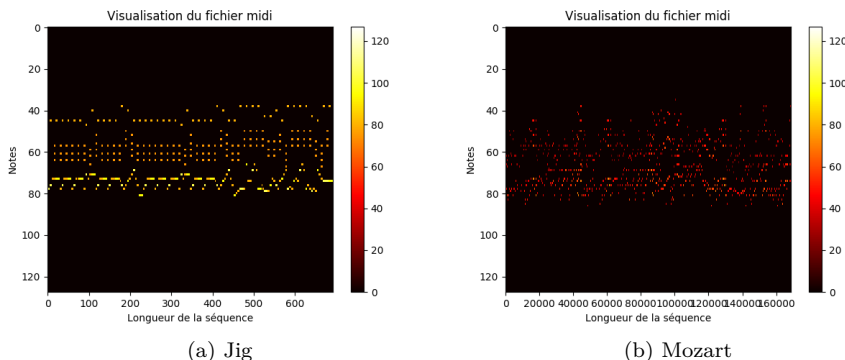


FIGURE 9.3 – Visualisation de fichiers midi

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.

9.3.3 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. Un fait des batchs de 50 séquences. On utilise RMSprop comme algorithme d’optimisation avec un learning rate de 0.95. La figure 9.4 permet de visualiser le fichier généré après le passage de 38 800 batchs lors de l’apprentissage.

Le résultat obtenu possède bien le style des jigs avec un accompagnement répétitif à l’arrière. Cet accompagnement, souvent commun aux jigs, est très ra-

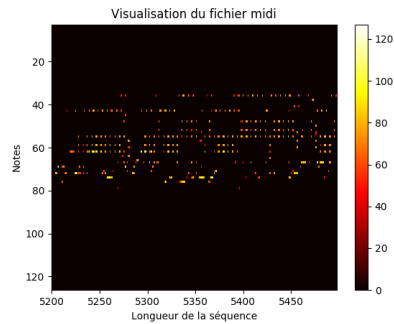


FIGURE 9.4 – Visualisation du fichier midi généré à partir des jigs après 38 800 batches

pidement 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 batches 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 9.5 représente le morceau généré après 79 500 passés.

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

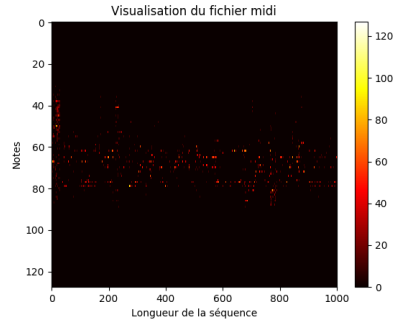


FIGURE 9.5 – Visualisation du fichier midi généré à partir des morceaux de Mozart après 79 500 batchs

9.4 Génération de partitions

9.4.1 Génération de notes

Architecture

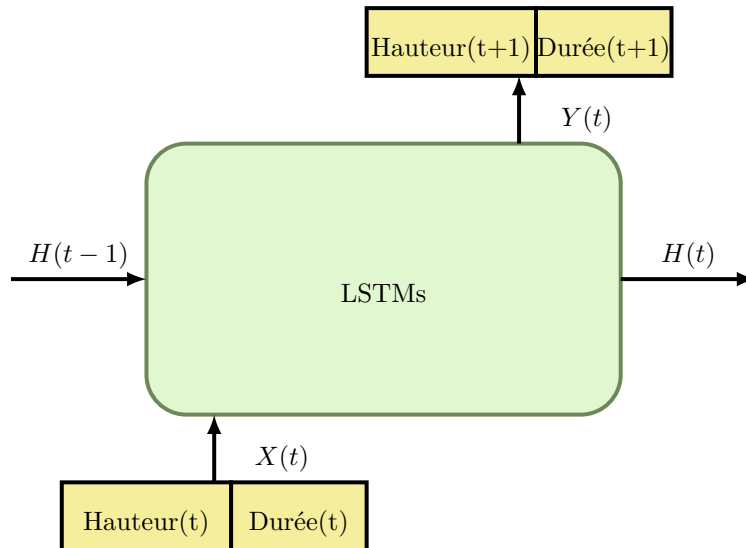


FIGURE 9.6 – Architecture du réseau utilisé pour la génération de notes.

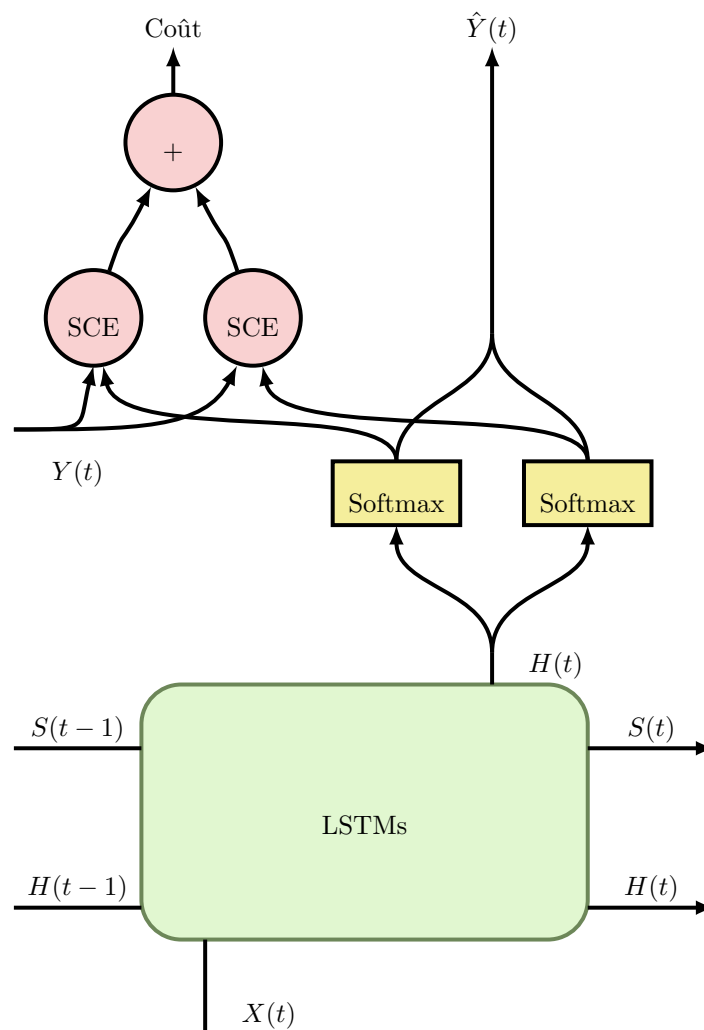


FIGURE 9.7 – Architecture plus détaillée du réseau utilisé pour la génération de notes.

Résultats

9.4.2 Génération de notes pour plusieurs instruments en série

Architecture

Résultats

9.4.3 Génération de notes pour plusieurs instruments en parallèle

Autoencodeur de mesures

90

Architecture

Résultats

9.5 Génération de signaux

9.5.1 Présentation du format



FIGURE 9.8 – Mélodie générée par le réseau *note-rnn*.

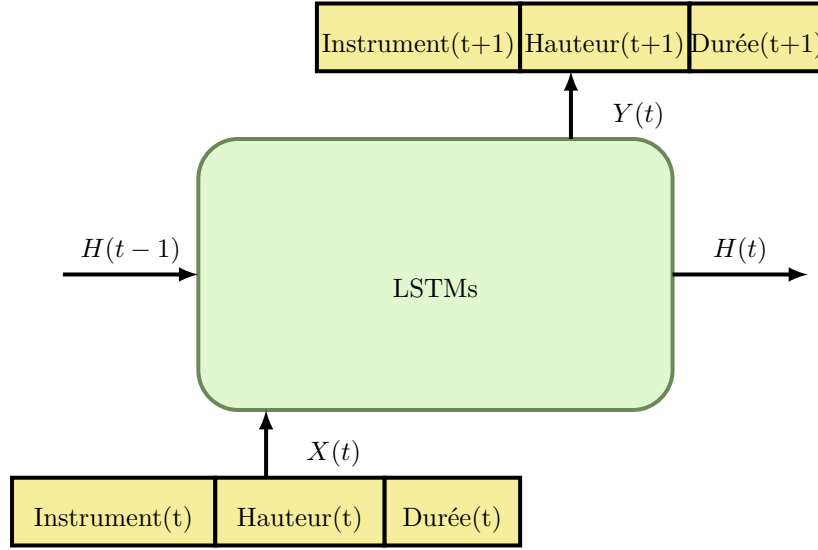


FIGURE 9.9 – Architecture du réseau utilisé pour la génération de notes pour plusieurs instruments en série.



FIGURE 9.10 – Partition générée par le réseau *series-rnn*.

de musique particulier et nous pourrions traiter directement des formats bien plus répandus comme le .mp3 ou le .wav. C'est précisément ce dernier que nous avons retenu, la librairie *scipy* de Python possédant des fonctions permettant d'appréhender très facilement ce format. L'entraînement se fera sur les coefficients de Fourier de ces signaux sonores, ceux-ci permettant de récupérer sous

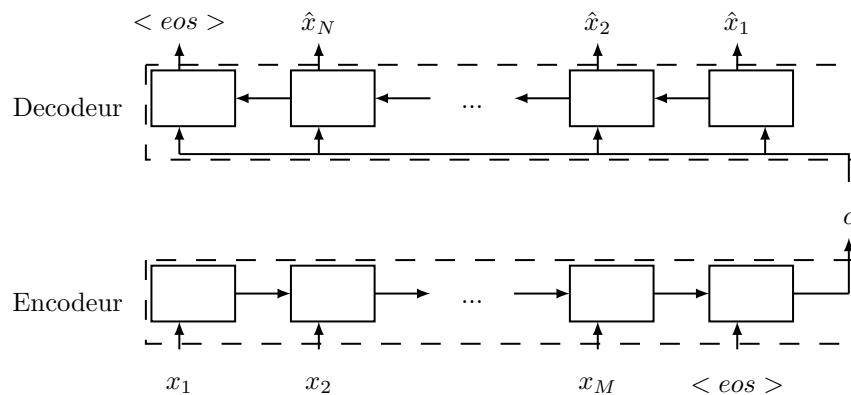


FIGURE 9.11 – Architecture d'un autoencodeur de séquences.

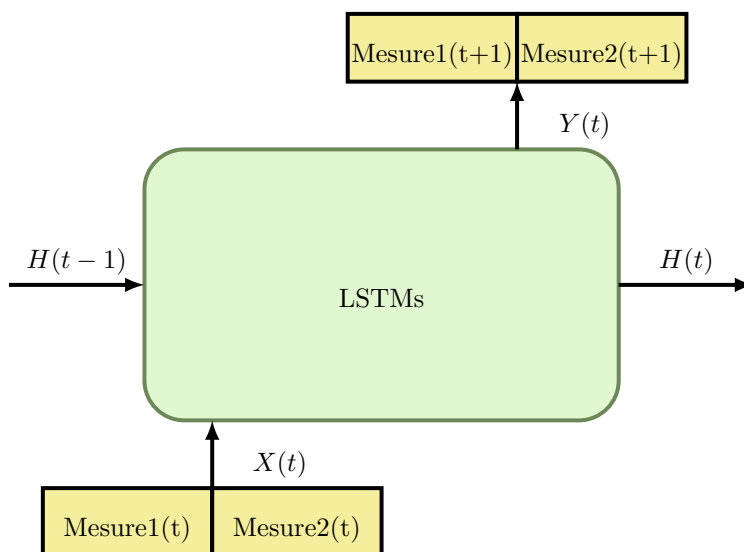


FIGURE 9.12 – Architecture du réseau utilisé pour la génération de notes pour plusieurs instruments en parallèle.

forme simple toute l'information relative aux morceaux.

9.5.2 Principe

La variété des morceaux .wav (durée, fréquence d'échantillonnage...) et les contraintes matérielles (espace mémoire, temps de calcul...) nous ont poussé à devoir réaliser un prétraitement des données dans notre programme. Celui-ci



FIGURE 9.13 – Partition générée par le réseau *measure-rnn*.

sous échantillonne tout d’abord les morceaux à une fréquence $F_e = 16kHz$, la plupart des morceaux étant à $44kHz$ par défaut. Il s’agit ensuite de les découper en morceaux de durée T de l’ordre de la dizaine de millisecondes (0, 25 dans notre implémentation). En effet, l’étape suivante consiste à appliquer la transformée de Fourier rapide sur chacun de ces morceaux élémentaires, il y aura donc $F_e \times T = 4000$ coefficients complexe au total, on voit donc tout l’intérêt du sous-échantillonnage et du choix d’une durée T peu élevée.

L’étape suivante est de séparer les contributions réelles et imaginaires de chaque coefficient, que l’on place alors dans des vecteurs de taille 8000. On possèdera finalement pour chaque morceau élémentaire un vecteur contenant ses coefficients de Fourier. Pour obtenir le morceau initial, il s’agira tout simplement de reformer les coefficients de Fourier avec leurs parties réelles et imaginaires, d’appliquer la transformée de Fourier rapide inverse et de concaténer les morceaux élémentaires.

Le réseau sera capable après entraînement de générer des morceaux de longueurs variables, décidée à l’avance par l’utilisateur, celle-ci étant simplement tributaire du nombre de vecteurs à générer.

9.5.3 Résultat

Le dataset est composé de morceaux de piano de durées variables. Le réseau est constitué de 2 cellules lstm avec un état caché de taille 512. L’algorithme d’optimisation utilisé est RMSProp, avec un learning rate de 0.95. Le coût, calculé avec la norme 2, décroît de plus de 1000 à 0.4 en 600 batchs. On soulignera l’importance de la normalisation des coefficients de Fourier pour éviter d’avoir des valeurs trop faibles lors du passage des valeurs dans les sigmoïdes. La génération donne cependant un morceau très bruité dont la représentation spectrale est visible ci-dessous :

Deux explications sont alors possibles. La première serait qu’il y a un problème dans le programme. La seconde consisterait à dire que le bruit est la structure la plus évidente à apprendre, étant présente en permanence. Nous avons été également limités par la taille de notre état caché : lorsque nous avons essayé de l’augmenter de façon significative, des erreurs mémoires ont été observées, la place occupée étant trop importante. Or plus l’état caché est de taille importante plus les cellules seront capables de saisir les spécificités des liens

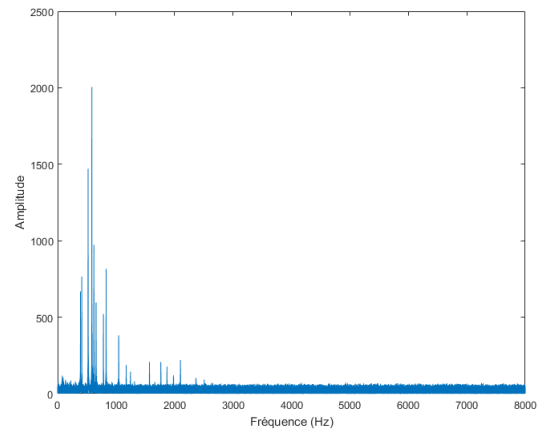


FIGURE 9.14 – Représentation spectrale du morceau généré

entre les différents instants. On peut donc raisonnablement penser que la limitation viendrait de là. Le manque de temps et de matériel nous a empêché de vérifier cette possibilité.