Supélec

PinaPL
Projet long : réseaux neuronaux

Auteurs :
Maxime Amossé
Julien Hemery
Hugo Hervieux
Sylvain Pascou

Référents :
Arpad Rimmel
Joanna Tomasik

11 juin 2017

Résumé

Ce projet a pout but d'étudier trois grandes familles de réseaux neuronaux et d'en distinguer les capacités : les réseaux simple, les réseaux récurrents, et les réseaux *LSTM* (*Long Short Term Memory*). Pour chacun d'entre eux nous étudions des algorithmes d'apprentissage afin d'obtenir des résultats aussi proches que souhaités de données sources, pour des expériences variées : réalisation d'opérations logiques, reconnaissance de chiffres manuscrits, apprentissage de grammaires simples.

Table des matières

1 Présentation du projet **1**

1.1 Objectifs . . . 1

1.2 Équipe . . . 1

1.3 Outils . . . 1

1.3.1 Langage de programmation . . . 1

1.3.2 Outil de versionnement . . . 2

1.3.3 Gestion de la bibliographie et références . . . 2

1.3.4 Écriture du présent rapport . . . 2

2 Réseau neuronal simple **3**

2.1 Théorie . . . 3

2.1.1 Le perceptron . . . 3

2.1.2 Le réseau . . . 4

2.1.3 Les réseaux à couche . . . 5

2.1.4 L'apprentissage . . . 5

2.1.5 La méthode des gradients . . . 5

2.1.6 La rétropropagation . . . 6

2.2 L'implémentation . . . 8

2.2.1 Le neurone . . . 8

2.2.2 Le réseau . . . 8

2.2.3 Améliorations apportées . . . 8

2.3 Résultats . . . 9

2.3.1 Le XOR . . . 9

2.3.2 Le MNIST . . . 11

2.3.2.1 Le classificateur linéaire . . . 11

2.3.2.2 Réseau sans couche cachées . . . 12

2.3.3 Réseau avec 1 couche cachée . . . 12

2.3.3.1 Réseau avec deux couches cachées . . . 12

3 Real-Time Recurrent Learning **13**

3.1 Théorie . . . 13

3.1.1 La grammaire de Reber . . . 13

3.1.2 Réseaux neuronal récurrents . . . 14

3.2 Algorithme RTRL . . . 15

3.3 Implémentation . . . 17

3.3.1 Structure du code . . . 17

3.3.2 Structure du réseau utilsée . . . 18

3.3.3 Exemple . . . 18

3.4 Résultats . . . 19

3.4.1 Grammaire de Reber simple . . . 19

3.4.2 Grammaire de Reber double . . . 19

4 Back Propagation Through Time **21**

4.1 Théorie . . . 21

4.1.1 Réseau BPTT . . . 21

4.2 Implémentation . . . 21

4.2.1 Structure de données . . . 21

4.2.1.1 Les poids . . . 22

4.2.1.2 La couche de neurones . . . 22

4.2.1.3 Le réseau . . . 22

4.3 Résultats . . . 22

4.3.1 Grammaire de Reber simple . . . 22

4.3.2 Grammaire de Reber symétrique . . . 22

5 Long Short Term Memory **24**

5.1 Théorie . . . 24

5.1.1 Cellule LSTM . . . 24

5.1.2 Propagation . . . 24

5.1.3 Algorithmes d'apprentissage . . . 25

5.2 Implémentation . . . 25

5.2.1 Structure de données . . . 26

5.2.1.1 Les poids LSTM . . . 26

5.2.1.2 La cellule LSTM 26
5.2.1.3 Le réseau LSTM 27
5.3 Résultats 27
5.3.1 Grammaire de Reber simple 28
5.3.2 Grammaire de Reber symétrique 28

Annexes **31**

Chapitre 1

Présentation du projet

1.1 Objectifs

Ce projet a été lancé le 23 septembre 2016 sur proposition de M^{me} Joana Tomasik et M. Arpad Rimmel. Le but est d'implémenter en un peu moins d'un an un algorithme d'apprentissage sur un réseau neuronal avec mémoire appelé LSTM [1], en étudiant d'autres structures de réseaux plus simples auparavant.

Un tel projet représente de fortes contraintes techniques (implémentation de la structure de réseau neuronal, gestion de la mémoire, performances, ...) mais aussi un défi théorique (élaboration des algorithmes, leur justification formelle, ...).

Pour arriver à ce résultat, l'année est découpée en différents grandes étapes et dans chacune nous devons découvrir et implémenter une spécificité des réseaux neuronaux. Tout d'abord nous voulons implémenter un réseau neuronal simple pour nous familiariser avec le concept des réseaux de neurones et mettre en place les outils adéquats pour les simuler. Puis, nous modifierons ces réseaux basiques pour y introduire des propriétés plus complexes, tels les réseaux récurrents (RTRL, BPTT). Enfin, nous adapterons le réseau et son algorithme pour qu'il corresponde à celui du LSTM.

1.2 Équipe

L'équipe du projet est composée de quatre élèves-ingénieurs de CentraleSupélec, cursus ingénieur Supélec : Maxime Amossé, Julien Hemery, Hugo Hervieux et Sylvain Pascou, encadrés par deux enseignants-chercheurs du même établissement, M^{me} Joana Tomasik et M. Arpad Rimmel.

1.3 Outils

1.3.1 Langage de programmation

Le langage de programmation choisi est le C++, pour sa rapidité d'exécution, sa grande versatilité, la richesse de sa communauté, ainsi que le aspects pédagogiques de son utilisation. Il est important de faire remarquer que ce langage est compilé et donc ne permet donc pas directement à l'utilisateur d'interagir simplement avec les variables durant l'exécution, ce qui complique significativement les phases de déboggage.

1. Long Short Term Memory

1.3.2 Outil de versionnement

Pour travailler en groupe de manière efficace nous avons utilisé le gestionnaire de version Git et un dépôt partagé sur la plateforme GitHub : `https://github.com/supelec-lstm/PinaPL`. Il nous a fallu une dizaine d'heures et quelques règles de bonne conduite pour éviter de créer des conflits et faciliter le travail de chacun.

Chaque développeur travaille donc sur la branche de la fonction qu'il implémente, puis il la fusionne dans la branche principale lorsqu'il a une version stable qui compile et se comporte correctement.

1.3.3 Gestion de la bibliographie et références

Zotero est un outil gratuit de gestion de bibliographie. Il permet à tous les membres du projet d'y enregistrer des références utiles, des articles papiers, mais aussi les fichiers PDF contenant lesdits articles. De plus, il nous permet d'exporter régulièrement la bibliographie sous un format standard pour l'inclure dans ce rapport.

1.3.4 Écriture du présent rapport

Enfin, pour rédiger ce rapport, nous avons utilisé LaTeX. Il s'agit d'un langage de création de documents axé autour du document scientifique ; il est reconnu par la communauté pour sa facilité d'utilisation et sa capacité à générer des documents propres et ordonnés.

Chapitre 2

Réseau neuronal simple

2.1 Théorie

2.1.1 Le perceptron

Le perceptron est le neurone le plus basique que l'on puisse trouver dans la littérature. Un perceptron est défini par :

— n entrées x_i
— 1 sortie y
— n poids w_i
— 1 biais θ
— 1 fonction de composition $g : \mathbb{R}^n \to \mathbb{R}$
— 1 fonction d'activation $f : \mathbb{R} \to \mathbb{R}$

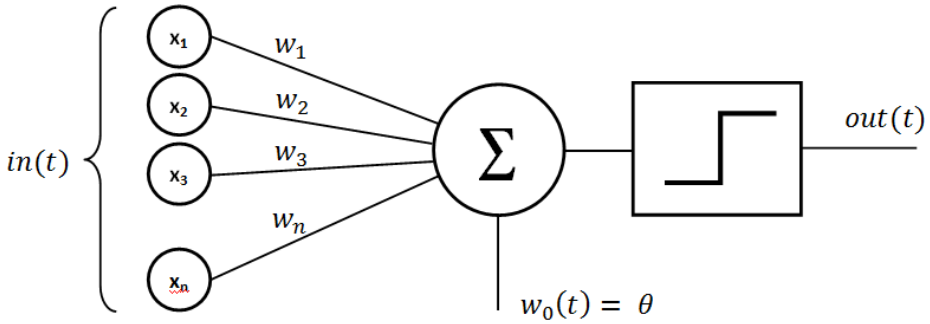

FIGURE 2.1 – Schéma d'un perceptron simple

Sur sa construction, le perceptron est fortement inspiré sur le neurone humain, sans pour autant en être une représentation réaliste. Le perceptron détermine avec ses entrées si il "active" ou non la sortie, c'est à dire s'il relaie le signal. Pour cela il rassemble toutes les données des entrées x_i à l'aide de la fonction de composition g. Le neurone ne donnant pas la même importance à chaque entrée, on les pondère préalablement par les poids notés w_i. Finalement, on décide du signal de sortie à l'aide de la fonction d'activation f. De base, le seuil d'activation est souvent centré en 0. Pour palier à ce problème, on prend également en entrée de la fonction d'activation un biais θ.

En résumé, on a :

$$y = f(g(x_1 w_1, x_2 w_2, \ldots, x_n w_n) + \theta)$$

Usuellement, la fonction d'activation est la somme :

$$y = f(\sum_{i=1}^{n} x_i w_i + \theta)$$

On remarque que le biais agit comme le poids d'une entrée du neurone qui serait toujours 1.

2.1.2 Le réseau

Une réseau de neurones permet de créer des fonctions de bien plus grande complexité qu'un simple neurone, permettant de résoudre des problèmes jusqu'alors inaccessibles à la machine. Il permet par exemple de faire des classifications sur le MNIST. Le MNIST est un problème classique où l'on a des images représentant des chiffres manuscrits et pour lequel l'on doit déterminer le chiffre représenté. C'est un problème d'une extrême simplicité pour un être humain, mais presque impossible à résoudre avec une programmation classique sans réseau neuronal. Ainsi, pour construire un réseau, chaque perceptron est mis en relation avec ses pairs (un perceptron prend en entrée les sorties d'autres perceptrons). On construit alors un graphe orienté dans lequel chaque sommet est un perceptron. On se limitera dans un premier temps au cas d'un réseau acyclique.

On peut définir plusieurs types de neurones dans un réseau :
— les neurones d'entrée
— les neurones cachés
— les neurones de sortie

Il faut donc autant de neurones d'entrée que de dimensions qu'a l'échantillon que l'on veut soumettre au réseau. Par exemple dans le cadre du MNIST on veut en entrée une image de dimension 28×28, on place donc 784 neurones d'entrées. En pratique, les neurones d'entrée sont des neurones fictifs ; ils sont présents pour faciliter la construction du réseau de neurone. En effet, ils ne sont soumis à aucun apprentissage et leur sortie est la même que leur entrée. Nous ne les considérerons pas dans la théorie qui suit.

Les neurones des couches cachées sont présents entre les neurones d'entrée et de sortie. Ils sont utiles uniquement pour le calcul de la sortie. Le nombre de couches et la taille des couches influent sur l'action du réseau. Un réseau à multiples couches cachées sera capable de traiter des problèmes beaucoup plus complexes qu'un réseau à simple couche cachée. Il est évident que cela augmente néanmoins la complexité des calculs et le temps d'exécution.

Les neurones de sortie sont ceux qui servent pour la classification de l'échantillon d'entrée. Si l'on souhaite classifier une entrée il faut le même nombre de neurones de sortie que de classes différentes possibles. Ainsi dans l'exemple du MNIST, le but est de déterminer un chiffre donné sur une image. Il y a donc 10 possibilités (les 10 chiffres). Il y a donc 10 neurones de sortie.

Par la suite, on appellera $\{x_i\}_{i\leq n}$ les entrées, $\{y_i\}_{i\leq m}$ les sorties des neurones $\{y_i\}_{m+1-M\leq i\leq m}$ les sorties des neurones de sorties, $\{\sigma_i\}_{i\leq m}$ les fonctions d'activations et $\{\theta_i\}_{i\leq m}$ les biais.

On définit enfin $\{F_i\}_{i\leq m}$ tel que $j \in E_i$ si et seulement si la sortie du neurone j est reliée au neurone i. On peut ainsi numéroter les poids : $\{w_{ij}\}_{i\leq m, j\in F_i}$ le poids associé à l'entrée reliant le neurone j au neurone i.

D'après ce qui précède, on obtient $\forall i \in [1, m]$:

$$y_i = \sigma_i(\sum_{j\in F_i} y_j w_{ij} + \theta_i)$$

2.1.3 Les réseaux à couche

Il existe un type de réseau simplifié très utilisé. On partitionne l'ensemble des neurones en K ensembles. On chacun de ces ensembles une "couche". De plus, chaque neurone d'une couche a comme entrées l'ensemble (ou partie) des sorties des neurones de la couche précédente. On notera $\alpha_j^{(k)}$ l'élément j de la couche k de α. On notera également N_k le nombre de neuroneq à la couche k. On a donc pour tout $k > 1$:

$$y_i^{(k)} = \sigma_i^{(k)}(\sum_{j=0}^{N_{k-1}} y_j^{(k-1)} w_{ij}^{(k)} + \theta_i^{(k)})$$

On remarque que l'on peut simplifier la notation en considérant la formule ci-dessus avec une approche vectorielle :

$$\overline{y^{(k)}} = w^{(k)} \times y^{(k-1)} + \theta^{(k)}$$
$$y^{(k)} = \sigma^{(k)}(\overline{y^{(k)}})$$

On peut montrer que tout réseau acyclique peut se ramener à un réseau à couches dont certains poids sont imposés comme nuls. Nous prenons donc l'hypothèse que le réseau est un réseau à K couches, que la première couche est l'ensemble des neurones d'entrée et la dernière couche l'ensemble des neurones de sortie, sans perte de généralité.

2.1.4 L'apprentissage

L'efficacité d'un réseau de neurones se mesure à la qualité de sa classification . Celle-ci dépend des poids qui sont attribués à chacune de ses entrées. Il faut donc déterminer la bonne combinaison de poids qui permettra au réseau de simuler la fonction voulue. Le nombre de poids présents dans un réseau augmente très rapidement et il devient complexe d'estimer cette bonne combinaison. Pour cela, on procède à une phase apprentissage : on utilise un échantillon de données dont on connaît le résultat pour construire un réseau avec les bon poids . On part ainsi d'un réseau avec des poids aléatoires, choisis dans un intervalle restreint et centré en zéro, et on les modifie en prenant en compte les erreurs entre les valeurs obtenues et les valeurs théoriques. Dans la suite, on s'intéressera à toute la démarche nécessaire pour arriver à cette modification de poids.

On notera $\{x_i\}_{i \leq n}$ et $\{Y_i\}_{i \leq M}$ les entrées et sorties des échantillons.

Pour déterminer les modifications à effectuer, on calcule la sortie du réseau de neurones à un échantillon de test donné et on mesure l'erreur. Pour cela, on choisira une fonction qui mesurera la différence entre le vecteur de sortie et le vecteur des sorties théoriques. Classiquement, on utilise la méthode des moindres carrés E_m ou bien une fonction softmax à laquelle on rajoute une entropie croisée E_s :

$$E_m = \sum_{j=1}^{M} \frac{(Y_j - y_j^{(K)})^2}{2}$$
$$E_s = \sum_{j=1}^{M} Y_j \log\left(\frac{e^{y_j^{(K)}}}{\sum e^{y_i^{(K)}}}\right)$$

2.1.5 La méthode des gradients

On veut donc minimiser E en modifiant les w_{ij}. Le problème ici est que l'on a une connaissance limitée de E en fonction des w_{ij} car on ne dispose des valeurs théoriques de sortie que pour un nombre fini de valeurs. Or

les méthodes de minimisation de fonctions reposent souvent sur une connaissance continue de ce que l'on veut optimiser. La seule méthode viable est la méthode des gradients.

On a une fonction f, appelée fonction de coût, que l'on veut minimiser par rapport à un facteur x. On crée alors une suite (x_n) telle que $x_{n+1} = x_n - \dfrac{\partial f}{\partial x}(x_n)$. L'idée est se déplacer sur le potentiel de f grâce à son gradient. Avec cette méthode, on peut calculer facilement la suite (x_n) car il suffit d'évaluer le gradient en un point et non plus en un nombre continument infini.

Cependant, cette méthode est imprécise et il arrive qu'elle converge vers un minimum local. En pratique, l'ajout de neurones va augmenter le nombre de dimensions du gradient et donc permettre de limiter le nombre de minima locaux.

2.1.6 La rétropropagation

D'après ce qui précède, l'objectif est donc d'évaluer pour tout $w_{ij}^{(k)}$: $\dfrac{\partial E}{\partial w_{ij}^{(k)}}$.

$$\begin{aligned}
\frac{\partial E}{\partial w_{ij}^{(k)}} &= \sum_{l=1}^{M} \frac{\partial y_l^{(K)}}{\partial w_{ij}^{(k)}} \times \frac{\partial E}{\partial y_l} \\
&= \left\langle \frac{\partial y^{(K)}}{\partial w_{ij}^{(k)}}, J_E \right\rangle \\
&= \left\langle \frac{\partial \overline{y^{(K)}}}{\partial w_{ij}^{(k)}} \odot \sigma'(\overline{y^{(K)}}), J_E \right\rangle \\
&= \left\langle \frac{\partial \overline{y^{(K)}}}{\partial w_{ij}^{(k)}}, \sigma'(\overline{y^{(K)}}) \odot J_E \right\rangle
\end{aligned}$$

Avec J_E la jacobienne de E au point considéré. Calculons maintenant $\dfrac{\partial \overline{y^{(l)}}}{\partial w_{ij}^{(k)}}$. Tout d'abord, remarquons que si $l = k$ alors on a simplement :

$$\frac{\partial \overline{y^{(k)}}}{\partial w_{ij}^{(k)}} = E_{ij} y^{(k-1)} = y_j^{(k-1)} e_i$$

Ceci vient du fait que le réseau étant acyclique, $y^{(l)}$ ne dépend pas de $w_{ij}^{(k)}$ pour $l < k$. De même, pour $l > k$, on obtient :

$$\begin{aligned}
\frac{\partial \overline{y^{(l)}}}{\partial w_{ij}^{(k)}} &= w^{(l)} \times \frac{\partial y^{(l-1)}}{\partial w_{ij}^{(k)}} \\
&= w^{(l)} \times \left(\sigma(\overline{y^{(l-1)}}) \odot \frac{\partial \overline{y^{(l-1)}}}{\partial w_{ij}^{(k)}} \right)
\end{aligned}$$

Donc en supposant qu'il existe un vecteur v tel que $\dfrac{\partial E}{\partial w_{ij}^{(k)}} = \left\langle \dfrac{\partial \overline{y^{(l)}}}{\partial w_{ij}^{(k)}}, v \right\rangle$:

$$\begin{aligned}\frac{\partial E}{\partial w_{ij}^{(k)}} &= \left\langle w^{(l)} \times \left(\sigma(\overline{y^{(l-1)}}) \odot \frac{\partial \overline{y^{(l-1)}}}{\partial w_{ij}^{(k)}} \right), v \right\rangle \\ &= \left\langle \sigma(\overline{y^{(l-1)}}) \odot \frac{\partial \overline{y^{(l-1)}}}{\partial w_{ij}^{(k)}}, {}^t w^{(l)} \times v \right\rangle \\ &= \left\langle \frac{\partial \overline{y^{(l-1)}}}{\partial w_{ij}^{(k)}}, \sigma(\overline{y^{(l-1)}}) \odot ({}^t w^{(l)} \times v) \right\rangle\end{aligned}$$

Donc il existe $u = \sigma(\overline{y^{(l-1)}}) \odot ({}^t w^{(l)} \times v)$ tel que $\dfrac{\partial E}{\partial w_{ij}^{(k)}} = \left\langle \dfrac{\partial \overline{y^{(l-1)}}}{\partial w_{ij}^{(k)}}, u \right\rangle$

Donc par récurrence, pour tout $l \geq k$ il existe $\delta y^{(l)}$ tel que :

$$\frac{\partial E}{\partial w_{ij}^{(k)}} = \left\langle \frac{\partial \overline{y^{(l)}}}{\partial w_{ij}^{(k)}}, \delta y^{(l)} \right\rangle$$

On remarquera que $\delta y^{(l)}$ ne dépend pas de $w_{ij}^{(k)}$. On appellera $\delta y_j^{(l)}$ le gradient du neurone j de la couche l. Ce gradient vérifie la relation de récurrence suivante d'après ce qui précède :

$$\begin{cases} \delta y^{(K)} = & \sigma'(\overline{y^{(K)}}) \odot J_E \\ \delta y^{(k)} = & \sigma(\overline{y^{(k)}}) \odot ({}^t w^{(k+1)} \times \delta y^{(k+1)}) \end{cases}$$

On a alors :

$$\begin{aligned}\frac{\partial E}{\partial w_{ij}^{(k)}} &= \left\langle \frac{\partial \overline{y^{(k)}}}{\partial w_{ij}^{(k)}}, \delta y^{(k)} \right\rangle \\ &= \left\langle y_j^{(k-1)} e_i, \delta y^{(k)} \right\rangle \\ &= y_j^{(k-1)} \times \delta y_i^{(k)}\end{aligned}$$

Donc :

$$\Delta w_{ij}^{(k)} = \lambda \times y_j^{(k-1)} \times \delta y_i^{(k)}$$

$$\Delta w^{(k)} = \lambda \times (\delta y^{(k)} \times {}^t y^{(k-1)})$$

On obtient donc un algorithme d'apprentissage qui se fait en deux temps : tout d'abord le calcul du gradient qui se fait récursivement, puis le calcul de la différence de poids à appliquer. On remarque dans ce cas que l'on propage le gradient de la fin du réseau vers le début. C'est ceci qui donne le nom à la méthode employée : la rétropropagation.

2.2 L'implémentation

2.2.1 Le neurone

Le neurone est une classe, il a pour attributs :
— le nombre de ses entrées (`int`).
— les valeurs de ses entrées (`vector<double>`).
— les poids qu'il leur attribue (`vector<double>`).
— son biais (`int`).
— sa fonction de composition (`compositionFunction`).
— sa fonction d'activation (`activationFunction`).

Il dispose des méthodes suivantes :
— `description()` : indique l'état du neurone.
— `reset()` : remet à zero ses entrées et sa sortie.
— les getters et les setters pour les poids, le nombre d'entrées, les fonctions de composition...
— `calculateOutput()` : calcul la sortie du neurone en fonction de ses entrées.
— `getActivationDerivative` : calcul de la dérivée de la fonction d'activation au point observé.
— `getCompositionDerivative` : idem pour la fonction de composition.

2.2.2 Le réseau

Le réseau est aussi implémenté en tant que classe. Un réseau dispose :
— d'un nom (`string`).
— d'une date de création (`string`).
— de ses neurones (`vector<Neuron>`).
— de la liste de ses neurones d'entrée (`vector<unsigned long>`).
— de la liste de ses neurones de sortie (`vector<unsigned long>`).
— de la matrice des liens entre neurones (`vector<vector<double>>`).
— de son facteur d'apprentissage (`unsigned long`).
— de ses valeurs en entrée (`vector<double>`).
— de la matrice des sorties des neurones (`vector<double>`).

Il dispose des méthodes nécessaires à la propagation du signal ainsi qu'à sa rétropropagation.

2.2.3 Améliorations apportées

Nous avons ensuite amélioré le code pour diminuer le temps de calcul et clarfier la structure. Les éléments à améliorer sont :
— La structure orientée objet
— Le single-threading
— calcul à chaque pas des sorties de chaque neurone

Nous nous sommes progressivement débarrassés de la structure d'objet du neurone en effectuant les conversions suivantes :

structure objet	nouvelle structure
neurones.poids + matrice des poids + matrice des relations	matrice des poids
neuron.activationFunction	vecteur de fonctions d'activation
neuron.compositionFunction	on ne considère plus que la somme
neuron.inputs/output	vecteur des entrées/sorties de tout le réseau
neuron.bias	vecteur des biais de chaque neurone du réseau
dérivée de la fonction de composition	vaut 1

De plus, nous avons déterminé en amont les neurones voisins qui nécessitaient un rafraîchissement de leur sortie. Cela permet de ne pas calculer à chaque itération la sortie de tous les neurones du réseau. Lors de la création du réseau est construite une liste de vecteurs des neurones dont il faut évaluer la sortie au tour i.

2.3 Résultats

Après une longue période de déboggage, nous avons obtenu des résultats satisfaisants.

2.3.1 Le XOR

Les premiers tests simples à réaliser avec un réseau neuronal sont les opérations logiques : en effet, de telles opérations prennent deux arguments en entrées A et B, de valeur 0 ou 1, et donnent en sortie un valeur S égale à 0 ou à 1. Cela se représente aisément avec un réseau de neurones, avec deux neurones d'entrée A et B, et un neurone de sortie S ; les neurones A et B sont tous deux liés deux neurones C et D, qui forment une couche cachée, et ceux-ci sont reliés à S.

Si les opérations logiques rudimentaires telles que FAUX : $(A, B) \mapsto 0$, A : $(A, B) \mapsto A$ ou OR : $(A, B) \mapsto (A \vee B)$ ne posent aucun problème, le cas de XOR : $(A, B) \mapsto (A \wedge \neg B) \vee (\neg A \wedge B)$ est plus intéressant car il requiert l'utilisation d'une couche cachée ainsi que des biais.

On réalise l'apprentissage à l'aide de 100000 entrée successives, où les valeurs de A et B sont choisies aléatoirement. La propagation, rétro-propagation sont effectués à chaque étape, en revanche la modification des poids est faite toute les 100 entrées. Le facteur d'apprentissage est fixé à 0.5 et les poids initialisés par des valeurs aléatoires entre -1 et 1 ; on choisit la sigmoïde comme fonction d'activation pour chacun des neurones et un coût quadratique pour l'évaluation de l'erreur.

Dans ces conditions, l'apprentissage est quasiment instantané. On réalise ensuite les quatre tests correspondant aux quatre cas possibles pour les valeurs de $(A, B) \in \{0, 1\}^2$, et on regarde la valeur obtenue et le coût final c défini par

$$c = \sum_{(A,B)\in\{0,1\}^2} (f(A, B) - (A \text{ XOR } B))^2$$

où $f(A, B)$ est le résultat donné par le réseau de neurones.

On obtient généralement les résultats suivants, à 10^{-2} près :

$$\begin{aligned} f(0, 0) &= 0.01 \\ f(1, 0) &= 0.99 \\ f(0, 1) &= 0.99 \\ f(1, 1) &= 0.01 \\ \text{et } c &= 0.02 \end{aligned}$$

On obtient donc des résultats très correct. Comme la fonction f du réseau de neurone n'est pas définie exclusivement sur $\{0, 1\}^2$, mais sur $\mathbb{R}^2$, on peut visualiser l'évolution de $f(A, B)$ pour $(A, B) \in [-0.5, 1.5]^2$. Cela donne le graphique de la figure 2.2, où les valeur usuelles de (A, B) sont indiquées par des points. On peut voir que le réseau à identifié une bande dans laquelle $f(A, B)$ vaut 1, et vaut 0 ailleurs.

Cependant, dans certains cas le réseau apprend mal le XOR. Cela se traduit par la convergence vers un minimum local de la fonction de coût dans lequel le réseau va se bloquer et dont il ne pourra sortir. De tels erreurs arrivent environ dans 1 cas sur 3, et sont dues à l'initialisation aléatoire des poids. Des résultats typiques d'un mauvais apprentissage sont présentés en figure 2.3 et 2.4.

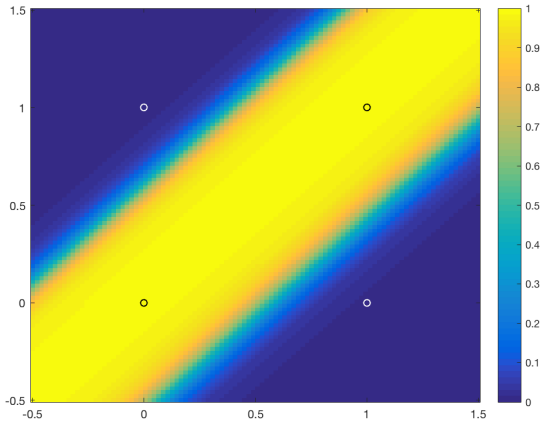

FIGURE 2.2 – Résultats corrects de XOR

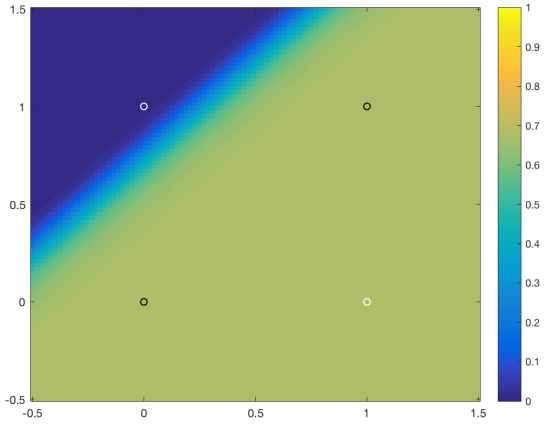

FIGURE 2.3 – Résultat erroné de XOR ($c = 0.83$)

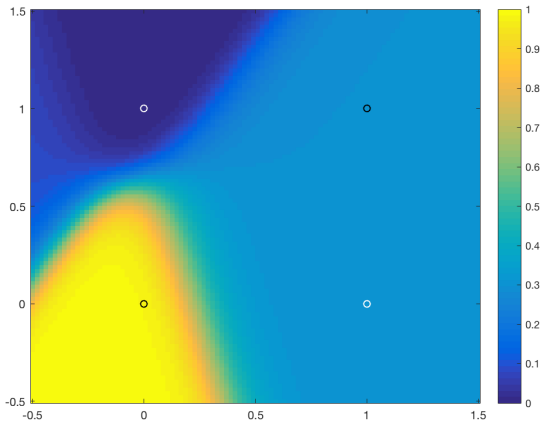

FIGURE 2.4 – Autre résultat erroné de XOR ($c = 0.76$)

Enfin, on peut aussi introduire une erreur dont les données d'apprentissage. Cela revient à inverser la sortie souhaitée de temps en temps ; ainsi si on veut apprendre à l'étape i que 0 XOR 1 vaut 1, introduire une erreur reviendra à fixer à l'étape i $A = 0$, $B = 1$ et $S = 0$ (au lieu de $S = 1$ donc). En faisant cela de façon régulière, on peut mesurer la résilience du réseau et sa capacité à apprendre malgré des données d'apprentissage de moindre

qualité.

Par exemple, en introduisant une erreur dans 20% des cas, on obtient les résultats suivants :

$$f(0,0) = 0.11$$
$$f(1,0) = 0.85$$
$$f(0,1) = 0.87$$
$$f(1,1) = 0.17$$
$$\text{et } c = 0.28$$

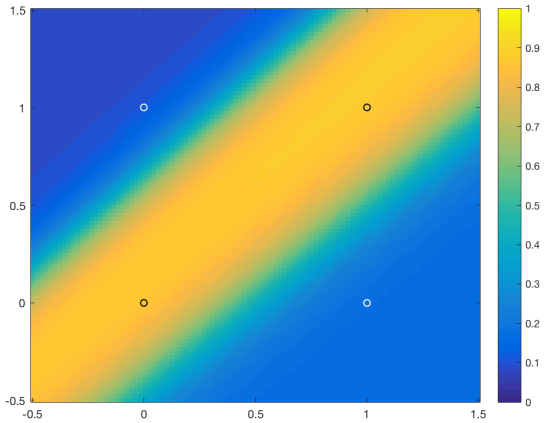

FIGURE 2.5 – Résultat avec des données erronées

Malgré la proportion d'erreurs qui peut sembler importante, on voit que le réseau arrive tout de même à séparer le bruit des bonnes valeurs. Ainsi, il est encore capable de distinguer la bande où f est égale à 1, même si la différence entre la bande et le reste est amoindrie. Cependant, le taux de mauvais apprentissages est aussi plus important, et si la proportion d'erreur est trop grande le réseau n'est plus capable d'appendre correctement.

2.3.2 Le MNIST

Les données du problème du MNIST sont réparties en deux fichiers :
— Le set d'apprentissage qui contient 60000 entrées (images de 784 pixels des chiffres manuscrits suivis des données des chiffres représentés)
— Le set de test qui contient 10000 entrées (différentes de celle du set d'apprentissage)

Les valeurs des entrées sont stockées dans les fichiers entre 0 et 255, nous les avons centrées et normalisées (entre -0.5 et 0.5).

Tous les résultats présentés par la suite sont établis en soumettant au réseau après apprentissage les 10000 entrées du set de test et en comparant la sortie attendue et la sortie obtenue. On obtient le résultat du calcul du réseau en prenant la sortie du réseau avec la valeur maximale.

2.3.2.1 Le classificateur linéaire

En créant un réseau neuronal sans bias ni couche cachées on obtient un classificateur linéaire. Les 10 neurones de sortie sont reliés chacun aux 784 neurones d'entrées. Les poids sont initialisés aléatoirement entre -1 et 1 selon une loi uniforme. La fonction d'activation est une siogmoïde, la composition est une somme pondérée, et la

fonction de coût est l'écart quadratique. Le facteur d'apprentissage λ est fixé à 0.3 en accord avec la littérature sur le sujet.

Les résultats d'un tel réseau sont forts intéressants car permettent après un apprentissage stochastique des 60000 échantillons de test d'obtenir un pourcentage moyen d'erreur de XX %. (Moyenne effectuée sur 100 réalisations).

2.3.2.2 Réseau sans couche cachées

2.3.3 Réseau avec 1 couche cachée

2.3.3.1 Réseau avec deux couches cachées

Chapitre 3

Real-Time Recurrent Learning

À partir de cette section, on va chercher à reconnaître une chaîne de caractère en temps réel : c'est à dire que en donnant un ou plusieurs caractères, on doit être capable de prédire la fin de la chaine, ou tout du moins un fin possible, selon une grammaire prédéfinie. Pour résoudre ce genre de problème, on utilise des réseaux neuronaux récurrents, qui vont permettre de se souvenir des états précédents pour pouvoir prédire efficacement les états suivants ; on ajoute donc une notion de mémoire au réseau. Dans un premier temps, nous allons nous intéresser aux réseaux récurrents simples, et à deux algorithmes qui vont permettre d'ajuster les poids du réseau : Real-Time Recurrent Learning (RTRL) et BackPropagation Through Time (BPTT).

3.1 Théorie

Dans la suite, nous allons nous intéresser à la grammaire de Reber, qui nous fournira des échantillons de test pour la suite.

3.1.1 La grammaire de Reber

Une grammaire de Reber est un langage défini par l'automate déterministe cyclique suivant :

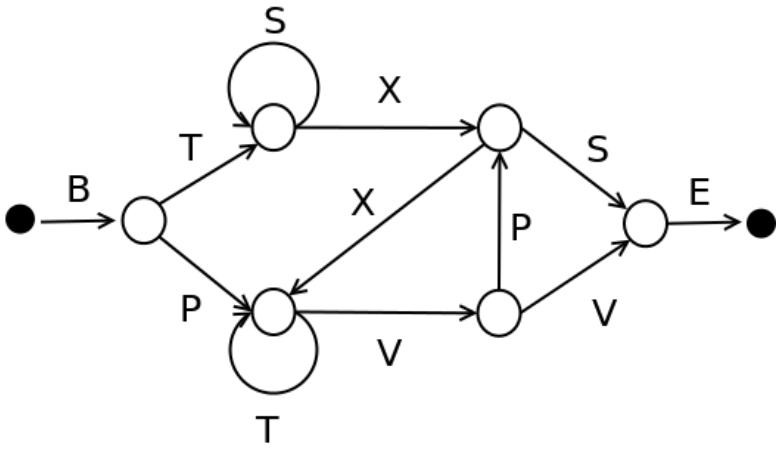

FIGURE 3.1 – Grammaire de Reber simple

On considère une probabilité uniforme de choisir l'état suivant parmi les états suivants possibles. La lettre B et la lettre E sont des lettres indiquant simplement le début et la fin de la chaîne, elles n'ont pas d'intérêt

propre pour la grammaire. Les autres lettres présentes sur les arêtes peuvent varier, mais elles doivent respecter les règles suivantes :

— Chaque lettre doit apparaitre exactement deux fois ;
— On ne peut pas obtenir deux lettres consécutives en passant par des états différents.

L'intérêt de la grammaire de Reber est qu'il s'agit un automate simple qui ne nécessite que la mémoire de la dernière et de l'avant dernière lettre pour déterminer les lettres suivantes possibles. En effet, d'après la dernière règle, connaitre les deux dernières lettres impose l'état actuel dans l'automate. En outre, chaque lettre apparaissant deux fois, la connaissance seule de la dernière lettre ne suffit pas à prédire la suivante correctement. On remarque que l'on peut résoudre ce problème avec un perceptron classique si on donne en entrée du perceptron les deux dernières lettres du mot. Cependant cette approche reste un bricolage propre à ce cas, et n'est pas du tout généralisable. On va donc chercher ici à résoudre ce problème en passant en entrée du réseau chaque lettre l'une après l'autre, à l'aide de la mémoire du réseau. Nous pourront ensuite utiliser des modèles de grammaire plus complexes, comme la grammaire de Reber symmétrique.

Le problème de la grammaire symmétrique est un problème similaire à la grammaire simple. L'automate le représentant est :

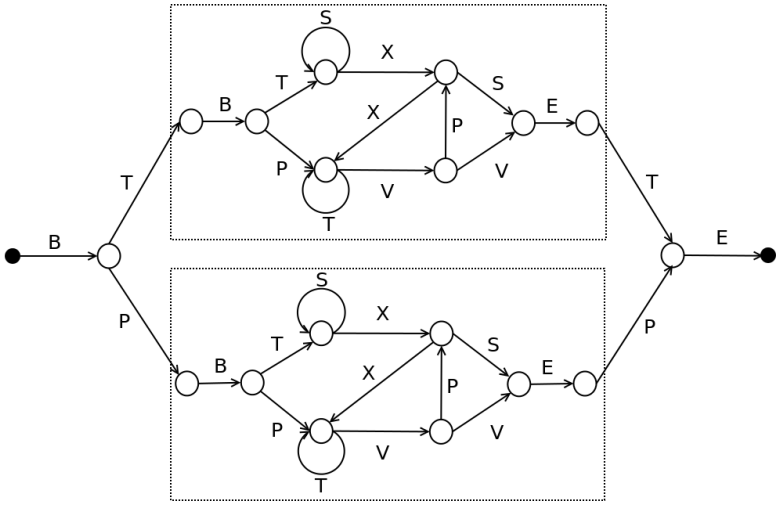

FIGURE 3.2 – Grammaire de Reber symétrique

On remarque qu'il est constitué de deux grammaires de Reber simples qui sont reliées uniquement en entrée et en sortie. La difficulté de ce problème est qu'il faut mémoriser, et se rappeler de la première valeur pour déterminer la dernière. Dans ce cas, une mémoire infinie, au tout du moins dépassant le court terme, est nécessaire pour se souvenir de la première entrée. Il est ici impensable d'utiliser de la même façon un réseau de perceptron classique. On a alors besoin d'un réseau récurrent, dont la sortie à un instant t va dépendre de la sortie à un instant $t-1$.

3.1.2 Réseaux neuronal récurrents

Le principe d'un réseau neuronal récurrent est d'utiliser le résultat obtenue par la sortie précédente à l'entrée du calcul suivant. On aura donc en appelant $x(t)$ la donnée en entrée à l'instant t et $y(t)$ la sortie associé. On a alors $y(t) = f(x(t), y(t-1))$. En appelant α un paramètre de la fonction f. Le but est de faire un apprentissage sur α pour minimiser à chaque temps t l'erreur $E(t)$ entre la sortie théorique et la sortie pratique. De la même façon que précédemment, on va donc calculer $\dfrac{\partial E(t)}{\partial \alpha}$ pour utiliser la méthode du gradient : Nous allons nous intéresser au cas où f est de la forme :

$$f(x(t), y(t-1)) = \sigma\left(Wx(t) + Ry(t-1) + b\right)$$

Avec W, R qui sont des matrices de poids, b un biais et σ une fonction d'activation. On reconnait donc d'après ce qui précède la formule d'un réseau perceptron à 0 couche caché et "fully connected". On pose $y(t) = W \times x(t) + R \times y(t-1) + b$. Les paramètres du systèmes sont donc les éléments de la matrice On obtient donc pour l'apprentissage :

$$\frac{\partial E(t)}{\partial \alpha} = \left\langle \frac{\partial y(t)}{\partial \alpha}, J_E \right\rangle$$

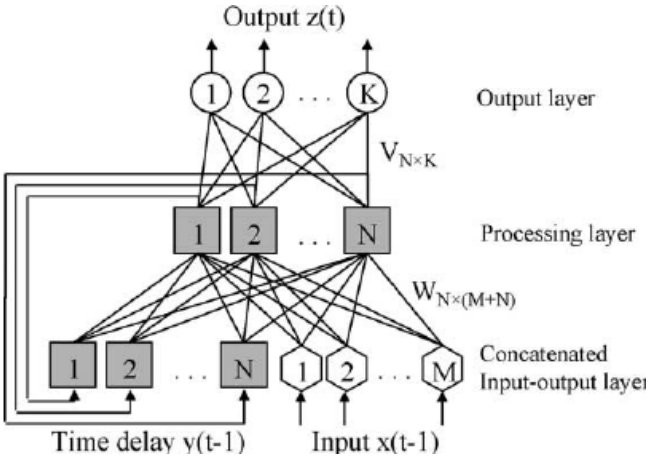

FIGURE 3.3 – Réseau récurrent RTRL

Cependant ici, à cause de la nature bouclée du réseau, l'algorithme de rétro-propagation simple vu ci-dessus ne peut s'appliquer. Nous allons donc commencer par étudier l'algorithme Real-Time Reculent Learning, ou RTRL, qui permet de calculer les erreurs en temps réel.

3.2 Algorithme RTRL

Le premier algorithme est aussi le plus simple mathématiquement, mais comme on va le voir cela va de pair avec une complexité temporelle et spatiale élevée.

Afin de prendre en compte les biais simplement, on considerera une entrée supplémentaire de valeur toujours égale à 1, et connectée à tous les neurones du réseau. Ainsi, le vecteur des biais b sera les poids correspondant à cette entrée unitaire.

On suppose donc que l'on dispose de m entrées (dont une unitaire pour les biais), prenant les valeurs $\boldsymbol{x}(t)$ à l'instant t. Ces entrées sont reliées à n neurones, dont les sorties sont notées $\boldsymbol{y}(t)$ à l'instant t. Par commodité, on note $\boldsymbol{z}(t)$ la concaténation de $\boldsymbol{x}(t)$ et de $\boldsymbol{y}(t)$. Soit de plus les ensembles I l'ensemble des entrées et U l'ensemble des neurones, tels que

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

Ainsi, on peut définir une unique matrice de poids $\boldsymbol{W} = (w_{i,j})$, de taille $n \times (m+n)$. On considérera que le temps t est discret.

On pose ensuite

$$s_k(t) = \sum_{i \in I \cup U} w_{k,i} z_i(t)$$

le vecteur des entrées pondérées à l'instant t, et on obtient par construction

$$y_k(t+1) = f_k(s_k(t)) \qquad \forall k \in U$$

le vecteur des sorties à l'instant $t+1$. La fonction f_k, pour tout $k \in U$, est la fonction d'activation du neurone k. Nous prendrons ici la fonction sigmoïde comme fonction d'activation : pour tout $k \in U$ et $t > 0$, $f_k(t) = \frac{1}{1+e^{-t}}$.

Nous avons maintenant défini l'évolution du réseau en fonction du temps grâce aux équation ci-dessus. On va maintenant chercher à estimer, puis propager l'erreur dans le réseau.

Soit $T(t)$ l'ensemble dont on connaît *a priori* la valeur à l'instant t ; il s'agit simplement des neurones de sortie dont la valeur est imposée par l'apprentissage. On remarque que cet ensemble des neurones de sortie peut varier au cours du temps, bien que cela ne soit pas nécessaire dans le cadre de l'apprentissage des grammaires de Reber. On note de plus, pour tout $kinT(t)$, la valeur attendue $d_k(t)$ en sortie du neurone k à l'instant t.

On définit alors l'erreur $e_k(t)$, pour tout neurone k, par :

$$e_k(t) = \begin{cases} d_k(t) - y_k(t) & \text{si } k \in T(t) \\ 0 & \text{sinon} \end{cases}$$

En prenant pour fonction de coût l'erreur quadratique moyenne, la quantité à minimiser au court du temps s'écrit

$$J(t) = \tfrac{1}{2} \sum_{k \in U} e_k^2(t)$$

et l'objectif est de minimiser la quantité totale

$$J_{\text{total}} = \sum_t J(t)$$

On procède pour cela par une descente de gradient sur les poids $\boldsymbol{W}$, en les ajustant de temps en temps d'une valeur $\Delta w_{i,j}$.

Pour cela, on s'intéresse à la variation de l'erreur due au poids $w_{i,j}$ à l'instant t, à un facteur d'apprentissage $\alpha > 0$ près :

$$\Delta w_{i,j}(t) = -\alpha \frac{\partial J(t)}{\partial w_{i,j}}$$

Et bien sûr, $\Delta w_{i,j} = \sum_t \Delta w_{i,j}(t)$.

Or on a

$$-\frac{\partial J(t)}{\partial w_{i,j}} = \sum_k \left(e_k(t) \times \frac{\partial y_k(t)}{\partial w_{i,j}} \right)$$

et à l'aide des équations qui régissent la propagation il vient que, pour tout $k \in U$, $i \in U$ et $j \in I \cup U$,

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

où $\delta_{i,k}$ est la fonction delta de Kronecker (égale à 1 si $i = k$, et nulle sinon), et, si f est la fonction sigmoïde, $f'_k(s_k(t)) = y_k(t) \times (1 - y_k(t))$.

Cette équation récurrente en $\left(\frac{\partial y_l(t)}{\partial w_{i,j}} \right)_t$ va nous permettre de tout calculer par la suite, puisque l'on sait de plus qu'en $t = 0$, cette quantité $\frac{\partial y_l(0)}{\partial w_{i,j}}$ est nulle.

En posant

$$p_{i,j}^k(t) = \frac{\partial y_l(t)}{\partial w_{i,j}} \qquad \text{et } p_{i,j}^k(0) = 0$$

l'équation ci-dessus se réécrit

$$p_{i,j}^k(t+1) = f_k'(s_k(t)) \times \sum_{l \in U} \left(w_{k,l} p_{i,j}^l(t) + \delta_{i,k} z_j(t) \right)$$

et dans le cas de la fonction sigmoïde

$$p_{i,j}^k(t+1) = y_k(t)(1 - y_k(t)) \times \sum_{l \in U} \left(w_{k,l} p_{i,j}^l(t) + \delta_{i,k} z_j(t) \right)$$

Enfin, les variations de poids se notent alors

$$\Delta w_{i,j}(t) = -\alpha \sum_{k \in U} e_k(t) p_{i,j}^k(t)$$

On voit donc que l'on arrive à calculer $\Delta w_{i,j}$ itérativement pour chaque pas de temps t. L'implémentation de l'algorithme va donc consister à calculer itérativement toutes les valeurs utiles afin de déterminer les $p_{i,j}^k(t)$.

La complexité de cet algorithme est en $O(n^3(n+m))$, à le calcul de $p_{i,j}^k(t)$ se faisant pour tout i, j, k et demandant une somme de n éléments. En espace, la complexité est de $O(n^2(n+m))$.

3.3 Implémentation

3.3.1 Structure du code

Une première tentative d'implémentation a consisté à adapter le code du programme précédent, utilisé pour les réseaux neuronaux simples, aux réseaux neuronaux récurrents. Cependant, après de longues semaines de déboggage sans résultats probants, nous avons décider de commencer un nouveau programme le plus simple possible, intitulé *SimpleRTRL*.

À cette fin, nous avons utilisé la bibliothèque C++ Eigen, qui contient toutes les structures et algorithmes nécessaires à l'algèbre linéaire. Le diagramme de classes UML est présenté ci-dessous :

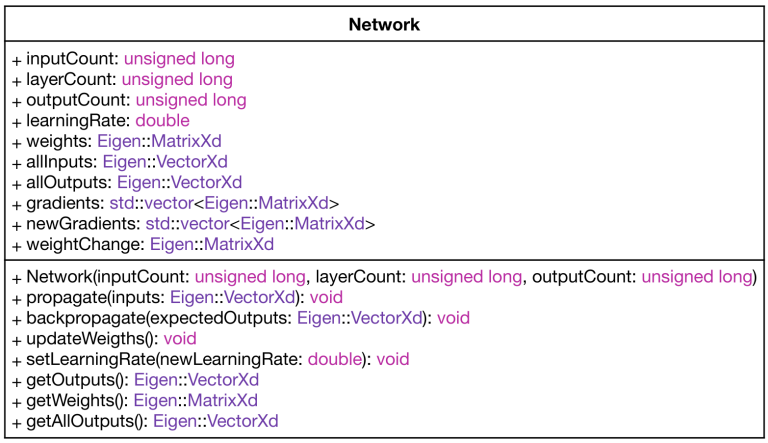

FIGURE 3.4 – Diagramme de classes de *SimpleRTRL*

On voit donc qu'il y a une unique classe, `network`, qui effectuera tous les calculs, selon les étapes suivantes :
— on initialise un réseau avec `inputCount` entrées et `layerCount` neurones dont `outputCount` seront considérés comme des sorties
— on propage un vecteur d'entrées `inputs` dans le réseau avec la méthode `propagate`
— on rétro-propage l'erreur avec la méthode `backpropagate` à l'aide du vecteur des sorties attendues `expectedOutputs`
— après avoir propagé et rétro-proagé plusieurs fois, on applique les changements de poids avec `updateWeights`

À tout moment, on peut récupérer les sorties avec `getOuputs`, ou plus généralement voir l'état du réseau avec les accesseurs. Lorsque les changements de poids sont effectués, toutes les données intermédiaires sont effacées et on revient à l'instant initial $t = 0$.

Les poids du réseau sont initialisés avec une valeur entre -0.1 et 0.1. L'ensemble des sorties des neurones sont stockés dans une unique liste, dont les $n' = |T(t)|$ premiers sont considérés comme des neurones de sortie.

3.3.2 Structure du réseau utilsée

Afin de faire de réaliser l'apprentissage, on utilise une liste de 2,4 millions de mots générés à partir de la grammaire de Reber simple et autant générés à partir de la grammaire double. Pour les tests, on utilise une liste disctincte de 1 million de mots pour chaque grammaire.

Afin de tester l'apprentissage des grammaires de Reber par les réseaux récurrents et l'algorithme RTRL, nous avons choisi une certaine structure de réseau. Celle ci consiste à mettre une entrée par lettre possible (en l'occurence, 7 : `B`, `T`, `P`, `S`, `X`, `V` et `E`), et autant de sorties, avec au total 30 neurones (dont 7 considérés comme des neurones de sortie). Ensuite, à chaque instant, l'entrée correspondant à la lettre courante est mise à 1, et les autres à 0. La propagation est réalisée, puis la rétro-propagation, et enfin, lorsqu'on a fait passer le mot entier de cette façon, on applique les variations de poids, avec un facteur d'apprentissage de 0.1.

Pour valider cet apprentissage, on procède de même mais sans rétro-propagation ni apprentissage. Les lettres indiquées en sortie par le réseau sont celles dont les sorties correspondantes ont une valeur supérieur à un seuil, fixé à 0.3.

Des fonctions utilitaires permettent de réaliser l'apprentissage puis de le tester dans le cadre des grammaires de Reber. Elles enchaînent l'apprentissage de 50 mots différents par un test du réseau sur 1000 mots, le tout répété 2000 fois. Pour chaque test, on vérifie que la lettre suivante du mot est bien parmi les lettres suivantes possibles selon le réseau (c'est-à-dire une sortie dont la valeur est supérieure au seuil de 0.3). Le score final de réussite est le nombre de lettres prédites correctement sur le nombre total de lettres, et on suit l'évolution de ce score au fur et à mesure des apprentissages.

Dans le cas des grammaires doubles, on fait de même en se restreignant uniquement à vérifier la prédiction de l'avant-dernière lettre, celle que l'on ne peut trouver qu'en se souvenant de la seconde lettre du mot.

3.3.3 Exemple

Voici un exemple pour illustrer le processus ci-dessus. Supposons que l'on veuille apprendre le mot `BPVPSE`.

On propage donc le vecteur d'entrée $(1, 0, 0, 0, 0, 0, 0)$ (qui correspond à 1 pour l'entrée de la lettre `B` et 0 pour les autres). En sortie, on obtient le vecteur $\boldsymbol{y}(0)$, que l'on fixe ici par exemple à $\boldsymbol{y}(0) = (0.1, 0.2, 0.5, 0.2, 0.6, 0.1, 0.1)$ (ce qui correspond à 0.1 pour `B`, 0.2 pour `T`, 0.5 pour `P`, 0.2 pour `S`, 0.6 pour `X`, 0.1 pour `V` et 0.1 pour `E`). Dans ce cas, le réseau à bien prédit la lettre `P` comme étant possible (car $0.5 > 0.3$), mais il n'a pas prédit correctement l'autre possibilité : il estime que `T` n'est pas possible (alors que c'est le cas), mais que `X` est possible (ce qui n'est pas le cas).

Pour la rétro-propagation, on lui donne le vecteur des sorties désirées $\boldsymbol{d}(0) = (0, 0, 1, 0, 0, 0, 0)$, puisque la lettre suivante est un `P`. Le réseau pourra apprendre que `T` est une autre lettre possible après un `B` initial avec d'autre mots. Une fois la rétro-propagation réalisée, on met en entrée $y(1) = d(0)$, et ainsi de suite.

Une fois que l'on a propagé et rétro-propagé chaque lettre du mot, on effectue les changements de poids avec les variations calculées pour chaque lettre.

Lors des tests, si la lettre suivante est effectivement prédite, comme c'est le cas ci-dessus, on comptabilise un point. En revanche, si on avait une sortie de 0.1 associée à la lettre `P`, le point ne serait pas comptabilisé.

3.4 Résultats

Étudions maintenant les résultats obtenus pour l'apprentissage des grammaires de Reber simples et doubles.

3.4.1 Grammaire de Reber simple

On réalise 100 apprentissages successifs et indépendants de 100000 mots chacun, en testant l'apprentissage tout les 50 mots, dans les conditions décrites ci-dessus. On obtient la courbe du taux de prédiction de lettre suivante ci-dessous, avec en bleu la moyenne sur les 100 essais et en gris le taux de confiance à 95%. L'échelle des ordonnées est en pourcentages.

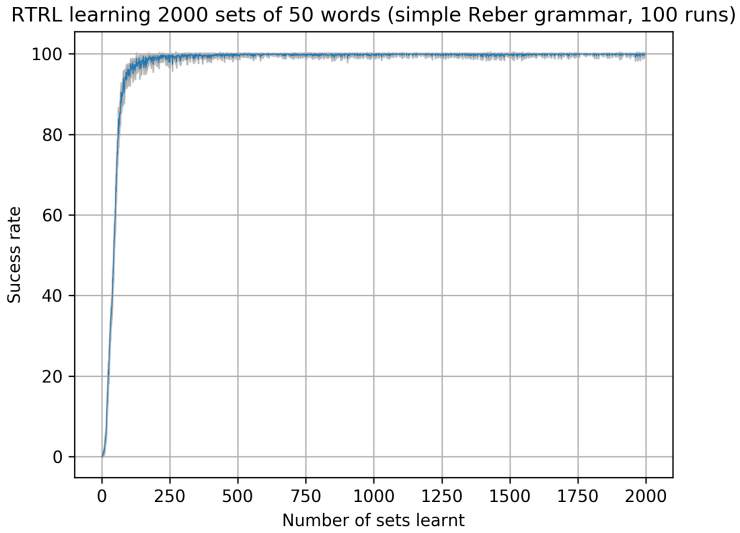

FIGURE 3.5 – Taux de succès au cours de l'apprentissage de la grammaire simple

On remarque que l'apprentissage est de bonne qualité, et que le réseau arrive sans problème deviner la suite du mots dès le $500^{\text{ème}}$ test, c'est-à-dire après avoir appris 25000 mots.

La durée d'un essai complet est d'environ 30 minutes sur un cœur Intel® Xeon® cadencé à 2.40 GHz.

3.4.2 Grammaire de Reber double

De même que pour la grammaire simple, on réalise 100 apprentissages successifs et indépendants de 100000 mots, en testant l'apprentissage tout les 50 mots. On obtient la courbe du taux de prédiction de l'avant-dernière lettre en figure 3.6, avec en bleu la moyenne sur les 100 essais et en gris le taux de confiance à 95%. L'échelle des ordonnées est en pourcentages.

La durée d'un essai complet est d'environ 40 minutes sur un cœur Intel® Xeon® cadencé à 2.40 GHz.

Ici, l'apprentissage est de moindre qualité, on voit immédiatement que le réseau a plus de difficultés pour apprendre cette grammaire, et qu'il n'y arrive asymptotiquement pas. Il y a tout d'abord un plateau jusqu'au $750^{\text{ème}}$ test environ, où le taux de succès stagne autour de 25%, puis il augmente progressivement jusqu'à atteindre 70%. Là, la marge de confiance à 95% reste élevée ; en effet de nombreux essais présentent des comportements erratiques où le taux de réussite redescend subitement vers 25%, avant de remonter. La figure 3.7 présente un tel essai.

De tels phénomènes sont observés en moyenne sur un essai sur 3 (soit environ 33% des cas). Le délais avant de remonter peut être plus ou moins long, mais cette remontée se produit systématiquement. Il arrive que ce

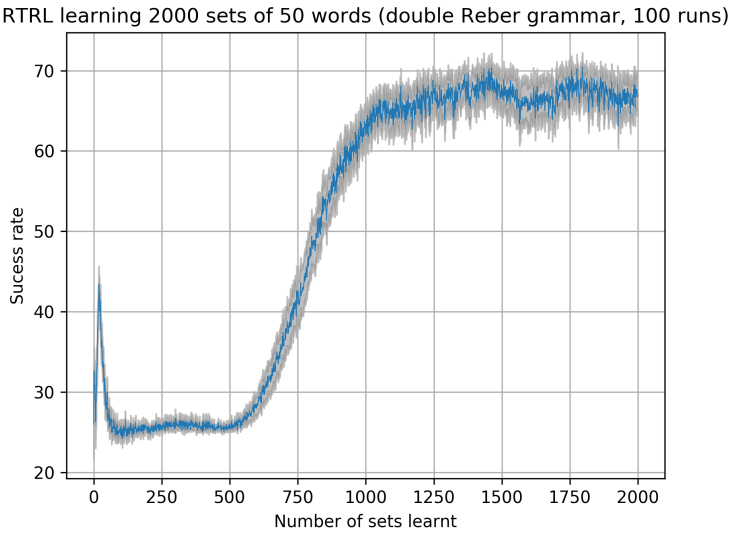

FIGURE 3.6 – Taux de succès au cours de l'apprentissage de la grammaire double

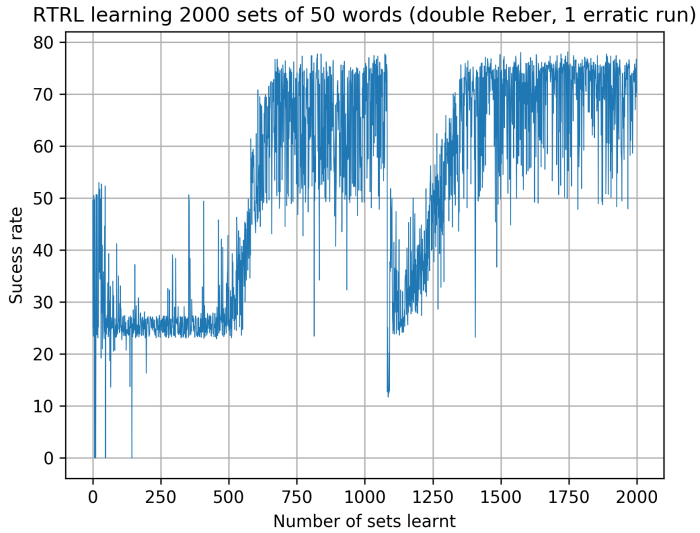

FIGURE 3.7 – Un essai erratique lors de l'apprentissage de la grammaire double

soit beaucoup plus court, comme plus long que sur l'exemple présenté ici.

De plus, on peut voir que le taux de succès est de moins en moins stable lorsqu'il augmente. Outre la complexité temporelle élevée de l'algorithme RTRL, on peut conclure ici qu'il n'est pas adapté à l'apprentissage de la grammaire de Reber symétrique. Dans la suite, nous allons essayer de mettre en place un autre algorithme, BackPropagation Through Time (BPTT), sur la même structure de réseau, puis une structure tout à fait différente, afin d'améliorer ces résultats.

Chapitre 4

Back Propagation Through Time

L'algorithme BPTT appliqué à un réseau neuronal récurrent a pour particularité de déplier le temps dans l'espace ; par exemple, pour apprendre un mot de cinq caractères, on va créer cinq réseaux non-récurrents qui représentent chaqun un "temps", soit une lettre de la séquence.

Cet algorithme a pour avantage, par rapport à celui RTRL, d'avoir une complexité temporelle inférieure.

4.1 Théorie

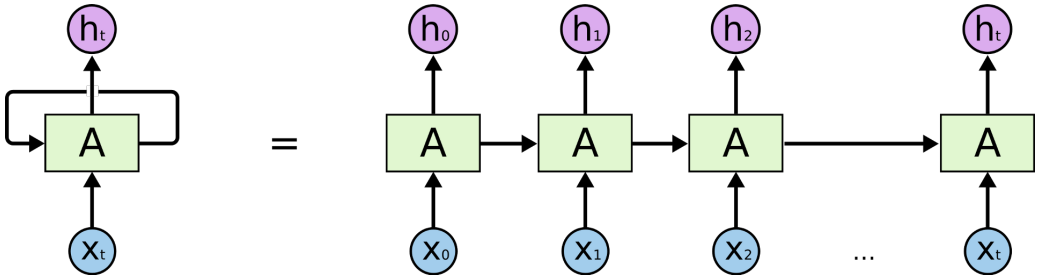

FIGURE 4.1 – Dépliement du temps dans l'espace pour BPTT

4.1.1 Réseau BPTT

4.2 Implémentation

L'implémenation est effectuée en C++ via la librairie de calcul matriciel Eigen3. Toutes les matrices sont des objets de type Eigen : :MatrixXd (matrice de double) et les vecteurs des objets de type Eigen : :VectorXd.

L'aléatoire utilisé est celui natif en C et C++ : rand. La génération de la graine se fait à partir du temps à la milliseconde pour éviter une initialisation déterministe dans le cas de l'execution de plusieurs runs consécutifs. Pour cela la librairie 'sys/time.h' est utilisée, avec un appel propre aux systèmes UNIX.

4.2.1 Structure de données

Le code se décompose en plusieurs éléments :

— Les poids
— Une couche de neurones fully-connected
— Le réseau de couches dépliées dans le temps

4.2.1.1 Les poids

Enfin, les méthodes de l'objet Poids sont le constructeur et l'application des variations de poids (qui remet par la même occasion à 0 les delta-poids).

4.2.1.2 La couche de neurones

4.2.1.3 Le réseau

4.3 Résultats

Ci-dessous des représentations de l'apprentissage au cours du temps du réseau sur des grammaires de Reber, simple et double.
En abscisse, le nombre de mots appris, en ordonnée le taux de réussite, testé à intervalles réguliers sur un échantillon de données de test choisies aléatoirement dans l'ensemble de test.
La zone grise correspond à l'intervalle de confiance à 95% sur le nombre d'exécutions précisé.
Le réseau utilisé est composé d'une couche cachée de 30 neurones, avec un learning rate de 0.1.

4.3.1 Grammaire de Reber simple

Pour une grammaire de Reber simple, la réussite est déterminée par la prédiction correcte de toutes les transitions des mots testés.

4.3.2 Grammaire de Reber symétrique

Pour une grammaire de Reber symétrique, réussite est déterminée par la prédiction correcte de la première et la dernière transition des mots.

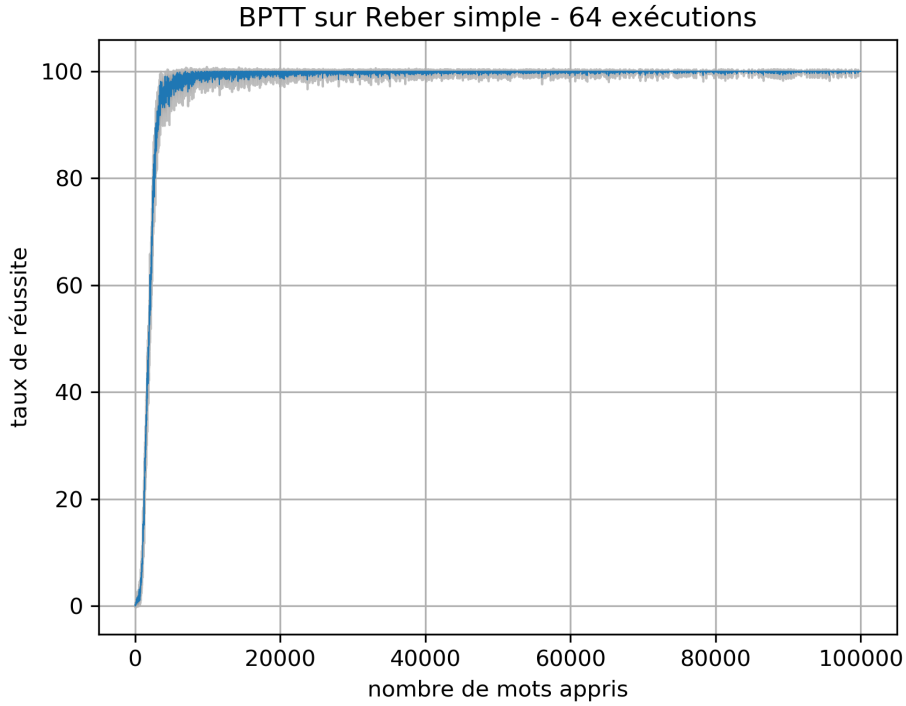

FIGURE 4.2 – Apprentissage au cours du temps, BPTT sur grammaire de Reber simple

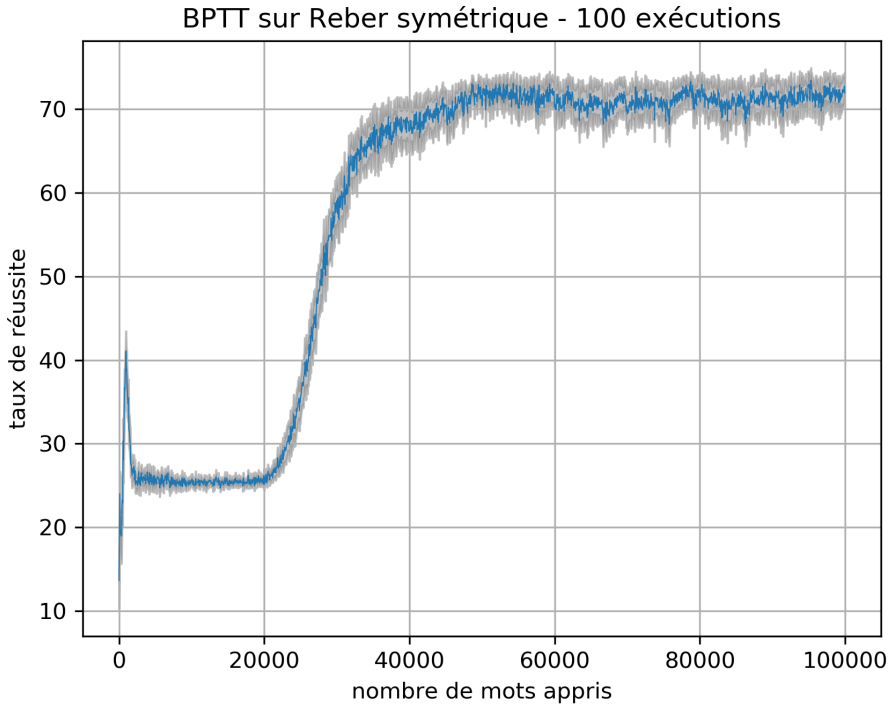

FIGURE 4.3 – Apprentissage au cours du temps, BPTT sur grammaire de Reber symétrique

Chapitre 5

Long Short Term Memory

Objectif principal de ce projet, l'architecture neuronale Long Short Term Memory (LSTM) est décrite dans cette partie. Tout comme les autres architectures neuronales, elle est constituée d'un assemblage de blocs élémentaires qui disposent d'un ensemble de variables, appelés poids, à adapter lors de la phase d'apprentissage afin de reproduire une fonction. Cependant, la cellule élémentaire d'un réseau LSTM est bien plus complexe que celle d'un réseau neuronal à perceptrons.

La dénomination LSTM vient du fait que ce type de réseau possède une mémoire de plus longue durée que des structures plus simples avec une seule couche de neurones. Ainsi, il sera possible d'apprendre des fonctions telles que la grammaire de Reber double, ou bien de générer du texte après avoir appris des écrits de Shakespeare.

LSTM est notamment utilisé dans des applications de reconnaissance vocale.

5.1 Théorie

5.1.1 Cellule LSTM

La cellule LSTM est constituée de différents réseaux de perceptrons. La principale différence avec un simple réseau à perceptrons est que la cellule possède une mémoire. Il s'agit d'un vecteur qui sera pris en entrée (en même temps que l'entrée au temps t et la sortie du temps $t-1$, modifié par les entrées au temps t et renvoyé au temps $t+1$ dans la même cellule.

Chaque réseau de perceptrons représente une opération sur la mémoire, tous prennent en entrée l'entrée de la cellule au temps t et sa sortie au temps $t-1$:

- La "input gate" est le réseau qui effectue des opérations d'addition sur la mémoire.
- La "block input" est le réseau qui définit les coordonnées de la mémoire qui seront affectées par l'input gate.
- La "forget gate" est le réseau qui détermine quelles coordonnées de la précédente mémoire garder à t. Ce réseau n'est pas présent dans toutes les implémentations, on pourra étudier l'impact de sa présence.
- La "output gate" est le réseau qui détermine les sorties de la cellule.

5.1.2 Propagation

Soit N la taille du vecteur de sortie et M la taille du vecteur d'entrée, on peut définir les matrices de poids :

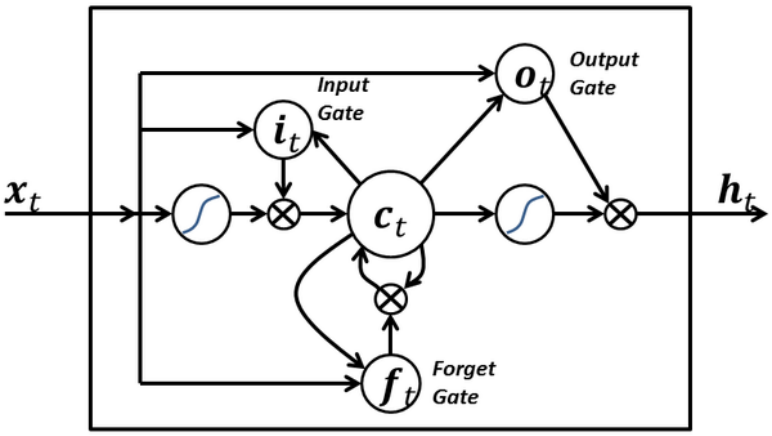

FIGURE 5.1 – Cellule LSTM

— Poids relatifs à l'entrée : W_z, W_i, W_f, $W_o \in \mathbb{R}^{M \times N}$
— Poids relatifs à la sortie précédente : R_z, R_i, R_f, $R_o \in \mathbb{R}^{N \times N}$
— Poids des biais : b_z, b_i, b_f, $b_o \in \mathbb{R}^N$

Soit σ la fonction d'activation sigmoide $\sigma(x) = \frac{1}{1+e^{-x}}$, x^t l'entrée au temps t, y^t la sortie de la cellule au temps t, c^t la mémoire de la cellule au temps t et $\odot$ le produit d'Hadamard, on peut définir les expressions suivantes pour les sorties de chaque couche de perceptrons :

— block input :

$$\overline{z}^t = W_z x^t + R_z y^{t-1} + b_z$$
$$z^t = \tanh(\overline{z}^t)$$

— input gate :

$$\overline{i}^t = W_i x^t + R_i y^{t-1} + b_i$$
$$i^t = \sigma(\overline{i}^t)$$

— forget gate :

$$\overline{f}^t = W_f x^t + R_f y^{t-1} + b_f$$
$$f^t = \sigma(\overline{f}^t)$$

— output gate :

$$\overline{o}^t = W_o x^t + R_o y^{t-1} + b_o$$
$$o^t = \sigma(\overline{o}^t)$$

5.1.3 Algorithmes d'apprentissage

L'algorithme d'apprentissage est un algorithme BPTT appliqué aux cellules LSTM considérées.

5.2 Implémentation

L'implémenation est effectuée en C++ via la librairie de calcul matriciel Eigen3. Toutes les matrices sont des objets de type Eigen : :MatrixXd (matrice de double) et les vecteurs des objets de type Eigen : :VectorXd.

L'aléatoire utilisé est celui natif en C et C++ : rand. La génération de la graine se fait à partir du temps à la milliseconde pour éviter une initialisation déterministe dans le cas de l'execution de plusieurs runs consécutifs. Pour cela la librairie 'sys/time.h' est utilisée, avec un appel propre aux systèmes UNIX.

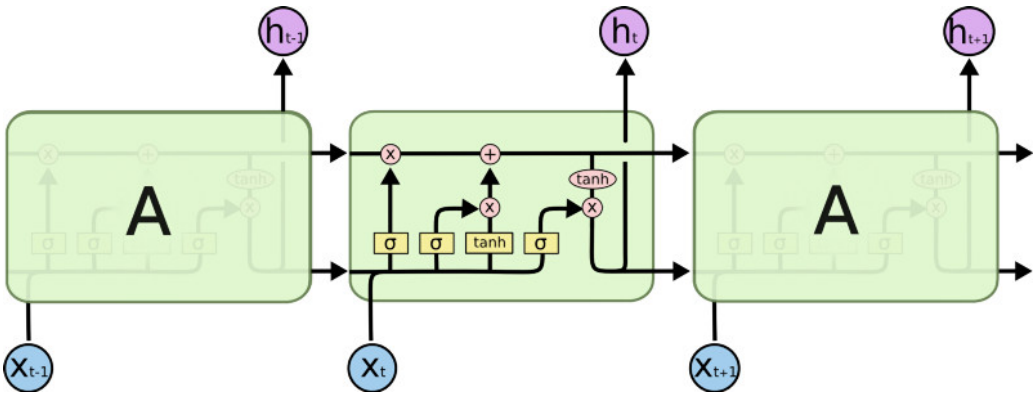

FIGURE 5.2 – Dépliement du temps dans l'espace style BPTT

5.2.1 Structure de données

Le code se décompose en plusieurs éléments :

— Les poids LSTM
— La cellule LSTM
— Le réseau LSTM

5.2.1.1 Les poids LSTM

La classe poids regroupe toutes les matrices de poids des différents noeuds de la cellule LSTM. Ces dernier sont : input_gate, input_block, output_gate. Pour chacun des noeuds il existe deux matrices de poids : une relative à la sortie précédente de la cellule, et l'autre relative à l'entrée de la cellule. Enfin, il existe aussi pour chaque noeud un vecteur de biais.

Tous ces éléments sont amenés à être modifiés, c'est pourquoi l'on crée pour chacun une matrice (ou un vecteur) de delta : modifications à appliquer.

Enfin, les méthodes de l'objet poids LSTM sont le constructeur et l'application des variations de poids (qui remet par la même occasion à 0 les delta-poids).

5.2.1.2 La cellule LSTM

La cellule LSTM est l'objet élémentaire du réseau. Elle est créée en prenant pour arguments un pointeur vers un objet de type poids, et les informations de dimentions d'entrée et sortie.

Elle dispose des méthodes nécessaires à la propagation et la rétropropagation à travers une cellule LSTM. Pour renvoyer plusieurs vecteurs (la sortie, la mémoire) on utilise ici des std : :vector<Eigen : :VectorXd> dont chaque case correspond à un vecteur que l'on souhaite renvoyer.

La propagation n'est rien d'autre qu'une application directe des formules de la propagation à travers une cellule LSTM. Sont stockées certaines valeurs intermédiaires pertinentes pour éviter leur calcul à nouveau lors de la rétropropagation.

Lors de la rétropropagation, les variations de poids sont calculées et ajoutées aux attributs delta_poids de l'objet weightsLSTM. La fonction de cout choisie est la fonction de coût quadratique (divisée par deux), sa dérivée étant alors la différence entre sortie obtenue et sortie attendue.

Les attributs sont donc les suivants :

— `WeightsLSTM* weights` l'objet poids qui sera utilisé par la cellule pour calculer les sorties de chaque noeud ;
— `Eigen::VectorXd input` vecteur des entrées de la cellule à l'instant t ;
— `Eigen::VectorXd previous\_output` vecteur des sorties de la cellule à l'instant $t-1$;
— `Eigen::VectorXd previous\_cell\_state` mémoire de la cellule à l'instant $t-1$;
— `Eigen::VectorXd input\_gate\_out` sortie du noeud input_gate à l'instant t ;
— `Eigen::VectorXd input\_block\_out` sortie du noeud input_block à l'instant t ;
— `Eigen::VectorXd output\_gate\_out` sortie du noeud output_gate à l'instant t ;
— `Eigen::VectorXd cell\_state` mémoire de la cellule à l'instant t ;
— `Eigen::VectorXd cell\_out` sortie de la cellule à l'instant t.

Les méthodes sont donc les suivantes :

— `compute( Eigen::VectorXd previous\_output, Eigen::VectorXd previous\_memory, Eigen::VectorXd input` la methode qui calcule la sortie de chaque noeud de la cellule, puis la sortie de la cellule. Renvoie `<cell\_out, cell\_state>` ;
— `compute\_gradient(Eigen::VectorXd deltas, Eigen::VectorXd previous\_delta\_cell\_in, Eigen::Vecto` la methode qui calcule le gradient en entrée de la cellule en fonction des gradients en sortie. Renvoie `<delta\_input, delta\_cell\_state>`.

5.2.1.3 Le réseau LSTM

Le réseau LSTM est principalement constitué d'un std : :vector<CellLSTM>. A chaque temps t, une nouvelle cellule est crée et stockée à l'index i du vector.

La propagation s'effectue cellule par cellule, chaque cellule prenant en entrée la sortie de la précédente à $t-1$ ainsi que l'entrée du réseau au temps t. De même, la rétropropagation s'effectue de la dernière cellule du vector à la première.

Les attributs sont donc les suivants :

—
— `WeightsLSTM* weights` l'objet poids utilisé pour la création de chaque cellule ;
— `int input\_size` la taille de l'entrée ;
— `int output\_size` la taille de la sortie ;
— `int layer\_size` la taille des couches cachées ;
— `std::vector<Cell> cells` le vecteur dans lequel sont stockées les cellules crées pour chaque instant t de la propagation.

Les méthodes sont donc les suivantes :

— `propagate(std::vector<Eigen::VectorXd> inputs)` crée à chaque instant t une nouvelle cellule et utilise sa methode `compute` pour calculer l'entrée de la suivante ;
— `backpropagate(std::vector<Eigen::VectorXd> expected\_outputs)` rétropropage le gradient de la dernière cellule à la première en utilisant la methode `compute\_gradient` de chaque cellule ;

5.3 Résultats

Ci-dessous des représentations de l'apprentissage au cours du temps du réseau sur des grammaires de Reber, simple et double.
En abscisse, le nombre de mots appris, en ordonnée le taux de réussite, testé à intervalles réguliers sur un

échantillon de données de test choisies aléatoirement dans l'ensemble de test.
La zone grise correspond à l'intervalle de confiance à 95% sur le nombre d'exécutions précisé.
Le réseau utilisé est composé d'une couche de 30 cellules LSTM, avec un learning rate de 0.1.

5.3.1 Grammaire de Reber simple

Pour une grammaire de Reber simple, la réussite est déterminée par la prédiction correcte de toutes les transitions des mots testés.

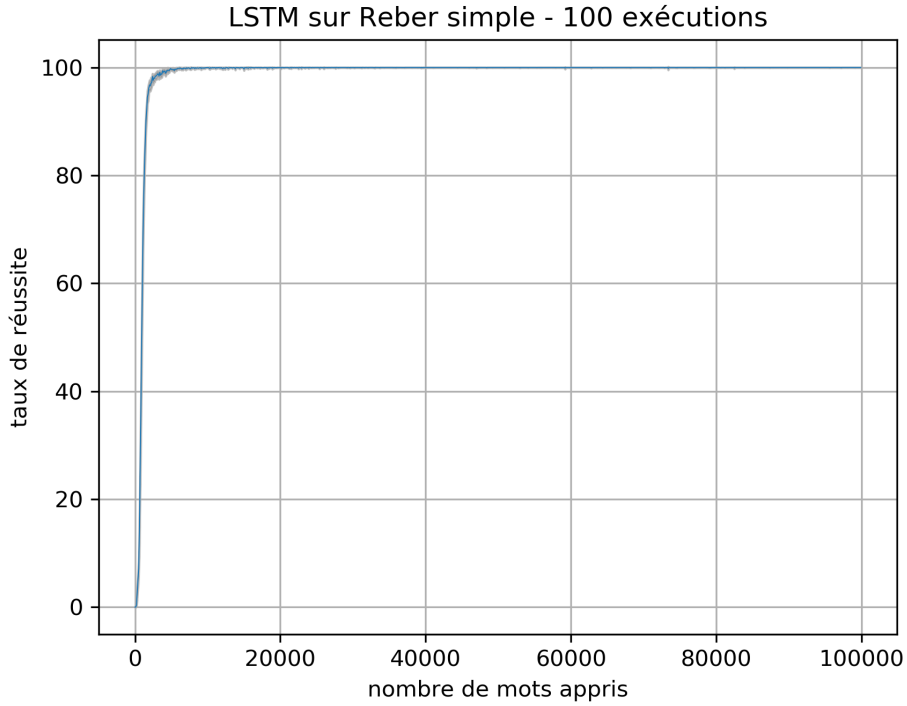

FIGURE 5.3 – Apprentissage au cours du temps, LSTM sur grammaire de Reber simple

5.3.2 Grammaire de Reber symétrique

Pour une grammaire de Reber symétrique, réussite est déterminée par la prédiction correcte de la première et la dernière transition des mots testés.

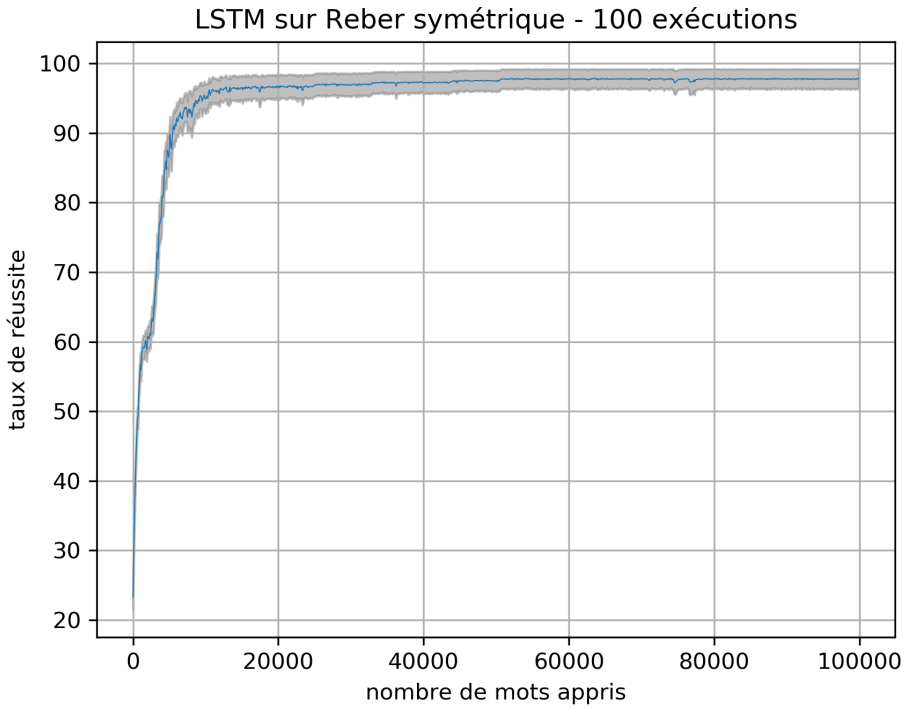

FIGURE 5.4 – Apprentissage au cours du temps, LSTM sur grammaire de Reber symétrique

Annexes

Bibliographie

[1] Neural computing : new challenges and perspectives for the new millennium : proceedings of the IEEE-INNS-ENNS international joint conference on neural networks, IJCNN 2000, como, italy, 24 - 27 july 2000.

[2] Alex Graves and Jürgen Schmidhuber. Framewise phoneme classification with bidirectional LSTM and other neural network architectures. 18(5) :602–610.

[3] Marius-Constantin Popescu, Valentina E. Balas, Liliana Perescu-Popescu, and Nikos Mastorakis. Multilayer perceptron and neural networks. 8(7) :579–588.

[4] Alberto Prieto, Beatriz Prieto, Eva Martinez Ortigosa, Eduardo Ros, Francisco Pelayo, Julio Ortega, and Ignacio Rojas. Neural networks : An overview of early research, current frameworks and new challenges.

[5] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112.

[6] Paul J. Werbos. Backpropagation through time : what it does and how to do it. 78(10) :1550–1560.

[7] Ronald J. Williams and David Zipser. Gradient-based learning algorithms for recurrent networks and their computational complexity. pages 433–486.

[8] Ronald J. Williams and David Zipser. A Learning Algorithm for Continually Running Fully Recurrent Neural Networks. *Neural Comput.*, 1(2) :270–280, June 1989.