Supélec

PinaPL
Projet long : réseaux neuronaux cycliques

Auteurs :
Maxime Amosse
Julien Hemery
Hugo Hervieux
Sylvain Pascou

Référents :
Arpad Rimmel
Joanna Tomasik

21 mai 2017

Résumé

(résumé)

Table des matières

Chapitre 1

Présentation du projet

1.1 Objectifs

Ce projet a été lancé le 23 septembre 2016 sur proposition de Mme. Joana Tomasik et Mr. Arpad Rimmel. Le but est de réimplémenter en moins d'un an l'algorithme LSTM [1]. Il s'agit d'un réseau neuronal récurrent avec mémoire.

Un tel projet représente de fortes contraintes techniques (implémentation du réseau, gestion de la mémoire, performances, ...) mais aussi un défi théorique (preuve de terminaison, gestion des cycles, ...).

Pour arriver à ce résultat, l'année est découpée en différents "runs" et dans chacun 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 Equipe

L'équipe du projet est composée de 4 élèves-ingénieurs de CentraleSupélec Gif : Maxime Amossé, Julien Hemery, Hugo Hervieux et Sylvain Pascou, encadrés par deux enseignants-chercheurs du même établissement, Mme. Joana Tomasik et Mr. Arpad Rimmel.

1.3 Outils

1.3.1 Le langage

Le langage de programmation choisi est le C++, pour sa rapidité d'execution, sa plus grande versatilité que le C 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'intéragir avec les variables durant l'exécution, ce qui complique les phases de débugage.

1. Long Short Term Memory

1.3.2 Git

Pour travailler en groupe de manière efficace nous avons utilisé le gestionnaire de version Git et un dépot distant 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 ne pas créer des conflits à chaque merge et faciliter le travail de chacun.

Chaque développeur travaille donc sur la branche de la fonction qu'il implémente puis la merge dans la branche master lorsqu'il à une version stable qui compile et se comporte correctement.

1.3.3 Zotero

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 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. Ce rapport a été rédigé grâce à LaTeX.

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 $f : \mathbb{R} \to \mathbb{R}$

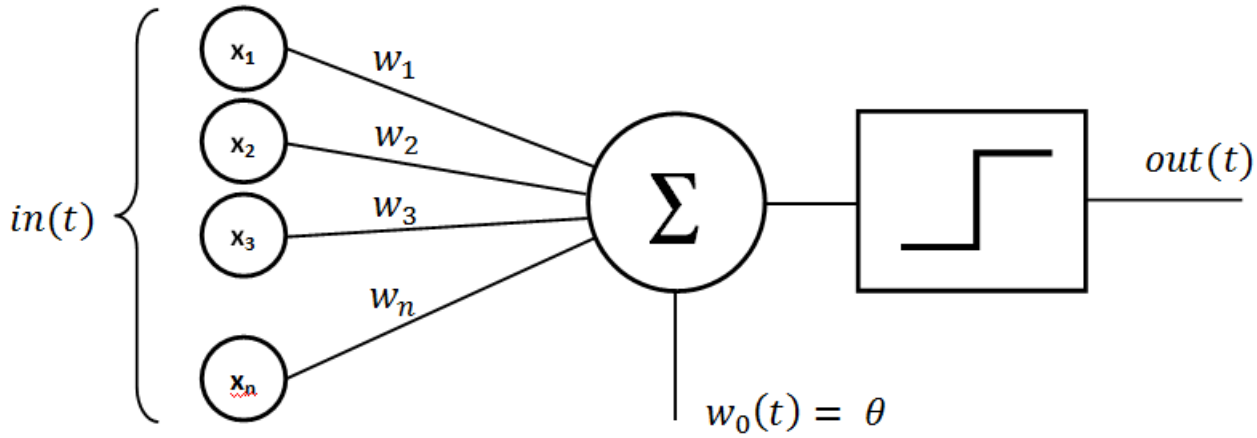

FIGURE 2.1 – Schéma d'un perceptron simple

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 :

$$y = f(g(x_1w_1, x_2w_2, \ldots, x_nw_n) + \theta)$$

Usuellement, la fonction d'activation est la somme :

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

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 $\forall i \in [1, m]$:

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

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 $\frac{\partial E}{\partial w_{ij}^{(k)}} = \left\langle \frac{\partial \overline{y^{(l-1)}}}{\partial w_{ij}^{(k)}}, u \right\rangle$

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 débogage, nous avons obtenu des résultats satisfaisants.

2.3.1 Le XOR

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 reccurent learning

A partir de cette section, les objectifs sont de reconnaître une chaine 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. Ce genre de problème peut être étendu à la recherche comportementale en temps réel. Pour résoudre ce genre de problème, on utilise des réseaux neuronaux récurrents, qui ont l'avantage de se souvenir des états précédents pour pouvoir prédire efficacement les états suivants ; ils comportent une mémoire courte. Dans un premier temps, nous allons nous intéresser aux réseaux RTRL.

3.1 Théorie

Dans la suite, nous allons nous intéresser au problème de la grammaire de Reber, qui servira d'échantillon de test pour RTRL.

3.1.1 La grammaire de Reber

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

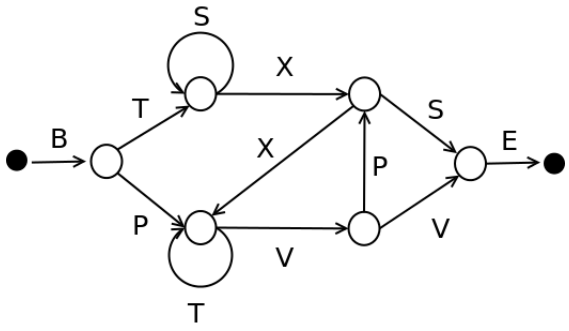

FIGURE 3.1 – Grammaire de Reber simple

De base, on considère une probabilité uniforme de choisir l'état suivant parmi les états possibles suivants. 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 variées, 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 que c'est un automate simple qui ne nécessite que la mémoire de la

dernière et de l'avant dernière lettre pour trouver la suivante. 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. Ce modèle bien que résolvant ce problème, n'est pas adapté au calcul en temps réel. De plus il ne résout pas le problème de la grammaire de Reber double.

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

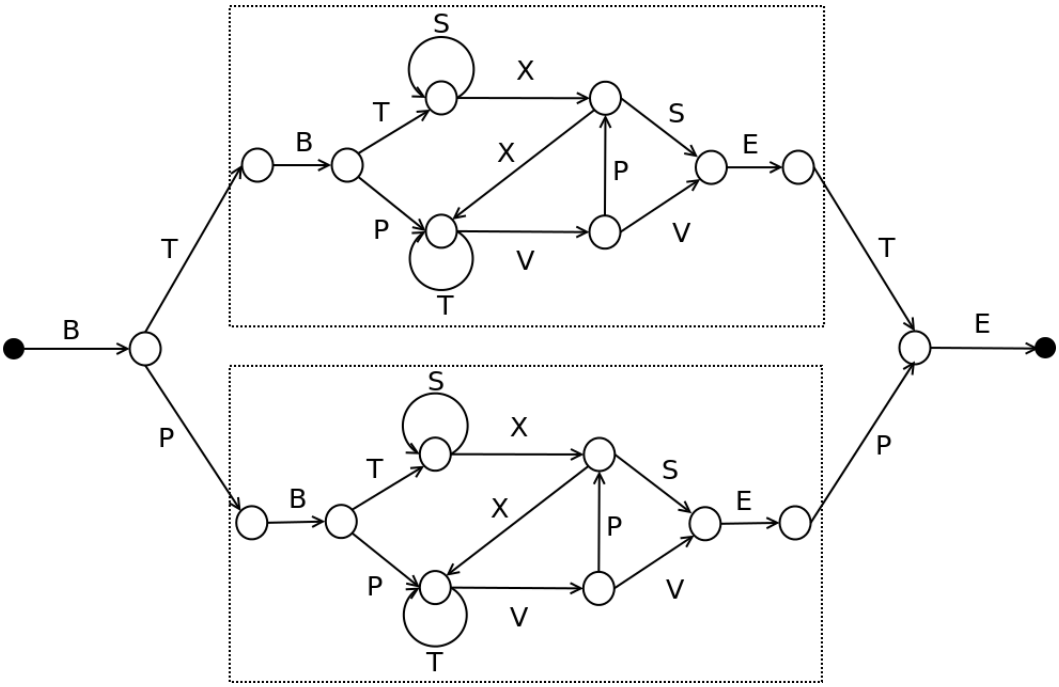

FIGURE 3.2 – Grammaire de Reber symétrique

On remarque qu'il est constitué de deux grammaires de Reber simple qui sont reliés en entrée et en sortie. La difficulté de ce problème est qu'il faut mémorisé la première valeur pour en déduire la dernière. Dans ce cas, une mémoire "infini" est nécessaire théoriquement pour se souvenir de la première entrée. Il est donc impensable d'utiliser de la même façon un réseau de perceptron classique. On a alors besoin de réseau récurrent, dont la sortie à un instant t va dépendre de la sortie à un instant $t-1$.

3.1.2 Réseau RTRL

Le principe d'un réseau 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 : Dans un premier temps, 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 $\overline{y(t)} = W \times x(t) + R \times y(t-1) + b$. Les paramètres du systèmes sont donc les éléments de la matrice On obtient donc pour l'apprentissage :

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

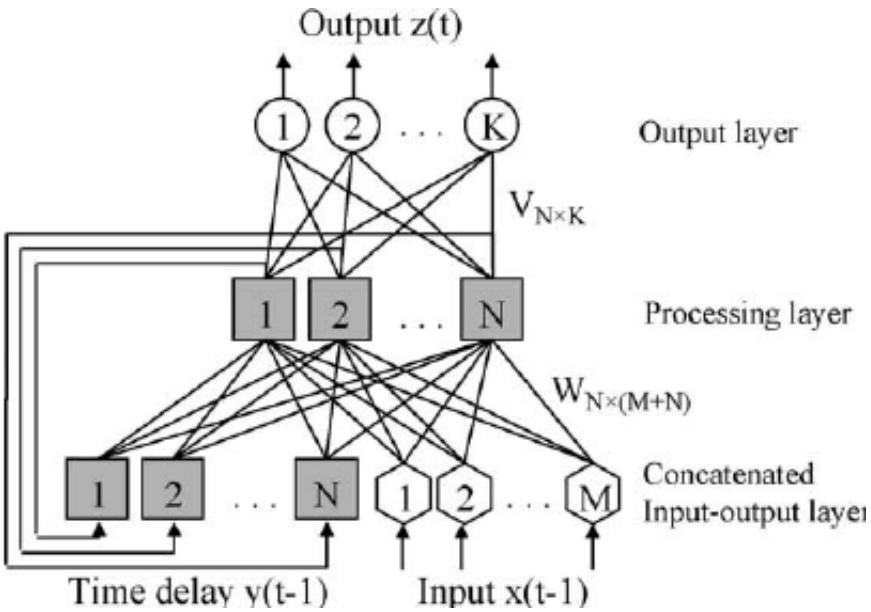

FIGURE 3.3 – Réseau récurrent RTRL

3.2 Implémentation

3.3 Résultats

3.3.1 Grammaire de Reber simple

3.3.2 Grammaire de Reber double

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

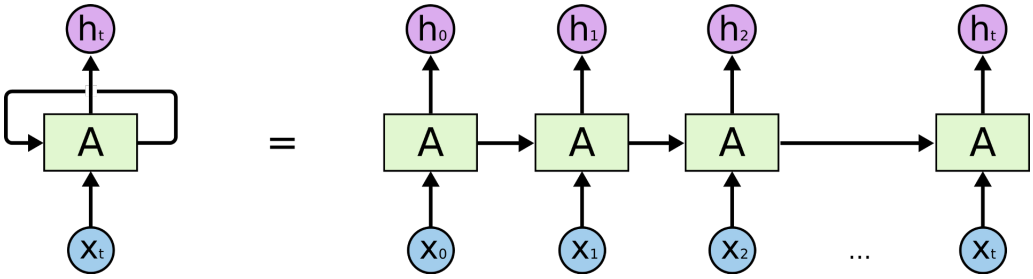

FIGURE 4.1 – Dépliement du temps dans l'espace pour BPTT

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

4.3.1 Grammaire de Reber simple

4.3.2 Grammaire de Reber double

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 un petit réseau de perceptrons récurrent présentant des élément nommés en fonction de leur rôle dans la cellule.

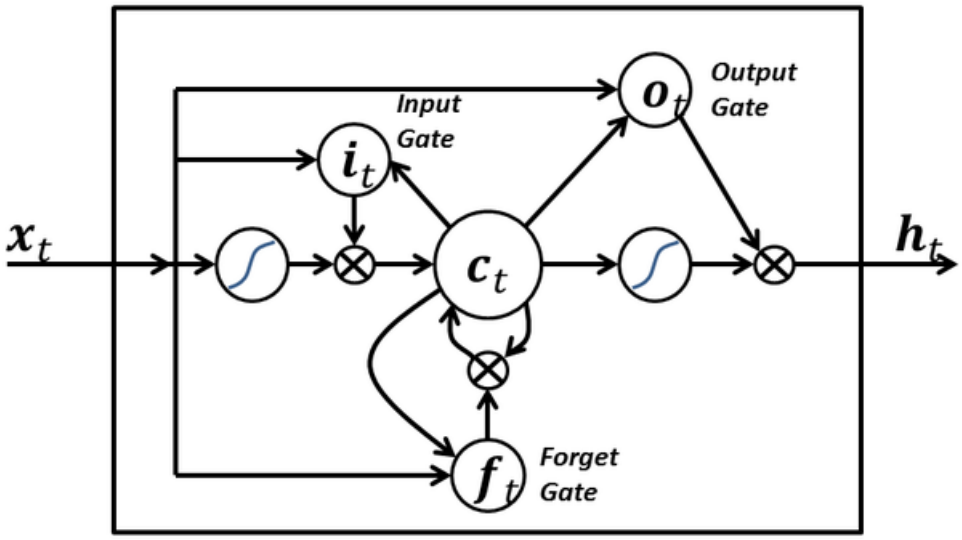

Figure 5.1 – Cellule LSTM

5.1.2 Propagation

5.1.3 Algorithmes d'apprentissage

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

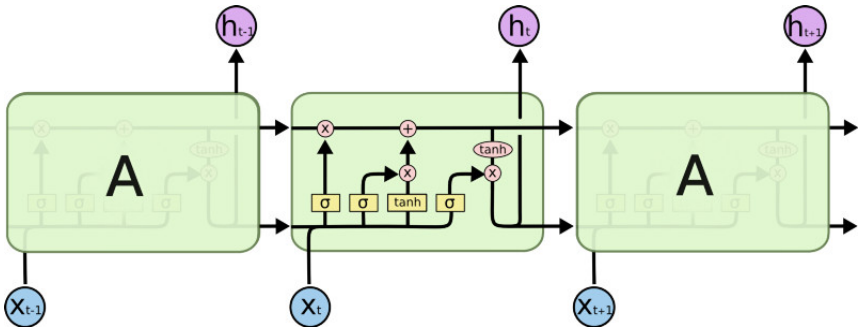

FIGURE 5.2 – Dépliement du temps dans l'espace style BPTT

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.

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 :

Les méthodes sont donc les suivantes :

5.2.1.3 Le réseau LSTM

Le réseau LSTM est principalement constitué d'un std : :vector<CellLSTM>. A chaque temps t_i, 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_{i-1} ainsi que l'entrée du réseau au temps t_i.

De même, la rétropropagation s'effectue de la dernière cellule du vector à la première.

Les attributs sont donc les suivants :

Les méthodes sont donc les suivantes :

5.3 Résultats

5.3.1 Apprentissage sur un mot

5.3.2 Grammaire de Reber simple

5.3.3 Grammaire de Reber double

Annexes

Annexe 1

Intro

1 Partie 1

Bla

1.1 Sous-partie 1

Bla

1.2 Sous-partie 2

Bla

1.3 Sous-partie 3

Bla

2 Partie 2

Bla

2.1 Sous-partie 1

Bla

2.2 Sous-partie 2

Bla

2.3 Sous-partie 3

Bla

Annexe 2

Intro

Prérequis

Bla

— item1 ;
— item2 ;
— item3 ;
— item4.

Bla

1 Partie 1

Bla

1.1 Sous-parie 1

Bla

1.2 Sous-parie 2

Bla

2 Partie 2

Attention !

Texte d'avertissement

Bla

3 Partie 3

Bla

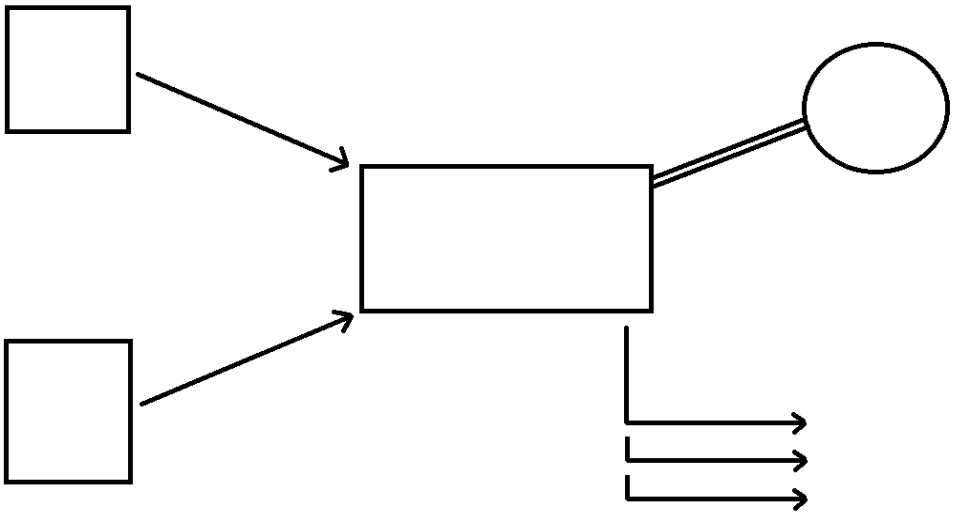

FIGURE 5.3 – Presentation schema

Paragraphe 1

Bla

Paragraphe 2

Bla

Paragraphe 3

Bla