Les réseaux récurrents supervisés : une revue critique

Nicolas SZILAS
Leibniz-Imag
INPG-UJF
46, av. Félix Viallet
38031 Grenoble Cedex, France

E-mail : Nicolas.Szilas@imag.fr

Février 1997

1. Taxonomie des réseaux récurrents supervisés *2*
2. Deux principaux algorithmes d'apprentissage dans les réseaux dynamiques *6*
2.a Rétropropagation dans le temps 6
2.b Propagation avant du gradient 12
3. Analyse et difficultés de ces réseaux *14*
3.a Coûts computationnels 14
3.b Non-localités spatiales 15
3.c algorithmes "temps réels" ? 19
3.d problèmes de convergence 20
4. Variantes et améliorations existantes *22*
4.a Forçage du réseau 22
4.b Elagage de l'historique 25
4.c Architectures évolutives 26
Conclusion *27*
Références *27*

Ce rapport propose un état de l'art concernant les réseaux de neurones dits récurrents. Il s'agit d'une version adaptée et mise à jour de [Szilas 95, p. 70-105]. Cet état de l'art n'est ni exhaustif ni équilibré, car un livre entier serait nécessaire pour un tel projet (étant donné que la plupart des réseaux sont récurrents). Ce rapport va plutôt tenter de fournir une présentation synthétique des réseaux récurrents, et clarifier certaines notions parfois confuses dans la littérature.

Il existe de nombreux types de réseaux récurrents. Dans ce chapitre seuls les réseaux récurrents supervisés dédiés à l'apprentissage de séquences sont abordés : pendant l'apprentissage, un comportement désiré est fourni par l'environnement, comportement que le réseau a pour tâche de reproduire. En particulier, les réseaux fonctionnant en mémoires associatives [Hopfield 82] [Labbi 93] ne seront pas abordés.

Dans ce qui suit est proposée une taxonomie des réseaux récurrents supervisés, suivie de l'exposé détaillé des deux algorithmes d'apprentissage les plus utilisés, leurs limites et leurs variantes.

1. Taxonomie des réseaux récurrents supervisés

Plutôt que de présenter les différents modèles de réseaux récurrents un à un, nous avons préféré faire ressortir dans cet état de l'art les *traits* qui distinguent les algorithmes les uns des autres. Il en ressort qu'un même réseau peut être cité plusieurs fois. Pour un exposé plus conventionnel des réseaux récurrents, on pourra se référer à [Pearlmutter 90,95] (pour ce qui concerne les réseaux en temps continu), [Williams 90][Rohwer 94], [Piché 94], les thèses [Raysz 91], [Jodouin 93], et [Crucianu 94], les chapitres des ouvrages généraux [Hecht-Nielsen 91] et [Haykin 94].

Nous allons donc énumérer et commenter les différentes caractéristiques qui permettent de distinguer les différents modèles existants.

Types de comportements appris

De même qu'on apprend aux réseaux unidirectionnels à couches des paires d'entrées-sorties statiques, on apprend généralement aux réseaux récurrents supervisés à associer des paires de patterns *dynamiques*. Selon le type de patterns traités, plusieurs cas ont été étudiés. Tout d'abord, on peut s'intéresser uniquement au comportement asymptotique du réseau, c'est-à-dire au bout d'un temps très grand. On peut ainsi faire apprendre à un réseau des attracteurs de différents types : ponctuels, cycliques voire chaotiques. Par exemple, pour l'algorithme de "rétropropagation récurrente" [Pineda 87], on fait apprendre un ensemble d'apprentissage analogue à celui qu'on utiliserait pour entraîner un réseau unidirectionnel, mais le deuxième élément de chaque paire de l'ensemble (la "sortie désirée") est l'attracteur ponctuel désiré. Une extension de ce type d'apprentissage consiste à faire apprendre une succession d'attracteurs ponctuels [Crucianu 94].

Par ailleurs, de nombreux réseaux récurrents peuvent être utilisés pour *générer* des séquences. A un pattern fixe en entrée doit être associée une succession de patterns donnée. C'est le cas de l'algorithme présenté dans [Jordan 88] utilisé en robotique pour commander un bras de robot (simulé), où un réseau à couches bouclé est entraîné à atteindre une cible donnée. Dans [Guyon *et al.* 88], une variante du réseau de Hopfield est utilisée pour stocker puis restituer une séquence donnée. Dans [Williams & Zipser 89a,b], un réseau est entraîné à produire un signal sinusoïdal (mais l'algorithme peut aussi réaliser d'autres types d'associations).

A l'inverse, on peut utiliser certains réseaux récurrents pour classifier des séquences temporelles, c'est-à-dire qu'en entrée sont présentées des séquences de patterns, et en sortie une classe statique.

Enfin, le cas général est celui où et les entrées et les sorties sont des séquences ([Rumelhart *et al.* 86, p. 354][Le Cun 87, p. 121][Robinson 89][Williams & Zipser 89a][Rohwer 90], etc.). Bien sûr, tous les

algorithmes qui peuvent effectuer des associations de séquences peuvent être utilisés en génération ou reconnaissance de séquences.

Structure

La structure du réseau, c'est-à-dire la topologie du graphe d'interconnexion, est variable selon les études.

La structure entièrement connectée est souvent utilisée [Rumelhart *et al.* 86 p. 354][Le Cun 87 p. 121][Pearlmutter 89][Gherrity 89][Williams & Zipser 89a][Robinson 89][Rohwer 90][Sun *et al.* 91][Toomarian & Barhen 92]. Elle est très générale, puisqu'elle permet d'approcher n'importe quelle séquence [Li 92], mais nécessite beaucoup de calculs.

La structure "récurrente à couches" [Jodouin 93], qui consiste en un réseau entièrement connecté à l'intérieur d'un réseau à couches (voir figure 1) est très utilisée, en particulier par le réseau appelé "Simple Recurrent Network" développé par J.L. Elman. [Cleeremans *et al.* 89][Elman 90][Jodouin 93][Crucianu 94].

Une autre structure est celle où la couche cachée possède seulement des connections auto-récurrentes [Mozer 88][Watrous *et al.* 90]. Cette structure entre dans la classe générale des algorithmes "localement récurrents globalement unidirectionnels" [Tsoi & Back 94].

Le réseau décrit dans [Jordan 88] est de type unidirectionnel bouclé, c'est-à-dire que le cœur du réseau est unidirectionnel, et la récurrence se fait des sorties vers les entrées de ce bloc unidirectionnel.

Enfin, nous mentionnons aussi la structure décrite dans [Narendra & Parthasarathy 90][Lin *et al.* 95], qui est une structure à deux couches, la deuxième rebouclée sur la première par l'intermédiaire d'unités à retard, grâce auxquelles chaque unité de la première couche reçoit un historique temporel des unités de sortie. Une variante de cette structure consiste à ajouter une couche cachée [Tsung & Cotrell 95].

Ces structures sont illustrées sur la figure 1.

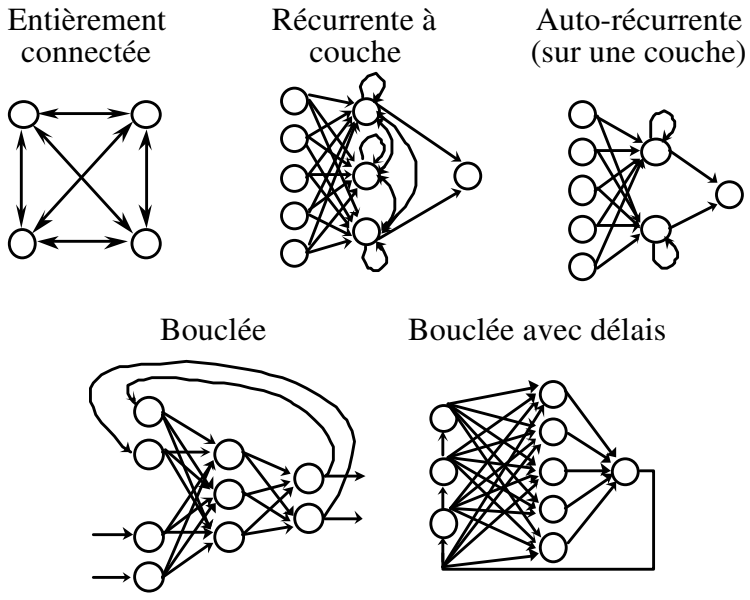

Figure 1. Différentes structures récurrentes

Formalisme

Deux types de formalisme sont utilisés pour décrire les algorithmes, et en particulier les automates utilisés :

- Le formalisme en temps discret :

$$y_i(t+1) = f\left(\sum_j w_{ij} y_j(t) \right)$$

- Le formalisme en temps continu :

$$\tau \frac{dy_i}{dt}(t) = -y_i(t) + f\left(\sum_j w_{ij} y_j(t)\right)$$

avec $y_i(t)$ l'activité de l'automate i à l'instant t et w_{ij} le poids de la connection de l'automate i vers l'automate j.

Quels sont les avantages et inconvénients de l'un et l'autre des deux formalismes ?

D'une part, l'implantation du réseau va guider le choix entre les deux formalismes. Si l'objectif est de programmer le réseau sur un ordinateur fonctionnant en discret, alors même si l'on choisit un formalisme continu, il faudra bien discrétiser finalement les équations en continu, ce qui introduira un bruit de quantification dont le modèle ne tient pas compte, alors qu'avec les équation discrètes, il y a identité entre le modèle formel et le modèle effectivement simulé. Si par contre une implantation *analogique* est envisagée, dans laquelle en particulier aucune horloge ne régule les échanges d'information, alors le formalisme continu doit être utilisé [Gherrity 89][Crucianu 94].

D'autre part, les deux formalismes amènent à utiliser des théories mathématiques différentes. De notre point de vue, l'approche en temps discret conduit à des calculs plus simples ; Cependant, certains algorithmes ont été d'abord démontrés en formalisme continu [Sun *et al.* 91][Toomarian & Barhen 92].

Enfin, comme, dans les deux équations ci-dessus, la première n'est pas la discrétisation de la seconde, on n'obtient pas le même comportement. En effet, l'automate décrit en temps continu a la propriété de mieux maintenir son activité au cours du temps, même si aucun signal ne lui est transmis [Pearlmutter 90] alors que dans le cas discret, l'activité devient instantanément nulle quand aucun signal n'est transmis. En choisissant une formulation discrète qui est effectivement une discrétisation de la formulation continue, on obtient [Tsung 90]:

$$y_i(t + \Delta t) = \left(1 - \frac{\Delta t}{\tau}\right) y_i(t) + \frac{\Delta t}{\tau} f\left(\sum_j w_{ij} y_j(t)\right)$$

Δt étant le pas de discrétisation.

Une telle équation reproduit le comportement en temps continu, et peut donner de meilleurs résultats que la formulation discrète classique [Lambert & Hecht-Nielsen 91]. Plus qu'une différence entre discret et continu, comme cela est soutenu dans [Pearlmutter 95], il s'agit donc d'une différence de choix de la fonction de transfert de l'automate.

De même, une autre propriété parfois avancée en faveur du formalisme continu, à savoir la possibilité de varier la discrétisation au cours du temps [Pearlmutter 95], s'avère ne pas être incompatible avec le formalisme discret, puisque dans la formule proposée ci-dessus, on peut faire varier Δt au cours du temps.

Algorithme d'apprentissage

Il existe un nombre relativement limité d'algorithmes permettant d'adapter les poids des réseaux récurrents supervisés. Ils sont fondés sur la minimisation d'une fonction d'erreur entre le comportement du réseau et le comportement désiré, et utilisent l'information de gradient. Les deux principaux sont les algorithmes de "Back-Propagation Through Time" (*BPTT*) [Rumelhart *et al.* 86 p. 354][Le Cun 87 p. 121][Pearlmutter 90] et de "Real-Time Recurrent Learning" (*RTRL*) [Williams & Zipser 89a][Robinson 89][Gherrity 89] qui font l'objet du paragraphe qui suit.

L'algorithme nommé "Moving Target" [Rohwer 90] permet aussi l'adaptation des poids. Il présente cependant quelques sérieuses difficultés : d'une part il n'est pas "on-line" (voir le paragraphe 4.c pour la définition du terme), et d'autre part, selon l'auteur lui-même de l'algorithme, il est extrêmement lent [Rohwer 94].

Les algorithmes développés dans [Sun *et al.* 91] et [Toomarian & Barhen 92] sont des améliorations algorithmiques de l'algorithme *RTRL* qui réduisent la complexité des calculs. Il s'appliquent au formalisme continu.

Chacun de ces algorithmes existe en deux versions, selon que le formalisme discret ou continu est utilisé.

Un certain nombre d'algorithmes sont des cas particuliers de ces algorithmes très généraux. Par exemple, l'algorithme développé dans [Mozer 88] est en fait une application de l'algorithme *RTRL* au cas des réseaux auto-récurrents. De même, l'algorithme du "Simple Recurrent Network" est une variante de l'algorithme de *BPTT* (où le paramètre *h*, décrit plus loin au paragraphe 2.a vaut *1*).

D'autres caractéristiques sont encore à mentionner : la nature des activités (binaires, discrètes ou continues), l'existence ou non d'automates cachés, le type d'automates (fonction de transfert, réseaux du premier ordre ou du second ordre [Giles *et al.* 92], etc.), la nature fixe ou évolutive des structures, etc.

2. Deux principaux algorithmes d'apprentissage dans les réseaux dynamiques

BPTT et *RTRL* sont les deux principaux algorithmes utilisés pour entraîner les réseaux récurrents supervisés. Ils sont à la fois très proches et très éloignés : à partir d'un même type de réseau (réseaux récurrents), un même critère d'erreur à minimiser (généralement une erreur quadratique), une même technique d'optimisation (descente de gradient) on dérive deux algorithmes aux caractéristiques antithétiques. Plusieurs études ont replacé les algorithmes *BPTT* et *RTRL* au sein d'un cadre théorique unitaire : [Piché 94] dans le cadre des dérivées partielles ordonnées, [Beaufays & Wan 94] dans le cadre de la théorie des graphes de flux, [Nerrand *et al.* 94] dans le cadre du filtrage adaptatif.

Il existe de nombreuses manières d'exposer ces algorithmes ; ici ont été choisies des formulations avant tout claires et didactiques, largement inspirées de [Williams & Peng 90] et [Williams & Zipser 89a] respectivement. Dans le présent chapitre, seuls les algorithmes de base et leurs variantes immédiates sont exposés, pour laisser au chapitre suivant (chapitre 3) l'analyse plus critique des algorithmes en question, chapitre qui mettra en évidence un certain nombre de leurs difficultés. Le chapitre 4 examinera des solutions qui ont été proposées pour résoudre quelques unes de ces difficultés.

2.a Rétropropagation dans le temps

On peut voir cet algorithme comme une application de l'algorithme de rétropropagation du gradient dans les réseaux multicouches au cas des réseaux récurrents [Rumelhart *et al.* 86, p. 354][Le Cun 87, p. 121]. L'astuce consiste à représenter les états successifs d'un réseau récurrent comme autant de couches d'un réseau unidirectionnel, chaque couche correspondant à un intervalle de temps. On peut ainsi "déplier" un réseau récurrent en un réseau à couches, comme illustré sur la figure 2. L'erreur du réseau déplié de n couches est l'erreur du réseau récurrent au temps n. On peut alors modifier les poids du réseau à couches, selon l'algorithme classique de rétropropagation du gradient. Cette modification doit se faire en respectant la contrainte selon laquelle les poids sont les mêmes d'une couche à l'autre, ce qui revient à *partager* ces poids.

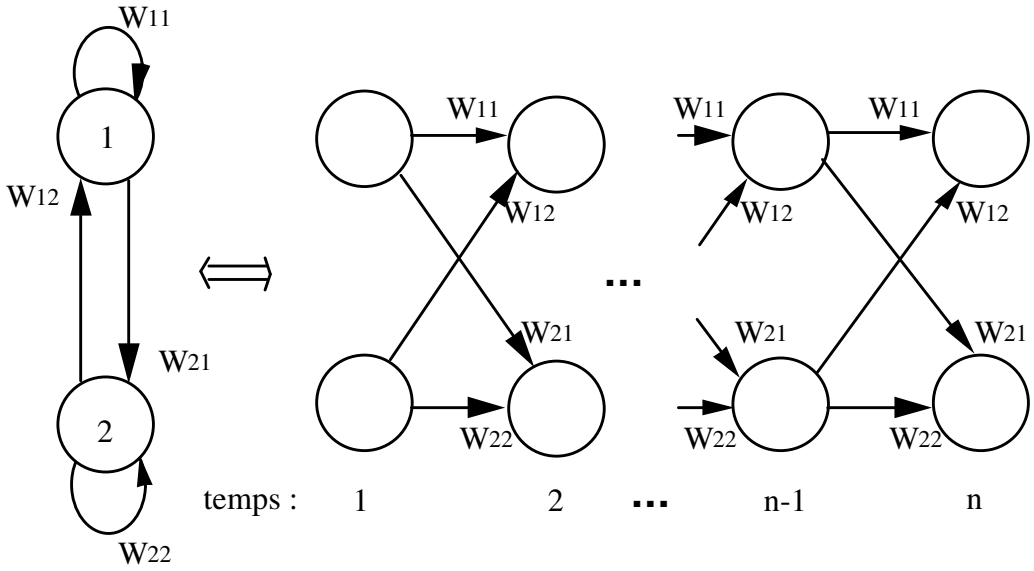

Figure 2. réseau récurrent déplié dans le temps

Cette équivalence entre réseaux récurrents et réseaux à couches, qui autorise une explication simple et visuelle de l'algorithme *BPTT*, doit cependant laisser la place à une formulation un peu plus mathématique, qui notamment explicite clairement comment tenir compte d'une erreur à chaque instant et évite le passage par la multiplication du nombre de poids puis le partage de ces poids.

Soient :

- $x^e{}_i(t)$ la valeur de l'automate d'entrée *i* à l'instant *t*
- $y_i(t)$ la valeur de la sortie de l'automate *i* à l'instant *t* (son activité)
- *U* l'ensemble des indices portant sur les automates du réseau
- *I* l'ensemble des indices portant sur les automates d'entrée
- w_{ij} la valeur du poids de la connexion allant de l'automate *j* à l'automate *i*

Posons ensuite :

$$x_i(t) = \begin{cases} x^e{}_i(t) \text{ si } i \in I \\ y_i(t) \text{ si } i \in U \end{cases} \tag{1}$$

Le calcul effectué par chaque unité peut s'écrire selon les deux équations :

$$\begin{cases} s_i(t+1) = \sum_{j \in U \cup I} w_{ij} x_j(t) \\ y_i(t+1) = f\left(s_i(t+1)\right) \end{cases} \tag{2}$$

avec *f* une fonction différentiable. *f* est généralement non linéaire. Dans la pratique, on choisit souvent la fonction sigmoïde :

$$X \to f(X) = \frac{1}{1+e^{-X}}$$

qui a la propriété simplificatrice suivante :

$$f'(X) = f(X) \times \left(1 - f(X)\right)$$

On remarquera que le réseau a une structure entièrement connectée. Mais cette propriété n'intervient pas dans la démonstration de l'algorithme, qui s'applique à toutes les structures.

La dynamique des activités étant définie, c'est bien sûr celle des poids qui va nous intéresser, puisqu'elle représente *l'apprentissage* du réseau.

Nous définissons alors :

- t_0,t_1 les instants de début et de fin de la séquence
- $T(t)$ l'ensemble des indices portant sur les automates pour lesquels l'environnement fournit une valeur désirée à l'instant t
- $d_i(t)$ la valeur désirée de l'activité de l'automate i à l'instant t
- $e_i(t)$ l'erreur locale de l'automate i à l'instant t :

$$e_i(t) = \begin{cases} d_i(t) - y_i(t) & \text{si } i \in T(t) \\ 0 & \text{sinon} \end{cases} \tag{3}$$

- $E(t)$ l'erreur à l'instant t :

$$E(t) = \frac{1}{2} \sum_{i \in U} e_i(t)^2 \tag{4}$$

- $E^{total}(t_0,t_1)$ l'erreur entre le temps t_0 et le temps t_1 :

$$E^{total}(t_0,t_1) = \sum_{\tau=t_0+1}^{t_1} E(\tau) \tag{5}$$

L'objectif de l'apprentissage est de réduire l'erreur $E^{total}(t_0,t_1)$. Pour cela, on utilise la dynamique du gradient, qui consiste à définir $E^{total}(t_0,t_1)$ comme une fonction des poids du réseau et d'écrire :

$$\begin{cases} \Delta w_{ij} = -\alpha \dfrac{\partial E^{total}(t_0,t_1)}{\partial w_{ij}} & \text{(a)} \\ w_{ij}^{\ nouveau} \leftarrow w_{ij}^{\ ancien} + \Delta w_{ij} & \text{(b)} \end{cases} \tag{6}$$

où Δwij est la modification à apporter au poids wij et α est une constante positive nommée pas d'apprentissage. Rappelons l'interprétation visuelle de cette loi de modification des poids : $E^{total}(t_0,t_1)$ en tant que fonction des poids définit une *surface* multidimensionnelle (voir figure 3).

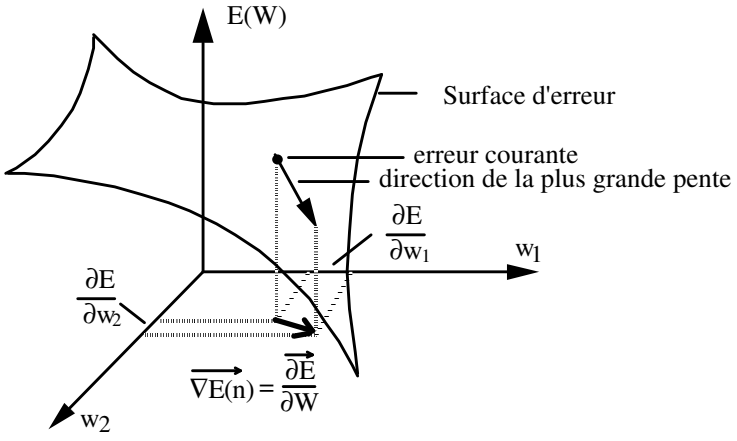

Figure 3. Représentation graphique du principe de descente de gradient

L'état courant du réseau (dans sa dynamique des poids) définit un *point* sur cette surface ; l'objectif étant de minimiser l'erreur, on va chercher à déplacer ce point *vers le minimum* de cette surface, ce qui peut se faire comme si on "laissait rouler" le point courant *selon la ligne de plus grande pente*, autrement dit le gradient. On se rend compte que le minimum atteint peut ne pas être un minimum global et peut donc correspondre à une solution non optimale.

Dérivons maintenant les équations du réseau.

On peut écrire, d'après (6) :

$$\Delta w_{ij} = -\alpha \sum_{\tau=t_0+1}^{t_1} \frac{\partial E^{total}(t_0,t_1)}{\partial s_i(\tau)} \cdot \frac{\partial s_i(\tau)}{\partial w_{ij}}$$

car $E^{total}(t_0,t_1)$ dépend de w_{ij} via les états de l'automate i aux temps successifs entre t_0 et t_1.

Si on pose :

$$\delta_i(\tau) = -\frac{\partial E^{total}(t_0,t_1)}{\partial s_i(\tau)}$$

Alors on obtient :

$$\Delta w_{ij} = \alpha \sum_{\tau=t_0+1}^{t_1} \delta_i(\tau) x_j(\tau-1) \tag{7}$$

Calculons maintenant $\delta_i(\tau)$:

$$\delta_i(\tau) = \frac{\partial E^{total}(t_0,t_1)}{\partial y_i(\tau)} \frac{\partial y_i(\tau)}{\partial s_i(\tau)}$$

Or $E^{total}(t_0,t_1)$ dépend de $y_i(\tau)$ d'une part via la contribution à l'erreur $e_i(\tau)$ et d'autre part via les automates qui reçoivent le signal $y_i(\tau)$. Donc :

$$\begin{cases} \dfrac{\partial E^{total}(t_0,t_1)}{\partial y_i(\tau)} = \dfrac{\partial E^{total}(t_0,t_1)}{\partial e_i(\tau)} \dfrac{\partial e_i(\tau)}{\partial y_i(\tau)} & \text{si } \tau = t_1 \\ \dfrac{\partial E^{total}(t_0,t_1)}{\partial y_i(\tau)} = \dfrac{\partial E^{total}(t_0,t_1)}{\partial e_i(\tau)} \dfrac{\partial e_i(\tau)}{\partial y_i(\tau)} + \displaystyle\sum_{j\in U} \frac{\partial E^{total}(t_0,t_1)}{\partial s_j(\tau+1)} \frac{\partial s_j(\tau+1)}{\partial y_i(\tau)} & \text{si } \tau \in]t_0,t_1[\end{cases}$$

Donc, on obtient :

$$\begin{cases} \delta_i(\tau) = f'(s_i(\tau)) e_i(\tau) & \text{si } \tau = t_1 \\ \delta_i(\tau) = f'(s_i(\tau)) \left[e_i(\tau) + \displaystyle\sum_{j\in U} w_{ji} \delta_j(\tau+1) \right] & \text{si } \tau \in]t_0,t_1[\end{cases}$$

Si l'on pose $\delta_j(t_1+1) = 0,\ \forall j \in U$, on obtient finalement :

$$\forall \tau \in]t_0,t_1],\ \delta_i(\tau) = f'(s_i(\tau)) \left[e_i(\tau) + \sum_{j\in U} w_{ji} \delta_j(\tau+1) \right] \tag{8}$$

Comme $\{\delta_i(\tau)\}_{i\in U}$ dépend de $\{\delta_i(\tau+1)\}_{i\in U}$, il faut calculer ces variables en "remontant le temps", donc une fois toute la séquence terminée (d'où le terme de "rétropropagation dans le temps"). Les équations des signaux δ_i décrivent l'évolution dynamique d'un autre réseau, dont le graphe orienté est l'inverse du graphe orienté des signaux y_i, et que l'on nomme parfois "système adjoint" [Srinivasan, *et al.* 94][Toomarian & Barhen 94].

Cet algorithme présente l'avantage d'être peu coûteux en calculs : si le calcul de la propagation avant prend t secondes, alors le calcul des δ_i prendra t secondes également, auxquelles il faudra seulement ajouter le coût de la modification des poids selon (8).

Par contre, cet algorithme nécessite de stocker les signaux x_i sur toute la séquence (voir équation (7)). Nous reviendrons sur les caractéristiques computationnelles de l'algorithme *BPTT* au paragraphe 3.a.

Plusieurs variantes de l'algorithme *BPTT* ont été proposées, visant à l'utiliser "en ligne" ("on-line"), c'est-à-dire à adapter les paramètres avant la fin de la séquence. Souvent, il s'agit *d'approximations* de l'algorithme original. Le principe de ces variantes est d'effectuer plusieurs rétropropagations dans le temps

dans une même séquence (puisqu'il faut faire plusieurs adaptations) et pour chacune de ces rétropropagations, jouer, selon les variantes, d'une part sur l'intervalle sur lequel est calculée l'erreur rétropropagée, et d'autre part sur le nombre de pas temporels sur lesquels est rétropropagée l'erreur.

Rétropropagation dans le temps en ligne : à chaque instant, on rétropropage l'erreur $E(t)$ définie dans (4). Pour une séquence de longueur T, il faut rétropropager T fois sur un intervalle de plus en plus long. Cette technique n'est donc pas efficace comparativement à la version de *BPTT* où l'on rétropropage une seule fois.

Rétropropagation dans le temps tronquée : c'est une variante de ce qui précède où l'on ignore les longues dépendances dans le temps. Au lieu de rétropropager jusqu'au début de la séquence, on rétropropage uniquement sur h intervalles de temps. il s'agit donc clairement d'une *approximation*, qui permet non seulement de diminuer la quantité de calculs mais aussi de la répartir uniformément dans le temps. Plus h est petit, plus forte est l'approximation et plus faible est le coût computationnel.

Rétropropagation dans le temps par blocs : il s'agit d'une extension de la rétropropagation dans le temps tronquée où l'erreur rétropropagée n'est plus calculée sur un seul instant mais sur h' pas temporels, avec $h'<h$ (pour que toutes les erreurs $e_i(t)$ soient prises en compte). Cette variante permet de rétropropager moins souvent, à savoir un nombre de fois égal à la partie entière de T/h'.

On peut représenter les quatre algorithmes (*BPTT* et ses trois variantes) dans un cadre unifié :

- soit T la longueur de la séquence
- soit t l'instant courant sur la séquence : $t \in [1, T]$
- soit b le nombre d'échantillons sur lesquels on calcule l'erreur à rétropropager
- soit p la profondeur temporelle sur laquelle on rétropropage cette erreur
- soit u l'indice de la rétropropagation courante.

alors on a, pour les quatre algorithmes, les valeurs de b et p suivantes :

	BPTT	BPTT en ligne	BPTT tronquée	BPTT par blocs
b	T	1	1	h'
p	T	t	h (pour t>h-1)	h (pour t>h'-1)

Pour les deux dernières versions, on peut en fait négliger les débuts de séquence en supposant $T>>h$, donc $t>h-1$ dans la plupart des cas (idem pour h'). Le nombre de rétropropagations effectuées vaut T/b, soit $1, T, T, T/h'$ respectivement (on suppose que h' divise T).

Le coût computationnel lié aux rétropropagations est proportionnel à :

$$\sum_{u=1}^{T/b} p(u)$$

c'est-à-dire T, $T(T+1)/2$, hT, hT/h' respectivement. On remarque que la version de base reste la plus efficace.

Sur la figure 4, nous proposons une représentation des quatre algorithmes sur un graphique temporel. L'échelle horizontale représente le temps t de la séquence et l'échelle verticale les opérations successivement réalisées par le réseau. Les flèches fines de gauche à droite représentent la propagation avant des activités x_i et leur stockage, l'échantillonnage des sorties désirées d_i, le calcul et le stockage des erreurs e_i. Les flèches en gras de droite à gauche représentent d'une part la rétropropagation relative à l'erreur courante à minimiser, c'est-à-dire le calcul des δ_i, et d'autre part le calcul final des Δw_{ij}. L'indice u est aussi représenté. Ce schéma résume les comparaisons que l'on peut faire entre les quatre algorithmes.

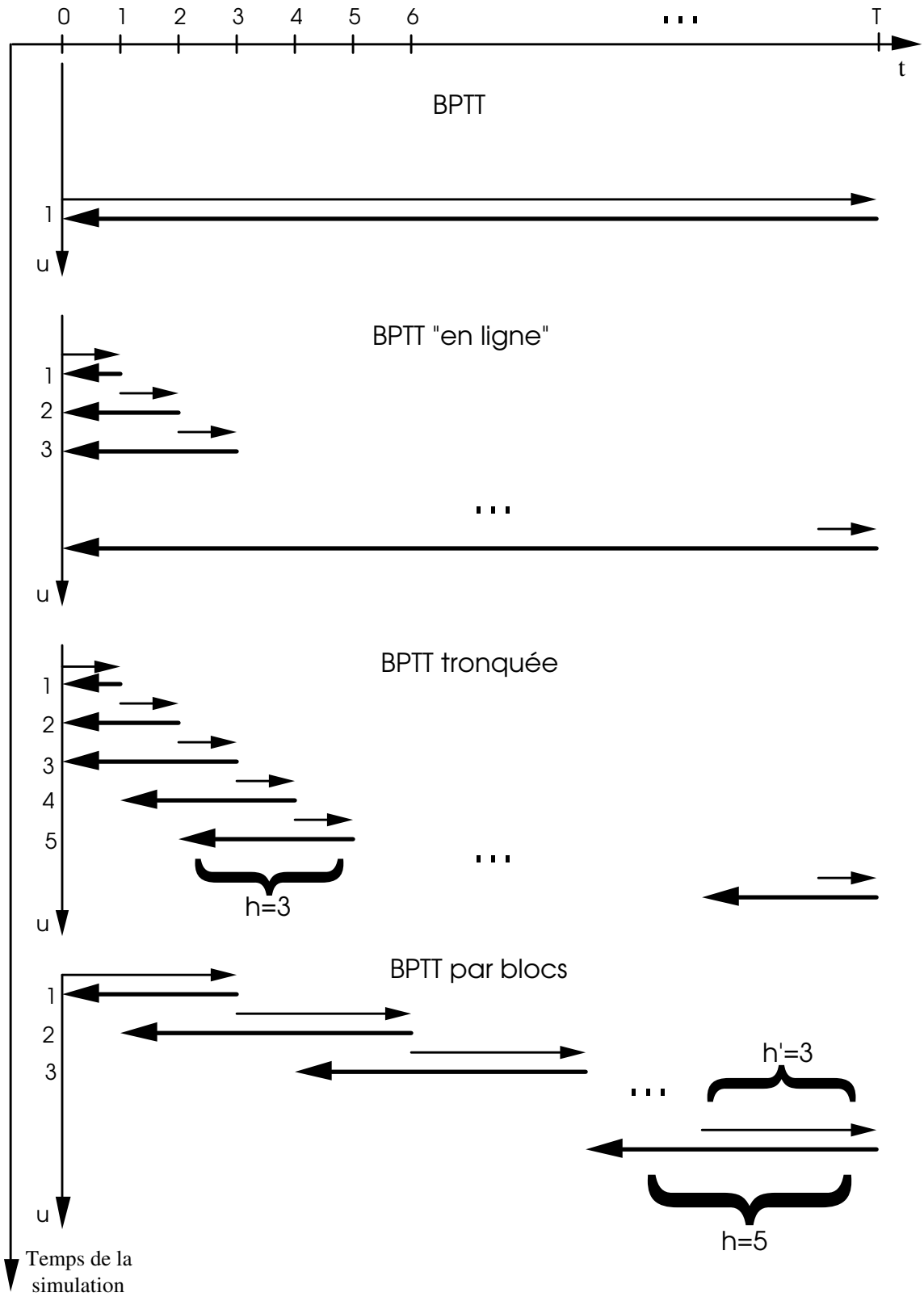

Figure 4. Comparaison des quatre variantes de l'algorithme BPTT

Pour finir, il convient de se demander si le cadre unifié qui vient d'être présenté peut conduire à d'autres versions de l'algorithme *BPTT*. Dans le cas général, les paramètres b et p peuvent être choisis variables dans le temps. Une multitude d'algorithmes pourraient alors être développés. Nous proposons deux pistes pour de nouveaux algorithmes. La première consiste, à partir de la rétropropagation par blocs ci-dessus décrite, à faire varier aléatoirement les paramètres h et h' autour de valeurs moyennes. Une telle introduction de l'aléatoire introduit une sorte de bruit qui peut améliorer la convergence. La deuxième

consiste à modifier la profondeur de rétropropagation dans le temps en fonction des valeurs désirées provenant de l'environnement, afin d'utiliser au mieux la puissance de calcul disponible. En effet, dans un certain nombre d'applications, les valeurs désirées ne sont pas disponibles à chaque instant. Si, entre t_1 et t_2, aucune valeur désirée n'est donnée, alors il n'y a aucune rétropropagation à effectuer : on peut alors prolonger la rétropropagation précédente. Ainsi, on profite mieux du temps de calcul disponible. Cette idée peut s'étendre au cas où l'algorithme *choisit* de ne pas prendre en compte toutes les données de l'environnement (échantillonnage sélectif).

2.b Propagation avant du gradient

Nous allons décrire l'algorithme "Real-Time Recurrent Learning" (*RTRL*). Ce terme a été introduit dans [Williams & Zipser 89a] pour désigner une méthode radicalement différente de la rétropropagation dans le temps pour faire apprendre un réseau récurrent. En fait, la technique utilisée dans *RTRL* n'était pas nouvelle, d'une part parce qu'elle avait été décrite deux ans plus tôt dans [Robinson & Fallside 87 *in* Robinson 89] dans un formalisme un peu différent et d'autre part parce qu'il s'agit en fait d'une technique classique en optimisation et en filtrage adaptatif. De l'avis même de l'un des auteurs [Williams 92], l'algorithme en question aurait été redécouvert huit fois dans le cadre des réseaux ! Dans ce qui suit sera néanmoins présentée la version de [Williams & Zipser 89a], essentiellement en vertu de sa clarté.

On reprend les notations adoptées pour l'algorithme *BPTT* dans le paragraphe 2.a. Comme pour ce dernier algorithme, l'objectif est de minimiser l'erreur $E^{total}(t_0,t_1)$ selon une descente de gradient :

$$\begin{cases} \Delta w_{ij} = -\alpha \dfrac{\partial E^{total}(t_0,t_1)}{\partial w_{ij}} \\ w_{ij}^{\ nouveau} \leftarrow w_{1ij}^{\ ancien} + \Delta w_{ij} \end{cases}$$

Mais la manière de dériver cette équation va être très différente.

Tout d'abord, en remplaçant $E^{total}(t_0,t_1)$ par son expression donnée dans (5), on écrit :

$$\Delta w_{ij} = \sum_{t=t_0+1}^{t_1} -\alpha \frac{\partial E(t)}{\partial w_{ij}}$$

$$\Delta w_{ij} = \sum_{t=t_0+1}^{t_1} \Delta w_{ij}(t)$$

avec $$\Delta w_{ij}(t) = -\alpha \frac{\partial E(t)}{\partial w_{ij}}$$

Il suffit donc de calculer $\Delta w_{ij}(t)$. Pour ce faire, remplaçons $E(t)$ par son expression donnée dans (4) :

$$\Delta w_{ij}(t) = -\alpha \sum_{k \in U} e_k(t) \frac{\partial y_k(t)}{\partial w_{ij}} = -\alpha \sum_{k \in U} e_k(t) p_{ij}^k(t) \qquad (9)$$

avec $$p_{ij}^k(t) = \frac{\partial y_k(t)}{\partial w_{ij}}$$

Il faut maintenant calculer $p_{ij}^k(t)$:

$$p_{ij}^k(t+1) = \frac{\partial y_k(t+1)}{\partial w_{ij}} = f'\left(s_k(t+1)\right) \frac{\partial s_k(t+1)}{\partial w_{ij}}$$

$$p_{ij}^k(t+1) = f'\left(s_k(t+1)\right) \left[\sum_{l \in U \cup I} w_{kl} \frac{\partial x_l(t)}{\partial w_{ij}} + \delta_{ik} x_j(t) \right]$$

où δ_{ik} dénote le delta de Kronecker :

$$\begin{cases} \delta_{ik} = 1 \text{ si } i = k \\ \delta_{ik} = 0 \text{ si } i \neq k \end{cases}$$

Comme :

$$l \in U \Rightarrow \frac{\partial x_l(t)}{\partial w_{ij}} = \frac{\partial y_l(t)}{\partial w_{ij}} = p_{ij}^l(t)$$

$$l \in I \Rightarrow \frac{\partial x_l(t)}{\partial w_{ij}} = \frac{\partial x^e{}_l(t)}{\partial w_{ij}} = 0$$

On obtient finalement :

$$p_{ij}^k(t+1) = f'\left(s_k(t+1)\right)\left[\sum_{l \in U} w_{kl} p_{ij}^l(t) + \delta_{ik} x_j(t)\right] \tag{10}$$

Les variables $\left\{p_{ij}^k\right\}$ forment donc un système dynamique dont on peut calculer les valeurs successivement dans le temps, en prenant $p_{ij}^k(0) = 0$ pour condition initiale. Ces variables s'appellent variables de sensibilité.

Contrairement à l'équation (8) de la rétropropagation dans le temps, on n'a pas de variable dont la valeur au temps *t* dépende de sa valeur au temps *t+1*. Les calculs d'apprentissage s'effectuent donc en même temps que les activités du réseau sont calculées. A la fin de la séquence, les poids peuvent être adaptés immédiatement selon la formule (6b).

Cependant, le calcul des $\left\{p_{ij}^k\right\}$ est coûteux car il s'agit d'une matrice à trois dimensions : pour un réseau à *n* automates entièrement connectés et *m* entrées, il y a n^2+mn poids w_{ij} et n^3+mn^2 variables p_{ij}^k, qui seront autant de variables supplémentaires à stocker. Comme de plus l'équation (10) contient une somme de *n* termes, la quantité d'opérations effectuées variera avec n^4.

L'algorithme *BPTT* est donc bien plus économique. Mais tout l'intérêt de l'algorithme *RTRL* réside dans sa variante immédiate dans laquelle les paramètres du réseau sont adaptés "en ligne".

En effet, puisqu'à chaque instant *t* on calcule une contribution $\Delta w_{ij}(t)$ au terme Δw_{ij}, on peut directement effectuer la modification des poids selon la formule :

$$w_{ij}(t) = w_{ij}(t-1) + \Delta w_{ij}(t)$$

L'algorithme *RTRL* permet ainsi de modifier les poids au cours de la simulation sans avoir besoin de stocker explicitement les activités du réseau sur une certaine fenêtre temporelle, comme c'est le cas pour la rétropropagation dans le temps.

Une variante intéressante a été proposée dans [Zipser 89]. Afin de diminuer les coûts computationnels, on peut regrouper les unités par blocs, et considérer que chaque bloc est indépendant, c'est-à-dire que l'erreur calculée au niveau d'une unité d'un bloc ne se propage pas dans les autres blocs. Ainsi, la matrice des variables p_{ij}^k contient beaucoup de zéros, ce qui simplifie les calculs. Une contrainte dans cette stratégie de regroupement est bien sûr d'avoir par bloc au moins une unité où un signal désiré est donné, afin que chaque bloc reçoive un signal d'erreur. Il s'agit d'une approximation de la descente de gradient rigoureuse, qui est efficace dans un certain nombre de cas [Zipser 89]. Le choix des blocs est laissé à l'utilisateur du réseau, qui devra faire le compromis entre un algorithme approximatif mais rapide et un algorithme plus exact mais plus lent.

3. Analyse et difficultés de ces réseaux

3.a Coûts computationnels

Deux caractéristiques sont généralement étudiées : le stockage et la quantité de calculs, que l'on mesure par le nombre de "réels" (nombres décimaux) stockés et le nombre d'opérations arithmétiques respectivement. On évalue les *ordres de grandeur* de ces quantités, selon la notation "O", qui se définit par :

f(x) est O(g(x)) si $\exists$ y0, c>0 tels que $\forall$ y>y0, f(y)<cg(y)

Dans un premier temps, on peut prendre un réseau entièrement connecté à *M* entrées et *N* automates, simulé pendant *T* instants successifs. On obtient le tableau comparatif suivant :

	BPTT	BPTT en ligne	BPTT tronquée	BPTT par blocs	RTRL
Nombre d'opérations arithmétiques	N^2T	N^2T^2	N^2Th	N^2Th/h'	$(N^4+MN^3)T$
Nombre de réels stockés	NT	NT	Nh	Nh	N^3+MN^2

Ce tableau signifie que le coût computationnel est en $O(N^2T)$ pour la *BPTT*, $O((N^4+MN^3)T)$ pour *RTRL*, etc. Les performances opposées des algorithmes *BPTT* et *RTRL* apparaissent clairement : *BPTT* est deux ordres de grandeur moins coûteux en calculs, mais s'il doit être utilisé en ligne sans tronquer la rétropropagation, le temps de calcul devient proportionnel au carré de la taille de la séquence, ce qui le rend complètement inadapté à ce type d'utilisation.

Sur le plan du stockage, *BPTT* a besoin d'un stockage temporel (ce qui le rend inadapté aux longues séquences) alors que *RTRL* a besoin d'un stockage spatial (ce qui le rend inadapté aux grands réseaux).

Dans un second temps, si l'on considère une structure plus simple, où il y a seulement $O(N)$ poids (structures en ligne, en anneau [Young & Chan 93]), alors on obtient le tableau suivant :

	BPTT	BPTT en ligne	BPTT tronquée	BPTT par blocs	RTRL
Nombre d'opérations arithmétiques	NT	NT^2	NTh	NTh/h'	N^2T
Nombre de réels stockés	NT	NT	Nh	Nh	N^2

Le handicap de *RTRL* est donc moins grand dans ce cas.

Il faut cependant noter que ce type de comparaison manque de précision [Logar 93]. En effet, les comparaisons ci-dessus n'ont de sens que quand *N* est grand. Or la plupart des applications ne vérifient pas cette condition.

Dans [Logar 93], les coûts computationnels sont estimés plus finement en tenant compte du coût relatif des différentes opérations (multiplications, division, etc.). Pour l'algorithme *BPTT*, on trouve comme coût de calcul :

$$7\,N^2T + 71\,NT + 2\,N^2$$

et pour l'algorithme *RTRL* :

$$(N^4 + 5\,N^3 + 6\,N^2 + 30\,N + 2N^3M + 5\,N^2M + 6\,NM)\,T$$

Enfin, remarquons que l'algorithme *RTRL* se parallélise bien, dans la mesure où l'expression (10) constitue un système de N^2 équations pouvant se calculer en parallèle. Le temps de calcul peut ainsi être réduit de $O(N^2)$ dans le cas d'une structure entièrement connectée.

3.b Non-localités spatiales

Un autre axe de comparaison entre *RTRL* et *BPTT* concerne la "non-localité".

L'algorithme *RTRL* est souvent qualifié de non local, dans la mesure où chaque poids doit avoir accès non seulement à l'ensemble de la matrice des poids (équation (10)) mais aussi au vecteur d'erreur $e_k(t)$ (équation (9))[Williams & Zipser 89a]. Il semble donc y avoir deux non-localités. Ce n'est pas le cas pour l'algorithme *BPTT*.

Dans [Schmidhuber 90], on dénote "localité spatiale", par opposition à "localité temporelle", la propriété selon laquelle la complexité des calculs par connexion dans une unité de temps est bornée (est $O(1)$), quelle que soit la taille du réseau. Cette formulation de la localité se ramène totalement à une comparaison computationnelle (et non structurelle) du type de celle proposée dans les deux tableaux du chapitre précédent (dans lesquels on diviserait les valeurs par N^2T).

Au delà de la disparité des définitions, le concept de non-localité apparaît en contradiction avec le fondement même des réseaux de neurones; en effet, selon [Hecht-Nielsen 91] (traduit par nos soins), dans un réseau de neurones, tout traitement effectué par une unité (en relaxation ou en apprentissage) "doit dépendre seulement des valeurs des signaux entrant dans l'unité via les connexions entrantes et des valeurs qui y sont stockées". L'algorithme *RTRL* ne serait-il pas un réseau de neurones ? ("réseau de neurones" étant pris dans son acceptation informatique.)

Le paradoxe provient tout simplement de la "tradition" connexionniste qui consiste à ne pas expliciter le réseau d'apprentissage, mais uniquement le réseau de propagation (voir chapitre I.3.5). Si un poids, à un endroit du réseau, doit avoir accès à un paramètre ailleurs dans le réseau, alors cela signifie, si l'on veut rester dans un cadre connexionniste, qu'il y a une connexion entre les deux points du réseau. Nous introduisons alors deux définitions :

- un apprentissage est local si le graphe du réseau fonctionnant en phase d'apprentissage est le même que le graphe du réseau en relaxation ;
- un apprentissage est strictement local si le graphe orienté du réseau fonctionnant en phase d'apprentissage est le même que le graphe orienté du réseau en relaxation.

Notons que selon cette définition, l'algorithme de rétropropagation du gradient dans un réseau à couches est local, mais pas strictement local ; il en est de même pour l'algorithme *BPTT*.

La question est donc d'exprimer clairement l'algorithme *RTRL* sous forme d'un réseau afin de pouvoir observer sa connectivité, et d'identifier quels traitements rendent l'apprentissage non local.

On remarque que la formule (10) du paragraphe 2.b décrit le comportement d'un réseau de N unités ayant, pour (i,j) fixé, les activités $\left\{p_{ij}^1, p_{ij}^2, \ldots, p_{ij}^N\right\}$ et une entrée $x_j(t)$. Il y a autant d'équations (10) que de poids adaptables, donc autant de réseaux supplémentaires que de poids adaptables, soit $(M+N)M$ dans le cas entièrement connecté.

La formule (9) montre que tout poids adaptable reçoit une information des unités où une erreur $e_k(t)$ peut être calculée, ainsi que certains signaux p. On peut aussi considérer, et c'est la solution que nous adoptons, que des unités supplémentaires calculent le produit $v_{ij}^k(t) = e_k(t)p_{ij}^k(t)$, dont les sorties sont envoyées au poids correspondant.

La figure 5 illustre le réseau qui apprend, pour une structure particulière donnée comportant cinq automates partiellement connectés dont une entrée et deux sorties (le poids sur $x0$ est supposé fixé à 1). La complexité du réseau qui apprend explique certainement en partie pourquoi il n'est jamais représenté. Le réseau initial est répliqué autant de fois que de paramètres adaptables, chaque réseau recevant une activité

du réseau initial (flèches rectilignes en diagonale) et les signaux $f'(s_i(t))$ (flèches en pointillés fins). Autant d'unités que de sorties désirées (on suppose $T(t)$ introduit dans la formule (3) constant) sont ajoutées pour calculer $e_k(t)$. Des unités sont aussi ajoutées pour calculer les produits $v_{ij}^k(t) = e_k(t)p_{ij}^k(t)$; les sorties de ces unités sont connectées vers les automates où sont stockés les poids (flèches en gras), qui doivent ensuite être transmis à tous les automates correspondants des réseaux supplémentaires (flèches en pointillés fins à nouveau).

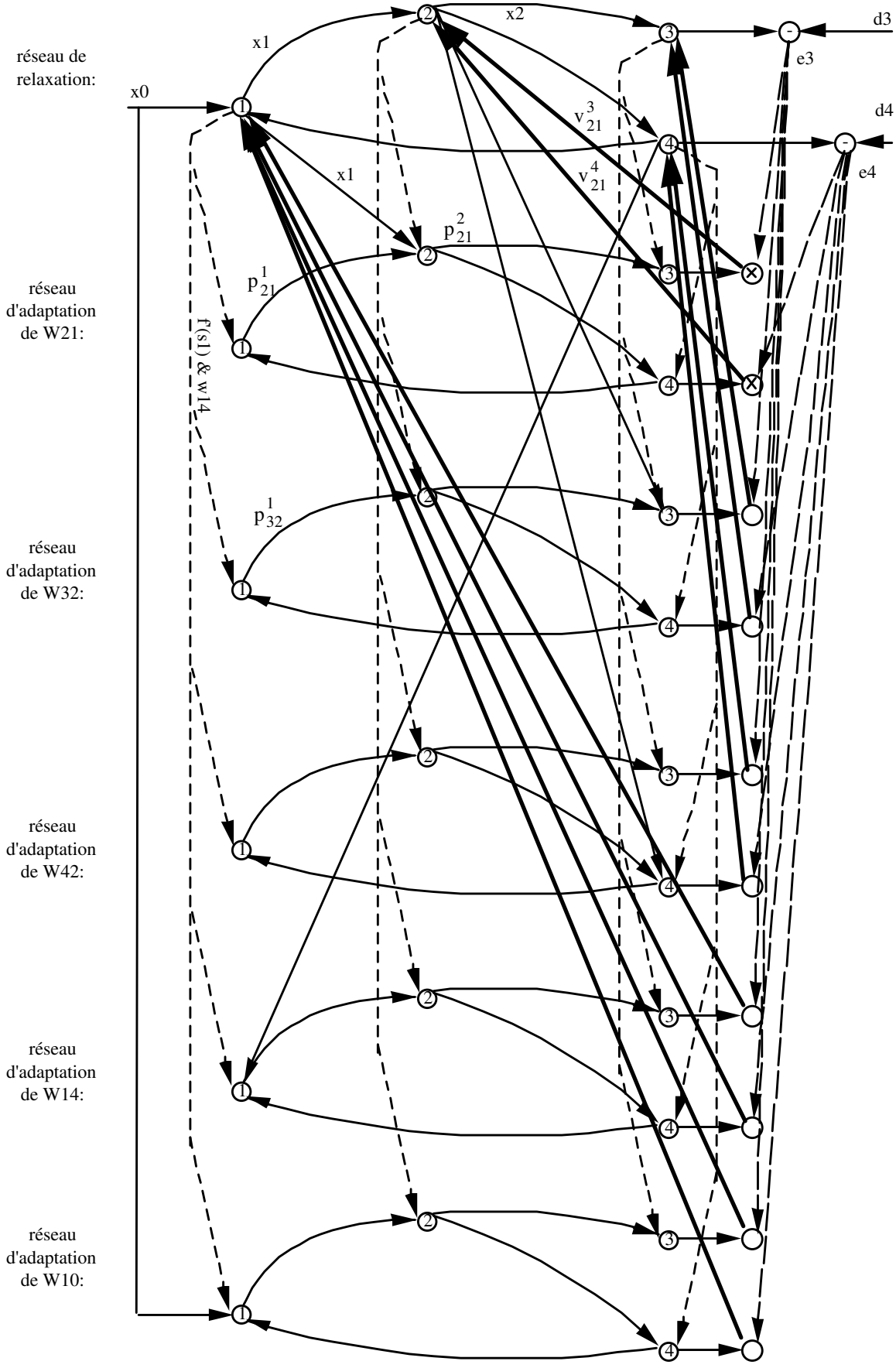

Figure 5. Représentation complète de l'algorithme RTRL

Une telle représentation n'est pas unique et dépend essentiellement du grain de description adopté ; par exemple, l'ajout des automates qui calculent les valeurs $v_{ij}^{k}(t)$ n'était pas obligatoire.

En particulier, une description en macro-unités de l'algorithme *RTRL* est intéressante à développer. Il s'agit, à partir du schéma 5, de former des colonnes, chaque colonne regroupant les automates ayant la même position dans le réseau initial et les réseaux ajoutés du schéma 5.

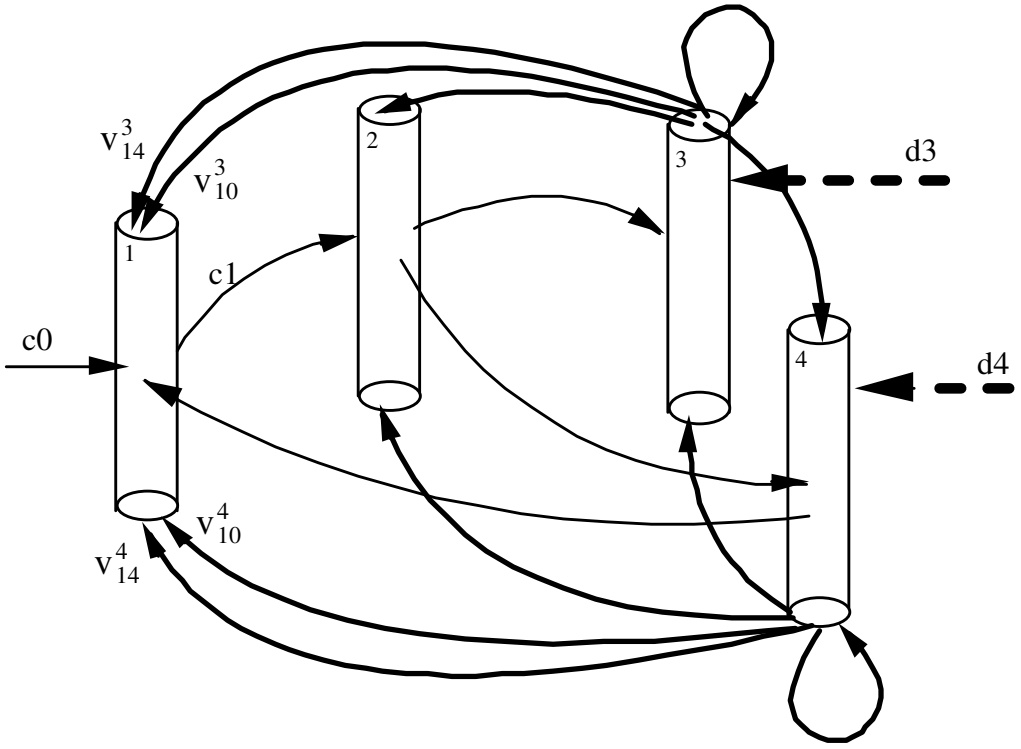

Figure 6. Réseau de colonnes pour l'algorithme RTRL

La figure 6 explicite ce réseau à colonnes. Les automates de ce nouveau réseau échangent d'une part les signaux $v_{ij}^{k}(t)$ (flèches en gras) et d'autre part des signaux multiples, c'est-à-dire des vecteurs, que l'on dénote c_i, i désignant l'automate d'où part le signal ; dans l'exemple ci-dessus, les signaux c_i sont :

$$c_i = (x_i, p_{21}^{i}, p_{32}^{i}, p_{42}^{i}, p_{14}^{i}, p_{10}^{i})$$

On constate alors que :

- les calculs des $p_{ij}^{k}(t)$ utilisent le même graphe orienté que les calculs des $x_i(t)$;
- l'apprentissage est non local à cause des variables $v_{ij}^{k}(t)$ c'est-à-dire du fait des calculs de $w_{ij}.(t)$ selon la formule (9).

Selon le réseau en colonnes, il n'y a donc qu'une seule source de non-localité, contrairement au réseau représenté sur la figure 5. Parmi les deux non-localités classiquement évoquées concernant l'algorithme *RTRL* [Williams & Zipser 89a], l'une d'elle est finalement *réductible* si l'on regroupe les traitements en colonnes. Il ne semble par contre pas possible de réaliser un tel groupement qui supprimerait la non-localité engendrée par l'équation (9).

Remarquons tout de même que dans chaque macro-unité de la figure 6, il y autant de variables *p* que de paramètres adaptables dans *tout* le réseau ; en conséquence, la *construction* du réseau doit tenir compte, en chaque unité, du réseau dans sa globalité.

On notera que l'exercice auquel nous venons de nous prêter, à savoir l'explicitation de toutes les connexions du réseau, a mis en évidence deux lacunes dans la théorie connexionniste : d'une part, le réseau d'apprentissage est souvent omis ; d'autre part, à une description en termes équations correspondent plusieurs descriptions sous forme de réseau.

3.c algorithmes "temps réels" ?

Les réseaux récurrents ci-dessus décrits ont des comportements temporels et peuvent se modifier en fonction des données issues de l'environnement.

La manière dont s'effectuent ces modifications fait l'objet de nombreuses classifications dichotomiques des algorithmes en question : temps réel/temps différé [Williams & Zipser 89a,b], adaptatifs/non adaptatifs [Nerrand *et al.* 93], "on-line"/"off-line" [Williams & Peng 90][Piché 94], local dans le temps ou non [Schmidhuber 90].

Toutes ces notions se recoupent mais ne sont pas équivalentes. Nous proposons donc de les définir avec précision.

Algorithmes "en ligne"

Il convient tout d'abord de distinguer les algorithmes qui autorisent une modification des paramètres *pendant* l'exécution du réseau, par opposition aux algorithmes où la modification des poids ne peut s'effectuer qu'à la *fin* de la séquence générée par le réseau. Dans le premier cas, il s'agit d'algorithmes "en ligne" ("on-line") [Williams & Peng 90][Piché 94], dans le second d'algorithmes "off-line", par epochs ("epochwise" [Piché 94]).

Applications temps réel

Par ailleurs, l'adjectif "temps réel" est souvent utilisé en même temps que "on-line" à propos de tel ou tel algorithme. Cependant, l'aspect temps réel n'est en fait pas relatif à un algorithme en tant que tel mais à une *application*, c'est-à-dire l'utilisation d'un algorithme pour un système physique donné, que ce soit pour le visualiser, l'identifier, le contrôler, etc. Un algorithme n'est donc pas temps réel, mais fonctionne en temps réel sur un certain système physique. Une application est dite temps réel si les calculs s'effectuent au moins aussi vite que le système physique concerné.

Cela amène à échantillonner le système physique, c'est-à-dire à saisir les données de l'environnement tous les τ unités de temps. Une application sera temps réel si elle effectue ses calculs relatifs à l'échantillon courant en un temps $\theta(t)$ inférieur à τ ("temps réel" ne veut donc pas dire "rapide", puisque τ peut valoir 15 minutes). Il convient de souligner que tout algorithme d'apprentissage "en ligne" peut fonctionner en temps réel : pour une séquence finie (durée de vie du système) la durée de calcul maximale $\theta = \max\{\theta(t)\}$ d'une modification des paramètres est nécessairement bornée, et l'algorithme peut donc fonctionner en temps réel pour tout système tel que $\tau > \theta$. Ainsi, même l'algorithme de "rétropropagation dans le temps en ligne" (voir paragraphe 2.a) peut fonctionner en temps réel.

Notons qu'un algorithme "en ligne" peut ne pas fonctionner en temps réel, si il modifie ses paramètres en cours d'exécution mais avec un *retard* par rapport au système réel ($\theta > \tau$).

Optimisation des calculs

Il apparaît que si l'on utilise un algorithme tel la "rétropropagation dans le temps en ligne", pour laquelle $\theta(t)$ est de l'ordre de t, alors les calculs sont concentrés à la fin de la séquence, et la puissance de calcul sera en moyenne sous utilisée. Autrement dit, pour une puissance de calcul donnée, la constante de temps τ à partir de laquelle on pourra utiliser cet algorithme en temps réel va augmenter avec la durée de la séquence. Une contrainte pratique essentielle est donc de travailler avec des algorithmes tels que l'ordre de grandeur de θ soit borné et ne dépende pas de t, soit θ=O(1). Ceci est exactement la définition donnée dans [Schmidhuber 91] de la "localité dans le temps". Cette définition est proche de celle d'un système adaptatif donnée dans [Nerrand *et al.* 93] : un système adaptatif est un système entraîné en continu

(équivalent de “en ligne”) et utilisé sur un nombre infini d’échantillons. En effet, si la séquence est infinie, l’ordre de grandeur de θ doit être borné.

De manière plus précise, on aura tendance à rechercher des algorithmes dont la répartition des calculs est uniforme ($\theta(t) = constante$) afin d’utiliser toute la puissance de calcul à chaque instant, dans le cas où l’échantillonnage est lui aussi uniforme.

Enfin, pour une application temps réel rapide, on s’intéressera à réduire le plus possible θ, non seulement en ordre de grandeur mais aussi en valeur. Ainsi, pour la rétropropagation dans le temps par blocs (paragraphe 2.a), on cherchera à minimiser h et h' tout en gardant une bonne approximation du gradient. Dans le cas d’une implémentation parallèle, l’algorithme *RTRL* est le plus rapide, puisque $\theta(t)$ est sensiblement égal au temps de calcul des activités du réseau.

3.d problèmes de convergence

Les “problèmes de convergence” désignent les situations où la dynamique des poids du réseau ne conduit pas à une solution satisfaisante. Ces situations sont nombreuses dans les réseaux récurrents. Nous passons d’abord en revue quatre des problèmes qui existent aussi dans les réseaux à couches, pour ensuite aborder des problèmes spécifiques des réseaux récurrents.

Un grand nombre de résultats théoriques sur les réseaux concernent l’*existence* d’un réseau pouvant réaliser telle fonction. Ainsi, pour les réseaux récurrents, on montre que toute trajectoire peut être approchée avec une précision arbitraire par un réseau récurrent entièrement connecté [Li 92]. De même, on montre dans [Kremer 95] que le "Simple Recurrent Network" [Elman 90] peut approcher avec une précision arbitraire n'importe quel automate à états finis.

Ce type de résultats ne prédit pas si l’on peut trouver les paramètres du réseau en question par apprentissage. Le mode d’apprentissage souvent choisi, qui consiste à utiliser le gradient d’un critère d’erreur, garantit uniquement que sous certaines conditions, l’on peut aboutir à un minimum (local ou global) de la surface d’erreur.

La première source de problème réside donc dans le fait que le minimum atteint peut donc correspondre à une erreur élevée (minimum local). Dans de nombreuses utilisations des réseaux à couches, on se satisfait de cette situation, estimant que les minima locaux sont rares ou correspondent à une erreur assez faible. Cependant, d’après [Piché 94], les problèmes de minima locaux à valeur élevée seraient plus fréquents dans les réseaux récurrents que dans les réseaux à couches ; nos propres simulations confirment ce résultat [Szilas 95]. Certains résultats théoriques établissent des conditions dans lesquelles un réseau récurrent n’a pas de minimum local [Bianchini *et al.* 94].

Deuxièmement, la convergence vers un minimum n’est assurée que pour un pas infiniment petit, ce qui, dans la pratique, n’est pas réalisable. En fait, au contraire, le concepteur d’algorithmes neuronaux maximise le pas, afin de rendre l’algorithme plus rapide. On n’est alors plus du tout dans les conditions de validité de l’algorithme.

Troisièmement, la surface d’erreur comporte un certain nombre de *pièges* [Le Cun 87] en particulier des plateaux : la surface d’erreur est presque horizontale et les paramètres n’évoluent donc pratiquement pas. Empiriquement, l’effet est le même que dans le cas d’un minimum local : l’apprentissage devient de plus en plus lent.

Quatrièmement, la convergence vers un minimum se fonde sur un formalisme *continu* alors qu’au niveau de l’ordinateur, seules des valeurs *discrètes* sont manipulées. La convergence vers un minimum n’est donc pas garantie, sur un ordinateur actuel. Par exemple, au niveau d’un plateau, le gradient est très faible et peut s’annuler, alors qu’il ne s’agit pas d’un minimum local. Une erreur d’arrondi peut aussi faire remonter l’erreur.

Nous abordons maintenant les problèmes plus spécifiques des réseaux récurrents ; tous se manifestent lorsque le réseau manipule de longues séquences.

Tout d'abord, un algorithme qui adapte les paramètres en ligne obtenu à partir d'un algorithme qui adapte les paramètres en fin de séquence ne suit pas la direction exacte du gradient [Williams & Zipser 89a]. En effet, ce type d'algorithme minimise une erreur qui est fonction des poids du réseau : ces poids sont implicitement supposés fixes. Or dans les versions en ligne, ces poids sont modifiés au cours du déroulement de la séquence. Ce n'est donc plus le gradient de l'erreur calculée avec les poids initiaux fixes qui est évalué.

D'après [Williams & Zipser 89a] à propos de l'algorithme *RTRL*, on évite ce problème si la variation des poids est bien plus lente que celle des activités, et il suffit donc de choisir un pas d'apprentissage suffisamment petit. Or, comme on le fait remarquer dans [Catfolis 93], même avec un pas d'apprentissage très faible, les poids finissent toujours par varier de manière importante, pour une séquence suffisamment longue, et l'algorithme devient faux. Autrement dit, l'algorithme RTRL n'est valide que si les poids évoluent peu, ce qui restreint fortement son utilisation. Bien sûr, on peut présenter plusieurs fois la même séquence avec un pas suffisamment petit pour que sur chaque présentation les poids évoluent peu et faire ainsi fortement varier les poids sans trop s'éloigner de la direction du gradient, mais l'algorithme *RTRL* n'est alors plus utilisé en continu, et l'algorithme *BPTT* est dans ce cas préférable.

Un dernier problème relatif aux longues séquences est le problème de "l'explosion du gradient". Dans un certain nombre de situations, la dérivée de l'erreur instantanée $E(t)$ selon un poids du réseau peut devenir très importante au fur et à mesure que la simulation se prolonge. Cette explosion du gradient pose en particulier des problèmes quand on dépasse les capacités de codage de l'ordinateur. Nous avons pu relever deux situations où les gradients explosent.

La première situation se réfère au problème des bifurcations. Certains systèmes dynamiques sont tels qu'en certains points de l'espace des paramètres, un variation infinitésimale d'un paramètre peut aboutir à un comportement totalement différent à l'infini [Pearlmutter 90][Doya 94][Crucianu 94]. Pour certains algorithmes, comme celui décrit dans [Pineda 87], qui sont fondés sur la dérivée de l'état à l'infini par rapport aux poids, un problème théorique évident se pose : cette dérivée s'avère ne pas être définie en certains points [Pearlmutter 90][Doya 94]. Pour les apprentissages de séquences *finies*, l'erreur est dérivable partout (c'est une somme finie de termes dérivables), mais le gradient peut croître rapidement [Doya 94][Crucianu 94].

La deuxième situation concerne les systèmes oscillatoires, pour lesquels le problème ne se rencontre pas uniquement pour des points de bifurcation particuliers. Si on entraîne un réseau à osciller, une très faible modification d'un poids va entraîner un décalage de phase et donc fortement modifier l'erreur à un instant t grand, dans le cas habituel de l'erreur quadratique.

Supposons ainsi qu'un réseau soit entraîné à produire un signal $y(t) = \sin(\omega t)$. Supposons que ω soit explicitement codé dans l'un des paramètres w du réseau, ce qui n'est pas le cas, mais nous fournit une première idée. Alors on aurait :

$$\frac{\partial y(t)}{\partial w} = t\cos(wt)$$

Donc le gradient oscillerait avec une amplitude de plus en plus élevée. Nous avons vérifié ce comportement en entraînant avec *RTRL* un réseau de 12 automates entièrement connectés sur une trajectoire circulaire centrée en *0,5* et de rayon *0,4*, tâche qui a été étudiée dans [Pearlmutter 89] (les unités sont à fonction de transfert sigmoïde entre *0* et *1*, les poids initialisés entre *-0,5* et *0,5*, le réseau entraîné sur *19* points - un cercle entier comportant *16* points- pendant *10000* présentations). La dérivée de l'erreur instantanée par rapport à un poids arbitraire du réseau est représentée sur la figure 7. On trouve effectivement une évolution non bornée du gradient : les maxima de cette courbe croissent linéairement.

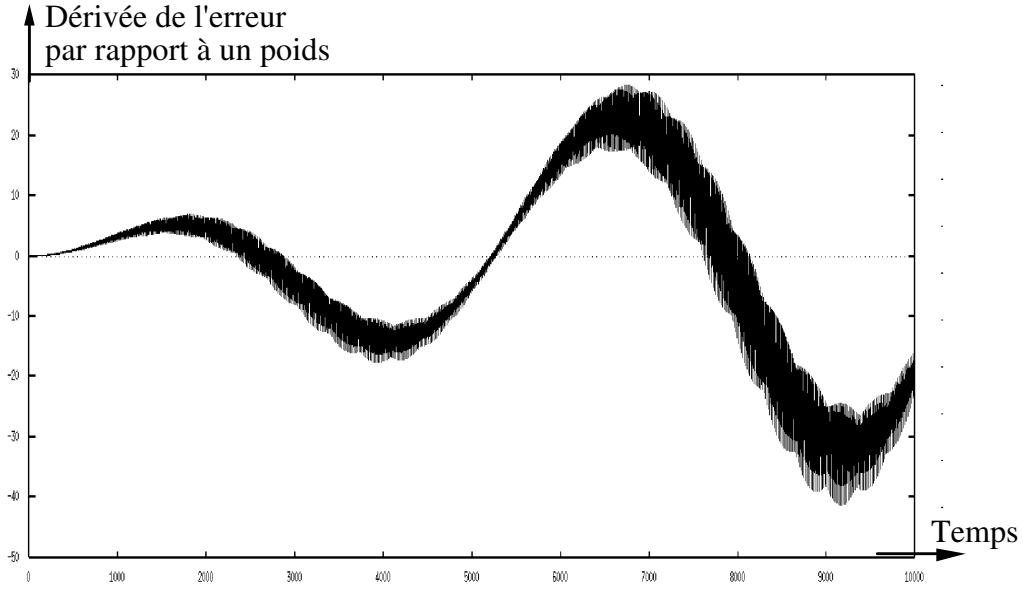

Figure 7. Explosion du gradient

4. Variantes et améliorations existantes

4.a Forçage du réseau

Une variante est très souvent associée à l'algorithme *RTRL*, que l'on nomme "teacher forcing", *forçage* en français. Nous en décrivons d'abord les aspects algorithmiques, pour ensuite aborder les aspects cognitifs que sous-tendent certaines formules exposées.

Le forçage consiste à obliger le réseau à suivre la trajectoire désirée, même si naturellement il s'en écarte. Le comportement désiré $d_i(t)$ qui provient de l'environnement sert non seulement à calculer une erreur, selon la formule (3) du paragraphe 2.a, mais est aussi utilisé pour modifier les activités des unités où un comportement désiré est fourni, selon la formule :

$$x_i(t) = \begin{cases} x^e{}_i(t) & \text{si } i \in I \\ d_i(t) & \text{si } i \in T(t) \\ y_i(t) & \text{si } i \in U - T(t) \end{cases}$$

Cette expression remplace l'expression (1) du paragraphe 2.a. De plus, les termes $p_{ij}^l(t)$ sont mis à zéro après la modification des poids selon (9) (voir paragraphe 2.b), ce qui revient à remplacer l'équation (10) par :

$$p_{ij}^k(t+1) = f'\left(s_k(t+1)\right)\left[\sum_{l \in U-T(t)} w_{kl} p_{ij}^l(t) + \delta_{ik} x_j(t)\right]$$

Cette variante évite que le comportement du réseau ne s'éloigne trop du comportement désiré, et est surtout intéressante pour les longues séquences [Crucianu 94]. Dans certains cas, l'algorithme ne converge vers une bonne solution que si cette variante est utilisée [Williams & Zipser 89a,b]. En particulier, le forçage permet d'éviter l'explosion du gradient [Doya 94]. Nous l'avons vérifié en répliquant l'expérience décrite à la fin du paragraphe 3.d (voir aussi la figure 7) : avec le forçage, le gradient reste borné.

En fait, si le *terme* "teacher forcing" est introduit dans [Williams & Zipser 89a], cette technique est empruntée au domaine du contrôle, et se dénote alors "series parallel model" [Narendra & Parthasarathy 90] ainsi qu'au domaine du filtrage adaptatif et se dénote alors "equation error" [Nerrand *et al.* 93]. Dans [Nerrand *et al.* 93], les versions forcées et non forcées de *RTRL* sont replacées dans le cadre général du filtrage adaptatif, ce qui permet d'envisager plusieurs autres variantes.

Il faut préciser que le forçage du réseau peut être utilisé quel que soit l'algorithme d'apprentissage du réseau récurrent (algorithmes *RTRL* et *BPTT*, apprentissage du point fixe [Pineda 88 *in* Crucianu 94]), à condition bien sûr que l'automate de sortie dont on force la valeur fasse partie d'un circuit dans le graphe d'interconnexion.

Une autre alternative pour forcer le réseau a été proposée dans [Toomarian & Barhen 91,92], que l'on peut nommer "forçage par l'erreur". Au lieu d'affecter les valeurs désirées aux activités du réseau, on ajoute une entrée supplémentaire au réseau dont la valeur est liée à l'erreur commise par le réseau. La formulation originale est en formalisme continu ; en formalisme discret, on obtient, à la place du premier terme de l'équation (2) :

$$s_i(t+1) = \sum_{j \in U \cup I} w_{ij} x_j(t) + \lambda \left(\left[d_i(t) \right]^{1-\beta} \left[d_i(t) - y_i(t) \right]^{\beta} \right)$$

avec λ et β deux nombre positifs.

De plus, les auteurs proposent de diminuer λ au cours du temps (λ est modifié uniquement entre deux présentations de la séquence entière), selon la loi :

$$\lambda(\tau) = 1 - e^{-E^{total}(\tau)}$$

avec τ le numéro de la présentation de la séquence entière.

Expérimentalement, les auteurs obtiennent de bons résultats, mais difficilement exploitables car d'une part le forçage est combiné avec une modification de l'algorithme même d'adaptation des poids et d'autre part le nouveau type de forçage n'est pas comparé avec l'algorithme de forçage classique.

De manière générale, l'efficacité du forçage n'est pas garantie du point de vue théorique [Jodouin 93]. Au niveau expérimental, les travaux décrits dans [Williams & Zipser 89a] reviennent à constater que sur certains problèmes (apprentissage d'oscillations), le forçage est indispensable (ce qui est confirmé dans [Tsung 90]) sans systématiquement comparer les versions avec et sans forçage pour *tous* les problèmes, comme on le fait remarquer dans [Hecht-Nielsen 91]. De même, aucune étude expérimentale n'a été réalisée à notre connaissance pour comparer les deux algorithmes de forçage ci-dessus décrits.

Le principal problème du forçage est le suivant : *l'erreur mesurée pendant l'exécution forcée n'est pas identique à l'erreur mesurée quand le réseau s'exécute librement* : elle est en l'occurrence plus faible. Ainsi, une erreur très faible en cours d'apprentissage peut finalement donner une mauvaise solution (voir [Tsung 90]).

Enfin, nous souhaitons maintenant aborder les aspects cognitifs de l'algorithme de forçage. En effet, dans [Toomarian & Barhen 91, 92], sans qu'il soit question de modélisation, on insiste particulièrement sur l'inspiration provenant de données humaines dans la conception de l'algorithme de forçage. Selon [Toomarian & Barhen 92] (traduit en français pour le présent document) :

> *"supposez qu'un père veuille apprendre à son enfant à faire du vélo. Bien sûr, le père ne va pas rester à la maison, laisser son enfant faire du vélo, et, de temps en temps, lui dire dans quelle mesure les performances sont bonnes ou mauvaises (comme cela se passe en apprentissage supervisé classique). La meilleure méthode serait pour le père d'accompagner son enfant pendant l'apprentissage"*

Comparer l'algorithme de forçage et l'apprentissage moteur nous apparaît très intéressant. Cependant, les termes précis dans lesquels l'analogie est effectuée sont à discuter.

D'une part, l'apprentissage supervisé classique ne correspond pas à un jugement donné "de temps en temps" mais au contraire le signal désiré $d_i(t)$ peut être présent à tout instant.

D'autre part, la position imposée par le père n'est pas nécessairement la position désirée, car il y a une certaine souplesse dans le forçage. L'analogie ne vaut donc que pour le forçage par l'erreur, et non

pour la version originale du forçage [Williams & Zipser 89a].

Ensuite, l'attitude du parent n'est pas uniforme avec le temps.

Ce dernier point a amené les auteurs à proposer l'analogie suivante :

> *"Quand un père apprend à son enfant à faire du vélo, dans les premières étapes il garde les mains sur le vélo pour accompagner son enfant. Cependant, dès que l'enfant fait preuve d'une certaine habileté pour contrôler ses mouvements, le père retire ses mains de plus en plus souvent, afin de laisser l'enfant avancer tout seul."*

Cette analogie a amené les auteurs à proposer la formule $\lambda(\tau) = 1 - e^{-E^{total}(\tau)}$ ci-dessus. Cependant, réaliser algorithmiquement ce qui est cité ci-dessus à propos de l'apprentissage humain conduirait à faire varier la *fréquence* du forçage, comme il est suggéré dans [Tsung 90], et non l'*intensité* du forçage (paramètre λ).

Pour finir, précisons que la réelle interaction parent-enfant dans l'exemple de l'apprentissage du vélo est certainement bien plus sophistiquée qu'un simple forçage élémentaire. A ce titre, le terme de "*teacher* forcing" est évidemment abusif, ce qui explique que nous l'ayons seulement traduit par "forçage". Nous estimons malgré tout que cette analogie mérite d'être creusée : les expériences concernant le guidage chez l'homme (voir [Holding 70] pour un état de l'art), certes rares et anciennes, peuvent peut-être aider à mieux maîtriser le forçage dans les réseaux, et réciproquement.

4.b Elagage de l'historique

Même si, pour une certaine clarté mathématique, les algorithmes d'apprentissage sont exposés sous la forme de la minimisation d'un critère d'erreur calculé sur une séquence entière, voire infinie, un certain nombre de raisons conduisent au contraire à "découper" cette séquence.

En plus des aspects techniques que posent les longues séquences (voir paragraphe 3.d), un raisonnement intuitif invite aussi à "découper" la séquence : si le gradient est calculé sur la séquence entière, il risque alors de contenir des termes calculés alors que les poids étaient totalement différents, termes qui peuvent en fait faire augmenter l'erreur estimée avec les poids actuels.

Il semble donc que dans un apprentissage en ligne, il y ait une fenêtre temporelle à considérer, partant du présent vers le passé. Les versions tronquées et par blocs de l'algorithme *BPTT* (voir paragraphe 2.a) introduisent explicitement cette notion de fenêtre, et peuvent d'ailleurs répondre au problème de l'explosion des gradients [Doya 94][Crucianu 94]. De même, en ce qui concerne l'algorithme *RTRL*, une variante a été proposée dans laquelle les signaux p_{ij}^k sont initialisés à zéro de temps en temps [Catfolis 93] : cette modification rend l'algorithme *RTRL* plus efficace.

On pourrait penser que la version de [Gherrity 89] introduit aussi une fenêtre temporelle pouvant aider l'apprentissage, comme il est écrit dans [Williams & Peng 90], puisque elle utilise comme fonction d'erreur instantanée (adaptée ici au formalisme discret) :

$$E(t) = \frac{1}{2} \sum_{\tau=0}^{t} \lambda^{(t-\tau)} \sum_{i \in U} e_i(\tau)^2$$

mais avec une telle expression de $E(t)$, le critère global qui est minimisé, à savoir $E^{total}(0,T)$, prend en fait plus en compte le début de la séquence que la fin.

L'introduction d'une fenêtre temporelle, qui revient à ignorer les longues dépendances, pose un nouveau problème : comment choisir la taille de cette fenêtre ? Ce problème est d'autant plus aiguë que dans le domaine du traitement de séquences temporelles, les réseaux récurrents ont justement été introduits afin d'éviter de fixer une fenêtre temporelle ; en effet, il est possible d'utiliser des réseaux à couches pour traiter des données temporelles, en "spatialisant le temps", c'est-à-dire en donnant en entrée une fenêtre temporelle du signal [Waibel *et al.* 89].

La réponse classique est de dire qu'il faut choisir la taille temporelle en fonction de ce qu'on connaît du système à identifier ; cette connaissance a priori facilite l'apprentissage. Dans le cas des réseaux

récurrents, cette fenêtre temporelle peut être variable, contrairement à la fenêtre d'un réseau à couches. Dans cet esprit, des idées sont certainement à explorer, par exemple tronquer l'historique quand le gradient devient trop grand, tronquer plus ou moins selon les paramètres, tronquer aléatoirement, etc.

4.c Architectures évolutives

Par opposition aux algorithmes neuronaux classiques, où l'apprentissage réside dans la *modification* des poids des connexions entre les unités, certains algorithmes peuvent apprendre par ajout et/ou suppression d'unités et/ou de connexions, et possèdent donc une architecture évolutive.

Deux grandes classes d'algorithmes sont utilisées [Alpaydin 91] : les algorithmes constructifs (ou incrémentaux), dont la structure est minimale au départ pour ensuite grossir selon les besoins, et les algorithmes destructifs (ou d'élagage) qui, après apprentissage, suppriment les unités et connexions inutiles.

Ces architectures sont beaucoup étudiées pour les réseaux à couches (voir [Alpaydin 91], [Fiesler 94] pour des revues bibliographiques). Elle offrent les avantages suivants :

- optimisation du temps de calcul, puisque une structure minimale, ou presque minimale est obtenue ;
- meilleure généralisation, pour la même raison ;
- facilitation de l'apprentissage dans les réseaux constructifs : au départ, le réseau est de petite taille, donc apprend plus facilement, et est réutilisé dans le réseau plus grand.

Très peu d'études ont été consacrées aux architectures récurrentes évolutives.

Le premier algorithme proposé, de type constructif, est la "Recurrent Cascade Correlation" [Fahlman 91], transformation pour les réseaux récurrents de l'algorithme de "Cascade Correlation" [Fahlman & Lebiere 91] destiné aux réseaux à couches. Afin de pouvoir réaliser cette transformation, on est obligé de se restreindre à une structure limitée, auto-récurrente en cascade (voir figure 8), proche de celle étudiée dans [Mozer 88].

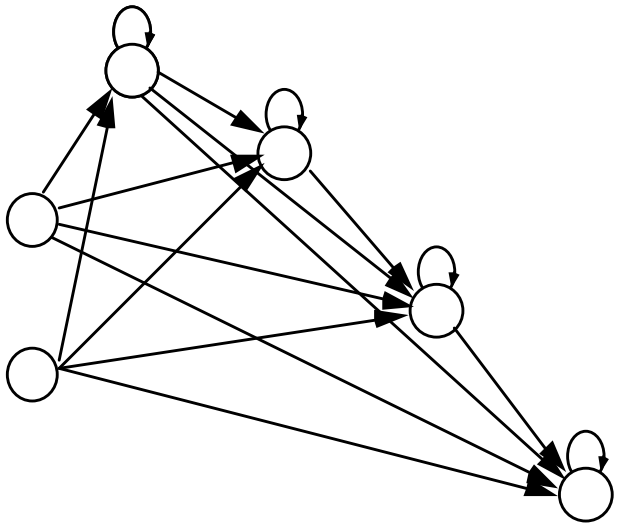

Figure 8. "Recurrent Cascade Correlation"

Ce type de structure est moins général que la structure entièrement connectée, ce qui restreint le champ d'application [Chen *et al.* 93].

Dans [Chen *et al.* 93] est abordé le problème des réseaux constructifs entièrement connectés, mais seul un schéma très rudimentaire de croissance est testé : on rajoute une unité toutes les cinquante présentations. Les réseaux convergent très rapidement sur une application simple.

En ce qui concerne les techniques d'élagage, un algorithme est proposé dans [Omlin & Giles 93] qui s'avère efficace. Dans cet algorithme, les unités inutiles (selon un critère donné) sont successivement retirées du réseau après apprentissage.

La technique classique de décroissance des poids ("weight decay"), qui peut être utilisée à des fins d'élagage, a aussi été testée sur les réseaux récurrents [Kamimura 92][Chen *et al.* 93]. Celle-ci consiste à ajouter dans l'erreur un terme dont la minimisation entraînera une diminution des poids : ainsi, certains poids nuls ou quasi nuls peuvent être supprimés. Les résultats expérimentaux sont positifs [Kamimura] mais moins bons que la technique d'élagage [Chen *et al.* 93].

Malgré ces travaux intéressants, il apparaît que beaucoup reste à faire en ce qui concerne les architectures évolutives récurrentes : expérimentations, nouveaux algorithmes et théorie.

Conclusion

Ce rapport a mis en évidence les algorithmes et caractéristiques des réseaux récurrents. Nous précisions en introduction que cet état de l'art était partiel. Ajoutons qu'il est aussi daté : les recherches sur les réseaux récurrents se multiplient : il n'est qu'à voir le nombre croissant de publications sur le sujet dans les revues internationales. Plusieurs champs disciplinaires s'attaquent au problème (intelligence artificielle, systèmes dynamiques, filtrage adaptatif, etc.) et vont certainement faire sensiblement évoluer le domaine. Souhaitons que sous peu verront le jour des algorithmes à la fois simples, efficaces, plausible sur le plan biologique et théoriquement maîtrisés.

Références

[Alpaydin 91] Ethem Alpaydin. GAL: Networks that grow when they learn and shrink when they forget. Rapport technique, TR 91-032, ICSI, Berkeley, mai 1991.

[Beaufays & Wan 94] Françoise Beaufays & Eric A. Wan. Relating Real-Time Backpropagation and Backpropagation-Through-Time: An Application of Flow Graph Interreciprocity. *Neural Computation*, **6**, p. 296-306, 1994.

[Bianchini *et al.* 92] M. Bianchni, M. Gori & M. Maggini. On the Problem of Local Minima in Recurrent Neural Networks. *IEEE Trans. on Neural Networks*, **5**(2), p. 167-177, mars 1994.

[Catfolis 93] Thierry Catfolis. A method for Improving the Real-Time Recurrent Learning Algorithm. *Neural Networks*, **6**(6), p. 807-821, 1993.

[Chen *et al.* 93] D. Chen, C. L. Giles, G. Z. Sun, H. H. Chen, Y. C. Lee, M. W. Goudreau. Constructive Learning of Recurrent Neural Networks. *Proc. IEEE ICNN*, vol. III, p. 1196-1201, San Francisco, 1993.

[Cleeremans *et al.* 89] Axel Cleeremans, David Servan Schreiber & James L. McClelland. Finite State Automata and Simple Recurrent Networks. *Neural Computation*, **1**, p. 372-381, 1989.

[Crucianu 94] Mihail Crucianu. Représentation structurée dans les réseaux connexionnistes. Thèse de doctorat, Univ. Paris XI Orsay, 1994.

[Doya 94] Kenji Doya. Bifurcations of Recurrent neural networks in Gradient Decent Learning. Submitted to *IEEE Transactions on neural networks*, 1994.

[Elman 90] Jeffrey L. Elman. Finding Structure in Time. *Cognitive Science*, **14**, p. 179-211, 1990.

[Fahlman & Lebiere 91] Scott E. Fahlman & Christian Lebiere. The Cascade-Correlation Learning Architecture. Rapport Technique, CMU-CS-90-100, août 1991.

[Fahlman 91] Scott E. Fahlman. The Recurrent Cascade-Correlation Architecture. Rapport Technique, CMU-CS-91-100, mai 1991.

[Gherrity 89] Michael Gherrity. A learning algorithm for analog fully recurrent neural networks. *Proc. IJCNN*, vol. I, p. 643-644, Washington DC, 1989.

[Giles *et al* 92] C. L. Giles, C. B. Miller, D. Chen, H. H. Chen, G. Z. Sun, Y. C. Lee. Learning and Extracting Finite State Automata with Second Order Recurrent Neural Networks. *Neural Computation*, **4**, p. 393-405, 1992.

[Guyon *et al.* 88] I. Guyon, L. Personnaz, G. Dreyfus. Of points and loops. In *Neural Computers,* Springer Verlag, 1988.

[Haykin 94] Simon Haykin. *Neural networks - A Comprehensive Foundation*. New-York: Macmillan College Publishing Company, 1994.

[Hecht-Nielsen 91] Robert Hecht-Nielsen. *Neurocomputing*. Addison-Wesley, 1991.

[Holding 70] D.H. Holding. Learning Without Errors. In Leon E. Smith (Ed.) *Psychology of Motor Learning*, p. 59-81, Chicago: The Athletic Institute, 1970.

[Hopfield 82] J. J. Hopfield. Neural Networks and physical systems with emergent collective computationnal abilitie. *Proc. Natl. Acad. Science USA*, **79**, p. 2554-2558, 1982.

[Jodouin 93] Jean-François Jodouin. Réseaux de neurones et Traitement du Langage Naturel - Etude des réseaux récurrents et de leurs représentations. Thèse de doctorat, Univ. Paris XI Orsay, mars 1993.

[Jordan 88] Michael I. Jordan. Supervised learning and systems with excess degrees of freedom. COINS Technical Report 88-27, MIT, 1988.

[Kamimura 92] Ryotaro Kamimura. Activated Hidden Connections to Accelerate the Learning in Recurrent Networks. *Proc. IJCNN,* vol. I, p. 693-700, Baltimore,1992.

[Kremer 95] Stephan C. Kremer. On the Computationnal Power of Elman-Style Recurrent Networks. *IEEE Trans. on Neural Networks*., **6**(4), juillet 1995.

[Labbi 93] Abderrahim Labbi. Sur l'approximation et les systèmes dynamiques dans les réseaux neuronaux. Thèse de doctorat, Université Joseph Fourier, Grenoble, 1993.

[Lambert & Hecht-Nielsen 91] Jean Michel Lambert & Robert Hecht-Nielsen. Application of feedforward and Recurrent neural Networks to Chemical Plant Modelling. *Proc. IJCNN*, vol. I, p. 373-378, Seattle, 1991.

[Le Cun 87] Y. Le Cun. Modèles connexionistes de l'apprentissage. Thèse de doctorat, Univ. Paris VI, 1987.

[Li 92] Leong Kwan Li. Approximation theory and recurrent networks. *Proc. IJCNN,* vol. II, p. 286-271, Baltimore, 1992.

[Lin *et al.* 95] Tsungnan Lin, Bill G. Horne, Peter Tino & C. Lee Giles. Learning long-term dependencies is not as difficult with NARX recurrent neural networks. Tech. report UMIACS-TR-95-78 and CS-TR-3500, Institute for Advanced Computer Studies, University of Maryland, College Park, MD, 1995.

[Logar *et al.* 93] Antonette M. Logar, Edward M. Corwin & William J.B. Oldham. A comparison of Recurrent Neural Networks Learning Algorithms. *Proc. IEEE Conf. on Neural Networks*, p. 1129-1134, San Francisco, 1993.

[Mozer 88] Michael C. Mozer. A focused back-propagation algorithm for temporal pattern recognition, Rapport Technique, Univ. of Toronto, Dep^{ts} Psychology and Computer Science, CRG-TR-88-3, 1988.

[Narendra & Parthasarathy 90] Kumpati S. Narendra & Kannan Parthasarathy. Identifiaction and Control of Dynamical Systems Using Neural Ntworks. *IEEE Trans. on Neural Networks*, **1**(1), p. 4-27, mars 1990.

[Nerrand *et al.* 93] O. Nerrand, P. Roussel-Ragot, L. Personnaz & G. Dreyfus. Neural Networks and Nonlinear Adaptive Filtering: Unifying Concepts and New Algorithms. *Neural Computation*, **5**, p. 165-199, 1993.

[Omlin & Giles 93] Christian W. Omlin & C. Lee Giles. Pruning Recurrent Neural Networks for Improved Generalization Performance. Rapport Technique, Computer Science Departement, Rensselear Polytechnic Institute, Troy, N.Y, n° 93-6, avril 1993.

[Pearlmutter 89] Barak A. Pearlmutter. Learning State Space Trajectories in Recurrent Neural Networks, *Neural Computation*, **1**, p. 263-269, 1989.

[Pearlmutter 90] Barak A. Pearlmutter. Dynamic Recurrent Neural Networks. Rapport Technique, CMU-CS-90-196, décembre 1990.

[Pearlmutter 95] Barak A. Pearlmutter. Gradient Calculations for Dynamic Recurrent Neural Networks: A Survey. *IEEE Trans. on Neural Networks*, **6**(5), p. 1212-1228, septembre 1995.

[Piché 94] Stephen W. Piché. Steepest Descent Algorithms for Neural Network Controllers and Filters. *IEEE Trans. on Neural Networks*, **5**(2), p. 198-212, 1994.

[Pineda 87] Fernando J. Pineda. Generalization of back propagation to recurrent neural networks, *Physical review letters*, **59**(19), p. 2229-2232, 1987.

[Raysz 91] Jean-Pierre Raysz. Algorithmes d'apprentissage pour réseaux connexionnistes récurrents, application à la modélisation de processus d'interprétation. Thèse de doctorat, Univ. de Caen, 1991.

[Robinson 89] Anthony John Robinson. Dynamic Error Propagation Networks. PhD Thesis, Cambridge Univ., Engineering Departement, 1989.

[Rohwer 90] The "moving target" training algorithm. In L. B. Almeida & C. J. Wellekens (Eds.): *Proc. EURASIP Workshop on Neural Networks*, p. 101-109, Sesimbra, Portugal, février 15-17 1990. Dans la série Lecture notes in computer sciences, vol. 412, Springer, 1990.

[Rumelhart & McClelland 86] David E. Rumelhart & James L. McClelland (Eds). *Parallel Distributed Processing*, volume 1, Cambridge, Mass.: MIT Press, 1986.

[Schmindhuber 90] Jürgen Schmindhuber. A local learning algorithm for dynamic feedforward and recurrent networks. Rapport Technique, TUM, Munich, Allemagne, FKI-124-90, 1990.

[Srinivasan, *et al.* 94] B. Srinivasan, U.R. Prasad & N.J. Rao. Back Propagation Through Adjoints for the Identification of Nonlinear Dynamic Systems Using Recurent Neural models. *IEEE Transaction on Neural Networks*, **5**(2), p. 213-228, mars 1994.

[Sun *et al.* 91] Guo-Zheng Sun, Hsing-Hen Che & Yee-Chun Lee. A Fast On-Line Learning Algorithm For Recurrent Neural Networks. *Proc. IJCNN*, vol. II, p. 13-18, Seattle,1991.

[Szilas 94] Nicolas Szilas. Utilisation de la "stratégie d'apprentissage" dans les systèmes connexionnistes. *Proc. Colloque Jeunes Chercheurs en Sciences Cognitives*, La Motte d'Aveillans, Isère, France, 23-25 mars 1994.

[Szilas 95] Nicolas Szilas. Apprentissage dans les réseaux mécaniques et étude de leurs interactions avec l'environnement, PhD Thesis, INPG, Grenoble, France, Dec. 1995.

[Toomarian & Bahren 91] N. Toomarian & J. Bahren. Fast Temporal Neural Learning Using Teacher Forcing. *Proc. IJCNN*, vol I, p. 817-822, Seattle, 1991.

[Toomarian & Bahren 92] Nikzad Benny Toomarian & Jacob Bahren. Learning a Trajectory Using Adjoint Functions and Teacher Forcing. *Neural Networks*, **5**, p. 473-484, 1992.

[Tsoi & Back 94] A.C. Tsoi & A. Back. Locally recurrent Globally Feedforward Networks. *IEEE Trans. on Neural Networks*, **5**(2), p. 229-239, 1994.

[Tsung & Cotrell 95] Fu-Sheng Tsung & Garrison W. Cottrell. Phase-Space-Learning. In *NIPS-7*, p. 481-488, Cambridge, MA: MIT Press, 1995.

[Tsung 90] Fu-Sheng Tsung. Learning in recurrent Finite Difference Networks. In D. E. Touretzky, J. L. Elman, T. J. Sejnowski & G. E. Hinton (Eds.): *Proc. of the 1990 Connectionist Models Summer School*, p. 124-130, 1990 (San Mateo, CA: Morgan Kaufmann, 1991).

[Waibel *et al.* 89] Alexander Waibel, Toshiyuki Hanazawa, Geoffrey Hinton, Kiyohiro Shikano & Kevin J. Lang. Phoneme Recognition Using Time-Delay Neural Networks. *IEEE Trans. on Accoustics, speech, and signal processing*, **37**(3), p. 328-339, mars 1989.

[Watrous *et al.* 90] Raymond L. Watrous, Bruce Ladendorf, Gary Kuhn (1990). Complete gradient optimization of a recurrent network applied to /b/, /d/, /g/ discrimination, *Journal of Acoutical Soc. Am.*, **87**(3), p. 1301-1309, mars 1990.

[Williams 90] Ronald J. Williams. Adaptive State Representation and Estimation Using Recurrent Connectionist Networks. In W.T. Miller, R.S. Sutton & P.J. Werbos (Eds.): *Neural Networks for Control*, Cambridge, MA: MIT press, 1990.

Williams 92] Ronald J. Williams. Training Recurrent Networks Using the Extended Kalman Filter. *Proc. IJCNN*, vol. IV, p. 241-246, Baltimore, 1992.

[Williams & Peng 90] Ronald J. Williams and Jing Peng. An efficient gradient-based algorithm for on-line training of recurrent network trajectories. *Neural Computation*, **2**, p. 490-501, 1990.

[Williams & Zipser 89a] Ronald J. Williams & David Zipser. A Learning Algorithm for Continually Running Fully Recurrent Neural Networks. *Neural Computation*, **1**, p. 270-280, 1989.

[Williams & Zipser 89b] Ronald J. Williams & David Zipser. Experimental Analysis of the Real-time Recurrent Learning Algorithm. *Connection Science*, **1**(1), p. 87-111, 1989.

[Young & Chan 93] Evan Fung-Yu Young & Lai-Wan Chan. Ring-Structured Recurrent Backpropagation Network. *Proc. of the World Congress On Neural Networks*, vol. IV, p. 328-331, Portland, Oregon, 11-15 juillet 1993.

[Zipser 89] David Zipser. A Subgrouping Strategy that Reduces Complexity and Speeds Up Learning in Reurrent Networks. Neural Computation, 1, p. 552-558, 1989.