Université de Montréal

Génération et Reconnaissance de Rythmes au moyen de Réseaux de Neurones à Réservoir

par
Tariq Daouda

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Mémoire présenté à la Faculté des arts et des sciences
en vue de l'obtention du grade de Maître ès sciences (M.Sc.)
en informatique

Août, 2010

Université de Montréal
Faculté des arts et des sciences

Ce mémoire intitulé:

Génération et Reconnaissance de Rythmes au moyen de Réseaux de Neurones à Réservoir

présenté par:

Tariq Daouda

a été évalué par un jury composé des personnes suivantes:

Pascal Vincent,	président-rapporteur
Douglas Eck,	directeur de recherche
François Major,	membre du jury

Mémoire accepté le: . . . . . .

RÉSUMÉ

Les réseaux de neurones à réservoir, dont le principe est de combiner un vaste réseau de neurones fixes avec un apprenant ne possédant aucune forme de mémoire, ont récemment connu un gain en popularité dans les communautés d'apprentissage machine, de traitement du signal et des neurosciences computationelles. Ces réseaux qui peuvent être classés en deux catégories :

1. les réseaux à états échoïques (ESN)[29] dont les activations des neurones sont des réels
2. les machines à états liquides (LSM)[43] dont les neurones possèdent des potentiels d'actions,

ont été appliqués à différentes tâches [11][64][49][45][38] dont la génération de séquences mélodiques [30].

Dans le cadre de la présente recherche, nous proposons deux nouveaux modèles à base de réseaux de neurones à réservoir. Le premier est un modèle pour la reconnaissance de rythmes utilisant deux niveaux d'apprentissage, et avec lequel nous avons été en mesure d'obtenir des résultats satisfaisants tant au niveau de la reconnaissance que de la résistance au bruit. Le second modèle sert à l'apprentissage et à la génération de séquences périodiques. Ce modèle diffère du modèle génératif classique utilisé avec les ESN à la fois au niveau de ses entrées, puisqu'il possède une Horloge, ainsi qu'au niveau de l'algorithme d'apprentissage, puisqu'il utilise un algorithme que nous avons spécialement développé pour cette tache et qui se nomme *"Orbite"*. La combinaison de ces deux éléments, nous a permis d'obtenir de bons résultats, pour la génération, le sur-apprentissage et l'extraction de données. Nous pensons également que ce modèle ouvre une fenêtre intéressante vers la réalisation d'un orchestre entièrement virtuel et nous proposons deux architectures possibles que pourrait avoir cet orchestre. Dans la dernière partie de ce travail nous présentons les outils que nous avons développés pour faciliter notre travail de recherche.

Mots clés: réseaux à états échoïques, machines à états liquides, computation à réservoir, modèle génératif, réseaux de neurones récurrents, réseaux de neurones à réservoir, musique, rythme, séquences périodiques, apprentissage machine, intelligence artificielle.

ABSTRACT

Reservoir computing, the combination of a recurrent neural network and one or more memoryless readout units, has seen recent growth in popularity in and machine learning, signal processing and computational neuro-sciences. Reservoir-based methods have been successfully applied to a wide range of time series problems [11][64][49][45][38] including music [30], and usually can be found in two flavours: Echo States Networks (ESN)[29], where the reservoir is composed of mean rates neurons, and Liquid Sates Machines (LSM),[43] where the reservoir is composed of spiking neurons. In this work, we propose two new models based upon the ESN architecture. The first one is a model for rhythm recognition that uses two levels of learning and with which we have been able to get satisfying results on both recognition and noise resistance. The second one is a model for learning and generating periodic sequences, with this model we introduced a new architecture for generative models based upon ESNs where the reservoir receives inputs from a clock, as well as a new learning algorithm that we called *"Orbite"*. By combining these two elements within our model, we were able to get good results on generation, over-fitting and data extraction. We also believe that a combination of several instances of our model can serve as a basis for the elaboration of an entirely virtual orchestra, and we propose two architectures that this orchestra may have. In the last part of this work, we briefly present the tools that we have developed during our research.

Keywords: echo state networks, liquide state machines, reservoir computing, generative model, recurrent neural networks, music, rhythm, periodic time series, machine learning, artificial intelligence.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES FICHIERS

Fichiers D'entrainement :

- 2ROCK-16.mid
- 6BOSA-16.mid
- 2CHACHA-16.mid
- 6SAMBA-16.mid
- BAIAO-16.mid
- DIDDLEY-16.mid

Résultats :

- 2Rock :
  - 2Rock.mid
  - 2Rock.txt
  - 2Rock-Extraction.mid
  - 2Rock-Extraction.txt
- Bosa :
  - Bosa1.mid
  - Bosa1.txt
  - Bosa2.mid
  - Bosa2.txt
  - BosaOverfit1.mid
  - BosaOverfit1.txt
  - BosaOverfit1.png
  - BosaOverfit2.txt
  - BosaOverfit2.text
  - BosaOverfit2.png
- Chacha :
  - Chacha1.mid
  - Chacha1.txt
- Samba :

  - Samba1.mid
  - Samba1.txt
  - Samba-claire.mid
  - Samba-claire.txt
- Baiao :
  - Baiao1.mid
  - Baiao1.txt
  - Baiao2.mid
  - Baiao2.txt
- Diddley :
  - Diddley1.mid
  - Diddley1.txt
  - Diddley2.mid
  - Diddley2.txt
- Transitions :
  - Transitions.mid
  - Transitions.txt

LISTE DES SIGLES

BPTT	Rétro-propagation à travers le temps (Back-Propagation Through Time)
ESN	Réseau à états echoïques (Echo State Network)
ESP	Propriété des états échoïques (Echo State Property)
IP	Plasticité Intrinséque (Intrinsic Plasticity)
Li-ESN	Réseau à états echoïques à fuite (Leaky Intégrator Echo State Network)
LSM	Machine à états liquides (Liquid State Machine)
LSTM	Réseaux à longue mémoire à court terme (Long Short Term Memory)
MLP	Réseau de neurones à plusieurs couches (Multi-Layer Perceptron)
RC	Computations à réservoir (Reservoir Computing)
RTRL	Apprentissage récurrent en temps réel (Real Time Recurrent Learning)
RNN	Réseaux de neurones récurrents (Recurrent Neural Network)
RLS	Moindres carrés récursifs (Recursive Least Squares)
STDP	Plasticité en fonction de l'occurence du potentiel d'action (Spike Time Dependant Plasticity)

NOTATION

T	Cible.
U	Entrées.
N	Cardinalité du réservoir.
W_{in}	Poids reliant les entrées au réservoir.
W_{res}	Poids récurrents du réservoir.
W_{out}	Poids reliant le réservoir aux sorties.
W_{feed}	Poids reliant le retour d'informations au réservoir.
X	État du système.
Y	Sortie intermediaire.
Z	Sortie finale.

I really think that it's important to be in a position, both in art and in life where you don't understand what's going on - *John Cage*. [68]

REMERCIEMENTS

A l'entame de ce modeste travail je voudrais tout d'abord remercier mes parents et ma famille pour leur soutien inconditionnel et leur compréhension. J'aimerais également remercier tous les membres du GAMME, du LISA et du BRAMS pour tous les moments passés ensemble et en particulier :

- Douglas Eck pour m'avoir dirigé et pour m'avoir laissé la liberté de m'expérimenter.
- Yoshua Bengio et Pascal Vincent pour leurs l'excellence des cours d'apprentissage machines et leurs explications fournies généreusement.
- Sean Wood pour ses conseils précieux.
- Razvan Pascanu dont l'expertise m'a beaucoup aidé et qui est arrivé juste à temps à Montréal pour faire avancer ma recherche.

Au moment de terminer ce mémoire, j'ai également une pensée pour Nicolas Rougier et les membres de l'équipe Cortex du LORIA, auprès desquels j'ai eu mes premières expériences en recherche, ainsi que pour Mounir avec qui j'ai programmé mon premier Perceptron, il y a maintenant 7 années de cela.

CHAPITRE 1

INTRODUCTION

L'apprentissage de données issues de séquences périodiques est souvant une tâche difficile à réaliser, puisqu'elle demande de pouvoir prendre des décisions en fonction d'un contexte en perpétuelle évolution. En d'autres termes, un bon modèle de traitement de séquences périodiques doit pouvoir conserver une représentation du contexte suffisamment précise à même de lui permette de reconnaître l'influence d'entrées distantes sur la valeur de la cible actuelle. La musique représente d'ailleurs un excellent ensemble de données pour tester les performances de tels modèles, puisque le choix de la note suivante dépend d'informations contextuelles à plusieurs niveaux : telle que la position courante dans le morceau ou dans la mesure ou encore de la suite de notes récemment jouées. A priori, les réseaux de neurones récurrents seraient des candidats idéaux pour la réalisation de ce genre de tâches notamment par ce qu'ils sont capables de maintenir une représentation non linéaire de l'historique des entrées au sein de leurs dynamiques internes. Malheureusement, ces réseaux ont historiquement souffert du problème de dissipation du gradient qui apparaît dès que l'on essaye d'apprendre dans la partie récurrente du réseau, et qui peut les empêcher de reconnaître l'influence d'entrées lointaines sur la cible actuelle [24][8].

Récemment, un nouveau paradigme a été proposé en apprentissage machine et traitement du signal sous le nom de réseaux de neurones à réservoirs (Reservoir Computing, RC) [29][43]. Le principe derrière le RC est de séparer mémoire et traitement de l'information en déléguant la première à un large réseau de neurones récurrents fixes appelé réservoir, et la seconde à un apprenant quelconque dont au moins une partie des entrées est constituée de l'activation du réservoir. Ces réseaux ont par conséquent une capacité intrinsèque à conserver dans les dynamiques du réservoir une représentation non linéaire de l'historique sans avoir à souffrir du problème de dissipation du gradient puisque, aucun apprentissage n'est réalisé dans la partie récurrente du modèle, le réservoir. Ce sont ces caractéristiques, qui ont éveillé notre intérêt pour ces modèles et nous ont convaincus

de les utiliser pour nos recherches.

Le mémoire que nous présentons commence par une présentation générale des réseaux de neurones, et des réseaux de neurones récurrents et à réservoir en particulier. Ensuite nos travaux de recherches seront présentés d'abord en commençant par notre modèle de reconnaissance de rythme avant d'enchaîner sur notre modèle génératif, qui s'inscrit dans la lignée de travaux précédemment réalisés dans le laboratoire sur la génération musicale [30]. Dans ce chapitre, nous proposons un nouveau modèle génératif à base d'un réseau de neurones à réservoir, utilisant un nouvel algorithme d'apprentissage que nous avons appelé *"Orbite"* et que nous avons développé spécialement pour l'apprentissage de séquences périodiques. Ce modèle est capable d'apprendre à partir d'exemples à génerer automatiquement des séquences rythmiques complexes de plusieurs instruments.

L'avant dernier chapitre est consacré aux outils que nous avons développé pour faciliter nos recherches. Une conclusion récapitulant nos principaux résultats est proposée à la fin du présent mémoire.

CHAPITRE 2

LES RÉSEAUX DE NEURONES

Le terme réseaux de neurones regroupe une vaste catégorie de modèles d'apprentissage ayant tous en commun le fait de s'inspirer du fonctionnement du système nerveux : tous ces modèles mettent en jeu des unités appelées neurones dont les interactions sont gouvernées par des poids les reliant [19][44], les synapses. L'apprentissage d'un réseau de neurones consistant donc à faire varier ces poids jusqu'à obtenir les valeurs désirées en sortie.

De par leur robustesse, leur flexibilité et leur grande capacité de généralisation, les réseaux de neurones ont su s'imposer comme l'un des modèles majeurs en apprentissage machine [44] [22], ils sont aussi au centre de ce mémoire. Afin de donner au lecteur une bonne appréciation de ce que sont les réseaux de neurones, nous avons décidé de parler de divers modèles. Nous verrons tout d'abord les réseaux à propagation avant à travers le perceptron multicouche (MLP). Par la suite, nous parlerons des réseaux de neurones récurrents, nous introduirons divers modèles que nous avons jugé pertinents avant de terminer par le cas particulier des réseaux de neurones à réservoir qui occupent une place centrale dans ce mémoire.

2.0.1 Les réseaux à propagation avant

Un réseau à propagation avant est un réseau où l'information ne circule que dans un sens : de l'entrée vers la sortie. A aucun moment dans le réseau, il n y a de retour d'informations. Dans cette partie, nous verrons le modèle de réseaux de neurones le plus commun à savoir le perceptron, mais nous attaquerons directement la variante multicouche de l'algorithme.

Tout d'abord, introduisons la notion de couche. Un réseau de neurones peut avoir plusieurs couches, chaque couche étant un ensemble de neurones correspondant à un niveau de traitement. Ainsi dans un réseau de neurones à n couches, l'information subit n niveaux de traitements entre l'entrée est la sortie. Le nombre minimum de couche

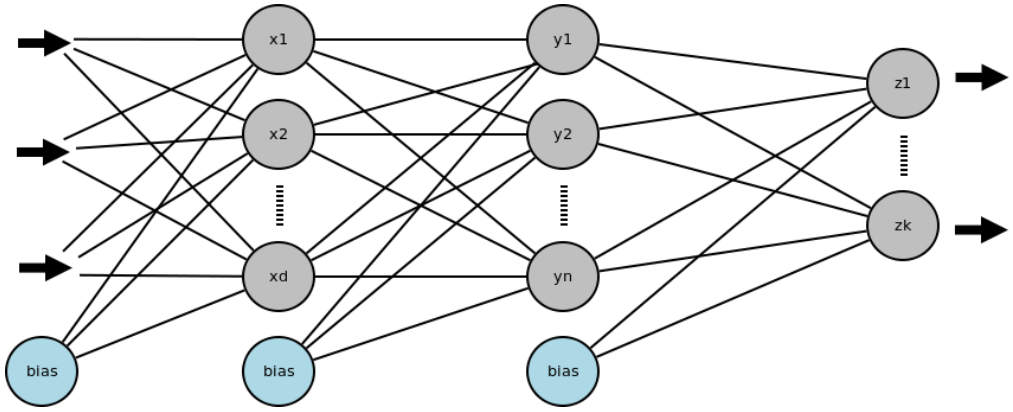

Figure 2.1 – Schéma d'un réseau de neurones multicouche classique, les fléches en gras indiquent le sens dans lequel se propage l'information dans le réseau.

est 2, une couche d'entrée et une couche de sortie, ce sont donc les couches situées entre ces deux niveaux qui font la différence. Ces couches portent le nom de *couches cachées*, et les neurones qui les composent celui de *neurones cachés* ou *unités cachées*. Un perceptron multicouche est un perceptron qui possède au moins une couche cachée.

La deuxième notion de première importance que nous introduisons est la notion d'activation, qui peut se traduire comme étant la somme pondérée des informations que reçoit un neurone des neurones situés dans la couche précédant la sienne. L'activation est donc une combinaison linéaire qui s'écrit comme ceci :

$$a_j^c = \sum_{j=1}^{n} w_{ji} y_i^{(c-1)} + b_j^k \tag{2.1}$$

où a_j^c est l'activation du neurone j de la c^{ieme} couche, $y_i^{(c-1)}$ la valeur retournée par le i^{ieme} neurone de la couche $c-1$, $w_j i$ le poids de la connexion entre ces deux neurones, b_j^c le biais du j^{ieme} neurone de la couche c et n le nombre de neurones dans la couche $c-1$. Une première remarque que l'on peut faire à ce niveau est qu'il est possible de simplifier l'équation en supprimant le biais, il suffit pour cela d'ajouter un neurone à la couche $(c-1)$ dont la valeur retournée est toujours 1.

L' équation devient alors :

$$a_j^c = \sum_{j=1}^{n} w_{ji} y_i^{(c-1)} \tag{2.2}$$

Par la suite nous avons :

$$y_j^c = h(a_j^c) \tag{2.3}$$

où h est la plupart du temps une non linéarité de type $tanh$. La fonction h peut s'interpréter comme étant le traitement de l'information que fournit le neurone et est appelée *fonction d'activation*. La figure 2.1 montre un schéma d'une architecture classique de perceptron multicouche, on peut y voir que chaque neurone d'une couche est relié à tous les neurones de la couche suivante et que l'entrée est propagée vers la sortie en suivant un chemin qui ne revient jamais en arrière.

2.0.2 L'algorithme de retro-propagation du gradient

Nous allons maintenant voir l'algorithme de rétro-propagation qui est le principal algorithme utilisé pour entraîner des perceptrons. Il existe différentes variantes de cet algorithme et d'importants travaux ont été réalisées afin de rendre l'algorithme le plus efficace possible. Nous suggérons en particulier au lecteur intéressé de lire l'article de Y. LeCun [35] ainsi que le livre de C. Bishop [10].

Le principe de l'algorithme de rétro- propagation est de calculer l'erreur de la cible à la sortie du réseau et de propager cette erreur de la sortie vers l'entrée en modifiant les poids au fur et à mesure. La figure 2.2 montre le flux d'information dans le réseau, les flèches noires indiquent le sens dans lequel se propage le signal d'entrée dans le réseau, alors que les flèches rouges indiquent le sens dans lequel se propage la fonction d'erreur et donc dans lequel s'effectue l'apprentissage. Dans cet exemple, nous prenons le cas général d'un réseau de neurones quelconque ayant une topologie et un nombre de couches quelconque, la seule condition étant que la fonction d'erreur et les fonctions d'activation de chaque neurone soient dérivables.

Soit E une fonction d'erreur telle que :

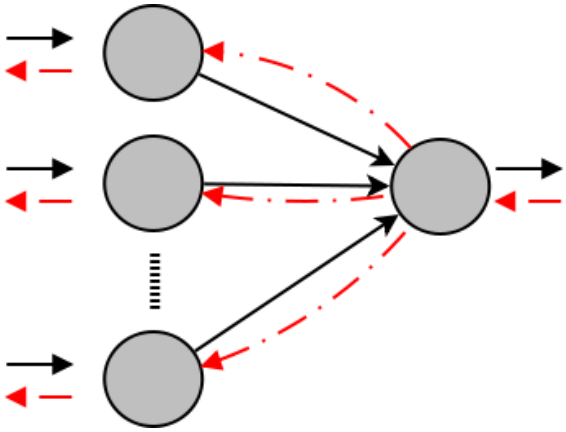

Figure 2.2 – Flux d'informations dans un réseau de neurones. Les flèches noirs indiquent le sens dans lequel se propage le signale d'entrée dans le réseaux, alors que les flèches rouges indiquent le sens dans lequel se propage l'erreur.

$$E = \frac{1}{2}\sum_{k}(z_k - t_k)^2 \tag{2.4}$$

ou Z et T sont réspéctivement les vecteurs de sortie de du modèle et le vecteur cible, z_k correspondant à la valeur du k^ieme neurone de la couche de sortie et t_k la valeur cible lui correspondant. Ce dont nous avons besoin afin de faire évoluer les poids dans la bonne direction est d'une mesure qui nous donne à la fois la *responsabilité* de chaque poids dans l'erreur et la direction dans laquelle doit s'effectuer la variation. En d'autres termes, nous devons dériver l'erreur par rapport aux poids. Intéressons nous d'abord à la couche de sortie, pour dériver l'erreur par rapport à un poids w_{kj} nous pouvons appliquer la règle de dérivation en chaîne et introduire l'activation du neurone k, a_k dont nous rappelons la formule :

$$a_k = \sum_{j} w_{jk} y_j \tag{2.5}$$

Il est maintenant possible de dériver l'erreure E par rapport à un poids $w_k j$:

$$\frac{\partial E}{\partial w_{kj}} = \frac{\partial E}{\partial a_k}\frac{\partial a_k}{\partial w_{kj}} \tag{2.6}$$

Nous introduisons maintenant une notation pratique en posant : $-\delta_k = \frac{\partial E}{\partial a_k}$, $-\delta_k$ étant appelé la sensibilité. On a donc d'après l'équation 2.5 :

$$\frac{\partial E}{\partial w_{kj}} = -\delta_k \frac{\partial a_k}{\partial w_{kj}} = -\delta_k y_j \tag{2.7}$$

Intéressons nous maintenant au calcul de la sensibilité, nous appliquons encore une fois la règle de dérivation en chaîne :

$$\delta_k = \frac{-\partial E}{\partial a_k} = -\frac{\partial E}{\partial z_k}\frac{\partial z_k}{\partial a_k} \tag{2.8}$$

et comme : $z_k = h(a_k)$, on à :

$$\delta_k = -(t_k - h(a_k))h'(a_k) \tag{2.9}$$

La formule pour faire varier le poids w_{kj} est donc :

$$w_{kj} = \mu(t_k - h(a_k))h'(a_k) \tag{2.10}$$

où μ est un scalaire appelé taux d'apprentissage.

Il est temps maintenant de passer à la couche cachée, nous appliquons encore une fois de plus la règle de dérivation en chaîne :

$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial a_j}\frac{\partial a_j}{\partial w_{ji}} = (\sum_k w_{kj}\delta_k)h'(a_j)x_i \tag{2.11}$$

La formule de variation est donc :

$$w_{ji} = \mu(\sum_k w_{kj}\delta_k)h'(a_j)x_i \tag{2.12}$$

où encore une fois μ est le taux d'apprentissage.

2.0.3 Les Réseaux de Neurones Récurrents (Recurrent Neural Networks)

Les réseaux de neurones récurrents (RNN) partagent avec les réseaux à propagation leurs caractéristiques fondamentales. Ils sont également composés de neurones inter-

agissant au travers de connexions pondérées et leur apprentissage consiste également à trouver la valeur de ces poids, mais ces réseaux ont la particularité de posséder des cycles dans leur topologie. Cette unique différence en fait des modèles au comportement complètement différent, puisqu'elle en fait des systèmes dynamiques contrairement aux réseaux à propagation avant qui sont des fonctions. Ils sont par conséquent beaucoup plus difficiles à maîtriser et leur apprentissage est plus difficile à mettre en place. Mais ces réseaux possèdent également de grands avantages sur leurs cousins à propagation avant, puisque la présence de cycle leur permet :

– De conserver une représentation interne de l'historique des entrées voir des traitements effectués sur ces entrées. En d'autres termes, les dynamiques observées dans les réseaux correspondent à la transformation non linéaire de l'historique des entrées. Cette qualité est de première importance lorsqu'il s'agit de prendre des décisions en fonction du contexte, comme en improvisation musicale par exemple, où le modèle doit en général prendre en compte plusieurs facteurs, comme la position courante dans la pièce jouée, l'accord courant, ou encore les n dernières notes qu'il a joué.
– De développer une activité autonome même en l'absence d'entrées.

Ces réseaux de neurones sont aussi beaucoup plus plausibles biologiquement ce qui en fait également des modèles très étudiés dans le domaine des neurosciences computationelles. Afin de donner une bonne compréhension au lecteur de ce que sont les réseaux de neurones récurrents nous débuterons avec une brève explication de ce que sont les systèmes dynamiques, nous verrons par la suite les modèles de réseaux de neurones réccurents les plus courants ainsi que leurs limitations avant de terminer par les réseaux de neurones à réservoir.

2.1 Définition générale des systèmes dynamiques

La définition la plus triviale et la plus générale que l'on peut donner d'un système dynamique est celle d'un système dont l'état évolue en fonction du temps. Il est donc impératif pour pouvoir parler de systèmes dynamiques de définir clairement la notion

d'état. L'état du système peut être représenté par un ensemble de variables $x_i(t)$ à $x_n(t)$ (que nous regroupons au sein d'un vecteur $X(t)$ de cardinalité N) dont les valeurs renferment suffisamment d'informations pour prédire le prochain état. En d'autres termes, il existe une fonction f continue et en générale non linéaire telle que [1] :

$$\frac{\partial X(t)}{\partial t} = f(X(t)) \tag{2.13}$$

Les valeurs contenues dans $X(t)$ définissent donc entièrement celle de $X(t+\varepsilon)$.

Pour être sûr que f soit continue et afin de garantir l'unicité de la solution il est nécessaire que f satisfasse la condition de Lipschitz à savoir qu'il existe une constante k tel que :

$$\|f(X) - f(U)\| \leq k\|X - U\| \tag{2.14}$$

Il existe deux types de systèmes dynamiques, les systèmes autonomes, dont l'état ne dépend pas explicitement du temps, et les systèmes non autonomes [22]. Nous ne nous intéresserons qu'aux systèmes de la première catégorie puisque notre modèle en fait partie.

2.1.1 L'espace d'états

L'une des façons les plus intuitives d'étudier un système dynamique est de le considérer comme évoluant dans un espace à N dimensions, défini par l'ensemble des valeurs que peut prendre le vecteur X. Nous nommerons cet espace l'espace des états du système, la figure 2.3 montre l'évolution que pourrait avoir un système dynamique dans un espace d'états à deux dimensions. On y voit également le rôle de f dans la définition de la trajectoire que suit le système. Il est important de noter que la trajectoire empruntée par le système dépend des conditions initiales $X(t=0)$. Il existe par conséquent toute une famille de trajectoires possibles, toutes issues de conditions initiales différentes. Par

1. Il s'agit ici d'un formalisme général, dans le cas particulier des réseaux de neurones, le temps est en général défini de façon discrette. L'équation s'écrit alors : $X(t+\Delta t) = f(X(t))$ ou Δt est le pas de temps, en général 1.

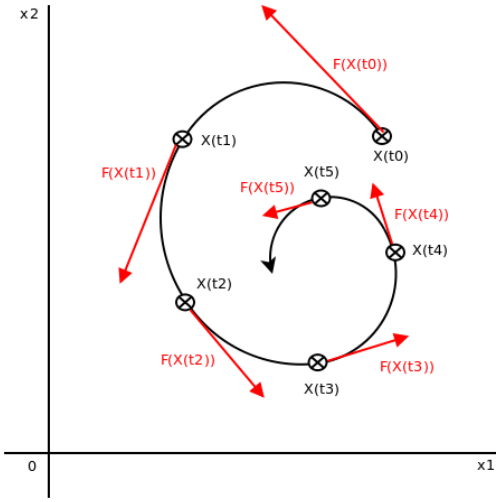

Figure 2.3 – Exemple d'évolution d'un système dynamique dans son espace d'états à deux dimensions.

conséquent, il est également possible de voir l'espace d'états comme un champs vectoriel des différents $f(X)$.

2.1.2 L'algorithme de retro-propagation à travers le temps (Back propagation through time)

Cet algorithme est comme son nom l'indique, un algorithme de rétro-propagation où l'erreur se propage à la fois dans le réseau et dans le temps. Tout se passe comme si le réseau gagnait une couche cachée de plus à chaque pas de temps. Le concept peut être déroutant au premier abord mais il s'explique simplement en déroulant l'activité du réseau dans le temps. Considérons le réseau de la figure 2.4, le premier neurone reçoit l'entrée et l'activité du second constitue la sortie, si l'on déroule l'activité du réseau de

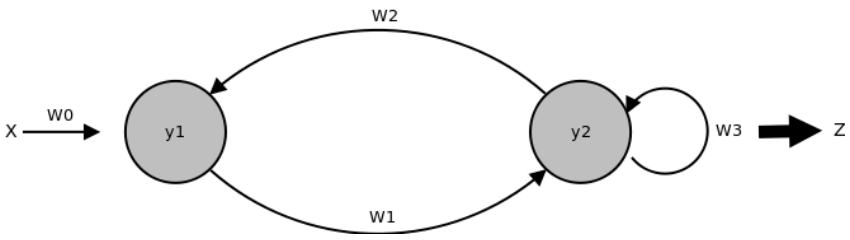

Figure 2.4 – Un exemple simple de réseau de neurones récurrent.

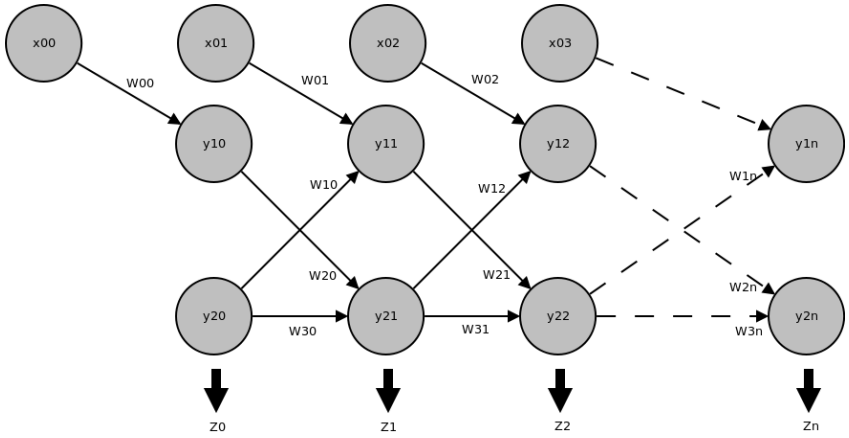

Figure 2.5 – L'activité du réseau de la figure 2.4 déroulée sur n pas temps.

$t = 0$ à $t = n$ on obtient le réseau de neurones multicouche de la figure 2.5. L'apprentissage du réseau récurrent à $t = n$ est donc équivalent à une simple rétro-propagation de l'erreur dans le réseau de la figure 2.5.

2.1.3 Apprentissage récurrent en temps réel (Real Time Recurrent Learning)

L'apprentissage récurrent en temps réel est un autre algorithme pour l'entraînement de réseaux de neurones récurrents utilisant une descente de gradient, mais contrairement à la rétro-propagation à travers le temps il est principalement conçu pour l'apprentissage en ligne [22]. La figure 2.6 montre un exemple d'architecture classique de réseaux utilisant l'apprentissage récurent en temps réel. Pour un réseaux de q neurones recevant m entrés chacun des trois neurones reçoit :

- Les entrées du modèle
- Un Biais
- L'état du système X qui correspond aux valeurs retournées par les trois neurones, et dont la formule est :

$$X_{n+1}^T = [\phi(w_1^T M_n)...\phi(w_j^T M_n)...\phi(w_q^T M_n)] \tag{2.15}$$

où ϕ est une fonction d'activation arbitraire dérivable, w_j le vecteur de poids cor-

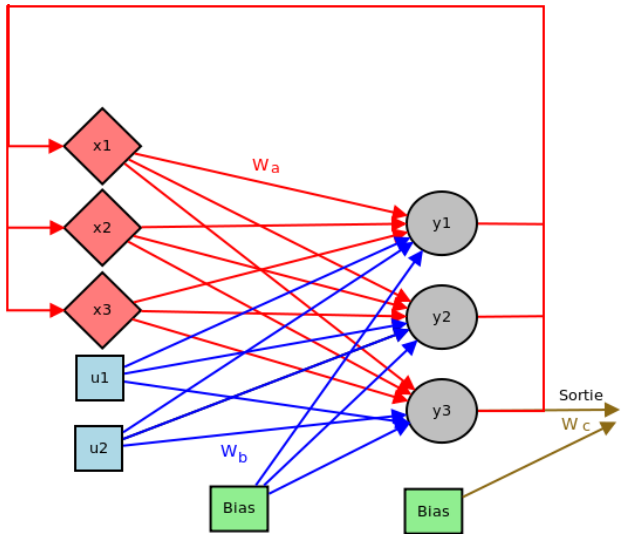

Figure 2.6 – Un réseaux utilisant l'apprentissage récurrent en temps réel.

respondant au neurone j[2], et M_n l'état du système X au temps n augmenté de l'entrée U_n tel que :

$$M_n = \begin{pmatrix} X_n \\ U_n \end{pmatrix} \tag{2.16}$$

La sortie du modèle ne dépend directement que de la valeur du neurone q. Typiquement ce type de réseaux possède un jeu de 3 matrices de poids W_a, W_b et W_c, reliant respectivement les neurones à l'état du système, à l'entrée du modèle et à la sortie.

Pour simplifier les calculs nous omettons d'indiquer explicitement le biais dans les équations, nous introduisons également les notations suivantes pour le j^{ieme} neurone :

$$w_j = \begin{pmatrix} w_{a,j} \\ w_{b,j} \end{pmatrix} \tag{2.17}$$

$$\Delta_{j,n} = \frac{\partial X_n}{\partial w_j} \tag{2.18}$$

2. Ce vecteur contiens $m+q+1$ pour m entrées, q neurones, et un bias.

$$U_{j,n} = \begin{pmatrix} 0 \\ \vdots \\ M_n^T \\ \vdots \\ 0 \end{pmatrix} \tag{2.19}$$

M_n^T est situé au niveau du j^{ieme} élement.

On peut maintenant dériver l'équation 2.15 par rapport au poids w_j, ce qui nous donne l'équation récursive suivante pour un neurone j quelconque :

$$\Delta_{j,n+1} = \Phi_n(W_{a,n})\Delta_{j,n} + U_{j,n}) \tag{2.20}$$

où Φ_n est une matrice diagonale telle que : .

$$\Phi_n = \begin{pmatrix} \phi'(w_1^T M_n) & 0 & \cdots & 0 \\ 0 & \phi'(w_2^T M_n) & \cdots & \vdots \\ \vdots & \cdots & \ddots & \vdots \\ \vdots & \cdots & \cdots & \phi'(w_q^T M_n) \end{pmatrix} \tag{2.21}$$

Nous introduisons maintenant la fonction d'erreur du modèle qui n'est autre que l'erreur quadratique :

$$E_n = \frac{1}{2} e_n^T e_n \tag{2.22}$$

où $e_n = d_n - y_n$

d_n étant la valeur de la cible et y_n la sortie du modèle au temps n.

$$\begin{aligned} \frac{\partial E_n}{\partial w_j} &= (\frac{\partial e_n}{\partial w_j}) e_n \\ &= -W_c(\frac{\partial X_n}{\partial w_j}) e_n \\ &= W_c \Delta_{j,n} e_n \end{aligned} \tag{2.23}$$

La règle de modification des poids w_j est donc :

$$w_{j,n} = -\mu W_c \Delta_{j,n} e_n \tag{2.24}$$

où μ est le taux d'apprentissage. Afin que tout soit cohérent il ne reste plus qu'à initialiser Δ_j à qui l'on donnera la valeur 0 pour tout j, à $t = 0$.

2.1.4 Limitations de ces approches

Il est souvent nécessaire pour la plupart des applications pratiques que le réseau puisse retenir l'information pour une durée de temps non déterminée, ce phénomène est appelé *verrouillage* (latching en anglais). Malheureusement, les deux algorithmes BPTT et RTRL souffrent du même problème de *dissipation du gradient* (vanishing gradiant). Ce problème qui à été démontré par Bengio et al. [8][24] implique qu'il peut être très difficile d'entraîner l'intégralité de réseaux récurrents (ou très profonds) à l'aide de méthodes à base de descente de gradient. En effet les conditions nécessaires au verrouillage sont également les conditions suffisantes pour obtenir une dissipation de gradient. Dans ces articles, les auteurs montrent que l'espace d'états des unités cachées peut être divisé en deux types de régions : celles où le gradient décroît exponentiellement et celles où il augmente exponentiellement. Cette étude s'intéresse particulièrement aux attracteurs hyperboliques qui sont des points stables dans l'espace d'état.

Soit $X(t)$ l'état du système autonome (par exemple le vecteur d'activation des neurones cachés) au temps t, et soit M une fonction continue telle que :

$$X(t+1) = M(X(t)) \tag{2.25}$$

L'évolution du système peut alors être décomposée en fonction de la norme de la Jacobienne M', $\|M'\|$:

- Si $\|M'\| > 1$, il n'y a aucun verrouillage possible puisque le gradient explose exponentiellement, il devient par conséquent très sensible au bruit et une légère perturbation peut propulser le système hors du bassin d'attraction de son attracteur actuel.

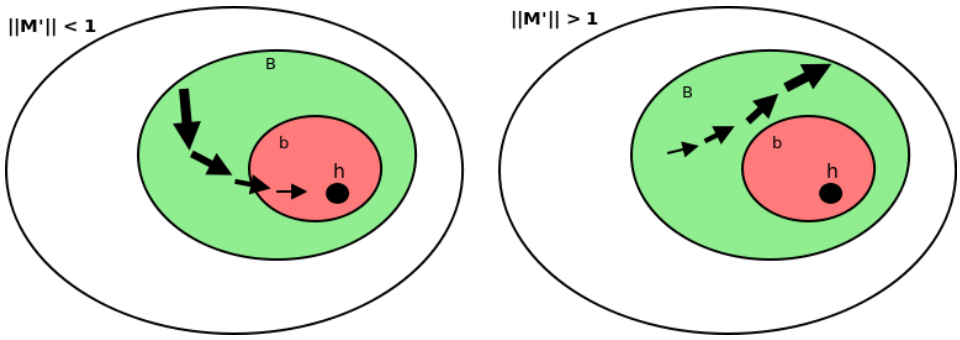

Figure 2.7 – Dissipation du gradient. Si $\|M'\| < 1$, le verrouillage est possible mais le gradient contient de moins en moins d'information. Si $\|M'\| > 1$, le modèle est très sensible aux perturbations ce qui peut rendre le verrouillage impossible. h est l'attracteur courrant, B son bassin d'attraction étendu et b son bassin d'attraction réduit.

– Si $\|M'\| < 1$, alors le verrouillage est possible mais, le gradient décroît également exponentiellement, ce qui signifie que le modèle devient incapable de reconnaître l'influence d'entrées distantes sur la cible actuelle.

Ce dilemme explique pourquoi il peut être très difficile d'entraîner des réseaux de neurones récurrents à l'aide de descente de gradient, la figure 2.7 montre une schématisation de ce phénomène. En pratique, il arrive souvent que ce genre de réseaux soit incapable de prendre en compte des entrées arrivées une dizaine de temps plus tôt, ce qui les rend moins fiables qu'un réseau de neurones classique avec une fenêtre de temps en entrée. Ces réseaux ont aussi le désavantage de "croître" avec le temps ce qui implique qu'ils demandent de plus en plus de ressources au fur et à mesure que l'entraînement avance.

2.1.5 Les réseaux à longue mémoire à court terme (Long Short Term Memory)

Un type de réseaux de neurones récurrents très utilisé sont les Les réseaux à longue mémoire à court terme (LSTM). Ces réseaux introduits en 1997 contournent le problème de dissipation du gradient en l'emprisonnant dans des carrousels possédant à la fois une porte d'entrée et une porte sortie [25]. Ces réseaux ont été appliqués à des tâches similaires à celles que l'on désire entreprendre dans ce mémoire [20][18], mais malheureusement, il arrive qu'ils aient une certaine sensibilité au bruit, ce qui nous à

conduit à explorer d'autres méthodes comme les réseaux de neurones à réservoir. Pour une description plus précise de ce que sont les LSTM nous référons le lecteur aux travaux de Gers et al [21], Schmidhuber et al. [51] ou encore de Perez et al. [47].

2.2 Réseaux de neurones à réservoir (Reservoir Computing)

Nous allons maintenant nous intéresser au modèle de réseaux de neurones récurrents que nous avons utilisé dans ce travail, à savoir les réseaux de neurones à réservoir (RC).

Cette architecture particulière a pour principal avantage de ne propager aucune erreur dans la partie récurrente du réseau, ce qui permet à la fois de contourner le problème de la dissipation du gradient et d'éviter que le réseau n'augmente en taille effective au fur et à mesure que l'apprentissage se poursuit. Elle est également beaucoup plus plausible biologiquement puisqu'elle fournit une explication des mécanismes permettant au cerveau d'effectuer des calculs justes avec des entrées bruitées. Les réseaux de neurones biologiques étant extrêmement sujets au bruit [16]. Les réseaux de neurones à réservoir présentent donc un intérêt aussi bien pour la recherche en apprentissage machine et traitement du signal que pour celle en neurosciences computationelles, ce qui donne souvent lieu à des échanges intéressants entre ces différentes communautés.

Depuis leur introduction, les réseaux de neurones à réservoirs ont été appliqués à différentes tâches avec succès [11][64][49][45], permettant même d'obtenir des résultats supérieurs à ceux de l'état de l'art sur certaines d'entres elles [31][32]. Il a également été démontré que ces réseaux sont capables de computation universelle si munie d'un retour d'informations [40]. Parmi ces nombreuses applications et principales motivations pour ce mémoire nous retiendrons en particulier les travaux de Jaerger et Eck où un modèle de ce genre a été utilisé pour la génération de séquences mélodiques [30]. Les réseaux de neurones récurrents sont en théorie particulièrement adaptés à la génération musicale et au traitement de séquences temporelles en général, puisqu'ils possèdent intrinsèquement la capacité de conserver une transformation non linéaire de la séquence d'entrées. Les réseaux de neurones à réservoir possèdent cette qualité sans avoir à souffrir du problème de dissipation du gradient et sans présenter les problèmes de stabilité que l'on rencon-

tre parfois avec les LSTM. Ils sont aussi beaucoup plus faciles à implémenter que ces dernier. Toutes ces caractéristiques en font des modèles prometteurs pour le traitement d'informations musicales et nous ont convaincu de les utiliser pour nos recherches.

2.2.1 Les différents paradigmes du RC

Il existe différents types de réseaux de neurones à réservoir [3], certains pré datant les contributions faites indépendamment par Jaeger [28][27] et Maas [43], comme par exemple le modèle de Décorrelation de la Rétro-propagation (Backpropagation-Decorrelation) [58], ou encore certaines contributions en neurosciences computationelles [17]. Mais c'est véritablement à la suite des travaux entrepris par les laboratoires des Docteurs Jaeger et Maas que l'approche dorénavant connue sous le nom *Computation à réservoir* (Reservoir Computing) à pris sa forme et son formalisme actuel.

Ces contributions sont également à l'origine des deux types de réservoirs les plus utilisés aujourd'hui à savoir les *réseaux à états échoïques*[28] (Echo State Networks, ESN) et les *machines à états liquides*[43] (Liquid State Machines, LSM).

Dans cette section, nous allons tout d'abord nous intéresser au formalisme général du RC avant de nous attarder sur les fonctionnements particuliers des ESN et des LSM, nous finirons par un court chapitre présentant quelques méthodes d'optimisation du réservoir. Bien qu'aucune de ces méthodes n'ait été utilisée dans ce travail, nous pensons qu'elles présentent des ouvertures prometteuses quant à l'évolution du domaine.

2.2.2 Formalisme Général

Le principe général du RC est de séparer mémoire et traitement de l'information. Le réseau est composé de deux parties distinctes et indépendantes l'une de l'autre, le réservoir et l'apprenant. La tâche du premier étant de traiter l'information avant de l'envoyer au second qui s'en sert comme entrée, seuls les poids en direction de l'apprenant participent activement à l'apprentissage ce qui constitue la principale différence entre les méthodes à réservoir et les réseaux de neurones récurrents classiques. Les figures 2.8

3. certains auteurs ont même fourni l'efforts de proposer des modèles unificateurs du RC [63]

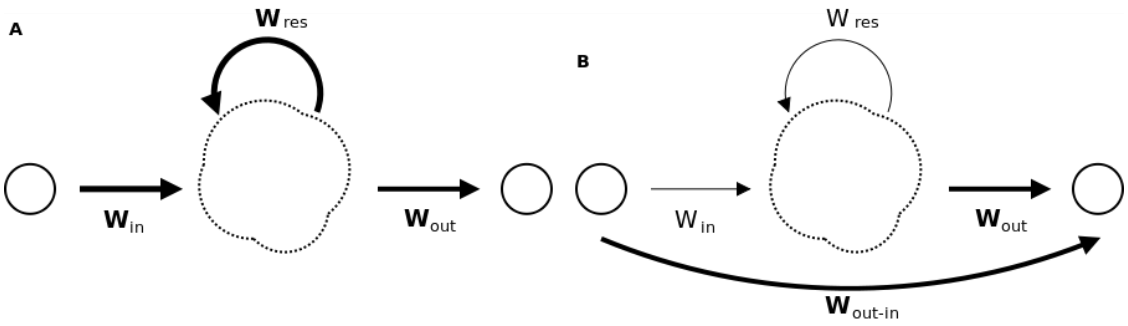

Figure 2.8 – Différences entre un réseau de neurones classique (A) et un réseau de neurones à réservoir (B). Seuls les poids en gras sont modifiés lors de l'apprentissage.

montrent les différences d'architectures entre ces deux types de modèles.

Le modèle de RC classique proposé indépendamment par Jaeger [28] et Maas [43] comporte donc :

- Un réservoir, qui n'est autre qu'un grand réseau de neurones récurrents généré aléatoirement et fixé durant l'intégralité de l'apprentissage au travers duquel transite les entrées. Le rôle de cette unité passive étant de conserver une transformation non linéaire de l'historique des entrées.
- Un apprenant, en général une régression linéaire, dont le rôle est d'apprendre la correspondance entre l'état du réservoir et la cible.

Le traitement temporel de l'information est donc complètement séparé de l'apprentissage, on peut par ailleurs facilement dresser un parallèle entre les méthodes à réservoir et les méthodes à noyau [10], puisque ces deux méthodes utilisent le même principe fondateur qui est de projeter l'entrée dans un espace à haute dimension (à l'aide d'une transformation en générale non-linéaire) avant de la fournir à l'apprenant.

On à donc :

$$Y(n) = W_{out}(n)X(n) \tag{2.26}$$

Où $X(n)$ désigne l'état du réservoir au temps n. Le rôle du réservoir est de fournir un vecteur d'état X le plus riche et cohérent possible pour tout n. $X(n)$ étant définit de la façon récurrente suivante :

$$X(n+1) = f(W_{in}U(n) + W_{res}X(n) + W_{feed}Y(n)) \tag{2.27}$$

où f est la fonction d'activation des neurones du réservoir, W_{in} la matrice de poids entre l'entrée et le réservoir, W_{res} est la matrice des poids récurrents du réservoir et W_{feed}, la matrice des poids entre le retour d'informations et le réservoir. Ce retour d'informations est facultatif et le choix de l'inclure ou non dans l'équation dépend de la tâche d'apprentissage à effectuer.

2.2.3 Les machines à états liquides (Liquide State Machines)

Les machines à états liquides sont des modèles issus des neurosciences computationelles [43][39] et sont en conséquence biologiquement réalistes. Le réservoir de ces réseaux modélise souvent le fonctionnement d'une colonne micro-corticale [41]. Par conséquent, les neurones utilisés sont en général des neurones à train d'action, le réseau est tri-dimentionel et les valeurs des hyper-paramètres le régissant sont tirées d'observations empiriques. Ces réseaux sont donc par définition complexes et difficiles à implémenter et à configurer.

2.2.4 Les réseaux à états échoïques (Echo State Networks)

Les réseaux à état échoïques (ESN)[28] sont quant à eux issus d'une approche plus pratique, liée à l'apprentissage machine et au traitement de signal [38]. Le réservoir est par conséquent un réseau de neurones récurrent classique dont la matrice de poids W_{res} est générée aléatoirement selon une distribution uniforme, les neurones qui le composent sont en général parcimonieusement connectés et ont une sigmoïde (en général *tanh*) comme non-linéarité. Le réservoir est fixé au début de l'apprentissage et aucun apprentissage n'y est effectué. La taille du réservoir est choisie en fonction de la tâche et peut aller de quelques dizaines de neurones à plusieurs milliers. Les matrices W_{in} et W_{feed} si nécessaires sont également fixées et générées aléatoirement selon une distribution uniforme, ces matrices sont en général denses mais peuvent également être creuses. Il arrive également que l'on ait besoin de redimensionner les entrées ou le retour d'in-

formations, de les translater ou de définir un biais au réservoir, qui n'est autre qu'une entrée dont la valeur est une constante réelle. Tout dépend du régime dans le lequel on désire que le réservoir fonctionne, plus les valeurs en sont élevées et plus le régime est non-linéaire. Ceci étant due à la fonction *tanh* qui est fortement linéaire proche de 0 et qui gagne en non-linéarité au fur et mesure que l'on s'éloigne de cette valeur.

La simplicité des ESN en fait des modèles particulièrement aisés à implémenter et à utiliser puisqu'à la fois ils possèdent un jeu réduit d'hyper paramètres mais réclament également moins d'attention (comparativement aux LSM) dans le choix des valeurs de ces hyper paramètres [38][52].

La sortie $Y(n)$ d'un ESN classique est généralement calculée en fonction de l'état du réservoir $X(n)$ et de l'entrée fournie au modèle $U(n)$ [29] :

$$Y(n) = f_{out}(W_{out}[U(n)|X(n)]) \tag{2.28}$$

où f_{out} est l'activation du neurone de sortie, en général l'identité , W_{out} une matrice de poids réels, U l'entrée, et $|$ le symbole de la concaténation.

Par ailleurs, l'un des fondements des ESN est de posséder ce que les auteurs nomment la *"propriété d'états échoïques"* (Echo State Property, ESP) [38][28], qui donne d'ailleurs son nom aux ESN. Selon cette propriété, l'influence d'une entrée sur le réservoir doit graduellement s'estomper au fur et à mesure que le temps avance. Le nom vient de l'analogie entre le fonctionnement de ces réseaux et le phénomène sonore d'écho : une stimulation envoyée dans le réservoir se réverbère dans celui-ci et finit par s'estomper comme le ferait un signal sonore dans une pièce vide.

Pour posséder cette propriété il est suffisant que la matrice W_{res} soit contractive [38], en d'autres termes que $\rho(W_{res}) < 1$ où ρ désigne le rayon spectral (valeur de la valeur absolue la plus grande) de la matrice, cette condition n'est pas nécessaire et il est tout à fait possible d'avoir des réservoirs possédant l'ESP mais avec un $\rho(W_{res}) > 1$. $\rho(W_{res})$ est également une mesure approximative, mais en général suffisamment juste, de la mémoire du réservoir, plus sa valeur est proche de 1 et plus l'influence des entrées reçues par le réservoir dure longtemps. En pratique on se contentera souvent de redimensionner

les valeurs de la matrice W_{res} pour obtenir une valeur $\rho(W_{res})$ en accord avec la tâche que l'on désire accomplir [38].

L'apprentissage est en général effectué à l'aide d'une régression linaire, puisque cette méthode est à la fois peu coûteuse en temps de calculs et en ressource, mais n'importe quel autre algorithme d'apprentissage peut être utilisé par dessus le réservoir. Dans le cas où l'apprentissage nécessite un retour d'informations, il est supprimé durant l'apprentissage et est remplacé par la cible juste. Cette méthode qui porte le nom de *"teacher forcing"* en anglais, empêche que le retour d'informations ne fasse diverger le modèle pendant l'apprentissage en propageant des entrées erronées dans le réservoir.

2.2.5 ESN à Intégrateurs à fuite (Leaky Integrator ESN)

Une version légèrement plus complexe du réservoir est également courante, il sagit des ESN à Intégrateurs à fuite (Li-ESN), cette version consiste à remplacer les neurones par des intégrateurs à fuite. Il existe deux grandes familles de réservoirs à intégrateurs à fuite en posant par exemple :

$$X(n) = (1 - m\Delta t)X(n-1) + \Delta t f(W_{in}U(n) + W_{res}X(n-1)) \tag{2.29}$$

où Δt est la constante définissant le pas de temps, et m la constante qui gouverne le taux de fuite. La seule contrainte quant au choix de valeurs de ces paramètres est que $\Delta t m$ soit dans $[0,1]$. Une version plus compacte et plus simple de cette équation peut être obtenue en posant $m = 1$ et en donnant à Δt, que l'on renomme a et que l'on borne entre $[0,1]$, le rôle précédemment échue à cette constante :

$$X(n) = (1 - a)X(n-1) + af(W_{in}U(n) + W_{res}X(n-1)) \tag{2.30}$$

Cette version est non seulement plus simple à utiliser, mais a aussi l'avantage de maintenir l'activation du réservoir entre les bornes de f. Il est également intéressant de remarquer que si l'on pose $a = 1$ on obtient un réservoir classique, ce qui implique qu'un Li-ESN dont le réservoir possèdent ce type de non-linéarité est au moins aussi bon qu'un ESN classique. C'est cette version de la formule que nous avons utilisée dans nos réser-

voirs en y ajoutant le retour d'informations lorsque la tâche le nécessite. D'un point de vue pratique, utiliser des intégrateurs à fuite comme neurones a de nombreux avantages, puisqu'en changeant la valeur de a il est possible d'influer sur la mémoire du réservoir ou plutôt la vitesse à laquelle le réservoir réagit en présence d'entrées (plus la valeur de a est faible et plus le réservoir réagit lentement), la valeur de a a par conséquent une influence directe sur l'ESP. L'équation 2.30 est également l'équation d'un filtre passe-bas dont la fréquence de référence est :

$$fc = \frac{a}{2\pi(1-a)\delta t} \tag{2.31}$$

où δt est le pas de discrétisation temporel. Changer a permet donc de modifier l'intervalle de fréquences auquel sont sensibles les neurones du réservoir. Certaines approches proposent par ailleurs d'utiliser des réservoirs avec des neurones sensibles à différents intervalles de fréquences afin d'augmenter sa capacité à rendre compte d'entrées se produisant à différentes échelles de temps [55].

2.2.6 Mesurer la qualité du réservoir

Dans cette section, nous présentons certaines méthodes de mesure de la qualité du réservoir, bien que la seule méthode dont nous ayons eu besoin dans la version de ce travail est la mesure du rayon spectral [4] [38], nous avons utilisé certaines de ces methodes pendant notre recherche et nous pensons qu'elles valent néanmoins la peine d'être documentées.

Les deux premières methodes de mesures que nous présentons ont été introduites dans [36], ces méthodes sont la *qualité du noyau* (kernel-quality en anglais) et le *rang de généralisation* (generalisation-rank). Ces deux mesures sont fondées sur la récolte des activités du réservoir au sein d'une matrice et du calcul du rang [5], de cette matrice.

4. Valeur de la valeur absolue la plus grande de la matrice de poids du résersoire W_res

5. L'utilisation de ces mesures avec des ESN pose problème puisque calculer le rang d'une matrice composée de flottants n'a pas beaucoup de sens. Une adaptation que nous avons utilisée au début de notre recherche lorsque nous nous intéressions à l'optimisation du réservoir a été de remplacer le calcul du rang par la moyenne de la distance 2 à 2 des lignes de la matrice. On obtient ainsi une mesure de combien en moyenne deux lignes d'une matrice sont distantes.

Qualité du noyau

La qualité du noyau mesure la capacité de discrimination du réservoir, en d'autres termes, avec quelle exactitude il est capable de rendre compte d'entrées différentes. Pour mesurer la qualité du noyau, on envoie N entrées aléatoires de même longueur dans le réservoir, et pour chacune de ces entrées le réservoir est récolté jusqu'à obtenir une matrice de taille NxN. Le rang de cette matrice NxN fournit alors une mesure de la capacité de discrimination du réservoir.

Rang de généralisation

Le rang de généralisation est obtenu de la même façon que la qualité du noyau, sauf qu'au lieu d'envoyer N séquences complètement aléatoires les n derniers éléments de chaque séquence sont identiques. Intuitivement, le rang de la matrice NxN obtenue mesure jusqu'à quel point le réservoir est capable de généraliser les séquences en fonction de leurs n derniers éléments.

Une mesure peut être plus adaptée au ESN est sans doute la mesure de l'entropie de la distribution des $X(n)$ [26][38]. L'entropie étant une mesure de la "quantité d'information" fournie par une variable aléatoire, un réservoir avec une large entropie est par conséquent un réservoir capable de fournir une grande quantité d'information. Une notion qui revient souvent en RC est la notion de *"limite de chaos* (edge of chaos). On dit qu'un système dynamique opère à la "limite du chaos" lorsque ses paramètres sont de telle sorte que son comportement est à la limite du chaotique et du non-chaotique. C'est à cette limite que les systèmes dynamiques possédent leur plus grande puissance de calculs [57][52][36][9]. En pratique, la limite de chaos est détectée empiriquement la plupart du temps grâce à la méthode de l'exposant de Lyapunov [22]. Malheureusement, cette méthode n'est pas toujours évidente à utiliser niveau d'expertisequ'il n'existe pas de méthode systématique pour le faire.

2.3 Optimisation du réservoir

Au tout début de notre recherche, nous nous sommes intéressés à deux types de réservoirs, le premier étant composé de neurones dont la fonction d'activation est une fonction binaire [6], et le second étant un réservoir classique de ESN, c'est à dire que la fonction d'activation des neurones est directement la non linéarité *tanh*. Ces investigations premières qui nous ont permis de nous familiariser avec le RC, nous ont convaincu que les réservoirs du second type réclament moins d'attention quant au choix des valeurs de leurs hyper-paramètres, mais aussi que ces réservoirs sont moins sensibles à l'optimisation. L'une de nos constatations importantes qui a défini notre démarche dans la réalisation de ce mémoire est que la *qualité* de l'information fournie par le réservoir (au moins dans le cas des ESN) dépend plus de la nature de l'entrée propagée dans celui-ci que d'une quelconque optimisation. Nous avons par conséquent choisi de porter notre attention sur l'entrée plutôt que sur l'optimisation. Néanmoins, nous présentons ici certaines approches, en particulier celles que nous pensons être prometteuses parce qu'elles ont prouvé avoir des résultats positifs sur les LSM.

2.3.1 Optimisation des paramètres à l'aide d'une descente de gradient

Au début, de notre recherche nous avons cherché à optimiser les valeurs des différents hyper-paramètres du réservoir à l'aide d'une descente de gradient. Nous avons testé cette méthode sur deux types de réservoirs de type ESN, le premier étant composé entièrement de neurones dont l'activité est binaire, le second est un ESN classique. Nous avons réussi à obtenir quelques améliorations avec la version binaire du réservoir mais les résultats sur la version classique étaientt beaucoup plus mitigés, ce qui est en accord avec la découverte de Schrauwen et al. qui rapportent que les réservoirs avec des neurones binaires sont les plus sensibles aux changements de topologie [52]. Le fait que l'effet sur les réservoirs ESN classique soit limité est en accord avec les résultats de H. Jaeger qui suggèrent que la surface d'erreur lorsque l'on combine les hyper-paramètres du réservoir et W_{out} auraient de multiples minimums locaux [32].

6. La fonction d'activation des neurones est une fonction seuil

2.3.2 Optimisation non supervisée locale

Étant donné leur réalisme biologique les LSM se prettent bien aux algorithmes d'apprentissage Hebbien [16]. Il a par ailleurs été montré qu'utiliser un algorithme d'apprentissage comme la STPD (spike time dependent plasticity) [16][19], (qui est une forme d'apprentissage Hebbien où les variations de poids entre deux neurones dépendent à la fois de l'ordre dans lequel ils ont été activés et de la durée séparant leurs activations), peut permettre d'améliorer les capacités du réservoir [46]. Il a par ailleurs été montré que l'entraînement de réseaux de neurones récurrents, biologiquement réalistes à l'aide de STPD, réorganise l'activité du réseau de sorte que des groupes de neurones dont l'activité est synchrone se forment [62].

Malheureusement, l'application de méthodes similaires à des ESN s'est souvent soldée par des résultats beaucoup plus mitigés : l'application de l'apprentissage Hebbien et anti-Hebbien classique n'a jusqu'à présent, à notre connaissance apporté aucune amélioration significative à l'exception peut être de la règle "Anti-Oja" dont l'application à des ESN a été proposée [67].

Un autre algorithme inspiré du fonctionnement du cerveau, mais que nous n'avons pas pu expérimenter dans cette recherche, est la *plasticité- intrinsèque* (Intrinsic-Plasticity, IP) qui est une méthode qui modifie la densité de probabilité des sorties d'un neurone en une distribution exponentielle [13]. Contrairement aux méthodes d'apprentissage Hebbien, l'IP agit sur le neurone et non les synapses en modifiant l'excitabilité intrinsèque du neurone. L'application de cette méthode a donné de bons résultats sur différents types de réservoirs dont les ESN classiques [53][59]. Cette méthode est également utilisable en tandem avec des algorithmes d'apprentissage Hebbien et il a été montré qu'un réseau récurrent réaliste biologiquement et entraîné de cette façon est capable d'appendre des représentations sensorielles similaires à celles présentes dans le cortex visuel primaire [12]. L'application de l'IP et de la STPD sur des LSM peut également améliorer la robustesse de ceux-ci ainsi que la capacité de prédiction du modèle [34].

2.4 Problèmes récurrents des ESN utilisés comme modèles génératifs et bref aperçus des solutions proposées

Deux problèmes récurrents que l'on rencontre avec les ESN sont de première importance pour ce travail. Le premier est la relation entre stabilité et retour d'informations, problème de première importance puisque c'est grâce au retour d'informations que le système acquière un pouvoir computationel [40][37].

Cette difficulté se rencontre lorsque, dans le modèle génératif, le retour d'information est renvoyé au réservoir. Intuitivement, il est clair qu'avec un modèle de ce genre les petites erreurs en sortie, aussi petites soient elles, se retrouvent amplifiées et additionnées une fois dans le réservoir, ce qui peut conduire le modèle à diverger. Cet effet indésirable étant maximal lors de la phase de génération, où l'entrée est supprimée et remplacée par la sortie du modèle.

L'approche usuelle pour atténuer ce problème lors de l'apprentissage est de supprimer le retour d'informations et de le remplacer par la cible juste[29]. Malheureusement, ceci a également l'effet néfaste d'empêcher le modèle d'apprendre à s'adapter au bruit, et augmente le problème de stabilité lors de la génération. L'un des moyens utilisé est alors d'ajouter du bruit blanc au réservoir [38], mais ce bruit n'étant évidemment pas corrélé aux erreurs du modèle, cette approche n'est pas optimale puisque la situation d'apprentissage ne correspond pas vraiment à la situation que le modèle rencontrera lors de la génération. Une autre méthode est d'utiliser une régression de Ridge, et de faire en sorte que le rayon spectral de $W^* = W_{res} + W_{out} + W_{feed}$ soit inférieure à 1.

Nous avons atténué significativement ce problème dans notre modèle génératif, premièrement en utilisant un algorithme d'apprentissage où le réservoir reçoit la véritable valeur du retour d'informations pendant l'apprentissage. La situation que rencontre le modèle lors de l'apprentissage est par conséquent proche de la situation qu'il aura à gérer lors de la génération. Cet algorithme se nomme Force et son utilisation améliore significativement le modèle génératif [60]. Deuxièmement, en optant pour une approche complètement différente en utilisant des fonctions périodiques comme entrée, ce qui ne fait du retour d'informations qu'une entrée parmi d'autres.

Le deuxième problème est l'incapacité du réservoir à pouvoir rendre compte d'événements se produisant à différentes échelles de temps [38], or cette capacité est très souvent nécessaire lorsque l'on construit un modèle génératif de données réalistes issues d'observations, en musique par exemple, où les instruments de percussion d'un morceau sont rarement joués à la même fréquence. La solution que nous proposons ici aussi, est également à doubles facettes : Premièrement, nous introduisons un nouvel algorithme : *Orbite* 4.2, qui est capable d'augmenter la capacité de discrimination du modèle sans avoir à toucher au réservoir, et nous assignons un apprenant à chaque événement périodique cible. Deuxièmement, nous choisissons nos fonctions périodiques d'entrées de façon à ce qu'elles induisent une activité périodique sur plusieurs échelles de temps dans le réservoir 4.3.2.

Grâce à ces approches, notre modèle est capable de générer des séquences d'une stabilité parfaite, tout en prenant en compte des événements se produisant à des échelles de temps différentes. Ces résultats sont à notre connaissance une première dans le domaine. Nous présentons notre modèle génératif plus en détails dans le chapitre sur la génération 4.

CHAPITRE 3

RECONNAISSANCE DE RYTHMES

Au début de notre recherche, la reconnaissance de rythme ne faisait pas partie de notre sujet qui ne comprenait que la génération, mais nous avons néanmoins décidé d'ajouter cette partie à la fois, pour évaluer les capacités des ESN dans ce domaine, et pour tester un nouveau modèle d'apprentissage à l'aide d'un réseau de neurones profond (qui contient plus de trois couches). La reconnaissance de séquences temporelles est une tâche ardue puisqu'elle implique que le modèle soit capable de prendre des décisions en fonction d'un contexte en perpétuelle évolution. Or comme nous l'avons vu dans la section 2.0.3 les réseaux de neurones récurrents et en particulier les réservoirs, sont a priori les modèles idéaux pour ce genre de tâches, puisqu'ils conservent justement une représentation non-linéaire de l'historique d'entrées dans leurs dynamiques, les ESN ont par ailleurs déjà été appliqués à la reconnaissance de séquences avec un certain succès [27].

séquences temporelles construit par dessus un ESN, ainsi que les résultats que nous avons obtenus sur l'ensemble de données anthems [54].

3.1 Architecture du modèle

Deux séquences rythmiques peuvent être très différentes ou si proches qu'elles ne diffèrent l'une de l'autre que sur quelques quarts de temps. Pour être efficace, le modèle doit pouvoir récolter de l'information contextuelle à différents moments et ensuite rassembler ces informations d'une façon qui lui permette de prendre la bonne décision. C'est en prenant compte de ces contraintes que nous avons bâti l'architecture de notre modèle qui comprend trois parties distinctes :

- Un réservoir Li-ESN 2.2.5 qui joue le rôle d'une mémoire dynamique et dont le rayon spectral est légèrement inférieur à 1.
- Un premier niveau d'apprenants dont le rôle est d'extraire du réservoir une représen-

tation de plus haut niveau, et mieux définie de l'historique d'entrées.

– Un second niveau dont le rôle est de fournir la décision finale du modèle à partir des sorties du premier niveau.

Étant donné que le réservoir est celui d'un ESN à fuite comme nous l'avons vu dans la section 2.2.5, nous ne présentons dans ce qui suit que les premiers et seconds niveaux d'apprentissage.

3.2 Apprentissage

L'apprenant du modèle est une structure construite à partir de réseaux de neurones à une couche cachée (MLP) 2.0.1. En entraînant le premier puis le second niveau, nous évitons le problème de disparition du gradient (voir sec. 2.1.4) et nous sommes capables d'entraîner un réseau de neurones possédant au final quatre couches cachées avec une bonne précision.

3.3 Premier niveau d'apprentissage

Le premier niveau d'apprentissage est composé de L réseaux de neurones à une couche cachée (voir sec. 2.0.1), possédant chacun $N+1$ entrées, où N est la taille du réservoir, l'entrée supplémentaire est l'entrée courante du modèle, et R sorties, où R est le nombre de rythmes à reconnaître. La non-linéarité de la couche cachée est $tanh$ et la non linéarité de sortie est une *softmax* [1]. Chaque apprenant ne voit l'état du réservoir qu'au temps $t = t_i$ ou t_i étant une constante différente pour chaque apprenant. Ainsi au lieu d'avoir à apprendre à généraliser sur l'ensemble des états du réservoir induit par la séquence, chaque apprenant n'apprend à généraliser que sur l'ensemble des $t = t_i$ lui correspondant, son ensemble d'apprentissage est ainsi beaucoup plus restreint et il est à même de fournir une représentation précise et locale. Comme nous avons utilisé *softmax* comme non-linéarité de sortie, la sortie de chaque apprenant de la première couche peut être interprétée comme un vecteur de probabilités, ce qui nous donne :

1. La particularité de cette fonction et que toutes les sorties de l'apprenant sont dans $[0,1]$ et que leur somme est égale à 1, ce qui permet d'interpréter les sorties comme des probabilités [10]

$$P(r,t_i)_{|(X(t_i),U(t_i))} = l_1^i(X(t_i)|U(t_i))[r] \tag{3.1}$$

où la probalité $P(r,t_i)_{|X(t_i),U(t_i))}$ que le rythme r soit le rythme courant au temps t_i étant donné l'état du réservoir $X(t_i)$ et l'entrée $U(t_i)$, est approximé par le r^{ieme} élèment du vecteur retourné par l'apprenant correspondant au temps $t = t_i$.

3.4 Second niveau d'apprentissage

Le second niveau d'apprentissage est un autre MLP identique à ceux du premier niveau, qui possède $R * L$ entrées et R sorties. Le rôle de cette unité est de fournir la sortie finale du modèle en fonction des données renvoyées par le premier niveau. Une fois que la séquence a été envoyée au modèle, les sorties des L MLP du premier niveau sont regroupées au sein d'un vecteur qui sert d'entrée au second niveau. Par conséquent, la sortie de ce niveau est telle que :

$$Q(r) = l_2(l_1^{0..L})[r] \tag{3.2}$$

où la probabilité $Q(r)$ que le rythme courant soit le rythme r est approximé par la sortie de l'apprenant du second niveau étant donné les sorties des L apprenant de la première couche.

En combinant les sorties des apprenants de la première couche, l'apprenant de la seconde couche obtient une représentation de la séquence sur une vaste période de temps. Cette représentation est également de plus haut niveau que celle fournie directement par le réservoir puisque prétraitée par le premier niveau. Si l'on considère la sortie de chaque apprenant du premier niveau comme étant un vote sur la nature de la séquence jouée en fonction de l'état du réservoir et de l'entrée à un certain temps. Le rôle du second niveau d'apprentissage est alors de regrouper l'intégralité de ces votes et de les combiner pour fournir la décision finale. La figure 3.1 montre l'intégralité du modèle ainsi que son fonctionnement.

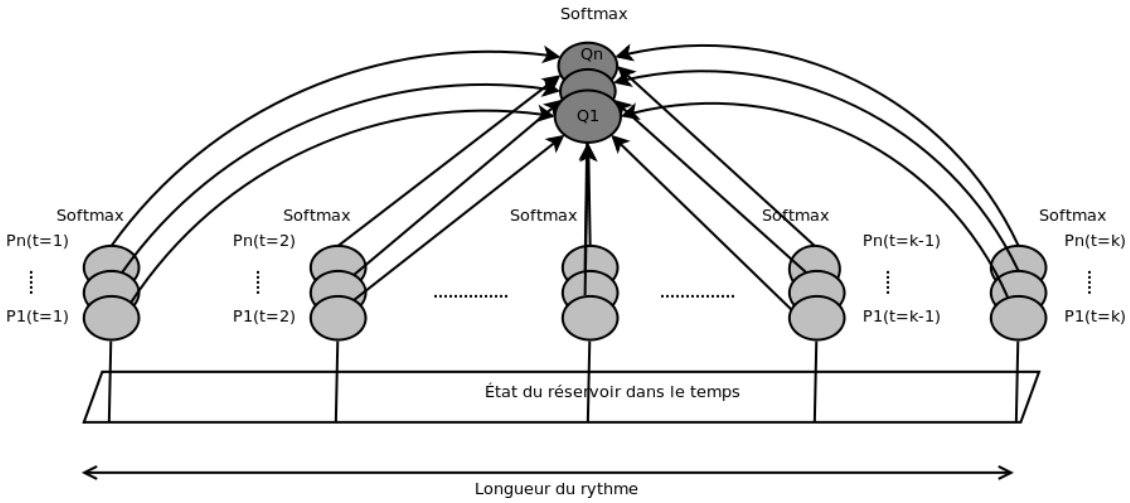

Figure 3.1 – Achitecture du modèle de reconnaissance de rythmes.

3.5 Protocole expérimental

Entraînement

Premièrement, nous créons un réservoir de type Li-ESN avec un rayon spectral de 0.9 et nous définissons une constante O qui indique le décalage temporel entre deux apprenants du premier niveau. Ainsi, si l'apprenant i apprend en fonction de l'état du réservoir à l'instant $t = t_i$ l'apprenant $i+1$ apprend lui en fonction de l'état à l'instant $t = t_i + O$. L'entraînement commence par celui du premier niveau. Pour chaque séquence de l'ensemble de données, nous commençons par réinitialiser le réservoir avant d'envoyer plusieurs fois l'intégralité de la séquence dans le réservoir. Une fois que l'activité du réservoir est très influencée par la séquence, l'apprentissage commence et dure un certain nombre d'époques. Une fois que l'entraînement du premier niveau est terminé, on passe à celui du second niveau dont le protocole est très similaire à celui du premier. Pour chaque séquence, on réinitialise le réservoir, avant d'y envoyer plusieurs fois l'intégralité de celle-ci. Une fois que cette étape est terminée, on envoie l'intégralité de la séquence dans le réservoir et on récupère les sorties des apprenants du premier niveau au sein d'un vecteur dont on se sert comme entrée pour le second niveau. Cette étape est répétée un certain nombre d'époques.

Test

Le protocole de test est le même que le protocole d'entraînement du second niveau, la seule différence est que l'on récolte la sortie du niveau au lieu de l'entraîner.

3.6 Résultats

Nature de la cible et Reconnaissance

Nous avons effectué nos expériences sur 57 hymnes nationaux de l'ensemble anthems **??**. Pour pouvoir utiliser cet ensemble de données avec notre modèle, nous avons tout d'abord changé l'encodage des hymnes en un encodage binaire $-1/1$, où 1 correspond à un temps joué et -1 à un silence, comme le montre la figure 3.2, la longueur des séquences obtenues va de 59 unités à 1055. Il est important de noter que notre encodage ne prend pas en compte le tempo, un peu à la manière d'une partition. Notre modèle est par conséquent insensible aux variations de tempo. Comme le montre le tableau 3.I, le taux d'érreur de classification sur les 57 sont bons, les seules erreurs que nous avons eu, ont été pour les hymnes de la Finlande, de l'Iraq et de la Pologne. Ces hymnes ont été mal classifiés comme étant les hymnes du Groenland (0.987323), du Qatar (0.992327) et du Vietnam (0.989862) . Malheureusement, nous n'avons pas eu le temps d'investiguer ces erreurs qui viennent peut être d'un manque de capacité du modèle, puisque nous ne rencontrons pas ce probléme avec un nombre de cible moins élevé. Notre modèle souffre en effet d'un handicap qui est qu'augmenter le nombre de cibles augmente le nombre d'entrées de la seconde couche de façon exponentielle. Nous proposons dans la partie discussion une solution à ce problème.

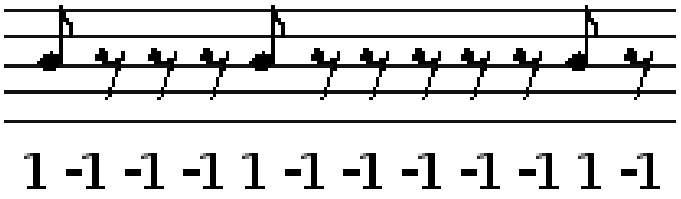

Figure 3.2 – Encodage des données utilisées.

1	ALBANIA	0.999480
2	AMERICA	0.999493
3	ANDORRA	0.998885
4	ARGENTINE	0.999107
5	BELGIUM	0.998993
6	BRAZIL	0.999072
7	BULGARIA	0.998966
8	CAMBODIA	0.999181
9	CANADA	0.999304
10	CHINA	0.999145
11	COLOMBIA	0.998856
12	CUBA	0.997908
13	ETHIOPIA	0.998551
14	FAROE-ISLANDS	0.999134
15	FINLAND	0.012299
16	FRANCE	0.999065
17	GREECE	0.999190
18	GREENLAND	0.987323
19	GUATEMALA	0.998921
20	HAITI	0.999459
21	HONDURAS	0.995919
22	INDONESIA	0.998968
23	IRAN	0.999393
24	IRAQ	0.005710
25	ISLE-OF-MAN	0.997190
26	ISRAEL	0.999166
27	ITALY	0.999536
28	KENYA	0.999212
29	KUWAIT	0.998610
30	LAOS	0.995013
31	LEBANON	0.998517
32	LIBYA	0.999416

Tableau 3.I – Taux d'érreur de classification pour les 57 hymnes de l'ensemble de données.

33	MOROCCO	0.999371
34	NEW-ZEALAND	0.999190
35	NICARAGUA	0.998743
36	NORWAY	0.998594
37	PAKISTAN	0.999556
38	PANAMA	0.998763
39	PARAGUAY	0.999624
40	PERU	0.999870
41	POLAND	0.010008
42	QATAR	0.992328
43	RUMANIA	0.998372
44	SOUTH-AFRICA	0.999729
45	SPAIN	0.999761
46	SUDAN	0.999114
47	SURINAM	0.999753
48	SWEDEN	0.999298
49	SWITZERLAND	0.998832
50	THAILAND	0.999303
51	UNITED-ARAB-REPUBLIC	0.999478
52	URUGUAY	0.999192
53	USSR	0.996454
54	VATICAN	0.999253
55	VENEZUELA	0.997980
56	VIETNAM	0.999111
57	YUGOSLAVIA	0.989862

Tableau 3.II – Taux d'érreur de classification pour les 57 hymnes de l'ensemble de données.

Résistance au bruit sous la forme de variations

Il arrive souvent que l'on observe des variations dans une séquence, dans le cas de séquences de rythmique il arrive souvent que les musiciens introduisent des syncopassions. Pour tester la résistance de notre modèle bruit nous avons introduit des variations de ce type en :

- Remplaçant des silences par des notes jouées.
- Remplaçant des notes jouées par des silences.
- Appliquant les deux transformations.

La méthode que nous avons utilisé est la suivante : à chaque temps la valeur de l'entrée a une probabilité a d'être altérée. Cette façon de procéder transforme très rapidement la nature de la séquence surtout dans le cas où les deux transformations sont appliquées. Les résultats obtenus sont visibles sur la figure 3.3. Dans le cas où l'on remplace les silences par des coups, on remarque que les performances du modèle décroissent quasilinéairement à mesure que a augmente, même lorsque la séquence est à 80% différente, le modèle est capable de fournir 20% de réponses correctes.

Les résultats pour la seconde transformation sont moins bons, ce qui est intéréssant puisque les notes étant moins fréquentes que les silences, cette tranformation altére quantitativement moins l'ensemble d'entrainement que la première, mais cette même raison implique que les notes véhiculent une quantité d'informations supérieure. Les supprimer de la séquence prive ainsi le modèle d'informations discriminantes essentielles. Comme on pouvait s'y attendre, c'est lorsqu'on applique les deux transformations en même temps que les performances sont les moins bonnes, cette façon de faire transforme en effet radicalement la séquence d'une façon qui la rend rapidement méconnaissable même à un auditeur averti.

3.7 Conclusion et discussion

Dans cette section, nous avons proposé un nouveau modèle pour la reconnaissance de séquences temporelles basé sur le RC. Notre modèle comporte une architecture en trois niveaux avec, un réservoir qui sert de mémoire dynamique, un premier niveau

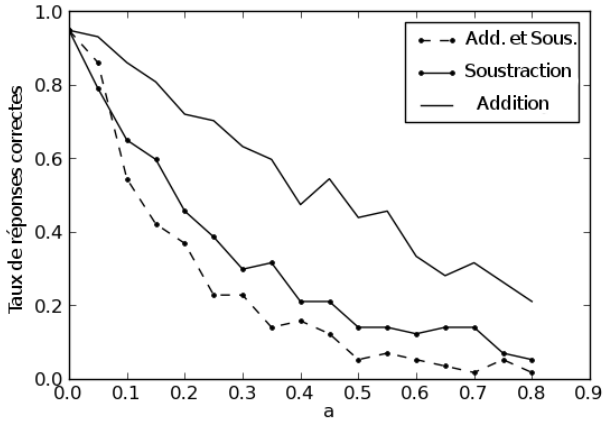

Figure 3.3 – Résultats du modèle en fonction des différentes formes d'altération.

d'apprenant dont le rôle est de fournir une représentation contextuelle d'un niveau plus haut que celle du réservoir et un deuxième niveau d'apprentissage qui utilise les entrées du premier niveau pour rendre la décision finale. Nous avons testé notre modèle sur 57 hymnes nationaux de l'ensemble anthems [54], et avons obtenu de bons résultats pour la reconnaissance avec une assez bonne résistance au bruit. Malheureusement, nous n'avons pas pu expérimenter avec ce modèle, que nous pensons prometteur, autant que nous l'aurions voulu et il reste sans doute beaucoup de choses à faire. Nous pensons par exemple à investiguer l'effet d'ajout de bruits dans l'ensemble d'apprentissage, ou à remplacer les MLP du premier niveau par des auto-encodeurs. En effet, le rôle de la première couche n'est pas de fournir une réponse mais une représentation de plus haut niveau que celle du réservoir. Le principal handicap de notre architecture qui utilise des MLP au premier niveau, est que la cardinalité de la sortie de chaque MLP est le nombre de séquences cibles. Ainsi, en augmentant le nombre de séquences cibles, on augmente aussi la cardinalité de l'entrée de la seconde couche. Les auto-encodeurs sont en quelques sortes des réseaux de neurones à une couche cachée dont la cardinalité est en général inférieure à celle de l'entrée et dont le but est d'apprendre à reproduire l'entrée. L'activation de la couche cachée d'un auto-encodeur est par conséquent une représentation de l'entrée dans un espace à moindre dimension. Ainsi, en regroupant les activations des couches cachées on obtiendrait une représentation de plus haut niveau

mais dont la cardinalité n'augmenterait pas avec le nombre de cibles. Notre idée est donc de remplacer les MLP du premier niveau par des "denoising auto-encoders" [65], que l'on entraînerait préalablement avant de les priver de leur couche de sortie pour ne se servir que des activations de leurs couches cachées comme entrées pour le second niveau. Il serait également intéressant d'essayer d'autres types d'apprenants pour le second niveau, ou même d'imaginer une architecture encore plus profonde à partir de réseaux de neurones profonds [23].

CHAPITRE 4

GÉNÉRATION DE RYTHMES

Nous commencerons ce chapître par une présentation des deux algorithmes que nous avons utilisé pour l'apprentissage à savoir *Force* et *Orbite*, que nous avons développé dans le cadre de ce mémoire. Par la suite, nous présenterons l'intégralité de notre modèle.

4.1 Force

4.1.1 Généralités

L'algorithme d'apprentissage Force à été proposé par Sussilo et Abbott en 2009 [60] comme une alternative à la régression linéaire permettant d'obtenir des sorties complexes plus fiables et stables dans le cas où la fonction cible est une fonction périodique. Cet algorithme développé entre autre pour modéliser l'apprentissage moteur, est particulièrement adapté pour être utilisé en tandem avec de larges réseaux récurrents au comportement chaotique. Il peut par ailleurs se plier à différentes architectures et peut même servir à apprendre en l'absence d'entrée voir à entraîner le réservoir. Mais cet aspect dépasse le cadre de ce mémoire puisque nous ne nous intéressons qu'au cas où le réservoir est fixe et où la totalité de l'apprentissage se passe en dehors de celui-ci. Lorsque l'on tente de faire de l'apprentissage en utilisant de larges réseaux de neurones récurrents, il faut garder à l'esprit certains problèmes récurrents, en particulier ceux liés au retour d'informations (voir sec. 2.4). En effet, le fait d'envoyer des retours d'informations erronés au réservoir peut avoir pour effet de faire diverger entièrement le modèle et de compliquer grandement l'apprentissage [38]. Le réservoir de par sa nature, un large réseau de neurones parcimonieusement connectés, est particulièrement sensible à ce genre de stimulations qui une fois introduites peuvent perturber son activité sur une très longue période de temps, à cause des "échos" qu'elles engendrent.

Ce problème est résolu par exemple dans les travaux du Dr. Jaeger en verrouillant complètement le retour d'informations pendant la durée de l'apprentissage et en envoy-

ant à la place la valeur exacte de ce que devrait être le retour d'informations [29]. Mais cette méthode expose le réseau au problème de sur-apprentissage puisqu'elle l'empêche de s'adapter aux éventuelles divergences que présentera le retour d'informations lors de la génération. Ce problème peut néanmoins être atténué en ajoutant par exemple du bruit au réservoir [38]. Sussilo et Abbot proposent quant à eux une approche différente dont le fondement est de propager le retour d'informations dans le réservoir pendant l'apprentissage.

4.1.2 Force plus en détails

Dans cette partie, nous nous intéresserons plus en détail au fonctionnement de l'algorithme d'apprentissage Force.

Les équations régissant l'apprentissage sont les suivantes :

Soit Z la sortie du réseau, T la fonction cible, X l'activité du réservoir et W_{out} le vecteur de poids entre le réservoir et la sortie. La sortie au temps t est obtenue en multipliant la transposée du vecteur de poids par l'activité du réservoir au temps t :

$$Z(t) = W_{out}^{T}(t)X(t) \tag{4.1}$$

Le réseau possède aussi une constante temporelle Δt, pour des raisons de cohérence biologique, toutes les modifications de poids et les calculs d'erreur se font en fonction de cet hyper-paramètre, auquel Sussilo et Abbot donnent généralement la valeur 0.1 [61]. Ainsi l'erreur calculée par le modèle avant le processus d'apprentissage est :

$$e(t) = W_{out}^{T}(t-\Delta t)X(t) - T(t) \tag{4.2}$$

Et une fois le processus passé :

$$e_f(t) = W_{out}^{T}(t)X(t) - T(t) \tag{4.3}$$

Le critère de convergence étant que :

$$\frac{e_f(t)}{e(t)} = 1 \tag{4.4}$$

L'apprentissage commence à $t = \Delta t$. Pour que l'algorithme puisse fonctionner, il faut que la convergence soit rapide. Autrement dit que les premières valeurs calculées de $e(t)$ soient faibles. Le modèle entre par la suite dans une phase de contrôle où le but est de faire tendre la quantité $\frac{e_f(t)}{e(t)}$ vers 1.

Selon Sussilo et Abbot, plusieurs lois de modification des poids peuvent être utilisées avec Force, mais leur choix s'est porté sur l'algorithme des *moindres carrées récursifs* (Recursive least-squares, RLS)[22] qui permet d'obtenir une convergence plus rapide qu'avec la méthode des moindres carrés [50]. La régle de modification des poids est donc :

$$W_{out}(t) = W_{out}(1 - \Delta t) - e(t)P(t)X(t) \tag{4.5}$$

uù $P(t)$ est une matrice $N * N$ mise à jour en même temps que les poids selon la règle suivante :

$$P(t) = P(t - \Delta t) - \frac{P(t - \Delta t)X(t)X^T(t)P(t - \Delta t)}{1 + X^T(t)P(1 - \Delta t)X(t)} \tag{4.6}$$

La valeur initiale de P étant :

$$P(0) = \frac{I}{a} \tag{4.7}$$

Où I : est la matrice identité et a une constante.

On constate par ailleurs que l'équation 4.5 a la forme d'une règle d'apprentissage delta [16] usuelle où le taux d'apprentissage unique est remplacé par une matrice de taux d'apprentissage P. C'est le choix de la valeur de l'hyper-paramètre a qui conditionne la vitesse d'apprentissage : plus la valeur de a est petite et plus l'apprentissage est rapide. Il en résulte que la valeur de a doit être choisie avec soin, puisqu'une valeur trop petite serait à l'origine de problèmes de convergences évidents en condamnant les valeurs des poids à osciller sans cesse entre des valeurs extrêmes, alors qu'avec une valeur trop

grande, la convergence pourrait être trop lente pour que l'algorithme puisse entrer et maintenir la phase de contrôle afin de garder l'erreur petite. Pour éviter ces problèmes, Sussilo et Abbot préconisent de choisir une valeur de a comprise entre 1 et 100 très inférieure au nombre de neurones du réservoir.

A ces trois hyper-paramètres s'ajoute un autre de moindre importance, g_{inn} qui gouverne la magnitude des connexions intérieures du réservoir. Ce paramètre conditionne le comportement interne du réservoir, puisqu'un $g_{inn} > 1$ [66][56] donne au réseau une activité chaotique spontanée qui rend possible l'apprentissage en l'absence d'entrées [1]. Sussilo et Abbot ont montré par ailleurs que l'apprentissage est plus efficace si $g_{inn} > 1$ et qu'une augmentation de la valeur de g_{inn} s'accompagne d'un gain au niveau de la robustesse de l'apprentissage tout en réduisant le nombre d'iterations nécessaires à la convergence. Mais il y a une limite au delà de laquelle le retour d'informations n'est plus assez fort pour contrecarrer l'activité chaotique du réservoir. Selon les expériences des auteurs, cette limite se situerait au delà de 1.56.

Le fait d'utiliser une valeur de g_{inn} supérieure à 1 tranche aussi radicalement avec l'approche classique des ESN, où le réseau est en général complètement inactif en l'absence d'entrée. La matrice de poids du réservoir des ESN classiques possèdent un rayon spectral [2] inférieur à 1 (voir sec. 2.2.6), or augmenter g_{inn} conduit inévitablement à augmenter le rayon du spectre.

4.2 Orbite

Orbite est un algorithme que nous avons développé pour l'apprentissage et la génération de séquences périodiques. Il permet d'obtenir une convergence très rapide, une génération très stable et ce indépendamment de la longueur de la séquence à apprendre. Il a également besoin de moins de ressources de calculs puisque des réservoirs de quelques centaines de neurones sont généralement suffisants pour obtenir de bons résultats, ce qui diminue énormément le temps nécessaire à l'apprentissage. Le principe

1. Un résultat similaire à été démontré pour des ESN dont le rayon spectral de la matrice Interne est supérieure à 1 [38]
2. valeur de la valeur absolue la plus grande

d'Orbite est d'augmenter la capacité du modèle sans toucher à la taille du réservoir en répartissant cette capacité à travers le temps. Bien qu'il puisse être utilisé avec n'importe quel algorithme d'apprentissage, et devenir en quelque sorte une généralisation de celui-ci, nous avons décidé d'utiliser Force étant donné la grande stabilité de génération obtenue avec cet algorithme [60].

4.2.1 Équations :

L'idée derrière l'algorithme est de converger non pas vers une matrice fixe de poids, mais plutôt vers une séquence périodique de τ matrices fixes. À la fin de l'apprentissage, le modèle transite d'une matrice de poids à une autre, comme une planète graviterait en suivant son orbite.

Les équations régissant le comportement d'Orbite utilisées avec Force sont quasiment identiques à celles de ce dernier, à la différence que l'on ajoute l'hyper-paramètre entier τ.

Ainsi l'erreur calculée par le modèle avant le processus d'apprentissage est :

$$e(t) = W_{out}^T(t - \Delta t - \tau)X(t) - T(t) \tag{4.8}$$

Et une fois le processus passé :

$$e_f(t) = W_{out}^T(t)X(t) - T(t) \tag{4.9}$$

Les poids sont modifiés selon :

$$W_{out}(t) = W_{out}(1 - \Delta t - \tau) - e(t)P(t)X(t) \tag{4.10}$$

où P évolue de la façon suivante :

$$P(t) = P(t - \Delta t - \tau) - \frac{P(t - \Delta t - \tau)X(t)X^T(t)P(t - \Delta t - \tau)}{1 + X^T(t)P(1 - \Delta t - \tau)X(t)} \tag{4.11}$$

Sa valeur initiale étant toujours :

$$P(0) = \frac{I}{a} \tag{4.12}$$

Au vu de ces équations, la relation entre Orbite et Force devient très clair, puisque Force n'est que le cas où $\tau = 0$.

Dans la suite de cette présentation, étant donné que nous ne sommes pas intéressés par la modélisation de fonctions biologiques nous choisissons pour des raisons de lisibilité de poser $\Delta t = 1$ et de l'omettre dans les prochaines équations, nous ajoutons seulement la condition suivante : $\tau \geq 1$.

On a donc :

$$e(t) = W_{out}^T(t-\tau)X(t) - T(t) \tag{4.13}$$

$$W_{out}(t) = W_{out}(1-\tau) - e(t)P(t)X(t) \tag{4.14}$$

$$P(t) = P(t-\tau) - \frac{P(t-\tau)X(t)X^T(t)P(t-\tau)}{1+X^T(t)P(1-\tau)X(t)} \tag{4.15}$$

Où $W_{out}^T(t-\tau)$ et $P(t-\tau)$ représentent respectivement la matrice de poids et la matrice P, au temps $t-\tau$. L'algorithme n'apprend donc pas en fonction de l'erreur au temps précédent, mais plutôt en fonction de l'erreur $t-\tau$ temps plus tôt. Cette caractéristique est au coeur de l'algorithme, nous en parlons plus en détails dans la partie suivante.

4.2.2 Explications

Orbite n'utilise pas une mais τ matrices de poids évoluant indépendamment les unes des autres, pour y voir plus clair introduisons premiérement une fonction K qui retourne les poids aux temps t, on a donc :

$$W_{out}(t) = K(W_{out}(t-\tau), P(t), X(t), T(t)) \tag{4.16}$$

Or $P(t)$ ne dépendant à son tour que de $P(t-\tau)$ et de $X(t)$, K peut donc également prendre la forme suivante :

$$W_{out}(t) = K(W_{out}(t-\tau), P(t-\tau), X(t), T(t)) \tag{4.17}$$

Il nous est maintenant aisé de suivre l'évolution de W_{out} au cours du temps :

$$W_{out}(t-\tau) = K(W_{out}(t-2\tau), P(t-2\tau), X(t-\tau), T(t-\tau))$$

$$W_{out}(t-\tau+1) = K(W_{out}(t-2\tau+1), P(t-2\tau+1), X(t-\tau+1), T(t-\tau+1))$$

$$W_{out}(t-\tau+2) = K(W_{out}(t-2\tau+2), P(t-2\tau+2), X(t-\tau+2), T(t-\tau+2))$$

$$\vdots$$

$$W_{out}(t) = K(W_{out}(t-\tau), P(t-\tau), X(t), T(t))$$

$$W_{out}(t+1) = K(W_{out}(t-\tau+1), P(t-\tau+1), X(t+1), T(t+1))$$

$$W_{out}(t+2) = K(W_{out}(t-\tau+2), P(t-\tau+2), X(t+2), T(t+2))$$

On peut voir à travers cet exemple que $W_{out}(t+m)$ ne dépend effectivement que des $W_{out}((t-m) \bmod \tau)$ et des $P((t-m) \bmod \tau)$ précédents. Tout se passe donc comme si l'algorithme possédait τ matrices de poids évoluant complètement indépendamment les unes des autres, la k^{ieme} matrice W_{out} n'apprenant que la correspondance entre les états du réservoir et la cible aux temps $(t+k) \bmod \tau$. Il est également important de noter que tous les W_{out} partagent les mêmes conditions initiales et évoluent par conséquent dans des répliques identiques mais indépendantes du même espace d'états (voir sec. 2.1.1). Il est en effet nécessaire, qu'à la fin de l'apprentissage toutes les matrices puissent être rassemblées dans un espace commun, la figure 4.1 résume l'évolution du modéle. Bien qu'il ne soit pas nécessaire pour cela d'initialiser toutes les matrices aux mêmes valeurs, le fait de procéder ainsi va nous permettre une fois l'apprentissage terminé, de réduire le nombre de matrices nécessaires à la génération (voir sec. 4.2.3).

Le principe fondamental d'Orbite est à la fois d'augmenter la capacité du modèle en augmentant la taille de la partie apprenante du modèle par rapport à la non-apprenante (le réservoir), mais également de distribuer cette capacité d'une façon qui convienne à

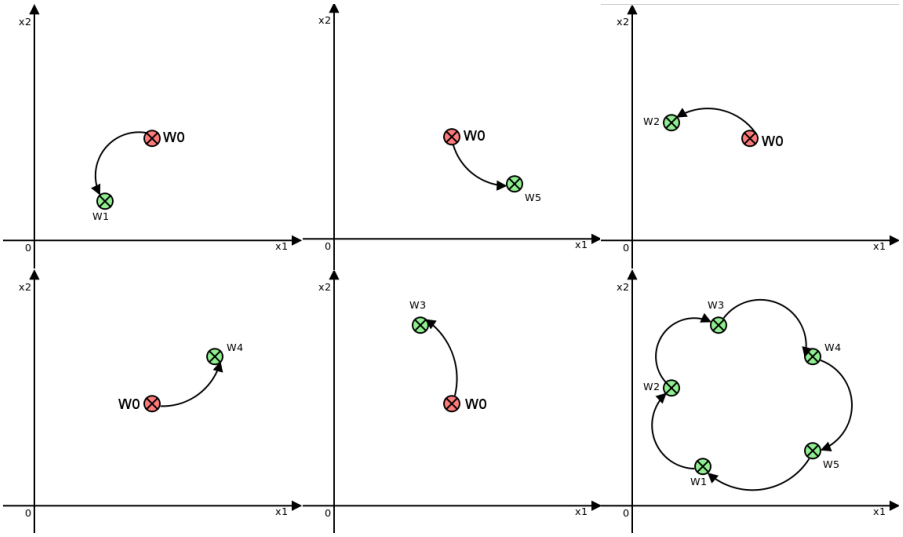

Figure 4.1 – Évolution d'Orbite lors de l'apprentissage. Chaque matrice évolue indépendamment dans sa propre copie de l'espace d'état. À la fin de l'entrainement, toutes les matrices sont rassemblées dans le même espace et l'algorithme passe de l'une à l'autre lors de la génération.

la nature de la cible, ici une séquence périodique. Cette façon de faire augmente la capacité de discrimination du modèle sans avoir avec à augmenter celle du réservoir 2.2.6, ce qui constitue un avantage certain puisque comme le montre la figure 4.4, le temps nécessaire pour effectuer une passe augmente exponentiellement avec la taille du réservoir. Ceci explique la rapidité de convergence, la robustesse et l'économie du modèle, puisqu'au lieu de devoir converger vers une solution prenant en compte l'intégralité des couples $(X(t), T(t))$ vu au cours de l'apprentissage, l'algorithme converge indépendamment vers τ sous ensembles de l'ensemble de ces couples. Ceci explique comment Orbite permet d'améliorer les capacité discriminantes du modèle sans avoir à toucher à celles du réservoir. En effet, le fait d'avoir des activations du réservoir similaires correspondant à des cibles différentes n'est pas un problème tant que les couples formés de ces activations et de ces cibles sont gérrés par des matrices de poids différentes. On voit également pourquoi les résultats de l'apprentissage ne dépendent pas directement de la longueur de la séquence cible, puisqu'il suffit de changer la valeur de τ pour être capable d'apprendre des séquences plus longues. Il faut également noter que la "qualité" de l'information fournie par le réservoir, et donc la nature de l'entrée fournie à celui-ci joue

un rôle essentiel dans l'apprentissage.

Les figures 4.2 et 4.3 montrent les performances d'Orbite pour différentes valeurs de τ, dont $\tau = 1$ qui correspond à Force. L'une des particularités d'Orbite est que sa capacité depend essentiellement de la valeur de τ. Comme le montre les figures 4.2 et 4.3, les pérformances ne semblent pas dépendre de la compléxité de la cible puisqu'elles sont sensiblement les mêmes avec 1 ou 100 séquences. Elles ne dépendent pas non plus de la simple valeur de τ, puisque augmenter τ ne conduit pas forcément à une amélioration des résultats. En effet, comme le montre la figure 4.2 le facteur déterminant quand au choix de la valeur de τ est sa corrélation avec la longeure de la séquence à apprendre. Cette particularité d'Orbite est aussi un avantage puisqu'elle lui permet de fonctionner un peu à la maniére d'un réseaux à convolution [10]. Il est en effet possible de choisir τ pour que sa valeur corresponde à une périodicité au sein de la séquence, comme par exemple la taille d'une ou plusieurs mesures. Définir la valeur de τ permet ainsi de décider de ce que nous appelons la *"résolution"* de l'apprentissage.

4.2.3 Optimisation d'Orbite

Amélioration des résultats

Le point sensible de l'algorithme se situe au niveau du choix de τ. Malheureusement, bien choisir τ demande en général d'avoir des connaissances à priori sur la nature de la cible. De plus, si le but est la généralisation plutôt que le sur-apprentissage, il est en général plus judicieux de choisir un τ dont la valeur est inférieure à celle de la longueur de la cible.

Une solution pour atténuer les problèmes dûs au choix de τ est d'utiliser deux apprenants Orbite plutôt qu'un en introduisant un second apprenant dont la tâche est de corriger l'erreur du premier. En d'autres termes, la cible du second est l'erreur du premier à savoir $(T - Z_1)$, où T est la cible générale du modèle et Z_1 la sortie du premier apprenant. La sortie Z du modèle est ensuite obtenue en additionnant les sorties des deux apprenants :

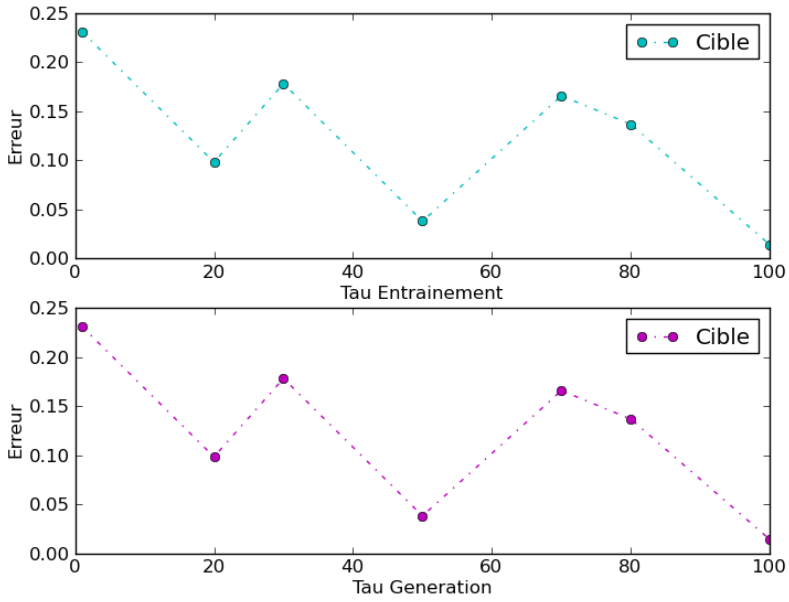

Figure 4.2 – Erreur quadratique moyenne pour différentes valeurs de τ. Les valeurs présentés ici sont issues des moyennes calcuées sur 100 séquences composées de nombres aléatoires, et de période 100. On remarque que les résultats du modèle sont les meilleurs pour $\tau = 50$ et $\tau = 100$. On remarque également que les résultats sont identiques pour l'entrainement et la génération, ce qui démontre la grande stabilité de l'algorithme.

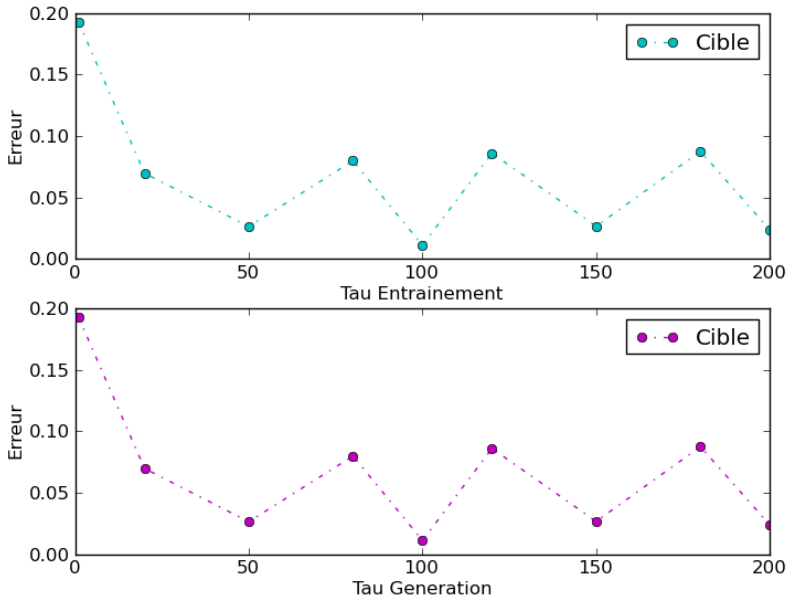

Figure 4.3 – Expérience similaire à celle de la figure 4.2 mais avec une séquence composée de 0 et de 1. Les résultats sont meilleurs pour les multiples de 50.

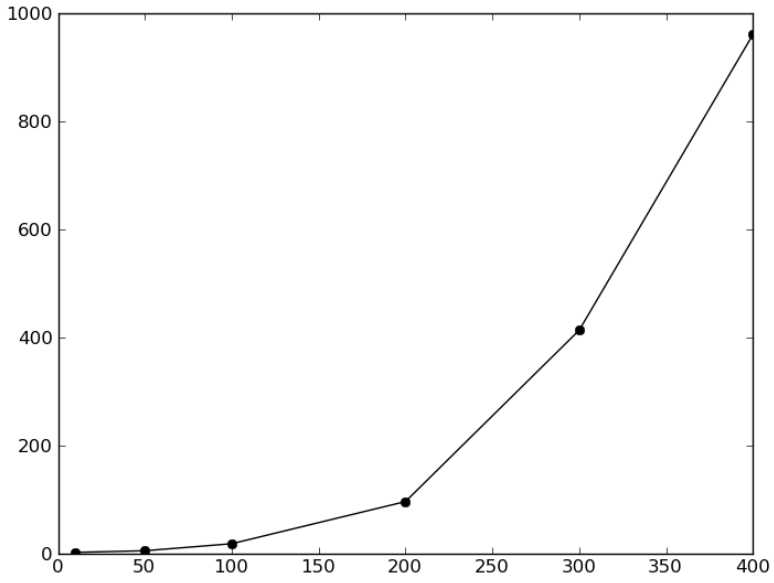

Figure 4.4 – Temps moyen en secondes pour une passe en fonction de la taille du réservoir.

$$Z = Z_1 + Z_2 \tag{4.18}$$

Par la suite, nous nommerons cette approche *Double-Orbite.*

Réduction du nombre de matrices

Dans cette partie, nous traiterons des moyens que nous utilisons pour diminuer le nombre de matrices de poids nécessaires à la génération. Les deux méthodes que nous proposons sont basées sur deux faits importants :

1. Il n'est pas possible de modifier grandement l'état du réservoir en l'espace d'un temps très court sans perturber dangereusement le modèle. En effet, le fait de couper l'entrée du réservoir n'entraîne pas une chute abrupte de son activité puisque celui-ci possède une forte activité spontanée voir autonome, et le fait d'envoyer des valeurs trop grandes en entrée ne peut que saturer le réservoir et le faire entrer dans un régime dont il est difficile de sortir et où l'apprentissage est impossible.
2. Toutes les matrices de poids partagent les mêmes conditions initiales, à savoir la même initialisation et le même état du réservoir, et leurs évolutions sont gouvernées par les mêmes valeurs d'hyper-paramètres.

De ces deux faits, nous déduisons que si $T(t \bmod \tau)$ et $T((t-1) \bmod \tau)$ sont proches (comme par exemple dans le cas où T est une fonction continue), alors il en va de même pour $W_{out}(t \bmod \tau)$ et $W_{out}((t-1) \bmod \tau)$ (la figure 4.5 illustre ce concept). Nous proposons donc deux stratégies pour réduire le nombre de matrices nécessaires à la génération. Ces deux stratégies sont appliquées une fois que le modèle a convergé.

1. **Remplacer une matrice par une autre**

   La façon la plus simple et la moins coûteuse en temps et en calculs est de remplacer une matrice par une autre. Soit m un réel, et D une distance, la méthode est la suivante :

   Pour tout i, j dans $[1, \tau]$:

   si $D(W_{out}(i), W_{out}(j)) < m$ alors :

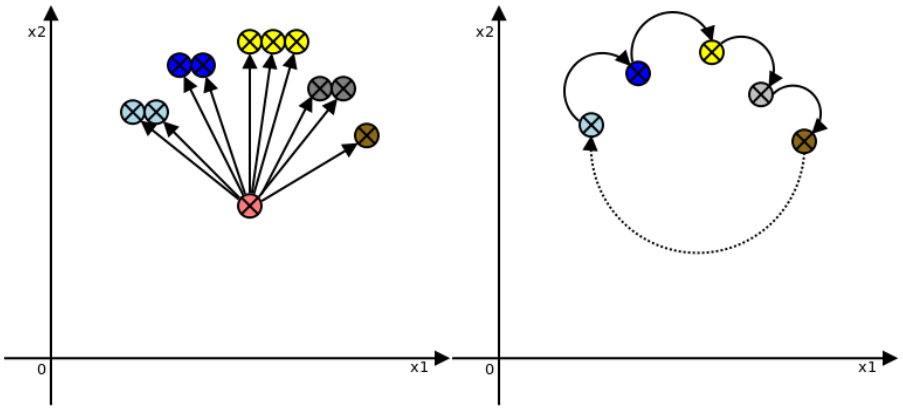

Figure 4.5 – Une fois qu'Orbite a convergé, il est possible de réduire le nombre de matrices nécéssaires à la génération en regroupant les matrices similaires.

$$W_{out}(j) = W_{out}(i)$$

2. **Rassembler les matrices**

   Il est également possible de rassembler les matrices. Soit m un réel, et D une distance, la méthode est la suivante :

   Tant qu'il existe deux matrices W^1_{out}, W^2_{out}
   telles que $D(W^1_{out}, W^2_{out}) < m$ et que l'optimisation n'a pas convergé :
   Pour tout i, j dans $[1, \tau]$:
   si $D(W_{out}(i), W_{out}(j)) < m$ alors :
   $W_{out}(j) = W_{out}(i) = Moyenne(W_{out}(j), W_{out}(i))$

   toutes deux

4.3 Le modèle génératif

Lorsque des musiciens improvisent, ils le font en général en fonction d'informations relatives au contexte courant. Ces informations peuvent être de différentes natures et l'ensemble des informations retenues dépend de la pièce jouée et du contexte lui même. Néanmoins, certaines informations sont de première importance, ce qui les amènent à être systématiquement retenues. Parmi ces informations, on note particulièrement les informations temporelles, informations grâce auxquelles le musicien arrive à déduire

par exemple la position courante dans la pièce ou dans la mesure, mais qui permettent également à n'importe qui de taper du pied au bon moment et ce, sans connaissances théoriques particulières aucune. Il est important de signaler ici que nous ne faisons pas référence à des informations conscientes de haut niveau qui requièrent un minimum de bagage en théorie musicale, mais plutôt aux mécanismes qui permettent à tout un chacun d'apprécier la musique et d'y voir une structure. En nous inspirant de cette constatation, nous avons imaginé un modèle en plusieurs parties où le générateur reçoit ses entrées de plusieurs unités dont une "Horloge" [3]. Dans la partie qui suit, nous verrons en détail la composition du modèle.

4.3.1 Présentation générale du modèle

Pour obtenir nos résultats, nous avons développé une nouvelle architecture de modèle génératif utilisant un ESN, cette approche nous a permis d'obtenir une génération d'une stabilité parfaite dont la cible est composée de séquences périodiques de périodes différentes. Le modèle génératif que nous avons développé comprend trois parties, aux fonctions bien distinctes :

- Une **Horloge** qui fournit des informations temporelles de bas niveau.
- Un **Réservoir de type Li-ESN** 2.2.5 qui sert de mémoire et dont le rôle est de conserver une représentation interne non linéaire du contexte.
- Une couche d'apprenants **Double-Orbite** (voir sec. 4.2.3) qui décide de la sortie et qui retourne l'information au réservoir.

L'intégralité du modèle ainsi que le modèle génératif généralement utilisé avec les ESN [29] sont représentés dans les figures 4.6 et 4.7. L'Horloge et la couche d'apprenants sont séparées par le réservoir qui reçoit à la fois la sortie de l'Horloge et le retour d'informations fourni par la couche d'apprenants. Ces deux stimulations excitent par ailleurs le réservoir à des endroits différents.

Les principaux obstacles que nous avons rencontrés au cours de cette recherche sont deux problèmes récurrents avec les ESN : arriver à stabiliser la génération en présence de

3. L'existence de mécanismes similaires dans le cerveau est par ailleurs une vielle hypothése qui a fait l'objet de nombreuses contributions comme celle-ci de Povel et Essens [48].

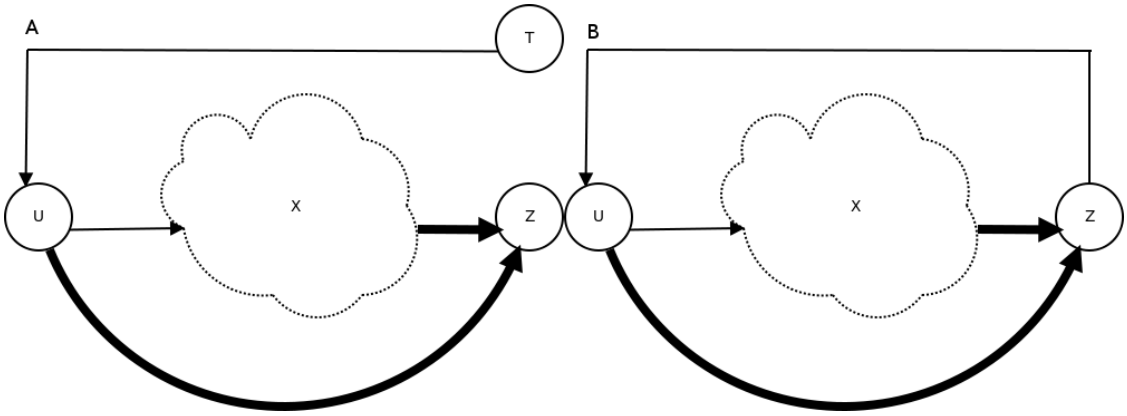

Figure 4.6 – Modèle génératif classique [29]. Durant l'apprentissage (A) le retour d'informations est remplacé par la cible, puis rétabli pour la génération (B). Seuls les poids en gras sont modifiés par l'apprentissage.

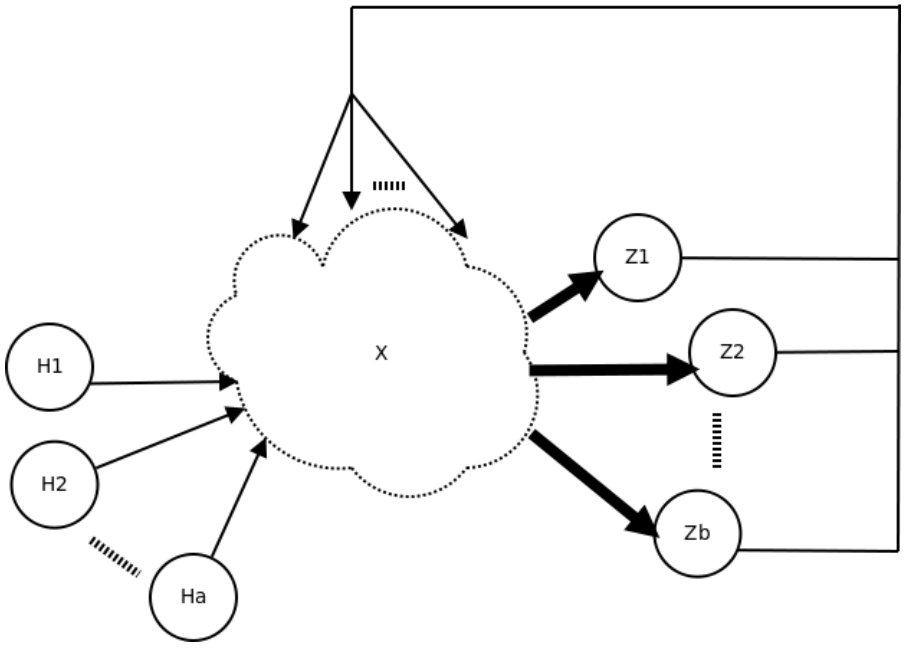

Figure 4.7 – Notre modèle génératif. Le réservoir reçoit le véritable retour d'informations en tout temps et posséde en plus de celui-ci des entrées qui lui sont fournies par une horloge. Toutes les entrées du modèle (y compris le retour d'informations) sont connectées parcimonieusement. Seuls les poids en gras sont modifiés par l'apprentissage.

retour d'informations et à être capable de rendre compte d'événements se réalisant à des fréquences de temps différentes (voir sec. 2.4). Cet aspect essentiel lorsque l'on construit un modèle génératif de données réalistes issues d'observations, est particulièrement important dans notre cas, puisque dans un morceau de musique, les différents instruments percussifs sont rarement frappés à la même fréquence. En rock par exemple, il arrive qu'il y ait 4 coups de charleston pour un coup de grosse caisse, de plus le modèle doit également être capable d'apprendre les roulements dans un morceau. Nous sommes parvenus à résoudre ces deux problèmes à la fois en utilisant des apprenants Double-Orbite (voir sec. 4.2.3) avec Force et par le recours à une horloge complexe comme entrée du modèle. Dans cette section, nous verrons plus en détails chaque élément du modèle génératif avant de présenter nos résultats.

4.3.2 L'Horloge

Le rôle de cette unité est de fournir les informations temporelles de bas niveau nécessaires au fonctionnement du générateur. Plutôt que d'essayer de prédire le prochain élément de la séquence cible seulement en fonction des précédents, nous avons opté pour une approche différente qui consiste à apprendre la transformation d'une séquence périodique (la sortie de l'Horloge) augmentée du retour d'informations, en une autre séquence périodique (la cible). Ainsi le modèle génératif possède en permanence une référence temporelle lui indiquant où il se trouve. Ceci lui évite d'avoir comme par exemple, dans d'autres types d'architecture où il ne disposerait que de son propre retour d'informations, à apprendre à "compter" le temps écoulé pour en déduire sa position actuelle. Cette façon de faire peut en effet être à l'origine de problèmes importants de stabilité lors de la génération dus aux erreurs propagées par le retour d'informations (voir sec. 2.4). En utilisant une entrée induisant une activité périodique, la tâche de l'apprenant s'en trouve grandement simplifiée puisqu'il est libéré de toute "gestion" temporelle. Moins de choses à appendre signifie en général une moindre marge d'erreur, la qualité de la génération s'en trouve alors grandement améliorée puisqu'elle gagne énormément en stabilité. Il n'y a ni accélération ni ralentissement et ce indépendamment de la longueur de la génération. Il est intéressant de remarquer qu'un article récent traitant

de la mémoire à court terme chez le macaque a mis en évidence un fonctionnement trés similaire dans le cortex préfrontal [42]. En effet, l'un des résultats importants de l'article est que plus de 80% de l'activité du substrat neuronale sert à encoder de l'information temporelle à différentes échelles. Le fonctionnement d'un réservoir avec une Horloge est très similaire à ces observations, puisque la quasi-totalité de l'activité sert à encoder une information temporelle de ce type, le reste étant dédié à la gestion de l'information fournie par le retour d'informations.

Quant à la façon de définir l'Horloge, notre intuition était qu'elle devrait être : Régulière et Périodique.

Par ailleurs les premières expériences effectuées au début de cette recherche nous ont convaincu d'ajouter une autre caractéristique : la Continuité. En effet, une Horloge continut fournie au réservoir de l'information en tout temps et conditionne ainsi beaucoup plus fortement son activité en comparaison avec une Horloge discrète qui n'aurait qu'une influence sporadique sur son activité.

Étant donné les trois caractéristiques précédentes, nous avons tout naturellement décidé d'utiliser des sinusoïdes ou des fonctions dérivées comme sortie de l'Horloge. Plusieurs types d'horloges sont alors possibles, la sortie de cette unité pouvant être unique ou multiple. Dans tous les cas, les résultats ont montré que pour un résultat optimal, les valeurs renvoyées par l'Horloge doivent pouvoir rendre compte de différents intervalles de temps, comme une horloge usuelle qui affiche l'heure, en heures, minutes et secondes. Le fait de procéder ainsi induit, en effet, une activité périodique complexe et sur différentes échelles de temps dans le réservoir. L'apprenant reçoit par conséquent, une entrée plus riche correspondant à une activité temporelle plus fine. Cette façon de faire joue un rôle primordial dans la solution que nous proposons pour résoudre la difficulté souvent rencontrée avec les ESN lorsqu'il s'agit d'apprendre des séquences complexes, nécessitant de prendre des décisions sur différentes échelles de temps (voir sec. 2.4). L'Horloge est également, une unité fondamentale qui conditionne indirectement la capacité du modèle. Un bon choix d'horloge pouvant faciliter grandement l'apprentissage.

Dans le cas où la sortie est une valeur unique, la fonction de l'Horloge peut aller

d'une simple sinusoïde à une combinaison complexe de plusieurs. Une simple sinusoïde correspondrait par exemple, à une horloge n'affichant l'heure qu'en heures, alors qu'une combinaison de plusieurs sinusoïdes serait grosso modo l'équivalent d'une horloge capable de rendre compte d'intervalles temporels plus fins et subtiles. D'un point de vue pratique, combiner plusieurs sinusoïdes permet d'obtenir des dynamiques de réservoirs plus complexes. Cette méthode a l'avantage d'être légère. Pour créer la fonction d'horloge nous procédons de la façon suivante : Nous créons tout d'abord une sinusoïde de longueur d'onde λ à laquelle nous additionnons un nombre $comp - 1$ de sinusoïdes de longueurs d'ondes de plus en plus petites, *comp* étant le nombre de sinusoïdes dont est composée chaque fonction, ou la complexité de la fonction. La fonction ainsi obtenue est ensuite normalisée pour qu'elle prenne ses valeurs dans l'intervalle $[0-1]$.

La deuxième solution est d'utiliser plusieurs fonctions de fréquences voir de phases différentes, la sortie de l'horloge devenant un vecteur plutôt qu'un scalaire, et de connecter parcimonieusement chaque sortie d'horloge à des neurones différents du réservoir. Cette méthode, qui revient à combiner plusieurs horloges du type précédent, est celle qui induit les dynamiques les plus complexes au sein du réservoir, en particulier à cause du fait que chaque fonction excite un ensemble différent de neurones. Mais cette méthode est néanmoins légèrement plus difficile à mettre en place puisqu'elle présente plus de risque de saturer le réservoir qui se retrouve avec plusieurs entrées plutôt qu'une, ou de noyer l'information du retour d'informations. Une solution est alors de normaliser l'Horloge, pour que la somme des entrées soit 1, mais là encore, on court le risque que l'influence de l'horloge sur le réservoir soit trop faible par rapport à celles du retour d'informations, en particulier si l'Horloge comprend un grand nombre de fonctions. Une horloge de ce type peut être définie par deux constantes en plus de la longueur d'onde λ, *card* qui définie le nombre de fonctions (la cardinalité de l'Horloge) et *comp* qui définie le nombre de sinusoïdes dont est composée chaque fonction. Pour obtenir des résultats optimaux, il convient donc de bien choisir les valeurs de ces deux constantes. Comme cette méthode est généralement celle qui induit les dynamiques les plus complexes au sein du réservoir, comme le montre les figures 4.8 et 4.9, c'est elle que nous avons décidé d'utiliser pour notre modèle.

On remarque également dans la figure 4.10 que la plus part des stimulations induites par l'Horloge dans le réservoir sont de même fréquence. Il y a donc des redondances dans l'information temporelle fournie par le réservoir, ce qui le rapproche encore plus d'un système biologique. Ces redondances ont sans doute un effet bénéfique et stabilisateur puisque si l'activité d'un neurone se retrouve perturbée, l'apprenant peut toujours se référer à l'information que lui fournissent d'autres neurones. Le réservoir et les entrées étant parcimonieusement connectés, une perturbation reçue par un neurone ne se reflète pas forcement sur tout le réservoir.

La façon dont nous avons défini notre horloge est la suivante :

```
Créer Horloge(lambda, card, comp, pasFreq, pasPhase) :
    Pour i de 1 à card :
        lambda = lambda/(i+1))
        h[i] = creerFonction(lambda, comp, pasFreq, pasPhase)

    Retourne h.

creerFonction(lambda, comp, pasLong, pasPhase) :

    s = une sinusoïde de longueur d'onde λ et de phase 0
    phase = 0
    pour i de 2 à comp :
        lambda = lambda + pasLong
        phase = phase + pasPhase
        s1 = une sinusoïde de longueur d'onde λ et de phase phase
        s = s + s1

    s = s/maximum(s)
    retourne s
```

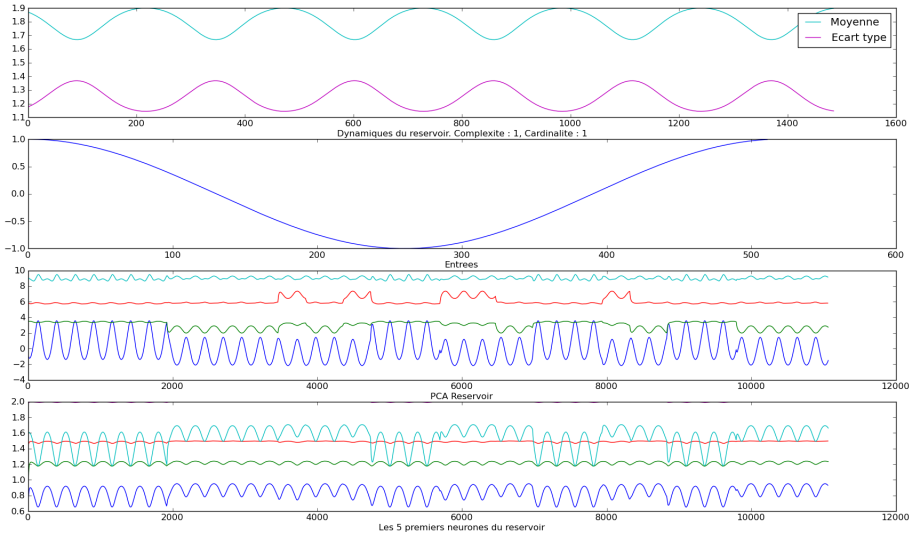

Figure 4.8 – Dynamiques induites par une horloges de cardinalité 1 et de compléxité 1.

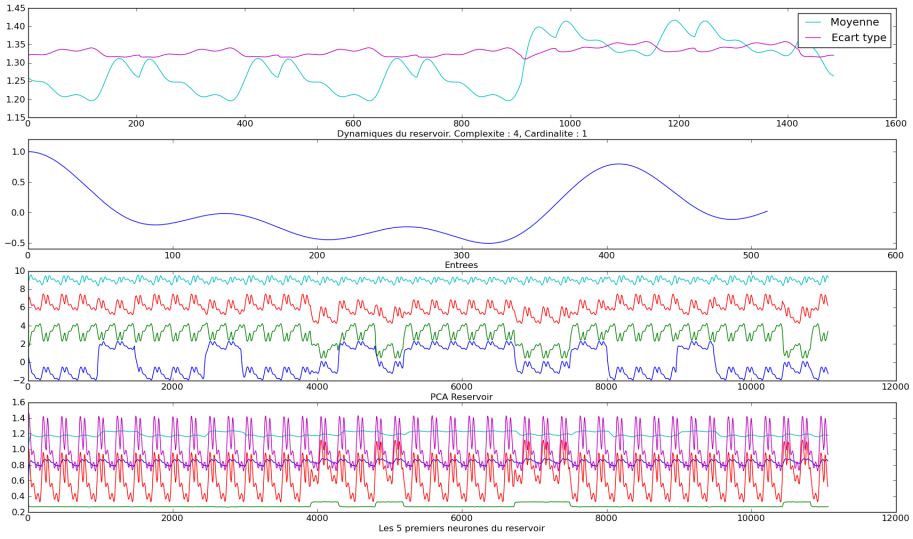

Figure 4.9 – Dynamiques induites par une horloges de cardinalité 1 et de compléxité 4.

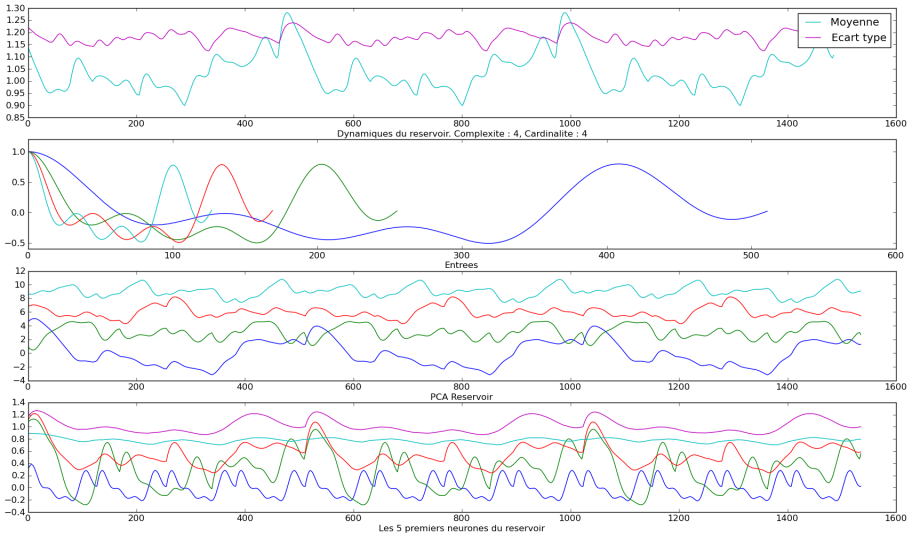

Figure 4.10 – Dynamiques induites par une horloges de cardinalité 4 et de compléxité 4. Les activités des neurones du réservoir n'ont pas toutes la même période, on voit par exemple que la période de la courbe bleue est quatre fois celle de la courbe rouge

4.3.3 La couche d'apprenants

Note modèle possède également une couche d'apprenants prenant en charge l'intégralité de l'apprentissage. Ces apprenants sont tous de type Double-Orbite (voir sec. 4.2.3) et possèdent en général les mêmes hyper-paramètres. Cependant, il est bien entendu possible de donner à chaque apprenant ses propres valeurs. Le nombre d'apprenants est le nombre d'instruments de percussion présents dans le morceau. La tâche de chaque apprenant peut se résumer à l'aide la formule suivante :

$$y_i(t) = F(H(t), Y_{0..s}(t-1), F(t-1)) \tag{4.19}$$

où $y_i(t)$ est la sortie du iéme apprenant correspondant à la iéme séquence, F est une non linéarité récurrente issue de l'activité du réservoir, $Y_{0..s}$ la sortie de tous les apprenants et H la sortie de l'Horloge.

Si l'on considère la sortie de chaque instrument comme une séquence cible de péri-

ode unique, cette approche revient donc à assigner un apprenant à chaque séquence, ce qui augmente significativement la capacité du modèle à apprendre plusieurs séquences périodiques de périodes différentes. Cette approche constitue la seconde partie de la solution que nous proposons pour palier la difficulté qu'ont les ESN à apprendre des séquences complexes nécessitant la prise de décisions sur différentes échelles de temps. Nous avons considéré la génération comme une classification à deux classes, coup et silence. Chaque fonction cible est donc une suite de 0 et de 1, de même que la sortie de chaque apprenant. Pour obtenir une telle sortie d'un modèle régressif, nous appliquons une fonction seuil à la sortie de chaque apprenant, si la valeur en sortie est supérieure au seuil, alors la classe est 1 (coup), sinon elle est 0 (silence).

4.3.4 Le Réservoir

Le réservoir est une variante du Li-ESN classique (voir sec. 2.2.5) auquelle nous avons ajouté un hyper-paramètre permettant de décider si les neurones peuvent être à la fois excitateurs et inhibiteurs. Nous avons eu l'occasion d'expérimenter cette architecture qui correspond à la réalité biologique dans le cadre du projet final du cours *"Neurosciences computationelles"* NRL6084. L'une de nos conclusions était qu'elle peut améliorer la stabilité du réservoir dans le cas où il est composé de neurones dont l'activation est binaire. En l'essayant sur un Li-ESN, nous avons remarqué que les résultats peuvent s'en trouver améliorés dans certains rares cas. Il est important de noter ici que dans notre modèle, le taux de neurones inhibiteurs est le même que celui de neurones excitateurs. Nous n'avons malheureusement pas disposé du temps nécessaire afin d'investiguer l'effet de la variation de ces taux sur les performances du réservoir. Cet aspect pourrait constituer une ouverture intéressante pour de futurs travaux. Nous ajoutons également un léger bruit au réservoir puisque ceci améliore la stabilité de la génération [38]. La plus part du temps, le réservoir comporte 100 neurones, et sa taille ne dépassant jamais 200.

4.3.5 Données d'apprentissage

Toutes nos données sont des fichiers midi *"quantisés"* au 16e de temps, ce qui veut dire, que les notes sont alignées précisément au 16e de temps près. Notre ensemble d'entraînement provient d'une base de données disponible sur internet regroupant des exercices de batterie pour différents styles musicaux [6].

4.3.6 Résultats

Nous n'avons pas pu trouver de mesure quantitative des performances du modèle, qui puisse prendre en compte l'intégralité des demandes que nous avions vis à vis des générations. Nous voulions initialement :

- Que les générations soient stables au niveau temporel, c'est à dire sans accélération ou ralentissement.
- Qu'elles soient toujours cohérentes par rapport au rythme joué et par rapport au style de musique, et qu'elles conservent "l'impression" des fichiers d'entraînement.
- Que le modèle puisse transiter de lui même d'un état à un autre, comme par exemple passer d'un rythme à un autre à l'aide d'un roulement.
- Que le modèle possède un certain degré "d'improvisation". Nous entendons par là que les générations présentent des différences par rapport à l'ensemble d'entraînement.

Le tableau 4.I résume nos résultats pour un certain nombre de rythmes générés. Ces rythmes de même que ceux qui ont servi à l'entraînement ainsi que les paramètres sont disponibles en annexe de ce mémoire. [4]

La première remarque que l'on peut faire est que la stabilité est toujours parfaite et ce, indépendamment de la longueur de la séquence générée. Ceci démontre la très bonne gestion temporelle de notre modèle. La cohérence est également très bonne pour toutes

4. Les générations peuvent êtres plus lentes ou plus rapides que l'ensemble de données, mais ceci est conséquent à la façon dont les midi ont été écrits et non de la génération. Le format midi permettant d'indiquer à quelle vitesse le morceau doit être joué, une différence de tempo entre l'ensemble d'entraînement et la génération n'a pas d'importance, ce qui importe est qu'il n'y ait pas de différence au sein du morceau.

les séquences générées, cette caractéristique est selon nous la plus importante car elle implique, que le modèle soit capable d'extraire des informations pertinentes quant à la nature du style de musique.

Les problèmes que nous avons rencontré, se situent invariablement au niveau des deux derniers critères. Nos expériences ont montré, que trouver les bons hyper- paramètres pour obtenir de bons résultats dans les deux peut parfois être difficile.

Les hyper-paramètres qui influent le plus sur la génération sont τ et la longueur d'onde de l'entrée λ. Faire varier τ permet de choisir la "résolution" de l'apprentissage, par exemple dans notre cas où chaque temps est subdivisé en 16, mettre $\tau = 64$ signifie que l'apprentissage se déroulera sur une échelle de quatre temps. Il est ainsi possible d'obtenir des résultats différents tous cohérents et stables, à partir du même ensemble de données. Comme la plus part des fichiers sur lesquels nous avons entraîné le modèle ont des mesures à quatre temps, nous avons souvent utilisé des τ multiples de 64 (une mesure) ou 96 (une mesure et demi). Quelque soit la valeur choisie de τ, pour que l'apprentissage puisse avoir lieu, il est impératif que sa valeur soit corrélée à une périodicité présente dans l'exemple d'entraînement.

Le rôle de λ est de fournir les fréquences de variation de l'activité du réservoir. Il a été montré expérimentalement lors d'expériences sur la mémoire à court terme qu'il existe une forte corrélation entre l'activité des neurones et la fréquence du stimulus [42]. Faire varier λ permet d'émuler un comportement similaire dans le réservoir. Il est possible de choisir λ comme étant ou non un multiple de τ. Dans le second cas la génération est en général légèrement plus instable ce qui conduit le modèle à produire plus de variations.

Le choix de α (le taux de fuite du réservoir) joue également un rôle non négligeable dans l'apprentissage et la stabilité du modèle. En effet un α trop grand conduit invariablement à des divergences notables. Comme nous l'avons vu dans la section 2.2.5, plus on augmente la valeur de α et plus le réservoir est sensible aux signaux de hautes fréquences, or dans notre cas le réservoir reçoit deux types de signaux, la sortie multiple de l'Horloge et le retour d'informations qui constitue un stimulus de fréquence plus élevé. Par conséquent, augmenter α revient à augmenter l'influence du retour d'infor-

mations sur le réservoir. Ceci peut avoir un effet bénéfique et forcer le modèle à donner plus d'importance à l'information véhiculée par le retour d'informations, mais si α est trop grand, une fois que le modèle passe en phase de génération et que l'effet stabilisateur de l'apprentissage n'existe plus, les erreurs véhiculées par le retour d'informations se retrouvent amplifiées dans le réservoir. La figure 4.12 illustre bien ce phénomène. On remarque que l'activité moyenne du réservoir pendant la génération n'est plus bornée, de même que l'effet que cela produit sur la séquence générée qui perd toute cohérence.

4.3.7 Présentation des résultats

BAIAO

Les principales variations contenues dans ce morceau ne sont pas dues à des changements de rythmes mais à un mini solo de congas. La tâche du modèle est par conséquent d'apprendre à reproduire la rythmique tout en étant capable de produire un solo non identique à celui du morceau d'entraînement, mais ayant de fortes similitudes avec ce dernier. Comme le montre les fichiers du dossier "Baiao", le modèle est capable de s'acquitter des deux aspects de cette tâche.

Rock

Il est difficile d'apprendre à partir d'exemples comme "2Rock" puisque tous les rythmes contenus dans le morceau sont formés à partir d'un noyau commun. Le modèle tend donc à généraliser sur ces rythmes ce qui le fait converger vers le noyau commun, la

Fichier	Stabilité	Cohérence	Transitions	Improvisation
BAIAO	Parfaite	Excellente	Bonne	Bonne
Rock	Parfaite	Excellente	Bonne	Bonne
Chacha	Parfaite	Excellente	Très Bonne	Très Bonne
Bossa	Parfaite	Excellente	Très Bonne	Très Bonne
Diddley	Parfaite	Excellente	Très Bonne	Très Bonne
Samba	Parfaite	Excellente	Bonne	Bonne

Tableau 4.I – Resultats de génération.

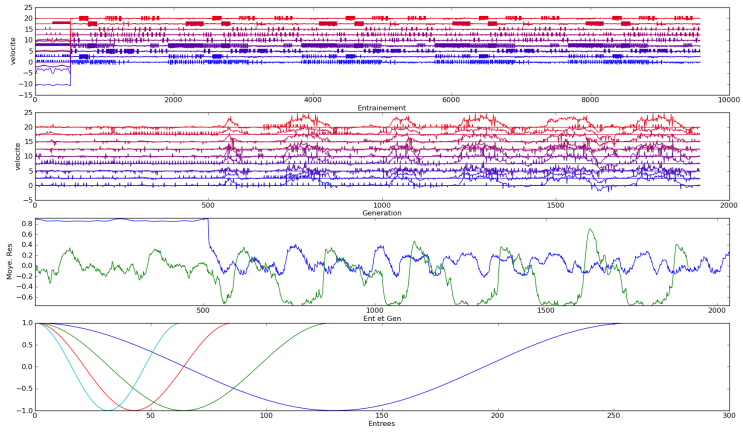

Figure 4.11 – Le modèle diverge lors de la génération à cause d'une trop grande sensibilité au retour d'informations due à une valeur d'α trop grande. Le premier graphique montre à la fois la cible et la sortie du modèle (en mirroir) lors de l'entrainement. Le second graphique présente les mêmes informations mais pour la phase de génération. Le graphique du milieu montre l'activation moyenne du réservoir, en bleu durant l'entrainement et en vert durant la génération. On remarque que l'activation durant la phase de génération est non seulement signifiactivement plus élevée que durant l'entrainement mais qu'elle tend également à augmenter, ce qui montre que le modèle a perdu le contrôle.

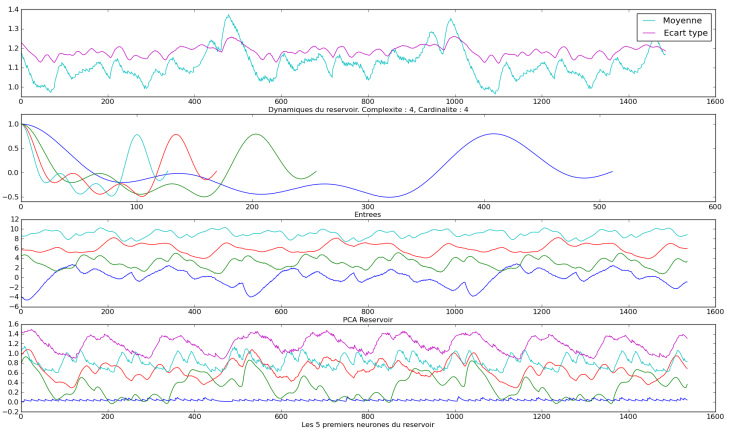

Figure 4.12 – Effet de α sur les dynamiques du réservoir. Les motifs en dents de scie que l'on peut voir sur les courbes sont dus au retour d'informations, en agmentant la valeur de α on augmente la sensibilité du modèle à celui-ci.

génération contient alors très peu de transitions et de rythmes différents. Pour contourner ce phénomène, plusieurs stratégies peuvent êtres appliquées :

- Changer la résolution de l'apprentissage au moyen de la valeur de τ. Si le noyau commun est très stable (quasiment toujours le même) à l'échelle d'une mesure, il l'est peut être pas moins à celui d'une mesure et demi ou de n mesures. Ce problème peut aussi venir d'un manque de capacité du modèle. Une simple augmentation de la valeur τ peut parfois le résoudre.
- Choisir la valeur de λ pour qu'elle ne soit pas corrélée à celle de τ. Cette approche déstabilise légèrement le modèle puisque l'activation du réservoir induite par l'Horloge dans le réservoir n'a pas la même période que l'apprentissage. Cette instabilité peut empêcher le modèle de converger vers le rythme noyau, sans trop altérer sa capacité d'apprentissage.
- Agmenter la valeur de λ. Intuitivement, plus λ est grand et plus l'appréciation temporelle du modèle est grande. On peut faire le parallèle entre la valeur de λ et le concept de profondeur de champs en photographie (voir figure 4.13). Plus la profondeur de champs est élevée, plus les objets lointains dans l'images apparaissent nets. D'une façon similaire, plus la valeur de λ est grande et plus le modèle est capable d'avoir une appréciation précise de la séquence loin dans le temps. Avec une assez bonne "profondeur de champs temporelle", le modèle peut être à même d'apprécier des motifs plus complexes et plus étendus dams le temps. En d'autre termes, augmenter λ peut permettre de faire un meilleur usage de la capacité du modèle.

Pour être capable d'obtenir des résultats sur "2Rock" nous avons à la fois augmenté la valeur de τ et celle de λ.

Chacha et Bossa

Chacha et Bossa sont de bons exemples de ce que notre modèle puisse faire. La génération est reste très cohérente avec les fichiers d'entraînement tout en présentant des différences par rapport à ceux-ci, tant au niveau des rythmes que des roulements.

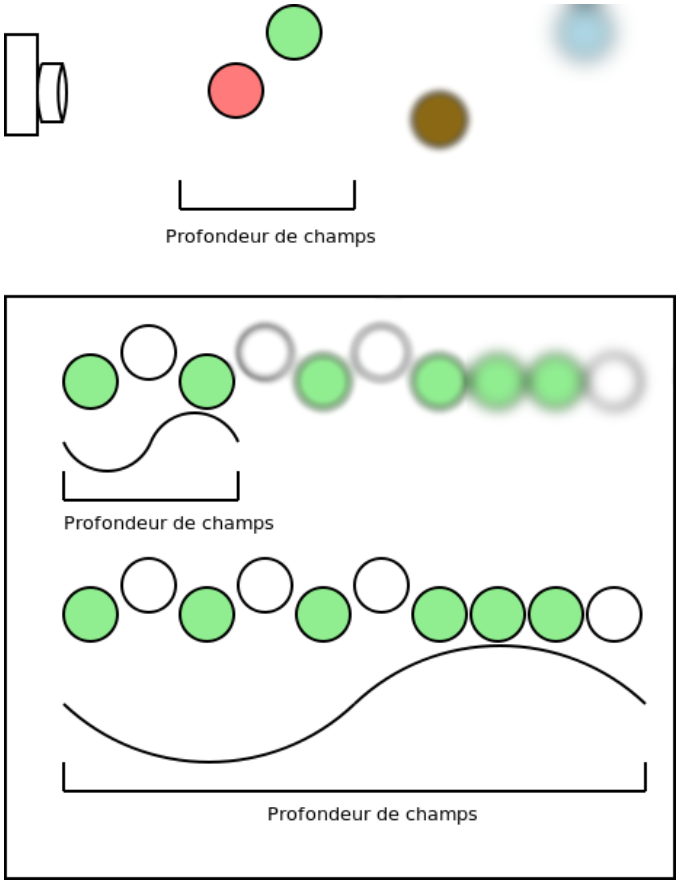

Figure 4.13 – Paralléle entre le concept le profondeur de champs et l'influence de λ. Plus la valeur de λ est élevée et plus le modèle est capable d'appréciation précise loin dans le temps.

Diddley

Contrairement aux autres fichiers d'entraînement, Diddley provient des exemples fournis avec la boite à rythme Hydrogen [2]. Il s'agit d'un rock où les roulements s'enchaînent avant de laisser la place à un rythme complexe. En conséquence, ce fichier constitue un bon test pour les capacités de transition de notre modèle. Comme on peut le voir en écoutant le fichier de génération le modèle s'en sort très bien. Les roulements et les rythmes sont cohérents de même que leurs positions dans le morceau généré.

Samba

Ce morceau est composé d'une rythmique constante pendant la grande partie du morceau auquelle viennent s'ajouter une clave et un tome. Le morceau contient également une caisse claire mais qui n'intervient que dans le roulement de fin, en compagnie de tomes. Dans le premier exemple que nous présentons "Samba", le modèle est capable, comme dans le fichier d'entraînement, de faire entrer la clave puis le tome, et bien que les partitions de ces instruments dans le fichier généré sont plus simples que celles dans le fichier d'entraînement, elles possèdent néanmoins de fortes similitudes avec ces dernières. Le modèle arrive également à très bien insérer le roulement. Le deuxième exemple "Samba-Claire" est selon nous plus intéressant puisque non seulement les variations de la clave et du tome sont plus complexes, mais le modèle est également arrivé à insérer la caisse claire d'une façon tout à fait cohérente avec le style musicale, mais en dehors du contexte de roulement.

4.3.8 Passage d'une séquence à une autre

Lorsqu'un utilisateur final utilise notre système, il se place dans un paradigme différent de la composition au sens classique du terme puisqu'il perd le contrôle sur "quelle note est jouée quand". Nous comprenons que cette perte de contrôle puisse rebuter certains et c'est pourquoi nous avons imaginé un système différent où l'on entraîne le modèle à passer d'un rythme à un autre. On octroie ainsi à l'utilisateur la possibilité de faire passer le modèle d'un rythme à un autre en temps réel pendant la génération, en modifi-

ant les entrées du réseau. Il se retrouve ainsi dans une position similaire à celle d'un chef d'orchestre qui indique aux musiciens quand est ce qu'ils doivent jouer. Cette capacité de transition n'est d'ailleurs pas seulement intéressante dans notre cas seulement puisqu'il arrive qu'avec n'importe quel type de séquence temporelle complexe, on ait besoin de passer rapidement d'un état à un autre. Pour permettre au modèle de transition, nous avons utilisé un modèle similaire à celui que Sussilo et Abbot ont utilisé pour obtenir un modèle génératif capable de transiter d'une séquence de marche à une séquence de course [60] : nous avons ajouté une entrée par séquence dont la valeur est soit 0 soit 1. En choisissant laquelle de ces entrées mettre à 1, l'utilisateur indique au modèle quel type de séquences générer. Il est important de noter que ces nouvelles entrées sont connectées parcimonieusement au réservoir, ce qui implique que chacune d'entre elles excite directement un ensemble différent de neurones et par conséquent, provoque dans le réservoir une influence qui lui est propre. La figure 4.14 montre le modèle dans son intégralité. Comme on peut le voir en écoutant le fichier "Transitions.mid" le modèle arrive en effet à passer efficacement d'un rythme à un autre, tout en conservant la cohérence de la génération. Cette capacité du modèle à s'adapter en temps réel ouvre également la voie à de fascinantes applications en intelligence artificielle dont nous discutons dans la conclusion.

4.3.9 Autres applications du modèle

Le sur-apprentissage de séquences

Notre modèle est suffisamment puissant pour pouvoir apprendre à reproduire à l'identique une séquence de grande longueur sans pour autant posséder un τ égale à la longueur de la séquence. Par exemple, dans le fichier "BossaOverfit" le modèle a appris à reproduire à l'identique une séquence de longueur 1008 avec un $\tau = 128$. Il est plus facile d'obtenir ce phénomène si la longueur de la cible est un multiple de τ.

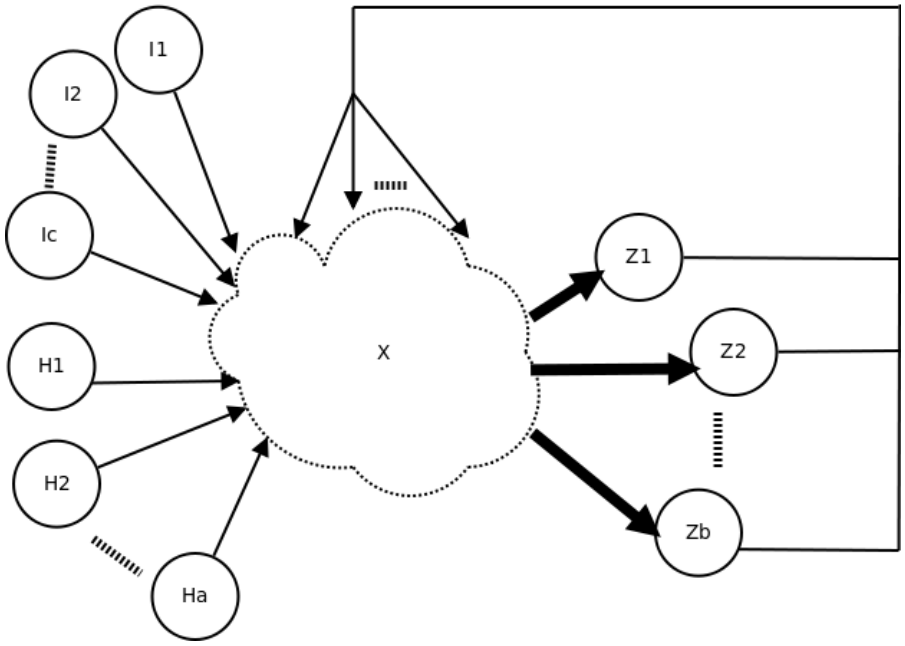

Figure 4.14 – Modèle génératif à transition. Nous ajoutons au modèle un certain nombre d'entrées parcimonieusement connéctées, qui une fois placées à 1 indique au modèle quel type de séquence générer.

L'extraction d'informations

Le modèle que nous avons proposé peut également servir à extraire des informations pertinentes d'une séquence. Dans "2Rock-Extraction" où la production du modèle n'est pas une séquence complexe de suites rythmiques mais un rythme de rock "noyau" basique. Il est plus facile d'obtenir de tels résultats si à la fois τ et λ ont la même valeur que la longueur de la séquence noyau que l'on désire apprendre.

4.4 Conclusion et discussion

Dans ce chapitre, nous avons présenté un nouveau modèle génératif, capable d'apprendre à générer des séquences temporelles complexes qui demandent de pouvoir traiter de l'information à différentes fréquences. La particularité de notre modèle est d'utiliser à la fois un ensemble d'apprenants Double-Orbite (voir sec. 4.2.3) (un apprenant par instrument) et d'avoir en plus du retour d'informations une entrée qui lui est fournie

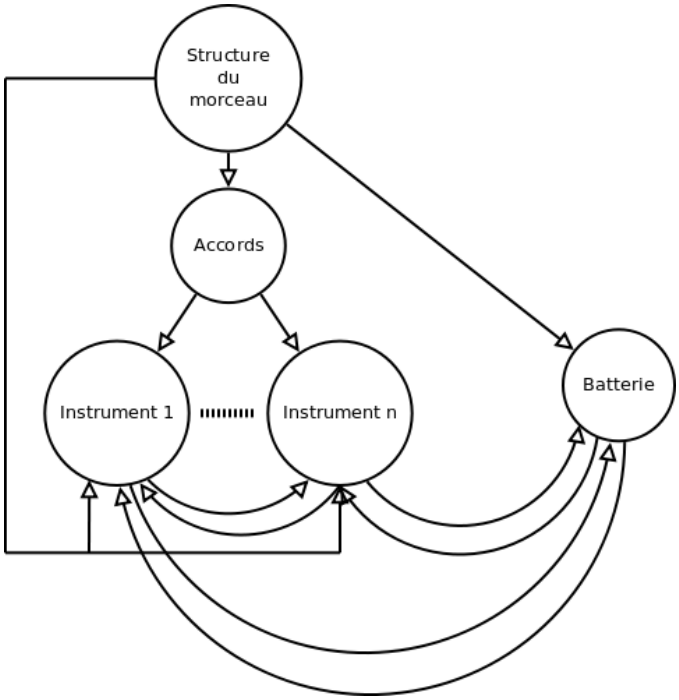

Figure 4.15 – Architecture d'un orchestre virtuel. Les différents générateurs d'instruments communiquent directement.

par une Horloge dont le rôle est d'induire dans le réservoir une activité périodique à plusieurs échelles. Grâce à la combinaison de ces deux éléments, le modèle est capable de produire, avec des réservoirs de seulement une centaine de neurones des séquences complexes cohérentes et stables, et ce indépendamment de la longueur de la cible ou de la génération. La combinaison de toutes ces caractéristiques constitue à notre connaissance une première dans le domaine.

Ce modèle peut également apprendre à passer d'une séquence à une autre sur demande. Ce point est selon nous de première importance puisqu'il ouvre la porte à de fascinantes applications furures du modèle en intelligence artificiel. En effet, si l'on remplaçait les entrées fournies consciemment par l'utilisateur lors de la génération, par des données pertinentes extraites directement du morceau, il serait peut être possible d'obtenir un véritable batteur virtuel. Ce batteur serait alors capable de créer de lui même des partitions de batteries qui conviennent au morceau, voire d'improviser en temps réel en s'adaptant au jeu des autres musiciens. En poussant les choses davantage, on pourrait

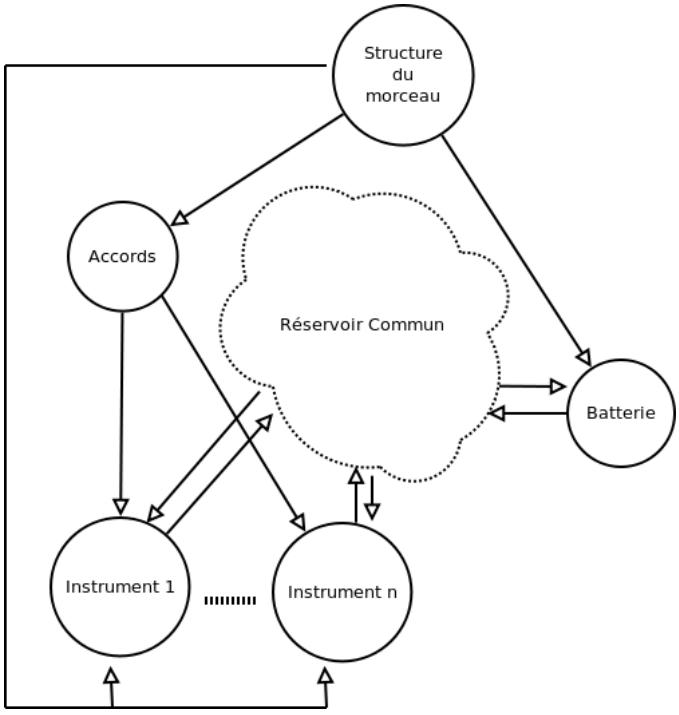

Figure 4.16 – Architecture d'un orchestre virtuel. Les différents générateurs d'instruments communiquent au travers d'un reservoir dont le rôle est de conserver un historique du morceau.

même imaginer un orchestre entièrement virtuel capable d'improviser ou de composer sur demande, puisqu'à priori, il suffirait d'ajouter la gestion de la durée des notes pour transformer notre batteur en joueur de n'importe quel instrument [5]. Les figures 4.15 et 4.16 montrent de possibles architectures que pourrait avoir cet orchestre virtuel. Dans les deux cas, l'orchestre est composé de plusieurs générateurs possédant chacun ses propres valeurs d'hyper-paramètres, son propre réservoir et sa propre horloge. La structure commune aux deux modèles est la suivante : A la tête de l'architecture, il y a un générateur dont le rôle est de générer la structure des morceaux. Cette unité fournit l'information de base nécessaire au générateur de suite d'accords et à la batterie, et influence également les générations produites par les instruments, ce qui leur permettra d'ajuster la mélodie à la structure. Le générateur d'accords dont le rôle est de produire la structure harmonique du morceau, fournit l'accord courant aux générateurs d'instruments, qui eux produisent la partie mélodique. Tous les instruments communiquent entre eux échangeant de l'information en temps réel, s'écoutant mutuellement comme le feraient de véritables musiciens. La seule différence entre les deux modèles se situe d'ailleurs au niveau de la façon avec laquelle s'effectue l'échange d'informations entre les "musiciens". Dans le premier modèle, ils communiquent directement alors que dans le second les informations sont envoyées dans un réservoir commun. La première approche est sans doute la plus facile à implémenter, et il revient à chaque générateur d'instruments de conserver une représentation de l'historique du morceau dans son réservoir. La seconde quant à elle, a l'avantage de posséder un reservoir auxiliaire qui sert d'une seconde mémoire dynamique et qui permettrait à priori aux générateurs de séquences mélodiques et de batteries, de prendre leurs décisions en fonction d'une représentation plus complexe de l'historique du morceau.

Une fois la génération obtenue, il sera alors possible de l'utiliser comme entrée pour un modèle d'interprétation automatique [33] afin d'obtenir un morceau aux sonorités plus humaines et moins mécaniques.

Néanmoins, notre modèle manque encore de capacité d'adaptation pour pouvoir être utilisé efficacement dans de telles conditions. En effet, le choix des hyper- paramètres

5. En midi chaque instrument de percussion correspond en fait à une note.

peut se révéler ardu. Comme le montre la figure 4.10, la plupart des stimulations induites par l'Horloge dans le réservoir sont de même fréquence, bien que ceci aide sans doute à la stabilité du modèle. Induire plus de fréquences dans le réservoir fournirait aux apprenants une information temporelle plus riche, et pourrait améliorer les performances du modèle. Malheureusement, aucune des méthodes que nous avons essayé jusque là n'a été en mesure de modifier significativement cette particularité du comportement du réservoir. Nous pensons néanmoins que remplacer les neurones du réservoir par des filtres passe-band de différentes fréquences pourrait apporter une amélioration significative [55].

L'autre amélioration qui pourrait avoir son importance est l'adaptation de τ. En effet tout les apprenants Double-Orbite utilisés pour apprendre à partir d'un morceau, possèdent les mêmes valeurs d'hyper-paramètres. Or ceci sous entend de posséder des connaissances à prioris sur la nature de la cible, de plus les instruments percussifs d'un morceau sont rarement joués à la même fréquence, par conséquent, le modèle bénéficierait énormément d'une méthode permettant d'adapter automatiquement la valeur de τ à la séquence entrain d'être apprise.

Bien que notre modèle ait été créé pour la génération musicale, son domaine d'application dépasse sans doute le cadre la musique, et il pourrait très bien servir à générer et analyser d'autres types de séquences périodiques.

CHAPITRE 5

PYRESERVOIR ET EXPYUTILS

Ce chapitre est principalement consacré à PyReservoir [15], un cadriciel en python spécialisé dans les réseaux de neurones à réservoir que nous avons développé dans le cadre de cette maîtrise. Ce cadriciel qui est avant tout très simple d'utilisation, est également très simple à étendre de sorte que l'utilisateur peut très facilement y ajouter de nouvelles fonctionnalités et ainsi le plier à ses besoins. Nous commencerons ce chapitre par quelques exemples qui attestent de la simplicité d'utilisation de PyReservoir. Nous verrons ensuite comment y ajouter de nouvelles fonctionnalités. La dernière partie de ce chapitre est quant à elle consacrée à ExPyUtils [14] qui est une suite d'outils en python regroupant des fonctionnalités courantes en expérimentation.

5.1 PyReservoir

PyReservoir est un module python composé de deux sous modules :

- reservoirComputing qui contient tous les réservoirs et tous les apprenants.
- rhythm qui contient les classes pour lire et écrire des fichiers Midi, générer les horloges et des cibles aléatoires.

5.2 Dépendances

Les dépendances obligatoires de PyReservoir sont les suivantes : Theano [7]pour l'optimisation pour des parties critiques et numpy [4] pour les calculs numériques. A ces dépendances s'ajoute une facultative qui est Pygmy [5] pour la gestion de tout ce qui est en rapport avec le MIDI.

5.3 Exemple d'utilisation de PyReservoir

5.3.1 Création d'un lecteur et d'un écrivain Midi

```
from rhythm.MidiTrackDispatcher import MidiTrackDispatcher
from rhythm.MidiWriter import MidiWriter

    #Crée un lecteur Midi pour lire la première piste 1 du fichier 'monMidi.mid'.
    #True indique l'on désire que les entrées ne soient lues qu'à partir d'une seule piste,
    #16 indique que l'on désire que la piste soit lue tout les 16émes de temps.

    lecteurMidi = MidiTrackDispatcher('monMidi.mid', 1, True, 16)

    #Crée un écrivain Midi pour autant d'instruments qu'il y en a dans monMidi.mid.100 est la
    #vélocité par défaut des coups et 16 indique que les entrées qui seront envoyées à l'écrivain
    #correspondent à des 16émes de temps.

    ecrivainMidi = MidiWriter(lecteurMidi.getPitches(), 100, 16)
```

5.3.2 Création d'un réservoir

```
from reservoirComputing import *

    #Crée un réservoir de 200 neurones avec 4 entrées et qui reçoit
    #un retour d'information de cardinalité 3.

    reservoir = ESN.ESN(4, 200, 3)
    reservoir.setParam("insideConnectivityRatio", 0.01)
    reservoir.setParam("leakyA", 0.1)

    #construit le réservoir
    reservoir.make()
```

5.3.3 Création d'un apprenant de séquences

```
from reservoirComputing import *

    #Crée un apprenant de séquences, contenant 3 apprenants Double-Orbite.
    #True signifie que l'on désire qu'il soit Double-Orbite et non Orbite.

    seqLearner = MultiSequenceLearner.MultiSequenceLearner(3, reservoir, True)

    #défini les paramètres des apprenants
    seqLearner.setLearnersParam("alpha", 1)
    seqLearner.setLearnersParam("dTau", 16)
    seqLearner.setLearnersParam("connectivityRatio", 0.1)

    #défini les paramètres des corrections
    seqLearner.setCorrectionsParam("alpha", params["Alpha"])
    seqLearner.setCorrectionsParam("dTau", params["dTau"])
    seqLearner.setCorrectionsParam("connectivityRatio", 0.1)
    #construit l'apprenant

    seqLearner.make()
```

5.3.4 Création d'une horloge

```
from rhythm.RandomInputDispatcher import RandomInputDispatcher

    #Crée une horloge de cardinalité 4, de longueur d'onde 512 et complexité 2.//
    horloge = RandomInputDispatcher(4)
    horloge.createComplexSines(512, 2)
```

5.3.5 Apprentissage

```
#Apprend sur 100 époques et affiche la sortie retournée par l'apprenant

for i in xrange(100) :
    for j in xrange(len(lecteurMidi)) :
        #L'apprenant transmet l'entrée au réservoir
        #et lui fournit un retour d'informations qui lui convient.
        #Le tout est transparent à l'utilisateur.
        seqLearner.learnOnline(horloge.nextLooped(), lecteurMidi.nextLooped())
```

5.3.6 Génération

```
#Génère sur 100 époques et affiche la sortie retournées par l'apprenant
for i in xrange(100) :
    for j in xrange(len(lecteurMidi)) :
        print seqLearner.genOnline(horloge.nextLooped())
```

5.3.7 Étendre les fonctionnalités de PyReservoir

Au début de notre recherche, nous avons expérimenté avec différents types de réservoirs et d'apprenants, nous n'avions encore aucune idée de quel réservoir nous allions utiliser et avec quels apprenants. Pour nous éviter de devoir copier/coller ou réécrire du code à chaque changement d'architecture, et pour simplifier la maintenance, nous avons conçu PyReservoir pour être le plus facile à étendre possible. La figure 5.1 montre l'architecture d'héritage de PyReservoir, il y a trois classes abstraites :

- **ReservoirComputingObject** qui est la mère de toutes les classes du module, elle contient les fonctions de base de gestion de paramètres et d'affichages du contenu de l'objet en chaîne de caractères. Grâce aux fonctions de cette classe, il suffit par exemple de faire : "print object", pour obtenir un descriptif détaillé de tous les paramètres de l'objet et de leurs valeurs, comme le montre la figure 5.2.
- **Reservoir** qui est la classe dont héritent tous les réservoirs.

– **Learner** qui est la classe dont héritent les apprenants.

5.3.8 Ajout d'un nouveau type de réservoir

L'exemple le plus simple pour la création d'un nouveau réservoir est la classe ESN :

```
from Reservoir import Reservoir
#ESN hérite de réservoir
class ESN(Reservoir) :

    def __init__(self, inSize, resSize, fbSize = 0, *args, **kwargs) :
        #On appel le constructeur de la classe mère

        Reservoir.__init__(self, inSize, resSize, fbSize, *args, **kwargs)
        self._name = "Echo State Network"
        #On ajoute les paramètres propres à la classe ESN.
        #L'utilisateur pour changer leur valeur grâce à la fonction "setParam"

        self._params["useHeuristics"] = True
        self._params["pWScaling"] = 0.9
```

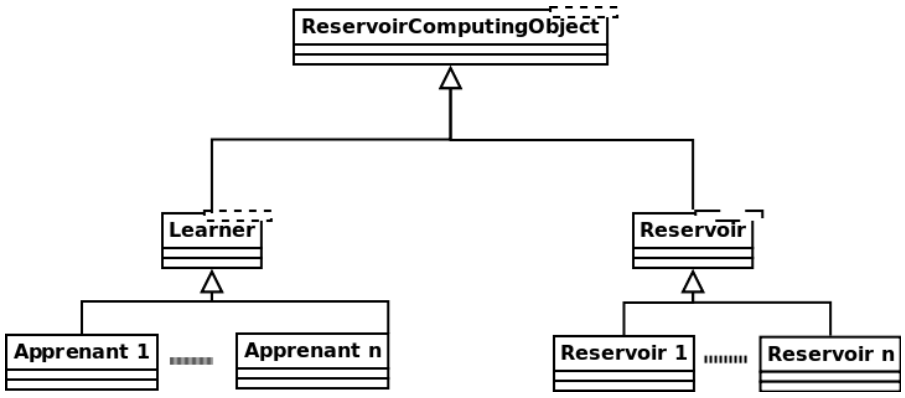

Figure 5.1 – Architecture de PyReservoir.

```
Name : Orbit Learner with RLS
First name :
Params :
------
        reservoir : Echo State Network
        connectivityRatio : 0.100000
        dTau : 40.000000
        stopCritrerionThreshold : 0.000001
        useStopCriterion : 0.000000
        alpha : 1.000000
Infos :
------
        additionalInfos :


====
Name : Echo State Network
First name :
Params :
------
        nbWashouts : 200.000000
        magnWFb : 0.900000
        noisePower : 0.020000
        insideConnectivityRatio : 0.010000
        magnWRes : 1.468332
        onePolarity : 0.000000
        spreadWRes : 0.000000
        fbSize : 12.000000
        spreadWInp : 0.000000
        spreadWFb : 0.000000
        resSize : 196.000000
        bias : 0.000000
        inputConnectivityRatio : 0.200000
        fbConnectivityRatio : 0.200000
        pWScaling : 0.900000
        useHeuristics : 1.000000
        inSize : 4.000000
        magnWInp : 0.700000
        leakyA : 0.100000
        unsupervisedLR : 0.000000
Infos :
------
        maxResW : 0.099945
        additionalInfos :
        meanResW : 0.051123
        resPw : 0.755285
        minResW : 0.000049

         --> Echo State Network : p(W) =  0.755285324453
```

Figure 5.2 – Exemple de sortie de l'impression du réservoir.

```
    #on ajoute la fonction pour réduire le rayon spectrale de la
    #matrice de poids du réservoir
    def useHeuristics(self) :
        if (self.getResPw() >= 1) and self._params["useHeuristics"] :
            self.tReservoir.innW.value = _
            self.tReservoir.innW.value * 1/(self.getResPw()*self._params["pWScaling"])
        self._infos["resPw"] = self.getResPw()

    #On réecri la fonction make pour qu'elle réduise
    #également le rayon spectrale de la matrice de poids
    def make(self) :
        Reservoir.make(self)
        self.useHeuristics()
```

5.3.9 Ajout d'un nouvel apprenant

Voici un exemple vide d'apprenant :

```
from Learner import Learner

class Exemple(Learner) :
    def __init__(self, nbDirectInputs, reservoir, nbOutputs) :
        #On appelle le constructeur de la classe mère
        Learner.__init__(self, nbDirectInputs, reservoir, nbOutputs)

        #On ajoute les paramètres
        self._params["param1"] = 10
        self._params["param2"] = 0.01

        self._name = "Exemple d'apprenant"
```

```
def make(self) :
    #On appelle l la fonction make de la classe mère
    #S'il y a des choses à faire avant la construction
    #de l'apprenant c'est ici qu'il faut les mettre.
    Learner.make(self)

def _learn(self, targets, directInputs = None) :
    #L'algorithme d'apprentissage viens ici. Il est possible d'utiliser
    #self._inputs contient une concaténation du réservoir et des entrées directes.

def _computeOutputs(self, directInputs = None) :
    #L'algorithme pour le calcul de la sortie de l'apprenant, ici aussi
    #on peut utiliser self._inputs
```

5.4 ExPyUtils

ExPyUtils est un module qui contient quatre fichiers :

- **SimulationPlots** qui simplifie la gestion de l'affichage de courbes, matrices et points. Elle permet également de faire des vidéos de l'évolution des graphiques
- **StatTimer** qui est un compteur de temps qui affiche le temps pris pour chaque passe, le temps moyen d'une passe et une estimation du temps restant
- **SystemArguments** qui simplifie la gestion du passage d'arguments systèmes au programme
- **usefulfuncs** qui est un ensemble de fonctions utiles, comme par exemple pour la sauvegarde de fichiers ou de matrices.

5.4.1 Dépendances obligatoires

ExPyUtils à deux dépendances obligatoires mumpy [4] et Matplotlib [3] pour l'affichage. ExPyUtils a également besoin de ffmpeg [1] mais seulement si l'on désire générer des vidéos.

5.4.2 Exemples d'affichages

5.4.2.1 Graphique

```
from expyutils import SimulationPlots as sp

import pylab as P
importnumpy as N

a = N.linspace(0,10, 100)
#Creer une figure
f = sp.Figure()

#Ajoute un graphe. Le décalage de 0.5 signifie que chaque nouvelle
#courbe doit être placée 0.5 unitées au dessus de la précédente
subPlot = f.createSubplot('abscices', 'ordonnees', offset = 0.5)

#ajoute 3 sinusoïdes au graphique. Si Gradient est à True alors
#les couleurs des courbes seront issues d'un dégradé, la couleur de
#la première étant bleu pur et celle de la dernière rouge pur.
#Il est possible d'ajouter des courbes à partir d'un fichier
#à l'aide de la fonction addCurveToLoad().

subPlot.addCurve(N.sin(a), '-', 'sin', Gradient = True)
subPlot.addCurve(N.sin(a), 'o', 'sin (decalage = 0.5)', Gradient = True)
subPlot.addCurve(N.sin(a), '+', 'sin (decalage = 1)', Gradient = True)
```

```
#ajoute une courbe verte en pointillés
subPlot.addCurve(N.sin(a), '+g.-', 'sin (decalage = 1.5 verte)')

#On peut créer autant de graphiques que nécessaire.
subPlot2 = f.createSubplot('abscices', 'ordonnees')
subPlot2.addCurve(N.sin(a), '+r.-', 'sin rouge')

#Desine le graphique.
f.plot()

#En appelant cette fonction on enregistre une image du dernier état du graphique.
f.saveImage('./testPlot.png')

#finalement, il est nécessaire d'appeler la fonction show de matplotlib
P.show()
```

Selon les mêmes principes, il est également possible d'ajouter aux graphiques des matrices, des *"scatter plots"* et des points. La figure 5.3 montre le graphique généré à l'aide de ce code.

5.4.2.2 Vidéo

```
#pour faire une vidéo le principe est exactement le même, il suffit simplement d'appeler
#la fonction addState() à chaque modification du graphique.
for i in xrange(400) :
    r = N.random.rand(10)
    f.createSubplot('animation', '').addCurve(r, "r", "")
    f.addState()
```

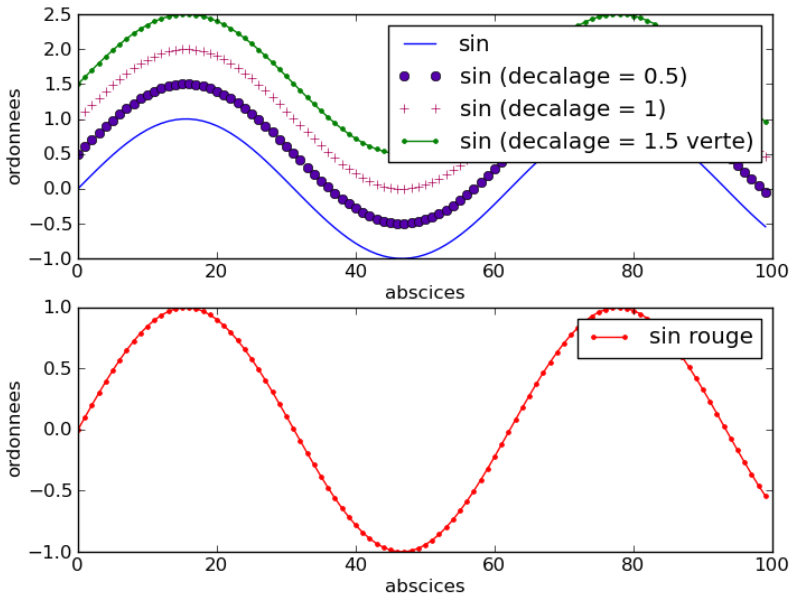

Figure 5.3 – Exemple de graphique obtenu avec ExPyUtils.

#Et finalement, d'appeler cette fonction pour générer la vidéo. './m' est le dossier
#dans lequel sera placée la vidéo de même que toutes les images qui la composent.
#Si './m' existe déjà un dossier './m_<date>' ou date une chaîne de caractères
#contenant la date et l'heure courante sera créée.
f.statesToMpg('./m')

5.4.2.3 Compteur de temps

from expyutils import StatTimer as st

#pour cet exemple, nous reprenons l'exemple d'apprentissage 5.3.5

nbEpoques = 100
#On initialise un compteur qui affiche les résultats au moins toutes les 20 secondes
compteur= st.StatTimer(20)

```
#On démarre le compteur et on lui donne le nombre total d'époques d'apprentissage.
compteur.startExperimentationTimer()
compteur.setStartPoint(nbEpoques)

for i in xrange(100) :
    for j in xrange(len(lecteurMidi)) :
        seqLearner.learnOnline(horloge.nextLooped(), lecteurMidi.nextLooped())

        #Affiche les statistiques si au moins 20 secondes se sont écoulées
        compteur.showStats() (1)

#L'expérimentation est terminée, on affiche un compte rendu.
compteur.showExperimentationStats() (2)
```

La ligne **(1)** affichera par exemple :

```
===
    pass : 3/10, left : 7
    ElapsedTime : 0h 1m 39s
    Time Left : 0h 3m 51s
    Average pass time : 0h 0m 33s
```

et la ligne **(2)** :

```
=-=-=-=-
    Expermimentation started at : 356035h 54m 54s Ended at : 356036h 0m 32s
    Took : 0h 5m 37s
=-=-=-=-
```

5.4.2.4 Gestion des arguments systèmes

Pour utiliser la classe **SystemArguments**, il faut tout d'abord, créer un dictionnaire qui possède le nom des arguments comme clefs et leurs valeurs par défaut comme valeurs. La classe se charge ensuite de définir les noms des arguments à passer au programme et de vérifier que les types de ces arguments correspondent aux types par défaut.

Les clefs peuvent êtres entièrement en minuscules ou contenir des majuscules. Dans le cas où la clef est entièrement minuscule comme par exemple "epoques", la classe essayera de prendre la première lettre pour le nom court de l'argument passé au programme qui sera ici : "-e". Si cette lettre est prise elle essayera les suivantes et si aucune d'entres elles n'est libre, une lettre libre de l'alphabet sera choisie.

Si l'on désire qu'une autre lettre que la première soit choisie pour le nom de l'argument, il suffit de mettre cette lettre en majuscule. Par exemple si la clef du dictionnaire est "taille-Reservoir", le nom court de l'argument à passer au programme sera "-R". Dans tous les cas, le nom long de l'argument est "–clef" soit ici : "–époques" et "–taille-Reservoir".

Voici un exemple de code :

```
from expyutils import SystemArguments as sa

params = {}
#On définit les arguments et leurs paramètres par défaut.
params["epoques"] = 10
params["taille-Reservoir"] = 100
params["bool"] = True
params["flotant"] = 1.2
params["chaine"] = "aaaa"

#On passe le dictionnaire au constructeur de SystemArguments
gestionnaireParam = sa.SystemArguments(params)
#Il est possible d'obtenir l'usage grâce à
```

print gestionnaireParam.usage() **(1)**

#On récupère les valeurs des paramètres passés au programme grâce à
params = gestionnaireParam.getArguments()
#params contient maintenant les valeurs passées en argument au programme.

La ligne **(1)** affiche :

```
Usage :
—-
-e (–epoques) <int>
-R (–taille-Reservoir) <int>
-b (–bool) <boolt>
-f (–flotant) <float>
-c (–chaine) <string>
```

5.5 Conclusion

Dans ce chapître, nous avons présenté les outils que nous avons développés pour notre recherche. Il s'agit de PyReservoir [15], un cadriciel en python pour le RC, qui est à la fois simple d'utilisation et simple à étendre et de ExPyUtils [14], qui est un module python regroupant entre autre des fonctionnalités dont on a souvent besoin lors d'expérimentations à savoir : l'affichage de graphiques, de la gestion des arguments systèmes ou encore de l'affichage de statistiques, quant au déroulements de l'expérimentation. PyReservoir et ExPyUtils sont tout deux disponibles à l'adresse : https ://launchpad.net/ tariq-daouda.

CHAPITRE 6

CONCLUSION

Dans ce travail, nous nous sommes intéressés aux réseaux de neurones et en particulier aux réseaux de neurones à réservoirs et leur application à la génération et à la reconnaissance de rythmes. Après une présentation générale sur les réseaux de neurones, nous avons introduit deux modèles l'un pour la reconnaissance l'autre pour la génération, ainsi que les outils que nous avons développé pour nos recherches.

6.1 Reconnaissance

Pour la reconnaissance de rythmes, nous avons développé un modèle dont l'architecture comporte trois parties distinctes, un réservoir et deux niveaux d'apprentissage, nous a permis d'obtenir d'excellents résultats sur 54 des 57 hymnes nationaux de l'ensemble [54]. Des travaux futurs pourraient avoir comme but d'invéstiguer la raison de ces trois erreurs du modèle, que nous pensons dues à un manque de capacité. Nous proposons d'ailleurs dans la conclusion du chapitre sur la reconnaissance de rythmes une solution pour résoudre ce problème qui consiste à remplacer les apprenants de la première couche par des auto-encodeurs [65].

Nous avons également testé la résistance au bruit de notre modèle en altérant les séquences :

- En remplaçant des silences par des notes
- En remplaçant des notes par des silences
- En appliquant les deux types de transformations en même temps.

Nous avons ainsi démontré que notre modèle possède une certaine résistance au bruit, notamment dans le cas du premier type de transformation où la résistance du modèle est quasiment de $1 - a$ (a étant le taux d'altération). Ainsi le modèle a été capable de reconnaître 20% des séquences lorsqu'elles étaient altérées à 80%. Les résultats sur le second type d'altérations ont été moins bons. Ce qui est intéressant puisque les notes étant plus

rares que les silences dans notre ensemble de données, une séquence altérée seulement par le premier type de transformation présente quantitativement plus de différences par rapport à l'ensemble d'entraînement que si on lui avait appliqué seulement la seconde. Cependant, cette même raison implique que les notes véhiculent plus d'informations que les silences, les supprimer prive ainsi le modèle d'informations discriminatrices essentielles. Comme on pouvait s'y attendre, c'est lorsqu'on applique les deux transformations en même temps que les performances sont les moins bonnes. Cette façon de faire transforme en effet radicalement la séquence d'une façon qui la rend rapidement méconnaissable même pour un auditeur averti.

6.2 Génération

Le deuxième modèle que nous proposons est un modèle pour l'apprentissage et la génération de séquences périodiques. Ce modèle présente deux différences par rapport au modèle génératif classiquement utilisé avec les ESN [29] puisqu'il possède à la fois une horloge et une couche d'apprenant Double-Orbite (voir sec. 4.2.3). Le rôle de l'Horloge, dont la sortie est constituée de plusieurs fonctions construites à base de sinusoides, est d'induire dans le reservoir une activité périodique à plusieurs échelles, le modèle possède ainsi en tout temps une information temporelle périodique et stable. La deuxième particularité de notre modèle est d'utiliser un apprenant Double-Orbite par instrument. Cet algorithme que nous avons spécialement développé pour cette tâche permet d'augmenter la capacité du modèle sans toucher à la taille du réservoir et de répartir cette capacité dans le temps. Il permet ainsi d'obtenir de très bon résultats de même qu'une convergence trés rapide avec des réservoirs de seulement une centaine de neurones. En testant notre modèle sur de veritables exercices de batterie [6], nous avons été en mesure de montrer qu'il est capable de générer des séquences d'une stabilité parfaite et ce, indépendamment de la longueur de la génération ou de celle de la séquence d'entrainement. Notre modèle est non seulement capable de générer avec une trés grande précision des séquences au sein desquelles cohéxistent des événements de périodes différentes, mais est également capable d'apprendre à transiter d'un rythme à un autre et d'introduire des roulements dans un

morceau tout en conservant la cohérence de celui-ci. L'ensemble de ces caractéristiques constituent à notre connaissance une première dans le domaine des réseaux de neurones à réservoir. Nous avons également mis en évidence d'autres applications possibles de notre modèle comme le sur-apprentissage de séquences, l'extraction d'informations ou la composition et montré comment notre modèle pourrait servir de base à l'élaboration d'un orchestre virtuel.

6.3 Outils : PyReservoir et ExPyUtils

Dans cette partie, nous avons présenté les deux outils que nous avons développé pour nous aider dans nos recherches :

- PyReservoir [15] qui est un cadriciel python pour les réseaux de neurones a réservoir, il est à la fois trés simple à utiliser et trés simple à étendre.
- ExPyUtils [14] qui est une suite d'outils regroupant des fonctionnalités dont on à souvent besoin lors d'expérimentations.

BIBLIOGRAPHIE

[1] ffmpeg website. URL http://www.ffmpeg.org/.

[2] hydrogen-music.org. URL http://www.hydrogen-music.org/.

[3] matplotlib website. URL http://matplotlib.sourceforge.net/.

[4] numpy website. URL http://numpy.scipy.org/.

[5] pygmy website. URL http://bitbucket.org/douglaseck/pygmy/.

[6] rhythmpatterns.com. URL http://www.rhythmpatterns.com/exercises/.

[7] Theano website. URL http://deeplearning.net/software/theano/.

[8] Y. Bengio, P. Simard et P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2), 1994.

[9] N. Bertschinger et T. Natschläger. Real-time computation at the edge of chaos in recurrent neural networks. *Neural Computation*, 16(7):1413–1436, 2004.

[10] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. 2006.

[11] H. Burgsteiner, M. Kröl, A. Leopold et G. Steinbauer. Movement prediction from real-world images using a liquid state machine. *Applied Intelligence*, 26(2):99–109, 2007.

[12] N.J. Butko et J. Triesch. Learning sensory representations with intrinsic plasticity. *Neurocomputing*, 70(7-9):1130–1138, 2007.

[13] R. H. Cudmore et N. S. Desai. Intrinsic plasticity. *Scholarpedia*, 3(2):1363, 2008.

[14] Tariq Daouda. Expyutils in launchpad, 2010. URL https://launchpad.net/expyutils.

[15] Tariq Daouda. Pyreservoir in launchpad, 2010. URL `https://launchpad.net/pyreservoir`.

[16] Peter Dayan et L. F. Abbott. *Theorical neuroscience*. MIT Press, 2001.

[17] P.F. Dominey. Complex sensory-motor sequence learning based on recurrent state representation and reinforcement learning. *Biological Cybernetics*, 73(3):265–274, 1995.

[18] D. Eck et J. Schmidhuber. Finding temporal structure in music : Blues improvisation with LSTM recurrent networks. Dans *Neural Networks for Signal Processing XII, Proc. 2002 IEEE Workshop*, pages 747–756. Citeseer, 2002.

[19] Dario Floreano et Claudio Mattiussi. *Bio-Inspired Artificial Intelligence*. MIT Press, 2008.

[20] F.A. Gers, D. Eck et J. Schmidhuber. Applying LSTM to time series predictable through time-window approaches. *Lecture Notes in Computer Science*, pages 669–676, 2001.

[21] F.A. Gers, J. Schmidhuber et F. Cummins. Learning to forget : Continual prediction with LSTM. *Neural Computation*, 12(10):2451–2471, 2000.

[22] Simon Haykin. *Neural Networks and Learning Machines*. Third edition édition, 2009.

[23] G. E. Hinton. Deep belief networks. *Scholarpedia*, 4(5):5947, 2009.

[24] S. Hochreiter, Y. Bengio, P. Frasconi et J. Schmidhuber. Gradient flow in recurrent nets : the difficulty of learning long-term dependencies. 2001.

[25] S. Hochreiter et J. Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[26] H. Jaeger. Reservoir riddles : Suggestions for echo state network research. Dans *2005 IEEE International Joint Conference on Neural Networks, 2005. IJCNN'05. Proceedings*, pages 1460–1462.

[27] H. Jaeger. Short term memory in echo state networks. Rapport technique, Tech. Rep, 2002.

[28] H. Jaeger. Tutorial on training recurrent neural networks, covering BPPT, RTRL, EKF and the" echo state network" approach. *Fraunhofer Institute for Autonomous Intelligent Systems (AIS), International University Bremen*, 2002.

[29] H. Jaeger. Echo state network. *Scholarpedia*, 2(9):2330, 2007.

[30] H. Jaeger et D. Eck. Can't Get You Out of My Head : A Connectionist Model of Cyclic Rehearsal. *Lecture Notes in Computer Science*, 4930:310, 2008.

[31] H. Jaeger et H. Haas. Harnessing nonlinearity : Predicting chaotic systems and saving energy in wireless communication. *Science*, 304(5667):78, 2004.

[32] H. Jaeger, M. Lukosevicius, D. Popovici et U. Siewert. Optimization and applications of echo state networks with leaky-integrator neurons. *Neural Networks*, 20 (3):335–352, 2007.

[33] Stanislas Lauly. Modélisation de l'interprétation des pianistes et découverte d'applications d'auto-encodeurs sur les réseaux de neurones récurrents. Mémoire de maîtrise, Université de Montreal, 2009.

[34] A. Lazar, G. Pipa et J. Triesch. Fading memory and time series prediction in recurrent networks with different forms of plasticity. *Neural Networks*, 20(3):312–322, 2007.

[35] Y. LeCun, L. Bottou, G. Orr et K. Müller. Efficient backprop. *Neural networks : Tricks of the trade*, pages 546–546, 1998.

[36] R. Legenstein et W. Maass. Edge of chaos and prediction of computational performance for neural circuit models. *Neural Networks*, 20(3):323–334, 2007.

[37] M. Lukoševicius. Echo State Networks with Trained Feedbacks. 2007.

[38] M. Lukoševičius et H. Jaeger. Reservoir computing approaches to recurrent neural network training. *Computer Science Review*, 3(3):127–149, 2009.

[39] Wolfgang Maas. Liquid computing. *In Proceedings of the Conference CiE'07 : COMPUTABILITY IN EUROPE*, 2007.

[40] W. Maass, P. Joshi et E.D. Sontag. Principles of real-time computing with feedback applied to cortical microcircuit models. *Advances in neural information processing systems*, 18:835, 2006.

[41] W. Maass, T. Natschläger et H. Markram. Real-time computing without stable states : A new framework for neural computation based on perturbations. *Neural computation*, 14(11):2531–2560, 2002.

[42] C.K. Machens, R. Romo et C.D. Brody. Functional, But Not Anatomical, Separation of" What" and" When" in Prefrontal Cortex. *Journal of Neuroscience*, 30(1): 350, 2010.

[43] T. Natschlaeger, W. Maass et H. Markram. The "liquid computer", a novel strategy for real-time computing on time series. *Special issue on Foundations of Information Processing of TELEMATIK*, 8(1):39–43, 2002.

[44] Nils J. Nilsson. *The quest of artificial intelligence*. Cambridge University Press, 2010.

[45] O. Obst, X.R. Wang et M. Prokopenko. Using echo state networks for anomaly detection in underground coal mines. Dans *Proceedings of the 7th international conference on Information processing in sensor networks*, pages 219–229. IEEE Computer Society, 2008.

[46] H. Paugam-Moisy, R. Martinez et S. Bengio. Delay learning and polychronization for reservoir computing. *Neurocomputing*, 71(7-9):1143–1158, 2008.

[47] J.A. Pérez-Ortiz, F.A. Gers, D. Eck et J. Schmidhuber. Kalman filters improve LSTM network performance in problems unsolvable by traditional recurrent nets. *Neural Networks*, 16(2):241–250, 2003.

[48] D.J. Povel et P. Essens. Perception of temporal patterns. *Music Perception*, pages 411–440, 1985.

[49] Y.N. Rao, S.P. Kim, J.C. Sanchez, D. Erdogmus, J.C. Principe, J.M. Carmena, M.A. Lebedev et M.A. Nicolelis. Learning mappings in brain machine interfaces with echo state networks. Dans *IEEE International Conference on Acoustics, Speech, and Signal Processing, 2005. Proceedings.(ICASSP'05)*, 2005.

[50] Ali H. Sayed. *Fundamentals of Adaptive Filterimg*. 2003.

[51] J. Schmidhuber, F. Gers et D. Eck. Learning nonregular languages : A comparison of simple recurrent networks and LSTM. *Neural Computation*, 14(9):2039–2041, 2002.

[52] B. Schrauwen, L. Büsing et R. Legenstein. On computational power and the order-chaos phase transition in reservoir computing. Dans *Proceedings of NIPS*, volume 2009, 2008.

[53] B. Schrauwen, M. Wardermann, D. Verstraeten, J.J. Steil et D. Stroobandt. Improving reservoirs using intrinsic plasticity. *Neurocomputing*, 71(7-9):1159–1171, 2008.

[54] M. Shaw et H. Coleman. National Anthems of the World. 1960.

[55] U. Siewert et W. Wustlich. Echo-state networks with band-pass neurons : towards generic time-scale-independent reservoir structures. *Internal status report, PLANET intelligent systems GmbH*, 2007.

[56] H. Sompolinsky, A. Crisanti et HJ Sommers. Chaos in random neural networks. *Physical Review Letters*, 61(3):259–262, 1988.

[57] H. Soula, A. Alwan et G. Beslon. Learning at the edge of chaos : Temporal coupling of spiking neuron controller of autonomous robotic. Dans *AAAI Spring Symposium on Developmental Robotics. Stanford, CA, USA*, 2005.

[58] J.J. Steil. Backpropagation-Decorrelation : Online recurrent learning with O (N) complexity. Dans *Proceedings of the International Joint Conference on Neural Networks (IJCNN)*, volume 1, pages 843–848, 2004.

[59] J.J. Steil. Online reservoir adaptation by intrinsic plasticity for backpropagation-decorrelation and echo state learning. *Neural Networks*, 20(3):353–364, 2007.

[60] D. Sussillo et LF Abbott. Generating coherent patterns of activity from chaotic neural networks. *Neuron*, 63(4):544–557, 2009.

[61] D. Sussillo et LF Abbott. Generating coherent patterns of activity from chaotic neural networks, Source Code. *Neuron*, 63(4):544–557, 2009.

[62] J.P. Thivierge et P. Cisek. Nonperiodic synchronization in heterogeneous networks of spiking neurons. *Journal of Neuroscience*, 28(32):7968, 2008.

[63] D. Verstraeten, B. Schrauwen, M. D'Haene et D. Stroobandt. An experimental unification of reservoir computing methods. *Neural Networks*, 20(3):391–403, 2007.

[64] D. Verstraeten, B. Schrauwen, D. Stroobandt et J. Van Campenhout. Isolated word recognition with the liquid state machine : a case study. *Information Processing Letters*, 95(6):521–528, 2005.

[65] P. Vincent, H. Larochelle, Y. Bengio et P.A. Manzagol. Extracting and composing robust features with denoising autoencoders. Dans *Proceedings of the 25th international conference on Machine learning*, pages 1096–1103. ACM, 2008.

[66] C. Vreeswijk et H. Sompolinsky. Chaos in neuronal networks with balanced excitatory and inhibitory activity. *Science*, 274(5293):1724, 1996.

[67] Š. Babinec et J. Pospíchal. Improving the prediction accuracy of echo state neural networks by anti-Oja's learning. *Artificial Neural Networks–ICANN 2007*, pages 19–28, 2007.

[68] John Zorn et al. *Arcana*, volume II. Hips road, Tzadik, 2008.