Modèles probabilistes et vérification de réseaux de neurones

Cédric Girard Riboulleau

RAPPORT DE STAGE

MASTER 1 - BIOLOGIE, INFORMATIQUE ET MATHÉMATIQUES

Modèles probabilistes et vérification de réseaux de neurones

Auteur :
Cédric GIRARD RIBOULLEAU
Master : Science de la Vie et de la Santé
Spécialité : Génétique, Immunité et Développement

Encadrante :
Dr. Annie RESSOUCHE
INRIA,
2004 Route des Lucioles,
06902 Valbonne

Co-Encadrante :
Dr. Elisabetta DE MARIA
I3s, Euclide B,
Les Algorithmes,
2000 Route des Lucioles,
06900 Sophia Antipolis

Membre de UNIVERSITÉ CÔTE D'AZUR

Année 2016/2017

Abstract

Research advances in the field of neurobiology imply that neural networks are becoming larger and more complex. However, this complexity increases the computation time of the model simulations and therefore the speed and the memory used by softwares.

During this internship we choose to model neural networks as LI&F models (Leaky Integrate and Fire) represented by Markov chains with PRISM, a probabilistic model checker. With this software, we have the possibility to include probability in spike emission in our models according to a sigmoid curve. After having implemented several network models containing different numbers of neurons, we test several properties encoded in PCTL (Probabilistic Computation Tree Logic). We established the pseudo-code of a reduction algorithm which takes as input a network and a property and gives as output a reduced network. This algorithm removes the "wall" neurons that block the transmission of the membrane potential and those whose suppression does not affect the output neurons or the topology of the network. The reduced networks obtained have a significantly lower complexity.

Mots clés :
Réseaux neuronaux, Chaîne de Markov, PRISM, Model Checking, Probabilité.

Table des matières

1 Introduction

Depuis des années, les neurobiologistes et bio-informaticiens travaillent de concert pour modéliser des réseaux neuronaux afin de faire avancer la recherche dans les sciences du cerveau. Cependant, ces réseaux deviennent de plus en plus grands et complexes. Il faut donc trouver un moyen de les réduire sans pour autant les dénaturer. C'est avec l'équipe STARS de l'INRIA en collaboration avec le projet MS&N de l'équipe MDSC de l'I3S, que nous allons modéliser des réseaux neuronaux probabilistes, tester si ces modèles vérifient des propriétés données et enfin en déduire de possibles réductions de réseaux.

1.1 Les cellules nerveuses

Dans ce chapitre, nous décrivons les neurones ainsi que les fonctions que nous allons modéliser. Le neurone (ou cellule nerveuse) est l'unité de base du système nerveux [16], on en trouve entre 86 et 100 milliards dans le cerveau humain. Cette cellule a la capacité de recevoir des signaux électriques ou chimiques, de les stocker et de les envoyer vers d'autres neurones (ou cellules). Ces signaux sont stockés sous forme de *potentiel de membrane*. Ce potentiel (aussi appelé potentiel de repos) est dû à une différence de potentiel négative que l'on trouve entre les faces intracellulaire et extracellulaire du neurone. Cette différence de potentiel provient des différentes concentrations d'ions présentes à l'intérieur et à l'extérieur du neurone. Et pour se propager, ce signal utilise des courants ioniques ainsi que la perméabilité sélective de la membrane. Il faut noter aussi les courants de fuites vers le milieu extracellulaire qui entraîne une légère perte du signal. Pour recevoir et envoyer ces signaux, le neurone possède deux types de prolongement :

— L'axone (unique), qui se termine par des ramifications afin de transmettre le potentiel de membrane aux autres cellules, qui ne sont pas nécessairement d'autres neurones.

— Les dendrites (multiples) qui se ramifient énormément, jusqu'à former des épines dendritiques d'une taille de deux microns, pour recevoir les influx nerveux des autres neurones.

Sur l'extrémité de ces deux prolongements on retrouve les *synapses* où ont lieu les transferts de signaux, on note une moyenne de 10 000 synapses par neurone. Ces signaux sont matérialisés par le potentiel électrique des neurones. Ce potentiel peut être converti au niveau des synapses en neurotransmetteurs pour être envoyé chimiquement ou être directement transmis électriquement (rare pour les mammifères). Le signal transféré est appelé *potentiel d'action*, cet événement est caractérisé par une augmentation puis une diminution soudaine du potentiel électrique de la cellule, autrement dit, le potentiel de la membrane (visualisable par un pic sur un graphique). Ainsi cette différence de potentiel se répand le long de l'axone pour rejoindre les synapses. On appelle aussi cet événement un *spike*. Un neurone peut recevoir et envoyer un ou plusieurs signaux différents qui peuvent exciter ou inhiber les fonctions des autres cellules nerveuses.

Ici nous nous focalisons sur les interactions entre plusieurs neurones, c'est pour cela qu'on parle de réseaux neuronaux.

1.2 Les réseaux neuronaux

Dans la littérature, les réseaux neuronaux sont communément représentés comme des *graphes* où les noeuds des graphes montrent les neurones et les arcs représentent les connexions synaptiques. Ainsi le poids des arcs indique la fonction du neurone. A savoir, un poids positif pour une fonction excitatrice et un poids négatif pour une fonction inhibitrice.
Il y a eu plusieurs façons de modéliser les réseaux de neurones au cours du temps :

- 1^{ere} génération (1957)[18] : Dans ces réseaux, chaque neurone est vu comme une porte logique avec des sorties numériques. Cette génération rassemble les "portes seuils" de McCulloch et Pitt, les réseaux Hopfield, les machines Boltzmann et le perceptron.
- 2^{eme} génération (1986)[19] : Les réseaux sont basés sur des unités de calcul qui appliquent une "fonction d'activation" avec un ensemble continu de valeurs de sortie possibles (ex : exponentielle, sigmoïde, polynomiale...). Ces résultats réels peuvent être interprétés comme des taux de déclenchement de spike de neurones naturels. Cette génération est encore très utilisée en Intelligence Artificielle mais très peu en biologie à cause de son manque de pertinence biologique.
- 3^{eme} génération : Les réseaux sont basés sur des neurones à intégration et tir (ou "spiking neurons"). Dans cette génération, la notion de temps est cruciale. En effet, le temps exact d'émission de spike est pris en compte. On trouve de nombreux modèles dans la 3^{eme} génération [12], qui sont classés selon deux critères :
  1. La proximité à la réalité biologique ;
  2. La complexité computationnelle qui représente le temps de calcul et la mémoire occupée.

  Ces deux critères sont inversement proportionnels.

Actuellement, les deux équipes de l'I3S et de l'INRIA travaillent sur des modèles de 3^{eme} génération et plus particulièrement sur le modèle "Leaky Integrate and Fire" (*LI&F*) [4] car il est un bon compromis entre réalité biologique et complexité computationnelle.

1.3 Modèle Leaky Integrate and Fire (LI&F) avec probabilité

Ce modèle est actuellement parmi les plus utilisés à cause de sa simplicité [14] . Il se base sur une variable principale : le potentiel de membrane.

Définitions 1. ***Réseaux neuronaux Leaky Integrate and Fire.*** *Un réseau neuronal LI&F est un ensemble (V,A,w), où :*

- *V sont les neurones à intégration et tir,*
- *$A \subseteq V \times V$ sont les synapses,*

— $w : A \to \mathbb{Q} \cap [-1, 1]$ *est la fonction du poids de synapse associée à chaque synapse.*

On distingue trois ensembles de neurones disjoints : V_i *(neurones d'entrée(input)),* V_{int} *(neurones intermédiaires),* V_o *(neurones de sortie (output)), avec* $V = V_i \cup V_{int} \cup V_o$.

Un neurone LI&F v *est un ensemble* (Θ_v, s_v, y_v)*, où :*

— $\Theta_v \in \mathbb{N}$ *est le seuil d'émission,*

— $s_v : \mathbb{N} \to \mathbb{Q}_0^+$ [1] *est la fonction du potentiel de membrane définie par :*

$$s_v(t) = r.s_v(t-1) + \sum_{i=1}^{m} X_i(t).w_i \tag{1}$$

Où $s_v(0) = 0$*,* $X_i(t) \in \{0, 1\}$ *est le signal reçu au temps* t *par le neurone à travers sa* i^{eme} *synapse d'entrée, et* $r \in [0, 1]$ *est un nombre rationnel représentant le coefficient de fuite,*

— $y_v : \mathbb{N} \to \{0, 1\}$ *est la fonction de sortie du neurone, définie par :*

$$y_v(t) = \begin{cases} 1 & si \;\; s_v(t) \geqslant \Theta_v \\ 0 & sinon \end{cases} \tag{2}$$

Nous travaillons sur la formule à temps discret de cette définition pour déterminer l'état d'un neurone.
$s_v(t)$ correspond au potentiel de membrane à l'instant t.
r représente le coefficient de fuite du neurone, autrement dit une perte du signal due à sa dégradation au cours du temps ou aux courants ioniques de fuite. r est compris entre zéro et un où un serait l'absence de fuite et zéro la perte de tout le potentiel accumulé au temps $t - 1$.
w_i représente le poids du signal envoyé par les neurones prédécesseurs, sa valeur n'est pris en compte uniquement quand $X_i = 1$. Donc la somme des w_i représente la somme de tous les signaux entrant venant des autres neurones actifs. La valeur de w_i peut être positive (pour une activation) ou négative (pour une inhibition).

Pour résumer, cette formule nous dit que le potentiel de membrane d'un neurone est égal au potentiel déjà présent dans le neurone que l'on multiplie par un coefficient de fuite auquel on ajoute la somme des signaux venant des autres neurones à l'instant courant. Quand le potentiel dépasse le seuil d'émission Θ_v, il y a émission d'un spike. Ce seuil est constant chez un neurone mais peut varier d'un neurone à l'autre. Une fois le spike émis, il y a un phénomène de réinitialisation qui remet le potentiel de membrane à son état initial.
Ces neurones sont des simplifications des neurones de Hodgkin-Huxley. Par rapport au modèle de Hodgkin-Huxley, le neurone LI&F implique que la forme des potentiels d'action est négligée et que chaque pic est considéré comme un événement uniforme au moment de son apparition.

1. ensemble des nombres rationnels positifs incluant zéro

Cependant, dans le cadre de ce stage, nous introduisons des probabilités dans ce modèle dans le but d'avoir un modèle plus fin et plus proche de la réalité, ce qui n'est pas le cas dans la définition 1 au niveau du calcul de y_v. Il y a dans la littérature, différentes façons d'y parvenir, comme par exemple ajouter un "bruit" au potentiel de membrane calculé précédemment ; ce que l'on retrouve dans le modèle *Noisy Integrate and Fire* [5, 6]. Mais, parmi les différentes méthodes, nous avons décidé d'introduire les probabilités au niveau de l'émission des spikes car cela nous semble plus pertinent biologiquement. En supposant de diviser $s_v(t) - \Theta_v$ en l_k intervalles positifs et l_k intervalles négatifs, il faut remplacer l'équation 2 par :

$$P(y_v(t) = 1) = \begin{cases} 1 & si \ \ s_v(t) - \Theta_v \geq l_k \\ p_{2k} & si \ \ l_{k-1} \leq s_v(t) - \Theta_v < l_k \\ . & \\ . & \\ p_{k+1} & si \ \ 0 \leq s_v(t) - \Theta_v < l_1 \\ p_k & si \ \ -l_1 \leq s_v(t) - \Theta_v < 0 \\ . & \\ . & \\ p_i & si \ \ -l_k \leq s_v(t) - \Theta_v < -l_{k-1} \\ 0 & si \ \ s_v(t) - \Theta_v < -l_k \end{cases} \tag{3}$$

Avec $P(y_v(t) = 1)$ la probabilité que le neurone v émette un spike au temps t. De plus, nous avons la valeur du potentiel de membrane qui varie selon l'équation 1. Pour déterminer quelle probabilité correspond aux valeurs, nous suivons une courbe sigmoïde (que l'on peut voir dans la figure 1).

Nous distinguons trois types de noeuds fondamentaux dans ces réseaux :

— Entrée : les premiers neurones du réseau qui recevront uniquement des impulsions choisies par l'expérimentateur.

— Intermédiaire : qui reçoivent une ou plusieurs impulsions venant d'autres neurones et qui sont connectés à d'autres neurones.

— Sortie : qui affichent le résultat final et dont le signal de sortie n'est pas l'entrée d'autres neurones du réseau.

Il faut noter qu'il existe ce que l'on appelle une période réfractaire après un spike. Cette période est un moment après le spike où le potentiel ne peut pas augmenter et où il ne peut y avoir d'émission de nouveaux spikes. Sa durée peut varier d'un neurone à l'autre. Cependant elle est difficile à représenter en temps discret, nous avons alors décidé de la négliger partiellement car notre système nous permet de remettre le neurone à son état initial, empêchant ainsi une activation constante. Cependant nous ne pouvons régler sa durée.

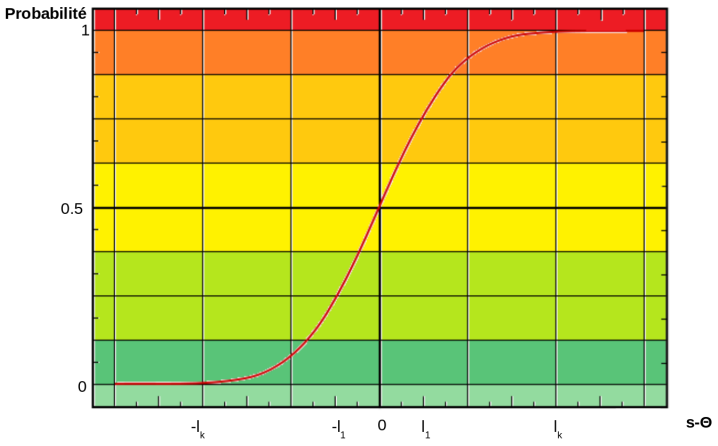


FIGURE 1 – **Graphique d'une courbe sigmoïde représentant les probabilités d'émettre un spike associées aux valeurs de la différence entre le potentiel du neurone et la valeur seuil :** Nous avons donc établi, que si la valeur du potentiel était très faible par rapport au seuil, soit $s_v(t) \ll \Theta_v$; il y aurait une faible probabilité d'émission, voir 0 si le potentiel est extrêmement bas (couleur verte). Si ces valeurs sont proches, $s_v(t) \simeq \Theta_v$; il y aura une plus grande probabilité d'émission. Enfin, si le potentiel dépasse de beaucoup le seuil,$s_v(t) \gg \Theta_v$; il atteint un palier il y aura une probabilité d'émission de 1 qui serait due à la saturation des neurones (couleur rouge).

1.4 Objectifs

Ce stage a pour objectif tout d'abord de modéliser des réseaux neuronaux probabilistes sur PRISM, puis d'élaborer des propriétés, et enfin de tester si les modèles vérifient ces propriétés. Ces propriétés nous permettent de déduire de possibles réductions de réseaux. Par exemple, si un neurone ne s'active jamais dans un réseau, on peut le supprimer du modèle. Ces réductions de réseaux pourront permettre de réduire les modèles existants, tout en permettant le respect de certaines propriétés cruciales, et donc de décomplexifier leur analyse. Il existe déjà des techniques de réductions différentes, mais celles-ci s'orientent plus vers des réseaux de 2^{eme} génération [7, 13]. Sur du long terme, ces réductions pourraient avoir une application dans le domaine médical, par exemple la possibilité de détecter des zones peu actives (voir inactives) du cerveau.

2 Matériels et méthodes

Tout le travail de modélisation, de formulation de propriétés et de vérification est effectué sur le logiciel *PRISM*. PRISM est un vérificateur de modèles probabilistes (probabilistic model checker)[11]. C'est un outil pour la modélisation formelle et l'analyse de systèmes qui présentent un comportement aléatoire ou probabiliste. Il est codé en Java et en C++, et fonctionne sur tous les principaux systèmes d'exploitation.

Il permet de construire et d'analyser de nombreux modèles différents comme par exemple : des chaînes de Markov discrètes ou continues, des automates probabilistes, des processus Markoviens, etc.
Il a été utilisé pour analyser des systèmes provenant de nombreux domaines d'applications différents, y compris des protocoles de communication et multimédia, des protocoles de sécurité et des systèmes biologiques. PRISM est open source, publié sous la GNU General Public License (GPL).

2.1 PRISM

En ce qui concerne la modélisation, nous travaillons avec des chaînes de Markov à temps discret [15] (ou DTMC sur PRISM) afin de modéliser les réseaux neuronaux.

Définitions 2. *Une suite $(X_n)_{n\geq 0}$ de variables aléatoires à valeurs dans un ensemble au plus dénombrable E est une* ***chaîne de Markov*** *d'espace d'états E si et seulement si pour tout $k \in \mathbb{N}$, pour tout $(x_0, ..., x_{k+1})$ dans E tels que $\mathbb{P}(X_k = x_k, ..., X_0 = x_0) > 0$,*

$$\mathbb{P}(X_{k+1} = x_{k+1} \mid X_k = x_k, ..., X_0 = x_0 = \mathbb{P}(X_{k+1} = x_{k+1} \mid X_k = x_k) \quad (4)$$

Les chaînes de Markov peuvent être représentées comme des graphes où les états sont reliés entre eux par les probabilités de passer d'un état à l'autre (dans la représentation en graphe, on parle de *poids w* (voir définition 1)). Une règle fondamentale des chaînes de Markov est que la somme des probabilités sortantes d'un état doit obligatoirement être égale à un. De plus il faut bien noter que ici, l'information utile est uniquement contenue dans l'état présent ; autrement dit, pour déterminer l'état à $t + 1$, on ne s'intéresse qu'à l'état à l'instant t.

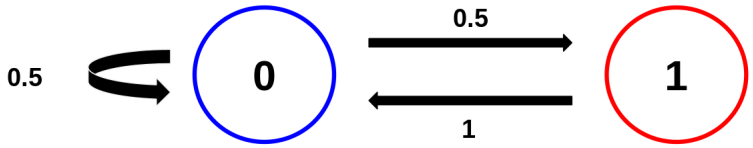


FIGURE 2 – **Exemple de chaîne de Markov à temps discret représentant un neurone simplifié :** Quand le neurone est inactif (la valeur du neurone est $n = 0$) il a une probabilité de 0.5 de rester inactif et de 0.5 de s'activer ($n = 1$) et d'émettre un spike. Dès que le neurone est actif, il y a une probabilité de 1 que le neurone repasse dans son état inactif.

Il est important de préciser que le langage de PRISM est basé sur le formalisme des *modules* réactifs d'Alur et Henzinger [1]. Le code est composé de deux parties :

— La partie déclarative, où l'on déclare toutes les constantes du programme soit en y entrant directement la valeur soit par l'intermédiaire de formules mathématiques.

— Les commandes, qui sont séparées en deux parties.

1. La *guard* qui représente les conditions à respecter pour passer à la deuxième partie de la commande.
2. L'*update* (ou mise à jour) qui indique les possibles transitions du module si la guard est respectée. Chacune de ces transitions est précédée d'une probabilité qui permet de choisir la transition effectuée.

On peut ajouter au début de chacune de ces commandes (entre les deux crochets au début de la ligne), une "étiquette" pour les *synchroniser* autrement dit pour les forcer à s'exécuter en même temps. En l'absence d'étiquettes, PRISM essaie d'exécuter ces commandes successivement au lieu de les exécuter en même temps.

Une commande se construit donc comme ceci :

$$[\,]guard -> probability_1 : update_1 + ... + probability_n : update_n\ ;$$

Enfin, PRISM met à disposition un système de coût et de récompense [10] qui permet d'associer des valeurs réelles à des états ou à des transitions comme par exemple augmenter une récompense de un à chaque fois qu'un neurone est actif (voir figure 3).

```
dtmc //Indique que l'on utilise des chaînes de Markov.

module exemple //Début du module.

        // Définit les états du module
        n : [0..1] init 0;
        // Définit les actions
        [] n=0 -> 0.5:(n'=0) + 0.5:(n'=1);
        [] n=1 -> 1:(n'=0);

endmodule //Fin du module

rewards "n"
        n=1:1;
endrewards
```

FIGURE 3 – **Exemple de code sous PRISM d'un neurone simplifié (issue de la figure 2) :** Tout ce qui est écrit après "//" constitue les commentaires, ce n'est pas pris en compte lors de l'exécution du code. n' est la valeur que prendra n au temps suivant. Ici aucune variable n'est renseignée. Quand la guard est $n = 0$, la mise à jour correspondante indique qu'il y a une probabilité de 0.5 que n reste à zéro et 0.5 de passer à 1. Quand $n = 1$, il y a une probabilité de 1 de passer à $n = 0$. Dans cet exemple les étiquettes sont inutiles, mais elles sont nécessaires pour synchroniser les neurones entres eux dans des réseaux plus complexes. De plus, une récompense a été ajoutée, ici chaque fois que $n = 1$ (donc à chaque émission de spike), la récompense "n" augmente sa valeur de 1.

2.2 La logique temporelle PCTL

Et en ce qui concerne l'élaboration des propriétés pour tester le modèle, nous utilisons dans PRISM le langage PCTL (Probabilistic Computation Tree Logic)[9] qui est une extension de la logique temporelle CTL [2]. Cette logique se compose de :

— *Propositions atomiques* AP tels que $x = 0$ ou $y < 10$.

— *Connecteurs logiques* comme $\vee$ (ou), $\wedge$ (et), ¬(négation), -> (implication),etc.

— *Opérateurs temporels* qui permettent de décrire l'évolution du système au cours du temps. Les opérateurs disponibles sont :

Path Quantifier

A : All $\rightarrow$ pour tout chemins.

E : Exist $\rightarrow$ il existe un chemin.

State Quantifier

X : Next $\rightarrow$ à l'état d'après.

F : Future (Finally) $\rightarrow$ dans un futur état.

G : Globally $\rightarrow$ toujours.

U : (p1) Until (p2) $\rightarrow$ propriété 1 vraie jusqu'à ce que la propriété 2 soit vraie.

Ainsi, avec ces opérateurs, nous pouvons définir des séquences de transitions entre les états.

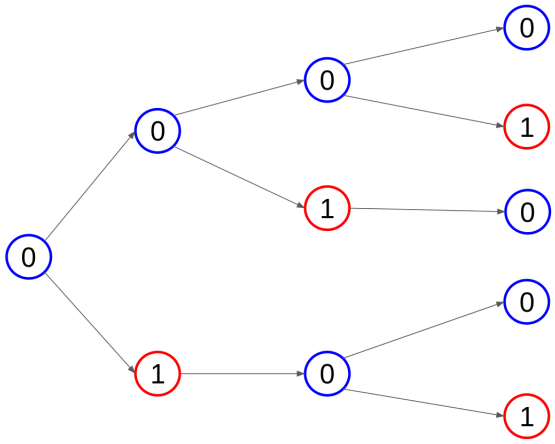


FIGURE 4 – **Exemple d'arbre représentant les évolutions possibles des états de la figure 2 :** Chaque noeud représente un état du neurone. Soit il est actif (1), soit il est inactif (2) et ses changements d'états sont ceux représentés dans la figure 2.

Exemple de propriétés en langage CTL basées sur la figure 4 :
$AG\ \ (n = 0)$: La proposition $n = 0$ est vraie tout le temps pour tout les chemins possibles. Cette propriété n'est pas vérifiée car n peut passer à 1.
$EF\ \ (n = 1)$: La proposition $n = 1$ finira par être vraie sur au moins un chemin. Cette propriété est vérifiée dès que $n = 1$.

Pour utiliser la logique PCTL, il suffit de rajouter l'opérateur P des probabilités en amont du chemin et des propriétés :
$P = 0,5\ \ X\ \ (n = 1)$: Il y a une probabilité égale à 0.5 que la proposition $n = 1$ soit vraie sur le prochain état en partant de l'état initial (mais pas forcément après). Cette propriété renvoie true comme résultat.
$P =?\ \ G\ \ (n = 0)$: Calcule la probabilité d'avoir un chemin où n ne passe jamais à un. Cette probabilité est très proche de zéro.

De plus, les opérateurs A et E ne sont pas disponibles tel quel avec la logique probabiliste sur PRISM. Par défaut, PRISM utilise A pour toutes les propriétés.

Enfin il est aussi possible d'élaborer des propriétés basées sur les *récompenses* [10]. Voici quelques exemples des connecteurs temporels utilisés :

F (Reachability reward) : Calcule le temps estimé pour qu'une proposition atomique donnée finisse par être vraie. Ici en précisant la récompense, PRISM calcule la valeur estimée de cette récompense.

C (Cumulative reward) : Calcule la valeur moyenne de la récompense dans un intervalle de temps fixe partant de zéro.

Cependant, l'outil récompense (ainsi que les propriétés liées) est encore expérimental, on peut donc s'attendre à quelques erreurs durant le déroulement des algorithmes.

2.3 Le model-checking

Le model checking [3] est un outil informatique qui permet de calculer si un modèle de système vérifie une spécification (propriété) donnée. Il a l'avantage d'être complètement automatisé, ainsi il nous répond par *true* ou *false* lorsqu'il teste une propriété du système. De plus, dans le cas où la propriété n'est pas vérifiée, le model checker nous fournit un contre exemple de la propriété. Par exemple, pour reprendre les données de la figure 4, $P = 1\ \ G\ \ (n = 1)$ renvoie un *false* car l'état initial est à $n = 0$. Ce qui permet soit de corriger le modèle soit d'affirmer ou de refuser des formules. Il prend comme modèle des graphes orientés appelés *structure de Kripke* [9] et utilise la logique temporelle pour établir les propriétés.

Définitions 3. *Soit AP un ensemble de propositions atomiques. Une structure de Kripke est un ensemble (S,$\bar{s}$,P,L) où :*

- *S est un ensemble fini d'états s ;*
- *$\bar{s} \in S$ est l'état initial ;*
- *$P : S \times S \rightarrow [0,1]$ est la matrice de probabilité de transition où $\sum_{s' \in S} P(s,s') = 1$ pour tout $s \in S$;*
- *$L : S \rightarrow 2^{AP}$ est une fonction d'étiquetage qui attribue à chaque état $s \in S$ l'ensemble de propositions atomiques AP qui sont valides dans l'état.*

Les Structures de Kripke sont des graphes orientés où chaque noeud représente un état du système. Si il y a une probabilité non nulle de passer d'un état à un autre, alors il y a un arc entre les deux noeuds. Notons que PRISM convertit des chaînes de Markov en structures de Kripke.

3 Résultat et discussion

Au début de ce stage, je suis parti de zéro : il n'y avait pas de code de réseaux neuronaux déjà écrit sur PRISM, mes encadrants avaient déjà modélisés des réseaux neuronaux mais sous un autre modèle de comportement avec LUSTRE [8]. C'est donc à partir d'articles sur les réseaux neuronaux, le manuel d'utilisation de PRISM et des informations fournies par mes encadrants que j'ai développé le code pour modéliser un neurone puis un réseau de neurones.

3.1 Modélisation d'un neurone

Pour modéliser un seul neurone, il nous faut déjà définir toutes les constantes qui vont être utilisées : le seuil d'émission d'un spike, le coefficient de fuite et les valeurs des impulsions des neurones (poids des arc). On désigne ensuite en tant que formule l'équation qui caractérise la modélisation LI&F du "potentiel"(voir annexe .1 ou équation 1) avec un arrondi car sa valeur étant un état, elle ne peut être qu'un nombre entier. Pour envoyer des signaux à ce premier neurone on commence d'abord par écrire un module de *générateur d'entrées* avec un seul état qui prend en valeur soit 0 (inactif) soit 1 (actif, donc émission d'un spike) et une ligne de commandes pour le contrôler. Maintenant pour le neurone il faut d'abord définir un module (voir annexe .1) de deux variables : la variable représentant les états inactif et actif du neurone ; et une variable qui aura comme valeur le potentiel de membrane de la formule de l'équation 1. Ensuite on définit plusieurs lignes de commandes qui prennent comme guard le fait que l'état du neurone est égal à zéro (inactif) et la valeur de "potentiel" par rapport au seuil. Et en fonction de cette différence entre "potentiel" et le seuil (voir figure 1), les commandes feront passer, avec une certaine probabilité, l'état du neurone à 1 (actif) ou bien rester à 0 mais dans ce dernier cas la valeur de "potentiel" sera mise à jour. Toutes ces lignes de commandes sont synchronisées avec une même étiquette afin qu'il n'y ait qu'un seul changement par unité de temps. Enfin on ajoute une dernière ligne qui prend comme guard le fait que l'état du neurone soit 1 et comme commande le fait de remettre le "potentiel" et l'état du neurone à leur état initial. Ceci nous permet ainsi de simuler une période réfractaire, ici sa durée n'est pas variable mais cela reste un instant où le potentiel de membrane n'augmente pas et où il ne peut pas y avoir de spike. Cela nous permet d'éviter d'avoir des neurones ayant une activité constante et sans interruption. Cette ligne dispose aussi d'une étiquette, nommée *reset*, afin de synchroniser la sortie du neurone avec l'entrée du prochain.

3.2 Modélisation d'un réseau

Commençons par un réseau simple de deux neurones. Pour des réseaux plus complexes il suffira de répéter ces opérations. Pour faire le deuxième neurone à partir du premier, PRISM nous fournit une alternative très pratique : le renommage de module (voir annexe .2). Cette fonction nous permet de dupliquer un module en le renommant ainsi que ses variables, permettant l'utilisation des nouvelles constantes (en prenant en compte qu'il faille aussi renommer les variables de la formule du potentiel). Etant donné les deux neurones, quand le premier s'activera, sa valeur passera à 1 et

sera multipliée par la valeur du poids de l'arc sortant (jusque là multipliée par 0), augmentant la valeur du potentiel du deuxième neurone. Cependant cela ne marche pas aussi facilement, en effet, en pratique, le premier neurone se réinitialise (se remet à 0) avant que le deuxième neurone ne prenne en compte son activation. Pour résoudre ce problème, nous avons créé un module de *transfert* (voir annexe .2) qui contient un seul état compris entre 0 et 1 et initialisé à 0, synchronisé avec la sortie du premier neurone. Grâce aux étiquettes, à chaque "reset" l'état du module passe à 1 et repasse à 0 en se synchronisant avec le neurone suivant. Ainsi la valeur 1 est prise en compte dans la formule "Potentiel" du neurone suivant. Cette synchronisation fait que les cas séquentiels et parallèles sont traités de façon homogène. En conséquence, tous les neurones envoient et reçoivent leurs signaux en même temps, ce qui est rarement le cas dans la réalité. Cependant cette synchronisation est nécessaire car son absence provoque l'activation de façon aléatoire des lignes de commandes.
Nous avons donc nos deux neurones synchronisés. Maintenant, pour chaque nouveau neurone, il suffit d'ajouter les constantes, un renommage de module "Neurone" et un renommage de module "Transfert" pour chaque sortie de neurone.

Cependant, un problème apparaît lorsque les réseaux commencent à se densifier. Il s'agit de la complexité computationnelle qui est l'étude formelle de la quantité de ressources (espace mémoire et/ou temps) nécessaire pour résoudre un problème algorithmique. En règle générale, plus un algorithme est proche de la réalité biologique plus il est complexe. Ici la complexité du programme dépend de deux facteurs : La taille du réseau et les paramètres (les constantes du seuil d'émission et des poids). Mais la compression des paramètres réduit la précision du modèle. Le seul moyen de réduire la complexité sans altérer le réseau est donc la réduction.
La taille du réseau implique un problème : l'étiquette reset provoque un "temps d'arrêt" durant lequel le signal est transféré au module. Durant cet instant, toutes les autres fonctions sont à l'arrêt. Donc un nombre important de spike émis par un neurone provoque de nombreux arrêt. Or, il faut multiplier cela par le nombre de neurones. Un réseau complexe aura donc plus de temps d'arrêts qu'un réseau simplifié.Il est donc important de réduire les réseaux.

3.3 Propriétés encodées et testées

Notre but est de pouvoir proposer des simplifications de modèles, cela consiste à pouvoir supprimer des neurones d'un réseau sans en perturber le fonctionnement [7, 13]. De manière générale, on peut retirer les neurones qui ne s'activent jamais car ils n'apportent rien aux réseaux. Ainsi, nous pouvons aussi enlever le neurone qui suit ce neurone *"mur"*, si il ne reçoit pas d'autre signal venant d'autres neurones alors il ne s'activera jamais et n'enverra pas son signal. Cette étape peut être répétée pour tous les neurones qui suivent tant qu'ils ne reçoivent pas de signaux venant d'autres neurones. Mais de la même manière, nous pouvons supprimer les neurones en amont du neurone"mur". Si le précédent n'envoie de signal qu'à ce neurone, le signal est perdu ; il en va de même pour tous les neurones en amont si ils n'ont qu'une sortie qui mène au neurone "mur".

Toutes les propriétés suivantes font référence à la figure 7. Commençons avec la

probabilité que le neurone 1 ne soit jamais actif (voir équation 5 [2]) malgré le fait qu'il reçoit des signaux en entrée. Si le résultat est une probabilité de 1, alors ce neurone est un neurone "mur" et peut être enlevé.

$$P =?\ [G\ \ n_1 = 0] \tag{5}$$

Pour simplifier ultérieurement ce réseau, nous utilisons une propriété basée sur les récompense que PRISM nous fournit :
Supposons vouloir enlever le neurone 4, cette propriété est basée sur la récompense du neurone de sortie v_o qui voit son score augmenter de un lorsque ce neurone émet un spike (voir figure 3), cette récompense représente donc le nombre d'activations de ce neurone final (voir équation 6). Le calcul effectué par le model checker ici est une moyenne du nombre d'activations de v_o en 100 unités de temps. Cette propriété doit être testée sur les deux modèles : le cas de base et le cas simplifié. Une absence de différence entre les deux résultats signifie que la simplification n'a pas affectée la fréquence d'activation du neurone, donc le neurone 4 est candidat à la suppression.

$$R\{"v_o"\} =?\ [C \leqslant 100] \tag{6}$$

Nous fixons également des propriétés qui doivent avoir le même résultat avant et après la réduction afin de vérifier que la dynamique du réseau reste inchangée. Ces propriétés concernent uniquement les neurones de sortie.

Par exemple, la propriété suivante teste la présence d'oscillations : elle est vérifiée si il y a des oscillations en sortie du neurone final mais elle ne l'est pas si il y a des oscillations qui s'arrêtent :

$$P =?\ [G\ ((n_{v_o} = 1\ => (n_{v_o} = 1\ U\ \ n_{v_o} = 0))\ \&\ (n_{v_o} = 0\ =>\ (n_{v_o} = 0\ U\ \ n_{v_o} = 1)))] \tag{7}$$

Cependant, nous ne pouvons pas faire de propriétés demandant qu'un neurone respecte une certaine forme d'oscillations, pour cela nous devons soit utiliser le système des récompenses qui ne peut pas être utilisé avec la logique PCTL, soit utiliser des X (pour next) imbriqués pour imposer un écart constant dans les oscillations. Mais puisque notre modèle est probabiliste, il existe toujours une probabilité non nulle que la propriété soit fausse une fois et donc, si l'on précise les oscillations, de renvoyer une probabilité de zéro systématiquement.

3.4 Algorithme de réduction de réseaux

Voici les pseudo-codes de l'algorithme de réduction :

2. où n_i est la sortie du neurone i

Algorithm 1 fonction : REDUCTION(v_i,V_{int},E)

$V'_{int} = V_{int} \setminus (v_i)$
$E' = E \setminus (v_i, v_j) + (v_k, v_i)$
$G' = (V'_{int}, E')$
for all v_j descendants de $v_i \in G'$ en visite DFS **do**
  if $\nexists (v_h, v_j)$ **then**
    $V'_{int} = V'_{int} \setminus (v_j)$ **and** $E' = E' \setminus (v_j, v_h)$
  end if
end for
for all v_k ancêtres de $v_i \in G'$ en visite DFS **do**
  if $\nexists (v_k, v_l)$ **then**
    $V'_{int} = V'_{int} \setminus (v_k)$ **and** $E' = E' \setminus (v_l, v_k)$
  end if
end for

FIGURE 5 – **Pseudo-code de la fonction REDUCTION :** Cette fonction est appelée par l'algorithme 2 de la figure 6 pour supprimer un neurone v_i ainsi que tous les autres neurones qui ont uniquement v_i comme descendant ou comme ancêtre.

L'algorithme de la figure 5 est une fonction qui prend en entrée un neurone et les arcs E et sommets V_{int} des neurones intermédiaires du réseau *G*. Elle commence par supprimer le neurone placé en entrée. Tout en suivant une visite *DFS*[3], elle analyse l'ensemble des arcs entrants dans les neurones descendants et supprime les neurones pour lesquels cet ensemble est vide (à cause de la suppression du premier neurone). Cette opération est effectuée sur tous les neurones descendants de celui placé en entrée, mais aussi sur tous les neurones ancêtres de ce neurone (en inversant le sens des arcs pour dérouler l'algorithme). Cet algorithme n'est qu'une fonction qui sera appelée plusieurs fois par l'algorithme de réduction de réseau :

3. Algorithme de parcours en profondeur (pour *Depth First Search*) récursif qui visite tous les sommets d'un graphe en partant de la racine, explore le premier chemin de ses successeurs jusqu'au moment où le sommet n'a plus de descendants non visités. Ensuite il remonte d'un niveau pour vérifier s'il ne reste pas de sommets à visiter, et ainsi de suite. Après déroulement, tous les neurones intermédiaires sont numérotés dans l'ordre précis de l'algorithme.

Algorithm 2 Réduction de réseau neural (*G*,*Prop*,ε)

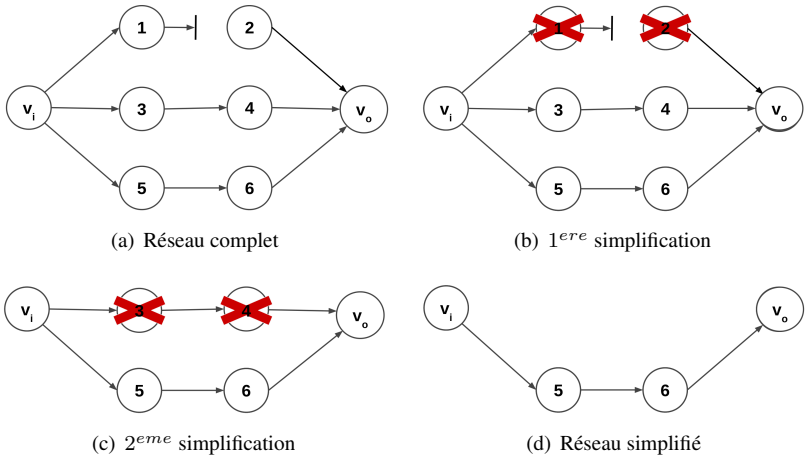
Soit $V = V_i \bigcup V_{int} \bigcup V_o$
visite $DFS(V_{int})$
for all $v_i \in V_{int}$ **do**
  $X_{i1}, ..., X_{im}$ sont les entrées de v_i
  $X_{i1}, ..., X_{im}$ valent 1
  if $Prob(G\ \ y_i = 0) = 1$ **then**
    REDUCTION(v_i,V_{int},E)
    $V_{int} = V'_{int}$ **and** $E = E'$
  end if
end for
mettre à 1 toutes les entrées des neurones appartenant à V_i
for all $v_o \in V_o$ **do**
  calcule $r_o = taux\ de\ spike(v_o)$ avec les récompenses de PRISM
end for
calcule $p = Prob(Prop\ is\ TRUE)$ avec PRISM
for all $v_i \in V_{int}$ **do**
  Soit $V' = V \setminus \{v_i\},\ E' = E \setminus (v_i, v_j) + (v_j, v_i)$
  for all $v_o \in V_o$ **do**
    calcule $r'_o = taux\ de\ spike(v_o)$ avec les récompenses de PRISM
  end for
  calcule $p' = Prob(Prop\ is\ TRUE)$ avec PRISM
  if $|r'_i - r'| \leq \varepsilon$ for all $v_o \in V_o$ **and** $|p' - p| \leq \varepsilon$ **then**
    REDUCTION(v_i,V_{int},E)
    $V_{int} = V'_{int}$ **and** $E = E'$
    for all $v_o \in V_o$ **do**
      calcule $r_o = taux\ de\ spike(v_o)$ avec les récompenses de PRISM
    end for
  end if
end for

FIGURE 6 – **Pseudo-code de l'algorithme de réduction de réseaux neuronaux :** Il est séparé en deux parties, une première qui teste si les neurones intermédiaires sont des "murs" et lance la fonction REDUCTION si besoin. Et une deuxième partie qui test si la suppression de neurones influe sur la propriété *Prop* du réseau et la fréquence de spike des neurones de sortie pour lancer aussi la fonction REDUCTION si besoin.Ici y_i correspond au n_i des formules précédentes

L'algorithme de la figure 6 prend en entrée un réseau *G* constitué d'un ensemble V de sommets et E d'arcs, une propriété *Prop* qui caractérise le réseau et une valeur ε qui représente une marge d'erreur. Il commence par faire une visite DFS de tout le réseau sauf pour les neurones d'entrées et de sorties. Ensuite pour chaque neurone, il

teste la probabilité que ce soit des “murs“. Si cette probabilité vaut un, alors il lance la fonction REDUCTION sur ce neurone. Puis il calcule le taux de spike des neurones de sortie et teste la véracité de la propriété. Toujours en suivant l'algorithme *DFS*, il teste si l'absence d'un neurone affecte le taux de spike final et la propriété. Si ce n'est pas le cas, alors il lance la fonction REDUCTION pour supprimer le neurone en question.

Voici un exemple schématique de ce que font, par étape, ces deux algorithmes sur un réseau de neurones simple :



(a) Réseau complet (b) 1^{ere} simplification

(c) 2^{eme} simplification (d) Réseau simplifié

FIGURE 7 – **Exemple du déroulement de l'algorithme de la figure 6 sur un réseau de huit neurones :** Les neurones intermédiaires sont numérotés selon une visite DFS. Étapes de simplifications : la première simplification (7(b)) est due à un neurone “mur” (le neurone 1) qui bloque le signal (représenté par une flèche interrompue sur son arc sortant). Son successeur est aussi supprimé car il ne reçoit aucun signal. La deuxième simplification (7(c)) est due au fait que la suppression du neurone 4 n'influe ni sur la propriété de réseau ni sur la fréquence de spike du neurone de sortie. De plus, son prédecesseur est aussi supprimé car son signal n'influe pas sur le dernier neurone.

Nous avons réunis dans le tableau 1 des valeurs de simulation de PRISM. A savoir, les fréquences de spikes des neurones de sortie, le nombre d'états possibles du modèle ainsi que le nombre de transitions et ceci aussi bien pour un modèle complet qu'un modèle réduit par l'algorithme.

réseau complet			réseau simplifié		
spikes	états	transitions	spikes	états	transitions
6.75	252 820	565 068	6.72	16 816	37 510
7.93	148 480	319 549	7.92	10 255	22 029
6.11	291 002	653 121	6.07	19 903	44 559
9.55	280 719	608 641	9.54	12 062	26 109
6.75	225 169	500 245	6.72	15 005	33 254
5.86	265 683	591 546	5.48	19 571	43 542
8.75	149 641	325 890	8.65	9 418	20 466
7.39	193 897	425 447	7.37	12 951	28 341
8.16	961 701	2 142 739	8.16	21 829	48 547
5.84	60 422	129 583	5.84	7 239	15 516
9.50	196 433	427 391	9.50	12 768	27 762
10.26	333 952	715 179	10.26	10 841	23 101
8.13	192 456	424 269	8.13	12 634	27 809
7.77	**273 260**	**602 205**	**7.72**	**13 946**	**30 657**

Tableau 1 – **Tableau des résultats de simulation des réseaux :** sont présents : les fréquences de spikes (en pourcentage) des neurones de sortie, le nombre d'états possibles, le nombre de transitions ainsi que leurs moyennes pour des réseaux complets de quatre neurones comportant un neurone "mur" et du réseau simplifié (après déroulement de l'algorithme) extraits de PRISM. Les réseaux considérés ont des paramètres différents mais cohérents avec le modèle. Ici un seul neurone a été supprimé.

Nous effectuons ici un test statistique, avec le logiciel R [17], sur les différents résultats de fréquences de spike afin de vérifier qu'il n'y a pas de différence significative provoquée par la réduction de notre algorithme. Pour cela nous utilisons d'abord un test de Shapiro pour tester la normalité des deux échantillons qui nous confirme que les échantillons ne suivent pas une loi normale [4] . On utilise donc un test non paramétrique : le *test de Wilcoxon - Mann - Whitney*. On pose comme hypothèses :

H_0 : Il n'y a pas de différence significative entre ces deux moyennes.

H_1 : Il y a une différence significative (qui n'est pas due au hasard) entre ces deux moyennes.

On obtient une *p-value* [5] de 0.77. Même sachant que la valeur de la p-value n'est pas exacte à cause d'ex-aequos dans les échantillons, $0.77 \gg 0.05$, on peut donc rejeter H_1 au risque de 5% de se tromper. On accepte donc H_0 : Il n'y a pas de différences significatives entre la fréquence de spike du neurone de sortie du réseau complet et celui du réseau réduit. Notre algorithme conserve donc bien cette propriété.

Au niveau de la complexité des algorithmes, nous regardons le nombre d'états possibles et le nombre de transitions du modèle (calculés automatiquement par

4. Loi de probabilité bien adaptée pour modéliser des phénomènes naturels issus d'événements aléatoire, aussi appelée loi de Gauss

5. Probabilité d'obtenir la même valeur du test si l'hypothèse nulle H_0 est vraie.

PRISM). Plus ils seront élevés, plus l'algorithme sera complexe. Nous utilisons ici aussi un test de Shapiro qui indique qu'aucun échantillon ne suit une loi normale [4], on utilise donc, là aussi, un test de Wilcoxon - Mann - Whitney avec les même hypothèses H_0 et H_1.
Pour les transitions, on obtient une p-value de $1.923e^{-7} \lll 0.5$. On peut donc rejeter H_0 et accepter H_1 : il y a une différence significative entre ces deux moyennes.
Pour les états possibles du modèle, on obtient aussi une p-value de $1.923e^{-7} \lll 0.5$. On peut donc accepter H_1 : il y a une différence significative entre ces deux moyennes.
On peut donc déduire de ces statistiques que notre algorithme affecte de manière significative la complexité du réseau sans affecter la fréquence de spike du neurone de sortie.

4 Conclusion et perspectives

Nous avons modélisé plusieurs réseaux neuronaux différents et établi plusieurs propriétés aussi bien avec les récompenses de PRISM qu'en PCTL. Avec ceci, nous avons créé un algorithme de réduction qui réduit significativement la complexité des réseaux sans altérer la fréquence de spike des neurones de sortie.

Pour la suite, il faut implémenter l'algorithme (ce qui n'a pu être fait par manque de temps) et ainsi continuer de le perfectionner. Pour cela il peut être intéressant de créer de nouvelles propriétés en PCTL pour pouvoir éventuellement éliminer les rewards si elles sont toujours expérimentales. De plus, le modèle lui-même peut être amélioré par exemple pour diminuer directement sa complexité, tester plusieurs propriétés en même temps pour être sûr de conserver la dynamique du réseau ou encore compenser voir supprimer le problème du "temps d'arrêt" provoqué par l'étiquette reset. Il faut aussi noter que notre algorithme ne propose qu'une seul réduction alors que certains réseaux peuvent avoir différents schémas de réductions possibles. Pour compenser cela, il faudrait lancer plusieurs fois l'algorithme avec différentes numérotations de neurone pour avoir plusieurs réductions différentes pour ensuite voir laquelle est la plus pertinente biologiquement. Ces résultats seront croisés avec ceux d'un autre stagiaire, Thibaud Lyvonnet, pour intégrer les probabilités aux modèles LUSTRE [6]. De plus, il est envisagé de faire une implémentation physique de nos neurones sur circuits électroniques.
La réduction de réseaux neuronaux reste donc un élément crucial dans le domaine de la bio-informatique.

6. Langage adapté à la de modélisation et spécification de réseaux de neurones [8].

Bilan des compétences acquises au cours du stage

Durant ce stage, j'ai appris à rédiger des documents avec LateX, un éditeur de texte scientifique, ce qui m'a permis de faire ce rapport. De plus, dès le début du stage, il m'a fallu rapidement apprendre le langage de modélisation que PRISM fournit pour les chaînes de Markov discrètes afin de commencer le plus tôt possible à modéliser des réseaux. De nombreux concepts vus dans ce stage qui avaient déjà été acquis durant les cours (étude statistique, chaîne de Markov, logique CTL...) ont fait l'objet d'une mise en pratique. J'ai appris à échanger avec d'autres personnes dont les sujets diffèrent complètement du miens et où chacun apportait des connaissances aux autres. Notamment en participant à la réunion du projet NeuComp de l'académie 1 de l'UCA dirigé par Benoit Moramond où nous avons évoqué la possibilité d'implémenter physiquement nos réseaux neuronaux sur circuits électroniques FPGA.

Enfin, travailler à l'INRIA, qui est un environnement international, fut très enrichissant.

Remerciement

Je voudrais remercier d'abord Elisabetta De Maria, Annie Ressouche et Daniel Gaffé dont les conseils m'ont toujours permis d'aller plus loin dans mon travail. J'aimerais aussi remercier Aude-Eva Chaudoreille pour son soutien tout au long de ce stage. Enfin je voudrais remercier tous les stagiaires et les membres de l'équipe STARS de l'INRIA pour leur gentillesse et dont la passion m'a encouragé à poursuivre dans cette voie.

Références

[1] R. Alur and T. Henzinger. Reactive modules. *Formal Methods in System Design*, 15(1) :7–48, 1999.

[2] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2) :244–263, 1986.

[3] E. M. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT press, 1999.

[4] E. de Maria, A. Muzy, D. GaffÃ©, A. Ressouche, and F. Grammont. Verification of temporal properties of neuronal archetypes using synchronous models. *HSB*, pages 97–112, 2016.

[5] V. Di Maio, P. Lansky, and R. Rodriguez. Different types of noise in leaky integrate-and-fire model of neuronal dynamics with discrete periodical input. *General physiology and biophysics*, 23 :21–38, 2004.

[6] N. Fourcaud and N. Brunel. Dynamics of the firing probability of noisy integrate-and-fire neurons. *Neural computation*, 14(9) :2057–2110, 2002.

[7] S. Gay, S. Soliman, and F. Fages. A graphical method for reducing and relating models in systems biology. *Bioinformatics*, 26 :i575–i581, 2010.

[8] N. Halbwachs. Synchronous programming of reactive systems.{T} kluwer academic publishers. *New York*, 1993.

[9] H. Hansson and B. Jonsson. A logic for reasoning about time and reliability. *Formal aspects of computing*, 6(5) :512–535, 1994.

[10] M. Kwiatkowska, G. Norman, and D. Parker. Stochastic model checking. In *International School on Formal Methods for the Design of Computer, Communication and Software Systems*, pages 220–270. Springer, 2007.

[11] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0 : Verification of probabilistic real-time systems. In G. Gopalakrishnan and S. Qadeer, editors, *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*, volume 6806 of *LNCS*, pages 585–591. Springer, 2011.

[12] W. Maass. Networks of spiking neurons : The third generation of neural network models. *Neural Networks*, 10(9) :1659–1671, 1997.

[13] J. E. Menke and T. R. Martinez. Artificial neural network reduction through oracle learning. *Intelligent Data Analysis*, 13(1) :135–149, 2009.

[14] A. Naldi, E. Remy, D. Thieffry, and C. Chaouiya. Dynamically consistent reduction of logical regulatory graphs. *Theorical Computer Science*, 412 :2207–2218, 2011.

[15] A. Papoulis. Brownian movement and markoff processes. *Probability, random variables, and stochastic processes*, pages 515–553, 1984.

[16] D. Purves, G. J. Augustine, D. Fitzpatrick, W. C. Hall, A. LaMantia, J. O. McNamara, and S. M. Williams, editors. *Neuroscience*. Sinauer Associates, Inc., 3rd edition, 2006.

[17] R Development Core Team. *R : A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2008. ISBN 3-900051-07-0.

[18] F. Rosenblatt. The perceptron : A probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6) :386, 1958.

[19] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. Technical report, DTIC Document, 1985.

Annexes

```
formula potentiell = floor(r*potentiela+k*kx+l*lx);

module neurone

        n : [0..1] init 0;// 0 = inactif et 1 = spike
        potentiela : [-500..1000]init 0;
        //chaque fois que n=0, on met à jour le potentiel
        [to] n=0 & potentiela<=0.2*T
        -> 1:(n'=0) &( potentiela'=potentiell);//pas de spike
        [to] n=0 & potentiela<=0.3*T & potentiela>0.2*T
        -> 0.05:(n'=1) + 0.95:(n'=0) &( potentiela'=potentiell);
        [to] n=0 & potentiela<=0.5*T & potentiela>0.3*T
        -> 0.10:(n'=1) + 0.90:(n'=0) &( potentiela'=potentiell);
        [to] n=0 & potentiela<=0.6*T & potentiela>0.5*T
        -> 0.20:(n'=1) + 0.80:(n'=0) &( potentiela'=potentiell);
        [to] n=0 & potentiela<=0.8*T & potentiela>0.6*T
        -> 0.30:(n'=1) + 0.70:(n'=0) &( potentiela'=potentiell);
        [to] n=0 & potentiela<=0.9*T & potentiela>0.8*T
        -> 0.40:(n'=1) + 0.60:(n'=0) &( potentiela'=potentiell);
        [to] n=0 & potentiela<=1.1*T & potentiela>0.9*T
        -> 0.50:(n'=1) + 0.50:(n'=0) &( potentiela'=potentiell);
        [to] n=0 & potentiela<=1.2*T & potentiela>1.1*T
        -> 0.60:(n'=1) + 0.40:(n'=0) &( potentiela'=potentiell);
        [to] n=0 & potentiela<=1.4*T & potentiela>1.2*T
        -> 0.70:(n'=1) + 0.30:(n'=0) &( potentiela'=potentiell);
        [to] n=0 & potentiela<=1.5*T & potentiela>1.4*T
        -> 0.80:(n'=1) + 0.20:(n'=0) &( potentiela'=potentiell);
        [to] n=0 & potentiela<=1.7*T & potentiela>1.5*T
        -> 0.90:(n'=1) + 0.10:(n'=0) &( potentiela'=potentiell);
        [to] n=0 & potentiela<=2.0*T & potentiela>1.7*T
        -> 0.95:(n'=1) + 0.05:(n'=0) &( potentiela'=potentiell);
        [to] n=0 & potentiela>2.0*T
        -> 1:(n'=1);//emission d'un spike

        [reset] n=1-> 1:(n'=0) & (potentiela'=0); //reset apres spike
endmodule
```

Annexe .1 – **Extrait du code d'un seul neurone sur PRISM :** sont présents ici la constante permettant de calculer le potentiel ainsi que le module "neurone" qui calcule les probabilités d'activation (les probabilités et les mises à jours ont été mises à la ligne pour plus de lisibilité).

```
module transfert12

        z : [0..1] init 0;
        [reset1] true -> 1:(z'=1);
        [to] true -> 1:(z'=0);

endmodule

module neurone2 = neurone1
[ n=n2, potentiela=potentielb, k=z,kx=nx, l=0, reset1=reset2]
endmodule
```

Annexe .2 – **Extrait du code d'un deuxième neurone sur PRISM :** sont présents ici le module "transfert" du neurone 1 au neurone 2 ainsi que le module du second neurone via la fonction de rennommage des modules.