31 JUILLET 2019

MÉMOIRE DE STAGE

Avec l'encadrement de M. Romain Laborde et M. Arnaud Oglaza.

Tuteur UPSSITECH : M. Abdelmalek Benzekri

STAGE M1

« UTILISATION DES RECURRENT NEURAL NETWORKS POUR LA DETECTION D'INTRUSION DANS LE RESEAU »

SYLVAIN LAPEYRADE
M1 STRI INGE UPSSITECH
118 Route de Narbonne, 31062 Toulouse

www.github.com/sylvainlapeyrade/RNN_Intrusion-Detection_Keras

Remerciements

Je souhaite tout d'abord remercier mon tuteur de l'UPSSITECH et directeur de l'équipe SIERA à l'IRIT, M. Abdelmalek Benzekri, pour m'avoir accordé l'opportunité de rejoindre son équipe et sa bienveillance tout le long de mon séjour chez eux.

Je remercie tout particulièrement mes encadrants à l'IRIT, M. Romain Laborde et M. Arnaud Oglaza pour leur confiance alors que je n'avais que peu d'expérience dans le domaine du stage. Ainsi que pour m'avoir permis d'utiliser les ressources de calcul de la plateforme de l'IRIT OSIRIM, sans laquelle je n'aurai pas pu obtenir de tels résultats. Enfin, un grand merci pour leurs nombreux conseils et le temps qu'ils m'ont consacré durant mon stage. J'ai énormément appris durant ma période à l'IRIT et ils y sont pour beaucoup.

Je voudrais aussi mentionner l'enseignement dispensé par la formation ingénieur STRI à l'UPSSITECH qui a su nourrir mes réflexions et entraîner ma curiosité et mon envie d'apprendre tout au long de ces deux années. Ainsi que les enseignements-chercheurs qui m'ont communiqué leur passion pour la recherche.

J'aimerais enfin exprimer ma gratitude à tout le personnel de l'IRIT et de l'UPSSITECH qui ont contribué de près ou de loin à la bonne réalisation de mon stage de Master 1 pendant cette période de quatre mois.

Table des matières

Introduction

Ce stage a été effectué dans l'Institut de Recherche en Informatique de Toulouse : IRIT[1], situé dans le quartier de Rangueil à Toulouse. Cette Unité Mixte de Recherche a été fondée en 1990 en partenariat avec l'Université Toulouse 3 : Paul Sabatier, le Centre National de Recherche Scientifique (CNRS), l'Institut National Polytechnique de Toulouse (INPT) et l'Université des Sciences Sociales de Toulouse : Toulouse 1 Capitole.

J'ai, durant ma période de stage allant du 1er avril 2019 jusqu'au 31 juillet 2019, pris place dans l'unité de recherche SIERA : Service IntEgration and netwoRk Administration, dirigée par mon tuteur UPSSITECH M. Abdelmalek Benzekri. Cette équipe appartient au département de recherche ASR : Architectures, Systèmes, Réseaux.

Mon stage intitulé « Utilisation des Recurrent Neural Networks pour la détection d'intrusion dans le réseau » s'inscrit dans la volonté de l'équipe de développer leur expertise sur "l'apprentissage machine" dans le cadre de la sécurité des réseaux. L'apprentissage automatique et les réseaux de neurones sont effectivement des concepts émergents au fort potentiel pour l'informatique. Ils promettent d'avoir un impact important dans les domaines des réseaux et de la sécurité, et logiquement mes encadrants chercheurs s'y intéressent.

Contexte

Le sujet d'étude se concentrait sur l'utilisation des réseaux de neurones récurrents pour la détection d'intrusion automatique dans le réseau. Jusqu'à présent, la détection d'intrusion se fait par le biais de systèmes de détection d'intrusion (IDS). Ce sont des mécanismes destinés à repérer des activités anormales sur une cible, comme un réseau sur la Figure 1 , permettant d'avoir une connaissance sur les tentatives réussies et échouées des intrusions.

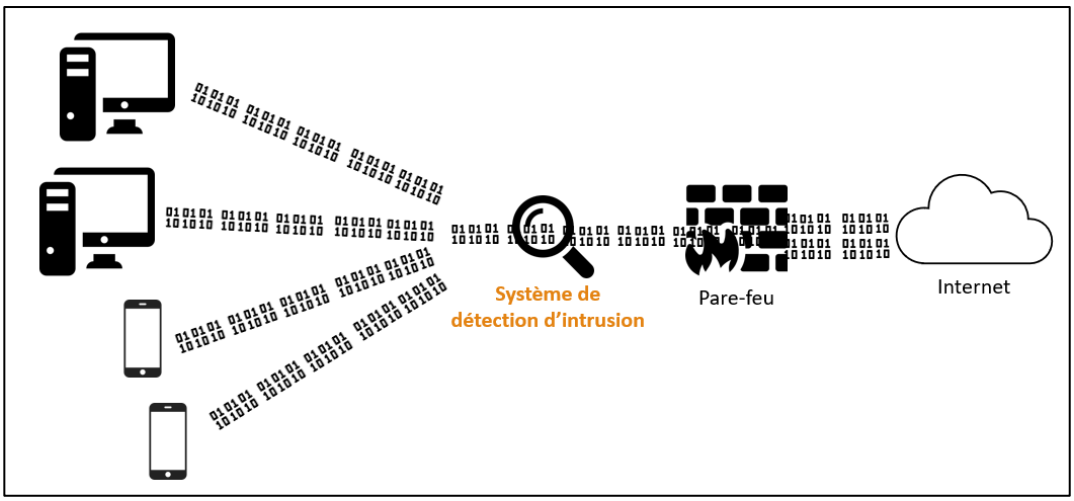


Figure 1 : Système de détection d'intrusion dans un réseau

Ces systèmes fonctionnent selon deux principales méthodes d'analyse :

- Par *Signature* : Des éléments spécifiques sont recherchés, comme une séquence de bits particulière.
- Par *Anomalie* : Les éléments que l'on sait normaux sont classifiés, et on analyse plus en profondeur tous les autres.

[1] www.irit.fr

Ces méthodes d'analyses sont efficaces pour la détection basée sur des paramètres bien connus, mais elles présentent cependant plusieurs limites. Tout d'abord, elles sont dépassées dans le cas de nouveaux types d'intrusions ou de schéma d'attaque non décrit au préalable. Ensuite, ces méthodes ne peuvent modéliser que de simples schémas de reconnaissance ce qui limite leur application à des comportements bien connus et précisément modélisable.

Une des solutions à cette limitation est l'utilisation de système d'autoapprentissage capable de repérer des menaces inconnues jusqu'alors. Ceci est dû à leur habileté à différencier le trafic « normal » et « anormal » en apprenant à partir des données d'un réseau supervisé. Les méthodes d'apprentissage peuvent apprendre des comportements complexes d'un système. En apprenant des classes entières de trafics normaux et d'attaques, les classificateurs entraînés ont le potentiel de détecter les irrégularités et les attaques inconnues jusqu'ici. De plus, les méthodes de Machine Learning promettent de fournir une solution capable de détecter d'éventuelles attaques en temps réel, de sorte que les contre-mesures puissent être prises en temps opportun. [1]

Enfin, ces solutions d'apprentissages automatiques présentent la caractéristique d'être dynamique contrairement à la mise en place de règles sur le comportement du réseau qui est très souvent statique.

Pour l'étude et la mise en place d'une solution d'apprentissage machine, nous nous sommes inspiré de l'article de Ralf C. Staudemeyer : « Applying long short-term memory recurrent neural networks to intrusion detection »[2]. Ce papier présentait les avantages de fournir des détails sur les procédures effectuées ainsi que de bons résultats eux-mêmes très complets.

Étude

1. Jeux de données KDD Cup '99

Un autre avantage de l'article de R. C. Staudemeyer [1] est qu'il étudie le jeu de données « KDD Cup '99 »[3] qui est sûrement l'ensemble de données le plus étudié pour la détection d'intrusion dans les réseaux. Il a été construit en 1998 par la DARPA et le MIT Lincoln Labs, et consiste en une variété de simulation de trafic normal et d'intrusions enregistrés pendant 9 semaines dans un réseau local de type militaire.

L'environnement a acquis pendant ce délai des informations sur près de cinq millions de connexions sous forme de données brutes de déversement TCP. Chaque connexion fut ensuite labélisée comme normale ou selon le nom de l'attaque. Elles sont représentées sur environs 100 bits, la Figure 2 et la Figure 3 en présentent un aperçu.

```
(0,udp,private,SF,105,146,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0.00,0.00,0.00,
0.00,1.00,0.00,0.00,255,254,1.00,0.01,0.00,0.00,0.00,0.00,0.00,0.00,.normal)
```

Figure 2 : Exemple d'une connexion "normale" du jeu de données KDD Cup '99

```
(0,icmp,ecr_i,SF,508,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,31,31,0.00,0.00,0.00,
0.00,1.00,0.00,0.00,255,255,1.00,0.00,1.00,0.00,0.00,0.00,0.00,0.00,.smurf)
```

Figure 3 : Exemple d'une connexion "anormale" du jeu de données KDD Cup '99 ici « smurf »

[2] sacj.cs.uct.ac.za/index.php/sacj/article/view/248
[3] kdd.ics.uci.edu/databases/kddcup99/kddcup99.html

Enfin, les attaques sont regroupées sous quatre types principaux :

1. **DoS** (Denial of Services) : dénis de service
2. **Probe** (Sonde) : surveillance non autorisée
3. **U2R** (User to Root) : accès non autorisé à des privilèges sur la machine
4. **R2L** (Remote to Local) : accès non autorisé d'une machine locale à partir d'une distante

L'ensemble de la collection de données a été divisé en plusieurs jeux de données pour qu'il y ait un jeu d'entraînement et un de validation. Cette nuance sera détaillée dans la partie suivante expliquant plus précisément le fonctionnement de l'apprentissage machine.

Le jeu de données a en effet été utilisé pour la compétition Knowledge Discovery and Data Mining 1999. L'objectif était d'entraîner un système d'apprentissage automatique à prédire correctement la nature des connexions fournies avec un premier jeu de données. Les performances, c'est-à-dire le pourcentage de bonnes prédictions, étaient ensuite évaluées sur un second ensemble de données. [2]

Un des objectifs premiers du stage était d'arriver au même ordre de précision que les vainqueurs de cette compétition et de R. C. Staudemeyer qui a obtenu des résultats meilleurs encore [1].

2. L'apprentissage automatique

La différence fondamentale entre la programmation classique et l'apprentissage automatique est que pour la première, le concepteur fournit à la machine un ensemble de règles (son programme) qui, appliqué à des données fournies, va générer un résultat. Pour le second, le développeur présente à la machine des données ainsi que les résultats attendus sur ces données afin de produire des règles. Ces règles peuvent ensuite être réutilisées sur de nouvelles données afin de trouver des résultats leur correspondant. La Figure 4 schématise le contraste entre les deux paradigmes. [3]

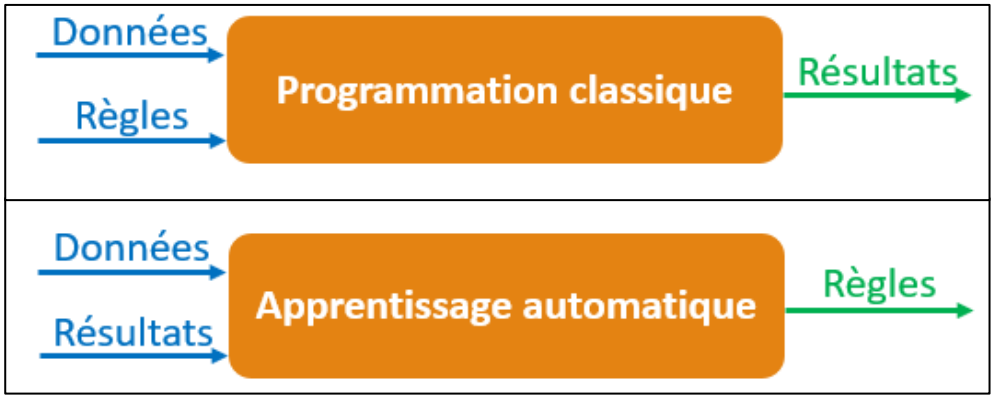


Figure 4 : Différence entre programmation classique et apprentissage automatique

a. Apprentissage supervisé

L'enjeu de l'apprentissage va donc être de trouver les meilleures règles possible dans l'optique que le système arrive à un très bon niveau de prédiction. En d'autres termes, il doit, à partir des données fournies, être en mesure de prédire les bons résultats le plus fréquemment possible. Nous pourrons ainsi, dans le cas des intrusions réseau, « apprendre » au système comment reconnaître une certaine attaque afin qu'il puisse la détecter la prochaine fois qu'il la rencontrera.

C'est là qu'interviennent les jeux de données. Ces derniers ont pour objectifs de fournir des informations sur une entité, ici sur une connexion, et de la caractériser, soit dans notre cas d'établir le type de la connexion. Quand le programmeur précise l'information attendue à la sortie système, c'est

de l'*apprentissage supervisé*. Bien sûr, plus il y a de données caractérisant l'entité de façon unique, plus il sera facile de l'identifier parmi les autres et donc de prédire la bonne sortie. [4]

b. Pondération des données

Toutefois, on s'aperçoit que toutes les caractéristiques n'ont pas la même importance pour chaque donnée. On peut se rendre compte qu'un caractère d'une connexion est propre à un certain type d'attaque. On attachera alors plus d'importance à cette valeur qu'aux autres. De même, il peut y avoir des informations identiques communes à toutes les connexions, des constantes. Ces informations ne pourront pas servir à différencier une connexion des autres. C'est pourquoi certaines données vont avoir une grande importance, donc un poids fort, quand d'autres de moindre pertinence vont avoir un poids faible, ou nul pour les constantes. C'est la *pondération* des données, voir le schéma Figure 5.

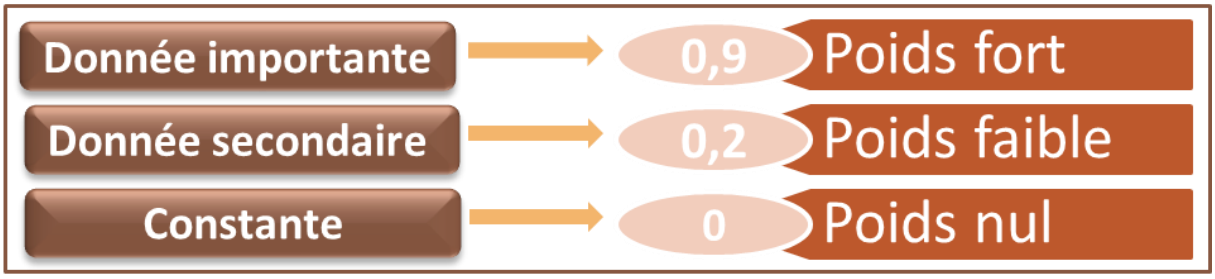


Figure 5 : Schéma de la pondération des données

c. Apprentissage optimal

Cependant, pour un bon apprentissage, la pondération ne doit pas être faite à l'extrême. En effet, si les règles d'apprentissages sont trop spécifiques au jeu de données avec lequel on apprend, elles ne seront pas forcément adaptées à de nouvelles données. C'est ce qu'on appelle le *surapprentissage*.

S'il y a, par exemple, une majorité d'attaques par dénis de service dans le jeu, les paramètres ayant des valeurs spécifiques pour ce type d'attaque auront un poids très important. Ceci afin d'identifier au mieux les comportements propres à ces intrusions et mieux les reconnaître. Toutefois, il pourra exister des jeux avec une répartition différente des données, une majorité d'attaques par sonde par exemple. Les paramètres avec un poids fort précédemment, pertinent pour les attaques par dénis de service, ne seront peut-être pas les plus importants pour le nouveau jeu. C'est pour cela qu'il faut essayer de ne pas « trop » apprendre à partir d'un seul échantillon de données. Il ne faut pas non plus tomber dans l'autre extrême et ne pas assez apprendre, et être en *sous-apprentissage*.

L'objectif va donc être de trouver l'intervalle d'apprentissage optimal pour avoir de bonnes prédictions sur le jeu de données d'entraînement tout en étant performant avec de nouvelles données. C'est pourquoi nous devons introduire un second jeu de données dit de validation. Ce second ensemble, dans notre cas également fourni par la DARPA, est simplement une autre partie de l'ensemble des données collectées. La répartition des types de connexions est toutefois volontairement différente pour que le système puisse vérifier qu'il reste bien généraliste et non trop spécifique à une répartition particulière.

La Figure 6 illustre l'évolution du taux d'erreur sur les prédictions faites par le système sur les deux jeux de données. On peut constater que le taux d'erreur diminue sur les deux jeux, jusqu'à un certain point où il remonte pour le jeu de validation.

C'est le moment où le système est en surapprentissage par rapport à ces données particulières. Il est donc important de vérifier tout au long de l'apprentissage si l'on reste général et donc que le taux d'erreur du jeu de validation ne remonte pas. Il faut alors arrêter l'apprentissage juste avant pour avoir des résultats optimaux.

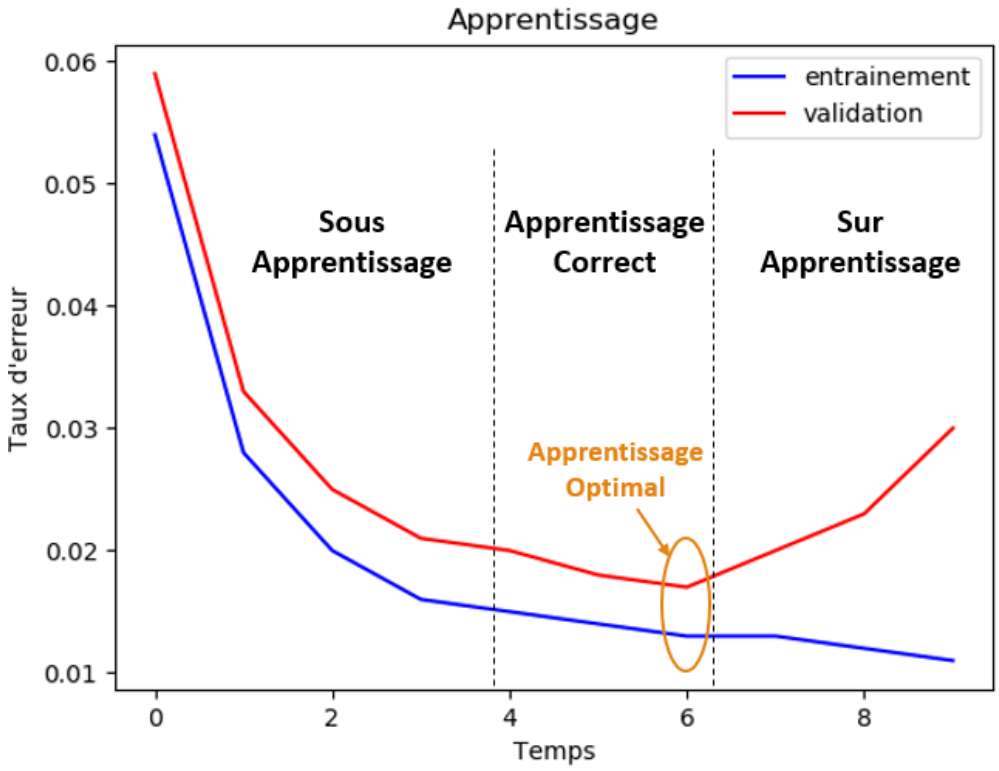


Figure 6 : Évolution du taux d'erreur en fonction du temps durant l'apprentissage

3. Réseaux de neurones

a. Principe du neurone artificiel

Les algorithmes d'apprentissage machine se basent sur le principe du neurone biologique contenu dans nos cerveaux. On peut voir ce neurone artificiel comme un système qui prendra des données en entrée et produira un résultat en sortie à la manière d'une fonction mathématique [5]. Une agrégation de fonction sera d'ailleurs contenue dans ce système. Ce regroupement sera plus ou moins complexe selon l'algorithme qui transformera les données en entrées. On peut voir un exemple de la transformation d'une connexion du jeu de donnée KDD Cup '99 à travers un neurone sur la Figure 7.

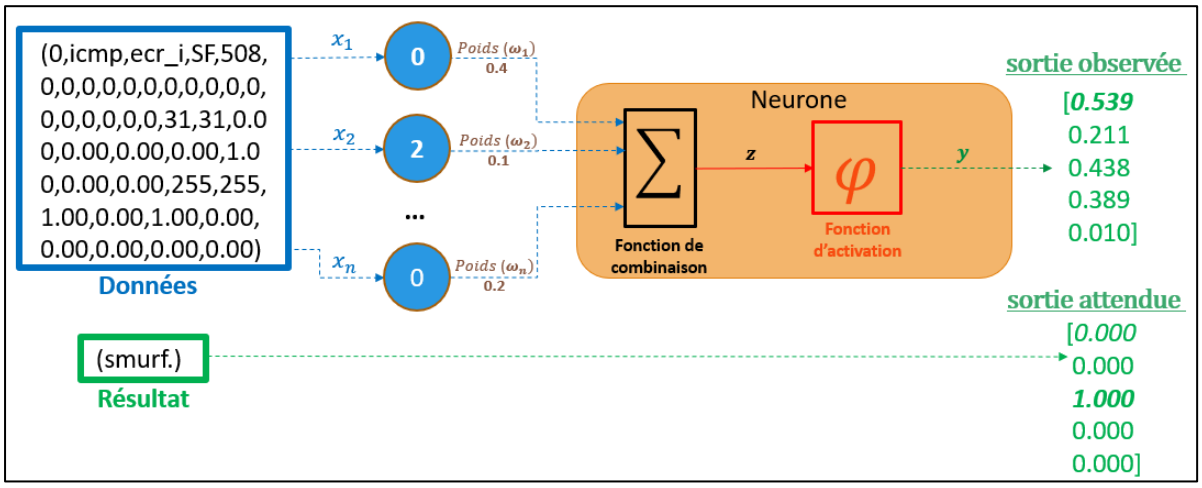


Figure 7 : Décomposition de l'apprentissage avec un neurone

Sur ce schéma, chacune des informations caractérisant la connexion, 41 entrées représentées de x_1 à $x_{n=41}$, devra être transformé en des valeurs que le système pourra utiliser pour faire des calculs. Ainsi, la première valeur de notre exemple « 0 » restera identique tandis que la deuxième « icmp » sera encodé en la valeur « 2 ». Avant d'entrer dans le neurone, chaque valeur sera pondérée, c'est-à-dire qu'un poids lui sera attribué. Sur l'exemple, la donnée x_2 (icmp) aura un poids de $2 * 0.1 = 0.2$.

Une fois chaque entrée pondérée, elles vont traverser des fonctions. Pour un neurone simple, ce sera une fonction de combinaison, comme celle de la Figure 8, puis une fonction d'activation, telle que celle présentée sur la Figure 9.

$$\textit{Somme pondérée:}\quad z = \sum_{i=1}^{n} (x_i * \omega_i)$$

Figure 8 : Fonction de combinaison - Somme pondérée

$$\textit{Fonction sigmoïde:}\quad y = f(z) = \frac{1}{1 + e^{-z}}$$

Figure 9 : Fonction d'activation - Fonction sigmoïde

Type de données	Numéro
Normal	0
Probe	1
DoS	2
U2R	3
R2L	4

Figure 10 : Correspondance entre types de donnée et leur numéro

Lorsque toutes les opérations sont effectuées, le neurone va produire en sortie une ou plusieurs valeurs, ici il y en aura une pour chaque type de données (4 types d'attaques et le type normal), soit cinq valeurs différentes. Dans notre contexte, elles correspondent à la probabilité que la connexion soit du type en question. Si l'on se réfère à notre exemple précédent, on peut observer sur la Figure 10 que le système prédit la probabilité que la connexion soit normale à 53.9%, Probe à 21.1%, DoS à 43.8%, U2R à 38.9% et de type R2L à 0.1%.

Pour sa prédiction finale, le système gardera le pourcentage maximal, dans ce cas 53.9% pour le type normal et identifiera donc la connexion comme normale. On constate cependant, toujours sur la Figure 7, que le résultat est une connexion de type DoS et donc la valeur maximale de sortie devrait se trouver dans la ligne correspondant à DoS comme ce qui est présenté pour la « sortie attendue » (valeur 1).

On retrouve alors des situations où le type de valeur maximale de la sortie observée est différent du type de la valeur maximale de la sortie attendue, ce qui est le cas ici (normale pour la sortie observée et DoS pour la sortie attendue). Dans ce cas, le système va modifier certains poids associés aux valeurs afin de modifier les résultats dans le but d'obtenir de meilleures prédictions. Cette modification sera effectuée au bout d'un certain nombre d'évaluations de connexions regroupées en « lots ».

Le gradient, soit la variation entre la sortie observée et la sortie attendue, sera calculé et les poids seront modifiés en fonction de la moyenne du gradient des connexions présentes dans le lot [6].

On testera ensuite ces nouveaux poids sur le lot de connexions suivant. C'est pourquoi il peut être intéressant de parcourir plusieurs fois l'ensemble de données pour soumettre à plusieurs reprises les mêmes données avec des poids différents et d'observer les changements de prédiction. La Figure 11 et la Figure 12 présentent la règle d'apprentissage sous forme pseudo-algorithmique et mathématiques.

```
APPRENTISSAGE
SI sortie observée ≠ sortie attendue ALORS
    calculer_gradient(erreur)
```

Figure 11: Pseudo-algorithme de l'apprentissage

$$\textit{Règle d'apprentisssage:}\quad w_i' = w_i + \propto (y_t - y) x_i$$

w_i' : poids i corrigé
w_i : poids i actuel
$\propto$: taux apprentissage
y_t : sortie attendue
y : sortie observée

Figure 12 : Formule de la règle d'apprentissage

b. Réseau de neurones à propagation avant

Il existe plusieurs méthodes pour augmenter l'efficacité des neurones. Premièrement, on peut ajouter un *biais* dans un neurone comme dans la Figure 13. Ceci permet de « biaiser » la valeur entrant dans la fonction d'activation. Ainsi, cette opération analogue et complémentaire au changement de poids permet à mesure que les données traversent le neurone de proposer chaque fois des résultats un peu différents. Cette pluralité pourra entraîner à terme de meilleures prédictions : plus il y a de prédictions différentes, plus il y a de chance d'en avoir des plus précises. C'est pourquoi ajouter du hasard dans les calculs aurait tendance à améliorer les résultats finaux.

Une autre façon de progresser dans les résultats est de mettre en réseau les neurones. Ce procédé illustré sur la Figure 14 va, en mettant bout à bout plusieurs cellules, générer des résultats en sortie plus élaborés qu'avec un seul neurone. Les données vont être modifiées pour chaque passage dans une cellule et seront pondérées à leur sortie. Ceci contribuera à avoir un plus large spectre de valeur. On peut observer cette rétropropagation du gradient de l'erreur sur le même schéma.

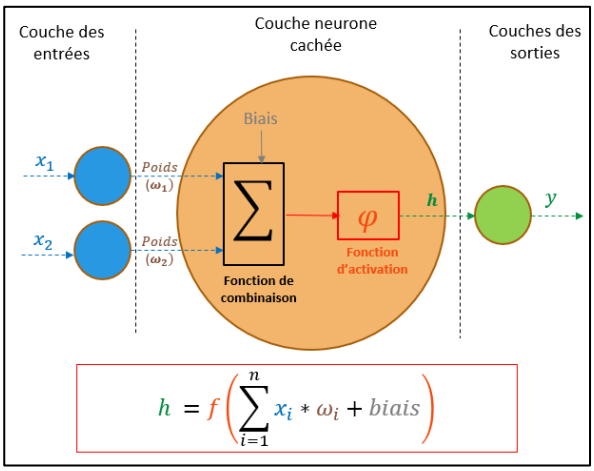


Figure 13: Généralisation d'un neurone

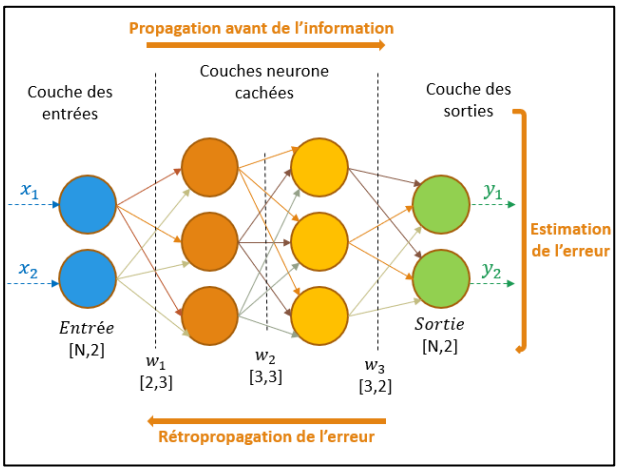


Figure 14: Mise en réseau des neurones

c. Réseau de neurones récurrents

Un moyen d'améliorer davantage les prédictions est de complexifier le système en utilisant le résultat d'un neurone pour le mettre en entrée dans l'itération suivante. Dans notre situation, il sera injecté dans la cellule, les données de la connexion étudiée actuellement et, en plus, le résultat de la connexion précédente. Ce paramètre supplémentaire va influencer le résultat actuel, comme pour le poids ou le biais. Le procédé est schématisé sur la Figure 14. Le fait d'utiliser le résultat précédent est appelé *mémoire à court terme* toujours par analogie avec notre cerveau [7]. D'une certaine manière, le système crée de « l'expérience ».

La mise en réseau de cette mémoire permet une rétropropagation du gradient de l'erreur plus efficace que dans le cas d'un réseau simple. Le réseau récurrent va d'une part profiter de l'estimation de l'erreur du réseau entier, résultant en un changement du poids sur l'ensemble du réseau. Et d'autre part, il va pouvoir bénéficier en entrée de chaque neurone du résultat de l'itération précédente du neurone, c'est la *rétropropagation à travers le temps* : RPTT sur la Figure 15 [6].

Toutefois, les réseaux de neurones récurrents classiques sont exposés aux problèmes de disparition et d'explosion de gradient. En effet, chacun des poids du réseau de neurones reçoit une mise à jour proportionnelle au gradient de l'erreur et du poids actuel à chaque itération de l'entraînement. Le problème est que dans certains cas, la variation de l'erreur va être tellement petite que le poids ne va pas être modifié assez pour que le résultat soit effectivement influencé.

Au contraire, si la variation est trop importante, l'influence sur le résultat sera proportionnelle et ainsi la valeur en sortie sera très différente de la précédente. Ces deux effets peuvent conduire à l'arrêt de l'apprentissage effectif du réseau. Les résultats seront presque identiques à chaque itération.

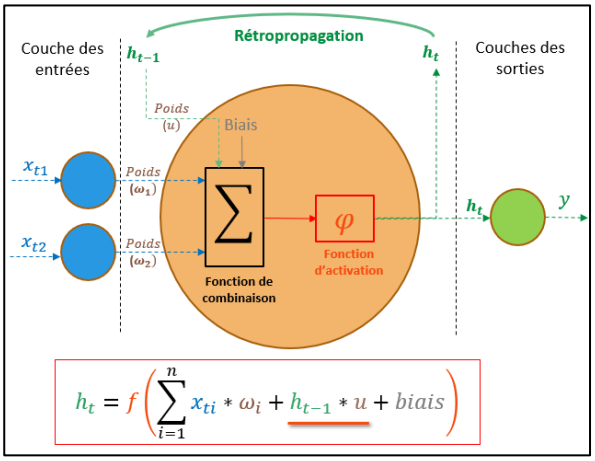


Figure 15: Cellule de réseau de neurones récurrent

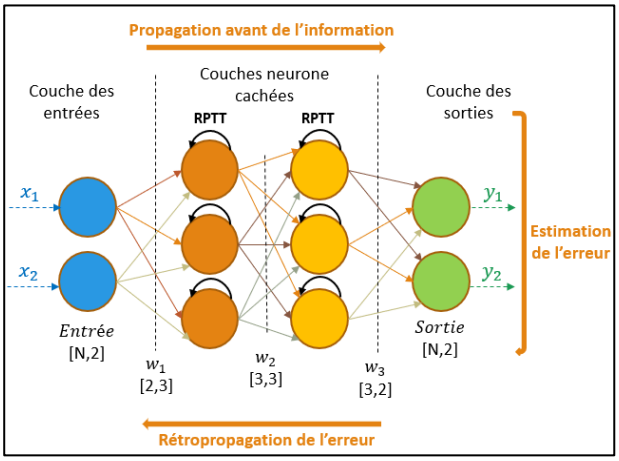


Figure 16: Réseau de neurones récurrents

d. Réseau de neurones LSTM

C'est pour résoudre les problèmes de gradient que l'architecture *Long Short-Term Memory* [8] ou « mémoire courte à long terme » a été mise en place. Une cellule LSTM reprend les bases d'une cellule de réseau de neurones récurrent en introduisant plusieurs mécanismes présents sur la Figure 17 :

- La *Cell State* : Cet *état de la cellule* apporte une « mémoire à long terme » en transmettant un nouveau paramètre en entrée de l'itération suivante, qui pourra être modifié ou non à chaque passage. La valeur de la première itération pourra donc influencer le résultat de la dernière.
- La *Forget Gate* : Comme son nom l'indique, cette porte va choisir les données en entrées que la cellule oubliera, c'est-à-dire celles qui n'influenceront pas la mémoire à long terme.
- L'*Input Gate* : Cette porte va déterminer, par d'autres opérations, si les données en entrée vont influencer la mémoire à long terme du réseau (*Cell state*).
- L'*Hidden State* : Comme pour les réseaux de neurones récurrents, c'est la « mémoire à court terme », cette valeur sera transmise en sortie du neurone. [9]

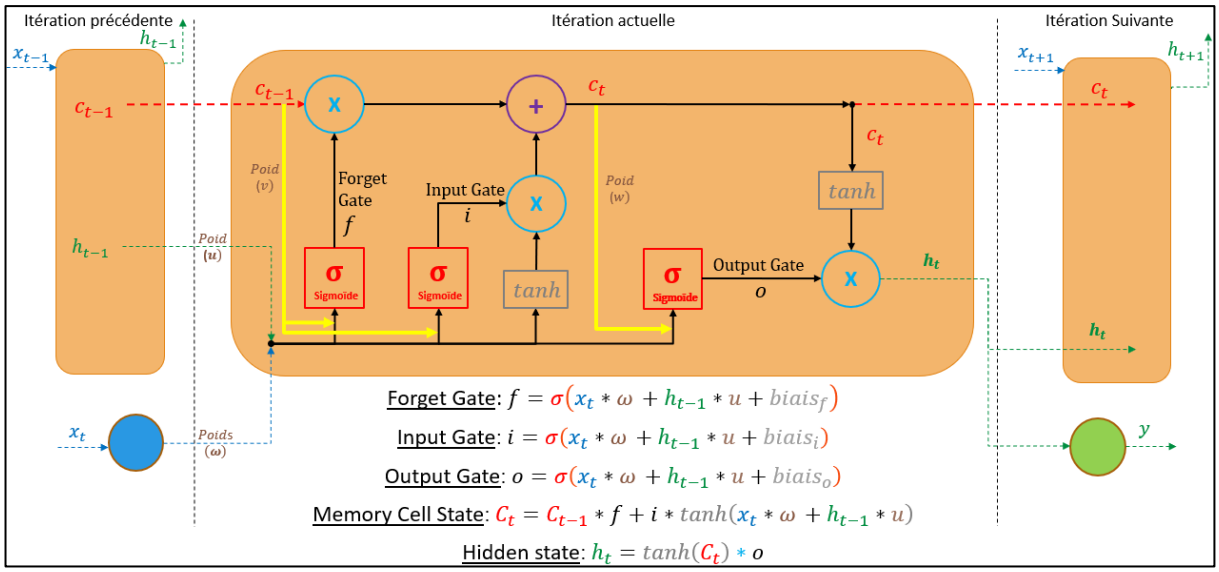


Figure 17: Constitution détaillée d'un neurone LSTM (Connexions « peephole » en jaune)

Une autre caractéristique importante de LSTM est l'utilisation de connexions *peepholes* [10] présentes en jaunes sur le schéma précédent. Ces liaisons ajoutent le paramètre responsable de la mémoire à long terme dans le calcul des autres mécanismes. Ainsi, cette valeur produite dans les itérations précédentes aura une plus grande importance pour le résultat de la sortie actuelle. La mémoire à long terme va ainsi avoir une plus grande influence sur la mémoire à court terme.

e. Réseau de neurones GRU

L'architecture *Gated Recurrent Units* a été introduite [11] comme une alternative à l'architecture LSTM. Elle a pour but de répondre aux problèmes du gradient et d'offrir une approche différente en ayant une meilleure efficience dans certains cas. La principale différence étant qu'il n'y a pas de paramètre de mémoire de la cellule pour représenter la mémoire à long terme. Comme le détaille la Figure 18, une cellule GRU est composée de plusieurs mécanismes :

- La *Reset Gate* : Détermine comment combiner les nouvelles données au paramètre de sortie de l'itération précédente.
- L'*Update Gate* : Comparable à l'Input et la Forget Gate de LSTM, elle décidera quelle part garder de l'état précédent pour l'état suivant.
- L'*Hidden State* : Comme précédemment, c'est la « mémoire à court terme », l'état qui sera transmis en sortie du neurone vers l'itération suivante.

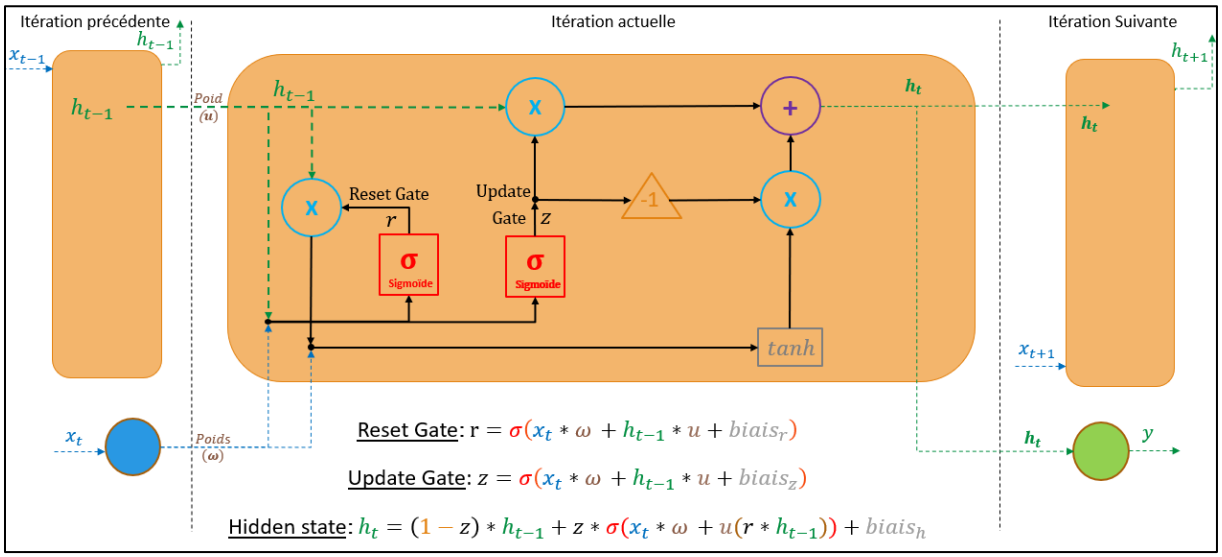


Figure 18: Constitution détaillée d'un neurone GRU

L'architecture Gated Recurrent Unit a l'avantage d'être moins complexe que Long Short Term Memory sans pour autant que les performances en soient significativement impactées. Cette approche réduite peut donc rendre l'entraînement plus facile et plus léger en coût de calcul.

4. Implémentation

a. Encodage

Pour que le système puisse effectuer des calculs avec les données en entrées, il est impératif de les transformer dans un format qui lui est compréhensible. C'est surtout le cas des chaînes de caractères qui pourront être transformés en nombre, par exemple en associant une même chaîne à un même nombre.

D'autres techniques, comme l'encodage « StandardScaler » sur la Figure 19, vont centrer et mettre à l'échelle indépendamment chaque caractéristique en calculant les statistiques pertinentes sur les

échantillons. La moyenne et l'écart-type sont ensuite stockés pour être utilisés sur des données ultérieures en utilisant la méthode de transformation [12].

Réduire les valeurs tout en conservant les écarts permettra d'optimiser l'apprentissage en évitant les problèmes de stagnation et en allégeant les calculs.

Paramètres d'entrée : X *(8 éléments)*

Index	0	1	2	3	4	5	6	7
0	http	181	0	0	5450	0	0	9
1	http	239	0	0	486	0	0	19
2	http	235	0	0	1337	0	0	29
3	http	219	0	0	1337	0	0	39

Encodage *Standard Scaler*

Index	0	1	2	3	4	5	6	7
0	-1,67319	-0,00287	-0,28286	-0,25209	0,13866	-0,04413	-0,00978	-1,69431
1	-1,67319	-0,00281	-0,28286	-0,25209	-0,01157	-0,04413	-0,00978	-1,60001
2	-1,67319	-0,00282	-0,28286	-0,25209	0,01417	-0,04413	-0,00978	-1,50570
3	-1,67319	-0,00284	-0,28286	-0,25209	0,01417	-0,04413	-0,00978	-1,41140

Figure 19: Encodage des données en entrée

Paramètres de sortie : Y *(5 éléments)*

Type de données	Numéro
Normal	0
Probe	1
DoS	2
U2R	3
R2L	4

Encodage One Hot

Numéro	0	1	2	3	4
0	**1**	0	0	0	0
1	0	**1**	0	0	0
2	0	0	**1**	0	0
3	0	0	0	**1**	0
4	0	0	0	0	**1**

Figure 20: Encodage des données en sortie

En sortie, un autre encodage comme le « OneHot » [13] pourra être utilisé. Chaque type de données sera d'abord associé à un numéro comme sur le premier tableau de la Figure 20. Ensuite, une colonne binaire sera créée pour chaque catégorie. Pour chaque ligne, la valeur de la colonne correspondant au type de donnée de la connexion sera mise à la valeur 1 et toutes les autres à 0. Par exemple, pour une donnée U2R, la colonne « 3 » sera à 1 et les quatre autres seront à 0.

Cet encodage présente le principal avantage de transformer chacune des valeurs pour qu'elles soient aussi différentes les unes des autres. Dans le premier tableau de la Figure 20, on peut observer que la valeur 1 est plus proche de 2 que de 4. Dans le second en revanche, les valeurs sont équidifférentes. Il est important de bien séparer ces sorties pour ne pas biaiser la classification.

b. Création d'un modèle d'apprentissage

Pour implémenter le réseau d'apprentissage, nous avons choisi d'utiliser le langage Python avec l'interface de programmation (API) Keras[4]. Elle présente en effet les avantages de pouvoir fonctionner sur un processeur central (CPU) et graphique (GPU). Elle permet rapidement et simplement le déploiement de prototypes ainsi que de la modularité et de l'extensibilité. De plus, elle peut utiliser la librairie Tensorflow pour le calcul en utilisant les graphes.

Nous allons créer un modèle que l'on entraînera en lui fournissant les données du jeu KDD Cup '99. La Figure 21 reprend sous forme de pseudo-code la création jusqu'à l'apprentissage d'un modèle. La première ligne correspond à la création d'un modèle séquentiel, c'est-à-dire que chaque couche sera ajoutée l'une à la suite de l'autre.

C'est ensuite aux lignes 3 et 4 que l'on ajoutera une couche de neurones LSTM, on peut voir ici que l'on aura 128 cellules, 8 paramètres en entrées (que nous avons sélectionné comme étant les plus impactant parmi les 41 présents dans le jeu de données) et le nombre d'échantillons correspondant au nombre de connexions que nous allons étudier. Nous ajoutons ensuite sur la ligne suivante une couche de « perte », qui va pondérer à 0 une part, sur l'exemple 20%, des données en entrée. C'est-à-

[4] www.keras.io

dire que ces données n'auront pas d'impact dans l'apprentissage. Ceci afin de prévenir le surapprentissage.

Les deux couches précédentes pourront être reproduites autant de fois que l'on veut de couche de neurones LSTM. Une fois ce nombre établi, une couche de liaison sera ajoutée au modèle pour lui indiquer qu'il y a cinq paramètres différents en sortie, un pour chaque type de données, cf. Figure 10. C'est dans cette couche que sera précisée la fonction d'activation produisant les estimations en sortie du réseau.

Sur les lignes 10, 11 et 12, le modèle sera compilé en paramétrant une fonction de la mesure du taux de perte, ce résultat nous permettra de voir de combien notre système se trompe dans ses prédictions. Le but sera donc de réduire ce taux de perte afin d'avoir des résultats le plus proche possible de la réalité. Le taux d'apprentissage ou « optimiseur » va déterminer sous quelle mesure les poids des neurones seront corrigés d'une itération à l'autre. Une correction avec un taux proche de 0 modifiera ces valeurs très lentement quand un taux proche de 1 changera radicalement les poids. La mesure de l'efficacité permettra d'afficher pour chaque itération la précision globale du modèle.

Il ne reste plus qu'à entraîner le modèle comme présenter sur les lignes 14, 15 et 16. On fournit les données et les résultats du jeu d'entraînement pour l'apprentissage et de validation pour évaluer la précision globale des prédictions, et le nombre de parcours de chaque ensemble.

```
1  modele = Sequentiel()
2
3  modele.ajouter(LSTM(cellule_LSTM=128,
4                      forme=(param_entree=8, echantillon=494021)))
5  modele.ajouter(Perte(20%))
6
7  modele.ajouter(Liason(param_sortie=5,
8                        fonction_activation='sigmoide'))
9
10 modele.compiler(taux_perte=erreur_quadratique_moyenne,
11                 taux_apprentissage=0.001,
12                 mesure_efficacite=['précision'])
13
14 modele.entrainement(x=entree_entrainement, y=resultat_entrainement,
15                     nb_iteration=20,
16                     validation=(x=entree_test, y=resultat_test))
```

Figure 21: Pseudo-Code de la création d'un modèle sous Keras

c. État du réseau

Il existe deux manières d'utiliser la mémoire à long terme du réseau. En effet, lors de l'apprentissage, plusieurs données sont mises en lot pour ensuite faire la moyenne sur le lot du gradient d'erreur des prédictions. Par exemple, sur la Figure 22, on prend 1000 connexions que l'on divise en 10 lots de 100 connexions chacun. L'enjeu est alors de savoir, si l'on veut transmettre la mémoire à long terme d'un lot à l'autre, donc à terme d'un bout à l'autre du jeu de données ou au contraire, si l'on ne veut garder cette mémoire seulement pour les connexions présentes dans le lot, puis réinitialiser la mémoire.

Le fait de conserver la mémoire à long terme au fil des lots peut être utile si la première connexion du jeu a de l'influence sur la dernière par exemple. Effectivement, dans les réseaux les connexions ne sont pas indépendantes les unes des autres. Pour une attaque par dénis de service, on pourra avoir plusieurs milliers de connexions envoyés sur le réseau en un court instant.

Le mode avec état est très adapté pour des séries temporelles, pour prédire le prix d'un billet d'avion par exemple. Dans ce cas, l'information datant de l'année d'avant, jour pour jour, sera souvent plus pertinente que la tendance de la semaine d'avant.

Le mode "sans état", lui, sera plus adapté pour des séries d'images d'une vidéo ou d'une suite de mots dans une phrase, dans lesquelles les images ou les mots immédiatement précédents auront beaucoup plus d'influence sur les suivants que les informations plus anciennes. Nous testerons les deux cas de figure dans la partie Résultats pour voir quel paradigme fonctionne le mieux.

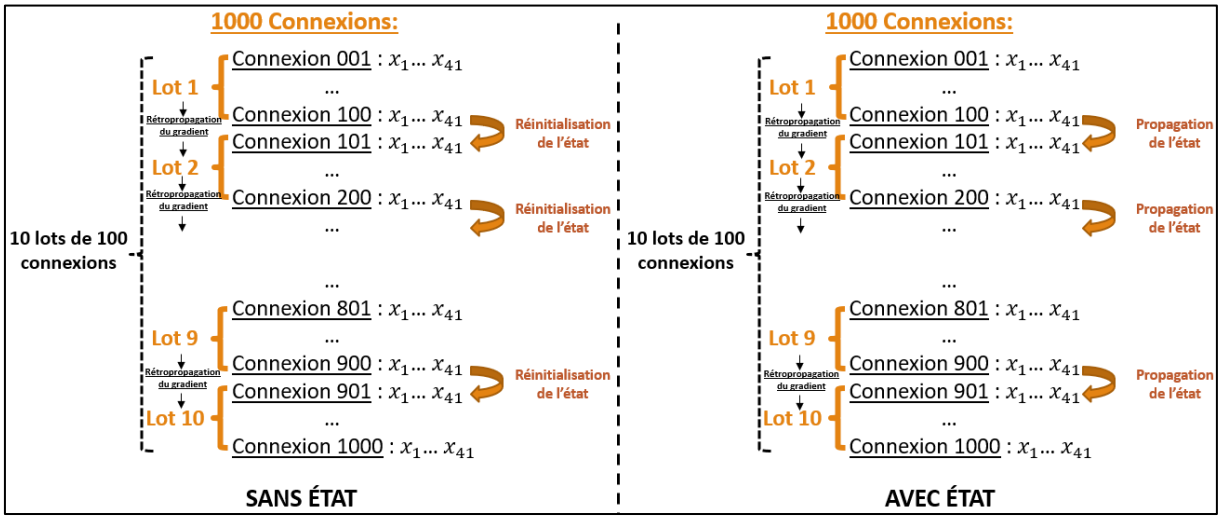


Figure 22: Réseau avec et sans état

d. Ordre des séquences

Une autre manière d'influencer l'apprentissage est de mélanger ou non les lots formés par les connexions. Ce procédé n'a de sens qu'en mode sans état puisqu'il fausserait complètement la mémoire du mode avec état. En changeant l'ordre dans lequel sont envoyées les séquences au modèle, on crée un biais supplémentaire permettant de ne pas avoir tout le temps les mêmes changements de poids au même moment.

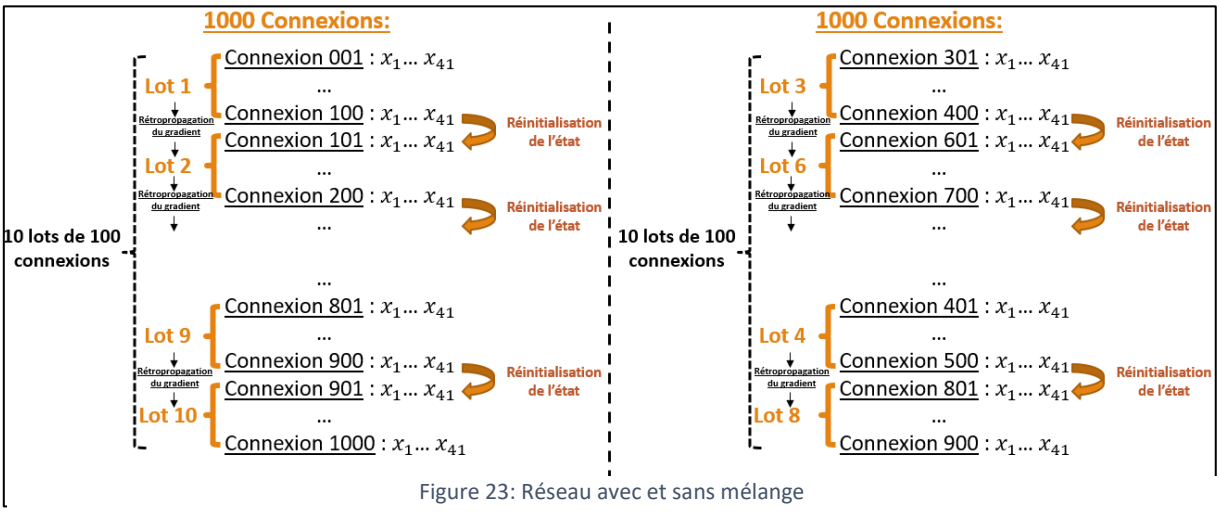


Figure 23: Réseau avec et sans mélange

Comme l'illustre la Figure 23, les poids sont changés à la fin du parcours du lot, et ainsi le lot 6 par exemple pourra directement bénéficier des changements du lot 3 sans altération. On garde cependant le principe de séquence de connexions et de mémoire à long terme sur ce lot. Les connexions précédentes appartenant au lot influenceront bien les suivantes.

5. Optimisation de l'entraînement

a. Équilibre des paramètres

Comme on vient de le voir, la création d'un modèle et l'entraînement en lui-même ne sont pas très compliqués avec l'API Keras. La difficulté réside plutôt dans la manipulation des données et le choix des paramètres du modèle pour arriver à obtenir un bon taux de prédictions correctes [14].

Chacun des paramètres va en effet pouvoir avoir un impact très important sur les performances et la précision du système. L'enjeu va donc être de trouver la meilleure combinaison possible entre tous ces facteurs. Cette recherche n'est toutefois pas sans coût, on peut constater sur la Figure 25 les différents temps d'entraînement totaux pour un même modèle que j'ai réalisé.

D'après ma comparaison, il faut près de 27 heures pour entraîner un seul modèle sur un processeur d'ordinateur portable. Cette durée est divisée par plus de six en utilisant une carte graphique de moyenne gamme. On peut encore gagner en efficience en utilisant la bibliothèque *CuDNN* optimisant l'utilisation des cartes graphiques notamment pour l'apprentissage machine.

Enfin, grâce à l'utilisation de la plateforme OSIRIM[5] rattachée à l'IRIT, ce même modèle est entraînable en moins de six minutes, soit environ 270 de fois plus vite qu'avec mon processeur de PC portable, ce qui représente un gain de temps considérable.

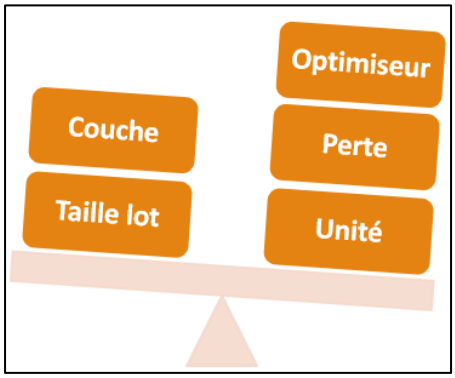


Figure 24: Équilibre des paramètres d'un modèle d'apprentissage

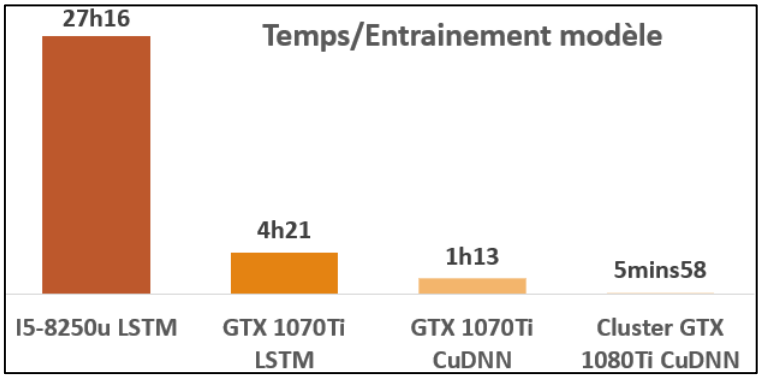


Figure 25: Temps total pour l'entraînement d'un même modèle avec différentes configurations

On comprend donc à quel point le matériel de calcul est important et pourquoi le machine learning devient de plus en plus populaire et accessible ces dernières années. Il faut en effet tester le modèle plusieurs fois pour chaque paramètre, à cause des divers mécanismes de hasard inhérent au système. Ceci permet d'obtenir une moyenne et de conclure sur la combinaison la plus efficace.

Néanmoins, même en ayant accès à un cluster de calcul comme OSIRIM, regroupant plusieurs cartes graphiques, on ne peut effectuer de recherche exhaustive, c'est-à-dire en testant toutes les combinaisons possibles de paramètres [15].

En effet, sur notre exemple même en prenant seulement quelques valeurs possibles pour quelques paramètres, on arrive à un temps de calcul extravagant. Comme schématisé sur la Figure 26 et la Figure

[5] osirim.irit.fr/site/fr/articles/presentation

27, il faudrait près de 1 150 000 entraînements soit l'équivalent de plus de 13 ans d'entraînements sans interruption pour un temps d'entraînement moyen de six minutes par modèle pour tester chaque combinaison de quelques valeurs sur quelques paramètres seulement.

Recherche Exhaustive	Recherche meilleure valeur
5 encodeurs	(5 encodeurs
* 6 optimiseurs	+ 6 optimiseurs
* 4 fonctions d'activation	+ 4 fonctions d'activation
* 6 valeurs de perte	+ 6 valeurs de perte
* 4 nombres de couches	+ 4 nombres de couches
* 10 nombres d'unités	+ 10 nombres d'unités
* 4 tailles du lot	+ 4 tailles du lot)
* 10 entrainement	* 10 entrainements
= **1 152 000** entrainements	= **390** entrainements

Figure 26: Comparaison d'une recherche exhaustive et de celle mise en place

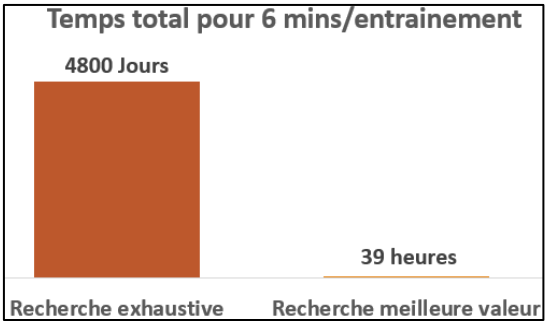


Figure 27: Comparaison de la durée d'une recherche exhaustive et de celle mise en place

Il est par conséquent impératif d'utiliser un algorithme de recherche plus optimisé pour accélérer la comparaison de paramètres tout en essayant un maximum de combinaison. La recherche empirique, consistant à essayer une certaine variation de paramète et d'observer les résultats de ce changement après, peut fonctionner pour un faible échantillon de paramètres. Mais à mesure que la complexité augmente, cette méthode est trop laborieuse et un algorithme automatique lui sera préféré.

b. Algorithme de recherche de paramètre

En automatisant la recherche empirique, on pourra, à partir d'un modèle de référence établi arbitrairement, comparer les résultats en modifiant seulement un paramètre à la fois et en testant un ensemble de valeurs. On entraînera par exemple dix modèles identiques et l'on fera la moyenne des cinq taux de pertes les plus bas pour chaque combinaison afin de privilégier la constance du modèle par rapport aux valeurs aberrantes. Nous pouvons observer les résultats de ce procédé pour le choix de l'encodage sur la Figure 28. Nous constatons alors ici que c'est l'encodage « StandarScaler », déjà explicité plutôt, qui a la moyenne la plus basse de ses 5 taux de perte de validation minimale.

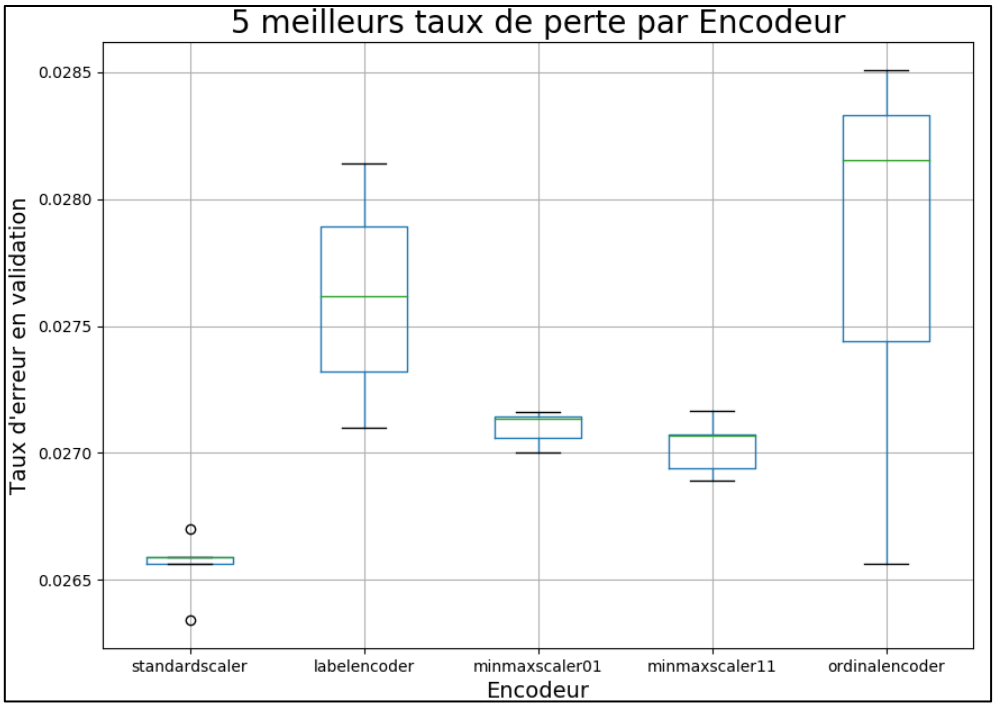


Figure 28: Recherche du meilleur encodage

Si la moyenne des pertes est inférieure à celle du modèle de référence, la combinaison deviendra alors référence et l'on pourra tester d'autres paramètres avec de nouvelles valeurs. Si au bout d'une recherche complète regroupant l'ensemble des valeurs, le modèle de référence ne change pas, alors on peut arrêter la recherche.

Cette recherche de la minimisation du résultat de la fonction de perte entre les différents modèles est illustrée par la Figure 29, où l'on voit que sur ce cas c'est le modèle de référence qui a le taux de perte le plus faible, il sera donc ici conservé. On peut également conclure sur la Figure 26 et la Figure 27 qu'il faudra beaucoup moins de temps à cet algorithme pour trouver la meilleure combinaison. En effet, chaque recherche complète ne prendre que 39h pour le modèle étudié.

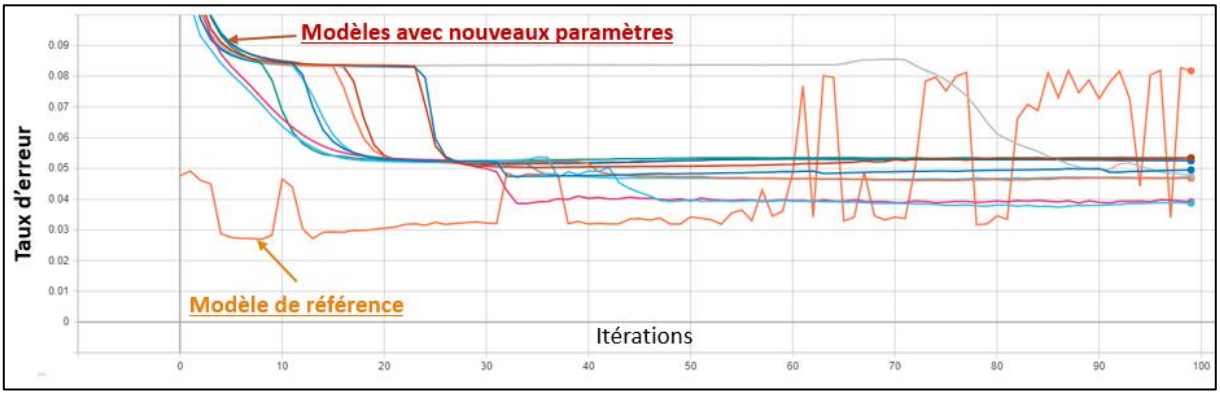


Figure 29: Comparaison du taux de perte de plusieurs modèles différents au fil des itérations

En outre, si le modèle de référence est bien choisi, par expérience, on peut n'avoir besoin que d'une ou deux recherches complètes, ce qui est respectivement 3000 et 1500 fois plus rapide qu'une recherche exhaustive. La Figure 30 présente un exemple de diminution du taux de perte en fonction des changements de paramètres. Chaque diminution correspond à la découverte d'une valeur d'une caractéristique plus performante que la précédente. Si la courbe reste constante pour l'ensemble des paramètres, alors aucun de plus performant n'est trouvé.

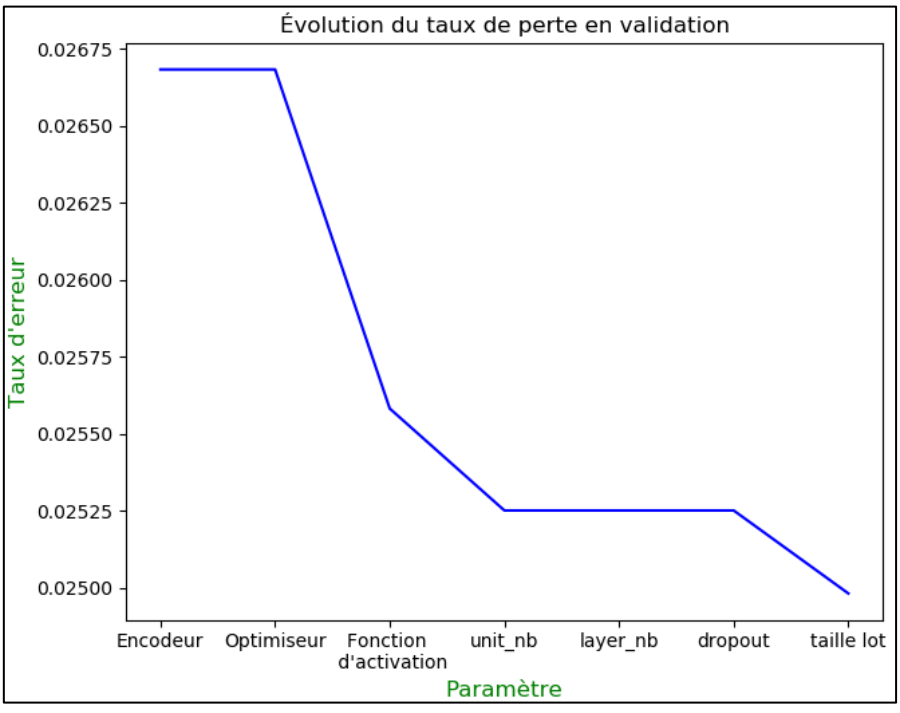


Figure 30: Évolution du taux de perte de validation sur application de l'algorithme

6. Résultats

a. État et mélange

Pour établir quel type de modèle fonctionnait le mieux sur notre ensemble de données, nous avons construits les architectures de réseaux présentées plus tôt. D'abord, nous pouvons observer sur la matrice de confusion de la Figure 31 que nous obtenons de meilleurs résultats en matière de précision dans un mode *sans état* par rapport au mode *avec état*.

De même, il est intéressant de constater que le mode avec état, conservant la mémoire à long terme tout le long du jeu de données , ne fait que des prédictions (se référer aux colonnes de la Figure 31) de type Normal ou DoS. Il n'y a donc aucune reconnaissance des types Probe, U2R et R2L. On peut cependant constater qu'avec seulement des prévisions effectives de deux types sur cinq, le modèle est très proche sur le plan des performances du mode sans état, ne conservant une mémoire à long terme que pour la durée du lot, qui lui détecte l'ensemble des types.

Pour expliquer cette différence, il faut se référer à la Figure 33 et la Figure 34 présentant la composition du jeu de données. On remarque dès le premier coup d'œil que les types Normal et DoS composent près de 99% du jeu de données. De même, on peut observer que les ensembles de connexions comportent des séquences de plusieurs milliers de données du même type DoS ou Normal, tandis que les données Probe, U2R et R2L sont représentés de façon plus disparate.

Cela semble être la raison pour laquelle le mode avec état, prenant en compte l'ensemble du jeu, ne fait pas de prédiction en dehors des catégories DoS et Normal tant le nombre de connexions ce ces types est élevé par rapport au reste. Le bon score de précision général s'explique donc par le fait que le jeu est très majoritairement composé des seules données prédites, sous forme de séquences.

Concernant le choix de mélanger ou non les séquences du jeu, nous constations sur la Figure 32 que c'est le mode avec mélange qui obtient les meilleurs résultats. En effet, le fait d'injecter les lots dans des ordres différents semble améliorer la détection générale et particulièrement des données R2L, non reconnu sans mélange. Ceci peut se traduire par le fait que les séquences R2L soient plus petites, ainsi avoir de nombreux poids différents peut être bénéfique pour leur identification. Au vu des performances, les prédictions suivantes seront faites sans état et avec mélange.

Prédiction / Réel

	Normal	Probe	DoS	U2R	R2L	% Correct
Normal	**59541**	0	1035	0	0	98.29
Probe	3555	**0**	611	0	0	00.00
DoS	16674	0	**213179**	0	0	92.74
U2R	69	0	1	**0**	0	00.00
R2L	16017	0	318	0	**0**	00.00
% Correct	62.11	00.00	99.08	00.00	00.00	PRÉCISION : 92.25 %

Jeu d'entrainement KDD'99 10% avec état

Prédiction / Réel

	Normal	Probe	DoS	U2R	R2L	% Correct
Normal	**59896**	242	436	2	17	98.84
Probe	91	**3359**	716	0	0	80.62
DoS	5293	937	**223623**	0	0	97.28
U2R	58	0	7	**2**	3	02.85
R2L	12386	78	109	0	**3774**	23.08
% Correct	77.06	72.76	99.43	50,00	99.47	PRÉCISION : 93.45 %

Jeu d'entrainement KDD'99 10% sans état

Figure 31: Matrices de confusion de l'entraînement avec et sans état

Prédiction / Réel

	Normal	Probe	DoS	U2R	R2L	% Correct
Normal	**59928**	229	436	0	0	98.90
Probe	306	**3239**	621	0	0	77.74
DoS	5343	740	**223770**	0	0	97.35
U2R	69	0	1	**0**	0	00.00
R2L	12497	55	3795	0	**0**	00.00
% Correct	76.69	75.97	97.87	00.00	00.00	PRÉCISION : 92.25 %

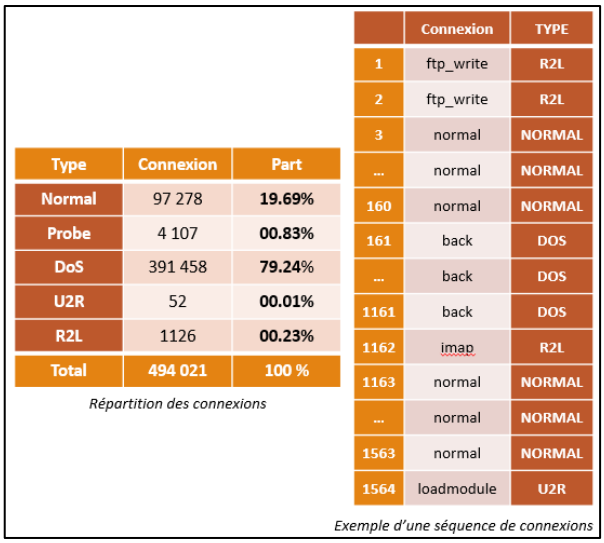
Jeu d'entrainement KDD'99 entier sans mélange

Prédiction / Réel

	Normal	Probe	DoS	U2R	R2L	% Correct
Normal	**60112**	230	251	0	0	99.20
Probe	254	**3594**	318	0	0	86.26
DoS	5877	1111	**222865**	0	0	96.95
U2R	63	0	1	**0**	6	00.00
R2L	13435	121	6	0	**2785**	17.03
% Correct	75.38	71.08	99.74	00.00	99.78	PRÉCISION : 93.03 %

Jeu d'entrainement KDD'99 entier avec mélange

Figure 32: Matrices de confusion de l'entraînement avec et sans mélange



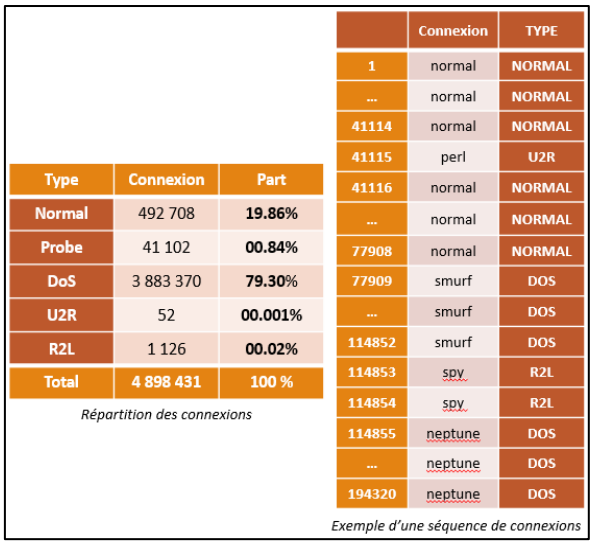
Type	Connexion	Part
Normal	97 278	19.69%
Probe	4 107	00.83%
DoS	391 458	79.24%
U2R	52	00.01%
R2L	1126	00.23%
Total	494 021	100 %

Répartition des connexions

	Connexion	TYPE
1	ftp_write	R2L
2	ftp_write	R2L
3	normal	NORMAL
...	normal	NORMAL
160	normal	NORMAL
161	back	DOS
...	back	DOS
1161	back	DOS
1162	imap	R2L
1163	normal	NORMAL
...	normal	NORMAL
1563	normal	NORMAL
1564	loadmodule	U2R

Exemple d'une séquence de connexions

Figure 34: Composition du jeu de données KDD'99 10%



Type	Connexion	Part
Normal	492 708	19.86%
Probe	41 102	00.84%
DoS	3 883 370	79.30%
U2R	52	00.001%
R2L	1 126	00.02%
Total	4 898 431	100 %

Répartition des connexions

	Connexion	TYPE
1	normal	NORMAL
...	normal	NORMAL
41114	normal	NORMAL
41115	perl	U2R
41116	normal	NORMAL
...	normal	NORMAL
77908	normal	NORMAL
77909	smurf	DOS
...	smurf	DOS
114852	smurf	DOS
114853	spy	R2L
114854	spy	R2L
114855	neptune	DOS
...	neptune	DOS
194320	neptune	DOS

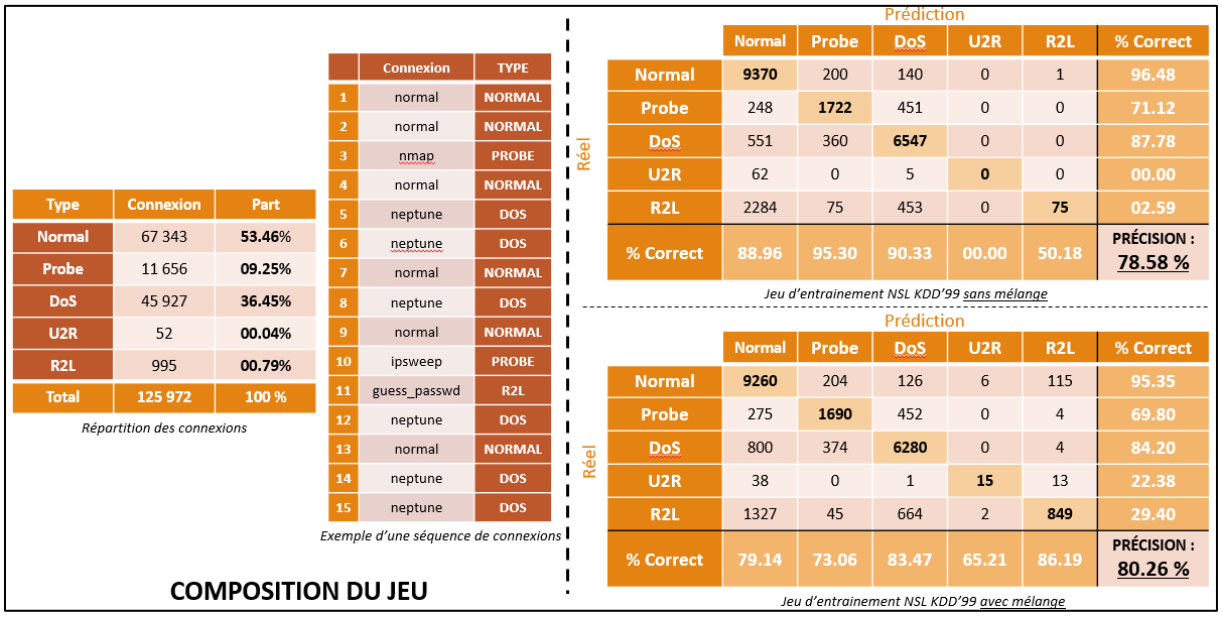
Exemple d'une séquence de connexions

Figure 33: Composition du jeu de données KDD'99 entier

b. Jeu de données NSL KDD

Nous avons voulu tester les paramètres qui marchaient le mieux avec une répartition différente des données afin d'observer si nous aurions un changement de performance. Le jeu de données NSL KDD[6] était parfait pour cet usage puisqu'il est composé d'une partie de KDD'99 mais la part de chaque type de connexions est différent afin d'équilibrer la répartition. La nouvelle partition et un exemple de séquences présentent dans le jeu peuvent être consultés sur la Figure 35.

Si le jeu est plus équilibré, il reste toutefois inégal dans son ensemble. Une fois entraîné, on constate sur la Figure 35 que les prédictions sont nettement moins bonnes que précedemment. Certes, dans la version mélangée les attaques U2R et R2L ont un meilleur pourcentage de bonne détection, mais les autres sont moins bonnes. En outre, les types U2R et R2L représentent moins de 1% des connexions totales. Un progrès dans leurs détections seules n'entraînera donc pas de meilleurs résultats globaux comme nous pouvons le constater.



Type	Connexion	Part
Normal	67 343	53.46%
Probe	11 656	09.25%
DoS	45 927	36.45%
U2R	52	00.04%
R2L	995	00.79%
Total	125 972	100 %

Répartition des connexions

	Connexion	TYPE
1	normal	NORMAL
2	normal	NORMAL
3	nmap	PROBE
4	normal	NORMAL
5	neptune	DOS
6	neptune	DOS
7	normal	NORMAL
8	neptune	DOS
9	normal	NORMAL
10	ipsweep	PROBE
11	guess_passwd	R2L
12	neptune	DOS
13	normal	NORMAL
14	neptune	DOS
15	neptune	DOS

Exemple d'une séquence de connexions

COMPOSITION DU JEU

Prédiction

Réel	Normal	Probe	DoS	U2R	R2L	% Correct
Normal	9370	200	140	0	1	96.48
Probe	248	1722	451	0	0	71.12
DoS	551	360	6547	0	0	87.78
U2R	62	0	5	0	0	00.00
R2L	2284	75	453	0	75	02.59
% Correct	88.96	95.30	90.33	00.00	50.18	PRÉCISION : 78.58 %

Jeu d'entrainement NSL KDD'99 sans mélange

Prédiction

Réel	Normal	Probe	DoS	U2R	R2L	% Correct
Normal	9260	204	126	6	115	95.35
Probe	275	1690	452	0	4	69.80
DoS	800	374	6280	0	4	84.20
U2R	38	0	1	15	13	22.38
R2L	1327	45	664	2	849	29.40
% Correct	79.14	73.06	83.47	65.21	86.19	PRÉCISION : 80.26 %

Jeu d'entrainement NSL KDD'99 avec mélange

Figure 35: Composition et résultats du jeu de données NSL KDD '99

[6] https://www.unb.ca/cic/datasets/nsl.html

Pour ce qui est de la baisse de précision sur les types Normal, Probe et DoS, ceci peut être lié à la grande discontinuité dans les séquences présentes dans le jeu. Pour KDD'99, il y a plusieurs séquences comprenant des milliers de connexions du même type à la suite. Par contre sur le jeu NSL KDD, on peut observer Figure 35 que les séquences sont extrêmement courtes en comparaison, et que les continuités sont de quelques connexions seulement. C'est pourquoi, en les regroupant, la mémoire du lot aura du mal à trouver un schéma logique d'une connexion à l'autre.

On peut d'ailleurs constater que plus un type de données est présent dans le jeu, plus il a de chances d'être correctement classifié. Ce qui n'était pas forcément le cas précédemment, d'où l'importance des séquences longues pour les réseaux de neurones récurrents et leur mémoire.

c. Résultats finaux

Si l'on regarde les résultats de Staudemeyer [1] retranscrits sur la Figure 36, on observe qu'il obtient de très bonnes prédictions sur les catégories Normal et DoS très présentes dans le jeu, des prédictions convenables sur les Probes et un pourcentage quasi nul sur U2R et R2L. La surreprésentation des types Normal et Probe ainsi que la longueur de leur séquence peut expliquer que LSTM marche aussi bien. Qu'en est-il cependant de Probe ? Ces attaques sont moins présentes que R2L dans le jeu de test, comme nous pouvons le voir sur les diverses matrices de confusions. Leurs représentations sont cependant du même ordre dans le jeu d'entraînement KDD'99 10%.

La réponse réside une nouvelle fois dans les séquences. Effectivement, les suites de connexions Probes sont plus longues et moins discontinues que celles de R2L. En outre, certaines attaques très présentes dans le jeu d'entraînement ne le sont pas dans le jeu de test. L'ensemble de ces facteurs peut expliquer un tel contraste dans les deux apprentissages.

En ce qui concerne GRU, s'il semble un peu moins performant, il existe probablement de meilleurs paramètres pour améliorer sa détection. Néanmoins, il respecte tout de même ses promesses d'offrir une précision comparable, ou un peu moindre, pour des calculs souvent plus rapides.

Prédiction / Réel

	Normal	Probe	DoS	U2R	R2L	% Correct
Normal	**60262**	243	78	4	6	99.5
Probe	511	**3471**	184	0	0	83.3
DoS	5299	1328	**223226**	0	0	97.1
U2R	168	20	0	**30**	10	13.2
R2L	14527	294	0	8	**1360**	08.4
% Correct	74,6	64.8	99.9	71.4	98.8	PRÉCISION : 92.71 %

Résultats Gagnant KDD Cup '99

Prédiction / Réel

	Normal	Probe	DoS	U2R	R2L	% Correct
Normal	**60182**	154	221	0	36	99.3
Probe	889	**2348**	928	0	1	56.4
DoS	723	195	**228935**	0	0	99.6
U2R	68	0	2	**0**	0	00.0
R2L	16229	9	81	0	**28**	00.2
% Correct	77,1	86.8	99.5	00.0	43.1	PRÉCISION : 93.72 %

Résultats article Staudemeyer (4 params)

Prédiction / Réel

	Normal	Probe	DoS	U2R	R2L	% Correct
Normal	**59868**	275	440	0	10	98.80
Probe	438	**3063**	665	0	0	73.52
DoS	5863	310	**223680**	0	0	97.31
U2R	69	0	1	**0**	0	00.0
R2L	15307	454	586	0	**0**	00.0
% Correct	73.41	74.67	99.24	00.0	00.0	PRÉCISION : 92.25 %

Résultats modèle entrainée GRU (4 params)

Prédiction / Réel

	Normal	Probe	DoS	U2R	R2L	% Correct
Normal	**59896**	242	436	2	17	98.84
Probe	91	**3359**	716	0	0	80.62
DoS	5293	937	**223623**	0	0	97.28
U2R	58	0	7	**2**	3	02.85
R2L	12386	78	109	0	**3774**	23.08
% Correct	77.06	72.76	99.43	50,00	99.47	PRÉCISION : 93.45 %

Résultats modèle entrainée LSTM (4 params)

Figure 36: Comparaison des résultats de la compétition, de l'article de Staudemeyer et des modèles entraînés lors du stage

L'ensemble du code réalisé pour ce projet ainsi que quelques instructions et précisions quant à ses utilisations peut être consulté à l'adresse suivante :

www.github.com/sylvainlapeyrade/RNN_Intrusion-Detection_Keras

7. Conclusion

Devant les résultats obtenus, nous sommes en droit de nous demander si les réseaux de neurones récurrents sont vraiment adaptés pour la détection d'intrusions dans le réseau. Certes, nous avons vu que nous pouvions approcher les 94 % de détections correctes sur le jeu de données KDD'99. Malgré cela, certains types d'attaque semblent hermétiques aux détections. Il semble donc apparaître que les RNN sont particulièrement efficaces pour la détection de séquences « mémorisables » (ce qui est logique puisque c'est leur raison d'être) , c'est-à-dire avec un schéma récurrent d'une donnée à l'autre (e.g. plusieurs données du même type à la suite).

Les Recurrent Neural Networks ne semblent donc pas être performants pour la reconnaissance de l'ensemble de l'éventail des attaques existantes. Ni même de certaines attaques décrites par l'ensemble des représentations possibles. Mais une telle technique ne semble tout simplement pas exister. Puisque les performances d'un algorithme d'apprentissage automatique est très dépendant des données avec lesquelles il fonctionne.

Que faut-il faire dans les cas où le jeu de données, ou du moins certaines données ne s'articulent pas sous forme de séquences mémorisables par un réseau récurrent ?

- Utiliser une méthode d'apprentissage (différente des RNN) spécifique *au jeu de données*. Pour KDD'99 les RNN semblent très adaptés en raison de la composition de celui-ci. En revanche, pour NSL KDD par exemple, d'autres algorithmes produiront de meilleurs résultats parce qu'ils sont plus adaptés.

- Utiliser une méthode d'apprentissage spécifique à chaque *type de données*. Dans le cas des types Normal et DoS pour KDD'99, les RNN semblent très appropriées, pour les types Probe, U2R et R2L beaucoup moins. D'autres algorithmes marcheraient peut-être beaucoup mieux sur ces derniers.

Si ces deux démarches paraissent logiques en théorie, elles n'amélioreront pas forcément la situation en pratique. En effet, même en prenant un algorithme différent pour chaque type et chaque jeu de données, rien ne garantit que le modèle soit pertinent hors du cadre du jeu de données avec lequel il a été entraîné. Nous avons constaté que la répartition et la distribution des données, d'un jeu à l'autre, peuvent radicalement changer. C'est également le cas pour les paramètres caractérisant les connexions et le type de connexions présents dans ces jeux. En pratique, on se rend compte que d'un jeu à l'autre, très souvent, on ne compare pas les mêmes éléments et que la comparaison n'a pas de sens.

Il n'existe effectivement pas une seule manière de représenter les connexions sous forme d'ensemble de données, comme pour les images représentées sous vecteurs de pixels, et chaque créateur de jeu de données choisit sa représentation. Ceci parce que les attaques sont très nombreuses et que les données nécessaires à la détection de chacune ne sont pas chaque fois les mêmes (alors qu'on représentera toujours les images avec des pixels pour reprendre l'analogie).

Ceci a pour conséquence d'avantager certains algorithmes sur certains jeux et types de données par rapport à d'autres. Il n'y a donc pas de méthode d'apprentissage prédominante comme ça peut être le cas dans d'autres domaines de l'apprentissage machine. En outre, une standardisation des jeux de données ne semble pas pour un futur proche tant la tâche semble complexe.

Une alternative pertinente pourrait être de faire apprendre au modèle un ensemble de données correspondant, voire un jeu créé *ad hoc,* au réseau cible de l'IDS. En effet, il n'est par exemple pas logique de faire apprendre un jeu de données construit sur un réseau privé pour ensuite tester ses

performances sur un réseau public. De la même manière, les types et la récurrence des attaques peuvent radicalement varier selon la catégorie de réseau. Il peut donc être intéressant, de faire apprendre, voire légèrement surapprendre, un système avec les données du réseau sur lequel il sera positionné en tant qu'IDS.

Cette approche nécessiterait toutefois de créer un jeu de données pour chaque réseau ou du moins pour chaque réseau similaire, ou alors de faire apprendre en temps réel le modèle en étiquetant systématiquement les données. La grande diversité des attaques peut rendre difficile cette identification, sachant que de nouveaux paradigmes apparaissent chaque jour. Mais cela semble être une solution des plus réalistes et efficace.

8. Bilan personnel

Cette expérience dans un institut de recherche en informatique fut particulièrement enrichissante personnellement. Les domaines du Machine Learning et de la recherche m'attirant beaucoup tous les deux, ce stage de Master 1 fut pour moi l'occasion parfaite pour développer mes connaissances dans ces domaines. Le fait d'évoluer dans une problématique de sécurité m'a également beaucoup plu puisque c'est, je crois, un enjeu très important des systèmes d'information et cela le restera très sûrement tout au long de leur existence.

N'ayant au début du stage que très peu d'expérience dans l'apprentissage automatique, j'ai pu consolider mes connaissances sur ses grands principes. J'ai de même développé de réelles connaissances sur les réseaux de neurones et d'autres algorithmes d'apprentissage que seule une mise en pratique comme celle-ci aurait pu m'apporter. Ces recherches ont en effet été l'occasion pour moi de créer concrètement plusieurs modèles d'apprentissage et pouvoir retranscrire tout l'aspect théorique que j'avais pu accumuler dans du code fonctionnant sur de vraies données.

Étant très intéressé par le domaine de la recherche, j'ai également pu découvrir les problématiques inhérentes au métier de chercheur. Ainsi que le fonctionnement et les démarches de ces derniers dans leur quotidien. J'ai aussi pu pratiquer, à mon échelle, des travaux de recherches par le biais de documentation et de lecture sur l'état de l'art. De même, je me suis rendu compte de l'esprit critique et la curiosité nécessaire à ce milieu pour pouvoir apporter une plus-value sur ce qui existe déjà.

Enfin, l'approfondissement de mes compétences dans la détection d'intrusion me fut très bénéfique dans la mesure où je suis sûr qu'elles me serviront dans le futur tant les enjeux de sécurité sont cruciaux. En outre, la mise en application des concepts de réseaux de neurones, et d'apprentissage automatique en général, à la sureté dans le réseau fut très passionnante. Cela m'a effectivement permis de comprendre à la fois les problématiques de l'autoapprentissage et de la reconnaissance d'attaque.

Je considère donc que ce stage a parfaitement répondu aux attentes que j'avais concernant les activités que j'ai citées et je suis très satisfait de son déroulement. Je remercie une nouvelle fois tous ceux qui l'ont rendu possible et qui m'ont accompagné pendant toute sa durée.

Table des illustrations

Bibliographie

[1] R. C. Staudemeyer, « Applying long short-term memory recurrent neural networks to intrusion detection », *South African Computer Journal*, vol. 56, nº 1, juill. 2015.

[2] « KDD-CUP-99 Task Description ». [En ligne]. Disponible sur: http://kdd.ics.uci.edu/databases/kddcup99/task.html.

[3] A. Oglaza, « État de l'art sur l'apprentissage automatique ». .

[4] S. J. Russell, P. Norvig, et E. Davis, *Artificial intelligence: a modern approach*, 3rd ed. Upper Saddle River: Prentice Hall, 2010.

[5] F. Chollet, *Deep learning with Python*. Shelter Island, New York: Manning Publications Co, 2018.

[6] R. Caruana, S. Lawrence, et C. Lee Giles, *Overfitting in Neural Nets: Backpropagation, Conjugate Gradient, and Early Stopping.*, vol. 13. 2000.

[7] J. Chung, C. Gulcehre, K. Cho, et Y. Bengio, « Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling », *arXiv:1412.3555 [cs]*, déc. 2014.

[8] S. Hochreiter et J. Schmidhuber, « Long Short-Term Memory », *Neural Computation*, vol. 9, nº 8, p. 1735-1780, nov. 1997.

[9] « Understanding LSTM Networks -- colah's blog ». [En ligne]. Disponible sur: https://colah.github.io/posts/2015-08-Understanding-LSTMs/. [Consulté le: 22-juill-2019].

[10] F. A. Gers, J. Schmidhuber, et F. Cummins, « Learning to Forget: Continual Prediction with LSTM », *Neural Computation*, vol. 12, nº 10, p. 2451-2471, oct. 2000.

[11] K. Cho *et al.*, « Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation », *arXiv:1406.1078 [cs, stat]*, juin 2014.

[12] « StandardScaler — scikit-learn 0.21.2 documentation ». [En ligne]. Disponible sur: https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html.

[13] « OneHotEncoder — scikit-learn 0.21.2 documentation ». [En ligne]. Disponible sur: https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OneHotEncoder.html.

[14] A. Gullì et S. Pal, *Deep learning with Keras: implement neural networks with Keras on Theano and TensorFlow*. Birmingham Mumbai: Packt, 2017.

[15] J. Nievergelt, « Exhaustive Search, Combinatorial Optimization and Enumeration: Exploring the Potential of Raw Computing Power », in *SOFSEM 2000: Theory and Practice of Informatics*, vol. 1963, V. Hlaváč, K. G. Jeffery, et J. Wiedermann, Éd. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, p. 18-35.