Informatique - Semestre 8

Rapport de stage

Apprentissage des séries temporelles appliqué à la détection d'intrusions dans les réseaux

---

UNIVERSITÉ DU QUÉBEC EN OUTAOUAIS

Auteurs :
Marais Lucas

Directeur de stage :
Herbreteau Frédéric
Maître de stage :
Nguena Timo Omer

Table des matières

1 Résumé

Ce stage de recherche a été réalisé au Canada à l'Université du Québec en Outaouais. C'est lorsque Mr Nguena Timo Omer a proposé un sujet liant intelligence artificielle et cybersécurité, que j'ai répondu présent à sa demande souhaitant poursuivre mes études dans cette direction. Le sujet initial du stage proposé par Mr Nguena Timo était donc la **Détection de défauts dans des systèmes de sécurité et de contrôle d'accès évolutifs**. Cependant, ce sujet a été amené à être légèrement modifié lors du stage. En effet, l'objectif principal du stage était d'utiliser l'outil `Security Onion` afin d'automatiser le tri des alertes permettant de détecter les vraies alertes des fausses. Cet outil permet en effet de lever des alertes parmi les traces entrantes et sortantes d'un réseau. Il est ensuite nécessaire de trier ces alertes pour différencier les vrais positifs des faux. C'est normalement un humain qui réalise cette tâche. Le sujet proposé par Mr Nguena Timo consistait donc à automatiser ce tri avec l'intelligence artificielle et les réseaux de neurones.

Cependant, l'installation de l'outil `Security Onion` n'a pas été un succès pour diverses raisons (trop grande demande en mémoire RAM, nécessité d'une configuration des interfaces réseau très spécifiques et non adaptées à l'utilisation "normale" du pc en parallèle, ...). Nous avons donc décidé par la suite de remplacer le travail de `Security Onion` par des simples automates [1] : un automate de génération de traces, un automate qui lève des alertes parmi ces traces et un automate de tri des alertes (vrai positif vs faux positif).

Les traces réseau constituants une série chronologique, c'est un réseau de neurones récurrents qu'il a été nécessaire de développer. Dans ce stage, nous nous sommes intéressés aux `Long Short-Term Memory (LSTM)`, des réseaux de neurones récurrents complexes permettant un apprentissage précis.

La première partie du stage consistait ainsi à se familiariser avec les `LSTM` afin d'en apprendre les caractéristiques. Ensuite, il a été nécessaire de lier automates et réseaux de neurones. C'est pourquoi nous avons fait apprendre le comportement d'automates à des `LSTM`. Une hypothèse était que le réseau de neurones apprenait le bon nombre d'états de l'automate et non un nombre supérieur ou inférieur qui correspondrait à un autre automate équivalent. Pour cela, nous avons du loguer les valeurs des états et poids du réseau afin d'analyser les variations de valeurs lors des différentes prédiction.

Une fois ces observations et expérimentations effectuées, il a été possible d'implémenter les différents automates permettant de remplacer `Security Onion` et d'implémenter le réseau de neurones permettant l'apprentissage.

L'ensemble du code implémenté lors de ce stage est disponible sur mon Github[1].

Le travail réalisé dans ce stage peut devenir une source d'inspiration pour la réalisation de logiciels de tri d'alertes pouvant être déployés dans des entreprises, réseaux privés, etc.

1. Un automate fini correspond à un graphe orienté, dans lequel certains des nœuds (états) sont distingués et marqués comme initial ou finaux et dans lequel les arcs (transitions) sont étiquetés par des symboles de l'alphabet d'entrée de l'automate

2 Introduction

Ce stage est un stage de recherche réalisé au sein d'une université québécoise à Saint-Jérôme. L'université du Québec en Outaouais est une université membre du réseau des universités du Québec, elle possède plusieurs campus dont un à Saint-Jérôme. Mr Nguena Timo Omer est professeur au laboratoire d'informatique du campus de Saint-Jérôme. De plus, il étudie et propose des méthodes formelles pour améliorer la sureté et la sécurité des systèmes informatisés. Le sujet du stage, reliant recherche en intelligence artificielle et sécurité informatique rejoint donc parfaitement son domaine de recherche. Alors que ces approches sont plus basées sur la logique et les mathématiques discrètes, ce sujet de stage est une manière d'élargir les domaines afin de croiser les recherches et d'obtenir de nouveaux résultats prometteurs.

3 Les missions techniques

Le stage rassemble deux grands objectifs du monde de la recherche en informatique : l'apprentissage de séries chronologiques et des automates par les réseaux de neurones et l'automatisation de la sécurité informatique.

Les séries chronologiques se différencient des séries habituelles par la dépendance de ces éléments. En effet, chacun des éléments constituants la série est en dépendance avec l'élément précédent. Ainsi, lors de l'apprentissage, il est nécessaire de prendre en compte ce phénomène pour permettre une bonne prédiction. Les réseaux de neurones récurrents sont des réseaux de neurones permettant, avec une mémoire de la cellule, d'effectuer cet apprentissage.

L'apprentissage des automates par les réseaux de neurones est un concept développé par de nombreux chercheurs, cependant, peu sont ceux qui ont obtenu des résultats représentatifs sur l'équivalence entre l'automate appris par le réseau de neurones et celui ayant généré les données d'apprentissage. Certains articles abordent cette notion de façon mathématique et complexe [2]. Dans ce stage, une analyse expérimentale sera développée.

L'automatisation des comportements humains est un sujet d'actualité. Cependant, lorsqu'il est appliqué au tri de traces, il est nécessaire d'imiter un comportement d'analyse logique parmi plusieurs éléments d'une série chronologique. La difficulté réside donc dans le fait d'enregistrer toutes les étapes de réflexion d'un humain lors du tri (cette partie n'est pas abordée dans ce stage). Ensuite, il est possible d'implémenter un automate permettant d'imiter ces comportements puis de le faire apprendre à un réseau de neurones. Ainsi, un modèle de tri automatique des alertes réseaux pourrait être développé afin d'automatiser des tâches humaines dans le monde de la cybersécurité.

4 Réalisation du projet

4.1 Séries temporelles

Les **séries temporelles**, aussi appelées **séries chronologiques** [3], sont des suites de valeurs numériques qui dépendent du temps. C'est-à-dire que la valeur numérique à un instant t dépend de la valeur numérique à l'instant $t-1$. Ainsi, les séries temporelles constituent des tendances d'évolutions du système qu'elles symbolisent.

Par exemple, la température, en un lieu précis, à un instant t, dépend de la température à l'instant $t-1$. Alors que nous avons la suite des valeurs (en degrés) $(12, 14, 16, 16, 15)$, il est ainsi fortement possible de prévoir que la température à l'instant suivant sera inférieure ou égale à 15, comme cette variable était sur une pente descendante à cet instant. Cependant, il n'y a pas toujours uniquement le temps qui intervient.

On distingue ainsi les **séries temporelles à une variable**, et celles à **plusieurs variables**.

Une série chronologique à une seule variable est une suite de valeurs dont l'évolution ne dépend que d'un paramètre en plus du temps. Par exemple une suite de valeurs en degré qui résulte de l'analyse de la température en fonction du temps. Au contraire, une série temporelle à plusieurs variables est une suite de valeurs qui dépendent d'au moins deux paramètres en plus du temps. Par exemple, une suite de valeurs en degré en fonction de la température, de la pression, de la vitesse du vent, des précipitations ...

Ces séries temporelles peuvent ainsi constituer des sources d'analyse profondes pour prédire des phénomènes complexes. L'intelligence artificielle (IA) cherche à prédire ces variables futures en fonction des précédentes

observations. Cet enjeu de l'IA est un enjeu important pour notamment **automatiser les comportements humains** d'analyse des séries temporelles.

4.2 Réseaux de neurones récurrents

Afin de réaliser l'apprentissage des séries temporelles et d'automatiser la prédiction des valeurs suivantes, il est nécessaire que cet **apprentissage garde en mémoire** les états des cellules du réseau de neurones entre deux instants t et $t-1$. Les **réseaux de neurones récurrents (Recurent Neural Network - RNN)** permettent la mise en place de cette mémoire au sein des cellules du réseau.

Pour bien comprendre le fonctionnement des réseaux de neurones récurrents, faisons une analogie avec les **réseaux de neurones à propagation avant (Feedforward Neural Network- FNN)**.

Un FNN permet une sortie Y, souvent un vecteur, à partir d'une entrée X qui peut être un vecteur, une image, un texte, etc. Ainsi, pour chaque entrée, le réseau choisit une sortie en fonction de ce qu'il a déjà appris. Cette sortie ne dépend donc que de l'entrée.

La figure 2 permet de schématiser un FNN :

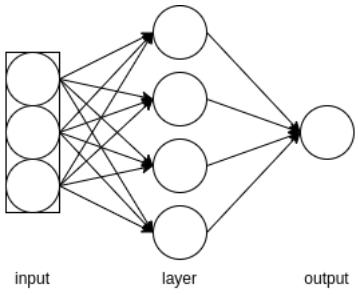


Figure 2 – Représentation d'un FNN

Afin de simplifier la schématisation du modèle, les vecteurs, et neurones de chaque couche, sont regroupés pour ne former qu'une entité, et le réseau est mis à la vertical, comme le montre la figure 3.

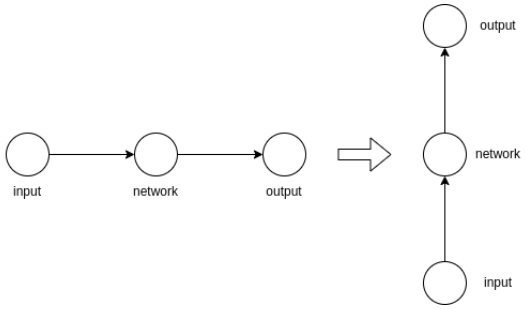


Figure 3 – Schématisation simplifiée d'un FNN

Pour les séries temporelles, comme une prédiction dépend des valeurs précédentes, il est nécessaire pour le réseau de garder en mémoire ce qu'il vient d'apprendre. **Ainsi une sortie h_t à l'instant t dépend de l'entrée x_t mais également de la sortie h_{t-1} à l'étape précédente**. Cela est possible grâce à **la mémoire** C que le réseau conserve et améliore entre deux époques.

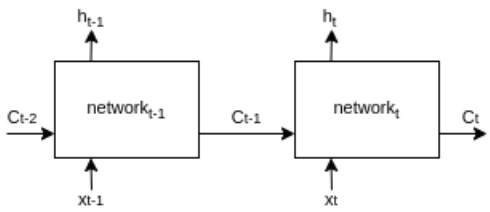


Figure 4 – Fonctionnement d'un RNN entre deux instants t et $t-1$

Par exemple, si on envoie "comment" dans le réseau, alors celui-ci va retourner "ça" qui est généralement le mot qui suit "comment" dans les phrases des français (la phrase "comment ça va ?" est en effet l'une des plus fréquentes). Ensuite si on envoie "ça" dans le réseau, alors il est difficile pour le réseau de comprendre ce qu'il faut prédire comme il ne connaît pas le mot qui était en entrée avant "ça". C'est pourquoi il faut garder en mémoire les états des cellules, c'est ce que fait le réseau récurrent. Le schéma en figure 5 illustre cet exemple.

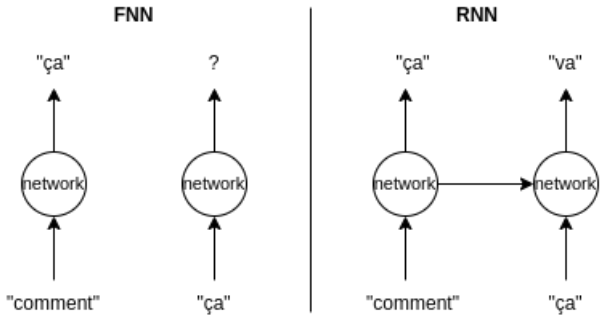
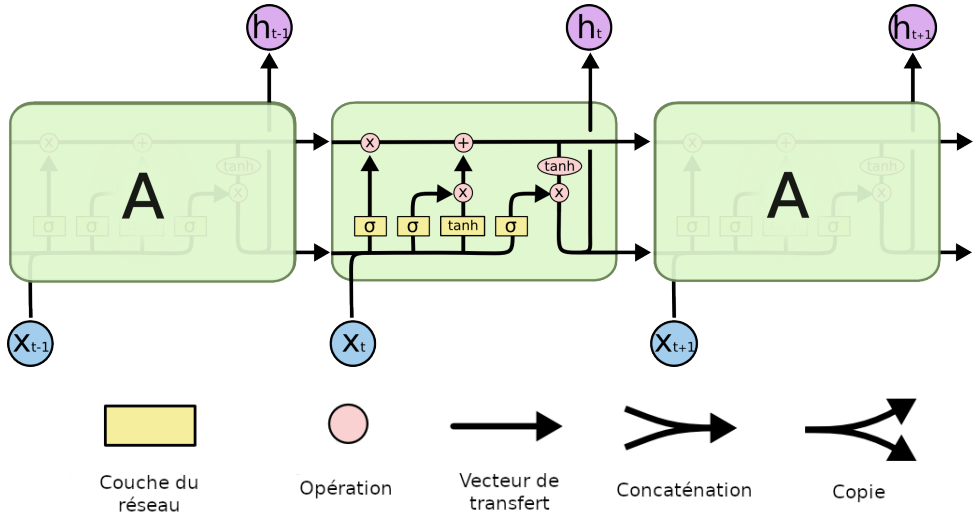


Figure 5 – FNN vs RNN pour l'apprentissage des séries temporelles

4.3 Long Short-Term Memory

Le RNN, comme expliqué précédemment, permet de conserver en mémoire l'apprentissage qu'il réalise à un instant t, pour prédire la sortie à l'instant $t+1$. Cependant, utilisant la fonction d'activation `tanh`, les valeurs sont comprises dans l'intervalle $[-1, 1]$. Le gradient calculé lors de chaque nouvelle époque est donc très faible et la modification des poids du réseau lors de la descente de gradient entraîne des multiplications par des nombres très proches de 0. Cette modification des poids n'a donc aucun effet après quelques époques, le **réseau n'apprend plus**. On appelle ce phénomène le problème du ***vanish gradient***. Afin d'améliorer cet apprentissage, tout en conservant les propriétés des RNN, il est nécessaire d'utiliser un **Long Short-Term Memory (LSTM)** [4] qui est un réseau de neurones récurrents plus complexe.

Alors que le RNN classique ne réalise qu'une opération sur les valeurs dans les neurones, le LSTM réalise plusieurs opérations avec un but précis. De la même manière qu'un RNN en figure 4, il est possible de schématiser un LSTM comme le montre la figure 6 ci-dessous.



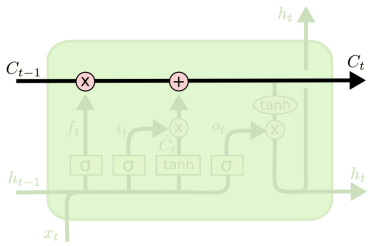
Source: http ://colah.github.io/posts/2015-08-Understanding-LSTMs
Figure 6 – Représentation de la structure d'un LSTM

Sur la figure 6, chaque cellule verte représente l'ensemble des neurones du réseau à différents instants. On a x **les entrées** aux différents instants et h **les sorties**. Les sorties h sont ensuite dirigées vers une couche exerçant une fonction d'activation permettant la mise en forme de la prédiction de la sortie.

Le LSTM, afin de ne pas rencontrer de soucis de *vanish gradient*, modifie toujours ses poids au cours de son

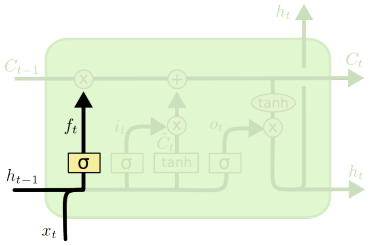
apprentissage. Pour cela, il possède plusieurs traitements au sein de ces cellules [2] afin d'**ajouter de l'information** dans les poids (input gate), **oublier de l'information** (foreget gate), **stocker cette information** (mémoire de la cellule) et **prédire la sortie** (output gate).

La mémoire de la cellule est stockée dans un vecteur C qui est transféré entre deux instants. Ainsi, lors de la prédiction à l'instant t, l'ensemble des neurones envoient la sortie h_t à la couche suivante du réseau mais également un vecteur de mémoire C_t à eux-mêmes pour prédire la sortie à l'instant $t+1$ en fonction de la prédiction à l'instant t.



Source: http ://colah.github.io/posts/2015-08-Understanding-LSTMs
Figure 7 – Stockage de la mémoire dans le vecteur C

Le LSTM, par son fonctionnement, doit être capable du supprimer les informations inutiles stockées dans les poids. Ainsi il possède dans chaque cellule un traitement permettant d'oublier. Ce traitement est réalisé par l'intermédiaire de la **forget gate** (figure 8.

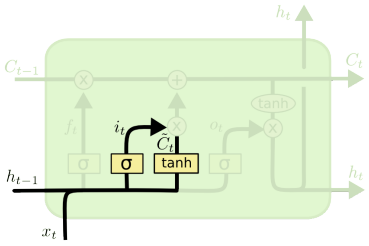


$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

Source: http ://colah.github.io/posts/2015-08-Understanding-LSTMs
Figure 8 – Forget Gate dans une cellule LSTM

La forget gate permet l'initialisation de la valeur de f_t en fonction de l'entrée x_t, de la sortie h_{t-1}, et des poids W_f et biais b_f du réseaux liés à cette forget gate. Le tout est passé dans une fonction `sigmoid` pour obtenir une valeur comprise dans l'intervalle $[0, 1]$ et quantifier la quantité d'information à oublier. C'est-à-dire que plus la valeur de f sera proche de 0 et plus l'information contenu dans C_{t-1} sera oubliée avant d'être transmise à C_t.

Le réseau de neurones doit également être en mesure d'ajouter de l'information dans le vecteur de la mémoire. L'**input gate** est la capacité du réseau à ajouter de l'information. En effet, la cellule réalise deux traitements dans la mémoire.



$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

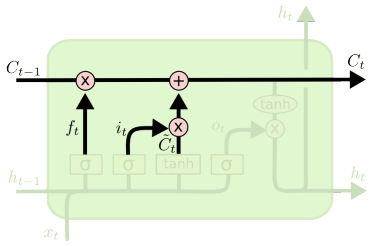
Source: http ://colah.github.io/posts/2015-08-Understanding-LSTMs
Figure 9 – Input Gate dans une cellule LSTM

La figure 9 illustre cet ajout d'information. La valeur $\hat{C}_t$ permet, en fonction de la sortie h_{t-1} et de l'entrée x_t et des paramètres de la couche liés à l'input gate W_i et b_i, de calculer les informations à ajouter avec la fonction

2. Dans l'ensemble du rapport, le terme *cellule* caractérise l'ensemble des neurones cachés constituant la couche LSTM

`tanh`. Cette valeur est multipliée par la valeur de i_t qui, grâce à la fonction `sigmoid`, permet de quantifier cette information à ajouter.

Afin de **modifier l'état de la cellule et donc la mémoire des neurones**, la cellule utilise la valeur de la forget gate f_t, celle de l'input gate i_t et son état à l'instant $t-1$. Pour cela, le vecteur de la mémoire C_{t-1} est multipliée par f_t pour oublier un certain nombre d'informations, puis on lui ajoute la valeur de sortie de l'input gate à savoir $i_t * \hat{C}_t$ pour ajouter de l'information.

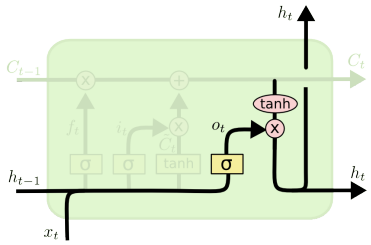


$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Source: http ://colah.github.io/posts/2015-08-Understanding-LSTMs
Figure 10 – Modification du vecteur C

Ce nouvel état C_t sera ensuite utilisé à l'instant $t+1$ pour permettre le transfert de mémoire, et est aussi utilisé pour prédire la sortie de l'instant t avec l'output gate.

L'**output gate** permet de prédire la sortie h_t en fonction de l'entrée x_t, de la sortie h_{t-1} et de la mémoire de la cellule C_t.



$$o_t = \sigma\left(W_o\ [h_{t-1}, x_t]\ +\ b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

Source: http ://colah.github.io/posts/2015-08-Understanding-LSTMs
Figure 11 – Ouput Gate dans une cellule LSTM

La figure 11 illustre ce traitement. Alors qu'un neurone classique ne réaliserait que le calcul de la valeur de o_t en fonction de l'entrée x_t, ce calcul est ici réalisé également en fonction de la sortie h_{t-1}. Il est donc nécessaire pour le neurone de connaître les traitements à appliquer sur cette valeur o_t en fonction des traitements effectués lors de la précédente prédiction. C'est pour cela que la fonction `tanh` est utilisée. Elle permet de sélectionner des valeurs dans la mémoire du neurone C_t et d'appliquer des traitements par multiplication avec la valeur de o_t. Le réseau prédit ainsi la sortie h_t qu'il garde en mémoire pour la prédiction suivante, et qu'il envoie à la couche suivante du réseau.

Comme dans tout réseau de neurones, il est nécessaire de **modifier les valeurs des poids et biais** des cellules pendant l'apprentissage. Ce sont ces poids et biais qui permettent de prédire précisément la sortie avec un minimum de perte.

La **descente de gradient (backpropagation)** [5] est l'étape de l'apprentissage entre deux prédictions qui permet de faire évoluer ces valeurs. En fin de prédiction, l'erreur entre la sortie prédite et la sortie attendue est calculée. Le gradient correspond à la dérivée de cette erreur par rapport à la sortie : $\Delta E = \frac{dE}{dh_t}$

La sortie dépendant de nombreux paramètres, il est nécessaire de modifier tous les poids permettant le calcul de ces paramètres. Ainsi, on réalise la descente de gradient sur chaque poids du réseau.

Afin de réaliser la descente de gradient, l'erreur $\Delta E = \frac{dE}{dh_t}$ est calculée en sortie du réseau. Ensuite, on calcule le gradient associé à chaque poids et biais du réseau. Chaque valeur de poids et biais est ensuite modifiée par soustraction de la valeur précédente par le produit du `learning rate` et du gradient associé.
Chaque poids est donc modifié par la formule suivante : $W = W - \alpha.\frac{dE}{dW}$ avec α le pas.

Par exemple, pour la descente de gradient liée à l'output gate, on calcule d'abord le gradient associé. Le gradient étant la dérivée de l'erreur par rapport à la sortie, celui lié à l'ouput gate est calculé avec la formule :

$$\frac{dE}{do_t} = \frac{dE}{dh_t}.\frac{dh_t}{do_t} = \Delta E.\frac{dh_t}{do_t}$$

Or $h_t = o_t.tanh(C_t)$
Donc

$$\frac{dE}{do_t} = \Delta E.tanh(C_t)$$

Il est ensuite nécessaire de calculer les gradients associés à chaque poids et biais de l'output gate. On note $z_o = W_{xo}.x_t + W_{ho}.h_{t-1} + b_o$, on obtient :

$$\frac{dE}{dW_{xo}} = \frac{dE}{do_t}.\frac{do_t}{dW_{xo}} = \Delta E.tanh(C_t).\sigma(z_o).(1-\sigma(z_o)).x_t$$
$$\frac{dE}{dW_{ho}} = \frac{dE}{do_t}.\frac{do_t}{dW_{ho}} = \Delta E.tanh(C_t).\sigma(z_o).(1-\sigma(z_o)).h_{t-1}$$
$$\frac{dE}{db_o} = \frac{dE}{do_t}.\frac{do_t}{db_o} = \Delta E.tanh(C_t).\sigma(z_o).(1-\sigma(z_o))$$

Ensuite, la backpropagation peut être effectuée.
Les valeurs des poids et biais liés à l'output gate sont ensuite modifiées :

$$b_o = b_o - \Delta E.tanh(C_t).\sigma(z_o).(1-\sigma(z_o)).\alpha$$
$$W_o = W_o - \Big(\frac{dE}{dW_{xo}} + \frac{dE}{dW_{ho}}\Big).\alpha = W_o - \Delta E.tanh(C_t).\sigma(z_o).(1-\sigma(z_o)).\big(x_t + h_{t-1}\big).\alpha$$

Cette opération est réalisée sur l'ensemble des poids du réseau liés à chaque gate du réseau de neurones.

4.4 LSTM et cybersécurité

Dans l'intégralité du projet, les réseaux de neurones sont implémentés en utilisant la librairie de machine learning `tensorflow` et notamment son API `keras` [6] qui implémente de nombreuses fonctions utiles à l'apprentissage automatique.

Dans cette partie sont développées différentes études permettant la compréhension et l'implémentation du projet final.

4.4.1 LSTM pour la classification des traces réseau

Une première approche aux LSTM est la **classification** de données textuelles telles que les traces réseau. En effet, un réseau internet reçoit de nombreuses traces qui sont des logs des requêtes entrantes et sortantes du réseau. Ces requêtes peuvent être classées en catégories, et on retrouve notamment des catégories liées aux cyber-attaques. L'objectif du réseau de neurones est donc d'**apprendre à classer les traces** afin de pouvoir **détecter les potentielles intrusions** sur le réseau.

L'étude réalisée pour illustrer la classification des traces est basée sur le travail de Supriya Shende [7]. Dans cette étude, la base de données utilisée est la base de données `NSL-KDD` qui est une base de données contenant des traces avec des attaques classées selon des familles (`guess_passwd`, `spy`, etc) et des traces `normales`, c'est-à-dire des traces n'étant pas des attaques. Il est alors possible de réaliser deux types de classification sur ces traces : la **classification binaire** et la **classification multiclasses**.

Le fichier `attack_classification.py`, présent dans le répertoire `Intrusion-Detection` du projet permet l'apprentissage de la classification binaire. C'est-à-dire qu'on classe les traces de la base de données en deux catégories : **attaque** et **normale**. Pour ce faire, les données subissent quelques traitements.

Après avoir chargé les données de la base `NSL-KDD`, il est nécessaire de les **reformater** afin d'adapter le format à la première couche du réseau qui est la couche LSTM. En effet, le réseau développé pour cette étude est un simple réseau constitué d'une **couche LSTM** puis d'une **couche** `Dense` afin de mettre en forme les données de sortie pour pouvoir les exploiter efficacement. De plus, les familles de traces qui définissent les catégories d'attaques ou non sont regroupées en 5 sous-familles et **la classification est multiclasse avec 5 classes** (normal, r2l, u2r probe, dos) qui correspondent à des classes d'attaques [8].

Le réseau est donc définit par le code suivant :

```
model = Sequential()
model.add(LSTM(128, dropout = 0.2, recurrent_dropout = 0.2, input_shape = (120, 1)))
model.add(Dense(5, activation = 'softmax'))
```

On définit ici 128 neurones cachés pour la couche LSTM, et on met en place un `dropout` de 0.2. Ce-dernier permet un meilleur apprentissage avec moins de surentraînement (overfitting). En effet la valeur de `dropout` fixe la quantité d'informations oubliées lors du calcul de i_t, celle de `recurrent_dropout` fixe la quantité d'informations à choisir lors du calcul de h_t sur l'ouput gate. Plus le dropout sera élevé et plus la quantité d'informations oubliées sera importante.
Sur la dernière couche, la fonction d'activation choisie est la fonction `softmax` car elle permet le calcul de **probabilités**. Ainsi, nous avons 5 neurones sur la dernière couche, et les valeurs ressorties par chacun des 5 neurones permettent de donner la probabilité que la trace en entrée appartienne à chacune des 5 classes de traces définies.

Le **modèle est compilé** avec la méthode `compile`. La fonction de calcul de perte est `categorical_crossentropy` afin de calculer la perte liée à la classification en différentes catégories.

Ensuite, le **modèle est entraîné** avec la méthode `fit`. Les données d'entraînement sont divisées en deux catégories afin de pouvoir **entraîner le modèle avec la première partie et de l'évaluer à chaque époque sur la deuxième partie des données**. Ainsi la perte est calculée à chaque étape et on peut extraire les données d'entraînement comme les valeurs de perte, de précision, qui sont stockées dans la variable `history`.

Le réseau est entraîné sur 20 époques :

```
history = model.fit(X_train, y_train,  = (X_test, y_test), batch_size = 32, epochs = 20)
```

Les données de test sont ensuite utilisées pour évaluer le modèle après entraînement. On peut donc évaluer le modèle et **vérifier les valeurs de perte et précision pour valider l'apprentissage**.

On obtient sur cette étude une perte de 2% et une précision de 99.24%. Le modèle est donc validé.

4.4.2 LSTM pour la prédiction de données textuelles

L'étude réalisée pour illustrer la **prédiction de données textuelles** est l'implémentation d'un générateur automatique du mot permettant de compléter une suite de mots pour former une phrase ayant un sens logique. La prédiction de ce mot est un cas typique d'utilisation des séries temporelles. En effet, **il est nécessaire de connaître les entrées et sorties précédentes afin de pouvoir prédire le dernier mot**.

Cette étude est basée sur le travail réalisé par IG Tech Team [9] adapté à nos besoins et le code se trouve dans le répertoire `Word-Prediction` du projet. Le fichier `text_prediction.py` réalise l'apprentissage du réseau de neurones, et le fichier `next_word.py` permet de tester le modèle sur des entrées choisies.

Afin d'entraîner le réseau, la base de données utilisée est le livre *Le tour du monde en 80 jours*. Cependant, pour permettre à l'apprentissage d'être plus rapide, ces données sont limitées à un certain nombre de lignes. Ici, les 10000 premières lignes du livre ont été utilisées. Avant l'apprentissage, **il est nécessaire de réaliser un reformatage de ces données** afin de les convertir en valeurs numériques pour pouvoir les passer à la couche d'entrée du réseau.

4.4.2.1 Embedding

Lorsqu'on travaille avec des données textuelles dans les réseaux de neurones, il est nécessaire de les **transformer en données numériques**. Il est possible de réaliser cela en créant des variables `dummy features` et en créant une table contenant des 1 et des 0 en fonction des `dummy features` contenues dans le message. Cependant, pour un vocabulaire de grande taille, cela n'est **pas réalisable de comparer et jouer avec des tables de grandes tailles**. C'est pourquoi l'`embedding` est important. Cela permet d'associer des **vecteurs de taille fixe à chaque groupe de mots** et donc d'éviter la création d'une table de correspondance de taille trop importante.

Notre réseau de neurones possède ainsi une couche `embedding` comme première couche. Cette couche permet à partir des valeurs encodées selon l'encodage choisi, de transformer les valeurs associées aux tokens (ici les mots) en vecteurs de valeurs.

L'**encodage** choisi ici est un encodage classique utilisant des valeurs numériques pour remplacer les mots. Ainsi, chaque mot constituant le vocabulaire est transformé en un **token**. L'ensemble des tokens ainsi formé permet d'associer une valeur numérique à chaque mot du vocabulaire défini. L'encodage dépend donc du nombre de mots dans le vocabulaire. L'**encodage des valeurs textuelles permet ainsi de transformer chaque**

groupe de mots en un vecteur des encodages.
Par exemple, si on a le groupe de mot *"Ceci est une intelligence artificielle"*, alors l'encodage sera de la forme $[12, 2, 1, 45, 34]$ en supposant que le mot *Ceci* est encodé avec la valeur 12, etc.

Afin d'obtenir des vecteurs que l'on peut passer en entrée de notre réseau de neurones, il est alors nécessaire de tous leur donner la même taille. On ajoute donc du `padding` aux vecteurs (séquences de mots) les plus courts de telle sorte à ce que tous les vecteurs possèdent la même longueur, c'est-à-dire la longueur du vecteur le plus long.

Ensuite, la **couche embedding** permet de transformer les mots en vecteurs denses de taille fixe. Ces vecteurs contiennent des valeurs réelles (plutôt que des 1 et 0 pour une analyse basique avec la fonction `get_dummies` de `keras` par exemple). Ainsi une séquence de mots est, par le passage dans la couche `embedding` transformée en un vecteur de vecteurs denses. Et chaque vecteur dense représente un mot initialement présent dans la séquence.

La couche d'embedding fonctionne de façon similaire à une table de comparaison, **les mots y sont les clés et les vecteurs dense les valeurs**. Ainsi, lorsqu'un mot (son encodage numérique), ou une séquence de mots, entre dans la couche, alors elle permet de renvoyer en sortie le vecteur dense qui encode ce mot, ou les vecteurs dense qui encodent les mots de cette séquence.

Dans `keras`, la couche d'embedding se définit avec 3 paramètres :

— `input_dim` : le nombre de mots dans le vocabulaire
— `output_dim` : la taille du vecteur dense (vecteur représentant un mot)
— `input_length` : la taille du vecteur d'entrée (séquence de mots)

Ainsi l'utilisation de l'embedding avec des données textuelles se fait comme suit :

— On encode les groupes de mots avec un encodage numérique
— On complète les vecteurs les plus courts avec du padding si nécessaire
— On choisit la taille de notre vocabulaire, ainsi que la taille des vecteurs en sortie de la couche Embedding (la taille d'un vecteur pour désigner un mot)

Les **poids du réseau**, une fois entraînés, correspondent ainsi à l'**encodage des mots du vocabulaire**. Il est donc facile pour le réseau, lorsqu'on lui passe un nouveau groupe de mots en entrée, de pouvoir renvoyer la sortie correspondante en analysant la composition du message. En effet, la couche embedding prend le message (le vecteur des vecteurs des mots du message), et renvoie un tensor des vecteurs des poids associés aux mots en entrées. Ainsi ce tensor est prêt à être envoyé à la couche LSTM et encode bien les mots présents dans le message.

Par exemple, si on a les poids de la couche embedding entrainée pour prédire des nombres de 0 à 3, avec des tailles de vecteurs (`output_dim`) de 4. Alors on a cela :

```
1    [[-0.04333381, -0.02326865, -0.00812379,  0.02167496],  --> 0
2     [ 0.04502351,  0.00151128,  0.01764284, -0.0089057 ],  --> 1
3     [-0.04007018,  0.02874336,  0.02772436,  0.00842067],  --> 2
4     [ 0.00512743,  0.03695237, -0.02774147, -0.03748262]]  --> 3
```

La ligne 1 encodant le chiffre 0, la 2 le chiffre 1, etc. Ainsi si on envoie le groupe de mots [1, 2], alors la couche embedding renvoie le tensor :

```
1    [[ 0.04502351,  0.00151128,  0.01764284, -0.0089057 ],  --> 1
2     [-0.04007018,  0.02874336,  0.02772436,  0.00842067]]  --> 2
```

La couche LSTM va donc travailler avec ce tensor et le réseau connaît à quel mot ces valeurs correspondent.

4.4.2.2 Implémentation du réseau de neurones

Après avoir réalisé le reformatage des données en entrée du réseau, ce-dernier est implémenté avec le code suivant :

```
1 model = Sequential()
2 model.add(Embedding(vocab_size, 10, input_length = WINDOW_SIZE))
3 model.add(LSTM(1000))
4 model.add(Dense(1000, activation = "relu"))
5 model.add(Dense(vocab_size, activation = "softmax"))
```

La première couche est donc la **couche d'embedding** formant pour chaque mot des vecteurs denses de taille 10.

On définit également une variable `WINDOW_SIZE` qui est la taille de la fenêtre analysée par le réseau de neurones. Ainsi, le réseau prend `WINDOW_SIZE` mots en entrée afin de prédire le mot suivant.

En fin de réseau, on retrouve la **couche** `Dense` avec la fonction d'activation `softmax`. Elle permet de renvoyer un **vecteur des probabilités** pour le mot prédit. En effet, ce vecteur fait la taille du vocabulaire, et chacune de ses composantes est la probabilité que le mot prédit soit le mot associé dans le vocabulaire. Le mot à prédire est donc le mot avec la plus forte probabilité. Il est également possible de prédire plusieurs mots en prenant par exemple plusieurs mots de fortes probabilités et de renvoyer une liste de ces mots.

Afin de tester le modèle, lors de son apprentissage, **le modèle (les poids et biais) est sauvegardé** dans un fichier `next_word.h5`. Ainsi, à chaque époque, si la perte est plus faible qu'à l'époque précédente, alors on sauvegarde le modèle. En fin d'apprentissage, le meilleur modèle, c'est-à-dire **la meilleure configuration des poids** pour prédire, est stocké dans le fichier et il est possible de le charger pour pouvoir l'utiliser. Lors de cet apprentissage, il est nécessaire de faire attention au nombre d'époques choisi. En effet, le **sur-entraînement** est rapidement atteint si le texte en entrée est un texte trop court et que le nombre d'époques est trop important. Dans ce cas, le réseau apprend simplement le texte par coeur et ne sait pas s'adapter à des phrases qui ne sont pas présentes dans le texte d'entraînement par exemple.
Le fichier `next_word.py` permet de charger le modèle et de le tester. Ainsi, si on lance l'exécution du fichier, alors on donne `WINDOW_SIZE` mots et le modèle prédit le mot suivant en fonction de ce qu'il a appris. Il est nécessaire d'utiliser des mots présents dans le vocabulaire appris par le réseau, sinon il ne les reconnaît pas et ne peut pas prédire.

4.4.3 LSTM pour l'apprentissage de machines à états finis

L'objectif principal du stage concerne la **mise en place d'un réseau de neurones pour l'apprentissage d'une machine à états finis**. C'est-à-dire que l'on souhaite faire apprendre le comportement humain à un réseau de neurones, et ce comportement humain est ici défini par le comportement d'une machine à états finis.

La première idée était d'utiliser `Security-Onion` qui est un analyseur de traces réseau permettant la mise en avant d'alertes. Un humain, par observation des alertes peut les trier en plusieurs catégories (les vraies alertes et les fausses alertes). L'objectif était donc de faire apprendre ce comportement à un réseau de neurones. Toutefois, la mise en place de `Security-Onion` a posé de nombreux problèmes comme expliqué en partie 1, le projet a donc été de créer notre propre générateur de traces et d'alertes.

L'objectif est donc d'implémenter un **automate qui génère des traces aléatoirement**, un **automate pour analyser ces traces** et accepter celles qui sont des **alertes**. Ces deux automates remplacent le comportement de `Security-Onion`. Enfin, le dernier automate à implémenter est un **automate de décision** si la trace alerte donnée en entrée est une **réelle alerte** ou non. L'objectif est donc de faire apprendre le comportement de ce-dernier automate à un réseau de neurones afin d'automatiser au maximum la tâche effectuée. De plus, il serait possible de faire fonctionner ces automates en parallèle pour pipeliner le modèle et qu'il tourne en fond tel un logiciel de sécurité informatique.

Avant de commencer l'implémentation de ces divers automates et réseau de neurones, il est nécessaire de vérifier le bon **apprentissage d'un automate par un réseau de neurones**. Il est donc nécessaire d'implémenter une **machine à états finis (FSM)** qui génère une série temporelle en fonction des entrées choisies pour passer d'états en états. Afin de vérifier que le réseau apprend bien l'automate, chaque **transition** est caractérisée par un **état précédent** et un **état suivant**, ainsi que d'une **entrée** et d'une **sortie**. De telle sorte, la série ainsi générée est constituée de **couples (entrée, sortie)** qui forment les traces constituant la série. On cherche donc à vérifier que si l'on donne en entrée une suite d'entrées, alors le réseau de neurones est capable de prédire la suite des sorties correspondantes.

Une telle machine à états comprenant des couples (entrée, sortie) sur chaque transition est appelée **Finite State Transducer (FST)** [10].

Ce FST est définit par un 6-tuple $(Q, \Sigma, \Gamma, I, F, \delta)$ avec :
— $Q = \{q_0, q_1, q_2, q_3\}$ ensemble fini d'états
— $\Sigma = \{a, b\}$ alphabet d'entrée (ensemble fini)
— $\Gamma = \{0, 1\}$ alphabet de sortie (ensemble fini)
— $I = \{q_0\}$ ensemble d'états initiaux (ici un unique)
— $F = \emptyset$ ensemble d'états finaux (ici aucun)

— $\delta \subseteq Q \times \Sigma \times \Gamma \times Q$ définissant les transitions

Le FST implémenté est le suivant :

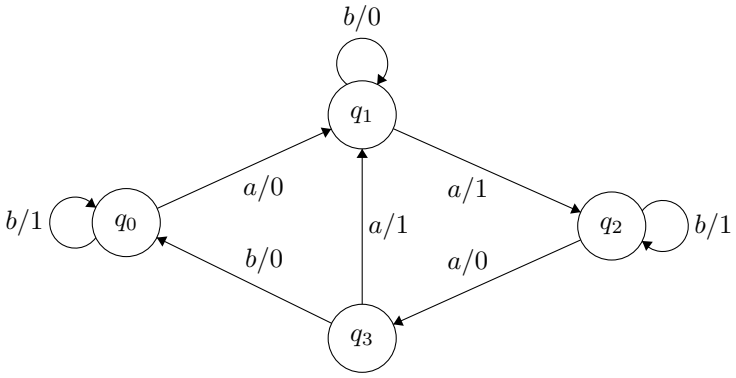


Figure 12 – Exemple de Finite State Transducer

Le fichier `finite_state_transducer.py` en annexe 7.1 définit la classe `FiniteStateTransducer` permettant la création de cet automate. Ainsi, un FST possède des attributs `states, inputs, outputs, transitions, initial_state, final_states` et des méthodes d'ajout et de suppression pour modifier la structure générale du FST (`add_sate`, `remove_state`, `change_initial_state`, etc).

On définit donc, dans le fichier `log_generation.py`, les états, entrées, sorties et l'état initial. Ensuite, afin de reproduire le FST en figure 12, les transitions sont définies comme suit :

```
transitions={'q0':{'a':{'output':'0','next_state':'q1'},'b':{'output':'1','next_state':'q0'}},
             'q1':{'a':{'output':'1','next_state':'q2'},'b':{'output':'0','next_state':'q1'}},
             'q2':{'a':{'output':'0','next_state':'q3'},'b':{'output':'1','next_state':'q2'}},
             'q3':{'a':{'output':'1','next_state':'q1'},'b':{'output':'0','next_state':'q0'}}}
```

Le FST ainsi implémenté correspond à celui en figure 12. Par exemple, depuis l'état q_0, l'entrée a est associée à la sortie 0 et permet la transition vers l'état q_1.

Afin de **générer des traces**, on code un algorithme qui à partir de l'état initial définit un chemin de longueur donnée (aléatoire si non donnée en paramètre) en sauvegardant les états, entrées et sorties constituant ce parcours entre les états.

```
def get_log(state_machine, length=random.randint(LOG_LENGTH_MIN, LOG_LENGTH_MAX)) :
    log = []
    state = state_machine.initial_state
    for i in range(length) :
        input = random.choice(state_machine.inputs)
        output = state_machine.transitions[state][input]['output']
        log.append((state, input, output))
        state = state_machine.transitions[state][input]['next_state']
        if state in state_machine.final_states :
            break
    return log

def logs_generator(state_machine, n, length=random.randint(LOG_LENGTH_MIN, LOG_LENGTH_MAX)) :
    logs = []
    for i in range(n) :
        logs.append(get_log(state_machine, length))
    return logs
```

Ainsi on obtient une série temporelle constituée d'une suite de couples (entrée, sortie). Après remise en forme on obtient deux suites, celle des entrées puis celle des sorties. La base de données des traces ainsi formée est ensuite utilisée par le fichier `neural_network.py` qui permet la définition et l'apprentissage du réseau.

Afin de passer des données dans bon format au réseau, et comme expliqué en partie 4.4.2.1, il est nécessaire de réaliser un traitement pour les reformater. On définit donc dans le fichier `utils.py` un **encodeur** `one-hot`. L'encodage `one-hot` consiste à encoder les variables sur `n` bits selon les `n` classes de valeurs possibles. Ainsi une variable est un mot sur `n` bits dont un seul possède la valeur 1.
Par exemple, pour encoder un mot constitué des lettres **a** ou **b**, alors on encode chaque lettre du mot avec 2

bits. Si la lettre est un **a**, elle est encodée `[1,0]` ; pour un **b**, l'encodage est `[0,1]`.

Voici un exemple d'encodeur one-hot développé et utilisé dans le projet :

```
#Get one hot encoding of a given input
def one_hot_encoder(X, classes):
    n_classes = len(classes)
    size = len(X)
    encoded = np.zeros((size, n_classes))
    for i in range(size):
        encoded[i][get_index(X[i], classes)] = 1
    return encoded.tolist()

#Get one hot encoding for all the inputs
def get_one_hots(sequence, classes):
    inputs = []
    for i in range(len(sequence)):
        inputs.append(one_hot_encoder(sequence[i], classes))
    return np.array(inputs)
```

Cet encodeur est appliqué à chacune des entrées, c'est-à-dire sur chacune des entrées d'une trace (chaque lettre). Ainsi la série temporelle des entrées subit les changements suivants :

```
[['a','a','b','a','b'],                          [[[1,0],[1,0],[0,1],[1,0],[0,1]],
 ['b','a','a','a','b'],          -------->        [[0,1],[1,0],[1,0],[1,0],[0,1]],
 ['a','b','b','b','a']]                           [[1,0],[0,1],[0,1],[0,1],[1,0]]]
```

En effet, le dictionnaire de mots possible ($'a','b','0','1'$) étant très petit, et les mots n'étant pas des réelles chaînes de caractères, il n'est pas cohérent d'intégrer une couche d'embedding qui n'est pas entraînée à encoder ce genre de mot. L'encodage `one-hot` prend, au contraire, tout son sens vis à vis de la quantité d'informations à modéliser très faible (peu de mots dans le dictionnaire). Face à des réelles traces réseaux qui sont des données textuelles, il serait alors cohérent d'utiliser une couche d'embedding. Après avoir divisé les données encodées en deux parties (entraînement et validation), on peut créer le réseau, le compiler puis lancer l'apprentissage avec la commande `$ python3 neural_network.py <NB_HIDDEN_NEURONS> <NB_EPOCHS>`.

```
#Create the model
model = Sequential()
model.add(RNN(LSTMCell(NB_HIDDEN_NEURONS), input_shape=(LOG_SIZE,2), return_sequences = True))
model.add(Dense(2, activation='softmax'))

#Create callback
checkpoint = ModelCheckpoint('saved_model.h5', monitor='val_loss', verbose=1, save_best_only=
    True)

#Compile the model
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
history = model.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=NB_EPOCHS,
    batch_size=50, callbacks=[checkpoint])
```

La couche de sortie est une couche `Dense` constituée de deux neurones et utilisant la fonction d'activation `softmax`. Ainsi, le vecteur en sortie est un tensor de dimensions `(LOG_SIZE, 2)` avec `LOG_SIZE` la taille des séries temporelles d'entrée et de sortie. La dimension 2 correspond aux probabilités que la valeur de sortie prédite soit un **0** ou un **1**. Ainsi, le tensor de sortie donne les probabilités de prédiction d'un **0** ou d'un **1** pour chaque trace contenue dans la série temporelle en entrée (ici des **a** ou des **b**).

Le réseau de neurones est entraîné et la meilleure configuration permettant la perte la plus faible est sauvegardée dans le fichier `saved_model.h5`. Le modèle ainsi défini, est ensuite chargé par le fichier `test_prediction.py` qui demande en entrée une suite de traces (ici des **a** et **b**) et prédit la sortie correspondante.

Une première réflexion concerne le **nombre de neurones cachés nécessaires à l'apprentissage de l'automate à 4 états**. Pour cela, on réalise un apprentissage avec un unique neurone caché, puis 2 etc. Initialement, on générait les logs à chaque nouvel apprentissage. Ainsi la base de données d'apprentissage n'était jamais la même. Cependant, lors de cette expérimentation, cela a posé quelques problèmes. En effet, comme les données d'apprentissage ne sont jamais exactement les mêmes, alors **l'apprentissage est plus ou moins performant** suivant l'ordre des traces dans les données. Ainsi, avec un même nombre d'époques, les mêmes données, et le même nombre de neurones cachés, des résultats différents étaient obtenus. La solution a donc été de **fixer une base de données**. Pour cela, les logs générés par le fichier `log_generation.py` sont stockés dans le fichier `data.txt` puis chargés lors de l'apprentissage par le fichier `neural_network.py`. Après expérimentation, on obtient le tableau de comparaison suivant pour 100 époques d'apprentissage :

nb neurons	loss	accuracy	val_loss	val_accuracy
1	0.5866	0.6254	0.5884	0.6241
2	0.0193	0.9996	0.0188	0.9995
3	1.4735e-05	1.0000	1.4797e-05	1.0000
4	6.5338e-04	1.0000	6.0655e-04	1.0000
5	9.8681e-04	1.0000	9.4669e-04	1.0000

On remarque que l'apprentissage avec **un seul neurone caché ne montre pas de bons résultats**. De plus, la valeur de la précision d'environ 60% est constante durant tout l'apprentissage, ainsi le réseau n'apprend rien et n'apprendra jamais avec un seul neurone caché. Au contraire, **2 neurones cachés au sein du LSTM suffisent à l'apprentissage** de l'automate par le réseau de neurones. On remarque également un surentraînement pour les apprentissages avec plus de 2 neurones cachés. Cependant, le sur-apprentissage n'est pas dérangeant dans ce cas là car l'automate à apprendre est un automate fixe, donc on ne souhaite pas que le réseau de neurones puisse réaliser des prédictions pour des traces non générées par l'automate.
On fixe donc le nombre de neurones cachés à 2 pour la suite du projet.

La seconde réflexion concerne le bon apprentissage de l'automate par le LSTM. En effet, il est cohérent de se **demander si le réseau de neurone à bien appris l'automate tel qu'il est ou s'il a appris un autre automate équivalent**. L'hypothèse que nous avons émise est que le **réseau de neurone apprend le bon nombre d'états**, ainsi il serait cohérent que **2 neurones cachés suffisent à l'apprentissage d'un automate à 4 états (2 bits permettent de coder 4 valeurs)**. Pour expérimenter cela, il est nécessaire de loguer les états du vecteur de la mémoire de la cellule LSTM (vecteur `C`). L'API tensorflow étant définie pour réaliser des actions entre chaque batch (ensemble de séquences d'entrées), ou entre chaque époque, il est difficile de loguer les états au sein même de l'apprentissage sur une séquence. Pour ce faire, il a donc été nécessaire de **modifier une partie de l'API** de tensorflow. En effet, au lieu de passer par une cellule LSTM, qui a un comportement propre difficile à modifier, pour créer le réseau de neurones, on utilise une cellule `RNN` que l'on configure comme étant une `LSTMCell`. Il est ainsi possible d'aller modifier le comportement de l'API directement dans le code source afin de loguer les valeurs du vecteur de la mémoire. On ajoute donc les lignes suivantes avant le retour de valeurs dans la méthode `call` de la classe `LSTMCell` :

```
f = open("c.txt", "w")
tf.print(c, output_stream="file:///home/lucas/Documents/Cours/Internship/src/Finite-State-Machine/c.txt", summarize=-1)
f.close()
```

Lors de l'apprentissage et lors des prédictions, à chaque appel d'une nouvelle entrée, le réseau appelle la méthode `call` de la cellule concernée. Ainsi, on retrouve dans le fichier `c.txt` toutes les valeurs du vecteur de mémoire `C` lors de chaque prédiction. Afin d'analyser le bon apprentissage de l'automate par le réseau de neurones, on exécute le fichier `test_prediction.py` qui charge le modèle sauvegardé et attend une séquence d'entrées (suite de **a** et **b**) afin de prédire la sortie (suite de **0** et **1**). Lors de la prédiction, en fonction de la séquence donnée, à chaque appel du réseau sur les entrées (**a** et **b**), les valeurs du vecteur `C` sont enregistrées dans le fichier `c.txt`. Il est alors possible d'**analyser les variations de ces valeurs et d'identifier le nombre d'états appris par le réseau**.

Les données en annexe 7.2 correspondent à une séquence d'entrées, la séquence de sorties prédite par le réseau, ainsi que les valeurs des états de la mémoire pour chaque entrée. L'automate utilisé est celui décrit en partie 4.4.3. Ainsi la première entrée est composée d'une suite de 6 **b** afin de boucler sur le premier état q_0, puis un **a** qui permet la transition sur le deuxième état q_1, puis une suite de 6 **b** pour boucler sur cet état, puis un **a** pour aller sur le troisième état q_2. Une nouvelle suite de 6 **b** permet de boucler sur cet état q_2, avant qu'un **a** permette la transition vers le quatrième état q_3. Ensuite on utilise une nouvelle fois des suites de **b** suivies d'un **a** permettant de revenir boucler sur les deux premiers états de l'automate afin d'observer si les valeurs du vecteur de la mémoire sont les mêmes, permettant de confirmer que le réseau ait appris le bon nombre d'états.

On remarque facilement que les 6 premières valeurs sont très proches, le réseau comprend donc que les entrées permettait de boucler sur le premier état. Ensuite, le **a** entraîne une modification de ces valeurs. On passe en effet d'environ $[-0.2, -0.08]$ à environ $[0.8, -0.5]$, deux vecteurs dont la différence est importante. Le réseau à donc bien réalisé le changement d'état à la suite de l'entrée **a**. Une fois de plus, les valeurs restent les mêmes (à quelques modifications près) avant de changer à la suite du **a** suivant. Les valeurs du vecteur `C` varient pour atteindre environ $[-0.1, -0.8]$, le réseau est donc passé dans un troisième état. Après avoir bouclé, lorsqu'un nouveau **a** est passé en entrée, les valeurs atteignent $[0.4, -1.6]$ ce qui correspond bien à un nouvel état (le quatrième), puis le **b** suivant permet de revenir à l'état initial car les valeurs atteintes sont de l'ordre de $[-0.2, -0.1]$ ce qui correspond au tout premier état identifié. L'état suivant atteint par le réseau possède des valeurs proches du deuxième état également $[0.7, -0.4]$. Ainsi **le réseau à bel et bien appris 4 états et**

réalise les mêmes transitions sur ces états que l'automate.

La même expérience à été réalisée avec uniquement des **b** en entrée et le réseau boucle bien sur un unique état dont les valeurs sont comprises aux alentours de $[-0.2, -0.09]$ ce qui **correspond bien au premier état identifié par la première expérience**.

Lors de la troisième expérience on envoie uniquement des **a** au réseau de façon à observer s'il réalise bien une boucle entre les états q_1, q_2 et q_3 après être passé par l'état q_0. Dans les valeurs du vecteur de la mémoire, **on identifie les mêmes valeurs d'états** que lors de la première expérience et on remarque que le réseau réalise bien cette boucle.

A ce stade du projet, nous pouvons donc établir une conjecture (aucune démonstration mais uniquement des expérimentations) que **le réseau de neurones apprend le bon nombre d'états et qu'il réalise les mêmes transitions que l'automate**. Nous considérons donc notre hypothèse comme validée, **2 neurones cachés suffisent pour l'apprentissage d'un automate à 4 états** car le réseau apprend le bon nombre d'états et que 2 bits permettent d'encoder ces 4 états.

La même expérience à été réalisée avec 3 neurones cachés, sur le même automate, et on remarque des résultats légèrement différents. En effet, bien que les valeurs soit assez proches, elles évoluent (augmentent ou diminuent) toujours de façon monotone au fur et à mesure que les entrées bouclent sur un état par exemple. De plus, il semblerait que lorsqu'un **a** est passé en entrée pour transiter entre 2 états notamment, alors le réseau a appris un état intermédiaire et c'est l'enchaînement **a,b** qui permet de transiter d'un état à un autre.

Cependant, les valeurs forment (comme pour 2 neurones) des intervalles dans lesquels elles évoluent. On peut ainsi facilement distinguer les états et les différencier. De plus, ces intervalles sont bien distincts, c'est-à-dire qu'une valeur ne varie pas jusqu'à atteindre un nouvel intervalle caractérisant déjà un état.

Dans la troisième expérience, le réseau réalise pourtant bien la boucle d'état en état sans passer par un état intermédiaire. De plus, les valeurs ne sont pas les mêmes que celles obtenues lors des expériences 1 et 2. Il semblerait donc qu'ajouter des neurones cachés au réseau le pousse à être **plus précis** (plus de valeurs pour encoder les états) et donc à **développer un automate appris plus détaillé**. Cependant, les prédictions restent parfaitement justes.

Pour la suite du projet et la confection des analyseurs de traces, nous utiliserons donc **le plus petit nombre de neurones cachés possible**. Si c'est un automate à 4 états, alors nous utiliserons 2 neurones cachés, si il y a 8 états, alors nous utiliserons 3 neurones cachés, etc. Cela permet d'**éviter l'apprentissage de nombreux paramètres du réseau**, alors que le LSTM est tout à fait en mesure d'apprendre le comportement de l'automate avec un faible nombre de neurones cachés.

4.4.4 Automatisation de la sécurité informatique

Comme expliqué précédemment, l'objectif principal du stage est **la confection d'un analyseur de traces réseaux**. L'utilisation de `Security Onion` n'ayant pas été possible, nous avons fait le choix d'établir nos propres générateurs de traces et d'alertes. Ainsi, un automate génère des traces réseaux (ici des suites d'entrée $a_0, a_1, a_2, ...$), un automate identifie certaines traces comme étant des alertes. Pour cela, il possède un état acceptant qui permet d'identifier comme alertes les traces terminant dans cet état. Un automate a également été réalisé pour permettre d'identifier les vraies attaques parmi toutes les alertes levées. Cet automate possède lui aussi un état acceptant permettant de valider comme *attaque* les traces terminant dans cet état.

Pour des raisons de simplification et parce qu'il n'est pas utile de complexifier les automates tant que les traces manipulées ne sont pas des réelles traces, les automates déployés sont constitués d'un faible nombre d'états, et l'alphabet d'entrée est petit. Ainsi la confection des automates a été simplifiée, mais le fonctionnement reste le même que s'il y avait des centaines d'états et entrées possibles. Chacun des automates a été implémenté avec la classe `FiniteStateTransducer` disponible en annexe 7.1.

Le fichier `generator.py` du répertoire `Final-Project` permet donc la génération de traces. Le fichier `alerts_launcher.py` permet de lever des alertes parmi ces traces. Pour cela la méthodes `is_accepted` permet, pour chaque trace, de vérifier si cette trace est acceptée par l'automate correspondant :

```
1 def is_accepted(state_machine, log) :
2     state = state_machine.initial_state
3     for i in range(len(log)):
```

```
        input = log[i]
        if input in state_machine.transitions[state] :
            state = state_machine.transitions[state][input]['next_state']
        else:
            return False
    return state_machine.is_final_state(state)
```

Ensuite, le fichier `alerts_sorter.py` permet d'accepter uniquement les alertes étant des réelles attaques.

L'objectif a ensuite été d'implémenter un réseau de neurones qui apprend le comportement du dernier automate. Pour cela, les traces générées par le générateur sont récupérées puis triées en alertes. On crée donc une base de données à partir des alertes que l'on passe dans l'automate de tri des réelles alertes. Ainsi on obtient une base de données constituée d'alertes avec un flag 1 si ce sont des vraies alertes, 0 sinon.

On peut alors construire le model :

```
#Model LSTM
model = Sequential()
model.add(LSTM(3, input_shape=(LOG_LENGTH, gen.NB_INPUTS)))
model.add(Dense(2, activation='softmax'))

checkpoint = ModelCheckpoint('model.h5', monitor='val_loss', verbose=1, save_best_only=True)
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
history = model.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=100, batch_size
    =20, callbacks=[checkpoint])
```

Un unique neurone caché est utilisé ici car l'automate de décision est un automate à 2 états, il faut donc au minimum 1 bit pour encoder ces 2 états, et nous choisissons de prendre le minimum de neurones cachés possible. Le réseau est entraîné sur 150 époques et on obtient une précision de 94% pour une perte de 11%. Ces **résultats ne sont pas les meilleurs possibles car cela dépend énormément de la base de données créée par le générateur**, si elle est significative ou non. Cependant, ces résultats restent satisfaisants vis à vis des automates créés et des résultats attendus.

Ce travail pourrait être repris et amélioré afin de développer une réelle automatisation du tri des alertes dans un réseau privé ou public.

5 Conclusion

Ce **stage de recherche** avait pour objectif de lier **intelligence artificielle et cybersécurité**. Les réseaux de neurones récurrents et notamment les LSTM, par apprentissage des machines à états finis et de leur comportement, permettent cette automatisation. De plus, par expérimentation, il a été conjecturé que **l'automate appris par le réseau de neurones correspond bien à l'automate d'origine**, dont le réseau cherchait à imiter le comportement. Cette expérimentation conforte l'idée que les réseaux de neurones récurrents, par leur spécificité technique d'apprentissage des séries temporelles, sont **en capacité d'imiter un comportement humain de tri des alertes réseaux**.

Bien que l'outil `Security Onion` n'ait pas pu être exploité, les solutions trouvées pour le remplacer ont permis de développer un **modèle d'automatisation de tri des alertes réseaux**. Les objectifs principaux du stage et la finalité du projet a donc été réalisée, malgré le fait que le chemin pour y arriver ait été légèrement modifié au cours du stage.

Ce modèle ainsi développé pourra par la suite être utilisé et développé plus en détail afin de réaliser divers projets autour de l'automatisation de la sécurité informatique. Le tri automatique des alertes réseaux levées par un outil du type `Security Onion` ou encore `Wazuh` pourrait par exemple être développé dans un logiciel de sécurité par machine learning.

6 Remerciements

En premier lieu, je tiens à remercier Mr Omer Nguena Timo. En tant que maître de stage dans la recherche, il m'a beaucoup appris n'ayant jamais réalisé de projet de recherche auparavant. Le sujet très intéressant qu'il a proposé m'a permis de développer mes connaissances en intelligence artificielle et cybersécurité, domaines dans lesquels je souhaite poursuivre mes études..

Je remercie également l'équipe pédagogique de l'UQO qui m'a accompagné dans mes démarches. Arriver dans un nouveau pays, loin de son pays d'origine, n'est jamais quelque chose de facile. Ils ont permis de réaliser toutes les démarches de façon simple et efficace. De plus, ce stage est partiellement financé par la subvention RGPIN07248-2020 du CRSNG (Conseil de recherches en sciences naturelles et en génie du Canada), et je les en remercie.

Je désire aussi remercier l'équipe pédagogique de l'ENSEIRB-MATMECA et notamment Mr Frederic Herbreteau. Ils ont activement participé à l'organisation de cette mobilité internationale et de ce stage.

Pour finir je voudrais remercier Mr Antoine Rollet, professeur à l'ENSEIRB-MATMECA, qui m'a fait part de cette offre de stage, et sans qui je n'aurais peut être pas eu cette opportunité.

Références

[1] Code source du projet
https://github.com/luks-m/Intrusion-detection-with-machine-learning.git

[2] Distance and Equivalence between Finite State Machines and Recurrent Neural Networks : Computational results
https://arxiv.org/pdf/2004.00478.pdf

[3] Séries Chronologiques
https://www.canarie.ca/fr/nuage/propulseurs/catalogue/prevision-de-series-chronologiques-par-apprenti

[4] Christopher Olah
http://colah.github.io/posts/2015-08-Understanding-LSTMs

[5] Derivation of Back propagation through time
https://www.geeksforgeeks.org/lstm-derivation-of-back-propagation-through-time/?ref=lbp

[6] API Keras
https://keras.io/api

[7] Supriya Shende
https://www.researchgate.net/publication/342754573_Long_Short-Term_Memory_LSTM_Deep_Learning_Method_for_Intrusion_Detection_in_Network_Security

[8] Statistical Analysis Driven Optimized Deep Learning System for Intrusion Detection
https://www.researchgate.net/publication/327110465_Statistical_Analysis_Driven_Optimized_Deep_Learning_System_for_Intrusion_Detection

[9] IG Tech Team
https://ishwargautam.blogspot.com/2021/07/next-word-prediction-using-lstm.html

[10] Wikipédia
https://en.wikipedia.org/wiki/Finite-state_transducer

7 Annexe

7.1 finite_state_transducer.py

```
class FiniteStateTransducer :
    def __init__(self, states, inputs, outputs, transitions, initial_state, final_states) :
        self.states = states
        self.inputs = inputs
        self.outputs = outputs
        self.transitions = transitions
        self.initial_state = initial_state
        self.final_states = final_states


    def __str__(self) :
        return "FiniteStateTransducer(\n\
            states={}\n\
            inputs={}\n\
            outputs={}\n\
            transitions={}\n\
            initial_state={}\n\
            final_states={})".format(self.states, self.inputs, self.outputs, self.transitions,
     self.initial_state, self.final_states)


    def add_state(self, state) :
        if state not in self.states :
            self.states.append(state)


    def remove_state(self, state) :
        if state not in self.states :
            raise ValueError("Le state {} n'existe pas".format(state))
        else :
            self.states.remove(state)


    def add_input(self, input) :
        if input not in self.inputs :
            self.inputs.append(input)


    def remove_input(self, input) :
        if input not in self.inputs :
            raise ValueError("L'input {} n'existe pas".format(input))
        else :
            self.inputs.remove(input)


    def add_output(self, output) :
        if output not in self.outputs :
            self.outputs.append(output)


    def remove_output(self, output) :
        if output not in self.outputs :
            raise ValueError("L'output {} n'existe pas".format(output))
        else :
            self.outputs.remove(output)


    def add_transition(self, transition) :
        #transition should be in this format {'init_state': 'q0', 'input': 'a', 'output': '0',
     'next_state': 'q1'}
        if not 'init_state' in transition or not 'input' in transition or not 'output' in
    transition or not 'next_state' in transition :
            raise ValueError("La transition doit etre de la forme {'init_state': 'q0', 'input
    ': 'a', 'output': '0', 'next_state': 'q1'}")
        init_state = transition['init_state']
        input = transition['input']
        output = transition['output']
        next_state = transition['next_state']
        if init_state not in self.states :
            raise ValueError("Le state {} n'existe pas".format(transition['init_state']))
        elif input not in self.inputs :
```

```
68              raise ValueError("L'input {} n'existe pas".format(transition['input']))
69          elif output not in self.outputs :
70              raise ValueError("L'output {} n'existe pas".format(transition['output']))
71          elif next_state not in self.states :
72              raise ValueError("Le state {} n'existe pas".format(transition['next_state']))
73          else :
74              if init_state not in self.transitions : #check if a transition from this state
    does not already exist
75                  self.transitions[init_state] = {input: {'output': output, 'next_state':
    next_state}}
76                  return
77              if input not in self.transitions[init_state] : #check if a transition from this
    state with this input does not already exist
78                  self.transitions[init_state][input] = {'output': output, 'next_state':
    next_state}
79                  return
80              else :
81                  raise ValueError("Une transition existe deja depuis le state {} avec l'input
    {}".format(init_state, input))
82
83
84      def remove_transition(self, transition) :
85          #transition should be in this format {'init_state': 'q0', 'input': 'a', 'output': '0',
     'next_state': 'q1'}
86          if not 'init_state' in transition or not 'input' in transition or not 'output' in
    transition or not 'next_state' in transition :
87              raise ValueError("La transition doit etre de la forme {'init_state': 'q0', 'input
    ': 'a', 'output': '0', 'next_state': 'q1'}")
88          init_state = transition['init_state']
89          input = transition['input']
90          output = transition['output']
91          next_state = transition['next_state']
92          if init_state not in self.transitions :
93              raise ValueError("Aucune transition existe depuis le state {}".format(init_state))
94          if input not in self.transitions[init_state] :
95              raise ValueError("Aucune transition existe depuis le state {} avec l'input {}".
    format(init_state, input))
96          else :
97              del self.transitions[init_state][input]
98
99
100     def change_initial_state(self, state) :
101         if state not in self.states :
102             raise ValueError("Le state {} n'existe pas".format(state))
103         else :
104             self.initial_state = state
105
106
107     def add_final_state(self, state) :
108         if state not in self.states :
109             raise ValueError("Le state {} n'existe pas".format(state))
110         else :
111             self.final_states.append(state)
112
113
114     def remove_final_state(self, state) :
115         if state not in self.states :
116             raise ValueError("Le state {} n'existe pas".format(state))
117         elif state not in self.final_states :
118             raise ValueError("Le state {} n'est pas un etat final".format(state))
119         else :
120             self.final_states.remove(state)
121
122
123     def get_current_inputs(self, state) :
124         inputs = []
125         for input in self.inputs :
126             if input in self.transitions[state] :
127                 inputs.append(input)
128         return inputs
129
130
131     def is_final_state(self, state) :
132         return state in self.final_states
```

7.2 Expérimentation

Analyse des valeurs du vecteur mémoire avec 2 neurones cachés :

```
Input :
  b  b  b  b  b  b  a  b  b  b  b  b  b  a  b  b  b  b  b  b  a  b  b  b  b  a  b  b  b  b
Output :
[[1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0]]

C values :
b  ---  [[-0.277715981 -0.0609863]]
b  ---  [[-0.121935904 -0.071513474]]
b  ---  [[-0.261825264 -0.0893834308]]
b  ---  [[-0.169909298 -0.0886192247]]
b  ---  [[-0.243698359 -0.0959995911]]
b  ---  [[-0.192543894 -0.0943125784]]
a  ---  [[0.933422506 -0.674193621]]
b  ---  [[0.804650784 -0.128138065]]
b  ---  [[0.713337362 -0.210288972]]
b  ---  [[0.643343806 -0.304734826]]
b  ---  [[0.592847526 -0.419017732]]
b  ---  [[0.559230626 -0.527945518]]
b  ---  [[0.535945296 -0.584827244]]
a  ---  [[-0.0509327054 -0.970136881]]
b  ---  [[-0.0731718242 -0.865513742]]
b  ---  [[-0.0968646 -0.869320691]]
b  ---  [[-0.121167511 -0.873101115]]
b  ---  [[-0.146041363 -0.87654382]]
b  ---  [[-0.17144914 -0.879531801]]
b  ---  [[-0.197325334 -0.882010162]]
a  ---  [[0.419803202 -1.6367085]]
b  ---  [[0.190387115 -0.101187453]]
b  ---  [[-0.0320316702 -0.14535746]]
b  ---  [[-0.23271811 -0.159350067]]
b  ---  [[-0.249439582 -0.145445377]]
a  ---  [[0.92402786 -0.796385229]]
b  ---  [[0.788541377 -0.118969671]]
b  ---  [[0.69182688 -0.201235697]]
b  ---  [[0.617096364 -0.295161188]]
b  ---  [[0.562655747 -0.412141979]]

Input :
  b  b  b  b  b  b  b  b  b  b  b  b  b  b  b  b  b  b  b  b  b  b  b  b  b  b  b  b  b  b
Output :
[[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]]

C values :
b  ---  [[-0.277715981 -0.0609863]]
b  ---  [[-0.121935904 -0.071513474]]
b  ---  [[-0.261825264 -0.0893834308]]
b  ---  [[-0.169909298 -0.0886192247]]
b  ---  [[-0.243698359 -0.0959995911]]
b  ---  [[-0.192543894 -0.0943125784]]
b  ---  [[-0.231949121 -0.0977174193]]
b  ---  [[-0.204008952 -0.0964590237]]
b  ---  [[-0.225079939 -0.0981134623]]
b  ---  [[-0.209960073 -0.0973289758]]
b  ---  [[-0.221227616 -0.0981585309]]
b  ---  [[-0.213085026 -0.0977021]]
b  ---  [[-0.219111219 -0.0981268361]]
b  ---  [[-0.214737087 -0.0978700519]]
b  ---  [[-0.217961013 -0.0980906114]]
b  ---  [[-0.215614453 -0.0979487821]]
b  ---  [[-0.217339605 -0.098064445]]
b  ---  [[-0.216081619 -0.0979870111]]
b  ---  [[-0.217005119 -0.0980480835]]
b  ---  [[-0.216330871 -0.098006025]]
b  ---  [[-0.216825321 -0.0980383679]]
b  ---  [[-0.216463983 -0.0980156735]]
b  ---  [[-0.216728851 -0.0980329365]]
b  ---  [[-0.2165353 -0.0980206877]]
b  ---  [[-0.216677055 -0.0980298817]]
b  ---  [[-0.216573402 -0.0980233327]]
b  ---  [[-0.216649294 -0.0980282053]]
b  ---  [[-0.216593802 -0.098024711]]
b  ---  [[-0.216634393 -0.0980273]]
b  ---  [[-0.216604665 -0.0980254]]
```

```
Input :
  a  a  a  a  a  a  a  a  a  a  a  a  a  a  a  a  a  a  a  a  a  a  a  a  a  a  a  a
Output :
[[0, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0]]

C values :
a  ---  [[0.940785944 -0.464669168]]
a  ---  [[0.940785944 -0.464669168]]
a  ---  [[-0.149279535 -0.84660244]]
a  ---  [[0.385762602 -1.54715455]]
a  ---  [[0.758346498 -1.50694132]]
a  ---  [[-0.049001798 -0.905963421]]
a  ---  [[0.385538876 -1.55473256]]
a  ---  [[0.730348468 -1.54166734]]
a  ---  [[-0.0487197489 -0.928736866]]
a  ---  [[0.378766388 -1.56768167]]
a  ---  [[0.737681091 -1.55534124]]
a  ---  [[-0.0494268052 -0.923735499]]
a  ---  [[0.37824294 -1.56458211]]
a  ---  [[0.738668263 -1.55371857]]
a  ---  [[-0.049544692 -0.922938824]]
a  ---  [[0.378334969 -1.56414604]]
a  ---  [[0.738636 -1.55332494]]
a  ---  [[-0.0495391414 -0.922943115]]
a  ---  [[0.378360063 -1.56415367]]
a  ---  [[0.73860538 -1.55330873]]
a  ---  [[-0.0495356657 -0.922964871]]
a  ---  [[0.378360689 -1.56416655]]
a  ---  [[0.738602698 -1.55331707]]
a  ---  [[-0.0495353788 -0.922967374]]
a  ---  [[0.37836 -1.56416774]]
a  ---  [[0.738603175 -1.55331862]]
a  ---  [[-0.0495353788 -0.922966957]]
a  ---  [[0.37836 -1.5641675]]
a  ---  [[0.738603354 -1.5533185]]
a  ---  [[-0.0495354 -0.922966838]]
a  ---  [[0.378360033 -1.56416738]]
```

Analyse des valeurs du vecteur mémoire avec 3 neurones cachés :

```
Input :
  b  b  b  b  b  b  a  b  b  b  b  b  b  a  b  b  b  b  b  b  a  b  b  b  b  a  b  b  b  b
Output :
[[1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0]]

C values :
b  ---  [[0.798763573 0.939652 0.909811437]]
b  ---  [[1.20844162 1.54517817 0.906882405]]
b  ---  [[1.21613336 2.04345155 0.645792782]]
b  ---  [[1.33829856 2.49431825 0.513703883]]
b  ---  [[1.51091993 2.89181256 0.425754]]
b  ---  [[1.68114793 3.24356508 0.356183887]]
a  ---  [[-0.984261 2.09747458 -0.100903206]]
b  ---  [[-0.723080277 1.75492644 0.898726165]]
b  ---  [[-0.639973402 1.78066254 1.72914314]]
b  ---  [[-0.580848336 1.79201686 2.36966]]
b  ---  [[-0.534342289 1.79264593 2.82631278]]
b  ---  [[-0.494918197 1.78790271 3.12200451]]
b  ---  [[-0.45951879 1.7836467 3.28901339]]
a  ---  [[-0.0252290685 -0.788555264 0.963504374]]
b  ---  [[0.331623256 0.56968689 1.74341822]]
b  ---  [[0.386141539 1.33695507 1.93274951]]
b  ---  [[0.390200704 1.91014314 1.96959913]]
b  ---  [[0.385135293 2.33826303 1.98563313]]
b  ---  [[0.379219085 2.6548121 2.00355077]]
b  ---  [[0.373498291 2.88620615 2.02448583]]
a  ---  [[-0.384246349 1.20385361 -0.475074172]]
b  ---  [[0.355747163 1.82579136 0.526627183]]
b  ---  [[0.919881 2.43520331 0.783132195]]
b  ---  [[1.26561761 2.8933208 0.751357377]]
b  ---  [[1.32411492 3.27819586 0.55451721]]
a  ---  [[-0.96710676 1.95504725 -0.130247638]]
b  ---  [[-0.700389624 1.73557138 0.870856225]]
b  ---  [[-0.61137712 1.79111648 1.70361972]]
b  ---  [[-0.549868822 1.81950355 2.34028411]]
b  ---  [[-0.502933204 1.82795739 2.78618574]]
```

```
Input :
  b  b  b  b  b  b  b  b  b  b  b  b  b  b  b  b  b  b  b  b  b  b  b  b  b  b  b  b  b
Output :
[[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]]

C values :
b  ---  [[0.798763573 0.939652 0.909811437]]
b  ---  [[1.20844162 1.54517817 0.906882405]]
b  ---  [[1.21613336 2.04345155 0.645792782]]
b  ---  [[1.33829856 2.49431825 0.513703883]]
b  ---  [[1.51091993 2.89181256 0.425754]]
b  ---  [[1.68114793 3.24356508 0.356183887]]
b  ---  [[1.83730817 3.56046844 0.304240495]]
b  ---  [[1.97366858 3.8510778 0.267142504]]
b  ---  [[2.08799934 4.12149 0.241290927]]
b  ---  [[2.1808629 4.37583876 0.223506838]]
b  ---  [[2.2544589 4.61687946 0.211340219]]
b  ---  [[2.31167865 4.84646702 0.20302254]]
b  ---  [[2.35550141 5.06588268 0.197321832]]
b  ---  [[2.38866687 5.27604818 0.193396538]]
b  ---  [[2.41353178 5.47765493 0.190678179]]
b  ---  [[2.43203545 5.67124557 0.188783765]]
b  ---  [[2.44572449 5.85726595 0.187455237]]
b  ---  [[2.45580506 6.03609371 0.186518043]]
b  ---  [[2.463202 6.20806217 0.185853451]]
b  ---  [[2.46861458 6.37347031 0.185380071]]
b  ---  [[2.47256684 6.5325923 0.185041562]]
b  ---  [[2.47544765 6.68568182 0.184798643]]
b  ---  [[2.47754502 6.83297825 0.184623927]]
b  ---  [[2.47907066 6.97470856 0.184498057]]
b  ---  [[2.48017931 7.11108685 0.184407145]]
b  ---  [[2.48098469 7.24231815 0.18434149]]
b  ---  [[2.48156953 7.36859894 0.184294]]
b  ---  [[2.48199415 7.4901166 0.184259564]]
b  ---  [[2.48230219 7.60705185 0.18423453]]
b  ---  [[2.48252606 7.71957827 0.184216499]]

Input :
  a  a  a  a  a  a  a  a  a  a  a  a  a  a  a  a  a  a  a  a  a  a  a  a  a  a  a  a  a
Output :
[[0, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 0]]

C values :
a  ---  [[-0.880100727 0.386952132 0.172117174]]
a  ---  [[-0.1440911 -0.910424 0.968382835]]
a  ---  [[-0.523185909 0.675706148 -0.568709]]
a  ---  [[-0.862930596 -0.691522896 0.819635391]]
a  ---  [[-0.188848332 -0.64597 0.784945428]]
a  ---  [[-0.522061944 0.65036577 -0.554222584]]
a  ---  [[-0.849042475 -0.716813862 0.828411579]]
a  ---  [[-0.197568923 -0.618989348 0.754256845]]
a  ---  [[-0.527940512 0.637035 -0.537957311]]
a  ---  [[-0.841163278 -0.722542286 0.831925809]]
a  ---  [[-0.198503673 -0.606460035 0.736998081]]
a  ---  [[-0.533304453 0.631508052 -0.530333042]]
a  ---  [[-0.836124897 -0.727494061 0.834225833]]
a  ---  [[-0.200102031 -0.597934842 0.726244926]]
a  ---  [[-0.536239564 0.627299726 -0.524629414]]
a  ---  [[-0.832755268 -0.730167508 0.835649967]]
a  ---  [[-0.2008674 -0.592375815 0.718969345]]
a  ---  [[-0.538415551 0.624597669 -0.520841956]]
a  ---  [[-0.830371201 -0.732122421 0.83664763]]
a  ---  [[-0.201488197 -0.588395655 0.713841677]]
a  ---  [[-0.539925873 0.622612 -0.518049896]]
a  ---  [[-0.828636229 -0.733469069 0.837356]]
a  ---  [[-0.20191291 -0.585509479 0.710101366]]
a  ---  [[-0.541050553 0.62116766 -0.515997231]]
a  ---  [[-0.827342629 -0.734465718 0.837878525]]
a  ---  [[-0.20223546 -0.583351612 0.707312822]]
a  ---  [[-0.541890681 0.620078504 -0.514442861]]
a  ---  [[-0.826361835 -0.735206842 0.838270426]]
a  ---  [[-0.202477068 -0.581716299 0.705197811]]
a  ---  [[-0.542532265 0.61924988 -0.513254762]]
```