

# Projet Fin D'Étude - Utilisation de Réseaux de Neurones Récurrents (RNN) pour l'apprentissage de représentations (Embeddings) de programmes informatique

Matthieu Redor

July 2021

## 1 Introduction

Dans le cadre de la validation de la deuxième année de Master IMIS (Informatique Mobile Intelligente et Sécurisée) parcours recherche, pendant l'année universitaire 2020 / 2021, nous avons fait un projet basé sur un article scientifique informatique sur un sujet de notre choix. Étant donnée la situation sanitaire causée par la pandémie du Covid-19, le stage a dû être effectué à distance. Le sujet souhaité étant les réseaux de neurones, il a été donc proposé d'étudier l'article "Dynamic Neural Program Embedding For Program Repair" [1] du groupe de chercheur Ke Wang et Zhendong Su de l'université de Californie et de Rishabh Singh du groupe de recherche de Microsoft. Le code des réseaux de neurones présentés dans cet article est disponible sur Github.<sup>1</sup> L'objectif de ce stage est d'entamer la compréhension de l'article et de tester une des architectures proposées par l'article avec nos propres données, qui sont des codes étudiant en langage Python venant d'une plateforme éducative dans le cadre des recherches menés par l'équipe Contraintes et Apprentissage (CA) au Laboratoire d'Informatique Fondamentale d'Orléans (LIFO).

Sur les plateformes éducatives pour apprendre l'informatique, l'analyse de code peut être très utile en offrant un gain de temps non négligeable aux professeurs et aux élèves. Car avec un tel système, on peut très bien imaginer qu'un élève peut être plus autonome en ayant une indication de l'origine de son erreur directement sur son code. Ou bien, il pourrait permettre à un professeur d'expliquer l'erreur d'un étudiant en ayant de suite le type d'erreur sur l'exercice et ainsi gagner du temps qui pourra lui permettre d'en aider d'autres ou/et de terminer son cours.

Dans un premier temps, nous expliquerons ce que c'est que la fouille de données (ou Data Mining) et, en particulier, la représentation d'un programme

---

<sup>1</sup><https://github.com/keowang/dynamic-program-embedding>

et celui proposé par les auteurs de l'article en question. Ensuite, nous présenterons les réseaux de neurones et l'architecture que nous avons étudiée. Enfin, nous ferons une présentation de nos expérimentations et de nos résultats avec cette architecture et nos données.

## 2 Contexte

### 2.1 L'analyse de programme pour la détection automatique de bug

Le Data Mining est le domaine qui "a pour objet l'extraction d'un savoir ou d'une connaissance à partir de grandes quantités de données, par des méthodes automatiques ou semi-automatiques" (source Wikipedia). Ainsi l'objectif est de créer un système qui prend en entrée une grande quantité de données et qui donnera en sortie ce que l'on souhaite extraire de ces données. Ces systèmes peuvent être des clusters, des arbres de décisions ou bien, dans notre cas, des réseaux de neurones. Par exemple, à partir des analyses sanguines (quantité de plaquettes, globules blanc et rouges, etc) peut on apprendre un système (ou modèle) capable de prédire si un patient souffre ou non d'une maladie ? Ou bien à partir des contenus des e-mails d'un utilisateur, est-ce que celui-ci est un spam ou bien un mail authentique ?

Les modèles de prédiction reposent le plus souvent sur des représentations vectorielles des données à exploiter. Dans le cas de l'article étudié, le système a en entrée des codes étudiant provenant d'un site éducatif pour la programmation, incluant des erreurs classiques à ce niveau scolaire et en sortie on renvoie une prédiction du type d'erreur commise dedans. Dans cette étude nous nous intéresserons aux approches récentes d'apprentissage de représentations. Elles consistent le plus souvent à entraîner un réseau de neurones de façon supervisée ou non à paramétrer une couche intermédiaire transformant une donnée d'entrée en une représentation vectorielle de faible dimension. On parle de "plongement" (ou embedding) des données dans un nouvel espace de représentation.

Une des façons de représenter un code informatique consiste à traduire un programme en Arbres Syntaxiques Abstraits (Abstract Syntax Tree ou AST)[2] utilisé dans les analyses syntaxiques. Une des utilisations des AST est de regarder si un code comporte des erreurs syntaxiques (ex: oublie d'un point virgule à la fin d'une instruction dans un code de langage C) ou une erreur lexicale.

### 2.2 L'approche des traces de variable

Les auteurs du papier font remarquer qu'il existe des exemples de code qui ne représentent pas certaines informations.

Dans ces exemples avec les fonctions BubbleSort et InsertionSort (*Figure1* et *Figure2*) on peut voir que ces deux programmes sont syntaxiquement quasiment identiques. Ainsi si on construit les AST pour chacun de ces programmes

```

1 function BubbleSort(A) {
2   var left = 0;
3   var right = A.length - 1;
4   for( let i = right; i > left; i-- ){
5     for( let j = left; j < i; j++){
6       if(A[j] > A[j+1]){
7         var tmp = A[j];
8         A[j] = A[j+1];
9         A[j+1] = tmp;
10      }
11    }
12  }
13  return A;
14 }

```

Figure 1: La fonction BubbleSort en javascript

```

1 function InsertionSort(A) {
2   var left = 0;
3   var right = A.length;
4   for( let i = left; i < right; i++){
5     for( let j = i-1; j >= left; j--){
6       if(A[j] > A[j+1]){
7         var tmp = A[j];
8         A[j] = A[j+1];
9         A[j+1] = tmp;
10      }
11    }
12  }
13  return A;
14 }

```

Figure 2: La fonction InsertionSort en javascript



Figure 3: Représentation AST de la fonction BubbleSort en javascript

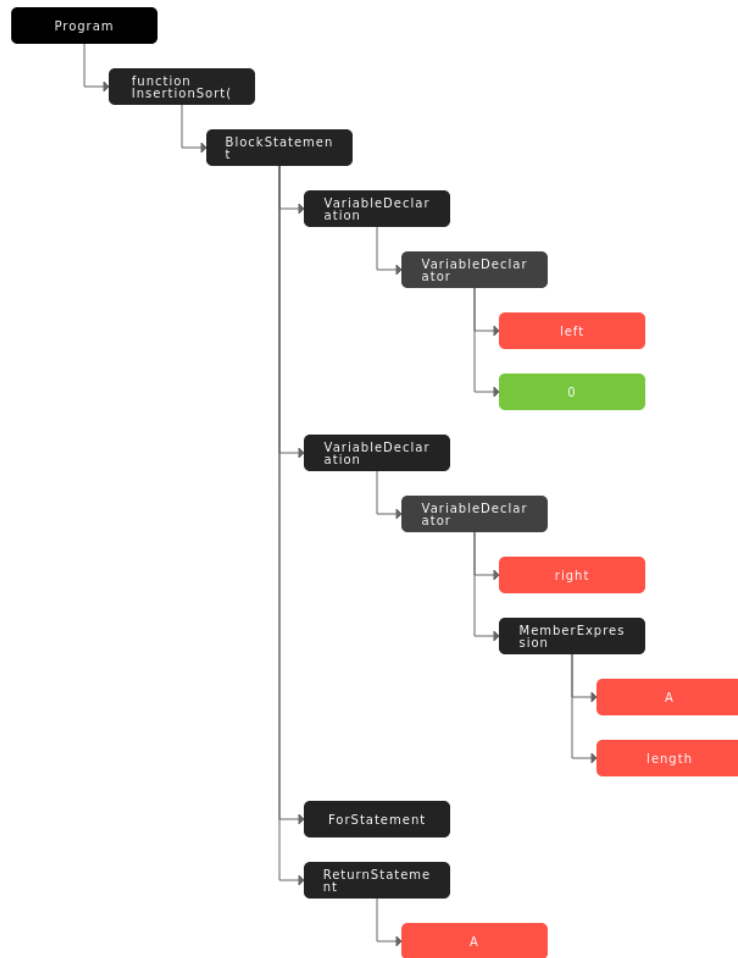


Figure 4: Représentation AST de la fonction InsertionSort en javascript

(*Figure3* et *Figure4*)<sup>2</sup>, on ne verra que des différences au niveau de l’instanciation de la variable "right" et des conditions des deux boucles for. Même si l’objectif de ces deux fonctions sont les mêmes, c’est-à-dire le tri d’une liste, les sémantiques de ces deux codes sont différentes. En effet, lorsque l’on regarde les traces de variable de la liste A de ces deux fonctions, on constate que le tri ne s’effectue pas de la même façon, même si le résultat est le même au final. Ces différences-là, selon les auteurs du papier ne sont pas visibles lors de la construction des AST, à cause de la similitude des deux codes.

Ainsi l’idée des chercheurs est de se concentrer sur les traces d’exécution du programme et non de se concentrer sur comment est écrit le programme. C’est-à-dire qu’à chaque pas d’exécution du code, on regarde comment ont évolué toutes les variables de ce programme. Un réseau de neurones par exemple, pourrait très bien apprendre à partir de ces traces d’exécution divers types de tâches. Les auteurs de l’article ont donc construit un réseau de neurones spécifique pour le plongement de programme. Et pour le tester, ils l’ont utilisé pour faire des prédictions sur le type d’erreur contenue dans un programme informatique d’apprenant, à partir de l’exploitation des traces de variables.

## 3 Principaux concepts

### 3.1 Réseaux de neurones

L’idée de base d’un réseau de neurones est l’utilisation de neurones artificiels inspiré de son homologue naturel. Ce réseau apprend à construire un modèle, une fonction ou une représentation à partir des données d’apprentissage. Plus ce réseau a des données d’exemple, plus celui-ci a des chances de faire de bonnes prédictions sur des nouvelles données. Un des tout premiers neurone artificiel utilisé est le perceptron.

Le perceptron (*Figure5*) a pour objectif "d’apprendre" avec des données dites d’apprentissage en réglant si nécessaire les différentes connexions des neurones, au fur et à mesure que l’on rentre ces données dans notre réseau[3].

Si notre réseau se trompe sur la classe de notre donnée d’entrée, alors on calcule l’erreur selon la formule suivante :  $\varepsilon = Y_d - Y$  où  $Y_d$  représente le résultat attendu et  $Y$  le résultat réel en sortie. Puis en fonction de ce dernier, de corriger plus ou moins les différents poids des connexions des neurones.

Grâce aux perceptrons, on peut par exemple classer des données en les séparants linéairement. Par exemple, si les données d’entrées sont représentables sur un plan, un perceptron apprendra à les séparer au mieux, en deux parties avec une droite. Si les données sont en trois dimensions, le perceptron apprendra à les sépareront en deux parties avec un plan et cetera. Cependant, on se rendra assez vite compte qu’un perceptron ne permet pas de classer des données non-linéairement séparable (e.g. problème bien connu du XOR) *Figure6*.

---

<sup>2</sup>Les AST en *Figure4* et *Figure5* ont été faites avec l’application en ligne suivante : <https://resources.jointjs.com/demos/rappid/apps/Ast/index.html>

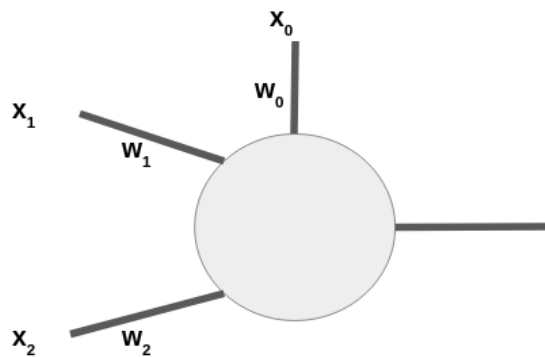


Figure 5: Le réseau de neurones "Schémas simplifié d'un perceptron à deux entrées  $x_1$  et  $x_2$ "

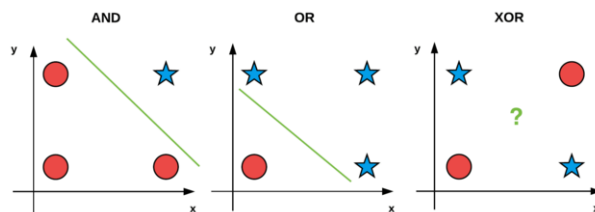


Figure 6: Le réseau de neurones "Séparation linéaire d'une simple couche de perceptron avec un AND, OR et XOR"

La solution pour traiter des apprendre des séparatrices non-linéaires est de mettre des neurones en série pour créer des couches de neurones et avec seulement une couche supplémentaire on peut résoudre cette classe de problème. (Figure7) L'algorithme utilisé quant à lui est celui de la descente de gradient (voir la partie sur les GRU), puisque l'algorithme de correction d'erreur n'est pas adapté à l'entraînement de réseaux multi-couches.


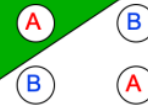



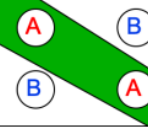
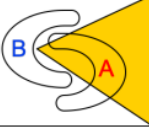

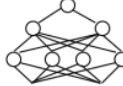
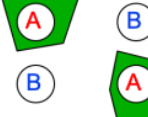


Structure	Types of Decision Regions	Exclusive-OR Problem	Classes with Meshed regions	Most General Region Shapes
<b>Single-Layer</b> 	<b>Half Plane Bounded By Hyperplane</b>			
<b>Two-Layer</b> 	<b>Convex Open Or Closed Regions</b>			
<b>Three-Layer</b> 	<b>Arbitrary (Complexity Limited by No. of Nodes)</b>			

Figure 7: Résolution des problèmes non-linéairement séparable en fonction du nombre de couche (1 à 3)

### 3.1.1 Les Réseaux de neurones récurrents (RNN)

Nous avons pour l'instant que des données qui ne sont pas séquentielles, mais lorsque ce sont des données séquentielles nous privilégions plutôt les réseaux de neurones récurrents, ou Recurrent Neural Network (RNN) en anglais[4][5]. Les données séquentielles sont des ensembles de données dont l'ordre est important. Il y a deux types de séquence : les séquences symboliques (par exemple des phrases, des codes ou bien des textes) et les séquences numériques (les séries temporelles). Par exemple pour le problème de complétion de phrase, un tel réseau doit prendre en compte l'ordre des mots de la phrase en cours d'écriture afin de bien prédire sa complétion. Il doit donc "mémoriser" les premiers mots. Un réseau de neurones est considéré comme récurrent s'il existe au moins une boucle dans ce réseau (Figure8) et peut donc prendre en compte des séquences des données en entrée.

Néanmoins, ce réseau se heurte au problème de disparition du gradient (ou 'vanishing gradient problem' en anglais) lorsque l'on applique la descente de gradient pendant la phase d'apprentissage.

En effet, plus le réseau a de couche, plus la mise-à-jour des poids sera limitée à mesure que l'on remontera vers les premiers neurones. Or, comme nous sommes

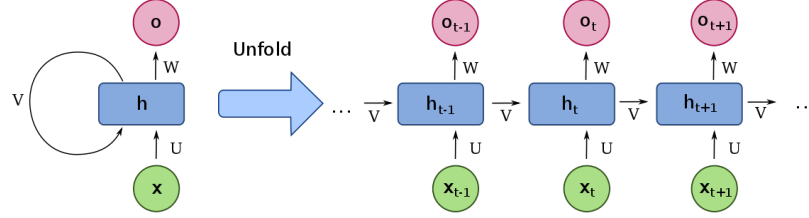


Figure 8: Représentation d'un réseau de neurones récurrents simple et sa version "dépliée"

sur des données séquentielles, cette mémoire est essentielle pour faire de bonnes prédictions.

Pour pallier ce problème, des architectures de réseaux de neurone ont été mises en place. On peut citer par exemple les architectures LSTM (Long Short Term Memories) et GRU [6] (Gated Recurrent Unit) (*Figure9*).

À noter que l'architecture GRU est une version simplifiée de l'architecture LSTM[7]. En effet, les LSTM ont trois entrées et deux sorties, alors que les GRU ont deux entrées et une seule sortie. De même, les architectures LSTM ont une performance meilleure que les GRU lorsque la profondeur des réseaux de neurones est très grande. On préférera alors les GRU au LSTM si on estime que la profondeur de notre réseau est suffisamment faible.

## 3.2 Variable Trace for Program Embedding

L'article présente trois réseaux de neurone différents, dans l'ordre "Variable Trace for Program Embedding" (*Figure10*), "State Trace for Program Embedding" et "Dependency enforcement Embedding". On ne s'intéressera qu'au premier réseau pendant ce stage de fin d'étude.

Il s'agit d'un réseau de neurones supervisé. C'est-à-dire que ce réseau apprend à prédire des étiquettes. Le réseau prend en entrée  $n$  traces de variables en One Hot Encodding, une trace étant une liste de valeurs. En sortie, le réseau donne une prédiction de l'étiquette qui est la plus probable pour les données en entrée. Il est divisé en quatre couches distinctes: Embedding, GRU, Max Pooling et un Softmax.

### 3.2.1 Embedding (Plongement)

Le principe d'un plongement (Embedding en anglais) est de transformer chaque entrée en vecteur dans une dimension donnée. Dans notre cas, la couche Embedding est là pour transformer chacune des valeurs de variable en entrée en plusieurs vecteurs de même dimension. Pour chacune des différentes variables pour un programme donné il y a un vecteur représentant toutes ses valeurs



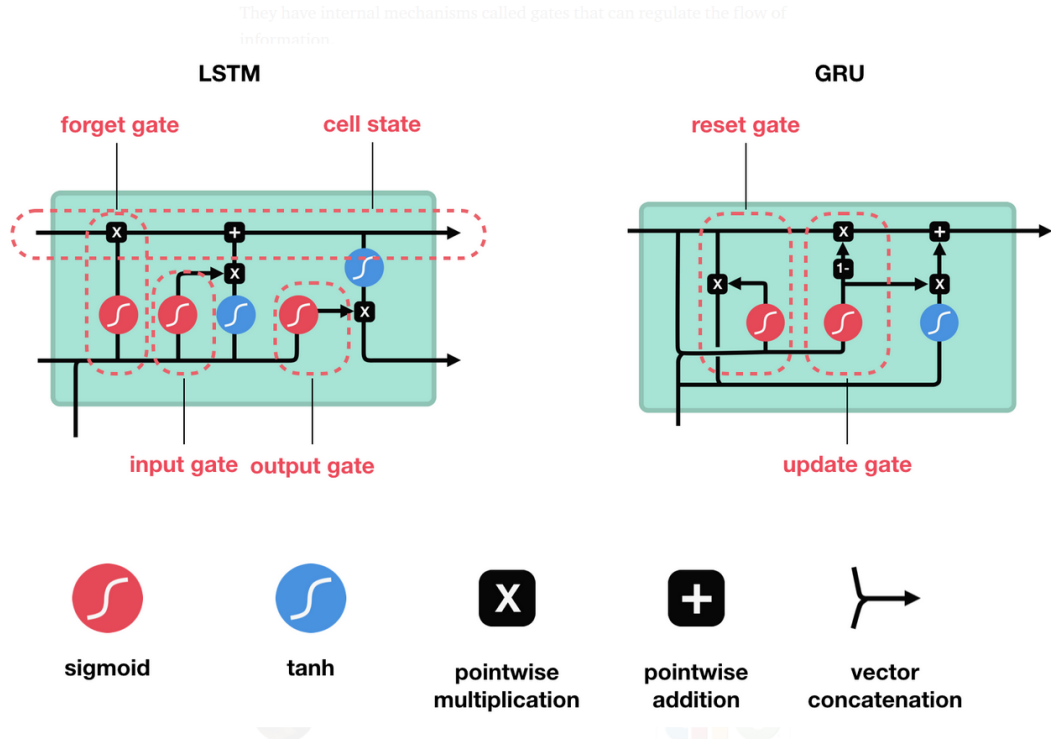


Figure 9: Schéma simplifié des architectures LSTM et GRU

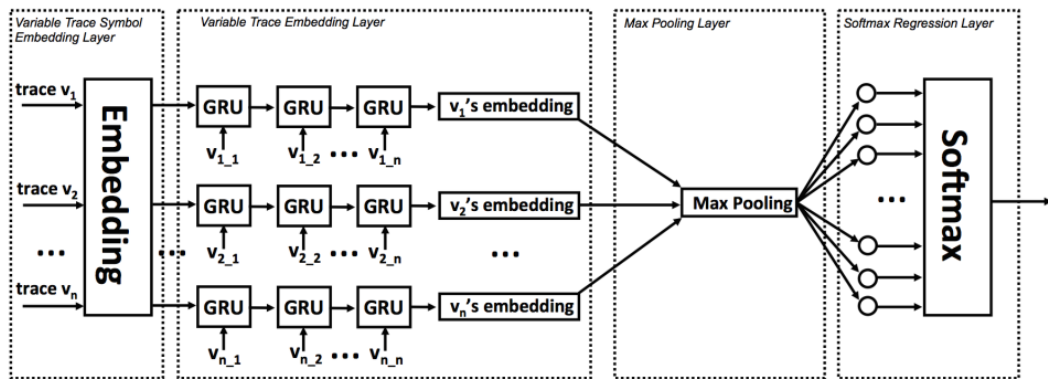


Figure 10: Le réseau de neurones "Variable Trace Program Embedding"

pendant l'exécution du programme (*trace*  $v_1$ , *trace*  $v_2$  à *trace*  $v_n$ ) et les autres vecteurs représentent chaque valeur prise par cette variable pendant cette même exécution ( $v_{1,1}$  à  $v_{1,n}$ ,  $v_{2,1}$  à  $v_{2,n}$  jusqu'à  $v_{n,1}$  à  $v_{n,n}$ ).

Comme dit précédemment l'entrée est encodé en One Hot Encoding. Le principe du One Hot Encoding est d'encoder un ensemble de  $n$  symboles en un vecteur de  $n$  bits (0 ou 1), dans lequel tout est affecté à 0 sauf l'élément correspondant au symbole en question qui aura comme valeur 1. Par exemple, si l'on doit encoder en One Hot Encoding les symboles suivants 0,5, *true*, [0, 2, 5], [0, 2, 6] et "*Hello*", alors on aura les résultats suivant dans le tableau ci-dessous, les encodages One Hot sont dans la colonne OHE (pour One Hit Encoding) pour chaque symbole :

Symbole	0	5	true	[0,2,5]	[0,2,6]	"Hello"	OHE
0	1	0	0	0	0	0	[1,0,0,0,0]
5	0	1	0	0	0	0	[0,1,0,0,0]
true	0	0	1	0	0	0	[0,0,1,0,0]
[0,2,5]	0	0	0	1	0	0	[0,0,0,1,0]
[0,2,6]	0	0	0	0	1	0	[0,0,0,0,1]
"Hello"	0	0	0	0	0	1	[0,0,0,0,0,1]

Une autre façon de faire du One Hot Encoding est d'encoder les symboles en valeur numérique. Avec l'exemple ci-dessus, on aurait respectivement pour 0,5, *true*, [0, 2, 5], [0, 2, 6] et "*Hello*" les encodages 0, 1, 2, 3, 4 et 5.

### 3.2.2 GRU

La couche GRU (Gated Recurrent Unit)(Figure 11) est ici pour traiter chaque trace de variable pas à pas, comme pour suivre l'évolution de cette variable dans le temps, tout en luttant contre le "vanishing gradient problem". Il y a deux entrées, une étant la sortie du précédent état dit caché et qui est noté  $h_{t-1}$ , sur le schéma ci-dessous en haut à gauche. Cette entrée est l'équivalent de la sortie de ce GRU, mais avec le vecteur représentant le plongement des données de l'instant  $t$  précédent. La deuxième entrée, représentée en bas à gauche, est noté  $x_t$  qui est le vecteur représentant le plongement des données à l'instant  $t$  de ce GRU. La sortie de ce GRU est un nouvel état caché qui sera transmis au prochain GRU avec le vecteur représentant le plongement des données de l'instant  $t + 1$ .

Dans cette architecture, l'intuition principale est de mettre en avant les données que l'on souhaite garder en mémoire et les données que l'on souhaite oublier. Ainsi, on a le "Reset Gate" réduisant les valeurs des vecteurs que l'on souhaite oublier. Cela se fait avec une sigmoïde du vecteur qui est l'addition terme par terme des vecteurs  $x_t$  et  $h_{t-1}$  tous deux pondérés respectivement par des poids différents  $U_r$  et  $W_r$ . Ensuite, la résultante de ce bloc est additionnée à nouveau avec  $x_t$  et tous deux pondérés respectivement par des poids  $U_h$  et  $W_h$ . Ce dernier résultat passera dans un bloc tangente hyperbolique, on notera  $H_t$  le vecteur sortant de ce bloc. De même, il y a le "Update Gate" qui augmente les valeurs des vecteurs que l'on souhaite retenir en mémoire. Il se fait avec une

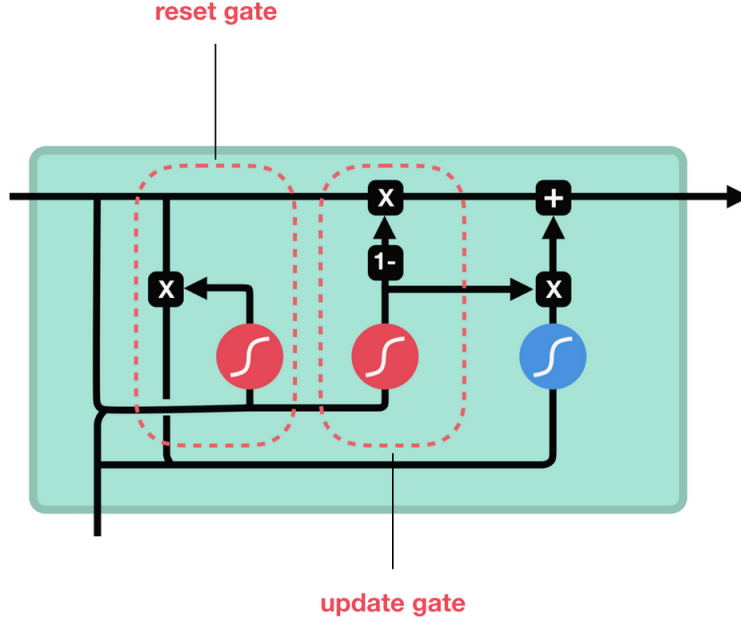


Figure 11: Schéma simplifié de l'architecture GRU

autre sigmoïde du vecteur qui est l'addition terme par terme des vecteurs  $x_t$  et  $h_{t-1}$  tous deux pondérés respectivement par des poids différents  $U_z$  et  $W_z$ . On notera  $Z_t$  le vecteur sortant de cette dernière sigmoïde. Enfin la sortie cachée  $h_t$  sera le résultat du calcul suivant :

$$h_t = (Z_t \odot h_{t-1}) + [(1 - Z_t) \odot H_t]$$

(où ici le symbole " $\odot$ " représente le produit d'Hadamard)

La sortie de dernier GRU correspondra à une représentation (embedding) d'une trace de variable. Chaque variable du programme exécuté aura donc un embedding de sa trace qui sera envoyé à la couche MaxPooling. Dans l'article, il est mentionné brièvement que ce sont des *double stacked GRU* (Figure 12) qui ont été implémentés dans le modèle. Le but étant de diminuer d'avantage l'effet du "vanishing gradient problem" dans le réseau.

### 3.2.3 Max Pooling

La couche de Max Pooling [9] a pour objectif de fusionner les embedding des traces de variables en un seul vecteur de même taille en prenant le maximum de chaque composant des vecteurs en entrée. Pour illustrer cela, soit trois vecteurs  $v_1, v_2$  et  $v_3$  en entrée de cette couche MaxPooling :

$$v_1 = (1, 5, 9), v_2 = (2, 6, 7) \text{ et } v_3 = (3, 4, 8)$$

En sortie de cette couche on aura alors :

$$v = \text{MaxPooling}(v_1, v_2, v_3)$$

$$v = (\text{Max}(1, 2, 3), \text{Max}(5, 6, 4), \text{Max}(9, 7, 8)) = (3, 6, 9)$$

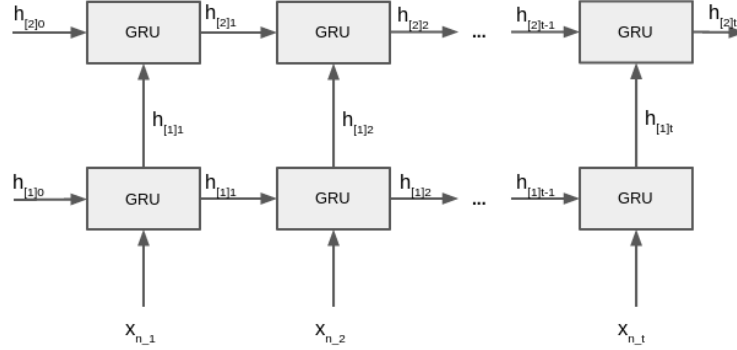


Figure 12: Un exemple simplifié d'une couche *double stacked GRU*

### 3.2.4 Softmax

La couche SoftMax (ou fonction exponentielle normalisée) permet de calculer, pour chaque classe de prédiction que le réseau a appris, sa probabilité de correspondre avec l'entrée :

$$\sigma(z)_i = \frac{e^{z_i}}{\sum_{j=1}^K e^{z_j}} \text{ pour } i = 1, \dots, K \text{ et } z = (z_1, \dots, z_K)$$

Dans le cas des réseaux de neurones, on va multiplier le vecteur en entrée avec une matrice "de poids" qui a autant de lignes qu'il y a de classes dans le modèle, afin de transformer le vecteur d'entrée en un nouveau vecteur de longueur égale au nombre de classe du modèle (appelés *logits*). Chacun des éléments de ce nouveau vecteur correspondra à la probabilité que sa classe correspondante soit correct en fonction de l'entrée du modèle.

## 4 Manipulations

### 4.1 Prise en main du programme

Le réseau de neurones étudié mis à disposition par les auteurs a été codé en Python avec le module Tensorflow de Google[10]. Le programme des chercheurs tournait à la base avec du Python 2.7. Mais nous avons choisi de le faire tourner avec du Python 3 et avec le module Tensorflow à la même version 1.14 qu'à l'origine. Par ailleurs, nos codes sont disponibles sur un dépôt git sur GitHub[11].

Nous avons effectué quelques modifications sur le code original des chercheurs (*Training.py*) afin qu'il puisse être exécuté avec la dernière version de Python. Ainsi nous avons remplacé les lignes :

```
from tensorflow import rnn
import tensorflow as tf
```

par les lignes suivantes :

```
import tensorflow.compat.v1 as tf
tf.disable_v2_behavior()
from tensorflow.compat.v1.nn import rnn_cell as rnn
```

De même, nous avons fait en sorte qu'à chaque itération, le réseau teste ses prédictions sur les données d'entraînement et sur les données de test. Respectivement, nous souhaitons visualiser ses erreurs d'apprentissage et ses erreurs de généralisation sur de nouvelles données.

Le modèle nécessite des hyper-paramètres :

*num\_epochs* : Nombre d'itération d'entraînement que le modèle exécutera.

*learning\_rate* : Le taux d'apprentissage du modèle.

*n\_hidden* : Le nombre d'état caché dans les RNN.

*vocabulary\_size* : La taille du vocabulaire du modèle, c'est à dire le nombre de symbole différent que peut rencontrer le modèle en entrée.

*CLASSES* : Le nombre d'étiquette différent que le modèle devra prédire.

*batch\_size* : Le nombre de variable dans les lots en entrée du programme. Dans notre cas, ce sera le nombre maximal de variable qu'aura au moins un des programmes exécutés avec ses cas de test.

*program\_number* : Le nombre de programme en tout dans chacun des lots en entrée du programme.

La construction et l'entraînement du modèle est réalisé par la classe *Training* prenant six entrées :

*all\_program\_symbol\_traces* : une liste de taille *program\_number* comprenant des listes de taille *batch\_size* pour chaque programme donné, de liste de même taille (pas forcément *batch\_size*) représentant chacun les traces d'une variable encodé en One Hot Encoding. Pour la phase d'entraînement.

*trace\_lengths* : une liste de taille *program\_number* comprenant des listes de taille *batch\_size* pour chaque programme donné, de valeur entière représentant le nombre de pas d'exécution pour chacune de ces variables. Pour la phase d'entraînement.

*labels* : une liste de taille *program\_number* comprenant des listes de taille un, représentant l'étiquette pour ce programme. Pour la phase d'entraînement.

*test\_all\_program\_symbol\_traces* : une liste de taille *program\_number* comprenant des listes de taille *batch\_size* pour chaque programme donné, de liste de même taille (pas forcément *batch\_size*) représentant chacun les traces d'une

variable encodé en One Hot Encoding. Pour la phase de test.

*test\_trace\_lengths* : une liste de taille *program\_number* comprenant des listes de taille *batch\_size* pour chaque programme donné, de valeur entière représentant le nombre de pas d'exécution pour chacune de ces variables. Pour la phase de test.

*test\_labels* : une liste de taille *program\_number* comprenant des listes de taille un, représentant l'étiquette pour ce programme. Pour la phase de test.

## 4.2 Expérimentations

### 4.2.1 Les données à traiter

Notre objectif est d'entraîner ce réseau de neurones à reconnaître un exercice à partir des traces de variable récupérées pendant son exécution. Nous avons utilisé un jeu de 5690 tentatives de code étudiant en Python sur 66 types d'exercices différents. Ces codes étudiant sont stockés dans un fichier json contenant ces 5690 codes et diverses informations complémentaires dans un dictionnaire. Les jeux de données utilisés sont NewCaledonia-5690 (ou NC5690) et NewCaledonia-1014 (ou NC1014) et sont des programmes créés par 60 étudiants de l'université de New-Caledonia, sur une plateforme d'éducation à la programmation développée par le département informatiques de l'université d'Orléans [12]. Comme ce dernier fichier est très volumineux, il y a également un autre fichier json contenant un sous ensemble du fichier précédent de 1014 codes, afin de tester plus facilement nos codes. Dans le dictionnaire de ces fichiers nous avons les clefs suivantes pour chaque code étudiant :

*exercise\_name* : Identifiant de l'exercice tenté par l'élève.

*date* : Date de la tentative.

*correct* : true si le code correspond à ce qui est attendu par les enseignants, false sinon.

*upload* : Code Python de la tentative de l'étudiant en String (chaîne de caractères).

*user* : Identifiant de l'étudiant.

*eval\_set* : "training" si le code servira en tant qu'exemple d'entraînement, ou "test" si le code servira en tant qu'exemple de test.

Puis nous avons un autre fichier json contenant quant à lui les solutions de chaque exercice et de ses cas de test. De même, il s'agit d'un dictionnaire possédant les clefs suivantes :

*solution* : code Python correspondant à la solution de l'exercice en String (chaîne de caractères).

*funcname* : nom de la fonction Python.

*entries* : liste des entrées des cas de test pour tester les tentatives étudiantes.  
Cette dernière clé sera la liste des entrées pour chaque cas de test.

*exo\_name* : identifiant de l'exercice.

Pour récupérer nos traces de variable, nous avons exécutés dans un script python à part en utilisant le *tracing* du module *sys* de Python[13], tous les codes étudiant sur tous les cas de test correspondant à leur type d'exercice.

Nous avons utilisé également le module *sys* de Python pour créer un objet Tracer afin de récupérer ces traces sur un Thread pendant l'exécution de tous les codes.

À chaque fin d'exécution, on remplit un dictionnaire que l'on écrira dans un nouveau fichier json qui contiendra les clés suivantes :

*user* : identifiant de l'étudiant.

*exo\_name* : identifiant de l'exercice.

*nb\_code* : "i-ème" programme au moment de son traitement pendant la récupération de ses traces de variable

*number\_cas\_test* : "i-ème cas de teste au moment où l'on récupère les traces de variable

*eval\_set* : "training" si le code servira en tant qu'exemple d'entraînement, "test" si le code servira en tant qu'exemple du premier jeu de test ou "test2" si le code servira en tant qu'exemple du deuxième jeu de test.

*date* : date de la tentative étudiante.

*label* : étiquette pour le réseau correspondant au type d'exercice.

*trace\_ohe* : liste de liste de taille *batch\_size* contenant des listes correspondant aux traces d'une des variables d'un programme exécuté et encodé en One Hot Encoding.

*longueur\_traces* : nombre de pas d'exécution dans le programme exécuté.

Comme les traces de variables sont encodées en One Hot Encoding, nous construisons un dictionnaire (qui sera le vocabulaire de ce modèle) répertoriant toutes les différentes valeurs de variables observées durant toutes les exécutions, que ce soit des chaîne de caractères, des entiers, des listes, etc.

Voici un exemple de code Python issu du jeu de donnée NC5690 en *Figure13* et un des cas de test utilisé en entrée [12, 1, 25, 7]. On obtient après exécution du programme les traces brutes :

```
{ 'liste': [12, 1, 25, 7] },  
{ 'liste': [12, 1, 25, 7] },  
{ 'liste': [12, 1, 25, 7] },
```

```
def minimum(liste):
    if len (liste)==0:
        res=None
    else:
        res=liste[0]
        for i in range (1,len(liste)):
            if liste[i]<res:
                res=liste[i]
        return res
```

Figure 13: Fonction minimum en Python issue du jeu de données NC5690

```
{'liste': [12, 1, 25, 7], 'res': 12},
{'liste': [12, 1, 25, 7], 'res': 12, 'i': 1},
{'liste': [12, 1, 25, 7], 'res': 12, 'i': 1},
{'liste': [12, 1, 25, 7], 'res': 1, 'i': 1},
{'liste': [12, 1, 25, 7], 'res': 1, 'i': 2},
{'liste': [12, 1, 25, 7], 'res': 1, 'i': 2},
{'liste': [12, 1, 25, 7], 'res': 1, 'i': 3},
{'liste': [12, 1, 25, 7], 'res': 1, 'i': 3},
{'liste': [12, 1, 25, 7], 'res': 1, 'i': 3}]
```

On voit qu'il y a 12 pas dans le programme et qu'il y a au maximum trois variables en tout dans le programme. On souhaite une liste de trace de variables encodées en One Hot Encoding. Dans cet exemple, on obtiendra une trace de variable de même taille pour chacune des variables *liste*, *res* et *i*. Lorsqu'une variable n'apparaît pas dans les premiers pas d'exécution, alors on mettra le symbole 0 dans notre encodage. Après le traitement des données, nous obtenons le résultat suivant :

```
[
[ [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1], # variable 'liste'
[0, 0, 0, 2, 2, 2, 3, 3, 3, 3, 3, 3], # variable 'res'
[0, 0, 0, 0, 3, 3, 3, 4, 4, 5, 5, 5] ] # variable 'i'
]
```

Lorsque l'on a traité tous les codes avec ses cas de test, on ajoute si nécessaire des listes de 0 pour avoir au final *batch\_size* traces de variables pour un programme avec un cas de test donné. Avec notre exemple si *batch\_size* = 5 alors on obtiendra après ce dernier traitement :

```
[
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
[0, 0, 0, 2, 2, 2, 3, 3, 3, 3, 3, 3],
[0, 0, 0, 0, 3, 3, 3, 4, 4, 5, 5, 5],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
]
```



Voici le résumé du jeu de données NC1014 après traitement :

**Taille du jeu d'entraînement :** 12781

**Taille du premier jeu de test :** 739

**Nombre de classe :** 8

**Nombre maximal de variable :** 9

**Taille du vocabulaire :** 795

Voici le résumé du jeu de données NC5690 après traitement :

**Taille du jeu d'entraînement :** 47876

**Taille du premier jeu de test :** 2840

**Nombre de classe :** 65

**Nombre maximal de variable :** 32

**Taille du vocabulaire :** 11553

#### **4.2.2 Expérimentation 1 : classification sur cas de tests inchangés**

Nous avons donc entraîné ce modèle "Variable Trace for Program Embedding" à prédire la fonctionnalité (ou le type d'exercice) de chacun des programmes des ensembles de données NC5690 et du sous-ensemble NC1014. C'est à dire que nous avons une étiquette correspondant à un exercice. Pour NC1014 nous avons 8 étiquettes pour 8 fonctionnalités différentes et pour NC5690 65 étiquettes pour 65 fonctionnalités différentes.

L'apprentissage a été réalisé à partir du sous-ensemble "training" puis testé ensuite en test pour vérifier l'erreur d'apprentissage à chaque itération.

Pendant l'apprentissage on affiche la perte du réseau. Il s'agit du résultat de la fonction d'erreur sur lequel se base le modèle pour configurer les poids des connexions entre les neurones. Dans notre cas, à partir du vecteur en sortie de la couche Softmax il s'agit de maximiser la probabilité de l'étiquette correct des exemples de la phase d'apprentissage aussi proche de 1 que possible. On utilise plutôt les logarithme de ces probabilités, car l'algorithme de descente de gradient converge plus rapidement en utilisant le logarithme de la fonction d'erreur. Comme ce sont des probabilités, donc des valeurs entre 0 et 1, les logarithmes seront des valeurs entre  $-\infty$  et 0. Or les réseaux de neurone ne sont capable que de minimiser, donc on utilisera l'opposé de ce logarithme ( $-\log()$ ) pour n'avoir que des valeurs positifs à minimiser au plus près de 0.

Sur la *Figure14* nous représentons la moyenne des pertes sur l'ensemble des données d'apprentissage en fonction du nombre d'itération d'entraînement. Nous avons mesuré également l'erreur d'apprentissage et l'erreur de généralisation du modèle. C'est à dire qu'après chaque itération, nous redonnons en test tous

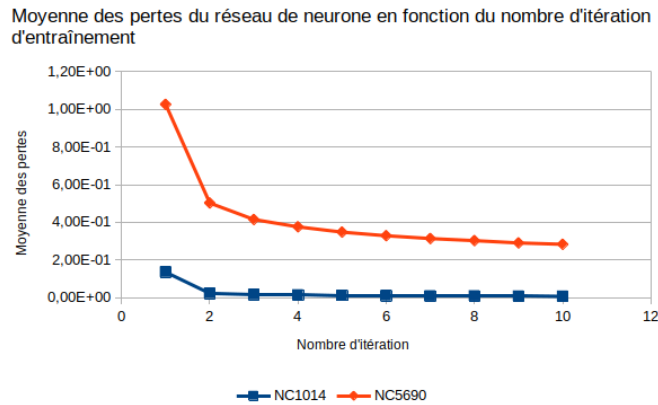


Figure 14: Graphique de la moyenne des pertes à chaque itération pendant la phase d'entraînement du réseau avec les deux jeux de données

les exemples d'apprentissage et calculons ensuite le taux d'erreur commis par le modèle et nous faisons de même avec le jeu de test, qui servira à savoir si le modèle parvient à bien généraliser avec de nouvelles données.

Sur la *Figure15* nous représentons l'erreur d'apprentissage et l'erreur de généralisation de NC1014 et NC5690. Le comportement du modèle est correct

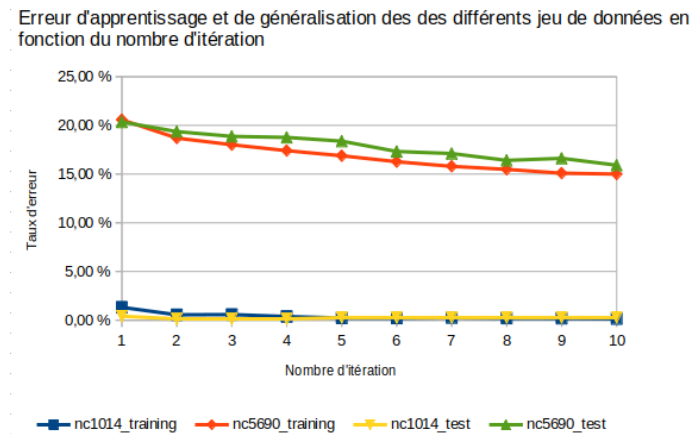


Figure 15: Graphique de l'erreur d'apprentissage et de généralisation du modèle avec les deux jeux de données NC1014 et NC5690

en terme d'apprentissage avec un taux d'erreur proche de 0% pour NC1014 et inférieur à 20% pour NC5690, de plus l'erreur d'apprentissage diminue à mesure des itérations pour les deux jeu de données. Mais surtout on voit que l'erreur de généralisation est similaire à celui de l'apprentissage, même si le taux d'erreur

reste globalement légèrement au-dessus de ce dernier.

On observe aussi que ces scores sont comparables à ceux observés sur cette même tâche et ces mêmes jeu de données par des approches d'apprentissage d'embeddings non-supervisés suivi d'une étape de classification supervisée [12].

Ces résultats peuvent s'expliquer par le vocabulaire One Hot Encodding utilisé. Car pour un exercice donné, nous utiliserons toujours les mêmes cas de test et donc les mêmes vocabulaires en entrée pour chacun de ces exercices. Selon le type d'exercice, on peut aller de deux cas de test à une dizaine. Nous ne pouvons pas négliger la possibilité que le réseau ne fait qu'apprendre à déduire le type d'exercice qu'avec les variables de départ. Sans oublier que le nombre de pas d'exécution peut aussi donner un indice sur le type d'exercice en entrée.

La représentation que nous utilisons ici s'appuie fortement sur les valeurs de variables. Donc nous faisons l'hypothèse que le modèle pourrait être sensible aux valeurs en entrées dans les cas de tests, que nous cherchons à valider par une expérimentation complémentaire.

#### 4.2.3 Expérimentation 2 : sensibilité aux cas de tests

Pour vérifier ça, nous avons donc lancés une seconde expérimentation pour être certain que le réseau a bien appris comment s'exécute les programmes et non pas seulement de ce qu'il y a en entrée. Pour ce faire nous avons créé des nouveau cas de test avec des entrées inédite par rapport aux exemples d'entraînement et nous avons fait en sorte que ces entrées soient les mêmes dans la mesure du possible pour tous les types d'exercices.

Ces nouveaux cas de tests seront mis à part et s'exécuteront avec la totalité des codes étudiant des jeux de données. La taille du second jeu de test pour NC1014 est 2698, tandis qu'il est de 13701 pour NC5690. Ainsi, nous avons relancé pour chacun des jeux de données NC1014 et NC5690 avec 10 itérations, en affichant à la fin de chaque cycle d'apprentissage la moyenne des pertes du réseau pendant son apprentissage, l'erreur d'apprentissage et les erreurs de généralisation sur chacun des deux jeux de test.

Les résultats de cette expérimentation sont les Figures, les données sont disponibles dans le fichier *Resultats.Experimentations.ods* du dépôt git :

On voit nettement que sur les figures *Figure16* et *Figure17* les erreurs de généralisation entre le jeu de test original et le nouveau passe de presque 0% à environs 55% pour NC1014 et d'environ 20% à 70% pour NC5690. Cela confirme l'hypothèse que ce modèle exploitant les traces de variables est très sensibles aux cas de tests. Ce réseau n'apprend donc pas la fonctionnalité à partir des traces de variables en entrées.

## 5 Conclusion

En conclusion, nous pouvons dire que le modèle "Variable Trace Program Embedding" proposé par les chercheurs n'est pas adapté pour la tâche de prédiction de la fonctionnalité (de l'exercice) à partir de ses traces de variables.

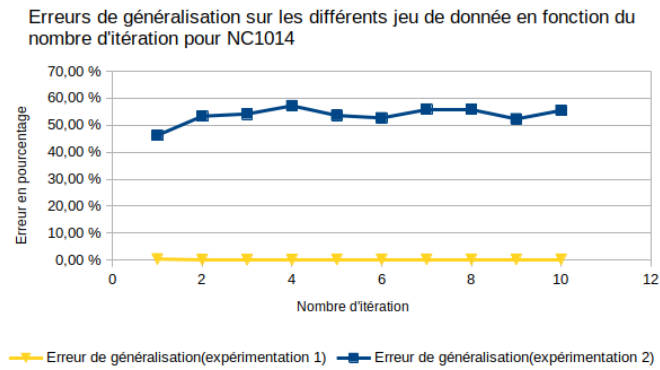


Figure 16: Graphique de l'erreur de généralisation à chaque itération d'entraînement du réseau de neurones pour chacun des deux jeux de test avec le jeu de données NC1014

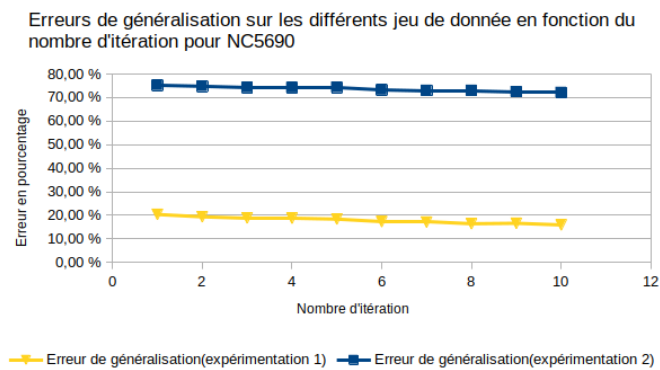


Figure 17: Graphique de l'erreur de généralisation à chaque itération d'entraînement du réseau de neurones pour chacun des deux jeux de test avec le jeu de données NC5690

En effet, comme ce modèle est très sensible aux cas de test utilisés, il est donc préférable de le faire apprendre des données provenant d'un seul programme. Dans ce contexte, on peut envisager de l'entraîner à prédire des erreurs pour un même exercice, comme ce que les auteurs l'ont fait dans leur papier. Ce qui signifie qu'il faudra alors utiliser un modèle différent pour chaque exercice.

Les deux autres modèles proposés par les auteurs, à savoir les modèles "State Trace for Program Embedding" et "Dependency Enforcement Embedding", peuvent faire l'objet de ces mêmes expérimentations pour vérifier si ces conclusions sont valables également pour eux. Ou bien au contraire, ils pourraient être plus adaptés dans cette tâche.

À l'occasion de ce stage, j'ai pu approfondir le sujet des réseaux de neurones, en particulier des réseaux de neurones récurrents. Que ce soit au niveau théorique, en me refaisant un schéma plus précis du modèle étudié avec l'évolution des vecteurs pendant son fonctionnement. En particulier le passage dans les GRU avec lequel il a fallu comprendre l'intuition derrière cet architecture.

Au niveau de la pratique, j'ai appris les bases du module Tensorflow, pour mieux comprendre le code des chercheurs. Même s'il n'a pas été nécessaire d'approfondir tant que ça au final, je pourrais très bien finir les cours de Tensorflow de Google. De même, je suis quand même satisfait d'avoir réussi à coder par moi-même tout ce qui a permis de traiter les données d'entrées.

Pour finir, je tiens à remercier à Guillaume Cleuziou et Matthieu Exbrayat pour m'avoir permis de faire ce projet à la place d'un stage en ces temps difficiles. Et aussi pour avoir pris de leur temps pour organiser des réunions par visioconférence très régulièrement pour m'aider à mener à bien ce projet et la rédaction de ce rapport.

## References

- [1] Wang, Ke and Singh, Rishabh and Su, Zhendong, *Dynamic Neural Program Embeddings For Program Repair*, International Conference on Learning Representations, 2018
- [2] Wikipédia, *Arbre de la syntaxe abstraite*  
[https://fr.wikipedia.org/wiki/Arbre\\_de\\_la\\_syntaxe\\_abstraite](https://fr.wikipedia.org/wiki/Arbre_de_la_syntaxe_abstraite)
- [3] Adrian Rosebrock, *Implementing the Perceptron Neural Network with Python*  
[https://www.pyimagesearch.com/2021/05/06/implementing-the-perceptron-neural-network-with-](https://www.pyimagesearch.com/2021/05/06/implementing-the-perceptron-neural-network-with-python/)
- [4] Wikipédia, *Réseau de neurones récurrents*  
[https://fr.wikipedia.org/wiki/Réseau\\_de\\_neurones\\_récurrents](https://fr.wikipedia.org/wiki/Réseau_de_neurones_récurrents)
- [5] Michael Phi, *Illustrated Guide to Recurrent Neural Networks*  
<https://towardsdatascience.com/illustrated-guide-to-recurrent-neural-networks-79e5eb8049>
- [6] Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk et Yoshua Bengio *Learning Phrase*

*Representations using RNN Encoder-Decoder for Statistical Machine Translation*

Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)

- [7] Michael Phi, *Illustrated Guide to LSTM's and GRU's: A step by step explanation*  
<https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explan>
- [8] Alexander Agung Santoso Gunawan( Binus University ), *Automatic Music Generator Using Recurrent Neural Network*  
[https://www.researchgate.net/publication/342444314\\_Automatic\\_Music\\_Generator\\_Using\\_Recurrent\\_Neural\\_Network](https://www.researchgate.net/publication/342444314_Automatic_Music_Generator_Using_Recurrent_Neural_Network)
- [9] Aphex34, *Illustration d'un MaxPooling*  
CC BY-SA 4.0, <https://commons.wikimedia.org/w/index.php?curid=45673581>
- [10] Google, *Cours en ligne de Tensorflow*  
<https://developers.google.com/machine-learning/crash-course?hl=fr>
- [11] Matthieu Redor, *Dépôt git des codes utilisés pendant le stage de fin d'étude*  
[https://github.com/MJJR/ProjetFinEtude.2021\\_DynamicNeuralProgramEmbedding](https://github.com/MJJR/ProjetFinEtude.2021_DynamicNeuralProgramEmbedding)
- [12] Guillaume Cleuziou et Frédéric Flouvat *Learning student program embeddings using abstract execution traces*  
EDM 2021 : Educational Data Mining, 2021
- [13] *Tracing a Program As It Runs*  
<https://pymotw.com/2/sys/tracing.html>