Étudiant N° 11294239
L3 INFO - IED
EDF5ALAA

LICENCE INFORMATIQUE

Chapitre 2 : projet

Rapport: ART1-Emulator

FOURCROY Mathieu

-

ALGORITHMIQUE AVANCÉE

Année 2014/2015

1. Introduction

a. Motivation et objectifs

Dans ce deuxième chapitre j'ai choisi la réalisation du projet car lors du premier projet j'avais déjà essayé d'implémenter un simulateur de réseau ART1 afin de l'utiliser pour reconnaitre les même lettres que dans le premier projet et de comparer les performances des réseaux BAM et ART1. Mais ici il s'agit d'utiliser les données concernant les champignons et non plus les lettres. Cependant, comme dans la premier projet le programme présenté dans ce projet est capable d'utiliser n'importe quel jeux de données du moment qu'il est correctement formaté. Ainsi, même si je n'ai pas eu le temps de tester avec le jeu de données d'empreintes digitales, ce programme devrait être capable de les classer.

Le but de ce projet est donc de réaliser un simulateur de réseau ART1 capable de classer des champignons en deux catégories : vénéneux / comestibles, en fonction de leurs attributs. Le but ultime étant d'apprendre un certain nombre de champignons au réseau, puis de le tester en le laissant classer d'autres champignons. Les résultats seront ensuite analyser afin de déterminer l'importance et les effets des paramètres du réseau (vigilance, bruit). À partir de quelles valeurs de système perd-t-il les pédales et devient inutilisable pour le classement ? Dans quelles conditions est-il le plus fiable ?

b. Installation

L'installation est simplissime. Vous aurez besoin de cmake :

```
# apt-get install cmake
```

Ensuite vous n'avez plus qu'à compiler :

```
$ cmake . && make
```

Vous pouvez ensuite lancer le programme en appelant le script *runart1* avec les options qu'il attend. Tapez la commande suivante pour connaitre ces options :

```
$ ./runart1 -h
```

Ici j'ai utilisé un script bash car getopt (C) ne gère pas les options de plus d'un caractère. Le script positionne les options puis appel le programme. Vous pouvez donc appeler le programme sans passer par le script en lui passant tous les arguments qu'il attend, dans le bon ordre.

c. Documentation

La documentation est disponible dans le répertoire ***doc*** et est, je pense, assez complète. Elle est au format HTML et a été générée grâce à *doxygen*. La page principale de la documentation reprend notamment les explications du paragraphe a., en plus détaillées. Je vous invite à lire cette documentation technique pour ce qui concerne le fonctionnement concret du programme : ce rapport ne présente qu'une étude de cas accompagnée de l'analyse des résultats.

2. Cahier des charges

Dans ce projet il faudra implémenter un simulateur de réseau ART1, capable, au minimum de classer (dans le sens classement et non classification) les champignons du jeu de données *mushrooms.data*. L'entrée du programme sera donc ce jeu de donnée ou une partie en mode apprentissage puis une autre partie du jeu de donnée ou éventuellement de nouveaux vecteurs représentant de nouveaux champignons, en mode de test. La sortie du programme en mode test sera la classe estimée du champignon (vénéneux / comestible), le mode apprentissage ne nécessite pas de sortie mais, quelque soit le mode on pourra éventuellement enregistrer des informations sur le réseau dans des fichiers.

Une interface graphique serait la bienvenue et permettrait de proposer de nombreuses options intuitives et de consulter facilement les résultats, voir pourquoi pas de les comparer, malheureusement je n'aurais pas le temps de réaliser une telle interface. Je me contenterais donc d'une commande acceptant les paramètres du réseau via les options. Les résultats pourront toutefois être stockés dans des fichiers séparés afin de pouvoir être exploités plus aisément.

Enfin une documentation suffisante sera nécessaire.

3. Environnement

Comme pour le premier projet, j'ai décidé d'utiliser le langage C pour l'implémentation. Comme pour tous les modèles de réseaux de neurones, les algorithmes sont assez simples donc je ne pense pas qu'une maquette soit nécessaire.

En revanche, j'ai utilisé la bibliothèque CCL (C containers library ; http://www.cs.virginia.edu/~lcc-win32/ccl/ccl.html) afin de faciliter la manipulation de vecteurs et de listes. En effet, la seule structure de données récurente des réseaux de neurones est le vecteur. Et le langage C ne possède aucune API standard pour les vecteurs. J'ai utilisé la CCL dans plusieurs projets universitaires et personnels et je trouve qu'elle remplie très bien son objectif. Vous n'aurez cependant pas à l'installer car je l'ai jointe dans le projet (répertoire *ccl*).

4. Génération des jeux de formes

Le projet contient le code source du simulateur d'ART1 dans le répertoire *src* mais contient également quelques scripts annexes qui sont indirectement liés au simulateur dans le répertoire *utils*. *delete_results* ne fait que supprimer ce qu'il y a dans le répertoire des résultats : *results*. *generate_binary_patterns*, en revanche, permet de générer des jeux de formes binaires à partir de jeux de formes non binaires, en appelant le script python *genbinpats.py*.

Voici comment appeler ce script python, via le script bash :

```
./generate_binary_patterns -i input_file -o output_file -s class_pos
```

input_file est le fichier qui contient le jeu de formes original, au format non binaire. Il doit contenir une forme par ligne et toutes les formes doivent avoir le même nombre d'attributs, séparés par une virgule (format csv donc).

class_pos est l'indice de l'attribut qui représente la classe de chaque forme. Cet attribut peut être soit tout au début, soit tout à la fin de la forme mais pas ailleurs. Par exemple :

```
class, attrib0, attrib1, attrib2
```

Dans ce cas class_pos vaut 1. Ou bien :

```
attrib0, attrib1, attrib2, class
```

Dans ce cas class_pos vaut -1.

Prenons un exemple. J'ai spécialement joint le jeu de données *Zoo* (https://archive.ics.uci.edu/ml/datasets/Zoo) afin d'effectuer une démonstration de "binarisation". Pour transformer ce jeu de données, appelez :

```
./generate_binary_patterns -i ./data/Zoo/zoo.data -o ./data/zoo.csv -s '-1'
```

Le script python va scanner chaque ligne du fichier d'entrée et compter le nombre de valeurs différentes de chaque attribut. Le processus ressemble à une addition telle que :

```
e,t,s,h,s,s,p
a,t,s,r,s,s,j
e,b,s,h,s,a,j
e,t,s,g,s,s,p
------------
2 2 1 3 1 2 2
```

On se rend compte que les troisièmes et cinquièmes attributs sont inutiles puisqu'ils ne peuvent prendre qu'une seule valeur. Du point de vue du réseau, cela n'apportera aucun renseignement quant à la façon de classer les formes.

Le script connait désormais le nombre de valeurs différentes que chaque attribut peut prendre, en plus de savoir combien il y a d'attributs. Il connait aussi ces valeurs car il a construit une liste à deux dimensions qui ressemble à :

```
[['e', 'a'],
 ['t', 'b'],
 ['s'],
 ['h', 'r', 'g'],
 ['s'],
 ['s', 'a'],
 ['p', 'j']]
```

Après cela, le script aplati la liste et attribue un 1 à chaque formes, pour chaque attribut qu'elle possède. Si elle ne possède pas cet attribut c'est un 0 qui est ajouté à la forme binaire. Dans notre exemple, la première forme sera convertie telle que :

```
['e', 'a', 't', 'b', 's', 'h', 'r', 'g', 's', 's', 'a', 'p', 'j']
-----------------------------------------------------------------
  1   0   1   0   1   1   0   0   1   1   0   1   0
```

Il faut noter que la classe des formes (qui peut se trouver en début ou en fin de forme) est automatiquement placée au début des formes binaires générées et elle n'est, évidemment, pas convertie en binaire.

5. Structures de données

Le programme utilise beaucoup la structure Vector de la C Containers Library et un peu la structure List. Ce sont des équivalents des vecteurs et listes standards du langage C++ et vous pouvez en apprendre d'avantage à la page suivante : http://www.cs.virginia.edu/~lcc-win32/ccl/ccl.html.

Le programme utilise aussi trois autres structures de données :

- CSVLine : contient la classe (un attribut) d'une forme ainsi que le reste de ses attributs. Il est important d'isoler la classe du reste car le réseau ne doit en aucun cas en tenir compte. La classe n'est utilisée que pour calculer la précision du classement (le ratio entre le taux de réussite et le taux d'échec).

- Cluster : contient le prototype du cluster : un vecteur de la même taille que les formes d'entrée qui ne contient de bits à 1 que là où toutes les formes qui le constituent ont un bit à 1. Il contient également la liste des formes qui ont contribué à construire son prototype. En effet, c'est le principe du réseau ART1, lorsqu'une forme est assignée à un cluster, le prototype de ce dernier va être modifié de façon à ressembler d'avantage à cette forme (c'est un peu le cas de tous les réseaux de neurones que j'ai étudié). Enfin il possède également une troisième variable qui n'est qu'un flag indiquant si le cluster a été inhibé ou non.

- InParam : regroupe les paramètres du réseau, pouvant être positionnés depuis les options de la ligne de commande. Ces paramètres sont :

  - beta : influence le nombre de clusters créé. Ce paramètre n'existe pas dans les réseaux ART1 mais je l'ai ajouté pour expérimenter d'avantage avec, cependant sa valeur par défaut est 1 et cette valeur n'a aucune influence sur le réseau, c'est comme si ce paramètre n'existait pas.

  - skip : indique l'emplacement de la classe dans les formes.

  - vigilance : le paramètre vigilance du réseau.

  - maxPasses : le nombre maximum d'itérations que l'algorithme pourra effectuer.

  - minFluc : la fluctuation minimum en-dessous de laquelle l'algorithme s'arrêtera. Ce paramètre n'existe pas dans les réseaux ART1 mais initialement je voulais trouver un moyen pour faire baisser la fluctuation d'une itération à l'autre, un peu comme le rayon de voisinage ou la capacité d'apprentissage d'un réseau SOM. Cela rendrait l'algorithme un peu plus intelligent mais ne parvenant à trouver de méthode efficace qui ne compromette pas le reste de l'algorithme, j'ai laissé cette fonctionnalité de côté. Ce paramètre reste utilisable mais son utilité est limitée.

  - trainNoise : la quantité de bruit à ajouter aux formes d'apprentissage. J'ai expérimenté en ajoutant du bruit dans les formes d'apprentissage même si cela n'a

pas trop de sens puisqu'en général les réseaux de neurones, supervisés ou non, apprennent depuis un jeu de données valide (non faussé) puis tentent de reconnaitre des formes bruitées et non l'inverse. Cependant, il était intéressant de noter que le réseau s'en sortait pas trop mal même après avoir appris des formes partiellement erronées.

- testNoise : la quantité de bruit à ajouter aux formes de test. Parfois on voudra demander au programme si un champignon est comestible ou non sans forcément connaitre tous ses attributs ou bien en ayant involontairement introduit des erreurs. Ici les erreurs sont introduites volontairement afin de voir jusqu'à quelle quantité de bruit le réseau reste fiable.

- prefix : est utilisé pour nommer les fichiers contenants les résultats.

- trainFile : indique le jeu de donné à utiliser pour l'apprentissage.

- testFile : indique le jeu de données à utiliser pour le test de classement.

6. Démonstration

Vous pouvez lancer le programme avec :

```
$ ./runart1 -t ./data/mushrooms_train.csv -T ./data/mushrooms_test.csv -s -n 5 -N 5
  -p 5
```

Dans cette commande nous demandons au programme d'utiliser le jeu de données des champignons d'apprentissage (mushrooms_train.csv) qui contient les 6124 premiers champignons afin d'apprendre au réseau, puis nous lui demandons de classer les 2000 derniers champignons à partir de ce qu'il aura appris. -n 5 indique au programme d'ajouter 5% de bruit à toutes les formes. -p 5 demande au programme de ne pas boucler plus de 5 fois. -s indique simplement que le premier attribut des formes doit être ignoré puisqu'il s'agit de leurs classes. Les classes ne seront pas utilisées dans le réseau en lui-même mais permettront de vérifier la fiabilité du classement. Appelez le programme avec -h ou --help afin d'en savoir plus sur ses différentes options.

Voici ce que le programme affiche sur stdout :

```
   _   ___ _____ ___ ___ ___ __  __ _   _ _      _ _____ ___  ___ 
  /_\ | _ \_   _|_ _/ __|_ _|  \/  | | | | |    /_\_   _/ _ \| _ \
 / _ \|   / | |  | |\__ \| || |\/| | |_| | |__ / _ \| || (_) |   /
/_/ \_\_|_\ |_| |___|___/___|_|  |_|\___/|____/_/ \_\_| \___/|_|_\

-------------------------------------------------------

Runing ART1 algorithm using following parameters:
    Training file: ./data/mushrooms_train.csv
    Testing file: ./data/mushrooms_train.csv
    Skipping first attribute of every patterns
    Output file prefix: art
    Fluctuation percentage: 5%
    Max number of iterations: 5
    Beta parameter: 1
    Vigilance parmeter: 0.5

------------- INTERNING TRAINING PATTERNS ------------

Opening file "./data/mushrooms_train.csv"... OK
Reading input file "./data/mushrooms_train.csv"... OK

--------- CHECKING TRAINING PATTERNS VALIDITY --------

6124 patterns have been scanned
Patterns length is 117
Removing patterns containing only 0... OK
Number of network patterns: 6124

-------------------- ADDING NOISE ------------------

Adding 5% of noise to each patterns... OK
6788 pattern's bits have been switched to 0

------------------ TRAINING STAGE -----------------

Pass n°  | No. reassigned | Fluctuation | No. clusters
--------+----------------+-------------+-------------
      1 |           6124 |        100% |          305
      2 |           5011 |    81.8256% |          366
```

```
        3 |           3400 |    55.5193% |          369
        4 |           2218 |    36.2182% |          369
        5 |           1925 |    31.4337% |          369

-------------- WRITING TRAINING RESULTS -------------

Writing results...
Opening file "./results/train/art_results"... OK
Opening file "./results/clusters/art_clust_0.csv"... OK
Opening file "./results/clusters/art_clust_1.csv"... OK
Opening file "./results/clusters/art_clust_2.csv"... OK
[...]
Opening file "./results/clusters/art_clust_368.csv"... OK
OK

------------ INTERNING TESTING PATTERNS ------------

Opening file "./data/mushrooms_test.csv"... OK
Reading input file "./data/mushrooms_test.csv"... OK

--------- CHECKING TESTING PATTERNS VALIDITY --------

2000 patterns have been scanned
Patterns length is 117
Removing patterns containing only 0... OK
Number of network patterns: 2000

-------------------- ADDING NOISE -------------------

Adding 5% of noise to each patterns... OK
2213 pattern's bits have been switched to 0

------------------ TESTING STAGE ------------------

SUCCESS: 1888 (94.4%)
FAIL: 112 (5.6%)
RATIO: 16.8571

-------------- WRITING TESTING RESULTS -------------

Opening file "./results/test/art_results"... OK
Writing results... OK
```

Examinons de plus près. Les premières ligne résument les paramètres du réseau. Le jeu de formes d'apprentissage utilisé est *mushrooms_train.csv*, le programme ne bouclera pas plus de 5 fois. Le programme ajoutera 5% de bruit aux formes (y compris aux formes d'apprentissage, c'est pourquoi le réseau comptent autant de clusters d'ailleurs...). Mais on savait déjà tout ça.

a. Apprentissage

Voici les étapes de l'algorithme d'apprentissage (pour des détails techniques consultez la documentation HTML dans */doc/index.html*) :

1. Le programme liste les paramètres du réseau. Si un paramètre n'est pas indiqué par l'utilisateur alors sa valeur par défaut est utilisée (une fois de plus, voyez l'aide du programme pour connaitre ces valeurs par défaut).

2. Le fichier du jeu de formes d'apprentissage est ouvert. Les formes sont extraites et

insérées dans des structures CSVLine (une structure par forme). Cette structure contient la classe de la forme (qui doit être isolée du reste puisque le réseau n'est pas sensé la connaitre) ainsi que les valeurs des attributs de la forme.

3. Le programme vérifie la taille des formes. Toutes les formes doivent avoir la même taille, si ce n'est pas le cas, le programme se termine avec un message d'erreur.

4. Les formes sont normalisées. Les attributs qui les composent sont arrondis à 0 ou 1. Si le jeu de formes a été générés avec le script generate_binary_patterns il est impossible que les formes présentent des valeurs d'attributs autres que 0 ou 1.

5. La vraie phase d'apprentissage commence :

   a. Le programme essaie d'assigner chaque forme à apprendre à un cluster en créant de nouveaux clusters si nécessaire.

   b. La passe se finie lorsque toutes les formes à apprendre ont été assignées à un cluster.

   c. Une nouvelle passe commence alors et le programme tente de réassigner les formes à d'autres clusters, plus appropriés, afin de tenté d'améliorer la fiabilité du classement. Le programme en profite pour calculer le taux de fluctuation de la dernière passe. Il s'agit de pourcentage de formes réassignées à un cluster différent.

   d. Le programme effectue passe après passe jusqu'à ce que le taux e fluctuation soit inférieur au taux minimum spécifié avec l'option -f ou bien jusqu'à ce que le nombre maximum de passes (spécifié avec -p) soit atteint.

6. Les résultats sont écrits dans un fichier spécialement créé (dans */results/train*). D'autres fichiers sont créés, contenant les informations des clusters (dans */results/clusters/*).

7. Le programme calcule les classes des clusters. La classe d'un cluster est déterminée par les classes des formes qu'il contient (les formes qui l'ont modelé). Pour déterminer la classe d'un cluster, le programme compte le nombre de chaque classe des formes du cluster. La classe majoritaire devient la classe du cluster. Idéalement, un cluster ne devrait contenir que des formes de même classe mais le réseau peut se tromper...

8. Le programme vérifie, pour chaque forme, qu'elle a été correctement classée. Le ratio de succès/erreur est calculé et ajouté à la fin du fichier contenant les résultats de l'apprentissage.

b. Test

Les étapes suivantes ne sont menées que si un jeu de forme à classer à été indiqué avec l'option -T.

9. Le programme répète les étapes 1 à 4 précédentes avec le jeu de formes à classer.

10. Le test à proprement parler débute :

    a. Pour chaque forme, le programme trouve le cluster le plus proche.

    b. Puis il compare la classe de ce cluster avec la classe de la forme. Si elles sont identiques alors la forme a été classée avec succès. Sinon l'algorithme n'a pas pu classer la forme dans le bon cluster et a introduit une erreur.

    c. Le ratio de succès/erreur est calculé.

11. Les résultats du test de classement sont écrits dans */results/test/*. Ce fichier indique quelles formes ont été correctement classées et lesquelles ont été classées dans un mauvais cluster.

c. Résultats de l'exemple

Toutes les étapes précédentes des algorithmes du réseau ART1 sont menées dans notre exemples. Inutile de détailler les 5 premières étapes où les formes d'apprentissage sont extraites du fichier *mushroms_train.csv*. Mais ensuite commence l'apprentissage du réseau, suivi du test :

- Lors de la première passe les formes sont assignées aux clusters et le taux de fluctuation est de 100% puisqu'il s'agit de la première passe, tous les clusters ont été créés.

- Dans la seconde passe environ 82% (sans ajouter de bruit le taux de fluctuation serait d'environ 5% seulement!) des formes sont réassignées à d'autres clusters.

- Le programme continu ainsi jusqu'à atteindre la sixième passe dans laquelle il n'entrera pas puisque le nombre maximum de passes (5) est dépassé.

- Les fichiers contenant les résultats sont écrits.

- L'apprentissage est terminé, le programme commence la phase de test en sélectionnant la première forme du jeu de test.

- Il trouve le cluster le plus proche de cette forme.

- Puis il compare la classe du cluster à celle de la forme et incrémente le nombre de classements corrects ou mauvais selon que les classes correspondent ou non.

- Le programme continu ainsi avec la seconde forme a tester et ainsi de suite jusqu'à ce que toutes les formes ont été classées.

- Maintenant que toutes les formes ont été classées, le ratio de succès/erreur est calculé et imprimé à l'écran. On voit que 1888 formes ont été classées dans un

cluster correspondant à leur classes. Les 112 autres ont été classées dans un cluster ne correspondant pas à leur classe.

d. Interprétation des résultats

Le classement de l'exemple précédent à été volontairement compliqué par l'ajout de bruit dans les formes d'apprentissage et de test. Malgré cela, le réseau présente des résultats satisfaisants. Il est parvenu à classer correctement 94.4 % des champignons présentés lors de la phase de test. Sur les 112 qui n'ont pas été correctement classés 9 ont été classés comme comestibles mais sont en réalité vénéneux. Donc vous auriez moins d'une chance sur 200 (0.45%) de manger un champignon vénéneux par erreur si vous vous fiez aux résultats du programme.

Les résultats peuvent être encore meilleurs en ajustant le taux de vigilance et, évidemment en évitant de bruiter les données comme on va le voir dans la prochaine section...

7. Simulation et analyse des résultats

a. Données

Le jeu de données utilisé répertorie 8124 espèces de champignons (https://archive.ics.uci.edu/ml/datasets/Mushroom). Chaque champignon comporte 22 attributs :

```
 1. cap-shape: bell=b,conical=c,convex=x,flat=f, knobbed=k,sunken=s
 2. cap-surface: fibrous=f,grooves=g,scaly=y,smooth=s
 3. cap-color: brown=n,buff=b,cinnamon=c,gray=g,green=r,
pink=p,purple=u,red=e,white=w,yellow=y
 4. bruises?: bruises=t,no=f
 5. odor: almond=a,anise=l,creosote=c,fishy=y,foul=f, musty=m,none=n,pungent=p,spicy=s
 6. gill-attachment: attached=a,descending=d,free=f,notched=n
 7. gill-spacing: close=c,crowded=w,distant=d
 8. gill-size: broad=b,narrow=n
 9. gill-color: black=k,brown=n,buff=b,chocolate=h,gray=g,
green=r,orange=o,pink=p,purple=u,red=e, white=w,yellow=y
 10. stalk-shape: enlarging=e,tapering=t
 11. stalk-root: bulbous=b,club=c,cup=u,equal=e, rhizomorphs=z,rooted=r,missing=?
 12. stalk-surface-above-ring: fibrous=f,scaly=y,silky=k,smooth=s
 13. stalk-surface-below-ring: fibrous=f,scaly=y,silky=k,smooth=s
 14. stalk-color-above-ring: brown=n,buff=b,cinnamon=c,gray=g,orange=o,
pink=p,red=e,white=w,yellow=y
 15. stalk-color-below-ring: brown=n,buff=b,cinnamon=c,gray=g,orange=o,
pink=p,red=e,white=w,yellow=y
 16. veil-type: partial=p,universal=u
 17. veil-color: brown=n,orange=o,white=w,yellow=y
 18. ring-number: none=n,one=o,two=t
 19. ring-type: cobwebby=c,evanescent=e,flaring=f,large=l,
none=n,pendant=p,sheathing=s,zone=z
 20. spore-print-color: black=k,brown=n,buff=b,chocolate=h,green=r,
orange=o,purple=u,white=w,yellow=y
 21. population: abundant=a,clustered=c,numerous=n, scattered=s,several=v,solitary=y
 22. habitat: grasses=g,leaves=l,meadows=m,paths=p, urban=u,waste=w,woods=d
```

Extrait de https://archive.ics.uci.edu/ml/datasets/Mushroom

Le jeu de données à été converti en binaire avec l'outil précédemment présenté car le réseau ART1 n'accepte que des vecteurs binaires.

Ce jeu de données à été divisé en deux afin d'apprendre une partie des données au réseau, en le laissant ignorer l'autre partie. Cette seconde partie sera le jeu de test que le réseau devra classer en fonction de ce qu'il connait. Les 6124 premiers champignons constituent l'échantillon d'apprentissage et les 2000 champignons restant constituent l'échantillon de test.

b. Hypothèse

Formulons l'hypothèse que le niveau de bruit influera négativement sur la précision du classement des formes de test. En effet, en bruitant les formes de test, on modifie les données qu'elles transportent et on les rend fausses. Et comme on le sait, la classification dans les réseaux ART dépend entièrement des données. Ainsi, on s'attend à ce que le bruitage des formes, en transmettant de mauvaises informations au réseaux, l'empêche de

classer correctement ces formes de test.

La vigilance, quant à elle, influe directement sur le nombre de clusters du réseau. S'il y a trop peu de vecteurs, le réseau ne parviendra pas à distinguer suffisamment les caractéristiques des champignons. Donc théoriquement, plus il y a de clusters, plus la précision du classement sera élevée. Cependant, il faut aussi tenir compte de la mémoire utilisée par le programme et du temps des simulations car plus il y a de clusters plus il faut de place pour les stocker et plus le classement nécessitera de calculs.

L'hypothèse est : "Plus la vigilance est élevée, meilleurs seront les résultats. Moins le bruit dans les formes est élevée, meilleurs seront les résultats."

c. Apprentissage

Pour l'apprentissage, l'algorithme présenté plus haut a été utilisé. Cet algorithme gère la plasticité (apprendre de nouvelles formes sans altérer celles qui ont déjà été apprises). J'ai décidé, arbitrairement (en fait surtout pour que certaines simulations ne durent pas un temps très long), de limiter le nombre de boucles d'apprentissage à 5.

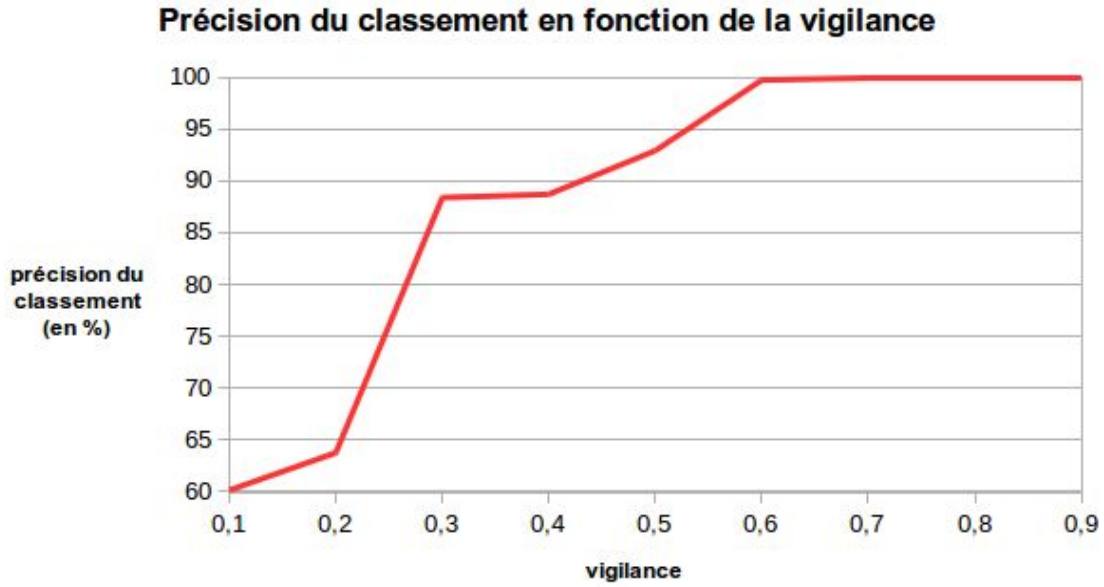
Dans cette phase d'apprentissage j'ai utilisé les 6124 premiers champignons, constituant les formes d'entrées. Elles sont présentées à l'algorithme dans l'ordre. 9 simulations ont été effectuées avec différent niveaux de vigilance. Les résultats sont rassemblés dans le tableau suivant :

vigilance	accuracy	nb clusters	time	fluctuation
0,1	60,1568	2	1,122	0
0,2	63,7655	4	8,207	14,6146
0,3	88,3899	6	3,21	5,19268
0,4	88,7002	9	3,455	8,06662
0,5	92,9131	14	2,37	7,34814
0,6	99,7224	23	2,02	1,12671
0,7	99,9837	60	3,474	10,663
0,8	100	257	16,509	22,0444
0,9	100	1191	649,204	15,2025

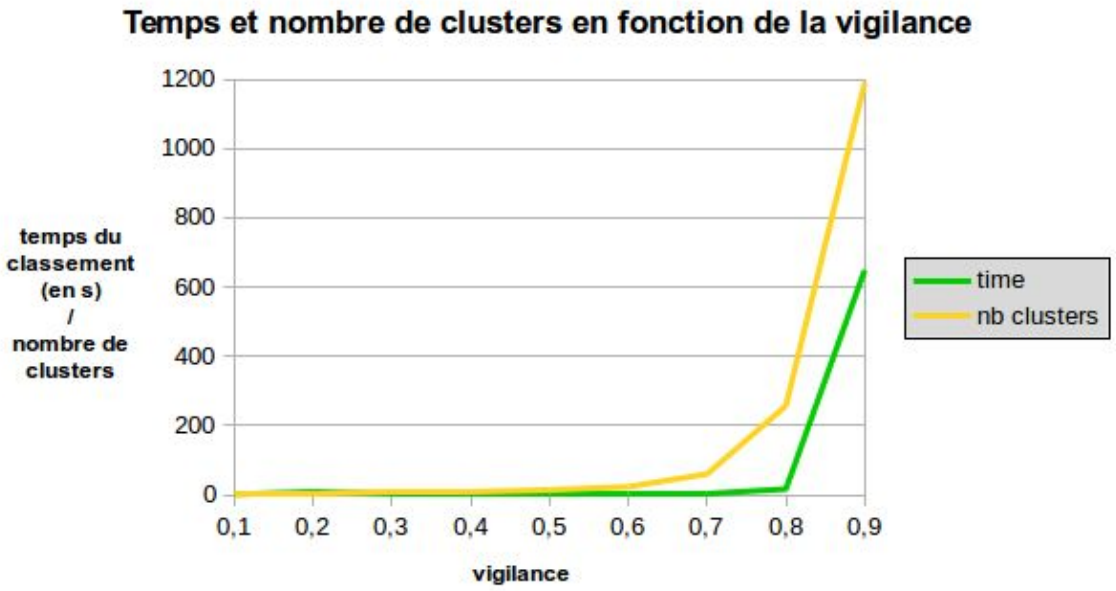
Précision du classement du jeu d'apprentissage en fonction de la vigilance.

La précision du classement (accuracy) correspond au taux de réussite c'est à dire le pourcentage des 6124 formes qui ont été classées dans un cluster correspondant à leur classe. On voit que les meilleurs résultats sont obtenus avec $p = 0.8$ et $p = 0.9$ (p est la vigilance). Le nombre de clusters augmente de façon significative. La courbe représentant cette augmentation (voir graphique ci-dessous) ressemble un peu à une courbe de fonction exponentielle, bien qu'elle ne le soit pas. Le temps de calcul des simulations semble être fonction du nombre de clusters et de la fluctuation : globalement, plus il y a de clusters et plus la fluctuation à l'intérieur de ces clusters est élevée plus le temps du classement est élevé. Cela semble logique puisque la création et la modification des clusters sont les opérations les plus couteuses en temps de l'algorithme.

Le graphique suivant montre la courbe de la précision du classement du jeu d'apprentissage en fonction de la vigilance :



Et ici on voit très clairement que plus la vigilance est élevée plus le classement est précis. Cependant cette courbe n'est pas linéaire, on peut observer un "palier" entre 0.3 et 0.4 et un autre entre 0.6 et 0.9. En effet, à partir de $p = 0.6$ la précision dépasse les 99%. Voyons maintenant les effets de la vigilance sur le temps de classement et le nombre de clusters créé.



Le graphique ci-dessous regroupe sur le même axe le temps de classement et le nombre de clusters générés.

Le nombre de clusters reste raisonnable (< 200) jusqu'à p = 0.7 mais décolle très rapidement passé ce seuil de vigilance. On en déduit que plus il y a de clusters plus la précision de classement est élevée.

Le temps de calcul est inférieur à 20 secondes jusqu'à p = 0.8 mais à p = 0.9 le classement prend plus de 10 minutes ! Les courbes décrites dans le graphique précédent se ressemblent beaucoup et on peut en déduire que plus il y a de clusters plus le temps de la simulation est élevé.

En définitive, le meilleur score est réalisé ici avec p = 0.8 puisque la précision est de 100%, comme avec p = 0.9 mais la simulation utilise 257 clusters et se termine après environ 16 secondes, contre 1191 clusters et plus de 10 minutes pour la simulation avec p = 0.9. La vigilance à 0.8 offre la meilleure précision possible tout en étant beaucoup plus économique en temps et en mémoire que la simulation avec une vigilance de 0.9.

d. Test

La phase de test est effectuée en deux temps par l'algorithme décrit précédemment. Dans un premier temps les formes du jeu de test sont classées par l'algorithme puis dans un second temps, le programme vérifie le classement en le comparant aux classes "réelles" des formes du jeu. De cette façon il est possible de calculer la précision et l'efficacité générale du classement.

Dans cette phase de test de classement le jeu de formes utilisé contient les 2000 derniers champignons du jeu de données binaire complet. Il s'agit de comparer les résultats de plusieurs simulations en fonction de la vigilance du réseau et du bruit des formes. Ainsi, 90 simulations ont été effectuées. Elles reprennent la phase d'apprentissage exposée dans la section précédente, qui apprend les 6124 premier champignons aux réseau ART1, puis tente de classer les 2000 derniers champignons dans le réseau ainsi créé.

Le bruit est ajouté aux formes en permutant un certain pourcentage de leur bits. C'est-à-dire que pour un jeu de 2000 formes de 117 bits = 234000 bits au total, et un taux de bruit de 20%, 46800 bits seront permutés (46000 en réalité, à cause des arrondis).

i. Précision des classements

Avant de se demander à partir de quel taux de bruit et à partir de quelle valeur de vigilance le système est fiable il faudrait convenir d'un seuil d'acceptabilité. La table ci-après associe à chaque fourchette de précision du classement une étiquette qualifiant sa fiabilité :

précision	fiabilité
100 - 90	Très élevée
90 - 80	élevée
80 - 70	moyenne
70 - 50	faible
50 - 0	Trop faible

Table des niveaux de fiabilité.

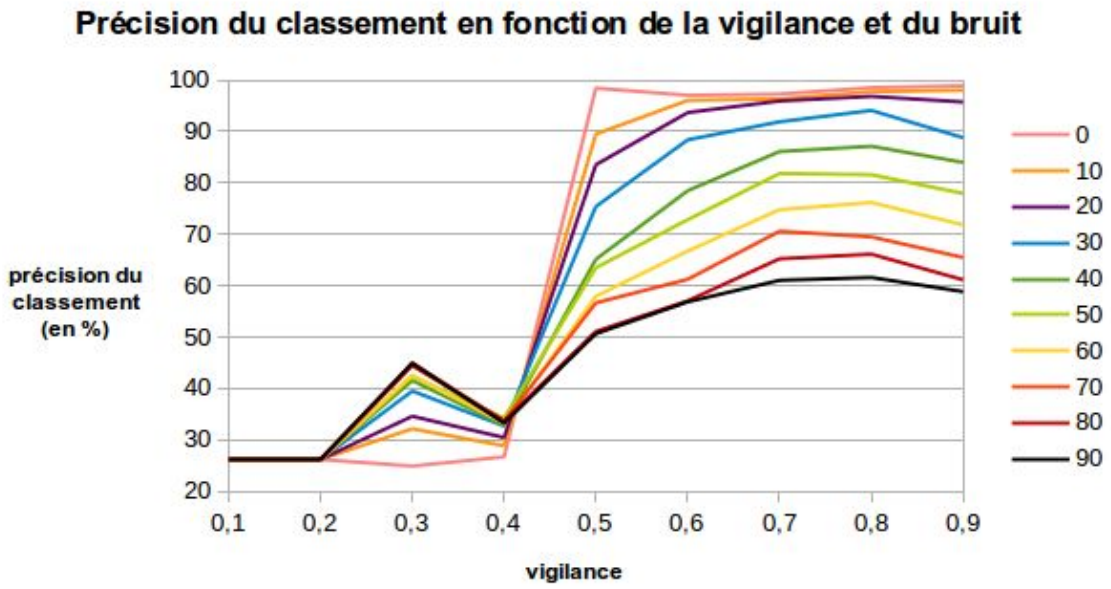
Je rappelle qu'on parle ici du niveau de fiabilité et j'estime qu'en dessous de 50 % (moins d'une chance sur deux) de fiabilité cela n'est pas acceptable.

La table ci-après regroupe les valeurs de précision observées dans les 90 simulations :

p\ns	0	10	20	30	40	50	60	70	80	90	avg
0,1	26,2	26,2	26,2	26,2	26,2	26,2	26,2	26,2	26,2	26,2	26,2
0,2	26,2	26,2	26,2	26,2	26,2	26,2	26,2	26,2	26,2	26,2	26,2
0,3	24,9	32,15	34,6	39,5	41,55	42,35	42,6	44,95	44,45	44,95	39,2
0,4	26,7	28,85	30,45	32,7	32,85	34,25	33,15	33,65	33,7	33,25	32
0,5	98,45	89,5	83,55	75,4	65,15	63,5	57,95	56,65	51,1	50,65	69,2
0,6	97,05	96,05	93,7	88,4	78,5	72,85	66,75	61,25	57,05	56,85	76,8
0,7	97,3	96,35	95,95	91,9	86,1	81,85	74,8	70,6	65,25	61,05	82,1
0,8	98,55	97,75	96,85	94,1	87,15	81,6	76,2	69,5	66,15	61,6	82,9
0,9	98,9	98,05	95,75	88,8	84	77,95	71,85	65,5	61,15	58,85	80,1
avg	66,03	65,08	64,81	62,58	58,63	56,31	56,86	50,50	47,92	46,62	57,19

Précision du classement en fonction de la vigilance et du bruit.

D'après la table précédente, on peut facilement déduire que sans bruit et en positionnant la vigilance à 0.9 on obtient le meilleur résultat avec 98,9% des formes correctement classées. Cependant, tout niveau de bruit confondus, c'est $p = 0.8$ qui offre les meilleurs résultats avec une précision moyenne de 82,9 %. Le graphique suivant montre les résultats du tableau ci-dessus.



Précision du classement en fonction de la vigilance et du bruit.

On voit encore plus clairement que, globalement, les simulations les plus précises sont obtenues avec $p = 0.8$. À partir de $p = 0.5$, toutes les simulations donnent une précision supérieur à 50%. Nous pouvons confirmer que, globalement, plus la vigilance est élevée, plus le classement est fiable même s'il faut noter que, tous niveaux de bruit confondu, $p = 0.8$ offre de meilleurs résultats que $p = 0.9$.

Le niveau de bruit quant à lui à globalement tendance à diminuer la précision des classements. Plus les formes sont bruitées et moins elles seront classées correctement. C'est tout à fait logique et conforme à l'hypothèse de départ. Cependant, un phénomène remarquable, visible dans ce graphique, est qu'avec p entre environ 0.3 et 0.4 la tendance s'inverse. C'est-à-dire que plus le niveau de bruit est élevé, plus la précision du classement l'est. Enfin on peut aussi noter que le bruit "adouci" la courbe : plus le bruit est important, moins la vigilance n'a d'impact sur la précision du classement. Cela se traduit par une courbe moins "anguleuse". On voit nettement que la courbe pour noise = 0 présente des pics beaucoup plus prononcés, notamment entre $p = 0.4$ et $p = 0.5$ où la courbe fait un saut de 71.7%. La précision passe soudainement d'un niveau de fiabilité trop faible (26.7%) à un niveau de fiabilité très élevé (98.4%). La **courbe pour noise = 90**, elle, reste beaucoup plus proche d'une précision moyenne de 50 %, elle n'augmente que de 7.6% entre $p = 0.4$ et $p = 0.5$.

De ce graphique nous pouvons donc conclure qu'avec une vigilance entre 0.1 et environ 0.4, le bruit dans les formes de test à un effet positif (ou au moins nul) sur la précision du classement. Au-delà de ce seuil de vigilance, le bruitage tend à impacter négativement la précision du classement. Il faut toutefois mesurer ces propos car le classement ne devient fiable qu'à partir de $p = 0.4 / 0.5$ en fonction de niveau de bruit. Ainsi, la partie dans laquelle le bruit améliore la précision n'offrira jamais de résultats fiables. Enfin on peut dire que le classement, peu importe le bruit, devient fiable à partir de $p = 0.5$ mais plus on

ajoute du bruit, au-delà de ce seuil, moins le classement sera précis.

ii. Nombre de clusters des simulations

Les clusters sont stockés en mémoire et, pour un même jeu de données, plus il y a de clusters, plus il faut de mémoire pour les stocker (en plus du temps nécessaire à leur création/modification). Voici une table qui, d'après les nombres de clusters observés dans les simulations, indiquent le niveau d'utilisation mémoire, bien sûr, ce n'est pas quantitatif en terme de taille mémoire et les appréciations peuvent varier selon les machines.

temps	utilisation mémoire
0 - 10 clusters	insignifiante
10 - 50 clusters	Très faible
50 - 200 clusters	faible
200 - 1000 clusters	modérée
1000 + clusters	importante

Table des niveaux d'utilisation mémoire.

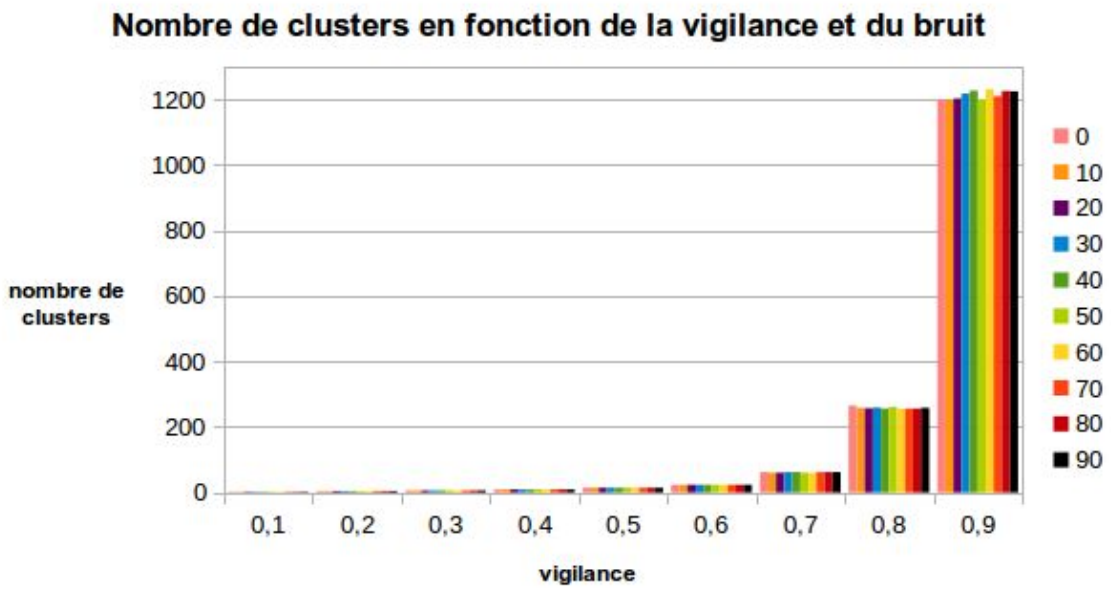
La table suivante met en comparaison les nombres de clusters observés dans les simulations effectuées en fonction de la vigilance et du bruit :

p\ns	0	10	20	30	40	50	60	70	80	90	avg
0,1	2	2	2	2	2	2	2	2	2	2	2
0,2	4	4	4	4	4	4	4	4	4	4	4
0,3	6	6	6	6	6	6	6	6	6	6	6
0,4	9	9	9	9	9	9	9	9	9	9	9
0,5	14	14	14	14	14	14	14	14	14	14	14
0,6	23	23	23	23	23	23	23	23	23	23	23
0,7	62	60	60	62	62	60	60	62	62	62	61,2
0,8	265	258	257	259	256	261	255	256	256	258	258,1
0,9	1201	1200	1203	1218	1227	1201	1232	1210	1226	1225	1214,
avg	176,2	175,1	175,3	177,4	178,1	175,5	178,3	176,2	178,0	178,1	176,8

Nombre de clusters du classement en fonction de la vigilance et du bruit.

On s'aperçoit d'abord que, pour une valeur de vigilance identique, peu importe la quantité de bruit, les nombres de clusters des réseaux sont presque identiques. De plus, jusqu'à $p = 0.6$ le nombre de clusters est exactement le même quelque soit le niveau de bruit. Passé $p = 0.6$ le nombre de cluster peut varier très légèrement à l'intérieur d'une même ligne de la table. Je doute fortement que cela soit lié à la quantité de bruit ou à la vigilance. En effet, les clusters "supplémentaires" sont créés suite aux fluctuations (voir section 7.d.iv.) ou bien à cause du caractère aléatoire du bruitage : les bits des formes qui sont permutés pour créer du bruit sont sélectionnés aléatoirement et ainsi le classement ne se déroulera presque jamais deux fois pareil en utilisant les même valeurs de paramètres car dans les réseaux de neurones ce sont les données qui dirigent le classement. C'est aussi pour cette raison que l'ordre de présentation des données fait sens. Le graphique

suivant montre la répartition des nombres de clusters en fonction de la vigilance et du bruit :



Nombre de clusters du classement en fonction de la vigilance et du bruit.

On distingue facilement que le nombre de clusters est toujours plus élevé d'un niveau de vigilance au suivant.

On peut conclure, concernant le nombre de clusters, que plus le paramètre vigilance est élevé, plus l'algorithme va créer des clusters. De plus, si on trace la courbe du nombre de cluster on voit que sa forme ressemble à celle d'une fonction exponentielle. Le nombre de clusters influent également sur le temps des classements...

iii. Temps des classements

Voici une table qui, d'après les temps observés, permet de distinguer les temps acceptables des temps jugés trop élevés.

temps	appréciation
0 - 1 seconde	Très rapide
1 - 3 secondes	rapide
3 - 5 secondes	moyen
5 - 20 secondes	long
20 + secondes	Trop long

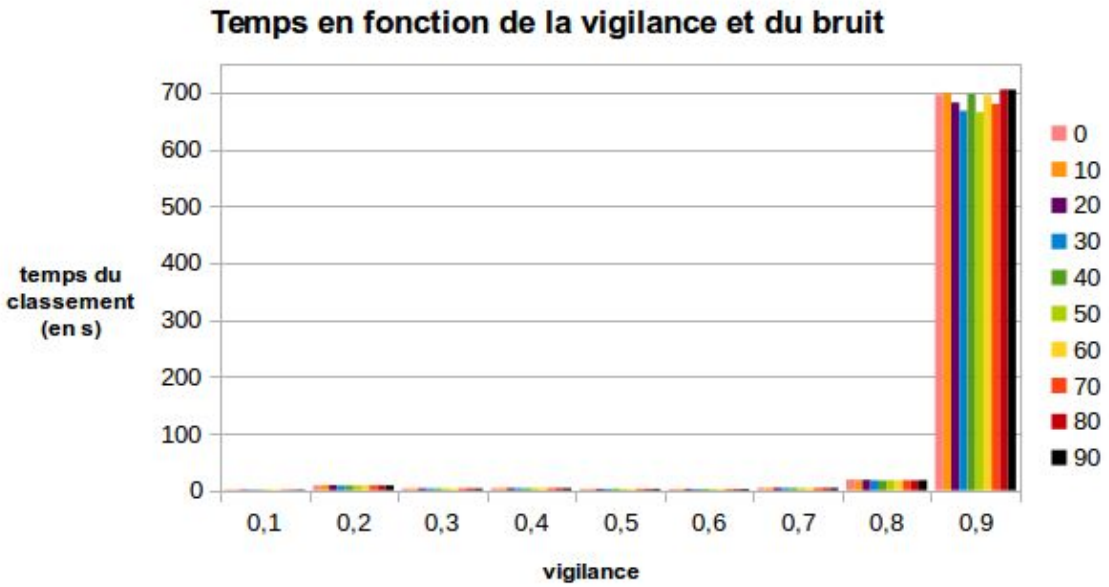
Table des fourchettes de temps.

Et voici la table qui regroupe les temps des 90 simulations.

p\ns	0	10	20	30	40	50	60	70	80	90	avg
0,1	1,211	1,336	1,253	1,171	1,174	1,184	1,207	1,195	1,205	1,201	1,213
0,2	8,142	8,732	8,668	7,979	8,129	7,918	8,234	8,346	8,245	8,205	8,259
0,3	3,13	3,138	3,21	3,049	3,079	3,177	3,078	3,177	3,186	3,094	3,131
0,4	3,537	3,79	3,812	3,551	3,43	3,495	3,57	3,505	3,585	3,546	3,582
0,5	2,463	2,529	2,523	2,531	2,559	2,33	2,405	2,462	2,435	2,404	2,464
0,6	2,185	2,306	2,294	2,202	2,212	2,182	2,18	2,178	2,184	2,169	2,209
0,7	4,022	3,963	3,952	3,85	3,857	3,717	3,778	3,885	3,869	3,931	3,882
0,8	18,81	18,23	18,09	17,34	16,97	17,67	17,23	17,44	17,25	17,46	17,65
0,9	695,2	699,8	682,0	667,5	697,2	665,0	695,2	679,8	704,9	704,8	689,1
avg	82,08	82,65	80,64	78,80	82,06	78,52	81,87	80,22	82,99	82,98	81,28

Temps du classement en fonction du la vigilance et du bruit.

On voit, d'après la table ci-dessus, que les temps, pour une même valeur de vigilance, peu importe le bruit, sont quasiment identiques. Cela conforte l'hypothèse : la création et la modification de clusters et ce qui coûte le plus de temps de calcul à l'algorithme mais lors de la phase de test on ne créé aucun cluster, pas plus qu'on en modifie leur prototype. En revanche, si vous bruitez le jeu de données d'apprentissage (il n'y a pas trop d'intérêt de travailler avec un réseau bancale, je suis d'accord) vous constaterez que le temps augmente en même temps que le bruit car le réseau, pour s'adapter, et forcé de créer d'avantages de clusters (j'ai fait des tests et on peut monter à plus de 10000 clusters pour interner les 8124 formes du jeu de données original avec une vigilance de 0.9, et la simulation dure plusieurs heures...). Les temps enregistrés pour les 90 simulations sont réunis dans le graphique suivant :



Temps du classement en fonction du la vigilance et du bruit.

Les temps des simulations pour $p = 0.8$ et $p = 0.9$ sont particulièrement élevés ce qui correspond probablement au fait que le nombre de clusters créés dans ces simulations est lui-aussi nettement plus important qu'avec $p < 0.8$. Cependant, comment expliquer que pour $p = 0.2$ les temps soient eux aussi particulièrement élevés ? Cela ne se voit pas forcément dans le graphique ci-dessus car il est fortement étiré par les valeurs des temps avec $p = 0.9$. Toutes les simulations effectuées avec une vigilance de 0.2 ont créé 4 clusters ce ne peut donc pas être le nombre de clusters qui affecte le temps de ces simulations. À mon avis il s'agit plutôt de la fluctuation de ces clusters, comme expliqué dans la prochaine section...

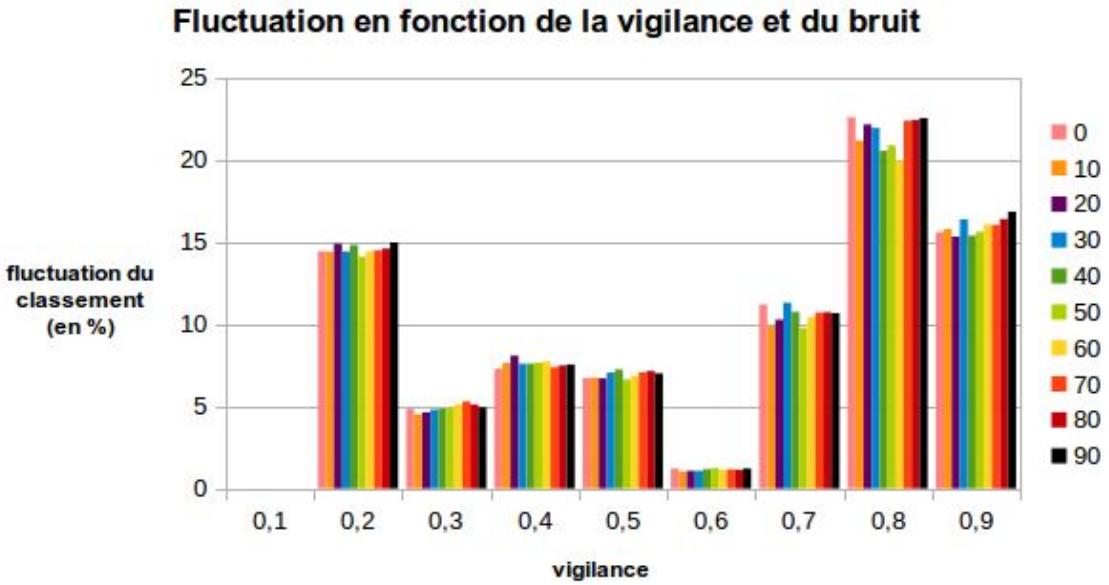
iv. Fluctuation des classements

Comme expliqué plus haut, la fluctuation des clusters correspond aux nombre de formes qui sont réassignées à des clusters différents entre deux passes de l'algorithme. Les fluctuations des simulations ont été enregistrées et regroupées dans la table suivante :

vigil	0	10	20	30	40	50	60	70	80	90	avg
0,1	0	0	0	0	0	0	0	0	0	0	0
0,2	14,41	14,38	14,87	14,40	14,79	14,09	14,41	14,46	14,58	14,95	14,53
0,3	4,817	4,506	4,621	4,784	4,833	4,931	5,111	5,274	5,094	4,931	4,890
0,4	7,266	7,625	8,066	7,576	7,576	7,642	7,707	7,364	7,478	7,527	7,583
0,5	6,711	6,727	6,694	7,037	7,233	6,629	6,825	7,054	7,135	6,988	6,903
0,6	1,208	1,028	1,061	1,061	1,175	1,224	1,126	1,159	1,126	1,208	1,138
0,7	11,18	9,846	10,25	11,28	10,72	9,748	10,40	10,69	10,71	10,64	10,55
0,8	22,56	21,14	22,12	21,91	20,52	20,85	19,88	22,35	22,40	22,51	21,62
0,9	15,57	15,77	15,30	16,36	15,36	15,61	16,06	16,01	16,37	16,81	15,92
avg	9,305	9,004	9,222	9,380	9,137	8,970	9,060	9,376	9,434	9,510	9,240

Fluctuation du classement en fonction du la vigilance et du bruit.

On en déduit que pour $p = 0.1$ il n'y a pas de fluctuation car il n'y que deux clusters. Les formes sont classées au bon endroit dès la première itération. Les valeurs à l'intérieur de chaque ligne sont quasiment identiques, je pense que le peu de différence est lié, comme pour le nombre de clusters, au fait que le bruitage est aléatoire. Le graphique suivant permet de déduire d'avantage de choses :



Fluctuation du classement en fonction du la vigilance et du bruit.

On voit très nettement que les fluctuations les plus importantes interviennent dans les clusters des simulation lorsque p = 0.2, 0.8 et 0.9. Et c'est justement avec ces valeurs de vigilance que les temps de simulation sont les plus élevés. C'est pourquoi j'en déduis que le temps d'une simulation dépend à la fois du nombre de clusters et aussi de la fluctuation à l'intérieur de ces clusters. Cela expliquerait pourquoi les temps des simulations avec p = 0.2 sont plus élevés qu'avec p = 0.3 à 0.7 bien que le nombre de clusters soit nettement inférieur avec p = 0.2. Il n'y a peut-être pas besoin de créer autant de clusters mais il faut effectuer plus de calculs afin de réassigner les formes d'un cluster à un autre.

8. Conclusions

Ce rapport a présenté un type de réseau de neurone appelé Adaptive Resonnance Theory 1 capable de classer et reconnaitre des espèces de champignons. Une base de données, ou plutôt un jeu de données de 8124 espèces a été utilisé afin de tester les performances du simulateur d'ART1 créé à l'occasion de ce projet. Le programme est capable de classer des espèces de champignons inconnues après en avoir appris un premier jeu. La précision du classement est très élevée (> 90%) avec une vigilance comprise entre 0.6 et 0.9 et lorsque la quantité de bruit des formes présentées n'excède pas 20%. La précision reste acceptable (> 65%) avec une vigilance comprise entre 0.5 et 0.9 et une quantité de bruit ne dépassant pas 40%. Le réseau est totalement inefficace (précision < 35%) avec une vigilance inférieur à 0.4. Un bruitage des formes supérieur à 60% ne permettra jamais d'atteindre une fiabilité élevée (> 80%), peu importe la vigilance.

Le meilleur résultat est atteint avec $p = 0.9$ et sans ajouter de bruit. En utilisant ces paramètres vous aurez donc environ une chance sur 100 d'obtenir un mauvais classement et donc de manger un champignon vénéneux ou de passé à côté d'un excellent champignon comestible...

Enfin, je pense que les attributs inconnus ont une part de responsabilité certaine dans le pourcentage d'erreurs de classement.

N'hésitez pas à tester le programme, avec le jeu de données "mushrooms" et même avec d'autres jeux de données. Le programme fonctionne avec n'importe quel jeu de données binaire respectant le format décrit dans la section 4. et dans le fichier README. Aussi, n'hésitez pas à lire ce fichier README qui contient beaucoup d'informations techniques complémentaires à la documentation HTML (*/doc/index.html*).

9. Références

[1] https://en.wikipedia.org/wiki/Adaptive_resonance_theory

[2] Grossberg, S. (1987), Cognitive Science (Publication), 11, 23-63, Competitive Learning: From Interactive Activation to Adaptive Resonnance http://www.cns.bu.edu/Profiles/Grossberg/Gro1987CogSci.pdf

[3] Gail A. Carpenter, Stephen Grossberg (1998), Adaptive Resonance Theory, Department of Cognitive and Neural Systems http://cns.bu.edu/Profiles/Grossberg/CarGro2003HBTNN2.pdf

[4] http://www.scholarpedia.org/article/Adaptive_resonance_theory

[5] The Link between Brain Learning, Attention, and Consciousness, Stephen Grossberg (1999), Consciousness and Cognition 8, 1–44
http://www.bowdoin.edu/~echown/courses/355/grossberg.pdf

[6] Carpenter, G.A.and Grossberg, S. (1987), ART2: Self-organization of stable category recognition codes for analog input patterns, Applied Optics, 26 (23): 4919 - 4930

[7] Flávio Henrique Teles Vieira, Luan Ling Lee (2007), A Neural Architecture Based on the Adaptive Resonant Theory and Recurrent Neural Networks, International Journal of Computer Science & Applications, Vol. 4 Issue 3, pp 45-56
http://www.tmrfindia.org/ijcsa/v4i34.pdf

[8] H. Glotin, P. Warnier, F. Dandurand, S. Dufau, B. Lété, C. Touzet, J.C. Ziegler, J. Grainger, An Adaptive Resonance Theory account of the implicit learning of orthographic word forms (2009), Journal of Psychology, Paris
http://gsite.univ-provence.fr/gsite/Local/lpc/dir/grainger/glotin.pdf