Master 1 : CRYPTIS Informatique

Rapport projet IA 2

Réalisé par :
Jean DE BONFILS
Sarra JLASSI

Version du
25 avril 2021

Table des matières

1 Important

Nous n'étions pas au courant que le rapport portait seulement sur la partie 2 du projet, nous avons donc bien fait le rapport sur la partie 2 (mois de 15 pages comme demandé). Cependant, nous incluons également la partie sur l'algorithme ACO que nous avions déjà faites avant de savoir que le rapport devait porter uniquement sur la partie 2.

2 Partie 1 : Algorithme ACO et Calcul du plus court chemin

2.1 Principe de l'algorithme de colonies de fourmis

Un algorithme de colonies de fourmis est un algorithme itératif à population, inspiré du comportement des fourmis ou d'autres espèces et qui constituent une famille de métaheuristiques d'optimisation. Ils partagent un savoir commun qui leurs permettent de guider leurs futurs choix et d'indiquer aux autres, des directions à suivre. Cette méthode a pour but de construire les meilleures solutions à partir des éléments qui ont été exploré par d'autres fourmis. Chaque fois qu'une fourmi découvre une solution au problème, elle enrichit la connaissance collective de la colonie. Ainsi, chaque fois qu'une nouvelle fourmi aura à faire des choix, elle pourra s'appuyer sur la connaissance collective pour pondérer ses choix.

2.2 Explication du fonctionnement du programme

2.2.1 Structure de données

Tout d'abord, nous avons modélisé l'espace de déplacement de la colonie de fourmis par un **graphe non orienté pondéré** représenté par une matrice de taille carré qui peut être converti en fichier dot par la suite. Pour connaître le poids d'une arrête entre un noeud X et un noeud Y, il suffit de regarder la valeur de la case de la matrice à la ligne X, colonne Y. Puisque le graphe est non orienté, la valeur à la colonne Y, ligne X est la même que la valeur à la colonne X, ligne Y. Cela se traduit donc, dans la matrice, par une symétrie axiale. Un poids de -1 dans la matrice signifie qu'il n'existe pas d'arête entre les deux noeuds. Représenter le graphe sous la forme d'une matrice permet de calculer rapidement et efficacement un certains nombre d'opérations sur le graphe même

si cette structure de données ne permet pas de facilement visualiser le graphe. C'est pourquoi cette matrice peut, par la suite, être convertie en .dot puis en image du graphe au format .png.

Exemple d'une matrice valide :

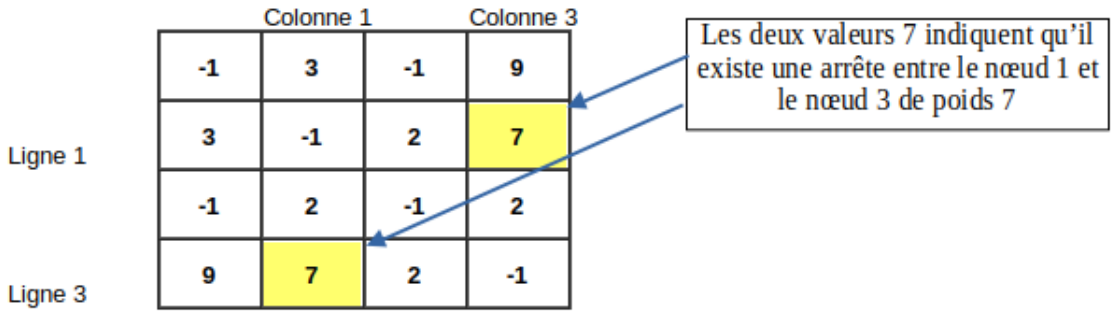


FIGURE 1 – Matrice représentant un graphe non orienté pondéré

Par exemple, cette matrice nous indique par exemple qu'il n'existe pas de lien entre le noeud 0 et le noeud 2 ou que l'arrête entre les noeuds 0 et 1 vaut 3. Concrètement, cette matrice est une liste à deux dimensions en Python : [[-1,3,-1,9],[3,-1,2,7],[-1,2,-1,2],[9,7,2,-1]]

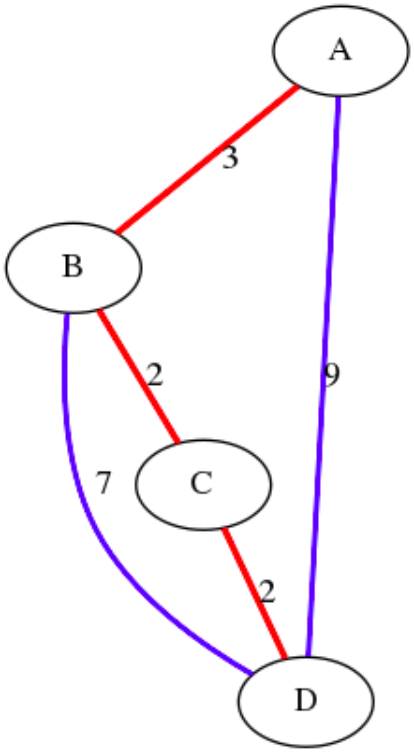


FIGURE 2 – Matrice modélisée par le graphe ci-dessus

Pour cela, nous avons donc défini une classe **Graph** comportant deux variables membres : la matrice qui représente les liens entre les noeuds ainsi que les poids associés **matDistances** et la matrice **matPheromones** initialisée à 0 qui représente la quantité de phénomène présente sur les liens entre les noeuds. Nous avons aussi implémenté deux fonctions permettant de vérifier la validité de la matrice et corriger les éventuelles erreurs dans la matrice :

- Une fonction **matValide(self)** qui retourne faux si la matrice n'est pas de taille carré (2x2 , 3x3 , 4x4) et si une arête de A vers B n'a pas le même poids que le lien de B vers A et retourne vrai sinon .
- Une procédure **corrigerErreur(self)** permettant de corriger les éventuelles erreurs dans la matrice, elle vérifie donc, pour chaque lien, s'il existe bien un lien de A vers B ayant le même poids qu'un lien de B vers A.

2.2.2 Implémentions de ACO

L'algorithme de colonies de fourmi est implémenté à travers la fonction shortestPathACO dans la classe Graphe. Cet objet est composé de deux matrices,

l'une représentant la distance entre les noeuds et l'autre la quantité de phéromones entre les noeuds (sur les arrêtes donc). La fonction shortestPath ACO prend en paramètre le noeud de départ et d'arrivée, le nombre de fourmis ainsi que les paramètres de réglage Alpha, Bêta, Gamma et Rhô.

Nous avons aussi implémenté un objet "Fourmi" composé d'une liste des noeuds visités ainsi que d'un entier correspondant à la distance parcourue par la Fourmi. Le dernier élément de cette liste permet de savoir sur quel noeud se trouve la fourmi. L'algorithme initialise, tout d'abord, les NBFOURMIS au point de départ. Pour chaque fourmi, elle "avance" tant qu'elle n'est pas arrivée à la destination. La méthode 'avancer' permet à la fourmi de choisir un prochain noeud grâce aux chemins disponibles par rapport au noeud sur lequel elle se trouve et à la règle aléatoire de transition proportionelle(on appellera cette règle P).

$$p_{ij}^{k}(t) = \begin{cases} \dfrac{\gamma+\tau_{ij}(t)^{\alpha}\cdot\eta_{ij}^{\beta}}{\sum_{l\in J_i^k}(\gamma+\tau_{il}(t)^{\alpha}\cdot\eta_{il}^{\beta})} & \text{si } j \in J_i^k \\ 0 & \text{si } j \notin J_i^k \end{cases}$$

FIGURE 3 – La règle aléatoire de transition proportionnelle

où l'on définit **J(k,i)** comme étant la liste des déplacements possibles pour une fourmi k lorsqu'elle se trouve sur une ville i, η_{ij} la visibilité, qui est égale à l'inverse de la distance de deux villes i et j (Constante/dij), $\tau_{ij}(t)$ l'intensité des phéromones sur l'arrête ij et **Alpha ,Bêta et Gamma** trois paramètres contrôlant l'importance relative de l'intensité et de la visibilité d'une arête (voir 2.2.3).

Cette règle P est calculée pour chaque noeud sur lequel la fourmi peut se déplacer (sauf le noeud où elle se trouvait au tour précédent, puisqu'un aller-retour sur un noeud est inutile et ne contribuerait pas à trouver un chemin optimal). En fonction des valeurs calculées pour chaque noeud grâce à cette règle P, on réalise une roulette russe pour choisir un noeud en fonction de la valeur de cette règle pour les noeuds disponibles en ajoutant de l'aléa. En effet, grâce à la roulette Russe, un noeud qui a une mauvaise valeur fournie par la règle P peut quand même être choisi mais avec une chance moindre. Le noeud choisi est ensuite ajouté à la liste des noeuds parcourus par la fourmi et la distance parcourue de celle-ci est mis à jour. Cette distance permettra de calculer la quantité de phéromones à déposer sur les arêtes empruntées.

Une fois arrivée à destination, on ne dépose pas de suite les phéromones pour ne pas biaiser l'algorithme. On fait donc avancer la fourmi suivante. Une fois que toutes les fourmis sont arrivés à destination, les fourmis déposent chacune une quantité de phéromones sur le chemin emprunté grâce à la fonction rependrePheromone() qui ajoute une certaine quantité de phéromones à la matrice des phéromones seulement sur les arêtes visitées. La quantité de phéromones déposées est proportionnelle à la qualité du chemin emprunté. Ensuite, chaque fourmi est renvoyée au point de départ ("réinitialiser"). Enfin, on simule une évaporation de Rhô sur la matrice représentant les quantités de phéromones sur chaque arête pour oublier, au fur et à mesure, les chemins qui ne seraient pas de bonne qualité.

On répète tout ceci un certain nombre de fois, cela dépendant d'un paramètre fixé par l'utilisateur dans le fichier main.

```
#Toutes les fourmis font nbGenerations aller-retour
for i in range(0,nbGeneration):
    #Toutes les fourmis font un "aller-retour"
    for fourmis in listFourmis:
        #Tant que la fourmi n'est pas arrivée à destination, elle avance
        while(fourmis.getNoeudCourant() != fin) :
            fourmis.avancer(self,alpha,beta,gamma)

    #Une fois que toutes les fourmis sont arrivées, elles répendent chacune leu
    for fourmis in listFourmis :
        self.matPheromones = fourmis.rependrePheromone(self.matPheromones)
        fourmis.reinitialiser(debut)
    #On simule un evaporation
    self.evaporation(pho)
```

FIGURE 4 – Principale partie de l'algorithme ACO

Une fois que toutes les fourmis sont arrivées à destination, on calcul le meilleur chemin du départ jusqu'à l'arrivée, c'est à dire le chemin ayant la plus forte quantité de phéromones. Ce chemin peut facilement être retrouvé grâce à la matrice de phéromones qui représente la quantité de phéromones entre chaque noeud.

Matrice représentant la quantité de phéromones entre les nœuds

	A	B	C	D
A	-1	30	-1	0,5
B	30	-1	0,2	28
C	-1	0,2	-1	27
D	0,5	28	27	-1

Nœud de départ : A

Nœud d'arrivé : C

FIGURE 5 – Exemple d'une matrice représentant l'intensité de phéromones sur les arêtes

Voici un exemple typique de matrice représentant la quantité de phéromones sur les différentes arrêtes. Dans ce cas là, pour calculer le plus court chemin, on part de A qui est le point de départ. On cherche, vers quel noeud la piste de phéromones est la plus forte. On prend donc la colonne dans laquelle se trouve la valeur maximale sur la ligne A. On obtient donc B. Puis sur la ligne B, on cherche à nouveau le max de la ligne en excluant la colonne A (puisque c'est le noeud d'où l'on vient), on obtient D, de même on cherche le maximum sur la ligne en excluant le dernier noeud (la colonne B). On obtient C qui est le point d'arrivée. On a donc, finalement, le chemin A -> B -> D -> C en suivant la piste de phéromones.

2.2.3 Règle aléatoire de transition proportionelle et importance des paramètres Alpha, Bêta, Gamma, Rhô

$$p_{ij}^{k}(t) = \begin{cases} \frac{\gamma+\tau_{ij}(t)^{\alpha}\cdot\eta_{ij}^{\beta}}{\sum_{l\in J_i^k}(\gamma+\tau_{il}(t)^{\alpha}\cdot\eta_{il}^{\beta})} & \text{si } j \in J_i^k \\ 0 & \text{si } j \notin J_i^k \end{cases}$$

FIGURE 6 – La règle aléatoire de transition proportionnelle

Les paramètres pris en compte pour l'algorithme de colonies de fourmis sont très importants, en particulier dans le cas de graphes assez spéciaux. Le paramètre Gamma introduit une probabilité, pour une fourmi, d'explorer un nouveau noeud ne **semblant** pas, selon la règle P, un bon choix (par exemple peu de phéromones ou une distance au prochain noeud très élevée). En effet, ce paramètre est indispensable pour découvrir de nouveaux chemins potentiellement plus court que ceux explorés jusqu'à lors. Le paramètre Rhô compris entre 0 et 1 correspond à un taux d'évaporation des phéromones sur toutes les arrêtes du graphe. Il permet notamment d'oublier au fur et à mesure les chemins de moins bonnes qualités. Le paramètre Bêta influe sur la visibilité ij d'un chemin. Plus ce paramètre Bêta sera élevé, plus une fourmi aura tendance à se fier à un chemin dont le prochain point est proche. Le paramètre Alpha influe sur la variable $\tau_{ij}(t)$ qui correspond à l'intensité de phéromones sur l'arrête ij au moment où la t ème fourmis arrive à la destination. Dans le programme, cette valeur τ_{ij} est calculée au fur et a mesure du passage des fourmis et peut être récupérée grâce à la matrice de phéromones : matPhéromones[i][j]. On peut donc définir la formule de récurrence suivante pour calculer $\tau_{ij}(t)$:

$$\tau_{ij}(t) = (1-\rho) * [\tau_{ij}(t-1) + L(t-1)]$$

- Avec t, la fourmi arrivée en t ème position
- $\tau_{ij}(t)$, l'intensité de la piste ij pour la fourmi arrivée en t ème position.
- ρ, le paramètre d'évaporation
- $L(t)$, la fonction calculant la quantité de phéromones à déposer par la fourmi arrivée en t ème position et qui dépend de la distance parcourue par cette même fourmi t

Plus le paramètre Alpha sera élevé, plus la fourmi aura tendance à se fier à l'intensité des phéromones et aura donc moins tendance à explorer un nouveau

chemin ou à en choisir un en fonction de sa distance à la prochaine ville. Si le paramètre Bêta est trop élevé par rapport à Alpha, alors la fourmi aura tendance à plus se fier à la distance jusqu'au prochain noeud et moins à la piste de phéromones. La fourmi privilégiera donc plus, les arêtes de faibles coût ce qui peux donner un chemin optimal dans **certains** cas comme le démontre les algorithmes gloutons. En fonction du graphe et des données en générales, les paramètres doivent être modifiés pour obtenir un chemin optimal. De manière général l'algorithme de colonies de fourmis utilise une part d'aléatoire avec la règle aléatoire de transition proportionelle et la roulette Russe pour choisir le chemin à emprunter. Il est donc possible que le chemin obtenu ne soit pas optimale. Augmenter le nombre de fourmis dans l'algorithme permet de réduire l'effet d'aléa.

2.2.4 Utilisation du programme python et exemples d'exécutions

Le programme exécute l'algorithme grâce à une matrice passé en paramètre représentant le graphe et se situant dans le fichier main. Cette matrice peut facilement être modifié notamment avec les exemples de matrices donnés en commentaires.

L'utilisateur à la possibilité d'indiquer le noeud de départ et d'arrivée directement en entré du programme comme ceux ci :
python3 main.py -s 0 -d 3 ou encore python3 main.py -s A -d D

Dans ce cas là on recherche le plus court chemin de 0 (A), la source, à 3 (D), la destination. Si l'utilisateur ne rentre pas ces paramètres alors des noeuds par défaut sont choisis. En sortie, le programme affiche d'abord la matrice de façon visible sur le terminal, puis affiche un message comme quoi la matrice est bien valide. Enfin, le plus court chemin entre les noeuds choisis est affichée. En l'occurrence, le chemin le plus court entre A et D est : A -> B -> E -> C -> D

```
Correction des eventuelles erreurs
 | -1  2  8  12  7   |

 | 2  -1  7  8  3   |

 | 8  7  -1  2  1   |

 | 12  8  2  -1  4   |

 | 7  3  1  4  -1   |

Matrice valide


graphe.png généré. Ouvrez le pour visualiser le graphe et son plus court chemin

Chemin le plus court entre A et D

['A', 'B', 'E', 'C', 'D']
```

FIGURE 7 – Résultat retourné par le programme dans le terminal

Observer le graphe sous forme de matrice n'est pas très pratique et visuel, c'est pourquoi le programme converti la matrice en un fichier .dot. Dot est un format de fichier et surtout un langage de balisage permettant de décrire un graphe. Après avoir créé le fichier .dot, le programme génère une image (appelé graphe.png) du graphe à partir de ce fichier dot.

```
strict graph  {
        A -- B [penwidth = 2.5, color=red,label=2,weight=2];
        A -- C [penwidth = 2,color=blue,label=8,weight=8];
        A -- D [penwidth = 2,color=blue,label=12,weight=12];
        A -- E [penwidth = 2,color=blue,label=7,weight=7];
        B -- A [penwidth = 2.5, color=red,label=2,weight=2];
        B -- C [penwidth = 2,color=blue,label=7,weight=7];
        B -- D [penwidth = 2,color=blue,label=8,weight=8];
        B -- E [penwidth = 2.5, color=red,label=3,weight=3];
        C -- A [penwidth = 2,color=blue,label=8,weight=8];
        C -- B [penwidth = 2,color=blue,label=7,weight=7];
        C -- D [penwidth = 2.5, color=red,label=2,weight=2];
        C -- E [penwidth = 2.5, color=red,label=1,weight=1];
        D -- A [penwidth = 2,color=blue,label=12,weight=12];
        D -- B [penwidth = 2,color=blue,label=8,weight=8];
        D -- C [penwidth = 2.5, color=red,label=2,weight=2];
        D -- E [penwidth = 2,color=blue,label=4,weight=4];
        E -- A [penwidth = 2,color=blue,label=7,weight=7];
        E -- B [penwidth = 2.5, color=red,label=3,weight=3];
        E -- C [penwidth = 2.5, color=red,label=1,weight=1];
        E -- D [penwidth = 2,color=blue,label=4,weight=4];
}
```

FIGURE 8 – Exemple d'un fichier dot généré par le programme

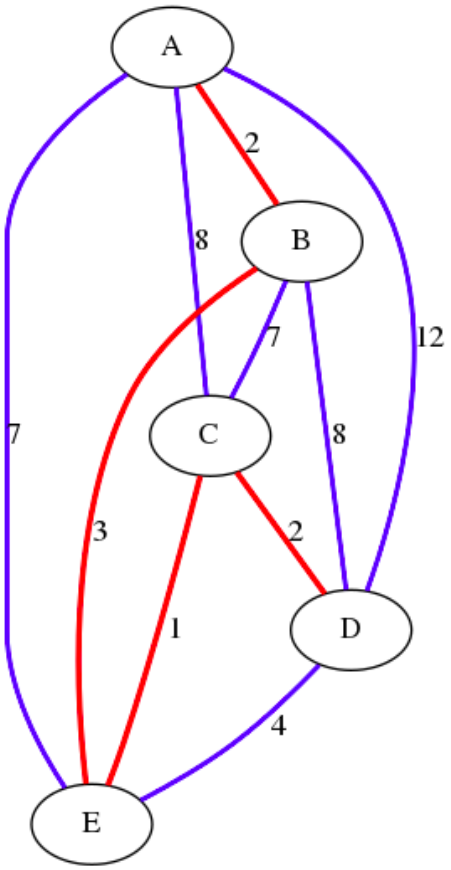


FIGURE 9 – Fichier image obtenu en sortie du programme

Le fichier image généré, permet de visualiser le graphe plus facilement qu'en observant seulement la matrice ainsi que de vérifier le plus court chemin obtenu s'affichant en rouge. Dans ce cas là, le programme nous a bien donné le chemin optimal entre les deux noeuds. Après avoir testé le programme sur de nombreux graphes, celui-ci semble donner un résultat optimal à chaque fois. Évidement, pour les graphes assez spéciaux des ajustements peuvent être nécessaires au niveau des paramètres Alpha, Bêta, Gamma. Le programme fonctionne pour tout types de graphes pondérés non orientés même ceux qui ne sont pas complets. Cependant, le graphe doit obligatoirement être connexe (en effet, si le graphe est en deux parties, il n'existe pas forcément de chemin possible entre deux noeuds)

2.3 Analyse des résultats obtenus

2.3.1 Cas d'un graphe complet à six noeuds

Voici, ci-dessous, le cas d'un graphe complet à six noeuds dont le chemin optimal entre A et E est de taille 5. Bien que le programme fonctionne pour tout types de graphe non orienté pondéré connexe, il est plus intéressant de travailler sur des graphes complets puisque ceux-ci offrent de nombreuses solutions de chemins possibles pour une fourmi. En effet, à chaque tour, la fourmi a le choix d'aller sur n'importe quel noeud du graphe à part le noeud courant et les noeuds déjà visités.

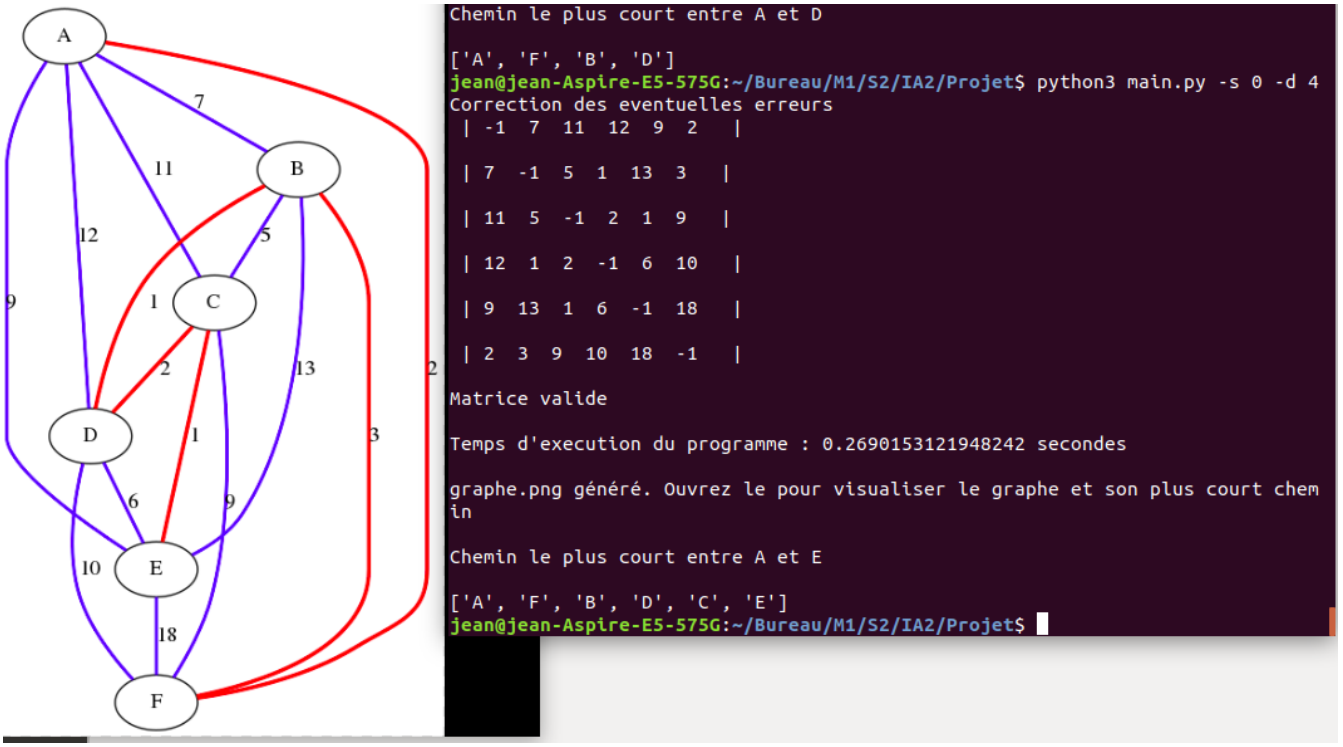


FIGURE 10 – Plus court chemin obtenu pour un graphe complet à six noeuds

2.3.2 Cas d'un graphe avec deux chemins optimaux

Le cas d'un graphe où deux chemins sont optimaux sont très intéressants pour constater l'importance des paramètres tels que Alpha et Bêta. En effet en fonction de ces paramètres, on aura plus souvent une solution que l'autre.

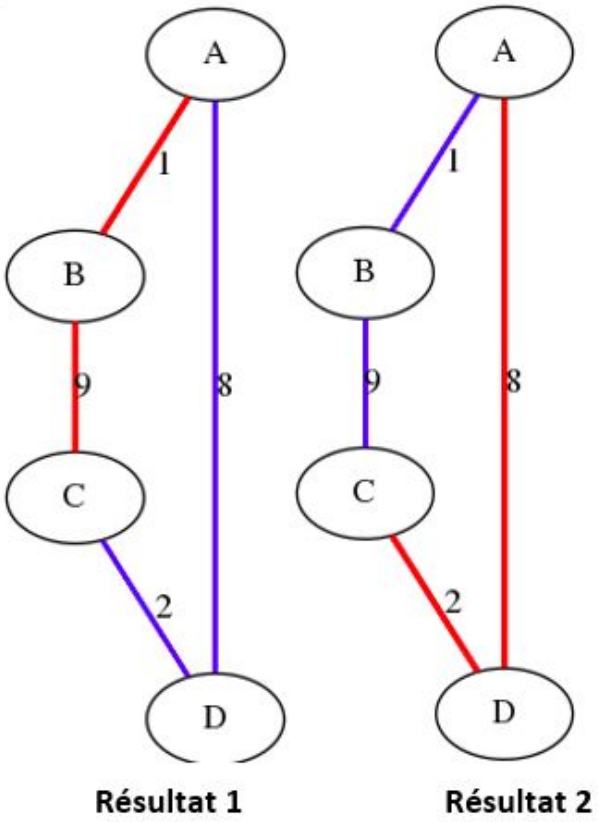


FIGURE 11 – Résultat de deux exécutions avec des paramètres Alpha, Bêta différents

Dans le cas où l'on choisit un Bêta très faible on obtient les deux résultats de façon pratiquement équiprobable. A cause de la faible valeur de Bêta, l'algorithme ne se base pratiquement pas sur la visibilité du prochain noeud et donc les deux solutions ont pratiquement la même chance d'être choisi. Cependant, dès qu'on augmente ce paramètre significativement, le programme retourne le résultat 1 à chaque fois. Cela s'explique par le fait que la fourmi, dans ce cas là, va beaucoup plus privilégier le choix où la distance au prochain noeud est faible (une visibilité élevée) ce qui explique qu'elle choisisse d'aller sur B (poids de 1) plutôt que sur D(poids de 8) à chaque fois.

2.3.3 Analyse des temps d'exécutions

Pour finir, nous pouvons constater que le temps d'exécution varie en fonction du nombre de noeuds et de la taille du chemin obtenu, cependant les temps de calculs ne semble pas exponentiels pour des graphes complets. Cela s'explique

par le fait qu'une fourmi ne passe jamais deux fois par le même noeud dans un graphe complet puisqu'elle a toujours le choix d'aller sur un noeud où elle n'est pas déjà allée durant son trajet. Au plus, la fourmi parcourra donc les N noeuds du graphe.

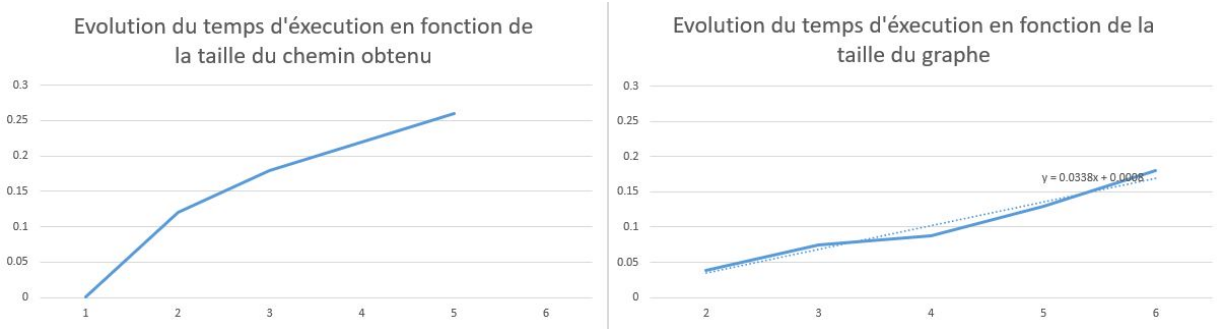


FIGURE 12 – Temps d'exécutions

3 Partie 2 : Apprentissage supervisé - Construction d'un Anti-Spam

3.1 Réseau de neurones avec Keras

Tout d'abord nous avons commencé par développer une fonction permettant de récupérer les données sous forme de listes à partir d'un fichier csv :

def getXYData(fic,maxcol=-1,maxlines=-1,indLabel=0,delimiter=";")

Cette fonction prend en paramètre le chemin d'un fichier csv, le nombre maximum de colonnes (attributs) ainsi que de lignes à prendre en compte (-1 = toutes les lignes, colonnes), l'indice de la colonne correspondant au label (la colonne "SPAM" dans notre cas) et un délimiteur permettant séparer les colonnes. La fonction retourne une liste 2D X contenant pour chaque tuple les attributs qui vont nous permettre de prédire les spams et une autre liste 2D Y correspondant au label de chaque tuple. Ici, il s'agit d'une classification binaire, les valeurs de Y sont donc 0 ou 1. Le programme va donc récupérer X_train, Y_train, X_test, Y_test, X_pred, Y_pred. Les listes X sont converties au format numpy.array et les liste Y au format One-Hot grâce à : **to_categorical(Y_train, 2)**. Le deuxième paramètre indique la taille en sortie pour chaque tuple. Ici la taille est de 2, dans ce cas-là un [1] sera convertie en [0. 1.] et 0 en [1. 0.]. Une fois que toutes les données sont extraites et prêtes à être utilisées, on crée le réseau de neurones et on ajoute seulement une fonction d'activation :

model.add(Dense(2, input_dim=nbColonnes, activation='sigmoid'))

En entrée, la dimension doit être de la taille d'un tuple de X, c'est-à-dire le nombre de colonnes dans le fichier csv mois un pour la colonne correspondant au label. La forme de sortie doit être de 2 pour correspondre à une classification binaire avec le format One-Hot. Après avoir testé plusieurs fonctions d'activations, il semblerait que la Sigmoid soit la plus adaptée à une classification binaire. En effet, nous aurions aussi pu utiliser Softmax, cependant si on se penche sur les formules de ces deux fonctions d'activations, on remarque que la fonction Sigmoid est équivalente à Softmax quand K=2. Donc, dans le cas d'une classification binaire, il vaut mieux utiliser Sigmoid. Enfin, on compile le modèle avec l' "optimzer" "Adam" qui est connu pour ses performances. Toutefois, l'optimizer "rmsprop" donne pratiquement les mêmes résultats. On utilise la fonction de loss

"binary_crossentropy" qui est adaptée à une classification binaire. En métrique on choisit "Accuracy" qui correspond au taux de prédictions correctes.

L'entraînement peux commencer :

model.fit(X_train, Y_train,epochs=epochs,verbose=1,validation_data=(X_test,Y_test))

On utilise les listes de trains pour entraîner le modèle et les listes de tests pour la validation. Le choix du nombre d'époque sera expliqué ultérieurement. Un résultat de l'entraînement s'affiche :

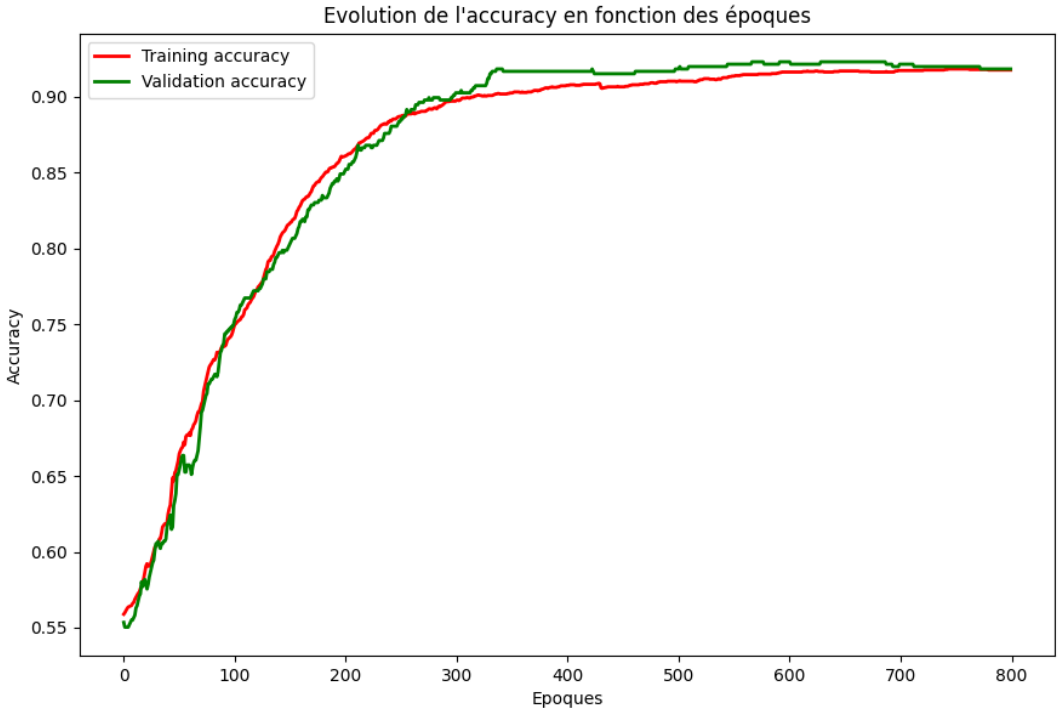
```
93/93 [==============================] - 0s 1ms/step - loss: 0.2258 - accuracy: 0.9406 - val_loss: 0.2014 - val_accuracy: 0.9418
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 2)                 116
=================================================================
Total params: 116
Trainable params: 116
Non-trainable params: 0
_________________________________________________________________
Evaluation terminée :

Test loss: 0.22497662901878357
Test accuracy: 0.9330417513847351
```

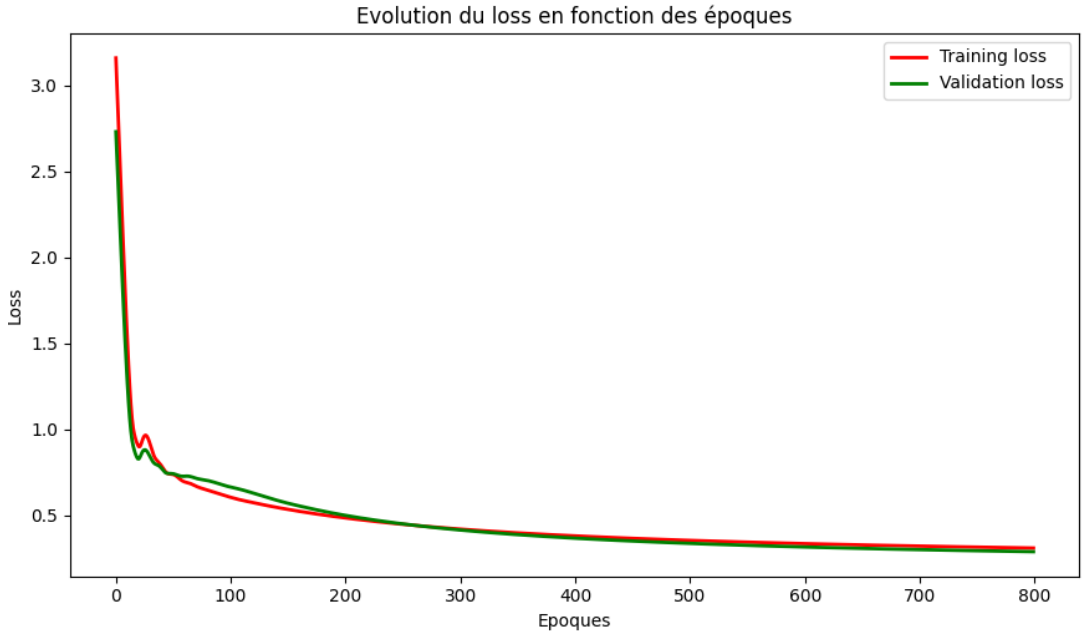
On remarque que l'Accuracy, c'est-à-dire le taux de réponses correctes est de 0.94 (pour 250 époques) ce qui paraît plutôt satisfaisant. Le Loss est une valeur donnée par une fonction et qui indique si la prédiction du label est proche ou non de la réalité. On essaie donc de minimiser cette valeur. D'autre part, on constate que le Loss et l'Accuracy entre les données d'entraînements et de validations sont très proches. Cela signifie que le réseau ne sur apprend pas et réussi à appliquer l'entraînement à de nouvelles données. Le programme affiche ensuite deux graphiques grâce à Matplot résumant l'évolution du Loss et de l'Accuracy en fonction des époques. Pour finir, le programme essaie de prédire une nouvelle source de données jamais vue.

model.predict_classes(np.array(X_pred))

```
Nombre de predictions correctes :598
Accuracy :0.9373040752351097
```

Nous avons pu choisir le nombre d'époques grâce au graphique généré avec Matplot et à l'historique généré par Keras.


Evolution de l'accuracy en fonction des époques
Training accuracy
Validation accuracy
Accuracy
0.90
0.85
0.80
0.75
0.70
0.65
0.60
0.55
0
100
200
300
400
500
600
700
800
Epoques


Evolution du loss en fonction des époques
Training loss
Validation loss
Loss
3.0
2.5
2.0
1.5
1.0
0.5
0
100
200
300
400
500
600
700
800
Epoques

Ces deux graphiques nous montrent l'évolution de l'Acurracy et du Loss en fonction du nombre d'époques. On constate que 300 époques semblent une bonne valeur dans ce cas-là. En effet, réaliser plus d'époques n'apporterait pas un changement significatif par rapport au temps perdu. Par contre, un nombre d'époques trop faible donne rapidement des résultats très peu satisfaisants.

3.2 Ajout d'une couche cachée

Pour le réseau de neurones avec une couche cachée, nous avons repris la base du premier programme.

Nous nous sommes d'abord demandé quelle couche, il était pertinent d'ajouter. Une couche Dropout met aléatoirement les unités d'entrée à 0 avec une certaines fréquence à chaque étape de l'entraînement. Cela permet notamment d'éviter que le modèle surapprenne ("Overfitting") pendant la phase d'entraînement. De par les résultats obtenus précédemment, il semblerait que le modèle de base ne surapprenne pas puisque les "accuracies" obtenues avec la base d'entraînement sont très proches des "accuracies" obtenues avec celles de validation. Il ne semble donc pas pertinent d'utiliser cette couche. Le "Flatten" permet de passer à une "Shape" de plus petite dimension, par exemple, de 5X3X2 à 5X6. Les "Shapes" sont des tuples représentant le nombre d'éléments d'un tableau ou d'un tensor dans chaque dimension. Dans notre cas, la Shape est de dimension 2 (nombre de tuples, taille d'un tuple) il n'est donc pas utile d'utiliser cette couche. Nous avons donc décidé d'ajouter la couche réalisant la fonction d'activation "Tanh" :

model.add(Dense(180, input_dim=nbColonnes,activation='tanh'))

Après avoir testé plusieurs fonctions d'activation (Relu, Softsing, Selu, Elu, Sigmoid) il semblerait que la fonction Tanh soit la plus adaptée dans ce cas-là.

De plus, dans certains cas, il peut être intéressant de normaliser les valeurs à une moyenne proche de 0 :

X_train /= np.max(np.abs(X_train),axis=0)

Comme précédemment, la première couche prend en entrée une dimension de taille égale à la taille d'une unité d'entrée (57 dans le cas des spams). On cherche à partir de là, quel est le nombre optimal de neurones cachés à utiliser (180 dans l'exemple du dessus). En effet, trop peu de neurones vont entraîner un sous-apprentissage et donc une accuracy faible. A l'inverse, un nombre trop élevé de neurones peut amener à un surapprentissage ("Overfitting") et des temps de

calculs trop longs. Il faut donc trouver le bon ratio entre ces trois variables.

Nombre de neurones cachés	Accuracy Train	Accuracy Test	Temps execution / par époque en ms
5	0,887	0,887	15
20	0,9115	0,9057	17
50	0,9263	0,9277	18
100	0,939	0,9371	19
140	0,9462	0,9391	19
180	0,9468	0,944	19
260	0,95	0,9387	22
340	0,9539	0,9387	23
420	0,9532	0,9418	23
500	0,9546	0,9418	25
580	0,9526	0,9387	27
660	0,9536	0,9418	28
740	0,9552	0,9418	29
820	0,9546	0,9418	30
900	0,9549	0,9438	32
980	0,9556	0,945	33
1060	0,9526	0,9418	35
1140	0,9552	0,9465	36
1220	0,956	0,9465	36
1300	0,9556	0,9481	38
1380	0,954	0,9418	40
1460	0,957	0,946	43
1540	0,956	0,944	45
1620	0,9552	0,9404	50

FIGURE 13 – Évolution de l'accuracy et du temps d'exécution en fonction du nombre de neurones cachés

Nous avons donc relevé les différentes valeurs retournées par le modèle en changeant à chaque fois le nombre de neurones cachés à utiliser. Au vu des résultats, il semblerait qu'utiliser 180 neurones soit un bon compromis dans ce cas-là, d'autant plus qu'à partir de là, augmenter le nombre de neurones ne fait plus augmenter l'accuracy significativement. Le temps d'exécution continue cependant d'augmenter.

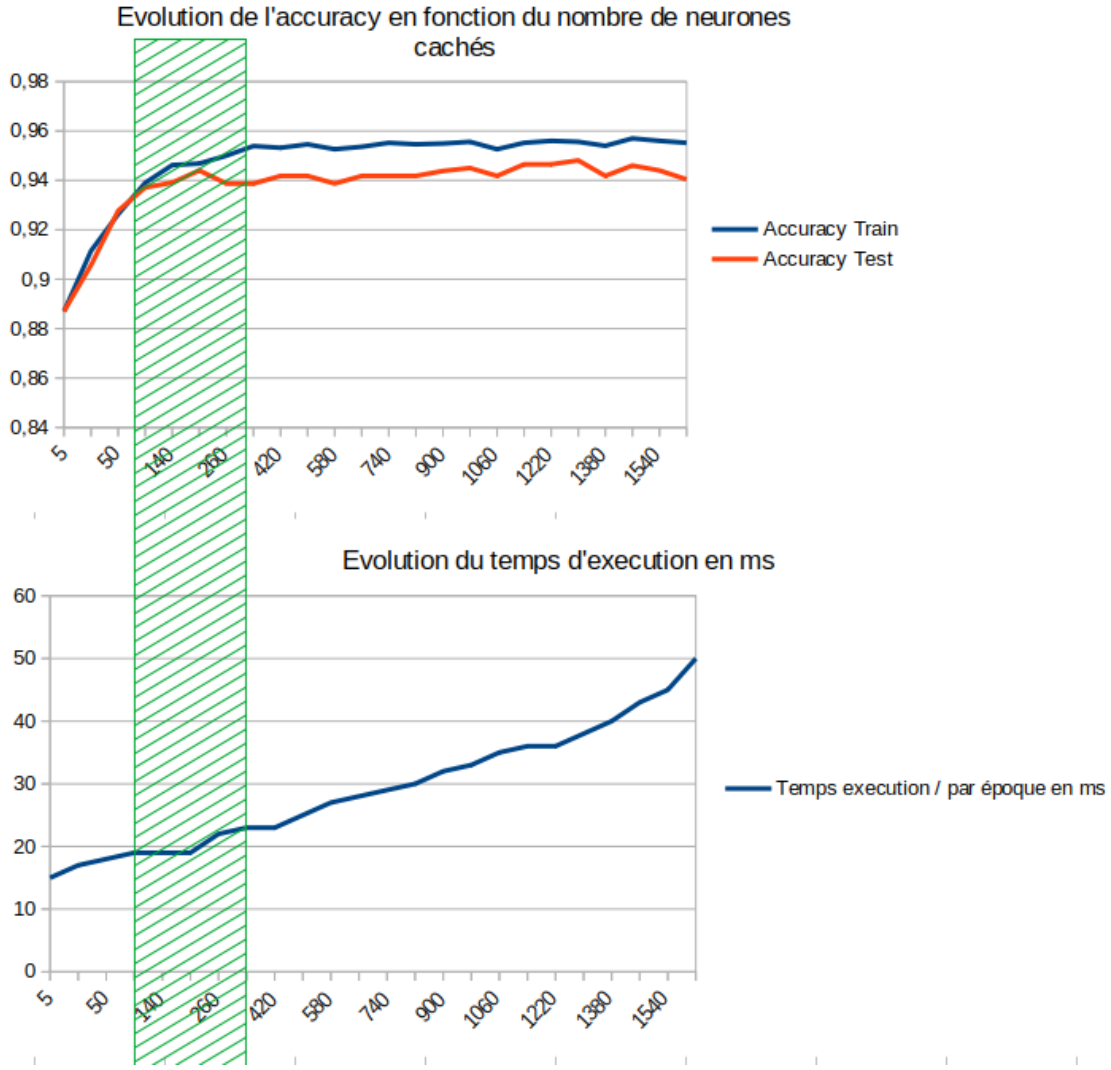


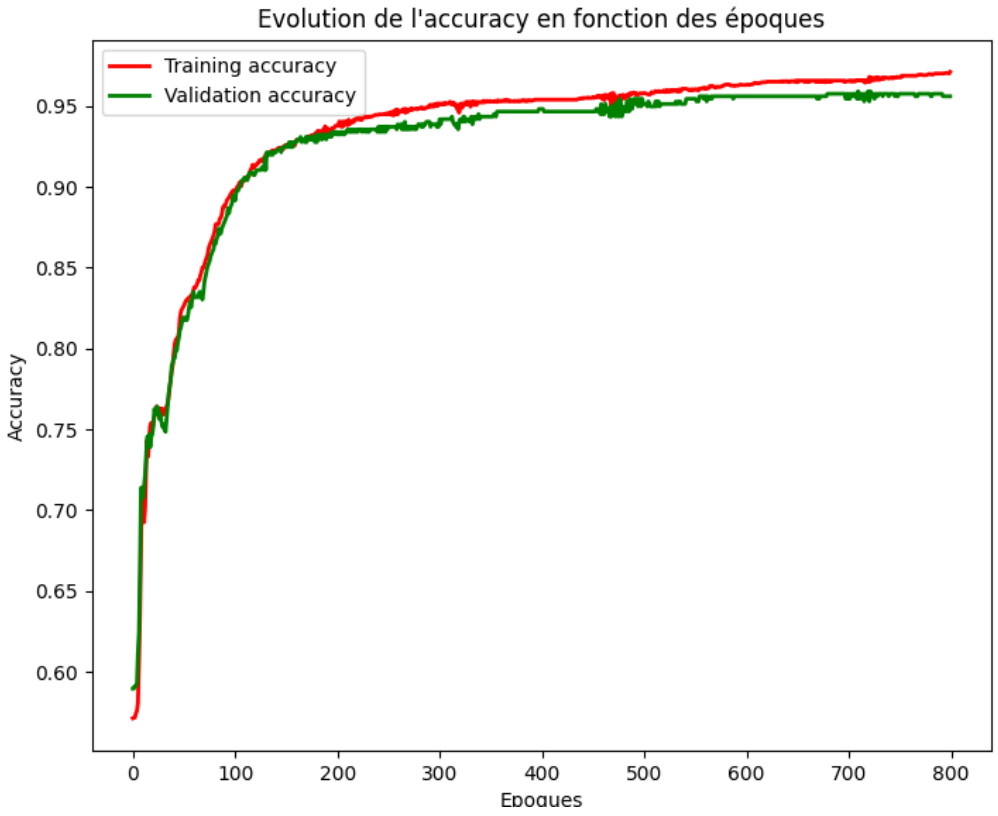
FIGURE 14 – Évolution de l'accuracy et du temps d'exécution en fonction du nombre de neurones cachés

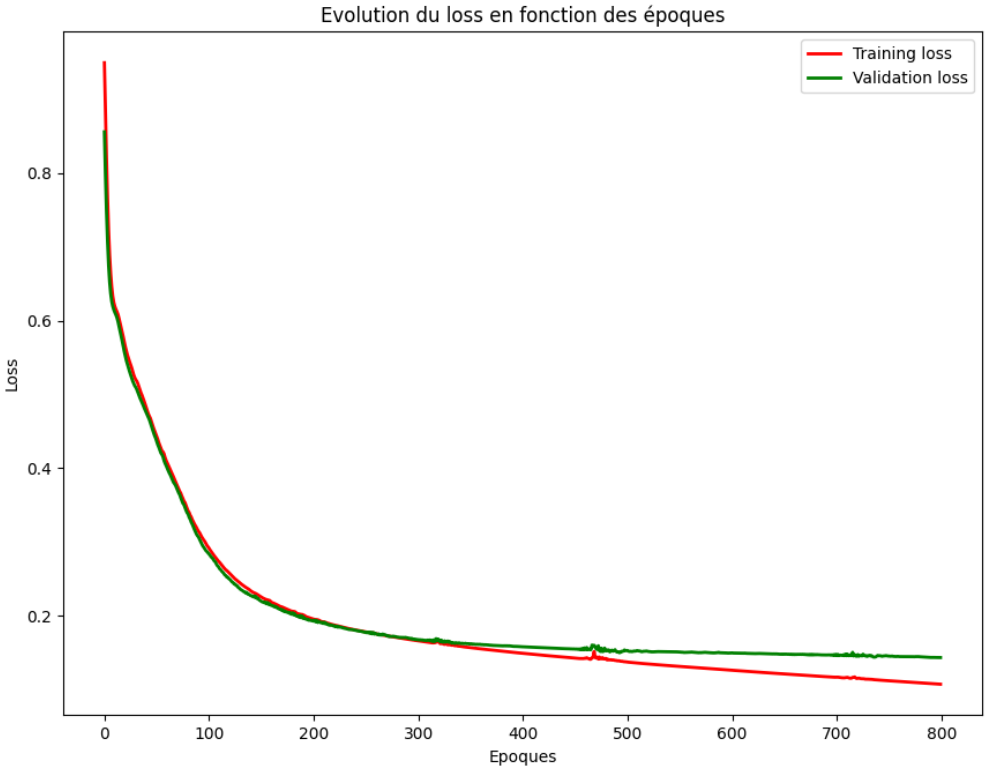
Voici, le résultat retourné par le modèle créé :

```
93/93 [==============================] - 0s 1ms/step - loss: 0.0752 - accuracy: 0.9769 - val_loss: 0.1295 - val_accuracy: 0.9591
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 180)               10440
_________________________________________________________________
dense_1 (Dense)              (None, 2)                 362
=================================================================
Total params: 10,802
Trainable params: 10,802
Non-trainable params: 0
_________________________________________________________________
Evaluation terminée :

Test loss: 0.0713382288813591
Test accuracy: 0.9747644662857056
```

FIGURE 15 – Résultat de l'entraînement du modèle à une couche cachée

De même que pour le réseau sans couche cachée, on choisit le nombre d'époque grâce aux graphiques récapitulatifs générés par Matplot et aux données récupérées de l'historique du "train". On constate un léger surapprentissage à partir d'une certaine époque. Sur la base à prédire, ce modèle obtient une accuracy de 0.954 environ pour 250 époques.




Evolution du loss en fonction des époques
Training loss
Validation loss
Loss
0.8
0.6
0.4
0.2
0
100
200
300
400
500
600
700
800
Epoques

3.3 Classifieur Bayésien naïf

Pour cette partie, nous avons essayé, au maximum, de ne pas utiliser de librairies externes, nous permettant d'avoir la meilleure vue possible du fonctionnement de l'algorithme et de pouvoir contrôler chaque instruction du début à la fin. Cela évite donc l'effet de boîtes noires. Nous avons donc utilisé seulement le package "csv" pour lire un fichier CSV et le package "math" pour l'utilisation de PI. Ces modules sont présents de base dans python, notre programme ne nécessite donc aucune installation annexe. Tout d'abord, le programme construit une liste de dictionnaires contenant les données d'entraînements. Cela permettant de facilement parcourir les données en fonction des champs.

```
#Creation d'une liste de dictionnaire à partir des donnees de train provenant du csv
with open('dataTrain.csv',newline='') as csvfile:
    trainingData = [{k: v for k, v in row.items()}
        for row in csv.DictReader(csvfile,delimiter=';', quotechar='|', skipinitialspace=True)]
```

Ensuite, le programme calcule, pour chaque champ du fichier d'entraînement, l'espérance et la variance en séparant d'un côté les tuples spam et les non-spams. Cela va nous permettre, dans un deuxième temps, de calculer les probabilités postérieures grâce à la règle de décision et à la formule de la densité de probabilité de la loi normale.

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} \, \mathrm{e}^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$$

On va donc calculer, pour chaque classe (spam et non spam), la probabilité postérieure que le tuple soit un spam et la probabilité postérieure que le tuple ne soit pas un spam. Pour cela, on applique la formule de densité de la loi normale pour chaque attribut et grâce aux variances et aux espérances de chaque attribut de chaque classe calculées précédemment grâce au fichier d'entraînement. Enfin, le programme considère que le tuple s'agit d'un spam si : $P_{posterieur}(SPAM) > P_{posterieur}(NON\_SPAM)$.

$$p(F_1, \ldots, F_n|C) = p(F_1|C)\, p(F_2|C)\, p(F_3|C)\, \cdots\, p(F_n|C) = \prod_{i=1}^{n} p(F_i|C).$$

FIGURE 16 – Formule permettant de calculer la probabilité postérieure

Cependant, il peut arriver, dans certains cas, que la variance d'un attribut pour une classe donnée soit égale à 0. Dans ce cas, on constate que le calcul de la densité de probabilité va entraîner une division par 0. Il faut donc traiter ce cas-là. Si la variance est égale à 0 mais que le x (dans la formule) est différent de l'espérance alors on renvoie 0. En effet, si l'espérance est égale à n et la variance égale à 0 alors cela signifie que pour cet attribut et cette classe spécifique toutes les valeurs étaient à n durant l'entraînement. Donc, si x est différent de n, alors à priori il ne s'agit pas de la bonne classe. Dans le cas où la variance est égale à zéro et que l'espérance est égale à x alors il s'agit peut-être de la bonne classe. Cette probabilité peut être calculée mais nécessite des temps de calculs trop élevés par rapport au gain qu'apporte cette probabilité. Le programme se contente donc de retourner 1. Comme ce résultat nous sert pour un produit, retourner 1 permet d'ignorer le cas où la variance est égale à 0 et où l'espérance est égale à x.

Dans notre programme, il est possible d'indiquer le nombre de lignes que doit utiliser le programme pour s'entraîner (calculer les variances et les espérances). Par exemple : python3 bayes.py 1000, le programme lira seulement les 1000 premières lignes du fichier d'entraînement.

3.4 Analyse et comparaison des résultats

3.4.1 Réseaux de neurones

Nous nous sommes intéressé ici à l'évolution de l'Accuracy en fonction de la taille de la base d'entraînement ainsi que la taille de l'entrée pour chaque tuple (nombre de colonnes). Dans les trois programmes, il est très facile de modifier facilement le nombre de lignes et de colonnes à prendre en compte lors de l'apprentissage.

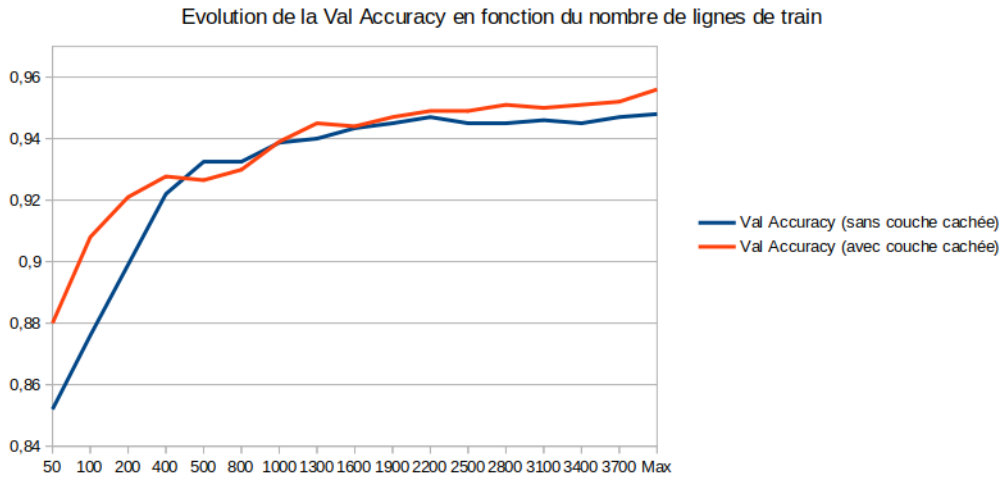


FIGURE 17 – Évolution de l'Accuracy en fonction de la taille de la base de test

On constate, logiquement, que l'accuracy augmente quand la taille de la base d'entraînement augmente. Cette courbe semble logarithmique ce qu'il signifie que pour avoir des résultats précis il faut une base minimum sur laquelle le modèle doit s'entraîner. Dans ce cas-là, il semblerait qu'il faille au moins 1000 tuples pour obtenir des résultats pertinents. Cependant, à partir d'un certain stade, l'accuracy n'augmente plus significativement. De plus, la taille de la base de test va impacter grandement les temps d'exécution. Que se soit avec ou sans couche cachées, l'évolution de l'accuracy en fonction de la taille de la base de test se comporte de la même manière.

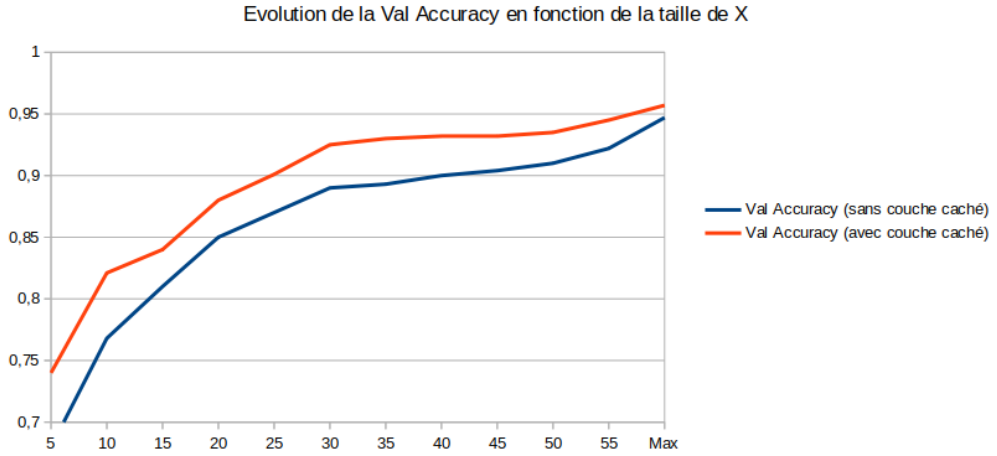


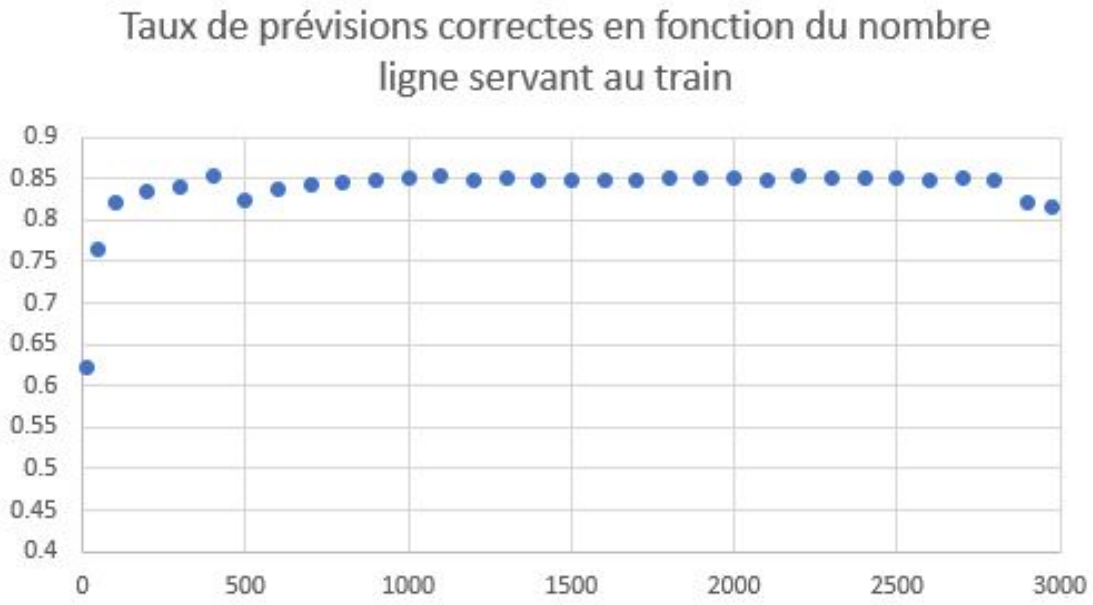
FIGURE 18 – Évolution de l'Accuracy en fonction de la taille d'une entrée

On remarque, que le nombre de colonnes, c'est-à-dire la taille d'un tuple en entrée, influe sur l'accuracy (ce qui est logique). D'après la courbe, il semble indispensable d'utiliser toutes les données disponibles pour obtenir une accuracy la plus proche de 1. Pour finir, on remarque que les accuracies semblent converger pour les deux réseaux de neurones.

3.4.2 Classification naïve Bayésienne

Le programme obtient un taux de prévisions correctes de 83.9% pour prédire les mails spams ou non spams.(En utilisant toutes les lignes du fichier d'entraînement)

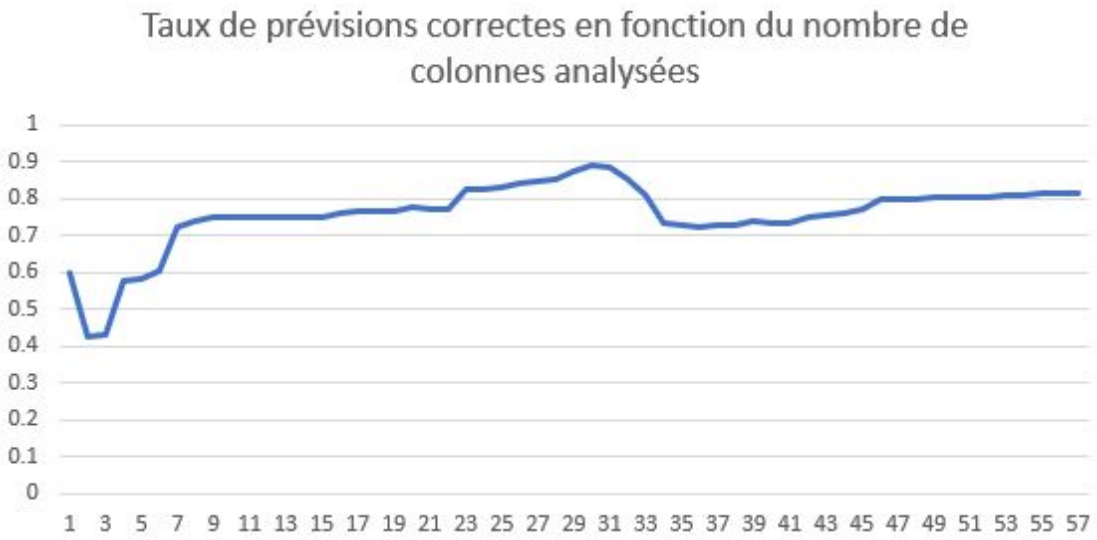
Nous avons cherché à observer l'évolution du taux de prévisions correctes en fonction du nombre de la taille du fichier de train, nous obtenons les résultats paraissant plutôt logiques.



On constate, dans ce cas, que pour obtenir un taux de prévisions correct, une base minimum de données est nécessaire pour ne pas avoir un mauvais taux de prévisions. En effet, un taux de prévisions proche de 0.5 est très mauvais, puisqu'on a deux classes (spam et non spam), il s'agit pratiquement d'une prédiction aléatoire. Cependant, on constate qu'à un certain point, le taux de prédiction n'augmente pas en conséquence de l'augmentation de la taille du

fichier d'entraînement. L'évolution du taux de prévisions correctes en fonction du nombre de lignes semble logarithmique.

Nous nous sommes ensuite intéressés au taux de prédictions en fonction du nombre d'attributs (colonne) prit en compte, nous permettant d'identifier certains attributs pas forcément pertinents.



La valeur sur l'axe des abscisses correspond au x ème premières colonnes analysées. On constate logiquement que plus on analyse d'attributs (colonne) plus l'algorithme obtient un taux de prédictions correctes élevé. Cependant on remarque aussi que la prise en compte des colonnes 31 à 35 fait baissé le taux de prédictions correct. On peut donc penser que ces colonnes sont peut-être moins pertinentes. En effet, si on ne prend pas en compte dans les calculs les colonnes 30,31,32,33 et 34 on obtiens un taux de prédiction supérieur. (On passe de 83.9% à 88.7%)

```
jean@jean-Aspire-E5-575G:~/Bureau/M1/S2/IA2/Projet/partie2/Spam$ python3 bayesClassifier.py
Taux de bonnes prévisions en ignorant les colonnes (30,31,32,33,34): 0.8877551020408163
```

3.4.3 Conclusion

Pour conclure, les réseaux de neurones obtiennent des résultats plus satisfaisants en moyenne. Pour les trois classifieurs obtenus, une base minimum

de test est nécessaire pour obtenir de bons résultats. Cependant, le nombre de colonnes utilisé semble moins impacter le classifieur Bayésien que les réseaux de neurones. De plus, celui-ci a le mérite de pouvoir être totalement contrôlable du début jusqu'à la fin et ne comporte aucune boîte noire contrairement aux réseaux de neurones avec Keras. En prenant en considération que les attributs sont indépendants entre eux, le classifieur Bayésien permet d'obtenir des résultats très satisfaisants pour une structure beaucoup plus légère que les réseaux de neurones.