Metaheuristics for Optimization

Romain Mencattini

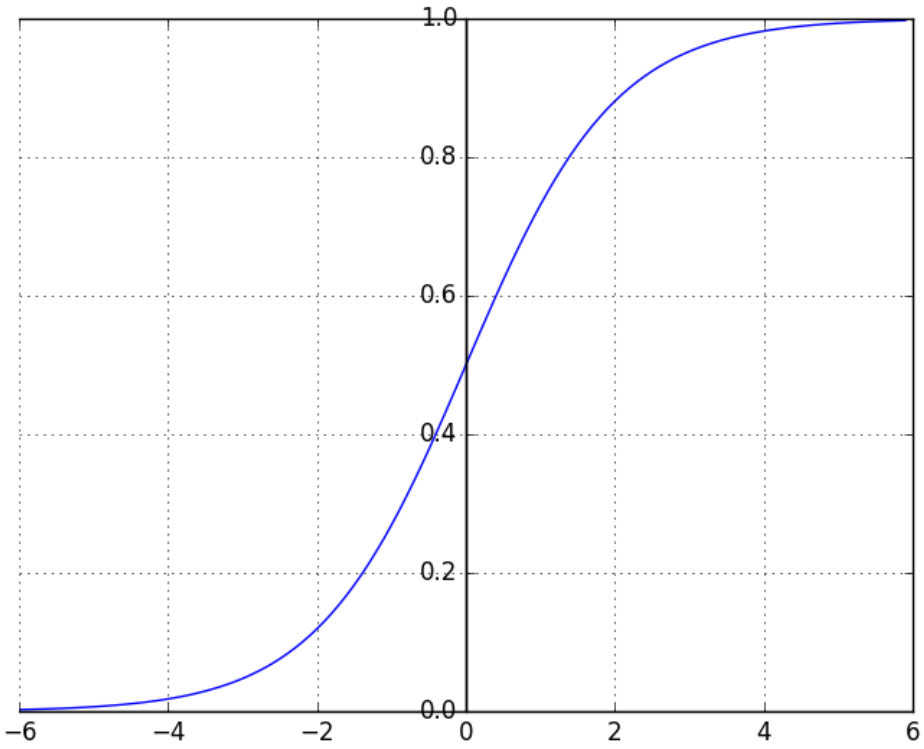
22 novembre 2016

Table des matières

1 Réseau de neurones

Les réseaux de neurones sont une méthodes d'apprentissage. Cela signifie qu'à partir d'un ensemble d'entrainement, ils vont *apprendre* puis pourrons sortir les bons *outputs* pour des *inputs* donnés.
Ils peuvent être composés de plusieurs couches , ou *layouts*. Chaque couche contient des poids. En terme mathématique, on a un input I de taille (m, n) qu'on passe dans une couche C_1 de taille (n, o). Ce résultat sera ensuite réinjecté dans l'éventuelle couche suivante C_2 de taille (o, p) etc. Le but est d'obtenir à la fin une unique valeur. Une fois cette valeur, on va la passer à une fonction non linéaire, comme la fonction *sigmoid* qui est définie comme :

$$g(z) = \frac{1}{1+\exp -z}$$



L'opération mathématique qui se cache derrière le passage d'*inputs* à une couche est la multiplication matricielle. C'est pour cela qu'il est important que les dimensions correspondent.

Dans notre cas, nous allons avoir des images de **400** pixels, sous formes d'un vecteur de taille $1x400$, avec deux couches. Petite subtilité, nous voulons introduire un biais. En effet, sans ce dernier la séparation des classes se fera uniquement en terme de rotation autour de l'origine, mais avec l'ajout du biais, on se retrouver avec une séparation affine et donc plus précise. Il faudra donc que la première couche soit de taille 401×25, car on a 400 pixels $+$ 1 biais.
La deuxième couche elle sera de taille 26×1, car on aura les 25 résultats de la couche précédente ainsi que le biais.
On obtient donc bien à la fin un résultat de taille 1×1.

2 Algorithme PSO

Le but de l'algorithme PSO sera d'optimiser les poids des matrices correspondantes aux deux couches de notre réseau. Nous allons présenter le pseudo-code puis l'implémentation :

1. On initialise tout les variables ainsi que les particules
2. Pour chaque particules on calcule la fitness et on met à jour le meilleur pour chaque particules.
3. On trouve le meilleur global en regardant chaque particule
4. Pour chaque particules on génère des nombres puis, on les fait bouger
5. On recommence à 2. tant que la condition d'arrêt n'est pas atteinte.

1. Initialisation Nous avons décidé de partir sur une logique objet. Chaque particule est un objet avec les caractéristiques suivantes :

— un index i pour identifier la particule
— les matrices Θ_1 et Θ_2 qui sont les matrices des poids à optimiser
— une vitesse v_i qui représente à une certaine itération sa vitesse, il s'agit d'un vecteur de la taille de tout les poids
— un meilleur local, qui correspond aux meilleures matrices rencontrées
— les valeurs de fitness pour les matrices ainsi que le meilleur local, (pour éviter les calculs superflus)

Pour initialiser les matrices, on génère des nombres $\in [-1, 1]$, pour la vitesse on génère des nombres $\in [0, 1]$ pour chaque composantes. Le meilleur local se trouve être les premières matrices générées vu qu'ils n'y en a aucune autre.

2. Calcul de fitness Concernant le calcul de fitness, c'est là que le réseau de neurones rentre en compte. En effet pour calculer la fitness, il faut prendre chacune des 200 images, l'évaluer grâce au réseau de neurones puis la fonction *sigmoid* et regarder si elle est bien classifiée ou non par rapport à son label. Ensuite, on regarde le nombre d'erreur, et on va tenter de minimiser ce nombre.
Plus précisément, si on pose que $h(x_k)$ correspond au résultat du réseau et de la fonction, on cherche à minimiser :

$$\sum_k (y_k - h(x_k))^2$$

Où k représente l'index de notre image.

Au niveau de l'implémentation et pour rester dans une logique objet, nous avons créer un objet *Espace* qui contient toutes les particules ainsi que le meilleur résultat global. Nous itérons donc sur toutes les particules de l'espace afin de calculer les fitness. Ces dernières disposent d'une méthode qui calcule la fitness et la stock dans leur attribut. Puis une fois cela fait, l'espace itère de nouveau sur toutes les particules afin de mettre à jour le meilleur local de chaque particule, vu qu'on a stocké la valeur de fitness du résultat courant et du best, l'opération se fait en $O(n)$ ce qui est agréable.

3. Minimum global En itérant sur chaque particule et en comparant avec le minimum global, on met à jour ce dernier. De la même manière que pour les particules, le fait d'avoir stocké ces informations, accélère le processus.

4. Mise à jour des particules Il est nécessaire de faire bouger les particules. Pour cela, nous allons d'abord générer des nombres aléatoires $\in [0, 1[$ nommé r_1, r_2. Puis on allons modifier la vitesse de chaque particule de la manière suivante :

$$v_i = \omega v_i + c_1 r_1 (b_i - s_i) + c_2 r_2 (b_{global} - s_i)$$

Où c_1,c_2,ω sont des constantes, s_i représente les matrices Θ_1 et Θ_2, b_i est le meilleur local et b_{global} est le meilleur de l'espace entier.

Une fois cela fait, on peut appliquer ,ou non cela dépend des paramètres souhaités, le cut_{off}. Cette opération consiste à mettre à 0 les composantes de la vitesse qui sont trop élevées. Afin d'éviter de traverser l'espace de manière trop brutale et donc d'osciller trop violemment.

Puis on fait :

$$s_i = s_i + v_i$$

Afin de mettre à jour la position. On remarque donc que la vitesse sera dépendante d'autres éléments, et donc ne sera pas influencée que par sa seule inertie. C'est de cette manière que les particules s'influencent les unes les autres.

5. Condition d'arrêt Dans notre cas, nous avons décidé d'avoir une condition d'arrêt, correspondant à un nombre d'itération maximal. Une fois ce nombre atteint , on stoppe l'algorithme et on récupère le meilleur minimum de l'espace.

3 Paramètres et détails d'implémentations

Nous l'avons vu dans la section précédente, il y a énormément de paramètres modifiables. Nous allons détailler l'influence de ces paramètres. Cette influence sera visible dans les résultats de la section suivante.

Tout d'abord la constante ω permet de modifier l'inertie de la particule. Plus il sera grand plus la particule continuera sur sa trajectoire, en revanche si on le diminue, la vitesse sera bien plus influencée par les autres particules et donc elle convergera plus rapidement. On a ici la différence entre l'intensification et l'exploration. Si on a un ω grand, on sera dans une logique d'exploration, et dans le cas contraire, dans une logique d'intensification.

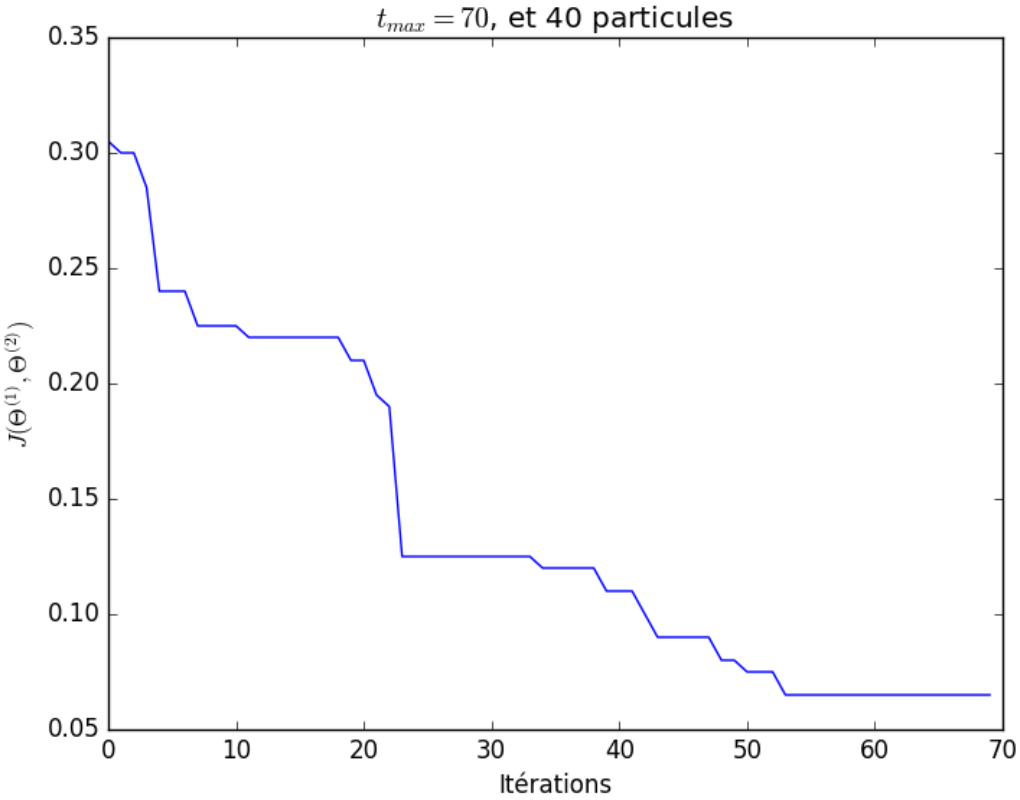
Concernant les c_1,c_2 cela permet d'influencer d'une manière plus ou moins forte la trajectoire. On peut donner plus d'importance au minimum local ou au minimum global. La meilleur solution reste de mettre la même valeur (2) afin de ne pas risquer de trop favoriser un type de solution par rapport à un autre. On risquerait de louper l'optimum global.

Le dernier point le cut_{off}. Il permet de limiter la vitesse des particules et donc d'éviter de sortir de l'espace de recherche ou bien de faire des pas trop important. En effet, la vitesse n'était pas capée par l'équation de base, il est possible d'obtenir des pas bien trop grand et donc d'osciller d'une manière trop importante. On obtient donc une variance très grande, ainsi qu'une exploration peut efficace et un risque de *buffer overflow*.
Une valeur acceptable pour le cut_{off} sera de prendre le quart de la distance max de l'espace de recherche. Comme cela on parcourait efficacement l'espace. C'est d'ailleurs notre choix, mais nous le verrons avec les résultats.

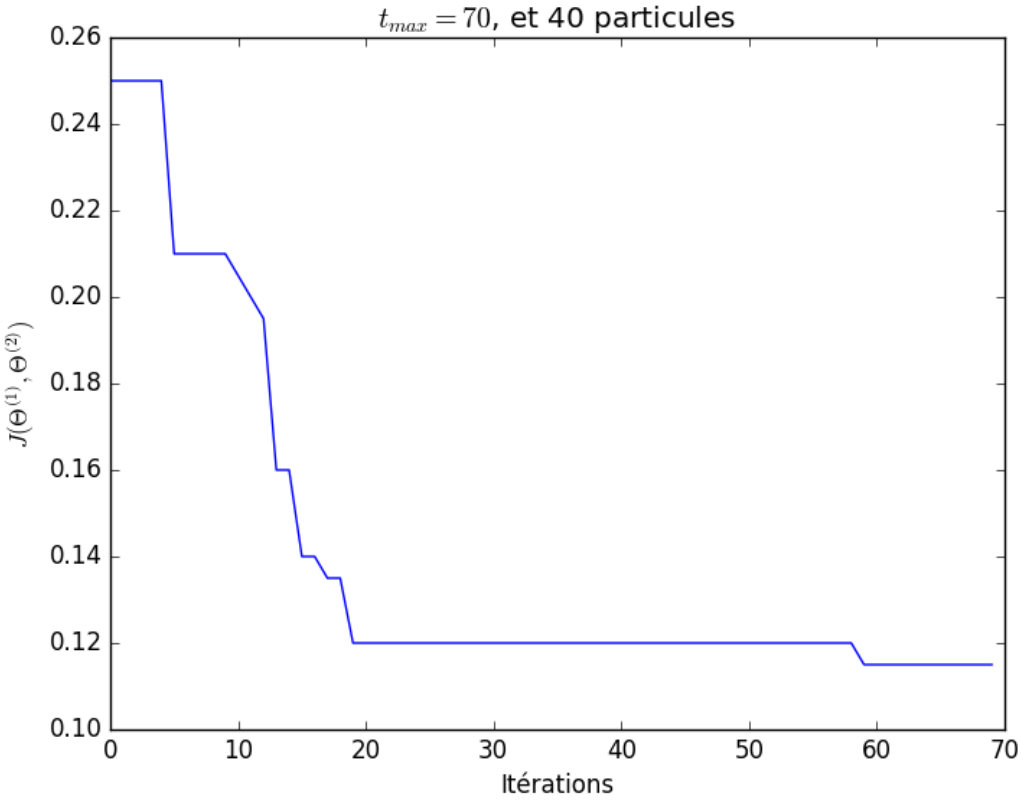
Au niveau de l'implémentation : nous avons abusé de la librairie *numpy* de calcul scientifique afin d'optimiser les calculs. Pour ce faire, nous avons vectorisé presque toutes les opérations afin de ne pas perdre du temps lors de boucle. En effet les boucles en Python sont particulièrement lentes, alors que l'utilisation de la vectorisation via *numpy* (écrite en C) est particulièrement efficace. Le code est plus efficace mais moins lisible et donc moins maintenable, c'est donc un choix à faire.

4 Résultats et Conclusion

Voici un exemples de *run* avec 40 particules, 70 itérations et un $\omega = 0.75$, avec cut_{off}

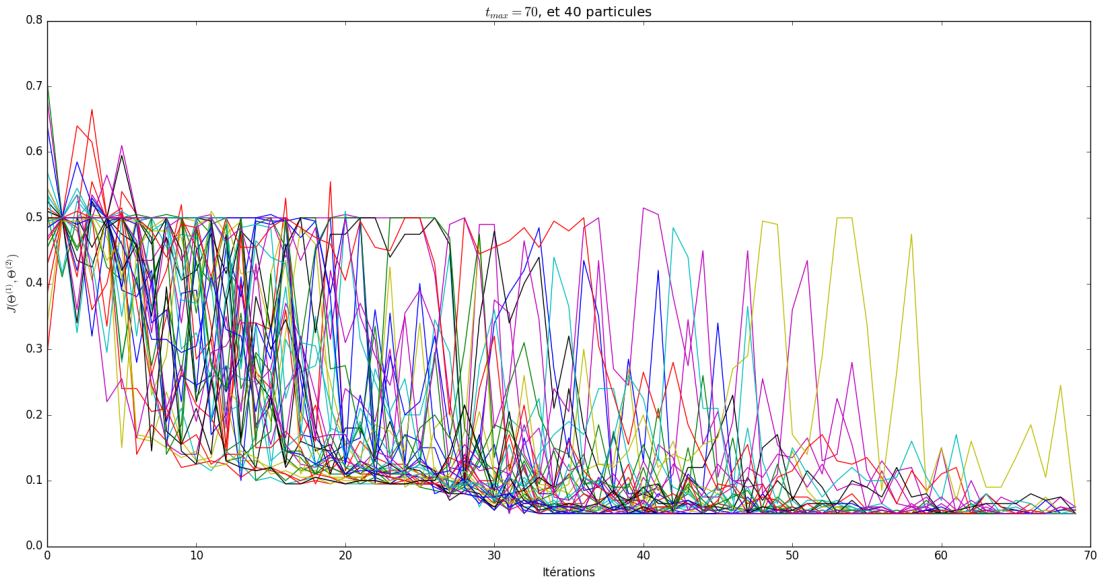


Un autre exemple avec 40 particules, 70 itérations et un $\omega = 0.75$, sans cut_{off}



On remarque que la manière dont décroit la fitness est bien plus violente dans la version sans cut_{off}. Cela vient du fait que la vitesse n'étant pas capée, on se retrouve avec d'énormes oscillations et donc l'exploration se fait pas à coup. Ce qui n'est pas le cas avec cut_{off}, on a donc une exploration , et une décroissance plus régulière.

On peut observer ce phénomène d'oscillations en dessinant les fitness de chaque particule au courant du temps :



Nous remarquons que malgré que $\omega = 0.5$ ce qui devrait encourager une convergence rapide, les particules mettent du temps à converger et sont chaotiques. Un autre élément important pour différencier ces deux *runs* est de s'intéresser aux min,max, à la déviation standard et la moyenne. On obtient donc :

Avec cut_{off}

	Θ_1	Θ_2
Min	-2.19	-1.61
Max	2.29	1.26
Mean	0.03	-0.06
Std	0.59	0.73

Sans cut_{off}

	Θ_1	Θ_2
Min	-14.44	-9.64
Max	13.20	9.72
Mean	-0.39	0.35
Std	5.03	4.94

On remarque que toutes les valeurs sont bien plus extrêmes dans le cas où on n'utilise pas le cut_{off}. On peut se concentrer donc dans une zone plus restreinte et on gagne de l'effort computationnel. On peut signaler qu'on a obtenu 93.5% dans le premier *run* et 88.5% pour la deuxième. Ce sont donc de bon résultats. Voyons lors qu'on lance 10 *runs*

Voici les résultats pour 10 *runs*, 30 particules, 40 itérations avec cut_{off}

$$[0.11, 0.085, 0.13, 0.12, 0.095, 0.100, 0.070, 0.125, 0.070, 0.170]\,;$$

ce qui nous donne :
Mean = 0.1075
Std = 0.0290

On obtient donc en moyenne : 89.25% de bons résultats, avec une déviation standard très faible.

Voici les résultats pour 10 *runs*, 30 particules, 40 itérations avec cut_{off}

$$[0.195, 0.165, 0.175, 0.165, 0.165, 0.315, 0.185, 0.155, 0.080, 0.11]\,;$$

ce qui nous donne :
Mean = 0.1710
Std = 0.0582

On obtient donc en moyenne : 86.9% de bons résultats, avec une déviation standard très faible.

On a donc de meilleurs résultats en moyenne avec le cut_{off}.