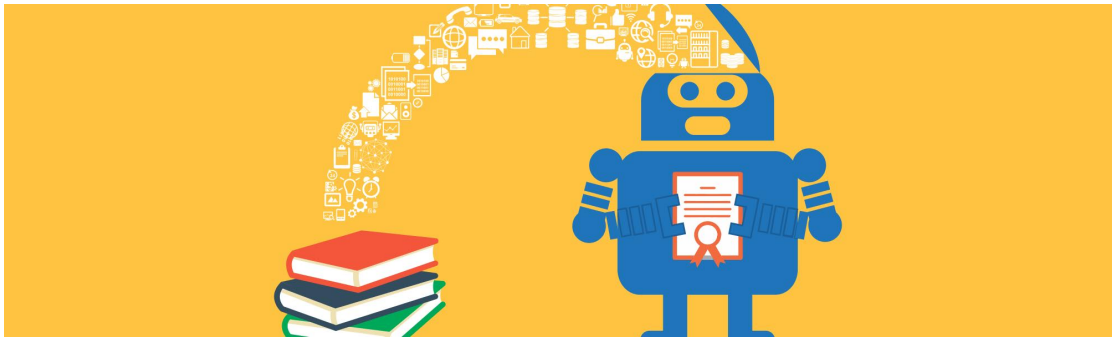


Recherche intelligente et collaborative

Crée par Mohamed Zennir (cloudschool.org/zennir)

Description

La construction de systèmes capables de résoudre de problèmes à travers des entités autonomes réalisant des tâches intelligentes et d'évoluer dans leur environnement. Ces entités seraient aptes de : planifier leurs actions, coopérer et mutualiser ainsi certaines de leurs ressources, d'entrer en compétition pour un meilleur résultat et où les ressources globales sont limitées.

Curriculum Area

Not Specified

Suggested Duration

Not Specified

Subject Area

Not Specified

Resources Required

Not Specified

List Summary

Introduction aux algorithmes de recherche

Introduction

Si les méthodes de résolution exactes permettent d'obtenir une solutions dont l'optimalité est garantie, dans certaines situations, on peut cependant chercher des solutions de bonne qualité, sans garantie d'optimalité, mais au profit d'un temps de calcul plus réduit. Pour cela, on applique des méthodes appelées métaheuristiques, adaptées à chaque problème traité, avec cependant l'inconvénient de ne disposer en retour d'aucune information sur la qualité des solutions obtenues.

Les heuristiques ou les méta-heuristiques exploitent généralement des processus aléatoires dans l'exploration de l'espace de recherche pour faire face à l'explosion combinatoire engendré par l'utilisation des méthodes exactes. En plus de cette base stochastique, les méta-heuristiques sont le plus souvent itératives, ainsi le même processus de recherche est répété lors de la résolution. Leur principal intérêt provient justement de leur capacité à éviter les minima locaux en admettant une dégradation de la fonction objectif au cours de leur progression.

L'optimisation combinatoire (OC) occupe une place très importante en recherche opérationnelle et en informatique. De nombreuses applications pouvant être modélisées sous la forme d'un problème d'optimisation combinatoire (POC) telles que le problème du voyageur de commerce, l'ordonnancement de tâches, le problème de la coloration de graphes, etc. (POC) comprend un ensemble fini de solutions, où chaque solution doit satisfaire un ensemble de contraintes relatives à la nature du problème, et une fonction objectif pour évaluer chaque solution trouvée. La solution optimale est celle dont la valeur de l'objectif est la plus petite (resp. grande) dans le cas de minimisation (resp. maximisation) parmi l'ensemble de solutions.

La résolution des problèmes combinatoires est assez délicate puisque le nombre fini de solutions réalisables croît généralement avec la taille du problème, ainsi que sa complexité. Cela a poussé les chercheurs à développer de nombreuses méthodes de résolution en recherche opérationnelle (RO) et en intelligence artificielle (IA). Ces approches de résolution peuvent être classées en deux catégories : les méthodes exactes et les méthodes approchées.
Les méthodes exactes ont permis de trouver des solutions optimales pour des problèmes de taille raisonnable et rencontrent généralement des difficultés face aux applications de taille importante. En revanche les méthodes approchées ne garantissent pas de trouver une solution exacte, mais seulement

une approximation.

Algorithmes d'approximation

En informatique théorique, un algorithme d'approximation est une méthode permettant de calculer une solution approchée à un problème algorithmique d'optimisation. Plus précisément, c'est une heuristique garantissant à la qualité de la solution qui fournie un rapport inférieur (si l'on minimise) à une constante, par-rapport à la qualité optimale d'une solution, pour toutes les instances possibles du problème.

L'intérêt de tels algorithmes est qu'il est parfois plus facile de trouver une solution approchée qu'une solution exacte, le problème pouvant par exemple être NP-complet mais admettre un algorithme d'approximation polynomial. Ainsi, dans les situations où l'on cherche une bonne solution, mais pas forcément la meilleure, un algorithme d'approximation peut être un bon outil.

Approximation

Selon la nature du problème (maximisation, minimisation etc.) la définition peut varier, on donne ici la définition classique pour un problème de minimisation avec facteur d'approximation constant.

Pour un problème de minimisation ayant une solution optimale de valeur , un algorithme d'approximation de facteur 1 (i.e. un **algorithme -approché**) est un algorithme donnant une solution de valeur , avec la garantie que .

Techniques algorithmiques

Parmi les techniques utilisées, on compte les méthodes d'algorithmique classique, par exemple un algorithme glouton permet parfois d'obtenir une bonne approximation à défaut de calculer une solution optimale. On peut aussi citer des algorithmes de recherche locale et de programmation dynamique.

Beaucoup d'algorithmes sont basées sur l'optimisation linéaire. On peut par exemple arrondir une solution fractionnaire ou utiliser un schéma primal-dual. Une technique plus avancée est d'utiliser l'optimisation SDP, comme pour le problème de la coupe maximum.

Historique

Des algorithmes d'approximation ont été découverts avant même la mise en place de la théorie de la NP-complétude, par exemple par Paul Erdős dans les années 1960, pour le problème de la coupe maximum. Cependant c'est à la suite de cette théorie que le domaine s'est vraiment développé. L'utilisation de l'optimisation linéaire est due à László Lovász dans les années 19704, pour le problème de couverture par ensembles.

Dans les années 1990, le théorème PCP a été une avancée très importante pour la non-approximabilité.

Heuristiques

En optimisation combinatoire, une heuristique est un algorithme approché qui permet d'identifier en temps polynomial au moins une solution réalisable rapide, pas obligatoirement optimale. L'usage d'une heuristique est efficace pour calculer une solution approchée d'un problème et ainsi accélérer le processus de résolution exacte. Généralement une heuristique est conçue pour un problème particulier, en s'appuyant sur sa structure propre sans offrir aucune garantit quant à la qualité de la solution calculée. Les heuristiques peuvent être classées en deux catégories :

– Méthodes constructives qui génèrent des solutions à partir d'une solution initiale en essayant d'en ajouter petit à petit des éléments jusqu'à ce qu'une solution complète soit obtenue,

– Méthodes de fouilles locales qui démarrent avec une solution initialement complète (probablement moins intéressante), et de manière répétitive essaie d'améliorer cette solution en explorant son voisinage.

Métaheuristiques

Face aux difficultés rencontrées par les heuristiques pour avoir une solution réalisable de bonne qualité pour des problèmes d'optimisation difficiles, les métaheuristiques ont fait leur apparition. Ces algorithmes sont plus complets et complexes qu'une simple heuristique, et permettent généralement d'obtenir une solution de très bonne qualité pour des problèmes issus des domaines de la recherche opérationnelle ou de l'ingénierie dont on ne connait pas de méthodes efficaces pour les traiter ou bien quand la résolution du problème nécessite un temps élevé ou une grande mémoire de stockage.

Le rapport entre le temps d'exécution et la qualité de la solution trouvée d'une métaheuristique reste alors dans la majorité des cas très intéressant par rapport aux différents types d'approches de résolution.

La plupart des métaheuristiques utilisent des processus aléatoires et itératifs comme moyens de rassembler de l'information, d'explorer l'espace de recherche et de faire face à des problèmes comme l'explosion combinatoire. Une métaheuristique peut être adaptée pour différents types de problèmes, tandis qu'une heuristique est utilisée à un problème donné. Plusieurs d'entre elles sont souvent inspirées par des systèmes naturels dans de nombreux domaines tels que : la biologie (algorithmes évolutionnaires et génétiques) la physique (recuit simulé), et aussi l'éthologie (algorithmes de colonies de fourmis).

Un des enjeux de la conception des métaheuristiques est donc de faciliter le choix d'une méthode et le réglage des paramètres pour les adapter à un problème donné.

Les métaheuristiques peuvent être classées de nombreuses façons. On peut distinguer celles qui travaillent avec une population de solutions de celles qui ne manipulent qu'une seule solution à la fois. Les méthodes qui tentent itérativement d'améliorer une solution sont appelées méthodes de recherche locale ou méthodes de trajectoire. Ces méthodes construisent une trajectoire dans l'espace des solutions en tentant de se diriger vers des solutions optimales. Les exemples les plus connus de ces méthodes sont : La recherche Tabou et le Recuit Simulé. Les algorithmes génétiques, l'optimisation par essaim de particules et Les algorithmes de colonies de fourmis présentent les exemples les plus connus des méthodes qui travaillent avec une population.

Discussion | Discussions

La méthode de la descente

Introduction

Les méthodes de recherche locale passent d'une solution à une autre dans l'espace des solutions candidates (l'espace de recherche) qu'on note S, jusqu'à ce qu'une solution considérée comme optimale soit trouvée ou que le temps imparti soit dépassé. La méthode de recherche locale la plus élémentaire est la méthode de descente.

Pour un problème de minimisation d'une fonction f , la méthode descente peut être décrite comme suit :

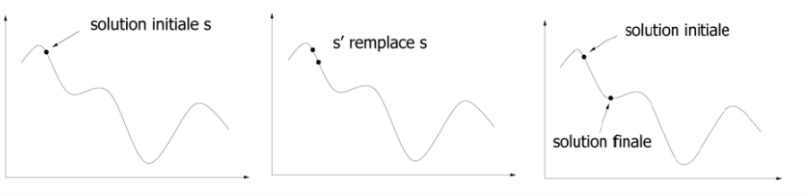
Algorithm 2 La méthode de descente

1. Solution initiale s ;
2. **Repeter** :
3. Choisir s' dans un voisinage $N(s)$ de s ;
4. Si $f(s') < f(s)$ alors $s := s'$;
5. **jusqu'à** ce que $f(s') \geq f(s)$, $\forall s' \in N(s)$.

qq

L'inconvénient majeur de la méthode de descente est son arrêt au premier minimum local rencontré. Pour améliorer les résultats, on peut lancer plusieurs fois l'algorithme en partant d'un jeu de solutions initiales différentes, mais la performance de cette technique décroît rapidement. Pour éviter d'être bloqué au premier minimum local rencontré, on peut décider d'accepter, sous certaines conditions, de se déplacer d'une solution s vers une solution s' ∈ N(s) telle que f(s') > f(s).

La figure suivante présente l'évolution d'une solution dans la méthode de descente.



Travaux pratiques

Trouvez les minimums des fonctions suivantes :

$$cos(x),\ \frac{1}{2}x^2 + \frac{3}{5}y^2 + 2,\ z.cos(x).cos(y)$$

Files | Réponse du TP

descente.py

La recherche tabou

Introduction

La recherche tabou (TS) est une méthode de recherche locale combinée avec un ensemble de techniques permettant d'éviter d'être piégé dans un minimum local ou la répétition d'un cycle. La recherche tabou est introduite principalement par Glover (Glover 1986), Hansen (Hansen 1986), Glover et Laguna dans (Glover et Laguna 1997). Cette méthode a montré une grande efficacité pour la résolution des problèmes d'optimisation difficiles. En effet, à partir d'une solution initiale s dans un ensemble de solutions local *S*, des sous-ensembles de solution *N(s)* appartenant au voisinage *S* sont générés. Par l'intermédiaire de la fonction d'évaluation nous retenons la solution qui améliore la valeur de f , choisie parmi l'ensemble de solutions voisines *N(s)* .

L'algorithme accepte parfois des solutions qui n'améliorent pas toujours la solution courante. Nous mettons en oeuvre une liste tabou (tabu list) T de longueur k contenant les k dernières solutions visitées, ce qui ne donne pas la possibilité a une solution déjà trouvée d'être acceptée et stockée dans la liste tabou. Alors le choix de la prochaine solution est effectué sur un ensemble des solutions voisines en dehors des éléments de cette liste tabou. Quand le nombre k est atteint, chaque nouvelle solution sélectionnée remplace la plus ancienne dans la liste. La construction de la liste tabou est basée sur le principe FIFO, c'est-à-dire le premier entré est le premier sorti. Comme critère d'arrêt on peut par exemple fixer un nombre maximum d'itérations sans amélioration de *s**, ou bien fixer un temps limite après lequel la recherche doit s'arrêter.

Algorithm 4 La recherche tabou

1: Initialisation :
   s_0 une solution initiale
   $s \longleftarrow s_0$, $s^* \longleftarrow s_0$, $c^* \longleftarrow f(s_0)$
   $T = \emptyset$
2: Générer un sous-ensemble de solution au voisinage de s
   $s' \in N(s)$ tel que $\forall x \in N(s),\ f(x) \geq f(s')$ et $s' \notin T$
   Si $f(s') < c^*$ alors $s^* \longleftarrow s'$ et $c^* \longleftarrow f(s')$
   Mise-à-jour de T
3: Si la condition d'arrêt n'est pas satisfaite retour à l'étape 2

Influence des paramètres

Contrairement à l'algorithme de la descente, le nouveau sommet s' qui est toujours choisi dans le voisinage du sommet courant l'est en comparant les sommets voisins entre eux et non plus avec le sommet courant. De ce fait, lorsque l'algorithme atteint

un minimum local, il peut en sortir car à la prochaine comparaison le sommet courant n'est pas pris en compte. Cependant, à l'itération suivante, il y a de très fortes probabilités que l'algorithme retombe dans le minima local dont il vient juste de sortir.

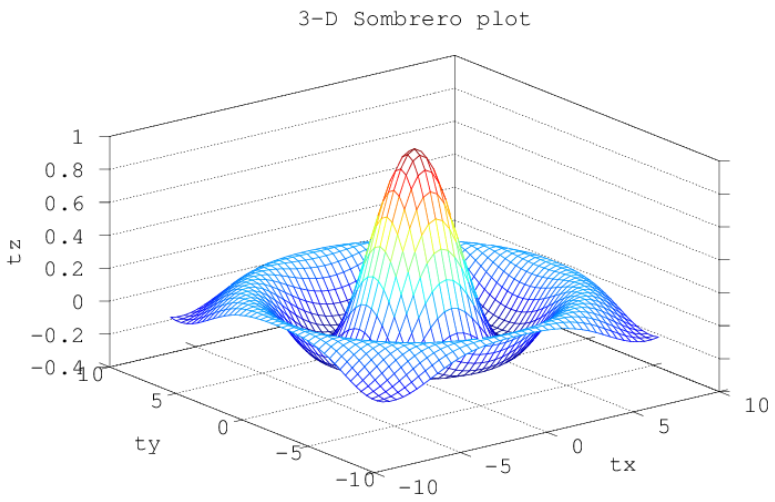
Pour éviter cela, l'algorithme utilise la notion de liste tabou qui répertorie les sommets visités (dans un passé immédiat plus ou moins long) qui sont interdits de revisite pour éviter de retomber très vite dans le minima local initial.

Le comportement de l'algorithme tabou est par conséquent très dépendant de sa mémoire, c-à-d, la taille de sa liste tabou :

- Si la liste taboue est courte, les interdictions (mouvements tabous) sont moindre et la recherche épouse mieux les optimas locaux rencontrés.
  L'algorithme tend à parcourir de moins grandes distances dans l'espace de recherche dont il n'en explore pas de grandes parties. Le risque de cycles (retomber sans cesses dans le même optimum local) est plus grand.
- Si la liste taboue est longue, les interdictions (mouvements tabous) sont plus nombreuses et la recherche risque de manquer de nombreux optimas locaux sur son chemin. L'algorithme tend à parcourir de plus grandes distances dans l'espace de recherche dont il en explore davantage de grands pans.
  Le risque de cycles est réduit et l'amélioration de la valeur de la fonction objective est plus fréquente.

Une recherche tabou dont la taille de la liste est 1 est équivalente à la méthode de la descente (avec une rechute infinie dans le voisinage de l'optimum local). Une recherche tabou dont la taille de la liste est infinie s'apparente à une recherche exhaustive bien qu'il n'est pas certain que l'algorithme puisse tester toutes les combinaisons de l'espace de recherche (possibilité d'emprisonnement d'un extremum global par une sur/dépression circulaire).

Exemple d'un cas où une liste tabou infinie peut échouer



Avantages et incovénients

La recherche tabou est une méthode de recherche locale, et la structure de son algorithme de base est proche de celle du recuit simulé, avec l'avantage d'avoir un paramétrage simplifié : le paramétrage consistera d'abord à trouver une valeur indicative t d'itérations pendant lesquelles les mouvements sont interdits. Il faudra également choisir une stratégie de mémorisation. En revanche, la méthode tabou exige une gestion de la mémoire de plus en plus lourde en mettant des stratégies de mémorisation complexe. L'efficacité de la méthode tabou offre son utilisation dans plusieurs problèmes d'optimisation combinatoire classiques tels que le problème de voyageur de commerce, le problème d'ordonnancement, le problème de tournées de véhicules, etc.

Travaux Pratiques

Problème du sac à dos

Soit un sac à dos qui ne peut supporter un poids supérieur à 20 kgs. Soient les produits suivants décris par leur prix et leur poids :

Nom	Prix	Poids
P1	100	100 g
P2	50	300 g
P3	90	120 g
P4	1000	10 g
P5	70	230 g
P6	2000	1 g

On peut acheter une quantité quelconque de n'importe quel produit.

Question

Proposez un algorithme de recherche taboue qui permet de remplir au maximum le sac à dos avec un minimum de dépense d'argent.

Files | TP

taboo.py

Le recuit simulé

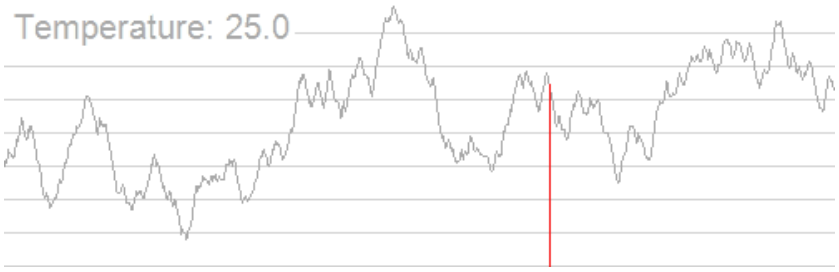
Introduction

Le **recuit simulé** est une méthode empirique (métaheuristique) inspirée d'un processus utilisé en métallurgie. On alterne dans cette dernière des cycles de refroidissement lent et de réchauffage (recuit) qui ont pour effet de minimiser l'énergie du matériau. Cette méthode est transposée en optimisation pour trouver les extrema d'une fonction.

Elle a été mise au point par trois chercheurs de la société IBM, S. Kirkpatrick, C.D. Gelatt et M.P. Vecchi en 1983, et indépendamment par V. Černy en 1985.

La méthode vient du constat que le refroidissement naturel de certains métaux ne permet pas aux atomes de se placer dans la configuration la plus solide. La configuration la plus stable est atteinte en maîtrisant le refroidissement et en le ralentissant par un apport de chaleur externe, ou bien par une isolation.

Recuit simulé cherchant un maximum. L'objectif ici est de trouver le plus haut point; seulement, ce n'est pas suffisant d'utiliser un simple algorithme de hill climbing car il y a de nombreux maximum locaux. En refroidissant la température lentement, le maximum global peut être trouvé.



Déroulement du processus

Le recuit simulé s'appuie sur l'algorithme de Metropolis-Hastings, qui permet de décrire l'évolution d'un système thermodynamique. Par analogie avec le processus physique, la fonction à minimiser deviendra l'énergie E du système. On introduit également un paramètre fictif, la température T du système.

Partant d'une solution donnée, en la modifiant, on en obtient une seconde. Soit celle-ci améliore le critère que l'on cherche à optimiser, on dit alors qu'on a fait baisser l'énergie du système, soit celle-ci le dégrade. Si on accepte une solution améliorant le critère, on tend ainsi à chercher l'optimum dans le voisinage de la solution de départ. L'acceptation d'une « mauvaise » solution permet alors d'explorer une plus grande

partie de l'espace de solution et tend à éviter de s'enfermer trop vite dans la recherche d'un optimum local.

Etat initial de l'algorithme

La solution initiale peut être prise au hasard dans l'espace des solutions possibles. À cette solution correspond une énergie initiale E=E_{0}. Cette énergie est calculée en fonction du critère que l'on cherche à optimiser. Une température initiale T=T_{0} élevée est également choisie. Ce choix est alors totalement arbitraire et va dépendre de la loi de décroissance utilisée.

Itérations de l'algorithme

À chaque itération de l'algorithme une modification élémentaire de la solution est effectuée. Cette modification entraîne une variation **ΔE** de l'énergie du système (toujours calculée à partir du critère que l'on cherche à optimiser). Si cette variation est négative (c'est-à-dire qu'elle fait baisser l'énergie du système), elle est appliquée à la solution courante. Sinon, elle est acceptée avec une probabilité e^(ΔE/T).Ce choix de l'exponentielle pour la probabilité s'appelle règle de Metropolis. On itère ensuite selon ce procédé en gardant la température constante.

Programme de recuit

Deux approches sont possibles quant à la variation de la température :

1. Pour la première, on itère en gardant la température constante. Lorsque le système a atteint un équilibre thermodynamique (au bout d'un certain nombre de changements), on diminue la température du système. On parle alors de paliers de température.
2. La seconde approche fait baisser la température de façon continue. On peut imaginer toute sorte de loi de décroissance, la plus courante étant T_{i+1} = λT_{i} avec λ <1 (assez couramment λ = 0.99).

La température joue un rôle important. À haute température, le système est libre de se déplacer dans l'espace des solutions (e^-(Δ_E/T) proche de 1) en choisissant des solutions ne minimisant pas forcément l'énergie du système. À basse température, les modifications baissant l'énergie du système sont choisies, mais d'autres peuvent être acceptées, empêchant ainsi l'algorithme de tomber dans un minimum local.

Algorithme

L'algorithme suivant met en œuvre le recuit simulé tel que décrit plus haut, en commençant à l'état s0 et continuant jusqu'à un maximum de kmax étapes ou jusqu'à ce qu'un état ayant pour énergie emax ou moins soit trouvé. E est une fonction calculant l'énergie de l'état s. L'appel voisin(s) génère un état voisin aléatoire d'un

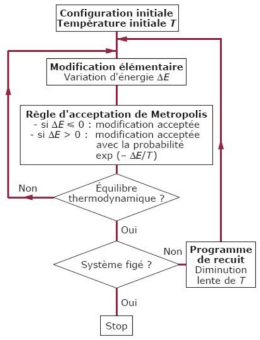
état s. T renvoie la température à utiliser selon la fraction r du temps total déjà dépensé, et P est une fonction de probabilité dépendant de la variation d'énergie et de la température.

Algorithm 3 Recuit simulé

1. Engendrer une configuration initiale S_0 de S : $S \longleftarrow S_0$
2. Initialiser la température T en fonction du schéma de refroidissement
3. Répéter
4. Engendrer un voisin aléatoire S' de S
5. Calculer $\Delta E = f(S') - f(S)$
6. Si $\Delta E \leq 0$ alors $S \longleftarrow S'$
7. Sinon accepter S' comme la nouvelle solution avec la probabilité $P(E,T) = \exp^{-\frac{\Delta E}{T}}$
8. Fin si
9. Mettre T à jour en fonction du schéma de refroidissement (réduire la température)
10. Jusqu'à la condition d'arrêt
11. Retourner la meilleure configuration trouvée

Organigramme de l'algorithme

Voici la version organigramme de l'algorithme.



Files | TP voyageur de commerce

travelling_sale.py

Files | TP

recuit.py

Recherche par essaims de particules

Introduction

L'optimisation est une branche des mathématiques qui permet de résoudre des problèmes en déterminant le meilleur élément d'un ensemble selon certains critères prédéfinis. De ce fait, l'optimisation est omniprésente dans tous les domaines et évolue sans cesse depuis Euclide.
En 1995, Russel Eberhart, ingénieur en électricité et James Kennedy, socio-psychologue, s'inspirent du monde du vivant pour mettre en place une méta-heuristique : l'optimisation par essaim particulaire. Cette méthode se base sur la collaboration des individus entre eux : chaque particule se déplace et à chaque itération, la plus proche de l'optimum communique aux autres sa position pour qu'elles modifient leur trajectoire. Cette idée veut qu'un groupe d'individus peu intelligents puisse posséder une organisation globale complexe.
De part sa récence, de nombreuses recherches sont faites sur la P.S.O., mais la plus efficace jusqu'à maintenant est l'élargissement au cadre de l'optimisation combinatoire. En effet, en 2000, Maurice Clerc, un chercheur de France Telecom met en place la D.P.S.O (Discrete Particle Swarm
Optimization), en remplaçant les points par des ordonnancements et les fonctions continues par des fonctions d'évaluation.
Nous avons donc, afin de comprendre l'efficacité et les limites de cette approche, appliqué cette méthode au Problème du Voyageur de Commerce, un problème d'optimisation combinatoire, qui, à priori, est très mauvais pour ce genre d'heuristique d'optimisation.

Eléments P.S.O

Pour appliquer la PSO il faut définir un espace de recherche constitué de particules et une
fonction objectif à optimiser. Le principe de l'algorithme est de déplacer ces particules afin qu'elles
trouvent l'optimum. Chacune de ces particules est dotée :

- D'une position, c'est-à-dire ses coordonnées dans l'ensemble de définition.
- D'une vitesse qui permet à la particule de se déplacer. De cette façon, au cours des itérations, chaque particule change de position. Elle évolue en fonction de son meilleur
  voisin, de sa meilleure position, et de sa position précédente. C'est cette évolution qui
  permet de tomber sur une particule optimale.

- D'un voisinage, c'est-à-dire un ensemble de particules qui interagissent directement sur la
  particule, en particulier celle qui a le meilleur critère.

A tout instant, chaque particule connait :

- Sa meilleure position visitée. On retient essentiellement la valeur du critère calculée ainsi
  que ses coordonnées.
- La position du meilleur voisin de l'essaim qui correspond à l'ordonnancement optimal.

- La valeur qu'elle donne à la fonction objectif car à chaque itération il faut une comparaison
  entre la valeur du critère donnée par la particule courante et la valeur optimale.

On se rend compte, en accord avec Maurice Clerc et Patrick Siarry [2], que l'évolution d'une particule est finalement une combinaison de trois types de comportements : égoïste (suivre sa
voie suivant sa vitesse actuelle), conservateur (revenir en arrière en prenant en compte sa meilleure
performance) et panurgien (suivre aveuglement le meilleur de tous en considérant sa performance).
On voit alors que la bio-inspiration à l'origine de l'optimisation par essaim particulaire ressort dans
l'algorithme sous la forme d'une intelligence collective : coordination du groupe, instinct individuel
et interaction locale entre les individus (grognements, phéromones…).

Notion de voisinage

Le voisinage d'une particule est le sous-ensemble de particules de l'essaim avec lequel il a
une communication directe. Ce réseau de rapports entre toutes les particules est connu comme la
sociométrie, ou la topologie de l'essaim.
Il existe deux principaux types de voisinage :

- Les voisinages géographiques : les voisins sont considérés comme les particules les
  plus proches. Cependant, à chaque itération, les nouveaux voisins doivent être recalculés à partir d'une distance prédéfinie dans l'espace de recherche. C'est donc
  un voisinage dynamique.
- Les voisinages sociaux : les voisins sont définis à l'initialisation et ne sont pas modifiés ensuite. C'est le voisinage le plus utilisé, pour plusieurs raisons :

  - Il est plus simple à programmer.
  - Il est moins coûteux en temps de calcul.

- En cas de convergence, un voisinage social tend à devenir un voisinage géographique

Pour ce faire, on dispose (virtuellement) les particules en cercle puis, pour la particule étudiée, on inclut progressivement dans ses informatrices, d'abord ellemême,
puis les plus proches à sa droite et à sa gauche, jusqu'à atteindre la taille voulue.
On peut aussi choisir les informatrices au hasard.

Principe fondamental

Nous allons donc nous intéresser à cette méta-heuristique dans le cadre où elle a, avant tout,
été définie à savoir l'optimisation de fonctions continues, telle qu'elle est définie dans les travaux
de Michel Gourgand et Sylverin Kemmoé Tchomté [4].
L'algorithme de base de la P.S.O. travaille sur une population appelée essaim de solutions
possibles, elles-mêmes appelées particules. Ces particules sont placées aléatoirement dans l'espace
de recherche de la fonction objectif.
A chaque itération, les particules se déplacent en prenant en compte leur meilleure position
(déplacement égoïste) mais aussi la meilleure position de son voisinage (déplacement panurgien).
Dans les faits, on calcule la nouvelle vitesse à partir de la formule suivante :

V_k+1 = c_1*V_k + c_2 *(best_particule - position_particule) + c_3 *(best_voisin - position_particule)

Où : V_k+1 et V_k sont les vitesses de la particule aux itérations k et k+1.
best_particule est la meilleure position de la particule
best_voisin est la meilleure position de son voisinage à l'itération k
position_particule est la position de la particule à l'itération k
c_1, c_2, c_3 sont des coefficients fixés, c_2 est généré aléatoirement à chaque itération et, en
général, c_3 = c_2

On peut ensuite déterminer la position suivante de la particule grâce à la vitesse que l'on
vient de calculer :

$$X_{k+1} = X_k + V_{k+1}$$

Où : X_k est la position de la particule à l'itération k
On génère X_0 et V_0 au début de notre algorithme.

Algorithme

L'algorithme de base est très simple :

On note g la meilleure position connue de l'essaim et f(x) la fonction qui calcule le critère de x.

Pour chaque particule :

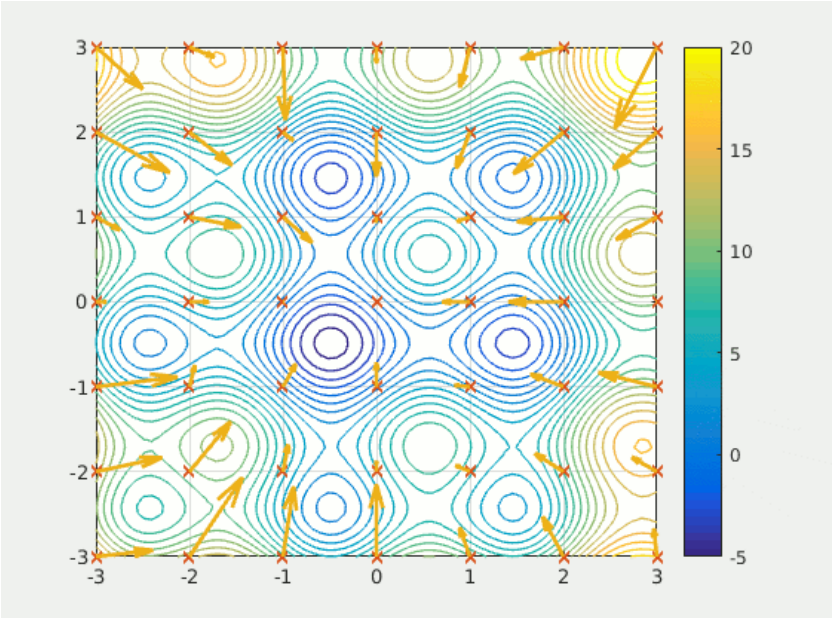
On initialise sa position
On initialise sa meilleure position p connue comme étant sa position initiale
Si f(p) < f(g), on met à jour la meilleure position de l'essaim
On initialise la vitesse de la particule.

Tant que l'on n'a pas atteint l'itération maximum ou une certaine valeur du critère :

Pour chaque particule i :

On tire aléatoire c2 et c3
On met à jour la vitesse de la particule suivant la formule vue précédemment
On met à jour la position xi
Si f(xi) < f(pi),
On met à jour la meilleure position de la particule
Si f(pi) < f(g), on met à jour la meilleure position de l'essaim
g est l'optimum.

ParticleSwarmArrowsAnimation.gif



Configuration de la méthode

A première vue, il semblerait que de nombreux paramètres sont à prendre en compte pour l'application de l'optimisation par essaim particulaire. Toutefois, la plupart d'entre eux peuvent être fixés, d'autres au contraire ne peuvent être définis qu'empiriquement.

C'est le cas, par exemple de la taille de l'essaim. La quantité de particules allouées dépend essentiellement de deux paramètres : la taille de l'espace de définition, et le rapport entre la capacité de calcul de la machine et le temps maximum de recherche. Le meilleur moyen d'affiner ce coefficient est donc de faire de nombreux essais afin de se doter de l'expérience nécessaire. Il faut aussi d'autre part considérer l'initialisation de l'essaim ; elle est généralement faite aléatoirement suivant une loi uniforme sur [0,1], cependant une répartition homogène des particules est préférable, comme avec l'utilisation d'un générateur de séquence de SOBOL.

Deux autres paramètres importants sont les coefficients de confiance que l'on nomme précédemment c2 et c3. Ils permettent de pondérer les tendances des particules à suivre leur instinct de conservation ou leur panurgisme. De manière générale, ces variables aléatoires sont évaluées à chaque itération suivant une loi uniforme sur le domaine de définition.

De même un paramètre important à prendre en compte est le coefficient d'inertie appelé c_1 dans la formule vue auparavant. Il permet de définir la capacité d'exploration de chaque particule en vue d'améliorer la convergence de la méthode. Fixer ce paramètre revient à trouver un compromis entre une exploration globale (c1 > 1) et une exploration locale (c_1 < 1). Il représente l'instinct aventureux de la particule.

Enfin, il reste à configurer le critère d'arrêt. En effet, la convergence vers la solution optimale globale n'est pas garantie dans tous les cas. Il est donc important de doter l'algorithme d'une porte de sortie en définissant un nombre maximum d'itérations. Le programme s'arrête alors si et seulement si le nombre maximum d'itérations est atteint ou que la valeur du critère obtenue est acceptable pour l'utilisateur.

Référence

Ce cours est issu du document "L'optimisation par essaimparticulaire pour des problèmes d'ordonnancement" écrit par : Maxime BOMBRUN et Abdoulaye SENE.

Les réseaux de neurones

Introduction

Il existe plusieurs types de réseaux de neurones. Nous décrivons ici l'utilisation de l'un d'entre-eux, le perceptron multi-couches. Un autre type de réseau de neurones, le réseau de Kohonen dans le cadre de l'apprentissage non supervisé.

Malgré leur nom, les réseaux de neurones n'ont qu'un très lointain rapport avec celui que nous avons dans notre tête, nous humains, ou même les vers de terre. Aussi, il ne faut pas s'attendre à des propriétés extraordinaires et miraculeuses que la difficulté d'appréhension du fonctionnement d'un réseau de neurones pourrait entraı̂ner dans des esprits non prévenus. Nous commençons par décrire l'unité de base d'un réseau de neurones, le neurone formel.

Le neurone formel

Nous considérons ici un type particulier de neurone formel, le perceptron. Un perceptron est une unité de traitement qui a les caractéristiques suivantes :

- il possède P + 1 entrées que l'on notera e_{i∈{0,...P }} . L'entrée e_0 est particulière : on la nomme le biais ou seuil et elle vaut toujours;
- il possède une sortie notée s;
- chacune des entrées est pondérée par un poids noté w_{i∈{0,...P }} ∈ R ;
-

une fonction d'activation, notée φ(.), détermine la valeur de s en fonction des valeurs présentes en entrée pondérées par leur potentiel :

$$s = \varphi(\sum_{i=0}^{i=P} w_i e_i)$$

pp

- on nommera potentiel la valeur v paramètre de φ. Donc, la sortie s est obtenue par l'application de la fonction d'activation à ce potentiel.

Le fonctionnement d'un perceptron est très simple : il fonctionne de manière itérative et, à chaque itération, calcule sa sortie en fonction de ses entrées. Dans une tâche de classification, cette sortie (numérique forcément) indique la classe prédite pour la donnée qui avait été placée en entrée sur les e_i.

Notons que la sortie du perceptron ne dépend que des poids w_i (une fois l'entrée fixée bien entendu). Aussi, l'apprentissage consiste ici à trouver la valeur des poids qui fait en sorte que lorsqu'une donnée est placée sur l'entrée du perceptron, la sortie prédit correctement sa classe. Donc, pour un perceptron, apprendre signifie fixer la valeur de ces P + 1 paramètres réels.

Pour une tâche de classification, il est d'usage d'utiliser un perceptron dont la sortie s ∈ {0, 1} ou un perceptron dont la sortie s ∈ {−1, 1}. D'un point de vue théorique, utiliser l'un ou l'autre ne change rien ; d'un point de vue pratique, c'est moins simple.

La fonction d'activation peut être la fonction de Heaviside :

$$\varphi(v) = \begin{cases} 1 \text{ si } v \geq 0 \\ 0 \text{ si } v < 0 \end{cases}$$

si $s \in \{0, 1\}$, ou

$$\varphi(v) = \begin{cases} 1 \text{ si } v \geq 0 \\ -1 \text{ si } v < 0 \end{cases}$$

si $s \in \{-1, 1\}$.

Ce peut être aussi une fonction sigmoide dite logistique :

$$\varphi(v) = \frac{1}{1 + e^{-av}}$$

avec a ∈ R si s ∈ [0, 1], ou tangente hyperbolique :

$$\varphi(v) = \tanh(v)$$

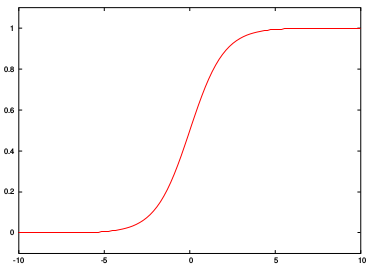
si s ∈ [−1, 1].

Ces deux dernières fonctions d'activation ont l'immense avantage d'être continues et dérivables autant de fois que l'on veut sur R. De plus, leurs dérivées s'expriment facilement en fonction d'elles-mêmes ce qui nous sera utile un peu plus loin.

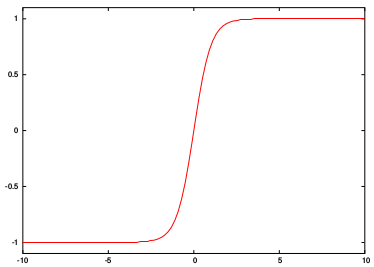
Enfin, ce peut être simplement la fonction linéaire φ(v) = v. Dans ce cas, la sortie du neurone est la somme pondérée des entrées. On parle alors de neurone linéaire . La sortie prend alors une valeur dans R.

Pour prédire la classe d'une donnée placée en entrée, on regarde la sortie du perceptron (classification binaire). Si on a utilisé une fonction d'activation de type seuil, chacune des deux valeurs possibles en sortie correspond à une classe. Si on a utilisé une fonction d'activation logistique (resp. tanh), on peut dire que la classe est donnée par la valeur extrême la plus proche, 0 (resp. -1) ou 1 (resp. 1). Avec un neurone linéaire, on peut classer l'entrée en fonction du signe de la sortie.

Notons tout de suite que l'équatio du potentiel v définit un hyperplan dans un espace à P + 1 dimensions, chacune correspondant à un attribut. Comme on l'a dit plus haut, notre objectif ici est de déterminer les w i pour prédire correctement la classe des données placées en entrée du perceptron. Une fois correctement appris, ces poids déterminent donc un hyperplan somme(w_i e_i) qui sépare les exemples positifs des exemples négatifs : idéalement, ils sont de part et d'autre de l'hyperplan. Cependant, il arrive, et c'est le cas général, que cet hyperplan n'existe pas : dans ce cas, on dit que les exemples ne sont pas linéairement séparables . Dans ce cas, on doit utiliser plusieurs perceptrons, autrement dit, un réseau de perceptrons . On commence par présenter l'apprentissage des poids par un perceptron avant de passer aux réseaux de neurones.



(a) fonction logistique



(b) fonction tanh

n'existe pas : dans ce cas, on dit que les exemples ne sont pas linéairement séparables . Dans ce cas, on doit utiliser plusieurs perceptrons, autrement dit, un réseau de perceptrons . On commence par présenter l'apprentissage des poids par un perceptron avant de passer aux réseaux de neurones.

Apprentissage des poids d'un perceptron

Cas séparable

On peut trouver les w_i par inversion de matrice, mais chacun sait que dans la pratique, l'inversion d'une matrice est quelque chose à éviter à tout prix. Un autre

algorithme simple pour trouver ces w_i est la règle d'apprentissage du perceptron (cf. algorithme 4).

Le passage de l'ensemble des exemples (la boucle pour de la règle d'apprentissage du perceptron) se nomme un épisode.

Notons bien que les exemples sont mélangés avant la boucle pour, ce qui signifie que cette boucle ne les passe pas toujours dans le même ordre. Ce point est très important pour que l'algorithme fonctionne. Notons aussi que ce faisant, nous introduisons du hasard dans cet algorithme : dans ce cours, c'est la première fois que nous étudions un algorithme dont le fonctionnement repose sur du hasard. Une autre source de hasard ici est l'initialisation des poids. Ce type d'algorithme dont le fonctionnement repose au moins en partie sur le hasard est qualifié de non déterministe , ou encore stochastique . Tous les algorithmes concernant les réseaux de neurones sont non déterministes. Une conséquence de cela est que deux exécutions successives de l'algorithme sur le même jeu de données donne généralement des résultats différents.

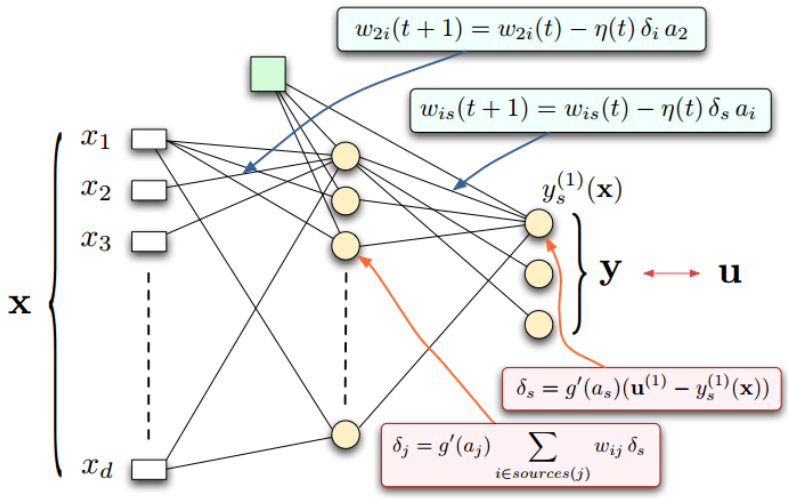
Algorithme 4 Règle d'apprentissage du perceptron

Nécessite: les N exemples d'apprentissage $\mathcal{X}$. Chaque exemple x_i possède P attributs dont les valeurs sont notées $x_{i,j}$. Pour chaque donnée, $x_{i,0}$ est un attribut virtuel, toujours égal à 1. La classe de l'exemple x_i est y_i.
Nécessite: taux d'apprentissage $\alpha \in]0, 1]$
Nécessite: un seuil ϵ
  initialiser les w_i aléatoirement
  répéter
    // E mesure l'erreur courante
    $E \leftarrow 0$
    mélanger les exemples
    pour tous les exemples du jeu d'apprentissage $\mathcal{X}$ **faire**
      $E_i \leftarrow y_i - \varphi(\sum_{j=0}^{j=P} w_j x_{i,j})$
      $E \leftarrow E \ + |E_i|$
      pour tous les poids $w_k, k \in \{0, 1, ...P\}$ **faire**
        $w_k \leftarrow w_k + \alpha \ E_i \ x_{i,k}$
      fin pour
    fin pour
  jusque $E < \epsilon$

Cas non-séparable

C'est seulement en 1986 que la généralisation de la règle delta aux réseaux à couches cachées a été formulée. Cette généralisation, la règle de la rétropropagation du gradient de l'erreur, consiste à propager l'erreur obtenue à une unité de sortie d'un réseau à couches comportant une ou plusieurs couches cachées à travers le réseau par descente du gradient dans le sens inverse de la propagation des activations.

Rétropropagation de l'erreur



Rappellons qu'un réseau à couches est composé d'un ensemble de neurones formels groupés
en sous-ensembles distincts (les couches) de telle sorte qu'il n'y ait aucune connexion entre deux
neurones d'une même couche.

À la fin de l'apprentissage, lorsque le réseau a appris à modéliser son environnement, le comportement
souhaité du réseau est le suivant : on présente un vecteur d'entrée au réseau, celui-ci propage vers la sortie les valeurs d'activation correspondantes (en utilisant une règle de propagation),
afin de générer, par l'intermédiaire des neurones de sortie, un vecteur de sortie. Celui-ci
devrait correspondre à la sortie désirée, telle qu'apprise lors de la phase d'apprentissage.

La généralisation de la règle delta aux réseaux multicouches utilise une méthode de descente
du gradient, permettant de calculer la modification des poids des connexions entre les couches
cachées. Afin de pouvoir calculer
le gradient de l'erreur par rapport aux poids du réseau, la fonction de sortie d'un neurone doit être
différentiable et non linéaire (sinon, on pourrait réduire le réseau à un perceptron).

La fonction
la plus souvent utilisée est, comme on l'a déjà dit, la sigmoïde.

Algorithme 7 Algorithme de rétro-propagation du gradient de l'erreur (version stochastique) pour un perceptron multi-couches ayant $P+1$ entrées (P attributs + le biais), $q+1$ couches numérotées de 0 (C_0 : couche d'entrée) à q (C_q : couche de sortie) et une seule sortie ; notation : $s(x_i)$ est la sortie du PMC pour la donnée x_i, $s_{l,k}$ est la sortie de la k^e unité (entrée ou neurone) de la couche l, $w_{l,k,m}$ est le poids de la connexion entre l'unité k de la couche l et le neurone m de la couche $l+1$ ($k=0$ correspond au biais), $|C_l|$ est le nombre d'unités composant la couche C_l. L'algorithme donné ici correspond à des neurones à fonction d'activation logistique $\frac{1}{1+e^{-x}}$.

Nécessite: les N instances d'apprentissage $\mathcal{X}$
Nécessite: taux d'apprentissage $\alpha \in]0,1]$
  initialiser les w_i
  tant-que critère d'arrêt non rempli **faire**
    mettre à jour α
    mélanger les exemples
    pour tout exemple x_i **faire**
      $s(x_i) \leftarrow$ sortie du réseau pour l'exemple x_i
      $\delta_{q,1} \leftarrow s(x_i)(1-s(x_i))(y_i - s(x_i))$
      pour toutes les couches cachées : l décroissant de $q-1$ à 1 **faire**
        pour tous les neurones k de la couche C_l **faire**
          $\delta_{l,k} \leftarrow s_{l,k}(1-s_{l,k}) \sum_{m \in \{0,\ldots,|C_{l+1}|\}} w_{l,k,m}\ \delta_{l+1,m}$
        fin pour
      fin pour
      // mise à jour des poids
      pour toutes les couches l croissant de 0 à $q-1$ **faire**
        pour toutes les unités k de la couche l, k variant de 1 à $|C_l|$ **faire**
          pour tous les neurones m connectés sur la sortie du neurone k de la couche l, m variant de 1 à $|C_{l+1}|$ **faire**
            $w_{l,k,m} \leftarrow w_{l,k,m} + \alpha\ \delta_{l+1,m}\ s_{l,k}$
          fin pour
        fin pour
      fin pour
    fin pour
  fin tant-que

Initialisation des poids

L'initialisation des poids n'est pas un simple détail (en fait, dans l'utilisation des PMC, rien n'est un détail...). Le Cun et al. [1998] conseille d'initialiser chaque poids avec une valeur tirée aléatoirement dans une distribution gaussienne centrée (de moyenne nulle) et d'écart-type égal à la racine carrée du nombre d'entrées de l'unité à l'entrée de laquelle se trouve ce poids.
Par exemple, les poids des neurones de la couche C1 seront initialisés en tirant des nombres pseudo-aléatoires d'une distribution normale de moyenne nulle et de variance égale à 6.

L'objectif est que le potentiel des neurones soit proche du point d'inflexion de la sigmoïde (en 0 pour tanh).

Prétraitement des sorties

Utilisant une fonction d'activation tanh, on pense naturellement à coder la classe des données par +1 et −1. Cependant, ±1 sont des valeurs asymptotiques pour la fonction tanh. Aussi, elles ne sont jamais atteintes et donc l'apprentissage n'est pas possible... Au lieu d'utiliser ces valeurs, on utilise généralement des valeurs un peu plus petite que 1 en valeur absolue ; on pourra utiliser ±0.6 par exemple.

Plus de deux classes

Si le nombre de classes est supérieur à 2, on utilise |Y| sorties. Chaque sortie correspond à une classe. Une donnée étant placée en entrée, la sortie la plus active indique la classe prédite.
Si l'on veut, on utilise une unité dite winner-takes-all qui prend en entrée les |Y| sorties du réseau et fournit en sortie le numéro de son entrée la plus active : c'est le numéro de la classe prédite.

Quelques problèmes

L'algorithme de rétro-propagation du gradient de l'erreur trouve un optimum local. Plusieurs exécutions en initialisant les poids différemment ou en présenter les exemples dans des ordres différents peuvent entraîner des convergences vers des poids différents (rappelons que les exemples sont mélangés à chaque épisode).
Si l'on n'y prend pas garde, on va rencontrer des problèmes de surapprentissage. Deux approches sont utilisées :

arrêt précoce : (early stopping en anglais) c'est une approche expérimentale : on utilise un jeu de validation compos´e d'exemples, disjoint du jeu d'apprentissage. Durant l'apprentissage, on mesure l'erreur sur le jeu de validation.
Quand cette erreur stagne, on arrête l'apprentissage. Remarquons que si l'on mesure l'erreur sur le jeu d'apprentissage, au moment o`u l'erreur sur le jeu de validation stagne, l'erreur sur le jeu d'apprentissage continue `a diminuer. C'est exactement cela le "sur-apprentissage" ;

régularisateur : c'est une approche qui s'appuie sur des résultats théoriques qui consiste `a minimiser une fonction combinant la mesure de l'erreur sur le jeu d'apprentissage avec la valeur des poids : on cherche une erreur minimale et des poids les plus petits possibles, soit $E = E_{app} + \lambda \sum_i w_i^2$

o`u
λ est une constante qui permet d'équilibrer l'importance des deux termes
et les wi représentent tous les poids du réseau. Typiquement, on teste
plusieurs valeurs de λ. Cette technique se nomme weight decay en anglais.
D'autres techniques de régularisation sont envisageables.

Le problème de l'apprentissage des poids : On pourra également consulter Sarle [1997a], Peterson et al. [1993], Le Cun et al. [1998] pour avoir des points de vue pratiques sur la mise en œuvre des réseaux de neurones ; très souvent, les expos´es des réseaux de neurones sont très académiques et ne permettent pas de les appliquer directement ; leur utilisation nécessite beaucoup de savoir-faire.

Avantages du MLP

Propriété 3 : Toute fonction booléenne peut être apprise sans erreur par un perceptron multi-couches ayant 1 seule couche cachée.

Cependant, ce résultat est essentiellement théorique. En effet, on démontre aussi qu'une fonction à P entrées peut nécessiter une couche cachée possédant O(2**P) neurones. Cela signifie que le calcul de cette fonction peut être non polynomial,
autrement dit, prendre beaucoup trop de temps (cf. annexe C).
Dans la pratique, pour éviter des temps d'apprentissage démeusurément longs, on utilise une deuxième couche cachée, voire plus.

Par ailleurs, Hornik [1991] a montré qu'un perceptron multi-couches avec 1 seule couche cachée de perceptrons sigmoïdes et un neurone linéaire en sortie peut calculer n'importe quelle fonction réelle et bornée, autrement dit, n'importe quelle séparatrice.
Donc, cela signifie que tout problème de classification peut être résolu par un perceptron constitué d'une couche cachée d'unités dont la fonction d'activation est une sigmoïde et d'une unité linéaire en sortie.

Donc, on a des théorèmes montrant que l'apprentissage de toute séparatrice avec différents types de perceptrons ayant une couche cachée d'unité dont la fonction d'activation est non linéaire et une sortie linéaire est possible. Cependant, ces théorèmes n'indiquent pas le nombre d'unités à mettre dans la couche cachée.

Files | A destination des étudiants : le programme Python pour le PMC

MLP.py