Dispersive Flies Optimisation

Taif Mohamed Amine[1] **Koussy Ayoub**[1]

L'optimisation est un domaine clé dans de nombreux domaines, allant de l'ingénierie aux sciences de la gestion. Son objectif est de trouver la meilleure solution possible parmi un ensemble de choix possibles, en tenant compte de certaines contraintes et objectifs spécifiques. Ce rapport présente une comparaison entre deux approches différentes pour l'entraînement des réseaux neuronaux : l'entraînement classique et l'optimisation par dispersion des mouches (DFO). L'objectif principal de cette étude est de déterminer quelle méthode permet d'obtenir le meilleur ensemble de poids optimal pour les réseaux neuronaux.

1. Introduction

L'optimisation est l'un des aspects les plus importants de la formation des modèles d'apprentissage automatique. Elle permet non seulement d'améliorer les performances, mais joue également un rôle significatif dans la configuration d'un modèle donné et aboutit à une meilleure précision tant dans la phase d'entraînement que dans la phase de test.

Particulèrement, l'optimisation par intelligence collective est une approche innovante dans le domaine de l'optimisation qui tire parti de la collaboration et de la coopération entre plusieurs agents pour résoudre des problèmes complexes. Elle s'inspire des principes de l'intelligence collective, où chaque agent contribue de manière autonome à la recherche de la meilleure solution globale.

Cette approche offre de nombreux avantages dans le domaine de l'optimisation. Elle permet d'explorer rapidement l'espace des solutions, de trouver des solutions de qualité et de s'adapter à des environnements changeants. De plus, elle est flexible et peut être adaptée à différents types de problèmes, qu'ils soient continus, discrets, mono-objectif ou multi-objectifs.

1.1. Dispersive Flies Optimisation

Dispersive Flies Optimization (DFO) est un algorithme d'optimisation métaheuristique inspiré de la nature, développé par le Dr. Wissam Alsalimi en 2014. L'idée principale derrière le DFO est de simuler le comportement des mouches à la recherche de nourriture dans un espace de recherche multidimensionnel. L'algorithme commence par une population initiale de mouches, représentant chacune une solution potentielle au problème d'optimisation générée de façon aléatoire. Les mouches sont ensuite dispersées dans l'espace de recherche, et leurs positions sont mises à jour de manière itérative en fonction de leurs valeurs de fitness qui mesure l'optimalité de leurs position et les positions de leurs voisines.

Le DFO présente plusieurs avantages par rapport à d'autres algorithmes d'optimisation :

- Simplicité : L'algorithme est relativement simple à comprendre et à mettre en œuvre, ce qui le rend accessible à un large éventail d'utilisateurs.
- Flexibilité : Le DFO peut être appliqué à une variété de problèmes d'optimisation, y compris des problèmes continus, discrets et combinatoires vu son espace de recherche multidimensionnel.
- Robustesse : L'algorithme est moins susceptible de rester coincé dans des optima locaux par rapport à d'autres techniques d'optimisation, grâce à son équilibre entre exploration et exploitation.

1.2. Modélisation mathématique de DFO

L'algorithme DFO est basé sur la mise à jour des positions des mouches (solutions candidates) dans l'espace de recherche suivant des conditions pseudo-aléatoires. La position d'une mouche i à l'itération t est représentée par le vecteur $\mathbf{x}_i^t = (x_{i1}^t, x_{i2}^t, \ldots, x_{id}^t)$, où d représente la dimension du problème d'optimisation.

La mise à jour de la position d'une mouche est basée sur la somme pondérée de trois composantes: la position actuelle de la mouche, la meilleure position trouvée par la mouche et la position moyenne des voisines de la mouche. La nouvelle position $\mathbf{x}_i^{t+1}$ de la mouche i à l'itération $t+1$ est calculée comme suit:

$$\mathbf{x}_i^{t+1} = \alpha \mathbf{x}_i^t + \beta \mathbf{p}_i^t + \gamma \mathbf{m}_i^t$$

où α, β et γ sont des coefficients qui contrôlent l'influence de chaque composante, $\mathbf{p}_i^t$ est la meilleure position trouvée par la mouche i jusqu'à l'itération t, et $\mathbf{m}_i^t$ est la position moyenne des voisines de la mouche i à l'itération t. La

position moyenne des voisines de la mouche i est calculée comme suit:

$$\mathrm{m}_i^t = \frac{1}{N} \sum_{j=1}^{N} \mathbf{x}_j^t$$

où N est le nombre de voisines considérées pour la mouche i. Après la mise à jour de la position, la nouvelle valeur de fitness de la mouche est évaluée. Si la nouvelle valeur de fitness est meilleure que l'ancienne, la meilleure position de la mouche est mise à jour:

$$\mathrm{p}_i^{t+1} = \begin{cases} \mathbf{x}_i^{t+1}, & \text{si } f(\mathbf{x}_i^{t+1}) < f(\mathbf{p}_i^t) \\ \mathbf{p}_i^t, & \text{sinon} \end{cases}$$

où $f(\cdot)$ est la fonction de fitness à optimiser.

L'algorithme DFO se poursuit jusqu'à ce qu'un critère d'arrêt soit atteint, comme un nombre maximal d'itérations ou une tolérance de convergence. À la fin, la meilleure mouche globale et sa valeur de fitness sont retournées comme solution optimale.

1.3. Pseudo code

L'algorithme commence par une population initiale de mouches, représentant chacune une solution potentielle au problème d'optimisation. Les mouches sont ensuite dispersées dans l'espace de recherche, et leurs positions sont mises à jour de manière itérative en fonction de leurs valeurs de performance et des positions de leurs voisines comme mentionné précédemment.

Voici un aperçu global de l'algorithme DFO :

1. Initialiser la population de mouches avec des positions aléatoires dans l'espace de recherche.
2. Évaluer la performance de chaque mouche (c'est-à-dire la qualité de la solution qu'elle représente).
3. Déterminer la meilleure mouche (celle ayant la plus grande valeur de performance).
4. Mettre à jour les positions des mouches en fonction de leurs valeurs de performance et des positions de leurs voisines.
5. Répéter les étapes 2 à 4 jusqu'à ce qu'un critère d'arrêt soit atteint (par exemple, un nombre maximal d'itérations ou un niveau de qualité de solution souhaité).

Algorithm 1 Dispersive flies optimization

procedure DFO (N, D, $\vec{x}_{\min}$, $\vec{x}_{\max}$, f)*
  for $i = 0 \to N-1$ **do** ▷ Initialization
    for $d = 0 \to D-1$ **do**
      $x_{id}^0 \leftarrow \mathrm{U}(x_{\min,d}, x_{\max,d})$
    end for
  end for
  while ! termination criteria **do** ▷ Main DFO loop
    for $i = 0 \to N-1$ **do**
      $\vec{x}_i$.fitness $\leftarrow f(\vec{x}_i)$
    end for
    $\vec{x}_s = \arg\min [f(\vec{x}_i)], \quad i \in \{0, 1, 2, \ldots, N-1\}$
    for $i = 0 \to N-1$ and $i \neq s$ **do**
      $\vec{x}_{i_n} = \arg\min [f(\vec{x}_{(i-1)\%N}), f(\vec{x}_{(i+1)\%N})]$
      for $d = 0 \to D-1$ **do**
        if $\mathrm{U}(0,1) < \Delta$ **then**
          $x_{id}^{t+1} \leftarrow \mathrm{U}(x_{\min,d}, x_{\max,d})$ ▷ Restart
        else
          $u \leftarrow U(0,1)$
          $x_{id}^{t+1} \leftarrow x_{i_n d}^t + u(x_{sd}^t - x_{id}^t)$ ▷ Update
          if $x_{id}^{t+1} < x_{\min,d}$ or $x_{id}^{t+1} > x_{\max,d}$ **then**
            $x_{id}^{t+1} \leftarrow \mathrm{U}(x_{\min,d}, x_{\max,d})$
          end if
        end if
      end for
    end for
  end while
  return $\vec{x}_s$
end procedure
* INPUT: swarm size, dimensions, lower/upper bounds, fitness function.

Figure 1. Pseudo code de DFO

2. L'implémentation

Dans cette section, on va élaborer notre hybridation de réseau neuronnes et Dispersive flies optimization.

2.1. Réseau neuronnes + DFO

Pour créer cette hybridation, nous avons choisi d'implémenter notre système à partir de zéro afin de bénéficier d'une plus grande flexibilité dans notre étude.

- La classe `DFO` est la classe responsable de l'initialisaton de notre environnement et la mise à jour des positions des agents, elle est inialisée comme suit:
  - `num_flies`: Le nombre de mouches (poids) dans l'environnement d'optimisation.
  - `fitness_function`: La fonction d'évaluation de la performance du réseau de neurones.
  - `NN_structure`: La structure du réseau de neurones pour savoir la dimensionnalité de notre espace de recherche.
  - `input_size`: Le nombre de caractéristique de notre données d'entrainement.
  - `env_bounds`: Les limites des valeurs des poids de notre réseau neuronnes.
  - `initial_weights`: Les poids initiaux du réseau de neurones (facultatif).
  - `delta`: Le seuil de perturbation utilisé pour explorer de nouvelles solutions potentielles.
  - `max_iter`: Le nombre maximal d'itérations.

- Une méthode `initialize_flies` qui est utilisée pour initialiser les positions des mouches (poids) en suivant une distribution uniforme bornée. Elle génère des poids aléatoires pour chaque mouche en respectant la structure du réseau de neurones.
- La méthode `train` effectue l'optimisation des poids du réseau de neurones en utilisant l'algorithme d'optimisation des mouches. Elle prend en compte les paramètres `Xt` et `Yt`, qui représentent les données d'entraînement et les étiquettes correspondantes. La méthode itère sur le nombre maximal d'itérations spécifié :
  - À chaque itération, elle calcule la performance (fitness) de chaque mouche en utilisant la fonction d'évaluation fournie.
  - Elle extrait la position de la meilleure mouche ayant la plus faible performance (fitness).
  - Si la performance de cette mouche est meilleure que la meilleure performance enregistrée jusqu'à présent, les poids de cette mouche deviennent les meilleurs poids.
  - Ensuite, elle procède à la perturbation des mouches en fonction du seuil delta. Certaines mouches sont sélectionnées aléatoirement pour être perturbées et leurs poids sont modifiés en conséquence.
  - Les mouches non perturbées mettent à jour leurs poids en se basant sur la meilleure mouche et la perturbation aléatoire.
  - La méthode affiche la performance de la meilleure mouche à chaque itération ainsi que le nombre de mouches perturbées.
  - Finalement, elle renvoie les meilleurs poids, la meilleure performance et les positions de toutes les mouches.
- La classe `DFO_NN` qui implémente notre réseau neuronnes et notre fonction de fitness.

2.2. Réseaux neuronnes conventionnel

Les réseaux neuronaux traditionnels, basés sur la propagation avant et la rétropropagation. La propagation avant consiste à transmettre les données d'entrée à travers les différentes couches du réseau, en appliquant une série d'opérations linéaires et non linéaires, telles que les multiplications matricielles et les fonctions d'activation. Chaque couche transforme les données d'entrée pour générer une représentation plus abstraite. Une fois que les données atteignent la couche de sortie, une fonction d'activation finale est appliquée pour obtenir les prédictions du réseau.

Après la propagation avant, la rétropropagation est utilisée pour calculer les gradients par rapport aux poids du réseau. Ces gradients sont ensuite utilisés pour mettre à jour les poids à l'aide d'un algorithme d'optimisation tel que la descente de gradient. La rétropropagation permet d'ajuster les poids du réseau de manière à minimiser l'erreur entre les prédictions du réseau et les valeurs cibles.

Nous avons choisi d'implémenter cette partie en utilisant Keras avec l'optimisateur Adam. Cette décision a été prise car l'utilisation de Keras nous permet de bénéficier d'une interface conviviale et d'une large gamme de fonctionnalités prêtes à l'emploi pour la création et l'entraînement de réseaux neuronaux. L'optimisateur Adam est un choix courant en raison de sa capacité à ajuster automatiquement le taux d'apprentissage pendant l'entraînement, ce qui peut conduire à une convergence plus rapide et à de meilleures performances du modèle.

2.3. Préparation des données

L'encodage numérique des classes de l'iris est une étape importante dans la préparation des données pour l'apprentissage automatique. Étant donné que les algorithmes d'apprentissage automatique ne peuvent traiter que des données numériques, il est nécessaire de convertir les classes de l'iris en une représentation numérique. Dans le cas de l'ensemble de données de l'iris, il y a trois classes possibles : setosa, versicolor et virginica.

3. Experimentation

3.1. datasets utilsé

Le jeu de données "Iris" est l'un des ensembles de données les plus populaires en apprentissage automatique et en exploration des données. Il a été introduit par le statisticien et biologiste Ronald Fisher en 1936 et est souvent utilisé comme exemple d'apprentissage supervisé pour la classification de données.

Le jeu de données Iris se compose de mesures de quatre caractéristiques différentes de fleurs d'iris : la longueur et la largeur des sépales et des pétales. Les trois espèces d'iris incluses dans le jeu de données sont l'iris setosa, l'iris versicolor et l'iris virginica. Chaque échantillon d'iris est décrit par quatre caractéristiques (longueur du sépale, largeur du sépale, longueur du pétale et largeur du pétale), et chaque échantillon est étiqueté avec l'espèce à laquelle il appartient.

Le jeu de données Iris est souvent utilisé pour la classification des fleurs d'iris en fonction de ces caractéristiques. Il est couramment utilisé pour illustrer des concepts tels que la visualisation de données, l'apprentissage automatique supervisé et la validation de modèles. Il est relativement petit, avec 150 échantillons au total, ce qui le rend facile à utiliser pour des expérimentations rapides.

	sepallength	sepalwidth	petallength	petalwidth	class
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa

Figure 2. Extrait de notre données d'apprentissage

3.2. L'architecture de NN utlisée

Afin d'établir une bonne étude entre les deux structures, il se trouve trivial de fixer la même structure de réseau neuronnes. Vu la simplicité de la base de données "Iris", on optera pour une simple structure pour notre réseau neuronnes pour diminuer la compléxité d'entrainement :

- Input layer : nombre de caractéristiques de base de données (4)
- Hidden layer 1 : 10 Neuronnes
- Hidden layer 1 : 10 Neuronnes
- Output layer : nombre de classes (3)

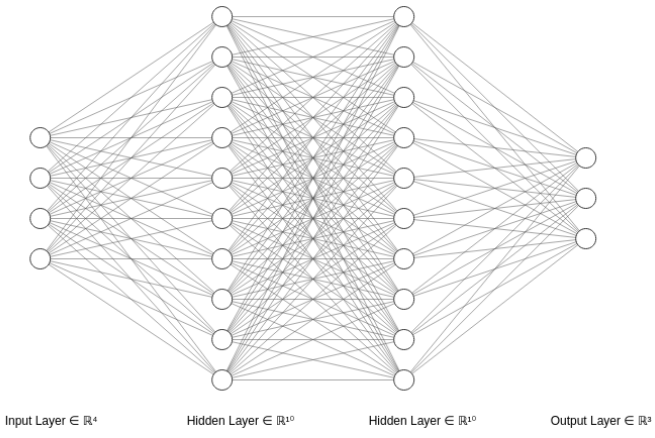

Figure 3. La structure de notre réseau neuronnes

3.3. Résultats et Interprétation

3.3.1. Neural Network conventionnel

Pour une implémentation basique comme mentionné précédemment, on obtient un f1 score de 97.55%. On a utilisé Adam comme optimisateur avec un learning rate de 0.01.

on remarque que le réseau neuronnal convolutionnel donne des réultats stables, avec un temps d'execution très rapide de .

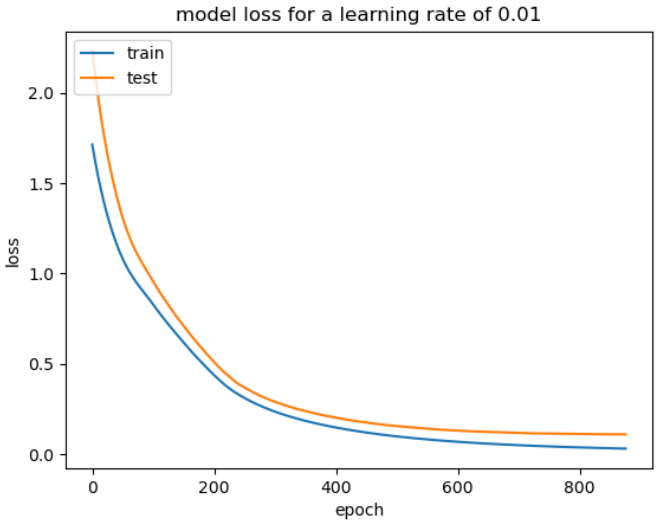

Figure 4. Évolution de perte

3.3.2. Neural Network + DFO

Puisque cette implémentation se base sur une multitude de paramètres, on applique un grid search pour chercher les paramètres optimals. On trouve donc les résultats suivants.

Pour le nombre maximal d'itérations:

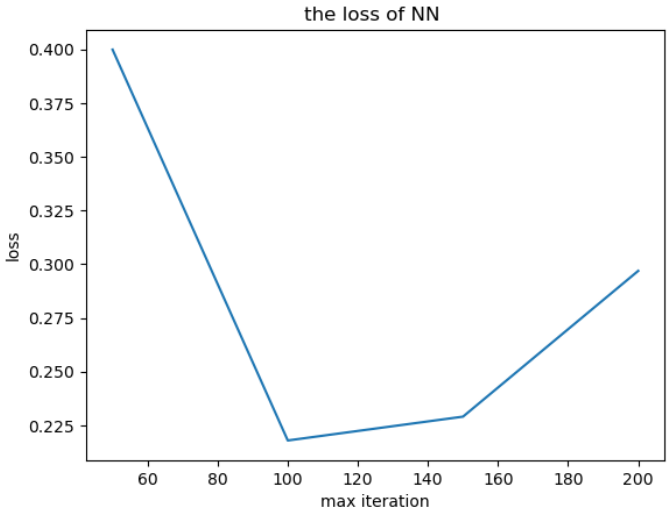

Figure 5. Évolution de perte avec nombre maximal d'itérations

On peut remarquer qu'avoir un nombre modéré d'iterations est nécessaire pour des raison de complexité temporelle ainsi que pour assurer la convergence.

Pour le facteur de perturbation delta, il se trouve important pour équilibrer l'exploration et l'exploitation de l'entrainement, On peut visualiser comment progresse notre

modèle en fonction de ce facteur dans la figure suivante :

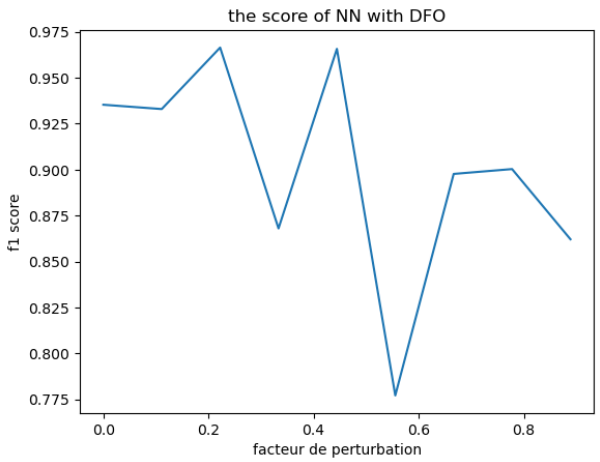

Figure 6. f1 score en fonction de delta

De même pour notre facteur de perturbation, une valeur entre [0.1, 0.45] est recommendée d'après ces résultats, ce facteur joue un role important dans l'exploration de l'espace et se trouve la seule manière de quitter un optimum local.

Pour les paramètres optimals qu'on a obtenu en effectuant un Grid search, on trouve un score f1 de 85.42% pour la version DFO de réseau neuronal, On peut visualiser l'entrainement dans la figure suivante:

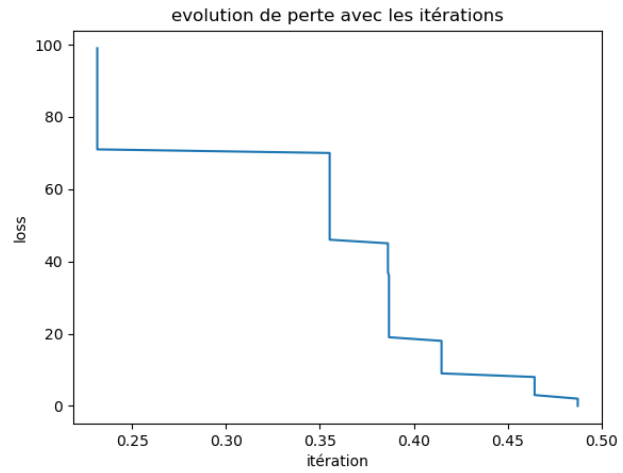

Figure 7. Entrainement avec DFO

On peut aussi remarquer que la perte ne varie pas pendant des intervalles d'iterations, ce qui signifie que le modèle à été dans un optimum local, et à puis s'échapper à l'aide de facteur de perturbation.

4. Conclusion

En conclusion, notre étude comparative entre les réseaux neuronaux conventionnels et les réseaux neuronaux avec l'optimisation par Dispersive Flies (DFO) met en évidence les avantages et les limitations de chaque approche.

Les réseaux neuronaux conventionnels, basés sur la propagation avant et arrière, offrent une méthode bien établie et largement utilisée pour résoudre des problèmes de classification. Ils sont efficaces pour apprendre des relations complexes entre les caractéristiques d'entrée et les étiquettes de sortie. Cependant, ils peuvent être sensibles à l'initialisation des poids et peuvent nécessiter un réglage minutieux des hyperparamètres pour obtenir de bonnes performances.

D'autre part, l'utilisation de l'optimisation par Dispersive Flies (DFO) apporte une approche novatrice pour rechercher l'ensemble optimal de poids dans un réseau neuronal. La nature inspirée par les insectes de l'algorithme DFO permet d'explorer efficacement l'espace de recherche et de trouver des solutions potentiellement meilleures. De plus, DFO offre une certaine robustesse face aux optima locaux grâce à sa capacité à sauter entre différentes régions de l'espace de recherche.

Cependant, notre étude a également révélé que l'application de DFO dans les réseaux neuronaux nécessite une attention particulière à la configuration des paramètres de l'algorithme, tels que la taille de la population de mouches et le nombre d'itérations. De plus, l'optimisation par DFO peut entraîner des coûts computationnels plus élevés par rapport aux réseaux neuronaux conventionnels, en raison du besoin d'exécuter plusieurs itérations de l'algorithme.