HEC MONTRÉAL

# Application de la recherche à voisinage variable pour l'entraînement de réseaux de neurones artificiels

**par**

**Paul Conerardy**

**Gilles Caporossi**
**HEC Montréal**
**Directeur de recherche**

**Sylvain Perron**
**HEC Montréal**
**Co-Directeur de recherche**

**Sciences de la gestion**
**(Spécialisation Science des données et analytique d'affaires)**

*Mémoire présenté en vue de l'obtention*
*du grade de maîtrise ès sciences*
*(M. Sc.)*

Janvier 2022

# Résumé

L'entraînement de réseaux de neurones est le processus d'optimisation par lequel le modèle apprend à effectuer des prédictions qui reflètent les relations contenues dans les données qui lui sont présentées. Cet entraînement correspond à la résolution d'un problème d'optimisation NP-Complet. Ce mémoire cherche à proposer un nouvel algorithme afin de résoudre ce problème qui combine les méthodes qui constituent l'état de l'art actuel avec une métaheuristique, la recherche à voisinage variable. Cette méthode nécessite la définition d'une notion de voisinage dans le cadre de ce problème. Cette dernière sera définie selon la structure du réseau, deux solutions seront considérées voisines en fonction de leurs similitudes entre différentes couches du réseau. Cette définition permet également à la méthode de profiter d'une implémentation plus rapide sur des GPUs. Les résultats obtenus démontrent une capacité de diversification accrue ainsi qu'une capacité de régularisation du processus d'entraînement tout en atteignant souvent des solutions de meilleures qualités.

## Mots-clés

Apprentissage machine, Réseaux de neurones, Métaheuristiques, Recherche à voisinage variable, Optimisation, GPU

# Abstract

Neural network training is the optimization process by which a model learns to make predictions that reflects the relationships contained in the data on which it learns. This training corresponds to the resolution of an NP-Complete optimization problem. This thesis seeks to propose a new algorithm in order to solve this problem which combines state-of-the-art methods with a metaheuristic, the variable neighborhood search. This method requires the definition of a neighborhood structure within the framework of this problem. The latter will be defined according to the structure of the network, two solutions will be considered neighboring solutions based on their similarities between different layers of the network. This definition also allows the method to benefit from a faster implementation on GPUs. The results obtained demonstrate an increased capacity for diversification as well as a capacity to regularize the training process while achieving better quality solutions in most cases.

## Keywords

Machine learning, Neural networks, Metaheuristics, Variable neighborhood search, Optimization, GPU

# Table des matières

# Liste des figures

# Liste des algorithmes

# Liste des abréviations

**ANN** Artificial Neural Network - *Réseau de neurones artificiels*

**MLP** Multilayer Perceptron - *Perceptron Multi-Couches*

**SGD** Stochastic Gradient Descent - *Descente de gradients stochastique*

**BFGS** Algorithme de Broyden-Fletcher-Goldfarb-Shanno

**CIFAR** Canadian Institute For Advanced Research - *Institut canadien de recherche avancée*

**RVV** Recherche à voisinage variables

**DVV** Descente à voisinage variables

**RVF** Recherche à voisinage fixes

**RAM** Random access memory - *Mémoire vive*

**VRAM** Video random access memory - *Mémoire vive vidéo*

**GPU** Graphical Processing Unit - *Unité de traitement graphique*

# Avant-propos

La motivation derrière la réalisation de ce mémoire provient d'un projet de session réalisé dans un cours d'algorithmique donné à la maîtrise. Dans le cadre de la réalisation de ce projet, j'avais fait le choix de présenter l'entraînement de réseaux de neurones à l'aide d'algorithmes génétiques. Mon objectif était de combiner en un sujet les notions d'optimisation et d'apprentissage machine couvertes par différents cours du cursus. A l'issue de ce projet j'avais pu identifier de nombreuses ressources et recherches pour diverses métaheuristiques appliquées à ce problème mais très peu pour la recherche à voisinage variable. J'y ai vu une opportunité de réaliser mon projet de mémoire sur ce sujet et après une discussion avec Gilles Caporossi et Sylvain Perron, ce projet a vu le jour.

# Remerciements

Je tiens tout d'abord à remercier mes deux directeurs de recherche, Gilles Caporossi et Sylvain Perron pour le temps qu'ils m'ont consacré tout au long de la réalisation de ce mémoire ainsi que pour leurs nombreux conseils.

Je remercie également ma famille ainsi que mon épouse pour leur support continu et inconditionnel pendant toute la durée de ce projet.

# Introduction

Les réseaux de neurones, ainsi que d'autres modèles d'apprentissage machine, trouvent de plus en plus d'applications dans le monde des affaires que ce soit par exemple pour la gestion de risques, le marketing ciblé, la prévision de ventes ou encore le contrôle des processus industriels (Mishra et Srivastava, 2014).

Afin de performer, ces modèles doivent être alimentés en un large volume de données puis entraînés ce qui peut représenter un large coût computationel. La valeur ajoutée offerte par l'application des modèles poussent les entreprises à investir dans des ressources informatiques propres ou encore de recourir à des services 'infonuagiques' offrant une capacité de calcul conséquente. Cette capacité de calcul informatique est une ressource de plus en plus importante pour les entreprises c'est pourquoi il est important de vérifier que son utilisation est efficiente afin de limiter les coûts.

Ce mémoire a pour objectif de proposer un nouvel algorithme d'entraînement pour les réseaux de neurones. Cela sera l'occasion de présenter en détails ce processus d'entraînement ainsi que la recherche récente autour de ce sujet.

## Organisation du mémoire

Le premier chapitre de ce mémoire cherchera à présenter le fonctionnement des réseaux de neurones artificiels et comment ces modèles sont ajustés. A partir de cette présentation, le processus d'entraînement des réseaux sera présenté comme un problème

d'optimisation. Une revue de différentes méthodes pour résoudre ce problème seront présentées ainsi que les difficultés inhérentes à l'optimisation de ces réseaux.

Le second chapitre de ce mémoire présentera différentes applications de méthodes de métaheuristiques et notamment l'intérêt récent dans la recherche pour leur application au domaine de l'apprentissage machine. Une revue plus précise de ces applications aux modèles de réseaux de neurones sera ensuite présentée afin de mettre en avant les avantages d'appliquer des métaheuristiques pour l'entraînement de ces modèles. Enfin, une rapide revue de recherches récentes sur la mise en oeuvre de différentes métaheuristiques sur du matériel informatique spécialisé nous assurera que ces méthodes proposées peuvent être compétitives avec l'état de l'art actuel dans le cadre de l'optimisation de réseaux.

Finalement, une présentation rapide de la recherche à voisinage variable sera proposée ainsi que les modifications qui devront être appliquées afin de pouvoir appliquer l'algorithme au problème de l'entraînement de réseaux de neurones. À la suite de cela, l'algorithme proposé par ce mémoire sera présenté et défini explicitement. Enfin, les performances de l'algorithme proposé seront mesurées dans différents contextes et présentées dans le chapitre de résultats.

# Chapitre 1

# Réseaux de neurones

## 1.1 Optimisation d'un réseau

Les réseaux de neurones, également connus sous le nom de réseaux de neurones artificiels (ANN) ou Perceptron Multi-Couches (MLP), sont un modèle d'apprentissage machine dont le nom et la structure sont inspirés par la génétique cérébrale. Ses applications sont extrêmement diverses et présentes dans de nombreuses disciplines (Abiodun, Jantan, Omolara, Dada, Mohamed et Arshad, 2018).

L'objectif de ce modèle est d'approximer une relation non linéaire entre des variables en entrée du modèle avec une ou plusieurs variables en sortie du modèle, tant pour des tâches de classification que de régression. Pour ce faire, le processus d'entraînement du modèle nécessitera d'ajuster de façon itérative les paramètres du modèle jusqu'à l'obtention de résultats satisfaisants.

### 1.1.1 Structure d'un réseau

Ces paramètres sont contenus dans les différentes couches du réseau, chacune de ces couches (Figure 1.1) étant composée de neurones ou *units*.

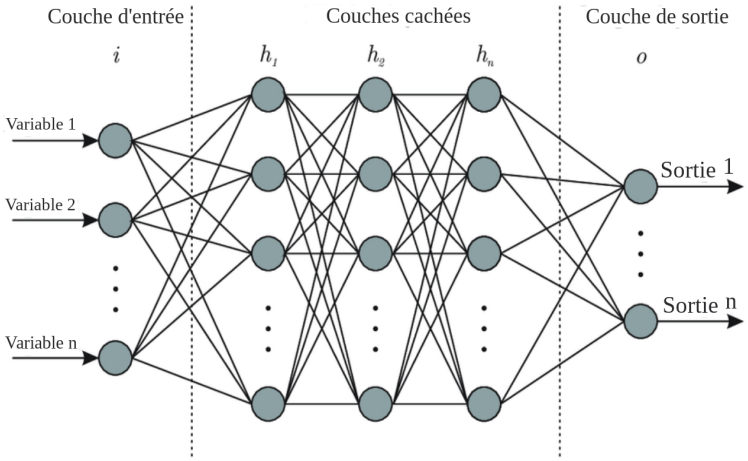



FIGURE 1.1 – Structure d'un réseau de neurones

Chacun de ces neurones sont définis de façon identique et contiennent chacun l'un des paramètres, ou *poids*, du réseau. Chaque neurone (Figure 1.2) prend simplement une moyenne pondérée des valeurs en sortie de la couche qui la précède, lui applique une transformation non-linéaire (fonction d'activation) pour finalement retourner une valeur unique qui sera transmise à la couche suivante.

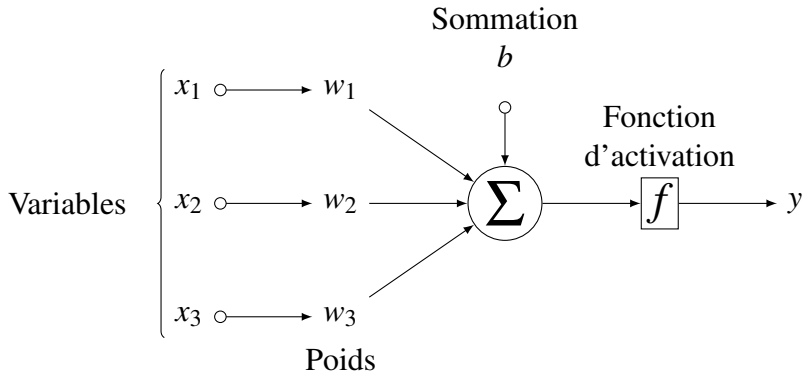



FIGURE 1.2 – Structure d'un neurone

Soit un vecteur de variables en entrée du modèle $x = (x_1, x_2, ..., x_n)$, l'on peut ainsi décrire les opérations effectuées par un neurone :

1. Sommation : $u_k = b_k + \Sigma_{j=1}^{n} w_{kj} x_j$
2. Fonction d'activation : $y_k = \phi(u_k)$

Ou $x_j$ est une observation des données, $w_{kj}$ est le poids correspondant au neurone $k$ et $b_k$ est un terme de biais. Pour la seconde opération, $\phi$ correspond à la fonction d'activation non-linéaire et $y_k$ à la sortie produite par le neurone $k$.

### 1.1.2 Autres composantes d'un réseau

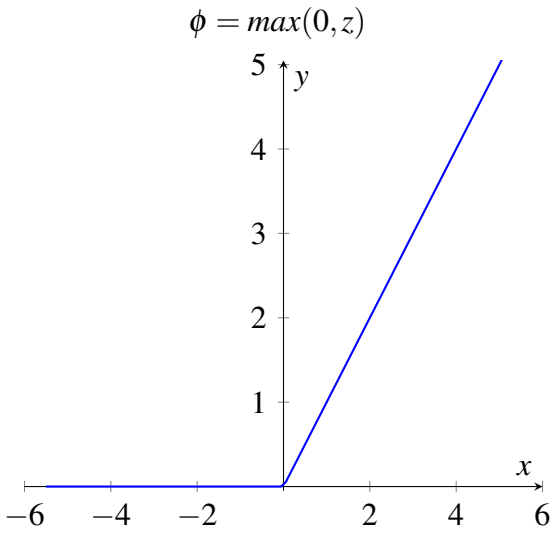
Le choix de la fonction d'activation $\phi$ à un effet important sur la vitesse de convergence de l'entraînement du modèle. Cette dernière a pour objectif de normaliser le résultat de la sommation effectuée par un neurone et devrait être efficiente à calculer.

Certaines propriétés sont désirables dans le choix de cette fonction :

— Non linéaire : L'objectif de cette fonction d'activation est de donner au modèle la capacité de modéliser des relations non-linéaires entre la variable dépendante ($y$) et les variables indépendantes ($x_i$).La fonction choisie devrait donc être non-linéaire.

— Différentiable en tout point : Cette propriété est nécessaire si l'optimisation du réseau repose sur des méthodes qui nécessitent le calcul de gradients.

La fonction *ReLU* sera la fonction d'activation employée dans le reste de ce mémoire. Bien qu'elle ne soit pas différentiable en 0, il est donc nécessaire de poser arbitrairement $f'(0)$, cette dernière présente des caractéristiques désirables et est définie ainsi :



Le choix du nombre de couches et de neurones contenus dans le réseau ainsi que le choix de la fonction d'activation auront une influence directe sur l'ajustement du modèle aux données fournies. Une fois la structure du réseau fixée, l'entraînement d'un réseau revient à trouver la valeur des poids $w_k$ qui vont maximiser la performance du modèle. Que le modèle soit appliqué à une tâche de classification ou de régression une fonction de perte sera définie afin de quantifier cette performance et d'ajuster les paramètres du modèle en conséquence.

Pour les tâches de régression il est possible d'utiliser une fonction classique comme l'erreur moyenne au carré donnée par :

$$C = \sum_{i=1}^{D} (x_i - y_i)^2$$

Un choix de fonction commun pour les tâches de classification est la fonction d'entropie croisée (*Cross Entropy*) :

Pour des tâches de classification binaire, où le nombre de classes à prédire est égal à deux, cette fonction peut être calculée ainsi :

$$C = -(y \log(p) + (1-y) \log(1-p))$$

Dans le cas où le nombre de classes à prédire est supérieur à deux alors il est nécessaire de calculer la fonction de perte pour chacune des classes et pour chaque observation et ensuite de sommer le tout. On obtient alors la fonction :

$$C = -\sum_{c=1}^{M} y_{o,c} \log(p_{o,c})$$

Où :

$M$ = Nombre de classes

$y$ = Indicateur binaire (0 ou 1) si la classe $c$ correspond à une classification correcte pour l'observation $o$

$p$ = Probabilité prédite que l'observation $o$ appartienne à la classe $c$

### 1.1.3 Problème d'optimisation associé

Ainsi, puisque l'entraînement d'un réseau revient à déterminer la valeurs de ses poids afin de minimiser la valeur de cette fonction de perte, il peut être exprimé comme un problème d'optimisation ou cette fonction en devient la fonction objectif. La fonction de perte sera exprimée comme la fonction $C$ dans la notation ci-dessous :

$$\min_{w} C(\hat{A}, w)$$

La minimisation de la fonction sera effectuée en fonction de la valeur des poids du réseau $w$ mais également en fonction des observations de l'ensemble d'entraînement $\hat{A}$.

La difficulté de ce problème sera dépendante du nombre de paramètres dans le réseau et donc du nombre de neurones qui le compose. Pour autant, il a été démontré qu'il suffit d'un réseau composé d'une seule couche cachée contenant trois neurones pour que le problème d'optimisation associé soit NP-Complet (Blum et Rivest, 1992).

Il existe donc un intérêt pratique à limiter le nombre de paramètres du modèle au strict nécessaire bien que cela soit difficile à déterminer a priori. En théorie, un réseau de neurones composé d'une seule couche cachée et d'une fonction d'activation non linéaire est un approximateur universel (Cybenko, 1989), un tel réseau serait donc capable d'approximer n'importe quelle fonction continue présente dans les données d'entraînement. Dans les faits, la qualité des prévisions d'un modèle dépend fortement de ses dimensions et du nombre de neurones qu'il contient (Baldi et Vershynin, 2019). Il est donc important de considérer le compromis entre la complexité de l'entraînement et la capacité d'un modèle.

## 1.2 Méthodes d'entraînement

Puisque la qualité de la solution obtenue par la résolution de ce problème d'optimisation implique directement le score de précision atteint par le modèle dans la tâche qu'il cherche a accomplir, le choix de la méthode de résolution est extrêmement important.

### 1.2.1 Considérations liées au problème

C'est pourquoi la recherche sur le sujet a cherché à considérer des méthodes d'optimisation globales pour résoudre ce problème. Certaines méthodes chercheront à limiter les redondances dans l'espace de recherche afin de pouvoir en considérer la totalité (Chen, 1993) alors que d'autres chercheront à combiner des approches globales et locales pour faciliter la recherche de solutions (Shang et Wah, 1996). En réalité, les gains en capacités de calcul et l'apparition de matériel informatique spécialisé au cours des dernières années ont permis l'ajustement de modèles contenant plusieurs millions de paramètres. Face à des problèmes d'optimisation aussi complexes, les méthodes globales ont été abandonnées au profit de méthodes locales plus efficientes.

L'ajout de nombreux paramètres complexifie la surface de recherche et la nature du problème d'optimisation non convexe, cela crée plusieurs considérations pour les méthodes qui lui seront appliquées. L'optimisation non-convexe implique la présence de multiples optimums et bien que l'objectif d'atteindre un optimum global n'est plus d'actualité dans la littérature, des méthodes locales pourraient rester bloquées dans des optimums locaux au cours de la recherche ce qui limiterait leur convergence et aurait un impact sur la qualité de la solution.

Une autre particularité du problème est que le processus d'optimisation dépend des donnés utilisées, si une solution de qualité est atteinte au cours de l'entraînement du modèle, elle ne le restera pas forcément lorsque le modèle sera appliqué à de nouvelles données pour des fins d'inférence. Ce point est d'autant plus important que ces modèles sont

bien souvent appliqués à des situations qui présentent des données dynamiques par nature. Par exemple, si l'on cherche à modéliser le risque de fraude pour un client de banque la nature temporelle des données implique que le problème d'optimisation évoluera lui aussi dans le temps.

Les différentes méthodes présentées ci-dessous s'inscrivent dans un algorithme plus large qui se nomme 'Rétropropagation des erreurs'. Initialement présenté dans les années 1970 (Werbos, 1974), ce dernier ne sera appliqué aux réseaux de neurones qu'à partir des années 1980 (Goodfellow, Bengio et Courville, 2016). Afin de faciliter la lecture de ce chapitre, un exemple concret de son fonctionnement est proposé en Annexe 2.

### 1.2.2 Descente de gradient

La descente de gradients est l'un des algorithmes les plus populaires pour l'optimisation de réseau de neurones. C'est une méthode qui va dépendre de la première dérivée de la fonction de perte choisie, c'est également une méthode directionelle qui va chercher à déterminer la direction dans laquelle les poids doivent être modifiés afin d'améliorer les performances du modèle :

$$w_{t+1} = w_t - \alpha(\nabla C(\hat{A}, w_t))$$

Les poids sont ainsi mis à jour dans la direction opposée des gradients calculés et le facteur $\alpha$, appelé taux d'apprentissage (ou encore *learning rate*), détermine l'importance de cette mise à jour.

Cependant, cette méthode implique que le calcul des gradients soit effectué sur la totalité de l'ensemble d'entraînement pour chaque mise à jour des poids. Si l'entraînement est effectué sur un très large ensemble de données cette mise à jour peut devenir très coûteuse voir être impossible si la totalité de l'ensemble de données ne peut être stocké en mémoire lors de son calcul.

Pour répondre à ces problèmes, il est possible d'effectuer une mise à jour de nature stochastique soit pour chaque observation $x_i$ et sa variable à prédire $y_i$ contenue dans le jeu de données et sélectionnées de façon aléatoire :

$$w_{t+1} = w_t - \alpha(\nabla C(w_t, x^{[i]}, y^{[i]}))$$

Cette méthode, nommée descente de gradient stochastique (ou *SGD*) évite donc les redondances liées au calcul des mêmes gradients pour chaque nouvelle mise à jour. Néanmoins, chaque poids des données n'est pas représentatif de l'ensemble des donnés ce qui amène beaucoup de variance entre chaque mise à jour des poids.

Un compromis entre ces deux approches, l'approche 'mini-batch', est de considérer non pas une seule observation mais un sous-ensemble du jeu de données pour chaque mise à jour. Si l'ensemble est suffisamment grand, le résultat devrait être représentatif du gradient qui aurait été calculé sur l'ensemble des données et fortement limiter la complexité du calcul tout en réduisant la variance introduite par l'approche stochastique. Soit :

$$w_{t+1} = w_t - \alpha(\nabla C(w_t, x^{[i:i+n]}, y^{[i:i+n]}))$$

Bien que ces modifications améliorent fortement les performances de la descente de gradients dans le cadre de l'entraînement de réseaux de neurones, il reste difficile de minimiser ces fonctions de perte car elles présentent de nombreux minimums locaux sous-optimaux. Cette difficulté provient non-seulement du grand nombre de minimums locaux dans la fonction de perte mais surtout des nombreux points-selle qu'elle comprend (Dauphin, Pascanu, Gülçehre, Cho, Ganguli et Bengio, 2014), soit un point ou une dimension décrit une pente ascendante alors que l'autre décrit une pente descendante.

Cette situation est spécifiquement compliquée pour la méthode de descente de gradient car le point-selle (Exemple donné par la figure 1.3) crée un plateau où le gradient est égal à zéro à partir du duquel il est difficile de déterminer la meilleure direction à

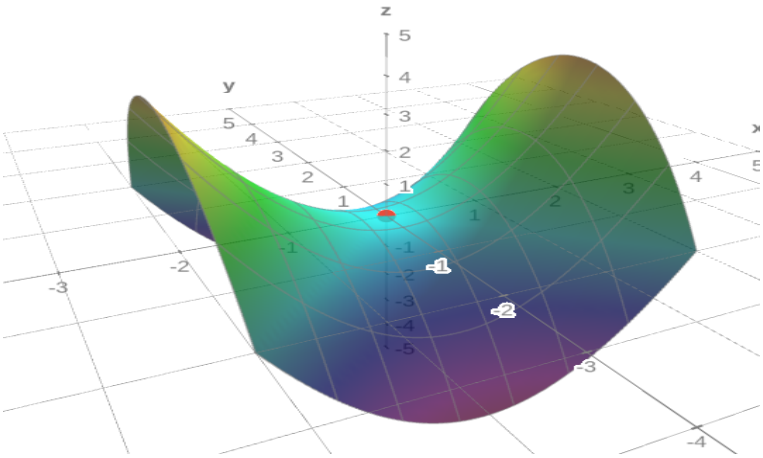



FIGURE 1.3 – Point-selle en $[0,0,0]$ de la fonction $x^2 - y^2$

suivre. Cela mènera finalement à de très petites mise à jour des poids du réseau et l'algorithme considérera avoir convergé.

Il est donc possible de tirer deux difficultés qui persistent pour l'entraînement de réseaux de neurones. L'approche *'mini-batch'* bien qu'elle facilite le calcul des gradient peut toujours introduire un peu de variance et donc réduire la vitesse de convergence. De plus, la descente de gradient est très propice à rester bloquée dans minimas locaux et points-selle. L'ajout d'un terme d'élan (ou Momentum) (Qian, 1999) considère la direction prise par les derniers déplacements pour la nouvelle mise à jour.

$$v_t = w_t - w_{t-1}$$
$$w_{t+1} = w_t + \gamma v_t - \alpha(\nabla C(w_t, x^{[i:i+n]}, y^{[i:i+n]}))$$

Où $v_t$ le précédent déplacement suivit par l'algorithme et $\gamma$ un terme de pondération pour ce précédent mouvement dans la mise à jour des poids. Ainsi, si le processus d'optimisation se retrouve bloqué en un point ou la valeur des gradients est faible, il continuera à suivre la direction suivie à l'itération précédente (Exemple donné par la figure 1.4).

Les méthodes présentées ci-dessous ajoutent néanmoins de nouveaux paramètres aux règles de mises à jour des poids. Notamment $\gamma$ et $\alpha$ qui doivent être déterminés à priori de l'entraînement. Ces derniers sont d'autant plus importants qu'ils vont fortement influencer

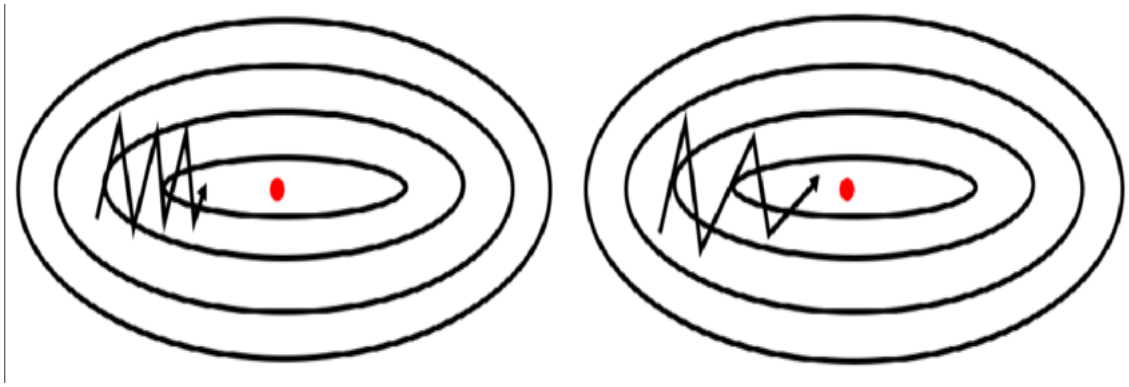


FIGURE 1.4 – SGD sans terme d'élan (Gauche), SGD avec terme d'élan (Droite) tiré de (Du, 2019)

à la fois la vitesse de convergence de l'algorithme mais également la qualité de la solution finalement atteinte.

### 1.2.3 Autres méthodes basées sur les gradients

De nombreuses autres méthodes d'optimisation vont prendre comme base cette règle de mise à jour pour les poids du réseau et lui ajouter des termes afin de répondre à certaines difficultés que peut présenter la descente de gradients classique.

L'algorithme RMSprop est une méthode d'optimisation proposée par Geoff Hinton mais qui n'a pas fait l'objet d'une publication scientifique. La motivation derrière cet algorithme est que l'ordre de grandeur des gradients calculés peut varier en fonction des poids du réseau et peut donc évoluer au cours de l'entraînement du modèle. Cet algorithme répond donc à ce problème en conservant une moyenne mobile des précédents gradients et la mise à jour des poids se fera en fonction de cet ordre de grandeur :

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma)[g^2]_k$$
$$w_{t+1} = w_t - \frac{\alpha}{\sqrt{E[g^2]_t + \varepsilon}}$$

Où $g$ correspond aux gradients, soit $\nabla C(w_k, x^{[i]}, y^{[i]})$ dans les équations précédentes.

L'algorithme ADAM (*ADAptive Moment*) (Kingma et Ba, 2017) est une autre méthode d'optimisation qui s'inspire des méthodes présentées ci-dessus. L'algorithme va chercher à combiner les avantages de RMSprop, qui considère l'ampleur des gradients, et de l'ajout du terme d'élan avec la descente de gradients classique, qui considère un lissage des gradients dans le calcul de la mise à jour. L'on retrouve ainsi le même terme que celui utilisé par RMSprop ici nommé $v_t$ :

$$v_t = \beta_1 v_{t-1} + (1-\beta_1)[g^2]_t$$

Ainsi qu'un terme $m_t$ qui ressemble fortement à un terme d'élan présenté plus tot :

$$m_t = \beta_2 m_{t-1} + (1-\beta_2)[g]_t$$

La mise à jour ressemblera alors fortement à la mise à jour effectuée par RMSprop avec l'ajout d'un terme d'élan $m^{[k]}$ :

$$w_{t+1} = w_t - \frac{\alpha}{\sqrt{v_t+\varepsilon}} m_t$$

De nombreuses autres méthodes locales pour l'entrainement de réseaux de neurones existent outre ces trois méthodes. Pour une revue plus exhaustive de différentes méthodes le lecteur est invité à se référer à (Shrestha et Mahmood, 2019). Pour autant ces trois méthodes présentées plus tôt seront utilisées comme base de comparaison pour l'algorithme proposé dans la partie méthodologie de ce mémoire.

## 1.3 Méthodes de recherche alternatives

### 1.3.1 Matrice Hessienne

La majorité des méthodes d'optimisation pour l'entraînement de réseaux de neurones, bien qu'elles proposent différentes variations, reposent donc sur la descente de gradients. Cependant, bien d'autres méthodes d'optimisation ont été considérées pour résoudre ce problème et permettent de mettre en lumière de nouvelles difficultés liées à l'entraînement d'un tel modèle.

En plus de méthodes utilisant le gradient pour trouver l'optimum d'une fonction, certaines méthodes vont reposer sur le calcul de la seconde dérivée de la fonction à optimiser ce qui revient à obtenir la matrice Hessienne, $H$, de cette fonction.

$$H(C) = \begin{bmatrix} \frac{\partial^2 C_1}{\partial x_1^2} & \frac{\partial^2 C_1}{\partial x_1 x_2} & .. & .. & \frac{\partial^2 C_1}{\partial x_1 x_n} \\ \frac{\partial^2 C_2}{\partial x_2 x_1} & \frac{\partial^2 C_2}{\partial x_2^2} & .. & .. & \frac{\partial^2 C_2}{\partial x_2 x_n} \\ .. & .. & .. & .. & .. \\ .. & .. & .. & .. & .. \\ \frac{\partial^2 C_n}{\partial x_n x_1} & \frac{\partial^2 C_n}{\partial x_n x_2} & .. & .. & \frac{\partial^2 C_n}{\partial x_n^2} \end{bmatrix}$$

Le calcul de cette matrice implique donc le calcul de $n \times n$ dérivées secondes soit une complexité de $O(n^2)$. Cela est donc très peu approprié à l'entraînement de réseaux de neurones dont le nombre de poids peut dépasser plusieurs millions.

Néanmoins la mise à jour des poids serait effectuée ainsi :

$$w_{t+1} = w_t - H^{-1}(\nabla C(w_k, x^{[i]}, y^{[i]}))$$

Il est à noter que certaines méthodes ont été développées, comme l'algorithme de Broyden-Fletcher-Goldfarb-Shanno (*BFGS*) (Goodfellow, Bengio et Courville, 2016), afin de simplifier ces calculs en utilisant une approximation de la matrice Hessienne $H$. Cependant, leur application à l'entraînement de réseau de neurones se heurte à un autre problème qui sera présenté dans la sous-section suivante.

### 1.3.2 Particularités de l'entraînement de réseaux de neurones

Il est très commun pour les réseaux de neurones que la matrice Hessienne $H$ soit mal conditionnée, ou encore que son obtention est numériquement instable (Saarinen, Bramley et Cybenko, 1993). Cela implique qu'un très petit changement dans les données en entrée peut impliquer en un très grand changement dans les résultats obtenus par le modèle. Cette propriété implique que le processus d'optimisation n'est pas forcément

stable. La méthode de Newton est généralement une bonne méthode pour minimiser des fonctions convexes avec des matrices Hessiennes mal conditionnées mais cette méthode s'applique mal aux réseaux de neurones (Goodfellow, Bengio et Courville, 2016).

De plus, de nombreuses recherches ont pu déterminer qu'en pratique les réseaux de neurones ne terminaient pas forcement leur entraînement sur un point critique, que ce soit un minimum ou un point de selle, et que ces points critiques n'existent pas nécessairement. En effet, l'optimisation ne s'arrête même pas forcément dans une région ou les gradients sont faibles (Goodfellow, Bengio et Courville, 2016). Il existe donc peu d'intérêt à essayer d'identifier ces points critiques à l'aide de méthodes comme la méthode de Newton.

Pour ces raisons, la descente de gradients et ses nombreuses variations se sont imposées comme la méthode de choix pour l'entraînement de réseau de neurones.

De plus, bien que les propriétés du problème d'optimisation d'un réseau de neurones semblent fortement limiter les méthodes qui peuvent lui être appliquées, la recherche sur le sujet ne se limite pas à définir de nouvelles méthodes de résolution mais également de nouvelles façons de les employer.

### 1.3.3 Structure du réseau et optimisation

Les règles de mise à jour des poids présentées à la section précédente impliquent que l'on considère le réseau dans son entièreté lorsque ce dernier est mis à jour. L'approche *Forward Thinking* (Réflexion Prospective) (Hettinger, Christensen, Ehlert, Humpherys, Jarvis et Wade, 2017) propose la construction et l'entraînement de réseaux une couche à la fois et présente de meilleures résultats que lorsque que le réseau est considéré dans sa totalité. Cela implique qu'un réseau sera initialement formé d'une seule couche cachée, que cette dernière sera entraînée jusqu'à l'atteinte d'un certain critère d'arrêt. À partir de ce point une couche cachée supplémentaire sera ajoutée au réseau et à nouveau entraînée alors que les paramètres de la première couche seront eux fixés. Si l'on représente le

processus d'entraînement de cette méthode pour les trois premières couches :

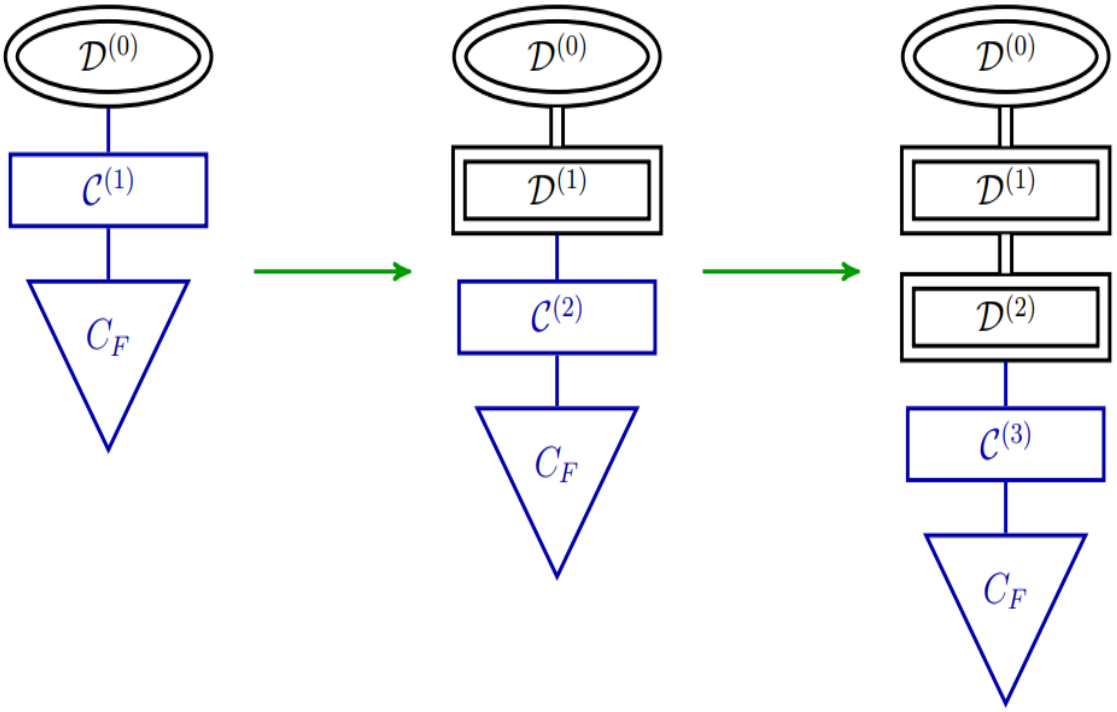



FIGURE 1.5 – Représentation de trois itérations d'un entraînement couche par couche, tiré de (Hettinger, Christensen, Ehlert, Humpherys, Jarvis et Wade, 2017)

Le jeu de données initial est représenté par l'ellipse $D^{(0)}$, la première couche cachée contenant des paramètres est $C^{(1)}$, $C^{(F)}$ correspond à la couche de sortie du réseau. Les éléments en bleu étant les éléments modifiables par l'entraînement du réseau alors que les éléments présentant une double bordure sont fixés. Au passage à l'itération suivante, symbolisée par la flèche verte, la couche de paramètres précédente est fixée et une nouvelle couche $C^{(2)}$ est ajoutée. Cela signifie que plutôt que de définir un problème d'optimisation pour la totalité du réseau, l'entraînement du modèle est subdivisé en multitude de problèmes qui seront résolus de façon successive.

Soit le problème de minimisation de la fonction d'erreur du réseau présentée plus tôt :

$$\min_{w} C(\hat{A}, w)$$

Si l'on exprime le problème d'optimisation correspondant à la seconde itération de la méthode ci-dessus, on obtient :

$$\min_{w_{C2}} C(D^{(1)}, w_{C2})$$

Le nombre de paramètres à considérer pour cette minimisation est donc extrêmement réduit puisque seul les poids contenus dans une seule couche du réseau sont considérés. Le problème sera décomposé en autant de problèmes d'optimisation que de couches cachées contenues dans le réseau. Il est donc possible de prendre en considération la structure du réseau dans le processus d'optimisation. Cet argument sera considéré plus en détails dans le chapitre de méthodologie de ce mémoire.

# Chapitre 2

# Métaheuristiques

## 2.1 Métaheuristiques et apprentissage machine

Un grand nombre de problèmes d'optimisation appliqués en science, ingénierie, économie et dans le monde des affaires en général sont complexes et difficiles à résoudre. Depuis de nombreuses années, les métaheuristiques sont devenues des outils importants pour la résolution de ces problèmes d'optimisation complexes (Sorensen, Sevaux et Glover, 2017).

Le terme de métaheuristique a été proposé pour la première fois par Fred Glover (Beyer et Schwefel, 2002). Une métaheuristique est un processus itératif qui guide et modifie les opérations d'une heuristique dans le but de produire des solutions de façon efficiente. Il est possible de citer quelques métaheuristiques considérées *classiques* comme le recuit simulé, la recherche par tabou, les algorithmes évolutionaires comme les algorithmes génétiques ou encore la recherche à voisinages variables.

Outre ces algorithmes très connus, il existe une grande variété de métaheuristiques et de nouveaux algorithmes sont régulièrement développés. (Faramarzi, Heidarinejad, Mirjalili et Gandomi, 2020). Pour autant, la recherche sur le sujet ne se limite pas au développement de nouvelles méthodes. Parmis ces sujets l'on retrouve la comparaison des

performances de différentes méthodes, le développement de méthodes coopératives qui combinent différentes approches, la parallélisation et la distribution des algorithmes ainsi que des applications à des problèmes réels (Hussain, Salleh, Cheng et Shi, 2019).

Comme pour les réseaux de neurones, présentés au chapitre précédent, l'entraînement de la majorité des modèles d'apprentissage machine revient à résoudre un problème d'optimisation dans le but d'ajuster les paramètres du modèle. Au cours des dernières années, les métaheuristiques ont été de plus en plus appliquées à différents problèmes en apprentissage machine, dont l'entraînement de modèles, cette section cherchera à présenter ces récentes recherches.

Les meilleurs résultats obtenus pour de nombreux problèmes d'optimisation en science et dans diverses industries sont obtenus par des algorithmes d'optimisation hybrides. La combinaisons d'outils d'optimisation tels que les métaheuristiques, la programmation mathématique, la programmation par contraintes et l'apprentissage automatique ont fourni des algorithmes d'optimisation très efficaces (Talbi, 2013).

Ce précédent article présente une taxonomie afin de catégoriser différents algorithmes combinant des métaheuristiques, méthodes exactes, programmation par contraintes et apprentissage machine. La particularité de l'article est qu'il propose une classification en deux niveaux. L'auteur distingue de cette façon les 'hybridations à bas niveau' pour lesquelles une fonction interne de la métaheuristiques est remplacée par une autre méthode d'optimisation. Et les 'hybridations à haut niveau' ou les techniques d'optimisation sont complètement autonomes. Le second critère de classification proposé par l'auteur correspond à l'interaction entre ces deux composantes. Dans une approche 'Relais', les composantes de la méthode hybride sont appliquées les unes après les autres chacune utilisant le résultat de la précédente. A l'inverse, l'approche 'Coopérative' implique que les composantes de la méthode évoluent en parallèle et puissent communiquer de façon non-séquentielle. La figure 2.1 représente l'organisation hiérarchique de la classification proposée par l'auteur.

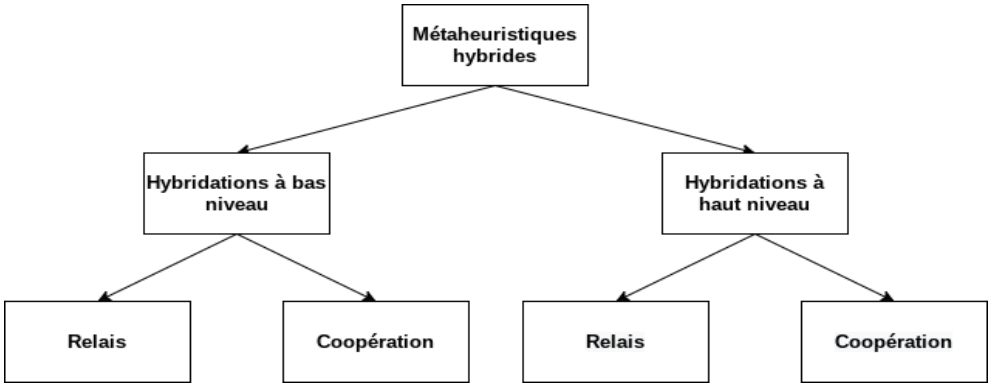



FIGURE 2.1 – Représentation graphique de la classification proposée par (Talbi, 2013)

L'algorithme proposé dans ce mémoire s'inspirera de ces 'hybridations à bas niveau' en intégrant l'une des recherche locales propres à l'entraînement des réseaux de neurones, présentées dans le précédent chapitre, à la métaheuristique de la recherche à voisinage variable. De plus la méthode proposée suivra une approche de relais qui sera décrite dans plus de détails dans le chapitre dédié à la méthodologie.

## 2.2 Métaheuristiques et réseaux de neurones

Tel que présenté dans le premier chapitre de ce mémoire, l'entraînement de réseaux de neurones correspond à la résolution d'un problème d'optimisation complexe. La particularité des métaheuristiques est qu'elles ne garantissent pas l'obtention de solutions optimales, mais permettent d'atteindre des solutions de qualité en un temps raisonnable. Tel que discuté au cours des sections précédentes, il est de toute façon impossible d'espérer atteindre des solutions optimales au cours de l'entraînement de réseaux de neurones.

L'application de métaheuristiques au problème de l'entraînement de réseaux de neurones n'a rien de récent (Koehn, 1994) mais gagne en intérêt depuis quelques années. Un point critique qui justifie cet intérêt est que pour d'autres domaines, l'emploi de métaheuristiques était justifié par la difficulté croissante des problèmes d'optimisation rencontrés

et c'est aujourd'hui également le cas pour les réseaux de neurones. En effet, la taille moyenne des réseaux de neurones continue de croître avec le temps et suit la croissance de la capacité de calcul informatique disponible, néanmoins les méthodes employées pour l'ajustement de ces modèles restent exclusivement des méthodes locales alors que leur entraînement est de plus en plus complexe. (Zhang et Qu, 2021).

La classification d'image est un problème de référence dans la littérature et est généralement utilisé afin de comparer les performances de nouvelles méthodes face à l'état de l'art actuel. Différents algorithmes trouvent beaucoup de succès dans ces applications. (Rere, Fanany et Arymurthy, 2016) présente une application des métaheuristiques du recuit simulé, de l'évolution différentielle et de la recherche harmonique. Bien que ces algorithmes accroissent le temps de calcul, le gain de précision face au problème de classification auxquels ils sont appliqués est très significatif soit jusqu'à 7.14% de gain de précision.

Les avantages de l'application de métaheuristique pour le processus d'entraînement de réseaux de neurones ne se limitent cependant pas à un gain de précision au détriment du temps de calcul nécessaire. Dans le cadre d'une modélisation des effets environnementaux d'un avion d'affaires, les auteurs ont ajusté un réseau de neurones à l'aide d'une méthode hybride qui intègre un algorithme génétique (Baklacioglu et collab., 2018). Les résultats de cette étude indique également un gain de précision du modèle mais aussi que cette méthode est plus efficiente et qu'elle favorise la capacité de généralisation du modèle. En effet, dans l'article (Tran-Ngoc et collab., 2021), les auteurs proposent également une méthode hybride basée sur les algorithmes génétiques pour la détection de dégats dans des structures composites mais dont l'objectif est de réduire significativement le coût computationel de l'entraînement du modèle.

Ainsi, bien que ces méthodes soient à la fois très efficaces tout en restant relativement flexibles dans le but qu'elles remplissent, elles présentent toujours une limite. La majorité

des applications dans lesquelles sont proposées l'ajout de métaheuristiques restent très dépendantes du contexte dans lequel ces méthodes sont appliquées. Par exemple, dans de nombreuses applications les données d'entraînement vont présenter une dimension temporelle comme c'est le cas pour des données financières (Gocken, Özçalici, Boru İpek et Dosdoğru, 2015). Dans cet article, les auteurs exploitent la recherche harmonique ainsi les algorithmes génétiques pour l'ajustement de réseaux de neurones.

Dans les faits, de nombreuses applications vont présenter des données qui incorporent une certaine temporalité. Par exemple, la prévision de l'efficacité énergétique d'une centrale d'énergie solaire est extrêmement importante afin de faire une utilisation efficiente de sa capacité de production. Les données dans ce contexte seront donc des mesures régulières de la production de la centrale en fonction de certaines variables. Or, ces séries chronologiques sont connues pour être non linéaires et non stationaires ce qui aura une influence le processus d'optimisation du modèle de prévision (Abedinia, Amjady et Ghadimi, 2018). Dans ce précédent article, la méthode d'apprentissage hybride, qui s'inspire d'algorithmes évolutionaires, est justifiée par la nature des données qui influence l'optimisation du réseaux de neurones. Cela met donc en évidence le fait qu'il est important de conceptualiser l'optimisation d'un réseau en fonction du problème auquel il est appliqué.

## 2.3 Métaheuristiques et GPUs

La considération d'heuristiques pour l'entraînement de réseaux de neurones fait l'objet de plus en plus de publications. L'algorithme *ADAM* présenté au cours de la section 1.3 est aujourd'hui encore considéré comme l'état de l'art pour l'entraînement de réseaux de neurones bien que sa publication remonte déjà à 2015.

Un argument important pour la difficulté de remplacer ces méthodes locales qui reposent sur des descentes de gradients comme *ADAM* est qu'elle sont capables de facile-

ment profiter de matériel informatique spécialisé comme les GPUs (*Unité de traitement graphique*). Par exemple, les gains de performance induits par l'utilisation de GPUs dans le cadre de l'entraînement de réseaux de neurones avec une descente de gradients stochastique est de 7x plus rapide qu'avec uniquement des CPUs (Ma, Rusu et Torres, 2019). Ces gains de performance sont principalement justifiés par la grande capacité de parallélisation des GPUs (Kaleem, Pai et Pingali, 2015).

Au cours des dernières années, une partie de la recherche s'est consacrée à l'implémentation de métaheuristiques sur des GPUs afin de répondre à divers problèmes d'optimisation de façon plus efficiente, dont l'entraînement de réseaux de neurones (Van Luong, 2011). L'intérêt pour ces recherches est extrêmement significatif, l'implémentation sur GPUs est jusqu'à 80 fois plus rapide pour des problèmes d'optimisation combinatoires et jusqu'à 240 fois plus rapide pour des problèmes d'optimisation continus (Luong, Melab et Talbi, 2013).

Ces recherches ont notamment mené au développement de différentes librairies et ressources de développement afin de faciliter l'application de ces méthodes comme *DNF* (Muñoz-Ordóñez, Cobos, Mendoza, Herrera-Viedma, Herrera et Tabik, 2018) dont les applications se concentrent autour de l'entraînement de réseaux de neurones et qui repose la librairie d'apprentissage machine *TensorFlow* (Abadi et collab., 2015). LibCudaOptimize à l'inverse cherche à fournir un environnement de développement plus général pour des applications en optimisation continue mais se concentre principalement sur des méthodes d'optimisation qui reposent sur des algorithmes évolutionnaires comme l'optimisation par essaims (Nashed, Ugolotti, Mesejo et Cagnoni, 2012).

Toutes les métaheuristiques ne se prêtent pas pour autant facilement à cette implémentation. Les métaheuristiques impliquant des opérations simples sur des vecteurs de solutions s'y prêtent particulièrement bien comme les algorithmes génétiques ou la recherche par tabou qui a été appliquée avec succès à des problèmes de transport tout en profitant d'une implémentation sur GPUs (Pandi, Ho, Nagavarapu, Tripathy et Dauwels, 2018).

Ce mémoire se concentre sur l'application de la métaheuristique de la recherche à voisinage variable pour l'entraînement de réseaux de neurones. La considération de voisinages dans la recherche de solutions complique cependant fortement son implémentation sur des GPUs. Une méthode proposée dans la recherche afin de contourner ce problème est de considérer une implémentation hybride, les calculs nécessaires au traitement des voisinages et difficiles à paralléliser seront traités par un CPU alors que le reste des calculs sera parallélisé sur des GPUs. Cette méthode hybride a notamment permis l'application de la recherche à voisinage variable à des problèmes d'optimisation d'inventaire et présente des solutions de meilleure qualité qu'une implémentation plus classique de la métaheuristique (Antoniadis et Sifaleras, 2017).

Une méthode similaire appliquée à un problème de routage de véhicules avec livraisons et ramassages sélectifs. La parallélisation de ce problème sur GPUs permet finalement d'obtenir de meilleures solutions dans 51 sur 68 expériences réalisées et l'obtention de ces résultats a été en moyenne environ 15 fois plus rapide (Coelho et collab., 2016).

L'algorithme proposé dans ce mémoire ne suivra néanmoins pas cette méthode hybride, en reformulant l'algorithme spécifiquement pour l'entraînement de réseau de neurones et en profitant de librairies informatiques et autres ressources récentes. La méthode proposée reposera quasiment exclusivement sur l'utilisation de GPUs. Ces considérations seront présentées en détails dans le chapitre 4 de ce mémoire.

# Chapitre 3

# Méthodologie

## 3.1 Recherche à voisinage variable

### 3.1.1 Définition générale

La recherche à voisinage variable (Mladenović et Hansen, 1997) (*RVV*) est une métaheuristique dont la méthodologie repose sur l'exploration successive de voisinages de solutions pour un problème d'optimisation donné. La RVV a été proposée principalement pour la résolution de problèmes d'optimisation combinatoires, par exemple l'un des problèmes caractéristique de ce domaine est le problème du voyageur de commerce auquel la RVV a été appliquée avec succès (Meng, Li, Dai et Dou, 2018). Cette métaheuristique a su trouver un grand nombre d'applications dans des domaines variés : théorie de la localisation, conception de réseaux, intelligence artificielle, etc.

L'un des grands avantages de cette méthode est qu'elle est relativement facile à mettre en oeuvre sans pour autant compromettre ses performances. Elle permet généralement d'obtenir de bonnes solutions en un temps raisonnable notamment grâce à la facilité avec laquelle elle peut être parallélisée, même face à des problèmes considérés NP-Complet (Herrán, Colmenar, Martí et Duarte, 2020).

La RVV peut être définie comme la combinaison de trois étapes qui seront répétées jusqu'à atteindre un critère d'arrêt définit par avance (Hansen, Mladenovic, Todosijević et Hanafi, 2016) :

— Perturbation de la solution

— Procédure d'amélioration

— Changement de voisinage

Pour la majorité de ses applications, une structure de voisinages sera définie à l'initialisation de la RVV ce qui signifie que la notion de voisinage sera définie explicitement dans le contexte de l'application et qu'une liste de voisinage pourra être formée afin qu'ils soient explorés. Soit $S$ une solution quelconque pour un problème d'optimisation donné, $k$ l'indice du voisinage courant et $N_k(S)$ le voisinage $k$ de cette solution.

La procédure de perturbation a pour objectif de sortir la procédure de recherche d'un minimum local dans lequel elle pourrait être bloquée. Une façon très simple de mettre en oeuvre cette procédure est de sélectionner aléatoirement une solution contenue dans le voisinage $N_k(S)$ pour un indice $k$ donné.

La procédure d'amélioration correspond à une recherche locale au sein d'un voisinage $N_k(S)$ , si une solution atteinte de cette façon est meilleure que la solution $S$ actuelle alors cette dernière est remplacée par la nouvelle solution $S$. Il est possible d'envisager plusieurs critères d'arrêt pour cette recherche, les plus communs sont les critères de la première amélioration ou de la meilleure amélioration. Dans le cas du critère de la première amélioration, l'exploration du voisinage $k$ s'arrêtera dès qu'une meilleure solution que la solution actuelle $S$ sera atteinte. Le critère de la meilleure amélioration impliquera par contre une recherche exhaustive du voisinage considéré, la solution $S$ sera remplacée par la meilleure solution du voisinage si cette dernière l'améliore.

Le changement de voisinage correspond à la méthode de diversification utilisée par la RVV. L'objectif est de guider la méthode de recherche à travers l'espace de solutions. Ce changement de voisinage revient donc à transformer le voisinage $N_k(S)$ afin de créer un nouvel ensemble de solutions qui seront à nouveau explorées par la procédure d'amélioration.

### 3.1.2 Descente à voisinage variable

Une variation de la RVV nommée Descente à Voisinage Variable (DVV) exploite le principe qu'une solution $S$ qui est un optimum local selon plusieurs voisinages est plus probable d'être un minimum global qu'une solution qui n'est le minimum local que d'un seul voisinage. Sur la base de cet argument, la DVV va explorer une suite de voisinages de façon séquentielle dans le but d'améliorer une solution. Soit la structure de voisinage définie a priori $N_k(S)$ ou $k$ correspond au nombre de voisinages définis, tel que présenté dans la section précédente.

Chacun de ces voisinages $k$ sera exploré à l'aide d'une recherche locale, il sera donc possible d'identifier un optimum local pour chacun de ces voisinages.

**Algorithme 1** Descente à voisinage variable

**Entree**
$k_{max}$ Nombre de voisinages définis
$S$ Solution initiale
$N_k(S)$ Voisinage initial
**Sortie**
$\hat{S}'$ Nouvelle solution
$k \leftarrow 1$
**Tant que** $k < k_{max}$ **faire**
$S' \leftarrow Recherche\_Locale(S, N_k(S))$
**Si** $S'$ *meilleure que* $S$ **Alors**
$S \leftarrow S'$
$k \leftarrow 1$
**Sinon**
$k \leftarrow k+1$
**Fin Si**
**Fin Tant que**

### 3.1.3 Recherche à voisinages fixes

Une autre variation de la RVV est la Recherche à Voisinages Fixes (RVF), aussi connue comme Recherche Locale Itérative (Lourenço, Martin et Stützle, 2003). Le fonctionnement de cette méthode repose sur l'obtention d'une solution voisine $S'$ de la solution actuelle $S$ à l'aide d'une perturbation, comme présenté dans la section 3.1.1 de ce chapitre. Ensuite, et comme pour la DVV, une recherche locale est effectuée à partir de cette solution $S'$. Les étapes de perturbation et de recherche sont ensuite répétées successivement jusqu'à l'atteinte d'un critère d'arrêt.

La particularité de cette variation est qu'aucune structure de voisinage explicite n'est définie à l'initialisation de l'algorithme. Cette méthode nécessite donc un autre critère d'arrêt que la DVV qui explorait un certain nombre de voisinages avant de sélectionner une solution. Un critère d'arrêt généralement utilisé pour la RVF est le temps de calcul (Hansen, Mladenovic, Todosijević et Hanafi, 2016).

L'algorithme 2 qui suit reviendra donc simplement à effectuer de façon séquentielle

des recherches locales sur la solution initiale jusqu'à l'atteinte d'un critère d'arrêt, ici un temps de calcul maximum.

**Algorithme 2** Recherche à voisinages fixes

**Entree**
  $t_{max}$ Temps de calcul maximum
  $S$ Solution initiale
**Sortie**
  $\hat{S}'$ Nouvelle solution
$k \leftarrow 0$
**Tant que** $t < t_{max}$ **faire**
  $S' \leftarrow Recherche\_Locale(S)$
  **Si** $S'$ *meilleure que* $S$ **Alors**
    $S \leftarrow S'$
    $t \leftarrow t + TempsCalcul()$
  **Fin Si**
**Fin Tant que**

Une définition générale de l'algorithme de la RVV, indépendamment de l'application à problème particulier, peut donc être présentée plus explicitement de cette façon :

L'algorithme 3 qui suit, reprend le fonctionnement de la RVV mais permet le changement de voisinage d'une solution lorsque la recherche locale ne permet d'améliorer la solution courante.

**Algorithme 3** Recherche à Voisinages Variables Générale

**Entree**
  $k_{max}$ Nombre de voisinages définis
  $S$ Solution initiale
  $N_k(S)$ Voisinage initial
**Sortie**
  $\hat{S}'$ Nouvelle solution
$k \leftarrow 1$
**Tant que** Critère d'arrêt **faire**
  $S' \leftarrow Recherche\_Locale(S, N_k(S))$
  **Si** $S'$ *meilleure que* $S$ **Alors**
    $S \leftarrow S'$
    $k \leftarrow 1$
  **Sinon**
    $k \leftarrow k+1$
  **Fin Si**
**Fin Tant que**

De nombreuses autres variations de la RVV pourraient être présentées et le lecteur est invité à se référer à (Hansen, Mladenovic, Todosijević et Hanafi, 2016) pour plus de détails. Cependant, les algorithmes 1, 2 et 3 présentés ci-dessus sont ceux ayant principalement inspiré le prochain chapitre de méthodologie de ce mémoire.

## 3.2 Présentation de l'algorithme proposé

L'algorithme proposé pour la réalisation de ce mémoire cherchera donc à adapter la RVV au problème d'optimisation que représente l'entraînement de réseaux de neurones et s'inspirera de la version générale de la RVV.

### 3.2.1 Considérations liées aux réseaux de neurones

Le premier point à considérer est la nature de la structure de voisinage qui sera employée pour l'application de la RVV à un problème particulier. Comme présenté au cours de la section 3.1, le voisinage d'une solution $S$ correspond à un ensemble de solutions qui peut être obtenu grâce à une transformation $t$. Dans le cas de son application à l'entraînement de réseaux de neurones, une solution de ce problème correspondra à un vecteur contenant les différents poids du réseau :

$$w = [0.74, 0.23, -0.61, ..., 0.97, -0.80, -0.12]$$

Néanmoins, ces solutions présentent une structure qui est induite par la nature du problème. En effet, chaque paramètre de cette solution correspond à une connection au sein du réseau (Cf. Chapitre 1) dont la position est relative aux autres. Certains neurones seront donc directement connectés à d'autres, certains neurones feront partie de la même couche cachée que d'autres, etc. L'on peut donc employer cette structure afin de définir une structure de voisinage pertinente pour l'application à ce problème.

Bien que peu de recherche ait été effectuée sur l'application de la RVV à l'entraînement de réseau de neurones, il est possible de s'inspirer de l'application d'autres méthodes de résolution à ce problème pour définir une structure de voisinage pertinente. L'entraînement de réseaux de neurones en considérant l'architecture même du réseau (C'est a dire les différentes connections qui le compose) a déjà été considérée par de précédentes recherches (El-Fallahi, Martí et Lasdon, 2006). La proposition faite par cet article est qu'il est possible de trier les neurones du réseau en fonction de leur contribution à la fonction de perte globale du réseau, sur la base du calcul de leur dérivée partielle. A partir de ce point, il est possible de considérer que la génération d'un nouveau voisinage $k$ se fera en modifiant les $k$-premiers neurones de cette liste (Alba et Marti, 2006). La figure 3.1 ci-dessous représente cette approche, l'on sélectionne un sous-ensemble des neurones du

réseau (ici les points noircis) qui seront les seuls paramètres modifies afin de générer une solution voisine.

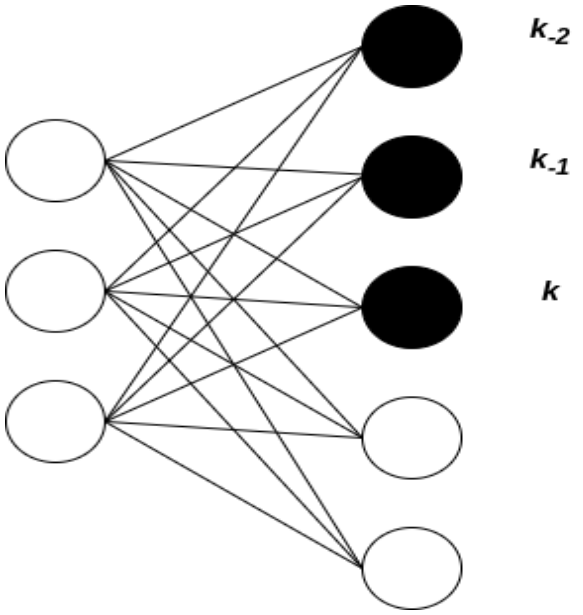



FIGURE 3.1 – Exemple de la structure de voisinage proposée par (Alba et Marti, 2006).

Une approche simplifiée de cette structure de voisinage pourrait également être de sélectionner aléatoirement l'un des neurones du réseau et modifier tous les poids du réseau sauf pour les connections en provenance et sortantes du neurone sélectionné. Un grand nombre de méthodes pour structurer le voisinage des solutions tout en considérant la structure du réseau pourraient donc être proposées. Il reste important de continuer à considérer la réalité de la mise en oeuvre de réseaux de neurones pour la définition de cette structure. Aujourd'hui, même un réseau très simple peut compter plusieurs millions de paramètres. La première méthode proposée dans le paragraphe précédent deviendrait bien trop coûteuse pour des modèles aussi importants qui impliqueraient de stocker à chaque changement de voisinage toutes les contributions de chaque neurones mais également de trier cette liste.

L'on peut également noter que cela impliquerait autant de voisinages, soit $k$, que de paramètres dans le réseau. Pour la méthode simplifiée, présentée au cours du paragraphe précèdent, cette dernière impliquerait trop peu de similitudes entre deux solutions voisines.

A mesure qu'un modèle devient de plus en plus profond (nombre de couches cachées), le nombre de ses paramètres va croître sans que le nombre de connections à un neurone donné n'augmente.

Sur la base de ces observations et de l'approche Forward Thinking (Hettinger et collab., 2017) présentée dans la section 1.3, qui propose l'entraînement de réseaux de neurones une couche à la fois. La structure de voisinage utilisée dans ce mémoire sera de considérer que deux solutions sont voisines si leurs seules différences sont contenues dans une couche du réseau. Soit, pour générer un nouveau voisinage de solutions l'on modifiera les poids associés aux connections en rouge dans ce diagramme :

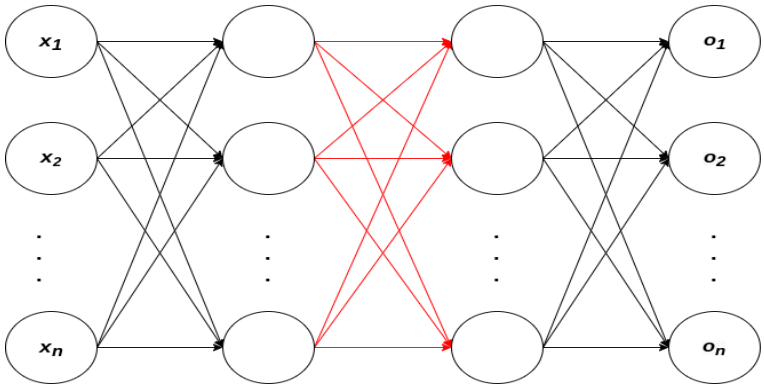



FIGURE 3.2 – Représentation de la structure de voisinage proposée

Pour cette figure 3.2, l'on sélectionne encore un sous-ensemble des neurones du réseau (Selon une méthode différente de celle présentée à la figure 3.1) afin de ne modifier que ces paramètres pour générer une nouvelle solution.

Tout en essayant de respecter la structure de l'algorithme original présenté dans la section 3.1, il est nécessaire d'anticiper quelques difficultés liées à son application à l'entraînement de réseau de neurones, autre que la définition de la structure de voisinage.

- La RVV est une métaheuristique initialement pensée pour être appliquée a des problèmes d'optimisation combinatoire pour lesquels le nombre de solutions est fini (Mladenović et Hansen, 1997). Dans le contexte des réseaux de neurones, une solution corres-

pond à un vecteur de nombre réels (vecteur des poids du réseau), un nombre quasi-infini de solutions sont donc admissibles. L'on considère déjà comme structure de voisinage que deux solutions sont voisines si les seules différences qu'elles contiennent sont comprises dans une seule couche du réseau. Plutôt que de générer toutes les solutions qui serviront aux changements de voisinage à l'initialisation de l'algorithme, une solution voisine à la solution actuelle sera générée lorsque cela est nécessaire dans l'algorithme. Le choix de la couche cachée concernée sera alors aléatoire. Ainsi, plutôt que de définir une structure de voisinage fixe, elle devra être définie de façon paramétrique, sur la base d'une distance par rapport à l'ancienne solution.

- Le critère d'arrêt pour la recherche locale et le passage au prochain voisinage défini dans l'algorithme original implique d'avoir atteint, soit la meilleure solution de ce voisinage soit la première solution améliorante. Dans le cas de la meilleure amélioration, c'est quelque chose que l'on peut difficilement espérer face à la complexité du problème à résoudre. Dans le cas de la première amélioration, cela serait bien trop réducteur puisque notre nombre de voisinages explorés reste relativement peu élevé ce qui nous ferait en changer trop rapidement. En effet, les réseaux sur les lesquels le nouvel algorithme sera évalué contiendront 5 à 15 couches ce qui laisserait donc 5 à 15 changements de voisinages différents. Sans pour autant chercher à explorer les voisinages de façon exhaustive, il est nécessaire de ne pas s'arrêter à la première solution pertinente. Le nouveau critère d'arrêt sera alors qu'un nombre fixe de recherches locales seront effectuées dans un voisinage, la meilleure solution atteinte sera conservée avant de passer au voisinage suivant.

- La procédure de changement de voisinage se fera donc sur la base d'une distance par rapport à la solution actuelle. Comme présenté dans le Chapitre 1, le problème d'optimisation de réseaux est généralement mal conditionné ce qui implique que de légers changements dans la solution peuvent impliquer de grandes différences dans la solution fournie. Cela signifie qu'il n'y a pas forcément de corrélation entre la proximité d'une solution et sa performance. Une solution intéressante pourrait donc se trouver dans une

région qui ne l'est globalement pas. Une nouvelle solution sera donc simplement générée en ajoutant à l'ancienne du bruit distribué de façon uniforme entre plus ou moins l'écart-type du vecteur de poids actuel. A l'inverse par exemple d'un bruit distribué normalement qui impliquerait un biais vers des solutions proches de la solution actuelle alors que cela n'est pas pertinent ici.

Soit l'algorithme proposé :

**Algorithme 4** Algorithme proposé : NN - RVV

**Entree**

$\hat{A}$ Données d'entraînement
$\hat{B}$ Données de test
$\hat{W}$ Vecteur des poids du réseau
$k$ Nombre d'itérations à effectuer
$m$ Nombre de couches dans le réseau

**Sortie**

$\hat{W}'$ Nouveau vecteur de poids

$best\_sol \leftarrow 0$
$last\_sol \leftarrow 0$
$i \leftarrow 0$
**Tant que** $i < k$ **faire**
// Recherche locale dans le voisinage courant
$\hat{W}' \leftarrow Recherche\_Locale(\hat{W}, \hat{A})$

// Évaluation de la performance du réseau, que l'on cherche à maximiser, après la recherche
$last\_sol \leftarrow$ Score_Précision($\hat{W}'$, $\hat{B}$)
// Le voisinage est exploré pour un nombre fixe d'itérations
**Si** i % 10 = 0 et i != 0 **Alors**
**Si** $last\_sol >= best\_sol$ **Alors**
// Si la solution obtenue après cette exploration est meilleure que la précédente, l'on sauvegarde les paramètres du réseau
sauvegarde_sol($\hat{W}'$, last_sol)
best_sol = last_sol
Sol_Voisine($\hat{W}$, $Entier\_Aleatoire(1, m-1)$)
**Sinon**
// Si la solution obtenue après cette exploration est inférieure à la dernière solution du précédent voisinage alors l'on recharge la configuration du réseau correspondante
chargement_sol($\hat{W}$)
**Fin Si**
**Fin Si**
i $\leftarrow i+1$
**Fin Tant que**

Une solution voisine sera donc générée de cette façon :

**Algorithme 5** Génération d'une solution voisine

**Input**

$\hat{W}$ Vecteur des poids du réseau

$c$ Couche du réseau à modifier

**Output**

$\hat{W}'_c$ Vecteur de poids voisin

// Extraction des poids du réseau correspondant à la couche cachée choisie pour former une solution voisine

$\hat{W}_c \leftarrow Extraire\_Poids(\hat{W}, c)$

// Perturbation sur les poids du réseau pour la couche cachée sélectionnée

$\hat{W}'_c \leftarrow \hat{W}_c + Bruit\_uniforme(-1.5 * Ecart\_Type(\hat{W}_c), 1.5 * Ecart\_Type(\hat{W}_c))$

Le choix du facteur de 1.5 qui multiplie l'écart-type du vecteur de poids est ici arbitraire. Cette valeur a été déterminée empiriquement après avoir essayé les valeurs 1, 1.5 et 2, 1.5 retournant généralement de meilleurs résultats.

L'algorithme proposé s'inspire finalement des variations de la Recherche à voisinages variables que sont la DVV et de la RVF, présentées plus tôt. Une solution voisine est généré par l'étape de perturbation comme dans l'algorithme de la RVF et une recherche locale est effectuée à partir de cette nouvelle solution. La question de la sélection du critère d'arrêt se pose également dans ce cas. Le temps de calcul était proposé dans le cas de RVF, ici un nombre fixe de recherches locales seront effectuées avant de générer une nouvelle solution voisine.

Néanmoins, l'algorithme proposé repose tout de même sur une structure de voisinage plus explicite que dans le cas de la RVF. Dans sa définition dans l'algorithme 4, la sélection de la couche cachée sur laquelle sera basée le changement de voisinage est sélectionnée aléatoirement. Il serait également également possible de sélectionner les couches de façon successive comme pour l'algorithme de la DVV.

# Chapitre 4

# Résultats

## 4.1 Les données

Tel que présenté précédemment, l'objectif de ce mémoire est de proposer une méthode d'entraînement hybride pour l'ajustement de réseaux de neurones. Cette méthode combinera recherches locales et heuristiques dans le but de proposer une méthode générale, indépendante de la nature des données sur lesquelles le modèle sera appliqué.

Afin de pouvoir tout de même évaluer les performances de la méthode proposée, les données qui seront utilisées correspondent à un problème de classification d'image. Ce problème est suffisamment complexe pour qu'il permette de mesurer les gains marginaux de performance liés à l'ajout de l'heuristique dans le processus d'optimisation. De plus le choix de ce problème nous assure également l'absence d'une structure particulière dans les données qui pourrait influencer le processus d'optimisation comme ce serait par exemple le cas pour des données temporelles. Ici, seule la valeur numérique des pixels composant l'image sera utilisée pour répondre au problème de classification.

Le jeu de données CIFAR-10 (Krizhevsky, 2009), développé par l'institut canadien de recherche avancée (CIFAR - *Canadian Institute For Advanced Research*), est l'un des jeu de données les plus utilisés pour la recherche en apprentissage machine (ML - *Machine Learning*) (Hamner, 2017). Le jeu de données contient 60000 images en couleur

chacune d'une résolution de $32 \times 32$ pixels. Chacune de ces images correspond à l'une de dix catégories, l'objectif étant de prédire correctement la catégorie à laquelle chacune correspond, ces dix catégories sont mutuellement exclusive ce qui signifie que chaque image est associée à une et une seule des dix catégories. De plus, le jeu de données est parfaitement balancé, chaque classe contient donc 6000 images.

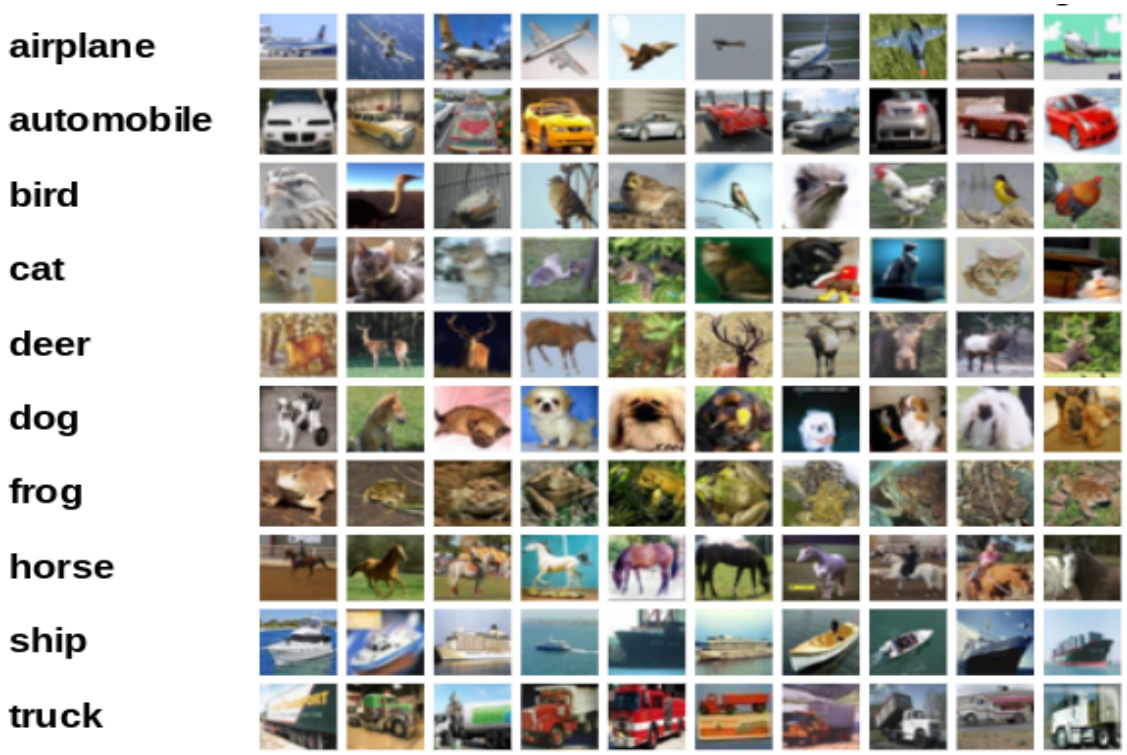



FIGURE 4.1 – Examples d'images du jeu de données CIFAR10, tiré de (Krizhevsky, 2009)

Chaque classe sera codifiée numériquement afin de pouvoir évaluer la valeur de la fonction de perte, présentée au chapitre deux, au cours de l'entraînement du modèle :

— Airplane : 0

— Automobile : 1

— Bird : 2

— Cat : 3

— Deer : 4

— Dog : 5

— Frog : 6

— Horse : 7

— Ship : 8

— Truck : 9

Le format des données nécessite quelques transformations afin de plus efficacement les exploiter. Chaque observation est donc une image en couleur de $32 \times 32$ pixels mais chacun de ces pixels va stocker une valeur pour chaque couleur primaire qui le compose soit rouge, vert et bleu. Ce qui signifie que pour chaque image, $32 \times 32 \times 3 = 3072$ valeurs numériques la compose.

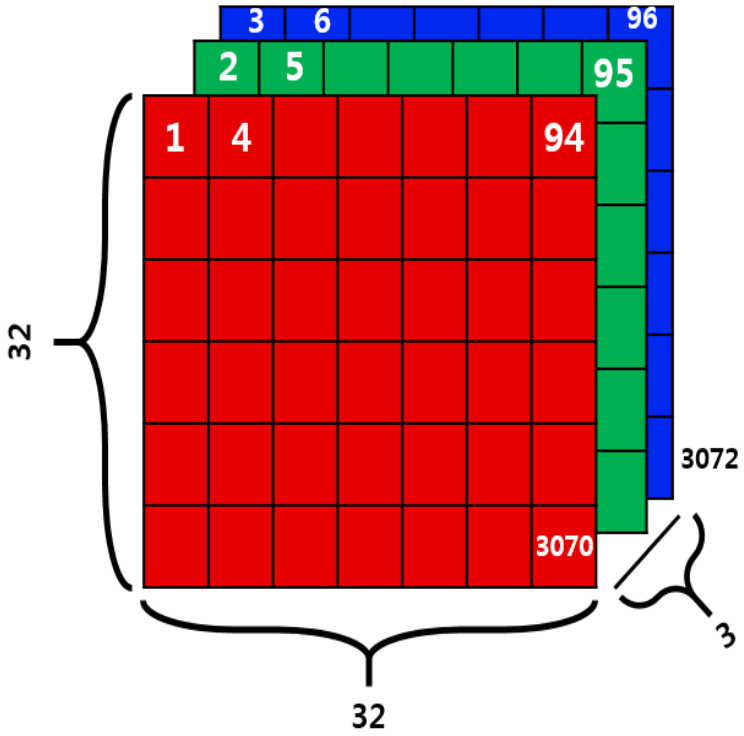



FIGURE 4.2 – Présentation des données sous forme de matrices, tiré de (Keshari, 2019)

Afin de faciliter leur traitement, la structure matricielle des données sera abandonnée au profit de la création d'un seul vecteur de données pour chaque image qui correspondra à la concaténation des trois matrices de couleurs représentées sur une seule dimension. Soit pour chaque image un vecteur numérique contenant 3072 points de données :

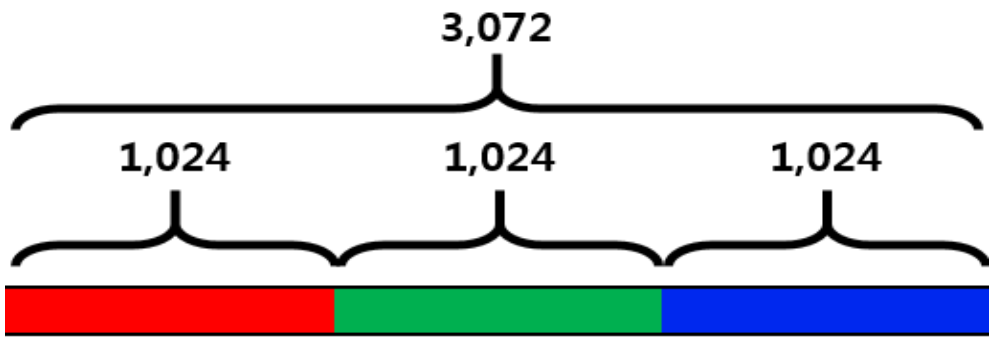



FIGURE 4.3 – Présentation des données sous forme de vecteur, tiré de (Keshari, 2019)

L'évaluation de modèles en apprentissage machine est généralement effectuée en utilisant des données sur lesquelles le modèle n'a pas été entraîné. La procédure implique de diviser la jeu de données utilisés en deux sous-ensembles. Le premier sera utilisé afin d'ajuster le modèle et est nommé 'ensemble d'entraînement'. Le second est uniquement uniquement utilisé à des fins de prédictions qui seront comparées aux valeurs attendues. Cet ensemble est nommé 'ensemble de test' (ou également 'ensemble de validation'). Pour ce mémoire, les images contenues dans ce jeu de donnés seront divisées ainsi : 50 000 images pour l'ensemble d'entraînement et 10 000 pour l'ensemble de test.

La présentation des résultats obtenus suivra également une structure fixe. Chaque graphique de résultat contiendra trois informations, la valeur de la fonction de perte (ou fonction objectif du problème d'optimisation), précision de la classification du modèle sur l'ensemble de données d'entraînement ainsi que le score de précision sur l'ensemble de test. L'échelle du graphique contenant les résultats de la fonction de perte sera définie automatiquement selon les résultats obtenus au cours de l'entraînement. L'échelle des graphiques des scores de précision sera elle un ratio des prévisions correctes compris entre 0 et 1. La valeur présentée dans les différents graphiques sera la solution actuelle obtenue par l'algorithme plutôt que la meilleure solution, ce qui explique son comportement non-monotone.

Il est à noter que le processus d'optimisation reste un processus stochastique et qu'une part d'aléatoire va influencer les résultats. Afin de considérer cela, l'annexe 1 de ce mémoire contiendra les mêmes graphiques de résultats que ceux présentés au cours de ce chapitre mais pour différentes expériences réalisées dans des conditions similaires.

## 4.2 Mise en oeuvre

Afin de vérifier les performance de l'algorithme proposé, des réseaux de neurones seront ajustés afin de comparer les gains apportés par l'ajout de la recherche à voisinage variable ainsi que les performances entre chacune des méthodes locales présentées au cours du premier chapitre.

Dans le but d'obtenir des résultats comparables, un maximum de paramètres seront fixés au cours de l'ajustement des modèles. Par exemple la valeur optimale du paramètre $\alpha$, qui correspond à l'importance de la mise à jour des poids du réseau après le calcul des gradients, sera différent en fonction de l'architecture du modèle mais également de la recherche locale utilisée. La valeur optimale de ces différentes paramètres seront ajustés afin de maximiser les performances des différentes méthodes, afin que leur comparaison soit juste, mais l'obtention de ces valeurs ne sera pas présentée dans plus de détails. Le lecteur

est invité à se référer à (Hutter, Lücke et Schmidt-Thieme, 2015) pour une présentation plus détaillée de différentes méthodes pour l'obtention de ces paramètres.

Tel que présenté dans le premier chapitre de ce mémoire, l'optimisation du modèle variera en fonction de la structure du réseau. Le nombre de couches contenues dans un réseau sera amené à évoluer, pour autant le nombre de neurones contenus dans chaque couche sera fixé à 128 neurones.

L'implémentation de l'algorithme et des différents cas de tests présentés dans le prochain chapitre ont été effectués dans le langage de programmation Python. La librairie *Numpy* sera utilisée pour l'implémentation de quelques fonctions mathématiques. L'utilisation de la librairie *Pytorch*, une librairie d'apprentissage machine open-source, permet de faciliter l'implémentation de ce projet sur des GPU. Finalement, la librairie *Matplotlib* facilite la production des différents graphiques de résultats. Les versions de ces librairies au moment de la production des résultats ci-dessous sont données par ce tableau :

| *Librairie* | *Version* |
|---|---|
| Numpy | 1.19.5 |
| Pytorch | 1.10.0 |
| Matplotlib | 3.2.2 |

Les résultats ont été obtenus sur une configuration contenant un processeur Intel(R) Xeon(R) @ 2.30GHz, 12GB de RAM et une carte graphique Tesla P100 avec 16GB de VRAM.

## 4.3 Résultats

### 4.3.1 Résultats de référence

Les trois méthodes de recherches locales sont ici utilisées dans le but de présenter des résultats de référence avant d'inclure les résultats de la nouvelle méthode proposées. Dans

le but de faciliter la présentation des résultats les acronymes en anglais pour les différentes méthodes seront utilisées. La descente de gradients seule sera dénommée « SGD » et lorsque qu'elle sera intégrée à l'algorithme de la RVV l'acronyme « SGD-VNS » sera utilisé. De la même façon, les acronymes « RMSprop-VNS » et « ADAM-VNS » seront utilisées pour dénommer respectivement les méthodes RMSprop et ADAM intégrées à la RVV.

Ces trois méthodes ont été introduites dans plus de détails au cours du chapitre 1.

Trois réseaux de neurones seront ajustés contenant chacun 5 couches cachées, chacune contenant 128 neurones. La fonction d'activation appliquée dans le réseau sera la fonction RElu également présentée au cours de la section 1.1.2 de ce mémoire. Finalement, la fonction de perte sera la fonction d'entropie-croisée, elle aussi présentée plus tôt.

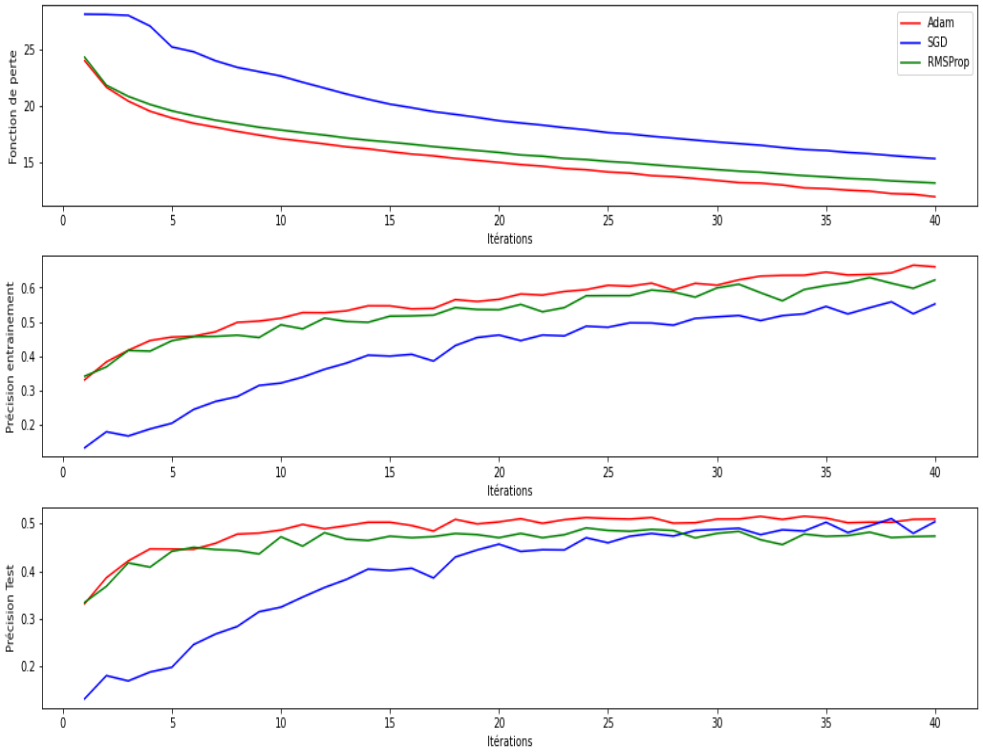



FIGURE 4.4 – Comparaison Adam, RMSprop, SGD pour 5 couches cachées et 40 itérations

Les scores de précisions obtenus sur l'ensemble de test par les trois méthodes sont relativement proches bien que la descente de gradients (SGD) converge bien moins rapide-

ment. L'on peut également noter que les recherches locales ADAM et RMSprop souffrent de sur-apprentissage, ce qui signifie qu'elles semblent continuer à obtenir un score de précision croissant sur l'ensemble d'entraînement alors que leur performance stagne très rapidement sur l'ensemble de test.

### 4.3.2 Résultats de référence pour 10 couches cachées

Tel que présenté au cours de la section 1.1.3, en augmentant le nombre de neurones du réseau la difficulté de son entraînement croit également. Les résultats suivants sont obtenus en ajoutant 5 couches cachées aux réseaux entraînés, soit trois réseaux contenant 10 couches cachées contenant chacune 128 neurones.

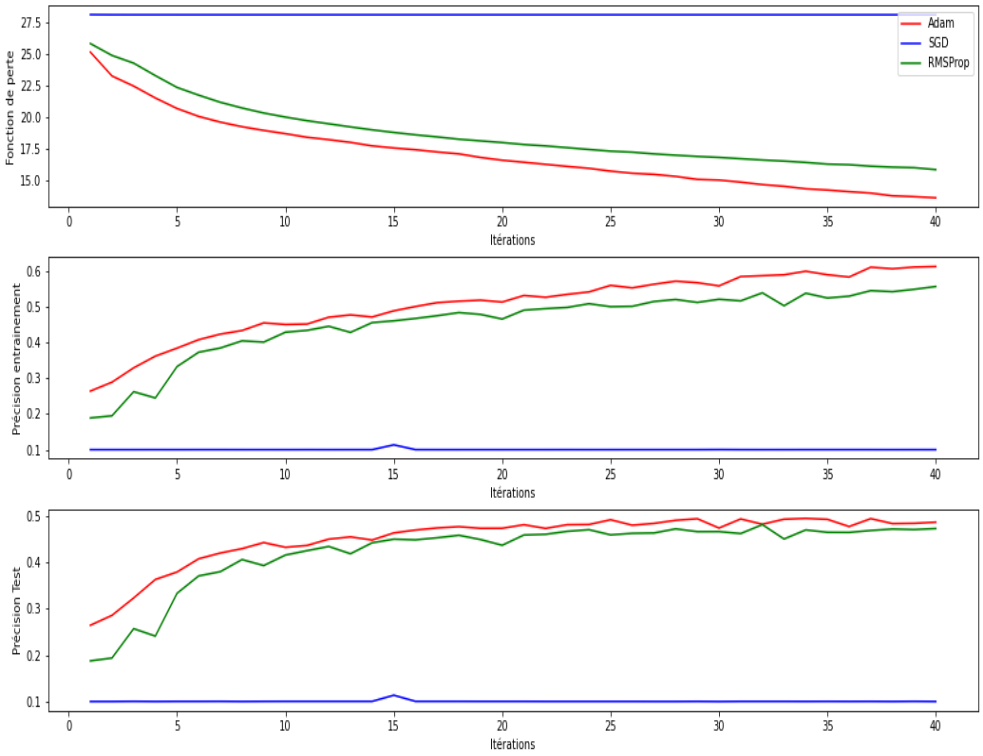



FIGURE 4.5 – Comparaison Adam, RMSprop, SGD pour 10 couches cachées et 40 itérations

L'on peut remarquer que l'ajout de seulement 5 couches cachées implique que la descente de gradients simple (SGD) est incapable d'entraîner le modèle et reste bloquée dans

des minimas locaux. De plus, RMSprop semble converger moins rapidement. ADAM à l'inverse ne semble pas présenter des résultats très différents.

### 4.3.3 Comparaison des recherches locales avec et sans RVV

La suite des résultats intégreront l'ajout de l'algorithme de RVV proposé au chapitre 3, l'objectif étant de comparer les méthodes locales seules avec les méthodes locales intégrées à la RVV.

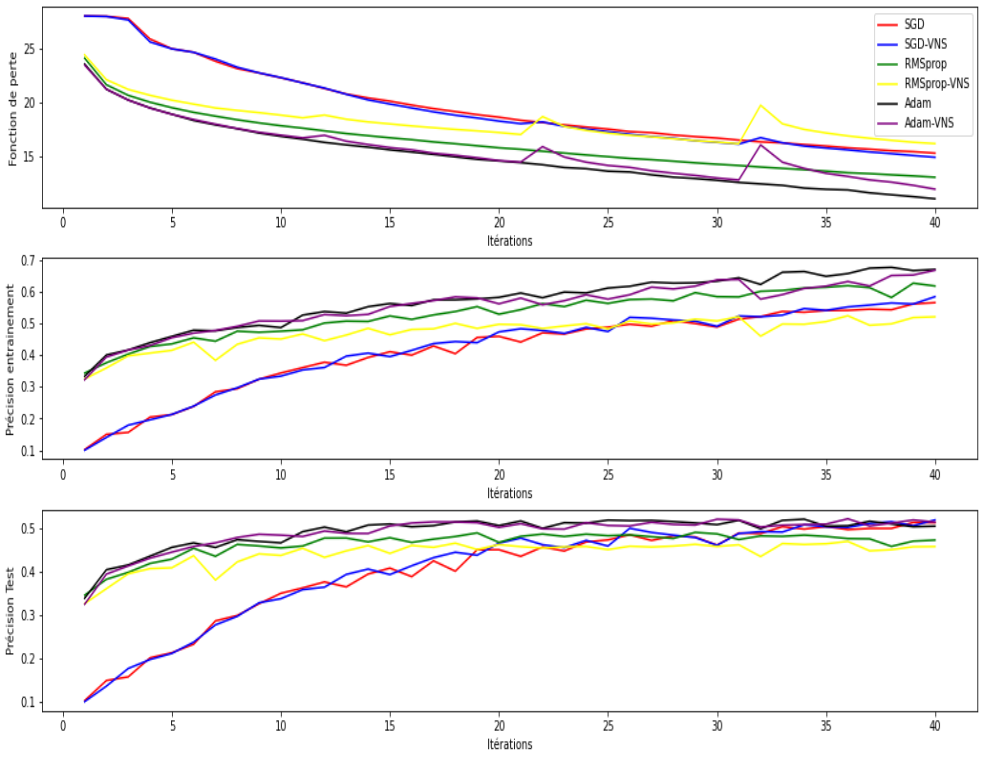



FIGURE 4.6 – Comparaison recherches locales avec et sans RVV pour 5 couches cachées et 40 itérations

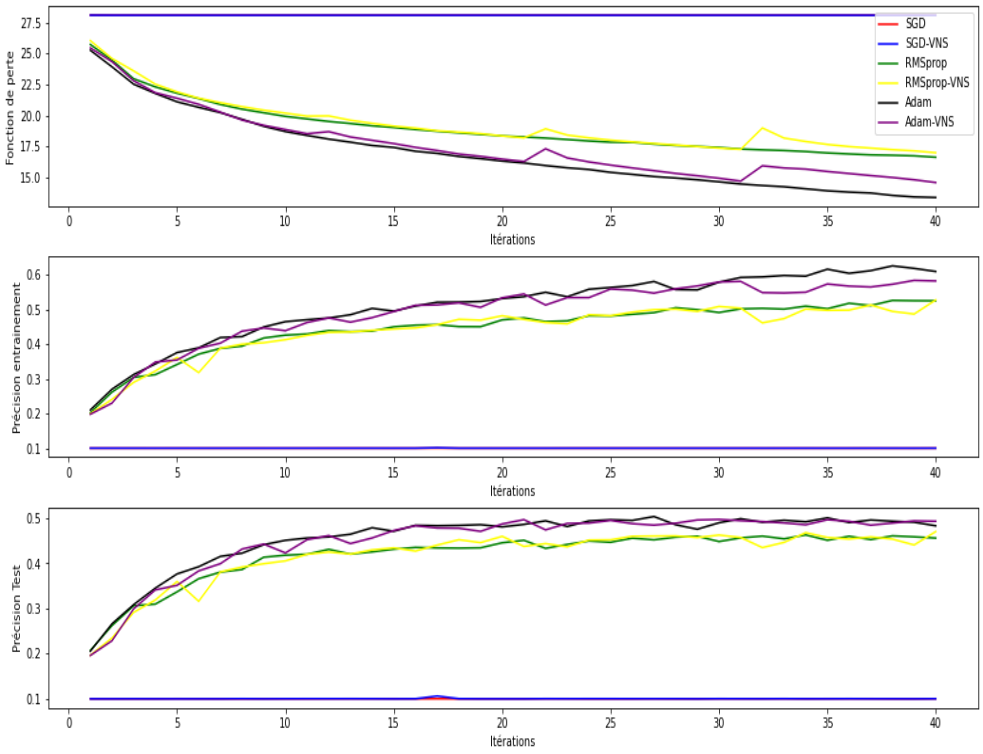



FIGURE 4.7 – Comparaison recherches locales avec et sans RVV pour 10 couches cachées et 40 itérations

RMSprop semble présenter des résultats inférieurs aux autres méthodes et ne semble pas gagner en efficacité avec l'ajout de la RVV. Les gains associés à l'ajout de la RVV pour chaque méthode sera présentée plus détails dans les prochains graphiques.

Un point intéressant est que l'ajout de la RVV semble offrir aux méthodes locales la capacité de sortir de minimas locaux. Face à un réseau de 10 couches cachées, scénario dans lequel la descente de gradient seule (SGD) était incapable d'entraîner le modèle dans la figure 4, l'ajout de la RVV (SGD-VNS dans la figure 4.6) permet à la descente de gradients de sortir de minimas locaux.

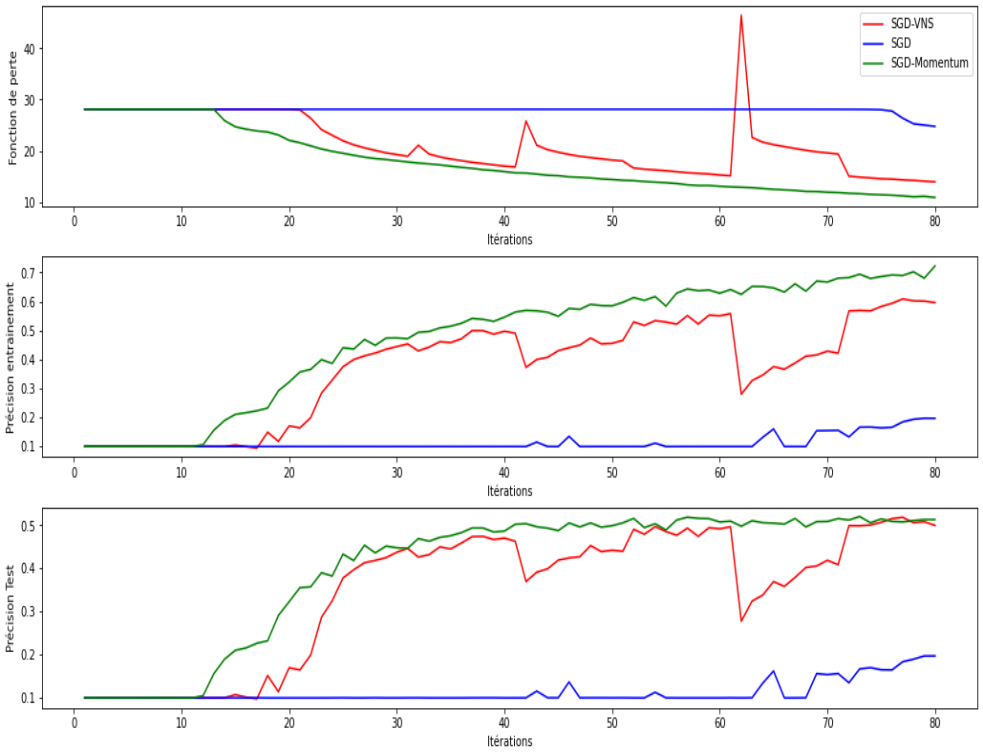



FIGURE 4.8 – Comparaison SGD avec et sans RVV pour 10 couches cachées et 80 itérations

Les résultats sont ici également comparés à la descente de gradients avec l'ajout d'un terme d'élan (SGD-Momentum) tel que présenté au chapitre 1. Le terme d'élan permet tout de même une convergence un peu plus rapide. Il est également important de noter que l'ajout du terme d'élan (SGD-Momentum) converge plus rapidement en considérant des déplacements plus importants en suivant les gradients ce qui semble bien performer dans cet exemple. Cependant cette méthode s'expose à dépasser des solutions pertinentes puisqu'elle limite la capacité d'intensification de la méthode, ce qui n'est pas le cas de la méthode reposant sur la RVV.

Cependant, la RVV (SGD-VNS) converge bien plus rapidement une fois le changement de voisinage effectué à l'itération 20. Il est également possible de remarquer les changement de voisinages aux itérations 40 et 60 qui diminuent fortement le score de précision. On peut en tirer que, tel qu'attendu, l'algorithme continue d'explorer d'autres solution à partir du moment ou la recherche locale ne suffit plus à améliorer la solution.

Finalement l'algorithme restaure une ancienne solution à l'itération 70 qui correspond à une solution très proche de celle atteinte par la descente de gradients avec une terme d'élan (SGD-Momentum).

### 4.3.4 Résultats selon la profondeur du réseau

Afin de mesurer la stabilité des performances de l'algorithme proposé face à des problèmes de tailles différentes, les méthodes de recherches locales seront comparées avec et sans l'ajout de la RVV face à des réseaux de tailles différentes. Cette taille du réseau sera mesurée selon le nombre de couches cachées qu'il contient, chaque couche cachée ajoutant 128 neurones au réseau comme précédemment. Ces résultats permettent également de vérifier les gains de l'ajout de la RVV à l'algorithme d'entraînement.

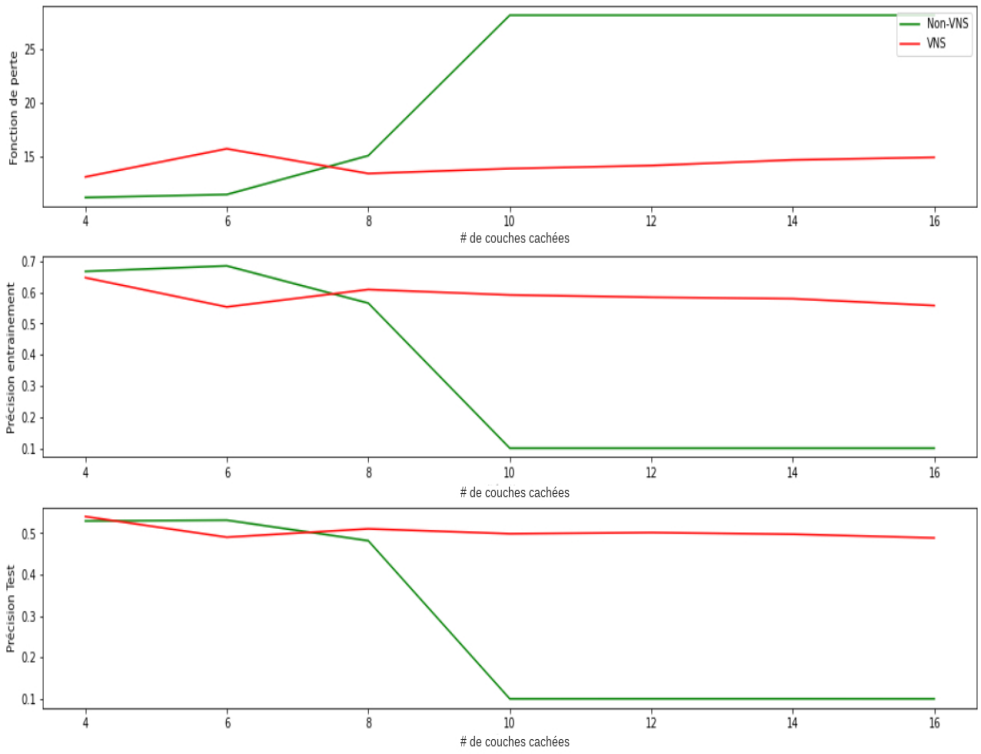



FIGURE 4.9 – Comparaison de la descente de gradients avec et sans RVV selon la profondeur du réseau pour 80 itérations chacun

Dans le cas de la descente de gradients classique la capacité de l'algorithme proposé à se sortir de minimas locaux continue de se vérifier. La descente de gradients seule (SGD) est incapable d'entraîner un modèle de 10 couches cachées ou plus. L'ajout de la RVV (SGD-VNS) présente des résultats bien meilleurs qui restent constant pour toutes les tailles de réseaux testées.

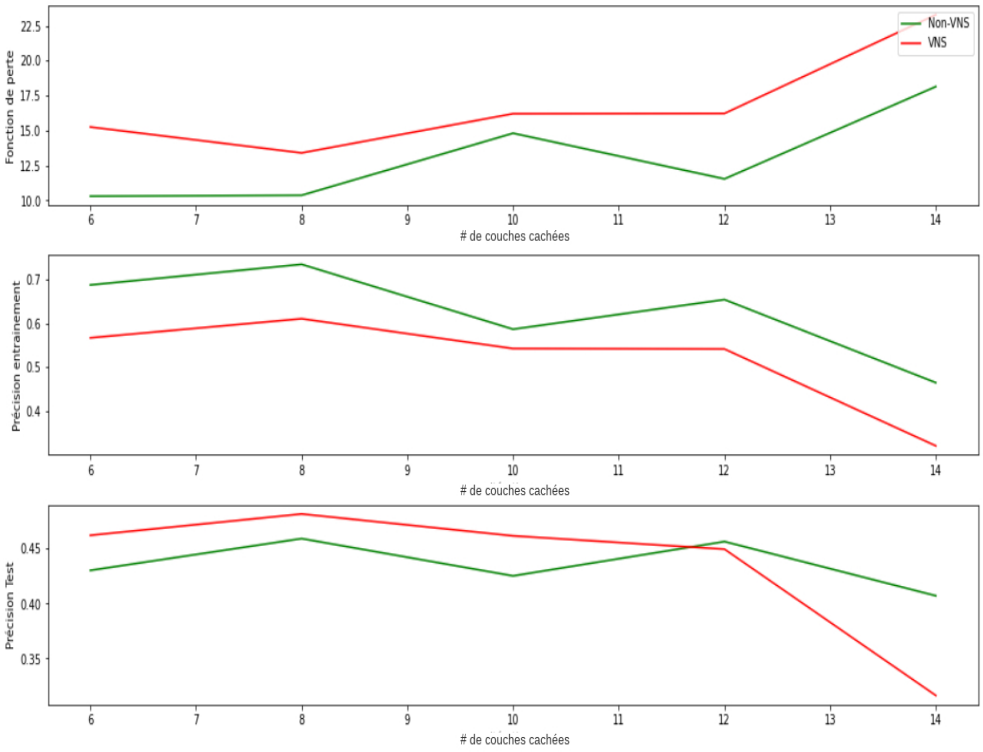



FIGURE 4.10 – Comparaison RMSprop avec et sans RVV selon la profondeur du réseau pour 80 itérations chacun

L'on remarque ici une baisse importante des performances de la RVV (SGD-VNS) pour le réseau de 14 couches cachées. Ce comportement n'a pas de justification particulière et tient simplement à la nature stochastique de l'entraînement. En effet, ce comportement ne se reproduit pas lorsque l'expérience est répétée (Voir Annexe 1).

Les gains sont moins significatifs pour la recherche locale RMSprop. L'ajout de la RVV apporte un léger gain de précision pour les plus petits réseaux mais finit par performer moins bien que la recherche locale seule pour des réseaux plus larges.

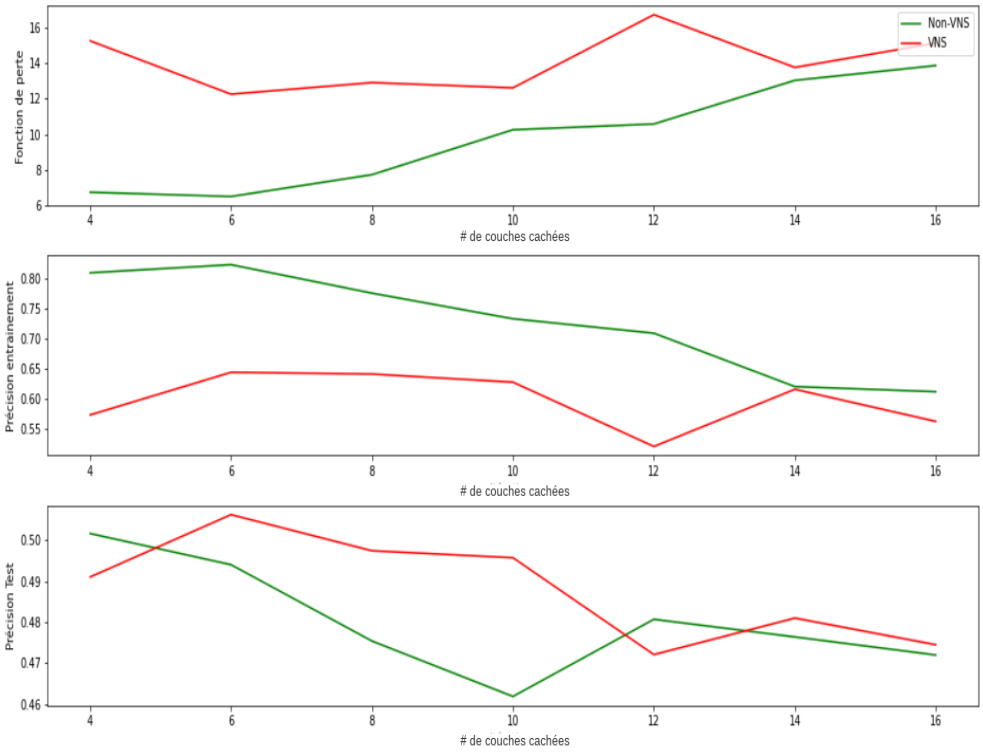



FIGURE 4.11 – Comparaison ADAM avec et sans RVV selon la profondeur du réseau pour 80 itérations chacun

Dans le cas de la recherche locale ADAM, les gains sont également plus importants pour les plus petits réseaux pour le score de précision sur l'ensemble de test.

Un point important à soulever est que la performance de l'algorithme proposé dépendra du nombre de voisinages explorés et que la probabilité de trouver un voisinage intéressant à chaque changement voisinage décroît à mesure que la taille du réseau croit, dans ces exemples la possibilité de changer de voisinage apparaît toutes les dix itérations et pour chacune des profondeurs testées les réseaux sont entraînés pendant 80 itérations. Les performances décroissantes en fonction de la taille du réseau pourraient donc être expliquées par un nombre d'itération trop faible pour que l'algorithme puisse raisonnablement trouver un changement de voisinage intéressant.

Un second point d'intéret est que la pertinence de l'ajout de la RVV dépendra de la recherche locale employée. Tel que présenté au chapitre 2, la méthode RMSprop intègre un terme à la descente de gradients classique afin de favoriser la diversification de la recherche. Ce qui est redondant avec l'ajout de la RVV et explique que les gains liés à l'ajout de la RVV soient moins significatifs pour la recherche locale RMSprop. À l'inverse, la méthode ADAM intègre entre autre un terme de pondération adaptatifs aux gradients ce qui facilite la convergence de la méthode. Cette particularité se transmet également à la RVV qui intègre la recherche locale ADAM.

Afin de vérifier cela, les prochains résultats présenteront l'entraînement d'un réseau de taille fixe mais pour un nombre d'itérations plus élevée. De plus, comme les précédents résultats n'ont pas présenté de synergie particulière entre la méthode RMSprop et la RVV, le point présenté au paragraphe précédent ne sera vérifié qu'uniquement pour les méthodes ADAM et la descente de gradients classique.

### 4.3.5 Résultats précis pour chaque recherche locale

Les résultats suivants présentent les performances de chaque recherche locale avec et sans l'ajout de la RVV pour 150 itérations et pour des réseaux plus profonds de respectivement 10 et 15 couches cachées.

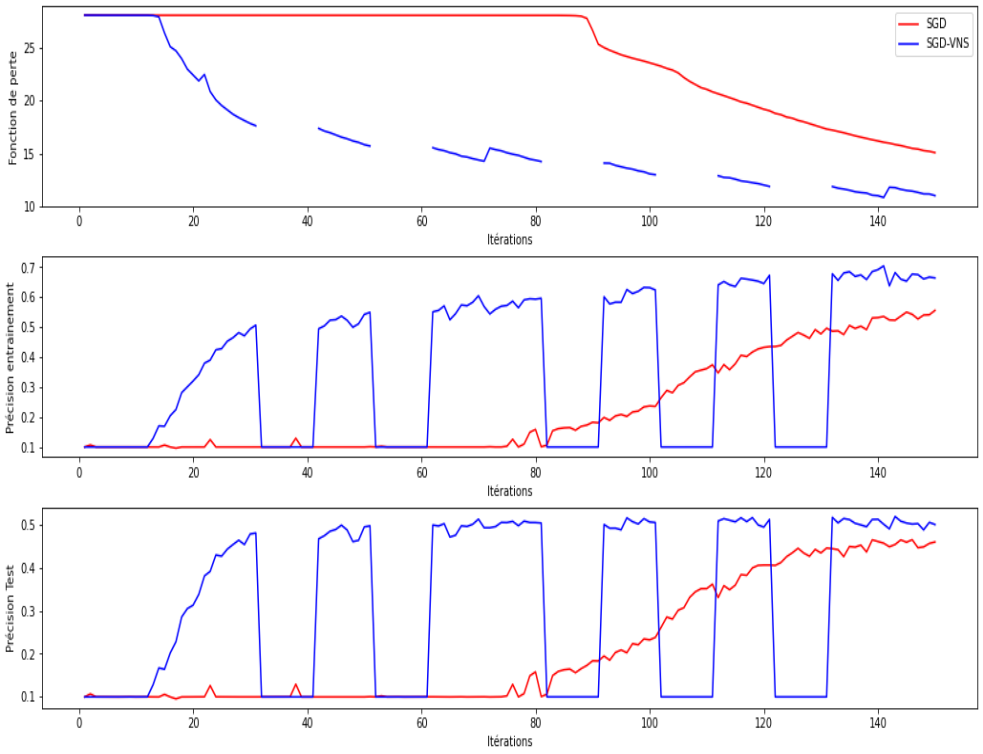



FIGURE 4.12 – Comparaison SGD avec et sans RVV pour 10 couches cachées et 150 itérations

En ajustant un modèle pendant 150 itérations, la descente de gradients seule (SGD) parvient finalement à se sortir du minima local dans lequel elle reste bloquée pendant 80 itérations. A l'inverse l'ajout de la RVV (SGD-VNS) offre une convergence bien plus rapide et atteint un meilleur score de précision sur l'ensemble de test.

Les espaces vides dans la courbe de la fonction de perte siginfient qu'un changement de voisinage a mené la fonction à une valeur supérieure à l'échelle du graphique.

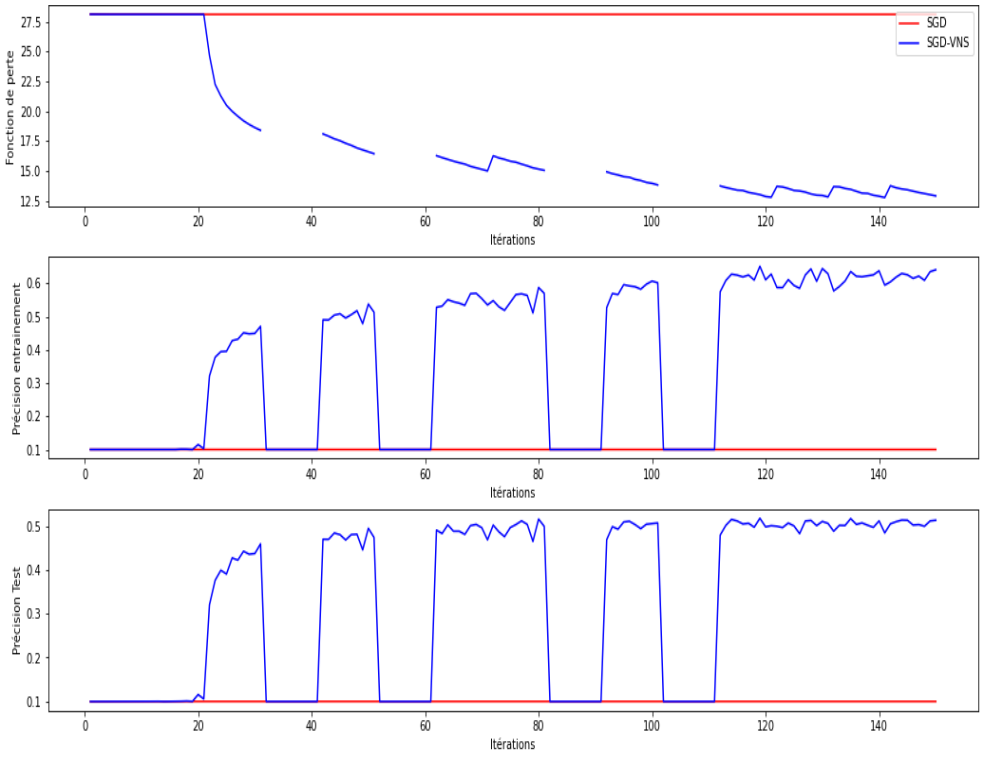



FIGURE 4.13 – Comparaison SGD avec et sans RVV pour 15 couches cachées et 150 itérations

Néanmoins, dès 15 couches cachées contenues dans le réseau la descente de gradients reste complètement bloquée sur la valeur initiale des poids. A l'inverse, l'algorithme proposé converge toujours relativement rapidement et atteint un score final équivalent au graphique précédent malgré que le problème d'optimisation soit ici plus difficile.

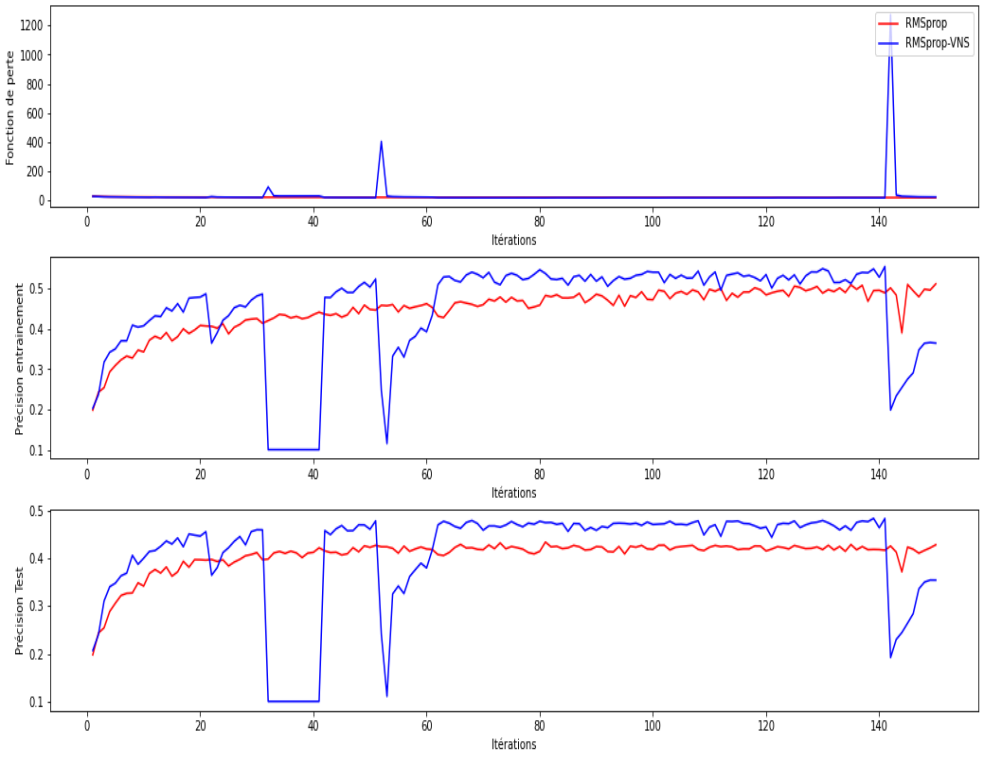



FIGURE 4.14 – Comparaison RMSprop avec et sans RVV pour 10 couches cachées et 150 itérations

Contrairement aux précédents résultats avec la recherche locale RMSprop, l'ajout de la RVV (RMSprop-VNS) semble apporter un gain de précision signicatif ainsi que de faciliter la convergence de l'algorithme.

L'algorithme RMSprop-VNS semble terminer sur un résultat inférieur à l'itération 150 du à un changement de voisinage inintéressant à l'itération 140. Ce n'est pas un problème ici puisque bien que cela ne soit pas visible, l'algorithme rechargera la précédente meilleure solution avant le changement de voisinage de l'itération 140 qui était déjà meilleure que la meilleure solution atteinte par RMSprop seule.

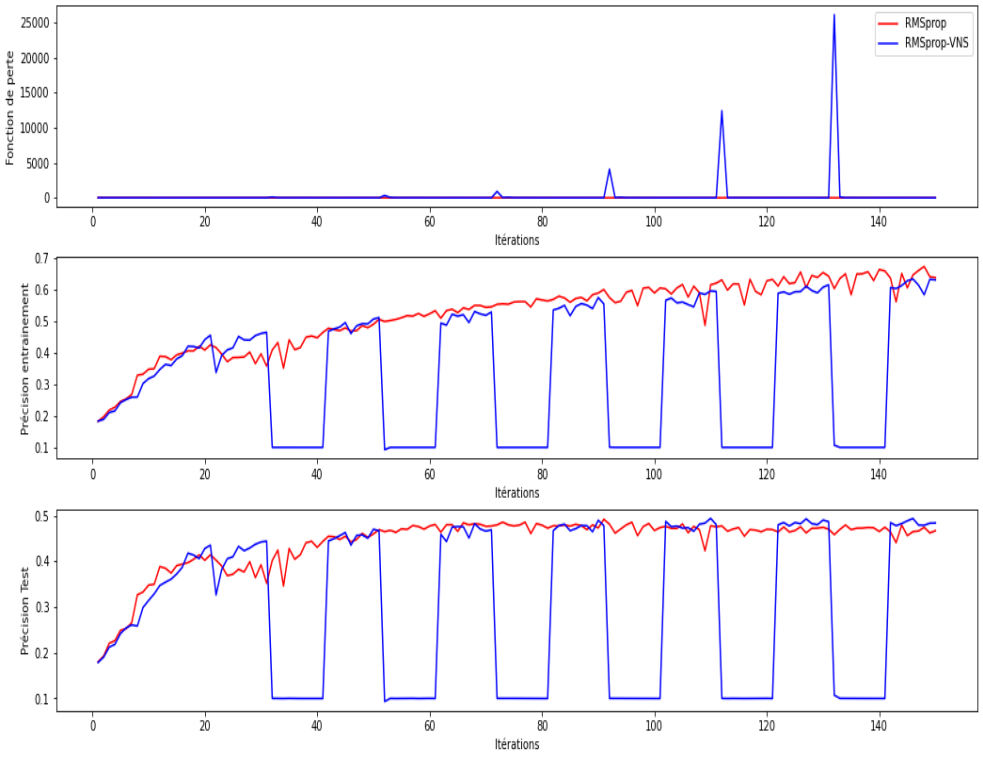



FIGURE 4.15 – Comparaison RMSprop avec et sans RVV pour 15 couches cachées et 150 itérations

Pour autant, ces gains liés à l'ajout de la RVV semblent bien moins pertinents avec ce réseau plus large. Les deux scores de précision finaux sont très proches et la convergence des deux méthodes sont très similaires.

Malgré cela, l'ajout de la RVV présente un avantage supplémentaire en offrant une meilleure capacité de généralisation au modèle. La méthode RMSprop seule surestime ses performances sur l'ensemble d'entraînement alors que ses véritables performances sur l'ensemble de test sont plus faibles. RMSprop-VNS, la méthode avec l'ajout de la RVV, à l'inverse présente des résultats similaires sur les deux ensembles de donnés ce qui signifie que le modèle entraîné généralisera mieux ses résultats sur de nouvelles données.

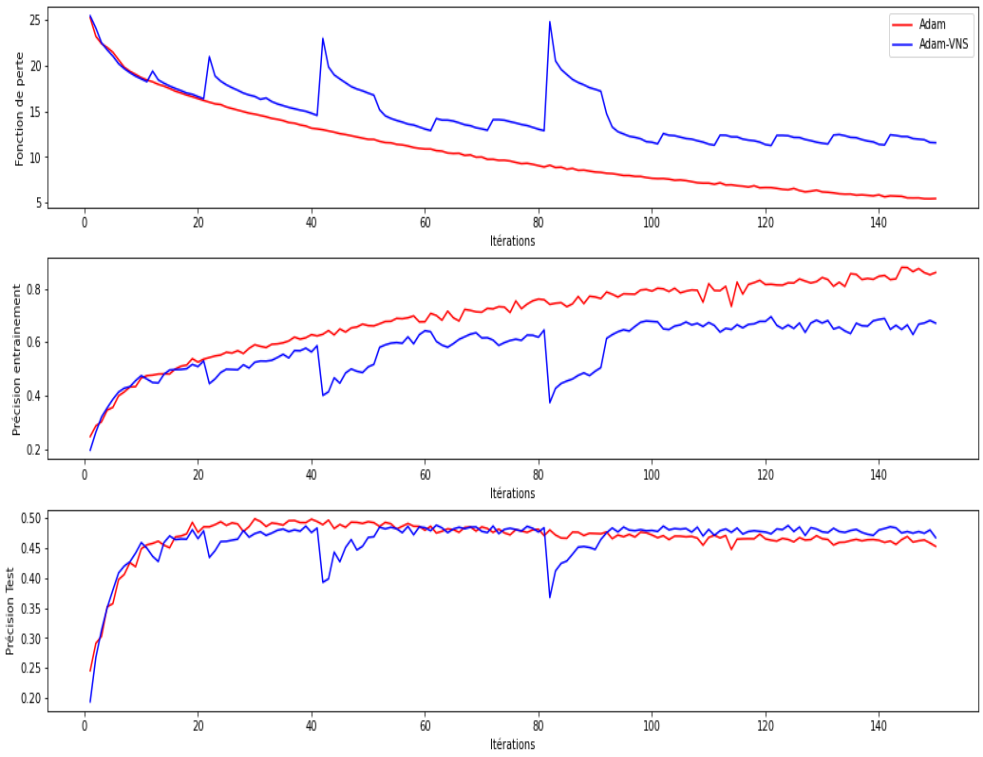



FIGURE 4.16 – Comparaison ADAM avec et sans RVV pour 10 couches cachées et 150 itérations

Finalement, l'on retrouve des conclusions similaires pour la recherche locale ADAM. Le gain de précision apporté par l'ajout de la RVV (ADAM-VNS) est visible mais reste assez faible alors que la nouvelle méthode ne modifie quasiment pas la vitesse de convergence de l'algorithme.

Cependant la capacité de l'algorithme proposé à favoriser une meilleure généralisation du modèle est d'autant plus significative ici. Les performances de la méthode ADAM seule apparaissent croissantes sur l'ensemble d'entraînement alors qu'elles stagnent très rapidement sur l'ensemble de test. A l'inverse les performances de l'algorithme proposé (ADAM-VNS) restent stables entre les deux ensembles.

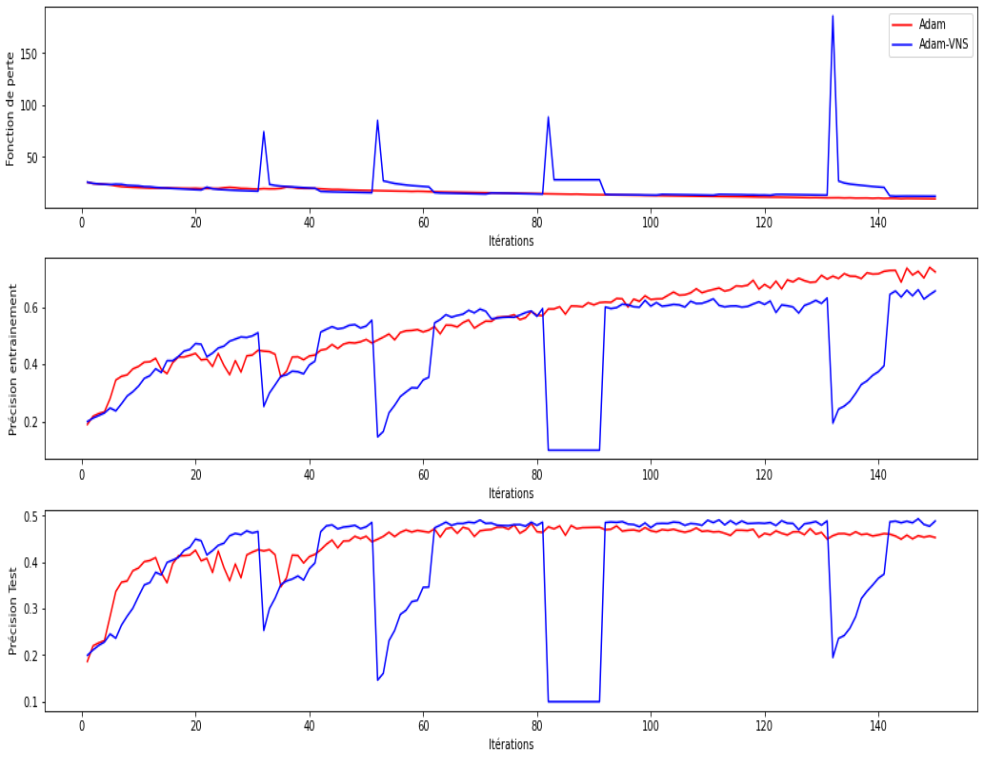



FIGURE 4.17 – Comparaison ADAM avec et sans RVV pour 15 couches cachées et 150 itérations

Enfin, les mêmes observations peuvent être tirées de l'entraînement d'un réseau de 15 couches cachées. La méthode proposée favorise toujours fortement la capacité de généralisation du modèle. Le gain de précision apporte par l'ajout de la RVV est tout de même plus significatif ici si l'on considère les résultats obtenus à l'itération 150. Ce point semble continuer de confirmer que les gains associés à l'algorithme proposé croit à mesure que le nombre de paramètres dans le réseau augmente, soit le nombre de couches cachées qu'il contient.

## 4.4 Stabilité de la méthode

Les tableaux 4.1, 4.2 et 4.3 présentés ci-dessous agrègent les résultats obtenus par les recherches locales seules et les méthodes qui intègrent la RVV lorsque l'expérience est répétée 11 fois. Ces tests correspondent à ceux présentés à la section 4.3.5 de l'analyse des résultats.

Un score moyen ou minimum de 10 dans ces tableaux indique que le ou les modèles n'ont pas été capables de converger ce qui revient à un classificateur aléatoire puisque le nombre de classes à prédire est de 10.

Aucune nouvelle conclusion n'est à tirer de ces tableaux si ce n'est que ces derniers semblent constants sur 11 expériences. L'on retrouve ainsi un gain de précision en moyenne ainsi que de stabilité pour l'algorithme proposé tout en offrant une meilleure capacité de généralisation entre les ensembles d'entraînement et de test.

Dans le cas de la descente de gradients simple (Tableau 4.1), les gains associés à l'ajout de la RVV sont très importants. L'on peut principalement noter que pour 10 couches cachées et 11 expériences, le nouvel algorithme (SGD-VNS) a toujours assuré la convergence de la recherche contrairement à la méthode locale seule. Néanmoins, bien que les résultats restent très intéressants pour 15 couches cachées, l'ajout de la RVV n'assure pas que la méthode parviendra à converger en 120 itérations comme le présente le score minimum de précision de 10.

TABLEAU 4.1 – Comparaison des scores de précision obtenus par SGD et SGD-VNS sur 11 expériences, pour 120 itérations chacunes.

| | **10 Couches Cachées** | | | | **15 Couches Cachées** | | | |
|---|---|---|---|---|---|---|---|---|
| | **SGD** | | **SGD-VNS** | | **SGD** | | **SGD-VNS** | |
| | **Entrainement** | **Test** | **Entrainement** | **Test** | **Entrainement** | **Test** | **Entrainement** | **Test** |
| **Moyenne** | 20.86 | 19.79 | 61.69 | 50.30 | 10.00 | 10.00 | 41.84 | 35.82 |
| **Ecart-type** | 15.43 | 13.17 | 3.43 | 0.84 | 0.00 | 0.00 | 25.33 | 20.49 |
| **Min** | 10.00 | 10.00 | 57.40 | 49.11 | 10.00 | 10.00 | 10.00 | 10.00 |
| **Max** | 55.60 | 46.00 | 66.45 | 51.50 | 10.00 | 10.00 | 64.00 | 52.00 |

Tel qu'initialement présenté au cours de la section 4.3, les gains associés a l'ajout de la RVV à la méthode RMSprop sont beaucoup moins intéressant que pour les autres méthodes. Cela se confirme par le tableau 4.2 qui, bien qu'il présente un léger gain de précision et de stabilité, ne se compare pas aux gains que l'on retrouve pour la descente de gradients simple et ADAM.

TABLEAU 4.2 – Comparaison des scores de précision obtenus par RMSprop et RMSprop-VNS sur 11 expériences, pour 120 itérations chacunes.

| | **10 couches** | | | | **15 couches** | | | |
|---|---|---|---|---|---|---|---|---|
| | **RMSprop** | | **RMSprop-VNS** | | **RMSprop** | | **RMSprop-VNS** | |
| | **Entrainement** | **Test** | **Entrainement** | **Test** | **Entrainement** | **Test** | **Entrainement** | **Test** |
| **Moyenne** | 62.98 | 45.76 | 52.86 | 46.23 | 55.81 | 45.83 | 55.31 | 47.38 |
| **Ecart-type** | 6.05 | 1.47 | 4.14 | 1.63 | 5.89 | 1.67 | 4.00 | 1.58 |
| **Min** | 51.20 | 42.80 | 45.52 | 42.80 | 47.80 | 43.20 | 48.03 | 43.93 |
| **Max** | 73.30 | 48.36 | 59.66 | 48.40 | 65.48 | 48.76 | 63.11 | 49.60 |

Il est tout de même intéressant de noter que les gains associés à l'ajout de l'algorithme proposé sont finalement très importants même s'il est associé à une méthode déjà complexe comme ADAM. Le tableau 4.3 présente un gain moyen de précision très significatif pour ADAM-VNS sur des réseaux contenant 15 couches cachées, 43.42% pour la mé-

thode seule contre 48.33% pour la nouvelle méthode. Le gain en stabilité est également très intéressant, 11.12 d'écart-type pour la méthode seule contre 0.91 pour la méthode intégrant la RVV.

TABLEAU 4.3 – Comparaison des scores de précision obtenus par ADAM et ADAM-VNS sur 11 expériences, pour 120 itérations chacunes.

| | **10 couches** | | | | **15 couches** | | | |
|---|---|---|---|---|---|---|---|---|
| | **ADAM** | | **ADAM-VNS** | | **ADAM** | | **ADAM-VNS** | |
| | **Entrainement** | **Test** | **Entrainement** | **Test** | **Entrainement** | **Test** | **Entrainement** | **Test** |
| **Moyenne** | 79.71 | 47.01 | 60.24 | 47.12 | 65.54 | 43.42 | 64.01 | 48.33 |
| **Ecart-type** | 3.78 | 1.06 | 10.07 | 4.71 | 20.00 | 11.12 | 3.91 | 0.91 |
| **Min** | 74.20 | 45.30 | 34.33 | 33.18 | 10.00 | 10.00 | 58.78 | 46.50 |
| **Max** | 86.20 | 48.90 | 69.32 | 49.43 | 79.67 | 48.49 | 69.60 | 49.80 |

# Conclusion

## Résumé

L'objectif de ce mémoire était la proposition d'un algorithme basé sur la métaheuristique de la recherche à voisinage variable dans le but d'entraîner des réseaux de neurones artificiels.

L'utilisation de métaheuristiques afin de résoudre des problèmes d'optimisation propre au domaine de l'apprentissage gagne en popularité depuis quelques années et présente des résultats très intéressants.

Il reste que les méthodes reposant sur la notion de voisinages de solutions, comme la recherche à voisinage variables, restent assez impopulaires principalement pour leur difficulté à être implémentées sur du matériel informatique spécialisé comme les GPUs.

Finalement, l'algorithme présenté propose une adaptation de la recherche à voisinage variable qui considère différentes particularités de l'entraînement de réseaux de neurones. Une définition relativement simple de la structure de voisinage du problème ainsi que l'emploi de différentes ressources informatiques récentes, comme des librairies de code, permet d'implémenter en totalité l'algorithme sur des GPUs.

Les résultats mesurés au cours du chapitre 4 présentent dans la majorité des cas un gain de précision du modèle grâce à l'ajout de la RVV mais permet principalement à la fois à l'algorithme de facilement sortir de minimas locaux mais aussi de régulariser directement les performances du réseau entre l'ensemble de données d'entraînement et de test. La

section 4.4 de l'analyse des résultats semble également confirmer la stabilité des résultats obtenus par cette nouvelle méthode. De plus, l'algorithme proposé offre ces avantages sans vraiment complexifier le processus d'entraînement d'un réseau de neurones. L'annexe 2 propose plus de détails sur ce processus ainsi qu'une analyse de complexité de ce dernier.

En conclusion, l'algorithme proposé présente plusieurs caractéristiques intéressantes pour un coût computationnel marginal extrêmement faible.

## 4.5 Limites et suites de l'étude

Bien que la section 4.4 présente des résultats qui restent consistants pour 11 expériences avec les conclusions présentées au cours de la section 4.3, cela ne suffit pas à conclure que les résultats sont statistiquement significatifs. Chaque expérience nécessite 8 à 10 heures de calcul sur une machine virtuelle dont les caractéristiques techniques ont été présentées à la section 4.2. Il serait donc intéressant de pouvoir effectuer un plus grand nombre de tests et valider la stabilité des résultats obtenus.

Une piste de recherche à considérer qui pourrait améliorer ces temps de calcul serait de considérer la capacité de parallélisation de l'algorithme proposé. Tel que présenté au cours du chapitre 3, les voisinages de solutions sont ici explorés de façon successive mais ils pourraient être explorés de façon parallèle puisqu'il n'existe pas de relation directe entre deux voisinages. Pour les résultats présentés plus tôt, chaque réseau de neurones implique l'exploration de 8 à 15 voisinages en fonction du nombre d'itérations considérées. L'on pourrait donc au mieux espérer un facteur de gain de vitesse égal au nombre de voisinages explorés, si ces derniers peuvent tous être effectués en parallèle.

Un autre piste de recherche future pourrait être de proposer un nouvelle structure de voisinage. La simplicité de celle proposée dans ce mémoire facilite son implémentation sur GPUs mais ne considère qu'en partie l'organisation spatiale du réseau auquel elle est appliquée (Une seule couche du réseau).

Finalement, les résultats présentés au chapitre 4 démontrent bien que les gains asso-

ciés à l'ajout de la RVV varient en fonction de la recherche locale employée. Puisque ces dernières sont généralement employées seules pour l'entraînement de réseaux de neurones, elles intègrent généralement une technique de diversification pour leur recherche ce qui peut être redondant avec l'objectif de l'ajout de la RVV. Il serait donc possible d'envisager le développement d'une recherche locale qui se prêterait plus a être associée à une métaheuristique comme la RVV. Cette méthode pourrait plus facilement se concentrer sur une intensification de la recherche puisque la RVV devrait déjà s'assurer de la diversification.

# Bibliographie

Abadi, M., A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu et X. Zheng. 2015, «TensorFlow : Large-scale machine learning on heterogeneous systems», URL `https://www.tensorflow.org/`, software available from tensorflow.org.

Abedinia, O., N. Amjady et N. Ghadimi. 2018, «Solar energy forecasting based on hybrid neural network and improved metaheuristic algorithm», *Computational Intelligence*, vol. 34, n° 1, doi :https://doi.org/10.1111/coin.12145, p. 241–260. URL `https://onlinelibrary.wiley.com/doi/abs/10.1111/coin.12145`.

Abiodun, O. I., A. Jantan, A. E. Omolara, K. V. Dada, N. A. Mohamed et H. Arshad. 2018, «State-of-the-art in artificial neural network applications : A survey», *Heliyon*, vol. 4, n° 11, doi :https://doi.org/10.1016/j.heliyon.2018.e00938, p. e00 938, ISSN 2405-8440. URL `https://www.sciencedirect.com/science/article/pii/S2405844018332067`.

Alba, E. et R. Marti. 2006, *Metaheuristic Procedures for Training Neutral Networks*, vol. 36, ISBN 978-0-387-33415-8, 71-86 p., doi :10.1007/0-387-33416-5.

Antoniadis, N. et A. Sifaleras. 2017, «A hybrid cpu-gpu parallelization scheme of variable neighborhood search for inventory optimization problems», *Electronic Notes in Discrete Mathematics*, vol. 58, doi :10.1016/j.endm.2017.03.007, p. 47–54.

Baklacioglu, T., O. Turan et H. Aydin. 2018, «Metaheuristic approach for an artificial neural network : Exergetic sustainability and environmental effect of a business aircraft», *Transportation Research Part D : Transport and Environment*, vol. 63, doi : https://doi.org/10.1016/j.trd.2018.06.013, p. 445–465, ISSN 1361-9209. URL `https://www.sciencedirect.com/science/article/pii/S1361920917306065`.

Baldi, P. et R. Vershynin. 2019, «The capacity of feedforward neural networks», *Neural networks : the official journal of the International Neural Network Society*, vol. 116, p. 288–311.

Beyer, H.-G. et H.-P. Schwefel. 2002, «Evolution strategies - a comprehensive introduction», *Natural Computing*, vol. 1, doi :10.1023/A:1015059928466, p. 3–52.

Blum, A. L. et R. L. Rivest. 1992, «Literate programming», *Neural Networks*, vol. 5, n° 1, p. 117–127.

Chen, L. 1993, «A global optimization algorithm for neural network training», dans *Proceedings of 1993 International Conference on Neural Networks (IJCNN-93-Nagoya, Japan)*, vol. 1, p. 443–446 vol.1, doi :10.1109/IJCNN.1993.713950.

Coelho, I., P. Munhoz, L. Ochi, M. Souza, C. Bentes et R. Farias. 2016, «An integrated cpu–gpu heuristic inspired on variable neighbourhood search for the single vehicle routing problem with deliveries and selective pickups», *International Journal of Production Research*, vol. 54, n° 4, doi :10.1080/00207543.2015.1035811, p. 945–962. URL `https://doi.org/10.1080/00207543.2015.1035811`.

Cybenko, G. 1989, «Approximation by superpositions of a sigmoidal function», *Math. Control Signal Systems*, vol. 2, p. 303–314.

Dauphin, Y. N., R. Pascanu, Ç. Gülçehre, K. Cho, S. Ganguli et Y. Bengio. 2014, «Identifying and attacking the saddle point problem in high-dimensional non-convex optimization», *CoRR*, vol. abs/1406.2572. URL http://arxiv.org/abs/1406.2572.

Du, J. 2019, «The frontier of sgd and its variants in machine learning», *Journal of Physics : Conference Series*, vol. 1229, doi :10.1088/1742-6596/1229/1/012046, p. 012 046.

El-Fallahi, A., R. Martí et L. Lasdon. 2006, «Path relinking and grg for artificial neural networks», *European Journal of Operational Research*, vol. 169, nº 2, doi :https://doi.org/10.1016/j.ejor.2004.08.012, p. 508–519, ISSN 0377-2217. URL https://www.sciencedirect.com/science/article/pii/S037722170400551X, feature Cluster on Scatter Search Methods for Optimization.

Faramarzi, A., M. Heidarinejad, S. Mirjalili et A. H. Gandomi. 2020, «Marine predators algorithm : A nature-inspired metaheuristic», *Expert Systems with Applications*, vol. 152, doi :https://doi.org/10.1016/j.eswa.2020.113377, p. 113 377, ISSN 0957-4174. URL https://www.sciencedirect.com/science/article/pii/S0957417420302025.

Gocken, M., M. Özçalici, A. Boru İpek et A. Dosdoğru. 2015, «Integrating metaheuristics and artificial neural networks for improved stock price prediction», *Expert Systems with Applications*, vol. 44, doi :10.1016/j.eswa.2015.09.029.

Goodfellow, I., Y. Bengio et A. Courville. 2016, *Deep Learning*, MIT Press. http://www.deeplearningbook.org.

Hamner, B. 2017, «Popular datasets over time», URL https://www.kaggle.com/benhamner/popular-datasets-over-time/data.

Hansen, P., N. Mladenovic, R. Todosijević et S. Hanafi. 2016, «Variable neighborhood search : basics and variants», *EURO Journal on Computational Optimization*, vol. 5, doi :10.1007/s13675-016-0075-x.

Herrán, A., J. M. Colmenar, R. Martí et A. Duarte. 2020, «A parallel variable neighborhood search approach for the obnoxious p-median problem», *International Transactions in Operational Research*, vol. 27, nº 1, doi :https://doi.org/10.1111/itor.12510, p. 336–360. URL https://onlinelibrary.wiley.com/doi/abs/10.1111/itor.12510.

Hettinger, C., T. Christensen, B. Ehlert, J. Humpherys, T. Jarvis et S. Wade. 2017, «Forward thinking : Building and training neural networks one layer at a time», .

Hussain, K., M. Salleh, S. Cheng et Y. Shi. 2019, «Metaheuristic research : a comprehensive survey», *Artificial Intelligence Review*, vol. 52, doi :10.1007/s10462-017-9605-z.

Hutter, F., J. Lücke et L. Schmidt-Thieme. 2015, «Beyond manual tuning of hyperparameters», *KI - Künstliche Intelligenz*, vol. 29, doi :10.1007/s13218-015-0381-0.

K., H. M. 2019, «Backpropagation step by step», URL https://hmkcode.com/ai/backpropagation-step-by-step/.

Kaleem, R., S. Pai et K. Pingali. 2015, «Stochastic gradient descent on gpus», *ACM International Conference Proceeding Series*, vol. 2015, doi :10.1145/2716282.2716289, p. 81–89.

Keshari, K. 2019, «Tensorflow image classification», .

Kingma, D. P. et J. Ba. 2017, «Adam : A method for stochastic optimization», .

Koehn, P. 1994, «Combining Genetic Algorithms and Neural Networks : The Encoding Problem», .

Krizhevsky, A. 2009, «Learning multiple layers of features from tiny images», .

Lourenço, H., O. Martin et T. Stützle. 2003, *Iterated Local Search*, ISBN 978-1-4020-7263-5, p. 321–353, doi :10.1007/0-306-48056-5_11.

Luong, T. V., N. Melab et E.-G. Talbi. 2013, «Gpu computing for parallel local search metaheuristic algorithms», *IEEE Transactions on Computers*, vol. 62, n° 1, doi :10.1109/TC.2011.206, p. 173–185.

Ma, Y., F. Rusu et M. Torres. 2019, «Stochastic gradient descent on modern hardware : Multi-core cpu or gpu ? synchronous or asynchronous ?», dans *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, p. 1063–1072, doi :10.1109/IPDPS.2019.00113.

Meng, X., J. Li, X. Dai et J. Dou. 2018, «Variable neighborhood search for a colored traveling salesman problem», *IEEE Transactions on Intelligent Transportation Systems*, vol. 19, n° 4, doi :10.1109/TITS.2017.2706720, p. 1018–1026.

Mishra, M. et M. Srivastava. 2014, «A view of artificial neural network», dans *2014 International Conference on Advances in Engineering Technology Research (ICAETR - 2014)*, p. 1–3, doi :10.1109/ICAETR.2014.7012785.

Mladenović, N. et P. Hansen. 1997, «Variable neighborhood search», *Computers Operations Research*, vol. 24, n° 11, doi :https://doi.org/10.1016/S0305-0548(97)00031-2, p. 1097–1100, ISSN 0305-0548. URL `https://www.sciencedirect.com/science/article/pii/S0305054897000312`.

Muñoz-Ordóñez, J., C. Cobos, M. Mendoza, E. Herrera-Viedma, F. Herrera et S. Tabik. 2018, «Framework for the training of deep neural networks in tensorflow using metaheuristics», dans *Intelligent Data Engineering and Automated Learning – IDEAL 2018*, édité par H. Yin, D. Camacho, P. Novais et A. J. Tallón-Ballesteros, Springer International Publishing, Cham, ISBN 978-3-030-03493-1, p. 801–811.

Nashed, Y., R. Ugolotti, P. Mesejo et S. Cagnoni. 2012, «libcudaoptimize : an open source library of gpu-based metaheuristics», doi :10.1145/2330784.2330803.

Pandi, R. R., S. G. Ho, S. C. Nagavarapu, T. Tripathy et J. Dauwels. 2018, «Gpu-accelerated tabu search algorithm for dial-a-ride problem», dans *2018 21st Interna-*

*tional Conference on Intelligent Transportation Systems (ITSC)*, p. 2519–2524, doi : 10.1109/ITSC.2018.8569472.

Qian, N. 1999, «On the momentum term in gradient descent learning algorithms», *Neural Networks*, vol. 12, n° 1, doi :https://doi.org/10.1016/S0893-6080(98)00116-6, p. 145–151, ISSN 0893-6080. URL https://www.sciencedirect.com/science/article/pii/S0893608098001166.

Rere, L. M. R., M. I. Fanany et A. M. Arymurthy. 2016, «Metaheuristic algorithms for convolution neural network», *Computational Intelligence and Neuroscience*, vol. 2016, doi :10.1155/2016/1537325, p. 1537 325, ISSN 1687-5265. URL https://doi.org/10.1155/2016/1537325.

Saarinen, S., R. Bramley et G. Cybenko. 1993, «Ill-conditioning in neural network training problems», *SIAM Journal on Scientific and Statistical Computing (Society for Industrial and Applied Mathematics); (United States)*, doi :10.1137/0914044, ISSN 0196-5204. URL https://www.osti.gov/biblio/6492379.

Shang, Y. et B. Wah. 1996, «Global optimization for neural network training», *Computer*, vol. 29, n° 3, doi :10.1109/2.485892, p. 45–54.

Shrestha, A. et A. Mahmood. 2019, «Review of deep learning algorithms and architectures», *IEEE Access*, vol. 7, doi :10.1109/ACCESS.2019.2912200, p. 53 040–53 065.

Sorensen, K., M. Sevaux et F. Glover. 2017, «A history of metaheuristics», *arXiv preprint arXiv :1704.00853*.

Talbi, E.-G. 2013, «A unified taxonomy of hybrid metaheuristics with mathematical programming, constraint programming and machine learning», *Studies in Computational Intelligence*, vol. 434, doi :10.1007/978-3-642-30671-6-1, p. 3–76.

Tran-Ngoc, H., S. Khatir, H. Ho-Khac, G. De Roeck, T. Bui-Tien et M. Abdel Wahab. 2021, «Efficient artificial neural networks based on a hybrid metaheuristic

optimization algorithm for damage detection in laminated composite structures», *Composite Structures*, vol. 262, doi :https://doi.org/10.1016/j.compstruct.2020.113339, p. 113 339, ISSN 0263-8223. URL https://www.sciencedirect.com/science/article/pii/S0263822320332657.

Van Luong, T. 2011, *Métaheuristiques parallèles sur GPU*, Theses, Université des Sciences et Technologie de Lille - Lille I. URL https://tel.archives-ouvertes.fr/tel-00638820, this thesis is written in English.

Werbos, P. 1974, «Beyond regression :" new tools for prediction and analysis in the behavioral sciences», *Ph. D. dissertation, Harvard University*.

Zhang, J. et S. Qu. 2021, «Optimization of backpropagation neural network under the adaptive genetic algorithm», *Complexity*, vol. 2021, doi :10.1155/2021/1718234, p. 1718 234, ISSN 1076-2787. URL https://doi.org/10.1155/2021/1718234.

# Annexe A – Résultats pour plusieurs expériences

## Résultats de référence

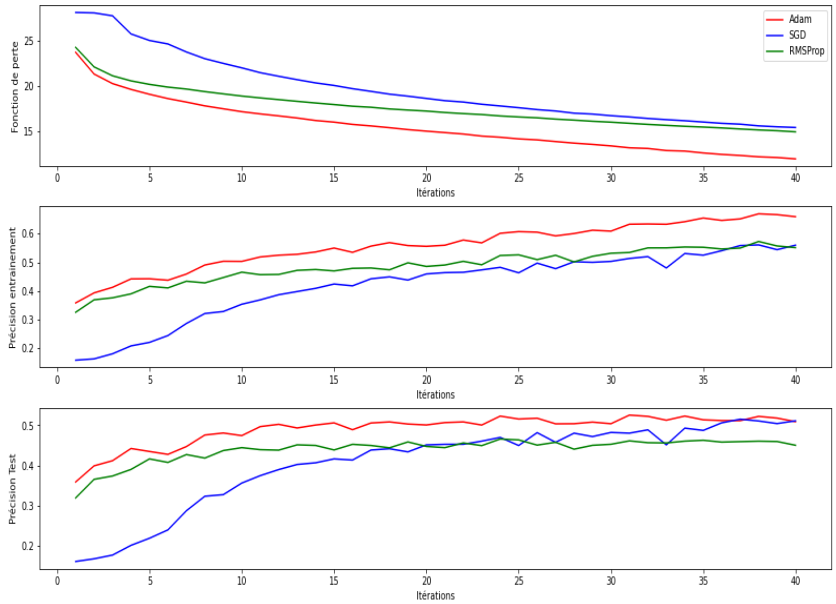



FIGURE 1 – Comparaison Adam, RMSprop, SGD pour 5 couches cachées et 40 itérations - Seed 2

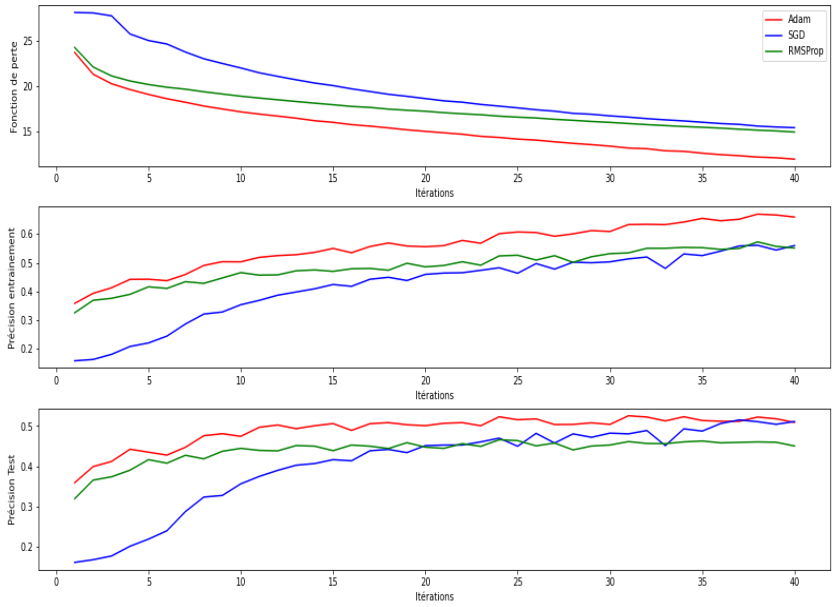



FIGURE 2 – Comparaison Adam, RMSprop, SGD pour 5 couches cachées et 40 itérations - Seed 3

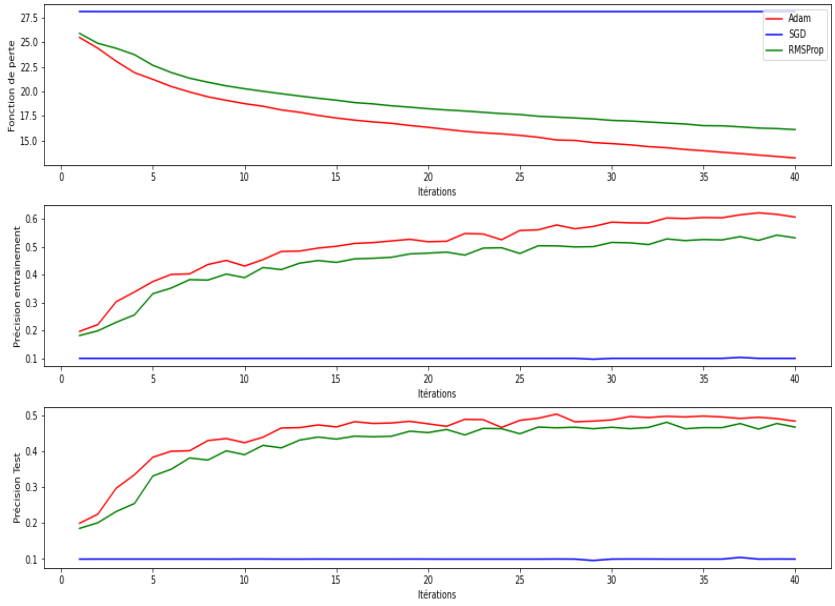



FIGURE 3 – Comparaison Adam, RMSprop, SGD pour 10 couches cachées et 40 itérations - Seed 2

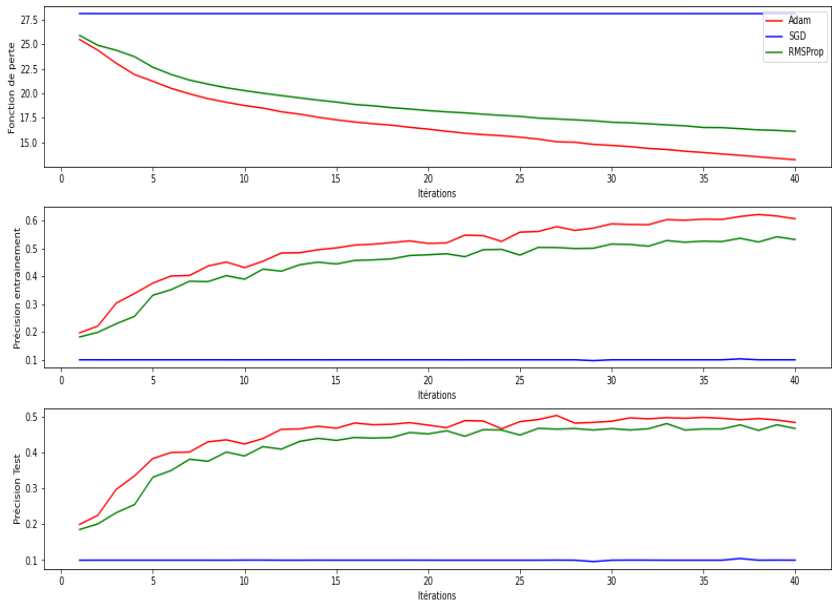



FIGURE 4 – Comparaison Adam, RMSprop, SGD pour 10 couches cachées et 40 itérations - Seed 3

# Comparaison des recherches locales avec et sans RVV

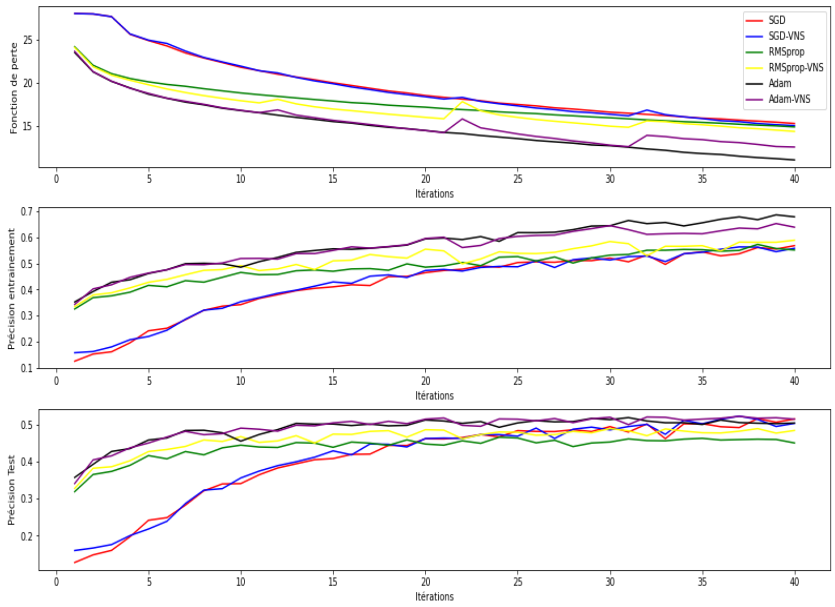



FIGURE 5 – Comparaison recherches locales avec et sans RVV pour 5 couches cachées et 40 itérations - Seed 2

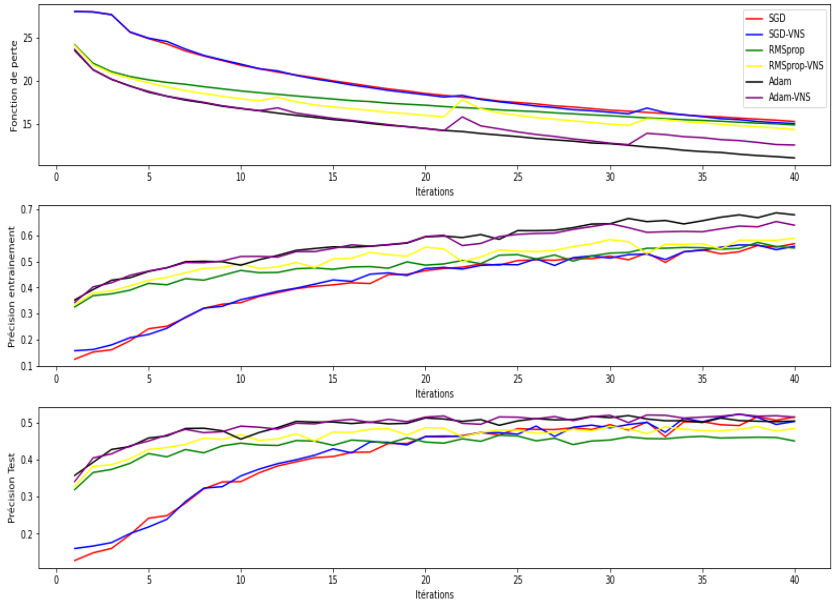



FIGURE 6 – Comparaison recherches locales avec et sans RVV pour 5 couches cachées et 40 itérations - Seed 3

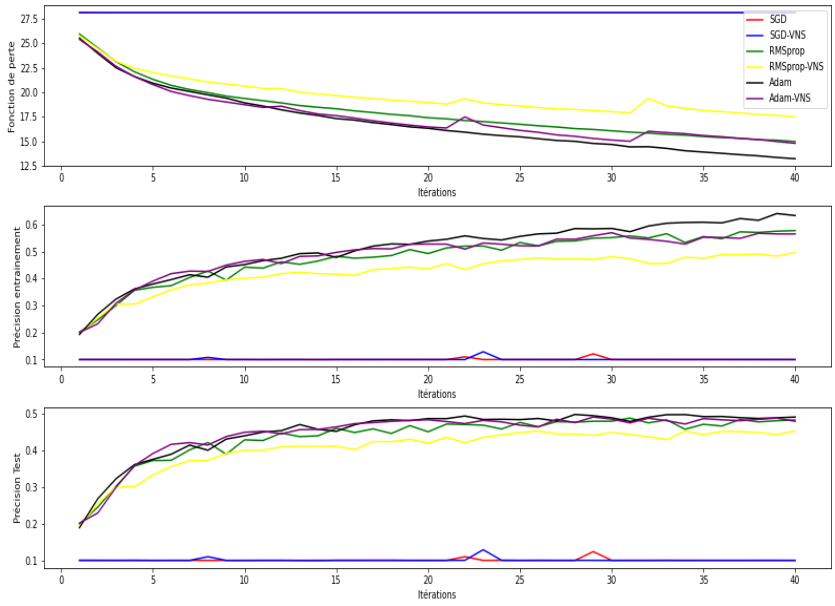



FIGURE 7 – Comparaison recherches locales avec et sans RVV pour 10 couches cachées et 40 itérations - Seed 2

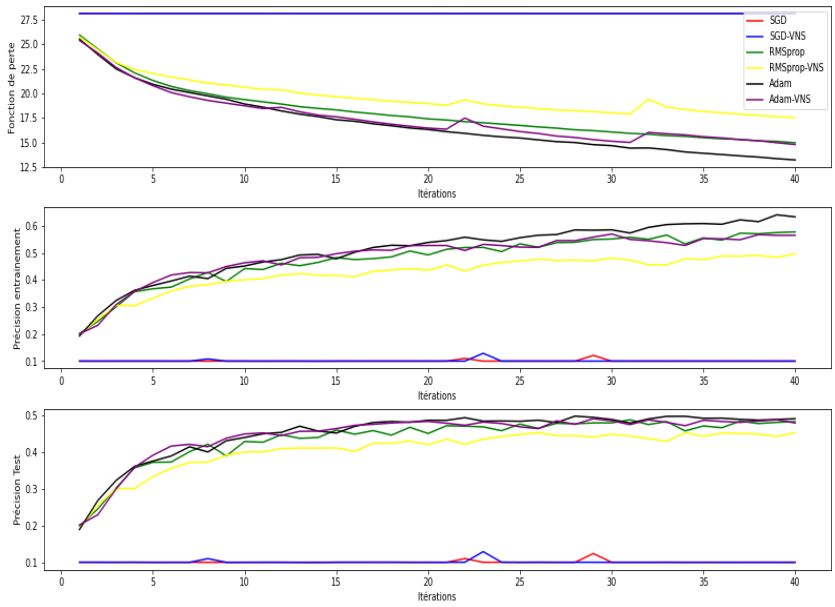



FIGURE 8 – Comparaison recherches locales avec et sans RVV pour 10 couches cachées et 40 itérations - Seed 3

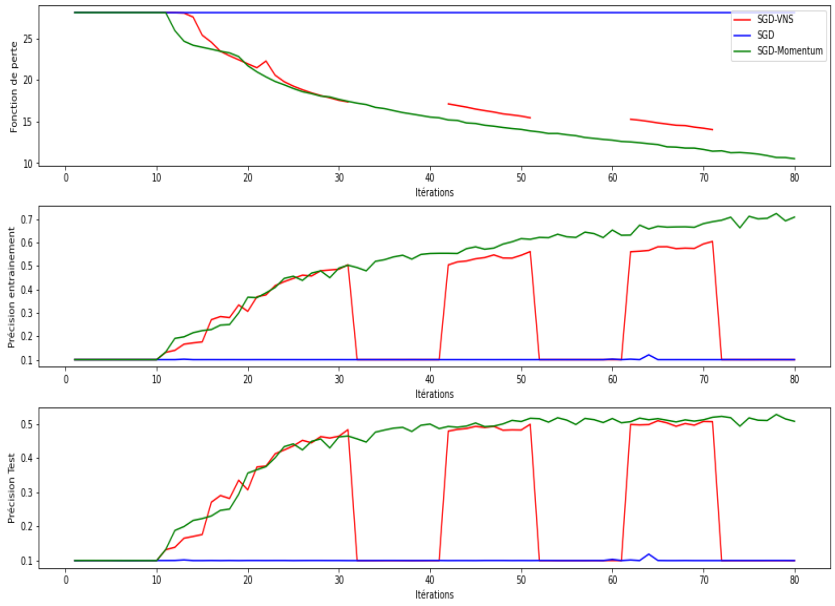



FIGURE 9 – Comparaison SGD avec et sans RVV pour 10 couches cachées et 80 itérations - Seed 2

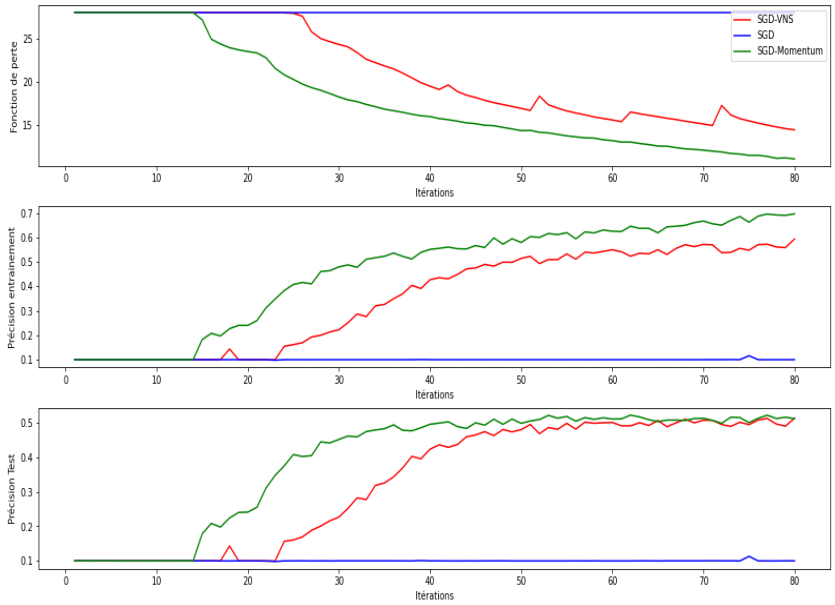



FIGURE 10 – Comparaison SGD avec et sans RVV pour 10 couches cachées et 80 itérations - Seed 3

# Résultats selon la profondeur du réseau

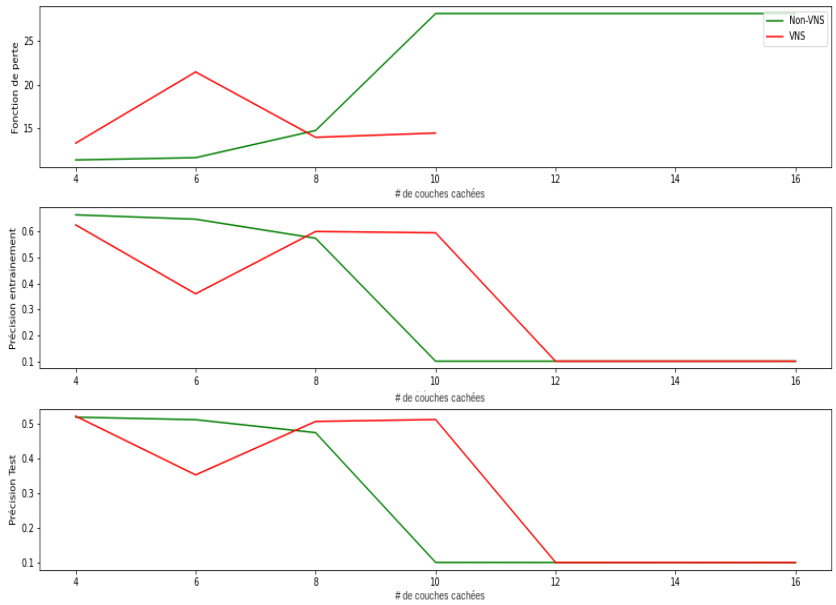



FIGURE 11 – Comparaison de la descente de gradients avec et sans RVV selon la profondeur du réseau pour 80 itérations chacun - Seed 2

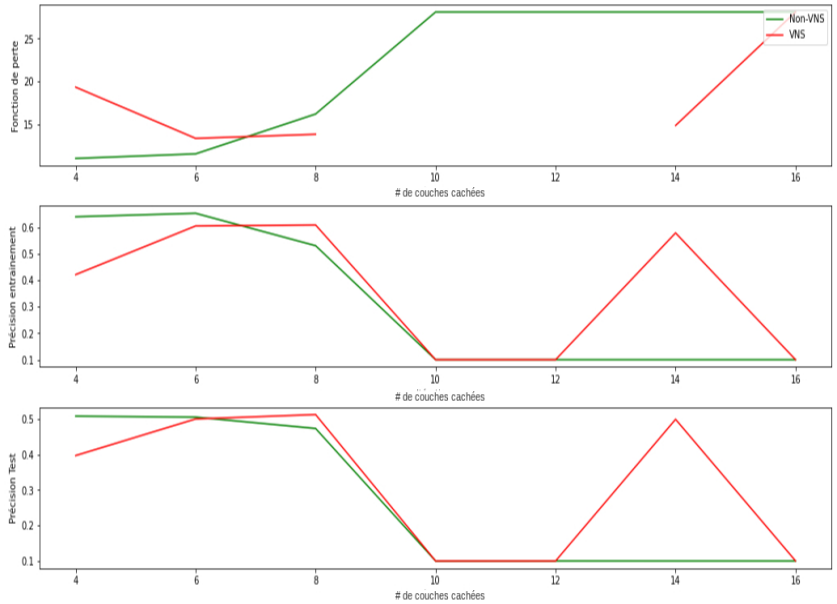



FIGURE 12 – Comparaison de la descente de gradients avec et sans RVV selon la profondeur du réseau pour 80 itérations chacun - Seed 3

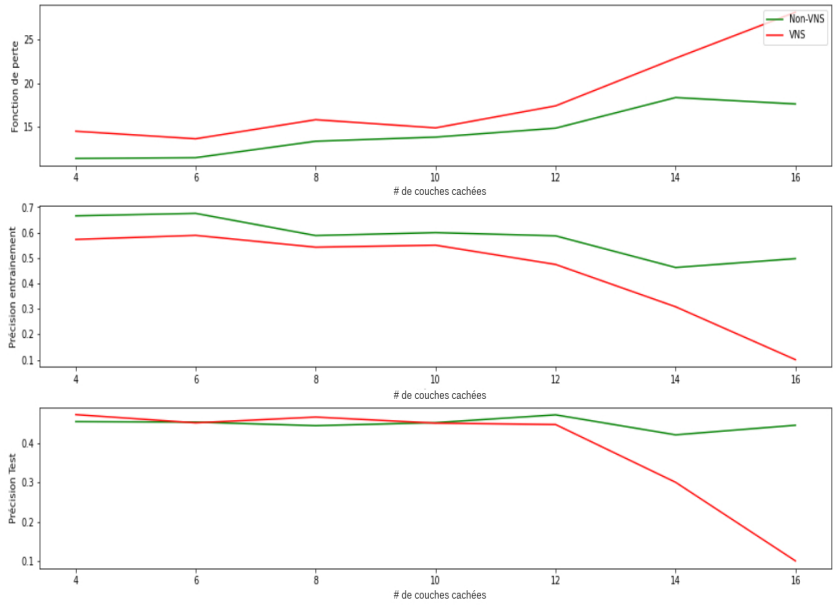



FIGURE 13 – Comparaison RMSprop avec et sans RVV selon la profondeur du réseau pour 80 itérations chacun - Seed 2

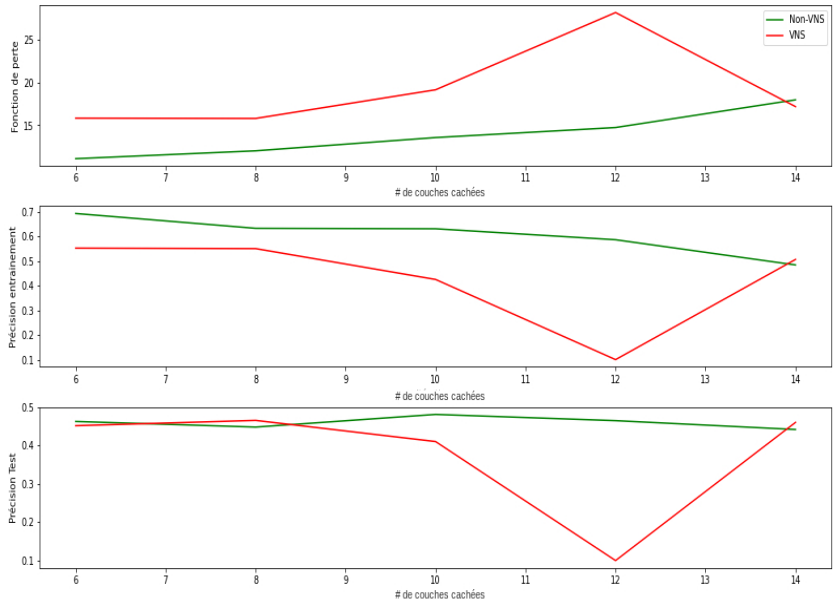



FIGURE 14 – Comparaison RMSprop avec et sans RVV selon la profondeur du réseau pour 80 itérations chacun - Seed 3

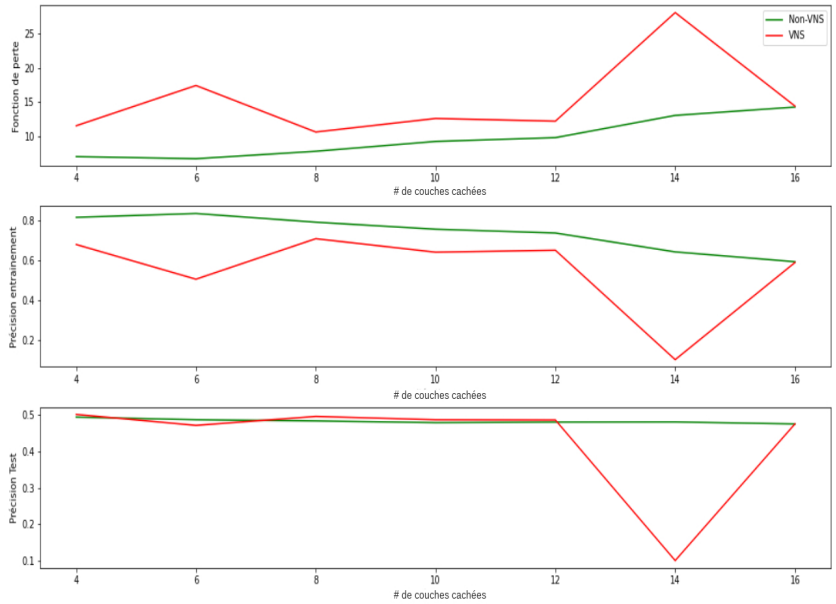



FIGURE 15 – Comparaison ADAM avec et sans RVV selon la profondeur du réseau pour 80 itérations chacun - Seed 2

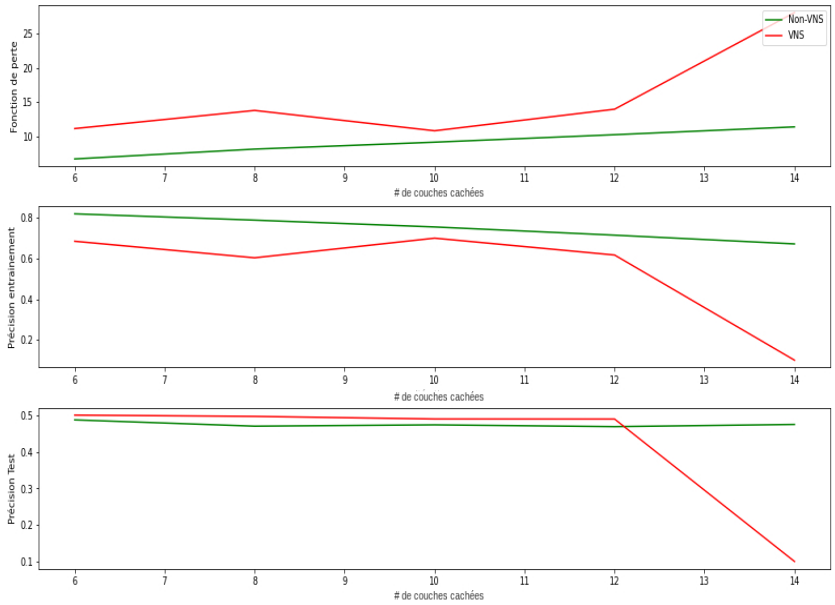



FIGURE 16 – Comparaison ADAM avec et sans RVV selon la profondeur du réseau pour 80 itérations chacun - Seed 3

# Résultats pour chaque recherche locale

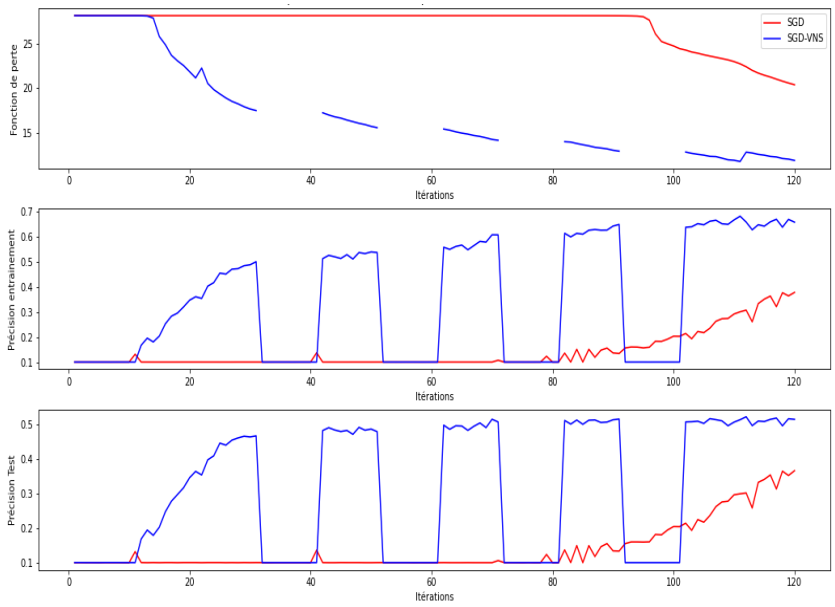



FIGURE 17 – Comparaison SGD avec et sans RVV pour 10 couches cachées et 120 itérations - Seed 2

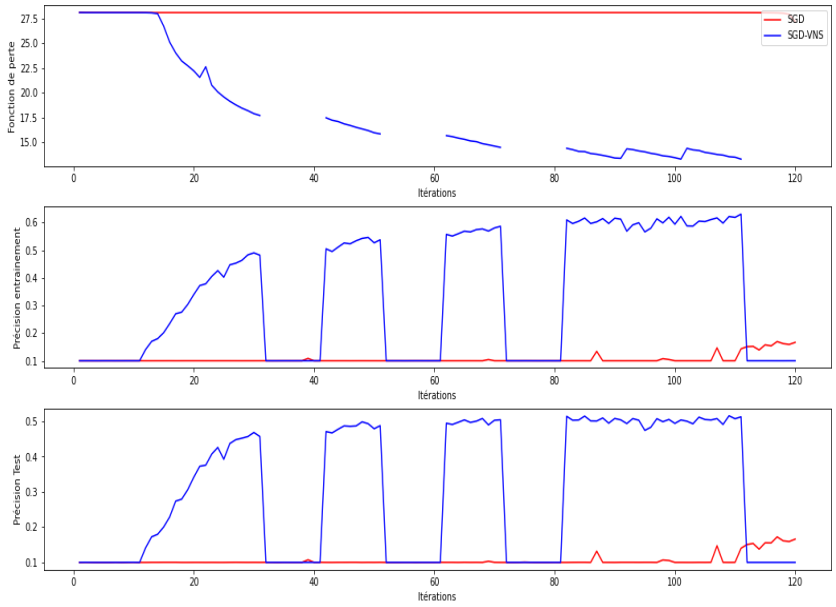



FIGURE 18 – Comparaison SGD avec et sans RVV pour 10 couches cachées et 120 itérations - Seed 3

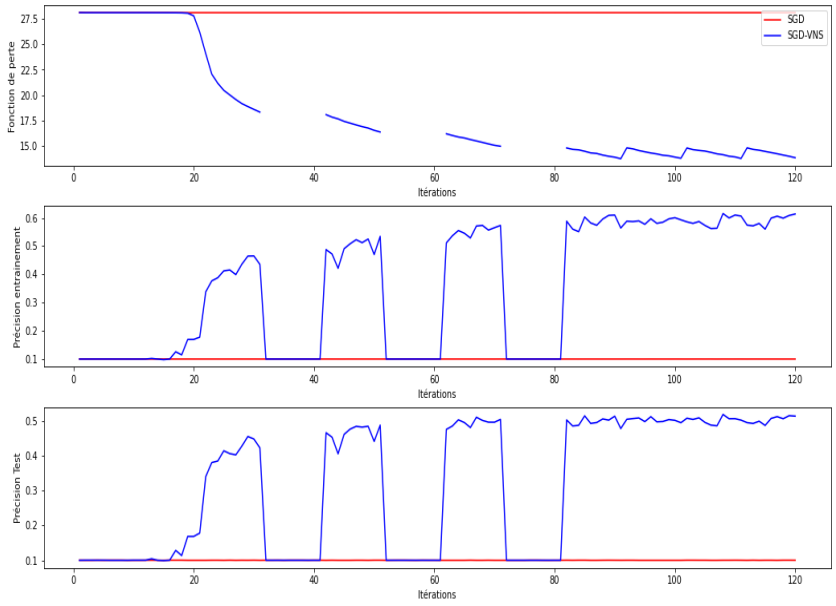



FIGURE 19 – Comparaison SGD avec et sans RVV pour 15 couches cachées et 120 itérations - Seed 2

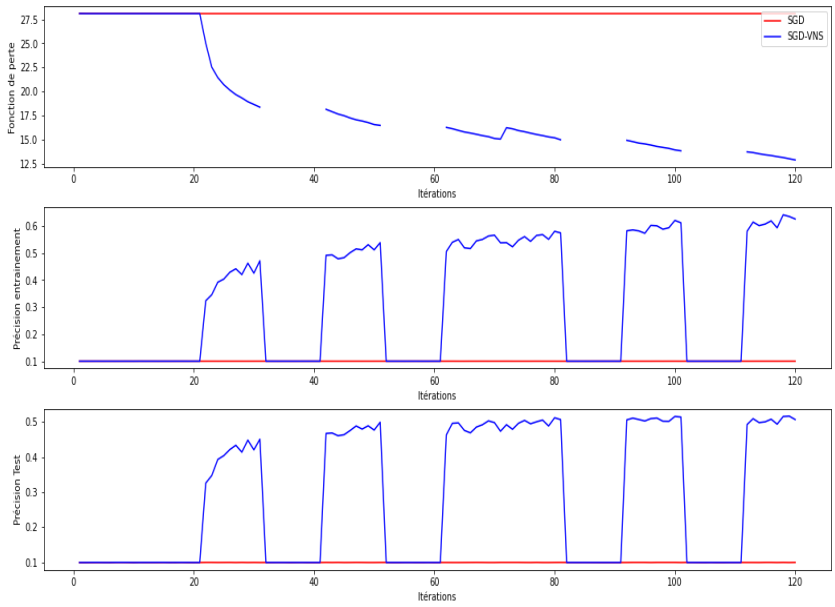



FIGURE 20 – Comparaison SGD avec et sans RVV pour 15 couches cachées et 120 itérations - Seed 3

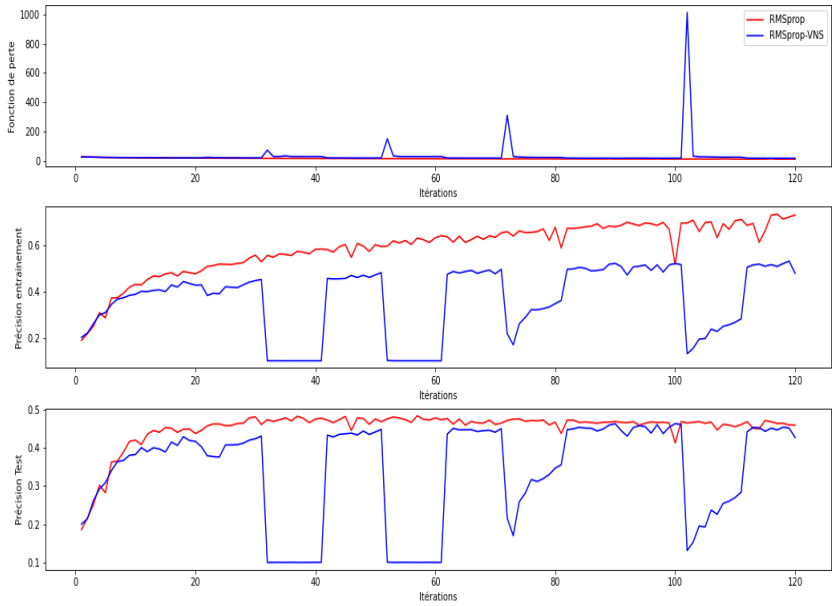



FIGURE 21 – Comparaison RMSprop avec et sans RVV pour 10 couches cachées et 120 itérations - Seed 2

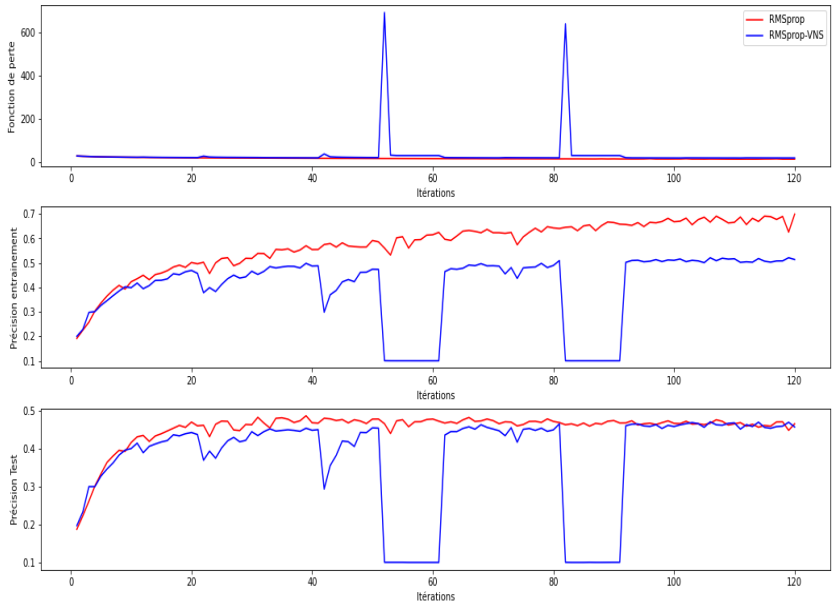



FIGURE 22 – Comparaison RMSprop avec et sans RVV pour 10 couches cachées et 120 itérations - Seed 3

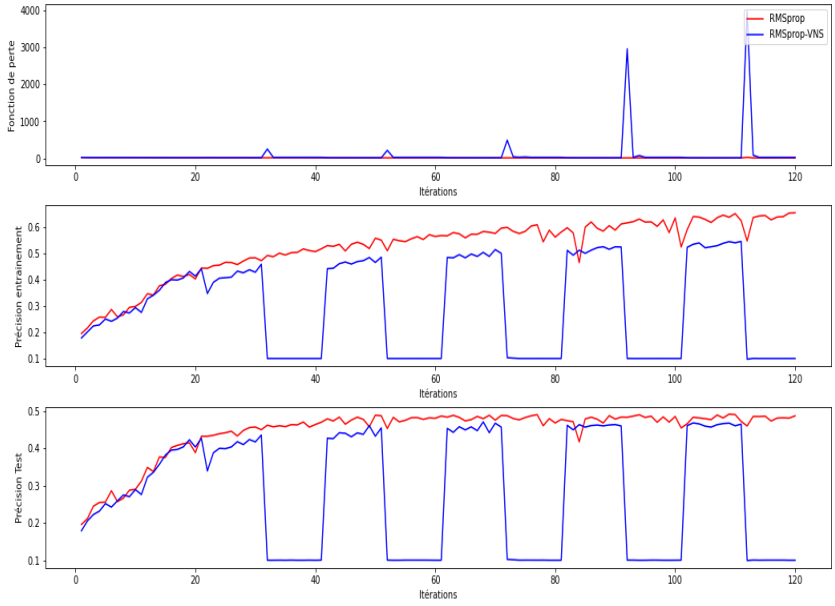



FIGURE 23 – Comparaison RMSprop avec et sans RVV pour 15 couches cachées et 120 itérations - Seed 2

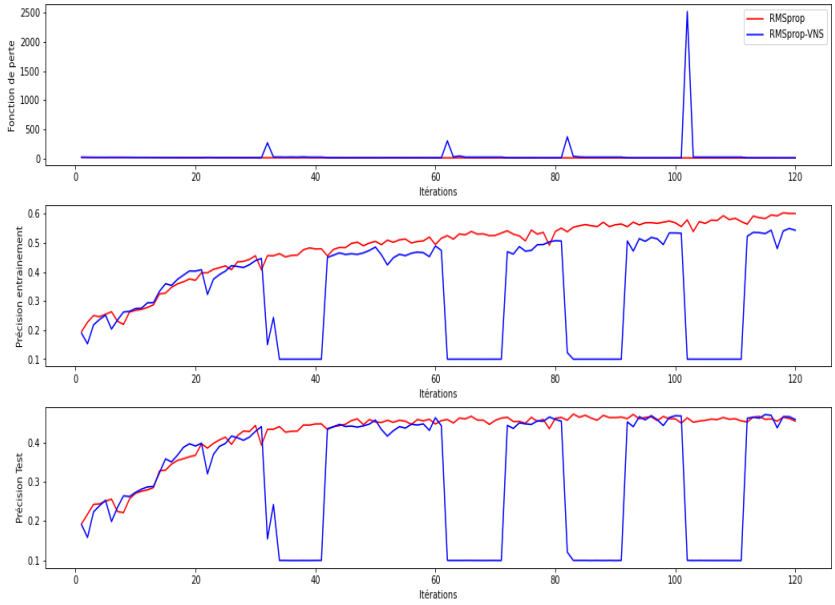



FIGURE 24 – Comparaison RMSprop avec et sans RVV pour 15 couches cachées et 120 itérations - Seed 3

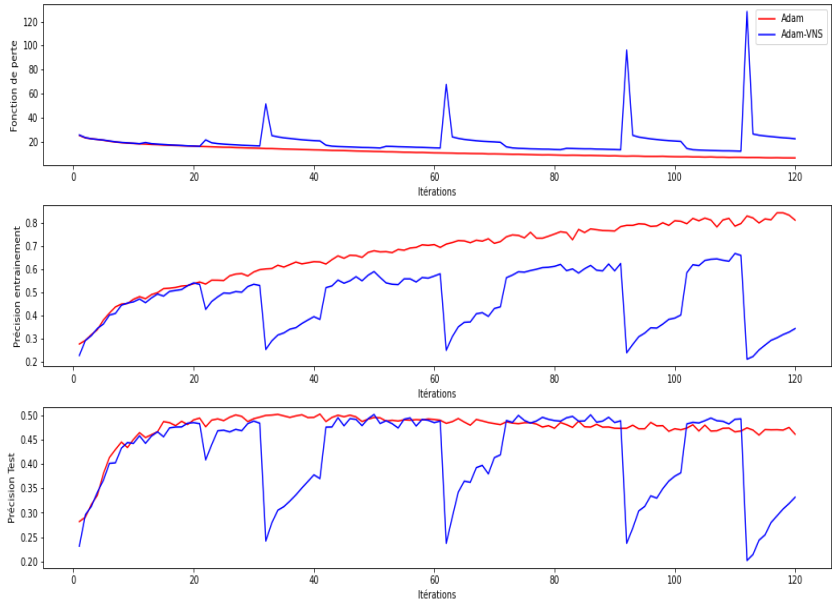



FIGURE 25 – Comparaison ADAM avec et sans RVV pour 10 couches cachées et 120 itérations - Seed 2

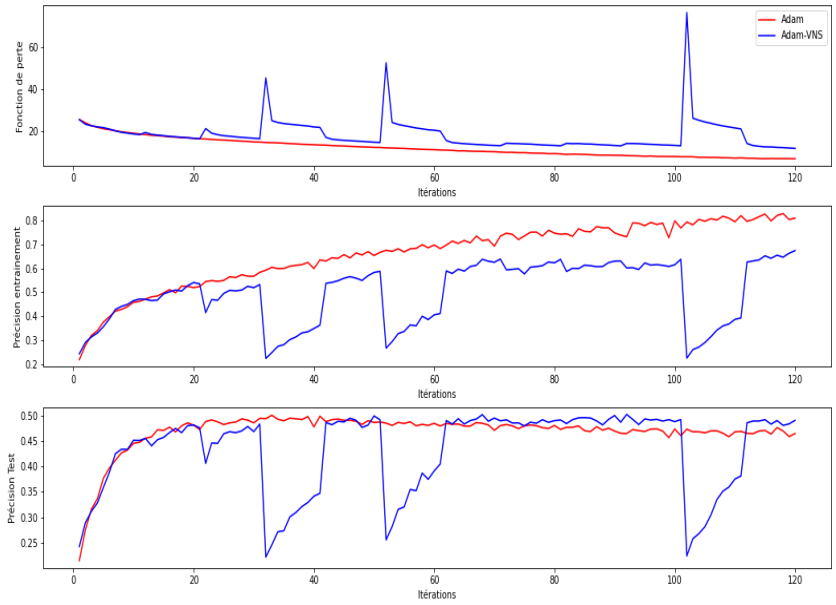



FIGURE 26 – Comparaison ADAM avec et sans RVV pour 10 couches cachées et 120 itérations - Seed 3

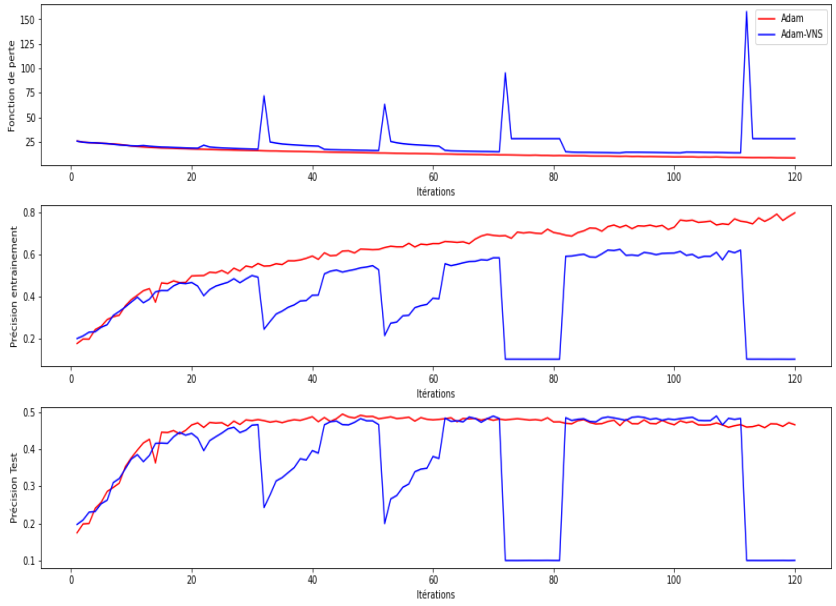



FIGURE 27 – Comparaison ADAM avec et sans RVV pour 15 couches cachées et 120 itérations - Seed 2

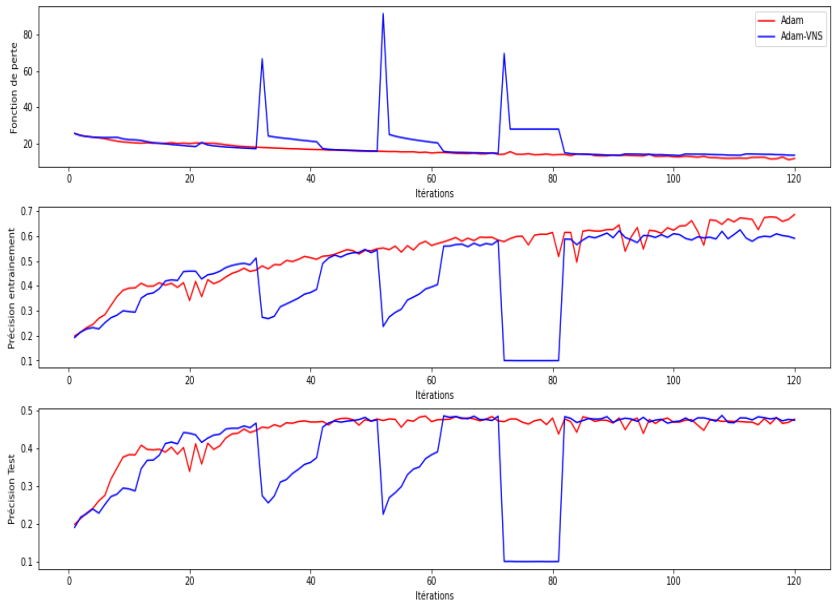



FIGURE 28 – Comparaison ADAM avec et sans RVV pour 15 couches cachées et 120 itérations - Seed 3

# Annexe B – Présentation de l'algorithme de rétropropagation

La complexité en temps d'une itération de l'entraînement d'un réseau de neurones est dominé par les multiplications de matrices nécessaires à l'algorithme de rétropropagation (Voir Annexe 2 pour les détails de calculs). Par exemple, la complexité en temps d'une multiplication de matrices $M_{ij} * M_{jk}$ serait $\mathcal{O}(i * j * k)$.

Soit $i$ le nombre de neurones dans la couche d'entrée d'un réseau, $j$ le nombre de neurones dans une couche cachée, $l$ le nombre de neurones dans la couche de sortie et $t$ le nombre d'observations de données considérées par itération (soit la taille du 'batch'). Dans les résultats présentés ci-dessus toutes les couches cachées font la meme taille, $n$ correspond alors simplement au nombre de couches cachées contenues dans le réseau. Ainsi la complexité en temps de l'algorithme de rétropropagation sera donné par $\mathcal{O}(t * (ij * 2nj * jl)$.
L'algorithme proposé ne nécessite à l'inverse que d'extraire les valeurs contenues dans une couche cachée du réseau, soit ici $j$ et ce toutes les 10 itérations. De cette façon, l'ajout de la RVV ne pourrait avoir aucune influence sur la complexité en temps de l'algorithme général. Cela est également confirmé empiriquement, l'obtention des résultats présentés plus tôt présentaient des temps de calcul très similaires qu'ils considèrent l'algorithme proposé ou non.

L'exemple suivant présente dans plus de détails les multiplications matricielles et autres calculs nécessaires à la mise à jour des poids du réseau.

Cet exemple est inspiré de la visualisation de l'algorithme de rétropropagation étape par étape présentée pour la libraire *netflow.js* (K., 2019) mais pour des paramètres différents.

Afin de simplifier cet exemple, un simple réseau contenant deux neurones en entrée, deux neurones dans une couche cachée et un seul neurone en sortie sera ajusté.

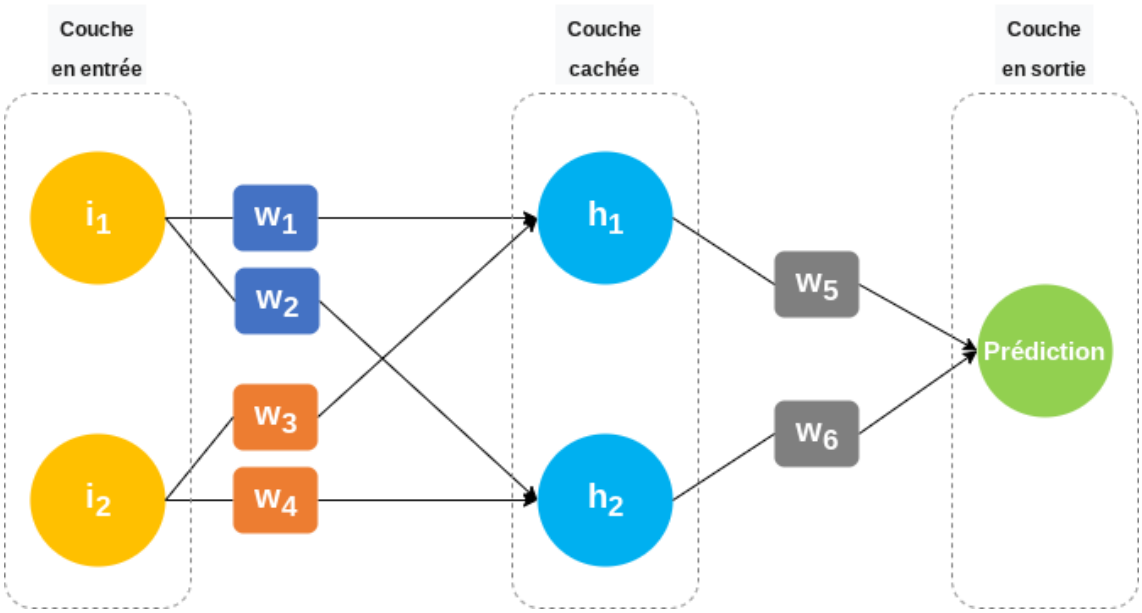



FIGURE 29 – Notation structurée d'un réseau simple

Les données utilisées pour cet exemple seront une seule observation pour laquelle $i_1 = 6, i_2 = 2$ et la valeur attendue est de 3. Les poids initiaux du réseau seront sélectionnés aléatoirement ce qui permet d'obtenir :

L'on peut ensuite remplir le reste du réseau de cette façon :

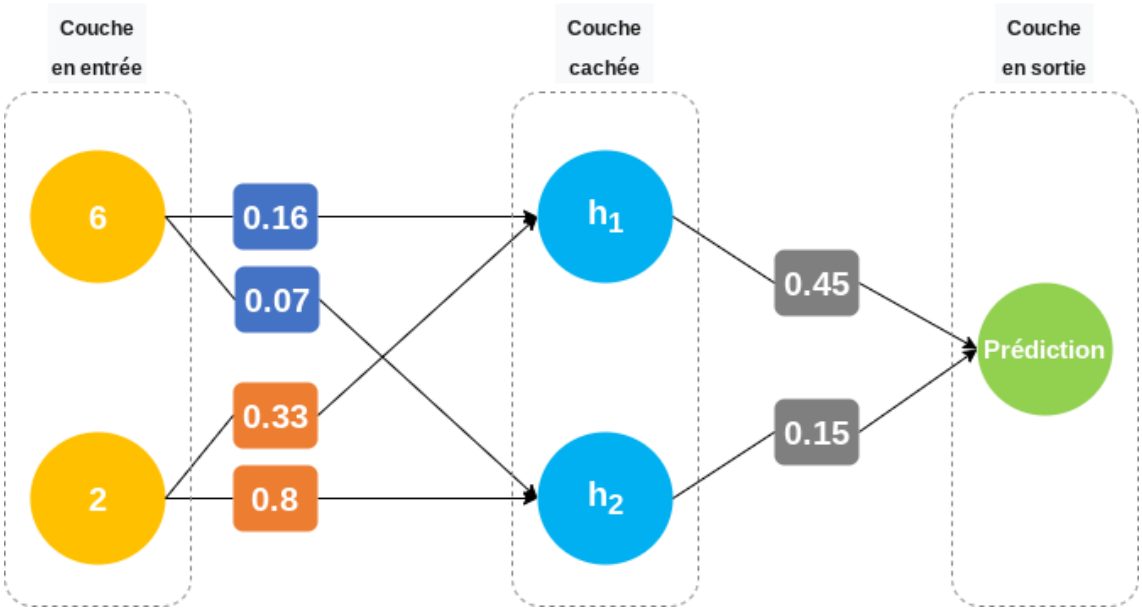



FIGURE 30 – Initialisation des poids du réseau

$$\begin{bmatrix} 6 & 2 \end{bmatrix} * \begin{bmatrix} 0.16 & 0.07 \\ 0.33 & 0.8 \end{bmatrix} = \begin{bmatrix} 1.62 & 2.02 \end{bmatrix}$$

$$\begin{bmatrix} 1.62 & 2.02 \end{bmatrix} * \begin{bmatrix} 0.45 & 0.2 \end{bmatrix} = 1.133$$

Soit :

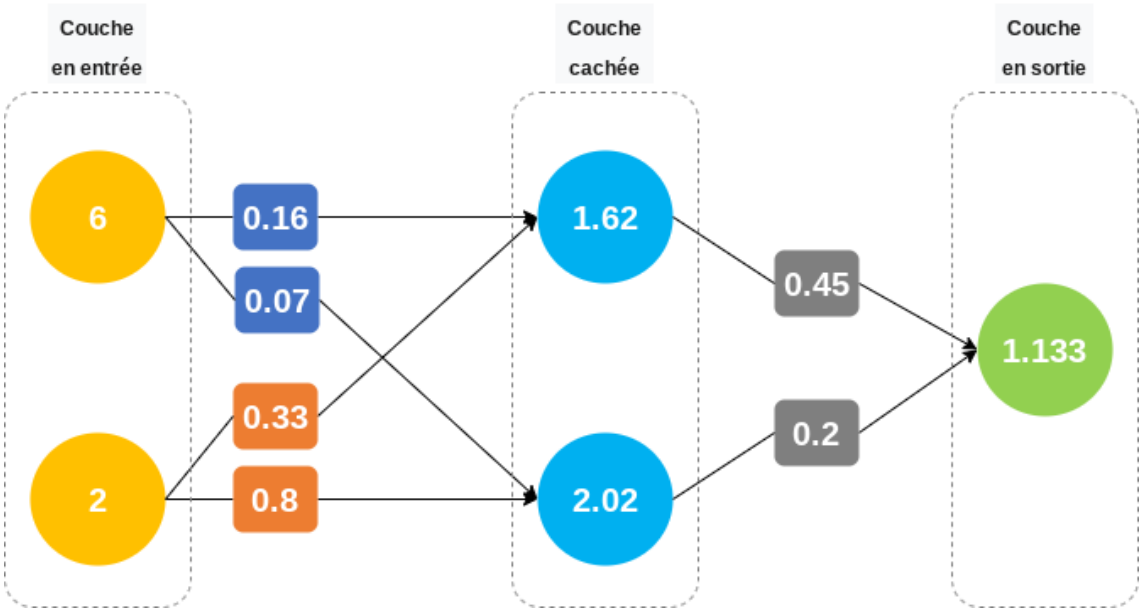



FIGURE 31 – Réseau complété après une première propagation des données en entrée

Ces calculs peuvent également être formulés ainsi :

$$Prédiction = h_1 \times w_5 + h_2 \times w_6$$

$$\Downarrow$$

$$Prédiction = (i_1 w_1 + i_2 w_2) w_5 + (i_1 w_3 + i_2 w_4) w_6$$

La prédiction initiale donnée par le réseau est donc de $1.133$.

Pour cet exemple, la fonction d'erreur sera simplement l'erreur moyenne au carré qui sera ici calculée ainsi :

$Erreur = \frac{1}{2}(1.133 - 3)^2 = 1.743$

La mise à jour des poids cherchera donc à minimiser cette valeur.

Afin d'ajuster ces poids en fonction de ce premier résultat obtenu, il est nécessaire de reprendre la règle de mise à jour basé sur le calcul des gradients présentée au chapitre 1 :

$$w_i' = w_i - \alpha \frac{\partial Erreur}{\partial w_i}$$

L'exemple suivant présente le calcul de la dérivée partielle de la fonction de perte pour le poids $w_5$.

$$\frac{\partial Erreur}{\partial w_5} = \frac{\partial Erreur}{\partial Prédiction} \times \frac{\partial Prdiction}{\partial w_5}$$

$$\frac{\partial Erreur}{\partial w_5} = \frac{1}{2} \frac{\partial (Prédiction - Vérité)^2}{\partial Prédiction} \frac{\partial (i_1 w_1 + i_2 w_2) w_5 + (i_1 w_3 + i_2 w_4) w_6}{\partial w_5}$$
$$\frac{\partial Erreur}{\partial w_5} = 2 * \frac{1}{2} (Prédiction - Vérité) * (i_1 w_1 + i_2 w_2)$$
$$\frac{\partial Erreur}{\partial w_5} = 2 * \frac{1}{2} (Prédiction - Vérité) * (h_1)$$
$$\frac{\partial Erreur}{\partial w_5} = \Delta(h_1)$$

Ce qui nous permet de compléter l'équation de mise à jour des poids :

$$w_5' = w_5 - \alpha \Delta(h_1)$$

L'on peut également en déduire l'équation de mise à jour pour le second neurone contenue dans la même couche cachée :

$$w_6' = w_6 - \alpha \Delta(h_2)$$

Pour autant la mise des poids contenus dans la première couche du réseau devra considérer les mises à jour de la dernière couche du réseau. Dans le cas du poids $w_2$ et considérant les règles de dérivation en chaine, sa mise à jour sera :

$$\frac{\partial Erreur}{\partial w_2} = \frac{\partial Erreur}{\partial Prédiction} * \frac{\partial Prdiction}{\partial h_1} * \frac{\partial h_1}{\partial w_2}$$

L'on sait que : $h_1 = (i_1 w_1 + i_2 w_2)$

$$\frac{\partial Erreur}{\partial w_2} = \frac{1}{2}\frac{\partial (Prédiction - Vérité)^2}{\partial Prédiction}\frac{\partial (i_1 w_1 + i_2 w_2) w_5 + (i_1 w_3 + i_2 w_4) w_6}{\partial h_1} * \frac{\partial (i_1 w_1 + i_2 w_2)}{\partial w_2}$$
$$\frac{\partial Erreur}{\partial w_2} = 2 * \frac{1}{2}(Prédiction - Vérité) * (w_5) * (i_2)$$
$$\frac{\partial Erreur}{\partial w_2} = \Delta(w_5 * i_2)$$

Comme précédemment, l'on peut en déduire les mises à jour pour le reste des poids de la couche concernée :

$$w'_6 = w_6 - \alpha(\Delta h_2)$$
$$w'_5 = w_5 - \alpha(\Delta h_1)$$
$$w'_4 = w_4 - \alpha(i_2 * \Delta w_6)$$
$$w'_3 = w_3 - \alpha(i_1 * \Delta w_6)$$
$$w'_2 = w_2 - \alpha(i_2 * \Delta w_5)$$
$$w'_1 = w_1 - \alpha(i_1 * \Delta w_5)$$

Ces calculs sont généralement exprimés sous forme matricielle afin de faciliter leur implémentation informatique :

$$\begin{bmatrix} w'_5 \\ w'_6 \end{bmatrix} = \begin{bmatrix} w_5 \\ w_6 \end{bmatrix} - \alpha\Delta \begin{bmatrix} h1 \\ h2 \end{bmatrix}$$
$$\begin{bmatrix} w_1 & w_3 \\ w_2 & w_4 \end{bmatrix} = \begin{bmatrix} w_1 & w_3 \\ w_2 & w_4 \end{bmatrix} - \alpha\Delta \begin{bmatrix} i_1 \\ i_2 \end{bmatrix} * \begin{bmatrix} w_5 & w_6 \end{bmatrix}$$

## Exemple de mise à jour des poids

Si l'on reprend les valeurs contenues dans le réseau de base :

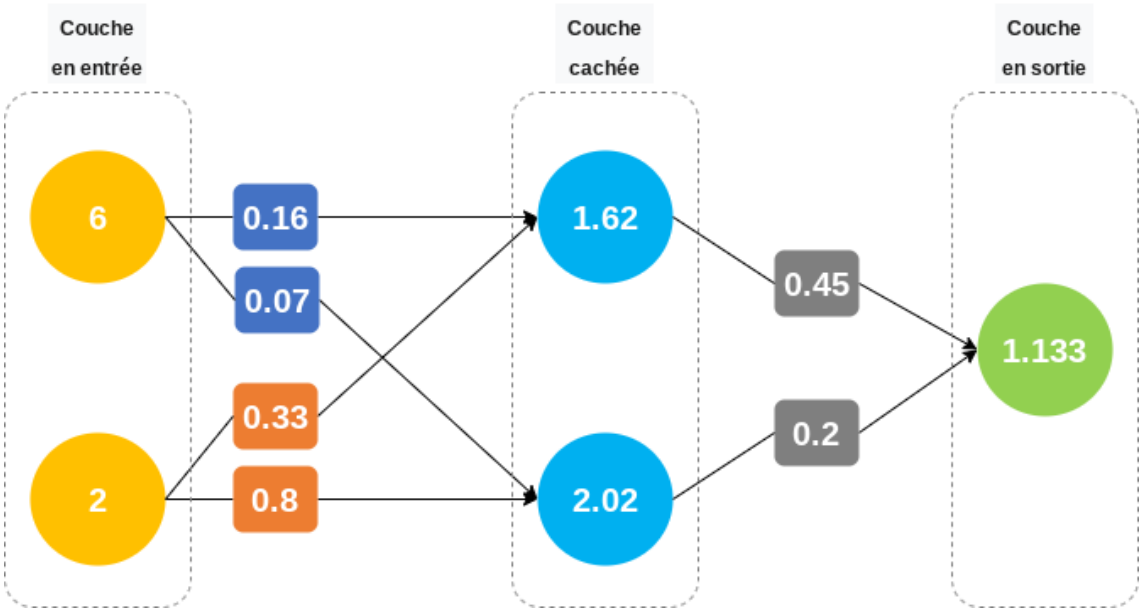



FIGURE 32 – Réseau complété après une première propagation des données en entrée

Si l'on considère pour cet exemple que $\alpha = 0.01$ et que l'on sait que $\Delta = (1.133 - 3) = -1.867$

En intégrant ces valeurs dans dernier calcul de la section précédente alors :

$$\begin{bmatrix} w_5 \\ w_6 \end{bmatrix} = \begin{bmatrix} 0.45 \\ 0.2 \end{bmatrix} - 0.01(-1.867) * \begin{bmatrix} 1.62 \\ 2.02 \end{bmatrix} = \begin{bmatrix} 0.48 \\ 0.24 \end{bmatrix}$$

$$\begin{bmatrix} w_1 & w_3 \\ w_2 & w_4 \end{bmatrix} = \begin{bmatrix} 0.16 & 0.33 \\ 0.07 & 0.8 \end{bmatrix} - 0.01(-1.867) * \begin{bmatrix} 6 \\ 2 \end{bmatrix} * \begin{bmatrix} 0.45 & 0.2 \end{bmatrix}$$
$$= \begin{bmatrix} 0.16 & 0.33 \\ 0.07 & 0.8 \end{bmatrix} - \begin{bmatrix} 0.051 & 0.023 \\ 0.017 & 0.008 \end{bmatrix} = \begin{bmatrix} 0.21 & 0.35 \\ 0.09 & 0.81 \end{bmatrix}$$

L'on obtient alors ce nouveau réseau, la modification des poids implique qu'il atteigne une nouvelle prédiction :

$$\begin{bmatrix} 6 & 2 \end{bmatrix} * \begin{bmatrix} 0.21 & 0.09 \\ 0.35 & 0.81 \end{bmatrix} = \begin{bmatrix} 1.96 & 2.16 \end{bmatrix}$$

$$\begin{bmatrix} 1.96 & 2.16 \end{bmatrix} * \begin{bmatrix} 0.48 & 0.24 \end{bmatrix} = 1.459$$

Soit :

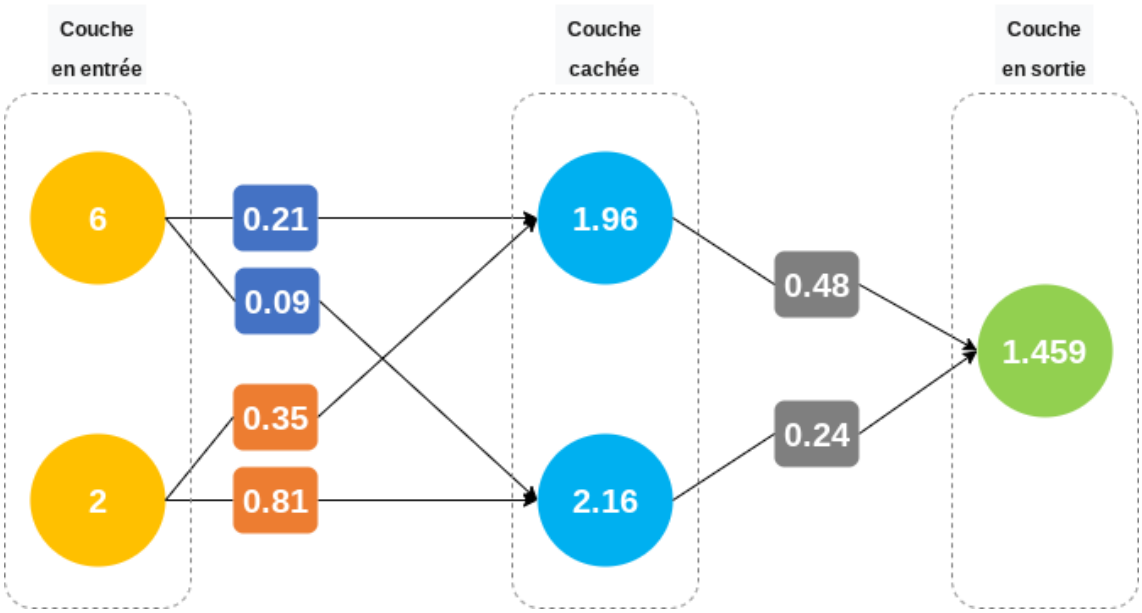



FIGURE 33 – Réseau complété après la rétropropagation des erreurs

Il apparaît donc que la nouvelle valeur obtenue après une itération, 1.459, est plus proche de la valeur actuelle contenue dans le jeu de donnée, 3, que l'ancienne valeur qui était de 1.133. Ce déplacement minimise donc l'erreur obtenue par le réseau.
Dans ce cas, une valeur du paramètre $\alpha$ plus grande aurait accrut l'importance de la mise à jour des poids et aurait pu mener à un résultat plus proche de la valeur attendue. Une valeur de $\alpha$ peut donc réduire le nombre d'itérations nécessaire au risque de 'dépasser' la valeur attendue.