Algorithmes de Descente de Gradient Stochastique avec le filtrage des paramètres pour l'entraînement des réseaux à convolution profonds

Pierre Gillot[1] Akka Zemmari[1] Jenny Benois-Pineau[1] Yurii Nesterov[2]

[1] Univ. Bordeaux, Bordeaux INP, CNRS, LaBRI, UMR5800, F-33400 Talence, France
[2] CORE, Catholic University of Louvain (UCL), Belgium

Résumé

Dans cet article, nous nous intéressons à la définition de nouveaux algorithmes d'apprentissage profond afin d'améliorer la qualité et la stabilité de l'entrainement des paramètres des réseaux de neurones à convolution. Notre contribution se présente sous forme de deux algorithmes, basés tous les deux sur le filtrage des paramètres du réseau de neurones. Les expériences menées montrent qu'un réglage fin de leurs (hyper) paramètres améliore clairement la qualité et la stabilité des résultats tant en phase d'entrainement qu'en phase de validation.

Mots Clef

Apprentissage profond, optimisation, descente de gradient.

1 Introduction

Les **réseaux de neurones artificiels** (**ANN**) sont des algorithmes inspirés de la biologie, capables *d'apprendre* à réaliser une tâche de classification. Un réseau de neurones est composé d'un ensemble de couches successives. Chaque couche est composée des neurones artificiels [1]. Un neurone de la $k^{\text{ième}}$ couche est connecté à tous les neurones de la $(k-1)^{\text{ième}}$. Entrainer un réseau revient alors à apprendre les poids synaptiques et les biais de chaque neurone.

Les réseaux de neurones présentent un problème de passage à l'échelle lorsqu'ils sont utilisés pour la classification d'ensembles importants de données en particulier d'images. Leur architecture de base, où l'entrée de chaque neurone d'une couche est connectée à toutes les sorties des neurones de la couche précédente, rend l'apprentissage du nombre trop grand de paramètres non envisageable.

Les **réseaux de neurones à convolution** (**CNN**) ont été conçus pour la classification d'images. Leur architecture est différente de celle des réseaux ANN. Les couches ne sont pas toutes de même nature : couches de convolution, transformation non linéaire, couches de *pooling*, etc. Les neurones ne sont plus connectés à tous les neurones de couches précédentes, ... Dans la plupart d'architectures des CNN profonds, l'apprentissage des paramètres se fait par l'algorithme de *descente de gradient stochastique* (SGD) [1].

Les méthodes de descente de gradient sont efficaces pour les problèmes convexes. Cependant, la fonction de perte (que l'on cherche à minimiser dans la phase d'apprentissage) n'est pas toujours convexe. Par expérience, on remarque qu'il est nécessaire d'effectuer un grand nombre d'itérations avec une grande instabilité de cette fonction. Dans cet article, nous explorons les méthodes d'optimisation basées sur la SGD. Nous proposons des méthodes d'optimisation (*moyenne pondérée* et *moyenne glissante pondérée*) permettant de stabiliser le processus d'entrainement. Nous étudions ces méthodes de manière expérimentale et ce sur un corpus de données image de référence, le corpus *MNIST* [2].

2 Travaux antérieurs

2.1 Apprentissage et optimisation dans les réseaux profonds

Descente de gradient. La méthode de descente de gradient a fait ses preuves pour l'optimisation des paramètres d'un réseau de neurones. Elle est ainsi l'une des approches les plus efficaces et les plus utilisées. Nous rappelons brièvement quelques notions sur cette méthode. Plus de détails peuvent être trouvés dans [1].

Soit θ le vecteur contenant tous les paramètres de notre réseau CNN. Soit $J\big(\theta, g(x), y\big)$ la fonction de coût représentant l'erreur entre la vérité terrain $g(x)$ (liée à la donnée d'entrainement x) et la valeur prédite y estimée par le réseau en utilisant les paramètres θ.

Un pas de l'itération de la descente de gradient est donné par :

$$\theta_{t+1} \leftarrow \theta_t - \eta \nabla_\theta J\big(\theta, g(x), y\big), \tag{1}$$

où η est une constante positive appelée *taux d'apprentissage*. La méthode s'applique pour tout espace de dimension quelconque (même infinie) et peut être utilisée aussi bien pour les fonctions linéaires que pour les non linéaires. L'algorithme ne converge vers la solution globale que si la fonction J est strictement convexe dans le domaine d'optimisation des paramètres.

En apprentissage profond, le nombre de paramètres à apprendre devient très vite important. Les arguments de la fonction de coût sont alors de très grande dimension, on

observe alors une prolifération de points-selles, lesquels peuvent être très pénalisant quand on cherche un minimum car ils sont, en général, entourés de plateaux [3].
D'autre part, le choix du taux d'apprentissage est crucial : il doit être suffisamment petit pour "assurer" la convergence de l'algorithme mais également pas trop petit pour ne pas trop retarder le temps de convergence du processus d'optimisation. Le choix d'un taux par itération peut alors améliorer considérablement les résultats. L'algorithme dit *line search* [4] est une solution classique à ce problème.

2.2 Descente de gradient stochastique.

En apprentissage profond, la fonction objective que l'on cherche à minimiser est souvent non convexe et non régulière. La convergence de la descente du gradient vers le minimum global n'est donc pas garantie et la convergence même vers un minimum local peut être extrêmement lente. Une solution à ce problème consiste en l'utilisation de l'algorithme de descente de gradient stochastique. L'idée de l'approche est de chercher à minimiser une fonction qui peut être écrite sous la forme de la somme de fonctions différentiables. Ce processus est alors réalisé de manière itérative sur des lots de données tirés aléatoirement. Chaque fonction objective minimisée de cette manière est une approximation de la fonction objective globale. L'équation suivante décrit cette méthode :

$$\theta_{t+1} \leftarrow \theta_t - \eta \frac{1}{B_s} \sum_{i=1}^{B_s} \nabla_\theta J_t^i(\theta_t, x_t^i, y_t^i), \tag{2}$$

où B_s est la taille du lot, $B_t = (x_t^i, y_t^i)_{i \in [|1, B_s|]}$ est le lot de données tiré à l'étape t, x_t^i et y_t^i sont respectivement la donnée de la vérité terrain et la donnée estimée dans ce lot, et $J_t = \frac{1}{B_s} \sum_{i=1}^{B_s} J_t^i$ est l'approximation stochastique de la fonction de coût globale à l'étape t sur le lot B_t, décomposée en somme de fonctions différentiables J_t^i liées à chaque paire (x_t^i, y_t^i). La Descente de gradient stochastique (SGD) résout la plupart des problèmes rencontrés en apprentissage profond.

Momentum SGD. L'objectif principal de la méthode dite de *Momentum* est d'accélérer le processus de descente de gradient, et ceci en rajoutant un vecteur de vélocité à l'expression initiale :

$$\begin{aligned} v_{t+1} &\leftarrow \mu v_t - \eta \nabla J(\theta_t), \\ \theta_{t+1} &\leftarrow v_{t+1} + \theta_t. \end{aligned} \tag{3}$$

Le vecteur v_{t+1} est calculé au début de chaque itération et représente la mise à jour de la vélocité d'une "balle dévalant une pente".
La vélocité s'accumule à chaque itération d'où l'introduction d'un hyper-paramètre μ permettant d'amortir la vélocité quand on atteint une surface plate. Une bonne stratégie peut être de modifier μ en fonction du niveau d'apprentissage.

Descente de gradient accélérée de Nesterov. En 1983, Nesterov proposa dans [5] une modification de la méthode du Momentum et montra que son algorithme présente une meilleure convergence théorique pour l'optimisation des fonctions convexes. Cette approche devint très populaire grâce à ses performances en pratique comparée à la méthode classique.
La principale différence entre la méthode de Nesterov et la méthode du momentum réside dans le fait que cette dernière commence par calculer le gradient à l'emplacement courant θ_t avant de faire un pas dans la direction de la vélocité accumulée, alors que le momentum de Nesterov fait d'abords un pas de calcul pour obtenir une approximation du paramètre mis à jour, que l'on note $\widetilde{\theta}_{t+1}$, et corrige ensuite ce pas en calculant le gradient à cet emplacement.
Un pas du momentom de Nestrov est décrit par :

$$\begin{aligned} \widetilde{\theta}_{t+1} &\leftarrow \theta_t + \mu v_t, \\ v_{t+1} &\leftarrow \mu v_t - \eta \nabla J\big(\widetilde{\theta}_{t+1}\big), \\ \theta_{t+1} &\leftarrow v_{t+1} + \theta_t. \end{aligned} \tag{4}$$

2.3 Extensions de la descente de gradient

Il existe plusieurs variantes de l'algorithme de descente de gradient. Dans la suite, nous présentons trois méthodes différentes assez proches de nos méthodes présentées dans ce travail :
Descente de gradient moyennée. Cette méthode a été proposée et étudiée par Polyak dans [6]. L'idée est de remplacer le calcul du paramètre θ_t, par le calcul de la moyenne temporelle de ces valeurs, et ce à partir des mises à jour obtenues par la descente du gradient :

$$\overline{\theta}_T \leftarrow \frac{1}{T} \sum_{t=0}^{T} \theta_t. \tag{5}$$

Adagrad. Le principe de cette méthode, proposée en 2011 dans [7], est de faire que le taux d'apprentissage s'adapte aux paramètres, faisant de sorte qu'il s'ajuste automatiquement, en fonction de "l'éparsité" des paramètres. Adagrad abaisse progressivement le taux d'apprentissage mais pas de la même manière pour tous les paramètres : les dimensions à pente plus prononcée voient leur taux abaissé plus rapidement que celles à pente douce. Plus formellement, le pas est décrit par :

$$\forall i,\ (\theta_{t+1})_i \leftarrow (\theta_t)_i - \alpha \frac{\big(\nabla J(\theta_t)\big)_i}{\sqrt{\sum_{u=1}^{t} \big(\nabla J(\theta_u)\big)_i^2}},\ \alpha > 0. \tag{6}$$

RMSProp L'algorithme [8] ajuste automatiquement le taux d'apprentissage à chaque paramètre, comme Adagrad. Cependant, il ne cumule que les gradients issus des itérations récentes. Pour cela, il utilise une moyenne glissante :

$$\begin{aligned} &\forall i,\ (\nabla_{t+1})_i \leftarrow \delta(\nabla_t)_i + (1-\delta)\big(\nabla J(\theta_t)\big)_i^2, \\ &\forall i,\ (\theta_{t+1})_i \leftarrow (\theta_t)_i - \alpha \frac{\big(\nabla J(\theta_t)\big)_i}{\sqrt{(\nabla_{t+1})_i}},\ \alpha > 0. \end{aligned} \tag{7}$$

Ici, $\delta(\nabla_t)_i$ est la moyenne quadratique glissante du gradient. La division du gradient de la fonction objective par la racine de la moyenne quadratique glissante (c'est à dire l'amplitude) améliore la convergence.

Adam. Adam [9] est l'un des algorithmes les plus récents et les plus efficaces pour l'optimisation par descente de gradient. Le principe est le même que pour Adagrad et RMSProp : il adapte automatiquement le taux d'apprentissage pour chaque paramètre. Sa particularité est de calculer (m_t, v_t) des "estimations adaptatives des moments". Il peut donc être vu comme une généralisation de l'algorithme Adagrad :

$$\begin{array}{l} \forall i,\ (m_{t+1})_i \leftarrow \beta_1 \cdot (m_t)_i + (1-\beta_1) \cdot \left(\nabla J(\theta_t)\right)_i, \\ \forall i,\ (v_{t+1})_i \leftarrow \beta_2 \cdot (v_t)_i + (1-\beta_2) \cdot \left(\nabla J(\theta_t)\right)_i^2, \\ \forall i,\ (\theta_{t+1})_i \leftarrow (\theta_t)_i - \alpha \frac{\sqrt{1-\beta_2}}{1-\beta_1} \frac{(m_t)_i}{\sqrt{(v_t)_i + \epsilon}}, \\ \alpha, \epsilon > 0 \text{ et } \beta_1, \beta_2 \in]0,1[. \end{array} \tag{8}$$

Ici m_t est le premier moment du gradient (la moyenne) et v_t est son second moment (variance non-centrée). ϵ est un paramètre de précision. Sa valeur par défaut dans l'outil populaire d'apprentissage des CNN Caffe [10] est 10^{-8}. Les paramètres β_1 et β_2 sont utilisés pour réaliser des moyennes d'exécution sur les moments m_t et v_t respectivement.

3 Notre contribution : SGD avec filtrage des paramètres

Dans ce travail, nous nous intéressons à la mise à jour du vecteur de paramètres et proposons des méthodes compatibles avec les solveurs existant (sous Caffe notamment). L'idée principale est qu'au lieu de ne considérer que la dernière mise à jour d'un paramètre, nous considérons des moyennes pondérées de ses dernières mises à jour. Cela permet de filtrer les paramètres du réseau assurant une meilleure convergence aussi bien en terme de "profondeur" de la descente que de la stabilité. L'objectif étant de diminuer la perte et d'augmenter la précision en moyenne tout en réduisant leurs variances. La stabilité du processus permet alors de réduire le nombre d'itérations.

A noter que vue le corpus considéré dans ce travail (MNIST), nous ne nous sommes pas intéressés aux temps d'exécution des différents algorithmes mais plutôt à leurs précisions.

3.1 Moyenne pondérée pour la descente de gradient

Notre première méthode de filtrage est appelée "Moyenne pondérée". L'idée générale de la méthode est de mettre à jour la valeur courante du paramètre en fonction de ses valeurs aux itérations précédentes et ce en utilisant la *mémoire* de l'entrainement. Le filtrage des paramètres du réseau peut s'opérer i) au début du processus d'optimisation (Méthode 1) ou encore ii) quand une stabilisation de la moyenne temporelle de la fonction objective est observée (Méthode 2). Dans la suite, nous avons choisi d'utiliser la méthode de momentum de Nesterov comme méthode de base. Cependant, il est clair que notre approche est compatible avec toutes les autres méthodes d'optimisation.

Méthode 1 : Moyenne pondérée. Nous introduisons un effet *mémoire* sur la mise à jour du paramètre de vélocité :

$$\begin{array}{l} \widetilde{\theta}_{t+1} \leftarrow \theta_t + \mu v_t,\ \ v_{t+1} \leftarrow \mu v_t - \eta \nabla J\big(\widetilde{\theta}_{t+1}\big), \\ \theta_{t+1} \leftarrow \begin{cases} v_{t+1} + \theta_t,\ \forall t < \delta \\ v_{t+1} + \sum\limits_{u=t-\delta+1}^{t} \lambda_u \theta_u,\ \forall t \geq \delta \end{cases} \end{array} \tag{9}$$

où δ est le nombre de mises à jour passées utilisées pour effectuer une nouvelle mise à jour, et (λ_u) est une suite décroissante de nombres positifs, telle que $\sum\limits_{u=t-\delta+1}^{t} \lambda_u = 1$. Nous avons utilisé le cas gaussien pour cette suite, i.e., avant la normalisation :

$$\forall u \in [|t-\delta+1, t|],\ \lambda_u = e^{-\frac{(t-u)^2}{\sigma^2}},\ \sigma \in \mathbb{R}. \tag{10}$$

Ce nouveau modèle utilise deux hyper-paramètres δ et σ. Le paramètre δ permet de contrôler le nombre de mises à jour à prendre en compte dans un pas de mise à jour, le paramètre σ permet lui de régler la pondération des mises à jour passées. À noter que les poids sont de plus en plus forts pour les dernières mises à jour. Les valeurs choisies pour σ sont en cohérence avec celles prises par δ. Plus précisément, nous nous assurons que σ vérifie bien $\lambda_t = \kappa \lambda_{t-\delta+1}$. Les valeurs prises par δ sont comprises entre 4 et 32 et pour ces valeurs, nous choisissons $\kappa = 10$. Cela permet de s'assurer que les mises à jour les plus récentes seront privilégiées dans le processus de mise à jour.

Méthode 2 : Moyenne pondérée avec délai. Le principe de cette est méthode est de commencer la régularisation des paramètres en partant de fonctions cibles déjà stables. Le moment de stabilisation est défini par des variations limitées de la moyenne temporelle $m(t)$ de la fonction cible (fonction de perte de l'entrainement ou fonction de précision) : on choisit le premier t tel que le critère :

$$\frac{|m(t) - m(t-1)|}{m(t-1)} < \alpha \tag{11}$$

soit satisfait pour un α donné. Le temps auquel le moyennage commence est noté T_{ba}. L'expression de notre solveur est donnée par :

$$\begin{array}{l} \widetilde{\theta}_{t+1} \leftarrow \theta_t + \mu v_t,\ \ v_{t+1} \leftarrow \mu v_t - \eta \nabla J\big(\widetilde{\theta}_{t+1}\big), \\ \theta_{t+1} \leftarrow \begin{cases} v_{t+1} + \theta_t,\ \forall t < \max\left\{\delta, T_{ba}\right\} \\ v_{t+1} + \sum\limits_{u=t-\delta+1}^{t} \lambda_u \theta_u,\ \forall t \geq \max\left\{\delta, T_{ba}\right\} \end{cases} \end{array} \tag{12}$$

3.2 Moyenne glissante pondérée avec délai

La Méthode 2 a montré, lors de nos expérimentations, une meilleure stabilité de la fonction objective (perte). Cependant, elle nécessite l'utilisation d'une mémoire-tampon

pour garder en mémoire les dernières mises à jour des paramètres du réseau. Cela peut s'avérer assez coûteux en mémoire quand on travaille avec des réseaux de neurones à des millions de paramètres. La *moyenne glissante pondérée* a pour objectif d'exploiter l'idée principale de la Méthode 2 tout en gardant à l'esprit la calculabilité de l'approche.

En ne retenant pas la propriété gaussienne des coefficients λ_u, on peut implémenter efficacement la méthode de moyenne glissante pondérée des mises à jour des paramètres et ce en utilisant le même principe que pour l'algorithme RMSProp. La seule différence est que l'on applique le principe directement à l'apprentissage des paramètres du réseau au lieu de l'appliquer aux magnitudes des gradients dans l'algorithme RMSProp. Cela permet d'établir une expression récursive qui n'implique aucune mémoire - tampon. La forme théorique des λ_u devient alors exponentielle au lieu d'être gaussienne.

Méthode 3 : Moyenne glissante pondérée avec délai défini par la stabilisation de la fonction objective. Nous proposons de garder le délai de moyennage, suivant le critère de l'expression de l'equation 12 :

$$\begin{aligned} &\widetilde{\theta}_{t+1} \leftarrow \theta_t + \mu v_t, \;\; v_{t+1} \leftarrow \mu v_t - \eta \nabla J(\widetilde{\theta}_{t+1}), \\ &\overline{\theta}_{t+1} \leftarrow \begin{cases} \theta_t, \; \forall t < T_{ba} \\ \epsilon \cdot \theta_t + (1-\epsilon) \cdot \overline{\theta}_t, \; \forall t \geq T_{ba} \end{cases} \\ &\theta_{t+1} \leftarrow v_{t+1} + \overline{\theta}_{t+1}, \end{aligned} \tag{13}$$

T_{ba} est l'instant auquel le filtrage commence, défini par le critère et $\overline{\theta}_t$ représente la moyenne glissante pondérée des mises à jour des paramètres du réseau et doit être comparée au terme $\sum_{u=t-\delta+1}^{t} \lambda_u \theta_u$ de la Méthode 2. (12). Le paramètre $\epsilon \in]0,1[$ permet de régler la pondération : ϵ est la confiance que l'on attribue à la dernière mise à jour θ_t dans le dernier pas.

Méthode 4 : Moyenne glissante pondérée adaptative avec délai défini par la fonction objective. Le dernière méthode proposée dans ce travail vise à étendre la méthode 3 an considérant un comportement adaptatif de la valeur de ϵ : l'idée principale est de chercher un comportement régulier du solveur, autrement dit, nous voulons éviter un changement brutal du pas de mise à jour. Par ailleurs, il serait intéressant, lors du pas de mise à jour de $\overline{\theta}_{t+1}$, de commencer par accentuer la dernière mise à jour des paramètres θ_t du réseau, puis de donner de plus en plus de poids au filtrage $\overline{\theta}_t$.

Nous considérons donc une fonction qui commence à décroitre très régulièrement, et presque linéairement, en partant de 1, ce qui nous convient très bien car la fonction objective tends souvent à devenir monotone en moyenne après stabilisation. Nous proposons donc la fonction suivante pour la confiance adaptative $\epsilon(t)$:

$$\epsilon(t) = \begin{cases} 1, \; \forall t < T_{ba} \\ \cos\left(\frac{\pi}{2} \log_2\left(\frac{y(t)}{\pi}\right)\right), \; \forall t \in [T_{ba}, \omega] \\ 0, \; \forall t \geq \omega, \end{cases} \tag{14}$$

CNN	LeNet
Batch size d'entrainement	64
Batch size de test	100
Data storage	HDF5
Mélange lors de l'entrainement	Au début et après chaque epoch
Mélange lors du test	Au début
Fonction de perte	Softmax
Initialisation des poids	Initialisation aléatoire (Xavier)

TABLE 1 – Paramètres du réseau CNN

où y est une application qui renvoie l'intervalle $[T_{ba}, \omega]$ sur $[\pi, 2\pi]$ et ω est choisi tel que la valeur de ϵ au dernier pas d'entrainement est définie par un nouveau paramètre (appelé "end_confidence" dans Caffe) :

$$\epsilon_{\text{end}} = \epsilon(max_iter - 1) = \text{end_confidence}, \tag{15}$$

où max_iter est le nombre total d'itérations d'entrainement dans Caffe.

4 Expérimentations et résultats

Nos expérimentations ont été menées en utilisant le réseau CNN LeNet et le corpus pour les chiffres manuscrits MNIST [2]. En effet, le processus d'apprentissage pour cette tâche est très rapide ce qui permet de régler facilement les méta-paramètres du modèle. L'ensemble d'entrainement est composé de 60000 éléments et celui pour les tests en contient 10000. Nous avons mené nos expérimentations en utilisant Caffe, sur un serveur avec une GPU NVIDIA Tesla K40 et une CPU INTEL i7-2600 @ 3.40GHz. Les paramètres de notre CNN sont données dans la Table 1.

Les quatre méthodes présentées dans ce travail ont été évaluées et comparées à la variante momentum de Nesterov de la descente de gradient stochastique (notée dans la suite base-line). Nous avons bien sûr gardé les mêmes hyper-paramètres dans les expérimentations : les batch size, le taux d'apprentissage et le coefficient de momentum. Les résultats des deux méthodes 1 et 2 sont présentées dans la Table 2. Nous exhibons les moyennes et les écart-types une fois la stabilisation atteinte. La méthode 2 a une faible moyenne de perte et un faible écart-type principalement durant la phase de validation.

Les résultats des méthodes 3 et 4 sont donnés dans la Table 3. La comparaison se fait sur les mêmes intervalles d'itérations soit de 7500 à 10000. En terme de perte moyenne et d'écart-type, la méthode 4 surpasse la méthode de base et la méthode 2 mais est assez proche des résultats de la méthode 3.

Les courbes de la fonction de perte pour les méthodes 3 et 4 sont données, respectivement, dans les Figures de 1 à 4. Ces figures permettent également d'avoir un zoom sur les courbes de précision et de perte à l'étape de validation. Le comportement est presque le même : la méthode 4 est meilleure que la méthode base-line et le gain en perte moyenne est observé progressivement vers la fin de l'entrainement, lequel est fixé à 10000 itérations.

En effet, la méthode 4 est conçue de manière à ce que les améliorations obtenues avec la méthode 3 le soient de manière régulière. Les résultats obtenus montrent que le fait d'adapter, de manière dynamique, la valeur du coefficient de filtrage ϵ (nommé "confidence") avec une fonction très régulière nous permet de maintenir

		Base-line	Moyenne pondérée (méthode 1) avec $T_{ba} = 0$	Moyenne pondérée (méthode 2) avec $T_{ba} = 2500$
Entrainement	Perte moyenne	0.016402	0.036133	0.011357
	Ecart-type	0.024027	0.0031367	0.010373
Validation	Perte moyenne	0.029964	0.042632	0.027818
	Ecart-type	0.004077	0.007111	0.000594

TABLE 2 – Pertes moyennes et écart-types observés pour les méthodes 1 et 2 pour n allant de 2500 à 10000 pour la méthode 1 et de 6250 à 10000 pour la méthode 2.

		Base-line	Moyenne glissante pondérée (méthode 3) avec $T_{ba} = 2500$	Moyenne glissante pondérée adaptative (méthode 4) avec $T_{ba} = 2500$
Entrainement	Perte moyenne	0.008076	0.008489	0.005645
	Ecart-typ	0.009694	0.006126	0.003660
Validation	Perte moyenne	0.027168	0.027370	0.026030
	Ecart-typ	0.001628	0.000977	0.000810

TABLE 3 – Pertes moyennes et écart-types observés pour les méthodes 3 et 4 et pour n allant de 7500 à 10000.

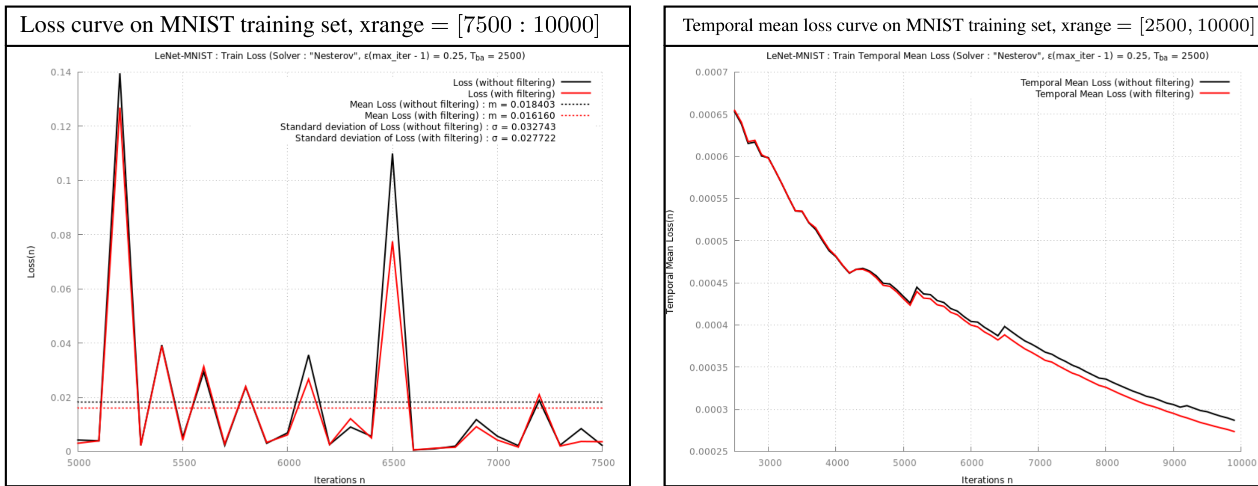


FIGURE 1 – Résultats de l'entrainement pour la méthode 3 : un zoom sur la courbe de la perte (à gauche), et la courbe de la perte moyenne (à droite)

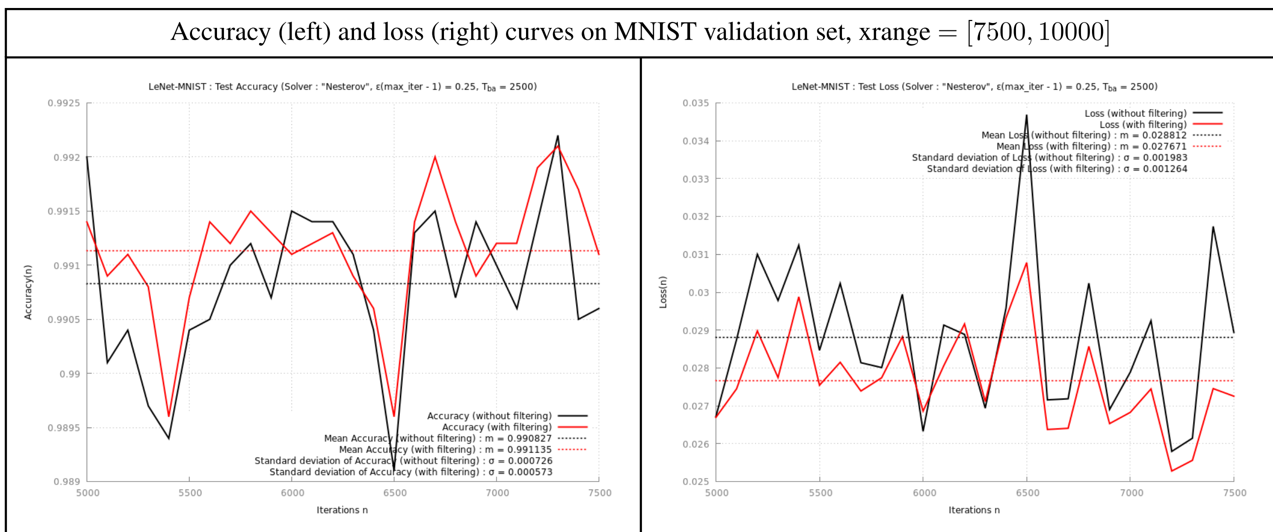
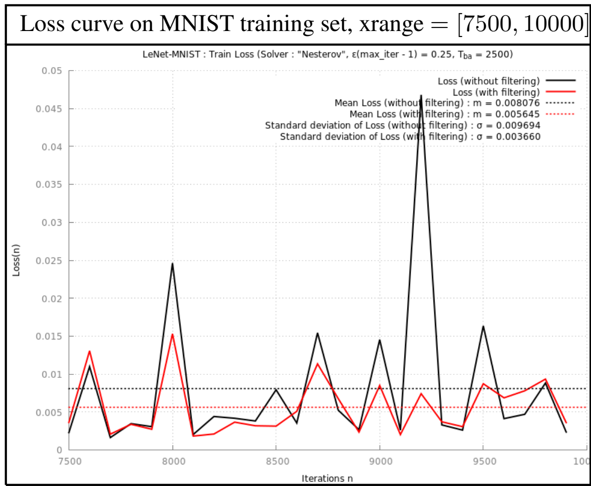


FIGURE 2 – Résultats de la validation pour la méthode 3 : zoom sur les courbes de précision et de perte.



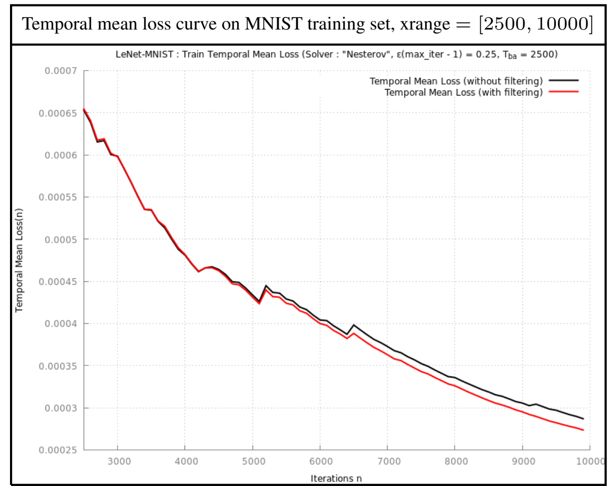


FIGURE 3 – Résultats de l'entrainement pour la méthode 4 : un zoom sur la courbe de la perte (à gauche), et la courbe de la perte moyenne (à droite)

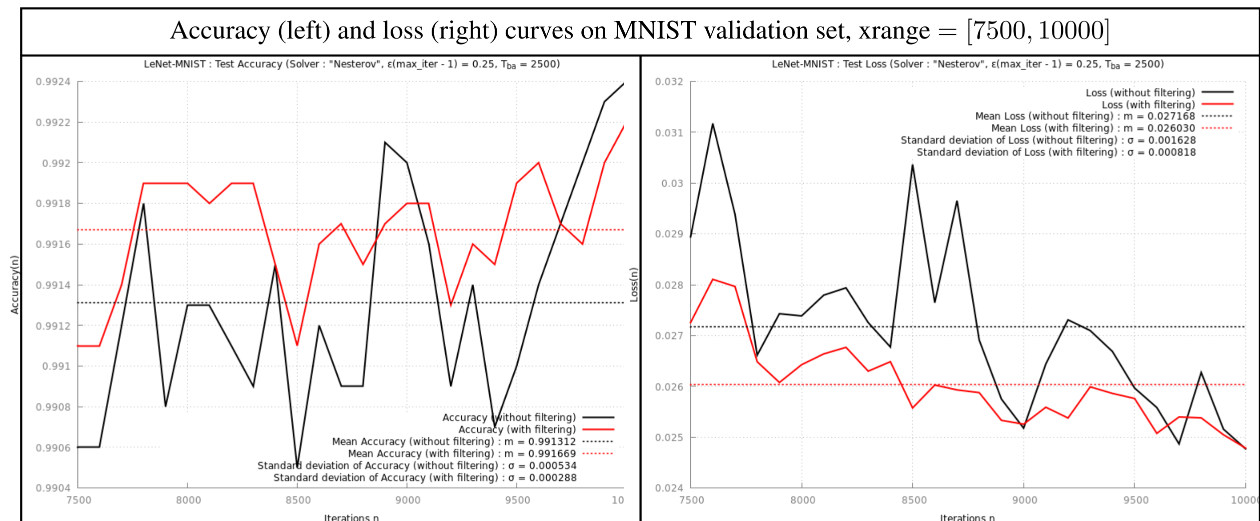


FIGURE 4 – Résultats de la validation pour la méthode 4 : zoom sur la précision (à gauche) et courbe de perte (à droite).

une diminution de l'écart-type des courbes et de réduire légèrement la perte ce qui produit une légère augmentation de la précision. Néanmoins, ces améliorations ne sont pas observables dès le début du filtrage mais plutôt de manière progressive.

5 Conclusion et perspectives

Nous avons proposé et testé différentes extensions des méthodes de descente de gradient stochastique avec le momentum de Nesterov. Nous avons considéré principalement les plus utilisées pour l'apprentissage profond. Notre objectif étant d'augmenter la stabilité de l'entrainement. Nos extensions sont compatibles avec les solveurs existant. Parmi les quatre méthodes de filtrage proposées, deux sont particulièrement intéressantes : la méthode 3 consiste à opérer une moyenne glissante pondérée et la seconde, méthode 4 est une version adaptative de la 3. Les deux méthodes améliorent la méthode de base. La méthode 4 présente un intérêt supplémentaire : elle combine un filtrage récursif des paramètres avec une adaptabilité tout au long du temps.
Le gain réalisé par nos méthodes de filtrage de solveurs montre qu'il est possible d'accélérer le temps d'entrainement tout en améliorant la stabilité ou encore d'améliorer la qualité et de renforcer la stabilité des résultats, et d'arrêter la processus d'optimisation plus tôt. Les méthodes ont été testées sur un corpus relativement simple (MNIST). Comme suite de ce travail, nous envisageons d'utiliser les méthodes proposées sur un problème de classification d'images de grande échelle.

Références

[1] L. V. Fausett. *Fundamentals of Neural Networks : Architectures, Algorithms, and Applications*. Prentice-Hall, 1994.

[2] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010.

[3] Y. Dauphin, R. Pascanu, Ç. Gülçehre, K. Cho, S. Ganguli, and Y. Bengio. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. *CoRR*, abs/1406.2572, 2014.

[4] R. Hauser. Line search methods for unconstrained optimisation. *Lecture 8, Numerical Linear Algebra and Optimisation Oxford University Computing Laboratory*, 2007.

[5] Y. Nesterov. A method of solving a convex programming problem with convergence rate o (1/k2). *Soviet Mathematics Doklady*, 27 :372–376, 1983.

[6] B. Polyak and A B. Juditsky. Acceleration of stochastic approximation by averaging. *SIAM Journal on Control and Optimization*, 30 :838–855, 07 1992.

[7] J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. 12 :2121–2159, July 2011. http ://jmlr.org/papers/volume12/duchi11a/duchi11a.pdf.

[8] G. Hinton, N. Srivastava, and K. Swersky. Neural networks for machine learning - lecture 6a - overview of mini-batch gradient descent. 2012. http ://www.cs.toronto.edu/ tijmen/csc321/slides/lecture_slides_lec6.pdf.

[9] Diederik P. Kingma and J. Ba. Adam : A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[10] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe : Convolutional architecture for fast feature embedding. *arXiv preprint arXiv :1408.5093*, 2014.