PARTIE 2 : ADAM

ADAM est un algorithme (adaptatif) d'optimisation qui permet donc de trouver le minimum d'une fonction coût donnée. L'objectif de cette partie est d'implémenter ADAM dans un algorithme de réseaux de neurones.

Il s'agit de manière sous-jacente de :

- Coder un algorithme de réseaux de neurones puisque les librairies Matlab ne sont a priori pas open-source et qu'il est difficile d'intégrer une fonction de coût spécifique dans un code « inconnu ».
- Tester notre algorithme de réseaux de neurones profonds avec une fonction coût gradient au préalable pour aider à la vérification de l'algorithme ADAM par la suite, et pour comparer leurs performances.
- Implémenter l'algorithme ADAM.

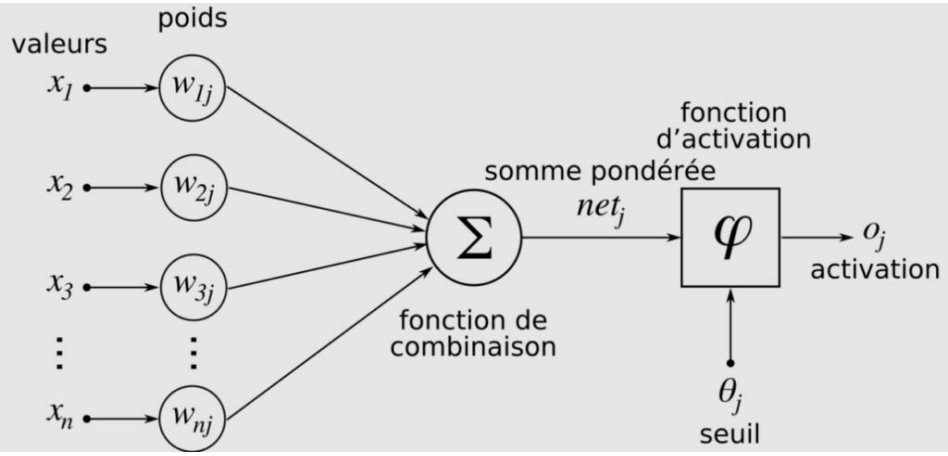
Rappels & Principes de bases

Le grand principe du machine Learning peut se résumer en 4 étapes :

- Le pré-traitement des données pour les rendre exploitables pour notre modèle (feature engineering).
- Le choix du modèle. En machine Learning, on utilise souvent des modèles paramétriques de type fonction : une droite (régression linéaire, régression logistique), des courbes polynomiales, exponentielles.
- On cherche ensuite à trouver les paramètres optimaux du modèle pour minimiser une certaine fonction de coût, par exemple l'écart entre la sortie des données et la sortie du modèle pour les entrés du jeu de données.
- Une fois les bons paramètres obtenus, on peut alors utiliser notre modèle pour faire des prédictions (classification, régression).

Le modèle en Deep Learning

Mais le fait est que les modèles sont parfois plus implicites que de simples fonctions. En effet, en Deep Learning, on constitue un réseau de neurones en reliant de manière spécifique des neurones : chaque neurone est en fait une fonction de la forme f(X) = g [W. X + b] où g est une fonction d'activation (fonction permettant de transformer l'input en valeur de probabilité, sigmoïde par exemple). Illustration :

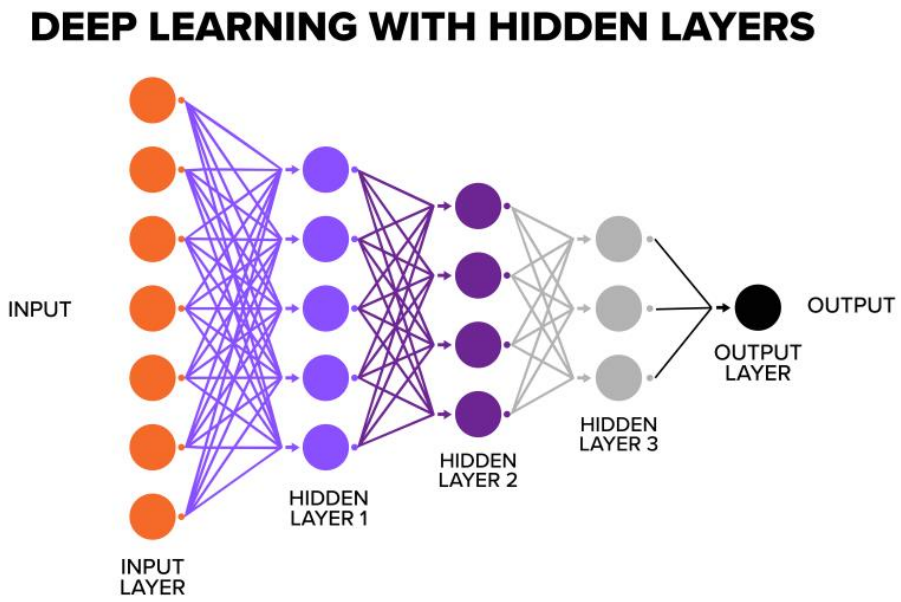


Les quantités w constituent les poids sur les entrés et b le biais.

L'ensemble des poids associés à tous les neurones du réseau sont les paramètres à optimiser. Le modèle est alors bien plus complexe puisqu'il s'agit de l'ensemble des fonctions de neurones interconnectées. Le choix du modèle en Deep Learning est donc lié au choix des couches (Layers) et à leur organisation.

Algorithmie de Réseaux de Neurones profonds

Dans le cadre de notre problème, on s'intéresse à un réseau de neurones profond (« Deep ») avec uniquement des couches Fully Connected Networks (toutes les sorties de réseaux sont connectées aux entrés des neurones de la couche suivante), et des données d'entrées déjà traités. Je fais ce choix pour avoir structure à la fois simple et fiable. Pour les cas plus compliqués, comme la reconnaissance d'images, on ajoute des couches de filtrage au réseau.



Légende : Il s'agit d'un deep neural network avec 3 couches Fully Connected.

Selon le nombre de neurones de la couche, on peut formuler avec différentes dimensions de matrices W,b,X la fonction f. De manière général pour le passage d'une couche à k à une couche k+1, on a :

$$Z^{(k+1)} = W^{(k+1)} \cdot A^{(k)} + b^{(k+1)}$$

$$A^{(k+1)} = \sigma(Z^{(k+1)})$$

Où σ est une fonction d'activation. Par exemple, pour une activation sigmoïde :

$$\sigma(Z^{(k+1)}) = \frac{1}{1 + \exp(-Z^{(k+1)})}$$

Pour le dimensionnement,

$Z^{(k+1)} \in \mathbb{R}^{p \times m}$: Résultats linéaires pour les p neurones de la couche $k+1$ sur m échantillons.

$W^{(k+1)} \in \mathbb{R}^{p \times n}$: Matrice des poids reliant n neurones de la couche k aux p neurones de la couche $k+1$.

$A^{(k)} \in \mathbb{R}^{n \times m}$: Activations en sortie de la couche k sur m échantillons.

$b^{(k+1)} \in \mathbb{R}^{p \times 1}$: Vecteur des biais pour les p neurones de la couche $k+1$.

Notons pour la première couche que :

$A^{(0)} = X$, où $X \in \mathbb{R}^{n \times m}$ est la matrice des données d'entrée.

n représente le nombre de features (dimensions des données d'entrée).

m est le nombre d'échantillons dans le lot d'entraînement.

Pour le fonctionnement, on fait passer les données les unes à la suite des autres dans le réseau, (propagation vers l'avant ou « forward propagation). Puis à chaque sortie pour un échantillon, on évalue l'erreur (différence entre les prédictions et les valeurs vraies) grâce à la fonction coût log-loss. Il faut ensuite propager les erreurs depuis la sortie du réseau vers ses couches précédentes. Il s'agit de la back-propagation, où on calcule toutes les dérivées d[Log-Loss] / d[W(couche$_k$)] et d[Log-Loss] / d[b(couche$_k$)] grâce à la règle de la chaine. Enfin, avec connaissances des paramètres de la back-propagation, on peut mettre à jour les paramètres w puis on recommence avec l'échantillon suivant. Les formules de mise à jour changent en fonction de l'algorithme utilisé. Ainsi pour un algorithme de descente du gradient, on a :

$$w \leftarrow w - \eta \frac{\partial \text{Loss}}{\partial w}$$

Cependant, pour un optimiseur avancé comme Adam, des moyennes mobiles des gradients et des carrés des gradients sont utilisées pour ajuster dynamiquement la taille des pas. Cela nécessite plusieurs étapes supplémentaires (mises à jour de m et v, correction des biais, etc.).

L'algorithme converge lorsque l'erreur (par exemple, la log-loss) atteint un seuil fixé ou lorsque les gradients deviennent très petits. Toutefois, la convergence dépend fortement des paramètres d'apprentissage et de la structure du réseau, et un mauvais paramétrage peut entraîner une convergence lente ou un piège dans des minima locaux.

L'algorithme ADAM

L'algorithme **ADAM (Adaptive Moment Estimation)** est un optimiseur utilisé pour ajuster les poids d'un modèle en minimisant une fonction de coût. ADAM combine deux idées importantes des méthodes d'optimisation :

- **Momentum** : Utilise une moyenne exponentielle des gradients pour accélérer la convergence et éviter les oscillations.
- **RMSProp** : Adapte la taille des pas en fonction de la variance des gradients.

L'équation générale pour mettre à jour les paramètres est :

$$\theta_t = \theta_{t-1} - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

- m_t : moyenne mobile des gradients (β1 contrôle son taux de décroissance).
- v_t : moyenne mobile des carrés des gradients (β2 contrôle son taux de décroissance).
- $\hat{m}_t$: biais corrigé de m_t (car m_t tend vers 0 pour les premiers itérations).
- $\hat{v}_t$: biais corrigé de v_t

Les paramètres principaux de l'algorithme ADAM sont :

- α (taux d'apprentissage) : Détermine la taille du pas. Un bon point de départ est souvent 0.001.
- β1 et β2 : Contrôlent la contribution du passé dans les moyennes mobiles. Par défaut, β1 =0.9 et β2=0.999.
- ϵ : Petit terme pour éviter la division par zéro (souvent 10^{-8}).

ADAM est particulièrement adapté aux problèmes non convexes complexes (comme ceux des réseaux de neurones…) grâce à sa capacité à s'ajuster dynamiquement à l'échelle des gradients, tout en offrant une grande stabilité même en présence de données bruitées.

Paramètres d'ADAM & convergence

Discutons des différents paramètres d'ADAM.

α (taux d'apprentissage) :

- Un taux trop élevé (α>0.01) peut provoquer une divergence.
- Un taux trop bas ralentit la convergence.
- Il est souvent utile de tester des valeurs logarithmiques ($\alpha=10^{-3},10^{-4},10^{-5}$).

β1 (moyenne des gradients) :

- Si β1 est trop faible (≈0.5), ADAM devient sensible aux variations rapides des gradients.
- Une valeur plus élevée (≈0.9) offre un bon compromis entre stabilité et réactivité.

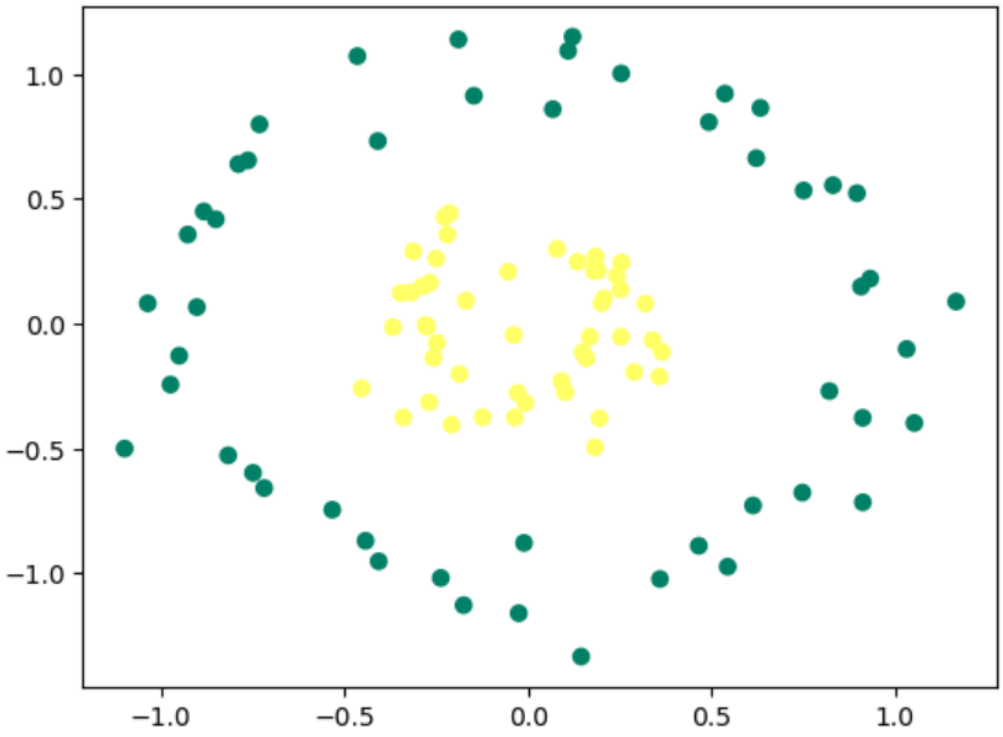
β2 (moyenne des carrés des gradients) :

- Des valeurs proches de 1 (β2=0.999) sont idéales pour des gradients très bruités.
- Si β2 est trop bas, ADAM agit comme une méthode de descente classique.

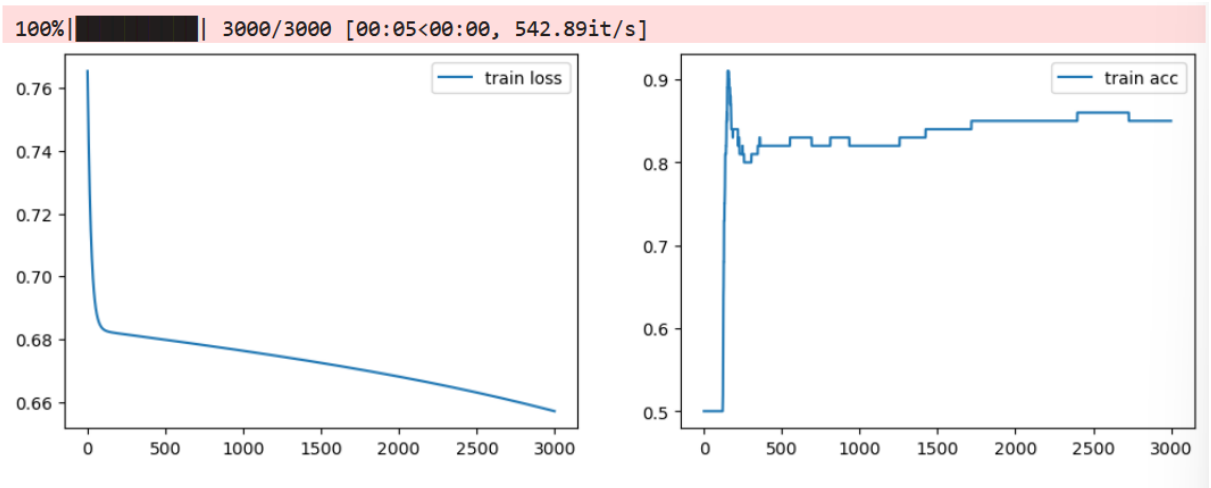
ϵ : Une petite valeur (10^{-8}) est généralement suffisante. Mais pour des gradients extrêmement faibles, augmenter ϵ peut améliorer la stabilité.

Résultats

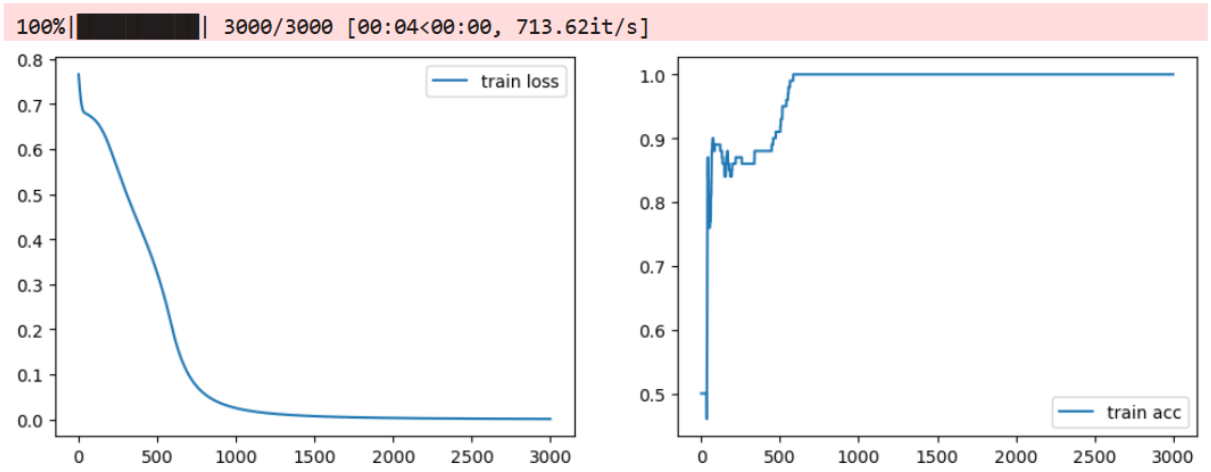
On génère un dataset à deux classes (données de sortie Y, Y(m) classe de l'échantillon m) contenant 100 échantillons. Ce datatset possède deux features X1 et X2, ce qui nous permet de représenter le dataset (et ensuite de dessiner les frontières) :



On passe ce dataset dans algorithme de réseaux de neurones profond de trois couches à 16 neurones chacun. Pour les mêmes paramétrages on obtient :



Cas avec un algorithme SGD

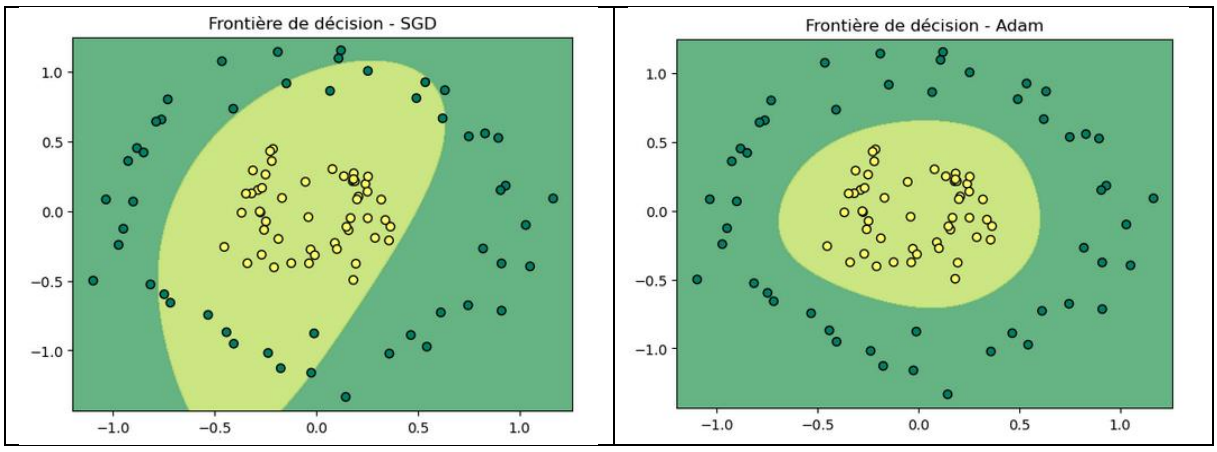


Cas avec un algorithme ADAM

Pour les mêmes hyperparamètres, on constate que ADAM a de bien meilleures performances que l'algorithme du gradient :

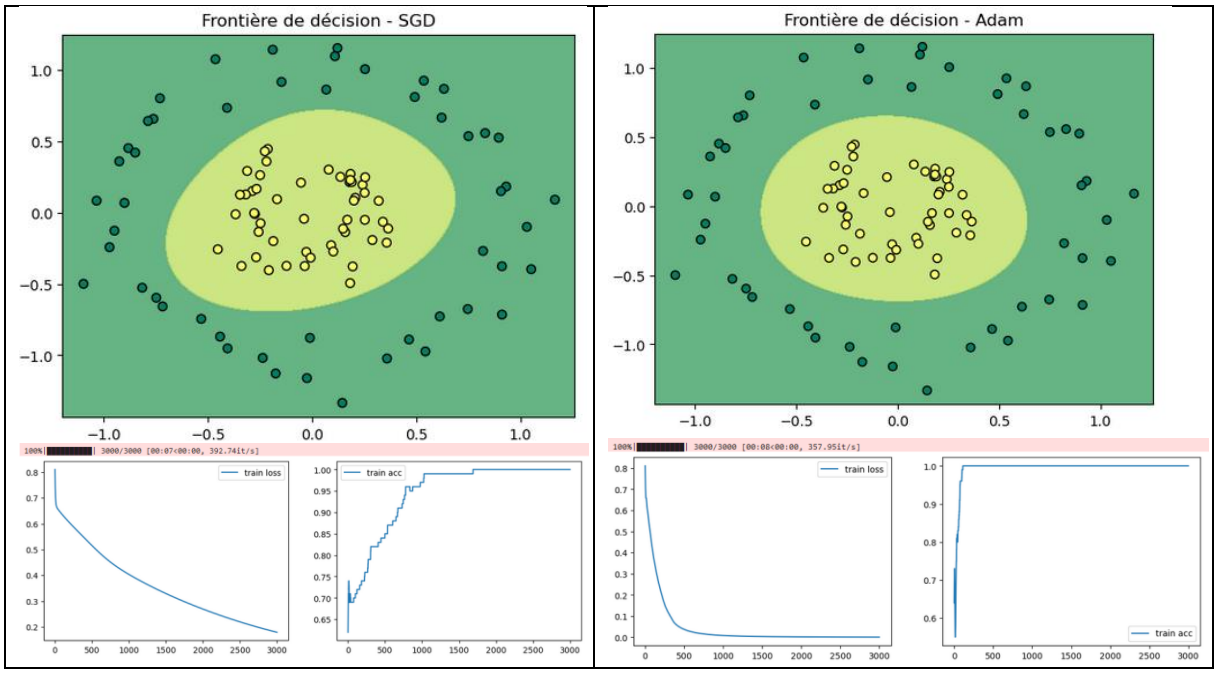
- La précision de la prédiction atteint 100% pour ADAM alors que ce n'est pas le cas pour le SGD.
- La convergence, donc l'entrainement du réseau est plus rapide pour le cas ADAM.

On peut à partir des paramètres W finaux du réseau pour ADAM et SGD afficher les frontières de décision :



Même pour un cas simple comme celui-ci (deux features et un dataset équilibré), Adam s'avère supérieur à SGD, même avec des hyperparamètres identiques.

A noter que j'ai volontairement pris un petit réseau de neurones pour mettre en avant la différence entre les deux algorithmes. Si on augmente la taille du réseau et que l'on passe de 16 à 32 neurones par couches, les deux algorithmes obtiennent alors des performances optimales pour ce cas simple.



Mais en complexifiant le cas (par exemple une application à la détection chat/chien) on se remmènerait au même écart entre les erreurs de deux algorithmes. **ADAM conserve son avantage relatif sur SGD**, en offrant à la fois une meilleure convergence et des erreurs finales plus faibles.