UMONS
Université de Mons

Faculté Polytechnique

First-order methods for large-scale optimization

Implémentation d'un perceptron pour la classification de documents

GLOESENER Maxime (191538)

SYED Tyemur (190865)

POLYTECH MONS

Prof. Nicolas Gillis

ACADÉMIE UNIVERSITAIRE WALLONIE-BRUXELLES

20 janvier 2023

Table des matières

I. Introduction 3
II. Analyse des données 3
III. Normalisation des données 4
1. Principe théorique 4
2. En pratique 5
IV. Perceptron 6
1. Algorithme 6
2. Initialisation des paramètres 7
3. Fonction d'activation 7
4. Erreur 8
6. Méthode d'optimisation 9
Descente de gradient 9
Descente de gradient stochastique 9
Descente du gradient stochastique par mini-batch 10
Gradient accéléré 10
Adam 11
7. Pas d'apprentissage 12
V. Implémentation et comparaison des résultats 14
Early Stopping 14
Analyse des résultats 14
Perspectives d'amélioration 16

I. Introduction

L'objectif de ce projet est d'implémenter un perceptron pour classifier des documents. Le perceptron est un algorithme d'apprentissage supervisé utilisé pour faire de la classification. Il existe de nombreux paramètres sur lesquels jouer pour améliorer les performances d'un perceptron comme le choix de la fonction d'activation, l'erreur utilisée pour évaluer le modèle, le pas d'apprentissage, l'initialisation des poids ou encore l'optimiseur utilisé pour mettre à jours les poids et les biais du perceptron. Dans le cadre du projet, nous nous sommes intéressés au perceptron classique avec une couche d'entrée et une couche de sortie, mais il est possible de jouer sur le nombre de couches, et d'ajouter des couches cachées, on parle alors d'un MLP pour Multi Layer Perceptron.

Ce projet nous a permis d'explorer ces différents paramètres et de mieux comprendre l'influence que chaque paramètre a sur la qualité des résultats obtenus et sur la vitesse de convergence de l'algorithme.

II. Analyse des données

Les données utilisées pour entraîner le modèle correspondent à l'occurrence de mots dans des documents. Les données sont au format suivant :

	Document 1	**Document 2**	…	…	**Document m**
Mot 1	0	5	0	2	0
Mot 2	1	0	5	1	0
….	0	2	0	3	1
Mot n	0	5	1	0	0

Les données sont de dimension nombre de mots n* nombre de documents m avec n qui vaut 43586 et m qui vaut 13960.

Avant de commencer à implémenter un algorithme, il est intéressant de bien connaître ses données. On se rend compte rapidement que la matrice est très creuse, il y a au total 608460560 entrées dans la matrice et 607355563 entrées sont nulles pour 1104997 entrées non nulles, ce qui équivaut à 99.82% d'entrées nulles.

Nous remarquons aussi que la valeur maximale d'occurrence d'un mot dans un document est de 549 pour le document 7684 et le mot 4757.

Nous avons comme données :

- X une matrice contenant m documents
- Chaque document est décrit pas un vecteur de taille n
- Le vecteur de taille n représente l'occurrence de chaque mot dans le document
- Y est un vecteur contenant les classes des documents

L'objectif est donc d'entraîner un modèle à partir du nombre de mots dans un document pour prédire la classe de celui-ci. Lorsque le modèle sera entraîné, on pourra évaluer sa qualité sur des données de test qui ont le même format que les données d'entraînement.

III. Normalisation des données

1. Principe théorique

Il est important lorsqu'on entraîne des algorithmes d'apprentissage supervisé de normaliser les données. La convergence de l'algorithme est en général plus rapide lorsque la moyenne des entrées est proche de zéro. Une des raisons qui explique cela est que les algorithmes d'optimisation utilisant la descente de gradient ont tendance à converger plus facilement lorsque la surface d'optimisation est « circulaire » au lieu d'être « elliptique ». Il est donc intéressant de transformer les données pour que la moyenne de chaque entrée soit proche de 0, que les covariances de chaque entrée soit similaire et, si possible, que les entrées ne soient pas corrélées.

En général, si les données ne sont pas transformées, l'algorithme de descente de gradient aura tendance à osciller beaucoup autour de l'optimum même si la fonction est fortement convexe.

Par exemple, si on prend la fonction : $f(x) = {x_1}^2 + 25\ {x_2}^2$ qui est strictement convexe car c'est une fonction quadratique avec des coefficients positifs. On peut remarquer que le minimum global est en x = [0 ; 0].

Cette fonction a un gradient qui vaut : $\nabla f(x) = [2\ x_1; 50\ x_2]$.

Nous allons essayer de trouver le minimum de cette fonction en utilisant la descente de gradient. On utilise un pas d'apprentissage de 0.035 et un itéré de départ à [0.8 ; 0.8]. Cela va nous permette d'étudier la convergence de l'algorithme lorsque les coefficients ne sont pas normalisés.

L'algorithme de descente de gradient suit cette formule : $x^{(1)} = x^{(0)} - \alpha \nabla f\ (\ x^{(0)})$. Si on étudie la convergence de l'algorithme, on remarque une forte oscillation. Nous avons pu implémenter cet exemple pour bien visualiser le problème :

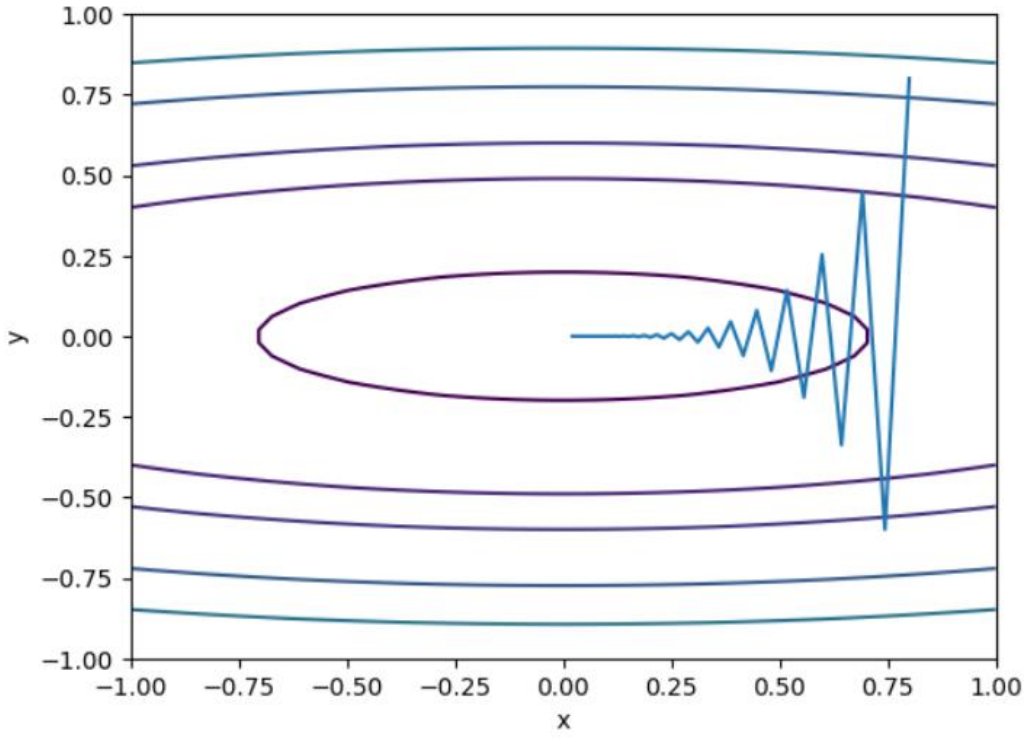

Figure 1 Exemple descente de gradient

Le code qui a permis de générer cet exemple se trouve dans le fichier « gradientdescent.ipynb ». C'est très intéressant de jouer avec les différents paramètres pour bien comprendre l'influence de ceux-ci avant de travailler sur les vraies données.

Ce phénomène de zigzag s'explique car la fonction est beaucoup plus raide selon x2 que selon x1. Du coup, la direction du gradient n'est pas toujours dirigée vers le minimum global. C'est une propriété connue quand les valeurs propres de la hessienne $\nabla^2 f(x)$ ont des échelles différentes. Le gradient va alors progresser lentement dans les directions correspondant aux petites valeurs propres et rapidement dans les directions avec les grandes valeurs propres. Le meilleur chemin pour se diriger vers le minimum global serait de se diriger en diagonale mais comme le gradient utilise simplement une information locale, il ne sait pas que cette stratégie est la meilleure.

Transformer les entrées pour que chaque entrée ait une moyenne proche de zéro, permet de préconditionner la matrice hessienne et de rendre la surface d'optimisation circulaire. Comme on travaille avec un cercle, on aura tendance à converger vers le minimum global alors que lorsqu'on prend une direction orthogonale à une ellipse, on ne converge pas forcément vers son centre et son minimum global.

L'intuition pour cela est que le gradient se dirige dans la direction la plus raide, orthogonalement aux courbes de niveaux. Si les dimensions des entrées ne sont pas à la même échelle alors les courbes de niveaux de l'erreur sont semblables à des ellipses, et la direction orthogonale ne pointe pas vers le centre de l'ellipse, or si la surface d'optimisation est circulaire, la direction orthogonale pointe vers le centre du cercle qui est le minimum global.

Pour plus d'informations sur le sujet, nous avons trouvé cet article très intéressant : https://cseweb.ucsd.edu/classes/wi08/cse253/Handouts/lecun-98b.pdf

2. En pratique

Pour traiter les données, il existe plusieurs méthodes. Nous avons testé deux méthodes :

I. Normaliser les données :

Normaliser les données va mettre à l'échelle chaque entrée pour que la norme 2 de chaque entrée soit égale à 1. Cela ne va pas forcément rendre la surface d'erreur sphérique mais cela va, en général, réduire l'excentricité de l'ellipse.

II. Standardiser les données

La standardisation des données est une méthode qui suit la formule suivante :

$$z = \frac{x - \mu}{\sigma}$$

Avec la moyenne qui vaut : $\mu = \frac{1}{N}\sum_{i=1}^{N} x_i$

Et l'écart-type qui vaut : $\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2}$

Dans le cadre du projet, nous avons pu tester les deux types de traitement de données et étudier l'influence sur la convergence de l'algorithme. Nous avons pu observer que la normalisation des entrées au vecteur unitaire donnait de meilleurs résultats que la standardisation. Les résultats présentés par la suite seront donc tous obtenus après avoir normalisé les données. Chaque vecteur pour chaque document est donc mis à l'échelle individuellement pour avoir une norme 2 égale à 1.

IV. Perceptron

1. Algorithme

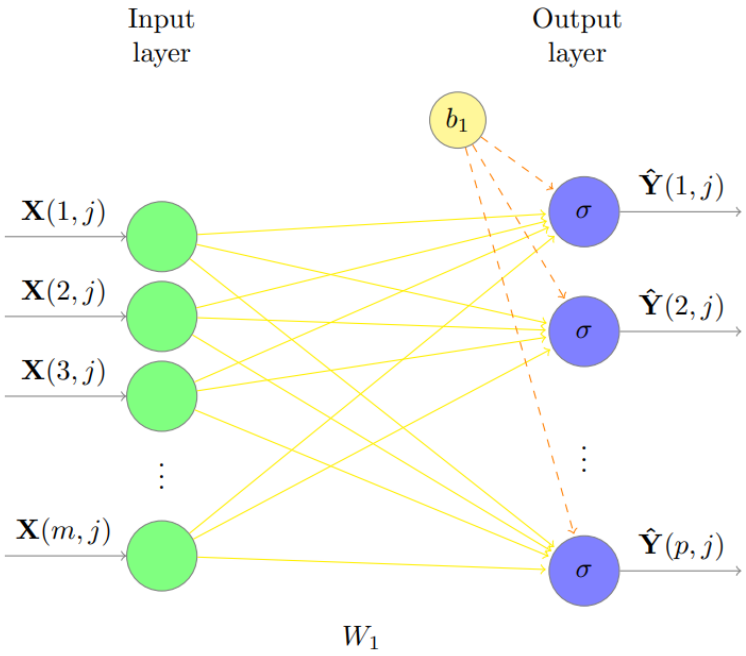

Figure 2 Perceptron

Le perceptron est un algorithme de classification supervisée. Il se base sur une combinaison linéaire des entrées pondérées par des poids, suivie d'une fonction d'activation. La sortie nous donne la classe à laquelle appartient l'entrée et l'objectif est d'ajuster les poids du réseau pour minimiser l'erreur du modèle.

Pour un modèle simple avec une couche d'entrée et une couche de sortie, la sortie du perceptron est caractérisée par l'équation suivante :

$$\sigma({W_1}^T X + B_1)$$

On multiplie l'entrée X par la matrice de poids et on y ajoute le biais, cela nous donne un vecteur de dimension n avec n qui est le nombre de neurones de la couche de sortie (20 dans notre cas car 20 classes). On utilise ensuite une fonction d'activation élément par élément et cela nous donne la valeur en sortie de l'algorithme. Comme nous sommes dans le cas d'apprentissage supervisé, nous connaissons la classe des entrées, nous pouvons donc comparer la vraie classe de l'entrée par rapport à la classe prédite et cela va nous donner l'erreur.

Les paramètres d'un perceptron sont :

- Les **poids** représentés sous la forme d'une matrice W de dimension m*n avec m qui est la taille du vecteur d'entrée et n qui est le nombre de neurones de la couche finale.
- Les **biais** représentés sous la forme d'un vecteur b de dimension n qui est le nombre de neurones de la dernière couche

Lors de l'entraînement, il faudra réussir à ajuster les biais et les poids pour que le modèle fasse les meilleures prédictions possibles, il faudra donc minimiser l'erreur.

2. Initialisation des paramètres

Il existe plusieurs méthodes pour initialiser les paramètres d'un perceptron :

- **Initialisation aléatoire** : on initialise les poids de façon aléatoire, en général entre -1 et 1.
- **Initialisation aléatoire gaussienne** : les poids sont initialisés de façon aléatoire en suivant une distribution normale
- **Initialisation uniforme de Xavier** : les poids sont initialisés de manière aléatoire selon une distribution uniforme dans un intervalle qui dépend de la taille de l'entrée et de la sortie du réseau.

  Les bornes sont calculées suivant : $x = \sqrt{\frac{6}{n+m}}$ avec n qui est la dimension de l'entrée du réseau et m la dimension de la sortie.
- **Initialisation à partir de poids déjà entrainés** : nous avons implémenté la possibilité de charger des poids qui ont été obtenu après un entraînement.

L'initialisation des paramètres est un facteur important pour la convergence de l'algorithme car cela va définir le point de départ.

Dans le cadre du projet, la méthode d'initialisation qui a donné les meilleurs résultats est la méthode Xavier. Les résultats présentés par la suite sont donc toujours obtenus avec cette méthode.

3. Fonction d'activation

La fonction d'activation est une fonction mathématique appliquée en sortie d'un neurone qui permet de calculer une valeur élément par élément pour la sortie de couche du réseau.

Dans le cadre du projet, nous avons travaillé avec la fonction d'activation sigmoïde.

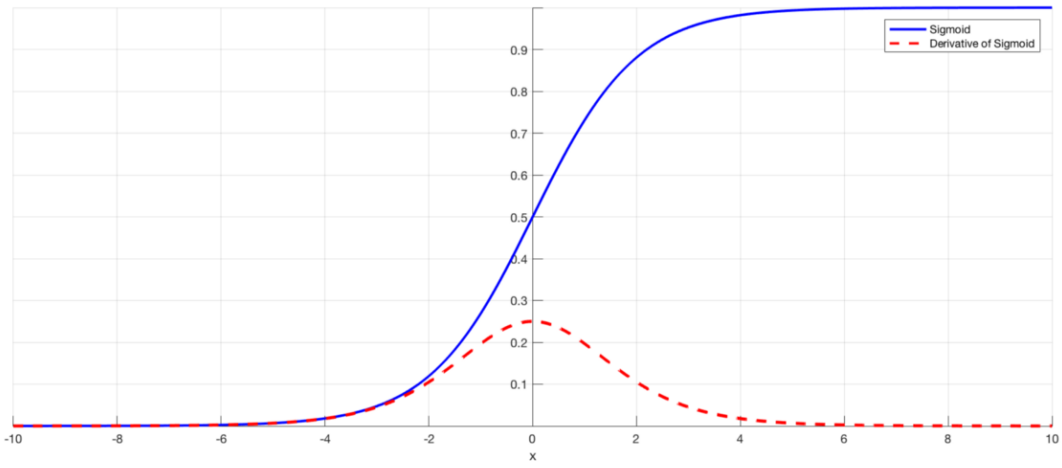

Figure 3 Fonction d'activation sigmoïde et dérivée

Nous avons toujours travaillé avec le paramètre alpha fixé à 1.

La formule de la sigmoïde est : $\frac{1}{1+e^{-\alpha z}}$ avec z qui vaut $X^T W + B$.

Quand la valeur de Z est grande, alors l'exponentielle est proche de 0 donc la sortie de la fonction d'activation est proche de 1, alors que lorsque Z est petit, la valeur de l'exponentielle est très grande et la sortie de l'activation est proche de 0.

Les fonctions d'activation sont non-linéaires, ce qui est important pour le fonctionnement des réseaux de neurones car cela permet d'approximer des phénomènes non linéaires.

Si on travaillait seulement avec des fonctions linéaires, toutes les couches d'un réseau de neurones profonds pourraient être rassemblées en une seule couche. Les réseaux de neurones profonds seraient donc moins performants et ne seraient plus profonds mais simplement des classifieurs linéaires.

Sans la non-linéarité qui est introduite grâce aux fonctions d'activation, on aurait un simple classifieur linéaire : $y = f(W_1 W_2 W_3 x) = f(Wx)$ avec W qui correspond à la matrice des poids et des biais du réseau.

Grâce à l'introduction des fonctions d'activation non linéaires après chaque transformation linéaire, ce problème est résolu et on peut avoir des réseaux profonds qui approximent des fonctions plus complexes. On a maintenant $y = f_1(W_1 f_2(W_2 f_3 W_3 x)))$. On peut donc construire des réseaux de neurones profonds où chaque couche se base sur la sortie de la couche précédente.

4. Erreur

Il existe plusieurs types d'erreurs, dans le cadre du projet nous avons travaillé avec la Mean Squared Error (MSE). L'erreur est définie par :

$$L(Y, Y') = \frac{1}{2n} \|Y - Y'\|_F^2$$

Avec Y qui est la vraie classe et Y' la prédiction. Pour comparer les deux, comme la prédiction est un vecteur de dimension de la sortie, la vraie classe en encodée au format one hot encoding. Par exemple, s'il y a 5 classes en sortie du réseau et que la vraie classe est 2 alors on encode cela sous la forme d'un vecteur de dimension 5 : [0 1 0 0 0] avec un 1 pour la classe numéro 2.

Après avoir calculé l'erreur, l'objectif étant de trouver les meilleurs poids pour minimiser l'erreur, nous allons mettre à jour les poids en utilisant la descente du gradient.

Voici l'expression du gradient de l'erreur en fonction de W_1 :

$$\frac{\partial \, \mathrm{L}}{\partial \, W_1} = X \left\{ \frac{1}{n} \left(\sigma\left({W_1}^T X + B_1 \right) - Y \right) * \sigma'({W_1}^T X + B_1) \right\}^T$$

En fonction de B, l'expression est très similaire :

$$\frac{\partial \, \mathrm{L}}{\partial \, B_1} = \left\{ \frac{1}{n} \left(\sigma\left({W_1}^T X + B_1 \right) - Y \right) * \sigma'({W_1}^T X + B_1) \right\}^T$$

Pour mettre à jour les paramètres d'un perceptron, plusieurs techniques existent. Dans la prochaine partie du rapport, nous allons explorer les différentes options qui existent.

6. Méthode d'optimisation

Il existe plusieurs méthodes pour mettre à jour les poids et les biais d'un réseau de neurones. La méthode choisie joue un rôle important pour la vitesse de convergence. Dans le cadre du projet, nous avons implémenté 5 méthodes d'optimisation :

- Descente de gradient
- Descente de gradient stochastique
- Descente de gradient stochastique par mini-batch
- Descente de gradient accéléré
- Adam

Descente de gradient

La descente de gradient consiste à mettre à jour les poids et les biais selon les formules suivantes :

$$W_{k+1} = W_k - \alpha_k \nabla f(W_k)$$

$$B_{k+1} = B_k - \alpha_k \nabla f(B_k)$$

Les gradients selon W et selon B sont calculés comme expliqué au-dessus, on a donc

$$\frac{\partial \mathrm{L}}{\partial W_1} = X \left\{ \frac{1}{n} \left(\sigma\left({W_1}^T X + B_1 \right) - Y \right) * \sigma'\left({W_1}^T X + B_1 \right) \right\}^T$$

$$\frac{\partial \mathrm{L}}{\partial B_1} = \left\{ \frac{1}{n} \left(\sigma\left({W_1}^T X + B_1 \right) - Y \right) * \sigma'\left({W_1}^T X + B_1 \right) \right\}^T$$

L'idée est de faire des pas répétés dans la direction opposée du gradient de la fonction au point actuel, car c'est la direction de la descente la plus raide. Cela permet de minimiser la fonction objectif qui est la MSE dans notre cas.

Les poids sont initialisés avec la méthode Xavier et pour chaque époque, on met à jour les paramètres. Le problème de cette méthode est qu'elle est assez couteuse étant donné qu'on se base sur l'information de toutes les données pour calculer le gradient et mettre à jour les poids et les biais. En effet, la descente de gradient est appliquée une fois sur toutes les données pour chaque époque. Cette méthode est plus lente que les autres méthodes présentées par la suite.

Descente de gradient stochastique

Pour diminuer la complexité de calcul, au lieu d'appliquer une descente de gradient par époque pour toutes les données, on va l'appliquer pour chaque entrée du jeu de données séparément.

$$W_{k+1} = W_k - \alpha_k \nabla f_k(W_k)$$

$$B_{k+1} = B_k - \alpha_k \nabla f_k(B_k)$$

Avec f_k qui est choisi aléatoirement parmi les données. Pour chaque époque, on va sélectionner une entrée aléatoirement et mettre à jour les paramètres par rapport à cette entrée. On peut donc calculer le gradient seulement pour le terme qui nous intéresse au lieu de calculer le gradient pour tous les termes. Toutes les données du jeu de données doivent être utilisées à chaque époque, au lieu de mettre à jour une fois les paramètres pour une époque, on met à jour n fois les paramètres par époque avec n qui est la taille des données.

Il est important de s'assurer que chaque point est bien pris à chaque époque.

Le problème de cette méthode est qu'on se base sur l'information d'un seul point de données pour mettre à jour les paramètres.

Descente du gradient stochastique par mini-batch

Au lieu d'utiliser l'information d'un seul point pour calculer le gradient, le mini-batch consiste à diviser le jeu de données en plusieurs mini-batch. On calculera alors le gradient pour chaque mini-batch pour mettre à jour les paramètres du réseau. Cela nous permet de résoudre le problème de complexité de calcul en calculant le gradient seulement pour les termes du mini-batch et non pas pour tous les termes. De plus, au lieu de prendre l'information de seulement une entrée, on se base sur plusieurs entrées, ce qui donné en général de meilleurs résultats.

Les avantages de la descente de gradient stochastique sont :

- Souvent beaucoup plus rapide que la descente de gradient
- Donne de meilleures solutions
- Moins coûteuse en calcul

Gradient accéléré

SGD a des difficultés à converger vers le minimum global lorsque les courbes de surface sont bien plus raides dans une dimension que dans une autre, ce qui est courant proche des minimums locaux.

Dans ces scénarios, SGD a tendance à osciller et à converger plus lentement.

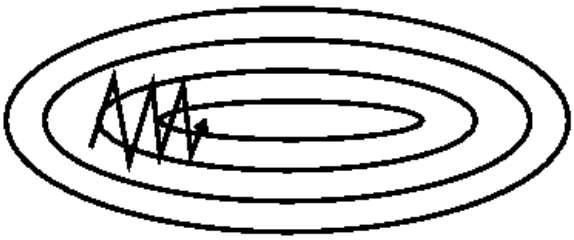

Figure 4 SGD

Lors de l'introduction, nous avions expliqué que la transformation des données pouvait aider à la convergence. Une autre solution souvent utilisée en pratique est d'ajouter un Momentum. Le Momentum permet de prendre en compte l'information des anciennes itérations pour mettre à jour le gradient et de ne pas utiliser seulement l'information locale. On crée un vecteur v fonction des gradients précédents en donnant une importance plus grande aux gradients les plus récents.

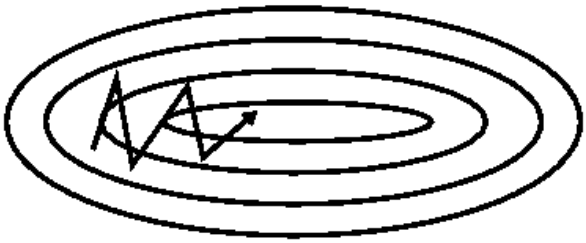

Figure 5 SGD + Momentum

Le gradient accéléré a pour objectif de prendre une information plus générale que simplement prendre la valeur du gradient localement. Pour cela, les gradients des itérations précédentes sont aussi pris en compte. Cela permet de réduire les oscillations et d'accélérer la vitesse de convergence.

Pour cette méthode, il faut tout d'abord calculer les moments :

$$v_{dW} = \beta v_{dW} + (1 - \beta)\mathrm{dW}$$

$$v_{dB} = \beta v_{dB} + (1 - \beta)\,\mathrm{db}$$

On met ensuite à jour les poids suivant la formule :

$$\mathrm{W} = \mathrm{W} - \alpha \,.\, v_{dW}$$

$$\mathrm{b} = \mathrm{b} - \alpha \,.\, v_{dB}$$

Adam

Adam est une méthode qui permet de combiner le Momentum ainsi que le pas d'apprentissage adaptatif pour chaque paramètre. En plus de stocker une moyenne décroissante exponentielle des gradients au carré précédents, Adam conserve aussi une moyenne décroissante exponentielle des gradients passés, de façon semblable au Momentum. Cette méthode est très intéressante et donne souvent de très bons résultats.

Pour cette méthode, on doit calculer les moyennes décroissantes des précédents gradients :

$$m_{dW} = \beta 1 m_{dW} + (1 - \beta 1)\mathrm{dW}$$

$$m_{dB} = \beta 1 m_{dB} + (1 - \beta 1)\,\mathrm{db}$$

On doit calculer ensuite les moyennes décroissantes des précédents gradients au carré :

$$v_{dW} = \beta 2 v_{dW} + (1 - \beta 2)\, dW^2$$

$$v_{dB} = \beta 2 v_{dB} + (1 - \beta 2)\, db^2$$

On doit ensuite calculer les valeurs corrigées :

$$m_{dW\_corr} = \frac{m_{dW}}{(1 - \beta 1^t)}$$

$$m_{db\_corr} = \frac{m_{db}}{(1 - \beta 1^t)}$$

$$v_{dW\_corr} = \frac{v_{dW}}{(1 - \beta 2^t)}$$

$$v_{dB\_corr} = \frac{v_{dB}}{(1 - \beta 2^t)}$$

On peut ensuite mettre à jour les poids et les biais :

$$W = W - \frac{\alpha}{\left(\sqrt{v_{dW} + epsilon}\right)} * m_{dW\_corr}$$

$$b = b - \frac{\alpha}{\left(\sqrt{v_{db} + epsilon}\right)} * m_{db\_corr}$$

Les paramètres utilisés pour Adam lors de ce projet, sont les paramètres conseillés dans le papier de recherche sorti pour cette méthode : $\beta 1 = 0.9,\ \beta 2 = 0.99,\ epsilon = 1e - 8.$

D'autres algorithmes d'optimisation existent mais dans le cadre du projet nous nous sommes concentrés sur ceux-ci.

7. Pas d'apprentissage

Le pas d'apprentissage intervient dans toutes les méthodes d'optimisation et il joue un rôle important pour la convergence de l'algorithme. Si le pas choisi est trop petit, la convergence risque d'être très lente, à l'inverse, si le pas est trop grand, il y a un risque d'oscillations autour du minimum global et on pourrait ne jamais atteindre le minimum.

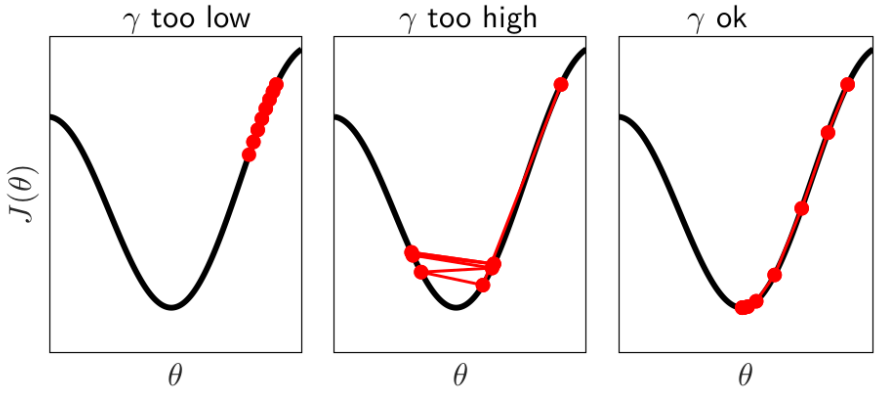

Figure 6 Visualisation du learning rate

En pratique, il est souvent intéressant de commencer l'optimisation avec un pas d'apprentissage assez grand car cela permet de se rapprocher rapidement du minimum, cependant, plus on avance dans l'apprentissage, plus il est intéressant de diminuer le pas pour mieux converger vers le minimum et éviter de trop se déplacer d'un coup.

Dans la littérature, plusieurs méthodes existent pour mettre à jour le pas d'apprentissage :

- **Time decay** : cette méthode consiste à mettre à jour le pas d'apprentissage à chaque époque en fonction d'un coefficient défini.

$$\alpha = \frac{pas\ initial}{epoch * \epsilon}$$

- **Step decay** : au lieu de diminuer le pas à chaque époque, on peut le diminuer après un certain nombre d'époques.

$$\alpha = \frac{pas\ initial}{nombre\ epoque * \epsilon}$$

- **Exponential decay** : le pas décroît de manière exponentielle

$$\alpha = pas\ initial * \exp(-\epsilon * ep)$$

Le pas d'apprentissage joue un rôle important pour la vitesse de converge de l'algorithme.

Pour bien visualiser l'importance du pas d'apprentissage, nous avons lancé le même modèle avec exactement les mêmes paramètres, le premier en utilisant Adam et un pas d'apprentissage de 0.001 et le deuxième avec Adam et un pas d'apprentissage de 3^{e}-4.

Voici les résultats obtenus, comme prévu, nous pouvons remarquer que le pas d'apprentissage plus grand va converger plus rapidement et mais suite à l'early stopping, il va s'arrêter plus rapidement car il aura tendance à overfit plus rapidement aussi. Le pas d'apprentissage plus petit va converger plus lentement et dans ce cas-ci, il donnera de meilleurs résultats.

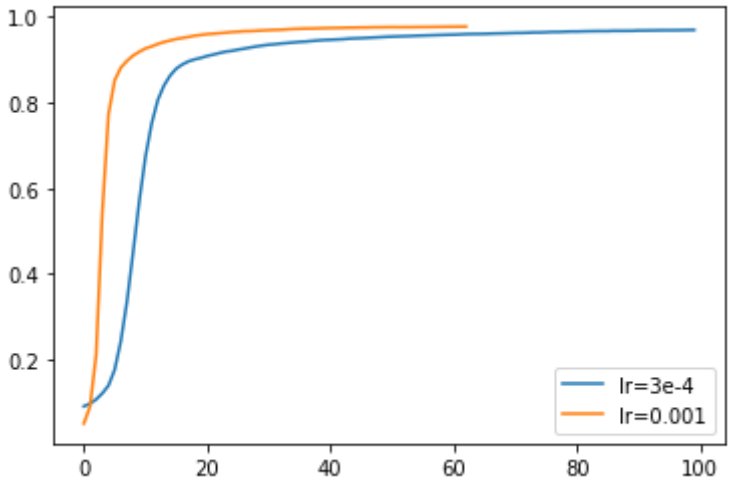

Figure 7 Influence du pas d'apprentissage

Dans le cadre du projet, les meilleurs résultats ont été obtenus avec l'optimiseur Adam et un pas d'apprentissage de 3^{e}-4 (cette valeur suit les conseils donnés par Andrew Karpathy dans cet article de blog : https://karpathy.github.io/2019/04/25/recipe/).

V. Implémentation et comparaison des résultats

Early Stopping

Nous avons implémenté les optimiseurs suivants :

- Descente de gradient
- Descente de gradient stochastique
- Descente de gradient stochastique par mini-batch
- Descente de gradient accéléré
- Adam

Cela nous a permis de tester différents paramètres et de comparer leur vitesse de convergence et les résultats obtenus avec chacun. Pour évaluer nos modèles, nous avons séparé le jeu de données entre un jeu de validation et un jeu d'entraînement. Cela nous a permis de mettre en place l'Early Stopping, une technique de régularisation pour éviter le surapprentissage.

L'**Early Stopping** permet d'éviter le surapprentissage durant l'entraînement d'un modèle. Pour cela, à chaque itération, on peut calculer la précision et la loss du modèle mais aussi la précision et la loss sur les données de validation pour voir comment le modèle généralise sur des données non vues.

Pour ce projet, nous avons travaillé avec un early stopping fixé à 5 sur la loss de validation. Cela veut dire que si la loss de validation augmente pendant 5 époques de suite sans jamais s'améliorer, alors on arrête l'entraînement pour éviter que le modèle overfit et qu'il commence à « apprendre » les données d'entraînement par cœur.

Analyse des résultats

Meilleurs résultats

Les meilleurs résultats ont été obtenus avec l'initialisation Xavier, l'optimiseur Adam, un pas d'apprentissage de 3^{e}-4, la MSE comme erreur et comme fonction d'activation la sigmoïde avec un paramètre fixé à 1.

Pour trouver les meilleurs paramètres qui nous ont permis d'obtenir les meilleurs résultats, nous avons divisé notre jeu de données en un jeu de données d'entraînement et un jeu de données de validation. Grâce à cela, nous avons pu lancer les différents optimiseurs et analyser les résultats obtenus en fonction des paramètres choisis. Nous avons pu fine tuner les paramètres à la main en fonction des résultats observés.

Pas d'apprentissage

Adam donne les meilleurs résultats parmi tous les optimiseurs. Pour le pas d'apprentissage nous avons pu en tester plusieurs entre 1 et 3^{e}-4. Quand le pas d'apprentissage est plus grand, la convergence se fait beaucoup plus rapidement, après quelques époques, nous sommes déjà à environ 95% de précision pour les données d'entrainement, cependant, si le pas reste constant à 1, alors, on ne va jamais converger vers le minimum global, on va simplement osciller autour. Le pas d'apprentissage qui nous a donné les meilleurs résultats est donc 3^{e}-4 car il permet d'éviter les grandes oscillations, par contre, comme il est beaucoup plus petit, la convergence se fait aussi plus lentement.

Batch size

Pour la batch size, nous avons testé une batch size de 8, 16, 32, 64, 128, 256 et 512. Nous n'avons pas remarqué de différence au niveau des résultats. La batch size influence très peu la qualité des résultats,

cependant, elle influence forcément la vitesse de convergence car elle détermine le nombre de mise à jour des paramètres du réseau qui est fait lors de chaque époque.

Séparation des données

Au niveau de la séparation des données entre entraînement et validation, nous avons travaillé avec 10% pour la validation, cela n'influence pas les résultats obtenus.

Paramètres d'Adam

Pour les paramètres d'Adam, nous avons travaillé avec les paramètres recommandés dans le papier de recherche de cette méthode. Nous avons testé de réduire les valeurs de Beta mais cela n'a pas changé la qualité des résultats dans notre cas.

Initialisation des paramètres

Pour l'initialisation des paramètres, la méthode Xavier donne une convergence plus rapide que l'initialisation selon une distribution gaussienne.

Fonction d'activation et erreur

La fonction d'activation sigmoïde est celle qui a donné les meilleurs résultats, combinée à la Mean Squared Error.

Early Stopping

Avant l'implémentation de l'Early Stopping, nous avions tendance à overfit, grâce à cette méthode de régularisation, nous avons pu éviter cela. Pour le choix du nombre d'époques, nous lançons simplement 1000 époques et nous laissons l'early stopping arrêter l'algorithme au bon moment.

Normalisation des données

La normalisation des données nous a permis de passer de 79-80% à 81-82%. La standardisation quant à elle n'a pas donné de si bons résultats.

Optimiseurs

Pour les optimiseurs, comme attendu, la descente de gradient converge beaucoup plus lentement que les autres méthodes. Il n'est donc jamais intéressant d'utiliser la descente de gradient classique. La méthode qui donne les meilleurs résultats est Adam.

Adam est la méthode la plus robuste, les paramètres choisis jouent principalement sur la vitesse de convergence, mais en utilisant Adam, avec différents paramètres, nous avons obtenu au moins 10 résultats supérieurs à 81.5% sur Kaggle.

Kaggle

Le meilleur résultat que nous avons pu obtenir sur le site Kaggle est 82.16% avec une seule couche et les paramètres cités juste avant.

Comparaison entre les différents optimiseurs

Nous n'avons pas stocké les valeurs de toutes nos expériences, mais voici les résultats obtenus pour différents paramètres testés sur les données d'entraînement et de validation.

Optimiseur	Learning Rate	Train accuracy	Validation Accuracy
Adam	3^{e}-4	97.6	86.2
Adam	0.01	97.6	85.8
SGD	0.01	95.8	83.3
SGD batch	0.01	96.5	84.2
GD	0.01	95.4	83.1
Momentum	0.01	96.6	84.7

Perspectives d'amélioration

Nous avons réussi à implémenter un perceptron une couche qui donne 82.16% comme précision sur Kaggle, ce qui nous place en quatrième position. Nous avons identifié plusieurs pistes qui pourraient nous permettre d'obtenir de meilleurs résultats.

Utilisation de plusieurs couches

A nos connaissances, 2 des 3 équipes devant nous ont implémenté au moins une couche cachée. Cela leur permet donc d'obtenir de meilleurs résultats car ils ont un réseau plus large. Nous avons tenté d'implémenter plusieurs couches pour ce projet, cependant, le problème est que tout est implémenté en python et en numpy. La première matrice de poids est de dimension 43586*20 et, lorsque nous avons essayé d'implémenter une deuxième couche de taille 100, la deuxième matrice étant de taille 100*20, notre ordinateur n'avait pas assez de mémoire RAM pour stocker les matrices de poids. Il aurait donc fallu utiliser une machine plus puissante pour gérer plusieurs couches en utilisant python/numpy, ou bien nous aurions dû utiliser un framework comme Pytorch qui stocke les matrices au format Tensor pour optimiser la mémoire, mais cela dévie du but premier du projet qui était de s'intéresser à l'aspect mathématique du perceptron et aux différentes méthodes de descente de gradient existantes.

Trouver des meilleurs paramètres

Une autre possibilité pour améliorer la solution serait de mieux fine tuner les paramètres du modèle. Comme expliqué, nous avons testé différentes configurations en modifiant le pas d'apprentissage, les paramètres des optimiseurs ou encore les batch size pour le calcul de gradient, mais il est probable que de meilleurs paramètres existent.

Utiliser un autre optimiseur

Une autre possibilité pour améliorer les résultats serait d'utiliser un autre optimiseur. Nous avons implémenté les 5 optimiseurs cités juste avant, mais il en existe d'autres. Il se peut qu'un autre optimiseur non testé donne de meilleurs résultats.

Faire du pre-processing sur les données avant l'entraînement

Enfin, après avoir analysé les données et remarqué que la matrice était très creuse, nous avons essayé de réduire la dimension des entrées. Pour cela, nous avons tenté de faire une analyse en composantes principales pour identifier les mots qui jouaient un rôle important et ne considérer que ces mots

comme entrées du réseau. Malheureusement, les résultats obtenus avec cette méthode n'étaient pas concluants, il aurait été intéressant de tester la PCA avec différents paramètres, mais cela prend beaucoup de temps à cause de la taille des données.