Réseaux de neurones

Ismat Benotsmane

Novembre 2020

1 Formalisation mathématique

1.1 Jeu de données

(Q1) L'ensemble d'apprentissage sert à apprendre notre modéle en optimisant ses paramètres. Cet ensemble ne doit pas contenir des éléments de l'ensemble de validation ou de test.
L'ensemble de validation permet d'optimiser les hyper-paramètres du modéle(learning rate, nombre de neurones,...) en choissisant ceux qui permettent d'avoir le meilleur score en validation. De plus, la validation nous permet d'observer si notre modéle n'est pas en sur-apprentissage dans le cas où le coût en apprentissage diminue mais en validation il augmente.
L'ensemble de test sert à tester notre modéle et évaluer sa performance sur un ensemble d'example qui lui sont inconnu.

(Q2) Plus le nombre d'example augmente, plus on peut mieux apprendre notre modéle sur l'ensemble d'apprentissage, et plus le modéle sera robuste.
Le nombre de N examples ideal est détérminé par le fait que si à chaque fois on ajoute des examples à notre dataset et que l'accuracy de notre augmente en conséquent, alors on continue d'ajouter jusqu'au moment où l'ajout ne permet plus d'améliorer la performance du modéle.

A noter qu'il est important d'avoir un jeu de données representatif des données réelles sur lesquelles le modéle sera amené à rencontrer. Car dans ce cas là, on aura beau avoir un dataset conséquent mais s'il est consituté uniquement d'exemple particulier, il ne sera pas performant en réalité.

1.2 Architecture du réseau (phase forward)

(Q3) Chaque couche de notre réseau de neurones est representé par une matrice. Une matrice est une fonction affine, et la composition de fonction affine est une fonction affine. Le but d'un réseau de neurones est de faire une classification sur des problèmes non-linéaires, or juste avec le produit des matrices on ne peut résoudre que les problèmes lineaires.
Le role de l'activation non-linéaire est de séparer l'espace de façon non-lineaire pour ensuite pouvoir separer notre dataset de façon linéaire. Ainsi cela nous permettera de résoudre des problèmes de grande compléxité.

(Q4) Sur la figure 1 on a $n_x = 2$, $n_h = 4$ et $n_y = 2$.
n_x représente le nombre de features de notre ensemble d'apprentissage, comme dans le cas du *BOW* le dataset a 1000 features qui correspondent à la taille du dictionnaire. n_x est donc déterminé par le pré-processing des données.

n_y représente la taille de la prédiction, dans le cas d'une classification entre chien et chat par exemple on aurait une prédiction de taille 2 et dans le cas d'une prédiction d'une valeur de température on aurait une prédiction de taille 1. n_y est donc déterminé par la nature du probléme qu'on doit résoudre.

n_h représente le nombre de neurones dans la couche caché, c'est un hyper-paramétre du modéle. Il est choisi par validation. Si n_h est trés grand on va pas compresser assez l'information et on sera en sur-apprentissage et s'il est trop petit, on va trop compresser l'information et par la meme occasion éliminer des détails important. Dans ce cas là, on sera en sous-apprentissage.

(Q5) y représente le label d'un élément x de notre dataset, et $\hat{y}$ représente le label prédit par notre modéle.
y est un vecteur en encodage one-hot de taille correspondant au nombre de classe où l'on met 1 à la classe à laquelle il correspond et 0 au reste. $\hat{y}$ est un vecteur de la même taille que y mais à chaque classe la valeur correspond à la probabilité d'appartenir à cette derniére.

(Q6) La fonction $SoftMax$ renvoie une distribution de probabilité des classes en sortie. Cette fonction est une approximation de la fonction Max qui n'est pas une fonction differentiable contrairement à la fonction $SoftMax$. Voilà pourquoi on utilise cette derniére dans les réseaux de neurones.

(Q7)

$$\tilde{h}_i = \sum_{j=1}^{n_x} w_{ij}^h x_j + b_i^h \iff \tilde{h} = W_h x + b_h$$

$$h = \tanh(\tilde{h})$$

$$\tilde{y}_i = \sum_{j=1}^{n_h} w_{ij}^y h_j + b_i^y \iff \tilde{y} = W_y h + b_y$$

$$\hat{y} = SoftMax(\tilde{y})$$

1.3 Fonction de coût

(Q8) Dans le cas de la MSE, pour réduire le coût il faut que la valeur de $\hat{y}$ tende vers y. Dans le cas de la *Cross Entropy*, il faut que la probabilité qu'on affecte à la classe réelle dans $\hat{y}$ soit la plus proche de 1 possible.

(Q9) La *Cross Entropy* et la MSE sont des fonctions de coût trés adapté aux problèmes de classification et de régréssion car plus vous etes mal classé ou loin de la valeur réelle et plus vous etes fortement pénalisé.

Dans le cas de la *Cross Entropy*, si on affecte en prédiction une probabilité proche de 0 pour la classe réelle on va tendre vers $-\infty$ et donc avoir un coût trés élevé, et à l'opposé si on affecte une probabilité proche de 1 pour la classe réelle on va tendre vers 0 et notre coût sera proche de 0.

De même pour la MSE, on va fortement pénaliser les distances en les mettant au carré. De ce fait, si nous prédiction est vraiment trés loin de la valeur réelle, elle sera encore plus pénaliser.

1.4 Méthode d'apprentissage

(Q10) L'avantage du batch c'est d'avoir une descente de gradient plus stable, avec un gradient le plus optimal et representatif possible. Mais le temps de calcul est énorme, et dans certains cas il est impossible de prendre tout le dataset en entrée lorsque ce dernier est trés grand.

L'avantage du stochastique c'est qu'on va être trés rapide en calcul mais comme on va tiré au hasard un exemple, et chaque exemple est bruité, notre gradient va osciller et va convergé plus lentement.

Le mini-batch est un compromis entre les deux, avec un bon temps de calcul tout en conservant une descente de gradient stable. Il permet d'avoir aussi un gradient légerement bruité qui va permettre de mieux généraliser en test, au contraire de la méthode batch.

(Q11) Le learning rate η s'il est trop petit on va convergé trés lentement vers la solution optimal, donc il faudra plus de temps pour apprendre. Il peut aussi avoir un autre effect où on reste bloqué dans un minimum local et donc on va moins bien optimiser.

Au contraire si η est trop grand, on va faire des mise-à-jour trés grande et on va divergé, donc avoir un modéle de moins au moins bon.

(Q12) L'approche naive du calcul de gradient est trés coûteuse car à chaque couche on va calculer l'ensemble des gradients sur toutes les couches précendentes. Alors que l'algorithme de la **backporpagation** calcul seulement le gradient de la valeur de gradient de la couche précédente qui a été calculé précedemment et enregistré.

(Q13) Le critère que doit respecter l'architecture du réseau pour permettre la backpropagation est d'avoir uniquement des fonctions differentiables.

(Q14)

$$
\begin{aligned}
l(y,\hat{y}) &= l(y, SoftMax(\tilde{y})) \\
&= -\sum_i y_i \log(\frac{e^{\tilde{y}_i}}{\sum_j e^{\tilde{y}_j}}) \\
&= -\sum_i (y_i \tilde{y}_i - y_i \log(\sum_j e^{\tilde{y}_j})) \\
&= -\sum_i y_i \tilde{y}_i + \sum_i y_i \log(\sum_j e^{\tilde{y}_j}) \quad \text{avec } \sum_i y_i = 1 \text{ car } y \text{ vecteur one-hot} \\
&= -\sum_i y_i \tilde{y}_i + \log(\sum_j e^{\tilde{y}_j})
\end{aligned}
$$

(Q15)

$$\frac{\partial l}{\partial \tilde{y}_i} = -y_i + \frac{e^{\tilde{y}_i}}{\sum_j e^{\tilde{y}_j}} = -y_i + \hat{y}_i$$

$$\nabla_{\tilde{y}} \mathfrak{l} = \begin{bmatrix} \frac{\partial l}{\partial \tilde{y}_1} \\ \vdots \\ \vdots \\ \frac{\partial l}{\partial \tilde{y}_{n\,y}} \end{bmatrix} = \begin{bmatrix} \hat{y}_1 - y_1 \\ \vdots \\ \vdots \\ \hat{y}_{n\,y} - y_{n\,y} \end{bmatrix}$$

(Q16) On a

$$\frac{\partial \tilde{y}_k}{\partial W_{y,ij}} = h_k$$

et

$$\frac{\partial \tilde{y}_k}{\partial b_{y,i}} = 1$$

Alors,

$$\frac{\partial l}{\partial W_{y,ij}} = \sum_k \frac{\partial l}{\partial \tilde{y}_k} \frac{\partial \tilde{y}_k}{\partial W_{y,ij}} = \sum_k (\hat{y}_k - y_k) h_k$$

$$\frac{\partial l}{\partial b_{y,i}} = \sum_k \frac{\partial l}{\partial \tilde{y}_k} \frac{\partial \tilde{y}_k}{\partial b_{y,i}} = \sum_k (\hat{y}_k - y_k)$$

(Q17)

$$\frac{\partial l}{\partial \tilde{h}_i} = \sum_j \frac{\partial l}{\partial h_j} \frac{\partial h_j}{\partial \tilde{h}_i} = (1 - h_i^2) \sum_j (\hat{y}_j - y_j) W_{y,ji}$$

$$\frac{\partial l}{\partial W_{h,ij}} = \frac{\partial l}{\partial \tilde{h}_i} \frac{\partial \tilde{h}_i}{\partial W_{h,ij}} = (1 - h_i^2) x_j \sum_j (\hat{y}_j - y_j) W_{y,ji}$$

$$\frac{\partial l}{\partial b_{h,i}} = \frac{\partial l}{\partial \tilde{h}_i} \frac{\partial \tilde{h}_i}{\partial b_{h,i}} = (1 - h_i^2) \sum_j (\hat{y}_j - y_j) W_{y,ji}$$

2 Implémentation

2.1 Circles data

La figure 1 est une représentation graphique de notre dataset sur lequel on veut apprendre notre modèle. Les couleurs rouges et bleues caractérisent nos 2 classes à séparer.
Comme on peut le remarquer sur la figure, en l'état on ne peut pas séparer ces 2 classes linéairement, d'où la necéssité de faire appelle à des séparateurs non-linéaires pour résoudre ce genre de probléme et ici en l'occurence un **NN**.

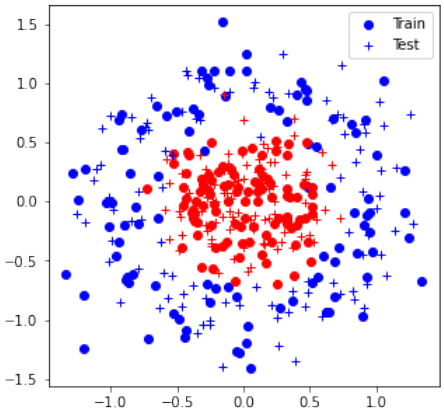

FIGURE 1 – Représentation graphique du dataset

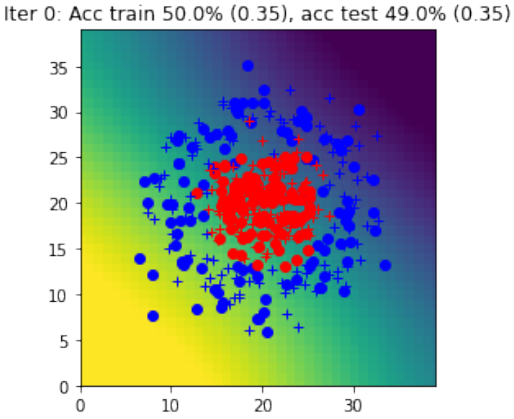

FIGURE 2 – Première itération du SGD

Pour entrainer notre modéle, nous avons utilisé une descente de gradient en *mini-batch* pour optimiser les paramètres du modéle afin de réduire la loss et augmenter l'accuracy en train, tout en observant le comportement de la loss et de l'accuracy en test. Car le but ultime est d'avoir les meilleurs résultats possible en test.

À la première itération comme on le voit à la figure 2, on est à **50%** d'accuracy et **0.35** de loss.

À mi-chemin de la descente de gradient, comme le montre la figure 3 on a réduit la loss en train à **0.28**, et on est passé en test d'une accuracy de **49%** à **85%**. Donc pour le moment la descente de gradient optimise bien le modéle et nous devons poursuivre le processus.

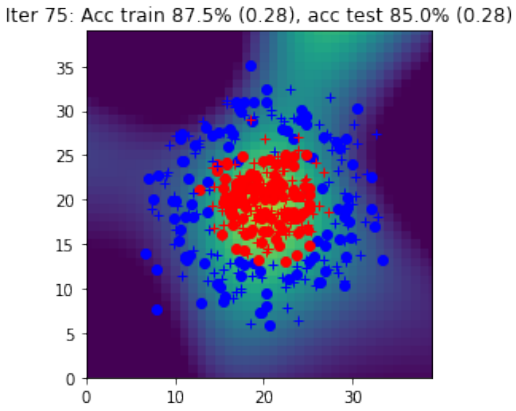

FIGURE 3 – À la moitié du SGD

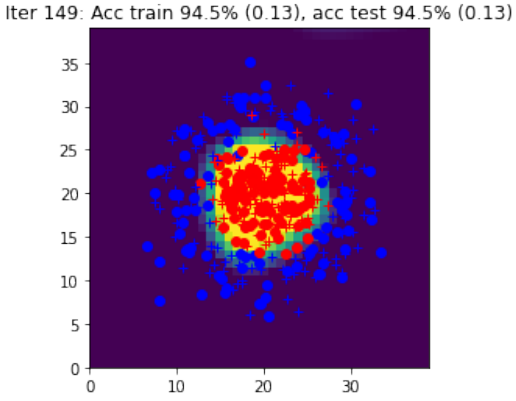

FIGURE 4 – À la fin du SGD

À la derniére itération, on a un classifieur plutôt robuste avec une accuracy de **94.5%** en train et **94.5%** en test.

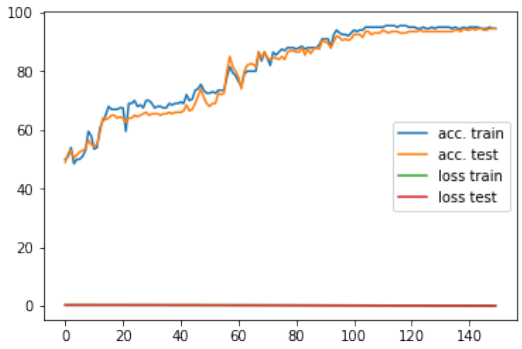

FIGURE 5 – Evolution accuracy/loss pendant le SGD

2.2 MNIST data

Nous avons appliqué notre modèle, un réseau de neurones à une couche caché de 10 neurones au jeu de donnés $MNIST$, composé d'image de taille 28x28 et qui ont été transformé en un vecteur de taille 784.
Le but est d'apprendre à classer ces images parmi les 10 classes.

On remarque avec l'apprentissage du modéle on est passé de **83%** à **95.4%** en test.
À noter qu'à partir de la 90éme itération malgré que la loss en train diminue, mais en test nous avant atteint le planché et l'accuracy en test ne s'améliore plus.

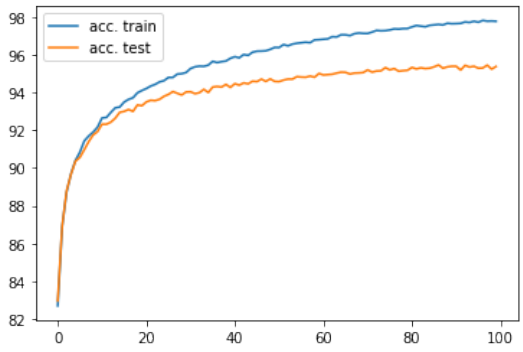

FIGURE 6 – Evolution accuracy en fonction de l'itération

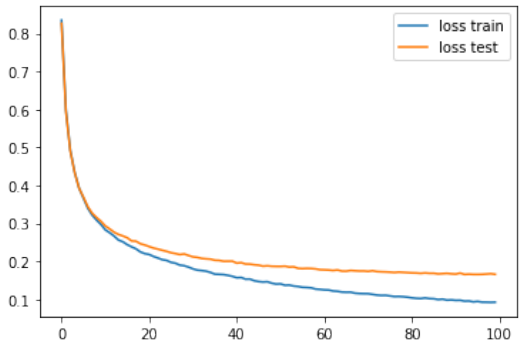

FIGURE 7 – Evolution loss en fonction de l'itération

2.3 SVM

Maintenant nous voulons voir comment un classifieur lineaire classique se comporte sur des problémes non-lineaire, et comparer ça avec ce qu'on appelle les techniques de **Kernel trick**, où l'on va projetter nos données dans un nouvel espace de plus grande dimension pour voir si malgré tout on peut résoudre ces problémes nos-lineaire avec des modéles linaires.

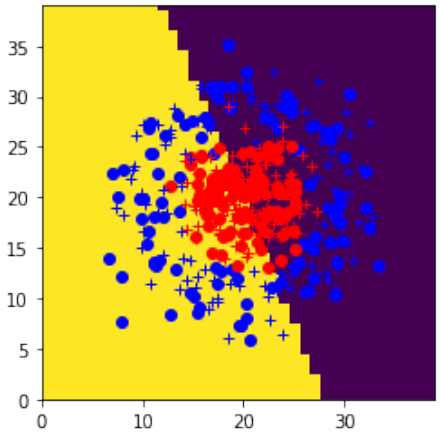

FIGURE 8 – Frontière du modéle SVM avec kernel lineaire

La figure 8 est le résultat du modéle SVM avec un kernel lineaire qu'on appliquer sur les données. On remarque bien qu'il separer les données en deux avec une simple droite en séparant en deux les données, ce qui explique l'accuracy en test qui égale à **53%**.

L'application d'un SVM avec un kernel polynomial de degré 2 à la figure

9 est bien meilleur que le résultat précédent et cela se confirme par les **95.5** d'accuracy en test.

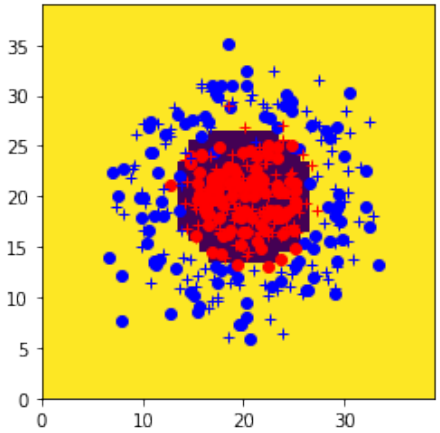

FIGURE 9 – Frontière du modéle SVM avec kernel polynomial

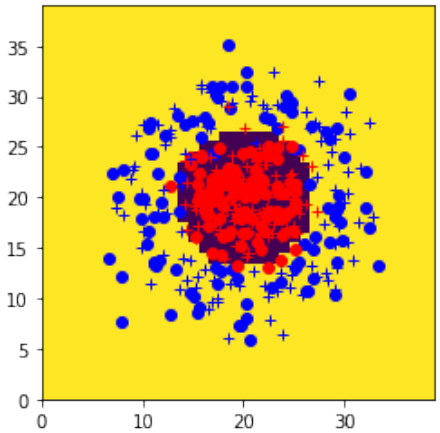

FIGURE 10 – Frontière du modéle SVM avec kernel gaussien

Le résultat est identique avec un kernel gaussien.

On peut donc constater que certains problèmes non-lineaire peuvent se résoudre avec des classifieurs lineaire avec les même résultats que les classifieurs non-lineaire.