Université Mohammed v Rabat
Ecole Nationale Supérieure d'Informatique et d'Analyse des Systèmes
Année universitaire 2020/2021

Mémoire de
Projet de Fin de Deuxième Année

Deuxième année cycle d'ingénieur

Filière Ingénierie Intelligence artificielle

Réseaux de neurones et Apprentissage Profonde

Réalisé par

Abdelhak ELBIARI

Imrane OU EL FAQUIR

Jugé par

Pr. M. LAZAAR	Président
Pr. R. CHIHEB	Président
Pr. A. EL AFIA	Encadrant

Année universitaire : 2020/2021

Remerciement

Avant tout, je rends grâce à Dieu le tout puissant pour l'énergie, l'esprit et l'endurance sans lesquelles ce projet ne serait pas achevé.

J'adresse mes sincères remerciements et profondes gratitudes à mes professeurs, qui n'ont pas cessé de me pousser à aller de l'avant, pour ses innombrables conseils et consignes, ses temps, et la confiance qu'ils m'ont témoigné.

Sans oublier de remercier toute l'équipe pédagogique du département informatique et d'aider à la décision de l'École Nationale Supérieure d'Informatique et d'Analyse des Systèmes, ainsi que tous les intervenants chargés de la formation du cycle ingénieur Ingénierie Intelligence Artificielle (IIA) qui font de leur mieux pour que les étudiants ont une formation solide et parfaite.

Contents

List of Figures

Introduction

Comprendre les dernières avancées en matière d'intelligence artificielle (IA) peut sembler écrasant, mais s'il s'agit d'apprendre les bases qui nous intéressent, et nous pouvons résumer de nombreuses innovations en matière d'IA à deux concepts : l'apprentissage automatique et l'apprentissage en profondeur. L'apprentissage automatique est une application de l'IA qui comprend des algorithmes qui analysent les données, apprennent de ces données, puis appliquent ce qu'ils ont appris pour prendre des décisions. L'apprentissage en profondeur est un sous-domaine de l'apprentissage automatique qui structure les algorithmes en couches pour créer un « réseau de neurones artificiels » qui peut apprendre et prendre des décisions intelligentes par lui-même.

Dans ce contexte, nous avons décidé d'implémenter un Framework « le modèle d'apprentissage en profondeur » pour la classification binaire, pour mieux comprendre qu'est ce qui se passe et avoir l'accès de modifier et optimiser notre modèle d'apprentissage en profondeur.

Le présent rapport comporte quatre chapitres. Dans le premier chapitre nous parlons du modèle mathématique des réseaux de neurones ainsi que son pseudo code. Le deuxième chapitre nous donnons l'architecture et le modèle mathématique des réseaux de neurones profonds, ainsi que son pseudo code. Le troisième chapitre contient les techniques d'optimisation proposé dans notre modèle. Finalement, le quatrième et le dernier chapitre qui contient la documentation du notre Framework ainsi que quelques d'exemple de performance de notre modèle. A la fin, une conclusion résume notre travail.

1 | Les réseaux de neurones

1.1 La régression logistique

La régression logistique est un modèle de régression binaire. Comme pour tous les modèles de régression binaire, l'objectif est de modéliser au mieux un modèle mathématique simple avec de nombreuses observations réelles. Autrement dit associer à un vecteur de variables aléatoires $X_1, \cdots, X_k$ une variable aléatoire binaire notée génériquement Y.

1.1.1 Modélisation mathématique

Soit Y la variable à prédire, et $X = (X_1, X_2, ..., X_{n_x})$ les variables prédictives (variables explicatives). Dans le cadre de la régression logistique binaire, la variable Y prend deux modalités possibles $\{0, 1\}$. Les variables X_i sont exclusivement continues ou binaires.

- Soit Ω un ensemble de n échantillons, comportant n_1 (resp. n_0) observations correspondant à la modalité 1 (resp. 0) de Y.
- $P(Y = 1)$ (resp. $P(Y = 0)$) est la probabilité a priori pour que $Y = 1$ (resp. $Y = 0$).
- $P(X|Y = 1)$ (resp. $P(X|Y = 0)$) est la distribution conditionnelle des X sachant la valeur prise par Y.
- La probabilité a posteriori d'obtenir la modalité 1 de Y (resp. 0) sachant la valeur prise par X est notée $P(Y = 1|X)$(resp. $P(Y = 0|X)$).

Nous devons modéliser une probabilité à l'aide d'une courbe p où le domaine prédicteur X peut être n'importe quoi et la probabilité conditionnelle que y soit vraie étant donné que X est comprise entre 0 et 1.

$$P(y|X) = \sigma(\omega^T X + b) \tag{1.1.1}$$

Avec ω et b sont des paramètres à estimer. Donc si nous arrivons à estimer les paramètres ω et b, nous pouvons prédire la valeur de Y de n'importe quel X donné via la formule suivant:

$$y_{pred}(X) = \begin{cases} 1 & si \quad P(y|X) \geq 0 \\ 0 & sinon \end{cases}$$

Nous devons estimer ω et b de sorte que:

- Pour les échantillons étiquetés 1 : Estimez ω tel que $P(X)$ est aussi proche de 1 que possible.

- Pour les échantillons étiquetés 0 : Estimez ω tel que $1-P(X)$ est aussi proche de 1 que possible.

Il existe de nombreuses façons d'y parvenir, la régression logistique utilise la fonction logistique. En fait une fonction sigmoïde spécifique pour accomplir cela, donc en utilisant cette fonction.

$$P(y|X) = \sigma_{\omega,b}(X) = \frac{1}{1+e^{-(\omega^T X+b)}} = \frac{1}{1+e^{-z}} \tag{1.1.2}$$

1.1.2 La fonction coût

Pour entraîner les paramètres ω et b du modèle de régression logistique, nous devons définir une fonction de coût σ pour prédire l'étiqueter des nouveaux X.

Pour chaque échantillon avec l'étiquette 1, nous voulons estimer ω de telle sorte que le produit de tous les échantillons de probabilités conditionnelles de classe 1 $\prod_{y_i=1} P(X_i)$ soit aussi proche de 1 que possible. De même pour les échantillons étiquetés 0, nous voulons estimer ω de telle sorte que le produit du complément de leurs probabilités conditionnelles $\prod_{y_i=0}(1-P(X_i))$ soit le plus proche de 1 possible. En combinant ces exigences, nous voulons trouver ω de telle sorte que le produit de ces deux produits soit maximum sur tous les éléments de l'ensemble de données.

$$L(\omega,b) = \prod_{y_i=1} P(X_i) \times \prod_{y_i=0} (1-P(X_i)) \tag{1.1.3}$$

Nous combinons les produits et nous obtenons la fonction coût à maximiser.

$$\begin{aligned} L(\omega,b) &= \prod_{\Omega} P(X_i)^{Y_i} \times (1-P(X_i))^{1-Y_i} \\ &= \sum_{i=1}^{n} Y_i \log(P(X_i)) + (1-Y_i)\log(1-P(X_i)) \end{aligned}$$

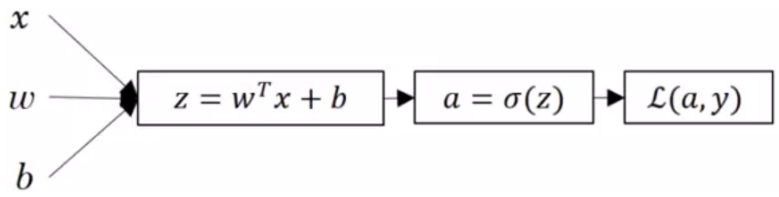
Le but est donc de trouver la valeur de ω et de b qui satisfasse le problème.

$$\begin{aligned} \omega^* &= \arg\max_{\omega\in\mathbb{R}^d} \{L(\omega,b)\} \\ &= \arg\max_{\omega\in\mathbb{R}^d} \left\{ \sum_{i=1}^{n} Y_i \log(P(X_i)) + (1-Y_i)\log(1-P(X_i)) \right\} \\ &= \arg\min_{\omega\in\mathbb{R}^d} \left\{ -\left(\sum_{i=1}^{n} Y_i \log(P(X_i)) + (1-Y_i)\log(1-P(X_i)) \right) \right\} \end{aligned} \tag{1.1.4}$$

1.2 Réseau de neurones

1.2.1 Modélisation mathématique

Dans la section précèdent nous avons vu le modèle mathématique de la régression logistique où nous entrons les caractéristiques X et les paramètres ω et b qui nous permettent de calculer z, puis nous calculons la fonction d'activation $a = \sigma(z)$, puis nous pouvons calculer la fonction de perte L.



Si nous représentons cette modélisation sous forme d'un neurones, nous trouvons le schéma ci-dessous:

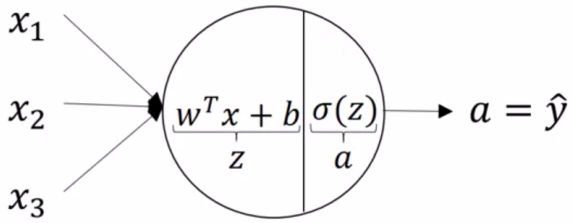


Figure 1.2.1: Modèle de Perceptron

Comme nous l'avions déjà mentionné, on obtint un réseau de neurones en empilant beaucoup de ces unités sigmoïdes. Alors qu'auparavant, ce nœud correspondait à deux étapes de calculs. Voici une image d'un réseau de neurones contient 4 noeuds dans la couche cachée.

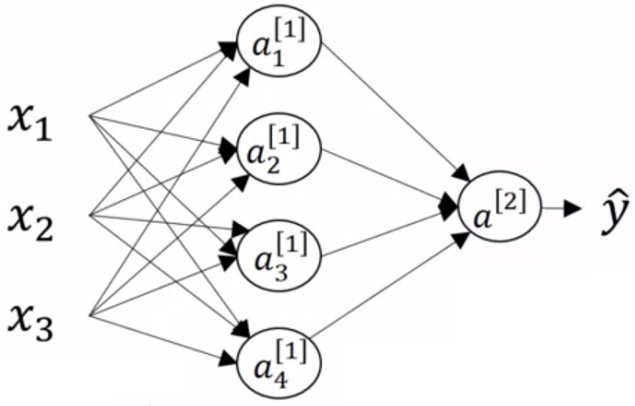


Figure 1.2.2: Réseau de neurones à une seule couche

Nous allons nommer les différentes parties de ce schéma. Nous avons les caractéristiques d'entrée, x1, x2, x3 empilés verticalement qui sont ce qu'on appelle la couche d'entrée du réseau neuronal. Alors, sans surprise, celle-ci contient les entrées pour le réseau de neurones. Puis il y a une autre couche de cercles. Et c'est ce qu'on appelle une couche cachée du réseau neuronal. Et finalement, la dernière couche est formée par un seul nœud qui est la couche de sortie et doit générer la valeur prédite $\hat{y}$.

1.2.1.1 Forward propagation

Concentrons nous sur un nœud i dans la couche cachée. De la même façons que dans la régression logistique, nous calculons $z_i^{[1]}$, puis nous calculons la fonction d'activation $a_i^{[1]} = \sigma(z_i^{[1]})$, et nous calculons finalement la deuxième fonction d'activation $a^{[2]} = \sigma(z^{[1]})$ de la couche de sortie. Nous faisons cela pour tous les nœuds, nous obtenons:

$$\begin{aligned}
z_1^{[1]} &= \omega_1^{[1]^T} X + b_1^{[1]}, \quad a_1^{[1]} = \sigma^{[1]}(z_1^{[1]}) \\
z_2^{[1]} &= \omega_2^{[1]^T} X + b_2^{[1]}, \quad a_2^{[1]} = \sigma^{[1]}(z_2^{[1]}) \\
z_3^{[1]} &= \omega_3^{[1]^T} X + b_3^{[1]}, \quad a_3^{[1]} = \sigma^{[1]}(z_3^{[1]}) \\
z_4^{[1]} &= \omega_4^{[1]^T} X + b_4^{[1]}, \quad a_4^{[1]} = \sigma^{[1]}(z_4^{[1]})
\end{aligned}$$

Pour simplifié, nous posons:

$$\omega^{[1]} = \begin{bmatrix} \omega_1^{[1]} \\ \omega_2^{[1]} \\ \omega_3^{[1]} \\ \omega_4^{[1]} \end{bmatrix}, \quad b^{[1]} = \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \\ b_4^{[1]} \end{bmatrix}, \quad z^{[1]} = \begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \\ z_4^{[1]} \end{bmatrix} \; et \quad a^{[1]} = \begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \\ a_4^{[1]} \end{bmatrix} \tag{1.2.1}$$

Prenons $\sigma^{[1]}$ (resp $\sigma^{[2]}$) la fonction d'activation de la couche 1 (resp 2).

$$\begin{aligned}
z^{[1]} &= \omega^{[1]T} X + b^{[1]} \\
a^{[1]} &= \sigma^{[1]}(z^{[1]}) \\
z^{[2]} &= \omega^{[2]T} a^{[1]} + b^{[2]} \\
a^{[2]} &= \sigma^{[2]}(z^{[2]})
\end{aligned} \tag{1.2.2}$$

$$Y_{prediction} = \begin{cases} 1 & si \quad a^{[2]} > 0.5 \\ 0 & sinon \end{cases} \tag{1.2.3}$$

1.2.1.2 La fonction de la perte

Compte tenu des prédictions sur tous les exemples, nous pouvons également calculer le coût J comme suit :

$$J = -\frac{1}{m} \sum_{i=0}^{m} \left(Y_i \log(a^{[2]}) + (1 - Y_i) \log(1 - a^{[2]} \right) \tag{1.2.4}$$

1.2.1.3 Backward propagation

Dans la rétro-propagation (Backward propagation), il faut calculer les dérivée des paramètres de notre modèle mathématique comme indiqué ci-dessous.

$$\begin{array}{lcl} dz^{[2]} & = & a^{[2]} - Y \\ dW^{[2]} & = & dz^{[2]} a^{[1]^T} \\ db^{[2]} & = & dz^{[2]} \\ dz^{[1]} & = & W^{[2]^T} dz^{[2]} \times \sigma^{[1]'}(z^{[1]}) \\ dW^{[1]} & = & dz^{[1]} X^T \\ db^{[1]} & = & dz^{[1]} \end{array} \tag{1.2.5}$$

Où $\sigma^{[1]}$ est la fonction d'activation tanh de la couche 1. Si $a = \sigma^{[1]}$ alors $\sigma^{[1]'} = 1 - a^2$.

1.2.2 Pseudo code

Pour estimer les paramètre de notre modèle mathématique, nous utilisons le gradient de descente:

T : nombre d'itération;
α : taux d'apprentissage;
$(X_i, y_i) \in \mathbb{R}^n \times \{0,1\}$ $i = 1, \cdots, m$ Jeu de données;
$\omega^{[1](0)}$, $\omega^{[2](0)}$: Sont initialisé aléatoirement;
while $t < T$ **do**
  Forward Propagation
    $z^{[1]} \leftarrow \omega^{[1]T} X + b^{[1]}$
    $a^{[1]} \leftarrow \sigma^{[1]}(z^{[1]})$
    $z^{[2]} \leftarrow \omega^{[2]T} a^{[1]} + b^{[2]}$
    $a^{[2]} \leftarrow \sigma^{[2]}(z^{[2]})$
  Calculer la fonction coût
    $J = -\frac{1}{m} \sum_{i=0}^{m} \left(Y_i \log(a^{[2]}) + (1 - Y_i) \log(1 - a^{[2]}) \right)$
  backward Propagation
    $dz^{[2]} \leftarrow a^{[2]} - Y$
    $dW^{[2]} \leftarrow dz^{[2]} a^{[1]^T}$
    $db^{[2]} \leftarrow dz^{[2]}$
    $dz^{[1]} \leftarrow W^{[2]^T} dz^{[2]} \times \sigma^{[1]'}(z^{[1]})$
    $dW^{[1]} \leftarrow dz^{[1]} X^T$
    $db^{[1]} \leftarrow dz^{[1]}$
  Mise à jour des paramètres
    $W^{[1]} \leftarrow W^{[1]} - \alpha dW^{[1]}$
    $b^{[1]} \leftarrow b^{[1]} - \alpha db^{[1]}$
    $W^{[2]} \leftarrow W^{[2]} - \alpha dW^{[2]}$
    $b^{[2]} \leftarrow b^{[2]} - \alpha db^{[2]}$
  t $\leftarrow t + 1$
end
Result: ω_t

Algorithm 1: Gradient de descente pour le réseaux de neurones

2 | Modèle d'apprentissage profonds

2.1 Architecture d'un modèle apprentissage profonds

Notre modèle est un réseau de neurones profonds de L couches.

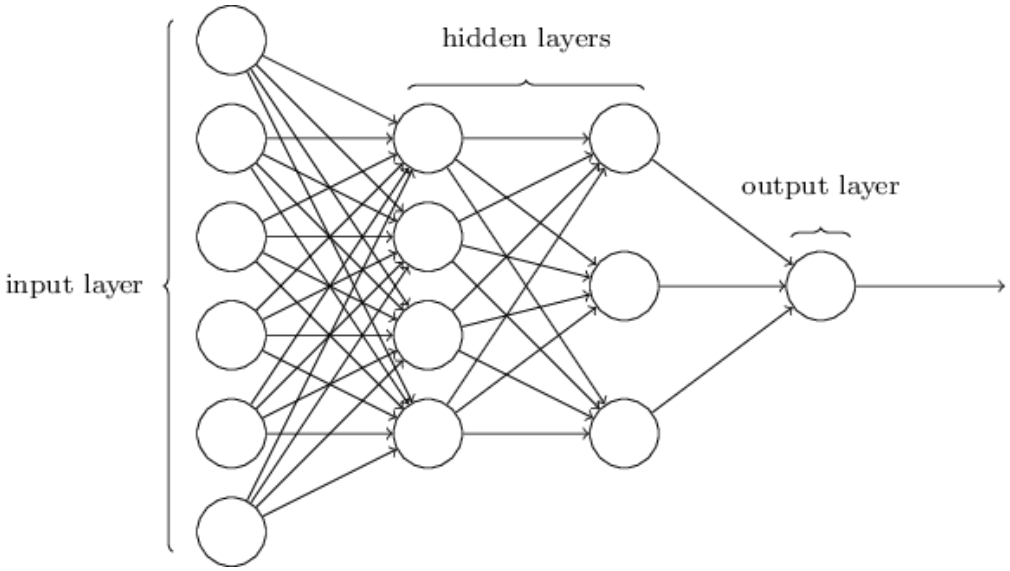


Figure 2.1.1: Architecture d'apprentissage profonds

Notation :

L'exposant l désigne une quantité associée à la i^{me} couche.

Exemple : a^L est la L^{me} couche d'activation. W^L et b^L sont les paramètres de la L^{me} couche.

L'exposant i désigne une quantité associée à la échantillon.

Exemple : $x^{(i)}$ est la i^{me} échantillon d'entraînement.

L'écriture minuscule de i désigne la i^{me} entrée d'un vecteur.

Exemple : a^l_i désigne la i^{me} entrée de la l^{me} couche activation.

Pour construire notre réseau de neurones, nous implémenterons plusieurs « fonctions d'assistance ». Ces fonctions d'assistance seront utilisées pour construire un réseau neuronal à L couche. Chaque petite fonction d'assistance que nous implémenterons aura des instructions détaillées qui nous guideront à travers les étapes nécessaires.

Voici un aperçu de cette mission, nous allons:

1. Initialisons les paramètres pour un réseau de neurones à L couches.
2. Implémentons le module de propagation vers l'avant «Forward».
    - Terminons la partie LINÉAIRE de l'étape de propagation vers l'avant d'une couche (résultant en $z^{[i]}$).
    - Nous mettons à notre disposition la fonction ACTIVATION.
    - Combinons les deux étapes précédentes dans une nouvelle fonction avant [LINEAR->ACTIVATION].
    - Empilons [LINEAR->Activation_function_chosen_for_hidden_layers] L-1 fois (pour les couches de 1 à L-1) et nous ajoutons un [LINEAR->Activation_function_chosen_for_output_layer] à la fin (pour la couche finale).
3. Calculons la perte.
4. Implémentons le module de propagation vers l'arrière «Backward».
    - Terminons la partie LINÉAIRE de l'étape de propagation en arrière d'une couche.
    - Nous calculons le gradient de la fonction d'ACTIVATION (relu_backward/sigmoid_backward).
    - Combinez les deux étapes précédentes dans une nouvelle fonction arrière [LINEAR->ACTIVATION].
    - Empilons [LINEAR->Activation_function_chosen_for_hidden_layers] en arrière L-1 fois et nous ajoutons [LINEAR->Activation_function_chosen_for_output_layer] en arrière dans une nouvelle fonction L_model_backward.
5. Enfin, mettons à jour les paramètres.

2.2 Modèle mathématique

2.2.1 Initialisation des paramètres

L'initialisation d'un réseau de neurones profonds de L couche est plus compliquée car il existe beaucoup plus de matrices de poids et de vecteurs de biais. Nous devons nous assurer que nos dimensions correspondent entre chaque couche. Rappelons que $n^{[l]}$ est le nombre d'unités dans la couche l. Ainsi par exemple si la taille de notre entrée X est (12288,209) (avec m=209 échantillons) alors :

- Utilisez l'initialisation aléatoire pour les matrices de poids.
- Utilisez l'initialisation des zéros pour les biais.

- Nous allons stocker $n^{[l]}$, le nombre d'unités dans différentes couches, dans une liste layer_dims. Par exemple, le layer_dims est [2,4,1], dans ce cas il y aura deux entrées, une couche cachée avec 4 unités cachées et une couche de sortie avec 1 unité de sortie. Cela signifie que la forme de $W[1]$ est (4,2), $b^{[1]}$ est (4,1), $W^{[2]}$ est (1,4) et $b^{[2]}$ est (1,1). Maintenant, nous pouvons généraliser cela à L couches !

2.2.2 Forward propagation

2.2.2.1 Linear Forward

Le module linéaire forward (vectorisé sur tous les exemples) calcule les équations suivantes :

$$Z^{[l]} = W^{[l]}A^{[l-1]} + b^{[l]} \tag{2.2.1}$$

Où $A^{[0]} = X$

2.2.2.2 Linear-Activation Forward

Nous allons implémenter la propagation vers l'avant de la couche LINEAR->ACTIVATION. La relation mathématique est :

$$A^{[l]} = g(Z^{[l]}) = g(W^{[l]}A^{[l-1]} + b^{[l]}) \tag{2.2.2}$$

où l'activation g peut être $sigmode()$ ou $relu()$.

- **Sigmoid:**

$$A^{[l]} = \sigma(Z^{[l]}) = \sigma(W^{[l]}A^{[l-1]} + b^{[l]}) = \frac{1}{1 + e^{-(W^{[l]}A^{[l-1]} + b^{[l]})}}$$

- **ReLU:**

$$A^{[l]} = ReLU(Z^{[l]}) = \max(0, Z^{[l]})$$

2.2.2.3 L-model forward

Pour encore plus de commodité lors de la mise en œuvre du réseau de neurones à L couches, nous aurons besoin d'une fonction qui réplique la précédente (linear_activation_forward avec une fonction d'activation convenable) L-1 fois, puis la suit avec une linear_activation_forward avec une fonction d'activation convenable au problème traité.

Etapes :

- Nous allons utiliser les fonctions que nous avions précédemment écrites.
- Nous allons utiliser une boucle for pour répliquer [LINEAR->Activation_Function_Chosen_for_hidden_lay (L-1) fois

2.2.3 La fonction de la perte

Nous devons calculer le coût, car nous voulons vérifier si notre modèle est réellement en train d'apprendre.

Calculez le coût d'entropie croisée J, en utilisant la formule suivante:

$$J = -\frac{1}{m}\sum_{i=1}^{m} y^{[l]}\log(a^{[l](i)}) + (1-y^{[l]})\log(1-a^{[l](i)}) \tag{2.2.3}$$

2.2.4 Backward propagation module

2.2.4.1 Linear backward

Supposons que nous ayons déjà calculé la dérivée $dZ^{[l]} = \frac{\partial J}{\partial Z^{[l]}}$. Nous voulons obtenir.

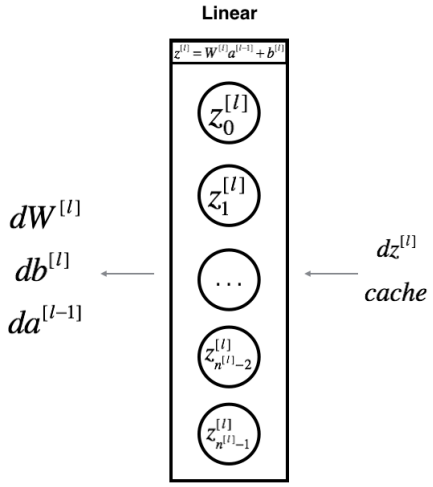


Figure 2.2.1: Linear backward

Les trois sorties sont calculées à l'aide de l'entrée $dZ^{[l]}$. Voici les formules dont nous avons besoin: $dW^{[l]}$,$db^{[l]}$,$dA^{[l-1]}$

$$\begin{aligned} dW^{[l]} &= \frac{\partial J}{\partial W^{[l]}} = \frac{1}{m} dZ^{[l]} A^{[l-1]T} \\ db^{[l]} &= \frac{\partial J}{\partial b^{[l]}} = \frac{1}{m}\sum_{i=1}^{m} dZ^{[l]i} \\ dA^{[l-1]} &= \frac{\partial J}{\partial A^{[l-1]}} = W^{[l]T} dZ^{[l]} \end{aligned} \tag{2.2.4}$$

2.2.4.2 Linear-Activation backward

Si $g(.)$ est la fonction d'activation, sigmoid_backward et relu_backward calculent

$$dZ^{[l]} = dA^{[l]} \times g'(Z^{[l]}) \tag{2.2.5}$$

2.2.4.3 L-Model Backward

Nous allons implémenter la fonction L_model_forward, et à chaque itération, nous allons stocker un cache qui contient X, W, b et Z. À chaque étape, nous utiliserons les valeurs mises en cache pour la couche l pour effectuer une rétro-propagation à travers cette couche et calculer les gradients nécessaires.

2.2.5 La mise à jour des paramètres

Dans cette section, nous mettrons à jour les paramètres du modèle, en utilisant la descente de gradient:

$$\begin{aligned} W^{[l]} &= W^{[l]} - \alpha dW^{[l]} \\ b^{[l]} &= b^{[l]} - \alpha db^{[l]} \end{aligned} \tag{2.2.6}$$

Où α est le taux d'apprentissage. Après avoir calculé les paramètres mis à jour, nous les stockons dans le dictionnaire des paramètres.

2.3 Pseudo code

Pour construire un modèle Deep Learning nous devions suivre la méthodologie suivante :

1. Initialiser les paramètres / Définir les hyperparamètres.
2. Boucler pour un nombre des itérations :
    - Propagation vers l'avant (Forward propagation).
    - Calculer la fonction de coût.
    - Propagation vers l'arrière (Backward propagation).
    - Mettre à jour les paramètres.
3. Utiliser des paramètres entraînés pour prédire les étiquettes.

3 | Technique d'optimisation

3.1 Initialisation des paramètres

L'entraînement de notre réseau de neurones nécessite de spécifier une valeur initiale des poids. Une méthode d'initialisation bien choisie facilitera l'apprentissage. Une initialisation bien choisie peut accélérer la convergence de la descente de pente et augmenter les chances de gradient de descente à converger vers une erreur d'entraînement (et de généralisation) inférieure.

Mais comment choisir l'initialisation d'un nouveau réseau de neurones ?

3.1.1 Initialisation par zéro

Il existe deux types de paramètres à initialiser dans un réseau de neurones:

- Les matrices de poids $W^{[1]}, W^{[2]}, \cdots, W^{[L-1]}, W^{[L]}$.
- Les vecteurs de biais $b^{[1]}, b^{[2]}, \cdots, b^{[L-1]}, b^{[L]}$.

En initialisant tous les paramètres à zéro, nous verrons plus tard que cela ne fonctionne pas bien car cela ne parvient pas à « casser la symétrie ».

3.1.2 Initialisation aléatoire

Pour casser la symétrie, nous initialisons les poids aléatoirement. Après une initialisation aléatoire, chaque neurone peut alors procéder à l'apprentissage d'une fonction différente de ses entrées.

On initialise les poids à valeurs aléatoires et les biais à zéro.

3.1.3 Initialisation de Xavier

Cette méthode est nommée d'après le nom de son créateur. C'est une initialisation aléatoire en utilisant un facteur d'échelle pour les poids $W^{[l]}$ de :

$$\sqrt{\frac{1}{dimension\ de\ la\ couche\ prcdente}} \tag{3.1.1}$$

On initialise les biais à zéro.

3.1.4 Initialisation de He

C'est similaire à l'initialisation de Xavier, sauf que cette méthode utilise un facteur d'échelle pour les poids $W^{[l]}$ de :

$$\sqrt{\frac{2}{dimension\ de\ la\ couche\ prcdente}} \tag{3.1.2}$$

On initialise les biais à zéro.

3.2 Régularisation

Les modèles d'apprentissage profond ont tellement de flexibilité et de capacité que le sur-apprentissage (overfittiong) peut être un problème sérieux, si l'ensemble de données d'entraînement n'est pas assez grand. Bien sûr, cela fonctionne bien sur l'ensemble d'entraînement, mais le réseau appris ne se généralise pas à de nouveaux exemples.

3.2.1 L2 Regularization

La méthode standard pour éviter le sur-apprentissage (overfitttiong) est appelée régularisation_$L2$.

Elle consiste à modifier de manière appropriée votre fonction de coût, à partir de :

$$J = -\frac{1}{m}\sum_{i=1}^{m} y^{[l]}\log(a^{[l](i)}) + (1-y^{[l]})\log(1-a^{[l](i)}) \tag{3.2.1}$$

À

$$J_{rgularis} = -\frac{1}{m}\sum_{i=1}^{m}\left(y^{[l]}\log(a^{[l](i)}) + (1-y^{[l]})\log(1-a^{[l](i)})\right) + \frac{1}{m}\frac{\lambda}{2}\sum_{l}\|W^{[l]}\|^2 \tag{3.2.2}$$

Puisque nous avons modifié le coût, nous devons également modifier la propagation vers l'arrière. Tous les gradients doivent être calculés par rapport à ce nouveau coût. Pour cela, on va mettre en œuvre les changements nécessaires en rétro-propagation pour prendre en compte la régularisation.

Les modifications ne concernent que $dW^{[l]}$. Pour chacun, il faut ajouter le gradient du terme de régularisation:

$$\frac{d}{dW}\left(\frac{1}{2}\frac{\lambda}{m}W^2\right) = \frac{\lambda}{m}W \tag{3.2.3}$$

Donc le modèle mathématique de la propagation dans l'arrière devient comme suit:

$$\begin{aligned} dW^{[l]} &= \frac{\partial J}{\partial W^{[l]}} = \frac{1}{m}dZ^{[l]}A^{[l-1]T} + \frac{\lambda}{W^{[l]}} \\ db^{[l]} &= \frac{\partial J}{\partial b^{[l]}} = \frac{1}{m}\sum_{i=1}^{m} dZ^{[l]i} \\ dA^{[l-1]} &= \frac{\partial J}{\partial A^{[l-1]}} = W^{[l]T}dZ^{[l]} \end{aligned} \tag{3.2.4}$$

Remarque:

- La valeur de λ est un hyperparamètre que nous pouvons régler.
- La régularisation_$L2$ rend notre frontière de décision plus lisse. Si λ est trop grand, il est également possible de « trop lisser », ce qui donne un modèle avec un biais élevé.

3.2.2 Dropout

C'est une technique de régularisation largement utilisée et spécifique au deep learning. Il arrête aléatoirement certains neurones à chaque itération.

À chaque itération, nous éteignons (= remettre à zéro) chaque neurone d'une couche avec une probabilité 1 - *keep_prob* ou le garderons avec une probabilité *keep_prob*.

Lorsque nous fermons certains neurones, nous modifions en fait notre modèle. L'idée derrière l'abandon (Dropout) est qu'à chaque itération, nous entraînons un modèle différent qui n'utilise qu'un sous-ensemble de nos neurones. Avec le décrochage, nos neurones deviennent donc moins sensibles à l'activation d'un autre neurone spécifique, car cet autre neurone peut être arrêté à tout moment.

3.2.2.1 Forward propagation with dropout

Nous voudrions arrêter certains neurones dans les couches cachées. Pour cela, nous allons effectuer 2 étapes :

1. Initialiser un vecteur $D^{[l]}$ rempli aléatoirement par des zéros et des uns, sachant que les uns représente un pourcentage de $keep\_prob \times 100$ des valeurs du vecteur $D^{[l]}$.
2. On abandonne certains neurones de $A^{[l]}$ comme suit:

$$A^{[l]} = \Pi_{i=1}^{n} A_i^l \times D_i^l \tag{3.2.5}$$

Les étapes de propagation vers l'avant se font comme d'habitude, sauf que après avoir calculé des $A^{[l]}$ on applique la procédure précédente.

3.2.2.2 Backward propagation with dropout

Nous calculions les dérivés nécessaires comme d'habitude sur les mêmes neurones sur lesquels nous avons appliqué la propagation vers l'avant.

Noter bien:
Une erreur courante lors de l'utilisation du technique du Dropout est de l'utiliser à la fois pour la formation et les tests. Nous ne devons utiliser le dropout (élimination aléatoire de certains nœuds) que pendant l'entraînement.

3.3 Méthodes d'optimisations

3.3.1 Mini-Batch and Stochastique Gradient descent

Avoir un bon algorithme d'optimisation peut faire la différence entre des jours d'attente et quelques heures pour obtenir un bon résultat. Jusqu'à présent, nous avons toujours utilisé Gradient Descent pour mettre à jour les paramètres et minimiser les coûts. Dans cette partie, nous apprendrons des méthodes d'optimisation plus avancées qui peuvent accélérer l'apprentissage et peut-être même nous permettre d'obtenir une meilleure valeur finale pour la fonction de coût.

Nous connaissons tous la fameuse règle de mise à jour du gradient de descente.
Pour $l = 1, \cdots, L$:

$$\begin{aligned} W^{[l]} &= W^{[l]} - \alpha dW^{[l]} \\ b^{[l]} &= b^{[l]} - \alpha db^{[l]} \end{aligned} \tag{3.3.1}$$

Une variante de ceci est la descente de gradient stochastique (SGD), qui équivaut à un gradient de descente en mini-lot où chaque mini-lot n'a qu'un seul exemple. La règle de gradient de descente reste la même. Ce qui change, c'est que nous calculerions les gradients sur un seul exemple d'entraînement à la fois, plutôt que sur l'ensemble de l'entraînement.

Lorsque l'ensemble d'entraînement est volumineux, SGD peut être plus rapide. Mais les paramètres « oscilleront » vers le minimum plutôt que de converger en douceur. Voici une illustration de ceci :

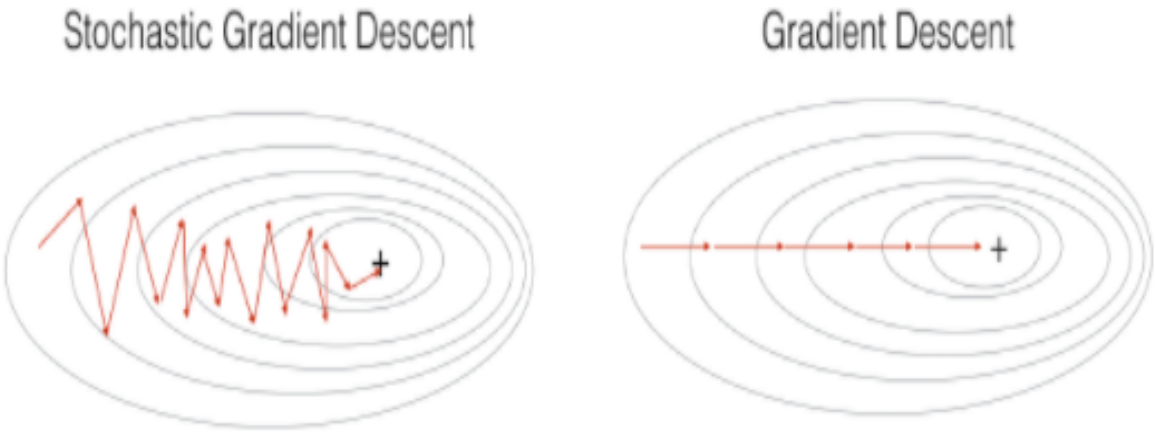


Figure 3.3.1: Illustration de la convergence de la fonction de perte en utilisant Gradient de Descente Stochastique et Gradient de Descente

Dans la pratique, nous utilisons le gradient de descente en mini-lot qui prend un nombre intermédiaire des échantillons pour chaque étape. Avec le gradient de descente par mini-lot, nous bouclons sur les mini-lots au lieu de boucler sur des échantillons d'entraînement individuels.

3.3.2 Momentum

Étant donné que le gradient de descente en mini-lot effectue une mise à jour des paramètres après avoir vu juste un sous-ensemble d'exemples, la direction de la mise à jour a une certaine variation,

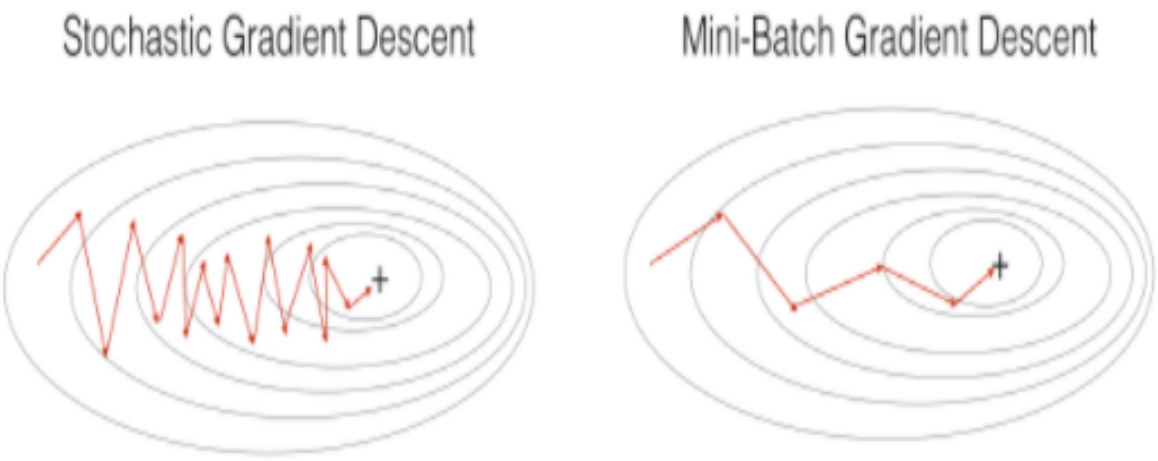


Figure 3.3.2: Illustration de la convergence de la fonction de perte en utilisant Gradient de Descente Stochastique et Gradient de Descente en lots

et donc le chemin emprunté par la descente de gradient en mini-lot va "osciller" vers la convergence. L'utilisation du Momentum peut réduire ces oscillations.

La règle de mise à jour avec le momentum est:
Pour $l = 1, \cdots, L$:

$$\begin{aligned} v_{dW^{[l]}} &= \beta v_{dW^{[l]}} + (1-\beta) dW^{[l]} \\ W^{[l]} &= W^{[l]} - \alpha dW^{[l]} \\ v_{db^{[l]}} &= \beta v_{db^{[l]}} + (1-\beta) db^{[l]} \\ b^{[l]} &= b^{[l]} - \alpha db^{[l]} \end{aligned} \tag{3.3.2}$$

où l est le nombre de couches, β est le momentum et α est le taux d'apprentissage.
Noter bien:

- Si $\beta = 0$, alors cela devient simplement une descente de gradient standard sans momentum.
- Plus β est grande, plus la mise à jour est fluide. Mais si est trop grand, cela pourrait aussi trop lisser les mises à jour.
- Les valeurs communes pour β vont de 0.8 à 0.999. Si nous ne nous sentons pas enclin à régler cela, $\beta = 0.9$ est souvent une valeur raisonnable par défaut.

3.3.3 Adam

Adam est l'un des algorithmes d'optimisation les plus efficaces pour l'entraînement des réseaux de neurones. Il combine des idées de RMSProp et de Momentum.

La règle de mise à jour est :

Pour $l = 1, \cdots, L$:

$$\begin{aligned}
v_{dW^{[l]}} &= \beta_1 v_{dW^{[l]}} + (1-\beta_1)\frac{\partial J}{\partial W^{[l]}} \\
v_{dW^{[l]}}^{Corrected} &= \frac{v_{dW^{[l]}}}{1-(\beta_1)^t} \\
S_{dW^{[l]}} &= \beta_2 S_{dW^{[l]}} + (1-\beta_2)\frac{\partial J}{\partial W^{[l]}} \\
S_{dW^{[l]}}^{Corrected} &= \frac{S_{dW^{[l]}}}{1-(\beta_2)^t} \\
W^{[l]} &= W^{[l]} - \alpha \frac{v_{dW^{[l]}}^{Corrected}}{\sqrt{S_{dW^{[l]}}^{Corrected} + \varepsilon}}
\end{aligned} \tag{3.3.3}$$

Pour le biais:

$$\begin{aligned}
v_{db^{[l]}} &= \beta_1 v_{db^{[l]}} + (1-\beta_1)\frac{\partial J}{\partial b^{[l]}} \\
v_{db^{[l]}}^{Corrected} &= \frac{v_{db^{[l]}}}{1-(\beta_1)^t} \\
S_{db^{[l]}} &= \beta_2 S_{db^{[l]}} + (1-\beta_2)\frac{\partial J}{\partial b^{[l]}} \\
S_{db^{[l]}}^{Corrected} &= \frac{S_{db^{[l]}}}{1-(\beta_2)^t} \\
b^{[l]} &= b^{[l]} - \alpha \frac{v_{db^{[l]}}^{Corrected}}{\sqrt{S_{db^{[l]}}^{Corrected} + \varepsilon}}
\end{aligned} \tag{3.3.4}$$

Où

- **t:** compte le nombre de pas d'Adam.
- **L:** est le nombre de couches
- β_1 et β_2 : sont des hyperparamètres qui contrôlent les deux moyennes pondérées exponentiellement.
- α: est le taux d'apprentissage
- ε: est un très petit nombre pour éviter de diviser par zéro

4 | Implémentation

4.1 Introduction

Ce chapitre sera organisé comme suit, tout d'abord nous présentons une documentation détaillée dans la première section sur l'implémentation du Framework qui comporte les réseaux de neurones et les réseaux de neurones profonds. Puis nous appliquons les deux modèles pour la classification deux ensembles de données différent en espace $\mathbb{R}^2$. Enfin, nous allons appliquer le modèle de l'apprentissage profond pour le diagnostic de la pneumonie.

4.2 Documentation du Framework

Notre Framework se compose de deux classes principales, la première pour les réseaux de neurones s'appelle **NeuralNets**, et la seconde pour les réseaux de neurones profonds s'appelle **DeepNeuralNets**.

4.2.1 Réseaux de neurones

4.2.1.1 Paramètres

- **X :** Est une matrice de de taille $(n \times m)$ contenant des vecteurs X_i disposés verticalement, avec m est la taille de l'échantillon et n le dimension de l'espace d'entrer.
- **Y :** Est un vecteur de taille m, où chaque $Y[i]$ correspond à l'étiqueter de $X[i]$.
- **n_h :** Le nombre des nuerons dans la couche caché.
- **num_iterations :** Le nombre d'itération.
- **print_cost :** Est un booléen pour écrire ou non l'évolution de la fonction coût pour chaque 100 itération.
- **nbr_print :** Est le période pour écrire la fonction coût.

4.2.1.2 Méthodes

- **L'initialisation des paramètres:**

```
def initialize_parameters(self, n_x, n_y):
```

Cette fonction prend en entré la taille de la couche d'entrer et de sortie, et retourne un dictionnaire *parameters* qui comporte les paramètres $W1$, $b1$, $W2$ et $b2$ initialisé aléatoirement.

- **Forward propagation :**

```
def forward_propagation(self, X, parameters):
```

Cette fonction prend en entré le vecteur X et le dictionnaire *parameters*, et retourne $a^{[2]} = sigmoid(Z^{[2]})$ et un dictionnaire qui comporte les paramètres $Z1$, $A1$, $Z2$ et $A2$ initialisé aléatoirement.

- **La fonction coût :**

```
def compute_cost(self, A2, parameters):
```

Cette fonction prend en entré $a^{[2]}$ et le dictionnaire *parameters* et retourne la fonction coût mentionné dans (1.2.4).

- **Backward propagation :**

```
def backward_propagation(self, parameters, cache):
```

Cette fonction prend en entré les dictionnaires *parameters* et *cache* et retourne un dictionnaire *grads* comporte les valeurs de $dW1$, $db1$, $dW2$ et $db2$.

- **Mise à jour des paramètre :**

```
def update_parameters(self, parameters, grads, learning_rate = 1.2):
```

Cette fonction prend en entré les deux dictionnaire *parameters* et *grands* et *parameters* et *learning_rate* qui est le taux d'apprentissage. Elle fait la mise à jour des paramètres $W1$, $b1$, $W2$ et $b2$ et retourne les nouveaux paramètres dans le dictionnaire *parameters*.

- **L'ajustement de l'algorithme :** Pour lancer l'algorithme à apprendre il suffit d'appeler la fonction *fit*.

```
def fit(self):
```

- **La fonction de prédiction :**

```
def predict(self, parameters, X):
```

la fonction *predict* prend en entré un vecteur X et le dictionnaire *parameters* et retourne une prédiction de l'étiqueter $Y_{predicted}$ de X.

4.2.1.3 Constructeur

Pour utiliser le réseaux de neurons de notre Framework **IIA**, il suffit de déclarer la classe *NeuralNet* comme montré ci-dessous.

```
from IIA import NeuralNets

NN = NeuralNets(
    X,
    y,
    n_h             = 4,
    num_iterations  = 5000,
    print_cost      = True,
    nbr_print       = 1000
)
```

Et pour lancer l'algorithme à apprendre ils suffit d'appeler la fonction *fit()* comme suit.

```
nn_parameters = NN.fit()
```

4.2.2 Réseaux de nuerons profonds

4.2.2.1 Paramètres

- **X :** Est une matrice de de taille $(n \times m)$ contenant les vecteurs X_i disposés verticalement, avec m est la taille de l'échantillon et n le dimension de l'espace d'entrer.
- **Y :** Est un vecteur de taille m, où chaque $Y[i]$ correspond à l'étiqueter de $X[i]$.
- **layers_dims :** Dimensions des couches de la réseaux de nuerons profond.
- **num_iterations :** Le nombre d'itération.
- **learning_rate :** Le taux d'apprentissage.
- **initialization :** Cette attribut permet de choisir la méthode d'initialisation des paramètre Wi et bi, il peut etre **zeros**, **random**, **xavier** ou **he**.
- **optimizer :** Cette attribut permet de choisir la méthode d'optimisation, ile peut etre **gd**, **momentum** ou **adam**.
- **beta**, **beta1** et **beta2 :** Sont les taux d'apprentissage utilisé pour l'algorithme Adam. Il doivent initialiser saut si l'attribut **optimiser** est initialisé par "Adam".
- **lambda :** Paramètres de régularisation.
- **kepp_prob :** Nombre réel varie de 0 à 1.
- **print_cost :** Est un booléen pour écrire ou non l'évolution de la fonction coût pour chaque 100 itération.
- **nbr_print :** Est le période pour écrire la fonction coût.

4.2.2.2 Méthodes

- **L'initialisation des paramètres :** Vu que la phase d'initialisation joue un rôle très importance dans l'accélération de la convergence du gradient de descente et une erreur d'entraînement et de génération inferieur, on a implémenté 4 méthodes d'initialisation. Toute ces quatre fonction retourneront retourne un dictionnaire qui contient 2 vecteurs initialisés selon la loi mentionnée dans la partie théorique.

```
def initialize_parameters_random(self):
def initialize_parameters_zeros(self):
def initialize_parameters_xavier(self):
def initialize_parameters_he(self):
```

- **Fonctions d'activation :** Dans le modèle des réseaux de nuerons profonds nous avons utilisé deux fonction d'activation *sigmoid* et *ReLU*.

```
def sigmoid(self, Z):
def relu(self, Z):
```

Ces deux fonctions prennent en entré une vecteur Z et retourneront $\sigma(W)$, avec σ c'est la fonction d'activation choisit.

- **Forward propagation :** Dans cette partie, nous avons implémenté fonctions pour le calcul du Forward propagation sans et avec régularisation.

  1. *Sans régularisation :*
     - *Linear model :* Prend en entrée "A", "W" et "b", et retourne "Z" et un "cache" qui contient "A", "W" et "b".

     ```
     def linear_forward(self, A, W, b):
     ```

     - *Linear Activation Forward :* Prend en entrée "A_prev", "W", "b" et "activation", et retourne "A" et "cache" qui contient "linear_cache" et "activation_cache" stocké pour calculer efficacement le passage en arrière.

     ```
     def linear_activation_forward(self, A_prev, W, b, activation):
     ```

     - *L-model Forward :* Prend en entrée "X" et "Parameters", et retourne "AL" et "caches".

     ```
     def L_model_forward(self, X, parameters):
     ```

  2. *Avec régularisation :*
     - *Activation Forward :* Prend en entrée "A_prev", "W", "b", "activation" et "keep_prob", et retourne "A" et "cache".

     ```
     def activation_forward(self, A_prev, W, b, activation, keep_prob):
     ```

     - *Model Forward :* Prend "X", "paramters" et "keep_prob", et retourne "AL" et "caches".

     ```
     def model_forward(self, X, parameters, keep_prob):
     ```

- **La fonction coût :** Pour la fonction coût, nous avons utilisé deux types, sans régularisation qui prend en entré *AL*.

```
def compute_cost(self, AL)
```

Et Avec régularisation qui prend en entré *AL*, *parameters* et *lambda*.

```
def compute_cost_with_regularization(self, AL, parameters, lambda):
```

- **Backward propagation :** Tout comme avec la Forward Propagation, nous implémenterons des fonctions d'assistance pour la propagation arrière. La Backward propagation est utilisée pour calculer le gradient de la fonction de perte par rapport aux paramètres.

  1. *Sans régularisation :*
     - *Linear backward :* Prend ene entrée "dZ" et "cache", et retourne "dA_prev", "dW" et "db".

       ```
       def linear_backward(self, dZ, cache):
       ```

     - *Linear-Activation backward :* Prend en entrée "dA", "cache" et "activation", et retourne "dA_prev", "dW" et "db".

       ```
       def linear_activation_backward(self, dA, cache, activation):
       ```

     - *L-model backwrad :* Prend en entrée "AL" et "caches", et retourne "grads".

       ```
       def L_model_backward(self, AL, caches):
       ```

  2. *Avec régularisation :*
     - *Activation Backward :* Prend en entrée "dA", "cache", "lambd", "keep_prob", "activation", et retourne "dA_prev", "dW" et "db".

       ```
       def activation_backward(self, dA, cache, lambda, keep_prob,
           activation):
       ```

     - *Model Backward :* Prend en entrée "AL", "Y", "caches", "lambd", "keep_prob", et retourne "grads".

       ```
       def model_bacward(self, AL, caches, lambda, keep_prob):
       ```

- **Mise à jour des paramètres :** Pour la mise à jour des paramètres se fait selon la méthode d'optimisation choisit, nous avons implémenter trois méthode d'optimisation qui sont:

  1. *Gradient de descente :* Prend en entrée "parameters", "grads" et "learning_rate" et retourne "parameters" modifié.

```
def update_parameters_with_gd(self, parameters, grads):
```

2. *Gradient de descente avec momentum :* Prend en entrée "parameters", "grads" et "v" et retourne "parameters" modifié.

```
def update_parameters_with_momentum(self, parameters, grads, v):
```

3. *Adam:* Prend en entrée "parameters", "grads" , "v" , "s" et "t" et retourne "parameters" modifié.

```
def update_parameters_with_adam(self, parameters, grads, v, s, t):
```

4.2.2.3 Constructeur

Pour utiliser le réseaux de nuerons profonds de notre Framework **IIA**, il suffit de déclarer la classe *DeepNeuralNet* et de lui affecter les attributs nécessaire comme montré ci-dessous.

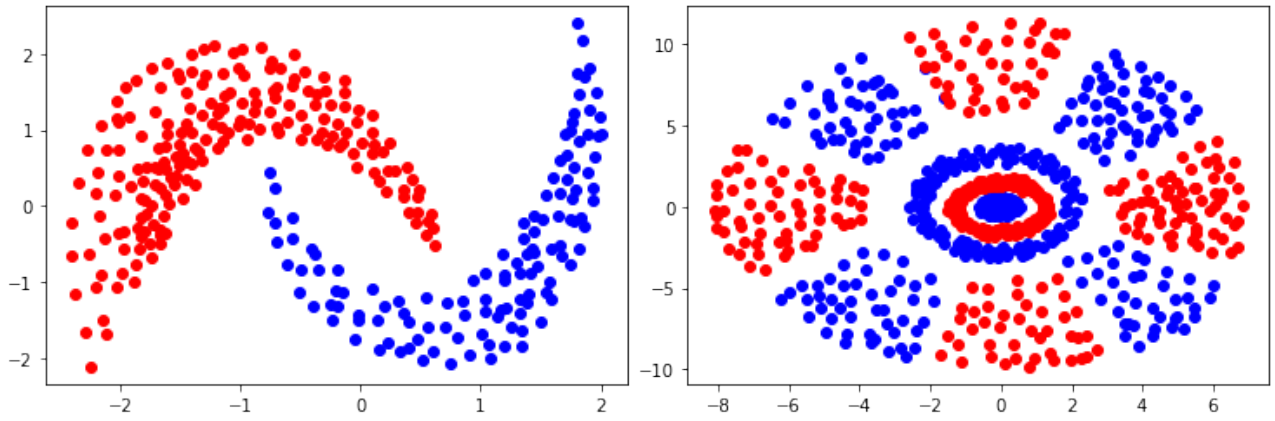
```
from IIA import DeepNeuralNets

DNN = DeepNeuralNets(
    X_train.T,
    y_train.T,
    layers_dims,
    learning_rate  = 0.0075,
    num_iterations = 1000,
    initialization = "random", # Méthode d'initialisation des paramètre W_i, b_i
    optimizer      = "gd",     # Méthode d'optimisation
    print_cost     = True
)
```

Et pour lancer l'algorithme à apprendre ils suffit d'appeler la fonction *fit()* comme suit.

```
dnn_parameters = DNN.fit()
```

4.3 Exemple d'application

Pour tester notre Framework IIA, nous allons choisi deux jeu de données, le première d'appel Two Crescents dataset, et le deuxième Freemasonry dataset. Nous allons utiliser le premier jeu de données pour tester *NeuralNets*, et le deuxième pour tester *DeepNeuralNets*



(a) Two Crescents dataset

(b) Freemasonry dataset

4.3.1 Application de *NeuralNets*

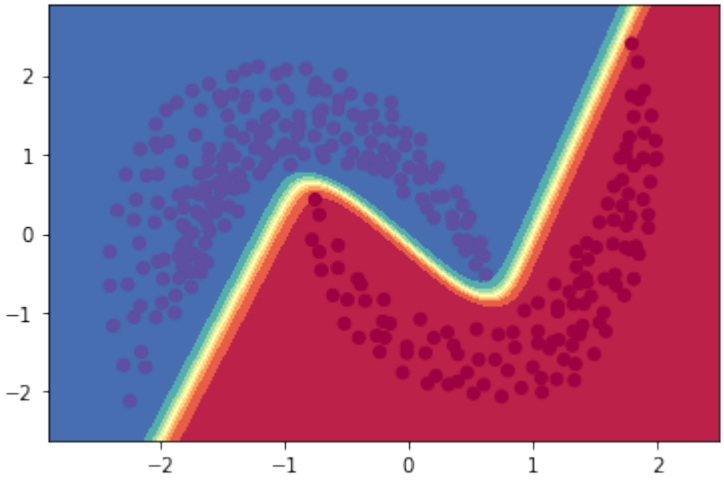
Chargement du jeu de données séparé en deux partie, données d'entrainement et données de test.

```
X_train, X_test, y_train, y_test = load_dataset('dataset.csv')
```

Déclaration de la classe *NeuralNets* et lancement de l'algorithme avec 4 neurones dans la couche cachée

```
from IIA import DeepNeuralNets, NeuralNets
NN = NeuralNets(
    X_train.T,
    y_train.T,
    n_h            = 4,
    num_iterations = 5000,
    print_cost     = True,
    nbr_print      = 1000
)
nn_parameters = NN.fit()
```

Le graph ci-dessous, illustre le résultat fournit par le modèle *NeuralNets*.



Pour la validation de l'algorithme nous avons utiliser la classe *metrics* pour calculer la précision, l'Accuracy et Recall:

	precision	recall	f1-score	support
0	0.43	1.00	0.61	10
1	1.00	0.86	0.93	94
accuracy			0.88	104
macro avg	0.72	0.93	0.77	104
weighted avg	0.95	0.88	0.89	104

Figure 4.3.2: Évaluation du modèle sur Dataset1

4.3.2 Application de *DeepNeuralNets*

Chargement du jeu de données séparé en deux partie, données d'entrainement et données de test.

```
X_train, X_test, y_train, y_test = load_dataset('dataset2.csv')
```

Initialisation de la dimension des couches de la réseaux de neurons profond

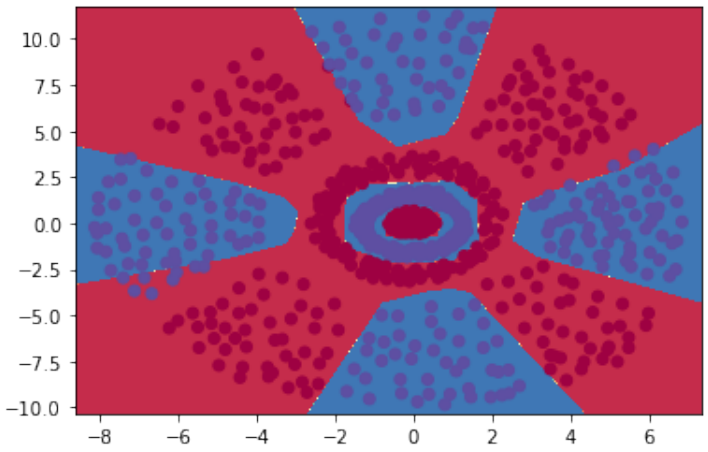
```
m   = 2   # couche d'entrer
n_1 = 20  # couche 1
n_2 = 10  # couche 2
n_3 = 1   # couche de sortie
layers_dims = [m, n_1, n_2, n_3]
```

Déclaration de la classe *DeepNeuralNets* et lancement de l'algorithme à apprendre dans les données d'entrainement.

```
DNN = DeepNeuralNets(
    X_train.T,
    y_train.T,
```

```
    layers_dims,
    learning_rate = 0.0075,
    num_iterations = 50000,
    initialization = "random",
    optimizer="gd",
    print_cost=True,
    nbr_print = 5000
)
dnn_parameters = DNN.fit()
```

Le graph ci-dessous, illustre le résultat fournit par le modèle *NeuralNets*.



Et ce graphe illustre l'évolution de la fonction cout en fonction des nomlbre d'itération.

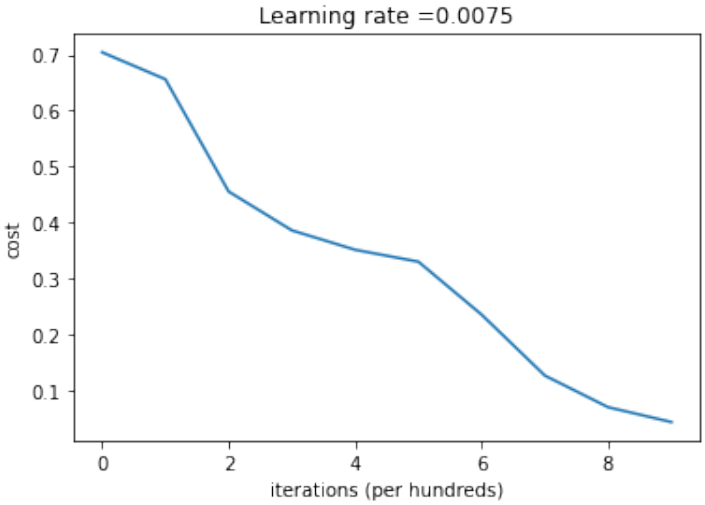


Figure 4.3.3: Évolution de la fonction de perte sur dataset2

Pour la validation de l'algorithme nous avons utiliser la classe *metrics* pour calculer la précision, l'Accuracy et Recall:

	precision	recall	f1-score	support
0	0.43	1.00	0.61	10
1	1.00	0.86	0.93	94
accuracy			0.88	104
macro avg	0.72	0.93	0.77	104
weighted avg	0.95	0.88	0.89	104

Figure 4.3.4: Évaluation du modèle sur Dataset2

4.4 Application du Deep Learning pour le Diagnostic de Pneumonie

Nous allons appliquer notre Framework pour le diagnostic de Pneumonie qui est une infection des poumons causée le plus souvent par un virus ou une bactérie.

Nous avons des images X_Ray étiquetées comme ensemble de données. Nous allons utiliser la régression logistique pour classifier les images X-Ray des personnes infecté et non infecter.

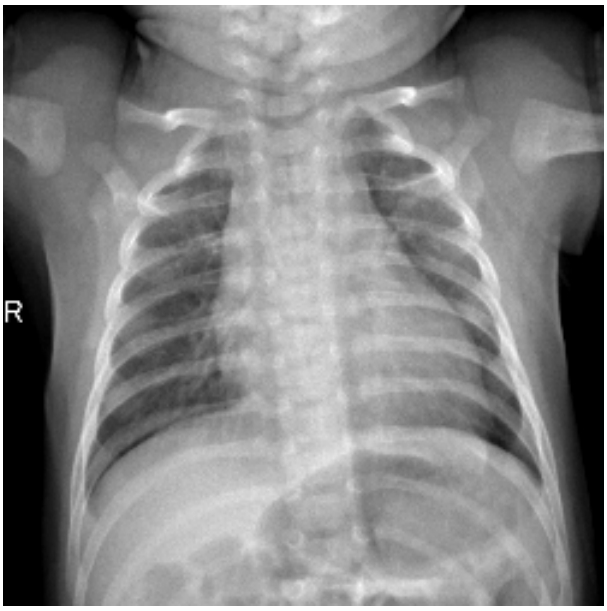
4.4.1 Présentation du jeu de donnée

L'ensemble de données est organisé en 2 dossiers (NORMAL, PNEUMONIA). Il existe 99 images de rayons X (PNG) pour chaque cas (Pneumonie / Normal).

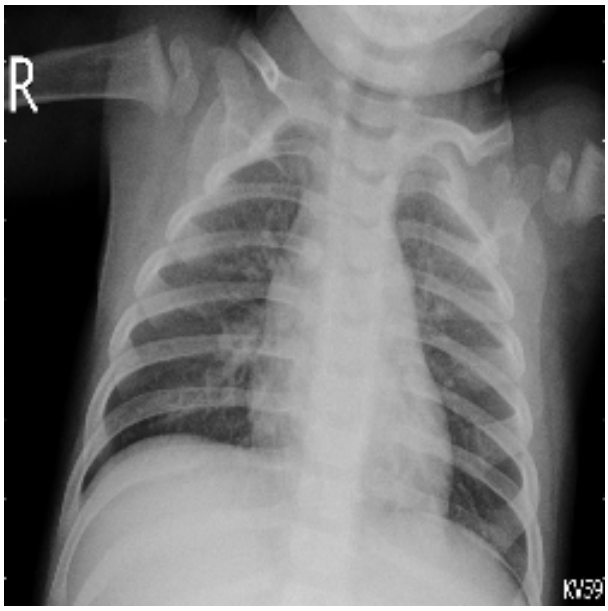
Des radiographies thoraciques (antéro-postérieures) ont été sélectionnées à partir de cohortes rétrospectives de patients pédiatriques âgés d'un à cinq ans du centre médical pour femmes et enfants de Guangzhou, à Guangzhou. Toutes les radiographies pulmonaires ont été réalisées dans le cadre des soins cliniques de routine des patients.

Pour l'analyse des images radio-graphiques thoraciques, toutes les radiographies thoraciques ont été initialement examinées pour le contrôle de la qualité en supprimant toutes les analyses de mauvaise qualité ou illisibles. Les diagnostics pour les images ont ensuite été notés par deux médecins experts avant d'être validés pour la formation du système d'IA. Afin de tenir compte d'éventuelles erreurs de notation, l'ensemble d'évaluation a également été vérifié par un troisième expert.

Voila deux images X-Ray dont la première est infecté et l'autre non.



(a) Image X-Ray d'une personne non infecté par le Pneumonie



(b) Image X-Ray d'une personne infecté par le Pneumonie

4.4.2 Application

Tout d'abord, nous chargeons notre ensemble de données et le divisons en deux parties, une pour l'entraînement et l'autre pour les tests. Nous avons écrit cette fonction qui retourne 70% de l'ensemble des données pour l'entraînement et 30% pour le test.

```
1 def load_data():
```

```
    X = np.array([image_to_array('./Dataset/NORMAL/1_1.png')/255])
    y = np.array([1])
    for k in range(99):
        try:
            X = np.append(
              X,
              np.array([image_to_array('./Dataset/NORMAL/1_'+str(k)+'.png')]),
              axis=0
            )
            y = np.append(y, 1)
            X = np.append(
              X,
              np.array([image_to_array('./Dataset/PNEUMONIA/0_'+str(k)+'.png')]),
              axis=0
            )
            y = np.append(y, 0)
        except: None
    y = y.reshape(y.shape[0],1)
    m = X.shape[0]
    return X[0:int(0.7*m)], X[int(0.7*m):m], y[0:int(0.7*m)], y[int(0.7*m):m]
# loading dataset
X_train, X_test, y_train, y_test = load_data()
```

Puis nous initialisons les dimensions des couches de réseaux de neurones. Avec

- **m :** dimension du vecteur d'entré X_i
- **n_1 :** nombre des neurones dans la première couche
- **n_2 :** nombre des neurones dans la deuxième couche
- **n_3 :** nombre des neurones dans la couche de sortie

```
m = 62500
n_1 = 20
n_2 = 10
n_3 = 1
layers_dims = [m, n_1, n_2, n_3]
```

Nous pouvons ajouter plus de n_i autant que nous le voulons. Puis nous initialisons le modèle d'apprentissage profond avec la méthode aléatoire d'initialisation des paramètres et le gradient de descente comme méthode d'optimisation.

```
from IIA import DeepNeuralNets

DNN = DeepNeuralNets(
    X_train.T,
    y_train.T,
    layers\_dims,
    learning\_rate   = 0.0075,    # Taux d'apprentissage
    num\_iterations = 1000,      # Nombre d'itération
```

```
    initialization = "random",
    optimizer      = "gd",
    print_cost     = True
)
```

Enfin, l'ajustement du Deep Learning sur l'ensemble de données d'entraînement avec la méthode *fit()*.

```
# Fitting the Deep learning on the training dataset
parameters = DNN.fit()
```

Comme résultat, nous avons tracer ce graphique qui nous illustre l'évolution de notre modèle de Deep Learning dans l'optimisation de la fonction coût en fonction du nombre d'itération.

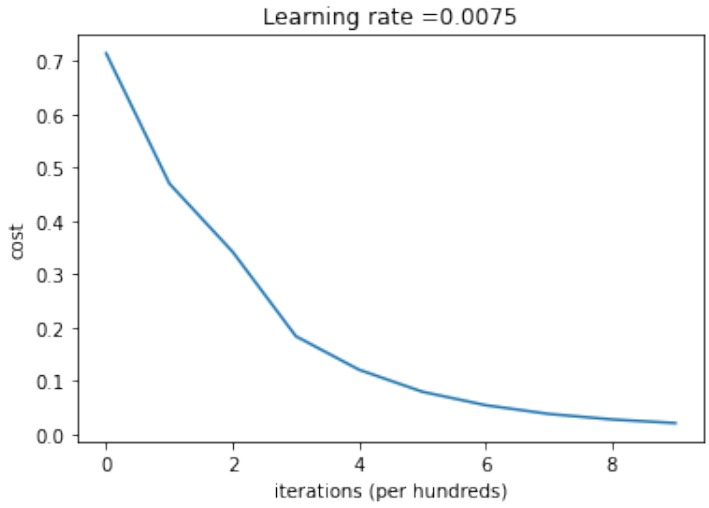


Figure 4.4.2: Evolution de la fonction de perte

Pour la validation de notre modèle, nous calculons l'accuracy, la précision et le recall en se basant sur l'ensemble de données de test.

	precision	recall	f1-score	support
0	0.86	0.86	0.86	29
1	0.87	0.87	0.87	30
accuracy			0.86	59
macro avg	0.86	0.86	0.86	59
weighted avg	0.86	0.86	0.86	59

Nous constatons que notre modèle nous a fournit une précision de 86% et une Accuracy de 86%.

Conclusion

Notre projet, consiste à développer une propre Framework d'apprentissage en profondeur, sous le nom de « IIA », et contient un modèle de réseau de neurones et réseau de neurones profonds.

Pour réaliser ce projet, nous avons commencé par le modèle mathématique des réseaux de neurones ainsi que son pseudo code. Ensuite, nous avons montré l'architecture et le modèle mathématique ainsi que son pseudo code. Puis, nous avons parlé des techniques d'optimisation proposées dans notre modèle de la réponse. Enfin, une documentation de notre Framework et son application sur quelque dataset.

En guise de conclusion, nous pouvons affirmer que ce projet fut une expérience très enrichissante pour nous sur les deux plans personnels et professionnels, nous avons pu approfondir nos connaissances techniques et nous avons bien développer les techniques du travail en groupe, et nous somme satisfait pour le travail que nous avons abouti, qui nous donne une flexibilité de modifier et d'optimiser notre modèle d'apprentissage en profondeur comme on veut, afin d'arriver au modèle qui va résoudre les problèmes qu'on confronte.