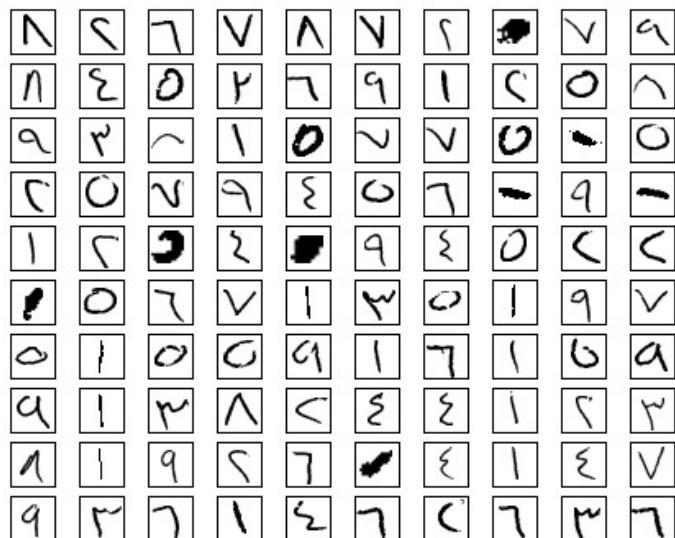


RAPPORT DE PROJET :

ARABIC HANDWRITTEN DIGIT RECOGNITION



Réalisé par :

BELKHO Zineb
EL KALKHA Omar
ZAHAR Amina

Encadré par :

LAKHEL El Hassan

REMERCIEMENT

Tout d'abord, nous tenons à remercier notre professeur encadrant **M.Lakhel El Hassan** qui nous a accordé tout au long de notre projet et nous a consacré pour doter de solide connaissances au niveau des théories mathématiques.

Nous remercions également tous les professeurs qui ont contribué, par leur disponibilité et leur bonne humeur, à rendre notre formation enflammée et enrichissante.

Nous tenons ensuite à remercier particulièrement nos parents qui n'ont cessé de nous soutenir et nous encourager au quotidien dans toute notre démarche, à la fois sur le plan professionnel et personnel.

RESUME

Ce projet vise à développer un modèle artificiel capable de reconnaître les chiffres arabes-orientaux qui sont écrits à la main en utilisant des techniques d'optimisation et des algorithmes avancés d'apprentissage profond (deep learning) tels que la rétropropagation et la régularisation. On a crée également une interface graphique d'essai.

Tout d'abord, on a importé notre base de données MAHD Base contenant les images des chiffres manuscrits pour entraîner notre système, après on a appliqué un prétraitement approprié pour transformer les données de test ou de validation en données utilisables par notre modèle artificiel crée.

En suite, on a implémenté un modèle d'apprentissage automatique, sous forme d'un classificateur basé sur les réseaux de neurones qui peut être entraîné à partir des données d'entraînement annotées en exploitant la descente du gradient stochastique et la rétropropagation. Une fois notre modèle a été évalué et validé, on a crée une interface graphique d'utilisateur pour le tester. Enfin, on a augmenté la précision en utilisants des techniques d'apprentissage avancées tels que la régularisation, l'augmentation artificielle de la base de données et l'initialisation de Xavier.

Mots clés : deep learning, sigmoïde, descente de gradient stochastique, rétropropagation, régularisation L2, augmentation artificielle de la base de donnée, initialisation de Xavier.

SUMMARY

This project aims to develop an artificial model capable of recognizing Arabic-Eastern handwritten numerals, using optimization techniques and advanced learning algorithms such as backpropagation and regularization. We have also created a graphical user interface for testing purposes.

Firstly, we imported a database containing images of handwritten numerals to train our system, and we applied appropriate preprocessing to transform the testing data into usable data for our created artificial model.

Next, we implemented a machine learning model, such as a neural network-based classifier, which can be trained using annotated training data by using stochastic gradient descent and back-propagation. Once our model was evaluated and validated, we created a user graphical interface to test it. Finally, we improve the accuracy by using advanced techniques such as regularisation, artificial expanding of data base and Xavier initialisation.

Key words : deep learning, sigmoïd, stochastic gradient descent, back-propagation, L2 Regularisation, artificial expanding of the database, Xavier initialisation.

Table des matières

1	MODELISATION	2
1.1	PERCEPTRON	2
1.2	SIGMOÏDE	4
1.3	ARCHITECTURE DU RNA	5
1.4	FONCTION DE COÛT	6
2	OPTIMISATION	8
2.1	DESCENTE DU GRADIENT	8
2.2	DESCENTE DU GRADIENT STOCHASTIQUE	9
2.3	ALGORITHME DE RÉTROPROPAGATION	9
3	SIMULATION	13
3.1	Fichier Network.py	13
3.2	Fichier training.py	15
3.3	Fichier MAHDBase.py	17
3.4	Prétraitement	18
3.4.1	Fonctions du prétraitement	18
3.4.2	Centrage	20
3.5	Outils et langages utilisées	24
4	AMELIORATION	25
4.1	Entropie croisée (cross entropy)	25
4.2	Régularisation	27
4.2.1	Régularisation L2	27
4.2.2	Augmentation artificielle des données	27
4.3	Initialisation de Xavier	27
4.3.1	Pourquoi n'initialiser pas les poids par une constante ?	27
4.3.2	Problème de Vanishing et Exploding du gradient	28
4.3.3	Justification mathématique	29
4.4	Implémentation	33
5	INTERFACE GRAPHIQUE	34
5.1	Evaluation et validation	34
5.2	Lancement du site	35

Table des figures

1.1	Perceptron	2
1.2	Port NAND implémenté par un perceptron	3
1.3	architecture d 'un RNA	5
1.4	Partie gauche-droite d'un chiffre	6
1.5	Tous les parties d'un chiffre	6
1.6	un arabe-orientau cinq	6
1.7	Training data	7
2.1	Réseau de neurones à trois couches	10
3.1	L'entraînnement du modèle	16
3.2	Fin d'entraînnement	16
3.3	Training data (MAHDBase)	17
3.4	Représentation d'un array d'une image de 8x8 pixels	20
3.5	Repère utilisé	21
3.6	Le vecteur translation	22
3.7	Le premier exemple	23
3.8	Le deuxième exemple	23
4.1	Graphe de la fonction sigmoid	25
4.2	Problème de Vanishing pour les valeurs d'activation	28
4.3	Problème de Vanishing pour les valeurs de la rétro-propagation	28
4.4	Distribution de gausse	30
4.5	Distribution de gauss	30
4.6	Problème de Vanishing est fixé 1	32
4.7	Problème de Vanishing est fixé 2	32
4.8	dernier apprentissage	33
5.1	premier test	34
5.2	deuxième test	35
5.3	site-page 1	35
5.4	site-page 2	36
5.5	site-page 4	36
5.6	site-test 1	37
5.7	site-test 2	37

INTRODUCTION

Le système visuel humain est capable d'exécuter plusieurs tâches spontanément. En suivant d'exemple, nous pouvons reconnaître les différents chiffres manuscrits sans difficultés. Cependant cela présente un grand défi pour la machine. Cette technologie est largement utilisée dans divers domaines tels que la numérisation de documents, la reconnaissance de caractères sur les chèques bancaires et la lecture automatique des codes postaux,etc.. Pour satisfaire le besoin que connaît le domaine de digitalisation des manuscrits arabes, nous avons pris la volonté de réaliser ce projet pour rendre la machine capable de bien identifier une chaîne de caractères, en se basant sur l'apprentissage profond (deep learning), tout en élaborant un réseau de neurones capable de faire un apprentissage performant.

Au cours de notre préparation on a fait face à plusieurs contraintes, tel que le manque des balises ou des connaissances qui peuvent nous guider à un point de départ, donc on a fait recours principalement à des ressources bien connues dans ce domaine de l'intelligence artificielle.

Afin de rendre notre rapport clair et cohérent, nous allons suivre un enchaînement bien structuré :

1. **Modélisation** : traduire notre problème marqué à un modèle mathématique.
2. **Optimisation** : élaborer des algorithmes d'apprentissage pour minimiser l'erreur.
3. **Simulation** : implémenter et entraîner le réseaux de neurones en utilisant NUMPY.
4. **Amélioration** : augmenter la précision en utilisant des techniques avancées.
5. **Evaluation** : essayer le modèle élaboré avec une interface utilisateur.

Chapitre 1

MODELISATION

1.1 PERCEPTRON

Il existe plusieurs types de neurones artificiels. Le modèle principal qui sera être utilisé est celui appelé le neurone sigmoïde que nous aborderons bientôt. Mais pour comprendre pourquoi les neurones sigmoïdes sont définis comme ils le sont, il vaut mieux de prendre le temps pour comprendre tout d'abord les perceptrons.

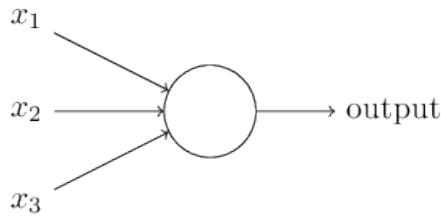


FIGURE 1.1 – Perceptron

La figure ci-dessus montre un perceptron avec 3 entrées binaires x_1, x_2, x_3 et une seule sortie.

Chacune des entrées est caractérisée par un poids (weights) w_j qui représente son importance par rapport à la sortie.

Le perceptron compare la somme pondérée $\sum_j w_j x_j$ avec une valeur appelée seuil (threshold), le neuron retourne 0 si la somme est inférieur ou égale au seuil et 1 dans le cas contraire.

$$\text{output} = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq \text{seuil} \\ 1 & \text{if } \sum_j w_j x_j > \text{seuil} \end{cases} \quad (1.1)$$

Exemple explicatif :

Supposons qu'un ami(e) vous a invité à la plage, votre décision va prendre trois critères en considération, qu'on peut représenter par des variables binaires correspondantes x_1, x_2 , et x_3 .

1. Est ce qu'il y aura du soleil ?

2. Est ce que l'un de vos amis va apporter un ballon de Volley ?
3. Est ce que votre mère va vous préparer un sandwich ?

Maintenant, supposons que vous intréssiez le plus du météo (critère 1), c-à-d vous allez partir si ce jour là sera ensoleillé même si les deux derniers critères ne seront pas validés, et que vous allez refuser l'invitation s'il y aura de brouillard et de brume. On peut utiliser des perceptrons pour modéliser ce type de prise de décision. Pour cela on choisit par exemple un poids $w_1 = 6$ pour la météo, et $w_2 = 2$ et $w_3 = 2$ pour les deux autres conditions. La plus grande valeur de w_1 indique que la météo compte beaucoup pour vous, bien plus que le fait que l'un de vos amis apporte le ballon et que votre mère vous prépare un sandwich. En variant les valeurs des poids et du seuil, on peut obtenir différents modèles de prise de décision. Par exemple, supposons qu'on choisit un seuil de 3. Alors le perceptron va décider que vous devez aller à la plage si votre ami(e) a le ballon et votre mère vous a préparé le sandwich même s'il fait brumé.

Simplification :

On peut simplifier la façon avec laquelle on a décrit le perceptron.

La condition $\sum_j w_j x_j > \text{seuil}$ est lourd, et on peut appliquer deux changements, **le premier** est d'écrire $\sum_j w_j x_j$ sous forme un produit scalaire, $w \cdot x \equiv \sum_j w_j x_j$, où w et x sont deux vecteurs dont les composantes sont respectivement les poids et les valeurs d'entrée (inputs). **le deuxième** changement est de déplacer le seuil à l'autre côté de l'inégalité, et le remplacer par ce qu'on l'appelle bias, $b \equiv -\text{seuil}$. La règle de prise de décision du perceptron peut être écrite comme suit :

$$\text{output} = \begin{cases} 0 & \text{si } w \cdot x + b \leq 0 \\ 1 & \text{si } w \cdot x + b > 0 \end{cases} \quad (1.2)$$

En utilisant le perceptron, on peut modéliser des différents ports logiques comme AND, OR et NAND. On guise d'exemple le perceptron suivant qui implémente le port logique NAND :

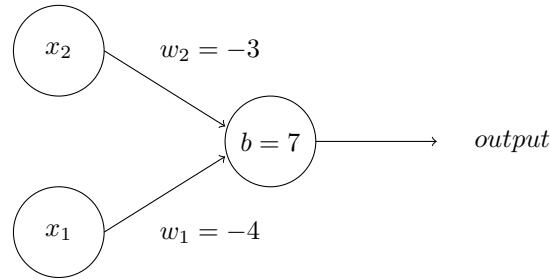


FIGURE 1.2 – Port NAND implémenté par un perceptron

x1	x2	output	$x_1 \text{ NAND } x_2$
0	0	1	1
1	0	1	1
0	1	1	1
1	1	0	0

Cet exemple de NAND qu'on a illustré, montre qu'on peut utiliser un réseau de perceptrons pour évaluer toutes les fonctions logiques. En effet, le NAND est universel.¹

1.2 SIGMOÏDE

Pour des raisons pratiques, On préfère d'utiliser un autre type de neurone appelé *neurone sigmoïde*(perceptrons multi-couches). Comme le perceptron, ce type de neurone choisi admet des entrées x_1, x_2, \dots qui peuvent avoir des valeurs continues entre 0 et 1 et non seulement des valeurs binaires, cela s'applique également pour la sortie. Autrement dit la sortie est calculée en utilisant $\sigma(w \cdot x + b)$ avec σ est la fonction *sigmoid* définie par :

$$\sigma(z) \equiv \frac{1}{1 + e^{-z}}.$$

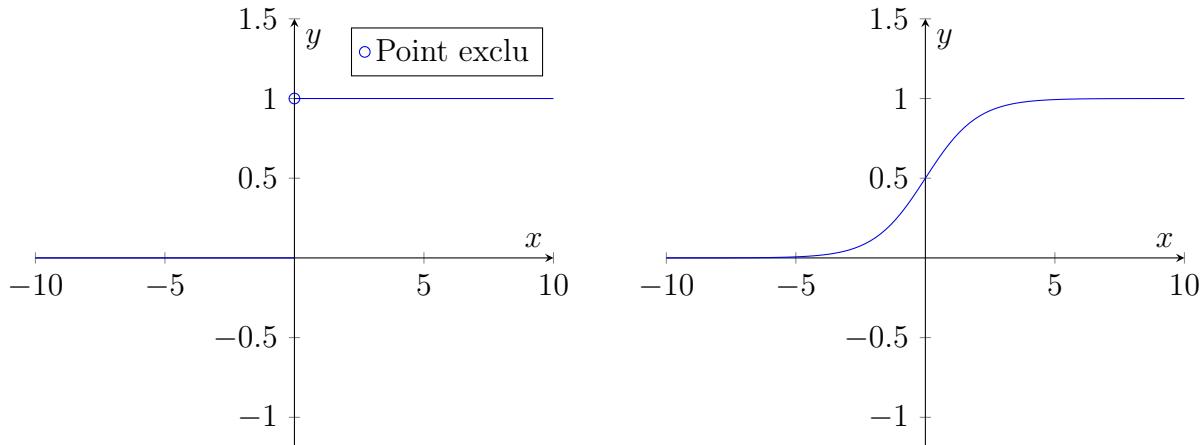
ou explicitement :

$$\frac{1}{1 + \exp(-\sum_j w_j x_j - b)}.$$

Au premier moment, les neurones sigmoïdes semblent différents au perceptron, mais en réalité leurs comportements sont similaires.

En effet :

- $\lim_{z \rightarrow \infty} \sigma(z) = \lim_{z \rightarrow \infty} \frac{1}{1+e^{-z}} = 1$
- $\lim_{z \rightarrow -\infty} \sigma(z) = \lim_{z \rightarrow -\infty} \frac{1}{1+e^{-z}} = 0$



La transition entre le perceptron et le sigmoïde neurone est justifiée par le fait que la fonction sigmoïde est dérivable alors que la fonction Heaviside² n'est pas dérivable en 0 et la

1. Elle permet d'exprimer toutes les fonctions de base : OUI,NON,ET,OU
 2. la fonction échelon avec $H(0) = 0$

dérivé ailleurs est nulle. Cette notion de dérivabilité est très importante car elle représente un principe solide sur lequel les algorithmes d'apprentissage se sont basés.

1.3 ARCHITECTURE DU RNA

Pour la reconnaissance des chiffres arabes-orientaux manuscrits, on va utiliser un Réseau Neuronal Artificiel (RNA) composé de trois couches :

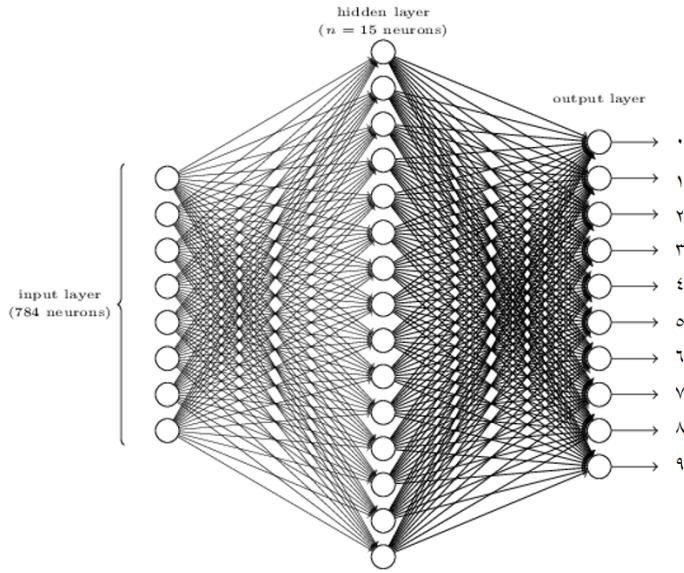


FIGURE 1.3 – architecture d'un RNA

La couche d'entrée du RN contient 784 entrées ayant comme valeurs l'intensité du noir des pixels d'entrée d'une grayscale image de taille $784 = 28 \times 28$. L'intensité 0.0 représente un pixel blanc et 1.0 représente un pixel noir, alors que les valeurs comprises entre 0.0 et 1.0 représentent le degré d'obscurité du pixel.

La couche-cachée (hidden layer) du RN est composées de n neurones. Ce nombre n est choisi expérimentalement et il n'y a pas de méthode précise pour le déterminer. Encore faut-il ajouter qu'on ne sait pas sûre en ce qui concerne le fonctionnement de cette couche et qu'on essaie seulement de poser des hypothèses probables.

La couche de sortie (output layer) contient 10 neurons. Si le premier neurone prend comme valeur de sortie ≈ 1 alors cela indique que le RNA pense que le chiffre d'entrée est 0. Plus précisément, on calcule les sorties des neurones de 0 à 9 et on choisit le neurone ayant la plus grande valeur d'activation.

Maintenant, on passe à présenter l'une des hypothèses probables sur le fonctionnement de notre seule couche-cachée. Alors, concentrons-nous sur le 6ème neurone de notre couche

de sortie consacrée à décider si le digit d'entrée est un 5 ou pas. Supposons que le premier neurone de la couche cachée détecte si une image comme la suivante est présente ou pas :

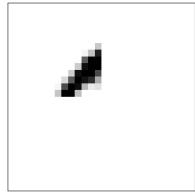


FIGURE 1.4 – Partie gauche-droite d'un chiffre

Il peut la faire d'une façon continue en pondérant les poids synaptiques d'entrée et détecter si elles sont similaires à ces images. Supposons que cela s'applique sur tous les autres neurones, le deuxième, le troisième et le quatrième neurone de la couche-cachée détectent si les images suivants sont présentes :

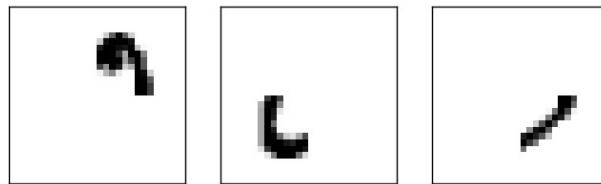


FIGURE 1.5 – Tous les parties d'un chiffre

Comme vous voyez, si on rassemble ces images on obtient une image d'un 5 en arabe.

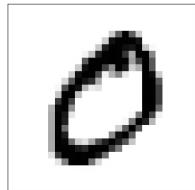


FIGURE 1.6 – un arabe-orientau cinq

1.4 FONCTION DE COÛT

Comme notre RNA est prêt, comment pourra-t-il donc apprendre à reconnaître les chiffres arabes manuscrits ?

Premièrement, on va utiliser MAHD Base³ contenant 70 000 images scanées des chiffres manuscrits, 60 000 pour entraîner le modèle artificiel et 10 000 pour le tester. Voici des exemples de ces images :

3. Modified Arabic Handwritten Digits Base Lien de MAHD Base

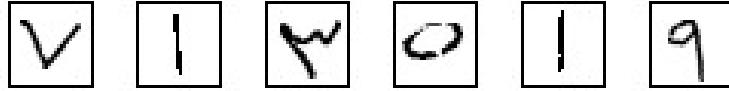


FIGURE 1.7 – Training data

On note les entrées d’entraînement par x , un vecteur colonne de dimension $784 = 28 \times 28$ chaque élément représente la valeur d’intensité du noir de chaque pixel de notre image scannée, et on note la sortie désirée par $y(x)$ un vecteur colonne de dimension 10. Pour plus d’explication, on guise d’exemple, si une entrée particulière est une image de 4 alors la sortie désirée s’écrit comme suit $y(x) = (0, 0, 0, 0, 1, 0, 0, 0, 0, 0)^T$.

Ce qu’on cherche est un algorithme qui ajuste les poids synaptiques et les biais de tel sorte que la sortie soit très proche de $y(x)$ pour tout x . Pour mesurer combien on est proche de ce but, on définit la fonction suivante appelée fonction de coût⁴ :

$$C(w, b) \equiv \frac{1}{2n} \sum_x \|y(x) - a\|^2.$$

On note par w le vecteur colonne dont les éléments sont toutes les poids synaptiques de notre RNA, b le vecteur colonne dont les éléments sont tous les biais, n le nombre totale de toutes les images d’entraînement et $a = \sigma(w \cdot x + b)$ la sortie correspondante à chaque x . On appelle C la fonction de coût quadratique⁵. Il est aussi clair que C est positive et tend vers 0 quand a s’approche de $y(x)$. Donc le but de notre algorithme d’entraînement est de trouver des valeurs de w et de b pour que la fonction $C(w, b)$ soit minimisée. Pour cela on va utiliser un algorithme connu sous *Descente de gradient*.

4. appelée parfois fonction objective ou fonction économique

5. MSE (mean squared error)

Chapitre 2

OPTIMISATION

2.1 DESCENTE DU GRADIENT

La descente du gradient est un algorithme très célèbre qui cherche à déterminer le jeu de paramètres d'entrée de la fonction C donnant à cette fonction une valeur minimale. Il consiste à mettre à jour tous les poids et les biais selon la règle suivante :

$$w_k \rightarrow w'_k = w_k - \eta \frac{\partial C}{\partial w_k}$$

$$b_l \rightarrow b'_l = b_l - \eta \frac{\partial C}{\partial b_l}.$$

Démonstration. Soit $C(v) = C(v_1, v_2, v_3, \dots, v_n)$ une fonction de n variables réelles à valeurs réelles dans \mathbb{R} .

L'algorithme de la descente du gradient définit par l'algorithme

$$v \rightarrow v' = v - \eta \nabla C \quad (2.1)$$

où η est le pas d'apprentissage et ∇C est le gradient de la fonction C .

Pour minimiser C , nous devons avoir un ΔC négatif.

On peut approximer ΔC par $\nabla C \cdot \Delta v$. Fixons la longueur du pas (la norme de Δv), notée $\|\Delta v\| = \epsilon$, avec $\epsilon > 0$.

Nous avons fixé la norme du pas pour chercher la direction qui réalise la plus grande descente de C .

D'après l'inégalité de Cauchy-Schwarz, on a :

$$\begin{aligned} |\nabla C \cdot \Delta v| &\leq \|\nabla C\| \cdot \|\Delta v\| \\ \implies -|\nabla C \cdot \Delta v| &\geq -\|\nabla C\| \cdot \|\Delta v\| \\ \implies \nabla C \cdot \Delta v &\geq -\|\nabla C\| \cdot \|\Delta v\| \end{aligned}$$

D'où la valeur minimale (la plus négative) que nous pouvons obtenir est :

$$\begin{aligned}
\nabla C \cdot \Delta v &= -\|\nabla C\| \cdot \|\Delta v\| \\
&= -\|\nabla C\| \cdot \epsilon \\
&= -\frac{\|\nabla C\|^2}{\|\nabla C\|} \cdot \epsilon \\
&= -\frac{\nabla C \cdot \nabla C}{\|\nabla C\|} \cdot \epsilon \\
&= \nabla C \cdot \left(-\frac{\epsilon}{\|\nabla C\|} \cdot \nabla C \right)
\end{aligned}$$

Alors $\Delta v = -\eta \cdot \nabla C$ en remplaçant $-\frac{\epsilon}{\|\nabla C\|}$ par η

□

On peut écrire la fonction de coût C comme une moyenne $C = \frac{1}{n} \sum_x C_x$ des coûts individuels $C_x \equiv \frac{\|y(x) - a\|^2}{2}$. En pratique, pour calculer le gradient ∇C , on calcule les ∇C_x pour chaque x et puis on calcule leur moyenne $\nabla C = \frac{1}{n} \sum_x \nabla C_x$.

2.2 DESCENTE DU GRADIENT STOCHASTIQUE

Il est clair qu'effectuer le calcul de la descente du gradient à chaque itération (epoch) va devenir très coûteux en temps et ressource machine. Alors au lieu de prendre toutes les données, on va travailler sur des échantillons(mini-batch) choisis aléatoirement à chaque epoch.

c'est exactement ce que propose une descente du gradient stochastique(ou SGD). C'est un algorithme avancé qui se focalise sur des échantillons différents du jeu de données global à chaque itération. Plus précisément, on choisit un petit nombre m d'échantillons aléatoires X_1, X_2, \dots, X_m . m doit être suffisamment large pour que la moyenne des ∇C_{X_j} soit approximative à la moyenne des ∇C_x :

$$\frac{\sum_{j=1}^m \nabla C_{X_j}}{m} \approx \frac{\sum_x \nabla C_x}{n} = \nabla C$$

Alors l'algorithme de la descente du gradient stochastique s'écrit comme suit :

$$w_k \rightarrow w'_k = w_k - \frac{\eta}{m} \sum_j \frac{\partial C_{X_j}}{\partial w_k}$$

$$b_l \rightarrow b'_l = b_l - \frac{\eta}{m} \sum_j \frac{\partial C_{X_j}}{\partial b_l}$$

2.3 ALGORITHME DE RÉTROPROPAGATION

Pour appliquer l'algorithme de la descente du gradient stochastique, nous avons besoin de calculer les dérivées partielles $\partial C / \partial w_{jk}^l$ et $\partial C / \partial b_j^l$.

Commençons par les notations w_{jk}^l et b_j^l . On va utiliser la notation w_{jk}^l pour désigner le poids de la connection du k ème neurone de la couche $(l - 1)$ avec le j ème neurone de la couche l . Par exemple, le diagramme suivant illustre :

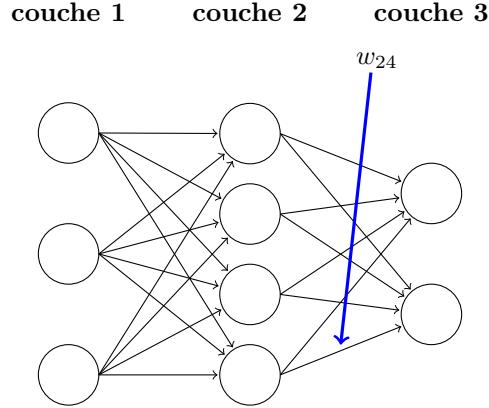


FIGURE 2.1 – Réseau de neurones à trois couches

Alors, la figure ci-dessus montre le poids synaptique w_{14}^3 qui connecte le 4ème neurone de la 2ème couche avec le 1er neurone de la 3ème couche. Or b_j^l désigne le seuil du j ème neurone de la couche l .

Avec ces notations, l'activation a_j^l du j ème neurone de la couche l est liée aux activations de la couche $l - 1$ selon l'équation :

$$a_j^l = \sigma \left(\sum_k w_{jk}^l a_k^{l-1} + b_j^l \right)$$

Pour simplifier cette expression, on peut l'écrire sous la forme vectorielle $a^l = \sigma(w^l a^{l-1} + b^l) = \sigma(z^l)$.

On définit une quantité très utile prochainement qui représente l'erreur du j ème neurone de la l ème couche :

$$\delta_j^l = \frac{\partial C}{\partial z_j^l}$$

La rétropropagation est basée sur quatre équations fondamentales qui permettent de calculer l'erreur δ_j^l et le gradient de la fonction du coût.

a. $\delta_j^L = \frac{\partial C}{\partial a_j^L} \sigma'(z_j^L)$

b. $\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$

c. $\frac{\partial C}{\partial b_j^l} = \delta_j^l$

d. $\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l$

Démonstration. Commençons tout d'abord par la première équation qui donne l'erreur de la sortie :

$$\delta_j^L = \frac{\partial C}{\partial a_j^L} \sigma'(z_j^L)$$

On a d'après la définition :

$$\delta_j^L = \frac{\partial C}{\partial z_j^L}$$

En appliquant la règle de la chaîne, on aura :

$$\delta_j^L = \sum_k \frac{\partial C}{\partial a_k^L} \frac{\partial a_k^L}{\partial z_j^L}$$

Et puisque l'activation a_k^L du kème neurone ne dépend que de l'entrée pondérée z_j^L du jème neurone quand $j = k$ alors la dérivée $\partial a_k^L / \partial z_j^L$ s'annule quand $k \neq j$.

On peut donc simplifier l'équation précédente :

$$\delta_j^L = \frac{\partial C}{\partial a_j^L} \frac{\partial a_j^L}{\partial z_j^L}$$

En rappelant que $a_j^L = \sigma(z_j^L)$, on peut écrire le terme à droite de l'équation obtenue comme suit $\sigma'(z_j^L)$

Et l'équation devient :

$$\delta_j^L = \frac{\partial C}{\partial a_j^L} \sigma'(z_j^L)$$

Passons maintenant à la démonstration de la deuxième équation qui exprime l'erreur δ^l en fonction de l'erreur de la couche suivante δ^{l+1} :

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$$

Pour ce faire, on peut reformuler $\delta_j^l = \partial C / \partial z_j^l$ en fonction de $\delta_k^{l+1} = \partial C / \partial z_k^{l+1}$ en utilisant la règle de la chaîne :

$$\delta_j^l = \frac{\partial C}{\partial z_j^l} = \sum_k \frac{\partial C}{\partial z_k^{l+1}} \frac{\partial z_k^{l+1}}{\partial z_j^l} = \sum_k \frac{\partial z_k^{l+1}}{\partial z_j^l} \delta_k^{l+1}$$

On note :

$$z_k^{l+1} = \sum_j w_{kj}^{l+1} a_j^l + b_k^{l+1} = \sum_j w_{kj}^{l+1} \sigma(z_j^l) + b_k^{l+1}$$

Puis on dérive :

$$\frac{\partial z_k^{l+1}}{\partial z_j^l} = w_{kj}^{l+1} \sigma'(z_j^l)$$

En remplaçant dans l'équation de δ_j^l , on obtient :

$$\delta_j^l = \sum_k w_{kj}^{l+1} \delta_k^{l+1} \sigma'(z_j^l)$$

De même, pour les deux dernières équations. □

Qu'elle est l'utilité de ces quatre équations ?

On a déjà motré qu'on s'intéresse principalement à calculer les dérivées $\frac{\partial C}{\partial w_{jk}^l}$ et $\frac{\partial C}{\partial b_j^l}$ et pour ce faire, on doit passer par le calcule des erreurs δ_j^l et δ_j^L (comme montrent les deux dernières équations de l'algorithme de la rétropropagation (c) et (d)).

Or, on ne peut calculer que l'erreur de la couche de sortie δ_j^L (voir l'équation (a)), on utilise l'erreur δ_j^l pour calculer l'erreur de la couche qui la précède δ_j^{l-1} (voir l'équation (b)). D'où vient le nom de la rétropropagation.

Chapitre 3

SIMULATION

3.1 Fichier Network.py

Expliquons tout d'abord la fonctionnalité de base de notre code de réseau de neurones. La classe *Network* est la classe qu'on va utiliser pour représenter notre RN. Voici le code qui initialise un objet *Network* :

```
1 import random
2 import numpy as np
3
4 class Network(object):
5
6     def __init__(self, sizes):
7         self.num_layers = len(sizes)
8         self.sizes = sizes #liste des nombres de neurons des couches respectives
9         self.biases = [np.random.randn(y, 1) for y in sizes[1:]]
10        self.weights = [np.random.randn(y, x)
11                        for x, y in zip(sizes[:-1], sizes[1:])]
```

Les poids et les biais sont initialisés aléatoirement en utilisant la fonction *numpy.random.randn*.

On donne l'initialisation de la fonction sigmoïde qui admet comme entrée un vecteur z et retourne un vecteur aussi.

```
1 def sigmoid(z):
2     return 1.0/(1.0+np.exp(-z))
```

On ajoute la méthode *feedforward* à la classe *Network* admettant une entrée a (vecteur colonne de dimension 784) et retournant la sortie calculée.

```
1 def feedforward(self, a):
2     """Return the output of the network if "a" is input."""
3     for b, w in zip(self.biases, self.weights):
4         a = sigmoid(np.dot(w, a)+b)
5     return a
```

Bien sûr que notre but principal est de construire un objet *Network* capable d'apprendre en implémentant l'algorithme de la descente du gradient stochastique.

```

1 def SGD(self, training_data, epochs, mini_batch_size, eta, test_data=None):
2     ''' training data est une liste de 50 000 tuples (x, y)
3         avec x est l'entrée et y est sa sortie désirée.
4         Les variables epochs et mini_batch_size et eta représentent
5         respectivement le nombre des itérations d'ajustement
6         et la taille des échantillons et le taux d'apprentissage'''
7
8     if test_data: n_test = len(test_data)
9     n = len(training_data)
10    for j in range(epochs):
11        random.shuffle(training_data)
12        mini_batches = [
13            training_data[k:k+mini_batch_size]
14            for k in range(0, n, mini_batch_size)]
15        for mini_batch in mini_batches:
16            self.update_mini_batch(mini_batch, eta)
17        if test_data:
18            print("Epoch {0}: {1} / {2}".format(
19                j, self.evaluate(test_data), n_test))
20        else:
21            print("Epoch {0} complete".format(j))

```

On mélange aléatoirement notre training-data et on la divise à des échantillons de taille mini_batch_size et pour chacune des échantillons, on applique un seul pas de la descente du gradient et cela est fait par l'instruction `self.update_mini_batch(mini_batch, eta)`.

```

1 def update_mini_batch(self, mini_batch, eta):
2     nabla_b = [np.zeros(b.shape) for b in self.biases]
3     nabla_w = [np.zeros(w.shape) for w in self.weights]
4     for x, y in mini_batch:
5         delta_nabla_b, delta_nabla_w = self.backprop(x, y)
6         nabla_b = [nb+dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
7         nabla_w = [nw+dnw for nw, dnw in zip(nabla_w, delta_nabla_w)]
8         self.weights = [w-(eta/len(mini_batch))*nw
9                         for w, nw in zip(self.weights, nabla_w)]
10        self.biases = [b-(eta/len(mini_batch))*nb
11                      for b, nb in zip(self.biases, nabla_b)]

```

La majorité du travail est faite par le code `nabla_b, nabla_w = self.backprop(x, y)`

La méthode *backprop* retourne un tuple de deux listes qui contiennent respectivement $\partial C_x / \partial b_j^l$ et $\partial C_x / \partial w_j^l$ en utilisant des petites fonctions assistantes pour calculer les fonctions σ et σ' .

```

1 def backprop(self, x, y):
2     nabla_b = [np.zeros(b.shape) for b in self.biases]
3     nabla_w = [np.zeros(w.shape) for w in self.weights]
4     # feedforward
5     activation = x
6     activations = [x]
7     zs = []

```

```

8     for b, w in zip(self.biases, self.weights):
9         z = np.dot(w, activation)+b
10        zs.append(z)
11        activation = sigmoid(z)
12        activations.append(activation)
13    # backward pass
14    delta = self.cost_derivative(activations[-1], y) * \
15        sigmoid_prime(zs[-1])
16    nabla_b[-1] = delta
17    nabla_w[-1] = np.dot(delta, activations[-2].transpose())
18
19    for l in range(2, self.num_layers):
20        z = zs[-l]
21        sp = sigmoid_prime(z)
22        delta = np.dot(self.weights[-l+1].transpose(), delta) * sp
23        nabla_b[-l] = delta
24        nabla_w[-l] = np.dot(delta, activations[-l-1].transpose())
25    return (nabla_b, nabla_w)
26
27 def evaluate(self, test_data):
28     test_results = [(np.argmax(self.feedforward(x)), y)
29                     for (x, y) in test_data]
30     return sum(int(x == y) for (x, y) in test_results)
31
32 def cost_derivative(self, output_activations, y):
33     return (output_activations-y)
34
35 ##### Miscellaneous functions
36 def sigmoid(z):
37     """The sigmoid function."""
38     return 1.0/(1.0+np.exp(-z))
39
40 def sigmoid_prime(z):
41     """Derivative of the sigmoid function."""
42     return sigmoid(z)*(1-sigmoid(z))

```

3.2 Fichier training.py

Le code ci-dessous implémente l'apprentissage profond de la machine, en téléchargeant la base de données mentionnée précédemment (MAHDBase), créant le modèle artificiel *net* et puis commençant l'entraînement. Finalement, on a enregistré le réseau entraîné dans un fichier binaire (objet **Pickle**).

```

1 #training.py file
2 #Libraries
3 import MAHDBase
4 import network
5 import pickle
6 import numpy as np
7 #Telecharger training_data et test_data
8 training_data=MAHDBase.load_training_data ()
9 test_data=MAHDBase.load_test_data()

```

```

10
11 #Creer le reseau de neurons
12 net = network.Network([784,30,10])
13
14 #Entrainner notre modele artificiel
15 net.SGD(training_data, 30, 10, 3, test_data=test_data)
16
17 #Enregistrer notre Reseau
18 f=open('fileh.pickle','wb')
19 pickle.dump(net,f)
20 f.close()

```

On lance l'apprentissage de notre modèle artificiel pendant 30 itérations :

```

useer@ubuntu3: ~/Desktop/project
File Edit View Search Terminal Help
(base) useer@ubuntu3:~$ conda activate myenv
(myenv) useer@ubuntu3:~$ cd Desktop/project/
(myenv) useer@ubuntu3:~/Desktop/project$ python training.py
Epoch 0: 9477 / 10000
Epoch 1: 9570 / 10000
Epoch 2: 9625 / 10000
Epoch 3: 9630 / 10000
Epoch 4: 9665 / 10000
Epoch 5: 9678 / 10000
Epoch 6: 9658 / 10000
Epoch 7: 9678 / 10000

```

FIGURE 3.1 – L’entraînement du modèle

A la fin de notre apprentissage, notre modèle atteint une précision de 97.19%

```

Epoch 25: 9711 / 10000
Epoch 26: 9712 / 10000
Epoch 27: 9726 / 10000
Epoch 28: 9715 / 10000
Epoch 29: 9719 / 10000

```

FIGURE 3.2 – Fin d’entraînement

3.3 Fichier MAHDBase.py

MAHDBase est une base de données où les données sont présentées sous forme des images .bmp et comme on a besoin des données sous forme un tuple de deux liste x et y , alors nous étions obligé de faire un prétraitement pour préparer notre training data et test data.

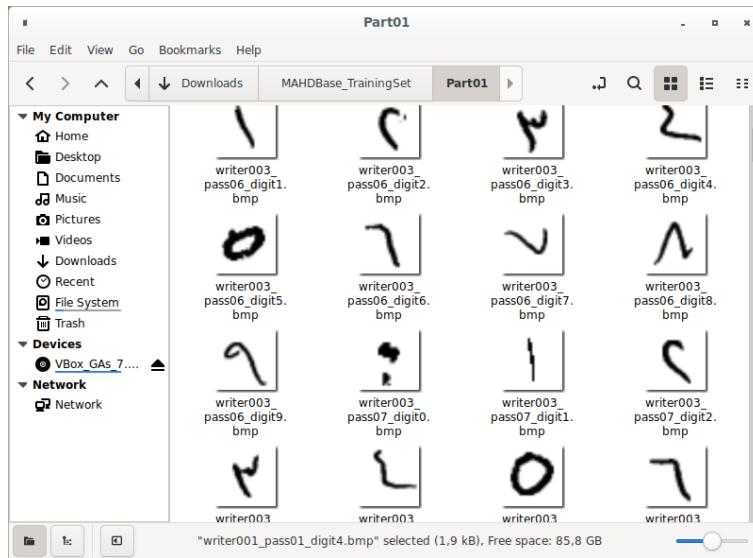


FIGURE 3.3 – Training data (MAHDBase)

On a parcouris toutes les images du dossier *MAHDBase – TrainingSet*, et on a transformé chacune des images en une matrice carrée de dimension 28, puis on l'a transformé en un vecteur colonne de dimension 784, après on a extrais la valeur exacte (dite désirée) du nom d'image est qui représente *filename*[−5] (par exemple : l'image "writer151-pass01-digit1.bmp" sa valeur est *filename*[−5]=1), et finalement on a vectorisé cette valeur exacte en utilisant la fonction *vectorised*.

Le même processus est appliqué également sur les données du dossier *MAHDBase – TestingSet* sauf que la valeur exacte ne sera pas vectorisée.

```
1 #MAHDBase.py file
2 #libraries :
3 import os
4 import pickle
5 import numpy as np
6 from PIL import Image
7
8 #functions :
9 def Vectorized (i):
10     vect=np.zeros((10,1))
11     vect[i]=1.0
12     return vect
13
14 def transform (x):
```

```

15     return 1-x/255
16
17 def load_training_data():
18     main_folder = "C:/Users/ZAHAR AMINA/Desktop/projet IA/MAHDBase_TrainingSet"
19     training_data=[]
20     for root, dirs, files in os.walk(main_folder):
21         for filename in files:
22             if filename.endswith(".bmp"):
23                 with Image.open(os.path.join(root, filename)) as img:
24                     img=np.array(img)
25                     img=transform(img)
26                     img=np.reshape(img, (784,1))
27                     n=np.int64(filename[-5])
28                     vect=Vectorized(n)
29                     training_data.append((img,vect))
30     return training_data
31
32 def load_test_data():
33     main_folder = "C:/Users/ZAHAR AMINA/Desktop/projet IA/MAHDBase_TestingSet"
34     testing_data=[]
35     for root, dirs, files in os.walk(main_folder):
36         for filename in files:
37             if filename.endswith(".bmp"):
38                 with Image.open(os.path.join(root, filename)) as img:
39                     img=np.array(img)
40                     img=transform(img)
41                     img=np.reshape(img, (784,1))
42                     n=np.int64(filename[-5])
43                     testing_data.append((img,n))
44     return testing_data

```

3.4 Prétraîtement

Avant de réaliser un prétraîtement des images qu'on a donné à notre modèle artificiel, nous étions obligés de lire le protocole adopté par les créateurs du Data Set afin d'ajuster les images avant les donner à notre modèle entraîné :

Normalisation de taille

Pour chaque chiffre on calcule sa longueur h et sa largeur w , puis on normalise sa taille pour obtenir des nouvelles dimensions h_{new} et w_{new} . Les valeurs attribuées à ces derniers dépendent de si l'une est plus grande que l'autre. Autrement dit, Si $h > w$, alors $h_{new} = 20$ et $w_{new} = \text{floor}(20 \times w/h)$, et si $w > h$, alors $w_{new} = 20$ et $h_{new} = \text{floor}(20 \times h/w)$.

Centrage

Après avoir normalisé le chiffre dans un bounding-box, on va procéder par un centrage du barycentre.

3.4.1 Fonctions du prétraitemet

```

1 #FunctionsApp.py
2 import numpy as np
3 from PIL import Image

```

```

4
5 def sigmoid (z):
6     return 1.0/(1.0+np.exp(-z))
7
8 def feedforward (a,w,b):
9     for b1, w1 in zip(b,w):
10        a = sigmoid(np.dot(w1, a)+b1)
11    return a
12
13 def transform (x):
14     return 1-x/255
15
16 #Bounding Box functions :
17 def invert_img(image):
18     arr=np.array(image)
19     arr=255-arr
20     return Image.fromarray(arr)
21
22 def size_normalisation (img):
23     img=invert_img(img)
24     img=img.crop(img.getbbox())
25     #Resizing :
26     h=img.height
27     w=img.width
28     if h>w:
29         img=img.resize((int(np.floor(20*w/h)), 20))
30     else:
31         img=img.resize((20, int(np.floor(20*h/w))))
32     return img
33
34 # Centring functions :
35 def invert_arr(arr):
36     arr = 255-arr
37     return arr
38
39 def massCenter(arr):
40     arr = invert_arr(arr)
41     totalMass = 0
42     sumX = 0
43     sumY = 0
44     for y in range(arr.shape[0]):
45         for x in range(arr.shape[1]):
46             totalMass += arr[y,x]
47             sumX += arr[y,x] * (x+0.5)
48             sumY += arr[y,x] * (y+0.5)
49     centerX = sumX/totalMass
50     centerY = sumY/totalMass
51     return (centerX, centerY)
52
53 def transVector(arr):
54     x, y = massCenter(arr)
55     Cx, Cy = (arr.shape[1]/2, arr.shape[0]/2)
56     return (int(np.round(Cx-x)), int(np.round(Cy-y)))
57
58 def center(img):
59     arr = np.array(img)

```

```

60 Tx, Ty = transVector(arr)
61 arr = invert_arr(arr)
62 newArr = np.zeros(arr.shape)
63 for y in range(arr.shape[0]):
64     for x in range(arr.shape[1]):
65         if (arr[y, x] >= 0 and (0 <= y+Ty <=arr.shape[0]-1) and (0 <= x+Tx <
66             arr.shape[1]-1) ):
67             newArr[y+Ty, x+Tx] = arr[y, x]
68 newArr = invert_arr(newArr)
69 img = Image.fromarray(newArr)
70 return img

```

La fonction **size-normalisation** représente l'implémentation de la première procédure (ligne 16).

La fonction **center** implémente la deuxième procédure que nous allons expliquer en détails tout à l'heure.

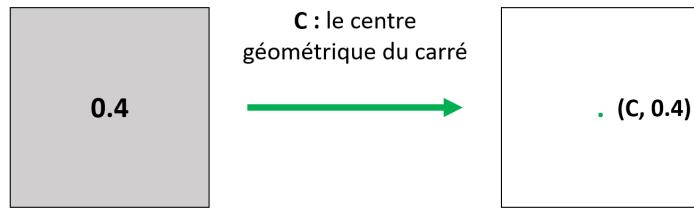
3.4.2 Centrage

On peut représenter une image par une matrice admettant comme valeurs les poids des pixels.

Indexes	0	1	2	3	4	5	6	7
0	0	0	0	0.4	0	0	0	0
1	0	0	0.6	0.7	0.6	0	0	0
2	0	0	0.6	0.9	0.6	0	0	0
3	0	0	0.6	0.9	0.6	0	0	0
4	0	0	0.6	0.9	0.6	0	0	0
5	0	0	0.6	0.9	0.6	0	0	0
6	0	0	0.6	0.6	0.6	0	0	0
7	0	0	0	0	0	0	0	0

FIGURE 3.4 – Représentation d'un array d'une image de 8x8 pixels

Chaque pixel de notre image a un poids selon l'intensité du noir (0 pour le blanc et 1 pour le noir). Pour une modélisation précise du problématique, on considère que chaque pixel est un carré plein uniforme en masse, et que la masse totale de chaque pixel est le poids correspondant. Il est évident que chaque pixel a comme centre de masse son centre géométrique. On prend donc le centre de masse de chaque pixel est son centre géométrique pondéré et comme poids le poids initial du pixel.



On utilise un repère comme illustré la figure ci-dessous, avec o , \vec{i} et \vec{j} représentent respectivement l'origine et les vecteurs unités des axes d'abscisses et d'ordonnées.

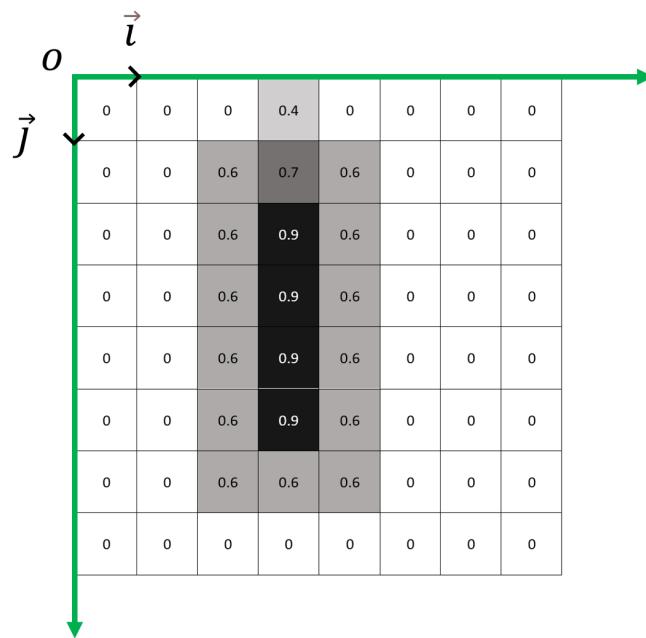


FIGURE 3.5 – Repère utilisé

Pour mieux comprendre, on note cet array arr . Le pixel $arr[0][0]$ a comme centre de masse le point $(0.5, 0.5)$, pour $arr[7][7]$ on a $(7.5, 7.5)$ et pour $arr[1][3]$ on a $(3.5, 1.5)$

Calcul du centre de masse

Pour chaque pixel, on a un centre de masse présenté par un point ponderé. On calcule alors

le centre de masse de ces centres. On le détermine avec les deux formules suivantes :

$$x_{cm} = \frac{\sum_{i,j} m_{i,j} x_{i,j}}{M_{Tot}}$$

$$y_{cm} = \frac{\sum_{i,j} m_{i,j} y_{i,j}}{M_{Tot}}$$

Avec :

- x_{cm}, y_{cm} sont respectivement l'abscisse et l'ordonné du centre de masse de notre array
- $m_{i,j}, x_{i,j}, y_{i,j}$ sont respectivement le poids, l'abscisse et l'ordonné du centre de masse du pixel $arr[i][j]$
- M_{Tot} est la somme de masses des tous les pixels

Détermination du vecteur de translation

D'abord, considérons notre repère précédent. Le centre de notre image ou array est le point $(8/2, 8/2)$ pour l'exemple. En générale le centre est $(arr.shape[1]/2, arr.shape[0]/2)$

Si M le centre de masse et C le centre de l'image, alors le vecteur de translation pour centrer notre image est $\vec{V} = C - M$

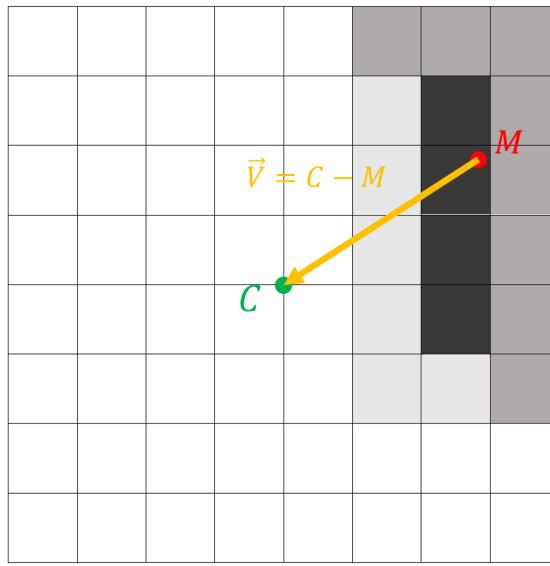


FIGURE 3.6 – Le vecteur translation

Donc dans cette étape, il nous reste que de translater chaque pixel par le vecteur \vec{V} pour obtenir le résultat cherché.

Un dernier problème figure, c'est qu'on obtient souvent des vecteurs de la forme $\vec{V} = \begin{pmatrix} -2.88 \\ 1.91 \end{pmatrix}$, translater un pixel par 2.88 à gauche et 1.91 en bas est impossible pour un array.

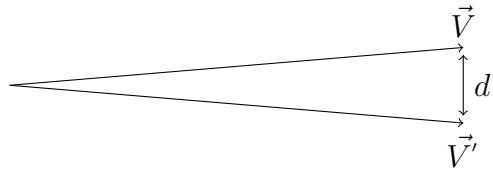
Donc on va approximer \vec{V} avec un vecteur \vec{V}' ayant des composants entiers.

On peut utiliser la partie entière ou bien la fonction `ceil`. Mais Il sera mieux d'utiliser l'arrondi.`(numpy.round)`.

En effet :

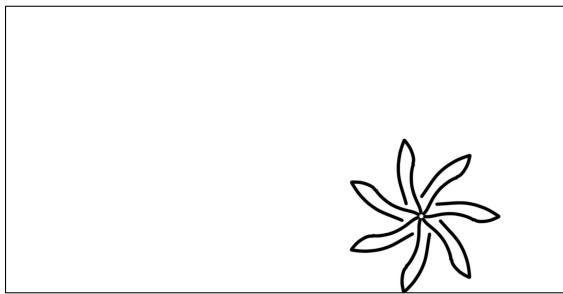
$$d = |\vec{V} - \vec{V}'| = \sqrt{(V_x - V'_x)^2 + (V_y - V'_y)^2}$$
 est minimale lorsque

$$\vec{V}' = \begin{pmatrix} V'_x \\ V'_y \end{pmatrix} = \begin{pmatrix} \text{numpy.round}(V_x) \\ \text{numpy.round}(V_y) \end{pmatrix}$$

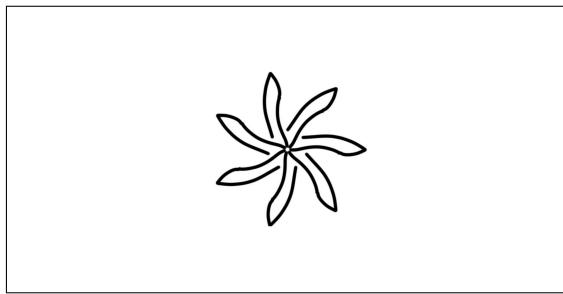


L'implémentation du centrage commence de la ligne 34 du code *FunctionsApp.py*.

Exemples d'images centrées par notre fonction :

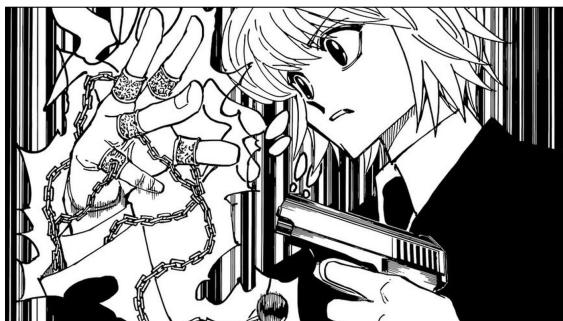


(a) L'image avant la centrage



(b) L'image après la centrage

FIGURE 3.7 – Le premier exemple



(a) L'image avant la centrage

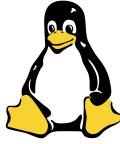


(b) L'image après la centrage

FIGURE 3.8 – Le deuxième exemple

Remarque : Comme illustre le deuxième exemple, On perd toujours des pixels de l'image initiale puisque la translation se fait sur les mêmes frontières de l'image.

3.5 Outils et langages utilisées



Linux : Système d'exploitation où on a crée et entrainné notre réseau.



Visual Studio code : éditeur de code extensible où on crée notre site web.



Python3 : Langage de programmation open source interprétée très utilisé dans plusieurs domaines notamment machine learning.



HTML5 : Langage de balisage pour écrire l'hypertexte et structurer sémantiquement une page web.



CSS : Langage de style qui permet de mettre en forme les éléments d'une page web.



Java Script : Langage de programmation de scripts principalement employé dans les pages web interactives.

Chapitre 4

AMELIORATION

Dans ce chapitre, on va essayer d'améliorer la précision et la façon avec laquelle notre réseau apprend.

4.1 Entropie croisée (cross entropy)

Comme l'initialisation des poids synaptiques se fait aléatoirement, parfois cette initialisation peut engendrer un lent apprentissage si sa valeur est **saturée** dans une valeur incorrecte. En effet la fonction du coût s'écrit :

$$C(w, b) \equiv \frac{1}{2n} \sum_x \|y(x) - a\|^2$$

Et les dérivées partielles :

$$\frac{\partial C}{\partial w} = \sum_x (a - y)\sigma'(z)x$$

$$\frac{\partial C}{\partial b} = \sum_x (a - y)\sigma'(z)$$

Il est clair que l'origine du problème de "slow down learning" est dû du fait que la dérivée $\sigma'(z)$ devient plus petite quand z est saturée, c-à-d la valeur de z est très grande ou très petite.

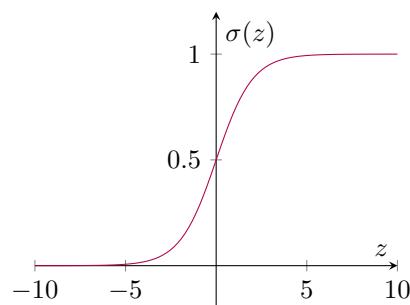


FIGURE 4.1 – Graph de la fonction sigmoid

On va donc, remplacer la fonction du coût quadratique par l'entropie croisée, afin de se débarasser de ce terme $\sigma'(z)$.

On cherche une fonction C qui vérifie :

$$\begin{aligned}\frac{\partial C}{\partial w_j} &= x_j(a - y) \\ \frac{\partial C}{\partial b} &= (a - y)\end{aligned}$$

Notons que d'après la dérivée des fonction composées, on a :

$$\frac{\partial C}{\partial b} = \frac{\partial C}{\partial a} \sigma'(z)$$

En remplaçant avec $\sigma'(z) = \sigma(z)(1 - \sigma(z)) = a(1 - a)$:

$$\frac{\partial C}{\partial b} = \frac{\partial C}{\partial a} a(1 - a).$$

En remplaçant avec $\frac{\partial C}{\partial b} = (a - y)$:

$$\frac{\partial C}{\partial a} = \frac{a - y}{a(1 - a)}$$

En intégrant cette équation :

$$C = -[y \ln a + (1 - y) \ln(1 - a)] + \text{constant}$$

L'entropie croisée est alors définie par :

$$C = -\frac{1}{n} \sum_x \sum_j [y_j \ln a_j^L + (1 - y_j) \ln(1 - a_j^L)]$$

On obtient les formules suivantes qui représentent les dérivées partielles par rapport à chaque poids synaptique et biais. Notez bien que pour des raisons de simplification on prend la couche de sortie avec un seul neurone pour ce débarasser de la somme suivant j :

$$\begin{aligned}\frac{\partial C}{\partial w_j} &= \frac{1}{n} \sum_x x_j(\sigma(z) - y) \\ \frac{\partial C}{\partial b} &= \frac{1}{n} \sum_x (\sigma(z) - y)\end{aligned}$$

Le comportement d'apprentissage de notre neurone est devenu plus performant, en effet plus que l'erreur est grande, plus l'apprentissage devient plus rapide.

Pourquoi l'entropie croisée est une fonction de coût ?

L'entropie croisée vérifie les propriétés d'une fonction de coût :

1. **La positivité** : C est toujours strictement positive.
2. **La minimalité** : C est minimale lorsque la valeur calculée s'approche de la valeur désirée, $\sigma(z) \approx y$.

4.2 Régularisation

Parmis les obstacles les plus fréquents qui empêchent l'apprentissage d'un modèle artificiel est le **surapprentissage** (overfitting).

Alors, la régularisation intervient pour éviter ce problème et augmenter la performance de notre modèle, en ajoutant un terme à la fonction de coût qui sert à empêcher ce dernier de devenir compliqués où d'avoir des paramètres à valeurs grandes.

Les méthodes de régularisation incluses sont L1, L2, dropout,.. On s'intéresse dans notre projet à la régularisation L2 seulement.

4.2.1 Régularisation L2

La régularisation se fait par l'optimisation de la nouvelle fonction de coût :

$$C = C_0 + \frac{\lambda}{2n} \cdot \sum_w \|w\|^2$$

C_0 peut être la fonction quadratique ou l'entropie croisée..

λ est un hyperparamètre ayant une valeur optimisée pour des meilleurs résultats.

4.2.2 Augmentation artificielle des données

La méthode la plus simple pour éviter le surapprentissage est d'augmenter la taille de la base de données d'entraînement, ce qui permet à notre réseau de rencontrer plusieurs cas de variations des données. Mais cela n'est pas toujours possible en pratique pour des raisons matérielles. Alors, on fait recours à l'augmentation artificielle, en appliquant une rotation sur chaque image avec une angle de 3, 6, -3 et -6 degré.

Notre base de données sera multiplier, dans ce cas, fois 5 (300 000 images).

4.3 Initialisation de Xavier

4.3.1 Pourquoi n'initialiser pas les poids par une constante ?

Supposons qu'on initialise tous les poids avec une constante α et les biais avec des zéros. Alors, les équations des algorithmes d'apprentissage vont échouer de changer les valeurs des poids appartenants à la même couche de manière différente. En effet, si on prends par exemple le i ème neurone et le j ème neurone de la même couche l , ils sont liés aux mêmes entrées de la couche précédente $l - 1$ avec les mêmes poids. Par conséquent, chacune des couches va avoir le même gradient. Et donc ce réseau ne sera jamais capable d'apprendre.

4.3.2 Problème de Vanishing et Exploding du gradient

1. Vanishing :

Expérimentalement lorsqu'on initialise les poids synaptiques par des valeurs très petites, les activations deviennent également plus petites. Alors que les gradients de la rétropropagation diminuent dans le sens inverse (de la dernière couche vers la première). Par conséquent, l'entraînement devient très lent.

La figure¹ ci-dessus démontre ce phénomène :

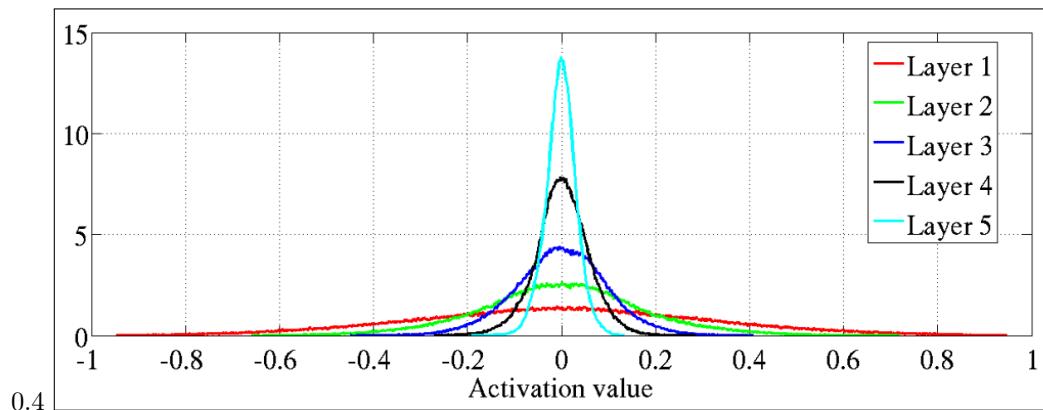


FIGURE 4.2 – Problème de Vanishing pour les valeurs d'activation

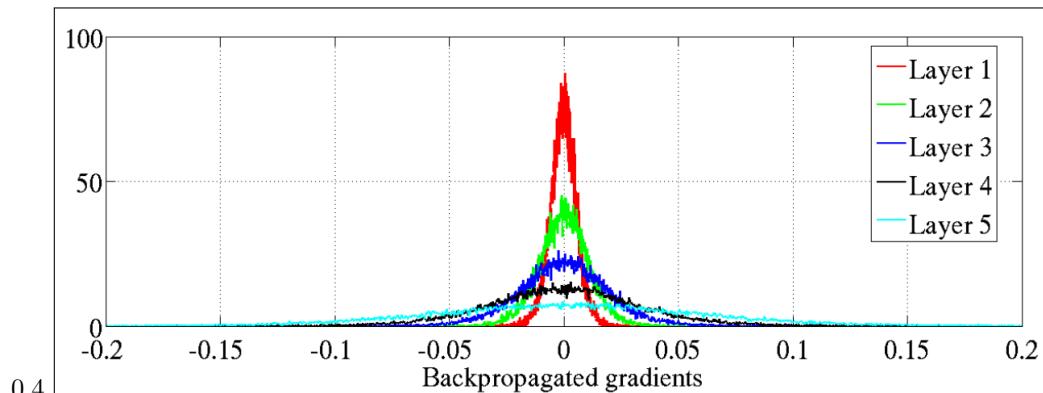


FIGURE 4.3 – Problème de Vanishing pour les valeurs de la rétro-propagation

1. prise de papier de recherche de Xavier Glorot et Yoshua Bengio

2. Exploding :

Expérimentalement, lorsqu'on initialise les poids synaptiques par des valeurs très grandes, les activations deviennent également plus grandes. Alors que les gradients de la rétro-propagation augmentent dans le sens inverse (de la dernière couche vers la première). Par conséquent, la fonction de coût fait des oscillations autour du minimum.

4.3.3 Justification mathématique

Quand on a crée notre réseau de neurones, on a initialisé les valeurs des poids synaptiques et des biais aléatoirement en utilisant `numpy.random.randn()`. Cette fonction génère des valeurs aléatoires suivant la loi normale centré réduite :

$$w, b \sim \mathcal{N}(0, 1)$$

Pour comprendre l'impact de l'initialisation des poids et biais sur z , on prend un cas spéciale qui peut être généralisé.

On s'intéresse à un neurone ayant 1000 entrées (une couche précédente de 1000 neurones) où la moitié est activée par 1 et l'autre moitié par 0. On a donc $z \sim \mathcal{N}(0, 501)$. En effet :

$$\begin{aligned} \mathbb{E}[z] &= \mathbb{E}\left[\sum_j w_j \cdot x_j + b\right] \\ &= \sum_j x_j \cdot \mathbb{E}[w_j] + \mathbb{E}[b] \\ &= 0 \end{aligned}$$

Et :

$$\begin{aligned} Var[z] &= Var\left[\sum_j w_j \cdot x_j + b\right] \\ &= \sum_j x_j^2 \cdot Var[w_j] + Var[b] \quad (\text{car } w_j \perp\!\!\!\perp b \text{ et } w_j \perp\!\!\!\perp w_k \quad \forall j \neq k) \\ &= 500 \times 1 + 1 \\ &= 501 \end{aligned}$$

On a alors $z \sim \mathcal{N}(0, 501)$, ce qui signifie que $|z|$ peut prendre des valeurs grandes ou petites avec une probabilité un peu adéquate. Autrement dit, z a une distribution Gaussienne plus large.

Alors, pour éviter la saturation de $\sigma(z)$, on va essayer de rendre le graphe ci-dessus plus étroite, c-à-d z suivra une loi normale à écart-type plus petit. Et comme on ne contrôle que l'initialisation des poids et des biais, et puisque ces derniers ont un impact négligeable sur z , alors on va changer seulement la loi des w_j .

Pour ce faire, on prend la variance des w_j égale à $1/n$ où n est le nombre des entrées (à justifier prochainement).

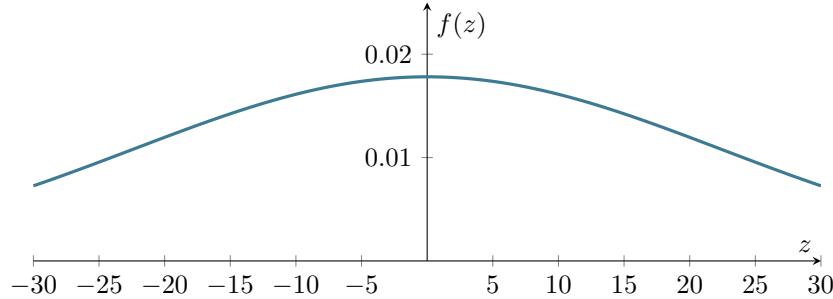


FIGURE 4.4 – Distribution de gaussse

Pour $n = 1000$, on obtient $z \sim \mathcal{N}(0, 3/2)$:

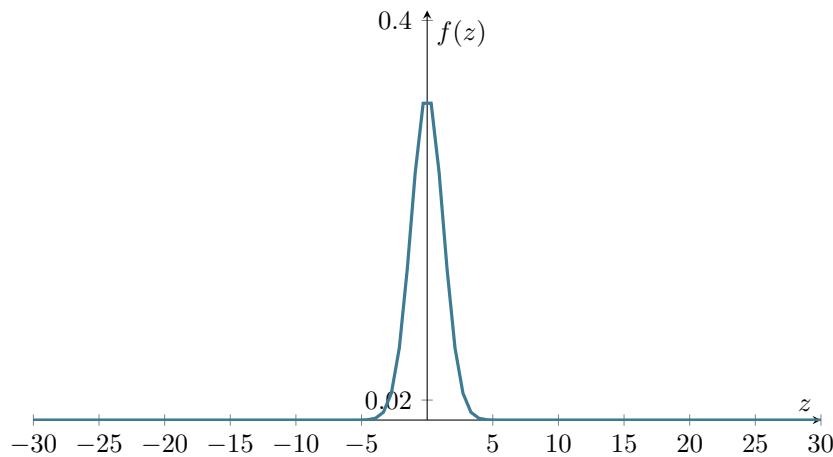


FIGURE 4.5 – Distribution de gauss

Mais pourquoi prendre une variance exactement de $1/n$?

Cette justification mathématique est basé sur la fonction tanH mais reste pratiquement valable pour la fonction sigmoïde et donne des bonnes résultats. On cherche à résoudre le problème de vanishing ou exploding du gradient en garantissant les deux propriétés :

1. L'espérance des activations doit être nulle : $\mathbb{E}[a^{l-1}] = \mathbb{E}[a^l] = 0$.
2. La variance des activations doit rester la même pour chaque couche : $\text{Var}[a^{l-1}] = \text{Var}[a^l]$.

Sous ces deux propriétés, on garantit que les gradients ne sera jamais multiplié par des valeurs ni très larges ni très petites.

Pour chaque couche l on a :

$$z_k^l = \sum_{j=1}^{n^{l-1}} w_{kj}^l a^{l-1} + b_k^l$$

$$a^l = \text{tanH}(z_k^l)$$

Notre but est de chercher une relation entre $Var[a^{l-1}]$ et $Var[a^l]$, pour savoir comment initialiser les poids de telle sorte que : $Var[a^{l-1}] = Var[a^l]$.

On suppose que les activations de notre réseau sont normalisées. On sait qu'au voisinage de 0 $\tan H[z_k^l] \approx z_k^l$, alors on a :

$$Var[a_k^l] = Var[z_k^l]$$

Pour simplifier, on prend $b_k^l = 0$ on a donc :

$$Var[a_k^l] = Var[z_k^l] = Var \left[\sum_{j=1}^{n^{l-1}} w_{kj}^l a^{l-1} \right]$$

Le raisonnement qui vient après nécessite les trois suppositions :

1. Les poids synaptiques sont indépendants et identiquement distribués (iid) ;
2. Les activations sont indépendantes et identiquement distribuées (iid) ;
3. Les poids et les activations sont mutuellement indépendants.

alors :

$$Var[a_k^l] = Var[z_k^l] = Var \left[\sum_{j=1}^{n^{l-1}} w_{kj}^l a^{l-1} \right] = \sum_{j=1}^{n^{l-1}} Var[w_{kj}^l a^{l-1}]$$

On sait que : $Var[XY] = \mathbb{E}[X]^2 Var[Y] + Var[X]\mathbb{E}[Y]^2 + Var[X]Var[Y]$

Et puisque $\mathbb{E}[w_{kj}] = \mathbb{E}[a^{l-1}] = 0$, donc :

$$\begin{aligned} Var[a_k^l] &= Var[z_k^l] = Var \left[\sum_{j=1}^{n^{l-1}} w_{kj}^l a_j^{l-1} \right] = \sum_{j=1}^{n^{l-1}} Var[w_{kj}^l a_j^{l-1}] \\ &= \sum_{j=1}^{n^{l-1}} Var[w_{kj}^l] Var[a_j^{l-1}] \\ &= \sum_{j=1}^{n^{l-1}} Var[W^l] Var[a^{l-1}] \text{ Puisque les variables sont identiquement distribuées} \\ &= n^{l-1} Var[W^l] Var[a^{l-1}] \end{aligned}$$

Il suffit donc de prendre : $n^{l-1} Var[W^l] = 1 \iff Var[W^l] = \frac{1}{n^{l-1}}$

N.B : $Var[W^l]$ et $Var[a^{l-1}]$ sont juste des notations pour désigner des constantes :

$$\begin{aligned} Var[w_{kj}^l] &= Var[w_{11}^l] = Var[w_{12}^l] = \dots = Var[W^l] \\ Var[a_j^{l-1}] &= Var[a_1^{l-1}] = Var[a_2^{l-1}] = \dots = Var[a^{l-1}] \end{aligned}$$

Cette initialisation de Xavier sur \tanh donne les bons résultats suivant :

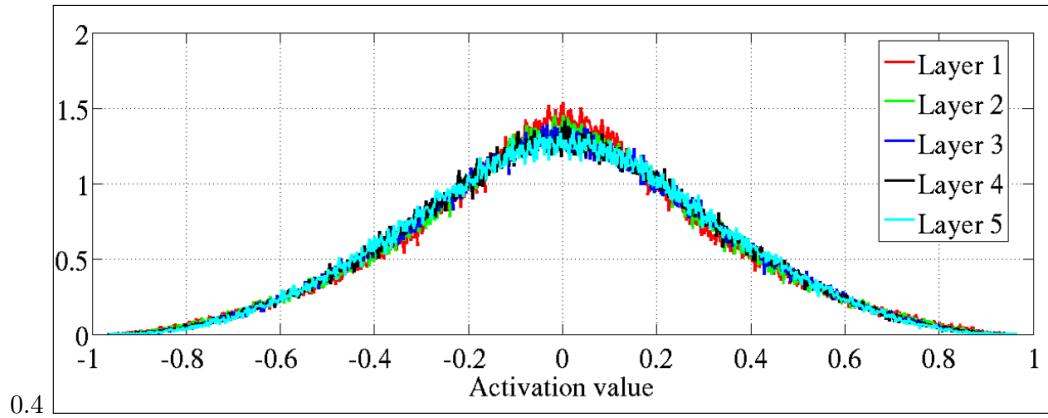


FIGURE 4.6 – Problème de Vanishing est fixé 1

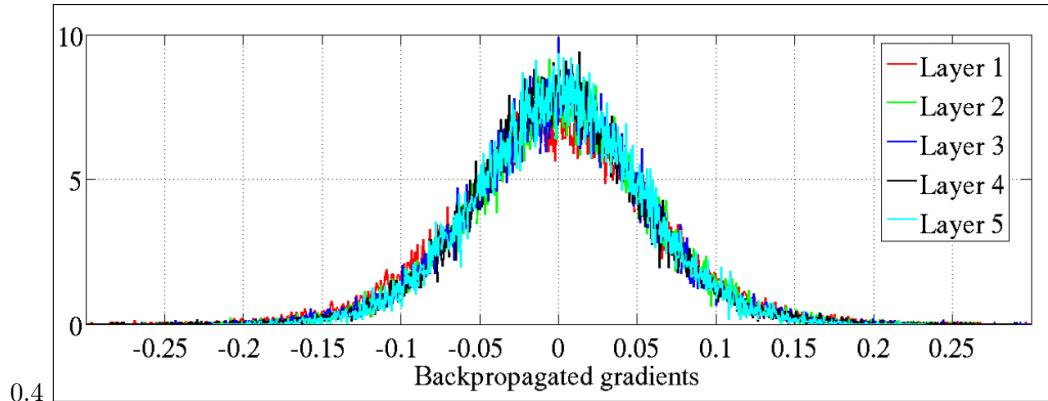


FIGURE 4.7 – Problème de Vanishing est fixé 2

Même que cette initialisation est dédiée originale pour la fonction \tanh , Mais elle reste pratiquement valable pour la fonction σ .

4.4 Implémentation

Après avoir appliquer tous ces méthodes mentionnées dans ce chapitre, on obtient les résultats suivantes :

```
Epoch 15 training complete
Cost on training data: 0.11777428926356363
Accuracy on training data: 297067 / 300000
Cost on evaluation data: 1.3429058500305766
Accuracy on evaluation data: 9790 / 10000
```

FIGURE 4.8 – dernier apprentissage

Alors comme montre la figure ci-dessus, la précision a augmenté jusqu'à 97.90% sur les données d'évaluation à l'époche 15.

Chapitre 5

INTERFACE GRAPHIQUE

5.1 Evaluation et validation

Pour évaluer et valider notre modèle artificiel, nous avons crée une interface graphique en exploitant de la bibliothèque python **Streamlit** qui est conçu pour être utilisé localement pour le développement et le prototypage du projet.

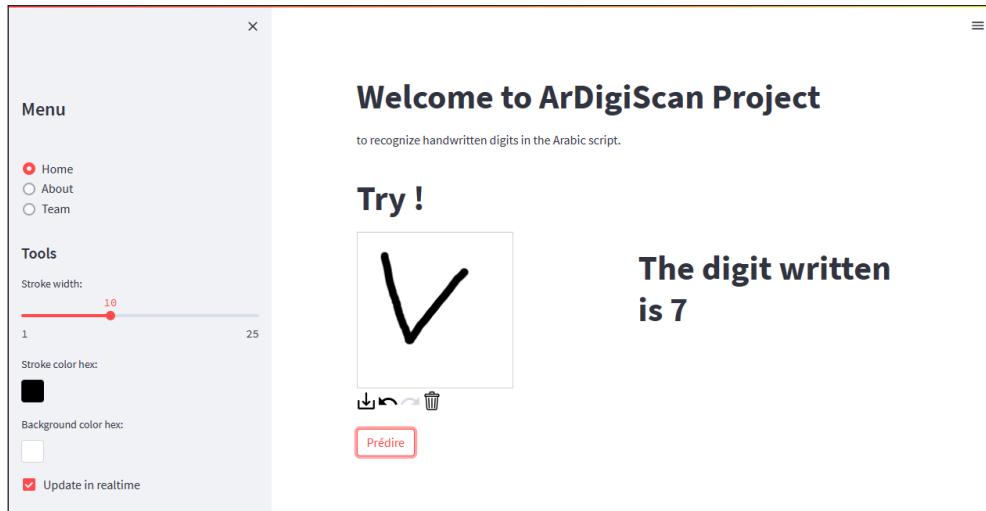


FIGURE 5.1 – premier test

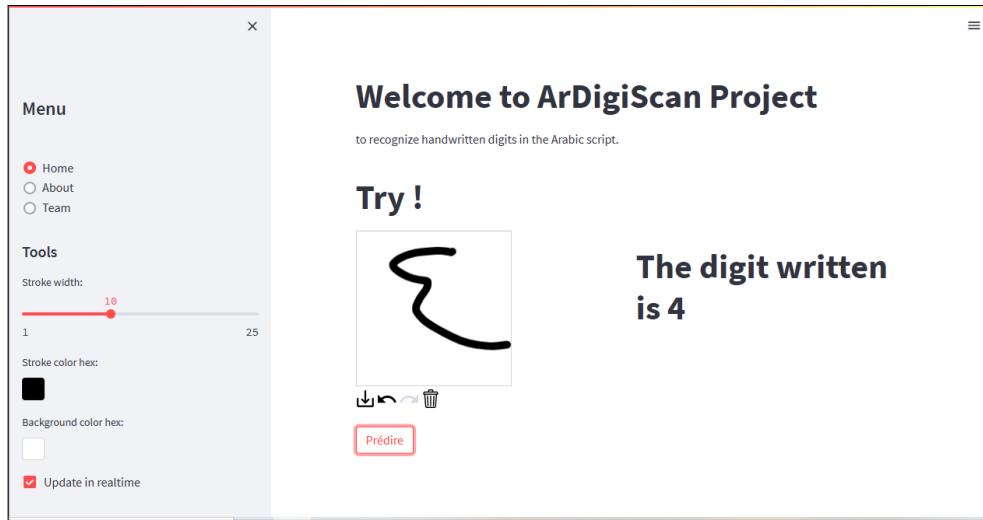


FIGURE 5.2 – deuxième test

5.2 Lancement du site

Après avoir validé notre modèle artificiel, nous avons crée un site web pour permettre à tous les utilisateurs de l'essayer et de le tester.

Ce site a été élaboré en utilisant les langages *html*, *css* et *JavaScript* et le framework *FastApi* pour intégrer notre code python avec notre interface web.

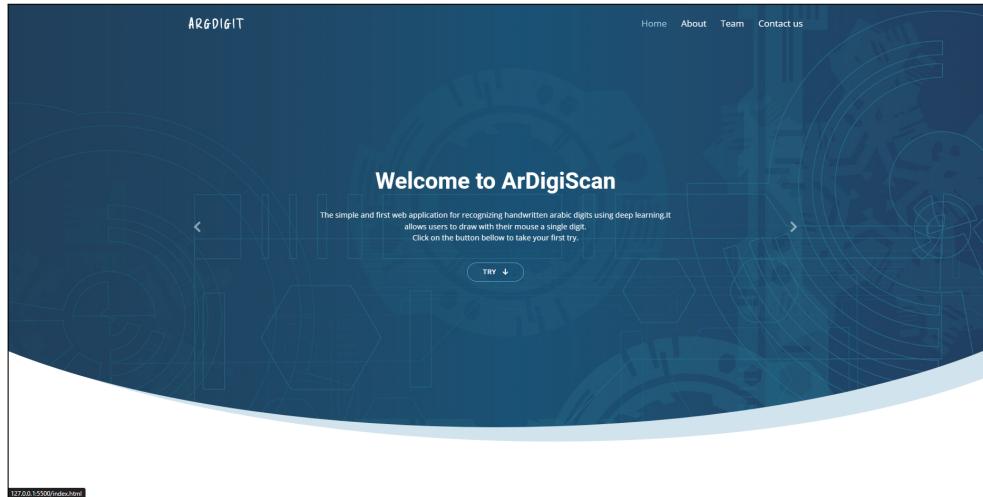


FIGURE 5.3 – site-page 1

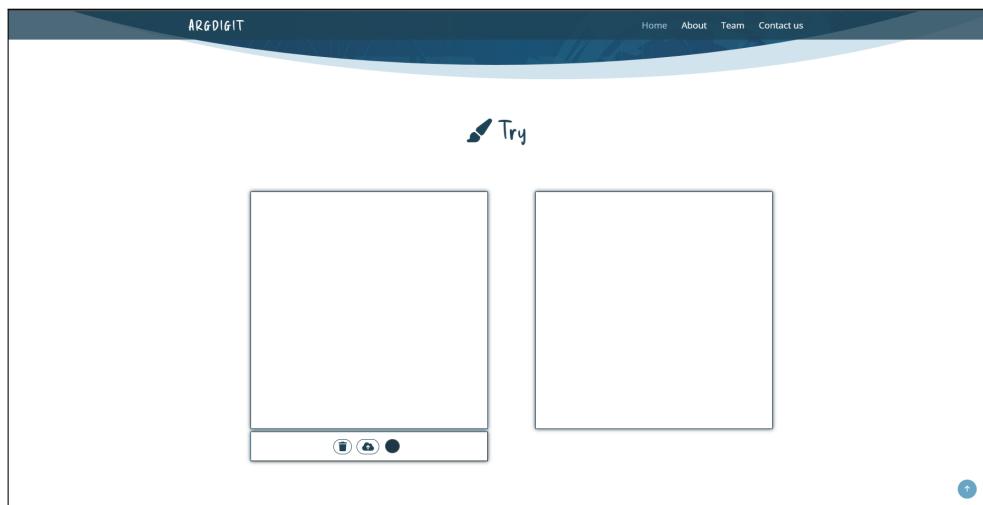


FIGURE 5.4 – site-page 2

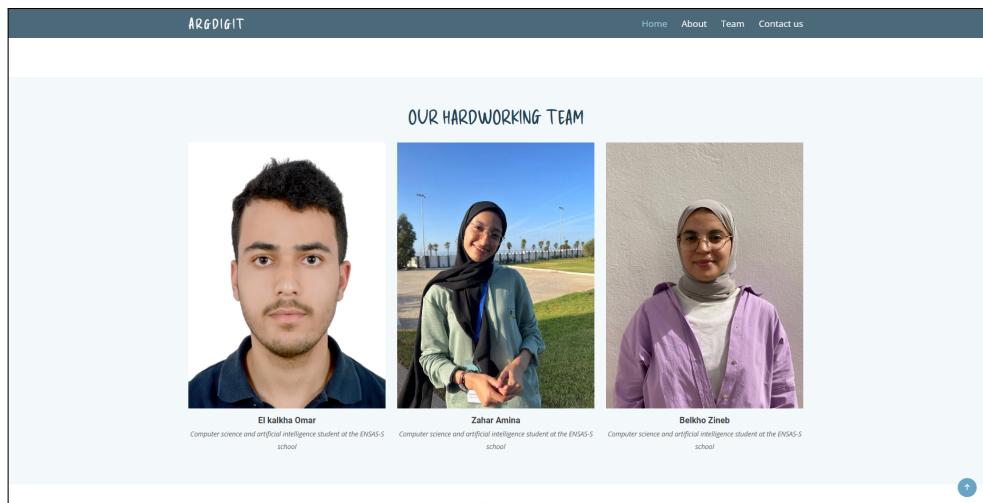


FIGURE 5.5 – site-page 4

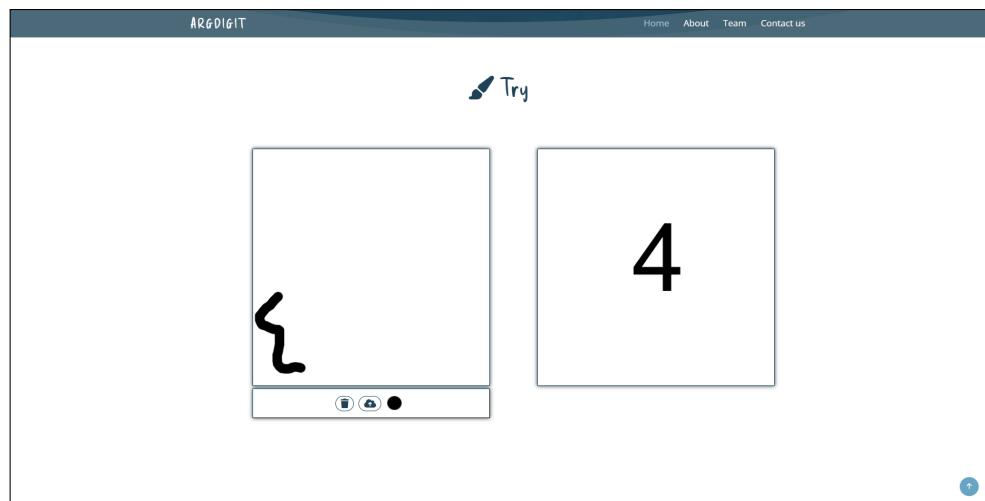


FIGURE 5.6 – site-test 1

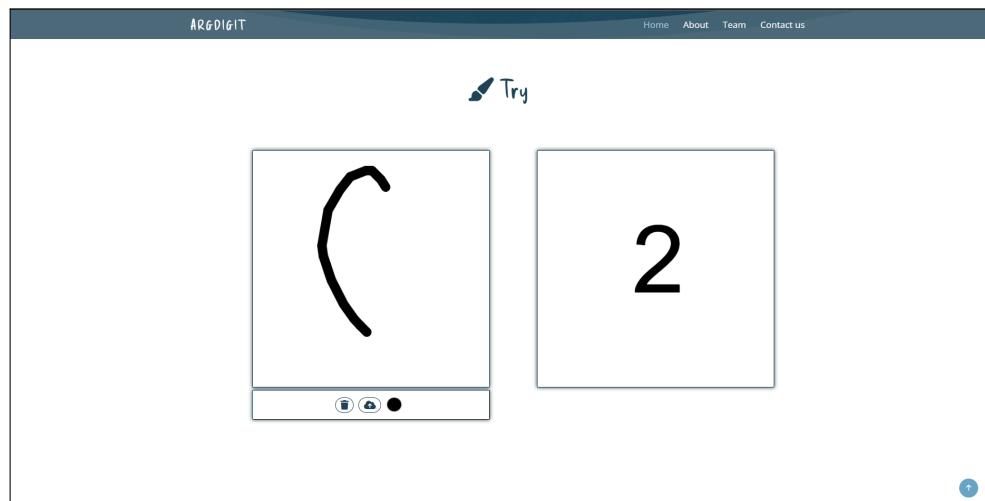


FIGURE 5.7 – site-test 2

CONCLUSION

En conclusion, ce projet de la reconnaissance des chiffres manuscrits nous démontre le potentiel de l'intelligence artificielle dans le domaine de la vision de l'ordinateur. Nous avons réussi à créer un modèle artificiel capable de reconnaître les chiffres arabes-orientaux manuscrits avec précision de 97.90% grâce à l'utilisation des algorithmes avancés. Cependant, il convient de noter que ce projet reste limité puisqu'il n'est pas capable de traiter une chaîne de chiffres(nombre) à cause du défi de segmentation qu'on souhaite adresser prochainement, et inclure les alphabets arabes, afin de digitaliser les manuscrits arabes.
Finalement, ce projet nous a permis de comprendre les fondements théoriques et pratiques de l'apprentissage en général, et de la reconnaissance des chiffres manuscrits en particulier.

Bibliographie

- [1] Sherif Abdelazeem and Ezzat El-Sherif. The arabic handwritten digits databases adbase madbase, 2016.
- [2] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feed-forward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256. JMLR Workshop and Conference Proceedings, 2010.
- [3] Shubham.jain Jain. An overview of regularization techniques in deep learning (with python code), Published On April 19, 2018.
- [4] Katanforoosh and Kunin. "initializing neural networks", deeplearning.ai, 2019.
- [5] Michael A. Nielsen. "neural networks and deep learning", nielsenneural, 2019.
- [6] Stuart Russell and Peter Norvig. *Artificial Intelligence : A Modern Approach*. Prentice Hall, 3 edition, 2010.