

T.I.P.E. - Réseau de neurones et analyse d'images sportives

Thomas KUMMEL

5 juin 2024

Table des matières

1	Introduction	5
1.1	Projet	5
1.2	Redéfinition d'intelligence artificielle	5
2	Théorie des réseaux de neurones	7
2.1	Définitions	7
2.1.1	Notation matricielle du réseau	8
2.2	Prédiction d'un résultat	9
2.2.1	Propagation des informations	9
2.2.2	Erreur d'un réseau	10
2.3	Entraînement d'un réseau	10
2.3.1	Préambule	10
2.3.2	Minimiser le coût	10
2.3.3	Algorithme de descente de gradient	10
2.3.4	Preuve	12
2.4	Initialisation d'un réseau de neurones denses	12
2.5	Code	13
2.6	MNIST	16
2.6.1	Présentation	16
2.6.2	Optimisation	16
2.6.3	Code	17
2.6.4	Résultats	17
3	Projet	19
3.1	Identification du joueur	19
3.1.1	Convolution d'images	19
3.1.2	Code	21
3.1.3	Première étude : filtres d'images	25
3.1.4	Deuxième étape : CNN à 1 niveau	27
3.1.5	Troisième étape : CNN à plusieurs niveaux	28
3.1.6	Quatrième étape : algorithme des moyennes	29
3.2	Identification du sport	30
4	Conclusion	31

Chapitre 1

Introduction

1.1 Projet

Lorsque l'on regarde un match sportif, on peut se poser la question de comment peut-on analyser les images enregistrées par les caméras présentes.

Notre projet a pour but d'évaluer la possibilité de l'identification des joueurs en utilisant leur numéro de dossard ainsi que l'identification du type de sport avec la balle utilisé.

1.2 Redéfinition d'intelligence artificielle

Ce qui distingue une simple intelligence artificielle d'un réseau de neurones (ou deep-learning) réside dans son principe de fonctionnement. On pourrait très bien programmer une intelligence artificielle comme étant une succession d'étapes conditionnelles, ce qui est le cas pour les assistants vocaux, et dans ce cas, il ne s'agit en aucun cas d'une notion d'intelligence, mais plutôt une capacité à différencier en cas d'un autre.

Le réseau de neurones, quand ta vie est beaucoup plus intelligent, son principe de fonctionnement, basé sur une allégorie du cerveau humain et de sa multitude de neurones présents, lui confère un caractère plus intelligent, qu'une simple intelligence artificielle basée sur des conditions. Le réseau de neurones fonctionne sur un principe d'entrée et de sortie et ne nécessite aucune condition. Grâce à un apprentissage supervisé ou pas, il sera capable après un certain nombre de répétitions de pouvoir répondre aux problèmes demander.

Intelligence artificielle : ensemble de théories et de techniques mises en œuvre en vue de réaliser des machines capables de simuler l'intelligence humaine.

D'après *larousse.fr*.

Chapitre 2

Théorie des réseaux de neurones

On souhaite étudier la théorie des réseaux de neurones. Pour se faire, nous nous concentrerons sur les réseaux de neurones convolutifs avec propagation de l'information

2.1 Définitions

Avant toute chose, nous devons définir plusieurs notions nécessaires à l'étude des réseaux de neurones.

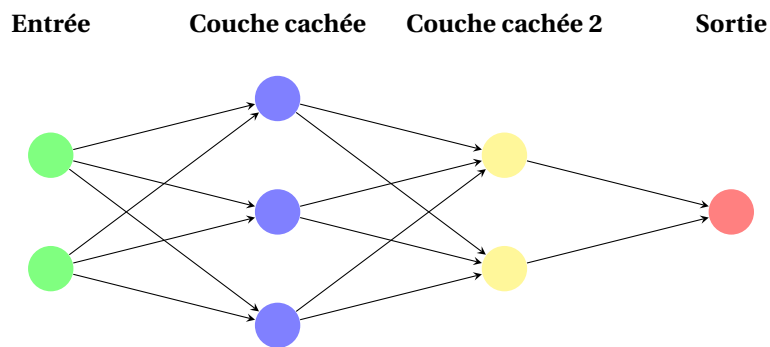


FIGURE 2.1 – Perceptron multicouches

➔ **Neurone** : un neurone, similaire dans sa conception à son homologue compris dans le cerveau humain, est un élément qui prend une valeur entrée décimale puis d'y appliquer

La valeur $v_i^{(k)}$ du neurone i de la couche k est comprise, généralement, entre 0 - état inactif - et 1 - neurone actif¹.

➔ **Couche** : une couche est une rangée de $(c_i)_{1 \leq i \leq n}$ neurones interconnectée aux couches précédentes. Un réseau de neurones est composé de 3 types de couches de neurones : une couche d'entrée (*input layer*), des couches cachées (*hidden layer*), et une couche de sortie (*output layer*) qui donne la prédiction du réseau.

1. Couche d'entrée : reçoit les données brutes en entrée. Par exemple, dans le cas de la reconnaissance d'images, cela pourrait être les valeurs de pixels brutes.
2. Couches cachées : effectuent des transformations sur les entrées en utilisant des poids et des biais. Ce sont des couches internes entre la couche d'entrée et la couche de sortie.
On ne sait pas combien de neurones et combien de couches internes sont nécessaires pour avoir un fonctionnement optimal du réseau de neurones. Uniquement des expériences permettent d'avoir une meilleure. Cela reste un des grands paradigmes des réseaux de neurones.
3. Couche de sortie : renvoie les valeurs de sortie. La couche de sortie contient c_n neurones correspondant aux différentes actions ou réponses possibles par le réseau.

1. Il s'agit d'approximations et non de valeurs exactes.

Nom	Expression mathématiques	Valeurs de sorties	Commentaires
Sigmoid	$f_a : x \mapsto \frac{1}{1 + \exp(-x)}$	$]0; 1[$	
Tanh	$f_a : x \mapsto \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)}$	$] -1; 1[$	
ReLU (Rectified Linear Unit)	$f_a : x \mapsto \max(0, x)$	$[0; +\infty[$	Elle est très utilisée dans les réseaux de neurones profonds.
Softmax	$f_a : x \mapsto$	$]0; 1[$	

TABLE 2.1 – Exemples de fonctions d'activation

- ➔ **Réseau** : un réseau de neurones dense contient n couches de neurones qui sont toutes interconnectées entre elles afin de permettre la transmission des informations
- ➔ **Poids** : Les poids sont les facteurs qui amplifient ou atténuent les connexions et permettent donc de pondérer les informations sur entre les couches. Chaque connexion possède son propre poids. On notera $w_{i,j}^{(k)}$ le poids du neurone i au neurone j de la couche k .
- ➔ **Biais** : le biais permet de décaler la fonction d'activation. Il s'agit d'une valeur caractéristique à chaque neurone. On notera $b_i^{(k)}$ le biais du neurone i de la couche k .
- ➔ **Fonctions d'activation** : Les fonctions d'activation introduisent des non-linéarités dans le réseau, ce qui lui permet d'apprendre des relations complexes.

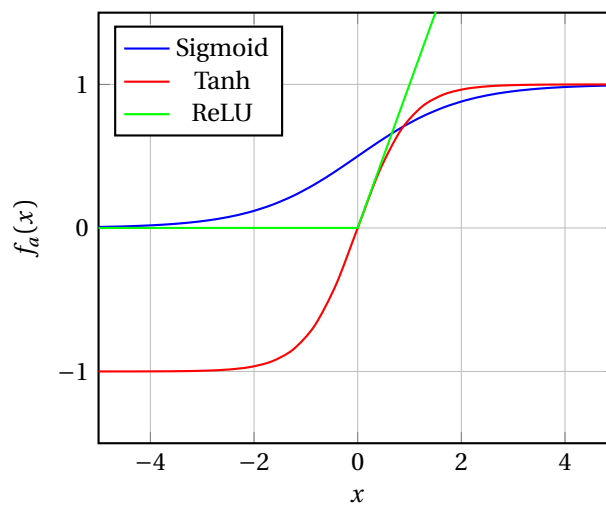


FIGURE 2.2 – Fonctions d'activations

Mathématiquement, un neurone est plus qu'un simple nombre, il s'agit d'une fonction mathématiques. Ainsi, la sortie $v_i^{(k)}$ d'un neurone est calculée comme suit :

$$v_i^{(k)} = f \left(\sum_{j=1}^{n_{k-1}} w_{j,i}^{(k)} \cdot v_j^{(k-1)} + b_i^{(k)} \right)$$

2.1.1 Notation matricielle du réseau

Après avoir exprimé les notations pour chaque définition, on peut définir les éléments d'un point de vue de mathématiques. Ainsi, on représente pour une couche k les éléments précédents de la manière suivante

Valeurs des neurones

$$V_k = \begin{pmatrix} v_1^{(k)} \\ v_2^{(k)} \\ \vdots \\ v_{c_k-1}^{(k)} \\ v_{c_k}^{(k)} \end{pmatrix} \quad (2.1)$$

Poids

$$W_k = \begin{pmatrix} w_{0,0}^{(k)} & w_{1,0}^{(k)} & \dots & w_{c_{k-1}-1,0}^{(k)} & w_{c_{k-1},0}^{(k)} \\ w_{0,1}^{(k)} & w_{1,1}^{(k)} & \dots & w_{c_{k-1}-1,1}^{(k)} & w_{c_{k-1},1}^{(k)} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ w_{0,c_{k-1}}^{(k)} & w_{1,c_{k-1}}^{(k)} & \dots & w_{c_{k-1}-1,c_{k-1}}^{(k)} & w_{c_{k-1},c_{k-1}}^{(k)} \\ w_{0,c_k}^{(k)} & w_{1,c_k}^{(k)} & \dots & w_{c_{k-1}-1,c_k}^{(k)} & w_{c_{k-1},c_k}^{(k)} \end{pmatrix} \quad (2.2)$$

Biais

$$B_k = \begin{pmatrix} b_1^{(k)} \\ b_2^{(k)} \\ \vdots \\ b_{c_k-1}^{(k)} \\ b_{c_k}^{(k)} \end{pmatrix} \quad (2.3)$$

2.2 Prédiction d'un résultat**2.2.1 Propagation des informations**

Dans un réseau de neurones denses représentée sur la figure 2.3, où tous les neurones sont interconnectés avec tous ceux de la couche précédente

FIGURE 2.3 – Réseau de neurones denses

Chaque information se propage de la manière suivante entre les couches.

$$x_i^{(k)} = f_a \left(\sum_{j=1}^{c_{k-1}} w_{j,i}^{(k)} + b_i^{(k)} \right) \quad (2.4)$$

On donne ainsi la formule suivante pour la propagation des informations dans un réseau de neurones :

$$V_k = f_a (W_k \cdot V_{k-1} + B_k) \quad (2.5)$$

Un réseau de neurones denses ne renvoi pas de réponses comme *chien, chat, oiseau, mammifère, ...* mais se contente de renvoyer la réponse la plus probable. On a pu voir la configuration d'un réseau de neurones que ce dernier comporte sur sa couche de sortie un ensemble de neurones qui représentent chacun UNE information possible (confère figure 2.1). La prédiction d'une information nécessite d'avoir correctement défini quel neurone correspond à quelle prédiction et de réaliser une lecture correcte de la réponse en lisant la valeur la plus élevée.

2.2.2 Erreur d'un réseau

Dans un premier temps, il est nécessaire de calculer l'erreur du réseau de neurones pour cela on utilise une fonction de coût afin de déterminer la précision de chaque neurones.

Il existe plusieurs fonctions de coûts pour un réseau de neurones, la fonction la plus simple est la somme des différences au carré. On rajoutera le facteur de $\frac{1}{2}$ pour simplifier les calculs de la propagation.

$$C = \sum_{i=1}^{c_n} \frac{1}{2} (x_i^{(n)} - y_i)^2 \quad (2.6)$$

Plus la valeur retournée par cette fonction est élevée, et plus le réseau se trompe, il s'agit donc de donner à l'ordinateur une information sur la correction de cet entraînement. À l'inverse si la valeur envoyée par cette fonction est proche de zéro, on peut en conclure que l'ensemble des neurones auront presque tous renvoyer la valeur attendue.

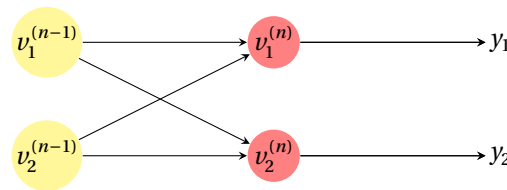


FIGURE 2.4 – Erreur du réseau

2.3 Entraînement d'un réseau

2.3.1 Préambule

Une question reste centrale :

Comment améliore-t-on le rendement d'un réseau de neurones ?

Afin d'améliorer un réseau de neurones, on cherche à minimiser le résultat de la fonction de coût.

2.3.2 Minimiser le coût

Le principe du gradient permet d'obtenir la direction dans laquelle il faut modifier les valeurs afin d'améliorer le résultat.

Chaque information du gradient informe sur deux points : la direction et l'impact que cela aura sur les résultats.

2.3.3 Algorithme de descente de gradient

Allégorie de la montagne

Partons quelques instants à la montagne! Si je vous laisse à un endroit que vous ne connaissez pas à la montagne qu'allez-vous faire? Monter au sommet de la montagne? ou bien descendre? Je vais vous donner la bonne réponse ...

Je vous pose à un endroit aléatoire de la montagne, et vous allez suivre ces étapes.

1. Faites un tour sur vous même (360 degrés).
2. Trouvez la direction où la pente est la plus abrupte.
3. Avancez dans cette direction pendant environ 300 mètres.
4. Recommencez les étapes 1 à 3, jusqu'à être arrivé en bas.
5. Félicitations, vous êtes arrivés en bas.

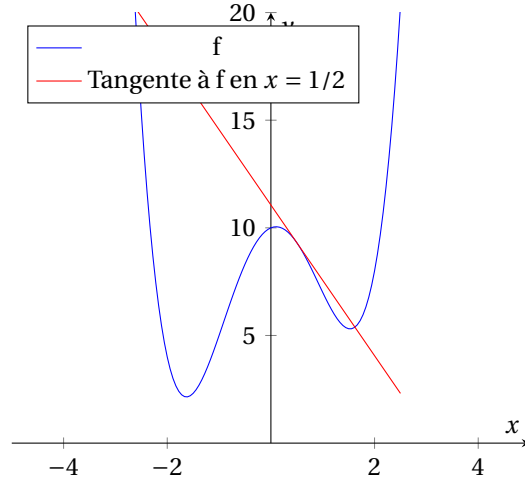


FIGURE 2.5 – Fonction d'exemple

Supposons que nous avons un réseau de neurones denses définie comme précédemment. On définit le **taux d'apprentissage** α du réseau. Qui est une variable comprise entre 0 et 1.

Un réseau de neurones comporte, comme informations que l'on peut faire évoluer, ses poids et ses biais. Conséquemment, la rétropropagation consiste à calculer comment l'erreur change en fonction de chaque poids dans le réseau (c'est-à-dire, les gradients de la fonction de coût par rapport aux poids) et à utiliser ces informations pour mettre à jour les poids et les biais afin de minimiser l'erreur et d'obtenir les meilleurs résultats.

Mise à jour du poids

Commençons par la dernière couche, on utilise la règle de la chaîne pour calculer le gradient de la fonction de coût par rapport à un poids

$$\frac{\partial C}{\partial w_{i,j}^{(n)}} = \frac{\partial C}{\partial x_j^{(n)}} \cdot \frac{\partial x_j^{(n)}}{\partial w_{i,j}^{(n)}} \quad (2.7)$$

1. $\frac{\partial C}{\partial x_j^{(n)}}$, est l'impact de la sortie X_n sur l'erreur. Pour la formule de coût donnée en (2.6), cela vaut :

$$\frac{\partial C}{\partial x_j^{(n)}} = \sum_{i=1}^{c_n} x_i^{(n)} - y_i$$

2. $\frac{\partial x_j^{(n)}}{\partial w_{i,j}^{(n)}}$, est l'impact du poids $w_{i,j}^{(n)}$ sur la valeur de sortie $x_j^{(n)}$. D'après (2.4), on peut écrire $\frac{\partial x_j^{(n)}}{\partial w_{i,j}^{(n)}}$ comme :

$$\frac{\partial x_j^{(n)}}{\partial w_{i,j}^{(n)}} = f'_a \left(\sum_{p=1}^{c_{k-1}} w_{p,j}^{(n)} \cdot x_i^{(n-1)} + b_j^{(n)} \right) \cdot x_i^{(n-1)}$$

Par conséquent, on a :

$$\frac{\partial C}{\partial w_{i,j}^{(n)}} = \frac{\partial C}{\partial x_j^{(n)}} \cdot f'_a \left(\sum_i w_{ij}^{(n)} \cdot x_i^{(n-1)} + b_j^{(n)} \right) \cdot x_i^{(n-1)} \quad (2.8)$$

On peut maintenant appliquer une correction à tous les biais du réseau en utilisant le taux d'apprentissage α fixé à la création du réseau.

$$w_{i,j}^{(n)} = w_{i,j}^{(n)} - \alpha \cdot \frac{\partial C}{\partial w_{i,j}^{(n)}} \quad (2.9)$$

Mise à jour du biais

Maintenant, passons à $\frac{\partial C}{\partial b_i^{(n)}} :$

$$\frac{\partial C}{\partial b_i^{(n)}} = \frac{\partial C}{\partial x_i^{(n)}} \cdot \frac{\partial x_i^{(n)}}{\partial b_i^{(n)}}$$

On reprendra les résultats du paragraphe précédent

Donc,

$$\frac{\partial C}{\partial b_i^{(n)}} = \frac{\partial C}{\partial x_i^{(n)}} \cdot f'_a \left(\sum_j w_{ij}^{(n)} \cdot x_j^{(n-1)} + b_j^{(n)} \right)$$

Enfin, on met à jour tous les biais en suivant cette formule :

$$b_i^{(n)} = b_i^{(n)} - \alpha \frac{\partial C}{\partial b_i^{(n)}} \quad (2.10)$$

Nous répétons ce processus pour tous les poids et tous les biais dans le réseau. Cela constitue une itération de l'entraînement. Nous répétons généralement ce processus pour plusieurs itérations jusqu'à ce que l'erreur soit suffisamment faible.

2.3.4 Preuve

Lorsque nous utilisons la descente de gradient pour minimiser une fonction de coût $L(\mathbf{w})$, notre objectif est de modifier le vecteur de paramètres \mathbf{w} de manière à réduire la valeur de $L(\mathbf{w})$ autant que possible.

L'inégalité de Cauchy-Schwarz, dans le contexte de la descente de gradient, nous indique à quel point la fonction de coût peut changer lorsque nous nous déplaçons dans une certaine direction. Plus précisément, elle indique que le changement dans la fonction de coût, lorsque nous nous déplaçons dans la direction du vecteur \mathbf{d} , est donné par le produit scalaire $\mathbf{g} \cdot \mathbf{d}$, où \mathbf{g} est le gradient de $L(\mathbf{w})$.

Si nous voulons que la fonction de coût diminue le plus rapidement possible, nous devons choisir \mathbf{d} de manière à minimiser $\mathbf{g} \cdot \mathbf{d}$.

L'inégalité de Cauchy-Schwarz nous indique que

$$\mathbf{g} \cdot \mathbf{d} \leq \|\mathbf{g}\| \|\mathbf{d}\| \cos(\theta),$$

où θ est l'angle entre \mathbf{g} et \mathbf{d} .

Le terme $\cos(\theta)$ est minimisé lorsque $\theta = 180^\circ$ (c'est-à-dire lorsque \mathbf{d} pointe dans la direction opposée à \mathbf{g}), car $\cos(180^\circ) = -1$.

Cela signifie que le choix de \mathbf{d} qui minimise $\mathbf{g} \cdot \mathbf{d}$ (et donc qui réduit le plus rapidement la fonction de coût) est lorsque \mathbf{d} pointe dans la direction opposée à \mathbf{g} . Autrement dit, la direction qui réduit le plus rapidement la fonction de coût est l'opposée de la direction du gradient.

C'est pourquoi, en descente de gradient, nous mettons à jour \mathbf{w} en le déplaçant dans la direction opposée au gradient, c'est-à-dire que nous faisons la mise à jour

$$\mathbf{w}_{\text{new}} = \mathbf{w}_{\text{old}} - \eta \mathbf{g},$$

où η est un scalaire positif appelé taux d'apprentissage.

2.4 Initialisation d'un réseau de neurones denses

La mise en place d'un réseau de neurones denses est complexe et aléatoire.

1. Tout d'abord, on **initialise les poids et les biais**. Le plus simple est de choisir ceux-ci nuls. Mais on choisira ici une distribution aléatoire suivant une loi normale (distribution gaussienne).
2. Pour chaque donnée du paquet d'entraînement, on effectue les actions suivantes :

- (a) On utilise le principe de propagation, pour obtenir la **prédiction** du réseau de neurones.
 - (b) On calcul le **coût de l'entraînement** avec la fonction de coût préalablement choisie.
 - (c) On applique le principe de descente du gradient, pour réaliser la **mise du jour des poids et des biais**.
3. On peut maintenant obtenir les prédictions finales du réseau de neurones.

2.5 Code

Annexe

```

1  import matplotlib.pyplot as plt
2  import numpy as np
3
4
5  def lineaire(mode, matrice):
6      if mode == "derivee":
7          return np.zeros(matrice.shape) + 1
8      return matrice
9
10
11 def sigmoid(mode, matrice):
12     if mode == "derivee":
13         return sigmoid("", matrice) * (1 - sigmoid("", matrice))
14     return 1 / (1 + np.exp(-matrice))
15
16
17 def ReLu(mode, matrice):
18     if mode == "derivee":
19         return matrice > 0
20     return np.maximum(matrice, 0)
21
22
23 def tanh(mode, matrice):
24     if mode == "derivee":
25         return 1 - tanh("", matrice) ** 2
26     return np.tanh(matrice)
27
28
29 def softmax(mode, matrice):
30     if mode == "derivee":
31         print("Erreur : la dérivée de la fonction softmax n'est pas définie")
32         return None
33     e = np.exp(matrice - np.max(matrice))
34     return e / sum(e)
35
36
37 def ys_to_matrice(y, nb_classe):
38     return np.array([[1 if i == j else 0 for i in range(nb_classe)] for j in y])
39
40
41 def comptage_resultats(y_pratiques, y_theoriques):
42     compteur = 0
43     for i in range(y_pratiques.size):
44         if y_pratiques[i] == y_theoriques[i]:
45             compteur += 1
46     return compteur
47
48
49 def cout(x, y):
50     return 0.5 * np.sum((x-y)**2)
51

```

```

52
53 def image(image):
54     plt.imshow(image, cmap='gray')
55     plt.show()
56
57
58 def image_resultat(image, valeurs, legendes):
59     maxi = np.argmax(valeurs)
60     couleurs = ["red" if i == maxi else "blue" for i in range(len(valeurs))]
61
62     plt.figure()
63     plt.subplot(211)
64     plt.imshow(image)
65
66     plt.subplot(212)
67     plt.bar(legendes, valeurs, color=couleurs)
68
69     plt.show()
70
71
72 def images_comparaison(image_originale, image_retouchee):
73     plt.figure()
74     plt.subplot(211)
75     plt.imshow(image_originale)
76
77     plt.subplot(212)
78     plt.imshow(image_retouchee)
79
80     plt.show()
81
82
83 def affichage(x, y, legendes, titre="", x_label="", y_label=""):
84     for i in range(len(y)):
85         plt.plot(x, y[i], label=legendes[i])
86     plt.title(titre)
87     plt.xlabel(x_label)
88     plt.ylabel(y_label)
89     plt.legend()
90     plt.show()

```

Réseau

```

1 import numpy as np
2 from annexe import *
3
4
5 class Réseau:
6     def __init__(self, dimensions, taux_apprentissage, fonctions_activation_noms):
7         print("\nDEFINITION DU RESEAU VECOTIRISE")
8         assert len(fonctions_activation_noms) == len(dimensions) - 1
9
10        self.informations_reseau = dimensions
11        self.taux_apprentissage = taux_apprentissage
12        self.fonctions_activation = fonctions_activation_noms
13
14        self.A, self.V = [], []
15        self.W, self.B = self.definition_reseau()
16
17        self.nb_classes = self.informations_reseau[-1]
18        self.nb_entrainements, self.nb_tests = 0, 0
19        self.taux_reussite = []
20
21        print("Forme du réseau : ", self.informations_reseau)

```

```

22     print("Fonctions d'activation : ", self.fonctions_activation)
23     print("Taux d'apprentissage : ", self.taux_apprentissage)
24
25     def definition_reseau(self):
26         w, b = [], []
27         for i in range(len(self.informations_reseau) - 1):
28             w.append(np.random.randn(self.informations_reseau[i + 1],
29                                     self.informations_reseau[i]) / 4)
29             b.append(np.random.randn(self.informations_reseau[i + 1], 1) / 4)
30         return w, b
31
32     def propagation(self, x):
33         self.A = [x]
34         self.V = [x]
35         for i in range(len(self.informations_reseau) - 1):
36             self.A.append(np.dot(self.W[i], self.V[i]) + self.B[i])
37             self.V.append(self.fonctions_activation[i](", self.A[i + 1]))
38
39     def retropropagation(self, y, d):
40         y = ys_to_matrice(y, self.nb_classes).T
41
42         for k in range(1, len(self.V)):
43             if k == 1:
44                 d_v = self.V[-1] - y
45             else:
46                 d_v = np.dot(self.W[-k + 1].T, d_v) *
47                     self.fonctions_activation[-k]("derivee", self.V[-k])
48
49                 d_w = 1 / d * np.dot(d_v, self.V[-k - 1].T)
50                 d_b = 1 / d * np.sum(d_v, axis=1).reshape(self.B[-k].shape[0], 1)
51
52                 self.W[-k] -= self.taux_apprentissage * d_w
53                 self.B[-k] -= self.taux_apprentissage * d_b
54
55     def entrainement(self, donnees, nb_repetitions):
56         x_train, y_train = donnees
57         x_train = x_train.reshape(x_train.shape[0], x_train.shape[1] * x_train.shape[2]).T
58
59         self.nb_entrainements += x_train.shape[0] * nb_repetitions
60         print("\nENTRAINEMENT")
61         for i in range(nb_repetitions):
62             avance = i / nb_repetitions * 100
63             self.propagation(x_train)
64             self.retropropagation(y_train, x_train.shape[1])
65             if avance % 5 == 0:
66                 resultats_corrects = comptage_resultats(np.argmax(self.V[-1], 0), y_train) /
67                     y_train.shape[0]
68                 self.taux_reussite.append(resultats_corrects)
69                 print(int(avance), " % : rendement ", int(resultats_corrects * 100))
70
71     def test(self, donnees):
72         print("\nTEST")
73
74         x_test, y_test = donnees
75         x_test = x_test.reshape(x_test.shape[0], x_test.shape[1] * x_test.shape[2])
76
77         nombre_succes, nombre_donnees_test = 0, len(x_test)
78
79         for i in range(nombre_donnees_test):
80             avancee = i / nombre_donnees_test * 100
81             self.propagation(x_test[i].reshape(x_test[i].shape[0], 1))
82             valeur_pratique = np.argmax(self.V[-1])
83             if avancee % 10 == 0:

```

```

82         print(avancee, " % : ")
83     if valeur_pratique == y_test[i]:
84         nombre_succes += 1
85
86     print("\nNombre de succès : ", nombre_succes)
87     print("Taux de réussite : ", self.taux_reussite * 100, "%")

```

2.6 MNIST

2.6.1 Présentation

Après avoir étudié toute cette théorie, il est plus que nécessaire de pouvoir l'illustrer à partir d'un exemple très concret qu'est la reconnaissance de chiffres manuscrits. Pour ce faire une base de données très connues d'images de chiffres qui se nomme MNIST² et qui contient 70 000 images. Ces images sont de tailles 28 par 28 pixels (confère figure 2.6) - soit 784 informations codées entre 0 et 255³ - qui sont distinguées en deux jeux : un jeu d'entraînement de 60 000 images et un jeu de tests de 10 000 images.



FIGURE 2.6 – Exemple de données de MNIST

Pour l'étude de ces images, nous avons choisi d'utiliser uniquement un réseau de neurones denses. La configuration choisie pour réaliser des prédictions est un réseau de **3 couches**. Naturellement, la couche d'entrée comporte **784 neurones** qui recevront les images et la couche de sortie comporte **10 neurones** qui correspondent aux 10 chiffres de 0 à 9. Nous placerons une couche cachée qui permettra d'effectuer plus de calculs et d'obtenir un réseau plus précis qu'un réseau sans couche cachée. Cependant pour limiter les calculs, nous réduirons cette couche à seulement **10 neurones**. Les fonctions d'activations choisies ici sont la **fonction ReLu** entre les couches 1 et 2, ainsi que la **fonction de probabilités softmax** afin d'obtenir un résultat compris entre 0 et 1 sur la dernière couche.

2.6.2 Optimisation

Estimation de la complexité

Cependant, on rencontre ici notre premier problème : le nombre de calculs nécessaires et la puissance disponible. En effet, il est nécessaire d'effectuer plusieurs entraînements - on en effectuera au moins 100 - sur les 60 000 images avec pour chaque entraînement, en ne se concentrant uniquement sur le réseau et pas les calculs annexes :

- ➔ Propagation : $784 \times 10 \times 10 = 78400$ calculs
- ➔ Rétropropagation : XXX calculs

Nous avons supposé que numpy s'exécutait en temps linéaire et ne possédait aucune optimisation matérielle.

2. Acronyme de *Mixed National Institute of Standards and Technology*.

3. On rappelle que le noir correspond à 0 et que le blanc correspond à 255.

Vectorisation

Il est donc plus que nécessaire de travailler à l'optimisation du réseau de neurones afin d'en améliorer la vitesse d'exécution. Pour ce faire, nous utiliserons la vectorisation des éléments. Il s'agit ici de ne plus travailler sur une image à la fois mais sur un jeu d'entraînement complet et de mettre à jour tous les éléments en une seule fois. Pour M.N.I.S.T. on raisonnera donc sur les 60 000 images de la base de donnée d'entraînement.

Cela ne change en rien la théorie de propagation de l'information, ni celle de la rétropropagation mais de légères adaptations du code ont été nécessaires afin que les produits matriciels puissent s'exécuter convenablement.

2.6.3 Code

```

1  from ReseauVectorise import Reseau
2  from annexe import ReLu, softmax, affichage
3
4  from matplotlib import pyplot as plt
5  import numpy as np
6
7  import keras
8  from keras import layers
9
10
11  (x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()
12
13
14  x_train = x_train.astype("float32") / 255
15  x_test = x_test.astype("float32") / 255
16
17  reseau_mnist = Reseau([28*28, 10, 10], 0.10, [ReLu, softmax])
18  reseau_mnist.entrainement((x_train, y_train), 500)
19  reseau_mnist.test((x_test, y_test))
20
21  taux_succes = reseau_mnist.taux_reussite
22
23  affichage([0.05*i*500 for i in range(len(taux_succes))], [taux_succes], ["Taux de succès"],
    ↳ "Évolution du rendement", "Nombre d'entraînements", "Taux de succès")

```

2.6.4 Résultats

Grâce à des expériences, on obtient l'évolution du rendement ⁴ du réseau de neurones, qui correspond au nombre de bonnes réponses par rapport au nombre de réponses totales.

4. On utilisera ce dernier plutôt que le coût dans un soucis de compréhension.

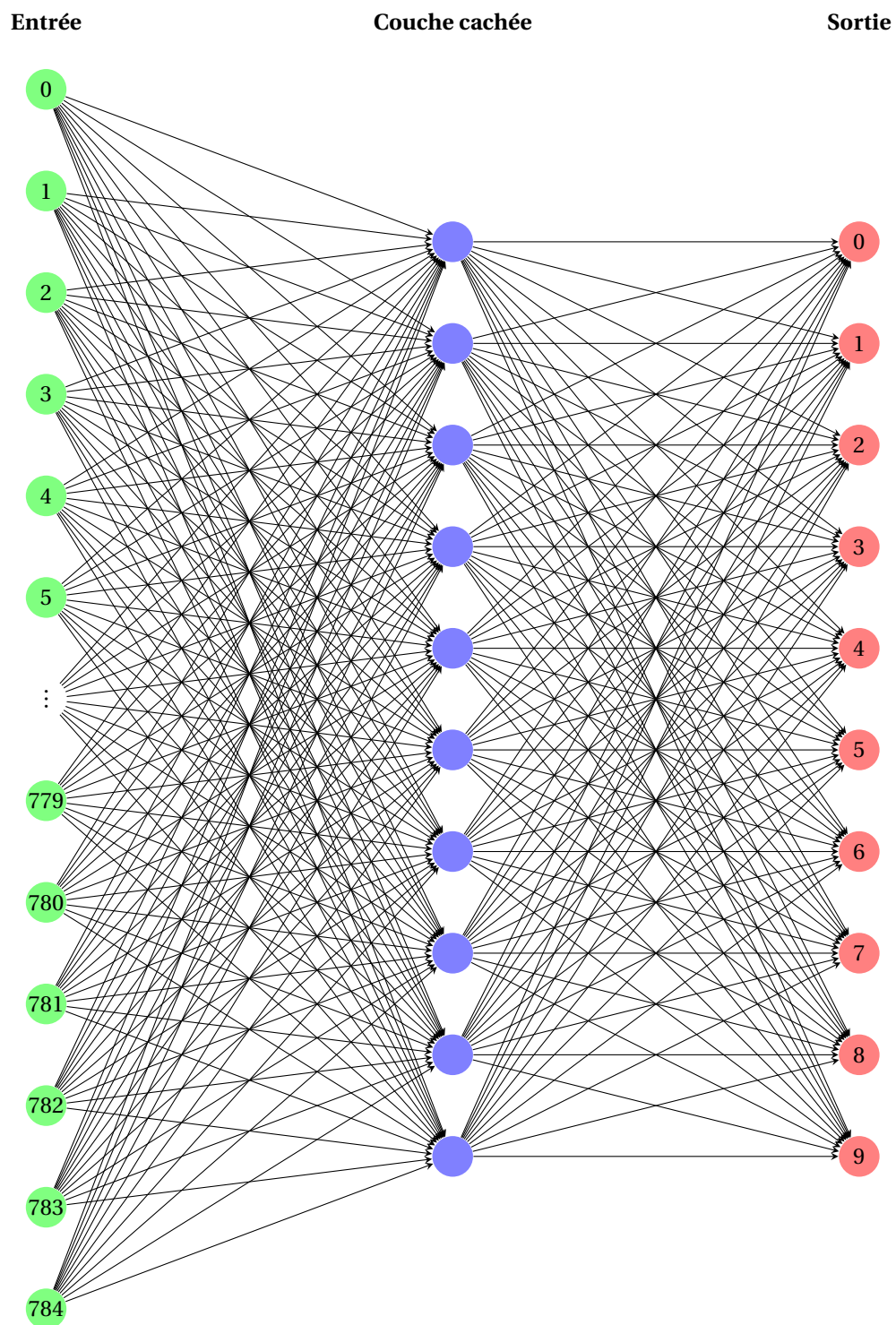


FIGURE 2.7 – Réseau de neurones pour MNIST

Chapitre 3

Projet

À la suite de la pose des bases théoriques nécessaires à la bonne compréhension de ce qui suit, nous allons évoquer notre projet de TIPE.

À l'origine, notre objectif était large. Être en capacité d'analyser des images, influx, vidéo, issue d'un match sportif, afin d'en tirer les joueurs présents, la position de la balle, les scores et autres informations pouvant être utile. Cependant, de rapides recherches nous ont contraintes à réduire nos ambitions au vu des difficultés inhérentes à la réalisation d'un tel projet : puissance de calcul, théories mathématiques et mise en œuvre de la théorie.

Toutefois, nous n'avons pas abandonné notre projet et avons décidé en étudier uniquement deux parties que sont l'identification d'un joueur, grâce à son numéro de maillot, ainsi que l'identification du sport grâce à la balle présente sur le terrain.

Pour ce faire, nous utiliserons des réseaux de neurones convolutifs qui sont très ressemblants, et prennent leur théorie à la source de ce que nous avons pu voir juste avant.

Les données que nous allons utiliser sont originaires de projets disponibles sur le Web et sont référencés dans la présentation.

3.1 Identification du joueur

3.1.1 Convolution d'images

Lors du traitement d'images, il est souvent plus que nécessaire de réaliser un traitement préalable de celles-ci afin d'y mettre en avant leurs informations essentielles. Cela s'effectue au moyen de convolutions. Il en existe plusieurs types : convolution simple, convolution avancée, pooling, dropout, ... Nous nous concentrerons ici sur l'étude uniquement de convolutions simples qui n'évoluent pas et de convolutions avancées qui évoluent en fonction des prédictions du réseau de neurones.

Propagation

Une convolution est une opération matricielle qui consiste à appliquer à une multitude de sous-matrices de l'image une matrice filtre préalablement inversée.

Nom	Variable
Couche	k
Matrice des valeurs	V_k
Fonction d'activation	f_a
Filtre matriciel	F_k

TABLE 3.1 – Définitions mathématiques

La propagation de la convolution s'effectue itérativement sur chaque filtre de convolution F_k .

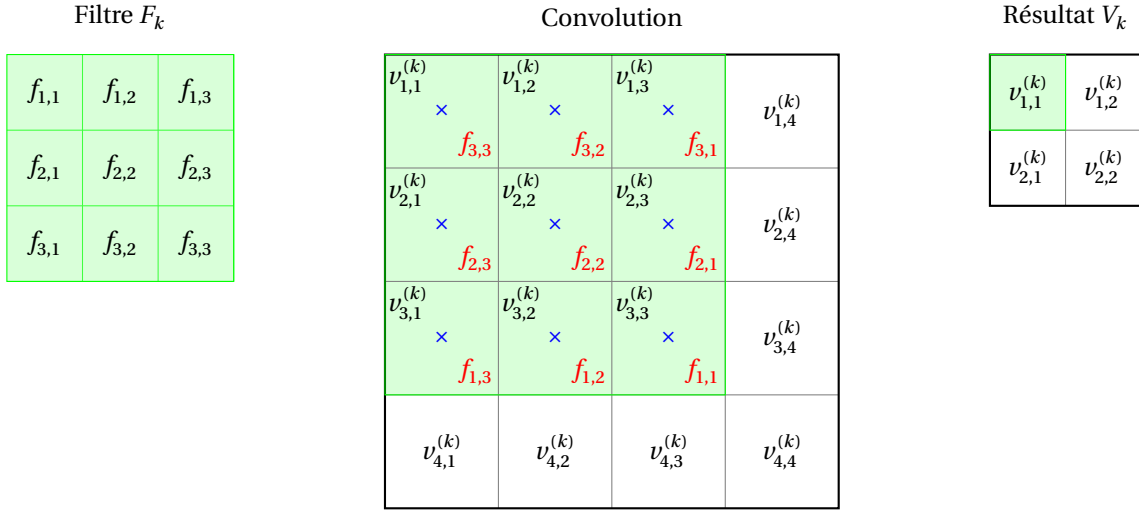


FIGURE 3.1 – Convolution - Propagation

Il s'agit donc d'effectuer les opérations suivantes pour chaque couche de convolution du réseau de neurones :

1. Récupérer les informations d'entrée;
2. Retourner le filtre;
3. Appliquer pour chaque valeur de la matrice l'opération de convolution comme vue précédemment. On ne dépassera pas en rajoutant des zéros!
4. Appliquer la fonction d'activation.

Et on peut passer à la couche suivante!

On n'oubliera pas que la convolution est une opération matricielle. Elle ne peut donc pas gérer la vectorisation du réseau comme réalisée avec MNIST. D'autre part, il faut penser à linéariser la matrice (et réciproquement à la rétropropagation) pour passer de la dernière couche de convolution à la première couche dense.

D'un point de vue mathématiques, la convolution s'exprime de la manière suivante :

$$v_{i,j}^{(k)} = f_a \left(\sum_{m=0}^2 \sum_{n=0}^2 v_{i+m,j+n}^{(k-1)} \times f_{m,n}^{(k)} + b_{i,j}^{(k)} \right) \quad (3.1)$$

Rétropropagation

La rétropropagation dans un réseau de neurones convolutifs suit exactement le même principe que pour un réseau de neurones denses. Il s'agit de l'algorithme de descente de gradient. On vient donc à calculer les dérivées partielles successives - le gradient - de la fonction de coût par rapport aux filtres qui sont en réalité similaires à des poids.

Pour une matrice de taille $H \times W$ et un filtre de taille $k_1 \times k_2$

$$\frac{\partial C}{\partial (F_k)_{m',n'}} = \sum_{a=1}^{H-k_1+1} \sum_{b=1}^{W-k_2+1} \frac{\partial C}{\partial v_{a,b}^{(k)}} \frac{\partial v_{a,b}^{(k)}}{\partial (F_k)_{m',n'}} \quad (3.2)$$

$$\frac{\partial v_{a,b}^{(k)}}{\partial (F_k)_{m',n'}} = \frac{\partial}{\partial (F_k)_{m',n'}} \left(\sum_{m=0}^2 \sum_{n=0}^2 v_{i+m,j+n}^{(k-1)} \times f_{m,n}^{(k)} + b_{i,j}^{(k)} \right) \quad (3.3)$$

$$= \quad (3.4)$$

$$\frac{\partial C}{\partial (F_k)_{i,j}} = \text{convolution} \left(V_{k-1}, \frac{\partial C}{\partial V_k} \right) \quad (3.5)$$

$$\frac{\partial C}{\partial (X_n)_{i,j}} = \sum_{k=1}^n \sum_{a=1}^{H-k_1+1} \sum_{b=1}^{W-k_2+1} \frac{\partial C}{\partial v_{a,v}^{(k)}} \frac{\partial v_{a,v}^{(k)}}{\partial (F_k)_{i,j}} \quad (3.6)$$

$$= \sum_{a=1}^{H-k_1+1} \sum_{b=1}^{W-k_2+1} \frac{\partial C}{\partial v_{a,v}^{(k)}} (V_{k-1})_{a+i,b+j} \quad (3.7)$$

$$= \quad (3.8)$$

$$\frac{\partial C}{\partial (F_k)_{i,j}} = \text{convolution} \left(V_{k-1}, \frac{\partial C}{\partial V_k} \right) \quad (3.9)$$

3.1.2 Code

Opération de convolution

```

1 import numpy as np
2 from annexe import lineaire, sigmoid, ReLu, tanh, softmax
3
4
5 def retournement(matrice):
6     return np.flip(matrice)
7
8 def convolution_2d(matrice, taille=3, fonction_activation=lineaire):
9     n, p = matrice.shape
10    h = taille // 2
11
12    W = retournement(np.random.randn(taille, taille))
13
14    matrice_3 = np.zeros((n + 2 * h, p + 2 * h))
15
16    matrice_3[h:-h, h:-h] = matrice
17
18    matrice_2 = np.zeros((n, p))
19
20    for i in range(n):
21        for j in range(p):
22            coucou = matrice_3[i-h+1:i+taille-h+1, j-h+1:j+taille-h+1]
23            matrice_2[i, j] = fonction_activation("", np.sum(coucou * W))
24    return matrice_2
25
26 def convolution_classiques(matrice, type):
27     n, p = matrice.shape
28     h = 1
29
30     if type == "moyenne":
31         W = 1/9 * np.array([[1, 1, 1], [1, 1, 1], [1, 1, 1]])
32     elif type == "gaussien":
33         W = 1/16 * np.array([[1, 2, 1], [2, 4, 2], [1, 2, 1]])
34     elif type == "pique":
35         W = np.array([[0, -1, 0], [-1, 5, -1], [0, -1, 0]])
36     elif type == "bords":
37         W = np.array([[ -1, -1, -1], [-1, 8, -1], [-1, -1, -1]])
38     elif type == "relief":
39         W = np.array([[ -2, -1, 0], [-1, 1, 1], [0, 1, 2]])
40     else:
41         print("Erreur : type de convolution inconnu")
42         return None
43
44
45    matrice_3 = np.zeros((n + 2 * h, p + 2 * h))
46
47    matrice_3[h:-h, h:-h] = matrice

```

```

48
49     matrice_2 = np.zeros((n, p))
50
51     for i in range(n):
52         for j in range(p):
53             coucou = matrice_3[i-h+1:i+3-h+1, j-h+1:j+3-h+1]
54             matrice_2[i, j] = np.sum(coucou * W)
55     return matrice_2
56

```

Réseau de convolution

```

1  import numpy as np
2  from annexe import ys_to_matrice, comptage_resultats
3  from scipy.signal import convolve2d
4
5  from convolution import retournement
6
7
8  class RéseauConvolutif:
9      def __init__(self, nb_classes, dimensions, taux_apprentissage, nb_rep_entrainement,
10         donnees_entrainement):
11
12         print("\nDEFINITION DU RÉSEAU")
13
14         self.nb_classes = nb_classes
15         self.informations_reseau = dimensions
16         self.fonctions_activation = []
17
18         self.taille_convolution = 2
19         self.taille_dense = 3
20
21         self.taux_apprentissage = taux_apprentissage
22         self.nb_rep_entrainement = nb_rep_entrainement
23         self.donnees_entrainement = donnees_entrainement
24
25         self.A, self.V = [], []
26         self.W, self.B, self.filtres, self.b2 =
27             self.definition_reseau(donnees_entrainement[0].shape[1])
28
29         self.nb_entrainements, self.nb_tests = 0, 0
30         self.taux_reussite = []
31         print(self.fonctions_activation)
32
33     def definition_reseau(self, hauteur_image):
34         w, b, f, b2 = [], [], [], []
35         for k in range(len(self.informations_reseau)-1):
36             if self.informations_reseau[k][0] == "C":
37                 f.append(np.random.randn(3, 3))
38                 b2.append(np.random.randn(hauteur_image-2*len(f), hauteur_image-2*len(f)))
39                 self.fonctions_activation.append(self.informations_reseau[k][1])
40             else:
41                 if self.informations_reseau[k-1][0] == "C":
42                     w.append(np.random.randn(self.nb_classes, (hauteur_image-2*len(f))**2))
43                     b.append(np.random.randn(self.nb_classes, 1))
44                     self.fonctions_activation.append(self.informations_reseau[k][1])
45                 elif k == len(self.informations_reseau)-1:
46                     w.append(np.random.randn(self.nb_classes, self.nb_classes))
47                     b.append(np.random.randn(self.nb_classes, 1))
48                 else:
49                     w.append(np.random.randn(self.nb_classes, self.nb_classes))
50                     b.append(np.random.randn(self.nb_classes, 1))
51                     self.fonctions_activation.append(self.informations_reseau[k][1])
52

```

```

50     for i in b2:
51         print(i.shape)
52     return w, b, f, b2
53
54 def propagation(self, x):
55     self.A = [x]
56     self.V = [x]
57     for k in range(len(self.filtres)):
58         self.A.append(convolve2d(self.V[-1], self.filtres[k], mode='valid') + self.b2[k])
59         self.V.append(self.fonctions_activation[k]("", self.A[-1]))
60
61     self.V[-1] = self.V[-1].reshape(self.V[-1].shape[0] * self.V[-1].shape[1], 1)
62
63     for i in range(len(self.informations_reseau) - len(self.filtres) - 1):
64         self.A.append(np.dot(self.W[i], self.V[-1]) + self.B[i])
65         self.V.append(self.fonctions_activation[i + len(self.filtres)]("", self.A[-1]))
66
67 def retropropagation(self, y):
68     y = ys_to_matrice(y, self.nb_classes).T
69
70     for k in range(1, self.taille_dense):
71         if k == 1:
72             d_v = self.V[-1] - y
73         else:
74             d_v = np.dot(self.W[-k + 1].T, d_v) *
75                 self.fonctions_activation[-k]("derivee", self.A[-k])
76             d_w = np.dot(d_v, self.V[-k - 1].T)
77             d_b = np.sum(d_v, axis=1).reshape(self.B[-k].shape[0], 1)
78
79             self.W[-k] -= self.taux_apprentissage * d_w
80             self.B[-k] -= self.taux_apprentissage * d_b
81
82     d_v = np.dot(self.W[-self.taille_dense + 1].T, d_v)
83     self.V[-k] = self.V[-k].reshape(int(self.V[-k].shape[0] ** .5),
84                                     int(self.V[-k].shape[0] ** .5))
85     d_v = d_v.reshape(self.V[-k].shape[0], self.V[-k].shape[1])
86
87     for k in range(self.taille_dense, self.taille_convolution + self.taille_dense):
88         i = k - self.taille_dense
89
90         d_f = convolve2d(self.V[-k - 1], d_v, mode='valid')
91         d_b2 = d_v
92         d_v = convolve2d(d_v, retournement(self.filtres[-i]), mode='full') *
93             self.fonctions_activation[-k]("derivee", self.A[-k])
94
95         self.filtres[-i] = self.filtres[-i] - self.taux_apprentissage * d_f
96         self.b2[-(i + 1)] = self.b2[-(i + 1)] - self.taux_apprentissage * d_b2
97
98 def entrainement(self):
99     x_train, y_train = self.donnees_entrainement
100     self.nb_entrainements += x_train.shape[0] * self.nb_rep_entrainement
101
102     print("\nENTRAINEMENT")
103
104     for i in range(self.nb_rep_entrainement):
105         avance = i / self.nb_rep_entrainement * 100
106         corr = 0
107
108         for j in range(len(x_train)):
109             x = x_train[j]
110             y = np.array([y_train[j]])
111             self.propagation(x)
112             self.retropropagation(y)

```

```

110         corr += comptage_resultats(np.argmax(self.V[-1], 0) + 1, y)
111
112     if avance % 5 == 0:
113         resultat = corr / len(x_train)
114         self.taux_reussite.append(resultat)
115         print(int(avance), " % : rendement ", int(resultat * 100))
116
117     def test(self, donnees):
118         print("\nTEST")
119
120         x_test, y_test = donnees
121         x_test = x_test.reshape(x_test.shape[0], x_test.shape[1] * x_test.shape[2])
122
123         nombre_succes, nombre_donnees_test = 0, len(x_test)
124
125         for i in range(nombre_donnees_test):
126             avancee = i / nombre_donnees_test * 100
127             self.propagation(x_test[i].reshape(x_test[i].shape[0], 1))
128             valeur_pratique = np.argmax(self.V[-1])
129             if avancee % 10 == 0:
130                 print(avancee, " % : ")
131             if valeur_pratique == y_test[i]:
132                 nombre_succes += 1
133
134         self.taux_reussite = nombre_succes / nombre_donnees_test
135
136         print("\nNombre de succès : ", nombre_succes)
137         print("Taux de réussite : ", self.taux_reussite * 100, "%")
138

```

On souhaite ici identifier un joueur lors d'un match sportif en utilisant un réseau de neurones convolutifs qui va donc réaliser dans un premier temps, plusieurs convolutions puis utiliser des couches denses pour terminer son analyse.

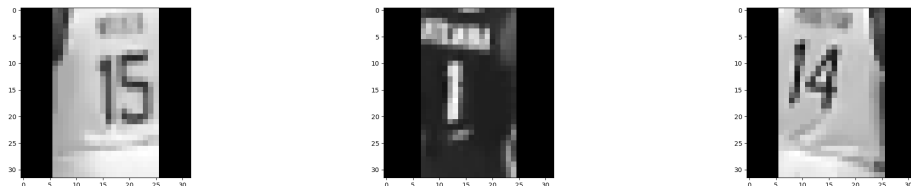


FIGURE 3.2 – Exemples de dossards

La base de donnée se compose de 1200 images de dossards avec une liste contenant les informations vérifiées. Le code suivant permet la lecture des images et leur décodage dans un tableau numpy sous une forme matricielle.

```

1  import numpy as np
2  import pandas as pd
3  from PIL import Image
4
5
6  valeurs = pd.read_csv('data.csv', index_col=0)
7
8  valeurs = valeurs.to_numpy().reshape(1200)
9
10 images = np.zeros((1200, 32, 32))
11
12 for i in range(1200):
13
14     if i < 10:

```



```

15     k = '000'+str(i)
16 elif i < 100:
17     k = '00'+str(i)
18 elif i < 1000:
19     k = '0'+str(i)
20 else:
21     k = str(i)
22
23 images[i] = np.array(Image.open('numbers/' + k + '.png'))

```



Attention !



La base de donnée est loin d'être parfaite, et sera même un problème pour le réseau de neurones. On notera que l'on retrouve plusieurs fois les mêmes images mais qui diffèrent quant à leur placement, orientation et autres modifications que l'on peut effectuer. En outre, ces dernières ne sont pas de très bonne qualité.

3.1.3 Première étude : filtres d'images

Dans un premier temps, on étudiera les résultats en utilisant une méthode de convolution basique que l'on appliquera à chaque image.

On va donc appliquer à chaque matrice des images une convolution en utilisant une conservation de la taille avec un *padding* pour les bords. On effectue les modifications sur l'image

On utilisera le code suivant afin d'effectuer les calculs de convolution - les résultats de ces convolutions sont représentés dans le tableau 3.2 - avec un perceptron multicouches de 3 couches (entrée, cachée et sortie) de tailles $32 \times 32 = 1024$ puis 45 et 45 neurones par couche. Les fonctions d'activations choisies sont ReLu et softmax. On effectuera 2000 tours d'entraînement.

```

1  from ReseauVectorise import Reseau
2  from MaillotsData import images, valeurs
3  from annexe import ReLu, softmax, affichage
4  from convolution import convolution_classiques
5
6  images = images.astype("float32") / 255
7
8  liste = ["moyenne", "gaussien", "pique", "bords", "relief"]
9  y, legendes = [], []
10
11 for k in range(len(liste)):
12     for i in range(len(images)):
13         images[i] = convolution_classiques(images[i], liste[k])
14
15     x_train, y_train = images[:1000], valeurs[:1000]
16     x_test, y_test = images[1000:], valeurs[1000:]
17
18     reseau = Reseau([32*32, 45, 45], 0.01, [ReLu, softmax])
19
20     reseau.entrainement((x_train, y_train), 2000)
21
22     taux_succes = reseau.taux_reussite
23
24     legendes.append("Taux pour "+liste[k])
25     y.append(taux_succes)
26
27 affichage([0.05*i*2000 for i in range(len(y[0]))], y, legendes, "Évolution du rendement",
28           "Nombre d'entraînements", "Taux de succès")

```

Les résultats concernant ce réseau et cette méthode de convolution sont classiques pour les 4 premiers (confère figure 3.3). Toutefois, on note que le filtre de mise en relief qui joue avec les contrastes produit des résultats tout à fait impressionnants tout en minimisant les calculs nécessaires.

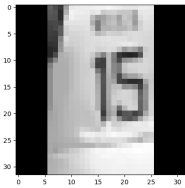
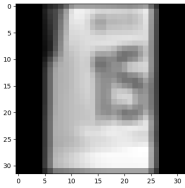
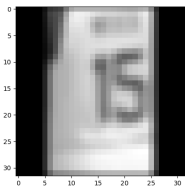
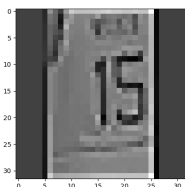
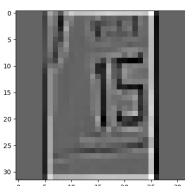
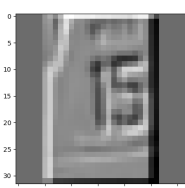
Nom	Filtre	Image
Original		
Flou (moyenne)	$\frac{1}{9} \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$	
Flou gaussien	$\frac{1}{9} \begin{pmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{pmatrix}$	
Piqué	$\frac{1}{9} \begin{pmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{pmatrix}$	
Bords	$\frac{1}{9} \begin{pmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{pmatrix}$	
Mise en relief	$\frac{1}{9} \begin{pmatrix} -2 & -1 & 0 \\ -1 & 1 & 1 \\ 0 & 1 & 2 \end{pmatrix}$	

TABLE 3.2 – Images des maillots traitées

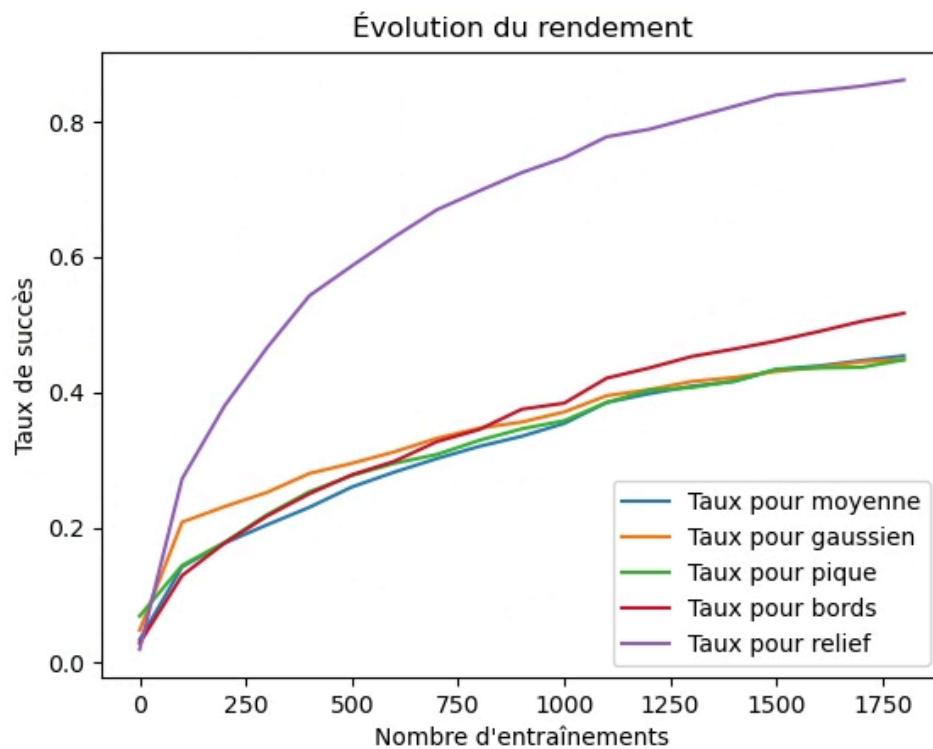


FIGURE 3.3 – Résultats des premières convolutions.

On peut donc conclure de cette première étude qu'elle est suffisante pour répondre à la problématique posée. Il est tout à fait possible d'utiliser une convolution simple sur les images suivi d'un perceptron multicouches traditionnel.

```

1 from ReseauConvolution import ReseauConvolutif
2 from MaillotsData import images, valeurs
3 from annexe import ReLu, softmax, lineaire
4
5
6 x_train, y_train = images[:1000], valeurs[:1000]
7 x_test, y_test = images[1000:], valeurs[1000:]
8
9 x_train = x_train.astype("float32") / 255
10 x_test = x_test.astype("float32") / 255
11
12 reseau = ReseauConvolutif(45, [
13     ("C", lineaire),
14     ("C", lineaire),
15     ("D", ReLu),
16     ("D", softmax),
17     ("D", lineaire)
18 ], 0.01, 1000, (x_train, y_train))
19
20 reseau.entrainement()

```

3.1.4 Deuxième étape : CNN à 1 niveau

Pour réaliser une analyse de telles images, on utilise un réseau de neurones convolutifs avec 2 couches de convolution et 2 couches de réseau dense qui devraient permettre de résoudre ce problème d'analyse d'images.

Les neurones de convolutions utiliseront la fonction d'activation ReLu pour propager leurs informa-

tions. Tandis que le réseau de neurones se verra équiper des fonctions ReLu et softmax.

```

1  from ReseauVectorise import Reseau
2  from MaillotsData import images, valeurs
3  from annexe import ReLu, softmax, affichage
4  from convolution import convolution_classiques
5
6  images = images.astype("float32") / 255
7
8  liste = ["moyenne", "gaussien", "pique", "bords", "relief"]
9  y, legendes = [], []
10
11 for k in range(len(liste)):
12     for i in range(len(images)):
13         images[i] = convolution_classiques(images[i], liste[k])
14
15     x_train, y_train = images[:1000], valeurs[:1000]
16     x_test, y_test = images[1000:], valeurs[1000:]
17
18     reseau = Reseau([32*32, 45, 45], 0.01, [ReLu, softmax])
19
20     reseau.entrainement((x_train, y_train), 2000)
21
22     taux_succes = reseau.taux_reussite
23
24     legendes.append("Taux pour "+liste[k])
25     y.append(taux_succes)
26
27 affichage([0.05*i*2000 for i in range(len(y[0]))], y, legendes, "Évolution du rendement",
28           "Nombre d'entraînements", "Taux de succès")

```

3.1.5 Troisième étape : CNN à plusieurs niveaux

Dans le réseau de neurones convolutifs précédemment codé, on se rend compte que les prédictions données par le réseau sont largement insuffisantes. Ce résultat n'est pas incohérent. En effet, il est plus rare que d'effectuer une convolution à 2 dimension dans un réseau de neurones convolutifs. Dans la plupart des cas, il est nécessaire de démultiplier les matrices d'images ainsi que le nombre de filtres, et par suite le nombre de paramètres modifiables par le réseau.

Dans un troisième temps, on utilise donc un CNN multicouches à 3 dimensions qui possédera 2 couches de convolution avec pour la première **32 filtres** et la seconde **64 filtres**. Ce qui nous donne 2326958 paramètres modifiables dans le réseau de neurones.

Pour des raisons de complexité et de faisabilité, on utilisera le module keras pour implémenter un tel réseau convolutif en utilisant les mêmes données.

```

1  import numpy as np
2  import keras
3  from keras import layers
4  from MaillotsData import images, valeurs
5
6  x_train, y_train = images[:1000], valeurs[:1000]
7  x_test, y_test = images[1000:], valeurs[1000:]
8
9  num_classes = 46
10 input_shape = (32, 32, 1)
11
12 x_train = x_train.astype("float32") / 255
13 x_test = x_test.astype("float32") / 255
14 x_train = np.expand_dims(x_train, -1)
15 x_test = np.expand_dims(x_test, -1)
16 print("x_train shape:", x_train.shape)
17 print(x_train.shape[0], "train samples")
18 print(x_test.shape[0], "test samples")

```

```

19
20 y_train = keras.utils.to_categorical(y_train, num_classes)
21 y_test = keras.utils.to_categorical(y_test, num_classes)
22
23 model = keras.Sequential(
24     [
25         keras.Input(shape=input_shape),
26         layers.Conv2D(32, kernel_size=(3, 3), activation="relu"),
27         layers.Conv2D(64, kernel_size=(3, 3), activation="relu"),
28         layers.Flatten(),
29         layers.Dense(num_classes, activation="softmax"),
30     ]
31 )
32
33 model.summary()
34
35 batch_size = 128
36 epochs = 1000
37
38 model.compile(loss="categorical_crossentropy", optimizer="adam", metrics=["accuracy"])
39
40 model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs, validation_split=0.1)
41
42 score = model.evaluate(x_test, y_test, verbose=0)
43 print("Test loss:", score[0])
44 print("Test accuracy:", score[1])

```

Afin d'obtenir des résultats corrects, on effectuera **10000 entraînements** des 1000 données. Après l'exécution d'un tel programme, on obtiendra un **rendement de 60 %**. On conservera la même logique pour les paramètres que les réseaux de neurones précédents.

3.1.6 Quatrième étape : algorithme des moyennes

Enfin, on conclura l'étude avec un dernier algorithme qui est celui des plus proches voisins.

```

1 import numpy as np
2 from MaillotsData import images, valeurs
3 from annexe import ReLu, softmax, lineaire
4 from matplotlib import pyplot as plt
5
6
7 x_train, y_train = images[:1000], valeurs[:1000]
8 x_test, y_test = images[1000:], valeurs[1000:]
9
10 x_train = x_train.astype("float32") / 255
11 x_test = x_test.astype("float32") / 255
12
13 assert x_train.shape == (1000, 32, 32) and y_train.shape == (1000,)
14 assert x_test.shape == (200, 32, 32)
15 print(y_train.shape)
16
17
18 def distance(p1, p2):
19     d = 0
20     for i in range(p1.shape[0]):
21         for j in range(p1.shape[1]):
22             d += (p1[i, j] - p2[i, j])**2
23     return d**0.5
24
25 print("nombre de données dans x_train :", len(x_train))
26 print("nombre de données dans x_test :", len(x_test))
27

```

```

28 def voisins(x, k):
29     indices = sorted(range(len(x_train)), key=lambda i: distance(x, x_train[i]))
30     return indices[:k]
31
32 def plus_frequent(L):
33     compte = {}
34     for e in L:
35         compte[e] = compte.get(e, 0) + 1
36     return max(compte, key=compte.get)
37
38 def knn(x, k):
39     V = voisins(x, k)
40     return plus_frequent([y_train[i] for i in V])
41
42 def precision(k):
43     n = 0
44     for i in range(len(x_test)):
45         if knn(x_test[i], k) == y_test[i]:
46             n += 1
47     return n / len(x_test)
48
49 def plot_precision(kmax):
50     import matplotlib.pyplot as plt
51     R = range(1, kmax)
52     L = [precision(k) for k in R]
53     print(L)
54     plt.plot(R, L)
55     plt.show()
56
57 plot_precision(10)

```

3.2 Identification du sport

TRAVAIL DE TRISTAN

Chapitre 4

Conclusion

- ➔ Résultats mitigés;
- ➔ Difficultés et problèmes rencontrés :
 - ➔ Algorithmique complexe;
 - ➔ Puissance de calcul;
 - ➔ Données nécessaires.