MÉMOIRE

PRÉSENTÉ À

L'UNIVERSITÉ DU QUÉBEC À RIMOUSKI

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN INFORMATIQUE

PAR

MARIE-PIER MARQUIS

DÉTECTION AUTOMATIQUE DE LA PRÉSENCE DE LA REINE D'ABEILLES SUR UN CADRE DE RUCHE

AOÛT 2021

TABLE DES MATIÈRES

TABLE DES FIGURES

LISTE DES TABLEAUX

RÉSUMÉ

La présence de la reine au sein d'une colonie d'abeilles joue un rôle vital. Il est important de savoir si la reine est présente dans la colonie, mais aussi de pouvoir la localiser précisément. Cette tâche est ardue pour l'apiculteur et son efficacité dépend grandement de son acuité visuelle. L'objectif de ce mémoire est d'automatiser la détection de la reine en utilisant une image d'un cadre de ruche d'abeilles. Nous proposons et comparons deux modèles de classification qui pourront distinguer une image contenant une reine d'une image ne contenant pas de reine. La première méthode utilise une technique de classification classique : les machines à vecteurs de support. La deuxième méthode fait intervenir l'apprentissage profond : les réseaux de neurones à convolution. Nous avons obtenu des résultats prometteurs pour les deux méthodes, mais la méthode avec un réseau de neurones à convolution (90.12%) a offert de meilleurs résultats que la méthode classique (84.05%). Notre équipe prévoit de poursuivre les recherches et d'améliorer ces résultats. Nous voulons intégrer le meilleur modèle dans une application mobile qui permettra à l'apiculteur de prendre une photo d'un cadre et d'avoir la confirmation de la présence ou de l'absence de la reine.

INTRODUCTION

0.1 CONTEXTE ET PROBLÉMATIQUE

Les abeilles sont entre autres importantes pour la pollinisation des fleurs. Elle est plus efficace que la simple pollinisation par le vent ou encore certaines méthodes manuelles. Elle améliore la qualité et la quantité que certains plants de fruits et de légumes peuvent produire (Klatt et al. (2014)). Dans les dernières décennies, on constate une crise en agriculture : les pollinisateurs ne sont pas suffisamment nombreux pour pouvoir répondre à la demande de production agricole qui ne cesse d'augmenter (Aizen et Harder (2009)). Le déclin de pollinisateurs, sauvages ou domestiques, a des impacts importants sur la production, mais aussi sur tout un écosystème (Potts et al. (2010)). L'étude des abeilles est donc importante afin de comprendre leur déclin et de pouvoir améliorer les techniques pour améliorer l'élevage et leur survie.

Lorsque l'on fait de l'élevage d'abeilles, il est important de suivre de près le comportement des colonies. L'apiculteur doit s'assurer de la stabilité de la colonie d'abeilles. En général, il tient un calendrier qui prévoit les interventions à faire auprès de la ruche à différents moments de la saison. Il doit par exemple vérifier si un parasite envahit la colonie. Si l'infestation est trop importante, il est possible de perdre une partie de la colonie ou la colonie complète. Il a

bien sûr dans ses tâches la récolte du miel produit par les abeilles.

Un des individus les plus importants de la ruche est la reine. Celle-ci doit entre autres coordonner les ouvrières et pondre les oeufs. Sa présence est requise pour la survie de la colonie. Il faut donc savoir si elle est présente au sein de la colonie. Parfois, il faut aussi être en mesure de la localiser précisément puisque l'apiculteur peut devoir la retirer pour mettre une nouvelle reine à la place, diviser la colonie en deux, etc. Cette dernière tâche de recherche de la reine est ardue. En effet, l'apiculteur doit prendre chaque cadre de la ruche et le parcourir du regard. Cela peut être long et dépend grandement de l'acuité visuelle de l'apiculteur. Il existe aussi des méthodes plus invasives qui consistent à faire passer chaque abeille à travers un filtre qui laisse seulement passer les ouvrières. On peut ainsi repérer la reine.

0.2 QUESTIONS DE RECHERCHE

Comme décrit plus haut, la tâche consistant à détecter et localiser la reine sur un cadre de ruche d'abeilles est ardue. Nous émettons comme question de recherche : Est-ce que la détection des reines de manière automatique serait plus efficace que la détection manuelle ?

Plusieurs méthodes de classification d'images existent. En considérant le nombre d'images bien identifiées, une méthodes de classification par apprentissage profond offre-t-elle un meilleur taux de reconnaissance qu'une méthode classique ?

0.3 CONTRIBUTION

La contribution principale de ce mémoire, qui a été publié dans un article scientifique (Marquis et al. (2021)), est le développement et la comparaison de deux modèles de classification d'images d'abeilles. Le premier modèle fait intervenir une méthode classique basée sur les machines à vecteurs de support (**SVM**, *Support Vector Machine*). Le deuxième modèle est basé sur un réseau de neurones à convolution (**CNN**, *Convolutional Neural Network*).

Comme contribution marginale, une application a été développée afin d'étiqueter la base de données. Cette application permet d'isoler la reine ou les ouvrières présentes sur une image de cadre d'abeilles et de leur associer des étiquettes selon des classes prédéfinies.

0.4 MÉTHODOLOGIE

La figure 1 présente les différentes étapes de l'acquisition des données jusqu'à l'obtention des résultats. Il y a d'abord la cueillette des images utilisées dans les expérimentations qui a été faite par Guillaume Gingras lors d'un stage de premier cycle. À partir d'une image contennant une reine, une imagette de reine est extraite manuellement. Mr Gingras a ensuite développé une application permettant de découper les images en imagettes et d'associer des étiquettes à ces imagettes. À partir des images ne contenant pas de reine ou des images dont la reine a été extraite, il a été possible de créer des imagettes d'ouvrière à l'aide de cette application. La banque d'images ainsi que l'application ont été bonifiées au cours de ce projet. Les images ont subi des prétraitements comme le redimensionnement. Il y a ensuite la création et l'entraînement des modèles ainsi que leur évaluation. Finalement, nous comparons et analysons les résultats de ces modèles.

0.5 ORGANISATION DU MÉMOIRE

Le mémoire est organisé comme suit. Le chapitre 1 présente les fondements qui sont utiles à la compréhension du domaine d'intérêt et du mémoire. L'état de l'art se retrouve au chapitre 2. Nos deux méthodes sont proposées dans le chapitre 3. Finalement, le chapitre 4 présente l'expérimentation, les résultats et l'analyse.

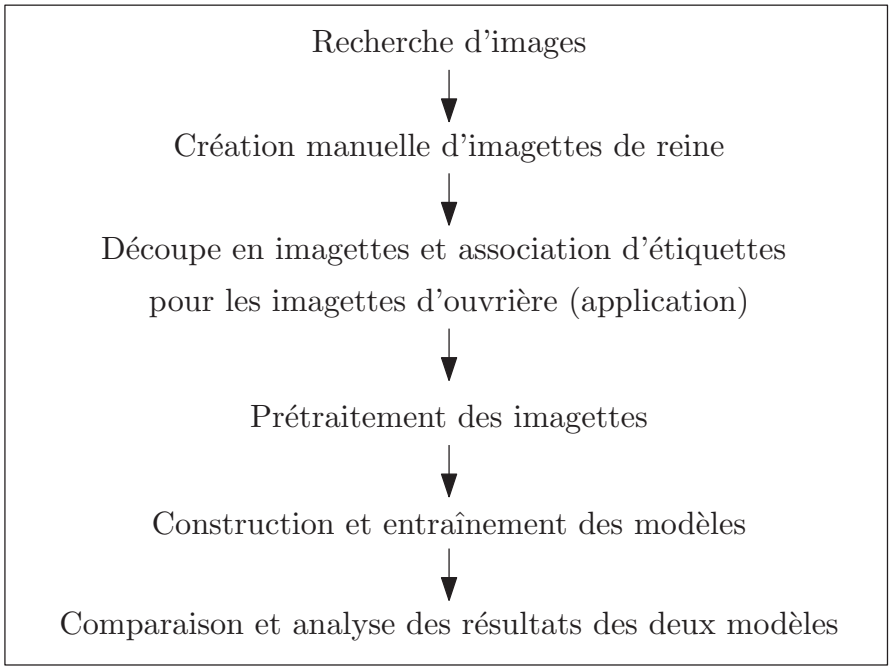


FIGURE 1 – Vue d'ensemble de la méthodologie.

CHAPITRE 1

FONDEMENTS DES TECHNIQUES D'APPRENTISSAGE AUTOMATIQUE

Ce chapitre présente les principales notions utiles à la compréhension de *l'apprentissage automatique*. Dans ce mémoire, le terme *apprentissage* est utilisé pour faire référence à l'apprentissage automatique. Ce sujet englobe les techniques qui permettent de construire un modèle qui pourra prédire sans avoir à programmer le problème. Bien que l'apprentissage automatique s'applique à plusieurs domaines, nous décrivons majoritairement les fondements pour la classification d'images. Différentes méthodes classiques et par apprentissage profond sont présentées. Les techniques utilisées dans ce mémoire y sont décrites avec plus de détails.

La section 1.1 présente les différents types d'apprentissage. Les méthodes d'apprentissage classiques sont détaillées à la section 1.2 et les méthodes par apprentissage profond à la section 1.3. Finalement, les contraintes liées à l'apprentissage sont présentées à la section 1.4.

1.1 TYPES D'APPRENTISSAGE

Dans cette section, nous présentons les trois principaux types d'apprentissage. Selon le problème, une approche supervisée, non supervisée ou par renforcement est adoptée. Nous pouvons aussi retrouver l'apprentissage semi-supervisé.

1.1.1 SUPERVISÉ

Il est question d'apprentissage **supervisé** lorsque chaque élément de l'ensemble de données a été étiqueté (Kotsiantis et al. (2007)). Nous connaissons donc le résultat que le modèle devrait prédire. Un élément peut s'écrire comme un couple (x_i, y_i) où x_i représente la i^e donnée et y_i son étiquette. Supposons que nous avons un ensemble de n données à classer : $(X,Y) = \{(x_1,y_1),(x_2,y_2),\cdots,(x_n,y_n)\}$. En apprentissage supervisé, nous essayons de trouver une fonction f qui donnera une bonne approximation de $f(X) = Y$. Prenons l'exemple de classification des images d'abeilles. Lorsque ces images sont accompagnées d'une étiquette renseignant sur la classe, reine ou non reine, nous parlons d'apprentissage supervisé.

Nous pouvons diviser ces données en deux ensembles distincts. Le premier est l'ensemble d'entraînement. C'est sur cet ensemble que se fait l'apprentissage et à partir duquel la fonction f est approximée. L'algorithme prend en compte la donnée et son étiquette. Une fois le modèle bâti, nous pouvons vérifier sa précision avec le deuxième ensemble qui est celui de validation. Puisque les données de cet ensemble n'ont jamais été vues lors de l'entraînement, nous pouvons les utiliser pour analyser l'état de l'apprentissage. Le modèle n'est pas ajusté à la suite du passage de l'ensemble de validation.

Dans ce type d'apprentissage, nous pouvons retrouver les problèmes de classification, comme dans ce présent mémoire, et les problèmes de régression. Les deux méthodes proposées

dans ce mémoire, utilisant les **SVM** ainsi que les **CNN**, sont des méthodes appartenant à l'apprentissage supervisé et sont présentés ci-dessous.

1.1.2 NON SUPERVISÉ

En apprentissage *non supervisé*, les données $x_1, x_2, x_3 \cdots x_n$ ne sont pas étiquetées (Ghahramani (2003), Khanum et al. (2015)). L'algorithme d'apprentissage tente de retrouver des similitudes et de faire des liens au sein des données.

Parmi les apprentissages non supervisés, nous pouvons y retrouver des problèmes de partitionnement de données. Prenons l'exemple des images de reines et d'ouvrières. L'algorithme pourrait classer ces images en deux catégories en trouvant des caractéristiques particulières pour les reines et pour les ouvrières. Contrairement à l'apprentissage supervisé, aucune information sur la classe n'est donnée avec l'image.

1.1.3 RENFORCEMENT

Dans ce type d'apprentissage, l'algorithme, aussi appelé *agent*, interagit avec son environnement en posant des actions (Arulkumaran et al. (2017)). Chaque action, mène soit à une récompense pour une bonne décision, soit à une punition pour une mauvaise décision. Elle peut aussi changer l'état de l'environnement. Cet état détermine quelles actions l'agent pourra par la suite effectuer. Pensons à un jeu d'échec où chaque action peu soit mettre le joueur en bonne posture ou en mauvaise posture. L'état du jeu donne les actions possibles que l'agent peut effectuer à son tour. À la figure 1.1, nous retrouvons un schéma qui représente l'interaction de l'agent avec son environnement.

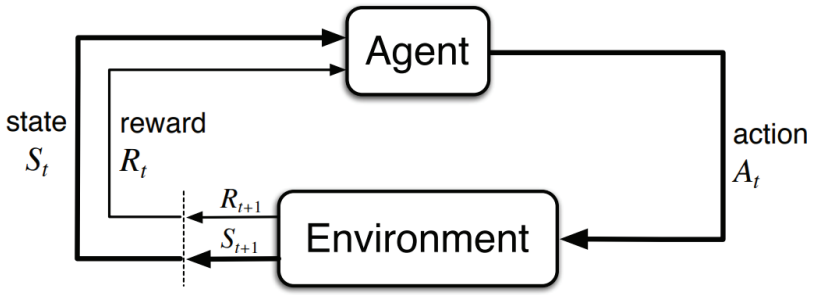


FIGURE 1.1 – Représentation de l'interaction entre l'agent et son environnement en apprentissage par renforcement (Sutton et Barto (2018)).

1.2 MÉTHODES D'APPRENTISSAGE CLASSIQUES

Une partie des méthodes d'apprentissage classiques, comme celle utilisant les **SVM**, peuvent se diviser en trois étapes, telles qu'illustrées à la figure 1.2 (Cortes et Vapnik (1995), Chandra et Bedi (2018)). La première étape consiste à extraire des caractéristiques de l'image. La deuxième étape est la réduction de la dimensionnalité. Enfin, la dernière étape est la classification.

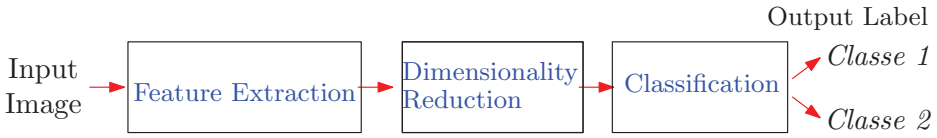


FIGURE 1.2 – Représentation d'une méthode d'apprentissage classique.

Plusieurs techniques existent pour ces trois étapes. Nous avons choisi une structure qui a fait ses preuves dans le domaine de reconnaissance d'expressions faciales (Yaddaden et al. (2021)) et de la détection de piétons (Kobayashi et al. (2007)). L'extracteur de caractéristique est le descripteur nommé histogramme de gradient orienté (**HOG**, *Histogram of oriented gradient*) (Dalal et Triggs (2005)), la dimensionnalité a été réduite avec l'analyse par composante principale (**PCA**, *Principal Component Analysis*) (Wold et al. (1987)) et la classification est effectuée avec un **SVM**.

Pour l'approche avec apprentissage profond, qui est détaillée à la section 1.3, nous avons utilisé les **CNN** (LeCun et al. (1990), Rawat et Wang (2017)). Les **CNN** s'appuient sur l'algorithme du perceptron multicouche (**MLP**, *Multilayer Perceptron*). Puisque le **MLP** est une méthode d'apprentissage classique, elle est décrite dans cette section (Hornik et al. (1989)).

1.2.1 EXTRACTION DES CARACTÉRISTIQUES

La première étape est l'extraction des caractéristiques. Elle permet de faire ressortir des caractéristiques importantes et utiles de l'image tout en enlevant l'information qui est moins pertinente. Nommons *speed up robust features* présentée dans Bay et al. (2006). Nous retrouvons aussi des extracteurs de caractéristiques dans le domaine de l'audio. Nommons-en quelques-uns : *mel frequency cepstral coefficient* (**MFCC**) présenté par Davis et Mermelstein (1980), *Short term Fourier transform* introduit par Lenssen et Needell (2014) et *tonnetz* que nous pouvons retrouver dans Cohn (1998).

Parmi les techniques, nous retrouvons un descripteur pour les images nommé motif binaire local (**LBP**, *Local Binary Pattern*) (Ojala et al. (1996), Ojala et al. (2002)). Il permet de mettre en avant l'information liée à la texture sur l'image. Son fonctionnement est simple et s'effectue en deux étapes. La première étape consiste à consulter les i pixels $x_{r,i}$ voisinant le pixel central $x_{0,0}$ dans un rayon $r \geq 1$. Les pixels $x_{r,i}$ représentent les pixels traversés par le cercle à la figure 1.3. Nous comparons ces pixels avec le pixel central suivant l'équation 1.1.

$$f(x_{r,i}) = \begin{cases} 1 & \text{si } x_{r,i} \geq x_{0,0} \\ 0 & \text{sinon} \end{cases} \tag{1.1}$$

La deuxième étape est le codage des résultats de l'étape précédente avec l'équation 1.2.

$$LBP = \sum_i f(x_{r,i}) \times 2^i \tag{1.2}$$

Nous remplaçons finalement le pixel central $x_{0,0}$ par cette somme. Nous pouvons voir le processus illustré à la figure 1.3.

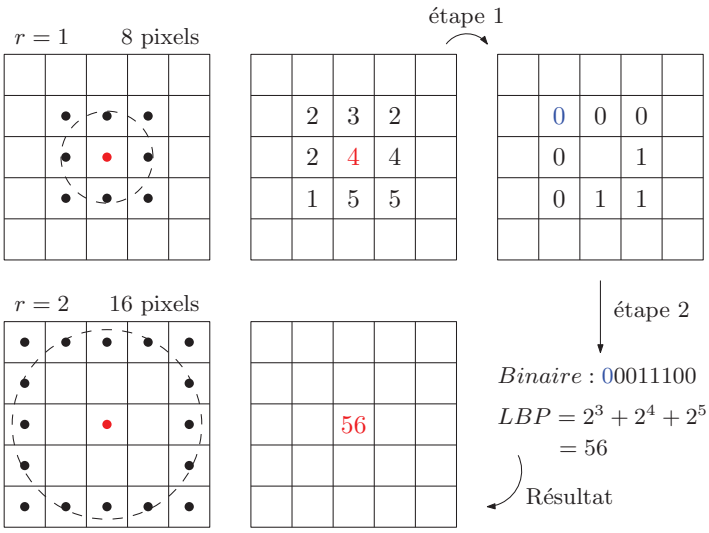


FIGURE 1.3 – Exemple du descripteur LBP avec un rayon de 1 et représentation avec un rayon de 2.

Finalement, nous devons construire un histogramme ayant pour axe des x les valeurs calculées à la deuxième étape et pour axe des y la fréquence à laquelle ces valeurs reviennent dans l'image.

La technique utilisée dans ce mémoire est le populaire descripteur **HOG**. Tout comme la technique précédente, il contient une partie locale où nous regardons localement le voisinage d'un pixel et une partie globale lorsque nous mettons en commun l'information locale de chaque pixel. Elle permet d'intensifier les changements brusques dans une image, comme un contour. Nous devons d'abord calculer le gradient horizontal et vertical pour chaque pixel d'une image en considérant le voisinage du pixel. Calculons le gradient du pixel $p_{i,j}$ illustré à la figure 1.4.

	$p_{i,j-1}$	
$p_{i-1,j}$	$p_{i,j}$	$p_{i+1,j}$
	$p_{i,j+1}$	

FIGURE 1.4 – Représentation du voisinage du pixel $p_{i,j}$.

Le gradient, $\nabla G = (G_x, G_y)$, est un vecteur à deux composantes où $G_x = p_{i+1,j} - p_{i-1,j}$ correspond à la partie horizontale et $G_y = p_{i,j+1} - p_{i,j-1}$ à la partie verticale. Nous pouvons représenter ce vecteur à l'aide d'un triangle rectangle, comme à la figure 1.5, où l'hypoténuse représente G alors que les côtés représentent les composantes horizontales et verticales. Ce qui nous intéresse est la norme et l'orientation de ce vecteur.

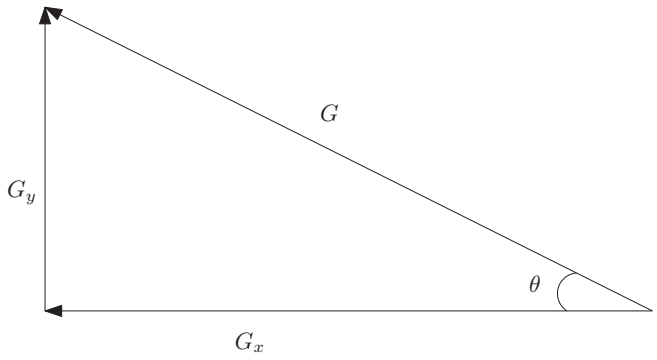


FIGURE 1.5 – La décomposition du vecteur gradient par ses composantes horizontales et verticales.

La norme est donnée par $Norme_2()$ tel que :

$$Norme_2(G) = \sqrt{G_x^2 + G_y^2}. \tag{1.3}$$

L'orientation est donnée par $\theta()$ tel que :

$$\theta(G) = \tan^{-1}\left(\frac{G_y}{G_x}\right). \tag{1.4}$$

Ainsi, pour chaque pixel, nous obtenons un gradient orienté G, dont la longueur est donnée par $Norme_2(G)$ et la direction est donnée par $\theta(G)$. Nous divisons l'image en cellules de $n \times n$ pixels (nous nous sommes assurés au départ que la taille de l'image se divise par $n \times n$). Nous construisons un histogramme pour les n^2 gradients orientés de chaque cellule comme suit. L'axe horizontal de l'histogramme est gradué de 0° à 180°. Nous divisons cet axe en classes. Pour l'exemple, prenons des classes d'une amplitude de 20°. Ainsi, il y a 9 classes. La fréquence de chaque classe dépend de la norme des gradients. Si un pixel a un gradient de norme 12 et une orientation de 20°, alors nous ajoutons un poids de 12 dans la classe 20. Si un pixel a un gradient de 80 et une orientation de 125°, alors nous plaçons $\frac{125-120}{20} * 80 = 20$ dans la classe 120 et $\frac{140-125}{20} * 80 = 60$ dans la classe 140. Ce processus se retrouve à la figure 1.6.

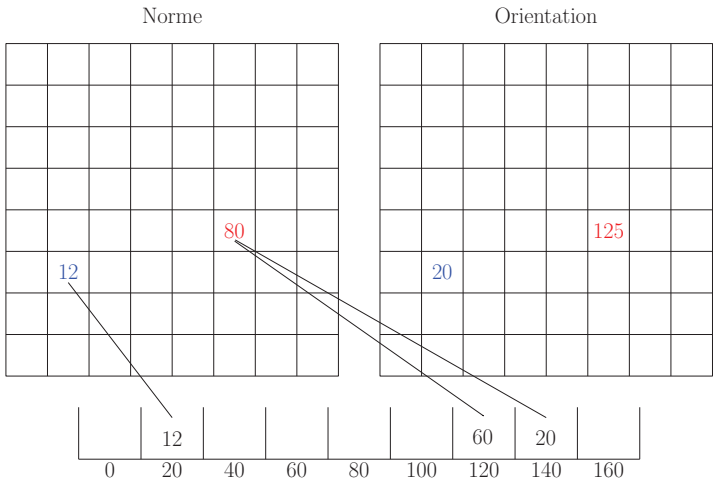


FIGURE 1.6 – Les normes sont ajoutées selon leur orientation pour créer l'amplitude de chaque classe.

Pour cette étape, nous obtenons un vecteur de 9 éléments pour chaque cellule de dimension

$n \times n$ de l'image. Pour une image de dimension $(k \cdot n) \times (l \cdot n)$, nous obtenons donc $k \times l$ vecteurs.

Pour atténuer les variations de lumière sur l'image, nous effectuons une normalisation sur nos histogrammes. Pour un vecteur quelconque $\overrightarrow{v} = (v_1, v_2, \cdots, v_k)$, un vecteur de même direction, mais normalisé, est donné par

$$\overrightarrow{v}_{norm} = \left(\frac{v_1}{nomre_2(\overrightarrow{v})}, \frac{v_2}{nomre_2(\overrightarrow{v})}, \cdots, \frac{v_k}{nomre_2(\overrightarrow{v})} \right).$$

Nous pouvons faire la normalisation sur chaque vecteur de 9 éléments des groupes de $n \times n$ pixels. En pratique, nous glissons une fenêtre de $m \times m$ cellules pour former ce que nous appelons des *blocs*. Nous concatènons les vecteurs des cellules d'un bloc que nous normalisons ensuite. Finalement, pour représenter l'image complète, nous concaténons tous les vecteurs normalisés des blocs afin d'obtenir un seul vecteur. Ce dernier sera utilisé à l'étape suivante.

1.2.2 RÉDUCTION DE LA DIMENSIONNALITÉ

Cette technique permet de réduire la quantité de variables en entrée du classificateur. Elle réduit la taille du vecteur caractéristique en omettant des données redondantes et bruitées. Cela permet d'obtenir un modèle moins complexe qui généralise mieux de nouvelles données. Aussi, il est plus rapide de construire le modèle quand la dimension est moindre (Witten et al. (2017)).

Parmi ses techniques, nous retrouvons la décomposition en valeurs singulières (**SVD**, *Singular value decomposition*) (Wall et al. (2003)) qui permet, selon le théorème, de réécrire toute matrice en un produit de trois matrices :

$$A = U\Sigma V^T,$$

où U et T ont des colonnes orthogonales et Σ est une matrice diagonale qui contient les valeurs singulières de la matrice A. Il est possible de réduire à 0 les k plus petites valeurs singulières et ainsi obtenir une approximation de la matrice A qui aura un rang inférieur.

Nous pourrions aussi nommer la technique nommée analyse discriminante linéaire ((**LDA**), *Linear Discriminant Analysis*) (Izenman (2013)) ou *Isomap Embedding* (Balasubramanian et al. (2002)). Pour ce mémoire, nous avons choisi la réduction de la dimensionnalité par **PCA**.

Soit les données obtenues après l'extraction des caractéristiques $X^{(1)}, X^{(2)}, \cdots, X^{(m)}$ où $X^{(i)} = (x_1^{(i)}, x_2^{(i)} \cdots x_n^{(i)})$, m est le nombre d'entrées différentes et n la dimension de chaque entrée. Plaçons ces vecteurs en lignes pour obtenir la matrice X. La première variable se retrouve sur la première colonne alors que la dernière variable se retrouve sur la n^e colonne. L'idée est de représenter les données $X^{(i)}$ selon de nouveaux axes orthogonaux. Voici les étapes de la méthode du **PCA**.

1. Une série de données avec des valeurs plus élevées qu'une autre série a un poids plus important. La première étape, pour éviter ce problème, est la normalisation des données. Pour ce faire, nous soustrayons chaque donnée par la moyenne et nous divisons ensuite par l'écart-type :

   $$z = \frac{valeur - moyenne}{cart - type}.$$

   Les nouvelles séries de données auront une moyenne de 0 et un écart-type de 1. Cela permet de mieux comparer les données. Nous obtenons ainsi de nouvelles données normalisées. Chaque $X^{(1)}, X^{(2)}, \cdots, X^{(m)}$ est remplacé par $Z^{(1)}, Z^{(2)}, \cdots, Z^{(m)}$. Notons cette nouvelle matrice Z dont les colonnes représentent les variables normalisées :

$$Z = \begin{bmatrix} z_1^{(1)} & z_2^{(1)} & z_3^{(1)} & \dots & z_n^{(1)} \\ z_1^{(2)} & z_2^{(2)} & z_3^{(2)} & \dots & z_n^{(2)} \\ z_1^{(3)} & z_2^{(3)} & z_3^{(3)} & \dots & z_n^{(4)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ z_1^{(m)} & z_2^{(m)} & z_3^{(m)} & \dots & z_n^{(m)} \end{bmatrix}. \tag{1.5}$$

2. La deuxième étape est le calcul de la matrice des covariances. Notons par Z_i le vecteur contenant la i^e variable de chaque donnée. La matrice des covariances permet de vérifier la corrélation entre deux variables Z_i et Z_j. La matrice est construite comme suit :

$$Cov(Z) = \begin{bmatrix} Cov(Z_1,Z_1) & Cov(Z_1,Z_2) & Cov(Z_1,Z_3) & \dots & Cov(Z_1,Z_n) \\ Cov(Z_2,Z_1) & Cov(Z_2,Z_2) & Cov(Z_2,Z_3) & \dots & Cov(Z_2,Z_n) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ Cov(Z_n,Z_1) & Cov(Z_n,Z_2) & Cov(Z_n,Z_3) & \dots & Cov(Z_n,Z_n) \end{bmatrix}. \tag{1.6}$$

La covariance est donnée par $Cov(Z_i,Z_j)$ tel que :

$$Cov(Z_i,Z_j) = \frac{1}{m}\sum_k (z_i^{(k)} - \mu_{Z_i})(z_j^{(k)} - \mu_{Z_j}), \tag{1.7}$$

où μ_{Z_i} et μ_{Z_j} sont les moyennes respectives de Z_i et Z_j. Puisque les données sont normalisées, les moyennes sont de 0 et l'équation 1.7 devient

$$Cov(Z_i,Z_j) = \frac{1}{m}\sum_k z_i^{(k)} z_j^{(k)}. \tag{1.8}$$

D'après l'équation 1.7, nous remarquons que $Cov(Z_i,Z_j) = Cov(Z_j,Z_i)$, ce qui implique que la matrice formée est symétrique. Notons que $Cov(Z_i,Z_j)$ est positif si Z_i et Z_j sont

corrélés dans le même sens et négatif s'ils sont corrélés dans le sens inverse.

À une constante près, le calcul de la matrice des covariances revient au produit de matrice suivant :

$$Cov(Z) \approx Z^T Z$$

3. Toute matrice carrée symétrique possède des valeurs propres réelles. Nous n'entrons pas dans le détail des calculs des valeurs et vecteurs propres. Nous pouvons décomposer la matrice $Cov(Z)$ comme un produit de matrices $Cov(Z) = PDP^{-1}$ où D est une matrice diagonale dont les valeurs propres se retrouvent sur la diagonale principale et P est la matrice dont les colonnes sont les vecteurs propres associés aux valeurs propres. Nous obtenons un nombre de valeurs propres n équivalent au nombre de variables. Nous notons par $\overrightarrow{v_i}$ les vecteurs propres associés aux valeurs propres λ_i. Chaque valeur propre λ_i explique un pourcentage de la variance correspondant à l'équation 1.9.

$$\text{pourcentage de variance}(\lambda_i) = \frac{\lambda_i}{\lambda_1 + \lambda_2 + \cdots + \lambda_n}. \tag{1.9}$$

Une plus grande valeur propre explique donc un plus grand pourcentage de la variance. Son vecteur propre associé, une des composantes principales, indique donc la direction dans laquelle s'explique le plus grand pourcentage de la variance. Pour cette raison, nous pouvons réorganiser la matrice P afin d'avoir le vecteur propre associé à la plus grande valeur propre sur la première colonne, le vecteur propre associée à la deuxième plus grande valeur propre sur la deuxième colonne et ainsi de suite jusqu'au vecteur propre associé à la plus petite valeur propre sur la dernière colonne. Nous notons cette matrice P^*.

Puisque les dernières colonnes de P^* sont celles expliquant le moins les données, nous

pouvons les laisser tomber. Il y a plusieurs façons de déterminer le nombre de colonnes à conserver. Nous pouvons utiliser une méthode exhaustive en faisant des tests en enlevant à chaque fois une colonne jusqu'à en garder une seule. Nous pouvons aussi déterminer un pourcentage seuil d'explication de la variance et garder les principales composantes associées. Pour ce faire, nous reprenons l'équation 1.9 et additionnons, en partant de la plus grande valeur propre, autant de pourcentage de variance qu'il faut pour atteindre le seuil voulu.

Par exemple, supposons que nous avons les valeurs propres suivantes : $\lambda_1 = 14$, $\lambda_2 = 9$, $\lambda_3 = 2$ et $\lambda_4 = 1$. Supposons que nous voulions utiliser les composantes principales qui expliquent 85% de la variance. La première composante à elle seule ne peut expliquer 85% de la variance puisque le

$$\text{pourcentage de variance}(\lambda_1) = \frac{14}{14+9+2+1} = \frac{14}{26} \approx 54\%.$$

Nous devons donc ajouter la deuxième composante et vérifier si nous atteignons le pourcentage avec celle-ci :

$$\text{pourcentage de variance}(\lambda_1 + \lambda_2) = \frac{14+9}{14+9+2+1} = \frac{23}{26} \approx 88\%.$$

Si l'objectif est d'expliquer 85% de la variance, nous pouvons utiliser les 2 premières composantes principales de la matrice P^* et laisser tomber les deux dernières colonnes.

4. La dernière étape consiste à multiplier cette matrice avec la matrice des données normalisée :

$$Z^* = ZP^* \tag{1.10}$$

Cette matrice représente la projection des données originales normalisée sur les compo-

santes principales.

1.2.3 CLASSIFICATION

La classification est la dernière étape du processus. Il existe de nombreux classificateurs et nous ne procédons pas à une vérification exhaustive pour savoir lequel performe mieux avec nos données. Nous en sélectionnons quelques-uns dont nous connaissons l'efficacité avec des données similaires. Certaines données se prêtent mieux à une catégorie des classificateurs. Par exemple, si les données sont linéairement séparables, un modèle linéaire pourrait suffire. Cependant, si les données sont complexes, il sera peut-être plus efficace d'utiliser les **CNN**.

Dans ce mémoire, il est question de classification binaire puisque le classificateur peut donner deux réponses possibles : reine ou pas reine. La classification multiclasse s'applique lorsqu'il y a plus de deux classes. La classification binaire avec le classificateur des k plus proches voisins (**k-NN**, *k-Nearest Neighbors*) est une technique populaire (Djouadi et Bouktache (1997), Bhatia et Vandana (2010)). Pour l'utiliser, il faut définir une distance entre deux entrées. La distance euclidienne est souvent utilisée. Si nous voulons connaître la prédiction de la classe d'une entrée de l'ensemble de validation, nous cherchons les k données de l'ensemble d'entraînement le plus près. La prédiction devient la modalité la plus populaire, obtenue par un vote à majorité, parmi les étiquettes de ces k données.

Il existe aussi la classification par les forêts d'arbres décisionnels (*Random Forest*) présentée par Breiman (2001), la classification naïve bayésienne, la classification par régression logistique et bien d'autres (Kotsiantis et al. (2007)). Les deux méthodes utilisées dans ce mémoire sont les **SVM** et les **CNN**. La classification par un **SVM** est décrite dans cette section. Pour introduire les **CNN** qui sont présentés à la section 1.3, les algorithmes du perceptron et les **MLP** sont décrits.

Machine à vecteurs de support

Puisque l'un des objectifs de ce mémoire est de faire la classification d'images appartenant à deux classes, nous allons nous attarder sur les **SVM** binaires, c'est-à-dire les **binary-SVM** (Goh et al. (2001)). Cette technique optimise les paramètres d'un hyperplan qui sépare des données appartenant à deux classes. La figure 1.7 permet de représenter la situation simplifiée où les éléments ont seulement deux dimensions. Pour cet exemple, l'hyperplan recherché correspond à une droite, en rouge sur la figure. Les vecteurs de support sont les points marqués en rouge.

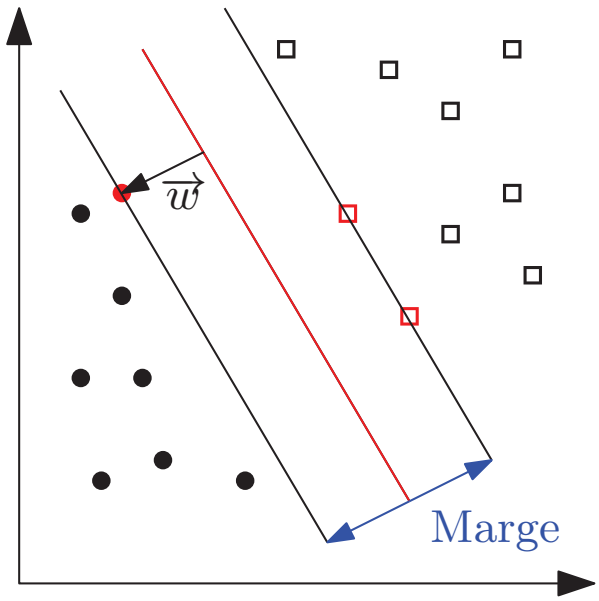


FIGURE 1.7 – Représentation en deux dimensions d'un problème linéairement séparable. En rouge, les points et les carrés représentent les supports de vecteurs tandis que la droite représente l'hyperplan. En bleu, la marge à maximiser.

L'objectif est de trouver l'hyperplan qui maximise la marge entre l'hyperplan et les vecteurs de support. Ainsi, un nouvel élément a plus de chance d'être classé dans la bonne catégorie. Chaque point représente un élément de l'ensemble d'entraînement. Pour un modèle qui prend en entrée une image, la dimension correspondra au nombre de pixels de l'image. Si des étapes

d'extraction de caractéristiques et de réduction de la dimensionnalité ont été effectuées, ce sera la sortie de la réduction de la dimensionalité qui sera utilisée en entrée du **SVM**.

Soit l'ensemble d'entraînement $[\overrightarrow{x_1}, \overrightarrow{x_2}, \overrightarrow{x_3}, \cdots, \overrightarrow{x_n}]$ et les classes associées $[y_1, y_2, y_3, \cdots, y_n]$ où $y_i = \pm 1$. Dans notre cas, $y_i = 1$ correspond à une image de reine et $y_i = -1$ à une image sans reine. L'hyperplan recherché peut être décrit, à l'aide d'un produit scalaire, comme suit :

$$\overrightarrow{w} \cdot \overrightarrow{x} - b = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n - b = 0 \tag{1.11}$$

où $\mathbf{w} = w_1, w_2, \cdots w_n$ est le vecteur des poids et b est l'ordonnée lorsque les $x_i = 0$. Après normalisation des données, nous pouvons poser $\overrightarrow{w} \cdot \overrightarrow{x} - b = 1$ pour l'hyperplan supporté par un point de la classe associé à 1 et $\overrightarrow{w} \cdot \overrightarrow{x} - b = 1$ pour celui associé à -1.

Portons notre attention à la norme de la marge que nous voulons maximiser. Nous voulons donc trouver la distance entre l'hyperplan recherché et l'hyperplan parallèle à celui-ci qui passe par un vecteur de support. Pour ce faire, nous prenons $\overrightarrow{x}$ un vecteur qui se trouve sur l'hyperplan recherché et $\frac{\overrightarrow{w}}{\|\overrightarrow{w}\|}$ un vecteur unitaire de même direction que $\overrightarrow{w}$. Nous nous demandons de combien nous devons déplacer $\overrightarrow{x}$ dans la direction de $\overrightarrow{w}$ pour atteindre l'hyperplan $\overrightarrow{w} \cdot \overrightarrow{x} - b = 1$. Cela se traduit par :

$$\overrightarrow{w} \cdot \left(\overrightarrow{x} + k_1 \frac{\overrightarrow{w}}{\|\overrightarrow{w}\|} \right) - b = 1 \tag{1.12}$$

où k_1 est la distance recherchée. En développant l'équation précédente, nous obtenons :

$$\overrightarrow{w} \cdot \overrightarrow{x} + \overrightarrow{w} \cdot k_1 \frac{\overrightarrow{w}}{\|\overrightarrow{w}\|} - b = 1$$

$$\overrightarrow{w} \cdot \overrightarrow{\mathrm{x}} + k_1 \frac{\overrightarrow{w} \cdot \overrightarrow{w}}{\| \overrightarrow{w} \|} - b = 1$$

$$\overrightarrow{w} \cdot \overrightarrow{\mathrm{x}} - b + k_1 \frac{\| \overrightarrow{w} \|^2}{\| \overrightarrow{w} \|} = 1$$

$$0 + k_1 \| \overrightarrow{w} \| = 1$$

$$k_1 = \frac{1}{\| \overrightarrow{w} \|}.$$

De façon analogue, nous trouvons que la distance k_2 est la même entre l'hyperplan recherché et $\overrightarrow{w} \cdot \overrightarrow{x} - b = 1$. Ainsi, la marge totale est de

$$k = k_1 + k_2 = \frac{2}{\| \overrightarrow{w} \|}. \tag{1.13}$$

Si nous voulons maximiser k, il faut minimiser $\| \overrightarrow{w} \|$.

Pour l'hyperplan trouvé, la règle de décision pour un nouvel élément $\overrightarrow{\mathrm{x}_j}$ est donnée à l'équation 1.14.

$$y_j = \begin{cases} 1 & \text{si } \overrightarrow{w} \cdot \mathrm{x} - b > 0 \\ -1 & \text{si } \overrightarrow{w} \cdot \mathrm{x} - b < 0 \end{cases} \tag{1.14}$$

Nous venons de décrire les détails pour un problème linéairement séparable. Lorsque nos données ne permettent pas une séparation par un hyperplan linéaire, il faut utiliser un *kernel*

(Hofmann et al. (2008)). Nous remplaçon le produit scalaire par une autre fonction. Nous appelons cette technique l'astuce du noyau. Celle utilisée dans ce mémoire est la fonction à base radiale (**RBF**, *Radial Basis Function*) (Orr et others (1996)). Dans ce cas, le produit scalaire d'un vecteur x par un vecteur y est remplacé par ce qui suit :

$$e^{-\gamma(x-y)^2}.$$

Perceptron

Un réseau de neurones tient son origine des connexions entre les neurones du cerveau. Le perceptron est un réseau de neurones à son plus simple : il contient un seul neurone (Kotsiantis et al. (2007)). Pour une entrée $\mathbf{x} = x_1, x_2, x_3 \cdots x_n$, nous associons une valeur de sortie y égale à 0 ou à 1. Soit les poids $\mathbf{w} = w_1, w_2, w_3 \cdots w_n$ et le biais b, appelés paramètres de l'algorithme du perceptron. L'objectif est d'optimiser ces paramètres. Cela revient à optimiser les paramètres d'un hyperplan qui sépare les éléments de la classe 0 de ceux de la classe 1. Nous construisons un modèle qui prédit $\hat{y}$, la valeur de sortie pour une entrée donnée jamais présentée au modèle. Nous effectuons le produit scalaire de l'entrée $\mathbf{x}$ avec les poids $\mathbf{w}$ auquel nous ajoutons le biais. Appelons ce calcul l'entrée du neurone.

$$z = x_1 \cdot w_1 + x_2 \cdot w_2 + x_3 \cdot w_3 + \cdots + x_n \cdot w_n + b = \sum x_i \cdot w_i + b = \mathbf{x} \cdot \mathbf{w} + b \tag{1.15}$$

Pour simplifier la notation, tel qu'illustré à la figure 1.8, nous pouvons ajouter une entrée de valeur 1 dont le poids associé est b.

L'équation 1.15 devient simplement $z = \mathbf{x} \cdot \mathbf{w}$. Si la combinaison z est plus grande ou égale à zéro, nous prédisons 1. Sinon, nous prédisons 0. Nous pouvons ensuite comparer la prédiction

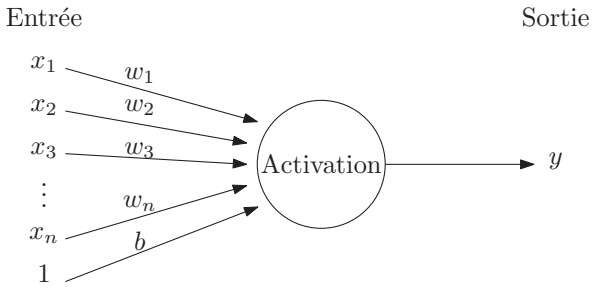


FIGURE 1.8 – Représentation graphique du perceptron.

$\hat{y}$ de l'équation 1.16 avec la valeur attendue y. Si le modèle fait la bonne prédiction, nous ne modifions pas le modèle. Sinon, il faut corriger les paramètres du modèle.

$$\hat{y} = \begin{cases} 1 & \text{si } z \geq 0 \\ 0 & \text{sinon.} \end{cases} \tag{1.16}$$

Si la valeur attendue est $y = 1$, mais que le modèle prédit $\hat{y} = 0$, il faut corriger les poids afin de faire augmenter la valeur de la prédiction et ainsi se rapprocher de la valeur réelle. Nous utilisons la règle d'apprentissage suivante :

$$w_i = w_i + \alpha(y - \hat{y}_i)x_i \tag{1.17}$$

où α est le taux d'apprentissage déterminé de façon arbitraire. Vérifions la règle avec le premier cas : si $y = 1$ et $\hat{y} = 0$, alors $(y - \hat{y}) = 1$ et nous obtenons $w_i = w_i + \alpha \cdot x_i$. Si l'entrée x_i est positive, le poids w_i est revu à la hausse pour participer à l'augmentation de la valeur de $\hat{y}$. Si l'entrée est négative, nous diminuons le poids w_i et cela contribuera aussi à faire augmenter la valeur de la prédiction.

Ce qui vient d'être présenté était le passage d'une seule entrée et l'ajustement des poids à partir de l'erreur commise. Nous répétons ce processus avec l'ensemble d'entraînement et nous

pouvons vérifier l'efficacité du modèle avec l'ensemble de validation. Nous recommençons ce processus jusqu'à ce qu'un critère d'arrêt soit atteint.

Perceptron multicouche

Le **MLP** est une méthode d'apprentissage supervisée qui est une version plus large du perceptron. C'est un réseau de neurones à propagation directe (*Feedfoward neural network*) constitué d'une couche d'entrée, d'une ou de plusieurs couches cachées et d'une couche de sortie. Une couche a des neurones et des liens relient ces neurones à ceux de la couche suivante. C'est ce type de réseau de neurones qui est décrit dans cette section. Contrairement aux réseaux de neurones récurrents, il ne permet pas de retour à une couche antérieure (Abraham (2005)). Comme illustré à la figure 1.9, l'objectif est de simuler une fonction f qui, pour une entrée x, retourne y. Un réseau de neurones est simple lorsqu'il a une seule couche cachée. Nous allons le qualifier de profond lorsque nous retrouvons plus d'une couche cachée. La couche de sortie peut comporter un seul ou plusieurs neurones.

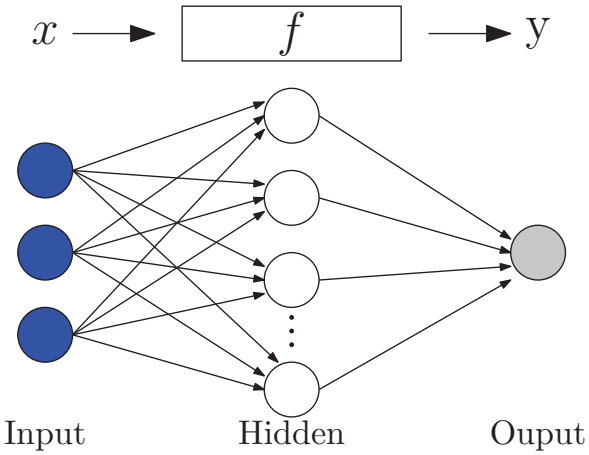


FIGURE 1.9 – Un réseau de neurones à propagation directe simple, c'est-à-dire qui comprend une seule couche cachée. Ce réseau comporte 3 neurones d'entrée et un seul neurone de sortie.

Les liens entre les neurones, représentés par des flèches sur la figure 1.9, ont des valeurs

appelées *poids*, qui sont des paramètres ajustables lors de l'entraînement. Tout comme le perceptron, un biais est ajouté, lui aussi ajustable. Pour ajuster les paramètres, nous procédons en trois phases, qui sont décrites ci-dessous, comme avec le perceptron. Une image traverse d'abord le réseau de la couche d'entrée jusqu'à la sortie. La couche de sortie fait la prédiction de ce qui a été donné en entrée. Nous calculons ensuite l'erreur commise sur la prédiction par rapport à ce qui était attendu. Finalement, nous propageons l'erreur dans le réseau de neurones de la couche de sortie jusqu'à la première couche cachée. Cette dernière phase s'appelle la rétropropagation et permet d'ajuster les différents paramètres. Ces phases sont répétées jusqu'à ce qu'un critère d'arrêt soit atteint. Tout au début de l'entraînement, une valeur arbitraire est assignée aux poids et biais et chaque itération permet de changer ces valeurs et d'espérer améliorer la perte. Voici les trois phases mentionnées : la propagation avant, le calcul de la perte et la rétropropagation.

Propagation avant Lors de cette phase, chaque neurone est activé selon une *fonction d'activation*, $g(\cdot)$, tel qu'illustré à la figure 1.10. Sans fonction d'activation, le réseau de neurones serait linéaire et permettrait de résoudre des problèmes linéaires. Pour ajouter de la complexité au réseau, nous utilisons une fonction non linéaire qui permet d'activer ou non un neurone.

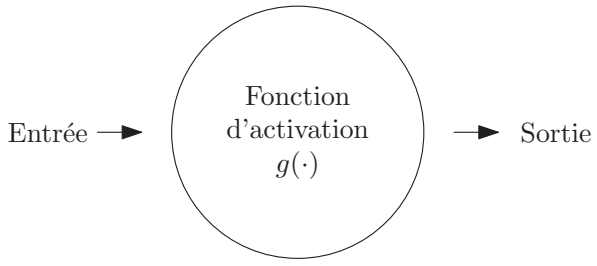


FIGURE 1.10 – Chaque neurone prend une valeur en entrée, applique la fonction d'activation et retourne une valeur de sortie.

Notons en exposant la couche dans laquelle le neurone se retrouve. L'entrée $z_j^{(l+1)}$ du neurone $a_j^{(l+1)}$ est le produit scalaire des valeurs de sortie des i neurones de la couche

précédente, $a_i^{(l)}$, par les poids $w_{i,j}$ des liens qui sont connectés :

$$z_j^{(l+1)} = \sum a_i^{(l)} \cdot w_{i,j}. \tag{1.18}$$

Au côté droit de l'équation 1.18, nous pourrions ajouter un biais b comme dans l'algorithme du perceptron. Nous l'omettons ici pour simplifier la présentation des calculs. Pour obtenir la valeur de sortie du neurone, nous appliquons la fonction d'activation à l'entrée du neurone tel qu'illustré à la figure 1.11, c'est-à-dire :

$$a_j^{(l+1)} = g\left(z_j^{(l+1)}\right). \tag{1.19}$$

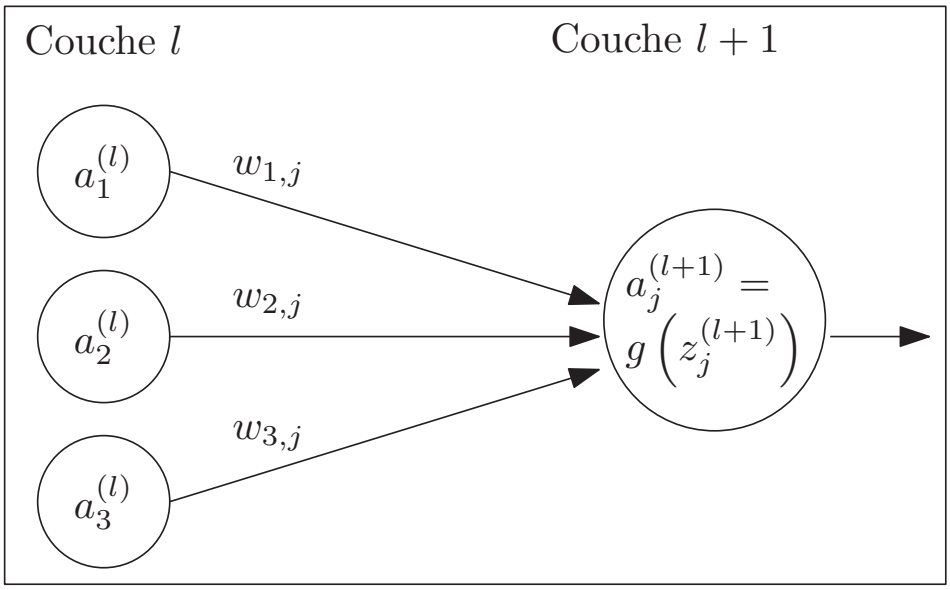


FIGURE 1.11 – L'entrée du neurone $a_j^{(l+1)}$ correspond au produit scalaire des neurones qui y sont connectés par les poids des liens qui les relient. Sa sortie est l'application de la fonction d'activation à la valeur d'entrée.

Nous effectuons d'abord ce calcul pour chaque neurone de la première couche cachée à partir des neurones de la couche d'entrée. Pour calculer la valeur de sortie des neurones de la deuxième couche cachée, nous utilisons à nouveau l'équation 1.19 avec la sortie

des neurones de la première couche et les poids des liens qui les relient la première couche à la deuxième couche. Nous reproduisons ces calculs jusqu'à la couche de sortie du réseau.

Attardons-nous maintenant aux fonctions d'activation $g(\cdot)$ possibles. Plusieurs fonctions existent et quelques-unes d'entre elles sont présentées à la figure 1.12. La sigmoïde et la tangente hyperbolique sont des fonctions d'activation classiques. Elles ont des asymptotes horizontales qui permettent de contrôler la valeur de sortie entre 0 et 1 pour la sigmoïde et entre -1 et 1 pour la tangente hyperbolique. La fonction $\tanh(x)$ possède l'avantage d'être symétrique par rapport à l'origine, ce qui permet de retourner une sortie qui est en moyenne près de 0 (LeCun et al. (2012)). Ce type de fonction fait apparaître le problème de la disparition du gradient (*vanishing gradient problem*) : le gradient tend vers 0 lorsque l'entrée de la fonction d'activation tend vers $-\infty$ ou ∞ et peut ainsi arrêter l'apprentissage (Witten et al. (2017)).

Pour contrer ce problème, nous pouvons utiliser une fonction dont la dérivée ne tend pas vers 0 lorsque l'entrée tend vers l'infini. L'une des plus populaires est la fonction *ReLU* (Rectify Linear Units). Cette fonction n'active pas le neurone lorsque la valeur en entrée est inférieure à 0. Le neurone aura donc 0 en sortie. Elle conserve la valeur de l'entrée sinon. La dérivée de cette fonction est nulle lorsque l'entrée est négative et est égale à 1 sinon. Bien qu'elle s'adapte à plusieurs situations, notons que cette fonction n'est pas sans problème : elle n'est pas différentiable en 0. Il existe certaines variantes à cette fonction, par exemple *Leaky ReLU* (Glorot et al. (2011), Apicella et al. (2021)).

En partant de la couche d'entrée, les neurones ont tous été activés jusqu'à la couche de sortie. Cette dernière couche utilise souvent une fonction d'activation comme la sigmoïde qui retourne une valeur entre 0 et 1, afin d'obtenir la prédiction $\hat{y}$ du modèle que nous pouvons comparer à la valeur réelle y.

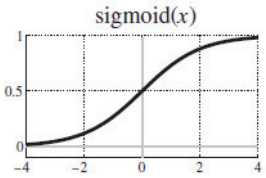
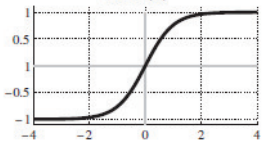
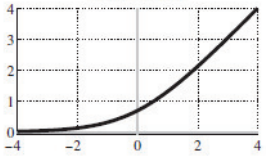
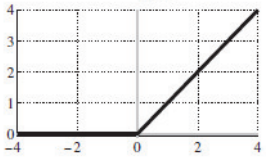
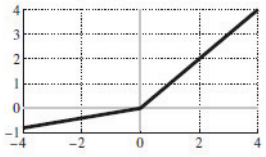
Name and Graph	Function	Derivative
sigmoid(x) 	$h(x) = \frac{1}{1+\exp(-x)}$	$h'(x) = h(x)[1-h(x)]$
tanh(x) 	$h(x) = \frac{\exp(x)-\exp(-x)}{\exp(x)+\exp(-x)}$	$h'(x) = 1-h(x)^2$
softplus(x) 	$h(x) = \log(1+\exp(x))$	$h'(x) = \frac{1}{1+\exp(-x)}$
rectify(x) 	$h(x) = \max(0,x)$	$h'(x) = \begin{cases} 1 \text{ if } x \geq 0 \\ 0 \text{ if } x < 0 \end{cases}$
pw_linear(x) 	$h(x) = \begin{cases} x \text{ if } x \geq 0 \\ ax \text{ if } x < 0 \end{cases}$	$h'(x) = \begin{cases} 1 \text{ if } x \geq 0 \\ a \text{ if } x < 0 \end{cases}$

FIGURE 1.12 – Différentes fonctions d'activation représentées graphiquement et leur dérivée. Source : Witten et al. (2017)

Calcul de la perte Lorsque chaque neurone a été activé, nous comparons la prédiction $\hat{y}$ avec la vraie valeur y. Nous notons la fonction de perte $Loss(\hat{y}, y)$. Il existe plusieurs fonctions de perte (Janocha et Czarnecki (2017)). Parmi elles, nous retrouvons la **MSE** (*Mean Squared Error*) qui prend le carré de la différence entre la prédiction $\hat{y}$ et la valeur réelle y. Dans le cas de classification binaire, le modèle prédit que x appartient à la première classe avec une probabilité $\hat{y}$ et à la deuxième classe avec une probabilité $(1-\hat{y})$. La fonction de perte généralement utilisée dans ce cas est la *Binary Cross Entropy* (Buja

et al. (2005)) :

$$Loss(y,\hat{y}) = -y\log\hat{y} - (1-y)\log(1-\hat{y}), \tag{1.20}$$

où y peut prendre les valeurs 0 ou 1, correspondant aux deux classes. Ainsi, nous pouvons réécrire l'équation 1.20 par partie comme à l'équation 1.21.

$$Loss(y,\hat{y}) = \begin{cases} -\log(\hat{y}) & \text{si } y = 1 \\ -\log(1-\hat{y}) & \text{si } y = 0 \end{cases} \tag{1.21}$$

Nous remarquons avec la figure 1.13 que lorsque $y = 1$, le gradient est négatif et pour minimiser la perte, il faut se déplacer vers la droite. Lorsque $y = 0$, le gradient est positif et il faut se déplacer vers la gauche pour se rapprocher de la valeur réelle. Nous remarquons donc qu'il faut se diriger dans le sens contraire du gradient pour minimiser la fonction de perte. Comme nous le verrons à l'étape suivante, nous commençons par calculer le gradient de la perte du neurone de sortie. Nous devons ensuite propager cette perte dans le réseau afin de déterminer à quel point chaque partie du réseau est responsable de cette perte et faire les ajustements des paramètres en conséquence.

Rétropropagation Dans cette dernière phase, les paramètres $w_{i,j}$ sont ajustés en utilisant une technique appelée la *descente du gradient* (Ruder (2016)). Nous ajustons $w_{i,j}$ qui relie le neurone $a_i^{(l)}$ au neurone $a_j^{(l+1)}$ en se dirigeant dans le sens inverse du gradient selon un facteur α appelé taux d'apprentissage. Ceci peut se traduire par l'équation suivante, que nous nommons règle d'apprentissage :

$$w_{i,j} = w_{i,j} - \alpha\frac{\partial Loss}{\partial w_{i,j}} \tag{1.22}$$

où $\dfrac{\partial Loss}{\partial w_{i,j}}$ représente l'impact de $w_{i,j}$ sur l'erreur totale. Nous utilisons la règle de

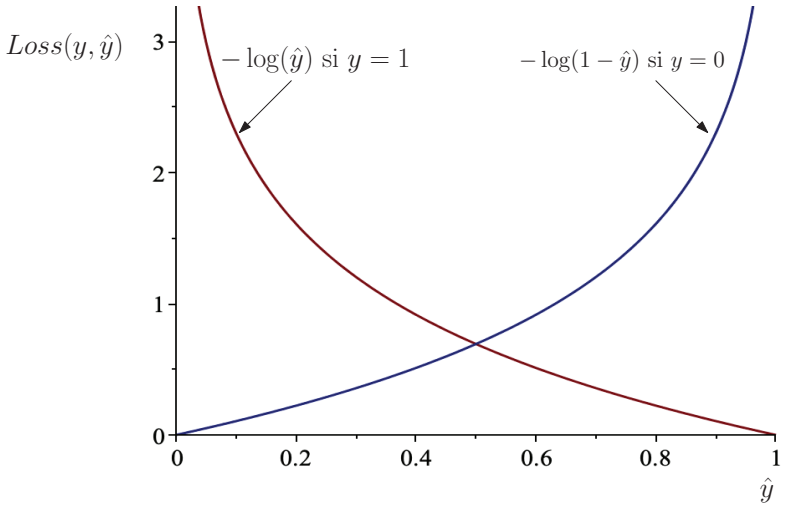


FIGURE 1.13 – Représentation graphique de la fonction de perte *Binary Cross Entropy*.

dérivation en chaîne pour développer cette dérivée. Ainsi, nous obtenons :

$$\frac{\partial Loss}{\partial w_{i,j}} = \frac{\partial Loss}{\partial a_j^{(l+1)}} \cdot \frac{\partial a_j^{(l+1)}}{\partial z_j^{(l+1)}} \cdot \frac{\partial z_j^{(l+1)}}{\partial w_{i,j}}. \tag{1.23}$$

Analysons les trois dérivées partielles de l'équation 1.23. Un support graphique est offert à la figure 1.14.

1. $\frac{\partial Loss}{\partial a_j^{(l+1)}}$ représente l'impact du neurone $a_j^{(l+1)}$ sur la perte. En utilisant l'équation 1.19, cette dérivée peut aussi s'écrire comme $\frac{\partial Loss}{\partial g\left(z_j^{(l+1)}\right)}$. La perte pour un neurone de la couche $l+1$ se calcule à partir des k neurones de la couche $l+2$. Nous introduisons les termes $z_k^{(l+2)}$ en utilisant de nouveau la règle de dérivation en chaîne :

$$\frac{\partial Loss}{\partial g\left(z_j^{(l+1)}\right)} = \sum_k \frac{\partial Loss}{\partial z_k^{(l+2)}} \cdot \frac{\partial z_k^{(l+2)}}{\partial g\left(z_j^{(l+1)}\right)}. \tag{1.24}$$

Concentrons-nous d'abord sur la dernière dérivée partielle de 1.24. Puisque $z_k^{(l+2)} = \sum_i w_{i,k} \cdot a_i^{(l+1)}$, le seul terme non nul de la somme lorsque nous déri-

vons par rapport à $g\left(z_j^{(l+1)}\right) = a_j^{(l+1)}$ est $\frac{\partial w_{j,k} \cdot a_j^{(l+1)}}{\partial a_j^{(l+1)}} = w_{j,k}$. Nous obtenons donc l'équation 1.25.

$$\frac{\partial Loss}{\partial g\left(z_j^{(l+1)}\right)} = \sum_k \frac{\partial Loss}{\partial z_k^{(l+2)}} \cdot w_{j,k} \tag{1.25}$$

2. $\frac{\partial a_j^{(l+1)}}{\partial z_j^{(l+1)}}$ représente la variabilité de $a_j^{(l+1)}$ lorsque $z_j^{(l+1)}$ varie. Puisque $\partial a_j^{(l+1)} = \partial g\left(z^{(l+1)}\right)$, nous obtenons l'équation suivante :

$$\frac{\partial a_j^{(l+1)}}{\partial z_j^{(l+1)}} = \frac{\partial g\left(z_j^{(l+1)}\right)}{\partial z_j^{(l+1)}} = g'\left(z_j^{(l+1)}\right). \tag{1.26}$$

3. $\frac{\partial z_j^{(l+1)}}{\partial w_{i,j}}$ représente l'impact de la variation du poids $w_{i,j}$ sur l'entrée $z_j^{(l+1)}$ du neurone $a_j^{(l+1)}$. Rappelons que $z_j^{(l+1)} = \sum_k a_k^{(l)} \cdot w_{k,j}$ où k est le nombre de neurones connectés à $z_j^{(l+1)}$. La seule dérivée partielle non nulle de cette somme, obtenue lorsque $k = i$, est $\frac{\partial a_i^{(l)} \cdot w_{i,j}}{\partial w_{i,j}} = a_i^{(l)}$. Ainsi,

$$\frac{\partial z_j^{(l+1)}}{\partial w_{i,j}} = a_i^{(l)}. \tag{1.27}$$

Revenons à l'équation 1.22. En rassemblant les équations 1.25, 1.26 et 1.27. Nous pouvons remplacer l'équation 1.22 par l'équation 1.28

$$w_{i,j} = w_{i,j} - \alpha \cdot a_i^{(l)} \cdot g'\left(z_j^{(l+1)}\right) \cdot \sum_k \left(\frac{\partial Loss}{\partial z_k^{(l+2)}} \cdot w_{j,k}\right) \tag{1.28}$$

Cette équation permet de modifier les poids $w_{i,j}$. La modification des paramètres $w_{i,j}$ se nomme la *descente du gradient*. On peut d'abord, pour chaque neurone $a_j^{(l+1)}$ en

partant de la couche de sortie, calculer $\frac{\partial Loss}{\partial z_j^{(l+1)}}$. Nous appelons cette étape la *propagation du gradient*.

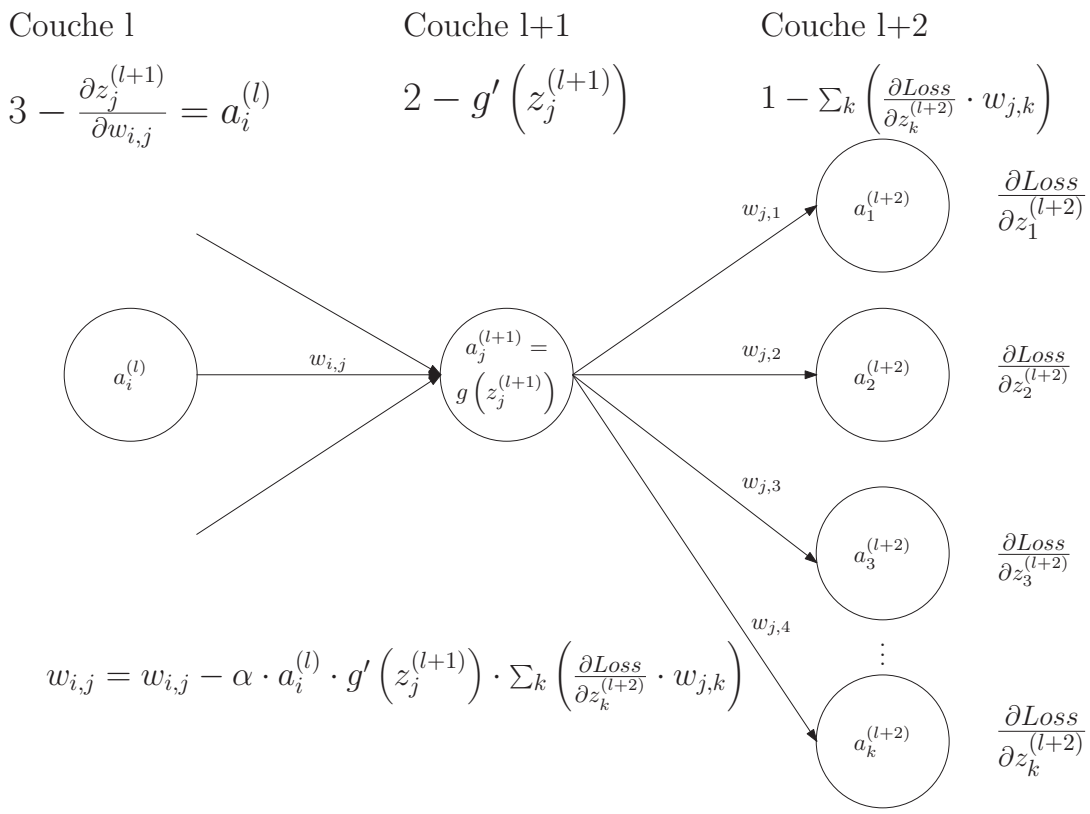


FIGURE 1.14 – Règle d'apprentissage pour la mise à jour des paramètres $w_{i,j}$.

Nous venons de voir quelques méthodes d'apprentissage classique. La dernière, utilisant un **MLP**, se retrouve dans la dernière partie d'un **CNN**, que nous verrons dans la section suivante.

1.3 APPRENTISSAGE AVEC RÉSEAU DE NEURONES À CONVOLUTION

Les méthodes classiques requièrent souvent l'utilisation de trois techniques : l'extraction des caractéristiques, la réduction de la dimensionnalité et la classification. Un des avantages des **CNN** est de pouvoir combiner ces trois étapes dans l'architecture du réseau (voir figure 1.15). En utilisant les méthodes classiques, la représentation est construite manuellement alors que dans les méthodes profondes, elles sont générées automatiquement par apprentissage.

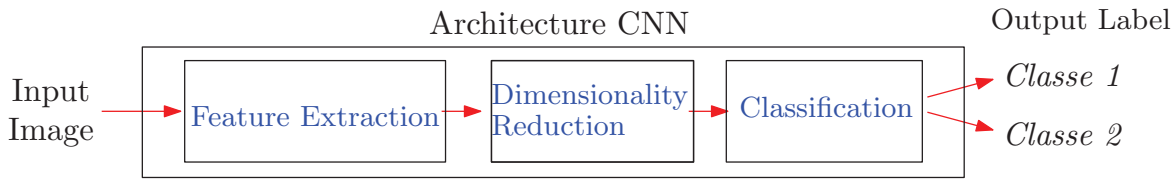


FIGURE 1.15 – Les trois étapes de certaines méthodes classiques (extraction des caractéristiques, réduction de la dimensionnalité et classification) sont regroupées.

Un **CNN**, comme le présentent LeCun et al. (2015), est un réseau de neurones qui est entre autres spécialisé dans la reconnaissance d'images. Il prend en entrée les pixels de l'image et donne en sortie les probabilités que l'image appartienne à une classe. Lorsque la classification est binaire, nous retrouvons en sortie un neurone qui donne la probabilité $\hat{y}$ d'appartenir à la classe y et donc la probabilité $1 - \hat{y}$ de ne pas appartenir à la classe y. Tel qu'illustré à la figure 1.16, nous pouvons séparer les **CNN** en deux parties. La première consiste en une succession de convolution et de *pooling*. Cela permet d'extraire des caractéristiques des images et réduire la dimensionnalité. La deuxième partie est celle où s'effectue la classification. Cette dernière partie est constituée de couches entièrement connectées.

Ce qui suit décrit les différentes couches d'un **CNN** : les couches de convolutions, les couches de *pooling* et finalement les couches entièrement connectées.

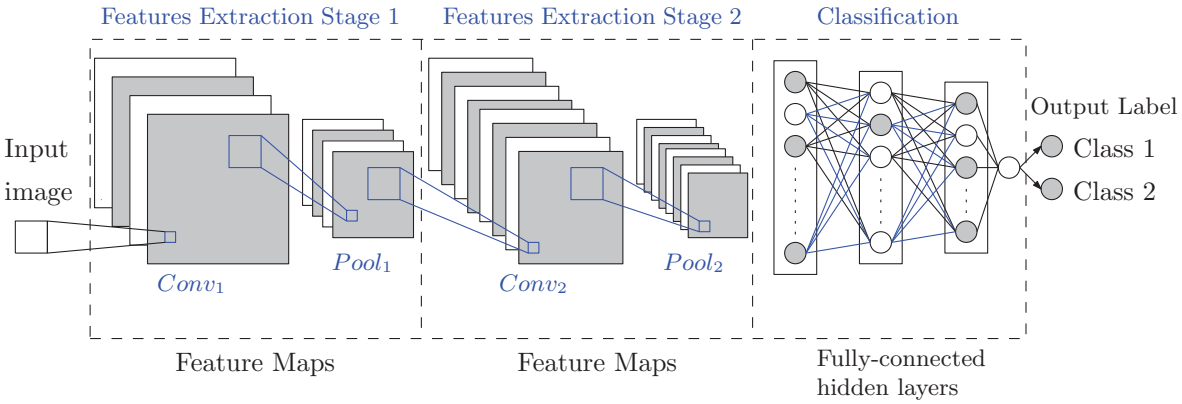


FIGURE 1.16 – Dans un CNN, nous retrouvons des étapes pour extraire les caractéristiques, composées de couches de convolution suivie d'une couche de *pooling*. Nous retrouvons ensuite des couches entièrement connectées qui permettent d'effectuer la classification. Cette méthode a été adoptée pour l'une des méthodes proposées du mémoire.

Couche de convolution

Une convolution sur une image permet de mettre en évidence des caractéristiques spécifiques comme des lignes et des courbes. Nous pouvons représenter une convolution par l'équation 1.29.

$$S(i,j) = I * K = \sum_m \sum_n I(i+m, j+n)K(m,n) \tag{1.29}$$

On appelle S la matrice qui représente l'image I dont les caractéristiques ont été extraites. En anglais, nous appelons cette matrice une *Feature Map*. i et j définissent la position du pixel. K est la matrice de convolution, appelée *Kernel* en anglais. m et n correspondent respectivement à la hauteur et à la largeur du *Kernel*. La figure 1.17 permet de visualiser l'équation 1.29. De plus, la figure 1.18 présente un exemple, dont les calculs ont été détaillés, d'une convolution sur une image. À la figure 1.19, le même filtre qu'à la figure 1.18 a été appliqué à une image de reine.

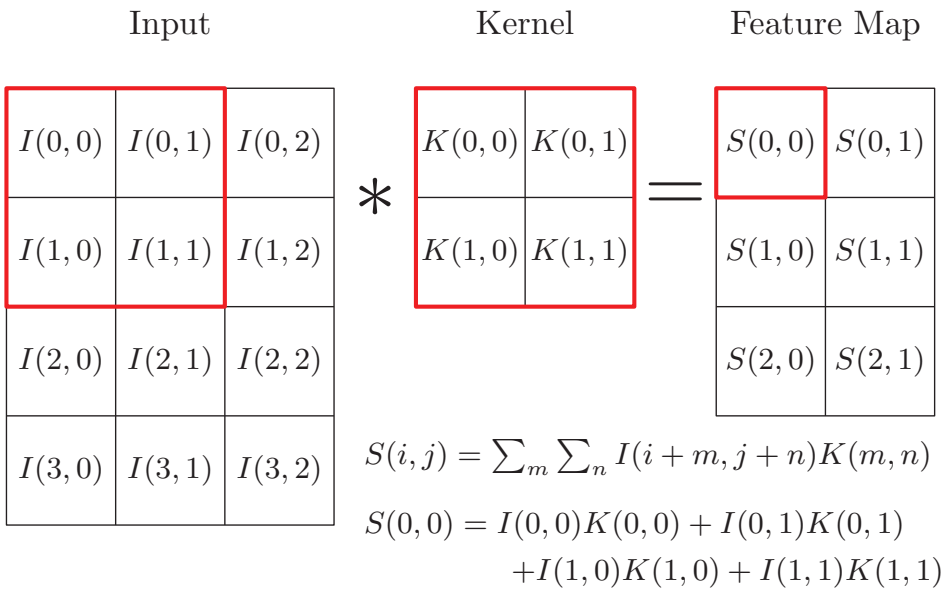


FIGURE 1.17 – Visualisation de l'équation 1.29.

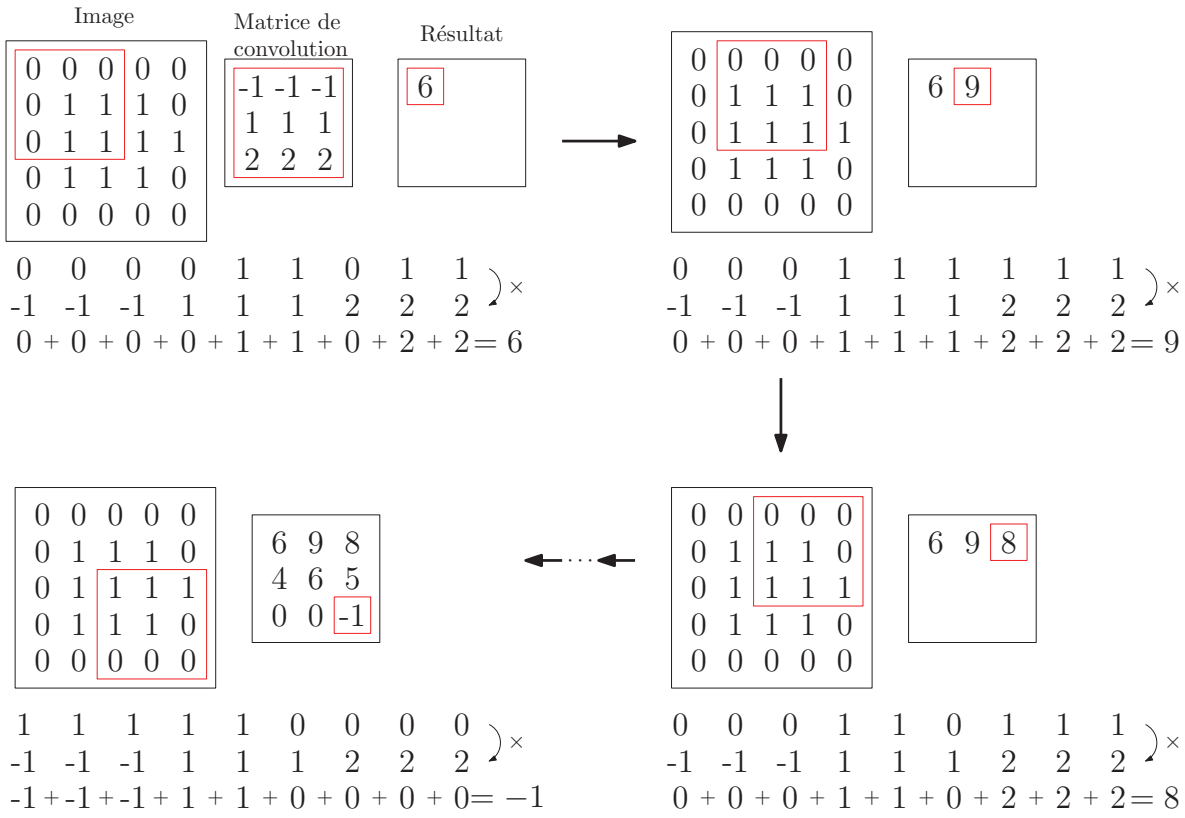


FIGURE 1.18 – Exemple d'une convolution sur une image.

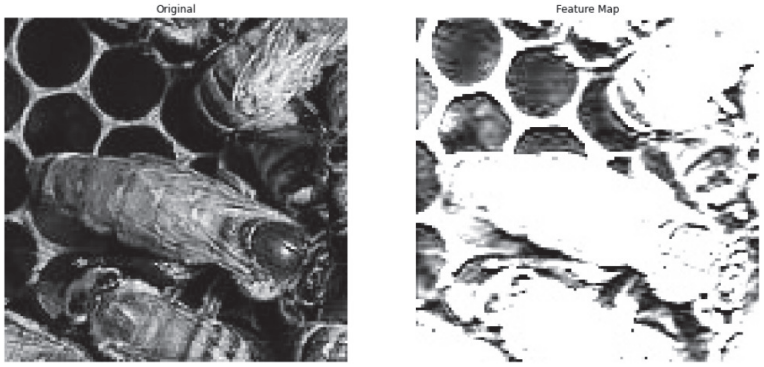


FIGURE 1.19 – Le filtre utilisé à la figure 1.18 appliqué à une image de reine. À gauche, l'image originale en ton de gris. À droite, la feature map résultante.

Goodfellow et al. (2016) décrit trois aspects qui peuvent améliorer l'apprentissage d'un **CNN** :

1. Le premier est ce qui est appelé *Sparse interactions*. Dans un **CNN** classique, chaque neurone d'une couche est relié par un poids à la couche précédente. Ces poids constituent les paramètres à modifier pendant l'entraînement. Pour les couches de convolutions d'un **CNN**, les matrices de convolutions sont les paramètres à optimiser lors de l'entraînement et sont de taille moins élevée que l'entrée d'une couche. Cela fait, pour les **CNN**, beaucoup moins de paramètres à garder en mémoire et accélère l'efficacité d'exécution.
2. Un autre aspect qui réduit la quantité de données gardées en mémoire est le *Parameter sharing*. Une même matrice de convolution est utilisée à plusieurs reprises sur différentes sous-sections de l'image en entrée. Nous réduisons encore une fois le nombre de paramètres à garder en mémoire et accélérons l'exécution
3. Le dernier aspect est *equivariance to translation*. L'équivariance est représentée par $f(g(x)) = g(f(x))$. Si la translation est effectuée avant ou après la convolution, il n'y aura pas d'impact sur la sortie.

À la fin de la convolution, nous appliquons une fonction d'activation à chaque élément de la convolution afin d'obtenir une sortie qui sera traitée par la couche de *pooling*.

Couche de *Pooling*

Cette couche suit une couche de convolution. Le *pooling* permet de réduire la dimensionnalité de ce qu'il reçoit en entrée et d'introduire une certaine invariance aux petites translations (Goodfellow et al. (2016)). Voici comment nous effectuons un *pooling*.

Nous devons choisir une fenêtre de dimension p par q sur laquelle le *pooling* aura lieu. Généralement, nous déplaçons la fenêtre de sorte qu'aucune sous-section ne se chevauche. Nous déplaçons donc la fenêtre horizontalement par bonds de p et verticalement par bonds de q pour couvrir entièrement la *feature map*. Chaque fenêtre est remplacée par un sommaire statistique des pixels s'y retrouvant : la moyenne des pixels, le pixel minimal, le pixel maximal, etc. Ainsi, nous obtenons en sortie de la couche de *pooling* une nouvelle matrice de taille inférieure à la *feature map* en entrée. Cette sortie peut soit servir d'entrée pour une nouvelle couche de convolution, soit pour le début de la classification du modèle.

À la figure 1.20, nous retrouvons un exemple d'un *max-pooling* et d'un *average-pooling*. Prendre la moyenne des pixels de la fenêtre a tendance à atténuer la variation de pixels dans la *feature map*. Prendre la valeur maximale a pour effet d'accentuer les contrastes puisque nous sélectionnons le pixel le plus clair pour remplacer la sous-section. Pour ce mémoire, nous utilisons le *max-pooling* pour accentuer les contours des abeilles.

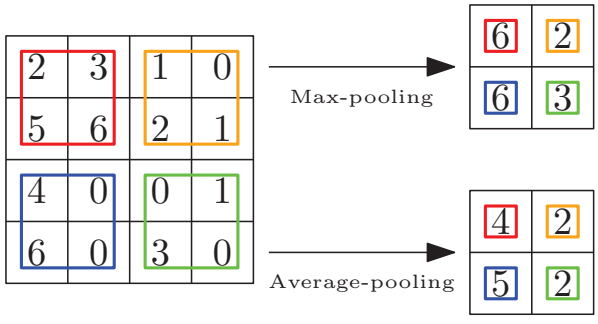


FIGURE 1.20 – Résultat d'un *max-pooling* et d'un *average-pooling* à partir de la même matrice, avec un glissement de la taille de la fenêtre.

On présente à la figure 1.21 un exemple sur une image en ton de gris. Une convolution y est d'abord appliquée. Le filtre suivant a été utilisé.

$$Conv = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

Cette matrice de convolution permet de faire ressortir les contours d'une image. L'opération de *max-pooling* est ensuite effectuée avec une fenêtre de 2×2 pixels et un glissement de la taille de la fenêtre. Nous remarquons que l'opération de *pooling* fait réduire de moitié la taille de l'image originale.

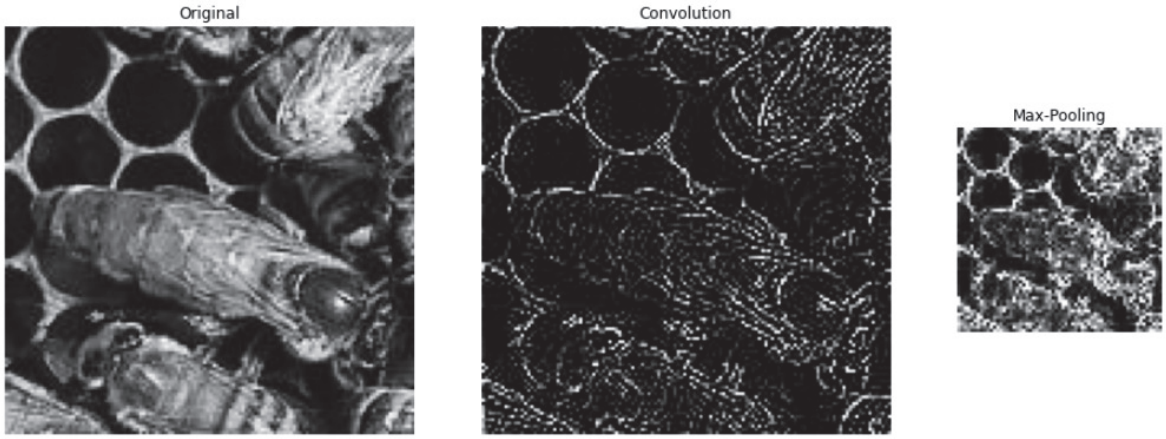


FIGURE 1.21 – À gauche, l'image originale en ton de gris. Au centre, l'image après une convolution mettant en évidence les contours. À droite, l'image de la convolution après le *pooling*.

Couches entièrement connectées

Après une ou plusieurs successions de couches de convolution et de *pooling*, nous obtenons des *feature maps* dont la dimension a été réduite. L'étape suivante est d'aplatir ces *feature maps* en un unique vecteur. Voir la figure 1.22 pour la visualisation de l'aplatissement.

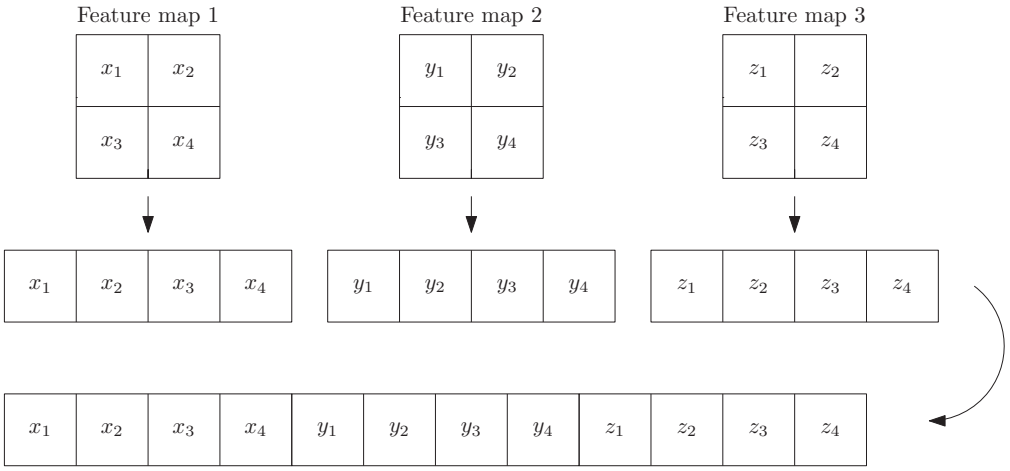


FIGURE 1.22 – Visualisation de l'aplatissement des *feature maps* en un vecteur.

Nous pouvons ensuite utiliser autant de couches entièrement connectées que désiré. Cette partie correspond au classificateur perceptron multicouche décrite à la section précédente.

1.3.1 ENTRAÎNEMENT D'UN MODÈLE

Tout comme un **MLP**, un **CNN** passe par une phase de propagation avant à la fin de laquelle il fait une prédiction. Cette prédiction est comparée à la vraie valeur par une fonction de perte. Finalement, l'ajustement des poids se fait avec la rétropropagation de l'erreur. Avec les **CNN**, les poids à ajuster se trouvent à deux niveaux. Tout comme les **MLP**, la partie classification du réseau possède des couches entièrement connectées par des liens qui sont à optimiser. La seconde partie à optimiser est les différentes matrices de convolution.

Rappelons que nos données sont séparées en ensemble d'entraînement et ensemble de validation. Nous passons premièrement toutes les images de l'ensemble d'entraînement et nous vérifions ensuite l'apprentissage avec l'ensemble de validation. C'est ce que nous appelons un *epoch*. En pratique, l'ajustement du modèle ne se fait pas après chaque image de l'ensemble d'entraînement. Nous préférons diviser nos images en groupe d'images appelé *batch*. Le

nombre d'images par groupe est défini par l'utilisateur. À la fin d'une *batch*, nous prenons l'erreur moyenne commise par le modèle et nous propageons ensuite cette erreur moyenne. Nous effectuons autant d'*epochs* que nécessaire pour l'apprentissage du modèle.

1.4 CONTRAINTES LIÉES À L'APPRENTISSAGE

Il faut faire varier plusieurs paramètres afin de trouver l'équilibre et obtenir un modèle qui fait des prédictions avec une bonne précision. Peu importe la technique utilisée, plusieurs facteurs peuvent influencer l'efficacité d'un modèle. En apprentissage supervisé, un bon modèle pourra généraliser les données vues en entraînement. Cela peut se traduire par un modèle qui performe autant pour les données d'entraînement que celles de validation. Lorsqu'un modèle n'a pas une bonne généralisation, il peut s'agir d'un problème de sous-apprentissage ou de surapprentissage. Avant de décrire ces concepts, nous allons d'abord aborder la notion d'erreur, de biais et de variance.

1.4.1 ERREUR, BIAIS ET VARIANCE

Nous optimisons le modèle en maximisant la précision et en minimisant l'erreur. Soit $y = f(x)$ et $\hat{y} = \hat{f}(x)$ où y est l'étiquette associée à la donnée x, $\hat{y}$ sa prédiction, $\hat{f}$ la fonction que simule le modèle pour approximer y. Pour faciliter la compréhension, prenons comme exemple une fonction de perte simple : l'erreur commise sur la prédiction est donnée par $y - \hat{y}$. En statistiques, il est commun de prendre le carré de l'erreur, soit

$$S = (y - \hat{y})^2. \tag{1.30}$$

Il faut minimiser S, mais d'abord, nous allons développer l'équation. En introduisant dans cette expression l'**espérance mathématique** de $\hat{y}$, c'est-à-dire ce que nous nous attendons à trouver en répétant une expérience un grand nombre de fois, nous obtenons $S = (y - E[\hat{y}] + E[\hat{y}] - \hat{y})^2$. En développant l'expression, l'équation 1.30 devient

$$S = (y - E[\hat{y}])^2 + (E[\hat{y}] - \hat{y})^2 + 2(y - E[\hat{y}])(E[\hat{y}] - \hat{y}).$$

Finalement, l'espérance mathématique de l'erreur au carrée est

$$E[S] = E[(y - E[\hat{y}])^2] + E[(E[\hat{y}] - \hat{y})^2] + 2E[(y - E[\hat{y}])(E[\hat{y}] - \hat{y})]. \tag{1.31}$$

La première partie de la somme de l'équation 1.31 correspond à ce qui est appelé le biais, élevé au carré :

$$\begin{aligned} E[(y - E[\hat{y}])^2] &= E[y^2 - 2yE[\hat{y}] + (E[\hat{y}])^2] \\ &= E[y^2] - E[2yE[\hat{y}]] + E[(E[\hat{y}])^2] \\ &= y^2 - 2yE[\hat{y}] + (E[\hat{y}])^2 \\ &= (y - E[\hat{y}])^2 \\ &= (Biais)^2. \end{aligned}$$

Un calcul similaire montre que la deuxième partie de la somme de l'équation 1.31 correspond à la variance :

$$E[(E[\hat{y}] - \hat{y})^2] = Variance$$

Finalement, en simplifiant la dernière partie de la somme, nous obtenons

$$\begin{aligned} 2E[(y-E[\hat{y}])(E[\hat{y}]-\hat{y})] &= 2(y-E[\hat{y}])E[E[\hat{y}]-\hat{y})] \\ &= 2(y-E[\hat{y}])(E[\hat{y}]-E[\hat{y}])) \\ &= 2(y-E[\hat{y}])(0) \\ &= 0. \end{aligned}$$

Avec ces calculs, nous pouvons réécrire l'équation 1.31 (Kong et Dietterich (1995)) :

$$E[S] = (Biais)^2 + Variance. \tag{1.32}$$

Le biais est la différence entre la moyenne des prédictions du modèle et la valeur réelle que le modèle aurait dû prédire. La variance représente la dispersion des données. C'est l'espérance mathématique de la différence au carré entre l'espérance de la moyenne des prédictions et les prédictions du modèle. Nous pouvons se représenter ces deux concepts à l'aide d'une cible. Comme illustré à la figure 1.23, une faible variance correspond à des points concentrés alors qu'un biais faible correspond à des points près du centre de la cible. Un biais ou une variance qui augmente implique une augmentation de l'erreur. Il faut tenter de trouver un juste milieu pour avoir un biais faible sans faire augmenter la variance et vice versa. C'est ce que nous appelons le *dilemme biais-variance*.

Nous pouvons voir à la figure 1.24 l'interdépendance entre l'erreur, le biais et la variance. Plus la complexité du modèle augmente, plus la variance augmente et notre modèle fait face à un problème de **surapprentissage**. Au contraire, lorsque la complexité du modèle n'est pas suffisante, le biais est élevé et nous parlons alors de **sous-apprentissage**.

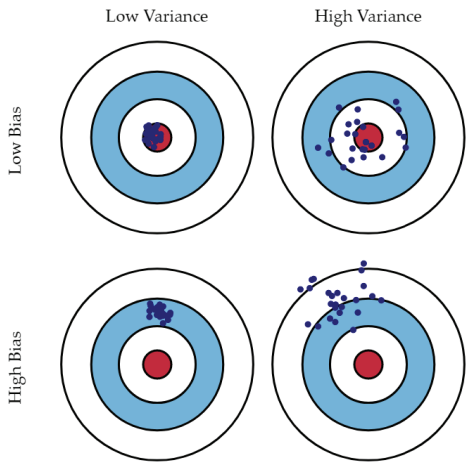


FIGURE 1.23 – Illustration du *dilemme biais-variance* par Fortmann-Roe (2012).

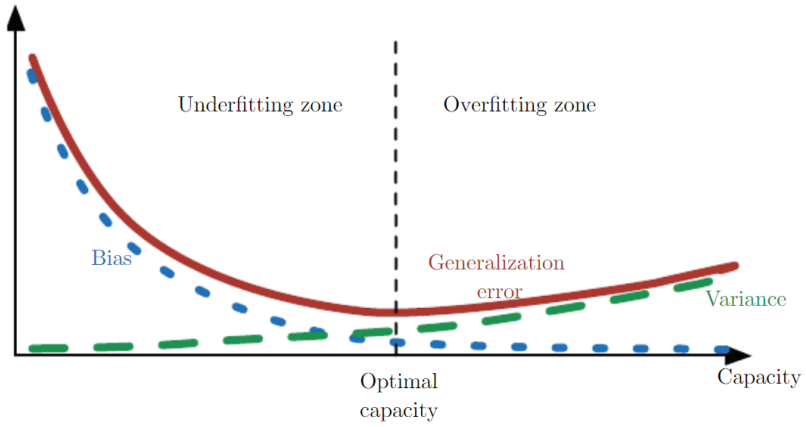


FIGURE 1.24 – L'erreur est grande lorsque le biais est élevé et la variance faible : nous parlons de sous apprentissage. Avec l'augmentation de la complexité de l'algorithme, la variance augmente et le biais diminue : nous parlons de surapprentissage. Entre les deux, nous retrouvons un point d'équilibre. Source : Goodfellow et al. (2016).

Une façon de visualiser le surapprentissage et le sous-apprentissage est à l'aide de la figure 1.25. Pour un problème de régression, nous voulons trouver une fonction qui généralise bien le nuage de points. En utilisant une fonction linéaire, la somme des erreurs commise entre les points et la fonction estimée est grande. Nous parlons ici d'un biais élevé et de sous-apprentissage. Nous obtenons une précision faible pour les données d'entraînement, mais aussi sur les données de validation. Au contraire, en utilisant une équation polynomiale qui passerait

par tous les points, nous obtiendrions une précision parfaite quant aux données d'entraînement. Cependant, pour les données de validation, nous obtiendrions une précision qui est plus basse. La variance est élevée et nous parlons de surapprentissage. Quelques techniques pour éviter ces deux problèmes liés à l'apprentissage sont présentés dans les deux sections suivantes.

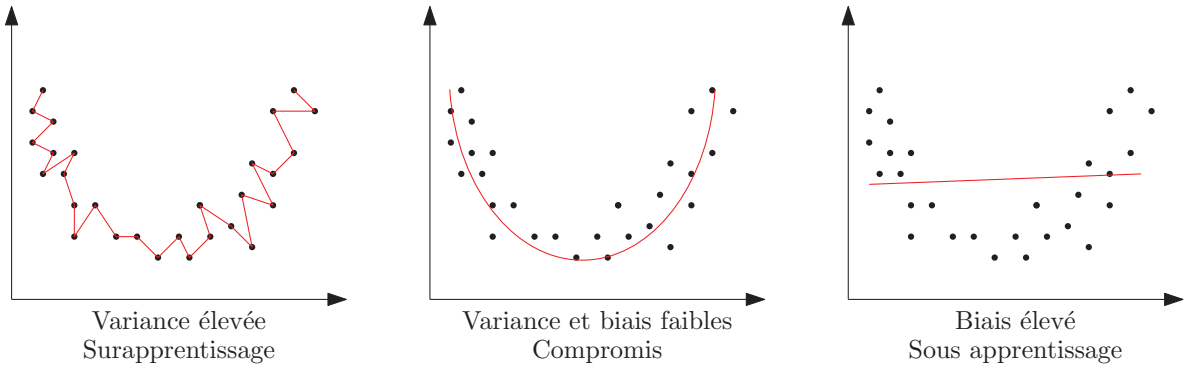


FIGURE 1.25 – À gauche : La fonction passe par chaque point du nuage, ce qui donne une variance élevée. La fonction ne généralise pas bien le nuage et nous parlons de surapprentissage. Au centre : Nous obtenons un compromis entre la variance et le biais. Cette fonction généralise bien le nuage de point. À droite : La fonction offre un biais élevé et ne généralise pas le nuage de point.

1.4.2 SURAPPRENTISSAGE

Nous pouvons détecter le surapprentissage en gardant un suivi de l'erreur et de la précision du modèle selon la complexité du modèle. Pour les réseaux de neurones, nous pouvons tracer ces deux indicateurs aux fils des *epochs*. Quant aux **SVM**, dans notre cas, la complexité peut dépendre du nombre de composantes principales conservé pour la classification. Dans les deux cas, la précision du modèle peut donner un taux de reconnaissance près de 1 pour les données d'entraînement, mais une convergence plus basse pour les données de validation. Nous pourrions dire que le modèle a appris «par coeur» les données d'entraînement. La précision du côté des données de validation se stabilise et il n'y a plus d'apprentissage. L'erreur commise par le modèle sur les données d'entraînement diminue et se stabilise. Pour les données de

validation, l'erreur diminue au départ, mais recommence à augmenter comme nous l'avions vu à la figure 1.24. Voici quelques techniques pour détecter ou prévenir le surapprentissage.

K-fold cross-validation Pour détecter le surapprentissage, il faut aussi avoir une bonne stratégie d'évaluation de notre modèle. Les données pourraient être séparées en entraînement et en validation de sorte que nous obtenions d'excellents résultats. Qu'adviendrait-il si les données étaient divisées autrement ? Pour éviter ce problème, nous utilisons le ***k-fold cross-validation*** ou la validation croisée. Les données sont divisées en k dossiers, telles qu'illustrées à la figure 1.26. À tour de rôle, chaque dossier constitue l'ensemble de validation alors que les $k-1$ dossiers restants servent d'ensemble d'entraînement. Cette méthode permet de confirmer la généralisation du modèle et de détecter un problème d'apprentissage.

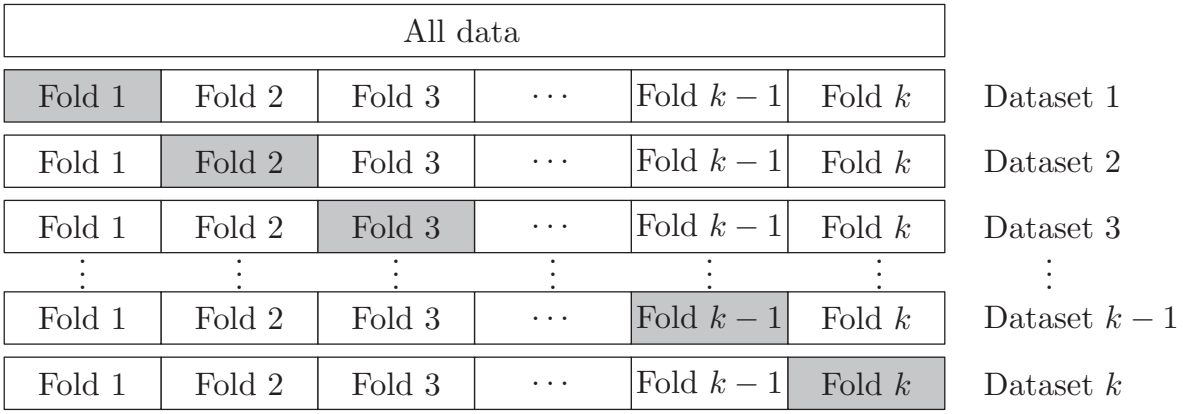


FIGURE 1.26 – Pour chaque ensemble, les dossiers en blanc correspondent aux données d'entraînement et celui en gris correspond aux données de validation.

Augmenter les données Avoir plus de données, lorsque cela est possible, réduit le surapprentissage. Sinon, nous pouvons faire de l'augmentation de données. Cela consiste à modifier légèrement nos données existantes afin d'en créer de nouvelles. Pour les images, nous pourrions appliquer des rotations, des translations, changer la luminosité, ajouter du bruit... Les augmentations doivent refléter la réalité. Par exemple, si notre

objectif est de classifier des chiffres, il faut éviter les rotations d'images au-delà de 90 degrés afin de ne pas confondre un six avec un neuf.

Early stopping Comme nous l'avons vu dans la section précédente, la complexité du modèle peut faire augmenter l'erreur. La technique *early-stopping* consiste à enregistrer le modèle avant le surapprentissage. Certains outils sont offerts dans les librairies d'apprentissage automatique. Nous pouvons aussi garder une trace de l'évolution de l'erreur et identifier à quel moment la courbe augmente, tel qu'illustré à la figure 1.27.

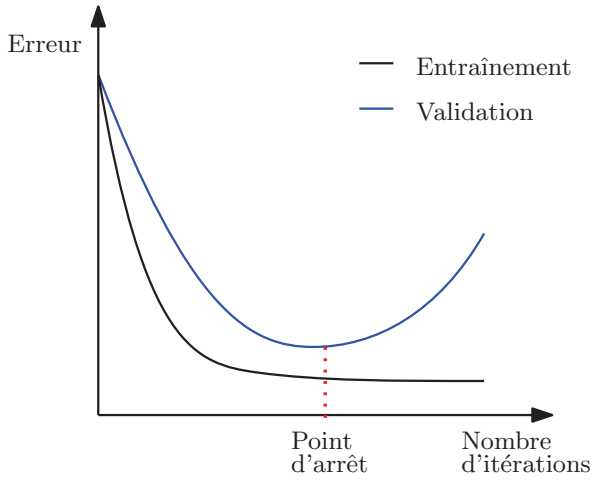


FIGURE 1.27 – Point d'arrêt de l'entraînement du modèle afin que l'erreur pour l'ensemble de validation ne commence pas à augmenter.

Feature selection Notre modèle peut rechercher des patrons beaucoup trop précis dans les données d'entraînement. Il faut alors réduire la complexité du modèle. Pour un **CNN**, nous pourrions le faire en réduisant le nombre de convolutions ou en diminuant le nombre d'*epochs*. Pour les **SVM**, nous pourrions choisir et comparer plusieurs méthodes d'extraction des caractéristiques.

Régularisation Goodfellow et al. (2016) décrit les régularisations comme une modification à l'algorithme qui permet de réduire l'erreur sur l'ensemble de validation, mais pas celle sur l'ensemble d'entraînement. Cela permet d'obtenir un modèle qui généralise mieux

des données jamais présentées au modèle pendant l'entraînement. Cela permet parfois de réduire la variance du modèle sans toutefois augmenter le biais.

Plusieurs techniques de régularisation existent. Parmi celles-ci, nous retrouvons les régularisations L^1 et L^2. L^2 est aussi connu sous le nom de *Ridge regression*. Nous ajoutons à la fonction de perte la norme euclidienne au carrée du vecteur formé des paramètres que nous voulons régulariser. Pour L^1, nous appliquons le même principe : nous ajoutons encore un terme à la perte, mais cette fois, nous ajoutons la somme des valeurs absolues des paramètres à régulariser.

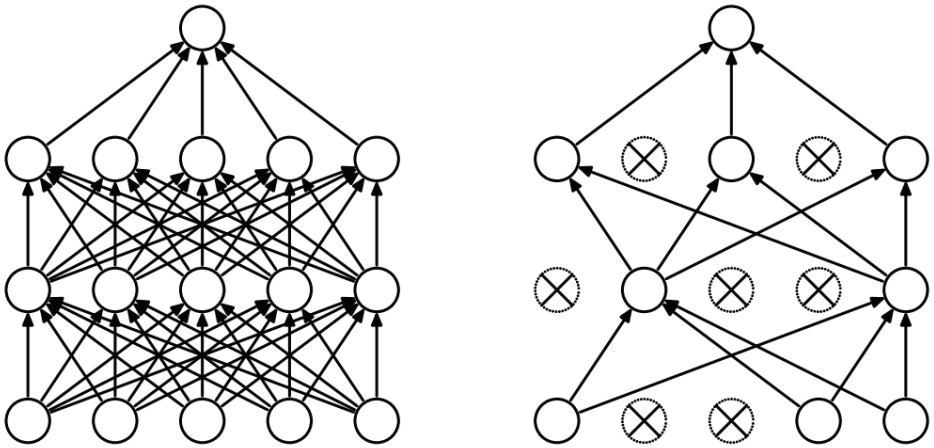
Dropout Cette technique est considérée comme une régularisation. Alors que les régularisations L^1 et L^2 intervenaient au niveau de la fonction de perte, *dropout* intervient à l'intérieur du réseau.

Le *dropout* ajoute de la robustesse au réseau de neurones et réduit les calculs. Elle laisse tomber temporairement et aléatoirement certains neurones du réseau, tel qu'illustré à la figure 1.28. Ceci est appliqué seulement lors de l'entraînement du modèle. Pendant l'évaluation, le réseau est complet.

1.4.3 SOUS-APPRENTISSAGE

Le sous-apprentissage se produit lorsque le modèle n'a pas été en mesure d'apprendre des données qui lui ont été fournies en entraînement. Puisque le problème est lié à celui du surapprentissage, les techniques pour le prévenir sont aussi liées. Tout comme le surapprentissage, la division des données en ensemble d'entraînement et de validation est importante. La technique *k-fold cross validation* peut donc être utilisée en tout temps.

Une solution est d'augmenter le nombre de données, mais ce n'est pas toujours possible. Lorsque nous voulons prévenir le surapprentissage, une solution était de diminuer la capacité



(a) Standard Neural Net (b) After applying dropout.

FIGURE 1.28 – À gauche, un réseau avec des couches entièrement connectées. À droite, un réseau avec un certain pourcentage de neurones temporairement laissé tombé. Source : Srivastava et al. (2014)

du modèle en diminuant l'extraction de caractéristiques. Cela diminue la complexité de l'algorithme et peut mener notre modèle à faire du sous-apprentissage.

L'objectif est donc de trouver un équilibre dans l'utilisation des techniques qui préviennent le surapprentissage et le sous-apprentissage.

1.5 CONCLUSION

En conclusion, nous venons de parcourir plusieurs techniques d'apprentissage automatique. Celles utilisées dans le mémoires ont été décrites en détail. La première méthode proposée dans ce mémoire utilise une combinaison du descripteur **HOG**, du **PCA** pour réduire la dimensionnalité et du **SVM** comme classificateur. La deuxième technique utilise un **CNN**. Nous avons aussi parcouru quelques techniques pour prévenir et diminuer les problèmes de surapprentissage ou de sous-apprentissage.

CHAPITRE 2

ÉTAT DE L'ART

Ce chapitre est divisé en deux sections. La section 2.1 présente une revue de la littérature. Puisque dans ce mémoire il est question de classification d'images, nous abordons d'abord, en 2.1.1, le sujet de classification d'objet de façon générale. Nous y présentons aussi des méthodes de classification dans le domaine de l'agriculture et enfin dans le domaine des insectes. Dans 2.1.2, nous présentons des méthodes de classifications plus spécifiquement dans le domaine des abeilles. Nous y traitons entre autres la classification d'images d'abeilles dans différents contextes et la détection de la reine par le son. Ces deux derniers sujets sont le cœur de ce mémoire : la détection de la reine à partir d'images.

La section 2.2 présente une revue des applications dans le domaine de l'apiculture. Nous y retrouvons des outils de gestion des ruches, de détection de l'état d'une ruche par le son et de détection à partir d'images.

2.1 REVUE DE LA LITTÉRATURE

2.1.1 CLASSIFICATION D'OBJETS

Dans le chapitre 1, les aspects théoriques des principales méthodes de classification utilisées dans ce mémoire ont été présentés. Nous pouvons retrouver des revues des techniques et méthodes pour améliorer les performances pour la classification d'images (Nawrocka et al. (2018), Lu et Weng (2007), Guo et al. (2016), Wu et al. (2019)). Certaines études ont conclu que l'apprentissage profond apporte de meilleurs résultats que les méthodes classiques, entre autres dans le domaine de la reconnaissance des expressions faciales (Yaddaden et al. (2018)). D'autres exemples d'études ayant obtenu de meilleurs résultats avec les **CNN** qu'avec les méthodes classiques seront présentés dans ce chapitre.

Des méthodes de classifications ont prouvé leur utilité dans le domaine de l'agriculture. Abdullahi et al. (2017) veulent distinguer des plants en santé, des plants partiellement en santé ou pas du tout en santé. Il a une base de données totalisant 1900 images en **RGB** de taille 224×224. Ces images ont été augmentées pour prévenir un surapprentissage, pour un total de 4588 images. Il utilise une technique de transfert d'apprentissage avec un modèle **CNN**, le VGG16 (Simonyan et Zisserman (2014)), pré entraîné. Ce modèle est utilisé pour la partie d'extraction de caractéristiques. La partie classification se fait avec un **SVM**. Deux tests sont présentés. Le premier utilise seulement deux classes d'images : en santé pour la première classe et pas du tout en santé pour la deuxième classe. Le deuxième test utilise les trois classes d'images en classification multi-classes avec une architecture un-contre-un ou un-contre-tous, menant donc à 3 sous-tests. Il utilise, comme stratégie d'évaluation, la validation croisée avec 10 dossiers. Le meilleur résultat est obtenu avec le premier test. Il obtient un taux de reconnaissance de 99.58%.

Kamilaris et Prenafeta-Boldú (2018) font une revue de la littérature sur l'apprentissage profond avec des images en agriculture. Les principaux domaines des 40 articles utilisés sont le cannabis, la couverture du sol, les plantes, le comptage de fruits et les cultures. Il analyse chaque article selon les sujets suivants : le prétraitement des images, les augmentations d'images, les techniques utilisées, les mesures de performance et la performance.

Shen et al. (2018) ont fait une étude sur la détection des insectes qui envahissent le grain stocké en utilisant des techniques d'apprentissage profond. Lorsque l'infestation n'est pas découverte à temps, cela peut causer la perte du grain et avoir de grandes conséquences sur la nourriture disponible pour les animaux de la ferme. Un échantillon de grain est pris en photos. Nous pouvons y retrouver des insectes, des grains complets ou fractionnés et des corps étrangés. Une identification manuelle de chaque échantillon pour y décrire ce qu'il contient est faite par un spécialiste. Les images ont été augmentées pour prévenir le surapprentissage. Ils utilisent la technique *Faster* **R-CNN** (Girshick (2015)) avec une modification du modèle *Inception* (Szegedy et al. (2015)) à 27 couches. Ils ont comparé son modèle en termes de précision et de temps d'exécutions avec le modèle VGG16, *Inception* et *ResNet101* (Dai et al. (2016). Ils obtiennet le meilleur taux de reconnaissance (87.99%) avec le modèle qu'ils proposent.

Xia et al. (2018) classifient 660 images de 24 insectes ravageurs pour les cultures comme le riz et le blé. Ils effectuent des augmentations avec ces images afin d'obtenir plus d'images et prévenir le surapprentissage. Le modèle proposé est composé des 16 premières couches de convolution du modèle VGG19. Chaque convolution est combinée à une technique appelée *Region Proposal Network* (Ren et al. (2015)) qui encadre des objets sur une image. Ainsi, l'insecte peut être localisé sur l'image. Ils obtiennet avec ce modèle de meilleurs résultats que l'état de l'art dans ce domaine, soit un taux de reconnaissance de 89.22%.

Thenmozhi et Reddy (2019) effectuent des travaux similaires. Ils utilisent trois jeux de

données. Deux d'entre eux contiennent 40 types d'insectes et le dernier est le même que Shen et al. (2018) ont utilisé. Ils appliquent eux aussi comme technique de prévention de surapprentissage l'augmentation de données. Leur modèle proposé est basé sur les **CNN**. Il consiste en 6 convolutions, 5 *pooling* et une couche entièrement connectée. Ils utilisent la fonction d'activation **ReLU**. Ils ont testé un nombre de convolutions allant de 3 à 7 et le modèle ayant 6 convolutions a mené au meilleur résultat. Ils ont aussi fait varier le taux d'apprentissage, obtenant les meilleurs résultats lorsqu'il est à 0.0001. Ils ont analysé l'impact du nombre d'*epochs* et la taille des *batchs* sur le taux de reconnaissance. Finalement, ils ont comparé leurs résultats en utilisant l'apprentissage par transfert. Ils ont utilisé *ResNet*, VGG, AlexNet (Krizhevsky et al. (2012)) et GoogLeNet (Szegedy et al. (2015)). Le modèle qu'ils proposent atteint de meilleurs résultats que les méthodes par transfert d'apprentissage.

Martineau et al. (2017) ont fait une revue de littérature sur différents modèles de classification d'insectes à partir d'images. Ils analysent 44 études d'images en 2D. Ils séparent leur article en trois parties : la provenance des images, les extracteurs de caractéristiques et les classificateurs utilisés.

2.1.2 CLASSIFICATION DANS LE DOMAINE DES ABEILLES

Cette section présente des études plus particulièrement dans le domaine des abeilles. Dans un premier temps, nous présentons la classification des abeilles par sous-espèce. Ensuite, nous présentons deux sujets de classification d'images d'abeilles : la présence de parasite sur une abeille et le transport du pollen par une abeille. Finalement, nous présentons quelques travaux antérieurs sur la détection de la reine par le son émis par la ruche.

Classification des abeilles par sous-espèces

Il existe environ 30 sous-espèces d'abeilles reconnues (Ruttner (2013)). Savoir les identifier est important pour assurer la diversité de l'espèce. Cela permet aussi de connaître à quel climat une sous-espèce est adaptée. Utzeri et al. (2018) sont parvenus à identifier la sous-espèce d'une colonie en analysant l'ADN laissé dans le miel produit. L'identification peut aussi se faire par l'analyse génétique de l'abeille (Garnery et al. (1998), Bodur et al. (2007), Syromyatnikov et al. (2018)). Plusieurs méthodes existent et peuvent être combinées afin de vérifier la sous-espèce (Meixner et al. (2013)). Oleksa et Tofilski (2015) ont combiné trois méthodes afin d'identifier les sous-espèces : deux d'entre elles utilisent l'ADN tandis que la dernière utilise la morphométrie géométrique des ailes. La dernière méthode consiste à positionner des points de repère aux croisements des veines des ailes. Ils obtiennet une forte corrélation dans le taux de reconnaissance entre les méthodes avec l'ADN et celle utilisant la morphométrie. Puisque l'analyse par ADN demande de l'équipement spécialisé, l'option d'identification par la morphométrie géométrique des ailes est intéressante.

Dans les expérimentations utilisant la géométrie morphométrique d'une aile, un spécialiste doit d'abord identifier à quelle sous-espèce appartient l'aile. Un travail manuel doit être ensuite fait afin d'identifier les points importants sur les ailes. Dans son expérimentation, da Silva et al. (2015) utilisent des images, dont 19 points de repère ont été identifiés. Ces points correspondent à la jonction des veines sur une aile d'abeille. L'étape d'extraction des caractéristiques se fait donc manuellement et les points de jonction des veines correspondent à une coordonnée. Leur objectif est de déterminer quelle combinaison de classificateurs et de sélection de caractéristiques offrirait le meilleur résultat dans la reconnaissance de la sous-espèce. Parmi les classificateurs utilisés, la classification naïve bayésienne a offert les meilleures performances.

Pour éviter l'extraction manuelle des caractéristiques, Buschbacher et al. (2020) utilisent les **CNN** pour faire la classification. Ils reprennent les travaux effectués par Steinhage et al. (2001) et proposent un système utilisant les images d'ailes d'abeilles sans les points de repère. La seule étape préliminaire manuelle nécessaire est l'identification de la sous-espèce appartenant à l'image. Cette étape devait déjà être faite lorsque la méthode avec les points de repère était utilisée. Ils font de l'augmentation de donnée et un balancement des classes. Ils utilisent le transfert d'apprentissage avec MobileNets (Howard et al. (2017)). En plus de supprimer l'étape de positionnement des points de repère, cette méthode a permis d'obtenir une forte corrélation entre la sous-espèce prédite et celle qui avait été identifiée.

Classification d'images d'abeilles

Une des causes qui peut ralentir la productivité et l'efficacité d'une ruche est la présence de *Varroa destructor*, un parasite que l'on retrouve chez les abeilles. Il existe des méthodes chimiques, mais néfastes pour la colonie, qui permettent d'éliminer ce parasite. Il est important de connaître le niveau d'infestation de la colonie afin que l'apiculteur puisse intervenir au bon moment. Une technique manuelle consiste à prendre un échantillon d'abeille et d'estimer ce niveau. C'est une opération qui peut perturber la colonie et qui est coûteuse en temps. Un système consiste à installer une caméra à l'entrée de la ruche et prendre en vidéo les allées et venues des abeilles. Ils découpent ensuite ces vidéos en images et nous revenons à un problème de classification d'images. Chazette et al. (2016) décrivent un système qui se détaille en deux temps. Premièrement, il identifie les abeilles atteintes par un parasite et ensuite il détruit le parasite à l'aide d'un laser qui ne tue pas l'abeille.

Schurischuster et al. (2018) découpent leur vidéo en images et ensuite en imagettes afin d'obtenir une seule abeille par imagette. Ils utilisent différentes combinaisons d'extracteurs

de caractéristiques et de classificateurs. Comme extracteur de caractéristiques, ils utilisent un histogramme de couleurs, un **LBP**, et *dense **SURF*** . Ils utilisent la classification naïve bayésienne, les **SVM** et *Random Forest*. L'objectif est de classifier les images d'abeilles en deux catégories : avec parasite ou sans parasite. Les meilleurs résultats ont été obtenus en utilisant une combinaison de l'extracteur de caractéristiques **SURF** et le classificateur *Random Forest*.

Bjerge et al. (2019) ont plutôt utilisé une méthode par apprentissage profond afin de classer des images provenant d'une vidéo captant les abeilles à l'entrée de la ruche. Deux séries de données ont été considérées. La première vise à ne pas perturber la ruche : nous retrouvons toutefois moins de jeunes abeilles qui restent à l'intérieur de la ruche, mais qui peuvent être infestées par le parasite. La deuxième consiste à perturber la ruche pour faire sortir les abeilles. De la fumée force les abeilles à regagner la ruche par le passage qui est filmé par la caméra. Pour établir le nombre réel d'abeilles infectées, deux stratégies sont employées. La première consiste à prendre un échantillon, de le nettoyer à l'aide de savon et de mesurer le poids. En guise de seconde stratégie pour établir le niveau d'infestation, un apiculteur a manuellement mesuré le taux d'infestation de la ruche. C'est ce dernier taux qui servira finalement de comparaison. Pour effectuer la classification, les auteurs utilisent un **CNN**. Celui-ci possède quatre couches de convolutions avec ReLU comme fonction d'activation. Après cette étape d'extraction de caractéristiques, nous retrouvons une couche entièrement connectée avec une fonction sigmoïde pour l'activation. Le modèle a mené à un taux de faux positifs s'élevant à environ 1%.

Dans un autre ordre d'idées, l'installation d'un système à l'entrée de la ruche qui capture des vidéos d'abeilles peut aussi servir à détecter si une abeille porte du pollen. La quantité de pollen qui est acheminée à la ruche est un indicateur de l'état de la ruche. En effet, le pollen est entre autres utilisé pour la préparation des nids des larves. Chaque sous-espèce

d'abeille porte le pollen de façon différente. Comme mentionné par Rodriguez et al. (2018), il est donc possible de devoir entraîner un modèle de classification pour chaque sous-espèce. Ces derniers auteurs ont comparé des méthodes classiques (**k-NN**, classification naïve bayésienne et un **SVM**) à une approche avec un **CNN**. Le meilleur résultat pour les méthodes classiques est de 91.16% et a été obtenu en normalisant les données, en utilisant **PCA** pour réduire la dimensionnalité et un **SVM** avec **RBF** *kernel* pour la classification. Pour la méthode utilisant les **CNN**, ils obtiennet un résultat de 96.4%.

Détection de la reine par le son

Les abeilles peuvent communiquer, même dans l'obscurité, par les vibrations et par le son (Kirchner (1993)). Elles peuvent exercer une danse qui indique à la colonie la direction et la distance d'une source de nourriture (Michelsen et al. (1986)). Certains apiculteurs expérimentés sont en mesure de détecter les variations de son dans la ruche. Des outils technologiques ont été développés pour faciliter la tâche des apiculteurs pour qu'ils puissent intervenir au bon moment lorsqu'il y a un changement dans le son de la colonie.

Kulyukin et al. (2018) ont distingué le son des abeilles de celui des criquets et du bruit ambiant. Ils font leurs expérimentations sur deux bases de données. Pour classifier les différents sons, leur première approche fait intervenir des méthodes classiques. Ils utilisent une combinaison d'extracteurs de caractéristiques **MFCC**, **STFT**, melspectogram, STFT spectral et tonnezt) et de classificateurs (régression logistique, **k-NN**, **SVM** et *random forest*). La deuxième approche fait intervenir les **CNN** (sur les données brutes et sur le spectrogramme). Bien que le temps d'apprentissage soit plus élevé avec la deuxième méthode, elle permet d'obtenir de meilleurs taux de reconnaissance lorsqu'elle est appliquée aux données brutes. Ils remarquent aussi qu'ajouter de la profondeur au réseau de neurones n'améliore pas nécessairement les résultats.

Un changement dans le son peut se faire remarquer lorsque la reine meurt ou est absente. Nous pouvons aussi remarquer un son différent quand la colonie se prépare à un essaimage. La reine pond des œufs de prochaines reines. Lorsqu'elles sont arrivées à maturité, une reine peut partir avec une partie des ouvrières de la colonie afin d'aller peupler une nouvelle ruche. L'essaimage est un phénomène naturel qui peut être contrôlé par l'apiculteur. En effet, un essaimage signifie une perte au niveau du miel puisqu'il y a moins d'abeilles pour le produire. Certains comportements peuvent être observés afin de détecter l'essaimage. Avant que l'évènement ne se produise, nous pouvons remarquer un changement au niveau du son, de la température et de l'humidité de la ruche. Les deux premiers facteurs sont plus facilement mesurables que le troisième puisque l'humidité interne de la ruche dépendant grandement de l'humidité de l'air. Ferrari et al. (2008) ont mené une expérience qui a pour but de construire un système qui informe l'apiculteur qu'un essaimage approche. Pour ce faire, ils ont installé sur trois ruches un équipement qui capte le son, la température interne et externe ainsi que l'humidité relative de la ruche. L'appareil de captation du son peut traiter des fréquences de 50 Hz à 16 000 Hz. Au total, 270 heures de son sont à analyser. La première étape de traitement du son consiste à le filtrer pour enlever les bruits environnants. Il y a ensuite une synchronisation avec les données de température et d'humidité. Enfin, l'écoute du son en analysant le spectrogramme permet de manuellement diviser et identifier le son par classes. Au total, 9 essaimages d'une durée moyenne de 35 minutes ont été détectés. Ils sont caractérisés par une fréquence passant de 100-300 HZ à 500-600 Hz et une diminution de la température de 35°C à 33°C. La diminution de température peut être expliquée par le fait que les abeilles créent une aération avec l'augmentation du battement d'ailes. Cette dernière méthode demande un traitement manuel des signaux. Certains ont créé des modèles afin de prédire un essaimage ou l'absence d'une reine.

Vérifier manuellement si la reine est présente en retirant un à un chaque cadre peut être pertur-

bant pour une colonie d'abeilles. Lorsqu'il n'est pas nécessaire de savoir exactement à quel endroit se retrouve la reine dans la ruche, il est possible d'utiliser une méthode non invasive, comme l'analyse du son, pour constater un changement. Un signal est envoyé à l'apiculteur et ce dernier peut effectuer des manipulations sur la ruche, au besoin. Dans ses travaux, Cejrowski et al. (2018) ont développé une application qui détecte l'absence de la reine. Un système est installé sur un des cadres de la ruche. Celui-ci est composé d'un capteur de son, de température et d'humidité. Toutes les 15 minutes, les données sont envoyées au serveur pour l'analyse. Pour bâtir leurs modèles, ils ont d'abord récolté des données. Puisqu'aucun essaimage n'a eu lieu, ils ont dû le simuler en retirant la reine manuellement. Un étiquetage des signaux audios a été effectué : état normal avant les manipulations, juste avant le retrait de la reine, aucune reine dans la ruche, après l'ajout d'une nouvelle reine, état normal après la nouvelle reine. Plusieurs modèles ont été testés sur les données récoltées. La technique d'extraction des caractéristiques utilisée est *Linear Predictive Coding* (O'Shaughnessy (1988)), qui permet de compresser le vecteur de son. Une réduction de la dimensionnalité est appliquée. Finalement, la classification se fait via les **SVM**. Ils ont été en mesure de détecter les changements dans le comportement des abeilles lors du retrait, de la transition et de l'ajout de la nouvelle reine. De plus, ils ont remarqué une différence entre l'état normal des abeilles selon l'ancienne et la nouvelle reine.

Dans le même esprit d'idées, Nolasco et al. (2019) ont conduit une expérience similaire, mais en introduisant cette fois une technique d'apprentissage profond, les **CNN**. Des données provenant de deux ruches ont été utilisées. Pour chacune d'elles, 24 heures de données ont été prises avec la reine et un temps égal sans la reine. Deux séparations de données ont été faites : une première expérience avec les données d'entraînement provenant d'une ruche alors que les données de validation proviennent de l'autre et une deuxième expérience avec les données aléatoirement séparées en entraînement et en test. Plusieurs techniques d'extraction

de caractéristiques ont été utilisées **MFCC**, *mel spectrogram* et *Hilbert-Huang transform* **HHT** (Huang (2014)) et combinées à deux classificateurs (**SVM** et **CNN**). Pour le **SVM**, les meilleurs résultats ont été obtenus avec une combinaison de **MFCC** et **HHT**. Pour le **CNN**, les meilleurs résultats sont atteints lorsqu'il est combiné à un **MFCC**. Cependant, le **CNN** ne semble pas bien généraliser le cas où les données sont séparées selon les ruches (pas de façons aléatoires).

La méthode de détection de la reine par le son permet de savoir si elle est dans la ruche, mais pas précisément sur quel cadre. Lorsqu'il est nécessaire de savoir exactement où est située la reine, il existe une méthode qui consiste à prendre les cadres et faire passer les abeilles à travers un filtre qui ne permet pas à la reine de passer. Cela permet d'isoler la reine et ensuite effectuer les manipulations nécessaires avec celle-ci. Cette méthode est très invasive et perturbe la colonie. Une autre méthode consiste à mettre une marque sur le thorax de la reine. Cette étape peut s'avérer stressante pour la reine. Qu'il y ait une marque ou non sur la reine, l'apiculteur doit retirer un à un les cadres et chercher la reine en parcourant les cadres du regard. Au meilleur de notre connaissance, il n'existe pas de méthode permettant de détecter la reine à partir d'une image du cadre.

2.2 REVUE DES APPLICATIONS

Il existe plusieurs applications pour la gestion des ruches d'abeilles. Certaines servent de journal de bord pour le suivi de la ruche à travers le temps (BoxyBEE (2021), Hive Tracks (2021), BeeKing (2021)). Certaines applications ont l'option d'enregistrer les sons de la ruche pendant l'inspection ou requièrent l'achat de matériel pour l'enregistrement (Brood Minder (2021), Bee Appy (2021), OSBeehives (2021)). D'autres applications incluent un système de gestion des ventes pour le miel et autres produits dérivés (BeePlus Beekeeping Manager

(2021), HiveKeepers (2021)).

Lorsque le projet de maîtrise a débuté, il n'y avait, au meilleur de notre connaissance, aucune application gratuite et *open source* qui permettait de prendre en photo un cadre de ruche d'abeilles et déterminer si la reine y est présente. Nous retrouvons maintenant une application qui permet de localiser et suivre en temps réel la reine en filmant le cadre à partir d'un appareil mobile (Bee Queen Detector (2021)). Il y a cependant peu d'informations concernant les techniques utilisées dans l'application et leurs performances . Ainsi, il est impossible de comparer nos résultats et nos méthodes proposées à ce qui a été implémenté dans cette application.

2.3 CONCLUSION

En conclusion, nous n'avons pas de comparatif pour le type de travaux effectués dans ce mémoire. Nous pouvons retrouver des exemples de classification d'images dans le domaine des abeilles pour la classification de sous-espèces, la détection de parasite sur une abeille ou encore la détection de pollen porté par une abeille.

CHAPITRE 3

MÉTHODES PROPOSÉES

Dans cette section, nous allons présenter les deux méthodes proposées pour la classification d'images d'abeilles par apprentissage automatique. La première fait intervenir une méthode classique utilisant un extracteur de caractéristiques, une technique pour la réduction de la dimensionnalité et un classificateur. Nous retrouvons les détails théoriques de ces techniques aux sections 1.2.1, 1.2.2 et 1.2.3. La seconde est une méthode par apprentissage profond, décrite théoriquement en 1.3 et 1.2.3.

Dans ce chapitre, les détails techniques sont présentés pour chaque méthode. La méthode basée sur les **SVM** est présentée à la section 3.1 tandis que la méthode basée sur les **CNN** est à la section 3.2

3.1 MÉTHODE BASÉE SUR LES MACHINES À VECTEURS DE SUPPORT

Cette première méthode tire son inspiration d'une méthode qui a fait ses preuves dans le cadre de détection de piétons (Kobayashi et al. (2007)) et de reconnaissance d'expressions faciales (Yaddaden et al. (2021)). La structure du modèle est présentée à la figure 3.1.

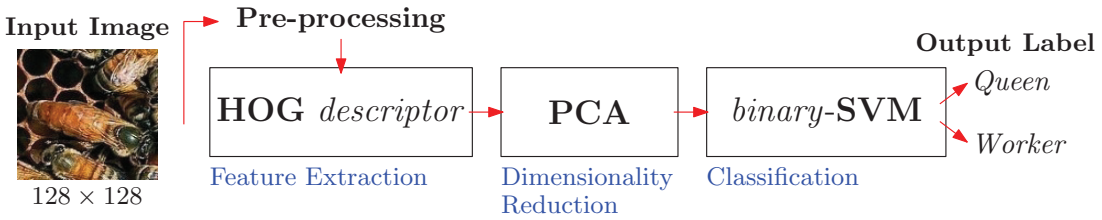


FIGURE 3.1 – Vue d'ensemble de la première méthode proposée.

3.1.1 ENTRÉE DU MODÈLE ET PRÉTRAITEMENT

Nous avons des images de dimension 128×128 pixels. Avant de commencer l'étape d'extraction de caractéristiques, les images subissent une étape de prétraitement. Une égalisation d'histogramme est appliquée aux images afin de contrer les problèmes de luminosité et améliorer les contrastes.

3.1.2 EXTRACTION DES CARACTÉRISTIQUES

L'extracteur de caractéristiques utilisé est le descripteur **HOG**. Les valeurs des paramètres de cet algorithme ont été fixées de façon empirique. Plusieurs valeurs de paramètre ont été testées et ce sont les suivantes qui ont été conservées. Nous formons des cellules de 16×16 pixels. Nous retrouvons ainsi, pour des images initiales ayant 128 pixels par côté, $8 \times 8 = 64$ cellules. Pour chacune des cellules, nous calculons l'histogramme des gradients en utilisant 16 classes pour l'orientation des gradients. Des blocs de 4 cellules par 4 cellules sont formés, pour un total de 16 cellules par bloc. En concaténant les histogrammes de chaque cellule, nous obtenons pour un bloc un vecteur de taille $16 \times 16 = 256$. Nous normalisons ensuite ce vecteur. Sur notre image, nous pouvons former 5 blocs horizontalement et 5 blocs verticalement pour un total de 25 blocs. En concaténant les 25 vecteurs de taille 256, nous obtenons un vecteur final de taille $25 \times 256 = 6400$.

3.1.3 RÉDUCTION DE LA DIMENSIONNALITÉ

Nous avons utilisé la **PCA** afin de réduire la dimensionnalité. Le nombre maximal de composantes principales qu'il est possible d'extraire est la valeur minimale entre le nombre d'échantillons et la taille du vecteur. À l'étape précédente, celle d'extraction des caractéristiques, nous avons obtenu, pour une image, un vecteur de taille 6400. Comme nous allons le voir à la section 4.1.3, le nombre d'échantillons retenu est de 476.

3.1.4 CLASSIFICATION

Pour classifier nos images, nous avons utilisé comme classificateur un **SVM**. Il a été déterminé empiriquement que le noyau **RBF** permettait d'obtenir les meilleurs résultats

3.2 MÉTHODE BASÉE SUR UN RÉSEAU DE NEURONES À CONVOLUTION

Par la seconde méthode proposée, nous souhaitons comparer les résultats obtenus avec ceux de la première méthode proposée. De plus, elle permet de regrouper en une seule architecture les trois étapes nécessaires pour la première méthode. L'architecture utilisée dans cette section est inspirée de **LeNet-5** introduit par LeCun et al. (1998) pour classifier des chiffres écrits à la main. Tout comme cette architecture, la nôtre possède deux étapes d'extraction des caractéristiques. Nous avons par la suite trois couches entièrement connectées et une couche pour la classification.

Dans cette section, nous décrivons l'architecture en détail qui est illustrée à la figure 3.2.

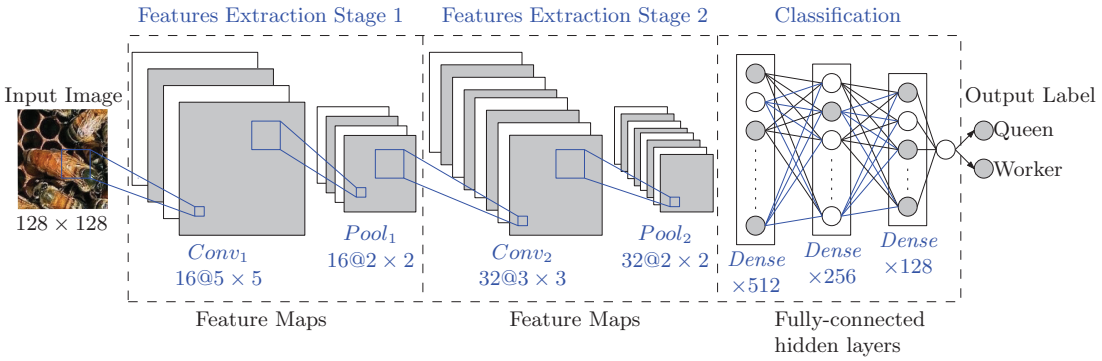


FIGURE 3.2 – Vue d'ensemble de la deuxième méthode proposée.

3.2.1 ENTRÉE DU MODÈLE

Comme nous l'avons vu au chapitre 1.4.2, un nombre restreint d'images peut mener notre modèle au surapprentissage et l'un des moyens d'éviter ce problème est l'augmentation d'images. Cette technique consiste à appliquer des transformations sur les images existantes afin d'en créer de nouvelles artificiellement. Une telle image doit être représentative d'une image qui aurait été prise par un appareil photo. De plus, il faut bien choisir le type d'augmentation utilisé puisque certains pourraient être risqués. Par exemple, si notre objectif est de classifier des chiffres, il faut éviter les rotations d'images au-delà de 90 degrés afin de ne pas confondre un six avec un neuf (Shorten et Khoshgoftaar (2019)). Un autre exemple de transformation risqué est présenté à la figure 3.3. L'image originale porte l'étiquette de reine ainsi que l'image transformée. Dans cet exemple, la version d'image transformée conserve l'étiquette de reine, mais ne comprend plus de reine. Ce n'est donc pas un bon choix de transformation à appliquer aux images.

Plusieurs compositions de ces transformations, qui réfètent la réalité, ont été testées empiriquement. Celles offrant le meilleur taux de reconnaissance sont celles qui ont été retenues afin de produire une plus grande variété d'images. Voici donc les types d'augmentation que nous avons utilisés.

(a) L'image originale dont la reine, au coin supérieur droit, a été marquée par un cercle rouge.

(b) L'image ayant subi une rotation trop grande et ayant ainsi mené à la perte de la reine.

FIGURE 3.3 – Transformation risquée appliquée à une image contenant une reine.

Rescale Cette augmentation est plutôt un prétraitement puisqu'elle est appliquée à toutes les images. Nous divisons chaque pixel par 255 afin de ramener chaque pixel dans un intervalle de 0 à 1.

Rotation Comme expliqué ci-dessus, en effectuant une rotation, nous pouvons risquer d'avoir des images qui ne portent plus la bonne étiquette. Nous avons analysé l'impact d'une rotation sur nos imagettes. Nous avons choisi une rotation ne dépassant pas $-30°$ et $30°$ par rapport à l'image originale.

Luminosité Nous avons opté pour un changement mineur dans la luminosité. L'idée est de reproduire une image qui aurait être prise par un apiculteur. Nul besoin de trop éclaircir ou trop assombrir l'image. Nous avons choisi un facteur de 0,8 à 1,2 à appliquer à nos imagettes.

Symétrie Une symétrie horizontale et verticale a été choisie. Combinée à la rotation, nous pouvons couvrir plusieurs positions possibles dans lesquelles une abeille peut se retrouver.

3.2.2 DESCRIPTION DE L'ARCHITECTURE

Nous décrivons chaque couche de l'architecture du **CNN**. La dimension et le nombre de convolutions, la dimension des fenêtres de *pooling*, le nombre et la taille de couches *denses* ont été choisis en faisant quelques tests. La combinaison donnant les meilleurs résultats a été conservée.

Couche 1 Nous avons utilisé une première couche de convolution. Un total de 16 filtres de dimension 5 par 5 a été utilisé.

Nous obtenons pour les filtres un total de $5 \times 5 \times 16 = 400$ paramètres. Comme notre image en entrée est en couleur, nous avons donc à multiplier le dernier nombre par 3, pour un total de $400 \times 3 = 1\ 200$ paramètre. Finalement, nous devons y ajouter les poids qui relient l'image de départ aux 16 convolutions, pour un total global de 1 216 paramètres au terme de cette couche.

Nous passons à la prochaine couche 16 *feature maps* de 128×128.

Couche 2 Une couche de *pooling* suit la couche de convolution. Nous avons utilisé un *max-pooling* avec une fenêtre de dimension 2 par 2. La taille de chaque *feature map* se voit diminuer de moitié. Après ces deux premières couches, nous avons pu extraire des caractéristiques de l'image de départ.

Nous obtenons 16 *feature maps* de 64×64 qui deviendra l'entrée de la prochaine couche.

Couche 3 Il y a une autre couche de convolution. Cette fois-ci, nous avant 32 filtres de dimension 3 par 3 à appliquer aux 16 *feature maps* de l'étape précédente.

Nous obtenons pour les filtres un total de $3 \times 3 \times 32 = 288$ paramètres que l'on doit multiplier par 16 *feature maps* créées à la convolution précédente, pour un total de $288 \times 16 = 4\ 608$ paramètre. Finalement, nous devons y ajouter les poids qui les relient

aux 16 *feature maps* de la couche précédente, pour un total global de 4 640 paramètres pour cette couche.

Nous passons à la prochaine couche 32 *feature maps* de 64×64.

Couche 4 Nous avons la même couche que la couche 2. Nous obtenons 32 *feature maps* de 32×32 qui deviendra l'entrée de la prochaine couche.

Couche 5 Une couche dropout avec un taux de 40% est ajoutée.

Couche 6 Les *feature maps* sont transformés pour former un seul vecteur. Nous obtenons un vecteur de dimension $32 \times 32 \times 32 = 32\,768$. C'est le début de la classification avec les couches entièrement connectées.

Couche 7 La première couche dense a 512 neurones. Les 32 768 neurones de la couche précédente sont entièrement connectés avec les 512 neurones de cette couche.

Nous obtenons $32\,768 \times 512 = 16\,777\,216$ paramètres. Nous devons ajouter 512 à ce nombre pour tenir compte des biais. Nous avons un total de 16 777 728 paramètres pour cette couche.

Couche 8 La deuxième couche dense de 256 neurones connecte les 512 neurones de la couche précédente.

En tenant compte encore une fois des biais, nous avons au total pour cette couche un nombre de paramètres égal à $256 \times 512 + 256 = 131\,328$.

Couche 9 Une troisième couche dense de 128 neurones connecte les 256 neurones de la couche précédente.

Nous retrouvons à cette étape $128 \times 256 + 128 = 32\,896$ paramètres.

Couche 10 Finalement, une dernière couche dense avec un seul neurone permet de prendre la décision à savoir si l'image en entrée est une reine ou une ouvrière.

Pour cette couche, nous dénombrons $1 \times 128 + 1 = 129$ paramètres.

Aux couches 1, 3, 7, 8 et 9, nous avons utilisé la fonction d'activation ReLU. Pour la dernière couche, nous avons utilisé une fonction sigmoïde. Le nombre total de paramètres à entraîner est de $1\,216 + 4\,640 + 32\,768 + 16\,777\,728 + 131\,328 + 32\,896 + 129 = 16\,947\,937$.

La compilation du modèle a été faite avec l'optimiseur *Adam* (Tensorflow (2021b)). Un taux d'apprentissage de $0,01$ est utilisé. La fonction de perte est la *Binary Crossentropy* (Tensorflow (2021a)). L'optimiseur, le taux d'apprentissage, la fonction de perte et les fonctions d'activation ont été choisis empiriquement. L'entraînement s'est fait sur 200 epoch. Des groupes de 32 images sont formés avant d'ajuster les paramètres. Ainsi, 14 ajustements sont faits à chaque epoch.

3.3 CONCLUSION

Pour conclure ce chapitre, nous avons présenté les détails techniques des deux méthodes proposées. La méthode classique utilisant un classificateur **SVM** est accompagnée d'un descripteur **HOG** pour extraire les caractéristiques des images et de la technique **PCA** pour réduire la dimensionnalité. La seconde méthode utilise un **CNN** avec deux couches de convolution et trois couches denses pour la classification. Au chapitre 4, nous présentons le traitement des images, les résultats et une analyse comparative des méthodes proposées.

CHAPITRE 4

EXPÉRIMENTATIONS ET RÉSULTATS

Dans cette section, nous présentons les étapes de l'obtention des données jusqu'aux résultats. Il est question, dans la section 4.1, du traitement des images originales afin de les préparer pour l'entraînement des modèles proposés. Le protocole d'expérimentation et la stratégie de validation sont présentés dans la section 4.2. Finalement, les résultats sont exposés à la section 4.3. Une analyse et une comparaison sont faites dans cette dernière section.

4.1 TRAITEMENT DES IMAGES

Dans cette section, comme illustrées à la figure 4.1, les différentes étapes pour la préparation des images sont présentées. La première étape consiste à former une banque d'images. Il faut ensuite extraire des imagettes de reine et d'ouvrière. Finalement, des prétraitements sont appliqués à chaque imagette créée. Après ces étapes, les images sont prêtes à être utilisées pour l'entraînement d'un modèle.

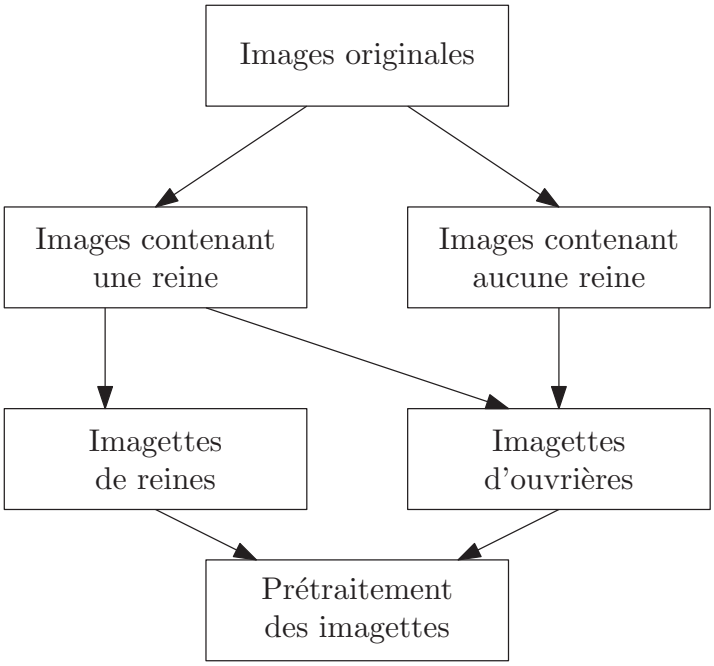


FIGURE 4.1 – Étapes de traitement des images.

4.1.1 CUEILLETTE ET TRI DES IMAGES

Lors d'un stage de premier cycle, Guillaume Gingras a fait la première cueillette des images à partir de *Google Images* et a ensuite visualisé les images afin de les classer dans deux catégories :

1. Reine : une reine, complète ou partielle, pouvant être accompagnée d'autres abeilles, se retrouve dans l'image ;
2. Pas reine : aucune reine ne se retrouve dans l'image. Nous utiliserons le terme *ouvrière* pour désigner cette classe. Il faut toutefois noter que nous pouvons retrouver des faux bourdons qu'il est facile de confondre avec des ouvrières.

Ces images ont été utilisées dans le cadre du projet dont il est question dans ce mémoire. Plusieurs passages sur l'ensemble des images ont été nécessaires. Celles contenant un droit

d'auteur ont été retirées. Certaines images contiennent des ajouts électroniques comme du texte ou des encadrements autour de la reine. D'autres ont des reines pointées par un doigt ou un bâton. Lorsque cela a été possible, que les ajouts soient électroniques ou physiques, ils ont été rognés afin de conserver le maximum d'images de reine. Si ce n'était pas possible, l'image a été exclue. Les apiculteurs peuvent marquer physiquement les reines d'un point sur l'abdomen afin de les reconnaître plus facilement lorsque le cadre est retiré de la ruche. Plusieurs images contiennent des reines ayant cette caractéristique. Comme l'objectif est de détecter la présence de la reine sans exiger à ce qu'elle soit marquée, celles-ci ont été retirées de la base de données pour ne pas biaiser les résultats. Une fois les images non désirées supprimées ou rognées, l'application *Awesome Duplicate Photo Finder* (Awesome Duplicate Photo Finer (2021)) a été utilisée afin de supprimer les doublons.

Un des membres de notre équipe possède des ruches d'abeilles. Nous avons ajouté quelques photos de cadres d'abeilles afin d'augmenter la quantité d'images en notre possession. De plus, à l'automne 2020, une nouvelle fouille sur *Google Images* a été effectuée afin d'augmenter la quantité de reines dans notre ensemble de données avec les nouvelles images qui se sont ajoutées depuis le début du projet. Les mêmes étapes que celles décrites précédemment ont été effectuées afin de trier ces nouvelles images.

Après toutes ces étapes, nous retrouvons 448 images contenant une reine que l'on nomme *original_queen*. Nous retrouvons aussi 1139 images ne contenant pas de reine. Appelons ces images *original_worker*. La prochaine étape consiste à isoler une reine ou des ouvrières sur les images *originales* afin de créer des imagettes.

4.1.2 IMAGETTES DE REINE ET D'OUVRIÈRE

Dans cette section, le processus de création d'une imagette de reine et d'une imagette d'ouvrière à partir d'une image originale est décrit.

Le dossier *original_queen* est d'abord traité pour créer des imagettes de reine. Les étapes sont détaillées à l'annexe .1.1. On obtient 366 imagettes de reine. Nous obtenons 350 images qui contiennent au moins une abeille qui n'est pas une reine et dont la reine a été remplacée par un carré blanc. Suite au traitement des images contenant des reines, nous nous retrouvons avec trois dossiers, soit le dossier d'images originales *original_queen*, le dossier d'imagettes de reine isolée *queen_isolated* et le dossier des images originales sans la reine *no_queen*. Le dossier *original_queen* est le seul qui ne sera plus utilisé pour la suite. Le dossier *queen_isolated* est utilisé à l'étape des prétraitements. Finalement, le dossier *no_queen* est traité par l'application de découpe afin d'obtenir des imagettes d'ouvrière.

Les applications trouvées, par exemple **CVAT** (CVAT (2018)), ne permettaient pas de répondre à nos besoins : découper l'image en imagettes et les étiqueter selon les classes. Lors de son stage, Guillaume Gingras a donc développé une application qui découpe les images en imagettes et qui étiquette ces imagettes. Cette application a été utilisée pour ce projet et des fonctionnalités pratiques y ont été ajoutées. Une description de cet outil est présentée à l'annexe .1.2. Quand les images des dossiers *no_queen* et *original_worker* ont toutes été passées dans l'application, nous pouvons extraire le dossier *Complete*. Celui-ci, qui contient 9289 imagettes d'abeille non reine complète, constituera notre banque d'images pour abeilles qui ne sont pas une reine.

On obtient donc 366 imagettes de reine et 9289 imagettes d'abeilles qui ne sont pas une reine.

4.1.3 SÉLECTION ET PRÉTRAITEMENT DES IMAGES

Nous retrouvons deux dossiers contenant des imagettes : un contenant des reines (*queen_isolated*) et un ne contenant pas de reine (*Complete*). Nous avons d'abord retenu les images ayant une taille supérieure à 128×128 pixels. Nous avons sélectionné les 238 imagettes de reine respectant ce critère. Nous devions sélectionner la même quantité d'imagettes d'ouvrière pour avoir un nombre équilibré d'images appartenant à chaque classe. Les imagettes d'ouvrière ont été enregistrées en conservant une trace de l'image originale d'où elle provient. Ainsi, nous avons sélectionné les 238 ouvrières de manière stratifiée afin de bien représenter les images originales.

Les images sont toutes carrées, mais n'ont pas nécessairement la même dimension. Les 476 images retenues ont été redimensionnées à 128×128 pixels.

4.2 PROTOCOLE D'EXPÉRIMENTATION

Rappelons d'abord les deux questions de recherche de ce mémoire. Nous voulions déterminer si c'est possible de détecter automatiquement la présence d'une reine sur un cadre de ruche pour remplacer cette tâche normalement manuelle. Nous voulions aussi comparer deux méthodes de classification d'images : une méthode classique utilisant les **SVM** et une méthode par apprentissage profond utilisant les **CNN**.

4.2.1 ENVIRONNEMENT MATÉRIEL

Pour les expérimentations de la méthode classique, nous avons utilisé les bibliothèques *Scikit-image 0.18.0* [1] (Van der Walt et al. (2014)) et *OpenCV 4.5.0* [2] (Bradski (2000)) spécialisées dans le traitement d'images. L'exécution est faite sur un ordinateur de bureau.

Pour les expérimentations de la méthode par apprentissage profond, l'environnement d'exécution utilisé est *Google Calab* (Google (2021)), une solution de *Google* pour effectuer de l'apprentissage automatique en utilisant l'infonuagique. Nous avons utilisé la librairie *Tensorflow v2.4.1* [3] (Abadi et al. (2016)), une librairie pour l'apprentissage automatique développé par *Google* et écrite en *Python 3.8* [4] (python (2019)). Dans *Google Colab*, nous utilisons les processeurs de *Google*. Pour ce projet, nous avons utilisé le **TPU** de *Google* (*Tensor Processing Unit*), spécialement conçu pour accélérer les calculs avec *Tensorflow*.

Pour l'application de découpe d'images en imagettes, l'environnement utilisé est *IntelliJ IDEA* (Jet Brains (2021)). Nous avons aussi utilisé le framework *Electron* (Jasim (2017)) pour développer l'application.

4.2.2 EXPÉRIMENTATIONS

En utilisant le jeu de données de 476 imagettes, nous avons effectué 2 expérimentations. Les résultats de la première expérimentation ont été publiés par Marquis et al. (2021). Le modèle classique a été exécuté à 5 reprises. Les résultats représentent une moyenne sur ces 5 reprises. Pour la méthode utilisant un **CNN**, un seul test a été effectué. Une deuxième expérimentation a été effectuée en concervant plus de détails. Un seul test pour chaque méthode a été exécuté.

1. `https://scikit-image.org/docs/0.18.x/`
2. `https://opencv.org/opencv-4-5-0/`
3. `https://www.tensorflow.org/versions/r2.4/api_docs/python/tf?hl=uk`
4. `https://www.python.org/downloads/release/python-380/`

4.2.3 STRATÉGIE DE VALIDATION

La même stratégie d'évaluation a été utilisée pour les méthodes proposées. Nous avons opté pour la populaire méthode *10-fold cross-validation* qui a été décrite à la section 1.4. Les images sont divisées en dix dossiers. À tour de rôle, chaque dossier constitue l'ensemble de validation alors que les neuf dossiers restants servent d'ensemble d'entraînement. Ainsi, dix jeux de données sont créés afin d'effectuer autant de tests pour chacun de nos modèles.

À chaque itération de la validation croisée, le modèle donnant le meilleur taux de reconnaissance est conservé. Nous obtenons donc 10 taux de reconnaissance. Une moyenne de ces taux permet d'obtenir la précision du modèle. La moyenne obtenue pour chaque modèle peut ensuite être comparée.

Une matrice de confusion est construite pour une meilleure comparaison entre les méthodes. Ainsi, nous pouvons non seulement connaître le taux de reconnaissance du modèle, mais aussi le taux de reconnaissance pour les reines et pour non reines. Notons que pour chaque expérimentation et pour chaque méthode, les dix dossiers d'images pour la validation croisée sont formés aléatoirement.

4.3 RÉSULTATS

Nous décrivons d'abord les résultats généraux obtenus. Nous retrouvons ensuite une section pour chaque méthode proposée. Finalement, une analyse des résultats est présentée.

Les deux méthodes proposées, correspondant à la première expérimentations, ont été publiées dans un article scientifique par (Marquis et al. (2021)). Celui-ci a été publié dans les actes de la *11th international conference on robotic, vision, signal processing and power applications*. Les

résultats sont présentés à la figure 4.2 et au tableau 4.1. Le taux de reconnaissance correspond au pourcentage d'images de reine et pas reine bien identifiées. Pour la méthode classique, les résultats ont été obtenus sur une moyenne de cinq tests. Nous avons effectué des tests pour les images en couleurs et en ton de gris et nous avons obtenu un taux de reconnaissance de 84.05% et de 83.8%, respectivement. Pour la méthode avec les **CNN**, nous avons obtenu un résultat moyen de 89.71%.

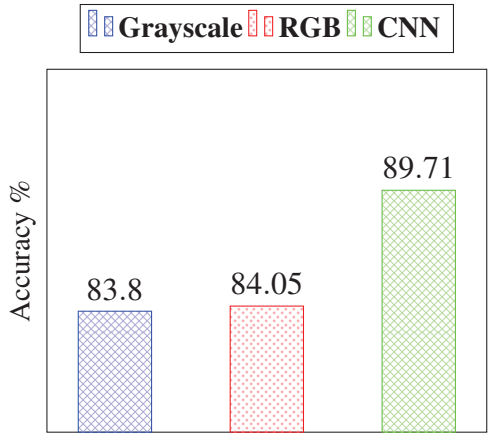


FIGURE 4.2 – Taux de reconnaissance pour la première expérimentation.

		Queen	*Worker*
Grayscale	*Queen*	**81.09**%	18.91%
	Worker	13.45%	**86.55**%
RGB	*Queen*	**83.61**%	13.39%
	Worker	15.55%	**84.45**%
CNN	*Queen*	**88.24**%	13.46%
	Worker	10.92%	**91.18**%

TABLEAU 4.1 – Matrice de confusion pour la première expérimentation.

Une autre expérimentation, nommée deuxième expérimentation, a été effectuée pour chacune des méthodes afin de pouvoir offrir plus de détails. Les résultats diffèrent de ce qui a été obtenu

dans l'article puisque les images sont distribuées aléatoirement pour former dix dossiers distincts. Les taux de reconnaissance et la matrice de confusion de chaque méthode sont présentés à la figure 4.3 et au tableau 4.2. Pour la méthode classique, nous avons obtenu un taux de reconnaissance moyen de 87.4% pour les images en ton de gris et de 86.98% pour les images en couleurs. Pour la méthode par apprentissage profond, nous avons obtenu un taux moyen de 90.12%. Les détails pour chaque méthode sont décrits dans les deux sections suivantes.

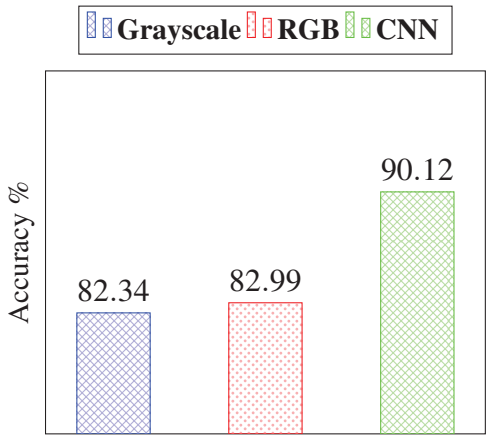


FIGURE 4.3 – Taux de reconnaissance pour la deuxième expérimentation.

		Queen	*Worker*
Grayscale	*Queen*	**80.25**%	19.75%
	Worker	15.55%	**84.45**%
RGB	*Queen*	**81.93**%	18.07%
	Worker	15.97%	**84.03**%
CNN	*Queen*	**89.50**%	10.50%
	Worker	9.24%	**90.76**%

TABLEAU 4.2 – Matrice de confusion pour la deuxième expérimentation.

4.3.1 RÉSULTATS SVM

Pour la méthode par l'approche classique, rappelons que nous avions utilisé le **PCA** comme technique pour réduire la dimensionnalité. Pour un nombre de composantes principales donné, nous avons calculé le taux de reconnaissance moyen sur les 10 itérations de la validation croisée. Comme mentionné plus haut, les expérimentations dans le cadre de l'article, à la figure 4.4, font intervenir une moyenne sur 5 tests. Pour la deuxième expérimentation, nous avons effectué un seul test, dont les résultats sont à la figure 4.5. Les résultats présentés ont été tronqués puisque les meilleurs taux de reconnaissance sont obtenus lorsqu'un nombre relativement petit de composantes principales est utilisé.

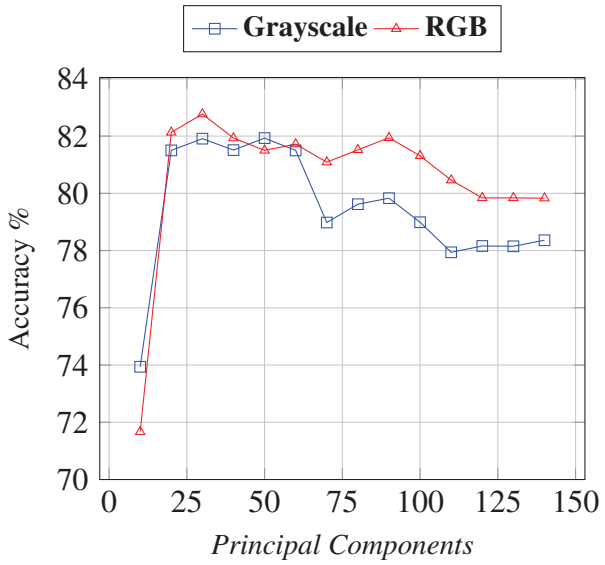


FIGURE 4.4 – Taux de reconnaissance moyen en fonction du nombre de composantes principales pour la première expérimentation.

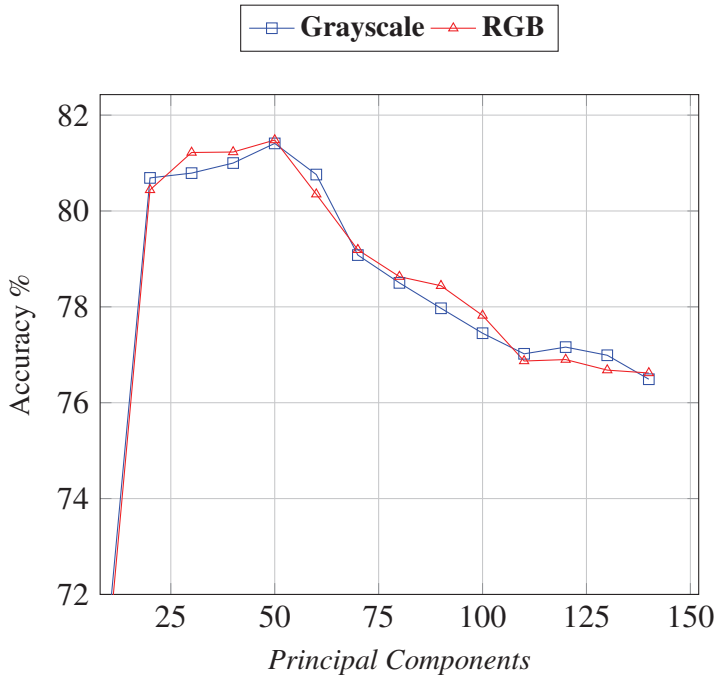


FIGURE 4.5 – Taux de reconnaissance moyen en fonction du nombre de composantes principales pour la deuxième expérimentation.

Le tableau 4.3 présente les meilleurs taux de reconnaissance pour chaque itération de la validation croisée de la deuxième expérimentation. Pour ces 10 itérations, nous retrouvons aux figures 17 à 26 de l'annexe .2.1 le taux de reconnaissance en fonction du nombre de composantes principales.

Itération	Nombre de composantes - RGB	Taux - RGB	Nombre de composantes - Grayscale	Taux - Grayscale
1	27	87.50%	27	87.50%
2	10	89.58%	23	91.67%
3	19	93.75%	17	91.67%
4	45	85.42%	101	85.42%
5	15	83.33%	18	85.42%
6	19	81.25%	24	81.25%
7	11	80.85%	17	82.98%
8	36	93.62%	55	93.62%
9	52	82.98%	58	82.98%
10	91	91.49%	107	91.49%
Moyenne	**32.5**	86.98%	**44.7**	87.40%

TABLEAU 4.3 – Meilleurs taux de reconnaissance selon le nombre de composantes des 10 itérations de la validation croisée pour la deuxième expérimentation de la méthode classique.

4.3.2 RÉSULTATS CNN

Débutons d'abord par ce qui a été publié dans l'article. Nous avions les résultats présentés à la figure 4.2 et au tableau 4.1. Nous avons fait la moyenne des dix itérations de la validation croisée pour chaque *epoch*. Nous obtenons la figure 4.6.

Présentons maintenant les détails de la deuxième expérimentation. Pour chaque *epoch*, nous avons effectué la moyenne sur les 10 itérations de la validation croisée, tout comme nous l'avions fait avec l'expérience présentée dans l'article. Nous obtenons la figure 4.7. Nous

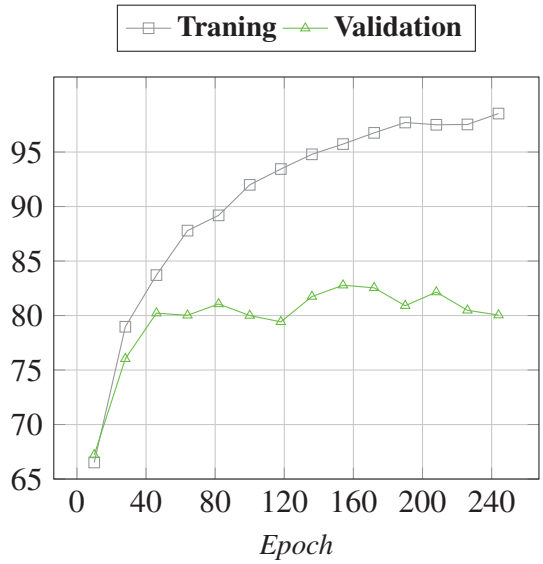


FIGURE 4.6 – Taux de reconnaissance moyen sur les 10 itérations de la validation croisée en fonction de l'*epoch* pour la première expérimentation.

avons ajouté un graphique (voir la figure 4.8) représentant la perte moyenne en fonction de l'*epoch*.

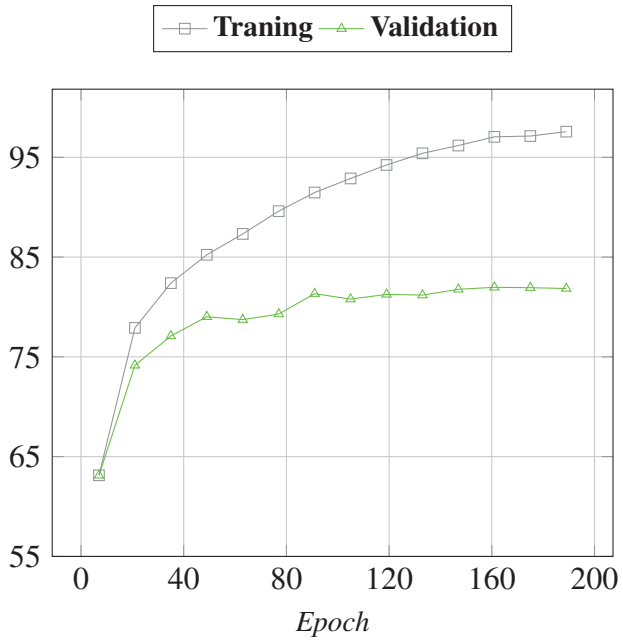


FIGURE 4.7 – Taux de reconnaissance moyen sur les 10 itérations de la validation croisée en fonction de l'*epoch* pour la deuxième expérimentation.

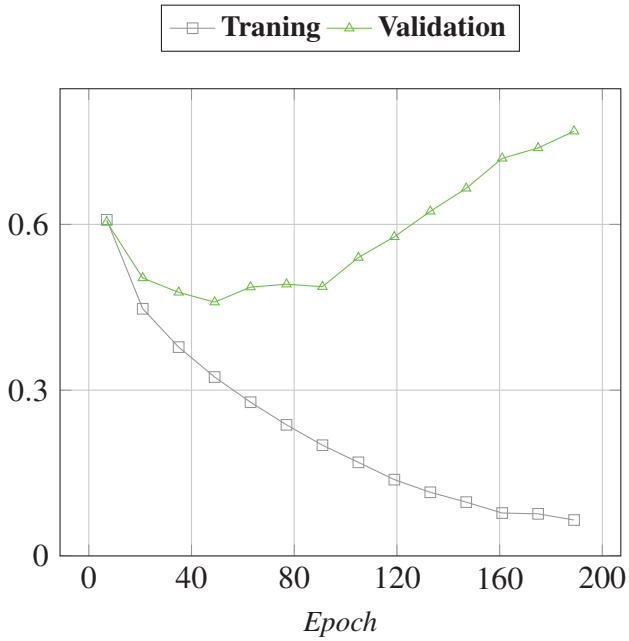


FIGURE 4.8 – Perte moyenne sur les 10 itérations de la validation croisée en fonction de l'*epoch* pour la deuxième expérimentation.

Au tableau 4.4, nous présentons, pour chaque itération, l'*epoch* où le modèle a obtenu le meilleur taux de reconnaissance. Le taux de reconnaissance moyen des dix itérations est de 90.12%. Le meilleur modèle a été obtenu, en moyenne, au 99^{e} *epoch*. À la suite de ce tableau, nous retrouvons des détails pour chaque itération de la validation croisée. En effet, les figures 27 à 46 de l'annexe .2.2 représentent le taux de reconnaissance et la perte de chaque itération en fonction de l'*epoch*.

Itération	Epoch	Taux
1	21	83.33%
2	199	93.75%
3	48	91.67%
4	89	93.75%
5	91	93.75%
6	112	93.75%
7	120	95.75%
8	83	87.23%
9	94	85.11%
10	156	82.98%
Moyenne	**99**	90.12%

TABLEAU 4.4 – Meilleurs taux de reconnaissance selon *l'epoch* des 10 itérations de la validation croisée pour la deuxième expérimentation de la méthode par apprentissage profond.

4.3.3 ANALYSE DES RÉSULTATS

Les figures 4.2 et 4.3 présentent le taux de reconnaissance moyen pour chaque méthode. Lorsque nous comparons les deux méthodes proposées, nous utilisons comme seul critère le taux de reconnaissance, c'est-à-dire le pourcentage d'images classées dans la bonne catégorie par le modèle. Le temps d'exécution et l'espace de stockage n'ont pas été pris en compte dans ce projet. Dans les deux expérimentations, la méthode utilisant les **CNN** offre un meilleur taux de reconnaissance par rapport à la méthode classique. Dans le cas des résultats de la première expérience, nous avions un taux de reconnaissance de 89.71% alors que la méthode classique offrait un résultat de 83.8% pour les images en ton de gris et de 84.05% pour les images en couleurs. L'écart entre les deux méthodes est donc de 5.66%. Pour la deuxième expérience, la différence entre la méthode classique et la méthode par apprentissage profond est de 2.72%. La méthode par apprentissage profond a offert un taux de reconnaissance de 90.12% alors que les taux pour les méthodes classiques en ton de gris et en couleurs sont respectivement de 82.34% et 82.99%.

La différence entre les résultats de la méthode classique, soit les images en ton de gris et les images en couleur, est de 0.25% pour la première expérimentation et de 0.65% pour la deuxième expérimentation. Portons notre attention aux matrices de confusion présentées aux tableaux 4.1 et 4.2. Pour le premier tableau, le taux de reconnaissance de la reine pour les images en ton de gris est de 81.09% et celui pour les images en couleur est de 83.61%. Il y a une différence de 2.52%. Pour le deuxième tableau, le taux de reconnaissance de la reine pour les images en ton de gris est de 81.09% et de 83.61% pour les images en couleurs. La différence est de 1.68%. Puisque l'objectif est de détecter la présence ou l'absence de la reine, la différence pour les images en couleurs nous permet de croire que les images en couleurs sont plus riches en information et sont intéressantes à considérer.

Les graphiques aux figures 4.4 et 4.5 représentent la moyenne des 10 itérations de la validation croisée pour chaque nombre de composantes principales. Nous constatons, pour la première expérimentation, que le taux de reconnaissance atteint un maximum lorsque le nombre de composantes principales est d'environ 25. Pour la deuxième expérience, le taux de reconnaissance est maximal quand le nombre de composantes principales est d'environ 50. La différence entre les deux expériences est due à la formation aléatoire des 10 dossiers pour la validation croisée.

Pour la deuxième expérimentation de la méthode classique, nous avons exposé plus de détails. Pour chaque itération de la validation croisée, le taux de reconnaissance maximal a été prélevé ainsi que le nombre de composantes principales avec lequel ce taux a été atteint. Nous obtenons un taux moyen de 86.98% pour les images de couleur et de 87.40% pour les images en ton de gris. Bien que le taux de reconnaissance moyen pour les images en ton de gris soit plus élevé, il est atteint en moyenne avec un nombre de composantes plus élevé. Ces résultats diffèrent de ceux présentés plus haut. Effectivement, pour la figure 4.3, nous avions fait la moyenne sur les 10 itérations de la validation croisée pour chaque nombre de composantes principales. Il est intéressant de voir la différence entre les deux taux moyens. Il faudra approfondir les expérimentations pour les images en couleurs par rapport aux images en niveau de gris afin de mieux comprendre l'impact sur le taux de reconnaissance.

En analysant les graphiques aux figures 17 à 26, nous apercevons une tendance : une augmentation rapide du taux de reconnaissance est visible pour un nombre petit de composantes principales. La progression du taux de reconnaissance est plus petite par la suite et, dans la majorité des itérations, diminue lorsque le nombre de composantes principales augmente.

Pour la méthode avec les **CNN**, nous avons d'abord construit un graphique qui représente, pour chaque *epoch*, le taux de reconnaissance moyen sur les 10 itérations de la validation croisée. Nous remarquons à la figure fig 4.6 que le taux de reconnaissance pour l'ensemble

de validation se stabilise à environ 50 *epoch*. Le taux de reconnaissance pour l'ensemble d'entraînement continue d'augmenter au-delà de 50 *epoch*, ce qui peut être le signe d'un sur apprentissage pour le modèle.

Pour vérifier cette intuition, une deuxième expérimentation a été faite et les détails de celle-ci sont plus nombreux. La figure 4.7 est homologue à la figure 4.6. Tout comme la première expérimentation, le taux de reconnaissance pour l'ensemble de validation semble converger, mais cette fois autour de 90 *epochs*. Le taux pour l'ensemble d'entraînement continue d'augmenter. L'hypothèse du surapprentissage du modèle semble se forger avec l'analyse du graphique à la figure 4.8. Celui-ci représente la perte moyenne de chaque itération de la validation croisée en fonction de l'*epoch*. Au même nombre d'*epochs* où le taux d'apprentissage semblait converger, la perte commence à augmenter pour l'ensemble de validation et diminue pour l'ensemble d'entraînement. Ce phénomène est typique lorsque le modèle fait du surapprentissage.

Le tableau 4.4 expose le taux de reconnaissance maximal moyen (90.12%) obtenu pour chaque itération de la validation croisée. En moyenne, ce taux est atteint au 99^{e} *epoch*. Les figures 27 à 46 montrent les détails de l'entraînement des modèles pour les 10 itérations de la validation croisée. Nous observons que chaque modèle semble effectivement faire du surapprentissage. Il serait intéressant d'utiliser des astuces ou d'autres techniques qui pourraient prévenir ce surapprentissage et améliorer les résultats.

4.4 CONCLUSION

En conclusion, nous avons présenté la méthode pour le traitement des images afin de créer des imagettes. L'application développée par notre équipe a été décrite en annexe. Nous avons opté pour la validation croisée comme stratégie de validation. Les modèles ont été évalués en terme de précision seulement. Bien que la méthode classique avec les **SVM** est plus rapide en

temps d'exécution et nécessite du matériel moins performant, nous avons obtenu de meilleurs résultats lorsque nous utilisions la méthode avec les **CNN**.

CONCLUSION ET PERSPECTIVES

Un apiculteur doit savoir si la reine est présente ou non dans la ruche. Sa présence a un rôle crucial pour la survie de la colonie. Certaines techniques, par exemple par le son, permettent de répondre à ce problème. Parfois, nous devons savoir exactement où se trouve la reine dans la ruche. Il existe une méthode physique qui consiste à faire passer les abeilles dans un filtre. Ce filtre laisse passer les ouvrières, mais pas la reine. Cette technique perturbe la colonie. Une autre méthode consiste à retirer les cadres un à un et localiser la reine. Dans ce projet, nous voulions automatiser cette tâche de détection de la reine dans la ruche en prenant des images de cadre d'abeilles.

Ce projet est concluant puisqu'il a été possible de détecter la présence d'une reine à partir d'une imagette extraite du cadre de la ruche. Ce projet a contribué à la création de deux modèles de classification d'images d'abeilles en deux classes : reine ou pas reine. Nous avons obtenu un taux de reconnaissance de 84.05% pour la méthode classique et de 90.12% pour la méthode par apprentissage profond. Nos résultats viennent confirmer nos deux questions de recherche. Il est possible d'automatiser la tâche de l'apiculteur qui consiste à rechercher la reine sur un cadre d'abeilles. De plus, les méthodes par apprentissage profond offrent un meilleur taux de reconnaissance que les méthodes classiques.

La deuxième contribution du projet est la création, en collaboration avec Guillaume Gingras, d'une application qui permet de découper une image en imagettes et d'identifier la classe à laquelle l'imagette appartient. Cette contribution marginale a permis d'augmenter la base de données et de faciliter l'identification des images.

La première perspective de ce projet est de rendre publique la base de données que nous avons créée. Nous pourrons partager le jeu de données utilisé pour l'entraînement de nos deux modèles. Nous pourrons aussi partager les images complètes ainsi que les imagettes de reine et d'ouvrières. Les images pourront être reprises afin que d'autres puissent faire des expérimentations et comparer leurs résultats aux nôtres.

Seulement deux méthodes ont été analysées : l'approche classique avec un **SVM** et l'approche par apprentissage profond avec les **CNN**. Nous voulons explorer d'autres méthodes. Pour la méthode utilisant les **CNN**, nous testerons d'autres valeurs pour la taille et le nombre de convolutions, la taille de fenêtre pour le *pooling*, la taille et le nombre de couches *denses*, le taux d'apprentissage, la fonction de perte ainsi que l'optimiseur.

Nous voulons développer une application *open source* qui intègrerait notre meilleur modèle. Cette application permettrait notamment de prendre en photo un cadre d'abeilles et confirmer ou non la présence de la reine. Cette application pourrait aussi détecter et découper le cadre sur l'image ainsi que détecter et localiser précisément la reine. En élargissant nos travaux plus loin que la détection de la présence de la reine, l'application pourrait permettre la détection de la présence d'un parasite, comme la *Varroa*, dans la colonie. Dans ce projet, seul le taux de reconnaissance a été pris en compte lors de la comparaison des deux modèles. Puisque nous avons pour objectif de les intégrer à l'application, nous devrons prendre d'autres paramètres en compte comme le temps d'exécution, le temps d'entraînement et l'espace de stockage.

Au début du projet, autant que nous sachions, il n'existait pas d'applications gratuites et

open source incluant l'option de détection de la reine. Plusieurs applications d'aide pour les apiculteurs ont été créées depuis, mais n'offrent pas beaucoup d'information quant aux algorithmes implémentés. Il serait intéressant d'entrer en contact avec les administrateurs de ces applications afin de créer des collaborations. Même si de telles applications existent, la nôtre pourrait être adaptée à la réalité des apiculteurs de la région. En effet, les abeilles que l'on retrouve au Québec ne sont pas forcément de la même sous-espèce que celles que l'on retrouve en Europe.

Dans cet esprit collaboratif, il serait intéressant de rencontrer les apiculteurs locaux afin de leur offrir de s'intégrer au projet. Ils pourraient alimenter la base de données en prenant des photos de leurs cadres de ruche. Ils pourraient aussi nous décrire leurs problématiques et nous dire en quoi nous pourrions aider à améliorer leur travail.

ANNEXES

.1 CRÉATION DES IMAGETTES

Dans cette section, nous retrouvons les détails pour la création des imagettes de reine et des imagettes d'ouvrière.

.1.1 IMAGETTES DE REINE

Pour une image dans *original_queen* (voir la figure 9a), les opérations suivantes ont été effectuées :

1. Nous avons utilisé le logiciel de traitement d'images *paint.net* (paint.net (2021)). Avec l'outil de sélection, une zone carrée a été créée autour de la reine. La taille n'est par considérée pour l'instant. Il s'agit à cette étape d'obtenir un encadrement qui laisse paraître entièrement la reine, lorsque ceci est possible. Nous excluons une imagette de reine si une partie de la reine est à l'extérieur du cadre de l'image. Certaines reines sont cachées derrière des ouvrières. Nous conservons quand même ces images puisque cela reflète ce qui pourrait se produire dans la réalité. (voir figure 9b)
2. La sélection est copiée dans un nouveau fichier qui est enregistré dans un dossier portant

le nom *queen_isolated*. Ce dossier renferme donc toutes les imagettes de reines. (voir figure 9c)

3. De retour sur l'image originale, la sélection est supprimée et remplacée par un carré blanc. S'il reste une ouvrière sur l'image, une copie de ce résultat est enregistrée dans le dossier *no_queen* qui sera utilisé par l'application de découpe. Cela nous permettra de récupérer les ouvrières qui se trouvaient aussi sur l'image de la reine. (voir figure 9d)

(a) Image originale contenant une reine.

(b) Sélection carrée autour de la reine.

(c) Enregistrement de l'imagette de la reine.

(d) Image sans le carré de la reine.

FIGURE 9 – Étapes appliquées à chaque image contenant une reine.

.1.2 IMAGETTES D'OUVRIÈRES

Suite à la création des imagettes de reine, nous retrouvons un dossier nommé *no_queen* qui contient les images dont la reine est remplacée par un carré blanc. Ce dossier, ainsi que le dossier *original_worker* ont été traités dans l'application afin d'obtenir des imagettes d'ouvrière. À l'ouverture, nous remarquons le menu principal à gauche. L'onglet *Home* est automatiquement sélectionné (figure 10).

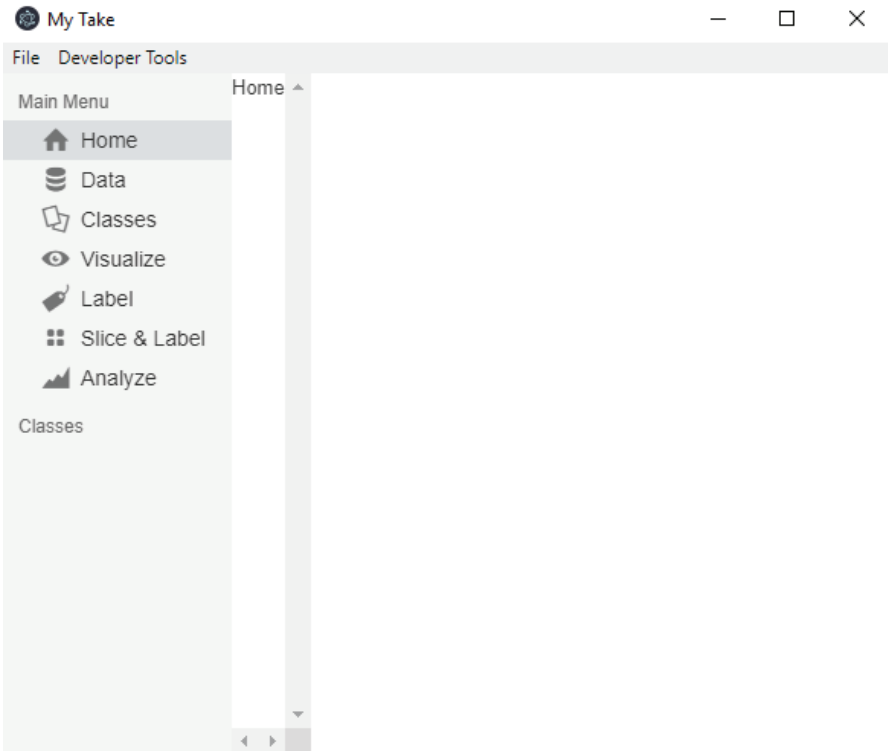


FIGURE 10 – Application de découpe et d'identification : Écran d'accueil.

Nous devons cliquer sur *Data* afin de choisir le dossier d'entrée qui contient les images que l'on veut découper en imagettes. Nous voit apparaître le chemin, le format des images, qui doit être *jgp*, et le nombre total d'images à découper (figure 11).

Nous devons ensuite entrer les classes. Elles permettront de catégoriser les imagettes. Le nom des classes est accompagné d'une couleur. La classe *complete* est sélectionnée quand nous

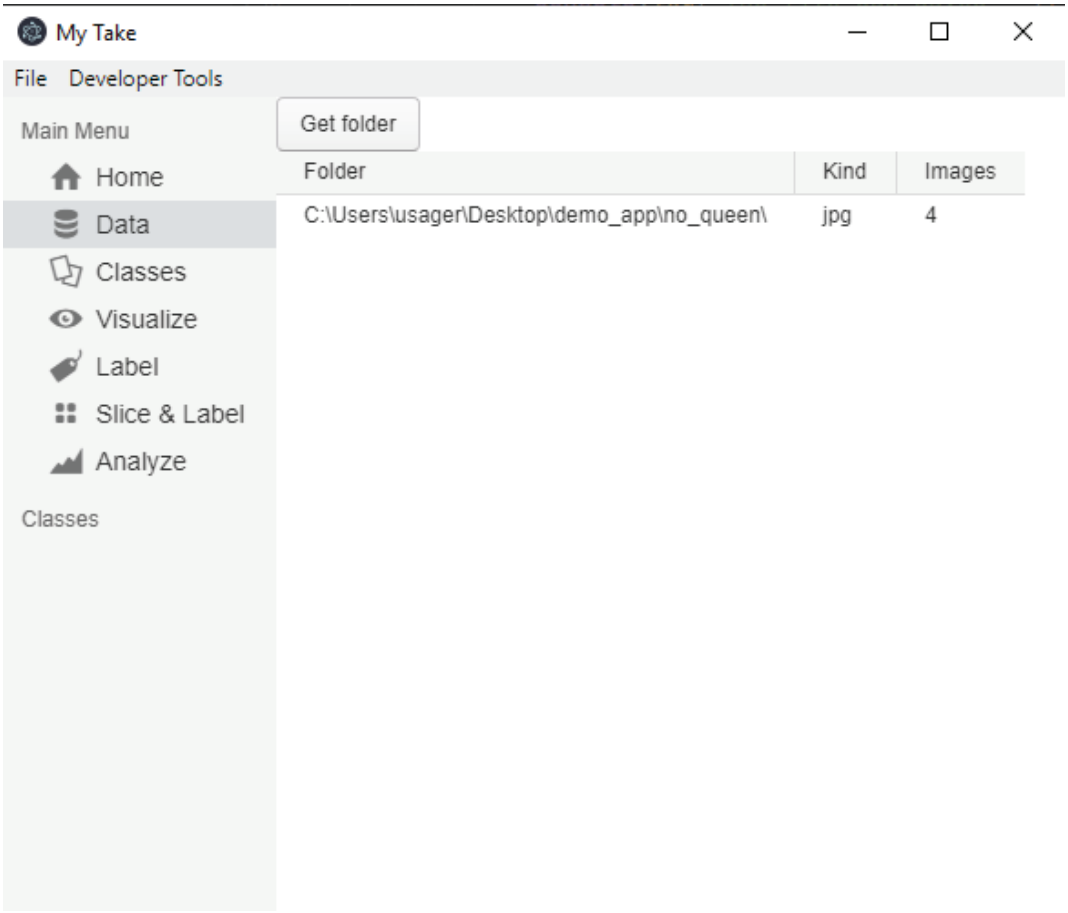


FIGURE 11 – Application de découpe et d'identification : Sélection du dossier contentant les images à découper en imagettes.

retrouvons une ouvrière entière dans l'imagette. La classe *part* est utilisée pour les imagettes contenant seulement une partie d'ouvrières. Cette dernière pourrait servir dans une future expérience. La classe *other* est utilisée lorsque l'on ne retrouve aucune ouvrière ou qu'il y a des ajouts électroniques, par exemple une partie du carré blanc laissé par le retrait de la reine (figure 12).

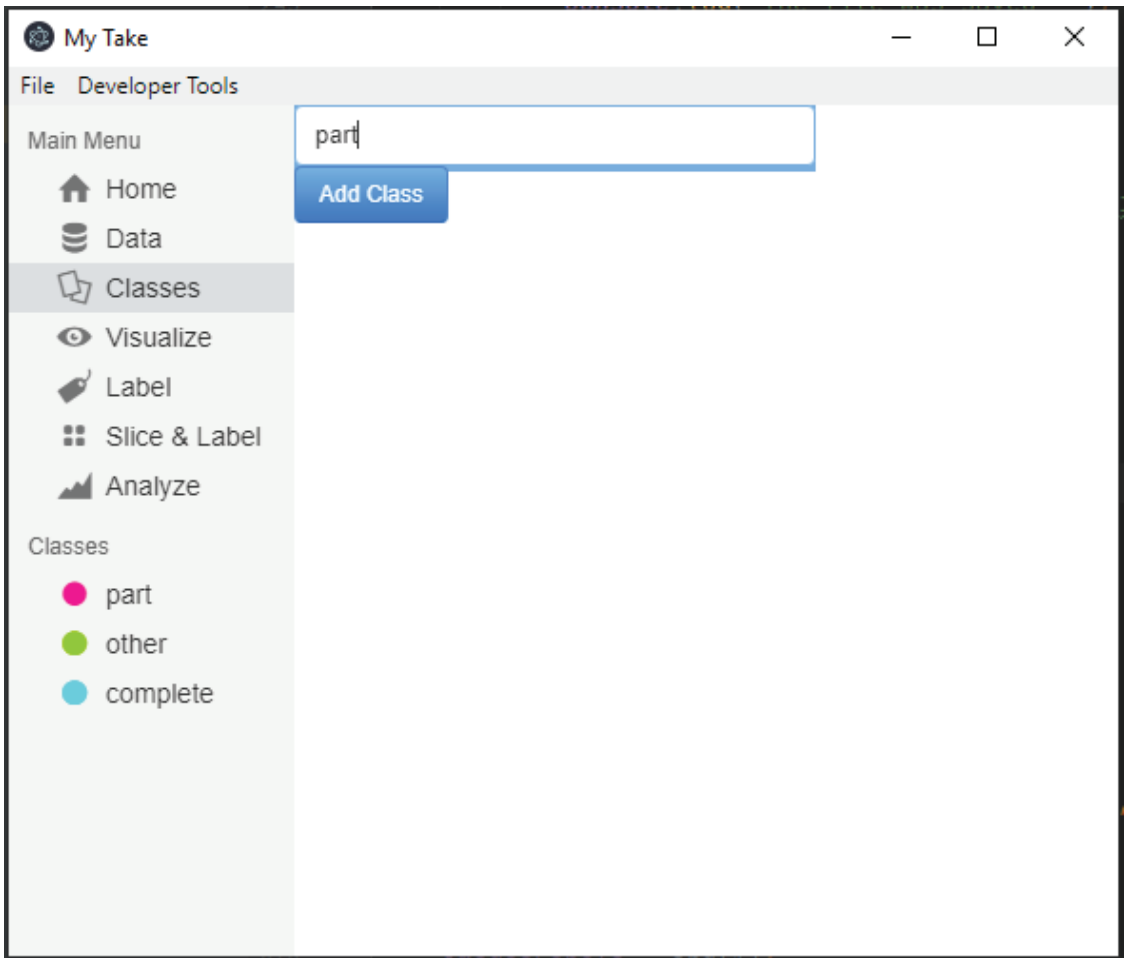


FIGURE 12 – Application de découpe et d'identification : Les classes dans lesquelles sont divisées les imagettes.

L'onglet *Visualize* permet de visualiser les images du dossier d'entrée. L'onglet *Label* est utilisé si l'on veut identifier une image complète selon une des classes établies, sans avoir à découper en imagettes. Enfin, l'onglet *Slice & Label* est celui qui divise l'image en imagettes et permet l'identification des imagettes selon les classes. Il faut d'abord choisir un dossier de sortie préalablement créé où les imagettes seront classées. Une fois le dossier de sortie sélectionné, nous voyons apparaître la première image à diviser (figure 13).

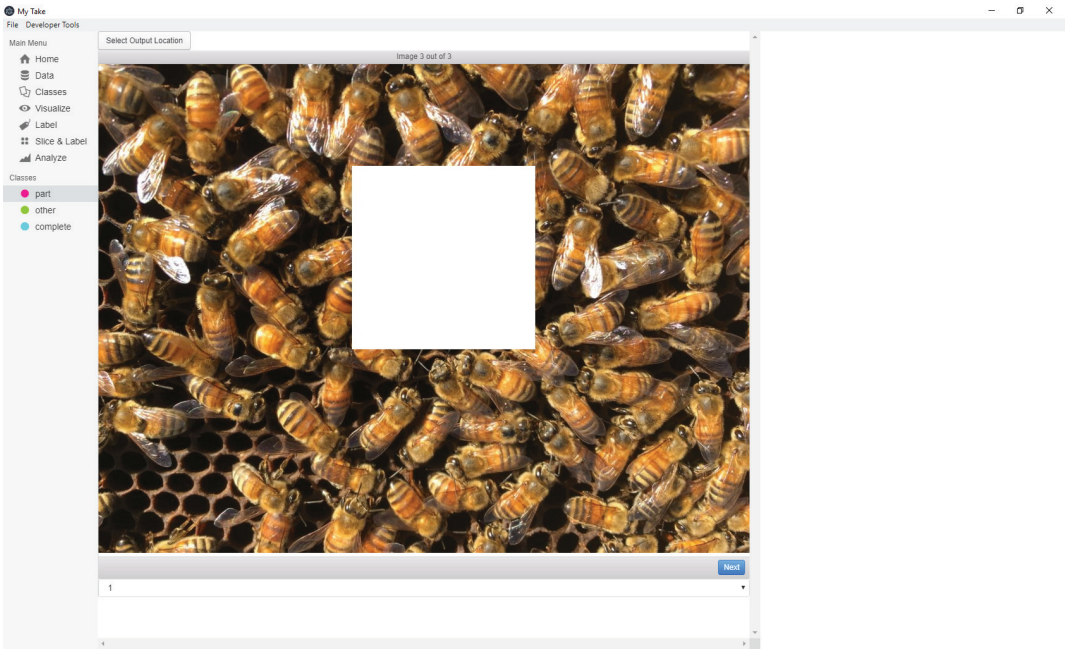


FIGURE 13 – Application de découpe et d'identification : Image sans le carré de la reine. L'option *Slice & Label* permet de diviser cette image en imagettes et d'identifier les imagettes selon les classes.

Dans le dossier de sortie, un nombre de dossiers égal au nombre de classes est créé. Ce sont dans ces dossiers que les imagettes sont classées. Un fichier temporaire est aussi créé. Les imagettes y sont conservées en attendant de passer à l'image suivante. Finalement, nous retrouvons un fichier *.txt* qui sert à mémoriser quelles images ont été découpées. Nous pouvons ainsi quitter l'application et recommencer la découpe des images à un moment ultérieur. La figure 14 présente les différents dossiers et fichiers créés dans le fichier de sortie.

Décrivons maintenant plus en détail la fonctionnalité *Slice & Label* de l'application. Les imagettes créées sont carrées et celles issues d'une même image sont de la même taille. Un menu déroulant permet de sélectionner un nombre, n, qui correspond au maximum d'imagettes qui sont formées sur le plus petit côté de l'image. Ainsi, il y aura au minimum $n \times n$ imagettes formées. Il y en a plus si l'image est de forme rectangulaire et qu'il est possible de former des imagettes supplémentaires sur le plus grand côté de l'image. Les imagettes non carrées

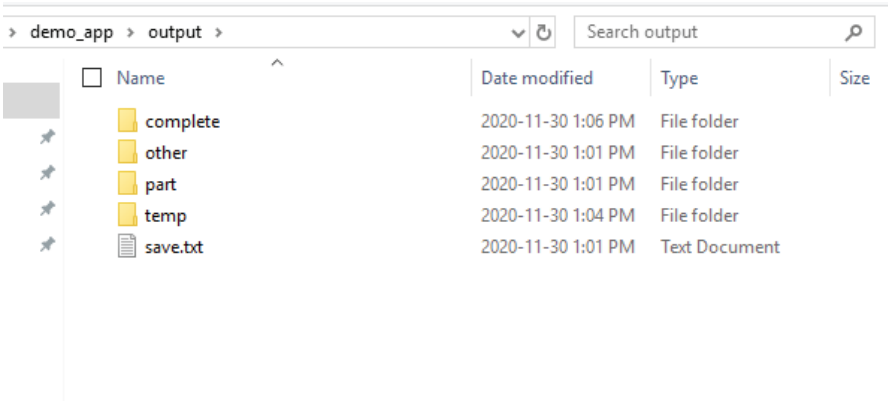


FIGURE 14 – Application de découpe et d'identification : Le fichier de sortie comprend les dossiers associés aux classes pour conserver les imagettes identifiées, un dossier temporaire pour contenir les découpes non terminées et un fichier .txt pour garder une trace des images déjà découpées.

résultant des bordures sont exclues. Un exemple de découpe est présenté à la figure 15. Plusieurs découpes peuvent être testées avant de choisir la découpe finale. Celle retenue engendrera le plus grand nombre d'imagettes contenant une ouvrière entière.

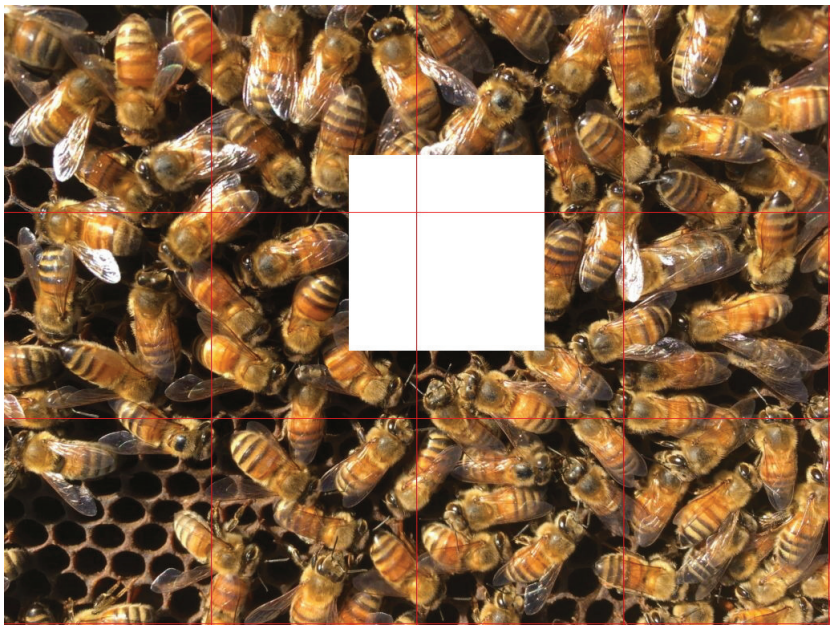


FIGURE 15 – Cette image a 1030×773 pixels. La largeur de l'image est divisée afin d'obtenir 3 imagettes. Il reste 1 pixel dans la bordure du bas qui est exclu. Il y a 4 imagettes sur la longueur et il reste 2 pixels dans la bordure de droite qui sont exclus. Il y a donc 12 imagettes générées au total.

Lorsque la découpe est choisie, il faut passer à l'identification des imagettes. Il suffit de cliquer sur la classe et sélectionner les imagettes qui y sont associées. La couleur du contour de l'imagette est la couleur associée à la classe. Lorsque l'identification est terminée, nous pouvons passer à l'image suivante. Les imagettes sont enregistrées dans les dossiers des classes. Le fichier *.txt* est mis à jour pour y inclure la dernière image traitée. Si une image n'a pas été identifiée, un message apparaît pour indiquer le numéro de l'imagette oubliée. La figure 16 présente une capture d'écran de l'application.

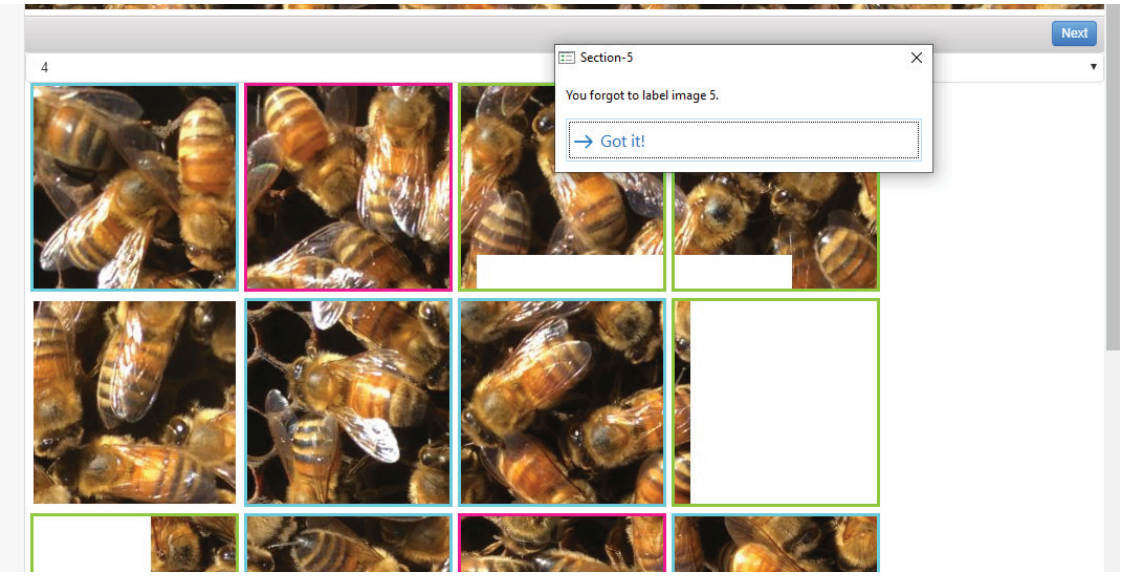


FIGURE 16 – L'utilisateur a choisi d'avoir 4 imagettes sur le côté le plus court de l'image. Un contour bleu représente une ouvrière complète. Un contour rose identifie une imagette où une ouvrière apparaît partiellement. Un contour vert est utilisé pour les imagettes n'ayant pas d'ouvrière ou ayant une partie du carré blanc. Lorsqu'une image n'a pas été identifiée selon une des classes, un message est envoyé à l'utilisateur.

.2 GRAPHIQUES DES RÉSULTATS DÉTAILLÉS

Nous retrouvons, aux deux prochaines sections, les graphiques des 10 itérations de la validation croisée pour la méthode classique et pour la méthode pas apprentissage profond.

.2.1 MÉTHODE CLASSIQUE

Voici les graphiques détaillés pour les 10 itérations de la validation croisée de la méthode classique.

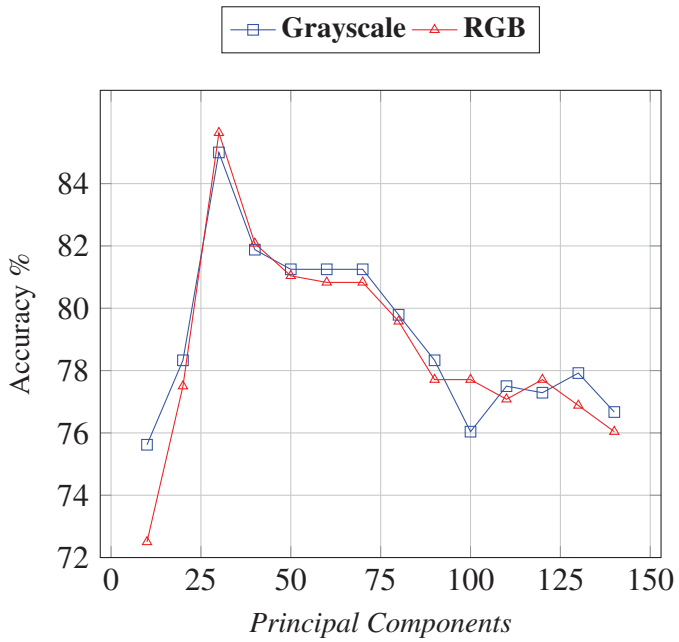


FIGURE 17 – Itération 1 : Taux de reconnaissance en fonction du nombre de composantes principales pour la deuxième expérimentation de la méthode classique.

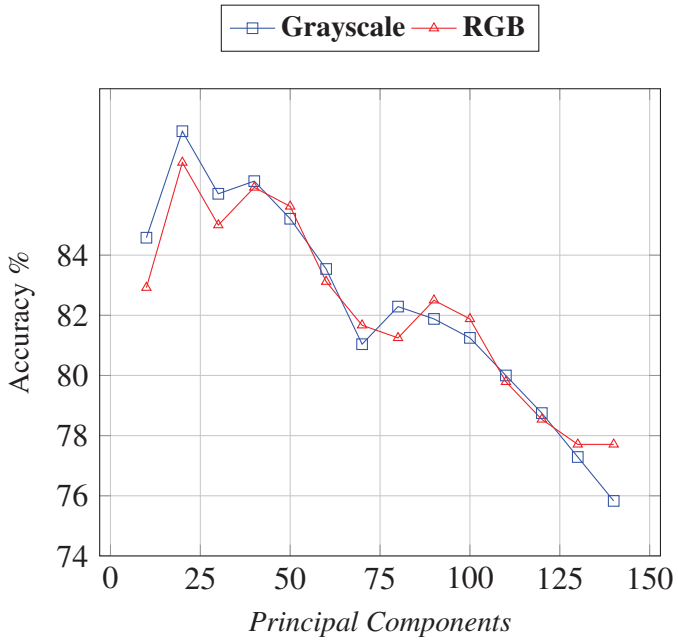


FIGURE 18 – Itération 2 : Taux de reconnaissance en fonction du nombre de composantes principales pour la deuxième expérimentation de la méthode classique.

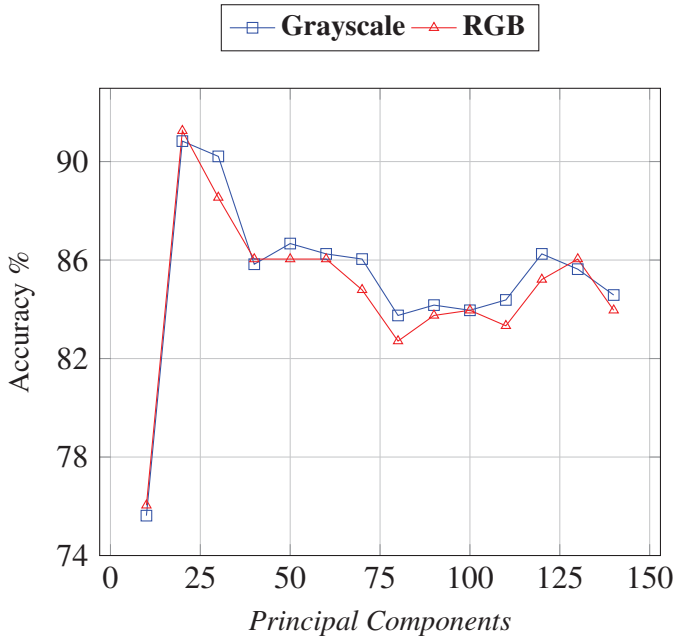


FIGURE 19 – Itération 3 : Taux de reconnaissance en fonction du nombre de composantes principales pour la deuxième expérimentation de la méthode classique.

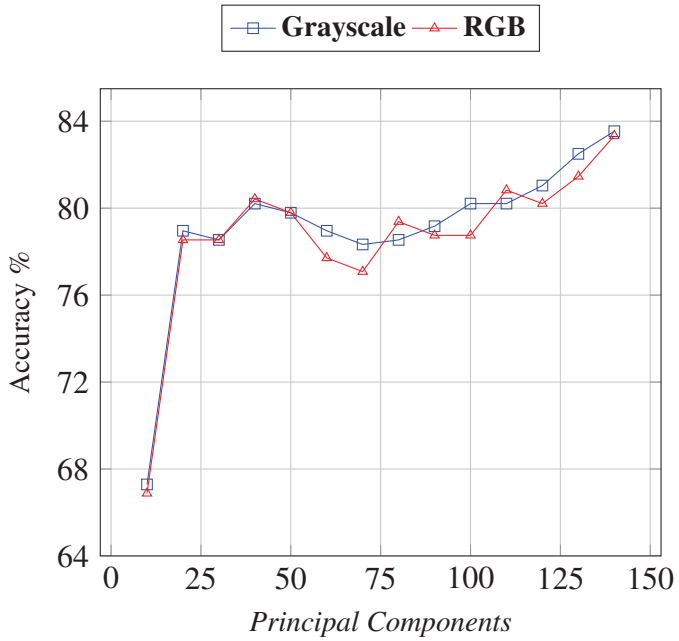


FIGURE 20 – Itération 4 : Taux de reconnaissance en fonction du nombre de composantes principales pour la deuxième expérimentation de la méthode classique.

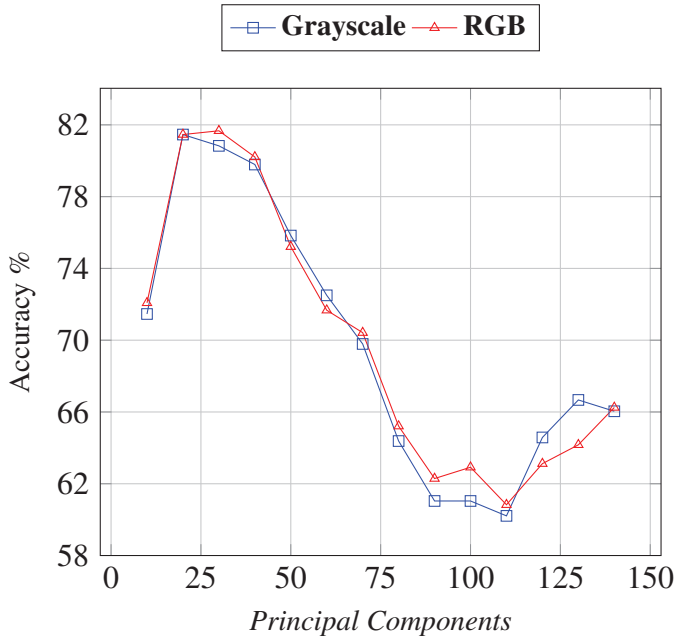


FIGURE 21 – Itération 5 : Taux de reconnaissance en fonction du nombre de composantes principales pour la deuxième expérimentation de la méthode classique.

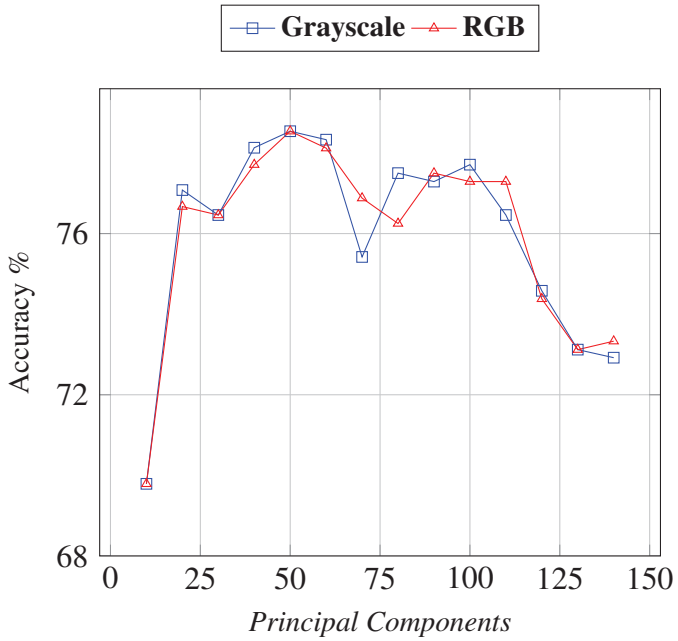


FIGURE 22 – Itération 6 : Taux de reconnaissance en fonction du nombre de composantes principales pour la deuxième expérimentation de la méthode classique.

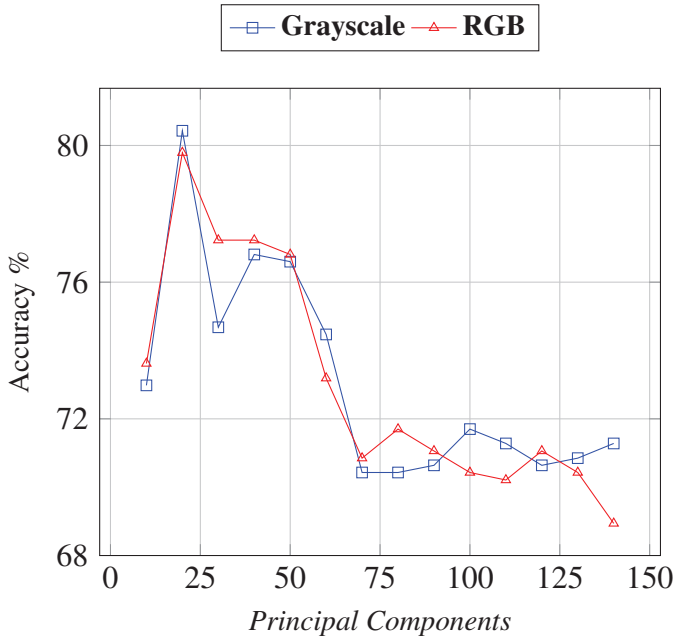


FIGURE 23 – Itération 7 : Taux de reconnaissance en fonction du nombre de composantes principales pour la deuxième expérimentation de la méthode classique.

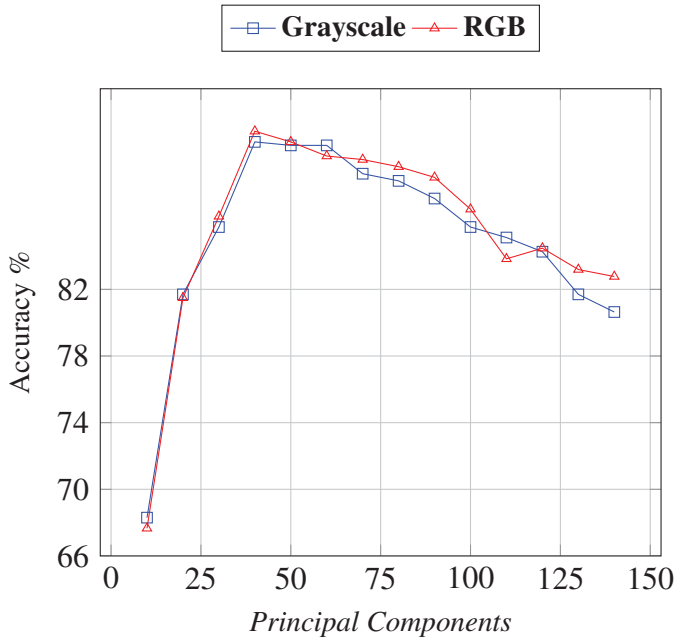


FIGURE 24 – Itération 8 : Taux de reconnaissance en fonction du nombre de composantes principales pour la deuxième expérimentation de la méthode classique.

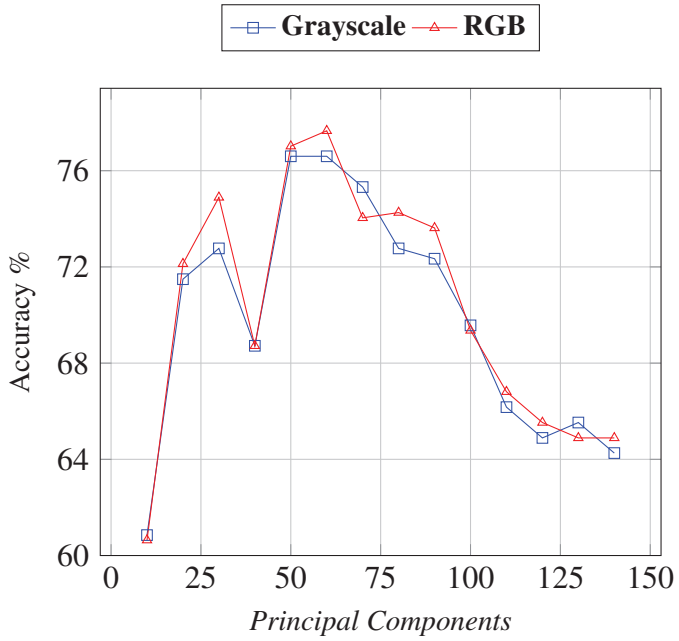


FIGURE 25 – Itération 9 : Taux de reconnaissance en fonction du nombre de composantes principales pour la deuxième expérimentation de la méthode classique.

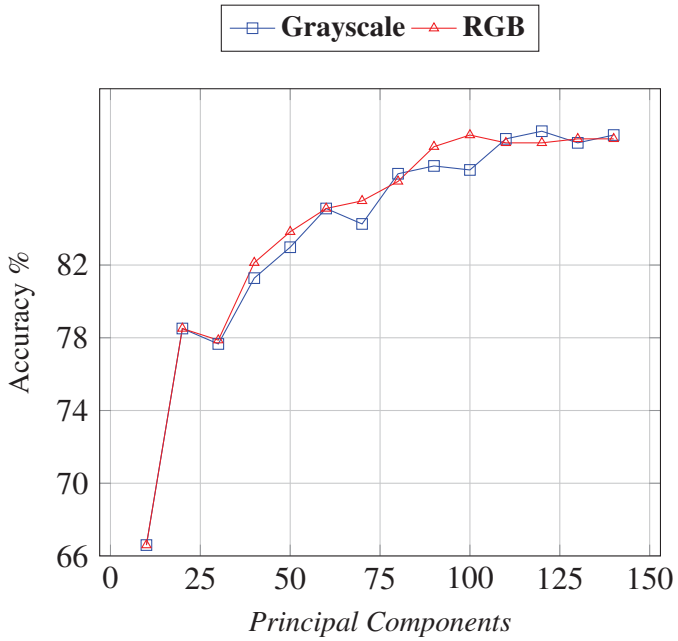


FIGURE 26 – Itération 10 : Taux de reconnaissance en fonction du nombre de composantes principales pour la deuxième expérimentation de la méthode classique.

.2.2 *MÉTHODE PAR APPRENTISSAGE PROFOND*

Voici les graphiques détaillés pour les 10 itérations de la validation croisée de la méthode par apprentissage profond.

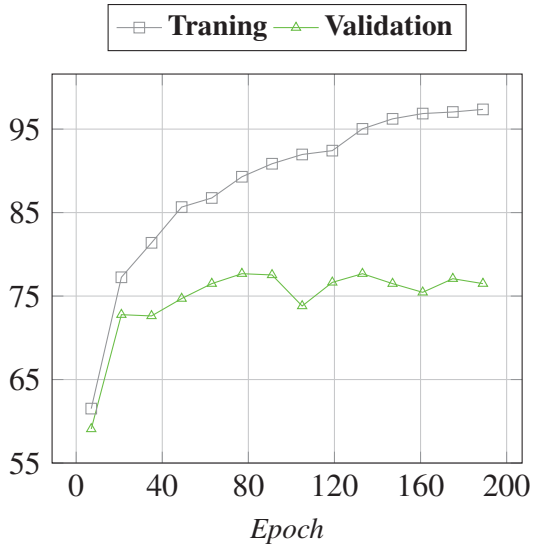


FIGURE 27 – Itération 1 : Taux de reconnaissance en fonction de l'*epoch* pour la deuxième expérimentation de la méthode par apprentissage profond.

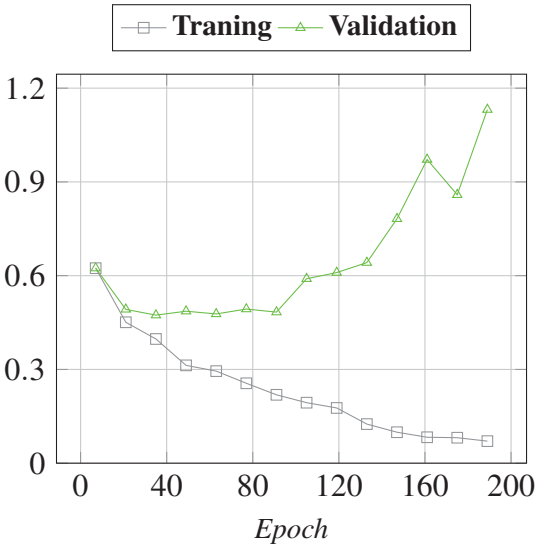


FIGURE 28 – Itération 1 : Perte en fonction de l'*epoch* pour la deuxième expérimentation de la méthode par apprentissage profond.

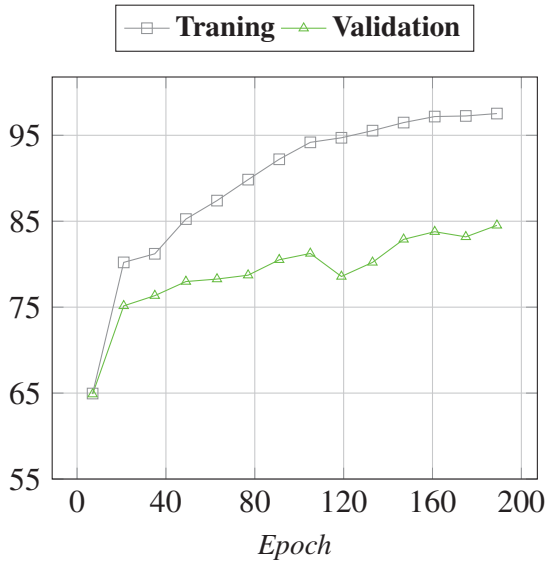


FIGURE 29 – Itération 2 : Taux de reconnaissance en fonction de l'*epoch* pour la deuxième expérimentation de la méthode par apprentissage profond.

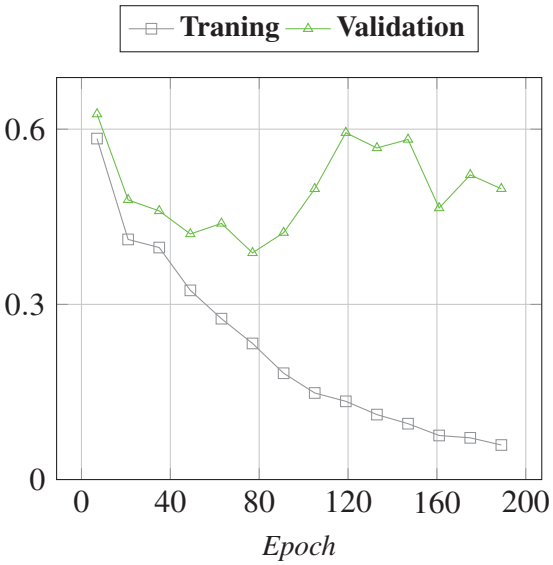


FIGURE 30 – Itération 2 : Perte en fonction de l'*epoch* pour la deuxième expérimentation de la méthode par apprentissage profond.

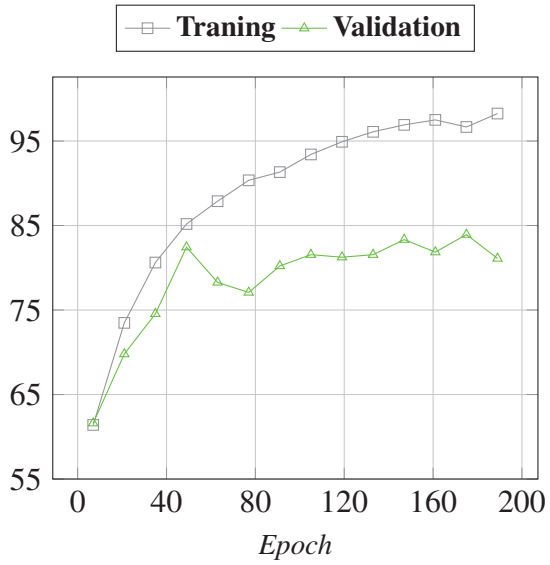


FIGURE 31 – Itération 3 : Taux de reconnaissance en fonction de l'*epoch* pour la deuxième expérimentation de la méthode par apprentissage profond.

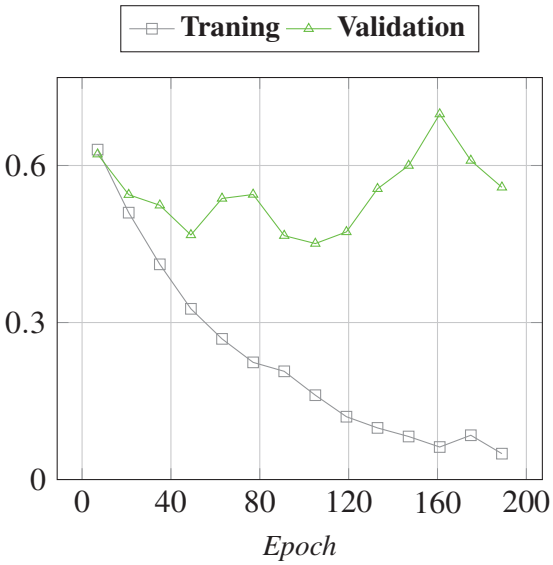


FIGURE 32 – Itération 3 : Perte en fonction de l'*epoch* pour la deuxième expérimentation de la méthode par apprentissage profond.

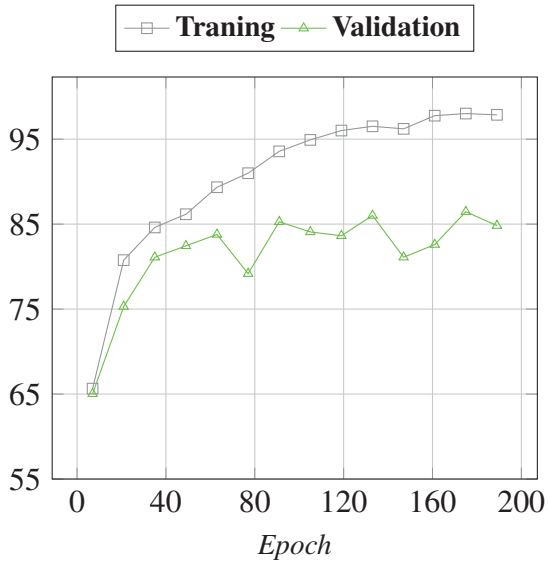


FIGURE 33 – Itération 4 : Taux de reconnaissance en fonction de l'*epoch* pour la deuxième expérimentation de la méthode par apprentissage profond.

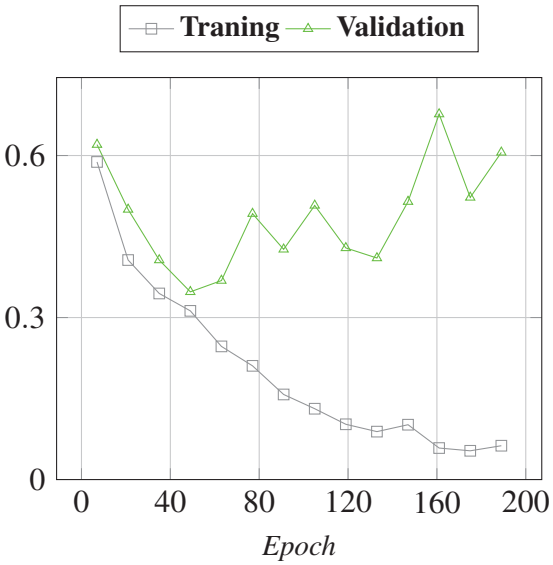


FIGURE 34 – Itération 4 : Perte en fonction de l'*epoch* pour la deuxième expérimentation de la méthode par apprentissage profond.

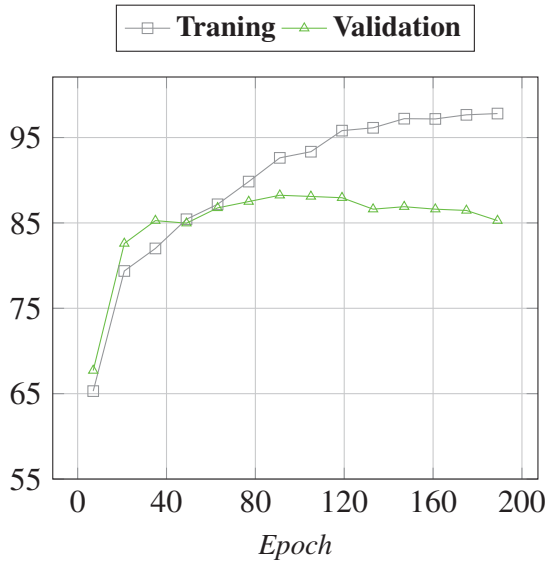


FIGURE 35 – Itération 5 : Taux de reconnaissance en fonction de l'*epoch* pour la deuxième expérimentation de la méthode par apprentissage profond.

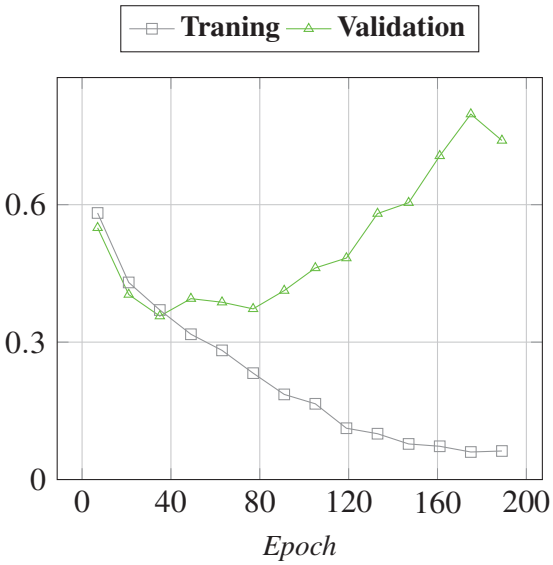


FIGURE 36 – Itération 5 : Perte en fonction de l'*epoch* pour la deuxième expérimentation de la méthode par apprentissage profond.

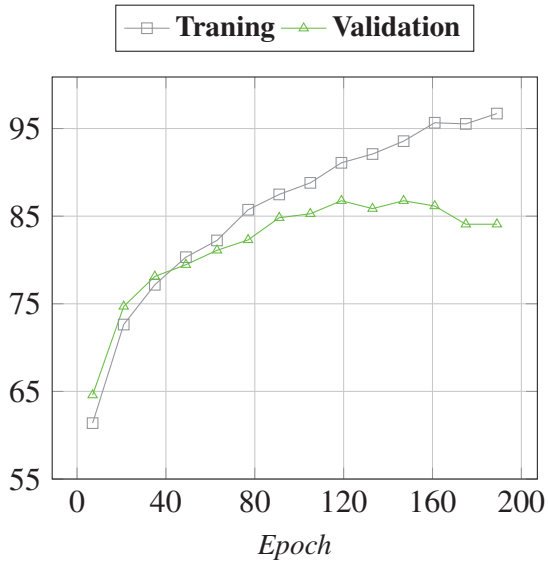


FIGURE 37 – Itération 6 : Taux de reconnaissance en fonction de l'*epoch* pour la deuxième expérimentation de la méthode par apprentissage profond.

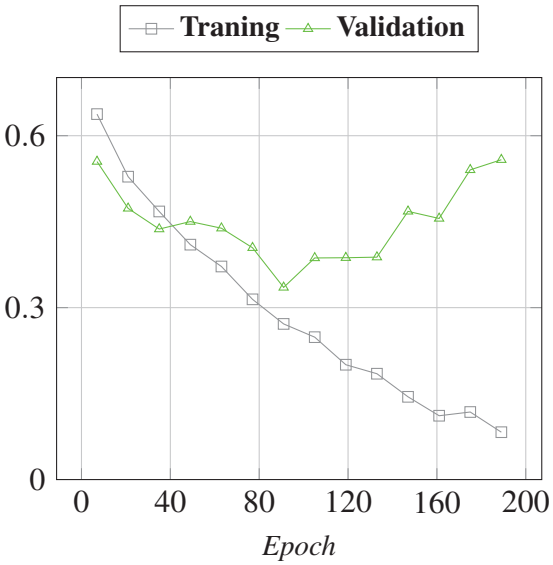


FIGURE 38 – Itération 6 : Perte en fonction de l'*epoch* pour la deuxième expérimentation de la méthode par apprentissage profond.

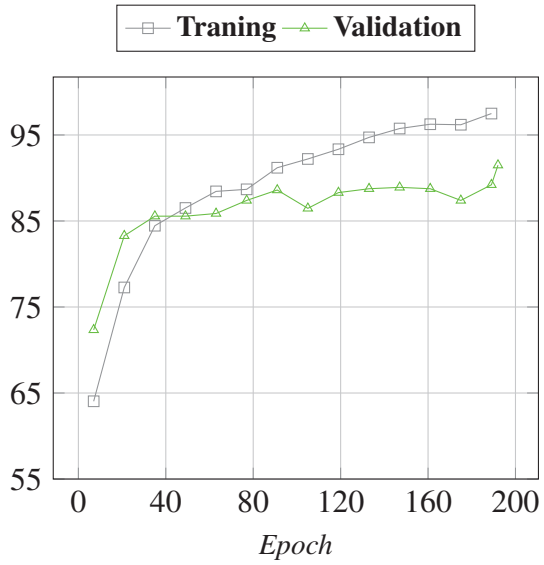


FIGURE 39 – Itération 7 : Taux de reconnaissance en fonction de l'*epoch* pour la deuxième expérimentation de la méthode par apprentissage profond.

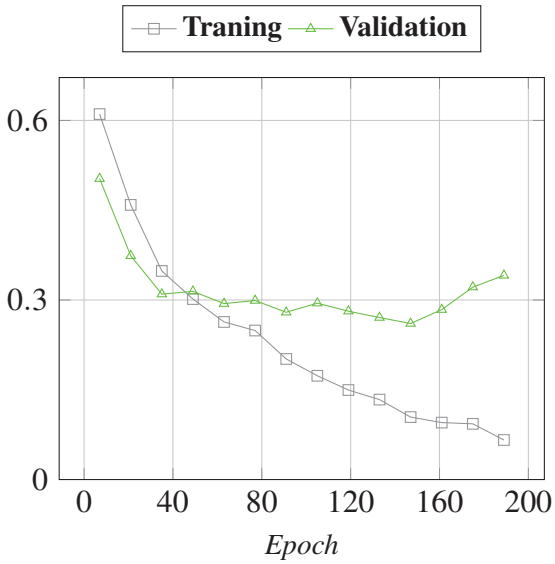


FIGURE 40 – Itération 7 : Perte en fonction de l'*epoch* pour la deuxième expérimentation de la méthode par apprentissage profond.

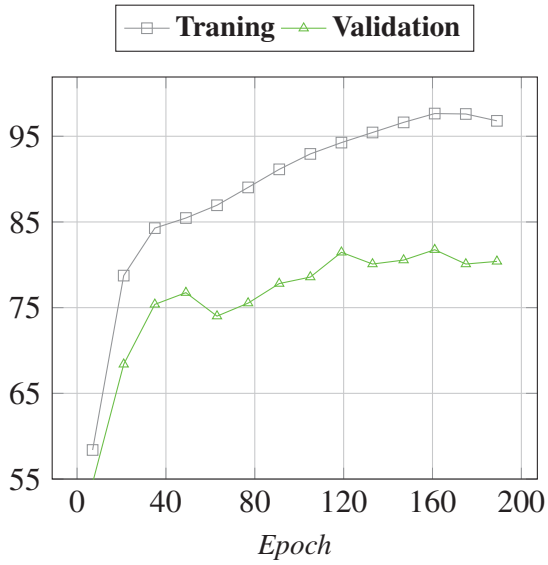


FIGURE 41 – Itération 8 : Taux de reconnaissance en fonction de l'*epoch* pour la deuxième expérimentation de la méthode par apprentissage profond.

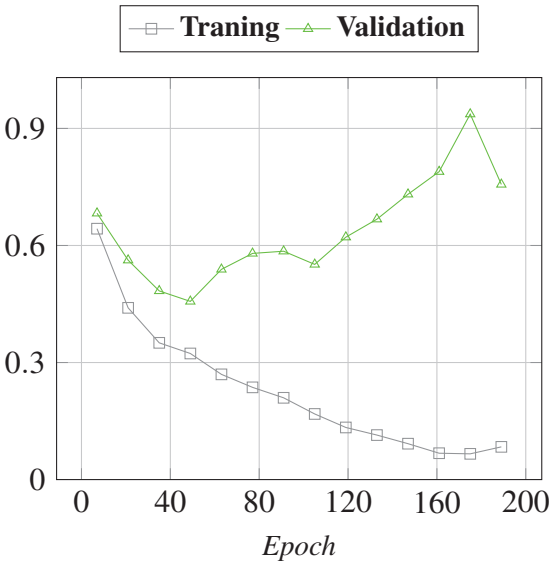


FIGURE 42 – Itération 8 : Perte en fonction de l'*epoch* pour la deuxième expérimentation de la méthode par apprentissage profond.

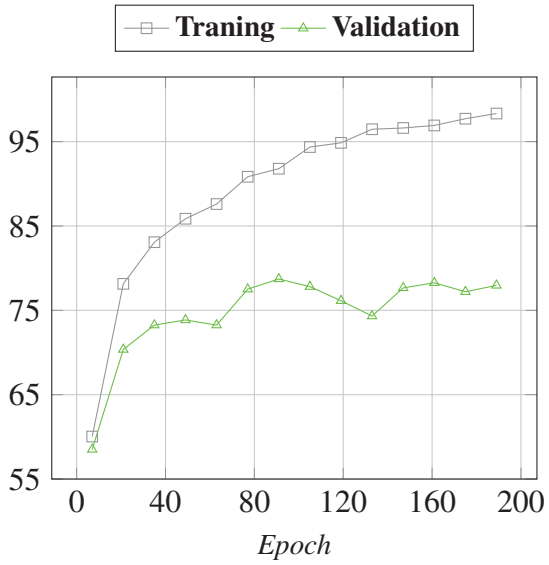


FIGURE 43 – Itération 9 : Taux de reconnaissance en fonction de l'*epoch* pour la deuxième expérimentation de la méthode par apprentissage profond.

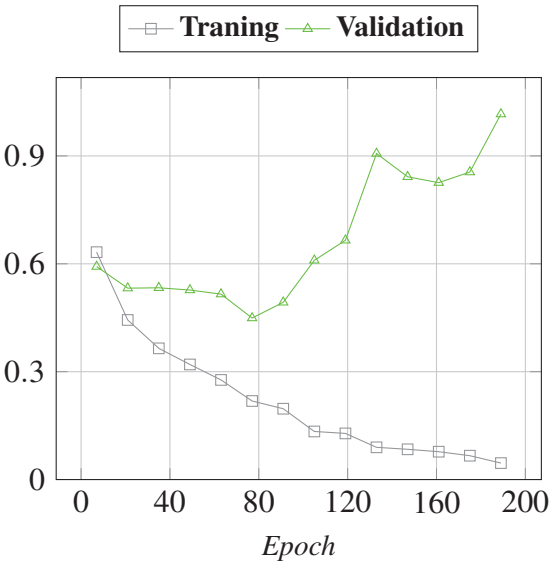


FIGURE 44 – Itération 9 : Perte en fonction de l'*epoch* pour la deuxième expérimentation de la méthode par apprentissage profond.

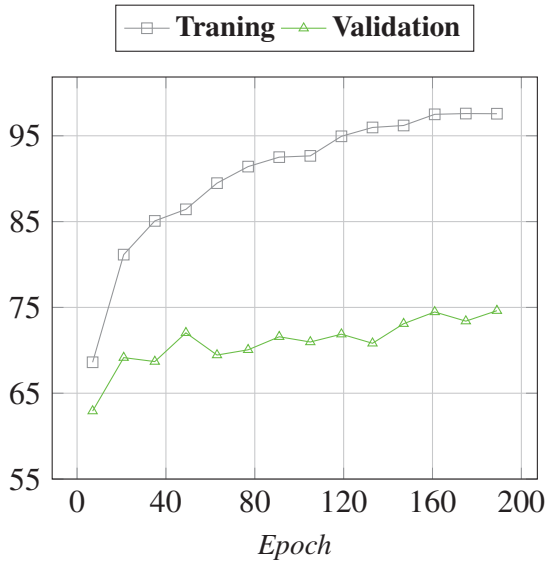


FIGURE 45 – Itération 10 : Taux de reconnaissance en fonction de l'*epoch* pour la deuxième expérimentation de la méthode par apprentissage profond.

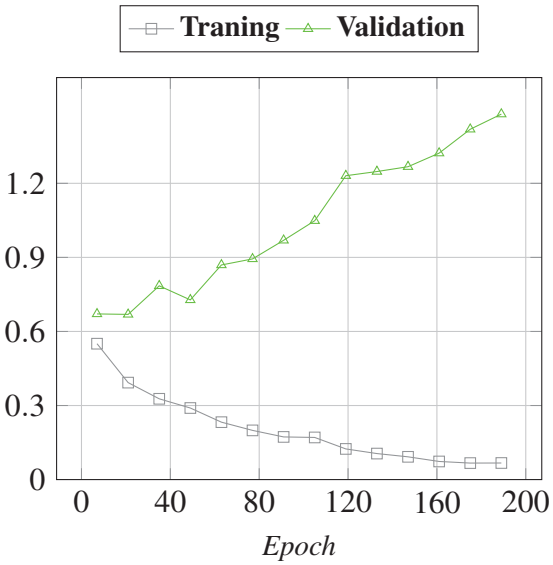


FIGURE 46 – Itération 10 : Perte en fonction de l'*epoch* pour la deuxième expérimentation de la méthode par apprentissage profond.

.3 CODE POUR LA PREMIÈRE MÉTHODE PROPOSÉE

```
# -*- coding: UTF-8 -*-

import os
import numpy as np
import cv2
import pickle
from skimage.feature import hog
from sklearn import preprocessing
from sklearn.svm import SVC
from sklearn.model_selection import StratifiedKFold
from sklearn.decomposition import PCA
from sklearn.metrics import accuracy_score
from sklearn.metrics import confusion_matrix


class QueenBeeDetection:
def __init__(self):
#####
print("-- Start session --\n")
self.hog_parameters = {
"orientations": 16,
"pixels_per_cell": (16, 16),
"cells_per_block": (4, 4),
}
self.text_labels = ["queen", "worker"]

def __import_data(self, src_folder, rgb_mode):
#####
data = []
labels = []

for sub_folder in self.text_labels:
list_img_filename = os.listdir("{}/{}/".format(src_folder, sub_folder))

for img_filename in list_img_filename:
labels.append(sub_folder)

img_in = cv2.imread(
"{}/{}/{}".format(src_folder, sub_folder, img_filename)
```

```
)

if rgb_mode:
img_in = cv2.cvtColor(img_in, cv2.COLOR_BGR2YUV)
img_in[:, :, 0] = cv2.equalizeHist(img_in[:, :, 0])
else:
if len(img_in.shape) == 3:
img_in = cv2.cvtColor(img_in, cv2.COLOR_BGR2GRAY)

img_in = cv2.equalizeHist(img_in)

hog_hist = hog(
img_in,
orientations=self.hog_parameters["orientations"],
pixels_per_cell=self.hog_parameters["pixels_per_cell"],
cells_per_block=self.hog_parameters["cells_per_block"],
visualize=False,
feature_vector=True,
multichannel=rgb_mode,
)

data.append(hog_hist)

self.data = np.array(data)
self.labels = np.array(labels)

def train_model(self, src_folder, rgb_mode=True, nbr_cpnt=30):
#####

# import data and extract hog features
self.__import_data(src_folder, rgb_mode)

# dimensionality reduction
self.data = preprocessing.minmax_scale(self.data, feature_range=(0, 1), axis=1)
self.data = self.data.astype("float32")
pca = PCA(n_components=nbr_cpnt)
self.data = pca.fit_transform(self.data)

# traning model
svm = SVC(
C=1.0,
kernel="rbf",
```

```
random_state=10,
decision_function_shape="ovr",
gamma="auto",
)
svm.fit(self.data, self.labels)

# save models
pickle.dump(pca, open("pca.csv", "wb"))
pickle.dump(svm, open("svm.csv", "wb"))

print("Model training completed.")

def recognition(self, src_image, rgb_mode=True):
#####

# load models
pca = pickle.load(open("pca.csv", "rb"))
svm = pickle.load(open("svm.csv", "rb"))

# read image and extract hog feature vector
img_in = cv2.imread(src_image)

if rgb_mode:
img_in = cv2.cvtColor(img_in, cv2.COLOR_BGR2YUV)
img_in[:, :, 0] = cv2.equalizeHist(img_in[:, :, 0])
else:
if len(img_in.shape) == 3:
img_in = cv2.cvtColor(img_in, cv2.COLOR_BGR2GRAY)

img_in = cv2.equalizeHist(img_in)

hog_hist = hog(
img_in,
orientations=self.hog_parameters["orientations"],
pixels_per_cell=self.hog_parameters["pixels_per_cell"],
cells_per_block=self.hog_parameters["cells_per_block"],
visualize=False,
feature_vector=True,
multichannel=rgb_mode,
)

data = np.array([hog_hist])
```

```
# apply dimensionality reduction
data = preprocessing.minmax_scale(data, feature_range=(0, 1), axis=1)
data = data.astype("float32")
data = pca.transform(data)

# perform recognition
predicted = svm.predict(data)[0]
print(
"From the input image, the predicted class/label -> {}.".format(predicted)
)

def __mean_conf_mat(self, conf_mats):
#####
conf_mat = np.zeros(conf_mats[0].shape)

for element in conf_mats:
for i in range(conf_mats[0].shape[0]):
for j in range(conf_mats[0].shape[0]):
conf_mat[i][j] += element[i][j]

conf_mat = conf_mat.astype("float") / conf_mat.sum(axis=1)[:, np.newaxis]

for idx in range(len(self.text_labels)):
for jdx in range(len(self.text_labels)):
conf_mat[idx][jdx] = "%0.2f" % (conf_mat[idx][jdx] * 100)

return conf_mat.tolist()

def __evaluate(self):
#####
accuracy_vect = []
conf_mat_vect = []

for fold in self.folds:
self.clf.fit(
self.data_r[fold["train_index"]], self.labels[fold["train_index"]]
)

predicted = self.clf.predict(self.data_r[fold["test_index"]])
accuracy_vect.append(
accuracy_score(self.labels[fold["test_index"]], predicted)
```

```
)
conf_mat_vect.append(
confusion_matrix(
self.labels[fold["test_index"]],
predicted,
labels=self.text_labels,
)
)

accuracy = "%0.2f" % ((float(sum(accuracy_vect)) / len(accuracy_vect)) * 100)
conf_mat = self.__mean_conf_mat(np.array(conf_mat_vect))

return accuracy, conf_mat

def model_evaluation(self, src_folder, rgb_mode=True):
#####

# import data and extract hog features
self.__import_data(src_folder, rgb_mode)

# dimensionality reduction
self.data = preprocessing.minmax_scale(self.data, feature_range=(0, 1), axis=1)
self.data = self.data.astype("float32")

# classification
self.clf = SVC(
C=1.0,
kernel="rbf",
random_state=10,
decision_function_shape="ovr",
gamma="auto",
)

skf = StratifiedKFold(n_splits=10, shuffle=True, random_state=10)
k_folds = skf.split(self.data, self.labels)

self.folds = []

for train_index, test_index in k_folds:
self.folds.append(
{"train_index": train_index.tolist(), "test_index": test_index.tolist()}
)
```

```
n_components_max = min(len(self.data), len(self.data[0]))
best_accuracy = 0

for nbr_cpnt in range(10, n_components_max, 5):

pca = PCA(n_components=nbr_cpnt)
self.data_r = pca.fit_transform(self.data)

accuracy, conf_mat = self.__evaluate()

if float(accuracy) > best_accuracy:
best_accuracy = float(accuracy)
best_conf_mat = conf_mat
best_nbr_cpnt = nbr_cpnt

# print accuracy
print(
"{}% \t<- Evaluation using cross-validation (10-folds).".format(
best_accuracy
)
)

# print number of principal components
print(
"{} \t<- Optimized number of principal components.".format(
str(best_nbr_cpnt).zfill(3)
)
)

# print confusion matrix
print("\n\tqueen\tworker")
print("queen\t{}%\t{}%".format(best_conf_mat[0][0], best_conf_mat[0][1]))
print("worker\t{}%\t{}%".format(best_conf_mat[1][0], best_conf_mat[1][1]))

def __del__(self):
#####
print("\n-- End session --")
```

.4 CODE POUR LA DEUXIÈME MÉTHODE PROPOSÉE

```
# In the current directory :
# 1 - One folder with both worker and queen in it.
# 2 - A csv file of images' name with the label.
# 3 - Create a file named : 'saved_models_no_tl/' (see save_dir)

import numpy as np
import pandas as pd
import os
from sklearn.model_selection import KFold, StratifiedKFold
import tensorflow as tf
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Conv2D, Flatten,
Dropout, MaxPooling2D
import matplotlib.pyplot as plt
from tensorflow.keras import datasets, layers, models
from sklearn.metrics import classification_report, confusion_matrix
import csv

# Folder of worker and queen is in current directory.
# Images of queen are named Queen01.jpg and images of worker are named
# Worker01.jpg.
image_dir = "bee/"
zip_name = "bee.zip"
!unzip -qq $zip_name

# A csv file of images' name with the label.
train_data = pd.read_csv('training_label.csv')
Y = train_data[['Label']]

# For the k-fold cross-validation. Same number of queen and worker (stratified).
skf = StratifiedKFold(n_splits = 10, random_state=10, shuffle = True)

# Total number of images.
n=train_data.shape[0]

# Image size.
IMG_HEIGHT = 128
IMG_WIDTH = 128
```

```
# Tensor for the on-time augmentation
idg_train = ImageDataGenerator(rescale=1./255,
rotation_range=30,
brightness_range=[0.8,1.2],
horizontal_flip=True,
vertical_flip = True)
idg_val = ImageDataGenerator(rescale=1./255)

# We will save the model. Here is to name them.
def get_model_name(k):
return 'model_'+str(k)+'.h5'

# The base model.
def create_new_model():
model = Sequential([
tf.keras.Input(shape = (128,128,3)),
layers.Conv2D(16, 5, padding='same', activation='relu'),
layers.MaxPooling2D(),
layers.Conv2D(32, 3, padding='same', activation='relu'),
layers.MaxPooling2D(),
layers.Dropout(0.4),
layers.Flatten(),
layers.Dense(512, activation='relu'),
layers.Dense(256, activation='relu'),
layers.Dense(128, activation='relu'),
layers.Dense(1)
])
return model

# To plot the accuracy and the loss.
def visualize_model(history):
acc = history.history['accuracy']
val_acc = history.history['val_accuracy']

loss = history.history['loss']
val_loss = history.history['val_loss']

epochs_range = range(epochs)

plt.figure(figsize=(8, 8))
plt.subplot(1, 2, 1)
```

```
plt.plot(epochs_range, acc, label='Training Accuracy')
plt.plot(epochs_range, val_acc, label='Validation Accuracy')
plt.legend(loc='lower right')
plt.title('Training and Validation Accuracy')

plt.subplot(1, 2, 2)
plt.plot(epochs_range, loss, label='Training Loss')
plt.plot(epochs_range, val_loss, label='Validation Loss')
plt.legend(loc='upper right')
plt.title('Training and Validation Loss')
plt.show()


# To save the accuracy and loss of the best model (model at best epoch).
VALIDATION_ACCURACY = []
VALIDAITON_LOSS = []

# To save the accuracy and loss over epoch.
save_train_accu = []
save_val_accu = []

save_train_loss = []
save_val_loss = []

# Keep all confusion matrix of the 10 tests.
MATRIXWW = []
MATRIXWQ = []
MATRIXQW = []
MATRIXQQ = []

# Model is saving when reach better accuracies.
save_dir = 'saved_models_no_tl/'

fold_var = 1
epochs = 200
BATCH_SIZE = 32

# Execute the 10 tests  of the 10-folds cross-validation.
for train_index, val_index in skf.split(np.zeros(n),Y):
print("model training ", fold_var)

# Select images for de test.
```

```
training_data = train_data.iloc[train_index]
validation_data = train_data.iloc[val_index]

# Generator of images with augmentations set earlier.
train_data_generator = idg_train.flow_from_dataframe(
training_data, batch_size=BATCH_SIZE, directory = image_dir,
x_col = "Filename", y_col = "Label", target_size=(IMG_HEIGHT, IMG_WIDTH),
class_mode = "binary", shuffle = True)

valid_data_generator  = idg_val.flow_from_dataframe(
validation_data, batch_size=BATCH_SIZE, directory = image_dir,
x_col = "Filename", y_col = "Label",target_size=(IMG_HEIGHT, IMG_WIDTH),
class_mode = "binary", shuffle = True)

# We use the same data as validation. It is only compute the accuracy with
# the best model saved.
test_data_generator = idg_val.flow_from_dataframe(
validation_data, batch_size=BATCH_SIZE, directory = image_dir,
x_col = "Filename", y_col = "Label",target_size=(IMG_HEIGHT, IMG_WIDTH),
class_mode = "binary", shuffle = False)

# Create the model to train.
model = create_new_model()

# Compile the model. We choose accuracy as metric.
model.compile(
optimizer=tf.keras.optimizers.Adam(learning_rate=0.001),
loss=tf.keras.losses.BinaryCrossentropy(from_logits=True),
metrics=['accuracy'])

# Get a summary of the model (the same for all tests).
model.summary()

# Callbacks for saving the model. We save only if we have a better result
# at the current epoch.
checkpoint = tf.keras.callbacks.ModelCheckpoint(
save_dir+get_model_name(fold_var), monitor='val_accuracy', verbose=1,
save_best_only=True, mode='max')
callbacks_list = [checkpoint]

# Fit the model with the images. (train the model)
history = model.fit(train_data_generator,
```

```
epochs=epochs,
callbacks=callbacks_list,
validation_data=valid_data_generator)

# Print the evolution of accuracy and loss for training and validation data
# over epochs. Save those values for further uses.
visualize_model(history)

save_train_accu.append(history.history['accuracy'])
save_val_accu.append(history.history['val_accuracy'])

save_train_loss.append(history.history['loss'])
save_val_loss.append(history.history['val_loss'])



# From the training, load wieghts of the last model saved. It is to try this
# model with the validation data
model.load_weights("saved_models_no_tl/model_"+str(fold_var)+".h5")

# Print the confusion matrix.
print('Les classes (0 pour abeille, 1 pour reine) : ',
test_data_generator.classes)
Y_pred = model.predict_classes(test_data_generator)
nb_test = len(test_data_generator.classes)
y_pred = np.reshape(Y_pred,(nb_test,))
print('La prédiction est : ', y_pred)
print('Confusion matrix : ')
matrix_conf = confusion_matrix(test_data_generator.classes, y_pred)
print(matrix_conf)
MATRIXWW.append(matrix_conf[0][0])
MATRIXWQ.append(matrix_conf[0][1])
MATRIXQW.append(matrix_conf[1][0])
MATRIXQQ.append(matrix_conf[1][1])

# Accuracy and loss of the model with de validation dataset.
results = model.evaluate(test_data_generator)
results = dict(zip(model.metrics_names,results))
VALIDATION_ACCURACY.append(results['accuracy'])
VALIDAITON_LOSS.append(results['loss'])
```

```
# Clear the session for the next test.
tf.keras.backend.clear_session()
fold_var += 1
print("all clean")
print(" ")

# Average accuracy of the 10-folds tests.
print(sum(VALIDATION_ACCURACY)/len(VALIDATION_ACCURACY))

# Average confusion matrix. Use in the article.
QQ = sum(MATRIXQQ)/len(MATRIXQQ)
WW = sum(MATRIXWW)/len(MATRIXWW)
QW = sum(MATRIXQW)/len(MATRIXQW)
WQ = sum(MATRIXWQ)/len(MATRIXWQ)
# Average accuracy from the confusion matrix.
Accu = (QQ+WW)/(QQ+WW+QW+WQ)

print(Accu)
print("queen as queen : ", QQ)
print("queen as worker : ", QW)
print("worker as worker : ", WW)
print("worker as queen : ", WQ)

print(save_train_accu)
print(save_val_accu)

print(save_train_loss)
print(save_val_loss)


# Create a csv file with the average accuracy and loss over epoch of the 10
# tests. Use in the article.
arrays_train = [np.array(x) for x in save_train_accu]
train_average = [np.mean(k) for k in zip(*arrays_train)]
print(train_average)
arrays_val = [np.array(x) for x in save_val_accu]
val_average = [np.mean(k) for k in zip(*arrays_val)]
print(val_average)
with open('average_accu.csv', 'w', newline='') as file1:
writer1 = csv.writer(file1)
writer1.writerow(["train accu over epoch for each iter", save_train_accu])
writer1.writerow(["val accu over epoch for each iter", save_val_accu])
```

```
writer1.writerow(["train loss over epoch for each iter", save_train_loss])
writer1.writerow(["val loss over epoch for each iter", save_val_loss])
writer1.writerow(["training_accuracy", train_average])
writer1.writerow(["validation_accuracy", val_average])
```

BIBLIOGRAPHIE

Abadi, M., P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. 2016. « Tensorflow : A system for large-scale machine learning ». In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, p. 265–283.

Abdullahi, H. S., R. Sheriff, et F. Mahieddine. 2017. « Convolution neural network in precision agriculture for plant image recognition and classification ». In *2017 Seventh International Conference on Innovative Computing Technology (INTECH)*. T. 10.

Abraham, A. 2005. « Artificial neural networks », *Handbook of measuring system design*.

Aizen, M. A. et L. D. Harder. 2009. « The global stock of domesticated honey bees is growing slower than agricultural demand for pollination », *Current biology*, vol. 19, no. 11, p. 915–918.

Apicella, A., F. Donnarumma, F. Isgrò, et R. Prevete. 2021. « A survey on modern trainable activation functions », *Neural Networks*, vol. 138, p. 14–32.

Arulkumaran, K., M. P. Deisenroth, M. Brundage, et A. A. Bharath. 2017. « Deep reinforcement learning : A brief survey », *IEEE Signal Processing Magazine*, vol. 34, no. 6, p. 26–38.

Awesome Duplicate Photo Finer. 2021. Awesome duplicate photo finer. https://www.duplicate-finder.com/photo.html. Accessed : 2021-05-12.

Balasubramanian, M., E. L. Schwartz, J. B. Tenenbaum, V. de Silva, et J. C. Langford. 2002. « The isomap algorithm and topological stability », *Science*, vol. 295, no. 5552, p. 7–7.

Bay, H., T. Tuytelaars, et L. Van Gool. 2006. « Surf : Speeded up robust features ». In *European conference on computer vision*, p. 404–417. Springer, Berlin, Heidelberg.

Bee Appy. 2021. Bee appy. https://thebeeappy.com/. Accessed : 2021-05-15.

Bee Queen Detector. 2021. Bee queen detector. https://play.google.com/store/apps/details?id=org.bqd.queen_detector&hl=fr_CA&gl=US. Accessed : 2021-05-15.

BeeKing. 2021. Beeking. https://beeking.eu/en/. Accessed : 2021-05-15.

BeePlus Beekeeping Manager. 2021. Beeplus. https://hive-tool.com/. Accessed : 2021-05-15.

Bhatia, N. et Vandana. 2010. « Survey of nearest neighbor techniques », *arXiv preprint arXiv :1007.0085*.

Bjerge, K., C. E. Frigaard, P. H. Mikkelsen, T. H. Nielsen, M. Misbih, et P. Kryger. 2019. « A computer vision system to monitor the infestation level of varroa destructor in a honeybee colony », *Computers and Electronics in Agriculture*, vol. 164, p. 104898.

Bodur, Ç., M. Kence, et A. Kence. 2007. « Genetic structure of honeybee, apis mellifera l.(hymenoptera : Apidae) populations of turkey inferred from microsatellite analysis », *Journal of Apicultural Research*, vol. 46, no. 1, p. 50–56.

BoxyBEE. 2021. Boxybee. https://apppage.net/preview/com.boxybee.beenary. Accessed : 2021-05-15.

Bradski, G. 2000. « The opencv library », *Dr Dobb's J. Software Tools*, vol. 25, p. 120–125.

Breiman, L. 2001. « Random forests », *Machine learning*, vol. 45, no. 1, p. 5–32.

Brood Minder. 2021. Brood minder. `https://broodminder.com/`. Accessed : 2021-05-15.

Buja, A., W. Stuetzle, et Y. Shen. 2005. « Loss functions for binary class probability estimation and classification : Structure and applications », *Working draft, November*, vol. 3.

Buschbacher, K., D. Ahrens, M. Espeland, et V. Steinhage. 2020. « Image-based species identification of wild bees using convolutional neural networks », *Ecological Informatics*, vol. 55, p. 101017.

Cejrowski, T., J. Szymański, H. Mora, et D. Gil. 2018. « Detection of the bee queen presence using sound analysis ». In *Asian Conference on Intelligent Information and Database Systems*, p. 297–306. Springer.

Chandra, M. A. et S. Bedi. 2018. « Survey on svm and their application in image classification », *International Journal of Information Technology*, p. 1–11.

Chazette, L., M. Becker, et H. Szczerbicka. 2016. « Basic algorithms for bee hive monitoring and laser-based mite control ». In *2016 IEEE Symposium Series on Computational Intelligence (SSCI)*, p. 1–8. IEEE.

Cohn, R. 1998. « Introduction to neo-riemannian theory : a survey and a historical perspective », *Journal of Music Theory*, vol. 42, no. 2, p. 167–180.

Cortes, C. et V. Vapnik. 1995. « Support-vector networks », *Machine learning*, vol. 20, no. 3, p. 273–297.

CVAT. 2018. Computer vision annotation tool. `https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/Adam`. Accessed : 2021-05-08.

da Silva, F. L., M. L. G. Sella, T. M. Francoy, et A. H. R. Costa. 2015. « Evaluating classification and feature selection techniques for honeybee subspecies identification using wing images », *Computers and Electronics in Agriculture*, vol. 114, p. 68–77.

Dai, J., Y. Li, K. He, et J. Sun. 2016. « R-fcn : Object detection via region-based fully convolutional networks ». In *Advances in neural information processing systems*, p. 379–387.

Dalal, N. et B. Triggs. 2005. « Histograms of oriented gradients for human detection ». In *2005 IEEE computer society conference on computer vision and pattern recognition (CVPR'05)*. T. 1, p. 886–893. Ieee.

Davis, S. et P. Mermelstein. 1980. « Comparison of parametric representations for monosyllabic word recognition in continuously spoken sentences », *IEEE transactions on acoustics, speech, and signal processing*, vol. 28, no. 4, p. 357–366.

Djouadi, A. et E. Bouktache. 1997. « A fast algorithm for the nearest-neighbor classifier », *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 19, no. 3, p. 277–282.

Ferrari, S., M. Silva, M. Guarino, et D. Berckmans. 2008. « Monitoring of swarming sounds in bee hives for early detection of the swarming period », *Computers and electronics in agriculture*, vol. 64, no. 1, p. 72–77.

Fortmann-Roe, S. 2012. « Understanding the bias-variance tradeoff », *URL : http ://scott. fortmann-roe. com/docs/BiasVariance. html (hämtad 2019-03-27)*.

Garnery, L., P. Franck, E. Baudry, D. Vautrin, J.-M. Cornuet, et M. Solignac. 1998. « Genetic diversity of the west european honey bee (apis mellifera mellifera and a. m. iberica) ii. microsatellite loci », *Genetics Selection Evolution*, vol. 30, no. Supplement, p. S49–S74.

Ghahramani, Z. 2003. « Unsupervised learning ». In *Summer School on Machine Learning*, p. 72–112. Springer, Berlin, Heidelberg.

Girshick, R. 2015. « Fast r-cnn ». In *Proceedings of the IEEE international conference on computer vision*, p. 1440–1448.

Glorot, X., A. Bordes, et Y. Bengio. 2011. « Deep sparse rectifier neural networks ». In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*. T. 15, p. 315–323. PMLR Workshop and Conference Proceedings.

Goh, K.-S., E. Chang, et K.-T. Cheng. 2001. « Svm binary classifier ensembles for image classification ». In *Proceedings of the tenth international conference on Information and knowledge management*, p. 395–402.

Goodfellow, I., Y. Bengio, A. Courville, et Y. Bengio. 2016. *Deep learning*. T. 1. MIT press Cambridge.

Google. 2021. Google colaboratory. `https://colab.research.google.com/`. Accessed : 2021-05-08.

Guo, Y., Y. Liu, A. Oerlemans, S. Lao, S. Wu, et M. S. Lew. 2016. « Deep learning for visual understanding : A review », *Neurocomputing*, vol. 187, p. 27–48.

Hive Tracks. 2021. Hive traks know your bees ! `https://www.hivetracks.com/`. Accessed : 2021-05-15.

HiveKeepers. 2021. Hivekeepers. `https://www.hivekeepers.com/aboutus`. Accessed : 2021-05-15.

Hofmann, T., B. Schölkopf, et A. J. Smola. 2008. « Kernel methods in machine learning », *The annals of statistics*, vol. 36, no. 3, p. 1171–1220.

Hornik, K., M. Stinchcombe, et H. White. 1989. « Multilayer feedforward networks are universal approximators », *Neural networks*, vol. 2, no. 5, p. 359–366.

Howard, A. G., M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, et H. Adam. 2017. « Mobilenets : Efficient convolutional neural networks for mobile vision applications », *CoRR*, vol. abs/1704.04861.

Huang, N. E. 2014. *Hilbert-Huang transform and its applications*. T. 16. World Scientific.

Izenman, A. J. 2013. *Linear discriminant analysis*. Coll. « Modern multivariate statistical techniques », p. 237–280. Springer.

Janocha, K. et W. M. Czarnecki. 2017. « On loss functions for deep neural networks in classification », *CoRR*, vol. abs/1702.05659.

Jasim, M. 2017. *Building cross-platform desktop applications with electron*. Packt Publishing Ltd.

Jet Brains. 2021. Intellij idea. `https://www.jetbrains.com/fr-fr/idea/`. Accessed : 2021-05-08.

Kamilaris, A. et F. X. Prenafeta-Boldú. 2018. « A review of the use of convolutional neural networks in agriculture », *The Journal of Agricultural Science*, vol. 156, no. 3, p. 312–322.

Khanum, M., T. Mahboob, W. Imtiaz, H. A. Ghafoor, et R. Sehar. 2015. « A survey on unsupervised machine learning algorithms for automation, classification and maintenance », *International Journal of Computer Applications*, vol. 119, no. 13.

Kirchner, W. 1993. « Acoustical communication in honeybees », *Apidologie*, vol. 24, no. 3, p. 297–307.

Klatt, B. K., A. Holzschuh, C. Westphal, Y. Clough, I. Smit, E. Pawelzik, et T. Tscharntke. 2014. « Bee pollination improves crop quality, shelf life and commercial value », *Proceedings of the Royal Society B : Biological Sciences*, vol. 281, no. 1775, p. 2013–2440.

Kobayashi, T., A. Hidaka, et T. Kurita. 2007. « Selection of histograms of oriented gradients features for pedestrian detection ». In *International conference on neural information processing*, p. 598–607. Springer, Berlin, Heidelberg.

Kong, E. B. et T. G. Dietterich. 1995. *Error-correcting output coding corrects bias and variance*. Coll. « Machine Learning Proceedings 1995 », p. 313–321. Elsevier.

Kotsiantis, S. B., I. Zaharakis, et P. Pintelas. 2007. « Supervised machine learning : A review of classification techniques », *Emerging artificial intelligence applications in computer engineering*, vol. 160, no. 1, p. 3–24.

Krizhevsky, A., I. Sutskever, et G. E. Hinton. 2012. « Imagenet classification with deep convolutional neural networks », *Advances in neural information processing systems*, vol. 25, p. 1097–1105.

Kulyukin, V., S. Mukherjee, et P. Amlathe. 2018. « Toward audio beehive monitoring : Deep learning vs. standard machine learning in classifying beehive audio samples », *Applied Sciences*, vol. 8, no. 9, p. 1573.

LeCun, Y., Y. Bengio, et G. Hinton. 2015. « Deep learning », *nature*, vol. 521, no. 7553, p. 436–444.

LeCun, Y., B. E. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. E. Hubbard, et L. D. Jackel. 1990. « Handwritten digit recognition with a back-propagation network ». In *Advances in neural information processing systems 2*, p. 396–404.

LeCun, Y., L. Bottou, Y. Bengio, et P. Haffner. 1998. « Gradient-based learning applied to document recognition », *Proceedings of the IEEE*, vol. 86, no. 11, p. 2278–2324.

LeCun, Y. A., L. Bottou, G. B. Orr, et K.-R. Müller. 2012. *Efficient backprop*. Coll. « Neural networks : Tricks of the trade, Second Edition », p. 9–48. Springer.

Lenssen, N. et D. Needell. 2014. « An introduction to fourier analysis with applications to music », *Journal of Humanistic Mathematics*, vol. 4, no. 1, p. 72–91.

Lu, D. et Q. Weng. 2007. « A survey of image classification methods and techniques for improving classification performance », *International journal of Remote sensing*, vol. 28, no. 5, p. 823–870.

Marquis, M.-P., Y. Yaddaden, M. Adda, G. Guillaume, et M. Corriveau-Côté. 2021. « Automatic honey bee queen presence detection on beehive frames using machine learning ». Springer.

Martineau, M., D. Conte, R. Raveaux, I. Arnault, D. Munier, et G. Venturini. 2017. « A survey on image-based insect classification », *Pattern Recognition*, vol. 65, p. 273–284.

Meixner, M. D., M. A. Pinto, M. Bouga, P. Kryger, E. Ivanova, et S. Fuchs. 2013. « Standard methods for characterising subspecies and ecotypes of apis mellifera », *Journal of Apicultural Research*, vol. 52, no. 4, p. 1–28.

Michelsen, A., W. H. Kirchner, et M. Lindauer. 1986. « Sound and vibrational signals in the dance language of the honeybee, apis mellifera », *Behavioral ecology and sociobiology*, vol. 18, no. 3, p. 207–212.

Nawrocka, A., İ. Kandemir, S. Fuchs, et A. Tofilski. 2018. « Computer software for identification of honey bee subspecies and evolutionary lineages », *Apidologie*, vol. 49, no. 2, p. 172–184.

Nolasco, I., A. Terenzi, S. Cecchi, S. Orcioni, H. L. Bear, et E. Benetos. 2019. « Audio-based identification of beehive states ». In *ICASSP 2019-2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, p. 8256–8260. IEEE.

Ojala, T., M. Pietikäinen, et D. Harwood. 1996. « A comparative study of texture measures with classification based on featured distributions », *Pattern recognition*, vol. 29, no. 1, p. 51–59.

Ojala, T., M. Pietikainen, et T. Maenpaa. 2002. « Multiresolution gray-scale and rotation invariant texture classification with local binary patterns », *IEEE Transactions on pattern analysis and machine intelligence*, vol. 24, no. 7, p. 971–987.

Oleksa, A. et A. Tofilski. 2015. « Wing geometric morphometrics and microsatellite analysis provide similar discrimination of honey bee subspecies », *Apidologie*, vol. 46, no. 1, p. 49–60.

Orr, M. J. et al. 1996. Introduction to radial basis function networks.

OSBeehives. 2021. Osbeehives app. `https://www.osbeehives.com/pages/osbeehives-app`. Accessed : 2021-05-15.

O'Shaughnessy, D. 1988. « Linear predictive coding », *IEEE potentials*, vol. 7, no. 1, p. 29–32.

paint.net. 2021. paint.net. `https://www.getpaint.net/index.html`. Accessed : 2021-05-12.

Potts, S. G., J. C. Biesmeijer, C. Kremen, P. Neumann, O. Schweiger, et W. E. Kunin. 2010. « Global pollinator declines : trends, impacts and drivers », *Trends in ecology & evolution*, vol. 25, no. 6, p. 345–353.

python. 2019. python. `https://www.python.org/`. Accessed : 2021-05-08.

Rawat, W. et Z. Wang. 2017. « Deep convolutional neural networks for image classification : A comprehensive review », *Neural computation*, vol. 29, no. 9, p. 2352–2449.

Ren, S., K. He, R. Girshick, et J. Sun. 2015. « Faster r-cnn : Towards real-time object detection with region proposal networks », *Advances in neural information processing systems*, vol. 28, p. 91–99.

Rodriguez, I. F., R. Megret, E. Acuna, J. L. Agosto-Rivera, et T. Giray. 2018. « Recognition of pollen-bearing bees from video using convolutional neural network ». In *2018 IEEE Winter Conference on Applications of Computer Vision (WACV)*, p. 314–322. IEEE.

Ruder, S. 2016. « An overview of gradient descent optimization algorithms », *CoRR*, vol. abs/1609.04747.

Ruttner, F. 2013. *Biogeography and taxonomy of honeybees*. Springer Science & Business Media.

Schurischuster, S., B. Remeseiro, P. Radeva, et M. Kampel. 2018. « A preliminary study of image analysis for parasite detection on honey bees ». In *International Conference Image Analysis and Recognition*, p. 465–473. Springer International Publishing.

Shen, Y., H. Zhou, J. Li, F. Jian, et D. S. Jayas. 2018. « Detection of stored-grain insects using deep learning », *Computers and Electronics in Agriculture*, vol. 145, p. 319–325.

Shorten, C. et T. M. Khoshgoftaar. 2019. « A survey on image data augmentation for deep learning », *Journal of Big Data*, vol. 6, no. 1, p. 1–48.

Simonyan, K. et A. Zisserman. 2014. « Very deep convolutional networks for large-scale image recognition », *arXiv preprint arXiv :1409.1556*.

Srivastava, N., G. Hinton, A. Krizhevsky, I. Sutskever, et R. Salakhutdinov. 2014. « Dropout : a simple way to prevent neural networks from overfitting », *The journal of machine learning research*, vol. 15, no. 1, p. 1929–1958.

Steinhage, V., T. Arbuckle, S. Schröder, A. Cremers, et D. Wittmann. 2001. « Abis : automated identification of bee species ». In *BIOLOG Workshop, German Programme on Biodiversity and Global Change Status Report, Bonn*, p. 194–195.

Sutton, R. S. et A. G. Barto. 2018. *Reinforcement learning : An introduction*. MIT press.

Syromyatnikov, M. Y., A. V. Borodachev, A. V. Kokina, et V. N. Popov. 2018. « A molecular method for the identification of honey bee subspecies used by beekeepers in russia », *Insects*, vol. 9, no. 1, p. 10.

Szegedy, C., W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, et A. Rabinovich. 2015. « Going deeper with convolutions ». In *Proceedings of the IEEE conference on computer vision and pattern recognition*, p. 1–9.

Tensorflow. 2021a. Keras losses binary crossentropy. `https://www.tensorflow.org/api_docs/python/tf/keras/losses/BinaryCrossentropy`. Accessed : 2021-05-08.

———. 2021b. Keras optimizers adam. `https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/Adam`. Accessed : 2021-05-08.

Thenmozhi, K. et U. S. Reddy. 2019. « Crop pest classification based on deep convolutional neural network and transfer learning », *Computers and Electronics in Agriculture*, vol. 164, p. 104906.

Utzeri, V. J., A. Ribani, et L. Fontanesi. 2018. « Authentication of honey based on a dna method to differentiate apis mellifera subspecies : Application to sicilian honey bee (a.

m. siciliana) and iberian honey bee (a. m. iberiensis) honeys », *Food Control*, vol. 91, p. 294–301.

Van der Walt, S., J. L. Schönberger, J. Nunez-Iglesias, F. Boulogne, J. D. Warner, N. Yager, E. Gouillart, et T. Yu. 2014. « scikit-image : image processing in python », *PeerJ*, vol. 2, p. e453.

Wall, M. E., A. Rechtsteiner, et L. M. Rocha. 2003. *Singular value decomposition and principal component analysis*. Coll. « A practical approach to microarray data analysis », p. 91–109. Springer.

Witten, I. H., E. Frank, et M. A. Hall. 2017. *Data Mining Practical Machine Learning Tools and Techniques Third Edition*. Elsevier.

Wold, S., K. Esbensen, et P. Geladi. 1987. « Principal component analysis », *Chemometrics and intelligent laboratory systems*, vol. 2, no. 1-3, p. 37–52.

Wu, H., Q. Liu, et X. Liu. 2019. « A review on deep learning approaches to image classification and object segmentation », *Comput. Mater. Continua*, vol. 60, no. 2, p. 575–597.

Xia, D., P. Chen, B. Wang, J. Zhang, et C. Xie. 2018. « Insect detection and classification based on an improved convolutional neural network », *Sensors*, vol. 18, no. 12, p. 4169.

Yaddaden, Y., M. Adda, et A. Bouzouane. 2021. *A Study of Dimensionality Reduction for Facial Expression Recognition*. Coll. « Advances in Computing Systems and Applications », p. 14–24. Springer International Publishing.

Yaddaden, Y., M. Adda, A. Bouzouane, S. Gaboury, et B. Bouchard. 2018. « User action and facial expression recognition for error detection system in an ambient assisted environment », *Expert Systems with Applications*, vol. 112, p. 173–189.