

# Word Blood Cell Classification

Corral, Damien  
[damien.corral@gmail.com](mailto:damien.corral@gmail.com)

Trushko Perney, Anastasiya  
[anastasia.trushko@gmail.com](mailto:anastasia.trushko@gmail.com)

Porcu, Jordan  
[jordan.porcu@gmail.com](mailto:jordan.porcu@gmail.com)

Lavergne, Jérémy  
[jeremy.lav2009@gmail.com](mailto:jeremy.lav2009@gmail.com)

November 2022

# Table des matières

Bibliographie . . . . .	2
<b>1 Introduction</b>	<b>3</b>
1.1 Objectif . . . . .	3
1.2 Contexte . . . . .	3
1.3 Problématique . . . . .	3
1.4 Le plan de rapport . . . . .	4
<b>2 L'analyse exploratoire des données</b>	<b>5</b>
2.1 Origine de la donnée . . . . .	5
2.1.1 Description du dataset Barcelone . . . . .	5
2.1.2 Aperçu technique . . . . .	5
2.2 Chargement des données . . . . .	6
2.3 Data visualization . . . . .	8
2.3.1 Data visualization avec matplotlib . . . . .	8
2.3.2 Balance des couleurs . . . . .	9
2.3.3 Représentation du nombre d'échantillons par classe et par groupe . . . . .	10
2.3.4 Visualisation de la répartition des images par label . . . . .	10
2.3.5 Analyse des images selon leur format . . . . .	11
2.3.6 Première conclusion . . . . .	12
<b>3 Preprocessing</b>	<b>13</b>
3.1 Traitement des images . . . . .	13
3.1.1 Dimension des images . . . . .	13
3.1.2 Balance des couleurs . . . . .	13
3.2 Segmentation par computer vision . . . . .	15
3.3 Segmentation par deep-learning . . . . .	19
<b>4 Méthodes : Pipeline de modélisation d'un réseau de convolution</b>	<b>23</b>
4.1 Étapes du pipeline de modélisation . . . . .	23
4.2 Modélisation d'un réseau de convolution . . . . .	23
4.2.1 Préparation de l'environnement . . . . .	23
4.2.2 Jeux de données . . . . .	23
4.2.3 Sélection de 4 modèles pré entraînés adaptés à la classification simple d'image	25
4.3 Création et utilisation des modèles . . . . .	25
4.3.1 Implémentation des modèles . . . . .	25
4.3.2 Implémentation du modèle VGG16 . . . . .	27
4.3.3 Implémentation du modèle MobileNetV2 . . . . .	28
4.3.4 Implémentation du modèle Xception . . . . .	30
4.3.5 Implémentation du modèle ResNet50V2 . . . . .	31
<b>5 Résultats et interprétations</b>	<b>34</b>
5.1 Choix des métriques et des visuels adapté à notre problématique . . . . .	34
5.2 Visualisation des différents résultats et prédictions . . . . .	34
5.2.1 Le rapport de classification . . . . .	34
5.2.2 Synthèse des résultats sur les 4 modèles . . . . .	35
5.2.3 Courbe de la fonction de perte et de l'accuracy . . . . .	35
5.2.4 Heatmap des matrices de confusion . . . . .	38
5.2.5 Prédictions à partir des modèles . . . . .	38

5.3	Grad-CAM . . . . .	41
5.3.1	Construction des modèles . . . . .	41
5.3.2	Génération de la heatmap . . . . .	42
<b>6</b>	<b>Conclusion</b>	<b>45</b>
	<b>Bibliographie</b>	<b>47</b>

# Chapitre 1

## Introduction

### 1.1 Objectif

L'objectif de ce projet est de classifier les cellules sanguines en fonction de leurs caractéristiques morphologiques en utilisant des techniques d'apprentissage profond (ou Deep Learning). Pour y parvenir, nous utiliserons un des réseaux neuronaux de convolution (CNN) avec une méthode d'apprentissage par transfert (Transfer Learning) ainsi qu'une optimisation des poids des modèles grâce à la méthode du Fine-tuning.

### 1.2 Contexte

Les diagnostics de la majorité des maladies hématologiques commencent par une analyse morphologique des cellules sanguines périphériques. Ces dernières circulent dans les vaisseaux sanguins et contiennent trois types de cellules principales suspendues dans du plasma :

- l'érythrocyte (globule rouge)
- le leucocyte (globule blanc)
- le thrombocyte (plaquette)

Les leucocytes sont les acteurs majeurs dans la défense de l'organisme contre les infections. Ce sont des cellules nucléées qui sont elles-mêmes divisées en trois classes :

- les granulocytes (subdivisés en neutrophiles segmentés et en bande, en éosinophiles et en basophiles)
- les lymphocytes
- les monocytes

Lorsqu'un patient est en bonne santé, la proportion des différents types de globules blancs dans le plasma est d'environ 54-62% pour les granulocytes, 25-33% pour les lymphocytes et 3-10% pour les monocytes. Cependant, en cas de maladie, par exemple une infection ou une anémie régénérative, cette proportion est modifiée en même temps que le nombre total de globules blancs, et on peut trouver des granulocytes immatures (IG) (promyélocytes, myélocytes et métamyélocytes) ou des précurseurs érythroïdes, comme les érythroblastes.

### 1.3 Problématique

L'identification morphologique des cellules sanguines nécessitent de l'expérience et des compétences. Il existe de nombreux systèmes automatiques de pré-classification des leucocytes (par exemple The easy cell assistant, Vision Hema, Cellavision, Cellavision DM9600). Ces systèmes utilisent des microscopes motorisés et sont souvent très coûteux. De plus, la classification se fait en plusieurs étapes, incluant la segmentation, l'extraction des caractéristiques, la sélection et la reconnaissance de patrons. L'étape qui affecte le plus la précision de la classification de la cellule est la segmentation.

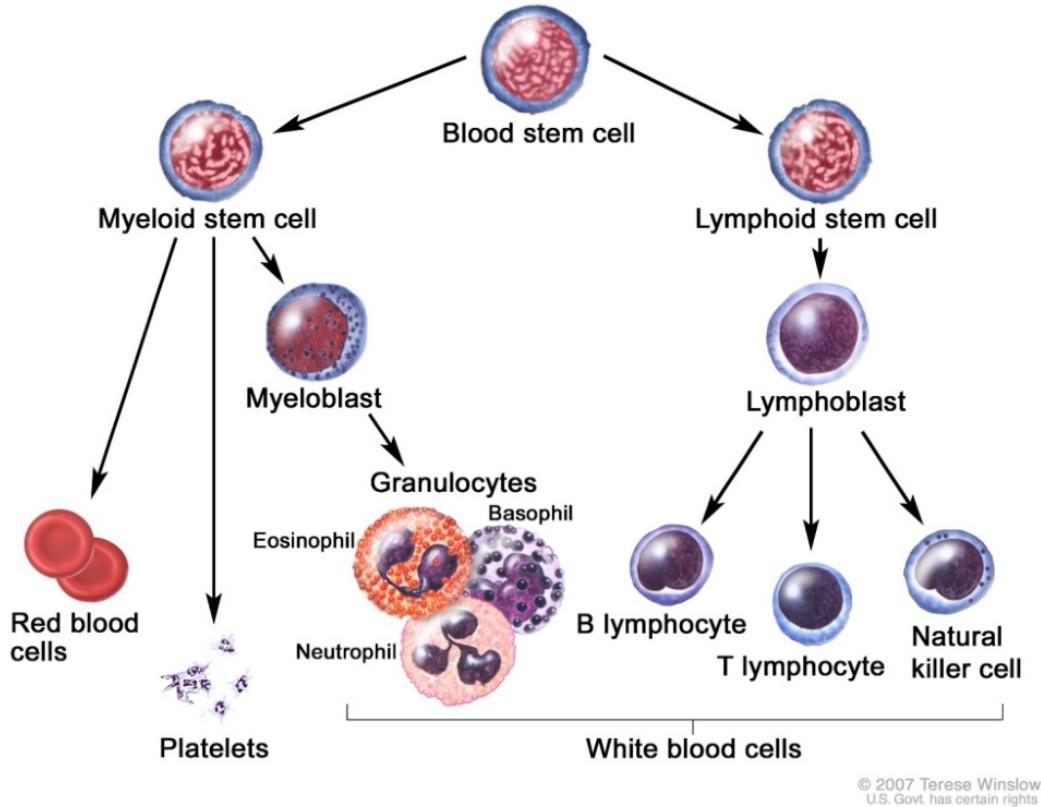


FIGURE 1.1 – Les 8 types de cellules

Il pourrait donc être très avantageux d'avoir un système moins coûteux qui ne dépend pas de la segmentation.

Les cliniciens s'appuient sur des analyses assistées par ordinateur pour les frottis sanguins et pour détecter des cellules anormales.

Jusqu'à présent, les analyses de reconnaissance des globules blancs se basaient sur des modèles d'apprentissage automatique peu profonds pour les segmenter, pour séparer des composants, pour extraire des caractéristiques puis pour les classifier.

Cette technique n'est pas des plus fiables puisqu'elle ne se généralise pas bien, notamment à cause de la variabilité des systèmes de coloration et d'acquisition de Romanowsky ainsi que des exigences importantes de sa conception en termes de pré-traitement et d'extraction de caractéristiques.

## 1.4 Le plan de rapport

C'est dans ce contexte que nous avons travaillé sur notre projet d'apprentissage profond, dans le cadre de la formation Data Scientist de DataScientest, baptisé YAWBCC (Yet Another White Blood Cells Classifier).

Dans ce rapport, nous vous présenterons :

- brièvement le dataset choisi pour le projet, ses spécificités et une analyse exploratoire
- les méthodes employées dans le cadre du preprocessing de nos données
- 4 modèles d'apprentissage implémentés par réseau neuronal profond et leurs performances respectives
- l'implémentation d'un pipeline
- une synthèse des interprétations et des pistes/points d'améliorations à apporter au projet

## Chapitre 2

# L'analyse exploratoire des données

### 2.1 Origine de la donnée

#### 2.1.1 Description du dataset Barcelone

Nous avons mené notre projet en partant d'un jeu de données contenant un total de 17 092 images de cellules normales individuelles, qui ont été acquises à l'aide de l'analyseur Cellavision DM96 dans le laboratoire central de la clinique hospitalière de Barcelone. L'ensemble de données est organisé en huit groupes suivants : neutrophiles, éosinophiles, basophiles, lymphocytes, monocytes, granulocytes immatures (promyélocytes, myélocytes et métamyélocytes), érythroblastes et plaquettes ou thrombocytes.

#### 2.1.2 Aperçu technique

Les images ont été annotées par des pathologistes cliniciens experts. Elles ont été capturées chez des individus sans infection, sans maladie hématologique ou oncologique et sans aucun traitement pharmacologique au moment du prélèvement sanguin.

Cet ensemble de données étiquetées de haute qualité peut être utilisé pour former et tester des modèles d'apprentissage automatique et d'apprentissage en profondeur afin de reconnaître différents types de cellules sanguines périphériques normales.

À notre connaissance, il s'agit du premier ensemble accessible au public avec un grand nombre de cellules sanguines périphériques normales, de sorte qu'il devrait s'agir d'un ensemble de données canonique pour l'analyse comparative des modèles (Acevedo A et al. 2019). Les exemples d'images de jeux de données avec les principales caractéristiques de chaque type de cellule sont représentés dans le Fig.2.1

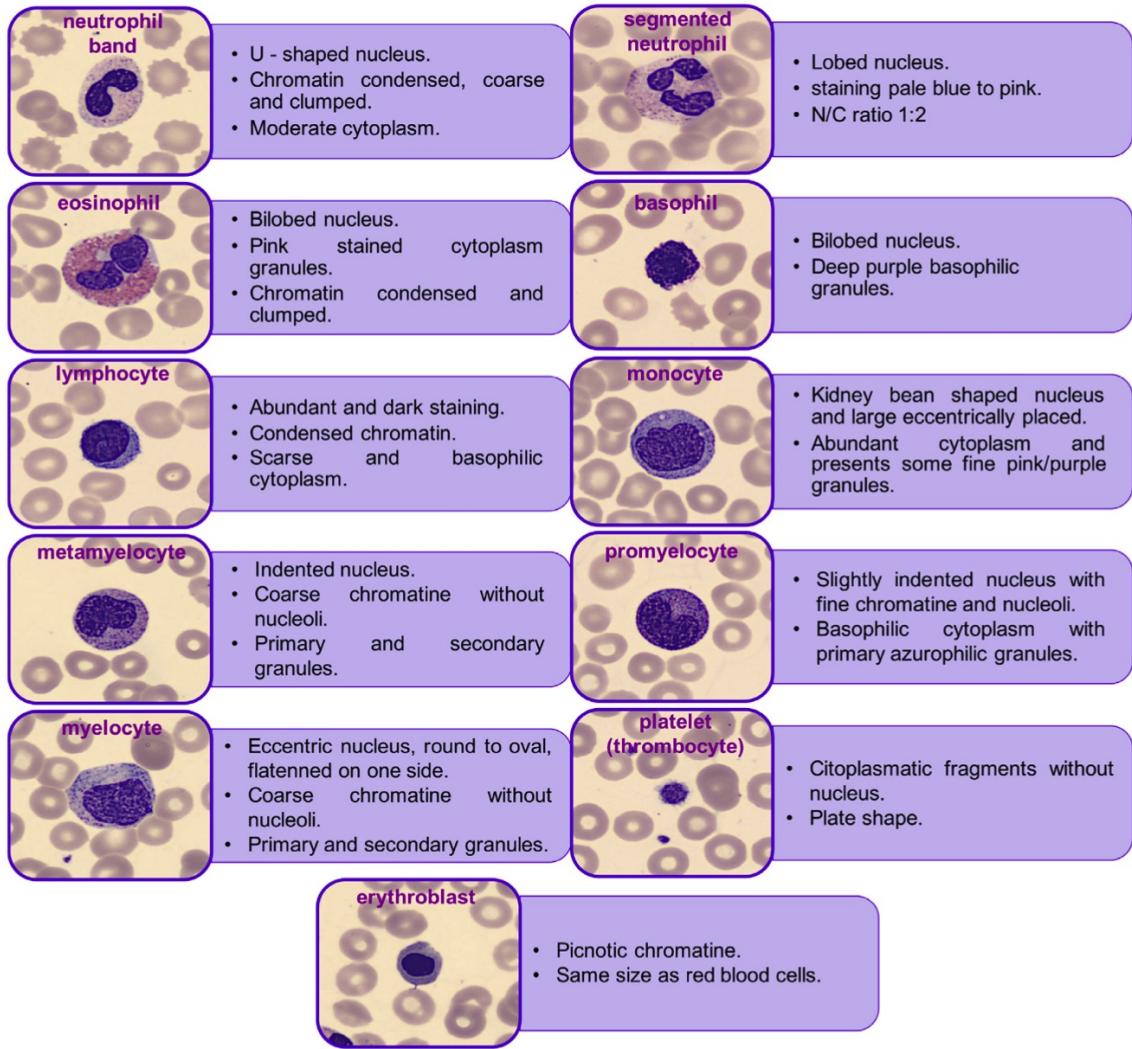


FIGURE 2.1 – Images of different types of leukocytes : neutrophil, eosinophil, basophil, lymphocyte, monocyte and immature granulocytes (promyelocytes, myelocytes and metamyelocytes), erythroblasts and platelets (thrombocytes) (Adopted from (Acevedo A et al. 2019))

## 2.2 Chargement des données

Notre 1er notebook nous permet de télécharger et extraire le jeu de données de Barcelone pour une première analyse exploratoire des données, via une fonction créée `load_barcelona_wbc()`. La fonction `load_barcelona_wbc()` va télécharger et extraire l'ensemble des images contenues dans l'archive.

```

1 def load_barcelona_wbc() -> pd.DataFrame:
2     """Load WBC dataset from Barcelona.
3     Returns:
4         A dataframe with some metadata.
5     """
6
7     DATA_DIR = DATA_ROOT_DIR / 'barcelona'
8
9     # Download dataset if needed
10    fetch_barcelona_wbc()
11
12    data = []
13    # Extract simple informations from dataset
14    for file in DATA_DIR.glob('**/*.jpg'):
15        img = Image.open(file)

```

```

16     d = {'image': file.name,
17           'group': file.parent.name.upper(),
18           'label': file.stem.split('_')[0].upper(),
19           'width': img.size[0],
20           'height': img.size[1],
21           'path': str(file)}
22     data.append(d)
23
24 # Create dataframe then post-process some columns
25 df = pd.DataFrame(data)
26 return df

```

Les fichiers sont extraits dans le sous-répertoire `data/barcelona` du répertoire de votre notebook. Nous avons par la suite créé le dataframe `df`, provenant de la fonction `load_barcelona_wbc()`. Voici un aperçu de notre dataframe :

	<b>image</b>	<b>group</b>	<b>label</b>	<b>width</b>	<b>height</b>	<b>path</b>
0	BNE_489159.jpg	NEUTROPHIL	BNE	360	363	/root/yawbcc_data/barcelona/neutrophil/BNE_489...
1	SNE_323373.jpg	NEUTROPHIL	SNE	360	363	/root/yawbcc_data/barcelona/neutrophil/SNE_323...
2	SNE_668285.jpg	NEUTROPHIL	SNE	360	363	/root/yawbcc_data/barcelona/neutrophil/SNE_668...
3	BNE_102779.jpg	NEUTROPHIL	BNE	360	363	/root/yawbcc_data/barcelona/neutrophil/BNE_102...
4	BNE_835325.jpg	NEUTROPHIL	BNE	360	363	/root/yawbcc_data/barcelona/neutrophil/BNE_835...

FIGURE 2.2 – Dataframe df.

Le jeu de donnée ne charge pas les images directement. La variable « path » permet d'accéder à chaque image.

## 2.3 Data visualization

### 2.3.1 Data visualization avec matplotlib

Dans le jeu de données de 17092 images, il y a 8 groupes : basophile, neutrophile, ig, monocyte, éosinophile, érythroblaste, lymphocyte, plaquette (Fig.2.3).

La classe « ig » comporte 4 types de labels : « MY », « PMY », « MMY », « IG ». (Fig.2.4) La classe « neutrophile » comporte 3 types de labels : « BNE », « SNE » et « neutrophile » (Fig.2.5).

Chaque autre classe n'a qu'un seul type de label : « BA » pour basophile, « EO » pour éosinophile, « ERB » pour érythroblast, « LY » pour lymphocyte, « PLATELET » pour platelet.

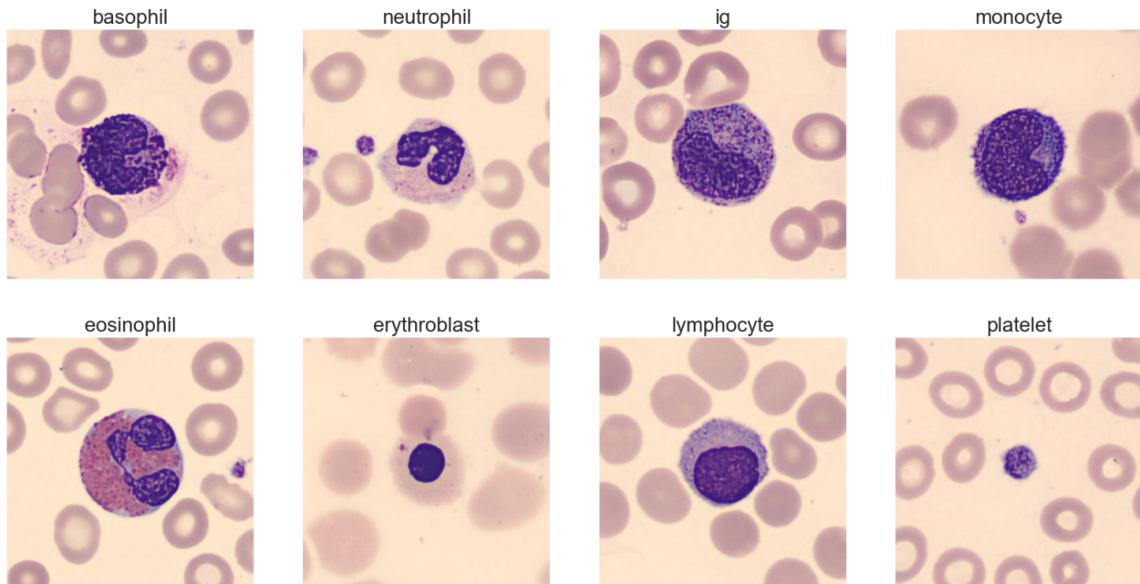


FIGURE 2.3 – Les images de cellules correspondant à chacun des 8 groupes

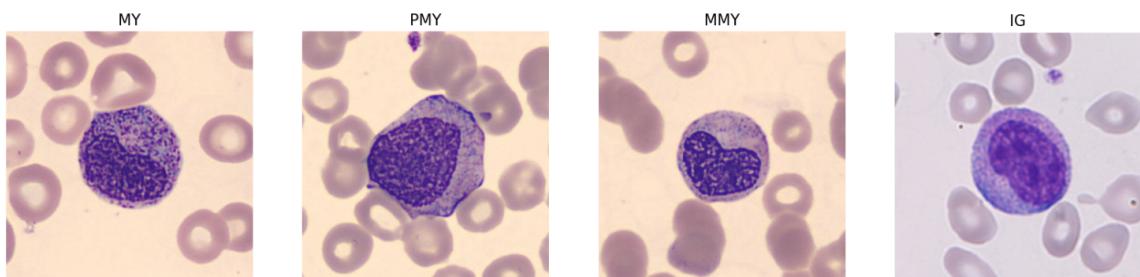


FIGURE 2.4 – Quatre classes dans le groupe IG (granulocytes immatures).

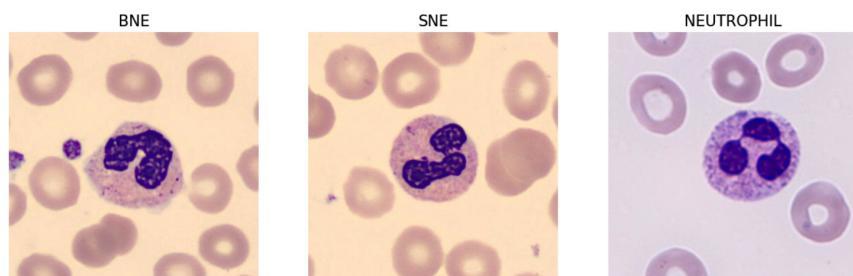


FIGURE 2.5 – Trois classes dans le groupe neutrophil .

### 2.3.2 Balance des couleurs

Le jeu de données se compose de images RGB (images couleur). Les images couleur sont divisées en luminance et chrominance. La luminance est la partie en niveaux de gris et est généralement traitée dans de nombreuses applications. Les histogrammes sont utilisés dans les images en niveaux de gris. Un histogramme compte le nombre d'occurrences des valeurs d'intensité des pixels, et c'est un outil utile pour comprendre et manipuler les images. Nous avons utilisé `cv2.calcHist()` pour générer l'histogramme afin d'inspecter la distribution d'intensité pour chaque canal pour chaque groupe de cellules.

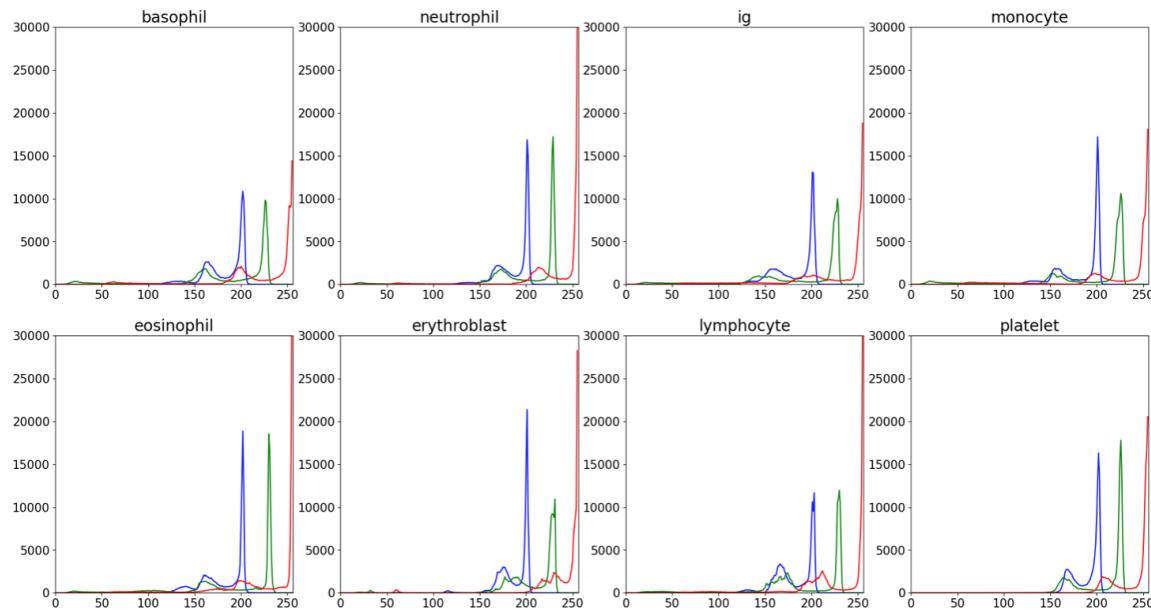


FIGURE 2.6 – Histogrammes d'intensité des canaux RGB correspondant aux cellules représentées à la fig.2.3

Dans l'échantillon précédent, nous constatons un problème au niveau des couleurs des images. Bien que les images des cellules aient été prises avec le même équipement, le traitement photographique est différent selon certaines classes (vérifié lors de l'extraction des données EXIF des images) (Fig. 2.7).

Pour rééquilibrer les couleurs, nous appliquons une méthode décrite dans l'article « Comparison of traditional image processing and deep learning approaches for classification of white blood cells in peripheral blood smear images ».

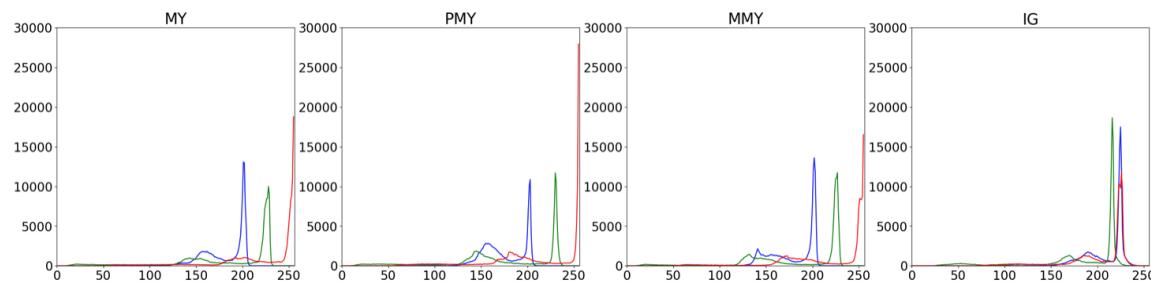


FIGURE 2.7 – Les histogrammes d'intensité des canaux RGB correspondant aux cellules représentées sur la fig 2.4. Ici, le panneau le plus à droite a des caractéristiques de couleur différentes des autres panneaux

Les images montrent à quel point la classification n'est pas une tâche aisée. La distinction entre les différentes cellules semble particulièrement difficile à première vue.

En effet, prenons l'exemple du degré de granularité chez certaines cellules. Cette particularité peut rendre la tâche plus complexe dans l'apprentissage de notre modèle et engendrer des erreurs de prédiction.

Cependant, nous pouvons constater que les images de notre dataset présentent une structure assez similaire et standard des cellules.

Il n'y a donc dans nos données aucune information ou biais liés à l'environnement / background. Il n'y aura donc aucun impact sur la modélisation.

L'ensemble de ces caractéristiques rendent la tâche de classification plus difficile mais elles sont essentielles dans l'apprentissage pour une meilleure performance du modèle.

### 2.3.3 Représentation du nombre d'échantillons par classe et par groupe

Le jeu de données contient 8 groupes (group) principaux de cellules dont certains sont subdivisés soit 13 classes (label) différentes (Table 2.1, Fig. ??, 2.9).

	group	label	size	pct (%)
0	BASOPHIL	BA	1218	7.13
1	EOSINOPHIL	EO	3117	18.24
2	ERYTHROBLAST	ERB	1551	9.07
3	IG	IG	151	0.88
4	IG	MMY	1015	5.94
5	IG	MY	1137	6.65
6	IG	PMY	592	3.46
7	LYMPHOCYTE	LY	1214	7.10
8	MONOCYTE	MO	1420	8.31
9	NEUTROPHIL	BNE	1633	9.55
10	NEUTROPHIL	NEUTROPHIL	50	0.29
11	NEUTROPHIL	SNE	1646	9.63
12	PLATELET	PLATELET	2348	13.74

TABLE 2.1 – Répartition des images selon les 13 classes

### 2.3.4 Visualisation de la répartition des images par label

La répartition entre les différents labels n'est pas uniforme (Fig. 2.9). Le label "EO" rassemble quasiment un cinquième de la totalité du jeu de données.

Le label Neutrophil ne contient que 50 images et l'IG, 151 images.

Il y a donc un fort déséquilibre dans notre base de données qu'il conviendra de corriger par des techniques de ré-échantillonnage (Over ou underSampling), d'augmentation de données ou de pondération.

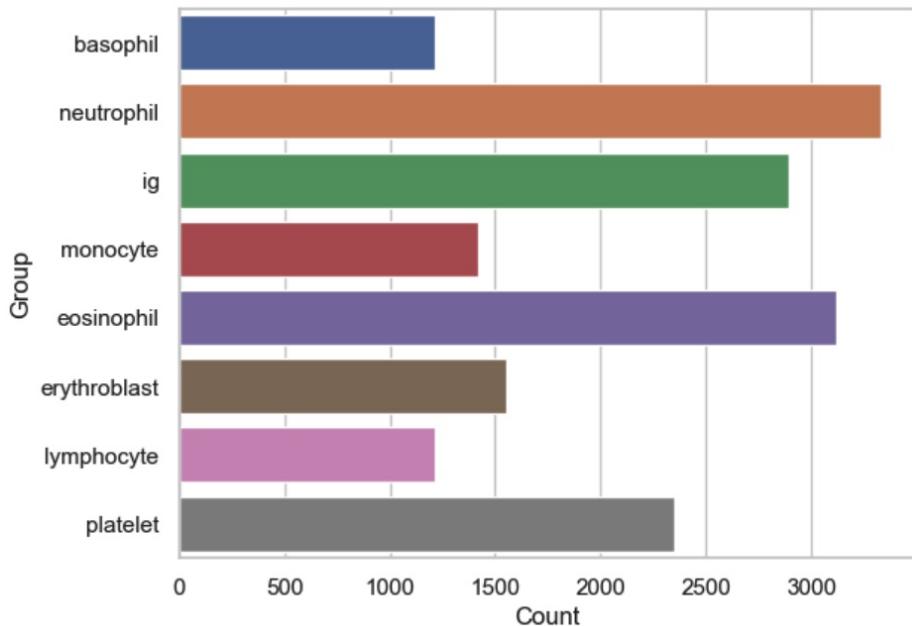


FIGURE 2.8 – Numéro de cellule par rapport au groupe de cellules

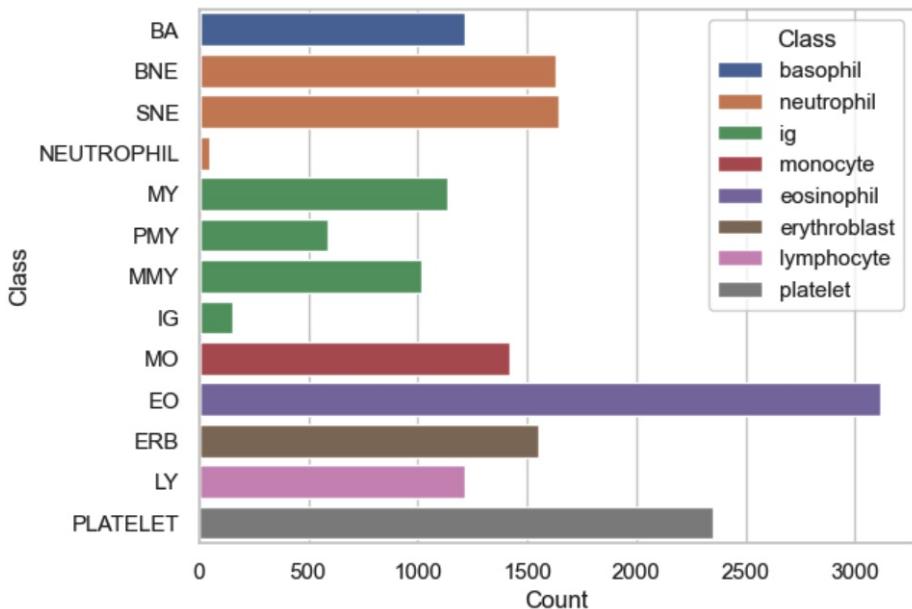


FIGURE 2.9 – Numéro de cellule par rapport au classe de cellules.

### 2.3.5 Analyse des images selon leur format

Width	Height	Count
359	360	1
	363	16639
360	360	198
	361	2
361	360	1
362	360	1
366	369	250

TABLE 2.2 – Different image sizes in the dataset.

Les images de la base de données ont une haute définition. Nous dénombrons 7 tailles d'image

différentes (Table 2.2). La dimension la plus commune est 360x363 pixels qui représentent 97% des échantillons. Il faudra donc procéder à un traitement d'image sur les 453 autres images pour les mettre à la même dimension.

Par la suite, pour réduire la mémoire (RAM) nécessaire et le temps de traitement sans pour autant perdre d'information, nous choisirons de réduire la taille des images.

Certaines cellules ont de grandes similarités, il nous a donc paru important de garder tout de même une bonne définition pour mieux les distinguer.

Pour le format initial, la modélisation porterait sur un nombre beaucoup plus élevé de paramètres que sur un format réduit.

### 2.3.6 Première conclusion

- Dans le jeu de données de 17 092 images, il y a 8 classes : 'basophile', 'neutrophile', 'ig', 'monocyte', 'éosinophile', 'érythrocyte', 'lymphocyte' et 'plasmacytoid dendritic cell'.
- La classe « neutrophile » comporte 3 types d'étiquettes : « BNE », « SNE » et « Neutrophile ».
- La classe "ig" comporte 4 types d'étiquettes : "MY", "PMY", "MMY", "IG".
- Chaque autre classe n'a qu'un seul type d'étiquette.
- Les images ont le même format.
- Les images sont de tailles différentes :
  - 16 639 images d'une taille de 363x360 (hxw),
  - 250 images de 369x366,
  - 201 images de 360x360,
  - 2 images de 361x360,
  - 1 image de 360x359,
  - 1 image de 360x362,
  - 1 image 360x361.
- **Les données sont déséquilibrées**

Cette 1ère analyse de données exploratoire (profilage des données) nous a permis d'obtenir des informations sur le contenu et la structure de notre jeu de données.

L'ensemble de nos visualisations apporte un éclairage sur le sens, la structure, les difficultés et les biais de notre jeu de données.

Détecter des biais éventuels dans les images permet d'adapter par la suite des techniques de preprocessing à effectuer sur notre jeu de données telles que :

- recadrer les images, ajouter ou supprimer de l'information équitablement sur chaque bord de l'image pour maintenir l'information utile au centre de l'image
- rééquilibrer les couleurs,
- augmenter les données dans certaines classes minoritaires pour rééquilibrer les classes
- et d'autres nouvelles techniques testées.

# Chapitre 3

## Preprocessing

### 3.1 Traitement des images

Lors de la phase d'exploration des images, nous avons constaté que le jeu de données n'était pas parfaitement homogène. Il présente 2 principaux problèmes :

1. Les dimensions des images ne sont pas toutes identiques. La taille majoritaire des images est de 360x363 pixels et environ 3% des images ne sont pas à cette taille.
2. Certaines images n'ont pas le même aspect visuel car elles ont été traitées avec des librairies JPEG différentes. Ainsi l'histogramme des couleurs n'est pas correctement équilibré d'une image à l'autre.

Nous aurions pu mettre de côté ces différentes images mais nous avons préféré les conserver et leur appliquer un traitement numérique pour harmoniser la totalité du jeu de données.

#### 3.1.1 Dimension des images

Plutôt que de redimensionner naïvement l'image et déformer son aspect (ratio), nous avons développé la fonction `central_pad_and_crop` qui pour une image donnée renvoie une nouvelle image standardisée sans toucher à la zone centrale de l'image (celle qui contient la cellule). Si la taille présente moins de pixels en hauteur et/ou largeur que la cible alors la fonction étend l'image en copiant à sa périphérie les pixels du bord autant de fois que nécessaire. Si, au contraire, la taille présente plus de pixels en hauteur et/ou largeur que la cible alors la fonction rogne l'image sur les bords. Ainsi, nous sommes assurés qu'aucune information n'est altérée au centre de l'image.

Le traitement se fait en 2 passes :

- Lorsqu'une dimension, largeur (width) ou hauteur (height), est plus petite que la cible (360x363) alors la fonction complète l'image (padding) en copiant l'information des bords (edge).
- Lorsqu'une dimension, largeur (width) ou hauteur (height), est plus grande que la cible (360x363) alors la fonction recadre l'image (cropping) en supprimant l'information des bords (edge).

Par exemple, dans le cas d'une image de 362x360, nous avons 2 pixels de plus sur la largeur et 3 pixels de moins sur la hauteur pour atteindre notre cible de 360x363. L'opération consiste donc à supprimer 2 pixels sur les bords gauche (1px) et droite de l'image (1px) et à ajouter 3 pixels sur les bords haut (1px) et bas (2px).

#### 3.1.2 Balance des couleurs

En ce qui concerne la colorimétrie, les figures 3.1 et 3.2 montrent 2 types de distribution de couleurs dans le jeu de données : un premier dans la majorité des échantillons où les canaux Rouge, Vert et Bleu ont différentes intensités (saturation en rouge) et un second dans lequel la distribution des couleurs est plus équilibrée.

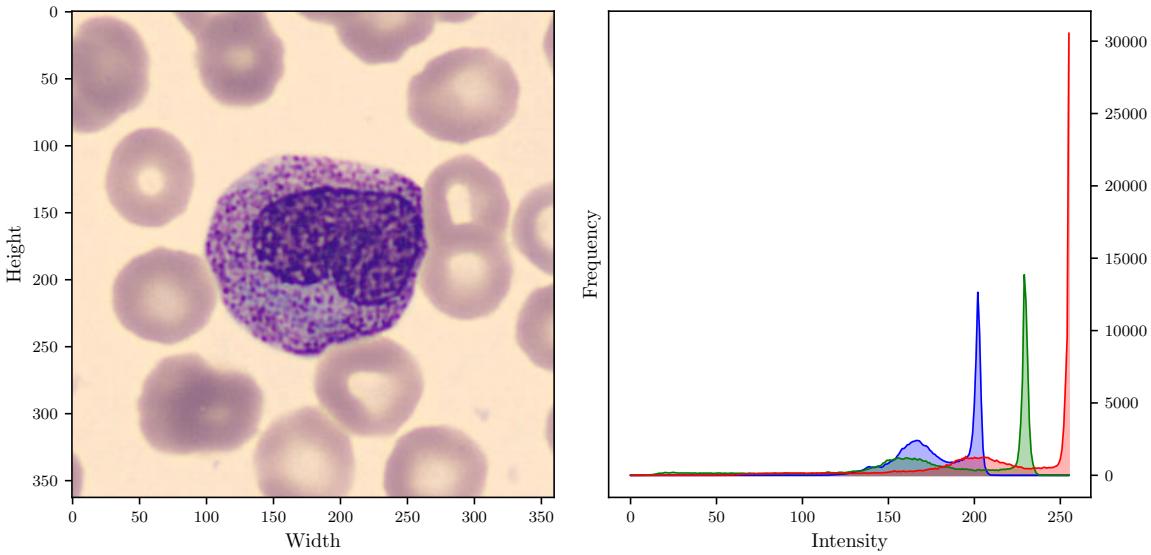


FIGURE 3.1 – Histogramme distribué

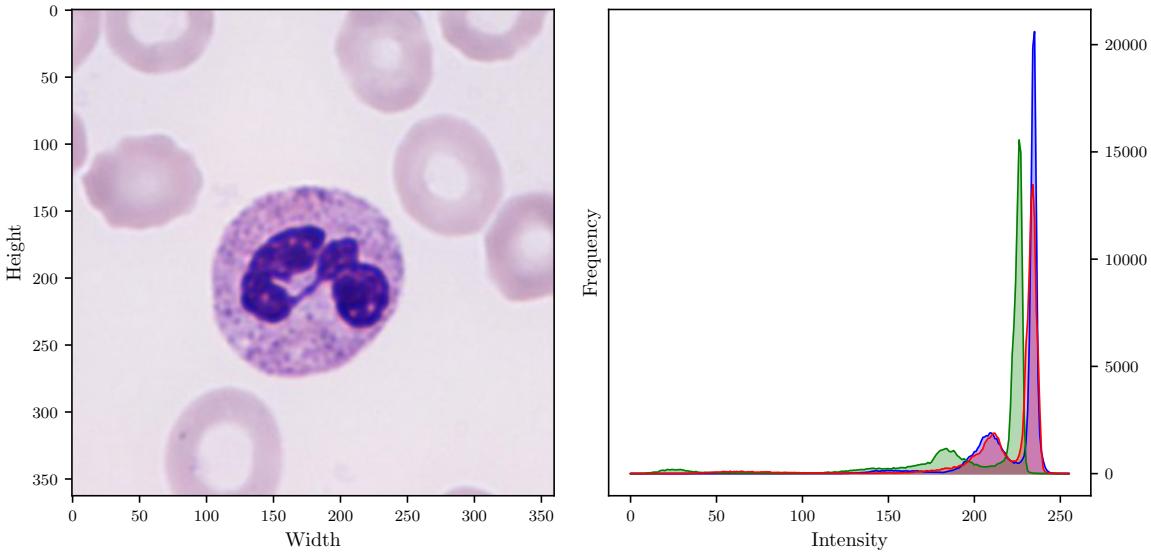


FIGURE 3.2 – Histogramme uniforme

Nous décidons de standardiser chaque image dans l'hypothèse que la couleur peut avoir une influence sur la prédiction dans le cas où une classe ne serait représentée que par un seul type d'histogramme. Pour cela, nous nous basons sur la méthode décrite dans l'article de [1] afin d'avoir un histogramme plus homogène. Cela consiste à régler la balance des couleurs pour chaque canal selon la formule suivante :

$$\text{new balanced component} = \text{old component} * \frac{\text{mean of grayscale image}}{\text{mean of old component}} \quad (3.1)$$

De plus, cette méthode a été reprise avec succès dans l'article [3] pour segmenter et extraire les cellules (noyau et une partie du cytoplasme) dans le cadre d'une classification par SVM. La figure 3.3 résulte de la normalisation des histogrammes des images précédentes. Nous observons une plus grande homogénéité dans les couleurs des images normalisées par rapport aux couleurs des images originales.

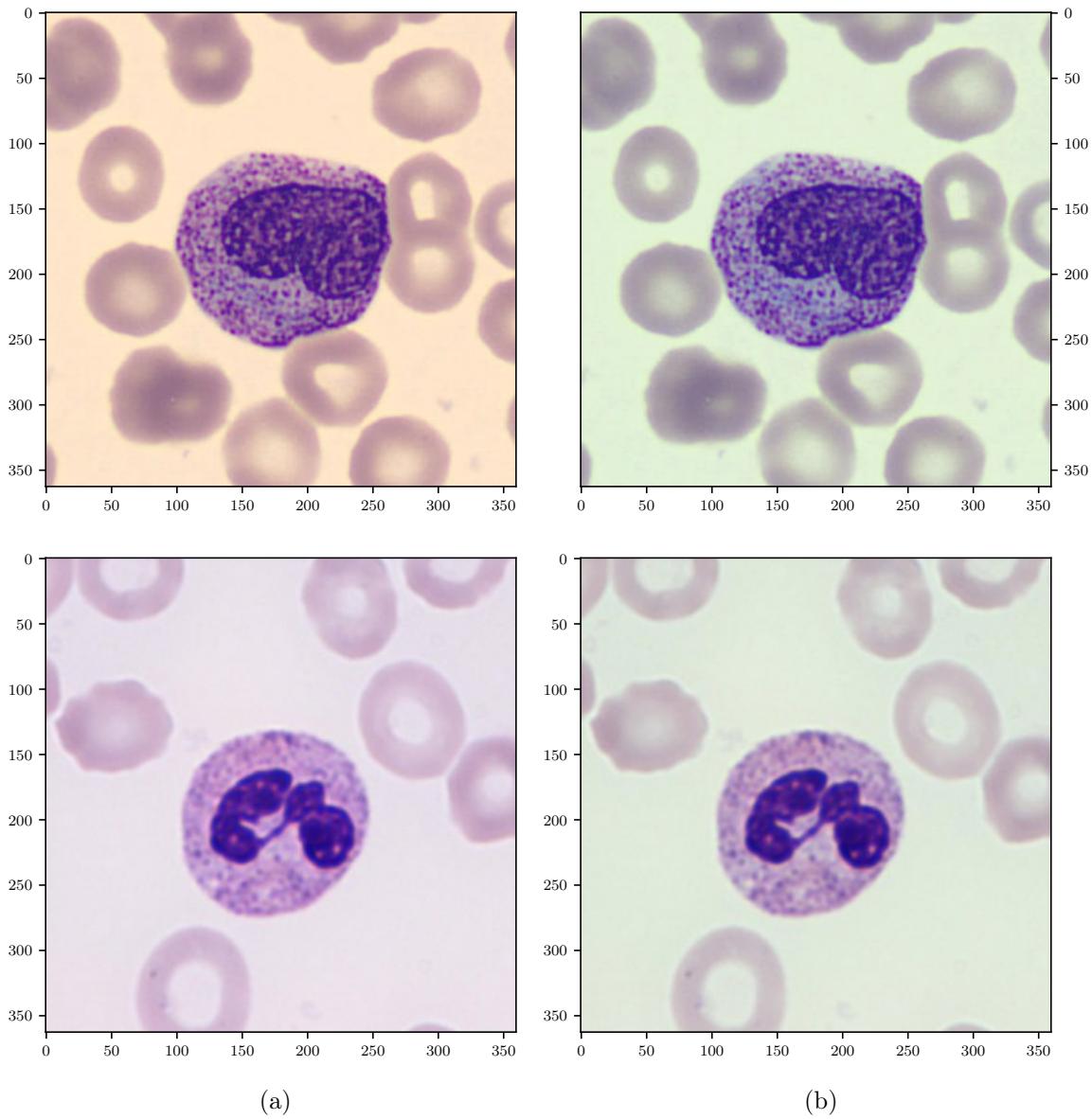


FIGURE 3.3 – Traitement des histogrammes. (a) images originales, (b) images normalisées

### 3.2 Segmentation par computer vision

Nous décidons d'aller plus loin et observons les images dans différents espaces de couleurs (RGB, HSV, HLS, CMYK) comme le montre la figure 3.4. Sur de très nombreux échantillons, nous nous apercevons qu'il est aisé d'extraire le noyau des cellules. Certes, la classification d'une cellule dépend en grande partie de son noyau mais également de son cytoplasme. Or, il apparaît que le cytoplasme se confond avec les érythrocytes (globules rouges) quand il est dense et le plasma sanguin quand il est clairsemé.

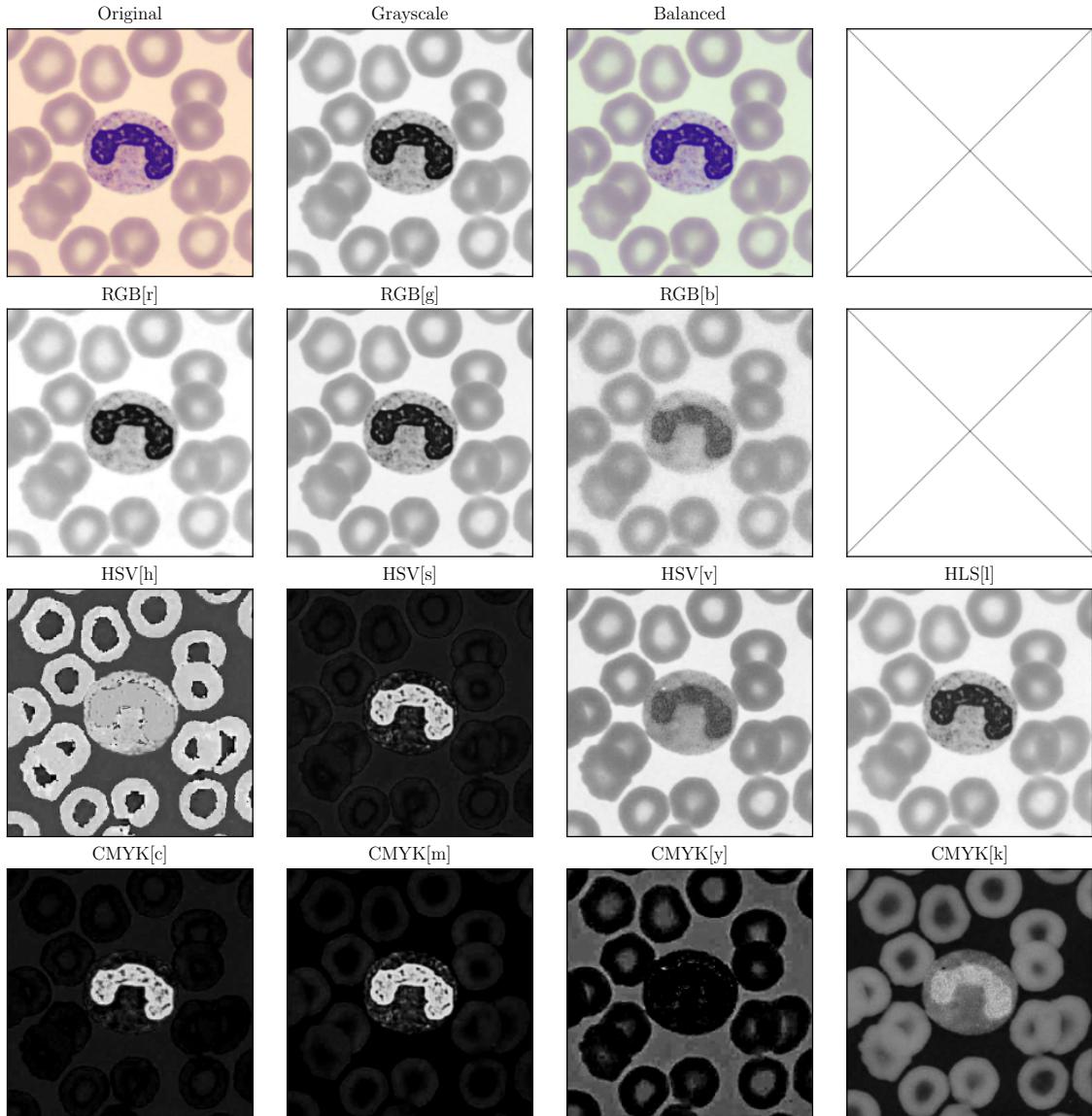


FIGURE 3.4 – Espaces colorimétriques

Nous émettons l'hypothèse qu'un modèle de classification donnera de meilleurs résultats s'il est concentré sur sa tâche sans bruit extérieur. Nous décidons donc d'extraire le lymphocyte de l'image et de masquer le fond grâce à des outils de vision par ordinateur, notamment la recherche de contours et l'algorithme du watershed.

Le principe est le suivant :

1. Charger une image du jeu de données
2. Convertir l'image en niveau de gris
3. Appliquer un filtre gaussien d'une taille de 5x5
4. Binariser l'image par seuillage (**threshold**)
5. Trouver les contours extérieurs (**findContours**)
6. Remplir les zones définies par les contours (**drawContours**)
7. Calculer la distance entre chaque pixel blanc et le pixel noir le plus proche (**distanceTransform**)
8. Trouver les coordonnées des maximums locaux à partir des distances précédentes (**peak\_local\_max**)
9. Segmenter les cellules présentes sur l'image selon l'algorithme [4]
10. Conserver uniquement la cellule la plus au centre de l'image (**regionprops**)

Ces opérations sont implémentées dans le code suivant :

```

1 # Charge l'image et la transforme en niveau de gris
2 img = cv2.cvtColor(cv2.imread('EO_45684.jpg'), cv2.COLOR_BGR2RGB)
3 gray = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
4 gray = cv2.GaussianBlur(gray, (5, 5), 0)
5
6 # Trouve les contours exterieurs des cellules
7 _, thresh = cv2.threshold(gray, 200, 255, cv2.THRESH_BINARY_INV)
8 contours, _ = cv2.findContours(thresh, cv2.RETR_EXTERNAL, 1)
9
10 # Cree un masque en remplissant les contours
11 zeros = np.zeros(thresh.shape, np.uint8)
12 mask = cv2.drawContours(zeros, contours, -1, 255, cv2.FILLED)
13
14 # Distance entre chaque pixel blanc et le pixel noir le plus proche
15 dist = cv2.distanceTransform(mask, cv2.DIST_L2, 5)
16
17 # Trouve les maxima locaux (environ le nombre de cellules de l'image)
18 coords = peak_local_max(dist, min_distance=20, labels=mask)
19
20 # Cree les marqueurs a partir des coordonnees des maxima
21 markers = np.zeros(dist.shape, dtype=np.uint8)
22 markers[tuple(coords.T)] = 1
23 markers = label(markers)
24
25 # Algorithme watershed
26 labels = watershed(-dist, markers, mask=mask)
27
28 # Extrait le leucocyte par minimisation de la distance
29 # entre le centre de la cellule et le centre de l'image
30 cx, cy = dist.shape[1] // 2, dist.shape[0] // 2
31 key = lambda x: np.sqrt((cx-x.centroid[1])**2 + (cy-x.centroid[0])**2)
32 wbc = min(regionprops(labels), key=key)

```

À l'issue de ces étapes, la cellule centrale est repérée dans l'image (figure 3.5)

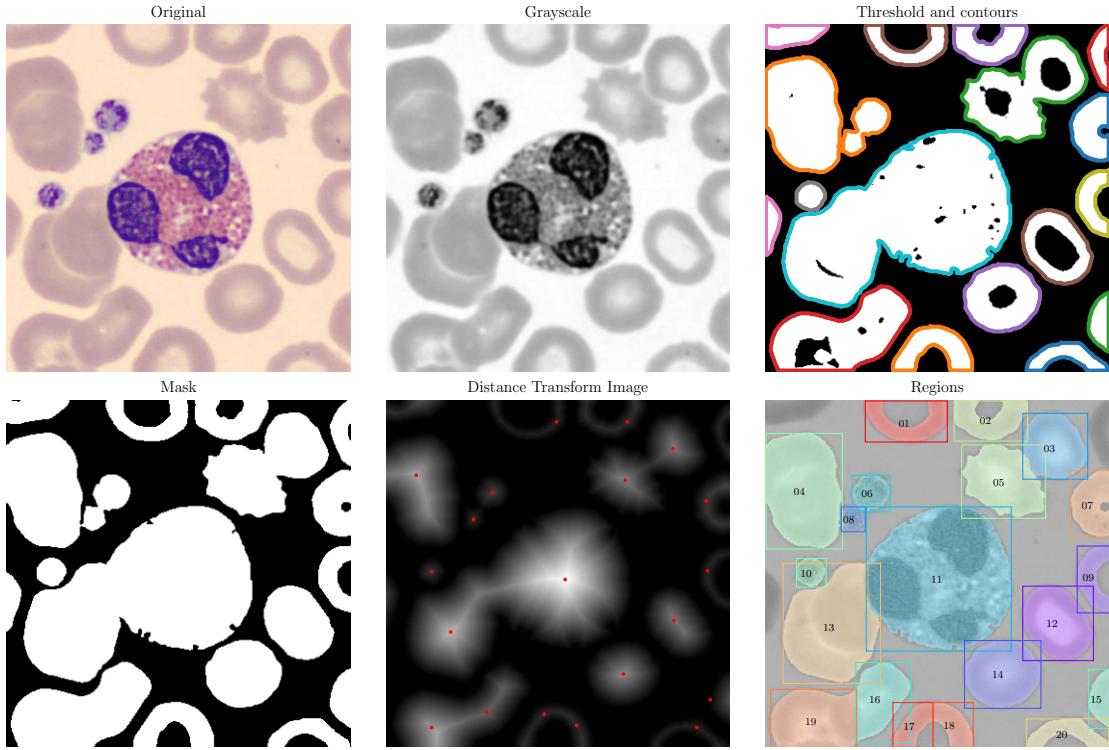


FIGURE 3.5 – Segmentation des cellules lymphocytaires.

Il ne reste plus qu'à récupérer le masque binaire pour isoler la cellule et à rogner l'image pour maximiser l'information utile tout en réduisant le nombre de features (pixels) en entrée du réseau de convolution. Le code ci-dessous est en charge de ce traitement :

```

1 y0, x0, y1, x1 = wbc.bbox
2 l = max(x1 - x0, y1 - y0)
3 r = (l // 2) + (l % 2)
4 cx, cy = (x0 + x1) // 2, (y0 + y1) // 2
5
6 cell1 = cv2.bitwise_and(rgb, rgb, mask=np.uint8(labels==wbc.label))
7 cell2 = cv2.resize(cell1[cy-r:cy+r, cx-r:cx+r], (256, 256))

```

Ce qui pour l'exemple précédent, nous permet d'obtenir l'image suivante :

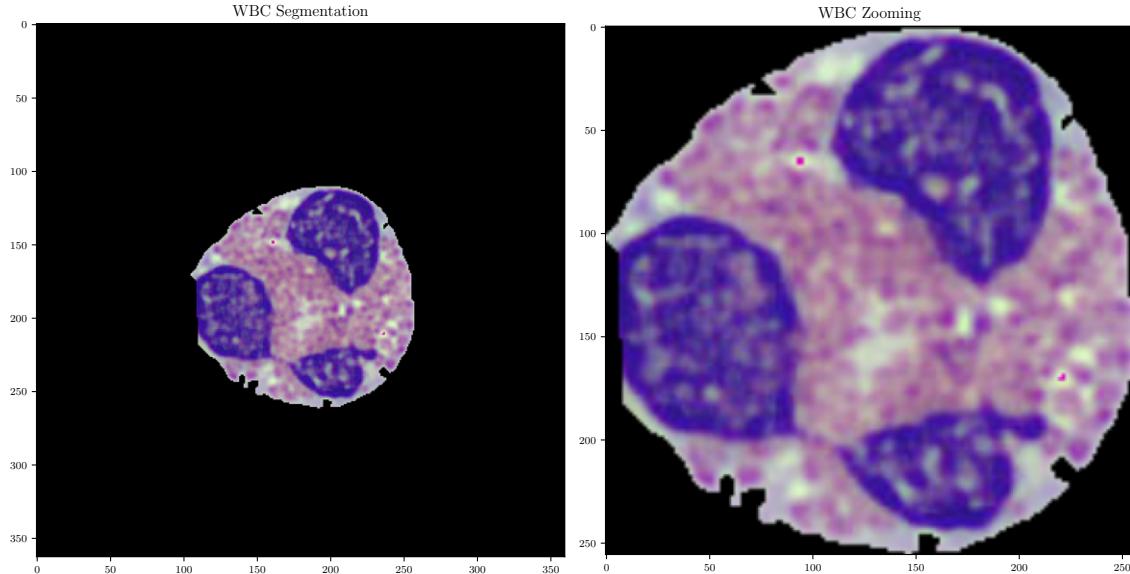


FIGURE 3.6 – Segmentation de la cellule

Le résultat de la segmentation basée sur les contrastes et la détection de contour donne des résultats plutôt satisfaisants sur l'ensemble du jeu de données. Néanmoins, il subsiste des erreurs d'extraction sur certaines cellules selon 3 origines identifiées :

- Par nature, les cellules sont plutôt convexes donc la détection des maximums locaux après la cartographie des distances nous indique qu'à un minimum local correspond une cellule. Or, certaines cellules présentes en certains endroits des parois concaves sur leur contour. Ainsi, il apparaît deux maximums locaux pour une seule et unique cellule réelle.
- A l'opposé du cas précédent, nous trouvons des érythroblastes qui se glissent sous le lymphocyte et l'algorithme de segmentation n'arrive pas à distinguer les 2 cellules.
- Quand le cytoplasme ne contient pas assez de granulocytes, l'intérieur de la cellule se confond avec le plasma et une partie de la cellule n'est pas segmentée par l'algorithme utilisé.

### 3.3 Segmentation par deep-learning

Nous essayons une autre méthode de segmentation basée sur le réseau spécialisé U-Net qui « *se compose d'une partie contractante et une voie expansive, ce qui lui confère une architecture en forme de «U». La partie contractante est un réseau de convolution typique qui consiste en une application répétée de convolutions, chacune suivie d'une unité linéaire rectifiée (ReLU) et d'une opération de pooling maximum. Pendant la contraction, les informations spatiales sont réduites tandis que les informations sur les caractéristiques sont augmentées. La voie expansive combine les informations de caractéristiques géographiques et spatiales à travers une séquence de convolutions et concaténations ascendantes avec des fonctionnalités haute résolution issues de la voie contractante*

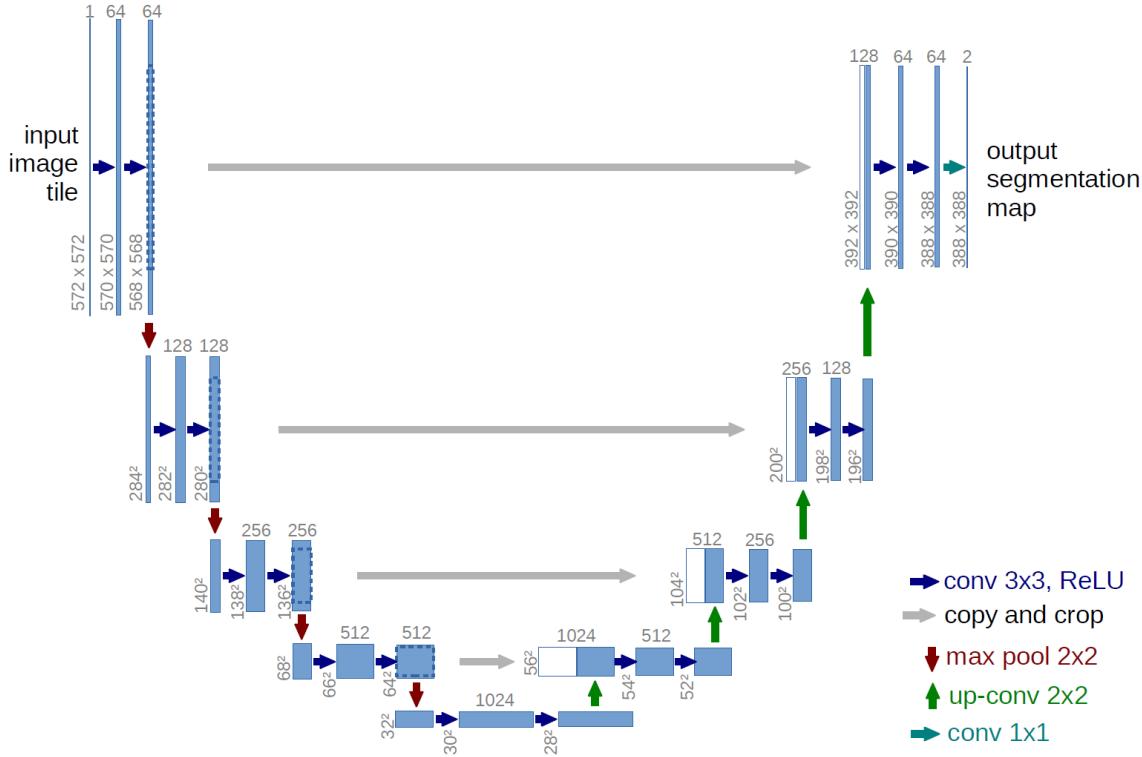


FIGURE 3.7 – Architecture du réseau U-Net

Nous utilisons U-Net dans sa version supervisée, c'est à dire que nous devons fournir au réseau une image source et un masque binaire de la région d'intérêt (ROI) qui doit être extraite. Le réseau va donc apprendre à repérer les ROI dans les images pour pouvoir les extraire plus tard par inférence. En sortie, le modèle nous renvoie un masque que nous pourrons comparer au masque original.

La problématique de cette méthode réside dans le fait qu'il faut créer les masques manuellement ce qui est une opération fastidieuse. Comme nous avons déjà généré des masques dans la tentative de segmentation précédente, nous allons les réutiliser en les contrôlant manuellement car nous avons une confiance limitée. Nous prélevons aléatoirement 1600 images ( $\simeq 10\%$ ) et masques que nous allons vérifier. Au final, nous retirons de notre jeu d'entraînement 259 images dont le masque n'est pas suffisamment précis. Il nous reste 1341 images que nous répartissons en 3 jeux de données : **train**, **valid** et **test**

Désormais, il est possible de construire le modèle. Comme ce dernier est très répétitif, nous développons des fonctions qui crée les groupes de blocs de convolution, d'encodage et de décodage :

```

1 from keras.models import Model
2 from keras.layers import (Input, Conv2D, BatchNormalization, Activation,
3 MaxPool2D, Conv2DTranspose, Concatenate)
4
5 INPUT_SHAPE = (128, 128, 3)
6 BATCH_SIZE = 16
7
8 def conv_block(inp, num_filters):
9     # Block 1
10    x = Conv2D(num_filters, (3, 3), strides=1, padding='same')(inp)
11    x = BatchNormalization()(x)
12    x = Activation('relu')(x)
13    # Block 2
14    x = Conv2D(num_filters, (3, 3), strides=1, padding='same')(x)
15    x = BatchNormalization()(x)
16    x = Activation('relu')(x)
17    return x
18

```

```

19 def encoder_block(inp, num_filters):
20     x = conv_block(inp, num_filters)
21     p = MaxPool2D((2, 2))(x)
22     return x, p
23
24 def decoder_block(inp, enc, num_filters):
25     x = Conv2DTranspose(num_filters, (2, 2), strides=2, padding='same')(inp)
26     x = Concatenate()([x, enc])
27     x = conv_block(x, num_filters)
28     return x
29
30 def create_model():
31     # Input
32     inputs = Input(INPUT_SHAPE)
33
34     # Contraction part (top-down)
35     e1, p1 = encoder_block(inputs, 64)
36     e2, p2 = encoder_block(p1, 128)
37     e3, p3 = encoder_block(p2, 256)
38     e4, p4 = encoder_block(p3, 512)
39
40     # Bottleneck
41     b1 = conv_block(p4, 1024)
42
43     # Expansion part (bottom-up)
44     d1 = decoder_block(b1, e4, 512)
45     d2 = decoder_block(d1, e3, 256)
46     d3 = decoder_block(d2, e2, 128)
47     d4 = decoder_block(d3, e1, 64)
48
49     # Output
50     outputs = Conv2D(1, 1, padding="same", activation='sigmoid')(d4)
51
52     model = Model(inputs, outputs, name="U-Net")
53     return model
54
55
56 model = create_model()
57 model.summary()
58 model.compile(optimizer='adam', loss='binary_crossentropy')

```

L'entraînement est très rapide malgré plus de 31 millions de paramètres à régler. Il est à noter que l'espace colorimétrique retenu est XYZ et non RGB car le modèle est plus stable.

Comme le modèle nous renvoie un masque binaire, nous pouvons le comparer avec le masque original. La classe positive est la valeur 255 (pixel blanc, ROI). Pour évaluer les performances d'extraction de notre modèle sur les images, nous évaluons à postériori les métriques suivantes :

- **precision** qui mesure le ratio entre les vrais positifs et les résultats réels,
- **recall** qui mesure le ratio entre les vrais positifs et les résultats prédicts,
- **f1-score** qui donne une note globale liant les 2 métriques précédentes :

$$f1\text{-score} = 2 * \frac{\text{precision} * \text{recall}}{\text{precision} + \text{recall}} \quad (3.2)$$

Afin de comparer les masques réels et prédicts, nous mettons en place une visualisation qui nous permettra de mieux comprendre les erreurs du modèle :

- les erreurs de Type I (faux positif) : les pixels noirs du masque original qui sont devenus blancs sur le masque prédict (pixels vert),

- les erreurs de Type II (faux négatif) : les pixels blancs du masque original qui sont restés noirs sur le masque prédit (pixels rouge).

Les résultats sont globalement bons comme le montre la figure 3.8 :

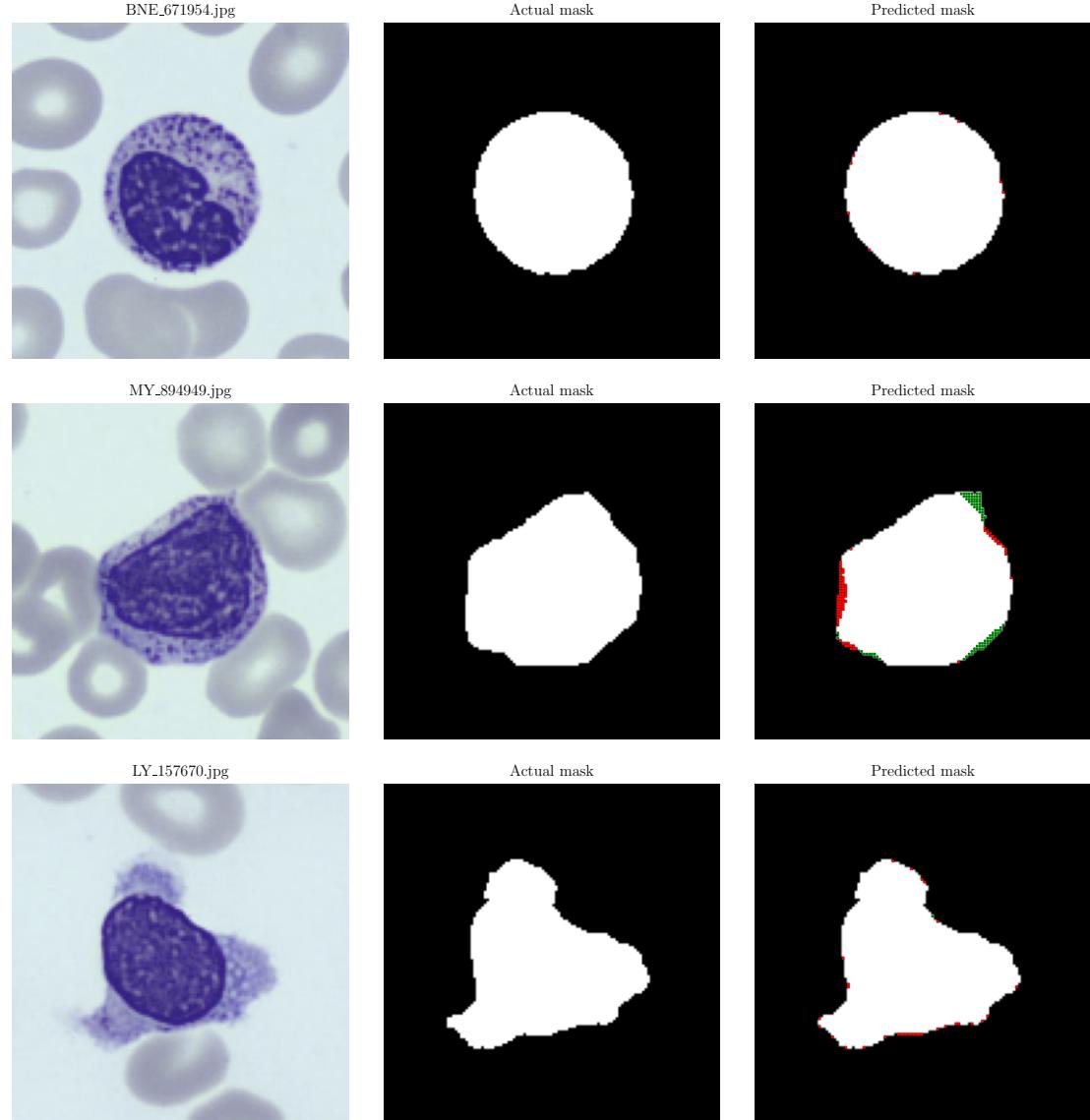


FIGURE 3.8 – Exemples de segmentation par U-Net

Il faut bien comprendre que les erreurs sont **relatives** au masque original (et ses erreurs de détection) et non pas par rapport à l'image de la cellule. Les erreurs peuvent très bien aller dans le bon sens quand la segmentation réalisée par U-Net est meilleure que la segmentation par contraste. En effet, sur plusieurs images, U-Net ajoute et retire des pixels au masque car il détecte mieux les contours, c'est notamment le cas dans la seconde image de la figure 3.8.

Cependant, pour être plus pertinent, il aurait fallu reprendre et corriger les masques des images que nous avons rejetés afin de donner une base plus solide d'apprentissage. A ce moment, les résultats sur l'extraction des cellules auraient été probablement bien meilleurs. En effet, il n'y a pas de raison que la segmentation par U-Net fasse beaucoup mieux que la première méthode étant donné que nous nous sommes contentés de donner à U-Net le meilleur de la première méthode d'extraction.

Malgré les bons résultats obtenus par les 2 méthodes de segmentation, le temps imparti pour réaliser ce projet nous a contraint de mettre de côté cette étape de préprocessing pour nous concentrer sur le pipeline de modélisation et plus particulièrement sur le fine-tuning.

# Chapitre 4

## Méthodes : Pipeline de modélisation d'un réseau de convolution

### 4.1 Étapes du pipeline de modélisation

Notre pipeline de modélisation respecte 6 étapes :

1. Installation de notre librairie et mise à jour des dépendances
2. Chargement des données d'imagerie médicale
3. Préparation des jeux de données : entraînement, validation et test
4. Modélisation d'un réseau de convolution :
  - (a) Apprentissage par transfer learning
  - (b) Optimisation par fine tuning
5. Prédiction sur un jeu d'images de test
6. Visualisation des résultats dont Grad-CAM

### 4.2 Modélisation d'un réseau de convolution

#### 4.2.1 Préparation de l'environnement

Afin de faciliter l'utilisation de notre travail et de nos fonctions, nous avons créé un paquet Python avec l'outil **Poetry**. Il est possible de l'installer avec la commande **pip** à partir d'un terminal ou d'un notebook (en le préfixant par «!»)

```
1 pip install git+https://github.com/DataScientest/yawbcc.git
```

Le paquet va installer et mettre à jour tous les paquets nécessaires au bon fonctionnement de nos codes comme installer la version 2.10 de Tensorflow et Keras. Le code a été testé et validé de Python 3.7 (environnement Colab) à Python 3.10 (environnement local).

#### 4.2.2 Jeux de données

Tout au long du projet, nous avons utilisé deux méthodes pour charger nos fichiers d'images du disque en mémoire et créer nos jeux de données :

1. Nous avons utilisé la classe **ImageDataGenerator** et ses méthodes **flow\_from\_directory** ou **flow\_from\_dataframe**. Cette dernière permettait de faire simultanément de l'augmentation de données en manipulant l'image (rotation, recadrage, translation, etc). Néanmoins, la classe a été marquée comme dépréciée par les développeurs de Keras.
2. Nous nous sommes tournés vers la fonction **image\_dataset\_from\_directory**. Cependant, cette fonction nécessite d'avoir une hiérarchie de répertoires qui représentent les jeux de données : un répertoire **train** pour le jeu d'entraînement puis des sous répertoires représentant les classes à prédire. C'était trop peu flexible pour répondre à notre besoin.

C'est ainsi que nous avons développé notre propre routine de chargement des images et de création des jeu de données. De notre expérience avec Scikit-Learn, nous voulions pouvoir utiliser la fonction `train_test_split` sur un dataframe pour pouvoir créer nos jeux de données.

Dans un premier temps, nous créons un dataframe des images trouvées récursivement dans un répertoire, nous enregistrons des informations comme le chemin absolu vers l'image, le nom de l'image, sa classe pour la prédiction, etc. À ce stade, aucune image n'est chargée en mémoire. Nous pouvons maintenant diviser notre dataframe en 3 jeux de données : `train`, `valid` et `test`.

```

1 # Find images of dataset
2 data = []
3 for file in DATA_DIR.glob('**/*.jpg'):
4     d = {'image': file.name,
5           'group': file.parent.name.upper(),
6           'label': file.stem.split('_')[0].upper(),
7           'path': str(file)}
8     data.append(d)
9
10 # Create dataframe and select columns
11 df = pd.DataFrame(data)
12 X = df['path']
13 y, cats = pd.factorize(df['group'])
14
15 # Split into 3 balanced datasets
16 X_train, X_test, y_train, y_test = \
17     train_test_split(X, y, test_size=0.2, stratify=y)
18
19 X_train, X_valid, y_train, y_valid = \
20     train_test_split(X_train, y_train, test_size=0.2, stratify=y_train)
```

Dans les variables  $X_*$ , nous avons le chemin absolu vers l'image et dans les variables  $y_*$  le classe sous forme numérique utilisable avec la fonction de perte `sparse_categorical_crossentropy`.

Pour éviter de saturer la mémoire, Tensorflow réalise son apprentissage en traitant les données par lot (`batch`) plus ou moins grand. Il nous reste donc à implémenter un mécanisme qui permet à Tensorflow de venir chercher nos données à la demande, c'est ce que permet la classe `Sequence` de Keras dont nous héritons :

```

1 class WBCDataSequence(tf.keras.utils.Sequence):
2     def __init__(self, image_set, label_set, batch_size, image_size):
3         self.image_set = np.array(image_set)
4         self.label_set = np.array(label_set)
5         self.batch_size = batch_size
6         self.image_size = image_size
7
8     def __get_image(self, image):
9         """Load an image from disk and convert it as numpy array."""
10        image = tf.keras.preprocessing.image.load_img(image)
11        image_arr = tf.keras.preprocessing.image.img_to_array(image)
12        image_arr = central_pad_and_crop(image_arr, self.image_size)
13        return image_arr
14
15    def __get_data(self, images, labels):
16        """Collect X and y data for a batch."""
17        image_batch = np.array([self.__get_image(i) for i in images])
18        label_batch = np.array(labels)
19        return image_batch, label_batch
20
21    def __getitem__(self, index):
22        """Get requested data for a batch."""
23        images = self.image_set[index * self.batch_size:(index + 1) *
24                                * self.batch_size]
```

```

25     labels = self.label_set[index * self.batch_size:(index + 1)
26                             * self.batch_size]
27     images, labels = self._get_data(images, labels)
28     return images, labels
29
30   def __len__(self):
31     """Get number of batch contains in the dataset."""
32     return len(self.image_set) // self.batch_size \
33           + (len(self.image_set) % self.batch_size > 0)

```

Il ne reste plus qu'à créer 3 instances de cette classe pour chacun de nos jeux de données :

```

1 INPUT_SHAPE = (256, 256, 3)
2 BATCH_SIZE = 32
3
4 params = {'image_size': INPUT_SHAPE[:2], 'batch_size': BATCH_SIZE}
5
6 train_ds = WBCDataSequence(X_train, y_train, **params)
7 valid_ds = WBCDataSequence(X_valid, y_valid, **params)
8 test_ds = WBCDataSequence(X_test, y_test, **params)

```

#### 4.2.3 Sélection de 4 modèles pré entraînés adaptés à la classification simple d'image

Pour répondre à notre problématique de classification d'images, les réseaux de neurones convolutifs (CNN) sont l'approche la mieux adaptée. De nombreux modèles de type CNN sont déjà créés et très efficaces dans le cas de notre problème. La création d'un réseau CNN équivalent à ceux déjà proposés aurait été d'un niveau bien supérieur à celui de la formation, et n'aurait pas été possible dans le temps imparti. C'est pourquoi nous utiliserons 4 modèles sélectionnés pour leur renommée dans le cadre de la classification d'images, parmi ceux présents dans la partie Applications de la librairie Keras. Voici les modèles choisis :

- VGG16
- MobileNetV2
- Xception
- ResNet50V2

### 4.3 Crédit et utilisation des modèles

Dans cette partie, nous allons détailler la manière dont nous avons mis en place les, de leur création à leur évaluation, en passant par la phase d'entraînement.

#### 4.3.1 Implémentation des modèles

##### Initialisation par Transfer Learning

Nous avons essayé de recréer l'architecture des modèles nous-mêmes, mais les résultats n'étaient pas concluant du tout. C'est pourquoi nous avons opté pour une méthode très efficace dans la classification d'image : le Transfer Learning. Elle consiste en l'initialisation d'un modèle, appelé applicatif, déjà entraîné sur un dataset conséquent, ici "ImageNet". Cet ajout comprend simplement le chargement de poids, non plus aléatoire, mais déjà définis et optimisé pour la reconnaissance d'images parmi 1000 classes (qui ne sont pas des cellules).

Le Transfer Learning se fait donc en plusieurs étapes :

- récupérer le modèle applicatif entraîné sur le dataset ImageNet
- geler les couches du modèle importé (pour ne pas modifier les poids importés)
- créer et ajouter des couches de têtes pour la classification
- entraîner le modèle complet uniquement sur les couches de classification nouvellement créées

Afin de standardiser le processus de création de nos modèles, nous avons créée la fonction `model_factory` qui se charge d'assembler les différentes couches en n'ayant besoin que de changer le nom de `app_model`, le modèle pré-entraîné disponible dans la liste des applications de Keras. Ici, nous avons donc utilisé 4 modèles différents, initialisés avec les poids "ImageNet" afin d'avoir un entraînement déjà efficace. Il nous a donc suffit de "geler" les couches déjà entraînées, et de rajouter les couches de classification adéquates.

`include_top` permet de charger les couches de classification du modèle. Nous l'avons initialisé "False" car nous voulons créer notre propre bloc de classification : le `top_net`. Les couches de têtes, qui permettent de récupérer la sortie du modèle pré-entraîné et de finaliser la classification, sont donc définies sous forme de bloc comme suit :

```
1 top_net = [
2     Dropout(0.2, seed=2022, name='dropout'),
3     Flatten('channels_last', name='flatten'),
4     Dense(256, activation='relu', name='fc1'),
5     Dense(256, activation='relu', name='fc2'),
6     Dense(NUM_CLASSES, activation='softmax', name='predictions')
7 ]
```

L'une de nos préoccupations était de réduire (voire supprimer) le sur-apprentissage. C'est pour ça qu'intervient la couche "Dropout(0.2)" qui va permettre au modèle d'oublier 20% de son entraînement à chaque itération, le forçant à "remettre en question" son apprentissage et donc réduire l'écart entre ce qu'il apprend, et ce qu'il prédit correctement. L'addition d'une couche Dropout est une méthode courante dans la réduction du phénomène de sur-apprentissage.

Nous avons appliqué de légères transformations géométriques sur nos images, via un bloc de couches de transformations, à savoir :

- une couche de symétrie horizontale
- une couche de rotation aléatoire sur une plage de -20% à 30% (de rotation de l'image)

```
1 transformers = [
2     RandomFlip(mode='horizontal', seed=2022, name='random_flip'),
3     RandomRotation(factor=(-0.2, 0.3), seed=2022, name='random_rotation')
4 ]
```

Ces couches ne sont actives que pendant l'entraînement. Elles sont inactives lorsque le modèle est utilisé en mode inférence dans `model.evaluate`. Elles permettent donc de fournir plus de diversité dans l'entraînement du modèle, de l'améliorer et de réduire le sur-apprentissage.

Enfin, pour optimiser d'avantage le modèle, nous utilisons la méthode du fine-tuning. Elle repose sur le fait de dégeler une partie des couches du modèle applicatif, afin d'entraîner un échantillon de ce dernier sur les images fournies en input, et non sur le dataset "ImageNet" comme expliqué précédemment. Ainsi, une partie de l'entraînement est effectuée avec le modèle applicatif complètement gelé, puis la deuxième partie de l'entraînement est réalisée avec le modèle applicatif partiellement gelé.

## Structure finale du modèle

Pour résumer, la structure de chaque modèle est la suivante :

- une couche d'entrée avec une forme de (256, 256, 3)
- un bloc de transformation
- un modèle applicatif
- un bloc de couches de tête de réseau dont la dernière couche de classification (8 classes)

## Entraînement du modèle

L'entraînement du modèle se déroule donc en 2 étapes :

- 1. L'apprentissage par Transfer Learning avec VGG16 complètement gelé : de l'epoch 1 à 10 (taux d'apprentissage par défaut)
- 2. L'apprentissage avec fine-tuning (VGG16 partiellement dégelé) : de l'epoch 11 à 20 (taux d'apprentissage de 0.00001 pour chercher plus finement les paramètres optimaux)

### Sauvegarde du modèle

Finalement, afin de pouvoir le réutiliser ultérieurement sans avoir besoin de l'entraîner à nouveau, nous récupérons les poids des modèles entraînés sous forme de fichiers .hdf5 comme suit :

- **WBC-{nom\_du\_modèle}\_tl.hdf5** pour le modèle issu du transfer learning sur l'ensemble du jeu de données de Barcelone,
- **WBC-{nom\_du\_modèle}\_ft.hdf5** pour le modèle issu du fine tuning sur l'ensemble du jeu de données de Barcelone

#### 4.3.2 Implémentation du modèle VGG16

Le premier modèle de base pour le Transfer Learning était VGG16. Le modèle final a été construit selon le processus décrit dans la partie 4.3.1. Dans le cadre de l'utilisation de VGG16 comme modèle applicatif, la structure du modèle (obtenue avec `model.summary()`) se présente de cette manière :

Model: "WBC-VGG16"		
Layer (type)	Output Shape	Param #
input (InputLayer)	[None, 256, 256, 3]	0
random_flip (RandomFlip)	(None, 256, 256, 3)	0
random_rotation (RandomRotation)	(None, 256, 256, 3)	0
preprocess (Lambda)	(None, 256, 256, 3)	0
vgg16 (Functional)	(None, 512)	14714688
dropout (Dropout)	(None, 512)	0
flatten (Flatten)	(None, 512)	0
fc1 (Dense)	(None, 256)	131328
fc2 (Dense)	(None, 256)	65792
predictions (Dense)	(None, 8)	2056
<hr/>		
Total params:	14,913,864	
Trainable params:	199,176	
Non-trainable params:	14,714,688	

FIGURE 4.1 – Résumé du modèle avant fine-tuning : 199 176 paramètres à entraîner

Model: "WBC-VGG16"		
Layer (type)	Output Shape	Param #
input (InputLayer)	[None, 256, 256, 3]	0
random_flip (RandomFlip)	(None, 256, 256, 3)	0
random_rotation (RandomRotation)	(None, 256, 256, 3)	0
preprocess (Lambda)	(None, 256, 256, 3)	0
vgg16 (Functional)	(None, 512)	14714688
dropout (Dropout)	(None, 512)	0
flatten (Flatten)	(None, 512)	0
fc1 (Dense)	(None, 256)	131328
fc2 (Dense)	(None, 256)	65792
predictions (Dense)	(None, 8)	2056
<hr/>		
Total params:	14,913,864	
Trainable params:	7,278,600	
Non-trainable params:	7,635,264	

FIGURE 4.2 – Résumé du modèle après fine-tuning : 7 278 600 paramètres à entraîner

Ainsi, la multiplication par 14 du nombre de paramètres à entraîner entre le modèle avec/sans fine-tuning, nous a montré tout l'intérêt d'utiliser cette méthode.

Pour le modèle VGG16, il a été fait le choix de ne dégeler que les 5 dernières couches du modèle applicatif, correspondant au dernier bloc de convolution. Voici le résultat des métriques au fil de l'entraînement, pour chaque époque :

```

Epoch 1/10
342/342 [=====] - 132s 358ms/step - loss: 0.9451 - accuracy: 0.6747 - val_loss: 0.3928 - val_accuracy: 0.8658
Epoch 2/10
342/342 [=====] - 111s 325ms/step - loss: 0.5598 - accuracy: 0.8031 - val_loss: 0.3779 - val_accuracy: 0.8658
Epoch 3/10
342/342 [=====] - 111s 326ms/step - loss: 0.4872 - accuracy: 0.8288 - val_loss: 0.3060 - val_accuracy: 0.8925
Epoch 4/10
342/342 [=====] - 112s 327ms/step - loss: 0.4443 - accuracy: 0.8457 - val_loss: 0.2923 - val_accuracy: 0.9046
Epoch 5/10
342/342 [=====] - 111s 325ms/step - loss: 0.4184 - accuracy: 0.8545 - val_loss: 0.2594 - val_accuracy: 0.9130
Epoch 6/10
342/342 [=====] - 112s 326ms/step - loss: 0.4045 - accuracy: 0.8592 - val_loss: 0.2705 - val_accuracy: 0.9075
Epoch 7/10
342/342 [=====] - 111s 324ms/step - loss: 0.4102 - accuracy: 0.8565 - val_loss: 0.2847 - val_accuracy: 0.9046
Epoch 8/10
342/342 [=====] - 111s 325ms/step - loss: 0.3851 - accuracy: 0.8615 - val_loss: 0.2536 - val_accuracy: 0.9177
Epoch 9/10
342/342 [=====] - 111s 325ms/step - loss: 0.3773 - accuracy: 0.8688 - val_loss: 0.2390 - val_accuracy: 0.9181
Epoch 10/10
342/342 [=====] - 111s 323ms/step - loss: 0.3656 - accuracy: 0.8681 - val_loss: 0.2520 - val_accuracy: 0.9185
107/107 [=====] - 25s 237ms/step - loss: 0.2551 - accuracy: 0.9175
Loss function: 0.2551
Accuracy: 0.9175

```

FIGURE 4.3 – Entraînement des epochs 1 - 10 sans fine-tuning

```

Epoch 11/20
342/342 [=====] - 122s 349ms/step - loss: 0.2038 - accuracy: 0.9266 - val_loss: 0.1202 - val_accuracy: 0.9547
Epoch 12/20
342/342 [=====] - 120s 351ms/step - loss: 0.1416 - accuracy: 0.9506 - val_loss: 0.1090 - val_accuracy: 0.9594
Epoch 13/20
342/342 [=====] - 120s 349ms/step - loss: 0.1058 - accuracy: 0.9639 - val_loss: 0.0930 - val_accuracy: 0.9700
Epoch 14/20
342/342 [=====] - 119s 347ms/step - loss: 0.0929 - accuracy: 0.9677 - val_loss: 0.0832 - val_accuracy: 0.9733
Epoch 15/20
342/342 [=====] - 118s 344ms/step - loss: 0.0854 - accuracy: 0.9715 - val_loss: 0.0757 - val_accuracy: 0.9773
Epoch 16/20
342/342 [=====] - 120s 352ms/step - loss: 0.0752 - accuracy: 0.9736 - val_loss: 0.0699 - val_accuracy: 0.9795
Epoch 17/20
342/342 [=====] - 120s 352ms/step - loss: 0.0635 - accuracy: 0.9769 - val_loss: 0.0654 - val_accuracy: 0.9781
Epoch 18/20
342/342 [=====] - 119s 347ms/step - loss: 0.0611 - accuracy: 0.9796 - val_loss: 0.0675 - val_accuracy: 0.9773
Epoch 19/20
342/342 [=====] - 118s 345ms/step - loss: 0.0566 - accuracy: 0.9807 - val_loss: 0.0644 - val_accuracy: 0.9795
Epoch 20/20
342/342 [=====] - 119s 346ms/step - loss: 0.0511 - accuracy: 0.9814 - val_loss: 0.0658 - val_accuracy: 0.9821
107/107 [=====] - 18s 168ms/step - loss: 0.0619 - accuracy: 0.9804
Loss function: 0.0619
Accuracy: 0.9804

```

FIGURE 4.4 – Entraînement des epochs 11 - 20 avec fine-tuning

#### 4.3.3 Implémentation du modèle MobileNetV2

Le deuxième modèle était MobileNetV2. Le procédé de création du modèle final ne change pas. Dans le cadre de l'utilisation de MobileNetV2 comme modèle applicatif, la structure du modèle (obtenue avec `model.summary()`) se présente de cette manière :

Model: "WBC-MobileNetV2"		
Layer (type)	Output Shape	Param #
input (InputLayer)	[ (None, 256, 256, 3) ]	0
random_flip (RandomFlip)	(None, 256, 256, 3)	0
random_rotation (RandomRotation)	(None, 256, 256, 3)	0
preprocess (Lambda)	(None, 256, 256, 3)	0
mobilenetv2_1.00_224 (Functional)	(None, 1280)	2257984
dropout (Dropout)	(None, 1280)	0
flatten (Flatten)	(None, 1280)	0
fc1 (Dense)	(None, 256)	327936
fc2 (Dense)	(None, 256)	65792
predictions (Dense)	(None, 8)	2056

=====

Total params: 2,653,768  
Trainable params: 395,784  
Non-trainable params: 2,257,984

FIGURE 4.5 – Résumé du modèle avant fine-tuning : 199 176 paramètres à entraîner

Model: "WBC-MobileNetV2"		
Layer (type)	Output Shape	Param #
input (InputLayer)	[ (None, 256, 256, 3) ]	0
random_flip (RandomFlip)	(None, 256, 256, 3)	0
random_rotation (RandomRotation)	(None, 256, 256, 3)	0
preprocess (Lambda)	(None, 256, 256, 3)	0
mobilenetv2_1.00_224 (Functional)	(None, 1280)	2257984
dropout (Dropout)	(None, 1280)	0
flatten (Flatten)	(None, 1280)	0
fc1 (Dense)	(None, 256)	327936
fc2 (Dense)	(None, 256)	65792
predictions (Dense)	(None, 8)	2056

=====

Total params: 2,653,768  
Trainable params: 1,601,864  
Non-trainable params: 1,051,904

FIGURE 4.6 – Résumé du modèle après fine-tuning : 807 944 paramètres à entraîner

Pour le modèle MobileNetV2, il a été fait le choix de ne dégeler que les 21 dernières couches du modèle applicatif, correspondant au dernier bloc de convolution. Voici le résultat des métriques au fil de l’entraînement, pour chaque époque :

```
Epoch 1/10
342/342 [=====] - 75s 205ms/step - loss: 0.4039 - accuracy: 0.8626 - val_loss: 0.2455 - val_accuracy: 0.9166
Epoch 2/10
342/342 [=====] - 68s 199ms/step - loss: 0.2193 - accuracy: 0.9234 - val_loss: 0.1774 - val_accuracy: 0.9335
Epoch 3/10
342/342 [=====] - 68s 198ms/step - loss: 0.1919 - accuracy: 0.9338 - val_loss: 0.1757 - val_accuracy: 0.9411
Epoch 4/10
342/342 [=====] - 67s 195ms/step - loss: 0.1910 - accuracy: 0.9336 - val_loss: 0.1841 - val_accuracy: 0.9342
Epoch 5/10
342/342 [=====] - 67s 194ms/step - loss: 0.1744 - accuracy: 0.9387 - val_loss: 0.1728 - val_accuracy: 0.9367
Epoch 6/10
342/342 [=====] - 66s 193ms/step - loss: 0.1673 - accuracy: 0.9411 - val_loss: 0.1457 - val_accuracy: 0.9514
Epoch 7/10
342/342 [=====] - 67s 195ms/step - loss: 0.1635 - accuracy: 0.9435 - val_loss: 0.1372 - val_accuracy: 0.9528
Epoch 8/10
342/342 [=====] - 66s 194ms/step - loss: 0.1564 - accuracy: 0.9457 - val_loss: 0.1448 - val_accuracy: 0.9477
Epoch 9/10
342/342 [=====] - 66s 194ms/step - loss: 0.1527 - accuracy: 0.9442 - val_loss: 0.1447 - val_accuracy: 0.9473
Epoch 10/10
342/342 [=====] - 66s 194ms/step - loss: 0.1451 - accuracy: 0.9505 - val_loss: 0.1260 - val_accuracy: 0.9565
107/107 [=====] - 12s 114ms/step - loss: 0.1285 - accuracy: 0.9547
Loss function: 0.1285
Accuracy: 0.9547
```

FIGURE 4.7 – Entraînement des epochs 1 - 10 sans fine-tuning

```

Epoch 11/20
342/342 [=====] - 75s 205ms/step - loss: 0.1163 - accuracy: 0.9571 - val_loss: 0.1087 - val_accuracy: 0.9572
Epoch 12/20
342/342 [=====] - 69s 202ms/step - loss: 0.0954 - accuracy: 0.9664 - val_loss: 0.1037 - val_accuracy: 0.9634
Epoch 13/20
342/342 [=====] - 69s 202ms/step - loss: 0.0903 - accuracy: 0.9683 - val_loss: 0.0883 - val_accuracy: 0.9678
Epoch 14/20
342/342 [=====] - 69s 200ms/step - loss: 0.0806 - accuracy: 0.9727 - val_loss: 0.0881 - val_accuracy: 0.9697
Epoch 15/20
342/342 [=====] - 70s 205ms/step - loss: 0.0726 - accuracy: 0.9742 - val_loss: 0.0786 - val_accuracy: 0.9737
Epoch 16/20
342/342 [=====] - 69s 202ms/step - loss: 0.0714 - accuracy: 0.9747 - val_loss: 0.0876 - val_accuracy: 0.9718
Epoch 17/20
342/342 [=====] - 68s 199ms/step - loss: 0.0660 - accuracy: 0.9769 - val_loss: 0.0822 - val_accuracy: 0.9733
Epoch 18/20
342/342 [=====] - 72s 210ms/step - loss: 0.0647 - accuracy: 0.9756 - val_loss: 0.0767 - val_accuracy: 0.9755
Epoch 19/20
342/342 [=====] - 69s 200ms/step - loss: 0.0623 - accuracy: 0.9768 - val_loss: 0.0775 - val_accuracy: 0.9748
Epoch 20/20
342/342 [=====] - 68s 200ms/step - loss: 0.0592 - accuracy: 0.9792 - val_loss: 0.0729 - val_accuracy: 0.9755
107/107 [=====] - 12s 111ms/step - loss: 0.0795 - accuracy: 0.9710
Loss function: 0.0795
Accuracy: 0.9710

```

FIGURE 4.8 – Entraînement des epochs 11 - 20 avec fine-tuning

#### 4.3.4 Implémentation du modèle Xception

Le troisième modèle applicatif était Xception. Le procédé de création du modèle final ne change pas. Dans le cadre de l'utilisation de Xception comme modèle applicatif, voici un aperçu de l'architecture du modèle (obtenue avec `model.summary()`) :

Model: "WBC-Xception"		
Layer (type)	Output Shape	Param #
input (InputLayer)	[None, 256, 256, 3]	0
random_flip (RandomFlip)	(None, 256, 256, 3)	0
random_rotation (RandomRotation)	(None, 256, 256, 3)	0
preprocess (Lambda)	(None, 256, 256, 3)	0
xception (Functional)	(None, 2048)	20861480
dropout (Dropout)	(None, 2048)	0
flatten (Flatten)	(None, 2048)	0
fc1 (Dense)	(None, 256)	524544
fc2 (Dense)	(None, 256)	65792
predictions (Dense)	(None, 8)	2056

Total params: 21,453,872  
Trainable params: 592,392  
Non-trainable params: 20,861,480

FIGURE 4.9 – Résumé du modèle avant fine-tuning : 592 392 paramètres à entraîner

Model: "WBC-Xception"		
Layer (type)	Output Shape	Param #
input (InputLayer)	[None, 256, 256, 3]	0
random_flip (RandomFlip)	(None, 256, 256, 3)	0
random_rotation (RandomRotation)	(None, 256, 256, 3)	0
preprocess (Lambda)	(None, 256, 256, 3)	0
xception (Functional)	(None, 2048)	20861480
dropout (Dropout)	(None, 2048)	0
flatten (Flatten)	(None, 2048)	0
fc1 (Dense)	(None, 256)	524544
fc2 (Dense)	(None, 256)	65792
predictions (Dense)	(None, 8)	2056

Total params: 21,453,872  
Trainable params: 7,380,776  
Non-trainable params: 14,073,096

FIGURE 4.10 – Résumé du modèle après fine-tuning : 6 090 760 paramètres à entraîner

Pour le modèle Xception, il a été fait le choix de ne dégeler que les 17 dernières couches du modèle applicatif, correspondant au dernier bloc de convolution. Voici le résultat des métriques au fil de l'entraînement, pour chaque epoch :

```

Epoch 1/10
342/342 [=====] - 127s 332ms/step - loss: 0.6230 - accuracy: 0.7842 - val_loss: 0.4776 - val_accuracy: 0.8278
Epoch 2/10
342/342 [=====] - 116s 340ms/step - loss: 0.3948 - accuracy: 0.8604 - val_loss: 0.3100 - val_accuracy: 0.8823
Epoch 3/10
342/342 [=====] - 116s 340ms/step - loss: 0.3628 - accuracy: 0.8744 - val_loss: 0.2627 - val_accuracy: 0.9020
Epoch 4/10
342/342 [=====] - 116s 340ms/step - loss: 0.3282 - accuracy: 0.8881 - val_loss: 0.2788 - val_accuracy: 0.9046
Epoch 5/10
342/342 [=====] - 116s 339ms/step - loss: 0.3214 - accuracy: 0.8897 - val_loss: 0.2392 - val_accuracy: 0.9115
Epoch 6/10
342/342 [=====] - 116s 339ms/step - loss: 0.3183 - accuracy: 0.8907 - val_loss: 0.3018 - val_accuracy: 0.8976
Epoch 7/10
342/342 [=====] - 117s 341ms/step - loss: 0.3054 - accuracy: 0.8924 - val_loss: 0.2731 - val_accuracy: 0.9016
Epoch 8/10
342/342 [=====] - 116s 339ms/step - loss: 0.2892 - accuracy: 0.8966 - val_loss: 0.2336 - val_accuracy: 0.9185
Epoch 9/10
342/342 [=====] - 116s 339ms/step - loss: 0.3007 - accuracy: 0.8963 - val_loss: 0.2878 - val_accuracy: 0.8969
Epoch 10/10
342/342 [=====] - 116s 339ms/step - loss: 0.2738 - accuracy: 0.9053 - val_loss: 0.2238 - val_accuracy: 0.9203
107/107 [=====] - 21s 194ms/step - loss: 0.2207 - accuracy: 0.9278
Loss function: 0.2207
Accuracy: 0.9278

```

FIGURE 4.11 – Entraînement des epochs 1 - 10 sans fine-tuning

```

Epoch 11/20
342/342 [=====] - 136s 381ms/step - loss: 0.1915 - accuracy: 0.9318 - val_loss: 0.1682 - val_accuracy: 0.9393
Epoch 12/20
342/342 [=====] - 128s 375ms/step - loss: 0.1709 - accuracy: 0.9417 - val_loss: 0.1556 - val_accuracy: 0.9426
Epoch 13/20
342/342 [=====] - 128s 375ms/step - loss: 0.1369 - accuracy: 0.9517 - val_loss: 0.1339 - val_accuracy: 0.9521
Epoch 14/20
342/342 [=====] - 130s 379ms/step - loss: 0.1207 - accuracy: 0.9579 - val_loss: 0.1261 - val_accuracy: 0.9547
Epoch 15/20
342/342 [=====] - 129s 377ms/step - loss: 0.1227 - accuracy: 0.9572 - val_loss: 0.1185 - val_accuracy: 0.9558
Epoch 16/20
342/342 [=====] - 134s 392ms/step - loss: 0.1038 - accuracy: 0.9637 - val_loss: 0.1200 - val_accuracy: 0.9532
Epoch 17/20
342/342 [=====] - 129s 376ms/step - loss: 0.0938 - accuracy: 0.9647 - val_loss: 0.1173 - val_accuracy: 0.9572
Epoch 18/20
342/342 [=====] - 129s 377ms/step - loss: 0.0958 - accuracy: 0.9661 - val_loss: 0.1137 - val_accuracy: 0.9594
Epoch 19/20
342/342 [=====] - 129s 377ms/step - loss: 0.0890 - accuracy: 0.9678 - val_loss: 0.1087 - val_accuracy: 0.9601
Epoch 20/20
342/342 [=====] - 128s 375ms/step - loss: 0.0906 - accuracy: 0.9682 - val_loss: 0.1110 - val_accuracy: 0.9605
107/107 [=====] - 20s 183ms/step - loss: 0.0988 - accuracy: 0.9681
Loss function: 0.0988
Accuracy: 0.9681

```

FIGURE 4.12 – Entraînement des epochs 11 - 20 avec fine-tuning

#### 4.3.5 Implémentation du modèle ResNet50V2

Le dernier modèle qui a été implémenté était ResNet50V2. En effet, une "V2" à été implémentée sur le module Keras, plus performant que le modèle initial. Nous profitons donc de cette amélioration dans le cadre de la création de notre modèle. Le procédé de création du modèle final ne change toujours pas.

Model: "WBC-ResNet50V2"		
Layer (type)	Output Shape	Param #
input (InputLayer)	[None, 256, 256, 3]	0
random_flip (RandomFlip)	(None, 256, 256, 3)	0
random_rotation (RandomRotation)	(None, 256, 256, 3)	0
preprocess (Lambda)	(None, 256, 256, 3)	0
resnet50v2 (Functional)	(None, 2048)	23564800
dropout (Dropout)	(None, 2048)	0
flatten (Flatten)	(None, 2048)	0
fc1 (Dense)	(None, 256)	524544
fc2 (Dense)	(None, 256)	65792
predictions (Dense)	(None, 8)	2056

Total params: 24,157,192  
Trainable params: 592,392  
Non-trainable params: 23,564,800

FIGURE 4.13 – Résumé du modèle avant fine-tuning : 592 392 paramètres à entraîner

Model: "WBC-ResNet50V2"		
Layer (type)	Output Shape	Param #
input (InputLayer)	[None, 256, 256, 3]	0
random_flip (RandomFlip)	(None, 256, 256, 3)	0
random_rotation (RandomRotation)	(None, 256, 256, 3)	0
preprocess (Lambda)	(None, 256, 256, 3)	0
resnet50v2 (Functional)	(None, 2048)	23564800
dropout (Dropout)	(None, 2048)	0
flatten (Flatten)	(None, 2048)	0
fc1 (Dense)	(None, 256)	524544
fc2 (Dense)	(None, 256)	65792
predictions (Dense)	(None, 8)	2056

Total params: 24,157,192  
Trainable params: 15,563,272  
Non-trainable params: 8,593,920

FIGURE 4.14 – Résumé du modèle après fine-tuning : 6 090 760 paramètres à entraîner

Pour le modèle ResNet50V2, il a été fait le choix de ne dégeler que les 37 dernières couches du modèle applicatif, correspondant au dernier bloc de convolution. Voici le résultat des métriques au fil de l’entraînement, pour chaque epoch :

```

Epoch 1/10
342/342 [=====] - 98s 271ms/step - loss: 0.5290 - accuracy: 0.8188 - val_loss: 0.2703 - val_accuracy: 0.9053
Epoch 2/10
342/342 [=====] - 89s 261ms/step - loss: 0.3145 - accuracy: 0.8942 - val_loss: 0.2764 - val_accuracy: 0.9024
Epoch 3/10
342/342 [=====] - 89s 261ms/step - loss: 0.2701 - accuracy: 0.9072 - val_loss: 0.2025 - val_accuracy: 0.9283
Epoch 4/10
342/342 [=====] - 89s 260ms/step - loss: 0.2510 - accuracy: 0.9120 - val_loss: 0.1716 - val_accuracy: 0.9389
Epoch 5/10
342/342 [=====] - 89s 259ms/step - loss: 0.2380 - accuracy: 0.9180 - val_loss: 0.1979 - val_accuracy: 0.9316
Epoch 6/10
342/342 [=====] - 89s 260ms/step - loss: 0.2258 - accuracy: 0.9232 - val_loss: 0.1764 - val_accuracy: 0.9367
Epoch 7/10
342/342 [=====] - 89s 259ms/step - loss: 0.2215 - accuracy: 0.9246 - val_loss: 0.1706 - val_accuracy: 0.9378
Epoch 8/10
342/342 [=====] - 89s 259ms/step - loss: 0.2074 - accuracy: 0.9274 - val_loss: 0.2567 - val_accuracy: 0.9119
Epoch 9/10
342/342 [=====] - 89s 259ms/step - loss: 0.2002 - accuracy: 0.9300 - val_loss: 0.1696 - val_accuracy: 0.9411
Epoch 10/10
342/342 [=====] - 90s 262ms/step - loss: 0.2015 - accuracy: 0.9310 - val_loss: 0.1687 - val_accuracy: 0.9433
187/187 [=====] - 15s 139ms/step - loss: 0.1749 - accuracy: 0.9371
Loss function: 0.1749
Accuracy: 0.9371

```

FIGURE 4.15 – Entraînement des epochs 1 - 10 sans fine-tuning

```
Epoch 11/20
342/342 [=====] - 113s 315ms/step - loss: 0.1302 - accuracy: 0.9539 - val_loss: 0.0906 - val_accuracy: 0.9711
Epoch 12/20
342/342 [=====] - 105s 305ms/step - loss: 0.0912 - accuracy: 0.9683 - val_loss: 0.0813 - val_accuracy: 0.9737
Epoch 13/20
342/342 [=====] - 105s 307ms/step - loss: 0.0776 - accuracy: 0.9731 - val_loss: 0.0731 - val_accuracy: 0.9784
Epoch 14/20
342/342 [=====] - 105s 307ms/step - loss: 0.0682 - accuracy: 0.9777 - val_loss: 0.0792 - val_accuracy: 0.9740
Epoch 15/20
342/342 [=====] - 105s 306ms/step - loss: 0.0609 - accuracy: 0.9800 - val_loss: 0.0664 - val_accuracy: 0.9777
Epoch 16/20
342/342 [=====] - 104s 305ms/step - loss: 0.0483 - accuracy: 0.9821 - val_loss: 0.0662 - val_accuracy: 0.9751
Epoch 17/20
342/342 [=====] - 104s 305ms/step - loss: 0.0474 - accuracy: 0.9850 - val_loss: 0.0602 - val_accuracy: 0.9788
Epoch 18/20
342/342 [=====] - 105s 307ms/step - loss: 0.0446 - accuracy: 0.9831 - val_loss: 0.0672 - val_accuracy: 0.9766
Epoch 19/20
342/342 [=====] - 105s 308ms/step - loss: 0.0401 - accuracy: 0.9866 - val_loss: 0.0583 - val_accuracy: 0.9821
Epoch 20/20
342/342 [=====] - 106s 308ms/step - loss: 0.0371 - accuracy: 0.9871 - val_loss: 0.0574 - val_accuracy: 0.9795
107/107 [=====] - 14s 130ms/step - loss: 0.0597 - accuracy: 0.9792
Loss function: 0.0597
Accuracy: 0.9792
```

FIGURE 4.16 – Entraînement des epochs 11 - 20 avec fine-tuning

# Chapitre 5

## Résultats et interprétations

Les métriques n'ont aucun impact sur l'entraînement des modèles. Elles servent à évaluer la performance des modèles que l'on sélectionne pour l'apprentissage.

### 5.1 Choix des métriques et des visuels adapté à notre problématique

La métrique choisie dépend du problème exposé. Ici, nous avons fait le choix de ne pas nous limiter à une seule, mais deux métriques :

- l'accuracy : afin d'avoir une idée globale de la performance de notre modèle
- le F1-Score : afin d'obtenir un retour de performance sur chacune des classes

Afin d'exploiter ces métriques et améliorer notre lecture de performances, nous avons également fait appel à des visuels très pratiques pour résumer l'évaluation d'un modèle :

- un rapport de classification : qui résume les métriques basiques sur l'ensemble et sur chaque classe
- une courbe représentative de l'évolution de l'accuracy sur l'ensemble d'entraînement et sur l'ensemble de test
- une courbe représentative de l'évolution de la fonction de perte sur les mêmes ensembles
- une matrice de confusion : qui répertorie les bonnes et mauvaises prédictions pour chaque classe

Combinés, ces outils d'analyse sont pertinents pour mesurer la qualité de nos modèles.

### 5.2 Visualisation des différents résultats et prédictions

#### 5.2.1 Le rapport de classification

	precision	recall	f1-score	support		precision	recall	f1-score	support
BASOPHIL	0.98	0.98	0.98	244	BASOPHIL	0.98	0.97	0.98	244
EOSINOPHIL	1.00	0.99	1.00	623	EOSINOPHIL	0.99	0.99	0.99	623
ERYTHROBLAST	0.97	1.00	0.98	310	ERYTHROBLAST	0.96	0.99	0.97	310
IG	0.95	0.96	0.95	579	IG	0.92	0.95	0.93	579
LYMPHOCYTE	0.99	1.00	0.99	243	LYMPHOCYTE	0.98	0.99	0.98	243
MONOCYTE	0.96	0.96	0.96	284	MONOCYTE	0.96	0.91	0.94	284
NEUTROPHIL	0.99	0.97	0.98	666	NEUTROPHIL	0.98	0.96	0.97	666
PLATELET	1.00	1.00	1.00	470	PLATELET	1.00	1.00	1.00	470
accuracy			0.98	3419	accuracy			0.97	3419
macro avg	0.98	0.98	0.98	3419	macro avg	0.97	0.97	0.97	3419
weighted avg	0.98	0.98	0.98	3419	weighted avg	0.97	0.97	0.97	3419

FIGURE 5.1 – VGG16

FIGURE 5.2 – MobileNet

	precision	recall	f1-score	support		precision	recall	f1-score	support
BASOPHIL	0.99	0.95	0.97	244	BASOPHIL	0.96	0.99	0.97	244
EOSINOPHIL	1.00	0.99	0.99	623	EOSINOPHIL	0.99	1.00	1.00	623
ERYTHROBLAST	0.98	0.97	0.98	310	ERYTHROBLAST	1.00	0.96	0.98	310
IG	0.90	0.95	0.93	579	IG	0.97	0.95	0.96	579
LYMPHOCYTE	0.98	0.96	0.97	243	LYMPHOCYTE	0.96	0.99	0.98	243
MONOCYTE	0.96	0.92	0.94	284	MONOCYTE	0.97	0.96	0.96	284
NEUTROPHIL	0.96	0.98	0.97	666	NEUTROPHIL	0.98	0.98	0.98	666
PLATELET	1.00	0.99	0.99	470	PLATELET	0.99	1.00	0.99	470
accuracy			0.97	3419	accuracy			0.98	3419
macro avg	0.97	0.96	0.97	3419	macro avg	0.98	0.98	0.98	3419
weighted avg	0.97	0.97	0.97	3419	weighted avg	0.98	0.98	0.98	3419

FIGURE 5.3 – Xception

FIGURE 5.4 – ResNet50

### 5.2.2 Synthèse des résultats sur les 4 modèles

Afin de faciliter la lecture directe des résultats, nous avons récapitulé dans un tableau le F1-score par classe et l'accuracy, pour chaque modèle. Il est important de préciser que l'accuracy est extraite du résultat du code `model.evaluate()` sur le jeu de test, et le F1-score est directement tiré des rapports de classification sur ce même jeu.

		VGG16	MobileNetV2	Xception	ResNet50V2
Accuracy		0.9804	0.9710	0.9681	0.9792
F1-score	BASOPHIL	0.98	0.98	0.97	0.97
	EOSINOPHIL	1.00	0.99	0.99	1.00
	ERYTHROBLAST	0.98	0.97	0.98	0.98
	IG	0.95	0.93	0.93	0.96
	LYMPHOCYTE	0.99	0.98	0.97	0.98
	MONOCYTE	0.96	0.94	0.94	0.96
	NEUTROPHIL	0.98	0.97	0.97	0.98
	PLATELET	1.00	1.00	0.99	0.99

FIGURE 5.5 – F1-score pour chaque classe et accuracy pour chaque modèle

Observations :

- les modèles sont plutôt performants (0.968 - 0.980% d'accuracy). En revanche, la problématique étant d'ordre médical, il est plus qu'important de minimiser l'erreur de prédiction, et le moindre pourcentage d'accuracy compte. C'est pourquoi le modèle VGG16 est considéré comme le plus efficace.
- il existe visiblement des disparités entre l'identification des classes. Par exemple, les plaquettes et les eosinophiles semblent être identifiées presque parfaitement par les 4 modèles, alors que la classe IG est toujours "difficilement" reconnue.

### 5.2.3 Courbe de la fonction de perte et de l'accuracy

Dans cette partie nous avons tracé pour chaque modèle :

- en bleu : la valeur de fonction de perte et d'accuracy sur l'ensemble d'entraînement
- en orange : la valeur de fonction de perte et d'accuracy sur l'ensemble de test
- en vert : le seuil d'epoch à partir duquel le fine-tuning est introduit

Training model: WBC-VGG16

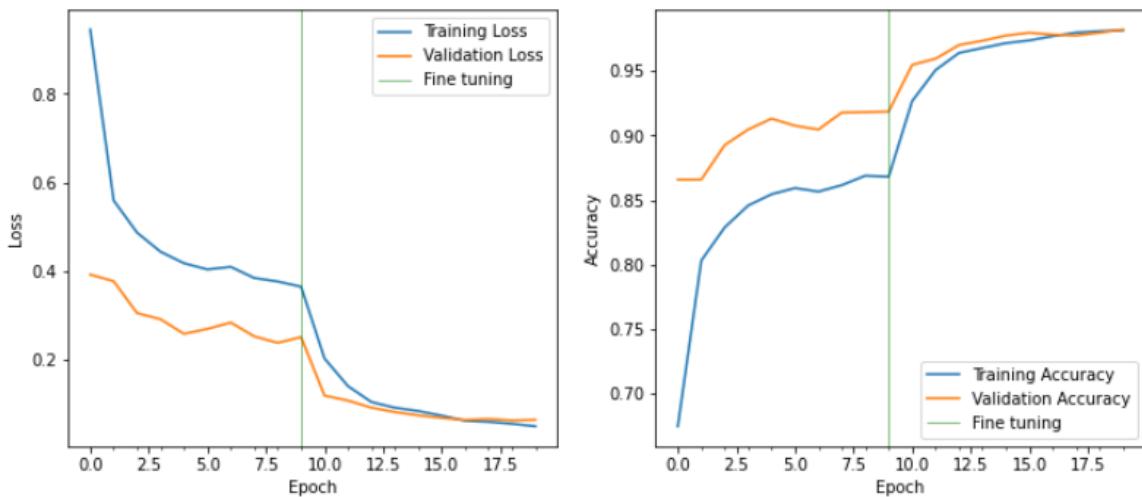


FIGURE 5.6 – Courbes pour VGG16

Training model: WBC-MobileNetV2

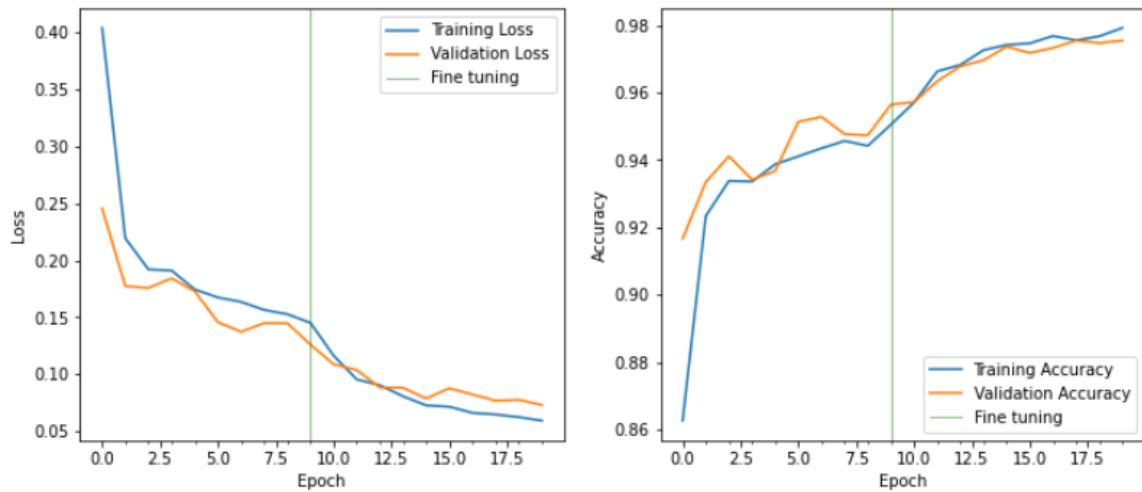


FIGURE 5.7 – Courbes pour MobileNet

Training model: WBC-Xception

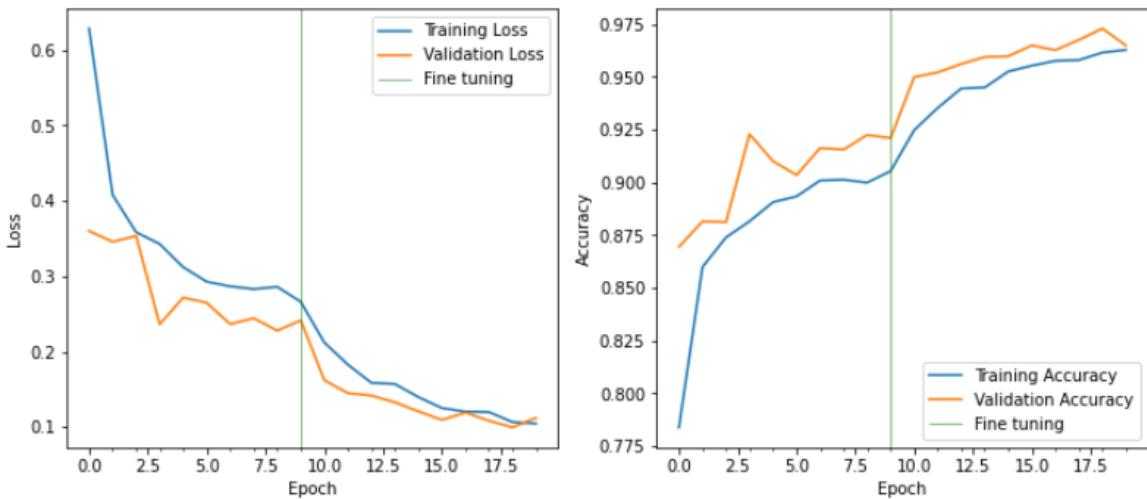


FIGURE 5.8 – Courbes pour Xception

Training model: WBC-ResNet50V2

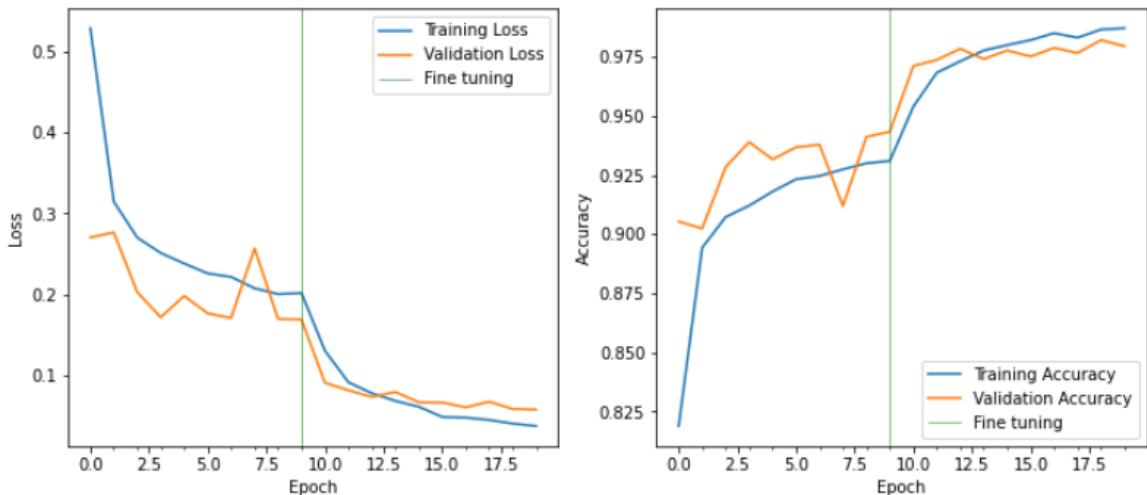


FIGURE 5.9 – Courbes pour ResNet50

Observations :

1. Le fine-tuning joue un rôle considérable dans la **réduction du sur-apprentissage**, puisque deux courbes se rapprochent l'une de l'autre après la ligne verte indiquant le début du fine-tuning. Cela vaut pour chaque modèle utilisé, cependant, le changement des courbes d'apprentissage du modèle VGG16 est particulièrement visible.
2. Le fine-tuning **améliore des performances du modèles**. Les quatre modèles ont atteint le plateau en précision et en perte avant le fine-tuning (avant la ligne verte). Cependant, au début du fine-tuning, nous voyons un saut brusque vers 0 en cas de perte et vers 1 en cas de précision suivi d'une diminution régulière en cas de perte et d'une augmentation régulière en cas de précision. Encore une fois, c'est particulièrement clair dans le cas du modèle VGG16.
3. Les quatre modèles deviennent plus stables (moins de pics dans les courbes pour le jeu de données de test) après fine-tuning.

### 5.2.4 Heatmap des matrices de confusion

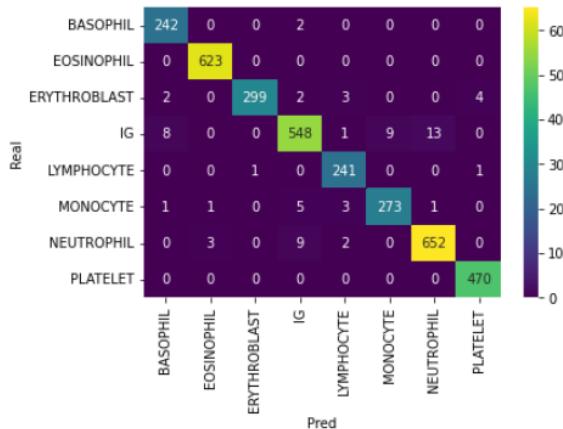


FIGURE 5.10 – VGG16

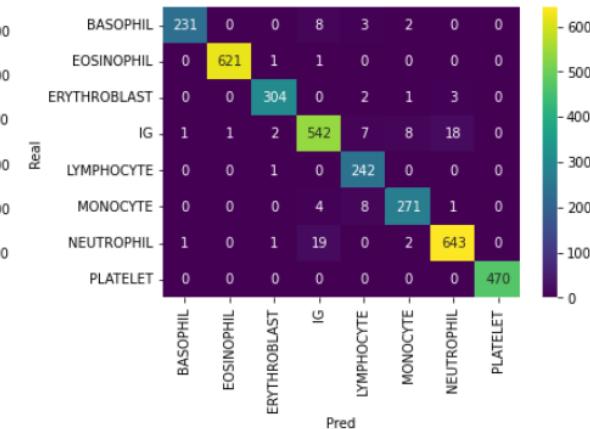


FIGURE 5.11 – MobileNetV2

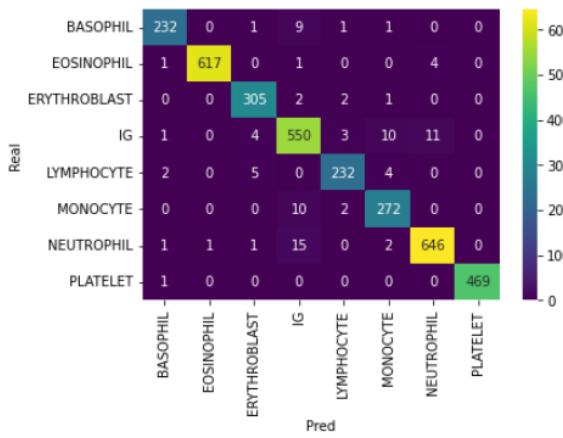


FIGURE 5.12 – Xception

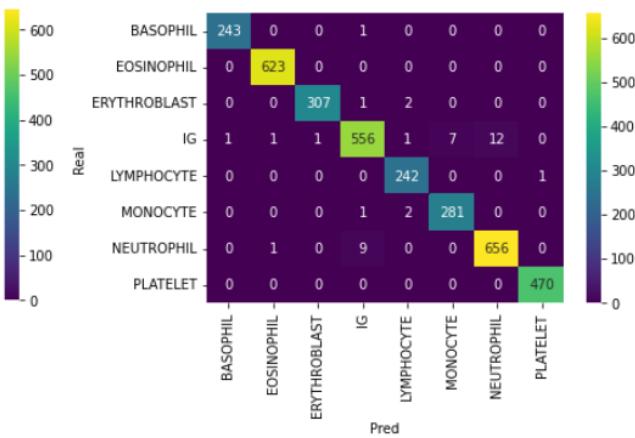


FIGURE 5.13 – ResNet50

Cette matrice de confusion confirme les bons résultats des modèles. On voit clairement que dans l'ensemble, les prédictions sont bonnes. On voit également, les classe qui "pose problème" sont la classe IG et neutrophil, comme nous avions pu le constater sur les rapports de classification.

Ainsi, même si les résultats sont assez similaires, le modèle le plus adapté pour résoudre notre problématique semble être le modèle VGG16.

Étudions de plus près les prédictions concrètes, afin d'en apprendre un peu plus sur la méthode de notre modèle.

### 5.2.5 Prédictions à partir des modèles

#### Observation des prédictions sous forme de dataframe

Les prédictions sont affichées sous la forme d'un tableau avec les score de certitude de chaque classe ainsi que la classe prédite. Nous l'avons fait avec ce code :

```

1 # creation des predictions
2 preds = model.predict(test_ds)
3
4 # creation du dataframe de predictions
5 res = pd.DataFrame(preds, columns=cats, index=X_test.index)
6 res['Pred'] = res.idxmax(axis=1)

```

Ce qui nous donne le dataframe suivant :

	EOSINOPHIL	PLATELET	LYMPHOCYTE	ERYTHROBLAST	IG	MONOCYTE	BASOPHIL	NEUTROPHIL	Pred	Score	Real
4180	0.0	1.0	0.000000	0.000000	0.000000	0.0	0.000000	0.000000	PLATELET	1.000000	PLATELET
8060	0.0	0.0	0.000000	1.000000	0.000000	0.0	0.000000	0.000000	ERYTHROBLAST	1.000000	ERYTHROBLAST
3767	0.0	1.0	0.000000	0.000000	0.000000	0.0	0.000000	0.000000	PLATELET	1.000000	PLATELET
621	1.0	0.0	0.000000	0.000000	0.000000	0.0	0.000000	0.000000	EOSINOPHIL	1.000000	EOSINOPHIL
6322	0.0	0.0	0.999104	0.000892	0.000003	0.0	0.000000	0.000000	LYMPHOCYTE	0.999104	LYMPHOCYTE
...	...	...	...	...	...	...	...	...	...	...	...
13213	0.0	0.0	0.000000	0.000000	0.000001	0.0	0.999999	0.000000	BASOPHIL	0.999999	BASOPHIL
12921	0.0	0.0	0.000000	0.000000	0.000079	0.0	0.999913	0.000008	BASOPHIL	0.999913	BASOPHIL
6457	0.0	0.0	0.999869	0.000056	0.000003	0.0	0.000071	0.000000	LYMPHOCYTE	0.999869	LYMPHOCYTE
10546	0.0	0.0	0.000000	0.000000	0.999997	0.0	0.000003	0.000000	IG	0.999997	IG
14267	0.0	0.0	0.000000	0.000000	0.000024	0.0	0.000000	0.999975	NEUTROPHIL	0.999975	NEUTROPHIL

FIGURE 5.14 – Les prédictions faites par le modèle VGG16

Nous pouvons également observer les images mal prédites en ajoutant deux colonnes : "Real" pour la vraie classe de l'image et "Score" pour le pourcentage de certitude du modèle concernant sa prédiction. Cela nous donne le dataframe suivant :

	EOSINOPHIL	PLATELET	LYMPHOCYTE	ERYTHROBLAST	IG	MONOCYTE	BASOPHIL	NEUTROPHIL	Pred	Score	Real
16477	0.000133	0.000000	0.000001	0.000054	0.544337	0.000015	0.000000	0.455460	IG	0.544337	NEUTROPHIL
12784	0.000001	0.000014	0.000311	0.000139	0.687245	0.000408	0.311793	0.000090	IG	0.687245	BASOPHIL
188	0.029451	0.000037	0.014020	0.006895	0.063273	0.041149	0.843972	0.000903	BASOPHIL	0.843972	EOSINOPHIL
16064	0.003162	0.000399	0.000109	0.000346	0.900393	0.005067	0.006821	0.083701	IG	0.900393	NEUTROPHIL
9946	0.000001	0.000003	0.000003	0.000006	0.296867	0.703062	0.000009	0.000048	MONOCYTE	0.703062	IG
...	...	...	...	...	...	...	...	...	...	...	...
8237	0.000062	0.000000	0.000000	0.000003	0.210913	0.000007	0.000001	0.789015	NEUTROPHIL	0.789015	IG
9676	0.000000	0.000000	0.000000	0.000000	0.296062	0.000000	0.000000	0.703937	NEUTROPHIL	0.703937	IG
9472	0.000000	0.000000	0.000557	0.998877	0.000537	0.000001	0.000026	0.000001	ERYTHROBLAST	0.998877	IG
10683	0.000000	0.000000	0.000000	0.000001	0.045154	0.000000	0.000000	0.954846	NEUTROPHIL	0.954846	IG
9915	0.000000	0.000001	0.000010	0.000017	0.286215	0.000525	0.713230	0.000002	BASOPHIL	0.713230	IG

FIGURE 5.15 – Les prédictions qui ont échoué par le modèle VGG16

### Observation des prédictions sous forme d'images

Par souci de visibilité, nous pouvons afficher aléatoirement 8 images qui ont été correctement prédites :

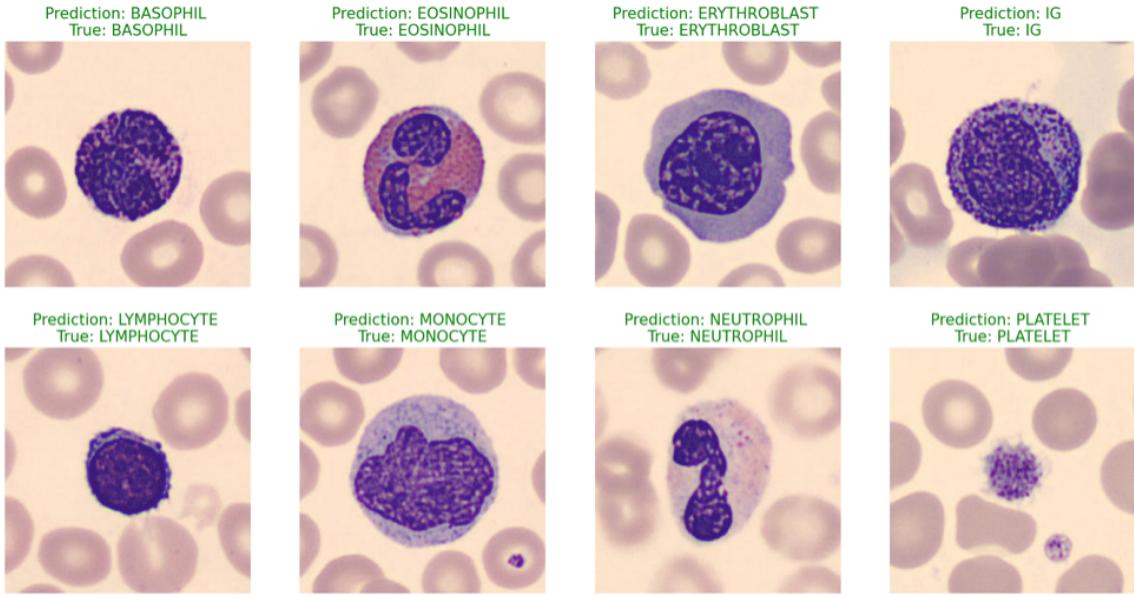


FIGURE 5.16 – Exemples de différentes images des cellules avec les prédictions et leurs véritables étiquettes

Et également 8 images qui elles, ont été mal prédites :

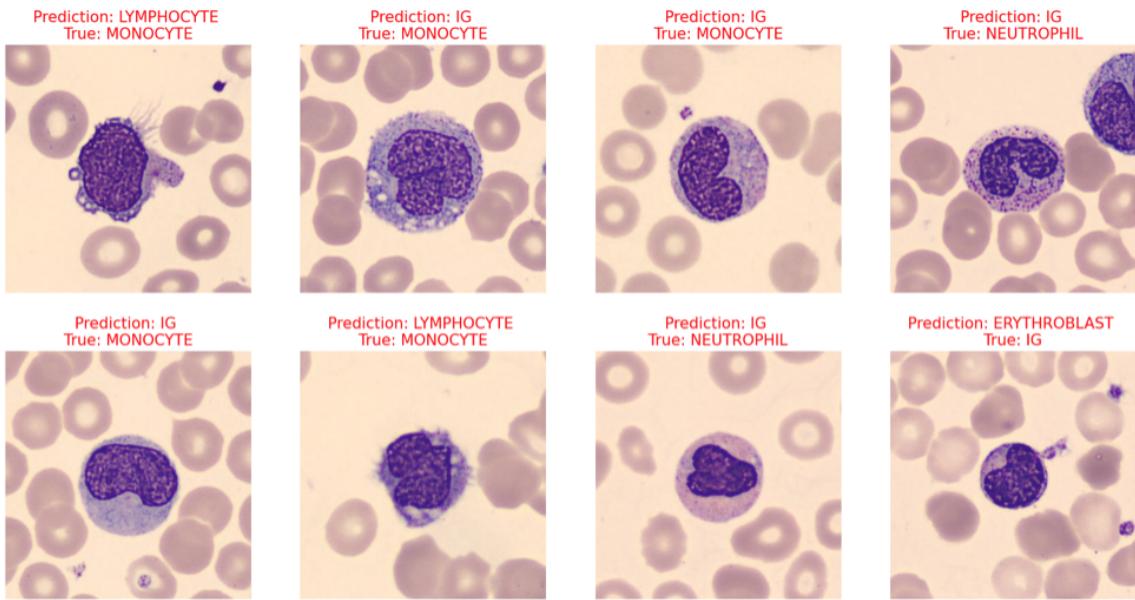


FIGURE 5.17 – Exemples de différentes images des cellules mal classées avec les prédictions et leurs véritables étiquettes

### Analyses statistiques des mauvaises prédictions

Finalement, pour mieux comprendre comment le modèle se trompe dans ses prédictions, nous avons étudié statistiquement ces erreurs. Nous avons tracé deux graphiques :

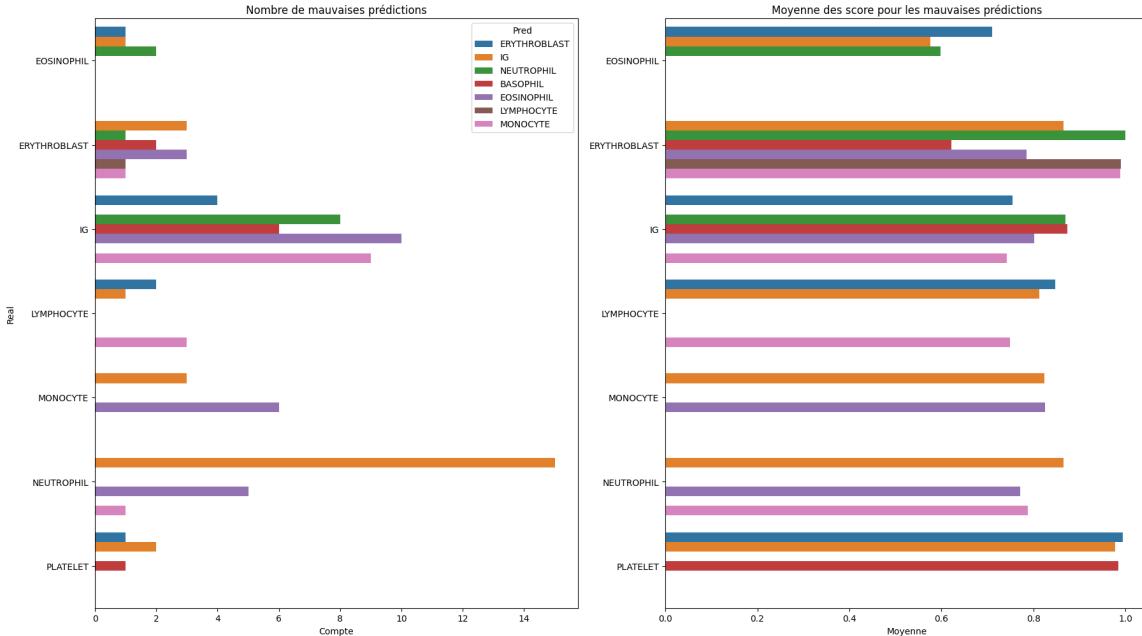


FIGURE 5.18 – Analyse des comptes et moyennes des mauvaises prédictions

Ainsi, on peut voir à gauche le nombre de fois où le modèle a confondu la classe prédite (couleur) et la classe réelle (axe y). À droite, on voit la moyenne des scores de certitude pour chaque situation d'erreur.

On remarque deux points intéressants :

1. Parmi les erreurs les plus fréquentes, que ce soit pour la classe prédite ou pour la classe réelle, IG est la classe la plus confondue.
2. Dans le cas d'une erreur, le modèle est souvent très sûr de lui. En effet, la certitude moyenne du modèle sur ses erreurs de prédiction varie entre 58% et plus de 99%.

## 5.3 Grad-CAM

Afin de mieux interpréter les résultats de classification obtenus par notre modèle, il serait intéressant de voir sur quelles parties de l'image le modèle se concentre. Pour cela, nous avons implémenté l'algorithme du Grad-CAM [2] dont le principe consiste à analyser la sortie de la dernière couche de convolution d'un modèle puis appliquer une dérivation automatique. Ainsi, il est possible de calculer le gradient de la classe prédite ( $y$ ) relatif au vecteur de sortie de la dernière couche de convolution ( $x$ ).

### 5.3.1 Construction des modèles

Dans toutes les implémentations trouvées, le modèle applicatif est utilisé directement pour faire l'analyse, ce qui simplifie le traitement. De plus l'analyse est faite sur une seule image et pas sur un lot d'images.

Cependant, dans le cas présent, le modèle applicatif est imbriqué dans notre modèle global. En amont, nous trouvons les couches de pré-traitement des images et en aval, les couches denses ainsi que la couche de prédiction. Il a donc été nécessaire de récupérer l'ensemble des couches des deux modèles imbriqués pour construire deux nouveaux modèles indépendants.

A l'issue de cette étape, nous disposons de :

- un modèle de convolution (Tableau 5.1) qui prend en entrée un lot d'images et en sortie les valeurs de la dernière couche de convolution,
- un modèle de classification (Tableau 5.2) qui prend en entrée la sortie de la dernière couche de convolution et en sortie le vecteur de probabilités de la couche de classification.

Model : WBC-VGG16-conv

Layer (type)	Output Shape	Param #
preprocess (Lambda)	(None, 256, 256, 3)	0
block1_conv1 (Conv2D)	(None, 256, 256, 64)	1792
block1_conv2 (Conv2D)	(None, 256, 256, 64)	36928
block1_pool (MaxPooling2D)	(None, 128, 128, 64)	0
block2_conv1 (Conv2D)	(None, 128, 128, 128)	73856
block2_conv2 (Conv2D)	(None, 128, 128, 128)	147584
block2_pool (MaxPooling2D)	(None, 64, 64, 128)	0
block3_conv1 (Conv2D)	(None, 64, 64, 256)	295168
block3_conv2 (Conv2D)	(None, 64, 64, 256)	590080
block3_conv3 (Conv2D)	(None, 64, 64, 256)	590080
block3_pool (MaxPooling2D)	(None, 32, 32, 256)	0
block4_conv1 (Conv2D)	(None, 32, 32, 512)	1180160
block4_conv2 (Conv2D)	(None, 32, 32, 512)	2359808
block4_conv3 (Conv2D)	(None, 32, 32, 512)	2359808
block4_pool (MaxPooling2D)	(None, 16, 16, 512)	0
block5_conv1 (Conv2D)	(None, 16, 16, 512)	2359808
block5_conv2 (Conv2D)	(None, 16, 16, 512)	2359808
block5_conv3 (Conv2D)	(None, 16, 16, 512)	2359808
Total params : 14,714,688		
Trainable params : 2,359,808		
Non-trainable params : 12,354,880		

TABLE 5.1 – Grad-CAM : modèle de convolution

Model : WBC-VGG16-clf

Layer (type)	Output Shape	Param #
block5_pool (MaxPooling2D)	(None, 8, 8, 512)	0
gbl_avg_pool2d (GlobalAveragePooling2D)	(None, 512)	0
dropout (Dropout)	(None, 512)	0
flatten (Flatten)	(None, 512)	0
fc1 (Dense)	(None, 256)	131328
fc2 (Dense)	(None, 256)	65792
predictions (Dense)	(None, 8)	2056
Total params : 199,176		
Trainable params : 199,176		
Non-trainable params : 0		

TABLE 5.2 – Grad-CAM : modèle de classification

### 5.3.2 Génération de la heatmap

À l'entrée du réseau de convolution, l'image a une dimension de (256, 256, 3) (hauteur, largeur, canaux de couleur). Tout au long de son parcours dans les différentes couches, l'image va subir des transformations et changer de dimension. Pour VGG16, la dernière couche de convolution a une dimension de (16, 16, 512) (hauteur, largeur, nombre de filtres). Cela signifie que la dimension spatiale de l'image a été divisée par 16 ( $256 \div 16 \rightarrow 16$ ) et les canaux de couleur ont été remplacés par des filtres. Autrement dit, nous avons en sortie du modèle de convolution, 512 images de 16x16 pixels.

C'est ce dernier vecteur qui nous intéresse dans le Grad-CAM. Après la dérivation automatique et la descente de gradient, nous réduisons la dimension de notre vecteur par la suppression de sa spatialité : pour chaque petite image de 16x16, nous ne conservons qu'une valeur moyenne, c'est l'équivalent du **GlobalAveragePooling2D**. Ainsi, à la fin de cette étape, nous disposons d'un vecteur de 512 valeurs correspondantes aux 512 filtres.

Pour obtenir la heatmap, il ne nous reste plus qu'à appliquer le produit matriciel entre les valeurs de sortie de la dernière couche de convolution (16, 16, 512) et la moyenne de nos gradients (512,). La

dernière étape de génération de la heatmap consiste à normaliser les valeurs dans l'intervalle  $[0, 1]$  puis entre  $[0, 255]$ . Il ne reste plus qu'à remettre à la heatmap à l'échelle de l'image et à les fusionner en jouant sur le niveau de transparence comme le montre la figure 5.19.

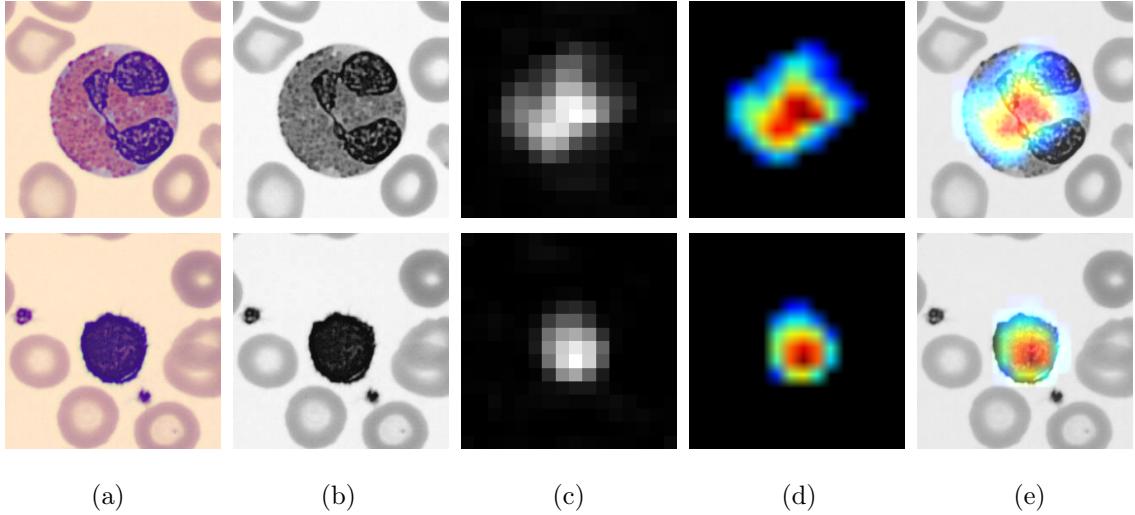


FIGURE 5.19 – Grad-CAM : visualisation des zones d'attention. (a) image originale, (b) image en niveau de gris, (c) niveaux du gradient, (d) colorisation, (e) surimposition de la heatmap.

Sur la figure 5.20, nous pouvons voir un exemple de heatmap pour un échantillon de chaque classe. Cette visualisation nous permet de juger de la pertinence de la segmentation de la cellule effectuée en amont par des procédés de vision par ordinateur. Sur le jeu de données original, nous voyons que le modèle va chercher de lui-même l'information utile dans la cellule, ce qui donne un sens à l'hypothèse de départ. En effet, nous voulions éliminer le bruit autour de la cellule à classifier pour que le modèle ne se concentre que sur cette dernière sans perturbation extérieure. Au final, le modèle CNN n'a pas besoin de ce pré-traitement pour faire une bonne classification : il n'est que peu perturbé par le fond et les globules rouges environnants.

Dans ce contexte, il donc plus intéressant d'avoir un jeu de données brut avec beaucoup d'échantillons ( $\simeq 17000$ ) plutôt qu'un jeu de données filtré mais sans suffisamment d'échantillons ( $\simeq 1350$ ).

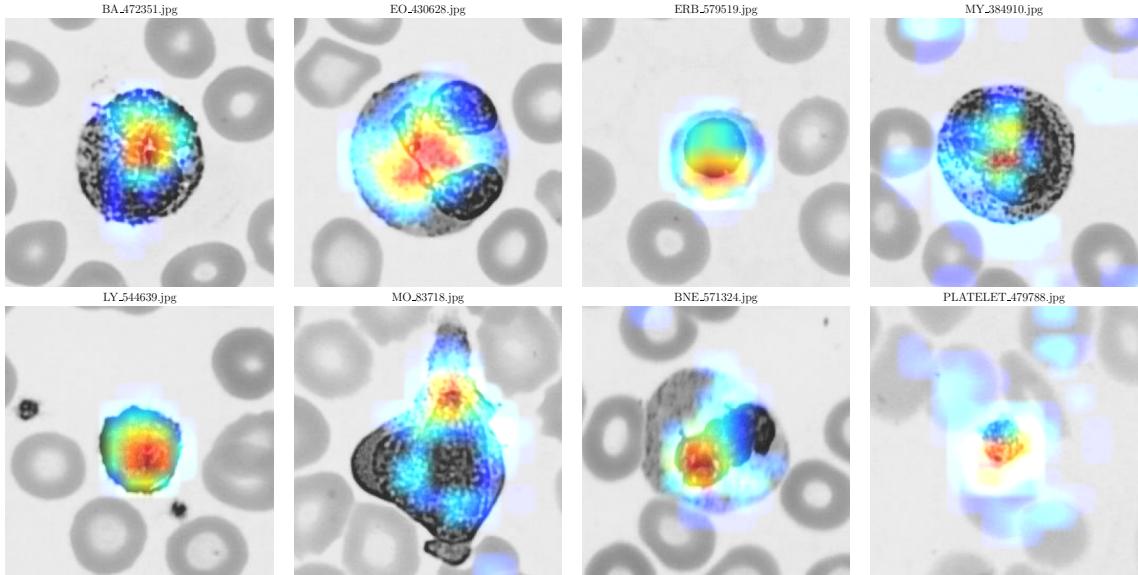


FIGURE 5.20 – Grad-CAM : exemple de heatmaps pour une cellule de chaque classe.

En ce qui concerne l'algorithme du Grad-CAM, il est certainement intéressant pour des images contenant plusieurs classes à prédire sur la même image mais dans le cadre de notre étude, cela

présente un intérêt modéré étant donné que les prédictions en amont sont déjà satisfaisantes. Néanmoins, l'implémentation est novatrice dans le sens où nous ne partons pas d'un modèle applicatif de base mais de modèles imbriqués. La contribution réside dans le traitement par lot des images et la décomposition en 2 modèles<sup>1</sup>.

---

1. La décomposition ne fonctionne qu'avec VGG16 pour le moment car les autres modèles applicatifs réinjectent leur vecteur d'entrée dans des couches plus hautes

# Chapitre 6

## Conclusion

Les 4 modèles implémentés dans notre rapport sont performants. Ils permettent de classifier les cellules sanguines en fonction de leurs caractéristiques morphologiques en utilisant des techniques d'apprentissage profond.

Pour obtenir ces résultats, nous avons dû effectuer plusieurs modifications dans le choix du jeu de données, puis sur les techniques de preprocessing de nos images, pour trouver la meilleure combinaison possible.

Au tout début du projet, nous avons constaté que les métriques étaient supérieurs en partant d'un jeu de données segmenté au lieu de partir sur notre jeu de données brut « Barcelone ».

Nous pensions effectivement que tout le travail de segmentation sur notre jeu de données était l'une des clés de performance des modèles d'apprentissage profond. Ce fut l'inverse dans notre cas. Nous avons vite constaté que les résultats étaient moins probants.

La segmentation fut une part importante de notre travail sur ce projet. Elle fut explorée en profondeur. Cependant, nous ne l'avons pas retenue dans notre processus de modélisation car nous n'avons pas pu consolidé la méthode dans le temps imparti.

Pour en arriver à ces performances concluantes, en fin de projet, nous avons finalement fait machine arrière pour repartir sur notre jeu de données brut « Barcelone », sans procéder à une segmentation par la suite. Les résultats furent nettement plus intéressants en testant 2 méthodes d'apprentissage qui sont très efficaces dans la classification d'image, à savoir, le Transfer Learning et le Fine-tuning.

En effet, le Transfer Learning nous a permis de partir de 4 modèles pré entraînés pour extraire des poids déjà définis et optimisés pour la reconnaissance d'images.

Quand au Fine-tuning, il optimise les poids de nos 4 modèles implémentés.

Grâce au Transfer Learning et au Fine-tuning, nous avons pu fiabiliser notre pipeline de modélisation d'un réseau de convolution. Ce dernier peut se généraliser sur d'autres modèles pré entraînés et peut prendre notamment en compte la variabilité des systèmes de coloration et d'acquisition de Romanowsky ainsi que des exigences importantes de sa conception en termes de pré-traitement et d'extraction de caractéristiques.

Concrètement, les résultats des métriques sont très satisfaisants. Les accuracy tournent entre 0,9681 et 0,9804. Quand aux F1-score par classe, ils vont de 0,93 jusqu'au maximum 1.

Un léger avantage est concédé au modèle VGG16, avec une accuracy de 98% et des F1-score pour chaque classe qui dépassent ceux des autres modèles, sauf pour la classe IG, qui semble mieux classée par le modèle ResNet50V2 (à 1% de différence).

Ainsi, si l'optimisation absolue repose sur le fait de maximiser tous les F1-score, alors le modèle ResNet50V2 serait le plus adapté. En revanche, si l'objectif est de maximiser l'accuracy, alors le VGG16 l'emporte.

Par ailleurs, notre problématique étant d'ordre médical, il faut attacher plus d'importance sur l'erreur de prédiction à minimiser.

Dans le cas des prédictions faites par le modèle VGG16, nous constatons que les erreurs les plus fréquentes restent la classe IG. Cette classe est la plus confondue, que ce soit pour la classe prédite ou pour la classe réelle. Dans le cas d'une erreur, le modèle est souvent très sûr de lui. En effet, la certitude moyenne du modèle sur ses erreurs de prédiction varie entre 58% et plus de 99%.

Malgré ces erreurs de prédiction, le modèle VGG16 est considéré comme le plus efficace, une fois de plus.

Le plus important dans l'interprétation de nos résultats, comme nous l'a montré l'entraînement des modèles, reste l'optimisation de leur création par transfer learning et par fine-tuning. C'est ces caractéristiques qui ont vraiment poussé les modèles à leur meilleur niveau, et leur a permis d'atteindre ces performances.

Nous pouvons donc souligner que notre approche se basant sur ces techniques d'apprentissage profond, est moins coûteuse et ne dépend pas de la segmentation. Le déploiement de cet apprentissage profond a donc toute son importance pour effectuer une évaluation visuelle et qualitative des frottis sanguins. L'identification des cellules sera moins fastidieuse. Les diagnostics pourront être réalisés plus rapidement, tout en diminuant le risque d'erreur humaine. Plus tôt seront réalisés ces diagnostics complets, plus les patients malades seront traités à un stade moins avancé, ce qui diminuera le taux de mortalité.

Après avoir présenté notre démarche de recherche des meilleurs modèles de prédiction possibles pour répondre aux objectifs de notre projet "yawbcc", il serait intéressant de s'interroger sur certains points à améliorer.

La contrainte majeure de ce projet était le déséquilibre initial du dataset, notamment le manque de données pour certaines classes de cellule. Il aurait été intéressant d'utiliser d'autres datasets afin d'avoir une représentation plus équilibrée des classes.

Par ailleurs, les images de l'entraînement étaient prises sur un background uniforme. Nous pouvons donc se poser la question sur la performance du modèle si le background change de couleur/forme, ou encore si la cellule n'est plus centrée sur l'image.

Dans une perspective à plus long terme, on pourrait étendre le scope du projet en exploitant une autre technique pour améliorer nos modèles en utilisant les Vision Transformers. Il s'agit d'une nouvelle gamme de réseau de neurone, ou plus précisément, une autre façon de traiter l'image en passant par cette technique ce modèle s'entraîne à se concentrer sur certaines zones de l'image.

Cette approche pourrait donner naissance à un projet commercialisable et serait d'une importance capitale pour le corps médical (les cabinets d'imagerie médicale, les médecins et les chercheurs...)

Une autre idée d'approche serait de partir sur une multi label classification (une classification non exclusive). Imaginons qu'au lieu d'entraîner un seul réseau de neurones qui prédit 1 des 8 classes, nous options pour 8 réseaux de neurones qui prédisent 1 classe (classification binaire). Chaque réseau serait entraîné sur ce qu'il est (un neutrophile par exemple) et ce qu'il n'est pas (un basophile, un éosinophile, etc). Pour la prédiction, chaque réseau reçoit une image à prédire et s'exprime sur la probabilité d'avoir reconnu sa classe.

Néanmoins, la classification actuelle dans notre projet d'implémentation n'est pas exclusive puisqu'elle donne une probabilité sur chaque classe. Sauf que nous retenons la classe dont la probabilité est la plus forte sans se soucier du score des autres. L'avantage de la "multi label classification" serait de permettre à chaque classe de s'exprimer librement sans la domination d'une autre classe qui aurait été sur-entraînée et qui dirait que toutes les images sont de sa famille.

Nous aurions au final 8 classifications binaire.

# Bibliographie

- [1] Roopa B. HEGDE et al. « Comparison of traditional image processing and deep learning approaches for classification of white blood cells in peripheral blood smear images ». In : *Biocybernetics and Biomedical Engineering* 39.2 (1<sup>er</sup> avr. 2019), p. 382-392. ISSN : 0208-5216. DOI : [10.1016/j.bbe.2019.01.005](https://doi.org/10.1016/j.bbe.2019.01.005). URL : <https://www.sciencedirect.com/science/article/pii/S0208521618304819> (visité le 04/10/2022).
- [2] Ramprasaath R. SELVARAJU et al. « Grad-CAM : Visual Explanations from Deep Networks via Gradient-based Localization ». In : *International Journal of Computer Vision* 128.2 (fév. 2020), p. 336-359. ISSN : 0920-5691, 1573-1405. DOI : [10.1007/s11263-019-01228-7](https://doi.org/10.1007/s11263-019-01228-7). arXiv : [1610.02391\[cs\]](https://arxiv.org/abs/1610.02391). URL : [http://arxiv.org/abs/1610.02391](https://arxiv.org/abs/1610.02391) (visité le 11/11/2022).
- [3] Sajad TAVAKOLI et al. « New segmentation and feature extraction algorithm for classification of white blood cells in peripheral smear images ». In : *Scientific Reports* 11.1 (30 sept. 2021). Number : 1 Publisher : Nature Publishing Group, p. 19428. ISSN : 2045-2322. DOI : [10.1038/s41598-021-98599-0](https://doi.org/10.1038/s41598-021-98599-0). URL : <https://www.nature.com/articles/s41598-021-98599-0> (visité le 14/11/2022).
- [4] *Watershed (image processing)*. In : *Wikipedia*. Page Version ID : 1109774665. 11 sept. 2022. URL : [https://en.wikipedia.org/w/index.php?title=Watershed\\_\(image\\_processing\)&oldid=1109774665](https://en.wikipedia.org/w/index.php?title=Watershed_(image_processing)&oldid=1109774665) (visité le 15/11/2022).