Machine Learning Project: 260 Bird Species

Ryan Belkhir, Salomé Papereux

1. Choix du Dataset

Pour ce projet, nous avons fait le choix de travailler sur des images, le domaine du Computer Vision nous ayant toujours beaucoup intéressé. C'est un outil de l'intelligence artificielle essentiel, largement répandu aujourd'hui, et dont l'expansion et les recherches ne risquent pas de s'arrêter si vite. On peut notamment citer la reconnaissance faciale présente sur de nombreux smartphones, les systèmes de reconnaissance de plaque d'immatriculation dans les parkings, ou encore les voitures autonomes. Ce sont ces attraits qui nous ont poussés à rechercher notre dataset dans cette direction.

Notre monde regorge d'espèces animales dont nous ne connaissons pas pour la plupart l'existence. Du désert du Sahara jusqu'à la forêt amazonienne, certaines même vivent avec nous au quotidien mais nous ne prenons pas le temps de les regarder. C'est le cas des oiseaux, dont il existe presque 11000 espèces différentes connues à ce jour (2021). Du colibri à la cigogne, certaines espèces peuvent sembler très différentes, alors que d'autres sont confondues, comme le Toucan et le Calao. Identifier une espèce à l'œil nu pour des milliers d'oiseaux est une tâche qui peut se révéler longue et fastidieuse et longue. Il est nécessaire d'identifier la forme de l'oiseau, sa taille, son plumage ainsi que les caractéristiques minutieuses qui sont souvent déterminantes, comme un bec recourbé ou la présence de taches noires sur le bout des ailes. La reconnaissance d'oiseaux grâce au Computer Vision pourrait permettre l'identification et la préservation des espèces plus facilement dans les espaces naturels. Ou bien tout simplement introduire plus facilement cet animal aux populations. Nous allons, tout au long de ce projet tenter de reconnaître un oiseaux à partir d'une simple image. Nous nous sommes concentrés sur un total de 260 espèces d'oiseaux, ce qui représente environ 2% des espèces du monde, et comptent parmi les plus répandues.

À cette fin, nous avons utilisé le data set support kaggle : 260 Birds Species . Il comporte environ 40000 images d'oiseaux, appartenant à une des 260 espèces d'oiseaux répertoriées. Alors, nous tenterons de répondre aux problématique suivantes : Existe-t-il un bon modèle pour classifier nos oiseaux ? Si oui, quel est le meilleur modèle, et dans quelle mesure pouvons-nous le qualifier de meilleur ? À travers ces problématiques nous nous focaliserons sur l'objectif de construire un modèle capable de déterminer à laquelle des 260 espèces un oiseau appartient, à partir d'une image donnée.

2. Description du dataset

Description du dataset

Notre data set est le data set kaggle : “260 Birds Species”. Il comporte au total 39 209 images d’oiseaux, appartenant à une des 260 espèces d’oiseaux répertoriées. Ce sont des images en couleurs, toutes de même taille, de dimensions $224 \times 224 \times 3$. Chacun des 260 dossiers contient au moins 100 images, bien que les données ne soient pas uniformément distribuées. L’espèce qui contient le plus d’images est l’espèce “SORA” avec 310 images. Le dataset ne présente aucun doublon. Il est constitué d’images centrées, de sorte que l’oiseau occupe 50% de l’image. Voici un aperçu de la distribution des données dans chaque classe :

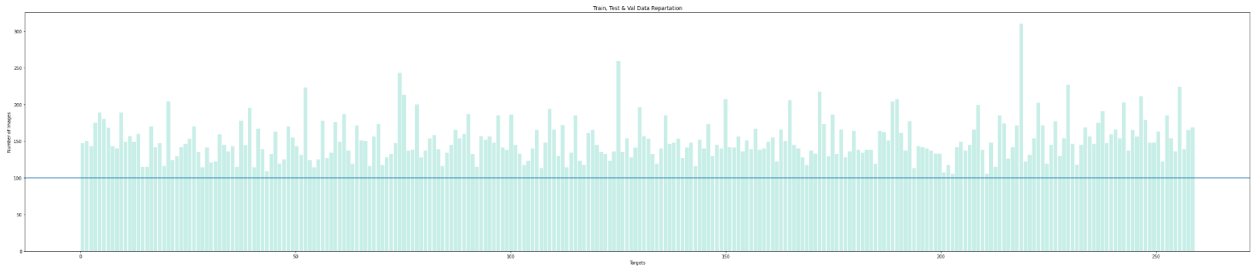

Figure 1: Nombre d’images par classes

Le dossier contenant notre jeu de données se divise en 4 dossiers :

- consolidated : ce dossier contient 260 dossiers, chacun représentant une espèce d’oiseau, contenant à leur tour des images sous le format .jpg
- train, test et valid : ces trois dossiers contiennent chacun 260 dossiers, chacun représentant une espèce d’oiseau, mais contenant respectivement les données du train set, du test set et du validation set.

Nous avons choisi d’utiliser le dossier consolidated afin de créer nos propres jeux de données d’entraînement, de test et de validation. Nous avons choisis de modeler nous-même les sets train, test et validation car ceux déjà crées découpaient les données selon les proportions 90% - 5% - 5%, ce qui à notre sens était trop déséquilibré. Voici un aperçu de la distribution des données dans chacun des sets de train, test et validation, par rapport à l’ensemble des données (une meilleure visualisation du graphique est présente dans le code jupyter) :

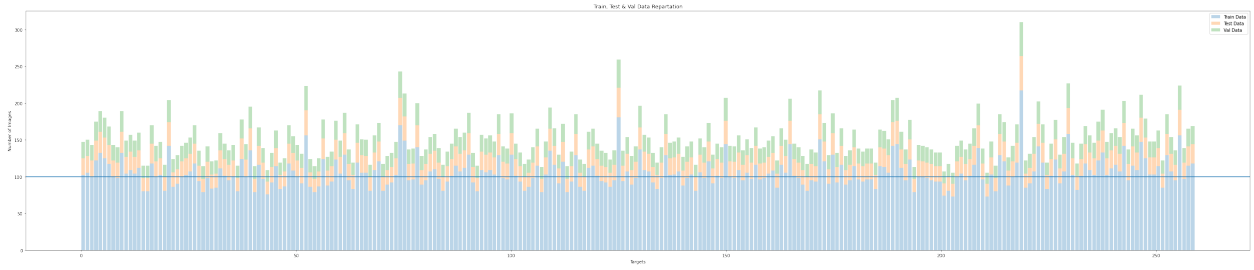

Figure 2: En bleu : train set, en orange le test set, et en vert le validation set

Le site kaggle nous informe qu'il existe un déséquilibre dans les données entre les images des oiseaux mâles et femelles : environ 80% des images sont des mâles et 20% des femelles. Les mâles sont très colorés tandis que les femelles sont généralement plus fades. Cet aspect est à prendre en compte dans nos résultats d'accuracy, et donc d'analyse.

Figure 3: Mâle vs. Femelle chez les American GoldFinch

Nettoyage des données

Ce dataset ne présente ni de données (images) manquantes, ni de données non complètes. Ce dataset n'a donc pas eu besoin d'être nettoyé. Un dataset ne contenant que des images peut difficilement être à nettoyer, d'autant plus que nos images sont sous format .jpg et ne peuvent disparaître, à contrario d'images url.

Toutefois, nous aurions pu effectuer diverses opérations sur les images afin de faciliter notre analyse. Si les données n'avaient pas été plus difficile à manipuler, nous aurions par exemple pu centrer les images sur les oiseaux avec la commande de la bibliothèque Pytorch ***torchvision.transforms.FiveCrop(size)***. Une autre possibilité aurait été de les redimensionner toutes de la même taille. Seulement le dataset a été créé de sorte que l'analyse du dataset puisse être faite immédiatement, et donc toutes ces étapes ne sont pas requises.

Avant de commencer...

Pour résoudre cette tâche, nous allons utiliser différentes méthodes et algorithmes. Nous devons assigner une image à une classe, ce seront donc des méthodes de classification. Nos données sont toutes labellisées, les méthodes d'apprentissage supervisé sont alors les plus adaptées.

Nous avons choisi d'utiliser la bibliothèque pytorch, qui est très intuitive pour manipuler des données images, notamment pour les transformer (normalisation / réduction) et les parcourir. Mais l'intérêt majeur a été pour la création de réseaux de neurones, de convolutions et l'implémentation directe des réseaux pré-entraînés. Nous en reparlerons plus tard. À travers le compte rendu, nous allons détailler les différentes méthodes approchées, pourquoi nous avons choisis d'implémenter celles-ci, et leur fonctionnement.

Le score que nous allons utiliser pour mesurer la qualité de prédiction de nos méthodes est l'accuracy. Nous avons choisi celle-ci car elle s'emploie dans un cadre où les données sont bien équilibrées entre toutes les classes. Comme vu précédemment chaque classe contient au moins 100 images et maximum 310. Étant donné qu'il y a 260 classes, et aux vues de leur répartition (105 en moyenne avec un écart-type de 21), mise en avant par l'histogramme nous avons considéré qu'elles étaient globalement bien équilibrées.

2. Perceptron Multicouches

Nous avons débuté le projet avec l'utilisation d'un réseau de neurones : le Multi Layer Perceptron (MLP). Un simple perceptron aurait été trop peu précis pour de telles données mais surtout non adapté. Cela nous permet de faire un premier pas dans le Deep Learning, en comprenant bien les bases d'un réseau neuronal dans sa globalité. Mais c'est surtout une solution adéquate pour faire de l'apprentissage sur nos 27000 images d'entraînement à des fins de classification. Un MLP distingue des données non linéairement séparables. Cela nous permettra aussi d'avoir une idée de la performance d'un réseau neuronal simple dans ce contexte, et d'avoir un ordre d'idée de précision de prédiction pour un tel data set. Et puis par la suite, de nous fixer une précision "objectif". Nous nous servirons des premiers résultats et observations pour procéder à d'éventuelles améliorations.

Comment fonctionne MLP?

Un MLP est un réseau neuronal. Il diffère d'un perceptron par le nombre de couches à l'intérieur du réseau. Un MLP consiste en trois types de couches :

- **Une couche d'entrée**
  Elle s'occupe de l'entrée des données
- **Des couches cachées**
  Elles s'occupent de transformer et de transporter les données dans le réseau. Leur spécificité est que tous les neurones de ces couches sont entièrement connectés à la couche précédente
- **Une couche de sortie**
  Chaque neurone en sortie correspond à une classe du dataset
- **Des couches Dropout**
  Nous détaillerons plus tard leur utilité

Il est important de noter pour la suite que, pour chacun des sets train, test et val, nous avons fait des groupes de 32 images, appelés batch. Insérés dans des réseaux de neurones, ils permettent d'éviter la modification des poids du neurones après chaque image, en bref, de gagner en temps computationnel.

Étape 1 : Chaque image du batch est passée une à une dans le réseau. Lors du passage d'une image dans le réseau, cette dernière est aplatie. Chaque image est découpée en pixels, eux-mêmes divisés en 3 canaux (images de couleurs), puis ces derniers sont regroupés dans un même vecteur. La valeur d'un canal est représentée par un neurone de la couche d'entrée. Les canaux traversent le réseau couche par couche à la même vitesse.

Étape 2 : Lors de la traversée d'une couche l donnée (position globale dans tout le réseau), chacun des neurones de cette couche va accueillir en entrée toutes les données de la couche précédente. Lors du transfert d'information d'un neurone de la couche précédente n_{l-1} à un neurone de la couche courante n_l, l'information transportée est associée un poids – l'arc (n_{l-1}, n_l) est associé à un poids – qui lui donne alors plus ou moins d'importance.
Alors le neurone j de la couche cachée l, rassemble et effectue des opérations sur les informations reçues de l'ensemble des neurones de la couche précédente $l-1$. Soit $n_{l,j}$ la sortie du neurone j de la couche l, $w_{ij,l}$ le poids de l'arc $(n_{l-1,i}, n_{l,j})$, N le nombre total de pixels et $s_{l,j}$ la somme pondérée du neurone j de la couche l. On observera que pour tout l, $n_{l,0} = 1$ et pour tout j, l $w_{0j,l} = 1$: c'est le biais. Alors pour toute couche cachée l :

- Le neurone effectue la somme pondérée des données :

$$s_{j,l} = \sum_{i=0}^{n} n_{l-1,i} \; w_{ij,l}$$

- Puis est appliqué à cette somme la fonction d'activation ReLU, la fonction $f(x) = \max(0, x)$:

$$n_{l,j} = \max(0, s_{j,l})$$

Ces opérations sont effectuées sur chaque couche cachée.

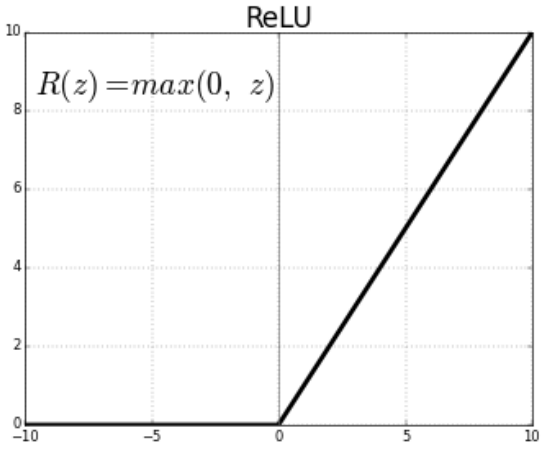

Figure 4: Fonction ReLU

Étape 3 : Pour la couche de sorties s, on se contente de calculer la somme pondérée des neurones de la dernière couche cachée d. Les neurones de s sont complètement connectés aux neurones de d. La couche de sortie contient un nombre de neurones correspondant au nombre de classes du dataset. Est renvoyé par le réseau neuronal un poids pour chacun des neurones. La classe prédite par le réseau correspond au numéro du neurone contenant un poids maximal.

Nous avons détaillé ci-dessus un passage d'un batch dans le réseau neuronal. À la fin de chaque batch, la prédiction des classes en sortie est comparée avec les vraies classes. Les poids des arcs sont mis à jour afin d'optimiser un critère choisit en amont, et de mieux prédire les prochaines données. C'est le backpropagation pour l'apprentissage. Tout ce procédé précédemment décrit est répété autant de fois qu'il y a de batch, autant de fois qu'il y a d'epoch (un entier que nous avons fixé). Plus l'epoch est grand, plus le réseau apprend, mais plus le temps d'exécution est long.

Notre MLP

Un point très important que nous souhaitons soulever avant de détailler notre structure : nous avons entraîné nos données sur le train set. À chaque epoch nous calculons l'accuracy sur le train set, et le validation set. Cela nous permet de vérifier que le modèle ne fait pas de surapprentissage, tout en évitant les fuites de données de notre test set. Nous choisissons le réseau final selon la meilleure accuracy sur la validation set. Seulement à la fin de l'apprentissage est calculée la précision sur le test set. Si les données du train set sont normalisées, alors l'accuracy sera effectuée sur les images normalisées du test set – avec l'espérance et l'écart-type du train set. Cela évite les fuites de données, et est valable pour toute la suite du codage des réseaux neuronaux.

Pour trouver le MLP aujourd'hui obtenu, nous avons construit plusieurs réseaux, plus ou moins effectifs, en faisant varier le nombre de couches cachées, et le nombre de neurones qu'elles contiennent. Nous avons gardé le réseau le plus efficace, en termes d'accuracy sur le validation set, de temps d'exécution, et qui n'effectue pas de surapprentissage.

On fait passer en entrée de notre MLP le train set normalisé. Notre réseau neuronal comprend :

- Une couche d'entrée de taille 224×224×3 : soit la taille d'une image 224×224×3=150 528 canaux.
- Trois couches cachées de tailles respectives 520, 1024 et 512
- Deux couches Dropout : une entre la première et la seconde couche cachée, l'autre entre la seconde et la troisième couche cachée
- Une couche de sorties ($l = 5$) constituée de 260 neurones, chaque neurone c_i correspondant à la classe i.
- Le critère à optimiser choisi est la CrossEntropy avec la Descente de Gradient Stochastique

N.B: Lors de l'entraînement du modèle, nous remplaçons le Learning rate par une fonction nommée lr_scheduler. Elle renvoie des Learning rate, qui diminuent progressivement à la fin de chaque série de x epochs, nombre fixé à l'avance. Dans notre exemple, le Learning rate est divisé par 10 toutes les 4 epochs. Nous avons employé cette technique pour toutes les méthodes d'apprentissage de notre projet.

N.B^2: Nous avons utilisé la Cross Entropy et la Descente de Gradient Stochastique pour l'ensemble des modèles de notre projet (fixé aux nombreux tests manuels).

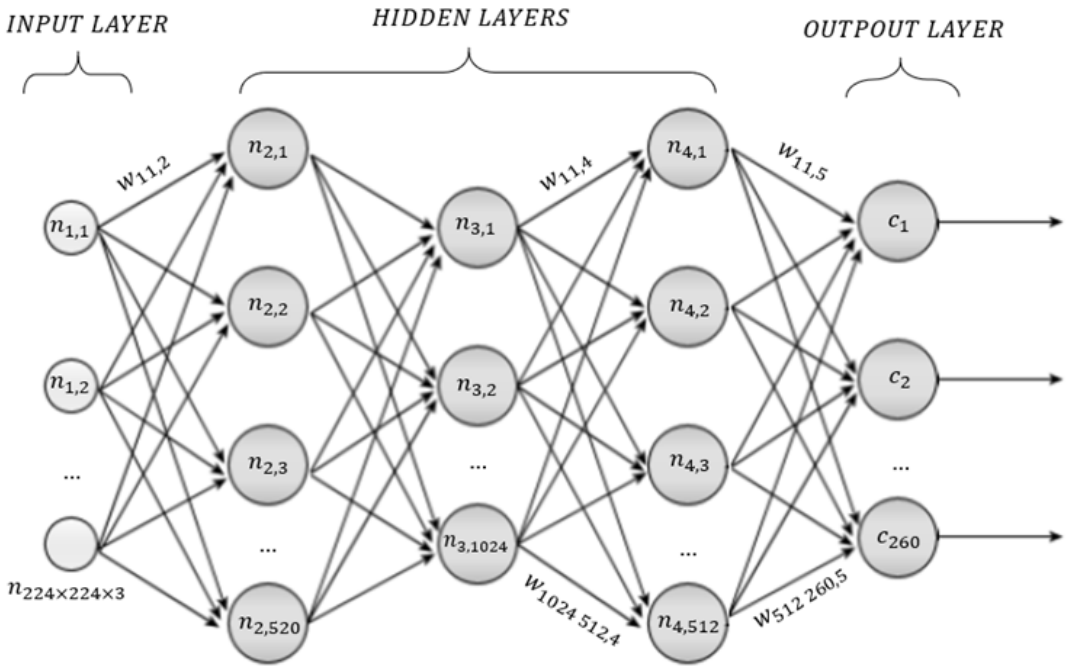

Figure 5: Aperçu de notre MLP

Attendu à priori : Pour être francs, nous n'avons aucune idée de la précision que nous pouvons attendre d'un tel schéma, aux vues des données volumineuses et des 260 classes. Ce premier résultat nous donnera un aperçu de l'accuracy finale que l'on pourrait espérer atteindre. Toutefois, nous ne nous attendons pas non plus à de bons résultats.

Premiers résultats : Nous avons une accuracy autour de 15%. Néanmoins le modèle surapprend énormément : l'accuracy du train set avoisine les 40%.

Changement à postériori : Ajout de couches dropout afin de réduire le surapprentissage de notre modèle.

COUCHE DROPOUT : Nous avons utilisé des couches Dropout à plusieurs reprises après les couches cachées. Elles permettent d'éviter le surapprentissage. Ces couches n'ont aucun paramètres mais ont la capacité de fixer les couches d'activation précédentes à zéro, avec un probabilité égale à leur hyperparamètre ratio. Effectivement, dans un réseau neuronal, l'apprentissage peut rendre certains neurones hautement dépendants d'autres neurones. Si un neurone reçoit une mauvaise entrée, alors le neurone dépendant va être affecté. Cela peut altérer la performance du modèle, qui peut alors provoquer un surapprentissage : c'est la coadaptation. Cela force le processus d'entraînement à incorporer un degré de redondance dans le modèle appris, qui évite la coadaptation. Après l'entrainement, ces couches dropout sont retirées pour profiter pleinement du pouvoir prédictif du réseau neuronal construit.

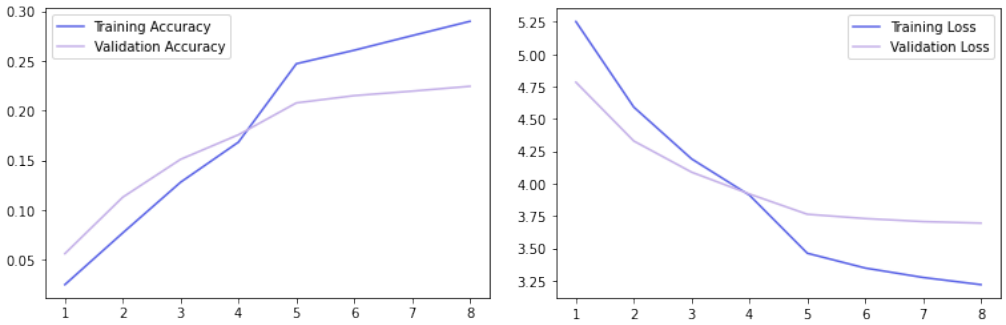

Figure 6: MLP: Accuracy et Loss par epoch

Résultat final : L'accuracy est d'environ 20%. C'est un résultat que nous trouvons très satisfaisant pour un premier pas dans les réseaux neuronaux, surtout aux vues de la quantité de données et de classes. Nous ne pensions pas avoir une telle précision : 1/5 des images sont correctement classées. Cela nous donne bon espoir pour la suite. Nous avons essayé d'augmenter le nombre de couches de neurones pour améliorer l'apprentissage. Seulement le temps d'exécution est beaucoup plus long, et les données sont surapprises ; le modèle est trop complexe. Les couches cachées du réseau de neurones devant être limitées pour ne pas aggraver la complexité, l'accuracy est d'autant plus freinée. Toutefois, nous avons en parallèle grandit un maximum le nombre de neurones de chacune des couches pour améliorer la précision, en ne perdant pas de vue l'objectif d'un temps d'exécution raisonnable. Nous souhaitons faire remarquer que le réseau surapprend légèrement. En termes de précision, nous serions ravis si nous pouvions atteindre une accuracy d'au moins 50% à la fin de ce projet. Cela semble être un objectif raisonnable et atteignable.

3. Réseaux Neuronaux Convolutifs

Un simple MLP ne peut pas complètement satisfaire les attentes que nous avons vis-à-vis de notre data set. Par la suite nous avons décidé de complexifier notre structure, aspirant à améliorer notre précision de prédiction. Une autre solution plus adaptée aux données volumineuses de notre dataset et au nombre de classe est le réseau de convolution (CNN).

Comment fonctionne un CNN ?

Un réseau convolutionnel est une version très améliorée d'un MLP, mais qui possède des structures en commun avec ce dernier. Le format, plus complexe, se divise selon les couches suivantes :

- **Une couche d'entrée**
- **Des couches convolutionnelles**
- **Des couches BatchNorm**
- **Des couches de pooling**
- **Des couches Fully Connected (FC)**
- **Une couche de sortie**

Étape 1: En entrée du CNN est passé une image. Puis les données traversent le réseau couche par couche. Le premier type de couche rencontré est la couche convolutionnelle. L'objectif final est d'extraire des caractéristiques propres à chaque image en les compressant de façon à réduire leur taille initiale. L'image entrée traverse une succession de filtres (petites matrice carrées 3D de réels), ce qui va créer de nouvelles images appelées cartes de caractéristiques. Les filtres correspondent exactement aux caractéristiques que l'on souhaite retrouver dans les images. On obtient pour chaque pair (image, filtre) une carte d'activation, qui nous indique où se situent les caractéristiques dans l'image : plus la valeur est élevée plus l'endroit correspondant dans l'image ressemble à la caractéristique. Voici le détails des calculs afin d'obtenir la taille de la sortie d'une couche de convolution :

Soit $W \times H \times D$ la taille de l'image. Un filtre est une matrice de taille $F \times F \times D$. Soit S le stride (pas de déplacement du filtre sur l'image), et P le zéro-padding : on ajoute à l'image en entrée de la couche un contour noir d'épaisseur P pixels. Pour chaque image de taille $W \times H \times D$ en entrée, la couche de convolution renvoie une matrice de dimensions $W_p \times H_p \times D_p$, où :

$$W_p = \frac{W - F + 2P}{S} + 1, \quad H_p = \frac{H - F + 2P}{S} \quad et \quad D_p = D$$

Grâce à cette transformation, on a extrait ce que l'on appelle des caractéristiques. Chaque passage dans une couche de convolution participe à la création de nouvelles caractéristiques, plus spécifiques que les précédentes. On effectue par la suite une correction ReLU sur les données en sortie.

Étape 2 : Pour faciliter la manipulation de cette nouvelle image à la sortie de la couche convolutionnelle, on va réduire sa taille. Pour ceci, on la passe dans la couche de max-pooling qui va diviser le nombre de pixels en longueur et en largeur : l'idée est de découper l'image en cellules carrées de taille $F \times F$ pixels, et de ne garder que le plus grand canal d'une région de l'image. On dénote F la taille du filtre de max-pooling. Les dimensions de sortie de la couche de pooling et d'entrée dans la couche suivante vont réduire selon les calculs décrits ci-après. Cela nous permet de gagner en temps d'exécution. Soit F la taille du filtre, et S le stride (pas de déplacement du filtre sur l'image). Pour

chaque image de taille $W \times H \times D$ en entrée, la couche de pooling renvoie une matrice de dimensions $W_P \times H_P \times D_P$, où :

$$W_p = \frac{W - F}{S} + 1, \quad H_p = \frac{H - F}{S} \quad et \quad D_p = D$$

On obtient en sortie le même nombre de cartes de caractéristiques qu'en entrée, mais celles-ci sont bien plus petites. Notre CNN répète plusieurs fois les étapes convolution + BatchNorm + ReLU + max-pooling.

Étape 3 : Enfin, les données traversent les couches Fully-Connected. De la même manière que pour un MLP, ses neurones sont entièrement connectés aux neurones de la couche précédente. Il peut s'ensuivre une multitude de couches FC.

Étape 4: La couche de sortie est une couche FC qui contient autant de neurones que le nombre de classes du dataset. Est renvoyé par notre réseau neuronal un poids pour chacun des neurones de la couche de sortie. La classe prédite par le réseau correspond au numéro du neurone contenant un poids maximal.

Nous avons utilisé des couches BatchNorm à plusieurs reprises, après les couches de convolution. La normalisation par batch permet de normaliser la sortie d'une couche de convolution. Elle possède trois intérêts majeurs :

- Elle améliore la vitesse de convergence de la loss, le modèle apprend plus vite et donc moins d'epochs sont nécessaires
- Elle permet d'avoir des meilleurs résultats indépendamment de l'initialisation aléatoire de notre modèle
- Elle aide à réduire le surapprentissage grâce à la formation de batch aléatoires

Ces réseaux possèdent des caractéristiques importantes et avantageuses pour traiter nos données :

- Un poids unique est associé aux données entrantes dans tous les neurones d'un même filtre de convolution. Cette caractéristique réduit l'utilisation de la mémoire.
- La dimension des images entre chaque couche de convolution est réduite grâce aux couches dites de max-pooling, ce qui nous permet en contrepartie d'avoir un grand réseau, donc une plus grande rapidité avec une meilleure classification en sortie.

Notre CNN

Pour trouver le CNN aujourd'hui obtenu, nous avons construit plusieurs réseaux, plus ou moins effectifs, en faisant varier le nombre de couches de convolutions + pooling et Fully-Connected. Nous avons inséré des BatchNorm aux endroits où les prédictions étaient les meilleures. Finalement, nous avons gardé le réseau le plus efficace, en termes d'accuracy sur le validation set, de temps d'exécution, et qui n'effectue pas de surapprentissage.

On fait passer en entrée de notre CNN les images normalisées. Notre CNN comprend :

- Une couche d'entrée de taille 3 correspondant au nombre de canaux, étant donné que nos images sont en couleurs
- Quatre couches de convolutions de tailles respectives 64, 128, 256 et 256
- Quatre couches BatchNorm de tailles égales aux couches de convolution auxquelles elles font suite
- Quatre couches de max-pooling, chacune faisant suite à une convolution + BatchNorm + ReLU.
  La taille du filtre de max-pooling est de 2×2×3, et le pas du stride S est 2
- Trois couches Fully Connected + ReLU de tailles respectives 1600 et 3400
- Une couche de sorties Fully Connected, constituée de 260 neurones, chaque neurone correspondant à une classe

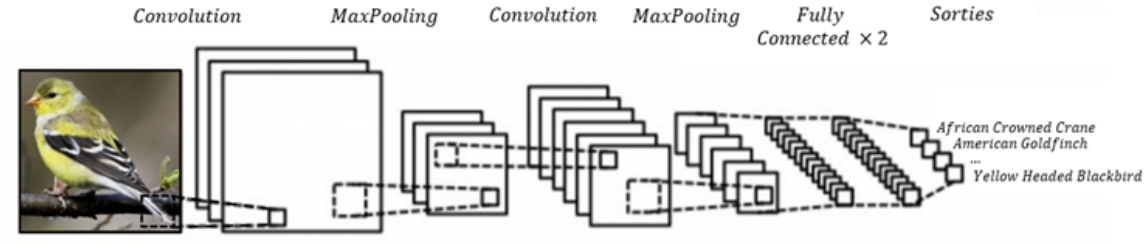

Figure 7: Fonctionnement des différentes couches

Difficultés rencontrées

Une difficulté que nous avons rencontré pendant la création des CNN est le choix du premier argument à passer dans la première couche linéaire, à la fin de la succession des couches de convolutions et de pooling. Cet argument se trouve grâce à des calculs, et correspond à la dimension renvoyée par le pooling de la dernière et quatrième couche de convolution. Alors, pour trouver cette dimension on doit trouver la dimension récupérée en sortie du pooling de la troisième couche de convolution, elle-même basée sur la sortie du pooling de la seconde couche, et ainsi de suite.

Soit W_i la largeur de l'image à la fin de chaque couche de convolution + de pooling : W_0 est la largeur de l'image avant son passage dans le réseau. F représente la taille du filtre de la couche de convolution, P le zéro-padding, S le stride de convolution. Enfin, notons F_P la taille du filtre pooling et S_P le stride du pooling. On note W_{c_i} la taille de l'image après la couche de convolution i et W_{P_i} la taille de l'image après le pooling i. L'avantage est que les images sont carrées, et donc $W_{c_i} = H_{c_i}$ et $W_{P_i} = H_{P_i}$, d'où $W_i = H_i$.

$$W_{c_i} = \frac{W_{i-1} - F + 2P}{S} + 1, \quad W_{p_i} = \frac{W_{c_i} - F_p}{S_p} + 1$$

$$\textit{Le resultat en sortie est donc } W_i = \frac{\frac{W_{i-1} - F + 2P}{S} + 1 - F_p}{S}$$

Un bon réseau de convolution que nous avons construit a les caractéristiques : $\begin{cases} W_0 = 224 \\ F = 5 \\ P = 0 \\ S = 1 \\ F_p = 2 \\ S_p = 2 \end{cases}$

$$Donc\ W_i = \frac{(W_{i-1} - 5) + 1 - 2}{2} + 1 = \frac{W_{i-1} - 6}{2} + 1 = \frac{W_{i-1}}{2} - 2$$

On a alors :

$$\begin{cases} W_1 = \frac{224}{2} - 2 = 110 \\ W_2 = \frac{110}{2} - 2 = 53 \\ W_3 = \lfloor \frac{53}{2} \rfloor - 2 = 24 \\ W_4 = \frac{24}{2} - 2 = 10 \end{cases}$$

Cette dimension récupérée en sortie est donc 10. Les cartes de caractéristiques en sortie sont donc de taille 10×10.

Attendu à priori : Nous nous attendons à une bien meilleure accuracy que le MLP aux vues des avantages cités précédemment. Mais nous n'avons pas d'idée de la marge de différence. Toutefois, on espère obtenir une accuracy entre 40 et 50%. Pour nous une accuracy $\frac{1}{2}$ serait idéale, aux vues du nombre de classes. De plus, la structure complexe d'un réseau de convolution et sa réputation dans le monde du Computer Vision nous poussent à espérer que nous pourrons obtenir une telle précision.

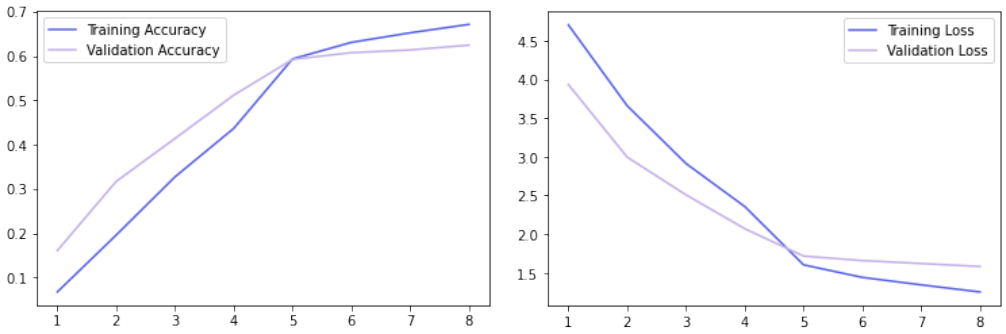

Figure 8: CNN: Accuracy et Loss par epoch

Résultats : Les résultats surpassent nos espérances : on observe une nette amélioration de l'accuracy, qui est de 60%. Ce résultat nous semble très efficace et pertinent pour un total de 260 classes. Nous avons atteint nos objectifs. Par ailleurs, durant nos recherches nous sommes tombés sur des articles parlant d'une méthode pour aller encore plus loin et d'améliorer notre accuracy. Nous allons utiliser des modèles pré-entraînés : le Transfer Learning.

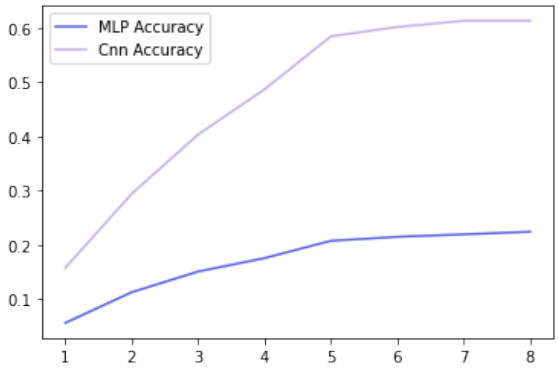

Figure 9: MLP vs. CNN sur le Validation test par epoch

4. Transfer Learning

Qu'est-ce que le Transfer Learning ?

Une solution très efficace pour pallier aux problèmes de rapidité et de la quantité de données est le Transfer Learning (TL). Le Transfer Learning nous permet de transférer la connaissance acquise sur un jeu de données source pour mieux traiter d'autres jeu de données, comme notre dataset des oiseaux.

Un réseau de TL est un réseau de convolution déjà entièrement fabriqué, sur lequel le jeu de données source est entraîné. On va alors récupérer ce réseau de convolution et à faire passer nos données en entrée. On va au préalable effectuer quelques modifications pour l'ajuster à nos images et à nos classes :

- L'idée va être de geler les poids du modèle de certaines couches pendant l'entraînement. On ne calculera plus tous les gradients pour mettre à jour les poids. Effectivement, nous ne voulons pas entièrement améliorer le réseau, mais s'appuyer dessus.
- La dernière couche FC du réseau de pré-entraînement, qui classifie le jeu de données source, est remplacée par une nouvelle couche FC qui va classifier nos données sur les oiseaux. La sortie de cette couche possèdera donc 260 neurones : le nombre d'espèces possibles.

Nous avons en premier lancé l'apprentissage de nos données sur le réseau DenseNet, sur des batchs d'images rétrécies de taille $52 \times 52 \times 3$ pixels pour gagner en rapidité. Toutefois nous nous sommes confrontés à une très faible accuracy. Après quelques recherches, nous nous sommes rendu compte qu'il était primordial d'avoir des données au moins de même longueur et au moins de même largeur que les images sur lequel le modèle a été pré entraîné. Nous avons conservé la taille de nos images: $224 \times 224 \times 3$. Cette taille est la dimension minimale acceptée pour les réseaux de Transfer Learning de l'ensemble de la bibliothèque pytorch.

Nous avons entraîné notre data set sur les cinq réseaux de pré-entraînement ci-dessous. Chacun prend en entrée les images du train set normalisées :

- **DenseNet161**
- **ResNet18**
- **GoogLeNet**
- **VGG16**
- **AlexNet**

Attendu à priori : Aux vues des caractéristiques présentées ci-dessus, nous nous attendons à une nette amélioration de la précision, et de la rapidité. On espère une meilleure accuracy que pour le CNN (qui était de 60%).

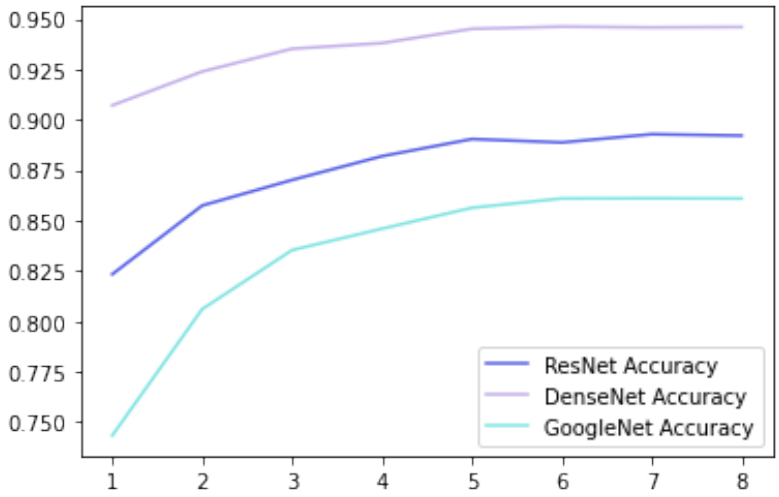

Figure 10: Accuracy sur le Validation set par epoch

Résultats : Nous sommes très agréablement surpris des accuracies obtenues (tableau ci-dessous). Malgré nos suspicions de meilleures performances comparé à nous réseaux construits jusque-là, nous ne nous attendions pas non plus à atteindre 94%. La meilleure accuracy depuis le début du projet. C'est un excellent score qui nous permet de mettre en avant un modèle qui fonctionne très bien, et qui fait moins de 6% d'erreurs de prédiction sur 260 classes. Nous réalisons l'efficience d'une telle stratégie d'apprentissage.

VGG et AlexNet n'ont pas été concluants. Nous avons donc gardé les trois réseaux suivants :

	DenseNet	ResNet	GoogLeNet
Accuracy Test Set	94.47%	89.54%	84.91%

5. Features Extraction

Nous avons ensuite utilisé l'approche de l'extraction de features : l'idée est de récupérer les connaissances apprises du réseau de neurones pour les appliquer sur d'autres méthodes de classification. À cette fin, nous nous sommes appuyés sur les réseaux de Transfer Learning et en avons extrait les features : on propage notre dataset d'images à travers le réseau, on extrait les activations à l'avant-dernière couche (en traitant les activations comme un vecteur de features), puis on sauvegarde les valeurs sur le disque. En effet la dernière couche de classification ne nous intéresse plus puisqu'on cherche à utiliser une autre méthode pour le faire. Voici différentes méthodes que nous avons appliqué :

- **Sector Vector Machine**
- **Stochastic Gradient Descent**
- **Random Forest**
- **K-Neighbors classifier**

Prenons un réseau de pré-entraînement arbitraire parmi les trois utilisés. Nous faisons traverser notre jeu de données d'entraînement et de validation à travers ce réseau. Comme l'apprentissage se fera tout seul et que nous n'aurons pas d'aperçu dessus, nous n'avons pas besoin d'un validation set, d'où la fusion du train set avec le val set. On extrait par la suite les activations de l'avant-dernière couche – en traitant les activations comme un vecteur de features – puis on sauvegarde les valeurs sur le disque. On a alors récupéré, pour chaque image de notre train set + validation set, des valeurs associées à chacune des caractéristiques extraites. On obtient alors une matrice, disons M. Soit $x_i^{(j)}$ la caractéristique j de l'image i, f le nombre de features extraites du modèle, et N le nombre d'images du dataset étudié (train + val et test).

$$M = \begin{pmatrix} x_1^{(1)} & \dots & x_1^{(f)} \\ \vdots & & \vdots \\ x_N^{(1)} & \dots & x_N^{(f)} \end{pmatrix}$$

Nous avons extrait les features de 5 modèles de pré-entraînement : DenseNet, ResNet, GoogLeNet, AlexNet et VGG. Toutefois, nous n'avons pu effectuer de classification sur deux d'entre eux car le nombre de features était trop important : 20 000+ pour VGG et 9000+ pour AlexNet. Leur exécution prend trop de place mémoire, occupe tout le processeur, en plus d'être très longues. Ces deux réseaux ne sont donc pas adaptés à nos données.

Pour lancer les méthodes, nous avons dû choisir leurs hyperparamètres. Ces choix se sont fait grâce à des Grid Searchs pour chaque méthode pour chaque réseau de Transfer Learning. Cela nous a pris énormément de temps en termes d'exécution, et nous avons dû restreindre les possibilités. À noter que parfois, lors d'un ajout d'une instanciation possible pour un hyperparamètre d'une méthode de classification (par exemple kernel = 'rbf' pour un SVM), les couples de meilleurs paramètres peuvent changer (le meilleur couple qui était alors (C=1, kernel = 'poly') peut devenir (C=10, kernel = 'rbf')). Alors aucun raccourci n'a pu être emprunté et nous avons décidé de resserrer les possibilités pour alléger le temps computationnel.

Les sélections ont été :

1. **Sector Vector Machine**
    . C = 0.001, 0.01, 0.1, 10, 100
    . kernel = 'linear', 'rbf'
2. **Stochastic Gradient Descent**
    . loss = 'hinge', 'loss', 'perceptron'
3. **Random Forest**
    . n_estimators = 10, 30, 50, 100
    . criterion = 'gini'
4. **K-Neighbors classifier**
    . n_neighbors = 7, 13
    . Ni la RAM ni le GPU n'ont supporté le lancement du Grid Search, même sur un total de deux possibilités. Nous avons donc trouvé à tâtons des hyperparamètres qui rendent le classifieur assez efficace

Attendu a priori : Nous nous attendons à une accuracy avoisinant les 50-60%. On se pose la question dans quelle mesure appliquer une méthode de classification autre que la méthode neuronale prévue pour, fonctionnera sur nos données. La quantité de features n'est pas rassurante pour le temps d'exécution, étant habitués à avoir peu de features – comparé au dataset Birds – lors des TD.

Voici les résultats que nous avons obtenu pour chacune des approches selon le modèle appliqué aux données :

Modèle / Accuracy	DenseNet (2208 features)	ResNet (512 features)	GoogLeNet (1024 features)
SVM	93.7%	88.9%	79.0%
Stochastic Gradient Descent	92.8%	85.30%	74.67%
Random Forest	75.95%	69%	56.19%
K-Neighbors Classifier	88.25%	78.57%	63.5%

Résultats : Nous sommes ravis de l'accuracy retournée, à laquelle nous n'étions pas préparé : des accuracies à 90%, pour les meilleurs méthodes. Le SVM sur l'extraction de features depuis le réseau DenseNet nous offre le joli score de 93.7% d'accuracy, presque autant que pour le Transfer Learning avec DenseNet, pour 4 minutes d'exécution. Ce dernier est le meilleur réseau pour faire de l'extraction de features sur notre dataset.
Nous souhaitons aussi soulever l'extrême rapidité d'exécution du classifieur KNN sur les features de DenseNet, tout en ayant une accuracy de 88% : c'est jusqu'ici le modèle le plus équilibré. Les accuracies des modèles SGD et SVM égalent les résultats obtenus avec le Transfer Learning. Globalement, le TL reste meilleur. Finalement, le moins bon modèle à appliquer sur les features extraites est le Random Forest.

6. Data Augmentation & Autres

Data Augmentation

Afin d'éviter le surapprentissage des images, allant de la petite feuille verte de l'arbre en bas à gauche, au reflet du soleil sur le lac où se baigne un cygne, nous allons appliquer l'approche de la data augmentation. Le principe est de retourner horizontalement une image, comme son reflet dans un miroir. Ainsi, on évite d'avoir des oiseaux systématiquement tournés vers l'Ouest, ou le Nord, si le dataset, ou une classe du dataset est constituée comme telle. Alors nécessairement on évitera le surapprentissage, et on diversifiera les données du dataset. Nous avons effectué les distorsions suivantes :

- Nous retournons horizontalement les images avec un probabilité de $p = 0.5$
- Nous faisons pivoter les images par angle dans la plage de degrés (-20 ; 20)

En plus, nous avons normalisés les images. Toutes ces modifications ont été effectuées grâce aux méthodes de la bibliothèques pytorch.

N.B. : Les changements cités ne sont effectués que sur le train set.

Attendu à priori : Ces changements sont à nos yeux un gain d'information : nous retournons l'image sans perdre son contenu. Nous espérons une accuracy non moins grande par rapport aux données non modifiées. Si toutefois l'accuracy n'est pas améliorée, nous nous attendons au moins à une forte corrélation entre l'accuracy du validation set et du test set.

Noir et Blanc

Une autre transformation des images que nous avons essayée est la conversion des images en noir et blanc.

Attendu à priori : Nous pensons que la précision de prédiction sera moins élevée, car de nombreux oiseaux se différencient par leurs couleurs. Nous pensions que cela pouvait être une bonne idée de le faire pour n'avoir qu'un seul canal au lieu de 3 et de gagner en complexité. Seulement, si nous voulons faire passer les images en noires et blanc dans les réseaux de Transfer Learning, elles doivent avoir exactement 3 canaux. Nous effectuons la transformation quand même pour identifier si justement les couleurs jouent un rôle important dans la prédiction, ou bien si ce sont surtout les formes qui justifient l'accuracy.

Images non normalisées

Nous avons tenté de comparer nos résultats avec ceux sur des images non initialement normalisées. Nous avons simplement repris notre dataset initial sans transformer les images.

Attendu à priori : Nous nous attendons à des accuracies moins bonnes et un apprentissage plus lent, nous voulons voir à quel point.

Résultats:

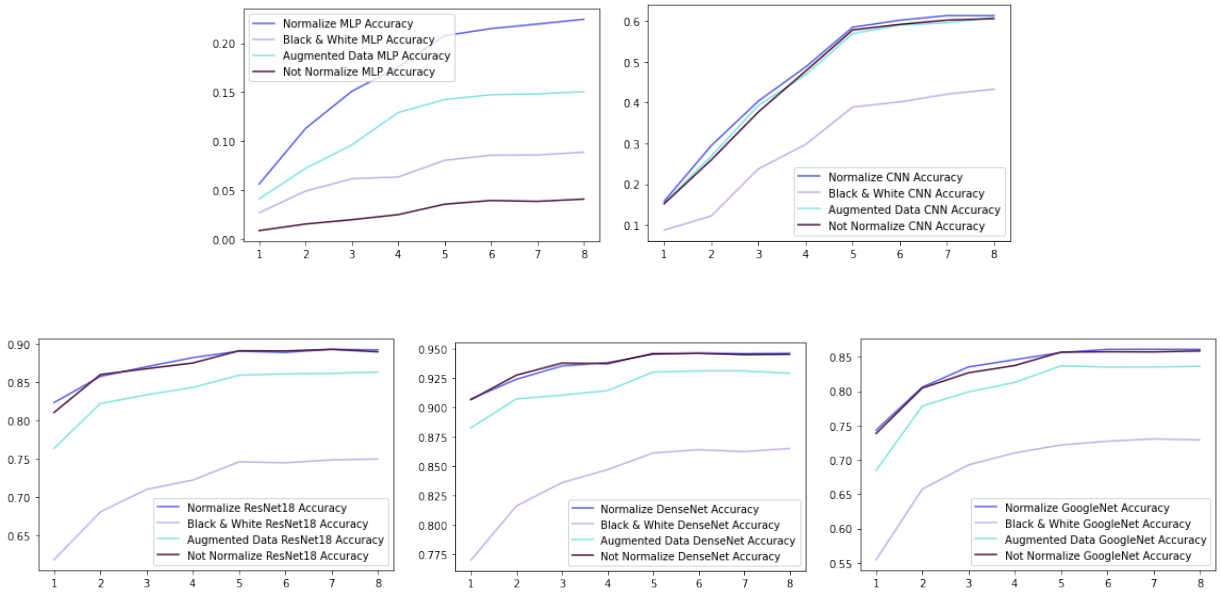

Figure 12: Accuracy sur le Validation Set par epoch

Les accuracy sur le Test set sont :

Modèle/Accuracy	MLP	CNN	DenseNet	ResNet	GoogLeNet
Data Augmentation	14.98%	61.05%	93.06%	86.4%	82.66%
Noir et Blanc	8.39%	42.57%	85.92%	74.66%	72.1%

Data Augmentation : Les résultats sont quasiment les mêmes que sur les données non augmentées. On les rappelle :

Modèle/Accuracy	MLP	CNN	DenseNet	ResNet	GoogLeNet
Normalize	20%	60%	94.47%	89.28%	85.10%

Noir et Blanc : Les résultats ne sont pas concluants, en tout cas beaucoup moins que les autres approches : la couleur est déterminante pour la prédiction d'une espèce d'oiseau. Les réseaux de pré-entraînement, quant à eux, ont des résultats très corrects et intéressants. Nous pouvons supposer qu'ils s'intéressent moins aux couleurs et davantage aux formes.

Données non normalisées : Les résultats sont presque les mêmes que pour les images normalisées. Nous sommes surpris. Comme nous pouvons le voir sur le graphique ci-dessous, la précision des données non normalisées est légèrement moins bonne pour les premières epochs, mais sa courbe finit par rattraper la courbe des images normalisées. Après réflexion, le facteur qui entre en jeux dans ces résultats sont les couches batchNorm, qui normalisent les données en sortie des couches de convolution, et qui nous ramènent à travailler sur presque les mêmes données que les données normalisées. Cette couche est présente dans les CNN, et les réseaux de pré-entraînement. Elles ne sont pas dans le MLP, d'où les résultats que nous obtenons. Nous notons tout de même que les résultats obtenus restent moins bons et que la convergence est moins homogènes que pour la version normalisée.

7. Nos manqués

OpenCV

Vous remarquerez dans le code du projet une section OpenCV. Nous nous sommes essayés à cette bibliothèque pour approcher une autre structure d'apprentissage supervisé, et comparer avec les autres classifieurs.

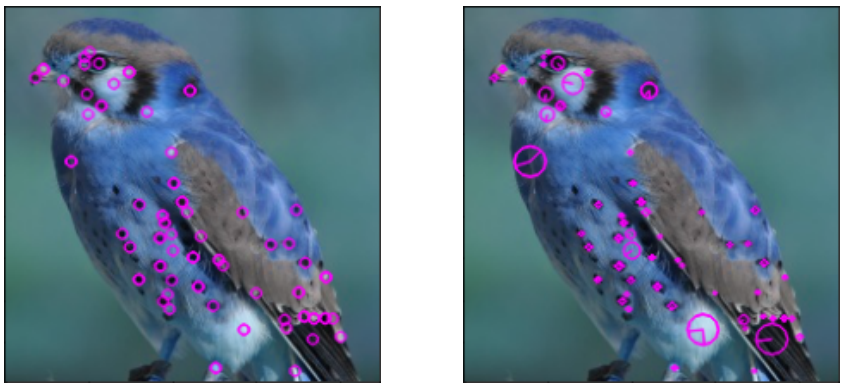

Figure 13: Extraction de features avec SIFT

Nous avons utilisé deux algorithmes d'extraction de features :

1. SIFT
2. SURF

Comparé à tout ce qui été fait précédemment, on extrait les features directement sur les images. Nous avons voulu extraire les features sur l'ensemble de notre train set, mais cela a été computationnellement impossible car nos processeurs ne sont pas assez puissants, nos espaces mémoires pas assez grands, et nous ne pouvons pas nous permettre de lancer le code pendant des semaines. Par soucis matériel, nous nous sommes concentrés sur 10% du train set pour apprendre.

Chaque feature extraite est un vecteur d'une taille prédéfinie (128 pour SIFT et 64 pour SURF). On en extrait au plus 80 par images. Après avoir étudié l'ensemble des images, on se retrouve avec une matrice, dont une dimension représente le nombre total de features extraites, l'autre la taille des features. Les algorithmes détectent tous seuls ces points clés. On regroupe les features en 50 clusters. Ces 50 clusters deviennent les nouvelles 50 features, qui ont été réduites : elles sont calculées à partir de la moyenne des features dans le même cluster. Par la suite, on crée un dictionnaire de features de taille 50, appelé Bag of Words .

Notre image est définie par un vecteur X de la taille de notre dictionnaire, donc 50. X_i correspond au nombre d'occurrences de la $i^{ème}$ feature du dictionnaire dans l'image.

Puis nous entrons dans la phase de transformation de nos images en features : nous traversons ensuite chacun de nos sets de train, test, et validation. Pour chaque image nous extrayons les features grâce aux algos SIFT ou SURF. Nous obtenons alors un vecteur de données, chacune représentant le cluster/feature à laquelle il a le plus de chance d'appartenir (donc un chiffre entre 1 et 50).

Puis nous faisons un histogramme sur ce même vecteur. En fonction des features plus ou moins présentes sur l'image, nous apprenons la classe à laquelle il appartient. Ainsi, par un exemple, un oiseau avec 10 points noir sur l'aile, 5 plumes sur la tête est un perroquet. C'est comme ceci que nous allons classifier nos données : à l'aide des histogrammes des données apprises.

Attendu à priori : Nous trouvons cette méthode très maligne et plutôt robuste. Nous nous attendons à avoir une bonne accuracy. Toutefois nous nous doutons qu'apprendre le modèle avec seulement 10% du train set sera pénalisant.

Résultat : Nous avons été très surpris de n'avoir que 11.84% d'accuracy et 14.86% pour SIFT et pour SURF, aux vues de telles méthodes. Nous pensions avoir de meilleurs résultats malgré le fait que nous n'ayons pu nous entraîner que sur 10% du set prévu. Plusieurs autres facteurs ont pu influencer ce résultat : nous aurions souhaité avoir plus que 50 features pour avoir un meilleur apprentissage, mais le temps d'exécution nous a restreint. En affichant quelques images avec les caractéristiques choisies par les algorithmes nous nous sommes rendu compte que les algorithmes apprenaient les éléments autour de l'oiseau, et par conséquent elles sont considérées comme des features. Comme l'on peut voir sur les photos ci-dessous, les branches des arbres sont des features : toutes la branche est couverte de cercles violets.

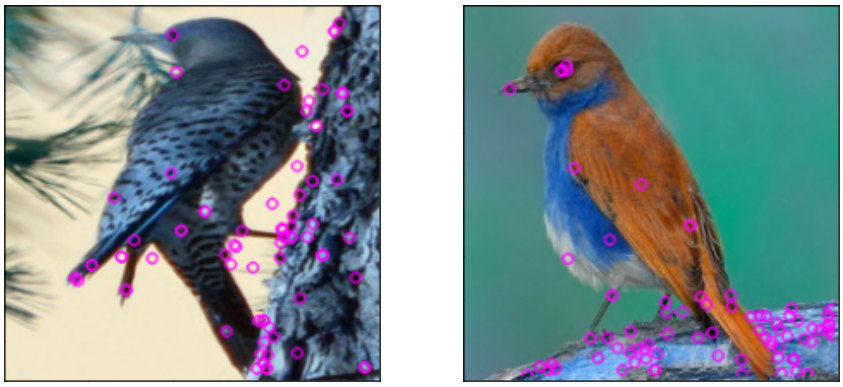

Figure 14: Erreurs de features SIFT/SURF

Ceci peut avoir fortement impacté nos prédictions. Toutefois nous pouvons difficilement les améliorer par soucis de complexité. Nous avons décidé d'en arrêter là, car trop long à exécuter pour une mauvaise accuracy, dont nous ne pourrons améliorer l'apprentissage.

SVM sur les images

Après avoir appliqué des SVM sur des features extraites des réseaux de pré-entraînement, nous nous sommes aperçus que le nombre de données pris en charge par la méthode SVM pouvait être grande. Nous avons alors pensé à l'appliquer - raisonnablement - directement sut nos images. Pour ce faire, nous avons redimensionné les images en une résolution de $32 \times 32 \times 3$ pixels, que nous avons ajouté, toutes ensemble dans un même vecteur. Les images restent nettes malgré leur mise à l'échelle.

Attendu à priori : On s'attend à une accuracy autour de 30-40% minimum. Même si nous sommes suspicieux d'un SVM directement sur des images.

Résultat : On obtient une accuracy à seulement 20%. C'est bien en-deçà de toutes les autres accuracies des méthodes approchées. Cette méthode n'est pas intéressante, malgré qu'elle soit aussi efficace qu'un MLP, car d'autres applications de SVM sont bien meilleures.

8. Conclusion

Matrice de confusion

Nous avons construit une matrice de confusion de taille 260×260 (cf. Annexe 1). Elle nous permet de voir la qualité de prédiction pour chacune des espèces, et de voir avec lesquelles elles sont confondues. Une des espèces qui se trompe le plus dans la prédiction est le Nothern Goshawk, qui se méprend avec le Philippin Eagle :

Figure 15: Nothern Goshawk vs. Philippin Eagle

Dans notre code, nous avons pris 5 images mal classifiées, et avons affiché l'image originale (à droite), et une image de la classe prédite (à gauche). Les erreurs sont cohérentes, les couleurs ou les formes pouvant porter à confusion. Le Gilded Flicker a les mêmes couleurs que Red Face Warbler. La Banded Broadbill et le Helmet Vanga ont la même forme. On remarque que les oiseaux de dos, ou avec des ailes déployées sont souvent confondus.

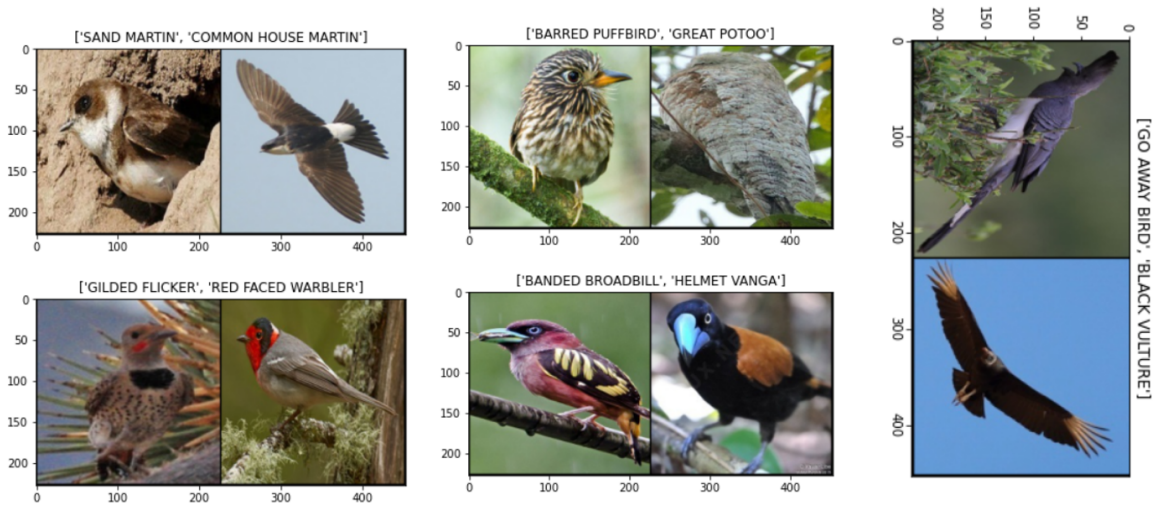

Figure 16: Aperçu de 5 erreurs de notre modèle

Clôture de l'analyse

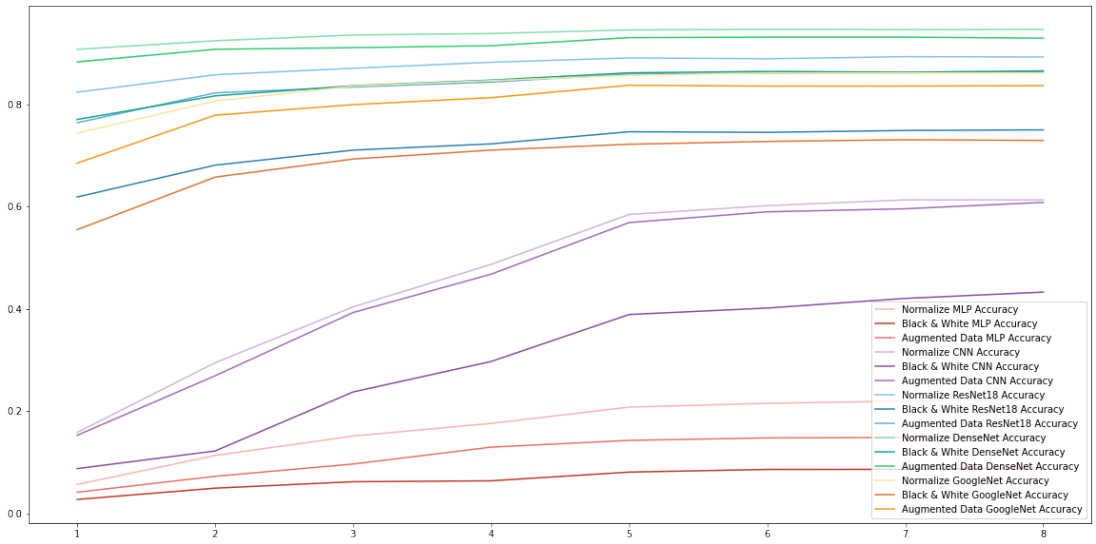

Figure 17: Comparaison graphique de nos modèles

Tout au long de ce projet, nous avons cheminé auprès des méthodes les plus réputées dans le domaine du Computer Vision. Depuis le Multi Layer Perceptron, jusqu'à l'application de méthodes de classification sur des features extraites de réseaux de Transfer Learning, nous avons tenté de construire un modèle prédictif stable adapté à nos images d'oiseaux. Prédire une classe parmi 260 est une tâche non aisée, mais finalement très accessibles si l'on s'entoure des bons procédés. Les méthodes d'apprentissages à des fins de classification, ou plus précisément, les réseaux de neurones dans leur globalité, sont des éléments forts de notre code.

Parmi l'ensemble des techniques que nous avons appliquées sur notre dataset, nous en avons trouvé une multitude qui sont adaptées à nos données (cf. Annexe 2). Par adaptation, nous voulons signifier des techniques qui supportent 40 000 données d'images volumineuses, et qui les classifient parmi 260 espèces différentes. Sur nombreuses d'entre elles nous observons de très bonnes accuracies. Il existe donc au moins un bon modèle qui classifie nos oiseaux. En termes d'accuracy, les meilleurs modèles sont les suivants, et les seuls dépassant 90% :

DenseNet Normalisé	SVM DenseNet	DenseNet Augmenté	SGD DenseNet
94.47%	93.7%	93.06%	92.8%

En termes de rapidité d'apprentissage, le classifieur KNN sur les features extraites de DenseNet est de loin le meilleur.

Toutefois, le modèle qui nous semble le plus conforme aux données est le réseau de Transfer Learning DenseNet sur les images normalisées et augmentées. Il fait partie du top 3 des meilleures accuracies. Mais le point important qui nous fait pencher en sa faveur est le fait que les images soient augmentées. Dans la mesure où l'objectif est de reconnaître un oiseau à partir d'une image, le modèle doit pouvoir prendre des images non parfaites. Si une application est développée avec notre code, et qu'une personne rentre une photo non cadrée, avec un oiseau étant travers, que l'oiseau regarde au Nord ou au Sud, il aura de meilleures chances d'avoir une classification correcte. C'est le modèle qui parait le plus robuste.

Limites de notre data set et de notre code

Un aspect que nous aurions aimé travailler lors de ce projet est le nettoyage de données et la préparation des images avant leur étude. Toutes les images étaient préalablement centrées, nettes, les oiseaux directement identifiables. Aucune étape de ce type n'a été nécessaire. Dans la mesure où un homme souhaiterait prendre un oiseau en photo lors de sa balade en forêt amazonienne, nous ne sommes pas sûrs d'une aussi bonne prédiction que celles qu'on a pu trouver tout au long de ce projet. Effectivement, l'oiseau risque d'être pris en photo furtivement, puis de s'envoler aussi vite. En résulterait une photo où l'oiseau ne serait pas centré, serait de dos, en train de s'envoler, flou, etc. Nous pointons du doigt le manque de réalisme de ce dataset, qui toutefois est excellemment bien préparé. Ce détail nous a toutefois permis de nous concentrer sur les aspects fondamentaux du Deep Learning, comme les réseaux de neurones en général, le Transfer Learning et l'extraction de features.

Au début du rendu, nous avons fait remarquer que le dataset était composé à 80% de mâles et à 20% de femelles, ces dernières étant beaucoup moins colorées. Une meilleure répartition du genre dans les images aurait probablement amélioré notre précision de prédiction globale.

Beaucoup d'images d'oiseaux de dos, de profil, ou avec des ailes déployées sont mal classifiées. On suppose que ces images affectent l'accuracy. En regardant nos images plus en détails, nous avons remarqué que certaines classes contiennent beaucoup plus d'images de vols d'oiseaux que d'autres. Un oiseau en vol aura donc beaucoup plus de chances d'être assimilé à une espèce où il y a beaucoup d'images d'oiseaux en vol. C'est aussi le cas pour les oiseaux de dos, et les oiseaux chantants. A terme, une solution pour palier à ce problème serait de rajouter des images de ces angles-là dans les classes déficitaires. Finalement, il faudrait équilibrer les classes.

Remarques

Lors de l'augmentation de données, et de la transformation des images en noir et blanc, nous n'avons pas fait d'extraction de features sur ces données. Comme nous avions les résultats pour le Transfer Learning sur les données augmentées, non augmentées, et sur les extractions de features sur les données non augmentées, nous pouvions en déduire le résultat à priori sur l'extraction de features des données augmentées. Si l'accuracy était moins bonne pour les données non augmentées qu'avec augmentation pour le Transfer Learning, alors nous avons supposé que les résultats seraient moins bons pour l'extraction de features sur les données augmentées comparé aux données non augmentées.

Le temps ne nous l'aurait de toute façon pas permis, mais il aurait été très intéressant de croiser les données des oiseaux avec leur habitat naturel, la partie du monde où peut les trouver, ou encore leur cri.

Ce que nous avons appris

Nous avons appris énormément de choses au cours de ce projet, tant dans la rigueur à adopter en Machine Learning et l'adaptabilité requise, qu'en approches de Deep Learning.

En termes d'adaptabilité, nous avons dû nous soumettre aux exigences qu'impliquaient la manipulation de données volumineuses. En premier nous pensons à l'utilisation d'un GPU à la place du CPU habituel. Nous avons perdu du temps aux premiers abords à vouloir lancer notre code sur le CPU. Dans la mesure où nous ne manipulons qu'une trentaine de milliers d'images, et que des entreprises comme les GAFA en gèrent massivement plus, nous pensions que nous n'aurions pas tant de problèmes de temps d'exécution. Nous nous sommes rendus compte que ce n'était pas si évident. Malgré l'utilisation du GPU, le temps d'exécution de certaines cellules de codes reste long, pouvant parfois aller jusqu'à quelques heures (6H pour le Grid Search, et une moyenne de 40 minutes pour l'apprentissage). Nous avons dû remodeler nos données pour rendre la complexité

correcte. Cela nous a permis de mieux appréhender les conséquences et les difficultés qu'engendrent la manipulation du "Big Data", et encore notre dataset est léger, compte tenu des données qu'une entreprise comme Google pourrait détenir...

Nous avons bien sûr découvert de nombreux modèles de réseaux de neurones, et des méthodes plus poussées comme l'extraction de features. Nous avons beaucoup appris à ces sujets tout en parvenant à appliquer les méthodes vues en classe.

Notre avis

Nous avons beaucoup aimé travailler sur ce sujet, qui nous a permit de mettre en pratique nos connaissances, et de concrétiser nos cours de Machine Learning. Nous avions le bagage théorique et les bases pratiques, et nous avons la sensation d'avoir encore plus évolué à travers ce projet. Nous avons appris énormément de choses.

Traiter une grande quantité de données et de classes a été très stimulant et formateur, d'autant plus que ces données sont des images. Ce premier pas dans le domaine du Deep Learning nous donne envie de nous y pencher encore plus profondément.

ANNEXE 1 :

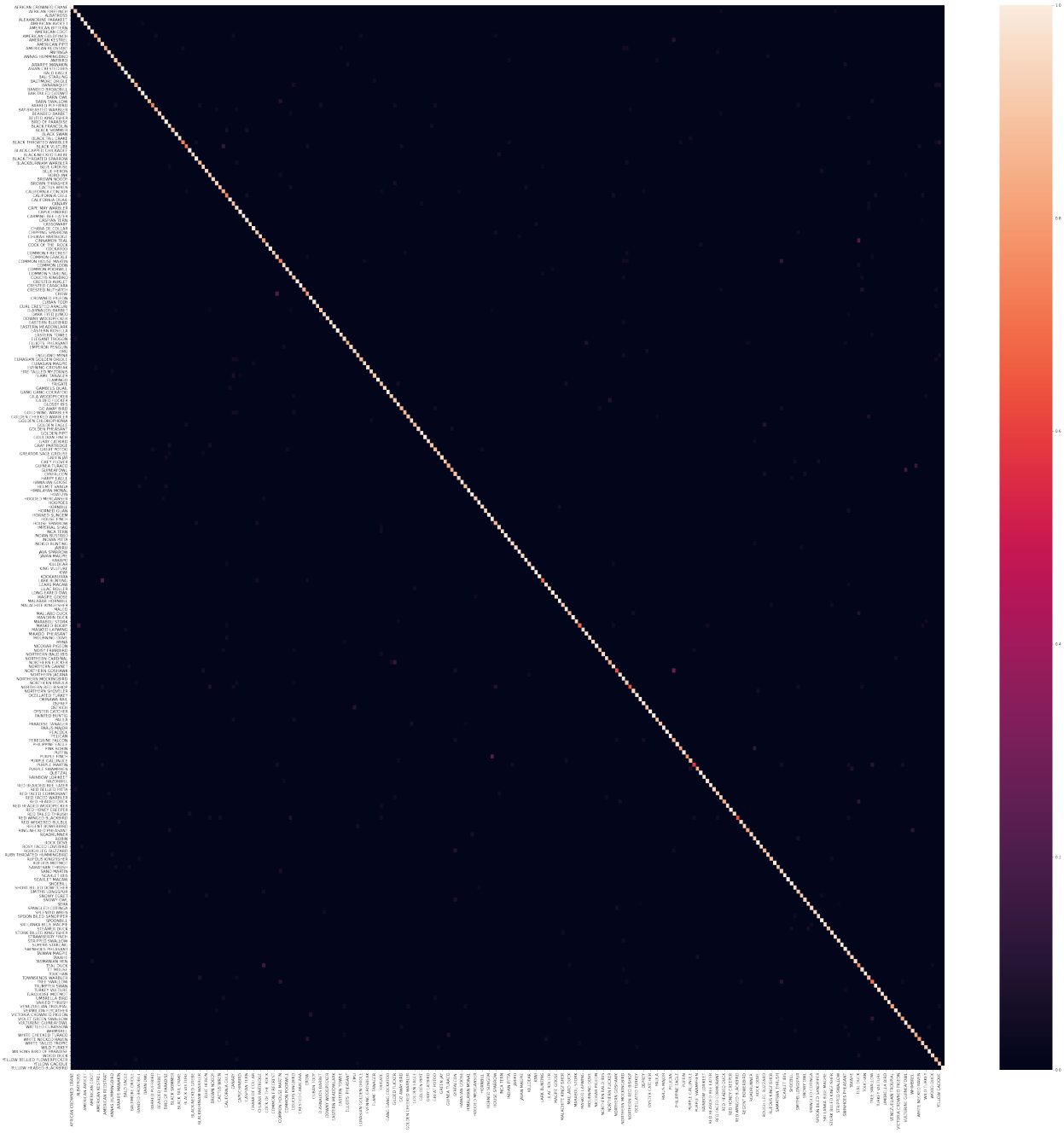

Figure 18: Matrice de confusion

ANNEXE 2 :

	Accuracy
DenseNet	94.47%
SVM DenseNet	93.70%
DenseNet Augmented	93.46%
SGD DenseNet	92.87%
ResNet	89.54%
SVM ResNet	88.94%
KNN DenseNet	88.25%
ResNet Augmented	86.38%
DenseNet N/B	85.89%
SGD ResNet	84.94%
GoogLeNet	84.91%
GoogLeNet Augmented	82.46%
SVM GoogLeNet	79.06%
KNN ResNet	78.58%
Random Forest DenseNet	75.95%
SGD GoogLeNet	74.68%
ResNet N/B	74.29%
GoogLeNet N/B	72.36%
Random Forest ResNet	68.56%
KNN GoogLeNet	63.50%
CNN	62.90%
CNN Augmented	61.49%
Random Forest GoogLeNet	56.19%
CNN N/B	41.33%
MLP	21.55%
MLP Augmented	15.04%
MLP N/B	8.32%

Table 1: Accuracy par modèle (trié par accuracy)