INSA Toulouse

Projet Intégrateur
Thales Alenia Space

Auteurs:
Duc Hau NGUYEN
Cindy PONIDJEM
Anaïs RABARY
Ezechiel TAPE

January 31, 2019

Contents

1 Contexte et objectifs

Thalès Alenia Space est l'une des entreprises du groupe Thales dont les activités gravitent principalement autour de "l'espace", et plus précisément des satellites de communication. À cet effet, elle produit et gère une grande quantité d'images satellite. Sur ces images, plusieurs opérations sont effectuées en vue de la détection d'objets (sols, nuages, cibles spécifiques comme des avions, des navires,...). La taille des images prises par les satellites étant grande (parfois plus de 250 Giga Octet), il est, alors, nécessaire de mettre en place un système distribué à la fois pour leur stockage et pour les diverses analyses souhaitées. De plus, ces analyses doivent pouvoir tirer profit des différentes techniques d'apprentissage automatique.

C'est dans ce cadre général que s'inscrit le projet intégrateur Thalès. L'objectif dudit projet est, d'une part, de créer un système de stockage distribué pour l'hébergement des images et, d'autre part, d'implémenter certaines méthodes d'apprentissage automatique pour leur traitement.

Pour mener à bien le projet, des choix d'outils tels que Minio, système de stockage distribué de type objet, et ElasticSearch, puissant moteur de recherche et d'indexation, sont imposés. L'utilisation de Spark, outil de calcul distribué, est recommandé.

Les grandes lignes du projet sont :

- créer une base de données de type Minio;
- créer une base de type Elasticsearch pour permettre l'indexation et la recherche d'informations à partir de méta-données des images;
- développer des algorithmes de classification pour enrichir les méta-données des images, en utilisant un système de calcul distribué comme Spark.

Dans ce rapport, est présenté, en premier lieu, la démarche que nous avons adoptée pour le projet et les différentes pistes explorées, puis les fonctionnalités implémentées et les résultats obtenus.

2 Démarche projet et pistes explorées

Dans cette section, nous avons choisi de présenter nos pistes de recherche personnelles. Il est à noter que, dans le cadre de notre management d'équipe, nous avons mis en commun nos recherches à chaque début de séance et tous les membres de l'équipe ont travaillé sur tous les sujets abordés (autant du côté architectural que du côté de l'apprentissage). Les sujets étudiés ont été introduits par l'un des membres de l'équipe, d'où l'organisation de cette partie.
Bien que tous les sujets étudiés n'aient pas porté leurs fruits, il est important d'en garder note pour des projets futurs.

2.1 Cindy PONIDJEM

Dans l'objectif de créer un système de stockage distribué, il nous semblait pertinent de partir d'une infrastructure simple et fonctionnelle, et de l'améliorer au fur et à mesure. Notre infrastructure de base comportait ainsi un cluster de 4 instances Minio pour le stockage et un noeud Elasticsearch unique, leurs docker-compose étant disponible sur internet.

2.1.1 Upload: Du script en local à un service dédié

Le premier travail à été de tester notre infrastructure en chargeant des images à titre d'exemple dans Minio et en les indexant dans Elasticsearch. Pour ce faire, nous avons choisi d'utiliser leurs clients Python respectifs. Nous avons donc un script Python qui se charge d'enregistrer temporairement chaque image du dataset au format .npy, puis de charger ce fichier dans Minio et de l'indexer avec son label sous forme de tableau et sous forme textuelle.

Dans un premier temps, ce script s'exécutait en local sur la machine hôte. Par souci de simplicité lors du déploiement, nous avons voulu packager ce script pour qu'il s'exécute dans un service au même titre que Minio et Elasticsearch. Nous avons donc construit une image docker, se basant sur une image Alpine, sur laquelle on a ajouté le dataset, le script et installé toutes les dépendances nécessaires. Nous avons ensuite ajouté cette image à notre docker-compose.

On peut maintenant envisager de stocker le dataset en ligne, sur un serveur ftp par exemple, qui sera récupéré par notre service à l'exécution. Cela permettrait de réduire la taille de notre image docker.

2.1.2 Elasticsearch: D'un mapping dynamique à un mapping défini

Lors de nos recherches pendant les premières séances, nous avons pu nous familiariser avec Elasticsearh et son fonctionnement, notamment à l'aide du chapitre qui lui est réservé dans l'ouvrage Modern Big Data Processing with Hadoop [10]. Un index est un ensemble de documents JSON. Un mapping correspond à la définition structurelle d'un index, précisant ainsi les champs qu'il contient, et si ces champs sont interrogeables ou non. Elasticsearch supporte le mapping dynamique : On peut créer un index sans son mapping, lors de l'envoi d'un document JSON pour l'indexation, Elasticsearch définit la structure de chaque champ et le configure comme interrogeable.

Une première version du script d'upload créait des indices dynamiques, et donc sans mapping. Dans la version finale, on définit le mapping des indices train et test en leur assignant des champs différents. De cette façon, Elasticsearch sera plus performant lorsqu'il recensera de très nombreuses images (d'après la littérature). De plus, seuls les champs correspondant à un label sous forme textuelle sont interrogeables.

Nous n'avons pas eu le temps de chiffrer l'écart de performance, entre un mapping dynamique et un mapping défini à la création de l'index, lors de la phase d'indexation.

2.2 Ezechiel TAPE

2.2.1 Spark-Minio : Lecture des données

L'un des avantages d'utiliser Spark est le RDD(Resilient Distributed Dataset)[15]. Afin de tirer profit de cette fonctionnalité, nous avons cherché à trouver le meilleur moyen pour télécharger les données depuis les serveurs Minio en des RDDs. L'utilisation du client Minio demandait de stocker temporairement les données téléchargées dans une structure de données puis de les convertir en des RDDs. Une telle procédure s'avère être coûteuse ! La solution mise en avant dans nos travaux est d'utiliser la capacité de Spark et de Minio à supporter le protocole S3. En effet, certaines fonctions de Spark permettent de charger des données, depuis un serveur de stockage supportant le protocole S3, directement en des RDDs.

2.2.2 Communication entre Spark et Elasticsearch

Concernant la communication entre Spark et Elasticsearch(ES), le client python d'ES fut préféré au vu du temps disponible. Simple d'usage, il offre une panoplie de fonctionnalités permettant de tirer profit de la puissance d'ES. Toutefois, nous aurions pu utiliser le connecteur Elasticsearch-Hadoop[2]. Ce connecteur permet la lecture ou l'écriture de données vers Elasticsearch en utilisant des RDDs. À priori, une telle utilisation permettrait un gain de temps.[3]

2.2.3 Elasticsearch en mode cluster

Comme indiqué auparavant par Cindy, notre architecture de base contenait un seul noeud ES. Avoir un seul noeud ES exposait notre infrastructure à des problèmes de scalabilité, en cas de manipulation de plusieurs images, et de tolérance aux pannes, puisque nous avions un seul noeud de stockage ES. Ainsi, nous avons opté pour ES en mode cluster. Dans un cluster ES, les données sont divisées en des shards, puis les shards sont répliquées au sein des noeuds du cluster.

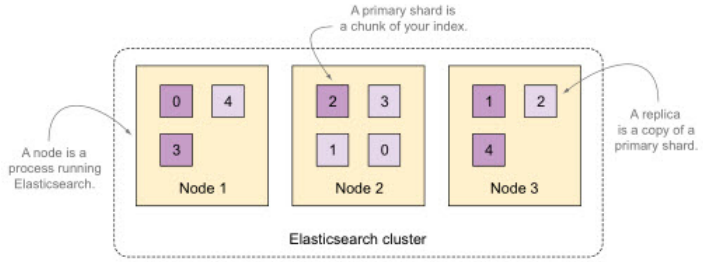

Figure 1: Exemple de fonctionnement d'un cluster à 3 noeuds [4]

Notre cluster ES est constitué de 3 noeuds, soit d'un noeud maître, soit de 2 noeuds esclaves. 3 noeuds est le nombre minimum pour prémunir contre une situation de split-brain dans notre cluster ES [14]. Un split-brain se produit lorsqu'un cluster se scinde, dû à un problème de communication, en un nombre de partitions indépendantes ne pouvant communiquer entre eux. Pour un cluster ES, une telle situation se présente lorsqu'au moins 2 noeuds s'érigent comme noeud maître.

2.2.4 Visualisation des données sur Elasticsearch

Bien que n'étant pas indiqué dans le cahier de charges, l'ajout d'un outil de visualisation et d'exploration des données sur Elasticsearch confère un atout supplémentaire. Un tel outil permettra de visualiser plus aisément les différentes métadonnées des images et d'effectuer facilement des requêtes sur la base de données ES. À cet effet, nous proposons Kibana, qui est un accessoire logiciel s'interfaçant agréablement avec ES en mode standalone ou cluster.

2.3 Anaïs RABARY

2.3.1 Affichage et traitement des images

Le premier travail a été de visualiser notre dataset pour faire connaissance avec nos données. En regardant les données des pixels RGB de différentes images, nous nous sommes rendu compte que les images n'étaient pas codées entre 0 et 255 ni entre 0 et 1. Un affichage sans traitement nous donne des images noires. Le codage des images n'est pas clairement défini.

Nous avons donc réalisé différents traitements pour essayer d'avoir un meilleur affichage. Une première technique consister à multiplier toutes les valeurs des pixels par un coefficient de 10. Un 2eme traitement, plus adéquat, récupère la valeur maximale des pixels et divise les pixels par cette valeur max. Ce dernier traitement fourni une image avec des couleurs plus équilibrées.

Figure 2: Image non traîtée puis égalisée d'un territoire agricole

2.3.2 Algorithmes de Segmentation

Avant même de regarder en détails les images à notre disposition, le sujet nous laissait penser à un problème de segmentation puis de classification d'image. Nous nous sommes donc documentés sur des problèmes de segmentation [13].
En parcourant différentes méthodes de segmentation, nous avons trouvé une méthode non supervisée sur Scikit Learn Image. C'est la méthode de Felzenszwalb qui segmente l'image et on obtient donc les contours.[7] Nous n'avons pas poursuivi les recherches plus loin car nous n'avons pas les données nécessaires pour faire de la segmentation d'image. Effectivement, nos images sont labélisées à 100%. Nous avons un label pour toute l'image. Pour faire de la segmentation notamment avec des RNN, il aurait fallu avoir un masque de labels (avec des portions de l'image ayant différents labels).

Ci-après une image illustrant les contours trouvés par la méthode de Felzenszwalb.

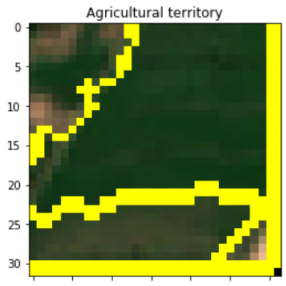

Figure 3: Image d'un territoire agricole avec les contours détectés

2.3.3 CNN pour la classification des images

Une bonne partie de l'étude a été de déterminer quel algorithme de Classification utiliser. Les recherches, accompagnées du code peuvent se retrouver dans le notebook DataMining_TAS.ipyn joint en annexe.

Nous avons d'abord passé en revue les algorithmes "classiques" de classifications comme Decision Tree, KNN, SVM, ... Mais nous avons privilégié les réseaux de neurones. En effet, même si l'apprentissage des poids est coûteux, la prédiction est quant à elle quasi-instantanée. Plutôt que de faire des réseaux multi-couches, nous avons préféré les CNN qui sont souvent utilisés pour des problèmes de classification d'image.[6]

Les fruits des recherches sont exposés dans la partie 3 de ce rapport car nous les avons mis en pratique dans le projet.

2.3.4 Spark et Keras

L'objectif de ce projet étant d'avoir une architecture et un pipeline complet et pouvant être distribué, il faut aussi prévoir la distribution de la partie apprentissage et/ou prédiction des données. Dans un premier temps, nous nous sommes concentrés uniquement sur la distribution de la partie prédiction. Il est aussi possible de distribuer l'apprentissage du modèle Keras sur Spark notamment pour faire la validation du modèle avec la méthode de cross-validation pour ne garder que le meilleur modèle.

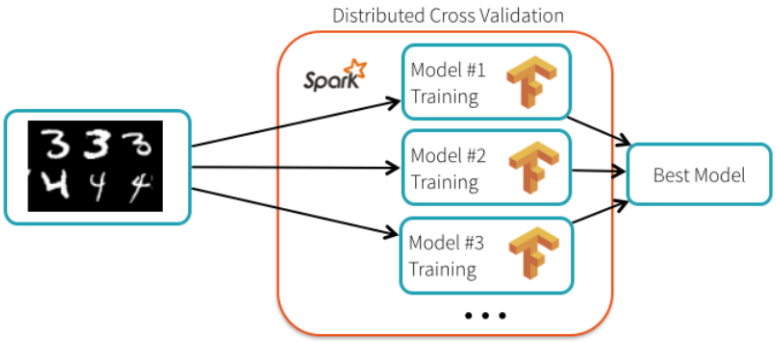

Figure 4: Schéma pour la validation distribuer d'un modèle

Spark est idéal car il nous permet de faire des calculs sur GPU et des calculs distribués sur d'autres machines. Dans le cadre du projet, nous cherchions à combiner Keras avec Spark. Nous avons testé différents outils. Nous avons choisi Elephas qui permet d'utiliser l'API pyspark en wrappant le modèle keras et en stockant les données sous forme de RDD. Le RDD sera partitionné en autant de parties que de coeurs disponibles pour le calcul. Ainsi on peut directement charger le modèle Keras déjà entraîné et l'envelopper dans un modèle Spark, grâce à Elephas.

Avec les conditions techniques de notre projet, nous ne disposons pas de machines dédiées pour faire ces calculs distribués. Nous avons donc effectué nos calculs sur des machines virtuelles. La performance n'a donc pas pu être évaluée correctement.

2.4 Duc Hau NGUYEN

2.4.1 Apprentissage

L'étude sur l'apprentissage a été menée dans un premier temps sur des méthodes de segmentation sémantique. Plusieurs cours théoriques présentant ce sujet ont été étudiés en diagonale [13].
Aussi, d'autres méthodes ont été proposées dans les articles de Review [5]. Lors de l'implémentation, nous

avons rencontré plusieurs difficultés techniques liées aux contraintes des machines à l'INSA.

Nous avons modifié la ligne directrice de nos recherches pour revenir sur de la classification "classique" d'image. Ce changement a été imposé par les données fournies car elles ne sont pas adaptées pour la segmentation (aucun masque d'objet n'a été fourni mais des labels de classes à la place). Le code fourni dans le Github du projet est inspiré par le tutoriel de Keras [8]

2.4.2 Déploiment Keras dans Spark

Lorsque nous avons eu les premiers résultats avec Keras (réussit à entraîner et prédire), nous avons cherché à déployer notre modèle Keras dans le cluster de calcul Spark. Le but est de tirer profit de l'apprentissage en ligne.

Lors de cette recherche, plusieurs lectures [1] [12] proposent la solution la plus simple: utilisant la librairie SystemML (proposé par IBM). Cependant cette librairie contient un bug dans son code source. Cette étude a alors été abandonnée en acceptant la méthode utilisant Elephas.

2.4.3 Intégration Spark - Minio

A cette étape, il nous reste à établir un lien entre le cluster de calcul et le cluster de stockage.

Grâce à l'étude préalable faite par Anaïs [11], nous avons pu réaliser l'intégration entre Spark et Minio dans le sens où Spark récupère les données depuis le cluster Minio (en tant que RDD) pour pouvoir par la suite entraîner le modèle et/ou prédire la classification des images. Cette intégration montre l'intérêt majeur du système distribué : (1) On peut bénéficier de l'avantage du stockage distribué et (2) on utilise en plus la puissance de calcul pour cette partie.

Le détail de configuration de Spark afin de pouvoir communiquer les clusters de Minio est disponible dans le readme de notre dépôt Git.

2.4.4 Prédiction en ligne

Cette dernière partie consiste à finir notre pipeline : Lorsque l'on a fini d'entraîner un modèle, il faut pouvoir utiliser la puissance de ce modèle. Cette partie a été réalisée en collaboration avec Cindy PONIDJEM et Anaïs RABARY afin d'orchestrer les flux de données.

La réalisation de cette dernière partie résulte en démo vidéo pendant la présentation.

3 Fonctionnalités implémentées et résultats

3.1 Minio et Elasticsearch

3.1.1 Architecture

Notre architecture finale comprend Minio en mode distribué et Elasticsearch en mode cluster.

En ce qui concerne Minio, on a 4 instances qui possèdent chacune leur propre espace de stockage mais forment un seul et même serveur de stockage objet. C'est à dire qu'il y a une couche d'abstraction : on ne sait pas où se trouve nos données dans le cluster. On peut utiliser une unique instance comme point d'entrée pour charger nos images, c'est Minio qui se charge de traiter et de répartir les données sur les différentes instances, avec la redondance nécessaire pour assurer la protection des données (Erasure coding).

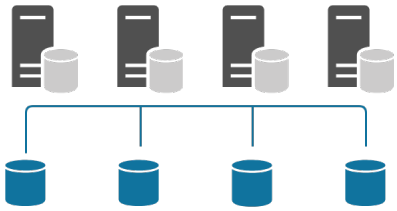

Figure 5: Schéma de notre architecture côté Minio

Côté Elasticsearch, pour les raisons citées plus haut, on a un cluster de trois nœuds, configurés de la manière suivante :

- 2 noeuds contiennent réellement les données d'indexation
- 3 nœuds sont éligibles master
- 2 nœuds éligibles master doivent être visible pour pouvoir former un cluster

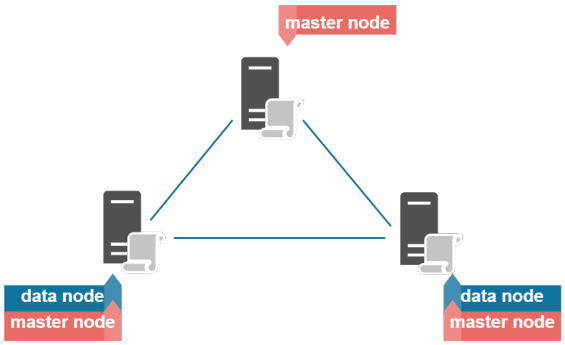

Figure 6: Schéma de notre architecture côté Elasticsearch

3.1.2 Déploiement Docker

Docker Compose permet la définition et l'exécution d'une application impliquant plusieurs containers Docker, aussi appelés services dans ce contexte.

C'est cet outil qui nous permet de déployer nos instances Minio, nos nœuds Elasticsearch et le service chargé de l'upload. En ajustant notre fichier .yaml, nous avons pu garrantir :

- la persistance des données même en cas de suppression des containers, en utilisant les volumes Docker

- la communication entre nos containers
- l'accès depuis la machine hôte en mappant les ports

Pour plus d'information sur comment déployer l'infrastructure, veuillez-vous référer au readme sur notre projet Github.

3.2 Apprentissage

Nous avons choisi de former nous-même notre propre réseau de neurones CNN avant d'utiliser un modèle plus complexe et déjà réalisé. CNN prend son sens dans la classification des images car il est "translation-invariant", c'est à dire insensible à des translations de l'image.
Ci-après, on se propose de décrire succinctement la constitution de notre premier modèle, avant de décrire le processus établi pour utiliser un réseau plus évolué et enfin comparer les résultats. Une description plus détaillée se trouve dans un notebook en annexe.

3.2.1 Modèle simple

Le modèle que nous avons construit attend une image de $32 * 32 * 3$ en entrée, soit une image de $32 * 32$ pixels RGB. En sortie, il fournit un array de longueur 5 en guise de label, décrivant la classification.
Notre modèle CNN est constitué de 7 couches cachées que l'on peut séparer en 2 parties :

- la partie d'extraction de caractéristiques (couches 1 à 6)
- la partie classification (couche 7)

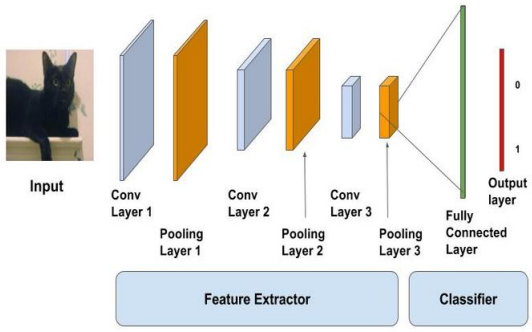

Figure 7: Schéma du réseau CNN simple

La 1ere partie est constituée d'une alternance de couches convolutionnelles (couches denses) et de couches de max-pooling. Les couches convolutionnelles sont celles qui permettent de détecter les features. A chaque fois que l'on rajoute une couche, le nombre de paramètres à déterminer augmente fortement. C'est là qu'interviennent les couches de pooling dont le but est justement de diminuer la taille de l'espace et donc le nombre de paramètres. Dans notre modèle, on utilise des couches de max-pooling qui sont souvent utilisées.

La 7eme couche, formant la 2ème partie, est une couche 100% connectée agissant comme classifier. Si sa fonction d'activation est "softmax", elle nous donne le label de l'image. Si c'est "sigmoid", elle nous donne la probabilité que l'image soit ce label.

3.2.2 Transfer learning

Dans l'idée de bénéficier un réseau plus profond et déjà entraîné dans les challenges d'opendata, on cherche à appliquer la technique de **transfer learning**.

Cette technique se résume en 3 étapes :

- Reconstituer le modèle de base, dont l'architecture est lourde mais performante en précision.
- Adapter cette architecture à notre problème en modifiant la couche sortante.
- "Fine-tune" ce nouveau réseau pour nos propres données, mais en modifiant seulement la couche ajoutée à l'entrée du modèle.

Pour le choix d'un réseau existant, nous avons choisis le modèle **DenseNet121** pour simplifier le développement car déjà existant dans Keras, mais aussi pour trouver un compromis entre le temps d'apprentissage et l'accuracy car, ayant un temps limité, entraîner un réseau plus lourd n'assure pas forcément un meilleur résultat. Notre choix est justifié par le tableau extrait depuis le site de Keras [9] :

Model	Size	Top-1 Accuracy	Parameters
Xception	88 MB	0.790	22,910,480
VGG16	528 MB	0.713	138,357,544
VGG19	549 MB	0.713	143,667,240
ResNet50	99 MB	0.749	25,636,712
InceptionV3	92 MB	0.779	23,851,784
InceptionResNetV2	215 MB	0.803	55,873,736
MobileNet	16 MB	0.704	4,253,864
MobileNetV2	14 MB	0.713	3,538,984
DenseNet121	14 MB	**0.713**	**3,538,984**
DenseNet169	57 MB	0.762	3,538,984
DenseNet201	80 MB	0.773	20,242,984

L'idée maintenant est d'ajouter une nouvelle couche sur cette architecture permettant au réseau de reconnaître notre image à l'entrée. Dans le notebook, c'est une seule **Fully Conntected Layer** dont la fonction d'activation est **ReLU**. Puis on entraîne seulement sur cette dernière couche (on modifie donc seulement les poids de cette couche). Cela permet de gagner en temps de calcul. Les poids de **DenseNet121** ont été optimisés pour reconnaître les features nécessaires, et donc un nouvel entraînement n'est plus nécessaire. Cette technique a prouvé un meilleur résultat en précision, dans un temps plus court. De plus, **DenseNet** réduit le nombre de paramètres, ce qui nous permet d'éviter des problèmes de surapprentissage (overfitting).

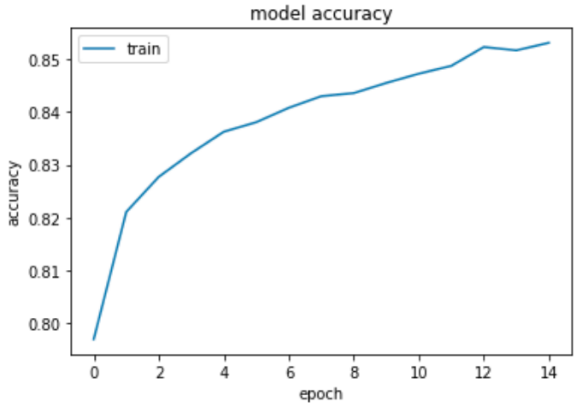

Figure 8: Vitesse de convergence avec le transfer learning déjà optimisé, on gagne du temps de calcul

3.2.3 Résultats

La performance des 2 modèles est évaluée sous les mêmes conditions :

- Backend Tensorflow
- Entraînement sur **163989** données prétraitées (les images noires sont virées)
- Validation sur **41011** données prétraitées

- Même environnement: **Ubuntu 18.0.1**, CPU **4 coeurs** de calculs, RAM **16 Go**

La validation a été faite en bénéficiant des données déjà séparées dans les 2 dossiers **train** et **test** (V-folds non employé). L'évaluation des performances s'est basée sur ces 2 paramètres :

- Métrique: **accuracy**
- Loss function: **categorical entropy** (nous avons des multi classes (5) et devons utiliser l'entropie par la nature des données (images).

	Modèle simple	Transfer Learning
Temps d'entraînement	18 mins 23	11 mins 23
Accuracy	72%	85%

Plusieurs tests ont été menés pour voir l'intérêt de la préparation des données. On constate qu'en enlevant les images noires, on a un meilleur résultat. Cependant, le traitement de l'égalisation des pixels n'est pas du tout pertinent pour notre apprentissage. En effet, le score chute à moins de 10%. Une évolution de notre apprentissage consisterait à faire une validation plus sérieuse du modèle, par exemple en faisant du K-fold validation.

3.3 Spark

3.3.1 Intégration dans l'architecture

Dans ce projet, **Spark** opère indépendamment du système **Hadoop**. Le détail technique est expliqué dans de dépôt Github du projet. Conceptuellement, **Hadoop** doit communiquer avec les instances de **Minio** via leur endpoint **localhost:9001**. Dans son fichier configuration, **Hadoop** remplace tous les *hdfs* par le style de *s3a*.
Le fait que la communication entre **Spark** et **Minio** passe par la façade de cluster **Minio** permet un couplage faible entre ces 2 clusters. L'administrateur de la base de données a plus de flexibilité d'ochestrer les instances **Minio** sans interrompre l'application tournée sous **Spark**.

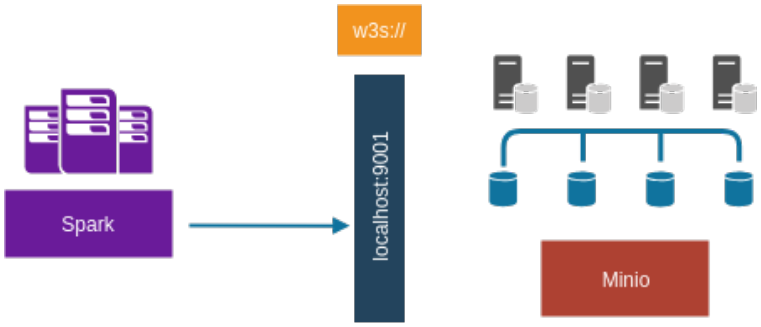

Figure 9: Schéma de notre architecture côté Minio

La communication entre **Spark** et **Elasticsearch** est en revanche relativement plus simple, comme ce moteur de recherche expose son service en API Rest (communication via requête HTTP avec des verbes prédéfinis GET/POST/DELETE/...). Donc pour cette partie, aucune configuration n'est nécessaire. Pour la visibilité du code, nous utilisons dans le code Python librairie du client **Elasticsearch**.

3.3.2 Intégration avec Keras

Dans le scénario que nous avons choisi pour ce projet, la prédiction de la classification des images se fait de façon distribuée. Nous avons donc importé sur Spark notre modèle CNN de Keras déjà entraîné. Au préalable, il faut avoir configuré Spark pour fonctionner avec Elephas.
On place donc les images à classifier dans un RDD que l'on donne à Spark qui effectue la classification sur son cluster puis on récupère les résultats de la classification

4 Conclusion

Pour conclure, ce projet nous a beaucoup apporté.

Il nous a à la fois permis de réinvestir des concepts abordés tout au long du semestre que d'apprendre de nouvelles choses. En effet, on a pu se familiariser avec des technologies récentes et tout à fait adaptées à notre domaine. On a également pu élargir notre connaissance conceptuelle en apprentissage automatique.

On peut aussi souligner l'apport concernant le travail en équipe, puisque nous avons mis en place un partage d'informations entre les membres du groupe ainsi qu'une répartition des tâches adaptée aux affinités de chacun.

References

[1] Coursera. Run keras models in parallel on apache spark using apache systemml.

[2] Elastic. Elasticsearch-hadoop: Apache spark suport.

[3] Elastic. Elasticsearch-hadoop: performance considerations.

[4] Radu Gheorghe, Matthew Lee Hinman, and Roy Russo. *Elasticsearch in Action*. https://www.amazon.com/Elasticsearch-Action-Radu-Gheorghe/dp/1617291625, 2015.

[5] Yanming Guo, Yu Liu, Theodoros Georgiou, and Michael S. Lew. A review of semantic segmentation using deep neural networks. *International journal of Multimedia Information Retrieval*, 7, October 2017.

[6] Vikas Gupta. Image classification using convolutional neural networks in keras.

[7] Scikit Image. Comparison of segmentation and superpixel algorithms.

[8] keras. Cs231n: Convolutional neural networks for visual recognition.

[9] Keras. Documentation for individual models.

[10] V. Naresh Kumar and Prashant Shindgikar. *Modern Big Data Processing with Hadoop*. Packt Publishing, 2018.

[11] Minio. Cookbook apache spark with minio server.

[12] Niloy Purkait. How to train your neural networks in parallel with keras and apache spark. *Medium*, October 2018.

[13] Stanford University. Cs231n: Convolutional neural networks for visual recognition.

[14] Adam Vanderbush. Avoiding the split brain problem in elasticsearch. *Qbox Blog*, June 2017.

[15] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. *NSDI*, 2012.