INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES TOULOUSE

Université Fédérale Toulouse Midi-Pyrénées

5e année Systèmes Distribués & Big Data

Rapport de Projet Intégrateur

Reconnaissance d'images satellites dans une infrastructure distribuée

Adrian MEGA, Hao HU, Mohamed EL FILALI

2018-2019

Table des matières

Remerciements

Nous souhaitons remercier les contributeurs de ce projet qui nous ont permis de le réaliser dans le cadre de notre formation et ainsi de mettre en application nos compétences.

Merci à Christophe GOUGUENHEIM, travaillant chez Thalès Alenia Space, pour nous avoir proposé ce sujet de projet intégrateur et pour avoir répondu à nos questions.

Merci à notre responsable de spécialité Marie-José HUGUET pour nous avoir encadré sur ce projet ainsi que sur tous les autres projets de cette 5e année en spécialité SDBD.

Merci également aux autres encadrants de ce projet intégrateur, Sami YANGUI et Mohamed SIALA, pour leur soutien durant nos heures de travail au GEI.

Et enfin merci aux autres étudiants de notre classe pour nos multiples échanges sur ce projet. Seul on va plus vite, ensemble on va plus loin !

Contexte et Objectifs

Le projet intégrateur de 5e année à l'INSA Toulouse doit rassembler toutes les connaissances acquises durant le dernier semestre d'études afin de les mettre en application. Dans le cadre de la spécialité Systèmes Distribués et Big Data, Thales Alenia Space nous a proposé son projet de reconnaissance d'images satellite via une infrastructure distribuée.

Une grande quantité d'images satellite sont générées et traitées par Thalès Alenia Space. Un certain nombre de calculs sont appliqués sur ces images. La taille des images réelles prises par les satellites (plus de 250 Go) nécessite de mettre un place un système distribué à la fois pour leur stockage et pour les diverses analyses souhaitées.

En termes d'analyses d'images, les deux pistes privilégiées sont d'une part la détection de sols (champs, océan, forêt, zone urbaine, …) et d'autre part la détection de cibles (navires, éoliennes, …). Ces analyses peuvent exploiter diverses techniques d'apprentissage automatique.

L'objectif du projet est d'une part de de créer un système distribué de stockage et d'autre part d'analyser ces images pour leur ajouter de la valeur. Le projet inclus les étapes suivantes :

- Création d'une base de données objet (avec Minio)
- Création d'une base d'indexation et de recherche d'information (avec Elasticsearch)
- Développement d'algorithmes Python de classification pour enrichir les métadonnées des images via un système de calcul distribué (avec Spark)

La restitution de ces développements se fera sous la forme de fichiers Docker et Docker Compose. L'ensemble du code sera déposé et rendu disponible sur un dépôt Git.

Les images fournies sont des images de taille réduite pour permettre leur exploitation dans un l'environnement de travail. Une évaluation du passage à l'échelle pourra être envisagée en fonction des premiers résultats obtenus.

Ce rapport a pour but de présenter l'infrastructure distribuée mise en place, les méthodes de reconnaissance d'images développées, la qualité de l'approche et une mise en perspective de l'application. Une soutenance orale viendra également accompagner la restitution du travail effectué.

Fonctionnalités Implémentées

Ce chapitre a pour but de faire le point sur toutes les fonctionnalités qui ont été implémentées durant la période de temps imparti pour ce projet intégrateur.

1) Architecture orientée services

Notre architecture est composée d'instances de containers chargés de traiter et organiser la donnée. Toute cette architecture est inscrite dans le Docker Compose de notre projet. Celui-ci rassemble des containers contenant les images suivantes.

NOM	NOMBRE	PORT(S)	DESCRIPTION
PORTAINER	1	9000	Cet outil d'administration nous sert à gérer tous les containers présents sur l'instance Docker locale. A partir de Portainer, nous pouvons voir l'état de fonctionnement de nos containers, les arrêter et redémarrer, regarder les journaux d'événement de chacun (logs), entrer dans le terminal d'un container, etc.
MINIO	4	9001 à 9004	Minio est un projet open source de stockage objet. Il a été nativement conçu pour les infrastructures cloud à grande échelle. C'est sur ces 4 instances que nous stockons nos données image. Le fait d'avoir plusieurs instances nous permet d'une part de gagner en efficacité de calcul, et d'autre part d'assurer la persistance des données.
SPARK (MASTER)	1	7077	Le Framework open source de calcul distribué Spark nous permet de faire du traitement de données. Cette première image Master sert à la transition de données depuis et vers les serveurs Minio via ses « workers ».
SPARK (WORKER)	4	8081 à 8084	Ces autres images Spark quant à elles sont dédiées au traitement des données images. Ce sont elles qui déposent les données sur leur instance Minio dédiée.
ELASTICSEARCH	1	9200	Elasticsearch est un référenceur de données ainsi qu'un moteur de recherche et d'analyse. C'est lui qui va indexer les données stockées sur les serveurs Minio afin qu'elles puissent être exploitées.
KIBANA	1	5601	Kibana est un greffon à Elasticsearch, servant à la visualisation des données qui y sont référencées.

2) Traitement des images

Pour le traitement des données de façon distribuée, nous utilisons le framework Spark. Pour l'utiliser, nous allons générer cinq conteneurs : un master et quatre worker. Spark assure l'enregistrement de nos données d'une façon parallèle dans les conteneurs Minio et cela en se basant sur la fonction « MAP » qui fait appel à un client Minio et un client Elasticsearch pour enregistrer chaque image dans ces derniers.

Pour la phase d'apprentissage, les images sont chargées dans les instances Minio en utilisant Spark pour assurer le parallélisme, en se basant sur les liens enregistrés dans Elasticsearch. Ensuite, à l'aide de la fonction « collecte », nous obtenons un dataset de données et labels prêts pour l'apprentissage. Une fois les données chargées, nous utilisons la donnée ainsi que le label pour un apprentissage supervisé à l'aide d'un algorithme qui manipule les réseaux de neurones. L'outil nous permettant cela est TensorFlow.

Pour aller plus loin et pour gagner en performance, nous utilisons un framework de TensorFlow qui s'appelle Keras. Ce dernier permet d'optimiser le temps de calcul en fournissant un traitement parallèle, sans oublier la liberté d'ajouter différents paramètres. Enfin pour la prédiction de labels, nous utilisons aussi le modèle Keras pour prédire la classe d'appartenance des images fournies par la suite.

Démarche

Afin de compléter notre projet, nous avons exploré différentes pistes sur chacune des fonctionnalités de notre application. Cette partie a pour but de présenter notre démarche de recherche.

1) Définition du Docker Compose et Portainer

Ce projet a été l'occasion de nous former sur l'utilisation de Docker. Entrevu dans les enseignements à l'INSA, nous n'avons jamais eu de réelle mise en pratique auparavant. Adrian ayant déjà travaillé en entreprise avec cet outil, il a été en mesure d'apporter à l'équipe son expérience avec Docker ainsi que Docker Compose.

Le premier outil ajouté au Docker Compose de notre projet fut Portainer. Bien qu'il ne réponde pas à une exigence du projet, il s'est avéré très utile durant tout le développement du projet. Véritable tableau d'administration de containers, il nous permettait de lancer des tests individuels, de s'assurer du bon fonctionnement de chaque container et de leur bonne communication. Portainer est facile d'installation car une image préconçue est rendue disponible sur les serveurs de Docker.

2) Minio et configuration de ports réseau

Minio doit nous servir à stocker les images satellites qui seront ensuite indexées par Elasticsearch. L'installation de Minio peut aussi se faire via une image des serveurs Docker déjà prête à l'emploi. Nous avons trouvé des exemples de projets incluant l'initialisation d'instances Minio sur un Docker Compose et nous nous en sommes inspiré pour notre propre configuration. En discutant avec nos enseignants, ainsi qu'avec l'intervenant de Thalès, il a été défini que pour des raisons de performance, il y aurait quatre instances Minio qui devraient tourner en parallèle. Les données seraient alors partagées entre les 4 serveurs.

Afin d'avoir une configuration cohérente, nous avons beaucoup travaillé sur l'optimisation de l'utilisation des ports afin d'accéder à chaque service. Tout en conservant la configuration de base, nous voulions simplifier au maximum les accès à chaque service. Notons qu'avec Docker, nous pouvons accéder à un service soit par une adresse IP donnée, soit par un port défini sur localhost (ou une autre adresse, il est possible de créer des réseaux autres que le réseau de base sur Docker). Nous avons choisi de mettre tous nos services sur le même serveur local de base, avec un port bien défini par service (la liste de ces ports est disponible dans la partie Fonctionnalités Implémentées de ce rapport).

Enfin, les quatre instances Minio sont capables de communiquer entre elles. Cela permet d'une part de référencer le contenu les autres instances depuis n'importe laquelle (elles ont conscience de fonctionner ensemble), d'autre part de pouvoir faire de la sauvegarde de données sur les autres instances.

3) Indexation avec Elasticsearch et Kibana

L'installation de Elasticsearch fut plus complexe que les installations précédentes. A la façon de Minio, nous avons cherché des exemples de configuration de Elasticsearch dans d'autres projets incluant un Docker Compose. La configuration trouvée ne fonctionnait jamais sur nos machines de tests. Nous avons donc fait abstraction des configurations conseillées pour nous concentrer sur les paramétrages essentiels uniquement. Après plusieurs tests, nous avons défini une configuration minimale et efficace pour notre application.

Kibana était souvent associé à Elasticsearch lorsque nous faisions nos recherches sur l'installation de ce dernier. Après avoir étudié son utilisation, nous avons jugé intéressant de l'inclure dans l'architecture de notre projet, afin de visionner les données indexées. Pour les deux services, nous utilisons des images préconçues disponibles sur les servers Docker Elastic.

4) Envoi des données avec Spark et PySpark

Pour un problème de transfert de donnée depuis un stockage vers les conteneurs, le copier-coller classique prend beaucoup de temps et l'enjeux était de trouver une façon plus innovante de le faire. Comme solution, nous avons pensé que l'utilisation d'un framework assurant le parallélisme sera meilleure pour notre architecture. Nous avons utilisé plusieurs instances Spark pour distribuer la donnée dans des containers Docker. Un client PySpark tourne dans la machine hôte pour envoyer les données originales vers les containers. Spark permet aussi d'utiliser nos données d'une manière dynamique et avoir cette indépendance hôte / container.

Nous souhaitions avoir un mécanisme de découpage de notre dataset et, par la suite, l'enregistrement de nos données d'une façon persistante. A l'aide de l'architecture distribué de Spark, nous utilisons le « master » de cette dernière pour envoyer les données découpées aux worker. Grâce à la fonction « MAP », chaque worker va appeler une fonction commune mais avec des contextes différents.

```
def addToServer(image):
        from elasticsearch import Elasticsearch
        from minio import Minio
        from minio.error import ResponseError

        es = Elasticsearch(['http://elasticsearch:9200'])
        minioClient = Minio('minio:9000',access_key='minio',secret_key='minio123',secure=False)
        ret = ""
        try:
                t = time.time()
                buf = pickle.dumps(image[0])

                if not minioClient.bucket_exists('dat'):
                        minioClient.make_bucket('dat')

                minioClient.put_object('dat', str(t)+".npy", io.BytesIO(buf),len(buf))

                doc = {
                    'image': str(t)+".npy",
                    'label': str(image[1])
                }

                es.index(index="images_classification", doc_type='images',body=doc)

                ret = t
        except :

                ret = 0

        return ret
```

Cette fonction prend en paramètre un objet image contenant la donnée et le label. Le worker se connecte ensuite au client Minio qui lui est attribué et au client commun Elasticsearch pour déposer et indexer la donnée.

5) Apprentissage actif avec Keras

D'après nos recherches, le modèle de réseaux de neurones le plus adéquat pour le traitement des images est le CNN (Convolutional Neural Network). Ce dernier est plus performant pour la détection des objets dans les images à petites tailles. La structure basic du CNN est la suivante :

CONVOLUTION -> POOLING -> CONVOLUTION -> POOLING -> FULLY CONNECTED LAYER -> OUTPUT

La convolution est l'acte consistant à prendre les données d'origine et à en créer des mappages. Le Pooling est un sous-échantillonnage, dans lequel nous sélectionnons la valeur maximale qui devient la nouvelle valeur pour toute la région. Les couches entièrement connectées sont des réseaux de neurones

typiques, où tous les nœuds sont "entièrement connectés". Les couches convolutives ne sont pas entièrement connectées comme un réseau de neurones traditionnel. Tous cette structure est faisable d'une façon simple avec l'utilisation de Keras. Ce dernier nous fournit la possibilité de détailler la fonction de chaque couche.

Layer (type)	Output Shape	Param #
conv2d_2 (Conv2D)	(None, 30, 30, 256)	7168
activation_3 (Activation)	(None, 30, 30, 256)	0
max_pooling2d_2 (MaxPooling2	(None, 15, 15, 256)	0
conv2d_3 (Conv2D)	(None, 13, 13, 256)	590080
activation_4 (Activation)	(None, 13, 13, 256)	0
max_pooling2d_3 (MaxPooling2	(None, 6, 6, 256)	0
flatten_1 (Flatten)	(None, 9216)	0
dense_2 (Dense)	(None, 64)	589888
dense_3 (Dense)	(None, 5)	325
activation_5 (Activation)	(None, 5)	0

Total params: 1,187,461
Trainable params: 1,187,461
Non-trainable params: 0

Résultats

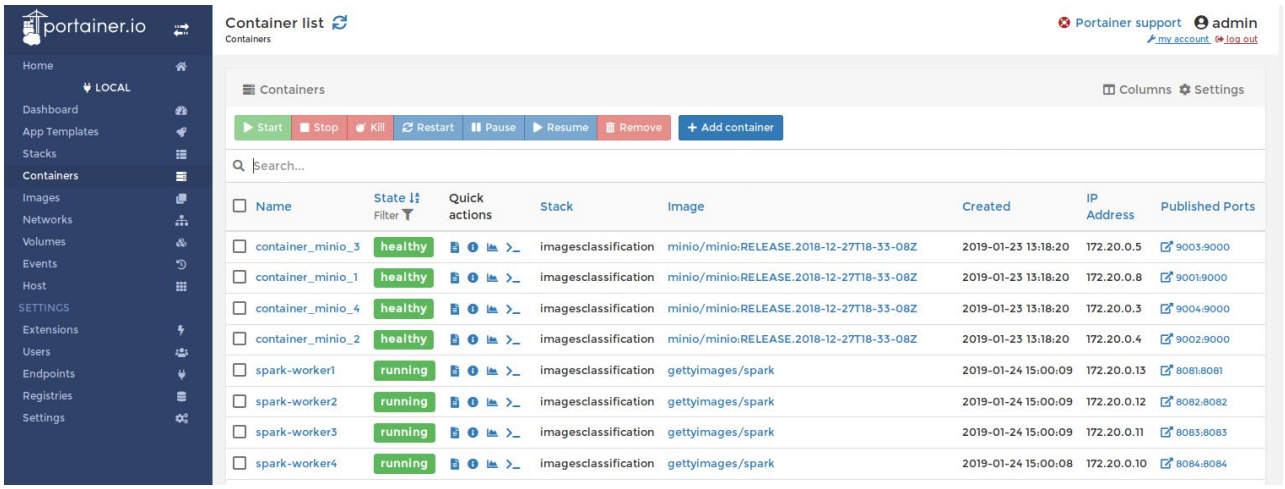
Cette partie du rapport montrera comment installer notre application et la mettre en fonctionnement avec les images déployées sur les serveurs et l'apprentissage actif lancé.

1) Initialisation de l'architecture

En se plaçant dans le répertoire de projet, il suffit de lancer la commande suivante pour générer l'architecture Docker sur la machine hôte :

docker-compose up -d

Nous pouvons consulter la bonne installation sur Portainer via l'adresse http://0.0.0.0:9000 :

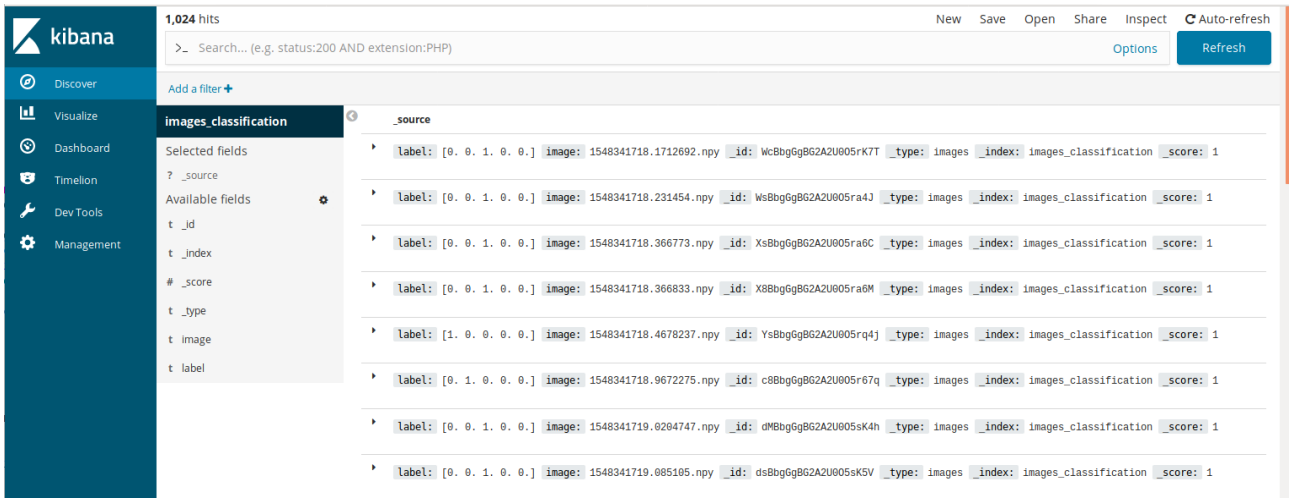


2) Ajout d'images sur Minio et indexation

En utilisant un client Spark, nous envoyons les images une à une d'une façon parallèle depuis le stockage vers le serveur. Les images sont alors indexées sur Elasticsearch une fois déposées sur les serveurs Minio. La commande à lancer depuis la machine hôte pour déposer les données est la suivante :

python3.5 split_data.py <RGB_image_location> <Label_image_location>

Nous pouvons ensuite consulter les données à partir de Kibana via l'adresse http://0.0.0.0:5601 :



3) Apprentissage actif

En utilisant à nouveau un client Spark et en se basant sur les liens fournis par Elasticsearch, nous pouvons importer les données depuis Minio. Nous pouvons enfin lancer l'apprentissage en utilisant la bibliothèque Keras qui s'exécute avec l'outil TensorFlow, via la commande suivante.

python3.5 learning.py <RGB_image_location> <Label_image_location>

Suite à cela un classifieur « classifier.h5 » est généré dans le répertoire source du projet. En utilisant le classifieur, la classe de l'image fournie est prédite. Cette dernière est ajoutée dans un conteneur Minio et son lien d'accès est inscrit dans Elasticsearch.

Concernant le score, nous avons pu obtenir à l'aide du modèle décrit précédemment un pourcentage de 92%, ce qui est selon nous une très bonne valeur atteinte.

```
Epoch 1/5
20000/20000 [==============================] - 250s 12ms/step - loss: 0.2975 - acc: 0.8725
Epoch 2/5
20000/20000 [==============================] - 219s 11ms/step - loss: 0.2402 - acc: 0.8986
Epoch 3/5
20000/20000 [==============================] - 208s 10ms/step - loss: 0.2188 - acc: 0.9094
Epoch 4/5
20000/20000 [==============================] - 208s 10ms/step - loss: 0.2043 - acc: 0.9159
Epoch 5/5
20000/20000 [==============================] - 209s 10ms/step - loss: 0.1968 - acc: 0.9185
500/500 [==============================] - 2s 4ms/step
Test loss: 0.18278470778465272
Test accuracy: 0.9223999819755554
```

Conclusion

Ce projet avait une dimension nouvelle par rapport aux autres projets que nous avons menés au cours de ce dernier semestre. Nous avons dû sur un même projet appliquer nos connaissances en architecture orientée services, virtualisation, analyse de données et apprentissage actif. Le fait d'aborder ces multiples problèmes fut très stimulant et très challengeant pour nous. Nous nous sommes également formés sur des technologies d'actualité utilisées en entreprise et nous avons découvert des outils performants qui nous resserviront pour sûr à l'avenir. L'ensemble du projet, incluant le code et la documentation, est disponible sur le dépôt ci-dessous :

---

https://github.com/elfilalime/images_classification

Nous livrons un projet prêt à l'emploi qui peut être installé et testé sur une autre machine. Les fondations que nous avons réalisées sont documentées et peuvent être facilement reprises pour un futur développement. Nous aurions aimé améliorer plusieurs détails sur notre projet, comme la division, l'envoi et la récupération de données qui pourraient être gérés automatiquement par un autre service plutôt que par la machine hôte. Confier l'exécution de l'apprentissage actif à Spark est également un souhait que nous aurions aimé réaliser et inclure dans le processus de l'application. Enfin, nous aurions pu imaginer une interface graphique dédiée à l'utilisation de notre application, pour l'envoi de données, le lancement de calculs et la consultation de résultats.